告知我们有关下载 PDF 体验的信息。

安装 Windows 调试器

项目 • 2024/03/05

WinDbg 是一个调试器，可用于分析故障转储、调试实时用户模式和内核模式代码，以及检查 CPU 寄存器和内存。

此最新版本具有更新的界面、完善的脚本功能、可扩展的调试数据模型、内置的时间旅行调试 (TTD) 支持和许多其他功能，提供更加现代化的用户体验。

有关详细信息，请参阅 WinDbg 概述。

下载 WinDbg

选择"安装"，调试器将下载并安装。

此外，WinDbg 还会定期在后台检查新版本，并在必要时自动更新。

> **备注**
>
> 此版本以前在 Microsoft Store 中作为 *WinDbg 预览版*发布，利用与 *WinDbg (classic)* 相同的基础引擎，并支持所有相同的命令、扩展和工作流。
>
> 要获取并继续使用最新版本，请按本页所述安装 WinDbg。 *WinDbg 预览版*不会在 Microsoft Store 中获得进一步更新。

要求

- 支持的操作系统：
  - Windows 11（所有版本）
  - Windows 10 周年更新（版本 1607）或更新版本
- 处理器体系结构：
  - x64 和 ARM64

疑难解答

如果在安装或使 WinDbg 保持更新时遇到问题，请参阅使用应用安装程序文件的解决安装问题。

如果发现任何错误或要提出功能请求，可以通过功能区中的反馈按钮转到 GitHub 页，可以在其中提出新问题。

WinDbg 入门

要开始使用 WinDbg，请参阅开始使用 Windows 调试。

若要开始调试内核模式驱动程序，请参阅调试通用驱动程序 -“逐步操作”实验室（Echo 内核模式）。 这是一个“逐步操作”实验室，演示了如何使用 WinDbg 调试 Echo（一个使用内核模式驱动程序框架 (KMDF) 的示例驱动程序）。

以前的版本和相关下载

要调试旧版本的 Windows，请使用 Windows 调试工具中的 WinDbg (Classic)。

- 下载 Windows 驱动程序工具包 (WDK)
- Windows 符号程序包
- Windows Hardware Lab Kit
- 下载并安装 Windows 评估和部署工具包 (Windows ADK)
- Windows 预览体验成员 - Windows 预览版本

Windows 调试工具

项目 • 2024/03/05

除了 WinDbg 等调试器之外，Windows 调试工具还包括一组可用于调试的工具。 有关这些工具的完整列表，请参阅 Windows 调试工具中包含的工具。

有关如何仅下载和安装 Windows 调试器的说明，请参阅下载并安装 WinDbg Windows 调试器。

安装 Windows 调试工具

Windows 调试工具既可以作为开发工具包的一部分获取，也可以作为独立的工具集获取：

- **作为 WDK 的一部分**

  Windows 调试工具包含在 Windows 驱动程序工具包 (WDK) 中。 若要获取 WDK，请参阅下载 Windows 驱动程序工具包 (WDK)。

- **作为 Windows SDK 的一部分**

  Windows 调试工具包含在 Windows 软件开发工具包 (SDK) 中。 若要下载安装程序或 ISO 映像，请参阅 Windows 开发人员中心上的 Windows SDK。

- **作为独立工具集**

  你可以在不安装 Windows SDK 或 WDK 的情况下单独安装 Windows 调试工具，方法是启动 Windows SDK 的安装，然后在要安装的功能列表中仅选择 **Windows 调试工具**（并清除所有其他功能的选择）。 若要下载安装程序或 ISO 映像，请参阅 Windows 开发人员中心上的 Windows SDK。

调试环境

如果计算机安装了 Visual Studio 和 WDK，就会有六个可用的调试环境。 有关这些环境的说明，请参阅调试环境。

所有这些调试环境都提供适用于同一基础调试引擎（在 Windows 符号调试程序引擎 (Dbgeng.dll) 中实现）的用户界面。 该调试引擎也称为“Windows 调试程序”，这六个调试环境统称为“Windows 调试程序”。

> **备注**

Visual Studio 包含自己的调试环境和调试引擎，它们统称为“Visual Studio 调试程序”。 若要了解如何在 Visual Studio 中进行调试，请参阅**在 Visual Studio 中调试**。 对于调试托管代码（例如 C#）而言，使用 Visual Studio 调试程序通常是最容易的入门方法。

Windows 调试程序

Windows 调试程序可以在基于 x86、x64 或 Arm 的处理器上运行，并且可以调试在那些相同体系结构上运行的代码。 有时候，调试程序和要调试的代码运行在同一计算机上，但另外一些时候，调试程序和要调试的代码则运行在不同的计算机上。 不管哪一种情况，运行调试程序的计算机均称为“主计算机”，被调试的计算机均称为“目标计算机”。 不管是主机计算机还是目标计算机，Windows 调试程序都支持以下 Windows 版本。

命令行调试器

有四种命令行调试器可用于专门的环境以及首选命令行接口的环境。

KD 和 NTKD

KD 和 NTKD 在各方面都完全相同，只是 NTKD 在启动时会生成一个新的文本窗口，而 KD 则继承了调用时的命令提示符窗口。 有关详细信息，请参阅使用 KD 和 NTKD 进行调试。

CDB 和 NTSD

此外，还有 Microsoft 控制台调试器 (CDB) 和 Microsoft NT 符号调试器 (NTSD)。 有关详细信息，请参阅使用 CDB 和 NTSD 进行调试。

符号和符号文件

符号文件存储了在运行可执行二进制文件时不需要的各种数据，但在调试代码时，符号文件非常有用。 有关创建和使用符号文件的详细信息，请参阅用于 Windows 调试的符号。

蓝屏和崩溃转储文件

如果 Windows 停止工作并显示蓝屏，计算机会突然关闭以防止数据丢失，并显示错误检查代码。 有关详细信息，请参阅错误检查（蓝屏）。 可以使用 WinDbg 和其他

Windows 调试器分析 Windows 关闭时创建的故障转储文件。 有关详细信息，请参阅使用 Windows 调试程序 (WinDbg) 进行故障转储分析。

要查找早期版 Windows 的调试工具?

若要下载旧版 Windows 的调试程序工具，需要从 Windows SDK 和模拟器存档下载适用于所要调试的版本的 Windows SDK。 在 SDK 的安装向导中选择“Windows 调试工具”，并取消选择所有其他组件。

详细了解调试程序

在下载并安装 WinDbg Windows 调试器中了解有关 WinDbg 的详细信息。

若要开始使用 Windows 调试，请参阅 Windows 调试入门。

如需与 Windows 调试工具相关的其他信息，请参阅调试资源。

查找其他下载?

- 下载 Windows 驱动程序工具包 (WDK)
- Windows 符号程序包
- Windows Hardware Lab Kit
- 下载并安装 Windows 评估和部署工具包 (Windows ADK)
- Windows 预览体验成员 - Windows 预览版本

开始使用 Windows 调试

项目 • 2024/03/05

本文介绍如何开始使用 Windows 调试。 如果你的目标是使用调试器分析故障转储，请参阅使用 WinDbg 分析故障转储文件。

要开始使用 Windows 调试，请完成以下步骤。

1. 标识主系统和目标系统

调试时通常使用两个独立的计算机系统，因为在调试过程中，处理器上的指令执行通常会暂停。 调试器在“主机”系统上运行，要调试的代码在“目标”系统上运行。

主系统 <----------> **目标系统**

在某些情况下，可以使用虚拟机作为第二个系统。 例如，虚拟电脑可以与需要调试的代码运行在同一台电脑上。 但是，如果代码与低级别硬件通信，使用虚拟电脑可能不是最佳方法。 有关详细信息，请参阅设置虚拟机的网络调试 - KDNET.

2. 确定调试器类型：内核模式或用户模式

接下来，需要确定是使用内核模式调试还是用户模式调试。

- 操作系统和特权程序在*内核模式*下运行。 内核模式代码有权访问系统的任何部分，并且不像用户模式代码那样受到限制。 内核模式代码可以访问在用户模式或内核模式下运行的任何其他进程的任何部分。 许多核心操作系统功能和许多硬件设备驱动程序在内核模式下运行。

- 计算机上的应用程序和子系统在*用户模式*下运行。 在用户模式下运行的进程在其自己的虚拟地址空间中运行。 它们被限制直接访问系统的许多部分，包括系统硬件、未分配供其使用的内存以及可能危及系统完整性的系统其他部分。 在用户模式下运行的进程与系统和其他用户模式进程有效隔离，因此它们不会干扰这些资源。

如果你的目标是调试驱动程序，请确定该驱动程序是内核模式驱动程序还是用户模式驱动程序。 Windows 驱动程序模型 (WDM) 驱动程序与内核模式驱动程序框架 (KMDF) 都是

内核模式驱动程序。 顾名思义，用户模式驱动程序框架 (UMDF) 驱动程序是用户模式驱动程序。

由于某些问题，可能很难确定代码以何种模式执行。 在这种情况下，可能需要选择一种模式，查看在该模式下可以获得哪些信息。 有些问题需要在用户模式和内核模式下使用调试器。

根据调试模式的不同，可能需要以不同的方式配置和使用调试器。 有些调试命令在两种模式下的操作相同，有些命令的操作则不同。

了解有关在内核模式下使用调试器的详细信息：

- 开始使用 WinDbg（内核模式）
- 调试通用驱动程序：分步实验室（回显内核模式）
- 调试驱动程序 ： 分步实验室（Sysvad 内核模式）

了解有关在用户模式下使用调试器的详细信息：

- 开始使用 WinDbg（用户模式）

3. 选择调试器环境

WinDbg 调试器在大多数情况下都运行良好，但有时可能需要使用其他调试器，例如用于自动化或 Visual Studio 的控制台调试器。 有关详细信息，请参阅调试环境。

4. 确定如何连接目标和主机

通常，目标和主机系统通过以太网连接。 如果正在进行早期调试工作，或者设备上没有以太网连接，还可以使用其他网络连接选项。 有关详细信息，请参阅以下文章：

- 自动设置 KDNET 网络内核调试
- 设置虚拟机的网络调试 - KDNET

5. 选择 32 位或 64 位调试工具

是需要 32 位还是 64 位调试器取决于在目标系统和主机系统上运行的 Windows 版本，以及调试的是 32 位还是 64 位代码。 有关详细信息，请参阅选择 32 位或 64 位调试工具。

6. 配置符号

若要使用 WinDbg 提供的所有高级功能，必须加载正确的符号。 如果没有正确配置符号，则在尝试使用依赖于符号的功能时，将收到指示符号不可用的消息。 有关详细信

息，请参阅用于 Windows 调试的符号。

7. 配置源代码

如果目标是调试自己的源代码，则需要配置指向源代码的路径。 有关详细信息，请参阅源路径。

8. 熟悉调试器操作

本文档的调试器操作部分描述了各种任务的调试器操作。 例如，加载调试器扩展 DLL 说明了如何加载调试器扩展。

9. 熟悉调试方法

标准调试方法适用于大多数调试方案，例如，设置断点、检查调用堆栈和查找内存泄漏。 专业调试方法适用于特定的技术或代码类型。 例如，即插即用调试、KMDF调 试和 RPC 调试。

10. 使用调试器引用命令

在调试器中工作时，可以使用不同的调试命令。 在调试器中使用 .hh 命令可显示有关任何调试命令的有用信息。 有关可用命令的详细信息，请参阅调试器参考。

11. 将调试扩展用于特定技术

有多个调试扩展可用于分析特定于域的数据结构。 有关详细信息，请参阅专用扩展。

12. 了解相关的 Windows 内部结构

本文档假定你已了解一些 Windows 核心内部知识。 要了解有关 Windows 内部的详细信息，包括内存使用情况、上下文、线程和进程，可以查看 Pavel Yosifovich、Mark E. Russinovich、David A. Solomon 和 Alex Ionescu 编写的 Windows Internals 等资源。

13. 查看其他调试资源

其他资源包括以下书籍和视频：

- 《在 Windows 调试中：实用的调试和跟踪策略》，作者：Tarik Soulami

- 《高级 Windows 调试》，作者：Mario Hewardt 和 Daniel Pravat
- *碎片整理工具*视频系列，第 13 集至第 29 集，全部与 WinDbg 有关

另请参阅

- 开始使用 WinDbg（内核模式）
- 开始使用 WinDbg（用户模式）
- 选择 32 位或 64 位调试工具
- 调试环境
- 设置调试（内核模式和用户模式）
- 调试通用驱动程序 - 分步实验室（回显内核模式）
- 调试驱动程序 - 分步实验室（Sysvad 内核模式）

开始使用 WinDbg（用户模式）

项目 • 2024/03/05

WinDbg 是 Windows 调试工具中包含的内核模式和用户模式调试器。 以下动手练习可帮助开始使用 WinDbg 作为用户模式调试器。

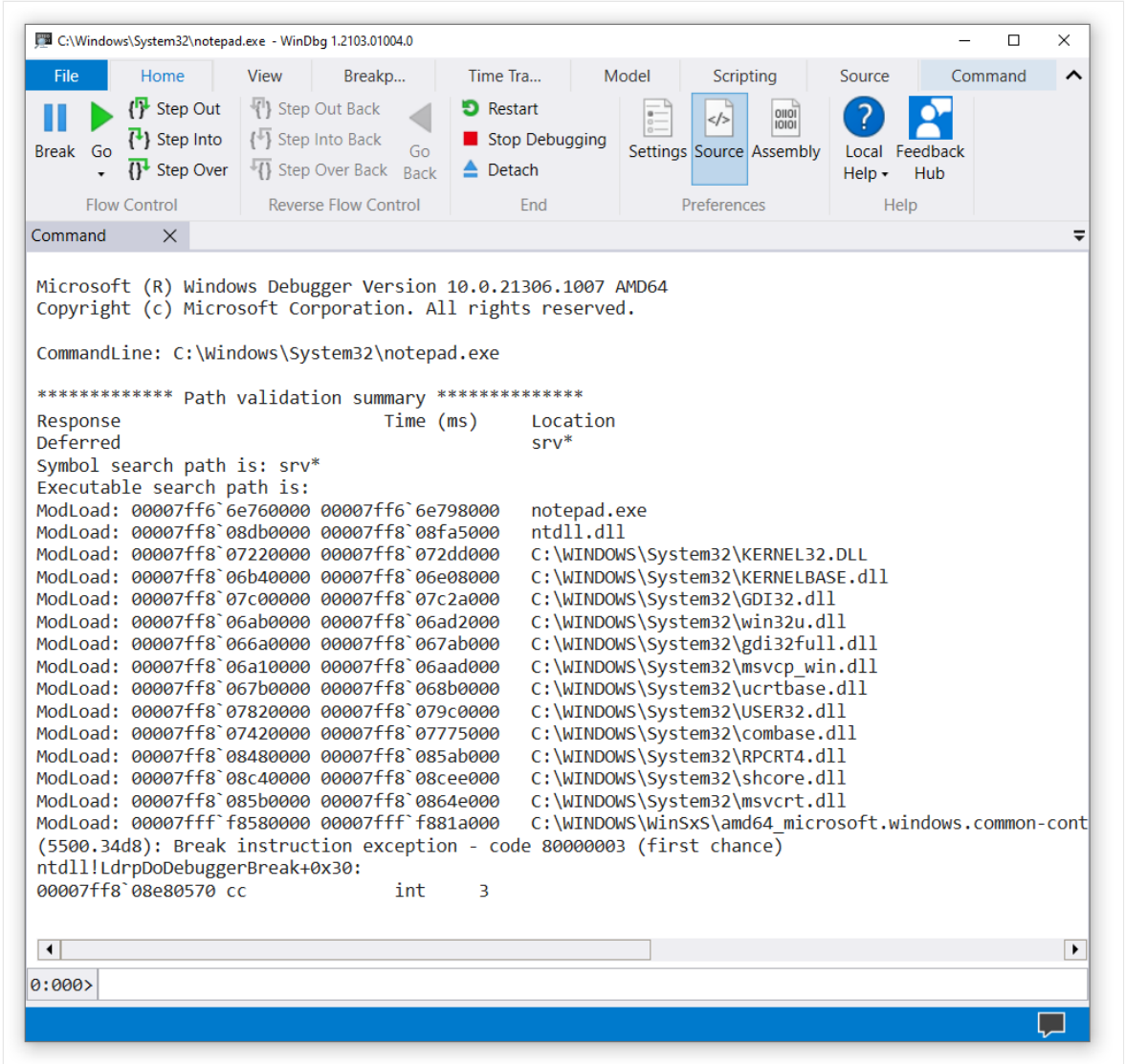
有关如何获取 Windows 调试工具的信息，请参阅下载和安装 Windows 调试器。

安装调试工具后，找到 64 位 (x64) 和 32 位 (x86) 版本工具的安装目录。 例如：

- **C:\Program Files (x86)\Windows Kits\10\Debuggers\x64**
- **C:\Program Files (x86)\Windows Kits\10\Debuggers\x86**

打开记事本并附加 WinDbg

1. 转到安装目录，然后打开 **WinDbg.exe**。
2. 在“文件”菜单上，选择“启动可执行文件”。 在“启动可执行文件”对话框中，转到包含 notepad.exe 的文件夹。 （notepad.exe 文件通常位于 C:\Windows\System32 中。 对于“文件名”，请输入 **notepad.exe**。 选择**打开**。



3. 在 WinDbg 窗口底部附近的命令行中输入以下命令：

   .sympath srv*

   输出类似于以下示例：

   ```
   dbgcmd

   Symbol search path is: srv*
   Expanded Symbol search path is: cache*;SRV
   ```

   符号搜索路径指示 WinDbg 查找符号 (PDB) 文件的位置。 调试器需要符号文件来获取有关代码模块的信息，例如函数名称和变量名称。

   然后，输入此命令：

   .reload

   `.reload` 命令指示 WinDbg 执行初始搜索以查找和加载符号文件。

4. 要查看 notepad.exe 模块的符号，请输入以下命令：

x notepad!*

> ⓘ **备注**
>
> 如果未显示任何输出，请输入 `.reload /f` 以尝试强制加载符号。 使用 !sym noisy 可显示更多符号加载信息。

要查看 notepad.exe 模块中包含 `main` 的符号，请使用 examine symbols 命令列出与掩码匹配的模块：

`x notepad!wWin*`

输出类似于以下示例：

```
dbgcmd

00007ff6`6e76b0a0 notepad!wWinMain (wWinMain)
00007ff6`6e783db0 notepad!wWinMainCRTStartup (wWinMainCRTStartup)
```

5. 要在 `notepad!wWinMain` 处设置断点，请输入此命令：

   bu notepad!wWinMain

   若要验证是否已设置断点，请输入以下命令：

   bl

   输出类似于以下示例：

```
dbgcmd

0 e Disable Clear  00007ff6`6e76b0a0     0001 (0001)  0:****
notepad!wWinMain
```

6. 要启动记事本进程，请输入以下命令：

   g

   记事本一直运行到 `WinMain` 函数，然后它中断到调试器。

```
dbgcmd

Breakpoint 0 hit
notepad!wWinMain:
00007ff6`6e76b0a0 488bc4          mov     rax,rsp
```

要查看当前在记事本进程中加载的代码模块的列表，请输入以下命令：

lm

输出类似于以下示例：

dbgcmd

```
0:000> lm
start             end                 module name
00007ff6`6e760000 00007ff6`6e798000   notepad    (pdb symbols)
C:\ProgramData\Dbg\sym\notepad.pdb\BC04D9A431EDE299D4625AD6201C8A4A1\no
tepad.pdb
00007ff8`066a0000 00007ff8`067ab000   gdi32full   (deferred)
00007ff8`067b0000 00007ff8`068b0000   ucrtbase   (deferred)
00007ff8`06a10000 00007ff8`06aad000   msvcp_win   (deferred)
00007ff8`06ab0000 00007ff8`06ad2000   win32u     (deferred)
00007ff8`06b40000 00007ff8`06e08000   KERNELBASE   (deferred)
00007ff8`07220000 00007ff8`072dd000   KERNEL32   (deferred)
00007ff8`07420000 00007ff8`07775000   combase    (deferred)
00007ff8`07820000 00007ff8`079c0000   USER32     (deferred)
00007ff8`079c0000 00007ff8`079f0000   IMM32      (deferred)
00007ff8`07c00000 00007ff8`07c2a000   GDI32      (deferred)
00007ff8`08480000 00007ff8`085ab000   RPCRT4     (deferred)
00007ff8`085b0000 00007ff8`0864e000   msvcrt     (deferred)
00007ff8`08c40000 00007ff8`08cee000   shcore     (deferred)
00007ff8`08db0000 00007ff8`08fa5000   ntdll      (pdb symbols)
C:\ProgramData\Dbg\sym\ntdll.pdb\53F12BFE149A2F50205C8D5D66290B481\ntdl
l.pdb
00007fff`f8580000 00007fff`f881a000   COMCTL32   (deferred)
```

若要查看堆栈跟踪，请输入以下命令：

k

输出类似于以下示例：

dbgcmd

```
0:000> k
00 000000c8`2647f708 00007ff6`6e783d36     notepad!wWinMain
01 000000c8`2647f710 00007ff8`07237034
notepad!__scrt_common_main_seh+0x106
02 000000c8`2647f750 00007ff8`08e02651
KERNEL32!BaseThreadInitThunk+0x14
03 000000c8`2647f780 00000000`00000000
ntdll!RtlUserThreadStart+0x21
```

7. 若要再次启动记事本运行，请输入以下命令：

g

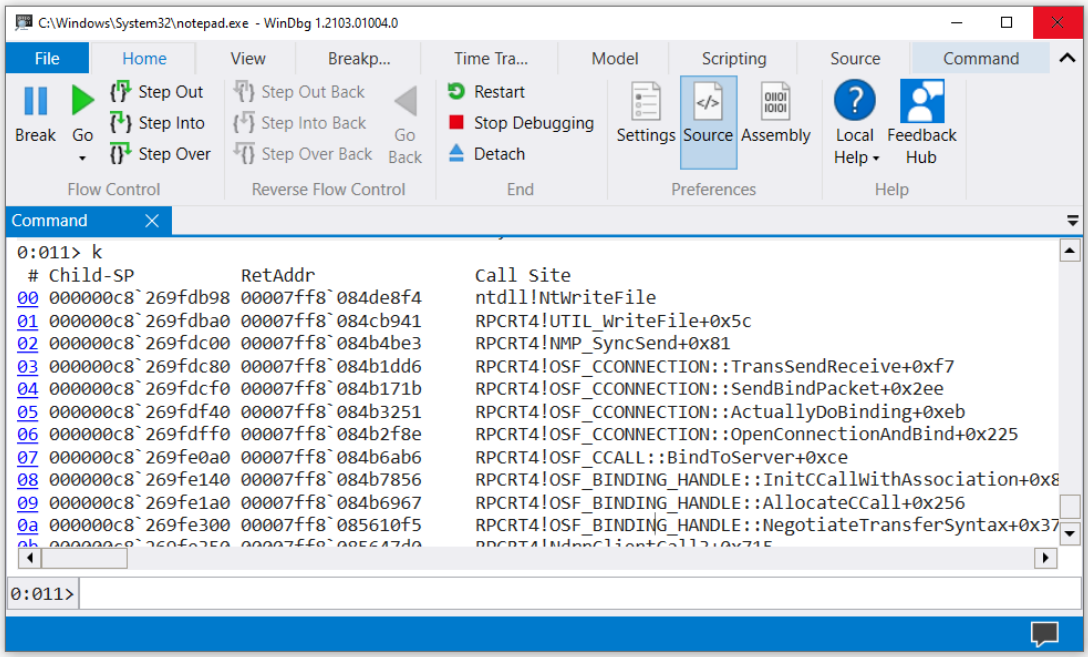
8. 要中断记事本，请在“文件”菜单上选择“中断”。

9. 要在 `ZwWriteFile` 处设置和验证断点，请输入以下命令：

   bu ntdll!ZwWriteFile

   bl

10. 要再次启动记事本运行，请输入 g。 在记事本窗口中，输入一些文本。 在“文件”菜单中，选择“保存”。 运行代码在 `ZwCreateFile` 处中断。 输入 k 命令以查看堆栈跟踪。



   在 WinDbg 窗口中，命令行左侧显示处理器和线程编号。 在本例中，当前处理器编号为 0，当前线程编号为 11 (`0:011>`)。 该窗口显示在 0 号处理器上运行的线程 11 的堆栈跟踪。

11. 若要查看记事本进程中所有线程的列表，请输入以下命令（波形符）：

    ~

    输出类似于以下示例：

```
dbgcmd
```

```
0:011> ~
   0  Id: 5500.34d8 Suspend: 1 Teb: 000000c8`262c4000 Unfrozen
   1  Id: 5500.3960 Suspend: 1 Teb: 000000c8`262c6000 Unfrozen
    2  Id: 5500.5d68 Suspend: 1 Teb: 000000c8`262c8000 Unfrozen
    3  Id: 5500.4c90 Suspend: 1 Teb: 000000c8`262ca000 Unfrozen
```

```
    4  Id: 5500.4ac4 Suspend: 1 Teb: 000000c8`262cc000 Unfrozen
    5  Id: 5500.293c Suspend: 1 Teb: 000000c8`262ce000 Unfrozen
    6  Id: 5500.53a0 Suspend: 1 Teb: 000000c8`262d0000 Unfrozen
    7  Id: 5500.3ca4 Suspend: 1 Teb: 000000c8`262d4000 Unfrozen
    8  Id: 5500.808 Suspend: 1 Teb: 000000c8`262da000 Unfrozen
   10  Id: 5500.3940 Suspend: 1 Teb: 000000c8`262dc000 Unfrozen
. 11  Id: 5500.28b0 Suspend: 1 Teb: 000000c8`262de000 Unfrozen
   12  Id: 5500.12bc Suspend: 1 Teb: 000000c8`262e0000 Unfrozen
   13  Id: 5500.4c34 Suspend: 1 Teb: 000000c8`262e2000 Unfrozen
```

在此示例中，14 个线程的索引为 0 到 13。

12. 若要查看线程 0 的堆栈跟踪，请输入以下命令：

    ~0s

    k

    输出类似于以下示例：

    dbgcmd

```
0:011> ~0s
0:011> ~0s
win32u!NtUserGetProp+0x14:
00007ff8`06ab1204 c3              ret
0:000> k
 # Child-SP          RetAddr               Call Site
00 000000c8`2647bd08 00007ff8`07829fe1     win32u!NtUserGetProp+0x14
01 000000c8`2647bd10 00007fff`f86099be     USER32!GetPropW+0xd1
02 000000c8`2647bd40 00007ff8`07d12f4d
COMCTL32!DefSubclassProc+0x4e
03 000000c8`2647bd90 00007fff`f8609aba
SHELL32!CAutoComplete::_EditWndProc+0xb1
04 000000c8`2647bde0 00007fff`f86098b7
COMCTL32!CallNextSubclassProc+0x9a
05 000000c8`2647be60 00007ff8`0782e858
COMCTL32!MasterSubclassProc+0xa7
06 000000c8`2647bf00 00007ff8`0782de1b
USER32!UserCallWinProcCheckWow+0x2f8
07 000000c8`2647c090 00007ff8`0782d68a
USER32!SendMessageWorker+0x70b
08 000000c8`2647c130 00007ff8`07afa4db     USER32!SendMessageW+0xda
```

13. 若要退出调试并从记事本进程分离，请输入以下命令：

    qd

打开自己的应用程序并附加 WinDbg

例如，假设已编写并构建了这个小型控制台应用程序：

C++

```
...
void MyFunction(long p1, long p2, long p3)
{
    long x = p1 + p2 + p3;
    long y = 0;
    y = x / p2;
}

void main ()
{
    long a = 2;
    long b = 0;
    MyFunction(a, b, 5);
}
```

在本练习中，假设构建的应用程序 (MyApp.exe) 和符号文件 (MyApp.pdb) 位于 C:\MyApp\x64\Debug 中。 此外，假设应用程序源代码位于 C:\MyApp\MyApp 中，并且目标计算机已编译了 MyApp.exe。

1. 打开 WinDbg。
2. 在“文件”菜单上，选择“启动可执行文件”。 在“启动可执行文件”对话框中，转到 C:\MyApp\x64\Debug。 对于“文件名称”，输入“MyApp.exe”。 选择**打开**。
3. 输入以下命令：

   .symfix

   .sympath+ C:\MyApp\x64\Debug

   这些命令告诉 WinDbg 在何处查找应用程序的符号和源代码。 在这种情况下，不需要使用 .srcpath 设置源代码位置，因为符号具有指向源文件的完全限定路径。
4. 输入以下命令：

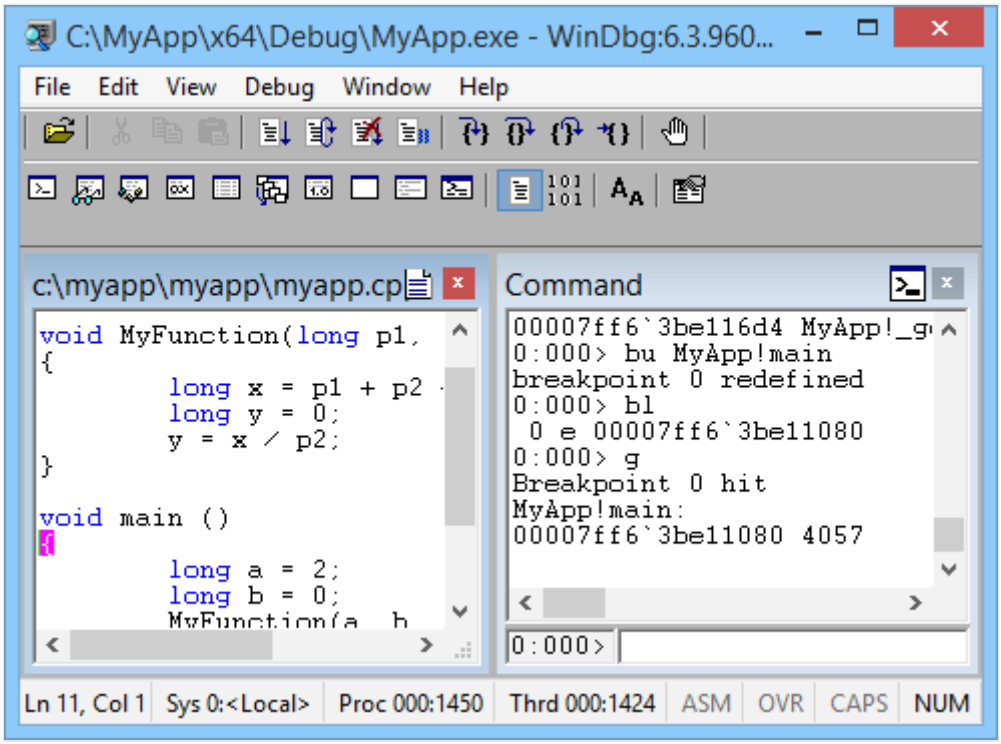
   .reload

   bu MyApp!main

   g

   应用程序在到达 `main` 功能时会中断到调试器。

   WinDbg 显示源代码和命令窗口。



5. 在“调试”菜单上，选择“单步执行”（或选择 F11）。 继续单步执行，直到进入 `MyFunction`。 在单步执行到行 `y = x / p2` 时，应用程序崩溃并中断到调试器。

   输出类似于以下示例：

   dbgcmd

   ```
   (1450.1424): Integer divide-by-zero - code c0000094 (first chance)
   First chance exceptions are reported before any exception handling.
   This exception may be expected and handled.
   MyApp!MyFunction+0x44:
   00007ff6`3be11064 f77c2428    idiv  eax,dword ptr [rsp+28h]
   ss:00000063`2036f808=00000000
   ```

6. 输入以下命令：

   !analyze -v

   WinDbg 显示对问题的分析（在本例中为除以 0）。

   dbgcmd

   ```
   FAULTING_IP:
   MyApp!MyFunction+44 [c:\myapp\myapp\myapp.cpp @ 7]
   00007ff6`3be11064 f77c2428        idiv    eax,dword ptr [rsp+28h]

   EXCEPTION_RECORD:  ffffffffffffffff -- (.exr 0xffffffffffffffff)
   ExceptionAddress: 00007ff63be11064
   (MyApp!MyFunction+0x0000000000000044)
      ExceptionCode: c0000094 (Integer divide-by-zero)
     ExceptionFlags: 00000000
   ```

```
NumberParameters: 0
...
STACK_TEXT:
00000063`2036f7e0 00007ff6`3be110b8 : ... : MyApp!MyFunction+0x44
00000063`2036f800 00007ff6`3be1141d : ... : MyApp!main+0x38
00000063`2036f840 00007ff6`3be1154e : ... :
MyApp!__tmainCRTStartup+0x19d
00000063`2036f8b0 00007ffc`b1cf16ad : ... : MyApp!mainCRTStartup+0xe
00000063`2036f8e0 00007ffc`b1fc4629 : ... :
KERNEL32!BaseThreadInitThunk+0xd
00000063`2036f910 00000000`00000000 : ... :
ntdll!RtlUserThreadStart+0x1d

STACK_COMMAND: dt ntdll!LdrpLastDllInitializer BaseDllName ;dt
ntdll!LdrpFailureData ;.cxr 0x0 ;kb

FOLLOWUP_IP:
MyApp!MyFunction+44 [c:\myapp\myapp\myapp.cpp @ 7]
00007ff6`3be11064 f77c2428        idiv    eax,dword ptr [rsp+28h]

FAULTING_SOURCE_LINE:  c:\myapp\myapp\myapp.cpp

FAULTING_SOURCE_FILE:  c:\myapp\myapp\myapp.cpp

FAULTING_SOURCE_LINE_NUMBER:  7

FAULTING_SOURCE_CODE:
     3: void MyFunction(long p1, long p2, long p3)
     4: {
     5:     long x = p1 + p2 + p3;
     6:     long y = 0;
>    7:  y = x / p2;
     8: }
     9:
    10: void main ()
    11: {
    12:     long a = 2;
...
```

命令摘要

- `Help` 菜单上的命令 `Contents`
- .sympath（设置符号路径）
- .reload（重新加载模块）
- x（检查符号）
- g（转到）
- `Debug` 菜单上的命令 `Break`
- lm（列出已加载的模块）
- k（显示堆栈回溯）

- bu（设置断点）
- bl（断点列表）
- ~（线程状态）
- ~s（设置当前线程）
- .sympath+（设置符号路径）附加到现有符号路径
- srcpath（设置源路径）
- `Debug` 菜单上的命令 `Step Into` (F11)
- !analyze -v
- qd（退出和分离）

另请参阅

开始使用 WinDbg（内核模式）

调试器操作

调试技术

下载并安装 WinDbg Windows 调试器

WinDbg 功能

开始使用 WinDbg（内核模式）

项目 • 2024/08/26

WinDbg 是 Windows 调试工具中包含的内核模式和用户模式调试器。 本文提供的练习帮助开始使用 WinDbg 作为内核模式调试器。

有关如何获取 Windows 调试工具的信息，请参阅下载和安装 Windows 调试器。 安装调试工具后，找到 64 位 (x64) 和 32 位 (x86) 版本工具的安装目录。 例如：

- **C:\Program Files (x86)\Windows Kits\10\Debuggers\x64**
- **C:\Program Files (x86)\Windows Kits\10\Debuggers\x86**

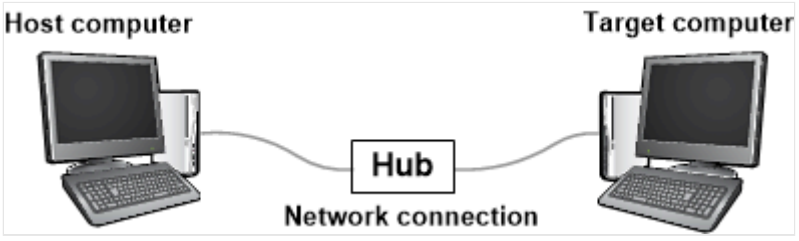
设置内核模式调试

内核模式调试环境通常有两台计算机，即*主计算机*和目标计算机。 调试程序在主计算机上运行，所调试的代码在目标计算机上运行。 主计算机和目标计算机由调试电缆连接。

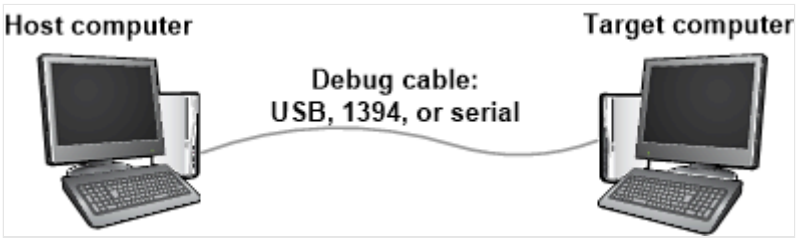
Windows 调试器支持以下类型的电缆：

- 以太网
- USB 2.0 / USB 3.0
- 串行（也称为零调制解调器）

为了获得速度和可靠性，应将以太网电缆与本地网络中心配合使用。 下图演示了使用以太网电缆连接主计算机和目标计算机进行调试的情况。



对于旧版的 Windows，可选择使用直接电缆，例如 USB 或串行电缆。



有关如何设置主计算机机和目标计算机的详细信息，请参阅手动设置内核模式调试。

虚拟机 - (VM)

有关将调试器连接到 Hyper-V 虚拟机的信息，请参阅设置虚拟机的网络调试 - KDNET。

建立内核模式调试会话

设置好主计算机和目标计算机并用调试电缆连接它们后，可以建立内核模式调试会话。 请按照设置时使用的同一主题中的说明进行操作。 例如，如果决定将主计算机和目标计算机设置为通过以太网进行调试，可以在下面的文章中找到建立内核模式调试会话的说明：

- 自动设置 KDNET 网络内核调试

开始使用 WinDbg

1. 在主计算机上打开 WinDbg，与目标计算机建立内核模式调试会话。
2. 要打开调试器文档 CHM 文件，请转到“帮助”菜单并选择“目录”。 调试器文档也可在 Windows 调试工具在线获取。
3. 建立内核模式调试会话时，WinDbg 可能会自动中断目标计算机。 如果 WinDbg 没有中断，请转到“调试”菜单，然后选择“中断”。
4. 在 WinDbg 窗口底部的命令行中，输入以下命令：

   .sympath srv*

   输出类似于以下示例：

   ```
   dbgcmd

   Symbol search path is: srv*
   Expanded Symbol search path is:
   cache*;SRV*https://msdl.microsoft.com/download/symbols
   ```

   符号搜索路径指示 WinDbg 查找符号 (PDB) 文件的位置。 调试器需要符号文件来获取有关代码模块的信息，例如函数名称和变量名称。

   输入以下命令，指示 WinDbg 执行其初始查找和加载符号文件：

   .reload

5. 要获取已部署模块的列表，请输入以下命令：

   lm

   输出类似于以下示例：

dbgcmd

```
0:000>3: kd> lm
start             end                 module name
fffff800`00000000 fffff800`00088000   CI         (deferred)
...
fffff800`01143000 fffff800`01151000   BasicRender   (deferred)
fffff800`01151000 fffff800`01163000   BasicDisplay  (deferred)
...
fffff800`02a0e000 fffff800`03191000   nt  (pdb symbols)
C:\...\ntkrnlmp.pdb
fffff800`03191000 fffff800`03200000   hal (deferred)
...
```

6. 要启动目标计算机运行，请输入以下命令：

   g

7. 要再次中断，请转到“调试”菜单，然后选择“中断”。

8. 输入以下命令以检查 `_FILE_OBJECT` 模块中的 `nt` 数据类型：

   dt nt!_FILE_OBJECT

   输出类似于以下示例：

dbgcmd

```
0:000>0: kd> dt nt!_FILE_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x008 DeviceObject     : Ptr64 _DEVICE_OBJECT
   +0x010 Vpb              : Ptr64 _VPB
   ...
   +0x0c0 IrpList          : _LIST_ENTRY
   +0x0d0 FileObjectExtension : Ptr64 Void
```

9. 输入以下命令以检查 `nt` 模块中的某些符号：

   x nt!*CreateProcess*

   输出类似于以下示例：

dbgcmd

```
0:000>0: kd> x nt!*CreateProcess*
fffff800`030821cc nt!ViCreateProcessCallbackInternal (<no parameter
info>)
...
fffff800`02e03904 nt!MmCreateProcessAddressSpace (<no parameter info>)
```

```
fffff800`02cece00 nt!PspCreateProcessNotifyRoutine = <no type
information>
...
```

10. 输入以下命令，将断点置于 **MmCreateProcessAddressSpace**：

    bu nt!MmCreateProcessAddressSpace

    要验证是否已设置断点，请输入以下命令：

    bl

    输出类似于以下示例：

    dbgcmd

    ```
    0:000>0: kd> bu nt!MmCreateProcessAddressSpace
    0: kd> bl
    0 e fffff800`02e03904     0001 (0001) nt!MmCreateProcessAddressSpace
    ```

    输入 g 让目标计算机运行。

11. 如果目标计算机没有立即中断调试器，请在目标计算机上执行一些操作（例如，打开记事本）。 调用 **MmCreateProcessAddressSpace** 时，目标计算机将中断调试器。 要查看堆栈跟踪，请输入以下命令：

    .reload

    k

    输出类似于以下示例：

    dbgcmd

    ```
    0:000>2: kd> k
    Child-SP          RetAddr           Call Site
    ffffd000`224b4c88 fffff800`02d96834 nt!MmCreateProcessAddressSpace
    ffffd000`224b4c90 fffff800`02dfef17 nt!PspAllocateProcess+0x5d4
    ffffd000`224b5060 fffff800`02b698b3 nt!NtCreateUserProcess+0x55b
    ...
    000000d7`4167fbb0 00007ffd`14b064ad KERNEL32!BaseThreadInitThunk+0xd
    000000d7`4167fbe0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
    ```

12. 在“视图”菜单上，选择“反汇编”。

    在“调试”菜单上，选择“单步跳过”（或按 F10）。 在观察反汇编窗口时，多次输入步骤命令。

13. 输入以下命令清除断点：

    bc *

    输入 g 让目标计算机运行。 要再次中断，请转到“调试”菜单并选择“中断”，或按 Ctrl-Break。

14. 要查看所有进程的列表，请输入以下命令：

    !process 0 0

    输出类似于以下示例：

    dbgcmd

```
0:000>0: kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS ffffe000002287c0
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 001aa000  ObjectTable: ffffc00000003000  HandleCount:
<Data Not Accessible>
    Image: System

PROCESS ffffe00001e5a900
    SessionId: none  Cid: 0124    Peb: 7ff7809df000  ParentCid: 0004
    DirBase: 100595000  ObjectTable: ffffc000002c5680  HandleCount:
<Data Not Accessible>
    Image: smss.exe
...
PROCESS ffffe00000d52900
    SessionId: 1  Cid: 0910    Peb: 7ff669b8e000  ParentCid: 0a98
    DirBase: 3fdba000  ObjectTable: ffffc00007bfd540  HandleCount:
<Data Not Accessible>
    Image: explorer.exe
```

15. 复制一个进程的地址，并输入以下命令：

    !process *Address* 2

    例如：`!process ffffe00000d5290 2`

    输出显示进程中的线程。

    dbgcmd

```
0:000>0:000>0: kd> !process ffffe00000d52900 2
PROCESS ffffe00000d52900
    SessionId: 1  Cid: 0910    Peb: 7ff669b8e000  ParentCid: 0a98
    DirBase: 3fdba000  ObjectTable: ffffc00007bfd540  HandleCount:
    Image: explorer.exe
```

```
        THREAD ffffe00000a0d880  Cid 0910.090c  Teb: 00007ff669b8c000
            ffffe00000d57700  SynchronizationEvent

        THREAD ffffe00000e48880  Cid 0910.0ad8  Teb: 00007ff669b8a000
            ffffe00000d8e230  NotificationEvent
            ffffe00000cf6870  Semaphore Limit 0xffff
            ffffe000039c48c0  SynchronizationEvent
        ...
        THREAD ffffe00000e6d080  Cid 0910.0cc0  Teb: 00007ff669a10000
            ffffe0000089a300  QueueObject
```

16. 复制一个线程的地址，并输入以下命令：

    !thread *Address*

    例如：`!thread ffffe00000e6d080`

    输出显示有关单个线程的信息。

    dbgcmd

```
0: kd> !thread ffffe00000e6d080
THREAD ffffe00000e6d080  Cid 0910.0cc0  Teb: 00007ff669a10000
Win32Thread: 0000000000000000 WAIT: ...
    ffffe0000089a300  QueueObject
Not impersonating
DeviceMap                 ffffc000034e7840
Owning Process            ffffe00000d52900       Image:
explorer.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      13777          Ticks: 2 (0:00:00:00.031)
Context Switch Count      2              IdealProcessor: 1
UserTime                  00:00:00.000
KernelTime                00:00:00.000
Win32 Start Address ntdll!TppWorkerThread (0x00007ffd14ab2850)
Stack Init ffffd00021bf1dd0 Current ffffd00021bf1580
Base ffffd00021bf2000 Limit ffffd00021bec000 Call 0
Priority 13 BasePriority 13 UnusualBoost 0 ForegroundBoost 0 IoPriority
2 PagePriority 5
...
```

17. 要查看即插即用设备树中的所有设备节点，请输入以下命令：

    !devnode 0 1

    dbgcmd

```
0:000>0: kd> !devnode 0 1
Dumping IopRootDeviceNode (= 0xffffe000002dbd30)
DevNode 0xffffe000002dbd30 for PDO 0xffffe000002dc9e0
```

```
  InstancePath is "HTREE\ROOT\0"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0xffffe000002d9d30 for PDO 0xffffe000002daa40
    InstancePath is "ROOT\volmgr\0000"
    ServiceName is "volmgr"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
    DevNode 0xffffe00001d49290 for PDO 0xffffe000002a9a90
      InstancePath is "STORAGE\Volume\{3007dfd3-df8d-11e3-824c-
806e6f6e6963}#0000000000100000"
      ServiceName is "volsnap"
      TargetDeviceNotify List - f 0xffffc0000031b520  b
0xffffc0000008d0f0
      State = DeviceNodeStarted (0x308)
      Previous State = DeviceNodeStartPostWork (0x307)
...
```

18. 要查看设备节点及其硬件资源，请输入以下命令：

    !devnode 0 9

    dbgcmd

```
0:000>...
        DevNode 0xffffe000010fa770 for PDO 0xffffe000010c2060
          InstancePath is
"PCI\VEN_8086&DEV_2937&SUBSYS_2819103C&REV_02\3&33fd14ca&0&D0"
          ServiceName is "usbuhci"
          State = DeviceNodeStarted (0x308)
          Previous State = DeviceNodeEnumerateCompletion (0x30d)
          TranslatedResourceList at 0xffffc00003c78b00  Version 1.1
Interface 0x5  Bus #0
            Entry 0 - Port (0x1) Device Exclusive (0x1)
              Flags (0x131) - PORT_MEMORY PORT_IO 16_BIT_DECODE
POSITIVE_DECODE
              Range starts at 0x3120 for 0x20 bytes
            Entry 1 - DevicePrivate (0x81) Device Exclusive (0x1)
              Flags (0000) -
              Data - {0x00000001, 0x00000004, 0000000000}
            Entry 2 - Interrupt (0x2) Shared (0x3)
              Flags (0000) - LEVEL_SENSITIVE
              Level 0x8, Vector 0x81, Group 0, Affinity 0xf
...
```

19. 要查看服务名称为 **disk** 的设备节点，请输入以下命令：

    !devnode 0 1 disk

    dbgcmd

```
0: kd> !devnode 0 1 disk
Dumping IopRootDeviceNode (= 0xffffe000002dbd30)
DevNode 0xffffe0000114fd30 for PDO 0xffffe00001159610
  InstancePath is
"IDE\DiskST3250820AS_____________________________3.CHL___\5&14544e82&0&
0.0.0"
  ServiceName is "disk"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
...
```

20. !devnode 0 1 的输出显示节点的物理设备对象 (PDO) 地址。 复制物理设备对象 (PDO) 的地址，并输入以下命令：

    !devstack *PdoAddress*

    例如：`<PdoAddress>!devstack 0xffffe00001159610`

    dbgcmd

    ```
    0:000>0: kd> !devstack 0xffffe00001159610
      !DevObj           !DrvObj            !DevExt           ObjectName
      ffffe00001d50040  \Driver\partmgr    ffffe00001d50190
      ffffe00001d51450  \Driver\disk       ffffe00001d515a0  DR0
      ffffe00001156e50  \Driver\ACPI       ffffe000010d8bf0
    ```

21. 要获取有关驱动程序 disk.sys 的信息，请输入以下命令：

    !drvobj disk 2

    dbgcmd

    ```
    0:000>0: kd> !drvobj disk 2
    Driver object (ffffe00001d52680) is for:
     \Driver\disk
    DriverEntry:   fffff800006b1270 disk!GsDriverEntry
    DriverStartIo: 00000000
    DriverUnload:  fffff800010b0b5c CLASSPNP!ClassUnload
    AddDevice:     fffff800010aa110 CLASSPNP!ClassAddDevice

    Dispatch routines:
    [00] IRP_MJ_CREATE                      fffff8000106d160
    CLASSPNP!ClassGlobalDispatch
    [01] IRP_MJ_CREATE_NAMED_PIPE           fffff80002b0ab24
    nt!IopInvalidDeviceRequest
    [02] IRP_MJ_CLOSE                       fffff8000106d160
    CLASSPNP!ClassGlobalDispatch
    [03] IRP_MJ_READ                        fffff8000106d160
    CLASSPNP!ClassGlobalDispatch
    ...
    ```

```
[1b] IRP_MJ_PNP                         fffff8000106d160
CLASSPNP!ClassGlobalDispatch
```

22. `!drvobj` 的输出显示调度例程的地址。 例如 `CLASSPNP!ClassGlobalDispatch`。 要在 `ClassGlobalDispatch` 处设置并验证断点，请输入以下命令：

    bu CLASSPNP!ClassGlobalDispatch

    bl

    输入 `g` 让目标计算机运行。

    如果目标计算机没有立即中断调试器，请在目标计算机上执行一些操作（例如，打开记事本或保存文件）。 调用 `ClassGlobalDispatch` 时，目标计算机将中断调试器。 要查看堆栈跟踪，请输入以下命令：

    .reload

    k

    输出类似于以下示例：

    dbgcmd

    ```
    2: kd> k
    Child-SP          RetAddr           Call Site
    ffffd000`21d06cf8 fffff800`0056c14e CLASSPNP!ClassGlobalDispatch
    ffffd000`21d06d00 fffff800`00f2c31d volmgr!VmReadWrite+0x13e
    ffffd000`21d06d40 fffff800`0064515d
    fvevol!FveFilterRundownReadWrite+0x28d
    ffffd000`21d06e20 fffff800`0064578b rdyboost!SmdProcessReadWrite+0x14d
    ffffd000`21d06ef0 fffff800`00fb06ad rdyboost!SmdDispatchReadWrite+0x8b
    ffffd000`21d06f20 fffff800`0085cef5 volsnap!VolSnapReadFilter+0x5d
    ffffd000`21d06f50 fffff800`02b619f7 Ntfs!NtfsStorageDriverCallout+0x16
    ...
    ```

23. 要结束调试会话，请输入以下命令：

    qd

命令摘要

- “帮助”菜单上的“内容”命令
- .sympath（设置符号路径）
- .reload（重新加载模块）
- x（检查符号）

- g（转到）
- dt（显示类型）
- 在“调试”菜单上“中断”命令
- lm（列出已加载的模块）
- k（显示堆栈回溯）
- bu（设置断点）
- bl（断点列表）
- bc（断点清除）
- “调试”菜单上的“单步调试”命令 (**F11**)
- !process
- !thread
- !devnode
- !devstack
- !drvobj
- qd（退出和分离）

另请参阅

- WinDbg 入门（用户模式）
- 自动设置 KDNET 网络内核调试
- 调试器操作
- 调试技术
- 下载并安装 WinDbg Windows 调试器
- WinDbg 功能

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

调试环境

项目 • 2024/03/05

有六种可用的调试环境：

- WinDbg
- WinDbg (Classic)
- 内核调试器 (KD)
- NTKD
- 控制台调试器 (CDB)
- NT 符号调试器 (NTSD)

以下部分介绍调试环境。

WinDbg

WinDbg 是 WinDbg 的最新版本，具有新式视觉效果、更快的窗口速度和成熟的脚本体验，使用可扩展的调试器数据模型构建。 WinDbg 使用与 WinDbg (Classic) 相同的基础引擎，因此用户所习惯使用的所有命令、扩展和工作流程仍可正常工作。

有关详细信息，请参阅 WinDbg 功能。

WinDbg (Classic)

Microsoft Windows 调试器 WinDbg (Classic) 是一种基于 Windows 的调试器，可进行用户模式和内核模式调试。 WinDbg 可对 Windows 内核、内核模式驱动程序和系统服务以及用户模式应用程序和驱动程序进行调试。

WinDbg 使用 Visual Studio 调试符号格式进行源级调试。 它可以访问拥有 PDB 符号文件的模块中的任何符号或变量，也可以访问使用 COFF 符号文件（如 Windows .dbg 文件）编译的模块所公开的任何公共函数名。

WinDbg 可以查看源代码、设置断点、查看变量（包括 C++ 对象）、堆栈跟踪和内存。 其调试器命令窗口允许用户发出各种命令。

对于内核模式调试，WinDbg 通常需要两台计算机（主计算机和目标计算机）。 WinDbg 还支持针对用户模式和内核模式目标的各种远程调试选项。

WinDbg 是 CDB/NTSD 和 KD/NTKD 的对应图形界面。

KD

Microsoft 内核调试器 (KD) 是基于字符的控制台程序，可对所有基于 NT 的操作系统的内核模式活动进行深入分析。 可以使用 KD 调试内核模式组件和驱动程序，或监视操作系统本身的行为。 KD 还支持多处理器调试。

通常，KD 不会在正在调试的计算机上运行。 内核模式调试需要两台计算机（*主计算机*和*目标计算机*）。

NTKD

KD 调试器有一个名为 NTKD 的变体。 它和 KD 在各方面都完全相同，只是它在启动时会生成一个新的文本窗口，而 KD 则继承了调用时的命令提示符窗口。

CDB

Microsoft 控制台调试器 (CDB) 是一种基于字符的控制台程序，可对 Windows 用户模式内存和构造进行低级别分析。 名称*控制台调试器*用于表示 CDB 被归类为控制台应用程序；但并不意味着目标应用程序必须是控制台应用程序。 事实上，CDB 完全能够调试控制台应用程序和图形 Windows 程序。

CDB 的功能非常强大，可用于调试正在运行或最近崩溃的程序（实时分析），而且设置简单。 它可用于调查正常工作的应用程序的行为。 在应用程序失败的情况下，CDB 可用于获取堆栈跟踪或查看有罪参数。 由于它是基于字符的，因此在网络上（使用远程访问服务器）运行良好。

借助 CDB，可以显示和执行程序代码、设置断点，以及检查和更改内存中的值。 CDB 可以通过反汇编二进制代码并显示汇编指令来分析二进制代码。 它还可以直接分析源代码。

由于 CDB 可以通过地址或全局符号访问内存位置，因此可以通过名称而不是地址来引用数据和说明，以便轻松查找和调试代码的特定部分。 CDB 支持调试多个线程和进程。 它是可扩展的，可以读取和写入分页和非分页内存。

如果目标应用程序本身是控制台应用程序，则目标将与 CDB 共享控制台窗口。 要为目标控制台应用程序生成单独的控制台窗口，请使用 *-2* 命令行选项。

NTSD

CDB 调试器有一个名为 Microsoft NT 符号调试器 (NTSD) 的变体。 它和 CDB 在各方面都完全相同，只是它在启动时会生成一个新的文本窗口，而 CDB 则继承了调用时的命令提示符窗口。

由于 **start** 命令还可用于生成新的控制台窗口，因此以下两个构造将给出相同的结果：

控制台

```
start cdb parameters
ntsd parameters
```

可以重定向 NTSD（或 CDB）的输入和输出，以便可以从内核调试器（Visual Studio、WinDbg 或 KD）对其进行控制。 如果此方法与 NTSD 一起使用，则不会显示任何控制台窗口。 因此，从内核调试器控制 NTSD 特别有用，因为它可以产生一个极其轻量级的调试器，几乎不会给包含目标程序的计算机带来任何负担。 此组合可用于调试系统进程、关闭和启动的后续阶段。 有关详细信息，请参阅从内核调试器控制用户模式调试器。

另请参阅

Windows 调试

WinDbg 功能

选择 32 位或 64 位调试工具

项目 • 2024/03/08

安装适用于 Windows 的调试工具时，可获取一组 32 位工具和一组 64 位工具。

如果使用其他调试环境之一（WinDbg、KD、CDB 或 NTSD），则必须自行做出选择。 若要确定要使用的调试工具集，需要知道在主计算机上运行的处理器类型以及主计算机是运行 32 位还是 64 位版本的 Windows。

运行调试器的计算机称为*主计算机*，正在调试的计算机称为*目标计算机*。

运行 32 位版本的 Windows 的主机

如果主计算机运行的是 32 位版本的 Windows，请使用 32 位调试工具。 （这种情况适用于基于 x86 和基于 x64 的目标。）

运行 64 位版本的 Windows 的基于 x64 的主计算机

如果主计算机使用基于 x64 的处理器，并且运行的是 64 位版本的 Windows，则适用以下规则：

- 如果要分析转储文件，可以使用 32 位调试工具或 64 位调试工具。 （转储文件是用户模式转储文件还是内核模式转储文件，以及转储文件是在基于 x86 还是基于 x64 的平台上进行的，都并不重要。）
- 如果要执行实时内核模式调试，可以使用 32 位调试工具或 x64 调试工具。 （这种情况适用于基于 x86 和基于 x64 的目标。）
- 如果要调试与调试器在同一台计算机上运行的实时用户模式代码，请使用 64 位工具调试在 WOW64 上运行的 64 位代码和 32 位代码。 要将调试器设置为 32 位或 64 位模式，请使用 **.effmach** 命令。
- 如果要调试在单独的目标计算机上运行的实时 32 位用户模式代码，请使用 32 位调试工具。

另请参阅

Windows 调试

设置调试（内核模式和用户模式）

项目 • 2023/11/08

- 设置内核模式调试后，可以使用 **WinDbg** 或 **KD** 建立调试会话。
- 设置用户模式调试后，可以使用 **WinDbg**、 **CDB** 或 **NTSD** 建立调试会话。

注意 Windows 调试器包含在 Windows 调试工具中。 这些调试器不同于 Visual Studio 附带的 Visual Studio 调试器。 有关详细信息，请参阅 Windows 调试。

本部分中的内容

配置传输

- 设置内核模式调试

支持的目标电脑 NIC

- Windows 10 中的网络内核调试支持的以太网 NIC
- Windows 8.1 中的网络内核调试支持的以太网 NIC
- Windows 8 中的网络内核调试支持的以太网 NIC

其他配置工具

- 配置 tools.ini
- 使用 KDbgCtrl

设置内核模式调试

项目 • 2024/10/31

本部分介绍如何设置内核模式调试。

> **提示**
>
> 建议的方法是使用网络（KDNET）调试，并使用 kdnet 实用工具自动配置。 有关详细信息，请参阅**设置 KDNET 网络内核自动调试**。

本节内容

KDNET 网络连接

- 自动设置 KDNET 网络内核调试
- 手动设置 KDNET 网络内核调试

虚拟机

- 设置虚拟机主机 - KDNET 的内核模式调试
- 使用 COM 端口手动设置虚拟机的内核模式调试

单个电脑

- 手动设置单个计算机的本地内核调试

电缆连接

- 设置 USB KDNET EEM 内核模式调试 （KDNET-EEM-USB）
- 设置 USB KDNET 内核模式调试 （KDNET-USB）
- 设置 USB 3.0 xHCI-DBC 内核模式调试 （KDUSB）
- 手动设置通过串行线缆进行的内核模式调试

另请参阅

设置调试（内核模式和用户模式）

自动设置 KDNET 网络内核调试

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

自动设置 KDNET 网络内核调试

项目 • 2024/08/31

Windows 调试工具支持通过网络进行内核调试。 本文介绍如何使用 *kdnet.exe* 安装工具自动设置网络调试。

运行调试器的计算机称为主计算机，正在调试的计算机称为目标计算机。 主机必须运行 **Windows 7** 或更高版本，并且目标计算机必须运行 **Windows 8** 或更高版本。

确定主机的 IP 地址。

1. 确认目标 PC 和主机 PC 已使用适当的网络电缆连接到网络集线器或交换机。
2. 在主机上，打开命令提示符，并输入 `IPConfig` 以显示 IP 配置。
3. 在命令输出中，找到以太网适配器的 IPv4 地址。

控制台

```
...

Ethernet adapter Ethernet:
...

IPv4 Address. . . . . . . . . . . : <YourHostIPAddress>
...
```

或者，使用设置、网络和 Internet、以太网收集 IPv4 地址信息。

4. 记下打算用于调试的网络适配器的 IPv4 地址。

设置主机和目标计算机

按照以下步骤，使用 kdnet.exe 实用工具在目标 PC 上自动配置调试器设置。

1. 确认主计算机系统上已安装 Windows 调试工具。 有关下载和安装调试器工具的信息，请参阅适用于 Windows 的调试工具。
2. 找到 *kdnet.exe* 和 *VerifiedNICList.xml* 文件。 默认情况下，文件位于以下位置：

   控制台

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64
```

这些说明假定目标电脑和主电脑都运行 64 位版本的 Windows。 如果情况并非如此，最好的办法是在主机上运行与目标计算机相同“位数”的工具。 例如，如果目标电脑运行的是 32 位 Windows，请在主电脑上运行 32 版本的调试器。 有关详细信息，请参阅选择 32 位或 64 位调试工具。

3. 在主计算机上，将这两个文件复制到网络共享或 U 盘，以便在目标计算机上可用。

4. 在目标计算机上，创建 **C:\KDNET** 目录，并将 *kdnet.exe* 和 *VerifiedNICList.xml* 文件复制到该目录。

> **ⓘ 重要**
>
> 在使用 kdnet.exe 更改启动信息之前，可能需要在测试 PC 上暂时挂起 Windows 安全功能，例如 BitLocker 和安全启动。 在已禁用这些安全功能的情况下，在测试完成后重新启用这些安全功能，并适当地管理测试 PC。

5. 在目标计算机上，以管理员身份打开命令提示符。 输入此命令以验证目标计算机是否具有受支持的网络适配器：

```
控制台

C:\KDNET>kdnet.exe
Network debugging is supported on the following NICs:
busparams=1.0.0, Broadcom NetXtreme Gigabit Ethernet, Plugged in.
This Microsoft hypervisor supports using KDNET in guest VMs.
```

6. 当来自 kdnet.exe 的输出指示目标上的网络适配器受支持时，可以继续操作。

7. 键入以下命令以设置主机系统的 IP 地址并生成唯一的连接密钥。 使用 IP 地址或主机系统的名称。 在建议的 50000-50039 范围内，为每个使用的目标/主计算机对选择唯一的端口地址。

```
控制台

C:\KDNET>kdnet.exe <HostComputerIPAddress> <YourDebugPort>

Enabling network debugging on Intel(R) 82577LM Gigabit Network
Connection.
Key=2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
```

8. 将返回的密钥复制到*记事本* *.txt* 文件中。

将 WinDbg 连接到目标进行内核调试

在主计算机上打开 WinDbg。 在“文件”菜单上，选择“内核调试”。 在“内核调试”对话框中，打开 **Net** 选项卡。粘贴以前保存在记事本 .txt 文件中的端口号和密钥。 选择“确定”。

还可以通过打开命令提示符并输入以下命令来启动 WinDbg 会话。 <YourPort> 是你之前选择的端口，<YourKey> 是 kdnet.exe 之前返回的密钥。 粘贴以前保存在记事本 .txt 文件中的密钥。

控制台

```
windbg.exe -k -d net:port=<YourDebugPort>,key=<YourKey>
```

示例中所示的可选 `-d` 参数启用早期中断。 有关详细信息，请参阅 WinDbg 命令行选项。

如果系统提示允许 WinDbg 通过防火墙访问端口，则允许 WinDbg 访问**所有三种**不同网络类型的端口。

此时，调试器将等待目标重新连接。 文本显示在调试器命令窗口中。 该文本将类似于以下内容：

控制台

```
Microsoft (R) Windows Debugger Version 1.0.1908.30002 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Using NET for debugging
Opened WinSock 2.0
Waiting to reconnect...
```

重启目标 PC

一旦调试器处于“正在等待重新连接...”阶段，请重新启动目标计算机。 重启 PC 的一种方法是从管理员的命令提示符处使用此命令。

控制台

```
shutdown -r -t 0
```

目标电脑重启后，调试器应自动连接。

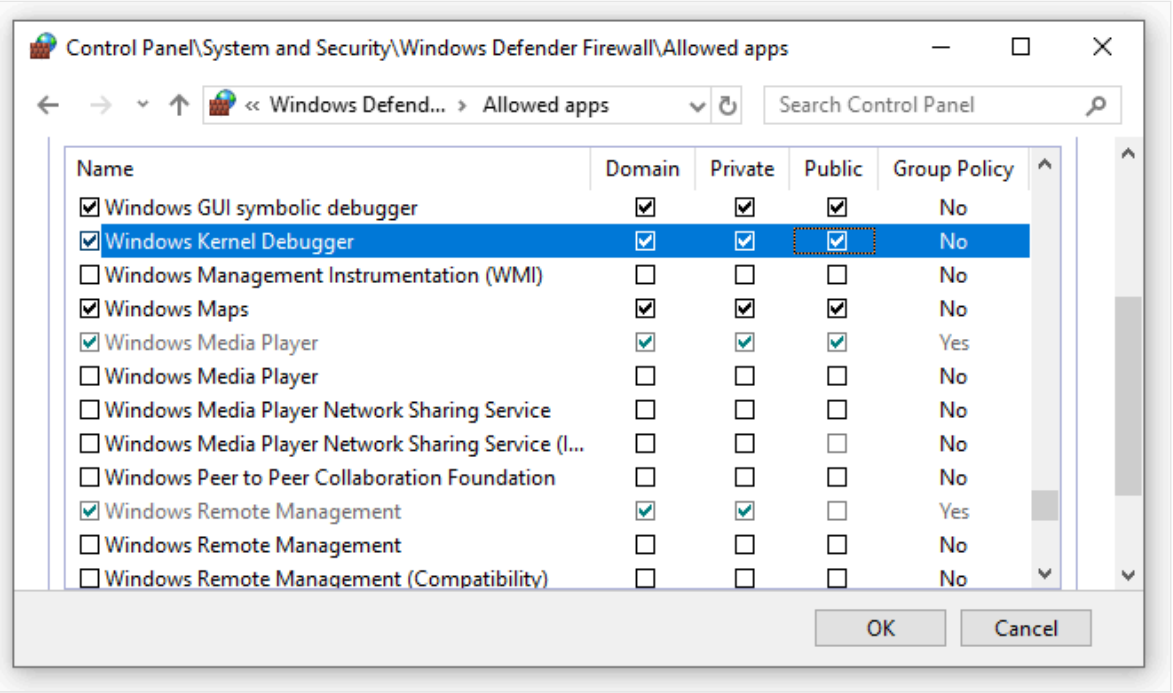
故障排除提示

必须允许通过防火墙调试应用程序

在运行调试器的主机端，必须启用所有类型的网络，以允许调试器通过防火墙与目标通信。

可以使用控制面板允许通过防火墙进行访问。

1. 打开“控制面板”>“系统和安全”，然后选择“允许应用通过 Windows 防火墙”。
    - (WinDbg) 在应用程序列表中，找到 *WinDbg 引擎进程 (TCP)（全部）*。
    - （WinDbg (classic)）在应用程序列表中，找到 *Windows GUI 符号调试器*和 *Windows 内核调试器*。
2. 使用复选框允许这两个应用程序通过**所有三种**不同网络类型。
3. 向下滚动并选择“确定”保存防火墙更改。 重启调试程序。



使用 ping 测试连接

如果调试器超时且未连接，请使用目标 PC 上的 ping 命令验证连接。

控制台

```
C:\>Ping <HostComputerIPAddress>
```

选择用于网络调试的端口

如果调试器超时且未连接，可能是因为默认端口号 50000 已被使用或被阻止。

可以选择从 49152 到 65535 的任何端口号。 建议的范围介于 50000 和 50039 之间。 选择的端口由在主机上运行的调试器打开以独占访问权限。

可用于网络调试的端口号范围可能受公司网络策略的限制。 要确定公司的策略是否限制了可用于网络调试的端口范围，请咨询网络管理员。

支持的网络适配器

如果在运行 kdnet.exe 时显示“此计算机中的任何 NIC 都不支持网络调试”，则表示不支持网络适配器

主机可以使用任何网络适配器，但目标计算机必须使用 Windows 调试工具支持的网络适配器。 有关支持的网络适配器列表，请参阅 Windows 10 中的网络内核调试支持的以太网 NIC 以及 Windows 8.1 中的网络内核调试支持的以太网 NIC。

启用其他调试类型

从 Windows 10 2020 年 10 月更新 (20H2) 开始，支持以下选项来启用四种类型的调试：

- *b* - 启用 bootmgr 调试。 请参阅 BCDEdit /bootdebug。
- *w* - 启用 winload 调试。 请参阅 BCDEdit /bootdebug。
- *h* - 启用虚拟机监控程序调试。 请参阅 BCDEdit /hypervisorsettings。
- *k* - 启用内核调试。 请参阅 WinDbg 入门指南（内核模式）。

可以指定调试类型的任意组合。

如果未指定调试类型，则启用内核调试。

如果同时启用了虚拟机监控程序和内核调试，虚拟机监控程序端口将设置为 `<YourDebugPort>` + 1。

用法示例

使用 - bkw 选项启用 bootmgr、内核和 winload 调试。

控制台

```
C:\>kdnet.exe <HostComputerIPAddress> <YourDebugPort> -bkw

Enabling network debugging on Intel(R) 82577LM Gigabit Network Connection.
Key=2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
```

调试类型选项摘要

展开表

KNDET 选项	说明	等效设置命令
b	启用 bootmgr 调试	bcdedit /bootdebug {bootmgr} on
h	启用虚拟机监控程序调试	bcdedit /set hypervisordebug on
k	启用内核调试	bcdedit /debug on
w	启用 winload 调试	bcdedit /bootdebug on

指定总线参数

如果 kdnet 无法自动确定传输的总线参数，请使用以下语法在命令行中使用 `/busparams` 选项指定它们。

```
kdnet.exe /busparams [b.d.f] [host] [port] [-[b][h][k][w]]
```

`b.d.f` 指定要配置的设备的总线参数。

使用目标计算机上的设备管理器确定要用于调试的适配器的 PCI 总线、设备和函数编号。对于总线参数，请输入 *b.d.f*，其中 *b*、*d* 和 *f* 是适配器的总线编号、设备编号和函数编号。 这些值显示在“常规”选项卡上的“位置”下的设备管理器中。

例如：

控制台

```
C:\>kdnet.exe /busparams 0.29.7 <HostComputerIPAddress> <YourDebugPort> -bkw
```

另请参阅

- Windows 10 中的网络内核调试支持的以太网 NIC
- Windows 8.1 中的网络内核调试支持的以太网 NIC
- 手动设置 KDNET 网络内核调试
- WinDbg 入门（内核模式）
- 调试通用驱动程序 - 分步实验（Echo 内核模式）

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

手动设置 KDNET 网络内核调试

项目 • 2024/03/08

Windows 调试工具支持通过网络进行内核调试。 本文介绍如何手动设置网络调试。

> **ⓘ 重要**
>
> 手动设置网络调试是一个复杂且容易出错的过程。 要自动设置网络调试，请参阅**自动设置 KDNET 网络内核调试**。 强烈建议对所有调试器用户使用 KDNET 实用工具。

运行调试器的计算机称为*主计算机*，正在调试的计算机称为*目标计算机*。 主机必须运行 Windows 7 或更高版本，并且目标计算机必须运行 Windows 8 或更高版本。

相对于其他类型的连接进行调试，通过网络进行调试具有以下优势。

- 主机和目标计算机可以位于本地网络上的任意位置。
- 从一台主机调试多个目标计算机很容易。
- 假设有任意两台计算机，可能它们都有以太网适配器。 它们都具有串行端口或 1394 端口的可能性较小。
- 网络调试比串行端口调试更快。

支持的网络适配器

主机可以使用任何网络适配器，但目标计算机必须使用 Windows 调试工具支持的网络适配器。 有关支持的网络适配器列表，请参阅 Windows 10 中的网络内核调试支持的以太网 NIC 以及 Windows 8.1 中的网络内核调试支持的以太网 NIC。

安装 Windows 调试工具

确认主机系统上已安装适用于 Windows 的调试工具。 有关下载和安装调试器工具的信息，请参阅适用于 Windows 的调试工具。

确定主机的 IP 地址。

使用以下过程之一来确定主机的 IP 地址。

1. 在主机上打开命令提示符，输入以下命令：

```
控制台
```

```
ipconfig
```

记下打算用于调试的网络适配器的 IPv4 地址。

2. 在目标计算机上，打开命令提示符并输入以下命令，其中 *YourIPAddress* 是主机的 IP 地址：

控制台

```
ping -4 <YourIPAddress>
```

选择用于网络调试的端口

选择用于在主机和目标计算机上调试的端口号。 可以选择从 49152 到 65535 的任意数字。 建议的范围为 50000 - 50039。 选择的端口由在主机上运行的调试器打开以独占访问权限。 请注意选择主机上运行的任何其他应用程序不使用的端口号。

> **备注**
>
> 可用于网络调试的端口号范围可能受公司网络策略的限制。 无法从主机中判断限制内容。 要确定公司的策略是否限制了可用于网络调试的端口范围，请咨询网络管理员。

如果将多台目标计算机连接到单个主机，则每个连接必须具有唯一的端口号。 例如，如果将 100 台目标计算机连接到单个主机，则可以将端口 50000 分配给第一个连接，将端口 50001 分配给第二个连接，依此类推。

> **备注**
>
> 另一台主机可以使用相同范围的端口（50000 到 50099）连接到其他 100 台目标计算机。

设置目标计算机

1. 验证目标计算机是否具有受支持的网络适配器。 有关详细信息，请参阅：

   - Windows 8.1 中的网络内核调试支持的以太网 NIC
   - Windows 10 中的网络内核调试支持的以太网 NIC

2. 使用适当的网络电缆将支持的适配器连接到网络中心或交换机。

> **ⓘ 重要**
>
> 在使用 BCDEdit 更改启动信息之前，可能需要在测试电脑上暂时挂起 Windows 安全功能，例如 BitLocker 和安全启动。 测试完成后重新启用这些安全功能，并在禁用安全功能时适当地管理测试电脑。

3. 在提升的命令提示符中，输入以下命令，其中 *w.x.y.z* 是主机的 IP 地址，*n* 是所选端口号：

   控制台

   ```
   bcdedit /debug on

   bcdedit /dbgsettings net hostip:w.x.y.z port:n
   ```

4. BCDEdit 显示自动生成的密钥。 复制密钥并将其存储在可移动存储设备上，例如 U 盘。 在主机上启动调试会话时，需要密钥。

5. 使用设备管理器确定要用于调试的适配器的 PCI 总线、设备和函数编号。 这些值显示在“常规”选项卡上的“位置”下的设备管理器中。然后在提升的命令提示符中，输入以下命令，其中 *b*、*d* 和 *f* 是适配器的总线编号、设备编号和函数编号：

   控制台

   ```
   bcdedit /set "{dbgsettings}" busparams b.d.f
   ```

6. 附加内核调试器后，将重新启动目标电脑。

> **ⓘ 备注**
>
> 如果打算在目标计算机上安装 Hyper-V 角色，请参阅**设置虚拟机主机的网络调试**。

注意如果目标计算机位于扩展坞中，并且已为作为扩展坞一部分的网络适配器启用网络调试，请不要从扩展坞中删除计算机。 如果需要从扩展坞中删除目标计算机，请先禁用内核调试。 要在目标计算机上禁用内核调试，请以管理员身份打开命令提示符并输入命令 `bcdedit /debug off`。 重新启动目标计算机。

启动调试会话

确认主机的网络适配器使用合适的网络电缆连接到网络中心或交换机。

在主计算机上打开 WinDbg。 在“文件”菜单上，选择“内核调试”。 在“内核调试”对话框中，打开“Net”选项卡。输入端口号和密钥。 选择“确定”。

还可以通过打开命令提示符并输入以下命令来启动与 WinDbg 的会话，其中 *n* 是端口号，*MyKey* 是 BCDEdit 在设置目标计算机时自动生成的密钥：

控制台

```
windbg -k net:port=<n>,key=<MyKey>
```

如果系统提示允许 WinDbg 通过防火墙访问端口，则允许 WinDbg 访问**所有三种**不同网络类型的端口。

使用 KD

在主机上，打开命令提示符。 输入以下命令，其中 *n* 是端口号，*MyKey* 是 BCDEdit 在设置目标计算机时自动生成的密钥：

控制台

```
kd -k net:port=<n>,key=<MyKey>
```

如果系统提示允许 WinDbg 通过防火墙访问端口，则允许 WinDbg 访问**所有三种**不同网络类型的端口。

重启目标电脑

调试器连接并等待连接后，重新启动目标计算机。 重启电脑的一种方法是从管理员的命令提示符处使用此命令：

控制台

```
shutdown -r -t 0
```

重新启动目标计算机后，主机操作系统中的调试器应连接。

连接到主机上的目标后，在调试器上命中中断，可以开始调试。

允许调试器通过防火墙

首次尝试建立网络调试连接时，系统可能会提示你允许调试应用程序（WinDbg 或 KD）通过防火墙进行访问。 Windows 的客户端版本显示提示，但 Windows 的服务器版本不显示提示。 应通过选中**所有三种**网络类型的复选框来响应提示：域、专用和公共。

如果未收到提示，或者当提示可用时未选中复选框，则必须使用控制面板允许通过运行调试器的主机计算机上的防火墙进行访问。 打开“控制面板”>“系统和安全”，然后选择“允许应用通过 Windows 防火墙”。

- (WinDbg) 在应用程序列表中，找到 *WinDbg 引擎进程 (TCP)（全部）*。
- （经典 WinDbg）在应用程序列表中，找到 *Windows GUI 符号调试器*和*Windows 内核调试器*。

使用复选框允许这些应用程序通过防火墙。 重启调试应用程序（WinDbg 或 KD）。

加密密钥

为了使目标计算机保持安全，在主机和目标计算机之间传输的数据包必须加密。 应使用自动生成的加密密钥（配置目标计算机时由 BCDEdit 提供）。 建议使用自动生成的加密密钥，因为它更安全，并提供用于与特定目标建立连接的唯一值。

网络调试使用指定为 4 个 64 位值的 256 位密钥，以 36 为底数，由句点分隔。 每个 64 位值最多使用 13 个字符指定。 有效字符是字母 *a* 到 *z*，数字 0 到 9。 不允许使用特殊字符。

要指定自己的密钥，请在目标计算机上打开提升的命令提示符。 输入以下命令，其中 `w.x.y.z` 是主机的 IP 地址，*n* 是端口号，*Key* 是密钥：

控制台

```
bcdedit /dbgsettings net hostip:w.x.y.z port:n key:Key
```

每当更改 *dbgsettings* 时，都需要重新启动目标计算机。

故障排除提示

必须允许调试应用程序通过防火墙

首次尝试建立网络调试连接时，系统可能会提示你允许调试应用程序（WinDbg 或 KD）通过防火墙进行访问。 Windows 的客户端版本显示提示，但 Windows 的服务器版本不显示提示。 应通过选中**所有三种**网络类型的复选框来响应提示：域、专用和公共。

如果未收到提示，或者当提示可用时未选中复选框，则必须使用控制面板允许通过防火墙进行访问。 打开“控制面板”>“系统和安全”，然后选择“允许应用通过 Windows 防火墙”。

- (WinDbg) 在应用程序列表中，找到 *WinDbg 引擎进程 (TCP)（全部）*。
- （WinDbg (classic)）在应用程序列表中，找到 *Windows GUI 符号调试器*和 *Windows 内核调试器*。

使用复选框允许这些应用程序通过防火墙。 向下滚动并选择“确定”保存防火墙更改。 重启调试程序。

端口号必须位于网络策略允许的范围内

可用于网络调试的端口号范围可能受公司网络策略的限制。 要确定公司的策略是否限制了可用于网络调试的端口范围，请咨询网络管理员。 在目标计算机上，以管理员身份打开命令提示符并输入命令 `bcdedit /dbgsettings`。 输出将如下所示：

控制台

```
C:\> bcdedit /dbgsettings
key                     XXXXXX.XXXXX.XXXXX.XXXXX
debugtype               NET
hostip                  169.168.1.1
port                    50085
dhcp                    Yes
The operation completed successfully.
```

在上述输出中，端口值为 50085。 如果端口的值超出了网络管理员允许的范围，请输入以下命令。 *w.x.y.z* 值是主机的 IP 地址，*YourDebugPort* 是允许范围内的端口号。

控制台

```
bcdedit /dbgsettings net hostip:w.x.y.z port:YourDebugPort
```

更改目标计算机调试器设置后，使用新端口设置重新运行主机上的调试器，然后重新启动目标计算机。

使用 ping 测试连接

如果调试器未连接，请使用目标电脑上的 ping 命令验证连接。

控制台

```
C:\>Ping <HostComputerIPAddress>
```

> **备注**
>
> 如果主机未配置为在网络上可发现，则可能不起作用，因为防火墙可能会阻止 ping 请求。 如果防火墙阻止 ping 请求，则 ping 主机时不会收到任何响应。

调试器如何获取目标计算机的 IP 地址

目标计算机上的 KDNET 尝试使用动态主机配置协议 (DHCP) 获取用于调试的网络适配器的可路由 IP 地址。 如果 KDNET 获取 DHCP 分配的地址，则目标计算机可由网络上任意位置的主机进行调试。 如果 KDNET 无法获取 DHCP 分配的地址，则它使用自动专用 IP 寻址（APIPA）来获取本地链接 IP 地址。 本地链接 IP 地址不可路由，因此主机和目标计算机无法使用本地链接 IP 地址通过路由器进行通信。 在这种情况下，如果将主机和目标计算机插入同一网络中心或交换机，网络调试将正常工作。

使用基于 PCI 的 NIC 在物理计算机上设置 KDNET 时，始终指定 busparams

如果要在具有 PCI 或基于 PCIe 的 NIC 的物理计算机上设置 KDNET，应始终为要用于 KDNET 的 NIC 指定 busparams。 要指定总线参数，请打开设备管理器，找到要用于调试的网络适配器。 打开网络适配器的属性页，并记下“常规”选项卡中“位置”下显示的总线编号、设备编号和函数编号。在提升的命令提示符中，输入以下命令，其中 *b*、*d* 和 *f* 是十进制格式的总线、设备和函数编号：

控制台

```
bcdedit /set "{dbgsettings}" busparams b.d.f
```

当调试器在主机上运行并等待连接时，请使用以下命令重新启动目标计算机：

控制台

```
shutdown -r -t 0
```

手动删除 BCDEdit 条目

通常不需要手动删除，但此处提供了异常情况下的故障排除程序。

使用 kdnet 实用工具时，不需要手动删除条目。 有关详细信息，请参阅自动设置 KDNET 网络内核调试。

使用 `bcdedit -deletevalue` 时，必须提供有效的 bcd 元素名称。 有关详细信息，请参阅 BCDEdit /deletevalue。

要手动删除 bcdedit 条目，请完成以下步骤：

1. 在目标计算机上，以管理员身份打开命令提示符。
2. 例如，输入以下命令以删除主机 IP 地址的 BCDEdit 调试条目：

```
控制台

bcdedit -deletevalue {dbgsettings} hostip
```

删除 hostip 时，需要在调试器命令行上指定 *target=*。

3. 另一个示例是，使用以下命令删除端口条目：

```
控制台

bcdedit -deletevalue {dbgsettings} port
```

删除端口条目时，KDNET 使用默认的 ICANN 认证的调试器端口 5364。

设置 Hyper-V

如果打算在目标计算机上安装 Hyper-V 角色，请参阅设置虚拟机主机的网络调试。

有关调试 Hyper-V 虚拟机 (VM) 的信息，请参阅设置虚拟机的网络调试 - KDNET。

在运行具有外部网络连接的 VM 的 Hyper-V 主机上启用 KDNET

有时会出现导致 VM 中的网络停止工作的情况：

- 已在电脑上启用 Hyper-V，已创建外部网络交换机并指向计算机中的物理 NIC，且 VM 已配置为使用该外部交换机进行其网络连接。
- 然后，使用外部网络交换机指向的相同物理 NIC 在 Hyper-V 主机操作系统上启用 KDNET。 主机已重新启动。
- 使用之前配置的外部交换机的所有 VM 在重新启动后会丢失其网络连接。

发生此停止情况是专门设计的，因为 KDNET 独占控制了其配置使用的 NIC。 该操作系统不会加载该 NIC 的本机 NDIS 微型端口。 之后，外部网络交换机无法再与本机 NDIS 微型端口驱动程序通信并停止工作。 要解决这种情况，请执行下列操作：

1. 从 Hyper-V 管理器打开虚拟交换机管理器，然后选择现有的虚拟交换机。 从下拉菜单中选择外部网络 NIC，然后在“虚拟交换机管理器”对话框中选择“确定”，这样可将外部网络 NIC 更改为 Microsoft 内核调试网络适配器。

2. 更新虚拟交换机 NIC 后，关闭并重启 VM。

当 KDNET 调试关闭时，需要遵循相同的程序将外部交换机重新指向 NIC 的本机 NDIS 微型端口。 否则，在禁用调试后重启计算机时，VM 连接将丢失。

IPv6

在 Windows 版本 1809 中添加了 IPv6 支持。

要将 IPv6 与调试器配合使用，请完成以下步骤：

1. Ping <debughostname>，并记下输出行回复中报告的 IPv6 地址。 使用此 IPv6 地址代替 `x:y:z:p:d:q:r:n`。

2. 使用 BCDEdit 删除 dbgsettings 中的任何现有 IP 地址值。

   控制台

   ```
   bcdedit -deletevalue {dbgsettings} hostip
   ```

3. 设置主机的 IPv6 地址。 字符串 `hostipv6=s:t:u:v:w:x:y:z` 中不得有任何空格。 <YourPort> 是用于此目标计算机的网络端口号。 <YourKey> 是由四部分构成的安全密钥。 <b.d.f> 是要用于 KDNET 的 NIC 的总线设备功能位置编号。

   控制台

   ```
   bcdedit /dbgsettings net hostipv6:s:t:u:v:w:x:y:z port:<YourPort> key:
   <YourKey> busparams:<b.d.f>
   ```

4. 键入此命令以确认已正确设置 dbgsettings。

   控制台

   ```
   C:\> bcdedit /dbgsettings
   busparams               0.25.0
   key
   2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
   ```

```
debugtype               NET
hostipv6                  2001:db8:0:0:ff00:0:42:8329
port                    50010
dhcp                    Yes
The operation completed successfully.
```

5. 在主机上，使用以下命令启动调试器：

控制台

```
Windbg -k net:port=<yournetworkportnumber>,key=
<key_output_from_kdnet>,target=::<YourIPv6Address>
```

6. 当调试器在主机上运行并等待连接时，请重新启动目标计算机。

7. 调试器应在启动期间尽早连接到主机调试器。 可以看到 KDNET 正在使用 IPv6 连接，因为连接消息中报告的 IP 地址是 IPv6 地址，而不是 IPv4 地址。

说明

- 允许指定 `hostip` 的每个调试器 bcd 设置都有相应的 `hostipv6` 元素。 共有三种。

展开表

IPv4	IPv6	使用情况
hostip	hostipv6	针对启动和内核调试
targethostip	targethostipv6	特定于内核调试
hypervisorhostip	hypervisorhostipv6	针对 hyper-v 调试

- 如果为任一类型的调试设置 `hostipv6` 样式地址，则表示需要并将获取 IPv6。
- 如果为任一类型的调试设置 `hostip` 样式地址，则表示需要并将获取 IPv4。
- 目标只能执行 IPv4 或 IPv6，而不是同时执行两者。 所使用的 IP 协议的版本由目标计算机 dbgsettings 控制。 如果已设置 `hostip`，则目标使用 IPv4。 如果已设置 `hostipv6`，则目标使用 IPv6。
- 主机调试器通常自动选择使用 IPv4 或 IPv6。 默认情况下，调试器同时侦听 IPv4 套接字和 IPv6 套接字，并在其中一个套接字上自动连接到目标计算机。
- 如果在主机上要强制在调试器中使用 IPv6，但希望调试器侦听来自目标的连接，则可以将 `target=::` 添加到调试器命令行。 `::` 是 0 的 IPv6 地址。

- 如果在主机上要强制在调试器中使用 IPv4 调试，但希望调试器侦听来自目标的连接，则可以将 `target=0.0.0.0` 添加到调试器命令行。 `0.0.0.0` 是 0 的 IPv4 地址。
- 如果在调试器命令行上指定 `target=` 并使用计算机名称，调试器会将该计算机名称转换为 IPv4 地址和 IPv6 地址。 然后，调试器尝试在两者上进行连接。
- 在调试器命令行上指定 `target=` 并使用 IP 地址时，如果 IP 地址包含任何 : 字符，则调试器假定它是 IPv6 地址，并强制为该连接使用 IPv6。 如果 IP 地址包含任何 . 字符，则调试器假定它是 IPv4 地址，并强制为该连接使用 IPv4。
- 如果在目标上设置了 IPv6，并在调试器命令行上强制使用 IPv4，则不会获得连接。
- 如果在目标上设置了 IPv4，并在调试器命令行上强制使用 IPv6，则也不会获得连接。

另请参阅

- 自动设置 KDNET 网络内核调试
- Windows 10 中的网络内核调试支持的以太网 NIC
- Windows 8.1 中的网络内核调试支持的以太网 NIC

设置虚拟机的网络调试 - KDNET

项目 • 2024/03/08

本主题介绍如何配置与 Hyper-V 虚拟机 (VM) 的内核调试连接。

Hyper-V 虚拟机设置

要调试第 2 代 Hyper-V 虚拟机 (VM)，请完成以下步骤。

1. 创建已安装 Windows 的 VM

有关创建 VM 的信息，请参阅使用 Hyper-V 创建虚拟机。

2. 定义外部虚拟交换机

要与 VM 通信，可以使用虚拟外部网络交换机。 有关如何创建外部网络交换机的信息，请参阅创建虚拟网络。

配置外部网络交换机时，必须设置以下选项。

展开表

选项	值
连接类型	外部网络
允许管理操作系统共享此网络适配器	Enabled
VLAN ID	已禁用

3. 禁用安全启动

要允许 kdnet 实用工具更新 BCDEdit 启动设置，请按照以下步骤暂时禁用虚拟机上的安全启动。

1. 加载 Hyper-V 管理器并选择 VM 的属性。
2. 选择“安全性”设置。
3. 取消选中“启用安全启动”复选框。
4. 选择“确定”以保存设置。

完成调试并在目标 VM 上禁用内核调试后，可以重新启用安全启动。

4. 安装 Windows 调试工具

调试工具用于调试器和 kdnet 实用工具，必须安装。 有关如何下载和安装调试工具的信息，请参阅 Windows 调试工具。

设置虚拟机的网络调试 - KDNET

记录主机 IP 地址

要在与目标虚拟机相同的电脑上运行主机调试器，请执行以下步骤。

1. 在主计算机 OS 中，打开命令提示符窗口并输入 *IPConfig* 以显示 IP 配置。
2. 在命令输出中，找到配置为外部虚拟交换机的以太网适配器。

```
控制台

...

Ethernet adapter vEthernet (External Virtual Switch):

...

IPv4 Address. . . . . . . . . . . : <YourHostIPAddress>

...
```

> **提示**
>
> 可能有多个具有类似名称的适配器条目。 确保找到所创建的虚拟交换机。

3. 记录外部虚拟交换机的 IPv4 地址，该地址将用作调试的主机地址。
4. 要确认目标计算机与主计算机之间的连接，请在目标计算机上打开提升的命令提示符窗口并输入以下命令，其中 *YourHostIPAddress* 是主计算机的 IP 地址。

```
控制台

ping -4 <YourHostIPAddress>
```

设置 VM 目标计算机

按照以下步骤，使用 kdnet.exe 实用工具在目标电脑上自动配置调试器设置。

1. 找到 WDK *kdnet.exe* 和 *VerifiedNICList.xml* 文件。 默认情况下，它们位于此处。

```
控制台

C:\Program Files (x86)\Windows Kits\10\Debuggers\x64
```

> **① 备注**
>
> 这些说明假定目标电脑和主电脑都运行 64 位版本的 Windows。 如果情况并非如此，最好的办法是在主电脑上运行与目标电脑相同“位数”的工具。 例如，如果目标电脑运行的是 32 位 Windows，请在主电脑上运行 32 版本的调试器。 有关详细信息，请参阅**选择 32 位或 64 位调试工具**。

2. 要允许用于剪切和粘贴的长密钥，请启用增强的会话支持。 在 VM 窗口中的“视图”下拉菜单中，启用“增强会话”。

3. 在目标 VM 计算机上，创建 C:\KDNET 目录，并将 *kdnet.exe* 和 *VerifiedNICList.xml* 文件复制到该目录。

4. 在目标计算机上，以管理员身份打开“命令提示符”窗口。 输入此命令以验证目标计算机是否具有受支持的网络适配器。

   ```
   控制台

   C:\KDNET>kdnet

   Network debugging is supported on the following NICs:
   busparams=0.25.0, Intel(R) 82579LM Gigabit Network Connection, KDNET is
   running on this NIC.kdnet.exe
   ```

5. 键入此命令以设置主机系统的 IP 地址并生成唯一的连接密钥。 使用前面记录的主机系统的 IP 地址。 在 50000-50039 范围内，为每个使用的目标/主机对选择唯一的端口地址。 对于本示例，我们将选择 50005。

   ```
   控制台

   C:\>kdnet <YourIPAddress> <YourDebugPort>

   Enabling network debugging on Microsoft Hypervisor Virtual Machine.
   Key=3u8smyv477z20.2owh9gl90gbxx.3sfsihzgq7di4.nh8ugnmzb4l7

   To debug this vm, run the following command on your debugger host
   machine.
   windbg -k
   net:port=50005,key=3u8smyv477z20.2owh9gl90gbxx.3sfsihzgq7di4.nh8ugnmzb4
   l7
   ```

```
Then restart this VM by running shutdown -r -t 0 from this command
prompt.
```

6. 使用 CRTL+C 将提供的 Windbg 输出复制到命令缓冲区中。 这样做可避免尝试写下返回的长键值。

7. 配置调试器设置后，重新启用 BitLocker 和安全启动。

8. 由于具有“增强会话”支持的 VM 在停留在断点时可能会超时，因此请使用 VM 中的“查看”下拉菜单禁用“增强会话”支持。

9. 加载并运行调试器后，将重启 VM。 接下来将介绍此过程。

启动调试会话

1. 要连接到目标电脑，请使用 CTRL+V 在 OS 主命令窗口中粘贴之前复制的 kdnet 返回的 Windbg 字符串。

   控制台

   ```
   C:\Debuggers\windbg -k net:port=<YourDebugPort>,key=<YourKey>
   ```

首次尝试建立网络调试连接时，系统可能会提示你允许调试应用程序（WinDbg 或 KD）通过防火墙进行访问。 应通过选中**所有三种**网络类型的复选框来响应提示：域、专用和公共。

重启目标电脑

连接调试器后，重新启动目标计算机。 要强制 VM 完全重启，请从管理员的命令提示符处使用此命令。

控制台

```
shutdown -r -t 0
```

重启目标虚拟机后，主机 OS 中的调试器应连接。

连接到 VM 后，点击调试器上的中断，即可开始调试。

为支持调试连接，会添加一个内核调试网络适配器，并在 VM 重新启动后在网络属性中显示。

KDNET 虚拟机网络调试疑难解答

如果调试器未连接，请使用目标 VM 的 ping 命令验证连接。

控制台

```
C:\>Ping <HostComputerIPAddress>
```

有些事情不起作用，我不确定...

- 确保已让 WinDbg 穿过防火墙。
- 确认使用的是 BCDEdit 或 kdnet 生成的唯一密钥。

我的 VM 没有网络连接

- 从 Hyper-V 管理器打开虚拟交换机管理器，选择现有的虚拟交换机，并通过从下拉列表中选择外部网络 NIC 并将其更改为 Microsoft 内核调试网络适配器，然后在“虚拟交换机管理器”对话框中选择“确定”。 更新虚拟交换机 NIC 后，请确保关闭并重启 VM。

将 Hyper-V 角色添加到 Windows 电脑的顺序

如果目标计算机是虚拟机主机，则可以设置网络调试，并且仍具有虚拟机的网络访问权限。

假设要在以下情况下设置网络调试。

- 目标计算机只有一个网络接口卡。
- 打算在目标计算机上安装 Hyper-V 角色。
- 打算在目标计算机上创建一个或多个虚拟机。

最好的办法是在安装 Hyper-V 角色前在目标计算机上安装网络调试。 然后，虚拟机将有权访问网络。

如果决定在目标计算机上安装 Hyper-V 角色后设置网络调试，则必须更改虚拟机的网络设置，以便将其桥接到 Microsoft 内核网络调试适配器。 否则，虚拟机将无法访问网络。

另请参阅

使用虚拟 COM 端口手动设置虚拟机的内核模式调试

手动设置网络连接

使用虚拟 COM 端口手动设置虚拟机的内核模式调试

项目 • 2024/03/01

Windows 调试工具支持虚拟机的内核调试。 虚拟机可以位于调试器所在的同一物理计算机上，也可以位于连接到同一网络的其他计算机上。 本主题介绍如何通过 KDCOM 使用虚拟 COM 端口手动设置虚拟机调试。

建议使用 KDNET 虚拟网络，这样更快。 有关详细信息，请参阅 使用 KDNET 设置虚拟机的网络调试。

设置目标虚拟机

运行调试器的计算机称为*主机*，被调试的虚拟机称为 *目标虚拟机*。

> **ⓘ 重要**
>
> 在使用 BCDEdit 更改启动信息之前，可能需要在测试电脑上暂时挂起 Windows 安全功能，例如 BitLocker 和安全启动。 测试完成后重新启用这些安全功能，并在禁用安全功能时适当地管理测试电脑。

1. 在虚拟机中提升的“命令提示符”窗口输入以下命令。

   bcdedit /debug on

   bcdedit /dbgsettings serial debugport:*n*baudrate:115200

   其中 *n* 是虚拟机上的 COM 端口数。

2. 在虚拟机中，配置 COM 端口以映射到命名管道。 调试器将通过此管道进行连接。 有关如何创建此管道的详细信息，请参阅虚拟机的文档。

3. 在提升模式下启动调试器，例如从管理员命令提示符处启动。 当通过串行管道调试 VM 时，调试器必须以提升模式运行。 附加并运行调试器后，重新启动目标 VM。

使用 WinDbg 启动调试会话

在主机上以管理员身份打开 WinDbg。 当通过串行管道调试 VM 时，调试器必须以提升模式运行。 在“**文件**”菜单上，选择“**内核调试**”。 在“内核调试”对话框中，打开**COM**选项

卡。选中 **Pipe** 框，然后选中 **Reconnect** 框。 对于**波特率**，请输入 115200。 对于 **重置**，请输入 0。

如果调试器与虚拟机在同一台计算机上运行，请为**端口**输入以下内容。

\\.\pipe*PipeName*.

如果调试器与虚拟机在不同的计算机上运行，请为**端口**输入以下内容。

*****VMHost***\pipe***PipeName*

选择“确定”。

还可以在命令行中启动 WinDbg。 如果调试器与虚拟机在同一物理计算机上运行，请在“命令提示符”窗口中输入以下命令。

windbg -k com:pipe,port=\\.\pipe*PipeName***,resets=0,reconnect**

如果调试器与虚拟机在不同的物理计算机上运行，请在“命令提示符”窗口中输入以下命令。

windbg -k com:pipe,port=*VMHost***\pipe***PipeName***,resets=0,reconnect**

使用 KD启动调试会话

若要调试在调试器所在的同一物理计算机上运行的虚拟机，请在“命令提示符”窗口中输入以下命令。

kd -k com:pipe,port=\\.\pipe*PipeName***,resets=0,reconnect**

若要调试与调试器在不同的物理计算机上运行的虚拟机，请在“命令提示符”窗口中输入以下命令。

kd -k com:pipe,port=*VMHost***\pipe***PipeName***,resets=0,reconnect**

参数

VMHost
指定运行虚拟机的计算机名称。

PipeName
指定在虚拟机上创建的管道名称。

resets=0
指定在主机和目标同步时，可以向目标发送不限数量的重置数据包。 对 Microsoft 虚拟

电脑和其他管道丢弃多余字节的虚拟机使用 **resets=0** 参数。 如果 VMware 或其他虚拟机的管道不会丢弃所有多余字节，请勿使用此参数。

reconnect
使调试器在发生读/写失败时自动断开连接并重新连接管道。 此外，如果调试器在启动调试器时找不到命名管道，则 *reconnect* 参数会导致调试器等待名为 *PipeName* 的管道出现。 对虚拟电脑和其他虚拟机使用 *reconnect*，它们会在计算机重启时销毁并重新创建管道。 对于 VMware 或其他在计算机重启期间保留管道的虚拟机，请勿使用此参数。

有关其他命令行选项的详细信息，请参阅 **KD 命令行选项**或 WinDbg 命令行选项。

第 2 代虚拟机

默认情况下，第 2 代虚拟机中不显示COM 端口。 可以通过 PowerShell 或 WMI 添加 COM 端口。 若要在 Hyper-V 管理器控制台中显示 COM 端口，必须使用路径创建它们。

若要在第 2 代虚拟机上使用 COM 端口来启用内核调试，请执行以下步骤：

1. 输入以下 PowerShell 命令禁用安全启动：

   Set-VMFirmware –Vmname*VmName***–EnableSecureBoot Off**

   其中 *VmName* 是虚拟机的名称。

2. 输入以下 PowerShell 命令，将 COM 端口添加到虚拟机：

   Set-VMComPort –VMName*VmName***1 \\.\pipe***PipeName*

   例如，以下命令在虚拟机 TestVM 上配置第一个 COM 端口，以在本地计算机上连接到名为 TestPipe 的命名管道：

   Set-VMComPort –VMName TestVM 1 \\.\pipe\TestPipe

3. 附加并运行调试器后，停止并冷启动 VM 以激活 VM 中的 COM 端口。 除非至少有一个实际配置了管道名称并且无法进行热添加，否则模拟的 UARTS 无法进行调试。

4. 在完成更新配置更改后，重新启用安全启动。

有关第 2 代 VM 的详细信息，请参阅 Generation 2 Virtual Machine Overview。

备注

如果目标计算机已停止响应，目标计算机仍因早期内核调试操作而停止，或者使用了 **-b** 命令行选项，调试器会立即进入目标计算机。

否则，目标计算机将继续运行，直到调试器命令它中断。

排查防火墙和网络访问问题

调试器（WinDbg 或 KD）必须通过防火墙进行访问。 甚至网络适配器支持的虚拟串行端口也会出现这种情况。

如果 Windows 提示您在加载调试器时关闭防火墙，请选择所有三个框。

根据所用 VM 的具体信息，可能需要更改虚拟机的网络设置，以将它们桥接到 Microsoft 内核网络调试适配器。 否则，虚拟机将无法访问网络。

Windows 防火墙

可以使用控制面板允许通过 Windows 防火墙进行访问。 打开“控制面板”>“系统和安全”，然后选择“允许应用通过 Windows 防火墙”。 在应用程序列表中，找到 *Windows GUI 符号调试器*和*Windows 内核调试器*。 使用复选框允许这两个应用程序通过防火墙。 重启调试应用程序（WinDbg 或 KD）。

第三方 VM

VMWare如果通过使用 VMWare 设施（例如重置按钮）重新启动虚拟机，请退出 WinDbg，然后重启 WinDbg 以继续调试。 在虚拟机调试期间，VMWare 通常消耗 100% 的 CPU。

另请参阅

使用 KDNET 设置虚拟机的网络调试

手动设置内核模式调试

设置虚拟机主机的网络调试

手动设置一台计算机的本地内核调试

项目 • 2024/03/05

Windows 调试工具支持*本地内核调试*。 这是在一台计算机上进行的内核模式调试。 换句话说，调试器运行在与被调试计算机相同的计算机上。 通过本地调试，可以检查状态，但不能中断内核模式进程，否则会导致 OS 停止运行。

local bcdedit 选项在 Windows 8.0 和 Windows Server 2012 及更高版本中可用。

设置本地内核模式调试

> **ⓘ 重要**
>
> 在使用 bcdedit 更改启动信息之前，可能需要在测试电脑上暂时挂起 Windows 安全功能，例如 BitLocker 和安全启动。 完成调试并在本地计算机上禁用内核调试后，可以重新启用安全启动。

1. 以管理员身份打开命令提示符窗口。 输入 **bcdedit /debug on**
2. 如果计算机尚未配置为调试传输的目标，请输入 **bcdedit /dbgsettings local**
3. 重新启动计算机。

启动调试会话

使用 WinDbg

以管理员身份打开 WinDbg。 在“文件”菜单上，选择“内核调试”。 在“内核调试”对话框中，打开“本地”选项卡。选择“确定”。

还可以通过以管理员身份打开命令提示符窗口并输入以下命令来启动 WinDbg 会话：

windbg -kl

使用 KD

以管理员身份打开命令提示符窗口并输入以下命令：

kd -kl

另请参阅

本地内核模式调试

手动设置内核模式调试

设置 USB KDNET EEM 内核模式调试（KDNET-EEM-USB）

项目 • 2024/10/31

Windows 调试工具支持使用以太网仿真模式通过 USB 电缆进行内核调试（EEM），本主题介绍如何使用 kdnet.exe 实用工具设置 USB EEM。

运行调试器的计算机称为*主计算机*，正在调试的计算机称为*目标计算机*。

内核模式 USB EEM 设备要求

以下为必需项：

- 在目标计算机上，Synopsys USB 3.0 控制器连接到 USB 类型 C 端口。
- 在主计算机上，需要外部可访问的 USB 3.0 端口。
- Windows 10 2020 年 10 月更新 （20H2） 或更高版本

可以为以下项配置 KDNET-EEM-USB 传输：

- PCI 调试设备。 这些调试设备是使用 dbgsettings：： busparams=seg.bus.dev.fun 设置的。
- ACPI-DBG2 表调试设备。 这些调试设备是使用 dbgsettings：： busparams=1|2|3 设置的，其中 1|2|3 指向包含调试设备配置的 ACPI DBG2 表数组条目。

电缆要求

- 将主机类型 A 端口连接到目标 C 端口需要标准 USB 3.0 类型 C 到类型 A 电缆。

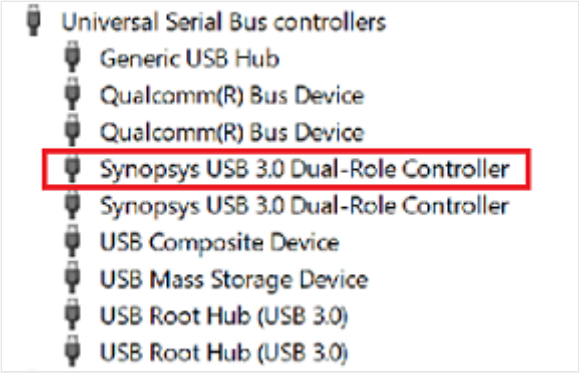
二进制传输文件

x64 的二进制文件kd_0C_8086.dll，ARM kd_8003_5143.dll用于支持 KDNET-EEM-USB 调试器传输。

确认目标上是否提供了受支持的 USB 控制器

在目标计算机上，启动设备管理器。

确认 *已列出 Synopsys USB 3.0 双角色控制器* 。



确定多个端口可用时的调试端口

确定支持调试的端口后，下一步是找到与该端口关联的物理 USB 连接器。

例如，在 Surface Pro X 上，使用两个 USB C 端口的下部用于 KDNET EEM 调试。

使用 kdnet.exe 确认设备支持情况并查看 busparams 值

Intel/AMD 64 设备

BCDEDIT 调试选项存储在启动配置数据（BCD）存储中。 有关详细信息，请参阅 BCDEdit /debug。

ARM 设备

ARM 设备使用 ACPI DBG2 表来配置调试器，其中 busparams 指向 DBG2 表项。 若要指定要使用的调试端口，请使用 busparm。 通常只使用第一个 busparam，并且它取决于

设备 0 或 1。 通常，设备不使用 busparams=0，因为通常为串行设备 COM 保留 0 DBG2 表项。 有关 ACPI DBG2 表的详细信息，请参阅Microsoft调试端口表 2（DBG2）。

使用kdnet.exe配置 KDNET EEM USB

使用 kdnet.exe 实用工具显示支持 KDNET-EEM-USB 传输调试的控制器的参数信息。

1. 确认主机和目标系统上已安装 Windows 调试工具。 有关下载和安装调试器工具的信息，请参阅适用于 Windows 的调试工具。

2. 找到kdnet.exe实用工具。 默认情况下，文件位于此处。

   ```
   C:\Program Files (x86)\Windows Kits\10\Debuggers\x64
   ```

3. 在目标计算机上，以管理员身份打开“命令提示符”窗口。 输入此命令，以验证目标计算机是否具有受支持的网络适配器，并查看 busparams 值。

   控制台

   ```
   C:\KDNET>kdnet.exe

   Network debugging is not supported on any of the NICs in this machine.
   KDNET supports NICs from Intel, Broadcom, Realtek, Atheros, Emulex,
   Mellanox
   and Cisco.

   Network debugging is supported on the following USB controllers:
   busparams=1, Device-mode USB controller with Vendor ID: 5143 (Default)
   busparams=2, Device-mode USB controller with Vendor ID: 5143
   busparams=3, Device-mode USB controller with Vendor ID: 5143
   busparams=4, Device-mode USB controller with Vendor ID: 5143

   This Microsoft hypervisor supports using KDNET in guest VMs.
   ```

4. 由于来自kdnet.exe的输出指示支持具有总线参数值为 1 的 USB 控制器可用，我们可以继续操作。

设置目标计算机

按照以下步骤，使用 kdnet.exe 实用工具在目标电脑上配置调试器设置。

> **ⓘ 重要**
>
> 在使用 bcdedit 更改启动信息之前，可能需要在测试电脑上暂时挂起 Windows 安全功能，例如 BitLocker 和安全启动。 使用 BCDEdit 更新启动信息后，可以重新启用

Bit Locker 和安全启动。 在禁用安全功能时，适当地管理测试电脑。

1. 使用下面所示的命令设置总线参数值、IP 地址和主机系统的端口，并生成了唯一的连接密钥。 169.254.255.255 IP 地址用于所有 USB EMM 连接。

2. 在建议的 50000-50039 范围内，为每个使用的目标/主计算机对选择唯一的端口地址。 示例中显示了 50005。

控制台

```
   C:\>kdnet.exe 169.254.255.255 50005

   Enabling network debugging on Intel(R) 82577LM Gigabit Network
Connection.
   Key=2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
```

3. 将返回的密钥复制到记事本 .txt 文件中。 在所示的示例中，生成的密钥值为：

   `2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p`

4. 使用 BCDEdit 命令检查参数是否按预期显示。 有关详细信息，请参阅 BCDEdit /dbgsettings

控制台

```
   C:\>bcdedit /dbgsettings

   busparams               1
   key
2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
   debugtype               NET
   hostip                  169.254.255.255
   port                    50005
   dhcp                    No
   The operation completed successfully.
```

禁用主计算机上的防火墙

在主机上，禁用调试器的防火墙。

将 WinDbg 连接到目标进行内核调试

在主计算机上打开 WinDbg。 在“文件”菜单上，选择“内核调试”。 在“内核调试”对话框中，打开“ **Net** ”选项卡。粘贴之前保存在记事本.txt文件中的端口号和密钥。 选择“确

定”。

还可以通过打开命令提示符窗口并输入以下命令来启动 WinDbg 会话，其中内容为上述所选择的端口以及上述由 kdnet.exe 返回的密钥。 粘贴之前保存在记事本.txt文件中的密钥。

```
windbg -k -d net:port=<YourDebugPort>,key=<YourKey>
```

重新启动目标计算机

连接调试器后，重新启动目标计算机。 重启电脑的一种方法是使用 `shutdown -r -t 0` 管理员命令提示符中的命令。

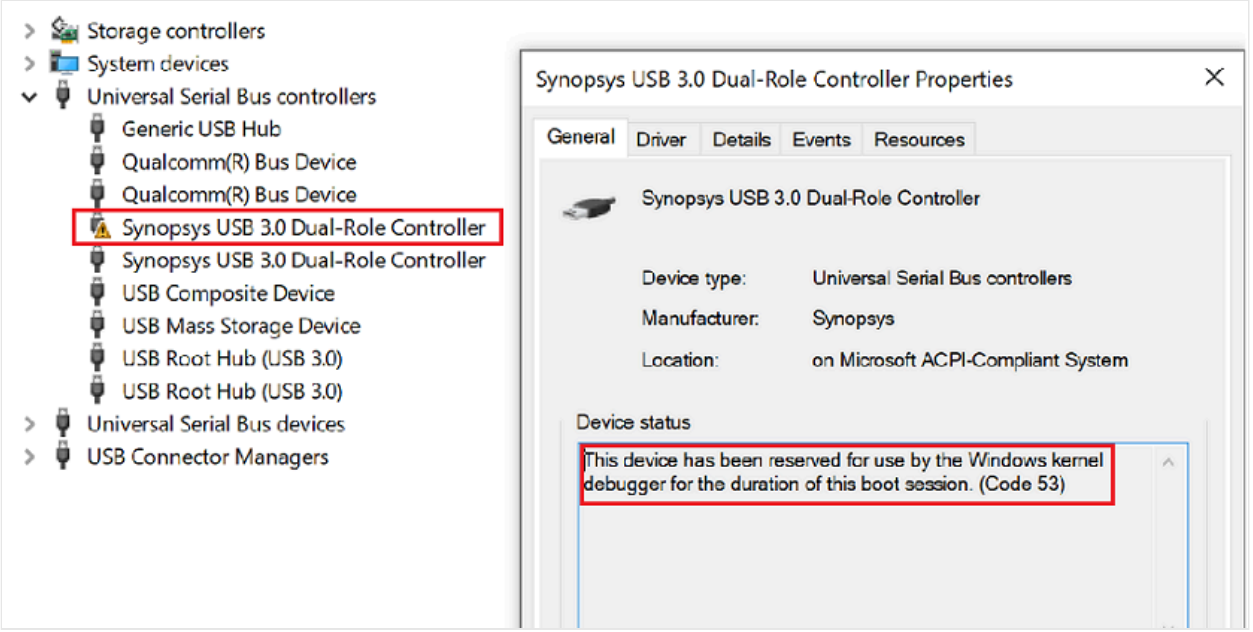
目标电脑重启后，调试器应自动连接。

故障排除

故障排除目标

确认 Windows 设备管理器中的网络适配器下存在 Windows KDNET-USB-EMM 网络适配器。

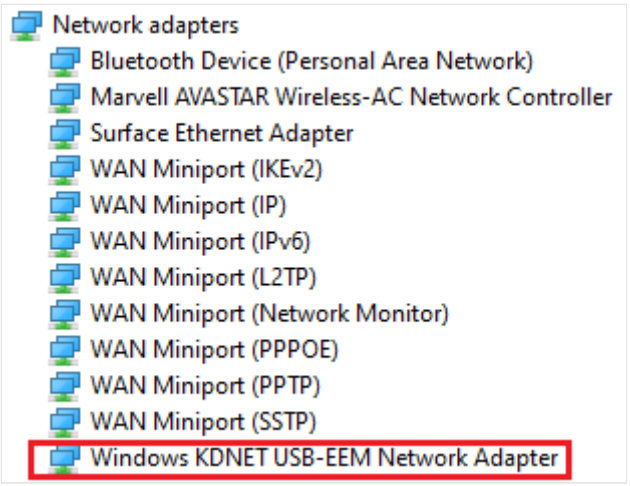
设备属性显示控制器何时保留供 Windows 内核调试器使用。



主机疑难解答

确认 Windows 设备管理器中的网络适配器下存在 Windows KDNET-USB-EMM 网络适配器。

在主机上，会显示使用 USB 类型 A 端口的 KDNET-EEM 连接。



Intel PCI - 连接在调试器控制台窗口中重试消息，无法闯入目标 - SkipPciProbeDebugDevice

如果在 KDNET 调试器控制台中遇到以下消息，则无法启动进入目标中的中断，或者遇到某些命令（例如 kdfiles）的问题，则可能是因为 KDNET 收到序列外 ping 数据包。

控制台

```
... Retry sending the same data packet for 128 times.

The transport connection between host kernel debugger and target Windows
seems lost.
please try resync with target, recycle the host debugger, or reboot the
target Windows.
```

此问题可能发生，因为pci.sys驱动程序错误地探测调试设备。 若要消除错误，请在管理员命令提示符下在 TARGET 设备上创建以下注册表项。

控制台

```
reg add HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\SERVICES\kdnet /v
SkipPciProbeDebugDevice /t REG_DWORD /d 1 /f
```

然后重启目标计算机。

控制台

```
shutdown /r /t 0
```

设备重新启动后，错误应消失，命令应按预期工作。

另请参阅

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

设置 USB 3.0 xHCI-DBC 内核模式调试 （KDUSB）

设置 USB KDNET 内核模式调试 （KDNET-USB）

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

设置 KDNET USB 内核模式调试（KDNET-USB）

项目 • 2024/10/31

Windows 调试工具支持通过 USB 3.0 电缆使用 KDNET 通过 USB 进行内核模式调试。 本文介绍如何配置此传输选项。

运行调试器的计算机称为*主计算机*，正在调试的计算机称为*目标计算机*。

通过 USB 3.0 电缆进行调试需要以下硬件：

- 在主计算机上，xHCI USB 3.0+ 主机控制器和 USB3 端口。
- 在目标计算机上，xHCI USB 3.0+ 主机控制器和支持调试的 USB3 端口（DBC）
- 目标计算机 USB 主机控制器必须支持 Intel xHCI 调试功能接口（DBC）。 有关详细信息，请参阅 Intel 网站上的 xHCI 规范。

二进制传输文件

kdstub.dll和kdnic.sys驱动程序用于支持 KDNET-USB 调试器传输。

电缆要求

- 橙色Microsoft USB 调试电缆，它是 A-A 交叉电缆，具有两个男性 A 型插入和无 Vbus 连接。 此电缆可供 DataPro - *USB 3.0 超速 A/A 调试电缆*等供应商使用。
- 将主机类型 A 端口连接到目标 C 端口需要标准 USB 3.0 类型 C 到类型 A 适配器。

为了简化故障排除，请直接在目标计算机和主计算机之间连接电缆，避免任何集线器或停靠站。

设置目标计算机

1. 在目标计算机上，找到并启动 *UsbView* 工具。 UsbView 工具包含在 Windows 调试工具中。 对于 x64 系统，UsbView 将位于 *C：\Program Files （x86）\Windows Kits\10\Tools\KitVersion\x64\usbview.exe*中。
2. 在 UsbView 中，找到所有 xHCI 主机控制器。
3. 在 UsbView 中，展开 xHCI 主机控制器的节点。 查找主机控制器上的端口支持调试的指示。

控制台

```
[Port1]

Is Port User Connectable:         yes
Is Port Debug Capable:            yes
Companion Port Number:            3
Companion Hub Symbolic Link Name: USB#ROOT_HUB30#5&32bab638&0&0#{...}
Protocols Supported:
 USB 1.1:                         no
 USB 2.0:                         no
 USB 3.0:                         yes
```

4. 记下要用于调试的 xHCI 控制器的总线、设备和函数编号。 UsbView 显示这些数字。 在以下示例中，总线号为 48，设备编号为 0，函数号为 0。

控制台

```
USB xHCI Compliant Host Controller
...
DriverKey: {36fc9e60-c465-11cf-8056-444553540000}\0020
...
Bus.Device.Function (in decimal): 48.0.0
```

5. 如果需要确认总线参数，请使用设备管理器。 在设备管理器中，找到要用于调试的 USB 控制器。 在“**常规**”**选项卡上的**“**位置**”下，将显示总线、设备和函数号。 *b*、 *d* 和 *f* 是 USB3 主机控制器的总线、设备和函数号。 总线、设备和功能编号必须采用十进制格式。

6. 还可以使用kdnet.exe来帮助收集有关 USB 控制器的信息。

控制台

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64>kdnet

Network debugging is supported on the following USB controllers:
busparams=0.20.0, Intel(R) USB 3.0 eXtensible Host Controller - 1.0
(Microsoft)

This Microsoft hypervisor supports using KDNET in guest VMs.
```

7. 确定支持调试的 xHCI 控制器后，下一步是找到与 xHCI 控制器上的端口关联的物理 USB 连接器。 若要查找物理连接器，请将任何 USB 3.0 设备插入目标计算机上的任何 USB 连接器。 刷新 UsbView 以查看设备所在的位置。 如果 UsbView 显示连接到所选 xHCI 主机控制器的设备，则找到可用于 USB 3.0 调试的物理 USB 连接器。

> **ⓘ 重要**
>
> 在使用 `bcdedit` 或kdnet.exe更改启动信息之前，可能需要暂时暂停测试电脑上的 Windows 安全功能，例如 BitLocker 和安全启动。 在已禁用这些安全功能的情况下，在测试完成后重新启用这些安全功能，并适当地管理测试 PC。

8. 在建议的 50000-50039 范围内，为每个使用的目标/主机对选择唯 `<port address>` 一。 以下示例显示了 50005。

9. 在与 CPU 类型匹配的 WDK 调试器目录中找到KDNet.exe实用工具，例如 x64。

10. 在目标计算机上，以管理员身份打开命令提示符窗口，并输入此命令以使用选项启用内核调试 `-k` 。 -w、-b 和 -h 将为 winload、bootmgr 和虚拟机监控程序系统应用程序启用内核调试。有关 WinDbg 选项的详细信息，请参阅 WinDbg - 命令行启动选项。

    控制台

    ```
    kdnet.exe 169.254.255.255 50005 -k
    ```

    - **169.254.255.255** 不可路由的本地链路静态 IP 地址必须用于通过 USB3 的 KDNET。
    - kdnet.exe将返回 WinDbg 用于连接到目标设备的密钥 `w.x.y.z` 。

    若要使用特定的 USB 端口，请使用 *-busparams* 参数。

    控制台

    ```
    kdnet.exe -busparams 0.13.0 169.254.255.255 50005 -k
    ```

    建议使用 KDNET 实用工具。 此工具设置使用 bcdedit 设置更复杂的选项，以及检查和设置支持 PCI 和电源管理的注册表值。

    控制台

    ```
    bcdedit /dbgsettings NET hostip:169.254.255.255 port:50001 key:1.2.3.4
    busparams:0.20.0 noDhcp
    ```

禁用电源管理

在某些情况下，电源转换可能会干扰 USB 3.0 的调试。 若要避免这些问题，请为 xHCI 主机控制器及其根中心禁用选择性挂起，用于调试。

1. 在设备管理器中，导航到 xHCI 主机控制器的节点。 右键单击节点，然后选择“ **属性**”。 **如果有电源管理**选项卡，请打开该选项卡，并清除“**允许计算机关闭此设备以保存电源**”复选框。

2. 在设备管理器中，导航到 xHCI 主机控制器根中心的节点。 右键单击节点，然后选择“ **属性**”。 **如果有电源管理**选项卡，请打开该选项卡，并清除“**允许计算机关闭此设备以保存电源**”复选框。

使用 xHCI 主机控制器进行调试后，请为 xHCI 主机控制器重新启用选择性挂起。

使用 WinDbg 启动调试会话

1. 将 USB 3.0 调试电缆连接到你选择用于在主机和目标计算机上调试的已识别 USB 3.0 端口。

2. 在主计算机上，打开与在主计算机上运行的 Windows 的位数匹配的 WinDbg 版本（以管理员身份）。 例如，如果主计算机运行的是 64 位版本的 Windows，请以管理员身份打开 64 位版本的 WinDbg。

3. 在 “**文件** ”菜单上，选择“ **附加到内核**”。 在“内核调试”对话框中，打开“ **Net** ”选项卡。输入以下信息，然后单击“ **确定**”。

    - 每个 `<port address>` 目标/主机对都是唯一的，在运行kdnet.exe时，建议的范围为 50000-50039。

    - `<session key>`运行kdnet.exe时生成的 w.x.y.z 及其值显示在命令输出中。

使用 WinDbg 的命令行会话

还可以通过在命令提示符窗口中输入此命令来启动与 WinDbg 的会话。

控制台

```
Windbg /k NET:port=<port address>,key=<session key>
```

重新启动目标计算机

加载调试器并准备好运行后，重新启动目标计算机。 重启电脑的一种方法是使用 `shutdown -r -t 0` 管理员命令提示符中的命令。

目标电脑重启后，调试器应自动连接。

故障排除

USB 设备无法识别

如果在插入调试电缆时无法识别文本 USB 设备的主机上显示 Windows 通知，则可能命中已知的 USB 3.1 到 3.1 兼容性问题。 当调试电缆连接到主机上的 USB 3.1 控制器和目标上的 Intel（Ice Lake 或 Tiger Lake）3.1 USB 控制器时，此问题会影响调试配置。

有关详细信息和处理器模型列表，请参阅 Ice Lake（微控制器）和老虎湖（微控制器）。 若要查找目标计算机的处理器模型，请打开“设置”应用，然后转到“**系统**” **处理器**在设备规范**下**列出。

若要验证此问题，请打开设备管理器，并在通用串行总线控制器下查找 **USB 调试连接设备。** 如果找不到此设备，请在“其他设备”下检查未知**设备。** 右键单击设备以打开其属性页。 设备状态文本框将显示 Windows 已停止此设备的文本，**因为它报告了问题（代码 43），USB 设备返回了无效的 USB BOS 描述符。**

若要解决此问题，请从管理员命令提示符运行以下命令，对注册表进行更改：

控制台

```
reg add
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\usbflags\349500E00000 /v
SkipBOSDescriptorQuery /t REG_DWORD /d 1 /f
reg add
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\usbflags\045E06560000 /v
SkipBOSDescriptorQuery /t REG_DWORD /d 1 /f
```

直接编辑注册表时请谨慎，因为不正确的更改可能导致系统不稳定。

连接在调试器控制台窗口中重试消息，无法进入目标 - SkipPciProbeDebugDevice

如果在 KDNET 调试器控制台中遇到以下消息，则无法启动进入目标的中断，或者遇到某些命令（例如 kdfiles）的问题，则可能是因为 KDNET 接收了序列外 ping 数据包。

控制台

```
... Retry sending the same data packet for 128 times.

The transport connection between host kernel debugger and target Windows
seems lost.
please try resync with target, recycle the host debugger, or reboot the
target Windows.
```

此问题可能发生，因为pci.sys驱动程序正在探测调试设备。 若要消除这些错误消息，请在管理员命令提示符下在 TARGET 设备上创建以下注册表项。

如果初始 KD 传输在启动时无法连接，则此设置还可以允许调试器进行连接，原因有一些其他原因，例如在启动时无法配置调试设备。

控制台

```
reg add HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\SERVICES\kdnet /v
SkipPciProbeDebugDevice /t REG_DWORD /d 1 /f
```

然后重启目标计算机。

控制台

```
shutdown /r /t 0
```

设备重新启动后，错误应消失，命令应按预期工作。

用于增强性能和可靠性的NO_KDNIC设置

如果在目标电脑上安装了以太网 NIC，可以通过设置 `NO_KDNIC` 选项来实现额外的可靠性和性能改进。

控制台

```
bcdedit /set {current} loadoptions NO_KDNIC
```

添加 `NO_KDNIC` 是可选的，仅当目标具有额外的 NIC、Wi-Fi 或 USB 端口来连接 USB 以太网适配器以提供对 Windows 的网络访问时，才能使用。

添加 `NO_KDNIC` 将阻止kdnic.sys驱动程序（基于微型端口计时器的驱动程序）在 KDNET 上运行，这意味着不会通过 KDNET 传输路由 Windows TCP/IP 流量。 然后，KDNET 传输只能用于在目标 KDNET 和主机调试器之间路由与调试相关的数据包。

这可以帮助解决kdnic.sys驱动程序在 kdnet 上运行时可能会受到影响的网络性能。 在这种情况下，目标永远不会进入睡眠状态，防止电源滴水测试，或者通过 RDP 访问目标时会发生延迟。 这是因为 KDNET 接口需要在运行kdnic.sys时路由调试器数据包和 Windows TCP/IP 网络数据包。

另请参阅

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

手动设置内核模式调试

设置 USB 3.0 xHCI-DBC 内核模式调试 （KDUSB）

设置 USB KDNET EEM 内核模式调试 （KDNET-EEM-USB）

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

设置 USB 3.0 xHCI 内核模式调试（KDUSB xHCI-DBC USB 3.0）

项目 • 2024/10/31

Windows 调试工具支持通过 USB 3.0 电缆进行内核模式调试。 本文介绍如何手动设置 USB 3.0 调试。

运行调试器的计算机称为*主计算机*，正在调试的计算机称为*目标计算机*。

通过 USB 3.0 电缆进行调试需要以下硬件：

- 在主计算机上，xHCI （USB 3.0） 主机控制器
- 在目标计算机上，支持调试的 xHCI （USB 3.0） 主机控制器
- 目标计算机 USB 主机控制器必须支持 Intel xHCI-Debug 功能接口（DBC）。 有关详细信息，请参阅 Intel 网站上的 xHCI 规范。

电缆要求

- 橙色Microsoft USB 调试电缆，它是 A-A 交叉电缆，具有两个男性 A 型插入和无 Vbus 连接。 此电缆可供 DataPro - *USB 3.0 超速 A/A 调试电缆*等供应商使用。

为了简化故障排除，请直接在目标计算机和主计算机之间连接电缆，避免任何集线器或停靠站。

二进制传输文件

kdstub.dll用于支持 KDUSB xHCI-DBC USB 3.0 调试器传输。

设置目标计算机

1. 在目标计算机上，启动 *UsbView* 工具。 UsbView 工具包含在 Windows 调试工具中。 对于 x64 系统，UsbView 将位于 *C：\Program Files （x86）\Windows Kits\10\Tools\kitversion\x64\usbview.exe*中。

2. 在 UsbView 中，找到所有 xHCI 主机控制器。

3. 在 UsbView 中，展开 xHCI 主机控制器的节点。 查找主机控制器上的端口支持调试的指示。

   ```
   控制台
   ```

```
[Port1]

Is Port User Connectable:         yes
Is Port Debug Capable:            yes
Companion Port Number:            3
Companion Hub Symbolic Link Name: USB#ROOT_HUB30#5&32bab638&0&0#{...}
Protocols Supported:
 USB 1.1:                         no
 USB 2.0:                         no
 USB 3.0:                         yes
```

4. 记下要用于调试的 xHCI 控制器的总线、设备和函数编号。 UsbView 显示这些数字。 在以下示例中，总线号为 48，设备编号为 0，函数号为 0。

控制台

```
USB xHCI Compliant Host Controller
...
DriverKey: {36fc9e60-c465-11cf-8056-444553540000}\0020
...
Bus.Device.Function (in decimal): 48.0.0
```

5. 确定支持调试的 xHCI 控制器后，下一步是找到与 xHCI 控制器上的端口关联的物理 USB 连接器。 若要查找物理连接器，请将任何 USB 3.0 设备插入目标计算机上的任何 USB 连接器。 刷新 UsbView 以查看设备所在的位置。 如果 UsbView 显示连接到所选 xHCI 主机控制器的设备，则找到可用于 USB 3.0 调试的物理 USB 连接器。

> **ⓘ 重要**
>
> 在用于 `bcdedit` 更改启动信息之前，可能需要暂时暂停测试电脑上的 Windows 安全功能，例如 BitLocker 和安全启动。 在已禁用这些安全功能的情况下，在测试完成后重新启用这些安全功能，并适当地管理测试 PC。

6. 在目标计算机上，以管理员身份打开命令提示符窗口，并输入以下命令：

控制台

```
bcdedit /debug on
bcdedit /dbgsettings usb targetname:<TargetName>
```

TargetName 是为目标计算机创建的名称。 请注意， *TargetName* 不必是目标计算机的正式名称;只要它满足以下限制，就可以创建的任何字符串：

- 字符串不得在 TargetName *中的任何位置包含大写或小写组合中的*"debug"。 例如，如果在目标名称中的任何位置使用"DeBuG"或"DEBUG"，则调试无法正常工作。
- 字符串中唯一的字符是连字符（-）、下划线（_）、数字 0 到 9，以及字母 A 到 Z（大写或小写）。
- 字符串的最大长度为 24 个字符。

7. 在设备管理器中，找到要用于调试的 USB 控制器。 在"**常规**"**选项卡上的**"**位置**"下，将显示总线、设备和函数号。 输入以下命令：

   控制台

   ```
   bcdedit /set "{dbgsettings}" busparams <b.d.f>
   ```

   B、 *d* 和 *f* 是 USB 主机控制器的总线、设备和函数号。 总线、设备和功能编号必须采用十进制格式。

   示例：

   控制台

   ```
   bcdedit /set "{dbgsettings}" busparams 48.0.0
   ```

8. 重新启动目标计算机。

禁用电源管理

在某些情况下，电源转换可能会干扰 USB 3.0 的调试。 若要避免这些问题，请为 xHCI 主机控制器及其根中心禁用选择性挂起，用于调试。

1. 在设备管理器中，导航到 xHCI 主机控制器的节点。 右键单击节点，然后选择" **属性**"。 **如果有电源管理**选项卡，请打开该选项卡，并清除"**允许计算机关闭此设备以保存电源**"复选框。

2. 在设备管理器中，导航到 xHCI 主机控制器根中心的节点。 右键单击节点，然后选择" **属性**"。 **如果有电源管理**选项卡，请打开该选项卡，并清除"**允许计算机关闭此设备以保存电源**"复选框。

使用 xHCI 主机控制器进行调试后，请为 xHCI 主机控制器重新启用选择性挂起。

首次启动调试会话

1. 将 USB 3.0 调试电缆连接到你选择用于在主机和目标计算机上调试的 USB 3.0 端口。
2. 确定在主计算机上运行的 Windows 的位数（32 位或 64 位）。
3. 在主计算机上，打开与在主计算机上运行的 Windows 的位数匹配的 WinDbg 版本（以管理员身份）。 例如，如果主计算机运行的是 64 位版本的 Windows，请以管理员身份打开 64 位版本的 WinDbg。
4. 在“文件”菜单上，选择“内核调试”。 在“内核调试”对话框中，打开 “**USB** ”选项卡。输入在设置目标计算机时创建的目标名称。 单击“确定”。

此时，USB 调试驱动程序安装在主计算机上，这就是为什么将 WinDbg 的位数与 Windows 的位性相匹配非常重要。 安装 USB 调试驱动程序后，可以使用 32 位或 64 位版本的 WinDbg 进行后续调试会话。

启动调试会话

使用 WinDbg

在主计算机上打开 WinDbg。 在“文件”菜单上，选择“内核调试”。 在“内核调试”对话框中，打开 “**USB** ”选项卡。输入在设置目标计算机时创建的目标名称。 选择“确定”。

还可以通过在命令提示符窗口中输入以下命令来启动与 WinDbg 的会话，其中 *TargetName* 是设置目标计算机时创建的目标名称：

控制台

```
windbg /k usb:targetname=<TargetName>
```

使用 KD

在主计算机上，打开命令提示符窗口并输入以下命令，其中 *TargetName* 是设置目标计算机时创建的目标名称：

控制台

```
kd /k usb:targetname=<TargetName>
```

重新启动目标计算机

连接调试器后，重新启动目标计算机。 重启电脑的一种方法是使用 `shutdown -r -t 0` 管理员命令提示符中的命令。

目标电脑重启后，调试器应自动连接。

故障排除

USB 设备无法识别

如果在插入调试电缆时无法识别文本 USB 设备的主机上显示 Windows 通知，则可能命中已知的 USB 3.1 到 3.1 兼容性问题。 当调试电缆连接到主机上的 USB 3.1 控制器和目标上的 Intel（Ice Lake 或 Tiger Lake）3.1 USB 控制器时，此问题会影响调试配置。

有关详细信息和处理器模型列表，请参阅 Ice Lake（微控制器）和老虎湖（微控制器）。 若要查找目标计算机的处理器模型，请打开“设置”应用，然后转到“**系统**” **处理器**在设备规范**下**列出。

若要验证此问题，请打开设备管理器，并在通用串行总线控制器下查找 **USB 调试连接设备。** 如果找不到此设备，请在“其他设备”下检查未知**设备。** 右键单击设备以打开其属性页。 设备状态文本框将显示 Windows 已停止此设备的文本，**因为它报告了问题（代码 43），USB 设备返回了无效的 USB BOS 描述符。**

若要解决此问题，请从管理员命令提示符运行以下命令，对注册表进行更改：

控制台

```
reg add
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\usbflags\349500E00000 /v
SkipBOSDescriptorQuery /t REG_DWORD /d 1 /f
reg add
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\usbflags\045E06560000 /v
SkipBOSDescriptorQuery /t REG_DWORD /d 1 /f
```

然后，删除并重新插入调试电缆。

连接在调试器控制台窗口中重试消息，无法进入目标 - SkipPciProbeDebugDevice

如果在 KDNET 调试器控制台中遇到以下消息，则无法启动进入目标中的中断，或者遇到某些命令（例如 kdfiles）的问题，则可能是因为 KDNET 收到序列外 ping 数据包。

控制台

```
... Retry sending the same data packet for 128 times.

The transport connection between host kernel debugger and target Windows
seems lost.
```

```
please try resync with target, recycle the host debugger, or reboot the
target Windows.
```

此问题可能发生，因为pci.sys驱动程序错误地探测调试设备。 若要消除错误，请在管理员命令提示符下在 TARGET 设备上创建以下注册表项。

控制台

```
reg add HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\SERVICES\kdnet /v
SkipPciProbeDebugDevice /t REG_DWORD /d 1 /f
```

然后重启目标计算机。

控制台

```
shutdown /r /t 0
```

设备重新启动后，错误应消失，命令应按预期工作。

另请参阅

手动设置内核模式调试

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

设置 USB KDNET EEM 内核模式调试 （KDNET-EEM-USB）

设置 USB KDNET 内核模式调试 （KDNET-USB）

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用 KDNET 设置 2PF 内核模式调试

项目 • 2024/03/08

通过对 PCI 配置空间进行分区，Windows 调试工具支持使用多个物理功能 (PF) 在受支持的 NIC 上通过网络电缆进行内核调试。

通过 2PF 调试，每个 PF 都可以连接到单个网络端口，因此它允许内核调试功能连接到一个 PF，而标准网络堆栈与另一个 PF 通信。 因此，KDNIC 不需要通过 KDNET 路由 Windows 网络流量，KDNET 将只负责路由主计算机内核调试器流量。 这导致性能显著提高。

本主题介绍如何使用 kdnet.exe 实用工具设置 2PF 调试。

鼓励网络卡供应商启用对此功能的支持。 有关详细信息，请参阅调试器 2PF KDNET 微型端口网络驱动程序支持。

两个驱动程序将在分区的 PCI 配置空间上运行

- Windows 收件箱驱动程序将在 bus.dev.**fun0.0** PCI 位置用完主网络端口。
- KDNET-Ext. 模块将在 bus.dev **fun0.1** 用完添加的 PF，此技术可确保 Windows 收件箱 NIC 驱动程序不会受到与 KDNET 共享 NIC 的影响。

运行调试器的计算机称为*主计算机*，正在调试的计算机称为*目标计算机*。

内核模式 2PF 设备要求

以下为必需项：

- 在目标计算机上，一个受支持的 2PF 网卡。
- 在主计算机上，一个网卡。
- 目标与主计算机之间的网络连接。
- Windows 10 版本 21313 及更高版本。

受支持的 2PF 网卡

NVIDIA Mellanox 和 Cisco 等供应商提供支持 2PF 网络调试的 NIC。 请咨询网卡供应商，查看支持哪些网卡模型。 请注意，某些供应商在共享相同 PnP ID 的网卡子集上支持 2PF。

使用 kdnet.exe 确认设备支持情况并查看 busparams 值

使用 kdnet.exe 实用工具显示支持 KDNET 2PF 传输调试的控制器的参数信息。

1. 确认主计算机系统上已安装 Windows 调试工具。 有关下载和安装调试器工具的信息，请参阅适用于 Windows 的调试工具。

2. 找到 kdnet.exe 和 VerifiedNICList.xml 文件。 默认情况下，它们位于此处。

   `C:\Program Files (x86)\Windows Kits\10\Debuggers\x64`

3. 在主计算机上，将这两个文件复制到网络共享或 U 盘，以便在目标计算机上可用。

4. 在目标计算机上，创建 `C:\KDNET` 目录，并将 kdnet.exe 和 VerifiedNICList.xml 文件复制到该目录。

5. 在目标计算机上，以管理员身份打开“命令提示符”窗口。 输入此命令，以验证目标计算机是否具有受支持的网络适配器，并查看 busparams 值。

   dbgcmd

   ```
   C:\KDNET>kdnet.exe

   Network debugging is supported on the following NICs:
   busparams=141.0.0, Mellanox ConnectX-4 Lx Ethernet Adapter #2, Plugged
   in, Primary function, multiple physical functions are supported.

   Network debugging is supported on the following USB controllers:
   busparams=128.15.0, Standard USB 3.0 eXtensible Host Controller - 1.0
   (Microsoft)
   busparams=0.15.0, Standard USB 3.0 eXtensible Host Controller - 1.0
   (Microsoft)
   busparams=128.15.1, Standard USB 3.0 eXtensible Host Controller - 1.0
   (Microsoft)
   busparams=0.15.1, Standard USB 3.0 eXtensible Host Controller - 1.0
   (Microsoft)
   ```

   由于上面所示的输出不包括 *“KDNET 正在此 NIC 上运行”*，这表示未在任何适配器上启用传统的 KDNET 调试。

   如果 NIC **不**支持多个 PF 功能，则显示的信息将会省略（为空白）*“支持多个物理功能”*PF 状态通知。

   如果 NIC 支持多个 PF，则实际显示的信息将取决于网络端口的组合（根端口/PF 添加的端口），以及与 NIC 物理端口的连接/断开连接状态。

下表汇总了主 NIC 的不同 PF 通知。

展开表

NIC 适配器 bus.dev.fun 对应于	电缆状态	PF 状态
原始（主要）	已连接的 PF 电缆	主要函数，已启用多个物理函数
原始（主要）	断开连接的 PF 电缆	主要函数，支持多个物理函数

下表汇总了辅助 NIC 的不同 PF 通知。

展开表

NIC 适配器 bus.dev.fun 对应于	电缆状态	PF 状态
新（辅助）PF 端口	Kdnet 正在运行	次要函数
新（辅助）PF 端口	电缆断开连接或未知状态	主要函数，启用了多个物理函数，但不使用次要函数

6. 如果 kdnet.exe 的输出指示受支持的 NIC 控制器可用，则可以继续操作。

为 2PF 设置目标计算机

按照以下步骤，使用 kdnet.exe 实用工具在 2PF 的目标电脑上配置调试器设置。

> **重要**
>
> 在使用 bcdedit 更改启动信息之前，可能需要在测试电脑上暂时挂起 Windows 安全功能，例如 BitLocker 和安全启动。 使用 BCDEdit 更新启动信息后，可以重新启用 Bit Locker 和安全启动。 在禁用安全功能时，适当地管理测试电脑。

此过程将向 NIC 添加新的物理函数 (PF)，由 `bus.device.function` 指定。 新 PF 只能由 KDNET 使用，因为 Windows 收件箱驱动程序设置为不在添加的次要 PF 上运行。 按照以下步骤添加将由调试设备使用的新 PF。

在添加新的物理函数之前，确认调试已禁用

1. 在 NIC 上添加新 PF 之前，使用 BCDEdit 命令确认已在目标上禁用 KD。 这需要确保加载标准供应商 NIC 驱动程序，以便可用于添加新 PF。

控制台

```
C:\> bcdedit /enum
...

debug           No
```

或者，使用不带参数的 kdnet.exe 来查看是否启用了调试。 以下输出显示了在一个 NIC 上启用了调试的系统上运行的 KDNET。 这是性能较低的旧版设置。

控制台

```
c:\Debuggers>kdnet

Network debugging is supported on the following NICs:
busparams=141.0.0, Mellanox ConnectX-4 Lx Ethernet Adapter #2, KDNET is
running on this NIC.

Network debugging is supported on the following USB controllers:
busparams=128.15.0, Standard USB 3.0 eXtensible Host Controller - 1.0
(Microsoft)
busparams=0.15.0, Standard USB 3.0 eXtensible Host Controller - 1.0
(Microsoft)
busparams=128.15.1, Standard USB 3.0 eXtensible Host Controller - 1.0
(Microsoft)
busparams=0.15.1, Standard USB 3.0 eXtensible Host Controller - 1.0
(Microsoft)
```

2. 如果调试值设置为“是”，请使用设置命令禁用调试。

控制台

```
C:\> bcdedit.exe /debug off
C:\> bcdedit.exe /set {default} bootdebug off
C:\> bcdedit.exe /set {bootmgr} bootdebug off
```

3. 使用管理员命令提示符中的 `shutdown -r -t 0` 命令重新启动。

目标电脑重启并禁用调试后，可以添加新的物理函数。

添加新的物理函数

1. 打开提升的命令提示符，并运行以下命令以添加第二个 PF。 所有值都使用十进制值提供。

控制台

```
C:\KDNET> kdnet -addpf 141.0.0 198.51.100.1 50001

Succeeded adding a Pci PF on :141.0.1. Please power off or reboot the
machine.

Enabling network debugging on Mellanox ConnectX-4 Lx Ethernet Adapter #2.
Manage-bde.exe not present.  Bitlocker presumed disabled.

To debug this machine, run the following command on your debugger host
machine.
windbg -k
net:port=50001,key=2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p

Then reboot this machine by running shutdown -r -t 0 from this command
prompt.
```

`bus.dev.fun` 是支持多个 PF 功能的 NIC 适配器的 PCI 位置端口，因此新 PF 将添加/附加到此网络设备。

`-addpf` 选项可在添加的 PF 端口上通过 KDNET 传输自动启用内核调试。

`[host name/host ip address]` 是主计算机的 TCP/IP 地址。 使用主计算机上的 `ipconfig` 命令确定此地址。

`[port number]` 是 TCP/IP 端口号。 可以选择从 49152 到 65535 的任何端口号。 建议的范围介于 50000 和 50039 之间。 选择的端口将由在主计算机上运行的调试器打开以独占访问权限。 在建议的 50000-50039 范围内，为每个使用的目标/主计算机对选择唯一的端口地址。 示例中显示了 50005。

请注意，-addpf 还会将 `NO_KDNIC` 属性添加到操作系统安装 {default} loadoptions。 这是因为不再需要 KDNIC 在 KDNET 上运行。

将 loadoptions = NO_KDNIC 添加到 {default} OS 标记，以确保 kdnic.sys 不会用完新添加的 pf (141.0.1)

使用 bcdedit 命令确认已设置 NO_KDNIC。

控制台

```
C:\KDNET> bcdedit /enum {default}

Windows Boot Loader
-------------------
identifier              {current}
device                  partition=C:
path                    \Windows\system32\winload.efi
description             Windows Server
locale                  en-US
```

```
loadoptions             NO_KDNIC
inherit                 {bootloadersettings}
recoverysequence        {c23c4005d-12ae-11eb-9399-ac9840c152e7}
displaymessageoverride  Recovery
recoveryenabled         Yes
bootdebug               No
testsigning             Yes
isolatedcontext         Yes
flightsigning           Yes
allowedinmemorysettings 0x15000075
osdevice                partition=C:
systemroot              \Windows
resumeobject            {c23c4005d-12ae-11eb-9399-ac9840c152e7}
nx                      OptOut
debug                   Yes
hypervisordebug         No

2. Run the bcdedit /enum command to display the generated key.

```console
C:\KDNET> bcdedit /dbgsettings
busparams               141.0.1
key
2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
debugtype               NET
hostip                  198.51.100.1
port                    50001
dhcp                    Yes
The operation completed successfully.
```

3. 将返回的密钥复制到记事本 .txt 文件中。 在所示的示例中，生成的密钥值为：

   `2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p`

4. （可选）使用 kdnet.exe 来确认已启用多个物理函数。

控制台

```
C:\KDNET> kdnet.exe

Network debugging is supported on the following NICs:
busparams=141.0.0, Mellanox ConnectX-4 Lx Ethernet Adapter #2, Plugged in,
Primary function, multiple physical functions are enabled.
busparams=141.0.1, Mellanox ConnectX-4 Lx Ethernet Adapter, KDNET is running
on this NIC, Secondary function.

Network debugging is supported on the following USB controllers:
busparams=128.15.0, Standard USB 3.0 eXtensible Host Controller - 1.0
(Microsoft)
busparams=0.15.0, Standard USB 3.0 eXtensible Host Controller - 1.0
(Microsoft)
busparams=128.15.1, Standard USB 3.0 eXtensible Host Controller - 1.0
```
```

```
(Microsoft)
busparams=0.15.1, Standard USB 3.0 eXtensible Host Controller - 1.0
(Microsoft)
```

禁用主计算机上的防火墙

在主计算机上，禁用调试器端口的防火墙。

将 WinDbg 连接到目标进行内核调试

在主计算机上打开 WinDbg。 在“文件”菜单上，选择“内核调试”。 在“内核调试”对话框中，打开 **Net** 选项卡。粘贴以前保存在记事本 .txt 文件中的端口号和密钥。 选择“确定”。

还可以通过打开命令提示符窗口并输入以下命令来启动 WinDbg 会话，其中内容为上述所选择的端口以及上述由 kdnet.exe 返回的密钥。 粘贴以前保存在记事本 .txt 文件中的密钥。

```
windbg -k -d net:port=<YourDebugPort>,key=<YourKey>
```

重新启动目标计算机

连接调试器后，重新启动目标计算机。 重启电脑的一种方法是使用管理员命令提示符中的 `shutdown -r -t 0` 命令。

目标电脑重启后，调试器应自动连接。

计算机重新启动后，NIC 固件会将新的 MAC 地址分配给新添加的 KDNET PF，dbgsettings::busparams 将指向新添加的 PF。

查找 2PF 适配器的 MAC 地址

由于新添加的 PF 是 PCI 总线配置端口，因此将有一个新的 MAC 地址值由 NIC 固件分配给新添加的 PF。 kdnet.exe 工具当前不支持显示已添加 2PF 的 MAC 地址。

可通过两种方式查找新的 MAC 地址：

将 WinDbg/KD 与本地 KD 会话配合使用

在目标上运行本地内核调试器 windbg.exe -kl。

通过运行 .reload /f kdnet.dll，确保有权访问 kdnet.pdb 符号文件

运行 .kdtargetmac 命令以获取 MAC 地址。

dbgcmd

```
kd> .kdtargetmac
Force unload of kdnet.dll
ModLoad: fffff800`18510000 fffff800`18557000   kdnet.dll
Loading symbols for fffff800`18510000        kdnet.dll ->   kdnet.dll

The target machine MAC address in open-device format is: DC9840C151E8
```

运行供应商提供的固件工具

找到 MAC 地址的一种方法是运行供应商提供的固件工具。 有关下载、安装和使用供应商工具的信息，请咨询 NIC 供应商。

控制台

```
...
Base MAC:             98039baa757c           4
```

查找 MAC 地址字段。 通过将一个地址值按顺序添加到根 MAC 设备的最后一位数字来计算 KDNET 2PF MAC 地址值。 因此，对于具有地址 `98039baa757c` 的根设备，KDNET 2PF 设备将具有地址 `98039baa757d`。

还原以前的配置状态 - 删除第二个 PCI PF

可以使用 `kdnet -removepf` 选项和原始 *bus.device.function* 值从设备删除以前添加的 PF。 PF 将从 NIC 分离，PF 分配的资源将由 NIC 固件释放。

要从设备中删除 KDNET PF，请打开提升的命令提示符并运行以下命令。

```
kdnet -removepf [bus.dev.fun] [host name/host ip address] [port number]
```

其中 `bus.dev.fun` 是最初连接 PF 的 NIC 适配器的 PCI 位置端口。 这与最初传递给 kdnet -addpf 的 PCI 位置相同。

使用 -removepf 选项还可以在原始 bus.dev.fun 上通过 KDNET 重新启用内核调试。

控制台

```
C:\KDNET> kdnet -removepf 141.0.0 198.51.100.1 50001

Succeeded removing a Pci PF on :141.0.0. Please power off or reboot the
machine.

Enabling network debugging on Mellanox ConnectX-4 Lx Ethernet Adapter #2.
Manage-bde.exe not present.  Bitlocker presumed disabled.
```

kdnet.exe -removepf 命令还将从操作系统安装 {default} loadoptions 中删除 NO_KDNIC 属性，因为 KDNET 将在原始 bus.dev.fun 上启用，即 dbgsettings::busparams 将指向原始网络端口。 这将导致再次使用 KDNIC，从而在 KDNET 上再次提供网络连接。

删除 PF 后，需要重新启动计算机才能应用 BCD 更改。

控制台

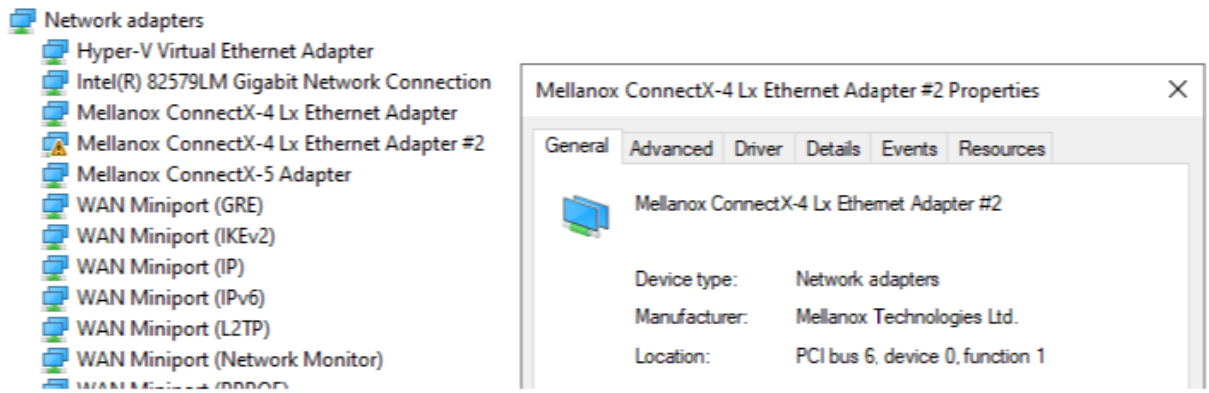
```
shutdown -r -t 0
```

主计算机适配器配置疑难解答

验证设备管理器中是否存在 2PF 适配器

通过检查新 NIC 适配器在 Windows 设备管理器适配器列表中是否具有新的 bus.dev.fun 端口来验证是否已成功添加 KDNET PF。

此图显示了三个不同的适配器，其中保留适配器 #2 供内核调试器使用。



常见错误消息 - 添加新 PF

控制台

```
C:\KDNET> kdnet -addpf 28.0.0 192.168.137.1 50005
```

```
Device Name:\\.\Mlx5Util

Pci Bus:28.0.0
```

PCI PF 已在此端口上配置：*错误=(0x80004004) 调试设备上 PF 操作失败。未为 KDNET 配置调试设备。*

- 不要在根端口上添加/删除 PF（已在此将其添加为 PF）。

常见错误消息 - 删除 PF

控制台

```
C:\KDNET> kdnet -removepf 28.0.1 192.168.137.1 50005

Adapter is not active: Error=(0x80070002)

Device Name:\\.\Mlx5Util

Pci Bus:28.0.1
```

适配器未处于活动状态：错误=(0x80070002) *调试设备上 PF 操作失败。未为 KDNET 配置调试设备*

- 不要将添加的 PF 端口与 “-removepf/-addpf” 命令行参数一起使用，因为对添加的 PF 端口执行任何操作都会导致失败（错误：适配器在端口上未处于活动状态），因为供应商 NIC 收件箱驱动程序明确设置为在添加的 PF 上*不*运行。
- 这两个命令行选项 (-addpf/-removepf) 都必须只在根 PCI 设备上使用。

控制台

```
C:\KDNET> kdnet -removepf 28.0.0 192.168.137.1 50005

Device Name:\\.\Mlx5Util

Pci Bus:28.0.0
```

在此端口上没有要删除的 PCI PF：*错误=(0x80004005) 调试设备上 PF 操作失败。未为 KDNET 配置调试设备。*

- 如果添加新 PF，然后决定在不重新启动的情况下将其删除，这将导致失败，因为供应商 NIC 固件需要重新启动/重置 NIC HW，然后才能识别新添加的 PF。

一些常见的错误消息 - BCDEdit

BCD OS {default} 安装中*不存在* NO_KDNIC。

- 不建议在添加新 PF 后使用 bcdedit.exe 修改/更改调试设备 (dbgsettings)。 kdnet -addpf/removepf 命令行选项将配置调试设备，还将自动向/从 `{default}::loadoptions` 中添加/删除 `NO_KDNIC` 令牌。

另请参阅

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

手动设置通过 USB 3.0 线缆进行的内核模式调试

手动设置内核模式调试

配置 EXDI 调试器传输

项目 • 2024/11/01

本主题介绍如何使用 EXDI 来设置内核模式调试。 扩展调试接口 (EXDI) 是软件调试器和调试目标之间的适应层。

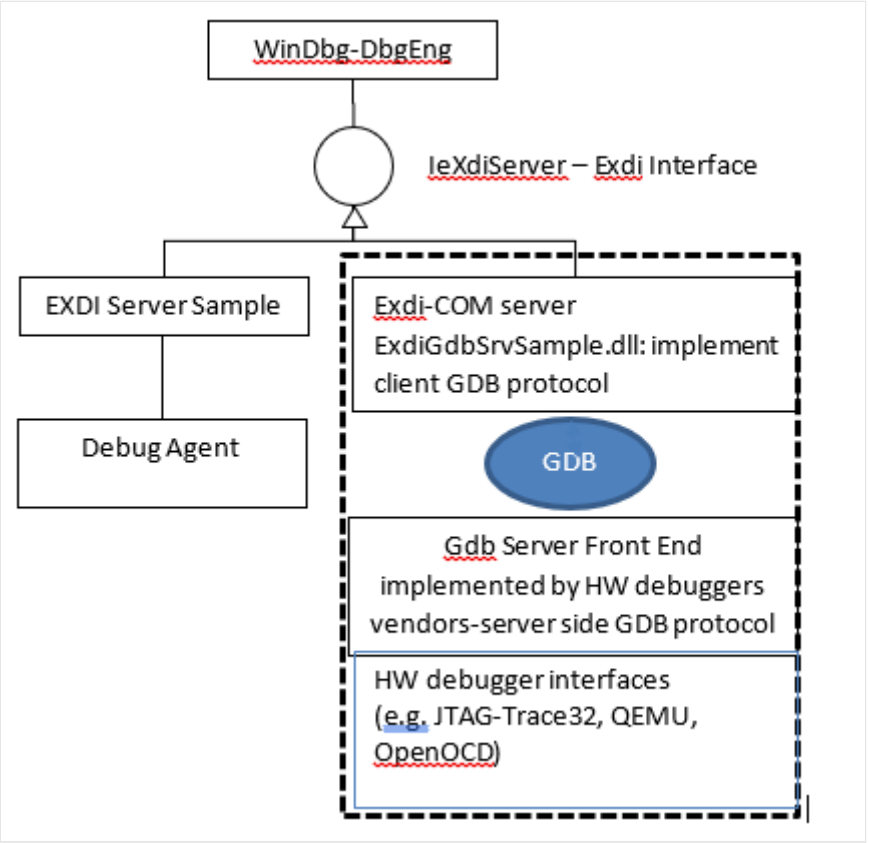
- 从 Windows 版本 22000 开始，Windows 调试工具支持使用 EXDI 进行内核调试。
- 从版本 1.2410.11001.0 开始，调试器中提供了配置 EXDI 的用户界面。

EXDI 可用于与 QEMU 虚拟环境建立连接。 有关详细信息，请参阅 使用 EXDI 设置 QEMU 内核模式调试。

> **备注**
>
> EXDI 是一种针对特定环境的高级专业调试方式。 使用标准 KDNET 连接更容易配置，因此建议使用。 要自动设置网络调试，请参阅**设置 KDNET 网络内核自动调试**。

EXDI COM 服务器概述

EXDI 是一个接口，通过添加对硬件调试器（例如基于 JTAG 或基于 GdbServer）的支持来扩展 WinDbg。 下图说明了 EXDI-GdbServer 的作用。



COM 服务器是指实现 COM 接口的二进制组件。 在本例中，ExdiGdbSrv.dll为 Windows 调试器协议客户端实现 exdi3.idl。

ExdiGdbsrv.dll本身实现 GDB-RSP 协议的客户端，GDB 服务器端（有时称为 GDB 服务器存根）由 QEMU GDB 服务器（或 Trace32/OpenOCD/UEFI GDB 服务器存根等）实现。

由于 EXDI 连接不依赖于 Windows 或目标电脑上加载的 Windows 调试 KDNET 协议。 由于不需要这些软件调试器组件，因此 EXDI 在早期设备启动和调试 OS 启动问题中非常有用。

> ⓘ **重要**
>
> 由于 EXDI 未使用 KDNET 协议，因此连接的调试器所掌握的有关电脑上运行情况的信息要少得多，许多命令的运行方式也会不同，甚至根本无法运行。 访问被调试代码的专用符号有助于调试器更好地理解目标系统的代码执行。 有关详细信息，请参阅**公共符号和专用符号**。

EXDI 内核模式设备要求

运行调试器的计算机称为*主计算机*，正在调试的计算机称为*目标计算机*。

以下为必需项：

- 在目标和主机计算机上，有所需环境（例如 QEMU）支持的支持网卡。

- 目标和主机之间使用 TCP/IP 的网络连接。
- Windows 10 或 Windows 11 版本 22000 或更高版本。

限制

如上所述，由于 EXDI 不使用 KDNET 协议，因此连接的调试器掌握的目标系统信息较少，并且调试器的使用不同。 如果不能访问目标代码的专用符号，许多使用符号来了解目标系统状态的命令将不起作用。 在这种情况下，可以查看内存并注册内容和反汇编代码。 如果没有私有符号，确定运行代码的位置或执行其他常见调试器任务就会非常困难和耗时。

并发 EXDI 和 KDNET 调试

在某些复杂的方案中，例如在早期设备启动中，与目标设备建立两个连接可能很有用。 一个 EXDI 和一个 KDNET。 如果目标是 Windows OS，则 KDNET 软件调试将按正常方式配置，例如连接到虚拟机。 在此设置中，两个并发调试器中的任何一个都可以中断以调试目标计算机上的代码。

进程服务器中的 WinDbg

二进制 EXDI 组件可以在 Windbg 进程或 Windbg 进程中运行。 使用 EXDI UI 或 `Inproc=<EXDI COM server binary>` 大幅提高可靠性，方法是减少 COM 启动错误。 因此，建议始终使用 Inproc 参数运行 EXDI 会话，该参数在使用 UI 时始终启用。

对于命令行启动，默认选项已退出进程，但应使用 `InProc=ExdiGdbDrv.dll` 参数启用进程内选项。

COM GDB 服务器客户端

本主题介绍实现 EXDI COM 调试器接口的 EXDI COM GDB 服务器客户端（ExdiGdbSrv.dll）。 可以使用同一 COM 接口来实现其他接口，例如 JTAG-DCI 的 EXDI COM 服务器。

使用 EXDI 连接的过程摘要

使用此过程将 EXDI 连接与 WinDbg 配合使用。

1. 在主机系统中下载并安装 Windows 调试工具。
2. 使用 UI 或 -kx 选项启动 WinDbg 以连接到 EXDI 服务器。
3. 使用 WinDbg 使用一组可用的调试器命令调试目标系统。

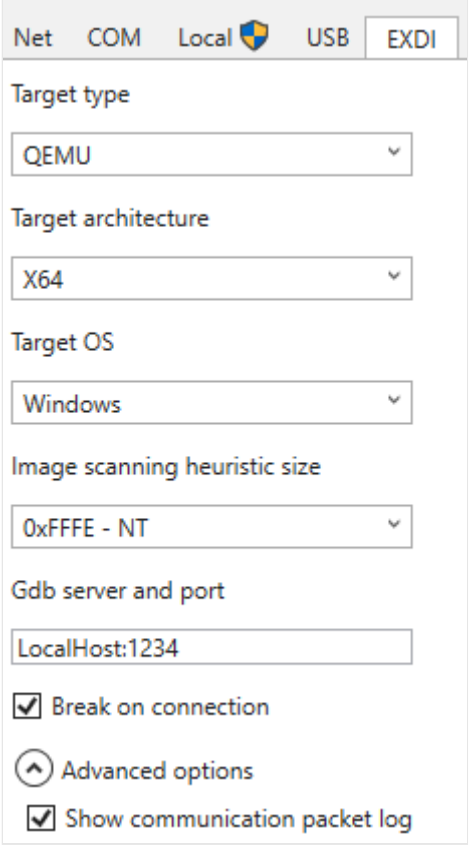
有关 EXDI 使用方案的示例，请参阅 使用 EXDI 设置 QEMU 内核模式调试。

下载并安装 Windows 调试器工具

在主机系统上安装 Windows 调试工具。 有关下载和安装调试器工具的信息，请参阅适用于 Windows 的调试工具。

启动 WinDbg 并连接到 EXDI 服务器

可以在 EXDI 内核连接 UI 中配置以下选项。



- **目标类型** `[Trace32|BMC-OpenOCD|QEMU|VMWare|UEFI]` 根据要调试的目标类型选择。 以下目标类型可用。
  - Trace32：Lauterbach Trace32 HW 调试器 GDB 服务器配置
  - BMC-OpenOCD ： BMC-OpenOCD HW 调试器 GDB 服务器配置
  - QEMU：QEMU SW 模拟器 GDB 服务器配置
  - VMWare：VMWare GDB 服务器配置
  - UEFI：UEFI 固件调试
- **目标体系结构** `[x86 | ARM64 | x64]` - 目标处理器体系结构。 请注意，所有目标类型都可能不支持所有目标体系结构。
- **目标 OS** `[Windows|Linux]` - 根据要调试的目标 OS 进行选择。

- **图像扫描启发式大小** `[None | 0xFE - PreNT |0xFFE - NT]` - 选择三个选项之一来确定图像扫描的启发式大小。 此值配置调试器引擎如何扫描查找 PE DOS 签名的内存，该签名用于收集代码执行状态。 如果未*指定属性值*（或"0"），则调试器引擎不会使用快速启发法并回退到扫描整个内存以查找 PE DOS 签名的旧试探法。 为每个目标类型选择默认值，建议使用默认值。

- **Gdb 服务器和端口** `TargetIPAddress:TargetPortAddress` - 设置为包含的字符串、IPTargetAddress、冒号和目标 PortAddress。 例如 `LocalHost:1234` 或 `168.82.1.5:5555`。

- **连接中断** `[on|off]` 选中复选框以在建立连接后闯入目标。

- *高级选项*

  显示通信数据包日志 `[on|off]` - 选中复选框以显示原始 GDBServer 通信数据包日志中的十六进制值，以便进行调试和故障排除。

选择所需选项后，选择" **确定** "进行连接。

使用 EXDI 配置 XML 文件配置高级选项

本主题中所述的用户界面中提供了大多数必需的选项。 有关使用 EXDI 配置 XML 文件配置高级选项的信息，请参阅 EXDI XML 配置文件。

WinDbg 命令行 EXDI 示例

若要在命令提示符处启动使用 EXDI 接口的 windbg 会话，请使用这些选项。

控制台

```
c:\Debuggers> windbg.exe -v -kx exdi:CLSID={29f9906e-9dbe-4d4b-b0fb-
6acf7fb6d014},Kd=Guess,InProc=ExdiGdbDrv.dll,DataBreaks=Exdi
```

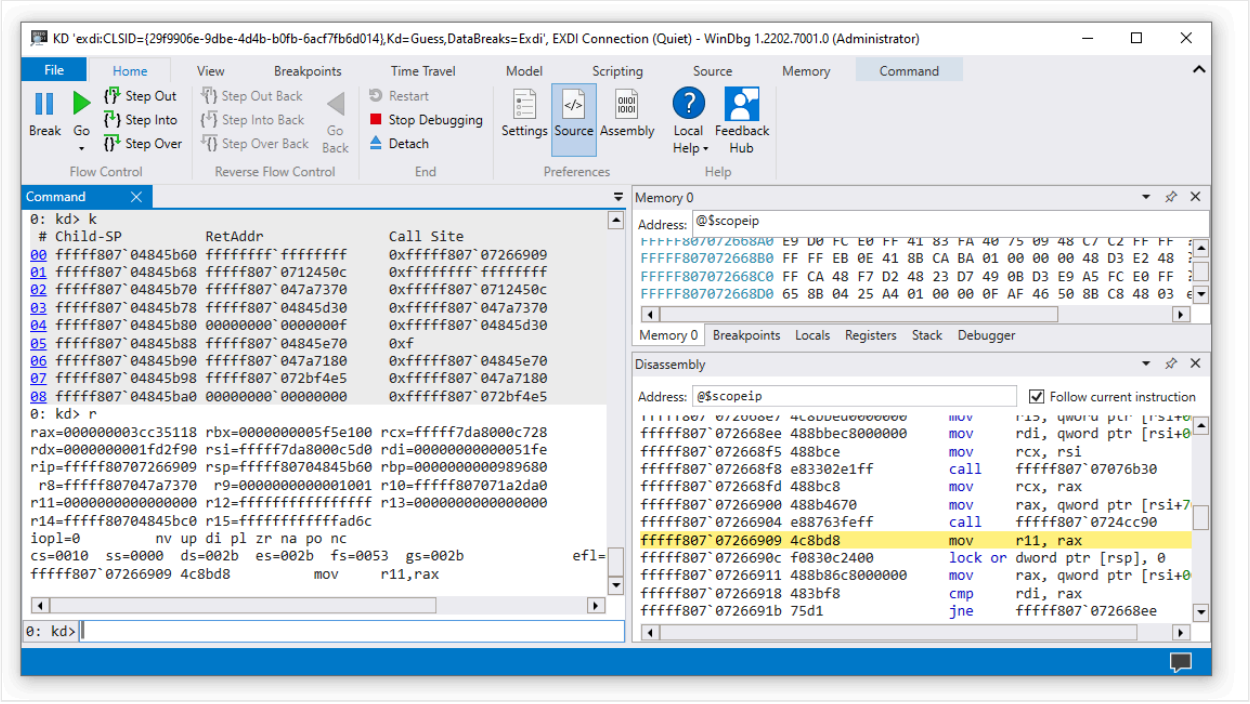
要显示用于诊断目的的其他输出，可以使用 **-v**： 详细会话。 有关 WinDbg 选项的一般信息，请参阅 WinDbg 命令行选项。 有关详细信息，请参阅下方的 EXDI WinDbg 加载参数。

调试器控制台将显示 EXDI 传输初始化。

dbgcmd

```
EXDI: DbgCoInitialize returned 0x00000001
EXDI: CoCreateInstance() returned 0x00000000
EXDI: QueryInterface(IExdiServer3) returned 0x00000000
```

```
EXDI: Server::GetTargetInfo() returned 0x00000000
EXDI: Server::SetKeepaliveInterface() returned 0x00000000
EXDI: Server::GetNbCodeBpAvail() returned 0x00000000
EXDI: ExdiNotifyRunChange::Initialize() returned 0x00000000
EXDI: LiveKernelTargetInfo::Initialize() returned 0x00000000
EXDI: Target initialization succeeded
Kernel Debugger connection established
```



EXDIGdbServer 控制台窗口还可以显示有关 EXDI 连接状态的信息。 有关控制台的详细信息，请参阅故障排除。

EXDI WinDbg 加载参数

以下参数用于 WinDbg 来启动 EXDI 内核会话。

-kx EXDI：*Options*

以下 EXDI 选项可用于 -kx 选项。 应使用逗号分隔每个选项。

展开表

参数	说明
CLSID	分配给 LiveExdiGdbSrvServer 的类 ID（如 ExdiGdbSrv.idl 文件中定义）。
Kd=Guess -or-NtBaseAddr	调试器引擎将使用常规启发式机制 -或 - 查找 NT 基址。
ForceX86	强制调试器引擎使用 IeXdiX86Context3 接口获取/设置 CPU 上下文。

参数	说明
DataBreaks=Exdi	允许使用数据断点。
Inproc	允许使用 inproc Exdi-Server。 建议使用此选项 - *InProc=ExdiGdbDrv.dll*
PathToSrvCfgFiles	EXDI 的 XML 配置文件的路径。

控制启发式搜索和启发式大小

如前所述，调试器使用启发式算法来查找 NT 基址。 若要取消启发式搜索，请在通过命令行启动 WinDbg 时完成以下步骤。

- 将要附加到的目标服务器的exdiconfigdata.xml文件中的启发式ScanSize 设置为 0。
- 在 `kd=NtBaseAddr` Windbg 命令行上使用启发式类型。

有关使用 XML 配置文件的详细信息，请参阅 EXDI XML 配置文件。

使用 WinDbg 调试目标系统 - 断点

dbgeng.dll 使用启发式算法来查找中断命令发生时 NT 基本加载地址的位置。 如果没有专用符号，此过程将失败。

这意味着在许多连接序列下，中断将无法发挥预期的功能。 如果手动侵入代码，它将是 Windows 当时正在执行的一个随机位置。 由于可能无法获得目标代码的符号，因此很难使用符号设置断点。

调试器命令

以下直接访问内存的命令可以正常使用。

k、kb、kc、kd、kp、kP、kv（显示堆栈回溯）

r（寄存器）

d、da、db、dc、dd、dD、df、dp、dq、du、dw（显示内存）

u（取消汇编）

可以使用 p (Step) 单步执行代码。

还可以使用一些命令来尝试查找要调试的代码。

s（搜索内存）

.imgscan（查找映像标头）

Imgscan 对 EDXI 调试很有帮助，因为与传统的基于 KDNET 的内核调试不同，它可能无法根据符号设置断点。 定位所需的目标图像，可以方便地使用其位置来设置内存访问断点。

.exdicmd（EXDI 命令）

.exdicmd 使用活动 EXDI 调试连接向目标系统发送 EXDI 命令。 有关详细信息，请参阅 .exdicmd（EXDI 命令）。

故障排除

使用 ExdiGdbServer 窗口中的输出监视连接序列。

问题：错误：无法与 GbDServer 建立连接。 验证连接字符串 `<hostname/ip>:portnumber`

此问题可能是由以下原因造成的：

- ExdiGdbSrv.dll无法连接到目标 GDB 服务器。
- GDB 服务器尚未在目标上运行。
- 防火墙问题，使用 ping、tracert 或其他工具验证 GDB 流量是否可以通过防火墙，以确保两个 IP 地址都可以访问。

问题：目标系统的错误方案不可用 - DbgCoInitialize 返回 0x00000001

如果目标系统未加载或无法使用，可能会返回以下输出结果。

控制台

```
Microsoft (R) Windows Debugger Version 10.0.20317.1 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

EXDI: DbgCoInitialize returned 0x00000001
```

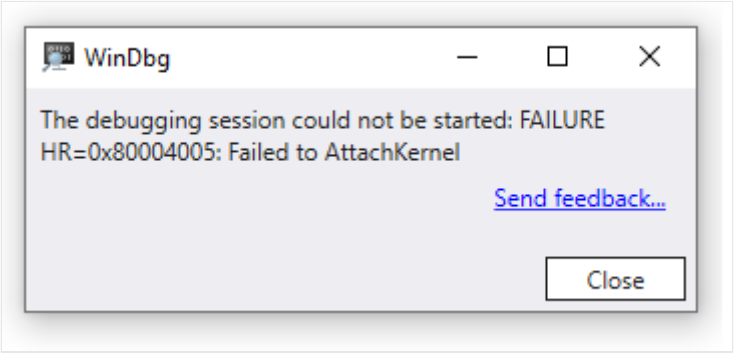
当 ExdiGdbSrv.dll COM 服务器无法连接到 QEMU GDServer 时，这是一个常见错误，因此可能会由于以下原因而失败：

- ExdiGdbSrv.dll 的上一个会话仍由 dllhost.exe 进程托管，因此需要终止 dllhost.exe 进程。 在命令提示符下使用 `TaskList` ，找到承载ExdiGdbSrv.dll的dllhost.exe的 PID。 使用 `TaskKill /PID <PID ID> /f` 并终止关联的 PID。 有关使用 PID 的详细信息，请参阅 查找进程 ID。
- QEMU gdbserver 尚未启动，或者 exdiconfigdata.xml 文件包含无效的 IP:Port 值。 如果 WinDbg 会话是在与 QEMU Windows 虚拟机相同的主机上启动的，则 IP=LocalHost.
- 通过 dllhost.exe 进程（COM 相关）启动 EXDI COM 服务器 (ExdiGDbSrv.dll) 时失败。 要解决此问题，请重启主机调试器电脑或注销 Windows 并再次登录。 如果这不起作用，请在再次重启/登录后重新注册 EXDI COM 服务器。
  - `regsvr32.exe <full path to the ExdiGdbSrv.dll)`

问题：无法启动调试会话：FAILURE HR=0x80004005:Failed to AttachKernel。

此问题可能是由以下原因造成的：

- 如上所述，ExdiGdbSrv.dll 的上一个会话可能仍然处于活动状态。 如上所述，找到并终止相关的 DLL 主机。



问题：无法使用 EXDI 启动内核调试。

此问题可能是由以下原因造成的：

- 在主机调试器计算机上运行 ExdiGdbSrv.dll（由 dllhost.exe 托管）的另一个实例。
- 终止托管 ExdiGdbSrv.dll 的 COM 服务的额外实例。
  - 首先使用主机电脑上的实用工具（如 TList）列出进程。 承载 ExdiGdbSrv.dll 的 DLLHost 将显示 *ExdiGdbServer*。

    > tlist 261928 dllhost.exe ExdiGdbServer

  - 在调试器命令提示符处使用 `kill -f XXXXX` 通过进程编号终止进程。

问题：错误：无法配置 GdbServer 会话。

此问题可能是由以下原因造成的：

- 查找会话信息时出错，例如 XML 配置文件的路径。

问题：错误：未定义 EXDI_GDBSRV_XML_CONFIG_FILE 环境变量。

此问题可能是由以下原因造成的：

- ExdiGdbSrv.dll 环境变量未设置或在环境中不可用。

问题：错误：未定义EXDI_GDBSRV_XML_CONFIG_FILE环境变量。 Exdi-GdbServer 示例目前不会继续。 设置 Exdi XML 配置文件的完整路径。

此问题可能是由以下原因造成的：

- 未*设置* EXDI_GDBSRV_XML_CONFIG_FILE 环境变量。 在某些情况下，如果按“OK”键，ExdiGDbSrv.dll 将继续工作，但 windbg.exe 将无法查询系统寄存器（例如通过 rdmsr/wrmsr 函数）。

另请参阅

使用 EXDI 设置 QEMU 内核模式调试

.exdicmd（EXDI 命令）

EXDI XML 配置文件

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

手动设置内核模式调试

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

EXDI XML 配置文件

项目 • 2024/11/01

本主题介绍如何使用 EXDI XML 配置文件配置高级选项。 有关使用 WinDbg 用户界面配置 EXDI 的一般信息，请参阅 配置 EXDI 调试器传输。 用户界面中提供了最常见的设置，这是一种更简单的方法，然后手动编辑此处所述的 EXDI XML 配置文件。

扩展调试接口 (EXDI) 是软件调试器和调试目标之间的适应层。 从 Windows 版本 22000 开始，Windows 调试工具支持使用 EXDI 进行内核调试。

> ⓘ **备注**
>
> EXDI 是一种针对特定环境的高级专业调试方式。 使用标准 KDNET 连接更容易配置，因此建议使用。 要自动设置网络调试，请参阅**设置 KDNET 网络内核自动调试**。

使用 EXDI 配置 XML 文件配置高级选项

EXDI GDB COM 服务器 (ExdiGdbSrv.dll) 需要使用两个 xml 文件。

1. *exdiConfigData.xml* - 该文件包含 GDB 服务器客户端与 HW 调试器 GDB 服务器目标成功建立 GDB 会话所需的主要配置数据，因此如果 EXDI_GDBSRV_XML_CONFIG_FILE 环境变量*未*设置该文件位置，GDB 服务器客户端将无法运行。 每个 xml 标记都可以配置特定的 GDB 服务器功能集。 有关可以在 XML 中修改的属性列表和示例 XML，请参阅下文。

2. Systemregister.xml - 此文件包含系统寄存器与用于访问寄存器的代码之间的映射。 之所以需要这样做，是因为 GDB 服务器没有在 xml 文件中提供访问代码，而调试器是通过访问代码来访问每个系统寄存器的。 如果未通过环境变量 `EXDI_SYSTEM_REGISTERS_MAP_XML_FILE` 设置文件，则 ExdiGdbSrv.dll 将继续工作，但调试器将无法通过 rdmsr 或 wrmsr 命令访问任何系统注册。 GDB 服务器硬件调试器应支持这些寄存器的列表（系统 xml 文件中发送的寄存器列表中应存在特定的系统注册名称）。

EXDI UI 和 XML 配置文件

WinDbg 中的 EXDI UI 使用 XML 文件参数，并在 UI 所需的参数（如 IP：Port 值）中合并。 如果需要修改默认 XML 文件参数，请使用参数 `PathToSrvCfgFiles=<path to the modified exdiconfigdata.xml file>` 从命令行启动 WinDbgNext 应用程序。

目标体系结构

EXDI UI 目标体系结构字段值需要匹配仅针对未实现描述 Windows OS 目标体系结构的目标描述 XML 文件的目标 GDB 服务器存根上运行的 Windows OS。 此target.xml文件信息由 GDB 服务器存根在 GDB 客户端与 GDB 服务器存根之间的 GDB-RSP 协议握手期间发送。

Windbg-ExdiGdbSrv 仍可以设置正确的目标 OS 体系结构，即使用户在 EXDI UI 中设置了不正确的目标体系结构字段输入值。 目标 OS 体系结构将从 GDB 服务器提供的target.xml说明文件（包括 GDB 服务器握手中的target.xml文件）中获取和配置。 若要检查目标体系结构，请使用有效的计算机 .effmach 调试器命令。

GDBServer 标记和属性

下表描述了 `exdiConfigData.xml` 文件中定义的 GDBServer 标记和属性。

展开表

参数	说明
ExdiTargets	指定 ExdiGgbSrv.dll 将使用哪些特定的 GDB 服务器目标配置来与 GDB 服务器目标建立 GDB 连接，因为 exdiConfigData.xml 文件包括 ExdiGdbSrv.dll 当前支持的所有 GDB 服务器（在将 ExdiGdbSrv.dll 用于特定 GDB 服务器之前必须填充此文件）。
CurrentTarget	指定 GDB 服务器目标的名称（例如，此属性值应与 exdiConfigData.xml 文件包含的 `<ExdiTarget Name=` 标记之一的名称值匹配。
ExdiTarget	这是每个 GDB 服务器目标组件包含的所有配置数据的开始标记。
名称	指定 GDB 服务器的名称（例如 QEMU、BMC-OpenOCD、Trace32、VMWare）。
agentNamePacket	这是 GDB 客户端的名称，因为 GDB 服务器 HW 调试器可以识别它。 GDB 服务器 HW 调试器可以使用这一点为特定 GDB 客户端（例如 Trace32 GDB 服务器）配置自身（例如 Trace32 GDB 服务器）需要 ExdiGdbSrv.dll 发送"QMS.windbg"名称来标识 windbg-GDB 客户端，然后启用仅支持 MS GDB 服务器客户端 (exdiGdbSrv.dll) 的自定义 GDB 内存数据包。
ExdiGdbServerConfigData	指定与组件相关的 ExdiGdbSrv.dll 配置参数。
uuid	指定 ExdiGdbSrv.dll 组件的 UUI。

参数	说明
displayCommPackets	如果标记为“是”，则会在命令日志窗口中显示 RSP 协议通信字符。 如果为“否”，则仅显示请求-响应对文本。
enableThrowExceptionOnMemoryErrors	当存在 GDB 错误响应数据包 (E0x) 时，GDB 服务器客户端将检查此属性，以确定客户端是否应引发异常并停止读取内存。
qSupportedPacket	这允许将 GDB 客户端配置为请求 GDB 服务器 HW 调试器应按照 xml 目标说明文件发送哪个 xml 寄存器体系结构文件（基本上，客户端将通知 GDB 服务器客户端客户端当前支持 x64 体系结构）。
ExdiGdbServerTargetData	指定与 GdbServer 会话调试的硬件目标相关的参数。
targetArchitecture	包含目标硬件体系结构的字符串。 可能的值：X86、X64、ARM、ARM64。 目前，exdiGdbSrv.dll 仅支持 X86 和 ARM。
targetFamily	包含目标硬件系列的字符串。 可能的值：ProcessorFamilyX86、ProcessorFamilyX64、ProcessorFamilyARM、ProcessorFamilyARM64。
numberOfCores	目标支持的处理器核心数。 使用多 Gdbserver 会话（T32-GdbServer 会话）时，将验证此参数。 下面的“MultiCoreGdbServerSessions”属性应设置为“yes”。
EnableSseContext	如果标记为“yes”，则“g”上下文 RSP 数据包将包括浮点寄存器值。 此参数仅适用于 Intel 系列目标。
heuristicScanSize	这将配置调试器引擎快速启发式算法，以按指定大小减少扫描的内存探测（如果属性值*未被*指定（或为“0”），则调试器引擎不会使用快速启发式，并回退到扫描整个内存以查找 PE DOS 签名的旧试探法。 常见的扫描大小值是0xfffe（最适合 NT）或0xffe（适用于预 NT 应用）。
targetDescriptionFile	指定 GDB 服务器是否在发送每个单独的 xml 文件之前发送目标说明头文件。 此字段为空，则 GDB 服务器客户端不会请求 xml 体系结构系统注册（例如，不支持在单独的 xml 文件中发送体系结构寄存器的 Trace32 GDB 服务器）。
GdbServerConnectionParameters	指定 GdbServer 会话参数。 这些参数用于控制 ExdiGdbSrv.dll 组件和 GdbServer 之间的 RSP GdbServer 会话。
MultiCoreGdbServerSessions	标志如果为“yes”，则我们将具有多核 GdbServer 会话（T32-GdbServer 后端使用的会话）。 如果为“no”，我们将仅与 GdbServer 的一个实例通信。

参数	说明
MaximumGdbServerPacketLength	这是一个数据包支持的最大 GdbServer 长度。
MaximumConnectAttempts	这是最大连接尝试次数。 尝试建立与 GdbServer 的 RSP 连接时，ExdiGdbSrv.dll 会使用它。
SendPacketTimeout	这是 RSP 发送超时。
ReceivePacketTimeout	这是 RSP 接收超时。
HostNameAndPort	这是格式`<hostname/ip address:Port number>`的连接字符串。 可以有多个 GdbServer 连接字符串（例如 T32 多核 GdbServer 会话）。 连接字符串数应与核心数匹配。
ExdiGdbServerMemoryCommands	指定发出 GDB 内存命令的各种方法，以便在不同的异常 CPU 级别获取系统寄存器值或读/写访问内存（例如 BMC-OpenOCD 通过"aarch64 mrs nsec/sec `<access code>`"自定义命令提供对 CP15 寄存器的访问权限）。
GdbSpecialMemoryCommand	如果为"yes"，则 GDB 服务器支持自定义内存命令（例如系统注册，应为 Trace32 GDB 服务器设置）。
PhysicalMemory	如果为"yes"，则 GDB 服务器支持用于读取物理内存的自定义命令（为 Trace32 GDB 服务器设置）。
SupervisorMemory	如果为"yes"，则 GDB 服务器支持用于读取主管内存的自定义命令（为 Trace32 GDB 服务器设置）。
SpecialMemoryRegister	如果为"yes"，则 GDB 服务器支持用于读取系统寄存器的自定义命令（为 Trace32 GDB 服务器设置）。
SystemRegistersGdbMonitor	如果为"yes"，则 GDB 服务器通过 GDB 监视器命令支持自定义命令（为 BMC Open-OCD 设置）。
SystemRegisterDecoding	如果为"yes"，则 GDB 客户端在发送 GDB 监视器命令之前接受解码访问代码。
ExdiGdbServerRegisters	指定特定的体系结构注册核心集。
体系结构	定义寄存器集的 CPU 体系结构。
FeatureNameSupported	这是由 xml 系统注册说明文件提供的系统注册组的名称。 在 GDB 服务器发送 xml 文件时，需要用它来识别 xml 文件中的系统注册 xml 组。
SystemRegistersStart	这是为了识别作为核心寄存器集一部分报告的第一个系统寄存器（低寄存器编号/顺序）（例如，在 X64 上，QEMU 不会将 x64 系统寄存器集作为单独的 xml 目标说明文件报告，因此系统寄存器是核心寄存器的一部分）。

参数	说明
SystemRegistersEnd	这是为了标识作为核心寄存器集的一部分报告的最后一个系统寄存器（高寄存器编号/顺序）。
名称	寄存器的名称。
订单	这是一个数字，用于标识寄存器数组中的索引。 GDB 客户端和服务器集/查询（`p<number>”/”q<number>`）将使用此数字来注册数据包。
大小	这是寄存器大小（以字节为单位）。

Sample exdiConfigData.xml file

XML

```
<ExdiTargets CurrentTarget = "QEMU">
<!-- QEMU SW simulator GDB server configuration -->
    <ExdiTargets CurrentTarget="QEMU">
    <!--  QEMU SW simulator GDB server configuration  -->
    <ExdiTarget Name="QEMU">
    <ExdiGdbServerConfigData agentNamePacket="" uuid="72d4aeda-9723-4972-
b89a-679ac79810ef" displayCommPackets="yes" debuggerSessionByCore="no"
enableThrowExceptionOnMemoryErrors="yes"
qSupportedPacket="qSupported:xmlRegisters=aarch64,i386">
    <ExdiGdbServerTargetData targetArchitecture="ARM64"
targetFamily="ProcessorFamilyARM64" numberOfCores="1" EnableSseContext="no"
heuristicScanSize="0xfffe" targetDescriptionFile="target.xml"/>
    <GdbServerConnectionParameters MultiCoreGdbServerSessions="no"
MaximumGdbServerPacketLength="1024" MaximumConnectAttempts="3"
SendPacketTimeout="100" ReceivePacketTimeout="3000">
    <Value HostNameAndPort="LocalHost:1234"/>
    </GdbServerConnectionParameters>
    <ExdiGdbServerMemoryCommands GdbSpecialMemoryCommand="no"
PhysicalMemory="no" SupervisorMemory="no" HypervisorMemory="no"
SpecialMemoryRegister="no" SystemRegistersGdbMonitor="no"
SystemRegisterDecoding="no"> </ExdiGdbServerMemoryCommands>
        <ExdiGdbServerRegisters Architecture = "ARM64" FeatureNameSupported
= "sys">
            <Entry Name ="X0"  Order = "0" Size = "8" />
            <Entry Name ="X1"  Order = "1" Size = "8" />
            <Entry Name ="X2"  Order = "2" Size = "8" />
            <Entry Name ="X3"  Order = "3" Size = "8" />
            <Entry Name ="X4"  Order = "4" Size = "8" />
            <Entry Name ="X5"  Order = "5" Size = "8" />
            <Entry Name ="X6"  Order = "6" Size = "8" />
            <Entry Name ="X7"  Order = "7" Size = "8" />
            <Entry Name ="X8"  Order = "8" Size = "8" />
            <Entry Name ="X9"  Order = "9" Size = "8" />
            <Entry Name ="X10" Order = "a"  Size = "8" />
```

```
            <Entry Name ="X11" Order = "b"  Size = "8" />
            <Entry Name ="X12" Order = "c"  Size = "8" />
            <Entry Name ="X13" Order = "d"  Size = "8" />
            <Entry Name ="X14" Order = "e"  Size = "8" />
            <Entry Name ="X15" Order = "f"  Size = "8" />
            <Entry Name ="X16" Order = "10" Size = "8" />
            <Entry Name ="X17" Order = "11" Size = "8" />
            <Entry Name ="X18" Order = "12" Size = "8" />
            <Entry Name ="X19" Order = "13" Size = "8" />
            <Entry Name ="X20" Order = "14" Size = "8" />
            <Entry Name ="X21" Order = "15" Size = "8" />
            <Entry Name ="X22" Order = "16" Size = "8" />
            <Entry Name ="X23" Order = "17" Size = "8" />
            <Entry Name ="X24" Order = "18" Size = "8" />
            <Entry Name ="X25" Order = "19" Size = "8" />
            <Entry Name ="X26" Order = "1a" Size = "8" />
            <Entry Name ="X27" Order = "1b" Size = "8" />
            <Entry Name ="X28" Order = "1c" Size = "8" />
            <Entry Name ="fp"  Order = "1d" Size = "8" />
            <Entry Name ="lr"  Order = "1e" Size = "8" />
            <Entry Name ="sp"  Order = "1f" Size = "8" />
            <Entry Name ="pc"  Order = "20" Size = "8" />
            <Entry Name ="cpsr" Order = "21" Size = "8" />
            <Entry Name ="V0" Order = "22" Size = "16" />
            <Entry Name ="V1" Order = "23" Size = "16" />
            <Entry Name ="V2" Order = "24" Size = "16" />
            <Entry Name ="V3" Order = "25" Size = "16" />
            <Entry Name ="V4" Order = "26" Size = "16" />
            <Entry Name ="V5" Order = "27" Size = "16" />
            <Entry Name ="V6" Order = "28" Size = "16" />
            <Entry Name ="V7" Order = "29" Size = "16" />
            <Entry Name ="V8" Order = "2a" Size = "16" />
            <Entry Name ="V9" Order = "2b" Size = "16" />
            <Entry Name ="V10" Order = "2c" Size = "16" />
            <Entry Name ="V11" Order = "2d" Size = "16" />
            <Entry Name ="V12" Order = "2e" Size = "16" />
            <Entry Name ="V13" Order = "2f" Size = "16" />
            <Entry Name ="V14" Order = "30" Size = "16" />
            <Entry Name ="V15" Order = "31" Size = "16" />
            <Entry Name ="V16" Order = "32" Size = "16" />
            <Entry Name ="V17" Order = "33" Size = "16" />
            <Entry Name ="V18" Order = "34" Size = "16" />
            <Entry Name ="V19" Order = "35" Size = "16" />
            <Entry Name ="V20" Order = "36" Size = "16" />
            <Entry Name ="V21" Order = "37" Size = "16" />
            <Entry Name ="V22" Order = "38" Size = "16" />
            <Entry Name ="V23" Order = "39" Size = "16" />
            <Entry Name ="V24" Order = "3a" Size = "16" />
            <Entry Name ="V25" Order = "3b" Size = "16" />
            <Entry Name ="V26" Order = "3c" Size = "16" />
            <Entry Name ="V27" Order = "3d" Size = "16" />
            <Entry Name ="V28" Order = "3e" Size = "16" />
            <Entry Name ="V29" Order = "3f" Size = "16" />
            <Entry Name ="V30" Order = "3f" Size = "16" />
            <Entry Name ="V31" Order = "3f" Size = "16" />
```

```
            <Entry Name ="fpsr" Order = "40" Size = "4" />
            <Entry Name ="fpcr" Order = "41" Size = "4" />
        </ExdiGdbServerRegisters>


        <!-- x64 GDB server core resgisters -->
        <ExdiGdbServerRegisters Architecture = "X64" FeatureNameSupported =
"sys" SystemRegistersStart = "18" SystemRegistersEnd = "20" >
            <Entry Name ="rax" Order = "0" Size ="8" />
            <Entry Name ="rbx" Order = "1" Size ="8" />
            <Entry Name ="rcx" Order = "2" Size ="8" />
            <Entry Name ="rdx" Order = "3" Size ="8" />
            <Entry Name ="rsi" Order = "4" Size ="8" />
            <Entry Name ="rdi" Order = "5" Size ="8" />
            <Entry Name ="rbp" Order = "6" Size ="8" />
            <Entry Name ="rsp" Order = "7" Size ="8" />
            <Entry Name ="r8"  Order = "8" Size ="8" />
            <Entry Name ="r9"  Order = "9" Size ="8" />
            <Entry Name ="r10" Order = "a" Size ="8" />
            <Entry Name ="r11" Order = "b" Size ="8" />
            <Entry Name ="r12" Order = "c" Size ="8" />
            <Entry Name ="r13" Order = "d" Size ="8" />
            <Entry Name ="r14" Order = "e" Size ="8" />
            <Entry Name ="r15" Order = "f" Size ="8" />
            <Entry Name ="rip" Order = "10" Size ="8" />
            <!-- <flags id="x64_eflags" size="4">
                <field name="" start="22" end="31"/>
                <field name="ID" start="21" end="21"/>
                <field name="VIP" start="20" end="20"/>
                <field name="VIF" start="19" end="19"/>
                <field name="AC" start="18" end="18"/>
                <field name="VM" start="17" end="17"/>
                <field name="RF" start="16" end="16"/>
                <field name="" start="15" end="15"/>
                <field name="NT" start="14" end="14"/>
                <field name="IOPL" start="12" end="13"/>
                <field name="OF" start="11" end="11"/>
                <field name="DF" start="10" end="10"/>
                <field name="IF" start="9" end="9"/>
                <field name="TF" start="8" end="8"/>
                <field name="SF" start="7" end="7"/>
                <field name="ZF" start="6" end="6"/>
                <field name="" start="5" end="5"/>
                <field name="AF" start="4" end="4"/>
                <field name="" start="3" end="3"/>
                <field name="PF" start="2" end="2"/>
                <field name="" start="1" end="1"/>
                <field name="CF" start="0" end="0"/>
            </flags> -->
            <Entry Name ="eflags" Order = "11" Size ="4" />

            <!-- Segment registers -->
            <Entry Name ="cs" Order = "12" Size ="4" />
            <Entry Name ="ss" Order = "13" Size ="4" />
            <Entry Name ="ds" Order = "14" Size ="4" />
```

```
            <Entry Name ="es" Order = "15" Size ="4" />
            <Entry Name ="fs" Order = "16" Size ="4" />
            <Entry Name ="gs" Order = "17" Size ="4" />

            <!-- Segment descriptor caches and TLS base MSRs -->
            <!--Entry Name ="cs_base" Order = "18" Size="8"/
            <Entry Name ="ss_base" Order = "18" Size ="8" />
            <Entry Name ="ds_base" Order = "19" Size ="8" />
            <Entry Name ="es_base" Order = "1a" Size ="8" /> -->
            <Entry Name ="fs_base" Order = "18" Size ="8" />
            <Entry Name ="gs_base" Order = "19" Size ="8" />
            <Entry Name ="k_gs_base" Order = "1a" Size ="8" />

            <!-- Control registers -->
            <!-- the cr0 register format fields:
            <flags id="x64_cr0" size="8">
            <field name="PG" start="31" end="31"/>
            <field name="CD" start="30" end="30"/>
            <field name="NW" start="29" end="29"/>
            <field name="AM" start="18" end="18"/>
            <field name="WP" start="16" end="16"/>
            <field name="NE" start="5" end="5"/>
            <field name="ET" start="4" end="4"/>
            <field name="TS" start="3" end="3"/>
            <field name="EM" start="2" end="2"/>
            <field name="MP" start="1" end="1"/>
            <field name="PE" start="0" end="0"/>
            </flags> -->
            <Entry Name ="cr0" Order = "1b" Size ="8" />
            <Entry Name ="cr2" Order = "1c" Size ="8" />

            <!-- the cr3 register format fields:
            <flags id="x64_cr3" size="8">
                <field name="PDBR" start="12" end="63"/>
                <field name="PCID" start="0" end="11"/>
            </flags> -->
            <Entry Name ="cr3" Order = "1d" Size ="8" />

            <!-- the cr4 register format fields:
            <flags id="x64_cr4" size="8">
                <field name="PKE" start="22" end="22"/>
                <field name="SMAP" start="21" end="21"/>
                <field name="SMEP" start="20" end="20"/>
                <field name="OSXSAVE" start="18" end="18"/>
                <field name="PCIDE" start="17" end="17"/>
                <field name="FSGSBASE" start="16" end="16"/>
                <field name="SMXE" start="14" end="14"/>
                <field name="VMXE" start="13" end="13"/>
                <field name="LA57" start="12" end="12"/>
                <field name="UMIP" start="11" end="11"/>
                <field name="OSXMMEXCPT" start="10" end="10"/>
                <field name="OSFXSR" start="9" end="9"/>
                <field name="PCE" start="8" end="8"/>
                <field name="PGE" start="7" end="7"/>
                <field name="MCE" start="6" end="6"/>
```

```
            <field name="PAE" start="5" end="5"/>
            <field name="PSE" start="4" end="4"/>
            <field name="DE" start="3" end="3"/>
            <field name="TSD" start="2" end="2"/>
            <field name="PVI" start="1" end="1"/>
            <field name="VME" start="0" end="0"/>
        </flags> -->
        <Entry Name ="cr4" Order = "1e" Size ="8" />
        <Entry Name ="cr8" Order = "1f" Size ="8" />

        <!-- the efer register format fields:
        <flags id="x64_efer" size="8">
        <field name="TCE" start="15" end="15"/>
        <field name="FFXSR" start="14" end="14"/>
        <field name="LMSLE" start="13" end="13"/>
        <field name="SVME" start="12" end="12"/>
        <field name="NXE" start="11" end="11"/>
        <field name="LMA" start="10" end="10"/>
        <field name="LME" start="8" end="8"/>
        <field name="SCE" start="0" end="0"/>
        </flags> -->
        <Entry Name ="efer" Order = "20" Size ="8"/>

        <!-- x87 FPU -->
        <Entry Name ="st0" Order = "21" Size ="10" />
        <Entry Name ="st1" Order = "22" Size ="10" />
        <Entry Name ="st2" Order = "23" Size ="10" />
        <Entry Name ="st3" Order = "24" Size ="10" />
        <Entry Name ="st4" Order = "25" Size ="10" />
        <Entry Name ="st5" Order = "26" Size ="10" />
        <Entry Name ="st6" Order = "27" Size ="10" />
        <Entry Name ="st7" Order = "28" Size ="10" />
        <Entry Name ="fctrl" Order = "29" Size ="4" />
        <Entry Name ="fstat" Order = "2a" Size ="4" />
        <Entry Name ="ftag"  Order = "2b" Size ="4" />
        <Entry Name ="fiseg" Order = "2c" Size ="4" />
        <Entry Name ="fioff" Order = "2d" Size ="4" />
        <Entry Name ="foseg" Order = "2e" Size ="4" />
        <Entry Name ="fooff" Order = "2f" Size ="4" />
        <Entry Name ="fop" Order = "30" Size ="4" />
        <Entry Name ="xmm0" Order = "31" Size ="16"  />
        <Entry Name ="xmm1" Order = "32" Size ="16"  />
        <Entry Name ="xmm2" Order = "33" Size ="16"  />
        <Entry Name ="xmm3" Order = "34" Size ="16"  />
        <Entry Name ="xmm4" Order = "35" Size ="16"  />
        <Entry Name ="xmm5" Order = "36" Size ="16"  />
        <Entry Name ="xmm6" Order = "37" Size ="16"  />
        <Entry Name ="xmm7" Order = "38" Size ="16"  />
        <Entry Name ="xmm8" Order = "39" Size ="16"  />
        <Entry Name ="xmm9" Order = "3a" Size ="16"  />
        <Entry Name ="xmm10" Order = "3b" Size ="16"  />
        <Entry Name ="xmm11" Order = "3c" Size ="16"  />
        <Entry Name ="xmm12" Order = "3d" Size ="16"  />
        <Entry Name ="xmm13" Order = "3e" Size ="16"  />
        <Entry Name ="xmm14" Order = "3f" Size ="16"  />
```

```
            <Entry Name ="xmm15" Order = "40" Size ="16"  />

            <!-- the mxcsr register format fields:
            <flags id="x64_mxcsr" size="4">
                <field name="IE" start="0" end="0"/>
                <field name="DE" start="1" end="1"/>
                <field name="ZE" start="2" end="2"/>
                <field name="OE" start="3" end="3"/>
                <field name="UE" start="4" end="4"/>
                <field name="PE" start="5" end="5"/>
                <field name="DAZ" start="6" end="6"/>
                <field name="IM" start="7" end="7"/>
                <field name="DM" start="8" end="8"/>
                <field name="ZM" start="9" end="9"/>
                <field name="OM" start="10" end="10"/>
                <field name="UM" start="11" end="11"/>
                <field name="PM" start="12" end="12"/>
                <field name="FZ" start="15" end="15"/>
            </flags> -->
            <Entry Name ="mxcsr" Order = "41" Size ="4" />

        </ExdiGdbServerRegisters>
    </ExdiGdbServerConfigData>
    </ExdiTarget>
    </ExdiTargets>
</ExdiTargets>
```

示例 EXDI PowerShell 脚本

此示例 PowerShell 脚本安装 EXDI，然后启动调试器。 Start-ExdiDebugger.ps1 脚本将根据需要安装 ExdiGdbSrv.dll，配置 xml 设置文件，检查用于运行 dllhost.exe 进程，并启动调试器以连接到已运行的 gdb 服务器硬件调试目标。

将用户界面与 WinDBg 一起使用时，已安装ExdiGdbSrv.dll，并且此脚本不相关。 有关使用内置用户界面的详细信息，请参阅 配置 EXDI 调试器传输。

这是调用启动脚本的示例。

PowerShell

```
PS>.\Start-ExdiDebugger.ps1 -ExdiTarget "QEMU" -GdbPort 1234 -Architecture
x64
```

如有必要，还可以指定生成的文件。

PowerShell

```
PS>.\Start-ExdiDebugger.ps1 -ExdiTarget "QEMU" -GdbPort 1234 -Architecture
```

```
x64 -ExdiDropPath "C:\path\to\built\exdi\files"
```

Start-ExdiDebugger.ps1 具有以下设置选项。

展开表

参数	说明
ExdiTarget	要连接到的目标类型。 这与设置 xml 文件中的特定部分相对应
HostName	托管 gdb 服务器会话的计算机的 IP 地址或主机名（默认为“LocalHost”）
GdbPort	Gdb 服务器正在侦听的端口。
体系结构	硬件调试目标的体系结构（此参数还表示 xml 设置文件中的 ArchitectureFamily 参数）
ExdiDropPath	ExdiGdbSrv.dll、exdiConfigData.xml 和 systemregisters.xml 文件的位置。 仅当 ExdiGdbSrv.dll 未安装或安装错误时，才会复制这些内容。
ExtraDebuggerArgs	可在调试器命令行上传递的额外参数
PreNTAppDebugging	将 heuristicScanSize 的值从 0xfffe（最适合 NT）更改为 0xffe（适用于 pre-N 应用）
DontTryDllHostCleanup	检查 dllhost.exe 中 ExdiGdbSrv.dll 的现有运行的实例需要提升级别。 提供此开关将允许脚本在没有提升的情况下运行（尽管调试器可能无法正常运行）。

要启用要显示的数据包，请将 *displayCommPackets* 的值设置为“yes”。

PowerShell

```
    [pscustomobject]@{ Path = 'displayCommPackets'
; value = "yes" }
```

PowerShell

```
.\Start-ExdiDebugger.ps1 -ExdiTarget "QEMU" -GdbPort 1234 -Architecture x64
-ExdiDropPath "C:\path\to\built\exdi\files"
```

有关更多设置选项，请参阅代码注释。

PowerShell

```
<#
.Synopsis
    Installs and launches exdi debugger (automating xml file editing)
```

```
.Description
    This script will install ExdiGdbSrv.dll if required, configure the xml
settings
    files, check for running dllhost.exe processes, and launch the debugger
to connect to
    an already running gdb server hardware debugging target.

.Parameter ExdiTarget
    Type of target to connect to. This corresponds to a specific section in
the settings xml file

.Parameter HostName
    IP address or hostname of the computer hosting the gdb server session
(defaults to "LocalHost")

.Parameter GdbPort
    Port that the gdb server is listening on.

.Parameter Architecture
    Architecture of the hardware debugging target (this parameter also
implies the ArchitectureFamily
    parameter in the xml settings file)

.Parameter ExdiDropPath
    Location of the ExdiGdbSrv.dll, exdiConfigData.xml, and
systemregisters.xml files. These will
    only be copied if ExdiGdbSrv.dll is not installed or is installed
incorrectly.

.Parameter ExtraDebuggerArgs
    Extra arguments to pass on the debugger command line

.Parameter PreNTAppDebugging
    Changes the value of the heuristicScanSize from 0xfffe (best for NT) to
0xffe (for pre-NT Apps)

.Parameter DontTryDllHostCleanup
    Checking for existing running instances of ExdiGdbSrv.dll in dllhost.exe
requires elevation.
    Providing this switch will allow the script to run without elevation
(although the debugger may not
    function correctly).

.Example
    >---------------- (first run) ------------<
    .\Start-ExdiDebugger.ps1 -ExdiTarget "QEMU" -GdbPort 1234 -Architecture
x64 -ExdiDropPath "C:\path\to\built\exdi\files"

.Example
    PS>.\Start-ExdiDebugger.ps1 -ExdiTarget "QEMU" -GdbPort 1234 -
Architecture x64
#>

[CmdletBinding()]
```

```
param
(
    [ValidateSet("QEMU")]
    [string]
    $ExdiTarget = "QEMU",

    [string]
    $HostName = "LocalHost",

    [Parameter(Mandatory=$true)]
    [Int]
    $GdbPort,

    [Parameter(Mandatory=$true)]
    [string]
    [ValidateSet("x86", "x64", "arm64")]
    $Architecture,

    [string]
    $ExdiDropPath,

    [string]
    $DebuggerPath,

    [string[]]
    $ExtraDebuggerArgs = @(),

    [switch]
    $PreNTAppDebugging,

    [switch]
    $DontTryDllHostCleanup
)

$ErrorActionPreference = "Stop"

#region Functions

Function Test-Admin
{
    ([Security.Principal.WindowsPrincipal]
[Security.Principal.WindowsIdentity]::GetCurrent()).IsInRole([Security.Princ
ipal.WindowsBuiltInRole]"Administrator")
}

Function Find-PathToWindbgX
{
    $InternalWindbgXPath = "$env:LOCALAPPDATA\DBG\UI\WindbgX.exe"
    $ExternalWindbgXPath =
"$env:LOCALAPPDATA\Microsoft\WindowsApps\WinDbgX.exe"

    if (Test-Path $InternalWindbgXPath -PathType Leaf)
    {
        return $InternalWindbgXPath
    }
```

```
    elseif (Test-Path $ExternalWindbgXPath -PathType Leaf)
    {
        return $ExternalWindbgXPath
    }
}

Function Test-ParameterValidation
{
    $CommandName = $PSCmdlet.MyInvocation.InvocationName
    $ParameterList = (Get-Command -Name $CommandName).Parameters

    foreach ($Parameter in $ParameterList) {
        Get-Variable -Name $Parameter.Values.Name -ErrorAction
SilentlyContinue | Out-String | Write-Verbose
    }

    if (-not $DebuggerPath)
    {
        throw "WindbgX is not installed"
    }
    elseif (-not (Test-Path $DebuggerPath -PathType Leaf))
    {
        throw "DebuggerPath param ($DebuggerPath) does not point to a
debugger."
    }

    # Searching for loaded instances of ExdiGdbSrv.dll in dllhost.exe
requires elevation
    if (-not $DontTryDllHostCleanup -and
        -not $(Test-Admin))
    {
        throw "Searching for loaded instances of ExdiGdbSrv.dll in
dllhost.exe requires elevation. Run with the -DontTryDllHostCleanup
parameter to skip this check (debugger session init may fail)."
    }
}

Function Get-ExdiInstallPath
{
    Get-ItemPropertyValue -Path "Registry::HKEY_CLASSES_ROOT\CLSID\
{29f9906e-9dbe-4d4b-b0fb-6acf7fb6d014}\InProcServer32" -Name "(default)" -
ErrorAction SilentlyContinue
}

Function Test-ExdiServerInstalled
{
    # Check registration of exdi server class
    if ($(Get-ExdiInstallPath) -ne $null -and $(Test-Path "$(Get-
ExdiInstallPath)"))
    {
        Write-Verbose "Exdi server is installed. Checking installation..."
        $ExdiInstallDir = [System.IO.Path]::GetDirectoryName($(Get-
ExdiInstallPath))
        if (-not (Test-Path $ExdiInstallDir))
        {
```

```
            Write-Host "Currently Registered exdi server does not exist.
Reinstalling..."
            return $false
        }
        elseif (-not ((Test-Path "$ExdiInstallDir\exdiConfigData.xml") -and
(Test-Path "$ExdiInstallDir\systemregisters.xml")))
        {
            Write-Host "Currently Registered exdi server does not have
required xml settings files. Reinstalling..."
            return $false
        }
        else
        {
            Write-Verbose "Exdi server is insalled correctly. Skipping
installation..."
            return $true
        }
    }
    else
    {
        Write-Host "Exdi server is not installed. Installing..."
        return $false
    }
}

Function Install-ExdiServer
{
    [CmdletBinding()]
    param
    (
        [string] $InstallFrom,
        [string] $InstallTo
    )

    if (-not $(Test-Admin))
    {
        throw "Script needs to be run as an Admin to install exdi software."
    }

    New-Item -ItemType Directory $InstallTo -ErrorAction SilentlyContinue |
Write-Verbose
    Copy-Item -Path "$InstallFrom\ExdiGdbSrv.dll" -Destination $InstallTo -
ErrorAction stop | Write-Verbose
    Copy-Item -Path "$InstallFrom\exdiConfigData.xml" -Destination
$InstallTo -ErrorAction stop | Write-Verbose
    Copy-Item -Path "$InstallFrom\systemregisters.xml" -Destination
$InstallTo -ErrorAction stop | Write-Verbose
    regsvr32 /s "$InstallTo\ExdiGdbSrv.dll"

    if ($(Get-ExdiInstallPath) -eq $null)
    {
        throw "Unable to install exdi server"
    }
}
```

```
Function Edit-ExdiConfigFile
{
    [CmdletBinding()]
    param
    (
        [string] $ExdiFilePath,
        [string] $ExdiTargetType,
        [PSCustomObject[]] $XmlSettingPathValueList
    )

    # Edit exdiConfigData.xml
    [xml]$exdiConfigXml = Get-Content "$ExdiFilePath"

    # Set current target
    $exdiConfigXml.ExdiTargets.CurrentTarget = $ExdiTarget

    # set HostNameAndPort
    $ExdiTargetXmlNode =
$exdiConfigXml.SelectSingleNode("//ExdiTargets/ExdiTarget[@Name='$ExdiTarget
']/ExdiGdbServerConfigData")

    foreach ($XmlSettingPathValue in $XmlSettingPathValueList)
    {
        Write-Verbose "Processing $XmlSettingPathValue"
        if ($XmlSettingPathValue.Value -eq $null)
        {
            continue
        }

        $PathParts = $XmlSettingPathValue.Path.Split(".")
        $curNode = $ExdiTargetXmlNode
        if ($PathParts.Count -gt 1)
        {
            foreach ($PathPart in $PathParts[0..($PathParts.Count-2)])
            {
                Write-Verbose $PathPart
                $curNode = $curNode.($PathPart)
            }
        }
        $curNode.($PathParts[-1]) = $XmlSettingPathValue.Value
    }

    $exdiConfigXml.Save("$ExdiFilePath")
}

Function Stop-ExdiContainingDllHosts
{
    $DllHostPids = Get-Process dllhost | ForEach-Object { $_.Id }
    foreach ($DllHostPid in $DllHostPids)
    {
        $DllHostExdiDlls = Get-Process -Id $DllHostPid -Module | Where-
Object { $_.FileName -like "*ExdiGdbSrv.dll" }
        if ($DllHostExdiDlls.Count -ne 0)
        {
            Write-Verbose "Killing dllhost.exe with pid $DllHostPid
```

```
(Contained instance of ExdiGdbSrv.dll)"
            Stop-Process -Id $DllHostPid -Force
        }
    }
}

#endregion

#region Script

# Apply defaults for $DebuggerPath before Parameter validation
if (-not $DebuggerPath)
{
    $DebuggerPath = Find-PathToWindbgX
}

Test-ParameterValidation

# look clean up dllhost.exe early since it can hold a lock on files which
# need to be overwritten
if (-not $DontTryDllHostCleanup)
{
    Stop-ExdiContainingDllHosts
}

if (-not $(Test-ExdiServerInstalled))
{
    if (-not $ExdiDropPath)
    {
        throw "ExdiServer is not installed and -ExdiDropPath is not valid"
    }

    $ExdiInstallDir = Join-Path -Path
"$([System.IO.Path]::GetDirectoryName($DebuggerPath))" -ChildPath "exdi"
    Install-ExdiServer -InstallFrom "$ExdiDropPath" -InstallTo
"$ExdiInstallDir"
}

$SystemRegistersFilepath = Join-Path -Path
"$([System.IO.Path]::GetDirectoryName($(Get-ExdiInstallPath)))" -ChildPath
"systemregisters.xml"
$ExdiConfigFilepath      = Join-Path -Path
"$([System.IO.Path]::GetDirectoryName($(Get-ExdiInstallPath)))" -ChildPath
"exdiConfigData.xml"

# Calculate implied parameters
$HeuristicScanSize = if ($PreNTAppDebugging) { "0xffe" } else { "0xfffe" }
$ArchitectureFamily = switch($Architecture)
{
    x64   { "ProcessorFamilyx64" }
    x86   { "ProcessorFamilyx86" }
    arm64 { "ProcessorFamilyARM64" }
}

# Path is evaluated relative to the relevant ExdiTarget's
```

```
ExdiGdbServerConfigData node in the xml schema
$SettingsToChange = @(
    [pscustomobject]@{ Path =
'GdbServerConnectionParameters.Value.HostNameAndPort' ; Value =
"${HostName}:$GdbPort" },
    [pscustomobject]@{ Path = 'ExdiGdbServerTargetData.targetArchitecture'
; Value = "$Architecture" },
    [pscustomobject]@{ Path = 'ExdiGdbServerTargetData.targetFamily'
; Value = "$ArchitectureFamily" },
    [pscustomobject]@{ Path = 'ExdiGdbServerTargetData.heuristicScanSize'
; Value = "$HeuristicScanSize" },
    [pscustomobject]@{ Path = 'displayCommPackets'
; value = "no" }
)
Edit-ExdiConfigFile -ExdiFilePath "$ExdiConfigFilepath" -ExdiTargetType
"$ExdiTarget" -XmlSettingPathValueList $SettingsToChange

# Set env vars for debugger
[System.Environment]::SetEnvironmentVariable('EXDI_GDBSRV_XML_CONFIG_FILE',"
$ExdiConfigFilepath")
[System.Environment]::SetEnvironmentVariable('EXDI_SYSTEM_REGISTERS_MAP_XML_
FILE',"$SystemRegistersFilepath")

$DebuggerArgs = @("-v", "-kx exdi:CLSID={29f9906e-9dbe-4d4b-b0fb-
6acf7fb6d014},Kd=Guess,DataBreaks=Exdi")
Write-Verbose "DebuggerPath = $DebuggerPath"
Start-Process -FilePath "$DebuggerPath" -ArgumentList ($DebuggerArgs +
$ExtraDebuggerArgs)

#endregion
```

另请参阅

配置 EXDI 调试器传输

使用 EXDI 设置 QEMU 内核模式调试

.exdicmd（EXDI 命令）

自动设置 KDNET 网络内核调试

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用 EXDI 设置 QEMU 内核模式调试

项目 • 2024/11/01

本主题介绍如何在 Windows 调试器中使用 EXDI 设置 QEMU 内核模式调试。

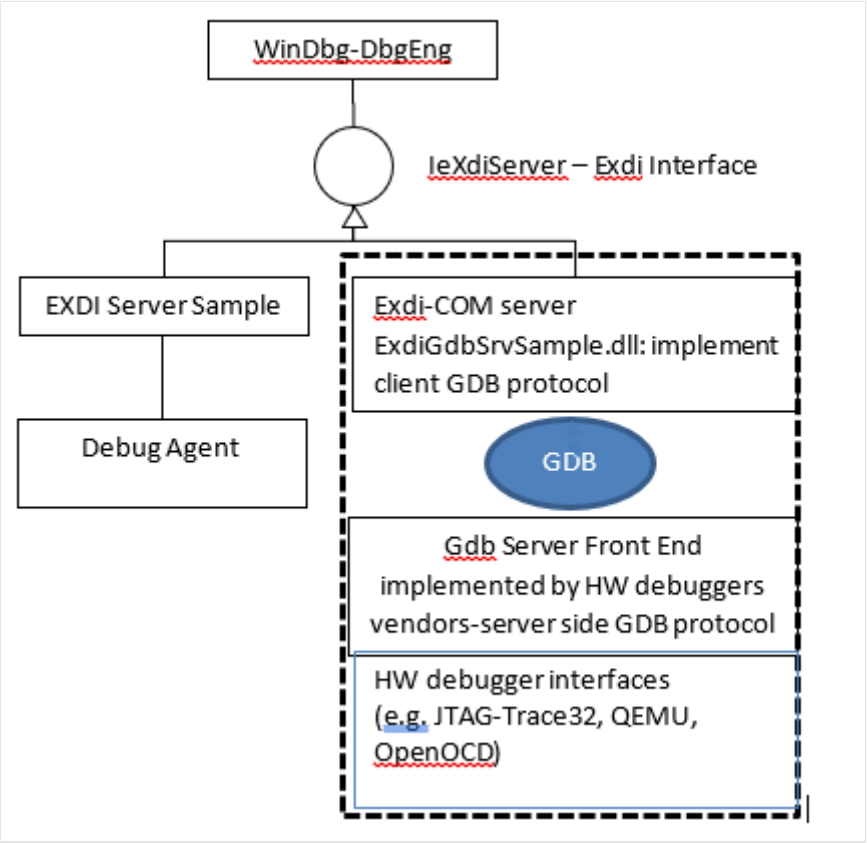
有关设置配置 EXDI 连接和故障排除的一般信息，请参阅配置 EXDI 调试器传输。

使用 QEMU（虚拟化和计算机仿真软件），可以连接到充当主机的其他操作系统，例如 Linux。 QEMU 本身可以在许多体系结构上运行，例如 x64 和 Arm64。 ExdiGdb 调试服务器还支持其他处理器，例如，可以使用 WinDbg 调试 x64 上运行的 QEMU，从而模拟 Arm64。 使用 EXDI 还允许在启动过程中尽早调试 VM，甚至在 OS 加载之前也是如此。

> **备注**
>
> EXDI 是一种针对特定环境的高级专业调试方式。 使用标准 KDNET 连接更容易配置，因此建议使用。 要自动设置网络调试，请参阅**设置 KDNET 网络内核自动调试**。

EXDI COM 服务器

EXDI 是一个接口，可通过添加对硬件调试器（如基于 JTAG 或基于 GdbServer）的支持来扩展 WinDbg。 下图说明了 EXDI-GdbServer 的作用。



> **ⓘ 重要**
>
> 由于 EXDI 未使用 KDNET 协议，因此连接的调试器所掌握的有关电脑上运行情况的信息要少得多，许多命令的运行方式也会不同，甚至根本无法运行。 访问被调试代码的专用符号有助于调试器更好地理解目标系统的代码执行。 有关详细信息，请参阅**公共符号和专用符号**。

在 QEMU 上设置与 Windows 映像的调试器连接

这些步骤介绍如何附加到 Windows x64 虚拟机，将 GDB 服务器公开给 Windbg 客户端（使用 EXDI COM 服务器）也在 Windows 上运行。 使用 WinDbg ExdiGdbSrv.dll（GDB 服务器客户端）和 QEMU GDB 服务器之间的 GdbServer RSP 会话。

1. 在 Windows 上下载并安装 QEMU。
2. 配置目标 QEMU 虚拟 Windows 映像，以使用所需的网络和 BIOS/UEFI 设置进行调试。
3. 使用启动脚本启动 QEMU 环境。
4. 在 QEMU 上启动 GdbServer。
5. 检查网络连接，找到并记录目标图像 IP 地址。 （LocalHost 的主机 IP 默认地址和端口为 1234）。
6. 在主机系统中下载并安装 Windows 调试工具。
7. 使用命令行或 UI 启动 WinDbg 以连接到 EXDI 服务器。
8. 使用 WinDbg 来调试目标 QEMU Windows 映像。

QEMU 开源计算机模拟器

QEMU 是一个通用的开源计算机模拟器和虚拟化程序，用于进行动态转换。 当 QEMU 用作计算机模拟器时，它可以在不同的计算机上（x64 电脑）上运行为一种处理器（如 Arm64）开发的操作系统和程序。 它还可以运行/托管不同操作系统 (Windows/Linux/Mac) 的虚拟机镜像。

QEMU 可以与其他虚拟机监控程序（如 KVM）一起操作，以使用 CPU 扩展（HVM）进行虚拟化。 将 QEMU 用作虚拟化程序时，QEMU 可直接在主机 CPU 上执行来宾代码，从而实现接近本地的性能。 QEMU 可利用 OS 虚拟机监控程序的功能将 CPU 和 MMU 仿真卸载到实际硬件上。

下载并安装 QEMU

在本演练中，适用于 Windows x64 的 QEMU 将安装在一台 x64 电脑上，而 Windows 调试器也将在这台电脑上运行。

从 QEMU 下载页面下载 QEMU：https://www.qemu.org/download/

有关安装 QEMU 的信息，请参阅 QEMU 文档：https://www.qemu.org/documentation/

配置目标虚拟磁盘

找到或创建一个包含要调试的软件的虚拟磁盘镜像。

在本例中，将使用 Windows x64 VHDX 虚拟机磁盘映像。 要了解有关 Windows 虚拟机映像的详细信息，请参阅在 Windows 10 上使用 Hyper-V 创建虚拟机。

在 Windows 映像中注入 VirtIO 驱动程序

要实现网络功能和合理的存储设备性能，可在 Windows 虚拟机磁盘镜像中注入或安装 VirtIO 驱动程序。 此处提供了 VirtIO 驱动程序： https://github.com/virtio-win/kvm-guest-drivers-windows

VirtIO 是一个标准化接口，允许虚拟机访问抽象硬件，如块设备、网络适配器和控制台。 VirtIO 充当虚拟化环境中硬件设备的抽象层，如 QEMU。

若要将 VirtIO 驱动程序注入 Windows 映像，请执行以下步骤：

1. 例如 `C:\VirtIo_Drivers`，提取文件夹中的 Virtlo 驱动程序。
2. 双击文件资源管理器中的 VHDX（也可以使用 diskpart），装载包含 Windows x64 虚拟机的 VHDX。 Windows 将使用特定字母装载 VHDX，例如“L：”
3. 使用 Dism 在装载的映像中注入驱动程序： `dism /image:L: /Add-Driver /driver:C:\VirtIo_Drivers` 有关 DISM 的详细信息，请参阅 DISM 概述。
4. 该过程完成后，可以卸载映像，并继续将 VHDX 转换为 QEMU。

将 VHDX 转换为 QEMU

这一步并非必需但建议使用，因为使用本地 QEMU QCOW 映像而不是 VHDX 可以获得更好的性能。

使用以下 qemu-img.exe 命令转换 vhdx。 此实用工具位于安装 QEMU 的位置，例如 `C:\Program Files\qemu`。

Console

```
C:\Program Files\qemu> qemu-img convert -c -p -O qcow2 MyVHDXFile.vhdx
MyQEMUFile.qcow2
```

下载 UEFI 固件

为获得最佳效果，请下载或编译 UEFI 固件文件 (OVMF.fd)。 之所以需要固件，是因为 QEMU 默认情况下会模拟旧版 BIOS 系统。

UEFI 固件的一个来源是 Open Clear Linux 项目：https://clearlinux.org/

UEFI `OVMF.fd` 文件示例可在此处获取：
https://github.com/clearlinux/common/tree/master/OVMF.fd

提取 `C:\Program Files\qemu\Firmware` 中下载文件的内容。

对于 Intel AMD64 以外的平台，应从 EDK2 中编译固件。 有关详细信息，请参阅 https://github.com/tianocore/tianocore.github.io/wiki/How-to-build-OVMF 。

配置 QEMU 启动脚本

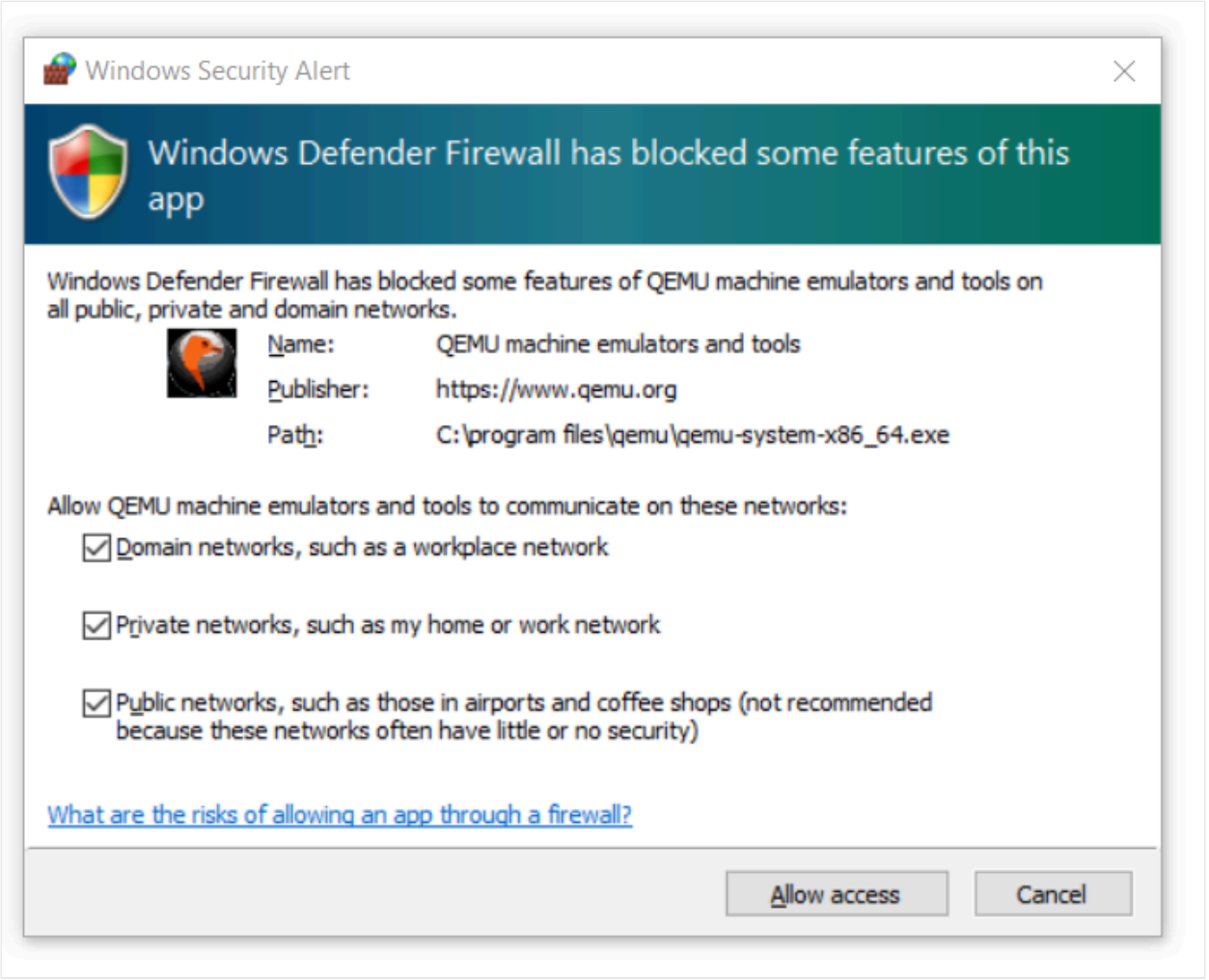
在 QEMU 中创建配置文件。 例如，在 QEMU 根目录下创建一个 `StartQEMUx64Windows.bat` 文件。 请参阅以下示例文件。

使用 QEMU 启动脚本启动 QEMU

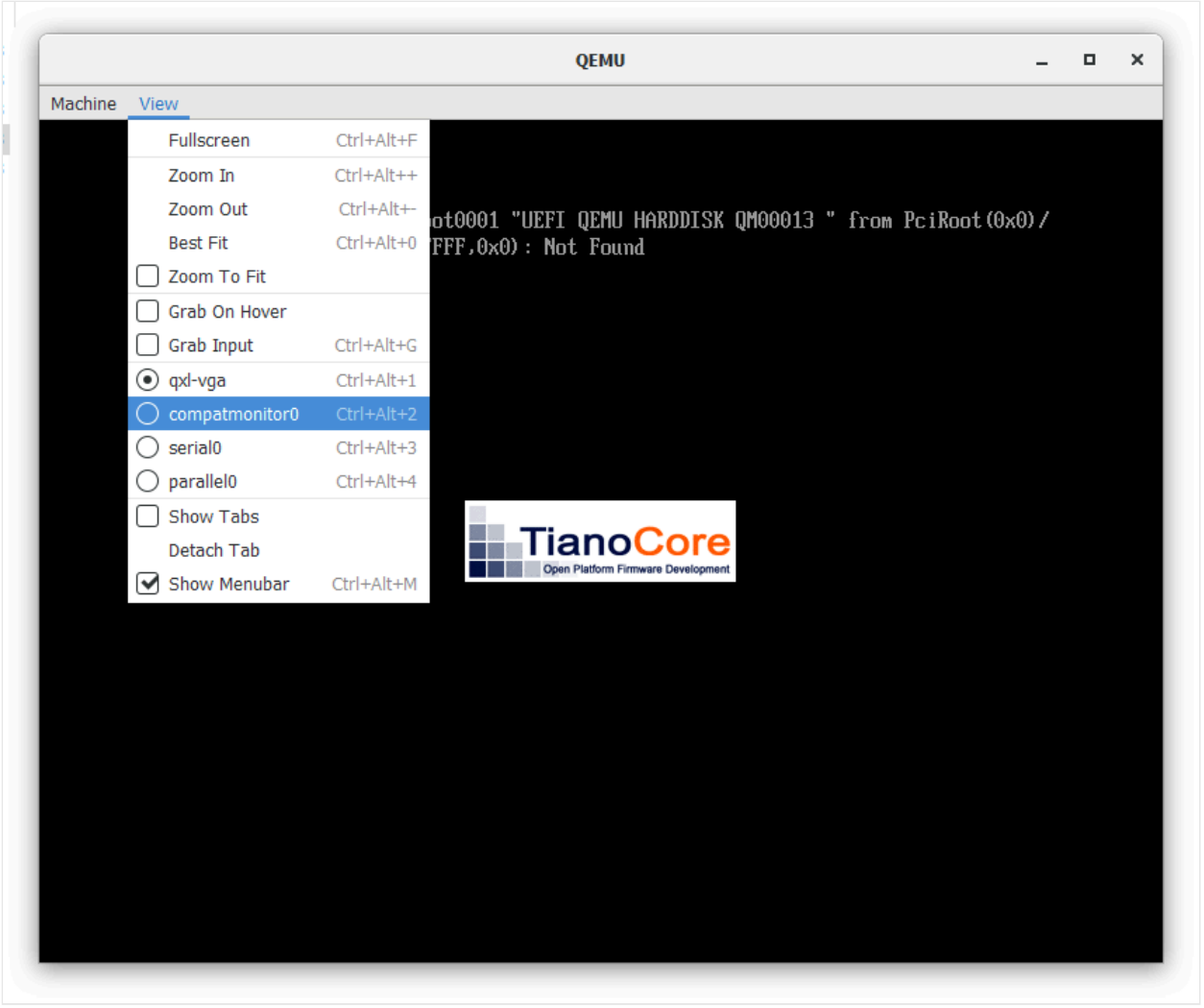
执行 QEMU 启动脚本以启动 QEMU。

```bat
c:\Program Files\qemu\StartQEMUx64Windows.bat
```

如果出现防火墙防御程序提示，请授予该程序对所有类型网络的所有权限，以便通过调试器主机的 Windows 防火墙启用 Windbg。



一旦 Windows 虚拟机在 QEMU 环境中启动，QEMU UI 就会出现。



使用 CTRL+ALT+ 数字组合键进入 QEMU 监控控制台。 使用 *View->compatmonitor* 也可以使用此监视器。

键入 `gdbserver` 以便在 QEMU 上启动前端 GDB 服务器。

QEMU 应显示 `Waiting for gdb connection on device ‘tcp::1234’`

使用 CTRL+ALT+1 组合键返回主窗口。

提示：GDB 控制台窗口支持 `system_reset` 命令快速重启仿真。 键入 `help` GDB 控制台命令列表。

QEMU x64 Windows VM 启动脚本示例

下面是可用于 AMD64 虚拟机的 QEMU 配置脚本示例。 将指向 DISK 和 CDROM 文件的链接替换为电脑上的位置。

bat

```
    REM
    REM  This script is used to run a Windows x64 VM on QEMU that is hosted
by a Windows x64 host system
```

```
    REM  The Host system is a PC with Intel(R) Xeon(R) CPU.
    REM
    set EXECUTABLE=qemu-system-x86_64
    set MACHINE=-m 6G -smp 4

    REM No acceleration
    REM generic cpu emulation.
    REM to find out which CPU types are supported by the QEMU version on
your system, then run:
    REM  qemu-system-x86_64.exe -cpu help
    REM the see if your host system CPU is listed
    REM

    set CPU=-machine q35

    REM Enables x64 UEFI-BIOS that will be used by QEMU :
    set BIOS=-bios "C:\Program Files\qemu\Firmware\OVMF.fd"

    REM  Use regular GFX simulation
    set GFX=-device ramfb -device VGA
    set USB_CTRL=-device usb-ehci,id=usbctrl
    set KEYB_MOUSE=-device usb-kbd -device usb-tablet

    REM # The following line enable the full-speed HD controller (requires
separate driver)
    REM # Following line uses the AHCI controller for the Virtual Hard Disk:
    set DRIVE0=-device ahci,id=ahci -device ide-hd,drive=disk,bus=ahci.0

    REM
    REM This will set the Windows VM x64 disk image that will be launched by
QEMU
    REM The disk image is in the qcow2 format accepted by QEMU.
    REM You get the .qcow2 image, once you get the VHDX Windows VM x64 image
    REM and apply the script to inject the virtio x64 drivers and then run
the
    REM the QEMU tool to convert the .VHDX image to .qcow2 format
    REM     i.e.
    REM qemu-img convert -c -p -O qcow2
Windows_VM_VHDX_with_injected_drivers_file.vhdx file.qcow2
    REM file : points to the specified qcow2 image path.
    REM
    set DISK0=-drive id=disk,file="C:\Program
Files\qemu\MyQEMUFile.qcow2",if=none

    REM
    REM for kdnet on, then best option:
    REM   NETWORK0="-netdev
user,id=net0,hostfwd=tcp::53389-:3389,hostfwd=tcp::50001-:50001 -device
virtio-net,netdev=net0,disable-legacy=on"
    REM
    REM Create a mapping for the RDP service from port 3389 to 3589.
    REM
    set NETHOST=-netdev user,id=net0,hostfwd=tcp::3589-:3389
    set NETGUEST=-device e1000,netdev=net0
```

```
    REM # The following line should enable the Daemon (instead of
interactive)
    set DAEMON=-daemonize"
    %EXECUTABLE% %MACHINE% %CPU% %BIOS% %GFX% %USB_CTRL% %DRIVE0% %DISK0%
%NETHOST% %NETGUEST%
```

网络连接

本地主机

如果 GDB 服务器已正确启动，则会看到 GDB 服务器将侦听的端口号，并且需要使用此端口来设置主机调试器 `IP:Port` 对。

如果主机调试器位于托管 QEMU 来宾的同一台计算机上，则会在 IP：端口对中使用 Localhost 标识符。 在此示例中，将使用同一台电脑上 `LocalHost:1234` 的服务器和主机调试器。

远程主机

如果在远程电脑上工作，请找到 Windows IP 地址（如果调试器主机会话不会与 QEMU VM 位于同一 Windows 计算机）。

目标 QEMU IP `<address>`： `<port number>` 将在 EXDI UI 中配置。

可在 QEMU 控制台 (compatmonitor0) 上发出以下命令，以显示有关网络和连接状态的信息。

控制台

```
info network
info usernet
```

有关 QEMU 网络连接的详细信息，请参阅 https://wiki.qemu.org/Documentation/Networking 。

在主机系统中下载并安装 Windows 调试工具

在主机系统上安装 Windows 调试工具。 有关下载和安装调试器工具的信息，请参阅适用于 Windows 的调试工具。

在主机系统上启动 WinDbg

在此处所述的方案中，在 EXDI UI 中设置以下选项进行连接。

目标类型 - *QEMU*

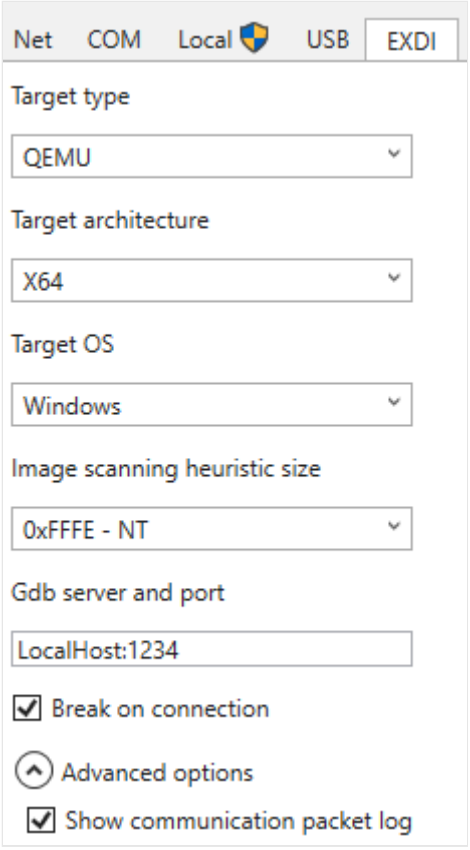
目标体系结构 - *x64*

目标 OS - *Windows*

图像扫描启发式大小 - *0xFFE - NT*

Gdb 服务器和端口 - *LocalHost：1234*

连接中断 - *是*



尽管建议使用 EXDI UI，但也可以使用类似于此处所示的命令行选项启动 WinDbg。

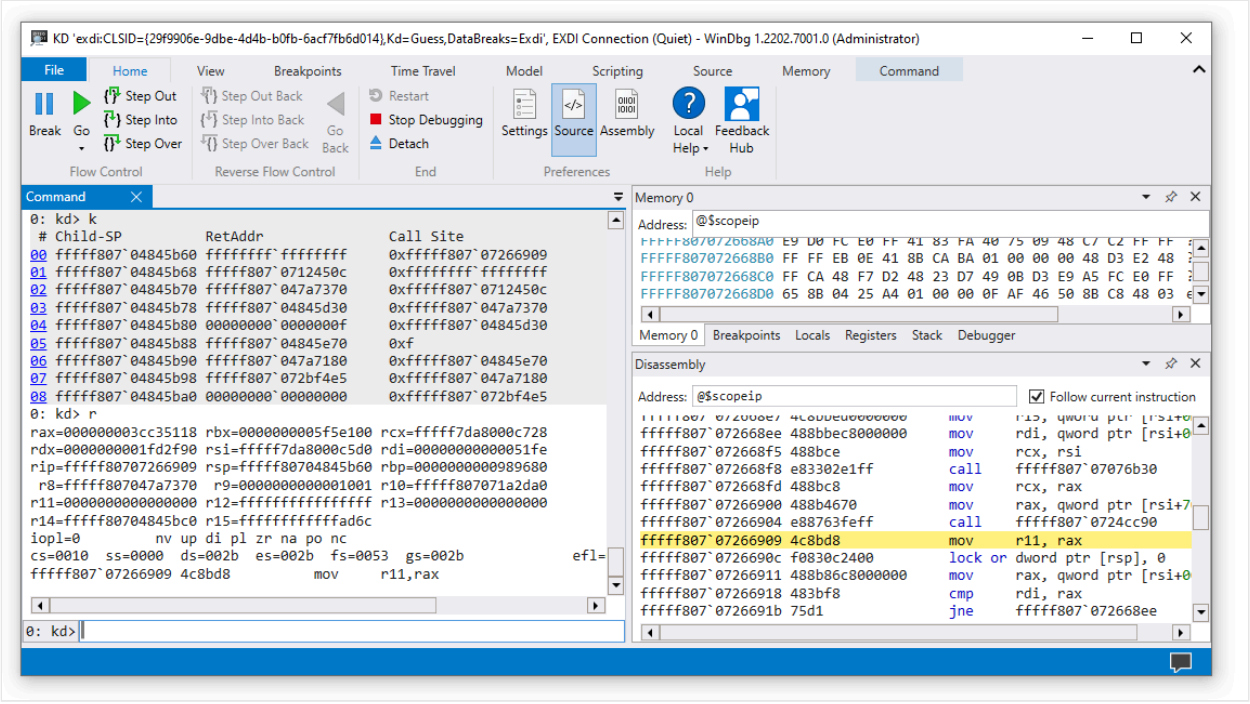
dbgcmd

```
c:\Debuggers> windbg.exe -v -kx exdi:CLSID={29f9906e-9dbe-4d4b-b0fb-
6acf7fb6d014},Kd=Guess,Inproc=ExdiGdbSrv.dll,DataBreaks=Exdi
```

使用命令行时，将使用 exdiConfigData.xml 文件配置 IP 地址和端口。 有关详细信息，请参阅 EXDI XML 配置文件。

要显示更多输出，可使用 **-v**: 详细会话。 有关 WinDbg 选项的一般信息，请参阅 WinDbg 命令行选项。

调试器应启动并连接至 QEMU GdbServer。



调试器将显示 EXDI 传输初始化成功。

dbgcmd

```
EXDI: DbgCoInitialize returned 0x00000001
EXDI: CoCreateInstance() returned 0x00000000
EXDI: QueryInterface(IExdiServer3) returned 0x00000000
Target command response: QEMU
exdiCmd: The function: 'ExdiDbgType' was completed.
EXDI: Server::GetTargetInfo() returned 0x00000000
EXDI: Server::SetKeepaliveInterface() returned 0x00000000
EXDI: Server::GetNbCodeBpAvail() returned 0x00000000
EXDI: ExdiNotifyRunChange::Initialize() returned 0x00000000
EXDI: LiveKernelTargetInfo::Initialize() returned 0x00000000
EXDI: Target initialization succeeded
```

EXDIGdbServer 控制台数据包窗口还可以显示有关 EXDI 连接状态的信息，如果*“*在“高级选项下显示通信数据包日志*”设置为”打开”。 有关详细信息，请参阅配置 EXDI 调试器传输中的故障排除信息。

使用 WinDbg 来调试目标 QEMU Windows 映像

dbgeng.dll 使用启发式算法来查找中断命令发生时 NT 基本加载地址的位置。 如果没有专用符号，此过程将失败。

这意味着在许多连接序列下，中断将无法发挥预期的功能。 如果手动侵入代码，它将是 Windows 当时正在执行的一个随机位置。 由于可能无法获得目标代码的符号，因此很难使用符号设置断点。

可用的调试器内存访问命令

以下直接访问内存的命令可以正常使用。

k、kb、kc、kd、kp、kP、kv（显示堆栈回溯）

r（寄存器）

d、da、db、dc、dd、dD、df、dp、dq、du、dw（显示内存）

u（取消汇编）

还可以通过代码进行操作。

p（步进）

还可以使用一些命令来尝试查找要调试的代码。

s（搜索内存）

.imgscan（查找映像标头）

Imgscan 对 EDXI 调试很有帮助，因为与传统的基于 KDNET 的内核调试不同，它可能无法根据符号设置断点。 定位所需的目标图像，可以方便地使用其位置来设置内存访问断点。

.exdicmd（EXDI 命令）

.exdicmd 使用活动 EXDI 调试连接向目标系统发送 EXDI 命令。 有关详细信息，请参阅 .exdicmd（EXDI 命令）。

故障排除

请参阅配置 EXDI 调试器传输中的故障排除信息。

另请参阅

配置 EXDI 调试器传输

EXDI XML 配置文件

.exdicmd（EXDI 命令）

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Linux 实时远程进程调试

项目 • 2024/05/27

本文介绍如何建立与 Linux 的实时 WinDbg 连接。 在 Linux 上进行实时远程进程调试需要使用 WinDbg 1.2402.24001.0 或更高版本。

GNU 调试器 GDBServer 用于在 Linux 上支持 WinDbg 连接。 有关 GDBServer 的详细信息，请参阅 https://en.wikipedia.org/wiki/Gdbserver。 提供查看远程 gdb 调试文档的一站式位置是 - https://sourceware.org/gdb/current/onlinedocs/gdb#Remote-Debugging

以下示例使用的适用于 Linux 的 Windows 子系统 (WSL)，但也可以使用其他 Linux 实现。

WinDbg 远程进程调试类型

使用 WinDbg 进行远程调试有两种主要方法 - **进程服务器**或 **KD 连接服务器**。 进程服务器用于用户模式调试；KD 连接服务器用于内核模式调试。 有关这些 WinDbg 连接类型的一般信息，请参阅进程服务器（用户模式）和KD 连接服务器（内核模式）。

有两种方法可以开始调试 Linux 用户模式进程。 可以在特定进程上启动 gdbserver，也可以将 gdbserver 作为*进程服务器*启动，后者可以列出并附加到现有进程。 这与 Windows 上的 DbgSrv (dbgsrv.exe) 进程服务器非常类似。 有关详细信息，请参阅激活进程服务器。

用户模式 Linux 进程调试

可以连接到特定的单用户模式进程，或者在多模式下，在列表中查看所有进程并选择一个进行连接。 本主题介绍这两种方法。 下面介绍了这两种方法共享相同的连接字符串语法。

gdbserver 连接字符串的格式

用于连接 gdbserver 的格式是“protocol:arguments”，其中 arguments 是以逗号分隔的“argument=value”列表。 对于用户模式的 gdbserver 连接，协议是 gdb，参数集如下所示。

`server=<address>` - 必填：指明要连接的 gdbserver 的 IP 地址。

`port=<port>` - 必填：指明要连接到的 gdbserver 的端口号。

`threadEvents=<true|false>` - 可选：指明此版本的 gdbserver 的线程事件在停止模式下是否正常工作。

当前的 gdbserver 版本中存在一个问题，即在服务器处于停止模式（WinDbg 使用的模式）时启用线程事件会导致 gdbserver 崩溃。 如果此值为 false（默认值），则线程启动和停止事件将被合成，但可能会比线程创建/销毁的实际时间晚很多。 当 gdbserver 中的修复程序可用时，可通过此选项启用实际事件。

连接到单个用户模式进程

本部分将介绍如何使用 WinDbg 在 Linux 中识别和连接单个用户模式进程。

WSL（适用于 Linux 的 Windows 子系统）

以下示例使用的 WSL（适用于 Linux 的 Windows 子系统），但也可以使用其他 Linux 实现。 有关设置环境和使用 WSL 的信息，请参阅：

- 如何使用 WSL 在 Windows 上安装 Linux
- 设置 WSL 开发环境
- 使用 WSL 访问网络应用程序
- WSL 的 Intune 设置

选择所需进程

在 Linux 中使用 `ps -A` 命令列出进程，以确定要连接的运行中进程。

dbgcmd

```
user1@USER1:/mnt/c/Users/USER1$ ps -A
    PID TTY          TIME CMD
    458 pts/1    00:00:00 bash
    460 ?        00:00:00 rtkit-daemon
    470 ?        00:00:00 dbus-daemon
    482 ?        00:00:19 python3
   1076 ?        00:00:00 packagekitd
   1196 pts/0    00:00:00 ps
```

在本示例演练中，我们将连接到 python3。

找到目标系统 IP 地址

如果连接到远程 linux 目标计算机，请使用 `ip route show` 等命令确定外部 IP 地址。

```linux
user1@USER1:/mnt/c/Users/USER1$ ip route show
default via 192.168.1.1 dev enp3s0 proto dhcp metric 100
172.25.144.0/24 dev enp3s0 proto kernel scope link src 192.168.1.107 metric
100
```

在本演练中，我们将连接到在同一台电脑上运行的 WSL，并使用 *localhost* 的 IP 地址。

将 GDBServer 附加到所选进程

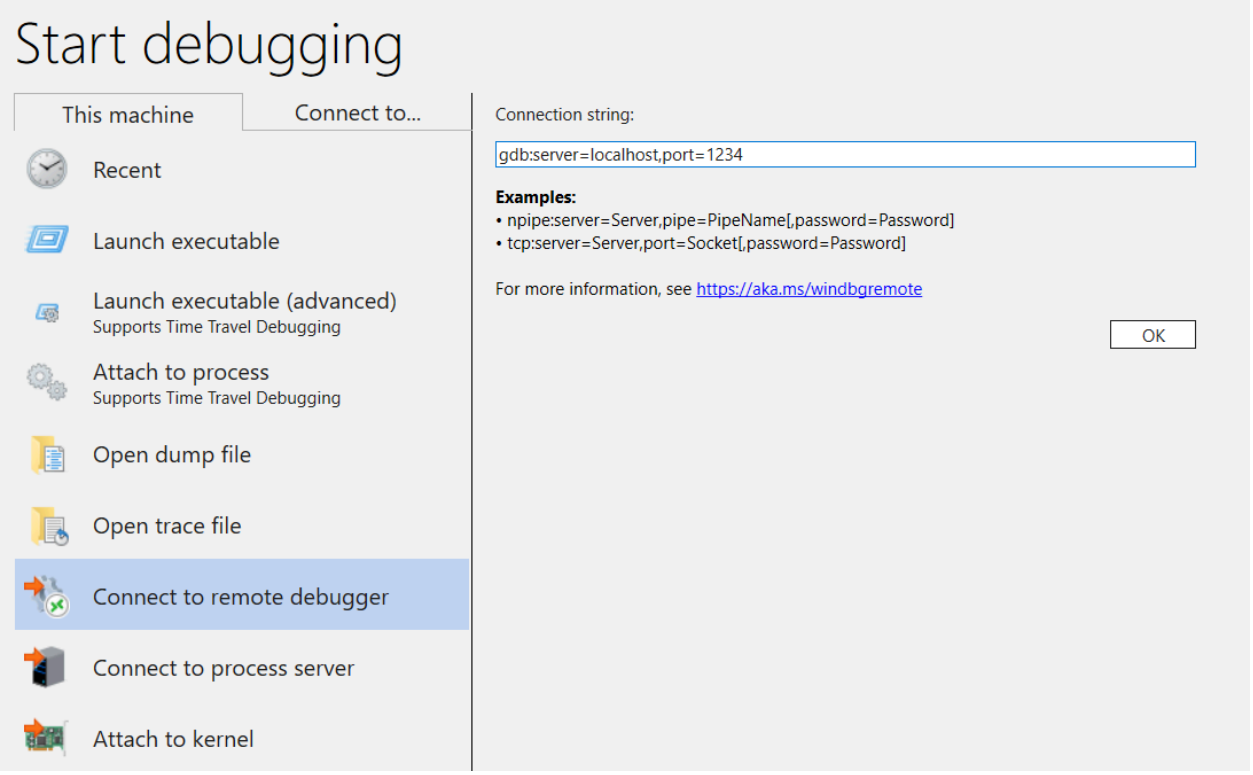
在 WSL linux 控制台上输入 `gdbserver localhost:1234 python3`，以便在 1234 端口上启动 gdbserver，并将其附加到 python3 进程。

```linux
USER1@USER1:/mnt/c/Users/USER1$ gdbserver localhost:1234 python3
Process python3 created; pid = 1211
Listening on port 1234
```

对于某些 Linux 环境，可能需要以管理员身份运行该命令，例如使用 sudo - `sudo gdbserver localhost:1234 python3`。 在启用调试器管理员根级别访问权限时要谨慎，仅在需要时方可使用。

在 WinDbg 中创建进程服务器连接

打开 WinDbg，然后选择“文件/连接到远程调试器”，然后输入用于连接的协议字符串。此处使用 `gdb:server=localhost,port=1234` 为例。



单击“确定”按钮后，调试器应连接到 gdbserver，并且应在初始进程开始中断时。

达到初始断点后，可以按几次“g”。 将收到模块加载消息（sxe 样式的“模块加载时中断”事件应能正常工作）。

请注意，由于调试符号会加载到缓存中，因此可能需要一些时间才能达到该点。 除了通过符号服务器或本地搜索路径查找符号和二进制文件外，如果无法通过 symsrv 或本地找到这些文件，GDBServer 集成还能从远程文件系统中提取这些文件。 这通常比从 symsrv 或本地搜索路径获取符号要慢得多，但通过查找适当的符号可以让整体体验变得更好。

使用 k stacks 命令列出堆栈。 它会显示 python3 模块，因此可以确认正在使用 WinDbg 在 Linux 上调试 python3。

dbgcmd

```
0:000> k
 # Child-SP          RetAddr               Call Site
00 00007fff`ffffce10 00007fff`f786d515     libc_so!_select+0xbd
01 00007fff`ffffce80 00005555`55601ce8
readline_cpython_310_x86_64_linux_gnu!PyInit_readline+0xac5
02 00007fff`ffffcf60 00005555`556f06a1     python3!PyOS_Readline+0x109
03 00007fff`ffffcfa0 00005555`556eee7e
python3!PyFrame_LocalsToFast+0x62a1
04 00007fff`ffffd000 00005555`556edcf0
python3!PyFrame_LocalsToFast+0x4a7e
05 00007fff`ffffdb80 00005555`557a18e9
python3!PyFrame_LocalsToFast+0x38f0
06 00007fff`ffffdc00 00005555`557a1470     python3!PyCodec_LookupError+0xb09
07 00007fff`ffffdc50 00005555`557b89dc     python3!PyCodec_LookupError+0x690
08 00007fff`ffffdc70 00005555`5560b42f     python3!PyUnicode_Tailmatch+0xc6c
```

```
09 00007fff`ffffdcb0 00005555`5560b012
python3!PyRun_InteractiveLoopObject+0x4e0
0a 00007fff`ffffdd50 00005555`557b7678
python3!PyRun_InteractiveLoopObject+0xc3
0b 00007fff`ffffdda0 00005555`555f55c8     python3!PyRun_AnyFileObject+0x68
0c 00007fff`ffffddd0 00005555`555ea6e8     python3!PyRun_AnyFileExFlags+0x4f
0d 00007fff`ffffde00 00005555`55780cad     python3!Py_str_to_int+0x2342a
0e 00007fff`ffffdef0 00007fff`f7c7cd90     python3!Py_BytesMain+0x2d
0f 00007fff`ffffdf20 00007fff`f7c7ce40     libc_so!_libc_init_first+0x90
10 00007fff`ffffdfc0 00005555`55780ba5     libc_so!_libc_start_main+0x80
11 00007fff`ffffe010 ffffffff`ffffffff     python3!start+0x25
12 00007fff`ffffe018 00000000`00000000     0xffffffff`ffffffff
```

此时，通过远程进程服务器连接到远程 Windows 调试器的 WinDbg 几乎可以完成所有操作。 可以单步执行、源级别调试、设置断点、检查局部变量等。

调试完成后，使用 CTRL+D 退出 WSL 中的 gbdserver 窗口。

连接到进程服务器

除了通过用户模式 GDBServer 连接到单个进程外，还可以将其设置为进程服务器，并列出和附加到系统上的现有进程。 为此，gdbserver 以“--multi”命令行参数开头 - `gdbserver --multi localhost:1234`

```
dbgcmd
```

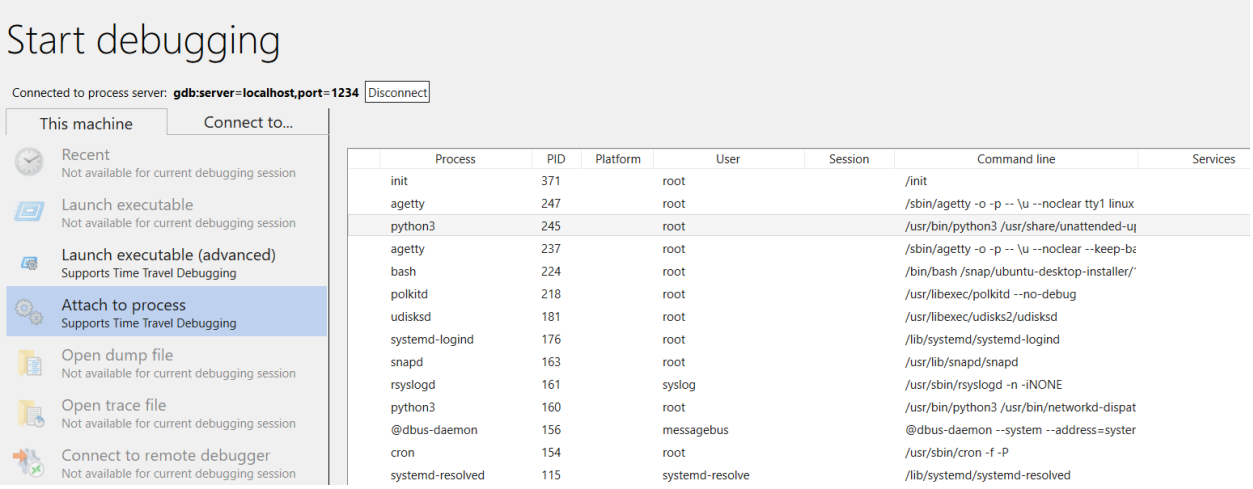
```
user1@USER1:/mnt/c/Users/USER1$ sudo gdbserver --multi localhost:1234
Listening on port 1234
```

要连接进程服务器，请在 WinDbg 中选择“文件/连接进程服务器”，然后输入与上述单进程 gdbserver 示例相同的协议字符串：

`gdb:server=localhost,port=1234`

单击“确定”按钮后，应作为进程服务器连接到 gdbserver。 与 dbgsrv 一样，可以启动一个新进程，也可以列出现有进程并附加到其中之一。

在本例中，使用“附加到进程”选项。



请注意，将看到许多对 Windows 进程可见的相同内容（包括 PID、用户和命令行）。 “附加到进程”对话框中的某些列与 Linux 无关，也不包含数据。

结束会议

使用 CTRL+D 退出 WSL 中的 gbdserver 窗口，并在 WinDbg 中选择停止调试。 要结束会话，在某些情况下可能需要退出调试器。

重新连接进程服务器

WinDbg 通过 gdbserver 是否连接到进程来识别“进程服务器”和“单个目标”。 如果连接到某个进程，让它处于冻结状态，关闭调试器，然后尝试重新连接到进程服务器，我们很可能无法将其识别为进程服务器。 在这种情况下，请重新启动目标 gdbserver 并重新连接调试器。

Linux WinDbg 功能

虽然调试器的大部分功能在调试内核转储（例如：堆栈行走、符号、类型信息、局部变量、反汇编等）时都能按预期运行，但必须注意的是，整个调试工具链尚未意识到 ELF、DWARF 以及由此产生的与 Windows 语义的差异。 调试器中的某些命令当前可能会导致意外输出。 例如，`lm` 仍然会显示 ELF 模块的错误消息，因为它需要手动分析 PE 标头。

通过 EXDI 的 Linux 内核模式

Windows 调试器支持使用 EXDI 进行内核调试。 这样就可以调试各种硬件和操作系统。 有关设置配置 EXDI 连接和故障排除的一般信息，请参阅配置 EXDI 调试器传输。

有关如何使用 EXDI 设置 QEMU 内核模式调试的信息，请参阅使用 EXDI 设置 QEMU 内核模式调试。

Linux 符号和源

本部分介绍 Linux 符号的基本用法和可用性。 有关详细信息，请参阅 Linux 符号和源以及源代码扩展访问。

DebugInfoD 符号服务器

从 WinDbg 版本 1.2104 开始，源路径命令（.srcpath、.lsrcpath （设置源路径））支持通过 `DebugInfoD*` 标记从 DebugInfoD 服务器 检索文件。

`DebugInfoD*` 标记可以指向一个或多个 DebugInfoD 服务器，其中每个服务器 URL 的格式设置为`https://domain.com`，并用 `*` 分隔。 将按源路径中列出的顺序搜索服务器，并从第一个匹配的 URL 检索文件。 有关详细信息，请参阅源代码扩展访问。

例如，可以使用 .sympath（设置符号路径） 命令来设置如下所示的 DebugInfoD 路径。

```
.sympath+ DebugInfoD*https://debuginfod.elfutils.org
```

有关设置符号路径的一般信息，请参阅使用符号。

要显示正在加载的符号信息，请使用 `!sym noisy`。 有关详细信息，请参阅 !sym。

此外，还支持从 DebugInfoD 服务器自动下载源，这些源支持返回该项目类型。 实际上，可以执行以下操作：

```
.srcpath+ DebugInfoD*https://debuginfod.elfutils.org
```

有关使用 DWARF 符号和 Linux 符号实用工具（如 `!sourcemap` 和 `!diesym`）的详细信息，请参阅 Linux 符号和源。

C++ 应用演练

1. 使用文本编辑器（如 nano 或 vi）创建 C++ 文件。 例如：

```
nano DisplayGreeting.cpp
```

2. 在文本编辑器中编写 C++ 程序。 这是一个显示问候语的简单程序，需要对其进行调试：

```
C++

#include <array>
#include <cwchar>
#include <cstdio>
#include <iostream>
```

```
using namespace std;

void GetCppConGreeting(wchar_t* buffer, size_t size)
{
    wchar_t const* const message = L"HELLO FROM THE WINDBG TEAM. GOOD LUCK
IN ALL OF YOUR TIME TRAVEL DEBUGGING!";
    wcsncpy(buffer, message, size);
}

int main()
{
    std::array<wchar_t, 50> greeting{};
    GetCppConGreeting(greeting.data(), greeting.size());

    cin.get();
    wprintf(L"%ls\n", greeting.data());

    return 0;
}
```

3. 保存 (CTRL-O) 并退出 (CTRL-X) nano 编辑器。

4. 使用 g++ 编译 C++ 文件。 -o 选项用于指定输出文件名，而 -g 选项用于生成符号文件：

```
g++ DisplayGreeting.cpp -g -o DisplayGreeting
```

5. 如果代码中没有错误，g++ 命令将在目录中创建一个名为 DisplayGreeting 的可执行文件。

6. 可以使用以下命令来运行程序：

```
./DisplayGreeting
```

7. 按返回键后，应用中将显示该消息。 从输出结果来看，“问候语”似乎被截断了，取而代之的是“????”。

```
HELLO FROM THE WINDBG TEAM. GOOD LUCK IN ALL OF YO????
```

调试 DisplayGreeting

1. 代码准备好运行后，就可以使用 gdbserver 启动应用。

```
gdbserver localhost:1234 DisplayGreeting
```

2. 打开 WinDbg，然后选择“文件/连接到远程调试器”，然后输入用于连接的协议字符串。 此处使用 `gdb:server=localhost,port=1234` 为例。

3. 连接完成后，输出应显示它正在侦听 1234 端口，并已建立远程调试连接。

dbgcmd

```
Bob@Bob6:/mnt/c/Users/bob$ gdbserver localhost:1234 DisplayGreeting
Process /mnt/c/Users/bob/DisplayGreeting created; pid = 725
Listening on port 1234
Remote debugging from host 127.0.0.1, port 47700
```

如前所述，对于某些 Linux 环境，可能需要以管理员身份运行该命令，通常使用 sudo。 在启用调试器管理员根级别访问权限时要谨慎，仅在需要时方可使用。

在调试器会话中添加源和符号路径

要设置断点并查看源代码和变量，请设置符号和源路径。 有关设置符号路径的一般信息，请参阅使用符号。

使用 `.sympath` 将符号路径添加到调试器会话。 在本例中，代码运行在 WSL Linux Ubuntu 的这个位置，用户名为 Bob。

`\\wsl$\Ubuntu\mnt\c\Users\Bob\`

在 WSL 中，此目录映射到 Windows OS 的位置：`C:\Users\Bob\`

因此会使用这两个命令。

`.sympath C:\Users\Bob\`

`.srcpath C:\Users\Bob\`

有关 WSL 文件系统的详细信息，请参阅 WSL 的文件权限。

1. 要使用额外的 Linux OS 符号，请使用 .sympath 位置来添加 DebugInfoD 符号，如下所示。

`.sympath+ DebugInfoD*https://debuginfod.elfutils.org`

2. 此外，还支持从 DebugInfoD 服务器自动下载源，这些源支持返回该项目类型。 要利用这一点，请使用 .srcpath 添加 elfutils 服务器。

`.srcpath+ DebugInfoD*https://debuginfod.elfutils.org`

设置断点

在 DisplayGreeting 应用的主位置中设置一个断点。

dbgcmd

```
0:000> bp DisplayGreeting!main
0:000> bl
     0 e Disable Clear  00005555`55555225
[/mnt/c/Users/bob/DisplayGreeting.cpp @ 14]     0001 (0001)  0:****
DisplayGreeting!main
```

使用 Go 命令或菜单选项重新启动代码执行。

加载源代码

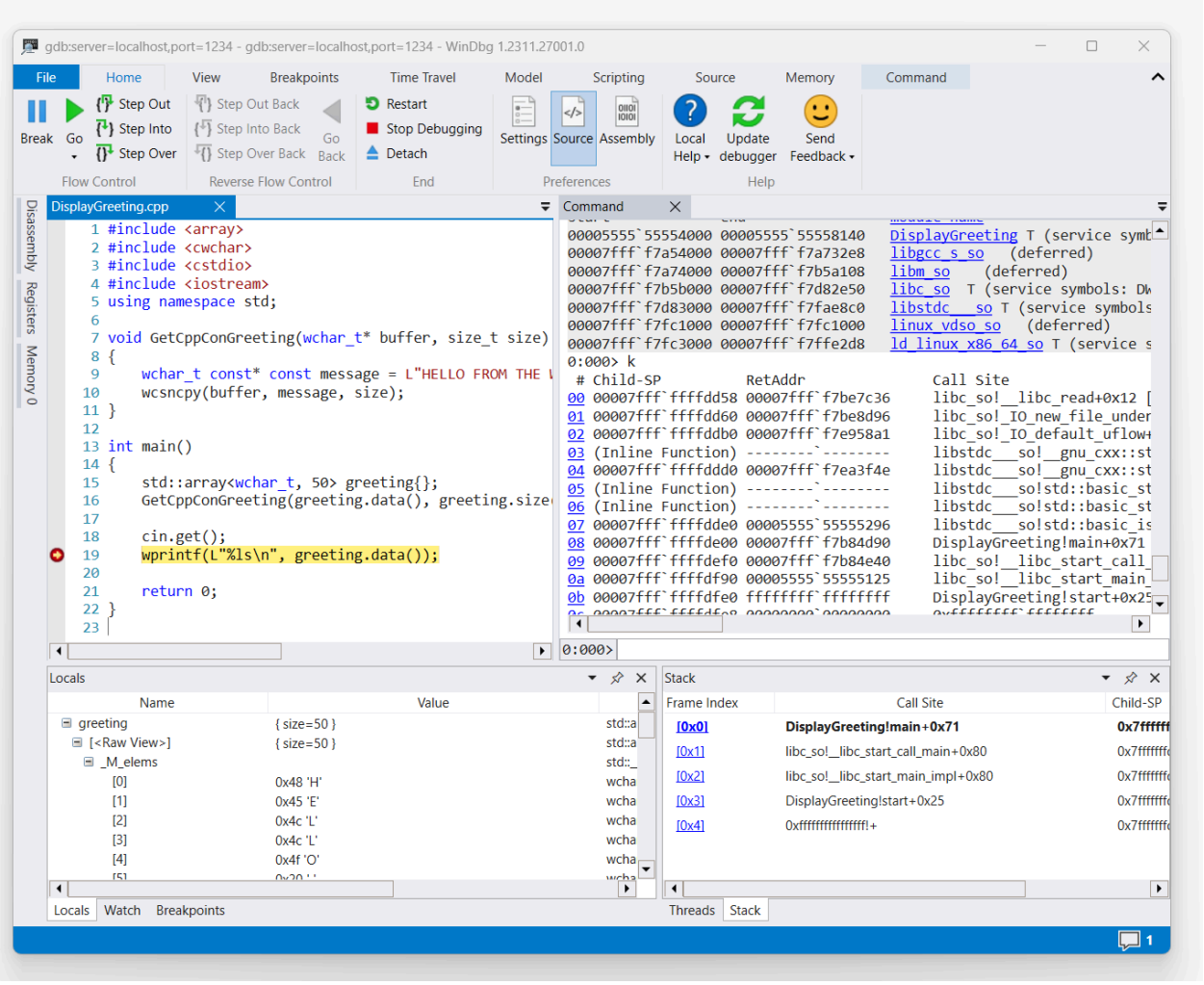
使用 .reload 命令重新加载符号。

使用 `lm` 命令确认正在运行 DisplayGreeting 应用。

dbgcmd

```
0:000> lm
start             end                 module name
00005555`55554000 00005555`55558140   DisplayGreeting T (service symbols:
DWARF Private Symbols)        c:\users\bob\DisplayGreeting
00007fff`f7a54000 00007fff`f7a732e8   libgcc_s_so   (deferred)
00007fff`f7a74000 00007fff`f7b5a108   libm_so    (deferred)
00007fff`f7b5b000 00007fff`f7d82e50   libc_so  T (service symbols: DWARF
Private Symbols)        C:\ProgramData\Dbg\sym\_.debug\elf-buildid-sym-
a43bfc8428df6623cd498c9c0caeb91aec9be4f9\_.debug
00007fff`f7d83000 00007fff`f7fae8c0   libstdc___so   (deferred)
00007fff`f7fc1000 00007fff`f7fc1000   linux_vdso_so   (deferred)
00007fff`f7fc3000 00007fff`f7ffe2d8   ld_linux_x86_64_so T (service symbols:
DWARF Private Symbols)        C:\ProgramData\Dbg\sym\_.debug\elf-buildid-
sym-9718d3757f00d2366056830aae09698dbd35e32c\_.debug
```

一旦命令触发了对显示问候语代码的访问，它就会显示在 WinDbg 中。



使用“k”命令列出堆栈。

dbgcmd

```
0:000> k
 # Child-SP          RetAddr               Call Site
00 00007fff`ffffde00 00007fff`f7b84d90     DisplayGreeting!main+0x1f
[/mnt/c/Users/BOB/DisplayGreeting.cpp @ 15]
01 00007fff`ffffdef0 00007fff`f7b84e40
libc_so!__libc_start_call_main+0x80 [./csu/../sysdeps/x86/libc-start.c @ 58]
02 00007fff`ffffdf90 00005555`55555125
libc_so!__libc_start_main_impl+0x80
[./csu/../sysdeps/nptl/libc_start_call_main.h @ 379]
03 00007fff`ffffdfe0 ffffffff`ffffffff     DisplayGreeting!start+0x25
04 00007fff`ffffdfe8 00000000`00000000     0xffffffff`ffffffff```
```

使用 dx 命令查看局部变量 *greeting*。 请注意，它的大小为 50。

dbgcmd

```
0:000> dx greeting
greeting                 : { size=50 } [Type: std::array<wchar_t, 50>]
    [<Raw View>]     [Type: std::array<wchar_t, 50>]
```

仔细浏览代码，注意 50 的大小可能无法满足问候语消息的需要。

```
wchar_t const* const message = L"HELLO FROM THE WINDBG TEAM. GOOD LUCK IN ALL OF YOUR TIME TRAVEL
```

通过扩展问候语的局部变量并看到问候语被截断来确认这一点。

gdbserver 连接故障排除

使用 `--debug` 选项可在 gdbserver 控制台上显示更多信息，以收集更多有关连接状态的信息。 例如，使用此命令启动进程服务器。

```
gdbserver --debug --multi localhost:1234
```

另请参阅

Linux 符号和源

源代码扩展访问

ELFUTILS debuginfod

选择最佳的远程调试方法

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Linux 故障转储

项目 • 2024/11/18

本文介绍如何创建和查看不同类型的 Linux 故障转储文件。 查看 Linux 故障转储需要 WinDbg 版本 1.2402.24001.0 或更高版本。

在 WinDbg 中打开 Linux（非 Windows）核心转储时，基本调试器命令应全部正常工作，但引用 Windows 结构的扩展和特定于 Windows 的命令将不起作用。

支持的故障转储文件

Linux 内核转储

在 Windows 调试器中提供了打开 Linux 内核压缩的 KDUMP，并使用完整的专用 DWARF 符号执行事后调试和分析。

WinDbg 仅支持 ZLIB 压缩的 KDUMP 文件。 不支持 LZO 和 Snappy 压缩 KDUMP。

有关 Linux KDUMP 的一般信息，请参阅 KDump （Linux） 维基百科页和 核心转储。

ELF 核心转储

作为支持 Open Enclave 的一部分，WinDbg 可以从 Enclave 和 Linux 应用程序打开 ELF 核心转储和二进制文件以及 DWARF 符号（DWARF 5 不受支持）。 有关 Open Enclave 的详细信息，请参阅 Open Enclave 调试。

进程转储

支持单个进程转储。 有多种方法可以收集进程转储，包括 Linux Sysinternals ProcDump for Linux 实用工具。 另一个选项是使用 GNU 调试器 - GDBServer 生成核心转储。 有关 GDBServer 的详细信息，请参阅 https://en.wikipedia.org/wiki/Gdbserver。 Sourceware 网站提供了远程 gdb 调试文档 - 调试远程程序。

启用 Linux NATVIS 可视化和 DML 链接遍历

Windows 调试器中的标准模板库 （STL） 扩展性由 NatVis 文件提供， `stl.natvis` 该文件可了解随 Visual Studio 和 Windows 一起随附的许多版本的 STL。 有关 NATVIS 的常规

信息，请参阅 NatVis 中的本机调试器对象。 用于 Linux 组件（GCC 或 LLDB）的 STL 版本非常不同。

若要为 Linux 启用 NATVIS 可视化和 DML 链接遍历优化，请先卸载默认 natvis 文件 .scriptunload `stl.natvis`。

然后 .scriptload 文件 `gstl.natvis` 。 使用 .scriptlist 确认是否 `gstl.natvis` 处于活动状态。

dbgcmd

```
0: kd> .scriptlist
Command Loaded Scripts:
...
    NatVis script from
'C:\Users\Bob\AppData\Local\dbg\UI\2402.24001.0\amd64\Visualizers\gstl.natvi
s'
```

有关使用 DML 的详细信息，请参阅 使用 DML 自定义调试器输出。

DisplayGreeting 应用的单个进程核心转储

此示例演示如何使用 gdb 创建单个进程核心转储。 有关将 GDBServer 与 WinDbg 配合使用以及代码演练的详细信息，请参阅 Linux 实时远程进程调试。 有关 DisplayGreeting 的示例代码，请参阅 C++应用演练。

找到所需的进程

我们可以使用 `ps -A` 命令列出 Linux 中的所有进程，或者将 -f 选项与 pgrep 一起使用，因为我们知道我们要查找 DisplayGreeting 应用。

dbgcmd

```
$ pgrep -f DisplayGreeting
9382
```

在本示例演练中，它显示了 9382 的进程 ID。

使用 gdb 附加到进程并生成核心转储

使用 gdb 附加到进程。

dbgcmd

```
$ gdb -p 9382
```

显示 gdb 命令的 `generate-core-file` 帮助。

```dbgcmd
(gdb) help generate-core-file
Save a core file with the current state of the debugged process.
Argument is optional filename.  Default filename is 'core.<process_id>'.
```

然后在 （gdb） 提示符下，生成包含默认文件名的进程核心转储文件。

```dbgcmd
(gdb) generate-core-file
Saved corefile core.9382
(gdb) quit
```

加载并检查 Linux 进程核心转储

使用 WinDbg 中的“打开转储文件”菜单选项加载生成的核心转储。

在调试器会话中添加源和符号路径

若要查看源代码和变量，请设置符号和源路径。 有关设置符号路径的一般信息，请参阅使用符号。 有关 Linux 符号的详细信息，请参阅 Linux 符号和源。

使用 `.sympath` 将符号路径添加到调试器会话。 在此 WSL Linux Ubuntu 示例中，DisplayGreetings 代码和符号可用于名为 Bob 的用户。

`\\wsl$\Ubuntu\mnt\c\Users\Bob\`

在 WSL 中，此目录映射到 Windows OS 位置： `C:\Users\Bob\` 因此使用这两个命令。

`.sympath C:\Users\Bob\`

`.srcpath C:\Users\Bob\`

有关在 Windows 中访问 WSL 文件系统的详细信息，请参阅 WSL 的文件权限。

若要从其他 Linux OS 符号中受益，请使用 .sympath 位置添加 DebugInfoD 符号。

`.sympath+ DebugInfoD*https://debuginfod.elfutils.org`

`.reload`使用命令重新加载符号。

此外，还支持从 DebugInfoD 服务器自动下载源，这些源支持返回该项目类型。 若要利用此功能，请使用 .srcpath 添加 elfutils 服务器。

```
.srcpath+ DebugInfoD*https://debuginfod.elfutils.org
```

检查进程转储

使用 `lm` 命令确认转储文件是否包含 DisplayGreeting 应用。

dbgcmd

```
0:000> lm
start             end                 module name
00005555`55554000 00005555`55558140   DisplayGreeting T (service symbols:
DWARF Private Symbols)        c:\users\bob\DisplayGreeting
00007fff`f7a54000 00007fff`f7a732e8   libgcc_s_so   (deferred)
00007fff`f7a74000 00007fff`f7b5a108   libm_so    (deferred)
00007fff`f7b5b000 00007fff`f7d82e50   libc_so  T (service symbols: DWARF
Private Symbols)        C:\ProgramData\Dbg\sym\_.debug\elf-buildid-sym-
a43bfc8428df6623cd498c9c0caeb91aec9be4f9\_.debug
00007fff`f7d83000 00007fff`f7fae8c0   libstdc___so   (deferred)
00007fff`f7fc1000 00007fff`f7fc1000   linux_vdso_so   (deferred)
00007fff`f7fc3000 00007fff`f7ffe2d8   ld_linux_x86_64_so T (service symbols:
DWARF Private Symbols)        C:\ProgramData\Dbg\sym\_.debug\elf-buildid-
sym-9718d3757f00d2366056830aae09698dbd35e32c\_.debug
```

请注意，第一个命令执行可能需要一些时间，因为调试符号将加载到缓存中。 除了通过符号服务器或本地搜索路径查找符号和二进制文件之外，由于 GDBServer 集成，如果无法在本地找到这些文件，它可能会从远程文件系统加载这些文件。 此操作通常比从 symsrv 或本地搜索路径获取符号慢。

`x`使用命令显示 DisplayGreeting 中可用的函数。

dbgcmd

```
0:000> x /D /f DisplayGreeting!*
 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

*** WARNING: Unable to verify timestamp for DisplayGreeting
00005651`7935b331 DisplayGreeting!_GLOBAL__sub_I__Z17GetCppConGreetingPwm
(void)
00005651`7935b2db DisplayGreeting!__static_initialization_and_destruction_0
(int, int)
00005651`7935b37b DisplayGreeting!std::__array_traits<wchar_t, 50>::_S_ptr
(wchar_t (*)[50])
00005651`7935b368 DisplayGreeting!std::array<wchar_t, 50>::size
(std::array<wchar_t, 50> *)
```

```
00005651`7935b34a DisplayGreeting!std::array<wchar_t, 50>::data
(std::array<wchar_t, 50> *)
00005651`7935b225 DisplayGreeting!main (void)
00005651`7935b1e9 DisplayGreeting!GetCppConGreeting (wchar_t *, size_t)
```

`dx`使用命令查看局部变量*问候语*。

dbgcmd

```
0:000> dx greeting
...
Error: Unable to bind name 'greeting'
```

由于在进行转储时尚未使用参数问候语，因此在转储文件中不可用。

`dx`使用命令检查转储文件中可用的进程。

dbgcmd

```
:000> dx @$cursession.Processes.Take(30)
@$cursession.Processes.Take(30)
    [0x24a6]         : DisplayGreeting [Switch To]

Click on the `[Switch To]` DML link to switch to the 9382 process.

```dbgcmd
0:000> dx -s @$cursession.Processes.Take(30)[9382].SwitchTo()
0:000> dx -r1 @$cursession.Processes.Take(30)[9382]
@$cursession.Processes.Take(30)[9382]                 : DisplayGreeting
[Switch To]
    Name             : DisplayGreeting
    Id               : 0x24a6
    Index            : 0x0
    Handle           : 0x24a6
    Threads
    Modules
    Environment
    Direct3D
    Attributes
    Devices
    Io
    Memory
    TTD
    GroupedStacks
```

若要查看有关线程和模块的信息，请单击输出中生成的 DML 链接，或键入类似于此命令的命令进行故障转储。

dbgcmd
```

```
0:000> dx -r1 @$cursession.Processes.Take(30)[9382].Threads
@$cursession.Processes.Take(30)[9382].Threads
    [0x24a6]         [Switch To]
0:000> dx -r1 @$cursession.Processes.Take(30)[9382].Modules
@$cursession.Processes.Take(30)[9382].Modules
    [0x0]            : /mnt/c/Users/Bob/DisplayGreeting
    [0x1]            : /usr/lib/x86_64-linux-gnu/libgcc_s.so.1
    [0x2]            : /usr/lib/x86_64-linux-gnu/libm.so.6
    [0x3]            : /usr/lib/x86_64-linux-gnu/libc.so.6
    [0x4]            : /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.30
    [0x5]            : /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
    [0x6]            : linux-vdso.so.1
```

使用 ELF/CORE 诊断扩展显示转储文件信息

使用 Linux 诊断扩展 - ELFBinComposition.dll显示转储文件信息。 例如，用于 `!dumpdebug` 确认这是 ELF 用户核心转储，并显示其他信息。

dbgcmd

```
0:000> !dumpdebug
Dump Diagnostics: Format = ELF User Core
**************************************************************************
****
File Mapping Size:           0x151d78 (1 Mb)
Highest Memory Offset:       0x14e5f0 (1 Mb)
...
```

用于 `!ntprpsinfo` 显示NT_PRPSINFO数据。

dbgcmd

```
0:000> !ntprpsinfo
NT_PRPSINFO (process info):
    state: 0, sname: t, zomb: 0, nice: 0, flag: 0x4040000019999999
    uid: 1000, gid: 1000, pid: 9382, ppid: 388, pgrp: 9382, sid: 388
    fname: DisplayGreeting
    psargs: ./DisplayGreeting
```

内核 KDump 故障转储

在 Linux 中创建故障转储文件有多种方法。 例如，内核故障转储中 介绍了 Ubuntu Linux 的一个选项。

其他选项包括使用 kexectools 来启用 Kdump。 有关详细信息，请参阅 KDump（Linux）。 如果已启用 kdump，则可以验证 kdump 是否处于活动状态并使用它运行 `systemctl status kdump`。

在测试系统上触发 OS 崩溃后，将创建故障转储文件。

加载并检查 Linux OS 故障转储

使用“打开转储文件”菜单选项加载生成的 kdump。

如上一部分所述，通过加载 `gstl.natvis` 文件启用 NATVIS 可视化和 DML 链接遍历优化 Linux。

使用 ELF Bin 组合命令分析 Linux 内核转储

若要能够使用其他 ELF Bin 组合命令，请使用 .chain 命令确认加载了 ELFBinComposition.dll。

dbgcmd

```
0: kd> .chain
Extension DLL chain:
    ELFBinComposition: image 10.0.27606.1000, API 0.0.0,
        [path:
C:\Users\Bob\AppData\Local\dbg\UI\Fast.20240423.1\amd64\winext\ELFBinComposi
tion.dll]
...
```

如果未加载 ELFBinComposition，请使用 .load 加载它。 有关详细信息，请参阅 .load、.loadby （加载扩展 DLL）

`!ELFBinComposition.dumpdebug`使用命令显示有关已加载转储文件的信息。 在此示例中，已加载 ELF 用户核心转储文件。

dbgcmd

```
0: kd> !ELFBinComposition.dumpdebug
Dump Diagnostics: Format = Kernel KDump
**************************************************************************
****
File Mapping Size:           0x3b34090 (59 Mb)
Highest Memory Offset:       0x3b34090 (59 Mb)
```

使用 ELFBinComposition `!vmcoreinfo` 显示正在调试的 Linux 内核核心转储（KDUMP）中的 VMCOREINFO 表。

dbgcmd

```
0: kd> !vmcoreinfo
VMCOREINFO:
    OSRELEASE=6.5.0-25-generic
    BUILD-ID=8567ad7c7c2f78f3654f6cc90a9e1b3f9c3a4b32
    PAGESIZE=4096
    SYMBOL(init_uts_ns)=ffffded86e11b388
    OFFSET(uts_namespace.name)=0
    SYMBOL(node_online_map)=ffffded86dcceb40
    SYMBOL(swapper_pg_dir)=ffffded86d143000
    SYMBOL(_stext)=ffffded86ace0000
    SYMBOL(vmap_area_list)=ffffded86de48140
    SYMBOL(mem_section)=ffff0f2e1efe4600
    LENGTH(mem_section)=8192
...
```

使用从 `!kdumppagerange` 零开始转储文件的第一部分转储。

dbgcmd

```
0: kd> !kdumppagerange 0
    PFNs [0x540e0, 0x55643) -> Descs [0x0, 0x1563): File Offsets [0x307430,
0xeeb37a) 0xbe3f4a bytes across 5475 pages as ZLIB
    PFNs [0x55643, 0x55650) -> Descs [0x1563, 0x1570): File Offsets
[0x306430, 0x307430) 0x1000 bytes across 13 duplicate pages as Uncompressed
    PFNs [0x55650, 0x556d6) -> Descs [0x1570, 0x15f6): File Offsets
[0xeeb37a, 0xf0c405) 0x2108b bytes across 134 pages as ZLIB
    PFNs [0x556d6, 0x556dc) -> Descs [0x15f6, 0x15fc): File Offsets
[0xf0c405, 0xf12405) 0x6000 bytes across 6 pages as Uncompressed
    PFNs [0x556dc, 0x55e98) -> Descs [0x15fc, 0x1db8): File Offsets
[0xf12405, 0x1216d1b) 0x304916 bytes across 1980 pages as ZLIB
    PFNs [0x55e98, 0x55ea4) -> Descs [0x1db8, 0x1dc4): File Offsets
[0x1216d1b, 0x1222d1b) 0xc000 bytes across 12 pages as Uncompressed
    PFNs [0x55ea4, 0x56542) -> Descs [0x1dc4, 0x2462): File Offsets
[0x1222d1b, 0x14ba138) 0x29741d bytes across 1694 pages as ZLIB
    PFNs [0x56542, 0x56543) -> Descs [0x2462, 0x2463): File Offsets
[0x306430, 0x307430) 0x1000 bytes across 1 pages as Uncompressed
    PFNs [0x56543, 0x56544) -> Descs [0x2463, 0x2464): File Offsets
[0x14ba138, 0x14ba194) 0x5c bytes across 1 pages as ZLIB
    PFNs [0x56544, 0x5654f) -> Descs [0x2464, 0x246f): File Offsets
[0x306430, 0x307430) 0x1000 bytes across 11 duplicate pages as Uncompressed
```

！ kdumppagerange 的输出显示各种页面帧 （PFN） 值。 我们可以选择感兴趣的一个，并使用该 `!kdumppfn <PFN>` 信息显示有关 PFN 及其数据在 KDUMP 中的位置的信息。

dbgcmd

```
0: kd> !kdumppfn 0x540e0
    Page frame 0x540e0 = File offset [0x307430, 0x307b9f) 0x76f bytes as
```

```
ZLIB...
```

检查转储文件

`k`使用命令显示调用堆栈，调查发生崩溃时正在运行的代码。

dbgcmd

```
6: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff0000`0bc3bc90 ffff0000`085161f8     vmlinux!sysrq_handle_crash+0x24
[/usr/src/kernel/drivers/tty\sysrq.c @ 147]
01 ffff0000`0bc3bca0 ffff0000`08516824     vmlinux!__handle_sysrq+0x88
[/usr/src/kernel/drivers/tty\sysrq.c @ 583]
02 ffff0000`0bc3bcb0 ffff0000`08308990     vmlinux!write_sysrq_trigger+0xb4
[/usr/src/kernel/drivers/tty\sysrq.c @ 1110]
03 ffff0000`0bc3bcf0 ffff0000`08290070     vmlinux!proc_reg_write+0x80
[/usr/src/kernel/fs/proc\inode.c @ 245]
04 ffff0000`0bc3bd10 ffff0000`0829039c     vmlinux!__vfs_write+0x60
[/usr/src/kernel/fs\read_write.c @ 490]
05 ffff0000`0bc3bd50 ffff0000`08290704     vmlinux!vfs_write+0xac
[/usr/src/kernel/fs\read_write.c @ 550]
06 ffff0000`0bc3be00 ffff0000`082907a4     vmlinux!ksys_write+0x74
[/usr/src/kernel/fs\read_write.c @ 599]
07 (Inline Function) --------`--------     vmlinux!__do_sys_write+0xc
[/usr/src/kernel/fs\read_write.c @ 608]
08 (Inline Function) --------`--------     vmlinux!__se_sys_write+0xc
[/usr/src/kernel/fs\read_write.c @ 608]
09 ffff0000`0bc3be40 ffff0000`08095904     vmlinux!__arm64_sys_write+0x24
[/usr/src/kernel/fs\read_write.c @ 608]
0a ffff0000`0bc3be90 ffff0000`080834c8     vmlinux!el0_svc_handler+0x94
0b ffff0000`0bc3beb0 00000000`00000000     vmlinux!el0_svc+0x8
```

`dx`使用命令检查转储文件。 例如，使用此命令查看前 30 个进程。

dbgcmd

```
6: kd> dx @$cursession.Processes.Take(30)
@$cursession.Processes.Take(30)
    [0x0]            : swapper/0 [Switch To]
    [0x1]            : systemd [Switch To]
    [0x2]            : kthreadd [Switch To]
    [0x3]            : rcu_gp [Switch To]
    [0x4]            : rcu_par_gp [Switch To]
    [0x5]            : kworker/0:0 [Switch To]
    [0x6]            : kworker/0:0H [Switch To]
    [0x7]            : kworker/u16:0 [Switch To]
    [0x8]            : mm_percpu_wq [Switch To]
    [0x9]            : ksoftirqd/0 [Switch To]
    [0xa]            : rcu_sched [Switch To]
```

```
    [0xb]            : rcu_bh [Switch To]
    [0xc]            : migration/0 [Switch To]
...
```

单击 DML 链接或使用类似于此链接的命令查看感兴趣的进程的线程。

dbgcmd

```
6: kd> dx @$cursession.Processes[0x1a].Threads
@$cursession.Processes[0x1a].Threads
    [0x1a]           [Switch To]
6: kd> dx @$cursession.Processes[0x1a].Threads[0x1a]
@$cursession.Processes[0x1a].Threads[0x1a]                 [Switch To]
    KernelObject     [Type: thread_struct]
    Id               : 0x1a
    Stack
    Registers
    Environment
    Analysis
    WaitChain
    Scheduling
    IRPs
...
```

提供了每个线程的其他信息，如下所示。

dbgcmd

```
6: kd> dx @$cursession.Processes[0x1a].Threads[0x1a].KernelObject
@$cursession.Processes[0x1a].Threads[0x1a].KernelObject
[Type: thread_struct]
    [+0x000] cpu_context      [Type: cpu_context]
    [+0x070] uw               [Type: <unnamed-tag>]
    [+0x290] fpsimd_cpu       : 0x100 [Type: unsigned int]
    [+0x298] sve_state        : 0x0 [Type: void *]
    [+0x2a0] sve_vl           : 0x0 [Type: unsigned int]
    [+0x2a4] sve_vl_onexec    : 0x0 [Type: unsigned int]
    [+0x2a8] fault_address    : 0x0 [Type: long unsigned int]
    [+0x2b0] fault_code       : 0x0 [Type: long unsigned int]
    [+0x2b8] debug            [Type: debug_info]
6: kd> dx -s @$cursession.Processes[0x1a].Threads[0x1a].SwitchTo()
Process ffff8008`0f894380 has invalid page directories
```

使用 LinuxKernel.js 脚本分析 Linux 内核转储

LinuxKernel.js调试器扩展包含一组命令，旨在与 Linux 崩溃实用工具中用于打开和分析 Linux 内核模式崩溃的命令类似。

若要使用该脚本，请先加载该脚本。

dbgcmd

```
0: kd> .scriptload LinuxKernel.js
JavaScript script successfully loaded from
'C:\Users\Bob\AppData\Local\dbg\UI\Fast.20240423.1\amd64\winext\LinuxKernel.
js'
```

有关使用加载脚本的详细信息，请参阅 JavaScript 调试器脚本。

！文件

使用转 `!files` 储文件中显示有关 Linux 文件结构的信息。 它类似于崩溃文件命令。

dbgcmd

```
6: kd> !files
@$files()                  : Files for process 'sh' (pid 545) root dir = '/'
working dir = '/home/root'
    [0x0]             : /dev/ttyS0 [Type: file]
    [0x1]             : /proc/sysrq-trigger [Type: file]
    [0x2]             : /dev/ttyS0 [Type: file]
    [0xa]             : /dev/ttyS0 [Type: file]
    [0xff]            : /dev/ttyS0 [Type: file]
```

`!files` 语法：

`!files [<arg>]`

没有 `[<arg>]` - 等效于“文件”- 提供当前进程文件列表

`[<arg>]`：

`pid` - 为给定进程 ID 提供文件列表

`64-bit num` - 为给定地址的任务提供文件列表

`<task struct [*]>` - 按对象为给定任务结构提供文件列表

`<process object>` - 为进程对象表示的任务提供文件列表

！安装

使用转 `!mount` 储文件中显示有关 Linux 文件结构的信息。

dbgcmd

```
6: kd> !mount
@$mount()
    [0x0]            : (rootfs) rootfs at / [Type: mount]
    [0x1]            : (squashfs) /dev/mapper/nested_rootfs at / [Type:
mount]
    [0x2]            : (sysfs) sysfs at /sys [Type: mount]
    [0x3]            : (proc) proc at /proc [Type: mount]
    [0x4]            : (devtmpfs) devtmpfs at /dev [Type: mount]
    [0x5]            : (securityfs) securityfs at /kernel/security [Type:
mount]
```

`!mount` 语法：

没有 `[<arg>]` - 等效于“mount”命令 - 显示装载的文件系统

`[<arg>]`：

`pid` - 为具有给定 pid 的进程命名空间提供装载的文件系统

`64-bit num` - 为地址提供的task_struct命名空间提供装载的文件系统

`<task struct [*]>` - 为给定task_struct的命名空间提供装载的文件系统

`<process object>` - 为进程表示的任务命名空间提供装载的文件系统

！网

使用 “ `!net` 显示系统网络”列表。

dbgcmd

```
6: kd> !net
@$net()
    [0x0]            : lo (127.0.0.1) [Type: net_device]
    [0x1]            : enP8p1s0f0np0 (192.168.3.19) [Type: net_device]
    [0x2]            : enP8p1s0f1np0 [Type: net_device]
```

`!net` 语法：

`!net [<arg>]`

没有 `[<arg>]` - 等效于“net”- 提供系统网络列表

`[<arg>]`：

`pid` - 为具有给定 pid 的进程命名空间提供净列表

`64-bit num` - 为地址提供的task_struct命名空间提供净列表

`<task struct [*]>` - 为给定task_struct的命名空间提供净列表

`<process object>` - 为进程表示的任务命名空间提供网络列表

！runq

`!runq`使用该命令显示有关运行队列中任务的信息。

dbgcmd

```
0: kd> !runq
@$runq()
    [0x0]            : CPU 0 run queue [current = 'bash' (17ca)]
        Cpu              : 0x0
        RunQueue         [Type: rq]
        CurrentTask      : bash [Type: task_struct]
        RTTasks
        CfsTasks
            [0x16b3]         : kworker/0:7 [Type: task_struct]
    [0x1]            : CPU 1 run queue [current = 'swapper/1' (0)]
        Cpu              : 0x1
        RunQueue         [Type: rq]
        CurrentTask      : swapper/1 [Type: task_struct]
        RTTasks
        CfsTasks
```

`!runq` 语法：

`!runq`

！runq 没有任何命令参数。

其他内核转储命令

`!dev` - 显示有关字符和块设备、分配、I/O 端口使用情况、I/O 内存使用情况的设备数据。

`!log` - 显示内核log_buf内容。

`!vm` - 显示虚拟内存使用情况的摘要。

`!timer` - 显示计时器队列条目。

dx 命令和 Linux 对象

dx 命令可用于调查 kdumps。 显示 Sessions 对象以查看可用的各种子对象。

dbgcmd

```
0: kd> dx -r3 Debugger.Sessions[0]
Debugger.Sessions[0]                 : Target Composition Target
    Processes
        [0x0]            : swapper/0 [Switch To]
            KernelObject     : swapper/0 [Type: task_struct]
            Name             : swapper/0
            Id               : 0x0
            Index            : 0x0
            Threads
            Modules
            Environment
            Direct3D
            Attributes
            Devices
            Io
            Memory
            GroupedStacks
...
```

cursession Kernel 对象包含可用于查看内核日志的 PrintKLog 对象。

dbgcmd

```
6: kd> dx @$cursession.Kernel.PrintKLog.Take(4)
@$cursession.Kernel.PrintKLog.Take(4)
    [0x0]            : [     0.000000] Booting Linux on physical CPU
0x0000000000 [0x410fd083]
    [0x1]            : [     0.000000] Linux version 4.19.90-microsoft-
standard (oe-user@oe-host) (gcc version 8.2.0 (GCC)) #1 SMP Fri Mar 27
14:25:24 UTC 2020..
    [0x2]            : [     0.000002] sched_clock: 64 bits at 125MHz,
resolution 8ns, wraps every 4398046511100ns
    [0x3]            : [     0.000003]         17.250901928 MSFT: kernel
boot start
```

此 dx 命令显示用于 `.Contains()` 在日志中查找特定字符串。

dbgcmd

```
6: kd> dx @$cursession.Kernel.PrintKLog.Where(le =>
le.ToLower().Contains("oops") || le.ToLower().Contains("crash"))
@$cursession.Kernel.PrintKLog.Where(le => le.ToLower().Contains("oops") ||
le.ToLower().Contains("crash"))
    [0x0]            : [     0.000493] crashkernel reserved:
0x00000000dc600000 - 0x00000000fc600000 (512 MB)
    [0x1]            : [     0.078790] Kernel command line:
console=ttyS0,115200n8 earlycon=uart8250,mmio32,0x68A10000 crashkernel=512M
```

```
enforcing=0 ipe.enforce=0
    [0x2]            : [    26.621228] sysrq: SysRq : Trigger a crash
    [0x3]            : [    26.621254] Internal error: Oops: 96000044 [#1]
SMP
    [0x4]            : [    26.656655] pc : sysrq_handle_crash+0x24/0x30
    [0x5]            : [    26.753494]  sysrq_handle_crash+0x24/0x30
    [0x6]            : [    26.801441] Starting crashdump kernel...8J»=.
```

用于 `.Reverse()` 显示记录的最后一个事件。

dbgcmd

```
2: kd> dx @$cursession.Kernel.PrintKLog.Reverse().Take(5).Reverse()
@$cursession.Kernel.PrintKLog.Reverse().Take(5).Reverse()
   [0x0]           : [3147944.378367]  kthread+0x118/0x2a4
   [0x1]           : [3147944.381876]  ret_from_fork+0x10/0x18
   [0x2]           : [3147944.385747] Code: 78002507 36000042 39000107
d65f03c0 (cb0803e4)
   [0x3]           : [3147944.392221] SMP: stopping secondary CPUs
   [0x4]           : [3147944.397539] Starting crashdump kernel...
```

有关将 LINQ 查询与 dx 命令配合使用的详细信息，请参阅 将 LINQ 与调试器对象配合使用。

Linux 诊断扩展 - ELFBinComposition.dll

ELFBinComposition.dll中提供了以下 Linux 转储文件诊断扩展。

转储文件命令

这些命令可用于大多数转储文件。

`!dumpdebug` - 显示正在调试的核心转储的诊断。 这包括各种其他命令的输出。

`!ntprstatus` - 显示正在调试的核心转储中的NT_PRSTATUS记录。

`!vmcoreinfo` - 显示正在调试的内核核心转储（KDUMP）中的 VMCOREINFO 表。

ELF 转储命令

这些命令只能用于 ELF 核心转储文件。

`!corephdrs` - 显示正在调试的核心转储的程序标头表。

`!ntprpsinfo` - 显示正在调试的核心转储中的NT_PRPSINFO数据。

`!ntfile` - 显示正在调试的核心转储中的NT_FILE数据。

`!ntauxv` - 显示正在调试的核心转储中的NT_AUXV数据。

内核故障转储文件命令

这些命令只能在内核核心转储 （KDUMP） 文件上使用。

`!kdumpdescs` - 在 KDUMP 中显示页面范围和页面描述符的列表。

`!kdumppagerange <n>` - 显示 KDUMP 中第 n 个页面分组的信息。

`!kdumppfn <pfn>` - 显示有关页面框架 `<pfn>` 及其数据在 KDUMP 中的位置的信息。

其他诊断命令

`!cppex` - 使用 libstdc++的内部结构和 DWARF 符号显示有关当前正在进行的（未捕获和“just”捕获）C++异常的信息。

`!cppfilt [-n] <mangled name>` - 将一个C++混乱的名称解开，就好像它通过 c++filt 工具运行一样。 有关 C==filtr 工具的详细信息，请参阅 c++filt（1） — Linux 手册页。

`!rustdemangle <mangled name>` - 解开 Rust 混乱的名称。 有关 rust 符号名称混乱的详细信息，请参阅 Rust 符号管理 - RFC 2603。

另请参阅

Linux 符号和源

Linux 实时远程进程调试

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Linux 符号和源

项目 • 2024/04/18

本文介绍 WinDbg 如何支持标准 Linux 符号和源。 在 Linux 上支持调试需要 WinDbg 版本 1.2402.24001.0 或更高版本。

DebugInfoD 符号服务器

Window 调试器使用 DebugInfoD 标准自动下载适用于 Linux 的生成项目。 相比之下，DebugInfoD 是 Microsoft 符号服务器和源服务器技术的组合。 它允许基于生成 ID 自动下载三种项目类型（可执行文件 (ELF)、调试信息 (DWARF) 和源代码（代码）。现在，Linux 的各种分发版托管自己的 DebugInfoD 服务器，这些服务器提供一些项目类型。 ELFUTILS https://debuginfod.elfutils.org 中列出了各种 DebugInfoD 服务器。

有关 DebugInfoD 的一般信息，请访问此处：

- DebugInfoD 服务器
- Ubuntu 关于 debuginfod

`DebugInfoD*` 标记可以指向一个或多个 DebugInfoD 服务器，其中每个服务器 URL 的格式设置为`https://domain.com`，并用 `*` 分隔。 将按源路径中列出的顺序搜索服务器，并从第一个匹配的 URL 检索文件。

例如，可以这样设置符号路径。

```
.sympath+ DebugInfoD*https://debuginfod.elfutils.org
```

使用 `!sym noisy` 命令可以显示有关符号加载的信息。 有关详细信息，请参阅 !sym。

源路径命令（.srcpath, .lsrcpath（设置源路径））支持通过 `DebugInfoD*` 标记从 DebugInfoD 服务器检索文件，从而允许检索源代码项目。 例如，可以这样设置源路径。

```
.srcpath+ DebugInfoD*https://debuginfod.elfutils.org
```

有关详细信息，请参阅源代码扩展访问。

DWARF 符号

DWARF 是一种广泛使用的标准化调试数据格式。 DWARF 最初与可执行文件和可链接格式 (ELF) 一起设计的，但是它独立于对象文件格式。 有关详细信息，请参阅

https://en.wikipedia.org/wiki/DWARF；如需了解版本 5 标准，请参阅 DWARF 版本 5。

使用对象转储命令可以确定 DWARF 符号版本。 在本示例中，为版本 5。

linux

```
bob@BOB:/mnt/c/Users/BOB$ objdump -g DisplayGreeting | grep -A 2
'Compilation Unit @'
  Compilation Unit @ offset 0x0:
   Length:        0x285c (32-bit)
   Version:       5
```

WinDbg DWARF 支持

WinDbg 支持 DWARF 和 ELF 的以下用法。

- *Linux 用户模式* - 打开 Linux ELF 核心转储 (`-z <core dump>`)，并使用完整的专用 DWARF 符号执行事后调试和分析。
- *Linux 内核模式* - 打开 Linux 内核 (ELF VMCORE) 转储，并使用完整的专用 DWARF 符号执行事后调试和分析。
- *Linux 内核模式* - 打开 Linux 内核压缩的 KDUMP，并使用完整的专用 DWARF 符号执行事后调试和分析（WinDbg 仅支持 ZLIB 压缩的 KDUMP 文件。不支持 LZO 和 Snappy 压缩的 KDUMP。）
- 打开 ELF 图像 (`-z <ELF image>`)，并检查内容、反汇编等。
- *其他方案* - 了解混合 PE/ELF 环境中的 ELF 映像和 DWARF 符号（例如：调试 Windows 上加载的 Open Enclave 组件。有关详细信息，请参阅 Open Enclave 调试。）

WinDbg GDBServer Linux 支持

GNU 调试器 GDBServer 在 Linux 上用于支持 WinDbg 连接。 有关 GDBServer 的详细信息，请参阅 https://en.wikipedia.org/wiki/Gdbserver。 提供查看远程 gdb 调试文档的一站式位置是 - https://sourceware.org/gdb/current/onlinedocs/gdb#Remote-Debugging

有关将 GDBServer 与 WinDbg 配合使用以及代码演练的详细信息，请参阅 Linux 实时远程进程调试。 以下示例使用在适用于 Linux 的 Windows 子系统 (WSL) 下运行的 Ubuntu，但也可以使用其他 Linux 实现。

DWARF 实现

支持 DWARF 符号嵌入到原始映像（调试二进制文件）中，或剥离到单独的 ELF 映像（调试包）中。

为了使 Linux DWARF 堆栈审核成功，必须能够找到加载到 Linux 进程中的任何模块的原始二进制映像。

DWARF 符号/ELF 映像（无论剥离与否）可以通过调试器的 sympath 或符号服务器（通过 GNU 生成 ID 哈希按 .NET Core 编制索引）找到。

可以通过 Linux 样式调试包安装找到 DWARF 符号。 这是由符号路径中名为 `.build-id` 的目录给出的。 下面是根据 GNU 生成 ID 哈希的第一个字节命名的目录。 在每个这样的目录下都有一个名为 `<remaining 18 bytes of GNU Build ID hash>`.debug 的文件。

当调试器打开 DWARF 符号时，将执行初始索引步骤，因为格式本身不包括必要的查找表。 对于大型 DWARF 符号集（例如：Linux 内核的专用 DWARF 信息），可能需要 10 - 30 秒。

!addsourcemap 用于从已知 repo / commit 中自动检索源

如果要调试从已知存储库和提交生成的组件，则使用扩展 `!addsourcemap` 调试器命令可以告知调试器你希望从已知 URL 自动检索源的给定模块和路径。 扩展的用法为：

`!addsourcemap <module> <local spec> <remote spec>`

其中：

`<module>` 是感兴趣的模块的名称。

`<local spec>` 是该模块中将通过 URL 查找的源的路径。 此路径应以通配符结尾。

`<remote spec>` 是查找与 `<local spec>` 匹配的文件的 URL。 此路径应以通配符结尾，通配符将被 `<local spec>` 中的通配符如何匹配特定的源路径所替换。

若要设置源映射，请使用 lm（列出加载的模块）确认模块是否存在。 然后确定源的远程位置。

此示例将 vmlinux 模块设置为 GitHub 上可用的特定内部版本。

```
dbgcmd
```

```
0:000> !addsourcemap vmlinux /build/linux/*
https://raw.githubusercontent.com/torvalds/linux/6e61dde82e8bfe65e8ebbe43da4
5e615bc529236/
Source map /build/linux/* ->
https://raw.githubusercontent.com/torvalds/linux/6e61dde82e8bfe65e8ebbe43da4
5e615bc529236/ successfully added
```

发出 sourcemap 命令后，许多操作将触发源加载。例如，使用 .reload 命令来回切换帧或重新加载。 之后，将自动从 GitHub 中提取源代码。

!sourcemaps

使用 `!sourcemaps` 命令列出现有的源映射。

dbgcmd

```
0:000> !sourcemaps
Source maps for vmlinux.6:
    /build/linux/* ->
https://raw.githubusercontent.com/torvalds/linux/6e61dde82e8bfe65e8ebbe43da4
5e615bc529236/
```

!removesourcemaps

使用 `!removesourcemaps` 命令删除现有的源映射。

dbgcmd

```
0:000> !removesourcemaps vmlinux /build/linux/*
https://raw.githubusercontent.com/torvalds/linux/6e61dde82e8bfe65e8ebbe43da4
5e615bc529236/
1 source maps successfully removed
```

DWARF 符号疑难解答

如果要调试 Linux/Android 转储（或使用 DWARF 符号的其他目标），可能需要查看符号的原始内容，以了解局部变量、类型定义或函数定义不正确的原因。 为此，可以使用调试器具有的一些内置扩展命令来转储 DWARF 符号的原始内容。 此外，可以使用 Readelf 和 dumpdwarf 等 Linux 实用工具显示符号内部信息。

readelf 命令

在 Linux 命令提示符下使用 readelf 命令显示为 Linux 实时远程进程调试中创建的示例 DisplayGreeting 程序创建的调试版本 ID。 在此示例中，将返回 *aba822dd158b997b09903d4165f3dbfd37f5e5c1* 的内部版本 ID。

linux

```
Bob@BOB6:/mnt/c/Users/Bob$ readelf -n DisplayGreeting

Displaying notes found in: .note.gnu.property
  Owner                Data size        Description
  GNU                  0x00000020       NT_GNU_PROPERTY_TYPE_0
      Properties: x86 feature: IBT, SHSTK
        x86 ISA needed: x86-64-baseline

Displaying notes found in: .note.gnu.build-id
  Owner                Data size        Description
  GNU                  0x00000014       NT_GNU_BUILD_ID (unique build ID
bitstring)
    Build ID: aba822dd158b997b09903d4165f3dbfd37f5e5c1

Displaying notes found in: .note.ABI-tag
  Owner                Data size        Description
  GNU                  0x00000010       NT_GNU_ABI_TAG (ABI version tag)
    OS: Linux, ABI: 3.2.0
```

Readelf 可以与 grep 一起使用以返回符号版本。

linux

```
 readelf --debug-dump=info DisplayGreeting | grep -A 2 'Compilation Unit @'
  Compilation Unit @ offset 0x0:
   Length:        0x285c (32-bit)
   Version:       5
```

dwarfdump

dwarfdump linux 命令根据特定选项的要求输出或检查 DWARF 节。 使用 dwarfdump -h 可查看许多选项。

linux

```
bob@BOB6:/mnt/c/Users/BOB$ dwarfdump -h
```

有关在 Ubuntu 上使用 dwarfdump 的详细信息，请参阅 dwarfdump。

!diesym

此调试器命令将显示给定表达式（可以是地址、函数名称等）中具有可选指定递归级别的符号的 DIE（或 DIE 子树）。 它为给定地址中包含的符号（通常是函数，但也可能是数据等）定位 DIE，并执行 DIE 的诊断转储，类似于在符号上运行 dwarfdump 或 llvm-dwarfdump，并查找 DIE。

`!diesym [options] <expression>`

`-r#`：以递归方式转储 N 个级别。 通常，这是一个，只有 DIE 本身被转储。

`<expression>` - 用于定位 DIE 的地址由一个表达式给出。 它可以是一个平面十六进制地址（`0x<blah>`），也可以是一个其他唯一的函数名。

需要通过数据模型的标准评估进行评估。 使用 dx 命令验证模型表达式。 有关使用 dx 命令的详细信息，请参阅 dx（显示调试器对象模型表达式）。

dbgcmd

```
0:000> dx DisplayGreeting!GetCppConGreeting
DisplayGreeting!GetCppConGreeting                 :
DisplayGreeting!GetCppConGreeting+0x0 [Type: GetCppConGreeting]
```

显示示例 DisplayGreeting 程序 GetCppConGreeting 函数的 DIE 符号信息。

dbgcmd

```
0:000> !diesym DisplayGreeting!GetCppConGreeting
0x2816: DW_TAG_subprogram [^^^]
    DW_AT_external          (true)
    DW_AT_name              'GetCppConGreeting'
    DW_AT_decl_file         1 ('/mnt/c/Users/BOB/DisplayGreeting.cpp')
    DW_AT_decl_line         0x7
    DW_AT_decl_column       0x6
    DW_AT_linkage_name      '_Z17GetCppConGreetingPwm'
    DW_AT_low_pc            0x11E9
    DW_AT_high_pc           +0x3c (== 0x1225)
    DW_AT_frame_base        DW_OP_call_frame_cfa
    DW_AT_call_all_tail_calls   (true)
```

使用 -r2 选项可显示附加级别的 DIE 符号信息。

dbgcmd

```
0:000> !diesym -r2 DisplayGreeting!GetCppConGreeting
0x2816: DW_TAG_subprogram [^^^]
    DW_AT_external          (true)
    DW_AT_name              'GetCppConGreeting'
    DW_AT_decl_file         1 ('/mnt/c/Users/BOB/DisplayGreeting.cpp')
    DW_AT_decl_line         0x7
```

```
    DW_AT_decl_column       0x6
    DW_AT_linkage_name      '_Z17GetCppConGreetingPwm'
    DW_AT_low_pc            0x11E9
    DW_AT_high_pc           +0x3c (== 0x1225)
    DW_AT_frame_base        DW_OP_call_frame_cfa
    DW_AT_call_all_tail_calls   (true)

    0x2834: DW_TAG_formal_parameter [^^^]
        DW_AT_name              'buffer'
        DW_AT_decl_file         1 ('/mnt/c/Users/BOB/DisplayGreeting.cpp')
        DW_AT_decl_line         0x7
        DW_AT_decl_column       0x21
        DW_AT_type              (CU + 0x12f7 == 0x12f7)
        DW_AT_location          DW_OP_fbreg(-40)
```

!die

`!die` 将在 DWARF 调试部分中以可选指定的递归级别显示给定偏移量表达式处的任何 DIE 的 DIE（或 DIE 子树）。

`!die [-r#] [-t] -m <module base expression> <offset expression>`

`-r#`： 以递归方式转储 N 个级别。

`-t`： ： 如果 DIE 位于 .debug_types 的类型单元中，而不是 .debug_info 中的编译单元，则必须指定 -t 开关。

提供一个 `-m <module base expression>`，提供要查询的任何模块的基址。

`<offset expression>` 是 DIE 偏移量的大小。

在 Linux 提示符下，将 dwarfdump 与 -r 一起使用，输出 DWARF 文件的 *.debug_aranges* 部分，以查找 DIE 偏移量。

linux

```
bob@BOB6:/mnt/c/Users/BOB$ dwarfdump -r DisplayGreeting

.debug_aranges

COMPILE_UNIT<header overall offset = 0x00000000>:
< 0><0x0000000c>  DW_TAG_compile_unit
                    DW_AT_producer              GNU C++17 11.4.0 -
mtune=generic -march=x86-64 -g -fasynchronous-unwind-tables -fstack-
protector-strong -fstack-clash-protection -fcf-protection
                    DW_AT_language              DW_LANG_C_plus_plus_14
                    DW_AT_name                  DisplayGreeting.cpp
                    DW_AT_comp_dir              /mnt/c/Users/BOB
                    DW_AT_ranges                0x0000000c
```

```
      Offset of rnglists entries: 0x0000000c
      [ 0] start,end             0x000011e9 0x0000134a
      [ 1] start,end             0x0000134a 0x00001368
      [ 2] start,end             0x00001368 0x0000137b
      [ 3] start,end             0x0000137b 0x0000138d
      [ 4] end of list
                  DW_AT_low_pc                0x00000000
                  DW_AT_stmt_list             0x00000000


arange starts at 0x000011e9, length of 0x00000161, cu_die_offset =
0x0000000c
arange starts at 0x0000134a, length of 0x0000001e, cu_die_offset =
0x0000000c
arange starts at 0x00001368, length of 0x00000013, cu_die_offset =
0x0000000c
arange starts at 0x0000137b, length of 0x00000012, cu_die_offset =
0x0000000c
```

记下 `0x0000000c` 的 DW_AT_ranges 值。 在调试器中，使用该偏移值和 DisplayGreeting 的模块名称来显示 DIE 符号信息。

dbgcmd

```
0:000> !die -m DisplayGreeting 0x0000000c
0xc: DW_TAG_compile_unit [^^^]
    DW_AT_producer          'GNU C++17 11.4.0 -mtune=generic -march=x86-64 -
g -fasynchronous-unwind-tables -fstack-protector-strong -fstack-clash-
protection -fcf-protection'
    DW_AT_language          0x21
    DW_AT_name
    DW_AT_comp_dir
    DW_AT_ranges
        [0x11e9 - 0x134a)
        [0x134a - 0x1368)
        [0x1368 - 0x137b)
        [0x137b - 0x138d)
    DW_AT_low_pc            0x0
    DW_AT_stmt_list
```

!dieancestry

`!dieancestry` 命令的行为类似于 `!die`，只是它沿着 DIE 树向上移动到包含编译单元或类型单元，而不是沿着树向下移动。

`!dieancestry [-r#] [-t] -m <module base expression> <offset expression>`

`-r#`： 以递归方式转储 N 个级别。

提供一个 `-m <module base expression>`，提供要查询的任何模块的基址。

`<offset expression>` 是 DIE 偏移量的大小。

示例：

dbgcmd

```
0:000> !dieancestry -m DisplayGreeting 0x0000000c
0xc: DW_TAG_compile_unit [^^^]
    DW_AT_producer          'GNU C++17 11.4.0 -mtune=generic -march=x86-64 -
g -fasynchronous-unwind-tables -fstack-protector-strong -fstack-clash-
protection -fcf-protection'
    DW_AT_language          0x21
    DW_AT_name
    DW_AT_comp_dir
    DW_AT_ranges
        [0x11e9 - 0x134a)
        [0x134a - 0x1368)
        [0x1368 - 0x137b)
        [0x137b - 0x138d)
    DW_AT_low_pc            0x0
    DW_AT_stmt_list
```

请注意，可以单击链接（例如，到父节点或同级节点的链接），以便进一步遍历 DWARF 符号树。

dbgcmd

```
0:000> !die -r2 -m 0x555555554000 0xc
0xc: DW_TAG_compile_unit [^^^]
    DW_AT_producer          'GNU C++17 11.4.0 -mtune=generic -march=x86-64 -
g -fasynchronous-unwind-tables -fstack-protector-strong -fstack-clash-
protection -fcf-protection'
    DW_AT_language          0x21
    DW_AT_name
    DW_AT_comp_dir
    DW_AT_ranges
        [0x11e9 - 0x134a)
        [0x134a - 0x1368)
        [0x1368 - 0x137b)
        [0x137b - 0x138d)
    DW_AT_low_pc            0x0
    DW_AT_stmt_list

    0x2a: DW_TAG_namespace [^^^]
        DW_AT_name              'std'
        DW_AT_decl_file         9
('/usr/include/c++/11/bits/exception_ptr.h')
        DW_AT_decl_line         0x116
        DW_AT_decl_column       0xb
        DW_AT_sibling           (CU + 0xf01 == 0xf01)
```

```
    0xf01: DW_TAG_base_type [^^^]
        DW_AT_byte_size         0x1
        DW_AT_encoding          DW_ATE_boolean (2)
        DW_AT_name              'bool'

    0xf08: DW_TAG_base_type [^^^]
        DW_AT_byte_size         0x8
        DW_AT_encoding          DW_ATE_unsigned (7)
        DW_AT_name              'long unsigned int'

...
```

未显示所有输出。

!dwunwind

`!dwunwind` 有点类似于 PE 映像的 .fnent（显示函数数据）。 它显示表达式给定的地址的 DWARF 展开规则。 它也类似于 readelf --unwind 命令，后者在可用时显示展开信息。

`!dwunwind <expression>`

此示例显示 DisplayGreeting 程序中 GetCppConGreeting 函数的展开规则。

dbgcmd

```
0:000> !dwunwind DisplayGreeting!GetCppConGreeting
DW_FRAME_SAME_VAL: 0('rax'), 1('rdx'), 2('rcx'), 3('rbx'), 4('rsi'),
5('rdi'), 6('rbp'), 7('rsp'), 8('r8'), 9('r9'), 10('r10'), 11('r11'),
12('r12'), 13('r13'), 14('r14'), 15('r15')
0('CFA'): DW_EXPR_OFFSET 7('rsp') + 8
16('<Return Address>'): DW_EXPR_OFFSET 12290('CFA') + -8
```

这将显示指令指针寄存器的展开堆栈。

dbgcmd

```
0:000> !dwunwind @rip
DW_FRAME_SAME_VAL: 0('rax'), 1('rdx'), 2('rcx'), 4('rsi'), 5('rdi'),
7('rsp'), 8('r8'), 9('r9'), 10('r10'), 11('r11'), 14('r14'), 15('r15')
0('CFA'): DW_EXPR_OFFSET 7('rsp') + 208
3('rbx'): DW_EXPR_OFFSET 12290('CFA') + -40
6('rbp'): DW_EXPR_OFFSET 12290('CFA') + -32
12('r12'): DW_EXPR_OFFSET 12290('CFA') + -24
13('r13'): DW_EXPR_OFFSET 12290('CFA') + -16
16('<Return Address>'): DW_EXPR_OFFSET 12290('CFA') + -8
```

下面是程序计数器示例。

dbgcmd

```
   0:000> !dwunwind @pc
   DW_FRAME_SAME_VAL: 0('x0'), 1('x1'), 2('x2'), 3('x3'), 4('x4'), 5('x5'),
6('x6'), 7('x7'), 8('x8'), 9('x9'), 10('x10'), 11('x11'), 12('x12'),
13('x13'), 14('x14'), 15('x15'), 16('x16'), 17('x17'), 18('x18'), 31('sp'),
32('pc')
   0('CFA'): DW_EXPR_OFFSET 31('sp') + 208
   19('x19'): DW_EXPR_OFFSET 1436('CFA') + -192
   20('x20'): DW_EXPR_OFFSET 1436('CFA') + -184
   21('x21'): DW_EXPR_OFFSET 1436('CFA') + -176
   22('x22'): DW_EXPR_OFFSET 1436('CFA') + -168
   23('x23'): DW_EXPR_OFFSET 1436('CFA') + -160
   24('x24'): DW_EXPR_OFFSET 1436('CFA') + -152
   25('x25'): DW_EXPR_OFFSET 1436('CFA') + -144
   26('x26'): DW_EXPR_OFFSET 1436('CFA') + -136
   27('x27'): DW_EXPR_OFFSET 1436('CFA') + -128
   28('x28'): DW_EXPR_OFFSET 1436('CFA') + -120
   29('fp'): DW_EXPR_OFFSET 1436('CFA') + -208
   30('lr'): DW_EXPR_OFFSET 1436('CFA') + -200
```

!dietree

在给定的递归级别上为给定模块转储 DIE 树，类似于在符号上运行 dwarfdump 或 llvm-dwarfdump，并查找 DIE。

`!dietree [OPTIONS] -m <module base> <offset expression>`

`-r#`：指定递归级别

`-t`：转储 .debug_types 而不是 .debug_info

包含 DIE 的模块的模块基必须由 `-m <expression>` 选项给出。

`<offset expression>` 是 DIE 偏移量的大小。

示例：

dbgcmd

```
0:000> !dietree -m DisplayGreeting 0x0000000c
0xc: DW_TAG_compile_unit [^^^]
    DW_AT_producer          'GNU C++17 11.4.0 -mtune=generic -march=x86-64 -
g -fasynchronous-unwind-tables -fstack-protector-strong -fstack-clash-
protection -fcf-protection'
    DW_AT_language          0x21
    DW_AT_name
```

```
    DW_AT_comp_dir
    DW_AT_ranges
        [0x11e9 - 0x134a)
        [0x134a - 0x1368)
        [0x1368 - 0x137b)
        [0x137b - 0x138d)
    DW_AT_low_pc            0x0
    DW_AT_stmt_list
```

使用 -r2 选项可以在 dietree 中显示其他值。

dbgcmd

```
0:000> !dietree -r2 -m DisplayGreeting 0x0000000c
0xc: DW_TAG_compile_unit [^^^]
    DW_AT_producer          'GNU C++17 11.4.0 -mtune=generic -march=x86-64 -
g -fasynchronous-unwind-tables -fstack-protector-strong -fstack-clash-
protection -fcf-protection'
    DW_AT_language          0x21
    DW_AT_name
    DW_AT_comp_dir
    DW_AT_ranges
        [0x11e9 - 0x134a)
        [0x134a - 0x1368)
        [0x1368 - 0x137b)
        [0x137b - 0x138d)
    DW_AT_low_pc            0x0
    DW_AT_stmt_list

    0x2a: DW_TAG_namespace [^^^]
        DW_AT_name              'std'
        DW_AT_decl_file         9
('/usr/include/c++/11/bits/exception_ptr.h')
        DW_AT_decl_line         0x116
        DW_AT_decl_column       0xb
        DW_AT_sibling           (CU + 0xf01 == 0xf01)

    0xf01: DW_TAG_base_type [^^^]
        DW_AT_byte_size         0x1
        DW_AT_encoding          DW_ATE_boolean (2)
        DW_AT_name              'bool'

    0xf08: DW_TAG_base_type [^^^]
        DW_AT_byte_size         0x8
        DW_AT_encoding          DW_ATE_unsigned (7)
        DW_AT_name              'long unsigned int'

    0xf0f: DW_TAG_base_type [^^^]
        DW_AT_byte_size         0x1
        DW_AT_encoding          DW_ATE_unsigned_char (8)
        DW_AT_name              'unsigned char'
```

```
...
```

未显示所有输出。 请注意，可以单击链接（例如，到同级节点的链接），以便进一步遍历 DWARF 符号树。

!dielocal

为名为“name”的局部变量定位 DIE，并执行 DIE 的诊断转储，类似于在符号上运行 dwarfdump 或 llvm-dwarfdump，并查找DIE。

`!dielocal [options] <name>`

`-r#`：以递归方式转储 N 个级别。 通常，这是一个，只有 DIE 本身被转储。

`<name>`：名为“name”的局部变量。

示例：

dbgcmd

```
0:000> !dielocal greeting
0x2806: DW_TAG_variable [^^^]
    DW_AT_name              'greeting'
    DW_AT_decl_file         1 ('/mnt/c/Users/BOB/DisplayGreeting.cpp')
    DW_AT_decl_line         0xf
    DW_AT_decl_column       0x1d
    DW_AT_type              (CU + 0xb18 == 0xb18)
    DW_AT_location          DW_OP_fbreg(-240)
```

另请参阅

源代码扩展访问

ELFUTILS debuginfod

DWARF 版本 5

使用符号

!sym

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Linux 故障转储

项目 • 2024/11/18

本文介绍如何创建和查看不同类型的 Linux 故障转储文件。 查看 Linux 故障转储需要 WinDbg 版本 1.2402.24001.0 或更高版本。

在 WinDbg 中打开 Linux（非 Windows）核心转储时，基本调试器命令应全部正常工作，但引用 Windows 结构的扩展和特定于 Windows 的命令将不起作用。

支持的故障转储文件

Linux 内核转储

在 Windows 调试器中提供了打开 Linux 内核压缩的 KDUMP，并使用完整的专用 DWARF 符号执行事后调试和分析。

WinDbg 仅支持 ZLIB 压缩的 KDUMP 文件。 不支持 LZO 和 Snappy 压缩 KDUMP。

有关 Linux KDUMP 的一般信息，请参阅 KDump （Linux） 维基百科页和 核心转储。

ELF 核心转储

作为支持 Open Enclave 的一部分，WinDbg 可以从 Enclave 和 Linux 应用程序打开 ELF 核心转储和二进制文件以及 DWARF 符号（DWARF 5 不受支持）。 有关 Open Enclave 的详细信息，请参阅 Open Enclave 调试。

进程转储

支持单个进程转储。 有多种方法可以收集进程转储，包括 Linux Sysinternals ProcDump for Linux 实用工具。 另一个选项是使用 GNU 调试器 - GDBServer 生成核心转储。 有关 GDBServer 的详细信息，请参阅 https://en.wikipedia.org/wiki/Gdbserver。 Sourceware 网站提供了远程 gdb 调试文档 - 调试远程程序。

启用 Linux NATVIS 可视化和 DML 链接遍历

Windows 调试器中的标准模板库 （STL） 扩展性由 NatVis 文件提供， `stl.natvis` 该文件可了解随 Visual Studio 和 Windows 一起随附的许多版本的 STL。 有关 NATVIS 的常规

信息，请参阅 NatVis 中的本机调试器对象。 用于 Linux 组件（GCC 或 LLDB）的 STL 版本非常不同。

若要为 Linux 启用 NATVIS 可视化和 DML 链接遍历优化，请先卸载默认 natvis 文件 .scriptunload `stl.natvis`。

然后 .scriptload 文件 `gstl.natvis` 。 使用 .scriptlist 确认是否 `gstl.natvis` 处于活动状态。

dbgcmd

```
0: kd> .scriptlist
Command Loaded Scripts:
...
    NatVis script from
'C:\Users\Bob\AppData\Local\dbg\UI\2402.24001.0\amd64\Visualizers\gstl.natvi
s'
```

有关使用 DML 的详细信息，请参阅 使用 DML 自定义调试器输出。

DisplayGreeting 应用的单个进程核心转储

此示例演示如何使用 gdb 创建单个进程核心转储。 有关将 GDBServer 与 WinDbg 配合使用以及代码演练的详细信息，请参阅 Linux 实时远程进程调试。 有关 DisplayGreeting 的示例代码，请参阅 C++应用演练。

找到所需的进程

我们可以使用 `ps -A` 命令列出 Linux 中的所有进程，或者将 -f 选项与 pgrep 一起使用，因为我们知道我们要查找 DisplayGreeting 应用。

dbgcmd

```
$ pgrep -f DisplayGreeting
9382
```

在本示例演练中，它显示了 9382 的进程 ID。

使用 gdb 附加到进程并生成核心转储

使用 gdb 附加到进程。

dbgcmd

```
$ gdb -p 9382
```

显示 gdb 命令的 `generate-core-file` 帮助。

```dbgcmd
(gdb) help generate-core-file
Save a core file with the current state of the debugged process.
Argument is optional filename.  Default filename is 'core.<process_id>'.
```

然后在 （gdb） 提示符下，生成包含默认文件名的进程核心转储文件。

```dbgcmd
(gdb) generate-core-file
Saved corefile core.9382
(gdb) quit
```

加载并检查 Linux 进程核心转储

使用 WinDbg 中的“打开转储文件”菜单选项加载生成的核心转储。

在调试器会话中添加源和符号路径

若要查看源代码和变量，请设置符号和源路径。 有关设置符号路径的一般信息，请参阅使用符号。 有关 Linux 符号的详细信息，请参阅 Linux 符号和源。

使用 `.sympath` 将符号路径添加到调试器会话。 在此 WSL Linux Ubuntu 示例中，DisplayGreetings 代码和符号可用于名为 Bob 的用户。

`\\wsl$\Ubuntu\mnt\c\Users\Bob\`

在 WSL 中，此目录映射到 Windows OS 位置： `C:\Users\Bob\` 因此使用这两个命令。

`.sympath C:\Users\Bob\`

`.srcpath C:\Users\Bob\`

有关在 Windows 中访问 WSL 文件系统的详细信息，请参阅 WSL 的文件权限。

若要从其他 Linux OS 符号中受益，请使用 .sympath 位置添加 DebugInfoD 符号。

`.sympath+ DebugInfoD*https://debuginfod.elfutils.org`

`.reload`使用命令重新加载符号。

此外，还支持从 DebugInfoD 服务器自动下载源，这些源支持返回该项目类型。 若要利用此功能，请使用 .srcpath 添加 elfutils 服务器。

```
.srcpath+ DebugInfoD*https://debuginfod.elfutils.org
```

检查进程转储

使用 `lm` 命令确认转储文件是否包含 DisplayGreeting 应用。

dbgcmd

```
0:000> lm
start             end                 module name
00005555`55554000 00005555`55558140   DisplayGreeting T (service symbols:
DWARF Private Symbols)        c:\users\bob\DisplayGreeting
00007fff`f7a54000 00007fff`f7a732e8   libgcc_s_so   (deferred)
00007fff`f7a74000 00007fff`f7b5a108   libm_so    (deferred)
00007fff`f7b5b000 00007fff`f7d82e50   libc_so  T (service symbols: DWARF
Private Symbols)        C:\ProgramData\Dbg\sym\_.debug\elf-buildid-sym-
a43bfc8428df6623cd498c9c0caeb91aec9be4f9\_.debug
00007fff`f7d83000 00007fff`f7fae8c0   libstdc___so   (deferred)
00007fff`f7fc1000 00007fff`f7fc1000   linux_vdso_so   (deferred)
00007fff`f7fc3000 00007fff`f7ffe2d8   ld_linux_x86_64_so T (service symbols:
DWARF Private Symbols)        C:\ProgramData\Dbg\sym\_.debug\elf-buildid-
sym-9718d3757f00d2366056830aae09698dbd35e32c\_.debug
```

请注意，第一个命令执行可能需要一些时间，因为调试符号将加载到缓存中。 除了通过符号服务器或本地搜索路径查找符号和二进制文件之外，由于 GDBServer 集成，如果无法在本地找到这些文件，它可能会从远程文件系统加载这些文件。 此操作通常比从 symsrv 或本地搜索路径获取符号慢。

`x`使用命令显示 DisplayGreeting 中可用的函数。

dbgcmd

```
0:000> x /D /f DisplayGreeting!*
 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

*** WARNING: Unable to verify timestamp for DisplayGreeting
00005651`7935b331 DisplayGreeting!_GLOBAL__sub_I__Z17GetCppConGreetingPwm
(void)
00005651`7935b2db DisplayGreeting!__static_initialization_and_destruction_0
(int, int)
00005651`7935b37b DisplayGreeting!std::__array_traits<wchar_t, 50>::_S_ptr
(wchar_t (*)[50])
00005651`7935b368 DisplayGreeting!std::array<wchar_t, 50>::size
(std::array<wchar_t, 50> *)
```

```
00005651`7935b34a DisplayGreeting!std::array<wchar_t, 50>::data
(std::array<wchar_t, 50> *)
00005651`7935b225 DisplayGreeting!main (void)
00005651`7935b1e9 DisplayGreeting!GetCppConGreeting (wchar_t *, size_t)
```

`dx`使用命令查看局部变量*问候语*。

dbgcmd

```
0:000> dx greeting
...
Error: Unable to bind name 'greeting'
```

由于在进行转储时尚未使用参数问候语，因此在转储文件中不可用。

`dx`使用命令检查转储文件中可用的进程。

dbgcmd

```
:000> dx @$cursession.Processes.Take(30)
@$cursession.Processes.Take(30)
    [0x24a6]         : DisplayGreeting [Switch To]

Click on the `[Switch To]` DML link to switch to the 9382 process.

```dbgcmd
0:000> dx -s @$cursession.Processes.Take(30)[9382].SwitchTo()
0:000> dx -r1 @$cursession.Processes.Take(30)[9382]
@$cursession.Processes.Take(30)[9382]                 : DisplayGreeting
[Switch To]
    Name             : DisplayGreeting
    Id               : 0x24a6
    Index            : 0x0
    Handle           : 0x24a6
    Threads
    Modules
    Environment
    Direct3D
    Attributes
    Devices
    Io
    Memory
    TTD
    GroupedStacks
```

若要查看有关线程和模块的信息，请单击输出中生成的 DML 链接，或键入类似于此命令的命令进行故障转储。

dbgcmd
```

```
0:000> dx -r1 @$cursession.Processes.Take(30)[9382].Threads
@$cursession.Processes.Take(30)[9382].Threads
    [0x24a6]         [Switch To]
0:000> dx -r1 @$cursession.Processes.Take(30)[9382].Modules
@$cursession.Processes.Take(30)[9382].Modules
    [0x0]            : /mnt/c/Users/Bob/DisplayGreeting
    [0x1]            : /usr/lib/x86_64-linux-gnu/libgcc_s.so.1
    [0x2]            : /usr/lib/x86_64-linux-gnu/libm.so.6
    [0x3]            : /usr/lib/x86_64-linux-gnu/libc.so.6
    [0x4]            : /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.30
    [0x5]            : /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
    [0x6]            : linux-vdso.so.1
```

使用 ELF/CORE 诊断扩展显示转储文件信息

使用 Linux 诊断扩展 - ELFBinComposition.dll显示转储文件信息。 例如，用于 `!dumpdebug` 确认这是 ELF 用户核心转储，并显示其他信息。

dbgcmd

```
0:000> !dumpdebug
Dump Diagnostics: Format = ELF User Core
**************************************************************************
****
File Mapping Size:           0x151d78 (1 Mb)
Highest Memory Offset:       0x14e5f0 (1 Mb)
...
```

用于 `!ntprpsinfo` 显示NT_PRPSINFO数据。

dbgcmd

```
0:000> !ntprpsinfo
NT_PRPSINFO (process info):
    state: 0, sname: t, zomb: 0, nice: 0, flag: 0x4040000019999999
    uid: 1000, gid: 1000, pid: 9382, ppid: 388, pgrp: 9382, sid: 388
    fname: DisplayGreeting
    psargs: ./DisplayGreeting
```

内核 KDump 故障转储

在 Linux 中创建故障转储文件有多种方法。 例如，内核故障转储中 介绍了 Ubuntu Linux 的一个选项。

其他选项包括使用 kexectools 来启用 Kdump。 有关详细信息，请参阅 KDump（Linux）。 如果已启用 kdump，则可以验证 kdump 是否处于活动状态并使用它运行 `systemctl status kdump`。

在测试系统上触发 OS 崩溃后，将创建故障转储文件。

加载并检查 Linux OS 故障转储

使用“打开转储文件”菜单选项加载生成的 kdump。

如上一部分所述，通过加载 `gstl.natvis` 文件启用 NATVIS 可视化和 DML 链接遍历优化 Linux。

使用 ELF Bin 组合命令分析 Linux 内核转储

若要能够使用其他 ELF Bin 组合命令，请使用 .chain 命令确认加载了 ELFBinComposition.dll。

dbgcmd

```
0: kd> .chain
Extension DLL chain:
    ELFBinComposition: image 10.0.27606.1000, API 0.0.0,
        [path:
C:\Users\Bob\AppData\Local\dbg\UI\Fast.20240423.1\amd64\winext\ELFBinComposi
tion.dll]
...
```

如果未加载 ELFBinComposition，请使用 .load 加载它。 有关详细信息，请参阅 .load、.loadby （加载扩展 DLL）

`!ELFBinComposition.dumpdebug`使用命令显示有关已加载转储文件的信息。 在此示例中，已加载 ELF 用户核心转储文件。

dbgcmd

```
0: kd> !ELFBinComposition.dumpdebug
Dump Diagnostics: Format = Kernel KDump
**************************************************************************
****
File Mapping Size:           0x3b34090 (59 Mb)
Highest Memory Offset:       0x3b34090 (59 Mb)
```

使用 ELFBinComposition `!vmcoreinfo` 显示正在调试的 Linux 内核核心转储（KDUMP）中的 VMCOREINFO 表。

dbgcmd

```
0: kd> !vmcoreinfo
VMCOREINFO:
    OSRELEASE=6.5.0-25-generic
    BUILD-ID=8567ad7c7c2f78f3654f6cc90a9e1b3f9c3a4b32
    PAGESIZE=4096
    SYMBOL(init_uts_ns)=ffffded86e11b388
    OFFSET(uts_namespace.name)=0
    SYMBOL(node_online_map)=ffffded86dcceb40
    SYMBOL(swapper_pg_dir)=ffffded86d143000
    SYMBOL(_stext)=ffffded86ace0000
    SYMBOL(vmap_area_list)=ffffded86de48140
    SYMBOL(mem_section)=ffff0f2e1efe4600
    LENGTH(mem_section)=8192
...
```

使用从 `!kdumppagerange` 零开始转储文件的第一部分转储。

dbgcmd

```
0: kd> !kdumppagerange 0
    PFNs [0x540e0, 0x55643) -> Descs [0x0, 0x1563): File Offsets [0x307430,
0xeeb37a) 0xbe3f4a bytes across 5475 pages as ZLIB
    PFNs [0x55643, 0x55650) -> Descs [0x1563, 0x1570): File Offsets
[0x306430, 0x307430) 0x1000 bytes across 13 duplicate pages as Uncompressed
    PFNs [0x55650, 0x556d6) -> Descs [0x1570, 0x15f6): File Offsets
[0xeeb37a, 0xf0c405) 0x2108b bytes across 134 pages as ZLIB
    PFNs [0x556d6, 0x556dc) -> Descs [0x15f6, 0x15fc): File Offsets
[0xf0c405, 0xf12405) 0x6000 bytes across 6 pages as Uncompressed
    PFNs [0x556dc, 0x55e98) -> Descs [0x15fc, 0x1db8): File Offsets
[0xf12405, 0x1216d1b) 0x304916 bytes across 1980 pages as ZLIB
    PFNs [0x55e98, 0x55ea4) -> Descs [0x1db8, 0x1dc4): File Offsets
[0x1216d1b, 0x1222d1b) 0xc000 bytes across 12 pages as Uncompressed
    PFNs [0x55ea4, 0x56542) -> Descs [0x1dc4, 0x2462): File Offsets
[0x1222d1b, 0x14ba138) 0x29741d bytes across 1694 pages as ZLIB
    PFNs [0x56542, 0x56543) -> Descs [0x2462, 0x2463): File Offsets
[0x306430, 0x307430) 0x1000 bytes across 1 pages as Uncompressed
    PFNs [0x56543, 0x56544) -> Descs [0x2463, 0x2464): File Offsets
[0x14ba138, 0x14ba194) 0x5c bytes across 1 pages as ZLIB
    PFNs [0x56544, 0x5654f) -> Descs [0x2464, 0x246f): File Offsets
[0x306430, 0x307430) 0x1000 bytes across 11 duplicate pages as Uncompressed
```

！ kdumppagerange 的输出显示各种页面帧 （PFN） 值。 我们可以选择感兴趣的一个，并使用该 `!kdumppfn <PFN>` 信息显示有关 PFN 及其数据在 KDUMP 中的位置的信息。

dbgcmd

```
0: kd> !kdumppfn 0x540e0
    Page frame 0x540e0 = File offset [0x307430, 0x307b9f) 0x76f bytes as
```

```
ZLIB...
```

检查转储文件

`k`使用命令显示调用堆栈，调查发生崩溃时正在运行的代码。

dbgcmd

```
6: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff0000`0bc3bc90 ffff0000`085161f8     vmlinux!sysrq_handle_crash+0x24
[/usr/src/kernel/drivers/tty\sysrq.c @ 147]
01 ffff0000`0bc3bca0 ffff0000`08516824     vmlinux!__handle_sysrq+0x88
[/usr/src/kernel/drivers/tty\sysrq.c @ 583]
02 ffff0000`0bc3bcb0 ffff0000`08308990     vmlinux!write_sysrq_trigger+0xb4
[/usr/src/kernel/drivers/tty\sysrq.c @ 1110]
03 ffff0000`0bc3bcf0 ffff0000`08290070     vmlinux!proc_reg_write+0x80
[/usr/src/kernel/fs/proc\inode.c @ 245]
04 ffff0000`0bc3bd10 ffff0000`0829039c     vmlinux!__vfs_write+0x60
[/usr/src/kernel/fs\read_write.c @ 490]
05 ffff0000`0bc3bd50 ffff0000`08290704     vmlinux!vfs_write+0xac
[/usr/src/kernel/fs\read_write.c @ 550]
06 ffff0000`0bc3be00 ffff0000`082907a4     vmlinux!ksys_write+0x74
[/usr/src/kernel/fs\read_write.c @ 599]
07 (Inline Function) --------`--------     vmlinux!__do_sys_write+0xc
[/usr/src/kernel/fs\read_write.c @ 608]
08 (Inline Function) --------`--------     vmlinux!__se_sys_write+0xc
[/usr/src/kernel/fs\read_write.c @ 608]
09 ffff0000`0bc3be40 ffff0000`08095904     vmlinux!__arm64_sys_write+0x24
[/usr/src/kernel/fs\read_write.c @ 608]
0a ffff0000`0bc3be90 ffff0000`080834c8     vmlinux!el0_svc_handler+0x94
0b ffff0000`0bc3beb0 00000000`00000000     vmlinux!el0_svc+0x8
```

`dx`使用命令检查转储文件。 例如，使用此命令查看前 30 个进程。

dbgcmd

```
6: kd> dx @$cursession.Processes.Take(30)
@$cursession.Processes.Take(30)
    [0x0]            : swapper/0 [Switch To]
    [0x1]            : systemd [Switch To]
    [0x2]            : kthreadd [Switch To]
    [0x3]            : rcu_gp [Switch To]
    [0x4]            : rcu_par_gp [Switch To]
    [0x5]            : kworker/0:0 [Switch To]
    [0x6]            : kworker/0:0H [Switch To]
    [0x7]            : kworker/u16:0 [Switch To]
    [0x8]            : mm_percpu_wq [Switch To]
    [0x9]            : ksoftirqd/0 [Switch To]
    [0xa]            : rcu_sched [Switch To]
```

```
    [0xb]            : rcu_bh [Switch To]
    [0xc]            : migration/0 [Switch To]
...
```

单击 DML 链接或使用类似于此链接的命令查看感兴趣的进程的线程。

dbgcmd

```
6: kd> dx @$cursession.Processes[0x1a].Threads
@$cursession.Processes[0x1a].Threads
    [0x1a]           [Switch To]
6: kd> dx @$cursession.Processes[0x1a].Threads[0x1a]
@$cursession.Processes[0x1a].Threads[0x1a]                 [Switch To]
    KernelObject     [Type: thread_struct]
    Id               : 0x1a
    Stack
    Registers
    Environment
    Analysis
    WaitChain
    Scheduling
    IRPs
...
```

提供了每个线程的其他信息，如下所示。

dbgcmd

```
6: kd> dx @$cursession.Processes[0x1a].Threads[0x1a].KernelObject
@$cursession.Processes[0x1a].Threads[0x1a].KernelObject
[Type: thread_struct]
    [+0x000] cpu_context      [Type: cpu_context]
    [+0x070] uw               [Type: <unnamed-tag>]
    [+0x290] fpsimd_cpu       : 0x100 [Type: unsigned int]
    [+0x298] sve_state        : 0x0 [Type: void *]
    [+0x2a0] sve_vl           : 0x0 [Type: unsigned int]
    [+0x2a4] sve_vl_onexec    : 0x0 [Type: unsigned int]
    [+0x2a8] fault_address    : 0x0 [Type: long unsigned int]
    [+0x2b0] fault_code       : 0x0 [Type: long unsigned int]
    [+0x2b8] debug            [Type: debug_info]
6: kd> dx -s @$cursession.Processes[0x1a].Threads[0x1a].SwitchTo()
Process ffff8008`0f894380 has invalid page directories
```

使用 LinuxKernel.js 脚本分析 Linux 内核转储

LinuxKernel.js调试器扩展包含一组命令，旨在与 Linux 崩溃实用工具中用于打开和分析 Linux 内核模式崩溃的命令类似。

若要使用该脚本，请先加载该脚本。

dbgcmd

```
0: kd> .scriptload LinuxKernel.js
JavaScript script successfully loaded from
'C:\Users\Bob\AppData\Local\dbg\UI\Fast.20240423.1\amd64\winext\LinuxKernel.
js'
```

有关使用加载脚本的详细信息，请参阅 JavaScript 调试器脚本。

！文件

使用转 `!files` 储文件中显示有关 Linux 文件结构的信息。 它类似于崩溃文件命令。

dbgcmd

```
6: kd> !files
@$files()                  : Files for process 'sh' (pid 545) root dir = '/'
working dir = '/home/root'
    [0x0]             : /dev/ttyS0 [Type: file]
    [0x1]             : /proc/sysrq-trigger [Type: file]
    [0x2]             : /dev/ttyS0 [Type: file]
    [0xa]             : /dev/ttyS0 [Type: file]
    [0xff]            : /dev/ttyS0 [Type: file]
```

`!files` 语法：

`!files [<arg>]`

没有 `[<arg>]` - 等效于“文件”- 提供当前进程文件列表

`[<arg>]`：

`pid` - 为给定进程 ID 提供文件列表

`64-bit num` - 为给定地址的任务提供文件列表

`<task struct [*]>` - 按对象为给定任务结构提供文件列表

`<process object>` - 为进程对象表示的任务提供文件列表

！安装

使用转 `!mount` 储文件中显示有关 Linux 文件结构的信息。

dbgcmd

```
6: kd> !mount
@$mount()
    [0x0]            : (rootfs) rootfs at / [Type: mount]
    [0x1]            : (squashfs) /dev/mapper/nested_rootfs at / [Type:
mount]
    [0x2]            : (sysfs) sysfs at /sys [Type: mount]
    [0x3]            : (proc) proc at /proc [Type: mount]
    [0x4]            : (devtmpfs) devtmpfs at /dev [Type: mount]
    [0x5]            : (securityfs) securityfs at /kernel/security [Type:
mount]
```

`!mount` 语法：

没有 `[<arg>]` - 等效于“mount”命令 - 显示装载的文件系统

`[<arg>]`：

`pid` - 为具有给定 pid 的进程命名空间提供装载的文件系统

`64-bit num` - 为地址提供的task_struct命名空间提供装载的文件系统

`<task struct [*]>` - 为给定task_struct的命名空间提供装载的文件系统

`<process object>` - 为进程表示的任务命名空间提供装载的文件系统

！网

使用 “ `!net` 显示系统网络”列表。

dbgcmd

```
6: kd> !net
@$net()
    [0x0]            : lo (127.0.0.1) [Type: net_device]
    [0x1]            : enP8p1s0f0np0 (192.168.3.19) [Type: net_device]
    [0x2]            : enP8p1s0f1np0 [Type: net_device]
```

`!net` 语法：

`!net [<arg>]`

没有 `[<arg>]` - 等效于“net”- 提供系统网络列表

`[<arg>]`：

`pid` - 为具有给定 pid 的进程命名空间提供净列表

`64-bit num` - 为地址提供的task_struct命名空间提供净列表

`<task struct [*]>` - 为给定task_struct的命名空间提供净列表

`<process object>` - 为进程表示的任务命名空间提供网络列表

！runq

`!runq`使用该命令显示有关运行队列中任务的信息。

dbgcmd

```
0: kd> !runq
@$runq()
    [0x0]            : CPU 0 run queue [current = 'bash' (17ca)]
        Cpu              : 0x0
        RunQueue         [Type: rq]
        CurrentTask      : bash [Type: task_struct]
        RTTasks
        CfsTasks
            [0x16b3]         : kworker/0:7 [Type: task_struct]
    [0x1]            : CPU 1 run queue [current = 'swapper/1' (0)]
        Cpu              : 0x1
        RunQueue         [Type: rq]
        CurrentTask      : swapper/1 [Type: task_struct]
        RTTasks
        CfsTasks
```

`!runq` 语法：

`!runq`

！runq 没有任何命令参数。

其他内核转储命令

`!dev` - 显示有关字符和块设备、分配、I/O 端口使用情况、I/O 内存使用情况的设备数据。

`!log` - 显示内核log_buf内容。

`!vm` - 显示虚拟内存使用情况的摘要。

`!timer` - 显示计时器队列条目。

dx 命令和 Linux 对象

dx 命令可用于调查 kdumps。 显示 Sessions 对象以查看可用的各种子对象。

dbgcmd

```
0: kd> dx -r3 Debugger.Sessions[0]
Debugger.Sessions[0]                 : Target Composition Target
    Processes
        [0x0]            : swapper/0 [Switch To]
            KernelObject     : swapper/0 [Type: task_struct]
            Name             : swapper/0
            Id               : 0x0
            Index            : 0x0
            Threads
            Modules
            Environment
            Direct3D
            Attributes
            Devices
            Io
            Memory
            GroupedStacks
...
```

cursession Kernel 对象包含可用于查看内核日志的 PrintKLog 对象。

dbgcmd

```
6: kd> dx @$cursession.Kernel.PrintKLog.Take(4)
@$cursession.Kernel.PrintKLog.Take(4)
    [0x0]            : [     0.000000] Booting Linux on physical CPU
0x0000000000 [0x410fd083]
    [0x1]            : [     0.000000] Linux version 4.19.90-microsoft-
standard (oe-user@oe-host) (gcc version 8.2.0 (GCC)) #1 SMP Fri Mar 27
14:25:24 UTC 2020..
    [0x2]            : [     0.000002] sched_clock: 64 bits at 125MHz,
resolution 8ns, wraps every 4398046511100ns
    [0x3]            : [     0.000003]         17.250901928 MSFT: kernel
boot start
```

此 dx 命令显示用于 `.Contains()` 在日志中查找特定字符串。

dbgcmd

```
6: kd> dx @$cursession.Kernel.PrintKLog.Where(le =>
le.ToLower().Contains("oops") || le.ToLower().Contains("crash"))
@$cursession.Kernel.PrintKLog.Where(le => le.ToLower().Contains("oops") ||
le.ToLower().Contains("crash"))
    [0x0]            : [     0.000493] crashkernel reserved:
0x00000000dc600000 - 0x00000000fc600000 (512 MB)
    [0x1]            : [     0.078790] Kernel command line:
console=ttyS0,115200n8 earlycon=uart8250,mmio32,0x68A10000 crashkernel=512M
```

```
enforcing=0 ipe.enforce=0
    [0x2]            : [    26.621228] sysrq: SysRq : Trigger a crash
    [0x3]            : [    26.621254] Internal error: Oops: 96000044 [#1]
SMP
    [0x4]            : [    26.656655] pc : sysrq_handle_crash+0x24/0x30
    [0x5]            : [    26.753494]  sysrq_handle_crash+0x24/0x30
    [0x6]            : [    26.801441] Starting crashdump kernel...8J»=.
```

用于 `.Reverse()` 显示记录的最后一个事件。

dbgcmd

```
2: kd> dx @$cursession.Kernel.PrintKLog.Reverse().Take(5).Reverse()
@$cursession.Kernel.PrintKLog.Reverse().Take(5).Reverse()
   [0x0]           : [3147944.378367]  kthread+0x118/0x2a4
   [0x1]           : [3147944.381876]  ret_from_fork+0x10/0x18
   [0x2]           : [3147944.385747] Code: 78002507 36000042 39000107
d65f03c0 (cb0803e4)
   [0x3]           : [3147944.392221] SMP: stopping secondary CPUs
   [0x4]           : [3147944.397539] Starting crashdump kernel...
```

有关将 LINQ 查询与 dx 命令配合使用的详细信息，请参阅 将 LINQ 与调试器对象配合使用。

Linux 诊断扩展 - ELFBinComposition.dll

ELFBinComposition.dll中提供了以下 Linux 转储文件诊断扩展。

转储文件命令

这些命令可用于大多数转储文件。

`!dumpdebug` - 显示正在调试的核心转储的诊断。 这包括各种其他命令的输出。

`!ntprstatus` - 显示正在调试的核心转储中的NT_PRSTATUS记录。

`!vmcoreinfo` - 显示正在调试的内核核心转储（KDUMP）中的 VMCOREINFO 表。

ELF 转储命令

这些命令只能用于 ELF 核心转储文件。

`!corephdrs` - 显示正在调试的核心转储的程序标头表。

`!ntprpsinfo` - 显示正在调试的核心转储中的NT_PRPSINFO数据。

`!ntfile` - 显示正在调试的核心转储中的NT_FILE数据。

`!ntauxv` - 显示正在调试的核心转储中的NT_AUXV数据。

内核故障转储文件命令

这些命令只能在内核核心转储 （KDUMP） 文件上使用。

`!kdumpdescs` - 在 KDUMP 中显示页面范围和页面描述符的列表。

`!kdumppagerange <n>` - 显示 KDUMP 中第 n 个页面分组的信息。

`!kdumppfn <pfn>` - 显示有关页面框架 `<pfn>` 及其数据在 KDUMP 中的位置的信息。

其他诊断命令

`!cppex` - 使用 libstdc++的内部结构和 DWARF 符号显示有关当前正在进行的（未捕获和“just”捕获）C++异常的信息。

`!cppfilt [-n] <mangled name>` - 将一个C++混乱的名称解开，就好像它通过 c++filt 工具运行一样。 有关 C==filtr 工具的详细信息，请参阅 c++filt（1） — Linux 手册页。

`!rustdemangle <mangled name>` - 解开 Rust 混乱的名称。 有关 rust 符号名称混乱的详细信息，请参阅 Rust 符号管理 - RFC 2603。

另请参阅

Linux 符号和源

Linux 实时远程进程调试

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

手动设置通过串行电缆进行的内核模式调试

项目 • 2024/03/05

Windows 调试工具支持通过零调制解调器电缆进行内核调试。 零调制解调器电缆是已经配置为在两个串行端口之间发送数据的串行电缆。 不要将零调制解调器电缆与标准串行电缆混淆。 标准串行电缆不相互连接串行端口。 有关如何连接零调制解调器电缆的信息，请参阅零调制解调器电缆连接。

运行调试器的计算机称为*主计算机*，正在调试的计算机称为*目标计算机*。

设置目标计算机

> **重要**
>
> 在使用 bcdedit 更改启动信息之前，可能需要在测试电脑上暂时挂起 Windows 安全功能，例如 BitLocker 和安全启动。 完成调试并禁用内核调试后，可以重新启用安全启动。

1. 在目标计算机上，以管理员身份打开命令提示符窗口，并输入以下命令，其中 *n* 是用于在目标计算机上调试的 COM 端口数，*速率*是用于调试的波特率：

   bcdedit /debug on

   bcdedit /dbgsettings serial debugport:*n*baudrate:*rate*

   注意 主计算机和目标计算机上的波特率必须相同。 建议的速率为 115200。

2. 重新启动目标计算机。

启动调试会话

将零调制解调器电缆连接到你选择用于在主计算机和目标计算机上进行调试的 COM 端口。

使用 WinDbg

在主计算机上打开 WinDbg。 在“文件”菜单上，选择“内核调试”。 在“内核调试”对话框中，打开“COM”选项卡。在“波特率”框中，输入已为调试选择的速率。 在“端口”框中，输

入 COM*n*，其中 *n* 是选择用于在主计算机上进行调试的 COM 端口号。 选择“确定”。

还可以通过在命令提示符窗口中输入以下命令来启动与 WinDbg 的会话；*n* 是用于在主计算机上进行调试的 COM 端口号，*速率*是用于调试的波特率：

windbg -k com:port=COM*n*,baud=*rate*

使用 KD

在主计算机上，打开命令提示符窗口并输入以下命令，其中 *n* 是用于在主计算机上进行调试的 COM 端口号，*速率*是用于调试的波特率：

kd -k com:port=COM*n*,baud=*rate*

使用环境变量

在主计算机上，可以使用环境变量来指定 COM 端口和波特率。 然后，每次启动调试会话时，不必指定端口和波特率。 要使用环境变量来指定 COM 端口和波特率，请打开命令提示符窗口并输入以下命令，其中 *n* 是用于在主计算机上进行调试的 COM 端口号，*速率*是用于调试的波特率：

- **set _NT_DEBUG_PORT=COM*n***
- **set _NT_DEBUG_BAUD_RATE=*rate***

要启动调试会话，请打开命令提示符窗口，然后输入以下命令之一：

- **kd**
- **windbg**

通过串行电缆进行调试的疑难解答提示

在主计算机和目标上指定正确的 COM 端口

确定用于在主计算机和目标计算机上进行调试的 COM 端口号。 例如，假设你已将零调制解调器电缆连接到主计算机上的 COM1，并在目标计算机上连接到 COM2。

在目标计算机上，以管理员身份打开命令提示符窗口，然后输入 **bcdedit /dbgsettings**。 如果在目标计算机上使用 COM2，则 **bcdedit** 的输出应显示 `debugport 2`。

在主计算机上启动调试器或设置环境变量时指定正确的 COM 端口。 如果在主计算机上使用 COM1，请使用下列方法之一来指定 COM 端口。

- 在 WinDbg 的“内核调试”对话框中，在“端口”框中输入 COM1。

- **windbg -k com:port=COM1, ...**
- **kd -k com:port=COM1, ...**
- **set _NT_DEBUG_PORT=COM1**

主计算机和目标上的波特率必须相同

用于通过串行电缆进行调试的波特率必须在主计算机和目标计算机上设置为相同值。 例如，假设选择了 115200 的波特率。

在目标计算机上，以管理员身份打开命令提示符窗口，然后输入 **bcdedit /dbgsettings**。 **bcdedit** 的输出应显示 `baudrate 115200`。

在主计算机上启动调试器或设置环境变量时指定正确的波特率。 使用以下方法之一指定 115200 的波特率。

- 在 WinDbg 的“内核调试”对话框中，在“波特率”框中输入 115200。
- **windbg -k ..., baud=115200**
- **kd -k ..., baud=115200**
- **set _NT_DEBUG_BAUD_RATE=115200**

零调制解调器电缆连接

下表显示了零调制解调器电缆的连接方式。

9 引脚连接器

展开表

连接器 1	连接器 2	信号
2	3	Tx - Rx
3	2	Rx - Tx
7	8	RTS - CTS
8	7	CTS - RTS
4	1+6	DTR - (CD+DSR)
1+6	4	(CD+DSR) - DTR
5	5	信号用接地

25 引脚连接器

展开表

连接器 1	连接器 2	信号
2	3	Tx - Rx
3	2	Rx - Tx
4	5	RTS - CTS
5	4	CTS - RTS
6	20	DSR - DTR
20	6	DTR - DSR
7	7	信号用接地

信号缩写

展开表

缩写	Signal
Tx	传输数据
Rx	接收数据
RTS	将请求发送到：
CTS	清除到发送
DTR	数据终端就绪
DSR	数据集就绪
CD	运营商检测

其他信息

有关 bcdedit 命令的完整文档，请参阅 BCDEdit 选项参考。

另请参阅

手动设置内核模式调试

Windows 10 中的网络内核调试支持的以太网 NIC

项目 • 2023/04/14

若要通过以太网网线进行内核调试，目标计算机必须具有受支持的网络接口卡 (NIC) 。

在内核调试期间，运行调试器的计算机称为 *主计算机*，正在调试的计算机称为 *目标计算机*。 有关详细信息，请参阅 自动设置 KDNET 网络内核调试。

若要通过网络电缆进行内核调试，目标计算机必须具有受支持的网络适配器。 当目标计算机运行 Windows 时，支持此处列出的网络适配器进行内核调试。

版本信息

本主题列出了以下版本的 Windows 支持的适配器

- Windows 10版本 20H2 内部版本 19042

对以前版本的 Windows 10 的适配器支持

有关以前版本的 Windows 10 中支持的 NIC 版本的信息，请参阅这些主题。

Windows 10 版本 2004 中的网络内核调试支持的以太网 NIC

Windows 10 版本 1909 中的网络内核调试支持的以太网 NIC

Windows 10 版本 1903 中的网络内核调试支持的以太网 NIC

Windows 10 版本 1809 中的网络内核调试支持的以太网 NIC

Windows 10 版本 1803 中的网络内核调试支持的以太网 NIC

Windows 10 版本 1709 中的网络内核调试支持的以太网 NIC

Windows 10 版本 1703 中的网络内核调试支持的以太网 NIC

使用 VerifiedNicList.xml确定 NIC 支持

若要确定任何特定版本的 Windows 支持哪组 NIC，请检查 `VerifiedNicList.xml` 随该特定版本的 Windows 附带的 WDK 安装的调试器目录中的文件。 对于 64 位 Windows，默认情况下，它将安装在此目录中：

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\VerifiedNicList.xml
```

需要检查 WDK 中随附的特定版本的VerifiedNicList.xml，因为向以前版本中不存在的 Windows 新版本添加了额外的硬件支持。 因此，必须为该特定版本检查 VerifiedNicLIst.xml文件。

查找供应商 ID 和设备 ID

在目标计算机上查找网络适配器的供应商 ID 和设备 ID。

- 在目标计算机上，打开设备管理器 (在命令提示符窗口中输入 **devmgmt**) 。
- 在“设备管理器”中，找到要用于调试的网络适配器。
- 选择并按住 (，或右键单击) 网络适配器节点，然后选择“**属性**”。
- 在“ **详细信息** ”选项卡的 “**属性**”下，选择“ **硬件 ID**”。

供应商和设备 ID 显示为 VEN_*VendorID* 和 DEV_*DeviceID*。 例如，如果看到 PCI\VEN_8086&DEV_104B，则供应商 ID 为 8086，设备 ID 为 104B。

供应商 ID 8086，Intel Corporation

对于供应商 ID 8086，支持以下设备 ID：

0001 0008 000C 000D 0438 043A 043C 0440 0470 0D4E 0D4F 0D4C 0D4D 0D53 0D55 1000 1001 1004 100 8 1009 100C 100D 100E 100F 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 101A 101D 101E 1026 1027 1028 1049 104A 104B 104C 104D 105E 105F 1060 1071 1075 1076 1077 1078 1079 107A 107B 107C 107D 107E 107F 108A 108B 108C 1096 1098 1099 109A 10A4 10A5 10A7 10A9 10B5 10B9 10BA 10BB10BC 10BD 10BF 10C0 10C2 10C3 10C4 10C5 10C6 10C7 10C8 10C9 10CB 10CC 10CD 10CE 10D3 10D5 10D6 10D9 10DA 10DB 10DD 10DE 10DF 10E1 10E5 10E6 10E7 10E8 10EA 10EB 10EC 10EF 10F0 10F1 10F4 10F5 10F6 10F7 10F8 10F9 10FB 10FC 11A9 1501 1502 1503 1507 150A 150B 150C 150D 150E 150F 1510 1511 1514 1516 1517 1518 151C 1521 1522 1523 1524 1525 1526 1527 1528 1529 152A 1533 1534 15351536 1537 1538 1539 153A 153B 1546 154A 154D 1557 1558 1559 155A 1560 1563 156F 1570 157B 157C 15A0 15A1 15A2 15A3 15AA 15AB 15AC 15AD 15AE 15B7 15B8 15B9 15BB 15BC 15BD 15BE 15D6 15D7 15D8 15DF 15E0 15E1 15E2 15E3 15F 4 15F9 15FA 15FB 15FC 17D0 1F40 1F41 1F45 1F63 1F72 211B 2159 294C 8976

供应商 ID 10EC，Realtek Semiconductor Corp.

对于供应商 ID 10EC，支持以下设备 ID：

2502 2600 3000 8125 8136 8137 8161 8166 8167 8168 8169 8225

供应商 ID 14E4，Broadcom

对于供应商 ID 14E4，支持以下设备 ID：

1600 1601 1614 1639 163A 163B 163C 163D 163E 1641 1642 1643 1644 1645 1646 1647 1648 164A 164C 164D 164E 164F 1650 1653 1654 1655 1656 1657 1659 165A 165B 165C 165D 165E 165F 1662 1663 16 65 1668 1669 166A 166B 166D 166E 1672 1673 1674 1676 1677 1678 1679 167A 167B 167C 167D 167F 1 68A 168D 168E 1680 1681 1682 1683 1684 1686 1687 1688 1690 1691 1692 1693 1694 1696 1698 1699169A 169B 169D 16A0 16A1 16A2 16A4 16A5 16A6 16A7 16A8 16AA 16AC 1 6AE 16B0 16B1 16B2 16B3 16B4 16B5 16B6 16B7 16C6 16C7 16C9 16CA 1 6CE 16CF 16D0 16D1 16D2 16D5 16D6 16D7 16D8 16D9 16DD 16DD 16DF 16E0 16E2 16E3 16E4 16E9 16F0 16F1 16F7 16FD 16FE 16FF 170D 170E 170F D802

供应商 ID 1969，Atheros Communications

对于供应商 ID 1969，支持以下设备 ID：

1062 1063 1073 1083 1090 1091 10A0 10A1 10B0 10B1 10C0 10C1 10D0 10D1 10E0 10E1 10F0 10F1 2060 2062 E091 E0A1 E0B1 E0C1 E0D1 E0E1 E0F1

供应商 ID 19A2、ServerEngines (Emulex)

对于供应商 ID 19A2，支持以下设备 ID：

0211 0215 0221 0700 0710

供应商 ID 10DF，Emulex Corporation

对于供应商 ID 10DF，支持以下设备 ID：

0720 E220

供应商 ID 15B3，Mellanox 技术

对于供应商 ID 15B3，支持以下设备 ID：

1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 100A 100B 100C 100D 100E 100F 1010 1013 1015 1017 1019 101B 101B 101D 101F 1021 10 23 1025 1027 1029 102B 102F 6340 6341 634A 634B 6354 6368 6369 6 372 6732 6733 673C 673D 6746 6750 6751 675A 6764 6765 676E 6778

供应商 ID 1137，Cisco Systems Inc

对于供应商 ID 1137，支持以下设备 ID：

0043

XML 支持的 NIC 列表

这与上面以 XML 格式显示的信息相同。

XML


```xml
<?xml version="1.0" encoding="utf-8"?>
<SupportedNetworkInterfaceCards>
  <NIC>
    <manufacturer>8086</manufacturer>
      <deviceid>1000</deviceid>
      <deviceid>1001</deviceid>
      <deviceid>1004</deviceid>
      <deviceid>1008</deviceid>
      <deviceid>1009</deviceid>
      <deviceid>100C</deviceid>
      <deviceid>100D</deviceid>
      <deviceid>100E</deviceid>
      <deviceid>1015</deviceid>
      <deviceid>1016</deviceid>
      <deviceid>1017</deviceid>
      <deviceid>101E</deviceid>
      <deviceid>100F</deviceid>
      <deviceid>1011</deviceid>
      <deviceid>1026</deviceid>
      <deviceid>1027</deviceid>
      <deviceid>1028</deviceid>
      <deviceid>1010</deviceid>
      <deviceid>1012</deviceid>
      <deviceid>101D</deviceid>
      <deviceid>1079</deviceid>
      <deviceid>107A</deviceid>
      <deviceid>107B</deviceid>
      <deviceid>108A</deviceid>
      <deviceid>1099</deviceid>
      <deviceid>10B5</deviceid>
      <deviceid>1013</deviceid>
      <deviceid>1018</deviceid>
      <deviceid>1014</deviceid>
      <deviceid>1078</deviceid>
      <deviceid>1076</deviceid>
      <deviceid>107C</deviceid>
      <deviceid>1077</deviceid>
      <deviceid>1019</deviceid>
      <deviceid>101A</deviceid>
```

```
        <deviceid>1075</deviceid>
        <deviceid>105E</deviceid>
        <deviceid>105F</deviceid>
        <deviceid>1060</deviceid>
        <deviceid>10D9</deviceid>
        <deviceid>10DA</deviceid>
        <deviceid>10A4</deviceid>
        <deviceid>10D5</deviceid>
        <deviceid>10A5</deviceid>
        <deviceid>10BC</deviceid>
        <deviceid>107D</deviceid>
        <deviceid>107E</deviceid>
        <deviceid>107F</deviceid>
        <deviceid>10B9</deviceid>
        <deviceid>108B</deviceid>
        <deviceid>108C</deviceid>
        <deviceid>109A</deviceid>
        <deviceid>10D3</deviceid>
        <deviceid>10F6</deviceid>
        <deviceid>150C</deviceid>
        <deviceid>1096</deviceid>
        <deviceid>1098</deviceid>
        <deviceid>10BA</deviceid>
        <deviceid>10BB</deviceid>
        <deviceid>1501</deviceid>
        <deviceid>1049</deviceid>
        <deviceid>104A</deviceid>
        <deviceid>104B</deviceid>
        <deviceid>104C</deviceid>
        <deviceid>10C4</deviceid>
        <deviceid>10C5</deviceid>
        <deviceid>104D</deviceid>
        <deviceid>10BF</deviceid>
        <deviceid>10F5</deviceid>
        <deviceid>10CB</deviceid>
        <deviceid>10BD</deviceid>
        <deviceid>10E5</deviceid>
        <deviceid>294C</deviceid>
        <deviceid>10C0</deviceid>
        <deviceid>10C3</deviceid>
        <deviceid>10C2</deviceid>
        <deviceid>10CC</deviceid>
        <deviceid>10CD</deviceid>
        <deviceid>10CE</deviceid>
        <deviceid>10DE</deviceid>
        <deviceid>10DF</deviceid>
        <deviceid>1525</deviceid>
        <deviceid>10EA</deviceid>
        <deviceid>10EB</deviceid>
        <deviceid>10EF</deviceid>
        <deviceid>10F0</deviceid>
        <deviceid>1502</deviceid>
        <deviceid>1503</deviceid>
        <deviceid>153A</deviceid>
        <deviceid>153B</deviceid>
```

```
        <deviceid>155A</deviceid>
        <deviceid>1559</deviceid>
        <deviceid>15A0</deviceid>
        <deviceid>15A1</deviceid>
        <deviceid>15A2</deviceid>
        <deviceid>15A3</deviceid>
        <deviceid>156F</deviceid>
        <deviceid>1570</deviceid>
        <deviceid>15B7</deviceid>
        <deviceid>15B8</deviceid>
        <deviceid>15B9</deviceid>
        <deviceid>15D7</deviceid>
        <deviceid>15D8</deviceid>
        <deviceid>15E3</deviceid>
        <deviceid>15D6</deviceid>
        <deviceid>15BD</deviceid>
        <deviceid>15BE</deviceid>
        <deviceid>15BB</deviceid>
        <deviceid>15BC</deviceid>
        <deviceid>15DF</deviceid>
        <deviceid>15E0</deviceid>
        <deviceid>15E1</deviceid>
        <deviceid>15E2</deviceid>
        <deviceid>10C9</deviceid>
        <deviceid>10E6</deviceid>
        <deviceid>10E7</deviceid>
        <deviceid>10E8</deviceid>
        <deviceid>1526</deviceid>
        <deviceid>150A</deviceid>
        <deviceid>1518</deviceid>
        <deviceid>150D</deviceid>
        <deviceid>10A7</deviceid>
        <deviceid>10A9</deviceid>
        <deviceid>10D6</deviceid>
        <deviceid>150E</deviceid>
        <deviceid>150F</deviceid>
        <deviceid>1510</deviceid>
        <deviceid>1511</deviceid>
        <deviceid>1516</deviceid>
        <deviceid>1527</deviceid>
        <deviceid>1521</deviceid>
        <deviceid>1522</deviceid>
        <deviceid>1523</deviceid>
        <deviceid>1524</deviceid>
        <deviceid>1546</deviceid>
        <deviceid>1533</deviceid>
        <deviceid>1534</deviceid>
        <deviceid>1535</deviceid>
        <deviceid>1536</deviceid>
        <deviceid>1537</deviceid>
        <deviceid>1538</deviceid>
        <deviceid>157B</deviceid>
        <deviceid>157C</deviceid>
        <deviceid>1539</deviceid>
        <deviceid>1F40</deviceid>
```

```
        <deviceid>1F41</deviceid>
        <deviceid>1F45</deviceid>
        <deviceid>0438</deviceid>
        <deviceid>043A</deviceid>
        <deviceid>043C</deviceid>
        <deviceid>0440</deviceid>
        <deviceid>10C6</deviceid>
        <deviceid>10C7</deviceid>
        <deviceid>10C8</deviceid>
        <deviceid>150B</deviceid>
        <deviceid>10DB</deviceid>
        <deviceid>10DD</deviceid>
        <deviceid>10EC</deviceid>
        <deviceid>10F1</deviceid>
        <deviceid>10E1</deviceid>
        <deviceid>10F4</deviceid>
        <deviceid>10F7</deviceid>
        <deviceid>1514</deviceid>
        <deviceid>1517</deviceid>
        <deviceid>10F8</deviceid>
        <deviceid>000C</deviceid>
        <deviceid>1F63</deviceid>
        <deviceid>10F9</deviceid>
        <deviceid>10FB</deviceid>
        <deviceid>11A9</deviceid>
        <deviceid>1071</deviceid>
        <deviceid>1F72</deviceid>
        <deviceid>17D0</deviceid>
        <deviceid>0470</deviceid>
        <deviceid>211B</deviceid>
        <deviceid>8976</deviceid>
        <deviceid>2159</deviceid>
        <deviceid>000D</deviceid>
        <deviceid>0008</deviceid>
        <deviceid>152A</deviceid>
        <deviceid>1529</deviceid>
        <deviceid>1507</deviceid>
        <deviceid>154D</deviceid>
        <deviceid>154A</deviceid>
        <deviceid>1558</deviceid>
        <deviceid>1557</deviceid>
        <deviceid>0001</deviceid>
        <deviceid>10FC</deviceid>
        <deviceid>151C</deviceid>
        <deviceid>1528</deviceid>
        <deviceid>1560</deviceid>
        <deviceid>1563</deviceid>
        <deviceid>ABCD</deviceid>
        <deviceid>15AA</deviceid>
        <deviceid>15AB</deviceid>
        <deviceid>15AC</deviceid>
        <deviceid>15AD</deviceid>
        <deviceid>15AE</deviceid>
        <deviceid>0D4E</deviceid>
        <deviceid>0D4F</deviceid>
```

```
      <deviceid>0D4C</deviceid>
      <deviceid>0D4D</deviceid>
      <deviceid>0D53</deviceid>
      <deviceid>0D55</deviceid>
      <deviceid>15FB</deviceid>
      <deviceid>15FC</deviceid>
      <deviceid>15F9</deviceid>
      <deviceid>15FA</deviceid>
      <deviceid>15F4</deviceid>
      <deviceid>15F5</deviceid>
      <deviceid>1A1E</deviceid>
      <deviceid>1A1F</deviceid>
      <deviceid>1A1C</deviceid>
      <deviceid>1A1D</deviceid>
      <deviceid>550A</deviceid>
      <deviceid>550B</deviceid>
      <deviceid>550C</deviceid>
      <deviceid>550D</deviceid>
      <deviceid>1572</deviceid>
      <deviceid>1574</deviceid>
      <deviceid>1580</deviceid>
      <deviceid>1581</deviceid>
      <deviceid>1583</deviceid>
      <deviceid>1584</deviceid>
      <deviceid>1585</deviceid>
      <deviceid>1586</deviceid>
      <deviceid>1587</deviceid>
      <deviceid>1588</deviceid>
      <deviceid>1589</deviceid>
      <deviceid>158A</deviceid>
      <deviceid>158B</deviceid>
      <deviceid>37D0</deviceid>
      <deviceid>37CC</deviceid>
      <deviceid>37D2</deviceid>
      <deviceid>37CE</deviceid>
      <deviceid>37CF</deviceid>
      <deviceid>37D0</deviceid>
      <deviceid>37D1</deviceid>
      <deviceid>37D2</deviceid>
      <deviceid>37D3</deviceid>
      <deviceid>37D4</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10EC</manufacturer>
      <deviceid>8136</deviceid>
      <deviceid>8137</deviceid>
      <deviceid>8168</deviceid>
      <deviceid>8167</deviceid>
      <deviceid>8169</deviceid>
      <deviceid>8166</deviceid>
      <deviceid>8161</deviceid>
      <deviceid>8125</deviceid>
      <deviceid>8225</deviceid>
      <deviceid>2502</deviceid>
```

```
      <deviceid>2600</deviceid>
      <deviceid>3000</deviceid>
  </NIC>

  <NIC>
    <manufacturer>14E4</manufacturer>
      <deviceid>1644</deviceid>
      <deviceid>1645</deviceid>
      <deviceid>1646</deviceid>
      <deviceid>16A6</deviceid>
      <deviceid>16C6</deviceid>
      <deviceid>1647</deviceid>
      <deviceid>16A7</deviceid>
      <deviceid>16C7</deviceid>
      <deviceid>164D</deviceid>
      <deviceid>1648</deviceid>
      <deviceid>16A8</deviceid>
      <deviceid>1653</deviceid>
      <deviceid>166E</deviceid>
      <deviceid>1654</deviceid>
      <deviceid>165D</deviceid>
      <deviceid>165E</deviceid>
      <deviceid>166D</deviceid>
      <deviceid>170D</deviceid>
      <deviceid>170E</deviceid>
      <deviceid>1696</deviceid>
      <deviceid>1676</deviceid>
      <deviceid>1677</deviceid>
      <deviceid>1659</deviceid>
      <deviceid>167C</deviceid>
      <deviceid>167D</deviceid>
      <deviceid>169D</deviceid>
      <deviceid>16F7</deviceid>
      <deviceid>16FD</deviceid>
      <deviceid>16FE</deviceid>
      <deviceid>16DD</deviceid>
      <deviceid>1668</deviceid>
      <deviceid>1669</deviceid>
      <deviceid>1678</deviceid>
      <deviceid>1679</deviceid>
      <deviceid>1600</deviceid>
      <deviceid>1601</deviceid>
      <deviceid>166A</deviceid>
      <deviceid>166B</deviceid>
      <deviceid>16FF</deviceid>
      <deviceid>170F</deviceid>
      <deviceid>169B</deviceid>
      <deviceid>1693</deviceid>
      <deviceid>167F</deviceid>
      <deviceid>169A</deviceid>
      <deviceid>1698</deviceid>
      <deviceid>1692</deviceid>
      <deviceid>1694</deviceid>
      <deviceid>1690</deviceid>
      <deviceid>1691</deviceid>
```

```
        <deviceid>1699</deviceid>
        <deviceid>16A0</deviceid>
        <deviceid>167B</deviceid>
        <deviceid>1673</deviceid>
        <deviceid>165A</deviceid>
        <deviceid>1674</deviceid>
        <deviceid>1681</deviceid>
        <deviceid>1680</deviceid>
        <deviceid>1688</deviceid>
        <deviceid>167A</deviceid>
        <deviceid>1672</deviceid>
        <deviceid>1684</deviceid>
        <deviceid>165B</deviceid>
        <deviceid>16B1</deviceid>
        <deviceid>16B5</deviceid>
        <deviceid>16B0</deviceid>
        <deviceid>16B4</deviceid>
        <deviceid>16B2</deviceid>
        <deviceid>16B6</deviceid>
        <deviceid>1682</deviceid>
        <deviceid>1686</deviceid>
        <deviceid>16B3</deviceid>
        <deviceid>16B7</deviceid>
        <deviceid>1655</deviceid>
        <deviceid>1665</deviceid>
        <deviceid>1656</deviceid>
        <deviceid>1657</deviceid>
        <deviceid>165F</deviceid>
        <deviceid>165C</deviceid>
        <deviceid>1683</deviceid>
        <deviceid>1641</deviceid>
        <deviceid>1642</deviceid>
        <deviceid>1643</deviceid>
        <deviceid>1687</deviceid>
        <deviceid>164A</deviceid>
        <deviceid>16AA</deviceid>
        <deviceid>164C</deviceid>
        <deviceid>16AC</deviceid>
        <deviceid>1639</deviceid>
        <deviceid>163A</deviceid>
        <deviceid>163B</deviceid>
        <deviceid>163C</deviceid>
        <deviceid>164E</deviceid>
        <deviceid>164F</deviceid>
        <deviceid>1650</deviceid>
        <deviceid>1662</deviceid>
        <deviceid>1663</deviceid>
        <deviceid>168A</deviceid>
        <deviceid>168D</deviceid>
        <deviceid>16A1</deviceid>
        <deviceid>16A2</deviceid>
        <deviceid>168E</deviceid>
        <deviceid>163D</deviceid>
        <deviceid>16A5</deviceid>
        <deviceid>16A4</deviceid>
```

```
        <deviceid>16AE</deviceid>
        <deviceid>163E</deviceid>
        <deviceid>16C0</deviceid>
        <deviceid>16C9</deviceid>
        <deviceid>16CA</deviceid>
        <deviceid>16CC</deviceid>
        <deviceid>16CD</deviceid>
        <deviceid>16CE</deviceid>
        <deviceid>16CF</deviceid>
        <deviceid>16D0</deviceid>
        <deviceid>16D1</deviceid>
        <deviceid>16D2</deviceid>
        <deviceid>16D4</deviceid>
        <deviceid>16D5</deviceid>
        <deviceid>16D6</deviceid>
        <deviceid>16D7</deviceid>
        <deviceid>16D8</deviceid>
        <deviceid>16D9</deviceid>
        <deviceid>16DA</deviceid>
        <deviceid>16DB</deviceid>
        <deviceid>16DC</deviceid>
        <deviceid>16DE</deviceid>
        <deviceid>16DF</deviceid>
        <deviceid>16E0</deviceid>
        <deviceid>16E2</deviceid>
        <deviceid>16E3</deviceid>
        <deviceid>16E4</deviceid>
        <deviceid>16E7</deviceid>
        <deviceid>16E8</deviceid>
        <deviceid>16E9</deviceid>
        <deviceid>16EA</deviceid>
        <deviceid>16EB</deviceid>
        <deviceid>16EC</deviceid>
        <deviceid>16ED</deviceid>
        <deviceid>16EE</deviceid>
        <deviceid>16EF</deviceid>
        <deviceid>16F0</deviceid>
        <deviceid>16F1</deviceid>
        <deviceid>1614</deviceid>
        <deviceid>D802</deviceid>
        <deviceid>1604</deviceid>
        <deviceid>1605</deviceid>
        <deviceid>1606</deviceid>
        <deviceid>1607</deviceid>
        <deviceid>1608</deviceid>
        <deviceid>1609</deviceid>
        <deviceid>D804</deviceid>
        <deviceid>D812</deviceid>
        <deviceid>D814</deviceid>
        <deviceid>D818</deviceid>
        <deviceid>D82A</deviceid>
        <deviceid>D82B</deviceid>
        <deviceid>D82C</deviceid>
        <deviceid>D82D</deviceid>
        <deviceid>D82E</deviceid>
```

```
      <deviceid>D82F</deviceid>
      <deviceid>1902</deviceid>
      <deviceid>1903</deviceid>
      <deviceid>1906</deviceid>
      <deviceid>1907</deviceid>
      <deviceid>190A</deviceid>
      <deviceid>190B</deviceid>
      <deviceid>1799</deviceid>
      <deviceid>1041</deviceid>
      <deviceid>1042</deviceid>
      <deviceid>1043</deviceid>
      <deviceid>1750</deviceid>
      <deviceid>1751</deviceid>
      <deviceid>1752</deviceid>
      <deviceid>1800</deviceid>
      <deviceid>1801</deviceid>
      <deviceid>1802</deviceid>
      <deviceid>1803</deviceid>
      <deviceid>1804</deviceid>
      <deviceid>1805</deviceid>
      <deviceid>1806</deviceid>
      <deviceid>1807</deviceid>
      <deviceid>1808</deviceid>
      <deviceid>1809</deviceid>
      <deviceid>16C1</deviceid>
      <deviceid>16C5</deviceid>
      <deviceid>16BD</deviceid>
  </NIC>

  <NIC>
    <manufacturer>1969</manufacturer>
      <deviceid>1062</deviceid>
      <deviceid>1063</deviceid>
      <deviceid>2060</deviceid>
      <deviceid>2062</deviceid>
      <deviceid>1073</deviceid>
      <deviceid>1083</deviceid>
      <deviceid>1090</deviceid>
      <deviceid>1091</deviceid>
      <deviceid>E091</deviceid>
      <deviceid>10A0</deviceid>
      <deviceid>10A1</deviceid>
      <deviceid>E0A1</deviceid>
      <deviceid>10B0</deviceid>
      <deviceid>10B1</deviceid>
      <deviceid>E0B1</deviceid>
      <deviceid>10C0</deviceid>
      <deviceid>10C1</deviceid>
      <deviceid>E0C1</deviceid>
      <deviceid>10D0</deviceid>
      <deviceid>10D1</deviceid>
      <deviceid>E0D1</deviceid>
      <deviceid>10E0</deviceid>
      <deviceid>10E1</deviceid>
      <deviceid>E0E1</deviceid>
```

```
      <deviceid>10F0</deviceid>
      <deviceid>10F1</deviceid>
      <deviceid>E0F1</deviceid>
  </NIC>

  <NIC>
    <manufacturer>19A2</manufacturer>
      <deviceid>0211</deviceid>
      <deviceid>0215</deviceid>
      <deviceid>0221</deviceid>
      <deviceid>0700</deviceid>
      <deviceid>0710</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10DF</manufacturer>
      <deviceid>0720</deviceid>
      <deviceid>E220</deviceid>
  </NIC>

  <NIC>
    <manufacturer>15B3</manufacturer>
      <deviceid>1000</deviceid>
      <deviceid>1001</deviceid>
      <deviceid>1002</deviceid>
      <deviceid>1003</deviceid>
      <deviceid>1004</deviceid>
      <deviceid>1005</deviceid>
      <deviceid>1006</deviceid>
      <deviceid>1007</deviceid>
      <deviceid>1008</deviceid>
      <deviceid>1009</deviceid>
      <deviceid>100A</deviceid>
      <deviceid>100B</deviceid>
      <deviceid>100C</deviceid>
      <deviceid>100D</deviceid>
      <deviceid>100E</deviceid>
      <deviceid>100F</deviceid>
      <deviceid>1010</deviceid>
      <deviceid>6340</deviceid>
      <deviceid>6341</deviceid>
      <deviceid>634A</deviceid>
      <deviceid>634B</deviceid>
      <deviceid>6354</deviceid>
      <deviceid>6368</deviceid>
      <deviceid>6369</deviceid>
      <deviceid>6372</deviceid>
      <deviceid>6732</deviceid>
      <deviceid>6733</deviceid>
      <deviceid>673C</deviceid>
      <deviceid>673D</deviceid>
      <deviceid>6746</deviceid>
      <deviceid>6750</deviceid>
      <deviceid>6751</deviceid>
      <deviceid>675A</deviceid>
```

```
        <deviceid>6764</deviceid>
        <deviceid>6765</deviceid>
        <deviceid>676E</deviceid>
        <deviceid>6778</deviceid>
        <deviceid>1013</deviceid>
        <deviceid>1015</deviceid>
        <deviceid>1017</deviceid>
        <deviceid>1019</deviceid>
        <deviceid>101B</deviceid>
        <deviceid>101D</deviceid>
        <deviceid>101F</deviceid>
        <deviceid>1021</deviceid>
        <deviceid>1023</deviceid>
        <deviceid>1025</deviceid>
        <deviceid>1027</deviceid>
        <deviceid>1029</deviceid>
        <deviceid>102B</deviceid>
        <deviceid>102F</deviceid>
    </NIC>

    <NIC>
      <manufacturer>1137</manufacturer>
        <deviceid>0043</deviceid>
    </NIC>

</SupportedNetworkInterfaceCards>
```

另请参阅

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

Windows 8 中的网络内核调试支持的以太网 NIC

Windows 8.1 中的网络内核调试支持的以太网 NIC

Windows 11 23H2 中网络内核调试支持的以太网 NIC

项目 • 2024/10/17

若要通过以太网网络电缆执行内核调试，目标计算机必须具有受支持的网络接口卡（NIC）。

在内核调试期间，运行调试器的计算机称为 *主计算机*，正在调试的计算机称为 *目标计算机*。 有关详细信息，请参阅设置 KDNET 网络内核自动调试。

若要通过网络电缆进行内核调试，目标计算机必须具有受支持的网络适配器。 当目标计算机运行 Windows 时，内核调试支持此处列出的网络适配器。

版本信息

本主题列出了以下版本的 Windows 支持的适配器

- Windows 11,23H2 内部版本 22631

有关 Windows 11 版本的信息，请参阅 Windows 11 版本信息。

使用 VerifiedNicList.xml 确定 NIC 支持

若要确定任何特定版本的 Windows 支持哪个 NIC 集，请检查 `VerifiedNicList.xml` 随该特定版本的 Windows 附带的 WDK 或 SDK 安装的调试器目录中的文件。 对于 64 位 Windows，默认情况下，它将安装在此目录中：

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\VerifiedNicList.xml
```

需要检查特定版本的VerifiedNicList.xml，因为将其他硬件支持添加到以前版本中不存在的 Windows 新版本中。 因此，必须检查该特定版本的VerifiedNicLIst.xml文件。

查找供应商 ID 和设备 ID

在目标计算机上查找网络适配器的供应商 ID 和设备 ID。

- 在目标计算机上，打开设备管理器（在命令提示符窗口中输入 **devmgmt**）。
- 在设备管理器中，找到要用于调试的网络适配器。
- 选择并按住（或右键单击）网络适配器节点，然后选择“ **属性**”。
- 在“**详细信息**”选项卡的“属性”**下**，选择“**硬件 ID**”。

供应商和设备 ID 显示为 VEN_*VendorID* 和 DEV_*DeviceID*。 例如，如果看到 PCI\VEN_8086&DEV_104B，则供应商 ID 为 8086，设备 ID 为 104B。

供应商 ID 8086，Intel Corporation

对于供应商 ID 8086，支持以下设备 ID：

0001 0008 000C 000D 0438 043A 043C 0440 0470 0D4E 0D4F 0D4C 0D4D 0D53 0D55 1000 1004 1004 100 8 1009 100C 100D 100E 100F 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 101A 101D 101D 101E 1026 1027 1028 1049 104A 104B 104C 104D 105E 105F 1060 1071 1075 1076 1077 1078 1079 1079 107A 107B 107B 107B 107C 107D 107E 107F 108A 108B 108C 1096 1098 1099 109A 10A4 10A5 10A7 10A9 10B5 10B5 10B9 10BA 10BB10BC 10BD 10BF 10C0 10C2 10C3 10C4 10C5 10C6 10C7 10C8 10C9 10CB 10CC 10CD 10CE 10D3 10D5 10D6 10D9 10DA 10DB 10DD 10DE 10DF 10E1 10E5 10E6 10E7 10E8 10EA 10EB 10EC 10EF 10F0 10F1 10F4 10F5 10F6 10F7 10F8 10F9 10FB 10FC 11A9 1501 1502 1503 1507 150A 150B 150C 150D 150E 150F 150F 1510 1511 15111514 1516 1517 1518 151C 1521 1522 1523 1524 1525 1526 1527 1528 1529 152A 1533 1534 15351536 1537 1538 1539 153A 153B 1546 154A 154D 1557 1558 1559 155A 1560 1563 156F 1570 157B 1 57C 15A0 15A1 15A2 15A3 15AA 15AB 15AC 15AD 15AE 15B7 15B8 15B9 15BB 15BC 15BD 15BE 15D6 15D6 15D7 15D8 15DF 15E0 15E1 15E2 15E3 15F4 15F9 15FA 15FB 15FC 17D0 1F40 1F41 1F45 1F63 1F72 211B 2159 294C 8976 1572 1574 1580 1581 1583 1584 1585 1586 1587 1588 1589 15F5 1A1E 1A1F 1A1C 1A1D 158A158B 37D0 37CC 37D2 37CE 37CF 37D0 37D1 37D2 37D3 37D4 550A 550B 550C 550D 550E 550F 5510 5511 0DC7 0DC8 0DC5 0DC50DC6 57A0 57A1 57B3 57B4 57B5 57B6 57B7 57B8 15F2 15F3 0D9F 3100 3101 3102 5502 125B 125C 125D 125E 5503

供应商 ID 10EC，Realtek 半导体公司。

对于供应商 ID 10EC，支持以下设备 ID：

2502 2600 3000 8125 8136 8137 8161 8166 8167 8168 8169 8225 8126 8162 5000

供应商 ID 14E4，Broadcom

对于供应商 ID 14E4，支持以下设备 ID：

1600 1601 1614 1639 163A 163B 163C 163D 163E 1641 1642 1644 1645 1646 1647 1648 1648 164A 164C 164D 164E 164F 1650 1653 1654 1655 1656 1657 1659 165A 165B 165C 165D 165E 165F 165F 1662 1663 1665 1668 1669 166A 166B 166D 166E 1672 1673 1674 1676 1677 1678 1679 167A 167B 167C 167D 167F 168A 168D 168E 1680 1681 1682 1683 1684 1686 1687 1688 1690 1691 1692 1693 1694 1696 1698 1699169A 169B

169D 16A0 16A1 16A2 16A4 16A5 16A6 16A7 16A8 16AA 16AC 16AE 16B0 16B1 16B2 16B3 16B4 16B5 16B6 16B7 16C6 16C7 16C9 16CA 16CE 16CF 16D0 16D1 16D2 16D5 16D6 16D7 16D8 16D9 16D9 16DD 16DF 16E0 16E2 16E3 16E4 16E9 16F0 16F1 16F7 16FD 16FE 16FF 170D 170E 170F D802 1041 1042 1043 160 4 1605 1606 1607 1608 1609 16C1 16C5 16BD 1750 1751 1752 1799 1800 1801 1802 1803 1804 18051806 1807 1808 1809 1902 1903 1906 1907 190A 190B D804 D812 D814 D818 D82A D82B D82C D82D D82E D82F

供应商 ID 1969，Atheros Communications

对于供应商 ID 1969，支持以下设备 ID：

1062 1063 1073 1083 1090 1091 10A0 10A1 10B0 10B1 10C0 10C0 10C1 10D0 10D1 10E0 10E1 10F0 10F1 2060 2062 E091 E0A1 E0B1 E0C1 E0D1 E0E1 E0E1 E0F1

供应商 ID 19A2，ServerEngines （Emulex）

对于供应商 ID 19A2，支持以下设备 ID：

0211 0215 0221 0700 0710

供应商 ID 10DF，Emulex Corporation

对于供应商 ID 10DF，支持以下设备 ID：

0720 E220

供应商 ID 15B3，Mellanox 技术

对于供应商 ID 15B3，支持以下设备 ID：

1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 100A 100B 100C 100D 100E 100F 1010 1013 1015 1017 1019 101B 101D 101F 1021 10 23 1025 1027 1029 102B 102F 6340 6341 634A 634B 6354 6368 6369 6372 6732 6733 673C 673D 6746 6750 6751 675A 6764 6765 676E 6778

供应商 ID 1137，Cisco Systems Inc

对于供应商 ID 1137，支持以下设备 ID：

0043

供应商 ID 1414，Microsoft Corporation

对于供应商 ID 1414，支持以下设备 ID：

00B9

供应商 ID 15AD， VMware Inc

Windows 23H2 中的新增功能

对于供应商 ID 15AD，支持以下设备 ID：

07B0

XML 支持的 NIC 列表

这是上面以 XML 格式显示的相同信息。

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<SupportedNetworkInterfaceCards>
  <NIC>
    <manufacturer>8086</manufacturer>
      <deviceid build="9200">1000</deviceid>
      <deviceid build="9200">1001</deviceid>
      <deviceid build="9200">1004</deviceid>
      <deviceid build="9200">1008</deviceid>
      <deviceid build="9200">1009</deviceid>
      <deviceid build="9200">100C</deviceid>
      <deviceid build="9200">100D</deviceid>
      <deviceid build="9200">100E</deviceid>
      <deviceid build="9200">1015</deviceid>
      <deviceid build="9200">1016</deviceid>
      <deviceid build="9200">1017</deviceid>
      <deviceid build="9200">101E</deviceid>
      <deviceid build="9200">100F</deviceid>
      <deviceid build="9200">1011</deviceid>
      <deviceid build="9200">1026</deviceid>
      <deviceid build="9200">1027</deviceid>
      <deviceid build="9200">1028</deviceid>
      <deviceid build="9200">1010</deviceid>
      <deviceid build="9200">1012</deviceid>
      <deviceid build="9200">101D</deviceid>
      <deviceid build="9200">1079</deviceid>
      <deviceid build="9200">107A</deviceid>
      <deviceid build="9200">107B</deviceid>
      <deviceid build="9200">108A</deviceid>
      <deviceid build="9200">1099</deviceid>
      <deviceid build="9200">10B5</deviceid>
```

```
        <deviceid build="9200">1013</deviceid>
        <deviceid build="9200">1018</deviceid>
        <deviceid build="9200">1014</deviceid>
        <deviceid build="9200">1078</deviceid>
        <deviceid build="9200">1076</deviceid>
        <deviceid build="9200">107C</deviceid>
        <deviceid build="9200">1077</deviceid>
        <deviceid build="9200">1019</deviceid>
        <deviceid build="9200">101A</deviceid>
        <deviceid build="9200">1075</deviceid>
        <deviceid build="9200">105E</deviceid>
        <deviceid build="9200">105F</deviceid>
        <deviceid build="9200">1060</deviceid>
        <deviceid build="9200">10D9</deviceid>
        <deviceid build="9200">10DA</deviceid>
        <deviceid build="9200">10A4</deviceid>
        <deviceid build="9200">10D5</deviceid>
        <deviceid build="9200">10A5</deviceid>
        <deviceid build="9600">10BC</deviceid>
        <deviceid build="9200">107D</deviceid>
        <deviceid build="9200">107E</deviceid>
        <deviceid build="9200">107F</deviceid>
        <deviceid build="9600">10B9</deviceid>
        <deviceid build="9200">108B</deviceid>
        <deviceid build="9200">108C</deviceid>
        <deviceid build="9200">109A</deviceid>
        <deviceid build="9600">10D3</deviceid>
        <deviceid build="9200">10F6</deviceid>
        <deviceid build="9200">150C</deviceid>
        <deviceid build="9200">1096</deviceid>
        <deviceid build="9200">1098</deviceid>
        <deviceid build="9600">10BA</deviceid>
        <deviceid build="9600">10BB</deviceid>
        <deviceid build="9200">1501</deviceid>
        <deviceid build="9200">1049</deviceid>
        <deviceid build="9200">104A</deviceid>
        <deviceid build="9200">104B</deviceid>
        <deviceid build="9200">104C</deviceid>
        <deviceid build="9600">10C4</deviceid>
        <deviceid build="9600">10C5</deviceid>
        <deviceid build="9200">104D</deviceid>
        <deviceid build="9600">10BF</deviceid>
        <deviceid build="9200">10F5</deviceid>
        <deviceid build="9600">10CB</deviceid>
        <deviceid build="9600">10BD</deviceid>
        <deviceid build="9200">10E5</deviceid>
        <deviceid build="9200">294C</deviceid>
        <deviceid build="9600">10C0</deviceid>
        <deviceid build="9600">10C3</deviceid>
        <deviceid build="9600">10C2</deviceid>
        <deviceid build="9600">10CC</deviceid>
        <deviceid build="9600">10CD</deviceid>
        <deviceid build="9600">10CE</deviceid>
        <deviceid build="9600">10DE</deviceid>
        <deviceid build="9600">10DF</deviceid>
```

```
        <deviceid build="9600">1525</deviceid>
        <deviceid build="9200">10EA</deviceid>
        <deviceid build="9200">10EB</deviceid>
        <deviceid build="9200">10EF</deviceid>
        <deviceid build="9200">10F0</deviceid>
        <deviceid build="9200">1502</deviceid>
        <deviceid build="9200">1503</deviceid>
        <deviceid build="9600">153A</deviceid>
        <deviceid build="9600">153B</deviceid>
        <deviceid build="9600">155A</deviceid>
        <deviceid build="9600">1559</deviceid>
        <deviceid build="15063">15A0</deviceid>
        <deviceid build="15063">15A1</deviceid>
        <deviceid build="15063">15A2</deviceid>
        <deviceid build="15063">15A3</deviceid>
        <deviceid build="15063">156F</deviceid>
        <deviceid build="15063">1570</deviceid>
        <deviceid build="15063">15B7</deviceid>
        <deviceid build="15063">15B8</deviceid>
        <deviceid build="16299">15B9</deviceid>
        <deviceid build="16299">15D7</deviceid>
        <deviceid build="16299">15D8</deviceid>
        <deviceid build="16299">15E3</deviceid>
        <deviceid build="16299">15D6</deviceid>
        <deviceid build="16299">15BD</deviceid>
        <deviceid build="16299">15BE</deviceid>
        <deviceid build="16299">15BB</deviceid>
        <deviceid build="16299">15BC</deviceid>
        <deviceid build="16299">15DF</deviceid>
        <deviceid build="16299">15E0</deviceid>
        <deviceid build="16299">15E1</deviceid>
        <deviceid build="16299">15E2</deviceid>
        <deviceid build="9600">10C9</deviceid>
        <deviceid build="9200">10E6</deviceid>
        <deviceid build="9200">10E7</deviceid>
        <deviceid build="9200">10E8</deviceid>
        <deviceid build="9200">1526</deviceid>
        <deviceid build="9200">150A</deviceid>
        <deviceid build="9200">1518</deviceid>
        <deviceid build="9200">150D</deviceid>
        <deviceid build="9200">10A7</deviceid>
        <deviceid build="9200">10A9</deviceid>
        <deviceid build="9600">10D6</deviceid>
        <deviceid build="9200">150E</deviceid>
        <deviceid build="9200">150F</deviceid>
        <deviceid build="9200">1510</deviceid>
        <deviceid build="9200">1511</deviceid>
        <deviceid build="9200">1516</deviceid>
        <deviceid build="9600">1527</deviceid>
        <deviceid build="9200">1521</deviceid>
        <deviceid build="9200">1522</deviceid>
        <deviceid build="9200">1523</deviceid>
        <deviceid build="9200">1524</deviceid>
        <deviceid build="9600">1546</deviceid>
        <deviceid build="9600">1533</deviceid>
```

```
        <deviceid build="9600">1534</deviceid>
        <deviceid build="9600">1535</deviceid>
        <deviceid build="9600">1536</deviceid>
        <deviceid build="9600">1537</deviceid>
        <deviceid build="9600">1538</deviceid>
        <deviceid build="9600">157B</deviceid>
        <deviceid build="9600">157C</deviceid>
        <deviceid build="9600">1539</deviceid>
        <deviceid build="9600">1F40</deviceid>
        <deviceid build="9600">1F41</deviceid>
        <deviceid build="9600">1F45</deviceid>
        <deviceid build="9600">0438</deviceid>
        <deviceid build="9600">043A</deviceid>
        <deviceid build="9600">043C</deviceid>
        <deviceid build="9600">0440</deviceid>
        <deviceid build="9600">10C6</deviceid>
        <deviceid build="9600">10C7</deviceid>
        <deviceid build="9600">10C8</deviceid>
        <deviceid build="9600">150B</deviceid>
        <deviceid build="9600">10DB</deviceid>
        <deviceid build="9600">10DD</deviceid>
        <deviceid build="9600">10EC</deviceid>
        <deviceid build="9600">10F1</deviceid>
        <deviceid build="9600">10E1</deviceid>
        <deviceid build="9600">10F4</deviceid>
        <deviceid build="9600">10F7</deviceid>
        <deviceid build="9600">1514</deviceid>
        <deviceid build="9600">1517</deviceid>
        <deviceid build="9600">10F8</deviceid>
        <deviceid build="15063">000C</deviceid>
        <deviceid build="15063">1F63</deviceid>
        <deviceid build="9600">10F9</deviceid>
        <deviceid build="9600">10FB</deviceid>
        <deviceid build="15063">11A9</deviceid>
        <deviceid build="15063">1071</deviceid>
        <deviceid build="15063">1F72</deviceid>
        <deviceid build="15063">17D0</deviceid>
        <deviceid build="15063">0470</deviceid>
        <deviceid build="15063">211B</deviceid>
        <deviceid build="15063">8976</deviceid>
        <deviceid build="15063">2159</deviceid>
        <deviceid build="15063">000D</deviceid>
        <deviceid build="15063">0008</deviceid>
        <deviceid build="9600">152A</deviceid>
        <deviceid build="9600">1529</deviceid>
        <deviceid build="9600">1507</deviceid>
        <deviceid build="9600">154D</deviceid>
        <deviceid build="9600">154A</deviceid>
        <deviceid build="9600">1558</deviceid>
        <deviceid build="9600">1557</deviceid>
        <deviceid build="15063">0001</deviceid>
        <deviceid build="9600">10FC</deviceid>
        <deviceid build="9600">151C</deviceid>
        <deviceid build="9600">1528</deviceid>
        <deviceid build="9600">1560</deviceid>
```

```
        <deviceid build="15063">1563</deviceid>
        <deviceid build="15063">ABCD</deviceid>
        <deviceid build="15063">15AA</deviceid>
        <deviceid build="15063">15AB</deviceid>
        <deviceid build="15063">15AC</deviceid>
        <deviceid build="15063">15AD</deviceid>
        <deviceid build="15063">15AE</deviceid>
        <deviceid build="19565">0D4E</deviceid>
        <deviceid build="19565">0D4F</deviceid>
        <deviceid build="19565">0D4C</deviceid>
        <deviceid build="19565">0D4D</deviceid>
        <deviceid build="19565">0D53</deviceid>
        <deviceid build="19565">0D55</deviceid>
        <deviceid build="19565">15FB</deviceid>
        <deviceid build="19565">15FC</deviceid>
        <deviceid build="19565">15F9</deviceid>
        <deviceid build="19565">15FA</deviceid>
        <deviceid build="19565">15F4</deviceid>
        <deviceid build="20271">15F5</deviceid>
        <deviceid build="20271">1A1E</deviceid>
        <deviceid build="20271">1A1F</deviceid>
        <deviceid build="20271">1A1C</deviceid>
        <deviceid build="20271">1A1D</deviceid>
        <deviceid build="20271">550A</deviceid>
        <deviceid build="20271">550B</deviceid>
        <deviceid build="20271">550C</deviceid>
        <deviceid build="20271">550D</deviceid>
        <deviceid build="20271">1572</deviceid>
        <deviceid build="20271">1574</deviceid>
        <deviceid build="20271">1580</deviceid>
        <deviceid build="20271">1581</deviceid>
        <deviceid build="20271">1583</deviceid>
        <deviceid build="20271">1584</deviceid>
        <deviceid build="20271">1585</deviceid>
        <deviceid build="20271">1586</deviceid>
        <deviceid build="20271">1587</deviceid>
        <deviceid build="20271">1588</deviceid>
        <deviceid build="20271">1589</deviceid>
        <deviceid build="20271">158A</deviceid>
        <deviceid build="20271">158B</deviceid>
        <deviceid build="20271">37D0</deviceid>
        <deviceid build="20271">37CC</deviceid>
        <deviceid build="20271">37D2</deviceid>
        <deviceid build="20271">37CE</deviceid>
        <deviceid build="20271">37CF</deviceid>
        <deviceid build="20271">37D0</deviceid>
        <deviceid build="20271">37D1</deviceid>
        <deviceid build="20271">37D2</deviceid>
        <deviceid build="20271">37D3</deviceid>
        <deviceid build="20271">37D4</deviceid>
        <deviceid build="22536">550E</deviceid>
        <deviceid build="22536">550F</deviceid>
        <deviceid build="22536">5510</deviceid>
        <deviceid build="22536">5511</deviceid>
        <deviceid build="22536">0DC7</deviceid>
```

```
      <deviceid build="22536">0DC8</deviceid>
      <deviceid build="22536">0DC5</deviceid>
      <deviceid build="22536">0DC6</deviceid>
      <deviceid build="25215">57A0</deviceid>
      <deviceid build="25215">57A1</deviceid>
      <deviceid build="25215">57B3</deviceid>
      <deviceid build="25215">57B4</deviceid>
      <deviceid build="25215">57B5</deviceid>
      <deviceid build="25215">57B6</deviceid>
      <deviceid build="25215">57B7</deviceid>
      <deviceid build="25215">57B8</deviceid>
      <deviceid build="25250">15F2</deviceid>
      <deviceid build="25250">15F3</deviceid>
      <deviceid build="25250">0D9F</deviceid>
      <deviceid build="25250">3100</deviceid>
      <deviceid build="25250">3101</deviceid>
      <deviceid build="25250">3102</deviceid>
      <deviceid build="25250">5502</deviceid>
      <deviceid build="25250">125B</deviceid>
      <deviceid build="25250">125C</deviceid>
      <deviceid build="25250">125D</deviceid>
      <deviceid build="25250">125E</deviceid>
      <deviceid build="25250">5503</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10EC</manufacturer>
      <deviceid build="9200">8136</deviceid>
      <deviceid build="9200">8137</deviceid>
      <deviceid build="9200">8168</deviceid>
      <deviceid build="9200">8167</deviceid>
      <deviceid build="9200">8169</deviceid>
      <deviceid build="15063">8166</deviceid>
      <deviceid build="18362">8161</deviceid>
      <deviceid build="18362">8125</deviceid>
      <deviceid build="18362">8225</deviceid>
      <deviceid build="18362">2502</deviceid>
      <deviceid build="18362">2600</deviceid>
      <deviceid build="18362">3000</deviceid>
      <deviceid build="25876">8126</deviceid>
      <deviceid build="25876">8162</deviceid>
      <deviceid build="25876">5000</deviceid>
  </NIC>

  <NIC>
    <manufacturer>14E4</manufacturer>
      <deviceid build="9200">1644</deviceid>
      <deviceid build="9200">1645</deviceid>
      <deviceid build="9200">1646</deviceid>
      <deviceid build="9200">16A6</deviceid>
      <deviceid build="9200">16C6</deviceid>
      <deviceid build="9200">1647</deviceid>
      <deviceid build="9200">16A7</deviceid>
      <deviceid build="9200">16C7</deviceid>
      <deviceid build="9200">164D</deviceid>
```

```
        <deviceid build="9200">1648</deviceid>
        <deviceid build="9200">16A8</deviceid>
        <deviceid build="9200">1653</deviceid>
        <deviceid build="9200">166E</deviceid>
        <deviceid build="9200">1654</deviceid>
        <deviceid build="9200">165D</deviceid>
        <deviceid build="9200">165E</deviceid>
        <deviceid build="9200">166D</deviceid>
        <deviceid build="9200">170D</deviceid>
        <deviceid build="9200">170E</deviceid>
        <deviceid build="9200">1696</deviceid>
        <deviceid build="9200">1676</deviceid>
        <deviceid build="9200">1677</deviceid>
        <deviceid build="9200">1659</deviceid>
        <deviceid build="9200">167C</deviceid>
        <deviceid build="9200">167D</deviceid>
        <deviceid build="9200">169D</deviceid>
        <deviceid build="9200">16F7</deviceid>
        <deviceid build="9200">16FD</deviceid>
        <deviceid build="9200">16FE</deviceid>
        <deviceid build="9200">16DD</deviceid>
        <deviceid build="9200">1668</deviceid>
        <deviceid build="9200">1669</deviceid>
        <deviceid build="9200">1678</deviceid>
        <deviceid build="9200">1679</deviceid>
        <deviceid build="9200">1600</deviceid>
        <deviceid build="9200">1601</deviceid>
        <deviceid build="9200">166A</deviceid>
        <deviceid build="9200">166B</deviceid>
        <deviceid build="9200">16FF</deviceid>
        <deviceid build="9200">170F</deviceid>
        <deviceid build="9200">169B</deviceid>
        <deviceid build="9200">1693</deviceid>
        <deviceid build="9200">167F</deviceid>
        <deviceid build="9200">169A</deviceid>
        <deviceid build="9200">1698</deviceid>
        <deviceid build="9200">1692</deviceid>
        <deviceid build="9200">1694</deviceid>
        <deviceid build="9200">1690</deviceid>
        <deviceid build="9200">1691</deviceid>
        <deviceid build="9200">1699</deviceid>
        <deviceid build="9200">16A0</deviceid>
        <deviceid build="9200">167B</deviceid>
        <deviceid build="9200">1673</deviceid>
        <deviceid build="9200">165A</deviceid>
        <deviceid build="9200">1674</deviceid>
        <deviceid build="9200">1681</deviceid>
        <deviceid build="9200">1680</deviceid>
        <deviceid build="9200">1688</deviceid>
        <deviceid build="9200">167A</deviceid>
        <deviceid build="9200">1672</deviceid>
        <deviceid build="9200">1684</deviceid>
        <deviceid build="9200">165B</deviceid>
        <deviceid build="9200">16B1</deviceid>
        <deviceid build="9200">16B5</deviceid>
```

```
        <deviceid build="9200">16B0</deviceid>
        <deviceid build="9200">16B4</deviceid>
        <deviceid build="9200">16B2</deviceid>
        <deviceid build="9200">16B6</deviceid>
        <deviceid build="15063">1682</deviceid>
        <deviceid build="15063">1686</deviceid>
        <deviceid build="15063">16B3</deviceid>
        <deviceid build="15063">16B7</deviceid>
        <deviceid build="9200">1655</deviceid>
        <deviceid build="15063">1665</deviceid>
        <deviceid build="9200">1656</deviceid>
        <deviceid build="9200">1657</deviceid>
        <deviceid build="9200">165F</deviceid>
        <deviceid build="9200">165C</deviceid>
        <deviceid build="15063">1683</deviceid>
        <deviceid build="15063">1641</deviceid>
        <deviceid build="15063">1642</deviceid>
        <deviceid build="15063">1643</deviceid>
        <deviceid build="15063">1687</deviceid>
        <deviceid build="9200">164A</deviceid>
        <deviceid build="9200">16AA</deviceid>
        <deviceid build="9200">164C</deviceid>
        <deviceid build="9200">16AC</deviceid>
        <deviceid build="9200">1639</deviceid>
        <deviceid build="9200">163A</deviceid>
        <deviceid build="9200">163B</deviceid>
        <deviceid build="9200">163C</deviceid>
        <deviceid build="15063">164E</deviceid>
        <deviceid build="15063">164F</deviceid>
        <deviceid build="15063">1650</deviceid>
        <deviceid build="15063">1662</deviceid>
        <deviceid build="15063">1663</deviceid>
        <deviceid build="15063">168A</deviceid>
        <deviceid build="15063">168D</deviceid>
        <deviceid build="15063">16A1</deviceid>
        <deviceid build="15063">16A2</deviceid>
        <deviceid build="15063">168E</deviceid>
        <deviceid build="15063">163D</deviceid>
        <deviceid build="15063">16A5</deviceid>
        <deviceid build="15063">16A4</deviceid>
        <deviceid build="15063">16AE</deviceid>
        <deviceid build="15063">163E</deviceid>
        <deviceid build="18362">16C0</deviceid>
        <deviceid build="18362">16C9</deviceid>
        <deviceid build="18362">16CA</deviceid>
        <deviceid build="18362">16CC</deviceid>
        <deviceid build="18362">16CD</deviceid>
        <deviceid build="18362">16CE</deviceid>
        <deviceid build="18362">16CF</deviceid>
        <deviceid build="18362">16D0</deviceid>
        <deviceid build="18362">16D1</deviceid>
        <deviceid build="18362">16D2</deviceid>
        <deviceid build="18362">16D4</deviceid>
        <deviceid build="18362">16D5</deviceid>
        <deviceid build="18362">16D6</deviceid>
```

```
        <deviceid build="18362">16D7</deviceid>
        <deviceid build="18362">16D8</deviceid>
        <deviceid build="18362">16D9</deviceid>
        <deviceid build="18362">16DA</deviceid>
        <deviceid build="18362">16DB</deviceid>
        <deviceid build="18362">16DC</deviceid>
        <deviceid build="18362">16DE</deviceid>
        <deviceid build="18362">16DF</deviceid>
        <deviceid build="18362">16E0</deviceid>
        <deviceid build="18362">16E2</deviceid>
        <deviceid build="18362">16E3</deviceid>
        <deviceid build="18362">16E4</deviceid>
        <deviceid build="18362">16E7</deviceid>
        <deviceid build="18362">16E8</deviceid>
        <deviceid build="18362">16E9</deviceid>
        <deviceid build="18362">16EA</deviceid>
        <deviceid build="18362">16EB</deviceid>
        <deviceid build="18362">16EC</deviceid>
        <deviceid build="18362">16ED</deviceid>
        <deviceid build="18362">16EE</deviceid>
        <deviceid build="18362">16EF</deviceid>
        <deviceid build="18362">16F0</deviceid>
        <deviceid build="18362">16F1</deviceid>
        <deviceid build="18362">1614</deviceid>
        <deviceid build="18362">D802</deviceid>
        <deviceid build="20241">1604</deviceid>
        <deviceid build="20241">1605</deviceid>
        <deviceid build="20241">1606</deviceid>
        <deviceid build="20241">1607</deviceid>
        <deviceid build="20241">1608</deviceid>
        <deviceid build="20241">1609</deviceid>
        <deviceid build="20241">D804</deviceid>
        <deviceid build="20241">D812</deviceid>
        <deviceid build="20241">D814</deviceid>
        <deviceid build="20241">D818</deviceid>
        <deviceid build="20241">D82A</deviceid>
        <deviceid build="20241">D82B</deviceid>
        <deviceid build="20241">D82C</deviceid>
        <deviceid build="20241">D82D</deviceid>
        <deviceid build="20241">D82E</deviceid>
        <deviceid build="20241">D82F</deviceid>
        <deviceid build="20241">1902</deviceid>
        <deviceid build="20241">1903</deviceid>
        <deviceid build="20241">1906</deviceid>
        <deviceid build="20241">1907</deviceid>
        <deviceid build="20241">190A</deviceid>
        <deviceid build="20241">190B</deviceid>
        <deviceid build="20241">1799</deviceid>
        <deviceid build="20241">1041</deviceid>
        <deviceid build="20241">1042</deviceid>
        <deviceid build="20241">1043</deviceid>
        <deviceid build="20241">1750</deviceid>
        <deviceid build="20241">1751</deviceid>
        <deviceid build="20241">1752</deviceid>
        <deviceid build="20241">1800</deviceid>
```

```
      <deviceid build="20241">1801</deviceid>
      <deviceid build="20241">1802</deviceid>
      <deviceid build="20241">1803</deviceid>
      <deviceid build="20241">1804</deviceid>
      <deviceid build="20241">1805</deviceid>
      <deviceid build="20241">1806</deviceid>
      <deviceid build="20241">1807</deviceid>
      <deviceid build="20241">1808</deviceid>
      <deviceid build="20241">1809</deviceid>
      <deviceid build="20241">16C1</deviceid>
      <deviceid build="20241">16C5</deviceid>
      <deviceid build="20241">16BD</deviceid>
  </NIC>

  <NIC>
    <manufacturer>1969</manufacturer>
      <deviceid build="9200">1062</deviceid>
      <deviceid build="9200">1063</deviceid>
      <deviceid build="9200">2060</deviceid>
      <deviceid build="9200">2062</deviceid>
      <deviceid build="9200">1073</deviceid>
      <deviceid build="9200">1083</deviceid>
      <deviceid build="9200">1090</deviceid>
      <deviceid build="9200">1091</deviceid>
      <deviceid build="9200">E091</deviceid>
      <deviceid build="9200">10A0</deviceid>
      <deviceid build="9200">10A1</deviceid>
      <deviceid build="9200">E0A1</deviceid>
      <deviceid build="9200">10B0</deviceid>
      <deviceid build="9200">10B1</deviceid>
      <deviceid build="9200">E0B1</deviceid>
      <deviceid build="9200">10C0</deviceid>
      <deviceid build="9200">10C1</deviceid>
      <deviceid build="9200">E0C1</deviceid>
      <deviceid build="9200">10D0</deviceid>
      <deviceid build="9200">10D1</deviceid>
      <deviceid build="9200">E0D1</deviceid>
      <deviceid build="9200">10E0</deviceid>
      <deviceid build="9200">10E1</deviceid>
      <deviceid build="9200">E0E1</deviceid>
      <deviceid build="9200">10F0</deviceid>
      <deviceid build="9200">10F1</deviceid>
      <deviceid build="9200">E0F1</deviceid>
  </NIC>

  <NIC>
    <manufacturer>19A2</manufacturer>
      <deviceid build="9600">0211</deviceid>
      <deviceid build="9600">0215</deviceid>
      <deviceid build="9600">0221</deviceid>
      <deviceid build="9600">0700</deviceid>
      <deviceid build="9600">0710</deviceid>
  </NIC>

  <NIC>
```

```
    <manufacturer>10DF</manufacturer>
      <deviceid build="9600">0720</deviceid>
      <deviceid build="9600">E220</deviceid>
  </NIC>

  <NIC>
    <manufacturer>15B3</manufacturer>
      <deviceid build="9600">1000</deviceid>
      <deviceid build="9600">1001</deviceid>
      <deviceid build="9600">1002</deviceid>
      <deviceid build="9600">1003</deviceid>
      <deviceid build="9600">1004</deviceid>
      <deviceid build="9600">1005</deviceid>
      <deviceid build="9600">1006</deviceid>
      <deviceid build="9600">1007</deviceid>
      <deviceid build="9600">1008</deviceid>
      <deviceid build="9600">1009</deviceid>
      <deviceid build="9600">100A</deviceid>
      <deviceid build="9600">100B</deviceid>
      <deviceid build="9600">100C</deviceid>
      <deviceid build="9600">100D</deviceid>
      <deviceid build="9600">100E</deviceid>
      <deviceid build="9600">100F</deviceid>
      <deviceid build="9600">1010</deviceid>
      <deviceid build="9600">6340</deviceid>
      <deviceid build="9600">6341</deviceid>
      <deviceid build="9600">634A</deviceid>
      <deviceid build="9600">634B</deviceid>
      <deviceid build="9600">6354</deviceid>
      <deviceid build="9600">6368</deviceid>
      <deviceid build="9600">6369</deviceid>
      <deviceid build="9600">6372</deviceid>
      <deviceid build="9600">6732</deviceid>
      <deviceid build="9600">6733</deviceid>
      <deviceid build="9600">673C</deviceid>
      <deviceid build="9600">673D</deviceid>
      <deviceid build="9600">6746</deviceid>
      <deviceid build="9600">6750</deviceid>
      <deviceid build="9600">6751</deviceid>
      <deviceid build="9600">675A</deviceid>
      <deviceid build="9600">6764</deviceid>
      <deviceid build="9600">6765</deviceid>
      <deviceid build="9600">676E</deviceid>
      <deviceid build="9600">6778</deviceid>
      <deviceid build="15063">1013</deviceid>
      <deviceid build="15063">1015</deviceid>
      <deviceid build="15063">1017</deviceid>
      <deviceid build="15063">1019</deviceid>
      <deviceid build="15063">101B</deviceid>
      <deviceid build="15063">101D</deviceid>
      <deviceid build="15063">101F</deviceid>
      <deviceid build="15063">1021</deviceid>
      <deviceid build="15063">1023</deviceid>
      <deviceid build="15063">1025</deviceid>
      <deviceid build="15063">1027</deviceid>
```

```
        <deviceid build="15063">1029</deviceid>
        <deviceid build="15063">102B</deviceid>
        <deviceid build="15063">102F</deviceid>
    </NIC>

    <NIC>
      <manufacturer>1137</manufacturer>
        <deviceid build="15063">0043</deviceid>
    </NIC>

    <NIC>
      <manufacturer>1414</manufacturer>
        <deviceid build="25500">00B9</deviceid>
    </NIC>

    <NIC>
      <manufacturer>15AD</manufacturer>
        <deviceid build="26006">07B0</deviceid>
    </NIC>

</SupportedNetworkInterfaceCards>
```

另请参阅

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

Windows 11 21H2 中网络内核调试支持的以太网 NIC

Windows 11 22H2 中网络内核调试支持的以太网 NIC

Windows 10 中网络内核调试支持的以太网 NIC

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Windows 11 22H2 中网络内核调试支持的以太网 NIC

项目 • 2024/10/17

若要通过以太网网络电缆执行内核调试，目标计算机必须具有受支持的网络接口卡（NIC）。

在内核调试期间，运行调试器的计算机称为 *主计算机*，正在调试的计算机称为 *目标计算机*。 有关详细信息，请参阅设置 KDNET 网络内核自动调试。

若要通过网络电缆进行内核调试，目标计算机必须具有受支持的网络适配器。 当目标计算机运行 Windows 时，内核调试支持此处列出的网络适配器。

版本信息

本主题列出了以下版本的 Windows 支持的适配器

- Windows 11、22H2 内部版本 22621

有关 Windows 11 版本的信息，请参阅 Windows 11 版本信息。

使用 VerifiedNicList.xml 确定 NIC 支持

若要确定任何特定版本的 Windows 支持哪个 NIC 集，请检查 `VerifiedNicList.xml` 随该特定版本的 Windows 附带的 WDK 或 SDK 安装的调试器目录中的文件。 对于 64 位 Windows，默认情况下，它将安装在此目录中：

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\VerifiedNicList.xml
```

需要检查特定版本的VerifiedNicList.xml，因为将其他硬件支持添加到以前版本中不存在的 Windows 新版本中。 因此，必须检查该特定版本的VerifiedNicLIst.xml文件。

查找供应商 ID 和设备 ID

在目标计算机上查找网络适配器的供应商 ID 和设备 ID。

- 在目标计算机上，打开设备管理器（在命令提示符窗口中输入 **devmgmt**）。
- 在设备管理器中，找到要用于调试的网络适配器。
- 选择并按住（或右键单击）网络适配器节点，然后选择“ **属性**”。
- 在“**详细信息**”选项卡的“属性”**下**，选择“**硬件 ID**”。

供应商和设备 ID 显示为 VEN_*VendorID* 和 DEV_*DeviceID*。 例如，如果看到 PCI\VEN_8086&DEV_104B，则供应商 ID 为 8086，设备 ID 为 104B。

供应商 ID 8086，Intel Corporation

对于供应商 ID 8086，支持以下设备 ID：

0001 0008 000C 000D 0438 043A 043C 0440 0470 0D4E 0D4F 0D4C 0D4D 0D53 0D55 1000 1004 1004 100 8 1009 100C 100D 100E 100F 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 101A 101D 101D 101E 1026 1027 1028 1049 104A 104B 104C 104D 105E 105F 1060 1071 1075 1076 1077 1078 1079 1079 107A 107B 107B 107B 107C 107D 107E 107F 108A 108B 108C 1096 1098 1099 109A 10A4 10A5 10A7 10A9 10B5 10B5 10B9 10BA 10BB10BC 10BD 10BF 10C0 10C2 10C3 10C4 10C5 10C6 10C7 10C8 10C9 10CB 10CC 10CD 10CE 10D3 10D5 10D6 10D9 10DA 10DB 10DD 10DE 10DF 10E1 10E5 10E6 10E7 10E8 10EA 10EB 10EC 10EF 10F0 10F1 10F4 10F5 10F6 10F7 10F8 10F9 10FB 10FC 11A9 1501 1502 1503 1507 150A 150B 150C 150D 150E 150F 150F 1510 1511 15111514 1516 1517 1518 151C 1521 1522 1523 1524 1525 1526 1527 1528 1529 152A 1533 1534 15351536 1537 1538 1539 153A 153B 1546 154A 154D 1557 1558 1559 155A 1560 1560 1563 156F 1570 157B 157C 15A0 15A1 15A2 15A3 15AA 15AB 15AC 15AD 15AE 15B7 15B8 15B9 15BB 15BC 15BD 15BE 15D6 15D7 15D8 15DF 15E0 15E1 15E2 15E3 15F 4 15F9 15FA 15FB 15FC 17D0 1F40 1F41 1F45 1F63 1F72 211B 2159 294C 8976

Windows 11 21H2 内部版本 22000 中的新增功能

1572 1574 1580 1581 1583 1584 1585 1586 1587 1588 1588 1589 15F5 1A1E 1A1F 1A1C 1A1D 158A 158B 37D0 37CC 37D2 37CE 37CF 37D0 37D1 37D2 37D3 37D4 550A 550B 550C 550D

Windows 11 22H2 内部版本 22621 中的新增功能

550E 550F 5510 5511 0DC7 0DC8 0DC5 0DC6 57A0 57A1 57B3 57B4 57B5 57B6 57B7 57B8 15F2 15F3 0D9F 3100 3101 3102 5502 125B 125C 125D 125E 5503

供应商 ID 10EC，Realtek 半导体公司。

对于供应商 ID 10EC，支持以下设备 ID：

2502 2600 3000 8125 8136 8137 8161 8166 8167 8168 8169 8225

Windows 11 22H2 内部版本 22621 中的新增功能

8126 8162 5000

供应商 ID 14E4，Broadcom

对于供应商 ID 14E4，支持以下设备 ID：

1600 1601 1614 1639 163A 163B 163C 163D 163E 1641 1642 1644 1645 1646 1647 1648 1648 164A 164C 164D 164E 164F 1650 1653 1654 1655 1656 1657 1659 165A 165B 165C 165D 165E 165F 165F 1662 1663 1665 1668 1669 166A 166B 166D 166E 1672 1673 1674 1676 1677 1678 1679 167A 167B 167C 167D 167F 168A 168D 168E 1680 1681 1682 1683 1684 1686 1687 1688 1690 1691 1692 1693 1694 1696 1698 1699169A 169B 169D 16A0 16A1 16A2 16A4 16A5 16A6 16A7 16A8 16AA 16AC 1 6AE 16B0 16B1 16B2 16B3 16B4 16B5 16B6 16B7 16C6 16C7 16C9 16CA 1 6CE 16CF 16D0 16D1 16D2 16D5 16D6 16D7 16D8 16D9 16DD 16DF 16E0 16E2 16E3 16E4 16E9 16F0 16F1 16F7 16FD 16FE 16FF 170D 170E 170F D802

Windows 11 21H2 内部版本 22000 中的新增功能

1041 1042 1043 1604 1605 1606 1607 1608 1609 16C1 16C5 16BD 1750 1751 1752 1799 1800 1801 1802 1803 1804 18041805 1806 1807 1808 1809 1902 1903 1906 1907 190A 190B D804 D812 D814 D818 D82A D82B D82C D82D D82E D82F

供应商 ID 1969，Atheros Communications

对于供应商 ID 1969，支持以下设备 ID：

1062 1063 1073 1083 1090 1091 10A0 10A1 10B0 10B1 10C0 10C0 10C1 10D0 10D1 10E0 10E1 10F0 10F1 2060 2062 E091 E0A1 E0B1 E0C1 E0D1 E0E1 E0E1 E0F1

供应商 ID 19A2，ServerEngines （Emulex)

对于供应商 ID 19A2，支持以下设备 ID：

0211 0215 0221 0700 0710

供应商 ID 10DF，Emulex Corporation

对于供应商 ID 10DF，支持以下设备 ID：

0720 E220

供应商 ID 15B3，Mellanox 技术

对于供应商 ID 15B3，支持以下设备 ID：

1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 100A 100B 100C 100D 100E 100F
1010 1013 1015 1017 1019 101B 101D 101F 1021 10 23 1025 1027 1029 102B 102F
6340 6341 634A 634B 6354 6368 6369 6372 6732 6733 673C 673D 6746 6750 6751
675A 6764 6765 676E 6778

供应商 ID 1137，Cisco Systems Inc

对于供应商 ID 1137，支持以下设备 ID：

0043

供应商 ID 1414，Microsoft Corporation

Windows 11 22H2 内部版本 22621 中的新增功能

对于供应商 ID 1414，支持以下设备 ID：

00B9

XML 支持的 NIC 列表

这是上面以 XML 格式显示的相同信息。

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<SupportedNetworkInterfaceCards>
  <NIC>
    <manufacturer>8086</manufacturer>
      <deviceid build="9200">1000</deviceid>
      <deviceid build="9200">1001</deviceid>
      <deviceid build="9200">1004</deviceid>
      <deviceid build="9200">1008</deviceid>
      <deviceid build="9200">1009</deviceid>
      <deviceid build="9200">100C</deviceid>
      <deviceid build="9200">100D</deviceid>
      <deviceid build="9200">100E</deviceid>
      <deviceid build="9200">1015</deviceid>
      <deviceid build="9200">1016</deviceid>
      <deviceid build="9200">1017</deviceid>
      <deviceid build="9200">101E</deviceid>
      <deviceid build="9200">100F</deviceid>
      <deviceid build="9200">1011</deviceid>
      <deviceid build="9200">1026</deviceid>
      <deviceid build="9200">1027</deviceid>
      <deviceid build="9200">1028</deviceid>
      <deviceid build="9200">1010</deviceid>
      <deviceid build="9200">1012</deviceid>
```

```
        <deviceid build="9200">101D</deviceid>
        <deviceid build="9200">1079</deviceid>
        <deviceid build="9200">107A</deviceid>
        <deviceid build="9200">107B</deviceid>
        <deviceid build="9200">108A</deviceid>
        <deviceid build="9200">1099</deviceid>
        <deviceid build="9200">10B5</deviceid>
        <deviceid build="9200">1013</deviceid>
        <deviceid build="9200">1018</deviceid>
        <deviceid build="9200">1014</deviceid>
        <deviceid build="9200">1078</deviceid>
        <deviceid build="9200">1076</deviceid>
        <deviceid build="9200">107C</deviceid>
        <deviceid build="9200">1077</deviceid>
        <deviceid build="9200">1019</deviceid>
        <deviceid build="9200">101A</deviceid>
        <deviceid build="9200">1075</deviceid>
        <deviceid build="9200">105E</deviceid>
        <deviceid build="9200">105F</deviceid>
        <deviceid build="9200">1060</deviceid>
        <deviceid build="9200">10D9</deviceid>
        <deviceid build="9200">10DA</deviceid>
        <deviceid build="9200">10A4</deviceid>
        <deviceid build="9200">10D5</deviceid>
        <deviceid build="9200">10A5</deviceid>
        <deviceid build="9600">10BC</deviceid>
        <deviceid build="9200">107D</deviceid>
        <deviceid build="9200">107E</deviceid>
        <deviceid build="9200">107F</deviceid>
        <deviceid build="9600">10B9</deviceid>
        <deviceid build="9200">108B</deviceid>
        <deviceid build="9200">108C</deviceid>
        <deviceid build="9200">109A</deviceid>
        <deviceid build="9600">10D3</deviceid>
        <deviceid build="9200">10F6</deviceid>
        <deviceid build="9200">150C</deviceid>
        <deviceid build="9200">1096</deviceid>
        <deviceid build="9200">1098</deviceid>
        <deviceid build="9600">10BA</deviceid>
        <deviceid build="9600">10BB</deviceid>
        <deviceid build="9200">1501</deviceid>
        <deviceid build="9200">1049</deviceid>
        <deviceid build="9200">104A</deviceid>
        <deviceid build="9200">104B</deviceid>
        <deviceid build="9200">104C</deviceid>
        <deviceid build="9600">10C4</deviceid>
        <deviceid build="9600">10C5</deviceid>
        <deviceid build="9200">104D</deviceid>
        <deviceid build="9600">10BF</deviceid>
        <deviceid build="9200">10F5</deviceid>
        <deviceid build="9600">10CB</deviceid>
        <deviceid build="9600">10BD</deviceid>
        <deviceid build="9200">10E5</deviceid>
        <deviceid build="9200">294C</deviceid>
        <deviceid build="9600">10C0</deviceid>
```

```
        <deviceid build="9600">10C3</deviceid>
        <deviceid build="9600">10C2</deviceid>
        <deviceid build="9600">10CC</deviceid>
        <deviceid build="9600">10CD</deviceid>
        <deviceid build="9600">10CE</deviceid>
        <deviceid build="9600">10DE</deviceid>
        <deviceid build="9600">10DF</deviceid>
        <deviceid build="9600">1525</deviceid>
        <deviceid build="9200">10EA</deviceid>
        <deviceid build="9200">10EB</deviceid>
        <deviceid build="9200">10EF</deviceid>
        <deviceid build="9200">10F0</deviceid>
        <deviceid build="9200">1502</deviceid>
        <deviceid build="9200">1503</deviceid>
        <deviceid build="9600">153A</deviceid>
        <deviceid build="9600">153B</deviceid>
        <deviceid build="9600">155A</deviceid>
        <deviceid build="9600">1559</deviceid>
        <deviceid build="15063">15A0</deviceid>
        <deviceid build="15063">15A1</deviceid>
        <deviceid build="15063">15A2</deviceid>
        <deviceid build="15063">15A3</deviceid>
        <deviceid build="15063">156F</deviceid>
        <deviceid build="15063">1570</deviceid>
        <deviceid build="15063">15B7</deviceid>
        <deviceid build="15063">15B8</deviceid>
        <deviceid build="16299">15B9</deviceid>
        <deviceid build="16299">15D7</deviceid>
        <deviceid build="16299">15D8</deviceid>
        <deviceid build="16299">15E3</deviceid>
        <deviceid build="16299">15D6</deviceid>
        <deviceid build="16299">15BD</deviceid>
        <deviceid build="16299">15BE</deviceid>
        <deviceid build="16299">15BB</deviceid>
        <deviceid build="16299">15BC</deviceid>
        <deviceid build="16299">15DF</deviceid>
        <deviceid build="16299">15E0</deviceid>
        <deviceid build="16299">15E1</deviceid>
        <deviceid build="16299">15E2</deviceid>
        <deviceid build="9600">10C9</deviceid>
        <deviceid build="9200">10E6</deviceid>
        <deviceid build="9200">10E7</deviceid>
        <deviceid build="9200">10E8</deviceid>
        <deviceid build="9200">1526</deviceid>
        <deviceid build="9200">150A</deviceid>
        <deviceid build="9200">1518</deviceid>
        <deviceid build="9200">150D</deviceid>
        <deviceid build="9200">10A7</deviceid>
        <deviceid build="9200">10A9</deviceid>
        <deviceid build="9600">10D6</deviceid>
        <deviceid build="9200">150E</deviceid>
        <deviceid build="9200">150F</deviceid>
        <deviceid build="9200">1510</deviceid>
        <deviceid build="9200">1511</deviceid>
        <deviceid build="9200">1516</deviceid>
```

```
        <deviceid build="9600">1527</deviceid>
        <deviceid build="9200">1521</deviceid>
        <deviceid build="9200">1522</deviceid>
        <deviceid build="9200">1523</deviceid>
        <deviceid build="9200">1524</deviceid>
        <deviceid build="9600">1546</deviceid>
        <deviceid build="9600">1533</deviceid>
        <deviceid build="9600">1534</deviceid>
        <deviceid build="9600">1535</deviceid>
        <deviceid build="9600">1536</deviceid>
        <deviceid build="9600">1537</deviceid>
        <deviceid build="9600">1538</deviceid>
        <deviceid build="9600">157B</deviceid>
        <deviceid build="9600">157C</deviceid>
        <deviceid build="9600">1539</deviceid>
        <deviceid build="9600">1F40</deviceid>
        <deviceid build="9600">1F41</deviceid>
        <deviceid build="9600">1F45</deviceid>
        <deviceid build="9600">0438</deviceid>
        <deviceid build="9600">043A</deviceid>
        <deviceid build="9600">043C</deviceid>
        <deviceid build="9600">0440</deviceid>
        <deviceid build="9600">10C6</deviceid>
        <deviceid build="9600">10C7</deviceid>
        <deviceid build="9600">10C8</deviceid>
        <deviceid build="9600">150B</deviceid>
        <deviceid build="9600">10DB</deviceid>
        <deviceid build="9600">10DD</deviceid>
        <deviceid build="9600">10EC</deviceid>
        <deviceid build="9600">10F1</deviceid>
        <deviceid build="9600">10E1</deviceid>
        <deviceid build="9600">10F4</deviceid>
        <deviceid build="9600">10F7</deviceid>
        <deviceid build="9600">1514</deviceid>
        <deviceid build="9600">1517</deviceid>
        <deviceid build="9600">10F8</deviceid>
        <deviceid build="15063">000C</deviceid>
        <deviceid build="15063">1F63</deviceid>
        <deviceid build="9600">10F9</deviceid>
        <deviceid build="9600">10FB</deviceid>
        <deviceid build="15063">11A9</deviceid>
        <deviceid build="15063">1071</deviceid>
        <deviceid build="15063">1F72</deviceid>
        <deviceid build="15063">17D0</deviceid>
        <deviceid build="15063">0470</deviceid>
        <deviceid build="15063">211B</deviceid>
        <deviceid build="15063">8976</deviceid>
        <deviceid build="15063">2159</deviceid>
        <deviceid build="15063">000D</deviceid>
        <deviceid build="15063">0008</deviceid>
        <deviceid build="9600">152A</deviceid>
        <deviceid build="9600">1529</deviceid>
        <deviceid build="9600">1507</deviceid>
        <deviceid build="9600">154D</deviceid>
        <deviceid build="9600">154A</deviceid>
```

```
        <deviceid build="9600">1558</deviceid>
        <deviceid build="9600">1557</deviceid>
        <deviceid build="15063">0001</deviceid>
        <deviceid build="9600">10FC</deviceid>
        <deviceid build="9600">151C</deviceid>
        <deviceid build="9600">1528</deviceid>
        <deviceid build="9600">1560</deviceid>
        <deviceid build="15063">1563</deviceid>
        <deviceid build="15063">ABCD</deviceid>
        <deviceid build="15063">15AA</deviceid>
        <deviceid build="15063">15AB</deviceid>
        <deviceid build="15063">15AC</deviceid>
        <deviceid build="15063">15AD</deviceid>
        <deviceid build="15063">15AE</deviceid>
        <deviceid build="19565">0D4E</deviceid>
        <deviceid build="19565">0D4F</deviceid>
        <deviceid build="19565">0D4C</deviceid>
        <deviceid build="19565">0D4D</deviceid>
        <deviceid build="19565">0D53</deviceid>
        <deviceid build="19565">0D55</deviceid>
        <deviceid build="19565">15FB</deviceid>
        <deviceid build="19565">15FC</deviceid>
        <deviceid build="19565">15F9</deviceid>
        <deviceid build="19565">15FA</deviceid>
        <deviceid build="19565">15F4</deviceid>
        <deviceid build="20271">15F5</deviceid>
        <deviceid build="20271">1A1E</deviceid>
        <deviceid build="20271">1A1F</deviceid>
        <deviceid build="20271">1A1C</deviceid>
        <deviceid build="20271">1A1D</deviceid>
        <deviceid build="20271">550A</deviceid>
        <deviceid build="20271">550B</deviceid>
        <deviceid build="20271">550C</deviceid>
        <deviceid build="20271">550D</deviceid>
        <deviceid build="20271">1572</deviceid>
        <deviceid build="20271">1574</deviceid>
        <deviceid build="20271">1580</deviceid>
        <deviceid build="20271">1581</deviceid>
        <deviceid build="20271">1583</deviceid>
        <deviceid build="20271">1584</deviceid>
        <deviceid build="20271">1585</deviceid>
        <deviceid build="20271">1586</deviceid>
        <deviceid build="20271">1587</deviceid>
        <deviceid build="20271">1588</deviceid>
        <deviceid build="20271">1589</deviceid>
        <deviceid build="20271">158A</deviceid>
        <deviceid build="20271">158B</deviceid>
        <deviceid build="20271">37D0</deviceid>
        <deviceid build="20271">37CC</deviceid>
        <deviceid build="20271">37D2</deviceid>
        <deviceid build="20271">37CE</deviceid>
        <deviceid build="20271">37CF</deviceid>
        <deviceid build="20271">37D0</deviceid>
        <deviceid build="20271">37D1</deviceid>
        <deviceid build="20271">37D2</deviceid>
```

```
      <deviceid build="20271">37D3</deviceid>
      <deviceid build="20271">37D4</deviceid>
      <deviceid build="22536">550E</deviceid>
      <deviceid build="22536">550F</deviceid>
      <deviceid build="22536">5510</deviceid>
      <deviceid build="22536">5511</deviceid>
      <deviceid build="22536">0DC7</deviceid>
      <deviceid build="22536">0DC8</deviceid>
      <deviceid build="22536">0DC5</deviceid>
      <deviceid build="22536">0DC6</deviceid>
      <deviceid build="25215">57A0</deviceid>
      <deviceid build="25215">57A1</deviceid>
      <deviceid build="25215">57B3</deviceid>
      <deviceid build="25215">57B4</deviceid>
      <deviceid build="25215">57B5</deviceid>
      <deviceid build="25215">57B6</deviceid>
      <deviceid build="25215">57B7</deviceid>
      <deviceid build="25215">57B8</deviceid>
      <deviceid build="25250">15F2</deviceid>
      <deviceid build="25250">15F3</deviceid>
      <deviceid build="25250">0D9F</deviceid>
      <deviceid build="25250">3100</deviceid>
      <deviceid build="25250">3101</deviceid>
      <deviceid build="25250">3102</deviceid>
      <deviceid build="25250">5502</deviceid>
      <deviceid build="25250">125B</deviceid>
      <deviceid build="25250">125C</deviceid>
      <deviceid build="25250">125D</deviceid>
      <deviceid build="25250">125E</deviceid>
      <deviceid build="25250">5503</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10EC</manufacturer>
      <deviceid build="9200">8136</deviceid>
      <deviceid build="9200">8137</deviceid>
      <deviceid build="9200">8168</deviceid>
      <deviceid build="9200">8167</deviceid>
      <deviceid build="9200">8169</deviceid>
      <deviceid build="15063">8166</deviceid>
      <deviceid build="18362">8161</deviceid>
      <deviceid build="18362">8125</deviceid>
      <deviceid build="18362">8225</deviceid>
      <deviceid build="18362">2502</deviceid>
      <deviceid build="18362">2600</deviceid>
      <deviceid build="18362">3000</deviceid>
      <deviceid build="25876">8126</deviceid>
      <deviceid build="25876">8162</deviceid>
      <deviceid build="25876">5000</deviceid>
  </NIC>

  <NIC>
    <manufacturer>14E4</manufacturer>
      <deviceid build="9200">1644</deviceid>
      <deviceid build="9200">1645</deviceid>
```

```
        <deviceid build="9200">1646</deviceid>
        <deviceid build="9200">16A6</deviceid>
        <deviceid build="9200">16C6</deviceid>
        <deviceid build="9200">1647</deviceid>
        <deviceid build="9200">16A7</deviceid>
        <deviceid build="9200">16C7</deviceid>
        <deviceid build="9200">164D</deviceid>
        <deviceid build="9200">1648</deviceid>
        <deviceid build="9200">16A8</deviceid>
        <deviceid build="9200">1653</deviceid>
        <deviceid build="9200">166E</deviceid>
        <deviceid build="9200">1654</deviceid>
        <deviceid build="9200">165D</deviceid>
        <deviceid build="9200">165E</deviceid>
        <deviceid build="9200">166D</deviceid>
        <deviceid build="9200">170D</deviceid>
        <deviceid build="9200">170E</deviceid>
        <deviceid build="9200">1696</deviceid>
        <deviceid build="9200">1676</deviceid>
        <deviceid build="9200">1677</deviceid>
        <deviceid build="9200">1659</deviceid>
        <deviceid build="9200">167C</deviceid>
        <deviceid build="9200">167D</deviceid>
        <deviceid build="9200">169D</deviceid>
        <deviceid build="9200">16F7</deviceid>
        <deviceid build="9200">16FD</deviceid>
        <deviceid build="9200">16FE</deviceid>
        <deviceid build="9200">16DD</deviceid>
        <deviceid build="9200">1668</deviceid>
        <deviceid build="9200">1669</deviceid>
        <deviceid build="9200">1678</deviceid>
        <deviceid build="9200">1679</deviceid>
        <deviceid build="9200">1600</deviceid>
        <deviceid build="9200">1601</deviceid>
        <deviceid build="9200">166A</deviceid>
        <deviceid build="9200">166B</deviceid>
        <deviceid build="9200">16FF</deviceid>
        <deviceid build="9200">170F</deviceid>
        <deviceid build="9200">169B</deviceid>
        <deviceid build="9200">1693</deviceid>
        <deviceid build="9200">167F</deviceid>
        <deviceid build="9200">169A</deviceid>
        <deviceid build="9200">1698</deviceid>
        <deviceid build="9200">1692</deviceid>
        <deviceid build="9200">1694</deviceid>
        <deviceid build="9200">1690</deviceid>
        <deviceid build="9200">1691</deviceid>
        <deviceid build="9200">1699</deviceid>
        <deviceid build="9200">16A0</deviceid>
        <deviceid build="9200">167B</deviceid>
        <deviceid build="9200">1673</deviceid>
        <deviceid build="9200">165A</deviceid>
        <deviceid build="9200">1674</deviceid>
        <deviceid build="9200">1681</deviceid>
        <deviceid build="9200">1680</deviceid>
```

```
        <deviceid build="9200">1688</deviceid>
        <deviceid build="9200">167A</deviceid>
        <deviceid build="9200">1672</deviceid>
        <deviceid build="9200">1684</deviceid>
        <deviceid build="9200">165B</deviceid>
        <deviceid build="9200">16B1</deviceid>
        <deviceid build="9200">16B5</deviceid>
        <deviceid build="9200">16B0</deviceid>
        <deviceid build="9200">16B4</deviceid>
        <deviceid build="9200">16B2</deviceid>
        <deviceid build="9200">16B6</deviceid>
        <deviceid build="15063">1682</deviceid>
        <deviceid build="15063">1686</deviceid>
        <deviceid build="15063">16B3</deviceid>
        <deviceid build="15063">16B7</deviceid>
        <deviceid build="9200">1655</deviceid>
        <deviceid build="15063">1665</deviceid>
        <deviceid build="9200">1656</deviceid>
        <deviceid build="9200">1657</deviceid>
        <deviceid build="9200">165F</deviceid>
        <deviceid build="9200">165C</deviceid>
        <deviceid build="15063">1683</deviceid>
        <deviceid build="15063">1641</deviceid>
        <deviceid build="15063">1642</deviceid>
        <deviceid build="15063">1643</deviceid>
        <deviceid build="15063">1687</deviceid>
        <deviceid build="9200">164A</deviceid>
        <deviceid build="9200">16AA</deviceid>
        <deviceid build="9200">164C</deviceid>
        <deviceid build="9200">16AC</deviceid>
        <deviceid build="9200">1639</deviceid>
        <deviceid build="9200">163A</deviceid>
        <deviceid build="9200">163B</deviceid>
        <deviceid build="9200">163C</deviceid>
        <deviceid build="15063">164E</deviceid>
        <deviceid build="15063">164F</deviceid>
        <deviceid build="15063">1650</deviceid>
        <deviceid build="15063">1662</deviceid>
        <deviceid build="15063">1663</deviceid>
        <deviceid build="15063">168A</deviceid>
        <deviceid build="15063">168D</deviceid>
        <deviceid build="15063">16A1</deviceid>
        <deviceid build="15063">16A2</deviceid>
        <deviceid build="15063">168E</deviceid>
        <deviceid build="15063">163D</deviceid>
        <deviceid build="15063">16A5</deviceid>
        <deviceid build="15063">16A4</deviceid>
        <deviceid build="15063">16AE</deviceid>
        <deviceid build="15063">163E</deviceid>
        <deviceid build="18362">16C0</deviceid>
        <deviceid build="18362">16C9</deviceid>
        <deviceid build="18362">16CA</deviceid>
        <deviceid build="18362">16CC</deviceid>
        <deviceid build="18362">16CD</deviceid>
        <deviceid build="18362">16CE</deviceid>
```

```
        <deviceid build="18362">16CF</deviceid>
        <deviceid build="18362">16D0</deviceid>
        <deviceid build="18362">16D1</deviceid>
        <deviceid build="18362">16D2</deviceid>
        <deviceid build="18362">16D4</deviceid>
        <deviceid build="18362">16D5</deviceid>
        <deviceid build="18362">16D6</deviceid>
        <deviceid build="18362">16D7</deviceid>
        <deviceid build="18362">16D8</deviceid>
        <deviceid build="18362">16D9</deviceid>
        <deviceid build="18362">16DA</deviceid>
        <deviceid build="18362">16DB</deviceid>
        <deviceid build="18362">16DC</deviceid>
        <deviceid build="18362">16DE</deviceid>
        <deviceid build="18362">16DF</deviceid>
        <deviceid build="18362">16E0</deviceid>
        <deviceid build="18362">16E2</deviceid>
        <deviceid build="18362">16E3</deviceid>
        <deviceid build="18362">16E4</deviceid>
        <deviceid build="18362">16E7</deviceid>
        <deviceid build="18362">16E8</deviceid>
        <deviceid build="18362">16E9</deviceid>
        <deviceid build="18362">16EA</deviceid>
        <deviceid build="18362">16EB</deviceid>
        <deviceid build="18362">16EC</deviceid>
        <deviceid build="18362">16ED</deviceid>
        <deviceid build="18362">16EE</deviceid>
        <deviceid build="18362">16EF</deviceid>
        <deviceid build="18362">16F0</deviceid>
        <deviceid build="18362">16F1</deviceid>
        <deviceid build="18362">1614</deviceid>
        <deviceid build="18362">D802</deviceid>
        <deviceid build="20241">1604</deviceid>
        <deviceid build="20241">1605</deviceid>
        <deviceid build="20241">1606</deviceid>
        <deviceid build="20241">1607</deviceid>
        <deviceid build="20241">1608</deviceid>
        <deviceid build="20241">1609</deviceid>
        <deviceid build="20241">D804</deviceid>
        <deviceid build="20241">D812</deviceid>
        <deviceid build="20241">D814</deviceid>
        <deviceid build="20241">D818</deviceid>
        <deviceid build="20241">D82A</deviceid>
        <deviceid build="20241">D82B</deviceid>
        <deviceid build="20241">D82C</deviceid>
        <deviceid build="20241">D82D</deviceid>
        <deviceid build="20241">D82E</deviceid>
        <deviceid build="20241">D82F</deviceid>
        <deviceid build="20241">1902</deviceid>
        <deviceid build="20241">1903</deviceid>
        <deviceid build="20241">1906</deviceid>
        <deviceid build="20241">1907</deviceid>
        <deviceid build="20241">190A</deviceid>
        <deviceid build="20241">190B</deviceid>
        <deviceid build="20241">1799</deviceid>
```

```
      <deviceid build="20241">1041</deviceid>
      <deviceid build="20241">1042</deviceid>
      <deviceid build="20241">1043</deviceid>
      <deviceid build="20241">1750</deviceid>
      <deviceid build="20241">1751</deviceid>
      <deviceid build="20241">1752</deviceid>
      <deviceid build="20241">1800</deviceid>
      <deviceid build="20241">1801</deviceid>
      <deviceid build="20241">1802</deviceid>
      <deviceid build="20241">1803</deviceid>
      <deviceid build="20241">1804</deviceid>
      <deviceid build="20241">1805</deviceid>
      <deviceid build="20241">1806</deviceid>
      <deviceid build="20241">1807</deviceid>
      <deviceid build="20241">1808</deviceid>
      <deviceid build="20241">1809</deviceid>
      <deviceid build="20241">16C1</deviceid>
      <deviceid build="20241">16C5</deviceid>
      <deviceid build="20241">16BD</deviceid>
  </NIC>

  <NIC>
    <manufacturer>1969</manufacturer>
      <deviceid build="9200">1062</deviceid>
      <deviceid build="9200">1063</deviceid>
      <deviceid build="9200">2060</deviceid>
      <deviceid build="9200">2062</deviceid>
      <deviceid build="9200">1073</deviceid>
      <deviceid build="9200">1083</deviceid>
      <deviceid build="9200">1090</deviceid>
      <deviceid build="9200">1091</deviceid>
      <deviceid build="9200">E091</deviceid>
      <deviceid build="9200">10A0</deviceid>
      <deviceid build="9200">10A1</deviceid>
      <deviceid build="9200">E0A1</deviceid>
      <deviceid build="9200">10B0</deviceid>
      <deviceid build="9200">10B1</deviceid>
      <deviceid build="9200">E0B1</deviceid>
      <deviceid build="9200">10C0</deviceid>
      <deviceid build="9200">10C1</deviceid>
      <deviceid build="9200">E0C1</deviceid>
      <deviceid build="9200">10D0</deviceid>
      <deviceid build="9200">10D1</deviceid>
      <deviceid build="9200">E0D1</deviceid>
      <deviceid build="9200">10E0</deviceid>
      <deviceid build="9200">10E1</deviceid>
      <deviceid build="9200">E0E1</deviceid>
      <deviceid build="9200">10F0</deviceid>
      <deviceid build="9200">10F1</deviceid>
      <deviceid build="9200">E0F1</deviceid>
  </NIC>

  <NIC>
    <manufacturer>19A2</manufacturer>
      <deviceid build="9600">0211</deviceid>
```

```
      <deviceid build="9600">0215</deviceid>
      <deviceid build="9600">0221</deviceid>
      <deviceid build="9600">0700</deviceid>
      <deviceid build="9600">0710</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10DF</manufacturer>
      <deviceid build="9600">0720</deviceid>
      <deviceid build="9600">E220</deviceid>
  </NIC>

  <NIC>
    <manufacturer>15B3</manufacturer>
      <deviceid build="9600">1000</deviceid>
      <deviceid build="9600">1001</deviceid>
      <deviceid build="9600">1002</deviceid>
      <deviceid build="9600">1003</deviceid>
      <deviceid build="9600">1004</deviceid>
      <deviceid build="9600">1005</deviceid>
      <deviceid build="9600">1006</deviceid>
      <deviceid build="9600">1007</deviceid>
      <deviceid build="9600">1008</deviceid>
      <deviceid build="9600">1009</deviceid>
      <deviceid build="9600">100A</deviceid>
      <deviceid build="9600">100B</deviceid>
      <deviceid build="9600">100C</deviceid>
      <deviceid build="9600">100D</deviceid>
      <deviceid build="9600">100E</deviceid>
      <deviceid build="9600">100F</deviceid>
      <deviceid build="9600">1010</deviceid>
      <deviceid build="9600">6340</deviceid>
      <deviceid build="9600">6341</deviceid>
      <deviceid build="9600">634A</deviceid>
      <deviceid build="9600">634B</deviceid>
      <deviceid build="9600">6354</deviceid>
      <deviceid build="9600">6368</deviceid>
      <deviceid build="9600">6369</deviceid>
      <deviceid build="9600">6372</deviceid>
      <deviceid build="9600">6732</deviceid>
      <deviceid build="9600">6733</deviceid>
      <deviceid build="9600">673C</deviceid>
      <deviceid build="9600">673D</deviceid>
      <deviceid build="9600">6746</deviceid>
      <deviceid build="9600">6750</deviceid>
      <deviceid build="9600">6751</deviceid>
      <deviceid build="9600">675A</deviceid>
      <deviceid build="9600">6764</deviceid>
      <deviceid build="9600">6765</deviceid>
      <deviceid build="9600">676E</deviceid>
      <deviceid build="9600">6778</deviceid>
      <deviceid build="15063">1013</deviceid>
      <deviceid build="15063">1015</deviceid>
      <deviceid build="15063">1017</deviceid>
      <deviceid build="15063">1019</deviceid>
```

```
        <deviceid build="15063">101B</deviceid>
        <deviceid build="15063">101D</deviceid>
        <deviceid build="15063">101F</deviceid>
        <deviceid build="15063">1021</deviceid>
        <deviceid build="15063">1023</deviceid>
        <deviceid build="15063">1025</deviceid>
        <deviceid build="15063">1027</deviceid>
        <deviceid build="15063">1029</deviceid>
        <deviceid build="15063">102B</deviceid>
        <deviceid build="15063">102F</deviceid>
    </NIC>

    <NIC>
      <manufacturer>1137</manufacturer>
        <deviceid build="15063">0043</deviceid>
    </NIC>

    <NIC>
      <manufacturer>1414</manufacturer>
        <deviceid build="25500">00B9</deviceid>
    </NIC>

</SupportedNetworkInterfaceCards>
```

另请参阅

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

Windows 11 21H2 中网络内核调试支持的以太网 NIC

Windows 10 中网络内核调试支持的以太网 NIC

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Windows 11 21H2 中支持用于网络内核调试的以太网 NIC

项目 • 2023/11/17

若要通过以太网网线进行内核调试，目标计算机必须具有受支持的网络接口卡 (NIC) 。

在内核调试期间，运行调试器的计算机称为 *主计算机*，正在调试的计算机称为 *目标计算机*。 有关详细信息，请参阅 设置 KDNET 网络内核自动调试。

若要通过网络电缆进行内核调试，目标计算机必须具有受支持的网络适配器。 当目标计算机运行 Windows 时，支持此处列出的网络适配器进行内核调试。

版本信息

本主题列出了以下版本的 Windows 支持的适配器

- Windows 11,21H2 内部版本 22000

有关Windows 11版本的信息，请参阅Windows 11版本信息。

使用 VerifiedNicList.xml确定 NIC 支持

若要确定任何特定版本的 Windows 支持哪一组 NIC，请检查 `VerifiedNicList.xml` WDK 或 SDK 所安装的调试器目录中的文件（随该特定版本的 Windows 一起提供）。 对于 64 位 Windows，默认情况下，它将安装在此目录中：

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\VerifiedNicList.xml
```

检查特定版本附带的 VerifiedNicList.xml 是必需的，因为向以前版本中不存在的新版本的 Windows 添加了其他硬件支持。 因此，必须检查该特定版本的 VerifiedNicLIst.xml 文件。

查找供应商 ID 和设备 ID

在目标计算机上查找网络适配器的供应商 ID 和设备 ID。

- 在目标计算机上，打开设备管理器 (命令提示符窗口中输入 **devmgmt**) 。
- 在“设备管理器”中，找到要用于调试的网络适配器。
- 选择并按住 (或右键单击) 网络适配器节点，然后选择“**属性**”。
- 在“ **详细信息** ”选项卡的 “**属性**”下，选择“ **硬件 ID**”。

供应商和设备 ID 显示为 VEN_*VendorID* 和 DEV_*DeviceID*。 例如，如果看到 PCI\VEN_8086&DEV_104B，则供应商 ID 为 8086，设备 ID 为 104B。

供应商 ID 8086，Intel Corporation

对于供应商 ID 8086，支持以下设备 ID：

0001 0008 000C 000D 0438 043A 043C 0440 0470 0D4E 0D4F 0D4C 0D4D 0D53 0D55 1000 1001 1004 1004 100 8 1009 100C 100D 100E 100F 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 101A 101D 101E 1026 1027 1028 1049 104A 104B 104C 104D 105E 105F 1060 1071 1075 1076 1077 1078 1079 107A 107B 107C 107D 107E 107F 108A 108B 108C 1096 1098 1099 109A 10A4 10A5 10A7 10A9 10B5 10B9 10BA 10BB10BC 10BD 10BF 10C0 10C2 10C3 10C4 10C5 10C6 10C7 10C8 10C9 10CB 10CC 10CD 10CE 10D3 10D5 10D6 10D9 10DA 10DB 10DD 10DE 10DF 10E1 10E5 10E6 10E7 10E8 10EA 10EB 10EC 10EF 10F0 10F1 10F4 10F5 10F6 10F7 10F8 10F9 10FB 10FC 11A9 1501 1502 1503 1507 150A 150B 150C 150D 150E 150F 1510 1511 1514 1516 1517 1518 151C 1521 1522 1523 1524 1525 1526 1527 1528 1529 152A 1533 1534 15351536 1537 1538 1539 153A 153B 1546 154A 154D 1557 1558 1559 155A 1560 1563 156F 1570 157B 157C 15A0 15A1 15A2 15A3 15AA 15AB 15AC 15AD 15AE 15B7 15B8 15B9 15BB 15BC 15BD 15BE 15D6 15D7 15D8 15DF 15E0 15E1 15E2 15E3 15F 4 15F9 15FA 15FB 15FC 17D0 1F40 1F41 1F45 1F63 1F72 211B 2159 294C 8976

Windows 11 21H2 内部版本 22000 中的新增功能

1572 1574 1580 1581 1583 1584 1585 1586 1587 1588 1589 15F5 1A1E 1A1F 1A1C 1A1D 158A 158B 37D0 37CC 37D2 37CE 37CF 37D0 37D1 37D2 37D3 37D4 550A 550B 550C 550D

供应商 ID 10EC，Realtek Semiconductor Corp.

对于供应商 ID 10EC，支持以下设备 ID：

2502 2600 3000 8125 8136 8137 8161 8166 8167 8168 8169 8225

供应商 ID 14E4，Broadcom

对于供应商 ID 14E4，支持以下设备 ID：

1600 1601 1614 1639 163A 163B 163C 163D 163E 1641 1642 1643 1644 1645 1646 1647 1648 164A 164C 164D 164E 164F 1650 1653 1654 1655 1656 1657 1659 165A 165B 165C 165D 165E 165F 1662 1663 16 65 1668 1669 166A 166B 166D 166E 1672 1673 1674 1676 1677 1678 1679 1679 167A 167B 167C 167D 167F 1 68A 168D 168E 1680

1681 1682 1683 1684 1686 1687 1688 1690 1691 1692 1693 1694 1696 1698 1699169A 169B 169D 16A0 16A1 16A2 16A4 16A5 16A6 16A7 16A8 16AA 16AC 1 6AE 16B0 16B1 16B2 16B3 16B4 16B5 16B6 16B7 16C6 16C7 16C9 16CA 1 6CE 16CF 16D0 16D1 16D2 16D5 16D6 16D7 16D8 16D9 16DD 16DF 16E0 16E2 16E3 16E4 16E9 16F0 16F1 16F7 16FD 16FE 16FF 170D 170E 170F D802

Windows 11 21H2 内部版本 22000 中的新增功能

1041 1042 1043 1604 1605 1606 1607 1608 1609 16C1 16C5 16BD 1750 1751 1752 1799 1800 1801 1802 1803 1804 1804 1805 1806 1807 1808 1809 1902 1903 1906 1907 190A 190B D804 D812 D814 D818 D82A D82B D82C D82D D82E D82F

供应商 ID 1969，Atheros Communications

对于供应商 ID 1969，支持以下设备 ID：

1062 1063 1073 1083 1090 1091 10A0 10A1 10B0 10B1 10C0 10C1 10D0 10D1 10E0 10E1 10F0 10F1 2060 2062 E091 E0A1 E0B1 E0C1 E0D1 E0E1 E0F1

供应商 ID 19A2，ServerEngines (Emulex)

对于供应商 ID 19A2，支持以下设备 ID：

0211 0215 0221 0700 0710

供应商 ID 10DF，Emulex Corporation

对于供应商 ID 10DF，支持以下设备 ID：

0720 E220

供应商 ID 15B3，Mellanox Technology

对于供应商 ID 15B3，支持以下设备 ID：

1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 100A 100B 100C 100D 100E 100F 1010 1013 1015 1017 1019 101B 101D 101F 1021 10 23 1025 1027 1029 102B 102F 6340 6341 634A 634B 6354 6368 6369 6 372 6732 6733 673C 673D 6746 6750 6751 675A 6764 6765 676E 6778

供应商 ID 1137，Cisco Systems Inc

对于供应商 ID 1137，支持以下设备 ID：

0043

XML 支持的 NIC 列表

这与上面以 XML 格式显示的信息相同。

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<SupportedNetworkInterfaceCards>
  <NIC>
    <manufacturer>8086</manufacturer>
      <deviceid build="9200">1000</deviceid>
      <deviceid build="9200">1001</deviceid>
      <deviceid build="9200">1004</deviceid>
      <deviceid build="9200">1008</deviceid>
      <deviceid build="9200">1009</deviceid>
      <deviceid build="9200">100C</deviceid>
      <deviceid build="9200">100D</deviceid>
      <deviceid build="9200">100E</deviceid>
      <deviceid build="9200">1015</deviceid>
      <deviceid build="9200">1016</deviceid>
      <deviceid build="9200">1017</deviceid>
      <deviceid build="9200">101E</deviceid>
      <deviceid build="9200">100F</deviceid>
      <deviceid build="9200">1011</deviceid>
      <deviceid build="9200">1026</deviceid>
      <deviceid build="9200">1027</deviceid>
      <deviceid build="9200">1028</deviceid>
      <deviceid build="9200">1010</deviceid>
      <deviceid build="9200">1012</deviceid>
      <deviceid build="9200">101D</deviceid>
      <deviceid build="9200">1079</deviceid>
      <deviceid build="9200">107A</deviceid>
      <deviceid build="9200">107B</deviceid>
      <deviceid build="9200">108A</deviceid>
      <deviceid build="9200">1099</deviceid>
      <deviceid build="9200">10B5</deviceid>
      <deviceid build="9200">1013</deviceid>
      <deviceid build="9200">1018</deviceid>
      <deviceid build="9200">1014</deviceid>
      <deviceid build="9200">1078</deviceid>
      <deviceid build="9200">1076</deviceid>
      <deviceid build="9200">107C</deviceid>
      <deviceid build="9200">1077</deviceid>
      <deviceid build="9200">1019</deviceid>
      <deviceid build="9200">101A</deviceid>
      <deviceid build="9200">1075</deviceid>
      <deviceid build="9200">105E</deviceid>
      <deviceid build="9200">105F</deviceid>
```

```
        <deviceid build="9200">1060</deviceid>
        <deviceid build="9200">10D9</deviceid>
        <deviceid build="9200">10DA</deviceid>
        <deviceid build="9200">10A4</deviceid>
        <deviceid build="9200">10D5</deviceid>
        <deviceid build="9200">10A5</deviceid>
        <deviceid build="9600">10BC</deviceid>
        <deviceid build="9200">107D</deviceid>
        <deviceid build="9200">107E</deviceid>
        <deviceid build="9200">107F</deviceid>
        <deviceid build="9600">10B9</deviceid>
        <deviceid build="9200">108B</deviceid>
        <deviceid build="9200">108C</deviceid>
        <deviceid build="9200">109A</deviceid>
        <deviceid build="9600">10D3</deviceid>
        <deviceid build="9200">10F6</deviceid>
        <deviceid build="9200">150C</deviceid>
        <deviceid build="9200">1096</deviceid>
        <deviceid build="9200">1098</deviceid>
        <deviceid build="9600">10BA</deviceid>
        <deviceid build="9600">10BB</deviceid>
        <deviceid build="9200">1501</deviceid>
        <deviceid build="9200">1049</deviceid>
        <deviceid build="9200">104A</deviceid>
        <deviceid build="9200">104B</deviceid>
        <deviceid build="9200">104C</deviceid>
        <deviceid build="9600">10C4</deviceid>
        <deviceid build="9600">10C5</deviceid>
        <deviceid build="9200">104D</deviceid>
        <deviceid build="9600">10BF</deviceid>
        <deviceid build="9200">10F5</deviceid>
        <deviceid build="9600">10CB</deviceid>
        <deviceid build="9600">10BD</deviceid>
        <deviceid build="9200">10E5</deviceid>
        <deviceid build="9200">294C</deviceid>
        <deviceid build="9600">10C0</deviceid>
        <deviceid build="9600">10C3</deviceid>
        <deviceid build="9600">10C2</deviceid>
        <deviceid build="9600">10CC</deviceid>
        <deviceid build="9600">10CD</deviceid>
        <deviceid build="9600">10CE</deviceid>
        <deviceid build="9600">10DE</deviceid>
        <deviceid build="9600">10DF</deviceid>
        <deviceid build="9600">1525</deviceid>
        <deviceid build="9200">10EA</deviceid>
        <deviceid build="9200">10EB</deviceid>
        <deviceid build="9200">10EF</deviceid>
        <deviceid build="9200">10F0</deviceid>
        <deviceid build="9200">1502</deviceid>
        <deviceid build="9200">1503</deviceid>
        <deviceid build="9600">153A</deviceid>
        <deviceid build="9600">153B</deviceid>
        <deviceid build="9600">155A</deviceid>
        <deviceid build="9600">1559</deviceid>
        <deviceid build="15063">15A0</deviceid>
```

```
        <deviceid build="15063">15A1</deviceid>
        <deviceid build="15063">15A2</deviceid>
        <deviceid build="15063">15A3</deviceid>
        <deviceid build="15063">156F</deviceid>
        <deviceid build="15063">1570</deviceid>
        <deviceid build="15063">15B7</deviceid>
        <deviceid build="15063">15B8</deviceid>
        <deviceid build="16299">15B9</deviceid>
        <deviceid build="16299">15D7</deviceid>
        <deviceid build="16299">15D8</deviceid>
        <deviceid build="16299">15E3</deviceid>
        <deviceid build="16299">15D6</deviceid>
        <deviceid build="16299">15BD</deviceid>
        <deviceid build="16299">15BE</deviceid>
        <deviceid build="16299">15BB</deviceid>
        <deviceid build="16299">15BC</deviceid>
        <deviceid build="16299">15DF</deviceid>
        <deviceid build="16299">15E0</deviceid>
        <deviceid build="16299">15E1</deviceid>
        <deviceid build="16299">15E2</deviceid>
        <deviceid build="9600">10C9</deviceid>
        <deviceid build="9200">10E6</deviceid>
        <deviceid build="9200">10E7</deviceid>
        <deviceid build="9200">10E8</deviceid>
        <deviceid build="9200">1526</deviceid>
        <deviceid build="9200">150A</deviceid>
        <deviceid build="9200">1518</deviceid>
        <deviceid build="9200">150D</deviceid>
        <deviceid build="9200">10A7</deviceid>
        <deviceid build="9200">10A9</deviceid>
        <deviceid build="9600">10D6</deviceid>
        <deviceid build="9200">150E</deviceid>
        <deviceid build="9200">150F</deviceid>
        <deviceid build="9200">1510</deviceid>
        <deviceid build="9200">1511</deviceid>
        <deviceid build="9200">1516</deviceid>
        <deviceid build="9600">1527</deviceid>
        <deviceid build="9200">1521</deviceid>
        <deviceid build="9200">1522</deviceid>
        <deviceid build="9200">1523</deviceid>
        <deviceid build="9200">1524</deviceid>
        <deviceid build="9600">1546</deviceid>
        <deviceid build="9600">1533</deviceid>
        <deviceid build="9600">1534</deviceid>
        <deviceid build="9600">1535</deviceid>
        <deviceid build="9600">1536</deviceid>
        <deviceid build="9600">1537</deviceid>
        <deviceid build="9600">1538</deviceid>
        <deviceid build="9600">157B</deviceid>
        <deviceid build="9600">157C</deviceid>
        <deviceid build="9600">1539</deviceid>
        <deviceid build="9600">1F40</deviceid>
        <deviceid build="9600">1F41</deviceid>
        <deviceid build="9600">1F45</deviceid>
        <deviceid build="9600">0438</deviceid>
```

```
        <deviceid build="9600">043A</deviceid>
        <deviceid build="9600">043C</deviceid>
        <deviceid build="9600">0440</deviceid>
        <deviceid build="9600">10C6</deviceid>
        <deviceid build="9600">10C7</deviceid>
        <deviceid build="9600">10C8</deviceid>
        <deviceid build="9600">150B</deviceid>
        <deviceid build="9600">10DB</deviceid>
        <deviceid build="9600">10DD</deviceid>
        <deviceid build="9600">10EC</deviceid>
        <deviceid build="9600">10F1</deviceid>
        <deviceid build="9600">10E1</deviceid>
        <deviceid build="9600">10F4</deviceid>
        <deviceid build="9600">10F7</deviceid>
        <deviceid build="9600">1514</deviceid>
        <deviceid build="9600">1517</deviceid>
        <deviceid build="9600">10F8</deviceid>
        <deviceid build="15063">000C</deviceid>
        <deviceid build="15063">1F63</deviceid>
        <deviceid build="9600">10F9</deviceid>
        <deviceid build="9600">10FB</deviceid>
        <deviceid build="15063">11A9</deviceid>
        <deviceid build="15063">1071</deviceid>
        <deviceid build="15063">1F72</deviceid>
        <deviceid build="15063">17D0</deviceid>
        <deviceid build="15063">0470</deviceid>
        <deviceid build="15063">211B</deviceid>
        <deviceid build="15063">8976</deviceid>
        <deviceid build="15063">2159</deviceid>
        <deviceid build="15063">000D</deviceid>
        <deviceid build="15063">0008</deviceid>
        <deviceid build="9600">152A</deviceid>
        <deviceid build="9600">1529</deviceid>
        <deviceid build="9600">1507</deviceid>
        <deviceid build="9600">154D</deviceid>
        <deviceid build="9600">154A</deviceid>
        <deviceid build="9600">1558</deviceid>
        <deviceid build="9600">1557</deviceid>
        <deviceid build="15063">0001</deviceid>
        <deviceid build="9600">10FC</deviceid>
        <deviceid build="9600">151C</deviceid>
        <deviceid build="9600">1528</deviceid>
        <deviceid build="9600">1560</deviceid>
        <deviceid build="15063">1563</deviceid>
        <deviceid build="15063">ABCD</deviceid>
        <deviceid build="15063">15AA</deviceid>
        <deviceid build="15063">15AB</deviceid>
        <deviceid build="15063">15AC</deviceid>
        <deviceid build="15063">15AD</deviceid>
        <deviceid build="15063">15AE</deviceid>
        <deviceid build="19565">0D4E</deviceid>
        <deviceid build="19565">0D4F</deviceid>
        <deviceid build="19565">0D4C</deviceid>
        <deviceid build="19565">0D4D</deviceid>
        <deviceid build="19565">0D53</deviceid>
```

```
      <deviceid build="19565">0D55</deviceid>
      <deviceid build="19565">15FB</deviceid>
      <deviceid build="19565">15FC</deviceid>
      <deviceid build="19565">15F9</deviceid>
      <deviceid build="19565">15FA</deviceid>
      <deviceid build="19565">15F4</deviceid>
      <deviceid build="20271">15F5</deviceid>
      <deviceid build="20271">1A1E</deviceid>
      <deviceid build="20271">1A1F</deviceid>
      <deviceid build="20271">1A1C</deviceid>
      <deviceid build="20271">1A1D</deviceid>
      <deviceid build="20271">550A</deviceid>
      <deviceid build="20271">550B</deviceid>
      <deviceid build="20271">550C</deviceid>
      <deviceid build="20271">550D</deviceid>
      <deviceid build="20271">1572</deviceid>
      <deviceid build="20271">1574</deviceid>
      <deviceid build="20271">1580</deviceid>
      <deviceid build="20271">1581</deviceid>
      <deviceid build="20271">1583</deviceid>
      <deviceid build="20271">1584</deviceid>
      <deviceid build="20271">1585</deviceid>
      <deviceid build="20271">1586</deviceid>
      <deviceid build="20271">1587</deviceid>
      <deviceid build="20271">1588</deviceid>
      <deviceid build="20271">1589</deviceid>
      <deviceid build="20271">158A</deviceid>
      <deviceid build="20271">158B</deviceid>
      <deviceid build="20271">37D0</deviceid>
      <deviceid build="20271">37CC</deviceid>
      <deviceid build="20271">37D2</deviceid>
      <deviceid build="20271">37CE</deviceid>
      <deviceid build="20271">37CF</deviceid>
      <deviceid build="20271">37D0</deviceid>
      <deviceid build="20271">37D1</deviceid>
      <deviceid build="20271">37D2</deviceid>
      <deviceid build="20271">37D3</deviceid>
      <deviceid build="20271">37D4</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10EC</manufacturer>
      <deviceid build="9200">8136</deviceid>
      <deviceid build="9200">8137</deviceid>
      <deviceid build="9200">8168</deviceid>
      <deviceid build="9200">8167</deviceid>
      <deviceid build="9200">8169</deviceid>
      <deviceid build="15063">8166</deviceid>
      <deviceid build="18362">8161</deviceid>
      <deviceid build="18362">8125</deviceid>
      <deviceid build="18362">8225</deviceid>
      <deviceid build="18362">2502</deviceid>
      <deviceid build="18362">2600</deviceid>
      <deviceid build="18362">3000</deviceid>
  </NIC>
```

```
<NIC>
  <manufacturer>14E4</manufacturer>
    <deviceid build="9200">1644</deviceid>
    <deviceid build="9200">1645</deviceid>
    <deviceid build="9200">1646</deviceid>
    <deviceid build="9200">16A6</deviceid>
    <deviceid build="9200">16C6</deviceid>
    <deviceid build="9200">1647</deviceid>
    <deviceid build="9200">16A7</deviceid>
    <deviceid build="9200">16C7</deviceid>
    <deviceid build="9200">164D</deviceid>
    <deviceid build="9200">1648</deviceid>
    <deviceid build="9200">16A8</deviceid>
    <deviceid build="9200">1653</deviceid>
    <deviceid build="9200">166E</deviceid>
    <deviceid build="9200">1654</deviceid>
    <deviceid build="9200">165D</deviceid>
    <deviceid build="9200">165E</deviceid>
    <deviceid build="9200">166D</deviceid>
    <deviceid build="9200">170D</deviceid>
    <deviceid build="9200">170E</deviceid>
    <deviceid build="9200">1696</deviceid>
    <deviceid build="9200">1676</deviceid>
    <deviceid build="9200">1677</deviceid>
    <deviceid build="9200">1659</deviceid>
    <deviceid build="9200">167C</deviceid>
    <deviceid build="9200">167D</deviceid>
    <deviceid build="9200">169D</deviceid>
    <deviceid build="9200">16F7</deviceid>
    <deviceid build="9200">16FD</deviceid>
    <deviceid build="9200">16FE</deviceid>
    <deviceid build="9200">16DD</deviceid>
    <deviceid build="9200">1668</deviceid>
    <deviceid build="9200">1669</deviceid>
    <deviceid build="9200">1678</deviceid>
    <deviceid build="9200">1679</deviceid>
    <deviceid build="9200">1600</deviceid>
    <deviceid build="9200">1601</deviceid>
    <deviceid build="9200">166A</deviceid>
    <deviceid build="9200">166B</deviceid>
    <deviceid build="9200">16FF</deviceid>
    <deviceid build="9200">170F</deviceid>
    <deviceid build="9200">169B</deviceid>
    <deviceid build="9200">1693</deviceid>
    <deviceid build="9200">167F</deviceid>
    <deviceid build="9200">169A</deviceid>
    <deviceid build="9200">1698</deviceid>
    <deviceid build="9200">1692</deviceid>
    <deviceid build="9200">1694</deviceid>
    <deviceid build="9200">1690</deviceid>
    <deviceid build="9200">1691</deviceid>
    <deviceid build="9200">1699</deviceid>
    <deviceid build="9200">16A0</deviceid>
    <deviceid build="9200">167B</deviceid>
```

```
        <deviceid build="9200">1673</deviceid>
        <deviceid build="9200">165A</deviceid>
        <deviceid build="9200">1674</deviceid>
        <deviceid build="9200">1681</deviceid>
        <deviceid build="9200">1680</deviceid>
        <deviceid build="9200">1688</deviceid>
        <deviceid build="9200">167A</deviceid>
        <deviceid build="9200">1672</deviceid>
        <deviceid build="9200">1684</deviceid>
        <deviceid build="9200">165B</deviceid>
        <deviceid build="9200">16B1</deviceid>
        <deviceid build="9200">16B5</deviceid>
        <deviceid build="9200">16B0</deviceid>
        <deviceid build="9200">16B4</deviceid>
        <deviceid build="9200">16B2</deviceid>
        <deviceid build="9200">16B6</deviceid>
        <deviceid build="15063">1682</deviceid>
        <deviceid build="15063">1686</deviceid>
        <deviceid build="15063">16B3</deviceid>
        <deviceid build="15063">16B7</deviceid>
        <deviceid build="9200">1655</deviceid>
        <deviceid build="15063">1665</deviceid>
        <deviceid build="9200">1656</deviceid>
        <deviceid build="9200">1657</deviceid>
        <deviceid build="9200">165F</deviceid>
        <deviceid build="9200">165C</deviceid>
        <deviceid build="15063">1683</deviceid>
        <deviceid build="15063">1641</deviceid>
        <deviceid build="15063">1642</deviceid>
        <deviceid build="15063">1643</deviceid>
        <deviceid build="15063">1687</deviceid>
        <deviceid build="9200">164A</deviceid>
        <deviceid build="9200">16AA</deviceid>
        <deviceid build="9200">164C</deviceid>
        <deviceid build="9200">16AC</deviceid>
        <deviceid build="9200">1639</deviceid>
        <deviceid build="9200">163A</deviceid>
        <deviceid build="9200">163B</deviceid>
        <deviceid build="9200">163C</deviceid>
        <deviceid build="15063">164E</deviceid>
        <deviceid build="15063">164F</deviceid>
        <deviceid build="15063">1650</deviceid>
        <deviceid build="15063">1662</deviceid>
        <deviceid build="15063">1663</deviceid>
        <deviceid build="15063">168A</deviceid>
        <deviceid build="15063">168D</deviceid>
        <deviceid build="15063">16A1</deviceid>
        <deviceid build="15063">16A2</deviceid>
        <deviceid build="15063">168E</deviceid>
        <deviceid build="15063">163D</deviceid>
        <deviceid build="15063">16A5</deviceid>
        <deviceid build="15063">16A4</deviceid>
        <deviceid build="15063">16AE</deviceid>
        <deviceid build="15063">163E</deviceid>
        <deviceid build="18362">16C0</deviceid>
```

```
        <deviceid build="18362">16C9</deviceid>
        <deviceid build="18362">16CA</deviceid>
        <deviceid build="18362">16CC</deviceid>
        <deviceid build="18362">16CD</deviceid>
        <deviceid build="18362">16CE</deviceid>
        <deviceid build="18362">16CF</deviceid>
        <deviceid build="18362">16D0</deviceid>
        <deviceid build="18362">16D1</deviceid>
        <deviceid build="18362">16D2</deviceid>
        <deviceid build="18362">16D4</deviceid>
        <deviceid build="18362">16D5</deviceid>
        <deviceid build="18362">16D6</deviceid>
        <deviceid build="18362">16D7</deviceid>
        <deviceid build="18362">16D8</deviceid>
        <deviceid build="18362">16D9</deviceid>
        <deviceid build="18362">16DA</deviceid>
        <deviceid build="18362">16DB</deviceid>
        <deviceid build="18362">16DC</deviceid>
        <deviceid build="18362">16DE</deviceid>
        <deviceid build="18362">16DF</deviceid>
        <deviceid build="18362">16E0</deviceid>
        <deviceid build="18362">16E2</deviceid>
        <deviceid build="18362">16E3</deviceid>
        <deviceid build="18362">16E4</deviceid>
        <deviceid build="18362">16E7</deviceid>
        <deviceid build="18362">16E8</deviceid>
        <deviceid build="18362">16E9</deviceid>
        <deviceid build="18362">16EA</deviceid>
        <deviceid build="18362">16EB</deviceid>
        <deviceid build="18362">16EC</deviceid>
        <deviceid build="18362">16ED</deviceid>
        <deviceid build="18362">16EE</deviceid>
        <deviceid build="18362">16EF</deviceid>
        <deviceid build="18362">16F0</deviceid>
        <deviceid build="18362">16F1</deviceid>
        <deviceid build="18362">1614</deviceid>
        <deviceid build="18362">D802</deviceid>
        <deviceid build="20241">1604</deviceid>
        <deviceid build="20241">1605</deviceid>
        <deviceid build="20241">1606</deviceid>
        <deviceid build="20241">1607</deviceid>
        <deviceid build="20241">1608</deviceid>
        <deviceid build="20241">1609</deviceid>
        <deviceid build="20241">D804</deviceid>
        <deviceid build="20241">D812</deviceid>
        <deviceid build="20241">D814</deviceid>
        <deviceid build="20241">D818</deviceid>
        <deviceid build="20241">D82A</deviceid>
        <deviceid build="20241">D82B</deviceid>
        <deviceid build="20241">D82C</deviceid>
        <deviceid build="20241">D82D</deviceid>
        <deviceid build="20241">D82E</deviceid>
        <deviceid build="20241">D82F</deviceid>
        <deviceid build="20241">1902</deviceid>
        <deviceid build="20241">1903</deviceid>
```

```
      <deviceid build="20241">1906</deviceid>
      <deviceid build="20241">1907</deviceid>
      <deviceid build="20241">190A</deviceid>
      <deviceid build="20241">190B</deviceid>
      <deviceid build="20241">1799</deviceid>
      <deviceid build="20241">1041</deviceid>
      <deviceid build="20241">1042</deviceid>
      <deviceid build="20241">1043</deviceid>
      <deviceid build="20241">1750</deviceid>
      <deviceid build="20241">1751</deviceid>
      <deviceid build="20241">1752</deviceid>
      <deviceid build="20241">1800</deviceid>
      <deviceid build="20241">1801</deviceid>
      <deviceid build="20241">1802</deviceid>
      <deviceid build="20241">1803</deviceid>
      <deviceid build="20241">1804</deviceid>
      <deviceid build="20241">1805</deviceid>
      <deviceid build="20241">1806</deviceid>
      <deviceid build="20241">1807</deviceid>
      <deviceid build="20241">1808</deviceid>
      <deviceid build="20241">1809</deviceid>
      <deviceid build="20241">16C1</deviceid>
      <deviceid build="20241">16C5</deviceid>
      <deviceid build="20241">16BD</deviceid>
  </NIC>

  <NIC>
    <manufacturer>1969</manufacturer>
      <deviceid build="9200">1062</deviceid>
      <deviceid build="9200">1063</deviceid>
      <deviceid build="9200">2060</deviceid>
      <deviceid build="9200">2062</deviceid>
      <deviceid build="9200">1073</deviceid>
      <deviceid build="9200">1083</deviceid>
      <deviceid build="9200">1090</deviceid>
      <deviceid build="9200">1091</deviceid>
      <deviceid build="9200">E091</deviceid>
      <deviceid build="9200">10A0</deviceid>
      <deviceid build="9200">10A1</deviceid>
      <deviceid build="9200">E0A1</deviceid>
      <deviceid build="9200">10B0</deviceid>
      <deviceid build="9200">10B1</deviceid>
      <deviceid build="9200">E0B1</deviceid>
      <deviceid build="9200">10C0</deviceid>
      <deviceid build="9200">10C1</deviceid>
      <deviceid build="9200">E0C1</deviceid>
      <deviceid build="9200">10D0</deviceid>
      <deviceid build="9200">10D1</deviceid>
      <deviceid build="9200">E0D1</deviceid>
      <deviceid build="9200">10E0</deviceid>
      <deviceid build="9200">10E1</deviceid>
      <deviceid build="9200">E0E1</deviceid>
      <deviceid build="9200">10F0</deviceid>
      <deviceid build="9200">10F1</deviceid>
      <deviceid build="9200">E0F1</deviceid>
```

```
</NIC>

<NIC>
  <manufacturer>19A2</manufacturer>
    <deviceid build="9600">0211</deviceid>
    <deviceid build="9600">0215</deviceid>
    <deviceid build="9600">0221</deviceid>
    <deviceid build="9600">0700</deviceid>
    <deviceid build="9600">0710</deviceid>
</NIC>

<NIC>
  <manufacturer>10DF</manufacturer>
    <deviceid build="9600">0720</deviceid>
    <deviceid build="9600">E220</deviceid>
</NIC>

<NIC>
  <manufacturer>15B3</manufacturer>
    <deviceid build="9600">1000</deviceid>
    <deviceid build="9600">1001</deviceid>
    <deviceid build="9600">1002</deviceid>
    <deviceid build="9600">1003</deviceid>
    <deviceid build="9600">1004</deviceid>
    <deviceid build="9600">1005</deviceid>
    <deviceid build="9600">1006</deviceid>
    <deviceid build="9600">1007</deviceid>
    <deviceid build="9600">1008</deviceid>
    <deviceid build="9600">1009</deviceid>
    <deviceid build="9600">100A</deviceid>
    <deviceid build="9600">100B</deviceid>
    <deviceid build="9600">100C</deviceid>
    <deviceid build="9600">100D</deviceid>
    <deviceid build="9600">100E</deviceid>
    <deviceid build="9600">100F</deviceid>
    <deviceid build="9600">1010</deviceid>
    <deviceid build="9600">6340</deviceid>
    <deviceid build="9600">6341</deviceid>
    <deviceid build="9600">634A</deviceid>
    <deviceid build="9600">634B</deviceid>
    <deviceid build="9600">6354</deviceid>
    <deviceid build="9600">6368</deviceid>
    <deviceid build="9600">6369</deviceid>
    <deviceid build="9600">6372</deviceid>
    <deviceid build="9600">6732</deviceid>
    <deviceid build="9600">6733</deviceid>
    <deviceid build="9600">673C</deviceid>
    <deviceid build="9600">673D</deviceid>
    <deviceid build="9600">6746</deviceid>
    <deviceid build="9600">6750</deviceid>
    <deviceid build="9600">6751</deviceid>
    <deviceid build="9600">675A</deviceid>
    <deviceid build="9600">6764</deviceid>
    <deviceid build="9600">6765</deviceid>
    <deviceid build="9600">676E</deviceid>
```

```
        <deviceid build="9600">6778</deviceid>
        <deviceid build="15063">1013</deviceid>
        <deviceid build="15063">1015</deviceid>
        <deviceid build="15063">1017</deviceid>
        <deviceid build="15063">1019</deviceid>
        <deviceid build="15063">101B</deviceid>
        <deviceid build="15063">101D</deviceid>
        <deviceid build="15063">101F</deviceid>
        <deviceid build="15063">1021</deviceid>
        <deviceid build="15063">1023</deviceid>
        <deviceid build="15063">1025</deviceid>
        <deviceid build="15063">1027</deviceid>
        <deviceid build="15063">1029</deviceid>
        <deviceid build="15063">102B</deviceid>
        <deviceid build="15063">102F</deviceid>
    </NIC>

    <NIC>
      <manufacturer>1137</manufacturer>
        <deviceid build="15063">0043</deviceid>
    </NIC>

</SupportedNetworkInterfaceCards>
```

另请参阅

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

Windows 11 中的网络内核调试支持的以太网 NIC

Windows 10 中的网络内核调试支持的以太网 NIC

Windows 10 版本 2004 (20H1) 内部版本 19041 中支持用于网络内核调试的以太网 NIC

项目 • 2023/04/14

若要通过以太网网线进行内核调试，目标计算机必须具有支持的网络接口卡 (NIC) 。

在内核调试期间，运行调试器的计算机称为 *主计算机*，正在调试的计算机称为 *目标计算机*。 有关详细信息，请参阅 设置 KDNET 网络内核自动调试。

若要通过网络电缆进行内核调试，目标计算机必须具有受支持的网络适配器。 当目标计算机运行 Windows 时，支持此处列出的网络适配器进行内核调试。

版本信息

本主题列出了以下版本的 Windows 支持的适配器

- Windows 10版本 2004 (20H1) 内部版本 19041

XML 支持的 NIC 列表

这与上面以 XML 格式显示的信息相同。

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<SupportedNetworkInterfaceCards>
  <NIC>
    <manufacturer>8086</manufacturer>
      <deviceid>1000</deviceid>
      <deviceid>1001</deviceid>
      <deviceid>1004</deviceid>
      <deviceid>1008</deviceid>
      <deviceid>1009</deviceid>
      <deviceid>100C</deviceid>
      <deviceid>100D</deviceid>
      <deviceid>100E</deviceid>
      <deviceid>1015</deviceid>
      <deviceid>1016</deviceid>
      <deviceid>1017</deviceid>
      <deviceid>101E</deviceid>
      <deviceid>100F</deviceid>
      <deviceid>1011</deviceid>
      <deviceid>1026</deviceid>
```

```
        <deviceid>1027</deviceid>
        <deviceid>1028</deviceid>
        <deviceid>1010</deviceid>
        <deviceid>1012</deviceid>
        <deviceid>101D</deviceid>
        <deviceid>1079</deviceid>
        <deviceid>107A</deviceid>
        <deviceid>107B</deviceid>
        <deviceid>108A</deviceid>
        <deviceid>1099</deviceid>
        <deviceid>10B5</deviceid>
        <deviceid>1013</deviceid>
        <deviceid>1018</deviceid>
        <deviceid>1014</deviceid>
        <deviceid>1078</deviceid>
        <deviceid>1076</deviceid>
        <deviceid>107C</deviceid>
        <deviceid>1077</deviceid>
        <deviceid>1019</deviceid>
        <deviceid>101A</deviceid>
        <deviceid>1075</deviceid>
        <deviceid>105E</deviceid>
        <deviceid>105F</deviceid>
        <deviceid>1060</deviceid>
        <deviceid>10D9</deviceid>
        <deviceid>10DA</deviceid>
        <deviceid>10A4</deviceid>
        <deviceid>10D5</deviceid>
        <deviceid>10A5</deviceid>
        <deviceid>10BC</deviceid>
        <deviceid>107D</deviceid>
        <deviceid>107E</deviceid>
        <deviceid>107F</deviceid>
        <deviceid>10B9</deviceid>
        <deviceid>108B</deviceid>
        <deviceid>108C</deviceid>
        <deviceid>109A</deviceid>
        <deviceid>10D3</deviceid>
        <deviceid>10F6</deviceid>
        <deviceid>150C</deviceid>
        <deviceid>1096</deviceid>
        <deviceid>1098</deviceid>
        <deviceid>10BA</deviceid>
        <deviceid>10BB</deviceid>
        <deviceid>1501</deviceid>
        <deviceid>1049</deviceid>
        <deviceid>104A</deviceid>
        <deviceid>104B</deviceid>
        <deviceid>104C</deviceid>
        <deviceid>10C4</deviceid>
        <deviceid>10C5</deviceid>
        <deviceid>104D</deviceid>
        <deviceid>10BF</deviceid>
        <deviceid>10F5</deviceid>
        <deviceid>10CB</deviceid>
```

```
        <deviceid>10BD</deviceid>
        <deviceid>10E5</deviceid>
        <deviceid>294C</deviceid>
        <deviceid>10C0</deviceid>
        <deviceid>10C3</deviceid>
        <deviceid>10C2</deviceid>
        <deviceid>10CC</deviceid>
        <deviceid>10CD</deviceid>
        <deviceid>10CE</deviceid>
        <deviceid>10DE</deviceid>
        <deviceid>10DF</deviceid>
        <deviceid>1525</deviceid>
        <deviceid>10EA</deviceid>
        <deviceid>10EB</deviceid>
        <deviceid>10EF</deviceid>
        <deviceid>10F0</deviceid>
        <deviceid>1502</deviceid>
        <deviceid>1503</deviceid>
        <deviceid>153A</deviceid>
        <deviceid>153B</deviceid>
        <deviceid>155A</deviceid>
        <deviceid>1559</deviceid>
        <deviceid>15A0</deviceid>
        <deviceid>15A1</deviceid>
        <deviceid>15A2</deviceid>
        <deviceid>15A3</deviceid>
        <deviceid>156F</deviceid>
        <deviceid>1570</deviceid>
        <deviceid>15B7</deviceid>
        <deviceid>15B8</deviceid>
        <deviceid>15B9</deviceid>
        <deviceid>15D7</deviceid>
        <deviceid>15D8</deviceid>
        <deviceid>15E3</deviceid>
        <deviceid>15D6</deviceid>
        <deviceid>15BD</deviceid>
        <deviceid>15BE</deviceid>
        <deviceid>15BB</deviceid>
        <deviceid>15BC</deviceid>
        <deviceid>15DF</deviceid>
        <deviceid>15E0</deviceid>
        <deviceid>15E1</deviceid>
        <deviceid>15E2</deviceid>
        <deviceid>10C9</deviceid>
        <deviceid>10E6</deviceid>
        <deviceid>10E7</deviceid>
        <deviceid>10E8</deviceid>
        <deviceid>1526</deviceid>
        <deviceid>150A</deviceid>
        <deviceid>1518</deviceid>
        <deviceid>150D</deviceid>
        <deviceid>10A7</deviceid>
        <deviceid>10A9</deviceid>
        <deviceid>10D6</deviceid>
        <deviceid>150E</deviceid>
```

```
        <deviceid>150F</deviceid>
        <deviceid>1510</deviceid>
        <deviceid>1511</deviceid>
        <deviceid>1516</deviceid>
        <deviceid>1527</deviceid>
        <deviceid>1521</deviceid>
        <deviceid>1522</deviceid>
        <deviceid>1523</deviceid>
        <deviceid>1524</deviceid>
        <deviceid>1546</deviceid>
        <deviceid>1533</deviceid>
        <deviceid>1534</deviceid>
        <deviceid>1535</deviceid>
        <deviceid>1536</deviceid>
        <deviceid>1537</deviceid>
        <deviceid>1538</deviceid>
        <deviceid>157B</deviceid>
        <deviceid>157C</deviceid>
        <deviceid>1539</deviceid>
        <deviceid>1F40</deviceid>
        <deviceid>1F41</deviceid>
        <deviceid>1F45</deviceid>
        <deviceid>0438</deviceid>
        <deviceid>043A</deviceid>
        <deviceid>043C</deviceid>
        <deviceid>0440</deviceid>
        <deviceid>10C6</deviceid>
        <deviceid>10C7</deviceid>
        <deviceid>10C8</deviceid>
        <deviceid>150B</deviceid>
        <deviceid>10DB</deviceid>
        <deviceid>10DD</deviceid>
        <deviceid>10EC</deviceid>
        <deviceid>10F1</deviceid>
        <deviceid>10E1</deviceid>
        <deviceid>10F4</deviceid>
        <deviceid>10F7</deviceid>
        <deviceid>1514</deviceid>
        <deviceid>1517</deviceid>
        <deviceid>10F8</deviceid>
        <deviceid>000C</deviceid>
        <deviceid>1F63</deviceid>
        <deviceid>10F9</deviceid>
        <deviceid>10FB</deviceid>
        <deviceid>11A9</deviceid>
        <deviceid>1071</deviceid>
        <deviceid>1F72</deviceid>
        <deviceid>17D0</deviceid>
        <deviceid>0470</deviceid>
        <deviceid>211B</deviceid>
        <deviceid>8976</deviceid>
        <deviceid>2159</deviceid>
        <deviceid>000D</deviceid>
        <deviceid>0008</deviceid>
        <deviceid>152A</deviceid>
```

```
      <deviceid>1529</deviceid>
      <deviceid>1507</deviceid>
      <deviceid>154D</deviceid>
      <deviceid>154A</deviceid>
      <deviceid>1558</deviceid>
      <deviceid>1557</deviceid>
      <deviceid>0001</deviceid>
      <deviceid>10FC</deviceid>
      <deviceid>151C</deviceid>
      <deviceid>1528</deviceid>
      <deviceid>1560</deviceid>
      <deviceid>1563</deviceid>
      <deviceid>ABCD</deviceid>
      <deviceid>15AA</deviceid>
      <deviceid>15AB</deviceid>
      <deviceid>15AC</deviceid>
      <deviceid>15AD</deviceid>
      <deviceid>15AE</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10EC</manufacturer>
      <deviceid>8136</deviceid>
      <deviceid>8137</deviceid>
      <deviceid>8168</deviceid>
      <deviceid>8167</deviceid>
      <deviceid>8169</deviceid>
      <deviceid>8166</deviceid>
      <deviceid>8161</deviceid>
      <deviceid>8125</deviceid>
      <deviceid>8225</deviceid>
      <deviceid>2502</deviceid>
      <deviceid>2600</deviceid>
      <deviceid>3000</deviceid>
  </NIC>

  <NIC>
    <manufacturer>14E4</manufacturer>
      <deviceid>1644</deviceid>
      <deviceid>1645</deviceid>
      <deviceid>1646</deviceid>
      <deviceid>16A6</deviceid>
      <deviceid>16C6</deviceid>
      <deviceid>1647</deviceid>
      <deviceid>16A7</deviceid>
      <deviceid>16C7</deviceid>
      <deviceid>164D</deviceid>
      <deviceid>1648</deviceid>
      <deviceid>16A8</deviceid>
      <deviceid>1653</deviceid>
      <deviceid>166E</deviceid>
      <deviceid>1654</deviceid>
      <deviceid>165D</deviceid>
      <deviceid>165E</deviceid>
      <deviceid>166D</deviceid>
```

```
        <deviceid>170D</deviceid>
        <deviceid>170E</deviceid>
        <deviceid>1696</deviceid>
        <deviceid>1676</deviceid>
        <deviceid>1677</deviceid>
        <deviceid>1659</deviceid>
        <deviceid>167C</deviceid>
        <deviceid>167D</deviceid>
        <deviceid>169D</deviceid>
        <deviceid>16F7</deviceid>
        <deviceid>16FD</deviceid>
        <deviceid>16FE</deviceid>
        <deviceid>16DD</deviceid>
        <deviceid>1668</deviceid>
        <deviceid>1669</deviceid>
        <deviceid>1678</deviceid>
        <deviceid>1679</deviceid>
        <deviceid>1600</deviceid>
        <deviceid>1601</deviceid>
        <deviceid>166A</deviceid>
        <deviceid>166B</deviceid>
        <deviceid>16FF</deviceid>
        <deviceid>170F</deviceid>
        <deviceid>169B</deviceid>
        <deviceid>1693</deviceid>
        <deviceid>167F</deviceid>
        <deviceid>169A</deviceid>
        <deviceid>1698</deviceid>
        <deviceid>1692</deviceid>
        <deviceid>1694</deviceid>
        <deviceid>1690</deviceid>
        <deviceid>1691</deviceid>
        <deviceid>1699</deviceid>
        <deviceid>16A0</deviceid>
        <deviceid>167B</deviceid>
        <deviceid>1673</deviceid>
        <deviceid>165A</deviceid>
        <deviceid>1674</deviceid>
        <deviceid>1681</deviceid>
        <deviceid>1680</deviceid>
        <deviceid>1688</deviceid>
        <deviceid>167A</deviceid>
        <deviceid>1672</deviceid>
        <deviceid>1684</deviceid>
        <deviceid>165B</deviceid>
        <deviceid>16B1</deviceid>
        <deviceid>16B5</deviceid>
        <deviceid>16B0</deviceid>
        <deviceid>16B4</deviceid>
        <deviceid>16B2</deviceid>
        <deviceid>16B6</deviceid>
        <deviceid>1682</deviceid>
        <deviceid>1686</deviceid>
        <deviceid>16B3</deviceid>
        <deviceid>16B7</deviceid>
```

```
        <deviceid>1655</deviceid>
        <deviceid>1665</deviceid>
        <deviceid>1656</deviceid>
        <deviceid>1657</deviceid>
        <deviceid>165F</deviceid>
        <deviceid>165C</deviceid>
        <deviceid>1683</deviceid>
        <deviceid>1641</deviceid>
        <deviceid>1642</deviceid>
        <deviceid>1643</deviceid>
        <deviceid>1687</deviceid>
        <deviceid>164A</deviceid>
        <deviceid>16AA</deviceid>
        <deviceid>164C</deviceid>
        <deviceid>16AC</deviceid>
        <deviceid>1639</deviceid>
        <deviceid>163A</deviceid>
        <deviceid>163B</deviceid>
        <deviceid>163C</deviceid>
        <deviceid>164E</deviceid>
        <deviceid>164F</deviceid>
        <deviceid>1650</deviceid>
        <deviceid>1662</deviceid>
        <deviceid>1663</deviceid>
        <deviceid>168A</deviceid>
        <deviceid>168D</deviceid>
        <deviceid>16A1</deviceid>
        <deviceid>16A2</deviceid>
        <deviceid>168E</deviceid>
        <deviceid>163D</deviceid>
        <deviceid>16A5</deviceid>
        <deviceid>16A4</deviceid>
        <deviceid>16AE</deviceid>
        <deviceid>163E</deviceid>
        <deviceid>16C0</deviceid>
        <deviceid>16C9</deviceid>
        <deviceid>16CA</deviceid>
        <deviceid>16CC</deviceid>
        <deviceid>16CD</deviceid>
        <deviceid>16CE</deviceid>
        <deviceid>16CF</deviceid>
        <deviceid>16D0</deviceid>
        <deviceid>16D1</deviceid>
        <deviceid>16D2</deviceid>
        <deviceid>16D4</deviceid>
        <deviceid>16D5</deviceid>
        <deviceid>16D6</deviceid>
        <deviceid>16D7</deviceid>
        <deviceid>16D8</deviceid>
        <deviceid>16D9</deviceid>
        <deviceid>16DA</deviceid>
        <deviceid>16DB</deviceid>
        <deviceid>16DC</deviceid>
        <deviceid>16DE</deviceid>
        <deviceid>16DF</deviceid>
```

```
      <deviceid>16E0</deviceid>
      <deviceid>16E2</deviceid>
      <deviceid>16E3</deviceid>
      <deviceid>16E4</deviceid>
      <deviceid>16E7</deviceid>
      <deviceid>16E8</deviceid>
      <deviceid>16E9</deviceid>
      <deviceid>16EA</deviceid>
      <deviceid>16EB</deviceid>
      <deviceid>16EC</deviceid>
      <deviceid>16ED</deviceid>
      <deviceid>16EE</deviceid>
      <deviceid>16EF</deviceid>
      <deviceid>16F0</deviceid>
      <deviceid>16F1</deviceid>
      <deviceid>1614</deviceid>
      <deviceid>D802</deviceid>
  </NIC>

  <NIC>
    <manufacturer>1969</manufacturer>
      <deviceid>1062</deviceid>
      <deviceid>1063</deviceid>
      <deviceid>2060</deviceid>
      <deviceid>2062</deviceid>
      <deviceid>1073</deviceid>
      <deviceid>1083</deviceid>
      <deviceid>1090</deviceid>
      <deviceid>1091</deviceid>
      <deviceid>E091</deviceid>
      <deviceid>10A0</deviceid>
      <deviceid>10A1</deviceid>
      <deviceid>E0A1</deviceid>
      <deviceid>10B0</deviceid>
      <deviceid>10B1</deviceid>
      <deviceid>E0B1</deviceid>
      <deviceid>10C0</deviceid>
      <deviceid>10C1</deviceid>
      <deviceid>E0C1</deviceid>
      <deviceid>10D0</deviceid>
      <deviceid>10D1</deviceid>
      <deviceid>E0D1</deviceid>
      <deviceid>10E0</deviceid>
      <deviceid>10E1</deviceid>
      <deviceid>E0E1</deviceid>
      <deviceid>10F0</deviceid>
      <deviceid>10F1</deviceid>
      <deviceid>E0F1</deviceid>
  </NIC>

  <NIC>
    <manufacturer>19A2</manufacturer>
      <deviceid>0211</deviceid>
      <deviceid>0215</deviceid>
      <deviceid>0221</deviceid>
```

```
      <deviceid>0700</deviceid>
      <deviceid>0710</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10DF</manufacturer>
      <deviceid>0720</deviceid>
      <deviceid>E220</deviceid>
  </NIC>

  <NIC>
    <manufacturer>15B3</manufacturer>
      <deviceid>1000</deviceid>
      <deviceid>1001</deviceid>
      <deviceid>1002</deviceid>
      <deviceid>1003</deviceid>
      <deviceid>1004</deviceid>
      <deviceid>1005</deviceid>
      <deviceid>1006</deviceid>
      <deviceid>1007</deviceid>
      <deviceid>1008</deviceid>
      <deviceid>1009</deviceid>
      <deviceid>100A</deviceid>
      <deviceid>100B</deviceid>
      <deviceid>100C</deviceid>
      <deviceid>100D</deviceid>
      <deviceid>100E</deviceid>
      <deviceid>100F</deviceid>
      <deviceid>1010</deviceid>
      <deviceid>6340</deviceid>
      <deviceid>6341</deviceid>
      <deviceid>634A</deviceid>
      <deviceid>634B</deviceid>
      <deviceid>6354</deviceid>
      <deviceid>6368</deviceid>
      <deviceid>6369</deviceid>
      <deviceid>6372</deviceid>
      <deviceid>6732</deviceid>
      <deviceid>6733</deviceid>
      <deviceid>673C</deviceid>
      <deviceid>673D</deviceid>
      <deviceid>6746</deviceid>
      <deviceid>6750</deviceid>
      <deviceid>6751</deviceid>
      <deviceid>675A</deviceid>
      <deviceid>6764</deviceid>
      <deviceid>6765</deviceid>
      <deviceid>676E</deviceid>
      <deviceid>6778</deviceid>
      <deviceid>1013</deviceid>
      <deviceid>1015</deviceid>
      <deviceid>1017</deviceid>
      <deviceid>1019</deviceid>
      <deviceid>101B</deviceid>
      <deviceid>101D</deviceid>
```

```
      <deviceid>101F</deviceid>
      <deviceid>1021</deviceid>
      <deviceid>1023</deviceid>
      <deviceid>1025</deviceid>
      <deviceid>1027</deviceid>
      <deviceid>1029</deviceid>
      <deviceid>102B</deviceid>
      <deviceid>102F</deviceid>
  </NIC>

  <NIC>
    <manufacturer>1137</manufacturer>
      <deviceid>0043</deviceid>
  </NIC>

</SupportedNetworkInterfaceCards>
```

另请参阅

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

Windows 8 中的网络内核调试支持的以太网 NIC

Windows 8.1 中的网络内核调试支持的以太网 NIC

Windows 10版本 1909 (19H2) 内部版本 18363 中支持用于网络内核调试的以太网 NIC

项目 • 2024/02/07

若要通过以太网网线进行内核调试，目标计算机必须具有受支持的网络接口卡 (NIC) 。

在内核调试期间，运行调试器的计算机称为 *主计算机*，正在调试的计算机称为 *目标计算机*。 有关详细信息，请参阅 自动设置 KDNET 网络内核调试。

有关支持的网络适配器的一般信息，请参阅 Windows 10 中支持的网络内核调试的以太网 NIC。

版本信息

支持的适配器列表适用于以下版本的 Windows

- Windows 10版本 1909 (19H2) 内部版本 18363

XML

```
<?xml version="1.0" encoding="utf-8"?>
<SupportedNetworkInterfaceCards>
  <NIC>
    <manufacturer>8086</manufacturer>
      <deviceid>1000</deviceid>
      <deviceid>1001</deviceid>
      <deviceid>1004</deviceid>
      <deviceid>1008</deviceid>
      <deviceid>1009</deviceid>
      <deviceid>100C</deviceid>
      <deviceid>100D</deviceid>
      <deviceid>100E</deviceid>
      <deviceid>1015</deviceid>
      <deviceid>1016</deviceid>
      <deviceid>1017</deviceid>
      <deviceid>101E</deviceid>
      <deviceid>100F</deviceid>
      <deviceid>1011</deviceid>
      <deviceid>1026</deviceid>
      <deviceid>1027</deviceid>
      <deviceid>1028</deviceid>
      <deviceid>1010</deviceid>
      <deviceid>1012</deviceid>
      <deviceid>101D</deviceid>
      <deviceid>1079</deviceid>
      <deviceid>107A</deviceid>
```

```
        <deviceid>107B</deviceid>
        <deviceid>108A</deviceid>
        <deviceid>1099</deviceid>
        <deviceid>10B5</deviceid>
        <deviceid>1013</deviceid>
        <deviceid>1018</deviceid>
        <deviceid>1014</deviceid>
        <deviceid>1078</deviceid>
        <deviceid>1076</deviceid>
        <deviceid>107C</deviceid>
        <deviceid>1077</deviceid>
        <deviceid>1019</deviceid>
        <deviceid>101A</deviceid>
        <deviceid>1075</deviceid>
        <deviceid>105E</deviceid>
        <deviceid>105F</deviceid>
        <deviceid>1060</deviceid>
        <deviceid>10D9</deviceid>
        <deviceid>10DA</deviceid>
        <deviceid>10A4</deviceid>
        <deviceid>10D5</deviceid>
        <deviceid>10A5</deviceid>
        <deviceid>10BC</deviceid>
        <deviceid>107D</deviceid>
        <deviceid>107E</deviceid>
        <deviceid>107F</deviceid>
        <deviceid>10B9</deviceid>
        <deviceid>108B</deviceid>
        <deviceid>108C</deviceid>
        <deviceid>109A</deviceid>
        <deviceid>10D3</deviceid>
        <deviceid>10F6</deviceid>
        <deviceid>150C</deviceid>
        <deviceid>1096</deviceid>
        <deviceid>1098</deviceid>
        <deviceid>10BA</deviceid>
        <deviceid>10BB</deviceid>
        <deviceid>1501</deviceid>
        <deviceid>1049</deviceid>
        <deviceid>104A</deviceid>
        <deviceid>104B</deviceid>
        <deviceid>104C</deviceid>
        <deviceid>10C4</deviceid>
        <deviceid>10C5</deviceid>
        <deviceid>104D</deviceid>
        <deviceid>10BF</deviceid>
        <deviceid>10F5</deviceid>
        <deviceid>10CB</deviceid>
        <deviceid>10BD</deviceid>
        <deviceid>10E5</deviceid>
        <deviceid>294C</deviceid>
        <deviceid>10C0</deviceid>
        <deviceid>10C3</deviceid>
        <deviceid>10C2</deviceid>
        <deviceid>10CC</deviceid>
```

```
        <deviceid>10CD</deviceid>
        <deviceid>10CE</deviceid>
        <deviceid>10DE</deviceid>
        <deviceid>10DF</deviceid>
        <deviceid>1525</deviceid>
        <deviceid>10EA</deviceid>
        <deviceid>10EB</deviceid>
        <deviceid>10EF</deviceid>
        <deviceid>10F0</deviceid>
        <deviceid>1502</deviceid>
        <deviceid>1503</deviceid>
        <deviceid>153A</deviceid>
        <deviceid>153B</deviceid>
        <deviceid>155A</deviceid>
        <deviceid>1559</deviceid>
        <deviceid>15A0</deviceid>
        <deviceid>15A1</deviceid>
        <deviceid>15A2</deviceid>
        <deviceid>15A3</deviceid>
        <deviceid>156F</deviceid>
        <deviceid>1570</deviceid>
        <deviceid>15B7</deviceid>
        <deviceid>15B8</deviceid>
        <deviceid>15B9</deviceid>
        <deviceid>15D7</deviceid>
        <deviceid>15D8</deviceid>
        <deviceid>15E3</deviceid>
        <deviceid>15D6</deviceid>
        <deviceid>15BD</deviceid>
        <deviceid>15BE</deviceid>
        <deviceid>15BB</deviceid>
        <deviceid>15BC</deviceid>
        <deviceid>15DF</deviceid>
        <deviceid>15E0</deviceid>
        <deviceid>15E1</deviceid>
        <deviceid>15E2</deviceid>
        <deviceid>10C9</deviceid>
        <deviceid>10E6</deviceid>
        <deviceid>10E7</deviceid>
        <deviceid>10E8</deviceid>
        <deviceid>1526</deviceid>
        <deviceid>150A</deviceid>
        <deviceid>1518</deviceid>
        <deviceid>150D</deviceid>
        <deviceid>10A7</deviceid>
        <deviceid>10A9</deviceid>
        <deviceid>10D6</deviceid>
        <deviceid>150E</deviceid>
        <deviceid>150F</deviceid>
        <deviceid>1510</deviceid>
        <deviceid>1511</deviceid>
        <deviceid>1516</deviceid>
        <deviceid>1527</deviceid>
        <deviceid>1521</deviceid>
        <deviceid>1522</deviceid>
```

```
        <deviceid>1523</deviceid>
        <deviceid>1524</deviceid>
        <deviceid>1546</deviceid>
        <deviceid>1533</deviceid>
        <deviceid>1534</deviceid>
        <deviceid>1535</deviceid>
        <deviceid>1536</deviceid>
        <deviceid>1537</deviceid>
        <deviceid>1538</deviceid>
        <deviceid>157B</deviceid>
        <deviceid>157C</deviceid>
        <deviceid>1539</deviceid>
        <deviceid>1F40</deviceid>
        <deviceid>1F41</deviceid>
        <deviceid>1F45</deviceid>
        <deviceid>0438</deviceid>
        <deviceid>043A</deviceid>
        <deviceid>043C</deviceid>
        <deviceid>0440</deviceid>
        <deviceid>10C6</deviceid>
        <deviceid>10C7</deviceid>
        <deviceid>10C8</deviceid>
        <deviceid>150B</deviceid>
        <deviceid>10DB</deviceid>
        <deviceid>10DD</deviceid>
        <deviceid>10EC</deviceid>
        <deviceid>10F1</deviceid>
        <deviceid>10E1</deviceid>
        <deviceid>10F4</deviceid>
        <deviceid>10F7</deviceid>
        <deviceid>1514</deviceid>
        <deviceid>1517</deviceid>
        <deviceid>10F8</deviceid>
        <deviceid>000C</deviceid>
        <deviceid>1F63</deviceid>
        <deviceid>10F9</deviceid>
        <deviceid>10FB</deviceid>
        <deviceid>11A9</deviceid>
        <deviceid>1071</deviceid>
        <deviceid>1F72</deviceid>
        <deviceid>17D0</deviceid>
        <deviceid>0470</deviceid>
        <deviceid>211B</deviceid>
        <deviceid>8976</deviceid>
        <deviceid>2159</deviceid>
        <deviceid>000D</deviceid>
        <deviceid>0008</deviceid>
        <deviceid>152A</deviceid>
        <deviceid>1529</deviceid>
        <deviceid>1507</deviceid>
        <deviceid>154D</deviceid>
        <deviceid>154A</deviceid>
        <deviceid>1558</deviceid>
        <deviceid>1557</deviceid>
        <deviceid>0001</deviceid>
```

```
      <deviceid>10FC</deviceid>
      <deviceid>151C</deviceid>
      <deviceid>1528</deviceid>
      <deviceid>1560</deviceid>
      <deviceid>1563</deviceid>
      <deviceid>ABCD</deviceid>
      <deviceid>15AA</deviceid>
      <deviceid>15AB</deviceid>
      <deviceid>15AC</deviceid>
      <deviceid>15AD</deviceid>
      <deviceid>15AE</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10EC</manufacturer>
      <deviceid>8136</deviceid>
      <deviceid>8137</deviceid>
      <deviceid>8168</deviceid>
      <deviceid>8167</deviceid>
      <deviceid>8169</deviceid>
      <deviceid>8166</deviceid>
      <deviceid>8161</deviceid>
      <deviceid>8125</deviceid>
      <deviceid>8225</deviceid>
      <deviceid>2502</deviceid>
      <deviceid>2600</deviceid>
      <deviceid>3000</deviceid>
  </NIC>

  <NIC>
    <manufacturer>14E4</manufacturer>
      <deviceid>1644</deviceid>
      <deviceid>1645</deviceid>
      <deviceid>1646</deviceid>
      <deviceid>16A6</deviceid>
      <deviceid>16C6</deviceid>
      <deviceid>1647</deviceid>
      <deviceid>16A7</deviceid>
      <deviceid>16C7</deviceid>
      <deviceid>164D</deviceid>
      <deviceid>1648</deviceid>
      <deviceid>16A8</deviceid>
      <deviceid>1653</deviceid>
      <deviceid>166E</deviceid>
      <deviceid>1654</deviceid>
      <deviceid>165D</deviceid>
      <deviceid>165E</deviceid>
      <deviceid>166D</deviceid>
      <deviceid>170D</deviceid>
      <deviceid>170E</deviceid>
      <deviceid>1696</deviceid>
      <deviceid>1676</deviceid>
      <deviceid>1677</deviceid>
      <deviceid>1659</deviceid>
      <deviceid>167C</deviceid>
```

```
        <deviceid>167D</deviceid>
        <deviceid>169D</deviceid>
        <deviceid>16F7</deviceid>
        <deviceid>16FD</deviceid>
        <deviceid>16FE</deviceid>
        <deviceid>16DD</deviceid>
        <deviceid>1668</deviceid>
        <deviceid>1669</deviceid>
        <deviceid>1678</deviceid>
        <deviceid>1679</deviceid>
        <deviceid>1600</deviceid>
        <deviceid>1601</deviceid>
        <deviceid>166A</deviceid>
        <deviceid>166B</deviceid>
        <deviceid>16FF</deviceid>
        <deviceid>170F</deviceid>
        <deviceid>169B</deviceid>
        <deviceid>1693</deviceid>
        <deviceid>167F</deviceid>
        <deviceid>169A</deviceid>
        <deviceid>1698</deviceid>
        <deviceid>1692</deviceid>
        <deviceid>1694</deviceid>
        <deviceid>1690</deviceid>
        <deviceid>1691</deviceid>
        <deviceid>1699</deviceid>
        <deviceid>16A0</deviceid>
        <deviceid>167B</deviceid>
        <deviceid>1673</deviceid>
        <deviceid>165A</deviceid>
        <deviceid>1674</deviceid>
        <deviceid>1681</deviceid>
        <deviceid>1680</deviceid>
        <deviceid>1688</deviceid>
        <deviceid>167A</deviceid>
        <deviceid>1672</deviceid>
        <deviceid>1684</deviceid>
        <deviceid>165B</deviceid>
        <deviceid>16B1</deviceid>
        <deviceid>16B5</deviceid>
        <deviceid>16B0</deviceid>
        <deviceid>16B4</deviceid>
        <deviceid>16B2</deviceid>
        <deviceid>16B6</deviceid>
        <deviceid>1682</deviceid>
        <deviceid>1686</deviceid>
        <deviceid>16B3</deviceid>
        <deviceid>16B7</deviceid>
        <deviceid>1655</deviceid>
        <deviceid>1665</deviceid>
        <deviceid>1656</deviceid>
        <deviceid>1657</deviceid>
        <deviceid>165F</deviceid>
        <deviceid>165C</deviceid>
        <deviceid>1683</deviceid>
```

```
      <deviceid>1641</deviceid>
      <deviceid>1642</deviceid>
      <deviceid>1643</deviceid>
      <deviceid>1687</deviceid>
      <deviceid>164A</deviceid>
      <deviceid>16AA</deviceid>
      <deviceid>164C</deviceid>
      <deviceid>16AC</deviceid>
      <deviceid>1639</deviceid>
      <deviceid>163A</deviceid>
      <deviceid>163B</deviceid>
      <deviceid>163C</deviceid>
      <deviceid>164E</deviceid>
      <deviceid>164F</deviceid>
      <deviceid>1650</deviceid>
      <deviceid>1662</deviceid>
      <deviceid>1663</deviceid>
      <deviceid>168A</deviceid>
      <deviceid>168D</deviceid>
      <deviceid>16A1</deviceid>
      <deviceid>16A2</deviceid>
      <deviceid>168E</deviceid>
      <deviceid>163D</deviceid>
      <deviceid>16A5</deviceid>
      <deviceid>16A4</deviceid>
      <deviceid>16AE</deviceid>
      <deviceid>163E</deviceid>
  </NIC>

  <NIC>
    <manufacturer>1969</manufacturer>
      <deviceid>1062</deviceid>
      <deviceid>1063</deviceid>
      <deviceid>2060</deviceid>
      <deviceid>2062</deviceid>
      <deviceid>1073</deviceid>
      <deviceid>1083</deviceid>
      <deviceid>1090</deviceid>
      <deviceid>1091</deviceid>
      <deviceid>E091</deviceid>
      <deviceid>10A0</deviceid>
      <deviceid>10A1</deviceid>
      <deviceid>E0A1</deviceid>
      <deviceid>10B0</deviceid>
      <deviceid>10B1</deviceid>
      <deviceid>E0B1</deviceid>
      <deviceid>10C0</deviceid>
      <deviceid>10C1</deviceid>
      <deviceid>E0C1</deviceid>
      <deviceid>10D0</deviceid>
      <deviceid>10D1</deviceid>
      <deviceid>E0D1</deviceid>
      <deviceid>10E0</deviceid>
      <deviceid>10E1</deviceid>
      <deviceid>E0E1</deviceid>
```

```
      <deviceid>10F0</deviceid>
      <deviceid>10F1</deviceid>
      <deviceid>E0F1</deviceid>
  </NIC>

  <NIC>
    <manufacturer>19A2</manufacturer>
      <deviceid>0211</deviceid>
      <deviceid>0215</deviceid>
      <deviceid>0221</deviceid>
      <deviceid>0700</deviceid>
      <deviceid>0710</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10DF</manufacturer>
      <deviceid>0720</deviceid>
      <deviceid>E220</deviceid>
  </NIC>

  <NIC>
    <manufacturer>15B3</manufacturer>
      <deviceid>1000</deviceid>
      <deviceid>1001</deviceid>
      <deviceid>1002</deviceid>
      <deviceid>1003</deviceid>
      <deviceid>1004</deviceid>
      <deviceid>1005</deviceid>
      <deviceid>1006</deviceid>
      <deviceid>1007</deviceid>
      <deviceid>1008</deviceid>
      <deviceid>1009</deviceid>
      <deviceid>100A</deviceid>
      <deviceid>100B</deviceid>
      <deviceid>100C</deviceid>
      <deviceid>100D</deviceid>
      <deviceid>100E</deviceid>
      <deviceid>100F</deviceid>
      <deviceid>1010</deviceid>
      <deviceid>6340</deviceid>
      <deviceid>6341</deviceid>
      <deviceid>634A</deviceid>
      <deviceid>634B</deviceid>
      <deviceid>6354</deviceid>
      <deviceid>6368</deviceid>
      <deviceid>6369</deviceid>
      <deviceid>6372</deviceid>
      <deviceid>6732</deviceid>
      <deviceid>6733</deviceid>
      <deviceid>673C</deviceid>
      <deviceid>673D</deviceid>
      <deviceid>6746</deviceid>
      <deviceid>6750</deviceid>
      <deviceid>6751</deviceid>
      <deviceid>675A</deviceid>
```

```
        <deviceid>6764</deviceid>
        <deviceid>6765</deviceid>
        <deviceid>676E</deviceid>
        <deviceid>6778</deviceid>
        <deviceid>1013</deviceid>
        <deviceid>1015</deviceid>
        <deviceid>1017</deviceid>
        <deviceid>1019</deviceid>
        <deviceid>101B</deviceid>
        <deviceid>101D</deviceid>
        <deviceid>101F</deviceid>
        <deviceid>1021</deviceid>
        <deviceid>1023</deviceid>
        <deviceid>1025</deviceid>
        <deviceid>1027</deviceid>
        <deviceid>1029</deviceid>
        <deviceid>102B</deviceid>
        <deviceid>102F</deviceid>
    </NIC>

    <NIC>
      <manufacturer>1137</manufacturer>
        <deviceid>0043</deviceid>
    </NIC>

</SupportedNetworkInterfaceCards>
```

另请参阅

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

Windows 8 中的网络内核调试支持的以太网 NIC

Windows 8.1 中的网络内核调试支持的以太网 NIC

Windows 10版本 1903 (19H1) 内部版本 18362 中支持用于网络内核调试的以太网 NIC

项目 • 2024/02/06

若要通过以太网网线进行内核调试，目标计算机必须具有受支持的网络接口卡 (NIC) 。

在内核调试期间，运行调试器的计算机称为 *主计算机*，正在调试的计算机称为 *目标计算机*。 有关详细信息，请参阅 设置 KDNET 网络内核自动调试。

有关支持的网络适配器的一般信息，请参阅支持在 Windows 10 中进行网络内核调试的以太网 NIC。

版本信息

支持的适配器列表适用于以下版本的 Windows

- Windows 10版本 1903 (19H1) 内部版本 18362

XML

```
<?xml version="1.0" encoding="utf-8"?>
<SupportedNetworkInterfaceCards>
  <NIC>
    <manufacturer>8086</manufacturer>
      <deviceid>1000</deviceid>
      <deviceid>1001</deviceid>
      <deviceid>1004</deviceid>
      <deviceid>1008</deviceid>
      <deviceid>1009</deviceid>
      <deviceid>100C</deviceid>
      <deviceid>100D</deviceid>
      <deviceid>100E</deviceid>
      <deviceid>1015</deviceid>
      <deviceid>1016</deviceid>
      <deviceid>1017</deviceid>
      <deviceid>101E</deviceid>
      <deviceid>100F</deviceid>
      <deviceid>1011</deviceid>
      <deviceid>1026</deviceid>
      <deviceid>1027</deviceid>
      <deviceid>1028</deviceid>
      <deviceid>1010</deviceid>
      <deviceid>1012</deviceid>
      <deviceid>101D</deviceid>
      <deviceid>1079</deviceid>
      <deviceid>107A</deviceid>
```

```
        <deviceid>107B</deviceid>
        <deviceid>108A</deviceid>
        <deviceid>1099</deviceid>
        <deviceid>10B5</deviceid>
        <deviceid>1013</deviceid>
        <deviceid>1018</deviceid>
        <deviceid>1014</deviceid>
        <deviceid>1078</deviceid>
        <deviceid>1076</deviceid>
        <deviceid>107C</deviceid>
        <deviceid>1077</deviceid>
        <deviceid>1019</deviceid>
        <deviceid>101A</deviceid>
        <deviceid>1075</deviceid>
        <deviceid>105E</deviceid>
        <deviceid>105F</deviceid>
        <deviceid>1060</deviceid>
        <deviceid>10D9</deviceid>
        <deviceid>10DA</deviceid>
        <deviceid>10A4</deviceid>
        <deviceid>10D5</deviceid>
        <deviceid>10A5</deviceid>
        <deviceid>10BC</deviceid>
        <deviceid>107D</deviceid>
        <deviceid>107E</deviceid>
        <deviceid>107F</deviceid>
        <deviceid>10B9</deviceid>
        <deviceid>108B</deviceid>
        <deviceid>108C</deviceid>
        <deviceid>109A</deviceid>
        <deviceid>10D3</deviceid>
        <deviceid>10F6</deviceid>
        <deviceid>150C</deviceid>
        <deviceid>1096</deviceid>
        <deviceid>1098</deviceid>
        <deviceid>10BA</deviceid>
        <deviceid>10BB</deviceid>
        <deviceid>1501</deviceid>
        <deviceid>1049</deviceid>
        <deviceid>104A</deviceid>
        <deviceid>104B</deviceid>
        <deviceid>104C</deviceid>
        <deviceid>10C4</deviceid>
        <deviceid>10C5</deviceid>
        <deviceid>104D</deviceid>
        <deviceid>10BF</deviceid>
        <deviceid>10F5</deviceid>
        <deviceid>10CB</deviceid>
        <deviceid>10BD</deviceid>
        <deviceid>10E5</deviceid>
        <deviceid>294C</deviceid>
        <deviceid>10C0</deviceid>
        <deviceid>10C3</deviceid>
        <deviceid>10C2</deviceid>
        <deviceid>10CC</deviceid>
```

```
        <deviceid>10CD</deviceid>
        <deviceid>10CE</deviceid>
        <deviceid>10DE</deviceid>
        <deviceid>10DF</deviceid>
        <deviceid>1525</deviceid>
        <deviceid>10EA</deviceid>
        <deviceid>10EB</deviceid>
        <deviceid>10EF</deviceid>
        <deviceid>10F0</deviceid>
        <deviceid>1502</deviceid>
        <deviceid>1503</deviceid>
        <deviceid>153A</deviceid>
        <deviceid>153B</deviceid>
        <deviceid>155A</deviceid>
        <deviceid>1559</deviceid>
        <deviceid>15A0</deviceid>
        <deviceid>15A1</deviceid>
        <deviceid>15A2</deviceid>
        <deviceid>15A3</deviceid>
        <deviceid>156F</deviceid>
        <deviceid>1570</deviceid>
        <deviceid>15B7</deviceid>
        <deviceid>15B8</deviceid>
        <deviceid>15B9</deviceid>
        <deviceid>15D7</deviceid>
        <deviceid>15D8</deviceid>
        <deviceid>15E3</deviceid>
        <deviceid>15D6</deviceid>
        <deviceid>15BD</deviceid>
        <deviceid>15BE</deviceid>
        <deviceid>15BB</deviceid>
        <deviceid>15BC</deviceid>
        <deviceid>15DF</deviceid>
        <deviceid>15E0</deviceid>
        <deviceid>15E1</deviceid>
        <deviceid>15E2</deviceid>
        <deviceid>10C9</deviceid>
        <deviceid>10E6</deviceid>
        <deviceid>10E7</deviceid>
        <deviceid>10E8</deviceid>
        <deviceid>1526</deviceid>
        <deviceid>150A</deviceid>
        <deviceid>1518</deviceid>
        <deviceid>150D</deviceid>
        <deviceid>10A7</deviceid>
        <deviceid>10A9</deviceid>
        <deviceid>10D6</deviceid>
        <deviceid>150E</deviceid>
        <deviceid>150F</deviceid>
        <deviceid>1510</deviceid>
        <deviceid>1511</deviceid>
        <deviceid>1516</deviceid>
        <deviceid>1527</deviceid>
        <deviceid>1521</deviceid>
        <deviceid>1522</deviceid>
```

```
        <deviceid>1523</deviceid>
        <deviceid>1524</deviceid>
        <deviceid>1546</deviceid>
        <deviceid>1533</deviceid>
        <deviceid>1534</deviceid>
        <deviceid>1535</deviceid>
        <deviceid>1536</deviceid>
        <deviceid>1537</deviceid>
        <deviceid>1538</deviceid>
        <deviceid>157B</deviceid>
        <deviceid>157C</deviceid>
        <deviceid>1539</deviceid>
        <deviceid>1F40</deviceid>
        <deviceid>1F41</deviceid>
        <deviceid>1F45</deviceid>
        <deviceid>0438</deviceid>
        <deviceid>043A</deviceid>
        <deviceid>043C</deviceid>
        <deviceid>0440</deviceid>
        <deviceid>10C6</deviceid>
        <deviceid>10C7</deviceid>
        <deviceid>10C8</deviceid>
        <deviceid>150B</deviceid>
        <deviceid>10DB</deviceid>
        <deviceid>10DD</deviceid>
        <deviceid>10EC</deviceid>
        <deviceid>10F1</deviceid>
        <deviceid>10E1</deviceid>
        <deviceid>10F4</deviceid>
        <deviceid>10F7</deviceid>
        <deviceid>1514</deviceid>
        <deviceid>1517</deviceid>
        <deviceid>10F8</deviceid>
        <deviceid>000C</deviceid>
        <deviceid>1F63</deviceid>
        <deviceid>10F9</deviceid>
        <deviceid>10FB</deviceid>
        <deviceid>11A9</deviceid>
        <deviceid>1071</deviceid>
        <deviceid>1F72</deviceid>
        <deviceid>17D0</deviceid>
        <deviceid>0470</deviceid>
        <deviceid>211B</deviceid>
        <deviceid>8976</deviceid>
        <deviceid>2159</deviceid>
        <deviceid>000D</deviceid>
        <deviceid>0008</deviceid>
        <deviceid>152A</deviceid>
        <deviceid>1529</deviceid>
        <deviceid>1507</deviceid>
        <deviceid>154D</deviceid>
        <deviceid>154A</deviceid>
        <deviceid>1558</deviceid>
        <deviceid>1557</deviceid>
        <deviceid>0001</deviceid>
```

```
      <deviceid>10FC</deviceid>
      <deviceid>151C</deviceid>
      <deviceid>1528</deviceid>
      <deviceid>1560</deviceid>
      <deviceid>1563</deviceid>
      <deviceid>ABCD</deviceid>
      <deviceid>15AA</deviceid>
      <deviceid>15AB</deviceid>
      <deviceid>15AC</deviceid>
      <deviceid>15AD</deviceid>
      <deviceid>15AE</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10EC</manufacturer>
      <deviceid>8136</deviceid>
      <deviceid>8137</deviceid>
      <deviceid>8168</deviceid>
      <deviceid>8167</deviceid>
      <deviceid>8169</deviceid>
      <deviceid>8166</deviceid>
      <deviceid>8161</deviceid>
      <deviceid>8125</deviceid>
      <deviceid>8225</deviceid>
      <deviceid>2502</deviceid>
      <deviceid>2600</deviceid>
      <deviceid>3000</deviceid>
  </NIC>

  <NIC>
    <manufacturer>14E4</manufacturer>
      <deviceid>1644</deviceid>
      <deviceid>1645</deviceid>
      <deviceid>1646</deviceid>
      <deviceid>16A6</deviceid>
      <deviceid>16C6</deviceid>
      <deviceid>1647</deviceid>
      <deviceid>16A7</deviceid>
      <deviceid>16C7</deviceid>
      <deviceid>164D</deviceid>
      <deviceid>1648</deviceid>
      <deviceid>16A8</deviceid>
      <deviceid>1653</deviceid>
      <deviceid>166E</deviceid>
      <deviceid>1654</deviceid>
      <deviceid>165D</deviceid>
      <deviceid>165E</deviceid>
      <deviceid>166D</deviceid>
      <deviceid>170D</deviceid>
      <deviceid>170E</deviceid>
      <deviceid>1696</deviceid>
      <deviceid>1676</deviceid>
      <deviceid>1677</deviceid>
      <deviceid>1659</deviceid>
      <deviceid>167C</deviceid>
```

```
        <deviceid>167D</deviceid>
        <deviceid>169D</deviceid>
        <deviceid>16F7</deviceid>
        <deviceid>16FD</deviceid>
        <deviceid>16FE</deviceid>
        <deviceid>16DD</deviceid>
        <deviceid>1668</deviceid>
        <deviceid>1669</deviceid>
        <deviceid>1678</deviceid>
        <deviceid>1679</deviceid>
        <deviceid>1600</deviceid>
        <deviceid>1601</deviceid>
        <deviceid>166A</deviceid>
        <deviceid>166B</deviceid>
        <deviceid>16FF</deviceid>
        <deviceid>170F</deviceid>
        <deviceid>169B</deviceid>
        <deviceid>1693</deviceid>
        <deviceid>167F</deviceid>
        <deviceid>169A</deviceid>
        <deviceid>1698</deviceid>
        <deviceid>1692</deviceid>
        <deviceid>1694</deviceid>
        <deviceid>1690</deviceid>
        <deviceid>1691</deviceid>
        <deviceid>1699</deviceid>
        <deviceid>16A0</deviceid>
        <deviceid>167B</deviceid>
        <deviceid>1673</deviceid>
        <deviceid>165A</deviceid>
        <deviceid>1674</deviceid>
        <deviceid>1681</deviceid>
        <deviceid>1680</deviceid>
        <deviceid>1688</deviceid>
        <deviceid>167A</deviceid>
        <deviceid>1672</deviceid>
        <deviceid>1684</deviceid>
        <deviceid>165B</deviceid>
        <deviceid>16B1</deviceid>
        <deviceid>16B5</deviceid>
        <deviceid>16B0</deviceid>
        <deviceid>16B4</deviceid>
        <deviceid>16B2</deviceid>
        <deviceid>16B6</deviceid>
        <deviceid>1682</deviceid>
        <deviceid>1686</deviceid>
        <deviceid>16B3</deviceid>
        <deviceid>16B7</deviceid>
        <deviceid>1655</deviceid>
        <deviceid>1665</deviceid>
        <deviceid>1656</deviceid>
        <deviceid>1657</deviceid>
        <deviceid>165F</deviceid>
        <deviceid>165C</deviceid>
        <deviceid>1683</deviceid>
```

```
      <deviceid>1641</deviceid>
      <deviceid>1642</deviceid>
      <deviceid>1643</deviceid>
      <deviceid>1687</deviceid>
      <deviceid>164A</deviceid>
      <deviceid>16AA</deviceid>
      <deviceid>164C</deviceid>
      <deviceid>16AC</deviceid>
      <deviceid>1639</deviceid>
      <deviceid>163A</deviceid>
      <deviceid>163B</deviceid>
      <deviceid>163C</deviceid>
      <deviceid>164E</deviceid>
      <deviceid>164F</deviceid>
      <deviceid>1650</deviceid>
      <deviceid>1662</deviceid>
      <deviceid>1663</deviceid>
      <deviceid>168A</deviceid>
      <deviceid>168D</deviceid>
      <deviceid>16A1</deviceid>
      <deviceid>16A2</deviceid>
      <deviceid>168E</deviceid>
      <deviceid>163D</deviceid>
      <deviceid>16A5</deviceid>
      <deviceid>16A4</deviceid>
      <deviceid>16AE</deviceid>
      <deviceid>163E</deviceid>
  </NIC>

  <NIC>
    <manufacturer>1969</manufacturer>
      <deviceid>1062</deviceid>
      <deviceid>1063</deviceid>
      <deviceid>2060</deviceid>
      <deviceid>2062</deviceid>
      <deviceid>1073</deviceid>
      <deviceid>1083</deviceid>
      <deviceid>1090</deviceid>
      <deviceid>1091</deviceid>
      <deviceid>E091</deviceid>
      <deviceid>10A0</deviceid>
      <deviceid>10A1</deviceid>
      <deviceid>E0A1</deviceid>
      <deviceid>10B0</deviceid>
      <deviceid>10B1</deviceid>
      <deviceid>E0B1</deviceid>
      <deviceid>10C0</deviceid>
      <deviceid>10C1</deviceid>
      <deviceid>E0C1</deviceid>
      <deviceid>10D0</deviceid>
      <deviceid>10D1</deviceid>
      <deviceid>E0D1</deviceid>
      <deviceid>10E0</deviceid>
      <deviceid>10E1</deviceid>
      <deviceid>E0E1</deviceid>
```

```
      <deviceid>10F0</deviceid>
      <deviceid>10F1</deviceid>
      <deviceid>E0F1</deviceid>
  </NIC>

  <NIC>
    <manufacturer>19A2</manufacturer>
      <deviceid>0211</deviceid>
      <deviceid>0215</deviceid>
      <deviceid>0221</deviceid>
      <deviceid>0700</deviceid>
      <deviceid>0710</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10DF</manufacturer>
      <deviceid>0720</deviceid>
      <deviceid>E220</deviceid>
  </NIC>

  <NIC>
    <manufacturer>15B3</manufacturer>
      <deviceid>1000</deviceid>
      <deviceid>1001</deviceid>
      <deviceid>1002</deviceid>
      <deviceid>1003</deviceid>
      <deviceid>1004</deviceid>
      <deviceid>1005</deviceid>
      <deviceid>1006</deviceid>
      <deviceid>1007</deviceid>
      <deviceid>1008</deviceid>
      <deviceid>1009</deviceid>
      <deviceid>100A</deviceid>
      <deviceid>100B</deviceid>
      <deviceid>100C</deviceid>
      <deviceid>100D</deviceid>
      <deviceid>100E</deviceid>
      <deviceid>100F</deviceid>
      <deviceid>1010</deviceid>
      <deviceid>6340</deviceid>
      <deviceid>6341</deviceid>
      <deviceid>634A</deviceid>
      <deviceid>634B</deviceid>
      <deviceid>6354</deviceid>
      <deviceid>6368</deviceid>
      <deviceid>6369</deviceid>
      <deviceid>6372</deviceid>
      <deviceid>6732</deviceid>
      <deviceid>6733</deviceid>
      <deviceid>673C</deviceid>
      <deviceid>673D</deviceid>
      <deviceid>6746</deviceid>
      <deviceid>6750</deviceid>
      <deviceid>6751</deviceid>
      <deviceid>675A</deviceid>
```

```
        <deviceid>6764</deviceid>
        <deviceid>6765</deviceid>
        <deviceid>676E</deviceid>
        <deviceid>6778</deviceid>
        <deviceid>1013</deviceid>
        <deviceid>1015</deviceid>
        <deviceid>1017</deviceid>
        <deviceid>1019</deviceid>
        <deviceid>101B</deviceid>
        <deviceid>101D</deviceid>
        <deviceid>101F</deviceid>
        <deviceid>1021</deviceid>
        <deviceid>1023</deviceid>
        <deviceid>1025</deviceid>
        <deviceid>1027</deviceid>
        <deviceid>1029</deviceid>
        <deviceid>102B</deviceid>
        <deviceid>102F</deviceid>
    </NIC>

    <NIC>
      <manufacturer>1137</manufacturer>
        <deviceid>0043</deviceid>
    </NIC>

</SupportedNetworkInterfaceCards>
```

另请参阅

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

Windows 8 中的网络内核调试支持的以太网 NIC

Windows 8.1 中的网络内核调试支持的以太网 NIC

支持在 Windows 10 版本 1809 (Redstone 5) 内部版本 17763 中进行网络内核调试的以太网 NIC

项目 • 2024/02/06

若要通过以太网网线进行内核调试，目标计算机必须具有受支持的网络接口卡 (NIC) 。

在内核调试期间，运行调试器的计算机称为 *主计算机*，正在调试的计算机称为 *目标计算机*。 有关详细信息，请参阅 设置 KDNET 网络内核自动调试。

有关支持的网络适配器的一般信息，请参阅支持在 Windows 10 中进行网络内核调试的以太网 NIC。

版本信息

支持的适配器列表适用于以下版本的 Windows

- Windows 10 版本 1809 (Redstone 5) 内部版本 17763

XML

```
<?xml version="1.0" encoding="utf-8"?>
<SupportedNetworkInterfaceCards>
  <NIC>
    <manufacturer>8086</manufacturer>
      <deviceid>1000</deviceid>
      <deviceid>1001</deviceid>
      <deviceid>1004</deviceid>
      <deviceid>1008</deviceid>
      <deviceid>1009</deviceid>
      <deviceid>100C</deviceid>
      <deviceid>100D</deviceid>
      <deviceid>100E</deviceid>
      <deviceid>1015</deviceid>
      <deviceid>1016</deviceid>
      <deviceid>1017</deviceid>
      <deviceid>101E</deviceid>
      <deviceid>100F</deviceid>
      <deviceid>1011</deviceid>
      <deviceid>1026</deviceid>
      <deviceid>1027</deviceid>
      <deviceid>1028</deviceid>
      <deviceid>1010</deviceid>
      <deviceid>1012</deviceid>
      <deviceid>101D</deviceid>
      <deviceid>1079</deviceid>
      <deviceid>107A</deviceid>
```

```
        <deviceid>107B</deviceid>
        <deviceid>108A</deviceid>
        <deviceid>1099</deviceid>
        <deviceid>10B5</deviceid>
        <deviceid>1013</deviceid>
        <deviceid>1018</deviceid>
        <deviceid>1014</deviceid>
        <deviceid>1078</deviceid>
        <deviceid>1076</deviceid>
        <deviceid>107C</deviceid>
        <deviceid>1077</deviceid>
        <deviceid>1019</deviceid>
        <deviceid>101A</deviceid>
        <deviceid>1075</deviceid>
        <deviceid>105E</deviceid>
        <deviceid>105F</deviceid>
        <deviceid>1060</deviceid>
        <deviceid>10D9</deviceid>
        <deviceid>10DA</deviceid>
        <deviceid>10A4</deviceid>
        <deviceid>10D5</deviceid>
        <deviceid>10A5</deviceid>
        <deviceid>10BC</deviceid>
        <deviceid>107D</deviceid>
        <deviceid>107E</deviceid>
        <deviceid>107F</deviceid>
        <deviceid>10B9</deviceid>
        <deviceid>108B</deviceid>
        <deviceid>108C</deviceid>
        <deviceid>109A</deviceid>
        <deviceid>10D3</deviceid>
        <deviceid>10F6</deviceid>
        <deviceid>150C</deviceid>
        <deviceid>1096</deviceid>
        <deviceid>1098</deviceid>
        <deviceid>10BA</deviceid>
        <deviceid>10BB</deviceid>
        <deviceid>1501</deviceid>
        <deviceid>1049</deviceid>
        <deviceid>104A</deviceid>
        <deviceid>104B</deviceid>
        <deviceid>104C</deviceid>
        <deviceid>10C4</deviceid>
        <deviceid>10C5</deviceid>
        <deviceid>104D</deviceid>
        <deviceid>10BF</deviceid>
        <deviceid>10F5</deviceid>
        <deviceid>10CB</deviceid>
        <deviceid>10BD</deviceid>
        <deviceid>10E5</deviceid>
        <deviceid>294C</deviceid>
        <deviceid>10C0</deviceid>
        <deviceid>10C3</deviceid>
        <deviceid>10C2</deviceid>
        <deviceid>10CC</deviceid>
```

```
        <deviceid>10CD</deviceid>
        <deviceid>10CE</deviceid>
        <deviceid>10DE</deviceid>
        <deviceid>10DF</deviceid>
        <deviceid>1525</deviceid>
        <deviceid>10EA</deviceid>
        <deviceid>10EB</deviceid>
        <deviceid>10EF</deviceid>
        <deviceid>10F0</deviceid>
        <deviceid>1502</deviceid>
        <deviceid>1503</deviceid>
        <deviceid>153A</deviceid>
        <deviceid>153B</deviceid>
        <deviceid>155A</deviceid>
        <deviceid>1559</deviceid>
        <deviceid>15A0</deviceid>
        <deviceid>15A1</deviceid>
        <deviceid>15A2</deviceid>
        <deviceid>15A3</deviceid>
        <deviceid>156F</deviceid>
        <deviceid>1570</deviceid>
        <deviceid>15B7</deviceid>
        <deviceid>15B8</deviceid>
        <deviceid>15B9</deviceid>
        <deviceid>15D7</deviceid>
        <deviceid>15D8</deviceid>
        <deviceid>15E3</deviceid>
        <deviceid>15D6</deviceid>
        <deviceid>15BD</deviceid>
        <deviceid>15BE</deviceid>
        <deviceid>15BB</deviceid>
        <deviceid>15BC</deviceid>
        <deviceid>15DF</deviceid>
        <deviceid>15E0</deviceid>
        <deviceid>15E1</deviceid>
        <deviceid>15E2</deviceid>
        <deviceid>10C9</deviceid>
        <deviceid>10E6</deviceid>
        <deviceid>10E7</deviceid>
        <deviceid>10E8</deviceid>
        <deviceid>1526</deviceid>
        <deviceid>150A</deviceid>
        <deviceid>1518</deviceid>
        <deviceid>150D</deviceid>
        <deviceid>10A7</deviceid>
        <deviceid>10A9</deviceid>
        <deviceid>10D6</deviceid>
        <deviceid>150E</deviceid>
        <deviceid>150F</deviceid>
        <deviceid>1510</deviceid>
        <deviceid>1511</deviceid>
        <deviceid>1516</deviceid>
        <deviceid>1527</deviceid>
        <deviceid>1521</deviceid>
        <deviceid>1522</deviceid>
```

```
        <deviceid>1523</deviceid>
        <deviceid>1524</deviceid>
        <deviceid>1546</deviceid>
        <deviceid>1533</deviceid>
        <deviceid>1534</deviceid>
        <deviceid>1535</deviceid>
        <deviceid>1536</deviceid>
        <deviceid>1537</deviceid>
        <deviceid>1538</deviceid>
        <deviceid>157B</deviceid>
        <deviceid>157C</deviceid>
        <deviceid>1539</deviceid>
        <deviceid>1F40</deviceid>
        <deviceid>1F41</deviceid>
        <deviceid>1F45</deviceid>
        <deviceid>0438</deviceid>
        <deviceid>043A</deviceid>
        <deviceid>043C</deviceid>
        <deviceid>0440</deviceid>
        <deviceid>10C6</deviceid>
        <deviceid>10C7</deviceid>
        <deviceid>10C8</deviceid>
        <deviceid>150B</deviceid>
        <deviceid>10DB</deviceid>
        <deviceid>10DD</deviceid>
        <deviceid>10EC</deviceid>
        <deviceid>10F1</deviceid>
        <deviceid>10E1</deviceid>
        <deviceid>10F4</deviceid>
        <deviceid>10F7</deviceid>
        <deviceid>1514</deviceid>
        <deviceid>1517</deviceid>
        <deviceid>10F8</deviceid>
        <deviceid>000C</deviceid>
        <deviceid>1F63</deviceid>
        <deviceid>10F9</deviceid>
        <deviceid>10FB</deviceid>
        <deviceid>11A9</deviceid>
        <deviceid>1071</deviceid>
        <deviceid>1F72</deviceid>
        <deviceid>17D0</deviceid>
        <deviceid>0470</deviceid>
        <deviceid>211B</deviceid>
        <deviceid>8976</deviceid>
        <deviceid>2159</deviceid>
        <deviceid>000D</deviceid>
        <deviceid>0008</deviceid>
        <deviceid>152A</deviceid>
        <deviceid>1529</deviceid>
        <deviceid>1507</deviceid>
        <deviceid>154D</deviceid>
        <deviceid>154A</deviceid>
        <deviceid>1558</deviceid>
        <deviceid>1557</deviceid>
        <deviceid>0001</deviceid>
```

```
      <deviceid>10FC</deviceid>
      <deviceid>151C</deviceid>
      <deviceid>1528</deviceid>
      <deviceid>1560</deviceid>
      <deviceid>1563</deviceid>
      <deviceid>ABCD</deviceid>
      <deviceid>15AA</deviceid>
      <deviceid>15AB</deviceid>
      <deviceid>15AC</deviceid>
      <deviceid>15AD</deviceid>
      <deviceid>15AE</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10EC</manufacturer>
      <deviceid>8136</deviceid>
      <deviceid>8137</deviceid>
      <deviceid>8168</deviceid>
      <deviceid>8167</deviceid>
      <deviceid>8169</deviceid>
      <deviceid>8166</deviceid>
  </NIC>

  <NIC>
    <manufacturer>14E4</manufacturer>
      <deviceid>1644</deviceid>
      <deviceid>1645</deviceid>
      <deviceid>1646</deviceid>
      <deviceid>16A6</deviceid>
      <deviceid>16C6</deviceid>
      <deviceid>1647</deviceid>
      <deviceid>16A7</deviceid>
      <deviceid>16C7</deviceid>
      <deviceid>164D</deviceid>
      <deviceid>1648</deviceid>
      <deviceid>16A8</deviceid>
      <deviceid>1653</deviceid>
      <deviceid>166E</deviceid>
      <deviceid>1654</deviceid>
      <deviceid>165D</deviceid>
      <deviceid>165E</deviceid>
      <deviceid>166D</deviceid>
      <deviceid>170D</deviceid>
      <deviceid>170E</deviceid>
      <deviceid>1696</deviceid>
      <deviceid>1676</deviceid>
      <deviceid>1677</deviceid>
      <deviceid>1659</deviceid>
      <deviceid>167C</deviceid>
      <deviceid>167D</deviceid>
      <deviceid>169D</deviceid>
      <deviceid>16F7</deviceid>
      <deviceid>16FD</deviceid>
      <deviceid>16FE</deviceid>
      <deviceid>16DD</deviceid>
```

```
        <deviceid>1668</deviceid>
        <deviceid>1669</deviceid>
        <deviceid>1678</deviceid>
        <deviceid>1679</deviceid>
        <deviceid>1600</deviceid>
        <deviceid>1601</deviceid>
        <deviceid>166A</deviceid>
        <deviceid>166B</deviceid>
        <deviceid>16FF</deviceid>
        <deviceid>170F</deviceid>
        <deviceid>169B</deviceid>
        <deviceid>1693</deviceid>
        <deviceid>167F</deviceid>
        <deviceid>169A</deviceid>
        <deviceid>1698</deviceid>
        <deviceid>1692</deviceid>
        <deviceid>1694</deviceid>
        <deviceid>1690</deviceid>
        <deviceid>1691</deviceid>
        <deviceid>1699</deviceid>
        <deviceid>16A0</deviceid>
        <deviceid>167B</deviceid>
        <deviceid>1673</deviceid>
        <deviceid>165A</deviceid>
        <deviceid>1674</deviceid>
        <deviceid>1681</deviceid>
        <deviceid>1680</deviceid>
        <deviceid>1688</deviceid>
        <deviceid>167A</deviceid>
        <deviceid>1672</deviceid>
        <deviceid>1684</deviceid>
        <deviceid>165B</deviceid>
        <deviceid>16B1</deviceid>
        <deviceid>16B5</deviceid>
        <deviceid>16B0</deviceid>
        <deviceid>16B4</deviceid>
        <deviceid>16B2</deviceid>
        <deviceid>16B6</deviceid>
        <deviceid>1682</deviceid>
        <deviceid>1686</deviceid>
        <deviceid>16B3</deviceid>
        <deviceid>16B7</deviceid>
        <deviceid>1655</deviceid>
        <deviceid>1665</deviceid>
        <deviceid>1656</deviceid>
        <deviceid>1657</deviceid>
        <deviceid>165F</deviceid>
        <deviceid>165C</deviceid>
        <deviceid>1683</deviceid>
        <deviceid>1641</deviceid>
        <deviceid>1642</deviceid>
        <deviceid>1643</deviceid>
        <deviceid>1687</deviceid>
        <deviceid>164A</deviceid>
        <deviceid>16AA</deviceid>
```

```
      <deviceid>164C</deviceid>
      <deviceid>16AC</deviceid>
      <deviceid>1639</deviceid>
      <deviceid>163A</deviceid>
      <deviceid>163B</deviceid>
      <deviceid>163C</deviceid>
      <deviceid>164E</deviceid>
      <deviceid>164F</deviceid>
      <deviceid>1650</deviceid>
      <deviceid>1662</deviceid>
      <deviceid>1663</deviceid>
      <deviceid>168A</deviceid>
      <deviceid>168D</deviceid>
      <deviceid>16A1</deviceid>
      <deviceid>16A2</deviceid>
      <deviceid>168E</deviceid>
      <deviceid>163D</deviceid>
      <deviceid>16A5</deviceid>
      <deviceid>16A4</deviceid>
      <deviceid>16AE</deviceid>
      <deviceid>163E</deviceid>
  </NIC>

  <NIC>
    <manufacturer>1969</manufacturer>
      <deviceid>1062</deviceid>
      <deviceid>1063</deviceid>
      <deviceid>2060</deviceid>
      <deviceid>2062</deviceid>
      <deviceid>1073</deviceid>
      <deviceid>1083</deviceid>
      <deviceid>1090</deviceid>
      <deviceid>1091</deviceid>
      <deviceid>E091</deviceid>
      <deviceid>10A0</deviceid>
      <deviceid>10A1</deviceid>
      <deviceid>E0A1</deviceid>
      <deviceid>10B0</deviceid>
      <deviceid>10B1</deviceid>
      <deviceid>E0B1</deviceid>
      <deviceid>10C0</deviceid>
      <deviceid>10C1</deviceid>
      <deviceid>E0C1</deviceid>
      <deviceid>10D0</deviceid>
      <deviceid>10D1</deviceid>
      <deviceid>E0D1</deviceid>
      <deviceid>10E0</deviceid>
      <deviceid>10E1</deviceid>
      <deviceid>E0E1</deviceid>
      <deviceid>10F0</deviceid>
      <deviceid>10F1</deviceid>
      <deviceid>E0F1</deviceid>
  </NIC>

  <NIC>
```

```
    <manufacturer>19A2</manufacturer>
      <deviceid>0211</deviceid>
      <deviceid>0215</deviceid>
      <deviceid>0221</deviceid>
      <deviceid>0700</deviceid>
      <deviceid>0710</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10DF</manufacturer>
      <deviceid>0720</deviceid>
      <deviceid>E220</deviceid>
  </NIC>

  <NIC>
    <manufacturer>15B3</manufacturer>
      <deviceid>1000</deviceid>
      <deviceid>1001</deviceid>
      <deviceid>1002</deviceid>
      <deviceid>1003</deviceid>
      <deviceid>1004</deviceid>
      <deviceid>1005</deviceid>
      <deviceid>1006</deviceid>
      <deviceid>1007</deviceid>
      <deviceid>1008</deviceid>
      <deviceid>1009</deviceid>
      <deviceid>100A</deviceid>
      <deviceid>100B</deviceid>
      <deviceid>100C</deviceid>
      <deviceid>100D</deviceid>
      <deviceid>100E</deviceid>
      <deviceid>100F</deviceid>
      <deviceid>1010</deviceid>
      <deviceid>6340</deviceid>
      <deviceid>6341</deviceid>
      <deviceid>634A</deviceid>
      <deviceid>634B</deviceid>
      <deviceid>6354</deviceid>
      <deviceid>6368</deviceid>
      <deviceid>6369</deviceid>
      <deviceid>6372</deviceid>
      <deviceid>6732</deviceid>
      <deviceid>6733</deviceid>
      <deviceid>673C</deviceid>
      <deviceid>673D</deviceid>
      <deviceid>6746</deviceid>
      <deviceid>6750</deviceid>
      <deviceid>6751</deviceid>
      <deviceid>675A</deviceid>
      <deviceid>6764</deviceid>
      <deviceid>6765</deviceid>
      <deviceid>676E</deviceid>
      <deviceid>6778</deviceid>
      <deviceid>1013</deviceid>
      <deviceid>1015</deviceid>
```

```
      <deviceid>1017</deviceid>
      <deviceid>1019</deviceid>
      <deviceid>101B</deviceid>
      <deviceid>101D</deviceid>
      <deviceid>101F</deviceid>
      <deviceid>1021</deviceid>
      <deviceid>1023</deviceid>
      <deviceid>1025</deviceid>
      <deviceid>1027</deviceid>
      <deviceid>1029</deviceid>
      <deviceid>102B</deviceid>
      <deviceid>102F</deviceid>
  </NIC>

  <NIC>
    <manufacturer>1137</manufacturer>
      <deviceid>0043</deviceid>
  </NIC>

</SupportedNetworkInterfaceCards>
```

另请参阅

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

Windows 8 中的网络内核调试支持的以太网 NIC

Windows 8.1 中的网络内核调试支持的以太网 NIC

Windows 10版本 1803 (Redstone 4) 内部版本 17134 中支持用于网络内核调试的以太网 NIC

项目 • 2023/04/14

若要通过以太网网线进行内核调试，目标计算机必须具有受支持的网络接口卡 (NIC) 。

在内核调试期间，运行调试器的计算机称为 *主计算机*，正在调试的计算机称为 *目标计算机*。 有关详细信息，请参阅 自动设置 KDNET 网络内核调试。

有关支持的网络适配器的一般信息，请参阅 Windows 10 中支持的网络内核调试的以太网 NIC。

版本信息

支持的适配器列表适用于以下版本的 Windows。

- Windows 10版本 1803 (Redstone 4) 内部版本 17134

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<SupportedNetworkInterfaceCards>
  <NIC>
    <manufacturer>8086</manufacturer>
      <deviceid>1000</deviceid>
      <deviceid>1001</deviceid>
      <deviceid>1004</deviceid>
      <deviceid>1008</deviceid>
      <deviceid>1009</deviceid>
      <deviceid>100C</deviceid>
      <deviceid>100D</deviceid>
      <deviceid>100E</deviceid>
      <deviceid>1015</deviceid>
      <deviceid>1016</deviceid>
      <deviceid>1017</deviceid>
      <deviceid>101E</deviceid>
      <deviceid>100F</deviceid>
      <deviceid>1011</deviceid>
      <deviceid>1026</deviceid>
      <deviceid>1027</deviceid>
      <deviceid>1028</deviceid>
      <deviceid>1010</deviceid>
      <deviceid>1012</deviceid>
      <deviceid>101D</deviceid>
      <deviceid>1079</deviceid>
      <deviceid>107A</deviceid>
```

```
        <deviceid>107B</deviceid>
        <deviceid>108A</deviceid>
        <deviceid>1099</deviceid>
        <deviceid>10B5</deviceid>
        <deviceid>1013</deviceid>
        <deviceid>1018</deviceid>
        <deviceid>1014</deviceid>
        <deviceid>1078</deviceid>
        <deviceid>1076</deviceid>
        <deviceid>107C</deviceid>
        <deviceid>1077</deviceid>
        <deviceid>1019</deviceid>
        <deviceid>101A</deviceid>
        <deviceid>1075</deviceid>
        <deviceid>105E</deviceid>
        <deviceid>105F</deviceid>
        <deviceid>1060</deviceid>
        <deviceid>10D9</deviceid>
        <deviceid>10DA</deviceid>
        <deviceid>10A4</deviceid>
        <deviceid>10D5</deviceid>
        <deviceid>10A5</deviceid>
        <deviceid>10BC</deviceid>
        <deviceid>107D</deviceid>
        <deviceid>107E</deviceid>
        <deviceid>107F</deviceid>
        <deviceid>10B9</deviceid>
        <deviceid>108B</deviceid>
        <deviceid>108C</deviceid>
        <deviceid>109A</deviceid>
        <deviceid>10D3</deviceid>
        <deviceid>10F6</deviceid>
        <deviceid>150C</deviceid>
        <deviceid>1096</deviceid>
        <deviceid>1098</deviceid>
        <deviceid>10BA</deviceid>
        <deviceid>10BB</deviceid>
        <deviceid>1501</deviceid>
        <deviceid>1049</deviceid>
        <deviceid>104A</deviceid>
        <deviceid>104B</deviceid>
        <deviceid>104C</deviceid>
        <deviceid>10C4</deviceid>
        <deviceid>10C5</deviceid>
        <deviceid>104D</deviceid>
        <deviceid>10BF</deviceid>
        <deviceid>10F5</deviceid>
        <deviceid>10CB</deviceid>
        <deviceid>10BD</deviceid>
        <deviceid>10E5</deviceid>
        <deviceid>294C</deviceid>
        <deviceid>10C0</deviceid>
        <deviceid>10C3</deviceid>
        <deviceid>10C2</deviceid>
        <deviceid>10CC</deviceid>
```

```
        <deviceid>10CD</deviceid>
        <deviceid>10CE</deviceid>
        <deviceid>10DE</deviceid>
        <deviceid>10DF</deviceid>
        <deviceid>1525</deviceid>
        <deviceid>10EA</deviceid>
        <deviceid>10EB</deviceid>
        <deviceid>10EF</deviceid>
        <deviceid>10F0</deviceid>
        <deviceid>1502</deviceid>
        <deviceid>1503</deviceid>
        <deviceid>153A</deviceid>
        <deviceid>153B</deviceid>
        <deviceid>155A</deviceid>
        <deviceid>1559</deviceid>
        <deviceid>15A0</deviceid>
        <deviceid>15A1</deviceid>
        <deviceid>15A2</deviceid>
        <deviceid>15A3</deviceid>
        <deviceid>156F</deviceid>
        <deviceid>1570</deviceid>
        <deviceid>15B7</deviceid>
        <deviceid>15B8</deviceid>
        <deviceid>15B9</deviceid>
        <deviceid>15D7</deviceid>
        <deviceid>15D8</deviceid>
        <deviceid>15E3</deviceid>
        <deviceid>15D6</deviceid>
        <deviceid>15BD</deviceid>
        <deviceid>15BE</deviceid>
        <deviceid>15BB</deviceid>
        <deviceid>15BC</deviceid>
        <deviceid>15DF</deviceid>
        <deviceid>15E0</deviceid>
        <deviceid>15E1</deviceid>
        <deviceid>15E2</deviceid>
        <deviceid>10C9</deviceid>
        <deviceid>10E6</deviceid>
        <deviceid>10E7</deviceid>
        <deviceid>10E8</deviceid>
        <deviceid>1526</deviceid>
        <deviceid>150A</deviceid>
        <deviceid>1518</deviceid>
        <deviceid>150D</deviceid>
        <deviceid>10A7</deviceid>
        <deviceid>10A9</deviceid>
        <deviceid>10D6</deviceid>
        <deviceid>150E</deviceid>
        <deviceid>150F</deviceid>
        <deviceid>1510</deviceid>
        <deviceid>1511</deviceid>
        <deviceid>1516</deviceid>
        <deviceid>1527</deviceid>
        <deviceid>1521</deviceid>
        <deviceid>1522</deviceid>
```

```
        <deviceid>1523</deviceid>
        <deviceid>1524</deviceid>
        <deviceid>1546</deviceid>
        <deviceid>1533</deviceid>
        <deviceid>1534</deviceid>
        <deviceid>1535</deviceid>
        <deviceid>1536</deviceid>
        <deviceid>1537</deviceid>
        <deviceid>1538</deviceid>
        <deviceid>157B</deviceid>
        <deviceid>157C</deviceid>
        <deviceid>1539</deviceid>
        <deviceid>1F40</deviceid>
        <deviceid>1F41</deviceid>
        <deviceid>1F45</deviceid>
        <deviceid>0438</deviceid>
        <deviceid>043A</deviceid>
        <deviceid>043C</deviceid>
        <deviceid>0440</deviceid>
        <deviceid>10C6</deviceid>
        <deviceid>10C7</deviceid>
        <deviceid>10C8</deviceid>
        <deviceid>150B</deviceid>
        <deviceid>10DB</deviceid>
        <deviceid>10DD</deviceid>
        <deviceid>10EC</deviceid>
        <deviceid>10F1</deviceid>
        <deviceid>10E1</deviceid>
        <deviceid>10F4</deviceid>
        <deviceid>10F7</deviceid>
        <deviceid>1514</deviceid>
        <deviceid>1517</deviceid>
        <deviceid>10F8</deviceid>
        <deviceid>000C</deviceid>
        <deviceid>1F63</deviceid>
        <deviceid>10F9</deviceid>
        <deviceid>10FB</deviceid>
        <deviceid>11A9</deviceid>
        <deviceid>1071</deviceid>
        <deviceid>1F72</deviceid>
        <deviceid>17D0</deviceid>
        <deviceid>0470</deviceid>
        <deviceid>211B</deviceid>
        <deviceid>8976</deviceid>
        <deviceid>2159</deviceid>
        <deviceid>000D</deviceid>
        <deviceid>0008</deviceid>
        <deviceid>152A</deviceid>
        <deviceid>1529</deviceid>
        <deviceid>1507</deviceid>
        <deviceid>154D</deviceid>
        <deviceid>154A</deviceid>
        <deviceid>1558</deviceid>
        <deviceid>1557</deviceid>
        <deviceid>0001</deviceid>
```

```
      <deviceid>10FC</deviceid>
      <deviceid>151C</deviceid>
      <deviceid>1528</deviceid>
      <deviceid>1560</deviceid>
      <deviceid>1563</deviceid>
      <deviceid>ABCD</deviceid>
      <deviceid>15AA</deviceid>
      <deviceid>15AB</deviceid>
      <deviceid>15AC</deviceid>
      <deviceid>15AD</deviceid>
      <deviceid>15AE</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10EC</manufacturer>
      <deviceid>8136</deviceid>
      <deviceid>8137</deviceid>
      <deviceid>8168</deviceid>
      <deviceid>8167</deviceid>
      <deviceid>8169</deviceid>
      <deviceid>8166</deviceid>
  </NIC>

  <NIC>
    <manufacturer>14E4</manufacturer>
      <deviceid>1644</deviceid>
      <deviceid>1645</deviceid>
      <deviceid>1646</deviceid>
      <deviceid>16A6</deviceid>
      <deviceid>16C6</deviceid>
      <deviceid>1647</deviceid>
      <deviceid>16A7</deviceid>
      <deviceid>16C7</deviceid>
      <deviceid>164D</deviceid>
      <deviceid>1648</deviceid>
      <deviceid>16A8</deviceid>
      <deviceid>1653</deviceid>
      <deviceid>166E</deviceid>
      <deviceid>1654</deviceid>
      <deviceid>165D</deviceid>
      <deviceid>165E</deviceid>
      <deviceid>166D</deviceid>
      <deviceid>170D</deviceid>
      <deviceid>170E</deviceid>
      <deviceid>1696</deviceid>
      <deviceid>1676</deviceid>
      <deviceid>1677</deviceid>
      <deviceid>1659</deviceid>
      <deviceid>167C</deviceid>
      <deviceid>167D</deviceid>
      <deviceid>169D</deviceid>
      <deviceid>16F7</deviceid>
      <deviceid>16FD</deviceid>
      <deviceid>16FE</deviceid>
      <deviceid>16DD</deviceid>
```

```
        <deviceid>1668</deviceid>
        <deviceid>1669</deviceid>
        <deviceid>1678</deviceid>
        <deviceid>1679</deviceid>
        <deviceid>1600</deviceid>
        <deviceid>1601</deviceid>
        <deviceid>166A</deviceid>
        <deviceid>166B</deviceid>
        <deviceid>16FF</deviceid>
        <deviceid>170F</deviceid>
        <deviceid>169B</deviceid>
        <deviceid>1693</deviceid>
        <deviceid>167F</deviceid>
        <deviceid>169A</deviceid>
        <deviceid>1698</deviceid>
        <deviceid>1692</deviceid>
        <deviceid>1694</deviceid>
        <deviceid>1690</deviceid>
        <deviceid>1691</deviceid>
        <deviceid>1699</deviceid>
        <deviceid>16A0</deviceid>
        <deviceid>167B</deviceid>
        <deviceid>1673</deviceid>
        <deviceid>165A</deviceid>
        <deviceid>1674</deviceid>
        <deviceid>1681</deviceid>
        <deviceid>1680</deviceid>
        <deviceid>1688</deviceid>
        <deviceid>167A</deviceid>
        <deviceid>1672</deviceid>
        <deviceid>1684</deviceid>
        <deviceid>165B</deviceid>
        <deviceid>16B1</deviceid>
        <deviceid>16B5</deviceid>
        <deviceid>16B0</deviceid>
        <deviceid>16B4</deviceid>
        <deviceid>16B2</deviceid>
        <deviceid>16B6</deviceid>
        <deviceid>1682</deviceid>
        <deviceid>1686</deviceid>
        <deviceid>16B3</deviceid>
        <deviceid>16B7</deviceid>
        <deviceid>1655</deviceid>
        <deviceid>1665</deviceid>
        <deviceid>1656</deviceid>
        <deviceid>1657</deviceid>
        <deviceid>165F</deviceid>
        <deviceid>165C</deviceid>
        <deviceid>1683</deviceid>
        <deviceid>1641</deviceid>
        <deviceid>1642</deviceid>
        <deviceid>1643</deviceid>
        <deviceid>1687</deviceid>
        <deviceid>164A</deviceid>
        <deviceid>16AA</deviceid>
```

```
      <deviceid>164C</deviceid>
      <deviceid>16AC</deviceid>
      <deviceid>1639</deviceid>
      <deviceid>163A</deviceid>
      <deviceid>163B</deviceid>
      <deviceid>163C</deviceid>
      <deviceid>164E</deviceid>
      <deviceid>164F</deviceid>
      <deviceid>1650</deviceid>
      <deviceid>1662</deviceid>
      <deviceid>1663</deviceid>
      <deviceid>168A</deviceid>
      <deviceid>168D</deviceid>
      <deviceid>16A1</deviceid>
      <deviceid>16A2</deviceid>
      <deviceid>168E</deviceid>
      <deviceid>163D</deviceid>
      <deviceid>16A5</deviceid>
      <deviceid>16A4</deviceid>
      <deviceid>16AE</deviceid>
      <deviceid>163E</deviceid>
  </NIC>

  <NIC>
    <manufacturer>1969</manufacturer>
      <deviceid>1062</deviceid>
      <deviceid>1063</deviceid>
      <deviceid>2060</deviceid>
      <deviceid>2062</deviceid>
      <deviceid>1073</deviceid>
      <deviceid>1083</deviceid>
      <deviceid>1090</deviceid>
      <deviceid>1091</deviceid>
      <deviceid>E091</deviceid>
      <deviceid>10A0</deviceid>
      <deviceid>10A1</deviceid>
      <deviceid>E0A1</deviceid>
      <deviceid>10B0</deviceid>
      <deviceid>10B1</deviceid>
      <deviceid>E0B1</deviceid>
      <deviceid>10C0</deviceid>
      <deviceid>10C1</deviceid>
      <deviceid>E0C1</deviceid>
      <deviceid>10D0</deviceid>
      <deviceid>10D1</deviceid>
      <deviceid>E0D1</deviceid>
      <deviceid>10E0</deviceid>
      <deviceid>10E1</deviceid>
      <deviceid>E0E1</deviceid>
      <deviceid>10F0</deviceid>
      <deviceid>10F1</deviceid>
      <deviceid>E0F1</deviceid>
  </NIC>

  <NIC>
```

```
    <manufacturer>19A2</manufacturer>
      <deviceid>0211</deviceid>
      <deviceid>0215</deviceid>
      <deviceid>0221</deviceid>
      <deviceid>0700</deviceid>
      <deviceid>0710</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10DF</manufacturer>
      <deviceid>0720</deviceid>
      <deviceid>E220</deviceid>
  </NIC>

  <NIC>
    <manufacturer>15B3</manufacturer>
      <deviceid>1000</deviceid>
      <deviceid>1001</deviceid>
      <deviceid>1002</deviceid>
      <deviceid>1003</deviceid>
      <deviceid>1004</deviceid>
      <deviceid>1005</deviceid>
      <deviceid>1006</deviceid>
      <deviceid>1007</deviceid>
      <deviceid>1008</deviceid>
      <deviceid>1009</deviceid>
      <deviceid>100A</deviceid>
      <deviceid>100B</deviceid>
      <deviceid>100C</deviceid>
      <deviceid>100D</deviceid>
      <deviceid>100E</deviceid>
      <deviceid>100F</deviceid>
      <deviceid>1010</deviceid>
      <deviceid>6340</deviceid>
      <deviceid>6341</deviceid>
      <deviceid>634A</deviceid>
      <deviceid>634B</deviceid>
      <deviceid>6354</deviceid>
      <deviceid>6368</deviceid>
      <deviceid>6369</deviceid>
      <deviceid>6372</deviceid>
      <deviceid>6732</deviceid>
      <deviceid>6733</deviceid>
      <deviceid>673C</deviceid>
      <deviceid>673D</deviceid>
      <deviceid>6746</deviceid>
      <deviceid>6750</deviceid>
      <deviceid>6751</deviceid>
      <deviceid>675A</deviceid>
      <deviceid>6764</deviceid>
      <deviceid>6765</deviceid>
      <deviceid>676E</deviceid>
      <deviceid>6778</deviceid>
      <deviceid>1013</deviceid>
      <deviceid>1015</deviceid>
```

```
      <deviceid>1017</deviceid>
      <deviceid>1019</deviceid>
      <deviceid>101B</deviceid>
      <deviceid>101D</deviceid>
      <deviceid>101F</deviceid>
      <deviceid>1021</deviceid>
      <deviceid>1023</deviceid>
      <deviceid>1025</deviceid>
      <deviceid>1027</deviceid>
      <deviceid>1029</deviceid>
      <deviceid>102B</deviceid>
      <deviceid>102F</deviceid>
  </NIC>

  <NIC>
    <manufacturer>1137</manufacturer>
      <deviceid>0043</deviceid>
  </NIC>

</SupportedNetworkInterfaceCards>
```

另请参阅

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

Windows 8 中的网络内核调试支持的以太网 NIC

Windows 8.1 中的网络内核调试支持的以太网 NIC

Windows 10版本 1709 (Redstone 3) 内部版本 16299 中支持用于网络内核调试的以太网 NIC

项目 • 2024/02/07

若要通过以太网网线进行内核调试，目标计算机必须具有受支持的网络接口卡 (NIC) 。

在内核调试期间，运行调试器的计算机称为 *主计算机*，正在调试的计算机称为 *目标计算机*。 有关详细信息，请参阅 设置 KDNET 网络内核自动调试。

有关支持的网络适配器的一般信息，请参阅支持在 Windows 10 中进行网络内核调试的以太网 NIC。

版本信息

支持的适配器列表适用于以下版本的 Windows

- Windows 10版本 1709 (Redstone 3) 内部版本 16299

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<SupportedNetworkInterfaceCards>
  <NIC>
    <manufacturer>8086</manufacturer>
      <deviceid>1000</deviceid>
      <deviceid>1001</deviceid>
      <deviceid>1004</deviceid>
      <deviceid>1008</deviceid>
      <deviceid>1009</deviceid>
      <deviceid>100C</deviceid>
      <deviceid>100D</deviceid>
      <deviceid>100E</deviceid>
      <deviceid>1015</deviceid>
      <deviceid>1016</deviceid>
      <deviceid>1017</deviceid>
      <deviceid>101E</deviceid>
      <deviceid>100F</deviceid>
      <deviceid>1011</deviceid>
      <deviceid>1026</deviceid>
      <deviceid>1027</deviceid>
      <deviceid>1028</deviceid>
      <deviceid>1010</deviceid>
      <deviceid>1012</deviceid>
      <deviceid>101D</deviceid>
      <deviceid>1079</deviceid>
      <deviceid>107A</deviceid>
```

```
        <deviceid>107B</deviceid>
        <deviceid>108A</deviceid>
        <deviceid>1099</deviceid>
        <deviceid>10B5</deviceid>
        <deviceid>1013</deviceid>
        <deviceid>1018</deviceid>
        <deviceid>1014</deviceid>
        <deviceid>1078</deviceid>
        <deviceid>1076</deviceid>
        <deviceid>107C</deviceid>
        <deviceid>1077</deviceid>
        <deviceid>1019</deviceid>
        <deviceid>101A</deviceid>
        <deviceid>1075</deviceid>
        <deviceid>105E</deviceid>
        <deviceid>105F</deviceid>
        <deviceid>1060</deviceid>
        <deviceid>10D9</deviceid>
        <deviceid>10DA</deviceid>
        <deviceid>10A4</deviceid>
        <deviceid>10D5</deviceid>
        <deviceid>10A5</deviceid>
        <deviceid>10BC</deviceid>
        <deviceid>107D</deviceid>
        <deviceid>107E</deviceid>
        <deviceid>107F</deviceid>
        <deviceid>10B9</deviceid>
        <deviceid>108B</deviceid>
        <deviceid>108C</deviceid>
        <deviceid>109A</deviceid>
        <deviceid>10D3</deviceid>
        <deviceid>10F6</deviceid>
        <deviceid>150C</deviceid>
        <deviceid>1096</deviceid>
        <deviceid>1098</deviceid>
        <deviceid>10BA</deviceid>
        <deviceid>10BB</deviceid>
        <deviceid>1501</deviceid>
        <deviceid>1049</deviceid>
        <deviceid>104A</deviceid>
        <deviceid>104B</deviceid>
        <deviceid>104C</deviceid>
        <deviceid>10C4</deviceid>
        <deviceid>10C5</deviceid>
        <deviceid>104D</deviceid>
        <deviceid>10BF</deviceid>
        <deviceid>10F5</deviceid>
        <deviceid>10CB</deviceid>
        <deviceid>10BD</deviceid>
        <deviceid>10E5</deviceid>
        <deviceid>294C</deviceid>
        <deviceid>10C0</deviceid>
        <deviceid>10C3</deviceid>
        <deviceid>10C2</deviceid>
        <deviceid>10CC</deviceid>
```

```
        <deviceid>10CD</deviceid>
        <deviceid>10CE</deviceid>
        <deviceid>10DE</deviceid>
        <deviceid>10DF</deviceid>
        <deviceid>1525</deviceid>
        <deviceid>10EA</deviceid>
        <deviceid>10EB</deviceid>
        <deviceid>10EF</deviceid>
        <deviceid>10F0</deviceid>
        <deviceid>1502</deviceid>
        <deviceid>1503</deviceid>
        <deviceid>153A</deviceid>
        <deviceid>153B</deviceid>
        <deviceid>155A</deviceid>
        <deviceid>1559</deviceid>
        <deviceid>15A0</deviceid>
        <deviceid>15A1</deviceid>
        <deviceid>15A2</deviceid>
        <deviceid>15A3</deviceid>
        <deviceid>156F</deviceid>
        <deviceid>1570</deviceid>
        <deviceid>15B7</deviceid>
        <deviceid>15B8</deviceid>
        <deviceid>15B9</deviceid>
        <deviceid>15D7</deviceid>
        <deviceid>15D8</deviceid>
        <deviceid>15E3</deviceid>
        <deviceid>15D6</deviceid>
        <deviceid>15BD</deviceid>
        <deviceid>15BE</deviceid>
        <deviceid>15BB</deviceid>
        <deviceid>15BC</deviceid>
        <deviceid>15DF</deviceid>
        <deviceid>15E0</deviceid>
        <deviceid>15E1</deviceid>
        <deviceid>15E2</deviceid>
        <deviceid>10C9</deviceid>
        <deviceid>10E6</deviceid>
        <deviceid>10E7</deviceid>
        <deviceid>10E8</deviceid>
        <deviceid>1526</deviceid>
        <deviceid>150A</deviceid>
        <deviceid>1518</deviceid>
        <deviceid>150D</deviceid>
        <deviceid>10A7</deviceid>
        <deviceid>10A9</deviceid>
        <deviceid>10D6</deviceid>
        <deviceid>150E</deviceid>
        <deviceid>150F</deviceid>
        <deviceid>1510</deviceid>
        <deviceid>1511</deviceid>
        <deviceid>1516</deviceid>
        <deviceid>1527</deviceid>
        <deviceid>1521</deviceid>
        <deviceid>1522</deviceid>
```

```
        <deviceid>1523</deviceid>
        <deviceid>1524</deviceid>
        <deviceid>1546</deviceid>
        <deviceid>1533</deviceid>
        <deviceid>1534</deviceid>
        <deviceid>1535</deviceid>
        <deviceid>1536</deviceid>
        <deviceid>1537</deviceid>
        <deviceid>1538</deviceid>
        <deviceid>157B</deviceid>
        <deviceid>157C</deviceid>
        <deviceid>1539</deviceid>
        <deviceid>1F40</deviceid>
        <deviceid>1F41</deviceid>
        <deviceid>1F45</deviceid>
        <deviceid>0438</deviceid>
        <deviceid>043A</deviceid>
        <deviceid>043C</deviceid>
        <deviceid>0440</deviceid>
        <deviceid>10C6</deviceid>
        <deviceid>10C7</deviceid>
        <deviceid>10C8</deviceid>
        <deviceid>150B</deviceid>
        <deviceid>10DB</deviceid>
        <deviceid>10DD</deviceid>
        <deviceid>10EC</deviceid>
        <deviceid>10F1</deviceid>
        <deviceid>10E1</deviceid>
        <deviceid>10F4</deviceid>
        <deviceid>10F7</deviceid>
        <deviceid>1514</deviceid>
        <deviceid>1517</deviceid>
        <deviceid>10F8</deviceid>
        <deviceid>000C</deviceid>
        <deviceid>1F63</deviceid>
        <deviceid>10F9</deviceid>
        <deviceid>10FB</deviceid>
        <deviceid>11A9</deviceid>
        <deviceid>1071</deviceid>
        <deviceid>1F72</deviceid>
        <deviceid>17D0</deviceid>
        <deviceid>0470</deviceid>
        <deviceid>211B</deviceid>
        <deviceid>8976</deviceid>
        <deviceid>2159</deviceid>
        <deviceid>000D</deviceid>
        <deviceid>0008</deviceid>
        <deviceid>152A</deviceid>
        <deviceid>1529</deviceid>
        <deviceid>1507</deviceid>
        <deviceid>154D</deviceid>
        <deviceid>154A</deviceid>
        <deviceid>1558</deviceid>
        <deviceid>1557</deviceid>
        <deviceid>0001</deviceid>
```

```
      <deviceid>10FC</deviceid>
      <deviceid>151C</deviceid>
      <deviceid>1528</deviceid>
      <deviceid>1560</deviceid>
      <deviceid>1563</deviceid>
      <deviceid>ABCD</deviceid>
      <deviceid>15AA</deviceid>
      <deviceid>15AB</deviceid>
      <deviceid>15AC</deviceid>
      <deviceid>15AD</deviceid>
      <deviceid>15AE</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10EC</manufacturer>
      <deviceid>8136</deviceid>
      <deviceid>8137</deviceid>
      <deviceid>8168</deviceid>
      <deviceid>8167</deviceid>
      <deviceid>8169</deviceid>
      <deviceid>8166</deviceid>
  </NIC>

  <NIC>
    <manufacturer>14E4</manufacturer>
      <deviceid>1644</deviceid>
      <deviceid>1645</deviceid>
      <deviceid>1646</deviceid>
      <deviceid>16A6</deviceid>
      <deviceid>16C6</deviceid>
      <deviceid>1647</deviceid>
      <deviceid>16A7</deviceid>
      <deviceid>16C7</deviceid>
      <deviceid>164D</deviceid>
      <deviceid>1648</deviceid>
      <deviceid>16A8</deviceid>
      <deviceid>1653</deviceid>
      <deviceid>166E</deviceid>
      <deviceid>1654</deviceid>
      <deviceid>165D</deviceid>
      <deviceid>165E</deviceid>
      <deviceid>166D</deviceid>
      <deviceid>170D</deviceid>
      <deviceid>170E</deviceid>
      <deviceid>1696</deviceid>
      <deviceid>1676</deviceid>
      <deviceid>1677</deviceid>
      <deviceid>1659</deviceid>
      <deviceid>167C</deviceid>
      <deviceid>167D</deviceid>
      <deviceid>169D</deviceid>
      <deviceid>16F7</deviceid>
      <deviceid>16FD</deviceid>
      <deviceid>16FE</deviceid>
      <deviceid>16DD</deviceid>
```

```
        <deviceid>1668</deviceid>
        <deviceid>1669</deviceid>
        <deviceid>1678</deviceid>
        <deviceid>1679</deviceid>
        <deviceid>1600</deviceid>
        <deviceid>1601</deviceid>
        <deviceid>166A</deviceid>
        <deviceid>166B</deviceid>
        <deviceid>16FF</deviceid>
        <deviceid>170F</deviceid>
        <deviceid>169B</deviceid>
        <deviceid>1693</deviceid>
        <deviceid>167F</deviceid>
        <deviceid>169A</deviceid>
        <deviceid>1698</deviceid>
        <deviceid>1692</deviceid>
        <deviceid>1694</deviceid>
        <deviceid>1690</deviceid>
        <deviceid>1691</deviceid>
        <deviceid>1699</deviceid>
        <deviceid>16A0</deviceid>
        <deviceid>167B</deviceid>
        <deviceid>1673</deviceid>
        <deviceid>165A</deviceid>
        <deviceid>1674</deviceid>
        <deviceid>1681</deviceid>
        <deviceid>1680</deviceid>
        <deviceid>1688</deviceid>
        <deviceid>167A</deviceid>
        <deviceid>1672</deviceid>
        <deviceid>1684</deviceid>
        <deviceid>165B</deviceid>
        <deviceid>16B1</deviceid>
        <deviceid>16B5</deviceid>
        <deviceid>16B0</deviceid>
        <deviceid>16B4</deviceid>
        <deviceid>16B2</deviceid>
        <deviceid>16B6</deviceid>
        <deviceid>1682</deviceid>
        <deviceid>1686</deviceid>
        <deviceid>16B3</deviceid>
        <deviceid>16B7</deviceid>
        <deviceid>1655</deviceid>
        <deviceid>1665</deviceid>
        <deviceid>1656</deviceid>
        <deviceid>1657</deviceid>
        <deviceid>165F</deviceid>
        <deviceid>165C</deviceid>
        <deviceid>1683</deviceid>
        <deviceid>1641</deviceid>
        <deviceid>1642</deviceid>
        <deviceid>1643</deviceid>
        <deviceid>1687</deviceid>
        <deviceid>164A</deviceid>
        <deviceid>16AA</deviceid>
```

```
      <deviceid>164C</deviceid>
      <deviceid>16AC</deviceid>
      <deviceid>1639</deviceid>
      <deviceid>163A</deviceid>
      <deviceid>163B</deviceid>
      <deviceid>163C</deviceid>
      <deviceid>164E</deviceid>
      <deviceid>164F</deviceid>
      <deviceid>1650</deviceid>
      <deviceid>1662</deviceid>
      <deviceid>1663</deviceid>
      <deviceid>168A</deviceid>
      <deviceid>168D</deviceid>
      <deviceid>16A1</deviceid>
      <deviceid>16A2</deviceid>
      <deviceid>168E</deviceid>
      <deviceid>163D</deviceid>
      <deviceid>16A5</deviceid>
      <deviceid>16A4</deviceid>
      <deviceid>16AE</deviceid>
      <deviceid>163E</deviceid>
  </NIC>

  <NIC>
    <manufacturer>1969</manufacturer>
      <deviceid>1062</deviceid>
      <deviceid>1063</deviceid>
      <deviceid>2060</deviceid>
      <deviceid>2062</deviceid>
      <deviceid>1073</deviceid>
      <deviceid>1083</deviceid>
      <deviceid>1090</deviceid>
      <deviceid>1091</deviceid>
      <deviceid>E091</deviceid>
      <deviceid>10A0</deviceid>
      <deviceid>10A1</deviceid>
      <deviceid>E0A1</deviceid>
      <deviceid>10B0</deviceid>
      <deviceid>10B1</deviceid>
      <deviceid>E0B1</deviceid>
      <deviceid>10C0</deviceid>
      <deviceid>10C1</deviceid>
      <deviceid>E0C1</deviceid>
      <deviceid>10D0</deviceid>
      <deviceid>10D1</deviceid>
      <deviceid>E0D1</deviceid>
      <deviceid>10E0</deviceid>
      <deviceid>10E1</deviceid>
      <deviceid>E0E1</deviceid>
      <deviceid>10F0</deviceid>
      <deviceid>10F1</deviceid>
      <deviceid>E0F1</deviceid>
  </NIC>

  <NIC>
```

```
    <manufacturer>19A2</manufacturer>
      <deviceid>0211</deviceid>
      <deviceid>0215</deviceid>
      <deviceid>0221</deviceid>
      <deviceid>0700</deviceid>
      <deviceid>0710</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10DF</manufacturer>
      <deviceid>0720</deviceid>
      <deviceid>E220</deviceid>
  </NIC>

  <NIC>
    <manufacturer>15B3</manufacturer>
      <deviceid>1000</deviceid>
      <deviceid>1001</deviceid>
      <deviceid>1002</deviceid>
      <deviceid>1003</deviceid>
      <deviceid>1004</deviceid>
      <deviceid>1005</deviceid>
      <deviceid>1006</deviceid>
      <deviceid>1007</deviceid>
      <deviceid>1008</deviceid>
      <deviceid>1009</deviceid>
      <deviceid>100A</deviceid>
      <deviceid>100B</deviceid>
      <deviceid>100C</deviceid>
      <deviceid>100D</deviceid>
      <deviceid>100E</deviceid>
      <deviceid>100F</deviceid>
      <deviceid>1010</deviceid>
      <deviceid>6340</deviceid>
      <deviceid>6341</deviceid>
      <deviceid>634A</deviceid>
      <deviceid>634B</deviceid>
      <deviceid>6354</deviceid>
      <deviceid>6368</deviceid>
      <deviceid>6369</deviceid>
      <deviceid>6372</deviceid>
      <deviceid>6732</deviceid>
      <deviceid>6733</deviceid>
      <deviceid>673C</deviceid>
      <deviceid>673D</deviceid>
      <deviceid>6746</deviceid>
      <deviceid>6750</deviceid>
      <deviceid>6751</deviceid>
      <deviceid>675A</deviceid>
      <deviceid>6764</deviceid>
      <deviceid>6765</deviceid>
      <deviceid>676E</deviceid>
      <deviceid>6778</deviceid>
      <deviceid>1013</deviceid>
      <deviceid>1015</deviceid>
```

```
      <deviceid>1017</deviceid>
      <deviceid>1019</deviceid>
      <deviceid>101B</deviceid>
      <deviceid>101D</deviceid>
      <deviceid>101F</deviceid>
      <deviceid>1021</deviceid>
      <deviceid>1023</deviceid>
      <deviceid>1025</deviceid>
      <deviceid>1027</deviceid>
      <deviceid>1029</deviceid>
      <deviceid>102B</deviceid>
      <deviceid>102F</deviceid>
  </NIC>

  <NIC>
    <manufacturer>1137</manufacturer>
      <deviceid>0043</deviceid>
  </NIC>

</SupportedNetworkInterfaceCards>
```

另请参阅

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

Windows 8 中的网络内核调试支持的以太网 NIC

Windows 8.1 中的网络内核调试支持的以太网 NIC

Windows 10版本 1703 (Redstone 2) 内部版本 15063 中支持用于网络内核调试的以太网 NIC

项目 • 2023/04/14

若要通过以太网网线进行内核调试，目标计算机必须具有受支持的网络接口卡 (NIC) 。

在内核调试期间，运行调试器的计算机称为 *主计算机*，正在调试的计算机称为 *目标计算机*。 有关详细信息，请参阅 自动设置 KDNET 网络内核调试。

有关支持的网络适配器的一般信息，请参阅 Windows 10 中支持的网络内核调试的以太网 NIC。

版本信息

支持的适配器列表适用于以下版本的 Windows。

- Windows 10版本 1703 Redstone 2 内部版本 15063

XML

```
<?xml version="1.0" encoding="utf-8"?>
<SupportedNetworkInterfaceCards>
  <NIC>
    <manufacturer>8086</manufacturer>
      <deviceid>1000</deviceid>
      <deviceid>1001</deviceid>
      <deviceid>1004</deviceid>
      <deviceid>1008</deviceid>
      <deviceid>1009</deviceid>
      <deviceid>100C</deviceid>
      <deviceid>100D</deviceid>
      <deviceid>100E</deviceid>
      <deviceid>1015</deviceid>
      <deviceid>1016</deviceid>
      <deviceid>1017</deviceid>
      <deviceid>101E</deviceid>
      <deviceid>100F</deviceid>
      <deviceid>1011</deviceid>
      <deviceid>1026</deviceid>
      <deviceid>1027</deviceid>
      <deviceid>1028</deviceid>
      <deviceid>1010</deviceid>
      <deviceid>1012</deviceid>
      <deviceid>101D</deviceid>
      <deviceid>1079</deviceid>
      <deviceid>107A</deviceid>
```

```
        <deviceid>107B</deviceid>
        <deviceid>108A</deviceid>
        <deviceid>1099</deviceid>
        <deviceid>10B5</deviceid>
        <deviceid>1013</deviceid>
        <deviceid>1018</deviceid>
        <deviceid>1014</deviceid>
        <deviceid>1078</deviceid>
        <deviceid>1076</deviceid>
        <deviceid>107C</deviceid>
        <deviceid>1077</deviceid>
        <deviceid>1019</deviceid>
        <deviceid>101A</deviceid>
        <deviceid>1075</deviceid>
        <deviceid>105E</deviceid>
        <deviceid>105F</deviceid>
        <deviceid>1060</deviceid>
        <deviceid>10D9</deviceid>
        <deviceid>10DA</deviceid>
        <deviceid>10A4</deviceid>
        <deviceid>10D5</deviceid>
        <deviceid>10A5</deviceid>
        <deviceid>10BC</deviceid>
        <deviceid>107D</deviceid>
        <deviceid>107E</deviceid>
        <deviceid>107F</deviceid>
        <deviceid>10B9</deviceid>
        <deviceid>108B</deviceid>
        <deviceid>108C</deviceid>
        <deviceid>109A</deviceid>
        <deviceid>10D3</deviceid>
        <deviceid>10F6</deviceid>
        <deviceid>150C</deviceid>
        <deviceid>1096</deviceid>
        <deviceid>1098</deviceid>
        <deviceid>10BA</deviceid>
        <deviceid>10BB</deviceid>
        <deviceid>1501</deviceid>
        <deviceid>1049</deviceid>
        <deviceid>104A</deviceid>
        <deviceid>104B</deviceid>
        <deviceid>104C</deviceid>
        <deviceid>10C4</deviceid>
        <deviceid>10C5</deviceid>
        <deviceid>104D</deviceid>
        <deviceid>10BF</deviceid>
        <deviceid>10F5</deviceid>
        <deviceid>10CB</deviceid>
        <deviceid>10BD</deviceid>
        <deviceid>10E5</deviceid>
        <deviceid>294C</deviceid>
        <deviceid>10C0</deviceid>
        <deviceid>10C3</deviceid>
        <deviceid>10C2</deviceid>
        <deviceid>10CC</deviceid>
```

```
        <deviceid>10CD</deviceid>
        <deviceid>10CE</deviceid>
        <deviceid>10DE</deviceid>
        <deviceid>10DF</deviceid>
        <deviceid>1525</deviceid>
        <deviceid>10EA</deviceid>
        <deviceid>10EB</deviceid>
        <deviceid>10EF</deviceid>
        <deviceid>10F0</deviceid>
        <deviceid>1502</deviceid>
        <deviceid>1503</deviceid>
        <deviceid>153A</deviceid>
        <deviceid>153B</deviceid>
        <deviceid>155A</deviceid>
        <deviceid>1559</deviceid>
        <deviceid>15A0</deviceid>
        <deviceid>15A1</deviceid>
        <deviceid>15A2</deviceid>
        <deviceid>15A3</deviceid>
        <deviceid>156F</deviceid>
        <deviceid>1570</deviceid>
        <deviceid>15B7</deviceid>
        <deviceid>15B8</deviceid>
        <deviceid>10C9</deviceid>
        <deviceid>10E6</deviceid>
        <deviceid>10E7</deviceid>
        <deviceid>10E8</deviceid>
        <deviceid>1526</deviceid>
        <deviceid>150A</deviceid>
        <deviceid>1518</deviceid>
        <deviceid>150D</deviceid>
        <deviceid>10A7</deviceid>
        <deviceid>10A9</deviceid>
        <deviceid>10D6</deviceid>
        <deviceid>150E</deviceid>
        <deviceid>150F</deviceid>
        <deviceid>1510</deviceid>
        <deviceid>1511</deviceid>
        <deviceid>1516</deviceid>
        <deviceid>1527</deviceid>
        <deviceid>1521</deviceid>
        <deviceid>1522</deviceid>
        <deviceid>1523</deviceid>
        <deviceid>1524</deviceid>
        <deviceid>1546</deviceid>
        <deviceid>1533</deviceid>
        <deviceid>1534</deviceid>
        <deviceid>1535</deviceid>
        <deviceid>1536</deviceid>
        <deviceid>1537</deviceid>
        <deviceid>1538</deviceid>
        <deviceid>157B</deviceid>
        <deviceid>157C</deviceid>
        <deviceid>1539</deviceid>
        <deviceid>1F40</deviceid>
```

```
        <deviceid>1F41</deviceid>
        <deviceid>1F45</deviceid>
        <deviceid>0438</deviceid>
        <deviceid>043A</deviceid>
        <deviceid>043C</deviceid>
        <deviceid>0440</deviceid>
        <deviceid>10C6</deviceid>
        <deviceid>10C7</deviceid>
        <deviceid>10C8</deviceid>
        <deviceid>150B</deviceid>
        <deviceid>10DB</deviceid>
        <deviceid>10DD</deviceid>
        <deviceid>10EC</deviceid>
        <deviceid>10F1</deviceid>
        <deviceid>10E1</deviceid>
        <deviceid>10F4</deviceid>
        <deviceid>10F7</deviceid>
        <deviceid>1514</deviceid>
        <deviceid>1517</deviceid>
        <deviceid>10F8</deviceid>
        <deviceid>000C</deviceid>
        <deviceid>1F63</deviceid>
        <deviceid>10F9</deviceid>
        <deviceid>10FB</deviceid>
        <deviceid>11A9</deviceid>
        <deviceid>1071</deviceid>
        <deviceid>1F72</deviceid>
        <deviceid>17D0</deviceid>
        <deviceid>0470</deviceid>
        <deviceid>211B</deviceid>
        <deviceid>8976</deviceid>
        <deviceid>2159</deviceid>
        <deviceid>000D</deviceid>
        <deviceid>0008</deviceid>
        <deviceid>152A</deviceid>
        <deviceid>1529</deviceid>
        <deviceid>1507</deviceid>
        <deviceid>154D</deviceid>
        <deviceid>154A</deviceid>
        <deviceid>1558</deviceid>
        <deviceid>1557</deviceid>
        <deviceid>0001</deviceid>
        <deviceid>10FC</deviceid>
        <deviceid>151C</deviceid>
        <deviceid>1528</deviceid>
        <deviceid>1560</deviceid>
        <deviceid>1563</deviceid>
        <deviceid>ABCD</deviceid>
        <deviceid>15AA</deviceid>
        <deviceid>15AB</deviceid>
        <deviceid>15AC</deviceid>
        <deviceid>15AD</deviceid>
        <deviceid>15AE</deviceid>
    </NIC>
```

```
  <NIC>
    <manufacturer>10EC</manufacturer>
      <deviceid>8136</deviceid>
      <deviceid>8137</deviceid>
      <deviceid>8168</deviceid>
      <deviceid>8167</deviceid>
      <deviceid>8169</deviceid>
      <deviceid>8166</deviceid>
  </NIC>

  <NIC>
    <manufacturer>14E4</manufacturer>
      <deviceid>1644</deviceid>
      <deviceid>1645</deviceid>
      <deviceid>1646</deviceid>
      <deviceid>16A6</deviceid>
      <deviceid>16C6</deviceid>
      <deviceid>1647</deviceid>
      <deviceid>16A7</deviceid>
      <deviceid>16C7</deviceid>
      <deviceid>164D</deviceid>
      <deviceid>1648</deviceid>
      <deviceid>16A8</deviceid>
      <deviceid>1653</deviceid>
      <deviceid>166E</deviceid>
      <deviceid>1654</deviceid>
      <deviceid>165D</deviceid>
      <deviceid>165E</deviceid>
      <deviceid>166D</deviceid>
      <deviceid>170D</deviceid>
      <deviceid>170E</deviceid>
      <deviceid>1696</deviceid>
      <deviceid>1676</deviceid>
      <deviceid>1677</deviceid>
      <deviceid>1659</deviceid>
      <deviceid>167C</deviceid>
      <deviceid>167D</deviceid>
      <deviceid>169D</deviceid>
      <deviceid>16F7</deviceid>
      <deviceid>16FD</deviceid>
      <deviceid>16FE</deviceid>
      <deviceid>16DD</deviceid>
      <deviceid>1668</deviceid>
      <deviceid>1669</deviceid>
      <deviceid>1678</deviceid>
      <deviceid>1679</deviceid>
      <deviceid>1600</deviceid>
      <deviceid>1601</deviceid>
      <deviceid>166A</deviceid>
      <deviceid>166B</deviceid>
      <deviceid>16FF</deviceid>
      <deviceid>170F</deviceid>
      <deviceid>169B</deviceid>
      <deviceid>1693</deviceid>
      <deviceid>167F</deviceid>
```

```
        <deviceid>169A</deviceid>
        <deviceid>1698</deviceid>
        <deviceid>1692</deviceid>
        <deviceid>1694</deviceid>
        <deviceid>1690</deviceid>
        <deviceid>1691</deviceid>
        <deviceid>1699</deviceid>
        <deviceid>16A0</deviceid>
        <deviceid>167B</deviceid>
        <deviceid>1673</deviceid>
        <deviceid>165A</deviceid>
        <deviceid>1674</deviceid>
        <deviceid>1681</deviceid>
        <deviceid>1680</deviceid>
        <deviceid>1688</deviceid>
        <deviceid>167A</deviceid>
        <deviceid>1672</deviceid>
        <deviceid>1684</deviceid>
        <deviceid>165B</deviceid>
        <deviceid>16B1</deviceid>
        <deviceid>16B5</deviceid>
        <deviceid>16B0</deviceid>
        <deviceid>16B4</deviceid>
        <deviceid>16B2</deviceid>
        <deviceid>16B6</deviceid>
        <deviceid>1682</deviceid>
        <deviceid>1686</deviceid>
        <deviceid>16B3</deviceid>
        <deviceid>16B7</deviceid>
        <deviceid>1655</deviceid>
        <deviceid>1665</deviceid>
        <deviceid>1656</deviceid>
        <deviceid>1657</deviceid>
        <deviceid>165F</deviceid>
        <deviceid>165C</deviceid>
        <deviceid>1683</deviceid>
        <deviceid>1641</deviceid>
        <deviceid>1642</deviceid>
        <deviceid>1643</deviceid>
        <deviceid>1687</deviceid>
        <deviceid>164A</deviceid>
        <deviceid>16AA</deviceid>
        <deviceid>164C</deviceid>
        <deviceid>16AC</deviceid>
        <deviceid>1639</deviceid>
        <deviceid>163A</deviceid>
        <deviceid>163B</deviceid>
        <deviceid>163C</deviceid>
        <deviceid>164E</deviceid>
        <deviceid>164F</deviceid>
        <deviceid>1650</deviceid>
        <deviceid>1662</deviceid>
        <deviceid>1663</deviceid>
        <deviceid>168A</deviceid>
        <deviceid>168D</deviceid>
```

```
      <deviceid>16A1</deviceid>
      <deviceid>16A2</deviceid>
      <deviceid>168E</deviceid>
      <deviceid>163D</deviceid>
      <deviceid>16A5</deviceid>
      <deviceid>16A4</deviceid>
      <deviceid>16AE</deviceid>
      <deviceid>163E</deviceid>
  </NIC>

  <NIC>
    <manufacturer>1969</manufacturer>
      <deviceid>1062</deviceid>
      <deviceid>1063</deviceid>
      <deviceid>2060</deviceid>
      <deviceid>2062</deviceid>
      <deviceid>1073</deviceid>
      <deviceid>1083</deviceid>
      <deviceid>1090</deviceid>
      <deviceid>1091</deviceid>
      <deviceid>E091</deviceid>
      <deviceid>10A0</deviceid>
      <deviceid>10A1</deviceid>
      <deviceid>E0A1</deviceid>
      <deviceid>10B0</deviceid>
      <deviceid>10B1</deviceid>
      <deviceid>E0B1</deviceid>
      <deviceid>10C0</deviceid>
      <deviceid>10C1</deviceid>
      <deviceid>E0C1</deviceid>
      <deviceid>10D0</deviceid>
      <deviceid>10D1</deviceid>
      <deviceid>E0D1</deviceid>
      <deviceid>10E0</deviceid>
      <deviceid>10E1</deviceid>
      <deviceid>E0E1</deviceid>
      <deviceid>10F0</deviceid>
      <deviceid>10F1</deviceid>
      <deviceid>E0F1</deviceid>
  </NIC>

  <NIC>
    <manufacturer>19A2</manufacturer>
      <deviceid>0211</deviceid>
      <deviceid>0215</deviceid>
      <deviceid>0221</deviceid>
      <deviceid>0700</deviceid>
      <deviceid>0710</deviceid>
  </NIC>

  <NIC>
    <manufacturer>10DF</manufacturer>
      <deviceid>0720</deviceid>
      <deviceid>E220</deviceid>
  </NIC>
```

```
<NIC>
  <manufacturer>15B3</manufacturer>
    <deviceid>1000</deviceid>
    <deviceid>1001</deviceid>
    <deviceid>1002</deviceid>
    <deviceid>1003</deviceid>
    <deviceid>1004</deviceid>
    <deviceid>1005</deviceid>
    <deviceid>1006</deviceid>
    <deviceid>1007</deviceid>
    <deviceid>1008</deviceid>
    <deviceid>1009</deviceid>
    <deviceid>100A</deviceid>
    <deviceid>100B</deviceid>
    <deviceid>100C</deviceid>
    <deviceid>100D</deviceid>
    <deviceid>100E</deviceid>
    <deviceid>100F</deviceid>
    <deviceid>1010</deviceid>
    <deviceid>6340</deviceid>
    <deviceid>6341</deviceid>
    <deviceid>634A</deviceid>
    <deviceid>634B</deviceid>
    <deviceid>6354</deviceid>
    <deviceid>6368</deviceid>
    <deviceid>6369</deviceid>
    <deviceid>6372</deviceid>
    <deviceid>6732</deviceid>
    <deviceid>6733</deviceid>
    <deviceid>673C</deviceid>
    <deviceid>673D</deviceid>
    <deviceid>6746</deviceid>
    <deviceid>6750</deviceid>
    <deviceid>6751</deviceid>
    <deviceid>675A</deviceid>
    <deviceid>6764</deviceid>
    <deviceid>6765</deviceid>
    <deviceid>676E</deviceid>
    <deviceid>6778</deviceid>
    <deviceid>1013</deviceid>
    <deviceid>1015</deviceid>
    <deviceid>1017</deviceid>
    <deviceid>1019</deviceid>
    <deviceid>101B</deviceid>
    <deviceid>101D</deviceid>
    <deviceid>101F</deviceid>
    <deviceid>1021</deviceid>
    <deviceid>1023</deviceid>
    <deviceid>1025</deviceid>
    <deviceid>1027</deviceid>
    <deviceid>1029</deviceid>
    <deviceid>102B</deviceid>
    <deviceid>102F</deviceid>
</NIC>
```

```
  <NIC>
    <manufacturer>1137</manufacturer>
      <deviceid>0043</deviceid>
  </NIC>

</SupportedNetworkInterfaceCards>
```

另请参阅

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

Windows 8 中的网络内核调试支持的以太网 NIC

Windows 8.1 中的网络内核调试支持的以太网 NIC

Windows 8.1 中的网络内核调试支持的以太网 NIC

项目 • 2023/04/14

当目标计算机运行Windows 8.1时，可以通过以太网网线进行内核调试。 目标计算机必须具有受支持的网络接口卡 (NIC) 或网络适配器。

在内核调试期间，运行调试器的计算机称为 *主计算机*，正在调试的计算机称为 *目标计算机*。 有关详细信息，请参阅 设置 KDNET 网络内核自动调试。

若要通过网络电缆进行内核调试，目标计算机必须具有受支持的网络适配器。 当目标计算机运行Windows 8.1时，支持此处列出的网络适配器进行内核调试。

注意支持通过所选的 10 千兆位网络适配器进行内核调试是 Windows 8.1 中的一项新功能。 Windows 8不支持调试超过 10 千兆位的网络适配器。 有关Windows 8支持的用于内核调试的网络适配器的列表，请参阅 Windows 8 中网络内核调试支持的以太网 NIC。

系统要求

通过以太网 NIC 进行内核调试需要某些低级别的平台支持。 对于此调试解决方案，Windows 要求通过 PCI/PCIe 附加这些 NIC。 在大多数情况下，只需插入其中一个受支持的 NIC 即可获得可靠的内核调试体验。 但是，在某些情况下，BIOS 配置详细信息可能会阻碍 Windows 调试路径。 应考虑以下平台要求：

- 系统固件应发现并配置 NIC 设备，使其资源不会与已配置 BIOS 的任何其他设备冲突。

查找供应商 ID 和设备 ID

首先在目标计算机上查找网络适配器的供应商 ID 和设备 ID。

- 在目标计算机上，打开设备管理器 (命令提示符窗口中输入 **devmgmt**) 。
- 在“设备管理器”中，找到要用于调试的网络适配器。
- 选择并按住 (或右键单击) 网络适配器节点，然后选择“**属性**”。
- 在“ **详细信息** ”选项卡的 “**属性**”下，选择“ **硬件 ID**”。

供应商和设备 ID 显示为 VEN_*VendorID* 和 DEV_*DeviceID*。 例如，如果看到 PCI\VEN_8086&DEV_104B，则供应商 ID 为 8086，设备 ID 为 104B。

供应商 ID 8086，Intel Corporation

对于供应商 ID 8086，支持以下设备 ID：

0438 043A 043C 0440 1000 1001 1004 1008 1009 100C 100D 100E 100F 1010 1011 1012 1013 1014 101 5 1016 1017 1018 1019 101A 101D 101E 1026 1027 1028 1049 104A 104B 104C 104D 105E 105F 1060 1075 1076 1077 1078 1079 107A 107B 107C 107D 107E 107F 108A 108B 108C 1096 1098 1099 109A 10A 4 10A5 10A7 10A9 10B5 10B9 10BA 10BB 10BC 10BD 10BF 10C0 10C2 10C3 10C4 10C5 10C6 10C7 10C8 10C910CB 10CC 10CD 10CE 10D3 10D5 10D6 10D9 10DA 10DB 10DD 10DE 10DF 10E1 10E5 10E6 10E7 10E8 10EA 10EB 10EC 10EF 10F0 10F1 10F4 10F5 10F6 10F7 10F8 10F9 10FA 10FB 10FC 1501 1502 1503 1507 150A 150B 150C 150D 150E 150F 1510 1511 1514 1516 1517 1518 151C 1521 1522 1523 1524 1525 1526 1527 1527 1528 1529 152A 1533 1534 1535 1536 1537 1538 1539 153A 153B 1546 154A 154D 1557 1558 1559155A 1560 157B 157C 1F40 1F41 1F45 294C

供应商 ID 10EC，Realtek Semiconductor Corp.

对于供应商 ID 10EC，支持以下设备 ID：

8136 8137 8167 8168 8169

供应商 ID 14E4，Broadcom

对于供应商 ID 14E4，支持以下设备 ID：

1600 1601 1639 163A 163B 163C 1644 1645 1646 1647 1648 164A 164C 164D 1653 1654 1655 1656 1655 7 1658 1659 165A 165B 165C 165D 165E 165F 1668 1669 166A 166B 166D 166E 1672 1673 1674 1676 1677 1678 1679 167A 167B 167C 167D 167F 1680 1681 1684 1688 1690 1691 1692 1693 1694 1696 1698 1698 1699 169A 169B 169D 16A0 16A6 16A7 16A8 16AA 16AC 16B0 16B1 16B2 16B4 16B5 16B6 16C6 16C7 16DD16F7 16FD 16FE 16FF 170D 170E 170F

供应商 ID 1969，Atheros Communications

对于供应商 ID 1969，支持以下设备 ID：

1062 1063 1073 1083 1090 1091 10A0 10A1 10B0 10B1 10C0 10C1 10D0 10D1 10E0 10E1 10F0 10F1 2060 2062 E091 E0A1 E0B1 E0C1 E0D1 E0E1 E0F1

供应商 ID 19A2，ServerEngines (Emulex)

对于供应商 ID 19A2，支持以下设备 ID：

0211 0215 0221 0700 0710

供应商 ID 10DF，Emulex Corporation

对于供应商 ID 10DF，支持以下设备 ID：

0720 E220

供应商 ID 15B3，Mellanox Technology

对于供应商 ID 15B3，支持以下设备 ID：

1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 100A 100B 100C 100D 100E 100F 1010 6340 63 41 634A 634B 6354 6368 6369 6372 6732 6733 673C 673D 6746 6750 6751 675A 6764 6765 676E 6778

另请参阅

自动设置 KDNET 网络内核调试

Windows 8 中的网络内核调试支持的以太网 NIC

Windows 8 中的网络内核调试支持的以太网 NIC

项目 • 2024/01/10

当目标计算机运行Windows 8时，可以通过以太网网线进行内核调试。 目标计算机必须具有受支持的网络接口卡 (NIC) 或网络适配器。

在内核调试期间，运行调试器的计算机称为 *主计算机*，正在调试的计算机称为 *目标计算机*。 有关详细信息，请参阅 设置 KDNET 网络内核自动调试。

若要通过网络电缆进行内核调试，目标计算机必须具有受支持的网络适配器。 当目标计算机运行Windows 8时，支持此处列出的网络适配器进行内核调试。

注意有关Windows 8.1支持的用于内核调试的网络适配器的列表，请参阅 Windows 8.1 中支持用于网络内核调试的以太网 NIC。

系统要求

通过以太网 NIC 进行内核调试需要某些低级别的平台支持。 对于此调试解决方案，Windows 要求通过 PCI/PCIe 附加这些 NIC。 在大多数情况下，只需插入其中一个受支持的 NIC 即可获得可靠的内核调试体验。 但是，在某些情况下，BIOS 配置详细信息可能会阻碍 Windows 调试路径。 应考虑以下一组平台要求：

- 系统固件应发现并配置 NIC 设备，使其资源不会与已配置 BIOS 的任何其他设备冲突。
- 系统固件应将 NIC 的资源放置在未标记为可预提取的地址窗口下。

查找供应商 ID 和设备 ID

首先在目标计算机上查找网络适配器的供应商 ID 和设备 ID。

- 在目标计算机上，打开设备管理器 (命令提示符窗口中输入 **devmgmt**) 。
- 在“设备管理器”中，找到要用于调试的网络适配器。
- 右键单击网络适配器节点，然后选择 “**属性**”。
- 在“ **详细信息** ”选项卡的 “**属性**”下，选择“ **硬件 ID**”。

供应商和设备 ID 显示为 VEN_*VendorID* 和 DEV_*DeviceID*。 例如，如果看到 PCI\VEN_8086&DEV_104B，则供应商 ID 为 8086，设备 ID 为 104B。

供应商 ID 8086，Intel Corporation

对于供应商 ID 8086，支持以下设备 ID：

1000 1001 1004 1008 1009 100C 100D 100E 100F 1010 1011 1012 1013 1014 1015 1016 1017 1018 10 19 101A 101D 101E 1026 1027 1028 1049 104A 104B 104C 104D 105E 105F 1060 1075 1076 1077 1077 8 1079 107A 107B 107C 107D 107E 107F 108A 108B 108C 1096 1098 1099 109A 10A4 10A5 10A7 10A9 10A9 10B5 10B9 10BA 10BB 10BC 10BD 10BF 10C9 10CB 10CC 10CD 10CE 10D3 10D5 10D6 10D9 10DA 10E5 10E6 10E710E8 10EA 10EB 10EF 10F0 10F5 10F6 1501 1502 1503 150A 150C 150D 150E 150F 1510 1511 1516 1518 1521 1522 1523 1524 1526 294C

供应商 ID 10EC，Realtek Semiconductor Corp.

对于供应商 ID 10EC，支持以下设备 ID：

8136 8137 8167 8168 8169

供应商 ID 14E4，Broadcom

对于供应商 ID 14E4，支持以下设备 ID：

1600 1601 1639 163A 163B 163C 1644 1645 1646 1647 1648 164A 164C 164D 1653 1654 1655 1656 1655 7 1658 1659 165A 165B 165C 165D 165E 165F 1668 1669 166A 166B 166D 166E 1672 1673 1674 1676 1677 1678 1679 167A 167B 167C 167D 167F 1680 1681 1684 1688 1690 1691 1692 1693 1694 1696 1698 1698 1699 169A 169B 169D 16A0 16A6 16A7 16A8 16AA 16AC 16B0 16B1 16B2 16B4 16B5 16B6 16C6 16C7 16DD16F7 16FD 16FE 16FF 170D 170E 170F

供应商 ID 1969，Atheros Communications

对 Atheros 网络适配器的支持由 Qualcomm 提供的单独模块提供。 支持这些设备 ID。

1062 1063 1073 1083 1090 1091 10A0 10A1 10B0 10B1 10C0 10C1 10D0 10D1 10E0 10E1 10F0 10F1 2060 2062 E091 E0A1 E0B1 E0C1 E0D1 E0E1 E0F1

另请参阅

自动设置 KDNET 网络内核调试

Windows 8.1 中的网络内核调试支持的以太网 NIC

Windows 10 中的网络内核调试支持的以太网 NIC

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

Windows driver documentation 反馈

Windows driver documentation 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

配置 tools.ini

项目 • 2024/07/08

文件tools.ini包含初始化命令行调试器的信息。 启动时，调试器会在tools.ini文件中搜索相应的节标头，并从标头下的条目中提取初始化信息。 每个命令行调试器都有自己的节标头 - [CDB]、[NTSD]和 [KD]。 环境变量 INIT 必须指向包含tools.ini文件的目录。

WinDbg 不使用tools.ini文件。 相反，WinDbg 会在工作区中保存初始化设置。

下表显示了tools.ini项。

关键字必须用空格或冒号分隔值。 关键字不区分大小写。

对于 **TRUE** 或 **FALSE** 值，“FALSE”是唯一的 false 值。 任何其他内容都是 **真的**。

展开表

条目	说明
$u 0： *值*... **$u 9：** *值*	将值分配给固定名称别名。 可以指定数值 *n* 或 *0xn* 或任何其他字符串。 有关详细信息，请参阅 “使用别名 ”。 无命令行等效项。
DebugChildren： *标志*	**TRUE** 或 **FALSE**。 如果 **为 TRUE，CDB** 将调试指定的应用程序以及它可能生成的任何子进程。 命令行等效项为 **-o**。
DebugOutput： *标志*	**TRUE** 或 **FALSE**。 如果 **为 TRUE**，则 CDB 通过终端发送输出并接收输入。 如果 **为 FALSE**，输出将转到用户屏幕。 命令行选项 **-d** 类似，但不完全相同。
IniFile： *文件*	指定 CDB 或 KD 在启动时从中获取命令的脚本文件的名称。 默认值为当前目录中ntsd.ini文件。 命令行等效项为 **-cf**。有关详细信息，请参阅 “使用脚本文件”。
LazyLoad： *flag*	**TRUE** 或 **FALSE**。 如果 **为 TRUE，CDB** 将执行延迟符号加载;也就是说，在需要之前不会加载符号。 命令行等效项为 **-s**。 有关详细信息，以及设置此选项的其他方法，请参阅 延迟符号加载。
SetDll： *filename*	设置扩展 DLL。 应省略.dll文件扩展名。 默认值为userexts.dll。 命令行等效项为 **-a**。

条目	说明
	有关详细信息以及设置此默认值的其他方法，请参阅加载调试器扩展 DLL。
StopFirst： *标志*	**TRUE** 或 **FALSE**。 如果 **为 true，CDB** 会在图像加载过程结束时在断点上停止。 命令行等效项为 **-g**。
StopOnProcessExit： *标志*	**TRUE** 或 **FALSE**。 如果 **为 TRUE**，则 CDB 在收到进程终止通知时停止。 命令行等效项为 **-G**。
sxd：*event sxe：* *event*	设置调试器响应和指定异常或事件的处理状态。 可以通过以下方式指定异常和事件： *：默认异常 *n*：异常 *n* （十进制） *0xn*：异常 *0xn* （十六进制） （其他）： 事件代码 有关此过程的详细信息以及控制这些设置的其他方法，请参阅 控制异常和事件 。
VerboseOutput： *标志*	**TRUE** 或 **FALSE**。 如果 **为 TRUE，CDB** 将显示有关符号处理、事件通知和其他运行时事件的详细信息。 命令行等效项为 **-v**。
行：*标志*	**TRUE** 或 **FALSE**。 行标志启用或禁用对源行信息的支持。
srcopt： *options*	设置控制源显示和程序单步执行选项的源行选项。 有关详细信息，请参阅 **l+、l- （设置源选项）。**
srcpath： *directory*	设置源文件搜索路径。 有关详细信息，请参阅 **.srcpath、.lsrcpath （设置源路径）。**
enable_unicode：*标志*	**TRUE** 或 **FALSE**。 enable_unicode标志指定调试器是否将 USHORT 指针和数组显示为 Unicode 字符串。
force_radix_output：*标志*	**TRUE** 或 **FALSE**。 force_radix_output标志指定是以十进制格式还是默认弧度显示整数。
col_mode：*标志*	**TRUE** 或 **FALSE**。 col_mode标志控制颜色模式设置。 启用颜色模式后，调试器可以生成彩色输出。 默认情况下，大多数颜色未设置，而是默认为当前控制台颜色。
col： *name colspec*	该 *名称* 指示要着色的元素。 *colspec* 是 [rR-][gG-][bB-] 形式的三字母 RGB 指示器。 小写字母表示较深，大写字母指示更亮，短划线表示无颜色分量贡献。 由于主机颜色限制，亮度实际

条目	说明
	上不是按组件，而是适用于所有组件（如果有任何请求亮）。 换句话说，rgB 与 RGB 相同。 出于此原因，建议在使用任何上限时使用所有上限。 示例用法： col： emphfg R--

tools.ini 文件中的示例 [NTSD] 节如下：

INF

```
[NTSD]
sxe: 3c
sxe: cc
$u0: VeryLongName
VerboseOutput:true
```

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用 KDbgCtrl

项目 • 2024/05/27

KDbgCtrl（内核调试控件， kdbgctrl.exe）工具可用于从目标计算机控制内核调试连接。

要使用该工具，目标计算机必须运行 Windows Server 2003 或更高版本的 Windows。

KDbgCtrl 可以控制五种不同的设置：完全内核调试、自动内核调试、用户模式错误处理、阻止内核调试以及 DbgPrint 缓冲区的大小。

要使用 KDbgCtrl，必须在最后一次启动前在目标计算机的启动设置中启用内核调试。 否则，将无法使用 KDbgCtrl 来启用内核调试。 有关这些启动设置的详细信息，请参阅用于启用调试的启动参数。

完全内核调试

启用"完全内核调试"后，主计算机机上运行的内核调试器可中断至目标计算机。 如果出现内核模式异常，目标计算机将中断至内核调试器。 还允许从目标计算机向主机发送消息，如 **DbgPrint** 输出、符号加载消息和重定向的用户模式调试器。

如果禁用此设置，则目标计算机将忽略来自主机的所有信号。

完全内核调试默认已启用。 要检查当前设置值，请使用 **kdbgctrl -c**。 要禁用此设置，请使用 **kdbgctrl -d**。 要启用此设置，请使用 **kdbgctrl -e**。

如果要检查当前设置并用它来控制批处理文件的执行，可以使用 **kdbgctrl -cx** 命令。 有关此命令的详细信息，请参阅 **KDbgCtrl 命令行选项**。

自动内核调试

如果启用了完全内核调试，则当前的自动内核调试设置将变得无关紧要 - 所有通信都会被允许。

当禁用"完全内核调试"并启用"自动内核调试"时，只有目标计算机才能启动调试连接。

在这种情况下，只有内核模式异常、断点或其他内核模式事件才会导致建立连接。 对于 **DbgPrint** 输出、符号加载消息、重定向用户模式调试器输入和输出或其他类似消息，将不会建立连接，这些消息将存储在 DbgPrint 缓冲区中，而不会被发送到内核调试器。

如果异常或事件导致目标计算机中断至内核调试器，则完全内核调试将自动打开，就像执行了 **kdbgctrl -e** 一样。

默认情况下，自动内核调试已被禁用（但这并不重要，除非完全内核调试也被禁用）。 要检查当前设置值，请使用 **kdbgctrl -ca**。 要禁用此设置，请使用 **kdbgctrl -da**。 要启用此设置，请使用 **kdbgctrl -ea**。

用户模式错误处理

启用用户模式错误处理后，某些用户模式事件会导致目标计算机中断至内核调试器。

具体而言，所有 **int 3** 中断（如调试器插入代码中的断点或调用 **DbgBreakPoint**）都将导致中断至内核调试器。 但是，标准异常（例如访问冲突和除以零）通常不会被发送到内核调试器。

如果用户模式调试器已附加到进程，则此调试器将捕获所有用户模式错误，并且不会更改内核调试器。 有关各种用户模式错误处理程序的优先级，请参阅启用事后调试。

要使用户模式错误处理发挥作用，必须同时启用完全内核调试或自动内核调试。

默认情况下，用户模式错误处理处于启用状态。 要检查当前设置值，请使用 **kdbgctrl -cu**。 要禁用此设置，请使用 **kdbgctrl -du**。 要启用此设置，请使用 **kdbgctrl -eu**。

阻止内核调试

某些情况下可能需要设置目标计算机进行内核调试，但要等到目标计算机启动后再启用内核调试。 可以通过阻止内核调试来实现这一点。

> **ⓘ 重要**
>
> 在使用 BCDEdit 更改启动信息之前，可能需要在测试电脑上暂时挂起 Windows 安全功能，例如 BitLocker 和安全启动。 测试完成后重新启用这些安全功能，并在禁用安全功能时适当地管理测试电脑。

要阻止内核调试，请使用类似以下的命令来设置目标计算机：

控制台

```
bcdedit /debug on
bcdedit /dbgsettings 1394 channel:32 /start DISABLE /noumex
```

重新启动目标计算机时，它将为内核调试做好准备，但内核调试和用户模式错误处理将被禁用。 此时，主计算机将无法连接到目标计算机，内核调试器无法捕捉错误检查，而用户模式异常也不会导致中断至内核调试器。

准备就绪后，输入以下命令即可启用内核调试（无需重启目标计算机）。

控制台

```
kdbgctrl -db
kdbgctrl -e
```

之后，可以通过输入以下命令禁用内核调试。

控制台

```
kdbgctrl -d
kdbgctrl -eb
```

可以使用 **kdbgctrl -cb** 检查内核调试是否被阻止。

DbgPrint 缓冲区大小

DbgPrint 缓冲区会存储目标计算机发送给内核调试器的消息。

如果启用了完全内核调试，这些消息将自动显示在内核调试器中。 但如果禁用了该选项，则这些消息将存储在缓冲区中。 稍后，可以启用内核调试，连接到内核调试器，并使用 **!dbgprint** 扩展来查看该缓冲区的内容。 有关该缓冲区的详细信息，请参阅 DbgPrint 缓冲区。

在 Windows 的免费版本中，DbgPrint 缓冲区的默认大小为 4 KB。 要确定当前缓冲区大小，请使用 **kdbgctrl -cdb**。 要更改缓冲区大小，请使用 **kdbgctrl -sdb***Size*，其中 *Size* 用于指定新的缓冲区大小。 有关语法详细信息，请参阅 **KDbgCtrl 命令行选项**。

示例

要显示所有当前设置，请使用以下命令：

控制台

```
kdbgctrl -c -ca -cu -cb -cdb
```

要恢复默认设置，请使用以下命令：

控制台

```
kdbgctrl -e -da -eu -db -sdb 0x1000
```

要锁定主机，使其只能在异常时与之联系，请使用以下命令：

控制台

```
kdbgctrl -d -ea -eu
```

要禁用所有内核调试，请使用以下命令：

控制台

```
kdbgctrl -d -da
```

如果要禁用所有内核调试，可能还需要增加 DbgPrint 缓冲区的大小。 这样可以确保保存所有消息，以备日后查看。 如果有一兆字节的内存要备用，则可以使用以下命令：

控制台

```
kdbgctrl -sdb 0x100000
```

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

调试 Windows 驱动程序分步实验室（Echo 内核模式）

项目 • 2024/04/18

本实验室介绍 WinDbg 内核调试器。 使用 WinDbg 调试 echo 内核模式示例驱动程序代码。

实验室目标

本实验室包括介绍调试工具、教授常见调试命令、说明断点的使用以及演示如何使用调试扩展的练习。

在本实验室中，您将使用实时内核调试连接来浏览以下操作：

- 使用 Windows 调试器命令
- 使用标准命令（调用堆栈、变量、线程、IRQL）
- 使用高级驱动程序调试命令 (!commands)
- 使用符号
- 在实时调试中设置断点
- 查看调用堆栈
- 显示即插即用设备树
- 使用线程和进程上下文

用户和内核模式调试

使用 Windows 调试器时，可以执行两种类型的调试：

用户模式- 计算机上的应用程序和子系统在用户模式下运行。 在用户模式下运行的进程在其自己的虚拟地址空间中运行。 它们被限制直接访问系统的许多部分，包括系统硬件、未分配供其使用的内存以及可能危及系统完整性的系统其他部分。 因为在用户模式下运行的进程与系统和其他用户模式进程有效隔离，因此它们不会干扰这些资源。

内核模式- 操作系统和特权程序在内核模式下运行。 内核模式代码有权访问系统的任何部分。 它不像用户模式代码那样受到限制 它可以访问在用户模式或内核模式下运行的任何其他进程的任何部分。 许多核心操作系统功能和许多硬件设备驱动程序在内核模式下运行。

本练习介绍在用户模式和内核模式调试期间经常使用的调试命令。 本练习还介绍了调试扩展（有时称为 *！命令*）用于内核模式调试。

实验室设置

完成实验需要以下硬件：

- 运行 Windows 10 的笔记本电脑或台式计算机（主机）
- 第二台运行 Windows 10 的笔记本电脑或台式计算机（目标）
- 用于连接两台计算机的网络中心或路由器和网络电缆
- 访问 Internet 以下载符号文件

完成实验需要以下软件：

- Visual Studio
- 适用于 Windows 10 的 Windows 软件开发工具包 (SDK)
- 适用于 Windows 10 的 Windows 驱动程序工具包 (WDK)
- 适用于 Windows 10 的示例 echo 驱动程序

该实验室包含以下部分：

- 连接到内核模式 WinDbg 会话
- 内核模式调试命令和技术
- 下载并构建 KMDF echo 驱动程序
- 在目标系统上安装 echo 驱动程序示例
- 使用 WinDbg 显示有关驱动程序的信息
- 显示即插即用设备树信息
- 使用断点和源代码
- 查看变量和调用堆栈
- 显示进程和线程
- IRQL、注册和结束 WinDbg 会话
- Windows 调试资源

连接到内核模式 WinDbg 会话

在本部分中，在主机和目标系统上配置网络调试。

此实验室中的计算机需要配置为使用以太网网络连接进行内核调试。

此实验室使用两台计算机。 Windows 调试器在*主机*系统上运行，内核模式驱动程序框架 (KMDF) echo 驱动程序在*目标*系统上运行。

使用网络中心或路由器和网络电缆来连接两台计算机。

要使用内核模式应用程序和 WinDbg，建议使用通过以太网传输的 KDNET。 有关如何使用以太网传输协议的信息，请参阅开始使用 WinDbg（内核模式）。 有关设置目标计算机的详细信息，请参阅为手动部署驱动程序准备计算机以及自动设置 KDNET 网络内核调试。

通过以太网配置内核模式调试

要在目标系统上启用内核模式调试，请执行以下操作：

1. 在主机系统上打开命令提示符窗口，键入 *ipconfig* 以确定其 IP 地址。

   输出

   ```
   Windows IP Configuration
   Ethernet adapter Ethernet:
      Connection-specific DNS Suffix  . :
      Link-local IPv6 Address . . . . . : fe80::c8b6:db13:d1e8:b13b%3
      Autoconfiguration IPv4 Address. . : 169.182.1.1
      Subnet Mask . . . . . . . . . . . : 255.255.0.0
      Default Gateway . . . . . . . . . :
   ```

2. 记录下主机系统的 IP 地址：_____

3. 在目标系统上打开命令提示符窗口，然后使用 `ping` 命令来确认两个系统之间的网络连接。

   控制台

   ```
   ping 169.182.1.1
   ```

   使用记录下的主机系统的实际 IP 地址，而不是示例输出中显示的 169.182.1.1。

   输出

   ```
   Pinging 169.182.1.1 with 32 bytes of data:
   Reply from 169.182.1.1: bytes=32 time=1ms TTL=255
   Reply from 169.182.1.1: bytes=32 time<1ms TTL=255
   Reply from 169.182.1.1: bytes=32 time<1ms TTL=255
   Reply from 169.182.1.1: bytes=32 time<1ms TTL=255

   Ping statistics for 169.182.1.1:
   ```

```
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 1ms, Average = 0ms
```

完成以下步骤，在目标系统上启用内核模式调试。

> **ⓘ 重要**
>
> 在使用 BCDEdit 更改启动信息之前，可能需要在测试计算机上暂时挂起 BitLocker 和安全启动等 Windows 安全功能。 在测试完成后重新启用这些安全功能。 在禁用安全功能时，适当管理测试计算机。 UEFI 中通常已禁用安全启动。 要访问 UEFI 设置，请使用系统、恢复、高级启动。 重新启动后，选择“故障排除”、“高级选项”、“UEFI 固件设置”。 请谨慎操作，因为错误设置 UEFI 选项或禁用 BitLocker 可能会导致系统无法运行。

1. 在目标计算机上，以管理员身份打开“命令提示符”窗口。 输入此命令可启用调试：

   控制台

   ```
   bcdedit /set {default} DEBUG YES
   ```

2. 输入此命令可启用测试签名：

   控制台

   ```
   bcdedit /set TESTSIGNING ON
   ```

3. 输入此命令可以设置主机系统的 IP 地址。 使用之前记录的主机系统 IP 地址，而不是显示的 IP 地址。

   控制台

   ```
   bcdedit /dbgsettings net hostip:192.168.1.1 port:50000
   key:2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
   ```

   > **⚠ 警告**
   >
   > 为提高连接的安全性并降低客户端调试器随机连接请求的风险，请使用自动生成的随机密钥。 有关详细信息，请参阅**自动设置 KDNET 网络内核调试**。

4. 输入此命令以确认 `dbgsettings` 的值已正确设置：

控制台

```
bcdedit /dbgsettings
```

输出

```
key
2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
debugtype               NET
hostip                  169.168.1.1
port                    50000
dhcp                    Yes
The operation completed successfully.
```

备注

如果收到来自防火墙的信息并希望使用调试器，请选中所有三个方框。

5. 在主计算机上，以管理员身份打开命令提示符窗口。 此实验室使用 Windows 驱动程序工具包 (WDK) 中 *WinDbg.exe* 的 x64 版本，该版本已作为 Windows 工具包安装的一部分安装。 更改为默认的 WinDbg 目录，默认位置如下所示。

控制台

```
cd C:\Program Files(x86)\Windows Kits\10\Debuggers\x64
```

此实验室假设在目标和主机两台计算机上都运行 64 位版本的 Windows。 如果情况并非如此，最好的办法是在主机上运行与目标运行相同*位数*的工具。 例如，如果目标计算机运行的是 32 位 Windows，请在主机计算机上运行 32 位版本的调试器。 有关详细信息，请参阅选择 32 位或 64 位调试工具。

6. 使用以下命令通过远程用户调试来打开 WinDbg。 密钥和端口的值与之前在目标计算机上使用 BCDEdit 设置的值一致。

控制台

```
WinDbg -k
net:port=50000,key=2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
```

7. 重新启动目标系统。

8. 一两分钟后，主机系统上就会显示调试输出。

输出

```
Microsoft (R) Windows Debugger Version 10.0.17074.1002 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Using NET for debugging
Opened WinSock 2.0
Waiting to reconnect...
Connected to target 169.182.1.1 on port 50005 on local IP 169.182.1.2
You can get the target MAC address by running .kdtargetmac command.
Connected to Windows 10 16299 x64 target at (Wed Feb 28 17:16:23.051
2018 (UTC - 8:00)), ptr64 TRUE
Kernel Debugger connection established.  (Initial Breakpoint requested)
Symbol search path is: srv*
Executable search path is:
Windows 10 Kernel Version 16299 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 16299.15.amd64fre.rs3_release.170928-1534
Machine Name:
Kernel base = 0xfffff800`9540d000 PsLoadedModuleList =
0xfffff800`95774110
Debug session time: Wed Feb 28 17:16:23.816 2018 (UTC - 8:00)
System Uptime: 0 days 0:00:20.534
```

“调试器命令”窗口是 WinDbg 的主要调试信息窗口。 可以在此窗口中输入调试器命令并查看命令输出。

调试器命令窗口被分为两个窗格。 在较小的窗格（即窗口底部的命令输入窗格）中输入命令，并在窗口顶部较大的窗格中查看命令输出。

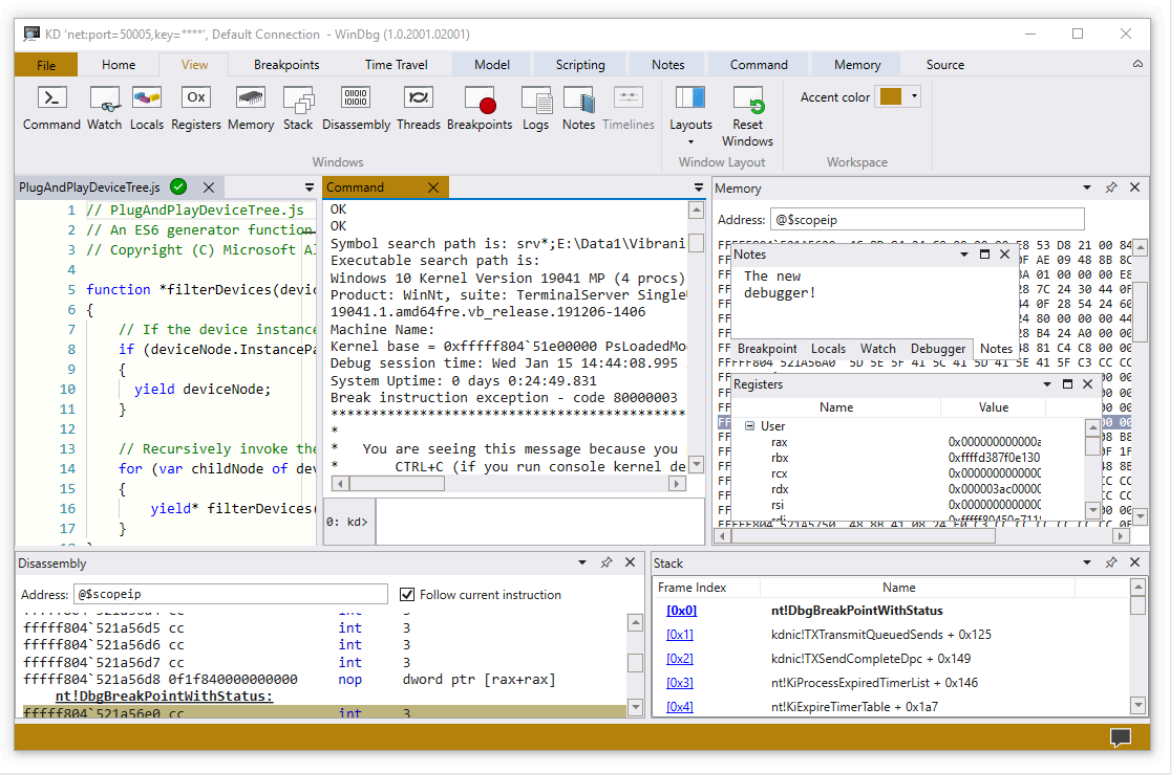
在命令输入窗格中，使用上箭头和下箭头键可以滚动浏览命令历史记录。 命令出现后，可对其进行编辑，或者按 Enter 运行命令。

内核模式调试命令和技术

在本部分中，将使用调试命令来显示目标系统的相关信息。

某些调试命令使用调试器标记语言 (DML) 显示文本，可以选择这些文本来快速收集更多信息。

1. 在主机系统上，使用 WinDBg 中的 Ctrl+Scroll Lock 键中断目标系统上运行的代码。 目标系统可能需要一些时间才会响应。



2. 在调试器命令窗口中输入以下命令以启用 DML：

```
dbgcmd
```

```
0: kd> .prefer_dml 1
DML versions of commands on by default
```

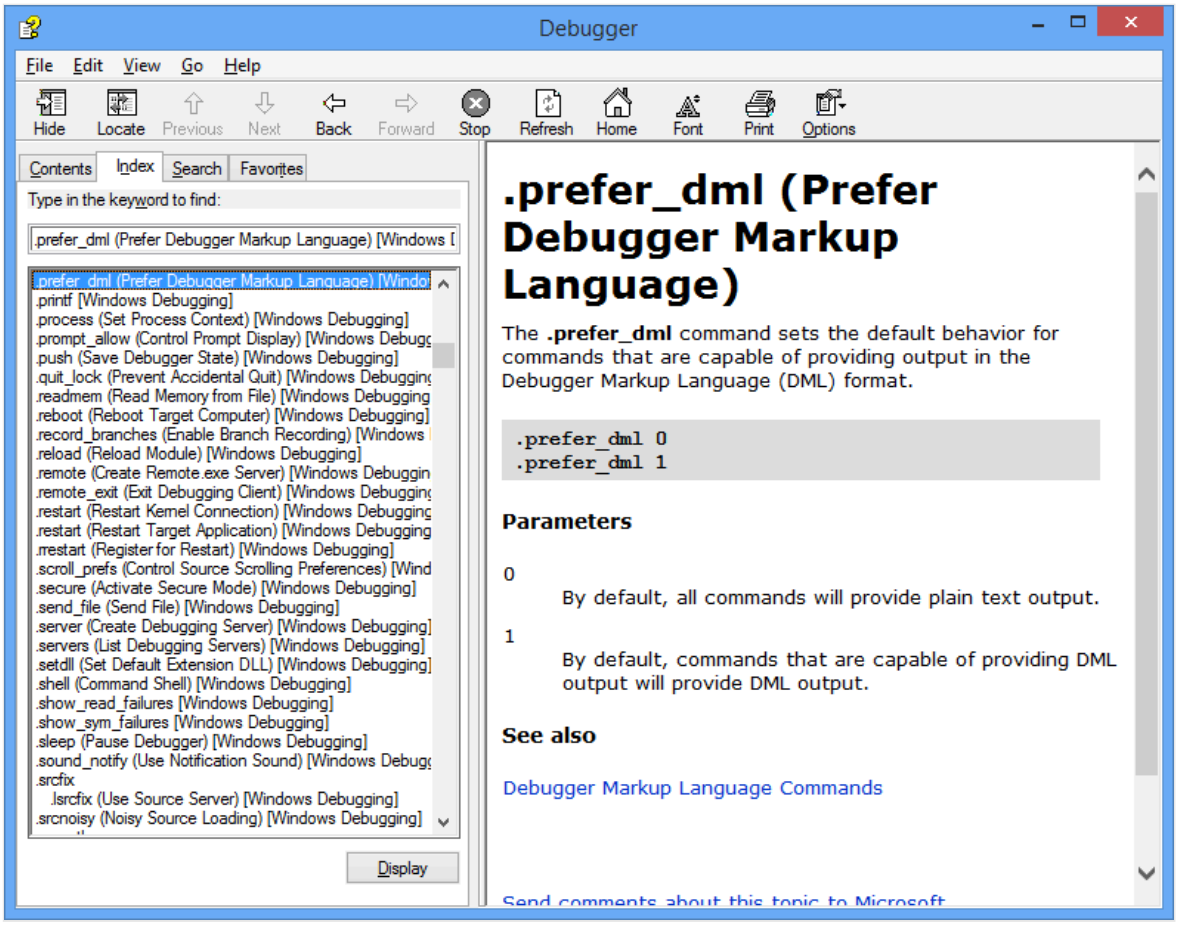
3. 可使用 `.hh` 命令来访问参考命令帮助。 输入以下命令以查看 `.prefer_dml` 的命令参考帮助：

```
```

```
dbgcmd
```

```
0: kd> .hh .prefer_dml
```

调试器帮助文件显示了 `.prefer_dml` 命令的帮助。



4. 要显示目标系统的详细版本信息，请在 WinDbg 窗口中输入 vertarget（显示目标计算机版本） 命令：

```
dbgcmd
```

```
0: kd> vertarget
Windows 10 Kernel Version 9926 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 9926.0.amd64fre.fbl_awesome1501.150119-1648
Machine Name: ""
Kernel base = 0xfffff801`8d283000 PsLoadedModuleList =
0xfffff801`8d58aef0
Debug session time: Fri Feb 20 10:15:17.807 2015 (UTC - 8:00)
System Uptime: 0 days 01:31:58.931
```

5. 要验证内核模式进程是否正确，请在 WinDbg 窗口输入 lm（列出加载的模块）命令，以显示已加载的模块：

```
dbgcmd
```

```
0: Kd> lm
start             end                 module name
fffff801`09200000 fffff801`0925f000   volmgrx    (no symbols)
fffff801`09261000 fffff801`092de000   mcupdate_GenuineIntel   (no
symbols)
fffff801`092de000 fffff801`092ec000   werkernel   (export symbols)
werkernel.sys
fffff801`092ec000 fffff801`0934d000   CLFS       (export symbols)
CLFS.SYS
fffff801`0934d000 fffff801`0936f000   tm         (export symbols)
tm.sys
fffff801`0936f000 fffff801`09384000   PSHED      (export symbols)
PSHED.dll
fffff801`09384000 fffff801`0938e000   BOOTVID    (export symbols)
BOOTVID.dll
fffff801`0938e000 fffff801`093f7000   spaceport   (no symbols)
fffff801`09400000 fffff801`094cf000   Wdf01000   (no symbols)
fffff801`094d9000 fffff801`09561000   CI         (export symbols)
CI.dll
...
```

此实验室中省略的输出用“...”表示。

6. 要获取特定模块的详细信息，请使用 `v`（详细）选项：

dbgcmd

```
0: Kd> lm v m tcpip
Browse full module list
start             end                 module name
fffff801`09eeb000 fffff801`0a157000   tcpip      (no symbols)
    Loaded symbol image file: tcpip.sys
    Image path: \SystemRoot\System32\drivers\tcpip.sys
    Image name: tcpip.sys
    Browse all global symbols  functions  data
    Timestamp:        Sun Nov 09 18:59:03 2014 (546029F7)
    CheckSum:         00263DB1
    ImageSize:        0026C000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4

Unable to enumerate user-mode unloaded modules, Win32 error 0n30
```

未设置符号路径和加载符号，因此调试器中能提供的信息非常有限。

下载并构建 KMDF echo 驱动程序

在本部分中，下载并生成 KMDF echo 驱动程序。

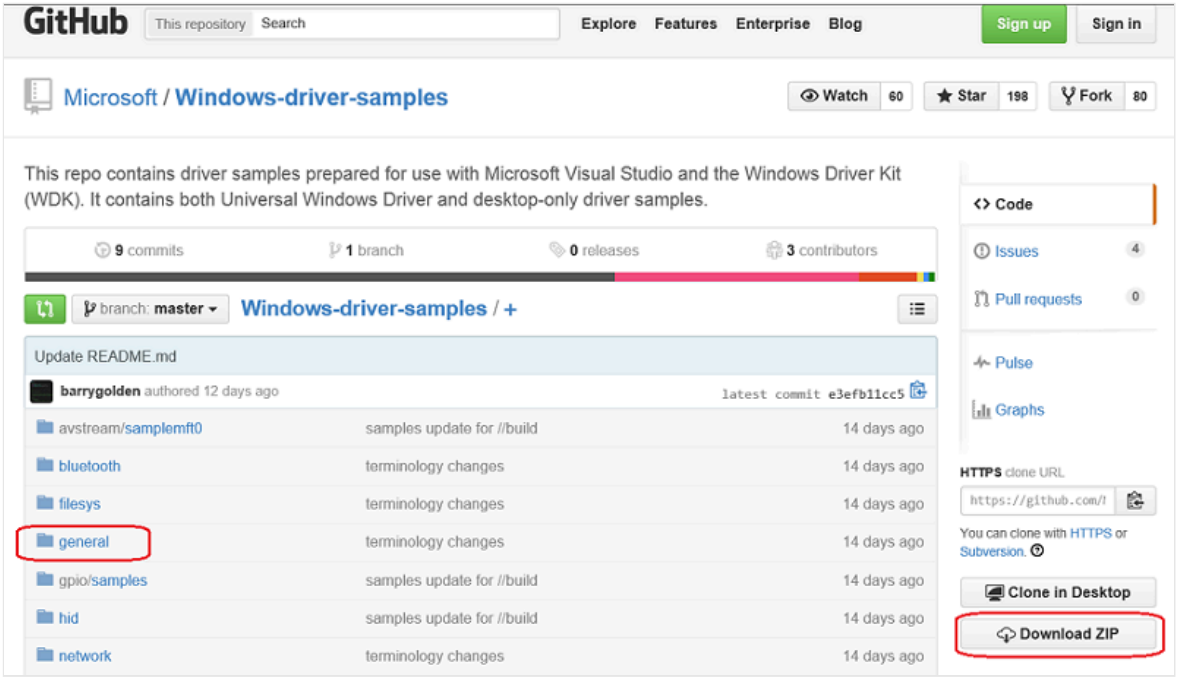
通常情况下，在使用 WinDbg 时需要用自己的驱动程序代码。 为了熟悉 WinDbg 的操作，本实验室使用 KMDF 模板“Echo”示例驱动程序。 为帮助理解 WinDbg 中显示的信息，我们提供了源代码。 此示例还用于说明如何通过本地内核模式代码进行单步操作。 此技术对于调试复杂的内核模式代码问题非常有用。

下载并生成 Echo 示例音频驱动程序：

1. 从 GitHub 下载并提取 KMDF Echo 示例。

   - 查看 GitHub 中的 echo 示例。
   - 阅读有关示例的信息。
   - 浏览所有 Windows 驱动程序示例。

   KMDF Echo 示例位于*常规*文件夹中。



   a. 以一个 zip 文件下载驱动程序示例：驱动程序示例

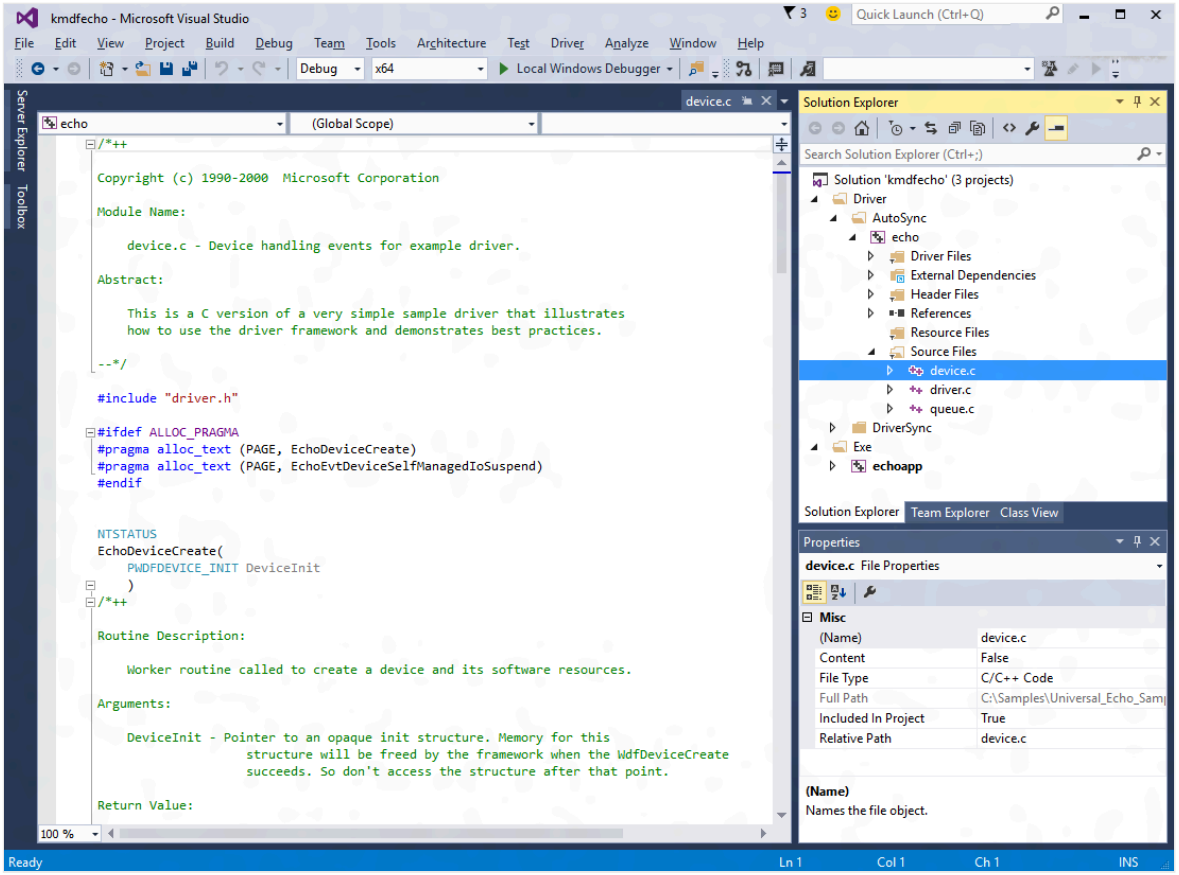
   b. 将 zip 文件下载到本地硬盘。

   c. 选中并按住或右键单击 zip 文件，然后选择**全部解压缩**。 指定一个新文件夹，或浏览到一个现有文件夹来存储提取的文件。 例如，可以指定 *C:\DriverSamples* 作为提取文件的新文件夹。

   d. 提取文件后，转到到以下子文件夹：*C:\DriverSamples\general\echo\kmdf*

2. 在 Microsoft Visual Studio 中，选择**文件**>**打开**>**项目/解决方案**...，并转到包含提取文件的文件夹（例如，*C:\DriverSamples\general\echo\kmdf*）。 双击 *kmdfecho* 解决方案文件将其打开。

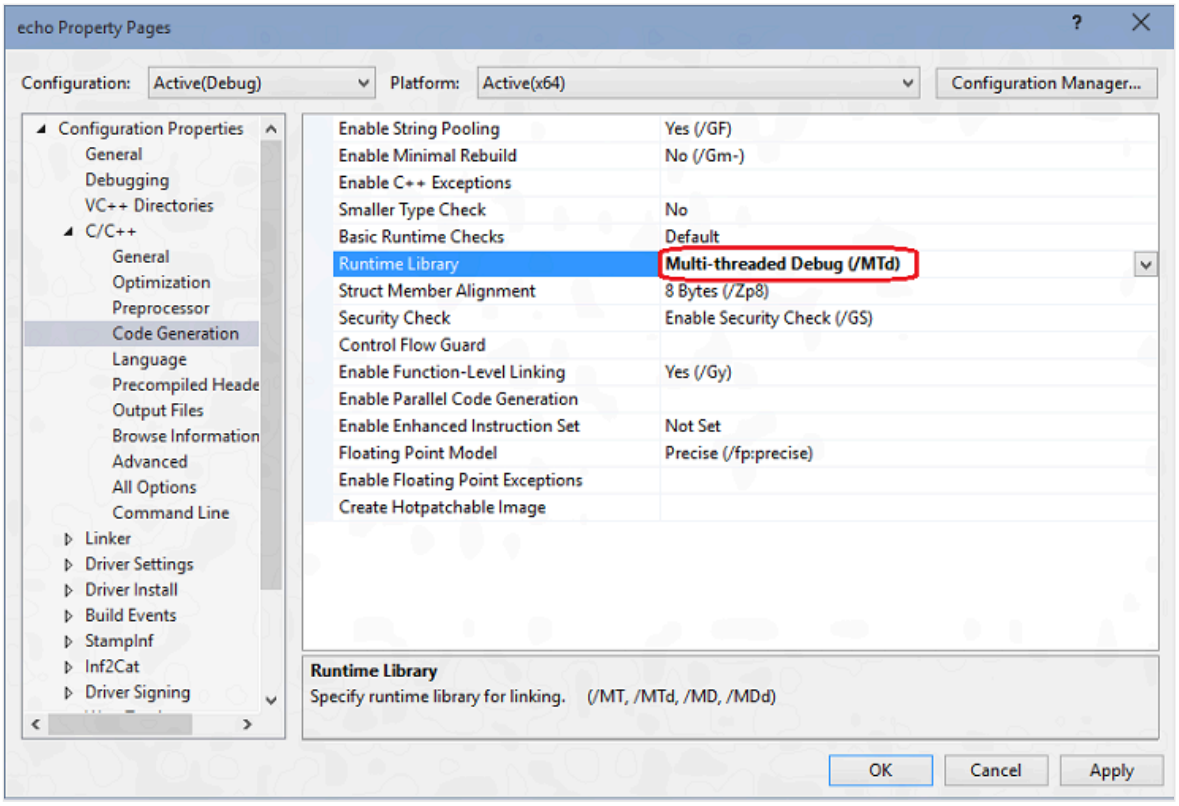
在 Visual Studio 中，找到“解决方案资源管理器”。 如果此窗口尚未打开，请从“视图”菜单中选择“解决方案资源管理器”。 在“解决方案资源管理器”中，可以看到一个包含三个项目的解决方案。



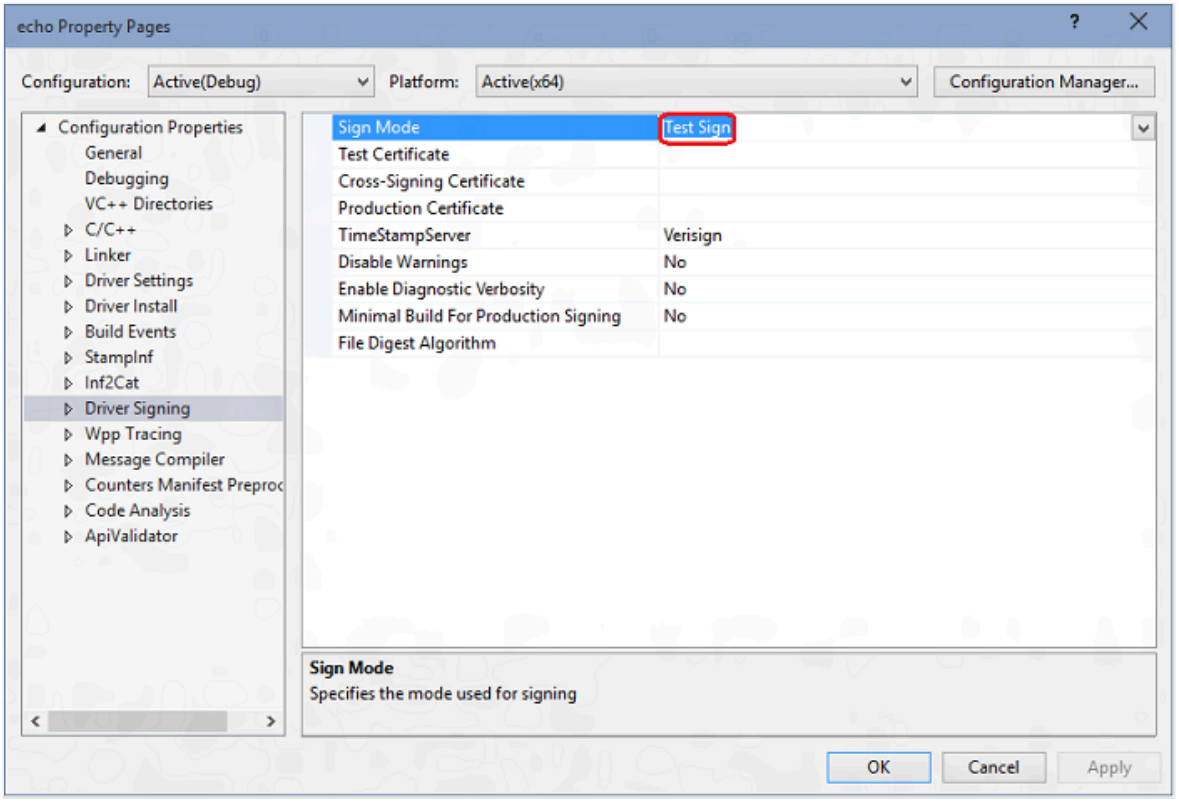
3. 设置示例的配置和平台。 在“解决方案资源管理器”中，选择并按住（或右键单击）**解决方案“kmdfecho”（3 个项目）**，然后选择**配置管理器**。 确保三个项目的配置和平台设置都相同。 默认情况下，所有项目的配置都设置为“Win10 调试”，而平台设置为“Win64”。 如果对一个项目进行任何配置或平台更改，请对其余三个项目进行同样的更改。

4. 需要修改驱动程序示例，以便使用不与现有驱动程序重叠的值。 请参阅从示例代码到生产驱动程序 - 示例中要更改的内容，了解如何创建与 Windows 中安装的现有真实驱动程序共存的独特驱动程序示例。

5. 设置运行时库。 打开 echo 驱动程序属性页面，找到 **C/C++>代码生成**。 将运行时库更改为多线程调试 (/MTd)。 有关生成选项的详细信息，请参阅 /MD、/MT、/LD（使用运行时库）。



6. 在驱动程序属性中，确保**驱动程序签名**>**签名模式**被设置为**测试签名**。



7. 在 Visual Studio 中，选择**生成**>**生成解决方案**。

   生成窗口应显示一条信息，指明所有三个项目均已成功生成。

提示

如果遇到生成错误信息，请使用生成错误编号来确定修复方法。 例如，***MSBuild error MSB8040*** 描述了如何使用 Spectre 已缓解的库。

8. 在文件资源管理器中，转到包含示例提取文件的文件夹。 例如，如果这是之前指定的文件夹，则转到 *C:\DriverSamples\general\echo\kmdf*。 在该文件夹中，编译的驱动程序文件的位置会因在“配置管理器”中选择的配置和平台设置而异。 如果不更改默认设置，那么编译后的驱动程序文件将保存到名为 *\x64\Debug* 的文件夹中，用于 64 位调试版本。

   转到包含自动同步驱动程序生成的文件的文件夹：*C:\DriverSamples\general\echo\kmdf\driver\AutoSync\x64\Debug*。

   文件夹中应包含这些文件：

   展开表

文件	说明
Echo.sys	驱动程序文件。
Echo.inf	一个信息 (INF) 文件，其中包含安装驱动程序所需的信息。

   另外，*echoapp.exe* 文件已经生成，它应该位于此位置：*C:\DriverSamples\general\echo\kmdf\exe\x64\Debug*。

   展开表

文件	说明
EchoApp.exe	与 echo.sys 驱动程序通信的命令提示符可执行文件测试文件。

9. 找到 U 盘或设置网络共享，将生成的驱动程序文件和测试 **EchoApp** 从主机复制到目标系统。

在下一部分中，将代码复制到目标系统，然后安装和测试驱动程序。

在目标系统上安装 KMDF echo 驱动程序示例

在本部分中，使用 DevCon 工具安装 echo 示例驱动程序。

安装驱动程序的计算机被称为*目标计算机*或*测试计算机*。 通常情况下，这台计算机与开发和生成驱动程序软件包的计算机是分开的。 开发和生成驱动程序的计算机被称为*主机计算机*。

将驱动程序软件包移动到目标计算机并安装驱动程序的过程被称为*部署*驱动程序。

在部署测试签名驱动程序之前，请通过启用测试签名让目标计算机做好准备。 还需要在 WDK 安装中找到 DevCon 工具，并将其复制到目标系统。

要在目标系统上安装驱动程序，请执行以下步骤。

在目标系统上，启用已测试签名的驱动程序：

1. 打开“**Windows 设置**”。
2. 在“更新和安全”中，选择“恢复”。
3. 在“高级启动”下，选择“立即重新启动”。
4. 在计算机重启时，选择“启动选项”。 在 Windows 10 中，选择**故障排除**>**高级选项**>**启动设置**，然后选择**重新启动**。
5. 按 F7 键，选择“禁用强制驱动程序签名”。
6. 重新启动目标计算机。

在主机系统上，转到 WDK 安装中的“工具”文件夹，然后找到 DevCon 工具。 例如，查看以下文件夹：*C:\Program Files (x86)\Windows Kits\10\Tools\x64\devcon.exe*。

在目标计算机上为生成的驱动程序包创建一个文件夹，例如 *C:\EchoDriver*。 将 *devcon.exe* 复制到目标系统。 在主机系统中找到 *.cer* 证书。 它位于主机上包含生成的驱动程序文件的同一文件夹中。 在主机上复制前面所述生成的驱动程序中的所有文件，然后将其保存到在目标计算机上创建的同一文件夹中。

在目标计算机上，选择并按住或右键单击证书文件，选择“安装”，然后按提示安装测试证书。

如果需要更详细的目标计算机设置说明，请参阅为手动部署驱动程序准备计算机。

以下说明将介绍如何安装和测试示例驱动程序。 以下是用于安装驱动程序的 devcon 工具的一般语法：

控制台

```
devcon install <INF file> <hardware ID>
```

安装此驱动程序需要的 INF 文件是 *echo.inf*。 inf 文件包含用于安装 *echo.sys* 的硬件 ID。 对于 echo 示例，硬件 ID 为 *root\ECHO*。

在目标计算机上，以管理员身份打开“命令提示符”窗口。 转到你的驱动程序包文件夹，然后输入以下命令：

控制台

```
devcon install echo.inf root\ECHO
```

如果收到一条关于 devcon 未被识别的错误消息，请尝试添加 devcon 工具的路径。 例如，如果将其复制到名为 *C:\Tools* 的文件夹中，请尝试使用以下命令：

控制台

```
c:\tools\devcon install echo.inf root\ECHO
```

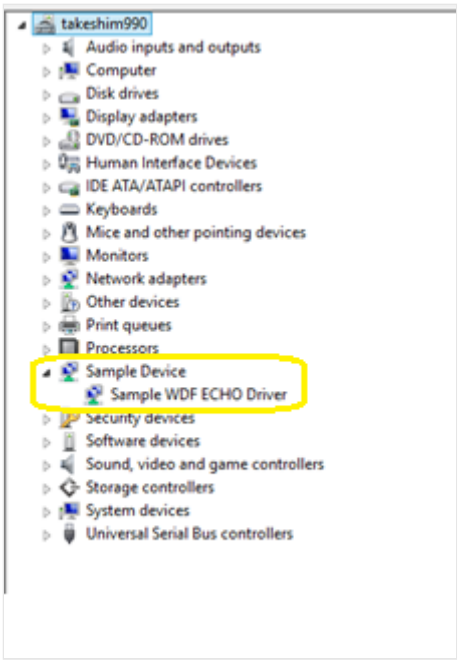
此时会出现一个对话框，指明测试驱动程序是未签名的驱动程序。 选择“仍然安装此驱动程序”以继续。

提示

如果在安装过程中遇到任何问题，请查看以下文件了解更多信息。
%windir%\inf\setupapi.dev.log

成功安装示例驱动程序后，即可对其进行测试。

在目标计算机的“命令提示符”窗口中，输入 *devmgmt* 以打开“设备管理器”。 在设备管理器中，在“查看”菜单中选择“按类型列出设备”。在设备树中，找到“示例设备”节点中的“示例 WDF Echo 驱动程序”。



输入 *echoapp* 以启动测试 echo 应用程序，从而确认驱动程序是否正常。

dbgcmd

```
C:\Samples\KMDF_Echo_Sample> echoapp
DevicePath: \\?\root#sample#0005#{cdc35b6e-0be4-4936-bf5f-5537380a7c1a}
Opened device successfully
512 Pattern Bytes Written successfully
512 Pattern Bytes Read successfully
Pattern Verified successfully
30720 Pattern Bytes Written successfully
30720 Pattern Bytes Read successfully
Pattern Verified successfully
```

使用 WinDbg 显示有关驱动程序的信息

本部分将设置符号路径，并使用内核调试器命令来显示 KMDF echo 示例驱动程序的相关信息。

要查看有关驱动程序的信息，请执行以下操作：

1. 在主机系统上，如果关闭了调试器，请在管理员命令提示符窗口中使用以下命令将其再次打开。

   dbgcmd

   ```
   WinDbg -k
   net:port=50000,key=2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz8
   74p
   ```

2. 使用 Ctrl+Break（滚动锁定）来中断目标系统上运行的代码。

3. 要在 WinDbg 环境中设置 Microsoft 符号服务器的符号路径，请使用 `.symfix` 命令。

   dbgcmd

   ```
   0: kd> .symfix
   ```

4. 要添加本地符号位置以使用本地符号，请依次使用 `.sympath+` 和 `.reload /f` 来添加路径。

   dbgcmd

   ```
   0: kd> .sympath+ C:\DriverSamples\general\echo\kmdf
   0: kd> .reload /f
   ```

   带有 `/f` 的 `.reload` 命令会删除指定模块的所有符号信息并重新加载符号。 在某些情况下，此命令还会重新加载或卸载模块本身。

必须加载适当的符号才能使用 WinDbg 提供的高级功能。 如果没有正确配置符号，在尝试使用依赖于符号的功能时，就会收到符号不可用的消息。

dbgcmd

```
0:000> dv
Unable to enumerate locals, HRESULT 0x80004005
Private symbols (symbols.pri) are required for locals.
Type “.hh dbgerr005” for details.
```

使用符号有很多方法。 在许多情况下，可以对计算机进行配置，以便在需要时从 Microsoft 提供的符号服务器访问符号。 此实验室就采用了该方法。 如果环境中的符号位于其他位置，请修改使用该位置的步骤。 有关详细信息，请参阅 Windows 调试器的符号路径。

要执行源代码调试，必须生成二进制文件的校验（调试）版本。 编译器会创建符号文件（*.pdb* 文件）。 这些符号文件会向调试器显示二进制指令与源代码行的对应关系。 实际源文件本身也必须要能被调试器访问。

符号文件不包含源代码文本。 为了便于调试，链接器最好不要优化代码。 如果代码经过了优化，源代码调试和访问局部变量就会变得更加困难，有时甚至几乎不可能。 如果在查看局部变量或源代码行时遇到问题，请设置以下生成选项：

控制台

```
set COMPILE_DEBUG=1
set ENABLE_OPTIMIZER=0
```

1. 在调试器的命令区输入以下命令，以便显示有关 echo 驱动程序的信息：

   dbgcmd

   ```
   0: kd> lm m echo* v
   Browse full module list
   start             end                 module name
   fffff801`4ae80000 fffff801`4ae89000   ECHO       (private pdb symbols)
   C:\Samples\KMDF_ECHO_SAMPLE\echo.pdb
       Loaded symbol image file: ECHO.sys
       Image path: \SystemRoot\system32\DRIVERS\ECHO.sys
       Image name: ECHO.sys
   ...
   ```

   有关详细信息，请参阅 lm。

2. 由于此实验室之前设置了 `prefer_dml`，因此输出中的某些元素是可供选择的热链接。 在调试输出中选择“浏览所有全局符号”链接，以便显示以字母“a”开头的项目符号信息。

   dbgcmd

   ```
   0: kd> x /D Echo!a*
   ```

3. echo 示例不包含任何以字母“a”开头的符号，因此键入 `x ECHO!Echo*` 可显示与 echo 驱动程序关联的所有以“Echo”开头的符号的信息。

   dbgcmd

   ```
   0: kd> x ECHO!Echo*
   fffff801`0bf95690 ECHO!EchoEvtIoQueueContextDestroy (void *)
   fffff801`0bf95000 ECHO!EchoEvtDeviceSelfManagedIoStart (struct
   WDFDEVICE__ *)
   fffff801`0bf95ac0 ECHO!EchoEvtTimerFunc (struct WDFTIMER__ *)
   fffff801`0bf9b120 ECHO!EchoEvtDeviceSelfManagedIoSuspend (struct
   WDFDEVICE__ *)
   ...
   ```

   有关详细信息，请参阅 x（检查符号）。

4. `!lmi` 扩展显示有关模块的详细信息。 输入 `!lmi echo`。 输出结果应类似于本示例中的文本：

dbgcmd

```
0: kd> !lmi echo
Loaded Module Info: [echo]
         Module: ECHO
   Base Address: fffff8010bf94000
     Image Name: ECHO.sys
…
```

5. 如本示例所示，使用 `!dh` 扩展名来显示标头信息：

dbgcmd

```
0: kd> !dh echo

File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
     14C machine (i386)
       6 number of sections
54AD8A42 time date stamp Wed Jan 07 11:34:26 2015
...
```

6. 输入以下内容以更改默认调试位掩码，从而在调试器中显示来自目标系统的所有调试信息：

dbgcmd

```
0: kd> ed nt!Kd_DEFAULT_MASK 0xFFFFFFFF
```

当使用 0xFFFFFFFF 的掩码时，某些驱动程序会显示额外的信息。 如果想减少显示的信息量，请将掩码设置为 0x00000000。

dbgcmd

```
0: kd> ed nt!Kd_DEFAULT_MASK 0x00000000
```

使用 `dd` 命令确认掩码已被设置为显示所有调试器消息。

dbgcmd

```
0: kd> dd nt!kd_DEFAULT_MASK
fffff802`bb4057c0  ffffffff 00000000 00000000 00000000
fffff802`bb4057d0  00000000 00000000 00000000 00000000
fffff802`bb4057e0  00000001 00000000 00000000 00000000
fffff802`bb4057f0  00000000 00000000 00000000 00000000
fffff802`bb405800  00000000 00000000 00000000 00000000
fffff802`bb405810  00000000 00000000 00000000 00000000
```

```
fffff802`bb405820  00000000 00000000 00000000 00000000
fffff802`bb405830  00000000 00000000 00000000 00000000
```

显示即插即用设备树信息

在本部分中，显示有关 echo 示例设备驱动程序的信息以及它在即插即用设备树中的位置。

即插即用设备树中有关设备驱动程序的信息对故障排除很有用。 例如，如果设备驱动程序没有驻留在设备树中，则表明设备驱动程序的安装可能存在问题。

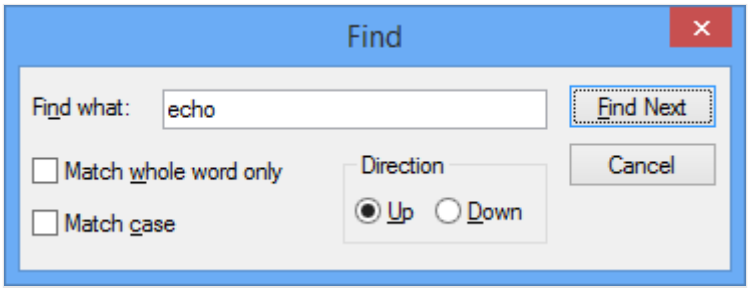
有关设备节点调试扩展的详细信息，请参阅 !devnode。

1. 在主机系统中，要查看即插即用设备树中的所有设备节点，请输入 `!devnode 0 1` 命令。

   dbgcmd

   ```
   0: kd> !devnode 0 1
   Dumping IopRootDeviceNode (= 0xffffe0005a3a8d30)
   DevNode 0xffffe0005a3a8d30 for PDO 0xffffe0005a3a9e50
     InstancePath is "HTREE\ROOT\0"
     State = DeviceNodeStarted (0x308)
     Previous State = DeviceNodeEnumerateCompletion (0x30d)
     DevNode 0xffffe0005a3a3d30 for PDO 0xffffe0005a3a4e50
       InstancePath is "ROOT\volmgr\0000"
       ServiceName is "volmgr"
       State = DeviceNodeStarted (0x308)
       Previous State = DeviceNodeEnumerateCompletion (0x30d)
       DevNode 0xffffe0005a324560 for PDO 0xffffe0005bd95ca0…
   …
   ```

2. 使用 Ctrl+F 在生成的输出中查找设备驱动程序的名称 *echo*。

3. 应加载 echo 设备驱动程序。 使用 `!devnode 0 1 echo` 命令显示与 echo 设备驱动程序关联的即插即用信息，如下例所示：

   dbgcmd

```
0: Kd> !devnode 0 1 echo
Dumping IopRootDeviceNode (= 0xffffe0007b725d30)
DevNode 0xffffe0007b71a630 for PDO 0xffffe0007b71a960
  InstancePath is "ROOT\SAMPLE\0000"
  ServiceName is "ECHO"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
…
```

4. 上一条命令显示的输出包括与运行中的驱动程序实例相关联的 PDO，在本例中为 *0xffffe0007b71a960*。 输入 `!devobj <PDO address>` 命令可显示与 echo 设备驱动程序关联的即插即用信息。 使用 `!devnode` 在计算机上显示的 PDO 地址，而不是此处显示的地址。

   dbgcmd

   ```
   0: kd> !devobj 0xffffe0007b71a960
   Device object (ffffe0007b71a960) is for:
    0000000e \Driver\PnpManager DriverObject ffffe0007b727e60
   Current Irp 00000000 RefCount 0 Type 00000004 Flags 00001040
   Dacl ffffc102c9b36031 DevExt 00000000 DevObjExt ffffe0007b71aab0
   DevNode ffffe0007b71a630
   ExtensionFlags (0x00000800)  DOE_DEFAULT_SD_PRESENT
   Characteristics (0x00000180)  FILE_AUTOGENERATED_DEVICE_NAME,
   FILE_DEVICE_SECURE_OPEN
   AttachedDevice (Upper) ffffe000801fee20 \Driver\ECHO
   Device queue is not busy.
   ```

5. `!devnode 0 1` 命令显示的输出包括与运行中的驱动程序实例相关联的 PDO 地址，在本例中为 *0xffffe0007b71a960*。 输入 `!devstack <PDO address>` 命令可显示与设备驱动程序关联的即插即用信息。 使用 `!devnode` 在计算机上显示的 PDO 地址，而不是本例中显示的地址。

   dbgcmd

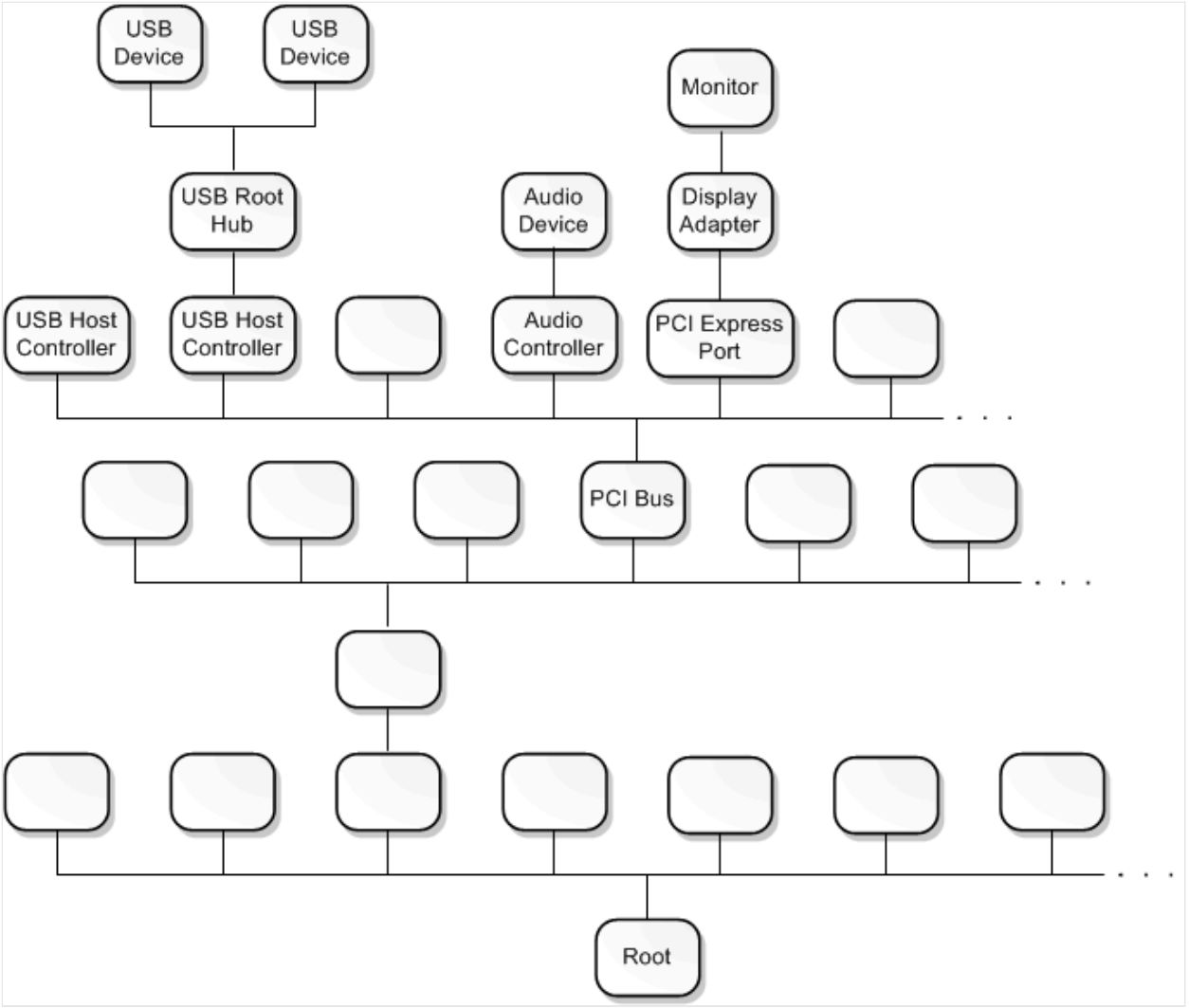
   ```
   0: kd> !devstack 0xffffe0007b71a960
     !DevObj           !DrvObj            !DevExt           ObjectName
     ffffe000801fee20  \Driver\ECHO       ffffe0007f72eff0
   > ffffe0007b71a960  \Driver\PnpManager 00000000  0000000e
   !DevNode ffffe0007b71a630 :
     DeviceInst is "ROOT\SAMPLE\0000"
     ServiceName is "ECHO"
   ```

输出显示有一个非常简单的设备驱动程序堆栈。 echo 驱动程序是 *PnPManager* 节点的子节点。 *PnPManager* 是根节点。

控制台

```
\Driver\ECHO
\Driver\PnpManager
```

此图显示了一个更为复杂的设备节点树。



注意 有关更复杂的驱动程序堆栈的详细信息，请参阅驱动程序堆栈和设备节点和设备堆栈。

使用断点和源代码

在本部分中，设置断点并单步执行内核模式源代码。

要能够逐步执行代码并实时检查变量的值，请启用断点并设置源代码路径。

断点可在特定代码行停止代码执行。 从这一点开始向前执行代码，对特定的代码部分进行调试。

要使用调试命令来设置断点，请使用以下 `b` 命令之一。

展开表

命令	说明
`bp`	设置一个将一直处于活动状态的断点，直到其所在模块被卸载。
`bu`	设置一个断点，该断点在卸载模块时未解析，并在重新加载模块时重新启用。
`bm`	为符号设置一个断点。 此命令将适当使用 `bu` 或 `bp`，并允许使用通配符 (`*`) 在每个匹配的符号（如类中的所有方法）上设置断点。

有关详细信息，请参阅 WinDbg 中的源代码调试。

1. 在主机系统上，使用 WinDbg UI 确认**调试**>**源模式**是否已在当前 WinDbg 会话中启用。

2. 输入以下命令，将本地代码位置添加到源路径中：

   ```
   dbgcmd

   .srcpath+ C:\DriverSamples\KMDF_Echo_Sample\driver\AutoSync
   ```

3. 输入以下命令，将本地符号位置添加到符号路径中：

   ```
   dbgcmd

   .sympath+ C:\DriverSamples\KMDF_Echo_Sample\driver\AutoSync
   ```

4. 使用 `x` 命令检查与 echo 驱动程序关联的符号，以确定用于断点的函数名称。 可以使用通配符或 Ctrl+F 来查找 `DeviceAdd` 函数名称。

   ```
   dbgcmd

   0: kd> x ECHO!EchoEvt*
   8b4c7490          ECHO!EchoEvtIoQueueContextDestroy (void *)
   8b4c7000          ECHO!EchoEvtDeviceSelfManagedIoStart (struct
   WDFDEVICE__ *)
   8b4c7820          ECHO!EchoEvtTimerFunc (struct WDFTIMER__ *)
   8b4cb0e0          ECHO!EchoEvtDeviceSelfManagedIoSuspend (struct
   WDFDEVICE__ *)
   8b4c75d0          ECHO!EchoEvtIoWrite (struct WDFQUEUE__ *, struct
   WDFREQUEST__ *, unsigned int)
   8b4cb170          ECHO!EchoEvtDeviceAdd (struct WDFDRIVER__ *, struct
   …
   ```

   输出显示，echo 驱动程序的 `DeviceAdd` 方法是 `ECHO!EchoEvtDeviceAdd`。

或者，查看源代码以查找断点的函数名称。

5. 使用驱动程序名称后跟要在其中设置断点的函数名称（例如 `AddDevice`）并以感叹号分隔，通过 `bm` 命令来设置断点。 此实验室使用 `AddDevice` 来监视驱动程序的加载情况。

   dbgcmd

   ```
   0: kd> bm ECHO!EchoEvtDeviceAdd
     1: fffff801`0bf9b1c0 @!"ECHO!EchoEvtDeviceAdd"
   ```

   可以将不同的语法与设置变量一起使用，例如 `<module>!<symbol>`、`<class>::<method>`、`'<file.cpp>:<line number>'` 或多次跳过 `<condition> <#>`。 有关详细信息，请参阅 WinDbg 和其他 Windows 调试器中的条件断点。

6. 列出当前断点，确认通过输入 `bl` 命令设置了断点：

   dbgcmd

   ```
   0: kd> bl
   1 e fffff801`0bf9b1c0     0001 (0001) ECHO!EchoEvtDeviceAdd
   ```

   此处显示的输出中的“e”表示启用断点编号 1 以进行触发。

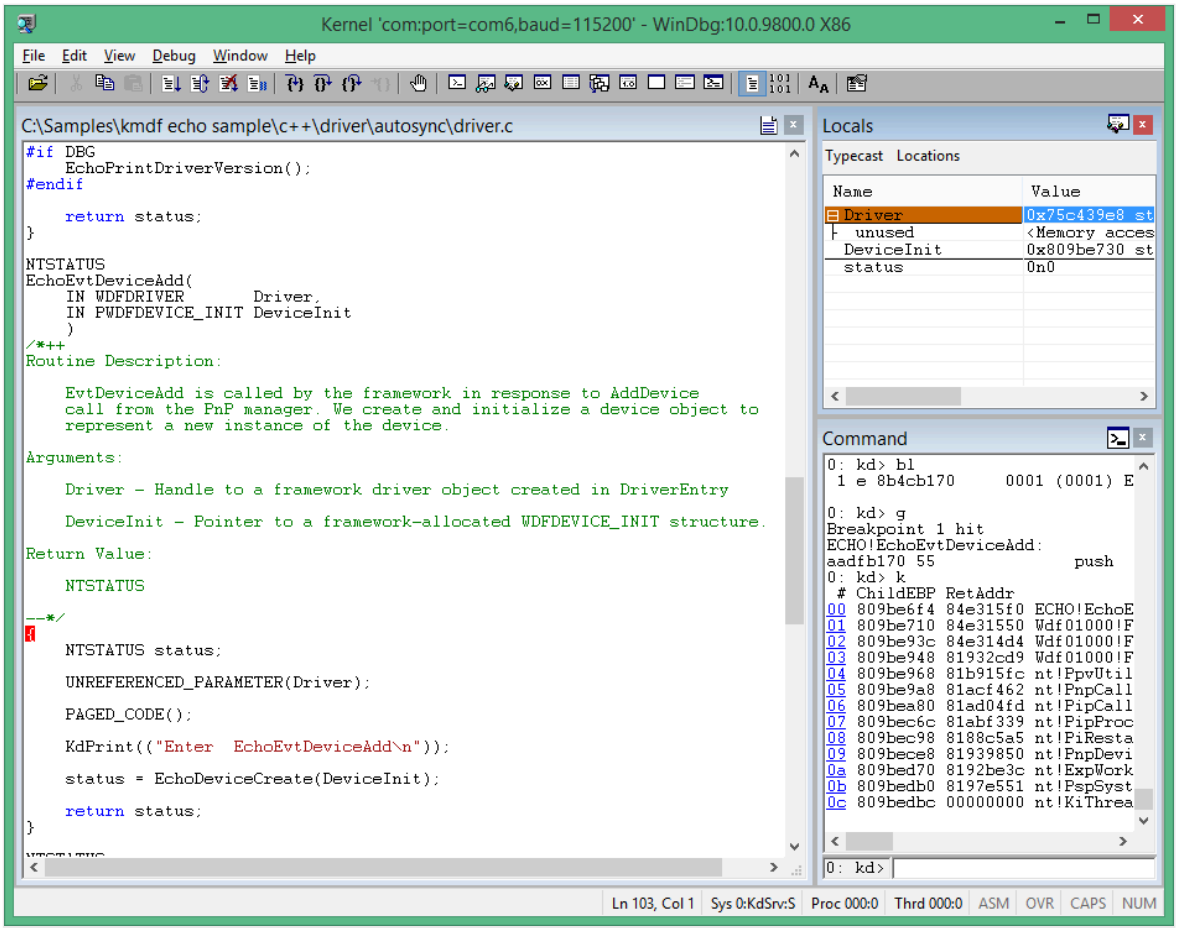
7. 输入 `g` (go) 命令在目标系统上重新启动代码执行。

8. 在目标系统的 Windows 中，使用图标或输入 **mmc devmgmt.msc** 打开“设备管理器”。 在“设备管理器”中，展开*示例*节点。

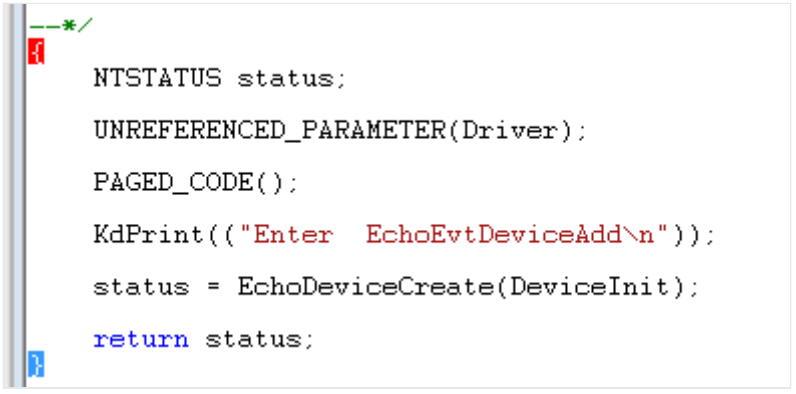
9. 选择并按住或右键单击 KMDF echo 驱动程序条目，然后从菜单中选择“禁用”。

10. 再次选择并按住或右键单击 KMDF echo 驱动程序条目，然后从菜单中选择“启用”。

11. 在主机系统上，启用驱动程序后，AddDevice 调试断点应触发。 目标系统上的驱动程序代码应停止执行。 当触发断点时，应在 *AddDevice* 例程开始时停止执行。 调试命令输出显示 `Breakpoint 1 hit`。



12. 输入 `p` 命令或按 F10 键，逐行执行代码行，直到 AddDevice 例程结束。 如下所示，突出显示大括号字符 (`}`)。



在下一部分中，将检查 DeviceAdd 代码执行后变量的状态。

可以使用以下命令修改现有的断点：

展开表

命令	说明
`bl`	列出断点。
`bc`	清除列表中的断点。 使用 `bc *` 清除所有断点。

命令	说明
`bd`	禁用断点。 使用 `bd *` 禁用所有断点。
`be`	启用断点。 使用 `be *` 启用所有断点。

另外，也可以在 WinDbg UI 中修改断点。

还可以设置在访问内存位置时触发的断点。 使用以下语法的 `ba`（中断访问）命令：

```
dbgcmd
```

```
ba <access> <size> <address> {options}
```

展开表

选项	说明
`e`	execute：当 CPU 从地址中提取指令时
`r`	read/write：当 CPU 读取或写入地址时
`w`	write：当 CPU 写入地址时

在任何给定时间只能设置四个数据断点。 需要确保正确对齐数据方可触发断点。 Word 必须以可被 2 整除的地址结尾，dword 必须可被 4 整除，quadword 必须可被 0 或 8 整除。

例如，要在特定内存地址上设置读/写断点，可以使用类似此示例的命令。

```
dbgcmd
```

```
ba r 4 0x0003f7bf0
```

可以使用以下命令单步执行代码，括号中显示的是相关的键盘快捷键。

- 中断 (Ctrl+Break)。 只要系统正在运行并与 WinDbg 通信，此命令就会中断系统。 内核调试器中的序列为 Ctrl+C。
- 运行到光标处（F7 或 Ctrl+F10）。 将光标放在要中断执行的源代码或反汇编窗口中，然后按 F7。 代码执行将运行到该点。 如果代码执行流没有到达光标指示的点，WinDbg 就不会中断。 如果未执行 IF 语句，就会出现这种情况。
- 运行 (F5)。 运行，直到遇到断点或发生错误检查等事件。
- 单步跳过 (F10)。 该命令可让代码一次只执行一条语句或指令。 如果遇到调用，则代码执行将越过调用，而不会进入被调用例程。 如果编程语言为 C 或 C++，且 WinDbg 处于源模式，则可以使用**调试**>**源模式**来打开或关闭源模式。

- 单步执行 (F11)。 该命令与单步跳过类似，只是调用的执行会进入被调用例程。
- 单步跳出 (Shift+F11)。 该命令可让代码执行到当前例程（调用栈中的当前位置）并退出。 如果已查看足够多的例程，此命令将非常有用。

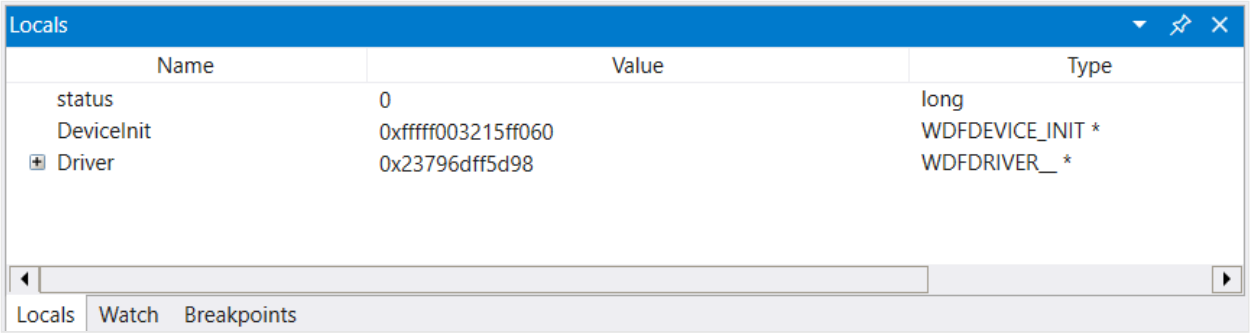
有关详细信息，请参阅 WinDbg 中的源代码调试。

查看变量和调用堆栈

在本部分中，显示有关变量和调用堆栈的信息。

此实验室假定使用前面所述的过程停止了 AddDevice 例程。 要查看此处显示的输出，请重复前面所述的步骤（如有必要）。

在主机系统上，要显示变量，可使用**查看**>**局部**菜单项来显示局部变量。



要查找全局变量地址的位置，请输入 `? <variable name>`。

- 单步跳出 (Shift+F11) – 该命令可让代码执行到当前例程（调用栈中的当前位置）并退出。 如果已查看足够多的例程，这将非常有用。

有关详细信息，请参阅调试参考文档中的 WinDbg 中的源代码调试（经典版）。

第 8 部分：查看变量和调用堆栈

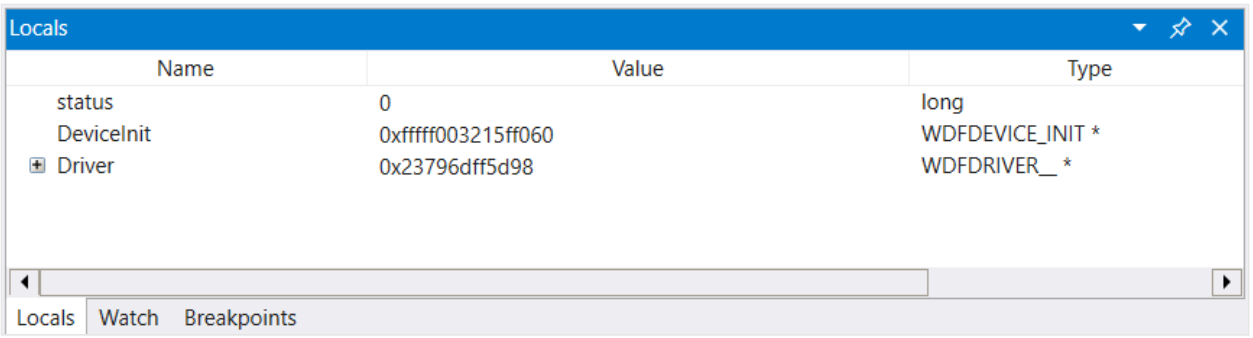
在第 8 部分中，将显示有关变量和调用堆栈的信息。

此实验室假定使用前面所述的过程停止了 *AddDevice* 例程。 要查看此处显示的输出，请重复前面所述的步骤（如有必要）。

<- 在主机系统上

显示变量

使用**查看**>**局部**菜单项显示局部变量。



全局变量

可以通过键入 *? <variable name>* 来查找全局变量地址的位置。

局部变量

输入 **dv** 命令可以显示指定帧的所有局部变量的名称和值。 要显示特定帧的所有局部变量的名称和值，请输入 `dv` 命令：

dbgcmd

```
0: kd> dv
          Driver = 0x00001fff`7ff9c838
      DeviceInit = 0xffffd001`51978190
          status = 0n0
```

调用堆栈是指向程序计数器当前位置的函数调用链。 调用堆栈上最上面的函数是当前函数，而下一个函数是调用当前函数的函数，依此类推。

要显示调用堆栈，请使用 `k*` 命令。

展开表

命令	说明
`kb`	显示堆栈和前三个参数。
`kp`	显示堆栈和完整的参数列表。
`kn`	允许查看堆栈和旁边的帧信息。

1. 在主机系统上，如果要保留可用的调用堆栈，请选择**查看**>**调用堆栈**进行查看。 选择窗口顶部的列可切换显示其他信息。

Stack

Frame Index	Call Site	Child-SP	Return Address
[0x0]	**ECHO!EchoEvtDeviceAdd+0x67**	**0xfffff003215fefe0**	**0xfffff8047d822155**
[0x1]	Wdf01000!FxDriverDeviceAdd::Invoke+0x45	0xfffff003215ff020	0xfffff8047d82207c
[0x2]	Wdf01000!FxDriver::AddDevice+0xc5	0xfffff003215ff020	0xfffff8047d82207c
[0x3]	Wdf01000!FxDriver::AddDevice+0x2c	0xfffff003215ff440	0xfffff8047a584437
[0x4]	nt!PpvUtilCallAddDevice+0x3b	0xfffff003215ff470	0xfffff8047a9f80d4
[0x5]	nt!PnpCallAddDevice+0x9c	0xfffff003215ff4b0	0xfffff8047a97e4c7
[0x6]	nt!PipCallDriverAddDevice+0x853	0xfffff003215ff570	0xfffff8047a97b43d
[0x7]	nt!PipProcessDevNodeTree+0x1b9	0xfffff003215ff730	0xfffff8047aa06e06
[0x8]	nt!PiRestartDevice+0xba	0xfffff003215ff7f0	0xfffff8047a4f3748
[0x9]	nt!PnpDeviceActionWorker+0x628	0xfffff003215ff840	0xfffff8047a4188a5
[0xa]	nt!ExpWorkerThread+0x155	0xfffff003215ff900	0xfffff8047a40f4a7
[0xb]	nt!PspSystemThreadStartup+0x57	0xfffff003215ffaf0	0xfffff8047a62de84
[0xc]	nt!KiStartSystemThread+0x34	0xfffff003215ffb40	0x0

2. 当调试处于中断状态的示例适配器代码时，使用 `kn` 命令显示调用堆栈。

dbgcmd

```
3: kd> kn
# Child-SP          RetAddr           Call Site
00 ffffd001`51978110 fffff801`0942f55b ECHO!EchoEvtDeviceAdd+0x66
[c:\Samples\kmdf echo sample\c++\driver\autosync\driver.c @ 138]
01 (Inline Function) --------`--------
Wdf01000!FxDriverDeviceAdd::Invoke+0x30
[d:\wbrtm\minkernel\wdf\framework\shared\inc\private\common\fxdrivercal
lbacks.hpp @ 61]
02 ffffd001`51978150 fffff801`eed8097d
Wdf01000!FxDriver::AddDevice+0xab
[d:\wbrtm\minkernel\wdf\framework\shared\core\km\fxdriverkm.cpp @ 72]
03 ffffd001`51978570 fffff801`ef129423 nt!PpvUtilCallAddDevice+0x35
[d:\9142\minkernel\ntos\io\pnpmgr\verifier.c @ 104]
04 ffffd001`519785b0 fffff801`ef0c4112 nt!PnpCallAddDevice+0x63
[d:\9142\minkernel\ntos\io\pnpmgr\enum.c @ 7397]
05 ffffd001`51978630 fffff801`ef0c344f nt!PipCallDriverAddDevice+0x6e2
[d:\9142\minkernel\ntos\io\pnpmgr\enum.c @ 3390]
...
```

调用堆栈显示，内核 (nt) 调用了即插即用代码 (PnP)，PnP 调用了驱动程序框架代码 (WDF)，WDF 随后又调用了 echo 驱动程序 `DeviceAdd` 函数。

显示进程和线程

本部分显示内核模式下运行的进程和线程信息。

进程

可以使用 !process 调试器扩展名来显示或设置进程信息。 设置一个断点，以检查播放声音时使用的进程。

1. 在主机系统上输入 `dv` 命令，以检查与 `EchoEvtIo` 例程关联的局部变量：

```
dbgcmd

0: kd> dv ECHO!EchoEvtIo*
ECHO!EchoEvtIoQueueContextDestroy
ECHO!EchoEvtIoWrite
ECHO!EchoEvtIoRead
```

2. 使用 `bc *` 清除之前的断点：

```
dbgcmd

0: kd> bc *
```

3. 使用以下命令在 `EchoEvtIo` 例程上设置符号断点：

```
dbgcmd

0: kd> bm ECHO!EchoEvtIo*
  2: aade5490          @!"ECHO!EchoEvtIoQueueContextDestroy"
  3: aade55d0          @!"ECHO!EchoEvtIoWrite"
  4: aade54c0          @!"ECHO!EchoEvtIoRead"
```

4. 列出断点，以便确认断点已正确设置：

```
dbgcmd

0: kd> bl
1 e aabf0490 [c:\Samples\kmdf echo sample\c++\driver\autosync\queue.c @
197]    0001 (0001) ECHO!EchoEvtIoQueueContextDestroy
...
```

5. 输入 `g` 以重新启动代码执行：

```
dbgcmd

0: kd> g
```

6. 在目标系统上运行 `EchoApp.exe` 驱动程序测试程序。

7. 在主机系统中，当测试程序运行时，会调用驱动程序中的 I/O 例程。 该调用会导致断点触发，而目标系统上的驱动程序代码将停止执行。

```
dbgcmd
```

```
Breakpoint 2 hit
ECHO!EchoEvtIoWrite:
fffff801`0bf95810 4c89442418      mov     qword ptr [rsp+18h],r8
```

8. 使用 `!process` 命令显示运行 *echoapp.exe*所涉及的当前进程：

```
dbgcmd
```

```
0: kd> !process
PROCESS ffffe0007e6a7780
    SessionId: 1  Cid: 03c4    Peb: 7ff7cfec4000  ParentCid: 0f34
    DirBase: 1efd1b000  ObjectTable: ffffc001d77978c0  HandleCount:
34.
    Image: echoapp.exe
    VadRoot ffffe000802c79f0 Vads 30 Clone 0 Private 135. Modified 5.
Locked 0.
    DeviceMap ffffc001d83c6e80
    Token                             ffffc001cf270050
    ElapsedTime                       00:00:00.052
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         33824
    QuotaPoolUsage[NonPagedPool]      4464
    Working Set Sizes (now,min,max)  (682, 50, 345) (2728KB, 200KB,
1380KB)
    PeakWorkingSetSize                652
    VirtualSize                       16 Mb
    PeakVirtualSize                   16 Mb
    PageFaultCount                    688
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      138

        THREAD ffffe00080e32080  Cid 03c4.0ec0  Teb: 00007ff7cfece000
Win32Thread: 0000000000000000 RUNNING on processor 1
```

输出显示该进程与 *echoapp.exe* 线程相关联，当驱动程序写入事件断点被触发时，该线程正在运行。 有关详细信息，请参阅 !process。

9. 使用 `!process 0 0` 显示所有进程的摘要信息。 在输出中，使用 Ctrl+F 查找与 *echoapp.exe* 映像关联的进程的相同进程地址。 在此示例中，进程地址为 `ffffe0007e6a7780`.

```
dbgcmd
```

```
...

PROCESS ffffe0007e6a7780
    SessionId: 1  Cid: 0f68    Peb: 7ff7cfe7a000  ParentCid: 0f34
    DirBase: 1f7fb9000  ObjectTable: ffffc001cec82780  HandleCount:
34.
    Image: echoapp.exe

...
```

10. 记录与 *echoapp.exe* 关联的进程 ID，以便以后在本实验室中使用。 还可以使用 Ctrl+C 将地址复制到复制缓冲区，以供今后使用。

    _____（echoapp.exe 进程地址）

11. 根据需要在调试器中输入 `g` 以便向前运行代码，直到 *echoapp.exe* 运行完毕。 它在读写事件中多次触发断点。 在 *echoapp.exe*完成后，通过按 Ctrl+ScrLk (Ctrl+Break) 切换到调试器。

12. 使用 `!process` 命令确认运行的是另一个进程。 在此处显示的输出中，映像值为 *System* 的流程与 *Echo* 的映像值不同。

    dbgcmd

    ```
    1: kd> !process
    PROCESS ffffe0007b65d900
        SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
        DirBase: 001ab000  ObjectTable: ffffc001c9a03000  HandleCount: 786.
        Image: System
        VadRoot ffffe0007ce45930 Vads 14 Clone 0 Private 22. Modified
    131605. Locked 64.
        DeviceMap ffffc001c9a0c220
        Token                             ffffc001c9a05530
        ElapsedTime                       21:31:02.516
    ...
    ```

    输出显示当停止操作系统时，系统进程 ffffe0007b65d900 正在运行。

13. 使用 `!process` 命令尝试查看与之前记录的 *echoapp.exe* 相关联的进程 ID。 提供之前记录的 *echoapp.exe* 进程地址，而不是本例中显示的示例进程地址。

    dbgcmd

    ```
    0: kd> !process ffffe0007e6a7780
    TYPE mismatch for process object at 82a9acc0
    ```

    进程对象已不可用，因为 *echoapp.exe* 进程已停止运行。

线程

用于查看和设置线程的命令与进程的命令类似。 使用 !thread 命令来查看线程。 使用 .thread 来设置当前线程。

1. 在主机系统上，在调试器中输入 `g` 以重新启动目标系统上的代码执行。

2. 在目标系统上，运行 EchoApp.exe 驱动程序测试程序。

3. 在主机系统上，中断点被命中，代码执行将停止。

```
dbgcmd

Breakpoint 4 hit
ECHO!EchoEvtIoRead:
aade54c0 55              push    ebp
```

4. 若要查看正在运行的线程，请输入 !thread。 应显示类似下例的信息：

```
dbgcmd

0: kd>  !thread
THREAD ffffe000809a0880  Cid 0b28.1158  Teb: 00007ff7d00dd000
Win32Thread: 0000000000000000 RUNNING on processor 0
IRP List:
    ffffe0007bc5be10: (0006,01f0) Flags: 00060a30  Mdl: 00000000
Not impersonating
DeviceMap                 ffffc001d83c6e80
Owning Process            ffffe0008096c900       Image:
echoapp.exe
...
```

   注意 *echoapp.exe* 的图像名称。 这表示您正在查看与测试应用关联的线程。

5. 使用`!process`命令确定此线程是否是与 *echoapp.exe* 关联的进程中运行的唯一线程。 进程中正在运行的线程的线程数与`!thread` 命令显示的线程数相同。

```
dbgcmd

0: kd> !process
PROCESS ffffe0008096c900
    SessionId: 1  Cid: 0b28    Peb: 7ff7d00df000  ParentCid: 0f34
    DirBase: 1fb746000  ObjectTable: ffffc001db6b52c0  HandleCount:
34.
    Image: echoapp.exe
    VadRoot ffffe000800cf920 Vads 30 Clone 0 Private 135. Modified 8.
Locked 0.
    DeviceMap ffffc001d83c6e80
    Token                             ffffc001cf5dc050
```

```
    ElapsedTime                           00:00:00.048
    UserTime                              00:00:00.000
    KernelTime                            00:00:00.000
    QuotaPoolUsage[PagedPool]             33824
    QuotaPoolUsage[NonPagedPool]          4464
    Working Set Sizes (now,min,max)  (681, 50, 345) (2724KB, 200KB,
1380KB)
    PeakWorkingSetSize                    651
    VirtualSize                           16 Mb
    PeakVirtualSize                       16 Mb
    PageFaultCount                        686
    MemoryPriority                        BACKGROUND
    BasePriority                          8
    CommitCharge                          138

        THREAD ffffe000809a0880  Cid 0b28.1158  Teb: 00007ff7d00dd000
Win32Thread: 0000000000000000 RUNNING on processor 0
```

6. 使用`!process 0 0`命令查找两个相关进程的进程地址，并在此处记录这些进程地址。

   Cmd.exe: _____

   EchoApp.exe: _____

   dbgcmd

   ```
   0: kd> !process 0 0

   …

   PROCESS ffffe0007bbde900
       SessionId: 1  Cid: 0f34    Peb: 7ff72dfa7000  ParentCid: 0c64
       DirBase: 19c5fa000  ObjectTable: ffffc001d8c2f300  HandleCount:
   31.
       Image: cmd.exe
   …
   PROCESS ffffe0008096c900
       SessionId: 1  Cid: 0b28    Peb: 7ff7d00df000  ParentCid: 0f34
       DirBase: 1fb746000  ObjectTable: ffffc001db6b52c0  HandleCount:
   34.
       Image: echoapp.exe
   …
   ```

   您也可使用 `!process 0 17` 显示有关每个进程的详细信息。 此命令的输出可能很长。 可以使用 Ctrl+F 搜索输出。

7. 使用`!process`命令列出运行计算机的两个进程的进程信息。 提供 `!process 0 0` 输出中的进程地址，而不是此示例中显示的地址。

此示例输出适用于前面记录的 *cmd.exe* 进程 ID。 此进程 ID 的映像名称为 *cmd.exe*。

dbgcmd

```
0: kd>  !process ffffe0007bbde900
PROCESS ffffe0007bbde900
    SessionId: 1  Cid: 0f34    Peb: 7ff72dfa7000  ParentCid: 0c64
    DirBase: 19c5fa000  ObjectTable: ffffc001d8c2f300  HandleCount:
31.
    Image: cmd.exe
    VadRoot ffffe0007bb8e7b0 Vads 25 Clone 0 Private 117. Modified 20.
Locked 0.
    DeviceMap ffffc001d83c6e80
    Token                             ffffc001d8c48050
    ElapsedTime                       21:33:05.840
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         24656
    QuotaPoolUsage[NonPagedPool]      3184
    Working Set Sizes (now,min,max)  (261, 50, 345) (1044KB, 200KB,
1380KB)
    PeakWorkingSetSize                616
    VirtualSize                       2097164 Mb
    PeakVirtualSize                   2097165 Mb
    PageFaultCount                    823
    MemoryPriority                    FOREGROUND
    BasePriority                      8
    CommitCharge                      381

        THREAD ffffe0007cf34880  Cid 0f34.0f1c  Teb: 00007ff72dfae000
Win32Thread: 0000000000000000 WAIT: (UserRequest) UserMode Non-
Alertable
            ffffe0008096c900  ProcessObject
        Not impersonating
...
```

此示例输出适用于前面记录的 *echoapp.exe* 进程 ID。

dbgcmd

```
0: kd>  !process ffffe0008096c900
PROCESS ffffe0008096c900
    SessionId: 1  Cid: 0b28    Peb: 7ff7d00df000  ParentCid: 0f34
    DirBase: 1fb746000  ObjectTable: ffffc001db6b52c0  HandleCount:
34.
    Image: echoapp.exe
    VadRoot ffffe000800cf920 Vads 30 Clone 0 Private 135. Modified 8.
Locked 0.
    DeviceMap ffffc001d83c6e80
    Token                             ffffc001cf5dc050
    ElapsedTime                       00:00:00.048
```

```
    UserTime                            00:00:00.000
    KernelTime                          00:00:00.000
    QuotaPoolUsage[PagedPool]           33824
    QuotaPoolUsage[NonPagedPool]        4464
    Working Set Sizes (now,min,max)  (681, 50, 345) (2724KB, 200KB,
1380KB)
    PeakWorkingSetSize                  651
    VirtualSize                         16 Mb
    PeakVirtualSize                     16 Mb
    PageFaultCount                      686
    MemoryPriority                      BACKGROUND
    BasePriority                        8
    CommitCharge                        138

        THREAD ffffe000809a0880  Cid 0b28.1158  Teb: 00007ff7d00dd000
Win32Thread: 0000000000000000 RUNNING on processor 0
        IRP List:
            ffffe0007bc5be10: (0006,01f0) Flags: 00060a30  Mdl:
00000000
        Not impersonating
...
```

8. 在此处记录与两个进程关联的第一个线程地址。

   Cmd.exe: _____

   EchoApp.exe: _____

9. 使用 `!Thread` 命令显示有关当前线程的信息。

   dbgcmd

```
0: kd>  !Thread
THREAD ffffe000809a0880  Cid 0b28.1158  Teb: 00007ff7d00dd000
Win32Thread: 0000000000000000 RUNNING on processor 0
IRP List:
    ffffe0007bc5be10: (0006,01f0) Flags: 00060a30  Mdl: 00000000
Not impersonating
DeviceMap                 ffffc001d83c6e80
Owning Process            ffffe0008096c900       Image:
echoapp.exe
Attached Process          N/A            Image:         N/A
...
```

   如预期所示，当前线程是与 *echoapp.exe* 关联的线程，并且处于运行状态。

10. 使用`!Thread`命令显示与 *cmd.exe* 进程关联的线程的相关信息。 提供前面记录的线程地址。

    dbgcmd

```
0: kd> !Thread ffffe0007cf34880
THREAD ffffe0007cf34880  Cid 0f34.0f1c  Teb: 00007ff72dfae000
Win32Thread: 0000000000000000 WAIT: (UserRequest) UserMode Non-
Alertable
    ffffe0008096c900  ProcessObject
Not impersonating
DeviceMap                 ffffc001d83c6e80
Owning Process            ffffe0007bbde900       Image:         cmd.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      4134621        Ticks: 0
Context Switch Count      4056           IdealProcessor: 0
UserTime                  00:00:00.000
KernelTime                00:00:01.421
Win32 Start Address 0x00007ff72e9d6e20
Stack Init ffffd0015551dc90 Current ffffd0015551d760
Base ffffd0015551e000 Limit ffffd00155518000 Call 0
Priority 14 BasePriority 8 UnusualBoost 3 ForegroundBoost 2 IoPriority
2 PagePriority 5
Child-SP          RetAddr           : Args to Child
: Call Site
ffffd001`5551d7a0 fffff801`eed184fe : fffff801`eef81180
ffffe000`7cf34880 00000000`fffffffe 00000000`fffffffe :
nt!KiSwapContext+0x76 [d:\9142\minkernel\ntos\ke\amd64\ctxswap.asm @
109]
ffffd001`5551d8e0 fffff801`eed17f79 : ffff03a5`ca56a3c8
000000de`b6a6e990 000000de`b6a6e990 00007ff7`d00df000 :
nt!KiSwapThread+0x14e [d:\9142\minkernel\ntos\ke\thredsup.c @ 6347]
ffffd001`5551d980 fffff801`eecea340 : ffffd001`5551da18
00000000`00000000 00000000`00000000 00000000`00000388 :
nt!KiCommitThreadWait+0x129 [d:\9142\minkernel\ntos\ke\waitsup.c @ 619]
...
```

此线程与 *cmd.exe* 关联，并且处于等待状态。

11. 提供等待 *CMD.exe* 线程的线程地址，以将上下文更改为该等待线程。

dbgcmd

```
0: kd> .Thread ffffe0007cf34880
Implicit thread is now ffffe000`7cf34880
```

12. 使用 `k` 命令查看与等待线程关联的调用堆栈。

dbgcmd

```
0: kd> k
  *** Stack trace for last set context - .thread/.cxr resets it
# Child-SP          RetAddr           Call Site
00 ffffd001`5551d7a0 fffff801`eed184fe nt!KiSwapContext+0x76
[d:\9142\minkernel\ntos\ke\amd64\ctxswap.asm @ 109]
```

```
01 ffffd001`5551d8e0 fffff801`eed17f79 nt!KiSwapThread+0x14e
[d:\9142\minkernel\ntos\ke\thredsup.c @ 6347]
02 ffffd001`5551d980 fffff801`eecea340 nt!KiCommitThreadWait+0x129
[d:\9142\minkernel\ntos\ke\waitsup.c @ 619]
03 ffffd001`5551da00 fffff801`ef02e642 nt!KeWaitForSingleObject+0x2c0
[d:\9142\minkernel\ntos\ke\wait.c @ 683]
...
```

调用堆栈元素，例如 `KiCommitThreadWait` 指示此线程未按预期运行。

有关线程和进程的详细信息，请参阅以下参考：

- 线程和进程
- 更改上下文

IRQL、注册和结束 WinDbg 会话

在本部分中，显示中断请求级别 (IRQL) 和寄存器的内容。

查看保存的 IRQL

IRQL 用于管理中断服务的优先级。 每个处理器都有一个 IRQL 设置，线程可以提高或降低该设置。 发生在处理器 IRQL 设置或低于 IRQL 设置的中断将被屏蔽，不会干扰当前操作。 高于处理器 IRQL 设置的中断优先于当前操作。

在主机系统上，!irql 扩展会显示调试器中断前目标计算机当前处理器上的 IRQL。 当目标计算机中断调试器时，IRQL 会发生变化，但调试器中断前有效的 IRQL 会被保存并通过 `!irql` 显示。

dbgcmd

```
0: kd> !irql
Debugger saved IRQL for processor 0x0 -- 2 (DISPATCH_LEVEL)
```

查看寄存器

在主机系统上，使用 r (Registers) 命令显示当前处理器上当前线程的寄存器的内容。

dbgcmd

```
0: kd> r
rax=000000000000c301 rbx=ffffe00173eed880 rcx=0000000000000001
rdx=000000d800000000 rsi=ffffe00173eed8e0 rdi=ffffe00173eed8f0
rip=fffff803bb757020 rsp=ffffd001f01f8988 rbp=ffffe00173f0b620
```

```
 r8=000000000000003e  r9=ffffe00167a4a000 r10=000000000000001e
r11=ffffd001f01f88f8 r12=0000000000000000 r13=ffffd001f01efdc0
r14=0000000000000001 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b
efl=00000202
nt!DbgBreakPointWithStatus:
fffff803`bb757020 cc              int     3
```

另外，也可以通过选择**查看**>**寄存器**来显示寄存器的内容。 有关详细信息，请参阅 r（寄存器）。

在单步执行程序集语言代码和其他情况下，查看寄存器的内容很有帮助。 有关程序集语言反汇编的详细信息，请参阅带批注的 x86 反汇编和带批注的 x64 反汇编。

有关寄存器内容的信息，请参阅 x86 体系结构和 x64 体系结构。

结束 WinDbg 会话

如果想保留调试器，但又想连接到目标计算机，请使用 `bc *` 清除所有断点，这样目标计算机就不会尝试连接到主机调试器。 然后使用 `g` 命令让目标计算机再次运行。

要结束调试会话，请在主机系统上中断调试器，输入 `qd`（退出并分离）命令，或者从菜单中选择**停止调试**。

dbgcmd

```
0: kd> qd
```

有关详细信息，请参阅在 WinDbg 中结束调试会话。

Windows 调试资源

有关 Windows 调试的更多信息。 其中一些书籍在示例中使用 Windows Vista 等较早版本的 Windows，但所讨论的概念适用于大多数版本的 Windows。

- 书籍
  - 《高级 Windows 调试》，作者：Mario Hewardt 和 Daniel Pravat
  - 《在 Windows 调试中：Windows® 调试和跟踪策略实用指南》，作者 Tarik Soulami
  - 《Windows 内部》，作者 Pavel Yosifovich、Alex Ionescu、Mark Russinovich 和 David Solomon
- 视频

- 碎片整理工具显示 WinDbg 集 13-29

- 培训供应商
  - OSR

另请参阅

- 标准调试方法
- 专业调试方法
- Windows 调试入门

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

调试驱动程序 - 分步实验室（Sysvad 内核模式）

项目 • 2024/04/18

此实验室提供动手练习，演示如何调试 Sysvad 音频内核模式设备驱动程序。

Microsoft Windows 调试器 (WinDbg) 是一款功能强大的基于 Windows 的调试工具，可用于执行用户模式和内核模式调试。 WinDbg 可对 Windows 内核、内核模式驱动程序和系统服务以及用户模式应用程序和驱动程序进行源级调试。

WinDbg 可以逐步分析源代码、设置断点、查看变量（包括 C++ 对象）、堆栈跟踪和内存。 其调试器命令窗口允许用户发出各种命令。

实验室设置

要完成实验室设置，需要以下硬件：

- 运行 Windows 10 的笔记本电脑或台式计算机（主机）
- 运行 Windows 10 的笔记本电脑或台式计算机（目标）
- 用于连接两台计算机的网络集线器/路由器以及网络电缆
- 访问 Internet 以下载符号文件

需要以下软件才能完成实验室设置。

- Microsoft Visual Studio 2017
- 适用于 Windows 10 的 Windows 软件开发工具包 (SDK)
- 适用于 Windows 10 的 Windows 驱动程序工具包 (WDK)
- 适用于 Windows 10 的示例 Sysvad 音频驱动程序

有关下载和安装 WDK 的信息，请参阅下载 Windows 驱动程序工具包 (WDK)。

Sysvad 调试演练

本实验室将指导你完成内核模式驱动程序的调试过程。 练习使用 Syvad 虚拟音频驱动程序示例。 由于 Syvad 音频驱动程序不与实际音频硬件交互，因此可用于大多数设备上。 实验室包括以下任务：

- 第 1 部分：连接到内核模式 WinDbg 会话
- 第 2 部分：内核模式调试命令和技术
- 第 3 部分：下载并生成 Sysvad 音频驱动程序
- 第 4 部分：在目标系统上安装 Sysvad 音频驱动程序

- 第 5 部分：使用 WinDbg 显示驱动程序的相关信息
- 第 6 部分：显示即插即用设备树信息
- 第 7 部分：使用断点和源代码
- 第 8 部分：查看变量
- 第 9 部分：查看调用堆栈
- 第 10 部分：显示进程和线程
- 第 11 部分：IRQL、寄存器和反汇编
- 第 12 部分：使用内存
- 第 13 部分：结束 WinDbg 会话
- 第 14 部分：Windows 调试资源

Echo 驱动程序实验室

Echo 驱动程序比 Sysvad 音频驱动程序更为简单。 如果是 WinDbg 的新手，可以考虑先完成调试通用驱动程序 - 分步实验室（Echo 内核模式）。 该实验室重复使用实验室中的设置说明，因此如果已经完成了该实验室，则可以跳过此处的第 1 和第 2 部分。

第 1 部分：连接到内核模式 WinDbg 会话

在第 1 部分中，将在主机和目标系统上配置网络调试。

此实验室中的计算机需要配置为使用以太网网络连接进行内核调试。

此实验室使用两台计算机。 WinDbg 在*主机*系统上运行，而 Sysvad 驱动程序在*目标*系统上运行。

使用网络集线器/路由器和网线连接两台计算机。

要使用内核模式应用程序和 WinDbg，建议使用通过以太网传输的 KDNET。 有关如何使用以太网传输协议的信息，请参阅 WinDbg（内核模式）入门指南。 有关设置目标计算机的详细信息，请参阅为手动部署驱动程序准备计算机以及自动设置 KDNET 网络内核调试。

使用以太网配置内核模式调试

要在目标系统上启用内核模式调试，请执行以下步骤。

<- 在主机系统上

1. 在主机系统上打开命令提示符，键入 **ipconfig /all** 以确定其 IP 地址。

控制台

```
C:\>ipconfig /all
Windows IP Configuration

 Host Name . . . . . . . . . . . . : TARGETPC
...

Ethernet adapter Ethernet:
   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::c8b6:db13:d1e8:b13b3
   Autoconfiguration IPv4 Address. . : 169.182.1.1
   Subnet Mask . . . . . . . . . . . : 255.255.0.0
   Default Gateway . . . . . . . . . :
```

2. 记录下主机系统的 IP 地址：_____

3. 记录下主机系统的主机名：_____

-> 在目标系统上

4. 在目标系统上打开命令提示符，使用 **ping** 命令来确认两个系统之间的网络连接。使用记录下的主机系统的实际 IP 地址，而不是示例输出中显示的 169.182.1.1。

控制台

```
C:\> ping 169.182.1.1

Pinging 169.182.1.1 with 32 bytes of data:
Reply from 169.182.1.1: bytes=32 time=1ms TTL=255
Reply from 169.182.1.1: bytes=32 time<1ms TTL=255
Reply from 169.182.1.1: bytes=32 time<1ms TTL=255
Reply from 169.182.1.1: bytes=32 time<1ms TTL=255

Ping statistics for 169.182.1.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 1ms, Average = 0ms
```

要使用 KDNET 实用工具在目标系统上启用内核模式调试，请执行以下步骤。

1. 在主机系统中，找到 WDK KDNET 目录。 默认情况下，它位于这里。

   C:\Program Files (x86)\Windows Kits\10\Debuggers\x64

此实验室假设目标计算机和主机都运行的 64 位版本的 Windows。 如果情况并非如此，最好的办法是在主电脑上运行与目标电脑相同“位数”的工具。 例如，如果目标计算机运行的是 32 位 Windows，请在主机上运行 32 版本的调试器。 有关详细信息，请参阅选择 32 位或 64 位调试工具。

2. 找到这两个文件，并将它们复制到网络共享或优盘中，以便在目标计算机上使用。

   kdnet.exe

   VerifiedNICList.xml

3. 在目标计算机上，以管理员身份打开“命令提示符”窗口。 输入此命令以验证目标计算机上的 NIC 是否支持。

控制台

```
C:\KDNET>kdnet

Network debugging is supported on the following NICs:
busparams=0.25.0, Intel(R) 82579LM Gigabit Network Connection, KDNET is
running on this NIC.kdnet.exe
```

4. 键入此命令可以设置主机系统的 IP 地址。 使用记录下的主机系统的实际 IP 地址，而不是示例输出中显示的 169.182.1.1。 为每个目标/主机配对选择一个唯一的端口地址，如 50010。

控制台

```
C:\>kdnet 169.182.1.1 50010

Enabling network debugging on Intel(R) 82577LM Gigabit Network Connection.
Key=2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
```

ⓘ 重要

在使用 BCDEdit 更改启动信息之前，可能需要在测试电脑上暂时挂起 Windows 安全功能，例如 BitLocker 和安全启动。 测试完成后重新启用这些安全功能，并在禁用安全功能时适当地管理测试电脑。 UEFI 中通常已禁用安全启动。 要访问 UEFI 设置，请使用系统、恢复、高级启动。 重新启动后，选择“故障排除”、“高级选项”、“UEFI 固件设置”。 请谨慎操作，因为错误设置 UEFI 选项或禁用 BitLocker 可能会导致系统无法运行。

5. 键入此命令以确认已正确设置 dbgsettings。

控制台

```
C:\> bcdedit /dbgsettings
busparams               0.25.0
key
2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
debugtype               NET
hostip                  169.182.1.1
port                    50010
dhcp                    Yes
The operation completed successfully.
```

将自动生成的唯一密钥复制到文本文件中，以避免在主机上输入。 将包含密钥的文本文件复制到主机系统中。

注意防火墙和调试器

如果收到来自防火墙的弹出消息并希望使用调试器，请选中**所有三个**复选框。

<- 在主机系统上

1. 在主计算机上，以管理员身份打开命令提示符窗口。 转到 WinDbg.exe 目录。 我们将使用安装 Windows 工具包过程中安装的 Windows 驱动程序工具包 (WDK) 中的 x64 版本 WinDbg.exe。

控制台

```
C:\> Cd C:\Program Files (x86)\Windows Kits\10\Debuggers\x64
```

2. 使用以下命令通过远程用户调试来启动 WinDbg。 密钥和端口的值与之前在目标计算机上使用 BCDEdit 设置的值一致。

控制台

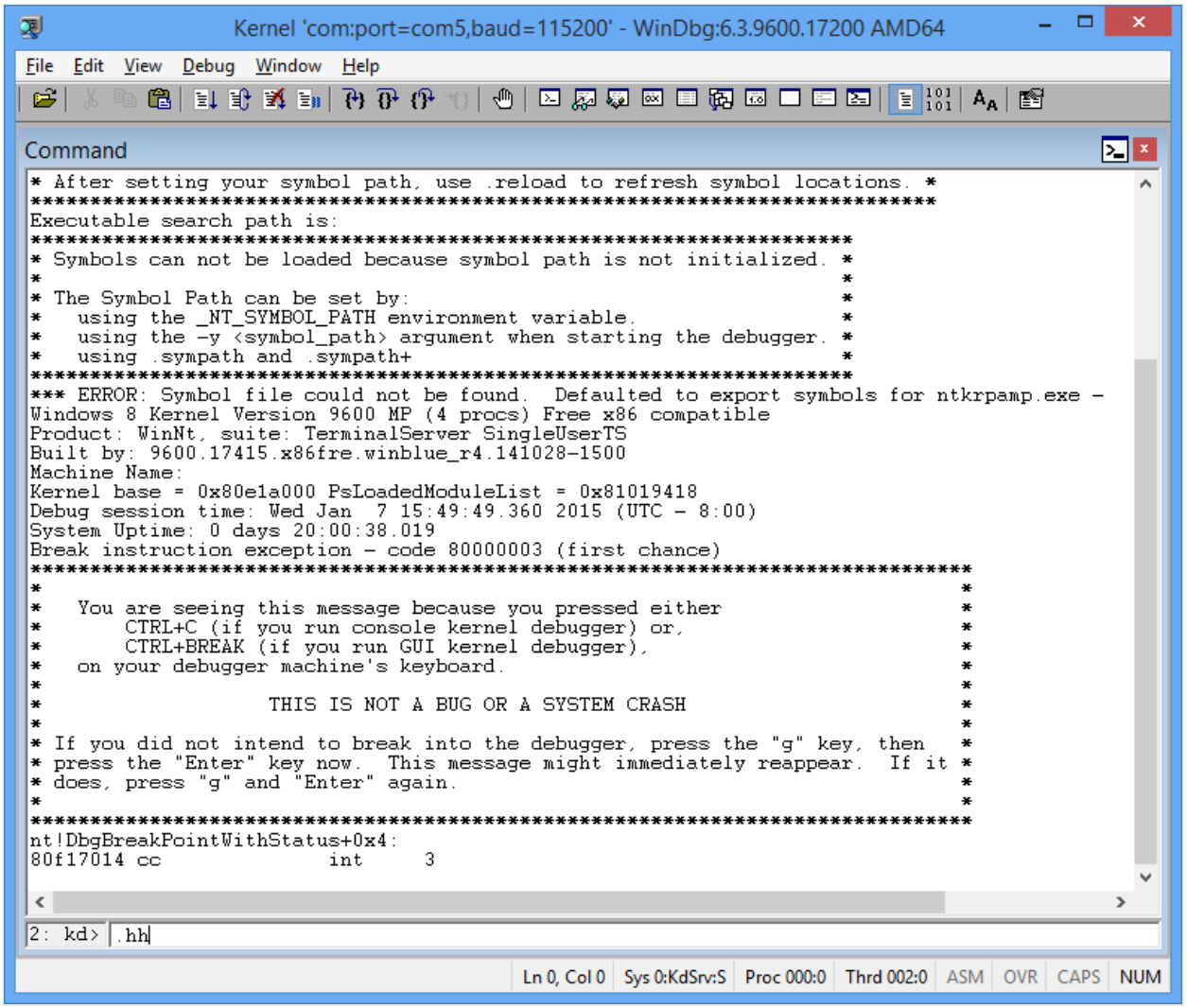
```
C:\> WinDbg -k
net:port=50010,key=2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
```

-> **在目标系统上**

重启目标系统。

<- **在主机系统上**

一两分钟后，主机系统上就会显示调试输出。



“调试器命令”窗口是 WinDbg 的主要调试信息窗口。 可以在此窗口中输入调试器命令并查看命令输出。

调试器命令窗口被分为两个窗格。 可以在窗口底部较小的窗格（命令输入窗格）中输入命令，并在窗口顶部较大的窗格中查看命令输出。

在命令输入窗格中，使用上箭头和下箭头键可以滚动浏览命令历史记录。 命令出现后，可对其进行编辑，或者按 **ENTER** 运行命令。

第 2 部分：内核模式调试命令和技术

在第 2 部分中，将使用调试命令来显示目标系统的相关信息。

<- 在主机系统上

使用 .prefer_dml 启用调试器标记语言 (DML)

某些调试命令使用调试器标记语言显示文本，可以选择这些文本来快速收集更多信息。

1. 在 WinDBg 中使用 Ctrl+Break（滚动锁定）来中断目标系统上运行的代码。 目标系统可能需要一些时间才会响应。
2. 在调试器命令窗口中键入以下命令以启用 DML。

```
dbgcmd
```

```
0: kd> .prefer_dml 1
DML versions of commands on by default
```

使用 .hh 获得帮助

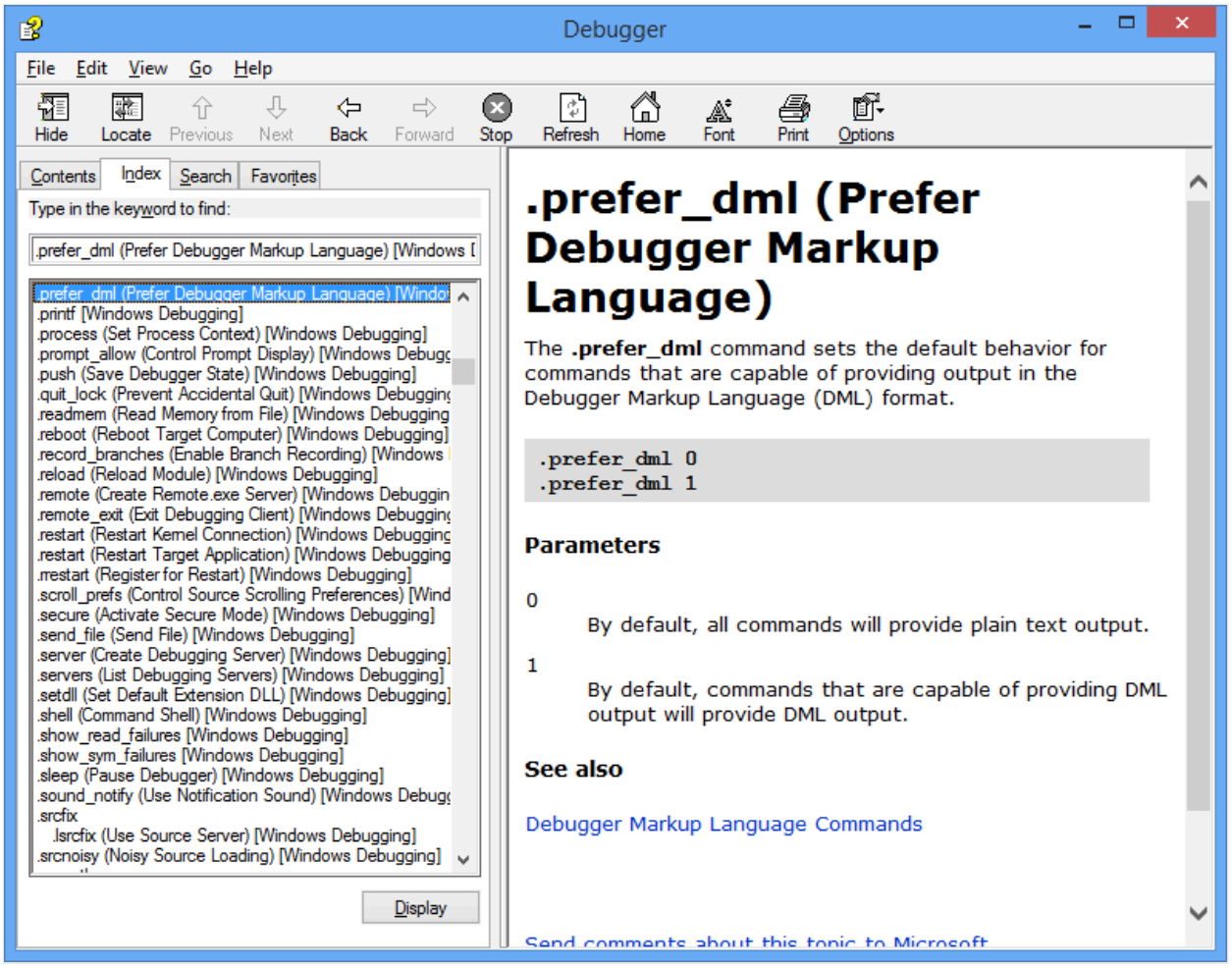
可使用 **.hh** 命令来访问参考命令帮助。

3. 键入以下命令可查看 **.prefer_dml** 的命令参考帮助。

```
dbgcmd
```

```
0: kd> .hh .prefer_dml
```

调试器帮助文件将显示 **.prefer_dml** 命令的帮助。



显示目标系统的 Windows 版本

5. 在 WinDbg 窗口中键入 **vertarget（显示目标计算机版本）** 命令，显示目标系统的详细版本信息。

```
dbgcmd
```

```
0: kd> vertarget
Windows 10 Kernel Version 9926 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 9926.0.amd64fre.fbl_awesome1501.150119-1648
Machine Name: ""
Kernel base = 0xfffff801`8d283000 PsLoadedModuleList = 0xfffff801`8d58aef0
Debug session time: Fri Feb 20 10:15:17.807 2015 (UTC - 8:00)
System Uptime: 0 days 01:31:58.931
```

列出已加载的模块

6. 在 WinDbg 窗口中键入 **lm（列出已加载的模块）** 命令以显示已加载的模块，从而验证是否使用了正确的内核模式进程。

```
dbgcmd
```

```
0: Kd> lm
start             end                 module name
fffff801`09200000 fffff801`0925f000   volmgrx    (no symbols)
fffff801`09261000 fffff801`092de000   mcupdate_GenuineIntel   (no symbols)
fffff801`092de000 fffff801`092ec000   werkernel   (export symbols)
werkernel.sys
fffff801`092ec000 fffff801`0934d000   CLFS       (export symbols)
CLFS.SYS
fffff801`0934d000 fffff801`0936f000   tm         (export symbols)
tm.sys
fffff801`0936f000 fffff801`09384000   PSHED      (export symbols)
PSHED.dll
fffff801`09384000 fffff801`0938e000   BOOTVID    (export symbols)
BOOTVID.dll
fffff801`0938e000 fffff801`093f7000   spaceport   (no symbols)
fffff801`09400000 fffff801`094cf000   Wdf01000   (no symbols)
fffff801`094d9000 fffff801`09561000   CI         (export symbols)
CI.dll
...
```

注意此实验室中省略的输出用“...” 表示。

由于尚未设置符号路径和加载符号，所以调试器中提供的信息非常有限。

第 3 部分：下载并生成 Sysvad 音频驱动程序

在第 3 部分中，将下载并生成 Sysvad 音频驱动程序。

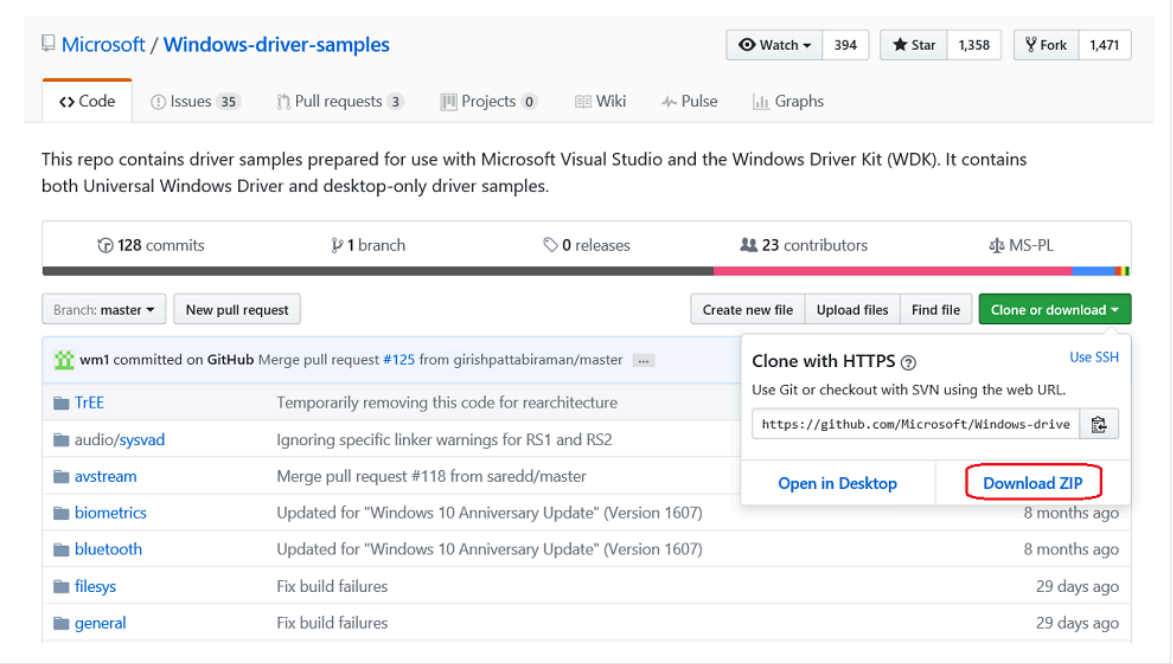
通常情况下，在使用 WinDbg 时需要用自己的驱动程序代码。 为了熟悉音频驱动程序的调试，使用了 Sysvad 虚拟音频采样驱动程序。 此示例用于说明如何通过本地内核模式代码进行单步操作。 此技术对于调试复杂的内核模式代码问题非常有用。

要下载并生成 Sysvad 示例音频驱动程序，请执行以下步骤。

1. **从 GitHub 下载并提取 Sysvad 音频示例**

   可以使用浏览器在这里查看 Sysvad 示例和 Readme.md 文件：

   https://github.com/Microsoft/Windows-driver-samples/tree/main/audio/sysvad



此实验室将展示如何以 zip 文件格式下载通用驱动程序示例。

a. 将 master.zip 文件下载到本地硬盘。

https://github.com/Microsoft/Windows-driver-samples/archive/master.zip

b. 选择并按住（或右键单击）*Windows-driver-samples-master.zip*，然后选择**全部提取**。 指定一个新文件夹，或浏览到将存储提取文件的现有文件夹。 例如，可以指定 *C:\WDK_Samples* 作为要将文件提取到的新文件夹。

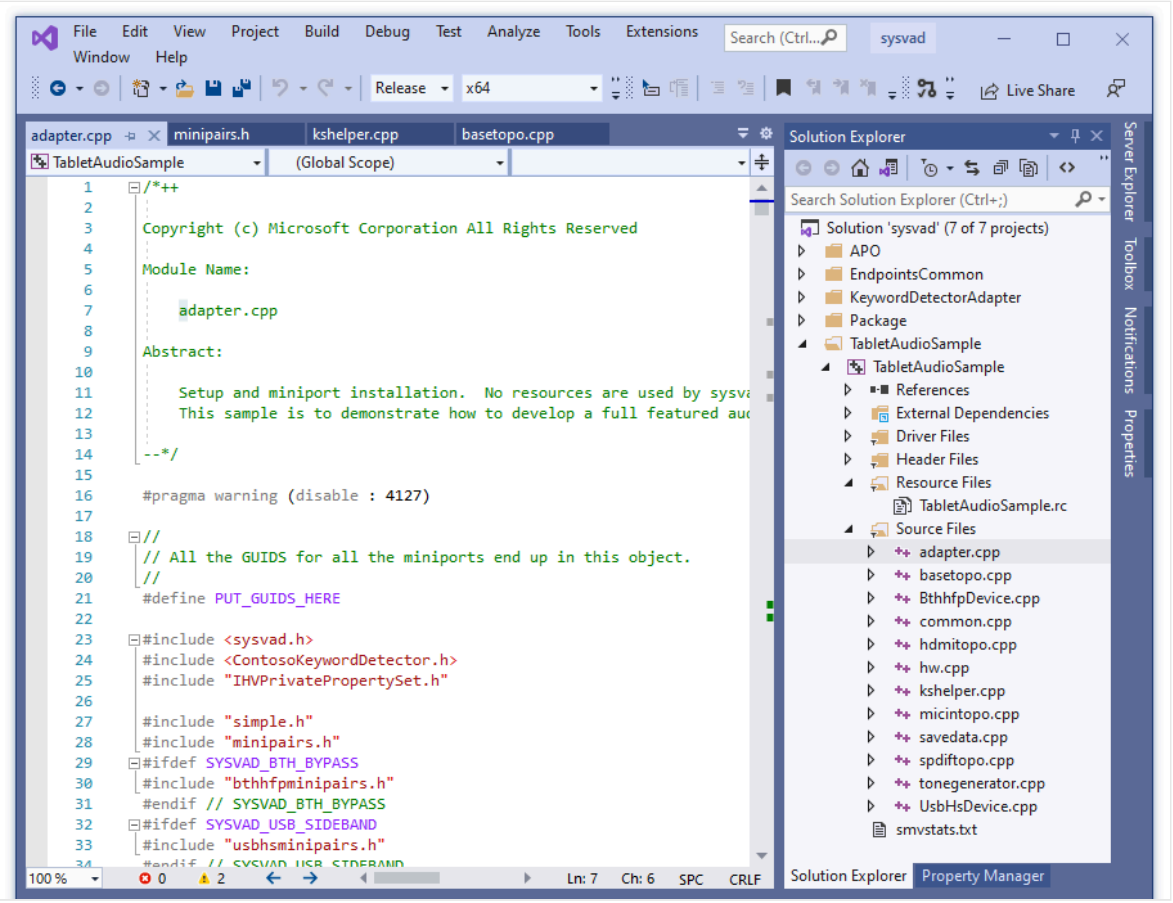
c. 提取文件后，导航到以下子文件夹。

C:\WDK_Samples\Sysvad

2. **在 Visual Studio 中打开驱动程序解决方案**

在 Visual Studio 中，选择**文件**>**打开**>**项目/解决方案**...，然后导航到包含提取文件的文件夹（例如，*C:\WDK_Samples\Sysvad*）。 双击 *Syvad* 解决方案文件。

在 Visual Studio 中，找到"解决方案资源管理器"。 （如果尚未打开，请从**视图**菜单中选择**解决方案资源管理器**。）在解决方案资源器中，可以看到一个包含几个项目的解决方案。



3. **设置示例的配置和平台**

   在“解决方案资源管理器”中，选择并按住（或右键单击）**解决方案“sysvad”（项目 7/7）**，然后选择**配置管理器**。 确保四个项目的配置和平台设置都相同。 默认情况下，所有项目的配置都设置为“Win10 调试”，而平台设置为“Win64”。 如果对一个项目进行任何配置和/或平台更改，则必须对其余三个项目进行同样的更改。

   注意此实验室假设使用的是 64 位 Windows。 如果使用的是 32 位 Windows，请生成 32 位的驱动程序。

4. **检查驱动程序签名**

   找到 TabletAudioSample。 打开 Sysvad 驱动程序的属性页面，确保将**驱动程序签名>签名模式**设置为“测试签名”。

5. 需要修改驱动程序示例，以便使用不与现有驱动程序重叠的值。 请参阅从示例代码到生产驱动程序 - 示例中要更改的内容，了解如何创建与 Windows 中安装的现有真实驱动程序共存的独特驱动程序示例。

6. **使用 Visual Studio 生成示例**

   在 Visual Studio 中，选择**生成>生成解决方案**。

   生成窗口应显示一条信息，指明所有六个项目均已成功生成。

> **提示**
>
> 如果遇到生成错误信息，请使用生成错误编号来确定修复方法。 例如，***MSBuild error MSB8040*** 描述了如何使用 Spectre 已缓解的库。

7. **找到生成的驱动程序文件**

   在文件资源管理器中，导航至包含示例提取文件的文件夹。 例如，如果先前指定的文件夹是 *C:\WDK_Samples\Sysvad*，则导航到该文件夹。 在该文件夹中，编译的驱动程序文件的位置会因在“配置管理器”中选择的配置和平台设置而异。 例如，如果不更改默认设置，那么编译后的驱动程序文件将保存到名为 *\x64\Debug* 的文件夹中，用于 64 位调试版本。

   导航至包含 TabletAudioSample 驱动程序内置文件的文件夹：

   C:\WDK_Samples\Sysvad\TabletAudioSample\x64\Debug. 该文件夹将包含 TabletAudioSample .SYS 驱动程序、符号 pdp 文件和 inf 文件。 还需要找到 DelayAPO、KWSApo 和 KeywordDetectorContosoAdapter 的 dll 和符号文件。

   安装驱动程序需要以下文件。

   展开表

文件名	说明
TabletAudioSample.sys	驱动程序文件。
TabletAudioSample.pdb	驱动程序符号文件。
tabletaudiosample.inf	一个信息 (INF) 文件，其中包含安装驱动程序所需的信息。
KeywordDetectorContosoAdapter.dll	示例关键字检测器。
KeywordDetectorContosoAdapter.pdb	示例关键字检测器符号文件。
DelayAPO.dll	示例延迟 APO。
DelayAPO.pdb	延迟 APO 符号文件。
KWSApo.dll	示例关键字检测工具 APO。
KWSApo.pdb	关键字检测工具符号文件。
TabletAudioSample.cer	TabletAudioSample 证书文件。

8. 找到 U 盘或设置网络共享，将内置驱动程序文件从主机复制到目标系统。

在下一部分中，将代码复制到目标系统，然后安装和测试驱动程序。

第 4 部分：在目标系统上安装 Sysvad 音频驱动程序示例

在第 4 部分中，将使用 devcon 来安装 Sysvad 音频驱动程序。

-> **在目标系统上**

安装驱动程序的计算机被称为*目标计算机*或*测试计算机*。 通常情况下，这是一台与开发和生成驱动程序软件包的计算机不同的计算机。 开发和生成驱动程序的计算机被称为*主机计算机*。

将驱动程序软件包移动到目标计算机并安装驱动程序的过程被称为*部署*驱动程序。

部署驱动程序前，必须打开测试签名，让目标计算机做好准备。 之后，就可以在目标系统上运行已生成的驱动程序示例了。

要在目标系统上安装驱动程序，请执行以下步骤。

1. **启用测试签名的驱动程序**

   启用运行测试签名驱动程序的功能：

   a. 打开“Windows 设置”。

   b. 在“更新和安全”中，选择“恢复”。

   c. 在“高级启动”下，选择“立即重新启动”。

   d. 计算机重启后，选择“故障排除”。

   e. 然后选择“高级选项”、“启动设置”，然后选择“重新启动”。

   f. 按 **F7** 键，选择“禁用强制驱动程序签名”。

   g. 计算机将以新的值启动。

2. -> **在目标系统上**

   安装驱动程序

   以下说明将介绍如何安装和测试示例驱动程序。

   安装此驱动程序需要的 INF 文件是 *TabletAudioSample.inf*。 在目标计算机上，以管理员身份打开“命令提示符”窗口。 导航至驱动程序包文件夹，右键单击

TabletAudioSample.inf 文件，然后选择“安装”。

此时将显示一个对话框，指示测试驱动程序是未签名驱动程序。 选择“仍然安装此驱动程序”以继续。

> **提示**
>
> 如果在安装过程中遇到任何问题，请查看以下文件了解更多信息。
> `%windir%\inf\setupapi.dev.log`

有关更详细的说明，请参阅为驱动程序部署、测试和调试配置计算机。

INF 文件包含用于安装 *tabletaudiosample.sys* 的硬件 ID。 对于 Syvad 示例，硬件 ID 为：`root\sysvad_TabletAudioSample`

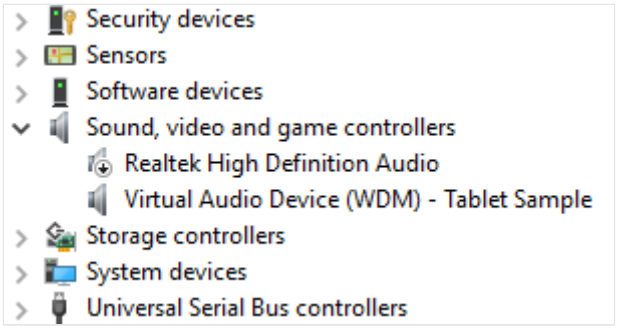
3. **在“设备管理器”中检查驱动程序**

在目标计算机的“命令提示符”窗口中，输入 **devmgmt** 以打开“设备管理器”。 在设备管理器的“查看”菜单中，选择“按类型列出设备”。

在设备树中，找到“音频设备”节点中的*虚拟音频设备 (WDM) - 平板电脑示例*。 它通常位于“声音、视频和游戏控制器”节点下。 确认它已安装并处于活动状态。

在“设备管理器”中突出显示计算机上实际硬件的驱动程序。 然后选择并按住（或右键单击）该驱动程序，选择“禁用”将该驱动程序禁用。

在“设备管理器”中确认音频硬件驱动程序是否显示了一个表示已禁用的下箭头。



成功安装示例驱动程序后，就可以对其进行测试了。

测试 Sysvad 音频驱动程序

1. 在目标计算机的"命令提示符"窗口中，输入 **devmgmt** 以打开"设备管理器"。 在"设备管理器"的"查看"菜单中，选择"按类型列出设备"。 在设备树中，找到"虚拟音频设备 (WDM) - 平板电脑示例"。

2. 打开控制面板并导航至**硬件和声音**>**管理音频设备**。 在"声音"对话框中，选择标记为"虚拟音频设备 (WDM) - 平板电脑示例"的扬声器图标，然后选择"设置默认值"，但不要选择"确定"。 这将使"声音"对话框保持打开状态。

3. 在目标计算机上找到 MP3 或其他音频文件，然后双击播放它。 然后，在"声音"对话框中，验证音量级别指示器中是否存在与"虚拟音频设备 (WDM) - 平板电脑示例"驱动程序关联的活动。

第 5 部分：使用 WinDbg 显示驱动程序的相关信息

在第 5 部分中，将设置符号路径，并使用内核调试器命令来显示 Sysvad 示例驱动程序的相关信息。

符号允许 WinDbg 显示变量名等其他信息，这些信息在调试时非常有用。 WinDbg 使用 Microsoft Visual Studio 调试符号格式进行源级调试。 它可以访问具有 PDB 符号文件的模块中的任何符号或变量。

要加载调试器，请执行以下步骤。

<- 在主机系统上

1. 如果关闭了调试器，请在管理员命令提示符窗口中使用以下命令再次将其打开。 将密钥和端口替换为之前配置的密钥和端口。

控制台

```
C:\> WinDbg -k
net:port=50010,key=2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz8
```

```
74p
```

2. 使用 Ctrl+Break（滚动锁定）来中断目标系统上运行的代码。

设置符号路径

1. 要在 WinDbg 环境中设置 Microsoft 符号服务器的符号路径，请使用 **.symfix** 命令。

```
dbgcmd

0: kd> .symfix
```

2. 要添加本地符号位置以使用本地符号，请使用 **.sympath+** 和 **.reload /f** 来添加路径。

```
dbgcmd

0: kd> .sympath+ C:\WDK_Samples\Sysvad
0: kd> .reload /f
```

注意 带有 **/f** 的 **.reload** 命令会删除指定模块的所有符号信息并重新加载符号。 在某些情况下，此命令还会重新加载或卸载模块本身。

注意必须加载适当的符号才能使用 WinDbg 提供的高级功能。 如果没有正确配置符号，则在尝试使用依赖于符号的功能时，将收到指示符号不可用的消息。

```
dbgcmd

0:000> dv
Unable to enumerate locals, HRESULT 0x80004005
Private symbols (symbols.pri) are required for locals.
Type “.hh dbgerr005” for details.
```

注意符号服务器

符号有多种使用方法。 在许多情况下，可以对计算机进行配置，以便在需要时从 Microsoft 提供的符号服务器访问符号。 本演练假定将使用这种方法。 如果环境中的符号位于其他位置，请修改使用该位置的步骤。 有关详细信息，请参阅 Windows 调试器的符号路径。

注意了解源代码符号要求

要执行源代码调试，必须生成二进制文件的校验（调试）版本。 编译器将创建符号文件（.pdb 文件）。 这些符号文件将向调试器显示二进制指令与源代码行的对应关系。 实际

源文件本身也必须要能被调试器访问。

符号文件不包含源代码文本。 为了便于调试，链接器最好不要优化代码。 如果代码经过了优化，源代码调试和访问局部变量就会变得更加困难，有时甚至几乎不可能。 如果在查看局部变量或源代码行时遇到问题，请设置以下生成选项。

设置 COMPILE_DEBUG=1

设置 ENABLE_OPTIMIZER=0

1. 在调试器的命令区键入以下内容，以便显示有关 Sysvad 驱动程序的信息。

   dbgcmd

   ```
   0: kd> lm m tabletaudiosample v
   Browse full module list
   start             end                 module name
   fffff801`14b40000 fffff801`14b86000   tabletaudiosample   (private pdb
   symbols)
   C:\Debuggers\sym\TabletAudioSample.pdb\E992C4803EBE48C7B23DC1596495CE18
   1\TabletAudioSample.pdb
       Loaded symbol image file: tabletaudiosample.sys
       Image path: \SystemRoot\system32\drivers\tabletaudiosample.sys
       Image name: tabletaudiosample.sys
       Browse all global symbols  functions  data
       Timestamp:        Thu Dec 10 12:20:26 2015 (5669DE8A)
       CheckSum:         0004891E
   ...
   ```

   有关详细信息，请参阅 lm。

2. 在调试输出中选择“浏览所有全局符号”链接，以便显示以字母 a 开头的项目符号信息。

3. 由于启用了 DML，因此输出中的某些元素是可以选择的热链接。 在调试输出中选择*数据*链接，以便显示以字母 a 开头的项目符号信息。

   dbgcmd

   ```
   0: kd> x /D /f tabletaudiosample!a*
    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

   fffff806`9adb1000 tabletaudiosample!AddDevice (struct _DRIVER_OBJECT *,
   struct _DEVICE_OBJECT *)
   ```

   有关信息，请参阅 **x（检查符号）**。

4. **!lmi** 扩展显示有关模块的详细信息。 键入 **!lmi tabletaudiosample**。 输出结果应与下图文本类似。

dbgcmd

```
0: kd> !lmi tabletaudiosample
Loaded Module Info: [tabletaudiosample]
         Module: tabletaudiosample
   Base Address: fffff8069ad90000
     Image Name: tabletaudiosample.sys
   Machine Type: 34404 (X64)
     Time Stamp: 58ebe848 Mon Apr 10 13:17:12 2017
           Size: 48000
       CheckSum: 42df7
Characteristics: 22
Debug Data Dirs: Type  Size     VA  Pointer
             CODEVIEW    a7,  e5f4,    d1f4 RSDS - GUID: {5395F0C5-
AE50-4C56-AD31-DD5473BD318F}
               Age: 1, Pdb: C:\Windows-driver-samples-
master\audio\sysvad\TabletAudioSample\x64\Debug\TabletAudioSample.pdb
                   ??   250,  e69c,    d29c [Data not mapped]
     Image Type: MEMORY   - Image read successfully from loaded memory.
    Symbol Type: PDB      - Symbols loaded successfully from image
header.
                 C:\Program Files (x86)\Windows
Kits\10\Debuggers\x64\sym\TabletAudioSample.pdb\5395F0C5AE504C56AD31DD5
473BD318F1\TabletAudioSample.pdb
       Compiler: Resource - front end [0.0 bld 0] - back end [14.0 bld
24210]
    Load Report: private symbols & lines, not source indexed
                 C:\Program Files (x86)\Windows
Kits\10\Debuggers\x64\sym\TabletAudioSample.pdb\5395F0C5AE504C56AD31DD5
473BD318F1\TabletAudioSample.pdb
```

5. 使用 **!dh** 扩展来显示标头信息，如下所示。

dbgcmd

```
0: kd> !dh tabletaudiosample

File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
    8664 machine (X64)
       9 number of sections
5669DE8A time date stamp Thu Dec 10 12:20:26 2015

       0 file pointer to symbol table
       0 number of symbols
      F0 size of optional header
      22 characteristics
            Executable
```

```
            App can handle >2gb addresses
...
```

第 6 部分：显示即插即用设备树信息

在第 6 部分中，将介绍有关 Sysvad 示例设备驱动程序的信息，以及它在即插即用设备树中的位置。

即插即用设备树中有关设备驱动程序的信息对故障排除很有用。 例如，如果设备驱动程序没有驻留在设备树中，则表明设备驱动程序的安装可能存在问题。

有关设备节点调试扩展的详细信息，请参阅 **!devnode**。

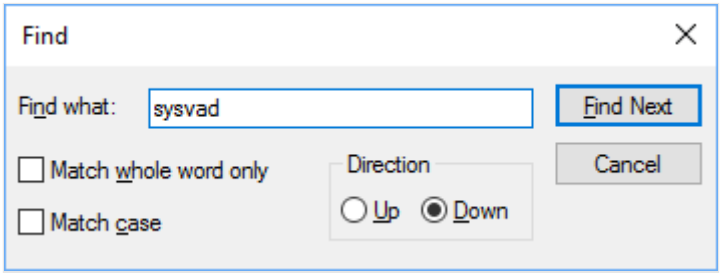
<- 在主机系统上

1. 要查看即插即用设备树中的所有设备节点，请输入 **!devnode 0 1** 命令。 运行该命令可能需要一两分钟。 在此期间，WinDbg 的状态区域中将显示“*忙”。

```
dbgcmd
```

```
0: kd> !devnode 0 1
Dumping IopRootDeviceNode (= 0xffffe0005a3a8d30)
DevNode 0xffffe0005a3a8d30 for PDO 0xffffe0005a3a9e50
  InstancePath is "HTREE\ROOT\0"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0xffffe0005a3a3d30 for PDO 0xffffe0005a3a4e50
    InstancePath is "ROOT\volmgr\0000"
    ServiceName is "volmgr"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
    DevNode 0xffffe0005a324560 for PDO 0xffffe0005bd95ca0…
...
```

2. 使用 Ctrl+F 在生成的输出中查找设备驱动程序的名称 *sysvad*。



在 Syvad 的 !devnode 输出中将出现一个名为 `sysvad_TabletAudioSample` 的设备节点条目。

```
```

dbgcmd

```
  DevNode 0xffffe00086e68190 for PDO 0xffffe00089c575a0
    InstancePath is "ROOT\sysvad_TabletAudioSample\0000"
    ServiceName is "sysvad_tabletaudiosample"
    State = DeviceNodeStarted (0x308)
...
```

请注意，显示的是 PDO 地址和 DevNode 地址。

3. 使用 `!devnode 0 1 sysvad_TabletAudioSample` 命令可显示与 Sysvad 设备驱动程序关联的即插即用信息。

dbgcmd

```
0: kd> !devnode 0 1 sysvad_TabletAudioSample
Dumping IopRootDeviceNode (= 0xffffe00082df8d30)
DevNode 0xffffe00086e68190 for PDO 0xffffe00089c575a0
  InstancePath is "ROOT\sysvad_TabletAudioSample\0000"
  ServiceName is "sysvad_tabletaudiosample"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0xffffe000897fb650 for PDO 0xffffe00089927e30
    InstancePath is "SWD\MMDEVAPI\{0.0.0.00000000}.{64097438-cdc0-4007-
a19e-62e789062e20}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
  DevNode 0xffffe00086d2f5f0 for PDO 0xffffe00089939ae0
    InstancePath is "SWD\MMDEVAPI\{0.0.0.00000000}.{78880f4e-9571-44a4-
a9df-960bde446487}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
  DevNode 0xffffe00089759bb0 for PDO 0xffffe000875aa060
    InstancePath is "SWD\MMDEVAPI\{0.0.0.00000000}.{7cad07f2-d0a0-4b9b-
8100-8dc735e9c447}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
  DevNode 0xffffe00087735010 for PDO 0xffffe000872068c0
    InstancePath is "SWD\MMDEVAPI\{0.0.0.00000000}.{fc38551b-e69f-4b86-
9661-ae6da78bc3c6}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
  DevNode 0xffffe00088457670 for PDO 0xffffe0008562b830
    InstancePath is "SWD\MMDEVAPI\{0.0.1.00000000}.{0894b831-c9fe-4c56-
86a6-092380fc5628}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
  DevNode 0xffffe000893dbb70 for PDO 0xffffe00089d68060
    InstancePath is "SWD\MMDEVAPI\{0.0.1.00000000}.{15eb6b5c-aa54-47b8-
959a-0cff2c1500db}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
```

```
  DevNode 0xffffe00088e6f250 for PDO 0xffffe00089f6e990
    InstancePath is "SWD\MMDEVAPI\{0.0.1.00000000}.{778c07f0-af9f-43f2-
8b8d-490024f87239}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
  DevNode 0xffffe000862eb4b0 for PDO 0xffffe000884443a0
    InstancePath is "SWD\MMDEVAPI\{0.0.1.00000000}.{e4b72c7c-be50-45df-
94f5-0f2922b85983}"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeStartPostWork (0x307)
```

4. 上一条命令显示的输出包括与运行中的驱动程序实例相关联的 PDO，在本例中为 *0xffffe00089c575a0*。 输入 **!devobj**<*PDO address*> 命令，以便显示与 Sysvad 设备驱动程序关联的即插即用信息。 使用 **!devnode** 在计算机上显示的 PDO 地址，而不是此处显示的地址。

dbgcmd

```
0: kd> !devobj 0xffffe00089c575a0
Device object (ffffe00089c575a0) is for:
0000004e \Driver\PnpManager DriverObject ffffe00082d47e60
Current Irp 00000000 RefCount 65 Type 0000001d Flags 00001040
SecurityDescriptor ffffc102b0f6d171 DevExt 00000000 DevObjExt
ffffe00089c576f0 DevNode ffffe00086e68190
ExtensionFlags (0000000000)
Characteristics (0x00000180)  FILE_AUTOGENERATED_DEVICE_NAME,
FILE_DEVICE_SECURE_OPEN
AttachedDevice (Upper) ffffe00088386a50
\Driver\sysvad_tabletaudiosample
Device queue is not busy.
```

5. **!devobj** 命令中显示的输出包括所附设备的名称：\Driver\sysvad_tabletaudiosample。 使用位掩码为 2 的 **!drvobj** 命令，以便显示与所连接设备关联的信息。

dbgcmd

```
0: kd> !drvobj \Driver\sysvad_tabletaudiosample 2
Driver object (ffffe0008834f670) is for:
\Driver\sysvad_tabletaudiosample
DriverEntry:   fffff80114b45310  tabletaudiosample!FxDriverEntry
DriverStartIo: 00000000
DriverUnload:  fffff80114b5fea0
tabletaudiosample!DriverUnload
AddDevice:     fffff80114b5f000  tabletaudiosample!AddDevice

Dispatch routines:
[00] IRP_MJ_CREATE                      fffff80117b49a20
portcls!DispatchCreate
[01] IRP_MJ_CREATE_NAMED_PIPE           fffff8015a949a00
```

```
nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       fffff80115e26f90
ks!DispatchCleanup
[03] IRP_MJ_READ                        fffff80115e32710
ks!DispatchRead
[04] IRP_MJ_WRITE                       fffff80115e327e0
ks!DispatchWrite
[05] IRP_MJ_QUERY_INFORMATION           fffff8015a949a00
nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION             fffff8015a949a00
nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                    fffff8015a949a00
nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                      fffff8015a949a00
nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS               fffff80115e32640
ks!DispatchFlush
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    fffff8015a949a00
nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION      fffff8015a949a00
nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           fffff8015a949a00
nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         fffff8015a949a00
nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL              fffff80115e27480
ks!DispatchDeviceIoControl
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     fffff8015a949a00
nt!IopInvalidDeviceRequest
[10] IRP_MJ_SHUTDOWN                    fffff8015a949a00
nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                fffff8015a949a00
nt!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                     fffff8015a949a00
nt!IopInvalidDeviceRequest
[13] IRP_MJ_CREATE_MAILSLOT             fffff8015a949a00
nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              fffff80115e326a0
ks!DispatchQuerySecurity
[15] IRP_MJ_SET_SECURITY                fffff80115e32770
ks!DispatchSetSecurity
[16] IRP_MJ_POWER                       fffff80117b3dce0
portcls!DispatchPower
[17] IRP_MJ_SYSTEM_CONTROL              fffff80117b13d30
portcls!PcWmiSystemControl
[18] IRP_MJ_DEVICE_CHANGE               fffff8015a949a00
nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                 fffff8015a949a00
nt!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA                   fffff8015a949a00
nt!IopInvalidDeviceRequest
[1b] IRP_MJ_PNP                         fffff80114b5f7d0
tabletaudiosample!PnpHandler
```

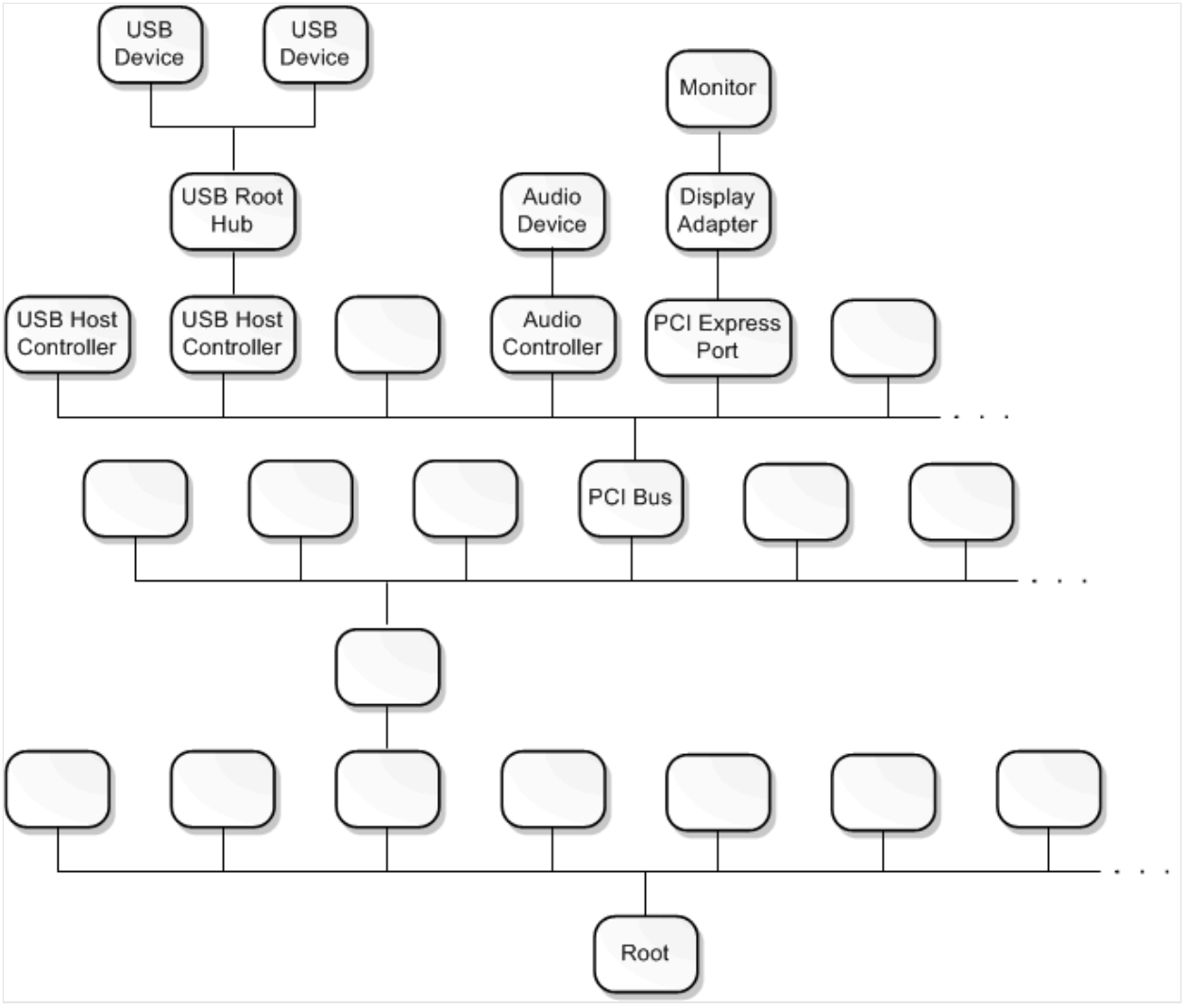
6. 输入 **!devstack**<*PDO address*> 命令，以便显示与设备驱动程序关联的即插即用信息。 **!devnode 0 1** 命令中显示的输出包括与正在运行的驱动程序实例相关联的 PDO 地址。 本例中，该地址为 *0xffffe00089c575a0*。 使用 **!devnode** 在计算机上显示的 PDO 地址，而不是下图所示的地址。

dbgcmd

```
0: kd> !devstack 0xffffe00089c575a0
  !DevObj           !DrvObj            !DevExt           ObjectName
  ffffe00088d212e0  \Driver\ksthunk    ffffe00088d21430  0000007b
  ffffe00088386a50  \Driver\sysvad_tabletaudiosampleffffe00088386ba0
0000007a
> ffffe00089c575a0  \Driver\PnpManager 00000000  0000004e
!DevNode ffffe00086e68190 :
  DeviceInst is "ROOT\sysvad_TabletAudioSample\0000"
  ServiceName is "sysvad_tabletaudiosample"
```

输出显示有一个非常简单的设备驱动程序堆栈。 sysvad_TabletAudioSample 驱动程序是 PnPManager 节点的子节点。 PnPManager 是一个根节点。

此图显示了一个更为复杂的设备节点树。



注意 有关更复杂的驱动程序堆栈的详细信息，请参阅驱动程序堆栈和设备节点和设备堆栈。

第 7 部分：使用断点

在第 7 部分中，将使用断点在特定点停止代码执行。

使用命令来设置断点

断点用于在特定代码行停止代码的执行。 然后，就可以从这一点开始向前执行代码，对特定的代码部分进行调试。

要使用调试命令来设置断点，请使用以下 **b** 命令之一。

展开表

bp	设置一个断点，该断点将一直处于活动状态，直到其所在模块被卸载。
bu	设置一个断点，该断点在卸载模块时未解析，并在重新加载模块时重新启用。
bm	为符号设置一个断点。 此命令将适当使用 bu 或 bp，并允许使用通配符 * 在每个匹配的符号（如类中的所有方法）上设置断点。

1. 使用 WinDbg UI 确认**调试**>**源模式**是否已在当前 WinDbg 会话中启用。
2. 输入以下命令，将本地代码位置添加到源路径中。

```
dbgcmd
```

```
.sympath+ C:\WDK_Samples\Sysvad
```

3. 键入以下命令，将本地符号位置添加到符号路径中。

```
dbgcmd
```

```
.sympath+ C:\WDK_Samples\Sysvad
```

4. **设置调试掩码**

   在使用驱动程序时，查看它可能显示的所有信息会非常方便。 键入以下内容以更改默认调试位掩码，从而在调试器中显示来自目标系统的所有调试信息。

```
```

```
dbgcmd

0: kd> ed nt!Kd_DEFAULT_MASK 0xFFFFFFFF
```

5. 使用驱动程序名称后跟要在其中设置断点的函数名称 (AddDevice)（以感叹号分隔），通过 **bm** 命令来设置断点。

```
dbgcmd

0: kd> bm tabletaudiosample!AddDevice
breakpoint 1 redefined
  1: fffff801`14b5f000 @!"tabletaudiosample!AddDevice"
```

   可以将不同的语法结合设置变量来使用，如 <module>!<symbol>、<class>::<method>、“<file.cpp>:<line number>”，或者跳过若干次 <condition><#>。 有关详细信息，请参阅 Using Breakpoints。

6. 列出当前断点，以确认通过键入 **bl** 命令设置了断点。

```
dbgcmd

0: kd> bl
1 e fffff801`14b5f000     0001 (0001) tabletaudiosample!AddDevice
```

7. 输入 go 命令 **g** 在目标系统上重新启动代码执行。

8. -> **在目标系统上**

   在 Windows 中，使用图标或输入 **mmc devmgmt.msc** 来打开“设备管理器”。 在“设备管理器”中展开“声音、视频和游戏控制器”节点。 选择并按住（或右键单击）虚拟音频驱动程序条目，然后从菜单中选择“禁用”。

9. 再次选择并按住（或右键单击）虚拟音频驱动程序条目，然后从菜单中选择“启用”。

10. <- **在主机系统上**

    这会导致 Windows 重新加载调用 AddDevice 的驱动程序。 这样将导致 AddDevice 调试断点触发，而目标系统上的驱动程序代码应停止执行。

```
dbgcmd

Breakpoint 1 hit
tabletaudiosample!AddDevice:
fffff801`14baf000 4889542410      mov     qword ptr [rsp+10h],rdx
```

如果源路径设置正确，则应停止 adapter.cpp 中的 AddDevice 例程位置

dbgcmd

```
{
    PAGED_CODE();

    NTSTATUS        ntStatus;
    ULONG           maxObjects;

    DPF(D_TERSE, ("[AddDevice]"));

    maxObjects = g_MaxMiniports;

    #ifdef SYSVAD_BTH_BYPASS
    //
    // Allow three (3) Bluetooth hands-free profile devices.
    //
    maxObjects += g_MaxBthHfpMiniports * 3;
    #endif // SYSVAD_BTH_BYPASS

    // Tell the class driver to add the device.
    //
    ntStatus =
        PcAddAdapterDevice
        (
            DriverObject,
            PhysicalDeviceObject,
            PCPFNSTARTDEVICE(StartDevice),
            maxObjects,
            0
        );
    return ntStatus;
} // AddDevice
```

11. 键入 **p** 命令或按 F10 键，逐行查看代码。 可以从 sysvad AddDevice 代码跳转到 PpvUtilCall、PnpCallAddDevice，然后跳转到 PipCallDriverAddDevice Windows 代码。 可以为 **p** 命令指定一个数字，以便向前执行多行，例如 *p 5*。

12. 完成代码单步执行后，使用 go 命令 **g** 在目标系统上重新启动执行。

设置内存访问断点

还可以设置在访问内存位置时触发的断点。 使用 **ba**（访问时中断）命令，语法如下。

dbgcmd

```
ba <access> <size> <address> {options}
```

展开表

选项	说明
e	execute（当 CPU 从地址中提取指令时）
r	read/write（当 CPU 读取或写入地址时）
w	write（当 CPU 写入地址时）

请注意，任何时候都只能设置四个数据断点，并且必须确保正确对齐数据，否则将无法触发断点（word 必须以可被 2 整除的地址结尾，dwords 必须可被 4 整除，quadword 必须可被 0 或 8 整除）

例如，要在特定内存地址上设置读/写断点，可使用如下命令。

dbgcmd

```
ba r 4 fffff800`7bc9eff0
```

修改断点状态

可以使用以下命令修改现有的断点。

展开表

bl	列出断点。
bc	清除列表中的断点。 使用 bc * 可以清除所有断点。
bd	禁用断点。 使用 bd * 禁用所有断点。
be	启用断点。 使用 be * 启用所有断点。

或者，也可以通过选择**编辑**>**断点**来修改断点。 请注意，断点对话框只对现有断点有效。 必须通过命令行来设置新的断点。

在 MixerVolume 上设置断点

设备驱动程序加载后，音频驱动程序代码的不同部分会被调用以响应各种事件。 在下一部分中将设置一个断点，当用户调整虚拟音频驱动程序的音量控制时，就会触发该断点。

要在 MixerVolume 上设置断点，请执行以下步骤。

1. <- **在主机系统上**

要找到更改卷的方法，请使用 x 命令列出 CAdapterCommon 中包含卷字符串的符号。

```
dbgcmd

kd> x tabletaudiosample!CAdapterCommon::*
...
fffff800`7bce26a0 tabletaudiosample!CAdapterCommon::MixerVolumeWrite
(unsigned long, unsigned long, long)
…
```

使用 CTRL+F 在输出中向上搜索卷，找到 MixerVolumeWrite 方法。

2. 使用 bc * 清除之前的断点。

3. 使用以下命令在 CAdapterCommon::MixerVolumeWrite 例程上设置符号断点。

```
dbgcmd

kd> bm tabletaudiosample!CAdapterCommon::MixerVolumeWrite
  1: fffff801`177b26a0
@!"tabletaudiosample!CAdapterCommon::MixerVolumeWrite"
```

4. 列出断点，以便确认断点已正确设置。

```
dbgcmd

kd> bl
1 e fffff801`177b26a0 [c:\WDK_Samples\audio\sysvad\common.cpp @ 1668]
0001 (0001) tabletaudiosample!CAdapterCommon::MixerVolumeWrite
```

5. 输入 go 命令 **g** 在目标系统上重新启动代码执行。

6. 在控制面板中选择**硬件和声音**>**声音**。 选择并按住（或右键单击）“接收器说明示例”，然后选择“属性”。 选择“音量”选项卡。调整滑块音量。

7. 这将导致 SetMixerVolume 调试断点触发，并且目标系统上的驱动程序代码停止执行。

```
dbgcmd

kd> g
Breakpoint 1 hit
tabletaudiosample!CAdapterCommon::MixerVolumeWrite:
fffff801`177b26a0 44894c2420      mov     dword ptr [rsp+20h],r9d
```

应该在 common.cpp 中的这一行停止运行

dbgcmd

```
{
    if (m_pHW)
    {
        m_pHW->SetMixerVolume(Index, Channel, Value);
    }
} // MixerVolumeWrite
```

8. 使用 dv 命令显示当前变量及其值。 有关变量的详细信息将在本实验室的下一部分中提供。

dbgcmd

```
2: kd> dv
           this = 0x00000000`00000010
         ulNode = 0x344
      ulChannel = 0x210a45f8
        lVolume = 0n24
```

9. 按 **F10** 单步执行代码。

10. 按 **F5** 完成 MixerVolumeWrite 代码的执行。

摘要 - 从"调试器命令"窗口单步执行代码

以下是可以用来单步执行代码的命令（括号中显示的是相关的键盘快捷键）。

- 中断 (Ctrl+Break) - 只要系统正在运行并与 WinDbg 通信（内核调试器中的顺序为 Ctrl+C），该命令就会中断系统。
- 单步跳过 (F10) - 该命令可让代码一次只执行一条语句或指令。 如果遇到调用，则代码执行将越过调用，而不会进入被调用例程。 （如果编程语言为 C 或 C++，且 WinDbg 处于源模式，则可以使用**调试**>**源模式**来打开或关闭源模式）。
- 单步执行 (F11) - 该命令与单步跳过类似，只是调用的执行会进入被调用例程。
- 单步跳出 (Shift+F11) – 该命令可让代码执行到当前例程（调用栈中的当前位置）并退出。 如果已查看足够多的例程，这将非常有用。
- 运行到光标（F7 或 Ctrl+F10）- 将光标放在源代码或反汇编窗口中需要中断执行的位置，然后按 F7；代码执行将运行到该点。 请注意，如果代码执行流没有到达游标指示的点（比如没有执行 IF 语句），则 WinDbg 不会中断，因为代码执行并未到达指示的点。

- 运行 (F5) – 运行，直到遇到断点或发生错误检查等事件。

高级选项

- 将指令设置到当前行 (Ctrl+Shift+I) - 在源窗口中，可以将光标放在某一行上，然后输入此快捷键，只要让它继续运行（例如使用 F5 或 F10），代码就会从该点开始执行。 如果要重试序列，这将非常有用，但需要注意一些事项。 例如，寄存器和变量不会被设置为代码执行到该行时的状态。

- 直接设置 eip 寄存器 - 可以在 eip 寄存器中输入一个值，一旦按下 F5（或 F10、F11 等），就会从该地址开始执行。 这与将指令设置到光标指定的当前行类似，只不过指定的是程序集指令的地址。

通过用户界面进行操作比从命令行进行操作更容易，因此推荐使用这种方法。 如有必要，可使用以下命令在命令行中单步执行源文件：

- .lines - 启用源行信息。

- bp main - 在模块开始处设置初始断点。

- l+t - 将按源行进行单步执行。

- 选择**调试**>**源模式**以进入源模式；只使用 `L+t` 命令是不够的。

- l+s - 在提示符下显示源行。

- g - 运行程序，直到输入了“main”。

- p - 执行一个源行。

有关详细信息，请参阅调试参考文档中的 WinDbg 中的源代码调试（经典版）。

在代码中设置断点

可以通过添加 `DebugBreak()` 语句、重新生成项目和重新安装驱动程序来在代码中设置断点。 每次启用驱动程序时，该断点都会被触发，因此这将是一种在早期开发阶段而非生产代码中使用的技术。 这种技术不像使用断点命令动态设置断点那样灵活。

提示：可能需要保留一份未添加断点的 Sysvad 驱动程序副本，以用于进一步的实验室工作。

1. 在示例代码中添加 `DebugBreak()` 语句，设置每次运行 AddDevice 方法时中断。

```
dbgcmd
```

```
...
    // Insert the DebugBreak() statment before the  PcAddAdapterDevice
is called.
```

```
    //

    DebugBreak()

    // Tell the class driver to add the device.
    //
    ntStatus =
        PcAddAdapterDevice
        (
            DriverObject,
            PhysicalDeviceObject,
            PCPFNSTARTDEVICE(StartDevice),
            maxObjects,
            0
        );

    return ntStatus;
} // AddDevice
```

2. 按照之前介绍的所有步骤，在 Microsoft Visual Studio 中重新生成驱动程序，并将其重新安装到目标计算机。 在安装更新的驱动程序之前，请务必卸载现有的驱动程序。

3. 清除之前的任何断点，并确保调试器已连接到目标计算机。

4. 当代码运行到 `DebugBreak` 语句时，将停止执行并显示一条信息。

```
dbgcmd
```

```
KERNELBASE!DebugBreak:
77b3b770 defe     __debugbreak
```

第 8 部分：显示变量

在第 8 部分中，将使用调试器命令来显示变量。

在代码执行过程中检查变量以确认代码是否按预期运行，可能会很有用。 此实验室会检查音频驱动程序产生声音时的变量。

1. 使用 **dv** 命令检查与 tabletaudiosample!CMiniportWaveRT::New* 关联的局部变量。

```
dbgcmd
```

```
kd> dv tabletaudiosample!CMiniportWaveRT::New*
```

2. 清除之前的断点

```
dbgcmd
```

```
bc *
```

3. 使用以下命令在 CMiniportWaveCyclicStreamMSVAD 例程上设置符号断点。

```
dbgcmd
```

```
0: kd> bm tabletaudiosample!CMiniportWaveRT::NewStream
  1: fffff801`177dffc0 @!"tabletaudiosample!CMiniportWaveRT::NewStream"
```

4. 输入 go 命令 **g** 在目标系统上重新启动代码执行。

5. -> **在目标系统上**

   找到一个小型媒体文件（如扩展名为 .wav 文件的 Windows 通知声音文件），然后选择该文件进行播放。 例如，可以使用 Windows\Media 目录中的 Ring05.wav。

6. <- **在主机系统上**

   播放媒体文件时，断点应被触发，目标系统上的驱动程序代码应停止执行。

```
dbgcmd
```

```
Breakpoint 1 hit
tabletaudiosample!CMiniportWaveRT::NewStream:
fffff801`177dffc0 44894c2420      mov     dword ptr [rsp+20h],r9d
```

源代码窗口应突出显示 NewStream 函数入口处的括号。

```
dbgcmd
```

```
/*++

Routine Description:

  The NewStream function creates a new instance of a logical stream
  associated with a specified physical channel. Callers of NewStream
should
  run at IRQL PASSIVE_LEVEL.

Arguments:

  OutStream -

  OuterUnknown -

  Pin -
```

```
  Capture -

  DataFormat -

Return Value:

  NT status code.

--*/
{

...
```

7. **局部变量**

   输入 **dv** 命令可以显示指定帧的所有局部变量的名称和值。

```
dbgcmd

0: kd> dv
                this = 0xffffe000`4436f8e0
           OutStream = 0xffffe000`49d2f130
        OuterUnknown = 0xffffe000`4436fa30
                 Pin = 0
             Capture = 0x01 '
          DataFormat = 0xffffe000`44227790
signalProcessingMode = {487E9220-E000-FFFF-30F1-D24900E0FFFF}
            ntStatus = 0n1055
              stream = 0x00000000`00000200
```

8. **使用 DML 显示变量**

   要使用 DML 来浏览变量，请选择带下划线的元素。 选择操作会生成一个 **dx（显示 NatVis 表达式）**命令，用于向下钻取嵌套数据结构。

```
dbgcmd

0: kd> dx -r1 (*((tabletaudiosample!CMiniportWaveRT
*)0xffffe001d10b8380))
(*((tabletaudiosample!CMiniportWaveRT *)0xffffe001d10b8380)) :  [Type:
CMiniportWaveRT]
    [+0x020] m_lRefCount      : 0
    [+0x028] m_pUnknownOuter  : 0xffffe001d1477e50 : [Type: IUnknown *]
    [+0x030] m_ulLoopbackAllocated : 0x2050
    [+0x034] m_ulSystemAllocated : 0x180
    [+0x038] m_ulOffloadAllocated : 0x0
    [+0x03c] m_dwCaptureAllocatedModes : 0x0

0: kd> dx -r1 (*((tabletaudiosample!_GUID *)0xffffd001c8acd348))
(*((tabletaudiosample!_GUID *)0xffffd001c8acd348)) : {487E9220-E000-
FFFF-30F1-D24900E0FFFF} [Type: _GUID]
```

```
    [<Raw View>]

0: kd> dx -r1 -n (*((tabletaudiosample!_GUID *)0xffffd001c8acd348))
(*((tabletaudiosample!_GUID *)0xffffd001c8acd348)) :  [Type: _GUID]
    [+0x000] Data1            : 0x487e9220
    [+0x004] Data2            : 0xe000
    [+0x006] Data3            : 0xffff
    [+0x008] Data4            :  [Type: unsigned char [8]]

0: kd> dx -r1 -n (*((tabletaudiosample!unsigned char (*)
[8])0xffffd001c8acd350))
(*((tabletaudiosample!unsigned char (*)[8])0xffffd001c8acd350)) :
[Type: unsigned char [8]]
    [0]              : 0x30
    [1]              : 0xf1
    [2]              : 0xd2
    [3]              : 0x49
    [4]              : 0x0
    [5]              : 0xe0
    [6]              : 0xff
    [7]              : 0xff
```

9. **全局变量**

   可以通过键入 **? <variable name>** 来查找全局变量的内存位置。

   dbgcmd

   ```
   0: kd> ? signalProcessingMode
   Evaluate expression: -52768896396472 = ffffd001`c8acd348
   ```

10. 这样将返回变量的内存位置，本例中为 *ffffd001`c8acd348*。 可以使用前一条命令返回的内存位置，通过键入 **dd** 命令来转储该位置的值，从而查看内存位置的内容。

    dbgcmd

    ```
    0: kd> dd ffffd001`c8acd348
    ffffd001`c8acd348  487e9220 ffffe000 49d2f130 ffffe000
    ffffd001`c8acd358  4837c468 ffffe000 18221570 ffffc000
    ffffd001`c8acd368  4436f8e0 ffffe000 487e9220 ffffe000
    ffffd001`c8acd378  18ab145b fffff801 4837c420 ffffe000
    ffffd001`c8acd388  4436f8e0 ffffe000 49d2f130 ffffe000
    ffffd001`c8acd398  4436fa30 ffffe000 00000000 00000000
    ffffd001`c8acd3a8  00000001 00000000 44227790 ffffe000
    ffffd001`c8acd3b8  18adc7f9 fffff801 495972a0 ffffe000
    ```

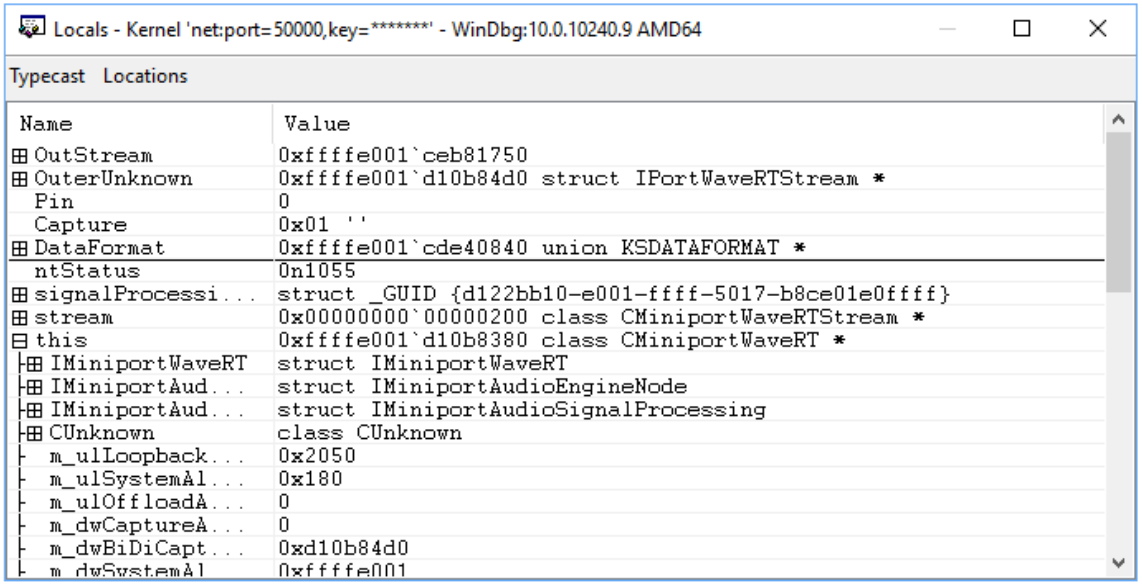
11. 还可以将变量名称与 **dd** 命令配合使用。

    dbgcmd

```
0: kd> dd signalProcessingMode
ffffd001`c8acd348  487e9220 ffffe000 49d2f130 ffffe000
ffffd001`c8acd358  4837c468 ffffe000 18221570 ffffc000
ffffd001`c8acd368  4436f8e0 ffffe000 487e9220 ffffe000
ffffd001`c8acd378  18ab145b fffff801 4837c420 ffffe000
ffffd001`c8acd388  4436f8e0 ffffe000 49d2f130 ffffe000
ffffd001`c8acd398  4436fa30 ffffe000 00000000 00000000
ffffd001`c8acd3a8  00000001 00000000 44227790 ffffe000
ffffd001`c8acd3b8  18adc7f9 fffff801 495972a0 ffffe000
```

12. **显示变量**

    使用**查看**>**局部**菜单项可显示局部变量。 该界面还提供深入了解更复杂的数据结构的功能。



13. 使用 p 或 F10 在代码中向前移动约 10 行，直到高亮显示 ntStatus = IsFormatSupported(Pin, Capture, DataFormat); 代码行。

```C++
    PAGED_CODE();

    ASSERT(OutStream);
    ASSERT(DataFormat);

    DPF_ENTER(("[CMiniportWaveRT::NewStream]"));

    NTSTATUS                    ntStatus = STATUS_SUCCESS;
    PCMiniportWaveRTStream      stream = NULL;
    GUID                        signalProcessingMode =
AUDIO_SIGNALPROCESSINGMODE_DEFAULT;

    *OutStream = NULL;
```

```
     //
    // If the data format attributes were specified, extract them.
    //
    if ( DataFormat->Flags & KSDATAFORMAT_ATTRIBUTES )
    {
        // The attributes are aligned (QWORD alignment) after the data
format
        PKSMULTIPLE_ITEM attributes = (PKSMULTIPLE_ITEM)
(((PBYTE)DataFormat) + ((DataFormat->FormatSize + FILE_QUAD_ALIGNMENT)
& ~FILE_QUAD_ALIGNMENT));
        ntStatus = GetAttributesFromAttributeList(attributes,
attributes->Size, &signalProcessingMode);
    }

    // Check if we have enough streams.
    //
    if (NT_SUCCESS(ntStatus))
    {
        ntStatus = ValidateStreamCreate(Pin, Capture,
signalProcessingMode);
    }

    // Determine if the format is valid.
    //
    if (NT_SUCCESS(ntStatus))
    {
        ntStatus = IsFormatSupported(Pin, Capture, DataFormat);
    }

...
```

14. 使用 **dv** 命令显示指定帧的所有局部变量的名称和值。 请注意，正如所预料的那样，这些值与上次运行该命令时的值有所不同，因为运行了额外的代码来更改局部变量，而且有些变量现在不在当前帧中，或者其值已发生变化。

dbgcmd

```
2: kd> dv
                this = 0xffffe001`d1182000
           OutStream = 0xffffe001`d4776d20
        OuterUnknown = 0xffffe001`d4776bc8
                 Pin = 0
             Capture = 0x00 '
          DataFormat = 0xffffe001`cd7609b0
signalProcessingMode = {4780004E-7133-41D8-8C74-660DADD2C0EE}
            ntStatus = 0n0
              stream = 0x00000000`00000000
```

第 9 部分：查看调用堆栈

在第 9 部分中，将查看调用堆栈，以检查调用者/并列代码。

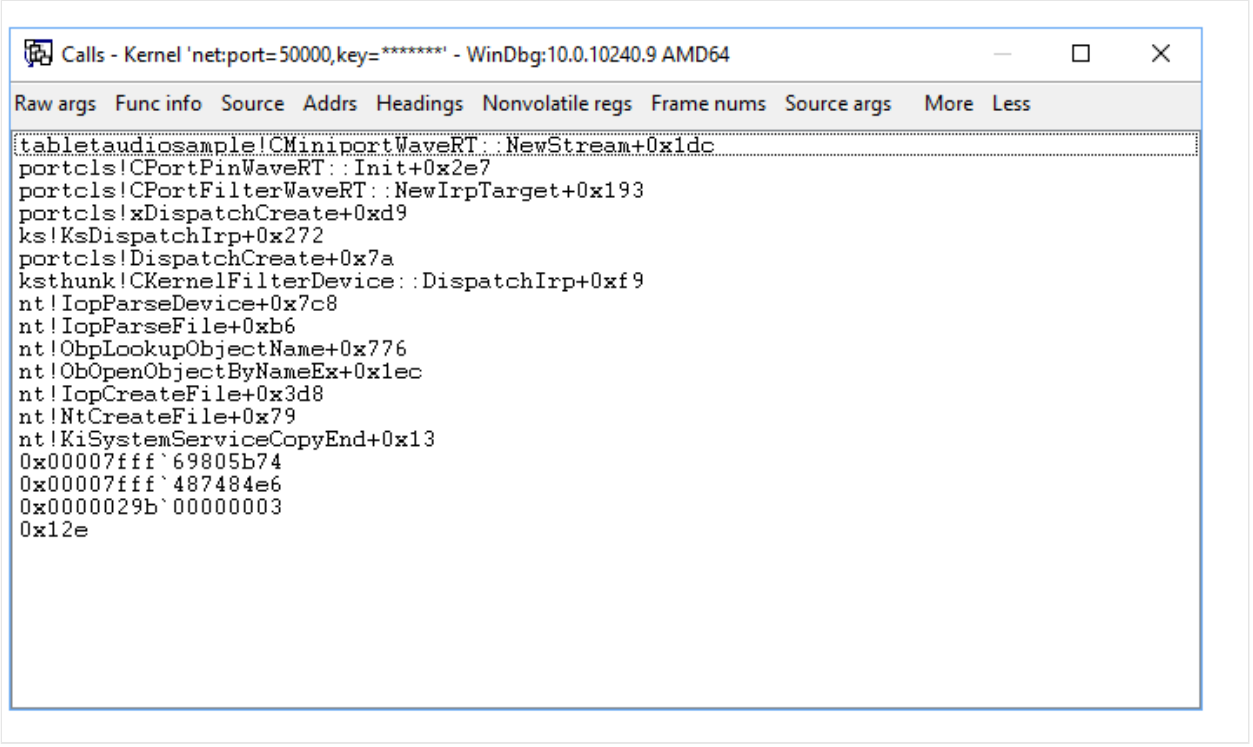
调用堆栈是指向程序计数器当前位置的函数调用链。 调用堆栈上最上面的函数是当前函数，而下一个函数是调用当前函数的函数，依此类推。

要显示调用堆栈，请使用 k* 命令：

展开表

kb	显示堆栈和前三个参数。
kp	显示堆栈和完整的参数列表。
kn	允许查看堆栈和旁边的帧信息。

如果想让调用堆栈可供使用，可以选择**查看**>**调用堆栈**进行查看。 选择窗口顶部的列可切换显示其他信息。



该输出显示了在中断状态下调试示例适配器代码时的调用堆栈。

dbgcmd

```
0: kd> kb
# RetAddr           : Args to Child
: Call Site
00 fffff800`7a0fa607 : ffffe001`d1182000 ffffe001`d4776d20 ffffe001`d4776bc8
ffffe001`00000000 : tabletaudiosample!CMiniportWaveRT::NewStream+0x1dc
[c:\data1\threshold\audio\endpointscommon\minwavert.cpp @ 597]
01 fffff800`7a0fb2c3 : 00000000`00000000 ffffe001`d122bb10 ffffe001`ceb81750
ffffe001`d173f058 : portcls!CPortPinWaveRT::Init+0x2e7
```

```
02 fffff800`7a0fc7f9 : ffffe001`d4776bc0 00000000`00000000 ffffe001`d10b8380
ffffe001`d122bb10 : portcls!CPortFilterWaveRT::NewIrpTarget+0x193
03 fffff800`7a180552 : 00000000`00000000 ffffe001`d10b8380 ffffe001`d122bb10
ffffe001`d4565600 : portcls!xDispatchCreate+0xd9
04 fffff800`7a109a9a : ffffe001`d10b84d0 ffffe001`d10b8380 00000000`00000000
ffffe001`00000000 : ks!KsDispatchIrp+0x272
05 fffff800`7bd314b1 : ffffe001`d122bb10 ffffd001`c3098590 ffffe001`d122bd90
ffffe001`ce80da70 : portcls!DispatchCreate+0x7a
06 fffff803`cda1bfa8 : 00000000`00000024 00000000`00000000 00000000`00000000
ffffe001`d122bb10 : ksthunk!CKernelFilterDevice::DispatchIrp+0xf9
07 fffff803`cda7b306 : 00000000`000001f0 ffffe001`d48ce690 ffffe001`d13d6400
ffffe001`d13d64c0 : nt!IopParseDevice+0x7c8
08 fffff803`cda12916 : 00000000`000001f0 ffffd001`c30988d0 ffffe001`d13d6490
fffff803`cda7b250 : nt!IopParseFile+0xb6
09 fffff803`cda1131c : ffffe001`d2ccb001 ffffd001`c30989e0 00ffffe0`00000040
ffffe001`cd127dc0 : nt!ObpLookupObjectName+0x776
0a fffff803`cd9fedb8 : ffffe001`00000001 ffffe001`d48ce690 00000000`00000000
00000000`00000000 : nt!ObOpenObjectByNameEx+0x1ec
0b fffff803`cd9fe919 : 000000ee`6d1fc8d8 000000ee`6d1fc788 000000ee`6d1fc7e0
000000ee`6d1fc7d0 : nt!IopCreateFile+0x3d8
0c fffff803`cd752fa3 : ffffc000`1f296870 fffff803`cd9d9fbd ffffd001`c3098be8
00000000`00000000 : nt!NtCreateFile+0x79
0d 00007fff`69805b74 : 00007fff`487484e6 0000029b`00000003 00000000`0000012e
00000000`00000000 : nt!KiSystemServiceCopyEnd+0x13
0e 00007fff`487484e6 : 0000029b`00000003 00000000`0000012e 00000000`00000000
00000000`00000000 : 0x00007fff`69805b74
0f 0000029b`00000003 : 00000000`0000012e 00000000`00000000 00000000`00000000
00000000`00000000 : 0x00007fff`487484e6
10 00000000`0000012e : 00000000`00000000 00000000`00000000 00000000`00000000
00000000`00000080 : 0x0000029b`00000003
11 00000000`00000000 : 00000000`00000000 00000000`00000000 00000000`00000080
00000000`00000000 : 0x12e
```

可以使用 DML 进一步浏览代码。 选择第一个 00 条目时，使用 **.frame（设置本地上下文）**命令来设置上下文，然后使用 **dv（显示局部变量）**命令来显示局部变量。

dbgcmd

```
0: kd> .frame 0n0;dv /t /v
00 ffffd001`c30981d0 fffff800`7a0fa607
tabletaudiosample!CMiniportWaveRT::NewStream+0x1dc
[c:\data1\threshold\audio\endpointscommon\minwavert.cpp @ 597]
ffffd001`c30982b0 class CMiniportWaveRT * this = 0xffffe001`d1182000
ffffd001`c30982b8 struct IMiniportWaveRTStream ** OutStream =
0xffffe001`d4776d20
ffffd001`c30982c0 struct IPortWaveRTStream * OuterUnknown =
0xffffe001`d4776bc8
ffffd001`c30982c8 unsigned long Pin = 0
ffffd001`c30982d0 unsigned char Capture = 0x00 '
ffffd001`c30982d8 union KSDATAFORMAT * DataFormat = 0xffffe001`cd7609b0
ffffd001`c3098270 struct _GUID signalProcessingMode = {4780004E-7133-41D8-
8C74-660DADD2C0EE}
```

```
ffffd001`c3098210 long ntStatus = 0n0
ffffd001`c3098218 class CMiniportWaveRTStream * stream = 0x00000000`00000000
```

第 10 部分：显示进程和线程

在第 10 部分中，将使用调试器命令来显示进程和线程。

处理

要更改当前进程上下文，请使用 .process <process> 命令。 以下示例演示了如何识别一个进程并将上下文切换到该进程。

- 使用 `!process` 命令显示当前播放声音的进程。

  有关详细信息，请参阅 !process

输出显示该进程与 audiodg.exe 关联。 如果仍处于本主题上一节所述的断点处，则当前进程应与 audiodg.exe 映像相关联。

<- 在主机系统上

dbgcmd

```
0: kd> !process
PROCESS ffffe001d147c840
    SessionId: 0  Cid: 10f0    Peb: ee6cf8a000  ParentCid: 0434
    DirBase: d2122000  ObjectTable: ffffc0001f191ac0  HandleCount: <Data Not
Accessible>
    Image: audiodg.exe
    VadRoot ffffe001d4222f70 Vads 70 Clone 0 Private 504. Modified 16.
Locked 0.
    DeviceMap ffffc00019113080
    Token                             ffffc0001f1d4060
    ElapsedTime                       <Invalid>
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         81632
    QuotaPoolUsage[NonPagedPool]      9704
    Working Set Sizes (now,min,max)  (2154, 1814, 2109) (8616KB, 7256KB,
8436KB)
    PeakWorkingSetSize                2101
    VirtualSize                       2097192 Mb
    PeakVirtualSize                   2097192 Mb
    PageFaultCount                    2336
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      1573

        THREAD ffffe001d173e840  Cid 10f0.1dac  Teb: 000000ee6cf8b000
Win32Thread: ffffe001d1118cf0 WAIT: (UserRequest) UserMode Non-Alertable
```

```
            ffffe001d16c4dd0  NotificationEvent
            ffffe001d08b0840  ProcessObject

        THREAD ffffe001ceb77080  Cid 10f0.16dc  Teb: 000000ee6cf8d000
Win32Thread: 0000000000000000 WAIT: (WrQueue) UserMode Alertable
            ffffe001cf2d1840  QueueObject

        THREAD ffffe001d112c840  Cid 10f0.0a4c  Teb: 000000ee6cf8f000
Win32Thread: 0000000000000000 WAIT: (WrQueue) UserMode Alertable
            ffffe001cf2d1840  QueueObject

        THREAD ffffe001d16c7840  Cid 10f0.13c4  Teb: 000000ee6cf91000
Win32Thread: 0000000000000000 WAIT: (WrQueue) UserMode Alertable
            ffffe001cf2d1840  QueueObject

        THREAD ffffe001cec67840  Cid 10f0.0dbc  Teb: 000000ee6cf93000
Win32Thread: 0000000000000000 WAIT: (WrQueue) UserMode Alertable
            ffffe001d173e5c0  QueueObject

        THREAD ffffe001d1117840  Cid 10f0.1d6c  Teb: 000000ee6cf95000
Win32Thread: 0000000000000000 WAIT: (WrQueue) UserMode Alertable
            ffffe001d173e5c0  QueueObject

        THREAD ffffe001cdeae840  Cid 10f0.0298  Teb: 000000ee6cf97000
Win32Thread: 0000000000000000 RUNNING on processor 2
```

请注意，与该进程关联的一个线程处于 RUNNING 状态。 此线程支持在触发断点时播放媒体剪辑。

使用 **!process 0 0** 命令显示所有进程的摘要信息。 在命令输出中使用 CTRL+F 查找与 audiodg.exe 映像关联的进程 ID。 在下面的示例中，进程 ID 为 *ffffe001d147c840*。

记录下电脑上与 audiodg.exe 关联的进程 ID，以便以后在此实验室中使用。

---

dbgcmd

```
...

PROCESS ffffe001d147c840
    SessionId: 0  Cid: 10f0    Peb: ee6cf8a000  ParentCid: 0434
    DirBase: d2122000  ObjectTable: ffffc0001f191ac0  HandleCount: <Data Not
Accessible>
    Image: audiodg.exe
...
```

在调试器中输入 g 以便向前运行代码，直到媒体剪辑播放完毕。 然后按 Ctrl+ScrLk (Ctrl+Break) 键进入调试器。使用 !process 命令确认现在运行的是另一个进程。

dbgcmd

```
!process
PROCESS ffffe001cd0ad040
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 001aa000  ObjectTable: ffffc00017214000  HandleCount: <Data Not
Accessible>
    Image: System
    VadRoot ffffe001d402b820 Vads 438 Clone 0 Private 13417. Modified 87866.
Locked 64.
    DeviceMap ffffc0001721a070
    Token                             ffffc00017216a60
    ElapsedTime                       05:04:54.716
    UserTime                          00:00:00.000
    KernelTime                        00:00:20.531
    QuotaPoolUsage[PagedPool]         0
    QuotaPoolUsage[NonPagedPool]      0
    Working Set Sizes (now,min,max)  (1720, 50, 450) (6880KB, 200KB, 1800KB)
    PeakWorkingSetSize                15853
    VirtualSize                       58 Mb
    PeakVirtualSize                   74 Mb
    PageFaultCount                    46128
   MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      66

        THREAD ffffe001cd0295c0  Cid 0004.000c  Teb: 0000000000000000
Win32Thread: 0000000000000000 WAIT: (Executive) KernelMode Non-Alertable
            fffff803cd8e0120  SynchronizationEvent

        THREAD ffffe001cd02a6c0  Cid 0004.0010  Teb: 0000000000000000
Win32Thread: 0000000000000000 WAIT: (Executive) KernelMode Non-Alertable
            fffff803cd8e0ba0  Semaphore Limit 0x7fffffff
...
```

上面的输出显示，*ffffe001cd0ad040* 的不同系统进程正在运行。 图像名称显示的是系统，而不是 audiodg.exe。

现在使用 !process 命令切换到与 audiodg.exe 关联的进程。 在示例中，进程 ID 为 *ffffe001d147c840*。 将示例中的进程 ID 替换为之前记录下的进程 ID。

dbgcmd

```
0: kd> !process  ffffe001d147c840
PROCESS ffffe001d147c840
    SessionId: 0  Cid: 10f0    Peb: ee6cf8a000  ParentCid: 0434
    DirBase: d2122000  ObjectTable: ffffc0001f191ac0  HandleCount: <Data Not
Accessible>
    Image: audiodg.exe
    VadRoot ffffe001d4222f70 Vads 60 Clone 0 Private 299. Modified 152.
Locked 0.
    DeviceMap ffffc00019113080
    Token                             ffffc0001f1d4060
```

```
    ElapsedTime                       1 Day 01:53:14.490
    UserTime                          00:00:00.031
    KernelTime                        00:00:00.031
    QuotaPoolUsage[PagedPool]         81552
    QuotaPoolUsage[NonPagedPool]      8344
    Working Set Sizes (now,min,max)  (1915, 1814, 2109) (7660KB, 7256KB,
8436KB)
    PeakWorkingSetSize                2116
    VirtualSize                       2097189 Mb
    PeakVirtualSize                   2097192 Mb
    PageFaultCount                    2464
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      1418

        THREAD ffffe001d173e840  Cid 10f0.1dac  Teb: 000000ee6cf8b000
Win32Thread: ffffe001d1118cf0 WAIT: (UserRequest) UserMode Non-Alertable
            ffffe001d16c4dd0  NotificationEvent
            ffffe001d08b0840  ProcessObject
        Not impersonating
        DeviceMap                 ffffc00019113080
        Owning Process            ffffe001d147c840       Image:
audiodg.exe
        Attached Process          N/A            Image:         N/A
        Wait Start TickCount      338852         Ticks: 197682
(0:00:51:28.781)
        Context Switch Count      36             IdealProcessor: 0
        UserTime                  00:00:00.015
        KernelTime                00:00:00.000
        Win32 Start Address 0x00007ff7fb928de0
        Stack Init ffffd001c2ec6dd0 Current ffffd001c2ec60c0
        Base ffffd001c2ec7000 Limit ffffd001c2ec1000 Call 0
        Priority 8 BasePriority 8 UnusualBoost 0 ForegroundBoost 0
IoPriority 2 PagePriority 5
        Kernel stack not resident.

        THREAD ffffe001d115c080  Cid 10f0.15b4  Teb: 000000ee6cf9b000
Win32Thread: 0000000000000000 WAIT: (WrQueue) UserMode Alertable
            ffffe001d0bf0640  QueueObject
        Not impersonating
        DeviceMap                 ffffc00019113080
        Owning Process            ffffe001d147c840       Image:
audiodg.exe
        Attached Process          N/A            Image:         N/A
        Wait Start TickCount      338852         Ticks: 197682
(0:00:51:28.781)
        Context Switch Count      1              IdealProcessor: 0
        UserTime                  00:00:00.000
        KernelTime                00:00:00.000
        Win32 Start Address 0x00007fff6978b350
        Stack Init ffffd001c3143dd0 Current ffffd001c3143520
        Base ffffd001c3144000 Limit ffffd001c313e000 Call 0
        Priority 8 BasePriority 8 UnusualBoost 0 ForegroundBoost 0
IoPriority 2 PagePriority 5
        Kernel stack not resident.
```

```
        THREAD ffffe001d3a27040  Cid 10f0.17f4  Teb: 000000ee6cf9d000
Win32Thread: 0000000000000000 WAIT: (WrQueue) UserMode Alertable
            ffffe001d173e5c0  QueueObject
        Not impersonating
        DeviceMap                 ffffc00019113080
        Owning Process            ffffe001d147c840       Image:
audiodg.exe
        Attached Process          N/A            Image:         N/A
        Wait Start TickCount      518918         Ticks: 17616
(0:00:04:35.250)
        Context Switch Count      9              IdealProcessor: 1
        UserTime                  00:00:00.000
        KernelTime                00:00:00.000
        Win32 Start Address 0x00007fff6978b350
        Stack Init ffffd001c70c6dd0 Current ffffd001c70c6520
        Base ffffd001c70c7000 Limit ffffd001c70c1000 Call 0
        Priority 9 BasePriority 8 UnusualBoost 0 ForegroundBoost 0
IoPriority 2 PagePriority 5
        Kernel stack not resident.
```

由于此代码未处于活动状态，因此所有线程都处于 WAIT 状态，正如预期的那样。

线程

用于查看和设置线程的命令与进程的命令非常相似。 使用 **!thread** 命令来查看线程。 使用 **.thread** 来设置当前线程。

要浏览与媒体播放器关联的线程，请再次播放媒体剪辑。 如果上一节所述的断点仍然存在，则将在 audiodg.exe 的上下文中停止。

使用 !thread -1 0 显示当前线程的简要信息。 这样会显示线程地址、线程和进程 ID、线程环境块 (TEB) 地址、创建线程运行的 Win32 函数地址（如有），以及线程的计划状态。

dbgcmd

```
0: kd> !thread -1 0
THREAD ffffe001d3a27040  Cid 10f0.17f4  Teb: 000000ee6cf9d000 Win32Thread:
0000000000000000 RUNNING on processor 0
```

要查看有关正在运行的线程的详细信息，请键入 **!thread**。 应显示类似如下的信息。

dbgcmd

```
0: kd> !thread
THREAD ffffe001d3a27040  Cid 10f0.17f4  Teb: 000000ee6cf9d000 Win32Thread:
0000000000000000 RUNNING on processor 0
IRP List:
    ffffe001d429e580: (0006,02c8) Flags: 000008b4  Mdl: 00000000
```

```
Not impersonating
DeviceMap                 ffffc00019113080
Owning Process            ffffe001d147c840       Image:         audiodg.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      537630         Ticks: 0
Context Switch Count      63             IdealProcessor: 1
UserTime                  00:00:00.000
KernelTime                00:00:00.015
Win32 Start Address 0x00007fff6978b350
Stack Init ffffd001c70c6dd0 Current ffffd001c70c6520
Base ffffd001c70c7000 Limit ffffd001c70c1000 Call 0
Priority 8 BasePriority 8 UnusualBoost 0 ForegroundBoost 0 IoPriority 2
PagePriority 5
Child-SP          RetAddr           : Args to Child
: Call Site
ffffd001`c70c62a8 fffff800`7a0fa607 : ffffe001`d4aec5c0 ffffe001`cdefd3d8
ffffe001`d4aec5c0 ffffe001`cdefd390 :
tabletaudiosample!CMiniportWaveRT::NewStream
[c:\data1\threshold\audio\endpointscommon\minwavert.cpp @ 562]
ffffd001`c70c62b0 fffff800`7a0fb2c3 : 00000000`00000000 ffffe001`d429e580
ffffe001`d4ea47b0 ffffe001`cdefd3d8 : portcls!CPortPinWaveRT::Init+0x2e7
ffffd001`c70c6340 fffff800`7a0fc7f9 : ffffe001`d4aec430 00000000`00000000
ffffe001`d10b8380 ffffe001`d429e580 :
portcls!CPortFilterWaveRT::NewIrpTarget+0x193
ffffd001`c70c63c0 fffff800`7a180552 : 00000000`00000000 ffffe001`d10b8380
ffffe001`d429e580 ffffe001`d4565600 : portcls!xDispatchCreate+0xd9
ffffd001`c70c6450 fffff800`7a109a9a : ffffe001`d10b84d0 ffffe001`d10b8380
00000000`00000000 ffffe001`00000000 : ks!KsDispatchIrp+0x272
ffffd001`c70c6510 fffff800`7bd314b1 : ffffe001`d429e580 ffffd001`c70c6590
ffffe001`d429e800 ffffe001`ce80da70 : portcls!DispatchCreate+0x7a
ffffd001`c70c6540 fffff803`cda1bfa8 : 00000000`00000025 00000000`00000000
00000000`00000000 ffffe001`d429e580 :
ksthunk!CKernelFilterDevice::DispatchIrp+0xf9
ffffd001`c70c65a0 fffff803`cda7b306 : 00000000`000002fc ffffe001`d5e0d510
00000000`00000000 ffffe001`d3341bd0 : nt!IopParseDevice+0x7c8
ffffd001`c70c6770 fffff803`cda12916 : 00000000`000002fc ffffd001`c70c68d0
ffffe001`d3341ba0 fffff803`cda7b250 : nt!IopParseFile+0xb6
ffffd001`c70c67d0 fffff803`cda1131c : ffffe001`ceb6c601 ffffd001`c70c69e0
00000000`00000040 ffffe001`cd127dc0 : nt!ObpLookupObjectName+0x776
ffffd001`c70c6970 fffff803`cd9fedb8 : ffff8ab8`00000001 ffffe001`d5e0d510
00000000`00000000 00000000`00000000 : nt!ObOpenObjectByNameEx+0x1ec
ffffd001`c70c6a90 fffff803`cd9fe919 : 000000ee`6d37c6e8 00000004`6d37c500
000000ee`6d37c5f0 000000ee`6d37c5e0 : nt!IopCreateFile+0x3d8
ffffd001`c70c6b40 fffff803`cd752fa3 : fffff6fb`7da05360 fffff6fb`40a6c0a8
fffff681`4d815760 ffff8ab8`92895e23 : nt!NtCreateFile+0x79
ffffd001`c70c6bd0 00007fff`69805b74 : 00007fff`487484e6 0000029b`00000003
00000000`0000012e 00000000`00000000 : nt!KiSystemServiceCopyEnd+0x13
(TrapFrame @ ffffd001`c70c6c40)
000000ee`6d37c568 00007fff`487484e6 : 0000029b`00000003 00000000`0000012e
00000000`00000000 00000000`00000000 : 0x00007fff`69805b74
000000ee`6d37c570 0000029b`00000003 : 00000000`0000012e 00000000`00000000
00000000`00000000 00000000`00000000 : 0x00007fff`487484e6
000000ee`6d37c578 00000000`0000012e : 00000000`00000000 00000000`00000000
00000000`00000000 00000000`00000080 : 0x0000029b`00000003
```

```
000000ee`6d37c580 00000000`00000000 : 00000000`00000000 00000000`00000000
00000000`00000080 00000000`00000000 : 0x12e
```

使用 k 命令查看与线程关联的调用堆栈。

dbgcmd

```
0: kd> k
# Child-SP          RetAddr           Call Site
00 ffffd001`c70c62a8 fffff800`7a0fa607
tabletaudiosample!CMiniportWaveRT::NewStream
[c:\data1\threshold\audio\endpointscommon\minwavert.cpp @ 562]
01 ffffd001`c70c62b0 fffff800`7a0fb2c3 portcls!CPortPinWaveRT::Init+0x2e7
02 ffffd001`c70c6340 fffff800`7a0fc7f9
portcls!CPortFilterWaveRT::NewIrpTarget+0x193
03 ffffd001`c70c63c0 fffff800`7a180552 portcls!xDispatchCreate+0xd9
04 ffffd001`c70c6450 fffff800`7a109a9a ks!KsDispatchIrp+0x272
05 ffffd001`c70c6510 fffff800`7bd314b1 portcls!DispatchCreate+0x7a
06 ffffd001`c70c6540 fffff803`cda1bfa8
ksthunk!CKernelFilterDevice::DispatchIrp+0xf9
07 ffffd001`c70c65a0 fffff803`cda7b306 nt!IopParseDevice+0x7c8
08 ffffd001`c70c6770 fffff803`cda12916 nt!IopParseFile+0xb6
09 ffffd001`c70c67d0 fffff803`cda1131c nt!ObpLookupObjectName+0x776
0a ffffd001`c70c6970 fffff803`cd9fedb8 nt!ObOpenObjectByNameEx+0x1ec
0b ffffd001`c70c6a90 fffff803`cd9fe919 nt!IopCreateFile+0x3d8
0c ffffd001`c70c6b40 fffff803`cd752fa3 nt!NtCreateFile+0x79
0d ffffd001`c70c6bd0 00007fff`69805b74 nt!KiSystemServiceCopyEnd+0x13
0e 000000ee`6d37c568 00007fff`487484e6 0x00007fff`69805b74
0f 000000ee`6d37c570 0000029b`00000003 0x00007fff`487484e6
10 000000ee`6d37c578 00000000`0000012e 0x0000029b`00000003
11 000000ee`6d37c580 00000000`00000000 0x12e
```

在调试器中输入 g 以便向前运行代码，直到媒体剪辑播放完毕。 然后按 Ctrl - ScrLk (Ctrl-Break) 键终端调试器。使用 !thread 命令确认现在运行的是不同的线程。

dbgcmd

```
0: kd> !thread
THREAD ffffe001ce80b840  Cid 17e4.01ec  Teb: 00000071fa9b9000 Win32Thread:
ffffe001d41690d0 RUNNING on processor 0
Not impersonating
DeviceMap                 ffffc0001974e2c0
Owning Process            ffffe001d1760840       Image:         rundll32.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      538040         Ticks: 0
Context Switch Count      3181840        IdealProcessor: 0
UserTime                  00:00:08.250
KernelTime                00:00:10.796
Win32 Start Address 0x00007ff6d2f24270
Stack Init ffffd001cd16afd0 Current ffffd001cd16a730
Base ffffd001cd16b000 Limit ffffd001cd165000 Call 0
```

```
Priority 8 BasePriority 8 UnusualBoost 0 ForegroundBoost 0 IoPriority 2
PagePriority 5

Child-SP          RetAddr           : Args to Child
: Call Site
fffff803`cf373d18 fffff800`7a202852 : fffff803`cf373e60 00000000`00000001
ffffe001`cf4ed330 00000000`0000ffff : nt!DbgBreakPointWithStatus
fffff803`cf373d20 fffff803`cd6742c6 : ffffe001`cf4ed2f0 fffff803`cf373e60
00000000`00000001 00000000`0004e4b8 : kdnic!TXSendCompleteDpc+0x142
fffff803`cf373d60 fffff803`cd74d495 : 00000000`00000000 fffff803`cd923180
fffff803`cde1f4b0 fffff901`40669010 : nt!KiRetireDpcList+0x5f6
fffff803`cf373fb0 fffff803`cd74d2a0 : 00000000`00000090 0000000e`0000006a
00000000`00000092 00000000`00000000 : nt!KxRetireDpcList+0x5 (TrapFrame @
fffff803`cf373e70)
ffffd001`cd16a6c0 fffff803`cd74bd75 : 00000000`00000000 fffff803`cd74a031
00000000`00000000 00000000`00000000 : nt!KiDispatchInterruptContinue
ffffd001`cd16a6f0 fffff803`cd74a031 : 00000000`00000000 00000000`00000000
ffffe001`cff4d2a0 fffff803`cd67738e : nt!KiDpcInterruptBypass+0x25
ffffd001`cd16a700 fffff960`50cdb5a4 : fffff901`400006d0 00000000`00000001
fffff901`40000d60 ffffd001`cd16a9f0 : nt!KiInterruptDispatchNoLockNoEtw+0xb1
(TrapFrame @ ffffd001`cd16a700)
ffffd001`cd16a890 fffff960`50c66b2f : 00000000`00000000 fffff901`40669010
fffff901`42358580 fffff901`40000d60 : win32kfull!Win32FreePoolImpl+0x34
ffffd001`cd16a8c0 fffff960`50c68cd6 : 00000000`00000000 ffffd001`cd16a9f0
fffff901`400006d0 fffff901`400c0460 : win32kfull!EXLATEOBJ::vAltUnlock+0x1f
ffffd001`cd16a8f0 fffff803`cd752fa3 : 00000000`00000000 00000000`00000000
ffffe001`ce80b840 00000000`00000000 : win32kfull!NtGdiAlphaBlend+0x1d16
ffffd001`cd16add0 00007fff`674c1494 : 00007fff`674b1e97 0000a7c6`daee0559
00000000`00000001 0000020b`741f3c50 : nt!KiSystemServiceCopyEnd+0x13
(TrapFrame @ ffffd001`cd16ae40)
00000071`fa74c9a8 00007fff`674b1e97 : 0000a7c6`daee0559 00000000`00000001
0000020b`741f3c50 00000000`00ffffff : 0x00007fff`674c1494
00000071`fa74c9b0 0000a7c6`daee0559 : 00000000`00000001 0000020b`741f3c50
00000000`00ffffff 00000000`00000030 : 0x00007fff`674b1e97
00000071`fa74c9b8 00000000`00000001 : 0000020b`741f3c50 00000000`00ffffff
00000000`00000030 00000000`01010bff : 0x0000a7c6`daee0559
00000071`fa74c9c0 0000020b`741f3c50 : 00000000`00ffffff 00000000`00000030
00000000`01010bff 00000000`00000000 : 0x1
00000071`fa74c9c8 00000000`00ffffff : 00000000`00000030 00000000`01010bff
00000000`00000000 00000000`000000c0 : 0x0000020b`741f3c50
00000071`fa74c9d0 00000000`00000030 : 00000000`01010bff 00000000`00000000
00000000`000000c0 00000000`00000030 : 0xffffff
00000071`fa74c9d8 00000000`01010bff : 00000000`00000000 00000000`000000c0
00000000`00000030 00000071`00000030 : 0x30
00000071`fa74c9e0 00000000`00000000 : 00000000`000000c0 00000000`00000030
00000071`00000030 00000071`01ff8000 : 0x1010bff
```

图像名称是 rundll32.exe，这确实不是与播放媒体剪辑关联的图像名称。

注意 要设置当前线程，请键入 .thread <thread number>。

有关线程和进程的详细信息，请参阅以下参考：

线程和进程

更改上下文

第 11 部分：IRQL、寄存器和反汇编

查看保存的 IRQL

在第 11 部分中，将显示 IRQL 和寄存器的内容。

<- 在主机系统上

中断请求级别 (IRQL) 用于管理中断服务的优先级。 每个处理器都有一个 IRQL 设置，线程可以提高或降低该设置。 发生在处理器 IRQL 设置或低于 IRQL 设置的中断将被屏蔽，不会干扰当前操作。 高于处理器 IRQL 设置的中断优先于当前操作。 **!irql** 扩展显示调试器中断之前目标计算机上当前处理器的中断请求级别 (IRQL)。 当目标计算机中断调试器时，IRQL 会发生变化，但调试器中断前有效的 IRQL 会被保存并通过 !irql 显示。

dbgcmd

```
0: kd> !irql
Debugger saved IRQL for processor 0x0 -- 2 (DISPATCH_LEVEL)
```

<查看寄存器和反汇编

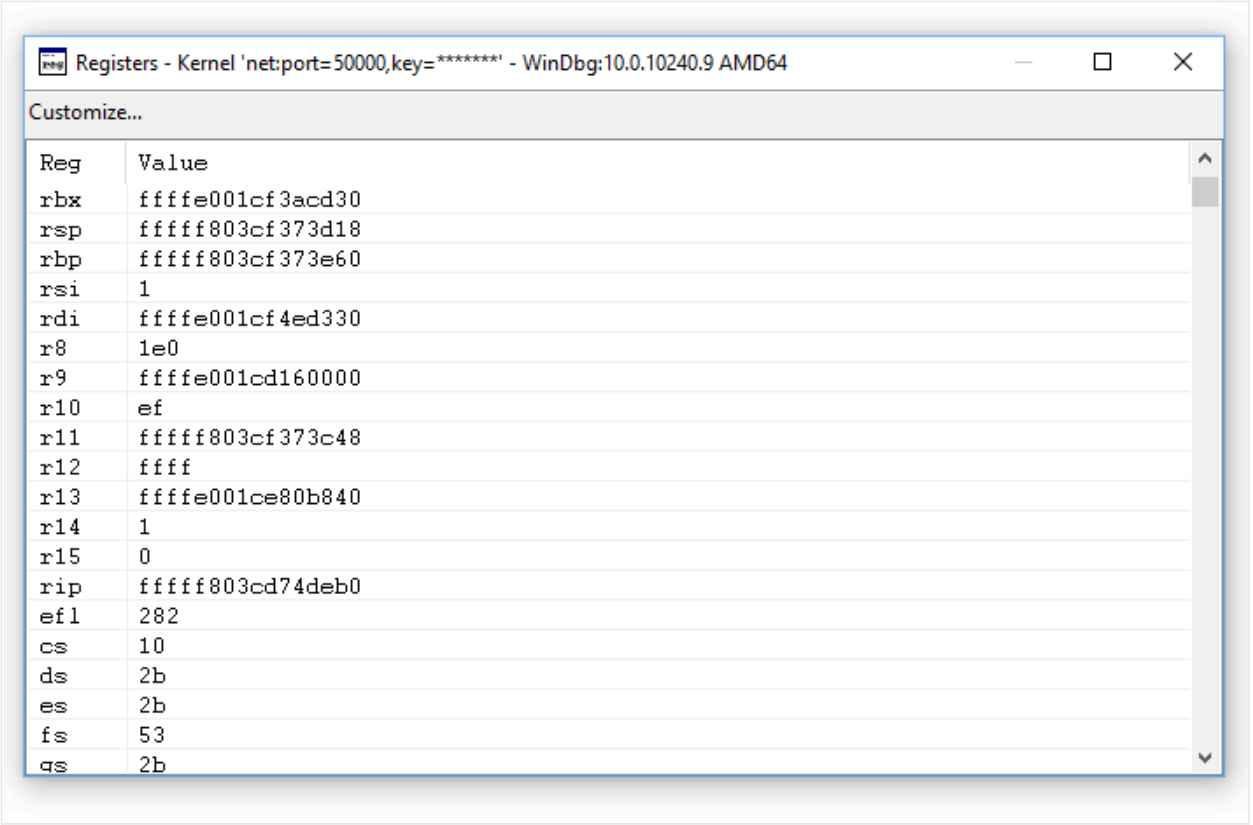
查看寄存器

使用 **r (Registers)** 命令显示当前处理器上当前线程的寄存器内容。

dbgcmd

```
0: kd> r
rax=000000000000c301 rbx=ffffe00173eed880 rcx=0000000000000001
rdx=000000d800000000 rsi=ffffe00173eed8e0 rdi=ffffe00173eed8f0
rip=fffff803bb757020 rsp=ffffd001f01f8988 rbp=ffffe00173f0b620
 r8=000000000000003e  r9=ffffe00167a4a000 r10=000000000000001e
r11=ffffd001f01f88f8 r12=0000000000000000 r13=ffffd001f01efdc0
r14=0000000000000001 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b
efl=00000202
nt!DbgBreakPointWithStatus:
fffff803`bb757020 cc              int     3
```

另外，也可以通过选择**查看**>**寄存器**来显示寄存器的内容。

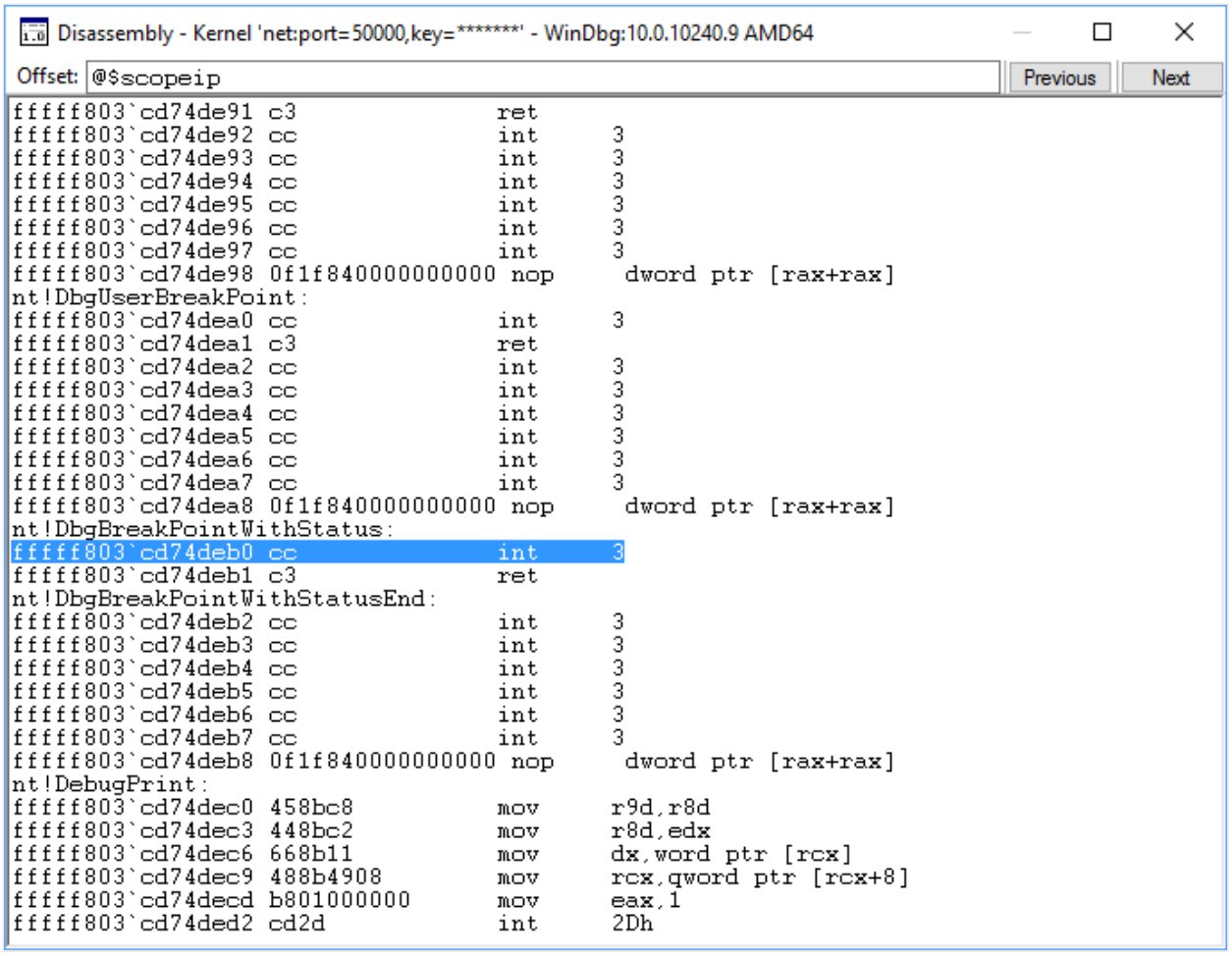


在单步执行程序集语言代码和其他情况下，查看寄存器的内容很有帮助。 有关详细信息，请参阅 **r（寄存器）**。

有关寄存器内容的信息，请参阅 x86 体系结构和 x64 体系结构。

反汇编

选择**视图**>**反汇编**，即可反汇编正在执行的代码，以便查看正在运行的汇编语言代码。



有关程序集语言反汇编的详细信息，请参阅带批注的 x86 反汇编和带批注的 x64 反汇编。

第 12 部分：使用内存

在第 12 部分中，将使用调试器命令来显示内存内容。

查看内存

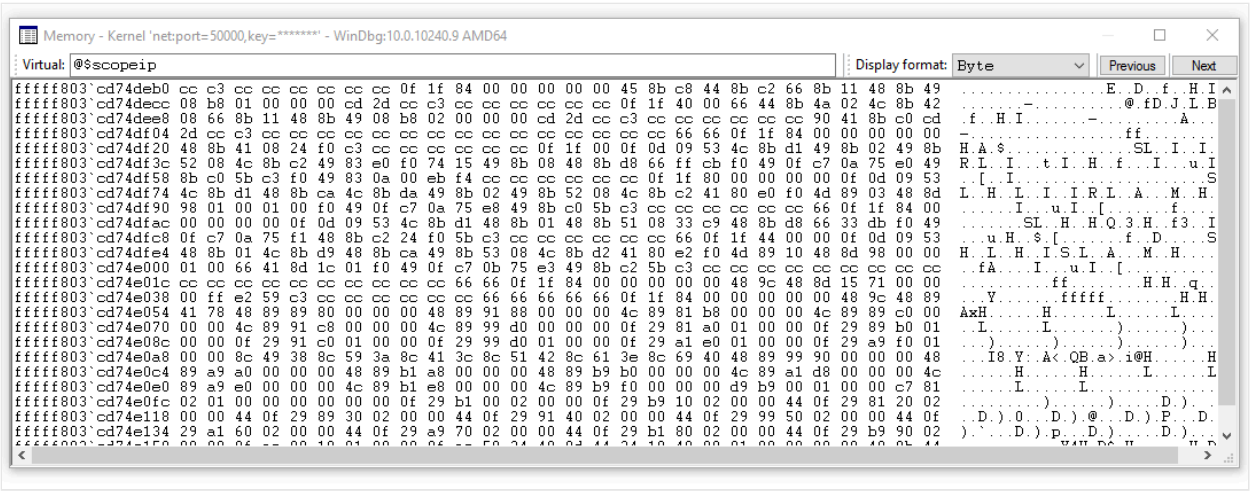
可能需要检查内存以识别问题，或者检查变量、指针等。 输入以下 **d*** **<address>** 命令之一即可显示内存。

展开表

db	以字节值和 ASCII 字符显示数据。
dd	以双宽单词（4 个字节）显示数据。
du	以 Unicode 字符显示数据。
dw	以单词值（2 个字节）和 ASCII 字符显示数据。

注意 如果尝试显示无效的地址，则其内容显示为问号 (?)。

或者，也可以通过选择**查看**>**内存**来查看内存。 使用“显示格式”下拉菜单更改内存的显示方式。



1. 要查看与音量控制关联的数据，请使用 bm 命令在 PropertyHandlerAudioEngineVolumeLevel 例程上设置断点。 在设置新的断点之前，我们将使用 bc * 来清除之前的所有断点。

```
dbgcmd
```

```
kd> bc *
```

2. 使用 bm 命令在 PropertyHandlerAudioEngineVolumeLevel 例程上设置一个断点。

```
dbgcmd
```

```
kd> bm tabletaudiosample!CMiniportWaveRT::SetDeviceChannelVolume
  1: fffff80f`02c3a4b0
@!"tabletaudiosample!CMiniportWaveRT::SetDeviceChannelVolume"
```

3. 列出断点，以便确认断点已正确设置。

```
dbgcmd
```

```
kd> bl
  1: fffff80f`02c3a4b0
@!"tabletaudiosample!CMiniportWaveRT::SetDeviceChannelVolume"
```

4. 使用 **g** 命令重新启动代码执行。

   在目标系统的系统托盘中调整音量。 这将导致断点触发。

```
dbgcmd
```

```
Breakpoint 1 hit
tabletaudiosample!CMiniportWaveRT::SetDeviceChannelVolume:
fffff80f`02c3a4b0 44894c2420      mov     dword ptr [rsp+20h],r9d
```

5. 使用**查看**>**局部**菜单项显示局部变量。 记录下 IVolume 变量的当前值。

6. 键入 **dt** 命令和变量名，即可显示示例代码中 IVolume 变量的数据类型和当前值。

dbgcmd

```
kd> dt lVolume
Local var @ 0xa011ea50 Type long
0n-6291456
```

7. 输入 SetDeviceChannelVolume 时断点被触发。

dbgcmd

```
STDMETHODIMP_(NTSTATUS) CMiniportWaveRT::SetDeviceChannelVolume(_In_
ULONG _ulNodeId, _In_ UINT32 _uiChannel, _In_  LONG  _Volume)
{
    NTSTATUS ntStatus = STATUS_INVALID_DEVICE_REQUEST;

    PAGED_CODE ();

    DPF_ENTER(("[CMiniportWaveRT::SetEndpointChannelVolume]"));
    IF_TRUE_ACTION_JUMP(_ulNodeId != KSNODE_WAVE_AUDIO_ENGINE, ntStatus
= STATUS_INVALID_DEVICE_REQUEST, Exit);

    // Snap the volume level to our range of steppings.
    LONG lVolume = VOLUME_NORMALIZE_IN_RANGE(_Volume);

    ntStatus = SetChannelVolume(_uiChannel, lVolume);
Exit:
    return ntStatus;
}
```

8. 尝试使用 **dt（显示类型）** 命令显示 IVolume 内存位置的值。

dbgcmd

```
kd> dt dt lVolume
Local var @ 0xffffb780b7eee664 Type long
0n0
```

因为变量尚未定义，所以不包含信息。

9. 按 F10 向前运行到 SetDeviceChannelVolume 中的最后一行代码。

```
dbgcmd
```

```
    return ntStatus;
```

10. 使用 **dt（显示类型）**命令显示 IVolume 内存位置的值。

```
dbgcmd
```

```
kd> dt lVolume
Local var @ 0xffffb780b7eee664 Type long
0n-6291456
```

变量处于活动状态后，本例中显示的值为 6291456。

11. 还可以使用 **?（对表达式进行评估）**）命令来显示内存位置。

```
dbgcmd
```

```
kd> ? lVolume
Evaluate expression: -79711507126684 = ffffb780`b7eee664
```

12. 所示地址 *ffffb780`b7eee664* 是 IVolume 变量的地址。 使用 dd 命令显示该位置的内存内容。

```
dbgcmd
```

```
kd>  dd ffffb780`b7eee664
ffffb780`b7eee664  ffa00000 00000018 00000000 c52d7008
ffffb780`b7eee674  ffffc98e e0495756 fffff80e c52d7008
ffffb780`b7eee684  ffffc98e 00000000 fffff80e 00000000
ffffb780`b7eee694  ffffc98e ffa00000 ffffb780 b7eee710
ffffb780`b7eee6a4  ffffb780 00000000 00000000 c7477260
ffffb780`b7eee6b4  ffffc98e b7eee7a0 ffffb780 b7eee6f0
ffffb780`b7eee6c4  ffffb780 e04959ca fffff80e 00000000
ffffb780`b7eee6d4  00000000 00000028 00000000 00000002
```

13. 通过指定范围参数 L4，可以显示地址的前四个字节。

```
dbgcmd
```

```
kd> dd ffffb780`b7eee664 l4
ffffb780`b7eee664  ffa00000 00000018 00000000 c52d7008
```

14. 要查看显示的不同类型内存输出，请键入 **du**、**da** 和 **db** 命令。

```
dbgcmd
```

```
kd> du ffffb780`b7eee664
ffffb780`b7eee664  ""

kd> a ffffb780`b7eee664
ffffb780`b7eee664  ""

kd> db 0xffffae015ff97664
ffffae01`5ff97664  00 80 bc ff 18 00 00 00-00 00 00 00 08 50 e0 51
.............P.Q
ffffae01`5ff97674  00 c0 ff ff 56 57 da 56-0e f8 ff ff 08 50 e0 51
....VW.V.....P.Q
ffffae01`5ff97684  00 c0 ff ff 00 00 00 00-0e f8 ff ff 00 00 00 00
................
ffffae01`5ff97694  00 c0 ff ff aa 80 bc ff-01 ae ff ff 10 77 f9 5f
.............w._
ffffae01`5ff976a4  01 ae ff ff 40 00 00 00-00 e6 ff ff 10 dc 30 55
....@.........0U
ffffae01`5ff976b4  00 c0 ff ff a0 77 f9 5f-01 ae ff ff f0 76 f9 5f
.....w._.....v._
ffffae01`5ff976c4  01 ae ff ff ca 59 da 56-0e f8 ff ff 00 00 00 00
.....Y.V........
ffffae01`5ff976d4  00 00 00 00 28 00 00 00-00 00 00 00 02 00 00 00
....(...........
```

使用 df float 选项可将数据显示为单精度浮点数（4 个字节）。

dbgcmd

```
df ffffb780`b7eee664
ffffb780`b7eee664          -1.#QNAN   3.3631163e-044                0
-2775.002
ffffb780`b7eee674          -1.#QNAN  -5.8032637e+019         -1.#QNAN
-2775.002
ffffb780`b7eee684          -1.#QNAN                0         -1.#QNAN
0
ffffb780`b7eee694          -1.#QNAN         -1.#QNAN         -1.#QNAN
-2.8479408e-005
```

写入内存

与用于读取内存的命令类似，也可以使用 e* 命令更改内存内容。

展开表

命令	说明
ea	ASCII 字符串（非 NULL 结尾）
eu	Unicode 字符串（非 NULL 结尾

命令	说明
ew	Word 值（2 个字节）
eza	以 NULL 结尾的 ASCII 字符串
ezu	以 Null 值结束的 Unicode 字符串
eb	字节值
ed	双单词值（4 个字节）

下面的示例显示了如何覆盖内存。

1. 首先，找到示例代码中使用的 lVolume 的地址。

```
dbgcmd
```

```
kd> ? lVolume
Evaluate expression: -79711507126684 = ffffb780`b7eee664
```

2. 使用 **eb** 命令用新字符覆盖该内存地址。

```
dbgcmd
```

```
kd> eb 0xffffb780`b7eee664 11 11 11 11 11
```

3. 输入 **db** 命令，显示内存位置，确认字符已被覆盖。

```
dbgcmd
```

```
kd> db 0xffffb780`b7eee664
ffffb780`b7eee664  11 11 11 11 11 00 00 00-00 00 00 00 08 70 2d c5
.............p-.
ffffb780`b7eee674  8e c9 ff ff 56 57 49 e0-0e f8 ff ff 08 70 2d c5
....VWI......p-.
ffffb780`b7eee684  8e c9 ff ff 00 00 00 00-0e f8 ff ff 00 00 00 00
................
ffffb780`b7eee694  8e c9 ff ff 00 00 a0 ff-80 b7 ff ff 10 e7 ee b7
................
ffffb780`b7eee6a4  80 b7 ff ff 00 00 00 00-00 00 00 00 60 72 47 c7
............`rG.
ffffb780`b7eee6b4  8e c9 ff ff a0 e7 ee b7-80 b7 ff ff f0 e6 ee b7
................
ffffb780`b7eee6c4  80 b7 ff ff ca 59 49 e0-0e f8 ff ff 00 00 00 00
.....YI.........
ffffb780`b7eee6d4  00 00 00 00 28 00 00 00-00 00 00 00 02 00 00 00
....(...........
```

或者，可以在监视或局部窗口中修改内存的内容。 在监视窗口中，可能会看到与当前帧无关的变量。 如果不符合上下文，修改它们就没有意义。

第 13 部分：结束 WinDbg 会话

<- 在主机系统上

如果想保留调试器，但又想连接到目标计算机，请使用 `bc *` 清除所有断点，这样目标计算机就不会尝试连接到主机调试器。 然后使用 `g` 命令让目标计算机再次运行。

要结束调试会话，请在主机系统上中断调试器，输入 `qd`（退出并分离）命令，或者从菜单中选择**停止调试**。

dbgcmd

```
0: kd> qd
```

有关详细信息，请参阅调试参考文档中的在 WinDbg 中结束调试会话（经典版）。

第 14 部分：Windows 调试资源

有关 Windows 调试的其他信息。 请注意，其中一些书籍会在示例中使用 Windows Vista 等较旧版本的 Windows，但所讨论的概念适用于大多数版本的 Windows。

书籍

- 《高级 Windows 调试》，作者：Mario Hewardt 和 Daniel Pravat
- *《在 Windows 调试中：Windows® 调试和跟踪策略实用指南》*，作者 Tarik Soulami
- *《Windows 内部》*，作者 Pavel Yosifovich、Alex Ionescu、Mark Russinovich 和 David Solomon

视频

碎片整理工具秀 WinDbg 第 13-29 集：</shows/defrag-tools/>

培训供应商：

OSR - https://www.osr.com/

另请参阅

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

调试资源

项目 • 2023/04/14

使用适用于 Windows 的调试工具调试 Windows 系统上的驱动程序、应用程序和服务。 工具集中的核心调试引擎称为 Windows 调试器。 可以使用 WinDbg、CDB 和 NTSD) (传统接口，这些接口作为 WDK 的一部分包含在 Windows 调试工具中。

Windows 调试工具入门

- Windows 调试入门
- 自动设置 KDNET 网络内核调试
- 调试 Windows 驱动程序分步实验室 (回显内核模式)

安装适用于 Windows 的调试工具

- 下载并安装适用于 Windows 的调试工具

调试器How-Tos

- 调试Kernel-Mode驱动程序框架驱动程序
- 调试 WDF 驱动程序
- BCDEdit /dbgsettings
- 驱动程序调试工具
- 避免调试器搜索不需要的符号

视频 - 碎片整理工具

- 碎片整理工具秀 - WinDbg 剧集 13-29
- 涵盖 WinDbg 的碎片整理工具 #182 和碎片整理工具 #183。
- 碎片整理工具 #175 - 调试 涵盖 NDIS 调试的网络堆栈。

书籍

- 《高级 Windows 调试》，作者：Mario Hewardt 和 Daniel Pravat
- Windows 内部调试：Windows 中调试和跟踪策略的实用指南，作者：Tarik Soulami

- Windows Internals 由 Pavel Yosifovich、Alex Ionescu、Mark E. Russinovich 和 David A. Solomon

Windows 调试工具中包含的工具

项目 • 2024/08/31

除了调试引擎和调试环境外，Windows 调试工具还包括多个工具。 这些工具位于 Windows 调试工具的安装目录中。

工具和实用工具

展开表

名称	描述
Breakin	导致进程中发生用户模式中断。 如需帮助，打开命令提示符窗口，转到安装目录，然后输入 `breakin /?`。
DumpChk	验证内存转储文件。
GFlags	控制注册表项和其他设置。
终止	终止进程。
列表（文件列表实用工具）	如需帮助，打开命令提示符窗口，转到安装目录，然后输入 `list /?`。
记录器和 LogViewer	记录和显示程序的函数调用和其他操作。
PLMDebug	使用 Windows 调试器，调试在进程生命周期管理 (PLM) 下运行的 Windows 应用。 使用 PLMDebug，可以手动控制暂停、恢复和终止 Windows 应用。
TList	列出所有正在运行的进程。
UMDH	分析堆分配。
USBView	显示 USB 主机控制器和连接的设备。
USBView2	显示 USB 主机控制器和连接的设备。 如需帮助，打开命令提示符窗口，转到安装目录，然后输入 `USBView2 /?`。

符号和源工具

展开表

名称	描述
AgeStore	删除符号服务器或源服务器的下游存储中的旧条目。
DBH	显示有关符号文件内容的信息。
PDBCopy	从符号文件中删除专用符号信息，并控制公共符号包含在文件中。
SrcSrv	一种源服务器，可用于在调试时传递源文件。
SymChk	将可执行文件与符号文件进行比较，以验证是否有正确的符号可用。
SymProxy	在网络上创建一个所有调试器都可以指向的 HTTP 符号服务器。 此方法的优点是使用单个符号路径指向多个符号服务器（内部和外部），处理所有身份验证，并通过符号缓存提高性能。 *Symproxy.dll* 位于安装目录中的 *SymProxy* 文件夹中。
SymSrv	一种符号服务器，调试器可使用它连接到符号存储区。 有关使用符号服务器的信息，请参阅 Microsoft 公共符号。
SymStore	创建符号存储。 请参阅使用 SymStore。

远程和代理调试器工具

展开表

名称	描述
DbgRpc	显示 Microsoft 远程过程调用 (RPC) 状态信息。 请参阅 RPC 调试和使用 DbgRpc 工具。
DbgSrv	用于远程调试的进程服务器。 请参阅进程服务器（用户模式）。
DbEngPrx	用于远程调试的中继器（小型代理服务器）。 请参阅中继器。
KDbgCtrl（内核调试控件）	控制和配置内核调试连接。 请参阅使用 KDbgCtrl。
KdSrv	用于远程调试的 KD 连接服务器。 请参阅 KD 连接服务器（内核模式）。
RTList（远程任务列表查看器）	通过 DbgSrv 进程服务器列出正在运行的进程。 如需帮助，打开命令提示符窗口，转到安装目录，然后输入 `rtlist /?`。
远程	远程控制任何控制台程序，包括 KD、CDB 和 NTSD。 请参阅使用 Remote.exe 进行远程调试。

安装目录

调试工具的 64 位操作系统安装的默认安装目录是 `C:\Program Files (x86)\Windows Kits\10\Debuggers\`。 如果您使用的是 32 位操作系统，可以在 `C:\Program Files` 下找到 Windows 工具包文件夹。 若要确定应该使用 32 位还是 64 位工具，请参阅选择 32 位或 64 位调试工具。

相关主题

与 Windows 调试工具相关的工具

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

DumpChk

项目 • 2023/12/21

DumpChk (Microsoft 故障转储文件检查器工具) 是一个对故障转储文件执行快速分析的程序。 此工具使你能够查看有关转储文件包含的内容的摘要信息。 可以使用 **DumpChk** 查找已损坏且无法由调试器打开的转储文件。

在何处获取 DumpChk

DumpChk.exe 包含在 Windows 调试工具中。

DumpChk 命令行选项

dbgcmd

```
DumpChk [-y SymbolPath] DumpFile
```

参数

-y SymbolPath
SymbolPath 指定 **DumpChk** 搜索符号的位置。 某些转储文件可能需要符号信息。 它还可以通过允许解析符号名称来帮助改进转储文件中显示的信息。

DumpFile
DumpFile 指定要分析的故障转储文件。 它可能包括绝对目录路径或相对目录路径或通用命名约定 (UNC) 路径。 如果 *DumpFile* 包含空格，则必须用引号括起来。

如何使用 DumpChk

以下示例显示了损坏的转储文件。 末尾 `DebugClient cannot open DumpFile` 显示的错误指示必须发生了某种损坏。

控制台

```
C:\Debuggers> dumpchk C:\mydir\dumpfile2.dmp

Loading dump file C:\mydir\dumpfile2.dmp

Microsoft (R) Windows Debugger Version 6.9.0003.113 X86
Copyright (C) Microsoft. All rights reserved.
```

```
Loading Dump File [C:\mydir\dumpfile2.dmp]
Could not match Dump File signature - invalid file format
Could not open dump file [C:\mydir\dumpfile2.dmp], HRESULT 0x80004002
    "No such interface supported"
**** DebugClient cannot open DumpFile - error 80004002
```

`DebugClient cannot open DumpFile`末尾的错误消息显示无法打开转储文件。 如果转储文件未损坏，则此显示以单词 `Finished dump check`结尾。

可能会列出其他错误，其中一些错误是良性的。 例如，以下错误消息不表示问题：

dbgcmd

```
error 3 InitTypeRead( nt!_PEB at 7ffd5000)
```

以下示例显示 **DumpChk** 在正常的用户模式小型转储上运行。 显示内容从转储文件的总体摘要开始，然后提供有关转储文件中包含的数据的详细信息：

控制台

```
C:\Debuggers> dumpchk C:\mydir\dumpfile1.dmp

Loading dump file C:\mydir\dumpfile1.dmp

Microsoft (R) Windows Debugger Version 6.9.0003.113 X86
Copyright (C) Microsoft. All rights reserved.


Loading Dump File [C:\mydir\dumpfile1.dmp]
User Mini Dump File with Full Memory: Only application data is available

Symbol search path is: srv*C:\CODE\LocalStore*\\symbols\symbols
Executable search path is:
Windows Vista Version 6000 MP (2 procs) Free x86 compatible
Product: WinNt, suite: SingleUserTS
Debug session time: Tue Jun 17 02:28:23.000 2008 (GMT-7)
System Uptime: 0 days 15:43:52.861
Process Uptime: 0 days 0:00:26.000
...
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.

----- User Mini Dump Analysis

MINIDUMP_HEADER:
Version         A793 (6903)
NumberOfStreams 12
Flags           1826
                0002 MiniDumpWithFullMemory
                0004 MiniDumpWithHandleData
```

```
                0020 MiniDumpWithUnloadedModules
                0800 MiniDumpWithFullMemoryInfo
                1000 MiniDumpWithThreadInfo

Streams:
Stream 0: type ThreadListStream (3), size 00000064, RVA 000001BC
  2 threads
  RVA 000001C0, ID 1738, Teb:000000007FFDF000
  RVA 000001F0, ID 1340, Teb:000000007FFDE000
Stream 1: type ThreadInfoListStream (17), size 0000008C, RVA 00000220
  RVA 0000022C, ID 1738
  RVA 0000026C, ID 1340
Stream 2: type ModuleListStream (4), size 00000148, RVA 000002AC
  3 modules
  RVA 000002B0, 00400000 - 00438000: 'C:\CODE\TimeTest\Debug\TimeTest.exe'
  RVA 0000031C, 779c0000 - 77ade000: 'C:\Windows\System32\ntdll.dll'
  RVA 00000388, 76830000 - 76908000: 'C:\Windows\System32\kernel32.dll'
Stream 3: type Memory64ListStream (9), size 00000290, RVA 00001D89
  40 memory ranges
  RVA 0x2019 BaseRva
  range#    RVA      Address      Size
       0 00002019    00010000   00010000
       1 00012019    00020000   00005000
       2 00017019    0012e000   00002000

 (additional stream data deleted)

Stream 9: type UnusedStream (0), size 00000000, RVA 00000000
Stream 10: type UnusedStream (0), size 00000000, RVA 00000000
Stream 11: type UnusedStream (0), size 00000000, RVA 00000000


Windows Vista Version 6000 MP (2 procs) Free x86 compatible
Product: WinNt, suite: SingleUserTS
kernel32.dll version: 6.0.6000.16386 (vista_rtm.061101-2205)
Debug session time: Tue Jun 17 02:28:23.000 2008 (GMT-7)
System Uptime: 0 days 15:43:52.861
Process Uptime: 0 days 0:00:26.000
  Kernel time: 0 days 0:00:00.000
  User time: 0 days 0:00:00.000
PEB at 7ffd9000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
    ImageBaseAddress:         00400000
    Ldr                       77a85d00
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 002c1e30 . 002c2148
    Ldr.InLoadOrderModuleList:           002c1da0 . 002c2138
    Ldr.InMemoryOrderModuleList:         002c1da8 . 002c2140
            Base TimeStamp                     Module
          400000 47959d85 Jan 21 23:38:45 2008
C:\CODE\TimeTest\Debug\TimeTest.exe
        779c0000 4549bdc9 Nov 02 02:43:37 2006 C:\Windows\system32\ntdll.dll
        76830000 4549bd80 Nov 02 02:42:24 2006
```

```
C:\Windows\system32\kernel32.dll
    SubSystemData:     00000000
    ProcessHeap:       002c0000
    ProcessParameters: 002c14c0
    WindowTitle:  'C:\CODE\TimeTest\Debug\TimeTest.exe'
    ImageFile:    'C:\CODE\TimeTest\Debug\TimeTest.exe'
    CommandLine:  '\CODE\TimeTest\Debug\TimeTest.exe'
    DllPath:
'C:\CODE\TimeTest\Debug;C:\Windows\system32;C:\Windows\system;
    Environment:  002c0808
        =C:=C:\CODE
        =ExitCode=00000000
        ALLUSERSPROFILE=C:\ProgramData
        AVENGINE=C:\PROGRA~1\CA\SHARED~1\SCANEN~1
        CommonProgramFiles=C:\Program Files\Common Files
        COMPUTERNAME=EMNET
        ComSpec=C:\Windows\system32\cmd.exe
        configsetroot=C:\Windows\ConfigSetRoot
        FP_NO_HOST_CHECK=NO
        HOMEDRIVE=C:
        NUMBER_OF_PROCESSORS=2
        OS=Windows_NT
        Path=C:\DTFW\200804~2.113\winext\arcade;C:\Windows\system32
        PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
        PROCESSOR_ARCHITECTURE=x86
        PROCESSOR_IDENTIFIER=x86 Family 6 Model 15 Stepping 13, GenuineIntel
        PROCESSOR_LEVEL=6
        PROCESSOR_REVISION=0f0d
        ProgramData=C:\ProgramData
        ProgramFiles=C:\Program Files
        PROMPT=$P$G
        PUBLIC=C:\Users\Public
        RoxioCentral=C:\Program Files\Common Files\Roxio Shared\9.0\Roxio
Central33\
        SESSIONNAME=Console
        SystemDrive=C:
        SystemRoot=C:\Windows
        USERDNSDOMAIN=NORTHSIDE.COMPANY.COM
        USERDOMAIN=NORTHSIDE
        USERNAME=myname
        USERPROFILE=C:\Users\myname
        WINDBG_DIR=C:\DTFW\200804~2.113
        windir=C:\Windows
        WINLAYTEST=200804~2.113
        _NT_SOURCE_PATH=C:\mysources
        _NT_SYMBOL_PATH=C:\mysymbols
Finished dump check
```

输出首先标识转储文件的特征。 在这种情况下，是包含完整内存信息 **的用户模式** 小型转储，包括应用程序数据，但不包括操作系统数据。 **DumpChk** 使用的符号路径紧随其后，然后是转储文件内容的摘要。

由于此显示以单词 `Finished dump check` 结尾，因此转储文件可能未损坏，可由调试器打开。 但是，文件中可能仍存在更微妙的损坏形式。

另请参阅

Windows 调试工具中包含的工具

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

Windows driver documentation 反馈

Windows driver documentation 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

GFlags

项目 • 2024/08/31

GFlags（全局标志编辑器）启用或禁用高级调试、诊断和故障排除功能。 它最常用于打开其他工具跟踪、计数和记录的指示器。

在何处获取 GFlags

Windows 10 调试工具 (WinDbg) 中包括 **GFlags.exe**。

安装调试工具后，默认情况下会在以下目录中安装 64 位版本的 **gflags.exe**。

```
控制台

C:\Program Files (x86)\Windows Kits\10\Debuggers\x64
```

如果运行的是 32 位版本的 Windows，请使用位于此处的 32 位版本的 **gflags.exe**：

```
控制台

C:\Program Files (x86)\Windows Kits\10\Debuggers\x86
```

GFlags 概述

驱动程序开发人员和测试人员通常使用 **GFlags** 直接或通过在测试脚本中包含 GFlags 命令来打开调试、日志记录和测试功能。 页堆验证功能可帮助识别**内核模式**驱动程序中的内存泄漏和缓冲区错误。

GFlags 同时具有对话框和命令行接口。 大多数功能都可以从两个界面获得，但有些功能只能从其中一个界面访问。 有关详细信息，请参阅 GFlags 详细信息。

功能

- 页堆验证。 GFlags 包括 `PageHeap` (pageheap.exe) 的功能，这是一个支持堆分配监视的工具。
- *特殊池*功能无需重新启动。 可以启用、禁用和配置特殊池功能，而无需重启计算机（“重启”）。 有关详细信息，请参阅特殊池。
- 对象引用跟踪。 标志允许在内核中跟踪对象引用和对象取消引用。 此功能检测对象引用计数何时递减过多或即使不再使用对象也不递减。

- GFlags 对话框包含选项卡式页面，便于导航。

要求

若要使用大多数 GFlags 功能，包括在注册表或内核模式下设置标志，或启用页堆验证，您必须是计算机上管理员组的成员。

本部分包括：

GFlags 概述

GFlags 详细信息

GFlags 命令

GFlags 标志表

GFlags 和 PageHeap

全局标志对话框

GFlags 示例

全局标志参考

> ⓘ **备注**
>
> 不正确使用此工具会降低系统性能或阻止 Windows 启动，需要你重新安装 Windows。

> ⓘ **重要**
>
> Windows 上已永久启用池标记。 **全局标志**对话框中的**启用池标记**复选框灰显，用于启用或禁用池标记的命令将失败。

另请参阅

GFlags 示例

全局标志参考

Windows 调试工具中包含的工具

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

GFlags 概述

项目 • 2023/10/04

GFlags (gflags.exe) 全局标志编辑器启用和禁用高级内部系统诊断和故障排除功能。 可以从命令提示符窗口运行 GFlags，也可以使用其图形用户界面对话框。

有关如何安装和查找 gflags.exe 的信息，请参阅 GFlags。

使用 GFlags 激活以下功能：

Registry
为计算机上运行的所有进程设置系统范围的调试功能。 这些设置存储在 **GlobalFlag** 注册表项 (**HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\GlobalFlag**) 中。 它们在你重启 Windows 时生效，并一直有效，直到你更改它们并再次重启。

内核标志设置
设置此会话的调试功能。 这些设置会立即生效，但在 Windows 关闭时会丢失。 这些设置会影响此命令完成后启动的所有进程。

图像文件设置
为特定程序设置调试功能。 这些设置存储在每个程序的 GlobalFlag 注册表项中，(**HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ImageFileName*\GlobalFlag**) 。 它们在你重启程序时生效，并在你更改它们之前保持有效。

调试程序
指定特定程序始终在调试器中运行。 此设置存储在注册表中。 它立即生效，并在更改之前保持有效。 (此功能仅在“ **全局标志** ”对话框中可用。)

发射
使用指定的调试设置运行程序。 调试设置在程序停止之前有效。 (此功能仅在“ **全局标志** ”对话框中可用。)

特殊池
请求从特殊池填充具有指定池标记或指定大小的分配。 此功能可帮助你检测和识别内核池使用中的错误来源，例如写入超出分配的内存空间，或引用已释放的内存。

从 Windows Vista 开始，你可以启用、禁用和配置特殊池功能，(**内核特殊池标记**) 为不需要重启的内核标志设置，或作为需要重新启动的注册表设置。

页堆验证
为程序启用、禁用和配置页面堆验证。 启用后，页堆监视动态堆内存操作（包括分配和

释放操作），并在检测到堆错误时导致调试器中断。

无提示进程退出

为进程启用、禁用和配置无提示退出的监视和报告。 可以指定进程以无提示方式退出时发生的操作，包括通知、事件日志记录和转储文件的创建。 有关详细信息，请参阅 监视无提示进程退出。

GFlags 详细信息

项目 • 2023/11/11

GFlags 通过编辑 Windows 注册表和内部设置来启用和禁用系统功能。 本部分详细介绍了 GFlag 的操作，并包括最有效地使用 GFlag 的提示。

常规信息

- 若要显示 GFlags 对话框，请在命令行中键入 **gflags**（，）没有参数。
- GFlags 系统级注册表设置会立即显示在注册表中，但在重启系统之前不会生效。
- GFlags 映像文件注册表设置立即显示在注册表中，但在重启该过程之前不会生效。
- GFlags 对话框中的调试器和启动功能特定于程序。 一次只能在一个图像文件上设置它们。

标志详细信息

- 若要清除所有标志，请将标志设置为 -FFFFFFFF。 将标志设置为 0 会将 0 添加到当前标志值。
- 将映像文件的标志设置为 FFFFFFFF (0xFFFFFFFF) 时，Windows 会清除映像文件的所有标志，并删除映像文件注册表项中的 **GlobalFlag** 条目。 保留映像文件注册表项。

对话框和命令行

可以使用 GFlags 的方便对话框或从命令行运行 GFlags。 大多数功能在这两种形式中都可用，但以下例外。

仅对话框

- 发射。 使用指定的标志启动程序。
- 在调试器中运行程序。
- Windows Vista 之前的系统上的特殊池。 在 Windows Vista 和更高版本的 Windows 上，可以在命令行或 Gflags 对话框中配置特殊池功能。

仅限命令行

- (/tracedb) 设置用户模式堆栈跟踪数据库的大小。
- 设置页堆验证选项。

注册表信息

在会话之间保存的 GFlags 设置存储在注册表中。 可以使用注册表 API、Regedit 或 reg.exe 来查询或更改这些值。 下表列出了设置的类型及其在注册表中的存储位置。

设置的类型	注册表位置
系统范围的设置 (“注册表”)	HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager**GlobalFlag**
针对计算机的所有用户 (“映像文件”) 程序特定的设置。	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ImageFileName***GlobalFlag**
特定程序的无提示退出设置 (“静默进程退出”) 计算机所有用户。	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SilentProcessExit***ImageFileName***
计算机所有用户的图像文件的页面堆选项	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ImageFileName***PageHeapFlags**
用户模式堆栈跟踪数据库大小 (**tracedb**)	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ImageFileName***StackTraceDatabaseSizeInMb**
创建用户模式堆栈跟踪数据库 (ust，0x1000) 图像文件	Windows 将映像文件名添加到 USTEnabled 注册表项 (HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options**USTEnabled**) 的值。
在可能的情况下大型页加载映像	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ImageFileName***UseLargePages**。
特殊池 (内核特殊池标记)	HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management**PoolTag**
验证开始/验证结束	HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PoolTagOverruns。 “ **验证开始**” 选项将值设置为 0。 “ **验证结束**” 选项将值设置为 1。

设置的类型	注册表位置
映像文件的调试器	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ImageFileName***调试器**
对象引用跟踪	HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Kernel**ObTraceProcessName**、 **ObTracePermanent** 和 **ObTracePoolTags**

GFlags 命令概述

项目 • 2024/07/08

有关如何安装和查找gflags.exe的一般信息，请参阅 GFlags。

可以互换使用 GFlags 命令和 全局标志对话框 。

GFlags 命令用法

若要使用 GFlags，请在命令行中键入以下命令。

打开 GFlags 对话框：

控制台

```
gflags
```

若要在注册表中设置或清除全局标志，请执行以下操作：

控制台

```
gflags /r [{+ | -}Flag [{+ | -}Flag...]]
```

设置或清除当前会话的全局标志：

控制台

```
gflags /k [{+ | -}Flag [{+ | -}Flag...]]
```

若要为映像文件设置或清除全局标志，请执行以下操作：

控制台

```
gflags /i ImageFile [{+ | -}Flag [{+ | -}Flag...]]
gflags /i ImageFile /tracedb SizeInMB
```

设置或清除特殊池功能（Windows Vista 及更高版本）

控制台

```
gflags {/r | /k} {+ | -}spp {PoolTag | 0xSize}
```

启用或禁用对象引用跟踪功能（Windows Vista 及更高版本）

控制台

```
gflags {/ro | /ko} [/p] [/i ImageFile | /t PoolTag;[PoolTag...]]
```

控制台

```
gflags {/ro | /ko} /d
```

若要启用和配置页面堆验证，请执行以下操作：

控制台

```
gflags /p /enable ImageFile  [ /full [/backwards] | /random Probability |
/size SizeStart SizeEnd | /address AddressStart AddressEnd | /dlls DLL
[DLL...] ]
[/debug ["DebuggerCommand"] | /kdebug] [/unaligned] [/notraces] [/fault Rate
[TimeOut]] [/leaks] [/protect] [/no_sync] [/no_lock_checks]
```

若要禁用页面堆验证，请执行以下操作：

控制台

```
gflags /p [/disable ImageFile] [/?]
```

显示帮助：

控制台

```
gflags /?
```

参数

旗
指定表示调试功能的三字母缩写（*FlagAbbr*）或十六进制值（*FlagHex*）。 缩写和十六进制值列在 GFlags 标志表中。

使用以下标志格式之一：

展开表

格式	说明
{+ | -}*FlagAbbr*	设置 （+） 或清除 （-） 由标志缩写表示的标志。 如果没有加号（+）或减号（-），命令将不起作用。
[+ | -]*FlagHex*	添加 （+） 或减去 （-） 标志的十六进制值。 当其值包含在总和中时，将设置标志。 添加 （+） 是默认值。 输入表示单个标志的十六进制值（不含 0x），或输入多个标志的十六进制值之和。

ImageFile
指定可执行文件的名称，包括文件扩展名（例如，notepad.exe或mydll.dll）。

/r
注册表。 显示或更改存储在注册表中的系统范围的调试标志。 这些设置在重启 Windows 时生效，在更改这些设置之前保持有效。

如果没有其他参数， **gflags /r** 会在注册表中显示系统范围的标志。

/k
内核标志设置。 显示或更改此会话的系统范围的调试标志。 这些设置立即生效，但在 Windows 关闭时会丢失。 这些设置会影响此命令完成后启动的进程。

如果没有其他参数， **gflags /k** 将显示为当前会话设置的系统范围标志。

/我
图像文件设置。 显示或更改特定进程的调试标志。 这些设置存储在注册表中。 它们对于此过程的所有新实例都有效，在更改这些实例之前，它们仍然有效。

如果没有其他参数， **gflags /i** *ImageFile* 将显示为指定进程设置的标志。

/tracedb *SizeInMB*
设置进程的用户模式堆栈跟踪数据库的最大大小。 若要使用此参数， 必须为进程设置“创建用户模式堆栈跟踪数据库 ”（ust）标志。

SizeInMB 是一个整数，表示以十进制单位表示的兆字节数。 默认值为最小大小，8 MB;没有最大大小。 若要还原为默认大小，请将 SizeInMB *设置为* 0。

spp
（Windows Vista 及更高版本）。设置或清除 特殊池 功能。 有关示例，请参阅 示例 14：配置特殊池。

PoolTag
（Windows Vista 及更高版本）。指定特殊池功能的池标记。 仅用于 **spp** 标志。

输入 PoolTag *的四个字符模式*，例如 Tag1。 它可以包括 **?** （替换任何单个字符） 和 * （替换多个字符） 通配符。 例如，Fat* 或 Av？4。 池标记始终区分大小写。

0x*大小*
（Windows Vista 及更高版本）。指定特殊池功能的大小范围。 仅用于 **spp** 标志。 有关选择大小值的指南，请参阅特殊池中的“选择分配大小”。

/ro
在注册表中启用、禁用和显示 对象引用跟踪 设置。 若要使此设置生效，必须重启 Windows。

如果没有其他参数， **/ro** 会在注册表中显示对象引用跟踪设置。

若要启用对象引用跟踪，必须在命令中包含至少一个池标记（**/t** *PoolTag*）或一个映像文件（**/i** ImageFile）。 有关详细信息，请参阅 示例 15：使用对象引用跟踪。

下表列出了对 **/ro** 有效的子参数。

展开表

/t *PoolTags*	将跟踪限制为具有指定池标记的对象。 使用分号 （;） 分隔标记名称。 输入最多 16 个池标记。 输入 *PoolTags* 的四个字符模式，例如 Tag1。 如果指定多个池标记，Windows 会跟踪具有任何指定池标记的对象。 如果未指定任何池标记，Windows 将跟踪映像创建的所有对象。
/i *ImageFile*	将跟踪限制为使用指定映像文件的进程创建的对象。 只能使用 **/i** 参数指定一个图像文件。 输入图像文件名（如notepad.exe），最多包含 64 个字符。 “系统”和“空闲”不是有效的映像名称。 如果未指定映像文件，Windows 将跟踪具有指定池标记的所有对象。 如果同时指定映像文件（**/i**）和一个或多个池标记（**/t**），Windows 会跟踪具有由指定映像创建的任何指定池标记的对象。
/d	清除对象引用跟踪功能设置。 与 /ro **一起使用**时，它会清除注册表中的设置。

/p	永久。 在禁用对象引用跟踪或计算机关闭或重启之前，跟踪数据将保留。 默认情况下，当对象被销毁时，将放弃对象的跟踪数据。

/ko
启用、禁用和显示内核标志（运行时） 对象引用跟踪 设置。 对此设置的更改会立即生效，但在关闭或重启系统时会丢失。 有关详细信息，请参阅 示例 15：使用对象引用跟踪。

如果没有其他参数， **/ko** 将显示内核标志（运行时）对象引用跟踪设置。

若要启用对象引用跟踪，必须在命令中包含至少一个池标记（**/t** *PoolTag*）或一个映像文件（**/i** *ImageFile*）。

下表列出了对 /ko **有效的**子参数。

展开表

/t *PoolTags*	将跟踪限制为具有指定池标记的对象。 使用分号 （;） 分隔标记名称。 输入最多 16 个池标记。 输入 *PoolTags* 的四个字符模式，例如 Tag1。 如果指定多个池标记，Windows 会跟踪具有任何指定池标记的对象。 如果未指定任何池标记，Windows 将跟踪映像创建的所有对象。
/i *ImageFile*	将跟踪限制为使用指定映像文件的进程创建的对象。 只能使用 **/i** 参数指定一个图像文件。 如果未指定映像文件，Windows 将跟踪具有指定池标记的所有对象。 如果同时指定映像文件（/i）和一个或多个池标记（/t），Windows 会跟踪具有由指定映像创建的任何指定池标记的对象。
/d	清除对象引用跟踪功能设置。 与 /ro **一起使用**时，它会清除注册表中的设置。
/p	永久。 在禁用对象引用跟踪或计算机关闭或重启之前，跟踪数据将保留。 默认情况下，当对象被销毁时，将放弃对象的跟踪数据。

/p
设置进程的页堆验证选项。

如果没有其他参数， **gflags /p** 将显示启用页面堆验证的图像文件列表。

页堆验证监视动态堆内存操作，包括分配操作和释放操作，并在检测到堆错误时导致调试器中断。

/禁用 *ImageFile*
关闭指定图像文件的页堆验证（标准或完整）。

此命令相当于关闭 进程的"启用页面堆 "标志（**gflags /i** *ImageFile* **-hpa**）。 可以互换使用命令。

/使 *ImageFile*
为指定的图像文件启用页面堆验证。

默认情况下， **/enable** 参数会打开 *图像文件的标准* 页面堆验证。 若要为图像文件启用*整*页堆验证，请将 /full **参数添加到**命令，或使用带有 +hpa **标志的 /i** 参数。

/满
为进程启用完整页面堆验证。 完整页堆验证在每个分配结束时放置一个保留虚拟内存区域。

使用此参数等效于打开进程（gflags /i **ImageFile** ***+hpa）的"启用页面堆"标志（hpa）。***
可以互换使用命令。

/向后
将保留的虚拟内存区域放在分配的开头，而不是放在末尾。 因此，调试器捕获缓冲区开头的溢出，而不是缓冲区末尾的溢出。 仅对 **/full** 参数有效。

/随机 *概率*
根据指定的概率为每个分配选择完整或标准页堆验证。

Probability 是 0 到 100 之间的十进制整数，表示整页堆验证的概率。 概率为 100 与使用 **/full** 参数相同。 0 的概率与使用标准页堆验证相同。

/大小 *SizeStart SizeEnd*
为指定大小范围内的分配启用整页堆验证，并通过该过程为所有其他分配启用标准页堆验证。

SizeStart 和 *SizeEnd* 是十进制整数。 默认值是所有分配的标准页堆验证。

/地址 *AddressStart AddressEnd*
为指定地址范围内的返回地址位于运行时调用堆栈上时分配的内存启用完整页堆验证。 它通过该过程为所有其他分配启用标准页堆验证。

若要使用此功能，请指定一个范围，其中包括调用具有可疑分配的函数的所有函数的地址。 当发生可疑分配时，调用函数的地址将位于调用堆栈上。

AddressStart 和 *AddressEnd* 指定在分配堆栈跟踪中搜索的地址范围。 地址以十六进制格式指定，例如0xAABBCCDD。

在 Windows Server 2003 和更早的系统上， **/address** 参数仅在 *基于 x*86 的计算机上有效。 在 Windows Vista 上，它在所有受支持的体系结构上都有效。

/dlls *DLL[， DLL*...]
为指定的 DLL 请求的分配启用完整页堆验证，并为进程进行所有其他分配的标准页堆验证。

DLL 是二进制文件的名称，包括其文件扩展名。 指定的文件必须是执行过程中进程加载的函数库。

/调试
自动启动由 /enable **参数在调试器下指定的**进程。

默认情况下，此参数将 NTSD 调试器与命令行 **ntsd -g -G -x** 和启用了页堆，但可以使用 *DebuggerCommand 变量指定其他调试器和* 命令行。

有关 NTSD 的信息，请参阅 使用 CDB 和 NTSD 进行调试。

此选项对于难以从命令提示符启动的程序和其他进程启动的程序非常有用。

"*DebuggerCommand*"
指定调试器和发送到调试器的命令。 此带引号的字符串可以包含调试器完全限定的路径、调试器名称和调试器解释的命令参数。 需要引号。

如果命令包含调试器的路径，则路径不能包含任何其他引号。 如果出现其他引号，命令 shell （*cmd.exe*） 将错误解释命令。

/kdebug
使用命令行 **ntsd -g -G -x** 在 NTSD 调试器下自动启动 /enable **参数指定的**进程，启用页堆，并控制重定向到内核调试器 NTSD。

有关 NTSD 的信息，请参阅 使用 CDB 和 NTSD 进行调试。

/unaligned
在页面结束边界处对齐每个分配的末尾，即使这样做意味着起始地址未在 8 字节块上对齐。 默认情况下，堆管理器保证分配的起始地址在 8 字节块上对齐。

此选项用于检测逐个字节错误。 当此参数与 /full **参数一起使用**时，预留虚拟内存的区域将在分配的最后一个字节之后开始，当进程尝试读取或写入超出分配的一个字节时，就会立即发生错误。

/decommit
此选项不再有效。 它已被接受，但被忽略。

Windows 2000 中包括的 PageHeap 程序（pageheap.exe），通过在分配后放置不可访问的页面来实现全页堆验证。 在该工具中，**/decommit** 参数替换了不可访问页的保留虚拟内存区域。 在此版本的 GFlags 中，始终使用保留虚拟内存区域来实现整页堆验证。

/notraces
指定不保存运行时堆栈跟踪。

此选项可稍微提高性能，但会使调试更加困难。 此参数有效，但不建议使用此参数。

/故障
强制程序内存分配按指定速率在指定超时后失败。

此参数将堆分配错误插入正在测试的映像文件中（称为"故障注入"的做法），以便某些内存分配失败，如程序在内存不足的情况下运行时可能出现的情况。 此测试有助于检测处理分配失败的错误，例如无法释放资源。

展开表

费率	指定从 1（.01%） 到 10000（1000%） 的十进制整数，表示分配失败的概率。 默认值为 100（1%）。
TimeOut	确定程序开始和故障注入例程的开始之间的时间间隔。 *TimeOut* 以秒为单位。 默认值为 5（秒）。

/泄漏
检查进程结束时是否存在堆泄漏。

/leaks 参数禁用整页堆。 使用 /leaks 时，**将忽略修改 /full 参数的 /full 参数和参数，例如 /向后，GFlags 会执行标准页堆验证**和泄漏检查。

/保护
保护堆内部结构。 此测试用于检测随机堆损坏。 这会使执行速度明显变慢。

/no_sync
检查是否未同步的访问。 如果此参数检测到使用HEAP_NO_SERIALIZE标志创建的堆被不同的线程访问，则会导致中断。

请勿使用此标志调试包含自定义堆管理器的程序。 同步堆访问的函数会导致页面堆验证程序报告不存在的同步错误。

/no_lock_checks
禁用关键节验证程序。

/?
显示 GFlags 的帮助。 使用 **/p** 时， **/?** 显示 GFlags 中页面堆验证选项的帮助。

备注

在没有参数的情况下键入 **gflag** 将打开“ **全局标志** ”对话框。

键入 **不带其他参数的 gflags /p** 将显示启用了页面堆验证的程序列表。

若要清除所有标志，请将 *Flag* 设置为 **-FFFFFFFF。** （设置 *标志* 到 0 将零添加到当前标志值。它未清除所有标志。

将映像文件的标志*设置为* **FFFFFFFF** 时，Windows 会清除所有标志并删除映像文件的注册表子项中的 GlobalFlag 条目。 子项保持不变。

页面堆 /enable 操作的 /full、/random、**/size**、**/address** 和 **/dlls** 参数确定哪些分配受页面堆验证和使用验证方法的约束。 每个命令中只能使用这些参数之一。 默认值是进程的所有分配的标准页堆验证。 其余参数设置页面堆验证的选项。

GFlags 中的页面堆功能仅监视使用标准 Windows 堆管理器函数（**HeapAlloc**、GlobalAlloc、LocalAlloc、**malloc**、**new**、**new[]**或其相应解除分配函数）的堆内存分配，或使用调用标准堆管理器函数的自定义操作的堆内存分配。

若要确定是否为程序启用了完整页堆验证或标准页堆验证，请在命令行中键入 **gflags /p**。 在生成的显示中， **跟踪** 指示为程序启用了标准页堆验证，完整 **跟踪** 指示为程序启用了整页堆验证。

/enable 参数为注册表中的映像文件设置“启用页堆”标志。 但是，**/enable** 参数默认为图像文件启用*标准*堆验证，这与带有 +hpa **标志的 /i 参数不同**，该标志为图像文件启用*完整*堆验证。

标准 页堆验证会在分配结束时放置随机模式，并在释放堆块时检查模式。 *完整* 页堆验证在每个分配结束时放置一个保留虚拟内存区域。

完整页堆验证可以快速消耗系统内存。 若要为内存密集型进程启用全页堆验证，请使用 **/size** 或 **/dlls** 参数。

使用全局标志进行调试后，提交 **gflags /p /disable** 命令以关闭页面堆验证并删除关联的注册表项。 否则，调试器读取的条目将保留在注册表中。 不能为此任务使用 **gflags /i hpa** 命令，因为它关闭页面堆验证，但不删除注册表项。

默认情况下，在 Windows Vista 和更高版本的 Windows 上，特定于程序的设置（图像文件标志和页面堆验证设置）存储在当前用户帐户中。

此版本的 GFlags 包括 -v 选项，这些选项支持为 GFlags 开发的功能。 但是，这些功能尚未完成，因此，不会记录这些功能。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

GFlags 标志表

项目 • 2023/06/15

下表列出了 GFlags 更改的标志、每个标志的十六进制值和缩写，以及注册表的目标 (R，K 表示内核，I 用于映像文件)，其中标志有效。

有关每个标志的详细说明，请参阅 全局标志参考。

有关使用 GFlag 的信息，请参阅 GFlags 概述 和 GFlags 详细信息。

重要 池标记在 Windows Server 2003 及更高版本中永久启用。 在这些系统上，“**全局**标志”对话框中的“**启用池标记**检查”框灰显，用于启用或禁用池标记的命令将失败。

注意 每个标志的符号名称仅供参考。 由于符号名称发生更改，因此它们不是全局标志的可靠标识符。

说明	符号名称	十六进制值	缩写	目标
缓冲区 DbgPrint 输出	FLG_DISABLE_DBGPRINT	0x08000000	Ddp	R，K
创建内核模式堆栈跟踪数据库	FLG_KERNEL_STACK_TRACE_DB	0x2000	kst	R
创建用户模式堆栈跟踪数据库	FLG_USER_STACK_TRACE_DB	0x1000	ust	R，K，I
调试初始命令	FLG_DEBUG_INITIAL_COMMAND	0x04	Dic	R
调试 WinLogon	FLG_DEBUG_INITIAL_COMMAND_EX	0x04000000	dwl	R
释放时禁用堆合并	FLG_HEAP_DISABLE_COALESCING	0x00200000	dhc	R，K，I
禁用内核堆栈分页	FLG_DISABLE_PAGE_KERNEL_STACKS	0x080000	dps	R
禁用受保护的 DLL 验证	FLG_DISABLE_PROTDLLS	0x80000000	Dpd	R，K，I
禁用堆栈扩展	FLG_DISABLE_STACK_EXTENSION	0x010000	dse	I

说明	符号名称	十六进制值	缩写	目标
提前创建关键节事件	FLG_CRITSEC_EVENT_CREATION	0x10000000	Cse	R，K，I
启用应用程序验证工具	FLG_APPLICATION_VERIFIER	0x0100	vrf	R，K，I
启用错误句柄检测	FLG_ENABLE_HANDLE_EXCEPTIONS	0x40000000	bhd	R，K
启用关闭异常	FLG_ENABLE_CLOSE_EXCEPTIONS	0x400000	ece	R，K
启用 Win32 子系统调试	FLG_ENABLE_CSRDEBUG	0x020000	d32	R
启用异常日志记录	FLG_ENABLE_EXCEPTION_LOGGING	0x800000	鳗鱼	R，K
启用堆释放检查	FLG_HEAP_ENABLE_FREE_CHECK	0x20	Hfc	R，K，I
启用堆参数检查	FLG_HEAP_VALIDATE_PARAMETERS	0x40	Hpc	R，K，I
启用堆标记	FLG_HEAP_ENABLE_TAGGING	0x0800	htg	R，K，I
启用按 DLL 的堆标记	FLG_HEAP_ENABLE_TAG_BY_DLL	0x8000	htd	R，K，I
启用堆尾部检查	FLG_HEAP_ENABLE_TAIL_CHECK	0x10	htc	R，K，I
启用调用时进行堆验证	FLG_HEAP_VALIDATE_ALL	0x80	hvc	R，K，I
启用内核调试程序符号加载	FLG_ENABLE_KDEBUG_SYMBOL_LOAD	0x040000	ksl	R，K
启用对象句柄类型标记	FLG_ENABLE_HANDLE_TYPE_TAGGING	0x01000000	eot	R，K
启用页堆	FLG_HEAP_PAGE_ALLOCS	0x02000000	Hpa	R，K，I
仅) (Windows 2000 和 Windows XP 启用池标记	FLG_POOL_ENABLE_TAGGING	0x0400	ptg	R

说明	符号名称	十六进制值	缩写	目标
启用系统关键突破	FLG_ENABLE_SYSTEM_CRIT_BREAKS	0x100000	渣 打 银行	R、K、I
在可能的情况下大型页加载映像			石油气	I
维护每个类型的对象列表	FLG_MAINTAIN_OBJECT_TYPELIST	0x4000	otl	R
启用静默进程退出监视	FLG_MONITOR_SILENT_PROCESS_EXIT	0x200		R
对象引用跟踪 (Windows Vista 及更高版本的)				R、K
显示加载器快照	FLG_SHOW_LDR_SNAPS	0x02	Sls	R，K，I
特殊池			spp	R R、K (Windows Vista 及更高版本)
异常时停止	FLG_STOP_ON_EXCEPTION	0x01	国企	R，K，I
GUI 挂起时停止	FLG_STOP_ON_HUNG_GUI	0x08	shg	K
出现未经处理的用户模式异常时停止	FLG_STOP_ON_UNHANDLED_EXCEPTION	0x20000000	苏	R，K，I

GFlags 和 PageHeap

项目 • 2024/07/08

此版本的 GFlags 包括 PageHeap （pageheap.exe）的功能，该工具可在 Windows 中启用堆分配监视。 PageHeap 支持在每个分配边界保留内存的 Windows 功能，以检测访问超出分配的内存尝试。

GFlags 中的页面堆选项允许你选择 *标准堆验证，该验证*会在每个堆分配结束时写入填充模式，并检查分配释放时模式，或 *全页堆验证*，这将在每个分配结束时放置不可访问的页面，以便程序在访问超出分配的内存时立即停止。 由于完整堆验证对每个分配使用整页内存，因此其广泛使用可能会导致系统内存短缺。

- 若要为所有进程启用标准页堆验证，请使用 **gflags /r +hpa** 或 **gflags /k +hpa**。
- 若要为一个进程启用标准页堆验证，请使用 **gflags /p /enable** *ImageFileName*。
- 若要为一个进程启用整页堆验证，请使用 **gflags /i** *ImageFileName* **+hpa** 或 **gflags /p /enable** *ImageFileName* **/full**。

除 /k **之外**的所有页面堆设置都存储在注册表中，并且一直有效，直到更改它们。

请注意，仅当启动映像之前启用了 PageHeap 验证时，PageHeap 功能才处于活动状态。 对于长时间运行的进程（例如生产环境中的用于 IIS 的 W3WP），这意味着在重启进程之前不会开始验证。 同样，如果在进程运行时禁用 PageHeap，验证将继续，直到重新启动进程。 重复发出相同的 GFlags 命令，同时进程正在运行或不运行，没有其他效果。 因此，使用 GFlags 命令启用 PageHeap 是幂等的。

PageHeap 配置在进程终止和系统重启时也持续存在。 系统重启可用于在配置所需的 PageHeap 设置后重新初始化进程，但在启用该功能后不能用于禁用该功能。 必须显式禁用 PageHeap。

在解释 **GFlags 对话框中图像文件的"启用页面堆** "复选框时使用注意。 它指示为图像文件启用了页面堆验证，但它不指示它是完整还是标准页面堆验证。 如果选中该复选框的复选结果，则会为图像文件启用整页堆验证。 但是，如果使用命令行接口的检查结果，则检查可以表示为图像文件启用完整页堆验证或标准页堆验证。

若要确定是否为程序启用了完整页堆验证或标准页堆验证，请在命令行中键入 **gflags /p**。 在生成的显示中， **跟踪** 指示为程序启用了标准页堆验证，完整 **跟踪** 指示为程序启用了整页堆验证。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

监视静默进程退出

项目 • 2023/11/15

从 Windows 7 开始，可以使用 GFlags 中的“ **无提示进程退出** ”选项卡输入要监视无提示退出的进程的名称。

在此监视功能的上下文中，我们使用术语 *无提示退出* 表示受监视的进程以下列方式之一终止。

自我终止

受监视的进程通过调用 **ExitProcess** 来终止自身。

跨进程终止

第二个进程通过调用 **TerminateProcess** 来终止受监视的进程。

监视功能不会检测到在进程的最后一个线程退出时发生的正常进程终止。 监视功能不会检测内核模式代码启动的进程终止。

若要注册无提示退出监视的进程，请在 GFlags 中打开 “**无提示进程退出** ”选项卡。 输入进程名称作为 **图像** ，然后按 Tab 键。 选中 “**启用无提示进程退出监视** ”框，然后选择“ **应用**”。 这会在以下注册表项中设置FLG_MONITOR_SILENT_PROCESS_EXIT标志。

HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ProcessName*\GlobalFlag

有关此标志的详细信息，请参阅 启用无提示进程退出监视。

有关在 GFlags 中使用 **无提示进程退出** 选项卡的详细信息，请参阅 配置无提示进程退出监视。

在 GFlags 的“ **静默进程退出** ”选项卡中，可以配置监视的进程以无提示方式退出时将发生的操作。 可以配置通知、事件日志记录和转储文件的创建。 可以指定在检测到无提示退出时将启动的进程，也可以指定监视器将忽略的模块列表。 其中一些设置可用于全局和单个应用程序。 全局设置适用于注册无提示退出监视的所有进程。 应用程序设置适用于单个进程并替代全局设置。

全局设置存储在注册表中的以下项下。

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SilentProcessExit

应用程序设置存储在注册表中的以下项下。

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SilentProcessExit*ProcessName*

报告模式

报告模式设置可用作应用程序设置，但不能作为全局设置使用。 可以使用以下检查框来设置报告模式。

启动监视进程启用转储收集启用通知ReportingMode 注册表项是以下标志的按位 OR。

标志	值	含义
LAUNCH_MONITORPROCESS	0x1	检测到无提示退出时，（“监视进程”框中指定的 **监视进程**) 启动。
LOCAL_DUMP	0x2	检测到无提示退出时，会为受监视的进程创建转储文件。 在跨进程终止的情况下，还会为导致终止的进程创建转储文件。
通知	0x4	检测到无提示退出时，将显示弹出通知。

忽略自退出

“**忽略自退出**”设置可用作应用程序设置，但不可用作全局设置。 可以使用“**忽略自退出**检查框来指定是否忽略自退出。

IgnoreSelfExits 注册表项具有以下值之一。

值	含义
0x0	检测并响应自我终止和跨进程终止。
0x1	忽略自终止。 检测并响应跨进程终止。

监视进程

可以通过在“监视进程”文本框中输入进程名称以及命令行参数来指定 **监视进程** 。 可以在命令行中使用以下变量。

Varaible	含义
%e	正在退出进程的 ID。 这是以静默方式退出的受监视进程。

Varaible	含义
%i	启动进程的 ID。 对于自终止，这与退出进程相同。 对于跨进程终止，这是导致终止的进程 ID。
%t	启动线程的 ID。 这是导致终止的线程。
%c	传递给 **ExitThread** 或 **TerminateThread** 的状态代码。

例如， **监视进程的** 以下值指定在无提示退出时启动 WinDbg 并附加到退出进程。

```
windbg -p %e
```

Monitor Process 命令行存储在 **MonitorProcess** 注册表项中。

转储文件夹位置

可以使用“ **转储文件夹位置** ”文本框指定在检测到无提示退出时写入的转储文件的位置。

为 **转储文件夹位置** 输入的字符串存储在 **LocalDumpFolder** 注册表项中。

如果未指定转储文件夹位置，则转储文件将写入默认位置，即 %TEMP%\无提示进程退出。

转储文件夹大小

可以使用“ **转储文件夹大小** ”文本框指定可写入转储文件夹的最大转储文件数。 以十进制整数的形式输入此值。

为 **转储文件夹大小** 输入的值存储在 **MaxNumberOfDumpFiles** 注册表项中。

默认情况下，可以写入的转储文件限制为 10 个。

转储类型

可以使用“ **转储类型** ”下拉列表指定在检测到无提示退出时写入的转储文件类型 (Micro、Mini、Heap 或 Custom) 。

转储类型存储在 **DumpType** 注册表项中，这是 **MINIDUMP_TYPE** 枚举的成员的按位 OR。 此枚举在 dbghelp.h 中定义，该枚举包含在 Windows 调试工具包中。

例如，假设你选择了一个转储类型 **Micro**，并且你看到 **DumpType** 注册表项的值为 0x88。 值0x88是以下两个 **MINIDUMP_TYPE** 枚举值的按位 OR。

MiniDumpFilterModulePaths：0x00000080

MiniDumpFilterMemory：0x00000008

如果选择“**自定义**”转储类型，请在“**自定义转储类型**”框中输入自己的按位 OR MINIDUMP_TYPE枚举值。 以十进制整数的形式输入此值。

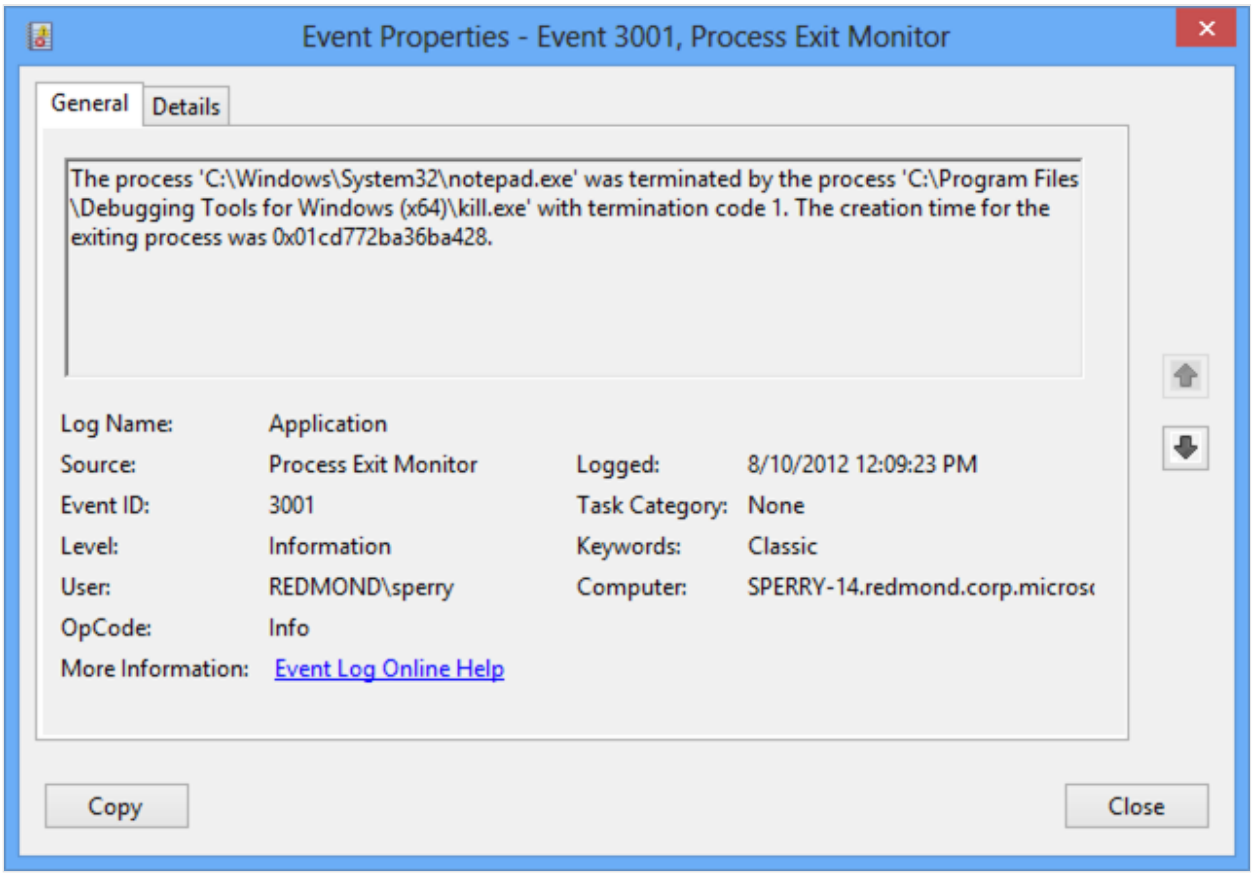
模块忽略列表

可以使用“ **模块忽略列表** ”框指定在检测到无提示退出时将忽略的模块列表。 如果受监视的进程由此列表中的某个模块终止，则忽略无提示退出。

在“ **模块忽略列表** ”框中输入的模块列表存储在 ModuleIgnoreList 注册表项中。

在 事件查看器 中读取进程退出报告

当监视的进程以无提示方式退出时，监视器会在 事件查看器 中创建一个条目。 若要打开事件查看器，请输入命令 eventvwr.msc。 导航到 “Windows **日志** > **应用程序**”。 查找具有进程退出监视器 **源** 的日志条目。



全局标志对话框

项目 • 2023/06/15

可以使用 " **全局标志** " 对话框从按名称列出所有标志的用户界面中设置和清除全局标志。 无需查找标志缩写或十六进制值。

此外，该对话框提供对命令行中不可用的以下功能的访问权限：

- **调试器** • 指定特定程序始终在调试器中运行。
- **Launch** • 使用指定的调试设置运行程序。
- **内核特殊池标记** • 配置特殊池功能。

本主题包括以下过程的说明：

打开对话框

设置和清除系统范围的标志

设置和清除内核标志

设置和清除映像文件标志

启动带标志的程序

在调试器中运行程序

配置特殊池

配置对象引用跟踪

池标记在 Windows Server 2003 及更高版本的 Windows 上永久启用。 在这些系统上，"**全局标志**"对话框中的"**启用池标记**检查"框灰显，用于启用或禁用池标记的命令将失败。

请谨慎解释"**全局标志**"对话框中图像文件的"**启用页面堆**检查"框。 此检查框指示为图像文件启用了页面堆验证，但不指示它是完整页堆验证还是标准页堆验证。 如果选择检查框导致检查，则会为映像文件启用整页堆验证。 但是，如果检查由于使用命令行接口而产生，则检查可以表示为映像文件启用完整页堆验证或标准页堆验证。

若要确定是否为程序启用完整页堆验证或标准页堆验证，请在命令行中键入 **gflags /p**。 在生成的显示中， **跟踪** 指示为程序启用了标准页堆验证， **完整跟踪** 指示为程序启用了整页堆验证。

打开对话框

项目 • 2023/06/15

打开“全局标志”对话框

- 双击 **gflags.exe** 图标，或在命令行中或在“ **运行** ”对话框中，键入 “**gflags** (不带参数) 。
- ○ OR
- 单击 “**开始**”，指向 “**所有程序**”，指向 “**Windows 调试工具**”， 然后单击“ **全局标志**”。

  在 Windows Vista 上，依次单击“ **开始**”、“ **所有程序**”、“ **Windows 调试工具**”、右键单击“ **全局标志** ”，然后单击“ **以管理员身份运行**”。 如果省略此步骤，Windows 将显示 **系统注册表 Gflags 错误：5。** “Gflags”对话框随即打开，但 Gflags 命令失败。

设置和清除系统范围的标志

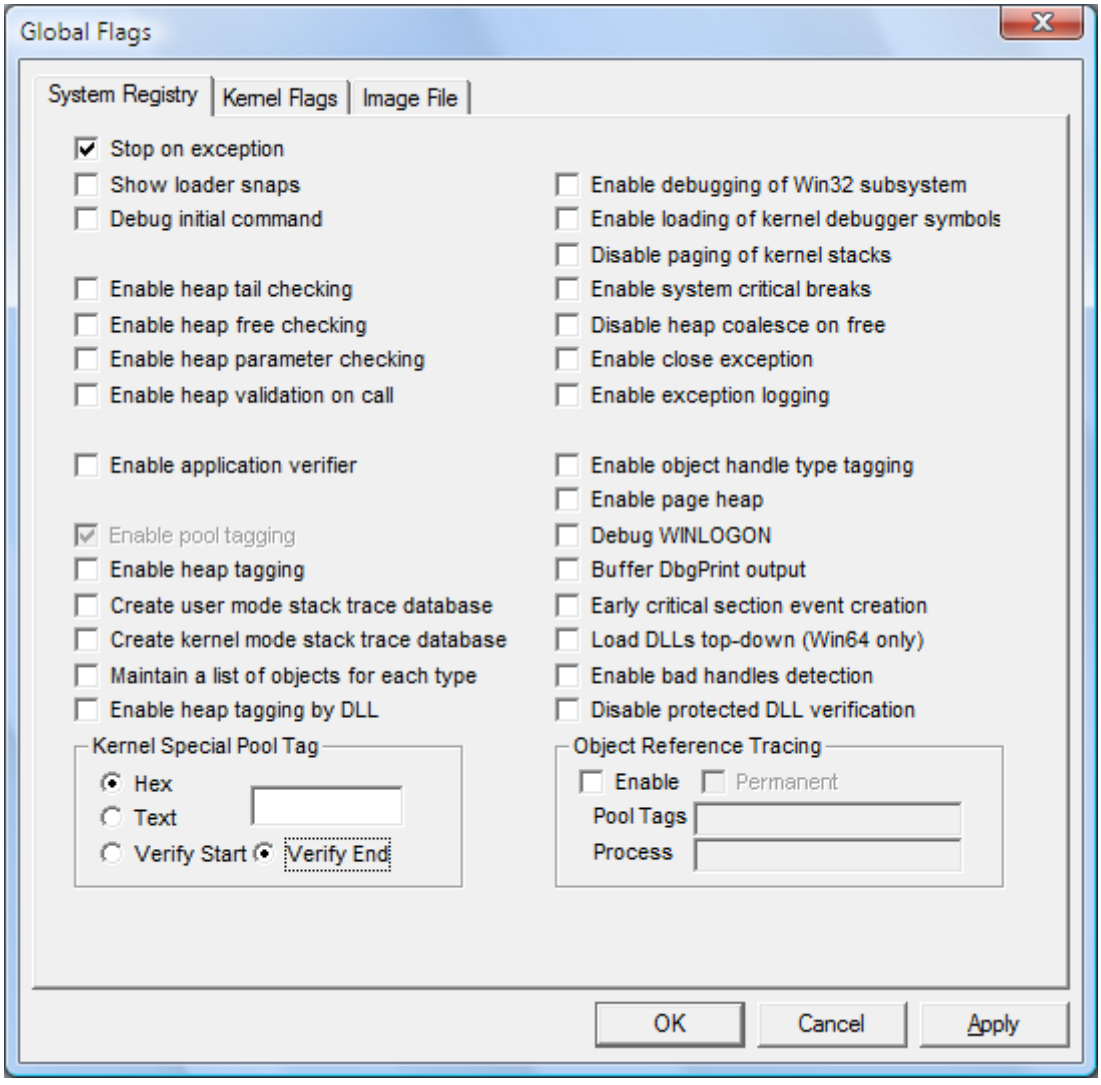
项目 • 2023/06/15

系统范围的注册表设置会影响在 Windows 上运行的所有进程。 它们保存在注册表中，并在更改它们之前保持有效。

设置和清除系统范围的注册表标志

1. 单击“ **系统注册表** ”选项卡。

   以下屏幕截图显示了 Windows Vista 中的“系统注册表”选项卡。



2. 通过选择或清除与标志关联的检查框来设置或清除标志。

3. 选择或清除所需的所有标志后，单击“ **应用**”。

4. 重启 Windows 以使更改生效。

设置和清除内核标志

项目 • 2023/06/15

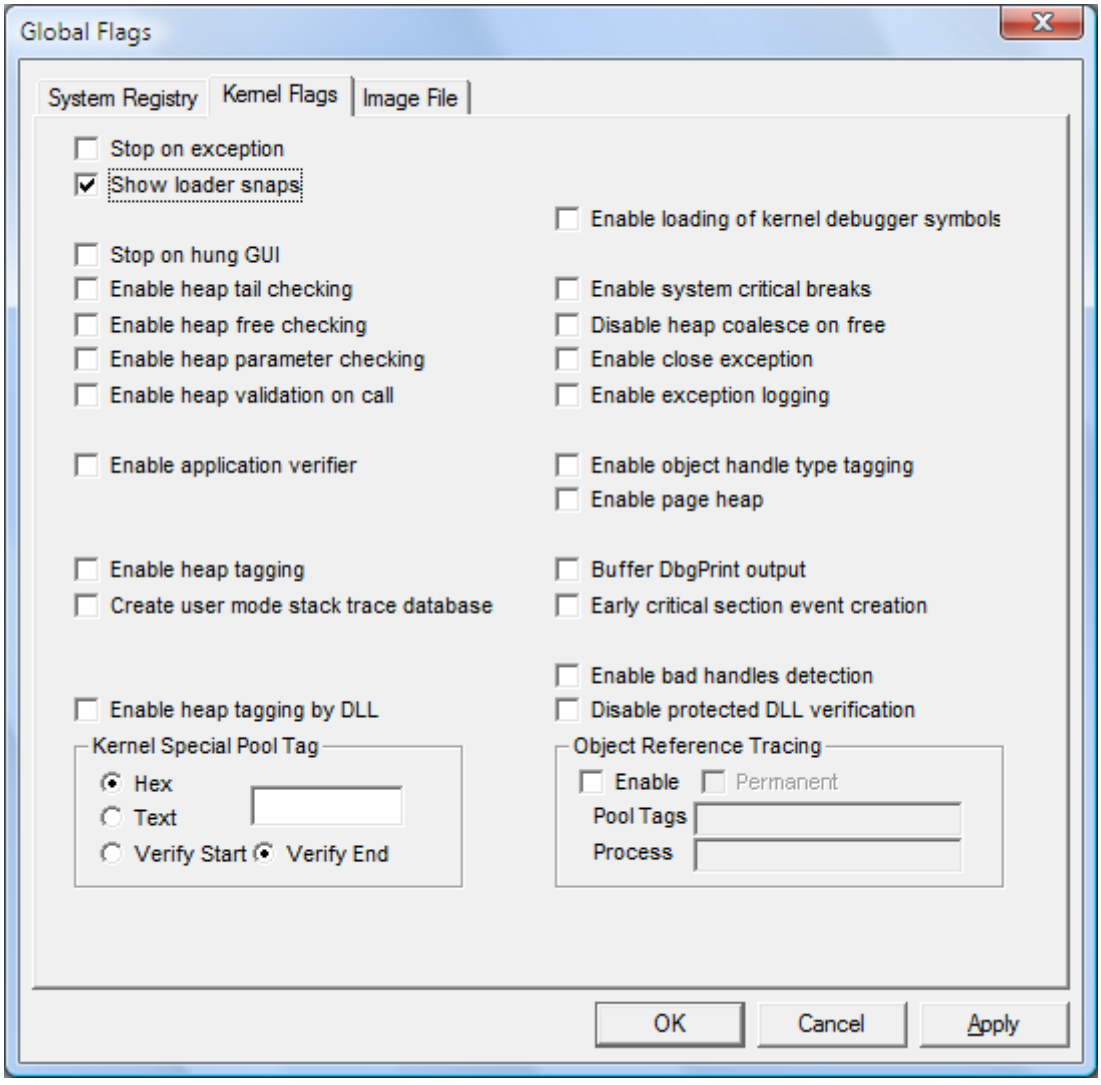
内核标志设置（也称为“运行时设置”）会影响整个系统。 它们无需重新启动即可立即生效，但如果关闭或重启系统，它们将丢失。

内核设置在运行时优先于注册表设置，但在关闭或重启系统时，内核标志设置将丢失，注册表设置将再次生效。

设置和清除内核标志

1. 单击“ **内核标志** ”选项卡。

   以下屏幕截图显示了 Windows Vista 中的“ **内核标志** ”选项卡。



2. 通过选择或清除与标志关联的检查框来设置或清除标志。

3. 选择或清除所需的所有标志后，单击“ **应用**”。

设置和清除映像文件标志

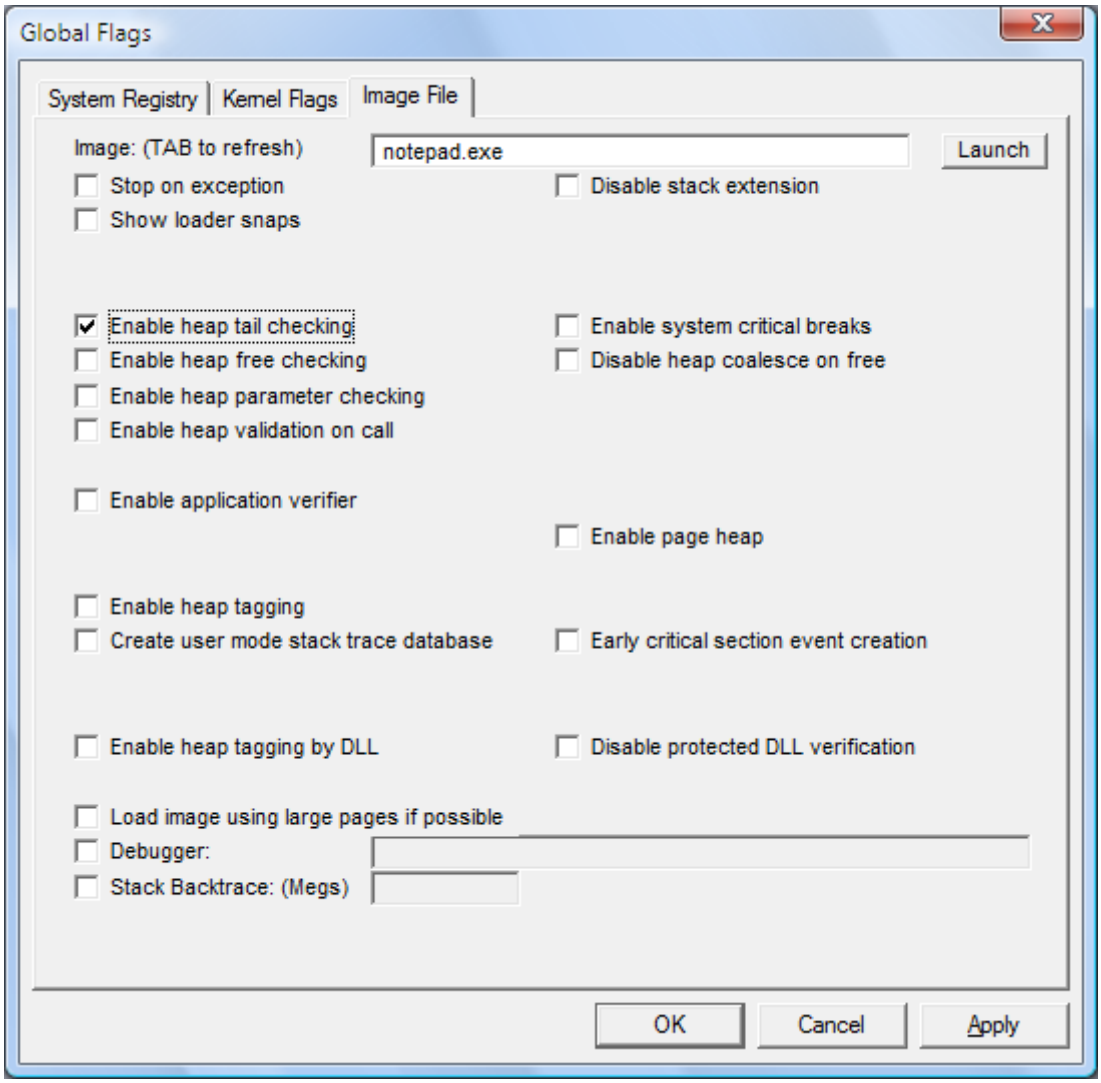
项目 • 2023/06/15

图像文件设置会影响在命令完成后启动的指定程序的实例。 它们保存在注册表中，并在更改它们之前保持有效。

设置和清除映像文件注册表标志

1. 单击“ **图像文件** ”选项卡。

   以下屏幕截图显示了 Windows Vista 中的“ **图像文件** ”选项卡。



2. 在“ **图像** ”框中，键入可执行文件或 DLL 的名称（包括文件扩展名），然后按 TAB 键。

   这会激活“**图像文件**”选项卡上的检查框。

3. 通过选择或清除与标志关联的检查框来设置或清除标志。

4. 选择或清除所需的所有标志后，单击“ **应用**”。

配置静默进程退出监视

项目 • 2023/11/15

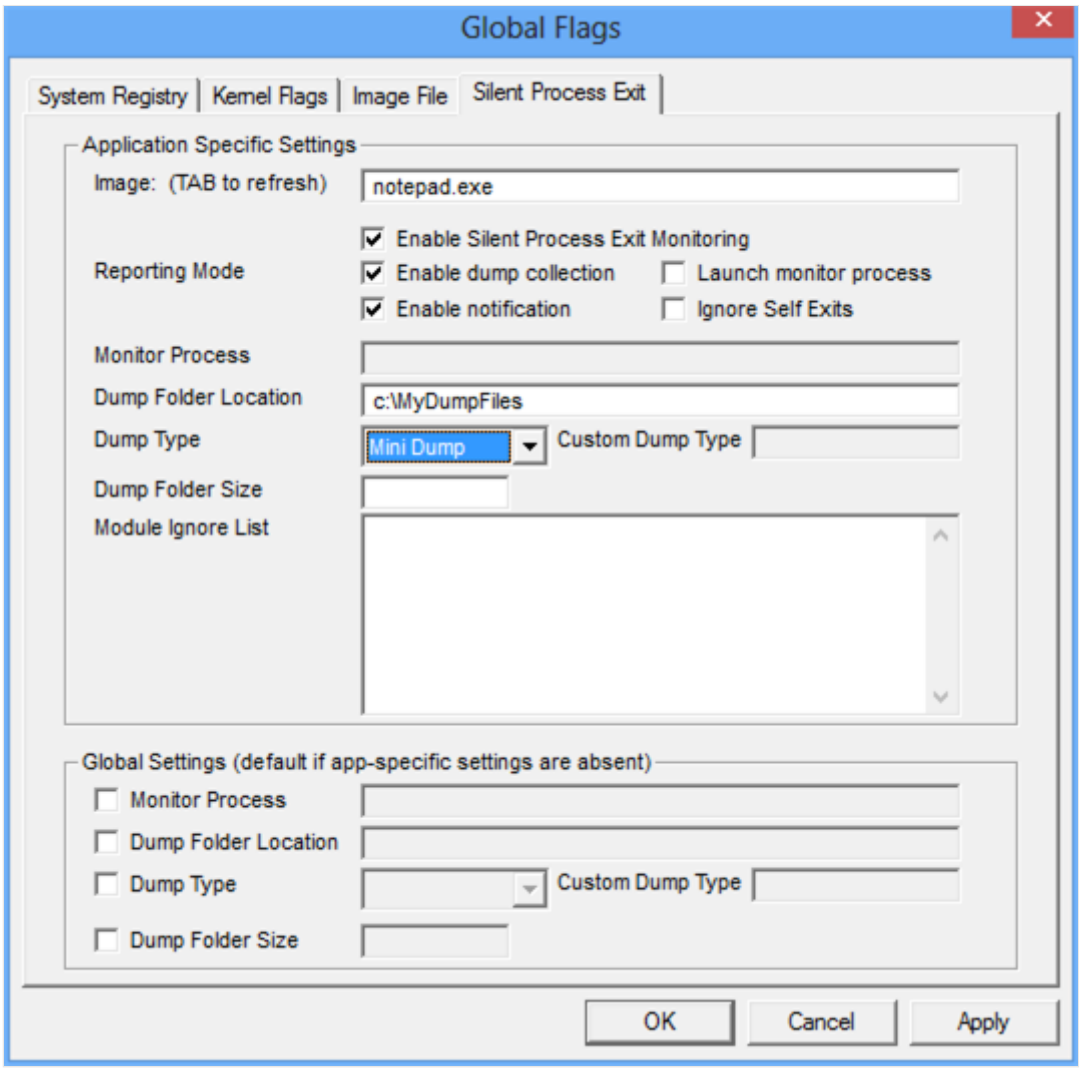
从 Windows 7 开始，可以使用“ **无提示进程退出** ”选项卡为进程启用和配置无提示退出的监视。

在 “**无提示进程退出** ”选项卡中指定的设置将保存在注册表中，并在进行更改之前保持有效。

启用和配置无提示进程退出监视

1. 单击“ **静默进程退出** ”选项卡。

   以下屏幕截图显示了 Windows 8 中的“**无提示进程退出**”选项卡。



2. 在“ **图像** ”框中，键入可执行文件的名称（包括文件扩展名），然后按 TAB 键。

   这会激活“**无提示进程退出**”选项卡上的检查框。

3. 通过选择或清除检查框并在文本框中输入值来指定首选项。

4. 指定所有首选项后，单击“ **应用**”。

另请参阅

监视静默进程退出

启用静默进程退出监视

启动带标志的程序

项目 • 2023/11/15

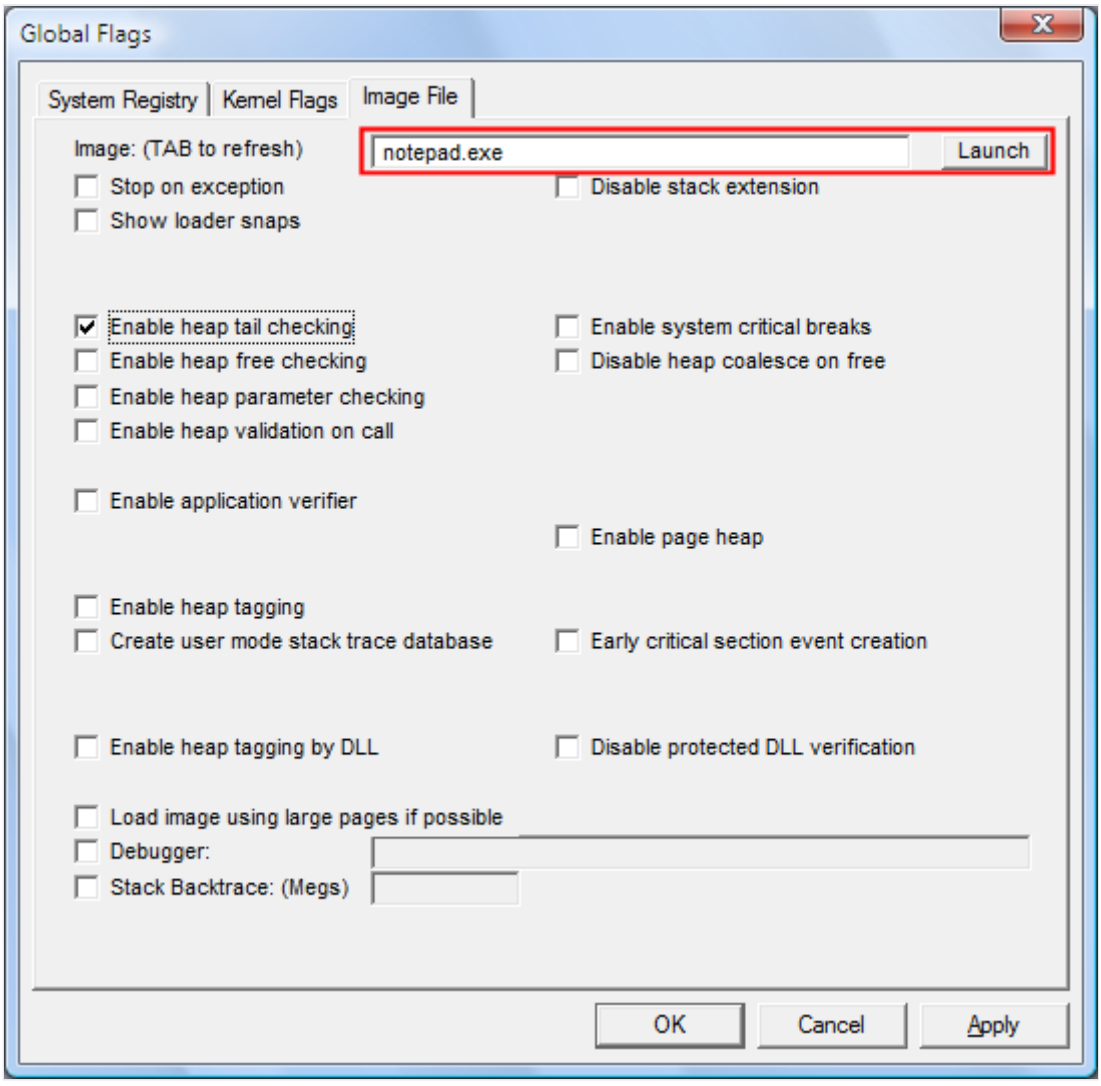
此功能使用指定的标志运行一次程序。 这些设置仅影响启动的程序实例。 它们不会保存在注册表中。

使用标志启动程序

1. 单击“ **图像文件** ”选项卡。
2. 在“ **映像** ”框中，键入可执行文件或 DLL 的名称（包括文件扩展名）以及程序的任何命令，然后按 TAB 键。

   这会激活“**启动**”按钮和“**图像文件**”选项卡上的“检查”框。
3. 通过选择或清除与标志关联的检查框来设置或清除标志。
4. 单击“ **启动** ”按钮。

   以下屏幕截图显示了 Windows Vista 中“**图像文件**”选项卡上的“**启动**”按钮。



注意 注册表中设置的标志不会影响启动的程序实例。 对话框中设置的标志用于启动的实例，即使它们不是图像文件标志。

在调试器中运行程序

项目 • 2023/06/14

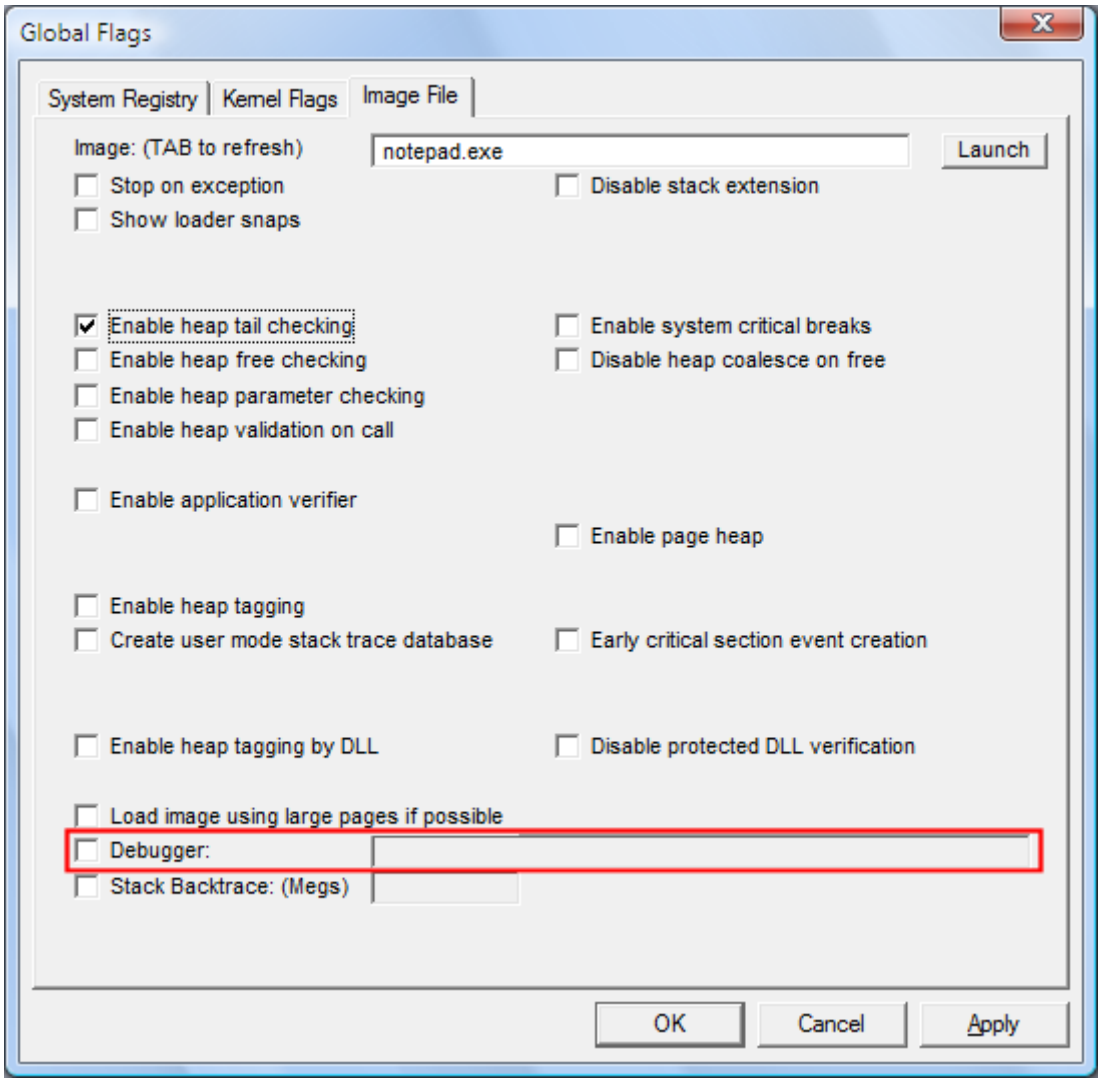
此功能配置程序，使其始终在具有指定选项的调试器中运行。 此设置保存在注册表中。 它会影响程序的所有新实例，并在你更改它之前保持有效。

在调试器中运行程序

1. 单击“ **图像文件** ”选项卡。
2. 在“ **图像** ”框中，键入可执行文件或 DLL 的名称（包括文件扩展名），然后按 TAB 键。

   这会激活“**图像文件**”选项卡上的检查框。
3. 单击“**调试器**检查”框将其选中。

   以下屏幕截图显示了 Windows Vista 中“**图像文件**”选项卡上的“**调试器**检查”框。



4. 在“ **调试器** ”框中，键入运行调试器的命令，包括路径 (可选) 以及调试器和参数的名称。 例如， **ntsd -d -g -G -x** 或 **c:\debuggers\cdb.exe -g -G**。

5. 单击“应用”。

配置特殊池

项目 • 2023/06/15

当使用指定的池标记分配内存或在指定大小范围内分配内存时，Gflags *特殊* 池功能指示 Windows 从保留的内存池请求内存分配。

有关此功能的详细信息，请参阅 特殊池。

在 Windows Vista 和更高版本的 Windows 中，可以将特殊池功能配置为系统范围的注册表设置或不需要重新启动的内核标志设置。 在早期版本的 Windows 中，特殊池仅作为注册表设置提供。

在 Windows Vista 和更高版本的 Windows 中，还可以使用命令行按池标记请求特殊池。 有关信息，请参阅 **GFlags 命令**。

本部分包括以下主题。

按池标记请求特殊池

按分配大小请求特殊池

取消针对特殊池的请求

检测超量运行和欠量运行

注意 使用 *驱动程序验证程序* 请求特定驱动程序分配的特殊池。 有关详细信息，请参阅 Windows 驱动程序工具包 (WDK) 的“驱动程序验证程序”部分中的“特殊池”主题。

按池标记请求特殊池

项目 • 2023/06/15

可以为使用指定池标记的所有分配请求特殊池。 系统上一次只能有一个池标记与内核特殊池请求相关联。

在 Windows Vista 和更高版本的 Windows 中，还可以使用命令行按池标记请求特殊池。 有关信息，请参阅 **GFlags 命令**。

按池标记请求特殊池

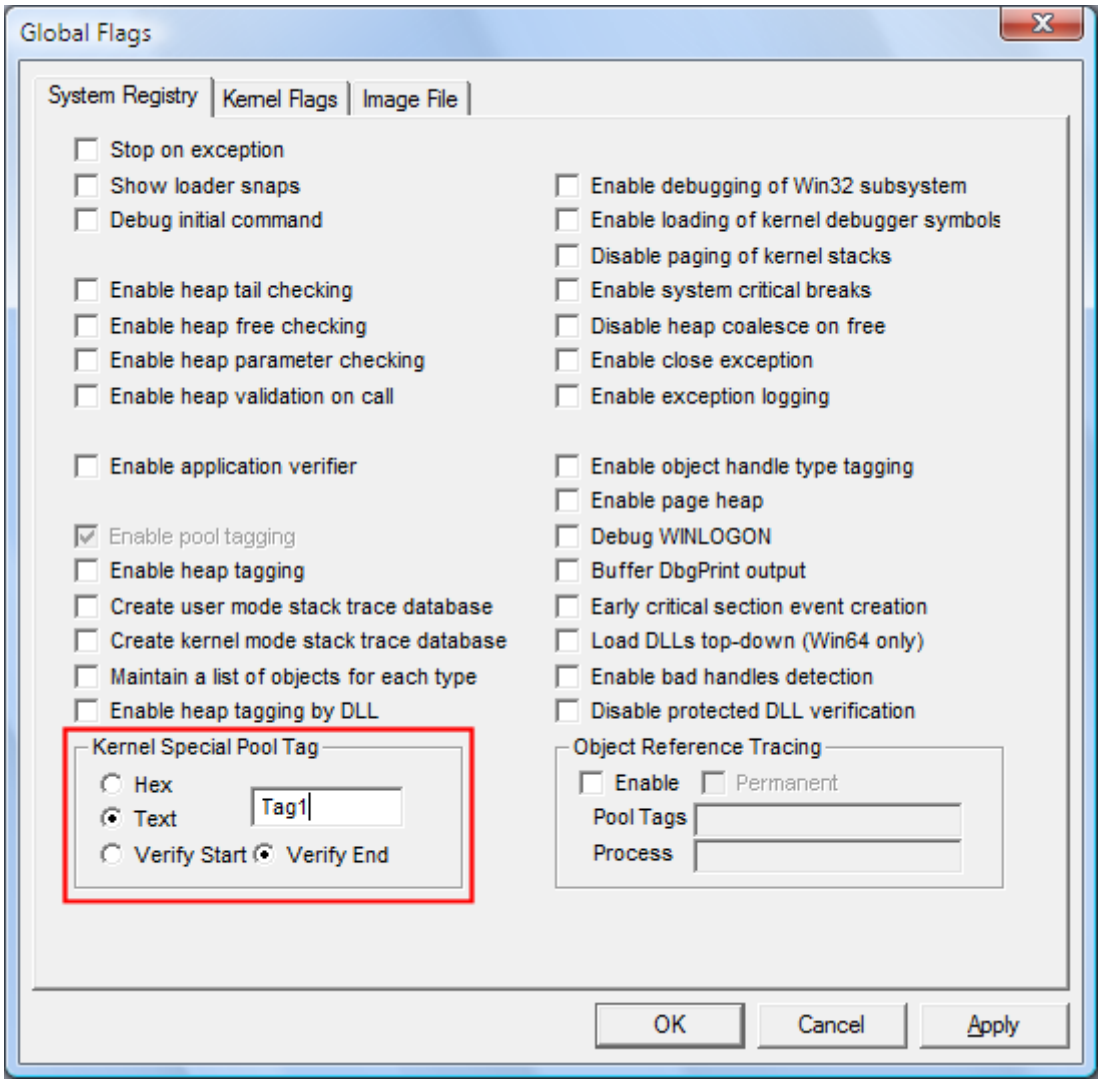
1. 选择" **系统注册表** "选项卡或" **内核标志** "选项卡。

   在 Windows Vista 和更高版本的 Windows 上，此选项在两个选项卡上都可用。 在早期版本的 Windows 上，它仅在 "**系统注册表** "选项卡上可用。

2. 在 "**内核特殊池标记** "部分中，单击" **文本**"，然后键入标记的四个字符模式。

   标记可以包含 **？** (单个字符) 和 * (多个字符) 通配符。 例如，Fat* 或 Av？4。

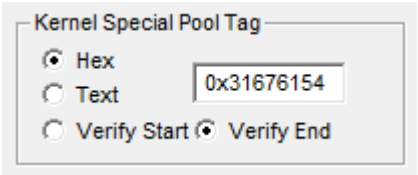
3. 以下屏幕截图显示了在"系统注册表"选项卡上以文本的形式输入的标记。



4. 单击“应用”。

   单击“ **应用**”时，GFlags 会将所选内容从 “**文本** ”更改为 “**十六** 进制”，并将 ASCII 字符显示为十六进制值，（低端）顺序。 例如，如果键入 **Tag1**，GFlags 会将标记显示为 **0x31676154** (1gaT) 。 这是存储在注册表中并由调试器和其他工具显示的方式。

   下图显示了单击“ **应用**”的效果。



言论

若要有效地使用此功能，请确保驱动程序或其他内核模式程序使用唯一的池标记。 如果怀疑驱动程序正在使用所有特殊池，请考虑在代码中使用多个池标记。 然后，可以多次测试驱动程序，在每个测试中将特殊池分配给一个池标记。

此外，选择具有十六进制值的池标记，该值大于系统的页大小。 对于内核模式代码，如果输入值小于 PAGE_SIZE 的池标记，则 Gflags 会为大小在相应范围内的所有分配请求特殊池，并为具有等效池标记的分配请求特殊池。 例如，如果选择大小 **30**，则特殊池将用于大小在 17 到 32 字节之间的所有分配，以及池标记0x0030的分配。

按分配大小请求特殊池

项目 • 2023/06/15

可以为指定大小范围内的分配请求特殊池。

在 Windows Vista 和更高版本的 Windows 中，还可以使用命令行按池标记请求特殊池。 有关信息，请参阅 **GFlags 命令**。

注意 此方法很少可用于诊断驱动程序错误，因为它会影响指定大小的所有内核池请求，而不管哪个驱动程序或内核模块请求分配。

按分配大小请求特殊池

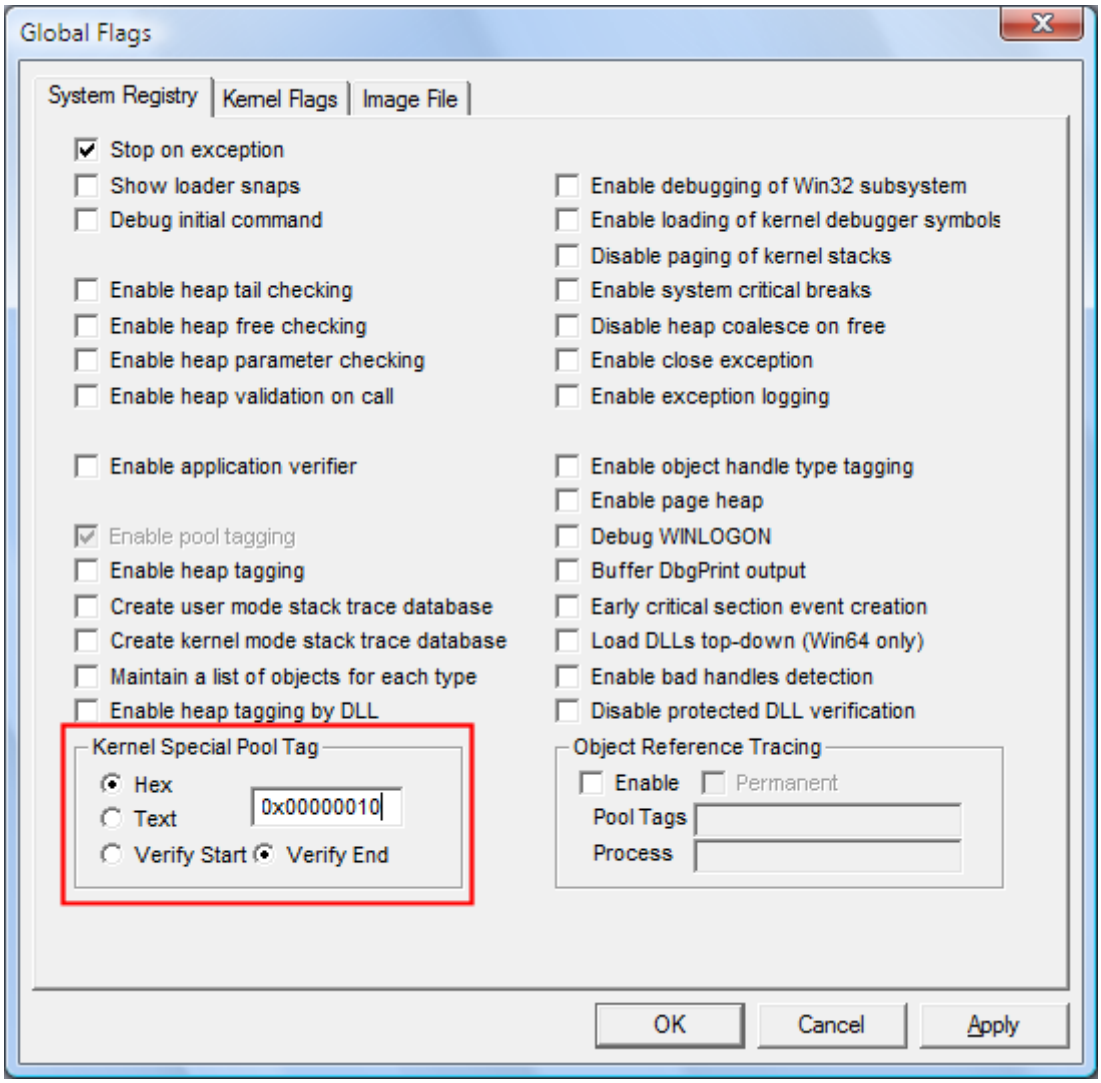
1. 选择" **系统注册表** "选项卡或" **内核标志** "选项卡。

   在 Windows Vista 和更高版本的 Windows 上，此选项在两个选项卡上都可用。 在早期版本的 Windows 上，它仅在" **系统注册表** "选项卡上可用。

2. 在 "**内核特殊池标记** "部分中，单击" **十六**进制"，然后以十六进制格式键入表示大小范围的数字。 此大小范围内的所有分配都将从特殊池进行分配。 此数字必须小于 PAGE_SIZE。

3. 单击"应用"。

   以下屏幕截图显示了以十六进制值形式输入的分配大小。


Global Flags
System Registry
Kernel Flags
Image File
Stop on exception
Show loader snaps
Debug initial command
Enable heap tail checking
Enable heap free checking
Enable heap parameter checking
Enable heap validation on call
Enable application verifier
Enable pool tagging
Enable heap tagging
Create user mode stack trace database
Create kernel mode stack trace database
Maintain a list of objects for each type
Enable heap tagging by DLL
Kernel Special Pool Tag
Hex
Text
0x00000010
Verify Start
Verify End
Enable debugging of Win32 subsystem
Enable loading of kernel debugger symbols
Disable paging of kernel stacks
Enable system critical breaks
Disable heap coalesce on free
Enable close exception
Enable exception logging
Enable object handle type tagging
Enable page heap
Debug WINLOGON
Buffer DbgPrint output
Early critical section event creation
Load DLLs top-down (Win64 only)
Enable bad handles detection
Disable protected DLL verification
Object Reference Tracing
Enable
Permanent
Pool Tags
Process
OK
Cancel
Apply

取消针对特殊池的请求

项目 • 2024/03/04

如果请求是通过使用 GFlags 发出的，则可以使用 GFlag 取消特殊池中的分配请求。 不能使用 GFlags 取消使用驱动程序验证程序发出的特殊池请求。

在 Windows Vista 和更高版本的 Windows 中，还可以使用命令行取消特殊池请求。 有关信息，请参阅 **GFlags 命令**。

取消特殊池的请求

1. 选择“系统注册表”选项卡或“内核标志”选项卡。

   在 Windows Vista 和更高版本的 Windows 上，此选项在两个选项卡上都可用。 在早期版本的 Windows 上，仅在**系统注册表**选项卡上可用。

2. 从**内核特殊池标记**框中删除文本或十六进制值。

3. 单击“应用”。

检测超量运行和欠量运行

项目 • 2023/06/15

可以使用 GFlags 中的 “**验证开始时间**” 或“ **验证结束** 时间”选项来对齐特殊池中的分配，以便最适合检测在分配) 开始之前访问内存 (访问内存) 或内存不足 (。

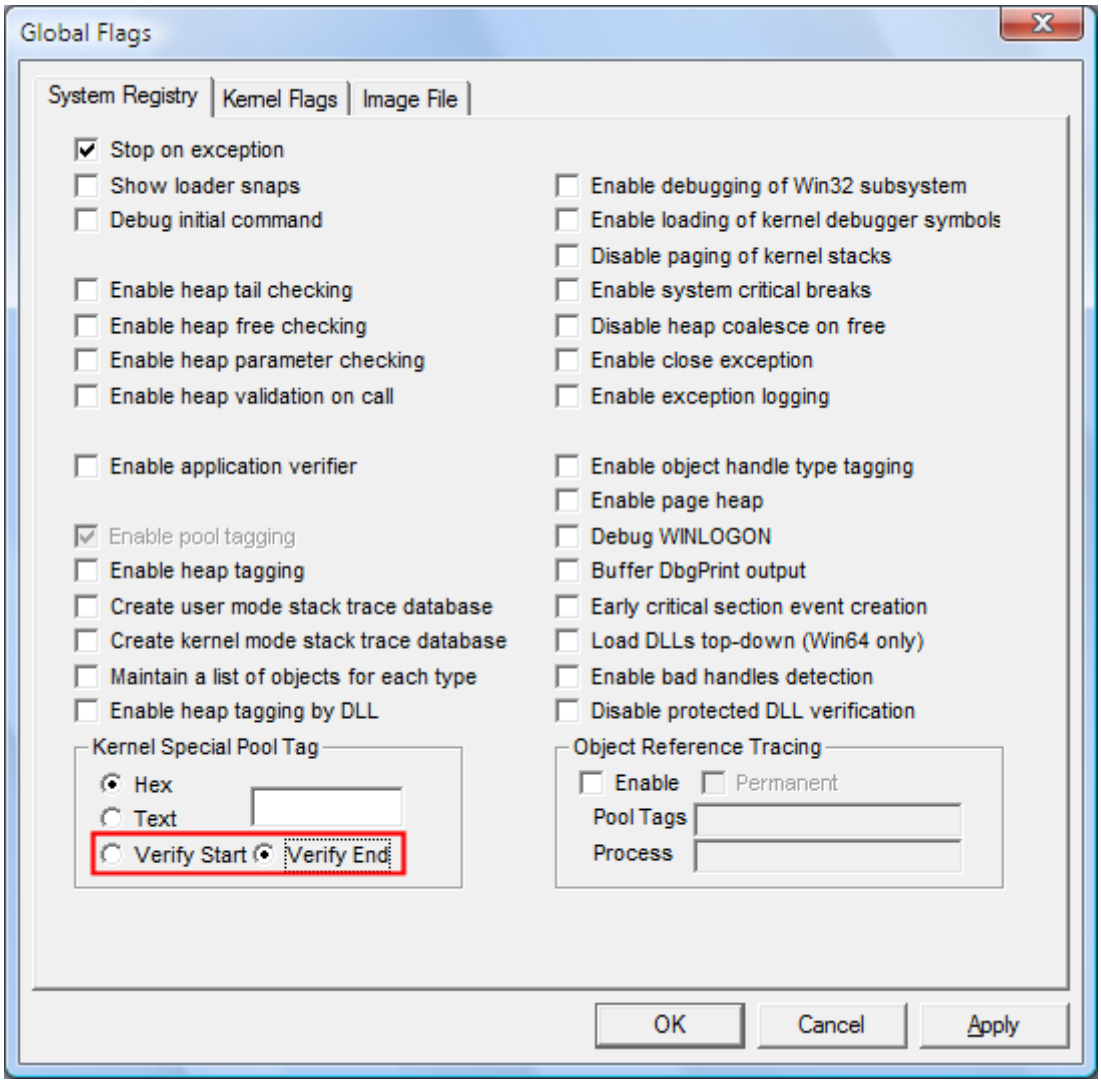
- **验证“开始”是否** 允许对特殊池中的分配进行不足检测。 当程序尝试在其特殊池内存分配之前访问内存时，这会导致 bug 检查。
- **验证 End** 是否允许对特殊池中的分配进行溢出检测。 当程序尝试访问超出其特殊池内存分配的内存时，这会导致 bug 检查。 由于溢出更为常见， **因此验证 End** 为默认值。

在 Windows Vista 及更高版本的 Windows 中，此选项在 “**系统注册表** ”和“ **内核标志** ”选项卡上可用。 在早期版本的 Windows 中，它仅在“ **系统注册表** ”选项卡上可用。

指定特殊池对齐方式

1. 单击“ **系统注册表** ”选项卡。
2. 单击“ **验证开始**” 或 “**验证结束**”。
3. 单击“应用”。

   以下屏幕截图显示了“系统 **注册表** ”选项卡上的“验证开始”和“验证结束”设置。



备注

“**验证开始**”和“**验证结束**”对齐方式设置适用于特殊池中的所有分配，包括在驱动程序验证程序中设置的特殊池分配请求。 如果在未指定池标记或分配大小的情况下设置对齐方式，则设置仅适用于驱动程序验证程序中设置的请求。

配置对象引用跟踪

项目 • 2023/11/15

可以使用 Gflags 启用、禁用和配置 Windows 的对象引用跟踪功能。 每当对象引用计数器递增或递减时，对象引用跟踪将记录顺序堆栈跟踪。 跟踪可帮助你检测对象引用错误，包括双重取消引用、引用失败和取消引用对象失败。 此功能仅在 Windows Vista 和更高版本的 Windows 中受支持。 有关此功能的详细信息，请参阅 对象引用跟踪。

启用对象引用跟踪

1. 在“Gflags”对话框中，选择“ **系统注册表** ”选项卡或“ **内核标志** ”选项卡。
2. 在“对象引用跟踪”部分中，选择“ **启用**”。

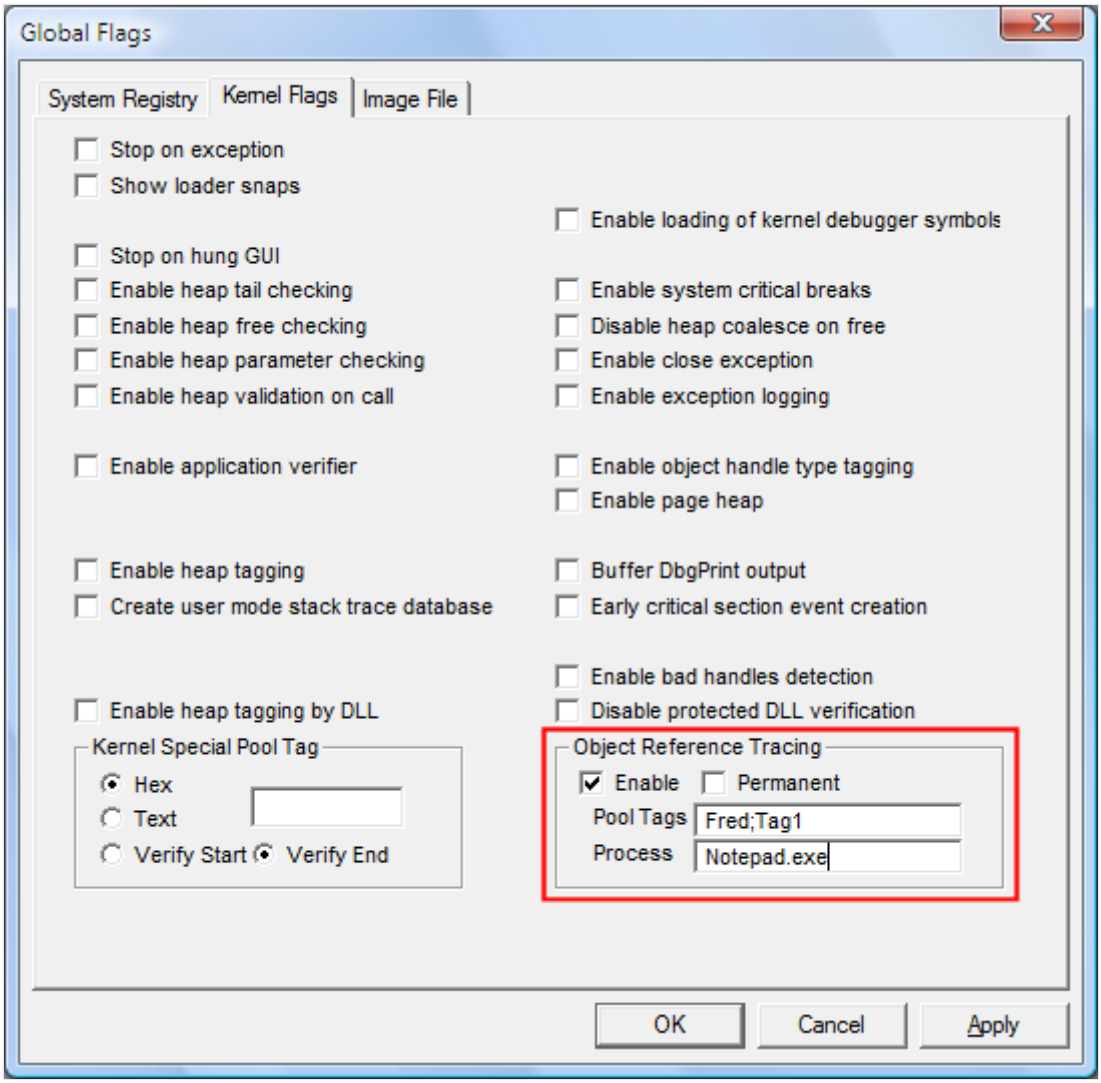
   必须将跟踪限制为具有指定池标记的对象和/或由指定进程创建的对象。
3. 若要将跟踪限制为具有特定池标记的对象，请键入池标记名称。 若要列出多个池标记，请使用分号 (;) 分隔池标记。 列出多个池标记时，跟踪将包含具有任何指定池标记的对象。 池标记区分大小写。 使用 ！ pool 和 ！ poolfind 调试器命令确定池名称。

   例如，Fred;Tag1。
4. 若要将跟踪限制为特定进程创建的对象，请键入进程的映像名称。 只能指定一个图像文件名。

   指定池标记和进程时，跟踪包括由具有任何指定池标记的进程创建的对象。
5. 若要在销毁跟踪对象后保留跟踪，请选择“ **永久**”。

   选择“ **永久**”时，跟踪将一直保留，直到禁用对象引用跟踪，或者关闭或重启 Windows。
6. 单击“**应用**”或“**确定**”。

以下屏幕截图显示了在“ **内核标志** ”选项卡上启用的对象引用跟踪。



此跟踪将仅包括由 notepad.exe 进程创建的具有池标记 **Fred** 或 **Tag1 的对象。** 由于这是(内核标志) 设置的运行时，因此跟踪会立即启动。 如果是注册表设置，则必须重启 Windows 才能启动跟踪。

禁用对象引用跟踪

1. 在“Gflags”对话框中，选择“ **系统注册表** ”选项卡或“ **内核标志** ”选项卡。“对象引用跟踪”将仅在 Windows Vista 及更高版本的 Windows 中显示在后一个选项卡上。

2. 在“对象引用跟踪”部分中，清除“**启用**检查”框。

全局标志参考

项目 • 2023/06/14

此参考介绍 GFlags 设置的标志。

注意 每个标志的符号名称仅供参考。 由于符号名称发生更改，因此它们不是全局标志的可靠标识符。

全局标志包括：

缓冲区 DbgPrint 输出

创建内核模式堆栈跟踪数据库

创建用户模式堆栈跟踪数据库

调试初始命令

调试 WinLogon

释放时禁用堆合并

禁用内核堆栈分页

禁用受保护的 DLL 验证

禁用堆栈扩展

提前创建关键节事件

启用应用程序验证工具

启用错误句柄检测

启用关闭异常

启用 Win32 子系统调试

启用异常日志记录

启用堆释放检查

启用堆标记

启用按 DLL 的堆标记

启用堆尾部检查

启用调用时进行堆验证

启用内核调试程序符号加载

启用对象句柄类型标记

启用页堆

启用池标记

启用系统关键突破

在可能的情况下大型页加载映像

维护每个类型的对象列表

对象引用跟踪

显示加载器快照

特殊池

异常时停止

GUI 挂起时停止

出现未经处理的用户模式异常时停止

缓冲区 DbgPrint 输出

项目 • 2024/03/05

Buffer DbgPrint Output 标志禁止来自 **DbgPrint**、**DbgPrintEx**、**KdPrint** 和 **KdPrintEx** 调用的调试器输出。

展开表

缩写	ddp
十六进制值	0x08000000
符号名称	FLG_DISABLE_DBGPRINT
目标	系统范围的注册表项，内核标志

备注

当调试器输出被禁止时，它不会自动显示在内核调试器中。 但是，消息始终发送到 DbgPrint 缓冲区，可以使用 **!dbgprint** 调试器扩展访问该缓冲区。

有关与调试器通信的函数的信息，请参阅向调试器发送输出。

创建内核模式堆栈跟踪数据库

项目 • 2023/06/15

“**创建内核模式堆栈跟踪数据库**”标志创建内核操作（例如资源对象和对象管理操作）的运行时堆栈跟踪数据库，并且仅在使用已检查的 Windows 内部版本时有效。 已检查的版本在 Windows 10 版本 1803 之前的较旧版本的 Windows 上可用。

缩写	kst
十六进制值	0x2000
符号名称	FLG_KERNEL_STACK_TRACE_DB
目标	系统范围的注册表项

备注

GFlags 将此标志显示为内核标志设置，但它在运行时无效，因为内核已启动。

创建用户模式堆栈跟踪数据库

项目 • 2023/06/15

创建用户模式堆栈跟踪数据库标志在特定进程的地址空间中创建运行时堆栈跟踪数据库，(映像文件模式) 或所有进程 (系统范围的) 。

缩写	ust
十六进制值	0x1000
符号名称	FLG_USER_STACK_TRACE_DB
目标	系统范围的注册表项、内核标志、映像文件注册表项

调试初始命令

项目 • 2023/06/15

调试初始命令标志调试客户端服务器运行时子系统 (CSRSS) 和 WinLogon 进程。

> **备注**
>
> 从 Windows 10 开始，CSRSS 是一个受保护的进程，只能在内核模式下进行调试。

缩写	Dic
十六进制值	0x4
符号名称	FLG_DEBUG_INITIAL_COMMAND
目标	系统范围的注册表项，内核标志

备注

NTSD 使用命令 **ntsd -d**) 调试 (的进程，但控制被重定向到内核调试器。

有关 NTSD 的详细信息，请参阅 使用 CDB 和 NTSD 进行调试。

另请参阅

启用 Win32 子系统调试

调试 WinLogon

项目 • 2023/06/15

调试 WinLogon 标志调试 WinLogon 服务。

缩写	dwl
十六进制值	0x04000000
符号名称	FLG_DEBUG_INITIAL_COMMAND_EX
目标	系统范围的注册表项

备注

NTSD 使用命令 **ntsd -d -g -x**) 调试 Winlogon (，但控件将重定向到内核调试器。

有关 NTSD 的详细信息，请参阅 使用 CDB 和 NTSD 进行调试。

另请参阅

调试初始命令， 启用 Win32 子系统调试

释放时禁用堆合并

项目 • 2024/02/06

释放时，“禁用堆合并”标志会使相邻的堆内存块分离。

展开表

缩写	dhc
十六进制值	0x00200000
符号名称	FLG_HEAP_DISABLE_COALESCING
目标	系统范围的注册表项、内核标志、映像文件注册表项

备注

默认情况下，Windows 将 (“合并”) 新释放的相邻块合并为单个块。 组合这些块需要一些时间，但可以减少在堆找不到连续内存时可能强制堆分配额外内存的碎片。

此标志用于保持与旧应用程序的兼容性。

禁用内核堆栈分页

项目 • 2024/02/06

“**禁用内核堆栈分页**”标志可防止非活动线程的内核模式堆栈分页。

展开表

缩写	dps
十六进制值	0x80000
符号名称	FLG_DISABLE_PAGE_KERNEL_STACKS
目标	系统范围的注册表项

备注

通常，内核模式堆栈无法分页;它保证驻留在内存中。 但是，Windows 有时会对非活动线程的内核堆栈进行分页。 此标志可防止出现这些情况。

仅当线程堆栈位于物理内存中时，内核调试器才能提供有关线程的信息。 在调试死锁时以及必须跟踪每个线程的其他情况下，此标志尤其重要。

禁用受保护的 DLL 验证

项目 • 2023/06/15

禁用受保护的 DLL 验证标志显示在 GFlags 中，但它对 Windows 没有影响。

缩写	Dpd
十六进制值	0x80000000
符号名称	FLG_DISABLE_PROTDLLS
目标	系统范围的注册表项、内核标志、映像文件注册表项

禁用堆栈扩展

项目 • 2024/02/07

“**禁用堆栈扩展**”标志可防止内核将进程中线程的堆栈扩展到初始提交的内存之外。

展开表

缩写	dse
十六进制值	0x10000
符号名称	FLG_DISABLE_STACK_EXTENSION
目标	映像文件注册表项

备注

此功能用于模拟堆栈扩展) 失败 (内存不足的情况，并测试即使在内存不足的情况下也能正常运行的战略系统进程。

提前创建关键节事件

项目 • 2024/02/06

“**早期关键节事件创建**”标志在初始化关键部分时创建事件句柄，而不是等到需要该事件。

展开表

缩写	Cse
十六进制值	0x10000000
符号名称	FLG_CRITSEC_EVENT_CREATION
目标	系统范围的注册表项、内核标志、映像文件注册表项

备注

当 Windows 无法创建事件时，它会在初始化期间生成异常，并且对进入和离开关键部分的调用不会失败。

由于此标志使用大量非分页池内存，因此仅在具有足够内存的非常可靠的系统上使用它。

启用应用程序验证工具

项目 • 2024/02/06

“**启用应用程序验证程序**”标志启用用于用户模式应用程序测试的系统功能，例如页面堆验证、锁定检查和句柄检查。

展开表

缩写	vrf
十六进制值	0x100
符号名称	FLG_APPLICATION_VERIFIER
目标	系统范围的注册表项、内核标志、映像文件注册表项

备注

此标志仅启用最基本的检测功能。 若要可靠地测试用户模式应用程序，请使用应用程序验证程序 (appverif.exe) 。 有关详细信息，请参阅 应用程序验证程序。

启用错误句柄检测

项目 • 2023/06/15

每当用户模式进程将无效句柄传递给对象管理器时， “**启用错误的句柄”检测** 标志都会引发用户模式异常 (STATUS_INVALID_HANDLE) 。

缩写	bhd
十六进制值	0x40000000
符号名称	FLG_ENABLE_HANDLE_EXCEPTIONS
目标	系统范围的注册表项，内核标志

启用关闭异常

项目 • 2023/06/15

每当将无效句柄传递到 CloseHandle 接口或相关接口（如 SetEvent）时，**启用关闭异常**标志将引发用户模式异常，这些接口会将句柄作为参数。

缩写	ece
十六进制值	0x00400000
符号名称	FLG_ENABLE_CLOSE_EXCEPTIONS
目标	系统范围的注册表项，内核标志

注意此标志仍受支持，但首选启用错误句柄检测标志 (bhd)，它执行更全面的句柄使用检查。

启用 Win32 子系统调试

项目 • 2023/06/15

> **备注**
>
> 从 Windows 10 开始，CSRSS 是一个受保护的进程，只能在内核模式下进行调试。

启用 Win32 子系统调试标志在 NTSD 调试器中调试客户端服务器运行时子系统 (csrss.exe) 。

缩写	d32
十六进制值	0x20000
符号名称	FLG_ENABLE_CSRDEBUG
目标	系统范围的注册表项

备注

NTSD 使用命令 **ntsd -d -p -1** 调试进程。

仅当 调试初始命令 标志 (dic) 或同时设置了 调试 WinLogon 标志 (dwl) 时，此标志才有效。

有关 NTSD 的详细信息，请参阅 使用 CDB 和 NTSD 进行调试。

启用异常日志记录

项目 • 2024/02/06

“**启用异常日志记录**”标志在内核运行时库中创建异常记录的日志。 可以从内核调试器访问日志。

展开表

缩写	鳗鱼
十六进制值	0x00800000
符号名称	FLG_ENABLE_EXCEPTION_LOGGING
目标	系统范围的注册表项，内核标志

启用堆释放检查

项目 • 2024/02/07

“**启用无堆**”**检查**标志在释放时验证每个堆分配。

展开表

缩写	Hfc
十六进制值	0x20
符号名称	FLG_HEAP_ENABLE_FREE_CHECK
目标	系统范围的注册表项、内核标志、映像文件注册表项

另请参阅

启用堆尾部检查、 启用堆参数检查

启用堆参数检查

项目 • 2024/02/06

每当调用堆函数时，“**启用堆参数检查**”标志会验证堆的选定方面。

展开表

缩写	Hpc
十六进制值	0x40
符号名称	FLG_HEAP_VALIDATE_PARAMETERS
目标	系统范围的注册表项、内核标志、映像文件注册表项

另请参阅

启用调用时进行堆验证

启用堆标记

项目 • 2023/06/15

“**启用堆标记**”标志将唯一标记分配给堆分配。

缩写	htg
十六进制值	0x800
符号名称	FLG_HEAP_ENABLE_TAGGING
目标	系统范围的注册表项、内核标志、映像文件注册表项

备注

可以使用带 -t 参数的 **！堆** 调试器扩展来显示标记。

另请参阅

启用按 DLL 的堆标记

启用按 DLL 的堆标记

项目 • 2023/06/15

“**按 DLL 启用堆标记**”标志将唯一标记分配给由同一 DLL 创建的堆分配。

缩写	htd
十六进制值	0x8000
符号名称	FLG_HEAP_ENABLE_TAG_BY_DLL
目标	系统范围的注册表项、内核标志、映像文件注册表项

备注

可以使用带 -t 参数的 **！堆** 调试器扩展来显示标记。

另请参阅

启用堆标记

启用堆尾部检查

项目 • 2024/02/06

“**启用堆尾部检查**”标志在释放堆时检查缓冲区溢出。

展开表

缩写	htc
十六进制值	0x10
符号名称	FLG_HEAP_ENABLE_TAIL_CHECK
目标	系统范围的注册表项、内核标志、映像文件注册表项

备注

此标志在每个分配的末尾添加一个短模式。 Windows 堆管理器在释放块时检测模式，如果块已修改，堆管理器会闯入调试器。

另请参阅

启用堆自由检查、 启用堆参数检查

启用调用时进行堆验证

项目 • 2023/06/15

每次 **调用堆函数时，** “**启用堆验证** ”标志都会验证整个堆。

缩写	hvc
十六进制值	0x80
符号名称	FLG_HEAP_VALIDATE_ALL
目标	系统范围的注册表项、内核标志、映像文件注册表项

备注

若要避免与此标志关联的高开销，请使用 **HeapValidate** 函数而不是设置此标志，尤其是在关键时刻，例如销毁堆时。 但是，此标志可用于检测池中的随机损坏。

另请参阅

启用堆参数检查

启用内核调试程序符号加载

项目 • 2023/06/15

下次 Windows 启动时， **启用加载内核调试器符号** 标志会将内核符号加载到内核内存空间中。

缩写	ksl
十六进制值	0x40000
符号名称	FLG_ENABLE_KDEBUG_SYMBOL_LOAD
目标	系统范围的注册表项，内核标志

备注

内核符号用于内核分析和高级内核调试工具。

启用对象句柄类型标记

项目 • 2023/06/15

“**启用对象句柄类型标记**”标志显示在 GFlags 中，但它对 Windows 没有影响。

缩写	eot
十六进制值	0x01000000
符号名称	FLG_ENABLE_HANDLE_TYPE_TAGGING
目标	系统范围的注册表项，内核标志

启用页堆

项目 • 2024/07/08

“ **启用页堆** ”标志启用页堆验证，它监视动态堆内存操作，包括分配和释放操作，并在验证程序检测到堆错误时导致调试器中断。

展开表

缩写	hpa
十六进制值	0x02000000
符号名称	FLG_HEAP_PAGE_ALLOCS
目标	系统范围的注册表项、内核标志、映像文件注册表项

备注

当在系统注册表或内核标志中设置映像文件和标准页面堆验证时，此选项启用完整页堆验证。

- *完整页堆验证* （对于 **/i**）在每个分配结束时放置一个保留虚拟内存区域。
- *标准页堆验证* （对于 **/r** 或 **/k**）在分配结束时放置随机模式，并在释放堆块时检查模式。

为图像文件设置此标志与在命令行中键入 **gflags /p /enable** *ImageFile* **/full** 相同。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

启用池标记

项目 • 2024/02/07

“**启用池标记**”标志收集数据并计算有关按池标记值排序的池内存分配的统计信息。

展开表

缩写	ptg
十六进制值	0x400
符号名称	FLG_POOL_ENABLE_TAGGING
目标	系统范围的注册表项

备注

此标志在 Windows Server 2003 及更高版本的 Windows 中永久设置。 在这些系统上，“**全局标志**”**对话框中的**“**启用池标记**检查”框灰显，用于启用或禁用池标记的命令将失败。

使用 **ExAllocatePoolWithTag** 或 **ExAllocatePoolWithQuotaTag** 设置标记值。 如果未 (ExAllocatePool、**ExAllocatePoolWithQuota**) 指定标记值，Windows 将创建一个默认值为“None”的标记。由于具有“None”标记的所有分配的数据是合并的，因此无法区分特定分配的数据。 有关这些例程的信息，请参阅 Windows 驱动程序工具包 (WDK) 。

> **重要**
>
> 本主题中讨论的 ExAllocatePool DDI 已在 Windows 10 版本 2004 中弃用，并已替换为 **ExAllocatePool2** 和 **ExAllocatePool3**。 有关详细信息，请参阅 **将已弃用的 ExAllocatePool 调用更新到 ExAllocatePool2 和 ExAllocatePool3**。

注意 若要显示 Windows 收集的有关标记分配的数据，请使用 PoolMon，这是 Windows 驱动程序工具包中包含的工具。

启用静默进程退出监视

项目 • 2024/02/07

启用无提示进程退出监视标志可为进程启用无提示退出监视。

展开表

十六进制值	0x200
符号名称	FLG_MONITOR_SILENT_PROCESS_EXIT
目标	映像文件注册表项

备注

有关监视进程是否无提示退出的详细信息，请参阅 监视无提示进程退出。

启用系统关键突破

项目 • 2024/02/07

“**启用系统关键中断**”标志强制系统中断进入调试器。

展开表

缩写	渣 打 银行
十六进制值	0x100000
符号名称	FLG_ENABLE_SYSTEM_CRIT_BREAKS
目标	系统范围的注册表项、内核标志、映像文件注册表项

备注

为进程 (映像文件) 设置时，每当指定进程异常停止时，此标志就会强制系统中断调试器。 仅当进程调用 **RtlSetProcessBreakOnExit** 和 **RtlSetThreadBreakOnExit** 接口时，此标志才有效。

当设置系统范围的 (注册表或内核标志) 时，每当调用 **RtlSetProcessBreakOnExit** 和 **RtlSetThreadBreakOnExit** 接口的进程异常停止时，此标志将强制系统中断调试器。

在可能的情况下大型页加载映像

项目 • 2023/06/15

在将二进制文件映射到进程地址空间时，“ **使用大页面加载图像（如果可能）** ”设置指示系统使用 (4 MB) 的大型页面，而不是 (4 KB) 的标准小页面。

此设置对大型可执行文件最有用，因为它可显著减少物理内存中的页表条目数。

缩写	石油气
十六进制值	（无）
符号名称	
目标	映像文件注册表项

备注

此设置在技术上不是全局标志，因为它的值存储在单独的注册表项中，而不是 GlobalFlag 注册表项的值。 因此，无法使用十六进制值对其进行设置，并且为图像文件选择此设置时，它将显示在显示器的 “**其他设置**” 字段中。

维护每个类型的对象列表

项目 • 2024/02/07

“**为每个类型维护对象列表**”标志按对象类型（例如事件、互斥体和信号灯）收集并维护活动对象列表。

展开表

缩写	otl
十六进制值	0x4000
符号名称	FLG_MAINTAIN_OBJECT_TYPELIST
目标	系统范围的注册表项

备注

若要显示对象列表，请使用 sysinternals 工具 句柄。

注意 设置此标志时创建的链接列表为每个对象使用 8 个字节的开销。 请记得在分析完成后清除此标志。

对象引用跟踪

项目 • 2023/10/26

每次递增或递减对象引用计数器时， **对象引用跟踪** 功能都会记录顺序堆栈跟踪。 跟踪可帮助你检测对象引用错误，包括双重取消引用、引用失败和取消引用对象失败。 此功能仅在 Windows Vista 和更高版本的 Windows 中受支持。

有关在“ **全局标志** ”对话框中配置对象引用跟踪功能的信息，请参阅 配置对象引用跟踪。 有关在命令提示符下配置对象引用跟踪功能的信息，请参阅 **GFlags 命令**。 有关示例，请参阅 示例 15：使用对象引用跟踪。

如果怀疑未正确引用或取消引用特定对象，则对象引用跟踪最有用，这通常是因为池使用量增加表明对象正在泄漏，或者进程或会话无法结束，即使其句柄计数为零。 与记录在日志中供以后查看的跟踪不同，对象引用跟踪设计为在进程正在运行且对象被引用和取消引用时实时使用。 使用 **！ obtrace 调试器扩展在调试器**中查看对象引用跟踪。 由于此扩展需要指定的对象地址，因此必须提前知道哪个对象是错误的可能来源。

以下规则适用于对象引用跟踪：

- 一次只能运行一个对象引用跟踪。
- 由于内核范围的跟踪不切实际，因此必须将跟踪限制为使用指定的池标记创建的对象，或者限制为由指定进程创建的对象，（由映像文件名）指示，或两者。
- 只能为每个跟踪指定一个图像文件。 如果指定图像文件，则跟踪仅限于由图像表示的进程创建的对象。 不会跟踪由进程引用但由其他进程创建的对象。
- 每个跟踪最多可以指定 16 个池标记。 跟踪具有任何指定池标记的对象。
- 如果同时指定映像文件和一个或多个池标记，则跟踪仅限于进程创建并具有任何指定池标记的对象。
- 对象引用跟踪无法跟踪启动时已在运行的进程。 跟踪仅包括跟踪开始后启动的进程的对象。
- 将跟踪标记为跟踪的对象，直到对象被销毁或禁用跟踪。 默认情况下，对象的跟踪仅保留到对象销毁为止，但你可以指定“永久”跟踪 (**/p**) 跟踪将保留到禁用跟踪为止。
- 可以将对象引用跟踪配置存储为注册表设置或内核标志 (运行时) 设置。 如果同时具有注册表和内核标志设置，则运行时设置优先，但在关闭或重新启动计算机时会丢失。

显示加载器快照

项目 • 2023/06/15

Show loader snaps 标志捕获有关加载和卸载可执行映像及其支持库模块的详细信息，并在内核调试器控制台中显示数据。

缩写	Sls
十六进制值	0x2
符号名称	FLG_SHOW_LDR_SNAPS
目标	系统范围的注册表项、内核标志、映像文件注册表项

备注

对于系统范围的 (注册表或内核标志)， 此标志显示有关驱动程序加载和卸载操作的信息。

对于每个进程 (图像文件)， 此标志显示有关加载和卸载 DLL 的信息。

特殊池

项目 • 2023/06/15

特殊池功能将 Windows 配置为在使用指定的池标记分配内存或在指定大小范围内分配内存时，从保留的内存池请求内存分配。

缩写	spp
十六进制值	（无）
符号名称	（无）
目标	系统范围的注册表项 (Windows Vista 及更高版本) 系统范围的注册表项、内核标志

选择池标记

为特定池标记请求特殊池时，请确保驱动程序或其他内核模式程序使用唯一的池标记。

此外，在创建池标记（（例如使用 **ExAllocatePoolWithTag**））时，请考虑按相反顺序输入标记字符。 例如，如果标记为 **Fred**，请考虑将其输入为 **derF** (0x64657246) 。 池标记存储在注册表中，并反向显示在调试器和其他工具中，（低端）顺序。 如果按相反顺序输入它们，它们将按正向顺序显示 (0x46726564)

如果怀疑驱动程序正在使用所有特殊池，请考虑在代码中使用多个池标记。 然后，可以多次测试驱动程序，在每个测试中将特殊池分配给一个池标记。

此外，选择具有大于系统页面大小的十六进制值的池标记。 对于内核模式代码，如果输入值小于 PAGE_SIZE 的池标记，则 Gflags 会为大小在相应范围内的所有分配请求特殊池，并为具有等效池标记的分配请求特殊池。 例如，如果选择大小 **为 30**，则特殊池将用于大小介于 17 到 32 字节之间的所有分配，以及具有池标记0x0030的分配。

选择分配大小

使用以下准则为特殊池功能选择分配大小。

在具有 x86 处理器的计算机上，PAGE_SIZE 0x1000，分配大小范围长度为 8 个字节。 若要为此范围内大小的所有分配配置特殊池功能，请输入一个等于此范围最大值加 8 的数字。 (此数字始终是 8.) 下表说明了这些值：

大小范围	**输入此数字**

大小范围	输入此数字
1 到 8 个字节	10 (十进制 16)
9 到 16 个字节	18 (十进制 24)
17 到 24 个字节	20 (十进制 32)
...	...
0xFE9到0xFF0字节	FF8 (decimal 4088)

在具有 AMD x86-64 处理器的计算机上，PAGE_SIZE 0x1000，分配大小范围长度为 16 字节。 若要为此范围内大小的所有分配配置特殊池功能，请输入一个等于此范围最大值加 16 的数字。 (此数字始终是 16.) 下表说明了这些值：

大小范围	输入此数字
1 到 16 个字节	20 (十进制 32)
17 到 32 个字节	30 (十进制 48)
33 到 48 字节	40 (十进制 64)
...	...
0xFD1到0xFE0字节	FF0 (十进制 4080)

在具有任何处理器的计算机上，可以使用星号 (*) 或0x2A (十进制 42) 为系统上的所有内存分配配置特殊池功能。

备注

有关在“全局标志”对话框中配置特殊池功能的信息，请参阅 配置特殊池。 有关在命令行上配置特殊池功能的信息，请参阅 **GFlags 命令**。 有关示例，请参阅 示例 14：配置特殊池。

当使用指定的池标记分配内存或在指定大小范围内分配内存时，Gflags 的特殊池功能指示 Windows 从保留的内存池请求内存分配。 若要为特定驱动程序的所有分配请求特殊池，请使用驱动程序验证程序。 有关详细信息，请参阅 Windows 驱动程序工具包 (WDK) 的“驱动程序验证程序”部分中的“特殊池”主题。

Gflags 和驱动程序验证程序的特殊池功能可帮助你检测和识别内核池使用中的错误来源，例如超出分配的内存空间，或引用已释放的内存。

并非所有特殊池请求都得到满足。 来自特殊池的每个分配使用一页不可分页的物理内存和两页虚拟地址空间。 如果特殊池已用尽，则会从标准池分配内存，直到特殊池再次可

用。 当从标准池填充特殊池请求时，请求函数将返回成功状态。 它不会返回错误，因为分配已成功，即使它不是从特殊池填充的。

特殊池的大小随着系统上的物理内存量而增加;理想情况下，这至少应为 1 GB (GB) 。 在 x86 计算机上，由于虚拟 (除了占用物理) 空间外，请勿在使用特殊池时使用 **/3GB** 启动选项。 最好将页面文件最小/最大数量增加 2 或 3 个因子。

还可以配置特殊池功能，使内存分配保持一致，以检测在分配 (“不足”) 之前对内存的引用，或者对内存的引用超出分配 (“溢出”) 。 此功能仅在所有版本的 Windows 上的“全局标志”对话框中可用。 有关详细信息，请参阅 检测溢出和不足。

在 Windows Vista 及更高版本的 Windows 上，你可以将特殊池功能配置为需要重新启动的注册表设置，但在更改它之前仍然有效，或者配置为不需要重新启动，但仅在重新启动或关闭 Windows 之前才生效的内核标志设置。 在早期版本的 Windows 中，特殊池仅可用作注册表设置。

在 Windows Vista 及更高版本的 Windows 上，可以使用“全局标志”对话框或命令行配置特殊池功能。 在早期版本的 Windows 中，此功能仅在“全局标志”对话框中可用。

异常时停止

项目 • 2023/06/15

每当发生内核模式异常时，“ **停止出现异常时** ”标志都会使内核中断到内核调试器中。

缩写	国企
十六进制值	0x1
符号名称	FLG_STOP_ON_EXCEPTION
目标	系统范围的注册表项、内核标志、映像文件注册表项

备注

Windows 将除严重性为“警告”或“错误”的STATUS_PORT_DISCONNECT) 之外的所有第一次机会异常 (传递给调试器，然后再将其传递给本地异常处理程序。

GUI 挂起时停止

项目 • 2024/02/06

挂起的 GUI 时停止标志显示在 GFlags 中，但它对 Windows 没有影响。

展开表

缩写	shg
十六进制值	0x8
符号名称	FLG_STOP_ON_HUNG_GUI
目标	内核标志

出现未经处理的用户模式异常时停止

项目 • 2023/06/15

每当发生 **未经处理的用户模式异常时，“停止用户模式”异常** 标志会导致内核调试器中断。

缩写	苏
十六进制值	0x20000000
符号名称	FLG_STOP_ON_UNHANDLED_EXCEPTION
目标	系统范围的注册表项、内核标志、映像文件注册表项

GFlags 示例

项目 • 2023/06/15

以下示例演示如何提交 GFlags 命令，以及如何在实际方案中使用 GFlags 功能。

本部分包括以下主题。

示例 1：显示全局标志

示例 2：使用标志缩写设置标志

示例 3：使用标志的十六进制值设置标志

示例 4：设置多个标志

示例 5：清除标志

示例 6：清除全部标志

示例 7：清除映像文件的所有标志

示例 8：扩大用户模式堆栈跟踪数据库

示例 9：检测池内存泄漏

示例 10：检测进程中的堆内存泄漏

示例 11：启用页堆验证

示例 12：使用页堆验证查找 Bug

示例 13：列出带有全局标志的映像文件

示例 14：配置特殊池

示例 15：使用对象引用跟踪

第二部分中的示例适用于仅在 Windows Vista 和更高版本的 Windows 中可用的功能。

示例 1：显示全局标志

项目 • 2023/06/15

此示例中演示的命令显示注册表中设置的系统范围标志、为会话 (内核模式) 设置的系统标志，以及为映像文件设置的标志。

以下 GFlags 命令显示注册表中设置的系统范围标志的当前值。 它使用 **/r** 参数指定系统范围的注册表项。

控制台

```
gflags /r
```

作为响应，GFlags 显示一个十六进制值，表示所有标志集和标志集列表的总和。

控制台

```
Current Boot Registry Settings are: 40001400
    ptg - Enable pool tagging
    ust - Create user mode stack trace database
    bhd - Enable bad handles detection
```

在此示例中，结果显示设置了三个标记，组合值为 0x40001400。

- 启用池标记 (ptg) = 0x400
- 创建用户模式堆栈跟踪数据库 (ust) = 0x1000
- 启用错误句柄检测 (bhd) = 0x40000000

以下命令显示为当前会话设置的标志。 它使用 **/k** 参数来指示内核模式。

控制台

```
gflags /k
```

以下命令显示注册表中为映像文件notepad.exe设置的标志。 它使用 **/i** 参数来指示图像文件模式并指定图像文件。

控制台

```
gflags /i notepad.exe
```

请记住，显示的标志值可能不是当前有效的标志值。 在重启 Windows 之前，对系统范围的标志的更改不会生效。 在重启程序之前，对映像文件标志设置的更改不会生效。

示例 2：使用标志缩写设置标志

项目 • 2023/06/15

以下命令设置notepad.exe图像文件的 “显示加载程序对齐 ”标志。 **Show Loader Snaps** 拍摄加载过程的快照，并详细捕获可执行映像及其支持库模块的加载和卸载。

命令使用 **/i** 参数来指示图像文件模式，并指定notepad.exe图像文件的名称。 为了标识标志，该命令使用 **sls**（ **显示加载程序对齐**的缩写），并在缩写前面加上加号 (+) 以指示已设置标志。 如果没有加号，该命令将不起作用。

控制台

```
gflags /i notepad.exe +sls
```

作为响应，GFlags 显示为notepad.exe设置的标志。 显示指示命令成功。 为记事本程序的所有新会话启用了 “**显示加载程序对齐** ”功能。

控制台

```
Current Registry Settings for notepad.exe executable are: 00000002
    sls - Show Loader Snaps
```

示例 3：使用标志的十六进制值设置标志

项目 • 2023/06/14

以下命令设置系统范围的 “启用”页堆 标志。 **启用页面堆** 会向每个堆分配添加一个保护页和其他跟踪功能。

命令使用 **/r** 参数来指示系统范围的注册表模式。 为了标识标志，命令使用 **2000000**，它表示“ **启用页堆**”的十六进制值0x2000000。

尽管命令设置标志，但它省略加号 (+) 符号。 使用十六进制值时，符号是可选的，添加 (+) 为默认值。

控制台

```
gflags /r 2000000
```

作为响应，GFlags 显示注册表中设置的系统范围标志。 显示指示命令成功。 Windows 重启时，将为所有进程启用 “**启用页面堆** ”功能。

控制台

```
Current Boot Registry Settings are: 02000000
    hpa - Enable page heap
```

示例 4：设置多个标志

项目 • 2023/06/15

以下命令为当前会话设置这三个标志：

- (hfc、0x20) 启用堆自由检查
- 启用堆参数检查 (hpc，0x40)
- (hvc、0x80) 调用启用堆验证

此命令使用 **/k** 参数指定仅) (会话的内核模式。 它将内核模式的值设置为 **E0** (0xE0) ，所选标志的十六进制值之和 (0x20 + 0x40 + 0x80) 。

控制台

```
gflags /k e0
```

作为响应，GFlags 显示为会话设置的标志的修订值。 显示指示命令成功，并且已设置命令中指定的三个标志。

控制台

```
Current Running Kernel Settings are: 000000e0
    hfc - Enable heap free checking
    hpc - Enable heap parameter checking
    hvc - Enable heap validation on call
```

可以使用多个不同的 GFlags 命令来设置标志。 下面的每一个命令都与此示例中使用的命令具有相同的效果，并且可以互换使用方法：

控制台

```
gflags /k +20 +40 +80
gflags /k +E0
gflags /k +hfc +hpc +hvc
```

内核 (运行时) 标志立即生效，并在 Windows 关闭之前保持有效。

示例 5：清除标志

项目 • 2023/06/15

以下命令清除注册表中设置的系统范围的 Enable 页堆 标志。 该命令使用 **/r** 参数来指示系统范围的注册表模式和 **hpa**，即 **Enable page heap** 标志的缩写。 减号 () 指示要清除标志。

控制台

```
gflags /r -hpa
```

作为响应，GFlags 显示系统范围的注册表项的当前值。 显示指示命令成功，并且不再设置任何标志。

控制台

```
Current Boot Registry Settings are: 00000000
```

请注意，以下命令使用 **Enable Page Heap** 标志的十六进制值，其效果与此示例中使用的命令相同。 这些命令可以互换使用：

控制台

```
gflags /r -02000000
```

示例 6：清除全部标志

项目 • 2023/06/15

此示例演示了清除注册表和会话中设置的所有标志的两种不同方法：

- 减去当前标志值。
- 减去高值。

注意 此示例演示的方法仅清除标志。 它们不会将最大堆栈跟踪大小或内核特殊池标记重置为默认值。

减去当前标志值

以下命令通过减去条目的当前值，清除注册表中系统范围的标志项中设置的所有标志。 在此示例中，当前值为 0xE0。 该命令使用 **/r** 参数指示系统范围的注册表模式，并使用减号 (-) 的 E0 值从标志值中减去 E0。

控制台

```
gflags /r -E0
```

作为响应，GFlags 显示系统范围标志注册表项的修订值。 值为零表示命令成功，并且注册表中不再设置任何系统范围的标志。

控制台

```
Current Boot Registry Settings are: 00000000
```

请注意，以下命令与此示例中使用的命令具有相同的效果，可以互换使用：

控制台

```
gflags /r -20 -40 -80
gflags /r -hfc -hpc -hvc
```

减去高值

以下命令通过从系统范围的标志设置中减去 (0xFFFFFFFF) 高值来清除所有系统范围的标志。

```
```

控制台

```
gflags /r -ffffffff
```

作为响应，GFlags 显示系统范围标志条目的修订值。 值为零表示命令成功，并且注册表中不再设置任何系统范围的标志。

控制台

```
Current Boot Registry Settings are: 00000000
```

提示 在记事本中键入此命令，然后将该文件另存为clearflag.bat。 此后，若要清除所有标志，只需键入 **ClearFlag**。

最后，以下示例演示清除所有标志的直观方法不起作用。

以下命令似乎将系统范围的标志条目的值设置为 0。 但是，它实际上将零添加到系统范围的标志值。 在此示例中，系统范围的标志条目的当前值0xE0。

控制台

```
gflags /r 0
```

作为响应，GFlags 在命令完成后显示系统范围的标志值：

控制台

```
Current Boot Registry Settings are: 000000e0
    hfc - Enable heap free checking
    hpc - Enable heap parameter checking
    hvc - Enable heap validation on call
```

命令不起作用，因为它将值 0 添加到系统范围的标志条目。

示例 7：清除映像文件的所有标志

项目 • 2023/06/15

以下命令清除图像文件的所有标志和图像调试器选项。 命令将高值 (0xFFFFFFFF) 添加到当前标志值。 GFlags 通过删除映像文件的 **GlobalFlag** 注册表项进行响应，从而删除它存储的所有值。

此命令不会影响系统范围的 **GlobalFlag** 注册表项中设置的标志或为会话 (内核模式) 设置的标志。

控制台

```
gflags /i notepad.exe ffffffff
```

作为响应，GFlags 显示一条消息，指示没有为图像文件设置标志：

控制台

```
No Registry Settings for notepad.exe executable
```

示例 8：扩大用户模式堆栈跟踪数据库

项目 • 2023/06/15

以下 GFlags 命令将 myapp.exe（一种虚构程序）的用户模式堆栈跟踪数据库的最大大小从 8 MB 增加到 24 MB。

命令使用 **/i** 参数指定图像文件。 它使用 **/tracedb** 参数设置最大堆栈跟踪数据库大小，并使用值 24 来指示大小（以兆字节为单位）。 命令使用十进制单位。（十六进制单位无效。）

控制台

```
gflags /i MyApp.exe /tracedb 24
```

如以下错误消息所示，此命令失败，因为未为 MyApp 映像文件设置 创建用户模式堆栈跟踪数据库 (+ust) 标志。 在创建跟踪数据库之前，无法设置跟踪数据库的大小。

控制台

```
Failed to set the trace database size for `MyApp.exe'
```

以下命令修复了此错误。 第一个命令创建用于myapp.exe的跟踪数据库，第二个命令将跟踪数据库的最大大小设置为 24 MB。 这些命令不能合并为单个命令。 以下显示显示了来自 GFlags 的命令和成功消息。

控制台

```
gflags /i MyApp.exe +ust

Current Registry Settings for MyApp.exe executable are: 00001000
    ust - Create user mode stack trace database

gflags /i MyApp.exe /tracedb 24

Trace database size for `MyApp.exe' set to 24 Mb.
```

GFlags 可以更改用户模式堆栈跟踪数据库的大小，但它不会显示它。 若要显示跟踪数据库大小，请使用注册表 API、Regedit 或 Reg (reg.exe) （Windows XP 和 Windows Server 2003 中包含的工具）检查 **STACKTraceDatabaseSizeInMB** 注册表项的值 (HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ImageFileName***StackTraceDatabaseSizeInMB**) 。

(Reg 版本包含在 Windows XP 中，但该版本不允许在同一命令中使用 **/v** 和 **/s** 开关。)

以下命令使用 Reg 查询 **StackTraceDatabaseSizeInMB** 的值：

控制台

```
reg query "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File
Execution Options\MyApp.exe" /v StackTraceDatabaseSizeInMB
```

在响应中，Reg 显示 **StackTraceDatabaseSizeInMB** 的值，该值确认已设置新值 24 (0x18)。 重新启动myapp.exe时，此值将生效。

控制台

```
! REG.EXE VERSION 3.0

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File
Execution Options\MyApp.exe
    StackTraceDatabaseSizeInMB  REG_DWORD       0x18
```

提示 在记事本中键入 **reg 查询** 命令，然后将该文件另存为tracedb.bat。 此后，若要显示 **StackTraceDatabaseSizeInMB** 的值，只需键入 **TraceDb** 即可。

示例 9：检测池内存泄漏

项目 • 2023/06/15

以下示例使用 GFlags 在注册表中设置系统范围的 Enable pool 标记 标志。 然后，它使用 PoolMon (poolmon.exe) （Windows 驱动程序工具包中的一个工具）显示内存池的大小。

PoolMon 监视分页和非分页内存池中的字节，并按池标记对其进行排序。 通过定期运行 PoolMon，可以识别随时间不断扩展的池。 此模式通常表示内存泄漏。

注意 池标记在 Windows Server 2003 及更高版本中永久启用。 在这些系统上，“**全局**标志”对话框中的“**启用池标记**检查”框灰显，用于启用或禁用池标记的命令将失败。 如果未启用池标记，PoolMon 将失败并显示以下消息：“查询池标记失败 c0000002。

检测池内存泄漏

1. 若要为早于 Windows Server 2003 的 Windows 版本中的所有进程启用池标记，请在注册表中设置系统范围的 “启用池标记” 标志。 以下命令行使用标志缩写方法，但可以按其十六进制值标识标志，也可以使用 “**全局标志** ”对话框：

   控制台

   ```
   gflags /r +ptg
   ```

2. 重新启动计算机，使注册表更改生效。

3. 使用以下命令定期运行 PoolMon。 在此命令中， **/b** 参数按大小降序对池进行排序。

   控制台

   ```
   poolmon /b
   ```

   作为响应，PoolMon 以降序显示内存池的分配，包括分配操作数和可用操作数，以及池中剩余的内存量 (字节列) 。

   控制台

   ```
   Memory: 16224K Avail: 4564K PageFlts: 31 InRam Krnl: 684K P: 680K
    Commit: 24140K Limit: 24952K Peak: 24932K  Pool N: 744K P: 2180K

    Tag  Type    Allocs          Frees         Diff   Bytes      Per Alloc
   ------------------------------------------------------------------------
   ```

```
  CM   Paged    1283 (   0)    1002 (   0)    281 1377312 (     0) 4901
 Strg  Paged   10385 (  10)    6658 (   4)   3727  317952 (   512)   85
  Fat  Paged    6662 (   8)    4971 (   6)   1691  174560 (   128)  103
 MmSt  Paged     614 (   0)     441 (   0)    173   83456 (     0)  482
```

如果分配的 “**字节数**” 列中的值由于没有明显原因而持续扩展，则该池中可能存在内存泄漏。

4. 清除 “**启用池标记**” 标志。

   以下命令行使用标志缩写方法，但可以按其十六进制值标识标志，也可以使用 “**全局标志** ”对话框：

   控制台

   ```
   gflags /r -ptg
   ```

5. 重启 Windows 以使注册表更改生效。

注意 使用追加符号 (>>) 将 PoolMon 输出重定向到日志文件。 稍后，可以检查日志文件中的池大小趋势。 例如：

控制台

```
poolmon.exe /b >> poolmon.log
```

示例 10：检测进程中的堆内存泄漏

项目 • 2023/06/15

此示例使用 GFlags 和用户模式转储堆 (UMDH，umdh.exe)，这是 Microsoft Windows 调试工具中包含的工具。

检测notepad.exe中的堆内存泄漏

1. 为notepad.exe映像文件设置 “创建用户模式堆栈跟踪数据库 ” (**ust**) 标志。

   以下命令使用 GFlags 设置 **创建用户模式堆栈跟踪数据库** 标志。 它使用 **/i** 参数来标识图像文件和 标志的 **ust** 缩写。

   控制台

   ```
   gflags /i Notepad.exe +ust
   ```

   此命令的结果是为记事本进程的所有新实例创建用户模式堆栈跟踪。

2. 设置符号文件路径。

   以下命令创建一个环境变量，用于存储符号文件目录的路径：

   控制台

   ```
   set _NT_SYMBOL_PATH=C:\Windows\symbols
   ```

3. 启动记事本。

4. (记事本进程的 PID) 查找进程标识符。

   可以从任务管理器或任务列表 (tasklist.exe) （Windows XP Professional 和 Windows Server 2003 操作系统中包含的工具）中找到任何正在运行的进程 PID。 在此示例中，记事本 PID 为 1228。

5. 运行 UMDH。

   以下命令运行 UMDH (umdh.exe) 。 它使用 **-p：** 参数指定在此示例中为 1228 的 PID。 它使用 **/f：** 参数指定堆转储（notepad.dmp）的输出文件的名称和位置。

   控制台

   ```
   umdh -p:1228 -f:notepad.dmp
   ```

作为响应，UMDH 将所有活动堆的完整转储写入 notepad.dmp 文件。

示例 11：启用页堆验证

项目 • 2023/06/15

以下命令为虚构程序myapp.exe启用完整和标准页面堆验证。

第一个命令为myapp.exe启用 *标准* 页堆验证。 它使用 **/p** 参数为进程启用页面堆。 默认情况下， **/p** 启用标准页堆。

控制台

```
gflags /p /enable myapp.exe
```

以下命令为myapp.exe程序启用 *整* 页堆验证。 尽管这些命令在注册表中创建不同的设置，但它们在功能上都等效于在“**全局标志**”对话框中为myapp.exe映像文件选择“**启用页堆**检查”框。 这些方法可以互换使用。

控制台

```
gflags /p /enable myapp.exe /full
gflags /i myapp.exe +hpa
gflags /i myapp.exe +02000000
```

无论用于启用页面堆验证的命令或对话框方法如何，以下命令都会禁用myapp.exe程序的完整页堆验证或标准页堆验证。

控制台

```
gflags /p /disable myapp.exe
gflags /i myapp.exe -hpa
gflags /i myapp.exe -02000000
```

注意 使用 **/debug** 或 **/kdebug** 参数时，请使用 **/p /disable** 参数关闭页面堆验证，（而不是 **/i -hpa** 参数）。 **/p /disable** 参数禁用页面堆验证并删除调试器读取的注册表项。

示例 12：使用页堆验证查找 Bug

项目 • 2023/06/15

以下一系列命令演示如何使用 GFlags 和 NTSD 调试器的页面堆验证功能来检测堆内存使用中的错误。 在此示例中，程序员怀疑虚构应用程序（pheap-buggy.exe）存在堆错误，并继续进行一系列测试以识别错误。

有关 NTSD 的详细信息，请参阅 使用 CDB 和 NTSD 进行调试。

步骤 1：启用标准页堆验证

以下命令为pheap-buggy.exe启用标准页堆验证：

控制台

```
gflags /p /enable pheap-buggy.exe
```

步骤 2：验证页面堆是否已启用

以下命令列出为其启用了页堆验证的图像文件：

控制台

```
gflags /p
```

作为响应，GFlags 显示以下程序列表。 在此显示中， **跟踪** 指示标准页堆验证， **完整跟踪** 表示整页堆验证。 在这种情况下，pheap-buggy.exe会列出 **跟踪**，指示已按预期启用标准页面堆验证。

控制台

```
pheap-buggy.exe: page heap enabled with flags (traces )
```

步骤 3：运行调试器

以下命令在 NTSD 中运行 pheap-buggy.exe 的 **CorruptAfterEnd** 函数，其中 **-g** (忽略初始断点)， () 参数对访问冲突异常设置第二次机会中断：

控制台

```
ntsd -g -x pheap-buggy CorruptAfterEnd
```

应用程序发生故障时，NTSD 将生成以下显示，指示它检测到pheap-buggy.exe错误：

dbgcmd

```
===========================================================
VERIFIER STOP 00000008: pid 0xAA0: corrupted suffix pattern

        00C81000 : Heap handle
        00D81EB0 : Heap block
        00000100 : Block size
#         00000000 :
===========================================================

Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00d81eb0 ecx=77f7e257 edx=0006fa18 esi=00000008
edi=00c81000
eip=77f7e098 esp=0006fc48 ebp=0006fc5c iopl=0         nv up ei pl zr na po
nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000
efl=00000246
ntdll!DbgBreakPoint:
77f7e098 cc               int     3
```

标头信息包括包含损坏块的堆的地址 (00C81000 ： 堆句柄) 、损坏块的地址 (00D81EB0：堆块) ，以及分配大小 (00000100：块大小) 。

“已损坏后缀模式”消息指示应用程序违反了 GFlags 在pheap-buggy.exe堆分配结束后插入的数据完整性模式。

步骤 4：显示调用堆栈

在下一步中，使用 NTSD 报告的地址查找导致错误的函数。 接下来的两个命令在调试器中打开行号转储，并使用行号显示调用堆栈。

dbgcmd

```
C:\>.lines

Line number information will be loaded

C:\>kb

ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be
wrong.
0006fc5c 77f9e6dd 00000008 77f9e3e8 00c81000 ntdll!DbgBreakPoint
```

```
0006fcd8 77f9f3c8 00c81000 00000004 00d81eb0
ntdll!RtlpNtEnumerateSubKey+0x2879
0006fcfc 77f9f5bb 00c81000 01001002 00000010
ntdll!RtlpNtEnumerateSubKey+0x3564
0006fd4c 77fa261e 00c80000 01001002 00d81eb0
ntdll!RtlpNtEnumerateSubKey+0x3757
0006fdc0 77fc0dc2 00c80000 01001002 00d81eb0
ntdll!RtlpNtEnumerateSubKey+0x67ba
0006fe78 77fbd87b 00c80000 01001002 00d81eb0 ntdll!RtlSizeHeap+0x16a8
0006ff24 010013a4 00c80000 01001002 00d81eb0 ntdll!RtlFreeHeap+0x69
0006ff3c 01001450 00000000 00000001 0006ffc0 pheap-
buggy!TestCorruptAfterEnd+0x2b
[d:\nttest\base\testsrc\kernel\rtl\pageheap\pheap-buggy.cxx @ 185]
0006ff4c 0100157f 00000002 00c65a68 00c631d8 pheap-buggy!main+0xa9
[d:\nttest\base\testsrc\kernel\rtl\pageheap\pheap-buggy.cxx @ 69]
0006ffc0 77de43fe 00000000 00000001 7ffdf000 pheap-buggy!mainCRTStartup+0xe3
[crtexe.c @ 349]
0006fff0 00000000 0100149c 00000000 78746341
kernel32!DosPathToSessionPathA+0x204
```

因此，调试器会显示具有行号的pheap-buggy.exe的调用堆栈。 调用堆栈显示显示，当 pheap-buggy.exe 中的 **TestCorruptAfterEnd** 函数尝试通过调用 **HeapFree**（一种重定向到 **RtlFreeHeap**）在0x00c80000释放分配时发生错误。

此错误的最可能原因是程序写入了它在此函数中分配的缓冲区末尾。

步骤 5：启用整页堆验证

与标准页堆验证不同，整页堆验证可以在发生此堆缓冲区后立即捕获误用。 以下命令为pheap-buggy.exe启用整页堆验证：

控制台

```
gflags /p /enable pheap-buggy.exe /full
```

步骤 6：验证是否启用了整页堆

以下命令列出为其启用了页堆验证的程序：

控制台

```
gflags /p
```

作为响应，GFlags 显示以下程序列表。 在此显示中， **跟踪** 指示标准页堆验证， **完整跟踪** 表示整页堆验证。 在这种情况下，pheap-buggy.exe列出 **包含完整跟踪**，指示已按预期启用整页堆验证。

控制台

```
pheap-buggy.exe: page heap enabled with flags (full traces )
```

步骤 7：再次运行调试器

以下命令在 NTSD 调试器中运行 pheap-buggy.exe 的 **CorruptAfterEnd** 函数，其中 **-g** (忽略初始断点) 和 **-x** (设置第二次机会中断的访问冲突异常) 参数：

控制台

```
ntsd -g -x pheap-buggy CorruptAfterEnd
```

应用程序发生故障时，NTSD 将生成以下显示，指示它检测到pheap-buggy.exe错误：

控制台

```
Page heap: process 0x5BC created heap @ 00880000 (00980000, flags 0x3)
ModLoad: 77db0000 77e8c000   kernel32.dll
ModLoad: 78000000 78046000   MSVCRT.dll
Page heap: process 0x5BC created heap @ 00B60000 (00C60000, flags 0x3)
Page heap: process 0x5BC created heap @ 00C80000 (00D80000, flags 0x3)
Access violation - code c0000005 (first chance)
Access violation - code c0000005 (!!! second chance !!!)
eax=00c86f00 ebx=00000000 ecx=77fbd80f edx=00c85000 esi=00c80000
edi=00c16fd0
eip=01001398 esp=0006ff2c ebp=0006ff4c iopl=0         nv up ei pl nz na po
nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000
efl=00000206
pheap-buggy!TestCorruptAfterEnd+1f:
01001398 889801010000     mov     [eax+0x101],bl
ds:0023:00c87001=??
```

启用整页堆验证后，调试器会在访问冲突时中断。 若要查找访问冲突的确切位置，请打开行号转储并显示调用堆栈跟踪。

编号的调用堆栈跟踪如下所示：

控制台

```
ChildEBP RetAddr  Args to Child
0006ff3c 01001450 00000000 00000001 0006ffc0 pheap-
buggy!TestCorruptAfterEnd+0x1f
[d:\nttest\base\testsrc\kernel\rtl\pageheap\pheap-buggy.cxx @ 184]
0006ff4c 0100157f 00000002 00c16fd0 00b70eb0 pheap-buggy!main+0xa9
[d:\nttest\base\testsrc\kernel\rtl\pageheap\pheap-buggy.cxx @ 69]
```

```
0006ffc0 77de43fe 00000000 00000001 7ffdf000 pheap-buggy!mainCRTStartup+0xe3
[crtexe.c @ 349]
WARNING: Stack unwind information not available. Following frames may be
wrong.
0006fff0 00000000 0100149c 00000000 78746341
kernel32!DosPathToSessionPathA+0x204
```

堆栈跟踪显示问题出现在pheap-buggy.exe的第 184 行中。 由于启用了整页堆验证，因此调用堆栈在程序代码中启动，而不是在系统 DLL 中启动。 因此，在发生冲突的位置捕获了违规行为，而不是在释放堆块时捕获。

步骤 8：在代码中找到错误

快速检查可揭示问题原因：程序尝试写入 256 字节 (0x100) 缓冲区的第 257 个字节 (0x101)，这是一个常见的逐一错误。

控制台

```
*((PCHAR)Block + 0x100) = 0;
```

示例 13：列出带有全局标志的映像文件

项目 • 2023/06/15

GFlags 显示为特定图像文件设置的标志，但它不显示设置了标志的所有图像文件。

Windows 将 **GlobalFlag** 注册表项的映像文件的标志存储在以下注册表路径中为映像文件命名的注册表子项中，**HKEY_LOCAL_MACHINE\ SOFTWARE\ Microsoft\ Windows NT\ CurrentVersion\ Image File Execution Options*ImageFileName*\GlobalFlag**。

若要确定哪些图像文件设置了标志，请使用 Windows Server 2003 中包含的 Reg (reg.exe) 。

以下 Reg **Query** 命令在指定的注册表路径中搜索 **GlobalFlag** 注册表项。 **-v** 参数指定 **GlobalFlag** 注册表项。 **/s** 参数使搜索是递归的。

控制台

```
reg query "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File
Execution Options" /v GlobalFlag /s
```

作为响应，Reg 会在路径和条目的值中显示 **GlobalFlag** 注册表项的所有实例。

控制台

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File
Execution Options\notepad.exe
    GlobalFlag    REG_SZ    0x00001000

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File
Execution Options\photohse.EXE
    GlobalFlag    REG_SZ    0x00200000

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File
Execution Options\printhse.EXE
    GlobalFlag    REG_SZ    0x00200000
```

提示 在记事本中键入 **Reg** 命令，然后将文件另存为imageflags.bat。 之后，若要查找已为其设置了标志的图像文件，只需键入 **ImageFlags** 即可。

示例 14：配置特殊池

项目 • 2023/06/15

从 Windows Vista 开始，可以将 特殊池 功能配置为内核标志设置或注册表设置。 如果将它配置为内核标志 (运行时) 设置，则无需重启计算机以使更改生效。 在早期版本的 Windows 中，特殊池仅可用作注册表设置。

此外，从 Windows Vista 开始，可以从命令行设置和配置特殊池功能。 在早期版本的 Windows 中，只能在“全局标志”对话框中设置和配置特殊池功能。

按池标记请求特殊池而不重新启动

以下命令请求使用 **Tag1** 池标记的所有分配的特殊池。 此设置会立即生效，但如果关闭或重启 Windows，它将丢失。

此命令使用 **/k** 参数指定内核标志 (运行时) 设置，并使用 +spp 缩写来设置特殊池请求。

控制台

```
gflags /k +spp Tag1
```

Gflags 通过打印做出响应：

控制台

```
Special Pool set to 0x31676154
PoolTagOverruns set to 0x1
Current Running Kernel Settings are: 00000000
```

请注意，特殊池分配请求不是内核标志设置，不会反映在内核设置值中。

此外，特殊池分配请求不会更改特殊池的溢出 (0x1) 或不足 (0x0) 设置的值。 若要从溢出更改为“溢出”，则默认使用“Gflags”对话框。如果更改为“溢出”，则使用“Gflags”对话框。有关信息，请参阅 检测溢出和不足。

无法在命令行中显示池标记。 若要验证池标记是否为内核设置，请使用 Gflags 对话框。

按注册表中的池标记请求特殊池

以下命令请求使用 **Tag1** 池标记的所有分配的特殊池。 由于此设置存储在注册表中，因此必须重启计算机才能使其生效，但在更改它之前，它仍然有效。

此命令使用 **/r** 参数指定注册表设置，并使用 +spp 缩写来设置特殊的池请求。

控制台

```
gflags /r +spp Tag1
```

Gflags 通过打印做出响应：

控制台

```
Special Pool set to 0x31676154
PoolTagOverruns set to 0x1
Current Boot Registry Settings are: 00000000
```

请注意，特殊池分配请求不是注册表标志设置，不会反映在注册表设置值中。

此外，特殊池分配请求不会更改特殊池的溢出 (0x1) 或不足 (0x0) 设置的值。 若要从溢出更改为“溢出”，则默认使用“Gflags”对话框。如果更改为“溢出”，则使用“Gflags”对话框。有关信息，请参阅 检测溢出和不足。

若要验证是否已将值添加到注册表中，请使用 Reg 或 Regedit 在 **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management**项中显示 **PoolTag** 条目的值。

例如：

控制台

```
c:>reg query "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory
Management" -v PoolTag
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory
Management
    PoolTag    REG_DWORD    0x31676154
```

在不重启的情况下按大小请求特殊池

以下命令请求在 x86 计算机上分配 1 到 8 个字节的特殊池，PAGE_SIZE为0x1000，分配粒度为 8 个字节。

此命令使用 **/k** 参数指定内核标志 (运行时) 设置，并使用 +spp 缩写来设置特殊池请求。 size 值前面有 0x，表示它是一个大小而不是池标记。

0x10，通过将分配粒度 (8 个字节) 添加到范围中的最大大小 (8 个字节)， (0x10) 总共 16 个字节来计算该值。 有关确定要输入的正确值的帮助，请参阅 特殊池中的“选择分配大小”。

控制台

```
gflags /k +spp 0x10
```

Gflags 通过打印做出响应：

控制台

```
Special Pool set to 0x10
PoolTagOverruns set to 0x1
Current Running Kernel Settings are: 00000000
```

同样，特殊池分配请求不是内核标志设置，也不会反映在内核设置值中。

此外，特殊池分配请求不会更改特殊池的溢出 (0x1) 或不足 (0x0) 设置的值。 若要从溢出更改为“溢出”，则默认使用“Gflags”对话框。如果更改为“溢出”，则使用“Gflags”对话框。有关信息，请参阅 检测溢出和不足。

按注册表中的大小请求特殊池

以下命令请求在 x64 计算机上分配 1024 到 1040 字节的特殊池，PAGE_SIZE为0x1000，分配粒度为 16 字节。

此命令使用 **/r** 参数指定系统范围的注册表设置，并使用 +spp 缩写来设置特殊的池请求。 size 值前面有 0x，表示它是一个大小而不是池标记。

0x420，通过将分配粒度 (16 个字节) 添加到 (1040 字节) 范围内的最大大小来计算 (0x420) 总共 1056 个字节。 有关确定要输入的正确值的帮助，请参阅 特殊池中的“选择分配大小”。

控制台

```
gflags /r +spp 0x420
```

Gflags 通过打印做出响应：

控制台

```
Special Pool set to 0x420
PoolTagOverruns set to 0x1
Current Boot Registry Settings are: 00000000
```

同样，特殊池分配请求不是注册表标志设置，也不会反映在注册表设置值中。

此外，特殊池分配请求不会更改特殊池的溢出 (0x1) 或不足 (0x0) 设置的值。 若要从溢出更改为“溢出”，则默认使用“Gflags”对话框。如果更改为“溢出”，则使用“Gflags”对话框。有关信息，请参阅 检测溢出和不足。

若要验证是否已将值添加到注册表中，请使用 Reg 或 Regedit 在 **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management**项中显示 **PoolTag** 条目的值。

例如：

控制台

```
c:>reg query "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory
Management" -v PoolTag
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory
Management
    PoolTag    REG_DWORD    0x420
```

取消特殊池请求

以下命令取消将特殊池作为内核标志的请求 (运行时) 设置。 命令对于按池标记或大小的请求相同。

控制台

```
gflags /k -spp
```

以下命令取消将特殊池作为注册表设置的请求。 命令对于按池标记或大小的请求相同。

控制台

```
gflags /r -spp
```

命令成功后，Gflags 会通过打印：

控制台

```
Special Pool value has been deleted.
```

示例 15：使用对象引用跟踪

项目 • 2024/07/08

对象引用跟踪是一项 Windows 功能，在引用或取消引用对象时记录顺序堆栈跟踪。 它旨在检测可能导致崩溃或内存泄漏的对象处理中的错误。 其中一些错误难以检测，因为它们不一致出现。 有关详细信息，请参阅 对象引用跟踪。

可以使用“全局标志”**对话框或在命令提示符下配置对象引用跟踪。** 以下示例使用命令提示符。 有关使用 “**全局标志** ”对话框配置对象引用跟踪的信息，请参阅 “配置对象引用跟踪”。

可以使用 Gflags 来启用、禁用和配置对象引用跟踪。 过程如下：

- **使用 Gflags 在注册表中启用对象引用跟踪** 或作为内核标志（运行时）设置。 如果将设置添加到注册表，则必须重新启动计算机以开始跟踪。 如果启用设置的运行时版本，跟踪会立即启动，但在关闭或重启计算机时，跟踪设置会还原到注册表项中的那些设置。
- **启动创建可疑对象**的过程。 跟踪仅包含由跟踪开始后启动的进程创建的对象。 如果进程在重新启动期间或不久启动，请将跟踪设置添加到注册表，然后重启系统。
- **使用 ！ obtrace 调试器扩展查看跟踪。** 默认情况下，跟踪将一直保留到对象被销毁为止，但可以使用 **/p** 参数来维护跟踪，直到禁用跟踪。
- **使用 Gflags 在注册表中禁用对象引用跟踪**，或作为内核标志（运行时）设置。 如果从注册表中删除该设置，则必须重新启动计算机以结束跟踪。 如果禁用设置的运行时版本，跟踪会立即结束，但在关闭或重启计算机时，跟踪设置会还原到注册表中的设置。

这些示例演示如何使用 Gflags 来启用和禁用对象引用跟踪。 \

启用运行时跟踪

以下命令在命令提示符下启用对象引用跟踪。 该命令使用 **/ko** 参数将对象引用跟踪启用为内核标志（运行时）设置。 该命令使用 **/t** 参数指定池标记 **Tag1** 和 **Fred**。 因此，跟踪使用 **Tag1** 或 **Fred** 创建的所有对象。

控制台

```
gflags /ko /t Tag1;Fred
```

由于命令更改了内核标志（运行时）设置，因此对象引用跟踪会立即启动。 跟踪将包含池标记 **Tag1** 或 **Fred** 的所有对象，这些对象由提交命令后启动的进程创建。

Gflags 通过打印以下消息做出响应：

控制台

```
Running Kernel Settings :
Object Ref Tracing Enabled
        Temporary Traces
        Pool Tags: Tag1;Fred
        Process Name: All Processes
```

此消息指示已启用对象引用跟踪。 “临时跟踪”指示在销毁对象时删除跟踪的所有记录。 若要使跟踪“永久”，请使用 **/p** 参数，该参数指示 Windows 在禁用对象引用跟踪之前保留跟踪数据，或者计算机关闭或重启。

在注册表中启用跟踪

以下命令将对象引用跟踪配置添加到注册表。 配置的跟踪将在重新启动计算机时开始。

该命令使用 **/ro** 参数启用对象引用跟踪作为注册表设置。 该命令使用 **/i** 指定notepad.exe 和 **/t** 参数的进程来指定池标记 **Tag1** 和 **Fred**。 因此，跟踪使用 Tag1 或 Fred **池标记的记事本进程创建的所有对象。** 该命令还使用 **/p** 参数，该参数将保留跟踪数据，直到禁用跟踪。

控制台

```
gflags /ro /t Tag1;Fred /i Notepad.exe /p
```

提交命令时，Gflags 会将信息存储在注册表中。 但是，由于注册表设置在重新启动计算机之前无效，因此已配置此对象引用跟踪，但尚未启动。

Gflags 通过打印以下消息做出响应：

控制台

```
Boot Registry Settings :
Object Ref Tracing Enabled
        Permanent Traces
        Pool Tags: Tag1;Fred
        Process Name: Notepad.exe
```

该消息指示在注册表中启用了对象引用跟踪。 “永久跟踪”表示跟踪数据将保留，直到关闭或重启计算机。 该消息还会列出将跟踪的池标记和映像文件名。

显示对象引用跟踪配置

可以显示当前有效或存储在注册表中的对象引用跟踪配置，以便在计算机重新启动时使用。

在此示例中，注册表中存储了一个对象引用跟踪配置，为运行时配置了不同的对象引用跟踪配置。 运行时跟踪会立即开始（并替代任何注册表设置）。 但是，如果重新启动系统，运行时设置将丢失，并且对象引用跟踪会话注册表设置将生效。

以下命令显示运行时对象引用跟踪配置。 它使用没有其他参数的 **/ko** 参数。

控制台

```
gflags /ko
```

控制台

```
Running Kernel Settings :
Object Ref Tracing Enabled
        Temporary Traces
        Pool Tags: Tag1;Fred
        Process Name: All Processes
```

如果启用了对象引用跟踪，如本示例中所示，显示的设置描述了正在进行的跟踪。

以下命令显示存储在注册表中的对象引用跟踪配置数据。 它使用没有其他参数的 **/ro** 参数。

控制台

```
gflags /ro
```

作为响应，Gflags 显示注册表中存储的数据：

控制台

```
Boot Registry Settings :
Object Ref Tracing Enabled
        Permanent Traces
        Pool Tags: Tag1;Fred
        Process Name: Notepad.exe
```

如果在将对象引用跟踪配置添加到注册表后重启了计算机，则响应 gflags /ro 命令时显示的设置描述了正在进行的跟踪。 但是，如果尚未重启或已重启，但随后启动运行时对象

引用跟踪（/ko），则注册表中存储的设置当前无效，但在重新启动系统时，它们将再次生效。

禁用对象引用跟踪

禁用运行时（内核标志）对象引用跟踪设置时，跟踪会立即停止。 在注册表中禁用对象引用跟踪设置时，跟踪会在重新启动计算机时停止。

以下命令禁用运行时对象引用跟踪。 它使用 **/d** 参数禁用所有设置。 不能有选择地禁用设置。

控制台

```
gflags /ko -d
```

命令成功后，Gflags 将响应以下消息：

控制台

```
Running Kernel Settings :
Object Ref Tracing Disabled
```

以下命令禁用运行时对象引用跟踪。

以下命令禁用注册表中的对象引用跟踪设置。 它使用 **/d** 参数禁用所有设置。 不能有选择地禁用设置。 重启计算机时，此命令有效。

控制台

```
gflags /ro -d
```

命令成功后，Gflags 将响应以下消息：

控制台

```
Boot Registry Settings :
Object Ref Tracing Disabled
```

反馈

此页面是否有帮助? 是 否

终止工具

项目 • 2024/07/08

Kill 工具kill.exe终止一个或多个进程及其所有线程。 此工具仅适用于在本地计算机上运行的进程。

在何处获取终止工具

Kill.exe包含在 Windows 调试工具中。

终止工具命令行选项

控制台

```
kill [/f] { PID | Pattern* }
```

参数

/f强制终止进程，而不提示用户进行确认。 若要终止受保护的进程（例如系统服务），需要此选项。

*PID*指定要终止的任务的进程标识符（PID）。

若要查找任务的 PID，请使用 Microsoft Windows XP 及更高版本或 Windows 2000 中的 TList 中的 TaskList 。

*模式**
指定任务或窗口的名称的所有或部分。 Kill 工具终止其进程名称或窗口名称与模式匹配的所有进程。 需要星号。

在使用可能与许多进程或窗口名称不匹配的模式之前，请使用 **tlist** *模式* 命令测试模式。 有关详细信息，请参阅 TList 。

示例

以下命令终止名称以“myapp”开头的进程。

控制台

```
kill myapp*
```

以下命令终止进程 ID 为 2520 的进程：

控制台

```
kill 2520
```

以下命令终止名称以“my*”开头的进程。它不会提示进行确认。 即使此过程是系统服务，此命令也会成功：

控制台

```
kill /f my*
```

另请参阅

Windows 调试工具中包含的工具

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

记录器和 LogViewer

项目 • 2023/04/14

Logger 和 *LogViewer* 工具提供了标准用户模式调试的替代方法。 记录器可以监视用户模式目标应用程序的操作，并记录其所有 API 调用。 生成的信息可以在调试器中显示、保存为文本文件，或者由 LogViewer 工具以功能强大的交互式格式显示。

在何处获取 Logger 和 LogViewer

记录器和 LogViewer 包含在 Windows 调试工具中。

本部分内容

记录器类

LogViewer

记录器清单

另请参阅

Windows 调试工具中包含的工具

记录器

项目 • 2023/06/15

记录器工具可以通过两个不同的车辆激活。 一种方法是使用独立Logger.exe程序。 另一种是启动 CDB 或 WinDbg，并使用Logexts.dll调试器扩展。 这两种方法将生成相同类型的日志输出。 但是，在任何基于 NT 的操作系统上使用的最佳工具是具有Logexts.dll扩展命令的 CDB 或 WinDbg。

记录器工具也有效，但使用调试器可提供调试器的全部功率以及记录器的强大功能。

本节包括：

使用调试器和 Logexts.dll

使用 Logger.exe

记录器的局限性和限制

使用调试器和 Logexts.dll

项目 • 2023/10/26

激活记录器的方法之一是启动 CDB 或 WinDbg，并像往常一样附加到用户模式目标应用程序。 然后，使用 **！ logexts.logi** 或 **！ logexts.loge** 扩展命令。

这会在当前断点处插入代码，该断点将跳到在目标应用程序进程中加载和初始化 Logexts.dll 的例程。 这称为“将记录器注入目标应用程序”。

实际上，Logexts.dll 有两个实例在运行，因为此模块既是调试器扩展 DLL，又是注入目标应用程序的程序。 Logexts.dll 的调试器和目标实例通过内存的共享部分进行通信，该部分包括输出文件句柄、当前类别掩码和指向日志输出缓冲区的指针。

附加到目标应用程序

有关将调试器附加到目标应用程序的信息，请参阅 使用 WinDbg 调试 User-Mode 进程 或使用 CDB 调试 User-Mode 进程。

使用记录器扩展命令

有关每个扩展的完整语法，请参阅其参考页。

！ logexts.logi
将 Logger 注入目标应用程序。 这会初始化日志记录，但不启用它。

！ logexts.loge
启用日志记录。 如果未使用 **！ logexts.logi** ，则此扩展将初始化并启用日志记录。

！ logexts.logd
禁用日志记录。 这将导致删除所有 API 挂钩，以允许程序自由运行。 不会删除 COM 挂钩，因为无法随时重新启用它们。

！ logexts.logo
显示或修改输出选项。 可以有三种类型的输出：直接发送到调试器的消息、文本文件或 .lgv 文件。 .lgv 文件包含的信息比其他两个文件多得多;可以使用 LogViewer 读取它。

如果禁用文本文件输出，仍会创建大小为零的 .txt 文件。 这可能会覆盖同一位置中以前保存的文本文件。

！ logexts.logc
显示可用的 API 类别，控制哪些类别将被记录，哪些将不记录，并显示包含在任何类别中的 API。

如果禁用某个类别，则将删除该类别中所有 API 的挂钩，以便不再有任何性能开销。 不会删除 COM 挂钩，因为无法随时重新启用它们。

仅当仅对程序与 Windows 进行的特定类型的交互（例如文件操作）感兴趣时，仅启用某些类别会很有用。 这可减小日志文件大小，并降低记录器对进程执行速度的影响。

! logexts.logb

显示或刷新当前输出缓冲区。 作为性能考虑，仅当输出缓冲区已满时，日志输出才会刷新到磁盘。 默认情况下，缓冲区为 2144 字节。

由于缓冲区内存由目标应用程序管理，因此，如果目标应用程序中存在访问冲突或其他一些不可恢复的错误，则不会自动将缓冲区写入磁盘上的日志文件。 在这种情况下，应使用此命令手动将缓冲区刷新到磁盘，否则最近记录的 API 可能不会显示在日志文件中。

! logexts.logm

显示或创建模块包含/排除列表。 通常最好只记录从特定模块或模块集进行的那些 API 调用。 为了方便实现此目标，Logger 允许指定模块包含列表，或者指定模块排除列表。 例如，如果只想记录来自一个或两个模块的调用，则使用包含列表。 如果要记录除模块的简短列表之外的所有模块发出的调用，可以使用排除列表。 始终排除 Logexts.dll 和 Kernel32.dll 的模块，因为不允许记录器自行记录。

使用 Logger.exe

项目 • 2023/06/15

激活 Logger 的一种方法是运行独立Logger.exe程序。 这实质上是一个非常小的调试器，只能获取单个目标。 若要运行它，请在命令行中包含目标应用程序的名称：

```
dbgcmd

logger Target
```

激活后，它将加载指定的应用程序，并将代码插入目标应用程序，该代码将跳到加载和初始化目标应用程序进程中Logexts.dll的例程。 这称为“将记录器注入目标应用程序”。

Logger.exe实用工具和Logexts.dll模块是此记录器车辆的两个组件。 它们通过内存的共享部分进行通信，其中包括输出文件句柄、当前类别掩码和指向日志输出缓冲区的指针。

将显示一个标题为 **Logger (调试器)** 的窗口。 此窗口将显示记录器进度。

“更改设置”对话框

初始化完成并完成初始显示后，将显示 “**更改设置**” 对话框。 这样，就可以配置记录器设置。 此处介绍了各种设置：

API 设置
此列表显示可用的 API 类别。 将记录突出显示的类别：未突出显示的类别不会。 首次运行记录器时，将突出显示所有类别。 但是，在后续的运行中，记录器将跟踪为给定目标应用程序选择的类别。

如果禁用某个类别，则将删除该类别中所有 API 的挂钩，以便不再有任何性能开销。 不会删除 COM 挂钩，因为无法随时重新启用它们。

仅当仅对程序与 Windows 进行的特定类型的交互（例如文件操作）感兴趣时，仅启用某些类别会很有用。 这可减小日志文件大小，并降低记录器对进程执行速度的影响。

测 井
本部分包含 “**启用**” 和“ **禁用** ”单选按钮。 禁用日志记录将导致删除所有 API 挂钩，以允许程序自由运行。 不会删除 COM 挂钩，因为无法随时重新启用它们。

包含/排除列表
本部分控制模块包含/排除列表。 通常最好只记录从特定模块或模块集进行的那些函数调用。 为了方便实现此目标，Logger 允许指定模块包含列表，或者指定模块排除列表。 例如，如果只想记录来自一个或两个模块的调用，则使用包含列表。 如果要记录除模块的

简短列表之外的所有模块发出的调用，可以使用排除列表。 始终排除Logexts.dll和Kernel32.dll的模块，因为不允许记录器自行记录。

刷新缓冲区
此按钮将刷新当前输出缓冲区。 作为性能考虑，仅当输出缓冲区已满时，日志输出才会刷新到磁盘。 默认情况下，缓冲区为 2144 字节。

由于缓冲区内存由目标应用程序管理，因此，如果目标应用程序中存在访问冲突或其他一些不可恢复的错误，则不会自动将缓冲区写入磁盘上的日志文件。 在这种情况下，应尝试激活目标应用程序的窗口并按 F12 获取此对话框，然后按 **刷新缓冲区**。 如果未执行此操作，则最近记录的函数可能不会显示在日志文件中。

去
这会导致目标应用程序开始执行。

运行目标应用程序

选择设置后，选择“ **转到**”。 对话框将关闭，目标应用程序将开始运行。

如果使目标应用程序的窗口处于活动状态并按 F12，它将进入 Logger。 这将导致目标应用程序冻结，并重新显示 “**更改设置**” 对话框。 可以根据需要更改设置，然后按 **Go** 继续执行。

可以根据需要让目标应用程序运行一段时间。 如果它正常终止或由于错误，日志记录将停止，并且无法重新启动。

如果要退出，请选择“ **文件|退出** 并选择“ **是**”。 如果目标应用程序仍在运行，它将终止。

Logger.exe的限制

通过 Logger.exe 工具运行 Logger 时，它将仅创建一个输出文件 - 一个 .lgv 文件。 不会写入任何文本文件。 但是，将创建一个大小为零的.txt文件;这可能会覆盖以前由调试器写入的文本日志。

输出文件将始终放置在桌面的 LogExts 子目录中;无法更改此位置。

如果通过调试器运行记录器并Logexts.dll，则这些限制将不适用。

记录器的局限性和限制

项目 • 2023/06/15

记录器会增加进程的堆栈消耗，因为它在实际函数调用之前引入了额外的“包装”函数。

这可以公开通常与未初始化变量相关的应用程序中的 bug。 由于 Logger 会更改堆栈用法，因此在函数调用中声明的局部变量可能采用与没有 Logger 时不同的初始值。 如果程序使用此变量而不初始化它，则程序可能会崩溃，或者其行为方式与 Logger 不存在时不同。

不幸的是，解决此类问题并不容易。 唯一的解决方法是尝试禁用函数类别，以尝试隔离导致该问题的区域。

LogViewer

项目 • 2023/06/15

LogViewer 实用工具可以操作 .lgv 文件，该文件是记录器工具生成的压缩日志文件。 若要加载文件，只需通过 Windows 资源管理器或命令提示符窗口启动Logviewer.exe。 分析清单文件需要一些时间。 完成此操作后，可以调用 **文件 |打开** 并选择所需的 .lgv 文件。

LogViewer 打开 .lgv 文件后，它将按记录顺序显示所有函数的列表。 LogViewer 支持强大的命令来筛选出你不感兴趣的函数。 它还可以将日志文件的筛选版本保存为文本文件。

本节包括：

阅读 LogViewer 显示内容

筛选 LogViewer 函数列表

将 LogViewer 文件导出为文本

阅读 LogViewer 显示内容

项目 • 2023/06/15

LogViewer 按记录顺序显示所有函数的列表。

显示的每一行都包含多个列。 每列的重要性如下所示。

列	含义
+/-	如果此列包含“+” (加号)，则表示函数采用一个或多个参数。 若要查看参数及其值，请双击该行，或者在行以红色边框显示时按向右键。 若要再次隐藏它，请再次双击它，或在行以红色边框显示时按向左键。 此列中还有一个“d#”值。 这表示函数调用 (的“深度”，即调用嵌套在其他记录的函数调用) 的深度。
#	函数调用的顺序行号。 如果应用了筛选器，并且有兴趣了解两个函数调用的距离，这非常有用。
Thrd	对其执行函数调用的线程号。 此数字不是线程 ID，而是基于线程在进程中创建的顺序分配的编号。
来电	发出函数调用的指令地址。 这派生自调用的返回地址。 它实际上是返回地址减去 5 个字节 (**调用 dword ptr** 指令) 的典型大小。
模块	包含调用指令的模块。
API 函数	函数的名称。 为简洁起见，省略包含函数的模块的名称。
返回值	函数返回的值（如果它不是 void 函数）。

双击查看器中的某一行将展开该行，以显示函数的参数及其值“进入”到函数。 如果它们被指定为 OUT 参数，则其值“out”将显示在右侧。

还可以使用 ENTER 键或向右键和向左键展开和折叠行。

返回的失败状态代码的函数调用以粉红色着色。

筛选 LogViewer 函数列表

项目 • 2023/06/15

记录器通常捕获分析不需要的一些函数调用。 这些可以在记录器创建日志文件时被记录器筛选掉。 但是，此过程不可逆，因此通常最好是允许记录所有函数，然后在 LogViewer 中筛选显示。

可通过多种方式在 LogViewer 中筛选出函数调用：

- 在main查看区域中，通过单击或使用光标键来选择函数。 (选择某个函数时，LogViewer 会用红色显示它。) 然后按 DELETE 键，或右键单击并选择“ **隐藏**”。 这会在视图中隐藏该函数调用的所有实例。

- 选择 “**视图**” |**API 显示**。 将出现一个对话框，其中包含三个区域。 右侧是所有函数的字母顺序列表，左侧是分类分组。 可以通过选中或清除函数名称左侧的框来启用或禁用任何函数的显示。

- 选择 “**视图**” |**模块显示**。 将出现一个对话框，可用于选择调用模块;仅显示从这些模块调用的函数。

- 选择 “**视图**” |**仅第一级调用**。 这将仅显示左列中具有“d0”的调用。 通常需要隐藏由其他记录的函数进行的函数调用。 (例如，知道 **ShellExecuteEx** 在执行过程中发出 30 个不同的注册表调用通常并不有趣。)

将 LogViewer 文件导出为文本

项目 • 2023/06/15

操作日志文件的一种强大方法是将其导出为文本。 这使你可以使用任何文本编辑器的"查找"工具查找特定参数值。 虽然可以由 Logger 直接生成文本文件，但如果使用 LogViewer 筛选函数调用或添加别名，然后将此信息导出到文本文件中，则可以获得更大的灵活性。

若要从 .lgv 文件创建文本文件，请在 LogViewer 中打开该文件，然后选择" **文件 |导出到文本**。 在对话框中，选择文件名和位置。

对话框底部有多个选项：

导出差异信息
此选项将为从执行更改为执行的所有指针、句柄值和其他值别名。 LogViewer 不会输出指针的实际值 (例如"0x00123FA2") ，而是输出"指针"。同样，使用差异工具比较这两个文件时，句柄将别名为"HeapHandle1"或其他一些不会注册为 *差异* 的值。

包括不可见行
此选项禁用导出时当前应用于视图的任何筛选器。

为每个线程创建单独的文件
此选项按线程拆分文本文件，以便更轻松地执行线程。 如果要"差异"两个执行实例，这非常有用。

导出范围
此选项指定要导出的行的范围。

记录器清单

项目 • 2024/03/20

本节包括：

日志记录清单概述

清单文件的位置

清单文件格式

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

日志记录清单概述

项目 • 2024/06/11

日志记录清单是一组“头文件”，用于定义拦截和记录的函数和 COM 接口。 这些并不是真正的 C++ 头文件 - 它们的格式略有不同，明确声明了日志程序所需的信息。

例如，清单格式具有以下功能：

- 指定 OUT 参数。 这些参数在进入函数和离开函数时都应被记录。
- 标志掩码的定义。 此功能允许日志查看器将 DWORD 标志分解为其组成的位标签，以便于阅读。
- 故障案例的定义。 此功能允许日志查看器让返回失败状态代码或其他错误代码的函数行显示为阴影。 此外，如果函数设置了线程的“LastError”值，日志查看器就可以存储错误代码，并将其扩展为相应的人类可读错误信息。
- 指定可为日志差异指定别名的参数。 该功能可让日志查看器在将数据导出到文件时，选择将常量字符串赋值给一个在不同执行过程中会发生变化的值，如指针和句柄。 然后，可以使用差异工具来比较两个执行日志，从而找出差异。 如果未对指针和句柄值进行别名处理，那么在比较两个文件时就会产生没有意义的差异。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

清单文件的位置

项目 • 2023/06/15

主清单文件必须命名为 Main.h。

运行 Logger 时，Main.h 必须位于包含Logexts.dll的目录的清单子目录中。

LogViewer 比 Logger 更灵活。 它将按以下顺序搜索 Main.h：

1. 从属于包含Logviewer.exe的目录的清单目录
2. WinExt\Manifest 目录从属于包含logviewer.exe
3. %WinDir%\System32\Manifest 目录
4. %WinDir%\System\Manifest 目录

所有其他清单文件必须与 Main.h 位于同一目录中。

清单文件格式

项目 • 2023/06/15

清单文件的文件格式尽可能多地从 C++ 和 IDL 借用。 因此，采用普通的 C++ SDK 头文件并将其修改为清单文件相当容易。 分析程序完全支持 C 和 C++ 样式注释，以帮助组织和记录文件。

如果尝试添加清单文件或对现有文件进行更改，最好的方法是进行试验。 在调试器中发出 **！ logexts.logi** 或 **！ logexts.loge** 命令时，Logger 将尝试分析清单文件。 如果遇到问题，将生成一条错误消息，可能指示错误。

清单文件由以下基本元素组成：模块标签、类别标签、函数声明、COM 接口定义和类型定义。 其他类型的元素也存在，但这些元素是最重要的。

模块标签

模块标签只是声明 DLL 导出随后声明的函数的内容。 例如，如果清单文件用于从 Comctl32.dll 记录一组函数，则会在声明任何函数原型之前包含以下模块标签：

```
C++

module COMCTL32.DLL:
```

模块标签必须显示在清单文件中的任何函数声明之前。 清单文件可以包含任意数量的模块标签。

类别标签

与模块标签类似，类别标签标识所有后续函数和/或 COM 接口所属的“类别”。 例如，如果要创建Comctl32.dll清单文件，则可以将以下内容用作类别标签：

```
C++

category CommonControls:
```

清单文件可以包含任意数量的类别标签。

函数声明

函数声明是实际提示 Logger 记录内容的内容。 它几乎与 C/C++ 头文件中的函数原型完全相同。 格式有一些值得注意的新增功能，以下示例可以最好地说明这一点：

```
C++

HANDLE [gle] FindFirstFileA(
       LPCSTR lpFileName,
       [out] LPWIN32_FIND_DATAA lpFindFileData);
```

函数 FindFirstFileA 采用两个参数。 第一个是 *lpFileName*，它是一个完整路径，（通常使用通配符）定义搜索文件的位置。 第二个是指向WIN32_FIND_DATAA结构的指针，该结构将用于包含搜索结果。 返回的 HANDLE 用于将来对 **FindNextFileA** 的调用。 如果 **FindFirstFileA** 返回INVALID_HANDLE_VALUE，则函数调用失败，可以通过调用 **GetLastError** 函数获取错误代码。

HANDLE 类型声明如下：

```
C++

value DWORD HANDLE
{
#define NULL                         0 [fail]
#define INVALID_HANDLE_VALUE        -1 [fail]
};
```

如果此函数返回的值为 0 或 -1 (0xFFFFFFFF)，则 Logger 将假定该函数失败，因为此类值在值声明中具有 [fail] 修饰符。 （请参阅本节后面的“值类型”部分。）由于函数名称前面有 [gle] 修饰符，Logger 识别此函数使用 **GetLastError** 返回错误代码，因此它会捕获错误代码并将其记录到日志文件。

lpFindFileData 参数上的 [out] 修饰符通知记录器数据结构由函数填充，应在函数返回时记录数据结构。

COM 接口定义

COM 接口基本上是可由 COM 对象的客户端调用的函数向量。 清单格式大量借用 COM 中用于定义接口的接口定义语言 (IDL)。

请考虑以下示例：

```
C++

interface IDispatch : IUnknown
{
    HRESULT GetTypeInfoCount( UINT pctinfo  );
```

```
    HRESULT GetTypeInfo(
        UINT iTInfo,
        LCID lcid,
        LPVOID ppTInfo );

    HRESULT GetIDsOfNames(
        REFIID riid,
        LPOLECHAR* rgszNames,
        UINT cNames,
        LCID lcid,
        [out] DISPID* rgDispId );

    HRESULT Invoke(
        DISPID  dispIdMember,
        REFIID  riid,
        LCID  lcid,
        WORD  wFlags,
        DISPPARAMS*  pDispParams,
        VARIANT*  pVarResult,
        EXCEPINFO*  pExcepInfo,
        UINT*  puArgErr );
};
```

这会声明一个名为 **IDispatch** 的接口，该接口派生自 **IUnknown**。 它包含四个成员函数，这些函数在接口的大括号内按特定顺序声明。 记录器将通过替换接口 vtable 中的函数指针来截获和记录这些成员函数，（运行时使用的函数指针的实际二进制向量，）替换为自己的函数指针。 有关 Logger 如何在分发接口时捕获接口的详细信息，请参阅本节后面的 COM_INTERFACE_PTR 类型部分。

类型定义

定义数据类型是清单文件开发中最重要的（和最繁琐）部分。 清单语言允许你为传入或从函数返回的数值定义用户可读的标签。

例如，Winerror.h 定义一个名为“WinError”的类型，该类型是大多数 Microsoft Win32 函数返回的错误值及其相应的用户可读标签的列表。 这允许 Logger 和 LogViewer 将非信息错误代码替换为有意义的文本。

还可以标记位掩码内的单个位，以允许 Logger 和 LogViewer 将 DWORD 位掩码分解为其组件。

清单支持 13 种基本类型。 下表中列出了这些字段。

类型	Length	显示示例
指针	4 个字节	0x001AF320

类型	Length	显示示例
VOID	0 字节	
BYTE	1 个字节	0x32
WORD	2 个字节	0x0A23
DWORD	4 个字节	-234323
BOOL	1 个字节	**TRUE**
LPSTR	长度字节加上任意数量的字符	“快速棕色狐狸”
LPWSTR	长度字节加上任意数量的 Unicode 字符	“跳过懒狗”
GUID	16 个字节	{0CF774D0-F077-11D1-B1BC-00C04F86C324}
COM_INTERFACE_PTR	4 个字节	0x0203404A
值	依赖于基类型	ERROR_TOO_MANY_OPEN_FILES
掩码	依赖于基类型	WS_MAXIMIZED |WS_ALWAYSONTOP
struct	取决于封装类型的大小	+ lpRect nLeft 34 nRight 54 nTop 100 nBottom 300

清单文件中的类型定义的工作方式类似于 C/C++ typedef。 例如，以下语句将 PLONG 定义为指向 LONG 的指针：

```
C++

typedef LONG *PLONG;
```

已在 Main.h 中声明了大多数基本 typedef。 只需添加特定于组件的 typedef。 结构定义的格式与 C/C++ 结构类型相同。

有四种特殊类型：值、掩码、GUID 和COM_INTERFACE_PTR。

值类型
值是按用户可读标签划分的基本类型。 大多数函数文档仅引用函数中使用的特定常量的 **#define** 值。 例如，大多数程序员不知道 **GetLastError** 返回的所有代码的实际值是什么，因此在 LogViewer 中看到一个神秘的数值毫无帮助。 清单值通过允许值声明来克服此问题，如以下示例所示：

```
C++
```

```
value LONG ChangeNotifyFlags
{
#define SHCNF_IDLIST      0x0000        // LPITEMIDLIST
#define SHCNF_PATHA       0x0001        // path name
#define SHCNF_PRINTERA    0x0002        // printer friendly name
#define SHCNF_DWORD       0x0003        // DWORD
#define SHCNF_PATHW       0x0005        // path name
#define SHCNF_PRINTERW    0x0006        // printer friendly name
};
```

这将声明派生自 LONG 的名为“ChangeNotifyFlags”的新类型。 如果将其用作函数参数，将显示人类可读别名，而不是原始数字。

掩码类型

与值类型类似，掩码类型是一种基本类型 (通常为 DWORD)，它针对每个具有意义的位划分为人类可读的标签。 请参见以下示例：

C++

```
mask DWORD DirectDrawOptSurfaceDescCapsFlags
{
#define DDOSDCAPS_OPTCOMPRESSED                 0x00000001
#define DDOSDCAPS_OPTREORDERED                  0x00000002
#define DDOSDCAPS_MONOLITHICMIPMAP              0x00000004
};
```

这会声明一个派生自 DWORD 的新类型，如果用作函数参数，则会在 LogViewer 中为用户细分各个值。 因此，如果值为 0x00000005，LogViewer 将显示：

C++

```
DDOSDCAPS_OPTCOMPRESSED | DDOSDCAPS_MONOLITHICMIPMAP
```

GUID 类型

GUID 是 COM 中广泛使用的 16 字节全局唯一标识符。 它们通过两种方式声明：

C++

```
struct __declspec(uuid("00020400-0000-0000-C000-000000000046")) IDispatch;
```

或

C++

```
class __declspec(uuid("11219420-1768-11D1-95BE-00609797EA4F"))
ShellLinkObject;
```

第一种方法用于 (IID) 声明接口标识符。 LogViewer 显示时，“IID_”将追加到显示名称的开头。 第二种方法用于 (CLSID) 声明类标识符。 LogViewer 将“CLSID_”追加到显示名称的开头。

如果 GUID 类型是函数的参数，则 LogViewer 会将该值与所有声明的 IID 和 CLSID 进行比较。 如果找到匹配项，它将显示 IID 友好名称。 否则，它将以标准 GUID 表示法显示 32 个十六进制字符值。

COM_INTERFACE_PTR类型

COM_INTERFACE_PTR类型是 COM 接口指针的基类型。 声明 COM 接口时，实际上是在定义派生自 COM_INTERFACE_PTR 的新类型。 因此，指向此类类型的指针可以是函数的参数。 如果COM_INTERFACE_PTR基本类型声明为函数的 OUT 参数，并且有一个单独的参数具有 [iid] 标签，则记录器会将在 IID 中传递的 与所有已声明的 GUID 进行比较。 如果存在匹配项，并且声明了与 IID 同名的 COM 接口，记录器将挂钩该接口中的所有函数并记录它们。

以下是示例：

```
C++

STDAPI CoCreateInstance(
  REFCLSID rclsid,     //Class identifier (CLSID) of the object
  LPUNKNOWN pUnkOuter, //Pointer to controlling IUnknown
  CLSCTX dwClsContext, //Context for running executable code
  [iid] REFIID riid,   //Reference to the identifier of the interface
  [out] COM_INTERFACE_PTR * ppv
                       //Address of output variable that receives
                       //the interface pointer requested in riid
);
```

在此示例中， *riid* 具有 [iid] 修饰符。 这向 Logger 指示 *ppv* 中返回的指针是由 *riid* 标识的接口的 COM 接口指针。

还可以声明函数，如下所示：

```
C++

DDRESULT DirectDrawCreateClipper( DWORD dwFlags, [out] LPDIRECTDRAWCLIPPER
*lplpDDClipper, IUnknown *pUnkOuter );
```

在此示例中，LPDIRECTDRAWCLIPPER 定义为指向 **IDirectDrawClipper** 接口的指针。 由于 Logger 可以标识 *lplpDDClipper* 参数中返回的接口类型，因此不需要对任何其他参数使用 [iid] 修饰符。

PLMDebug

项目 • 2024/07/08

PLMDebug.exe是一种工具，可用于使用 Windows 调试器调试进程生命周期管理（PLM）下运行的 Windows 应用。 使用 PLMDebug，可以手动控制暂停、恢复和终止 Windows 应用。

使用 Windows 10 版本 1607 或更高版本提示，可以使用 UWP 命令（如 .createpackageapp）调试 UWP 应用。 有关详细信息，请参阅 使用 WinDbg 调试 UWP 应用。

在何处获取 PLMDebug

PLMDebug.exe包含在 Windows 调试工具中。

控制台

```
plmdebug /query [Package]
plmdebug /enableDebug Package [DebuggerCommandLine]
plmdebug /terminate Package
plmdebug /forceterminate Package
plmdebug /cleanterminate Package
plmdebug /suspend Package
plmdebug /resume Package
plmdebug /disableDebug Package
plmdebug /enumerateBgTasks Package
plmdebug /activateBgTask "{TaskID}"
```

参数

包
包的全名或正在运行的进程 ID。

DebuggerCommandLine
用于打开调试器的命令行。 命令行必须包含调试器的完整路径。 如果路径有空格，则必须用引号括起来。 命令行还可以包含参数。 以下是一些示例：

```
"C:\Program Files (x86)\Windows Kits\8.0\Debuggers\x64\WinDbg.exe"
```

```
"\"C:\Program Files\Debugging Tools for Windows (x64)\WinDbg.exe\" -server npipe:pipe=test"
```

/query [*包*]
显示已安装包的运行状态。 如果未 *指定包*，此命令将显示所有已安装包的运行状态。

/enableDebug *包* [*DebuggerCommandLine*]
递增包的调试引用计数。 如果包具有非零调试引用计数，则包不受 PLM 策略的豁免。 对 **/enableDebug** 的每个调用都必须与对 **/disableDebug** 的调用配对。 如果指定 *DebuggerCommandLine*，则启动包中的任何应用时，调试器将附加。

/终止 *包*
终止包。

/forceTerminate *包*
强制终止包。

/cleanTerminate *包*
暂停并终止包。

/暂停 *包*
挂起包。

/恢复 *包*
恢复包。

/disableDebug *包*
递减包的调试引用计数。

/enumerateBgTasks *包*
枚举包的后台任务 ID。

/activateBgTask"{*TaskId*}"
激活后台任务。 请注意，并非所有后台任务都可以使用 PLMDebug 激活。 TaskID 必须用大括号和引号括起来。 例如：

```
plmdebug.exe /activatebgtask "{29421c11-1e1a-47a4-9121-949ce9e25456}"
```

注解

在调用任何挂起、恢复或终止函数之前，必须调用 **plmdebug /enableDebug** 。

PLMDebug 工具调用 IPackageDebugSettings 接口的方法。 通过此接口，你可以手动控制应用的流程生命周期管理。 通过此接口（因此，通过此工具），可以暂停、恢复和终止 Windows 应用。 请注意，IPackageDebugSettings 接口的方法适用于整个包。 暂停、恢复和终止会影响包中当前正在运行的所有应用。

示例

示例 1

启动应用时附加调试器

假设你有一个名为 MyApp 的应用，该应用位于名为 MyApp_1.0.0.0_x64__tnq5r49etfg3c 的包中。 通过显示所有已安装软件包的全名和运行状态来验证软件包是否已安装。 在命令提示符窗口中输入以下命令。

plmdebug /query

控制台

```
Package full name: 1daa103b-74e1-426d-8193-
b6bc7ed66fed_1.0.0.0_x86__tnq5r49etfg3c
Package state: Terminated

Package full name: 41fb5f27-7b60-4f5e-8459-
803673131dd9_1.0.0.0_x86__tnq5r49etfg3c
Package state: Suspended
...
Package full name: MyApp_1.0.0.0_x64__tnq5r49etfg3c
Package state: Terminated
...
```

递增包的调试引用计数，并指定在启动应用时希望 WinDbg 附加。

plmdebug /enableDebug MyApp_1.0.0.0_x64__tnq5r49etfg3c "C：\Program Files（x86）\Windows Kits\8.0\Debuggers\x64\WinDbg.exe"

启动应用时，WinDbg 将附加并中断。

完成调试后，分离调试器。 然后递减包的调试引用计数。

plmdebug /disableDebug MyApp_1.0.0.0_x64__tnq5r49etfg3c

示例 2

将调试器附加到正在运行的应用

假设要将 WinDbg 附加到已在运行的 MyApp。 在 WinDbg 的**文件**菜单中，选择**附加到进程**。 记下 MyApp 的进程 ID。 假设进程 ID 为 4816。

递增包含 MyApp 的包的调试引用计数。

plmdebug /enableDebug 4816

在 WinDbg 的 "**附加到进程** "对话框中，选择进程 4816，然后选择" **确定**"。 WinDbg 将附加到 MyApp。

完成 MyApp 调试后，分离调试器。 然后递减包的调试引用计数。

plmdebug /disableDebug 4816

示例 3

手动暂停和恢复应用

假设你想要手动暂停和恢复应用。 首先，递增包含应用的包调试引用计数。

plmdebug /enableDebug MyApp_1.0.0.0_x64__tnq5r49etfg3c

挂起包。 应用的挂起处理程序会被调用，这对于调试非常有用。

plmdebug /suspend MyApp_1.0.0.0_x64__tnq5r49etfg3c

调试完成后，继续运行包。

plmdebug /resume MyApp_1.0.0.0_x64__tnq5r49etfg3c

最后，递减包的调试引用计数。

plmdebug /disableDebug MyApp_1.0.0.0_x64__tnq5r49etfg3c

示例 4

手动激活后台任务

若要手动激活后台任务进行调试，可以查询已注册的后台任务列表，然后通过 plmdebug 激活它。

首先查询已注册的后台任务集：

plmdebug /enumeratebgtasks MyApp_1.0.0.0_x64__tnq5r49etfg3c

控制台

```
Package full name is MyApp_1.0.0.0_x64__tnq5r49etfg3c.
Background Tasks:
SampleTask : {50DB0363-D722-4E23-A18F-1EF49B226CC3}
```

如果要保证任务激活，请先启用调试模式。 例如，当系统处于节电模式时，时间Trigger激活的任务等机会性任务将不会激活。 在包上启用调试模式可确保系统忽略阻止激活的策略。

plmdebug /enabledebug MyApp_1.0.0.0_x64__tnq5r49etfg3c

然后使用所枚举的注册 GUID 激活所需的任务。

plmdebug /activatebgtask “{50DB0363-D722-4E23-A18F-1EF49B2226CC3}”

另请参阅

如何在 Visual Studio 中调试 UWP 应用时触发暂停、恢复和后台事件

Windows 调试工具中包含的工具

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

远程工具

项目 • 2023/04/14

远程工具（Remote.exe）是一种命令行工具，可用于从远程计算机运行和控制任何控制台程序。

在何处获取远程工具

Remote.exe包含在 Windows 调试工具中。

远程工具组件

远程工具包括以下组件：

- 启动控制台程序并为客户端连接打开命名管道的服务器应用程序。
- 与服务器建立连接的客户端应用程序。 在客户端计算机上键入的命令将发送到服务器上的控制台应用程序，远程客户端显示来自服务器的控制台窗口的输出。
- 列出在服务器计算机上运行的远程会话的查询功能。

使用远程工具，可以在一台计算机上启动多个服务器会话，其中多个客户端可以连接到每个会话。 会话仅受计算机资源的限制。

这是一个较旧的工具，已被更复杂的工具（主要是远程桌面）所掩盖。 但是，由于它很简单，并且使用的资源很少，因此它仍然广泛使用，通常用于远程调试。

远程工具要求在本地和远程计算机上提交命令。 因此，若要在没有本地用户的计算机上使用此工具，必须开发另一种方法来提交命令并在必要时重新启动计算机。

远程工具可能会损害计算机上的安全性，因为它不对用户进行身份验证或使用 Microsoft Windows 权限。 默认情况下，可以连接并提供会话名称的任何远程计算机都可以使用此工具创建的命名管道，但你可以使用远程工具选项来包括和排除特定的用户和组。

本节包括：

远程工具的概念

远程工具命令

远程工具示例

另请参阅

Windows 调试工具中包含的工具

远程工具的概念

项目 • 2023/06/15

远程工具中使用以下概念。

客户端和服务器

远程工具使用客户端-服务器范例，避免使用 *本地* 和 *远程*一词，当存在多个用户和多台计算机时，这些相对术语可能会造成混淆。

在客户端和服务器计算机上键入的命令会显示在两台计算机的命令提示符窗口中。

服务器

*服务器*是运行控制台程序的计算机。 *远程服务器*是在服务器上运行的远程工具的实例。 服务器建立远程会话并将其命名 (名为管道) ，发出命令以启动控制台程序，并确定允许谁连接到会话。

客户端

*客户端*是将命令发送到控制台程序的远程计算机。 *远程客户端*是在客户端计算机上运行的远程工具的实例。 客户端连接到服务器建立的远程会话，并使用远程会话 (服务器创建的名为管道) 将命令发送到服务器上运行的控制台程序。

远程工具在每个远程会话中支持多个客户端。 每个客户端运行一个远程客户端。 所有客户端都可以将命令发送到服务器上运行的控制台程序，并且所有客户端都可以看到发送的命令和显示的输出。

可见会话

当远程会话 *可见*时，它们将显示在计算机上的远程会话列表中。 若要显示列表，请使用 **/q**) (**远程服务器查询命令**。

默认情况下，只有调试器会话可见，即 *Command* 参数的值包含单词 **kd**、 **dbg**、 **remoteds**、 **ntsd** 或 **cdb** 的会话;否则，会话不可见。 *Command* 参数是服务器上的远程工具命令的一部分。

若要使会话可见，请将 **/v** 参数添加到 **远程服务器** 命令。 若要使调试器会话不可见，请将 **/-v** 参数添加到 命令。

有关远程服务器查询命令的帮助，请参阅 **远程服务器查询命令**。

远程工具命令

项目 • 2023/06/15

远程工具为会话的客户端和服务器端提供单独的命令语法。 必须先输入用于启动服务器会话的命令，因为它们会建立客户端连接到的通信管道。

远程工具还有一个查询命令和一组单独的命令，用于与远程工具 (通信，而不是远程会话期间) 控制台程序。

以下主题介绍了这些命令：

远程服务器语法

远程客户端语法

远程服务器查询命令

远程会话命令

远程服务器语法

项目 • 2024/07/08

若要启动远程工具的服务器端，请在命令行中使用以下语法。

控制台

```
remote /s Command SessionName [/f Color] [/b] [/u User [/u User...]] [/ud
User [/ud User...]] [/v | /-v]
```

参数

/s
启动服务器会话。

命令
指定启动基于控制台的程序的命令。 此命令可以包含参数。 如果命令包含空格，请用引号将其括起来。

SessionName
将名称分配给远程会话。 如果名称包含空格，请将其用引号括起来。 此参数不区分大小写。

/f
指定服务器命令窗口中文本的颜色。

/b
指定服务器命令窗口的背景色。

颜色
指定颜色。 有效值为黑色、蓝色、绿色、青色、红色、紫色、黄色、白色、lblack、lblue、lgreen、lred、lpurple、lyellow 和 lwhite。

/u
指定允许连接到远程会话的用户或组;默认情况下，允许所有人。 使用此参数时，除用户*子参数指定的*用户和组之外，每个人都被拒绝权限。

/ud
指定拒绝连接到远程会话的权限的用户或组;默认情况下，没有人被拒绝权限。

用户
以 [*域* | *计算机*\\{*用户组* | } 格式指定用户或组的名称。 在本地计算机上指定用户或组时，

省略计算机名称。

/v
使会话可见。 有关详细信息，请参阅 “可见会话”。

默认情况下，只有调试器会话可见，即 Command 参数的值*包含单词* ***kd***、*dbg*、*remoteds*、***ntsd*** *或* ***cdb****;否则，会话不*可见。

-v
使远程调试器会话不可见。 有关详细信息，请参阅 “可见会话”。

备注

Command 和 *SessionName* 参数必须按语法行上显示的顺序显示。

若要结束远程会话，请键入 **@k**。 有关详细信息，请参阅 远程会话命令。

远程工具不会创建当前用户无权加入的会话。

在单个计算机上启动多个远程会话时，为每个会话打开一个新的命令窗口。 此外，对每个会话使用不同的会话名称。 由于会话名称用于标记命名管道，因此它们必须在计算机上是唯一的。

示例用法

控制台

```
remote /s "i386kd -v" TestSession
remote /s "cmd" "My Remote Session" /f white /b black /u
Server01\Administrators
remote /s "ntsd -d -g -x" DebugIt /-v
remote /q Server01
```


反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

远程客户端语法

项目 • 2024/07/08

若要启动远程工具的客户端，请在命令行使用以下语法。

控制台

```
remote /c Server SessionName [/L Lines] [/f] [/b] [/k ColorFile]
```

参数

/c
将客户端连接到远程会话。

服务器
指定建立会话的服务器的计算机名称。

SessionName
指定远程会话的名称。 此参数不区分大小写。

/L *线*
指定发送到客户端计算机的控制台显示器中的行数。 默认值为 200 行。 *行* 是十进制数。

/f
指定服务器命令窗口中文本的颜色。

/b
指定服务器命令窗口的背景色。

颜色
指定颜色。 有效值为黑色、蓝色、绿色、青色、红色、紫色、黄色、白色、lblack、lblue、lgreen、lred、lpurple、lyellow 和 lwhite。

/k *ColorFile*
指示格式化文本文件的路径（可选）和名称，指定用于在客户端计算机上显示输出的颜色。

颜色文件将输出中的关键字与文本颜色相关联。 当关键字出现在输出行中时，远程工具会将关联的文本颜色应用于该行。 有关文件格式，请参阅“备注”。

/q *计算机*
显示指定计算机上可用的远程会话。 列表中仅显示可见会话。 （请参阅**远程服务器语法中的 /v**。

备注

*服务器*和 *SessionName* 参数必须按语法行上显示的顺序显示。

若要断开与远程会话的连接，请键入 **@q**。 有关详细信息，请参阅 远程会话命令。

关键字颜色文件。 关键字颜色文件的格式如下所示。 关键字解释器不区分大小写。

关键字或短语本身显示在行上。 与该关键字关联的颜色将单独显示在以下行中，如语法中所示：

```text
Keyword
TextColor[, BackgroundColor]
```

例如，以下文件指示 Remote 在浅红色背景上显示黑色文本中的单词“error”;以显示以浅蓝色（在默认背景上）包含单词“warning”的行，以及默认背景上以浅绿色显示短语“Windows Vista”的行。

```text
ERROR
black, lred
WARNING
lblue
Windows Vista
lgreen
```

示例用法

```控制台
remote /c Server01 TestSession
remote /c Domain1\ComputerA0 "cmd" "My Remote Session"
remote /c Server01 TestSession /L 50 /f black /b white /k c:\remote_file.txt
remote /q Server01
```

反馈

此页面是否有帮助? 是 否

远程服务器查询命令

项目 • 2023/06/15

若要显示本地或远程服务器上可用会话的列表，请使用以下语法。

控制台

```
remote /q Computer
```dbgcmd

## <span id="ddk_remote_server_query_command_dtools"></span><span
id="DDK_REMOTE_SERVER_QUERY_COMMAND_DTOOLS"></span>Parameters


<span id="________q______"></span><span id="________Q______"></span> **/q**
Query. Displays the visible remote sessions on the specified computer.

<span id="_______Computer______"></span><span id="_______computer______">
</span><span id="_______COMPUTER______"></span> *Computer*
Specifies the name of the computer. This parameter is required (even on the
local computer).

### <span id="comments"></span><span id="COMMENTS"></span>Comments

The query display includes only server sessions; it does not display client
connections to remote sessions.

The query display includes only visible sessions. There is no command to
display invisible sessions.

The query display includes all visible sessions, including those that
restrict users (**/u** and **/ud**). The user might not have permission to
connect to all of the sessions in the list.

When there are no remote sessions running on the server, the Remote tool
displays the following message:

```console
No Remote servers running on \\Computer
```

但是，当计算机上运行的唯一远程会话是不可见的远程会话时，远程工具将显示以下消息：

控制台

```
No visible sessions on server \\Computer
```

远程会话命令

项目 • 2024/07/08

使用以下命令在控制台会话期间与远程工具通信。

@H
显示服务器和客户端计算机上的会话命令。 适用于服务器和客户端计算机。

@M消息
显示会话中所有服务器和客户端计算机上的指定消息。 适用于服务器和客户端计算机。

@P消息
使用指定的消息在服务器计算机上生成一个弹出窗口。 在客户端计算机上，该消息将显示在命令窗口中。 适用于服务器和客户端计算机。

@Q
退出。 断开客户端计算机与会话的连接。 仅适用于客户端计算机。

@K
断开所有客户端的连接并结束远程会话。 仅适用于服务器计算机。

备注

有关示例，请参阅远程工具示例中的“使用会话命令”。

为了响应会话命令，远程工具将显示一个标签，其中包含执行命令的日期和时间。 所有事件的时间都显示在服务器计算机的时区中。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

远程工具示例

项目 • 2023/06/15

本节中的示例演示了远程工具的使用，并演示了示例输入和输出。

基本服务器命令

以下命令在计算机上启动远程会话。

命令使用 **/s** 参数来指示服务器端命令。 它使用命令 **cmd** 启动 Windows 命令行界面 (Cmd.exe)，并将会话命名为 **test1**。

控制台

```
remote /s cmd test1
```

作为响应，远程工具启动会话并显示客户端将用于连接到会话的命令。

控制台

```
**************************************
***********     REMOTE    ************
***********     SERVER    ************
**************************************
To Connect: Remote /C SERVER06 "test1"

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
```

基本客户端命令

以下命令连接到 Server01 计算机上的远程会话。 命令使用 **/c** 参数来指示客户端命令。 它指定服务器计算机的名称 **Server01**，以及该计算机上的会话名称 **test1**。

控制台

```
remote /c server01 test1
```

作为响应，远程工具会显示一条消息，报告客户端计算机已连接到服务器计算机上的会话。 该消息显示服务器计算机和本地用户的名称 (**Server04 user1**) 。

控制台

```
************************************
***********     REMOTE    ************
***********     CLIENT    ************
************************************
Connected...

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Program Files\Debugging Tools for Windows>
**Remote: Connected to SERVER04 user1 [Tue 9:39 AM]
```

客户端连接到服务器后，在客户端和服务器计算机上的命令提示符下键入的命令会显示在两个显示器上。

例如，如果在客户端计算机的命令提示符处键入 **dir**，目录显示将显示在客户端和服务器计算机上的命令提示符窗口中。

使用服务器选项

以下服务器端命令启动与 NTSD 调试器的远程会话。

命令使用 **/s** 参数来指示服务器端命令。 下一个参数 “**ntsd -d -v**”是启动调试器的控制台命令以及调试器选项。 由于控制台命令包含空格，因此它用引号引起来。 命令包含会话的名称 **debugit**。

命令使用 **/u** 参数仅允许计算机的管理员和特定用户（Domain01 中的 User03）连接到会话。 它使用 **/f** 和 **/b** 选项在白色背景上指定黑色文本 (前景) 。

最后，该命令使用 **/-v** 参数使会话对用户查询不可见。 默认情况下，调试器会话可见。

控制台

```
remote /s "ntsd -d -v" DebugIt /u Administrators /u Domain01\User03
/f black /b white /-v
```

作为响应，远程工具创建名为 DebugIt 的会话，并使用指定的参数启动 NTSD。 该消息指示只有指定的用户有权进行连接。 它还会将命令窗口更改为指定的颜色。

控制台

```
************************************
***********     REMOTE    ************
***********     SERVER    ************
************************************

Protected Server!  Only the following users or groups can connect:
```

```
    Administrators
    Domain01\User03
To Connect: Remote /C SERVER06 "debugit"

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
```

使用客户端选项

以下命令使用在上一示例中启动的 NTSD 调试器连接到远程会话。

命令使用 **/c** 参数来指示客户端命令。 它指定服务器计算机的名称 **server06** 和远程会话的名称 **debugit**。

该命令还包括 **/k** 参数，用于指定关键字 (keyword) 颜色文件的位置。

控制台

```
remote /c server06 debugit /k c:\remote_client.txt
```

颜色文件包含以下文本：

控制台

```
Registry
white, blue
Token
red, white
```

此文本指示远程工具显示带有“registry”一词的输出行，（蓝色背景上的白色文本中不区分大小写），并在白色背景上以红色文本显示带有“token”一词的输出行。

作为响应，远程工具将客户端连接到服务器会话，并显示以下消息。

控制台

```
**************************************
***********     REMOTE    ************
***********     CLIENT    ************
**************************************
Connected...

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
```

客户端现在可以向服务器计算机上的 NTSD 调试器发送命令。 命令的输出同时显示在客户端和服务器计算机上。

在客户端计算机上，带有“registry”一词的输出行在蓝色背景上以白色文本显示，在白色背景上以红色文本显示带有“内核”一词的输出行。

查询会话

远程工具包含一个查询参数 (**/q**)，用于显示特定计算机上的远程会话列表。 该显示仅包括可见会话 (启动的调试器会话（没有 **/-v** 参数）和以 **/v** 参数) 启动的非调试器会话。

可以从服务器或客户端计算机查询会话。 即使在本地计算机上查询会话，也必须指定计算机名称。

以下命令查询本地计算机 **Server04** 上的会话。

控制台

```
remote /q Server04
```

作为响应，远程工具报告本地计算机上没有远程会话运行。

控制台

```
Querying server \\Server04
No Remote servers running on \\Server04
```

相比之下，为了响应有关其他计算机上的会话的查询 **Server06**，远程工具会列出在该计算机上运行的会话。

控制台

```
Querying server \\Server06

Visible sessions on server Server06:

ntsd                            [Remote /C SERVER06 "debug"] visible
cmd                             [Remote /C SERVER06 "test"] visible
```

显示将列出可见会话、这些会话上运行的控制台程序 (NTSD 和命令提示符窗口)，以及连接到会话的命令。 会话名称以引号显示在命令语法中。

显示不显示为这些会话建立的权限（如果有）。 因此，显示内容可能包括您无权加入的会话。

使用会话命令

在远程会话期间，可以随时使用远程会话命令。

以下命令将消息发送到连接到会话的所有计算机。

控制台

```
@M I think I found the problem.
```

因此，该消息将显示在会话中所有计算机的命令提示符窗口中。 该消息包括计算机名以及消息的日期和时间。

控制台

```
@m I think I found the problem.     [SERVER01       Wed 11:53 AM]
```

从服务器计算机发送消息时，标签中会显示“Local”而不是计算机名称。

控制台

```
@m I think I found the problem.     [Local       Wed 11:52 AM]
```

以下命令生成显示在服务器计算机上的弹出消息。 在会话中的所有客户端计算机上，它会将消息写入命令提示符窗口。

控制台

```
@P Did you see that?
```

在客户端计算机上，弹出消息显示在命令窗口中。

控制台

```
From SERVER02  [Wed 11:58 AM]

 Did you see that?
```

消息标签中显示的时间始终是服务器计算机上的时间，即使发送消息的客户端计算机处于不同的时区也是如此。

结束远程会话

以下示例演示如何使用远程会话命令断开客户端计算机与会话的连接并结束远程会话。 只有启动远程会话的服务器计算机才能终止远程会话。

若要断开客户端计算机与远程会话的连接，请在客户端计算机上键入 **@q**。

作为响应，断开连接的客户端计算机上将显示以下消息。

控制台

```
*** SESSION OVER ***
```

在会话中的所有其他计算机上，远程工具会发布一条消息，其中包含断开连接的计算机和用户的名称，以及断开连接的日期和时间。

控制台

```
**Remote:  Disconnected from SERVER04 User01  [Wed 12:01 PM]
```

若要结束远程会话，请在服务器计算机上键入 **@k**。 此命令会自动断开客户端的连接，然后结束会话。

TList

项目 • 2023/06/15

TList (任务列表查看器) Tlist.exe 是一个命令行工具，用于显示在本地计算机上运行的进程以及每个进程的有用信息。

TList 显示：

- 计算机上运行的所有进程及其进程 ID (PID) 。
- 显示创建每个进程的树。
- 进程的详细信息，包括其虚拟内存使用情况和启动进程的命令。
- 在每个进程中运行的线程，包括其线程 ID、入口点、上次报告的错误和线程状态。
- 在每个进程中运行的模块，包括模块的版本号、属性和虚拟地址。

可以使用 TList 按名称或 PID 搜索进程，或查找已加载指定模块的所有进程。

在 Windows XP 及更高版本的 Windows 中，TList 已替换为 TaskList (Tasklist.exe) ，如帮助和支持这些系统中所述。 TList 包含在 Windows 调试工具中，以支持无权访问 TaskList 的用户。

本节包括：

TList 命令

TList 示例

TList 命令

项目 • 2024/07/08

TList 命令的语法如下所示：

```
dbgcmd

tlist [/p ProcessName | PID | Pattern | /t | /c | /e | /k | /m [Module] | /s
| /v
```

参数

tlist
如果没有其他参数，TList 将显示所有正在运行的进程、其进程标识符（PID），以及运行它们的窗口的标题（如果有）。

/p *ProcessName*
显示指定进程的进程标识符（PID）。

ProcessName 是进程的名称（带或不带文件扩展名），而不是模式。

如果 ProcessName *的值*与任何正在运行的进程不匹配，则 TList 将显示 -1。 如果它匹配多个进程名称，则 TList 仅显示第一个匹配进程的 PID。

PID
显示有关 PID 指定的进程的详细信息。 有关显示的信息，请参阅下面的“备注”部分。 若要查找进程 ID，请键入 **不包含其他参数的 tlist** 。

模式
显示有关名称或窗口标题与指定模式匹配的所有进程的详细信息。 模式可以是完整名称或正则表达式。

/t
显示一个任务树，其中每个进程显示为创建它的进程的子进程。

/c
显示启动每个进程的命令行。

/e
显示每个进程的会话标识符。

/k
显示每个进程中处于活动状态的 COM 组件。

/m *模块*
列出在其中加载指定 DLL 或可执行模块的任务。 模块可以是完整的模块名称或模块名称模式。

/s
显示每个进程中处于活动状态的服务。

/v
显示正在运行的进程的详细信息，包括进程 ID、会话 ID、窗口标题、命令行以及进程中运行的服务。

备注

在进程的详细显示（**tlist** *PID* 或 **tlist** *Pattern*），TList 将显示以下信息。

- 进程 ID、可执行名称、程序的友好名称。
- 当前工作目录（CWD）。
- 启动进程的命令行（CmdLine）。
- 当前虚拟地址空间值。
- 线程数。
- 进程中运行的线程列表。 对于每个线程，TList 显示线程 ID（TID）、线程正在运行的函数、入口点的地址、上次报告错误（如果有）的地址以及线程状态。
- 为进程加载的模块的列表。 对于每个模块，TList 显示模块的版本号、属性、虚拟地址和模块名称。

使用 **/e** 参数时，有效的会话标识符仅在以下条件下显示在进程列表中。 否则，会话标识符为零（0）。

- 在 Windows XP 上，必须启用快速用户切换，并且必须有多个用户连接到非控制台会话。
- 在 Windows Vista 上，默认情况下所有进程都与两个终端服务会话相关联，必须至少一个用户连接到非控制台会话。

反馈

此页面是否有帮助? 是 否

TList 示例

项目 • 2023/06/15

以下示例演示如何使用 TList。

最简单的 TList 命令 (tlist)

键入不带其他参数的 **tlist** 会显示正在运行的进程的列表、其进程 ID (PID)，以及它们正在运行的窗口的标题（如果有）。

控制台

```
c:\>tlist

   0 System Process
   4 System
 308 smss.exe
 356 csrss.exe
 380 winlogon.exe      NetDDE Agent
 424 services.exe
 436 lsass.exe
 604 svchost.exe
 776 svchost.exe
 852 spoolsv.exe
1000 clisvcl.exe
1036 InoRpc.exe
1064 InoRT.exe
1076 InoTask.exe
1244 WTTSvc.exe
1492 Sysparse_com.exe  OleMainThreadWndName
1980 explorer.exe      Program Manager
1764 launch32.exe      SMS Client User Application Launcher
1832 msmsgs.exe        MSBLNetConn
2076 ctfmon.exe
2128 ISATRAY.EXE       IsaTray
4068 tlist.exe
```

查找进程 ID (-p)

以下命令使用 **-p** 参数和进程名称查找Explorer.exe (资源管理器) 进程的进程 ID。

作为响应，TList 显示资源管理器进程的进程 ID 328。

控制台

```
c:\>tlist -p explorer
328
```

使用 PID 查找进程详细信息

以下命令使用运行 Explorer 的进程的进程 ID 来查找有关 Explorer 进程的详细信息。

控制台

```
c:\>tlist 328
```

作为响应，TList 显示资源管理器进程的详细信息，包括以下元素：

- 进程 ID、可执行文件名称、程序友好名称。
- 当前工作目录 (CWD) 。
- (CmdLine) 启动进程的命令行。
- 当前虚拟地址空间值。
- 线程数。
- 进程中运行的线程列表。 对于每个线程，TList 显示 (TID) 的线程 ID、线程运行的函数、入口点的地址、上一次报告错误的地址（（如果有任何））以及线程状态。
- 为进程加载的模块列表。 对于每个模块，TList 显示模块的版本号、属性、虚拟地址和模块名称。

下面是此命令生成的输出的摘录。

控制台

```
 328 explorer.exe      Program Manager
   CWD:     C:\Documents and Settings\user01\
   CmdLine: C:\WINDOWS\Explorer.EXE
   VirtualSize:    90120 KB   PeakVirtualSize:   104844 KB
   WorkingSetSize: 19676 KB   PeakWorkingSetSize: 35716 KB
   NumberOfThreads: 17
    332 Win32StartAddr:0x010160cc LastErr:0x00000008 State:Waiting
   1232 Win32StartAddr:0x70a7def2 LastErr:0x00000000 State:Waiting
   1400 Win32StartAddr:0x77f883de LastErr:0x00000000 State:Waiting
   1452 Win32StartAddr:0x77f91e38 LastErr:0x00000000 State:Waiting
   1484 Win32StartAddr:0x70a7def2 LastErr:0x00000006 State:Waiting
   1904 Win32StartAddr:0x74b02ed6 LastErr:0x00000000 State:Ready
   1948 Win32StartAddr:0x72d22ecc LastErr:0x00000000 State:Waiting
   ....  (thread data deleted here)
```

```
  6.0.2800.1106 shp  0x01000000  Explorer.EXE
  5.1.2600.1217 shp  0x77F50000  ntdll.dll
  5.1.2600.1106 shp  0x77E60000  kernel32.dll
  7.0.2600.1106 shp  0x77C10000  msvcrt.dll
  5.1.2600.1106 shp  0x77DD0000  ADVAPI32.dll
  5.1.2600.1254 shp  0x78000000  RPCRT4.dll
  5.1.2600.1106 shp  0x77C70000  GDI32.dll
  5.1.2600.1255 shp  0x77D40000  USER32.dll
  ....  (module data deleted here)
```

(模式) 查找多个进程

以下命令通过表示一个或多个进程的进程名称或窗口名称的正则表达式搜索进程。 在此示例中，命令搜索进程名称或窗口名称以“ino”开头的进程。

控制台

```
c:\>tlist ino*
```

作为响应，TList 显示Inorpc.exe、Inort.exe和Inotask.exe的进程详细信息。 有关输出的说明，请参阅上面的“使用 PID 查找进程详细信息”小节。

显示进程树 (/t)

以下命令显示一个树，表示计算机上运行的进程。 进程显示为创建它们的进程的子级。

控制台

```
c:\>tlist /t
```

生成的进程树如下所示。 除其他事项外，此树显示 System (4) 进程创建了Smss.exe进程，该进程创建了Csrss.exe、Winlogon.exe、Lsass.exe和Rundll32.exe。 此外，Winlogon.exe创建了Services.exe，从而创建了所有与服务相关的进程。

控制台

```
System Process (0)
System (4)
  smss.exe (404)
    csrss.exe (452)
    winlogon.exe (476) NetDDE Agent
      services.exe (520)
        svchost.exe (700)
        svchost.exe (724)
        svchost.exe (864)
```

```
        svchost.exe (888)
        spoolsv.exe (996)
        scardsvr.exe (1040)
        alg.exe (1172)
        atievxx.exe (1200) ATI video bios poller
        InoRpc.exe (1248)
        InoRT.exe (1264)
        InoTask.exe (1308)
        mdm.exe (1392)
        dllhost.exe (2780)
      lsass.exe (532)
      rundll32.exe (500)
explorer.exe (328) Program Manager
  WLANMON.exe (1728) TI Wireless LAN Monitor
  ISATRAY.EXE (1712) IsaTray
  cmmon32.exe (456)
  WINWORD.EXE (844) Tlist.doc - Microsoft Word
  dexplore.exe (2096) Platform SDK - CreateThread
```

按模块 (/m) 查找进程

以下命令查找计算机上运行的所有进程，以加载特定 DLL。

控制台

```
c:\>tlist /m
```

作为响应，TList 显示Inorpc.exe、Inort.exe和Inotask.exe的进程详细信息。 有关输出的说明，请参阅上面的“使用 PID 查找进程详细信息”小节。

UMDH

项目 • 2024/02/07

User-Mode 转储堆 (UMDH) 工具 Umdh.exe 分析给定进程的 Microsoft Windows 堆内存分配。 UMDH 具有以下模式。

- (“模式 1”) **分析正在运行的进程。** UMDH 捕获和分析进程的堆内存分配。 对于每个分配，UMDH 显示分配大小、开销大小、指向分配的指针和分配堆栈。 如果进程有多个活动内存堆，则 UMDH 会捕获所有堆。 此分析可以实时显示或保存在日志文件中。
- (“模式 2”) **分析 UMDH 日志文件。** UMDH 分析它之前创建的日志文件。 UMDH 可以比较在不同时间为同一进程创建的两个日志，并显示分配大小增加最大的调用。 此方法可用于查找内存泄漏。

本节包括：

准备使用 UMDH

UMDH 命令

解释 UMDH 日志

解释日志比较结果

准备使用 UMDH

项目 • 2024/07/08

在使用用户模式转储堆（UMDH）捕获进程的堆分配之前，必须先完成本节中所述的配置任务。 如果未正确配置计算机，UMDH 将不会生成任何结果，或者结果不完整或不正确。

创建用户模式堆栈跟踪数据库

在使用 UMDH 捕获进程的堆分配之前，必须将 Windows 配置为捕获堆栈跟踪。

若要为进程启用堆栈跟踪捕获，请使用 GFlags 为进程设置 **创建用户模式堆栈跟踪数据库** 标志。 这可以通过以下任一方法完成：

- 在 GFlags 图形界面中，选择“ **图像文件** ”选项卡。键入进程名称，包括文件扩展名（例如，Notepad.exe）。 **按 Tab** 键，选择“**创建用户模式堆栈跟踪数据库**”，然后选择“**应用**”。
- 或者，同样，请使用以下 GFlags 命令行，其中 *ImageName* 是进程名称（包括文件扩展名）：

  gflags /i *ImageName* **+ust**

默认情况下，Windows 收集的堆栈跟踪数据量限制为 x86 处理器上的 32 MB，x64 处理器上的堆栈跟踪数据数限制为 64 MB。 如果必须增大此数据库的大小，请在 **GFlags 图形界面中选择“图像文件** ”选项卡，键入进程名称，按 **TAB** 键，选中 **“堆栈回溯” (Megs)** 复选框，在关联的文本框中键入值（以 MB 为单位），然后选择“ **应用**”。

请注意 ，仅在必要时增加此数据库，因为它可能会耗尽有限的 Windows 资源。 当不再需要更大的大小时，将此设置返回到其原始值。

这些设置会影响程序的所有新实例。 它不会影响当前正在运行的程序实例。

访问必要的符号

在使用 UMDH 之前，必须有权访问应用程序的正确符号。 UMDH 使用环境变量 _NT_SYMBOL_PATH指定的符号路径。 将此变量设置为等于包含应用程序符号的路径。

如果还包括 Windows 符号的路径，则分析可能更完整。 此符号路径的语法与调试器使用的语法相同;有关详细信息，请参阅 符号路径。

例如，如果应用程序的符号位于 C：\MyApp\Symbols，并且已将 Windows 符号文件安装到 \\myshare\winsymbols，则可以使用以下命令设置符号路径：

控制台

```
set _NT_SYMBOL_PATH=c:\myapp\symbols;\\myshare\winsymbols
```

另一个示例是，如果应用程序的符号位于 C：\MyApp\Symbols，并且想要使用 Windows 符号的公共Microsoft符号存储（使用 C：\MyCache 作为下游存储），则可以使用以下命令设置符号路径：

控制台

```
set
_NT_SYMBOL_PATH=c:\myapp\symbols;srv*c:\mycache*https://msdl.microsoft.com/download/symbols
```

重要 说明：有两台计算机： *在其中创建 UMDH 日志的日志记录计算机* 和 *分析 UMDH 日志的分析计算机* 。 分析计算机上的符号路径必须指向生成日志时在日志记录计算机上加载的 Windows 版本的符号。 不要将分析计算机上的符号路径指向符号服务器。 如果这样做，UMDH 将检索在分析计算机上运行的 Windows 版本的符号，并且 UMDH 不会显示有意义的结果。

禁用 BSTR 缓存

自动化（以前称为 OLE 自动化）缓存 BSTR 字符串使用的内存。 这可以防止 UMDH 正确确定内存分配的所有者。 若要避免此问题，必须禁用 BSTR 缓存。

若要禁用 BSTR 缓存，请将 OANOCACHE 环境变量设置为 1（1）。 在启动要跟踪其分配的应用程序之前，必须进行此设置。

或者，可以通过调用 .NET Framework **SetNoOaCache** 函数从应用程序本身中禁用 BSTR 缓存。 如果选择此方法，应尽早调用此函数，因为调用 SetNoOaCache **时**已缓存的任何 BSTR 分配都将保持缓存状态。

如果需要跟踪服务进行的分配，必须将 OANOCACHE 设置为系统环境变量，然后重启 Windows，使此设置生效。

查找进程 ID

UMDH 通过其进程标识符（PID）来标识进程。 可以使用任务管理器、任务列表或 TList 查找任何正在运行的进程的 PID。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

UMDH 命令

项目 • 2023/06/15

UMDH 有两种不同的模式，每个模式都有自己的命令语法和参数：

- **分析正在运行的进程** - 显示进程中的所有堆分配。
- **分析 UMDH 日志** - 比较同一进程的两个不同时间生成的分配列表。 分析差异有助于识别内存泄漏。

分析正在运行的进程

项目 • 2024/07/08

使用以下命令记录和分析运行中进程的堆内存分配情况。 此分析侧重于堆栈跟踪。

```
dbgcmd
```

```
umdh -p:PID [-f:LogFile] [-v[:MsgFile]] | [-g] | [-h]
```

参数

-p： *PID*
指定要分析的进程。 *PID* 是进程的进程 ID。 此参数是必需的。

若要查找运行中进程的 PID，请使用任务管理器、任务列表或 TList。

-f： *LogFile*
将日志内容保存在文本文件中。 默认情况下，UMDH 会将日志写入 stdout（命令窗口）。

LogFile 指定文件的路径（可选）和名称。 如果指定了现有文件，UMDH 会覆盖文件。

请注意，如果未在管理员模式下启动 UMDH，或尝试写入“受保护的”路径，将被拒绝访问。

-v[： *MsgFile*]
详细模式。 生成详细的信息和错误消息。 默认情况下，UMDH 将这些消息写入 stderr。

MsgFile 指定文本文件的路径（可选）和名称。 使用此变量时，UMDH 会将详细消息写入指定文件，而不是 stderr。 如果指定了现有文件，UMDH 会覆盖文件。

-g
记录未被进程引用的堆块（“垃圾回收”）。

-h
显示帮助。

备注

在 Windows 2000 上，如果 UMDH 报告了查找堆栈跟踪数据库的错误，并且已在 GFlags 中启用了**创建用户模式堆栈跟踪数据库**选项，则可能存在符号文件冲突。 若要解决此问题，请将应用程序的 DBG 和 PDB 符号文件复制到同一目录，然后重试。

示例用法

dbgcmd

```
umdh -?
umdh -p:2230
umdh -p:2230  -f:dump_allocations.txt
umdh -p:2230 -f:c:\Log1.txt -v:c:\Msg1.txt
umdh -p:2230 -g -f:garbage.txt
```

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

分析 UMDH 日志

项目 • 2024/02/14

使用以下命令分析通过运行 UMDH 创建的用户模式转储堆（UMDH）日志，以及“分析正在运行的进程”**中所述的**语法。 此分析侧重于分配，而不是堆栈跟踪。

可以分析单个日志文件或比较不同运行中的日志，以检测程序或驱动程序的内存转储分配随时间推移的变化。

```
dbgcmd
```

```
umdh [-d] [-v] [-l] File1 [File2] [-h | ?]
```

参数

-d
以十进制数字显示数值数据。 默认值为十六进制。

-v
详细模式。 包括跟踪以及摘要信息。 分析单个日志文件时，跟踪最有用。

-l
在日志中包含文件名和行号。 （请注意，参数是小写字母“L”，而不是数字 1。

File1 [*File2*]
指定要分析的 UMDH 日志文件。

UMDH 在**分析正在运行的进程模式下运行日志文件**时创建日志文件，并将日志内容保存在文本文件 （**-f**） 中。

指定一个日志文件时，UMDH 会分析该文件，并在每个跟踪中按分配的字节数降序显示函数调用。

指定两个日志文件时，UMDH 会比较文件，并按降序显示函数调用，其分配在两个试验之间增长最大。

-h | ?
显示帮助。

示例用法

```
dbgcmd
```

```
umdh dump.txt
umdh -d -v dump.txt
umdh dump1.txt dump2.txt
```

注解

假设你有两台计算机： *在其中创建 UMDH 日志的日志记录计算机* ，以及 *分析 UMDH 日志的分析计算机* 。 分析计算机上的符号路径必须指向日志时在日志记录计算机上加载的 Windows 版本的符号。 不要将分析计算机上的符号路径指向符号服务器。 如果这样做，UMDH 将检索在分析计算机上运行的 Windows 版本的符号，UMDH 不会显示有意义的结果。

解释 UMDH 日志

项目 • 2023/06/15

User-Mode转储堆 (UMDH) 日志文件显示进程中的堆分配列表以及进行分配的堆栈。

此示例演示如何为 ID 为 1204 的进程生成日志。 日志将写入文件log1.txt。

控制台

```
umdh -p:1204 -f:log1.txt
```

日志文件不可读，因为符号未解析。 UMDH 在分析日志时解析符号。 此示例演示如何分析log1.txt并将结果存储在result.txt中。

控制台

```
umdh -v log1.txt  > result.txt
```

用于分析日志文件的符号文件

假设你有两台计算机：一 *台用于* 创建 UMDH 日志的日志记录计算机和一台分析 UMDH 日志的分析 *计算机* 。 分析计算机上的符号路径必须指向创建日志时在日志记录计算机上加载的 Windows 版本的符号。 不要将分析计算机上的符号路径指向符号服务器。 如果这样做，UMDH 将检索分析计算机上运行的 Windows 版本的符号，UMDH 将不会显示有意义的结果。

解释日志比较结果

项目 • 2023/06/15

随着时间的推移，可以生成多个User-Mode转储堆 (UMDH) 同一进程的日志。 然后，可以使用 UMDH 来比较日志，并确定在试用之间增长最大的调用堆栈分配。

例如，以下命令指示 UMDH 比较两个 UMDH 日志（Log1.txt和Log2.txt），并将输出重定向到第三个文件（Compare.txt）。

控制台

```
umdh -v Log1.txt Log2.txt > Compare.txt
```

生成的 Compare.txt 文件列出了每个日志中记录的调用堆栈，并为每个堆栈显示日志文件之间的堆分配更改。

例如，文件中的以下行显示了调用堆栈中标记为“Backtrace00053”的函数的分配大小更改。

在 Log1.txt 中，堆栈中的调用占 40，432 个 (0x9DF0) 字节，但在Log2.txt，相同的调用堆栈占 61，712 个 (0xF110) 字节，相差 21，280 (0x5320) 字节。

控制台

```
+ 5320 (f110 - 9df0) 3a allocs BackTrace00053
Total increase == 5320
```

下面是分配的堆栈：

控制台

```
ntdll!RtlDebugAllocateHeap+0x000000FD
ntdll!RtlAllocateHeapSlowly+0x0000005A
ntdll!RtlAllocateHeap+0x00000808
MyApp!_heap_alloc_base+0x00000069
MyApp!_heap_alloc_dbg+0x000001A2
MyApp!_nh_malloc_dbg+0x00000023
MyApp!_nh_malloc+0x00000016
MyApp!operator new+0x0000000E
MyApp!LeakyFunc+0x0000001E
MyApp!main+0x0000002C
MyApp!mainCRTStartup+0x000000FC
KERNEL32!BaseProcessStart+0x0000003D
```

对调用堆栈的检查显示 **LeakyFunc** 函数正在使用 Visual C++ 运行时库分配内存。 如果检查其他日志文件后发现分配随时间推移而增长，则可能能够得出未释放从堆分配的内存的结论。

用于分析日志文件的符号文件

假设你有两台计算机：一 *台用于* 创建 UMDH 日志的日志记录计算机，一 *台用于* 分析 UMDH 日志的分析计算机。 分析计算机上的符号路径必须指向创建日志时在日志记录计算机上加载的 Windows 版本的符号。 不要将分析计算机上的符号路径指向符号服务器。 如果这样做，UMDH 将检索分析计算机上运行的 Windows 版本的符号，UMDH 不会显示有意义的结果。

另请参阅

使用 UMDH 查找用户模式内存泄漏

Windows 中的通用串行总线查看器

项目 • 2024/08/05

通用串行总线查看器 (USBView) 或 usbview.exe 是一个 Windows 应用，可用于浏览计算机上的所有 USB 控制器和已连接的 USB 设备。 USBView 适用于所有版本的 Windows。

从何处获取 USBView

若要下载和使用 USBView，请完成以下步骤：

1. 下载并安装 Windows SDK。
2. 在安装过程中，仅选择“Windows 调试工具”框，然后清除所有其他框。
3. 默认情况下，在 x64 PC 上，SDK 会将 USBView 安装到以下目录。

   ```
   C:\Program Files (x86)\Windows Kits\10\Debuggers\x64
   ```

4. 打开正在运行的处理器类型的工具包调试程序目录，然后选择 usbview.exe 以启动该实用工具。

USBView 源代码

GitHub 上的 Windows 驱动程序示例 存储库中也提供了 USBView 。

使用 USBView

USBView 是一个示例应用程序，可枚举有关 USB 主机控制器、USB 集线器和附加 USB 设备的基本信息。 它还能从注册表中查询设备相关信息，并通过 USB 向设备发出请求。

主 USBView 窗口包含两个窗格。 左窗格显示了一个面向连接的树视图，你可以使用它来选择任何 USB 设备。

右窗格显示与所选 USB 设备有关的 USB 数据结构。 这些结构包括设备、配置、接口和端点描述符，以及当前设备的配置。

由于 USBView 是一款较老的应用程序，因此它可能无法显示较新 USB 的信息。 必要时应使用设备管理器等其他工具。 有关 USBView 用于查询信息的 IOCTL 的详细信息，请参阅 USBView 。

使用设备管理器显示 USB 信息

若要使用设备管理器显示 USB 信息，请执行以下操作：

1. 选择 Windows 徽标键+R，在弹出框中输入“devmgmt.msc”，然后选择 Enter。
2. 在设备管理器中，选择你的计算机，使其突出显示。
3. 选择“操作”，然后选择“扫描硬件更改”。
4. 选择“查看”，然后选择“显示隐藏的设备”，以显示更多设备（例如，当前未处于活动状态的设备）。
5. 在设备管理器中展开“通用串行总线控制器”节点，然后选择所讨论的设备。
6. 右键单击并选择“属性”以查看摘要设备状态信息。
7. 选择“详细信息”选项卡以查看详细信息。
8. 选择“属性”以查看详细信息，例如“状态”或“问题代码”。

常见 USB 问题故障排除

如果要诊断 USB 设备，请访问修复 Windows 中的 USB-C 问题 了解更多信息。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

与 Windows 调试工具相关的工具

项目 • 2023/04/14

本部分介绍与调试相关但未包含在 Windows 调试工具包中的工具。

应用程序验证工具

在应用程序运行时监视应用程序操作，使应用程序受到各种压力和测试，并生成有关应用程序执行或设计中潜在错误的报告。 应用程序验证程序包含在适用于 Windows 的 Windows 软件开发工具包 (SDK) 中。 有关下载和安装 Windows SDK 的信息，请参阅 Windows SDK。

Windows 错误报告

可以将 Windows 错误报告 (WER) 配置为在用户模式代码中发生异常和其他错误时编写用户模式转储文件。 WER 包含在 Windows Vista 和更高版本的 Windows 中。

另请参阅

- Windows 调试工具中包含的工具
- Windows 调试工具

Windows 错误报告

项目 • 2023/06/15

Windows 错误报告 (WER) 包含在 Windows Vista 及更高版本的 Windows 中。 可以将 WER 配置为在用户模式代码中发生异常和其他错误时编写用户模式转储文件。 有关详细信息，请参阅 启用事后调试 和 收集User-Mode转储。

WER 取代了 Windows XP 中包含的 Watson 博士。

源代码路径

项目 • 2023/11/15

*源路径*指定 C 和 C++ 源文件所在的目录。

如果要在生成可执行文件的计算机上调试用户模式进程，并且源文件仍位于其原始位置，则调试器可以自动找到源文件。

在大多数其他情况下，必须设置源路径或加载单个源文件。

通过调试器执行远程调试时，调试服务器将使用源路径。 如果使用 WinDbg 作为调试器，则每个调试客户端也有其自己的 *本地源路径*。 所有与源相关的命令都会访问本地计算机上的源文件。 必须在想要使用源命令的任何客户端或服务器上设置正确的路径。

此多路径系统还允许调试客户端使用与源相关的命令，而无需实际与其他客户端或服务器共享源文件。 如果某个用户有权访问的私有或机密源文件，此系统非常有用。

还可以随时加载源文件，而不考虑源路径。

源路径语法

调试器的源路径是一个字符串，由多个目录路径组成，用分号分隔。

支持相对路径。 但是，除非始终从同一目录启动调试器，否则应在每个路径之前添加驱动程序号或网络共享。 还支持网络共享。

注意 如果连接到公司网络，访问源文件的最有效方法是使用源服务器。 可以通过在源路径中使用 srv* 字符串来使用源服务器。 有关源服务器的详细信息，请参阅 使用源服务器。

控制源路径

若要控制源路径和本地源路径，可以执行下列操作之一：

- 在启动调试器之前，请使用 _NT_SOURCE_PATH 环境变量 来设置源路径。 如果尝试通过此环境变量添加无效目录，调试器将忽略此目录。
- 启动调试器时，使用 **-srcpath**命令行选项 设置源路径。
- 使用 **.srcpath (Set Source Path)** 命令显示、设置、更改或追加到源路径。 如果使用源服务器， **.srcfix (使用源服务器)** 稍微简单一些。

- (WinDbg 仅) 使用 **.lsrcpath (设置本地源路径)** 命令显示、设置、更改或追加到本地源路径。 如果使用源服务器， **.lsrcfix (使用本地源服务器)** 稍微简单一些。 还可以将 WinDbg Command-Line 与参数 -lscrpath 一起使用。 有关详细信息，请参阅 **WinDbg Command-Line 选项**。

- 仅) (WinDbg 使用 **[文件 |"源文件路径"** 命令或按 Ctrl+P 可显示、设置、更改或追加到源路径或本地源路径。

还可以通过执行下列操作之一直接打开或关闭源文件：

- 使用 **lsf (加载或卸载源文件)** 命令打开或关闭源文件。

- (WinDbg 仅) 使用 **.open (Open Source File)** 命令打开源文件。

- 仅 (WinDbg) 使用**文件 |开放源代码文件**命令或按 Ctrl+o 打开源文件。 还可以使用工具栏上的**开放源代码文件 (ctrl+o)** 按钮。

  注意 使用 **文件时 |打开源文件** (或其快捷菜单或按钮等效项) 打开源文件，该文件的路径会自动追加到源路径。

- 仅) 使用文件 (WinDbg **|"最近打开的文件"** 命令，用于打开最近在 WinDbg 中打开的四个源文件之一。

- 仅) 使用文件 (WinDbg **|关闭"当前窗口"**命令或选择"源"窗口一角的"**关闭**"按钮以关闭源文件。

另请参阅

有关如何使用源文件的详细信息，请参阅 在源模式下调试。

源代码扩展访问

项目 • 2023/08/10

从 WinDbg 版本 1.2104 开始，源路径命令 (.srcpath、 .lsrcpath (设置源路径)) 支持通过 `DebugInfoD*` 标记从 DebugInfoD 服务器 检索文件。

标记 `DebugInfoD*` 可以指向一个或多个 DebugInfoD 服务器，其中每个服务器 URL 的格式设置为 `https://domain.com` ，并由 `*`分隔。 将按源路径中列出的相同顺序搜索服务器，并从第一个匹配 URL 检索文件。

`DebugInfoD*`标记可以与 `srv*` 结合使用，以便确定从特定位置进行源检索的优先级。

某些符号文件包含有关源代码的校验和信息。 在这种情况下，将首先搜索源路径中的本地文件夹，查找文件名相同且校验和匹配的文件。 如果没有可用的校验和信息，或者找不到具有匹配名称和校验和的文件，则将按指定的顺序遍历搜索路径，如以下示例所示。

在此示例中，源路径可以使用 *DebugInfoD* ，如此处所示，它遵循 srv* 标记。

```
.srcpath srv*;DebugInfoD*url1*url2…*urlN;o:\src\folder
```

在此示例中，将按以下顺序搜索目标源代码位置：

- srv* (源链接版本 1 或版本 2)，
- 然后 debuginfoD urls： url1， url2， ... urlN
- lastly 本地文件夹 o：\src\folder

在此第二个示例中， *DebugInfoD* 标记使用了两次。

```
.srcpath DebugInfoD*url1;srv*;DebugInfoD*url2;o:\src\folder
```

对于第二个示例，搜索顺序为：

- *DebugInfoD* url1，
- 然后 srv* (源链接版本 1 或版本 2)，
- 然后 *DebugInfoD* url2
- lastly 本地文件夹 o：\src\folder

支持的源代码格式

标记`srv*`支持使用 Source Link 1.0 或 Source Link 2.0 自动检索文件，它不支持 *DebugInfoD* URL。

资源

.srcpath、.lsrcpath（设置源路径）

源路径

使用源服务器

源链接

ELFUTILS DEBUGINFOD

使用源服务器

项目 • 2023/06/15

源服务器使调试器能够自动检索与当前目标匹配的源文件。 若要使用源服务器，必须调试在生成时已编制源索引且其源文件位置嵌入 PDB 文件中的二进制文件。

Windows 调试工具包括源服务器 SrcSrv (Srcsrv.exe) 。

将 SrcSrv 与调试器配合使用

SrcSrv 可与 WinDbg、KD、NTSD 或 CDB 一起使用。

若要将 SrcSrv 与调试器配合使用，请输入以下命令将源路径设置为 srv*。

```
dbgcmd

.srcfix
```

可以通过输入以下命令获得相同的结果。

```
dbgcmd

.srcpath srv*
```

将源路径设置为 srv* 会告知调试器应从目标模块的符号文件中指定的位置检索源文件。

如果要使用 SrcSrv 并在源路径中包含目录列表，请使用分号与路径中的任何目录分开 `srv*` 。

例如：

```
dbgcmd

.srcpath srv*;c:\someSourceCode
```

如果源路径如前面的示例所示设置，则调试器首先使用 SrcSrv 从目标模块的符号文件中指定的位置检索源文件。 如果 SrcSrv 无法检索源文件，调试器会尝试从 c:\someSourceCode 检索它。 无论 srv* 是路径中的第一个元素还是稍后出现，调试器始终在搜索路径中列出的任何其他目录之前使用 SymSrv。

还可以使用 **.srcfix+** 追加 `srv*` 到现有源路径，如以下示例所示。

```
dbgcmd
```

```
3: kd> .srcpath c:\mySource
Source search path is: c:\mySource
3: kd> .srcfix+
Source search path is: c:\mySource;SRV*
```

如果源服务器检索了源文件，则调试会话结束后，该文件将保留在硬盘驱动器上。 源文件存储在主目录的 src 子目录中，（与符号服务器不同，源服务器不会在 `srv*` 语法本身）指定本地缓存。 主目录默认为调试器安装目录;可以使用 **！ homedir** 扩展或设置 DBGHELP_HOMEDIR环境变量来更改它。 如果此子目录尚不存在，则会创建它。

如果使用 **.open (Open Source File)** 命令通过 SrcSrv 打开新的源文件，则必须包含 -m Address 参数。

有关如何为源编制索引的信息，或者如果计划创建自己的源代码管理提供程序模块，请参阅 SrcSrv。

使用 AgeStore 减小缓存大小

调试会话结束后， SrcSrv 下载的任何源文件都将保留在硬盘驱动器上。 若要控制源缓存的大小，可以使用 AgeStore 工具删除早于指定日期的缓存文件，或将缓存的内容减少到指定大小以下。 有关详细信息，请参阅 AgeStore。

SrcSrv

项目 • 2023/06/15

SrcSrv 工具 (Srcsrv.dll) 使客户端能够检索用于生成应用程序的源文件的确切版本。 由于模块的源代码可能会因版本而异，因此，请务必查看生成有关模块版本时存在的源代码。

SrcSrv 从源代码管理检索相应的文件。 若要使用 SrcSrv，必须对应用程序进行源索引。

本节包括：

使用 SrcSrv

源索引

源代码管理系统

HTTP 站点和 UNC 共享

使用 SrcSrv

项目 • 2023/06/15

若要将 SrcSrv 与 WinDbg、KD、NTSD 或 CDB 配合使用，请验证是否已安装最新版本的 Windows 调试工具 包 (版本 6.3 或更高版本) 。 然后，在源路径中包括文本 `srv*` ，用分号分隔也位于源路径中的任何目录。

例如：

dbgcmd

```
.srcpath srv*;c:\someSourceCode
```

如果源路径如前面的示例所示设置，则调试器首先使用 SrcSrv 从目标模块的符号文件中指定的位置检索源文件。 如果 SrcSrv 无法检索源文件，调试器会尝试从 c:\someSourceCode 检索它。 无论 srv* 是路径中的第一个元素还是稍后出现，调试器始终在搜索路径中列出的任何其他目录之前使用 SymSrv。

如果 源文件由 SrcSrv 检索，则调试会话结束后，该文件将保留在硬盘驱动器上。 源文件存储在主目录的 src 子目录中， (与符号服务器不同，源服务器不会在 `srv*` 语法本身) 指定本地缓存。 主目录默认为适用于 Windows 的调试工具安装目录;可以使用 **！ homedir** 扩展或设置DBGHELP_HOMEDIR环境变量来更改它。 如果主目录的 src 子目录尚不存在，则会创建它。

调试 SrcSrv

如果在从调试器中提取源文件时遇到问题，请使用 -n 命令行参数启动调试器，以查看实际的源提取命令以及这些命令的输出。 ！ sym noisy 命令执行相同的操作，但你可能已经错过了以前尝试提取的重要信息。 这是因为调试器放弃尝试从似乎无法访问的版本控制存储库访问源。

检索源文件

如果使用 **.open (Open Source File)** 命令通过 SrcSrv 打开新的源文件，则必须包含 -m Address 参数。

为了便于从前面列出的调试器以外的工具使用 SrcSrv ， DbgHelp API 通过 **SymGetSourceFile** 函数提供对 SrcSrv 功能的访问权限。 若要检索要检索的源文件的名称，请调用 **SymEnumSourceFiles** 或 **SymGetLineFromAddr64** 函数。 有关 DbgHelp API 的更多详细信息，请参阅 dbghelp.chm 文档，该文档可以在 Windows 安装目录的调试工具的 sdk/help 子目录中找到，或参阅 调试帮助库。

使用 AgeStore 减小缓存大小

调试会话结束后， SrcSrv 下载的任何源文件都保留在硬盘驱动器上。 若要控制源缓存的大小，可以使用 AgeStore 工具删除早于指定日期的缓存文件，或将缓存内容减少到指定大小以下。 有关详细信息，请参阅 AgeStore。

源索引

项目 • 2023/06/15

通常，在生成应用程序后，在生成过程中对二进制文件进行源索引。 SrcSrv 所需的信息存储在 .pdb 文件中。 SrcSrv 目前支持以下源代码管理系统：

- Perforce
- Microsoft Team Foundation Server

还可以创建自定义脚本，为不同的源代码管理系统编制代码索引。 此包中包含 Subversion 的一个此类模块。

SrcSrv 包括源索引编制过程中使用的五个工具：

Srcsrv.ini 文件

Ssindex.cmd 脚本

SrcTool 实用工具

PDBStr 工具

这些工具随 Windows 调试工具一起安装在 srcsrv 子目录中，并且应与 SrcSrv 一起安装在同一路径中。 这些工具中的 PERL 脚本需要 PERL 5.6 或更高版本。

Srcsrv.ini 文件

项目 • 2023/06/15

Srcsrv.ini文件是所有源代码管理服务器的主列表。 每个条目均具有以下格式：

```ini
MYSERVER=ServerInfo
```

使用 Perforce 时， *ServerInfo* 由服务器的完整网络路径组成，后跟冒号，后跟其使用的端口号。 例如：

```ini
MYSERVER=machine.corp.company.com:1666
```

实际使用此包附带的模块对生成编制源索引时，Srcsrv.ini 是必需的文件。 此项创建用于描述服务器信息的别名。 对于支持的每个服务器，该值应是唯一的。

此文件也可以安装在运行调试器的计算机上。 当 SrcSrv 启动时，它会查看值Srcsrv.ini;这些值将覆盖 .pdb 文件中包含的信息。 这使用户能够将调试器配置为在调试时使用备用源代码管理服务器。 但是，如果管理好服务器且不重命名服务器，则无需将此文件包含在客户端调试器安装中。

此文件还用于客户端的其他用途。 有关详细信息，请参阅随 SrcSrv 工具一起安装的示例Srcsrv.ini文件。

使用不同的位置或文件名

默认情况下，SrcSrv 使用名为 Srcsrv.ini 的文件作为主配置文件，该文件位于 Windows 调试工具安装目录的 srcsrv 子目录中。

可以通过将SRCSRV_INI_FILE环境变量设置为所需文件的完整路径和文件名来为配置指定其他文件。

例如，如果多个用户想要共享单个配置文件，则可以将其放置在所有系统可访问的共享上，然后设置如下所示的环境变量：

```控制台
set SRCSRV_INI_FILE=\\ourserver\ourshare\bestfile.txt
```

Ssindex.cmd 脚本

项目 • 2023/06/14

Ssindex.cmd 脚本生成签入源代码管理的文件列表以及每个文件的版本信息。 它将此信息的子集存储在生成应用程序时生成的 .pdb 文件中。 SSIndex 使用以下 Perl 模块之一来与源代码管理进行交互：

- p4.pm (Perforce)
- tfs.pm (Team Foundation Server)
- svn.pm (Subversion)

有关详细信息，请使用 **-?** 或 **-??** 运行 Ssindex.cmd (详细帮助) 选项或检查脚本。

SrcTool 实用工具

项目 • 2023/06/15

SrcTool (Srctool.exe) 实用工具列出 .pdb 文件中索引的所有文件。 对于每个文件，它列出文件的完整路径、源代码管理服务器和版本号。 可以使用此信息进行参考。

还可以使用 SrcTool 列出 .pdb 文件中的原始源文件信息。 为此，请在命令行上使用 **-s** 开关。

SrcTool 还有其他选项。 使用 **?** 开关查看它们。 最感兴趣的是，此实用工具可用于从版本控制中提取所有源文件。 这是使用 **-x** 开关完成的。

注意 提取文件时，此程序的早期版本在当前目录下创建了一个名为 src 的目录。 这种情况不会再出现。 如果要使用 src 目录，则必须自行创建它，并从该目录运行 命令。

PDBStr 工具

项目 • 2023/06/15

索引脚本使用 PDBStr (Pdbstr.exe) 工具将版本控制信息插入目标 .pdb 文件的“srcsrv”备用流中。 它还可以从 .pdb 文件读取任何流。 可以使用此信息来验证索引脚本是否正常工作。

有关详细信息，请使用 **-?** 选项运行 PDBStr。

源代码管理系统

项目 • 2023/06/15

SrcSrv 包括用于 Perforce (p4.pm) 、Team Foundation Servers (tfn.pm) 和 Subversion (svn.pm) 的提供程序模块。 还有一个并发版本系统 (CVS) 模块用于 SrcSrv，但尚未经过广泛的测试。 还可以生成自己的模块来支持其他版本的控制系统。

本节包括：

使用 CVS

使用其他源代码管理系统

使用 CVS

项目 • 2023/06/15

源服务器的 CVS 模块是使用并发版本系统 (CVS) 1.11.17 (客户端) 开发的。 尚未使用任何其他版本的 CVS 对其进行测试。 此外，模块的当前版本是 beta 版本。

CVSROOT

在生成源索引的计算机上，CVSROOT 不能包含密码和用户信息。 使用 cvs.exe 设置凭据信息。

若要为 CVS 索引准备 Srcsrv.ini 文件，必须为存储库输入一个别名，以便将其与网络中的任何其他文件唯一区分开来。 此存储库必须与环境中的 CVSROOT 值匹配。 无需在与调试器客户端一起保留的Srcsrv.ini副本中设置此值，因为别名是在源索引 .pdb 文件中定义的。

客户端计算机

在调试期间提取文件的客户端计算机不需要 CVS 沙盒或 CVSROOT 集。 它确实需要路径中的 CVS 二进制文件，如果存储库已锁定，则必须使用 Cvs.exe 设置用户名和密码。

修订标记

CVS 无法按版本号提取文件。 相反，必须使用所谓的 *标记*来完成此操作。 为基于 CVS 的系统编制索引时，必须确保所有更改都签入存储库，然后使用“cvs 标记”命令应用标记。 然后，在为文件编制索引时，请确保使用“label”命令行参数来指定要与要编制索引的生成的关联的标记。 可以通过在环境中设置CVS_LABEL来实现相同的结果。 可以从环境或命令行设置其他值。 将 **-??** 命令行选项与 SSIndex 配合使用，以检查你的选择并验证是否已正确配置所有选项：

控制台

```
ssindex.cmd -system=cvs -??
```

使用其他源代码管理系统

项目 • 2023/06/15

此包中包含的 Svn.pm。 仔细查看此文件可显示如何修改现有脚本之一以支持 Subversion 版本控制系统。 可以通过创建自己的提供程序模块来支持其他控制系统，甚至是你自己创建的控制系统。

本节包括：

创建自己的提供程序模块

创建自己的源代码管理系统

创建自己的提供程序模块

项目 • 2023/06/15

通常，若要创建自己的提供程序模块，必须实现以下接口集。

$module：：SimpleUsage ()

目的
向 STDOUT 显示简单的模块使用情况信息。

参数
无

返回值
无

$module：：VerboseUsage ()

目的
向 STDOUT 显示深入的模块使用情况信息。

参数
无

返回值
无

$objref = $module：：new (*@CommandArguments*)

目的
初始化提供程序模块的实例。

参数

@CommandArguments
ssindex.cmd 未识别为常规参数的所有 @ARGV 参数。

返回值
可在后续操作中使用的引用。

$objref->GatherFileInformation (*$SourcePath*，*$ServerHashReference*)

目的
使模块能够收集 *$SourcePath* 参数指定的目录所需的源索引信息。 模块不应假定每个对象实例只调用一次此条目，因为 SSIndex 可能会针对不同的路径多次调用它。

参数

$SourcePath
包含要编制索引的源的本地目录。

$ServerHashReference
对包含指定Srcsrv.ini文件中的所有条目的哈希的引用。

返回值
无

(*$VariableHashReference*，*$FileEntry*) =**$objref-GetFileInfo** > (*$LocalFile)*

目的
提供从源代码管理系统中提取单个特定文件所需的信息。

参数

$LocalFile
完全限定的文件名。

返回值

$VariableHashReference
解释返回的$FileEntry所需的变量 *的*哈希引用。 Ssindex.cmd 为单个调试文件使用的每个源文件缓存这些变量，以减少写入源索引流的信息量。

$FileEntry
要写入源索引流以允许 SrcSrv 从源代码管理中提取此文件的文件条目。 此行的确切格式特定于源代码管理系统。

*$TextString*= **$objref-LongName**> ()

目的
提供一个描述性字符串，用于向最终用户标识源代码管理系统。

参数
无

返回值

$TextString
源代码管理系统的描述性名称。

@StreamVariableLines=$objref-SourceStreamVariables> ()

目的

使源代码管理系统能够将特定于源代码管理的变量添加到每个调试文件的源流中。 示例模块使用此方法编写所需的EXTRACT_CMD和EXTRACT_TARGET变量。

参数

无

返回值

@StreamVariableLines
源流变量的条目列表。

创建自己的源代码管理系统

项目 • 2023/06/15

本部分介绍如何准备检测脚本，以便 SrcSrv 可以集成到生成或版本控制系统中。 实现依赖于 SrcSrv 附带的语言规范版本。

本文讨论了符号处理程序代码如何调用 Srcsrv.dll 的许多细节。 但是，此信息不是静态的，将来也不会成为静态信息。 任何人都不应尝试编写直接调用 Srcsrv.dll 的代码。 这是为 DbgHelp 或 Visual Studio 产品保留的。 想要将此类功能集成到其产品中的第三方供应商应使用 **SymGetSourceFile** 函数。 DbgHelp 文档中介绍了此函数以及 DbgHelp API 中的其他函数，该文档可在 Windows 安装目录的调试工具的 sdk/help 子目录中找到。

本节包括：

语言规范 1

语言规范 2

语言规范 3

语言规范 1

项目 • 2023/10/26

> **① 备注**
>
> 本主题介绍 SrcSrv 的内部操作。 有关源路径工作原理的一般信息，请参阅 **源路径。** 有关使用 SrcSrv 的信息，请参阅 **使用 SrcSrv。** 若要确定环境中源加载的当前操作，请启用干扰源加载，如 .srcnoisy (**干扰源加载**)

SrcSrv 的第一个版本的工作方式如下。 (此行为在将来的版本中可能会更改。)

首先，客户端使用目标路径调用 **SrcSrvInit** ，以用作所有源文件提取的基础。 此路径存储在特殊变量 TARG 中。

当 DbgHelp 加载模块的 .pdb 文件时，它会从 .pdb 文件中提取 SrcSrv 流，并通过调用 **SrcSrvLoadModule 将此数据块传递给 SrcSrv。**

然后，当 DbgHelp 需要获取源文件时，它会调用 **SrcSrvGetFile** 以从版本控制检索指定的源文件。

SrcSrv 会评审数据块中的所有源文件条目，以查找与请求的源规范完全匹配的条目。 此匹配项在 VAR1 中找到。

SrcSrv 找到条目后，会用此源文件条目的内容填充 VAR1、VAR2 等) 特殊 (变量。 然后，使用这些特殊变量解析 SRCSRVTRG 变量。

下面显示了如何使用特殊变量解析 SRCSRVTRG 变量。 我们假设源路径仍为：

控制台

```
c:\proj\src\file.cpp*TOOLS_PRJ*tools/mytool/src/file.cpp*3
```

每行显示一个又一个特殊变量的分辨率。 解析的变量为粗体。

控制台

```
SRCSRVTRG=%sdtrg%
SDTRG=%targ%\%var2%\%fnbksl%(%var3%)\%var4%\%fnfile%(%var1%)
c:\src\%var2%\%fnbksl%(%var3%)\%var4%\%fnfile%(%var1%)
c:\src\WIN_SDKTOOLS\%fnbksl%(%var3%)\%var4%\%fnfile%(%var1%)
c:\src\WIN_SDKTOOLS\%fnbksl%( sdktools/debuggers/srcsrv/shell.cpp
)\%var4%\%fnfile%(%var1%)
c:\src\WIN_SDKTOOLS\ sdktools\debuggers\srcsrv\shell.cpp\%var4%\%fnfile%
(%var1%)
```

```
c:\src\WIN_SDKTOOLS\sdktools\debuggers\srcsrv\shell.cpp\3\%fnfile%(%var1%)
c:\src\WIN_SDKTOOLS\sdktools\debuggers\srcsrv\shell.cpp\3\%fnfile%(
c:\db\srcsrv\shell.cpp)
c:\src\WIN_SDKTOOLS\sdktools\debuggers\srcsrv\shell.cpp\3\shell.cpp
```

请注意，此生成的目标路径是唯一的，并且不允许将同一文件的两个版本提取到同一位置。

SrcSrv 现在查看文件是否已存在。 如果是，SrcSrv 会将位置返回给调用方。 否则，SrcSrv 会生成执行命令，通过解析 SRCSRVCMD 提取文件。

在以下示例中，每行显示一个又一个特殊变量的分辨率。 解析的变量为粗体。

控制台

```
DEPOT=//depot
WIN_SDKTOOLS= sserver.microsoft.com:4444
SRCSRVCMD=%sdcmd%
SDCMD=sd.exe -p %fnvar%(%var2%) print -o %srcsrvtrg% -q
%depot%/%var3%#%var4%
sd.exe -p %fnvar%(WIN_SDKTOOLS) print -o %srcsrvtrg% -q
%depot%/%var3%#%var4%
sd.exe -p sserver.microsoft.com:4444  print -o %srcsrvtrg% -q
%depot%/%var3%#%var4%
sd.exe -p sserver.microsoft.com:4444  print -o
c:\src\WIN_SDKTOOLS\sdktools\debuggers\srcsrv\shell.cpp\3\shell.cpp -q
%depot%/%var3%#%var4%
sd.exe -p sserver.microsoft.com:4444  print -o
c:\src\WIN_SDKTOOLS\sdktools\debuggers\srcsrv\shell.cpp\3\shell.cpp -q
//depot/%var3%#%var4%
sd.exe -p sserver.microsoft.com:4444  print -o
c:\src\WIN_SDKTOOLS\sdktools\debuggers\srcsrv\shell.cpp\3\shell.cpp -q
//depot/ sdktools/debuggers/srcsrv/shell.cpp#%var4%
sd.exe -p sserver.microsoft.com:4444  print -o
c:\src\WIN_SDKTOOLS\sdktools\debuggers\srcsrv\shell.cpp\3\shell.cpp -q
//depot/ sdktools/debuggers/srcsrv/shell.cpp#3
```

现在，SrcSrv 将执行此命令。 如果此命令的结果是预期位置中的文件，则此路径将返回到调用方。

请注意，如果变量无法解析，则会尝试将其查找为 OS 环境变量。 如果失败，将从正在处理的文本中删除变量名称。

连续两个百分号字符被解释为单个百分号。

源服务器数据块

SrcSrv 依赖于 .pdb 文件中的两个数据块：源文件列表和数据块。

生成模块时会自动创建源文件列表。 此列表包含用于生成模块的源文件的完全限定路径。

数据块是在源索引期间创建的。 此时，将名为“srcsrv”的备用流添加到 .pdb 文件。 插入此数据的脚本依赖于使用的特定生成进程和源代码管理系统。

数据块分为三个部分：ini、变量和源文件。 数据块具有以下语法。

控制台

```
SRCSRV: ini ------------------------------------------------
VERSION=1
VERCTRL=<source_control_str>
DATETIME=<date_time_str>
SRCSRV: variables ------------------------------------------
SRCSRVTRG=%sdtrg%
SRCSRVCMD=%sdcmd%
SRCSRVENV=var1=string1\bvar2=string2
DEPOT=//depot
SDCMD=sd.exe -p %fnvar%(%var2%) print -o %srcsrvtrg% -q
%depot%/%var3%#%var4%
SDTRG=%targ%\%var2%\%fnbksl%(%var3%)\%var4%\%fnfile%(%var1%)
WIN_SDKTOOLS= sserver.microsoft.com:4444
SRCSRV: source files ---------------------------------------
<path1>*<var2>*<var3>*<var4>
<path2>*<var2>*<var3>*<var4>
<path3>*<var2>*<var3>*<var4>
<path4>*<var2>*<var3>*<var4>
SRCSRV: end ------------------------------------------------
```

所有文本均按字面解释，括在百分比符号 (%) 中的文本除外。 括在百分比符号中的文本被视为要递归解析的变量名称，除非它是以下函数之一：

%fnvar% ()
参数文本应括在百分比符号中，并被视为要解析的变量。

%fnbksl% ()
参数文本中 (/) 的所有正斜杠都应替换为反斜杠 () 。

%fnfile% ()
应去除参数文本中的所有路径信息，只保留文件名。

数据块的 [ini] 部分包含描述要求的变量。 索引脚本可以向此部分添加任意数量的变量。 下面是一些示例：

版本
语言规范版本。 此变量是必需的。 如果根据当前语言规范开发脚本，请将此值设置为

1。 SrcSrv 客户端代码不会尝试执行任何值大于其本身的值的脚本。 SrcSrv 的当前版本使用值 2。

VERCTRL
描述源版本控制系统的字符串。 此变量是可选的。

Datetime
一个字符串，指示处理 .pdb 文件的日期和时间。 此变量是可选的。

数据块的 [variables] 部分包含描述如何从源代码管理中提取文件的变量。 它还可用于将常用文本定义为变量，以减小数据块的大小。

SRCSRVTRG
描述如何为提取的文件生成目标路径。 这是一个必需的变量。

SRCSRVCMD
描述如何生成命令以从源代码管理提取文件。 这包括可执行文件的名称及其命令行参数。 如果必须执行任何提取命令，则这是必需的。

SRCSRVENV
列出在文件提取过程中要创建的环境变量。 这是一个字符串。 使用退格字符 (\b) 分隔多个条目。 这是一个可选的变量。

SRCSRVVERCTRL
指定正在使用的版本控制系统。 对于 Perforce，这是 perforce。 对于 Team Foundation Server，这是 tfs。 此变量用于保留服务器错误。 这是一个可选的变量。

SRCSRVVERRDESC
指定在版本控制客户端无法联系包含要提取的源文件的服务器时要显示的文本。 SrcSrv 使用此值来检查连接问题。 这是一个可选的变量。

SRCSRVERRVAR
指示文件条目中的哪个变量对应于版本控制服务器。 SrcSrv 根据以前的故障使用它来识别不起作用的命令。 文本的格式为 **var***X* ，其中 *X* 是所指示变量的编号。 这是一个可选的变量。

数据块的 [源文件] 部分包含已编制索引的每个源文件的条目。 每一行的内容将解释为名为 VAR1、VAR2、VAR3 ... VAR10 的变量。 变量由星号分隔。 VAR1 必须指定源文件的完全限定路径，如 .pdb 文件中的其他位置所示。 例如：

控制台

```
c:\proj\src\file.cpp*TOOLS_PRJ*tools/mytool/src/file.cpp*3
```

解释如下：

控制台

```
VAR1=c:\proj\src\file.cpp
VAR2=TOOLS_PRJ
VAR3=tools/mytool/src/file.cpp
VAR4=3
```

在此示例中，VAR4 是修订号。 但是，大多数源代码管理系统都支持标签文件，以便可以还原给定生成的源状态。 因此，可以改用标签进行生成。 可以修改示例数据块以包含变量，如下所示：

控制台

```
LABEL=BUILD47
```

假设源代码管理系统使用 @ 符号来指示标签，则你可以如下所示修改 SRCSRVCMD 变量：

控制台

```
sd.exe -p %fnvar%(%var2%) print -o %srcsrvtrg% -q %depot%/%var3%@%label%
```

处理服务器错误

有时，客户端根本无法从单个版本控制服务器中提取任何文件。 原因可能是服务器关闭和离开网络，或者用户没有访问源的适当权限。 但是，尝试获取此源所花费的时间可能会显著减慢速度。 在这种情况下，最好禁用从已证明不可用的源中提取的任何尝试。

每当 SrcSrv 无法提取文件时，它都检查命令生成的输出文本。 如果此命令的任何部分包含 SRCSRVERRDESC 内容的完全匹配项，则会跳过对同一版本控制服务器的所有未来命令。 请注意，可以通过在 SRCSRVERRDESC 变量名称末尾添加数字或任意文本来定义多个错误字符串。 以下是示例：

控制台

```
SRCSRVERRDESC=lime: server not found
SRCSRVERRDESC_2=pineapple: network error
```

服务器的标识是从 SRCSRVERRVAR 获取的。 因此，如果 SRCSRVERRVAR 包含“var2”，并且 .pdb 文件中的文件条目如下所示：

控制台

```
c:\proj\src\file.cpp*TOOLS_PRJ*tools/mytool/src/file.cpp*3
```

将来使用变量 2 中包含“TOOLS_PRJ”的文件条目获取源的所有尝试都会被绕过。

还可以通过编辑 Srcsrv.ini在调试器客户端上添加错误指示器。 有关详细信息，请参阅包含的示例版本的 srcsrv.ini。

语言规范 2

项目 • 2023/06/15

此包附带srcsrv.dll，无需在 .pdb 文件的 srcsrv 数据流中定义 SRCSRVCMD 即可运行。 虽然此功能对从源代码管理系统中提取普通文件没有用处，但它对于从 UNC 共享或 HTTP 站点直接访问文件很有用。 有关更多详细信息，请参阅 HTTP 站点和 UNC 共享 。

语言规范 3

项目 • 2024/03/20

与 SrcSrv 语言规范 2 不一样。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

HTTP 站点和 UNC 共享

项目 • 2024/06/28

可以使用 SrcSrv 建立一个网站，为 WinDbg 提供特定版本的源代码。 这种机制并不能从版本控制中动态提取源文件，但它是一个很有价值的功能，因为它可以将 WinDbg 的源文件路径设置为一个统一的路径，提供许多模块的许多版本的源文件，而不必为每个调试场景设置单独的路径。 对于可以直接访问实际版本控制系统的调试客户端来说，这一点并不重要，但对于那些希望提供基于 HTTP 的远程源代码安全访问的客户端来说，这却很有帮助。 如果需要，可以通过 HTTPS 和智能卡来确保有关网站的安全。 同样的技术也可用于通过简单的 UNC 共享提供源文件。

本部分包括：

设置网站

提取源文件

修改 .pdb 文件中的源索引流

使用 UNC 共享

将 HTTP 站点和 UNC 共享与常规版本控制结合使用

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

设置网站

项目 • 2023/06/15

设置一个网站，从中共享源文件，而不是网站的根目录。 然后，可从站点获取源，例如：

```text
https://SrcMachineName/source
```

若要使源文件可通过 Internet 访问，必须配置包含源文件的目录。

首先选择索引源所在的目录。 在我们的示例中，我们将此目录称为 c：\source，并将服务器的名称称为网络 \SrcMachineName。 必须设置权限，以便用户可以访问站点，并且必须添加必须通过网络访问源内容的安全组。 要启用的安全性量因环境而异。 对于某些安装，此组为 “**所有人**”。

设置目录的权限：

1. 打开 **Windows 资源管理器。**
2. 展开 “**我的电脑**”。
3. 展开 C： 驱动器。
4. 右键单击 “**c：\source** ”，然后选择“ **共享和安全性**”。
5. 选中 “**共享此文件夹**” 按钮。
6. 单击 “**权限**” 按钮。
7. 通过在“ **组或用户名**” 下添加所需安全组并选中相应的框，验证它们是否具有读取访问权限。
8. 单击“ **确定**” 退出“权限”。
9. 单击“ **确定**” 退出“源属性”。

源目录现在可供源路径为 srv*\\SrcMachineName\source 的另一台计算机进行调试。

提取源文件

项目 • 2023/06/15

若要从要提供源的所有模块中提取所有源文件，请使用 命令：

控制台

```
srctool.exe -x
```

必须在已为具有版本控制访问权限的计算机上的版本控制系统编制源索引的 .pdb 文件上执行此工具。 这会将所有源文件放入通用目录树中。 将此树的内容复制到网站的根目录下。 可以在要添加的任何新产品或模块上任意频繁地执行此操作。 无需担心文件会相互覆盖，因为目录树结构使不同的文件保持隔离且唯一可访问。

走

Walk (Walk.cmd) 脚本包含在 Windows 调试工具中。 此脚本以递归方式搜索目录树，并在与指定文件掩码匹配的任何文件上执行任何指定的命令。 语法为：

控制台

```
walk.cmd FileMask Command
```

其中， *FileMask* 指定文件掩码（带或不带随附的起始目录）， *而 Command* 指定要执行的命令。

以下示例对 c：\symbols 及其子目录中的所有 .pdb 文件运行 srctool.exe 文件提取命令：

控制台

```
walk.cmd c:\symbols\*.pdb srctool.exe -x
```

修改 .pdb 文件中的源索引流

项目 • 2023/06/15

若要使调试器客户端使用 SrcSrv 网站，必须将 .pdb 文件修改为指向它。 若要手动执行此操作，请创建所有 .pdb 文件的副本，对其进行更改，并使它们从单独的位置（通常是网站本身）提供。

Windows 调试工具提供三个文件来帮助重新配置 .pdb 文件。 Cv2http.cmd 和 Cv2http.pl 文件提取 SrcSrv 流，使用 Perl 脚本对其进行修改，并将更改的流放回 .pdb 文件中。 语法如下所示：

控制台

```
cv2http.cmd PDB Alias URL
```

其中 *PDB* 指定要修改的 .pdbfile 的名称， *Alias* 指定要应用于网站的逻辑名称， *URL* 指定网站的完整 URL。 请注意， *Alias* 参数作为变量名称存储在 PDB 中，如果曾经移动网站的位置，可以在 Scrsrv.ini 调试器客户端上重写该名称。

此脚本要求所有标准 SrcSrv 工具在路径中可用，因为它同时调用 SrcTool 和 PDBStr。 请记住，Cv2http.pl 是 Perl 脚本，可以根据需要进行修改。

第三个文件 Walk (walk.cmd) 脚本修改整个 .pdb 文件集。 例如：

控制台

```
walk.cmd *.pdb cv2http.cmd HttpAlias https:///source
```

上述命令调用树中每个 .pdb 文件的 Cv2http.cmd，使用 HttpAlias 作为别名和 https://server/source URL。 有关 Walk 的更多详细信息，请参阅 提取源文件。

在 .pdb 文件的树上执行此命令后，它们已准备好安装到网站或要放置它们的任何位置。 请记住，可以使用 SrcTool 和 PDBStr 检查对 .pdb 文件的更改。

使用 UNC 共享

项目 • 2024/06/11

Cv2http.cmd、Cv2http.pl 和 Walk (Walk.cmd) 脚本用于从简单的 UNC 共享中提供源文件。 文件 Cv2http.cmd 和 Cv2http.pl 会提取 SrcSrv 数据流，使用 Perl 脚本对其进行修改，然后将修改后的数据流放回 .pdb 文件中。 语法如下所示：

```
cv2http.cmd PDB Alias SourceRoot
```

其中，*PDB* 指定要修改的 .pdb 文件的名称，*Alias* 指定应用于网站的逻辑名称，*SourceRoot* 指定提取源文件的 UNC 共享的根目录。 请注意，*Alias* 参数作为变量名存储在 PDB 中，如果移动了网站的位置，则可以在调试器客户端的 Scrsrv.ini 中覆盖该变量名。

该脚本要求路径中包含所有标准 SrcSrv 工具，因为它会同时调用 SrcTool 和 PDBStr。 请记住，Cv2http.pl 是一个 Perl 脚本，可以根据需要进行修改。

第三个文件是 Walk (walk.cmd) 脚本，用于修改一整套 .pdb 文件。 例如：

控制台

```
walk.cmd *.pdb cv2http.cmd SourceRoot \\server\share
```

上述命令在树中的每个 .pdb 文件上调用 Cv2http.cmd，使用 SourceRoot 作为别名，并使用 \server\share 作为 UNC 共享。 有关 Walk 的详细信息，请参阅提取源文件。

在 .pdb 文件树上执行该命令后，这些文件就可以安装到网站或想放置它们的任何位置。请记住，可以使用 SrcTool 和 PDBStr 来检查 .pdb 文件的更改。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

将 HTTP 站点和 UNC 共享与常规版本控制结合使用

项目 • 2024/05/27

你可能会发现，必须使用从版本控制中提取文件的标准 SrcSrv 功能为开发人员提供支持，但还必须通过网站或 UNC 共享提供源文件。 如果建立的测试实验室无法访问版本控制，就会出现这种情况。 使用同一组 .pdb 文件可以同时支持两个用户。

首先，使用 SrcTool 提取源文件；有关详细信息，请参阅提取源文件。 将共享作为网站或 UNC 共享。 目前，不应使用 Cv2http.cmd 脚本来转换 .pdb 文件。

现在，在将使用 HTTP/UNC 共享的计算机上，编辑调试器目录中的 Srcsrv.ini 文件。 在文件的变量部分，添加以下三个语句：

ini

```
MY_SOURCE_ROOT=\\server\share
 SRCSRVCMD=
 SRCSRVTRG=%MY_SOURCE_ROOT%\%var2%\%var3%\%var4%\%fnfile%(%var1%)
```

应将 \\server\share 替换为所提供 UNC 共享的根目录或包含源文件的网站 URL。 也可以将 MY_SOURCE_ROOT 更改为要用来描述该位置的任何别名。 除上述例外情况外，其他内容均应完全按照描述输入。

以这种方式设置的所有调试器都会忽略标准版本控制提取指令，并从指定的位置访问源文件。 同时，Srcsrv.ini 中不包含这些项的所有调试器都使用正常的版本控制机制来提取源文件。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

安全注意事项

项目 • 2024/03/20

本节包括：

安全漏洞

安全模式

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

调试特权

项目 • 2023/06/15

调试权限允许某人调试他们本来无权访问的进程。 例如，在其令牌上启用了调试权限的用户身份运行的进程可以调试作为本地系统运行的服务。

调试权限是一种安全策略设置，允许用户将调试器附加到进程或内核。 管理员可以修改用户组的安全策略，以包括或删除此功能。 调试自己的应用程序的开发人员不需要此用户权限。 正在调试系统组件或正在调试远程组件的开发人员将需要此用户权限。 此用户特权提供对敏感和关键操作系统组件的完全访问。 默认情况下，为具有管理员权限的用户启用此属性。 具有管理员权限的用户可以为其他用户组启用此属性。

修改进程的调试权限

下面的代码示例演示如何在进程中启用调试权限。 这使你能够调试你无权访问的其他进程。

C++

```
//
//  SetPrivilege enables/disables process token privilege.
//
BOOL SetPrivilege(HANDLE hToken, LPCTSTR lpszPrivilege, BOOL
bEnablePrivilege)
{
    LUID luid;
    BOOL bRet=FALSE;

    if (LookupPrivilegeValue(NULL, lpszPrivilege, &luid))
    {
        TOKEN_PRIVILEGES tp;

        tp.PrivilegeCount=1;
        tp.Privileges[0].Luid=luid;
        tp.Privileges[0].Attributes=(bEnablePrivilege) ?
SE_PRIVILEGE_ENABLED: 0;
        //
        //  Enable the privilege or disable all privileges.
        //
        if (AdjustTokenPrivileges(hToken, FALSE, &tp, NULL,
(PTOKEN_PRIVILEGES)NULL, (PDWORD)NULL))
        {
            //
            //  Check to see if you have proper access.
            //  You may get "ERROR_NOT_ALL_ASSIGNED".
            //
            bRet=(GetLastError() == ERROR_SUCCESS);
```

```
        }
    }
    return bRet;
}
```

以下示例演示如何使用此函数：

C++

```
HANDLE hProcess=GetCurrentProcess();
HANDLE hToken;

if (OpenProcessToken(hProcess, TOKEN_ADJUST_PRIVILEGES, &hToken))
{
    SetPrivilege(hToken, SE_DEBUG_NAME, TRUE);
    CloseHandle(hToken);
}
```

安全漏洞

项目 • 2024/11/25

本节包括：

内核模式调试期间的安全性

用户模式调试期间的安全性

事后调试期间的安全性

远程调试期间的安全性

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

内核模式调试期间的安全性

项目 • 2023/06/14

内核模式调试期间的安全性绝不是保护 *目标* 计算机。 目标完全容易受到调试器攻击 -- 这就是调试的本质。

如果在启动期间启用了调试连接，则在下次启动之前，它仍会通过调试端口受到攻击。

但是，你应关注 *主* 计算机上的安全性。 在理想情况下，调试器在主计算机上作为应用程序运行，但不与此计算机上的其他应用程序交互。 出现安全问题的可能方式有三种：

- 如果使用损坏或破坏性的扩展 DLL，它们可能会导致调试器执行意外操作，从而可能影响主计算机。
- 损坏或破坏性的符号文件也可能导致调试器执行意外操作，从而可能影响主计算机。
- 如果运行的是远程调试会话，则意外的客户端可能会尝试链接到服务器。 或者，你期望的客户端可能会尝试执行你未预料到的操作。

如果要阻止远程用户在主计算机上执行操作，请使用 安全模式。

有关如何防范意外远程连接的建议，请参阅 远程调试期间的安全性。

如果不执行远程调试，仍应注意错误的符号文件和扩展 DLL。 不要加载不信任的符号或扩展！

本地内核调试

只有具有调试权限的用户才能启动本地内核调试会话。 如果你是具有多个用户帐户的计算机的管理员，则应注意，任何拥有这些权限的用户都可以启动本地内核调试会话，从而有效地让他们控制计算机上的所有进程，从而也允许他们访问所有外围设备。

用户模式调试期间的安全性

项目 • 2023/06/15

用户模式调试器处于活动状态时，它可以有效地控制计算机上的任何进程。

在用户模式调试过程中，有三种可能出现安全问题的可能方式：

- 如果使用损坏或破坏性的扩展 DLL，它们可能会导致调试器执行意外操作，从而可能影响目标以外的应用程序。
- 损坏或破坏性的符号文件也可能导致调试器执行意外操作，从而影响目标以外的应用程序。
- 如果运行的是远程调试会话，则意外的客户端可能会尝试链接到服务器。 或者，你期望的客户端可能会尝试执行你未预料到的操作。

有关如何防范意外远程连接的建议，请参阅 远程调试期间的安全性。 远程客户端加入用户模式调试会话后，无法限制其对计算机上的进程的访问。

如果不执行远程调试，仍应注意错误的符号文件和扩展 DLL。 不要加载不信任的符号或扩展！

事后调试期间的安全性

项目 • 2024/11/25

只有管理员才能启用 事后调试。

但是，为整个系统（而不仅仅是一个用户）启用事后调试。 因此，在启用它后，任何应用程序崩溃都将激活已选择的调试器-即使当前用户没有管理权限。

此外，事后调试器会继承与崩溃的应用程序相同的特权。 因此，如果 WINDOWS 服务（如 CSRSS 和 LSASS）崩溃，调试器将具有非常高的级别权限。

选择启用事后调试时，应考虑到这一点。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

远程调试期间的安全性

项目 • 2023/10/26

可通过两种方式在远程调试期间提高安全性：限制可以连接到会话的人员，以及限制进行连接的人员的权限。

控制对调试会话的访问

如果要 通过调试器或使用 进程服务器 或 KD 连接服务器执行远程调试，则本地网络上的任何计算机都可以尝试附加到调试会话。

如果使用 TCP、1394、COM 或命名管道协议，则可以要求调试客户端提供密码。 但是，此密码在传输过程中未加密，因此这些协议不安全。

如果希望调试服务器安全，则必须使用安全套接字层 (SSL) 或安全管道 (SPIPE) 协议。

如果 通过 remote.exe执行远程调试 ，则可以使用 **/u** 参数来禁止未经授权的用户进行连接。

限制客户端的权限

如果要设置内核模式调试会话，则可以使用 安全模式限制调试器干扰主机的能力。

在用户模式下，安全模式不可用。 可以通过发出 **.noshell (禁止 shell 命令)** 命令来阻止侵入性客户端发出 Microsoft MS-DOS 命令和运行外部程序。 但是，客户端可通过许多其他方式干扰你的计算机。

请注意，安全模式和 **.noshell** 都会阻止调试客户端和调试服务器执行某些操作。 无法对客户端施加限制，但不能对服务器施加限制。

忘记的进程服务器

在远程计算机上启动进程服务器时，进程服务器以无提示方式运行。

如果通过此进程服务器执行远程调试，然后结束会话，则进程服务器将继续运行。

被遗忘的进程服务器是攻击的潜在目标。 应始终关闭不需要的进程服务器。 使用 Kill.exe 实用工具或任务管理器终止进程服务器。

安全模式

项目 • 2023/06/15

执行内核模式调试时，可以在 *安全模式*下运行调试器。 这可以防止调试器影响主计算机，但不会显著减少调试目标计算机的自由度。

如果要允许远程客户端加入调试会话，建议使用安全模式。

本节包括：

安全模式的功能

激活安全模式

安全模式的功能

项目 • 2023/10/26

当安全模式处于活动状态时，所有可用于影响 *主机* 的命令都会停用，并且符号服务器和调试器扩展存在一些限制。

安全模式的具体效果如下：

- .attach (**附加到进程**) 、.create (Create Process) 、.detach (Detach from Process) 、.abandon (Abandon Process) 、.kill (Kill Process) 、.tlist (**列出进程 ID**) 、.dump (**创建转储文件**) 、.opendump (Open Dump File) 、.writemem (**将内存写入文件**) 、.netuse (**控制网络连接**) ， 和 .quit_lock (**防止意外退出**) 命令不可用。
- **文件 |附加到进程**、**文件 |打开可执行文件**，**调试 |分离调试对象**，**调试 |停止调试**，**文件 |打开故障转储** WinDbg 菜单命令不可用。
- .shell (Command Shell) 命令不可用。
- 扩展 DLL 必须从本地磁盘加载;无法从 UNC 路径加载它们。
- 仅允许使用两种标准类型的扩展 DLL (wdbgexts.h 和 dbgeng.h) 。 其他类型的 DLL 不能作为扩展加载。
- 如果使用符号服务器，则有几个限制。 仅允许使用 SymSrv (symsrv.dll) ;将不接受其他符号服务器 DLL。 不得将下游存储用于符号，任何现有的下游存储将被忽略。 不允许使用 HTTP 和 HTTPS 连接。

激活后，无法关闭安全模式。 有关详细信息，请参阅 激活安全模式。

激活安全模式

项目 • 2024/07/08

安全模式仅适用于内核模式调试。 它必须在调试会话开始之前激活。无论是在调试器的命令行上，还是在调试器完全处于休眠状态且尚未用作服务器时。

若要激活安全模式，请使用以下方法之一：

- -secure 命令行选项
- .secure 1 命令
- .symopt+0x40000 命令

如果您已有内核调试会话，并且需要确定是否已处于安全模式，请使用不带参数的 .secure 命令。 这将告知安全模式的当前状态。

激活安全模式后，无法将其关闭。 即使结束调试会话也不会将其关闭。 只要调试器本身在运行，安全模式就会一直存在。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

处理器体系结构

项目 • 2024/03/20

本节包括：

x86 处理器

x64 处理器

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

x86 处理器

项目 • 2024/03/20

本节包括：

x86 体系结构

x86 指令

带批注的 x86 反汇编

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

x86 体系结构

项目 • 2024/07/08

Intel x86 处理器使用复杂的指令集计算机（CISC）体系结构，这意味着有少量的特殊用途寄存器，而不是大量的通用寄存器。 这也意味着复杂的特殊用途指令将占主导地位。

x86 处理器至少追溯到 8 位 Intel 8080 处理器的遗产。 x86 指令集中的许多特点是由于与该处理器的向后兼容性（及其 Zilog Z-80 变体）。

Microsoft Win32 在 *32 位平面模式*下使用 x86 处理器。 本文档将仅重点介绍平面模式。

寄存 器

x86 体系结构由以下无特权整数寄存器组成。

展开表

eax	累加器
ebx	基本寄存器
ecx	计数器寄存器
edx	数据寄存器 - 可用于 I/O 端口访问和算术函数
esi	源索引寄存器
edi	目标索引寄存器
ebp	基本指针寄存器
esp	堆栈指针

所有整数寄存器均为 32 位。 但是，其中许多子注册有 16 位或 8 位子注册。

展开表

斧头	低 16 位 **eax**
bx	低 16 位 **ebx**
cx	低 16 位 **ecx**
dx	低 16 位 **edx**
si	低 16 位 **esi**

di	低 16 位 **edi**
bp	低 16 位 **ebp**
sp	低 16 位 **esp**
al	低 8 位 **eax**
ah	高 8 位斧头
bl	低 8 位 **ebx**
bh	高 8 位 **bx**
cl	低 8 位 **ecx**
ch	高 8 位 **cx**
dl	低 8 位 **edx**
dh	高 8 位 **dx**

在子注册上操作仅影响子注册，并且子注册之外没有部分。 例如，存储到 **ax** 寄存器会使 eax **寄存器的高** 16 位保持不变。

使用**时？ （计算表达式）**命令，寄存器应以“at”符号（） 作为前缀。 @ 例如，应使用 **？ @ax** 而不是 **？ ax**。 这可确保调试器将 ax **识别**为寄存器而不是符号。

但是，r （Registers） 命令中 **不需要 （@）** 。 例如， **始终正确解释 r ax=5** 。

另外两个寄存器对于处理器的当前状态非常重要。

展开表

eip	指令指针
flags	flags

指令指针是所执行指令的地址。

标志寄存器是单位标志的集合。 许多指令更改标志以描述指令的结果。 然后，可以通过条件跳转指令测试这些标志。 有关详细信息，请参阅 x86 标志。

调用约定

x86 体系结构具有多个不同的调用约定。 幸运的是，它们都遵循相同的寄存器保留和函数返回规则：

- 函数必须保留除 eax、**ecx 和 edx 之外**的所有寄存器，这些寄存器可以跨函数调用进行更改，并且 **esp** 必须根据调用约定进行更新。
- **如果结果为 32 位或更小，eax** 寄存器将接收函数返回值。 如果结果为 64 位，则结果存储在 edx：eax **对中。**

下面是在 x86 体系结构上使用的调用约定的列表：

- Win32 （**__stdcall**）

  函数参数在堆栈上传递，向右推送到左侧，被调用方清理堆栈。

- 本机C++方法调用（也称为 thiscall）

  函数参数在堆栈上传递，向右向左推送，“this”指针在 ecx **寄存器中**传递，被调用方清理堆栈。

- COM （**C++方法调用的__stdcall** ）

  函数参数在堆栈上传递，向右推送到左侧，然后在堆栈上推送“this”指针，然后调用该函数。 被调用方清理堆栈。

- **__fastcall**

  前两个 DWORD 或较小的参数在 ecx 和 edx **寄存器中传递。** 其余参数在堆栈上传递，从右向左推送。 被调用方清理堆栈。

- **__cdecl**

  函数参数在堆栈上传递，向右推送到左侧，调用方清理堆栈。 __cdecl调用约定用于具有可变长度参数的所有函数。

寄存器和标志的调试器显示

下面是一个示例调试器寄存器显示：

dbgcmd

```
eax=00000000 ebx=008b6f00 ecx=01010101 edx=ffffffff esi=00000000
edi=00465000
eip=77f9d022 esp=05cffc48 ebp=05cffc54 iopl=0         nv up ei ng nz na po
nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000
efl=00000286
```

在用户模式调试中，可以忽略 **iopl** 和调试器显示的完整最后一行。

x86 标志

在前面的示例中，第二行末尾的双字母代码是 *标志*。 这些是单位寄存器，具有多种用途。

下表列出了 x86 标志：

展开表

标记代码	标志名称	值	标志状态	说明
之	溢出标志	0 1	**nvov**	无溢出 - 溢出
df	方向标志	0 1	**updn**	向上方向 - 方向向下
if	中断标志	0 1	**diei**	已禁用中断 - 已启用中断
sf	标记标志	0 1	**plng**	正 （或零） - 负数
zf	零标志	0 1	**nzzr**	非零 - 零
af	辅助携带标志	0 1	**naac**	无辅助携带 - 辅助携带
pf	奇偶校验标志	0 1	**pepo**	奇偶校验奇数 - 奇偶校验甚至
cf	携带标志	0 1	**nccy**	无携带 - 携带
tf	陷阱标志			如果 **tf** 等于 1，则处理器在执行一个指令后将引发 STATUS_SINGLE_STEP异常。 调试器使用此标志来实现单步跟踪。 它不应由其他应用程序使用。
iopl	I/O 特权级别			I/O 特权级别这是一个两位整数，其值介于零和 3 之间。 操作系统使用它来控制对硬件的访问。 应用程序不应使用它。

当寄存器显示为调试器命令窗口中某些命令的结果时，它是 *显示的标志状态* 。 但是，如果要使用 **r （Registers）** 命令更改标志，则应通过*标志代码*引用它。

在 WinDbg 的“寄存器”窗口中，标志代码用于查看或更改标志。 不支持标志状态。

以下是一个示例。 在前面的寄存器显示中，将显示标志状态 **ng** 。 这意味着签名标志当前设置为 1。 若要更改此项，请使用以下命令：

```
dbgcmd
```

```
r sf=0
```

这会将符号标志设置为零。 如果执行另一个寄存器显示， **则不会显示 ng** 状态代码。 相反， **将显示 pl** 状态代码。

标志、零标志和携带标志是最常用的标志。

条件

条件描述一个或多个标志的状态。 x86 上的所有条件操作都以条件表示。

汇编程序使用一两个字母缩写来表示条件。 条件可以由多个缩写表示。 例如，AE（“高于或等于”）与 NB 相同（“不低于”）。 下表列出了一些常见条件及其含义。

展开表

条件名称	标记	含义
Z	ZF=1	上次操作的结果为零。
NZ	ZF=0	上次操作的结果不是零。
C	CF=1	上次操作需要携带或借款。（对于无符号整数，这表示溢出。
NC	CF=0	上次操作不需要携带或借款。（对于无符号整数，这表示溢出。
S	SF=1	上次操作的结果具有其高位集。
NS	SF=0	最后一个操作的结果具有较高的位清除率。
O	OF=1	当被视为有符号整数操作时，最后一个操作会导致溢出或下溢。

条件名称	标记	含义
NO	OF=0	当被视为有符号整数运算时，最后一个操作不会导致溢出或下溢。

条件还可用于比较两个值。 **cmp** 指令比较其两个操作数，然后设置标志，就像从另一个操作数中减去一个操作数一样。 以下条件可用于检查 cmp ***value1*、*value2* 的结果。**

展开表

条件名称	标记	CMP 操作后的含义。
E	ZF=1	*value1* == *value2*。
NE	ZF=0	*value1* ！ = *value2*。
GE NL	SF=OF	*value1*>= *value2*。 值被视为有符号整数。
LE NG	ZF=1 或 SF！=OF	*value1*<= *value2*。 值被视为有符号整数。
G NLE	ZF=0 和 SF=OF	*value1*>*value2*。 值被视为有符号整数。
L NGE	SF！=OF	*value1*<*value2*。 值被视为有符号整数。
AE NB	CF=0	*value1*>= *value2*。 值被视为无符号整数。
BE NA	CF=1 或 ZF=1	*value1*<= *value2*。 值被视为无符号整数。
A NBE	CF=0 和 ZF=0	*value1*>*value2*。 值被视为无符号整数。
B NAE	CF=1	*value1*<*value2*。 值被视为无符号整数。

条件通常用于处理 cmp **或测试指令的结果。** 例如，

asm

```
cmp eax, 5
jz equal
```

通过计算表达式（eax - 5）并根据结果设置标志，将 eax 寄存器与数字 5 进行比较。 如果减法的结果为零，**则将设置 zr** 标志，jz 条件为 true，因此将执行跳转。

数据类型

- 字节：8 位
- word：16 位
- dword：32 位
- qword：64 位（包括浮点双精度）
- tword： 80 位 （包括浮点扩展双精度）
- oword：128 位

表示法

下表指示用于描述程序集语言指令的表示法。

展开表

表示法	含义
r、 **r1**、 **r2**...	寄存器
m	内存地址（有关详细信息，请参阅成功的寻址模式部分。
#n	即时常量
r/m	注册或内存
r/#n	注册或即时常量
r/m/#n	注册、内存或即时常量
cc	前面“条件”部分中列出的条件代码。
T	“B”、“W”或“D”（字节、单词或 dword）
acc*T*	大小 *T* 累加器： **al** 如果 *T* = “B”， **ax** if *T* = “W”， or **eax** if *T* = “D”

寻址模式

有多种不同的寻址模式，但它们都采用 T ptr [expr] 形式，**其中 T 是一些数据类型（请参阅前面的数据类型部分），expr 是涉及常量和寄存器的一些**表达式。

大多数模式的表示法可以推断出来，而不会造成太大的困难。 例如，**BYTE PTR [esi+edx*8+3]** 表示"获取 esi **寄存器的值，将其添加到 edx 寄存器的值**的八倍，再添加三个，然后在生成的地址访问字节。

流水线

奔驰是双重问题，这意味着它可以在一个时钟滴答声中执行多达两个操作。 但是，有关何时能够同时执行两个操作（称为 *配对*）的规则非常复杂。

由于 x86 是 CISC 处理器，因此无需担心跳转延迟槽。

同步内存访问

加载、修改和存储说明可以接收 **锁** 前缀，该前缀将修改指令，如下所示：

1. 发出指令之前，CPU 将刷新所有挂起的内存操作，以确保一致性。 所有数据预提取将被放弃。
2. 发出指令时，CPU 将具有对总线的独占访问权限。 这可确保加载/修改/存储操作的原子性。

每当 xchg 指令与内存交换值时，它都会自动遵守以前的规则。

所有其他指令默认为非锁定。

跳转预测

预计将采取无条件跳跃。

根据条件跳跃的上次执行时间，预计要采取还是不采取条件跳转。 用于录制跳转历史记录的缓存的大小有限。

如果 CPU 没有记录条件跳转是在上次执行条件跳转时是否拍摄的，则它会预测后向条件跳跃，而前向条件跳跃则不采用。

对准

x86 处理器会在性能损失时自动更正未对齐的内存访问。 不会引发异常。

如果地址是对象大小的整数倍数，则内存访问被视为对齐。 例如，所有 BYTE 访问都对齐（一切都是 1 的整数倍数），WORD 对偶数地址的访问是对齐的，DWORD 地址必须是

4 的倍数才能对齐。

锁前缀不应用于未对齐的内存访问。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

x86 指令

项目 • 2023/06/15

在本部分的列表中，标有星号 (*) 的说明尤其重要。 未如此标记的说明并不重要。

在 x86 处理器上，指令大小可变，因此向后反汇编是模式匹配的练习。 若要从地址向后拆解，应在比实际想要更远的点开始反汇编，然后向前看，直到指令开始生效。 前几个指令可能没有任何意义，因为你可能已在指令中间开始反汇编。 遗憾的是，反汇编可能永远不会与指令流同步，你必须尝试在不同的起点进行反汇编，直到找到有效的起点。

对于打包良好的 **switch** 语句，编译器将数据直接发出到代码流中，因此通过 **switch** 语句进行反汇编通常会偶然遇到 (没有意义的指令，因为它们实际上是数据) 。 找到数据的末尾，并继续在那里进行反汇编。

指令表示法

指令的一般表示法是将目标寄存器放在左侧，将源放在右侧。 但是，此规则可能有一些例外情况。

算术指令通常是由源寄存器和目标寄存器组合在一起的双寄存器。 结果将存储到目标中。

某些指令同时具有 16 位和 32 位版本，但此处仅列出了 32 位版本。 此处未列出仅在分段模型中使用的浮点指令、特权指令和指令 (Microsoft Win32 不使用) 。

为了节省空间，许多指令都以组合形式表示，如以下示例所示。

*	MOV	**r1**、 **r**/m/#n	**r1** = **r**/m/#n

表示第一个参数必须是寄存器，但第二个参数可以是寄存器、内存引用或即时值。

为了节省更多空间，还可以按如下所示表示指令。

*	MOV	**r1**/m、 **r**/m/#n	**r1**/m = **r**/m/#n

这意味着第一个参数可以是寄存器或内存引用，第二个参数可以是寄存器、内存引用或即时值。

除非另有说明，否则使用此缩写时，不能同时为源和目标选择内存。

此外，可以将位大小后缀 (8、16、32) 追加到源或目标，以指示参数必须具有该大小。 例如，r8 表示 8 位寄存器。

内存、数据传输和数据转换

内存和数据传输指令不会影响标志。

有效地址

*	Lea	**r**、m	加载有效地址。 (r = m) 的地址

例如，**LEA eax，[esi+4]** 表示 **eax esi** = + 4。 此指令通常用于执行算术。

数据传输

	MOV	**r1**/m、 **r2**/m/#n	**r1**/m = **r**/m/#n
	MOVSX	**r1**、 **r**/m	使用符号扩展名移动。
*	MOVZX	**r1**、 **r**/m	以零扩展名移动。

MOVSX 和 **MOVZX** 是 **mov** 指令的特殊版本，用于执行从源到目标的符号扩展或零扩展。 这是唯一允许源和目标为不同大小的指令。 (事实上，它们的大小必须不同。

堆栈操作

堆栈由 **esp** 寄存器指向。 **esp** 处的值是最近推送 (堆栈的顶部，首先弹出) ;较旧的堆栈元素驻留在更高的地址。

	推送	**r**/m/#n	将值推送到堆栈上。
	POP	**r**/m	堆栈中的 Pop 值。
	PUSHFD		将标志推送到堆栈上。
	POPFD		堆栈中的弹出标志。
	PUSHAD		推送所有整数寄存器。
	POPAD		弹出所有整数寄存器。
	Enter	#n、#n	生成堆栈帧。

*	离开	拆下堆栈帧

C/C++ 编译器不使用 **Enter** 指令。 (**Enter** 指令用于使用 Algol 或 Pascal.) 等语言实现嵌套过程

休假指令等效于：

```asm
mov esp, ebp
pop ebp
```

数据转换

CBW	将字节 (**al**) 转换为 word (**ax**) 。
CWD	将 word (**ax**) 转换为 dword (**dx**：**ax**) 。
CWDE	将 word (**ax**) 转换为 dword (**eax**) 。
CDQ	将 dword (**eax**) 转换为 qword (**edx**：**eax**) 。

所有转换都执行符号扩展。

算术和位操作

所有算术和位操作指令修改标志。

算法

	ADD	**r1**/m、 **r2**/m/#n	**r1**/m += **r2**/m/#n
	ADC	**r1**/m、 **r2**/m/#n	**r1**/m += **r2**/m/#n + 携带
	子项	**r1**/m、 **r2**/m/#n	**r1**/m -= **r2**/m/#n
	SBB	**r1**/m、 **r2**/m/#n	**r1**/m -= **r2**/m/#n + 携带
	负面	**r1**/m	**r1**/m = -**r1**/m
	公司	**r**/m	**r**/m += 1
	DEC	**r**/m	**r**/m -= 1

	Cmp	**r1**/m、 **r2**/m/#n	计算 **r1**/m - **r2**/m/#n

cmp 指令计算减法并根据结果设置标志，但会丢弃结果。 它通常后跟一个测试减法结果的条件 **跳转** 指令。

	MUL	**r**/m8	**斧头** = **铝** * **r**/m8
	MUL	**r**/m16	**dx**：**ax** = **ax** * **r**/m16
	MUL	**r**/m32	**edx**：**eax** = **eax** * **r**/m32
	IMUL	**r**/m8	**斧头** = **铝** * **r**/m8
	IMUL	**r**/m16	**dx**：**ax** = **ax** * **r**/m16
	IMUL	**r**/m32	**edx**：**eax** = **eax** * **r**/m32
	IMUL	**r1**、 **r2**/m	**r1** *= **r2**/m
	IMUL	**r1**、 **r2**/m、#n	**r1** = **r2**/m * #n

无符号乘法和有符号乘法。 乘法后的标志状态未定义。

	DIV	**r**/m8	(**ah**， **al**) = (**ax** % **r**/m8， **ax** / **r**/m8)
	DIV	**r**/m16	(**dx**， **ax**) = **dx**：**ax** / **r**/m16
	DIV	**r**/m32	(**edx**， **eax**) = **edx**：**eax** / **r**/m32
	IDIV	**r**/m8	(**ah**， **al**) = **ax** / **r**/m8
	IDIV	**r**/m16	(**dx**， **ax**) = **dx**：**ax** / **r**/m16
	IDIV	**r**/m32	(**edx**， **eax**) = **edx**：**eax** / **r**/m32

无符号和有符号除法。 伪代码解释中的第一个寄存器接收余数，第二个寄存器接收商。 如果结果溢出目标，则会生成除法溢出异常。

未定义除法后的标志状态。

*	SET*cc*	**r**/m8	将 **r**/m8 设置为 0 或 1

如果条件 *cc* 为 true，则 8 位值设置为 1。 否则，8 位值设置为零。

二进制编码的十进制

除非正在调试以 COBOL 编写的代码，否则不会看到这些说明。

	DAA	加法后的小数调整。
	DAS	减法后的小数调整。

这些指令在执行打包的二进制编码十进制运算后调整 **al** 寄存器。

Aaa	ASCII 在添加后进行调整。
AAS	ASCII 在减法后调整。

这些指令在执行解包的二进制编码十进制运算后调整 **al** 寄存器。

Aam	ASCII 在乘法后调整。
AAD	ASCII 在除法后调整。

这些指令在执行未打包的二进制编码十进制运算后调整 **al** 和 **ah** 寄存器。

位

	AND	**r1**/m、 **r2**/m/#n	**r1**/m = **r1**/m 和 **r2**/m/#n
	OR	**r1**/m、 **r2**/m/#n	**r1**/m = **r1**/m 或 **r2**/m/#n
	XOR	**r1**/m、 **r2**/m/#n	**r1**/m = **r1**/m xor **r2**/m/#n
	NOT	**r1**/m	**r1**/m = 按位非 **r1**/m
*	TEST	**r1**/m、 **r2**/m/#n	计算 **r1**/m 和 **r2**/m/#n

测试指令计算逻辑 AND 运算符并根据结果设置标志，但会丢弃结果。 它通常后跟一个条件跳转指令，用于测试逻辑 AND 的结果。

	SHL	**r1**/m、 **cl**/#n	**r1**/m <<= **cl**/#n
	Shr	**r1**/m、 **cl**/#n	**r1**/m >>= **cl**/#n 零填充

*	SAR	**r1**/m、 **cl**/#n	**r1**/m >>= **cl**/#n sign-fill

最后移出的位放置在携带中。

	SHLD	**r1**、 **r2**/m、 **cl**/#n	左双移。

按 **cl**/#n 左移 **r1**，用 **r2**/m 的顶部位填充。 最后移出的位放置在携带中。

SHRD	**r1**、 **r2**/m、 **cl**/#n	右双移位。

按 **cl**/#n 向右移动 **r1**，用 **r2**/m 的底部位填充。 最后移出的位放置在携带中。

ROL	**r1**、 **cl**/#n	按 **cl**/#n 向左旋转 **r1**。
Ror	**r1**、 **cl**/#n	按 **cl**/#n 向右旋转 **r1**。
RCL	**r1**、 **cl**/#n	按 **cl**/#n 向左旋转 **r1**/C。
RCR	**r1**、 **cl**/#n	按 **cl**/#n 向右旋转 **r1**/C。

旋转类似于移位，只不过移出的位作为传入的填充位重新出现。 旋转指令的 C 语言版本将携带位合并到旋转中。

BT	**r1**、 **r2**/#n	将 **r1** 的位 **r2**/#n复制到携带中。
Bts	**r1**、 **r2**/#n	设置 **r1** 的位 **r2**/#n，将上一个值复制到携带中。
Btc	**r1**、 **r2**/#n	清除 **r1** 的**位** **r2**/#n，将以前的值复制到携带中。

控制流

	J*cc*	dest	分支条件。
	JMP	dest	直接跳跃。
	JMP	**r**/m	跳转间接。
	CALL	dest	直接呼叫。
*	CALL	**r**/m	间接调用。

调用指令将返回地址推送到堆栈上，然后跳转到目标。

*	Ret	*#n*	返回

ret 指令弹出并跳转到堆栈上的返回地址。 **RET** 指令中的非零*#n*指示在弹出返回地址后，应将*#n*值添加到堆栈指针。

环	如果结果为非零，则递减 **ecx** 和跳转。
LOOPZ	如果结果为非零且**设置了 zr**，则递减 **ecx** 和 jump。
LOOPNZ	如果结果不为零且 **zr** 清晰，则递减 **ecx** 和跳转。
JECXZ	如果 **ecx** 为零，则跳转。

这些指令是 x86 的 CISC 遗产的残余，在最近的处理器中，实际上比写出的等效指令慢。

字符串操作

	MOVS*T*	将 *T* 从 **esi** 移动到 **edi。**
	CMPS*T*	将 **esi** 中的 *T* 与 **edi 进行比较。**
	SCAS*T*	从 **edi** 扫描 *T* 以获取 acc*T*。
	LODS*T*	将 *T* 从 **esi** 加载到 acc*T*。
	STOS*T*	将 *T* 存储到 acc T 中的 **edi。**

执行该操作后，根据方向标志 (向上或向下) 的方向标志的设置，按大小 (*T*) 递增或递减源寄存器和目标寄存器。

指令可以以 **REP** 作为前缀，以重复 **ecx** 寄存器指定的操作次数。

rep mov 指令用于复制内存块。

代表 stos 指令用于使用 acc*T* 填充内存块。

标志

LAHF	从标志加载 **ah** 。
SAHF	存储 **ah** 到标志。
STC	设置携带。

Clc	清除携带。
CMC	补码携带。
性病	将方向设置为 *向下。*
CLD	将方向设置为 *向上。*
性病	启用中断。
CLI	禁用中断。

互锁指令

XCHG	**r1**、 **r**/m	交换 **r1** 和 r/m。
XADD	**r1**、 **r**/m	将 **r1** 添加到 r/m，将原始值放在 **r1 中。**
CMPXCHG	**r1**、 **r**/m	比较和交换条件。

cmpxchg 指令是以下内容的原子版本：

```asm
   cmp     accT, r/m
   jz      match
   mov     accT, r/m
   jmp     done
match:
   mov     r/m, r1
done:
```

杂项

	INT	#n	捕获到内核。
	绑定	**r**、m	如果 **r** 不在范围内，则为陷阱。
*	NOP		无操作。
	XLATB		**al** = [**ebx** + **al**]
	BSWAP	**r**	寄存器中的交换字节顺序。

下面是 **int** 指令的特殊情况。

INT	3	调试器断点陷阱。

INT 3 的操作码0xCC。 NOP 的操作码0x90。

调试代码时，可能需要修补一些代码。 为此，可以将有问题的字节替换为0x90。

习语

	XOR	**r**、 **r**	**r** = 0
	TEST	**r**、 **r**	检查 **r** 是否 = 0。
*	ADD	**r**、 **r**	左移 **r** 1。

带批注的 x86 反汇编

项目 • 2024/02/14

以下部分将引导你完成反汇编示例。

源代码

下面是将分析的函数的代码。

C++

```
HRESULT CUserView::CloseView(void)
{
    if (m_fDestroyed) return S_OK;

    BOOL fViewObjectChanged = FALSE;
    ReleaseAndNull(&m_pdtgt);

    if (m_psv) {
        m_psb->EnableModelessSB(FALSE);
        if(m_pws) m_pws->ViewReleased();

        IShellView* psv;

        HWND hwndCapture = GetCapture();
        if (hwndCapture && hwndCapture == m_hwnd) {
            SendMessage(m_hwnd, WM_CANCELMODE, 0, 0);
        }

        m_fHandsOff = TRUE;
        m_fRecursing = TRUE;
        NotifyClients(m_psv, NOTIFY_CLOSING);
        m_fRecursing = FALSE;

        m_psv->UIActivate(SVUIA_DEACTIVATE);

        psv = m_psv;
        m_psv = NULL;

        ReleaseAndNull(&_pctView);

        if (m_pvo) {
            IAdviseSink *pSink;
            if (SUCCEEDED(m_pvo->GetAdvise(NULL, NULL, &pSink)) && pSink) {
                if (pSink == (IAdviseSink *)this)
                    m_pvo->SetAdvise(0, 0, NULL);
                pSink->Release();
            }
```

```
            fViewObjectChanged = TRUE;
            ReleaseAndNull(&m_pvo);
        }

        if (psv) {
            psv->SaveViewState();
            psv->DestroyViewWindow();
            psv->Release();
        }

        m_hwndView = NULL;
        m_fHandsOff = FALSE;

        if (m_pcache) {
            GlobalFree(m_pcache);
            m_pcache = NULL;
        }

        m_psb->EnableModelessSB(TRUE);

        CancelPendingActions();
    }

    ReleaseAndNull(&_psf);

    if (fViewObjectChanged)
        NotifyViewClients(DVASPECT_CONTENT, -1);

    if (m_pszTitle) {
        LocalFree(m_pszTitle);
        m_pszTitle = NULL;
    }

    SetRect(&m_rcBounds, 0, 0, 0, 0);
    return S_OK;
}
```

程序集代码

本部分包含带批注的反汇编示例。

使用 **ebp** 寄存器作为帧指针的函数开始如下：

dbgcmd

```
HRESULT CUserView::CloseView(void)
SAMPLE!CUserView__CloseView:
71517134 55              push    ebp
71517135 8bec            mov     ebp,esp
```

这会设置帧，以便函数可以将其参数作为 ebp **的**正偏移量进行访问，并将局部变量作为负偏移量进行访问。

这是专用 COM 接口上的一种方法，因此调用约定 **__stdcall。** 这意味着参数向右推送到左（在本例中，没有），将推送“this”指针，然后调用函数。 因此，在进入函数时，堆栈如下所示：

```
dbgcmd

[esp+0] = return address
[esp+4] = this
```

在上述两条指令之后，参数可访问：

```
dbgcmd

[ebp+0] = previous ebp pushed on stack
[ebp+4] = return address
[ebp+8] = this
```

对于使用 **ebp** 作为帧指针的函数，可在 [**ebp**+8] 访问第一个推送参数;后续参数可在连续更高的 DWORD 地址中访问。

```
dbgcmd

71517137 51              push    ecx
71517138 51              push    ecx
```

此函数只需要两个局部堆栈变量，因此一个 **子 esp，8** 指令。 然后，推送的值可用作 [**ebp-4**] 和 [**ebp-8**]。

对于使用 **ebp** 作为帧指针的函数，可通过 ebp **寄存器的**负偏移量访问堆栈局部变量。

```
dbgcmd

71517139 56              push    esi
```

现在，编译器将保存跨函数调用保留所需的寄存器。 实际上，它会将它们保存在位和片段中，与实际代码的第一行交错。

```
dbgcmd

7151713a 8b7508          mov     esi,[ebp+0x8]     ; esi = this
7151713d 57              push    edi               ; save another registers
```

因此，CloseView 是 ViewState 上的方法，该方法在基础对象中的偏移量为 12。 因此，**这是** 一个指向 ViewState 类的指针，尽管当可能与其他基类混淆时，它将被更仔细地指定为 （ViewState*） **这**一点。

```
dbgcmd
```

```
    if (m_fDestroyed)
7151713e 33ff            xor     edi,edi           ; edi = 0
```

XORing 寄存器本身是将其归零的标准方法。

```
dbgcmd
```

```
71517140 39beac000000    cmp     [esi+0xac],edi    ; this->m_fDestroyed ==
0?
71517146 7407            jz      NotDestroyed (7151714f)  ; jump if equal
```

cmp 指令比较两个值（通过减去它们）。 **如果结果为零，则 jz** 指令检查，指示两个比较值相等。

cmp 指令比较两个值：后续 j 指令会根据比较结果跳转。

```
dbgcmd
```

```
    return S_OK;
71517148 33c0            xor     eax,eax           ; eax = 0 = S_OK
7151714a e972010000      jmp     ReturnNoEBX (715172c1) ; return, do not
pop EBX
```

编译器延迟保存 EBX 寄存器，直到函数的后面部分，因此，如果程序要在此测试中“提前”，则退出路径必须是不还原 EBX 的路径。

```
dbgcmd
```

```
    BOOL fViewObjectChanged = FALSE;
    ReleaseAndNull(&m_pdtgt);
```

这两行代码的执行是交错的，因此请注意。

```
dbgcmd
```

```
NotDestroyed:
7151714f 8d86c0000000    lea     eax,[esi+0xc0]    ; eax = &m_pdtgt
```

lea 指令计算内存访问的效果地址，并将其存储在目标中。 实际内存地址未取消引用。

lea 指令采用变量的地址。

```
dbgcmd

71517155 53              push    ebx
```

应在损坏 EBX 寄存器之前保存该寄存器。

```
dbgcmd

71517156 8b1d10195071    mov ebx,[_imp__ReleaseAndNull]
```

由于经常调用 **ReleaseAndNull**， 因此最好在 EBX 中缓存其地址。

```
dbgcmd

7151715c 50              push    eax              ; parameter to
ReleaseAndNull
7151715d 897dfc          mov     [ebp-0x4],edi    ; fViewObjectChanged =
FALSE
71517160 ffd3            call    ebx              ; call ReleaseAndNull
    if (m_psv) {
71517162 397e74          cmp     [esi+0x74],edi   ; this->m_psv == 0?
71517165 0f8411010000    je      No_Psv (7151727c) ; jump if zero
```

请记住，在回退一段时间后将 EDI 寄存器归零，EDI 是跨函数调用保留的寄存器（因此对 ReleaseAndNull **的调用**没有更改它）。 因此，它仍保留值零，可以使用该值快速测试零。

```
dbgcmd

        m_psb->EnableModelessSB(FALSE);
7151716b 8b4638          mov     eax,[esi+0x38]   ; eax = this->m_psb
7151716e 57              push    edi              ; FALSE
7151716f 50              push    eax              ; "this" for callee
71517170 8b08            mov     ecx,[eax]        ; ecx = m_psb->lpVtbl
71517172 ff5124          call    [ecx+0x24]       ; __stdcall
EnableModelessSB
```

上述模式是 COM 方法调用的指示符号。

COM 方法调用非常受欢迎，因此最好学会识别它们。 具体而言，应该能够直接从其 Vtable 偏移量识别三个 IUnknown 方法：QueryInterface=0、AddRef=4 和 Release=8。

```
dbgcmd
```

```
        if(m_pws) m_pws->ViewReleased();
71517175 8b8614010000     mov     eax,[esi+0x114]   ; eax = this->m_pws
7151717b 3bc7             cmp     eax,edi           ; eax == 0?
7151717d 7406             jz      NoWS (71517185) ; if so, then jump
7151717f 8b08             mov     ecx,[eax]         ; ecx = m_pws->lpVtbl
71517181 50               push    eax               ; "this" for callee
71517182 ff510c           call    [ecx+0xc]         ; __stdcall ViewReleased
NoWS:
        HWND hwndCapture = GetCapture();
71517185 ff15e01a5071    call [_imp__GetCapture]    ; call GetCapture
```

通过全局的间接调用是在 Microsoft Win32 中实现函数导入的方式。 加载程序修复了全局，以指向目标的实际地址。 这是一种方便的方法，用于在调查崩溃的机器时获取轴承。 查找对导入的函数和目标中的调用。 通常，你将拥有一些导入的函数的名称，可用于确定源代码中的位置。

dbgcmd

```
        if (hwndCapture && hwndCapture == m_hwnd) {
            SendMessage(m_hwnd, WM_CANCELMODE, 0, 0);
        }
7151718b 3bc7             cmp     eax,edi           ; hwndCapture == 0?
7151718d 7412             jz      No_Capture (715171a1) ; jump if zero
```

函数返回值放置在 EAX 寄存器中。

dbgcmd

```
7151718f 8b4e44           mov     ecx,[esi+0x44]    ; ecx = this->m_hwnd
71517192 3bc1             cmp     eax,ecx           ; hwndCapture = ecx?
71517194 750b             jnz     No_Capture (715171a1) ; jump if not

71517196 57               push    edi               ; 0
71517197 57               push    edi               ; 0
71517198 6a1f             push    0x1f              ; WM_CANCELMODE
7151719a 51               push    ecx               ; hwndCapture
7151719b ff1518195071     call    [_imp__SendMessageW] ; SendMessage
No_Capture:
        m_fHandsOff = TRUE;
        m_fRecursing = TRUE;
715171a1 66818e0c0100000180 or    word ptr [esi+0x10c],0x8001 ; set both
flags at once

        NotifyClients(m_psv, NOTIFY_CLOSING);
715171aa 8b4e20           mov     ecx,[esi+0x20]    ; ecx =
(CNotifySource*)this.vtbl
715171ad 6a04             push    0x4               ; NOTIFY_CLOSING
715171af 8d4620           lea     eax,[esi+0x20]    ; eax =
(CNotifySource*)this
715171b2 ff7674           push    [esi+0x74]        ; m_psv
```

```
715171b5 50              push    eax             ; "this" for callee
715171b6 ff510c          call    [ecx+0xc]       ; __stdcall
NotifyClients
```

请注意，在与你自己的不同基类上调用方法时，必须如何更改“this”指针。

dbgcmd

```
        m_fRecursing = FALSE;
715171b9 80a60d0100007f  and     byte ptr [esi+0x10d],0x7f
        m_psv->UIActivate(SVUIA_DEACTIVATE);
715171c0 8b4674          mov     eax,[esi+0x74]  ; eax = m_psv
715171c3 57              push    edi             ; SVUIA_DEACTIVATE = 0
715171c4 50              push    eax             ; "this" for callee
715171c5 8b08            mov     ecx,[eax]       ; ecx = vtbl
715171c7 ff511c          call    [ecx+0x1c]      ; __stdcall UIActivate
        psv = m_psv;
        m_psv = NULL;
715171ca 8b4674          mov     eax,[esi+0x74]  ; eax = m_psv
715171cd 897e74          mov     [esi+0x74],edi  ; m_psv = NULL
715171d0 8945f8          mov     [ebp-0x8],eax   ; psv = eax
```

第一个局部变量是 **psv**。

dbgcmd

```
        ReleaseAndNull(&_pctView);
715171d3 8d466c          lea     eax,[esi+0x6c]  ; eax = &_pctView
715171d6 50              push    eax             ; parameter
715171d7 ffd3            call    ebx             ; call ReleaseAndNull
        if (m_pvo) {
715171d9 8b86a8000000    mov     eax,[esi+0xa8]  ; eax = m_pvo
715171df 8dbea8000000    lea     edi,[esi+0xa8]  ; edi = &m_pvo
715171e5 85c0            test    eax,eax         ; eax == 0?
715171e7 7448            jz      No_Pvo (71517231) ; jump if zero
```

请注意，编译器推测性地准备好了m_pvo**成员的**地址，因为你将经常使用它一段时间。因此，使用方便的地址将导致较小的代码。

dbgcmd

```
            if (SUCCEEDED(m_pvo->GetAdvise(NULL, NULL, &pSink)) && pSink) {
715171e9 8b08            mov     ecx,[eax]       ; ecx = m_pvo->lpVtbl
715171eb 8d5508          lea     edx,[ebp+0x8]   ; edx = &pSink
715171ee 52              push    edx             ; parameter
715171ef 6a00            push    0x0             ; NULL
715171f1 6a00            push    0x0             ; NULL
715171f3 50              push    eax             ; "this" for callee
715171f4 ff5120          call    [ecx+0x20]      ; __stdcall GetAdvise
715171f7 85c0            test    eax,eax         ; test bits of eax
```

```
715171f9 7c2c            jl      No_Advise (71517227) ; jump if less than
zero
715171fb 33c9            xor     ecx,ecx           ; ecx = 0
715171fd 394d08          cmp     [ebp+0x8],ecx     ; _pSink == ecx?
71517200 7425            jz      No_Advise (71517227)
```

请注意，编译器得出结论，传入的“this”参数不是必需的（因为它很久以前就将其藏在 ESI 寄存器中）。 因此，它将内存重用为局部变量 pSink。

如果函数使用 EBP 帧，则传入的参数将到达 EBP 的正偏移量，局部变量放置在负偏移量处。 但是，在这种情况下，编译器可以自由重复使用该内存，以用于任何目的。

如果密切关注，你将看到编译器可以更好地优化此代码。 它可能会延迟 **lea edi、[esi+0xa8]** 指令，直到两 **个推送0x0** 指令之后，将其替换为 **push edi**。 这会保存 2 个字节。

dbgcmd

```
                if (pSink == (IAdviseSink *)this)
```

接下来的几行是为了弥补 C++ 中 （IAdviseSink *） **NULL 必须为 NULL** 这一事实。 因此，如果“this”确实为“（ViewState*）NULL”，则转换的结果应为 **NULL**，而不是 IAdviseSink 和 IBrowserService 之间的距离。

dbgcmd

```
71517202 8d46ec          lea     eax,[esi-0x14]    ; eax = -
(IAdviseSink*)this
71517205 8d5614          lea     edx,[esi+0x14]    ; edx =
(IAdviseSink*)this
71517208 f7d8            neg     eax               ; eax = -eax (sets carry
if != 0)
7151720a 1bc0            sbb     eax,eax           ; eax = eax - eax -
carry
7151720c 23c2            and     eax,edx           ; eax = NULL or edx
```

尽管Pentum 有条件移动指令，但基本 i386 体系结构没有，因此编译器使用特定技术来模拟条件移动指令，而无需进行任何跳转。

条件评估的一般模式如下：

dbgcmd

```
        neg     r
        sbb     r, r
```

```
        and     r, (val1 - val2)
        add     r, val2
```

如果 r 为非零，则 neg r 设置携带标志，因为 **neg** 通过从零减去来否定值。 如果减去非零值，则从零减去将生成借款（设置携带）。 它还会损害 r **寄存器中的**值，但这是可以接受的，因为你无论如何都即将覆盖它。

接下来， **sbb r、r** 指令从自身中减去一个值，这始终会导致零。 但是，它还减去携带（借款）位，因此净结果是将 r **设置为**零或 -1，具体取决于携带是明确还是设置。

因此，**如果 r 的原始值为零，则 sbb r 将 r 设置为零，如果原始值为**非零，则设置为 -1。

第三个指令执行掩码。 **由于 r** 寄存器为零或 -1，因此"this"用于离开 **r** 零或将 r **从 -1 更改为 （val1 - val1），**因为 ANDing 任何值都保留原始值。

因此，如果 r 的原始值为零，则"and r， （val1 - val1）"的结果是将 r 设置为零;如果 r 的原始值为非零，则设置为"（val1 - val2）。"

最后，将 val2 添加到 **r，导致 val2** 或 **（val1 - val2） + val2 = val1。**

因此，此系列指令的最终结果是，如果它最初为零，则设置为 **val2**;如果它不是零，则设置为 **val1。** 这是 r = r？ val1 ： val2 **的程序集等效项。**

在此特定实例中，可以看到 **val2 = 0** 和 **val1 = （IAdviseSink*）这**一点。 （请注意，编译器已执行最终 **添加 eax，0** 指令，因为它不起作用。

dbgcmd

```
7151720e 394508          cmp     [ebp+0x8],eax ; pSink ==
(IAdviseSink*)this?
71517211 750b            jnz     No_SetAdvise (7151721e) ; jump if not
equal
```

在本部分的前面部分，将 EDI 设置为m_pvo**成员的**地址。 现在你将使用它。 前面还清除了 ECX 寄存器。

dbgcmd

```
                    m_pvo->SetAdvise(0, 0, NULL);
71517213 8b07            mov     eax,[edi]         ; eax = m_pvo
71517215 51              push    ecx               ; NULL
71517216 51              push    ecx               ; 0
71517217 51              push    ecx               ; 0
71517218 8b10            mov     edx,[eax]         ; edx = m_pvo->lpVtbl
7151721a 50              push    eax               ; "this" for callee
7151721b ff521c          call    [edx+0x1c]        ; __stdcall SetAdvise
No_SetAdvise:
                pSink->Release();
```

```
7151721e 8b4508          mov     eax,[ebp+0x8]     ; eax = pSink
71517221 50              push    eax               ; "this" for callee
71517222 8b08            mov     ecx,[eax]         ; ecx = pSink->lpVtbl
71517224 ff5108          call    [ecx+0x8]         ; __stdcall Release
No_Advise:
```

所有这些 COM 方法调用都应非常熟悉。

接下来两个语句的计算是交错的。 不要忘记 EBX 包含 ReleaseAndNull **的**地址。

dbgcmd

```
            fViewObjectChanged = TRUE;
            ReleaseAndNull(&m_pvo);
71517227 57              push    edi               ; &m_pvo
71517228 c745fc01000000  mov     dword ptr [ebp-0x4],0x1 ;
fViewObjectChanged = TRUE
7151722f ffd3            call    ebx               ; call ReleaseAndNull
No_Pvo:
        if (psv) {
71517231 8b7df8          mov     edi,[ebp-0x8]     ; edi = psv
71517234 85ff            test    edi,edi           ; edi == 0?
71517236 7412            jz      No_Psv2 (7151724a) ; jump if zero
            psv->SaveViewState();
71517238 8b07            mov     eax,[edi]         ; eax = psv->lpVtbl
7151723a 57              push    edi               ; "this" for callee
7151723b ff5034          call    [eax+0x34]        ; __stdcall
SaveViewState
```

以下是更多的 COM 方法调用。

dbgcmd

```
            psv->DestroyViewWindow();
7151723e 8b07            mov     eax,[edi]         ; eax = psv->lpVtbl
71517240 57              push    edi               ; "this" for callee
71517241 ff5028          call    [eax+0x28]        ; __stdcall
DestroyViewWindow
            psv->Release();
71517244 8b07            mov     eax,[edi]         ; eax = psv->lpVtbl
71517246 57              push    edi               ; "this" for callee
71517247 ff5008          call    [eax+0x8]         ; __stdcall Release
No_Psv2:
        m_hwndView = NULL;
7151724a 83667c00        and     dword ptr [esi+0x7c],0x0 ; m_hwndView = 0
```

将零的内存位置与将内存位置设置为零相同，因为任何 AND 零都是零。 编译器使用此窗体，因为即使速度较慢，它比等效 **的 mov** 指令要短得多。 （此代码已针对大小而不是速度进行优化。

dbgcmd

```
        m_fHandsOff = FALSE;
7151724e 83a60c010000fe   and     dword ptr [esi+0x10c],0xfe
        if (m_pcache) {
71517255 8b4670           mov     eax,[esi+0x70]    ; eax = m_pcache
71517258 85c0             test    eax,eax           ; eax == 0?
7151725a 740b             jz      No_Cache (71517267) ; jump if zero
            GlobalFree(m_pcache);
7151725c 50               push    eax               ; m_pcache
7151725d ff15b4135071     call    [_imp__GlobalFree]    ; call GlobalFree
            m_pcache = NULL;
71517263 83667000         and     dword ptr [esi+0x70],0x0 ; m_pcache = 0
No_Cache:
        m_psb->EnableModelessSB(TRUE);
71517267 8b4638           mov     eax,[esi+0x38]    ; eax = this->m_psb
7151726a 6a01             push    0x1               ; TRUE
7151726c 50               push    eax               ; "this" for callee
7151726d 8b08             mov     ecx,[eax]         ; ecx = m_psb->lpVtbl
7151726f ff5124           call    [ecx+0x24]        ; __stdcall
EnableModelessSB
        CancelPendingActions();
```

若要调用 **CancelPendingActions**，必须从 （ViewState*） 移动到 （CUserView*）此对象。 另请注意， **CancelPendingActions** 使用__thiscall调用约定，而不是__stdcall。 根据__thiscall，“this”指针在 ECX 寄存器中传递，而不是在堆栈上传递。

dbgcmd

```
71517272 8d4eec           lea     ecx,[esi-0x14]    ; ecx = (CUserView*)this
71517275 e832fbffff       call CUserView::CancelPendingActions (71516dac) ;
__thiscall
    ReleaseAndNull(&_psf);
7151727a 33ff             xor     edi,edi           ; edi = 0 (for later)
No_Psv:
7151727c 8d4678           lea     eax,[esi+0x78]    ; eax = &_psf
7151727f 50               push    eax               ; parameter
71517280 ffd3             call    ebx               ; call ReleaseAndNull
    if (fViewObjectChanged)
71517282 397dfc           cmp     [ebp-0x4],edi     ; fViewObjectChanged ==
0?
71517285 740d             jz      NoNotifyViewClients (71517294) ; jump if
zero
       NotifyViewClients(DVASPECT_CONTENT, -1);
71517287 8b46ec           mov     eax,[esi-0x14]    ; eax =
((CUserView*)this)->lpVtbl
7151728a 8d4eec           lea     ecx,[esi-0x14]    ; ecx = (CUserView*)this
7151728d 6aff             push    0xff              ; -1
7151728f 6a01             push    0x1               ; DVASPECT_CONTENT = 1
71517291 ff5024           call    [eax+0x24]        ; __thiscall
NotifyViewClients
NoNotifyViewClients:
```

```
    if (m_pszTitle)
71517294 8b8680000000     mov     eax,[esi+0x80]    ; eax = m_pszTitle
7151729a 8d9e80000000     lea     ebx,[esi+0x80]    ; ebx = &m_pszTitle (for
later)
715172a0 3bc7             cmp     eax,edi           ; eax == 0?
715172a2 7409             jz      No_Title (715172ad) ; jump if zero
        LocalFree(m_pszTitle);
715172a4 50               push    eax               ; m_pszTitle
715172a5 ff1538125071     call   [_imp__LocalFree]
        m_pszTitle = NULL;
```

请记住，EDI 仍然为零，EBX 仍为 &m_pszTitle，因为这些寄存器由函数调用保留。

dbgcmd

```
715172ab 893b             mov     [ebx],edi         ; m_pszTitle = 0
No_Title:
    SetRect(&m_rcBounds, 0, 0, 0, 0);
715172ad 57               push    edi               ; 0
715172ae 57               push    edi               ; 0
715172af 57               push    edi               ; 0
715172b0 81c6fc000000     add     esi,0xfc          ; esi = &this-
>m_rcBounds
715172b6 57               push    edi               ; 0
715172b7 56               push    esi               ; &m_rcBounds
715172b8 ff15e41a5071     call   [_imp__SetRect]
```

请注意，不再需要“this”的值，因此编译器使用 **添加** 指令来就地修改它，而不是使用另一个寄存器来保存地址。 这实际上是一场性能的胜利，因为 V 管道可以进行算术，但不能解决计算问题。

dbgcmd

```
    return S_OK;
715172be 33c0             xor     eax,eax           ; eax = S_OK
```

最后，还原需要保留的寄存器、清理堆栈并返回到调用方，删除传入参数。

dbgcmd

```
715172c0 5b               pop     ebx               ; restore
ReturnNoEBX:
715172c1 5f               pop     edi               ; restore
715172c2 5e               pop     esi               ; restore
715172c3 c9               leave                     ; restores EBP and ESP
simultaneously
715172c4 c20400           ret     0x4               ; return and clear
parameters
```

x64 处理器

项目 • 2023/06/15

本节包括：

x64 体系结构

x64 指令

带批注的 x64 反汇编

注意 x64 处理器体系结构有时称为“AMD64”、“x86-64”、“AMD x86-64”或“Intel64”。

x64 体系结构

项目 • 2023/12/21

x64 体系结构是 x86 的向后兼容扩展。 它提供新的 64 位模式和旧版 32 位模式，这与 x86 相同。

术语“x64”包括 AMD 64 和 Intel64。 指令集几乎完全相同。

寄存器

x64 将 x86 的 8 个常规用途寄存器扩展为 64 位，并添加了 8 个新的 64 位寄存器。 64 位寄存器的名称以“r”开头。 例如， **eax** 的 64 位扩展称为 **rax**。 新寄存器名为 **r8** 到 **r15**。

每个寄存器的低 32 位、16 位和 8 位可直接在操作数中寻址。 这包括寄存器，如 **esi**，其低 8 位以前无法寻址。 下表指定 64 位寄存器较低部分的汇编语言名称。

展开表

64 位寄存器	低 32 位	低 16 位	低 8 位
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	Dx	Dl
rsi	Esi	si	sil
rdi	Edi	di	dil
rbp	Ebp	bp	bpl
粒子	Esp	sp	Spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b

64 位寄存器	低 32 位	低 16 位	低 8 位
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

输出到 32 位子注册的操作会自动向整个 64 位寄存器零扩展。 输出到 8 位或 16 位子注册的操作不是零扩展的， (这与 x86 行为) 兼容。

ax、**bx**、**cx** 和 **dx** 的高 8 位仍可寻址为 **ah**、**bh**、**ch**、**dh**，但不能与所有类型的操作数一起使用。

指令指针 **eip** 和 **标志** 寄存器已分别扩展到 64 位， (**rip** 和 **rflags**，分别) 。

x64 处理器还提供多组浮点寄存器：

- 八个 80 位 x87 寄存器。
- 八个 64 位 MMX 寄存器。 (这些寄存器与 x87 registers 重叠。)
- 原来的一组 8 个 128 位 SSE 寄存器增加到 16 个。

调用约定

与 x86 不同，C/C++ 编译器仅支持 x64 上的一个调用约定。 此调用约定利用 x64 上可用的寄存器数量增加：

- 前四个整数或指针参数在 **rcx**、 **rdx**、 **r8** 和 **r9** 寄存器中传递。
- 前四个浮点参数在前四个 SSE 寄存器 **xmm0**-**xmm3** 中传递。
- 调用方在堆栈上为寄存器中传递的参数保留空间。 调用的函数可以使用此空间将寄存器的内容溢出到堆栈。
- 任何其他参数都在堆栈上传递。
- 整数或指针返回值在 **rax** 寄存器中返回，而浮点返回值以 **xmm0** **返回**。
- **rax**、 **rcx**、 **rdx**、 **r8**-**r11** 是可变的。
- **rbx**、 **rbp**、 **rdi**、 **rsi**、 **r12**-**r15** 是非易失性。

C++ 的调用约定类似。 **此**指针作为隐式第一个参数传递。 接下来的三个参数在剩余寄存器中传递，其余参数在堆栈上传递。

寻址模式

64 位模式下的寻址模式与 x86 相似，但不完全相同。

- 引用 64 位寄存器的指令以 64 位精度自动执行。 例如， **mov rax，[rbx]** 将 8 个字节从 **rbx** 开始移动到 **rax**。
- 为 64 位即时常量或常量地址添加了一种特殊形式的 **mov** 指令。 对于所有其他指令，即时常量或常量地址仍为 32 位。
- x64 提供了新的 **相对翻录**寻址模式。 引用单个常量地址的指令编码为 **翻录**的偏移量。 例如， **mov rax，** [*addr*] 指令将从 *addr* + **rip** 开始的 8 个字节移动到 **rax**。

隐式引用指令指针的指令（如 **jmp**、 **调用**、 **推送**和 **弹出**）在 x64 上将其视为 64 位寄存器。

另请参阅

- X86-64 维基百科
- AMD 64 开发人员资源
- Intel - x64 程序集简介
- x64 入门 - 开始编程 64 位 Windows 系统所需的全部知识 - Matt Pietrek
- 调用约定的历史，第 5 部分：amd64 Raymond Chen

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

Windows driver documentation 反馈

Windows driver documentation 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

x64 指令

项目 • 2023/06/15

大多数 x86 指令在 64 位模式下仍对 x64 有效。 64 位模式下不再支持某些很少使用的操作，例如：

- 二进制编码的十进制算术指令：AAA、AAD、AAM、AAS、DAA、DAS
- 绑定
- PUSHAD 和 POPAD
- 处理段寄存器的大多数操作，例如 PUSH DS 和 POP DS。 使用 FS 或 GS 段寄存器的 (操作仍然有效。)

x64 指令集包括 x86 的最新新增内容，例如 SSE 2。 为 x64 编译的程序可以自由使用这些说明。

数据传输

x64 提供了 MOV 指令的新变体，可以处理 64 位即时常量或内存地址。

MOV	**r**，#n	**r** = #n
MOV	**rax**，m	将 64 位地址处的内容移动到 **rax**。
MOV	m、 **rax**	将 **rax** 的内容移动到 64 位地址。

x64 还提供了将 32 位操作数签名为 64 位签名的新指令。

MOVSXD	**r1**、 **r**/m	将带符号扩展名的 DWORD 移动到 QWORD。

32 位子注册表中的普通 MOV 操作会自动零扩展到 64 位，因此没有 MOVZXD 指令。

可以使用两个 SSE 指令将 128 位值（（例如) GUID）从内存移动到 **xmm***n* 寄存器，反之亦然。

MOVDQA	**r1**/m、 **r2**/m	将 128 位对齐值移动到 **xmm***n* 寄存器，反之亦然。
MOVDQU	**r1**/m、 **r2**/m	移动 128 位值 (不一定) 进行注

	册，反之亦然。

数据转换

CDQE	将 dword (**eax**) 转换为 qword (**rax**) 。
CQO	将 qword (**rax**) 转换为 oword (**rdx**：**rax**) 。

字符串操作

MOVSQ	将 qword 从 **rsi** 移动到 **rdi**。
CMPSQ	比较 **rsi** 处的 qword 与 **rdi**。
SCASQ	在 **rdi** 处扫描 qword。 将 **rdi** 处的 qword 与 **rax** 进行比较。
LODSQ	将 qword 从 **rsi** 加载到 **rax***中*。
STOSQ	将 qword 存储到 **rax** 中的 **rdi**。

带批注的 x64 反汇编

项目 • 2024/03/01

以下这个非常简单的函数说明了 x64 调用约定。

C++

```
int Simple(int i, int j)
{
    return i*5 + j + 3;
}
```

这会编译为如下所示的代码：

dbgcmd

```
01001080 lea     eax,[rdx+rcx*4]        ; eax = rdx+rcx*4
01001083 lea     eax,[rcx+rax+0x3]      ; eax = rcx+rax+3
01001087 ret
```

i 和 *j* 参数分别在 **ecx** 和 **edx** 寄存器中传递。 由于只有两个参数，因此例程根本不使用堆栈。

生成的特定代码利用了三个技巧，其中一个特定于 x64：

1. **lea** 运算可用于将一系列简单的算术运算作为单个运算来执行。 第一个指令将 *j*+*i**4 存储在 **eax** 中，第二个指令将 *i*+3 添加到结果中，总共为 *j*+*i**5+3。
2. 许多运算（如加法和乘法）可以通过额外的精度完成，然后截断到正确的精度。 在此实例中，代码使用 64 位加法和乘法。 我们可以安全地将结果截断为 32 位。
3. 在 x64 上，输出到 32 位寄存器的任何操作都会自动零扩展结果。 在这种情况下，输出到 **eax** 的效果是将结果截断为 32 位。

返回值在 **rax** 寄存器中传递。 在这种情况下，结果已在 **rax** 寄存器中，因此函数返回。

接下来，我们考虑一个更复杂的函数来演示典型的 x64 反汇编：

C++

```
HRESULT Meaningless(IDispatch *pdisp, DISPID dispid, BOOL fUnique, LPCWSTR
pszExe)
{
    IQueryAssociations *pqa;
    HRESULT hr = AssocCreate(CLSID_QueryAssociations,
```

```
IID_IQueryAssociations, (void**)&pqa);
    if (SUCCEEDED(hr)) {
        hr = pqa->Init(ASSOCF_INIT_BYEXENAME, pszExe, NULL, NULL);
        if (SUCCEEDED(hr)) {
            WCHAR wszName[MAX_PATH];
            DWORD cchName = MAX_PATH;
            hr = pqa->GetString(0, ASSOCSTR_FRIENDLYAPPNAME, NULL, wszName,
&cchName);
            if (SUCCEEDED(hr)) {
                VARIANTARG rgvarg[2] = { 0 };
                V_VT(&rgvarg[0]) = VT_BSTR;
                V_BSTR(&rgvarg[0]) = SysAllocString(wszName);
                if (V_BSTR(&rgvarg[0])) {
                    DISPPARAMS dp;
                    LONG lUnique = InterlockedIncrement(&lCounter);
                    V_VT(&rgvarg[1]) = VT_I4;
                    V_I4(&rgvarg[1]) = fUnique ? lUnique : 0;
                    dp.rgvarg = rgvarg;
                    dp.cArgs = 2;
                    dp.rgdispidNamedArgs = NULL;
                    dp.cNamedArgs = 0;
                    hr = pdisp->Invoke(dispid, IID_NULL, 0, DISPATCH_METHOD,
&dp, NULL, NULL, NULL);
                    VariantClear(&rgvarg[0]);
                    VariantClear(&rgvarg[1]);
                } else {
                    hr = E_OUTOFMEMORY;
                }
            }
        }
        pqa->Release();
    }
    return hr;
}
```

我们将逐行浏览此函数和等效的装配。

输入后，函数的参数存储如下：

- **rcx** = *pdisp*。
- **rdx** = *dispid*.
- **r8** = *fUnique*.
- **r9** = *pszExe*.

回想一下，前四个参数是在寄存器中传递的。 由于此函数只有四个参数，因此不会在堆栈上传递任何参数。

装配开始如下：

dbgcmd

```
Meaningless:
010010e0 push    rbx                    ; save
010010e1 push    rsi                    ; save
010010e2 push    rdi                    ; save
010010e3 push    r12d                   ; save
010010e5 push    r13d                   ; save
010010e7 push    r14d                   ; save
010010e9 push    r15d                   ; save
010010eb sub     rsp,0x2c0              ; reserve stack
010010f2 mov     rbx,r9                 ; rbx = pszExe
010010f5 mov     r12d,r8d               ; r12 = fUnique (zero-extend)
010010f8 mov     r13d,edx               ; r13 = dispid  (zero-extend)
010010fb mov     rsi,rcx                ; rsi = pdisp
```

该函数首先保存非易失性寄存器，然后为局部变量预留堆栈空间。 然后，它将参数保存在非易失寄存器中。 请注意，中间两个 **mov** 指令的目标为 32 位寄存器，因此它们隐式零扩展到 64 位。

dbgcmd

```
    IQueryAssociations *pqa;
    HRESULT hr = AssocCreate(CLSID_QueryAssociations,
IID_IQueryAssociations, (void**)&pqa);
```

AssocCreate 的第一个参数是通过值传递的 128 位 CLSID。 由于这在 64 位寄存器并不适合，因此 CLSID 会被复制到堆栈，并传递指向堆栈位置的指针。

dbgcmd

```
010010fe movdqu  xmm0,oword ptr [CLSID_QueryAssociations (01001060)]
01001106 movdqu  oword ptr [rsp+0x60],xmm0  ; temp buffer for first
parameter
0100110c lea     r8,[rsp+0x58]          ; arg3 = &pqa
01001111 lea rdx,[IID_IQueryAssociations (01001070)] ; arg2 =
&IID_IQueryAssociations
01001118 lea     rcx,[rsp+0x60]         ; arg1 = &temporary
0100111d call qword ptr [_imp_AssocCreate (01001028)] ; call
```

movdqu 指令将 128 位值传入和传出 **xmm***n* 寄存器。 在此实例中，汇编代码使用它将 CLSID 复制到堆栈。 指向 CLSID 的指针在 **r8** 中传递。 另外两个参数在 **rcx** 和 **rdx** 中传递。

dbgcmd

```
    if (SUCCEEDED(hr)) {
```

```
01001123 test    eax,eax
01001125 jl      ReturnEAX (01001281)
```

代码检查返回值是否成功。

dbgcmd

```
        hr = pqa->Init(ASSOCF_INIT_BYEXENAME, pszExe, NULL, NULL);

0100112b mov     rcx,[rsp+0x58]         ; arg1 = pqa
01001130 mov     rax,[rcx]              ; rax = pqa.vtbl
01001133 xor     r14d,r14d              ; r14 = 0
01001136 mov     [rsp+0x20],r14         ; arg5 = 0
0100113b xor     r9d,r9d                ; arg4 = 0
0100113e mov     r8,rbx                 ; arg3 = pszExe
01001141 mov     r15d,0x2               ; r15 = 2 (for later)
01001147 mov     edx,r15d               ; arg2 = 2 (ASSOCF_INIT_BY_EXENAME)
0100114a call    qword ptr [rax+0x18]   ; call Init method
```

这是使用 C++ vtable 的间接函数调用。 此 **this** 指针作为第一个参数传入 **rcx**。 前三个参数在寄存器中传递，而最终参数在堆栈上传递。 该函数为寄存器中传递的参数保留 16 字节，因此第五个参数从 **rsp**+0x20 开始。

dbgcmd

```
        if (SUCCEEDED(hr)) {

0100114d mov     ebx,eax                ; ebx = hr
0100114f test    ebx,ebx                ; FAILED?
01001151 jl      ReleasePQA (01001274)  ; jump if so
```

汇编语言代码将结果保存在 **ebx** 中，并检查它是否为成功代码。

dbgcmd

```
            WCHAR wszName[MAX_PATH];
            DWORD cchName = MAX_PATH;
            hr = pqa->GetString(0, ASSOCSTR_FRIENDLYAPPNAME, NULL, wszName,
&cchName);
            if (SUCCEEDED(hr)) {

01001157 mov     dword ptr [rsp+0x50],0x104 ; cchName = MAX_PATH
0100115f mov     rcx,[rsp+0x58]         ; arg1 = pqa
01001164 mov     rax,[rcx]              ; rax = pqa.vtbl
01001167 lea     rdx,[rsp+0x50]         ; rdx = &cchName
0100116c mov     [rsp+0x28],rdx         ; arg6 = cchName
01001171 lea     rdx,[rsp+0xb0]         ; rdx = &wszName[0]
01001179 mov     [rsp+0x20],rdx         ; arg5 = &wszName[0]
0100117e xor     r9d,r9d                ; arg4 = 0
```

```
01001181 mov     r8d,0x4                ; arg3 = 4 (ASSOCSTR_FRIENDLYNAME)
01001187 xor     edx,edx                ; arg2 = 0
01001189 call    qword ptr [rax+0x20]   ; call GetString method
0100118c mov     ebx,eax                ; ebx = hr
0100118e test    ebx,ebx                ; FAILED?
01001190 jl      ReleasePQA (01001274)  ; jump if so
```

我们再次设置参数并调用函数，然后测试返回值是否成功。

dbgcmd

```
               VARIANTARG rgvarg[2] = { 0 };

01001196 lea     rdi,[rsp+0x82]         ; rdi = &rgvarg
0100119e xor     eax,eax                ; rax = 0
010011a0 mov     ecx,0x2e               ; rcx = sizeof(rgvarg)
010011a5 rep     stosb                  ; Zero it out
```

在 x64 上将缓冲区归零的惯用方法与 x86 相同。

dbgcmd

```
               V_VT(&rgvarg[0]) = VT_BSTR;
               V_BSTR(&rgvarg[0]) = SysAllocString(wszName);
               if (V_BSTR(&rgvarg[0])) {

010011a7 mov     word ptr [rsp+0x80],0x8 ; V_VT(&rgvarg[0]) = VT_BSTR
010011b1 lea     rcx,[rsp+0xb0]         ; arg1 = &wszName[0]
010011b9 call    qword ptr [_imp_SysAllocString (01001010)] ; call
010011bf mov     [rsp+0x88],rax         ; V_BSTR(&rgvarg[0]) = result
010011c7 test    rax,rax                ; anything allocated?
010011ca je      OutOfMemory (0100126f) ; jump if failed

                  DISPPARAMS dp;
                  LONG lUnique = InterlockedIncrement(&lCounter);

010011d0 lea     rax,[lCounter (01002000)]
010011d7 mov     ecx,0x1
010011dc lock    xadd [rax],ecx             ; interlocked exchange and add
010011e0 add     ecx,0x1
```

InterlockedIncrement 直接编译到计算机代码。 **lock xadd** 指令执行原子交换并添加。 最终结果存储在 **ecx** 中。

dbgcmd

```
                  V_VT(&rgvarg[1]) = VT_I4;
                  V_I4(&rgvarg[1]) = fUnique ? lUnique : 0;

010011e3 mov     word ptr [rsp+0x98],0x3    ; V_VT(&rgvarg[1]) = VT_I4;
```

```
010011ed mov     eax,r14d                    ; rax = 0 (r14d is still zero)
010011f0 test    r12d,r12d                   ; fUnique set?
010011f3 cmovne  eax,ecx                     ; if so, then set rax=lCounter
010011f6 mov     [rsp+0xa0],eax              ; V_I4(&rgvarg[1]) = ...
```

由于 x64 支持 **cmov** 指令，**因此无需使用跳转即可编译** ?: 构造。

dbgcmd

```
                   dp.rgvarg = rgvarg;
                   dp.cArgs = 2;
                   dp.rgdispidNamedArgs = NULL;
                   dp.cNamedArgs = 0;

010011fd lea     rax,[rsp+0x80]              ; rax = &rgvarg[0]
01001205 mov     [rsp+0x60],rax              ; dp.rgvarg = rgvarg
0100120a mov     [rsp+0x70],r15d             ; dp.cArgs = 2 (r15 is still 2)
0100120f mov     [rsp+0x68],r14              ; dp.rgdispidNamedArgs = NULL
01001214 mov     [rsp+0x74],r14d             ; dp.cNamedArgs = 0
```

此代码初始化 DISPPARAMS 的其余成员。 请注意，编译器重用 CLSID 之前使用的堆栈空间。

dbgcmd

```
                   hr = pdisp->Invoke(dispid, IID_NULL, 0, DISPATCH_METHOD,
&dp, NULL, NULL, NULL);

01001219 mov     rax,[rsi]                   ; rax = pdisp.vtbl
0100121c mov     [rsp+0x40],r14              ; arg9 = 0
01001221 mov     [rsp+0x38],r14              ; arg8 = 0
01001226 mov     [rsp+0x30],r14              ; arg7 = 0
0100122b lea     rcx,[rsp+0x60]              ; rcx = &dp
01001230 mov     [rsp+0x28],rcx              ; arg6 = &dp
01001235 mov     word ptr [rsp+0x20],0x1     ; arg5 = 1 (DISPATCH_METHOD)
0100123c xor     r9d,r9d                     ; arg4 = 0
0100123f lea     r8,[GUID_NULL (01001080)]   ; arg3 = &IID_NULL
01001246 mov     edx,r13d                    ; arg2 = dispid
01001249 mov     rcx,rsi                     ; arg1 = pdisp
0100124c call    qword ptr [rax+0x30]        ; call Invoke method
0100124f mov     ebx,eax                     ; hr = result
```

然后，代码设置参数并调用 **Invoke** 方法。

dbgcmd

```
                   VariantClear(&rgvarg[0]);
                   VariantClear(&rgvarg[1]);

01001251 lea     rcx,[rsp+0x80]              ; arg1 = &rgvarg[0]
```

```
01001259 call    qword ptr [_imp_VariantClear (01001018)]
0100125f lea     rcx,[rsp+0x98]              ; arg1 = &rgvarg[1]
01001267 call    qword ptr [_imp_VariantClear (01001018)]
0100126d jmp     ReleasePQA (01001274)
```

代码完成条件的当前分支，并跳过 **else** 分支。

dbgcmd

```
                } else {
                    hr = E_OUTOFMEMORY;
                }
            }

OutOfMemory:
0100126f mov     ebx,0x8007000e              ; hr = E_OUTOFMEMORY
        pqa->Release();
ReleasePQA:
01001274 mov     rcx,[rsp+0x58]              ; arg1 = pqa
01001279 mov     rax,[rcx]                   ; rax = pqa.vtbl
0100127c call    qword ptr [rax+0x10]        ; release
```

else 分支。

dbgcmd

```
    return hr;
}

0100127f mov     eax,ebx                     ; rax = hr (for return value)
ReturnEAX:
01001281 add     rsp,0x2c0                   ; clean up the stack
01001288 pop     r15d                        ; restore
0100128a pop     r14d                        ; restore
0100128c pop     r13d                        ; restore
0100128e pop     r12d                        ; restore
01001290 pop     rdi                         ; restore
01001291 pop     rsi                         ; restore
01001292 pop     rbx                         ; restore
01001293 ret                                 ; return (do not pop arguments)
```

返回值存储在 **rax** 中，然后在返回之前还原非易失性寄存器。

另请参阅

x64 体系结构

X86-64 维基百科

调试器编程扩展 API

项目 • 2023/10/26

本节包括：

调试器引擎概述

使用调试器引擎 API

编写 DbgEng 扩展

EngExtCpp 扩展

编写 WdbgExts 扩展

使用 DML 自定义调试器输出

使用 JavaScript 扩展调试器的功能

开发 KDNET 传输扩展性模块

本文档介绍如何使用调试器引擎提供的接口来编写将在 WinDbg、KD、CDB 和 NTSD 中运行的扩展。 执行用户模式或内核模式调试时，可以使用这些调试器扩展。

调试器引擎

调试器引擎提供了一个接口，用于在用户模式和内核模式下检查和操作调试目标。

调试器引擎可以获取目标、设置断点、监视事件、查询符号、读取和写入内存，以及控制目标中的线程和进程。

可以使用调试器引擎来编写调试器扩展库和独立应用程序。 此类应用程序是 *调试器引擎应用程序*。 使用调试器引擎的完整功能的调试器引擎应用程序是 *调试器*。 例如，WinDbg、CDB、NTSD 和 KD 是调试器;调试器引擎提供其功能的核心。

调试器引擎 API 由头文件 dbgeng.h 中的原型指定。

有关详细信息，请参阅 调试器引擎概述 和使用 调试器引擎 API。

扩展

可以通过编写和生成扩展 DLL 来创建自己的调试命令。 例如，你可能想要编写扩展命令来显示复杂的数据结构。

有三种不同类型的调试器扩展 DLL：

- *DbgEng 扩展 DLL*。 这些基于 dbgeng.h 头文件中的原型。 此类型的每个 DLL 都可以导出 DbgEng 扩展命令。 这些扩展命令使用调试器引擎 API，也可能使用 WdbgExts API。

  有关详细信息，请参阅 编写 DbgEng 扩展。

- *EngExtCpp 扩展 DLL*。 这些基于 engextcpp.h 和 dbgeng.h 头文件中的原型。 此类型的每个 DLL 都可以导出 DbgEng 扩展命令。 这些扩展命令同时使用调试器引擎 API 和 EngExtCpp 扩展框架，还可以使用 WdbgExts API。

- *WdbgExts 扩展 DLL*。 这些基于 wdbgexts.h 头文件中的原型。 此类型的每个 DLL 导出一个或多个 WdbgExts 扩展命令。 这些扩展命令以独占方式使用 WdbgExts API。 有关详细信息，请参阅 编写 WdbgExts 扩展。

DbgEng API 可用于创建扩展或独立应用程序。 WdbgExts API 包含调试器引擎 API 的一部分功能，只能由扩展使用。

应使用 Visual Studio 编译和生成所有调试器扩展。

如果你执行自定义安装并选择 **SDK** 组件及其所有子组件，则扩展代码示例将作为 Windows 调试工具包的一部分安装。 可以在 Windows 安装目录的调试工具的 sdk\samples 子目录中找到它们。

编写新调试器扩展的最简单方法是研究示例扩展。 每个示例扩展都包含生成文件和源文件，以便与生成实用工具一起使用。 这两种类型的扩展都表示在示例中。

编写自定义分析调试器扩展

可以通过编写分析扩展插件来扩展 **！analyze** 调试器命令的功能。 通过提供分析扩展插件，你可以以特定于你自己的组件或应用程序的方式参与对 bug 检查或异常的分析。 编写分析扩展插件时，还会编写一个元数据文件，用于描述要调用插件的情况。 **运行！analyze** 时，它会查找、加载并运行相应的分析扩展插件。 有关详细信息，请参阅 编写自定义分析调试器扩展

使用 DML 自定义调试器输出

可以使用 DML 自定义调试器输出。 有关详细信息，请参阅 使用 DML 自定义调试器输出。

使用 JavaScript 扩展调试器的功能

使用 JavaScript 创建了解调试器对象的脚本，并扩展和自定义调试器的功能。 JavaScript 提供程序将脚本语言桥接到调试器的内部对象模型。 JavaScript 调试器脚本提供程序允许将 JavaScript 与调试器配合使用。 有关详细信息，请参阅 JavaScript 调试器脚本。

开发 KDNET 传输扩展性模块

通过使用单独的硬件驱动程序扩展性模块 dll，KDNET 传输可以扩展为在任何硬件上运行。 KDNET 传输扩展性模块由网络卡供应商开发，以向特定网卡添加内核调试支持。

KDNET 是一种内核调试传输，支持通过网络对 Windows 进行内核调试。 它旨在将硬件支持层内置到独立于网络数据包处理和内核接口层的模块中。 此硬件驱动程序支持层称为 KDNET 扩展性模块。 有关详细信息，请参阅 开发 KDNET 传输扩展性模块。

调试器引擎简介

项目 • 2023/06/15

本文档介绍如何使用调试器引擎以及如何编写将在 WinDbg、KD、CDB 和 NTSD 中运行的扩展。 在 Microsoft Windows 上执行用户模式或内核模式调试时，可以使用这些调试器扩展。

调试器引擎

调试器引擎提供了一个接口，用于在 Microsoft Windows 上的用户模式和内核模式下检查和操作调试目标。

调试器引擎可以获取目标、设置断点、监视事件、查询符号、读取和写入内存，以及控制目标中的线程和进程。

可以使用调试器引擎编写调试器扩展库和独立应用程序。 此类应用程序是 *调试器引擎应用程序*。 使用调试器引擎的完整功能的调试器引擎应用程序是 *调试器*。 例如，WinDbg、CDB、NTSD 和 KD 是调试器;调试器引擎提供其功能的核心。

调试器引擎 API 由头文件 dbgeng.h 中的原型指定。

不完整的文档

这是初步文档，目前不完整。

有关此处尚未记录的调试器和调试器引擎的许多概念，请查看本文档的 调试技术 部分。

若要获取调试器引擎 API 的某些当前未记录的功能，请使用 **Execute** 方法执行各个调试器命令。

扩展

可以通过编写和生成扩展 DLL 来创建自己的调试命令。 例如，你可能想要编写扩展命令来显示复杂的数据结构。

有三种不同类型的调试器扩展 DLL：

- *DbgEng 扩展 DLL*。 它们基于 dbgeng.h 头文件中的原型。 此类型的每个 DLL 都可以导出 DbgEng 扩展命令。 这些扩展命令使用调试器引擎 API，也可能使用 WdbgExts API。

- *EngExtCpp 扩展 DLL*。 它们基于 engextcpp.h 和 dbgeng.h 头文件中的原型。 此类型的每个 DLL 都可以导出 DbgEng 扩展命令。 这些扩展命令同时使用调试器引擎 API 和 EngExtCpp 扩展框架，还可以使用 WdbgExts API。

- *WdbgExts 扩展 DLL*。 它们基于 wdbgexts.h 头文件中的原型。 此类型的每个 DLL 导出一个或多个 WdbgExts 扩展命令。 这些扩展命令以独占方式使用 WdbgExts API。

DbgEng API 可用于创建扩展或独立应用程序。 WdbgExts API 包含调试器引擎 API 的一部分功能，只能由扩展使用。

应使用生成实用工具编译和生成所有调试器扩展。 生成实用工具包含在 Windows 驱动程序工具包 (WDK) 中。

如果执行自定义安装并选择 **SDK** 组件及其所有子组件，扩展代码示例将作为 Windows 调试工具包的一部分进行安装。 可以在 Windows 安装目录的调试工具的 sdk\samples 子目录中找到它们。

编写新的调试器扩展的最简单方法是研究示例扩展。 每个示例扩展都包含用于生成实用工具的生成文件和源文件。 这两种类型的扩展都在示例中表示。

编写自定义分析调试器扩展

可以通过编写分析扩展插件来扩展 **！analyze** 调试器命令的功能。 通过提供分析扩展插件，你可以以特定于你自己的组件或应用程序的方式参与对 bug 检查或异常的分析。 编写分析扩展插件时，还会编写一个元数据文件，用于描述要调用插件的情况。 当 **！analyze** 运行时，它会查找、加载和运行相应的分析扩展插件。 有关详细信息，请参阅 编写自定义分析调试器扩展

使用 DML 自定义调试器输出

可以使用 DML 自定义调试器输出。 有关详细信息，请参阅 使用 DML 自定义调试器输出。

使用 JavaScript 扩展调试器的功能

使用 JavaScript 创建了解调试器对象以及扩展和自定义调试器功能的脚本。 JavaScript 提供程序将脚本语言桥接到调试器的内部对象模型。 JavaScript 调试器脚本提供程序允许将 JavaScript 与调试器一起使用。 有关详细信息，请参阅 JavaScript 调试器脚本。

调试器引擎概述

项目 • 2023/06/15

调试器引擎 (DbgEng.dll) （通常称为*引擎*）提供一个接口，用于在 Microsoft Windows 上的*用户模式*和*内核模式*下检查和操作调试目标。

调试器引擎可以获取目标、设置 断点、监视 事件、查询 符号、读取和写入内存，以及控制目标中的 线程 和 进程 。

可以使用调试器引擎编写调试器扩展库和独立应用程序。 此类应用程序称为 *调试器引擎应用程序*。 使用调试器引擎的完整功能的调试器引擎应用程序称为 *调试器*。 例如，WinDbg、CDB、NTSD 和 KD 是调试器;调试器引擎提供其功能的核心。

引擎概念：

调试会话和执行模型

客户端对象

输入和输出

检查和操作目标：

目标

事件

“断点”

符号

内存

线程和进程

不完整的文档

这是初步文档，目前不完整。

有关此处尚未记录的调试器和调试器引擎的许多概念，请查看本文档的 调试技术 部分。

若要获取调试器引擎 API 的某些当前未记录的功能，请使用 **Execute** 方法执行各个调试器命令。

调试会话和执行模型

项目 • 2023/06/15

调试器引擎可以同时调试多个目标。 *调试会话*在引擎获取目标时开始，并一直持续到所有目标被丢弃为止。 当目标正在执行时，调试会话 *不可访问*;当当前目标挂起时 *可访问* 调试会话。 该引擎只能用于在访问会话时检查和操作目标。

调试器的main循环通常包括设置执行状态、调用 WaitForEvent 方法以及处理生成的事件。 调用 WaitForEvent 时，会话变得不可访问。

当目标中发生事件时，引擎将挂起所有目标，并且会话变为可访问。 然后，引擎通知事件的事件回调，并遵循事件筛选器规则。 事件回调和事件筛选器确定目标中的执行方式。 如果他们确定引擎应中断调试器， **则** WaitForEvent 方法将返回，并且会话仍可访问;否则，引擎将以事件回调和事件筛选器确定的方式继续执行目标，会话将再次变得不可访问。

在 WaitForEvent 调用期间，特别是在通知事件回调和处理筛选规则时，引擎处于称为"等待内"的状态。 处于此状态时，无法调用 WaitForEvent ，（它不是可重入）。

在目标中启动执行涉及两个步骤：设置执行状态，然后调用 WaitForEvent。 可以使用 SetExecutionStatus 方法或通过执行设置执行状态的调试器命令（例如， g (Go) 和 p (Step) ）**来设置执行状态** 。

如果一系列调试器命令一起执行，例如，"g; ? @$ip"--如果命令不是序列中的最后一个命令，则要求在目标中执行的任何命令之后，将发生 *隐式等待* 。 当调试器引擎处于"等待中"状态时，无法发生隐式等待;在这种情况下，命令的执行将停止，当前命令（试图导致隐式等待的命令）将被解释为指示目标中的执行方式。 其余命令将被丢弃。

注意 确定会话是可访问还是不可访问时，目标 (有限执行时，引擎会将单步执行) 视为执行。 受限执行完成后，会话将变为可访问。

主机引擎

远程调试时，可以使用调试器引擎的多个实例。 正是其中一个实例维护调试会话;此实例称为 *主机引擎*。

所有调试器操作都相对于主机引擎，例如符号加载和扩展加载。

客户端对象

项目 • 2024/05/27

几乎所有与调试器引擎的交互都是通过*客户端对象*（通常简称为*客户端*）进行的。 每个客户端都提供一个顶级引擎接口的实现。 每个接口都会提供一组不同的方法，用于与引擎以及通过引擎与目标进行交互。 一个引擎实例可以有很多客户端，而每个客户端都有自己的状态。

主客户端

*主客户端*是指已加入当前调试会话的客户端。 最初，在创建新客户端对象时，它并不是主客户端。 当客户端用于获取目标（例如，通过调用 CreateProcess2）或使用 ConnectSession 连接到调试会话时，它就会成为主客户端。 调试器命令 .clients 只会列出主客户端。

回调对象

每个客户端都可以注册回调对象。 回调对象有三种类型：

1. **输入回调对象**（或*输入回调*）：引擎调用输入回调来请求输入。 例如，带有控制台窗口的调试器可以注册一个输入回调，向引擎提供来自用户的输入；或者调试器也可以注册一个输入回调，向引擎提供来自文件的输入。

2. **输出回调对象**（或*输出回调*）：引擎调用输出回调来显示输出。 例如，带有控制台窗口的调试器可以注册一个输出回调，将调试器的输出显示给用户；或者调试器也可以注册一个输出回调，将输出发送到日志文件。

3. **事件回调对象**（或*事件回调*）：每当目标中发生事件（或引擎状态发生变化）时，引擎都会调用事件回调。 例如，调试器扩展库可以注册一个事件回调来监控某些事件，或在特定事件发生时执行自动操作。

远程调试

客户端对象有助于与主机引擎的远程实例进行通信。 DebugConnect 函数创建了一个连接到远程引擎实例的客户端对象；远程引擎将执行在该客户端上调用的方法，并且当远程引擎进行回调调用时，将调用在客户端本地注册的回调对象。

其他信息

有关创建和使用客户端对象的详细信息，请参阅使用回调对象。 有关注册回调对象的详细信息，请参阅“使用回调对象”。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

输入和输出

项目 • 2023/06/15

调试器引擎的输入和输出设施可用于交互式调试器操作和日志记录。 输入通常表示用户键入的命令和响应，输出通常表示向用户显示或发送到日志文件的信息。

调试器引擎维护 *输入流* 和 *输出流*。 可以从输入流请求输入，并将输出发送到输出流。

当调用 **Input** 方法以从引擎的输入流请求输入时，引擎将调用所有已注册的 输入回调，以通知它们它正在等待输入。 然后，它通过调用 **ReturnInput** 方法等待输入回调提供输入。

将输出发送到引擎的输出流时，引擎将调用注册的 输出回调， 并将输出传递给它们。 将输出发送到输出流时，可以通过客户端对象对其进行筛选;在这种情况下，只有向特定客户端对象注册的输出回调才会接收输出。

输入和输出流以透明方式提供给远程客户端。 远程客户端可以请求输入并将输出发送到引擎的输入和输出流，引擎将调用注册到远程客户端的回调来请求输入或发送输出。

其他信息

有关使用输入和输出的详细信息，请参阅 使用输入和输出。 有关客户端对象以及输入和输出回调的详细信息，请参阅 客户端对象。

远程调试 (调试器引擎)

项目 • 2023/06/15

当客户端与目标通信是间接的（例如，通过网络连接）时，将发生远程调试。 远程调试时，调试器引擎的多个实例可以参与调试目标。 但是，正是其中一个实例负责调试会话;此实例称为 *主机引擎*。

有许多可能的配置：可以在主机引擎中创建客户端对象 (智能客户端) ，或 (调试客户端) 的其他引擎实例;主机引擎可以直接连接到目标 (调试服务器) ;或代理可以直接连接到目标 (进程服务器和内核连接服务器) 。

多个客户端可以同时连接到主机引擎。 主机引擎可以连接到同一调试会话中的多个目标。 （可选）客户端和主机引擎之间以及主机引擎与每个目标之间可以有一个或多个代理。

智能客户端是直接与主机引擎通信的客户端对象。 调试客户端是通过调用 DebugConnect 创建的;客户端使用 RPC 调用与主机引擎通信，这些调用表示引擎 API 中的方法调用 (包括主机引擎对客户端 回调对象) 的调用。

调试服务器是直接与目标通信的引擎实例，也是主机引擎。 进程服务器和内核连接服务器直接与目标通信，但不是主机引擎。 主机引擎通过发送低级别内存、处理器和操作系统请求来与进程服务器或内核连接服务器通信，服务器发送回结果。

注意 用于内核调试的典型双计算机设置（其中一台计算机是目标计算机，另一台是主计算机）不被视为远程调试，因为主计算机上只有一个引擎 (实例) 并且直接与目标通信。

其他信息

有关执行远程调试的详细信息，请参阅 远程目标。

目标

项目 • 2023/06/15

调试器引擎支持调试不同类型的目标、用户模式和内核模式目标、实时目标和故障转储文件以及本地和远程目标。 可通过不同的方法将引擎连接到这些不同类型的目标。

故障转储文件

用户模式和内核模式故障转储文件都使用 OpenDumpFile 打开。 该引擎还可以使用 WriteDumpFile2 从目标创建转储文件。

实时、User-Mode目标

调试器引擎可以创建和附加到用户模式进程。

创建进程是通过为新进程提供命令行（可选）初始目录和环境来完成的。 然后，引擎可以连接到新进程，或者在它连接到另一个进程时使新进程保持挂起状态。 例如，调试由客户端和服务器组成的应用程序时，可以创建处于挂起状态的客户端并附加到已运行的服务器，从而允许在客户端运行和引发服务器操作之前设置服务器断点。

从进程分离时，引擎可以选择让进程正常运行、终止进程或放弃进程，（暂停进程，直到另一个调试器连接到进程或）终止。

可以查询引擎，以获取有关计算机上运行的所有用户模式进程的信息，包括用于启动进程的进程 ID 和可执行映像的名称。 此信息可用于帮助查找要调试的进程。

实时、Kernel-Mode目标

方法 AttachKernel 将调试器引擎连接到 Windows 内核。

远程目标

使用调试器引擎进行远程调试时，可能还有两个额外的步骤：

1. 连接到主机引擎。 如果主机引擎不是本地引擎实例，请使用 DebugConnect 创建连接到主机引擎的客户端对象。

2. 将主机引擎连接到进程服务器或内核连接服务器。 如果主机引擎未直接连接到目标，则必须连接到进程服务器或内核连接服务器。

现在，客户端可以告知主机引擎通过进程服务器或内核连接服务器获取目标。

获取目标

获取目标时，在目标生成事件之前，目标获取不会完成。 通常，这意味着首先调用方法将调试器附加到目标，然后调用 **WaitForEvent** 让目标生成事件。 当目标为故障转储文件时，这仍然有效，因为这些文件始终存储一个事件（通常是导致创建转储文件的事件）。

其他信息

有关附加到目标的详细信息，请参阅 连接到目标。

事件

项目 • 2023/06/14

调试器引擎提供用于监视和响应目标中的事件的功能。 发生事件时，引擎通常会暂停目标 (通常只是短暂)，然后通知事件的所有客户端，这些客户端反过来又指示引擎如何在目标中继续执行。

为了通知客户端某个事件，引擎会调用注册到客户端的事件回调对象。 引擎为每个事件回调提供事件的详细信息，事件回调指示引擎应如何在目标中继续执行。 当不同的事件回调提供冲突的指令时，引擎对优先级最高的指令执行操作 (查看 DEBUG_STATUS_XXX)，这通常意味着选择涉及目标执行最少的指令。

注意 当事件回调处理事件时，目标将被挂起，调试会话可访问;但是，由于引擎在 WaitForEvent 调用期间显式等待事件，或者通过执行 g (Go) 或 p (Step) 等命令隐式等待事件，因此事件回调无法调用 WaitForEvent，并且如果它尝试执行任何会导致调试器执行的命令，例如 g (Go) 或 p (Step)， 引擎会将这些命令解释为有关如何继续的说明。

事件筛选器

调试器引擎还提供 *事件筛选器*，这是基本事件监视的更简单的替代方法。 事件筛选器提供了一些简单的规则，这些规则指定是应将事件输出到调试器的输出流中还是应中断到调试器中。 它们还可用于在事件发生时执行调试器命令。

其他信息

有关监视事件的详细信息，请参阅 监视事件。

断点

项目 • 2024/03/01

调试器引擎可以在目标中创建和监视断点。

引擎可以在目标中插入到两种类型的断点：软件断点和处理器断点。

- 通过修改断点位置的处理器指令，将*软件断点*插入目标代码中。 调试器引擎跟踪此类断点；它们对在该位置读取和写入的内存的客户端不可见。 当目标执行修改后的指令时，将触发软件断点。

- *处理器断点*由调试器引擎插入目标处理器中。 处理器断点可以通过不同的操作触发，例如，在位置（如软件断点）执行指令，或者在断点位置读取或写入内存。 对处理器断点的支持依赖于目标计算机中的处理器。

断点的地址可由显式地址、计算结果为地址的表达式或将来可能计算为地址的表达式来指定。 在最后一种情况下，每次在目标中加载或卸载模块时，引擎将尝试重新评估表达式并插入断点（如果它可以确定地址）；这样就可以在加载模块之前在模块中设置断点。

许多参数可以与断点相关联，以控制其行为：

- 断点可以与目标中的特定线程相关联，并且只能由该线程触发。

- 断点可以具有与之关联的调试器命令；触发断点时，这些命令将自动执行。

- 在目标传递指定次数之前，可以将断点标记为非活动状态。

- 在首次触发断点时，断点可以被自动删除。

其他信息

有关使用断点的详细信息，请参阅使用断点。

符号

项目 • 2023/06/15

*符号*是模块中显示的源文件中的数据或代码的命名单位。 有关符号的信息可以包括名称、类型（（如果适用））、存储该符号的地址或寄存器，以及任何父符号或子符号。 符号的示例包括变量 (局部和全局) 、函数以及模块中的任何入口点。

引擎使用符号信息来帮助解释目标中的数据和代码。 借助此信息，引擎可以按内存中的名称或位置搜索符号，并提供符号的说明。

引擎从位于本地文件系统或从符号服务器加载的符号文件获取有关符号的信息。 使用符号服务器时，引擎将自动使用正确版本的符号文件来匹配目标中的模块。 每当加载相应的模块时，都可以加载符号文件，也可以根据需要加载它们。

注意 通常，优化编译器不包含符号文件中的准确信息。 这可能导致引擎错误地解释某些变量的值，因为变量的位置或生存期描述可能不正确，导致引擎查看错误的内存片段，或认为变量值在 (死时处于活动状态，反之亦然) 。 优化编译器还可以更改执行顺序或将函数拆分为多个部分。 调试未优化的代码时，通常会获得最佳结果。

其他信息

有关使用符号的详细信息，请参阅 使用符号。 有关使用符号文件和符号服务器的概述，请参阅本文档的调试器部分中的 符号 。

内存

项目 • 2023/06/15

调试器引擎可以直接读取和写入目标main内存、寄存器和其他数据空间。 在内核模式调试中，所有目标内存都可用，包括虚拟内存、物理内存、寄存器、模型特定寄存器 (MSR)、系统总线内存、Control-Space内存和 I/O 内存。 在用户模式调试中，只有虚拟内存和寄存器可用。

引擎使用 64 位地址向客户端公开目标中的所有内存。 如果目标使用 32 位地址，则当与目标和客户端通信时，引擎将根据需要自动转换 32 位和 64 位地址。 如果从目标中恢复 32 位地址（例如，通过从内存或寄存器中读取），则必须将其签名扩展为 64 位，然后才能在调试器引擎 API 中使用。 签名扩展由 **ReadPointersVirtual** 方法自动执行。

其他信息

有关读取和写入内存的详细信息，请参阅 内存访问。

线程和进程

项目 • 2023/06/15

术语

用户模式调试和内核模式调试之间的线程和进程概念不同。

- 在 *用户模式调试*中， *进程* 是操作系统进程， *线程* 是操作系统线程。
- 在 *内核模式调试*中， 调试器引擎 为每个目标创建 *一个虚拟进程* ;此进程表示内核，不对应于任何操作系统进程。 对于目标计算机中的每个物理处理器，调试器将创建一个 *虚拟线程*;这些线程表示处理器，不对应于任何操作系统线程。

发生事件时，引擎将 *事件进程* 和 *事件线程* 设置为 (发生事件的操作系统或虚拟) 的进程和线程。

*当前线程*是引擎当前控制的操作系统或虚拟) (线程。 *当前进程*是引擎当前控制 (操作系统或虚拟) 的进程。 发生事件时，当前线程和进程最初设置为事件线程和进程;但是，可以在访问会话时使用客户端更改它们。

在内核模式下，调试器跟踪隐式进程和隐式线程。 *隐式进程*是操作系统进程，用于确定从虚拟到物理内存地址的转换。

*隐式线程*是操作系统线程，用于确定目标寄存器，包括调用堆栈、堆栈帧和指令偏移量。

发生事件时，隐式线程和隐式进程最初设置为事件线程和进程;可以在访问会话时更改它们。

线程和处理数据

引擎维护有关每个线程和进程的多个信息。 这包括系统线程、进程 ID 和系统句柄，以及 PEB) (进程环境、线程环境块 (TEB) ，以及它们在目标内存中的位置。

其他信息

有关使用线程和进程的详细信息，请参阅 控制线程和进程。

使用调试器引擎 API

项目 • 2024/03/20

本节包括：

调试器引擎 API 概述

调试器引擎参考

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

调试器引擎 API 概述

项目 • 2023/06/15

本节包括：

与引擎交互

使用输入和输出

监视事件

使用断点

内存访问

检查堆栈跟踪

控制线程和进程

使用符号

使用源文件

连接到目标

目标信息

目标状态

调用扩展和扩展函数

汇编和反汇编指令

重要 IDebug* 接口（如 **IDebugEventCallbacks** 接口）虽然 COM 类似，但不是正确的 COM API。 不支持从托管代码调用这些接口。 使用托管代码调用接口时，垃圾回收和线程所有权等问题会导致系统不稳定。

与引擎交互

项目 • 2023/10/26

命令和表达式

调试器引擎 API 提供执行命令和计算表达式的方法，如在 WinDbg 的 调试器命令窗口中键入的方法。 若要执行调试器命令，请使用 Execute。 或者，若要执行文件中的所有命令，请使用 ExecuteCommandFile。

方法 Evaluate 将使用 C++ 或 MASM 语法计算表达式。 调试器引擎用于计算表达式的语法（例如在 Evaluate 方法中）由 GetExpressionSyntax 提供，可以使用 SetExpressionSyntaxByName 和 SetExpressionSyntax 进行更改。 调试器识别的不同语法的数目由 GetNumberExpressionSyntaxes 返回，其名称由 GetExpressionSyntaxNames 返回。

Evaluate 返回的值的类型由计算的字符串中使用的符号和常量决定。 该值包含在 DEBUG_VALUE 结构中，可以使用 CoerceValue 和 CoerceValues 强制转换为不同的类型。

别名

别名 是在调试器命令和表达式中使用时自动替换为其他字符串的字符串。 有关别名的概述，请参阅 使用别名。 调试器引擎具有多个别名类。

*固定名称别名*按数字编制索引，其名称$u 0、$u 1、...$u 9。 可以使用 SetTextMacro 方法设置这些别名的值，并使用 GetTextMacro 方法检索这些别名的值。

*自动别名*和*用户命名别名*可以具有任何名称。 自动别名由调试器引擎定义，用户命名别名由用户通过调试器命令或调试器引擎 API 定义。 若要定义或删除用户命名别名，请使用 SetTextReplacement 方法。 GetTextReplacement 方法返回自动别名或用户命名别名的名称和值。 可以使用 RemoveTextReplacements 方法删除所有用户命名的别名。 GetNumberTextReplacements 方法将返回用户名和自动别名的数目;这可以与 GetTextReplacement 一起使用，以循环访问所有这些别名。 OutputTextReplacements 方法将打印所有用户命名别名的列表，包括其名称和值。

请注意，如果用户命名的别名与自动别名同名，则用户命名别名将隐藏自动别名，以便在按名称检索别名的值时，将使用用户命名的别名。

>引擎选项

引擎具有许多用于控制其行为的选项。 DEBUG_ENGOPT_XXX中列出了这些选项。 它们由 GetEngineOptions 返回，可以使用 SetEngineOptions 进行设置。 可以使用 AddEngineOptions 设置单个选项，使用 RemoveEngineOptions 取消设置。

中断

中断是强制中断调试器或告知引擎停止处理当前命令的一种方法，例如，在 WinDbg 中按 Ctrl+Break。

若要请求中断调试器或中断调试器的当前任务，请使用 SetInterrupt。 若要检查是否存在中断，请使用 GetInterrupt。

注意从调试器扩展执行长任务时，建议扩展定期检查 **GetInterrupt**，并在请求中断时停止处理。

请求中断调试器时，如果目标执行中断所需的时间过长，引擎可能会超时。 如果目标处于无响应状态，或者资源争用阻止或延迟了闯入请求，则可能会发生这种情况。 引擎等待的时间长度由 GetInterruptTimeout 返回，可以使用 SetInterruptTimeout 设置。

使用客户端对象

项目 • 2023/06/15

有关客户端对象在与调试器引擎交互中的角色的概述，请参阅 客户端对象。

通常，只能从创建客户端的线程调用客户端的方法。 通常，从错误线程调用的方法将立即失败。 此规则的显著例外是 **方法** CreateClient;此方法可以从任何线程调用，并返回可在调用该方法的线程中使用的新客户端。 参考部分记录了其他异常。

描述客户端对象的字符串由 GetIdentity 方法返回，或者可以使用 OutputIdentity 写入引擎的输出流。

COM 接口

调试器引擎 API 包含多个 COM 类似接口;它们实现 **IUnknown** 接口。

调试引擎接口部分中介绍的接口由客户端 (实现，但不一定是在最新版本) 实现的。 可以使用 COM 方法 **IUnknown：： QueryInterface** 从任何其他接口获取其中每个接口。

客户端实现 **IUnknown** COM 接口，并使用它来维护引用计数和接口选择。 但是，客户端不是注册的 COM 对象。 方法 **IUnknown：： AddRef** 用于递增对象的引用计数，方法 **IUnknown：： Release** 用于递减引用计数。 调用 **IUnknown：： QueryInterface** 时，引用计数递增，因此当不再需要客户端接口指针时，应调用 **IUnknown：： Release** 来递减引用计数。

使用 DebugCreate 或 DebugConnect 创建客户端对象时，引用计数将初始化为 1。

有关何时应递增和递减引用计数的详细信息，请参阅平台 SDK。

IUnknown：： QueryInterface、 **DebugCreate** 和 **DebugConnect** 各自采用接口 ID 作为其参数之一。 可以使用 **__uuidof** 运算符获取此接口 ID。 例如：

```
C++

IDebugClient * debugClient;
HRESULT Hr = DebugCreate( __uuidof(IDebugClient), (void **)&debugClient );
```

重要IDebugEventCallbacks **接口等** IDebug* 接口（尽管 COM 类似）不是适当的 COM API。 从托管代码调用这些接口是一种不支持的方案。 使用托管代码调用接口时，垃圾回收和线程所有权等问题会导致系统不稳定。

使用回调对象

项目 • 2024/05/27

引擎使用三个类似 COM 的回调接口：IDebugEventCallbacks 用于通知调试器扩展和应用程序引擎或目标的更改，IDebugInputCallbacks 用于请求输入，而 IDebugOutputCallbacks 用于发送输出。

回调对象在客户端注册。 每个客户端最多只能注册三个回调接口中的一个（Unicode 和 ASCII 版本的接口算作同一个接口）。

在创建客户端时，引擎会记住创建客户端的线程。 每当引擎调用客户端注册的回调实例时，就都会使用同一个线程。 如果线程正在使用中，则引擎会将需要进行的调用排入队列。 为了允许引擎进行这些调用，只要客户端的线程处于空闲状态，就应调用 *DispatchCallbacks* 方法。 **ExitDispatch** 方法将导致返回 *DispatchCallbacks*。 如果线程与用于启动调试器会话的线程相同，那么引擎可以在 **WaitForEvent** 方法期间进行回调调用，而无需调用 *DispatchCallbacks* 方法。

FlushCallbacks 方法会告诉引擎将所有缓冲输出发送到输出回调。

事件回调对象

引擎使用 IDebugEventCallbacks 接口向调试器扩展和应用程序通知事件以及引擎和目标的更改。 可以使用 *SetEventCallbacks* 向客户端注册 **IDebugEventCallbacks** 的实现。 可以使用 *GetEventCallbacks* 找到客户端注册的当前实现。 可以使用 *GetNumberEventCallbacks* 找到在所有客户端中注册的事件回调数。

有关引擎如何管理事件的详细信息，请参阅监控事件。

输入回调对象

IDebugInputCallbacks 接口供引擎用于从调试器扩展和应用程序请求输入。 可以使用 *SetInputCallbacks* 向客户端注册 **IDebugInputCallbacks** 的实现。 可以使用 *GetInputCallbacks* 找到客户端注册的当前实现。 可以使用 *GetNumberInputCallbacks* 找到在所有客户端中注册的输入回调数。

有关引擎如何管理输入的详细信息，请参阅输入和输出。

输出回调对象

IDebugOutputCallbacks 接口可供引擎用于向调试器扩展和应用程序发送输出。 可以使用 *SetOutputCallbacks* 向客户端注册 **IDebugOutputCallbacks** 的实现。 可以使用

GetOutputCallbacks 找到客户端注册的当前实现。 可以使用 *GetNumberOutputCallbacks* 找到在所有客户端中注册的输出回调数。

有关引擎如何管理输出的详细信息，请参阅输入和输出。

注意 对于 COM 对象来说，引擎会在回调 COM 对象与客户端注册时调用 **IUnknown::AddRef**，并在对象被替换或客户端被删除时调用 **IUnknown::Release**。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用输入和输出

项目 • 2023/06/15

有关调试器引擎中的输入和输出流的概述，请参阅 输入和输出。

输入

如果在客户端上调用 Input 方法，引擎将从其所有客户端请求 **输入** 。 输入将返回到 Input 的调用方。

输入回调

当引擎请求来自客户端的输入时，它将使用该客户端注册的 IDebugInputCallbacks 对象。 **可以使用** SetInputCallbacks 向客户端注册 IDebugInputCallbacks 对象。 每个客户端最多可以有一个注册到它的 **IDebugInputCallbacks** 对象。

输入请求从调用 **IDebugInputCallbacks：： StartInput** 方法的引擎开始。 这会通知 **IDebugInputCallbacks** 对象引擎正在等待输入。

如果 **IDebugInputCallbacks** 对象具有引擎的一些输入，则它可以调用任何客户端的 **ReturnInput** 方法。 调用 **ReturnInput** 方法后，引擎将不再接受任何输入。 此方法的后续调用方将收到未收到输入的通知。

然后，引擎将调用 **IDebugInputCallbacks：： EndInput** 以指示它已停止等待输入。

最后，引擎将此输入回显到每个客户端 (的已注册 **IDebugOutputCallbacks** 对象，但用于提供输入) 的对象除外，该对象使用 **IDebugOutputCallbacks：： Output** 并将位掩码设置为DEBUG_OUTPUT_PROMPT。

输出

可以使用多种客户端方法（例如 *Output* 和 *OutputVaList*）将输出发送到引擎。 收到输出后，引擎会将其发送到某些客户端。

客户端使用 *输出掩码* 来指示它们感兴趣的输出类型。 每当引擎生成输出时，它都附带一个指定其输出类型的掩码。 如果输出类型与客户端的输出掩码匹配，则客户端将收到输出。 可以通过调用 **SetOutputMask** 来设置输出掩码，并使用 **GetOutputMask** 进行查询。 有关输出掩码值的详细信息，请参阅 **DEBUG_OUTPUT_XXX** 。

引擎将向其发送输出的客户端列表由 *输出控件控制*。 通常，输出控件设置为将输出发送到所有客户端;但是，可以使用 *ControlledOutput 和 ControlledOutputVaList* 暂时更改它。

有关输出控件值的详细信息，**请参阅 DEBUG_OUTCTL_XXX** 。

输出可由引擎缓冲。 如果将多个输出片段传递给引擎，它可能会收集这些输出，并将其发送到一个大块中的客户端。 若要刷新此缓冲区，请使用 **FlushCallbacks**。

每个客户端对象都有一个 *输出宽度*，即客户端对象的输出显示的宽度。 虽然此宽度仅用作提示，但某些命令和扩展函数会基于此宽度格式化其输出。 宽度由 GetOutputWidth 方法返回，可以使用 SetOutputWidth 方法设置。

输出回调

当引擎将输出发送到客户端时，它使用向客户端注册的 IDebugOutputCallbacks 对象。 **可以使用 SetOutputCallbacks** 向客户端注册 IDebugOutputCallbacks 对象。 每个客户端最多可以有一个注册到它的 **IDebugInputCallbacks** 对象。

为了发送输出，引擎调用 **IDebugOutputCallbacks：：Output** 方法。

输出行前缀

每个客户端对象都有一个 *输出行前缀* ，该前缀位于发送到与客户端对象关联的输出回调的每一行输出前面。 这可用于缩进或在每个输出行上放置标识标记。

输出行前缀由 GetOutputLinePrefix 返回，可以使用 SetOutputLinePrefix 进行设置。 若要暂时更改输出行前缀，稍后再次更改它，请使用 PushOutputLinePrefix 和 PopOutputLinePrefix。

日志文件

调试器引擎支持打开日志文件来记录调试会话。 一次最多可以打开一个日志文件。 发送到输出回调的输出也会发送到此日志文件 (，除非标记为未记录) 。

若要打开日志文件，请使用 **OpenLogFile2** (或 **OpenLogFile**) 。 **GetLogFile2** (或 **GetLogFile**) 方法返回当前打开的日志文件。 若要关闭日志文件，请使用 **CloseLogFile**。

SetLogMask 方法可用于筛选发送到日志文件的输出，**GetLogMask** 将返回当前日志文件筛选器。

提示

在交互式调试会话中，可以使用提示向用户指示调试器正在等待用户输入。 使用 *OutputPrompt 和OutputPromptVaList* 方法将提示发送到输出回调。 标准提示的内容由 **GetPromptText** 返回。

监视事件

项目 • 2023/06/15

有关 调试器引擎中事件的概述，请参阅 事件。

可以使用 IDebugEventCallbacks 接口监视目标或调试器引擎中发生的事件。 可以使用 **SetEventCallbacks** 向客户端注册 IDebugEventCallbacks 对象。 每个客户端最多只能注册一个 **IDebugEventCallbacks** 对象。

向客户端注册 **IDebugEventCallbacks** 对象时，引擎将调用对象的 **IDebugEventCallbacks：：GetInterestMask** 以确定对象感兴趣的事件。 只有对象感兴趣的事件才会发送给它。

对于每种类型的事件，引擎在 IDebugEventCallbacks 上调用相应的回调方法。 对于来自目标的事件，这些调用返回 **的DEBUG_STATUS_XXX** 值指定目标执行应如何进行。 引擎从它调用的每个 **IDebugEventCallbacks** 对象收集这些返回值，并针对优先级最高的对象执行操作。

默认情况下，来自目标的事件会中断到调试器中

默认情况下，以下事件会中断到调试器中：

- 断点事件
- 异常事件 (未在此处记录)
- 系统错误

来自目标的事件，默认情况下不会闯入调试器

默认情况下，以下事件不会进入调试器：

- 创建进程事件
- 退出进程事件
- 创建线程事件
- 退出线程事件
- 加载模块事件
- 卸载模块事件

内部引擎更改

以下不是实际事件，而只是内部引擎更改：

- 目标更改
- 引擎更改
- 引擎符号更改
- 会话状态更改

事件筛选器

项目 • 2023/06/15

事件筛选器 提供简单的事件筛选;它们会影响目标中发生事件后调试器引擎的继续方式。 事件发生时，引擎将确定该事件是否与事件筛选器匹配。 如果存在，事件筛选器的中断状态会影响调试器是否会中断目标。 如果事件是异常事件，则处理状态确定该异常在目标中是否应被视为已处理。

注意 如果需要更复杂的事件筛选，则可以使用事件回调。

事件筛选器分为三个类别。

1. *特定事件筛选器*。 这些是所有非异常事件的筛选器。 有关这些事件的列表，请参阅 DEBUG_FILTER_XXX 。
2. *特定异常筛选器*。 第一个特定的异常筛选器是 *默认异常筛选器*。 其余部分是针对引擎具有内置筛选器的异常的筛选器。 有关 **特定异常** 筛选器的列表，请参阅特定异常。
3. *任意异常筛选器*。 这些是手动添加的异常事件的筛选器。

类别 1 和类别 2 中的筛选器统称为 *特定筛选器*，类别 2 和类别 3 中的筛选器统称为 *异常筛选器*。 GetNumberEventFilters 返回每个类别中的筛选器数。

如果事件的类型与筛选器的类型相同，则事件与特定事件筛选器匹配。 某些事件筛选器有一个附加参数，用于进一步限制它们匹配的事件。

如果异常事件的异常代码与异常筛选器的异常代码相同，则异常事件与异常筛选器匹配。 如果没有异常筛选器，其异常代码与异常事件相同，则默认异常筛选器将处理该异常事件。

命令和参数

事件筛选器可以具有与之关联的调试器命令。 当发生与筛选器匹配的事件时，引擎将执行此命令。 GetEventFilterCommand 和 SetEventFilterCommand 可用于获取和设置此命令。 对于异常筛选器，此命令在第一次发生异常时执行。 可以在发生第二次机会异常事件时执行单独的第二次机会命令。 若要获取和设置第二次机会命令，请使用 GetExceptionFilterSecondCommand 和 SetExceptionSecondChanceCommand。

特定事件筛选器和异常筛选器的参数由 GetSpecificFilterParameters 和 GetExceptionFilterParameters 返回。 可以使用 SetSpecificFilterParameters 和 SetExceptionFilterParameters 设置事件筛选器的中断状态和处理状态。

SetExceptionFilterParameters 还可用于添加和删除任意异常筛选器。

GetEventFilterText 返回特定筛选器的简短说明。

某些特定筛选器采用参数来限制筛选器匹配的事件。 GetSpecificFilterArgument 和 SetSpecificFilterArgument 将为支持参数的特定筛选器获取和设置参数。 如果特定筛选器没有参数，则它与哪些事件匹配没有限制。 下表列出了采用参数的事件筛选器，以及它们如何限制与参数匹配的事件：

事件	匹配条件
创建进程	创建的进程的名称必须与参数匹配。1
退出进程	退出进程的名称必须与参数匹配。1
加载模块	已加载模块的名称必须与参数匹配。1
卸载模块	已卸载模块的基址必须与参数相同。2
目标输出	目标的调试输出必须与参数匹配。3

注意

1. 参数使用 字符串通配符语法 ，并与图像名称进行比较， (在事件发生时忽略路径) 。 如果模块或进程的名称不可用，则被视为匹配项。
2. 参数是设置参数时引擎计算的表达式。
3. 参数使用字符串通配符语法，并与目标的调试输出进行比较。 如果输出未知，则将其视为匹配项。

索引和异常代码

每个事件筛选器都有一个索引。 索引是一个介于 0 到 1 之间的数字，小于筛选器总数 (非独占) 。 可从 GetNumberEventFilters 返回的 *SpecificEvents*、*SpecificExceptions* 和 *ArbitraryExceptions* 值中找到每个类别筛选器的索引范围，如下表所述：

事件筛选器	第一个筛选器的索引	筛选器数
特定事件筛选器	0	*SpecificEvents*
特定异常筛选器	*SpecificEvents*	*SpecificExceptions*
任意异常筛选器	*SpecificEvents* + *SpecificExceptions*	*ArbitraryExceptions*

特定事件筛选器的索引位于主题 DEBUG_FILTER_XXX的第一个表中。 默认异常筛选器 (第一个特定异常筛选器) 的索引为 *SpecificEvents*。 删除任意异常筛选器时，其他任意异常筛选器的索引可能会更改。

异常筛选器通常由异常代码指定。 但是，某些方法需要异常的索引。 若要查找给定异常的异常筛选器的索引，请使用 GetExceptionFilterParameters 循环访问所有异常筛选器，直到找到与异常具有相同异常代码的异常筛选器。 可在特定异常主题中找到特定异常筛选器的 **异常**代码。

系统错误

发生系统错误时，如果错误发生在指定级别或低于指定级别，引擎将中断调试器或将错误打印到输出流。 这些级别由 GetSystemErrorControl 返回，可以使用 SetSystemErrorControl 进行更改。

事件信息

项目 • 2023/06/15

每当调试会话可访问时，都有最后一个 *事件*。 这是导致会话变得可访问的事件。 *事件目标是*生成最后一个事件的目标。 当会话变得可访问时，当前目标将设置为事件目标。 最后一个事件的详细信息由 GetLastEventInformation 返回。 最后一个事件的指令指针以及事件发生时指令指针处的内存由 **请求** 操作 DEBUG_REQUEST_GET_CAPTURED_EVENT_CODE_OFFSET 和 DEBUG_REQUEST_READ_CAPTURED_EVENT_CODE_STREAM返回。

如果目标是故障转储文件，则最后一个 *事件* 是创建转储文件之前发生的最后一个事件。 此事件存储在转储文件中，当获取转储文件作为调试目标时，引擎会为事件回调生成它。

如果目标是内核模式目标并且发生了 *bug 检查*，则可以使用 ReadBugCheckData 找到 bug 检查代码和相关参数。

如果目标是用户模式微型转储，则转储文件生成器可能会存储其他事件。 通常，这是引发生成器保存转储文件的事件。 此事件的详细信息由 GetStoredEventInformation 和 **请求** 操作 DEBUG_REQUEST_TARGET_EXCEPTION_CONTEXT、DEBUG_REQUEST_TARGET_EXCEPTION_THREAD和 DEBUG_REQUEST_TARGET_EXCEPTION_RECORD返回。

转储文件可能包含静态事件列表。 每个事件表示目标在特定时间点的快照。 此列表中的事件数由 GetNumberEvents 返回。 有关列表中每个事件的说明，请使用 GetEventIndexDescription。 若要将此列表中的事件设置为当前事件，请使用 SetNextEventIndex 方法;调用 WaitForEvent 后，事件将成为当前事件。 若要确定列表中的哪个事件是当前事件，请使用 GetCurrentEventIndex。

通过调试器引擎 API 使用断点

项目 • 2023/06/15

断点是事件触发器，当满足断点的条件时，这些触发器将停止目标执行并中断调试器。 断点允许用户在执行达到特定点或访问特定内存位置时分析并修改目标。

调试器引擎通过在 *断点* 的位置修改处理器指令，将软件断点插入目标;此修改对引擎的客户端不可见。 当目标在断点位置执行指令时，将触发软件断点。 *处理器断点*由调试器引擎插入到目标处理器中;其功能特定于处理器。 当访问断点位置的内存时，它由处理器触发;创建断点时会指定哪种类型的访问将触发此断点。

本主题包括以下内容：

设置断点

控制断点标志和参数

设置断点

项目 • 2023/06/15

使用 **AddBreakpoint** 方法创建断点。 此方法创建表示断 点的 IDebugBreakpoint 对象。 它还 (软件 *断点* 或处理器断点) 设置断点类型。 创建断点后，无法更改其类型。

使用 **RemoveBreakpoint** 方法删除断点。 这还会删除 **IDebugBreakpoint** 对象;此对象不能再次使用。

注意 尽管 **IDebugBreakpoint** 实现 **IUnknown** 接口，但 **IUnknown：：AddRef** 和 **IUnknown：：Release** 方法不用于控制断点的生存期。 这些方法不会影响断点的生存期。 相反，在调用 **RemoveBreakpoint** 方法后，将删除 IDebugBreakpoint 对象。

创建断点时，会为其提供唯一的 *断点 ID*。 此标识符不会更改。 但是，删除断点后，其 ID 可用于另一个断点。 有关如何接收删除断点的通知的详细信息，请参阅 监视事件。

创建断点时，它最初被禁用;这意味着它不会导致目标停止执行。 可以使用 **AddFlags** 方法添加DEBUG_BREAKPOINT_ENABLED标志来启用此断点。

首次创建断点时，该断点具有与之关联的内存位置0x00000000。 可以通过将 **SetOffset** 与地址一起使用，或者将 **SetOffsetExpression** 与符号表达式一起使用来更改位置。 在使用断点之前，应从其初始值更改断点的位置。

控制断点标志和参数

项目 • 2023/06/15

有许多方法可用于确定有关断点的基本信息：

- GetId 返回断点 ID。
- GetType 返回软件或处理器) (断点类型，以及设置断点的有效处理器的类型。
- GetAdder 返回添加了断点的客户端。
- GetOffset 返回断点的地址。
- GetOffsetExpression 返回指定断点位置的表达式字符串。

除了位置和断点类型之外，断点还具有多个控制其行为的参数。

可以通过各种特定方法控制断点参数。 此外，可以使用 GetParameters 一起查询大多数参数。

断点标志

断点标志是一种断点参数。

可以使用 GetFlags 查询断点标志。 可以使用 AddFlags、 RemoveFlags 或 SetFlags 更改它们。

断点标志形成位字段。 可在此位字段中使用的可能标志及其含义如下所示：

DEBUG_BREAKPOINT_ENABLED
设置此标志后，将 *启用* 断点，并具有正常效果。 如果未设置此标志，则 *禁用* 断点，不会有任何影响。 如果要暂时停用断点，可以删除此标志;然后，当想要重新启用此断点时，可以轻松地重新添加此标志。

DEBUG_BREAKPOINT_ADDER_ONLY
设置此标志时，断点为 *专用断点*。 此断点仅对添加它的客户端可见。 在这种情况下，其他客户端将无法在引擎中查询断点，并且引擎不会将断点生成的事件发送到其他客户端。 与此断点相关的所有回调 (事件和 输出) 将仅发送到此客户端。 请参阅 GetAdder。

DEBUG_BREAKPOINT_GO_ONLY
设置此标志时，仅当目标处于不受限制的执行状态时，才会触发断点。 如果引擎单步执行目标中的指令，则不会触发该操作。

DEBUG_BREAKPOINT_ONE_SHOT
设置此标志后，断点将在第一次触发时自动删除自身。

DEBUG_BREAKPOINT_DEFERRED
设置此标志后，断点 *将延迟*。 当使用符号表达式指定断点的偏移量时，此标志由引擎设置，并且引擎无法计算表达式。 每次在目标中加载或取消定向模块时，引擎都会尝试重新计算其位置是使用表达式指定的所有断点的表达式。 无法评估的那些将被标记为延迟。 任何客户端都不能修改此标志。

其他断点参数

断点参数还包括：

通过计数
如果断点具有与之关联的传递计数，则在目标通过断点指定次数之前，不会激活该断点。 可以使用 GetPassCount 找到最初设置的传递计数。 可以使用 GetCurrentPassCount 找到引擎在激活断点之前通过断点的剩余次数。 可以使用 SetPassCount 将传递计数重置为新值。

匹配线程
如果断点有与之关联的线程，则当任何其他线程遇到断点时，引擎将忽略它。 可以使用 GetMatchThreadId 找到线程，也可以使用 SetMatchThreadId 更改线程。

命令
断点可能有与之关联的命令。 激活断点时，将执行命令。 可以使用 GetCommand 找到此命令，也可以使用 SetCommand 进行更改。

大小
如果断点是处理器断点，则必须具有指定的大小。 这决定了内存块的大小，该块的访问将激活断点 -- 块的开头是断点的位置。 可以使用 GetDataParameters 找到大小，也可以使用 SetDataParameters 进行更改。

访问类型
如果断点是处理器断点，则必须具有访问类型。 这将确定将激活断点的访问类型。 例如，如果目标从断点读取、写入或执行断点指定的内存，则可能会激活断点。 可以使用 GetDataParameters 找到访问类型，也可以使用 SetDataParameters 更改访问类型。

处理器断点的有效参数

以下访问类型可用于处理器断点：

值	说明

值	说明
DEBUG_BREAK_READ	当 CPU 读取断点的内存块中的内存时，将触发断点。
DEBUG_BREAK_WRITE	当 CPU 在断点的内存块中写入内存时，将触发断点。
DEBUG_BREAK_READ |DEBUG_BREAK_WRITE	当 CPU 读取或写入断点的内存块中的内存时，将触发断点。
DEBUG_BREAK_EXECUTE	当 CPU 提取断点内存块中的指令时，将触发断点。
DEBUG_BREAK_IO	访问断点内存块中的 I/O 端口时，将触发断点。 仅 (Windows XP 和 Microsoft Windows Server 2003，仅限内核模式，仅 x86)

并非所有访问类型和大小都支持所有处理器。 支持以下访问类型和大小：

x86
支持所有访问类型。 DEBUG_BREAK_READ的行为类似于 DEBUG_BREAK_READ |DEBUG_BREAK_WRITE。 大小必须为 1、2 或 4。 断点的地址必须是大小的倍数。

x64
支持所有访问类型。 DEBUG_BREAK_READ的行为类似于 DEBUG_BREAK_READ |DEBUG_BREAK_WRITE。 大小必须为 1、2、4 或 8。 断点的地址必须是大小的倍数。

内存访问

项目 • 2023/06/15

调试器引擎提供*接口*，用于直接读取和写入目标的main内存、寄存器和其他数据空间。

在用户模式调试中，只能访问虚拟内存和寄存器;无法访问物理内存和其他数据空间。

调试器引擎 API 在引用目标上的内存位置时，始终使用 64 位地址。

如果目标使用 32 位地址，则引擎会自动将本机 32 位地址的签名扩展到 64 位地址。 与目标通信时，引擎将在 64 位地址和本机 32 位地址之间自动转换。

本节包括：

虚拟和物理内存

寄存器

其他数据空间

虚拟和物理内存

项目 • 2023/06/15

引擎提供了许多用于读取和写入目标的虚拟和物理内存的方法。

虚拟内存

指定目标虚拟内存中的位置时，将使用目标的虚拟地址空间。 在用户模式调试中，这是当前进程的虚拟地址空间。 在内核模式调试中，这是隐式进程的虚拟地址空间。 有关当前和隐式进程的详细信息，请参阅 线程 和进程。

可以使用 ReadVirtual 读取目标) 的虚拟内存 (，并使用 WriteVirtual 进行写入。

可以使用便捷方法 ReadPointersVirtual 和 WritePointersVirtual 读取和写入目标内存中的指针。 这些方法将在引擎使用的 64 位指针和目标使用的本机指针之间自动转换。 请求包含将用于后续请求的指针的内存（例如，指向字符串的指针）时，这些方法非常有用。

SearchVirtual 和 SearchVirtual2 方法可用于在目标的虚拟内存中搜索字节模式。

FillVirtual 方法可用于多次将字节模式复制到目标的虚拟内存。

还可以使用 ReadVirtualUncached 和 WriteVirtualUncached 方法绕过调试器引擎的虚拟内存缓存来读取和写入目标的虚拟内存。 这些未缓存的版本可用于读取本质上易失的虚拟内存（例如内存映射设备区域），而不会污染缓存或使缓存失效。 应仅在需要未缓存的内存访问的情况下使用，因为未缓存的访问的性能可能明显低于缓存访问。

引擎提供了一些方便的方法，用于从目标的虚拟内存中读取字符串。 若要从目标读取多字节字符串，请使用 ReadMultiByteStringVirtual 和 ReadMultiByteStringVirtualWide。 若要从目标读取 Unicode 字符串，请使用 ReadUnicodeStringVirtual 和 ReadUnicodeStringVirtualWide。

若要查找有关内存位置的信息，请使用 GetOffsetInformation。 并非目标中的所有虚拟地址空间都包含有效的内存。 若要查找区域中的有效内存，请使用 GetValidRegionVirtual。 在目标中手动搜索有效内存时，方法 GetNextDifferentlyValidOffsetVirtual 将找到有效性可能更改的下一个位置。

物理内存

物理内存只能在内核模式调试中直接访问。

可以使用 ReadPhysical 和 ReadPhysical2 读取目标上的物理内存，并使用 WritePhysical 和 WritePhysical2 进行写入。

FillPhysical 方法可用于多次将字节模式复制到目标的物理内存。

可以使用 VirtualToPhysical 方法将目标虚拟地址空间中的地址转换为目标上的物理地址。 可以使用 GetVirtualTranslationPhysicalOffsets 找到用于将虚拟地址转换为物理地址的系统分页结构。

事件

更改目标的虚拟或物理内存时，将调用 IDebugEventCallbacks：：ChangeDebuggeeState 回调方法。

寄存器

项目 • 2023/06/15

调试器引擎可用于检查和更改目标寄存器。

目标上可用的寄存器取决于其处理器体系结构。 有关 x86 处理器寄存器的说明，请参阅处理器体系结构。 有关可用于处理器的寄存器的完整说明，请参阅该处理器的文档。

寄存器集

GetNumberRegisters 方法可用于查找目标上的寄存器数。

每个寄存器通过其索引进行引用。 第一个寄存器的索引为零，最后一个寄存器的索引是寄存器数减一。 若要查找名称已知的寄存器的索引，请使用 GetIndexByName。

方法 GetDescription 返回有关寄存器的信息。 这包括寄存器的名称、它可以保存的值的类型以及它是否为子注册。

*子注册*是包含在另一个寄存器中的寄存器。 当子注册发生更改时，包含它的寄存器也会更改。 例如，在 x86 处理器上， **ax** 子注册与 32 位 **eax** 寄存器的低 16 位相同。

有三个特殊寄存器，可以使用以下方法找到其值。 这些寄存器的值的解释取决于平台。

- 可以使用 GetInstructionOffset 和 GetInstructionOffset2 找到当前指令的位置。
- 可以使用 GetStackOffset 和 GetStackOffset2 找到当前处理器堆栈槽的位置。
- 可以使用 GetFrameOffset 和 GetFrameOffset2 找到当前函数的堆栈帧的位置。

操作寄存器

可以使用 GetValue 方法读取寄存器的值。 可以使用 GetValues 和 GetValues2 读取多个寄存器。

可以使用 SetValue 方法将值写入寄存器。 可以使用 SetValues 和 SetValues2 编写多个寄存器。

将值写入寄存器时，如果提供的值的类型与寄存器的类型不同，则该值将转换为寄存器的类型。 此转换与 CoerceValue 方法执行的转换相同。 如果寄存器的类型不能包含提供的值，则此转换可能会导致数据丢失。

Pseudo-Registers

伪寄存器 是由调试器引擎维护的变量，保存某些值，例如， **$teb** 是伪寄存器的名称，其值为当前线程的线程环境块 (TEB) 的地址。 有关详细信息和伪寄存器列表，请参阅 伪寄存器语法。

每个伪寄存器都有一个索引。 索引是介于零和伪寄存器数之间的数字 - GetNumberPseudoRegisters 返回的 () 减 1。 若要按名称查找伪寄存器的索引，请使用 GetPseudoIndexByName。 可以使用 GetPseudoValues 读取伪寄存器的值，并使用 SetPseudoValues 将值写入伪寄存器。 有关伪寄存器的说明（包括其类型），请使用 GetPseudoDescription。

注意 并非所有伪寄存器在所有调试会话中或在特定会话中随时可用。

显示寄存器

OutputRegisters 和 OutputRegisters2 方法设置目标寄存器的格式，并将其作为输出发送到客户端。

事件

每当目标寄存器的值发生更改时，引擎都会调用 IDebugEventCallbacks：：ChangeDebuggeeState 回调方法，并将参数 *Flags* 设置为 DEBUG_CDS_REGISTERS。

其他数据空间

项目 • 2023/06/15

在内核模式调试中，除了main内存和寄存器之外，还可以将数据读取和写入各种数据空间。 可以访问以下数据空间：

系统总线
ReadBusData 和 WriteBusData 方法读取和写入系统总线数据。

Control-Space Memory
ReadControl 和 WriteControl 读取和写入控制空间内存的方法。

I/O 内存。
方法 ReadIo 和 WriteIo 读取和写入系统和总线 I/O 内存。

特定于模型的寄存器 (MSR)
方法 ReadMsr 和 WriteMsr 读取和写入 MSR，它们是为特定 CPU 模型启用和禁用功能并支持调试的控制寄存器。

处理

在用户模式调试中，可以使用目标进程拥有的系统句柄获取有关系统对象的信息。 ReadHandleData 方法可用于读取此信息。

可以使用 GetCurrentThreadHandle **和** GetCurrentProcessHandle 方法获取线程和进程系统对象的系统 **句柄** 。 当发生 create-thread 和 create-process 调试事件时，这些句柄还会提供给 IDebugEventCallbacks：：CreateThread 和 IDebugEventCallbacks：：CreateProcess 回调方法。

注意 在内核模式下，进程句柄和线程句柄是人工句柄。 它们不是系统句柄。

检查堆栈跟踪

项目 • 2023/06/15

*调用堆栈*包含线程发出的函数调用的数据。 每个函数调用的数据称为 *堆栈帧*，包括返回地址、传递给函数的参数以及函数的局部变量。 每次调用函数时，都会将新的堆栈帧推送到堆栈顶部。 当该函数返回时，堆栈帧将从堆栈中弹出。

每个线程都有自己的调用堆栈，表示在该线程中执行的调用。

若要获取堆栈跟踪，请使用 GetStackTrace 和 GetContextStackTrace 方法。 可以使用 OutputStackTrace 和 OutputContextStackTrace 打印堆栈跟踪。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

控制线程和进程

项目 • 2023/06/14

有关调试器引擎中的线程和进程的概述，请参阅 线程和进程。

发生事件时，事件线程和事件进程将设置为发生事件的 (操作系统或虚拟) 的线程和进程。 可以分别使用 GetEventThread 和 GetEventProcess 找到它们。

隐式线程和进程

在内核模式调试中，调试器引擎将使用 *隐式进程* 来确定在执行虚拟到物理地址转换时要使用的虚拟地址空间，例如，在 VirtualToPhysical 和 ReadVirtual 方法中。 发生事件时，隐式进程设置为当前进程。

可以使用 SetImplicitProcessDataOffset 更改隐式进程。 若要确定隐式进程，请使用 GetImplicitProcessDataOffset。

注意 在实时内核调试会话期间设置 断点 时，调试器引擎会将断点的虚拟地址传递给目标，目标将设置断点。 在这种情况下，处理断点时仅使用目标的进程上下文;隐式进程的值无关紧要。

在内核模式调试中，调试器引擎将使用 *隐式线程* 来确定一些目标 寄存器。 这包括处理器堆栈 (请参阅 GetStackOffset)，帧偏移量 (请参阅 GetFrameOffset)，指令偏移量 (请参阅 GetInstructionOffset)。 发生事件时，隐式线程设置为当前线程。

可以使用 SetImplicitThreadDataOffset 更改隐式线程。 若要确定隐式线程，请使用 GetImplicitThreadDataOffset。

并非所有寄存器都由隐式线程确定。 更改隐式线程时，某些寄存器将保持不变。

警告 隐式进程和隐式线程是独立的。 如果隐式线程不属于隐式进程，则隐式线程的用户和会话状态将位于错误的虚拟地址空间中，并且尝试访问此信息将导致错误或提供不正确的结果。 访问内核内存时不会出现此问题，因为内核内存地址在所有虚拟地址空间中都是恒定的。 因此，可以独立于隐式进程访问位于内核内存中的隐式线程的信息。

线程

调试器 *引擎使用引擎线程 ID* 来标识目标的每个操作系统线程和每个虚拟线程。

停止目标时，每个线程还具有相对于其所属进程的索引。 对于任何进程，进程中第一个线程的索引为零，最后一个线程的索引是进程中的线程数减 1。 可以使用

GetNumberThreads 找到当前进程中的线程数。 可以使用 GetTotalNumberThreads 找到当前目标中所有进程中的线程总数。

可以使用 GetThreadIdsByIndex 从当前进程中一个或多个线程的索引中找到引擎线程 ID 和系统线程 ID。

引擎维护有关每个线程的几条信息。 此信息可以查询当前线程，并可用于查找线程的引擎线程 ID。

仅) (用户模式调试的系统线程 ID
可以使用 GetCurrentThreadSystemId 找到当前线程的系统线程 ID。 对于给定的系统线程 ID，可以使用 GetThreadIdBySystemId 找到相应的引擎线程 ID。

线程环境块 (TEB)
可以使用 GetCurrentThreadTeb 找到当前线程的 TEB 地址。 对于给定的 TEB 地址，可以使用 GetThreadIdByTeb 找到相应的引擎线程 ID。 在内核模式调试中， (虚拟) 线程的 TEB 是发生最后一个事件时在相应处理器上运行的系统线程的 TEB。

数据偏移量
在用户模式调试中， (系统) 线程的数据偏移量是该线程的 TEB 位置。 在内核模式调试中， (虚拟) 线程的数据偏移量是发生最后一个事件时在相应处理器上运行的系统线程的 KTHREAD 结构。 可以使用 GetCurrentThreadDataOffset 找到当前线程的数据偏移量。 对于给定的数据偏移量，可以使用 GetThreadIdByDataOffset 找到相应的引擎线程 ID。

系统句柄
可以使用 GetCurrentThreadHandle 找到当前线程的系统句柄。 对于给定的系统句柄，可以使用 GetThreadIdByHandle 找到相应的引擎线程 ID。 在内核模式调试中，为每个 (虚拟) 进程创建一个人工句柄。 此句柄只能与调试器引擎 API 查询一起使用。

过程

调试器 *引擎使用引擎进程 ID* 来标识目标的每个操作系统进程和每个虚拟进程。

停止目标时，每个进程都有一个相对于目标的索引。 目标中第一个进程的索引为零，最后一个进程的索引是目标中的进程数减 1。 可以使用 GetNumberProcesses 找到当前目标中的进程数。

可以使用 GetProcessIdsByIndex 从当前目标中的一个或多个线程的索引中找到引擎进程 ID 和系统进程 ID。

引擎维护有关每个进程的几条信息。 此信息可以查询当前进程，并可用于查找进程的引擎进程 ID。

仅) (用户模式调试的系统进程 ID
可以使用 GetCurrentProcessSystemId 找到当前进程的系统进程 ID。 对于给定的系统进程 ID，可以使用 GetProcessIdBySystemId 找到相应的引擎进程 ID。

处理环境块 (PEB)
可以使用 GetCurrentProcessPeb 找到当前进程的 PEB 地址。 对于给定的 PEB 地址，可以使用 GetProcessIdByPeb 找到相应的引擎进程 ID。 在内核模式调试中，(虚拟) 进程的 PEB 是上次事件发生时正在运行的系统进程的 PEB。

数据偏移量
在用户模式调试中，(系统) 进程的数据偏移量是该进程的 PEB 位置。 在内核模式调试中，(虚拟) 进程的数据偏移量是上次事件发生时正在运行的系统进程的 KPROCESS 结构。 可以使用 GetCurrentProcessDataOffset 找到当前进程的数据偏移量。 对于给定的数据偏移量，可以使用 GetProcessIdByDataOffset 找到相应的引擎进程 ID。

系统句柄
可以使用 GetCurrentProcessHandle 找到当前进程的系统句柄。 对于给定的系统句柄，可以使用 GetProcessIdByHandle 找到相应的引擎进程 ID。 在内核模式调试中，为 (虚拟) 进程创建人工句柄。 此句柄只能与调试器引擎查询一起使用。

事件

在实时用户模式调试中，每当在目标中创建或退出线程时，都会生成 create-thread 和 exit-thread 调试事件。 这些事件会导致调用 IDebugEventCallbacks：：CreateThread 和 IDebugEventCallbacks：：ExitThread 回调方法。

在实时用户模式调试中，每当在目标中创建或退出进程时，都会生成 create-process 和 exit-process 调试事件。 这些事件会导致调用 IDebugEventCallbacks：：CreateProcess 和 IDebugEventCallbacks：：ExitProcess 回调方法。

有关事件的详细信息，请参阅 监视事件。

其他信息

有关线程和进程（包括 TEB、KTHREAD、PEB 和 KPROCESS 结构）的详细信息，请参阅 Microsoft *Windows Internals* by David Solomon 和 Mark Russinovich。

使用符号

项目 • 2023/10/26

有关符号的概述（包括使用符号文件和符号服务器），请参阅 符号。

符号名称和位置

若要查找给定符号名称的位置，请使用 GetOffsetByName。 有关用于指定符号名称的语法的详细信息，请参阅 符号语法和符号匹配。

如果符号的确切名称未知，或者多个符号具有相同的名称， 则 StartSymbolMatch 将开始搜索名称与给定模式匹配的符号。 有关语法的详细信息，请参阅 字符串通配符语法。

若要查找给定其位置的符号的名称，请使用 GetNameByOffset。 若要在给定位置附近的模块中查找符号的名称，请使用 GetNearNamebyOffset。

注意尽可能使用模块名称限定符号-例如 **mymodule！main**。 否则，如果符号不存在 (例如，由于排版错误) 引擎必须加载并搜索每个模块的符号;此过程可能很慢，尤其是对于内核模式调试。 如果符号名称已使用模块名称进行限定，则引擎只需搜索该模块的符号。

符号使用 **结构**DEBUG_MODULE_AND_ID进行唯一标识。 此结构由 GetSymbolEntriesByName 和 GetSymbolEntriesByOffset 方法返回，它们分别根据符号名称和位置搜索符号。

方法 GetSymbolEntryInformation 使用 DEBUG_SYMBOL_ENTRY 结构返回符号的说明。

若要查找结构中某个字段的偏移量，请使用 GetFieldOffset。 若要在结构中查找给定索引的字段的名称，请使用 GetFieldName。 若要查找给定值的枚举常量的名称，请使用 GetConstantName。

方法 GetSymbolInformation 可以执行多个有关符号的信息请求。

符号选项

许多选项控制符号的加载和卸载方式。 有关这些选项的说明，请参阅 设置符号选项。

符号选项可以使用 AddSymbolOptions 打开，而通过使用 RemoveSymbolOptions 关闭。

GetSymbolOptions 返回当前符号选项。 若要同时设置所有符号选项，请使用 SetSymbolOptions。

重新加载符号

加载符号文件后，引擎将符号信息存储在内部缓存中。 若要刷新此缓存，请使用 **重新加载**。 现在或以后必须再次加载这些符号。

合成符号

合成符号 是标记任意地址以便于参考的一种方法。 可以在任何现有模块中创建合成符号。 AddSyntheticSymbol 方法创建新的合成符号。 可以使用 RemoveSyntheticSymbol 删除合成符号。 重新加载模块的符号会删除与该模块关联的所有合成符号。

符号路径

若要将目录或符号服务器添加到符号路径，请使用 AppendSymbolPath 方法。 整个符号路径由 GetSymbolPath 返回，可以使用 SetSymbolPath 进行更改。

模块

项目 • 2023/06/15

*映像*是 Windows 作为用户模式进程或内核的一部分加载的可执行文件、DLL 或驱动程序。 从中加载映像的文件称为其 *图像文件*。

调试器引擎缓存每个进程的*模块*列表，（在内核模式下，虚拟进程）。 通常，此列表中的每个模块都表示进程中的一个图像。 可以使用 **Reload** 将引擎的模块列表与目标同步。

注意 在内核模式调试中，虚拟进程的引擎模块列表同时包含系统范围的内核模式模块和当前进程的用户模式模块。

模块可以按其在目标虚拟地址空间中的基址指定，也可以按引擎为目标维护的模块列表中的索引来指定。 模块的索引等于它在模块列表中的位置，因此，如果卸载索引较低的模块，此索引将更改。 所有卸载的模块都有索引;这些值始终高于已加载模块的索引。 只要模块保持加载状态，基址就不会更改;在某些情况下，如果卸载模块然后重新加载，则它可能会更改。

索引是介于 0 和目标中的模块数减 1 之间的数字。 通过调用 GetNumberModules 可以找到当前进程中的模块数。

索引可用于通过调用 GetModuleByIndex 查找基址。 可以使用 GetSymbolModule 找到拥有具有给定名称的符号的模块的基址。

以下方法返回指定模块的索引和基址：

- 若要查找具有给定模块名称的模块，请使用 GetModuleByModuleName。
- 其虚拟地址范围包含给定地址的模块由 GetModuleByOffset 返回。 此方法可用于在给定模块基址的情况下查找模块索引。

以下方法返回基址或索引指定的模块的相关信息：

- 模块的名称由 GetModuleNames 和 GetModuleNameString 返回。
- 模块的版本信息由 GetModuleVersionInformation 返回。
- 用于描述模块的某些参数由 GetModuleParameters 返回。 有关此方法返回的参数的详细信息，请参阅 DEBUG_MODULE_PARAMETERS。

卸载的模块

在用户模式调试期间，仅在 Windows Server 2003 及更高版本的 Windows 中跟踪卸载的模块。 早期版本的 Windows 仅在内核模式下跟踪卸载的模块。 跟踪它们时，它们会索引到加载的模块之后。 因此，搜索目标模块的任何方法都将搜索所有已加载的模块，然后搜索卸载的模块。 卸载的模块可用于分析尝试调用卸载的代码导致的故障。

合成模块

可以创建*合成模块*来标记内存区域。 这些模块不能包含真实符号，但可以包含合成符号。 **AddSyntheticModule** 方法创建新的合成模块。 可以使用 **RemoveSyntheticModule** 删除合成模块。 重载目标中的所有模块会删除所有合成模块。

映像路径

搜索 *可执行映像* 时，引擎将使用可执行映像路径。

可执行映像路径可以包含多个以分号 (分隔的目录 ;) 。 这些目录按顺序搜索。

有关可执行映像路径的概述，请参阅可执行映像路径。

若要将目录添加到可执行映像路径，请使用 **AppendImagePath** 方法。 整个可执行映像路径由 **GetImagePath** 返回，可以使用 **SetImagePath** 进行更改。

其他信息

有关进程和虚拟进程的详细信息，请参阅 线程和进程。

类型

项目 • 2023/06/15

模块符号文件中的类型信息由两条信息标识：类型 ID 和类型所属模块的基址。 以下方法可用于查找类型 ID：

- GetTypeId 返回给定类型名称的类型 ID。
- GetSymbolTypeId 返回具有给定名称的符号类型的类型 ID。
- GetOffsetTypeId 返回在给定位置找到的符号的类型 ID。

类型的名称和大小分别由 GetTypeName 和 GetTypeSize 返回。

以下便捷方法可用于在目标的物理和虚拟内存中读取和写入类型化数据：

ReadTypedDataPhysical

WriteTypedDataPhysical

ReadTypedDataVirtual

WriteTypedDataVirtual

打印类型化数据

若要格式化类型化数据并将其发送到输出回调，请分别对目标物理内存和虚拟内存中的数据使用 *OutputTypedDataPhysical* 和 *OutputTypedDataVirtual* 。

DEBUG_TYPEOPTS_XXX中所述的类型选项会影响引擎在将类型化数据发送到输出回调之前设置格式的方式。

可使用 AddTypeOptions 打开类型选项，并使用 RemoveTypeOptions 关闭类型选项。

GetTypeOptions 返回当前类型选项。 若要同时设置所有类型选项，请使用 SetTypeOptions。

使用类型信息解释原始数据

调试器引擎 API 支持解释类型化数据。 这提供了一种在目标上遍历对象层次结构的方法，包括查找结构成员、取消引用指针和定位数组元素。

类型化数据由 DEBUG_TYPED_DATA 结构的实例描述，并表示目标强制转换为特定类型的内存区域。 DEBUG_REQUEST_EXT_TYPED_DATA_ANSI**请求**操作用于操作这些实例。

可以将其初始化为表达式的结果，也可以通过将内存区域强制转换为指定类型来初始化它们。 有关DEBUG_REQUEST_EXT_TYPED_DATA_ANSI **请求** 操作支持的所有子操作的列表，请参阅 **EXT_TDOP**。

其他信息

有关输出回调的详细信息，请参阅 输入和输出。

范围和符号组

项目 • 2023/06/15

*符号组*包含一组符号，用于作为一个组进行高效操作。 可以手动创建和填充符号组，也可以根据词法范围中的符号（例如局部变量和函数参数）自动生成和更新符号组。 接口 IDebugSymbolGroup 用于表示符号组。

有两种方法可以创建符号组。 CreateSymbolGroup 返回空符号组，当前词法范围的符号组由 GetScopeSymbolGroup 返回。

注意从当前范围生成的符号组是局部变量的快照。 如果目标中发生任何执行，符号可能不再准确。 此外，如果当前范围发生更改，符号组将不再表示 *当前* 作用域 (，因为它将继续表示) 为其创建的范围。

可以使用 AddSymbol 将符号添加到符号组，并使用 RemoveSymbolByIndex 或 RemoveSymbolByName 删除符号。 方法 OutputAsType 告知调试器在处理符号的数据时使用不同的符号类型。

注意 作用域内符号的值可能不准确。 具体而言，计算机体系结构和编译器优化可能会阻止调试器准确确定符号的值。

*符号条目信息*是符号的说明，包括其位置和类型。 若要在模块中查找符号的此信息，请使用 IDebugSymbols3：：GetSymbolEntryInformation。 若要查找符号组中某个符号的此信息，请使用 IDebugSymbolGroup2：：GetSymbolEntryInformation。 有关符号输入信息的详细信息，请参阅 DEBUG_SYMBOL_ENTRY 。

以下方法返回有关符号组中的符号的信息：

- GetSymbolName 返回符号的名称。
- 如果符号具有绝对地址，GetSymbolOffset 将返回符号的目标虚拟地址空间中的绝对地址。
- 如果符号包含在寄存器中，则 GetSymbolRegister 返回包含符号的寄存器。
- GetSymbolSize 返回符号的数据大小。
- GetSymbolTypeName 返回符号的类型的名称。
- GetSymbolValueText 将符号的值作为字符串返回。

如果符号存储在调试器引擎已知的寄存器或内存位置中，则可以使用 WriteSymbol 更改其值。

如果符号包含其他 *符号* ，则为父符号。 例如，结构包含其成员。 如果符号包含在另一个符号中，则符号是 *子* 符号。 符号可以是父符号和子符号。 每个符号组都有一个平面结构，并包含父符号及其子级。 每个符号都有 *一个深度* -- 符号组中没有父级的符号的深度为零，每个子符号的深度比其父符号的深度大 1。 父符号的子级可能存在于符号组中，也可能不存在。 当子级存在于符号组中时，父符号称为 *展开*符号。 若要添加或删除符号组中符号的子级，请使用 ExpandSymbol。

符号组中的符号数由 GetNumberSymbols 返回。 符号组中符号的 *索引* 是一个标识号;索引范围从零到符号数减 1。 每次向符号组添加或删除符号时（例如，通过展开符号），符号组中所有符号的索引都可能会更改。

可以使用 GetSymbolParameters 找到符号参数，包括有关父子关系的信息。 此方法返回 DEBUG_SYMBOL_PARAMETERS 结构。

可以使用 OutputSymbols 方法将符号组中的符号打印到调试器的输出流中。

范围

*当前范围*或*当前本地上下文*确定调试器引擎公开的局部变量。 范围包含三个组件：

1. 堆栈帧。
2. 当前指令。
3. 寄存器上下文。

如果堆栈帧位于调用堆栈的顶部，则当前指令是导致最后一个事件的指令。 否则，当前指令是导致下一个更高的堆栈帧的函数调用。

GetScope 和 SetScope 方法可用于获取和设置当前范围。 发生事件时，当前范围设置为事件的作用域。 可以使用 ResetScope 将当前范围重置为最后一个事件的范围。

线程上下文

*线程上下文*是 Windows 在切换线程时保留的状态。 这类似于寄存器上下文，只不过存在一些仅限内核的处理器状态，该状态是寄存器上下文的一部分，而不是线程上下文的一部分。 在内核模式调试期间，此额外状态可用作寄存器。

线程上下文由 ntddk.h 中定义的 CONTEXT 结构表示。 此结构依赖于平台，其解释取决于有效的处理器类型。 GetThreadContext 和 SetThreadContext 方法可用于获取和设置线程上下文。

使用源文件

项目 • 2023/06/15

调试器引擎维护*源路径*，该路径是包含与当前目标关联的源代码文件的目录和源服务器的列表。 调试器引擎可以在这些目录和源服务器中搜索源文件。 借助符号文件，调试器引擎可以将源文件中的行与目标内存中的位置进行匹配。

有关将源文件与调试器配合使用的概述，请参阅 在源模式下调试。 有关源路径的概述，请参阅 源路径。 有关从调试器引擎使用源服务器的概述，请参阅 使用源服务器。

源路径

若要将目录或源服务器添加到源路径，请使用 AppendSourcePath 方法。 整个源路径由 GetSourcePath 返回，可以使用 SetSourcePath 进行更改。 可以使用 GetSourcePathElement 从源路径检索单个目录或源服务器。

若要查找相对于源路径的源文件，请使用 FindSourceFile ;对于使用源服务器时更高级的选项，请使用 FindSourceFileAndToken。 FindSourceFileAndToken 还可以与 GetSourceFileInformation 一起使用，以检索与源服务器上的文件相关的变量。

将源文件与内存中的代码匹配

调试器引擎提供了三种方法，用于查找与源文件中的行对应的内存位置。 若要将单行源代码映射到内存位置，请使用 GetOffsetByLine。 若要搜索多个源行或附近源行的内存位置，请使用 GetSourceEntriesByLine。 GetSourceFileLineOffsets 方法将返回源文件中每一行的内存位置。

若要执行相反的操作并查找与目标内存中某个位置匹配的源文件行，请使用 GetLineByOffset。

注意 内存位置与源文件中的行之间的关系不一定是一对一的。 单个源代码行可以对应于多个内存位置，而单个内存位置可以对应于多行源代码。

连接到目标

项目 • 2024/03/20

本节包括：

实时用户模式目标

实时内核模式目标

转储文件目标

远程目标

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

实时用户模式目标

项目 • 2023/06/15

用于创建和附加到本主题中列出的进程的方法可用于本地计算机和运行进程服务器的远程计算机。

可以使用 Create Process 或 CreateProcess2 创建用户模式进程，后者执行给定的命令来创建进程。 方法 AttachProcess 可用于将 调试器引擎 附加到现有用户模式进程。 CreateProcessAndAttach 和 CreateProcessAndAttach2 创建一个新的用户模式进程，并将其附加到同一计算机上的其他用户模式进程。 **请求**操作 DEBUG_REQUEST_GET_ADDITIONAL_CREATE_OPTIONS、 DEBUG_REQUEST_SET_ADDITIONAL_CREATE_OPTIONS和 DEBUG_REQUEST_SET_LOCAL_IMPLICIT_COMMAND_LINE可用于设置一些用于创建进程的默认选项。

注意 在调用 WaitForEvent 方法之前，引擎不会完全附加到进程。 只有在进程生成事件（例如进程创建事件）后，它才会在调试器会话中可用。 有关更多详细信息，请参阅 调试会话和执行模型 。

GetRunningProcessSystemIds 方法将返回计算机上所有正在运行的进程的进程 ID。 可以使用 GetRunningProcessSystemIdByExecutableName 找到特定程序的进程 ID。 给定进程 ID 后， GetRunningProcessDescription 将返回进程说明。

处理选项

进程选项确定附加到用户模式进程时引擎的部分行为，包括调试器引擎是否会自动附加到目标进程创建的子进程，以及引擎在退出目标进程时对目标进程执行的操作。 有关过程选项的说明，请参阅 DEBUG_PROCESS_XXX 。

可以使用 GetProcessOptions 查询进程选项。 可以使用 AddProcessOptions、 RemoveProcessOptions 和 SetProcessOptions 更改它们。

断开与进程的连接

引擎可通过三种不同的方式与进程断开连接。

1. *分离*。 恢复进程中的所有线程，使其继续运行，不再被调试。 DetachCurrentProcess 将从当前进程分离引擎， DetachProcesses 将从所有进程分离引擎。 并非所有目标都支持分离。 如果目标支持分离，可以使用**请求**操作 DEBUG_REQUEST_TARGET_CAN_DETACH检查。

2. *终止*。 尝试终止进程。 TerminateCurrentProcess 将终止当前进程，TerminateProcesses 将终止调试器会话中的所有进程。

3. *放弃*。 从正在调试的进程列表中删除进程。 操作系统仍会将进程视为正在调试，并且它将保持挂起状态，直到另一个调试器附加到它或终止它。AbandonCurrentProcess 将放弃当前进程。

实时内核模式目标

项目 • 2023/06/15

若要将 调试器引擎 附加到目标计算机进行内核模式调试，请使用 AttachKernel 方法。

注意 在调用 WaitForEvent 方法之前，引擎不会完全附加到内核。 只有在内核生成事件（例如， 初始断点 ）后，它才会在调试器会话中可用。 有关更多详细信息，请参阅 调试会话和执行模型 。

如果调试器引擎附加到不是本地内核的内核，并且连接不是 eXDI 连接，则可以使用 GetKernelConnectionOptions 查询用于查找目标计算机的连接选项。 还可以使用 SetKernelConnectionOptions 重新同步连接或更改连接速度。

调试器可以连接到本地内核，但仅以有限的方式连接到本地内核，并且仅当计算机是使用 **/debug** 启动开关启动的。（在某些 Windows 安装中，使用其他开关（如 **/debugport**）时支持本地内核调试，但这不是 Windows 的保证功能，不应依赖。） IsKernelDebuggerEnabled 用于确定本地计算机是否可用于调试。 有关单个计算机上的内核调试的详细信息，请参阅 执行本地内核调试。

转储文件目标

项目 • 2023/06/15

有关故障转储文件的简介和概述，请参阅 故障转储文件。

打开转储文件

若要打开故障转储文件以用作调试器目标，请使用 OpenDumpFile 或 OpenDumpfileWide。 这些方法类似于 .opendump 调试器命令。

注意 在调用 WaitForEvent 方法之前，引擎不会完全附加到转储文件。 从进程或内核创建转储文件时，有关最后一个事件的信息将存储在转储文件中。 打开转储文件后，下次尝试执行时，引擎将为事件回调生成此事件。 只有这样，转储文件才会在调试会话中可用。 有关更多详细信息，请参阅 调试会话和执行模型 。

可以使用其他文件来帮助调试故障转储文件。 AddDumpInformationFile 和 AddDumpInformationFileWide 方法注册包含下一个转储文件时要使用的页文件信息的文件。 在打开转储文件之前，必须调用这些方法。 GetNumberDumpFiles 将返回打开当前转储文件时使用的此类文件数， GetDumpFile 将返回这些文件的说明。

用户模式小型转储文件包含多个信息流。 可以使用 **请求** 操作 DEBUG_REQUEST_READ_USER_MINIDUMP_STREAM读取这些流。

创建转储文件

若要创建当前目标（用户模式或内核模式）的故障转储文件，请使用 WriteDumpFile2。 此方法类似于 .dump 调试器命令。

远程目标

项目 • 2023/06/15

有两种不同形式的远程调试，具体取决于远程客户端或服务器) 是主计算机 (计算机。 *主计算机*是调试器引擎处于活动状态的计算机。 在其他计算机上，调试器引擎仅充当代理，将命令和数据中继到主机引擎。

所有调试器操作（例如执行命令和 扩展以及符号加载）都由主机引擎执行。 调试器会话也相对于主机引擎。

若要列出计算机上当前运行的调试服务器和进程服务器，请使用 **OutputServers**。

调试服务器和调试客户端

*调试服务器*是调试器引擎的实例，充当主机并侦听来自调试客户端的连接。 **StartServer** 方法将告知调试器引擎开始侦听来自调试客户端的连接。

*调试客户端*是充当代理的调试器引擎的实例，将调试器命令和 I/O 发送到调试服务器。 **DebugConnect** 函数可用于连接到调试服务器。

DebugConnect 返回的客户端对象不会自动加入调试服务器上的调试器会话。 **ConnectSession** 方法可用于加入会话，同步输入和输出。

调试服务器和调试客户端之间的通信主要包括发送到服务器的调试器命令和 RPC 调用，以及发送回客户端的命令输出。

进程服务器、内核连接服务器和智能客户端

进程服务器 和 *内核连接服务器* 都是调试器引擎的实例，充当代理、侦听来自智能客户端的连接，以及根据这些远程客户端的请求执行内存、处理器或操作系统操作。 *进程服务器*有助于调试在同一台计算机上运行的进程。 *内核连接服务器*有助于调试连接到运行连接服务器的计算机的 Windows 内核调试目标。 可以使用 **StartProcessServer** 方法或 DbgSrv 程序启动进程服务器。 **WaitForProcessServerEnd** 方法将等待使用 **StartProcessServer** 启动的进程服务器结束。 可以使用 **KdSrv** 程序激活内核连接服务器。

*智能客户端*是充当主机引擎并连接到进程服务器的调试器引擎的实例。 **ConnectProcessServer** 方法将连接到进程服务器。 连接后，可以使用 Live User-Mode Targets 中所述的方法。

远程客户端完成进程服务器后，可以使用 DisconnectProcessServer 断开连接，也可以使用 EndProcessServer 请求进程服务器关闭。 若要从运行进程服务器的计算机关闭进程服务器，请使用任务管理器结束进程。 如果使用 StartProcessServer 的调试器引擎实例仍在运行，则它可以使用 Execute 发出调试器命令 .endsrv 0，这将结束进程服务器 (这是 .endsrv 的常见行为的异常，这通常不会影响进程服务器) 。

进程服务器与智能客户端之间的通信通常包括从远程客户端发送到服务器的低级别内存、处理器和操作系统操作和请求。 然后，其结果将发送回客户端。

目标信息

项目 • 2024/05/27

GetDebuggeeType 方法会返回当前目标的性质（例如，是内核模式目标还是用户模式目标），以及调试器引擎与其连接的方式。

如果目标是崩溃转储文件，则 GetDumpFormatFlags 方法将指明转储中包含哪些信息。

目标计算机

目标计算机的页面大小由 GetPageSize 返回。 IsPointer64Bit 将指明计算机是使用 32 位还是 64 位地址。

注意在内部，调试器引擎始终使用 64 位目标地址。 如果目标只使用 32 位地址，则引擎会在与目标通信时自动转换这些地址。

目标计算机中的处理器数由 GetNumberProcessors 返回。

与目标计算机关联的处理器类型有三种：

- *实际处理器类型*是目标计算机中的物理处理器类型。 该值由 GetActualProcessorType 返回。
- *正在执行的处理器类型*是当前正在执行的处理器上下文中使用的处理器类型。 该值由 GetExecutingProcessorType 返回。
- *有效的处理器类型*是调试器在解释来自目标的信息（例如设置断点、访问寄存器和获取堆栈跟踪）时使用的处理器类型。 有效处理器类型由 GetEffectiveProcessorType 返回，可使用 SetEffectiveProcessorType 进行更改。

有效的处理器类型和执行处理器类型可能与实际处理器类型不同，例如，当物理处理器是 x64 处理器但在 x86 模式下运行时。

目标计算机上的物理处理器支持的不同执行处理器类型由 GetPossibleExecutingProcessorTypes 返回。 这些类型的数量由 GetNumberPossibleExecutingProcessorTypes 返回。

调试器引擎支持的处理器类型列表由 GetSupportedProcessorTypes 返回。 支持的处理器类型数量由 GetNumberSupportedProcessorTypes 返回。

处理器类型的名称（完整和缩写）由 GetProcessorTypeNames 返回。

目标计算机上的当前时间由 GetCurrentTimeDate 返回。 目标计算机自上次启动以来的运行时间长度由 GetCurrentSystemUpTime 返回。 时间信息可能不适用于所有目标。

目标版本

目标计算机上运行的 Windows 版本由 GetSystemVersionValues 和 Request 操作 DEBUG_REQUEST_GET_WIN32_MAJOR_MINOR_VERSIONS 返回，而 Windows 版本的描述由 GetSystemVersionString 返回。 GetSystemVersion 也会返回其中一些信息。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

目标状态

项目 • 2023/06/15

方法 OutputCurrentState 会将目标的当前状态输出到调试器的输出流中。

目标的当前执行状态由 GetExecutionStatus 返回。 如果目标已挂起，则 SetExecutionStatus 方法可用于恢复其中一种执行模式中的执行。

方法 GetReturnOffset 返回当前函数返回时将执行的指令的地址。

GetNearInstruction 返回相对于给定地址的指令的位置。

检查堆栈跟踪

*调用堆栈*包含线程进行的函数调用的数据。 每个函数调用的数据称为 *堆栈帧*，包括返回地址、传递给函数的参数以及函数的局部变量。 每次进行函数调用时，都会将新的堆栈帧推送到堆栈顶部。 当该函数返回时，堆栈帧将从堆栈中弹出。 每个线程都有自己的调用堆栈，表示在该线程中进行的调用。

注意 并非函数调用的所有数据都可以存储在堆栈帧中。 参数和局部变量有时可以存储在寄存器中。

若要检索调用堆栈或 *堆栈跟踪*，请使用 GetStackTrace 和 GetContextStackTrace 方法。 可以使用 OutputStackTrace 和 OutputContextStackTrace 打印堆栈跟踪。

调用扩展和扩展函数

项目 • 2024/03/11

若要加载扩展库（或获取已加载扩展库的句柄），请使用 AddExtension。 可以使用 RemoveExtension 卸载扩展库。

可以使用 CallExtension 调用扩展命令。

扩展函数

*扩展函数*是由扩展库导出的函数。 它们可以使用任何函数原型，并使用 C 函数指针直接调用。

它们不是扩展命令，也不能通过调试器命令使用。 扩展函数不能远程调用；必须直接调用。 因此，不能在调试客户端时使用。 只有当客户端对象位于主机引擎内部时（不进行远程调试或使用智能客户端时），才能调用。

扩展函数在扩展库中由名称前面追加的"_EFN_"标识。

若要获取指向扩展函数的指针，请使用 GetExtensionFunction。 此函数指针的类型应与扩展函数的原型相匹配。 扩展函数现在可以像 C 中的任何其他函数指针一样调用。

示例

如果以下扩展函数包含在扩展库中并加载到调试器引擎中：

C++

```
HRESULT CALLBACK
_EFN_GetObject(IDebugClient * client, SomeObject * obj);
```

可以使用以下方式调用：

C++

```
typedef ULONG (CALLBACK * GET_OBJECT)(IDebugClient * client, SomeObject *
obj);



HRESULT status = S_OK;
GET_OBJECT extFn = NULL;
SomeObject myObj;

if (g_DebugControl->
```

```
        GetExtensionFunction(0,
                             "GetObject",
                             (FARPROC *) &extFn ) == S_OK &&
    extFn)
{
    status = (*extFn)(client, &myObj);
}
```

汇编和反汇编指令

项目 • 2024/02/14

调试器引擎支持使用程序集语言在目标中显示和更改代码。 有关调试器中程序集语言用法的概述，请参阅 在程序集模式下调试。

请注意，所有体系结构都不支持程序集语言。 在某些体系结构上，并非所有说明都受支持。

若要组装单个程序集语言指令并将生成的处理器指令置于目标内存中，请使用 Assemble。

若要从目标获取处理器指令并生成表示程序集指令的字符串来反汇编单个指令，请使用反汇编。

GetDisassembleEffectiveOffset **方法**返回要反汇编的最后一个指令的第一个有效地址。 例如，如果要反汇编的最后一个指令是 `move ax, [ebp+4]`，则有效地址是值 `ebp+4`。 这对应于 **$ea** 伪寄存器。

若要将反汇编指令发送到输出回调，请使用 OutputDisassembly *和 OutputDisassemblyLines 的方法。*

调试器引擎具有一些控制程序集和反汇编的选项。 这些选项由 **GetAssemblyOptions** 返回。 可以使用 SetAssemblyOptions **设置**它们，并且某些选项可以使用 AddAssemblyOptions **打开**，或者通过 **RemoveAssemblyOptions** 关闭。

编写 DbgEng 扩展

项目 • 2024/03/20

本节包括：

DbgEng 扩展设计指南

DbgEng 扩展参考

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

DbgEng 扩展设计指南

项目 • 2023/03/08

本节包括：

DbgEng 扩展 DLL 剖析

使用客户端和引擎

编写 DbgEng 扩展代码

生成 DbgEng 扩展

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

DbgEng 扩展 DLL 剖析

项目 • 2024/02/14

DbgEng 扩展 DLL 导出多个回调函数，其中一些函数可能是扩展命令的实现。

这些扩展 DLL 由 调试器引擎 加载，可以在 Microsoft Windows 上执行用户模式或内核模式调试时提供额外的功能或任务自动化。

如果执行了 Windows 调试工具的完整安装，可以在安装目录的 sdk\samples\exts 子目录中找到名为“exts”的示例 DbgEng 扩展。

扩展命令

扩展 DLL 可以导出用于执行扩展命令的任意数量的函数。 每个函数在 .def 文件中显式声明为导出，其名称必须完全由小写字母组成。

用于实现扩展命令的函数必须与原型 **PDEBUG_EXTENSION_CALL**匹配。

这些函数是根据标准 C++ 约定命名的，但不允许使用大写字母。 导出的函数名称和扩展命令名称相同，但扩展命令以感叹号 （！） 开头。 例如，将myextension.dll加载到调试器中，然后在调试器命令窗口中键入 **！stack** 时，调试器将在myextension.dll中查找名为 **堆栈** 的导出函数。

如果尚未加载myextension.dll，或者如果其他扩展 DLL 中可能有同名的其他扩展命令，则可以在调试器命令窗口中键入 **！myextension.stack** ，以指示该 DLL 中的扩展 DLL 和扩展命令。

其他导出的函数

DbgEng 扩展 DLL 必须导出 DebugExtensionInitialize。 加载 DLL 时会调用此调用，以初始化 DLL。 DLL 可以使用它初始化全局变量。

扩展 DLL 可以导出 DebugExtensionUninitialize。 如果导出此代码，则在卸载扩展 DLL 之前将调用它。 DLL 可以使用它进行清理，然后再将其卸载。

扩展 DLL 可以导出 DebugExtensionNotify。 如果导出此项，则会在会话开始或结束以及目标启动或停止执行时调用它。 这些通知也提供给 向客户端注册的 IDebugEventCallbacks 对象。

扩展 DLL 可以导出 KnownStructOutput。 如果导出此内容，将在加载 DLL 时调用它。 此函数返回 DLL 知道如何在单行上打印的结构列表。 稍后可以调用它来设置这些结构的实例的格式以供打印。

用于加载 DbgEng 扩展 DLL 的引擎过程

加载扩展 DLL 时，按以下顺序由引擎调用回调函数：

1. **DebugExtensionInitialize** 被调用，以便扩展 DLL 可以初始化。
2. 如果导出， **则如果引擎具有活动会话，则调用 DebugExtensionNotify** ，并在会话挂起且可访问时再次调用。
3. 如果导出， **则调用 KnownStructOutput** 以请求 DLL 知道如何在单行上打印的结构列表。

有关如何 使用调试器加载和卸载扩展 DLL 的信息，请参阅加载调试器扩展 DLL ，并参阅 使用调试器扩展命令了解有关执行扩展命令 的信息。

调试器引擎将在调用扩展 DLL 时放置 **try/except** block。 这会保护引擎免受扩展代码中某些类型的 bug;但是，由于扩展调用在引擎所在的同一线程中执行，因此它们仍可能导致它崩溃。

使用客户端和引擎

项目 • 2023/06/15

DbgEng 扩展通过客户端对象与 调试器引擎 交互。

调用扩展函数时，会向客户端传递该扩展函数。 扩展函数应将此客户端用于其与调试器引擎的所有交互，除非它有使用另一个客户端的特定原因。

扩展库可以在初始化时使用 **DebugCreate** 创建自己的客户端对象。 此客户端可用于从 DLL 注册回调对象。

注意 修改传递给扩展函数的客户端时，应小心谨慎。 具体而言，向此客户端注册回调可能会中断调试器的输入、输出或事件处理。 建议创建新的客户端来注册回调。

编写 DbgEng 扩展代码

项目 • 2023/06/15

DbgEng 扩展 命令可以包含任何标准 C++ 代码。 除了 wdbgexts.h 头文件中出现的 C 函数外，它们还可以包括 dbgeng.h 头文件中出现的 C++ 接口。

如果打算使用 wdbgexts.h 中的函数，则需要在包含 wdbgexts.h 之前定义KDEXT_64BIT。 例如：

C++

```
#define KDEXT_64BIT
#include wdbgexts.h
#include dbgeng.h
```

有关 dbgeng.h 中可在扩展命令中使用的接口的完整列表，请参阅 调试器引擎参考。

有关 wdbgexts.h 中可在扩展命令中使用的函数的完整列表，请参阅 WdbgExts 函数。 其中许多函数出现在 32 位版本和 64 位版本中。 通常，64 位版本以“64”结尾，32 位版本没有数字结尾，例如 **ReadIoSpace64** 和 **ReadIoSpace**。 从 DbgEng 扩展调用 wdbgexts.h 函数时，应始终使用以“64”结尾的函数名称。 这是因为 调试器引擎 始终在内部使用 64 位指针，而不考虑目标平台。

如果在 DbgEng 扩展中包含 wdbgexts.h，则应在扩展 DLL 初始化期间调用 **GetWindbgExtensionApis64** (请参阅 *DebugExtensionInitialize*) 。

注意 不得尝试从任何调试器扩展调用任何 DbgHelp 或 ImageHlp 例程。 不支持调用这些例程，可能会导致各种问题。

生成 DbgEng 扩展

项目 • 2024/03/04

所有调试器扩展都应使用 Visual Studio 编译和生成。 生成实用工具不再与调试器扩展一起使用。

有关在 Visual Studio 中生成项目的文档，请参阅 Visual Studio projects - C++。

若要生成扩展，请使用以下过程：

生成调试器扩展

1. 在 Visual Studio 中打开 **dbgsdk.sln** 示例项目。
2. 检查 include 和 lib 文件项目设置。 如果 *%debuggers%* 表示适用于 Windows 的调试工具安装的根目录，则应按如下所示进行设置：

```text
Include Path
%debuggers%\sdk\inc
Library Path
%debuggers%\sdk\lib
```

如果已将这些标头和库移动到其他位置，请改为指定该位置。

3. 从 Visual Studio 的菜单中选择**生成**，然后选择**生成解决方案**。

EngExtCpp 扩展

项目 • 2024/03/20

本节包括：

EngExtCpp 扩展设计指南

EngExtCpp 扩展参考

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

EngExtCpp 扩展设计指南

项目 • 2023/06/15

本节包括：

EngExtCpp 扩展库

客户端对象和引擎

编写 EngExtCpp 扩展

生成 ExtEngCpp 扩展

分析扩展参数

类型化数据

EngExtCpp 扩展库

项目 • 2023/06/14

EngExtCpp 扩展库是一个 DLL，它使用 EngExtCpp.h 中的 EngExtCpp 扩展框架。 当调试器引擎加载此库时，其方法和函数可以在 Microsoft Windows 上执行用户模式或内核模式调试时提供额外的功能或任务自动化。

EngExtCpp 扩展框架构建在 DbgEng 扩展框架之上。 它提供用于与调试器引擎交互的相同调试器引擎 API。 但它还提供其他功能，使常见任务更简单。

如果执行了 Windows 调试工具的完整安装，则可以在安装目录的 sdk\samples\extcpp 子目录中找到名为“extcpp”的示例 EngExtCpp 扩展。

EXT_CLASS 和 ExtExtension

EngExtCpp 扩展库的核心是 **EXT_CLASS** 类的单个实例。 EngExtCpp 扩展库将提供此类的实现，该类包含库导出的格式设置结构的所有扩展命令和方法。

EXT_CLASS是 **ExtExtension** 的子类。 此类的单个实例是使用 **EXT_DECLARE_GLOBALS** 宏创建的，该宏必须在扩展库的源文件中恰好出现一次。

加载扩展库时，引擎会调用 类的 **Initialize** 方法，在卸载类之前调用 **Uninitialize** 方法。 此外，引擎调用 **OnSessionActive**、 **OnSessionInactive**、 **OnSessionAccessible** 和 **OnSessionInaccessible** 方法来通知扩展库调试会话的状态。

扩展命令

EXT_CLASS 类可以包含许多用于执行扩展命令的方法。 使用 **EXT_COMMAND_METHOD 宏在** EXT_CLASS 类中声明每个扩展命令。 命令的实现是使用 **EXT_COMMAND** 宏定义的。

已知结构

EXT_CLASS 类可以包含许多使用 *ExtKnownStructMethod* 原型的方法。 引擎可以使用这些方法来设置某些结构类型的实例的格式以供输出。

提供的值

EXT_CLASS 类可以包含许多使用 **ExtProvideValueMethod** 原型的方法。 引擎可以使用这些方法来评估扩展提供的一些伪寄存器。

客户端对象和引擎

项目 • 2024/05/27

EngExtCpp 扩展通过客户端对象与调试器引擎交互。 客户端对象的接口指针可通过 **ExtExtension** 基类的成员提供给扩展。 以下成员可访问第一个版本的引擎 API 接口。

展开表

引擎 API 接口	ExtExtension 成员
IDebugAdvanced	m_Advanced
IDebugClient	m_Client
IDebugControl	m_Control
IDebugDataSpaces	m_Data
IDebugRegisters	m_Registers
IDebugSymbols	m_Symbols
IDebugSystemObjects	m_System

以下成员可访问较新版本的引擎 API 接口。 并非所有版本的调试器引擎都提供这些接口。 如果它们不可用，任何使用它们的尝试都会引发异常。

展开表

引擎 API 接口	ExtExtension 成员
IDebugAdvanced2	m_Advanced2
IDebugAdvanced3	m_Advanced3
IDebugClient2	m_Client2
IDebugClient3	m_Client3
IDebugClient4	m_Client4
IDebugClient5	m_Client5
IDebugControl2	m_Control2
IDebugControl3	m_Control3
IDebugControl4	m_Control4

引擎 API 接口	ExtExtension 成员
IDebugData2	m_Data2
IDebugData3	m_Data3
IDebugData4	m_Data4
IDebugRegisters2	m_Registers2
IDebugSymbols2	m_Symbols2
IDebugSymbols3	m_Symbols3
IDebugSystemObjects2	m_System2
IDebugSystemObjects3	m_System3
IDebugSystemObjects4	m_System4

每次使用扩展库执行扩展命令或格式化输出结构时，这些表中的成员都会被初始化。 任务完成后，这些成员不会被初始化。 因此，扩展不应缓存这些成员的值，而应直接使用 ExtExtension 成员。

扩展库还可以使用方法 IDebugClient::CreateClient 或函数 DebugCreate 或 DebugConnect 来创建自己的客户端对象。

有关客户端对象的概述，请参阅客户端对象。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

编写 EngExtCpp 扩展

项目 • 2023/06/15

EngExtCpp 扩展库可以包含任何标准 C++ 代码。 除了 wdbgexts.h 头文件中出现的 C 函数外，它还包括出现在 engextcpp.h 和 dbgeng.h 头文件中的 C++ 接口。 dbgeng.h 和 wdbgexts.h 均包含在 engextcpp.h 中。

有关 dbgeng.h 中可在扩展命令中使用的接口的完整列表，请参阅 调试器引擎参考。

有关 wdbgexts.h 中可在扩展命令中使用的函数的完整列表，请参阅 WdbgExts 函数。 其中许多函数出现在 32 位版本和 64 位版本中。 通常，64 位版本以“64”结尾，32 位版本没有数字结尾，例如 **ReadIoSpace64** 和 **ReadIoSpace**。 从 DbgEng 扩展调用 wdbgexts.h 函数时，应始终使用以“64”结尾的函数名称。 这是因为 调试器引擎 始终在内部使用 64 位指针，而不考虑目标平台。 包含 wdbgexts.h 时，engextcpp.h 选择 64 位版本的 API。 WDbgExts API 使用的 **ExtensionApis** 全局变量在进入 EngExtCpp 方法时自动初始化，并在退出时清除。

当 EngExtCpp 扩展与远程 DbgEng 接口一起使用时，WDbgExts 接口将不可用， **并且 ExtensionApis** 结构可以归零。 如果 EngExtCpp 扩展预期在此类环境中正常运行，则应避免使用 WDbgExts API。

注意 不得尝试从任何调试器扩展调用任何 DbgHelp 或 ImageHlp 例程。 不支持调用这些例程，可能会导致各种问题。

生成 EngExtCpp 扩展

项目 • 2024/03/04

EngExtCpp 扩展库的生成方式与 DbgEng 扩展库几乎相同。 有关详细信息，请参阅生成 DbgEng 扩展。

使用 EngExtCpp 实现代码 (engextcpp.cpp)，而不是与静态库链接。

由于 EngExtCpp 扩展框架是基于 DbgEng 扩展框架生成的，因此 EngExtCpp 扩展 DLL 应导出与 DbgEng 扩展 DLL 相同的函数。

每个扩展都应该导出。 使用 **EXT_COMMAND** 宏定义扩展函数时，此宏还会创建与扩展名称相同的 C 函数。 此函数应从 DLL 导出。

以下由 engextcpp 提供的函数应从 EngExtCpp DLL 中导出。

- **DebugExtensionInitialize** -- 这样就可以调用 **Initialize** 方法来初始化库。
- **DebugExtensionUnitialize** -- 这样就可以调用 **Uninitialize** 方法来取消初始化库。
- **KnownStructOutputEx** -- 这样引擎就可以调用 *ExtKnownStructMethod* 方法来格式化已知的输出结构。
- **DebugExtensionNotify** -- 这样引擎就可以调用 **OnSessionActive**、**OnSessionInactive**、**OnSessionAccessible** 和 **OnSessionInaccessible** 方法，以通知扩展库对调试会话状态的更改。
- **help** -- 这样 EngExtCpp 扩展框架就可以自动提供 **!help** 扩展。

即使不需要这些函数提供的功能，也可以导出这些函数。 此外，如果不导出函数，则函数提供的功能将丢失。

必须导出 DebugExtensionInitialize，调试器引擎才能将 DLL 识别为有效的 DbgEng 扩展 DLL。

分析扩展参数

项目 • 2024/07/08

EngExtCpp 扩展框架提供了帮助分析传递给扩展的命令行参数的方法。 若要利用这些方法，扩展必须首先在EXT_COMMAND**宏中**声明命令行参数的格式。

若要绕过框架执行的命令行参数分析并让扩展本身分析参数，请将命令行说明设置为 `"{{custom}}"` 并使用方法 GetRawArgStr 获取用于分析的命令行参数。

打印时，将自动包装命令行说明字符串，以适应显示的列宽。 但是，换行符可以嵌入描述字符串中-使用“”`\n`来启动新行。

命令行说明可以是 NULL 或空字符串。 如果出现上述任一情况，则表示扩展命令不采用任何参数。

命令行说明

命令行参数的说明是一个序列，其中包含两种类型的组件：指令和参数。 说明可以选择包含每个指令之一，最多可以包含 64 个参数。

指令

指令指定如何分析参数。 它们由双大括号（`'{{'` 和 `'}}'`）括起来。 每个指令可以选择在描述参数的字符串中出现零次或一次。

以下指令可用：

`custom`
关闭扩展框架完成的分析，并允许该扩展执行自己的分析。

`l:str`
重写命令行参数的默认长说明。 扩展框架将使用 *str* 来完整描述所有参数。

`opt:str`
替代命名命令的默认前缀字符。 默认值为 `"/-"`，允许“”`/`或“”`-`用作标识命名参数的前缀。

`s:str`
替代命令行参数的默认简短说明。 扩展框架将针对所有参数的简短说明使用 *str* 。

下面是指令的一些示例。 以下字符串由解析其自己的参数的扩展命令使用。 它还提供用于自动 **！ help** extension 命令的简短和长说明：

```
dbgcmd
```

```
{{custom}}{{s:<arg1> <arg2>}}{{l:arg1 - Argument 1\narg2 - Argument 2}}
```

以下字符串将参数选项前缀字符更改为“”`/`或“”。 `-` 使用此指令，将使用“”和“”而不是“`+arg` `:arg`”和“`/arg`”指定`-arg`参数：

```
dbgcmd
```

```
{{opt:+:}}
```

参数

参数可以是两种类型：命名和未命名。 未命名的参数是按位置读取的。 这两种类型的参数也有显示名称，由帮助命令使用。

参数说明由单大括号（`'{'` 和 `'}'`）括起来。

每个参数说明具有以下语法：

```
dbgcmd
```

```
{[optname];[type[,flags]];[argname];[argdesc]}
```

其中：

optname
参数的名称。 这是命令和按名称提取参数的方法中使用的名称。 此名称是可选的。 如果存在该参数，则参数将变为“命名参数”;它可以显示在命令行上的任意位置，并按名称引用。 如果该参数不存在，则参数将变为“未命名参数”;它在命令行上的位置很重要，它由相对于其他未命名参数的位置引用。

type
自变量类型。 这会影响分析参数的方式及其检索方式。 *类型*参数可以具有以下值之一：

b
布尔类型。 该参数存在或不存在。 可以使用 HasArg **检索**命名的布尔参数。

e[d][s][bits]
表达式类型。 该参数具有数值。 可以使用 GetArgU64 **检索**命名表达式参数，并且可以使用 GetUnnamedArgU64 **检索**未命名的表达式参数。

d
表达式限制为参数字符串中的下一个空格字符。 如果不存在，表达式计算器将使用命令行中的字符，直到它确定它到达表达式的末尾。

s
表达式的值是有符号的。 否则，表达式的值为无符号。

位
参数值中的位数。 位*的*最大值为 64。

s
字符串类型。 字符串限制为下一个空格字符。 可以使用 GetArgStr **检索**命名字符串参数，可以使用 GetUnnamedArgStr **检索**未命名的字符串参数。

x
字符串类型。 参数是命令行的其余部分。 使用 **GetArgStr** 或 **GetUnnamedArgStr** 检索参数，就像使用 s 字符串类型一样。

flags
参数标志。 这些参数确定分析器将如何处理参数。 *标志*参数可以具有以下值之一：

d=expr
参数的默认值。 如果命令行中不存在该参数，则参数设置为 *expr*。 默认值是一个字符串，根据参数的类型进行分析。

ds
默认值不会显示在帮助提供的参数说明中。

o
参数是可选的。 这是命名参数的默认值。

r
参数是必需的。 这是未命名参数的默认值。

argname
参数的显示名称。 这是自动 **！ help** 扩展命令和自动 **/？** 或 **-？** 命令行参数使用的名称。 打印命令行选项摘要时使用。

argdesc
参数的说明。 这是由自动 **！ help** 扩展和自动"**/？** "或"-？"命令行参数打印的说明。

下面是参数说明的一些示例。 以下表达式定义一个采用单个可选表达式参数的命令。 该参数必须适合 32 位。 如果命令行中不存在该参数，将使用0x100的默认值。

```
dbgcmd
```

```
{;e32,o,d=0x100;flags;Flags to control command}
```

以下表达式定义具有可选的布尔值“/v”参数和必需的未命名字符串参数的命令。

```
dbgcmd
```

```
{v;b;;Verbose mode}{;s;name;Name of object}
```

以下表达式定义一个命令，该命令具有可选的命名表达式参数 **/oname** *expr* 和可选的命名字符串参数 **/eol** *str*。 如果 **存在 /eol** ，则其值将设置为命令行的其余部分，并且不会再分析任何参数。

```
dbgcmd
```

```
{oname;e;expr;Address of object}{eol;x;str;Commands to use}
```

命令行

下面是在命令行上分析参数的一些方法的列表：

- 命名表达式和字符串参数的值遵循命令行上的名称。 例如， **/name** *expr* 或 **/name** *str*。
- 对于命名的布尔参数，如果名称出现在命令行上，则值为 true;否则为 false。
- 可以在命令行上将多个单字符命名的布尔选项组合在一起。 例如，可以使用速记表示法“/abc”（除非已有名为“abc”的参数）来编写“/b /c”。
- 如果命令行包含命名参数“？ ” - 例如“/？ ”和“-？ ” - 参数分析结束，并显示扩展的帮助文本。

分析内部

参数分析器使用多种方法来设置参数。

SetUnnamedArg **方法**将更改未命名参数的值。 为方便起见，SetUnnamedArgStr **和** SetUnnamedArgU64 **方法**将分别设置未命名的字符串和表达式参数。

命名参数存在类似的方法。 SetArg 用于更改任何命名参数**的值，** SetArgStr 和 SetArgU64 分别用于命名字符串和表达式参数。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

类型化数据

项目 • 2023/06/15

EngExtCpp 扩展框架提供了几个类来帮助操作目标的内存。 ExtRemoteData 类描述一小部分目标内存。 如果此内存的类型已知，则它称为 *类型化数据* ，由 ExtRemoteTyped 对象描述。

可以使用 ExtRemoteList 循环访问 Windows 列表，如果列表中对象的类型已知，则 **使用** ExtRemoteTypedList。

注意 与 ExtExtension 中的客户端对象一样，这些类的实例仅在扩展库用于执行扩展命令或格式化结构以进行输出时有效。 具体而言，不应缓存它们。 有关客户端对象何时有效的详细信息，请参阅 客户端对象和引擎。 。

远程数据

应使用 **类** ExtRemoteData 处理远程数据。 此类是围绕目标内存的一小部分的包装器。 ExtRemoteData 会自动检索内存，并使用引发方法包装其他常见请求。

远程类型化数据

如果远程数据的类型已知，则应使用 ExtRemoteTyped 类对其进行处理。 此类是一个增强型远程数据对象，用于了解使用符号中的类型信息键入的数据。 它通过符号或强制转换初始化为特定对象，之后，它可以像给定类型的对象一样使用。

远程列表

若要处理远程列表，请使用 ExtRemoteList 类。 此类可用于单独链接列表或双链接列表。 如果列表是双重链接的，则假定上一个指针紧跟下一个指针。 类包含可循环访问列表并向前和向后检索节点的方法。 ExtRemoteList 还可以与以 null 结尾的列表或循环列表一起使用。

远程类型化列表

若要在已知列表中节点的类型时处理远程列表，请使用 ExtRemoteTypedList 类。 这是 ExtRemoteList 的增强版本。 除了 ExtRemoteList 的基本功能外， ExtRemoteTypedList 还自动确定类型信息的链接偏移量。

编写 WdbgExts 扩展

项目 • 2023/06/15

WdbgExts 扩展是原始类型的调试器扩展。 它们不如 DbgEng 扩展强大，但在 Microsoft Windows 上执行用户模式或内核模式调试时，它们仍然提供广泛的功能。

如果执行了 Windows 调试工具的完整安装，可以在安装目录的 sdk\samples\simplext 子目录中找到名为“simplext”的示例 WdbgExts 扩展。

本节包括：

WdbgExts 扩展设计指南

WdbgExts 扩展参考

WdbgExts 扩展设计指南

项目 • 2023/03/08

本节包括：

WdbgExts 扩展 API 概述

32 位指针和 64 位指针

使用 WdbgExts 扩展回调

使用 DECLARE_API 宏

编写 WdbgExts 扩展代码

生成 WdbgExts 扩展

WdbgExts 扩展 API 概述

项目 • 2023/06/15

每个 WdbgExts 扩展 DLL 导出一个或多个用于实现 *扩展命令的*函数。 这些函数是根据标准 C 约定命名的，但不允许使用大写字母。

函数名称和扩展命令名称相同，只是扩展命令以感叹号开头（**！** ）。 例如，将 Myextension.dll加载到调试器中，然后在“调试器命令”窗口中键入 **！ stack** 时，调试器将在 Myextension.dll 中查找名为 **stack** 的导出函数。

如果尚未加载Myextension.dll，或者其他扩展 DLL 中存在同名的其他扩展命令，则可以在“调试器命令”窗口中键入 **！ myextension.stack** 以指示扩展 DLL 和该 DLL 中的扩展命令。

每个 WdbgExts 扩展 DLL 还导出许多 *回调函数*。 加载 DLL 并使用扩展命令时，调试器会调用这些函数。

调试器引擎将在对扩展 DLL 的调用周围放置 **一个 try / except** 块。 这可以防止引擎在扩展代码中出现某些类型的 bug。 但是，由于扩展调用与引擎在同一线程中执行，因此它们仍可能导致引擎崩溃。

32 位指针和 64 位指针

项目 • 2024/02/15

WdbgExts.h 头文件支持 32 位和 64 位指针。 若要使用 64 位指针，只需按以下顺序在代码中包含以下两行：

```
C++

#define KDEXT_64BIT
#include wdbgexts.h
```

建议始终在代码中使用 64 位指针。 这样，扩展就可以在任何平台上工作，因为当目标为 32 位时，调试器会自动将 64 位指针强制转换为 32 位。

如果只想在 32 位平台上使用扩展，可以改为编写 32 位扩展。 在这种情况下，只需在代码中包含以下行：

```
C++

#include wdbgexts.h
```

有关使用 64 位指针的其他信息，请参阅使用 DECLARE_API 宏和编写 WdbgExts 扩展代码。 此外，检查作为 WDK 的一部分包含的示例代码。

使用 WdbgExts 扩展回调

项目 • 2023/06/15

编写 WdbgExts 扩展 DLL 时，可以导出某些函数：

- 必须导出名为 *WinDbgExtensionDllInit 的*函数。 当调试器加载扩展 DLL 时，它首先调用 *WinDbgExtensionDllInit* 并传递三个参数。
  - 指向 **WINDBG_EXTENSION_APIS64** 结构的指针，该结构包含指向由调试器实现并在 Wdbgexts.h 中声明的函数的指针。 必须将整个结构复制到在 DLL 中创建的全局变量。
  - 主版本号。 必须将主版本号复制到在 DLL 中创建的全局变量。
  - 次要版本号。 必须将次要版本号复制到在 DLL 中创建的全局变量。

  例如，可以创建名为 ExtensionApis、SavedMajorVersion 和 SavedMinorVersion 的全局变量，如以下示例所示。

  ```
  C++

  WINDBG_EXTENSION_APIS64 ExtensionApis;
  USHORT SavedMajorVersion;
  USHORT SavedMinorVersion;

  VOID WinDbgExtensionDllInit(PWINDBG_EXTENSION_APIS64 lpExtensionApis,
      USHORT MajorVersion, USHORT MinorVersion)
  {
     ExtensionApis = *lpExtensionApis;
     SavedMajorVersion = MajorVersion;
     SavedMinorVersion = MinorVersion;
      ...
  }
  ```

- 必须导出名为 *ExtensionApiVersion 的*函数。 调试器调用此函数，并返回指向包含扩展 DLL 版本号 **的EXT_API_VERSION** 结构的指针。 调试器在执行显示扩展版本号的 **.chain** 和 **version** 等命令时使用此版本号。

- 可以选择导出名为 *CheckVersion* 的函数。 每次使用扩展命令时，调试器都会调用此例程。 如果 DLL 的版本与调试器略有不同，但差异不大，无法阻止其运行，则可以使用此来输出版本不匹配警告。

使用 DECLARE_API 宏

项目 • 2023/06/15

WdbgExts 扩展 DLL 中的每个扩展命令都是使用 DECLARE_API 宏声明的。 此宏在 wdbgexts.h 中定义。

扩展命令的代码的基本格式为：

C++

```
DECLARE_API( myextension )
{
    code for myextension
}
```

DECLARE_API 宏为扩展命令设置标准接口。 例如，如果用户将任何参数传递给扩展命令，则整个参数字符串将存储为字符串，并且指向此字符串 (PCSTR) 的指针将作为 **参数** 传递给扩展函数。

如果使用 64 位指针，则DECLARE_API宏的定义如下：

C++

```
#define DECLARE_API(s)                              \
    CPPMOD VOID                                     \
    s(                                              \
        HANDLE                  hCurrentProcess,    \
        HANDLE                  hCurrentThread,     \
        ULONG64                 dwCurrentPc,        \
        ULONG                   dwProcessor,        \
        PCSTR                   args                \
     )
```

如果使用 32 位指针，DECLARE_API保持不变，只不过 **dwCurrentPc** 的类型为 ULONG 而不是 ULONG64。 但是，对于正在写入的任何扩展，建议使用 64 位指针。 有关详细信息，请参阅 32 位指针和 64 位指针 。

编写 WdbgExts 扩展代码

项目 • 2023/06/15

WdbgExts 扩展命令可以调用任何标准 C 函数，以及 WdbgExts.h 头文件中出现的与调试器相关的函数。

WdbgExts 函数仅用于调试器扩展命令。 它们可用于控制和检查正在调试的计算机或应用程序。 调用这些 WdbgExts 函数的任何代码都应包含 WdbgExts.h 头文件。

其中许多函数具有 32 位版本和 64 位版本。 通常，64 位 WdbgExts 函数的名称以“64”结尾，例如 **ReadIoSpace64**。 32 位版本没有数字结尾，例如 **ReadIoSpace**。 如果使用 64 位指针，则应使用以“64”结尾的函数名称;如果使用 32 位指针，则应使用“未修饰”函数名称。 建议将 64 位指针用于正在编写的任何扩展。 有关详细信息，请参阅 32 位指针和 64 位指针 。

WdbgExts 扩展不能使用出现在 DbgEng.h 头文件中的 C++ 接口。 如果要使用这些接口，应改为编写 DbgEng 扩展或 EngExtCpp 扩展。 DbgEng 扩展和 EngExtCpp 扩展都可以使用 DbgEng.h 以及 WdbgExts.h 中的所有接口。 有关详细信息，请参阅 编写 DbgEng 扩展 和 编写 EngExtCpp 扩展。

注意 不得尝试从调试器扩展调用任何 DbgHelp 或 ImageHlp 例程。 这不受支持，可能会导致各种问题。

以下主题概述了各种类别的 WdbgExts 函数：

WdbgExts 输入和输出

WdbgExts 内存访问

WdbgExts 线程和进程

WdbgExts 符号

WdbgExts 目标信息

有关这些函数的完整列表，请参阅 WdbgExts 函数。

WdbgExts 输入和输出

项目 • 2023/06/15

本主题简要概述了如何使用 WdbgExts API 执行输入和输出。 有关调试器引擎中的输入和输出流的概述，请参阅本文档调试器引擎概述部分中的输入和输出。

函数 **dprintf** 的工作方式类似于标准 C **printf** 函数，并将格式化的字符串打印到调试器命令窗口，或者更准确地说，格式化字符串将发送到 输出回调。 若要从调试器引擎读取一行输入，请使用函数 **GetInputLine**。

若要检查用户发起的中断，请使用 **CheckControlC**。 应在循环中使用此函数，以确定用户是否已尝试取消扩展的执行。

有关更强大的输入和输出 API，请参阅本文档的使用调试器引擎 API 部分中的使用输入和输出。

WdbgExts 内存访问

项目 • 2023/06/15

本主题简要概述了如何使用 WdbgExts API 执行内存访问。 有关调试器引擎中内存访问的概述，请参阅本文档的调试器引擎概述部分中的内存。

虚拟内存

可以使用 ReadMemory 函数读取目标的虚拟内存，并使用 WriteMemory 函数写入。 可以使用 ReadPointer、 ReadPtr 和 WritePointer 函数读取和写入目标内存中的指针。

若要在虚拟内存中搜索字节模式，请使用 SearchMemory 函数。

TranslateVirtualToPhysical 函数可用于将虚拟内存地址转换为物理内存地址。

Disasm 函数可用于对目标上的单个程序集指令进行反汇编。

若要在使用物理地址扩展 (PAE) 时检查 4 GB 的低内存损坏，请使用 Ioctl 操作 IG_LOWMEM_CHECK。

物理内存

只能在内核模式调试中直接访问物理内存。

可以使用 ReadPhysical 和ReadPhysicalWithFlags 函数读取目标上的物理内存，并使用 WritePhysical 和 WritePhysicalWithFlags 函数写入。

若要在物理内存中搜索指向指定范围内位置的指针，请使用 Ioctl 操作 IG_POINTER_SEARCH_PHYSICAL。

其他数据空间

在内核模式调试中，除了main内存之外，还可以将数据读取和写入各种数据空间。 可以访问以下数据空间：

Control-Space Memory
函数 ReadControlSpace、ReadControlSpace64、ReadTypedControlSpace32 和 ReadTypedControlSpace64 将从控件空间读取数据。 WriteControlSpace 函数会将数据写入控件空间。

I/O 内存
函数 ReadIoSpace、ReadIoSpace64、ReadIoSpace64、ReadIoSpaceEx64 将从系统 I/O

内存和总线 I/O 内存中读取数据。 **函数** WriteIoSpace、WriteIoSpace64、WriteIoSpaceEx 和 WriteIoSpaceEx64 会将数据写入系统 I/O 内存和总线 I/O 内存。

特定于模型的寄存器 (MSR)
函数 ReadMsr 和 WriteMsr 读取和写入 MSR。

系统总线
IoctI 操作IG_GET_BUS_DATA和IG_SET_BUS_DATA读取和写入系统总线数据。

其他信息

有关更强大的内存访问 API，请参阅本文档的使用调试器引擎 API 部分中的内存访问。

WdbgExts 线程和进程

项目 • 2023/06/15

本主题简要概述了如何使用 WdbgExts API 操作线程和进程。 有关调试器引擎中的线程和进程的概述，请参阅本文档的调试器引擎概述部分中的线程和进程。

线程

若要获取描述当前线程的 TEB) (线程环境块的地址，请使用 **GetTebAddress** 方法。 在内核模式调试中，KTHREAD 结构也可用于描述线程。 此结构由 **GetCurrentThreadAddr** (在用户模式调试中返回， **GetCurrentThreadAddr** 返回 TEB) 的地址。

线程上下文是 Windows 在切换线程时保留的状态;它由 CONTEXT 结构表示。 此结构因操作系统和平台而异，在使用 CONTEXT 结构时应小心谨慎。 线程上下文由 **GetContext** 函数返回，可以使用 **SetContext** 函数进行设置。

若要检查当前线程的堆栈跟踪，请使用 **StackTrace** 函数。 若要暂时更改用于检查堆栈跟踪的线程，请使用 **SetThreadForOperation** 或 **SetThreadForOperation64** 函数。 有关检查堆栈 的其他方法，请参阅本文档 的使用调试器引擎 API 部分中的检查堆栈跟踪。

若要获取有关目标中操作系统线程的信息，请使用 **Ioctl** 操作 **IG_GET_THREAD_OS_INFO**。

过程

若要 (描述当前进程的 PEB) 获取进程环境块的地址，请使用 **GetPebAddress** 方法。 在内核模式调试中，KPROCESS 结构也可用于描述进程。 此结构由 **GetCurrentProcessAddr** (在用户模式调试中返回， **GetCurrentProcessAddr** 返回 PEB) 的地址。

方法 **GetCurrentProcessHandle** 返回当前进程的系统句柄。

其他信息

有关更强大的线程操作和进程操作 API，请参阅本文档的使用调试器引擎 API 部分中的控制线程和进程。

WdbgExts 符号

项目 • 2023/06/15

本主题简要概述了如何使用 WdbgExts API 操作符号。 有关在调试器引擎中使用符号的概述，请参阅本文档调试器引擎概述部分中的符号。

若要计算 MASM 或 C++ 表达式，请使用 GetExpression 或 GetExpressionEx 函数。

若要读取结构中成员的值，请使用 GetFieldData 函数，或者，如果该成员包含基元值，则可以使用 GetFieldValue 。 若要确定目标内存中某个符号实例的大小，请使用 GetTypeSize 函数。

若要在 结构中查找成员的偏移量，请使用 GetFieldOffset 函数。

若要读取结构中的多个成员，请先使用 InitTypeRead 函数初始化结构。 然后，可以使用 ReadField 函数一次读取一个大小小于或等于 8 字节的成员。 对于物理内存中的结构地址，请使用 InitTypeReadPhysical 函数而不是 InitTypeRead。

有两个函数可用于循环访问链接列表。 对于使用 LIST_ENTRY32 或 LIST_ENTRY64 结构的双链接列表，可以使用 **函数** ReadListEntry 查找下一个和上一个条目。 函数 ListType 将循环访问链接列表中的所有条目，并为每个条目调用回调函数。

若要在目标内存中查找指定地址附近的符号，请使用 GetSymbol 函数。

若要从调试器引擎的缓存中删除所有符号信息，请使用 ReloadSymbols 函数。 若要读取或更改用于搜索符号文件的符号路径，请使用 GetSetSympath 函数。

调试器引擎提供的几乎所有符号操作都可以使用 Ioctl 操作 IG_DUMP_SYMBOL_INFO执行。 但是，虽然它是一个非常灵活的函数，但它是高级函数，我们建议你使用上述更简单的函数（如果适用）。

其他信息

有关功能更强大的符号 API，请参阅本文档的使用调试器引擎 API 部分中的使用符号。

WdbgExts 目标信息

项目 • 2023/06/14

若要确定目标是将 32 位指针还是 64 位指针用于内存地址，请使用函数 IsPtr64。

有关目标操作系统的信息，请使用 Ioctl 操作 IG_GET_KERNEL_VERSION。 若要获取目标上的处理器总数并找出哪个处理器是当前处理器，请使用函数 GetKdContext。

GetDebuggerData 函数返回一个KDDEBUGGER_DATA64或KDDEBUGGER_DATA32结构，其中包含调试器引擎在当前会话期间查询或确定的目标的相关信息。 此信息包括某些关键目标位置和特定状态值。

调试器缓存从目标获取的一些信息。 函数 GetDebuggerCacheSize 将返回此缓存的大小。

其他信息

有关功能更强大的目标 API，请参阅本文档的使用调试器引擎 API 部分中的目标信息。

生成 WdbgExts 扩展

项目 • 2024/03/08

所有调试器扩展都应使用 Build 实用工具进行编译和生成。 Build 实用工具包含在 Windows 驱动程序工具包 (WDK) 和早期版本的 Windows DDK 中。

请注意以下几点：

- WDK 具有多个不同的生成环境窗口。 安装 WDK 时，在**开始**菜单中都有相应的快捷方式。 若要生成调试器扩展，必须使用最新的 Windows 生成环境，无论将在哪个平台上运行扩展。
- Build 实用工具通常无法编译位于包含空格的目录路径中的代码。 扩展代码应位于完整路径不包含空格的目录中。 （具体而言，这意味着，如果将适用于 Windows 的调试工具安装到默认位置 Program Files\Debugging Tools for Windows，则将无法生成示例扩展。）

生成调试器扩展

1. 打开最新 Windows 生成环境窗口。 （可以选择“免费”版本或“检查”版本，除非已在代码中放置 **#ifdef DBG** 语句，否则它们将提供相同的结果。）
2. 设置变量 _NT_TARGET_VERSION，以指示要在其中运行扩展的最旧版本的 Windows。 _NT_TARGET_VERSION 可以设置为以下值。

展开表

值	Windows 版本
_NT_TARGET_VERSION_WIN2K	Windows 2000 及更高版本。
_NT_TARGET_VERSION_WINXP	Windows XP 及更高版本。
_NT_TARGET_VERSION_WS03	Windows Server 2003 和更高版本。
_NT_TARGET_VERSION_LONGHORN	Windows Vista 及更高版本。

如果未设置_NT_TARGET_VERSION，则扩展将仅在打开生成窗口的 Windows 版本（及更高版本）上运行。 例如，在 Sources 文件中放入以下行将生成一个将在 Windows 上运行的扩展名：`console _NT_TARGET_VERSION = $(_NT_TARGET_VERSION_WINXP)`

3. 设置 DBGSDK_INC_PATH 和 DBGSDK_LIB_PATH 环境变量，以分别指定调试器 SDK 标头和调试器 SDK 库的路径。 如果 *%debuggers%* 表示适用于 Windows 的调试工具安装的根目录，则应按如下所示设置这些变量：

控制台

```
set DBGSDK_INC_PATH=%debuggers%\sdk\inc
set DBGSDK_LIB_PATH=%debuggers%\sdk\lib
```

如果已将这些标头和库移动到其他位置，请改为指定该位置。

4. 将当前目录更改为包含扩展的 Dirs 文件或 Sources 文件的目录。

5. 运行 Build 工具：

控制台

```
build -cZMg
```

有关这些步骤的完整说明，以及如何创建 Dirs 文件和 Sources 文件的说明，请参阅 WDK 中的 Build 实用工具文档。

编写自定义分析调试器扩展

项目 • 2024/03/20

本部分中的内容

- 编写分析扩展插件以扩展 !analyze
- 分析扩展插件的元数据文件
- 故障分析项

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

编写分析扩展插件以扩展！ analyze

项目 • 2023/04/14

可以通过编写分析扩展插件来扩展 **！ analyze** 调试器命令的功能。 通过提供分析扩展插件，你可以以特定于你自己的组件或应用程序的方式参与对 bug 检查或异常的分析。

编写分析扩展插件时，还会编写一个元数据文件，用于描述要调用插件的情况。 当 **！ analyze** 运行时，它会查找、加载和运行相应的分析扩展插件。

若要编写分析扩展插件并将其提供给 **！ analyze**，请执行以下步骤。

- 创建用于导出 **_EFN_Analyze函数的** DLL。
- 创建一个与 DLL 同名且扩展名为 .alz 的元数据文件。 例如，如果 DLL 命名为 MyAnalyzer.dll，则元数据文件必须命名为 MyAnalyzer.alz。 有关如何创建元数据文件的信息，请参阅 分析扩展的元数据文件。 将元数据文件放在与 DLL 相同的目录中。
- 在调试器中，使用 **.extpath** 命令将目录添加到扩展文件路径。 例如，如果 DLL 和元数据文件位于名为 c：\MyAnalyzer 的文件夹中，请输入命令 **.extpath+ c:\MyAnalyzer**。

当 **！ analyze** 命令在调试器中运行时，分析引擎会在扩展名为 .alz 的元数据文件的扩展文件路径中查找。 分析引擎读取元数据文件以确定应加载哪些分析扩展插件。 例如，假设分析引擎正在运行以响应 bug 检查0xA IRQL_NOT_LESS_OR_EQUAL，并读取名为 MyAnalyzer.alz 的元数据文件，该文件包含以下条目。

text

```
PluginId       MyPlugin
DebuggeeClass  Kernel
BugCheckCode   0xA
BugCheckCode   0xE2
```

条目 `BugCheckCode 0x0A` 指定此插件希望参与 bug 检查分析0xA，因此分析引擎加载 MyAnalyzer.dll (，该MyAnalyzer.dll (必须与 MyAnalyzer.alz) 位于同一目录中，并调用其 **_EFN_Analyze** 函数。

注意 元数据文件的最后一行必须以换行符结尾。

框架示例

下面是一个可以用作起点的框架示例。

1. 生成一个名为 MyAnalyzer.dll 的 DLL，用于导出此处所示 **的 _EFN_Analyze** 函数。

C++

```
#include <windows.h>
#define KDEXT_64BIT
#include <wdbgexts.h>
#include <dbgeng.h>
#include <extsfns.h>

extern "C" __declspec(dllexport) HRESULT _EFN_Analyze(_In_
PDEBUG_CLIENT4 Client,
   _In_ FA_EXTENSION_PLUGIN_PHASE CallPhase, _In_
PDEBUG_FAILURE_ANALYSIS2 pAnalysis)
{
   HRESULT hr = E_FAIL;

   PDEBUG_CONTROL pControl = NULL;
   hr = Client->QueryInterface(__uuidof(IDebugControl),
(void**)&pControl);

   if(S_OK == hr && NULL != pControl)
   {
      IDebugFAEntryTags* pTags = NULL;
      pAnalysis->GetDebugFATagControl(&pTags);

      if(NULL != pTags)
      {
         if(FA_PLUGIN_INITILIZATION == CallPhase)
         {
            pControl->Output(DEBUG_OUTPUT_NORMAL, "My analyzer:
initialization\n");
         }
         else if(FA_PLUGIN_STACK_ANALYSIS == CallPhase)
         {
            pControl->Output(DEBUG_OUTPUT_NORMAL, "My analyzer: stack
analysis\n");
         }
         else if(FA_PLUGIN_PRE_BUCKETING == CallPhase)
         {
            pControl->Output(DEBUG_OUTPUT_NORMAL, "My analyzer:
prebucketing\n");
         }
         else if(FA_PLUGIN_POST_BUCKETING == CallPhase)
         {
            pControl->Output(DEBUG_OUTPUT_NORMAL, "My analyzer: post
bucketing\n");
            FA_ENTRY_TYPE entryType = pTags-
>GetType(DEBUG_FLR_BUGCHECK_CODE);
            pControl->Output(DEBUG_OUTPUT_NORMAL, "The data type for
the DEBUG_FLR_BUGCHECK_CODE tag is 0x%x.\n\n", entryType);
         }
      }

      pControl->Release();
   }
```

```
    return hr;
}
```

2. 创建包含以下条目的名为 MyAnalyzer.alz 的元数据文件。

```text
PluginId      MyPlugin
DebuggeeClass Kernel
BugCheckCode  0xE2
```

注意 元数据文件的最后一行必须以换行符结尾。

3. 在主机和目标计算机之间建立内核模式调试会话。

4. 在主计算机上，将 MyAnalyzer.dll 和 MyAnalyzer.alz 放在文件夹 c：\MyAnalyzer 中。

5. 在主计算机上的调试器中，输入以下命令。

   .extpath+ c：\MyAnalyzer

   。崩溃

6. .crash 命令在目标计算机上生成 Bug 检查0xE2 MANUALLY_INITIATED_CRASH，这会导致主计算机上的调试器中断。 bug 检查分析引擎 (主计算机上的调试器中运行，) 读取 MyAnalyzer.alz，并发现MyAnalyzer.dll能够参与分析 bug 检查 0xE2。因此，分析引擎加载MyAnalyzer.dll并调用其 _EFN_Analyze 函数。

   验证是否在调试器中看到类似于以下内容的输出。

```dbgcmd
*                        Bugcheck Analysis
*
*
*
*******************************************************************************
********

Use !analyze -v to get detailed debugging information.

BugCheck E2, {0, 0, 0, 0}

My analyzer: initialization
My analyzer: stack analysis
My analyzer: prebucketing
My analyzer: post bucketing
The data type for the DEBUG_FLR_BUGCHECK_CODE tag is 0x1.
```

前面的调试器输出显示分析引擎调用 _EFN_Analyze 函数四次：每个分析阶段调用一次。 分析引擎将 _EFN_Analyze 函数传递给两个接口指针。 *客户端* 是 IDebugClient4 接口，*pAnalysis* 是 IDebugFailureAnalysis2 接口。 前面的主干示例中的代码演示如何获取另外两个接口指针。 `Client->QueryInterface` 获取 IDebugControl 接口，并 `pAnalysis->GetDebugFATagControl` 获取 IDebugFAEntryTags 接口。

故障分析条目、标记和数据类型

分析引擎创建 DebugFailureAnalysis 对象来组织与特定代码故障相关的数据。 DebugFailureAnalysis 对象具有失败分析条目的集合 (FA 条目)，其中每个条目都由 FA_ENTRY结构表示。 分析扩展插件使用 IDebugFailureAnalysis2 接口来获取对此 FA 条目集合的访问权限。 每个 FA 条目都有一个标记，用于标识该条目包含的信息类型。 例如，FA 条目可能具有**标记**DEBUG_FLR_BUGCHECK_CODE，这会告诉我们该条目包含一个 bug 检查代码。 标记是 extsfns.h) 中定义的 DEBUG_FLR_PARAM_TYPE 枚举 (的值，也称为 FA_TAG 枚举。

```
C++

typedef enum _DEBUG_FLR_PARAM_TYPE {
    ...
    DEBUG_FLR_BUGCHECK_CODE,
    ...
    DEBUG_FLR_BUILD_VERSION_STRING,
    ...
} DEBUG_FLR_PARAM_TYPE;

typedef DEBUG_FLR_PARAM_TYPE FA_TAG;
```

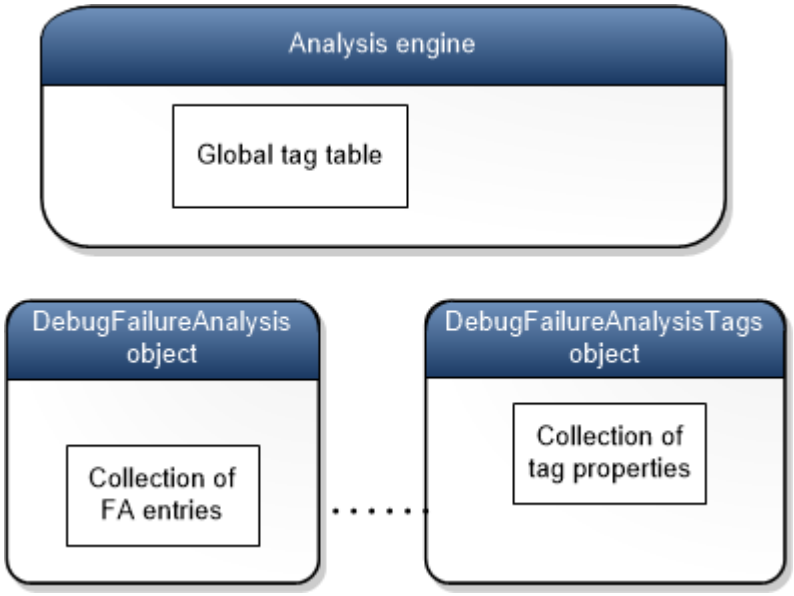
大多数 FA 条目 都有关联的数据块。 FA_ENTRY 结构的 DataSize 成员包含数据块的大小。 某些 FA 条目没有关联的数据块;所有信息都由 标记传达。 在这些情况下， DataSize 成员的值为 0。

```
C++

typedef struct _FA_ENTRY
{
    FA_TAG Tag;
    USHORT FullSize;
    USHORT DataSize;
} FA_ENTRY, *PFA_ENTRY;
```

每个标记都有一组属性：例如，名称、说明和数据类型。 DebugFailureAnalysis 对象与 DebugFailureAnalysisTags 对象相关联，该对象包含标记属性的集合。 下图说明了此关

联。



DebugFailureAnalysis 对象具有属于特定分析会话的 FA 条目的集合。 关联的 DebugFailureAnalysisTags 对象具有标记属性的集合，这些属性仅包含同一分析会话使用的标记。 如上图所示，分析引擎有一个全局标记表，其中包含有关分析会话通常可用的大量标记的有限信息。

通常，分析会话使用的大多数标记都是标准标记;也就是说，标记是 **FA_TAG** 枚举中的值。 但是，分析扩展插件可以创建自定义标记。 分析扩展插件可以将 FA 条目 添加到 **DebugFailureAnalysis** 对象，并为该条目指定自定义标记。 在这种情况下，自定义标记的属性将添加到关联的 DebugFailureAnalysisTags 对象中的标记属性集合中。

可以通过 IDebugFAEntry 标记接口访问 DebugFailureAnalysisTags 。 若要获取指向 IDebugFAEntry 接口的指针，请调用 **IDebugFailureAnalysis2** 接口的 **GetDebugFATagControl** 方法。

每个标记都有一个数据类型属性，你可以检查该属性以确定故障分析条目中数据的数据类型。 数据类型由 **FA_ENTRY_TYPE** 枚举中的值表示。

以下代码行获取 **DEBUG_FLR_BUILD_VERSION_STRING** 标记的数据类型。 在本例中，数据类型 **为DEBUG_FA_ENTRY_ANSI_STRING。** 在代码中， `pAnalysis` 是指向 **IDebugFailureAnalysis2** 接口的指针。

```
C++

IDebugFAEntryTags* pTags = pAnalysis->GetDebugFATagControl(&pTags);

if(NULL != pTags)
{
   FA_ENTRY_TYPE entryType = pTags->GetType(DEBUG_FLR_BUILD_VERSION_STRING);
}
```

如果失败分析条目没有数据块，则关联标记的数据类型 DEBUG_FA_ENTRY_NO_TYPE。

回想一下， DebugFailureAnalysis 对象具有 FA 条目的集合。 若要检查集合中的所有 FA 条目，请使用 NextEntry 方法。 以下示例演示如何循环访问整个 FA 条目集合。 假设 *pAnalysis* 是指向 IDebugFailureAnalysis2 接口的指针。 请注意，通过将 NULL 传递给 NextEntry 来获取第一个条目。

```
C++
```

```
PFA_ENTRY entry = pAnalysis->NextEntry(NULL);

while(NULL != entry)
{
   // Do something with the entry

   entry = pAnalysis->NextEntry(entry);
}
```

标记可以具有名称和说明。 在以下代码中， *pAnalysis* 是指向 IDebugFailureAnalysis 接口的指针， *pControl* 是指向 IDebugControl 接口的指针，是 `pTags` 指向 IDebugFAEntryTags 接口的 指针。 该代码演示如何使用 GetProperties 方法获取与 FA 条目关联的标记的名称和说明。

```
C++
```

```
#define MAX_NAME_LENGTH 64
#define MAX_DESCRIPTION_LENGTH 512

CHAR name[MAX_NAME_LENGTH] = {0};
ULONG nameSize = MAX_NAME_LENGTH;
CHAR desc[MAX_DESCRIPTION_LENGTH] = {0};
ULONG descSize = MAX_DESCRIPTION_LENGTH;

PFA_ENTRY pEntry = pAnalysis->NextEntry(NULL);
pTags->GetProperties(pEntry->Tag, name, &nameSize, desc, &descSize, NULL);
pControl->Output(DEBUG_OUTPUT_NORMAL, "The name is %s\n", name);
pControl->Output(DEBUG_OUTPUT_NORMAL, "The description is %s\n", desc);
```

另请参阅

编写自定义分析调试器扩展

_EFN_Analyze

分析扩展插件的元数据文件

IDebugFailureAnalysis2

分析扩展插件的元数据文件

项目 • 2023/10/26

编写分析扩展插件时，还会编写一个元数据文件，用于描述要调用插件的情况。 **当！analyze** 调试器命令运行时，它使用元数据文件来确定要加载的插件。

创建与分析扩展插件同名、扩展名为 .alz 的元数据文件。 例如，如果分析扩展插件名为 MyAnalyzer.dll，则元数据文件必须命名为 MyAnalyzer.alz。 将元数据文件放在分析扩展插件所在的同一目录中。

分析扩展插件的元数据文件是包含键值对的 ASCII 文本文件。 键和值用空格分隔。 键可以具有任何非空格字符。 键不区分大小写。

在键和以下空格之后，相应的值开始。 值可以具有以下形式之一。

- 行尾的任何字符集。 此窗体适用于不包含任何换行符的值。

  重要 如果元数据文件中的最后一个值具有此形式的值，则行必须以换行符结尾。

- 大括号 { } 之间的任意字符集。 窗体适用于包含换行符的值。

以 # 开头的行是注释，将被忽略。 注释只能从预期键的位置开始。

可以在元数据文件中使用以下键。

密钥	说明
PluginId	字符串 - 标识插件。
DebuggeeClass	字符串 - 可能的值为“Kernel”和“User”。 指示插件仅对分析内核模式故障或仅分析用户模式故障感兴趣。
BugCheckCode	32 位 bug 检查代码 - 表示插件有兴趣分析此 bug 检查代码。 单个元数据文件可以指定多个 bug 检查代码。
ExceptionCode	32 位异常代码 - 指示插件有兴趣分析此 异常代码。 单个元数据文件可以指定多个异常代码。
ExecutableName	字符串 - 指示插件仅对会话感兴趣，其中这是要分析的进程的运行可执行文件。 单个元数据文件可以指定多个可执行名称。
ImageName	字符串 - 指示插件仅对默认分析认为此图像 (dll、sys 或 exe) 出错的会话感兴趣。 分析确定哪个映像出错后，将调用该插件。 单个元数据文件可以指定多个图像名称。
MaxTagCount	Integer - 插件需要的最大自定义标记数。 自定义标记是 extsfns.h 中定义的标记以外的标记。

示例元数据文件

以下元数据文件描述了一个插件，该插件有兴趣分析 bug 检查代码0xE2。 (回想一下，最后一行必须以换行符结尾。)

```text
PluginId      MyPlugin
DebuggeeClass Kernel
BugCheckCode  0xE2
```

以下元数据文件描述了一个插件，该插件对分析 bug 检查0x8、0x9和0xA（如果 MyDriver.sys 被视为故障模块） 感兴趣。

```text
PluginId      MyPlugin
DebuggeeClass Kernel
BugCheckCode  0x8
BugCheckCode  0x9
BugCheckCode  0xA
ImageName     MyDriver.sys
```

以下元数据文件描述了一个插件，该插件对分析异常代码0xC0000005（如果 MyApp.exe 是所分析进程的运行可执行文件）感兴趣。 此外，该插件可能会创建多达三个自定义标记。

```text
PluginId        MyPlugin
DebuggeeClass   User
ExceptionCode   0xC0000005
ExecutableName  MyApp.exe
```

适用于 Windows 的调试工具有一个示例，可用于生成名为 dbgexts.dll 的调试器扩展模块。 此扩展模块实现多个调试器扩展命令，但它也可以用作分析扩展插件;也就是说，它导出 _EFN_Analyze 函数。 下面是一个元数据文件，将 dbgexts.dll 描述为分析扩展插件。

```text
PluginId         PluginSample
DebuggeeClass   User
ExceptionCode   0xc0000005
ExecutableName      cdb.exe
ExecutableName      windbg.exe
```

```
#
# Custom tag descriptions
#
TagDesc         0xA0000000  SAMPLE_PLUGIN_DEBUG_TEXT    {Sample debug help
text from plug-in analysis}
#
```

另请参阅

编写分析扩展插件以扩展 !analyze

_EFN_Analyze

！分析

故障分析项

项目 • 2023/04/14

DebugFailureAnalysis 对象具有失败分析条目的集合。 有关详细信息，请参阅 故障分析条目、标记和数据类型。

故障分析条目 (也称为 *FA 条目*) 是下列项之一：

- FA_ENTRY结构
- 后跟数据块 的FA_ENTRY 结构

FA_ENTRY 结构的 DataSize 成员包含数据块的大小（以字节为单位）。 如果没有数据块， 则 DataSize 等于 0。 FA_ENTRY 结构的 Tag 成员标识存储在 FA 条目中的信息类型。 例如，标记DEBUG_FLR_BUGCHECK_CODE指示FA_ENTRY的数据块包含一个 bug 检查代码。

在某些情况下，不需要数据块;所有信息都由 标记传达。 例如，具有标记 DEBUG_FLR_KERNEL_VERIFIER_ENABLED的FA_ENTRY没有数据块。

每个标记都与 FA_ENTRY_TYPE 枚举中的一种数据类型相关联。 例如，标记 DEBUG_FLR_BUGCHECK_CODE 与数据类型 DEBUG_FA_ENTRY_ULONG相关联。 若要确定标记的数据类型，请调用 IDebugFAEntryTags 接口的 GetType 方法。

若要获取或设置 FA 条目的数据块，请使用 IDebugFailureAnalysis2 接口。

另请参阅

编写分析扩展插件以扩展 !analyze

FA_ENTRY

开发 KDNET 传输扩展性模块

项目 • 2024/10/25

本主题介绍如何通过使用单独的硬件驱动程序扩展性模块 dll 将 KDNET 传输扩展到任何硬件上运行。 KDNET 传输扩展性模块由网卡供应商开发，用于为特定网卡添加内核调试支持。

KDNET 概述

KDNET 是一种内核调试传输工具，可通过网络对窗口进行内核调试。 最初，它用于支持以太网 NIC 的内核调试。 在设计上，硬件支持层内置于网络数据包处理和内核接口层的单独模块中。 此硬件驱动程序支持层被称为 KDNET 扩展性模块。

KDNET 是唯一可以通过使用单独的硬件驱动程序扩展性模块 dll 在任意硬件上运行的传输。 目标是通过 KDNET 和 KDNET 扩展性模块来支持 Windows 的所有调试。 所有其他内核传输 DLL（kdcom.dll、kd1394.dll、kdusb.dll 等）最终都将被弃用并从 Windows 中删除。

KDNET 用于与 KDNET 扩展性模块通信的接口有两个。 一个是基于数据包的接口，用于网卡、USB 和无线硬件；另一个是基于字节的接口，用于支持串行硬件上的 KDNET。

KDNET 扩展性模块必须符合非常严格的要求才能正常运行。 由于它们用于内核调试，因此当系统暂缓进一步执行代码时，它们将被调用和执行。 一般来说，除了通过内核调试传输与主机上运行的调试器应用程序通信的处理器外，系统中的所有处理器都被锁定在 IPI 中旋转。 该处理器通常在完全禁用中断的情况下运行，基本上是在调试传输硬件上旋转，等待来自调试器的命令。

导入和导出

KDNET 扩展性模块只有一个显式导出 – KdInitializeLibrary。 它们也没有显式导入。 KDNET 在调用 KdInitializeLibrary 时，会将一个指向结构的指针传递给 KDNET 扩展性模块，而该结构中包含 KDNET 允许它们调用的例程列表。 无法调用其他例程。 时段。 具有任何导入的 KDNET 扩展性模块设计不正确，将不受支持。

如果使用 link /dump /exports 和 link /dump /imports 转储 KDNET 扩展性模块的导入和导出，就会发现它们只有一个导出 (KdInitializeLibrary) 并且没有导入。 KDNET 扩展性模块向 KDNET 报告它们的额外导出，方法是在导出函数结构中填入函数指针，而 KDNET 会在调用 KdInitializeLibrary 时将指针传递给该结构。 然后，KDNET 使用该结构中的函数指针来调用扩展性模块，并使用模块支持的硬件进行数据传输。 KDNET 通过查看模块在结构中的导出函数表中填写了哪些特定函数来确定模块是基于数据包的模块还是基于字节的模块。 其中一些功能用于支持基于数据包的硬件，另一些则用于支持基于串行的硬

件。 表中的某些函数既适用于串行硬件，也适用于基于数据包的硬件（KdInitializeController、KdShutdownController、KdGetHardwareContextSize）。

代码设计

KDNET 扩展性模块应编写为单线程代码。 它们不应执行任何同步操作。 所有内核调试传输都依赖于 Windows 内核在调试器进入时进行适当的同步。 内核在进入内核调试器时会锁定一个调试器锁，而在进入调试器时也会通过 IPI 锁定系统中的其他处理器。 只有当主机上运行的内核调试器告诉目标计算机允许继续执行时，这些处理器才会被释放。 由于内核会进行同步，因此 KDNET 扩展性模块绝对不能在其代码中使用任何自旋锁、互斥体、门或任何其他 Windows 同步机制。 编写这些程序时，应直接对各自的硬件进行编程，以发送和接收数据包或字节。

KDNET 扩展性模块代码应尽可能简单。 这将有助于确保尽可能不出现错误，因为如果不使用硬件调试器，目前无法在计算机上实时调试 KDNET 扩展性模块代码。 不能使用内核调试器调试内核调试传输代码。 尝试这样做会导致内核堆栈损坏（通常以双重故障和重启结束）或死锁而重启计算机，或者导致重新输入传输程序，在大多数情况下会导致传输程序无法正常工作。

KDNET 扩展性模块命名约定

内核调试传输模块必须遵循 KDNET 扩展性模块的两种命名约定之一。 如果模块用于支持 PCI 硬件，则必须命名为 kd_YY_XXXX.dll，其中 XXXX 是硬件的 PCI 供应商 ID（十六进制），YY 是硬件的 PCI 类别。 在支持基于 PCI 的硬件的 Windows 框中提供了多个 KDNET 扩展性模块。 例如，Intel 的 kd_02_8086.dll、Broadcom 的 kd_02_14e4.dll 以及 Realtek 的 kd_02_10ec.dll。 可以在 https://www.pcisig.com/membership/member-companies 查找已注册的 PCI 供应商 ID。所有基于 PCI 的 KDNET 扩展性模块都将其支持的硬件的供应商 VID 用十六进制表示，作为模块名称的最后 4 个字符。 大多数盒模块的类代码为 02，因为它们是网络类设备，因此在 PCI 配置空间中的 PCI 类为 0x02。 Winload.exe 通过从 PCI 配置空间读取所选调试设备的 PCI 设备类和 PCI VID 来生成基于 PCI 的 KDNET 扩展性模块名称，并尝试加载名称中包含这些标识符的模块。 如果设备的 PCI 类别代码不是网络 0x02 类别，则必须在 KDNET 扩展性模块名称中使用设备的正确 PCI 类别代码（十六进制）。 否则，winload 将无法正确加载模块。 每个名称中的 `_02_` 是网络类设备的 PCI 类代码，以十六进制表示。 此代码还可从调试设备的 PCI 配置空间中找到并读取。

如果设备有 DBG2 表条目并且不是基于 PCI 的设备，那么模块的命名约定将有所不同。 DBG2 表调试设备的命名约定是 kd_XXXX_YYYY.dll，其中 XXXX 是十六进制 DBG2 表 PortType，YYYY 是 DBG2 表条目中的十六进制 DBG2 表 PortSubtype。 Kd_8003_5143.dll 是一个收件箱 DLL，用于支持子类型为 0x5143 的 net (0x8003) PortType。 在本例中，5143 是 Qualcomm PCI vid，因为它用于支持 Qualcomm USB 控制器上的 KDNET，而对

于 Net DBG2 表条目，PortSubtype 被定义为硬件供应商的 PCI VID。 请注意，可以使用此命名约定来支持串行、USB 和其他 DBG2 表设备。 以下是当前支持的 PortType 十六进制值：8000（串行设备）、8001（1394 设备）、8002（USB 设备）、8003（NET 设备）。 请注意，串行和 USB 设备的子类型必须在 Microsoft 中保留。 Microsoft 维护着一份已分配串行和 USB 子类型的列表。 如果现有的支持类型无法与硬件配合使用，请发送邮件至 kdnet@microsoft.com 以保留串行或 USB 子类型。

KDNET 扩展性导入

以下是可以从 KDNET 扩展性模块调用的例程列表。 请注意，所有这些例程都会传递给 KdInitializeLibrary 例程，而 kdnetextensibility.h 标头会重新映射对这些例程的正常调用，使其通过导入表。 代码必须通过导入表调用这些导入，这样模块就没有导入。 不得调用内核、HAL 或任何其他内核模块导出的任何其他例程。 只能调用这些例程。 事实证明，这组例程足以开发 KDNET 的所有盒扩展性模块，并且应该足以满足正常所需。 如果需要内核导出的其他例程未不在此列表中，请发送邮件至 kdnet@microsoft.com 解释相关情况，以及需要的其他例程和原因。 请注意，只有在主要 Windows 版本发布周期（如有）才会添加此列表。 请注意，这些例程大多都直接对应于内核或 HAL 支持的 Windows 内核 API。 有一两个是 KDNET 的专用例程。

必须在标头中正确包含 kdnetextensibility.h，以便通过导入表对例程进行正确重新映射。 如果未这样做，模块将有导入，并且将不受支持。

读取写入内存例程

应使用以下例程来读写内存映射设备内存。 这些例程具有相同的调用约定，并映射到相应的内核例程：READ_REGISTER_UCHAR、READ_REGISTER_USHORT、READ_REGISTER_ULONG、WRITE_REGISTER_UCHAR、WRITE_REGISTER_USHORT、WRITE_REGISTER_ULONG，在 64 位平台上只有 READ_REGISTER_ULONG64 和 WRITE_REGISTER_ULONG64。 所有设备内存访问都应通过这些例程完成，因为它们能确保读写操作不会被处理器重新排序。 请注意，msdn.microsoft.com 记录了在调用约定中对应于这些例程的 Windows CE Compact 2013 例程。 遗憾的是，NT 的例程似乎没有记录，但调用约定是一样的。

读取 IO 端口例程

应使用以下例程来读写设备 IO 端口。 这些例程具有相同的调用约定，并映射到相应的内核例程： READ_PORT_UCHAR、READ_PORT_USHORT、READ_PORT_ULONG、WRITE_PORT_UCHAR、WRITE_PORT_USHORT 和 WRITE_PORT_ULONG。 所有设备 IO 端口的访问都应通过这些例程来完成。 请注意，msdn.microsoft.com 记录了在调用约定中对应于这些例程的 Windows CE Compact 2013 例程。

其他例程

可以调用下列其他例程，并应使用指定的参数正常调用。 请注意，这样做虽然正确地包含了 kdnetextensibility.h 标头，但会通过 KDNET 扩展性导入表重新映射函数调用，从而导致模块中没有 KDNET 扩展性模块所需的显式导入。

C++

```
PHYSICAL_ADDRESS

KdGetPhysicalAddress (

    __in PVOID Va

    );


VOID

KeStallExecutionProcessor (

    __in ULONG Microseconds

    );


ULONG

KdGetPciDataByOffset (

    __in ULONG BusNumber,

    __in ULONG SlotNumber,

    __out_bcount(Length) PVOID Buffer,

    __in ULONG Offset,

    __in ULONG Length

    );


ULONG

KdSetPciDataByOffset (

    __in ULONG BusNumber,

    __in ULONG SlotNumber,

    __in_bcount(Length) PVOID Buffer,
```

```
    __in ULONG Offset,
    __in ULONG Length
    );

VOID
KdSetDebuggerNotPresent (
    __in BOOLEAN NotPresent
    );

VOID
PoSetHiberRange (
    _In_opt_ PVOID MemoryMap,
    _In_ ULONG     Flags,
    _In_ PVOID     Address,
    _In_ ULONG_PTR Length,
    _In_ ULONG     Tag
    );


VOID
KeBugCheckEx (
    __in ULONG BugCheckCode,
    __in ULONG_PTR BugCheckParameter1,
    __in ULONG_PTR BugCheckParameter2,
    __in ULONG_PTR BugCheckParameter3,
    __in ULONG_PTR BugCheckParameter4
    );

PVOID
KdMapPhysicalMemory64 (
```

```
    _In_ PHYSICAL_ADDRESS PhysicalAddress,

    _In_ ULONG NumberPages,

    _In_ BOOLEAN FlushCurrentTLB

    );


VOID

KdUnmapVirtualAddress (

    _In_ PVOID VirtualAddress,

    _In_ ULONG NumberPages,

    _In_ BOOLEAN FlushCurrentTLB

    );


ULONG64

KdReadCycleCounter (

    __out_opt PULONG64 Frequency

    );
```

请注意，PoSetHiberRange 函数只能从 KdSetHibernateRange 例程中调用。 此外，大多数 KDNET 扩展性模块应该不需要调用 KeBugCheckEx、KdMapPhysicalMemory64 和 KdUnmapVirtualAddress。 另一方面，基本上所有 KDNET 扩展性模块都需要调用 KdGetPhysicalAddress 来获取设备 DMA 引擎编程所需的物理内存地址，而且许多模块还需要调用 KeStallExecutionProcessor、KdGetPciDataByOffset 和 KdSetPciDataByOffset。 最后两个例程用于访问设备 PCI 配置空间。

KDNET 扩展性导出

下面简要介绍了 KDNET 的每个扩展例程。 必须实现基于数据包的 KDNET 扩展性模块或基于串行的 KDNET 扩展性模块所需的所有例程。 以下是 KDNET 扩展性模块导出的数据包。

KdInitializeLibrary

```
C++
```

```
/*++

Routine Description:

    This routine validates that the ImportTable is a supported version.
Makes
    a copy of the ImportTable in its own global memory, and writes pointers
to
    functions that it exports into the Exports pointer also located in that
    table.

    This routine also writes the size in bytes of the Memory it needs into
    the Length field of the Memory structure contained in the debug device
    descriptor passed to this routine.

    When kernel debugging is enabled, this routine will be called twice
during
    boot.  The first time by winload to determine how much memory to
allocate
    for KDNET and its extensibility module, and the second time by KDNET
when
    the kernel first initializes the kernel debugging subsystem.

Arguments:

    ImportTable - Supplies a pointer to the KDNET_EXTENSIBILITY_IMPORT
        structure.

    LoaderOptions - Supplies a pointer to the LoaderOptions passed to the
        kernel.  This allows settings to be passed to the KDNET
extensibility
        module using the loadoptions BCD setting.

    Device - Supplies a pointer to the debug device descriptor.

Return Value:

    STATUS_INVALID_PARAMETER if the version of the import or export table is
        incorrect.

    STATUS_SUCCESS if initialization succeeds.

--*/
NTSTATUS
KdInitializeLibrary (
    __in PKDNET_EXTENSIBILITY_IMPORTS ImportTable,
    __in_opt PCHAR LoaderOptions,
    __inout PDEBUG_DEVICE_DESCRIPTOR Device
    )
{
    NTSTATUS Status;
    PKDNET_EXTENSIBILITY_EXPORTS Exports;

    __security_init_cookie();
```

```
    Status = STATUS_SUCCESS;
    KdNetExtensibilityImports = ImportTable;
    if ((KdNetExtensibilityImports == NULL) ||
        (KdNetExtensibilityImports->FunctionCount != KDNET_EXT_IMPORTS)) {

        Status = STATUS_INVALID_PARAMETER;
        goto KdInitializeLibraryEnd;
    }

    Exports = KdNetExtensibilityImports->Exports;
    if ((Exports == NULL) || (Exports->FunctionCount != KDNET_EXT_EXPORTS))
{
        Status = STATUS_INVALID_PARAMETER;
        goto KdInitializeLibraryEnd;
    }

    //
    // Return the function pointers this KDNET extensibility module exports.
    //

    Exports->KdInitializeController = KdInitializeController;
    Exports->KdShutdownController = KdShutdownController;
    Exports->KdSetHibernateRange = KdSetHibernateRange;
    Exports->KdGetRxPacket = KdGetRxPacket;
    Exports->KdReleaseRxPacket = KdReleaseRxPacket;
    Exports->KdGetTxPacket = KdGetTxPacket;
    Exports->KdSendTxPacket = KdSendTxPacket;
    Exports->KdGetPacketAddress = KdGetPacketAddress;
    Exports->KdGetPacketLength = KdGetPacketLength;
    Exports->KdGetHardwareContextSize = KdGetHardwareContextSize;

    //
    // Return the hardware context size required to support this device.
    //

    Status = ContosoInitializeLibrary(LoaderOptions, Device);

KdInitializeLibraryEnd:
    return Status;
}
```

调用此例程是为了在 KDNET 和此 KDNET 扩展性模块之间传递导入和导出例程。 此例程应验证导入表和导出表的版本是否符合预期并受支持，如果不符合则会失败。 它应该在自己的全局内存中复制一份导入表。 它应将导出的例程写入导入表中“Exports”字段指向的结构中。 它还必须将传给此例程的调试设备描述符指针中的内存结构“Length”字段设置为支持硬件设备所需的内存字节数。

验证导入导出计数

例如，代码必须检查 Imports FunctionCount 是否与 OS 中可用的功能匹配，`(KdNetExtensibilityImports->FunctionCount != KDNET_EXT_IMPORTS)` 如果计数不匹配，则返回 `STATUS_INVALID_PARAMETER` 。

检查计数可确保正在运行的 Windows OS KDNET 版本（例如启动管理器/OS 加载程序/虚拟机监控程序/安全内核/NT OS）与用于生成当前 KDNET 扩展性模块的 WDK 版本之间的兼容性。 WDK 和 OS 版本必须同步;否则，如果导入/导出计数器值发生更改，上述检查将失败。 例如，如果 KDNET 扩展性接口添加了新功能函数，则计数将不再匹配。 若要确保它们匹配，请始终使用与托管 KDNET 版本的 OS 匹配的 WDK 版本。

自定义所需的内存

请注意，设备将使用为调试器选择的硬件。 如果需要，此例程应根据设备自定义所需的内存量。 例如，同时支持 1Gig 和 10Gig 硬件的扩展性模块可以增加为 10Gig 设备请求的内存大小。 它们可以通过检查调试设备描述符的 DeviceID 字段来确定正在使用的设备。

KdInitializeLibrary 是唯一的导出

请注意，winload 和 KDNET 在调用 KdInitSystem 时都会调用此例程。 请注意，这是 KDNET 扩展性模块导出的唯一例程。 它是放置在 .def 文件中的唯一例程。 KDNET 扩展性模块只有一个显式导出（本例程）而没有导入。

KdSetHibernateRange

C++

```
VOID

KdSetHibernateRange (

    VOID

    )

/*++
Routine Description:

    This routine is called to mark the code in the KDNET extensiblity module
    so that it can be properly handled during hibernate and resume from
    hibernate.

Arguments:
    None.
```

```
Return Value:
    None.
--*/
```

系统会在休眠前调用此例程，以便向系统正确注册 KDNET 扩展性模块使用的代码。 这样，系统就能在休眠和从休眠状态恢复时正确管理内存。 （内存的保存时间较晚，加载时间较早，因为它在恢复过程中很早就会被调用。）

KdInitializeController

C++

```
NTSTATUS

KdInitializeController(

    __in PVOID Adapter

    )

/*++
Routine Description:

    This function initializes the Network controller.  The controller is
setup
    to send and recieve packets at the fastest rate supported by the
hardware
    link.  Packet send and receive will be functional on successful exit of
    this routine.  The controller will be initialized with Interrupts masked
    since all debug devices must operate without interrupt support.

Arguments:

    Adapter - Supplies a pointer to the debug adapter object.

Return Value:
    STATUS_SUCCESS on successful initialization.  Appropriate failure code
if
    initialization fails.
--*/
```

调用此例程是为了初始化硬件。 当系统初始化时，以及当系统从低功耗状态唤醒时，都会调用 KdShutdownController。 此例程必须确保硬件已完全完成初始化，并在返回前做好发送数据包的准备。 此例程应等待 PHY 启动并建立链接。 请注意，如果没有连接电缆，则此例程不应无限期停止。 此例程在 KDNET 与此扩展性模块共享的 KDNET 共享数

据结构中设置链路速度和双工。 它还会将硬件使用的 MAC 地址写入 KDNET 共享数据结构中 TargetMacAddress 指向的位置。

KdShutdownController

```
C++

VOID

KdShutdownController (

    __in PVOID Adapter

    )

/*++
Routine Description:

    This function shuts down the Network controller.  No further packets can
    be sent or received until the controller is reinitialized.

Arguments:

    Adapter - Supplies a pointer to the debug adapter object.

Return Value:

    None.
--*/
```

至关重要的是，此个例程必须等待，直到所有待处理的发送数据包都在线路上发送完毕。 此例程需要等到所有发送数据包都已从主存储器中 DMA 出来并传输到线路上之后，才会关闭硬件上的发送功能。 一旦发送完所有待处理的发送数据包，此例程就会完全关闭硬件。 当系统关闭时，以及当系统决定将调试传输器电源管理到低功耗状态时，都会调用此例程。 除了在系统关闭时调用外，在系统进入待机、休眠、睡眠和连接待机时也可以调用。

KdGetHardwareContextSize

```
C++

ULONG

KdGetHardwareContextSize (

    __in PDEBUG_DEVICE_DESCRIPTOR Device

    )
```

```
/*++
Routine Description:

    This function returns the required size of the hardware context in
bytes.

Arguments:

    Device - Supplies a pointer to the debug device descriptor.

Return Value:

    None.

--*/
```

此例程将返回支持硬件所需的全部内存的字节数。 这包括上下文结构、所有用于接收和发送的数据包缓冲区，以及任何硬件数据包描述符和其他结构。 需要在此处报告所需的所有内存的大小。 包括因硬件对数据包、数据包描述符或其他结构的对齐限制而需要的额外内存。

请注意，KdInitializeLibrary 例程在设置调试设备描述符中的内存长度字段时，应调用此例程。

KdGetRxPacket

C++

```
NTSTATUS

KdGetRxPacket (

    __in PVOID Adapter,

    __out PULONG Handle,

    __out PVOID *Packet,

    __out PULONG Length

)

/*++

Routine Description:

    This function returns the next available received packet to the caller.
```

```
Arguments:

    Adapter - Supplies a pointer to the debug adapter object.

    Handle - Supplies a pointer to the handle for this packet.  This handle
        will be used to release the resources associated with this packet
back
        to the hardware.

    Packet - Supplies a pointer that will be written with the address of the
        start of the packet.

    Length - Supplies a pointer that will be written with the length of the
        received packet.

Return Value:

    STATUS_SUCCESS when a packet has been received.
    STATUS_IO_TIMEOUT otherwise.

--*/
```

此例程获取下一个已接收但尚未处理的可用数据包。 它会返回该数据包的句柄。 该句柄将用于调用 KdGetPacketAddress 获取数据包的地址，以及调用 KdGetPacketLength 获取数据包长度。 在调用 KdReleaseRxPacket 释放数据包之前，数据包和句柄必须保持可用和有效。 此例程还直接向调用方返回数据包地址和长度。

如果当前没有可用的数据包，则此例程必须立即返回 STATUS_IO_TIMEOUT。 此例程不得等待接收数据包。 请注意，句柄的前 2 位是保留位。 TRANSMIT_HANDLE 和 TRANSMIT_ASYNC 都必须保持明确。

KdReleaseRxPacket

C++

```
VOID

KdReleaseRxPacket (

    __in PVOID Adapter,

    ULONG Handle

)

/*++

Routine Description:

    This function reclaims the hardware resources used for the packet
```

```
    associated with the passed Handle.  It reprograms the hardware to use
those
    resources to receive another packet.

Arguments:

    Adapter - Supplies a pointer to the debug adapter object.
    Handle - Supplies the handle of the packet whose resources should be
        reclaimed to receive another packet.

Return Value:

    None.

--*/
```

此例程将与数据包句柄相关的资源释放回硬件，以便用于接收另一个数据包。 每次成功调用 KdGetRxPacket 后，都会使用 KdGetRxPacket 返回的句柄再次调用 KdReleaseRxPacket。 请注意，不保证在 KdGetRxPacket 成功后会立即调用 KdReleaseRxPacket。 可能会先调用另一个 KdGetRxPacket。 不过，每次成功的 KdGetRxPacket 调用都会通过 KdReleaseRxPacket 调用释放其资源。

此例程应正确对硬件进行编程，以便释放的资源可用于接收另一个数据包。

KdGetTxPacket

C++

```
NTSTATUS

KdGetTxPacket (

    __in PVOID Adapter,

    __out PULONG Handle

)

/*++

Routine Description:

    This function acquires the hardware resources needed to send a packet
and
    returns a handle to those resources.

Arguments:

    Adapter - Supplies a pointer to the debug adapter object.
```

```
    Handle - Supplies a pointer to the handle for the packet for which
hardware
        resources have been reserved.

Return Value:

    STATUS_SUCCESS when hardware resources have been successfully reserved.
    STATUS_IO_TIMEOUT if the hardware resources could not be reserved.
    STATUS_INVALID_PARAMETER if an invalid Handle pointer or Adapter is
passed.

--*/
```

此例程获取下一个可用的传输资源，并返回一个句柄。 此句柄将用于调用 KdGetPacketAddress 和 KdGetPacketLength。 KdGetPacketAddress 返回的数据包地址将用于直接写入数据包的内容。 数据包地址必须是数据包的起始地址，长度应是可写入数据包的最大字节数。 请注意，如果没有可用的硬件资源，因为它们都已被获取但尚未被传输，则此例程应立即返回 STATUS_IO_TIMEOUT。

必须在返回的句柄中设置 TRANSMIT_HANDLE。 请注意，句柄的前两位是为 TRANSMIT_ASYNC 和 TRANSMIT_HANDLE 标志保留的。

KdSendTxPacket

C++

```
NTSTATUS

KdSendTxPacket (

    __in PVOID Adapter,

    ULONG Handle,

    ULONG Length

)

/*++

Routine Description:

    This function sends the packet associated with the passed Handle out to
the
    network.  It does not return until the packet has been sent.

Arguments:

    Adapter - Supplies a pointer to the debug adapter object.
```

```
    Handle - Supplies the handle of the packet to send.

    Length - Supplies the length of the packet to send.

Return Value:

    STATUS_SUCCESS when a packet has been successfully sent.

    STATUS_IO_TIMEOUT if the packet could not be sent within 100ms.

    STATUS_INVALID_PARAMETER if an invalid Handle or Adapter is passed.

--*/
```

此例程将与已传递句柄相关的数据包发送到线路上。 请注意，句柄中可能有一个额外的设置位，用于指示发送是否为异步传输。 如果在句柄中设置了 TRANSMIT_ASYNC 标志，则该例程应编程让硬件发送数据包，然后立即返回，而无需等待硬件完成发送。 这意味着传输过程中出现的任何错误都将丢失。 这没有问题，而且是设计好的，因为数据包无论如何都可能在线路上丢失。 如果句柄中未设置 TRANSMIT_ASYNC 标志，则此例程必须等待数据包在线路上发送完毕，并返回发送过程中出现的任何错误（如有）。 请注意，在向调试器主机发送转储文件或通过 KDNET 从 KDNIC 发送 Windows 网络数据包时，将设置 TRANSMIT_ASYNC。 在发送所有其他调试器数据包时，TRANSMIT_ASYNC 将被明确。

如果一组数据包的 TRANSMIT_ASYNC 设置为 TRUE，随后又发送了一个未设置 TRANSMIT_ASYNC 的数据包，那么硬件必须等到未设置标记的数据包实际发送完毕，即使这意味着它必须等待前面的异步数据包也发送完毕。

KdGetPacketAddress

C++

```
PVOID

KdGetPacketAddress (

    __in PVOID Adapter,

    ULONG Handle

)

/*++

Routine Description:

    This function returns a pointer to the first byte of a packet associated
    with the passed handle.
```

```
Arguments:

    Adapter - Supplies a pointer to the debug adapter object.

    Handle - Supplies a handle to the packet for which to return the
        starting address.

Return Value:

    Pointer to the first byte of the packet.

--*/
```

此例程返回一个指针，指向与传递的句柄相关联的数据包的第一个字节。 请注意，句柄的 TRANSMIT_HANDLE 位将用于发送数据包，而 TRANSMIT_HANDLE 位将用于接收数据包。 返回的指针应该是一个 Windows 虚拟地址，可以被处理器读取或写入。 此地址应位于为 KDNET 扩展性模块预留的内存块内，而该内存块会在调试设备描述符内存结构中传递。 （请注意，KDNET 扩展性模块在访问内存时，绝对不能使用超过它在 KdInitializeLibrary 中请求的内存大小。块末尾的任何额外内存都将保留给 KDNET 使用，KDNET 扩展性模块不得触及。）

KdGetPacketLength

C++

```
ULONG

KdGetPacketLength (

    __in PVOID Adapter,

    ULONG Handle

)

/*++

Routine Description:

    This function returns the length of the packet associated with the
passed
    handle.

Arguments:

    Adapter - Supplies a pointer to the debug adapter object.

    Handle - Supplies a handle to the packet for which to return the
        length.
```

```
Return Value:

    The length of the packet.

--*/
```

此例程以字节为单位返回与传递的句柄相关的数据包长度。 请注意，句柄的 TRANSMIT_HANDLE 位将用于发送数据包，而 TRANSMIT_HANDLE 位将用于接收数据包。 对于发送数据包，此长度应为可写入数据包的最大字节数。 对于接收数据包，此长度应为接收数据包的实际字节数。

调试 KDNET 扩展性模块

要调试 KDNET 扩展性模块，需要在目标计算机上的高级命令提示符下运行以下 bcdedit 命令。

首先，也是最重要的一点 - 需要运行以下两条命令，以确保 Winload 允许重复启动失败，而不会进入特殊的失败路径，即进入调试器并阻止正常启动。 在运行这些命令后，就可以用新的位反复重启机器，并顺利调试这些新位。

```
Bcdedit -set {current} BootStatusPolicy IgnoreAllFailures
```

```
Bcdedit -set {current} RecoveryEnabled No
```

假设要使用目标计算机 com1 上的串行调试功能来调试扩展模块，请执行以下操作。

```
bcdedit -dbgsettings serial debugport:1 baudrate:115200
```

将默认调试传输设置为 com1 上的串行端口，波特率为 115200。 这些设置也将用于启动调试。

```
bcdedit -debug on
```

这样就可以进行内核调试。

```
bcdedit -bootdebug on
```

这样就可以在 winload.exe 上进行启动调试，可以用它来调试包括 KDNET 扩展模块在内的早期内核初始化。

```
bcdedit -set kerneldebugtype net
```

这样将强制内核调试类型为 net，与默认调试传输设置无关。 这将导致 winload.exe 将 kdnet.dll 作为内核调试传输载入。

```
bcdedit -set kernelbusparams b.d.f
```

其中 b 是总线号，d 是设备号，f 是正在为其编写 KDNET 扩展模块的硬件的函数号（全部以十进制为单位）。 这些数字取决于硬件所在的 PCI 插槽。 可以在 Windows 设备管理器的网络设备属性页面中找到位置字符串。 打开 Windows 设备管理器，双击网络设备，找到设备并双击它，在打开的窗口中应该有一个位置：其中包含 PCI 总线上硬件的总线、设备和功能的字段。 如果总线驱动程序导致该信息被屏蔽，则必须通过总线或其他方法来确定位置。

这样将强制将内核总线参数设置为 b.d.f，从而强制选择特定设备作为内核调试设备。

```
bcdedit -set kernelhostip N
```

其中 N 由以下公式确定。 如果调试器主机的 IPv4 地址为 w.x.y.z，则 N = (w*0x01000000*) + (*x*0x00010000) + (y*0x00000100*) + (*z*0x00000001)。 N 需要在命令行中指定为十进制，而不是十六进制。 实际上，需要将 IPv4 地址的每个字节连接起来（十六进制），以十六进制建立一个 32 位数字，然后将其转换为十进制。

```
bcdedit -set kernelport N
```

其中 N 为 50000 或其他不会在内部网络中被屏蔽的端口。

这样将强制 KDNET 使用端口 N 作为网络调试端口。

```
bcdedit -set kernelkey 1.2.3.4
```

这样将把 KDNET 调试密钥强制设为 1.2.3.4。 1.2.3.4 不是安全或唯一的网络密钥。 为了使目标计算机保持安全，在主机和目标计算机之间传输的数据包必须加密。 强烈建议使用自动生成的加密密钥。 有关详细信息，请参阅设置 KDNET 网络内核自动调试。

```
bcdedit -set kerneldhcp on
```

这将强制 KDNET 内核 dhcp 设置为开启。

假设在主机上使用 com1 作为串行调试端口，请在调试器主机上使用以下命令行运行调试器：

```
windbg -d -k com:port=com1,baud=115200
```

这样将运行调试器，并导致它在 windbg 启动调试器首次与主机通信时中断。

然后运行以下命令来重新启动目标计算机

```
shutdown -r -t 0
```

当调试器切换到 windbg 时，确保为 winload 加载了符号。 （可能需要设置 .sympath 并进行 .reload）。 然后运行 `x winload!*deb*tra*`。 列出的其中一个符号将类似于 BdDebugTransitions。

然后运行 `ed winload!BdDebugTransitions 1`，但要确保使用正确的符号名称。

然后运行 `bu winload!blbdstop` 以设置断点。

然后点击 `g` 开始。

应该在 winload!BlBdStop 处中断。

然后运行以下命令。

```
bu nt!KdInitSystem
```

```
bu kdnet!KdInitialize
```

```
bu kdstub!KdInitializeLibrary
```

请注意，在 KDNET 扩展模块中设置断点时，很可能会使用 kdstub；如果不起作用，请使用

```
bu kd_YY_XXXX!KdInitializeLibrary
```

其中，YY 是 PCI 类别，XXXX 是 PCI VID。 （即：使用 KDNET 扩展性模块的名称。）

通常在调试器中需要使用 kdstub，而不是扩展性模块的实际名称。

然后运行 `bl` 以列出断点。 确保断点已就位（它们旁边都应该有一个 e）。

然后点击 `g`。 应该点击 nt!KdInitSystem 断点。

再次点击 `g`，就会看到 kdnet!KdInitialize

再次点击 `g`，应该会在自己模块的 KdInitializeLibrary 处遇到一个断点。

然后就可以在 InitializeController 例程和所有其他例程上设置断点，并逐步完成代码。

一旦单步执行 KdInitializeLibrary，点击 g，如果在 InitializeController 例程上设置了断点，则下一个断点就会出现。

完成后，确保在 KdGetTxPacket、KdSendTxPacket、KdGetRxPacket 和 KdReleaseRxPacket 例程上设置了断点，然后再次点击 g，这些例程将在 KDNET 启动时作为网络初始化的一部分运行。

可能需要在 KdInitializeLibrary 或 KdInitializeController 例程中添加临时代码，以确保所有例程都被调用，这样就可以逐步完成所有代码。 （例如，KdShutdownController 在启动时不会被正常调用，因此需要从临时代码中明确调用它，这样就可以逐步检查并确保其正确性。）

在单步执行所有代码并确信无误后，重新启动目标计算机，但不要将 winload!BdDebugTransitions 标记设置为 true（默认为零）。

然后在调试器主计算机上运行另一个内核调试器实例。

```
Windbg -d -k net:port=50000,key=1.2.3.4
```

启动目标计算机，它就会通过网络连接到内核调试器的另一个实例。

然后在内核调试器中运行命令，确保它能正常工作，然后让目标继续启动，并确保你以后可以中断并运行命令。

> **备注**
>
> 在 winload 中设置调试转换标志可确保 Windows 不会启动。 如果在设置该标志后试图让 Windows 完成启动，Windows 就会直接崩溃或挂起。 如果希望 Windows 成功启动，就不能设置调试转换标志。 设置该标志后，就可以在调试器中调试代码，并通过单步执行来验证代码是否正确，但最终还是不需要该标志，以便在正常启动时验证调试是否正常。 这意味着，在正常启动系统时，无法单步执行的代码；而事实上，当 Windows 正常运行时，在硬件上启用了调试功能，KDNET 扩展模块是无法调试的。 任何使用内核调试器对其进行调试的尝试都会导致计算机崩溃。 （不能在内核调试路径中运行的代码中设置断点，否则会导致无限重新进入、堆栈损坏和重新启动。）

多个物理函数 - 2PF

除了 KDNET 的扩展性外，KDNET 还支持通过分割 PCI 配置空间，在支持的网卡上使用多个物理功能 (PF) 进行内核调试。 鼓励网络卡供应商启用对此功能的支持。 有关详细信息，请参阅调试器 2PF KDNET 微型端口网络驱动程序支持。

另请参阅

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

调试器 2PF KDNET 微型端口网络驱动程序支持

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

KDSerial 扩展性代码示例

项目 • 2024/05/27

Windows 调试工具支持扩展 KDNET 串行传输。 KDNET 传输扩展性模块由硬件供应商开发，用于为尚未支持的特定硬件添加内核调试支持。

一般而言，用于调试的串行连接速度较慢，因此在可能的情况下，最好通过以太网卡使用 KDNET。

KDSerial 扩展性代码示例

以下三个代码示例位于 WDK 安装目录中，显示了 KDNET Serial 和 KDSerial 传输扩展的使用情况。

KDNET 串行 16550 示例

KDNET 串行接口显示了 16550 串行接口硬件的使用情况。 它位于此目录中。

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\ddk\samples\kdnet\serial\16550
```

在测试 KDNET 串行 16550 示例代码时，串行电缆应按 NULL-MODEM RS232 电缆布线，其中 Tx-Rx 线交叉连接。

KDNET 串行 siig 示例

siig 示例使用 RS-232 硬件握手来实现基于串行的 KDNET。 它位于此目录中。

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\ddk\samples\kdnet\serial\siig
```

测试 KDNET 串行 siig 代码样本时，串行电缆应作为直通 RS-232 电缆进行接线。

KDSerial 示例

这表明使用了较早的 KDSerial 传输。 首选串行 KDNET 传输。

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\ddk\samples\kdserial
```

在测试 KDSerial 代码示例时，串行电缆应按 NULL-MODEM RS232 电缆布线，其中 Tx-Rx 线交叉连接。

部署测试串行内核调试传输的步骤

1. 加载和生成项目（输出 kdserial.dll）
2. 将 kdserial.dll 重命名为 kdcom.dll
3. 将 kdcom.dll 复制到目标计算机上的 `C:\windows\system32\kdcom.dll`（建议在覆盖原始 kdcom.dll 之前创建其备份）
4. 在目标上运行 `bcdedit /dbgsettings serial debugport:<port> baudrate:<rate>`
5. 在目标上运行 `bcdedit /debug on`
6. 在目标上运行 `bcdedit /bootdebug on`
7. 在目标上运行 `bcdedit /testsigning yes`
8. 重启目标 & 连接到启动调试器。 要在缺少有效 Microsoft 签名的情况下加载新的 kdcom.dll，则必须执行此步骤。 如果在加载 kdcom.dll 时未启用已连接的启动调试器 &，则 winload 将无法加载它。

注意：自定义 kdcom.dll 将不用于启动调试器传输

9. 继续通过 winload & 经由自定义 kdcom.dll 传输连接到内核调试器。

另请参阅

如何开发 KDNET 扩展性模块

自动设置 KDNET 网络内核调试

手动设置 KDNET 网络内核调试

手动设置内核模式调试

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

调试器编程参考

项目 • 2023/03/08

本主题介绍编程参考主题。

此参考部分包括：

调试器引擎参考

EngExtCpp 扩展参考

WdbgExts 扩展参考

自定义分析调试器扩展参考

调试引擎接口

项目 • 2023/03/08

本部分介绍以下接口：

- IDebugAdvanced
- IDebugAdvanced2
- IDebugAdvanced3
- IDebugAdvanced4
- IDebugClient
- IDebugClient2
- IDebugClient3
- IDebugClient4
- IDebugClient5
- IDebugClient6
- IDebugClient7
- IDebugControl
- IDebugControl2
- IDebugControl3
- IDebugControl4
- IDebugControl5
- IDebugControl6
- IDebugControl7
- IDebugDataSpaces
- IDebugDataSpaces2
- IDebugDataSpaces3

- IDebugDataSpaces4
- IDebugRegisters
- IDebugRegisters2
- IDebugSymbols
- IDebugSymbols2
- IDebugSymbols3
- IDebugSymbols4
- IDebugSymbols5
- IDebugSystemObjects
- IDebugSystemObjects2
- IDebugSystemObjects3
- IDebugSystemObjects4

调试器引擎参考

项目 • 2023/03/08

本节包括：

客户端函数

调试引擎接口

回调调试引擎接口

其他调试引擎接口

结构和常量

Microsoft Windows 上的所有 COM 接口都支持某些常见方法。 本参考部分未单独列出这些方法。 有关详细信息，请参阅 使用客户端对象。

DebugBaseEventCallbacks 类

项目 • 2024/02/07

DebugBaseEventCallbacks 类提供 IDebugEventCallbacks 接口的基本实现。

程序可以从 DebugBaseEventCallbacks 派生事件回调类，并仅实现所需的方法。

请小心适当地实现 GetInterestMask。

要求

标头

Dbgeng.h (包括 Dbgeng.h)

另请参阅

DebugBaseEventCallbacksWide

DebugBaseEventCallbacksWide 类

项目 • 2023/06/15

DebugBaseEventCallbacksWide 类提供 IDebugEventCallbacksWide 接口的基本实现。

程序可以从 DebugBaseEventCallbacksWide 派生事件回调类，并仅实现所需的方法。

请小心适当地实现 GetInterestMask。

要求

标头

Dbgeng.h (包括 Dbgeng.h)

另请参阅

DebugBaseEventCallbacks

调试 API 常量

项目 • 2023/03/08

DBG_ASMOPT_XXX

DBG_ATTACH_XXX

DBG_DUMP_XXX

DBG_DUMP_FIELD_XXX

DBG_ENGOPT_XXX

DBG_EVENT_XXX

DEBUG_FILTER_XXX

DEBUG_FORMAT_XXX

DEBUG_OUTCTL_XXX

DEBUG_OUTPUT_XXX

DEBUG_PROCESS_XXX

DEBUG_STATUS_XXX

DEBUG_SYMBOL_XXX

DEBUG_TYPEOPTS_XXX

DEBUG_ASMOPT_XXX

项目 • 2024/02/07

程序集和反汇编选项会影响调试器引擎如何为目标组装和反汇编处理器指令。

选项由具有以下位标志的位集表示。

展开表

一直	说明
DEBUG_ASMOPT_VERBOSE	设置此位后，反汇编中包含其他信息。 这等效于 .asm 命令中的 verbose 选项。
DEBUG_ASMOPT_NO_CODE_BYTES	设置此位后，指令的原始字节不包括在反汇编中。 这等效于 .asm 命令中的 no_code_bytes 选项。
DEBUG_ASMOPT_IGNORE_OUTPUT_WIDTH	设置此位后，调试器在反汇编过程中设置指令格式时会忽略输出显示的宽度。 这等效于 .asm 命令中的 ignore_output_width 选项。
DEBUG_ASMOPT_SOURCE_LINE_NUMBER	设置此位后，反汇编输出的每一行都以符号信息提供的源代码行号作为前缀。 这等效于 .asm 命令中的 source_line 选项。

备注

此外，值 DEBUG_ASMOPT_DEFAULT 表示程序集和反汇编选项的默认集。 这意味着上表中的所有选项都已关闭。

要求

展开表

标头	DbgEng.h (包括 DbgEng.h)

DEBUG_ATTACH_XXX

项目 • 2023/10/26

本主题中所述的 DEBUG_ATTACH_*XXX* 位标志控制调试器引擎如何附加到用户模式进程。 有关附加到内核目标时使用的DEBUG_ATTACH_XXX选项，请参阅 IDebugClient：：AttachKernel 方法。

可能的值包括以下内容。

一直	说明
DEBUG_ATTACH_NONINVASIVE	以无创方式附加到目标。 有关非侵入性调试的详细信息，请参阅 非侵入调试 (用户模式) 。 如果设置了此标志，则不得设置 DEBUG_ATTACH_EXISTING、DEBUG_ATTACH_INVASIVE_NO_INITIAL_BREAK 和 DEBUG_ATTACH_INVASIVE_RESUME_PROCESS 标志。
DEBUG_ATTACH_EXISTING	重新附加到调试器已附加到 (且可能已放弃) 的应用程序。 有关重新附加到目标的详细信息，请参阅 .attach (附加到进程) 。 如果设置了此标志，则不能设置其他 DEBUG_ATTACH_*XXX* 标志。
DEBUG_ATTACH_NONINVASIVE_NO_SUSPEND	在非侵入性附加时，不要挂起目标的线程。 如果设置了此标志，则还必须设置 DEBUG_ATTACH_NONINVASIVE标志。
DEBUG_ATTACH_INVASIVE_NO_INITIAL_BREAK	附加到目标时，请勿请求初始中断。 如果设置了此标志，则不得设置 DEBUG_ATTACH_NONINVASIVE标志和 DEBUG_ATTACH_EXISTING。
DEBUG_ATTACH_INVASIVE_RESUME_PROCESS	以侵入性方式连接时恢复目标的所有线程。 如果设置了此标志，则不得设置 DEBUG_ATTACH_NONINVASIVE标志和 DEBUG_ATTACH_EXISTING。

要求

标头	DbgEng.h (包括 DbgEng.h)

DBG_DUMP_XXX

项目 • 2023/06/15

SYM_DUMP_PARAM 结构的 **Options** 成员使用 DBG_DUMP_*XXX* 位标志来控制 IG_DUMP_SYMBOL_INFOIoctl 操作的行为。

可以存在以下标志。

标志	效果
DBG_DUMP_NO_INDENT	成员不会在输出中缩进。
DBG_DUMP_NO_OFFSET	不打印偏移量。
DBG_DUMP_VERBOSE	详细输出。
DBG_DUMP_CALL_FOR_EACH	为每个成员调用回调函数。
DBG_DUMP_LIST	符号是链接列表中的条目，IG_DUMP_SYMBOL_INFO **Ioctl** 操作将循环访问此列表。 指向列表中下一项的成员的说明由 SYM_DUMP_PARAM 结构的 **linkList** 成员指定。
DBG_DUMP_NO_PRINT	不打印任何内容 (仅调用回调函数，并且) 执行数据复制。
DBG_DUMP_GET_SIZE_ONLY	**Ioctl** 操作仅返回符号的大小;它不会打印成员信息或调用回调函数。
DBG_DUMP_COMPACT_OUT	不在每个成员之后打印换行符。
DBG_DUMP_ARRAY	符号是数组。 数组中的元素数由 SYM_DUMP_PARAM 结构的成员 **listLink 大小**> 指定。
DBG_DUMP_ADDRESS_OF_FIELD	**addr** 的值实际上是SYM_DUMP_PARAM结构的成员 **listLink-fName**> 的地址，而不是符号的开头。
DBG_DUMP_ADDRESS_AT_END	**addr** 的值实际上是符号末尾的地址，而不是符号开头的地址。
DBG_DUMP_COPY_TYPE_DATA	符号的值将复制到成员 **pBuffer 中**。 这只能用于基元类型（例如 ULONG 或 PVOID），不能与结构一起使用。
DBG_DUMP_READ_PHYSICAL	符号的值将直接从目标的物理内存中读取。

标志	效果
DBG_DUMP_FUNCTION_FORMAT	设置具有函数类型的符号的格式时，将使用函数格式，例如 `function(arg1, arg2, ...)`
DBG_DUMP_BLOCK_RECURSE	通过嵌套结构递归;但不遵循指针。

此外，可以将宏DBG_DUMP_RECUR_LEVEL (*Level*) 的结果添加到位集中，以指定要递归结构的深度。 *级别* 可以是介于 0 和 15 之间的数字。

要求

标头	Wdbgexts.h (包括 Wdbgexts.h、Wdbgexts.h 或 Dbgeng.h)

另请参阅

IG_DUMP_SYMBOL_INFO

Ioctl

DBG_DUMP_FIELD_XXX

项目 • 2024/02/06

FIELD_INFO **结构的fOptions** 成员使用 DBG_DUMP_FIELD_*XXX* 位标志来控制 IG_DUMP_SYMBOL_INFOIoctl 操作的行为。

可以存在以下标志。

展开表

标志	效果
DBG_DUMP_FIELD_CALL_BEFORE_PRINT	在打印成员之前调用回调函数。
DBG_DUMP_FIELD_NO_CALLBACK_REQ	未调用任何回调函数。
DBG_DUMP_FIELD_RECUR_ON_THIS	处理成员的子成员。
DBG_DUMP_FIELD_FULL_NAME	**fName** 必须完全匹配，而不是仅具有匹配的前缀，才能处理成员。
DBG_DUMP_FIELD_ARRAY	打印数组成员的数组元素。
DBG_DUMP_FIELD_COPY_FIELD_DATA	成员的值将复制到 **pBuffer 中**。
DBG_DUMP_FIELD_RETURN_ADDRESS	在回调期间或 **Ioctl** 返回时，FIELD_INFO。**address** 成员包含符号成员的地址。 如果未为类型提供地址，FIELD_INFO。**address** 包含成员与类型开头的总偏移量。
DBG_DUMP_FIELD_SIZE_IN_BITS	对于位字段，返回偏移量和大小（以位而不是字节为单位）。
DBG_DUMP_FIELD_NO_PRINT	请勿打印此成员 (仅调用回调函数并) 执行数据复制。
DBG_DUMP_FIELD_DEFAULT_STRING DBG_DUMP_FIELD_WCHAR_STRING DBG_DUMP_FIELD_MULTI_STRING DBG_DUMP_FIELD_GUID_STRING	如果成员是指针，则将其打印为字符串、ANSI 字符串、WCHAR 字符串、MULTI 字符串或 GUID。

此外，可以将宏DBG_DUMP_RECUR_LEVEL (*Level*) 的结果添加到位集中，以指定要递归结构的深度。 *级别* 可以是介于 0 和 15 之间的数字。

要求

展开表

标头	Wdbgexts.h (包括 Wdbgexts.h、Wdbgexts.h 或 Dbgeng.h)

另请参阅

IG_DUMP_SYMBOL_INFO

Ioctl

FIELD_INFO

DEBUG_ENGOPT_XXX

项目 • 2024/02/06

以下全局选项会影响调试器引擎的行为。

展开表

一直	说明
DEBUG_ENGOPT_IGNORE_DBGHELP_VERSION	如果 DbgHelp DLL 的版本与调试器引擎的版本不匹配，调试器引擎将生成警告而不是错误。
DEBUG_ENGOPT_IGNORE_EXTENSION_VERSIONS	禁用扩展的版本检查。 这会取消调试器引擎对 CheckVersion 的调用。
DEBUG_ENGOPT_ALLOW_NETWORK_PATHS	网络共享可用于加载符号和扩展。 此选项可防止引擎在调试某些系统进程时禁止网络路径，应谨慎使用。 如果设置了 DEBUG_ENGOPT_DISALLOW_NETWORK_PATHS，则无法设置此选项。
DEBUG_ENGOPT_DISALLOW_NETWORK_PATHS	网络共享不能用于加载符号和扩展。 在调试某些系统进程时，引擎会尝试设置此选项。 如果设置了 DEBUG_ENGOPT_ALLOW_NETWORK_PATHS，则无法设置此选项。
DEBUG_ENGOPT_NETWORK_PATHS	DEBUG_ENGOPT_ALLOW_NETWORK_PATHS和 DEBUG_ENGOPT_DISALLOW_NETWORK_PATHS的按位 OR。
DEBUG_ENGOPT_IGNORE_LOADER_EXCEPTIONS	忽略某些较旧版本的 Windows 中加载程序生成的预期的首次机会异常。
DEBUG_ENGOPT_INITIAL_BREAK	在目标的初始事件中中断调试器。
DEBUG_ENGOPT_INITIAL_MODULE_BREAK	当目标加载其第一个模块时，中断调试器。
DEBUG_ENGOPT_FINAL_BREAK	在目标的最后一个事件中中断调试器。 在实时用户模式目标中，这是进程退出时。 它在内核模式下不起作用。
DEBUG_ENGOPT_NO_EXECUTE_REPEAT	给定空命令时，调试器引擎不会重复上一个命令。
DEBUG_ENGOPT_FAIL_INCOMPLETE_INFORMATION	防止调试器加载无法映射映像的模块。

	调试器尝试在调试不包含图像的最小转数时加载映像。
DEBUG_ENGOPT_ALLOW_READ_ONLY_BREAKPOINTS	允许调试器引擎操作目标上的页面保护，以允许在内存的只读部分中设置软件断点。 设置软件断点时，引擎以透明方式更改目标的内存以插入中断指令。
DEBUG_ENGOPT_SYNCHRONIZE_BREAKPOINTS	在实时用户模式调试中，引擎在插入和删除断点时执行额外的工作，以确保目标中的所有线程始终具有一致的断点状态。 当多个线程可以使用设置了断点的代码时，此选项很有用。 但是，它可能会引入死锁的可能性。
DEBUG_ENGOPT_DISALLOW_SHELL_COMMANDS	禁止通过调试器执行 shell 命令。 设置此选项后，无法取消设置。
DEBUG_ENGOPT_KD_QUIET_MODE	打开安静模式。 有关详细信息，请参阅 **sq (设置安静模式)** 。
DEBUG_ENGOPT_DISABLE_MANAGED_SUPPORT	禁用调试器引擎对托管代码的支持。 如果已使用对托管代码的支持，则此选项不起作用。
DEBUG_ENGOPT_DISABLE_MODULE_SYMBOL_LOAD	调试器不会加载在设置此标志时加载的模块的符号。
DEBUG_ENGOPT_DISABLE_EXECUTION_COMMANDS	阻止导致目标开始执行的任何命令。
DEBUG_ENGOPT_DISALLOW_IMAGE_FILE_MAPPING	禁止映射磁盘中的图像文件。 例如，此选项禁止在调试小型转储文件时对内存内容进行图像映射。 此选项不会影响现有映射;它仅影响映射图像文件的后续尝试。
DEBUG_ENGOPT_PREFER_DML	默认情况下，调试器运行 DML 增强版本的命令和操作。
DEBUG_ENGOPT_DISABLESQM	禁止上传软件质量指标 (SQM) 数据。

要求

展开表

标头	DbgEng.h (包括 DbgEng.h)

DEBUG_EVENT_XXX

项目 • 2023/03/08

以下事件由目标生成。

标志	IDebugEventCallbacksMethod	事件描述
DEBUG_EVENT_BREAKPOINT	IDebugEventCallbacks：：Breakpoint	目标中发生断点异常。
DEBUG_EVENT_EXCEPTION	IDebugEventCallbacks：：Exception	目标中发生了异常调试事件。
DEBUG_EVENT_CREATE_THREAD	IDebugEventCallbacks：：CreateThread	目标中发生了创建线程调试事件。
DEBUG_EVENT_EXIT_THREAD	IDebugEventCallbacks：：ExitThread	目标中发生了退出线程调试事件。
DEBUG_EVENT_CREATE_PROCESS	IDebugEventCallbacks：：CreateProcess	目标中发生了创建进程调试事件。
DEBUG_EVENT_EXIT_PROCESS	IDebugEventCallbacks：：ExitProcess	目标中发生了退出进程调试事件。
DEBUG_EVENT_LOAD_MODULE	IDebugEventCallbacks：：LoadModule	目标中发生了模块加载调试事件。
DEBUG_EVENT_UNLOAD_MODULE	IDebugEventCallbacks：：UnloadModule	目标中发生了模块卸载调试事件。
DEBUG_EVENT_SYSTEM_ERROR	IDebugEventCallbacks：：SystemError	目标中发生系统错误。

以下事件由调试器引擎生成。

标志	IDebugEventCallbacksMethod	说明
DEBUG_EVENT_SESSION_STATUS	IDebugEventCallbacks：：SessionStatus	会话状态发生了更改。
DEBUG_EVENT_CHANGE_DEBUGGEE_STATE	IDebugEventCallbacks：：ChangeDebuggeeState	引擎已对目标状态进行了更改或检测到更改。
DEBUG_EVENT_CHANGE_ENGINE_STATE	IDebugEventCallbacks：：ChangeEngineState	引擎状态已更改。

DEBUG_EVENT_CHANGE_SYMBOL_STATE	**IDebugEventCallbacks：：ChangeSymbolState**	符号状态已更改。

要求

标头	DbgEng.h (包括 DbgEng.h)

DEBUG_FILTER_XXX

项目 • 2023/06/15

DEBUG_FILTER_*XXX* 常量用于三种不同的目的：指定单个特定事件筛选器、指定事件筛选器的中断状态以及指定异常筛选器的处理状态。

特定事件筛选器

以下常量用于指定特定的事件筛选器。

值	事件
DEBUG_FILTER_CREATE_THREAD	创建线程
DEBUG_FILTER_EXIT_THREAD	退出线程
DEBUG_FILTER_CREATE_PROCESS	创建进程
DEBUG_FILTER_EXIT_PROCESS	退出进程
DEBUG_FILTER_LOAD_MODULE	加载模块
DEBUG_FILTER_UNLOAD_MODULE	卸载模块
DEBUG_FILTER_SYSTEM_ERROR	系统错误
DEBUG_FILTER_INITIAL_BREAKPOINT	初始断点
DEBUG_FILTER_INITIAL_MODULE_LOAD	初始模块加载
DEBUG_FILTER_DEBUGGEE_OUTPUT	目标输出

中断状态

以下常量用于指定事件筛选器的中断状态。

值	说明
DEBUG_FILTER_BREAK	事件将中断到调试器中。
DEBUG_FILTER_SECOND_CHANCE_BREAK	如果是第二次偶然异常，该事件将中断到调试器中。
DEBUG_FILTER_OUTPUT	事件通知将输出到调试器控制台。
DEBUG_FILTER_IGNORE	忽略事件。

此外，对于任意异常筛选器，将中断状态设置为DEBUG_FILTER_REMOVE，将删除事件筛选器。

处理状态

以下常量用于指定异常筛选器的处理状态。

值	说明
DEBUG_FILTER_GO_HANDLED	已处理异常。
DEBUG_FILTER_GO_NOT_HANDLED	尚未处理异常。

要求

标头	DbgEng.h (包括 DbgEng.h)

DEBUG_FORMAT_XXX

项目 • 2023/06/15

WriteDumpFile2 和 WriteDumpFileWide 使用DEBUG_FORMAT_XXX位标志来确定故障转储文件的格式，以及要包含在文件中的用户模式 Minidumps 的信息。

以下位标志适用于所有故障转储文件。

值	说明
DEBUG_FORMAT_WRITE_CAB	将故障转储文件打包到 CAB 文件中。 提供的文件名或文件句柄用于 CAB 文件;故障转储首先在临时文件中创建，然后移动到 CAB 文件。
DEBUG_FORMAT_CAB_SECONDARY_FILES	在 CAB 文件中包括当前符号和映射的图像。 如果未设置DEBUG_FORMAT_WRITE_CAB，则忽略此标志。
DEBUG_FORMAT_NO_OVERWRITE	不要覆盖现有文件。

还可以为用户模式微型umps 包含以下位标志。

值	说明
DEBUG_FORMAT_USER_SMALL_FULL_MEMORY	添加完整内存数据。 将包含目标应用程序拥有的所有可访问的提交页面。
DEBUG_FORMAT_USER_SMALL_HANDLE_DATA	添加有关与目标应用程序关联的句柄的数据。
DEBUG_FORMAT_USER_SMALL_UNLOADED_MODULES	添加卸载的模块信息。 此信息仅在 Windows Server 2003 及更高版本的 Windows 中可用。
DEBUG_FORMAT_USER_SMALL_INDIRECT_MEMORY	添加间接内存。 包含一小块内存，它包围了堆栈或后备存储上的指针所引用的任何地址。
DEBUG_FORMAT_USER_SMALL_DATA_SEGMENTS	在可执行映像中添加所有数据段。
DEBUG_FORMAT_USER_SMALL_FILTER_MEMORY	将堆栈上和后备存储中的所有内存设置为零，这对重新创建堆栈跟踪没有用。 这可以使压缩小型转储更高效，并通过删除不必要的信息来增加隐私。
DEBUG_FORMAT_USER_SMALL_FILTER_PATHS	删除模块路径，只保留模块名称。 这对于通过隐藏目录结构（（可能包含用户名））来保护隐私非常有用。
DEBUG_FORMAT_USER_SMALL_FILTER_TRIAGE	此格式用于筛选出不是指向转储中捕获的其他数据的指针的任何数据。 标志可用于减少转储中存在的私有数据量，同时仍允许诊断崩溃。
DEBUG_FORMAT_USER_SMALL_PROCESS_THREAD_DATA	将进程环境块 (PEB) 和线程环境块 (TEB) 。 此标志可用于为线程和进程提供 Windows 系统信息。
DEBUG_FORMAT_USER_SMALL_PRIVATE_READ_WRITE_MEMORY	添加所有提交的专用读写内存页。
DEBUG_FORMAT_USER_SMALL_NO_OPTIONAL_DATA	防止将隐私敏感数据包含在微型转储中。 目前，此标志从由于设置了以下标志而添加的小型转储数据中排除： DEBUG_FORMAT_USER_SMALL_PROCESS_THREAD_DATA， DEBUG_FORMAT_USER_SMALL_FULL_MEMORY， DEBUG_FORMAT_USER_SMALL_INDIRECT_MEMORY， DEBUG_FORMAT_USER_SMALL_PRIVATE_READ_WRITE_MEMORY。

DEBUG_FORMAT_USER_SMALL_FULL_MEMORY_INFO	添加所有基本内存信息。 这是 IDebugDataSpaces2：：QueryVirtual 方法返回的信息。 包含所有内存的信息，而不仅仅是有效内存，它允许调试器从微型转储重建完整的虚拟内存布局。
DEBUG_FORMAT_USER_SMALL_THREAD_INFO	添加其他线程信息，包括执行时间、开始时间、退出时间、开始地址和退出状态。
DEBUG_FORMAT_USER_SMALL_CODE_SEGMENTS	使用可执行映像添加所有代码段。

要求

标头	DbgEng.h (包括 DbgEng.h)

DEBUG_OUTCB_XXX

项目 • 2024/02/07

DEBUG_OUTCB_XXX常量是输出标志。 输出标志形成一个位字段，指示其附带的输出的类型。

DEBUG_OUTCB_XXX定义可发送到 Output2 的不同类型的输出回调通知。

可能的值包括以下内容。

展开表

一直	说明
DEBUG_OUTCB_TEXT	纯文本内容，标志如下，参数为掩码。
DEBUG_OUTCB_DML	调试器标记内容，标志如下，参数为 mask。
DEBUG_OUTCB_EXPLICIT_FLUSH	显式输出刷新的通知，标志和参数为零。

要求

标头：DbgEng.h (包括 DbgEng.h)

DEBUG_OUTCTL_XXX

项目 • 2024/02/07

DEBUG_OUTCTL_*XXX* 常量用于输出控制。 常量形成一个位字段，用于指定输出发送位置的当前策略。 位字段分为两个部分。

较低的位必须正好是以下值之一。

展开表

值	说明
DEBUG_OUTCTL_THIS_CLIENT	此客户端调用的方法生成的输出将仅发送到此客户端的 输出回调。
DEBUG_OUTCTL_ALL_CLIENTS	输出将发送到所有客户端。
DEBUG_OUTCTL_ALL_OTHER_CLIENTS	输出将发送到所有客户端 (，但生成输出) 的客户端除外。
DEBUG_OUTCTL_IGNORE	输出将立即放弃，并且不会记录或发送到回调。
DEBUG_OUTCTL_LOG_ONLY	输出将记录，但不会发送到回调。

位字段的较高位可以包含以下值。

展开表

值	说明
DEBUG_OUTCTL_NOT_LOGGED	不要将此客户端的输出放在全局日志文件中。
DEBUG_OUTCTL_OVERRIDE_MASK	无论客户端的输出掩码是否允许，都向客户端发送输出。
DEBUG_OUTCTL_DML	对于支持调试器标记语言 (DML) 的输出，以 DML 格式发送输出。

若要创建有效的输出控件位字段，请从第一个表中获取一个值，以及第二个表中的零个或多个值，然后使用按位 OR 运算符合并它们。

输出控件位字段的默认值为DEBUG_OUTCTL_ALL_CLIENTS。

作为创建自己的输出控件位字段的替代方法，可以使用以下值之一。

展开表

值	说明
DEBUG_OUTCTL_AMBIENT_DML	将新的输出控件设置为与当前输出控件相同的值，并指定输出将采用 DML 格式。
DEBUG_OUTCTL_AMBIENT_TEXT	将新的输出控件设置为与当前输出控件相同的值，并指定输出将采用文本格式。
DEBUG_OUTCTL_AMBIENT	与 DEBUG_OUTCTL_AMBIENT_TEXT 相同。

要求

展开表

标头	DbgEng.h (包括 DbgEng.h)

DEBUG_OUTPUT_XXX

项目 • 2024/02/06

DEBUG_OUTPUT_*XXX* 常量是输出标志。 输出标志形成一个位字段，指示它们附带的输出类型。

可能的值包括以下内容。

展开表

一直	说明
DEBUG_OUTPUT_NORMAL	正常输出。
DEBUG_OUTPUT_ERROR	错误输出。
DEBUG_OUTPUT_WARNING	警告。
DEBUG_OUTPUT_VERBOSE	其他输出。
DEBUG_OUTPUT_PROMPT	提示输出。
DEBUG_OUTPUT_PROMPT_REGISTERS	在提示之前注册转储。
DEBUG_OUTPUT_EXTENSION_WARNING	特定于扩展操作的警告。
DEBUG_OUTPUT_DEBUGGEE	调试目标 (的输出，例如 OutputDebugString 或 DbgPrint) 。
DEBUG_OUTPUT_DEBUGGEE_PROMPT	目标 (预期的调试输入，例如 DbgPrompt) 。
DEBUG_OUTPUT_SYMBOLS	符号消息 (例如 ！ sym 干扰) 。
DEBUG_OUTPUT_STATUS	修改状态栏的输出。

要求

展开表

标头	DbgEng.h (包括 DbgEng.h)

DEBUG_PROCESS_XXX

项目 • 2023/10/26

进程选项是一个位集，用于控制 调试器引擎 如何处理用户模式进程。 其中一些进程选项是全局性的;其他是特定于某个进程的。

进程选项仅适用于实时用户模式调试。

位标志	说明
DEBUG_PROCESS_DETACH_ON_EXIT	调试器退出时，调试器会自动将自身与目标进程分离。 这是全局设置。
DEBUG_PROCESS_ONLY_THIS_PROCESS	调试器不会调试由此进程创建的子进程。

要求

标头	DbgEng.h (包括 DbgEng.h)

另请参阅

.childdbg

DEBUG_STATUS_XXX

项目 • 2024/02/07

DEBUG_STATUS_*XXX* 状态代码有两个用途。 它们指示引擎应如何继续在目标中执行，引擎使用它们来报告目标的执行状态。

事件发生后，引擎可以接收多个指令，告知其应如何继续在目标中执行。 在这种情况下，它作用于优先级最高的指令。 通常，优先级较高的状态代码表示目标执行越少。

下表中的值按优先级反向排序：前面出现在表中的值具有更高的优先级。

展开表

状态代码	报告时	指示时	优先级
DEBUG_STATUS_NO_DEBUGGEE	没有处于活动状态的调试会话。	不可用	
DEBUG_STATUS_OUT_OF_SYNC	调试器通信通道不同步。	不可用	
DEBUG_STATUS_WAIT_INPUT	目标正在等待用户的输入。	不可用	
DEBUG_STATUS_TIMEOUT	调试器通信通道已超时。	N/A	
DEBUG_STATUS_BREAK	目标已挂起。	挂起目标。	最高优先级
DEBUG_STATUS_STEP_INTO	目标正在执行单个指令。	继续执行单个指令的目标。	
DEBUG_STATUS_STEP_BRANCH	目标一直执行到下一个分支指令。	继续执行目标，直到下一个分支指令。	
DEBUG_STATUS_STEP_OVER	目标正在执行单个指令，或者（如果该指令是子例程调用）--子例程。	继续执行单个指令的目标。 如果指令是子例程调用，则输入调用并允许目标运行，直到子例程返回。	
DEBUG_STATUS_GO_NOT_HANDLED	N/A	继续执行目标，将事件标记为未处理。	

状态代码	报告时	指示时	优先级
DEBUG_STATUS_GO_HANDLED	不可用	继续执行目标，将事件标记为已处理。	
DEBUG_STATUS_GO	目标正在正常执行。	继续正常执行目标。	
DEBUG_STATUS_IGNORE_EVENT	不可用	继续上一个目标执行，忽略 事件。	
DEBUG_STATUS_RESTART_REQUESTED	目标正在重启。	重启目标。	
DEBUG_STATUS_NO_CHANGE	不可用	无指令。 当事件回调方法不希望指示引擎如何在目标中继续执行时，该值由它返回。	最低优先级

备注

状态代码的优先级不遵循常量的数字值。

要求

展开表

标头	DbgEng.h (包括 DbgEng.h)

DEBUG_SYMBOL_XXX

项目 • 2024/02/07

DEBUG_SYMBOL_*XXX* 常量用于符号标志位集。 符号标志描述 (部分) 符号组中的符号。

符号标志的最小有效位（在 DEBUG_SYMBOL_EXPANSION_LEVEL_MASK 中找到的位）构成一个数字，表示符号组中符号的扩展深度。 子符号的深度始终是其父符号的深度的一倍。 例如，若要 *查找变量标志*中包含其标志的符号的深度，请使用以下语句：

```
dbgcmd
```

```
depth = flags & DEBUG_SYMBOL_EXPANSION_LEVEL_MASK;
```

其余符号标志的位集可以包含以下位标志。

展开表

一直	说明
DEBUG_SYMBOL_EXPANDED	符号的子级是符号组的一部分。
DEBUG_SYMBOL_READ_ONLY	符号表示只读变量。
DEBUG_SYMBOL_IS_ARRAY	符号表示数组变量。
DEBUG_SYMBOL_IS_FLOAT	符号表示浮点变量。
DEBUG_SYMBOL_IS_ARGUMENT	符号表示传递给函数的参数。
DEBUG_SYMBOL_IS_LOCAL	符号表示作用域中的局部变量。

要求

展开表

标头	DbgEng.h (包括 DbgEng.h)

另请参阅

DEBUG_SYMBOL_PARAMETERS

DEBUG_TYPEOPTS_XXX

项目 • 2024/02/07

类型选项会影响引擎如何为输出设置数字和字符串的格式。

选项由具有以下位标志的位集表示。

展开表

一直	说明
DEBUG_TYPEOPTS_UNICODE_DISPLAY	设置此位后，USHORT 指针和数组将输出为 Unicode 字符。 这等效于调试器命令 **.enable_unicode 1**。
DEBUG_TYPEOPTS_LONGSTATUS_DISPLAY	设置此位时，LONG 整数输出为默认基数而不是十进制。 这等效于调试器命令 **.enable_long_status 1**。
DEBUG_TYPEOPTS_FORCERADIX_OUTPUT	设置此位后，除 LONG 整数) 以外的整数 (输出为默认基数而不是十进制。 这等效于调试器命令 **.force_radix_output 1**。

注解

默认情况下，所有类型格式设置选项都处于关闭状态。

有关类型的详细信息，请参阅 类型。

要求

展开表

标头	DbgEng.h (包括 DbgEng.h)

HRESULT 值

项目 • 2023/06/15

下面是函数和方法的常见返回值及其常用含义的列表。

成功结果。 这些值在 WinError.h 中定义。

S_OK
成功完成。

S_FALSE
已完成且未出错，但只获得了部分结果。

如果缓冲区的大小不足以容纳返回给它的信息，则通常截断返回的信息以适应缓冲区，并从 方法返回S_FALSE。

错误结果。 这些值在 WinError.h 中定义。

E_FAIL
无法执行该操作。

E_INVALIDARG
传入的参数之一无效。

E_NOINTERFACE
找不到搜索的对象。

E_OUTOFMEMORY
内存分配尝试失败。

E_UNEXPECTED
无法访问目标，或者引擎未处于可以处理函数或方法的状态。

E_NOTIMPL
未实现。

HRESULT_FROM_WIN32 (ERROR_ACCESS_DENIED)
操作被拒绝，因为调试器处于 安全模式。

NT 错误结果。 有时可能会出现其他错误代码，例如STATUS_CONTROL_C_EXIT和STATUS_NO_MORE_ENTRIES。 在返回之前，这些结果将传递到 winError.h 中定义的HRESULT_FROM_NT宏。

Win32 错误结果。 有时可能会出现其他错误代码，例如ERROR_READ_FAULT和ERROR_WRITE_FAULT。 在返回之前，这些结果将传递到 winError.h 中定义的HRESULT_FROM_WIN32宏。

要求

标头	DbgEng.h (包括 DbgEng.h)

IDebugAdvanced2： ： Request 方法

项目 • 2024/02/07

Request 方法执行各种不同的操作。

语法

C++

```
HRESULT Request(
  [in]            ULONG  Request,
  [in, optional]  PVOID  InBuffer,
  [in]            ULONG  InBufferSize,
  [out, optional] PVOID  OutBuffer,
  [in]            ULONG  OutBufferSize,
  [out, optional] PULONG OutSize
);
```

参数

请求 [in]
指定要执行的操作。 **请求** 可以是下表中的值之一。 可按照“请求”列中的链接找到每个操作的详细信息。

展开表

请求	操作
DEBUG_REQUEST_SOURCE_PATH_HAS_SOURCE_SERVER	检查源服务器的源路径。
DEBUG_REQUEST_TARGET_EXCEPTION_CONTEXT	返回用户模式小型转储文件中存储事件的 线程上下文 。
DEBUG_REQUEST_TARGET_EXCEPTION_THREAD	返回用户模式小型转储文件中存储事件的操作系统线程 ID。
DEBUG_REQUEST_TARGET_EXCEPTION_RECORD	返回用户模式小型转储文件中存储的事件的异常记录。
DEBUG_REQUEST_GET_ADDITIONAL_CREATE_OPTIONS	返回默认的进程创建选项。
DEBUG_REQUEST_SET_ADDITIONAL_CREATE_OPTIONS	设置默认的进程创建选项。
DEBUG_REQUEST_GET_WIN32_MAJOR_MINOR_VERSIONS	返回当前在目标上运行的 Windows 版本。

请求	操作
DEBUG_REQUEST_READ_USER_MINIDUMP_STREAM	从用户模式小型转储目标读取流。
DEBUG_REQUEST_TARGET_CAN_DETACH	检查调试器引擎是否可以从当前进程分离，(使进程保持运行，但不再) 调试。
DEBUG_REQUEST_SET_LOCAL_IMPLICIT_COMMAND_LINE	设置 调试器引擎的隐式命令行。
DEBUG_REQUEST_GET_CAPTURED_EVENT_CODE_OFFSET	返回当前事件的指令指针。
DEBUG_REQUEST_READ_CAPTURED_EVENT_CODE_STREAM	在当前事件的指令指针处返回最多 64 字节的内存。
DEBUG_REQUEST_EXT_TYPED_DATA_ANSI	执行有助于解释类型化数据的各种不同操作。

InBuffer [in， optional]
指定此方法的输入。 输入的类型和解释取决于 *Request* 参数。

InBufferSize [in]
指定输入缓冲区 *InBuffer* 的大小。 如果请求不需要任何输入， *则 InBufferSize* 应设置为零。

OutBuffer [out， optional]
从此方法接收输出。 输出的类型和解释取决于 *Request* 参数。 如果 *OutBuffer* 为 **NULL**，则不返回输出。

OutBufferSize [in]
指定输出缓冲区 *OutBufferSize* 的大小。 如果返回给 *OutBuffer* 的输出类型具有已知大小，则通常预期 *OutBufferSize* 正是该大小，即使 *OutBuffer* 设置为 **NULL** 也是如此。

OutSize [out， optional]
接收输出缓冲区 *OutBuffer* 中返回的输出的大小。 如果 *OutSize* 为 **NULL**，则不返回此信息。

返回值

返回值的解释取决于 *Request* 参数的值。 除非另有说明，否则可能会返回以下值。

展开表

返回代码	说明
S_OK	方法成功。

返回代码	说明
S_FALSE	方法成功。 但是，输出不适合输出缓冲区 *OutBuffer*，因此返回了截断的输出。
E_INVALIDARG	输入缓冲区 *InBufferSize* 的大小或输出缓冲区 *OutBufferSize* 的大小不是预期值。

此方法也可能返回错误值。 有关更多详细信息，**请参阅返回值**。

要求

展开表

目标平台	桌面
标头	Dbgeng.h (包括 Dbgeng.h)

另请参阅

IDebugAdvanced2

IDebugAdvanced3

DEBUG_REQUEST_SOURCE_PATH_HAS_SOURCE_SERVER

DEBUG_REQUEST_TARGET_EXCEPTION_CONTEXT

DEBUG_REQUEST_TARGET_EXCEPTION_THREAD

DEBUG_REQUEST_TARGET_EXCEPTION_RECORD

DEBUG_REQUEST_GET_ADDITIONAL_CREATE_OPTIONS

DEBUG_REQUEST_SET_ADDITIONAL_CREATE_OPTIONS

DEBUG_REQUEST_GET_WIN32_MAJOR_MINOR_VERSIONS

DEBUG_REQUEST_READ_USER_MINIDUMP_STREAM

DEBUG_REQUEST_TARGET_CAN_DETACH

DEBUG_REQUEST_SET_LOCAL_IMPLICIT_COMMAND_LINE

DEBUG_REQUEST_GET_CAPTURED_EVENT_CODE_OFFSET

DEBUG_REQUEST_READ_CAPTURED_EVENT_CODE_STREAM

DEBUG_REQUEST_SOURCE_PATH_HAS_SOURCE_SERVER

项目 • 2023/06/15

DEBUG_REQUEST_SOURCE_PATH_HAS_SOURCE_SERVER **请求** 操作检查源服务器的源路径。

参数

InBuffer
未使用。

OutBuffer
未使用。

返回值

S_OK
源路径包括源服务器

S_FALSE
源路径不包括源服务器。

另请参阅

Request

DEBUG_REQUEST_TARGET_EXCEPTION_CONTEXT

项目 • 2024/02/06

DEBUG_REQUEST_TARGET_EXCEPTION_CONTEXT **请求** 操作返回用户模式小型转储文件中存储事件的 线程上下文 。

参数

InBuffer
未使用。

OutBuffer
存储事件的线程上下文。 线程上下文的类型是事件发生时目标有效处理器的 CONTEXT 结构。 *OutBuffer* 必须足够大才能容纳此结构。

注解

此信息也会由 **GetStoredEventInformation** 方法返回到 *Context* 参数。

另请参阅

Request

GetStoredEventInformation

DEBUG_REQUEST_TARGET_EXCEPTION_THREAD

项目 • 2023/06/15

DEBUG_REQUEST_TARGET_EXCEPTION_THREAD **请求** 操作返回用户模式小型转储文件中存储事件的操作系统线程 ID。

参数

InBuffer
未使用。

OutBuffer
存储事件的操作系统线程 ID。 线程 ID 的类型为 ULONG。

另请参阅

Request

DEBUG_REQUEST_TARGET_EXCEPTION_RECORD

项目 • 2023/06/15

DEBUG_REQUEST_TARGET_EXCEPTION_RECORD **请求** 操作返回用户模式小型转储文件中存储的事件的异常记录。

InBuffer
未使用。

OutBuffer
存储事件的异常记录。 异常记录的类型EXCEPTION_RECORD64，在 winnt.h 中定义。

另请参阅

Request

DEBUG_REQUEST_GET_ADDITIONAL_CREATE_OPTIONS控制代码

项目 • 2023/06/15

DEBUG_REQUEST_GET_ADDITIONAL_CREATE_OPTIONS **请求** 操作返回默认的进程创建选项。

参数

InBuffer
未使用。

OutBuffer
默认的进程创建选项。 进程创建选项的类型 DEBUG_CREATE_PROCESS_OPTIONS。

注解

默认的进程创建选项由 CreateProcess 和 CreateProcessAndAttach 方法使用，这与 CreateProcess2 和 CreateProcessAndAttach2 不同，它们不指定进程创建选项的完整范围。

DEBUG_CREATE_PROCESS_OPTIONS 结构的 **CreateFlags** 字段不用作默认值，因为所有进程创建操作都提供此信息。

另请参阅

Request

DEBUG_REQUEST_SET_ADDITIONAL_CREATE_OPTIONS

DEBUG_CREATE_PROCESS_OPTIONS

CreateProcess

CreateProcessAndAttach

DEBUG_REQUEST_SET_ADDITIONAL_CREATE_OPTIONS

项目 • 2024/02/07

DEBUG_REQUEST_SET_ADDITIONAL_CREATE_OPTIONS **请求** 操作设置默认的进程创建选项。

参数

InBuffer
新的默认进程创建选项。 进程创建选项的类型 DEBUG_CREATE_PROCESS_OPTIONS。

OutBuffer
未使用。

注解

默认进程创建选项由 CreateProcess 和 CreateProcessAndAttach 方法使用，与 CreateProcess2 **和** CreateProcessAndAttach2 不同，它们不指定完整的进程创建选项范围。

DEBUG_CREATE_PROCESS_OPTIONS 结构的 **CreateFlags** 字段不用作默认值，因为所有进程创建操作都提供此信息。

另请参阅

Request

DEBUG_REQUEST_GET_ADDITIONAL_CREATE_OPTIONS

DEBUG_CREATE_PROCESS_OPTIONS

CreateProcess

CreateProcessAndAttach

DEBUG_REQUEST_GET_WIN32_MAJOR_MINOR_VERSIONS

项目 • 2024/02/06

DEBUG_REQUEST_GET_WIN32_MAJOR_MINOR_VERSIONS **请求** 操作返回当前在目标上运行的 Windows 版本。

参数

InBuffer
未使用。

OutBuffer
当前在目标上运行的 Windows 版本的主版本号和次要版本号。 版本号的类型是包含两个 ULONG 值的数组;数组中的第一个 ULONG 是主版本号，第二个是次要版本号

下表列出了按操作系统列出的主版本号和次要版本号。

展开表

Windows 的版本	主版本号	次要版本号
Windows 2000	5	0
Windows XP	5	1
Windows Server 2003	5	2

另请参阅

Request

DEBUG_REQUEST_TARGET_CAN_DETACH

项目 • 2023/06/15

DEBUG_REQUEST_TARGET_CAN_DETACH **请求** 操作检查调试器引擎是否可以从当前进程分离，（使进程保持运行状态，但不再）调试。

参数

InBuffer
未使用。

OutBuffer
未使用。

返回值

S_OK
可以将调试器与当前进程分离。

S_FALSE
无法将调试器从当前进程分离。

注解

只有在 Microsoft Windows XP 或更高版本的 Windows 上运行的目标才支持将调试器从进程中分离。

另请参阅

Request

DEBUG_REQUEST_SET_LOCAL_IMPLICIT_COMMAND_LINE

项目 • 2024/02/06

DEBUG_REQUEST_SET_LOCAL_IMPLICIT_COMMAND_LINE **请求** 操作设置 调试器引擎的隐式命令行。

参数

InBuffer
新的隐式命令行。 *InBuffer* 的类型是指向 PWSTR) (Unicode 字符串的指针。 复制指针，但不会复制它指向的字符串。

OutBuffer
未使用。

注解

创建进程时，可以将隐式命令行用作命令行。 进程创建选项 (DEBUG_CREATE_PROCESS_OPTIONS) 包含用于在创建进程时使用隐式命令行而不是提供的命令行的选项。

另请参阅

Request

DEBUG_CREATE_PROCESS_OPTIONS

DEBUG_REQUEST_GET_ADDITIONAL_CREATE_OPTIONS

DEBUG_REQUEST_SET_ADDITIONAL_CREATE_OPTIONS

CreateProcess2

CreateProcessAndAttach2

DEBUG_REQUEST_GET_CAPTURED_EVENT_CODE_OFFSET

项目 • 2023/06/14

DEBUG_REQUEST_GET_CAPTURED_EVENT_CODE_OFFSET **请求** 操作返回当前事件的指令指针。

参数

InBuffer
未使用。

OutBuffer
当前事件的指令指针。 指令指针的类型为 ULONG64。

返回值

S_OK
方法成功。

E_NOINTERFACE
当前事件的指令指针处的内存无效。

此方法也可能返回错误值。 有关更多详细信息，**请参阅返回值**。

注解

可以使用 **请求** 操作的 DEBUG_REQUEST_READ_CAPTURED_EVENT_CODE_STREAM读取当前事件的指令指针处的内存。

另请参阅

Request

DEBUG_REQUEST_READ_CAPTURED_EVENT_CODE_STREAM

DEBUG_REQUEST_READ_CAPTURED_EVENT_CODE_STREAM

项目 • 2023/06/14

DEBUG_REQUEST_READ_CAPTURED_EVENT_CODE_STREAM **请求** 操作在当前事件的指令指针处返回最多 64 个字节的内存。

参数

InBuffer
未使用。

OutBuffer
当前事件的指令指针处的内存。 最多可以返回 64 字节的内存。

注解

返回的内存是事件发生时占用的内存的快照。 它不反映自 事件以来可能对目标内存所做的任何更改。

当前事件的指令指针由 **请求** 操作
DEBUG_REQUEST_GET_CAPTURED_EVENT_CODE_OFFSET返回。

另请参阅

Request

DEBUG_REQUEST_GET_CAPTURED_EVENT_CODE_OFFSET

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

项目 • 2023/06/15

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI **请求** 操作执行各种不同的子操作，有助于解释类型化数据。

参数

InBuffer
指定要执行的子操作 **的EXT_TYPED_DATA** 结构。 此EXT_TYPED_DATA结构包含该子操作的输入参数以及任何 (可选) 附加数据。 其他数据包含在 *InBuffer* EXT_TYPED_DATA 结构之后。 *InBuffer* 的大小是包含EXT_TYPED_DATA结构和附加数据的缓冲区的总大小。 有关此结构的详细信息，请参阅 **EXT_TYPED_DATA** 以及如何包含其他数据。

支持以下子操作。

Sub-Operation	说明
EXT_TDOP_COPY	创建类型化数据说明的副本。
EXT_TDOP_RELEASE	发布类型化数据说明。
EXT_TDOP_SET_FROM_EXPR	返回表达式的值。
EXT_TDOP_SET_FROM_U64_EXPR	返回表达式的值。 可选地址可以作为表达式的参数提供。
EXT_TDOP_GET_FIELD	返回结构的一个成员。
EXT_TDOP_EVALUATE	返回表达式的值。 可选值可以作为表达式的参数提供。
EXT_TDOP_GET_TYPE_NAME	返回类型化数据的类型名称。
EXT_TDOP_OUTPUT_TYPE_NAME	打印类型化数据的类型名称。
EXT_TDOP_OUTPUT_SIMPLE_VALUE	打印类型化数据的值。
EXT_TDOP_OUTPUT_FULL_VALUE	打印类型化数据的类型和值。
EXT_TDOP_HAS_FIELD	确定结构是否包含指定成员。
EXT_TDOP_GET_FIELD_OFFSET	返回结构中成员的偏移量。
EXT_TDOP_GET_ARRAY_ELEMENT	从数组返回元素。

Sub-Operation	说明
EXT_TDOP_GET_DEREFERENCE	取消引用指针，并返回指针指向的值。
EXT_TDOP_GET_TYPE_SIZE	返回指定类型化数据的大小。
EXT_TDOP_OUTPUT_TYPE_DEFINITION	打印指定类型化数据的类型定义。
EXT_TDOP_GET_POINTER_TO	返回表示指向指定类型化数据的指针的新类型化数据说明。
EXT_TDOP_SET_FROM_TYPE_ID_AND_U64	从类型和内存位置创建类型化数据说明。
EXT_TDOP_SET_PTR_FROM_TYPE_ID_AND_U64	创建一个类型化数据说明，该说明表示指向具有指定类型的指定内存位置的指针。

OutBuffer
接收 EXT_TYPED_DATA 结构，该结构包含子操作的输出参数和任何其他数据。 与 *InBuffer* 一样， *OutBuffer* 的大小是包含EXT_TYPED_DATA结构和任何其他数据的缓冲区的总大小。

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI操作最初会将 *InBuffer* 复制到 *OutBuffer* 中，然后就地修改 *OutBuffer* 的内容。 这意味着 *OutBuffer* 将填充EXT_TYPED_DATA的输入参数以及 *InBuffer* 中提供的任何其他数据。 这也意味着 *OutBuffer* 的大小必须至少与 *InBuffer* 的大小一样大。

返回值

S_OK
操作成功。

此方法还可以返回错误值。 有关更多详细信息，**请参阅返回值**。

此操作返回的值也存储在 *OutBuffer* 的 **Status** 成员中。

注解

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI Request 操作执行的子操作由 EXT_TYPED_DATA 结构的 **Operation** 成员确定，该成员在 EXT_TDOP 枚举中获取值。

另请参阅

EXT_TYPED_DATA

EXT_TDOP

Request

EXT_TDOP_COPY

项目 • 2024/02/06

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI**请求**操作的EXT_TDOP_COPY子操作会复制类型化数据说明。

参数

操作
对于此子操作，设置为 EXT_TDOP_COPY。

InData
指定 DEBUG_TYPED_DATA 结构的原始实例。

OutData
接收**由** InData 指定的 DEBUG_TYPED_DATA 结构的实例副本。

地位
接收此子操作返回的状态代码。 这与 Request 返回的值相同。

注解

EXT_TDOP_COPY 是 EXT_TDOP 枚举中的值。

此子操作的参数是 EXT_TYPED_DATA 结构的成员。 此子操作不使用上一参数部分未列出的EXT_TYPED_DATA的成员，应设置为零。 上述参数部分中成员的说明指定了成员的用途。 有关成员的更多详细信息，请参阅 **EXT_TYPED_DATA**。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_RELEASE

项目 • 2024/02/07

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI**请求**操作的**EXT_TDOP_RELEASE**子操作会释放类型化数据说明。

参数

操作
对于此子操作，设置为 EXT_TDOP_RELEASE。

InData
指定要释放的 **DEBUG_TYPED_DATA** 结构的实例。

地位
接收此子操作返回的状态代码。 这与 **Request** 返回的值相同。

注解

EXT_TDOP_RELEASE 是 **EXT_TDOP** 枚举中的值。

此子操作的参数是 **EXT_TYPED_DATA** 结构的成员。 此子操作不使用上一参数部分未列出的EXT_TYPED_DATA的成员，应设置为零。 上述参数部分中成员的说明仅指定此特定设置中成员的用途。 有关更多详细信息，**请参阅**EXT_TYPED_DATA 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_SET_FROM_EXPR

项目 • 2024/02/06

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI Request 操作的EXT_TDOP_SET_FROM_EXPR 子操作返回表示表达式值的类型化数据说明。

参数

操作
对于此子操作，设置为 EXT_TDOP_SET_FROM_EXPR。

标志
指定用于描述表达式值所在的目标内存的位标志。 有关这些标志的详细信息，**请参阅** EXT_TYPED_DATA 。

OutData
接收表示表达式值的 DEBUG_TYPED_DATA 结构。

InStrIndex
指定要计算的表达式。 此表达式由默认表达式计算器计算。

地位
接收此子操作返回的状态代码。 这与 Request 返回的值相同。

注解

EXT_TDOP_SET_FROM_EXPR 是 EXT_TDOP 枚举中的值。

此子操作的参数是 EXT_TYPED_DATA 结构的成员。 此子操作不使用上一个参数部分未列出的EXT_TYPED_DATA成员，应设置为零。 上述"参数"部分中成员的说明指定了成员的用途。 有关详细信息，**请参阅**EXT_TYPED_DATA 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_SET_FROM_U64_EXPR

项目 • 2024/02/07

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI**请求**操作的EXT_TDOP_SET_FROM_U64_EXPR子操作返回表示表达式值的类型化数据说明。

操作
对于此子操作，设置为 EXT_TDOP_SET_FROM_U64_EXPR。

标志
指定描述表达式值所在的目标内存的位标志。 有关这些标志的详细信息，请参阅 EXT_TYPED_DATA 。

InData
指定可选类型化数据，其地址可在 **InStrIndex** 指定的表达式中使用目标内存中的地址。 此地址由表达式用作伪寄存器 **$extin**。

OutData
接收表示表达式值的 DEBUG_TYPED_DATA 结构。

InStrIndex
指定要计算的表达式。 此表达式由默认表达式计算器计算。

地位
接收此子操作返回的状态代码。 这与 Request 返回的值相同。

注解

EXT_TDOP_SET_FROM_U64_EXPR是 EXT_TDOP 枚举中的值。

此子操作的参数是 EXT_TYPED_DATA 结构的成员。 此子操作不使用上一参数部分未列出的EXT_TYPED_DATA的成员，应设置为零。 上述参数部分中成员的说明指定了成员的用途。 有关更多详细信息，**请参阅**EXT_TYPED_DATA 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_GET_FIELD

项目 • 2024/02/06

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI Request 操作的EXT_TDOP_GET_FIELD子操作返回表示给定结构成员的类型化数据说明。

参数

操作
对于此子操作，设置为 EXT_TDOP_GET_FIELD。

InData
指定 DEBUG_TYPED_DATA 的实例，该实例描述需要其成员的结构。

OutData
接收所请求成员 **的** DEBUG_TYPED_DATA 实例。

InStrIndex
指定所请求成员的名称。 成员的名称是一个以点分隔的路径，可以包含子成员，例如 **mymember.mysubmember**。 此点分隔路径上的指针将自动取消引用。 但是，点运算符 (。 此处仍应使用)，而不是通常的 C 指针取消引用运算符 (->)。

地位
接收此子操作返回的状态代码。 这与 Request 返回的值相同。

注解

EXT_TDOP_GET_FIELD 是 EXT_TDOP 枚举中的值。

此子操作的参数是 EXT_TYPED_DATA 结构的成员。 此子操作不使用上一个参数部分未列出的EXT_TYPED_DATA成员，应设置为零。 上述"参数"部分中成员的说明指定了成员的用途。 有关详细信息，**请参阅**EXT_TYPED_DATA 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_EVALUATE

项目 • 2023/06/15

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI**请求**操作的**EXT_TDOP_EVALUATE**子操作返回表示表达式值的类型化数据说明。

参数

操作
对于此子操作，设置为 EXT_TDOP_EVALUATE。

标志
指定描述表达式值所在的目标内存的位标志。 有关这些标志的详细信息，请参阅 EXT_TYPED_DATA 。

InData
指定可选类型化数据，其值可在 InStrIndex 指定的表达式中使用。 表达式将此值用作伪寄存器 $extin。

OutData
接收表示表达式值的 DEBUG_TYPED_DATA 结构。

InStrIndex
指定要计算的表达式。 此表达式由默认表达式计算器计算。

地位
接收此子操作返回的状态代码。 这与 Request 返回的值相同。

注解

EXT_TDOP_EVALUATE 是 EXT_TDOP 枚举中的值。

此子操作的参数是 EXT_TYPED_DATA 结构的成员。 此子操作不使用上一参数部分未列出的EXT_TYPED_DATA的成员，应设置为零。 上述参数部分中成员的说明指定了成员的用途。 有关更多详细信息，**请参阅**EXT_TYPED_DATA 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_GET_TYPE_NAME

项目 • 2024/02/07

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI**请求**操作的EXT_TDOP_GET_TYPE_NAME子操作返回指定类型化数据的类型名称。

参数

操作
对于此子操作，设置为 EXT_TDOP_GET_TYPE_NAME。

InData
指定正在请求其类型名称的类型化数据。

StrBufferIndex
接收类型名称。

StrBufferChars
指定 ANSI 字符串缓冲区 **StrBufferIndex** 的大小（以字符为单位）。

StrCharsNeeded
接收类型名称的大小（以字符为单位）。

地位
接收此子操作返回的状态代码。 这与 Request 返回的值相同。

注解

EXT_TDOP_GET_TYPE_NAME 是 EXT_TDOP 枚举中的值。

此子操作的参数是 EXT_TYPED_DATA 结构的成员。 此子操作不使用上一参数部分未列出的EXT_TYPED_DATA的成员，应设置为零。 上述参数部分中成员的说明指定了成员的用途。 有关更多详细信息，**请参阅**EXT_TYPED_DATA 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_OUTPUT_TYPE_NAME

项目 • 2024/02/12

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI**请求**操作的**EXT_TDOP_OUTPUT_TYPE_NAME**子操作输出指定类型化数据的类型名称。

参数

操作
对于此子操作，设置为 EXT_TDOP_OUTPUT_TYPE_NAME。

InData
指定将打印其类型名称的类型化数据。

地位
接收此子操作返回的状态代码。 这与 **Request** 返回的值相同。

注解

类型名称将发送到调试器引擎的 输出回调。

EXT_TDOP_OUTPUT_TYPE_NAME 是 **EXT_TDOP** 枚举中的值。

此子操作的参数是 **EXT_TYPED_DATA** 结构的成员。 此子操作不使用上一参数部分未列出的EXT_TYPED_DATA的成员，应设置为零。 上述参数部分中成员的说明指定了成员的用途。 有关更多详细信息，**请参阅**EXT_TYPED_DATA 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_OUTPUT_SIMPLE_VALUE

项目 • 2024/02/07

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI**请求**操作的
EXT_TDOP_OUTPUT_SIMPLE_VALUE子操作输出指定类型化数据的值。

参数

操作
对于此子操作，设置为 EXT_TDOP_OUTPUT_SIMPLE_VALUE。

InData
指定将打印其值的类型化数据。

地位
接收此子操作返回的状态代码。 这与 Request 返回的值相同。

注解

该值已格式化，并发送到调试器引擎的 输出回调。

EXT_TDOP_OUTPUT_SIMPLE_VALUE 是 EXT_TDOP 枚举中的值。

此子操作的参数是 EXT_TYPED_DATA 结构的成员。 此子操作不使用上一参数部分未列出的EXT_TYPED_DATA的成员，应设置为零。 上述参数部分中成员的说明指定了成员的用途。 有关更多详细信息，**请参阅**EXT_TYPED_DATA 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_OUTPUT_FULL_VALUE

项目 • 2023/06/15

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI**请求**操作的**EXT_TDOP_OUTPUT_FULL_VALUE**子操作输出指定类型化数据的类型和值。

参数

操作
对于此子操作，设置为 EXT_TDOP_OUTPUT_FULL_VALUE。

InData
指定将打印其类型名称和值的类型化数据。

地位
接收此子操作返回的状态代码。 这与 **Request** 返回的值相同。

注解

类型名称和格式化值将发送到调试器引擎的 输出回调。 与 **EXT_TDOP_OUTPUT_SIMPLE_VALUE 相比，** EXT_TDOP_OUTPUT_FULL_VALUE打印有关该值的详细信息。 例如，取消引用指针，并且也会打印指针指向的值。

EXT_TDOP_OUTPUT_FULL_VALUE 是 **EXT_TDOP** 枚举中的值。

此子操作的参数是 **EXT_TYPED_DATA** 结构的成员。 此子操作不使用上一个参数部分未列出的EXT_TYPED_DATA成员，应设置为零。 上述“参数”部分中成员的说明指定了成员的用途。 有关详细信息，**请参阅EXT_TYPED_DATA** 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_HAS_FIELD

项目 • 2023/06/15

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI Request 操作的EXT_TDOP_HAS_FIELD子操作确定结构是否包含指定成员。

参数

操作
对于此子操作，设置为 EXT_TDOP_HAS_FIELD。

InData
指定检查成员是否存在的类型化数据。 首先检查类型化数据以查看它是否表示结构的实例，然后检查结构以查看它是否包含指定的成员。

InStrIndex
指定成员的名称。 名称是一个以点分隔的路径，可以包含子成员，例如 **mymember.mysubmember**。 此点分隔路径上的指针将自动取消引用。 但是，点运算符 (。 此处仍应使用)，而不是通常的 C 指针取消引用运算符 (->)。

地位
接收此子操作返回的状态代码。 这与 Request 返回的值相同。

如果类型化数据包含成员， **则** Status 接收S_OK。 如果类型化数据不包含成员， **则** Status 接收E_NOINTERFACE。 也可能返回其他错误值。

注解

EXT_TDOP_HAS_FIELD 是 EXT_TDOP 枚举中的值。

此子操作的参数是 EXT_TYPED_DATA 结构的成员。 此子操作不使用上一个参数部分未列出的EXT_TYPED_DATA成员，应设置为零。 上述"参数"部分中成员的说明指定了成员的用途。 有关详细信息， **请参阅**EXT_TYPED_DATA 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_GET_FIELD_OFFSET

项目 • 2023/06/15

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI Request 操作的
EXT_TDOP_GET_FIELD_OFFSET子操作返回结构中成员的偏移量。

参数

操作
对于此子操作，设置为 EXT_TDOP_GET_FIELD_OFFSET。

InData
指定类型化数据，该数据表示结构的实例，该结构包含正在请求其偏移量的成员。

InStrIndex
指定要请求其偏移量的成员的名称。 名称是一个以点分隔的路径，可以包含子成员。 例如，**mymember.mysubmember**。 此点分隔路径上的指针将自动取消引用。 但是，此处仍应使用点运算符 (.)，而不是常用的 C 指针取消引用运算符 (->)。

Out32
接收 结构实例中成员的偏移量。 这是 结构实例的开头和 成员之间的字节数。

地位
接收此子操作返回的状态代码。 这与 Request 返回的值相同。

注解

EXT_TDOP_GET_FIELD_OFFSET是 EXT_TDOP 枚举中的值。

此子操作的参数是 EXT_TYPED_DATA 结构的成员。 此子操作不使用上一参数部分未列出的EXT_TYPED_DATA的成员，应设置为零。 上述参数部分中成员的说明指定了成员的用途。 有关更多详细信息，**请参阅EXT_TYPED_DATA** 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_GET_ARRAY_ELEMENT

项目 • 2023/06/15

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI Request 操作的
EXT_TDOP_GET_ARRAY_ELEMENT子操作从数组返回元素。

参数

操作
对于此子操作，设置为 EXT_TDOP_GET_ARRAY_ELEMENT。

InData
指定正在请求其元素的数组。 InData 还可以指定指针，在这种情况下，它被视为数组。

OutData
从数组接收请求的元素。

In64
指定数组中元素的索引。

地位
接收此子操作返回的状态代码。 这与 Request 返回的值相同。

注解

EXT_TDOP_GET_ARRAY_ELEMENT 是 EXT_TDOP 枚举中的值。

此子操作的参数是 EXT_TYPED_DATA 结构的成员。 此子操作不使用上一个参数部分未列出的EXT_TYPED_DATA成员，应设置为零。 上述“参数”部分中成员的说明指定了成员的用途。 有关详细信息，**请参阅**EXT_TYPED_DATA 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_GET_DEREFERENCE

项目 • 2024/02/07

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI**请求**操作的**EXT_TDOP_GET_DEREFERENCE**子操作取消引用指针并返回它指向的值。

参数

操作
对于此子操作，设置为 EXT_TDOP_GET_DEREFERENCE。

InData
指定指向取消引用的指针。 **InData** 还可以指定数组，在这种情况下，将返回数组中的第一个元素。

OutData
接收指向的值。

地位
接收此子操作返回的状态代码。 这与 **Request** 返回的值相同。

注解

EXT_TDOP_GET_DEREFERENCE 是 **EXT_TDOP** 枚举中的值。

此子操作的参数是 **EXT_TYPED_DATA** 结构的成员。 此子操作不使用上一参数部分未列出的EXT_TYPED_DATA的成员，应设置为零。 上述参数部分中成员的说明指定了成员的用途。 有关更多详细信息，**请参阅**EXT_TYPED_DATA 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_GET_TYPE_SIZE

项目 • 2024/02/06

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI**请求**操作的**EXT_TDOP_GET_TYPE_SIZE**子操作返回指定类型化数据的大小。

参数

操作
对于此子操作，设置为 EXT_TDOP_GET_TYPE_SIZE。

InData
指定正在请求其大小的类型化数据。

Out32
接收类型化数据的大小（以字节为单位）。

地位
接收此子操作返回的状态代码。 这与 **Request** 返回的值相同。

注解

EXT_TDOP_GET_TYPE_SIZE 是 **EXT_TDOP** 枚举中的值。

此子操作的参数是 **EXT_TYPED_DATA** 结构的成员。 此子操作不使用上一个参数部分未列出的EXT_TYPED_DATA成员，应设置为零。 上述"参数"部分中成员的说明指定了成员的用途。 有关详细信息，**请参阅EXT_TYPED_DATA** 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_OUTPUT_TYPE_DEFINITION

项目 • 2024/02/06

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI**请求**操作的
EXT_TDOP_OUTPUT_TYPE_DEFINITION子操作打印指定类型化数据的类型定义。

参数

操作
对于此子操作，设置为 EXT_TDOP_OUTPUT_TYPE_DEFINITION。

InData
指定将打印其类型定义的类型化数据。

地位
接收此子操作返回的状态代码。 这与 Request 返回的值相同。

注解

类型的定义已格式化并发送到调试器引擎的 输出回调。

EXT_TDOP_OUTPUT_TYPE_DEFINITION 是 EXT_TDOP 枚举中的值。

此子操作的参数是 EXT_TYPED_DATA 结构的成员。 此子操作不使用上一个参数部分未列出的EXT_TYPED_DATA成员，应设置为零。 上述"参数"部分中成员的说明指定了成员的用途。 有关详细信息，**请参阅**EXT_TYPED_DATA 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_GET_POINTER_TO

项目 • 2023/06/15

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI Request 操作的EXT_TDOP_GET_POINTER_TO 子操作返回一个新的类型化数据说明，该说明表示指向指定类型化数据的指针。

参数

操作
对于此子操作，设置为 EXT_TDOP_GET_POINTER_TO。

InData
指定指针将返回到的原始类型化数据说明。

OutData
接收类型化数据说明，该说明表示指向 **InData** 指定的类型化数据的指针。

地位
接收此子操作返回的状态代码。 这与 Request 返回的值相同。

注解

EXT_TDOP_GET_POINTER_TO 是 EXT_TDOP 枚举中的值。

此子操作的参数是 EXT_TYPED_DATA 结构的成员。 此子操作不使用上一个参数部分未列出的EXT_TYPED_DATA成员，应设置为零。 上述"参数"部分中成员的说明指定了成员的用途。 有关详细信息，**请参阅**EXT_TYPED_DATA 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_SET_FROM_TYPE_ID_AND_U64

项目 • 2024/02/07

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI**请求**操作的 EXT_TDOP_SET_FROM_TYPE_ID_AND_U64子操作从数据类型和内存位置创建类型化数据说明。

参数

操作
对于此子操作，设置为 EXT_TDOP_SET_FROM_TYPE_ID_AND_U64。

标志
指定描述类型化数据值所在的目标内存的位标志。 有关这些标志的详细信息，请参阅 EXT_TYPED_DATA 。

InData
指定类型和内存位置。 可以手动创建 DEBUG_TYPED_DATA 结构的此实例，并使用所需的成员进行填充。 使用以下成员：

ModBase
指定包含 类型的模块基址的目标虚拟内存中的位置。

抵消
指定目标内存中数据的位置。 **Offset** 是虚拟内存地址，除非 **标志** 中存在指定 Offset 为物理内存地址的 **标志** 。

TypeId
指定类型的类型 ID。

OutData
接收类型化数据说明。

地位
接收此子操作返回的状态代码。 这与 Request 返回的值相同。

注解

EXT_TDOP_SET_FROM_TYPE_ID_AND_U64是 EXT_TDOP 枚举中的值。

此子操作的参数是 EXT_TYPED_DATA 结构的成员。 此子操作不使用上一参数部分未列出的EXT_TYPED_DATA的成员，应设置为零。 上述参数部分中成员的说明指定了成员的用途。 有关更多详细信息，**请参阅**EXT_TYPED_DATA 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

EXT_TDOP_SET_PTR_FROM_TYPE_ID_AND_U64

项目 • 2024/02/06

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI**请求**操作的 EXT_TDOP_SET_PTR_FROM_TYPE_ID_AND_U64子操作将创建一个类型化的数据说明，该说明表示指向具有指定类型的指定内存位置的指针。

参数

操作
对于此子操作，设置为 EXT_TDOP_SET_PTR_FROM_TYPE_ID_AND_U64。

标志
指定用于描述类型化数据值所在的目标内存的位标志。 有关这些标志的详细信息，**请参阅**EXT_TYPED_DATA 。

InData
指定类型和内存位置。 DEBUG_TYPED_DATA **结构的此** 实例可以手动创建并使用所需成员填充。 使用以下成员：

ModBase
指定目标虚拟内存中包含 该类型的模块的基址的位置。

抵消
指定目标内存中数据的位置。 **Offset** 是虚拟内存地址，除非标志中存在指定 **Offset** 为物理内存地址的**标志。**

TypeId
指定类型的类型 ID。

OutData
接收类型化数据说明，该说明表示指向内存位置和类型的指针。

地位
接收此子操作返回的状态代码。 这与 Request 返回的值相同。

注解

EXT_TDOP_SET_PTR_FROM_TYPE_ID_AND_U64 是 EXT_TDOP 枚举中的值。

此子操作的参数是 EXT_TYPED_DATA 结构的成员。 此子操作不使用上一个参数部分未列出的EXT_TYPED_DATA成员，应设置为零。 上述“参数”部分中成员的说明指定了成员的用途。 有关详细信息，**请参阅**EXT_TYPED_DATA 。

另请参阅

DEBUG_REQUEST_EXT_TYPED_DATA_ANSI

EXT_TDOP

EXT_TYPED_DATA

Request

ExtExtension

项目 • 2023/06/15

ExtExtension 类是表示 EngExtCpp 扩展库的 C++ 类的基类。

ExtExtension 类包含以下方法，子类可以使用这些方法：

初始化

Uninitialize

OnSessionActive

OnSessionInactive

OnSessionAccessible

OnSessionInaccessible

IsUserMode

IsKernelMode

IsLiveLocalUser

IsMachine32

IsCurMachine32

IsMachine64

IsCurMachine64

Is32On64

CanQueryVirtual

HasFullMemBasic

IsExtensionRemote

AreOutputCallbacksDmlAware

RequireUserMode

RequireKernelMode

GetNumUnnamedArgs

GetUnnamedArgStr

GetUnnamedArgU64

HasUnnamedArg

GetArgStr

GetArgU64

HasArg

HasCharArg

SetUnnamedArg

SetUnnamedArgStr

SetUnnamedArgU64

SetArg

SetArgStr

SetArgU64

GetRawArgStr

GetRawArgCopy

Out

警告

犯 错

谓词

Dml

DmlWarn

DmlErr

DmlVerb

DmlCmdLink

DmlCmdExec

RefreshOutputCallbackFlags

WrapLine

OutWrapStr

OutWrapVa

OutWrap

DemandWrap

AllowWrap

TestWrap

RequestCircleString

CopyCircleString

PrintCircleStringVa

PrintCircleString

SetAppendBuffer

AppendBufferString

AppendStringVa

AppendString

IsAppendStart

SetCallStatus

GetCachedSymbolTypeId

GetCachedFieldOffset

GetCachedFieldOffset

AddCachedSymbolInfo

GetExpr64

GetExprU64

GetExprS64

ThrowCommandHelp

ThrowInterrupt

ThrowOutOfMemory

ThrowContinueSearch

ThrowReloadExtension

ThrowInvalidArg

ThrowRemote

ThrowStatus

ThrowLastError

ExtExtension 类还包含以下子类可以使用的字段：

C++

```
class ExtExtension
{
public:
    USHORT  m_ExtMajorVersion;
    USHORT  m_ExtMinorVersion;
    ULONG  m_ExtInitFlags;
    ExtKnownStruct *  m_KnownStructs;
    ExtProvidedValue *  m_ProvidedValues;
    ExtCheckedPointer<IDebugAdvanced>  m_Advanced;
    ExtCheckedPointer<IDebugClient>  m_Client;
    ExtCheckedPointer<IDebugControl>  m_Control;
    ExtCheckedPointer<IDebugDataSpaces>  m_Data;
    ExtCheckedPointer<IDebugRegisters>  m_Registers;
    ExtCheckedPointer<IDebugSymbols>  m_Symbols;
    ExtCheckedPointer<IDebugSystemObjects>  m_System;
    ExtCheckedPointer<IDebugAdvanced2>  m_Advanced2;
    ExtCheckedPointer<IDebugAdvanced3>  m_Advanced3;
    ExtCheckedPointer<IDebugClient2>  m_Client2;
    ExtCheckedPointer<IDebugClient3>  m_Client3;
    ExtCheckedPointer<IDebugClient4>  m_Client4;
    ExtCheckedPointer<IDebugClient5>  m_Client5;
    ExtCheckedPointer<IDebugControl2>  m_Control2;
    ExtCheckedPointer<IDebugControl3>  m_Control3;
    ExtCheckedPointer<IDebugControl4>  m_Control4;
    ExtCheckedPointer<IDebugDataSpaces2>  m_Data2;
    ExtCheckedPointer<IDebugDataSpaces3>  m_Data3;
    ExtCheckedPointer<IDebugDataSpaces4>  m_Data4;
    ExtCheckedPointer<IDebugRegisters2>  m_Registers2;
    ExtCheckedPointer<IDebugSymbols2>  m_Symbols2;
    ExtCheckedPointer<IDebugSymbols3>  m_Symbols3;
```

```
    ExtCheckedPointer<IDebugSystemObjects2>  m_System2;
    ExtCheckedPointer<IDebugSystemObjects3>  m_System3;
    ExtCheckedPointer<IDebugSystemObjects4>  m_System4;
    ULONG  m_OutputWidth;
    ULONG  m_ActualMachine;
    ULONG  m_Machine;
    ULONG  m_PageSize;
    ULONG  m_PtrSize;
    ULONG  m_NumProcessors;
    ULONG64  m_OffsetMask;
    ULONG  m_DebuggeeClass;
    ULONG  m_DebuggeeQual;
    ULONG  m_DumpFormatFlags;
    bool  m_IsRemote;
    bool  m_OutCallbacksDmlAware;
    ULONG  m_OutMask;
    ULONG  m_CurChar;
    ULONG  m_LeftIndent;
    bool  m_AllowWrap;
    bool  m_TestWrap;
    ULONG  m_TestWrapChars;
    PSTR  m_AppendBuffer;
    ULONG  m_AppendBufferChars;
    PSTR  m_AppendAt;
};
```

成员

m_ExtMajorVersion
扩展库的主版本号。 这应由 Initialize 方法设置。 如果未设置，则默认为 1。

m_ExtMinorVersion
扩展库的次要版本号。 这应由 Initialize 方法设置。 如果未设置，则默认为 0 (零) 。

m_ExtInitFlags
DebugExtensionInitialize 的 DbgEng 扩展初始化标志。

m_KnownStructs
扩展库能够格式化输出的 ExtKnownStruct 结构的数组。 此成员应由 Initialize 方法设置，并且不应在此方法返回后进行更改。

如果m_KnownStructs不为 NULL，则数组中最后一个 ExtKnownStruct 结构的 TypeName 成员必须为 NULL。

为输出设置目标结构的格式时，如果结构类型的名称与此数组中 ExtKnownStruct 结构之一的 TypeName 成员匹配，则会调用 Method 成员中指定的回调函数来执行格式设置。

m_ProvidedValues
ExtProvidedValue 结构的数组，其中列出了扩展库可以提供值的伪寄存器。 此成员应由 Initialize 方法设置，并且不应在此方法返回后进行更改。

如果 m_ProvidedValues 不为 NULL，则数组中最后一个 ExtProvidedValue 结构的 ValueName 成员必须为 NULL。

计算伪寄存器时，如果伪寄存器的名称与此数组中**某个** ExtProvidedValue 结构的 ValueName 成员匹配，则调用 Method 成员中指定的回调函数来计算伪寄存器。

m_Advanced
扩展库可以使用的客户端对象的 IDebugAdvanced 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。

m_Client
扩展库可以使用的客户端对象的 IDebugClient 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。

m_Control
扩展库可以使用的客户端对象的 IDebugControl 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。

m_Data
扩展库可以使用的客户端对象的 IDebugDataSpaces 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。

m_Registers
扩展库可以使用的客户端对象的 IDebugRegisters 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。

m_Symbols
扩展库可以使用的客户端对象的 IDebugSymbols 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。

m_System
扩展库可以使用的客户端对象的 IDebugSystemObjects 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。

m_Advanced2
扩展库可以使用的客户端对象的 IDebugAdvanced2 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_Advanced3
扩展库可以使用的客户端对象的 IDebugAdvanced3 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_Client2
扩展库可以使用的客户端对象的 IDebugClient2 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_Client3
扩展库可以使用的客户端对象的 IDebugClient3 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_Client4
扩展库可以使用的客户端对象的 IDebugClient4 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_Client5
扩展库可以使用的客户端对象的 IDebugClient5 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_Control2
扩展库可以使用的客户端对象的 IDebugControl2 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_Control3
扩展库可以使用的客户端对象的 IDebugControl3 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_Control4
扩展库可以使用的客户端对象的 IDebugControl4 接口指针。 它在调用外部调用的扩展

方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_Data2
可由扩展库使用的客户端对象的 **IDebugDataSpaces2** 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_Data3
扩展库可以使用的客户端对象的 **IDebugDataSpaces3** 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_Data4
扩展库可以使用的客户端对象的 IDebugDataSpaces4 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_Registers2
可由扩展库使用的客户端对象的 **IDebugRegisters2** 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_Symbols2
扩展库可以使用的客户端对象的 **IDebugSymbols2** 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_Symbols3
扩展库可以使用的客户端对象的 **IDebugSymbols3** 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_System2
扩展库可以使用的客户端对象的 **IDebugSystemObjects2** 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_System3
扩展库可以使用的客户端对象的 **IDebugSystemObjects3** 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 ExtProvideValueMethod。 此接口可能并非在所有版本的调试器引擎中都可用。

m_System4
扩展库可以使用的客户端对象的 IDebugSystemObjects4 接口指针。 它在调用外部调用的扩展方法期间有效，例如，执行扩展命令、调用 *ExtKnownStructMethod* 和 **ExtProvideValueMethod**。 此接口可能并非在所有版本的调试器引擎中都可用。

m_PtrSize
当前目标上的指针的大小。 如果目标使用 32 位指针， **m_PtrSize** 为 4。 如果目标使用 64 位指针， **m_PtrSize** 为 8。

m_AppendBuffer
用于将字符串从扩展库返回到引擎的字符缓冲区。 此缓冲区**的大小** **m_AppendBufferChars**。 **AppendBufferString**、**AppendStringVa** 和 **AppendString** 方法可用于将字符串写入此缓冲区。

m_AppendBufferChars
缓冲区 **m_AppendBuffer**的大小（以字符为单位）。

特定异常

项目 • 2023/06/15

下表列出了特定异常筛选器的异常代码。

异常代码	例外	头文件或值
STATUS_ACCESS_VIOLATION	访问冲突	NtStatus.h
STATUS_ASSERTION_FAILURE	断言失败	NtStatus.h
STATUS_APPLICATION_HANG	应用程序挂起	0xCFFFFFFF
STATUS_BREAKPOINT	中断指令异常	NtStatus.h
STATUS_CPP_EH_EXCEPTION	C++ 异常处理异常	0xE06D7363
STATUS_CLR_EXCEPTION	公共语言运行时 (CLR) 异常	0xE0434f4D
DBG_CONTROL_BREAK	Ctrl+Break 异常	NtStatus.h
DBG_CONTROL_C	CTRL+C 异常	NtStatus.h
STATUS_DATATYPE_MISALIGNMENT	数据未对齐	NtStatus.h
DBG_COMMAND_EXCEPTION	调试器命令异常	NtStatus.h
STATUS_GUARD_PAGE_VIOLATION	保护页冲突	NtStatus.h
STATUS_ILLEGAL_INSTRUCTION	非法指令	NtStatus.h
STATUS_IN_PAGE_ERROR	页内 I/O 错误	NtStatus.h
STATUS_INTEGER_DIVIDE_BY_ZERO	整数除以零	NtStatus.h
STATUS_INTEGER_OVERFLOW	整数溢出	NtStatus.h
STATUS_INVALID_HANDLE	无效句柄	NtStatus.h
STATUS_INVALID_LOCK_SEQUENCE	锁序列无效	NtStatus.h
STATUS_INVALID_SYSTEM_SERVICE	系统调用无效	NtStatus.h
STATUS_PORT_DISCONNECTED	端口已断开连接	NtStatus.h
STATUS_SINGLE_STEP	单步异常	NtStatus.h
STATUS_STACK_BUFFER_OVERRUN	堆栈缓冲区溢出	NtStatus.h
STATUS_STACK_OVERFLOW	堆栈溢出	NtStatus.h

异常代码	例外	头文件或值
STATUS_VERIFIER_STOP	应用程序验证程序停止	NtStatus.h
STATUS_WAKE_SYSTEM_DEBUGGER	唤醒调试器	NtStatus.h

WdbgExts 函数

项目 • 2023/06/15

wdbgexts.h 头文件包含以下函数的原型。 这些函数对 32 位和 64 位扩展使用相同的原型：

GetContext

SetContext

CheckControlC

GetCurrentProcessAddr

GetCurrentProcessHandle

GetCurrentThreadAddr

GetDebuggerCacheSize

GetDebuggerData

Disasm

dprintf

GetExpression

GetExpressionEx

GetInputLine

Ioctl

GetKdContext

ReadMemory

SearchMemory

WriteMemory

ReadMsr

WriteMsr

GetPebAddress

ReadPhysical

ReadPhysicalWithFlags

WritePhysical

WritePhysicalWithFlags

GetTebAddress

StackTrace

GetSymbol

ReloadSymbols

GetSetSympath

TranslateVirtualToPhysical

wdbgexts.h 头文件包含以下函数的原型。 这些函数具有 32 位和 64 位扩展的不同原型：

ReadControlSpace

ReadControlSpace64

ReadTypedControlSpace32

ReadTypedControlSpace64

WriteControlSpace

ReadIoSpace

ReadIoSpace64

ReadIoSpaceEx

ReadIoSpaceEx64

WriteIoSpace

WriteIoSpace64

WriteIoSpaceEx

WriteIoSpaceEx64

SetThreadForOperation

SetThreadForOperation64

wdbgexts.h 头文件包含以下函数的原型。 这些函数只能在 64 位扩展中使用：

GetFieldData

GetFieldOffset

GetFieldValue

GetShortField

ReadField

ReadListEntry

ReadPointer

WritePointer

IsPtr64

ReadPtr

GetTypeSize

InitTypeRead

InitTypeReadPhysical

ListType

使用 DML 自定义调试器输出

项目 • 2023/11/15

调试器标记语言 (DML) 提供了一种机制来增强调试器和扩展的输出。 与 HTML 类似，调试器的标记支持允许输出以标记的形式包含显示指令和额外的非显示信息。 调试器用户界面（如 WinDbg）会解析 DML 中提供的额外信息，以增强信息的显示并提供新行为，例如网格显示和排序。 本主题介绍如何使用 DML 自定义调试输出。 有关在调试器中启用和使用 DML 的一般信息，请参阅 使用调试器标记语言。

DML 在 Windows 10 及更高版本中可用。

DML 概述

DML 的主要优势是提供链接到调试器输出中相关信息的功能。 主要 DML 标记之一是 <链接> 标记，它允许输出生成者指示可以通过链接的所述操作访问与输出片段相关的信息。 与 Web 浏览器中的 HTML 链接一样，这允许用户导航超链接信息。

提供超链接内容的好处是，它可用于增强调试器和调试器扩展功能的可发现性。 调试器及其扩展包含大量功能，但可能很难确定用于不同方案的相应命令。 用户必须仅知道哪些命令可用于特定方案。 用户和内核调试之间的差异进一步增加了复杂性。 这通常意味着许多用户不知道可以帮助他们的调试命令。 DML 链接使任意调试命令能够包装在备用演示文稿中，例如描述性文本、可单击的菜单系统或链接的帮助。 使用 DML，可以增强命令输出，以引导用户获取与手头任务相关的其他相关命令。

调试器 DML 支持

- WinDbg 中的命令窗口支持所有 DML 行为，并显示颜色、字体样式和链接。
- 控制台调试器 （ntsd、cdb 和 kd） 仅支持 DML 的颜色属性，仅在启用了颜色模式的真正控制台中运行时，才支持 。
- 具有重定向 I/O、ntsd –d 或 remote.exe 会话的调试器不会显示任何颜色。

DML 内容规范

DML 不是完整的呈现语言（如 HTML）。 DML 特意非常简单，只有几个标记。

由于并非所有调试器工具都支持格式文本，因此 DML 旨在允许在 DML 和纯文本之间进行简单翻译。 这允许 DML 在所有现有调试器工具中运行。 可以轻松支持颜色等效果，因为删除它们不会删除包含实际信息的文本。

DML 不是 XML。 DML 不会尝试携带语义或结构化信息。 如上所述，DML 和纯文本之间必须有一个简单的映射，因此，DML 标记都是可丢弃的。

DML 不可扩展;所有标记都已预定义并经过验证，可跨所有现有调试器工具运行。

标记结构

与 XML 类似，DML 标记以起始 <标记名 [args]> 和以下 </tagname> 的形式提供。

特殊字符

DML 内容大致遵循特殊字符的 XML/HTML 规则。 字符 &、 <和 > “是特殊的，不能在纯文本中使用。 等效的转义版本为 &、 <、 > 和 ”。 例如，以下文本：

“Alice & Bob Think 3 < 4”

将转换为以下 DML。

```text
"Alice & Bob think 3 &lt 4"
```

C 编程语言格式字符

与 XML/HTML 规则的显著区别在于，DML 文本可以包含 C 编程语言流样式格式字符，例如 \b、\t、\r 和 \n。 这是为了支持与现有调试器文本生产和使用的兼容性。

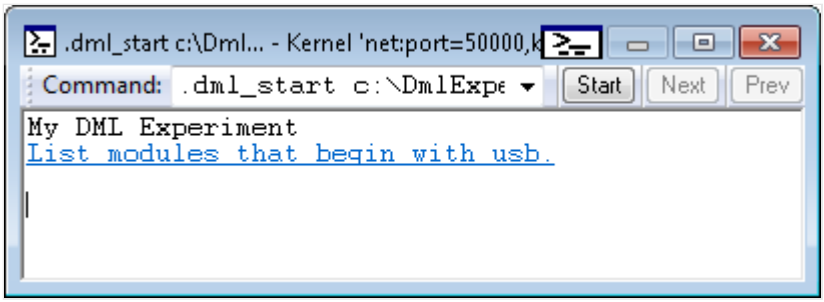
示例 DML

假设文件 C:\Dml_Experiment.txt 包含以下行。

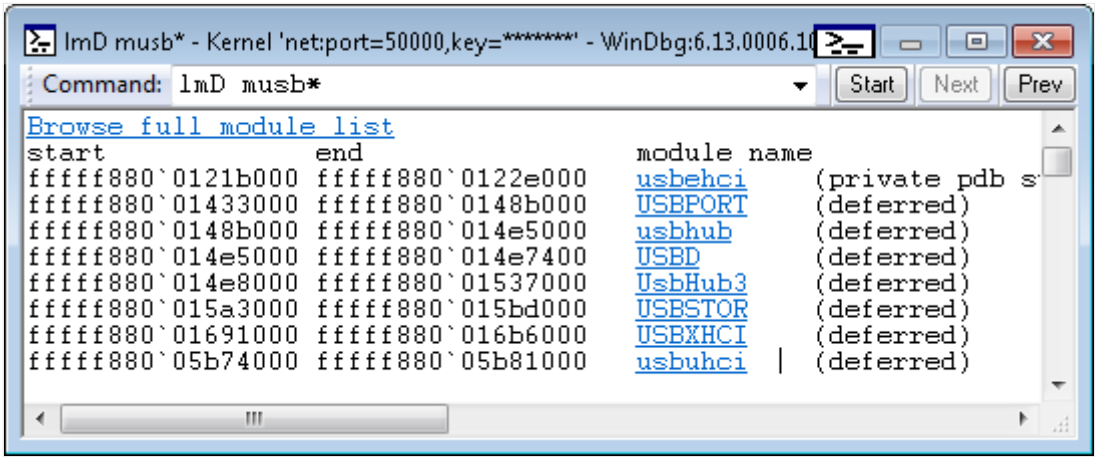
```text
My DML Experiment
<link cmd="lmD musb*">List modules that begin with usb.</link>
```

以下命令在“命令浏览器”窗口中显示文本和链接。

```dbgcmd
.browse .dml_start c:\Dml_Experiment.txt
```



如果单击 **以 usb 开头的“列出模块”** 链接，将看到类似于下图的输出。



DML 中的右键单击行为

DML 中提供右键单击行为。 此示例演示如何使用 <altlink> 定义右键单击行为，以发送断点 **bp (Set 断点)** 命令，并通过定期单击发送 **u (Unassemble)** 。

text

```
<link cmd="u MyProgram!memcpy">
<altlink name="Set Breakpoint (bp)" cmd="bp MyProgram!memcpy" />
u MyProgram!memcpy
</link>
```

DML 标记参考

<链接>

<link [name="text"] [cmd="debugger_command"][alt="悬停显示文本"] [section="name"]>链接文本</link>

链接标记是 DML 中的基本超链接机制。 它指示支持 DML 演示文稿的用户界面将链接文本显示为可单击的链接。 单击具有 cmd 规范的链接时，将执行调试器命令，其输出应替换当前输出。

名称和节参数允许在命名链接之间导航，类似于 HTML 的名称<>和 #name 支持。 在 UI 上单击具有 section 参数的链接时，将扫描名为 且名称匹配的链接，并将该链接滚动到视图中。 这允许链接指向同一页 (的不同部分或新页面) 的特定部分。 DML 的节名称是分开的，以避免必须定义允许在命令字符串末尾使用节名称的新语法。

转换为纯文本会删除标记。

示例

```
text
```

```
<b> Handy Links </b>
<link cmd="!dml_proc">Display process information with DML rendering.</link>
<link cmd="kM">Display stack information with DML rendering.</link>
```

示例

此示例演示如何使用 alt 属性创建将鼠标悬停在 DML 链接上时显示的文本。

```
text
```

```
<b>Hover Example</b>
<link cmd="lmD" alt="This link will run the list modules command and display
the output in DML format">LmD</link>
```

<altlink>

<altlink [name="text"] [cmd="debugger_command"] [section="name"]>alt link text</altlink>

<altlink> 标记提供右键单击行为，在 DML 中可用。 单击具有 cmd 规范的链接时，将执行调试器命令，其输出应替换当前输出。 <altlink> 选项卡通常与链接>标记配对，<以支持常规和右键单击行为。

转换为纯文本会删除标记。

示例

此示例演示如何使用 <altlink> 定义右键单击行为，以发送断点 **bp (Set 断点)** 命令，并使用常规单击发送 **u (Unassemble)** 。

```
text
```

```
<link cmd="u MyProgram!memcpy">
<altlink name="Set Breakpoint (bp)" cmd="bp MyProgram!memcpy" />
```

```
u MyProgram!memcpy
</link>
```

<Exec>

<exec cmd="debugger_command">描述性文本</exec>

exec 标记类似于链接标记，因为描述性文本应显示为可单击的项。 但是，在命令浏览器窗口中使用 exec 标记时，执行给定命令而不替换当前输出，此标记提供了一种通过单击从菜单执行命令的方法。

转换为纯文本会删除标记。

示例

此示例演示如何使用常规单击定义两个命令。

text

```
<b>Exec Sample</b>
<exec cmd="!dml_proc">Display process information with DML rendering.</exec>
<exec cmd="kM">Display stack information with DML rendering.</exec>
```

<B>

粗体文本

此标记请求加粗。 、 和 <u> 可以嵌套为混合属性。

转换为纯文本会删除标记。

示例

此示例演示如何加粗文本。

text

```
<b>This is bold Text</b>
```

<我>

斜体文本

此标记请求斜体。 、 和 <u> 可以嵌套为混合属性。

转换为纯文本会删除标记。

示例

此示例演示如何将文本斜体化。

```
text
<i>This is italicized Text</i>
```

<美国>

<u>带下划线的文本</u>

此标记请求带下划线的文本。 <b>、<i> 和 <u> 可以嵌套为混合属性。

转换为纯文本会删除标记。

示例

此示例演示如何为文本添加下划线。

```
text
<u>This is underlined Text</u>
```

示例

此示例演示如何将标记合并为加粗、下划线和斜体文本。

```
text
<b><u><i>This is bold, underlined and italizized text. </i></u></b>
```

<col>

<col fg="name" bg="name">text</col>

请求文本的前景色和背景色。 颜色作为已知颜色的名称而不是绝对值提供，因为这样客户就可以控制他们看到的颜色类型。 当前颜色名称 (默认值仅适用于 WinDbg) 。

前台和背景元素标记

设置	**说明/示例**

wbg - Windows 背景 wfg - Windows 前台	默认窗口背景和前景色。 默认为窗口和窗口文本的系统颜色。 <col fg="wfg" bg="wbg"> 这是标准的前景 / 背景文本 </col>
clbg - 当前行前景 clfg - 当前线条背景	当前线条背景和前景色。 默认为突出显示和突出显示文本的系统颜色。 <col fg="clfg" bg="clbg"> 测试文本 - Current Line</col>
empbg - 强调的背景 emphfg - 强调的前景	强调的文本。 默认为浅蓝色。 <col fg="empfg" bg="empbg"> 这是强调前景 / 背景文本 </col>
subbg - subdued 背景 subfg- Subdued 前台	subdued text。 默认为非活动描述文字文本和非活动字幕的系统颜色。 <col fg="subfg" bg="subbg"> 这是 subdued 前台 / 背景文本 </col>
normbg - 普通背景 normfg - 正常前景	普通 <col fg="normfg" bg="normbg"> 测试文本 - Normal (normfg / normbg) </col>
warnbg - 警告背景 warnfg - 警告前景	警告 <col fg="warnfg" bg="warnbg"> 测试文本 - 警告 (warnfg / warnbg) </col>
errbg - 错误背景 errfg - 前台错误	错误 <col fg="errfg" bg="errbg"> 测试文本 - 错误 (errfg / errbg) </col>
verbbg - 详细背景 verbfg - 详细前景	详细 <col fg="verbfg" bg="verbbg"> 测试文本 - Verbose (verbfg / verbbg) </col>

源代码单元素标记

srcnum - 源数值常量	源元素颜色。 <col fg="srcnum" bg="wbg"> 测试文本 - srcnum </col>
srcchar - 源字符常量	<col fg="srcchar" bg="wbg"> 测试文本 - srcchar </col>
srcstr - 源字符串常量	<col fg="srcstr" bg="wbg"> 测试文本 - srcstr </col>
srcid -源标识符	<col fg="srcid " bg="wbg"> 测试文本 - srcid </col>

srckw - Keyword	<col fg="srckw" bg="wbg"> 测试文本 - srckw </col>
srcpair - 源大括号或匹配的符号对	<col fg="srcpair" bg="empbbg"> 测试文本 - srcpair </col>
srccmnt - 源注释	<col fg="srccmnt" bg="wbg"> 测试文本 - srccmnt </col>
srcdrct - Source 指令	<col fg="srcdrct" bg="wbg"> 测试文本 - srcdrct </col>
srcspid - 源特殊标识符	<col fg="srcspid" bg="wbg"> 测试文本 - srcspid </col>
srcannot - 源注释	<col fg="srcannot" bg="wbg"> Test Text - srcannot </col>
已更改 - 已更改数据	用于自上一个停止点以来已更改的数据，例如 WinDbg 中的更改寄存器。 默认为红色。 <col fg="changed" bg="wbg"> 测试文本 - Changed</col>

DML 示例代码

此示例代码演示了以下内容。

- 调用调试命令
- 实现右键单击命令
- 实现将鼠标悬停在文本上
- 使用颜色和格式文本

XML

```
<col fg="srckw" bg="wbg"> <b>
*******************************************************
*** Example debug commands for crash dump analysis ****
*******************************************************
</b></col>
<col fg="srcchar" bg="wbg"><i>
**** Hover over commands for additional information ****
        **** Right-click for command help ****
</i></col>

<col fg="srccmnt" bg="wbg"><b>*** Common First Steps for Crash Dump Analysis
***</b> </col>
<link cmd=".symfix" alt="Set standard symbol path using
.symfix">.symfix<altlink name="Help about .symfix" cmd=".hh .symfix" />
</link> - Set standard symbol path
```

```
<link cmd=".sympath+ C:\Symbols" alt="This link adds additional symbol
directories">.sympath+ C:\Symbols<altlink name="Help for .sympath" cmd=".hh
.sympath" /> </link> - Add any additional symbol directories, for example
C:\Symbols
<link cmd=".reload /f" alt="This link reloads symbols">.reload /f<altlink
name="Help for .reload" cmd=".hh .reload" /> </link> - Reloads symbols to
make sure they are in good shape
<link cmd="!analyze -v" alt="This link runs !analyze with the verbose
option">!analyze -v<altlink name="Help for !analyze" cmd=".hh !analyze" />
</link> - Run !analyze with the verbose option
<link cmd="vertarget" alt="This link runs checks the target
version">vertarget<altlink name="Help for vertarget" cmd=".hh vertarget" />
</link> - Check the target version
<link cmd="version" alt="This link displays the versions in
use">version<altlink name="Help for version" cmd=".hh version" /></link> -
Display the versions in use
<link cmd=".chain /D" alt="This link runs .chain">.chain /D<altlink
name="Help for .chain" cmd=".hh .chain" /></link> - Use the .chain /D
command to list debugger extensions
<link cmd="kM" alt="This link displays the stack backtrace using
DML">kD<altlink name="Help for k" cmd=".hh k, kb, kc, kd, kp, kP, kv
(Display Stack Backtrace)" /> </link> - Display the stack backtrace using
DML rendering
<link cmd="lmD" alt="This link will run the list modules command and display
the output in DML format">LmD<altlink name="Help for lmD" cmd=".hh lm" />
</link> - List modules command and display the output in DML format
<link cmd=".help /D" alt="Display help for commands">.help /D <altlink
name="Help for .dot commands" cmd=".hh commands" /></link> - Display help
for commands in WinDbg
<link cmd=".hh" alt="Start help">.hh<altlink name="Debugger Reference
Help".hh Contents" cmd=".hh Debugger Reference" /></link> - Start help

<col fg="srccmnt" bg="wbg"><b>*** Registers and Context ***</b></col>
<link cmd="r" alt="This link displays registers">r<altlink name="Help about
r command" cmd=".hh r" /></link>  - Display registers
<link cmd="dt nt!_CONTEXT" alt="This link displays information about
nt_CONTEXT">dt nt!_CONTEXT<altlink name="Help about the dt command" cmd=".hh
dt" /></link> - Display information about nt_CONTEXT
<link cmd="dt nt!_PEB" alt="This link calls the dt command to display
nt!_PEB">dt nt!_PEB<altlink name="Help about dt command" cmd=".hh dt" />
</link> - Display information about the nt!_PEB
<link cmd="ub" alt="This link unassembles backwards">ub<altlink name="Help
about ub command" cmd=".hh u, ub, uu (Unassemble)" /></link> - Unassemble
Backwards

<col fg="srcchar" bg="wbg"><i>
**** Note: Not all of the following commands will work with all crash dump
data ****
</i></col>
<col fg="srccmnt" bg="wbg"><b>*** Device Drivers ***</b></col>
<link cmd="!devnode 0 1" alt="This link displays the devnodes">!devnode 0
1<altlink name="Help about !devnode command" cmd=".hh !devnode" /></link> -
Display devnodes
<link cmd=".load wdfkd.dll;!wdfkd.help" alt="Load wdfkd extensions and
display help">.load wdfkd.dll;!wdfkd.help<altlink name="Help about the wdfkd
```

```
extensions" cmd=".hh !wdfkd" /></link> - Load wdfkd extensions and display
help
<link cmd="!wdfkd.wdfldr" alt="This link displays
!wdfkd.wdfldr">!wdfkd.wdfldr<altlink name="Help about !wdfkd.wdfldr"
cmd=".hh !wdfkd.wdfldr" /></link>  - Display WDF framework driver loader
information
<link cmd="!wdfkd.wdfumtriage" alt="This link displays
!wdfkd.umtriage">!wdfkd.umtriage<altlink name="Help about !wdfkd.umtriage"
cmd=".hh !wdfkd_wdfumtriage" /></link> - Display WDF umtriage driver
information

<col fg="srccmnt" bg="wbg"><b>*** IRPs and IRQL ***</b></col>
<link cmd="!processirps" alt="This link displays process
IRPs">!processirps<altlink name="Help about !processirps command" cmd=".hh
!processirps" /></link> - Display process IRPs
<link cmd="!irql" alt="This link displays !irql">!irql<altlink name="Help
about !irql command" cmd=".hh !irql" /></link> - Run !irql

<col fg="srccmnt" bg="wbg"><b>*** Variables and Symbols ***</b></col>
<link cmd="dv" alt="This link calls the dv command">dv<altlink name="Help
about dv command" cmd=".hh dv" /></link> - Display the names and values of
all local variables in the current scope

<col fg="srccmnt" bg="wbg"><b>*** Threads, Processes, and Stacks ***</b>
</col>
<link cmd="!threads" alt="This link displays threads">!threads<altlink
name="Help about the !threads command" cmd=".hh !threads" /></link> -
Display threads
<link cmd="!ready 0xF" alt="This link runs !ready 0xF">!ready 0xF<altlink
name="Help about the !ready command" cmd=".hh !ready" /></link> - Display
threads in the ready state
<link cmd="!process 0 F" alt="This link runs !process 0 F ">!process 0
F<altlink name="Help about the !process command" cmd=".hh !process" />
</link> - Run !process 0 F
<link cmd="!stacks 2" alt="This link displays stack information using
!stacks 2 ">!stacks 2<altlink name="Help about the !stacks command" cmd=".hh
!stacks" /></link> - Display stack information using !stacks 2
<link cmd=".tlist" alt="This link displays a process list using TList
">tlist<altlink name="Help about the TList command" cmd=".hh .tlist" />
</link> - Display a process list using tlist
<link cmd="!process" alt="This link displays process ">!process<altlink
name="Help about the !process command" cmd=".hh !process" /></link> -
Display process information
<link cmd="!dml_proc" alt="This link displays process information with DML
rendering.">!dml_proc<altlink name="Help about the !dml_proc command"
cmd=".hh !dml_proc" /></link> - Display process information with DML
rendering
```

此示例代码演示了颜色和格式标记的使用。

XML

```
*** Text Tag Examples ****

<b>This is bold text</b>
<u>This is underlined text</u>
<i>This is italizized text</i>
<b><u><i>This is bold, underlined and italizized text</i></u></b>

<b>Color Tag Examples</b>
<col fg="wfg" bg="wbg"> This is standard foreground / background text </col>
<col fg="empfg" bg="empbg"> This is emphasis foreground / background text
</col>
<col fg="subfg" bg="subbg"> This is subdued foreground / background text
</col>
<col fg="clfg" bg="clbg"> Test Text - Current Line</col>

<b>Other Tags Sets</b>
<col fg="normfg" bg="normbg"> Test Text - Normal (normfg / normbg) </col>
<col fg="warnfg" bg="warnbg"> Test Text - Warning (warnfg / warnbg) </col>
<col fg="errfg" bg="errbg"> Test Text - Error (errfg / errbg) </col>
<col fg="verbfg" bg="verbbg"> Test Text - Verbose (verbfg / verbbg) </col>

<b>Changed Text Tag Examples</b>
<col fg="changed" bg="wbg"> Test Text - Changed</col>

<b>Source Tags - using wbg background</b>
<col fg="srcnum" bg="wbg"> Test Text - srcnum  </col>
<col fg="srcchar" bg="wbg"> Test Text - srcchar  </col>
<col fg="srcstr" bg="wbg"> Test Text - srcstr  </col>
<col fg="srcid " bg="wbg"> Test Text - srcid   </col>
<col fg="srckw" bg="wbg"> Test Text - srckw </col>
<col fg="srcpair" bg="empbbg"> Test Text - srcpair </col>
<col fg="srccmnt" bg="wbg"> Test Text - srccmnt  </col>
<col fg="srcdrct" bg="wbg"> Test Text - srcdrct </col>
<col fg="srcspid" bg="wbg"> Test Text - srcspid </col>
<col fg="srcannot" bg="wbg"> Test Text - srcannot </col>

<b>Source Tags - using empbg background</b>
<col fg="srcnum" bg="empbg"> Test Text - srcnum  </col>
<col fg="srcchar" bg="empbg"> Test Text - srcchar  </col>
<col fg="srcstr" bg="empbg"> Test Text - srcstr  </col>
<col fg="srcid " bg="empbg"> Test Text - srcid   </col>
<col fg="srckw" bg="empbg"> Test Text - srckw </col>
<col fg="srcpair" bg="empbbg"> Test Text - srcpair </col>
<col fg="srccmnt" bg="empbg"> Test Text - srccmnt  </col>
<col fg="srcdrct" bg="empbg"> Test Text - srcdrct </col>
<col fg="srcspid" bg="empbg"> Test Text - srcspid </col>
<col fg="srcannot" bg="empbg"> Test Text - srcannot </col>

<b>Source Tags - using subbg background</b>
<col fg="srcnum" bg="subbg"> Test Text - srcnum  </col>
<col fg="srcchar" bg="subbg"> Test Text - srcchar  </col>
<col fg="srcstr" bg="subbg"> Test Text - srcstr  </col>
<col fg="srcid " bg="subbg"> Test Text - srcid   </col>
<col fg="srckw" bg="subbg"> Test Text - srckw </col>
```

```
<col fg="srcpair" bg="subbg"> Test Text - srcpair </col>
<col fg="srccmnt" bg="subbg"> Test Text - srccmnt  </col>
<col fg="srcdrct" bg="subbg"> Test Text - srcdrct </col>
<col fg="srcspid" bg="subbg"> Test Text - srcspid </col>
<col fg="srcannot" bg="subbg"> Test Text - srcannot </col>
```

对 dbgeng 接口的 DML 添加

调试器引擎和扩展 API 提供了一个接口，用于使用调试器引擎创建自定义应用程序。 还可以编写将在 WinDbg、KD、CDB 和 NTSD 中运行的自定义扩展。 有关详细信息，请参阅 编写 DbgEng 扩展。 本部分介绍调试器引擎接口的可用 DML 增强功能。

dbgeng 已有一组处理输入方法和输出接口的文本，使用 DML 只需要指定输入和输出文本中携带的内容类型。

向 dbgeng 提供 DML 内容

输出控制标志DEBUG_OUTCTL_DML指示 dbgeng 方法生成的文本应作为 DML 内容处理。 如果未提供此标志，则文本被视为纯文本上下文。 DEBUG_OUTCTL_DML可以与以下方法一起使用。

- IDebugControl4：：ControlledOutput
- IDebugControl4：：ControlledOutputVaList
- IDebugControl4：：ControlledOutputWide
- IDebugControl4：：ControlledOutputVaListWide

对于有效字符，给定的文本必须遵循 DML 规则。

所有输出例程都已得到增强，以允许新的格式说明符 %[h|w]Y{t}。 此格式说明符具有字符串指针作为参数，指示给定文本是纯文本，应在输出处理期间转换为 DML 格式。 这为调用方提供了一种在 DML 内容中包含纯文本的简单方法，而无需自行预转换为 DML 格式。 h 和 w 限定符指示 ANSI 或 Unicode 文本，与 %s 一样。

下表汇总了 %Y 格式说明符的使用。

%Y{t}：带引号的字符串。 如果输出格式 (第一个参数) 为 DML，则会将文本转换为 DML。

%Y{T}：带引号的字符串。 无论输出格式如何，始终将文本转换为 DML。

%Y{s}：无引号字符串。 如果输出格式 (第一个参数) 为 DML，则会将文本转换为 DML。

%Y{S}：无引号字符串。 无论输出格式如何，始终将文本转换为 DML。

%Y{as}：ULONG64。 添加一个空字符串或 9 个字符的间距，以填充调试器格式化指针字段的高 32 位部分。 额外空间输出 9 个空格，其中包括上 8 个零加上 ' 字符。

%Y{ps}：ULONG64。 用于填充调试器格式的指针字段的额外空间 (包括上 8 个零加上字符) 。

%Y{l}：ULONG64。 地址作为源行信息。

此代码片段演示了 %Y 格式说明符的用法。

C++

```
HRESULT CALLBACK testout(_In_ PDEBUG_CLIENT pClient, _In_ PCWSTR
/*pwszArgs*/)
{
    HRESULT hr = S_OK;

    ComPtr<IDebugControl4> spControl;
    IfFailedReturn(pClient->QueryInterface(IID_PPV_ARGS(&spControl)));

    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL,
L"DML/NORMAL Y{t}: %Y{t}\n", L"Hello <World>");
    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL,
L"DML/NORMAL Y{T}: %Y{T}\n", L"Hello <World>");
    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL,
L"DML/NORMAL Y{s}: %Y{s}\n", L"Hello <World>");
    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL,
L"DML/NORMAL Y{S}: %Y{S}\n", L"Hello <World>");

    spControl->ControlledOutputWide(0, DEBUG_OUTPUT_NORMAL, L"TEXT/NORMAL
Y{t}: %Y{t}\n", L"Hello <World>");
    spControl->ControlledOutputWide(0, DEBUG_OUTPUT_NORMAL, L"TEXT/NORMAL
Y{T}: %Y{T}\n", L"Hello <World>");
    spControl->ControlledOutputWide(0, DEBUG_OUTPUT_NORMAL, L"TEXT/NORMAL
Y{s}: %Y{s}\n", L"Hello <World>");
    spControl->ControlledOutputWide(0, DEBUG_OUTPUT_NORMAL, L"TEXT/NORMAL
Y{S}: %Y{S}\n", L"Hello <World>");

    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL,
L"DML/NORMAL Y{a}: %Y{a}\n", (ULONG64)0x00007ffa7da163c0);
    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL,
L"DML/NORMAL Y{as} 64bit   : '%Y{as}'\n", (ULONG64)0x00007ffa7da163c0);
    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL,
L"DML/NORMAL Y{as} 32value : '%Y{as}'\n", (ULONG64)0x1);

    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL,
L"DML/NORMAL Y{ps} 64bit   : '%Y{ps}'\n", (ULONG64)0x00007ffa7da163c0);
    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL,
L"DML/NORMAL Y{ps} 32value : '%Y{ps}'\n", (ULONG64)0x1);

    spControl->ControlledOutputWide(DEBUG_OUTCTL_DML, DEBUG_OUTPUT_NORMAL,
L"DML/NORMAL Y{l}: %Y{l}\n", (ULONG64)0x00007ffa7da163c0);
```

```
    return hr;

}
```

此示例代码将生成以下输出。

```
dbgcmd

0:004> !testout
DML/NORMAL Y{t}: "Hello <World>"
DML/NORMAL Y{T}: "Hello <World>"
DML/NORMAL Y{s}: Hello <World>
DML/NORMAL Y{S}: Hello <World>
TEXT/NORMAL Y{t}: "Hello <World>"
TEXT/NORMAL Y{T}: "Hello &lt;World&gt;"
TEXT/NORMAL Y{s}: Hello <World>
TEXT/NORMAL Y{S}: Hello &lt;World&gt;
DML/NORMAL Y{a}: 00007ffa`7da163c0
DML/NORMAL Y{as} 64bit   : '          '
DML/NORMAL Y{as} 32value : '          '
DML/NORMAL Y{ps} 64bit   : '         '
DML/NORMAL Y{ps} 32value : '         '
DML/NORMAL Y{l}: [d:\th\minkernel\kernelbase\debug.c @ 443]
```

其他控件标志DEBUG_OUTCTL_AMBIENT_DML，允许在不修改任何输出控件属性的情况下规范 DML 上下文文本。 DEBUG_OUTCTL_AMBIENT_TEXT还添加为以前存在的DEBUG_OUTCTL_AMBIENT更具描述性的别名。 输出控制标志在 dbgeng.h 中定义。

```
C++

#define DEBUG_OUTCTL_DML               0x00000020

// Special values which mean leave the output settings
// unchanged.
#define DEBUG_OUTCTL_AMBIENT_DML       0xfffffffe
#define DEBUG_OUTCTL_AMBIENT_TEXT      0xffffffff

// Old ambient flag which maps to text.
#define DEBUG_OUTCTL_AMBIENT           DEBUG_OUTCTL_AMBIENT_TEXT
```

从调试对象提供 DML 内容

dbgeng 已得到增强，可扫描调试对象输出中是否有特殊标记 – 指示应将调试对象输出片段中的剩余文本视为 DML。 模式更改仅适用于单个调试对象输出（例如单个 OutputDebugString 字符串），不是全局模式开关。

此示例演示纯输出和 DML 输出的混合。

text

```
OutputDebugString("This is plain text\n<?dml?>This is <col fg=\"emphfg\"
>DML</col> text\n");
```

生成的输出将具有一行纯文本，后跟一行 DML，其中首字母缩略词 DML 以不同的颜色显示。

IDebugOutputCallbacks2

IDebugOutputCallbacks2 允许 dbgeng 接口客户端接收用于演示的完整 DML 内容。 IDebugOutputCallbacks2 是 IDebugOutputCallbacks (而不是 IDebugOutputCallbacksWide) 的扩展，以便它可以传递到现有的 SetOutputCallbacks 方法。 引擎将为 IDebugOutputCallbacks2 执行 QueryInterface，以查看传入输出回调对象支持哪个接口。 如果对象支持 IDebugOutputCallbacks2，则所有输出将通过扩展的 IDebugOutputCallbacks2 方法发送;将不使用基本 IDebugOutputCallbacks：： Output 方法。

新方法包括：

- IDebugOutputCallbacks2：： GetInterestMask – 允许回调对象描述它想要接收哪些类型的输出通知。 基本选择是纯文本内容 (DEBUG_OUTCBI_TEXT) 和 DML 内容 (DEBUG_OUTCBI_DML) 。 此外，回调对象还可以请求 (DEBUG_OUTCBI_EXPLICIT_FLUSH) 显式刷新的通知。

- IDebugOutputCallbacks2：： Output2 – 所有 IDebugOutputCallbacks2 通知都通过 Output2 发出。 哪个参数指示在 Flags、Arg 和 Text 参数携带通知有效负载时传入的通知类型。 通知包括：

  - DEBUG_OUTCB_TEXT - 纯文本输出。 标志来自 DEBUG_OUTCBF_*，Arg 是输出掩码，Text 是纯文本。 仅当在利益掩码中提供了DEBUG_OUTCBI_TEXT时，才会收到此消息。

  - DEBUG_OUTCB_DML - DML 内容输出。 标志来自 DEBUG_OUTCBF_*，Arg 是输出掩码，Text 是 DML 内容。 仅当在利益掩码中提供了DEBUG_OUTCBI_DML时，才会收到此消息。

  - DEBUG_OUTCB_EXPLICIT_FLUSH – 调用方调用 FlushCallbacks 时没有缓冲文本。 通常，刷新缓冲文本时，将设置DEBUG_OUTCBF_COMBINED_EXPLICIT_FLUSH标志，将两个通知折叠为一个。 如果没有缓冲文本，则发送仅刷新通知。

兴趣掩码标志在 dbgeng.h 中定义，如下所示。

C++

```
// IDebugOutputCallbacks2 interest mask flags.
//
// Indicates that the callback wants notifications
// of all explicit flushes.
#define DEBUG_OUTCBI_EXPLICIT_FLUSH 0x00000001
// Indicates that the callback wants
// content in text form.
#define DEBUG_OUTCBI_TEXT           0x00000002
// Indicates that the callback wants
// content in markup form.
#define DEBUG_OUTCBI_DML            0x00000004

#define DEBUG_OUTCBI_ANY_FORMAT     0x00000006
```

请注意，如果输出对象可以同时处理文本和 DML 内容，则可以注册它们。 在回调的输出处理期间，引擎将选取减少转换的格式，因此支持这两种格式可能会减少引擎中的转换。 不过，这不是必要的，并且仅支持一种格式是预期的操作模式。

自动转换

dbgeng 将在必要时自动在纯文本和 DML 之间转换。 例如，如果调用方将 DML 内容发送到引擎，则引擎将针对仅接受纯文本的所有输出客户端将其转换为纯文本。 或者，对于仅接受 DML 的所有输出回调，引擎会将纯文本转换为 DML。

另请参阅

使用调试器标记语言

JavaScript 调试器脚本

项目 • 2024/04/18

本主题介绍如何使用 JavaScript 创建理解调试器对象的脚本，以及如何扩展和自定义调试器的功能。

JavaScript 调试器脚本概述

脚本提供程序将脚本语言桥接至调试器的内部对象模型。 JavaScript 调试器脚本提供程序，允许将 JavaScript 与调试器一起使用。

通过 .scriptload 命令加载 JavaScript 时，将执行脚本的根代码，脚本中存在的名称会桥接到调试器（dx 调试器）的根命名空间中，并且脚本将一直驻留在内存中，直到卸载该脚本并释放对其对象的所有引用。 该脚本可以为调试器的表达式计算器提供新函数、修改调试器的对象模型，也可以像 NatVis 可视化工具那样充当可视化工具。

本主题介绍使用 JavaScript 调试器脚本可以执行的一些操作。

这两个主题提供有关在调试器中使用 JavaScript 的其他信息。

JavaScript 调试器示例脚本

JavaScript 扩展中的本机对象

JavaScript 脚本视频

碎片整理工具 #170 - Andy 和 Bill 演示调试器中的 JavaScript 扩展性和脚本功能。

调试器 JavaScript 提供程序

调试器随附的 JavaScript 提供程序充分利用了最新的 ECMAScript6 对象和类增强功能。有关详细信息，请参阅 ECMAScript 6 — 新功能：概述和比较 。

JsProvider.dll

JsProvider.dll 是为支持 JavaScript 调试器脚本而加载的 JavaScript 提供程序。

要求

JavaScript 调试器脚本设计用于所有受支持的 Windows 版本。

加载 JavaScript 脚本提供程序

在使用任何 .script 命令之前，需要加载脚本提供程序。 使用 .scriptproviders 命令确认 JavaScript 提供程序已加载。

dbgcmd

```
0:000> .scriptproviders
Available Script Providers:
    NatVis (extension '.NatVis')
    JavaScript (extension '.js')
```

JavaScript 脚本元命令

以下命令可用于使用 JavaScript 调试器脚本。

- **.scriptproviders（列出脚本提供程序）**
- **.scriptload（加载脚本）**
- **.scriptload（卸载脚本）**
- **.scriptrun（运行脚本）**
- **.scriptlist（列出已加载的脚本）**

要求

在使用任何 .script 命令之前，需要加载脚本提供程序。 使用 .scriptproviders 命令确认 JavaScript 提供程序已加载。

dbgcmd

```
0:000> .scriptproviders
Available Script Providers:
    NatVis (extension '.NatVis')
    JavaScript (extension '.js')
```

.scriptproviders（列出脚本提供程序）

.scriptproviders 命令将列出调试器当前理解的所有脚本语言以及注册它们的扩展。

在下面的示例中，将加载 JavaScript 和 NatVis 提供程序。

dbgcmd

```
0:000> .scriptproviders
Available Script Providers:
```

```
    NatVis (extension '.NatVis')
    JavaScript (extension '.js')
```

以".NatVis"结尾的任何文件均被理解为 NatVis 脚本，以".js"结尾的任何文件均被理解为 JavaScript 脚本。 可以使用 .scriptload 命令加载任一类型的脚本。

有关详细信息，请参阅 **.scriptproviders（列出脚本提供程序）**

.scriptload（加载脚本）

.scriptload 命令将加载脚本，并执行脚本的根代码和 *initializeScript* 函数。 如果初始加载和执行脚本的过程中出现任何错误，则错误将显示在控制台上。 以下命令显示成功加载 TestScript.js。

dbgcmd

```
0:000> .scriptload C:\WinDbg\Scripts\TestScript.js
JavaScript script successfully loaded from 'C:\WinDbg\Scripts\TestScript.js'
```

脚本所做的任何对象模型操作都将保持原位，直到脚本随后被卸载或再次使用不同的内容运行。

有关详细信息，请参阅 **.scriptload（加载脚本）**

.scriptrun

.scriptrun 命令将加载脚本、执行脚本的根代码、*initializeScript* 和 *invokeScript* 函数。 如果初始加载和执行脚本的过程中出现任何错误，则错误将显示在控制台上。

dbgcmd

```
0:000> .scriptrun C:\WinDbg\Scripts\helloWorld.js
JavaScript script successfully loaded from 'C:\WinDbg\Scripts\helloWorld.js'
Hello World!  We are in JavaScript!
```

脚本所做的任何调试器对象模型操作都将保持原位，直到脚本随后被卸载或再次使用不同的内容运行。

有关详细信息，请参阅 **.scriptrun（运行脚本）**。

.scriptload（卸载脚本）

.scriptunload 命令卸载加载的脚本，并调用 *uninitializeScript* 函数。 使用以下命令语法卸载脚本

```
dbgcmd

0:000:x86> .scriptunload C:\WinDbg\Scripts\TestScript.js
JavaScript script unloaded from 'C:\WinDbg\Scripts\TestScript.js'
```

有关详细信息，请参阅 **.scriptunload（加载脚本）**。

.scriptlist（列出已加载的脚本）

.scriptlist 命令将列出通过 .scriptload 或 .scriptrun 命令加载的任何脚本。 如果使用 .scriptload 成功加载 TestScript，则 .scriptlist 命令将显示加载的脚本的名称。

```
dbgcmd

0:000> .scriptlist
Command Loaded Scripts:
    JavaScript script from 'C:\WinDbg\Scripts\TestScript.js'
```

有关详细信息，请参阅 **.scriptlist（列出加载的脚本）**。

JavaScript 调试器脚本入门

HelloWorld 示例脚本

本节介绍如何创建和执行一个简单的 JavaScript 调试器脚本，该脚本将输出 Hello World。

```
dbgcmd

// WinDbg JavaScript sample
// Prints Hello World
function initializeScript()
{
    host.diagnostics.debugLog("***> Hello World! \n");
}
```

使用文本编辑器（如记事本）创建一个名为 *HelloWorld.js* 的文本文件，其中包含上面所示的 JavaScript 代码。

使用 .scriptload 命令加载并执行脚本。 由于使用了函数名称 *initializeScript*，因此在加载脚本时会运行函数中的代码。

dbgcmd

```
0:000> .scriptload c:\WinDbg\Scripts\HelloWorld.js
JavaScript script successfully loaded from 'c:\WinDbg\Scripts\HelloWorld.js'
***> Hello World!
```

加载脚本后，调试器中会提供其他功能。 使用 **dx（显示 NatVis 表达式）**命令显示 *Debugger.State.Scripts*，以查看脚本现在是否已驻留。

dbgcmd

```
0:000> dx Debugger.State.Scripts
Debugger.State.Scripts
    HelloWorld
```

在下一个示例中，我们将添加并调用一个命名函数。

添加两个值示例脚本

本节介绍如何创建和执行一个简单的 JavaScript 调试器脚本，该脚本添加输入并添加两个数字。

此简单脚本提供单个函数 addTwoValues。

dbgcmd

```
// WinDbg JavaScript sample
// Adds two functions
function addTwoValues(a, b)
 {
     return a + b;
 }
```

使用文本编辑器（如记事本）创建名为 *FirstSampleFunction.js* 的文本文件

使用 .scriptload 命令加载脚本。

dbgcmd

```
0:000> .scriptload c:\WinDbg\Scripts\FirstSampleFunction.js
JavaScript script successfully loaded from
'c:\WinDbg\Scripts\FirstSampleFunction.js'
```

加载脚本后，调试器中会提供其他功能。 使用 **dx（显示 NatVis 表达式）**命令显示 *Debugger.State.Scripts*，以查看脚本现在是否已驻留。

```
dbgcmd
```

```
0:000> dx Debugger.State.Scripts
Debugger.State.Scripts
    FirstSampleFunction
```

可以选择 *FirstSampleFunction*，查看它提供的函数。

```
dbgcmd
```

```
0:000> dx -r1 -v Debugger.State.Scripts.FirstSampleFunction.Contents
Debugger.State.Scripts.FirstSampleFunction.Contents                 :
[object Object]
    host             : [object Object]
    addTwoValues
 ...
```

为了使脚本更便于使用，请使用 dx 命令在调试器中指定一个变量来保存脚本的内容。

```
dbgcmd
```

```
0:000> dx @$myScript = Debugger.State.Scripts.FirstSampleFunction.Contents
```

使用 dx 表达式计算器调用 addTwoValues 函数。

```
dbgcmd
```

```
0:000> dx @$myScript.addTwoValues(10, 41),d
@$myScript.addTwoValues(10, 41),d : 51
```

还可以使用 *$@scriptContents* 内置别名来处理脚本。 *$@scriptContents* 别名合并加载的所有脚本的所有内容。

```
dbgcmd
```

```
0:001> dx @$scriptContents.addTwoValues(10, 40),d
@$scriptContents.addTwoValues(10, 40),d : 50
```

当完成对脚本的处理后，使用 .scriptunload 命令卸载脚本。

```
dbgcmd
```

```
0:000> .scriptunload c:\WinDbg\Scripts\FirstSampleFunction.js
JavaScript script successfully unloaded from
'c:\WinDbg\Scripts\FirstSampleFunction.js'
```

调试器命令自动化

本节介绍如何创建和执行一个简单的 JavaScript 调试器脚本，该脚本可自动发送 **u（取消汇编）**命令。 此示例还显示如何在循环中收集和显示命令输出。

此脚本提供单个函数 RunCommands()。

JavaScript

```
// WinDbg JavaScript sample
// Shows how to call a debugger command and display results
"use strict";

function RunCommands()
{
var ctl = host.namespace.Debugger.Utility.Control;
var output = ctl.ExecuteCommand("u");
host.diagnostics.debugLog("***> Displaying command output \n");

for (var line of output)
   {
   host.diagnostics.debugLog("  ", line, "\n");
   }

host.diagnostics.debugLog("***> Exiting RunCommands Function \n");

}
```

使用文本编辑器（如记事本）创建名为 *RunCommands.js* 的文本文件

使用 .scriptload 命令加载 RunCommands 脚本。

dbgcmd

```
0:000> .scriptload c:\WinDbg\Scripts\RunCommands.js
JavaScript script successfully loaded from
'c:\WinDbg\Scripts\RunCommands.js'
```

加载脚本后，调试器中会提供其他功能。 使用 **dx（显示 NatVis 表达式）**命令显示 *Debugger.State.Scripts.RunCommands*，以查看脚本现在是否已驻留。

dbgcmd

```
0:000>dx -r3 Debugger.State.Scripts.RunCommands
Debugger.State.Scripts.RunCommands
    Contents         : [object Object]
        host             : [object Object]
            diagnostics      : [object Object]
            namespace
            currentSession   : Live user mode: <Local>
            currentProcess   : notepad.exe
            currentThread    : ntdll!DbgUiRemoteBreakin (00007ffd`87f2f440)
            memory           : [object Object]
```

使用 dx 命令在 RunCommands 脚本中调用 RunCommands 函数。

dbgcmd

```
0:000> dx Debugger.State.Scripts.RunCommands.Contents.RunCommands()
  ***> Displaying command output
  ntdll!ExpInterlockedPopEntrySListEnd+0x17
[d:\rs1\minkernel\ntos\rtl\amd64\slist.asm @ 196]:
  00007ffd`87f06e67 cc              int     3
  00007ffd`87f06e68 cc              int     3
  00007ffd`87f06e69 0f1f8000000000  nop     dword ptr [rax]
  ntdll!RtlpInterlockedPushEntrySList
[d:\rs1\minkernel\ntos\rtl\amd64\slist.asm @ 229]:
  00007ffd`87f06e70 0f0d09          prefetchw [rcx]
  00007ffd`87f06e73 53              push    rbx
  00007ffd`87f06e74 4c8bd1          mov     r10,rcx
  00007ffd`87f06e77 488bca          mov     rcx,rdx
  00007ffd`87f06e7a 4c8bda          mov     r11,rdx
***> Exiting RunCommands Function
```

特殊 JavaScript 调试器函数

脚本提供程序本身调用的 JavaScript 脚本中有几个特殊函数。

initializeScript

当加载并执行 JavaScript 脚本时，在脚本中的变量、函数和其他对象影响调试器的对象模型之前，它要经历一系列步骤。

- 将脚本加载到内存中并进行解析。
- 执行脚本中的根代码。
- 如果脚本有一个名为 initializeScript 的方法，则调用该方法。
- initializeScript 中的返回值用于确定如何自动修改调试器的对象模型。
- 脚本中的名称将桥接到调试器的命名空间。

如前所述，initializeScript 将在脚本的根代码执行后立即被调用。 它的工作是向提供程序返回注册对象的 JavaScript 数组，指示如何修改调试器的对象模型。

JavaScript

```
function initializeScript()
{
    // Add code here that you want to run every time the script is loaded.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("***> initializeScript was called\n");
}
```

invokeScript

invokeScript 方法是主脚本方法，在运行 .scriptload 和 .scriptrun 时调用。

JavaScript

```
function invokeScript()
{
    // Add code here that you want to run every time the script is executed.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("***> invokeScript was called\n");
}
```

uninitializeScript

uninitializeScript 方法与 initializeScript 的行为相反。 当脚本取消链接并准备好卸载时，将调用它。 它的工作是撤销脚本在执行过程中对对象模型所做的任何更改和/或销毁脚本缓存的任何对象。

如果脚本既不对对象模型执行命令性操作，也不缓存结果，则无需使用 uninitializeScript 方法。 根据 initializeScript 的返回值执行的对对象模型的任何更改都将由提供程序自动撤消。 这样的更改不需要显式 uninitializeScript 方法。

JavaScript

```
function uninitializeScript()
{
    // Add code here that you want to run every time the script is unloaded.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("***> uninitialize was called\n");
}
```

脚本命令调用的函数摘要

此表总结了脚本命令调用的函数

展开表

命令	.scriptload	.scriptrun（运行脚本）	.scriptload（卸载脚本）
root	是	是	
initializeScript	是	是	
invokeScript		是	
uninitializeScript			是

使用此示例代码查看在加载、执行和卸载脚本时何时调用每个函数。

JavaScript

```
// Root of Script
host.diagnostics.debugLog("***>; Code at the very top (root) of the script
is always run \n");


function initializeScript()
{
    // Add code here that you want to run every time the script is loaded.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("***>; initializeScript was called \n");
}

function invokeScript()
{
    // Add code here that you want to run every time the script is executed.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("***>; invokeScript was called \n");
}


function uninitializeScript()
{
    // Add code here that you want to run every time the script is unloaded.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("***>; uninitialize was called\n");
}


function main()
{
    // main is just another function name in JavaScript
    // main is not called by .scriptload or .scriptrun
```

```
    host.diagnostics.debugLog("***>; main was called \n");
}
```

在 JavaScript 中创建调试器可视化工具

通过自定义可视化文件，可以在可视化结构中对数据进行分组和组织，以更好地反映数据关系和内容。 可以使用 JavaScript 调试器扩展编写调试器可视化工具，这些可视化工具的行为方式与 NatVis 非常相似。 这是通过创作 JavaScript 原型对象（或 ES6 类）来实现的，该对象充当给定数据类型的可视化工具。 有关 NatVis 和调试器的详细信息，请参阅 **dx（显示 NatVis 表达式）**。

示例类 - Simple1DArray

请考虑表示一维数组的 C++ 类的示例。 此类有两个成员，m_size 是数组的总大小，m_pValues 是指向内存中等于 m_size 字段的一些 ints 的指针。

C++

```
class Simple1DArray
{
private:

    ULONG64 m_size;
    int *m_pValues;
};
```

可以使用 dx 命令查看默认数据结构呈现。

dbgcmd

```
0:000> dx g_array1D
g_array1D                 [Type: Simple1DArray]
    [+0x000] m_size           : 0x5 [Type: unsigned __int64]
    [+0x008] m_pValues        : 0x8be32449e0 : 0 [Type: int *]
```

JavaScript 可视化工具

为了可视化这种类型，需要创作一个原型（或 ES6）类，其中包含我们希望调试器显示的所有字段和属性。 我们还需要让 initializeScript 方法返回一个对象，该对象指示 JavaScript 提供程序将原型作为给定类型的可视化工具链接。

JavaScript

```
function initializeScript()
{
```

```
    //
    // Define a visualizer class for the object.
    //
    class myVisualizer
    {
        //
        // Create an ES6 generator function which yields back all the values
in the array.
        //
        *[Symbol.iterator]()
        {
            var size = this.m_size;
            var ptr = this.m_pValues;
            for (var i = 0; i < size; ++i)
            {
                yield ptr.dereference();

                //
                // Note that the .add(1) method here is effectively doing
pointer arithmetic on
                // the underlying pointer.  It is moving forward by the size
of 1 object.
                //
                ptr = ptr.add(1);
            }
        }
    }

    return [new host.typeSignatureRegistration(myVisualizer,
"Simple1DArray")];
}
```

将脚本保存在名为 arrayVisualizer.js 的文件中。

使用 **.load（加载扩展 DLL）** 命令加载 JavaScript 提供程序。

dbgcmd

```
0:000> .load C:\ScriptProviders\jsprovider.dll
```

使用 .scriptload 加载数组可视化工具脚本。

dbgcmd

```
0:000> .scriptload c:\WinDbg\Scripts\arrayVisualizer.js
JavaScript script successfully loaded from
'c:\WinDbg\Scripts\arrayVisualizer.js'
```

现在，当使用 dx 命令时，脚本可视化工具将显示多行数组内容。

dbgcmd

```
0:000> dx g_array1D
g_array1D                 : [object Object] [Type: Simple1DArray]
    [<Raw View>]     [Type: Simple1DArray]
    [0x0]            : 0x0
    [0x1]            : 0x1
    [0x2]            : 0x2
    [0x3]            : 0x3
    [0x4]            : 0x4
```

此外，此 JavaScript 可视化效果还提供 LINQ 功能，例如 Select。

dbgcmd

```
0:000> dx g_array1D.Select(n => n * 3),d
g_array1D.Select(n => n * 3),d
    [0]              : 0
    [1]              : 3
    [2]              : 6
    [3]              : 9
    [4]              : 12
```

影响可视化效果的因素

通过从 initializeScript 返回 host.typeSignatureRegistration 对象而成为本机类型可视化工具的原型或类，将把 JavaScript 中的所有属性和方法添加到本机类型中。 此外，还适用以下语义：

- 任何不以两个下划线 (__) 开头的名称都将在可视化效果中可用。
- 属于标准 JavaScript 对象或 JavaScript 提供程序创建的协议的名称不会显示在可视化效果中。
- 可以通过 [Symbol.iterator] 的支持使对象可迭代。
- 通过支持由几个函数组成的自定义协议，可以使对象具有可索引性：getDimensionality、getValueAt 和可选的 setValueAt。

本机和 JavaScript 对象桥

JavaScript 与调试器的对象模型之间的桥是双向的。 本机对象可以传递到 JavaScript 中，JavaScript 对象可以传递到调试器的表达式计算器中。 例如，考虑在脚本中添加以下方法：

JavaScript

```
function multiplyBySeven(val)
{
    return val * 7;
}
```

现在可以在上面的示例 LINQ 查询中使用此方法。 首先，加载 JavaScript 可视化效果。

dbgcmd

```
0:000> .scriptload c:\WinDbg\Scripts\arrayVisualizer2.js
JavaScript script successfully loaded from
'c:\WinDbg\Scripts\arrayVisualizer2.js'

0:000> dx @$myScript = Debugger.State.Scripts.arrayVisualizer2.Contents
```

然后，可以使用 multiplyBySeven 函数内联函数，如下所示。

dbgcmd

```
0:000> dx g_array1D.Select(@$myScript.multiplyBySeven),d
g_array1D.Select(@$myScript.multiplyBySeven),d
    [0]              : 0
    [1]              : 7
    [2]              : 14
    [3]              : 21
    [4]              : 28
```

带 JavaScript 的条件断点

可以在遇到断点后使用 JavaScript 进行补充处理。 例如，脚本可用于检查其他运行时值，然后确定是要自动继续执行代码，还是要停止并执行其他手动调试。

有关使用断点的一般信息，请参阅控制断点的方法。

DebugHandler.js 示例断点处理脚本

本示例将评估记事本的打开和保存对话框：*notepad!ShowOpenSaveDialog*。 此脚本将评估 pszCaption 变量，以确定当前对话是“打开”对话框还是“另存为”对话框。 如果是“打开”对话框，代码执行将继续。 如果是“另存为”对话框，代码执行将停止，调试器将中断。

JavaScript

```
 // Use JavaScript strict mode
"use strict";

// Define the invokeScript method to handle breakpoints
```

```
function invokeScript()
{
   var ctl = host.namespace.Debugger.Utility.Control;

   //Get the address of my string
   var address = host.evaluateExpression("pszCaption");

   // The open and save dialogs use the same function
   // When we hit the open dialog, continue.
   // When we hit the save dialog, break.
   if (host.memory.readWideString(address) == "Open") {
       // host.diagnostics.debugLog("We're opening, let's continue!\n");
       ctl.ExecuteCommand("gc");
   }
   else
   {
       //host.diagnostics.debugLog("We're saving, let's break!\n");
   }
 }
```

此命令在 notepad!ShowOpenSaveDialog 上设置断点，并将在遇到断点时运行上述脚本。

dbgcmd

```
bp notepad!ShowOpenSaveDialog ".scriptrun
C:\\WinDbg\\Scripts\\DebugHandler.js"
```

然后，当在记事本中选择“文件”>“保存”选项时，脚本将运行，g 命令不会发送，代码执行中断。

dbgcmd

```
JavaScript script successfully loaded from
'C:\WinDbg\Scripts\DebugHandler.js'
notepad!ShowOpenSaveDialog:
00007ff6`f9761884 48895c2408      mov     qword ptr [rsp+8],rbx
ss:000000db`d2a9f2f0=0000021985fe2060
```

在 JavaScript 扩展中使用 64 位值

本节介绍传递到 JavaScript 调试器扩展的 64 位值的行为方式。 这个问题的出现是因为 JavaScript 只能使用 53 位存储数字。

64 位和 JavaScript 53 位存储

传递给 JavaScript 的序号值通常作为 JavaScript 数字进行封送。 问题在于 JavaScript 数字是 64 位双精度浮点值。 任何超过 53 位的序号都会在进入 JavaScript 时失去精度。 这对于 64 位指针和其他 64 位序号值（可能具有最高字节的标志）提出了问题。 为了解决此问题，任何输入 JavaScript 的 64 位本机值（无论是本机代码还是数据模型）都以库类型的形式输入，而不是以 JavaScript 数字的形式输入。 此库类型将往返到本机代码，而不会损失数值精度。

自动转换

64 位序号值的库类型支持标准 JavaScript valueOf 转换。 如果该对象用于需要转换值的数学操作或其他构造，它将自动转换为 JavaScript 数字。 如果发生精度损失（该值使用了超过 53 位的序号精度），则 JavaScript 提供程序将引发异常。

注意：如果在 JavaScript 中使用按位运算符，则序号精度进一步限制为 32 位。

此示例代码对两个数字求和，用于测试 64 位值的转换。

JavaScript

```
function playWith64BitValues(a64, b64)
{
    // Sum two numbers to demonstrate 64-bit behavior.
    //
    // Imagine a64==100, b64==1000
    // The below would result in sum==1100 as a JavaScript number.  No
exception is thrown.  The values auto-convert.
    //
    // Imagine a64==2^56, b64=1
    // The below will **Throw an Exception**.  Conversion to numeric results
in loss of precision!
    //
    var sum = a64 + b64;
    host.diagnostics.debugLog("Sum   >> ", sum, "\n");

}

function performOp64BitValues(a64, b64, op)
{
    //
    // Call a data model method passing 64-bit value.  There is no loss of
precision here.  This round trips perfectly.
    // For example:
    //  0:000> dx @$myScript.playWith64BitValues(0x4444444444444444ull,
0x3333333333333333ull, (x, y) => x + y)
    //  @$myScript.playWith64BitValues(0x4444444444444444ull,
0x3333333333333333ull, (x, y) => x + y) : 0x7777777777777777
    //
    return op(a64, b64);
}
```

使用文本编辑器（如记事本）创建名为 *PlayWith64BitValues.js* 的文本文件

使用 .scriptload 命令加载脚本。

dbgcmd

```
0:000> .scriptload c:\WinDbg\Scripts\PlayWith64BitValues.js
JavaScript script successfully loaded from
'c:\WinDbg\Scripts\PlayWith64BitValues.js'
```

为了使脚本更便于使用，请使用 dx 命令在调试器中指定一个变量来保存脚本的内容。

dbgcmd

```
0:000> dx @$myScript = Debugger.State.Scripts.PlayWith64BitValues.Contents
```

使用 dx 表达式计算器调用 addTwoValues 函数。

首先，我们将计算 2^53 =9007199254740992（十六进制 0x20000000000000）的值。

首先进行测试，我们将使用 (2^53) - 2，看看它是否返回正确的总和值。

dbgcmd

```
0:000> dx @$myScript.playWith64BitValues(9007199254740990, 9007199254740990)
Sum    >> 18014398509481980
```

然后，我们将计算 (2^53) -1 =9007199254740991。 这将返回一个错误，指示转换过程将损失精度，因此这是 JavaScript 代码中求和方法可以使用的最大值。

dbgcmd

```
0:000> dx @$myScript.playWith64BitValues(9007199254740990, 9007199254740991)
Error: 64 bit value loses precision on conversion to number
```

调用传递 64 位值的数据模型方法。 此时不会损失精度。

dbgcmd

```
0:001> dx @$myScript.performOp64BitValues( 0x7FFFFFFFFFFFFFF,
0x7FFFFFFFFFFFFFF, (x, y) => x + y)
@$myScript.performOp64BitValues( 0x7FFFFFFFFFFFFFF,  0x7FFFFFFFFFFFFFF,
(x, y) => x + y) : 0xfffffffffffffffe
```

比较

64 位库类型是 JavaScript 对象，而不是 JavaScript 数字之类的值类型。 这对比较操作有一些影响。 通常，对象的相等性 (==) 表示操作数引用的是同一个对象，而不是同一个值。 JavaScript 提供程序通过跟踪对 64 位值的实时引用并为未收集的 64 位值返回相同的“不可变”对象来缓解此问题。 这意味着为了进行比较，将出现以下情况。

JavaScript

```
// Comparison with 64 Bit Values

function comparisonWith64BitValues(a64, b64)
{
    //
    // No auto-conversion occurs here.  This is an *EFFECTIVE* value
comparison.  This works with ordinals with above 53-bits of precision.
    //
    var areEqual = (a64 == b64);
    host.diagnostics.debugLog("areEqual   >> ", areEqual, "\n");
    var areNotEqual = (a64 != b64);
    host.diagnostics.debugLog("areNotEqual   >> ", areNotEqual, "\n");

    //
    // Auto-conversion occurs here.  This will throw if a64 does not pack
into a JavaScript number with no loss of precision.
    //
    var isEqualTo42 = (a64 == 42);
    host.diagnostics.debugLog("isEqualTo42   >> ", isEqualTo42, "\n");
    var isLess = (a64 < b64);
    host.diagnostics.debugLog("isLess   >> ", isLess, "\n");
```

使用文本编辑器（如记事本）创建名为 *ComparisonWith64BitValues.js* 的文本文件

使用 .scriptload 命令加载脚本。

dbgcmd

```
0:000> .scriptload c:\WinDbg\Scripts\ComparisonWith64BitValues.js
JavaScript script successfully loaded from
'c:\WinDbg\Scripts\ComparisonWith64BitValues.js'
```

为了使脚本更便于使用，请使用 dx 命令在调试器中指定一个变量来保存脚本的内容。

dbgcmd

```
0:000> dx @$myScript =
Debugger.State.Scripts.comparisonWith64BitValues.Contents
```

首先进行测试，我们将使用 (2^53) - 2，看看它是否返回预期值。

dbgcmd

```
0:001> dx @$myScript.comparisonWith64BitValues(9007199254740990,
9007199254740990)
areEqual    >> true
areNotEqual    >> false
isEqualTo42    >> false
isLess    >> false
```

我们还将尝试将数字 42 作为第一个值，以验证比较操作符是否正常工作。

dbgcmd

```
0:001> dx @$myScript.comparisonWith64BitValues(42, 9007199254740990)
areEqual    >> false
areNotEqual    >> true
isEqualTo42    >> true
isLess    >> true
```

然后，我们将计算 (2^53) -1 =9007199254740991。 此值将返回一个错误，指示转换过程将损失精度，因此这是 JavaScript 代码中比较运算符可以使用的最大值。

dbgcmd

```
0:000> dx @$myScript.playWith64BitValues(9007199254740990, 9007199254740991)
Error: 64 bit value loses precision on conversion to number
```

在操作中保持精度

为了允许调试器扩展保持精度，将一组数学函数投射到 64 位库类型的顶部。 如果扩展需要（或可能需要）传入 64 位值的精度高于 53 位，则应使用以下方法，而不是依赖于标准操作符：

展开表

“方法名称”	Signature	描述
asNumber	.asNumber()	将 64 位值转换为 JavaScript 数字。 如果发生精度损失，**将引发异常**
convertToNumber	.convertToNumber()	将 64 位值转换为 JavaScript 数字。 如果发生精度损失，**不会引发异常**
getLowPart	.getLowPart()	将 64 位值的后 32 位转换为 JavaScript 数字
getHighPart	.getHighPart()	将 64 位值的前 32 位转换为 JavaScript 数字

“方法名称”	Signature	描述
add	.add(value)	将值添加到 64 位值并返回结果
subtract	.subtract(value)	从 64 位值中减去一个值并返回结果
multiply	.multiply(value)	将 64 位值乘以提供的值并返回结果
divide	.divide(value)	将 64 位值除以提供的值并返回结果
bitwiseAnd	.bitwiseAnd(value)	使用提供的值计算 64 位值的按位值并返回结果
bitwiseOr	.bitwiseOr(value)	使用提供的值计算 64 位值的按位“或”值并返回结果
bitwiseXor	.bitwiseXor(value)	使用提供的值计算 64 位值的按位“异或”值并返回结果
bitwiseShiftLeft	.bitwiseShiftLeft(value)	将 64 位值向左移动给定的量并返回结果
bitwiseShiftRight	.bitwiseShiftRight(value)	将 64 位值向右移动给定的量并返回结果
toString	.toString([radix])	将 64 位值转换为默认基数（或可选提供的基数）的显示字符串

这种方法也是可用的。

展开表

“方法名称”	Signature	描述
compareTo	.compareTo(value)	将 64 位值与另一个 64 位值进行比较。

JavaScript 调试

本节介绍如何使用调试器的脚本调试功能。 调试器集成了对使用 .scriptdebug（调试 JavaScript）命令调试 JavaScript 脚本的支持。

> **备注**
>
> 若要将 JavaScript 调试与 WinDb 配合使用，请以管理员身份运行调试器。

使用此示例代码探索调试 JavaScript。 在本演练中，我们将其命名为 DebuggableSample.js，并将其保存在 C:\MyScripts 目录中。

```
JavaScript
```

```
"use strict";

class myObj
{
    toString()
    {
        var x = undefined[42];
        host.diagnostics.debugLog("BOO!\n");
    }
}

class iterObj
{
    *[Symbol.iterator]()
    {
        throw new Error("Oopsies!");
    }
}

function foo()
{
    return new myObj();
}

function iter()
{
    return new iterObj();
}

function throwAndCatch()
{
    var outer = undefined;
    var someObj = {a : 99, b : {c : 32, d: "Hello World"} };
    var curProc = host.currentProcess;
    var curThread = host.currentThread;

    try
    {
        var x = undefined[42];
    } catch(e)
    {
        outer = e;
    }

    host.diagnostics.debugLog("This is a fun test\n");
    host.diagnostics.debugLog("Of the script debugger\n");
    var foo = {a : 99, b : 72};
    host.diagnostics.debugLog("foo.a = ", foo.a, "\n");

    return outer;
}

function throwUnhandled()
{
```

```
    var proc = host.currentProcess;
    var thread = host.currentThread;
    host.diagnostics.debugLog("Hello...  About to throw an exception!\n");
    throw new Error("Oh me oh my!  This is an unhandled exception!\n");
    host.diagnostics.debugLog("Oh...  this will never be hit!\n");
    return proc;
}

function outer()
{
    host.diagnostics.debugLog("inside outer!\n");
    var foo = throwAndCatch();
    host.diagnostics.debugLog("Caught and returned!\n");
    return foo;
}

function outermost()
{
    var x = 99;
    var result = outer();
    var y = 32;
    host.diagnostics.debugLog("Test\n");
    return result;
}

function initializeScript()
{
    //
    // Return an array of registration objects to modify the object model of
the debugger
    // See the following for more details:
    //
    //     https://aka.ms/JsDbgExt
    //
}
```

加载示例脚本。

dbgcmd

```
.scriptload C:\MyScripts\DebuggableSample.js
```

使用 **.scriptdebug** 命令，开始主动调试脚本。

dbgcmd

```
0:000> .scriptdebug C:\MyScripts\DebuggableSample.js
>>> ****** DEBUGGER ENTRY DebuggableSample ******
           No active debug event!

>>> Debug [DebuggableSample <No Position>] >
```

看到提示符 `>>> Debug [DebuggableSample <No Position>] >` 和输入请求后，将位于脚本调试器内部。

使用 **.help** 命令在 JavaScript 调试环境中显示命令列表。

dbgcmd

```
>>> Debug [DebuggableSample <No Position>] >.help
Script Debugger Commands (*NOTE* IDs are **PER SCRIPT**):
    ? ................................. Get help
    ? <expr>  ......................... Evaluate expression <expr> and
display result
    ?? <expr>  ........................ Evaluate expression <expr> and
display result
    |  ................................ List available scripts
    |<scriptid>s  ..................... Switch context to the given script
    bc \<bpid\>  ....................... Clear breakpoint by specified \
<bpid\>
    bd \<bpid\>  ....................... Disable breakpoint by specified \
<bpid\>
    be \<bpid\>  ....................... Enable breakpoint by specified \
<bpid\>
    bl  ............................... List breakpoints
    bp <line>:<column>  ............... Set breakpoint at the specified
line and column
    bp <function-name>  ............... Set breakpoint at the (global)
function specified by the given name
    bpc  .............................. Set breakpoint at current location
    dv  ............................... Display local variables of current
frame
    g  ................................ Continue script
    gu   .............................. Step out
    k  ................................ Get stack trace
    p  ................................ Step over
    q  ................................ Exit script debugger (resume
execution)
    sx  ............................... Display available events/exceptions
to break on
    sxe <event>  ...................... Enable break on <event>
    sxd <event>  ...................... Disable break on <event>
    t  ................................ Step in
    .attach <scriptId>  ............... Attach debugger to the script
specified by <scriptId>
    .detach [<scriptId>]  ............. Detach debugger from the script
specified by <scriptId>
    .frame <index>  ................... Switch to frame number <index>
    .f+  .............................. Switch to next stack frame
    .f-  .............................. Switch to previous stack frame
    .help  ............................ Get help
```

使用 **sx** 脚本调试器命令查看可捕获的事件列表。

dbgcmd

```
>>> Debug [DebuggableSample <No Position>] >sx
sx
    ab  [   inactive] .... Break on script abort
    eh  [   inactive] .... Break on any thrown exception
    en  [   inactive] .... Break on entry to the script
    uh  [     active] .... Break on unhandled exception
```

使用 **sxe** 脚本调试器命令打开中断条目，以便在脚本调试器中的任何代码执行后立即将脚本捕获到脚本调试器中。

dbgcmd

```
>>> Debug [DebuggableSample <No Position>] >sxe en
sxe en
Event filter 'en' is now active
```

退出脚本调试器，我们将对脚本进行函数调用，该脚本将捕获到调试器中。

dbgcmd

```
>>> Debug [DebuggableSample <No Position>] >q
```

此时，你又回到了正常的调试器中。 执行以下命令以调用脚本。

dbgcmd

```
dx @$scriptContents.outermost()
```

现在，你回到了脚本调试器中，并在最外层 JavaScript 函数的第一行上中断。

dbgcmd

```
>>> ****** SCRIPT BREAK DebuggableSample [BreakIn] ******
           Location: line = 73, column = 5
           Text: var x = 99

>>> Debug [DebuggableSample 73:5] >
```

除了在调试器中看到中断之外，还可以获取发生中断的行 (73) 和列 (5) 的信息，以及相关的源代码片段：*var x = 99*。

让我们执行几次操作，并进入脚本中的另一个位置。

dbgcmd

```
    p
    t
    p
    t
    p
    p
```

此时，应进入第 34 行的 throwAndCatch 方法。

dbgcmd

```
...
>>> ****** SCRIPT BREAK DebuggableSample [Step Complete] ******
           Location: line = 34, column = 5
           Text: var curProc = host.currentProcess
```

可以通过执行堆栈跟踪来验证这一点。

dbgcmd

```
>>> Debug [DebuggableSample 34:5] >k
k
    ##  Function                          Pos    Source Snippet
-> [00] throwAndCatch                     034:05 (var curProc =
host.currentProcess)
   [01] outer                             066:05 (var foo = throwAndCatch())
   [02] outermost                         074:05 (var result = outer())
```

从这里，可以研究变量的值。

dbgcmd

```
>>> Debug [DebuggableSample 34:5] >??someObj
??someObj
someObj          : {...}
    __proto__        : {...}
    a                : 0x63
    b                : {...}
>>> Debug [DebuggableSample 34:5] >??someObj.b
??someObj.b
someObj.b        : {...}
    __proto__        : {...}
    c                : 0x20
    d                : Hello World
```

我们在当前代码行上设置一个断点，看看现在设置了哪些断点。

dbgcmd

```
>>> Debug [DebuggableSample 34:5] >bpc
bpc
Breakpoint 1 set at 34:5
>>> Debug [DebuggableSample 34:5] >bl
bl
      Id State    Pos
       1 enabled  34:5
```

从这里，我们将使用 **sxd** 脚本调试器命令禁用条目 (en) 事件。

dbgcmd

```
>>> Debug [DebuggableSample 34:5] >sxd en
sxd en
Event filter 'en' is now inactive
```

然后继续，让脚本继续到最后。

dbgcmd

```
>>> Debug [DebuggableSample 34:5] >g
g
This is a fun test
Of the script debugger
foo.a = 99
Caught and returned!
Test
...
```

再次执行脚本方法，并观察遇到的断点。

dbgcmd

```
0:000> dx @$scriptContents.outermost()
inside outer!
>>> ****** SCRIPT BREAK DebuggableSample [Breakpoint 1] ******
           Location: line = 34, column = 5
           Text: var curProc = host.currentProcess
```

显示调用堆栈。

dbgcmd

```
>>> Debug [DebuggableSample 34:5] >k
k
    ##  Function                          Pos    Source Snippet
-> [00] throwAndCatch                     034:05 (var curProc =
host.currentProcess)
```

```
    [01] outer                              066:05 (var foo = throwAndCatch())
    [02] outermost                          074:05 (var result = outer())
```

此时，我们希望停止调试该脚本，因此我们将其与脚本分离。

dbgcmd

```
>>> Debug [DebuggableSample 34:5] >.detach
.detach
Debugger has been detached from script!
```

然后，键入 q 退出。

dbgcmd

```
q
This is a fun test
Of the script debugger
foo.a = 99
Caught and returned!
Test
```

再次执行该函数将不再中断调试器。

dbgcmd

```
0:007> dx @$scriptContents.outermost()
inside outer!
This is a fun test
Of the script debugger
foo.a = 99
Caught and returned!
Test
```

VSCode 中的 JavaScript - 添加 IntelliSense

如果要在 VSCode 中使用调试器数据模型对象，可以使用 Windows 开发工具包中提供的定义文件。 IntelliSense 定义文件提供对所有 host.* 调试器对象 API 的支持。 如果在 64 位电脑的默认目录中安装了该工具包，则它位于以下位置：

`C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\winext\JsProvider.d.ts`

若要在 VSCode 中使用 IntelliSense 定义文件，请执行以下操作：

1. 查找定义文件 - JSProvider.d.ts

2. 将定义文件复制到与脚本相同的文件夹中。

3. 将 `/// <reference path="JSProvider.d.ts" />` 添加到 JavaScript 脚本文件的顶部。

在 JavaScript 文件中有了这个引用，除了脚本中的结构外，VS Code 还会自动为你提供 JSProvider 提供的主机 API 上的 IntelliSense。 例如，键入“host”。 将看到所有可用调试器模型 API 的 IntelliSense。

JavaScript 资源

以下是开发 JavaScript 调试扩展时可能有用的 JavaScript 资源。

- 编写 JavaScript 代码
- JScript 语言教程
- Mozilla JavaScript 参考
- WinJS：Windows JavaScript 库
- ECMAScript 6 - 新功能：概述和比较

另请参阅

JavaScript 调试器示例脚本

JavaScript 扩展中的本机对象

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

JavaScript 调试器示例脚本

项目 • 2023/04/14

本主题提供以下用户和内核模式 JavaScript 代码示例。

- 确定进程体系结构
- 数据筛选：在 KD (内核模式下即插即用设备树)
- 扩展特定于多媒体 (内核模式的设备)
- 将总线信息添加到_DEVICE_OBJECT (内核模式)
- (用户模式) 查找应用程序标题

Microsoft GitHub 存储库示例脚本

调试器团队托管包含示例 JavaScript 脚本和扩展的 GitHub 存储库。

可在 以下位置找到它： https://github.com/Microsoft/WinDbg-Samples

自述文件描述可用的当前示例代码。

使用示例

使用常规过程来测试任何示例。

1. 确定示例 JavaScript 是用于内核还是用户模式调试。 然后加载适当的转储文件或与目标系统建立实时连接。
2. 使用文本编辑器（如记事本）创建名为 的文本文件，并使用.js文件扩展名保存该文件，例如 *HelloWorld.js*

JavaScript

```
// WinDbg JavaScript sample
// Says Hello World!

// Code at root will be run with .scriptrun and .scriptload
host.diagnostics.debugLog("***> Hello World! \n");

function sayHi()
{
   //Say Hi
   host.diagnostics.debugLog("Hi from JavaScript! \n");
}
```

3. 使用 **.scriptrun (运行脚本)** 命令加载和执行脚本。 .scriptrun 命令将在根/顶部运行代码，并在函数名称 *initializeScript* 和 *invokeScript* 下运行代码。

dbgcmd

```
0:000> .scriptrun c:\WinDbg\Scripts\HelloWorld.js
JavaScript script successfully loaded from 'c:\WinDbg\Scripts\HelloWorld.js'
***> Hello World!
```

4. 如果脚本包含唯一命名的函数，请使用 dx 命令执行位于 Debugger.State.Scripts 中的函数。*ScriptName*。内容。*FunctionName*。

dbgcmd

```
0:001> dx Debugger.State.Scripts.HelloWorld.Contents.sayHi()
Hi from JavaScript!
Debugger.State.Scripts.HelloWorld.Contents.sayHi()
```

有关使用 JavaScript 的其他信息，请参阅 JavaScript 调试器脚本 。

确定进程体系结构

此 JavaScript 代码将名为“ProcessArchitecture”的属性添加到调试器对象模型进程对象，以指示进程是 x86 还是 x64。

此脚本旨在支持内核模式调试。

JavaScript

```
"use strict";

class __CheckArchitecture
{
//
// Add a property called 'ProcessArchitecture' on process.
//
    get ProcessArchitecture()
    {
    var guestStates = this.Threads.Any(t=> (!(t.GuestState === undefined) &&
t.GuestState.Architecture =="x86"));

        if(guestStates)
            return "x86";
        else
            return "x64";
    }
};
```

```
function initializeScript()
{
//
// Extends our notion of a process to place architecture information on it.
//
return [new host.namedModelParent(__CheckArchitecture,
"Debugger.Models.Process")];
}
```

加载内核转储文件或建立与目标系统的内核模式连接。 然后加载 JavaScript 提供程序和示例脚本。

dbgcmd

```
0: kd> !load jsprovider.dll
```

dbgcmd

```
0: kd> .scriptload c:\WinDbg\Scripts\processarchitecture.js
JavaScript script successfully loaded from
'c:\WinDbg\Scripts\processarchitecture.js'
```

使用 dx 命令显示当前进程的进程体系结构。

dbgcmd

```
2: kd> dx @$curprocess
@$curprocess                 : System [Switch To]
    KernelObject     [Type: _EPROCESS]
    Name             : System
    Id               : 0x4
    Handle           : 0xf0f0f0f0
    Threads
    Modules
    Environment
    Devices
    Io
    ProcessArchitecture : x64
```

请注意，此示例代码可能并不总是能够正确确定体系结构。 例如，在某些情况下，在使用 32 位调试器时处理转储文件。

数据筛选：在 KD (内核模式下即插即用设备树)

此示例代码筛选设备节点树，以仅显示包含已启动的 PCI 路径的设备。

此脚本旨在支持实时内核模式调试。

可以使用！ devnode 0 1 命令显示有关设备树的信息。 有关详细信息，请参阅 **！devnode**。

JavaScript

```
// PlugAndPlayDeviceTree.js
// An ES6 generator function which recursively filters the device tree
looking for PCI devices in the started state.
//
function *filterDevices(deviceNode)
{
    //
    // If the device instance path has "PCI" in it and is started (state ==
776), yield it from the generator.
    //
    if (deviceNode.InstancePath.indexOf("PCI") != -1 && deviceNode.State ==
776)
    {
        yield deviceNode;
    }

    //
    // Recursively invoke the generator for all children of the device node.
    //
    for (var childNode of deviceNode.Children)
    {
        yield* filterDevices(childNode);
    }
}

//
// A function which finds the device tree of the first session in the
debugger and passes it to our filter function.
//
function filterAllDevices()
{
    return
filterDevices(host.namespace.Debugger.Sessions.First().Devices.DeviceTree.Fi
rst());
}
```

建立与目标系统的内核模式连接。

dbgcmd

```
0: kd> !load jsprovider.dll
```

dbgcmd

```
0: kd> .scriptload c:\WinDbg\Scripts\PlugAndPlayDeviceTree.js
JavaScript script successfully loaded from
'c:\WinDbg\Scripts\PlugAndPlayDeviceTree.js'
```

调用 filterAllDevices () 函数。

dbgcmd

```
0: kd> dx
Debugger.State.Scripts.PlugAndPlayDeviceTree.Contents.filterAllDevices()
Debugger.State.Scripts.PlugAndPlayDeviceTree.Contents.filterAllDevices()
: [object Generator]
    [0x0]            :
PCI\VEN_8086&DEV_D131&SUBSYS_304A103C&REV_11\3&21436425&0&00
    [0x1]            :
PCI\VEN_8086&DEV_D138&SUBSYS_304A103C&REV_11\3&21436425&0&18 (pci)
    [0x2]            :
PCI\VEN_10DE&DEV_06FD&SUBSYS_062E10DE&REV_A1\4&324c21a&0&0018 (nvlddmkm)
    [0x3]            :
PCI\VEN_8086&DEV_3B64&SUBSYS_304A103C&REV_06\3&21436425&0&B0 (HECIx64)
    [0x4]            :
PCI\VEN_8086&DEV_3B3C&SUBSYS_304A103C&REV_05\3&21436425&0&D0 (usbehci)
    [0x5]            :
PCI\VEN_8086&DEV_3B56&SUBSYS_304A103C&REV_05\3&21436425&0&D8 (HDAudBus)
...
```

上述每个对象都自动支持 DML，并且可以与任何其他 dx 查询一样进行选择。

或者，若要使用此脚本，还可以使用 LINQ 查询来实现类似的结果。

dbgcmd

```
0: kd> dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n
=> n.InstancePath.Contains("PCI") && n.State == 776)
@$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n =>
n.InstancePath.Contains("PCI") && n.State == 776)
    [0x0]            :
PCI\VEN_8086&DEV_D131&SUBSYS_304A103C&REV_11\3&21436425&0&00
    [0x1]            :
PCI\VEN_8086&DEV_D138&SUBSYS_304A103C&REV_11\3&21436425&0&18 (pci)
    [0x2]            :
PCI\VEN_10DE&DEV_06FD&SUBSYS_062E10DE&REV_A1\4&324c21a&0&0018 (nvlddmkm)
    [0x3]            :
PCI\VEN_8086&DEV_3B64&SUBSYS_304A103C&REV_06\3&21436425&0&B0 (HECIx64)
    [0x4]            :
PCI\VEN_8086&DEV_3B3C&SUBSYS_304A103C&REV_05\3&21436425&0&D0 (usbehci)
    [0x5]            :
PCI\VEN_8086&DEV_3B56&SUBSYS_304A103C&REV_05\3&21436425&0&D8 (HDAudBus)
...
```

将特定于多媒体的设备 (内核模式)

这个更大的 JavaScript 示例扩展了内核_DEVICE_OBJECT，以获取特定于多媒体的信息，并将 StreamingDevices 添加到调试器会话。

此脚本旨在支持内核模式调试。

请注意，使用 StreamingDevices 扩展会话的选择仅用于例如目的。 这应该只留到_DEVICE_OBJECT，或更深入地位于现有 下的命名空间内。Devices.* 层次结构。

JavaScript

```
// StreamingFinder.js
// Extends a kernel _DEVICE_OBJECT for information specific to multimedia
// and adds StreamingDevices to a debugger session.
//

"use strict";

function initializeScript()
{
    // isStreamingDeviceObject:
    //
    // Returns whether devObj (as a _DEVICE_OBJECT -- not a pointer) looks
like it is a device
    // object for a streaming device.
    //
    function isStreamingDeviceObject(devObj)
    {
        try
        {
            var devExt = devObj.DeviceExtension;
            var possibleStreamingExtPtrPtr =
host.createPointerObject(devExt.address, "ks.sys", "_KSIDEVICE_HEADER **",
devObj);
            var possibleStreamingExt =
possibleStreamingExtPtrPtr.dereference().dereference();
            var baseDevice = possibleStreamingExt.BaseDevice;

            if (devObj.targetLocation ==
baseDevice.dereference().targetLocation)
            {
                return true;
            }
        }
        //
        // The above code expects to fail (walking into invalid or paged out
memory) often.  A failure to read the memory
        // of the target process will result in an exception.  Catch such
exception and indicate that the object does not
        // match the profile of a "streaming device object".
        //
```

```
        catch(exc)
        {
        }

        return false;
    }

    // findStreamingFDO(pdo):
    //
    // From a physical device object, walks up the device stack and attempts
to find a device which
    // looks like a streaming device (and is thus assumed to be the FDO).
pdo is a pointer to
    // the _DEVICE_OBJECT for the pdo.
    //
    function findStreamingFDO(pdo)
    {
        for (var device of pdo.UpperDevices)
        {
            if (isStreamingDeviceObject(device.dereference()))
            {
                return device;
            }
        }

        return null;
    }

    // streamingDeviceList:
    //
    // A class which enumerates all streaming devices on the system.
    //
    class streamingDeviceList
    {
        constructor(session)
        {
            this.__session = session;
        }

        *[Symbol.iterator]()
        {
            //
            // Get the list of all PDOs from PNP using LINQ:
            //
            var allPDOs =
this.__session.Devices.DeviceTree.Flatten(function(dev) { return
dev.Children; })
                                                              .Select
(function(dev) { return dev.PhysicalDeviceObject; });

            //
            // Walk the stack up each PDO to find the functional device
which looks like a KS device.  This is
            // a very simple heuristic test: the back pointer to the device
in the KS device extension is
```

```
            // accurate.  Make sure this is wrapped in a try/catch to avoid
invalid memory reads causing
            // us to bail out.
            //
            // Don't even bother checking the PDO.
            //
            for (var pdo of allPDOs)
            {
                var fdo = findStreamingFDO(pdo);
                if (fdo != null)
                {
                    yield fdo;
                }
            }
        }
    }

    // streamingDeviceExtension:
    //
    // An object which will extend "session" and include the list of
streaming devices.
    //
    class streamingDeviceExtension
    {
        get StreamingDevices()
        {
            return new streamingDeviceList(this);
        }
    };

    // createEntryList:
    //
    // An abstraction over the create entry list within a streaming device.
    //
    class createEntryList
    {
        constructor(ksHeader)
        {
            this.__ksHeader = ksHeader;
        }

        *[Symbol.iterator]()
        {
            for (var entry of
host.namespace.Debugger.Utility.Collections.FromListEntry(this.__ksHeader.Ch
ildCreateHandlerList, "ks!KSICREATE_ENTRY", "ListEntry"))
            {
                                                                if
(!entry.CreateItem.Create.isNull)
                {
                    yield entry;
                }
            }
        }
    }
```

```
    // streamingInformation:
    //
    // Represents the streaming state of a device.
    //
    class streamingInformation
    {
        constructor(fdo)
        {
            this.__fdo = fdo;

            var devExt = fdo.DeviceExtension;
            var streamingExtPtrPtr =
host.createPointerObject(devExt.address, "ks.sys", "_KSIDEVICE_HEADER **",
fdo);
            this.__ksHeader =
streamingExtPtrPtr.dereference().dereference();
        }

        get CreateEntries()
        {
            return new createEntryList(this.__ksHeader);
        }
    }

    // createEntryVisualizer:
    //
    // A visualizer for KSICREATE_ENTRY
    //
    class createEntryVisualizer
    {
        toString()
        {
            return this.CreateItem.ObjectClass.toString();
        }

        get CreateContext()
        {
            //
            // This is probably not entirely accurate.  The context is not
*REQUIRED* to be an IUnknown.
            // More analysis should probably be performed.
            //
            return host.createTypedObject(this.CreateItem.Context.address,
"ks.sys", "IUnknown", this);
        }
    }

    // deviceExtension:
    //
    // Extends our notion of a device in the device tree to place streaming
information on
    // top of it.
    //
    class deviceExtension
```

```
    {
        get StreamingState()
        {
            if (isStreamingDeviceObject(this))
            {
                return new streamingInformation(this);
            }

            //
            // If we cannot find a streaming FDO, returning undefined will
indicate that there is no value
            // to the property.
            //
            return undefined;
        }
    }

    return [new host.namedModelParent(streamingDeviceExtension,
"Debugger.Models.Session"),
            new host.typeSignatureExtension(deviceExtension,
"_DEVICE_OBJECT"),
            new host.typeSignatureRegistration(createEntryVisualizer,
"KSICREATE_ENTRY")]
}
```

首先，如前所述加载脚本提供程序。 然后加载脚本。

dbgcmd

```
0: kd> .scriptload c:\WinDbg\Scripts\StreamingFinder.js
JavaScript script successfully loaded from
'c:\WinDbg\Scripts\StreamingFinder.js'
```

然后使用 dx 命令访问脚本提供的新 StreamingDevices 功能。

dbgcmd

```
0: kd> dx -r3 @$cursession.StreamingDevices.Select(d => d-
>StreamingState.CreateEntries)
@$cursession.StreamingDevices.Select(d => d->StreamingState.CreateEntries)
    [0x0]            : [object Object]
        [0x0]            : "e0HDMIOutTopo" [Type: KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: CPortTopology]
    [0x1]            : [object Object]
        [0x0]            : "AnalogDigitalCaptureTopo" [Type:
KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: CPortTopology]
        [0x1]            : "AnalogDigitalCapture1Topo" [Type:
KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
```

```
            CreateContext    [Type: CPortTopology]
        [0x2]            : "AnalogDigitalCapture2Topo" [Type:
KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: CPortTopology]
        [0x3]            : "AnalogDigitalCapture2Wave" [Type:
KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: CPortWaveRT]
        [0x4]            : "HeadphoneTopo" [Type: KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: CPortTopology]
        [0x5]            : "RearLineOutTopo" [Type: KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: CPortTopology]
        [0x6]            : "RearLineOutWave" [Type: KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: CPortWaveRT]
    [0x2]            : [object Object]
        [0x0]            : "GLOBAL" [Type: KSICREATE_ENTRY]
            [<Raw View>]     [Type: KSICREATE_ENTRY]
            CreateContext    [Type: IUnknown]
```

将总线信息添加到_DEVICE_OBJECT (内核模式)

此脚本扩展了 _DEVICE_OBJECT 的可视化效果，以添加其下方包含 PCI 特定信息的 BusInformation 字段。 仍在讨论此示例的方式和名称范围。 它应被视为 JavaScript 提供程序功能的示例。

此脚本旨在支持内核模式调试。

JavaScript

```javascript
"use strict";

/*********************************************
DeviceExtensionInformation.js:

An example which extends _DEVICE_OBJECT to add bus specific information
to each device object.

NOTE: The means of conditionally adding and the style of namespacing this
are still being discussed.  This currently serves as an example of the
capability
of JavaScript extensions.

*********************************************/

function initializeScript()
{
    // __getStackPDO():
```

```
    //
    // Returns the physical device object of the device stack whose device
object
    // is passed in as 'devObj'.
    //
    function __getStackPDO(devObj)
    {
        var curDevice = devObj;
        var nextDevice = curDevice.DeviceObjectExtension.AttachedTo;
        while (!nextDevice.isNull)
        {
            curDevice = nextDevice;
            nextDevice = curDevice.DeviceObjectExtension.AttachedTo;
        }
        return curDevice;
    }

    // pciInformation:
    //
    // Class which abstracts our particular representation of PCI
information for a PCI device or bus
    // based on a _PCI_DEVICE structure or a _PCI_BUS structure.
    //
    class pciInformation
    {
        constructor(pciDev)
        {
            this.__pciDev = pciDev;
            this.__pciPDO = __getStackPDO(this.__pciDev);
            this.__isBridge = (this.__pciDev.address !=
this.__pciPDO.address);

            if (this.__isBridge)
            {
                this.__deviceExtension =
host.createTypedObject(this.__pciPDO.DeviceExtension.address, "pci.sys",
"_PCI_DEVICE", this.__pciPDO);
                this.__busExtension =
host.createTypedObject(this.__pciDev.DeviceExtension.address, "pci.sys",
"_PCI_BUS", this.__pciPDO);

                this.__hasDevice = (this.__deviceExtension.Signature ==
0x44696350); /* 'PciD' */
                this.__hasBus = (this.__busExtension.Signature ==
0x42696350); /* 'PciB' */

                if (!this.__hasDevice && !this.__hasBus)
                {
                    throw new Error("Unrecognized PCI device extension");
                }
            }
            else
            {
                this.__deviceExtension =
host.createTypedObject(this.__pciPDO.DeviceExtension.address, "pci.sys",
```

```
"_PCI_DEVICE", this.__pciPDO);

                this.__hasDevice = (this.__deviceExtension.Signature ==
0x44696350); /* 'PciD' */
                this.__hasBus = false;

                if (!this.__hasDevice)
                {
                    throw new Error("Unrecognized PCI device extension");
                }
            }
        }

        toString()
        {

            if (this.__hasBus && this.__hasDevice)
            {
                return "Bus: " + this.__busExtension.toString() + " Device:
" + this.__deviceExtension.toString();
            }
            else if (this.__hasBus)
            {
                return this.__busExtension.toString();
            }
            else
            {
                return this.__deviceExtension.toString();
            }
        }

        get Device()
        {
            if (this.__hasDevice)
            {
                // NatVis supplies the visualization for _PCI_DEVICE
                return this.__deviceExtension;
            }

            return undefined;
        }

        get Bus()
        {
            if (this.__hasBus)
            {
                // NatVis supplies the visualization for _PCI_BUS
                return this.__busExtension;
            }

            return undefined;
        }
    }

    // busInformation:
```

```
    //
    // Our class which does analysis of what bus a particular device is on
and places information about
    // about that bus within the _DEVICE_OBJECT visualization.
    //
    class busInformation
    {
        constructor(devObj)
        {
            this.__devObj = devObj;
        }

        get PCI()
        {
            //
            // Check the current device object.  This may be a PCI bridge
            // in which both the FDO and PDO are relevant (one for the bus
information
            // and one for the bridge device information).
            //
            var curName = this.__devObj.DriverObject.DriverName.toString();
            if (curName.includes("\\Driver\\pci"))
            {
                return new pciInformation(this.__devObj);
            }

            var stackPDO = __getStackPDO(this.__devObj);
            var pdoName = stackPDO.DriverObject.DriverName.toString();
            if (pdoName.includes("\\Driver\\pci"))
            {
                return new pciInformation(stackPDO);
            }

            return undefined;
        }
    }

    // busInformationExtension:
    //
    // An extension placed on top of _DEVICE_OBJECT in order to provide bus
analysis and bus specific
    // information to the device.
    //
    class busInformationExtension
    {
        get BusInformation()
        {
            return new busInformation(this);
        }
    }

    return [new host.typeSignatureExtension(busInformationExtension,
"_DEVICE_OBJECT")];
}
```

首先，如前所述加载脚本提供程序。 然后加载脚本。

dbgcmd

```
0: kd> .scriptload c:\WinDbg\Scripts\DeviceExtensionInformation.js
JavaScript script successfully loaded from
'c:\WinDbg\Scripts\DeviceExtensionInformation.js'
```

我们需要找到感兴趣的设备对象的地址。 在此示例中，我们将检查音频 HDAudBus 驱动程序。

dbgcmd

```
0: kd>  !drvobj HDAudBus
Driver object (ffffb60757a4ae60) is for:
 \Driver\HDAudBus
Driver Extension List: (id , addr)
(fffff8050a9eb290 ffffb60758413180)
Device Object list:
ffffb60758e21810  ffffb60757a67c60
```

加载脚本后，使用 dx 命令显示设备对象的总线信息。

dbgcmd

```
0: kd> dx -r1 (*((ntkrnlmp!_DEVICE_OBJECT *)0xffffe00001b567c0))
(*((ntkrnlmp!_DEVICE_OBJECT *)0xffffe00001b567c0))                 : Device
for "\Driver\HDAudBus" [Type: _DEVICE_OBJECT]
    [<Raw View>]     [Type: _DEVICE_OBJECT]
    Flags            : 0x2004
    UpperDevices     : None
    LowerDevices     : Immediately below is Device for "\Driver\ACPI" [at
0xffffe000003d9820]
    Driver           : 0xffffe00001ccd060 : Driver "\Driver\HDAudBus" [Type:
_DRIVER_OBJECT *]
    BusInformation   : [object Object]

0: kd> dx -r1 (*((ntkrnlmp!_DEVICE_OBJECT
*)0xffffe00001b567c0)).@"BusInformation"
(*((ntkrnlmp!_DEVICE_OBJECT *)0xffffe00001b567c0)).@"BusInformation"
: [object Object]
    PCI              :  (d=0x14 f=0x2) Vendor=0x1022 Device=0x780d
Multimedia Device / Unknown Sub Class

0: kd> dx -r1 (*((ntkrnlmp!_DEVICE_OBJECT
*)0xffffe00001b567c0)).@"BusInformation".@"PCI"
(*((ntkrnlmp!_DEVICE_OBJECT *)0xffffe00001b567c0)).@"BusInformation".@"PCI"
:  (d=0x14 f=0x2) Vendor=0x1022 Device=0x780d Multimedia Device / Unknown
Sub Class
    Device           :  (d=0x14 f=0x2) Vendor=0x1022 Device=0x780d
Multimedia Device / Unknown Sub Class [Type: _PCI_DEVICE]
```

```
0: kd> dx -r1 (*((pci!_PCI_DEVICE *)0xffffe000003fe1b0))
(*((pci!_PCI_DEVICE *)0xffffe000003fe1b0))                 :  (d=0x14 f=0x2)
Vendor=0x1022 Device=0x780d Multimedia Device / Unknown Sub Class [Type:
_PCI_DEVICE]
    [<Raw View>]     [Type: _PCI_DEVICE]
    Device           : 0xffffe000003fe060 : Device for "\Driver\pci" [Type:
_DEVICE_OBJECT *]
    Requirements
    Resources

0: kd> dx -r1 (*((pci!_PCI_DEVICE *)0xffffe000003fe1b0)).@"Resources"
(*((pci!_PCI_DEVICE *)0xffffe000003fe1b0)).@"Resources"
    BaseAddressRegisters
    Interrupt        : Line Based -- Interrupt Line = 0x10 [Type:
_PCI_DEVICE_INTERRUPT_RESOURCE]

0: kd> dx -r1 (*((pci!_PCI_DEVICE
*)0xffffe000003fe1b0)).@"Resources".@"BaseAddressRegisters"
(*((pci!_PCI_DEVICE
*)0xffffe000003fe1b0)).@"Resources".@"BaseAddressRegisters"
    [0x0]            : Memory Resource: 0xf0340000 of length 0x4000 [Type:
_CM_PARTIAL_RESOURCE_DESCRIPTOR]
```

在用户模式) (查找应用程序标题

此示例循环访问调试器当前进程中的所有线程，查找包含 *__mainCRTStartup* 的帧，然后从 CRT 启动中的 StartupInfo.lpTitle 返回字符串。 此脚本显示了 JavaScript 中的迭代、字符串操作和 LINQ 查询的示例。

此脚本旨在支持用户模式调试。

JavaScript

```
// TitleFinder.js
// A function which uses just JavaScript concepts to find the title of an
application
//
function findTitle()
{
    var curProcess = host.currentProcess;
    for (var thread of curProcess.Threads)
    {
        for (var frame of thread.Stack.Frames)
        {
            if (frame.toString().includes("__mainCRTStartup"))
            {
                var locals = frame.LocalVariables;

                //
                // locals.StartupInfo.lpTitle is just an unsigned short *.
```

```
We need to actually call an API to
                // read the UTF-16 string in the target address space.  This
would be true even if this were
                // a char* or wchar_t*.
                //
                return
host.memory.readWideString(locals.StartupInfo.lpTitle);
            }
        }
    }
}

//
// A function which uses both JavaScript and integrated LINQ concepts to do
the same.
//
function findTitleWithLINQ()
{
    var isMainFrame = function(frame) { return
frame.toString().includes("__mainCRTStartup"); };
    var isMainThread = function(thread) { return
thread.Stack.Frames.Any(isMainFrame); };

    var curProcess = host.currentProcess;
    var mainThread = curProcess.Threads.Where(isMainThread).First();
    var mainFrame = mainThread.Stack.Frames.Where(isMainFrame).First();

    var locals = mainFrame.LocalVariables;

    //
    // locals.StartupInfo.lpTitle is just an unsigned short *.  We need to
actually call an API to
    // read the UTF-16 string in the target address space.  This would be
true even if this were
    // a char* or wchar_t*.
    //
    return host.memory.readWideString(locals.StartupInfo.lpTitle);
}
```

dbgcmd

```
0: kd> .scriptload c:\WinDbg\Scripts\TitleFinder.js
JavaScript script successfully loaded from
'c:\WinDbg\Scripts\TitleFinder.js'
```

调用 findTitle () 函数将返回notepad.exe

dbgcmd

```
0:000> dx Debugger.State.Scripts.TitleFinder.Contents.findTitle()
Debugger.State.Scripts.TitleFinder.Contents.findTitle() :
C:\Windows\System32\notepad.exe
```

调用 LINQ 版本时，findTitleWithLINQ () 还会返回notepad.exe

dbgcmd

```
0:000> dx Debugger.State.Scripts.TitleFinder.Contents.findTitleWithLINQ()
Debugger.State.Scripts.titleFinder.Contents.findTitleWithLINQ() :
C:\Windows\System32\notepad.exe
```

另请参阅

JavaScript 调试器脚本

JavaScript 扩展中的本机调试器对象

项目 • 2023/04/14

本机调试器对象表示调试器环境的各种构造和行为。 对象可以传递到 (中，或者在) JavaScript 扩展中获取，以操作调试器的状态。

示例调试器对象包括以下内容。

- 会话
- 线程/线程
- 流程/流程
- 堆栈帧/堆栈帧
- 局部变量
- 模块/模块
- 实用工具
- 状态
- 设置

例如，host.namespace.Debugger.Utility.Control.ExecuteCommand 对象可用于使用以下两行 JavaScript 代码将 u 命令发送到调试器。

dbgcmd

```
var ctl = host.namespace.Debugger.Utility.Control;
var outputLines = ctl.ExecuteCommand("u");
```

本主题介绍如何使用常见对象，并提供有关其属性和行为的参考信息。

有关使用 JavaScript 的一般信息，请参阅 JavaScript 调试器脚本。 有关使用调试器对象的 JavaScript 示例，请参阅 JavaScript 调试器示例脚本。 有关使用设置对象的信息，请参阅 **.settings () 设置调试设置** 。

若要浏览调试器会话中可用的对象，请使用 **dx (Display NatVis Expression)** 命令。 例如，可以使用此 dx 命令显示一些顶级调试器对象。

dbgcmd

```
0: kd> dx -r2 Debugger
Debugger
    Sessions         : [object Object]
        [0x0]            : Remote KD:
KdSrv:Server=@{<Local>},Trans=@{NET:Port=50000,Key=1.2.3.4,Target}
    Settings
        Debug
        Display
```

```
        EngineInitialization
        Extensions
        Input
        Sources
        Symbols
        AutoSaveSettings : false
    State
        DebuggerVariables
        PseudoRegisters
        Scripts
        UserVariables
    Utility
        Collections
        Control
        Objects
```

上面列出的所有项都是可单击的 DML，可以进一步递归以查看调试器对象结构。

通过数据模型扩展调试器

调试器数据模型允许创建一个接口，以访问 Windows 中具有以下属性的应用程序和驱动程序的相关信息。

- 可发现和组织 - 可以使用 dx 命令查询逻辑结构的名称空间。
- 可以使用 LINQ 查询 - 这允许使用标准查询语言提取和排序数据。
- 可以在逻辑上一致地扩展 - 使用本主题中所述的技术与调试器脚本提供程序（如 Natvis 和 JavaScript）可扩展。

在 JavaScript 中扩展调试器对象

除了能够在 JavaScript 中创建可视化工具外，脚本扩展还可以修改调试器的核心概念（会话、进程、线程、堆栈、堆栈帧、局部变量），甚至可以将自身发布为其他扩展可以使用的扩展点。

本部分介绍如何在调试器中扩展核心概念。 构建为共享的扩展应符合 JavaScript 扩展中的本机调试器对象 - 设计和测试注意事项中介绍的准则。

注册扩展

脚本可以注册它通过从 initializeScript 方法返回的数组中的条目提供扩展的事实。

dbgcmd

```
function initializeScript()
{
    return [new host.namedModelParent(comProcessExtension,
```

```
"Debugger.Models.Process")];
}
```

如果返回的数组中存在 host.namedModelParent 对象，则向调试器指示给定的原型对象或 ES6 类 (comProcessExtension 在本例中，) 将成为以 Debugger.Models.Process 名称注册的模型的父数据模型。

调试器对象扩展点

以下调试器扩展点是调试器的组成部分，可供 JavaScript 等脚本提供程序使用。

Debugger.Models.Sessions：调试器附加到) (目标的会话列表

Debugger.Models.Session：将调试器附加到实时用户模式、KD 等 (单个会话 (目标) ...)

Debugger.Models.Processes：会话中的进程列表

Debugger.Models.Threads：进程中的线程列表

Debugger.Models.Thread：进程内的单个线程 (，无论用户模式还是内核模式)

Debugger.Models.Stack：线程堆栈

Debugger.Models.StackFrames：构成堆栈的帧集合

Debugger.Models.StackFrame：堆栈中的单个堆栈帧

Debugger.Models.LocalVariables：堆栈帧中的局部变量

Debugger.Models.Parameters：堆栈帧内调用的参数

Debugger.Models.Module：进程地址空间中的单个模块

其他数据模型对象

此外，还有一些由核心数据模型定义的附加数据模型对象。

DataModel.Models.Intrinsic：(序号、浮点数等的内部值...)

DataModel.Models.String：字符串

DataModel.Models.Array：本机数组

DataModel.Models.Guid：GUID

DataModel.Models.Error：错误对象

DataModel.Models.Concepts.Iterable：应用于可迭代的每个对象

DataModel.Models.Concepts.StringDisplayable：应用于具有显示字符串转换的每个对象

示例 COM 调试器对象扩展概述

我们分析一个示例。 假设你想要创建一个调试器扩展来显示特定于 COM 的信息，例如全局接口表 (GIT) 。

过去，可能存在具有多个命令的现有调试器扩展，这些命令提供了访问有关 COM 的内容的方法。 一个命令可能会 (全局接口表（例如)) 显示以进程为中心的信息。 另一个命令可能会提供以线程为中心的信息，例如在哪个单元代码中执行。 可能需要了解并加载第二个调试器扩展，以探索 COM 的其他方面。

JavaScript 扩展可以修改调试器的进程和线程的概念，而不是有一组难以发现的命令，以自然、可探索且可与其他调试器扩展组合的方式添加此信息。

用户或内核模式调试器对象扩展

调试器和调试器对象在用户和内核模式下具有不同的行为。 创建调试器模型对象时，需要确定将在哪些环境中工作。 由于我们将在用户模式下使用 COM，因此我们将在用户模式下创建并测试此 com 扩展。 在其他情况下，你可能能够创建一个调试器 JavaScript，该调试器可在用户和内核模式调试中运行。

创建子命名空间

回到我们的示例，我们可以定义原型或 ES6 类 *comProcessExtension* ，其中包含要添加到进程对象的一组内容。

重要 使用 子命名空间的目的是创建一个逻辑结构化且自然可探索的范例。 例如，避免将不相关的项转储到同一子命名空间中。 在创建子命名空间之前，请仔细查看 JavaScript 扩展中的本机调试器对象 - 设计和测试注意事项 中讨论的信息。

在此代码片段中，我们将创建一个名为“COM”的子命名空间添加到现有进程调试器对象。

JavaScript

```
var comProcessExtension =
{
    //
    // Add a sub-namespace called 'COM' on process.
    //
    get COM()
    {
        //
        // What is 'this' below...?  It's the debugger's process object.
Yes -- this means that there is a cross-language
        // object hierarchy here.  A C++ object implemented in the debugger
has a parent model (prototype) which is
```

```
        // implemented in JavaScript.
        //
        return new comNamespace(this);
    }
}
```

命名空间实现

接下来，创建对象，该对象在进程上实现子命名空间 COM。

重要 可以有多个进程 (无论是在用户模式下还是在 KD) 下附加到此类进程。 此扩展不能假定调试器的当前状态是用户的预期状态。 有人可以在变量中捕获 <someProcess.COM> 并对其进行修改，这可能会导致显示来自错误进程上下文的信息。 解决方案是在扩展中添加代码，以便每个实例化都会跟踪它附加到的进程。 对于此代码示例，此信息通过 属性的“this”指针传递。

`this.__process = process;`

JavaScript

```
class comNamespace
{
    constructor(process)
    {
        //
        // This is an entirely JavaScript object.  Each instantiation of a
comNamespace will keep track
        // of what process it is attached to (passed via the ''this''
pointer of the property getter
        // we authored above.
        //
        this.__process = process;
    }

    get GlobalObjects()
    {
        return new globalObjects(this.__process);
    }
}
```

COM 全局接口表的实现逻辑

为了更清楚地分离 COM 全局接口表的实现逻辑，我们将定义一个 ES6 类 *gipTable* ，该类抽象出 COM GIP 表，另一个 *globalObjects*，后者将从上面所示的命名空间实现代码截图中定义的 GlobalObjects () getter 中返回。 所有这些详细信息都可以隐藏在 initializeScript 的闭包中，以避免将这些内部详细信息发布到调试器命名空间中。

JavaScript

```
// gipTable:
//
// Internal class which abstracts away the GIP Table.  It iterates objects
of the form
// {entry : GIPEntry, cookie : GIT cookie}
//
class gipTable
{
    constructor(gipProcess)
    {
        //
        // Windows 8 through certain builds of Windows 10, it's in
CGIPTable::_palloc.  In certain builds
        // of Windows 10 and later, this has been moved to
GIPEntry::_palloc.  We need to check which.
        //
        var gipAllocator = undefined;
        try
        {
            gipAllocator = host.getModuleSymbol("combase.dll",
"CGIPTable::_palloc", "CPageAllocator", gipProcess)._pgalloc;
        }
        catch(err)
        {
        }

        if (gipAllocator == undefined)
        {
            gipAllocator = host.getModuleSymbol("combase.dll",
"GIPEntry::_palloc", "CPageAllocator", gipProcess)._pgalloc;
        }

        this.__data = {
            process : gipProcess,
            allocator : gipAllocator,
            pageList : gipAllocator._pPageListStart,
            pageCount : gipAllocator._cPages,
            entriesPerPage : gipAllocator._cEntriesPerPage,
            bytesPerEntry : gipAllocator._cbPerEntry,
            PAGESHIFT : 16,
            PAGEMASK : 0x0000FFFF,
            SEQNOMASK : 0xFF00
        };
    }

    *[Symbol.iterator]()
    {
        for (var pageNum = 0; pageNum < this.__data.pageCount; ++pageNum)
        {
            var page = this.__data.pageList[pageNum];
            for (var entryNum = 0; entryNum < this.__data.entriesPerPage;
++entryNum)
            {
                var entryAddress =
page.address.add(this.__data.bytesPerEntry * entryNum);
```

```
                var gipEntry = host.createPointerObject(entryAddress,
"combase.dll", "GIPEntry *", this.__data.process);
                if (gipEntry.cUsage != -1 && gipEntry.dwType != 0)
                {
                    yield {entry : gipEntry, cookie : (gipEntry.dwSeqNo |
(pageNum << this.__data.PAGESHIFT) | entryNum)};
                }
            }
        }
    }

    entryFromCookie(cookie)
    {
        var sequenceNo = (cookie & this.__data.SEQNOMASK);
        cookie = cookie & ~sequenceNo;
        var pageNum = (cookie >> this.__data.PAGESHIFT);
        if (pageNum < this.__data.pageCount)
        {
            var page = this.__data.pageList[pageNum];
            var entryNum = (cookie & this.__data.PAGEMASK);
            if (entryNum < this.__data.entriesPerPage)
            {
                var entryAddress =
page.address.add(this.__data.bytesPerEntry * entryNum);
                var gipEntry = host.createPointerObject(entryAddress,
"combase.dll", "GIPEntry *", this.__data.process);
                if (gipEntry.cUsage != -1 && gipEntry.dwType != 0 &&
gipEntry.dwSeqNo == sequenceNo)
                {
                    return {entry : gipEntry, cookie : (gipEntry.dwSeqNo |
(pageNum << this.__data.PAGESHIFT) | entryNum)};
                }
            }
        }

        //
        // If this exception flows back to C/C++, it will be a failed
HRESULT (according to the type of error -- here E_BOUNDS)
        // with the message being encapsulated by an error object.
        //
        throw new RangeError("Unable to find specified value");
    }
}




// globalObjects:
//
// The class which presents how we want the GIP table to look to the data
model.  It iterates the actual objects
// in the GIP table indexed by their cookie.
//
class globalObjects
```

```
{
    constructor(process)
    {
        this.__gipTable = new gipTable(process);
    }

    *[Symbol.iterator]()
    {
        for (var gipCombo of this.__gipTable)
        {
            yield new host.indexedValue(gipCombo.entry.pUnk,
[gipCombo.cookie]);
        }
    }

    getDimensionality()
    {
        return 1;
    }

    getValueAt(cookie)
    {
        return this.__gipTable.entryFromCookie(cookie).entry.pUnk;
    }
}
```

最后，使用 host.namedModelRegistration 注册新的 COM 功能。

JavaScript

```
function initializeScript()
{
    return [new host.namedModelParent(comProcessExtension,
"Debugger.Models.Process"),
            new host.namedModelRegistration(comNamespace,
"Debugger.Models.ComProcess")];
}
```

使用记事本等应用程序将代码保存到GipTableAbstractor.js。

下面是加载此扩展之前在用户模式下可用的进程信息。

dbgcmd

```
0:000:x86> dx @$curprocess
@$curprocess                 : DataBinding.exe
    Name             : DataBinding.exe
    Id               : 0x1b9c
    Threads
    Modules
```

加载 JavaScript 扩展。

dbgcmd

```
0:000:x86> .scriptload C:\JSExtensions\GipTableAbstractor.js
JavaScript script successfully loaded from
'C:\JSExtensions\GipTableAbstractor.js'
```

然后使用 dx 命令通过预定义的 @$curprocess显示有关进程的信息。

dbgcmd

```
0:000:x86> dx @$curprocess
@$curprocess                 : DataBinding.exe
    Name             : DataBinding.exe
    Id               : 0x1b9c
    Threads
    Modules
    COM              : [object Object]
```

dbgcmd

```
0:000:x86> dx @$curprocess.COM
@$curprocess.COM                 : [object Object]
    GlobalObjects    : [object Object]
0:000:x86> dx @$curprocess.COM.GlobalObjects
@$curprocess.COM.GlobalObjects                  : [object Object]
    [0x100]          : 0x12f4fb0 [Type: IUnknown *]
    [0x201]          : 0x37cfc50 [Type: IUnknown *]
    [0x302]          : 0x37ea910 [Type: IUnknown *]
    [0x403]          : 0x37fcfe0 [Type: IUnknown *]
    [0x504]          : 0x12fe1d0 [Type: IUnknown *]
    [0x605]          : 0x59f04e8 [Type: IUnknown *]
    [0x706]          : 0x59f0eb8 [Type: IUnknown *]
    [0x807]          : 0x59f5550 [Type: IUnknown *]
    [0x908]          : 0x12fe340 [Type: IUnknown *]
    [0xa09]          : 0x5afcb58 [Type: IUnknown *]
```

此表也可通过 GIT Cookie 以编程方式访问。

dbgcmd

```
0:000:x86> dx @$curprocess.COM.GlobalObjects[0xa09]
@$curprocess.COM.GlobalObjects[0xa09]                 : 0x5afcb58 [Type:
IUnknown *]
    [+0x00c] __abi_reference_count [Type: __abi_FTMWeakRefData]
    [+0x014] __capture        [Type: Platform::Details::__abi_CapturePtr]
```

使用 LINQ 扩展调试器对象概念

除了能够扩展进程和线程等对象外，JavaScript 还可以扩展与数据模型相关的概念。 例如，可以向每个可迭代项添加新的 LINQ 方法。 考虑一个示例扩展“DuplicateDataModel”，它将可迭代 N 次中每个条目重复。 以下代码演示如何实现此方案。

JavaScript

```
function initializeScript()
{
    var newLinqMethod =
    {
        Duplicate : function *(n)
        {
            for (var val of this)
            {
                for (var i = 0; i < n; ++i)
                {
                    yield val;
                }
            };
        }
    };

    return [new host.namedModelParent(newLinqMethod,
"DataModel.Models.Concepts.Iterable")];
}
```

使用记事本等应用程序将代码保存到DuplicateDataModel.js。

如有必要，请加载 JavaScript 脚本提供程序，然后加载DuplicateDataModel.js扩展。

dbgcmd

```
0:000:x86> !load jsprovider.dll
0:000:x86> .scriptload C:\JSExtensions\DuplicateDataModel.js
JavaScript script successfully loaded from
'C:\JSExtensions\DuplicateDataModel.js'
```

使用 dx 命令测试新的 Duplicate 函数。

dbgcmd

```
0: kd> dx -r1
Debugger.Sessions.First().Processes.First().Threads.Duplicate(2),d
Debugger.Sessions.First().Processes.First().Threads.Duplicate(2),d
: [object Generator]
    [0]              : nt!DbgBreakPointWithStatus (fffff800`9696ca60)
```

```
    [1]              : nt!DbgBreakPointWithStatus (fffff800`9696ca60)
    [2]              : intelppm!MWaitIdle+0x18 (fffff805`0e351348)
    [3]              : intelppm!MWaitIdle+0x18 (fffff805`0e351348)
…
```

另请参阅

JavaScript 扩展中的本机调试器对象 - 调试器对象详细信息

JavaScript 扩展中的本机调试器对象 - 设计和测试注意事项

JavaScript 调试器脚本

JavaScript 调试器示例脚本

JavaScript 扩展中的本机调试器对象 - 调试器对象详细信息

项目 • 2023/04/14

本主题介绍有关在 JavaScript 扩展中使用本机调试器对象的其他详细信息。

本机调试器对象表示调试器环境的各种构造和行为。 这些对象可以传递到 (，也可以在) JavaScript 扩展中获取，以操作调试器的状态。

有关调试器对象 JavaScript 扩展的信息，请参阅 JavaScript 扩展中的本机调试器对象。

有关使用 JavaScript 的一般信息，请参阅 JavaScript 调试器脚本。

例如 JavaScript 脚本和扩展，调试器团队在 https://github.com/Microsoft/WinDbg-Samples 托管 GitHub 存储库。

JavaScript 扩展中的调试器对象

传递本机对象

调试器对象可以通过多种方式传入 JavaScript 扩展或在 JavaScript 扩展中获取。

- 它们可以传递到 JavaScript 函数或方法
- 它们可以是 JavaScript 原型的实例对象，（作为可视化工具，例如）
- 可以从旨在创建本机调试器对象的主机方法返回它们
- 可以从旨在创建调试器本机对象的主机方法返回它们

传递给 JavaScript 扩展的调试器对象具有本部分中介绍的一组功能。

- 属性访问
- 投影名称
- 与本机调试器对象相关的特殊类型
- 其他属性

属性访问

虽然对象上有一些属性由 JavaScript 提供程序本身放置，但进入 JavaScript 的本机对象上的大多数属性都由数据模型提供。 这意味着，对于 object.propertyName 或 object[propertyName] ---属性访问，将发生以下情况。

- 如果 *propertyName* 是 JavaScript 提供程序本身投影到对象上的属性的名称，则它将首先解析为：否则
- 如果 *propertyName* 是数据模型 (另一个可视化工具) 投影到对象上的键的名称，则它将解析为第二个名称：否则
- 如果 *propertyName* 是本机对象的字段的名称，它将解析为第三个名称：否则
- 如果对象是指针，则将取消引用指针，并且上述循环将继续 (取消引用对象的投影属性，后跟键，后跟本机字段)

JavaScript 中的正常属性访问方式 -- object.propertyName 和 object[propertyName] -- 将访问对象的底层本机字段，就像调试器中的“dx”命令一样。

投影名称

以下属性 (和方法) 投影到进入 JavaScript 的本机对象上。

方法	签名	说明
hostContext	属性	返回一个 对象，该对象表示对象在地址空间、调试目标等 (上下文...)
targetLocation	属性	返回一个 对象，该对象是对象在地址空间中的抽象， (虚拟地址、寄存器、子寄存器等...)
targetSize	属性	有效地返回对象的大小 (： sizeof (<TYPE OF OBJECT>)
addParentModel	.addParentModel (对象)	将新的父模型 (类似于 JavaScript 原型，但在数据模型端) 对象
removeParentModel	.removeParentModel (对象)	从 对象中删除给定的父模型
runtimeTypedObject	properties	对 对象执行分析，并尝试将其转换为运行时 (派生) 类型
targetType	properties	JavaScript 扩展可直接访问基础语言的类型系统。 此访问通过类型对象的概念表示。 有关详细信息，请参阅 JavaScript 扩展中的本机调试器对象 - 类型对象

如果对象是指针，则以下属性 (和方法) 投影到进入 JavaScript 的指针上：

属性名称	签名	说明
add	.add (值)	在指针和指定值之间执行指针数学加法
address	properties	以 64 位序号对象的形式返回指针的地址， (库类型)
引用	.dereference ()	取消引用指针并返回基础对象
isNull	属性	返回指针值是否为 nullptr (0)

与本机调试器对象相关的特殊类型

位置对象

从本机对象的 targetLocation 属性返回的位置对象包含以下属性 (和方法) 。

属性名称	签名	说明
add	.add (值)	将绝对字节偏移量添加到位置。
减 (subtract)	.减 (值)	从位置减去绝对字节偏移量。

其他属性

可迭代性

任何被数据模型理解为可迭代的对象， (它是一个本机数组，或者它有一个可视化工具 (NatVis 或其他) 使其可迭代) 将具有一个迭代器函数， (通过 ES6 标准 Symbol.iterator) 为其编制索引。 这意味着可以在 JavaScript 中循环访问本机对象，如下所示。

```
JavaScript
```

```
function iterateNative(nativeObject)
{
    for (var val of nativeObject)
    {
        //
        // val will contain each element iterated from the native object.  This would be
each element of an array,
        // each element of an STL structure which is made iterable through NatVis, each
element of a data structure
        // which has a JavaScript iterator accessible via [Symbol.iterator], or each element
of something
        // which is made iterable via support of IIterableConcept in C/C++.
        //
    }
}
```

可索引性

通过序号在一个维度中被理解为可编制索引的对象 (例如：本机数组) 将通过标准属性访问运算符 -- object[index] 在 JavaScript 中编制索引。 如果对象可按名称编制索引或在多个维度中可编制索引，则 getValueAt 和 setValueAt 方法将投影到对象上，以便 JavaScript 代码可以利用索引器。

JavaScript

```
function indexNative(nativeArray)
{
    var first = nativeArray[0];
}
```

字符串转换

任何通过支持 IStringDisplayableConcept 或 NatVis DisplayString 元素进行显示字符串转换的本机对象都可以通过标准 JavaScript toString 方法访问该字符串转换。

JavaScript

```
function stringifyNative(nativeObject)
{
    var myString = nativeObject.toString();
}
```

创建本机调试器对象

如前所述，JavaScript 脚本可以通过多种方式之一将其传递到 JavaScript 中来访问本机对象，也可以通过调用主机库来创建本机对象。 使用以下函数创建本机调试器对象。

方法	签名	说明

方法	签名	说明
host.getModuleSymbol	getModuleSymbol (moduleName, symbolName, [contextInheritor]) getModuleSymbol (moduleName, symbolName, [typeName], [contextInheritor])	返回特定模块中全局符号的 对象。 模块名称和符号名称是字符串。 如果提供了可选的 *contextInheritor* 参数，则将在同一上下文中查找模块和符号， (地址空间、调试目标) 作为传递的对象。 如果未提供参数，则将在调试器的当前上下文中查找模块和符号。 不是一次性测试脚本的 JavaScript 扩展应始终提供显式上下文。 如果提供了可选的 *typeName* 参数，则将假定符号为传递的类型，并且符号 () 中指示的类型将被忽略。 请注意，任何预期对模块的公共符号进行操作的调用方都应始终提供显式类型名称。
host.getModuleContainingSymbol	getModuleContainingSymbol (location, [contextInheritor])	返回符号 (例如：包含给定地址的函数或数据) 。 请注意，仅当模块具有包含给定地址的 *专用* 符号时，此功能才起作用。 如果提供了可选的 *contextInheritor* 参数，则将在同一上下文中查找模块和符号， (地址空间、调试目标) 作为传递的对象。 如果未提供参数，则将在调试器的当前上下文中查找模块和符号。 不是一次性测试脚本的 JavaScript 扩展应始终提供显式上下文。
host.createPointerObject	createPointerObject (address, moduleName, typeName, [contextInheritor])	在指定的地址或位置创建指针对象。 模块名称和类型名称是字符串。 如果提供了可选的 *contextInheritor* 参数，则将在同一上下文中查找模块和符号， (地址空间、调试目标) 作为传递的对象。 如果未提供参数，则将在调试器的当前上下文中查找模块和符号。 不是一次性测试脚本的 JavaScript 扩展应始终提供显式上下文。

方法	签名	说明
host.createTypedObject	createTypedObject (location, moduleName， typeName, [contextInheritor])	创建一个 对象，该对象表示指定位置的调试目标的地址空间内的本机类型化对象。 模块名称和类型名称是字符串。 如果提供了可选的 *contextInheritor* 参数，则将在同一上下文中查找模块和符号，(地址空间、调试目标) 作为传递的对象。 如果未提供参数，则将在调试器的当前上下文中查找模块和符号。 不是一次性测试脚本的 JavaScript 扩展应始终提供显式上下文。

适用于 JavaScript 扩展的主机 API

JavaScript 提供程序将名为 host 的对象插入到它加载的每个脚本的全局命名空间中。 此对象提供对脚本关键功能的访问权限以及对调试器命名空间的访问。 它分为两个阶段进行设置。

- **阶段 1**：在执行任何脚本之前，主机对象仅包含脚本初始化自身和注册其扩展点所需的最少功能集，(同时作为生成者和使用者) 。 根和初始化代码不用于操作调试目标的状态或执行复杂操作，因此，直到 initializeScript 方法返回之后，主机才完全填充。
- **阶段 2**：在 initializeScript 返回后，主机对象将填充操作调试目标状态所需的一切。

主机对象级别

一些关键功能直接位于主机对象下。 其余部分为子命名空间。 命名空间包括以下内容。

命名空间	说明
诊断	有助于诊断和调试脚本代码的功能
内存	用于在调试目标中启用内存读取和写入的功能

根级别

直接在主机对象中，可以找到以下属性、方法和构造函数。

名称	签名	阶段呈现	说明
createPointerObject	createPointerObject (address, moduleName， typeName, [contextInheritor])	2	在指定的地址或位置创建指针对象。 模块名称和类型名称是字符串。 可选的 **contextInheritor** 参数与 getModuleSymbol 一样工作。

名称	签名	阶段呈现	说明
createTypedObject	createTypedObject (location，moduleName， typeName，[contextInheritor])	2	创建一个 对象，该对象表示指定位置的调试目标的地址空间内的本机类型化对象。 模块名称和类型名称是字符串。 可选的 contextInheritor 参数与 getModuleSymbol 一样工作。
currentProcess	属性	2	返回表示调试器当前进程的 对象
currentSession	属性	2	返回表示调试器当前会话的对象 (正在调试的目标、转储等...)
currentThread	属性	2	返回表示调试器当前线程的 对象
evaluateExpression	evaluateExpression (表达式，[contextInheritor])	2	这会调用调试主机，以仅使用调试目标的语言计算表达式。 如果提供了可选的 *contextInheritor* 参数，则将在上下文中计算表达式 (例如：参数的地址空间和调试目标) ;否则，将在调试器的当前上下文中评估它
evaluateExpressionInContext	evaluateExpressionInContext (context， expression)	2	这会调用调试主机，以仅使用调试目标的语言计算表达式。 context 参数指示用于计算的隐式此指针。 表达式将在上下文中计算 (例如：由 *context* 参数指示的地址空间和调试目标) 。
getModuleSymbol	getModuleSymbol (moduleName，symbolName， [contextInheritor])	2	返回特定模块中全局符号的 对象。 模块名称和符号名称是字符串。 如果提供了可选的 *contextInheritor* 参数，则将在同一上下文中查找模块和符号， (地址空间、调试目标) 作为传递的对象。 如果未提供参数，则将在调试器的当前上下文中查找模块和符号。 不是一次性脚本的 JavaScript 扩展应始终提供显式上下文
getNamedModel	getNamedModel (modelName)	2	返回针对给定名称注册的数据模型。 请注意，对尚未注册的名称调用此名称是完全合法的。 这样做将为该名称创建存根，并在注册时对实际对象执行存根操作
indexedValue	new indexedValue (值，indexies)	2	对象的构造函数，可以从 JavaScript 迭代器返回，以便将一组默认的索引分配给迭代值。 索引集必须表示为 JavaScript 数组。
Int64	new Int64 (值，[highValue])	1	这将构造一个 Int64 类型的库。 单个参数版本将采用任何可以打包到 Int64 (中且无需转换) 的值，并将其放入此类值中。 如果提供了可选的第二个参数，则第一个参数的转换将打包到较低的 32 位中，第二个参数的转换将打包到较高的 32 位中。

名称	签名	阶段呈现	说明
namedModelParent	new namedModelParent (对象，name)	1	对象的构造函数，用于放置在从 **initializeScript** 返回的数组中，这表示使用 JavaScript 原型或 ES6 类作为具有给定名称的数据模型父扩展
namedModelRegistration	new namedModelRegistration (对象，name)	1	对象的构造函数，旨在放置在从 **initializeScript** 返回的数组中，这表示通过已知名称将 JavaScript 原型或 ES6 类注册为数据模型，以便其他扩展可以查找和扩展
命名空间	属性	2	允许直接访问调试器的根命名空间。 例如，可以通过 host.namespace.Debugger.Sessions.First() 访问第一个调试目标的进程列表。使用此属性的进程
registerNamedModel	registerNamedModel (对象，modelName)	2	这会将 JavaScript 原型或 ES6 类注册为给定名称下的数据模型。 此类注册允许由其他脚本或其他调试器扩展定位和扩展原型或类。 请注意，脚本应倾向于从其 **initializeScript** 方法返回 **namedModelRegistration** 对象，而不是以命令性方式执行此操作。 任何以命令性方式进行更改的脚本都需要具有 **initializeScript** 方法才能进行清理。
registerExtensionForTypeSignature	registerExtensionForTypeSignature (对象，typeSignature)	2	这会将 JavaScript 原型或 ES6 类注册为本机类型的扩展数据模型，由所提供的类型签名提供。 请注意，脚本应倾向于从其 **initializeScript** 方法返回 **typeSignatureExtension** 对象，而不是以命令性方式执行此操作。 任何以命令性方式进行更改的脚本都需要具有 **initializeScript** 方法才能进行清理。
registerPrototypeForTypeSignature	registerPrototypeForTypeSignature (对象，typeSignature)	2	这会将 JavaScript 原型或 ES6 类注册为规范数据模型，（例如：提供的类型签名提供的本机类型的可视化工具）。 请注意，脚本应倾向于从其 **initializeScript** 方法返回 **typeSignatureRegistration** 对象，而不是以命令方式执行此操作。 任何以命令性方式进行更改的脚本都需要具有 **uninitializeScript**方法才能进行清理。
parseInt64	parseInt64 (字符串，[radix])	1	此方法与标准 JavaScript parseInt 方法类似，只不过它返回 Int64 类型库。 如果提供了基数，则分析将按指示在基 2、8、10 或 16 中发生。

名称	签名	阶段呈现	说明
typeSignatureExtension	new typeSignatureExtension (object， typeSignature，[moduleName]， [minVersion]，[maxVersion])	1	对象的构造函数，旨在放置在从 **initializeScript** 返回的数组中，这表示通过 JavaScript 原型或 ES6 类的类型签名描述的本机类型的扩展。 此类注册"添加字段"到调试器的任何匹配签名类型的可视化效果中，而不是完全接管它。 可选的模块名称和版本可以限制注册。 版本指定为"1.2.3.4"样式字符串。
typeSignatureRegistration	new typeSignatureRegistration (object， typeSignature，[moduleName]， [minVersion]，[maxVersion])	1	对象的构造函数，用于放置在从 **initializeScript** 返回的数组中，它表示针对本机类型签名的 JavaScript 原型或 ES6 类的规范注册。 此类注册"接管"调试器对与签名匹配的任何类型的可视化效果，而不仅仅是扩展签名。 可选的模块名称和版本可以限制注册。 版本指定为"1.2.3.4"样式字符串。
unregisterNamedModel	unregisterNamedModel (modelName)	2	这会通过给定名称取消注册数据模型，从而撤消 **registerNamedModel** 执行的任何操作
unregisterExtensionForTypeSignature	unregisterExtensionForTypeSignature (object， typeSignature，[moduleName]， [minVersion]，[maxVersion])	2	这会取消注册 JavaScript 原型或 ES6 类，因为该类是提供的类型签名提供的本机类型的扩展数据模型。 它是 registerExtensionForTypeSignature 的逻辑撤消。 请注意，脚本应倾向于从其 **initializeScript** 方法返回 **typeSignatureExtension** 对象，而不是以命令性方式执行此操作。 任何以命令性方式进行更改的脚本都需要具有 **initializeScript** 方法才能进行清理。 可选的模块名称和版本可以限制注册。 版本指定为"1.2.3.4"样式字符串。
unregisterPrototypeForTypeSignature	unregisterPrototypeForTypeSignature (object， typeSignature，[moduleName]， [minVersion]，[maxVersion])	2	这会取消注册 JavaScript 原型或 ES6 类作为规范数据模型 (例如：由所提供的类型签名提供的本机类型的可视化工具) 。 它是 registerPrototypeForTypeSignature 的逻辑撤消。 请注意，脚本应倾向于从其 **initializeScript** 方法返回 **typeSignatureRegistration** 对象，而不是以命令方式执行此操作。 任何以命令性方式进行更改的脚本都需要具有 **uninitializeScript** 方法才能进行清理。 可选的模块名称和版本可以限制注册。 版本指定为"1.2.3.4"样式字符串。

诊断功能

主机对象的诊断子命名空间包含以下内容。

名称	签名	阶段呈现	说明
debugLog	debugLog(对象...)	1	这为脚本扩展提供 printf 样式调试。 目前，debugLog 的输出已路由到调试器的输出控制台。 在以后的某个时间点，计划提供此输出路由的灵活性。 注意：不应将其用作将用户输出打印到控制台的方式。 将来可能不会路由到那里。

内存功能

主机对象的内存子命名空间包含以下内容。

名称	签名	阶段呈现	说明
readMemoryValues	readMemoryValues(location， numElements，[elementSize]，[isSigned]，[contextInheritor])	2	这会从调试目标的地址空间中读取值的原始数组，并将类型化数组放置在此内存视图的顶部。 提供的位置可以是地址 (64 位值) 、位置对象或本机指针。 数组的大小由 *numElements* 参数指示。 数组中每个元素的大小 (和类型) 由可选的 *elementSize* 和 *isSigned* 参数提供。 如果未提供此类参数，则默认值为无符号 (字节/1 字节) 。 如果提供了可选的 *contextInheritor* 参数，则会在上下文 (读取内存，例如：参数指示的地址空间和调试目标) ;否则，将从调试器的当前上下文中读取它。 请注意，对 8 位、16 位和 32 位值使用此方法会导致在读取内存上放置快速类型化视图。 对 64 位值使用此方法会导致构造 64 位库类型的数组，这要高得多！

名称	签名	阶段呈现	说明
readString	readString (location, [contextInheritor]) readString (location, [length], [contextInheritor])	2	这会从调试目标的地址空间中读取窄 (当前代码页) 字符串，将其转换为 UTF-16，并将结果作为 JavaScript 字符串返回。 如果无法读取内存，可能会引发异常。 提供的位置可以是地址 (64 位值) 、位置对象或本机字符。 *如果提供了可选的 contextInheritor 参数，则会在上下文 (读取内存，例如：参数指示的地址空间和调试目标) ;否则，将从调试器的当前上下文中读取它。 如果提供了可选的 length 参数，则读取字符串将采用指定的长度。*
readWideString	readWideString (location, [contextInheritor]) readWideString (location, [length], [contextInheritor])	2	这会从调试目标的地址空间读取宽 (UTF-16) 字符串，并将结果作为 JavaScript 字符串返回。 如果无法读取内存，可能会引发异常。 提供的位置可以是地址 (64 位值) 、位置对象或本机 wchar_t。 如果提供了可选的 *contextInheritor* 参数，则会在上下文 (读取内存，例如：参数指示的地址空间和调试目标) ;否则，将从调试器的当前上下文中读取它。 如果提供了可选的 *length* 参数，则读取字符串将采用指定的长度。

JavaScript 中的数据模型概念

数据模型映射

以下数据模型概念映射到 JavaScript。

概念	本机接口	JavaScript 等效项
字符串转换	IStringDisplayableConcept	standard：toString (...) {...}
可迭代性	IIterableConcept	standard： [Symbol.iterator] () {...}
可索引性	IIndexableConcept	协议：getDimensionality (...) / getValueAt (...) / setValueAt (...)
运行时类型转换	IPreferredRuntimeTypeConcept	协议：getPreferredRuntimeTypedObject (...)

字符串转换

(IStringDisplayableConcept) 字符串转换概念直接转换为标准 JavaScript **toString** 方法。 由于所有 JavaScript 对象都具有由 Object.prototype 提供的字符串转换（（如果未在) 的其他位置提供），因此返回到数据模型的每个 JavaScript 对象都可以转换为显示字符串。 重写字符串转换只需实现自己的 toString。

JavaScript

```
class myObject
{
    //
    // This method will be called whenever any native code calls
IStringDisplayableConcept::ToDisplayString(...)
    //
    toString()
    {
        return "This is my own string conversion!";
    }
}
```

可迭代性

数据模型关于对象是否可迭代的概念直接映射到对象是否可迭代的 ES6 协议。 任何具有 [Symbol.iterator] 方法的对象都被视为可迭代。 实现此类将使对象可迭代。

仅可迭代的对象可以具有如下所示的实现。

JavaScript

```
class myObject
{
    //
    // This method will be called whenever any native code calls
IIterableConcept::GetIterator
    //
    *[Symbol.iterator]()
    {
        yield "First Value";
        yield "Second Value";
        yield "Third Value";
    }
}
```

对于既可迭代又可索引的对象，必须给予特殊考虑，因为从迭代器返回的对象必须包含索引以及通过特殊返回类型的值。

可迭代和可索引

可迭代且可索引的对象需要迭代器的特殊返回值。 迭代器生成 indexedValue 的实例，而不是生成值。 索引作为第二个参数中的数组传递给 indexedValue 构造函数。 它们可以是多维的，但必须与索引器协议中返回的维数匹配。

此代码显示一个示例实现。

JavaScript

```
class myObject
{
    //
```

```
    // This method will be called whenever any native code calls
IIterableConcept::GetIterator
    //
    *[Symbol.iterator]()
    {
        //
        // Consider this a map which mapped 42->"First Value", 99->"Second Value", and 107-
>"Third Value"
        //
        yield new host.indexedValue("First Value", [42]);
        yield new host.indexedValue("Second Value", [99]);
        yield new host.indexedValue("Third Value", [107]);
    }
}
```

可索引性

与 JavaScript 不同，数据模型在属性访问和索引之间进行了非常明确的区分。 任何希望在数据模型中显示为可编制索引的 JavaScript 对象都必须实现一个协议，该协议由 getDimensionality 方法组成，该方法返回索引器的维度，以及一对可选 getValueAt 和 setValueAt 方法，这些方法在提供的索引处执行对象的读取和写入。 如果对象是只读的或只写的，可以省略 getValueAt 或 setValueAt 方法

JavaScript

```
class myObject
{
    //
    // This method will be called whenever any native code calls
IIndexableConcept::GetDimensionality or IIterableConcept::GetDefaultIndexDimensionality
    //
    getDimensionality()
    {
        //
        // Pretend we are a two dimensional array.
        //
        return 2;
    }

    //
    // This method will be called whenever any native code calls IIndexableConcept::GetAt
    //
    getValueAt(row, column)
    {
        return this.__values[row * this.__columnCount + column];
    }

    //
    // This method will be called whenever any native code calls IIndexableConcept::SetAt
    //
    setValueAt(value, row, column)
    {
        this.__values[row * this.__columnCount + column] = value;
    }
}
```

运行时类型转换

这仅适用于针对类型系统注册的 JavaScript 原型/类 (本机) 类型。 调试器通常能够执行分析 (例如Run-Time类型信息 (RTTI) /v-table 分析)，以便从用代码表示的静态类型确定对象的真实运行时类型。 针对本

机类型注册的数据模型可以通过 IPreferredRuntimeTypeConcept 的实现来替代此行为。 同样，针对本机对象注册的 JavaScript 类或原型可以通过实现由 getPreferredRuntimeTypedObject 方法组成的协议来提供自己的实现。

请注意，虽然此方法在技术上可以返回任何内容，但它返回的内容实际上不是运行时类型或派生类型，这被视为错误的形式。 这可能会导致调试器用户的严重混淆。 但是，重写此方法对于实现的 C 样式标头+对象样式等内容非常有用。

JavaScript

```
class myNativeModel
{
    //
    // This method will be called whenever the data model calls
IPreferredRuntimeTypeConcept::CastToPreferredRuntimeType
    //
    getPreferredRuntimeTypedObject()
    {
        var loc = this.targetLocation;

        //
        // Perform analysis...
        //
        var runtimeLoc = loc.Add(runtimeObjectOffset);

        return host.createTypedObject(runtimeLoc, runtimeModule, runtimeTypeName);
    }
}
```

另请参阅

JavaScript 扩展中的本机调试器对象

JavaScript 扩展中的本机调试器对象 - 设计和测试注意事项

JavaScript 调试器脚本

JavaScript 调试器示例脚本

JavaScript 扩展中的本机调试器对象 - 类型对象

项目 • 2023/04/14

本机调试器对象表示调试器环境的各种构造。 JavaScript 扩展可直接访问基础语言的类型系统。 此访问通过 *类型对象*的概念表示。 本主题介绍与类型对象关联的属性。

本机调试器对象表示调试器环境的各种构造和行为。 这些对象可以传递到 (，也可以在) JavaScript 扩展中获取，以操作调试器的状态。

有关调试器对象 JavaScript 扩展的信息，请参阅 JavaScript 扩展中的本机调试器对象。

有关使用 JavaScript 的一般信息，请参阅 JavaScript 调试器脚本。

类型对象

可以通过多种方式获取类型对象：

- 从 对象：如果脚本在 JavaScript 中具有本机对象，则可以在该对象上访问 *targetType* 属性，以获取表示本机对象的静态类型的 *类型对象* 。
- 从主机：可以调用 *host.getModuleType* API，以便返回特定模块中定义的任何类型的类型 *对象* 。

获取类型对象后，它具有以下属性：

名称	Signature	说明
name	属性	返回类型的名称。
大小	properties	以 64 位值的形式返回类型的大小。
typeKind	properties	以字符串形式返回类型的类型。 这可以是以下值之一：“udt”、“pointer”、“memberPointer”、“array”、“function”、“typedef”、“enum”或“intrinsic”。
baseType	properties	返回此类型所基于的类型的类型对象。 这不表示 C++ 继承。 对于指针类型，这是指向的对象的类型。 对于数组类型，这是数组中包含的类型。
fields	属性	返回一个 对象，该对象具有作为命名属性可访问的类型的所有命名字段。 每个属性的值都是 *一个字段对象* ，如下所述。
baseClasses	properties	返回类型的所有直接基类的数组。 数组中的每个对象都是 *一个基类对象* ，如下所述。

functionReturnType	属性	对于函数类型，这将返回一个类型对象，该对象表示函数的返回类型。
functionParameterTypes	properties	对于函数类型，这将返回类型对象的数组，表示函数的参数类型。
functionCallingConvention	properties	对于函数类型，这会将函数的调用约定作为字符串返回。 这可以是以下值之一：“unknown”、“__cdecl”、“fastcall”、“stdcall”或“thiscall”。
pointerKind	properties	对于指针类型，这将返回字符串形式的指针类型。 这可以是以下值之一：“standard”、“reference”、“rValueReference”或“cxHat”。
memberType	properties	对于作为成员指针的指针类型，这将返回表示成员类的类型对象。
isGeneric	properties	返回类型是否为泛型。 这将为模板类型返回 true。
genericArguments	properties	对于泛型类型，这将返回泛型参数数组。 此类参数可以是类型参数，也可以是常量值。
isBitField	properties	返回类型的存储是否为位字段。
bitFieldPositions	properties	对于表示位字段存储的类型，这将返回一个位字段说明类型，指示位字段的位置。

所有这些条目在阶段 2 初始化期间都存在。

Field 对象

类型中的每个字段由具有属性的字段对象描述，如下所示：

名称	Signature	**说明**
name	属性	返回字段的名称。
type	properties	返回一个类型对象，该对象表示字段的静态类型。
locationKind	properties	以字符串的形式返回字段 (存储) 的位置类型。 这可以是以下值之一：“member”、“static”、“constant”或“none”。
offset	properties	对于具有指示偏移量 (例如：“member”) 的位置类型的字段，这将返回其包含类型中字段的偏移量为 64 位值。
location	properties	对于具有指示位置 (位置的字段，例如：“static”)，这将返回字段的位置作为 *位置对象*。
值	属性	对于具有指示值（（例如：“constant”））的位置类型的字段，这将返回字段的值。

所有这些条目在阶段 2 初始化期间都存在。

基类对象

类型中的每个基类由具有属性的基类对象描述，如下所示：

名称	Signature	说明
name	属性	返回基类的名称。
offset	properties	返回此基类在其包含类型中的偏移量。
type	properties	返回一个类型对象，该对象表示基类的静态类型。

所有这些条目在阶段 2 初始化期间都存在。

代码示例

有关代码示例，请参阅 ImageInfo.js 脚本。 有关代码示例的详细信息，请参阅 JavaScript 调试器示例脚本。

JavaScript

```
// fieldType references basic types that should be present in **ANY**
symbolic information.
// Just grab the first module as the "reference module" for this purpose.
We cannot grab
// "ntdll" generically as we want to avoid a situation in which the debugger
opens a module (-z ...)
// from failing.
//
var moduleName = contextInheritorModule.__ComparisonName;
var typeObject = host.getModuleType(moduleName, field.fieldType,
contextInheritorModule);
var result = host.createTypedObject(addr, typeObject);
```

另请参阅

JavaScript 扩展中的本机调试器对象

JavaScript 扩展中的本机调试器对象 - 设计和测试注意事项

JavaScript 调试器脚本

JavaScript 调试器示例脚本

JavaScript 扩展中的本机调试器对象 - 设计和测试注意事项

项目 • 2023/04/14

本主题介绍在 JavaScript 扩展中使用本机调试器对象的设计和测试注意事项。

本机调试器对象表示调试器环境的各种构造和行为。 这些对象可以传递到 (，也可以在) JavaScript 扩展中获取，以操作调试器的状态。

有关调试器对象 JavaScript 扩展的信息，请参阅 JavaScript 扩展中的本机调试器对象。

有关使用 JavaScript 的一般信息，请参阅 JavaScript 调试器脚本。

调试器数据模型设计注意事项

设计原理

请考虑以下原则，使调试器扩展提供可发现、可查询和可编写脚本的信息。

- 信息与需要信息的位置接近。 例如，有关注册表项的信息应显示为包含注册表项句柄的局部变量的一部分。
- 信息是结构化的。 例如，有关注册表项的信息显示在单独的字段中，例如键类型、键 ACL、密钥名称和值。 这意味着无需分析文本即可访问各个字段。
- 信息是一致的。 有关注册表项句柄的信息以与文件句柄信息类似的方式呈现。

避免采用不支持这些原则的方法。

- 不要将物品结构成一个平面的“厨房水槽”。 有条理的层次结构允许用户浏览他们正在查找的信息，而无需事先知道他们正在查找的内容，并支持可发现性。
- 不要转换经典 dbgeng 扩展，只需将其移动到模型，同时仍输出原始文本屏幕。 这不能与其他扩展组合，并且不能使用 LINQ 表达式查询。 而是将数据拆分为单独的可查询字段。

命名规则

- 字段的大写应为 PascalCase。 对于其他大小写中广为人知的名称（如 jQuery），可以考虑例外。
- 避免使用通常不会在 C++ 标识符中使用的特殊字符。 例如，避免使用包含空格) 的“总长度” (或包含方括号) 的“[size]” (等名称。 此约定允许从脚本语言（不允许这些字符作为标识符的一部分）更轻松地使用，也允许从命令窗口更轻松地使用。

组织和层次结构指南

- 不要扩展调试器命名空间的顶层。 相反，应扩展调试器中的现有节点，以便信息显示在最相关的位置。
- 不要重复概念。 如果要创建一个数据模型扩展，其中列出了有关调试器中已存在的概念的其他信息，请扩展现有信息，而不是尝试将其替换为新信息。 换句话说，显示模块详细信息的扩展应扩展现有的 *Module* 对象，而不是创建新的模块列表。
- 自由浮动实用工具命令必须是 *Debugger.Utility* 命名空间的一部分。 它们还应适当地 (子命名空间，例如 *Debugger.Utility.Collections.FromListEntry*)

向后兼容性和中断性变更

已发布的脚本不应破坏与依赖它的其他脚本的兼容性。 例如，如果函数发布到模型，则它应尽可能保留在同一位置，并使用相同的参数。

不使用外部资源

- 扩展不得生成外部进程。 外部进程可能会干扰调试器的行为，并且会在各种远程调试器方案中出现错误行为， (例如 dbgsrv remotes、ntsd remotes 和“ntsd -d remotes”)
- 扩展不得显示任何用户界面。 显示用户界面元素在远程调试方案中的行为不正确，并可能会中断控制台调试方案。
- 扩展不得通过未记录的方法操作调试器引擎或调试器 UI。 这会导致兼容性问题，并且在具有不同 UI 的调试器客户端上的行为不正确。
- 扩展只能通过记录的调试器 API 访问目标信息。 对于许多远程方案，甚至某些跨安全边界的本地调试方案，尝试通过 win32 API 访问有关目标的信息会失败。

不使用 Dbgeng 特定功能

用作扩展的脚本不得尽可能依赖于特定于 dbgeng 的功能， (例如执行“经典”调试器扩展) 。 脚本应在承载数据模型的任何调试器之上可用。

测试调试器扩展

扩展应适用于各种方案。 虽然某些扩展可能特定于方案 (例如内核调试方案) ，但大多数扩展应适用于所有方案，或者具有指示支持方案的元数据。

内核模式

- 实时内核调试
- 内核转储调试

用户模式

- 实时用户模式调试
- 用户模式转储调试

此外，请考虑这些调试器使用方案

- 多进程调试
- 多会话调试 (例如，在单个会话) 转储 + 实时用户

远程调试器使用情况

使用远程调试器使用方案测试是否正确操作。

- dbgsrv remotes
- ntsd remotes
- ntsd -d remotes

有关详细信息，请参阅 使用 CDB 和 NTSD 进行调试 和 **激活进程服务器**。

回归测试

在发布新版本的调试器时，调查测试自动化的使用，该自动化可以验证扩展的功能。

另请参阅

JavaScript 扩展中的本机调试器对象

JavaScript 扩展中的本机调试器对象 - 调试器对象详细信息。

JavaScript 调试器脚本

JavaScript 调试器示例脚本

NatVis 中的本机调试器对象

项目 • 2023/10/26

概述

本机调试器对象表示调试器环境的各种构造和行为。 示例调试器对象包括以下内容。

- 会话
- 线程/线程
- 流程/流程
- 堆栈帧/堆栈帧
- 局部变量
- 模块/模块
- 实用工具
- 状态
- 设置

可以使用 dx 命令和 LINQ 与调试器对象进行交互。 有关详细信息，请参阅 dx (Display Debugger 对象模型表达式) 和 将 LINQ 与调试器对象配合使用。

还可以使用 JavaScript 处理调试器对象。 有关此内容的详细信息，请参阅 JavaScript 扩展中的本机调试器对象。

本主题介绍如何创建自定义 NatVis 可视化工具以显示调试器对象。

NatVis 开发资源

有关使用 NatVis 的一般信息，请参阅这些资源。

创建本机对象的自定义视图

.nvload

.nvlist

.nvunload

.nvunloadall

自定义 NatVis 对象示例

创建具有 CDog 类实例的简单 C++ 应用程序。

C++

```
class CDog
{
public:
   CDog(){m_age = 8; m_weight = 30;}
   long m_age;
   long m_weight;
};

int main()
{
   CDog MyDog;
   printf_s("%d, %d\n", MyDog.m_age, MyDog.m_weight);
   return 0;
}
```

创建一个名为 Dog.natvis 的文件，其中包含此 XML：

XML

```
<?xml version="1.0" encoding="utf-8"?>
<AutoVisualizer
xmlns="https://schemas.microsoft.com/vstudio/debugger/natvis/2010">
   <Type Name="CDog">
      <DisplayString>{{Age = {m_age} years. Weight = {m_weight} pounds.}}
</DisplayString>
   </Type>
</AutoVisualizer>
```

将 Dog.natvis 复制到 Windows 调试工具安装目录中的可视化工具文件夹。 例如：

C：\Program Files\Debugging Tools for Windows (x64) \Visualizers

运行程序，并在 main 函数中中断。 执行一个步骤，以便初始化变量 `MyDog` 。 使用 **??** 显示`MyDog`，再次使用 **dx**。

dbgcmd

```
0:000> ??MyDog
class CDog
   +0x000 m_age        : 0n8
   +0x004 m_weight     : 0n30
0:000> *
0:000> dx -r1 MyDog
.....
MyDog     : {Age = 8 years. Weight = 30 pounds.} [Type: CDog]
```

另请参阅

dx（显示调试器对象模型表达式）

将 LINQ 与调试器对象配合使用

JavaScript 扩展中的本机调试器对象

将 LINQ 与调试器对象配合使用

项目 • 2024/02/06

LINQ 语法可与调试器对象一起使用，以搜索和操作数据。 与使用调试器命令相比，将 LINQ 语法与 dx 命令配合使用可实现更一致的体验。 无论查看哪个调试器对象，输出和选项都是一致的。 LINQ 查询允许你提出诸如“运行线程最多的前 5 个进程是什么？”等问题。

调试器对象投影到根位于“Debugger”的命名空间中。 进程、模块、线程、堆栈、堆栈帧和局部变量都可用于 LINQ 查询。

LINQ 在概念上类似于用于查询数据库的 结构化查询语言 (SQL) 。 可以使用许多 LINQ 方法来搜索、筛选和分析调试数据。 使用 LINQ C# 方法语法。 有关 LINQ 和 LINQ C# 语法的详细信息，请参阅在 C# 中使用 LINQ 入门

调试器支持中使用的 LINQ 使用 LINQ 的“方法语法”，而不是“查询语法”。 可以在 LINQ (语言集成查询) 中找到有关差异的更多详细信息。

LINQ 命令（如下所示）可与调试器对象一起使用。 所有。任何。计数。第一。扁平化。GroupBy， 。最后。OrderBy， 。OrderByDescending， 。选择 和 。其中。 这些方法在 C# LINQ 方法窗体) 尽可能紧密地遵循 (。

本机调试器对象

本机调试器对象表示调试器环境的各种构造和行为。 示例调试器对象包括以下内容。

- 会话
- 线程/线程
- Processes / Process
- 堆栈帧/堆栈帧
- 局部变量
- 模块/模块
- 实用工具
- 状态
- 设置

还可以使用 NatVis 处理调试器对象。 有关详细信息，请参阅 NatVis 中的本机调试器对象。 有关将调试器对象与 JavaScript 配合使用的信息，请参阅 JavaScript 扩展中的本机调试器对象。 有关使用 C++ 和驱动程序对象的信息，请参阅 调试器数据模型 C++ 概述。

Dx 命令

此处显示的示例使用 dx 命令，有关使用 dx 命令的详细信息，请参阅 dx (Display Debugger 对象模型表达式) 。

开发 LINQ 查询

开发 LINQ 调试器对象查询的一种方法是使用显示的 DML 链接浏览数据模型，以首先找到将在查询中使用的调试器对象。

对于此示例，我们希望显示内核调试会话中的进程列表以及每个进程的线程数。

若要开始探索，可以使用 dx 命令显示顶级调试器对象。

dbgcmd

```
0: kd> dx Debugger
Debugger
    Sessions
    Settings
    State
    Utility
```

选择顶级主题后，我们确定会话看起来最有趣，因此选择 DML 链接以显示它包含 *进程*。

dbgcmd

```
0: kd> dx -r1 Debugger.Sessions[0]
Debugger.Sessions[0]                 : Remote KD:
KdSrv:Server=@{<Local>},Trans=@{NET:Port=50005,Key=MyKey}
    Processes
    Id               : 0
    Attributes
```

然后，我们进一步选择查看特定进程，并看到与该进程关联的 *线程* 可用。 为其中一个进程选择 *“线程”* 时，可以看到与该进程关联的所有线程都可用。

dbgcmd

```
0: kd> dx -r1 Debugger.Sessions[0].Processes[1428].Threads
Debugger.Sessions[0].Processes[1428].Threads
    [0x598]          : <Unable to get stack trace> [Switch To]
    [0x1220]         : <Unable to get stack trace> [Switch To]
    [0x6f8]          : nt!KiSwapContext+0x76 (fffff806`4466a186)  [Switch
To]
    [0x128c]         : <Unable to get stack trace> [Switch To]
```

```
    [0x27e4]          : nt!KiSwapContext+0x76 (fffff806`4466a186)  [Switch
To]
```

我们现在知道，在调试器对象模型中提供了显示与进程关联的线程数所需的数据。

若要缩短 LINQ 查询时间，可以使用本主题后面所述的 系统定义变量 来显示与当前会话关联的进程。

dbgcmd

```
0: kd> dx @$cursession.Processes
@$cursession.Processes
    [0x0]             : Idle [Switch To]
    [0x4]             : System [Switch To]
    [0x90]            : Registry [Switch To]
...
```

接下来，添加 select 语句。 首先，可以指定“名称”字段。

dbgcmd

```
0: kd> dx @$cursession.Processes.Select(p => p.Name)
@$cursession.Processes.Select(p => p.Name)
    [0x0]             : Idle
    [0x4]             : System
    [0x90]            : Registry
...
```

对于我们的方案，我们还需要线程数。 由于有两个字段，因此请使用 *new* 创建匿名类型，类似于 C# 的匿名类型语法，如下 所述用户定义变量。

dbgcmd

```
dx @$cursession.Processes.Select(p => new {Name = p.Name, Threads =
p.Threads})
```

使用此命令时，“dx”实际上不再打印出名称，因此添加 -r2 (递归两个级别) 以显示“名称”和“线程”。

dbgcmd

```
dx -r2 @$cursession.Processes.Select(p => new {Name = p.Name, Threads =
p.Threads})
@$cursession.Processes.Select(p => new {Name = p.Name, Threads = p.Threads})
    [0x0]
        Name              : Idle
        Threads
```

```
    [0x4]
        Name              : System
        Threads
    [0x90]
        Name              : Registry
        Threads
```

此时，我们将显示进程名称和线程列表。 若要显示 ThreadCount，请使用 。*Count ()* 方法。

dbgcmd

```
0: kd> dx -r2 @$cursession.Processes.Select(p => new {Name = p.Name,
ThreadCount = p.Threads.Count()})
@$cursession.Processes.Select(p => new {Name = p.Name, ThreadCount =
p.Threads.Count()})
    [0x0]
        Name              : Idle
        ThreadCount       : 0x4
    [0x4]
        Name              : System
        ThreadCount       : 0xe7
    [0x90]
        Name              : Registry
        ThreadCount       : 0x4
...
```

若要查看哪些进程具有大量线程，请使用 *OrderByDescending* 按线程计数对列表进行排序。

dbgcmd

```
0: kd> dx -r2 @$cursession.Processes.Select(p => new {Name = p.Name,
ThreadCount = p.Threads.Count()}).OrderByDescending(p => p.ThreadCount)
@$cursession.Processes.Select(p => new {Name = p.Name, ThreadCount =
p.Threads.Count()}).OrderByDescending(p => p.ThreadCount)
    [0x4]
        Name              : System
        ThreadCount       : 0xe7
    [0xa38]
        Name              : svchost.exe
        ThreadCount       : 0x45
    [0x884]
        Name              : MemCompression
        ThreadCount       : 0x3e
```

若要在带格式的网格中呈现，请将“-r2”更改为“-g”。 不需要指定递归级别，因为网格选项会相应地显示列。 最后，将“，d”格式说明符添加到输出十进制值。

```
```

dbgcmd

```
0: kd> dx -g @$cursession.Processes.Select(p => new {Name = p.Name,
ThreadCount = p.Threads.Count()}).OrderByDescending(p => p.ThreadCount),d
================================================================================
===============
=            = Name
= ThreadCount =
================================================================================
===============
= [4]        - System
- 231         =
= [2616]     - svchost.exe
- 69          =
= [2180]     - MemCompression
- 62          =
= [968]      - explorer.exe
- 61          =
```

调试器对象示例

此示例显示运行最多线程的前 5 个进程：

dbgcmd

```
0: kd> dx -r2 Debugger.Sessions.First().Processes.Select(p => new { Name =
p.Name, ThreadCount = p.Threads.Count() }).OrderByDescending(p =>
p.ThreadCount),5
Debugger.Sessions.First().Processes.Select(p => new { Name = p.Name,
ThreadCount = p.Threads.Count() }).OrderByDescending(p => p.ThreadCount),5

:
    [0x4]            :
        Name             : <Unknown Image>
        ThreadCount      : 0x73
    [0x708]          :
        Name             : explorer.exe
        ThreadCount      : 0x2d
    [0x37c]          :
        Name             : svchost.exe
        ThreadCount      : 0x2c
    [0x6b0]          :
        Name             : MsMpEng.exe
        ThreadCount      : 0x22
    [0x57c]          :
        Name             : svchost.exe
        ThreadCount      : 0x15
    [...]
```

此示例演示即插即用设备树中的设备，这些设备按物理设备对象的驱动程序的名称分组。 并非所有输出都显示。

dbgcmd

```
kd> dx -r2 Debugger.Sessions.First().Devices.DeviceTree.Flatten(n =>
n.Children).GroupBy(n => n.PhysicalDeviceObject->Driver-
>DriverName.ToDisplayString())
Debugger.Sessions.First().Devices.DeviceTree.Flatten(n =>
n.Children).GroupBy(n => n.PhysicalDeviceObject->Driver-
>DriverName.ToDisplayString())

:
    ["\"\\Driver\\PnpManager\""] :
        [0x0]            : HTREE\ROOT\0
        [0x1]            : ROOT\volmgr\0000 (volmgr)
        [0x2]            : ROOT\BasicDisplay\0000 (BasicDisplay)
        [0x3]            : ROOT\CompositeBus\0000 (CompositeBus)
        [0x4]            : ROOT\vdrvroot\0000 (vdrvroot)
         ...
```

Dx 命令选项卡自动完成

上下文 TAB 键自动完成可识别 LINQ 查询方法，并且适用于 lambda 的参数。

例如，键入 (或将以下文本复制并粘贴到调试器) 。 然后多次点击 TAB 键以循环完成可能的完成。

dbgcmd

```
dx -r2 Debugger.Sessions.First().Processes.Select(p => new {Name = p.Name,
ThreadCount = p.Threads.Count() }).OrderByDescending(p => p.
```

按 TAB 键，直到“。将显示“名称”。 添加右括号“) ”，然后按 Enter 执行命令。

dbgcmd

```
kd> dx -r2 Debugger.Sessions.First().Processes.Select(p => new {Name =
p.Name, ThreadCount = p.Threads.Count() }).OrderByDescending(p => p.Name)
Debugger.Sessions.First().Processes.Select(p => new {Name = p.Name,
ThreadCount = p.Threads.Count() }).OrderByDescending(p => p.Name) :
    [0x274]          :
        Name             : winlogon.exe
        ThreadCount      : 0x4
    [0x204]          :
        Name             : wininit.exe
        ThreadCount      : 0x2
    [0x6c4]          :
```

```
        Name              : taskhostex.exe
        ThreadCount       : 0x8
         ...
```

此示例演示如何使用键比较器方法完成。 替换将显示字符串方法，因为键是字符串。

dbgcmd

```
dx -r2 Debugger.Sessions.First().Processes.Select(p => new {Name = p.Name,
ThreadCount = p.Threads.Count() }).OrderByDescending(p => p.Name, (a, b) =>
a.
```

按 TAB 键，直到“。显示“长度”。 添加右括号“) ”，然后按 Enter 执行命令。

dbgcmd

```
kd> dx -r2 Debugger.Sessions.First().Processes.Select(p => new {Name =
p.Name, ThreadCount = p.Threads.Count() }).OrderByDescending(p => p.Name,
(a, b) => a.Length)
Debugger.Sessions.First().Processes.Select(p => new {Name = p.Name,
ThreadCount = p.Threads.Count() }).OrderByDescending(p => p.Name, (a, b) =>
a.Length) :
    [0x544]          :
        Name             : spoolsv.exe
        ThreadCount      : 0xc
    [0x4d4]          :
        Name             : svchost.exe
        ThreadCount      : 0xa
    [0x438]          :
        Name             : svchost.exe
```

用户定义的变量

可以通过为变量名称添加 @$前缀来定义用户定义的变量。 用户定义的变量可以分配给 dx 可以利用的任何内容，例如 lambda、LINQ 查询的结果等。

可以创建和设置用户变量的值，如下所示。

dbgcmd

```
kd> dx @$String1="Test String"
```

可以使用 *Debugger.State.UserVariables* 或 *@$vars* 显示定义的用户变量。

dbgcmd

```
kd> dx Debugger.State.UserVariables
Debugger.State.UserVariables :
    mySessionVar     :
    String1          : Test String
```

可以使用 删除变量。删除。

dbgcmd

```
kd> dx @$vars.Remove("String1")
```

此示例演示如何定义用户变量以引用 Debugger.Sesssions。

dbgcmd

```
kd> dx @$mySessionVar = Debugger.Sessions
```

然后，可以使用用户定义的变量，如下所示。

dbgcmd

```
kd> dx -r2 @$mySessionVar
@$mySessionVar   :
    [0x0]            : Remote KD:
KdSrv:Server=@{<Local>},Trans=@{COM:Port=\\.\com3,Baud=115200,Timeout=4000}
        Processes        :
        Devices
```

系统定义的变量

可在任何 LINQ dx 查询中使用以下系统定义的变量。

- @$cursession - 当前会话
- @$curprocess - 当前进程
- @$curthread - 当前线程

此示例演示如何使用系统定义的变量。

dbgcmd

```
kd> dx @$curprocess.Threads.Count()
@$curprocess.Threads.Count() : 0x4
```

dbgcmd

```
kd> dx -r1 @$curprocess.Threads
@$curprocess.Threads :
    [0x4adc]         :
    [0x1ee8]         :
    [0x51c8]         :
    [0x62d8]         :
     ...
```

用户定义的变量 - 匿名类型

此动态对象的创建是使用 C# 匿名类型语法 (新的 { ... } 完成的) 。 有关匿名类型的详细信息，请参阅 匿名类型 (C# 编程指南) 。 此示例创建一个具有整数和字符串值的匿名类型。

dbgcmd

```
kd> dx -r1 new { MyInt = 42, MyString = "Hello World" }
new { MyInt = 42, MyString = "Hello World" } :
    MyInt            : 42
    MyString         : Hello World
```

Lambda 表达式) (函数对象

用于查询数据的许多方法都基于跨集合中的对象重复运行用户提供的函数的概念。 为了支持在调试器中查询和操作数据的功能，dx 命令支持使用等效 C# 语法的 lambda 表达式。 lambda 表达式由使用 => 运算符定义，如下所示：

(参数) => (结果)

若要了解如何将 LINQ 与 dx 配合使用，请尝试此简单示例将 5 和 7 相加。

dbgcmd

```
kd> dx ((x, y) => (x + y))(5, 7)
```

dx 命令回显 lambda 表达式并显示 12 的结果。

dbgcmd

```
((x, y) => (x + y))(5, 7)  : 12
```

此示例 lambda 表达式组合了字符串“Hello”和“World”。

```
dbgcmd
```

```
kd> dx ((x, y) => (x + y))("Hello", "World")
((x, y) => (x + y))("Hello", "World") : HelloWorld
```

支持的 LINQ 语法 - 查询方法

dx 定义为可迭代的任何对象 (是本机数组、NatVis 编写的类型（将它描述为容器）或调试器扩展对象) 具有一系列 LINQ (或 LINQ 等效) 方法投影到该对象上。 下面介绍了这些查询方法。 查询方法的参数签名在所有查询方法之后列出。

筛选方法

。其中，（PredicateMethod）： 返回一个新的 对象集合，其中包含谓词方法为其返回 true 的输入集合中的每个对象。

投影方法

。平展（[KeyProjectorMethod]）： 将容器的输入容器 (树)，并将其平展到树中具有每个元素的单个容器中。 如果提供了可选的密钥投影仪方法，则树被视为键的容器，这些密钥本身是容器，这些键由对投影方法的调用确定。

。选择“（KeyProjectorMethod）： 返回对象的新集合，其中包含对输入集合中的每个对象调用投影仪方法的结果。

分组方法

。GroupBy（KeyProjectorMethod，[KeyComparatorMethod]）： 通过将输入集合中的所有对象分组，返回新的集合集合，这些对象具有通过调用键投影仪方法确定的相同键。 可以提供可选的比较器方法。

联接 (InnerCollection、外部键选择器方法、内部键选择器方法、结果选择器方法、[ComparatorMethod])： 基于键选择器函数联接两个序列并提取值对。 还可以指定可选的比较器方法。

intersect (InnerCollection， [ComparatorMethod])： 返回集的交集，这意味着出现在两个集合中的每个元素。 还可以指定可选的比较器方法。

Union (InnerCollection， [ComparatorMethod])： 返回集联合，这意味着出现在两个集合中的任何一个中的唯一元素。 还可以指定可选的比较器方法。

数据集方法

包含 (Object， [ComparatorMethod])： 确定序列是否包含指定的元素。 可以提供一个可选的比较器方法，每次将元素与序列中的条目进行比较时都会调用该方法。

Distinct ([ComparatorMethod])： 从集合中删除重复值。 每次必须比较集合中的对象时，可以提供一个可选的比较器方法来调用。

除 (InnerCollection **外**，[ComparatorMethod])： 返回集差，这意味着一个集合中未出现在另一个集合中的元素。 可以指定可选的比较器方法。

Concat (InnerCollection)： 连接两个序列以形成一个序列。

排序方法

。OrderBy (KeyProjectorMethod，[KeyComparatorMethod])： 通过调用输入集合中每个对象的键投影方法，根据键按升序对集合进行排序。 可以提供可选的比较器方法。

。OrderByDescending (KeyProjectorMethod， [KeyComparatorMethod])： 根据通过调用输入集合中每个对象的键投影方法提供的键，按降序对集合进行排序。 可以提供可选的比较器方法。

聚合方法

Count ()： 返回集合中元素数的方法。

Sum ([ProjectionMethod])： 计算集合中值的总和。 可以选择指定投影仪方法，以在求和发生之前转换元素。

Skip 方法

跳过 (**计数**)： 跳过序列中指定位置的元素。

SkipWhile (PredicateMethod)： 跳过基于谓词函数的元素，直到元素不满足条件。

采用方法

采取 (**计数**)： 将元素置于序列中的指定位置。

TakeWhile (PredicateMethod)： 根据谓词函数获取元素，直到元素不满足条件。

比较方法

SequenceEqual (InnerCollection， [ComparatorMethod])： 通过以成对方式比较元素来确定两个序列是否相等。 可以指定可选的比较器。

错误处理方法

AllNonError (PredicateMethod)： 返回集合的所有非错误元素是否满足给定条件。

FirstNonError ([PredicateMethod])： 返回集合中不是错误的第一个元素。

LastNonError ([PredicateMethod])： 返回集合中不是错误的最后一个元素。

其他方法

。所有（PredicateMethod）： 返回对输入集合中每个元素调用指定谓词方法的结果是否为 true。

。任何（PredicateMethod）： 返回对输入集合中任何元素调用指定谓词方法的结果是否为 true。

。First（[PredicateMethod]）： 返回集合中的第一个元素。 如果传递了可选谓词，则返回集合中对谓词的调用返回 true 的第一个元素。

。上一（[PredicateMethod]）： 返回集合中的最后一个元素。 如果传递了可选的谓词，则返回集合中对谓词的调用返回 true 的最后一个元素。

Min ([KeyProjectorMethod])： 返回集合的最小元素。 可以指定一个可选的投影仪方法，以投影每个方法，然后再将其与其他方法进行比较。

Max ([KeyProjectorMethod])： 返回集合的最大元素。 可以指定一个可选的投影仪方法，以投影每个方法，然后再将其与其他方法进行比较。

Single ([PredicateMethod])： 如果集合包含多个元素)，则返回列表中的唯一元素 (或错误。 如果指定了谓词，则返回满足谓词 (如果多个元素满足谓词，则函数返回错误) 。

参数的签名

展开表

KeyProjectorMethod：（obj => 任意键）	获取集合的 一个 对象，并从该对象返回一个键。
KeyComparatorMethod：（(a，b) => 整数值）	获取两个键并比较它们返回的返回情况： 如果（< b），则为 -1 如果（a == b)，则为 0 如果（b）>，则为 1
PredicateMethod：（obj => 布尔值）	获取集合的 对象，并根据该对象是否满足特定条件返回 true 或 false。

支持的 LINQ 语法 - 字符串操作

所有字符串对象都具有以下投影方法，以便它们可供使用：

查询相关方法 & 属性

。**包含（OtherString）**：返回一个布尔值，该值指示输入字符串是否包含 OtherString。

。**EndsWith（OtherString）**：返回一个布尔值，该值指示输入字符串是否以 OtherString 结尾。

Length：返回字符串长度的属性。

。**StartsWith（OtherString）**：返回一个布尔值，该值指示输入字符串是否以 OtherString 开头。

。**子字符串（StartPos，[Length]）**：返回输入字符串中从给定起始位置开始的子字符串。 如果提供了可选长度，则返回的子字符串将采用指定的长度;否则， 它将转到字符串的末尾。

其他方法

。**IndexOf（OtherString）**：返回输入字符串中 OtherString 的第一个匹配项的索引。

。**LastIndexOf（OtherString）**：返回输入字符串中 OtherString 最后一个匹配项的索引。

格式设置方法

。**PadLeft（TotalWidth）**：根据需要向字符串左侧添加空格，以使字符串的总长度达到指定的宽度。

。**PadRight（TotalWidth）**：根据需要向字符串右侧添加空格，以使字符串的总长度达到指定的宽度。

。**删除（StartPos，[Length]）**：从输入字符串中删除字符，从指定起始位置开始。 如果提供了可选的 length 参数，则将删除该数量的字符;否则， 将删除字符串末尾的所有字符。

。**替换（SearchString、ReplaceString）**：将输入字符串中出现的 SearchString 替换为指定的 ReplaceString。

字符串对象投影

除了直接投影到字符串对象的方法外，任何本身具有字符串转换的对象都有以下投影方法，使其方法可供使用：

。**ToDisplayString（）**：返回 对象的字符串转换。 这是字符串转换，将在对象的 dx 调用中显示。 可以提供格式说明符来设置 ToDisplayString 输出的格式。 有关详细信息，请参阅 Visual Studio 调试器中 C++ 的格式说明符

以下示例演示了格式说明符的使用。

dbgcmd

```
kd> dx (10).ToDisplayString("d")
(10).ToDisplayString("d") : 10

kd> dx (10).ToDisplayString("x")
(10).ToDisplayString("x") : 0xa

kd> dx (10).ToDisplayString("o")
(10).ToDisplayString("o") : 012

kd> dx (10).ToDisplayString("b")
(10).ToDisplayString("b")  : 0y1010

kd> dx ("some wchar string here").ToDisplayString("su")
("some wchar string here").ToDisplayString("su")  : "some wchar string here"

kd> dx ("some wchar string here").ToDisplayString("sub")
("some wchar string here").ToDisplayString("sub")  : some wchar string here
```

调试即插即用示例

本部分说明如何使用与 LINQ 查询一起使用的内置调试器对象来调试即插即用对象。

查看所有设备

使用设备树上的 *Flatten* 查看所有设备。

dbgcmd

```
 1: kd> dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children)
@$cursession.Devices.DeviceTree.Flatten(n => n.Children)
    [0x0]            : HTREE\ROOT\0
    [0x1]            : ROOT\volmgr\0000 (volmgr)
    [0x2]            : ROOT\BasicDisplay\0000 (BasicDisplay)
    [0x3]            : ROOT\CompositeBus\0000 (CompositeBus)
    [0x4]            : ROOT\vdrvroot\0000 (vdrvroot)
    [0x5]            : ROOT\spaceport\0000 (spaceport)
    [0x6]            : ROOT\KDNIC\0000 (kdnic)
    [0x7]            : ROOT\UMBUS\0000 (umbus)
    [0x8]            : ROOT\ACPI_HAL\0000
...
```

网格显示

与其他 dx 命令一样，可以选择并按住 (或右键单击) 执行命令后，选择“显示为网格”或向命令添加“-g”以获取结果的网格视图。

dbgcmd

```
# 0: kd> dx -g @$cursession.Devices.DeviceTree.Flatten(n => n.Children)
================================================================================
================================================================================
================================================================================
================================================================================
=====
# =                                                                = (+)
DeviceNodeObject = InstancePath
= ServiceName               = (+) PhysicalDeviceObject
= State                           = (+) Resoures = (+) Children       =
================================================================================
================================================================================
================================================================================
================================================================================
=====
= [0x0] : HTREE\ROOT\0                                              - {...}
- HTREE\ROOT\0                                                      -
- 0xffffb6075614be40 : Device for "\Driver\PnpManager"        -
DeviceNodeStarted (776)        - {...}        - [object Object]    =
= [0x1] : ROOT\volmgr\0000 (volmgr)                                 - {...}
- ROOT\volmgr\0000                                                  - volmgr
- 0xffffb607561fbe40 : Device for "\Driver\PnpManager"        -
DeviceNodeStarted (776)        - {...}        - [object Object]    =
= [0x2] : ROOT\BasicDisplay\0000 (BasicDisplay)                     - {...}
- ROOT\BasicDisplay\0000                                            -
BasicDisplay              - 0xffffb607560739b0 : Device for
"\Driver\PnpManager"        - DeviceNodeStarted (776)        - {...}
- [object Object]    =
= [0x3] : ROOT\CompositeBus\0000 (CompositeBus)                     - {...}
- ROOT\CompositeBus\0000                                            -
CompositeBus              - 0xffffb607561f9060 : Device for
"\Driver\PnpManager"        - DeviceNodeStarted (776)        - {...}
- [object Object]    =
...
```

按状态查看设备

使用 *Where* 指定特定设备状态。

dbgcmd

```
dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n =>
n.State <operator> <state number>)
```

例如，若要查看处于 DeviceNodeStarted 状态的设备，请使用此命令。

dbgcmd

```
1: kd>  dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n
=> n.State == 776)
@$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.State
== 776)
    [0x0]            : HTREE\ROOT\0
    [0x1]            : ROOT\volmgr\0000 (volmgr)
    [0x2]            : ROOT\BasicDisplay\0000 (BasicDisplay)
    [0x3]            : ROOT\CompositeBus\0000 (CompositeBus)
    [0x4]            : ROOT\vdrvroot\0000 (vdrvroot)
...
```

查看未启动的设备

使用此命令查看未处于 DeviceNodeStarted 状态的设备。

dbgcmd

```
1: kd>  dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n
=> n.State != 776)
@$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n => n.State
!= 776)
    [0x0]            : ACPI\PNP0C01\1
    [0x1]            : ACPI\PNP0000\4&215d0f95&0
    [0x2]            : ACPI\PNP0200\4&215d0f95&0
    [0x3]            : ACPI\PNP0100\4&215d0f95&0
    [0x4]            : ACPI\PNP0800\4&215d0f95&0
    [0x5]            : ACPI\PNP0C04\4&215d0f95&0
    [0x6]            : ACPI\PNP0700\4&215d0f95&0 (fdc)
    [0x7]            : ACPI\PNP0C02\1
    [0x8]            : ACPI\PNP0C02\2
```

按问题代码查看设备

使用 *DeviceNodeObject.Problem* 对象可以查看具有特定问题代码的设备。

dbgcmd

```
dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n =>
n.DeviceNodeObject.Problem <operator> <problemCode>)
```

例如，若要查看具有非零问题代码的设备，请使用此命令。 这提供了与“**！devnode** 0 21”类似的信息。

dbgcmd

```
1: kd> dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n
=> n.DeviceNodeObject.Problem != 0)
```

```
@$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n =>
n.DeviceNodeObject.Problem != 0)
    [0x0]            : HTREE\ROOT\0
    [0x1]            : ACPI\PNP0700\4&215d0f95&0 (fdc)
```

查看所有设备（无问题）

使用此命令查看所有设备，不会出现问题

dbgcmd

```
1: kd> dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n
=> n.DeviceNodeObject.Problem == 0)
@$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n =>
n.DeviceNodeObject.Problem == 0)
    [0x0]            : ROOT\volmgr\0000 (volmgr)
    [0x1]            : ROOT\BasicDisplay\0000 (BasicDisplay)
    [0x2]            : ROOT\CompositeBus\0000 (CompositeBus)
    [0x3]            : ROOT\vdrvroot\0000 (vdrvroot)
...
```

查看具有特定问题的所有设备

使用此命令查看问题状态为0x16的设备。

dbgcmd

```
1: kd> dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n
=> n.DeviceNodeObject.Problem == 0x16)
@$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n =>
n.DeviceNodeObject.Problem == 0x16)
    [0x0]            : HTREE\ROOT\0
    [0x1]            : ACPI\PNP0700\4&215d0f95&0 (fdc)
```

按函数驱动程序查看设备

使用此命令按函数驱动程序查看设备。

dbgcmd

```
dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n =>
n.ServiceName <operator> <service name>)
```

若要查看使用特定函数驱动程序（如 atapi）的设备，请使用此命令。

dbgcmd

```
1: kd> dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n
=> n.ServiceName == "atapi")
@$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n =>
n.ServiceName == "atapi")
    [0x0]            : PCIIDE\IDEChannel\4&10bf2f88&0&0 (atapi)
    [0x1]            : PCIIDE\IDEChannel\4&10bf2f88&0&1 (atapi)
```

查看启动启动驱动程序的列表

若要查看作为启动启动驱动程序加载的 winload 的列表，需要位于你有权访问 LoaderBlock 且足够早的 LoaderBlock 仍然存在的上下文中。 例如，在 nt!IopInitializeBootDrivers。 可以将断点设置为在此上下文中停止。

dbgcmd

```
1: kd> g
Breakpoint 0 hit
nt!IopInitializeBootDrivers:
8225c634 8bff            mov     edi,edi
```

查询半结构化 命令以显示启动驱动程序结构。

dbgcmd

```
1: kd> ?? LoaderBlock->BootDriverListHead
struct _LIST_ENTRY
 [ 0x808c9960 - 0x808c8728 ]
   +0x000 Flink            : 0x808c9960 _LIST_ENTRY [ 0x808c93e8 -
0x808a2e18 ]
   +0x004 Blink            : 0x808c8728 _LIST_ENTRY [ 0x808a2e18 -
0x808c8de0 ]
```

使用 Debugger.Utility.Collections.FromListEntry 调试器对象查看数据，使用 nt!_LIST_ENTRY 结构的起始地址。

dbgcmd

```
1: kd> dx Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY
*)0x808c9960, "nt!_BOOT_DRIVER_LIST_ENTRY", "Link")
Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY *)0x808c9960,
"nt!_BOOT_DRIVER_LIST_ENTRY", "Link")
    [0x0]            [Type: _BOOT_DRIVER_LIST_ENTRY]
    [0x1]            [Type: _BOOT_DRIVER_LIST_ENTRY]
    [0x2]            [Type: _BOOT_DRIVER_LIST_ENTRY]
    [0x3]            [Type: _BOOT_DRIVER_LIST_ENTRY]
    [0x4]            [Type: _BOOT_DRIVER_LIST_ENTRY]
```

```
    [0x5]            [Type: _BOOT_DRIVER_LIST_ENTRY]
...
```

使用 -g 选项创建数据的网格视图。

dbgcmd

```
dx -r1 -g Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY
*)0x808c9960, "nt!_BOOT_DRIVER_LIST_ENTRY", "Link")
```

按功能查看设备

使用 DeviceNodeObject.CapabilityFlags 对象按功能查看设备。

dbgcmd

```
dx -r1 @$cursession.Devices.DeviceTree.Flatten(n => n.Children).Where(n =>
(n.DeviceNodeObject.CapabilityFlags & <flag>) != 0)
```

下表总结了 dx 命令与常见设备功能标志的用法。

展开表

可移动	`dbgcmd 0: kd> dx -r1 @$cursession。Devices.DeviceTree.Flatten (n => n.Children) 。其中 (n => (n.DeviceNodeObject.CapabilityFlags & 0x10) ! = 0) @$cursession。Devices.DeviceTree.Flatten (n => n.Children) 。其中 (n => (n.DeviceNodeObject.CapabilityFlags & 0x10) ! = 0) [0x0] : SWD\PRINTENUM{2F8DBBB6-F246-4D84-BB1D-AA8761353885} [0x1] : SWD\PRINTENUM{F210BC77-55A1-4FCA-AA80-013E2B408378} [0x2 F210BC77 : SWD\PRINTENUM{07940A8E-11F4-46C3-B714-7FF9B87738F8} [0x3] : DISPLAY\Default_Monitor\6&1a097cd8&0&UID5527112 (monitor)`
UniqueID	`dbgcmd 0: kd> dx -r1 @$cursession。Devices.DeviceTree.Flatten (n => n.Children) 。其中 (n => (n.DeviceNodeObject.CapabilityFlags & 0x40) ! = 0) @$cursession。Devices.DeviceTree.Flatten (n => n.Children) 。其中 (n => (n.DeviceNodeObject.CapabilityFlags & 0x40) ! = 0) [0x0] : HTREE\ROOT\0 [0x1] : ROOT\volmgr\0000`

	`(volmgr) [0x2] : ROOT\spaceport\0000 (spaceport) ...`
SilentInstall	`dbgcmd 0: kd> dx -r1 @$cursession。 Devices.DeviceTree.Flatten (n => n.Children) 。 其中 (n => (n.DeviceNodeObject.CapabilityFlags & 0x80) ! = 0) @$cursession。 Devices.DeviceTree.Flatten (n => n.Children) 。 其中 (n => (n.DeviceNodeObject.CapabilityFlags & 0x80) ! = 0) [0x0] : HTREE\ROOT\0 [0x1] : ROOT\volmgr\0000 (volmgr) [0x2] : ROOT\spaceport\0000 (spaceport) ...`
RawDeviceOk	`dbgcmd 0: kd> dx -r1 @$cursession。 Devices.DeviceTree.Flatten (n => n.Children) 。 其中 (n => (n.DeviceNodeObject.CapabilityFlags & 0x100) ! = 0) @$cursession。 Devices.DeviceTree.Flatten (n => n.Children) 。 其中 (n => (n.DeviceNodeObject.CapabilityFlags & 0x100) ! = 0) [0x0] : HTREE\ROOT\0 [0x1] : SWD\MMDEVAPI\MicrosoftGSWavetableSynth [0x2] : SWD\IP_TUNNEL_VBUS\IP_TUNNEL_DEVICE_ROOT ...`
SurpriseRemovalOK	`dbgcmd 0: kd> dx -r1 @$cursession。 Devices.DeviceTree.Flatten (n => n.Children) 。 其中 (n => (n.DeviceNodeObject.CapabilityFlags & 0x200) ! = 0) @$cursession。 Devices.DeviceTree.Flatten (n => n.Children) 。 其中 (n => (n.DeviceNodeObject.CapabilityFlags & 0x200) ! = 0) [0x0] : SWD\MMDEVAPI\MicrosoftGSWavetableSynth [0x1] : SWD\IP_TUNNEL_VBUS\IP_TUNNEL_DEVICE_ROOT [0x2] : SWD\PRINTENUM\PrintQueues ...`

有关 CapabilityFlags 的详细信息，请参阅 **DEVICE_CAPABILITIES**。

另请参阅

dx（显示调试器对象模型表达式）

NatVis 中的本机调试器对象

JavaScript 扩展中的本机调试器对象

调试器数据模型函数别名

项目 • 2023/10/26

函数别名是一个唯一的短名称，调试器用户可以通过它访问调试器扩展中定义的功能 (无论是用 C++ 还是某些脚本环境（如 JavaScript) 编写）。 此短名称与实现 IModelMethod) (数据模型函数对象相关联。 该函数采用任意数量的参数并返回单个值。 调用函数别名的效果和使用该值执行的操作取决于调用函数别名的方式以及调用该别名的主机调试器。

本主题假定读者熟悉调试器对象模型和 JavaScript。 有关将调试器对象与 JavaScript 配合使用的信息，请参阅 JavaScript 扩展中的本机调试器对象。

此处所示的一些示例使用 dx 命令，有关使用 dx 命令的详细信息，请参阅 dx (Display Debugger 对象模型表达式) 。 此外，还使用 LINQ，如 将 LINQ 与调试器对象配合使用 中所述。

使用函数别名作为扩展命令

可以调用 WinDbg 中创建的所有函数别名，就像它们是调试扩展 **一样！** “砰”命令。 如果函数不采用任何参数，则只需调用 ！ aliasName 将导致调用函数并显示结果值。 示例 (使用 JavaScript 扩展性)

例如，此函数提供两个常量值 ， *pi* 和 *e*。

C++

```
function __constants()
{
    return { math_constants : { pi :  3.1415926535 , e :  2.7182818284},
description : "Archimedes' pi and Euler's number e" };
}

function initializeScript()
{
    return [new host.functionAlias(__constants, "constants")];
}
```

此处显示了调用函数别名的结果。

dbgcmd

```
0:000> !constants
@$constants()                 : [object Object]
```

```
    math_constants    : [object Object]
    description       : Archimedes' pi and Euler's number e
```

将自动生成指向复杂对象的 DML 链接。 单击上面显示的math_constants将生成以下输出。

dbgcmd

```
0:000> dx -r1 @$constants().math_constants
@$constants().math_constants                 : [object Object]
    pi               : 3.141593
    e                : 2.718282
```

如果函数具有参数，则可以在函数别名命令本身之后提供参数。 请注意，函数别名命令之后出现的任何操作都被视为表达式，并按此类方式计算。 文本字符串不会直接传递给函数。 对于单个参数表达式，它可以位于函数别名命令本身之后。 对于多个参数，应将其括号括起来，就像它是函数调用一样，如下例所示。

C++

```
function __oneArgument(x)
{
    return -x;
}

function __twoArguments(x, y)
{
    return x + y;
}

function initializeScript()
{
    return [new host.functionAlias(__oneArgument, "neg"),
            new host.functionAlias(__twoArguments, "add")];
}
```

可按此处所示调用这两个函数。

dbgcmd

```
0:000> !neg 42
@$neg(42)        : -42

0:000> !add (5, 7)
@$add(5, 7)      : 0xc
```

函数别名与 dx 表达式计算器配合使用

除了调试扩展！ 用于调用别名函数的“bang”命令语法，当前缀 @$ 为 时，与函数别名关联的所有名称都直接在 dx 表达式计算器中可用，如下所示。

dbgcmd

```
0:000> dx @$neg(42)
@$neg(42)        : -42

0:000> dx @$add(99, 77)
@$add(99, 77)    : 0xb0
```

函数别名设计注意事项

函数别名绝不应是公开绝大多数数据模型扩展中功能的唯一方式。 数据模型扩展 (C++ 或 JavaScript) 几乎应始终包含它公开的与类型或其他调试器概念关联的数据。 与进程关联的内容应位于 Debugger.Models.Process 或该对象的子命名空间下。 使用函数别名可以方便地 (或转换可能需要更长查询的) 数据。

作为内核模式示例，以下查询采用 PnP 设备树并将其平展为简单的设备平面列表：

dbgcmd

```
0: kd> dx @$cursession.Devices.DeviceTree.Flatten(n => n.Children),5
@$cursession.Devices.DeviceTree.Flatten(n => n.Children),5
    [0x0]            : HTREE\ROOT\0
    [0x1]            : ROOT\volmgr\0000 (volmgr)
    [0x2]            : ROOT\BasicDisplay\0000 (BasicDisplay)
    [0x3]            : ROOT\CompositeBus\0000 (CompositeBus)
    [0x4]            : ROOT\vdrvroot\0000 (vdrvroot)
    [...]
```

此 JavaScript 代码演示如何将其作为函数别名实现。

dbgcmd

```
function __flatDevices()
{
    return host.currentSession.Devices.DeviceTree.Flatten(n => n.Children);
}

function initializeScript()
{
    return [new host.functionAlias(__flatDevices, "devices")];
}
```

然后，可以将函数别名作为调试扩展命令调用。

dbgcmd

```
0: kd> !devices
@$devices()
    [0x0]            : HTREE\ROOT\0
    [0x1]            : ROOT\volmgr\0000 (volmgr)
    [0x2]            : ROOT\BasicDisplay\0000 (BasicDisplay)
    [0x3]            : ROOT\CompositeBus\0000 (CompositeBus)
    [0x4]            : ROOT\vdrvroot\0000 (vdrvroot)
    [0x5]            : ROOT\spaceport\0000 (spaceport)

    ...
```

使用函数别名的优点之一是可以使用 dx 语法进一步优化它。 在此示例中，添加了 where 子句来查找包含“Harddisk”的设备节点。

dbgcmd

```
0: kd> dx @$devices().Where(n => n.InstancePath.Contains("Harddisk"))
@$devices().Where(n => n.InstancePath.Contains("Harddisk"))
    [0x0]            : STORAGE\VolumeSnapshot\HarddiskVolumeSnapshot1
    [0x1]            : STORAGE\VolumeSnapshot\HarddiskVolumeSnapshot2
    [0x2]            : STORAGE\VolumeSnapshot\HarddiskVolumeSnapshot3
    [0x3]            : STORAGE\VolumeSnapshot\HarddiskVolumeSnapshot4
    [0x4]            : STORAGE\VolumeSnapshot\HarddiskVolumeSnapshot5
    [0x5]            : STORAGE\VolumeSnapshot\HarddiskVolumeSnapshot6
```

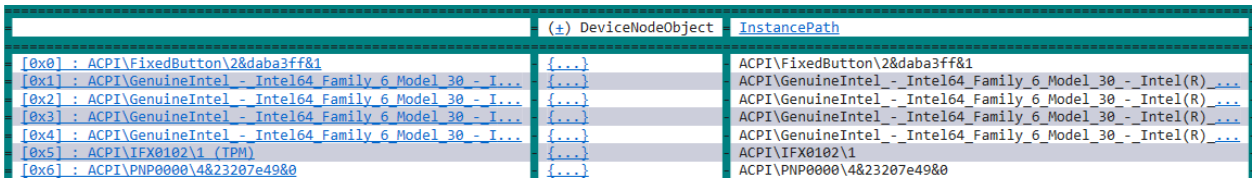
LINQ 命令（如以下）可与功能别名一起使用 - 。所有。任何。计数。第一。扁平 化。GroupBy， 。最后。OrderBy， 。OrderByDescending， 。选择 和 。其中。 这些方法在 C# LINQ 方法窗体) 尽可能紧密地遵循 (。 有关详细信息，请参阅 将 LINQ 与调试器对象配合使用。

网格显示

与其他 dx 命令一样，可以在执行命令后右键单击该命令，然后单击“显示为网格”或向命令添加“-g”以获取结果的网格视图。 然后，可以单击任意列进行排序，例如在 InstancePath 上。

dbgcmd

```
0: kd> dx -g @$devices().OrderBy(obj => obj.@"InstancePath")
```



	(+) DeviceNodeObject	InstancePath
[0x0] : ACPI\FixedButton\2&daba3ff&1	{...}	ACPI\FixedButton\2&daba3ff&1
[0x1] : ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_I...	{...}	ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_Intel(R)_...
[0x2] : ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_I...	{...}	ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_Intel(R)_...
[0x3] : ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_I...	{...}	ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_Intel(R)_...
[0x4] : ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_I...	{...}	ACPI\GenuineIntel_-_Intel64_Family_6_Model_30_-_Intel(R)_...
[0x5] : ACPI\IFX0102\1 (TPM)	{...}	ACPI\IFX0102\1
[0x6] : ACPI\PNP0000\4&23207e49&0	{...}	ACPI\PNP0000\4&23207e49&0

进程线程示例

调试器对象投影到根位于“Debugger”的命名空间中。 进程、模块、线程、堆栈、堆栈帧和局部变量都可用于 LINQ 查询。

此示例 JavaScript 演示如何显示当前会话进程的线程计数：

dbgcmd

```
function __Processes()
{
    return host.currentSession.Processes.Select(p => ({Name: p.Name,
ThreadCount: p.Threads.Count()}));
}

function initializeScript()
{
    return [new host.functionAlias(__Processes, "Processes")];
}
```

这显示了包含！ 的示例输出进程函数别名。

dbgcmd

```
0: kd> !Processes
@$Processes()
    [0x0]            : [object Object]
    [0x4]            : [object Object]
    [0x1b4]          : [object Object]
    [0x248]          : [object Object]
    [0x2c4]          : [object Object]
    [0x340]          : [object Object]
    [0x350]          : [object Object]
    [0x3d4]          : [object Object]
    [0x3e8]          : [object Object]
    [0x4c]           : [object Object]
    [0x214]          : [object Object]
    [0x41c]          : [object Object]
    [0x494]          : [object Object]

...
```

在此示例中，显示线程计数最大的前 5 个进程。

dbgcmd

```
0: kd> dx -r1 @$Processes().OrderByDescending(p =>p.ThreadCount),5
@$Processes().OrderByDescending(p =>p.ThreadCount),5
    [0x4]            : [object Object]
    [0x180]          : [object Object]
    [0x978]          : [object Object]
    [0xda4]          : [object Object]
    [0x3e8]          : [object Object]
    [...]
```

另请参阅

dx（显示调试器对象模型表达式）

将 LINQ 与调试器对象配合使用

NatVis 中的本机调试器对象

JavaScript 扩展中的本机调试器对象

调试器数据模型 C++ 概述

项目 • 2023/11/15

本主题概述如何使用调试器数据模型 C++ 接口来扩展和自定义调试器的功能。

本主题是一系列文章的一部分，该系列介绍了可从 C++ 访问的接口、如何使用这些接口生成基于 C++ 的调试器扩展，以及如何利用其他数据模型构造 (例如：JavaScript 或 NatVis) 从 C++ 数据模型扩展。

调试器数据模型 C++ 概述

调试器数据模型 C++ 接口

调试器数据模型 C++ 对象

调试器数据模型 C++ 的其他接口

调试器数据模型 C++ 的概念

调试器数据模型 C++ 脚本

调试器数据模型 C++ 接口概述

调试器数据模型是一种可扩展的对象模型，它能够让新调试器扩展（包括 JavaScript、NatVis 和 C++ 中的扩展）使用来自调试器的信息并生成可从调试器及其他扩展访问的信息。 写入数据模型 API 的构造在调试器的较新 (dx) 表达式计算器以及 JavaScript 扩展或 C++ 扩展中提供。

为了说明调试器数据模型的目标，请考虑此传统调试器命令。

控制台

```
0: kd> !process 0 0
PROCESS ffffe0007e6a7780
    SessionId: 1  Cid: 0f68    Peb: 7ff7cfe7a000  ParentCid: 0f34
    DirBase: 1f7fb9000  ObjectTable: ffffc001cec82780  HandleCount:  34.
    Image: echoapp.exe
...
```

调试器命令使用二进制掩码，并且它仅以非标准方式提供文本输出。 文本输出难以使用、格式化或扩展，并且布局特定于此命令。

将此与调试器数据模型 dx (显示调试器对象模型表达式) 命令进行对比。

控制台

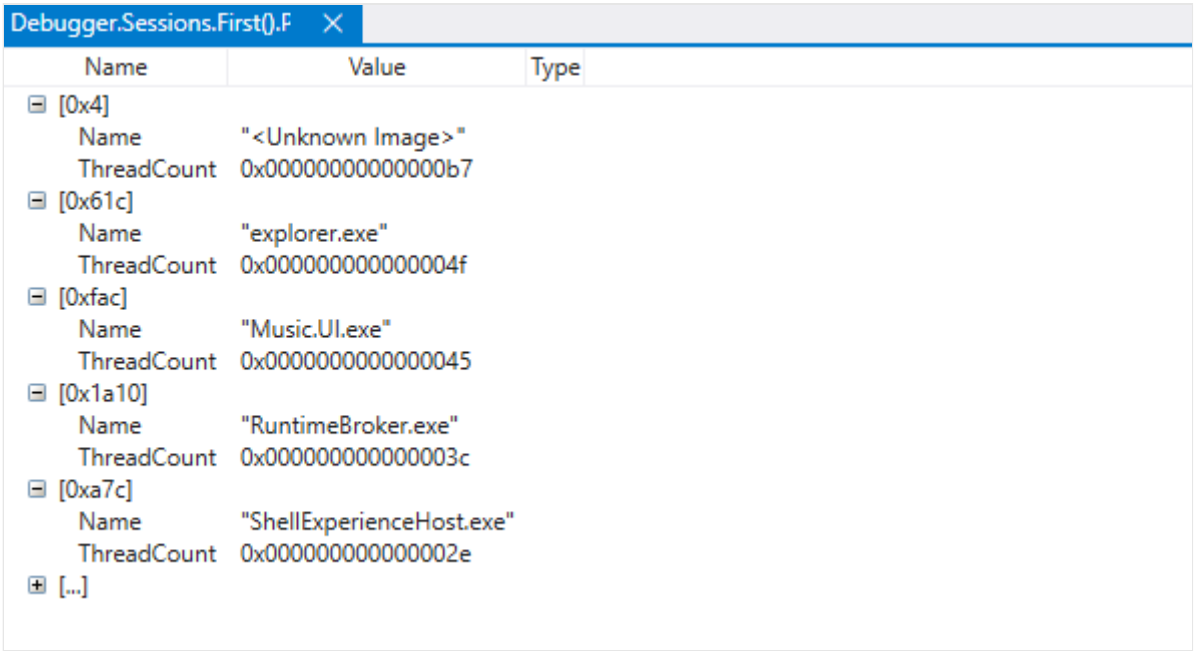
```
dx @$cursession.Processes.Where(p => p.Threads.Count() > 5)
```

此命令使用可发现、可扩展且可组合的统一方式的标准数据模型。

从逻辑上讲，名称间距和扩展特定对象允许发现调试器扩展功能。

> **提示**
>
> 因为数据模型 C++ 对象接口可以非常详细，以便为使用完整 C++ 异常的数据模型实现完整的 C++ 帮助程序库，建议使用模板编程范例。 有关详细信息，请参阅本主题后面的 **使用 DbgModelClientEx 库** 。

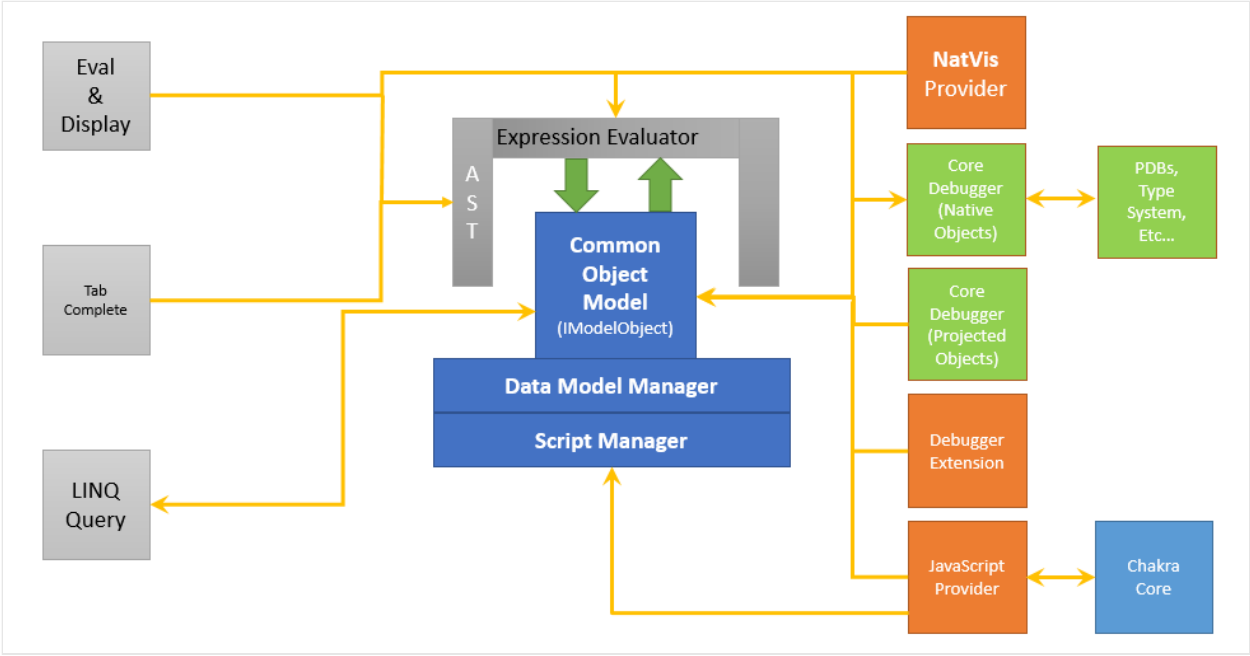
数据模型是 WinDbg 显示大多数内容的方式。 新 UI 中的许多元素可以查询、扩展或编写脚本，因为它们由数据模型提供支持。 有关详细信息，请参阅 WinDbg - 数据模型。



数据模型体系结构视图

下图汇总了调试器数据模型体系结构的主要元素。

- 左侧显示了 UI 元素，这些元素提供对对象的访问并支持 LINQ 查询等功能。
- 关系图右侧是向调试器数据模型提供数据的组件。 这包括自定义 NatVis、JavaScript 和 C++ 调试器数据模型扩展。

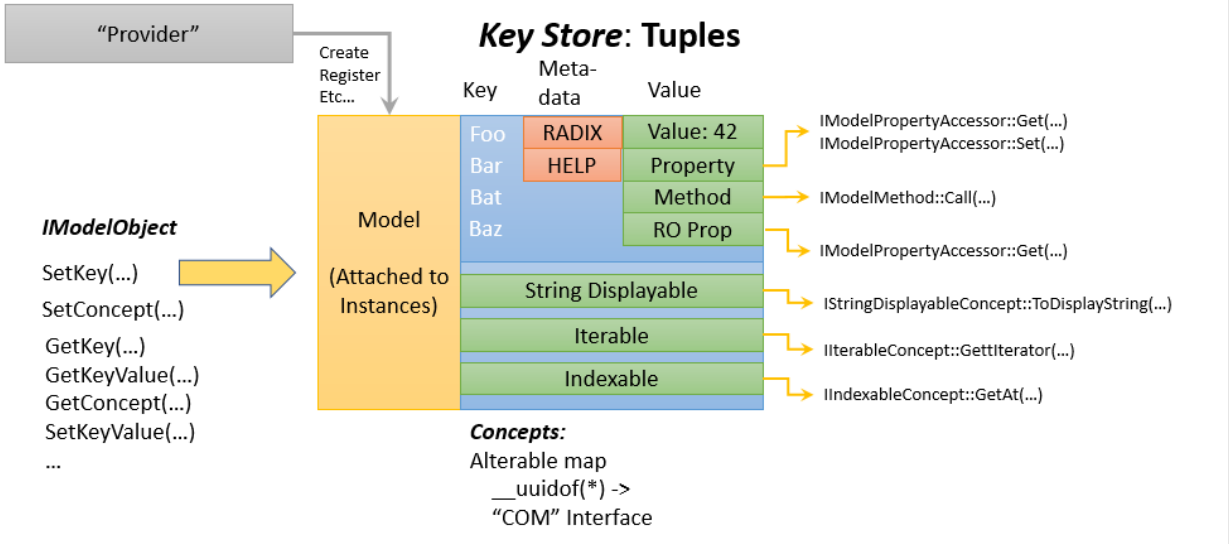


对象模型

调试器数据模型的中心是一个统一的对象表示形式，其中所有内容都是 IModelObject 接口的实例。 虽然此类对象可能表示内部 (例如：整数值) 或其他数据模型接口，但它通常表示一个动态对象 - 一个键/值/元数据元组字典和一组描述抽象行为的概念。

此图显示了 IModelObject 如何使用密钥存储来包含提供程序可以创建、注册和操作的值。

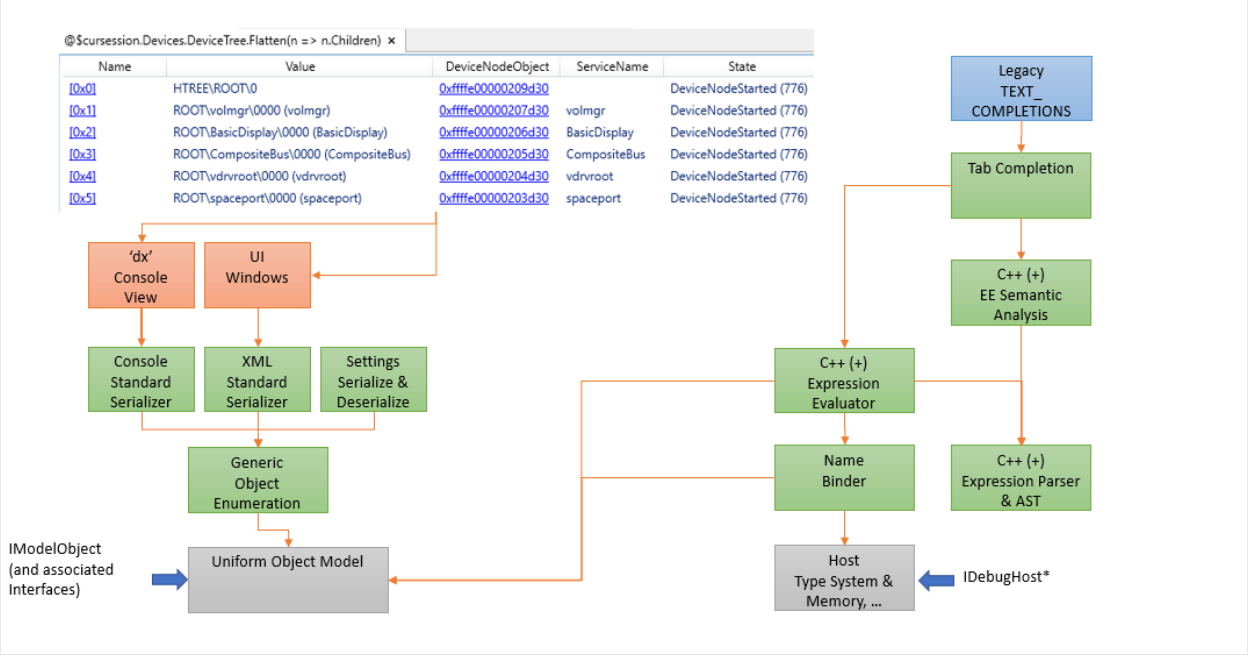
- 它显示了一个提供程序，该 *提供程序*向对象模型提供信息
- 左侧显示 *IModelObject*，即用于操作对象的通用对象模型。
- 中心是用于存储和访问值的 *密钥存储* 。
- 底部显示支持具有功能（例如转换为可显示字符串或编制索引）的对象 *的概念* 。



数据模型：使用者视图

下图显示了数据模型的使用者视图。 在此示例中， dx (Display Debugger Object Model Expression) 命令用于查询信息。

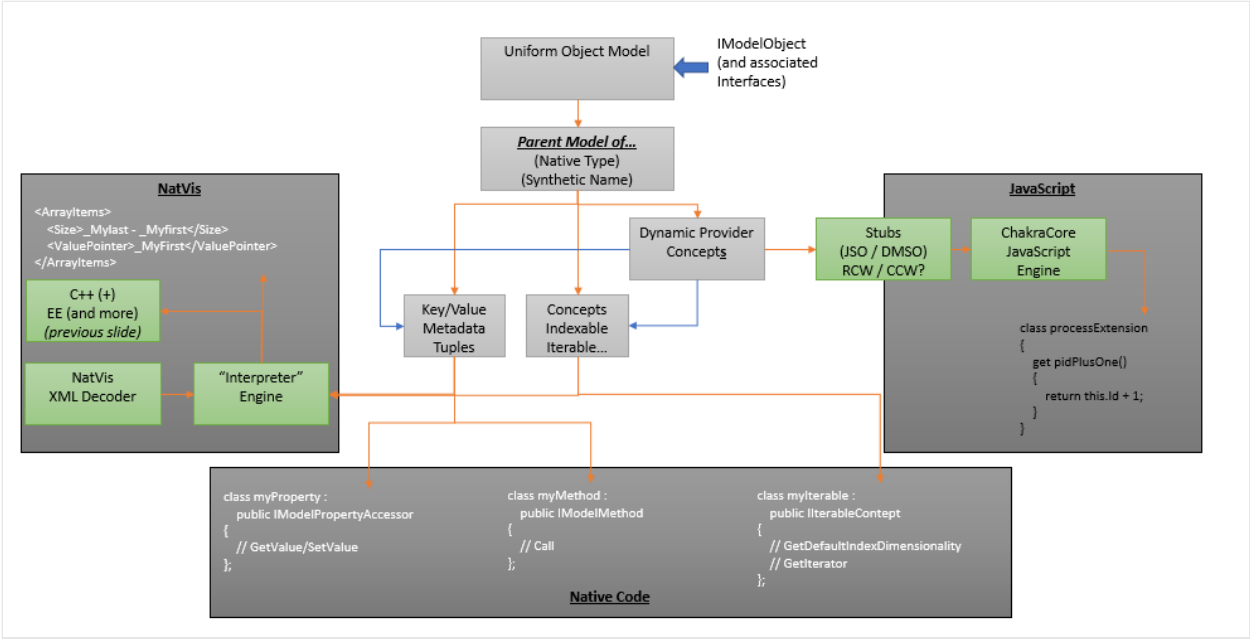
- Dx 命令通过序列化程序与对象枚举接口通信。
- IDebugHost* 对象用于从调试器引擎收集信息。
- 表达式和语义计算器用于将请求发送到调试器引擎。



数据模型：生成者视图

此图显示了数据模型的生成者视图。

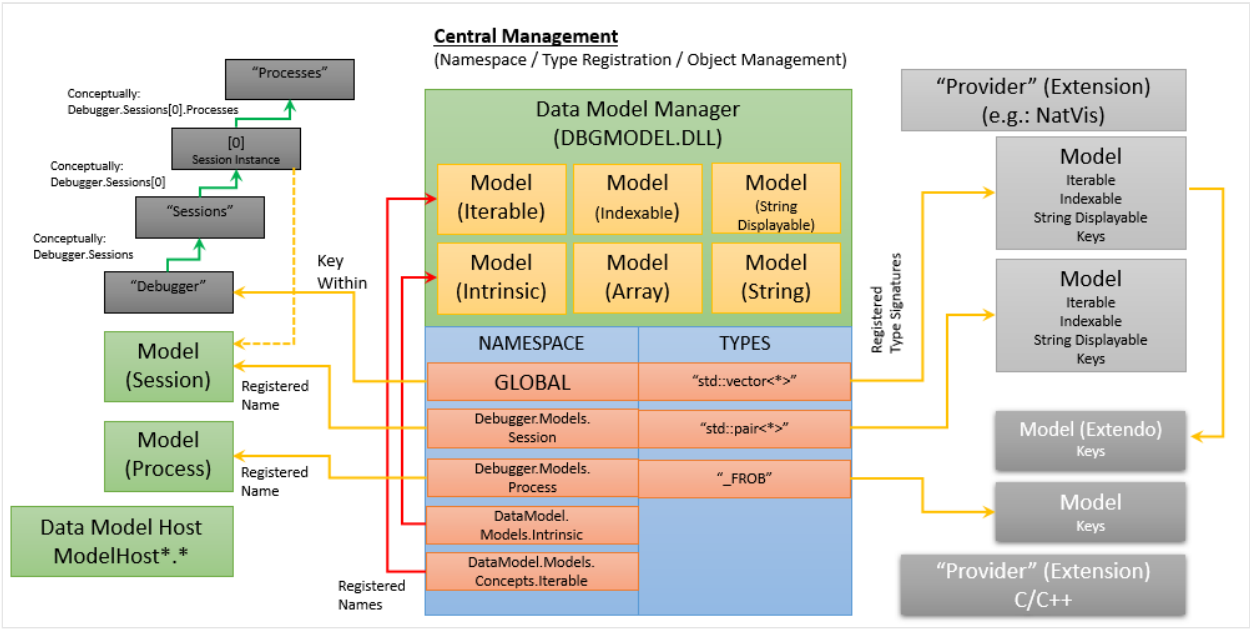
- 左侧显示了一个 NatVis 提供程序，该提供程序使用定义其他功能的 XML。
- JavaScript 提供程序可以利用 *动态提供程序概念* 来实时操作信息。
- 底部显示了本机代码提供程序，该提供程序还可以定义其他功能。



数据模型管理器

此图显示了数据模型管理器在对象管理中的核心作用。

- 数据模型管理器充当所有对象的中心注册器。
- 左侧显示了会话和进程等标准调试器元素的注册方式。
- 命名空间块显示中心注册列表。
- 图的右侧显示了两个提供程序，一个提供程序用于顶部的 NatVis，底部为 C/C++ 扩展。



调试器数据模型接口摘要

有许多 C++ 接口构成数据模型的不同部分。 为了以一致且简单的方式处理这些接口，它们按常规类别进行细分。 此处main区域：

常规对象模型

第一组也是最重要的接口定义如何访问核心数据模型以及如何访问和操作对象。 IModelObject 是表示数据模型中每个对象的接口，（非常类似于 C# 的对象）。 这是数据模型的使用者和生成者感兴趣的main接口。 其他接口是用于访问对象不同方面的机制。 为此类别定义了以下接口：

DbgEng 和数据模型之间的桥梁

IHostDataModelAccess

主接口

IModelObject

IKeyStore

IModelIterator

IModelPropertyAccessor

IModelMethod

IKeyEnumerator

IRawEnumerator

IModelKeyReference / IModelKeyReference2

概念接口

IStringDisplayableConcept

IIterableConcept

IIndexableConcept

IPreferredRuntimeTypeConcept

IDataModelConcept

IDynamicKeyProviderConcept

IDynamicConceptProviderConcept

管理数据模型和扩展性

数据模型管理器是管理所有扩展性发生方式的核心组件。 它是一组表的中心存储库，用于将本机类型映射到扩展点，并将合成构造映射到扩展点。 此外，它是一个实体，负责

对象的装箱 (序号值或字符串转换为 IModelObject 的) 。

为此类别定义了以下接口：

常规数据模型管理器访问权限

IDataModelManager / IDataModelManager2

脚本管理

IDataModelScriptManager

IDataModelScriptProviderEnumerator

访问调试器的类型系统和内存空间

详细公开了调试器的基础类型系统和内存空间，以供扩展使用。 为此类别定义了以下接口：

常规主机 (调试器) 接口

IDebugHost

IDebugHostStatus

IDebugHostContext

IDebugHostMemory / IDebugHostMemory2

IDebugHostErrorSink

IDebugHostEvaluator / IDebugHostEvaluator2

IDebugHostExtensibility

主机 (调试器) 类型系统接口

IDebugHostSymbols

IDebugHostSymbol / IDebugHostSymbol2

IDebugHostModule

IDebugHostType / IDebugHostType2

IDebugHostConstant

IDebugHostField

IDebugHostData

IDebugHostBaseClassIDebugHostPublic

IDebugHostModuleSignature

IDebugHostTypeSignature

主机 (调试器) 脚本支持

IDebugHostScriptHost

创作和使用脚本

数据模型还对什么是脚本以及如何调试脚本有一个常规概念。 调试器扩展完全有可能在数据模型和另一种动态语言之间定义一个通用桥，（通常是脚本环境）。 这组接口是实现的，也是调试器 UI 利用此类脚本的方式。

为此类别定义了以下接口：

常规脚本接口

IDataModelScriptProvider

IDataModelScript

IDataModelScriptClient

IDataModelScriptHostContext

IDataModelScriptTemplate

IDataModelScriptTemplateEnumerator

IDataModelNameBinder

脚本调试器接口

IDataModelScriptDebug

IDataModelScriptDebugClient

IDataModelScriptDebugStack

IDataModelScriptDebugStackFrame

IDataModelScriptDebugVariableSetEnumerator

IDataModelScriptDebugBreakpoint

IDataModelScriptDebugBreakpointEnumerator

使用 DbgModelClientEx 库

概述

数据模型 C++ 对象接口可以非常详细地实现。 虽然它们允许对数据模型进行完全操作，但它们需要实现多个小接口来扩展数据模型 (例如：每个动态可提取属性的 IModelPropertyAccessor 实现) 。 除此之外，基于 HRESULT 的编程模型增加了大量用于错误检查的锅炉板代码。

为了最大程度地减少其中的某些工作，有一个完整的 C++ 帮助程序库用于数据模型，该库使用完整的 C++ 异常和模板编程范例。 使用此库可以在使用或扩展数据模型时提供更简洁的代码，建议使用此库。

帮助程序库中有两个重要的命名空间：

Debugger：:D ataModel：： ClientEx - 使用数据模型的帮助程序

Debugger：:D ataModel：:P roviderEx - 数据模型扩展的帮助程序

有关使用 DbgModelClientEx 库的其他信息，请参阅此 github 站点上的自述文件：

https://github.com/Microsoft/WinDbg-Libraries/tree/master/DbgModelCppLib

HelloWorld C++ 示例

若要查看 DbgModelClientEx 库的使用方式，请查看此处的数据模型 HelloWorld C++ 示例。

https://github.com/Microsoft/WinDbg-Samples/tree/master/DataModelHelloWorld

该示例包括：

- HelloProvider.cpp - 这是提供程序类的实现，该提供程序类将新的示例属性“Hello”添加到调试器的进程概念中。
- SimpleIntroExtension.cpp - 这是一个简单的调试器扩展，它将新的示例属性“Hello”添加到调试器的进程概念中。 此扩展是针对数据模型 C++17 帮助程序库编写的。 更可取的是针对此库而不是原始 COM ABI 编写扩展，因为需要粘附代码的体积 (和复杂性) 。

JavaScript 和 COM 示例

为了更好地了解使用数据模型编写调试器扩展的不同方法，此处提供了三个版本的数据模型 HelloWorld 扩展：

https://github.com/Microsoft/WinDbg-Samples/tree/master/DataModelHelloWorld

- JavaScript - 用 JavaScript 编写的版本
- C++17 - 针对数据模型 C++17 客户端库编写的版本
- COM - 针对原始 COM ABI 编写的版本 (仅使用 WRL for COM 帮助程序)

另请参阅

调试器数据模型 C++ 接口

调试器数据模型 C++ 对象

调试器数据模型 C++ 的其他接口

调试器数据模型 C++ 的概念

调试器数据模型 C++ 脚本

调试器数据模型 C++ 接口

项目 • 2023/04/14

本主题提供并概述如何使用调试器数据模型 C++ 接口来扩展和自定义调试器的功能。

调试器数据模型 C++ 主机接口

调试器数据模型主机

调试器数据模型设计为可托管在各种不同的上下文中的组件化系统。 通常，数据模型托管在调试器应用程序的上下文中。 为了成为数据模型的主机，需要实现许多接口来公开调试器的核心方面：其目标、内存空间、计算器、符号和类型系统等...虽然这些接口由任何希望托管数据模型的应用程序实现，但它们由核心数据模型以及与数据模型互操作的任何扩展使用。

核心接口集包括：

接口名称	说明
IDebugHost	调试主机的核心接口。
IDebugHostStatus	允许客户端查询主机状态的接口。
IDebugHostContext	主机中上下文的抽象 (例如：特定目标、特定进程、特定地址空间等...)
IDebugHostErrorSink	调用方实现的接口，用于从主机和数据模型的某些部分接收错误
IDebugHostEvaluator / IDebugHostEvaluator2	调试主机的表达式计算器。
IDebugHostExtensibility	一个接口，用于扩展主机的功能或其部分 (，例如表达式计算器) 。

类型系统和符号接口包括：

InterfaceName	说明
IDebugHostSymbols	提供对符号的访问和解析的核心接口
IDebugHostSymbol / IDebugHostSymbol2	表示任何类型的单个符号。 特定符号是此接口的派生。
IDebugHostModule	表示在进程中加载的模块。 这是一种符号。

InterfaceName	说明
IDebugHostType / IDebugHostType2	表示本机/语言类型。
IDebugHostConstant	表示符号信息 (中的常量，例如：C++ 中的非类型模板参数)
IDebugHostField	表示结构或类中的字段。
IDebugHostData	表示模块中的数据 (如果此数据位于结构或类中，则为 IDebugHostField)
IDebugHostBaseClass	表示基类。
IDebugHostPublic	表示 PDB 的公共表内的符号。 这没有与之关联的类型信息。 它是名称和地址。
IDebugHostModuleSignature	表示模块签名 -- 将按名称和/或版本匹配一组模块的定义
IDebugHostTypeSignature	表示类型签名 -- 将按模块和/或名称匹配一组类型的定义

核心主机接口：IDebugHost

IDebugHost 接口是任何数据模型主机的核心接口。 它的定义如下：

C++

```
DECLARE_INTERFACE_(IDebugHost, IUnknown)
{
    STDMETHOD(GetHostDefinedInterface)(_COM_Outptr_ IUnknown** hostUnk)
PURE;
    STDMETHOD(GetCurrentContext)(_COM_Outptr_ IDebugHostContext** context)
PURE;
    STDMETHOD(GetDefaultMetadata)(_COM_Outptr_ IKeyStore**
defaultMetadataStore) PURE;
}
```

GetHostDefinedInterface

GetHostDefinedInterface 方法返回主机的main专用接口（如果给定主机存在）。 对于 Windows 调试工具，此处返回的接口是 IDebugClient (强制转换为 IUnknown) 。

GetCurrentContext

GetCurrentContext 方法返回一个接口，该接口表示调试器主机的当前状态。 具体含义由主机决定，但它通常包括会话、进程和调试主机用户界面中处于活动状态的地址空间等内容。 返回的上下文对象对调用方来说基本上是不透明的，但它是在对调试主机的调用之间传递的重要对象。 例如，当调用方正在读取内存时，必须知道从哪个进程和地址空间读取内存。 该概念封装在从此方法返回的上下文对象的概念中。

GetDefaultMetadata

GetDefaultMetadata 方法返回一个默认元数据存储区，该存储区可用于某些操作 (例如：未传递显式元数据时字符串转换) 。 这样，调试主机就可以对某些数据的显示方式进行某种控制。 例如，默认元数据可能包含 PreferredRadix 键，使主机能够指示在未指定的情况下，是应以十进制还是十六进制显示序号。

请注意，默认元数据存储中的属性值必须手动解析，并且必须传递要为其查询默认元数据的对象。 应使用 GetKey 方法代替 GetKeyValue。

状态接口：IDebugHostStatus

IDebugHostStatus 接口允许数据模型或调试主机的客户端查询调试主机状态的某些方面。 接口定义如下：

```
C++

DECLARE_INTERFACE_(IDebugHostStatus, IUnknown)
{
    STDMETHOD(PollUserInterrupt)(_Out_ bool* interruptRequested) PURE;
}
```

PollUserInterrupt

PollUserInterrupt 方法用于查询调试主机的用户是否已请求中断当前操作。 例如，数据模型中的属性访问器可以调用任意代码 (例如：javaScript 方法) 。 该代码可能需要任意时间。 为了使调试主机保持响应，任何可能需要任意时间的此类代码都应通过调用此方法检查中断请求。 如果 interruptRequested 值返回为 true，则调用方应立即中止并返回 E_ABORT的结果。

上下文接口：IDebugHostContext

上下文是数据模型和基础调试主机最重要的方面之一。 当你持有对象时，必须能够知道对象来自何处--它所在的进程，它与哪个地址空间相关联。 了解此信息可以正确解释指针值等内容。 必须将 IDebugHostContext 类型的对象传递给调试主机上的许多方法。 可以通过多种方式获取此接口：

- 通过获取调试器的当前上下文：调用 IDebugHost 的 GetCurrentContext 方法
- 通过获取对象的上下文：调用 IModelObject 的 GetContext 方法
- 通过获取符号的上下文：调用 IDebugHostSymbol 的 GetContext 方法

此外，还有两个值在 IDebugHostContext 接口的上下文中具有特殊含义，这些接口从数据模型或调试主机方法返回或传递给该接口：

nullptr：指示没有上下文。 对于某些对象来说，没有上下文是完全有效的。 数据模型的根命名空间中的 Debugger 对象不引用特定进程或地址空间中的任何内容。 它没有上下文。

USE_CURRENT_HOST_CONTEXT：一个 sentinel 值，指示应使用调试主机的当前 UI 上下文。 此值永远不会从调试主机返回。 但是，它可以传递给任何采用输入 IDebugHostContext 的调试主机方法，而不是显式调用 IDebugHost 的 GetCurrentContext 方法。 请注意，显式传递USE_CURRENT_HOST_CONTEXT通常比显式获取当前上下文的性能更高。

主机上下文的上下文对调用方而言基本上是不透明的。 核心调试主机外部的调用方可以对主机上下文执行的唯一操作是将其与其他主机上下文进行比较。

IDebugHostContext 接口定义如下：

```
C++

DECLARE_INTERFACE_(IDebugHostContext, IUnknown)
{
    STDMETHOD(IsEqualTo)(_In_ IDebugHostContext *pContext, _Out_ bool
*pIsEqual) PURE;
}
```

IsEqualTo

IsEqualTo 方法将主机上下文与另一个主机上下文进行比较。 如果两个上下文是等效的，则返回此指示。 请注意，此比较不是接口等效性。 这会比较上下文本身的基础不透明内容。

错误接收器： IDebugHostErrorSink

IDebugHostErrorSink 是客户端接收某些操作期间发生的错误的通知，并在需要时路由这些错误的方法。 接口定义如下：

```
C++

enum ErrorClass
{
    ErrorClassWarning,
    ErrorClassError
}
```

```
C++

DECLARE_INTERFACE_(IDebugHostErrorSink, IUnknown)
{
```

```
    STDMETHOD(ReportError)(_In_ ErrorClass errClass, _In_ HRESULT hrError,
_In_ PCWSTR message) PURE;
}
```

ReportError

ReportError 方法是对错误接收器的回调，用于通知错误已发生，并允许接收器将错误路由到任何合适的 UI 或机制。

主机计算器：IDebugHostEvaluator/IDebugHostEvaluator2

调试主机向客户端提供的最重要功能之一是访问其基于语言的表达式计算器。IDebugHostEvaluator 和 IDebugHostEvaluator2 接口是从调试主机访问该功能的方法。

接口定义如下：

```
C++
```

```
DECLARE_INTERFACE_(IDebugHostEvaluator2, IDebugHostEvaluator)
{
    //
    // IDebugHostEvaluator:
    //
    STDMETHOD(EvaluateExpression)(_In_ IDebugHostContext* context, _In_
PCWSTR expression, _In_opt_ IModelObject* bindingContext, _COM_Errorptr_
IModelObject** result, _COM_Outptr_opt_result_maybenull_ IKeyStore**
metadata) PURE;
    STDMETHOD(EvaluateExtendedExpression)(_In_ IDebugHostContext* context,
_In_ PCWSTR expression, _In_opt_ IModelObject* bindingContext,
_COM_Errorptr_ IModelObject** result, _COM_Outptr_opt_result_maybenull_
IKeyStore** metadata) PURE;
    //
    // IDebugHostEvaluator2:
    //
    STDMETHOD(AssignTo)(_In_ IModelObject* assignmentReference, _In_
IModelObject* assignmentValue, _COM_Errorptr_ IModelObject**
assignmentResult, _COM_Outptr_opt_result_maybenull_ IKeyStore**
assignmentMetadata) PURE;
}
```

EvaluateExpression

EvaluateExpression 方法允许请求调试主机计算语言 (例如：C++) 表达式，并返回该表达式计算的结果值（装箱为 IModelObject）。 方法的此特定变体仅允许语言构造。 调试主机的表达式计算器中呈现的、语言 (中不存在的任何其他功能，例如：) 的 LINQ 查询方法已关闭用于计算。

EvaluateExtendedExpression

EvaluateExtendedExpression 方法类似于 EvaluateExpression 方法，只不过它重新启用特定调试主机选择添加到其表达式计算器的其他非语言功能。 例如，对于 Windows 调试工具，这将启用匿名类型、LINQ 查询、模块限定符、格式说明符和其他非 C/C++ 功能。

IDebugHostEvaluator2

AssignTo

AssignTo 方法根据所调试语言的语义执行赋值。

主机扩展性接口：IDebugHostExtensibility

调试主机的某些功能（可选）受扩展性的约束。 例如，这可能包括表达式计算器。 IDebugHostExtensibility 接口是访问这些扩展点的方式。 接口定义如下：

```
C++

DECLARE_INTERFACE_(IDebugHostExtensibility, IUnknown)
{
    STDMETHOD(CreateFunctionAlias)(_In_ PCWSTR aliasName, _In_ IModelObject
*functionObject) PURE;
    STDMETHOD(DestroyFunctionAlias)(_In_ PCWSTR aliasName) PURE;
}
```

CreateFunctionAlias

CreateFunctionAlias 方法创建“函数别名”，即在某些扩展中实现的方法的“快速别名”。 此别名的含义特定于主机。 它可能会使用 函数扩展主机的表达式计算器，也可以执行完全不同的操作。

DestroyFunctionAlias

DestroyFunctionAlias 方法撤消先前对 CreateFunctionAlias 方法的调用。 该函数将不再在快速别名下可用。

访问数据模型

首先，数据模型扩展性 API 设计为对应用程序 (通常是充当数据模型的主机的调试器) 。 从理论上讲，任何应用程序都可以通过提供一组主机 API 来托管数据模型，这些 API 公开应用程序的调试目标的类型系统 () 和一组投影对象到数据模型的命名空间中，了解目标、进程、线程等...位于这些调试目标 () 中。

虽然数据模型 API（以 IDataModel、*IDebugHost* 和 IModelObject 的分支开头的 API）设计为可移植，但它们不定义什么是“调试器扩展”。 目前，希望扩展 Windows 调试工具的

组件及其提供的引擎必须编写引擎扩展才能访问数据模型。 该引擎扩展只需是引擎扩展，就像扩展的加载和引导机制一样。 因此，最小实现将提供：

- **DebugExtensionInitialize**：一种利用创建的 IDebugClient 来访问数据模型并设置对象模型操作的方法。
- **DebugExtensionUninitialize**：一种撤消在 DebugExtensionInitialize 中执行的对象模型操作的方法。
- **DebugExtensionCanUnload**：返回扩展是否可以卸载的方法。 如果扩展中仍有实时 COM 对象，则必须指示这一点。 这是调试器相当于 COM 的 DllCanUnloadNow。 如果这会返回无法卸载的S_FALSE指示，则调试器稍后可以查询此卸载是否安全，或者它可能通过再次调用 DebugExtensionInitialize 重新初始化扩展。 扩展必须准备好处理这两个路径。
- **DebugExtensionUnload**：在 DLL 卸载之前执行所需的任何最终清理的方法

Bridge 接口：IHostDataModelAccess

如前所述，调用 DebugExtensionInitialize 时，它会创建调试客户端并获取对数据模型的访问权限。 此类访问由 Windows 调试工具的旧 IDebug* 接口和数据模型之间的桥接接口提供。 此桥接口为“IHostDataModelAccess”，定义如下：

```
C++

DECLARE_INTERFACE_(IHostDataModelAccess, IUnknown)
{
   STDMETHOD(GetDataModel)(_COM_Outptr_ IDataModelManager** manager,
_COM_Outptr_ IDebugHost** host) PURE;
}
```

GetDataModel

GetDataModel 方法是桥接口上的方法，该方法提供对数据模型的两端的访问：调试主机(调试器) 的下边缘由返回的 IDebugHost 接口表示。数据模型的main组件 - 数据模型管理器由返回的 IDataModelManager 接口表示

调试器数据模型系统接口

数据模型主机

调试器数据模型设计为可托管在各种不同的上下文中的组件化系统。 通常，数据模型托管在调试器应用程序的上下文中。 为了成为数据模型的主机，需要实现许多接口来公开调试器的核心方面：其目标、内存空间、计算器、符号和类型系统等...虽然这些接口由任何希望托管数据模型的应用程序实现，但它们由核心数据模型以及与数据模型互操作的任何扩展使用。

类型系统和符号接口包括：

接口名称	说明
IDebugHostSymbols	提供对符号的访问和解析的核心接口
IDebugHostSymbol / IDebugHostSymbol2	表示任何类型的单个符号。 特定符号是此接口的派生。
IDebugHostModule	表示在进程中加载的模块。 这是一种符号。
IDebugHostType / IDebugHostType2	表示本机/语言类型。
IDebugHostConstant	表示符号信息 (中的常量，例如：C++ 中的非类型模板参数)
IDebugHostField	表示结构或类中的字段。
IDebugHostData	表示模块中的数据 (如果此数据位于结构或类中，则为 IDebugHostField)
IDebugHostBaseClass	表示基类。
IDebugHostPublic	表示 PDB 的公共表内的符号。 这没有与之关联的类型信息。 它是名称和地址。
IDebugHostModuleSignature	表示模块签名 -- 将按名称和/或版本匹配一组模块的定义
IDebugHostTypeSignature	表示类型签名 -- 将按模块和/或名称匹配一组类型的定义

其他核心接口包括：

接口名称	说明
IDebugHost	调试主机的核心接口。
IDebugHostStatus	允许客户端查询主机状态的接口。
IDebugHostContext	主机中上下文的抽象 (例如：特定目标、特定进程、特定地址空间等...)
IDebugHostErrorSink	调用方实现的接口，用于接收来自主机和数据模型某些部分的错误
IDebugHostEvaluator / IDebugHostEvaluator2	调试主机的表达式计算器。
IDebugHostExtensibility	一个接口，用于扩展主机或其部分的功能 (，例如表达式计算器) 。

主符号接口：IDebugHostSymbols

IDebugHostSymbols 接口是访问调试目标中的符号main起点。 可以从 IDebugHost 实例查询此接口，定义如下：

```
C++

DECLARE_INTERFACE_(IDebugHostSymbols, IUnknown)
{
    STDMETHOD(CreateModuleSignature)(_In_z_ PCWSTR pwszModuleName,
_In_opt_z_ PCWSTR pwszMinVersion, _In_opt_z_ PCWSTR pwszMaxVersion, _Out_
IDebugHostModuleSignature** ppModuleSignature) PURE;
    STDMETHOD(CreateTypeSignature)(_In_z_ PCWSTR signatureSpecification,
_In_opt_ IDebugHostModule* module, _Out_ IDebugHostTypeSignature**
typeSignature) PURE;
    STDMETHOD(CreateTypeSignatureForModuleRange)(_In_z_ PCWSTR
signatureSpecification, _In_z_ PCWSTR moduleName, _In_opt_z_ PCWSTR
minVersion, _In_opt_z_ PCWSTR maxVersion, _Out_ IDebugHostTypeSignature**
typeSignature) PURE;
    STDMETHOD(EnumerateModules)(_In_ IDebugHostContext* context,
_COM_Outptr_ IDebugHostSymbolEnumerator** moduleEnum) PURE;
    STDMETHOD(FindModuleByName)(_In_ IDebugHostContext* context, _In_z_
PCWSTR moduleName, _COM_Outptr_ IDebugHostModule **module) PURE;
    STDMETHOD(FindModuleByLocation)(_In_ IDebugHostContext* context, _In_
Location moduleLocation, _COM_Outptr_ IDebugHostModule **module) PURE;
    STDMETHOD(GetMostDerivedObject)(_In_opt_ IDebugHostContext *pContext,
_In_ Location location, _In_ IDebugHostType* objectType, _Out_ Location*
derivedLocation, _Out_ IDebugHostType** derivedType) PURE;
}
```

CreateModuleSignature

CreateModuleSignature 方法创建一个签名，该签名可用于按名称和版本（可选）匹配一组特定模块。 模块签名有三个组件：

- 名称：匹配模块的名称必须与签名中的名称完全不区分大小写
- 最低版本：如果指定，匹配的模块必须具有至少与此版本一样高的最低版本。 版本以“A.B.C.D”格式指定，每个后续部分都不像之前部分那么重要。 只有第一个段是必需的。
- 最大版本：如果指定，匹配的模块必须具有不高于此版本的最大版本。 版本以“A.B.C.D”格式指定，每个后续部分都不像之前部分那么重要。 只有第一个段是必需的。

CreateTypeSignature

CreateTypeSignature 方法创建一个签名，该签名可用于通过包含模块和类型名称来匹配一组具体类型。 类型名称签名字符串的格式特定于正在调试 (和调试主机) 的语言。 对于 C/C++，签名字符串等效于 NatVis 类型规范。 也就是说，签名字符串是一种类型名称，其中 (指定为 *) 的通配符可用于模板参数。

CreateTypeSignatureForModuleRange

CreateTypeSignatureForModuleRange 方法创建一个签名，该签名可用于按模块签名和类型名称匹配一组具体类型。 这类似于 CreateTypeSignature 方法，不同之处在于调用方传递创建模块签名所需的参数，而不是传递特定模块以匹配签名 (就好像模块签名是使用 CreateModuleSignature 方法) 创建的。

EnumerateModules

EnumerateModules 方法创建枚举器，该枚举器将枚举特定主机上下文中可用的每个模块。 该主机上下文可能封装进程上下文，也可能封装类似 Windows 内核的内容。

FindModuleByName

FindModuleByName 方法将查看给定的主机上下文，并找到具有指定名称的模块，并返回一个接口。 使用或不带文件扩展名按名称搜索模块是合法的。

FindModuleByLocation

FindModuleByLocation 方法将查看给定的主机上下文，并确定哪个模块包含指定位置提供的地址。 然后，它会将接口返回到此类模块。

GetMostDerivedObject

GetMostDerivedObject 将使用调试器的类型系统从其静态类型确定对象的运行时类型。 此方法将仅使用类型系统层中提供的符号信息和启发法来执行此分析。 此类信息可能包括 C++ RTTI (运行时类型信息，) 或分析对象的虚拟函数表的形状。 它不包括 IModelObject 上的首选运行时类型概念等内容。 如果分析找不到运行时类型或找不到不同于传递到 方法的静态类型的运行时类型，则可能会传递输入位置和类型。方法不会因为这些原因而失败。

核心单个符号接口：IDebugHostSymbol

可从数据模型主机返回的每个符号都将以某种方式从 IDebugHostSymbol 派生。 这是每个符号实现的核心接口，无论符号的类型如何。 根据符号类型，给定符号可以实现一组其他接口，这些接口返回此接口所表示的特定符号类型更独特的属性。
IDebugHostSymbol2 /IDebugHostSymbol 接口定义如下：

```
C++

DECLARE_INTERFACE_(IDebugHostSymbol2, IDebugHostSymbol)
{
    //
    // IDebugHostSymbol:
    //
    STDMETHOD(GetContext)(_COM_Outptr_ IDebugHostContext** context) PURE;
    STDMETHOD(EnumerateChildren)(_In_ SymbolKind kind, _In_opt_z_ PCWSTR
```

```
name, _Out_ IDebugHostSymbolEnumerator **ppEnum) PURE;
    STDMETHOD(GetSymbolKind)(_Out_ SymbolKind *kind) PURE;
    STDMETHOD(GetName)(_Out_ BSTR* symbolName) PURE;
    STDMETHOD(GetType)(_Out_ IDebugHostType** type) PURE;
    STDMETHOD(GetContainingModule)(_Out_ IDebugHostModule
**containingModule) PURE;
    STDMETHOD(CompareAgainst)(_In_ IDebugHostSymbol *pComparisonSymbol, _In_
ULONG comparisonFlags, _Out_ bool *pMatches) PURE;
    //
    // IDebugHostSymbol2
    //
    STDMETHOD(EnumerateChildrenEx)(_In_ SymbolKind kind, _In_opt_z_ PCWSTR
name, _In_opt_ SymbolSearchInfo* searchInfo, _Out_
IDebugHostSymbolEnumerator **ppEnum) PURE;
}
```

请务必注意，此接口表示多种符号 -- 由 SymbolKind 枚举描述，其值如下所示：

Enumarant	**含义**
符号	未指定的符号类型
SymbolModule	符号是一个模块，可以查询 IDebugHostModule
SymbolType	符号是一种类型，可以查询 IDebugHostType
SymbolField	符号是结构或类) 中的数据成员 (字段，可查询 IDebugHostField
SymbolConstant	符号是一个常量值，可查询 IDebugHostConstant
SymbolData	符号是非结构或类成员且可查询 IDebugHostData 的数据
SymbolBaseClass	符号是基类，可查询 IDebugHostBaseClass
SymbolPublic	符号是模块的公共表中的条目， (没有) 类型信息，并且可查询 IDebugHostPublic
SymbolFunction	符号是一个函数，可查询 IDebugHostData

GetContext

GetContext 方法返回符号有效的上下文。 虽然这将表示符号所在的调试目标和进程/地址空间等内容，但它可能不像从其他方式检索 (上下文那样具体，例如：从 *IModelObject*) 。

EnumerateChildren

EnumerateChildren 方法返回枚举器，该枚举器将枚举给定符号的所有子级。 例如，对于 C++ 类型，基类、字段、成员函数等都被视为类型符号的子级。

模块接口：IDebugHostModule

调试器对在某些地址空间中加载的模块的概念在数据模型中以两种不同的方式表示：在类型系统级别，通过 IDebugHostModule 接口。 在这里，模块是一个符号，模块的核心属性是接口方法调用，通过 Debugger.Models.Module 数据模型在数据模型级别投影。 这是模块的类型系统 IDebugHostModule 表示形式的可扩展封装。

IDebugHostModule 接口定义如下， (忽略 IDebugHostSymbol) 通用的方法：

```
C++
```

```
DECLARE_INTERFACE_(IDebugHostModule, IDebugHostSymbol)
{
    //
    // IDebugHostModule:
    //
    STDMETHOD(GetImageName)(_In_ bool allowPath, _Out_ BSTR* imageName)
PURE;
    STDMETHOD(GetBaseLocation)(_Out_ Location* moduleBaseLocation) PURE;
    STDMETHOD(GetVersion)(_Out_opt_ ULONG64* fileVersion, _Out_opt_ ULONG64*
productVersion) PURE;
    STDMETHOD(FindTypeByName)(_In_z_ PCWSTR typeName, _Out_ IDebugHostType**
type) PURE;
    STDMETHOD(FindSymbolByRVA)(_In_ ULONG64 rva, _Out_ IDebugHostSymbol**
symbol) PURE;
    STDMETHOD(FindSymbolByName)(_In_z_ PCWSTR symbolName, _Out_
IDebugHostSymbol** symbol) PURE;
}
```

GetImageName

GetImageName 方法返回模块的图像名称。 根据 allowPath 参数的值，返回的图像名称可能包含也可能不包含图像的完整路径。

GetBaseLocation

GetBaseLocation 方法将模块的基本加载地址作为位置结构返回。 模块返回的位置结构通常引用虚拟地址。

GetVersion

GetVersion 方法返回有关模块 (的版本信息，前提是此类信息可以成功从) 标头中读出。 如果通过非 nullptr 输出指针) 请求 (给定版本，并且无法读取该版本，则将从方法调用返回相应的错误代码。

FindTypeByName

FindTypeByName 方法查找模块中按类型名称定义的类型，并为其返回类型符号。 此方法可能会返回有效的 IDebugHostType，该类型永远不会通过模块子级的显式递归返回。 调试主机可能允许创建派生类型 -- 类型从未在模块本身中使用过，但派生自的类型。 例

如，如果结构 MyStruct 在模块的符号中定义，但从未使用 MyStruct ** 类型，则 FindTypeByName 方法可能会合法地返回 MyStruct ** 的类型符号，尽管该类型名称从未显式出现在模块的符号中。

FindSymbolByRVA

FindSymbolByRVA 方法将在模块中给定的相对虚拟地址处查找单个匹配符号。 如果提供的 RVA (没有单个符号，例如：) 有多个匹配项，则此方法将返回错误。 请注意，此方法更倾向于返回私有符号，而不是 publics 表中的符号。

FindSymbolByName

FindSymbolByName 方法将在模块中查找给定名称的单个全局符号。 如果没有与给定名称匹配的单个符号，此方法将返回错误。 请注意，此方法更倾向于返回私有符号，而不是 publics 表中的符号。

访问类型系统：IDebugHostType2 /IDebugHostType

给定的语言/本机类型由 IDebugHostType2 或 IDebugHostType 接口描述。 请注意，这些接口上的某些方法仅适用于特定类型的类型。 给定的类型符号可以引用 TypeKind 枚举所述的以下类型之一：

类型类型	说明
TypeUDT	用户定义的类型 (结构、类、联合等...) 。具有类型为 TypeUDT 的本机类型的模型对象具有 ObjectTargetObject 的规范表示形式，其中类型始终保留在相应的 IModelObject 中。
TypePointer	指针。 具有类型为 TypePointer 的本机类型的模型对象具有 ObjectIntrinsic 的规范表示形式，其中指针的值为零扩展到VT_UI8，并保留为 64 位形式的内部数据。 TypePointer 的任何类型符号都具有由 GetBaseType 方法返回的基类型 () 指针指向的类型。
TypeMemberPointer	指向类成员的指针。 具有类型为 TypeMemberPointer 的本机类型的模型对象具有规范表示形式，该规范表示形式是内部 (值与指针值) 相同。 此值的确切含义是特定于编译器/调试主机的。
TypeArray	一个数组。 具有类型为 TypeArray 的本机类型的模型对象具有 ObjectTargetObject 的规范表示形式。 数组的基址是对象的位置， (通过 GetLocation 方法检索) 并且始终保留数组的类型。 TypeArray 的任何类型符号都具有由 getBaseType 方法返回的基类型 () 数组所返回的类型。
TypeFunction	函数。

类型类型	说明
TypeTypedef	typedef。 具有本机类型的模型对象，其类型为 TypeTypedef，其规范表示形式与 typedef 基础的最终类型的规范表示形式相同。 除非使用 IDebugHostType2 的显式 typedef 方法查询 typedef 信息，或者存在针对 typedef 注册的显式数据模型，否则这看起来对对象和类型信息的最终用户完全透明。 请注意，GetTypeKind 方法永远不会返回 TypeTypedef。 每个方法都将返回 typedef 将返回的最终类型。 IDebugHostType2 上有特定于 typedef 的方法，可用于获取 typedef 特定信息。
TypeEnum	枚举。 具有类型为 TypeEnum 的本机类型的模型对象具有 ObjectIntrinsic 的规范表示形式，其中内部函数的值和类型与枚举值相同。
TypeIntrinsic	内部 (基类型) 。 具有类型为 TypeIntrinsic 的本机类型的模型对象具有 ObjectIntrinsic 的规范表示形式。 类型信息可能保留，也可能不保留 -- 特别是当基础类型由存储在 IModelObject 中的内部数据的变体数据类型 (VT_*) 完全描述时

整体 IDebugHostType2/IDebugHostType 接口定义如下 (不包括 IDebugHostSymbol 方法)：

C++

```
DECLARE_INTERFACE_(IDebugHostType2, IDebugHostType)
{
    //
    // IDebugHostType:
    //
    STDMETHOD(GetTypeKind)(_Out_ TypeKind *kind) PURE;
    STDMETHOD(GetSize)(_Out_ ULONG64* size) PURE;
    STDMETHOD(GetBaseType)(_Out_ IDebugHostType** baseType) PURE;
    STDMETHOD(GetHashCode)(_Out_ ULONG* hashCode) PURE;
    STDMETHOD(GetIntrinsicType)(_Out_opt_ IntrinsicKind *intrinsicKind,
_Out_opt_ VARTYPE *carrierType) PURE;
    STDMETHOD(GetBitField)(_Out_ ULONG* lsbOfField, _Out_ ULONG*
lengthOfField) PURE;
    STDMETHOD(GetPointerKind)(_Out_ PointerKind* pointerKind) PURE;
    STDMETHOD(GetMemberType)(_Out_ IDebugHostType** memberType) PURE;
    STDMETHOD(CreatePointerTo)(_In_ PointerKind kind, _COM_Outptr_
IDebugHostType** newType) PURE;
    STDMETHOD(GetArrayDimensionality)(_Out_ ULONG64* arrayDimensionality)
PURE;
    STDMETHOD(GetArrayDimensions)(_In_ ULONG64 dimensions,
_Out_writes_(dimensions) ArrayDimension *pDimensions) PURE;
    STDMETHOD(CreateArrayOf)(_In_ ULONG64 dimensions, _In_reads_(dimensions)
ArrayDimension *pDimensions, _COM_Outptr_ IDebugHostType** newType) PURE;
    STDMETHOD(GetFunctionCallingConvention)(_Out_ CallingConventionKind*
conventionKind) PURE;
    STDMETHOD(GetFunctionReturnType)(_COM_Outptr_ IDebugHostType**
returnType) PURE;
    STDMETHOD(GetFunctionParameterTypeCount)(_Out_ ULONG64* count) PURE;
    STDMETHOD(GetFunctionParameterTypeAt)(_In_ ULONG64 i, _Out_
```

```
IDebugHostType** parameterType) PURE;
    STDMETHOD(IsGeneric)(_Out_ bool* isGeneric) PURE;
    STDMETHOD(GetGenericArgumentCount)(_Out_ ULONG64* argCount) PURE;
    STDMETHOD(GetGenericArgumentAt)(_In_ ULONG64 i, _Out_ IDebugHostSymbol**
argument) PURE;
    //
    // IDebugHostType2:
    //
    STDMETHOD(IsTypedef)(_Out_ bool* isTypedef) PURE;
    STDMETHOD(GetTypedefBaseType)(_Out_ IDebugHostType2** baseType) PURE;
    STDMETHOD(GetTypedefFinalBaseType)(_Out_ IDebugHostType2**
finalBaseType) PURE;
    STDMETHOD(GetFunctionVarArgsKind)(_Out_ VarArgsKind* varArgsKind) PURE;
}
```

IDebugHostType2/IDebugHostType 常规方法

以下 IDebugHostType 方法适用于任何类型，无论从 GetTypeKind 方法返回哪种类型：

```
C++

STDMETHOD(GetTypeKind)(_Out_ TypeKind *kind) PURE;
STDMETHOD(GetSize)(_Out_ ULONG64* size) PURE;
STDMETHOD(GetBaseType)(_Out_ IDebugHostType** baseType) PURE;
STDMETHOD(GetHashCode)(_Out_ ULONG* hashCode) PURE;
```

GetTypeKind

GetTypeKind 方法返回 (指针、数组、内部函数等类型...) 符号所引用的类型。

GetSize

GetSize 方法返回类型 (的大小，就像在 C++) 中 (类型) 一样。

GetBaseType

如果类型是另一个类型 (例如：MyStruct * 派生自 MyStruct')，则 GetBaseType 方法返回派生的基类型。 对于指针，这将返回指向的类型。 对于数组，这将返回数组的数组。 如果该类型不是此类派生类型，则返回错误。

GetHashCode

GetHashCode 方法返回类型的 32 位哈希代码。 除了全局匹配 (例如：与 * 等效的类型签名（如果主机) 允许）匹配所有内容，任何可以匹配特定类型签名的类型实例都必须返回相同的哈希代码。 此方法与类型签名结合使用，以便将类型签名与类型实例匹配。

IDebugHostType2/IDebugHostType 内部方法

以下 IDebugHostType 方法特定于内部类型 (或保存内部数据（如枚举）) 的类型：

```
C++

STDMETHOD(GetIntrinsicType)(_Out_opt_ IntrinsicKind *intrinsicKind,
_Out_opt_ VARTYPE *carrierType) PURE;
```

GetIntrinsicType

GetIntrinsicType 方法返回有关类型内部类型的信息。 此方法返回两个值：

- 内部类型指示总体类型 (例如：整数、无符号、浮点) 但不是类型 (的大小，例如：8 位、16 位、32 位、64 位)
- 载体类型指示内部类型如何打包到 VARIANT 结构中。 这是一个 VT_* 常量。

这两个值的组合提供了有关内部函数的完整信息集。

IDebugHostType2/IDebugHostType Bitfield 方法

以下 IDebugHostType 方法特定于在位字段中存储数据的类型。 有关内部函数中位域放置的信息作为数据模型中的类型符号的一部分存储，而不是作为位置的属性。

```
C++

STDMETHOD(GetBitField)(_Out_ ULONG* lsbOfField, _Out_ ULONG* lengthOfField)
PURE;
```

GetBitField

如果数据结构的给定成员是位字段 (例如：ULONG MyBits：8)，则字段的类型信息附带有关位域放置的信息。 GetBitField 方法可用于检索该信息。 此方法在非位域的任何类型上都将失败。 这是该方法失败的唯一原因。 只需调用此方法并查看成功/失败即可区分位字段和非位字段。 如果给定的类型恰好是位字段，则字段位置由半开集 *(lsbOfField + lengthOfField ： lsbOfField] 定义。*

IDebugHostType2/IDebugHostType 指针相关方法

以下 IDebugHostType 方法特定于指针类型。 此类类型为 GetTypeKind 返回 TypePointer 或 TypeMemberPointer'：

```
C++

STDMETHOD(GetPointerKind)(_Out_ PointerKind* pointerKind) PURE;
STDMETHOD(GetMemberType)(_Out_ IDebugHostType** memberType) PURE;
```

GetPointerKind

对于指针类型，GetPointerKind 方法返回指针的类型。 这由 PointerKind 枚举定义。

GetMemberType

对于由 TypeMemberPointer) 类型所指示的指针到成员 (的类型，GetMemberType 方法返回指针指向成员的类。

IDebugHostType2/IDebugHostType 数组相关方法

数组是 GetTypeKind 返回 TypeArray 的类型。 请注意，由调试主机的类型系统定义的数组与 C 使用的单维、基于零索引的打包线性一维数组不同。 C 样式数组适合定义，但在 IDebugHostType 中，数组的总体范围更广。 调试主机中的数组可以是多维的，数组中的每个维度都由名为 ArrayDimension 的描述符定义。此描述符具有以下字段：

字段	含义
LowerBound	数组的基索引，作为有符号 64 位值。 对于 C 样式数组，这始终为零。 它不需要。 可以将数组的单个维度视为从任何 64 位索引开始，即使是负索引。
Length	作为无符号 64 位值的数组维度的长度。 数组的索引跨半打开集 [LowerBound，LowerBound + Length) 。
跨距	定义数组维度的步长。 如果此维度的索引将一 (从 N 增加到 N + 1) ，则指示在内存中向前移动的字节数。 对于 C 样式数组，这是数组中每个元素的大小。 它不需要。 元素之间的填充可以表示为大于每个单个元素大小的步长。 对于多维数组，此值将指示如何向前移动整个维度。 考虑 M x N 矩阵。 这可以在行主窗体中描述为两个维度：

```
C++
```

```
{ [LowerBound: 0, Length: M, Stride: N \* sizeof(element)], [LowerBound: 0, Length: N, Stride: sizeof(element)]}
```

或者，也可以以列主形式将其描述为两个维度：

```
C++
```

```
{ [LowerBound: 0, Length: M, Stride: sizeof(element)], [LowerBound: 0, Length: N, Stride: M \* sizeof(element)]}
```

ArrayDimension 概念允许这种程度的灵活性。

以下 IDebugHostType 方法特定于数组类型。

```
C++
```

```
STDMETHOD(GetArrayDimensionality)(\_Out_ ULONG64\* arrayDimensionality)
PURE;
STDMETHOD(GetArrayDimensions)(\_In_ ULONG64 dimensions,
\_Out_writes_(dimensions) ArrayDimension \*pDimensions) PURE;
```

GetArrayDimensionality

GetArrayDimensionality 方法返回数组索引的维度数。 对于 C 样式数组，此处返回的值将始终为 1。

GetArrayDimensions

GetArrayDimensions 方法返回一组描述符，一个描述符用于数组的每个维度，如 GetArrayDimensionality 方法所示。 每个描述符都是一个 ArrayDimension 结构，用于描述每个数组维度的起始索引、长度和前进步幅。 这允许描述比 C 类型系统中允许的更强大的数组构造。

对于 C 样式数组，此处返回一个数组维度，其值始终为：

- LowerBound = 0
- Length = ARRAYSIZE (array)
- Stride = sizeof (elementType)

IDebugHostType2/IDebugHostType 函数相关方法

通过 TypeFunction 类型指示它们是函数类型的类型支持 IDebugHostType 和 IDebugHostType2 中的以下方法。

C++

```
//
// IDebugHostType:
//
STDMETHOD(GetFunctionCallingConvention)(_Out_ CallingConventionKind*
conventionKind) PURE;
STDMETHOD(GetFunctionReturnType)(_COM_Outptr_ IDebugHostType** returnType)
PURE;
STDMETHOD(GetFunctionParameterTypeCount)(_Out_ ULONG64* count) PURE;
STDMETHOD(GetFunctionParameterTypeAt)(_In_ ULONG64 i, _Out_ IDebugHostType**
parameterType) PURE;
//
// IDebugHostType2:
//
STDMETHOD(GetFunctionVarArgsKind)(_Out_ VarArgsKind* varArgsKind) PURE;
```

GetFunctionCallingConvention

GetFunctionCallingConvention 方法返回函数的调用约定。 此类作为 CallingConventionKind 枚举的成员返回。

GetFunctionReturnType

GetFunctionReturnType 方法返回函数的返回类型。

GetFunctionParameterTypeCount

GetFunctionParameterTypeCount 方法返回函数采用的参数数。 请注意，此计数中不考虑基于 C/C++ 省略号的变量参数标记。 必须通过 GetFunctionVarArgsKind 方法检测是否存在此类内容。 这将仅包括省略号前的参数。

GetFunctionParameterTypeAt

GetFunctionParameterTypeAt 方法将 i-th 参数的类型返回到函数。

GetFunctionVarArgsKind 方法返回给定函数是否利用变量参数列表，如果是，则返回它利用的变量参数样式。 此类由 VarArgsKind 枚举的成员定义，定义如下：

枚举	含义
VarArgsNone	函数不采用任何变量参数。
VarArgsCStyle	函数是一个 C 样式 varargs 函数，（returnType (arg1、 arg2、 ...) ）。函数报告的参数数不包括省略号参数。 任何变量参数传递都发生在 GetFunctionParameterTypeCount 方法返回的参数数之后。

IDebugHostType2 GetFunctionVarArgsKind

GetFunctionVarArgsKind 方法返回给定函数是否利用变量参数列表，如果是，则返回它利用的变量参数样式。 此类由 VarArgsKind 枚举的成员定义，定义如下：

IDebugHostType2/IDebugHostType Typedef 相关方法

作为 typedef 的任何类型的行为就像类型是 typedef 的基础的最终类型一样。 这意味着 GetTypeKind 等方法不会指示类型为 typedef。 同样，GetBaseType 不会返回定义引用的类型。 它们将指示其行为就像是在 typedef 基础的最终定义上调用的。 示例：

```
C++

typedef MYSTRUCT *PMYSTRUCT;
typedef PMYSTRUCT PTRMYSTRUCT;
```

PMYSTRUCT 或 PTRMYSTRUCT 的 IDebugHostType 将报告以下信息：

- GetTypeKind 方法将返回 TypePointer。 最终的基础类型 MYSTRUCT * 确实是指针。
- 'GetBaseType 方法将返回 MYSTRUCT 的类型。 MYSTRUCT * 的基础类型是 MYSTRUCT。

此处唯一的区别在于 IDebugHostType2 上的 typedef 特定方法的行为方式。 这些方法是：

```
C++

STDMETHOD(IsTypedef)(_Out_ bool* isTypedef) PURE;
STDMETHOD(GetTypedefBaseType)(_Out_ IDebugHostType2** baseType) PURE;
STDMETHOD(GetTypedefFinalBaseType)(_Out_ IDebugHostType2** finalBaseType)
PURE;
```

在此示例中：

- IsTypedef 方法将为 PMYSTRUCT 和 PTRMYSTRUCT 返回 true
- GetTypedefBaseType 方法将为 PMYSTRUCT 返回 MYSTRUCT *，为 PTRMYSTRUCT 返回 PMYSTRUCT
- GetTypedefFinalBaseType 方法将返回两种类型的 MYSTRUCT *

IsTypedef

IsTypedef 方法是唯一能够查看类型是否为 typedef 的方法。 GetTypeKind 方法的行为就像在基础类型上调用一样。

GetTypedefBaseType

GetTypedefBaseType 方法将返回 typedef 的直接定义。 在文档中描述的示例中：

```
C++

typedef MYSTRUCT *PMYSTRUCT;
typedef PMYSTRUCT PTRMYSTRUCT;
```

此方法将为 PMYSTRUCT 返回 MYSTRUCT *，为 PTRMYSTRUCT 返回 PMYSTRUCT。

GetTypedefFinalBaseType

GetTypedefFinalBaseType 方法将返回 typedef 为其定义的最终类型。 如果 typedef 是另一个 typedef 的定义，则它将继续遵循定义链，直到它到达一个不是 typedef 的类型，并且将返回该类型。 在文档中描述的示例中：

```
C++
```

```
typedef MYSTRUCT *PMYSTRUCT;
typedef PMYSTRUCT PTRMYSTRUCT;
```

在 PMYSTRUCT 或 PTRMYSTRUCT 上调用时，此方法将返回 MYSTRUCT *。

IDebugHostType2/IDebugHostType 类型创建方法

```
C++

STDMETHOD(CreatePointerTo)(_In_ PointerKind kind, _COM_Outptr_
IDebugHostType** newType) PURE;
STDMETHOD(CreateArrayOf)(_In_ ULONG64 dimensions, _In_reads_(dimensions)
ArrayDimension *pDimensions, _COM_Outptr_ IDebugHostType** newType) PURE;
```

常量符号值：IDebugHostConstant

对于常量值存在于符号信息中的位置，（特定值是符号，该符号可以是或可能不是）常量值，IDebugHostConstant 接口表示此类常量的概念。 这通常用于模板参数等位置，其中给定参数通常为类型，但可能改为非类型模板参数（例如：常量）。

IDebugHostConstant 接口定义如下，（忽略 IDebugHostSymbol）实现的泛型方法：

```
C++

DECLARE_INTERFACE_(IDebugHostConstant, IDebugHostSymbol)
{
    STDMETHOD(GetValue)(_Out_ VARIANT* value) PURE;
}
```

GetValue

GetValue 方法返回打包到 VARIANT 中的常量的值。 请务必注意，IDebugHostSymbol 上的 GetType 方法可能会返回常量的特定类型符号。 在这种情况下，无法保证由类型符号定义的常量值的打包与此处的 GetValue 方法返回的打包相同。

数据成员访问：IDebugHostField

IDebugHostField 类表示一个符号，该符号是类、结构、联合或其他类型构造的数据成员。 它不表示免费数据（例如：全局数据）。 接口的定义如下，（忽略 IDebugHostSymbol）的泛型方法：

```
C++

DECLARE_INTERFACE_(IDebugHostField, IDebugHostSymbol)
{
    STDMETHOD(GetLocationKind)(_Out_ LocationKind *locationKind) PURE;
```

```
    STDMETHOD(GetOffset)(_Out_ ULONG64* offset) PURE;
    STDMETHOD(GetLocation)(_Out_ Location* location) PURE;
    STDMETHOD(GetValue)(_Out_ VARIANT* value) PURE;
}
```

GetLocationKind

GetLocationKind 方法根据 LocationKind 枚举返回符号所在的位置类型。 此类枚举可以是以下值之一：

枚举	含义
LocationMember	字段是类、结构、联合或其他类型构造的常规数据成员。 它具有相对于包含类型构造的基址的偏移量。 此类基址通常由此指针表示。 可以通过 GetOffset 方法检索字段的偏移量。 对于 LocationMember 字段，GetLocation 和 GetValue 方法将失败。
LocationStatic	字段是静态的，并且具有自己的地址。 GetLocation 方法将返回抽象位置 (例如静态字段的地址) 。 对于 LocationStatic 字段，GetOffset 和 GetValue 方法将失败。
LocationConstant	字段是一个常量，并且具有值。 GetValue 方法将返回常量的值。 对于 LocationConstant 字段，GetOffset 和 GetLocation 方法将失败
LocationNone	字段没有位置。 它可能已被编译器优化，也可能是声明但从未定义的静态字段。 无论这种领域是如何出现，它都没有物理存在或价值。 它仅在符号中。 对于 LocationNone 字段， (GetOffset、GetLocation 和 GetValue) 的所有获取方法都将失败。

GetOffset

对于具有偏移 (的字段，例如位置类型指示 LocationMember) 的字段，GetOffset 方法将返回包含类型的基址的偏移量， (此指针) 字段本身的数据。 此类偏移量始终表示为无符号 64 位值。 如果给定字段没有与包含类型的基址偏移的位置，则 GetOffset 方法将失败。

GetLocation

对于具有地址的字段，而不考虑特定类型实例 (例如：其位置类型指示 LocationStatic) 的字段，GetLocation 方法将返回该字段的抽象位置 (地址) 。 如果给定字段没有静态位置，则 GetLocation 方法将失败。

GetValue

对于在符号信息中定义了常量值的字段 (例如，其位置类型指示 LocationConstant) 的字段，GetValue 方法将返回字段的常量值。 如果给定字段没有常量值，则 GetValue 方法将失败。

免费数据访问：IDebugHostData

不是其他类型的成员的模块中的数据由 IDebugHostData 接口表示。 该接口的定义如下 (忽略 IDebugHostSymbol) 的泛型方法：

```
C++

DECLARE_INTERFACE_(IDebugHostData, IDebugHostSymbol)
{
    STDMETHOD(GetLocationKind)(_Out_ LocationKind *locationKind) PURE;
    STDMETHOD(GetLocation)(_Out_ Location* location) PURE;
    STDMETHOD(GetValue)(_Out_ VARIANT* value) PURE;
}
```

所有这些方法在语义上都等效于 IDebugHostField 中的对应方法。 唯一的区别是 GetLocationKind 方法永远不会返回免费数据的 LocationMember。

GetLocationKind

GetLocationKind 方法根据 LocationKind 枚举返回符号所在的位置类型。 此枚举的说明可在 IDebugHostField 的文档中找到。

GetLocation

对于具有地址的数据，GetLocation 方法将返回字段的抽象位置 (地址) 。 如果给定的数据没有静态位置，则 GetLocation 方法将失败。

GetValue

对于在符号信息 (定义常量值的数据，例如，其位置类型指示 LocationConstant) 的数据，GetValue 方法将返回字段的常量值。 如果给定数据没有常量值，则 GetValue 方法将失败。

基类：IDebugHostBaseClass

给定类型的继承层次结构通过类型符号的子级表示。 如果给定类型从一个或多个类型) (继承派生，则类型符号将有一个或多个 SymbolBaseClass 子级。 其中每个 SymbolBaseClass 符号都表示特定类型的直接继承。 基类的名称既是 SymbolBaseClass 符号的名称，也是基类的类型符号的名称。 SymbolBaseClass 符号上的 GetType 方法可用于获取基类本身的类型符号。 可以通过递归浏览 SymbolBaseClass 子符号来遍历完整的继承层次结构。 每个基类符号都由 IDebugHostBaseClass 接口表示，该接口定义如下 (忽略 IDebugHostSymbol) 的泛型方法：

```
C++

DECLARE_INTERFACE_(IDebugHostBaseClass, IDebugHostSymbol)
{
```

```
    STDMETHOD(GetOffset)(_Out_ ULONG64* offset) PURE;
}
```

GetOffset

GetOffset 方法从派生类的基址返回基类的偏移量。 此类偏移量可以是零，也可以是正无符号 64 位值。

公共符号：IDebugHostPublic

公共符号表示符号文件中公共表中的内容。 实际上，它们是导出地址。 没有与公共符号关联的类型信息 -- 只有地址。 除非调用方显式请求公共符号，否则调试主机更愿意为每个查询返回私有符号。 公共符号由 IDebugHostPublic 接口表示，该接口定义如下 (忽略 IDebugHostSymbol) 通用方法：

C++

```
DECLARE_INTERFACE_(IDebugHostPublic, IDebugHostSymbol)
{
    STDMETHOD(GetLocationKind)(_Out_ LocationKind *locationKind) PURE;
    STDMETHOD(GetLocation)(_Out_ Location* location) PURE;
}
```

所有这些方法在语义上都等效于 IDebugHostField 中的对应方法。 唯一的区别是 GetLocationKind 方法永远不会为此类符号返回 LocationMember 或 LocationConstant。

GetLocationKind

GetLocationKind 方法根据 LocationKind 枚举返回符号所在的位置类型。 此枚举的说明可在 IDebugHostField 的文档中找到。

GetLocation

对于具有地址的数据，GetLocation 方法将返回字段的抽象位置 (地址) 。 如果给定的公共没有静态位置，则 GetLocation 方法将失败。

模块签名和版本匹配：IDebugHostModuleSignature

模块签名表示一种检查给定模块是否满足一组有关命名和版本控制的条件的方法。 模块签名是通过 IDebugHostSymbols 上的 CreateModuleSignature 方法创建的。 它可以匹配模块名称和模块的可选版本号范围。 创建此类签名后，客户端将收到一个 IDebugHostModuleSignature 接口，该接口定义如下：

C++

```
DECLARE_INTERFACE_(IDebugHostModuleSignature, IUnknown)
{
    STDMETHOD(IsMatch)(_In_ IDebugHostModule* pModule, _Out_ bool* isMatch)
PURE;
}
```

IsMatch

IsMatch 方法将 IDebugHostModule 符号 (的特定模块) 与签名进行比较，并将模块名称和版本与签名中指示的名称和版本范围进行比较。 返回给定模块符号是否与签名匹配的指示。

类型签名和类型匹配：IDebugHostTypeSignature

类型签名表示检查给定类型实例是否满足有关类型名称、类型泛型参数以及类型所在模块的一组条件的方法。 类型签名是通过 IDebugHostSymbols 上的 CreateTypeSignature 方法创建的。 创建此类签名后，客户端将收到一个 IDebugHostTypeSignature 接口，该接口定义如下：

C++

```
DECLARE_INTERFACE_(IDebugHostTypeSignature, IUnknown)
{
    STDMETHOD(GetHashCode)(_Out_ ULONG* hashCode) PURE;
    STDMETHOD(IsMatch)(_In_ IDebugHostType* type, _Out_ bool* isMatch,
_COM_Outptr_opt_ IDebugHostSymbolEnumerator** wildcardMatches) PURE;
    STDMETHOD(CompareAgainst)(_In_ IDebugHostTypeSignature* typeSignature,
_Out_ SignatureComparison* result) PURE;
}
```

GetHashCode

GetHashCode 方法返回类型签名的 32 位哈希代码。 调试主机保证为类型实例返回的哈希代码与为类型签名返回的哈希代码之间的实现同步。 除了全局匹配之外，如果类型实例能够匹配类型签名，则两者将具有相同的 32 位哈希代码。 这允许在类型实例与向数据模型管理器注册的过多类型签名之间进行初始快速比较和匹配。

IsMatch

IsMatch 方法返回特定类型实例是否与类型签名中指定的条件匹配的指示。 如果已返回，则返回此指示以及一个枚举器，该枚举器将指示类型实例的所有特定部分 (作为符号) 与类型签名中的通配符匹配。

CompareAgainst

CompareAgainst 方法将类型签名与其他类型签名进行比较，并返回两个签名的比较方式。 返回的比较结果是 SignatureComparison 枚举的成员，定义如下：

枚举	含义
Unrelated	比较的两个签名或类型之间没有关系。
不明确	一个签名或类型与另一个签名或类型进行比较不明确。 对于两个类型签名，这意味着存在可能匹配任一签名的可能类型实例。 例如，下面显示的两个类型签名不明确。 签名 1： `std::pair<*, int>` 签名 2： `std::pair<int,*>` 因为类型实例 `std::pair<int, int>` 匹配任意一个同样良好，（两者都有一个具体匹配项和一个通配符匹配）。
LessSpecific	一个签名或类型不如另一个签名或类型具体。 通常，这意味着不太具体的签名具有通配符，其中更具体的签名具有具体类型。 例如，下面的第一个签名不如第二个签名具体。 签名 1： `std::pair<*, int>` 签名 2： `std::pair<int, int>` 因为它具有通配符 (`*`)，其中第二个具有具体类型 (int)。
MoreSpecific	一个签名或类型比另一个签名或类型更具体。 通常，这意味着更具体的签名具有具体类型，其中不太具体的签名具有通配符。 例如，下面的第一个签名比第二个签名更具体。 签名 1： `std::pair<int, int>` 签名 2： `std::pair<*, int>` 因为它具有具体类型 (int) 其中第二个具有通配符 (`*`)。
相同	这两个签名或类型相同。

另请参阅

本主题是一系列文章的一部分，该系列介绍了可从 C++ 访问的接口、如何使用接口生成基于 C++ 的调试器扩展，以及如何利用其他数据模型构造 (例如：JavaScript 或 NatVis) 从 C++ 数据模型扩展。

调试器数据模型 C++ 概述

调试器数据模型 C++ 接口

调试器数据模型 C++ 对象

调试器数据模型 C++ 的其他接口

调试器数据模型 C++ 的概念

调试器数据模型 C++ 脚本

调试器数据模型 C++ 对象

项目 • 2024/05/27

本主题介绍如何使用调试器数据模型 C++ 对象，以及这些对象如何扩展调试器的功能。

核心调试器对象模型

数据模型最基本但最强大的功能之一是，它规范了什么是对象以及如何与对象交互的定义。 *IModelObject* 接口封装了对象的概念 - 无论该对象是整数、浮点数值、字符串、调试器目标地址空间中的某种复杂类型，还是某种调试器概念（如进程或模块的概念）。

IModelObject 中可以容纳（或装入）几种不同的东西：

- **固有值** - *IModelObject* 可以是许多基本类型的容器：8、16、32 或 64 位有符号或无符号整数、布尔值、字符串、错误或空概念。
- **本地对象** - *IModelObject* 可以表示调试器目标地址空间中的复杂类型（由调试器的类型系统定义）。
- **合成对象** - *IModelObject* 可以是一个动态对象 - 可以说是一个字典：键/值/元数据元组的集合和一组**概念**，这些概念定义的行为并非简单地用键/值对表示。
- **属性** - *IModelObject* 可以表示一个属性：其值可以通过方法调用来检索或更改。 *IModelObject* 中的属性实际上是一个 *IModelPropertyAccessor* 接口，被装箱到了一个 *IModelObject* 中
- **方法** - 一个 *IModelObject* 可以代表一种方法：可以用一组参数来调用它，并得到一个返回值。 *IModelObject* 中的方法实际上是一个 *IModelMethod* 接口，它被装箱到了一个 *IModelObject* 中

对象模型中的可扩展性

IModelObject 并不是一个孤立的对象。 除了代表上述对象类型之一外，每个对象还具有父数据模型链的概念。 此链的行为非常类似于 JavaScript 原型链。 与 JavaScript 的线性原型链不同，每个数据模型对象都定义了**父模型**的线性链。 这些父模型中的每一个又有自己的一组父模型的线性链。 就本质而言，每个对象都是自身和树中每个对象的能力（属性等）的聚合。 在查询特定属性时，如果被查询对象不支持该属性，则查询结果将以线性顺序依次传递给每个父对象。 这样就会产生一种行为，即通过对聚合树进行深度优先搜索来解决属性搜索问题。

鉴于每个对象都是自身和父模型树的聚合，这种对象模型的可扩展性非常简单。 扩展可以传入并将自己添加到另一个对象的父模型列表中。 执行此操作会**扩展**对象。 通过这种

方式，可以在任何东西上添加功能：对象或值的特定实例、本地类型、调试器对进程或线程的概念，甚至是“所有可迭代对象”的概念。

上下文、上下文和上下文：此指针、地址空间和实现专用数据

上下文有三个概念，有必要在对象模型的上下文中加以理解。

上下文：此指针

由于给定的属性或方法可以在数据模型树的任何层级实现，因此方法或属性的实现必须能够访问原始对象（在 C++ 中称为**此**指针，在 JavaScript 中称为**此**对象）。 在所述方法中，该实例对象将作为称为**上下文**的第一个参数传递给各种方法。

上下文：地址空间

需要注意的是，与之前的扩展模型不同，**上下文**（正在查看的目标、进程、线程）是一个用户界面概念，所有 API 都与当前的用户界面状态相关，而数据模型接口通常会显式或隐式地将此上下文作为 *IDebugHostContext* 接口。 数据模型中的每个 *IModelObject* 都会随附这类上下文信息，并可将该上下文传播给其返回的对象。 这意味着，在从 *IModelObject* 中读出一个本地值或键值时，它将从目标和进程中读出该对象最初的获取位置。

有一个显式常量值 **USE_CURRENT_HOST_CONTEXT** 可以传递给包含 *IDebugHostContext* 参数的方法。 该值表示上下文应该是调试器当前的 UI 状态。 但是，这一概念必须加以明确。

上下文：实现专用数据

请记住，数据模型中的每个对象实际上都是对象实例和所附父模型树的聚合。 每个父模型（可以在许多不同对象的链中链接）都可以将专用实现数据与任何实例对象相关联。 概念上创建的每个 *IModelObject* 都有一个哈希表，该表会将特定父模型映射到 *IUnknown* 接口定义的专用实例数据。 这样，父模型就可以缓存每个实例的信息，或者具有其他任意数据关联。

此类上下文可通过 *IModelObject* 上的 *GetContextForDataModel* 和 *SetContextForDataModel* 方法来访问。

核心调试器对象接口：IModelObject

IModelObject 接口定义如下：

C++

```
DECLARE_INTERFACE_(IModelObject, IUnknown)
{
    STDMETHOD(QueryInterface)(_In_ REFIID iid, _COM_Outptr_ PVOID* iface);
    STDMETHOD_(ULONG, AddRef)();
    STDMETHOD_(ULONG, Release)() PURE;
    STDMETHOD(GetContext)
(_COM_Outptr_result_maybenull_ IDebugHostContext** context) PURE;
    STDMETHOD(GetKind)(_Out_ ModelObjectKind *kind) PURE;
    STDMETHOD(GetIntrinsicValue)(_Out_ VARIANT* intrinsicData);
    STDMETHOD(GetIntrinsicValueAs)
(_In_ VARTYPE vt, _Out_ VARIANT* intrinsicData) PURE;
    STDMETHOD(GetKeyValue)
(_In_ PCWSTR key, _COM_Errorptr_opt_ IModelObject** object, _COM_Outptr_opt_
result_maybenull_ IKeyStore** metadata) PURE;
    STDMETHOD(SetKeyValue)
(_In_ PCWSTR key, _In_opt_ IModelObject* object) PURE;
    STDMETHOD(EnumerateKeyValues)
(_COM_Outptr_ IKeyEnumerator** enumerator) PURE;
    STDMETHOD(GetRawValue)
(_In_ SymbolKind kind, _In_ PCWSTR name, _In_ ULONG searchFlags, _COM_Errorp
tr_ IModelObject** object) PURE;
    STDMETHOD(EnumerateRawValues)
(_In_ SymbolKind kind, _In_ ULONG searchFlags, _COM_Outptr_ IRawEnumerator**
 enumerator) PURE;
    STDMETHOD(Dereference)(_COM_Errorptr_ IModelObject** object) PURE;
    STDMETHOD(TryCastToRuntimeType)
(_COM_Errorptr_ IModelObject** runtimeTypedObject) PURE;
    STDMETHOD(GetConcept)
(_In_ REFIID conceptId, _COM_Outptr_ IUnknown** conceptInterface, _COM_Outpt
r_opt_result_maybenull_ IKeyStore** conceptMetadata) PURE;
    STDMETHOD(GetLocation)(_Out_ Location* location) PURE;
    STDMETHOD(GetTypeInfo)(_Out_ IDebugHostType** type) PURE;
    STDMETHOD(GetTargetInfo)
(_Out_ Location* location, _Out_ IDebugHostType** type) PURE;
    STDMETHOD(GetNumberOfParentModels)(_Out_ ULONG64* numModels) PURE;
    STDMETHOD(GetParentModel)
(_In_ ULONG64 i, _COM_Outptr_ IModelObject **model, _COM_Outptr_result_maybe
null_ IModelObject **contextObject) PURE;
    STDMETHOD(AddParentModel)
(_In_ IModelObject* model, _In_opt_ IModelObject* contextObject, _In_ bool o
verride) PURE;
    STDMETHOD(RemoveParentModel)(_In_ IModelObject* model) PURE;
    STDMETHOD(GetKey)
(_In_ PCWSTR key, _COM_Errorptr_opt_ IModelObject** object, _COM_Outptr_opt_
result_maybenull_ IKeyStore** metadata) PURE;
    STDMETHOD(GetKeyReference)
(_In_ PCWSTR key, _COM_Errorptr_opt_ IModelObject** objectReference, _COM_Ou
tptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
    STDMETHOD(SetKey)
(_In_ PCWSTR key, _In_opt_ IModelObject* object, _In_opt_ IKeyStore* metadat
```

```
a) PURE;
    STDMETHOD(ClearKeys)() PURE;
    STDMETHOD(EnumerateKeys)(_COM_Outptr_ IKeyEnumerator** enumerator) PURE;
    STDMETHOD(EnumerateKeyReferences)
(_COM_Outptr_ IKeyEnumerator** enumerator) PURE;
    STDMETHOD(SetConcept)
(_In_ REFIID conceptId, _In_ IUnknown* conceptInterface, _In_opt_ IKeyStore*
 conceptMetadata) PURE;
    STDMETHOD(ClearConcepts)() PURE;
    STDMETHOD(GetRawReference)
(_In_ SymbolKind kind, _In_ PCWSTR name, _In_ ULONG searchFlags, _COM_Errorp
tr_ IModelObject** object) PURE;
    STDMETHOD(EnumerateRawReferences)
(_In_ SymbolKind kind, _In_ ULONG searchFlags, _COM_Outptr_ IRawEnumerator**
 enumerator) PURE;
    STDMETHOD(SetContextForDataModel)
(_In_ IModelObject* dataModelObject, _In_ IUnknown* context) PURE;
    STDMETHOD(GetContextForDataModel)
(_In_ IModelObject* dataModelObject, _Out_ IUnknown** context) PURE;
    STDMETHOD(Compare)
(_In_ IModelObject* other, _COM_Outptr_opt_result_maybenull_ IModelObject **
ppResult) PURE;
    STDMETHOD(IsEqualTo)(_In_ IModelObject* other, _Out_ bool* equal) PURE;
}
```

基本方法

以下是适用于 IModelObject 所表示的任何类型对象的一般方法。

```
C++

STDMETHOD(GetKind)(_Out_ ModelObjectKind *kind) PURE;
STDMETHOD(GetContext)(_COM_Outptr_result_maybenull_ IDebugHostContext**
context) PURE;
STDMETHOD(GetIntrinsicValue)(_Out_ VARIANT* intrinsicData);
STDMETHOD(GetIntrinsicValueAs)(_In_ VARTYPE vt, _Out_ VARIANT*
intrinsicData) PURE;
STDMETHOD(Compare)(_In_ IModelObject* other,
_COM_Outptr_opt_result_maybenull_ IModelObject **ppResult) PURE;
STDMETHOD(IsEqualTo)(_In_ IModelObject* other, _Out_ bool* equal) PURE;
STDMETHOD(Dereference)(_COM_Errorptr_ IModelObject** object) PURE;
```

GetKind

GetKind 方法会返回 IModelObject 内部装箱的对象类型。

GetContext

GetContext 方法会返回与对象相关联的主机上下文。

GetIntrinsicValue

GetIntrinsicValue 方法会返回 IModelObject 内装箱的东西。 此方法只能在 IModelObject 接口上合法调用，而这些接口表示装箱的固有接口或特定的装箱接口。 它不能在本地对象、无值对象、合成对象和引用对象上调用。 GetIntrinsicValueAs 方法的行为与 GetIntrinsicValue 方法相似，只是它会将值转换为指定的变体类型。 如果无法进行转换，此方法将返回错误。

IsEqualTo

IsEqualTo 方法会比较两个模型对象，并返回它们的值是否相等的信息。 对于排序的对象，此方法返回 true 相当于 Compare 方法返回 0。 对于没有排序但相等的对象，Compare 方法会失败，但此方法不会失败。 基于值的比较的含义由对象类型定义。 目前，这只针对固有类型和错误对象。 目前还没有关于均等性的数据模型概念。

Dereference

Dereference 方法会取消引用一个对象。 此方法可用于取消引用基于数据模型的引用（ObjectTargetObjectReference、ObjectKeyReference）或本地语言引用（指针或语言引用）。 需要注意的是，此方法只会删除对象的单层引用语义。 例如，完全可以将数据模型引用到语言引用中。 在这种情况下，首次调用 Dereference 方法将删除数据模型引用而留下语言引用。 随后，在生成的对象上调用 Dereference 会移除语言引用，同时返回该引用下的本地值。

键操作方法

任何合成对象（键、值和元数据元组的字典）都有一系列方法来操作这些键、值以及与之关联的元数据。

基于值的 API 形式包括：

```
C++
```

```
STDMETHOD(GetKeyValue)(_In_ PCWSTR key, _COM_Errorptr_opt_ IModelObject**
object, _COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
STDMETHOD(SetKeyValue)(_In_ PCWSTR key, _In_opt_ IModelObject* object) PURE;
STDMETHOD(EnumerateKeyValues)(_COM_Outptr_ IKeyEnumerator** enumerator)
PURE;
The key based forms of the APIs (including those used for key creation) are:
STDMETHOD(GetKey)(_In_ PCWSTR key, _COM_Errorptr_opt_ IModelObject** object,
_COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
STDMETHOD(SetKey)(_In_ PCWSTR key, _In_opt_ IModelObject* object, _In_opt_
IKeyStore* metadata) PURE;
STDMETHOD(EnumerateKeys)(_COM_Outptr_ IKeyEnumerator** enumerator) PURE;
STDMETHOD(ClearKeys)() PURE;
```

基于引用的 API 形式包括：

```
C++
```

```
STDMETHOD(GetKeyReference)(_In_ PCWSTR key, _COM_Errorptr_opt_
IModelObject** objectReference, _COM_Outptr_opt_result_maybenull_
IKeyStore** metadata) PURE;
STDMETHOD(EnumerateKeyReferences)(_COM_Outptr_ IKeyEnumerator** enumerator)
PURE;
```

GetKeyValue

GetKeyValue 方法是客户端通过名称获取给定键值（以及与之关联的元数据）的第一种方法。 如果键是一个属性访问器 - 即它的值是一个 IModelObject（它是一个装箱的 IModelPropertyAccessor），则 GetKeyValue 方法将自动调用属性访问器的 GetValue 方法以检索实际值。

SetKeyValue

SetKeyValue 方法是客户端用于设置键值的第一种方法。 此方法不能用于在对象上创建新键。 它只会设置现有键的值。 请注意，很多键均为只读（例如：它们由属性访问器实现，该访问器的 SetValue 方法会返回 E_NOT_IMPL）。 在只读键上调用此方法时会失败。

EnumerateKeyValues

EnumerateKeyValues 方法是客户端要枚举对象上所有键值（包括父模型树中任何地方实现的所有键）时使用的第一种方法。 值得注意的是，EnumerateKeyValues 将枚举对象树中由重复名称定义的任何键；但是，GetKeyValue 和 SetKeyValue 等方法只能操作深度优先遍历发现的具有给定名称的键的第一个实例。

GetKey

GetKey 方法将按名称获取给定键的值（以及与之关联的元数据）。 大多数客户端都应改为使用 GetKeyValue 方法。 如果键是一个属性访问器，则调用此方法将返回装箱到 IModelObject 中的属性访问器（IModelPropertyAccessor 接口）。 与 GetKeyValue 不同，此方法不会通过调用 GetValue 方法来自动解析键的基础值。 这一责任由调用方承担。

SetKey

SetKey 方法是客户端在对象上创建键（并可能将元数据与创建的键相关联）时使用的方法。 如果给定对象已经有了一个名称相同的键，则会出现两种行为之一。 如果键位于 this 给出的实例中，则该键的值将被替换，就像原来的键不存在一样。 另一方面，如果该键在 this 给出的实例的父数据模型链中，则会在 this 给出的实例上创建一个带有给定名称的新键。 实际上，这将导致对象有两个同名键（类似于派生类对基类同名成员的阴影）。

EnumerateKeys

EnumerateKeys 方法的行为与 EnumerateKeyValues 方法类似，只是它不会自动解析对象上的属性访问器。 这意味着，如果键的值是一个属性访问器，那么 EnumerateKeys 方法将返回装箱到 IModelObject 中的属性访问器（一个 IModelPropertyAccessorInterface），而不是自动调用 GetValue 方法。 这类似于 GetKey 和 GetKeyValue 之间的区别。

ClearKeys

ClearKeys 方法会从 this 指定的对象实例中删除所有键及其关联的值和元数据。 此方法对附加到特定对象实例的父模型没有影响。

GetKeyReference

GetKeyReference 方法将在对象（或其父模型链）上搜索给定名称的键，并返回通过装箱到 IModelObject 中的 IModelKeyReference 接口给出的该键的引用。 该引用随后可用于获取或设置键值。

EnumerateKeyReferences

EnumerateKeyReferences 方法的行为类似于 EnumerateKeyValues 方法，不同之处在于它返回的是对所枚举的键（通过装箱到 IModelObject 中的 IModelKeyReference 接口给出）的引用，而不是键值。 此类引用可用于获取或设置键的基础值。

概念操作方法

模型对象不仅是键/值/元数据元组的字典，也是概念的容器。 概念是一种抽象的东西，可以在物体上或通过物体来实现。 从本质上讲，概念是对象支持的接口的动态存储。 如今，数据模型定义了许多概念：

展开表

概念接口	说明
IDataModelConcept	概念是一个父模型。 如果此模型通过注册的类型签名自动附加到本地类型，那么每次实例化该类型的新对象时，InitializeObject 方法都会被自动调用。
IStringDisplayableConcept	对象可转换为字符串以供显示。
IIterableConcept	对象是一个容器并可以迭代。
IIndexableConcept	对象是一个容器，可以在一个或多个维度上进行索引（通过随机存取进行访问）。
IPreferredRuntimeTypeConcept	对象对派生类型的理解比基础类型系统所能提供的要多，因此希望自己处理从静态类型到运行时类型的转换。

概念接口	说明
IDynamicKeyProviderConcept	该对象是键的动态提供程序，希望接管核心数据模型的所有键查询。 此接口通常用作连接 JavaScript 等动态语言的桥梁。
IDynamicConceptProviderConcept	该对象是概念的动态提供程序，希望接管核心数据模型的所有概念查询。 此接口通常用作连接 JavaScript 等动态语言的桥梁。

IModelObject 上的以下方法可用于操作对象所支持的概念。

```
C++

STDMETHOD(GetConcept)(_In_ REFIID conceptId, _COM_Outptr_ IUnknown**
conceptInterface, _COM_Outptr_opt_result_maybenull_ IKeyStore**
conceptMetadata) PURE;
STDMETHOD(SetConcept)(_In_ REFIID conceptId, _In_ IUnknown*
conceptInterface, _In_opt_ IKeyStore* conceptMetadata) PURE;
STDMETHOD(ClearConcepts)() PURE;
```

GetConcept

GetConcept 方法将在对象（或其父模型链）上搜索概念，并返回指向概念接口的接口指针。 概念接口上的行为和方法特定于每个概念。 但需要注意的是，许多概念接口都要求调用方显式传递上下文对象（或传统意义上的 this 指针）。 确保向每个概念接口传递正确的上下文对象非常重要。

SetConcept

SetConcept 方法将在 this 指针指定的对象实例上放置一个指定的概念。 如果此参数指定的对象实例的父模型也支持此概念，则实例中的实现将覆盖父模型中的实现。

ClearConcepts

ClearConcepts 方法将从 this 指定的对象实例中删除所有概念。

本地对象方法

虽然许多模型对象引用的固有构造（如整数、字符串）或合成构造（键/值/元数据元组和概念的字典），但模型对象也可能引用本地构造（如调试目标地址空间中的用户定义类型）。 IModelObject 接口上有一系列方法可以访问此类本地对象的信息。 这些方法包括：

```
C++

STDMETHOD(GetRawValue)(_In_ SymbolKind kind, _In_ PCWSTR name, _In_ ULONG
searchFlags, _COM_Errorptr_ IModelObject** object) PURE;
```

```
STDMETHOD(EnumerateRawValues)(_In_ SymbolKind kind, _In_ ULONG searchFlags,
_COM_Outptr_ IRawEnumerator** enumerator) PURE;
STDMETHOD(TryCastToRuntimeType)(_COM_Errorptr_ IModelObject**
runtimeTypedObject) PURE;
STDMETHOD(GetLocation)(_Out_ Location* location) PURE;
STDMETHOD(GetTypeInfo)(_Out_ IDebugHostType** type) PURE;
STDMETHOD(GetTargetInfo)(_Out_ Location* location, _Out_ IDebugHostType**
type) PURE;
STDMETHOD(GetRawReference)(_In_ SymbolKind kind, _In_ PCWSTR name, _In_
ULONG searchFlags, _COM_Errorptr_ IModelObject** object) PURE;
STDMETHOD(EnumerateRawReferences)(_In_ SymbolKind kind, _In_ ULONG
searchFlags, _COM_Outptr_ IRawEnumerator** enumerator) PURE;
```

GetRawValue

GetRawValue 方法会在给定对象中查找本地构造。 此类构造可以是一个字段、一个基类、基类中的一个字段、一个成员函数等...

EnumerateRawValues

EnumerateRawValues 方法枚举给定对象的所有本地子类（例如：字段、基类等...）。

TryCastToRuntimeType

TryCastToRuntimeType 方法将要求调试主机执行分析，并确定给定对象的实际运行时类型（例如：大多数派生类）。 具体的分析方法取决于调试主机，可能包括 RTTI（C++ 运行时类型信息）、检查对象的 V-Table（虚拟函数表）结构，或主机可以用来从静态类型可靠地确定动态/运行时类型的任何其他方法。 无法转换为运行时类型并不意味着此方法调用将失败。 在这种情况下，该方法将在输出参数中返回给定对象（this 指针）。

GetLocation

GetLocation 方法将返回本地对象的位置。 虽然此类位置通常是调试目标地址空间内的虚拟地址，但也不一定如此。 此方法返回的位置是一个抽象位置，可能是虚拟地址，可能表示寄存器或子寄存器内的位置，也可能表示调试主机定义的其他任意地址空间。 如果生成的 Location 对象的 HostDefined 字段为 0，则表明该位置实际上是一个虚拟地址。 可以通过检查结果位置的 Offset 字段来检索此类虚拟地址。 HostDefined 字段的任何非零值都表示一个备用地址空间，其中 Offset 字段是该地址空间内的偏移量。 这里的 HostDefined 非零值的确切含义是调试主机的专用值。

GetTypeInfo

GetTypeInfo 方法将返回给定对象的本地类型。 如果对象没有与之关联的本地类型信息（例如：它是一个固有类型等），调用仍将成功，但将返回空值。

GetTargetInfo

GetTargetInfo 方法实际上是 GetLocation 和 GetTypeInfo 方法的组合，同时返回给定对象的抽象位置和本地类型。

GetRawReference

GetRawReference 方法在给定对象中查找本地构造，并返回该本地构造的引用。 此类构造可以是一个字段、一个基类、基类中的一个字段、一个成员函数等。重要的是要将此处返回的引用（ObjectTargetObjectReference 类型的对象）与语言引用（例如：C++ & 或 && 风格的引用）区分开来。

EnumerateRawReferences

EnumerateRawReferences 方法枚举给定对象的所有本地子类（例如：字段、基类等...）的引用。

可扩展性方法

如前所述，模型对象的行为与 JavaScript 对象及其原型链非常相似。 除了某个 IModelObject 接口所代表的实例外，该对象还可能有任意数量的父模型（每个父模型又可能有任意数量的父模型）。 这是数据模型内可扩展性的主要手段。 如果给定的属性或概念无法在给定的实例中找到，就会对以实例为根的对象树（由父模型定义）进行深度优先搜索。

以下方法可操作与给定 IModelObject 实例相关的父模型链：

C++

```
STDMETHOD(GetNumberOfParentModels)(_Out_ ULONG64* numModels) PURE;
STDMETHOD(GetParentModel)(_In_ ULONG64 i, _COM_Outptr_ IModelObject **model,
_COM_Outptr_result_maybenull_ IModelObject **contextObject) PURE;
STDMETHOD(AddParentModel)(_In_ IModelObject* model, _In_opt_ IModelObject*
contextObject, _In_ bool override) PURE;
STDMETHOD(RemoveParentModel)(_In_ IModelObject* model) PURE;
STDMETHOD(SetContextForDataModel)(_In_ IModelObject* dataModelObject, _In_
IUnknown* context) PURE;
STDMETHOD(GetContextForDataModel)(_In_ IModelObject* dataModelObject, _Out_
IUnknown** context) PURE;
```

GetNumberOfParentModels

GetNumberOfParentModels 方法返回附加到给定对象实例的父模型的数量。 父模型按父模型链的线性排序深度优先搜索属性。

GetParentModel

GetParentModel 方法返回给定对象父模型链中的第 i 个父模型。 父模型按添加或枚举的线性顺序来搜索属性或概念。 在索引为 i + 1 的父模型之前，先搜索索引为 0 的父模型

（分层）。

AddParentModel

AddParentModel 方法可为给定对象添加一个新的父模型。 此类模型可以添加到搜索链的末尾（覆盖参数指定为 false）或搜索链的前端（覆盖参数指定为 true）。 此外，每个父模型可以选择性地调整给定父模型（或父模型层次结构中的任何模型）上任何属性或概念的上下文（this 指针的语义）。 上下文调整很少使用，但它允许使用一些强大的概念，如对象嵌入、构建命名空间等...

RemoveParentModel

RemoveParentModel 将从给定对象的父搜索链中删除指定的父模型。

SetContextForDataModel

数据模型的实现使用 SetContextForDataModel 方法在实例对象上放置实现数据。 就概念而言，每个 IModelObject（为简单起见，称其为实例）都包含一个状态哈希表。 哈希表由实例的父模型层次结构中的另一个 IModelObject（为简单起见，称其为数据模型）索引。 该哈希中包含的值是由 IUnknown 实例表示的一组引用计数状态信息。 一旦数据模型在实例上设置了这种状态，它就可以存储任意的执行数据，而这些数据可以在属性获取器等程序中进行检索。

GetContextForDataModel

GetContextForDataModel 方法用于检索之前调用 SetContextForDataModel 时设置的上下文信息。 这将检索由实例对象父模型层次结构中更高一级的数据模型在实例对象上设置的状态信息。 有关此上下文/状态及其含义的详细信息，请参阅 SetContextForDataModel 的文档。

调试器数据模型核心对象类型

数据模型中的对象类似于 .NET 中的 Object 概念。 它是一个通用容器，数据模型所能理解的构造都可以装入其中。 除了本地对象和合成（动态）对象外，还有一系列核心对象类型可以放入（或装箱）到 IModelObject 的容器中。 放置这些值的容器是一个标准的 COM/OLE 变量，并对该变量所能包含的内容做了一些额外的限制。 其中最基本的类型包括：

- 8 位无符号值和有符号值 (VT_UI1, VT_I1)
- 16 位无符号值和有符号值 (VT_UI2, VT_UI2)
- 32 位无符号值和有符号值 (VT_UI4, VT_I4)
- 64 位无符号值和有符号值 (VT_UI8, VT_I8)
- 单精度和双精度浮点值 (VT_R4, VT_R8)
- 字符串 (VT_BSTR)

- 布尔值 (VT_BOOL)

除了这些基本类型外，一些核心数据模型对象被放入由 VT_UNKNOWN 定义的 IModelObject 中，其中存储的 IUnknown 可以保证实现特定的接口。 这些类型包括：

- 属性访问器 (IModelPropertyAccessor)
- 方法对象 (IModelMethod)
- 键引用对象（IModelKeyReference 或 IModelKeyReference2）
- 上下文对象 (IDebugModelHostContext)

属性访问器：*IModelPropertyAccessor*

数据模型中的属性访问器是 IModelPropertyAccessor 接口的实现，它会被装箱到 IModelObject 中。 在查询时，模型对象将返回一种 ObjectPropertyAccessor，其固有值是保证 IModelPropertyAccessor 可以查询的 VT_UNKNOWN。 在此过程中，它可以保证静态转换为 IModelPropertyAccessor。

属性访问器是一种间接的方法，用于获取和设置数据模型中的键值。 如果给定键的值是一个属性访问器，GetKeyValue 和 SetKeyValue 方法会自动注意到这一点，并根据情况调用属性访问器的基础 GetValue 或 SetValue 方法。

IModelPropertyAccessor 接口定义如下：

C++

```
DECLARE_INTERFACE_(IModelPropertyAccessor, IUnknown)
{
    STDMETHOD(GetValue)(_In_ PCWSTR key, _In_opt_ IModelObject*
contextObject, _COM_Outptr_ IModelObject** value) PURE;
    STDMETHOD(SetValue)(_In_ PCWSTR key, _In_opt_ IModelObject*
contextObject, _In_ IModelObject* value) PURE;
}
```

GetValue

GetValue 方法是属性访问器的 getter。 每当客户端希望获取属性的基础值时，就会调用它。 请注意，任何直接获取属性访问器的调用方都有责任将键名和准确的实例对象（this 指针）传递给属性访问器的 GetValue 方法。

SetValue

SetValue 方法是属性访问器的 setter。 每当客户端希望为基础属性赋值时，就会调用它。 很多属性都是只读的。 在这种情况下，调用 SetValue 方法将返回 E_NOTIMPL。 请注意，任何直接获取属性访问器的调用方都有责任将键名和准确的实例对象（this 指针）传递给属性访问器的 SetValue 方法。

方法：*IModelMethod*

数据模型中的方法是 IModelMethod 接口的实现，它会被装箱到 IModelObject 中。 在查询时，模型对象将返回一种 ObjectMethod，其固有值是保证 IModelMethod 可以查询的 VT_UNKNOWN。 在此过程中，它可以保证静态转换为 IModelMethod。 数据模型中的所有方法本质上都是动态的。 它们会将一组 0 个或多个参数作为输入，并返回单个输出值。 没有重载解析，也没有关于参数名称、类型或期望值的元数据。

IModelMethod 接口定义如下：

C++

```
DECLARE_INTERFACE_(IModelMethod, IUnknown)
{
    STDMETHOD(Call)(_In_opt_ IModelObject *pContextObject, _In_ ULONG64
argCount, _In_reads_(argCount) IModelObject **ppArguments, _COM_Errorptr_
IModelObject **ppResult, _COM_Outptr_opt_result_maybenull_ IKeyStore
**ppMetadata) PURE;
}
```

调用

Call 方法是调用数据模型中定义的任何方法的方式。 调用方负责传递一个准确的实例对象（this 指针）和一组任意参数。 方法的结果以及与该结果相关的任何可选元数据都会返回。 逻辑上不返回值的方法仍必须返回一个有效的 IModelObject。 在这种情况下，IModelObject 是一个装箱的“no value”。 如果方法失败，它可以在输入参数中返回可选的扩展错误信息（即使返回的 HRESULT 失败）。 调用方必须检查这一点。

键引用：*IModelKeyReference 或 IModelKeyReference2*

键引用实质上是特定对象上一个键的句柄。 客户端可以通过 GetKeyReference 等方法来获取此类句柄，并在以后使用该句柄来获取或设置键值，而不必保留原始对象。 这种类型的对象是 IModelKeyReference 或 IModelKeyReference2 接口的实现，它被装箱到了 IModelObject 中。 在查询时，模型对象将返回一种 ObjectKeyReference，其固有值是保证 IModelKeyReference 可以查询的 VT_UNKNOWN。 在此过程中，它可以保证静态转换为 IModelKeyReference。

键引用接口定义如下：

C++

```
DECLARE_INTERFACE_(IModelKeyReference2, IModelKeyReference)
{
    STDMETHOD(GetKeyName)(_Out_ BSTR* keyName) PURE;
    STDMETHOD(GetOriginalObject)(_COM_Outptr_ IModelObject** originalObject)
PURE;
    STDMETHOD(GetContextObject)(_COM_Outptr_ IModelObject**
```

```
containingObject) PURE;
    STDMETHOD(GetKey)(_COM_Errorptr_opt_ IModelObject** object,
_COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
    STDMETHOD(GetKeyValue)(_COM_Errorptr_opt_ IModelObject** object,
_COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
    STDMETHOD(SetKey)(_In_opt_ IModelObject* object, _In_opt_ IKeyStore*
metadata) PURE;
    STDMETHOD(SetKeyValue)(_In_ IModelObject* object) PURE;
    STDMETHOD(OverrideContextObject)(_In_ IModelObject* newContextObject)
PURE;
}
```

GetKeyName

GetKeyName 方法会返回键引用作为句柄的键的名称。 返回的字符串是标准 BSTR，必须通过调用 SysFreeString 释放。

GetOriginalObject

GetOriginalObject 方法返回创建键引用的实例对象。 请注意，该键本身可能位于实例对象的父模型上。

GetContextObject

GetContextObject 方法返回上下文（this 指针），如果相关键值指向一个属性访问器，该上下文将被传递给属性访问器的 GetValue 或 SetValue 方法。 此处返回的上下文对象可能与从 GetOriginalObject 获取的原始对象相同，也可能不相同。 如果键位于父模型上，并且有一个上下文调整器与该父模型相关联，那么原始对象就是调用 GetKeyReference 或 EnumerateKeyReferences 的实例对象。 上下文对象将是原始对象和父模型之间的最终上下文调整器所产生的任何对象，父模型包含键，而对该键的引用是句柄。 如果没有上下文调整器，则原始对象和上下文对象完全相同。

GetKey

键引用上的 GetKey 方法与 IModelObject 上的 GetKey 方法行为相同。 它返回基础键的值以及与该键关联的任何元数据。 如果键值恰好是一个属性访问器，则将返回装箱到 IModelObject 中的属性访问器 (IModelPropertyAccessor)。 此方法不会调用属性访问器上的基础 GetValue 或 SetValue 方法。

GetKeyValue

键引用上的 GetKeyValue 方法与 IModelObject 上的 GetKeyValue 方法行为相同。 它返回基础键的值以及与该键关联的任何元数据。 如果键值恰好是一个属性访问器，则会自动调用属性访问器的基础 GetValue 方法。

SetKey

键引用上的 SetKey 方法与 IModelObject 上的 SetKey 方法行为相同。 它将分配键值。 如果原始键是一个属性访问器，则会取代属性访问器。 它不会调用属性访问器上的 SetValue 方法。

SetKeyValue

键引用上的 SetKeyValue 方法与 IModelObject 上的 SetKeyValue 方法行为相同。 它将分配键值。 如果原始键是一个属性访问器，则会调用属性访问器的基础 SetValue 方法，而不是替换属性访问器本身。

OverrideContextObject

OverrideContextObject 方法（仅存在于 IModelKeyReference2 中）是一种高级方法，用于永久更改上下文对象，而此键引用将把该上下文对象传递给任何基础属性访问器的 GetValue 或 SetValue 方法。 调用 GetContextObject 时也会返回传递给此方法的对象。 脚本提供程序可使用此方法来复制某些动态语言行为。 大多数客户端不应调用此方法。

上下文对象：*IDebugHostContext*

上下文对象是调试主机（与数据模型合作）与每个对象关联的不透明信息 blob。 它可能包括进程上下文或信息来自的地址空间等。上下文对象是 IDebugHostContext 在 IModelObject 中的实现。 请注意，IDebugHostContext 是主机定义的接口。 客户端永远不会实现这个接口。

有关上下文对象的详细信息，请参阅“调试器数据模型 C++ 接口”中的调试器数据模型 C++ 主机接口。

数据模型管理器

数据模型管理器 IDataModelManager2（或之前的 IDataModelManager）的核心接口定义如下：

C++

```
DECLARE_INTERFACE_(IDataModelManager2, IDataModelManager)
{
    //
    // IDataModelManager:
    //
    STDMETHOD(Close)() PURE;
    STDMETHOD(CreateNoValue)(_Out_ IModelObject** object) PURE;
    STDMETHOD(CreateErrorObject)(_In_ HRESULT hrError, _In_opt_ PCWSTR
pwszMessage, _COM_Outptr_ IModelObject** object) PURE;
    STDMETHOD(CreateTypedObject)(_In_opt_ IDebugHostContext* context, _In_
Location objectLocation, _In_ IDebugHostType* objectType, _COM_Errorptr_
IModelObject** object) PURE;
    STDMETHOD(CreateTypedObjectReference)(_In_opt_ IDebugHostContext*
```

```
context, _In_ Location objectLocation, _In_ IDebugHostType* objectType,
_COM_Errorptr_ IModelObject** object) PURE;
    STDMETHOD(CreateSyntheticObject)(_In_opt_ IDebugHostContext* context,
_COM_Outptr_ IModelObject** object) PURE;
    STDMETHOD(CreateDataModelObject)(_In_ IDataModelConcept* dataModel,
_COM_Outptr_ IModelObject** object) PURE;
    STDMETHOD(CreateIntrinsicObject)(_In_ ModelObjectKind objectKind, _In_
VARIANT* intrinsicData, _COM_Outptr_ IModelObject** object) PURE;
    STDMETHOD(CreateTypedIntrinsicObject)(_In_ VARIANT* intrinsicData, _In_
IDebugHostType* type, _COM_Outptr_ IModelObject** object) PURE;
    STDMETHOD(GetModelForTypeSignature)(_In_ IDebugHostTypeSignature*
typeSignature, _Out_ IModelObject** dataModel) PURE;
    STDMETHOD(GetModelForType)(_In_ IDebugHostType* type, _COM_Outptr_
IModelObject** dataModel, _COM_Outptr_opt_ IDebugHostTypeSignature**
typeSignature, _COM_Outptr_opt_ IDebugHostSymbolEnumerator**
wildcardMatches) PURE;
    STDMETHOD(RegisterModelForTypeSignature)(_In_ IDebugHostTypeSignature*
typeSignature, _In_ IModelObject* dataModel) PURE;
    STDMETHOD(UnregisterModelForTypeSignature)(_In_ IModelObject* dataModel,
_In_opt_ IDebugHostTypeSignature* typeSignature) PURE;
    STDMETHOD(RegisterExtensionForTypeSignature)(_In_
IDebugHostTypeSignature* typeSignature, _In_ IModelObject* dataModel) PURE;
    STDMETHOD(UnregisterExtensionForTypeSignature)(_In_ IModelObject*
dataModel, _In_opt_ IDebugHostTypeSignature* typeSignature) PURE;
    STDMETHOD(CreateMetadataStore)(_In_opt_ IKeyStore* parentStore,
_COM_Outptr_ IKeyStore** metadataStore) PURE;
    STDMETHOD(GetRootNamespace)(_COM_Outptr_ IModelObject** rootNamespace)
PURE;
    STDMETHOD(RegisterNamedModel)(_In_ PCWSTR modelName, _In_ IModelObject
*modeObject) PURE;
    STDMETHOD(UnregisterNamedModel)(_In_ PCWSTR modelName) PURE;
    STDMETHOD(AcquireNamedModel)(_In_ PCWSTR modelName, _COM_Outptr_
IModelObject **modelObject) PURE;
    //
    // IDataModelManager2:
    //
    STDMETHOD(AcquireSubNamespace)(_In_ PCWSTR modelName, _In_ PCWSTR
subNamespaceModelName, _In_ PCWSTR accessName, _In_opt_ IKeyStore *metadata,
_COM_Outptr_ IModelObject **namespaceModelObject) PURE;
    STDMETHOD(CreateTypedIntrinsicObjectEx)(_In_opt_ IDebugHostContext*
context, _In_ VARIANT* intrinsicData, _In_ IDebugHostType* type,
_COM_Outptr_ IModelObject** object) PURE;
}
```

管理方法

托管数据模型的应用程序（如调试器）使用以下方法集。

C++

```
STDMETHOD(Close)() PURE;
```

Close

托管数据模型的应用程序（如调试器）会在数据模型管理器上调用 Close 方法，以便启动数据模型管理器的关闭进程。 如果数据模型主机在数据模型管理器上释放其最终引用之前没有采用 Close 方法，则可能导致未定义的行为，包括但不限于数据模型管理基础架构的重大泄漏。

对象创建/装箱方法

以下一组方法用于创建新对象或将值装箱到 IModelObject 中 - 数据模型的核心接口。

```
C++

STDMETHOD(CreateNoValue)(_Out_ IModelObject** object) PURE;
STDMETHOD(CreateErrorObject)(_In_ HRESULT hrError, _In_opt_ PCWSTR
pwszMessage, _COM_Outptr_ IModelObject** object) PURE;
STDMETHOD(CreateTypedObject)(_In_opt_ IDebugHostContext* context, _In_
Location objectLocation, _In_ IDebugHostType* objectType, _COM_Errorptr_
IModelObject** object) PURE;
STDMETHOD(CreateTypedObjectReference)(_In_opt_ IDebugHostContext* context,
_In_ Location objectLocation, _In_ IDebugHostType* objectType,
_COM_Errorptr_ IModelObject** object) PURE;
STDMETHOD(CreateSyntheticObject)(_In_opt_ IDebugHostContext* context,
_COM_Outptr_ IModelObject** object) PURE;
STDMETHOD(CreateDataModelObject)(_In_ IDataModelConcept* dataModel,
_COM_Outptr_ IModelObject** object) PURE;
STDMETHOD(CreateIntrinsicObject)(_In_ ModelObjectKind objectKind, _In_
VARIANT* intrinsicData, _COM_Outptr_ IModelObject** object) PURE;
STDMETHOD(CreateTypedIntrinsicObject)(_In_ VARIANT* intrinsicData, _In_
IDebugHostType* type, _COM_Outptr_ IModelObject** object) PURE;
STDMETHOD(CreateMetadataStore)(_In_opt_ IKeyStore* parentStore, _COM_Outptr_
IKeyStore** metadataStore) PURE;
STDMETHOD(CreateTypedIntrinsicObjectEx)(_In_opt_ IDebugHostContext* context,
_In_ VARIANT* intrinsicData, _In_ IDebugHostType* type, _COM_Outptr_
IModelObject** object) PURE;
```

CreateNoValue

CreateNoValue 方法会创建一个“no value”对象，将其装箱到一个 IModelObject 中并将其返回。 返回的模型对象具有一种 ObjectNoValue。

“no value”对象有多种语义：

- (视语言而定）它在语义上相当于 void、null 或 undefined
- 任何属性访问器的 GetValue 方法如果返回成功，并产生一个“no value”对象，则表明该特定属性对给定实例没有值，应将其视为该特定实例不存在该属性。
- 从语义上讲没有返回值的数据模型方法使用此值作为 sentinel 来表示（因为方法必须返回一个有效的 IModelObject）。

CreateErrorObject

CreateErrorObject 方法会创建一个“error object”。 数据模型没有异常和异常流的概念。 属性/方法的失败有两种方式：

- 单个失败的 HRESULT，无扩展的错误信息。 要么是无法提供更多的错误信息，要么是错误本身在返回的 HRESULT 中已不言自明。
- 单个失败的 HRESULT，有扩展的错误信息。 扩展错误信息是属性/方法输出参数中返回的错误对象。

CreateTypedObject

CreateTypedObject 方法允许客户端在调试目标的地址空间中创建一个本地/语言对象的表示。 如果新创建对象的类型（如 objectType 参数所示）恰好与数据模型管理器注册为规范可视化工具或扩展的一个或多个类型签名相匹配，那么在将实例对象返回给调用方之前，这些匹配的数据模型将自动附加到创建的实例对象上。

CreateTypedObjectReference

CreateTypedObjectReference 方法在语义上与 CreateTypedObject 方法相似，只是它创建的是对基本本地/语言构造的引用。 创建的引用是一个具有 ObjectTargetObjectReference 类型的对象。 它不是基本语言可能支持的本地引用（例如：C++ & 或 &&）。 完全可以将 ObjectTargetObjectReference 作为 C++ 引用。 通过使用 IModelObject 上的 Dereference 方法，可以将 ObjectTargetObjectReference 类型的对象转换为基础值。 该引用还可以传递给基础主机的表达式计算器，以便以适合语言的方式重新分配给值。

CreateSyntheticObject

CreateSyntheticObject 方法会创建一个空数据模型对象 - 一个包含键/值/元数据元组和概念的字典。 在创建时，对象上既没有键也没有概念。 这是调用方可以利用的一个干净盖板。

CreateDataModelObject

CreateDataModelObject 方法是一个简单的帮助程序包装器，用于创建作为数据模型的对象，即作为父模型附加到其他对象的对象。 所有此类对象都必须通过 IDataModelConcept 支持数据模型概念。 此方法会创建一个没有显式上下文的新空白合成对象，并添加传入的 IDataModelConcept 作为新创建对象的数据模型概念实现。 这同样可以通过调用 CreateSyntheticObject 和 SetConcept 来实现。

CreateIntrinsicObject

CreateIntrinsicObject 方法是将固有值装箱到 IModelObject 中的方法。 调用方将值放入 COM VARIANT，然后调用此方法。 数据模型管理器会返回一个表示该对象的

IModelObject。 请注意，该方法也用于装箱基于 IUnknown 的基本类型：属性访问器、方法、上下文等。在这种情况下，objectKind 方法表示对象代表哪种基于 IUnknown 的构造，而传递变量的 punkVal 字段就是 IUnknown 派生类型。 在过程中，该类型必须可静态地投向相应的模型接口（例如：IModelPropertyAccessor、IModelMethod、IDebugHostContext 等...）。 此方法支持的 VARIANT 类型包括 VT_UI1、VT_I1、VT_UI2、VT_I2、VT_UI4、VT_I4、VT_UI8、VT_I8、VT_R4、VT_R8、VT_BOOL、VT_BSTR 和 VT_UNKNOWN（用于由枚举 ModelObjectKind 表示的一组专门的 IUnknown 派生类型）。

CreateTypedIntrinsicObject

CreateTypedintrinsicObject 方法与 CreateIntrinsicObject 方法类似，但它允许将本地/语言类型与数据关联起来，并随装箱值一起携带。 这样，数据模型就可以表示诸如本地枚举类型（即简单的 VT_UI* 或 VT_I* 值）之类的构造。 指针类型也可以用这种方法来创建。 数据模型中的本地指针是一个零扩展的 64 位数，表示调试目标虚拟地址空间的偏移量。 它被装箱在 VT_UI8 内，使用此方法和表示本地/语言指针的类型创建。

CreateMetadataStore

CreateMetadataStore 方法会创建一个键存储 - 一个键/值/元数据元组的简化容器，用于保存与属性和其他各种值相关联的元数据。 元数据存储区可能只有一个父级（而父级也可能只有一个父级）。 如果给定的元数据键未在给定的存储空间中，则会检查其父级。 大多数元数据存储都没有父级。 但是，它确实提供了一种轻松共享通用元数据的方法。

CreateTypedIntrinsicObjectEx

CreateTypedIntrinsicObjectEx 方法在语义上类似于 CreateTypedIntrinsicObject 方法。 两者唯一的区别是，此方法允许调用方指定固有数据有效的上下文。 如果没有传递上下文，则数据将在类型参数继承的任何上下文中被视为有效（CreateTypedIntrinsicObject 的行为方式）。 这样就可以在调试目标中创建类型化指针值，而这些指针值所需的特定上下文比从类型中继承的更多。

可扩展性/注册方法以下方法集管理数据模型的可扩展性机制，允许客户端扩展或注册现有模型，或要求数据模型自动将给定的父模型附加到符合给定标准的本地类型上。

C++

```
    STDMETHOD(GetModelForTypeSignature)(_In_ IDebugHostTypeSignature*
typeSignature, _Out_ IModelObject** dataModel) PURE;
    STDMETHOD(GetModelForType)(_In_ IDebugHostType* type, _COM_Outptr_
IModelObject** dataModel, _COM_Outptr_opt_ IDebugHostTypeSignature**
typeSignature, _COM_Outptr_opt_ IDebugHostSymbolEnumerator**
wildcardMatches) PURE;
    STDMETHOD(RegisterModelForTypeSignature)(_In_ IDebugHostTypeSignature*
typeSignature, _In_ IModelObject* dataModel) PURE;
    STDMETHOD(UnregisterModelForTypeSignature)(_In_ IModelObject* dataModel,
```

```
_In_opt_ IDebugHostTypeSignature* typeSignature) PURE;
    STDMETHOD(RegisterExtensionForTypeSignature)(_In_
IDebugHostTypeSignature* typeSignature, _In_ IModelObject* dataModel) PURE;
    STDMETHOD(UnregisterExtensionForTypeSignature)(_In_ IModelObject*
dataModel, _In_opt_ IDebugHostTypeSignature* typeSignature) PURE;
    STDMETHOD(GetRootNamespace)(_COM_Outptr_ IModelObject** rootNamespace)
PURE;
    STDMETHOD(RegisterNamedModel)(_In_ PCWSTR modelName, _In_ IModelObject
*modeObject) PURE;
    STDMETHOD(UnregisterNamedModel)(_In_ PCWSTR modelName) PURE;
    STDMETHOD(AcquireNamedModel)(_In_ PCWSTR modelName, _COM_Outptr_
IModelObject **modelObject) PURE;
```

GetModelForTypeSignature

GetModelForTypeSignature 方法会返回通过事先调用 RegisterModelForTypeSignature 方法针对特定类型签名注册的数据模型。 此方法返回的数据模型被认为是与传递的类型签名相匹配的任何类型的规范可视化工具。 作为一个规范可视化工具，该数据模型会接管该类型的显示。 默认情况下，显示引擎会隐藏对象的本地/语言结构，而采用数据模型所呈现的对象视图。

GetModelForType

GetModelForType 方法返回的数据模型是给定类型实例的规范可视化工具。 实际上，此方法会找到事先调用 RegisterModelForTypeSignature 方法注册的最佳匹配类型签名，并返回相关的数据模型。

RegisterModelForTypeSignature

RegisterModelForTypeSignature 方法是调用方用来为给定类型（或类型集）注册规范可视化工具的主要方法。 规范可视化工具是一种数据模型，它实际上接管了给定类型（或类型集）的显示。 在任何调试器用户界面中都不会显示类型的本地/语言视图，而是显示注册数据模型所呈现的类型视图（同时还为需要的用户提供了返回本地/语言视图的方法）。

UnregisterModelForTypeSignature

UnregisterModelForTypeSignature 方法会撤销之前对 RegisterModelForTypeSignature 方法的调用。 此方法可以删除作为与特定类型签名匹配的类型的规范可视化工具的给定数据模型，也可以删除为该数据模型所注册的每个类型签名的规范可视化工具的给定数据模型。

RegisterExtensionForTypeSignature

RegisterExtensionForTypeSignature 方法与 RegisterModelForTypeSignature 方法相似，但有一个主要区别。 传递给此方法的数据模型不是任何类型的典型可视化工具，也不会取代该类型的本地/语言视图的显示。 传递给此方法的数据模型将自动被添加为任何符合

所提供类型签名的具体类型的父类型。 与 RegisterModelForTypeSignature 方法不同的是，注册相同或模糊类型签名作为给定类型（或类型集）的扩展没有限制。 类型签名与给定具体类型实例相匹配的每个扩展都会导致通过此方法注册的数据模型作为父模型自动附加到新创建的对象上。 实际上，这样将允许任意数量的客户端使用新字段或功能来扩展一个类型（或一组类型）。

UnregisterExtensionForTypeSignature

UnregisterExtensionForTypeSignature 方法会撤销之前对 RegisterExtensionForTypeSignature 的调用。 它可以将特定数据模型取消注册为特定类型签名的扩展，或取消注册为该数据模型所对应的所有类型签名的扩展。

GetRootNamespace

GetRootNamespace 方法会返回数据模型的根命名空间。 这是数据模型管理的对象，调试主机会将某些对象放入其中。

RegisterNamedModel

RegisterNamedModel 方法将给定的数据模型注册为一个众所周知的名称，以便希望扩展该模型的客户端可以找到它。 这就是 API 的主要用途 - 作为可以通过检索以这个众所周知的名称注册的模型并向其添加父模型来扩展的数据模型进行发布。

UnregisterNamedModel

UnregisterNamedModel 方法会撤销之前对 RegisterNamedModel 的调用。 它消除了数据模型与名称之间的关联，而数据模型可以在名称下查找。

AcquireNamedModel

如果调用方希望扩展以给定名称注册的数据模型，则需要调用 AcquireNamedModel 方法，以获取其希望扩展的数据模型的对象。 此方法将返回通过之前调用 RegisterNamedModel 方法注册的任何数据模型。 由于 AcquireNamedModel 方法的主要目的是扩展模型，如果给定名称下尚未注册任何模型，则此方法会产生特殊行为。 如果给定名称下尚未注册任何模型，则会创建一个存根对象，并临时注册到给定名称下，然后返回给调用方。 当通过调用 RegisterNamedModel 方法注册真实数据模型时，对存根对象所做的任何更改实际上都是对真实模型所做的更改。 这样将消除组件之间相互扩展时的许多负载顺序依赖问题。

帮助器方法

以下方法是一般帮助程序方法，可帮助对数据模型中的对象执行复杂操作。 虽然可以通过数据模型或其对象上的其他方法来执行这些操作，但这些方便的方法大大简化了操作：

```
C++
```

```
STDMETHOD(AcquireSubNamespace)(_In_ PCWSTR modelName, _In_ PCWSTR
subNamespaceModelName, _In_ PCWSTR accessName, _In_opt_ IKeyStore *metadata,
_COM_Outptr_ IModelObject **namespaceModelObject) PURE;
```

AcquireSubNamespace

AcquireSubNamespace 方法有助于构建一个更像语言命名空间的东西，而不是动态语言中的一个新对象。 例如，如果调用方希望对进程对象上的属性进行分类，以便让流程对象更有条理，属性更容易被发现，那么一种方法就是为进程对象上的每个类别创建一个子对象，并将这些属性放在该对象中。

另请参阅

本专题是系列专题的一部分，主要介绍 C++ 可访问的接口、如何使用这些接口构建基于 C++ 的调试器扩展，以及如何通过 C++ 数据模型扩展使用其他数据模型构造（如 JavaScript 或 NatVis）。

调试器数据模型 C++ 概述

调试器数据模型 C++ 接口

调试器数据模型 C++ 的其他接口

调试器数据模型 C++ 的概念

调试器数据模型 C++ 脚本

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

调试器数据模型 C++ 的其他接口

项目 • 2024/07/25

本主题介绍与调试器 C++ 数据模型关联的一些其他接口，例如元数据、概念和对象枚举。

调试器数据模型元数据接口

数据模型中的核心概念之一是，对象（尤其是合成对象）是键/值/元数据元组的字典。 每个键可以有一个完整的元数据存储，用于描述键及其潜在值周围的各种内容。 请注意，元数据不会以任何方式更改键的值。 只有与键及其值相关联的辅助信息才会影响键及其值的表示或其他相关属性。

从某种意义上说，元数据存储与数据模型中对象本质的键/值/元数据元组并没有太大不同。 然而，从这个角度来看，它被简化了。 元数据存储由 IKeyStore 接口表示。 虽然也是键/值/元数据元组的集合，但与模型对象相比，元数据键存储的功能存在局限性：

- 一个键存储只能有一个父存储，而不能有任意的父模型链。
- 键存储没有概念。 它只能有键/值/元数据元组的字典。 这意味着键存储中存在的键是静态的。 它们不能由动态语言系统按需创建。
- 仅按照约定，元数据定义的键存储中的值仅限于基本值（内部函数和属性访问器）

虽然键存储可以具有任意数量（和任意命名）的键，但也有某些名称定义了语义值。 目前，这些名称为：

展开表

密钥名称	值类型	说明
PreferredRadix	整数：2、8、10 或 16	指示序号值应以何种基数显示
PreferredFormat	整数：由 PreferredFormat 枚举定义	指示显示值的首选格式类型
PreferredLength	Integer	对于数组和其他容器，指示默认情况下应显示多少个元素
FindDerivation	布尔	指示调试主机在使用前是否应对值执行派生类型分析（例如：显示）
帮助	字符串	按键的工具提示样式帮助文本，可以通过用户界面以适当有用的方式呈现。

密钥名称	值类型	说明
ActionName	字符串	指示给定的方法（不采用任何参数且不返回任何值）是一个操作。 操作的名称在元数据中指定。 用户界面可以利用此名称在上下文菜单或其他适当的界面中显示该选项
ActionIsDefault	布尔	仅当指定 ActionName 键时有效，指示这是对象的默认操作。
ActionDescription	字符串	仅当指定 ActionName 键时有效，这才为操作提供工具提示样式说明。 此类文本可以通过用户界面以适当有用的方式呈现。

请注意，虽然元数据存储中的键可以有自己的元数据 (ad infiniteum)，但目前没有此类用途。 大多数调用方将为 IKeyStore 接口上的方法中的任何元数据参数指定 null。

核心元数据接口：IKeyStore

IKeyStore 接口定义如下：

C++

```
DECLARE_INTERFACE_(IKeyStore, IUnknown)
{
   STDMETHOD(GetKey)(_In_ PCWSTR key, _COM_Errorptr_opt_ IModelObject**
object, _COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
   STDMETHOD(SetKey)(_In_ PCWSTR key, _In_opt_ IModelObject* object,
_In_opt_ IKeyStore* metadata) PURE;
   STDMETHOD(GetKeyValue)(_In_ PCWSTR key, _COM_Errorptr_opt_ IModelObject**
object, _COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
   STDMETHOD(SetKeyValue)(_In_ PCWSTR key, _In_ IModelObject* object) PURE;
   STDMETHOD(ClearKeys)() PURE;
}
```

GetKey

GetKey 方法类似于 IModelObject 上的 GetKey 方法。 如果指定键存在于键存储或键存储的父存储中，它将返回指定键的值。 请注意，如果键的值是属性访问器，则不会在属性访问器上调用 GetValue 方法。 将返回装箱到 IModelObject 中的实际 IModelPropertyAccessor。 出于这个原因，客户端通常会调用 GetKeyValue。

SetKey

SetKey 方法类似于 IModelObject 上的 SetKey 方法。 它是唯一能够在键存储中创建键并将元数据与之关联的方法。

GetKeyValue

GetKeyValue 方法是客户端在元数据存储中查找特定键的值时将采用的第一种方法。 如果键参数指定的键存在于存储（或其父存储）中，则将返回该键的值以及与之关联的任何元数据。 如果键的值是属性访问器（装箱到 IModelObject 中的 IModelPropertyAccessor），则 GetKeyValue 将自动调用属性访问器的 GetValue 方法，并返回属性的基础值。

SetKeyValue

SetKeyValue 方法类似于 IModelObject 上的 SetKeyValue 方法。 此方法无法在元数据存储中创建新键。 如果键参数指示存在现有键，则会按指示设置其值。 如果键是属性访问器，则将在属性访问器上调用 SetValue 方法，以设置基础值。 请注意，元数据创建后通常是静态的。 很少在元数据键存储中使用此方法。

ClearKeys

ClearKeys 方法类似于 IModelObject 上的 ClearKeys 方法。 它将从给定的元数据存储中删除每个键。 此方法对任何父存储都没有任何影响。

数据模型中的对象枚举

枚举数据模型中的对象

数据模型中有两个核心键枚举接口：IKeyEnumerator 和 IRawEnumerator。 虽然这是两个核心接口，但它们可用于以三种样式之一枚举对象：

键 - 可以通过调用 EnumerateKeys 来获取 IKeyEnumerator 接口，以便枚举对象的键及其值/元数据，而无需解析任何基础属性访问器。 此枚举样式可以返回装箱到 IModelObjects 的原始 IModelPropertyAccessor 值。

值 - 可以通过调用 EnumerateKeyValues 或 EnumerateRawValues 来获取 IKeyEnumerator 和 IRawEnumerator 接口，以便枚举对象的键/原始值及其值/元数据。 枚举中存在的任何属性访问器都会在枚举过程中通过调用基础 GetValue 方法自动解析。

引用 - 可以通过调用 EnumerateKeyReferences 或 EnumerateRawReferences 来获取 IKeyEnumerator 和 IRawEnumerator 接口，以便枚举对对象上键/原始值的引用。 可以保存这些引用，并在以后用于获取或设置基础键或原始值。

KeyEnumerator：合成键的枚举

IKeyEnumerator 接口是用于枚举实例对象内的所有键（按键、值或引用）及其父模型链中所有关联的父模型的单个接口。 接口定义如下：

```
C++
```

```
DECLARE_INTERFACE_(IKeyEnumerator, IUnknown)
{
    STDMETHOD(Reset)() PURE;
    STDMETHOD(GetNext)(_Out_ BSTR* key, _COM_Errorptr_opt_ IModelObject**
value, _COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
}
```

重置

Reset 方法将枚举器重置为首次获取枚举器时所处的位置（例如：在枚举中的第一个元素之前）。 对 GetNext 的后续调用将返回第一个枚举键。

GetNext

GetNext 方法既向前移动枚举器，又返回枚举中该位置的键。

IRawEnumerator：枚举本机语言或基础语言 (C/C++) 构造

IRawEnumerator 接口是用于枚举对象内所有本机/语言构造（按值或引用）的单个接口，该对象表示调试目标地址空间内的本机构造。 接口定义如下：

```
C++
```

```
DECLARE_INTERFACE_(IRawEnumerator, IUnknown)
{
    STDMETHOD(Reset)() PURE;
    STDMETHOD(GetNext)(_Out_opt_ BSTR* name, _Out_opt_ SymbolKind *kind,
_COM_Errorptr_opt_ IModelObject** value) PURE;
}
```

重置

Reset 方法将枚举器重置为首次获取枚举器时所处的位置（例如：在枚举中的第一个元素之前）。 对 GetNext 的后续调用将返回第一个枚举的本机/语言构造。

GetNext

GetNext 方法既向前移动枚举器，又返回枚举中该位置的本机/语言构造。

另请参阅

本专题是系列专题的一部分，主要介绍 C++ 可访问的接口、如何使用这些接口构建基于 C++ 的调试器扩展，以及如何通过 C++ 数据模型扩展使用其他数据模型构造（如 JavaScript 或 NatVis）。

调试器数据模型 C++ 概述

调试器数据模型 C++ 接口

调试器数据模型 C++ 对象

调试器数据模型 C++ 的其他接口

调试器数据模型 C++ 的概念

调试器数据模型 C++ 脚本

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

调试器数据模型 C++ 的概念

项目 • 2024/06/04

本主题介绍调试器 C++ 数据模型中的概念。

数据模型中的概念

数据模型中的合成对象实际上是两种东西：

- 键/值/元数据元组的字典。
- 数据模型支持一组概念（接口）。 概念是客户端（而不是数据模型）实现的接口，以提供一组指定的语义行为。 此处列出了当前支持的概念集。

展开表

概念接口	说明
IDataModelConcept	概念是一个父模型。 如果此模型通过注册的类型签名自动附加到本地类型，那么每次实例化该类型的新对象时，InitializeObject 方法都会被自动调用。
IStringDisplayableConcept	对象可转换为字符串以供显示。
IIterableConcept	对象是一个容器并可以迭代。
IIndexableConcept	对象是一个容器，可以在一个或多个维度上进行索引（通过随机存取进行访问）。
IPreferredRuntimeTypeConcept	对象对派生类型的理解比基础类型系统所能提供的要多，因此希望自己处理从静态类型到运行时类型的转换。
IDynamicKeyProviderConcept	该对象是键的动态提供程序，希望接管核心数据模型的所有键查询。 此接口通常用作连接 JavaScript 等动态语言的桥梁。
IDynamicConceptProviderConcept	该对象是概念的动态提供程序，希望接管核心数据模型的所有概念查询。 此接口通常用作连接 JavaScript 等动态语言的桥梁。

数据模型概念：IDataModelConcept

作为父模型附加到另一个模型对象的任何模型对象都必须直接支持数据模型概念。 数据模型概念需要接口的支持，IDataModelConcept 定义如下。

```
C++
```

```
DECLARE_INTERFACE_(IDataModelConcept, IUnknown)
{
    STDMETHOD(InitializeObject)(_In_ IModelObject* modelObject, _In_opt_
IDebugHostTypeSignature* matchingTypeSignature, _In_opt_
IDebugHostSymbolEnumerator* wildcardMatches) PURE;
    STDMETHOD(GetName)(_Out_ BSTR* modelName) PURE;
}
```

InitializeObject

可以通过数据模型管理器的 RegisterModelForTypeSignature 或 RegisterExtensionForTypeSignature 方法将数据模型注册为规范可视化工具或作为给定本机类型的扩展。 通过上述任一方法注册模型时，数据模型会自动作为父模型附加到类型与注册中传递的签名匹配的任何本地对象。 在自动创建附件时，会在数据模型上调用 InitializeObject 方法。 它传递了实例对象、引起附加的类型签名以及生成与类型签名中的任何通配符匹配的类型实例（按线性顺序）的枚举器。 数据模型实现可以使用此方法调用来初始化它所需的任何缓存。

GetName

如果给定的数据模型通过 RegisterNamedModel 方法以默认名称注册，则已注册的数据模型的 IDataModelConcept 接口必须从此方法返回该名称。 请注意，一个模型用多个名称注册是完全合法的（此处应返回默认名称或最佳名称）。 模型可以完全不命名（只要没有注册名称） 在这种情况下，GetName 方法应返回 E_NOTIMPL。

字符串可显示的概念：IStringDisplayableConcept

希望为显示目的提供字符串转换的对象，可以通过实现 IStringDisplayableConcept 接口来实现字符串可显示的概念。 接口定义如下：

C++

```
DECLARE_INTERFACE_(IStringDisplayableConcept, IUnknown)
{
    STDMETHOD(ToDisplayString)(_In_ IModelObject* contextObject, _In_opt_
IKeyStore* metadata, _Out_ BSTR* displayString) PURE;
}
```

ToDisplayString

每当客户端希望将对象转换为字符串以进行显示（到控制台、在 UI 中等）时，就会调用 ToDisplayString 方法。此类字符串转换不应作为其他编程操作的基础。 字符串转换本身可能会受到传递给调用的元数据的影响。 字符串转换应每次尝试都应尽量遵循 PreferredRadix 和 PreferredFormat 键。

可迭代概念：IIterableConcept 和 IModelIterator

如果一个对象是其他对象的容器，并希望表达循环访问这些包含对象的能力，可以通过实现 IIterableConcept 和 IModelIterator 接口来支持可迭代的概念。 可迭代概念的支持与可索引概念的支持之间存在着非常重要的关系。 一个支持随机访问所包含对象的对象，除了支持可迭代概念外，还可以支持可索引概念。 在这种情况下，迭代元素还必须生成默认索引，当传递给可索引的概念时，该索引引用同一对象。 如果不能满足这个不变量，将导致调试主机中出现未定义的行为。

IIterableConcept 定义如下：

```
C++

DECLARE_INTERFACE_(IIterableConcept, IUnknown)
{
    STDMETHOD(GetDefaultIndexDimensionality)(_In_ IModelObject*
contextObject, _Out_ ULONG64* dimensionality) PURE;
    STDMETHOD(GetIterator)(_In_ IModelObject* contextObject, _Out_
IModelIterator** iterator) PURE;
}
```

IModelIterator 概念的定义如下：

```
C++

DECLARE_INTERFACE_(IModelIterator, IUnknown)
{
   STDMETHOD(Reset)() PURE;
   STDMETHOD(GetNext)(_COM_Errorptr_ IModelObject** object, _In_ ULONG64
dimensions, _Out_writes_opt_(dimensions) IModelObject** indexers,
_COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
}
```

IIterableConcept 的 GetDefaultIndexDimensionality

GetDefaultIndexDimensionality 方法返回默认索引的维数。 如果对象不可编制索引，此方法应返回 0 并成功 (S_OK)。 任何从此方法返回非零值的对象都声明支持协议协定，该协定规定：

- 该对象通过支持 IIndexableConcept 支持可索引的概念
- 从可迭代概念的 GetIterator 方法返回的 IModelIterator 的 GetNext 方法将为每个生成的元素返回唯一的默认索引。 此类索引的维数在此标明。
- 将从 IModelIterator 的 GetNext 方法返回的索引传递给可索引概念上的 GetAt 方法 (IIndexableConcept) 将引用 GetNext 生成的同一对象。 返回同样的值。

IIterableConcept 的 GetIterator

可迭代概念上的 GetIterator 方法返回可用于循环访问对象的迭代器接口。 返回的迭代器必须记住传递给 GetIterator 方法的上下文对象。 它不会传递给迭代器本身上的方法。

IModelIterator 的 重置

从可迭代概念返回的迭代器上的 Reset 方法将迭代器的位置还原到首次创建迭代器时的位置（在第一个元素之前）。 虽然强烈建议迭代器支持 Reset 方法，但这不是必需的。 迭代器可以是 C++ 输入迭代器的等效项，并且只允许单次向前迭代。 在这种情况下，Reset 方法可能会失败并出现 E_NOTIMPL。。

IModelIterator 的 GetNext

GetNext 方法向前移动迭代器并提取下一个迭代元素。 如果对象除可迭代外还可索引，且 GetDefaultIndexDimensionality 参数返回的值非零，则此方法可选择返回默认索引，以便从索引器返回生成的值。 请注意，调用方可以选择传递 0/nullptr，而不检索任何索引。 调用方请求部分索引（例如：小于 GetDefaultIndexDimensionality 生成的数字）是非法的。

如果迭代器成功向前移动，但在读取迭代元素的值时出错，该方法可能会返回错误 *并* 用错误对象填充“object”。 在包含元素的迭代结束时，迭代器将从 GetNext 方法返回 E_BOUNDS。 任何后续调用（除非有干预重置呼叫）也会返回 E_BOUNDS。

可索引概念：IIndexableConcept

希望提供对一组内容的随机访问的对象可以通过支持 IIndexableConcept 接口来支持可索引的概念。 通过对可迭代概念的支持，大多数可索引的对象也是可迭代的。 但是，这并不是必需的。 如果支持，则迭代器和索引器之间存在重要关系。 迭代器必须支持 GetDefaultIndexDimensionality，从该方法返回非零值，并支持其中记录的协定。 indexer 概念接口定义如下：

C++

```
DECLARE_INTERFACE_(IIndexableConcept, IUnknown)
{
    STDMETHOD(GetDimensionality)(_In_ IModelObject* contextObject, _Out_
ULONG64* dimensionality) PURE;
    STDMETHOD(GetAt)(_In_ IModelObject* contextObject, _In_ ULONG64
indexerCount, _In_reads_(indexerCount) IModelObject** indexers,
_COM_Errorptr_ IModelObject** object, _COM_Outptr_opt_result_maybenull_
IKeyStore** metadata) PURE;
    STDMETHOD(SetAt)(_In_ IModelObject* contextObject, _In_ ULONG64
indexerCount, _In_reads_(indexerCount) IModelObject** indexers, _In_
IModelObject *value) PURE;
}
```

下面显示了使用索引器（及其与迭代器交互）的示例。 此示例迭代可索引容器的内容，并使用索引器返回刚刚返回的值。 虽然该操作在功能上是无用的，但它演示了这些接口的交互方式。 请注意，以下示例不处理内存分配失败。 它假设引发新异常（这可能是一个糟糕的假设，具体取决于代码所在的环境 - 数据模型的 COM 方法不能有 C++ 异常转义）：

C++

```
ComPtr<IModelObject> spObject;

//
// Assume we have gotten some object in spObject that is iterable (e.g.: an
object which represents a std::vector<SOMESTRUCT>)
//
ComPtr<IIterableConcept> spIterable;
ComPtr<IIndexableConcept> spIndexer;
if (SUCCEEDED(spObject->GetConcept(__uuidof(IIterableConcept), &spIterable,
nullptr)) &&
    SUCCEEDED(spObject->GetConcept(__uuidof(IIndexableConcept),
&spIndexable, nullptr)))
{
    ComPtr<IModelIterator> spIterator;

    //
    // Determine how many dimensions the default indexer is and allocate the
requisite buffer.
    //
    ULONG64 dimensions;
    if (SUCCEEDED(spIterable->GetDefaultIndexDimensionality(spObject.Get(),
&dimensions)) && dimensions > 0 &&
        SUCCEEDED(spIterable->GetIterator(spObject.Get(), &spIterator)))
    {
        std::unique_ptr<ComPtr<IModelObject>[]> spIndexers(new
ComPtr<IModelObject>[dimensions]);

        //
        // We have an iterator.  Error codes have semantic meaning here.
E_BOUNDS indicates the end of iteration.  E_ABORT indicates that
        // the debugger host or application is trying to abort whatever
operation is occurring.  Anything else indicates
        // some other error (e.g.: memory read failure) where the iterator
MIGHT still produce values.
        //
        for(;;)
        {
            ComPtr<IModelObject> spContainedStruct;
            ComPtr<IKeyStore> spContainedMetadata;

            //
            // When we fetch the value from the iterator, it will pass back
the default indices.
            //
            HRESULT hr = spIterable->GetNext(&spContainedStruct, dimensions,
```

```
reinterpret_cast<IModelObject **>(spIndexers.get()), &spContainedMetadata);
            if (hr == E_BOUNDS || hr == E_ABORT)
            {
                break;
            }

            if (FAILED(hr))
            {
                //
                // Decide how to deal with failure to fetch an element.
Note that spContainedStruct *MAY* contain an error object
                // which has detailed information about why the failure
occurred (e.g.: failure to read memory at address X).
                //
            }

            //
            // Use the indexer to get back to the same value.  We already
have them, so there isn't much functional point to this.  It simply
            // highlights the interplay between iterator and indexer.
            //
            ComPtr<IModelObject> spIndexedStruct;
            ComPtr<IKeyStore> spIndexedMetadata;

            if (SUCCEEDED(spIndexer->GetAt(spObject.Get(), dimensions,
reinterpret_cast<IModelObject **>(spIndexers.get()), &spIndexedStruct,
&spIndexedMetadata)))
            {
                //
                // spContainedStruct and spIndexedStruct refer to the same
object.  They may not have interface equality.
                // spContainedMetadata and spIndexedMetadata refer to the
same metadata store with the same contents.  They may not have interface
equality.
                //
            }
        }
    }
}
```

GetDimensionality

GetDimensionality 方法返回对象索引的维数。 请注意，如果对象既可迭代又可索引，则 GetDefaultIndexDimensionality 的实现必须与 GetDimensionality 的实现一致，即索引器具有的维度数。

GetAt

GetAt 方法从索引对象内检索特定 N 维索引处的值。 必须支持 N 维索引器，其中 N 是从 GetDimensionality 返回的值。 请注意，一个对象可以在不同的域中通过不同的类型进行

索引（例如：可以通过序数和字符串进行索引）。 如果索引不在范围（或无法访问），该方法将返回失败；但是，在这种情况下，输出对象可能仍设置为错误对象。

SetAt

SetAt 方法试图在索引对象中设置特定 N 维索引的值。 必须支持 N 维索引器，其中 N 是从 GetDimensionality 返回的值。 请注意，一个对象可以在不同的域中通过不同的类型进行索引（例如：可以通过序数和字符串进行索引）。 某些索引器为只读。 在这种情况下，任何对 SetAt 方法的调用都将返回 E_NOTIMPL。

首选运行时类型概念：IPreferredRuntimeTypeConcept

可以查询调试主机，尝试根据符号信息中的静态类型确定对象的实际运行时类型。 此转换可能基于完全准确的信息（例如 C++ RTTI），也可以基于强启发法，例如对象中任何虚拟函数表的形状。 但是，某些对象无法从静态对象转换为运行时类型，因为它们不适合调试主机的启发式（例如：它们没有 RTTI 或虚拟函数表）。 在这种情况下，对象的数据模型可以选择替代默认行为，并声明它比调试主机能够理解的更了解对象的"运行时类型"。 这是通过首选运行时类型概念和支持 IPreferredRuntimeTypeConcept 接口完成的。

IPreferredRuntimeTypeConcept 接口声明如下：

```
C++

DECLARE_INTERFACE_(IPreferredRuntimeTypeConcept, IUnknown)
{
    STDMETHOD(CastToPreferredRuntimeType)(_In_ IModelObject* contextObject,
_COM_Errorptr_ IModelObject** object) PURE;
}
```

CastToPreferredRuntimeType

每当客户端希望将静态类型实例转换为该实例的运行时类型时，都将调用 CastToPreferredRuntimeType 方法。 如果相关对象支持（通过其附加的父模型之一）首选运行时类型概念，将调用此方法来执行转换。 此方法可能返回原始对象（没有转换或无法分析）、返回运行时类型的新实例、出于非语义原因（例如：内存不足）或返回 E_NOT_SET。 EE_NOT_SET 错误代码是一个非常特殊的错误代码，它向数据模型指示实现不希望重写默认行为，并且数据模型应回退到调试主机执行的任何分析（例如：RTTI 分析、虚拟函数表的形状检查、 etc...）

动态提供程序概念：IDynamicKeyProviderConcept 和 IDynamicConceptProviderConcept

虽然数据模型本身通常会处理对象的关键和概念管理，但有时这种概念不太理想。 具体而言，当客户端希望在数据模型与真正动态的其他内容（例如 JavaScript）之间创建桥梁时，从数据模型中的实现中接管关键和概念管理可能很有价值。 由于核心数据模型是

IModelObject 的唯一实现，因此要通过两个概念的组合来实现：动态密钥提供者概念和动态概念提供者概念。 虽然典型的做法是两者都执行或两者都不执行，但并没有这样的要求。

如果两者都已实现，则必须在动态概念提供程序概念之前添加动态密钥提供程序概念。 这两个概念都是特殊的。 它们有效地在对象上翻转开关，将其从“静态托管”更改为“动态管理”。 仅当对象上没有数据模型管理的键/概念时，才能设置这些概念。 一旦这些概念被添加到一个对象上，这样做的动作是不可撤销的。

作为动态概念提供程序的 IModelObject 与非动态概念提供程序在可扩展性方面存在额外的语义差异。 这些概念旨在允许客户端在数据模型和动态语言系统（如 JavaScript）之间创建桥梁。 数据模型具有可扩展性的概念，它与 JavaScript 等系统有根本的不同，因为它有一个父模型树，而不是像 JavaScript 原型链那样的线性链。 为了更好地与此类系统建立关系，作为动态概念提供者的 IModelObject 具有单一数据模型父级。 该单个数据模型父模型是一个普通的 IModelObject，其父模型可以具有任意数量的父模型，就像数据模型的典型一样。 向动态概念提供程序添加或删除父级的任何请求都会自动重定向到单个父级。 从局外人的角度看，动态概念提供者似乎有一个普通的树状父模型链。 动态概念提供程序概念的实现者是唯一了解中间单个父级的对象（核心数据模型外部）。 单个父类可以与动态语言系统相连接，从而提供一个桥梁（例如：放入 JavaScript 原型链中）。

动态密钥提供程序概念的定义如下：

C++

```
DECLARE_INTERFACE_(IDynamicKeyProviderConcept, IUnknown)
{
    STDMETHOD(GetKey)(_In_ IModelObject *contextObject, _In_ PCWSTR key,
_COM_Outptr_opt_result_maybenull_ IModelObject** keyValue,
_COM_Outptr_opt_result_maybenull_ IKeyStore** metadata, _Out_opt_ bool
*hasKey) PURE;
    STDMETHOD(SetKey)(_In_ IModelObject *contextObject, _In_ PCWSTR key,
_In_ IModelObject *keyValue, _In_ IKeyStore *metadata) PURE;
    STDMETHOD(EnumerateKeys)(_In_ IModelObject *contextObject, _COM_Outptr_
IKeyEnumerator **ppEnumerator) PURE;
}
```

动态概念提供程序概念的定义如下：

C++

```
DECLARE_INTERFACE_(IDynamicConceptProviderConcept, IUnknown)
{
    STDMETHOD(GetConcept)(_In_ IModelObject *contextObject, _In_ REFIID
conceptId, _COM_Outptr_result_maybenull_ IUnknown **conceptInterface,
_COM_Outptr_opt_result_maybenull_ IKeyStore **conceptMetadata, _Out_ bool
*hasConcept) PURE;
```

```
    STDMETHOD(SetConcept)(_In_ IModelObject *contextObject, _In_ REFIID
conceptId, _In_ IUnknown *conceptInterface, _In_opt_ IKeyStore
*conceptMetadata) PURE;
    STDMETHOD(NotifyParent)(_In_ IModelObject *parentModel) PURE;
    STDMETHOD(NotifyParentChange)(_In_ IModelObject *parentModel) PURE;
    STDMETHOD(NotifyDestruct)() PURE;
}
```

IDynamicKeyProviderConcept 的 GetKey

动态密钥提供程序上的 GetKey 方法在很大程度上是对 IModelObject 上的 GetKey 方法的覆盖。 动态密钥提供程序应返回密钥的值以及与该密钥关联的任何元数据。 如果密钥不存在（但未发生其他错误），提供程序必须在 hasKey 参数中返回 false，并成功执行 S_OK。 该调用失败将被视为获取密钥失败，并将明确停止通过父模型链搜索密钥。 在 hasKey 中返回 false，成功将继续搜索密钥。 请注意，GetKey 返回一个盒式属性访问器作为键是完全合法的。 这在语义上与 IModelObject 上返回属性访问器的 GetKey 方法相同。

IDynamicKeyProviderConcept 的 SetKey

动态密钥提供程序上的 SetKey 方法实际上是 IModelObject 上的 SetKey 方法的覆盖。 这会在动态提供程序中设置密钥。 它实际上是在提供程序上创建新属性。 请注意，不支持创建 expando 属性等任何概念的提供程序应在此处返回 E_NOTIMPL。

IDynamicKeyProviderConcept 的 EnumerateKeys

动态密钥提供程序上的 EnumerateKeys 方法实际上是 IModelObject 上的 EnumerateKeys 方法的覆盖。 这会枚举动态提供程序中的所有密钥。 返回的枚举器具有多个限制，实现必须遵循这些限制：

- 它的行为必须是对 EnumerateKeys 的调用，而不是 EnumerateKeyValues 或 EnumerateKeyReferences。 它必须返回不解析任何基础属性访问器的键值（如果提供程序中存在此类概念）。
- 从单个动态密钥提供程序的角度来看，枚举多个具有相同名称且物理上不同的密钥是非法的。 这可能发生在通过父模型链连接的不同提供商上，但从单个提供商的角度来看，这不可能发生。

IDynamicConceptProviderConcept 的 GetConcept

动态概念提供程序上的 GetConcept 方法实际上是 IModelObject 上的 GetConcept 方法的覆盖。 动态概念提供程序必须返回查询概念的接口（如果存在），以及与该概念关联的任何元数据。 如果提供程序上不存在概念，则必须通过 hasConcept 参数中返回的 false 值和成功的返回来表明这一点。 此方法失败则无法获取概念，并将明确停止对该概念的搜索。 如果 hasConcept 返回 false 且代码成功，则将继续通过父模型树搜索概念。

IDynamicConceptProviderConcept 的 SetConcept

动态概念提供程序上的 SetConceptt 方法实际上是 IModelObject 上的 SetConcept 方法的覆盖。 动态提供程序将分配概念。 这可能使对象可迭代、可索引、字符串可转换等...请注意，不允许对其创建概念的提供程序应在此处返回 E_NOPTIMPL。

IDynamicConceptProviderConcept 的 NotifyParent

核心数据模型使用对动态概念提供程序的 NotifyParent 调用，以通知创建的单个父模型的动态提供程序，以便将数据模型的“多个父模型”范例桥接为更多动态语言。 对单个父模型的任何操作都会导致向动态提供程序发出进一步通知。 请注意，此回调是在分配动态概念提供程序概念时立即进行的。

IDynamicConceptProviderConcept 的 NotifyParentChange

动态概念提供程序上的 NotifyParent 方法是在对对象的单个父模型进行静态操作时由核心数据模型进行的回调。 对于任何已添加的父模型，在添加父模型时会第一次调用该方法，如果/当删除父模型时会第二次调用该方法。

IDynamicConceptProviderConcept 的 NotifyDestruct

动态概念提供程序上的 NotifyDestruct 方法是由核心数据模型在销毁对象（动态概念提供程序）开始时进行的回调。 它为有需要的客户提供额外的清理机会。

--

另请参阅

本专题是系列专题的一部分，主要介绍 C++ 可访问的接口、如何使用这些接口构建基于 C++ 的调试器扩展，以及如何通过 C++ 数据模型扩展使用其他数据模型构造（如 JavaScript 或 NatVis）。

调试器数据模型 C++ 概述

调试器数据模型 C++ 接口

调试器数据模型 C++ 对象

调试器数据模型 C++ 的其他接口

调试器数据模型 C++ 的概念

调试器数据模型 C++ 脚本

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

调试器数据模型 C++ 脚本

项目 • 2023/03/08

本主题介绍如何使用调试器数据模型 C++ 调试器数据模型 C++ 脚本来支持使用脚本通过调试器引擎实现自动化。

调试器数据模型中的脚本管理

除了数据模型管理器作为对象创建和扩展性的中心机构的角色外，它还负责管理脚本的抽象概念。 从数据模型管理器的脚本管理器部分的角度来看，脚本可由提供程序动态加载、卸载和调试，以便向数据模型扩展或提供新功能。

脚本提供程序是将语言（（例如 NatVis、JavaScript 等）桥接到数据模型的组件，）。 它注册一个或多个文件扩展名 (例如：“。NatVis“、”.js”) 由提供程序处理，允许调试器客户端或用户界面通过委托给提供程序加载具有该特定扩展名的脚本文件。

核心脚本管理器：IDataModelScriptManager

核心脚本管理器接口定义如下。

C++

```
DECLARE_INTERFACE_(IDataModelScriptManager, IUnknown)
{
    STDMETHOD(GetDefaultNameBinder)(_COM_Outptr_ IDataModelNameBinder
**ppNameBinder) PURE;
    STDMETHOD(RegisterScriptProvider)(_In_ IDataModelScriptProvider
*provider) PURE;
    STDMETHOD(UnregisterScriptProvider)(_In_ IDataModelScriptProvider
*provider) PURE;
    STDMETHOD(FindProviderForScriptType)(_In_ PCWSTR scriptType,
_COM_Outptr_ IDataModelScriptProvider **provider) PURE;
    STDMETHOD(FindProviderForScriptExtension)(_In_ PCWSTR scriptExtension,
_COM_Outptr_ IDataModelScriptProvider **provider) PURE;
    STDMETHOD(EnumerateScriptProviders)(_COM_Outptr_
IDataModelScriptProviderEnumerator **enumerator) PURE;
}
```

GetDefaultNameBinder

GetDefaultNameBinder 方法返回数据模型的默认脚本名称绑定器。 名称绑定器是解析对象上下文中名称的组件。 例如，给定表达式“foo.bar”，将调用名称联编程序来解析对象 foo 上下文中的名称栏。 此处返回的绑定器遵循数据模型的一组默认规则。 脚本提供程序可以使用此联编程序跨提供程序提供名称解析的一致性。

RegisterScriptProvider

RegisterScriptProvider 方法通知数据模型存在一个新的脚本提供程序，该提供程序能够将新语言桥接到数据模型。 调用此方法时，脚本管理器将立即回调给定的脚本提供程序，并查询它所管理的脚本的属性。 如果已在给定脚本提供程序指示的名称或文件扩展名下注册了提供程序，则此方法将失败。 只能将单个脚本提供程序注册为特定名称或文件扩展名的处理程序。

UnregisterScriptProvider

UnregisterScriptProvider 方法撤消对 RegisterScriptProvider 方法的调用。 传入的脚本提供程序给定的名称和文件扩展名将不再与之关联。 请务必注意，即使取消注册，也可能有大量对脚本提供程序的未完成 COM 引用。 此方法仅阻止加载/创建给定脚本提供程序所管理的类型的脚本。 如果该提供程序加载的脚本仍在加载或已操作调试器的对象模型(或数据模型)，则这些操作可能仍有引用返回到脚本中。 可能存在直接引用脚本中的构造的数据模型、方法或对象。 脚本提供程序必须准备好处理该操作。

FindProviderForScriptType

FindProviderForScriptType 方法在脚本管理器中搜索具有脚本类型字符串的提供程序，如此方法所示。 如果找不到，此方法将失败;否则，此类脚本提供程序将返回到调用方。

EnumerateScriptProviders

EnumerateScriptProviders 方法将返回一个枚举器，该枚举器将枚举通过之前调用 RegisterScriptProvider 方法注册到脚本管理器的每个脚本提供程序。

脚本提供程序枚举：IDataModelScriptProviderEnumerator

EnumerateScriptProviders 方法将返回以下形式的枚举器：

```
C++

DECLARE_INTERFACE_(IDataModelScriptProviderEnumerator, IUnknown)
{
    STDMETHOD(Reset)() PURE;
    STDMETHOD(GetNext)(_COM_Outptr_ IDataModelScriptProvider **provider)
PURE;
}
```

重置

Reset 方法会将枚举器移到返回第一个元素之前所在的位置。

GetNext

GetNext 方法将枚举器向前移动一个元素，并返回位于该元素的脚本提供程序。 当枚举器到达枚举末尾时，将返回E_BOUNDS。 收到此错误后调用 GetNext 方法将继续无限期返回E_BOUNDS。

用于脚本的调试器数据模型 C++ 主机接口

主机在脚本中的角色

调试主机公开了一系列非常低级别的接口，用于了解其调试目标的类型系统的性质 () 、用其调试目标的语言计算表达式 () 等...通常，它不涉及更高级别的构造，如脚本编写。 这会留给整个调试器应用程序或提供这些功能的扩展。 但是，这有一个例外。 任何想要参与数据模型提供的整体脚本编写体验的调试主机都需要实现几个简单的接口，以便为脚本提供上下文。 实际上，调试主机控制它希望脚本环境在数据模型的命名空间中放置函数和其他脚本提供的功能的位置。 参与此过程允许主机 (或不) 在其表达式计算器中使用此类函数。 从主机的角度来看，涉及的接口如下：

接口	说明
IDebugHostScriptHost	接口，指示调试主机参与脚本环境的能力。 此接口允许创建上下文，告知脚本引擎放置对象的位置。
IDataModelScriptHostContext	由脚本提供程序用作脚本内容的容器的主机接口。 脚本的内容如何呈现，而不是对调试器应用程序的对象模型执行的操作，由特定的调试主机决定。 此接口允许脚本提供程序获取有关放置其内容的位置的信息。 有关详细信息，请参阅本主题后面的 数据模型 C++ 脚本接口 。

调试主机的脚本主机：IDebugHostScriptHost

IDebugHostScriptHost 接口是脚本提供程序用于从新创建的脚本的调试主机获取上下文的接口。 此上下文包括由调试主机 (提供的对象) ，其中脚本提供程序可以在数据模型和脚本环境之间放置任何桥。 例如，此类桥可能是调用脚本函数的数据模型方法。 这样做允许数据模型端的调用方利用 IModelMethod 接口上的 Call 方法调用脚本方法。

IDebugHostScriptHost 接口的定义如下。

```
C++

DECLARE_INTERFACE_(IDebugHostScriptHost, IUnknown)
{
    STDMETHOD(CreateContext)(_In_ IDataModelScript* script, _COM_Outptr_
IDataModelScriptHostContext** scriptContext) PURE;
}
```

CreateContext

脚本提供程序调用 CreateContext 方法以创建一个新上下文，用于放置脚本的内容。 此类上下文由“数据模型 C++ 脚本接口”页上详细介绍的 IDataModelScriptHostContext 接口表示。

调试器数据模型 C++ 脚本接口

脚本和脚本接口

数据模型的整体体系结构允许第三方定义某种语言和数据模型的对象模型之间的桥梁。 通常，桥接的语言是一种脚本语言，因为数据模型的环境非常动态。 在语言和数据模型的对象模型之间定义和实现此桥的组件称为脚本提供程序。 初始化后，脚本提供程序会将自身注册到数据模型管理器的脚本管理器部分，管理扩展性的任何接口随后都允许编辑、加载、卸载和调试以脚本提供程序管理的语言编写的脚本。

请注意，适用于 Windows 的调试工具目前定义了两个脚本提供程序。

- NatVis 提供程序。 此提供程序嵌入在 NatVis XML 和数据模型之间的DbgEng.dll和桥中，允许本机/语言数据类型的可视化。
- JavaScript 提供程序。 此提供程序包含在旧版调试器扩展中：JsProvider.dll。 它在以 JavaScript 语言编写的脚本和数据模型之间架起桥梁，从而允许任意形式的调试器控件和扩展性。

可以编写新的提供程序，以桥接其他语言 (例如 Python 等...) 到数据模型。 此类当前封装在旧版调试器扩展中，以便进行加载。 脚本提供程序本身应最大程度地减少旧引擎接口的依赖项，并且应尽可能仅利用数据模型 API。 这样，提供程序可以更方便地移植到其他环境。

有两类与脚本提供程序相关的接口。 第一类接口用于对脚本提供程序及其所管理的脚本进行常规管理。 第二类接口用于支持脚本调试。 虽然对第一个集的支持是必需的，但对第二个集的支持是可选的，对于每个提供程序都可能没有意义。

常规管理接口包括：

接口	说明
IDataModelScriptProvider	脚本提供程序必须实现的核心接口。 这是向数据模型管理器的脚本管理器部分注册的接口，用于播发提供程序对特定类型脚本的支持并针对特定文件扩展名进行注册
IDataModelScript	由提供程序管理的特定脚本的抽象。 加载或编辑的每个脚本都有单独的 IDataModelScript 实例

接口	说明
IDataModelScriptClient	脚本提供程序用于将信息传达给用户界面的客户端接口。 脚本提供程序不实现此接口。 托管希望使用脚本提供程序的数据模型的应用程序会使用脚本提供程序。 脚本提供程序将调用脚本客户端的方法来报告状态、错误等...
IDataModelScriptHostContext	脚本提供程序用作脚本内容的容器的主机接口。 除了脚本对调试器应用程序的对象模型执行的操作之外，脚本的内容如何呈现，由特定的调试主机决定。 此接口允许脚本提供程序获取有关放置其内容的位置的信息。
IDataModelScriptTemplate	脚本提供程序可以提供一个或多个模板，用作用户创作脚本的起点。 提供内置编辑器的调试器应用程序可以使用提供程序通过此接口播发的模板内容预填充新脚本。
IDataModelScriptTemplateEnumerator	脚本提供程序实现的枚举器接口，用于播发它支持的所有各种模板。
IDataModelNameBinder	一个名称绑定器 - 一个对象，它可以将上下文中的名称与值相关联。 对于给定的表达式（如“foo.bar”），名称绑定器能够在对象“foo”的上下文中绑定名称“bar”，并生成值或对它的引用。 名称绑定器通常不由脚本提供程序实现; 相反，可以从数据模型获取默认绑定程序，并由脚本提供程序使用

调试接口包括：

接口	说明
IDataModelScriptDebug	脚本提供程序为使脚本可调试而必须提供的核心接口。 如果脚本可调试，则 IDataModelScript 接口的实现类必须为 IDataModelScriptDebug 的 QueryInterface。
IDataModelScriptDebugClient	希望提供脚本调试功能的用户界面实现 IDataModelScriptDebugClient 接口。 脚本提供程序利用此接口来 (来回传递调试信息，例如发生的事件、断点等...)
IDataModelScriptDebugStack	脚本提供程序实现此接口，以向脚本调试器公开调用堆栈的概念。
IDataModelScriptDebugStackFrame	脚本提供程序实现此接口以公开调用堆栈中特定堆栈帧的概念。
IDataModelScriptDebugVariableSetEnumerator	脚本提供程序实现此接口以公开一组变量。 此集可以表示函数的参数集、局部变量集或特定范围内的变量集。 确切的含义取决于获取接口的方式。

接口	说明
IDataModelScriptDebugBreakpoint	脚本提供程序实现此接口以公开脚本中特定断点的概念和控制。
IDataModelScriptDebugBreakpointEnumerator	脚本提供程序实现此操作以枚举脚本中当前存在的所有断点 (无论是否) 。

核心脚本提供程序：IDataModelScriptProvider

任何想要成为脚本提供程序的扩展都必须提供 IDataModelScriptProvider 接口的实现，并通过 RegisterScriptProvider 方法向数据模型管理器的脚本管理器部分注册此类扩展。 必须实现的此核心接口定义如下。

```
C++

DECLARE_INTERFACE_(IDataModelScriptProvider, IUnknown)
{
    STDMETHOD(GetName)(_Out_ BSTR *name) PURE;
    STDMETHOD(GetExtension)(_Out_ BSTR *extension) PURE;
    STDMETHOD(CreateScript)(_COM_Outptr_ IDataModelScript **script) PURE;
    STDMETHOD(GetDefaultTemplateContent)(_COM_Outptr_
IDataModelScriptTemplate **templateContent) PURE;
    STDMETHOD(EnumerateTemplates)(_COM_Outptr_
IDataModelScriptTemplateEnumerator **enumerator) PURE;
}
```

GetName

GetName 方法返回 (类型或) 脚本语言的名称，提供程序将其作为通过 SysAllocString 方法分配的字符串进行管理。 调用方负责通过 SysFreeString 释放返回的字符串。 可能从此方法返回的字符串示例为“JavaScript”或“NatVis”。 返回的字符串可能会显示在托管数据模型的调试器应用程序的用户界面中。 (不区分大小写) ，任何两个脚本提供程序都不能返回同一名称。

GetExtension

GetExtension 方法返回此提供程序管理的脚本的文件扩展名， (没有点) 作为通过 SysAllocString 方法分配的字符串。 托管具有脚本支持) 的数据模型 (的调试器应用程序会将使用此扩展名打开的脚本文件委托给脚本提供程序。 调用方负责通过 SysFreeString 释放返回的字符串。 可能从此方法返回的字符串示例为“js”或“NatVis”。

CreateScript

调用 CreateScript 方法以创建新脚本。 每当调用此方法时，脚本提供程序都必须返回由返回的 IDataModelScript 接口表示的新空脚本。 请注意，无论用户界面是创建新的空白脚本供用户编辑，还是调试器应用程序是否正在从磁盘加载脚本，都会调用此方法。 提

供程序不参与文件 I/O。 它仅通过传递给 IDataModelScript 上方法的流来处理来自托管应用程序的请求。

GetDefaultTemplateContent

GetDefaultTemplateContent 方法返回提供程序的默认模板内容的接口。 这是脚本提供程序希望在编辑窗口中为新创建的脚本预先填充的内容。 如果脚本提供程序没有 (模板，或者没有指定为默认内容) 的模板内容，则脚本提供程序可能会从此方法返回E_NOTIMPL。

EnumerateTemplates

EnumerateTemplates 方法返回一个枚举器，该枚举器能够枚举脚本提供程序提供的各种模板。 模板内容是脚本提供程序希望在创建新脚本时“预填充”到编辑窗口中的内容。 如果支持多个不同的模板，则可以 (命名这些模板，例如：“命令性脚本”、“扩展脚本”)，托管数据模型的调试器应用程序可以选择如何向用户呈现“模板”。

核心脚本接口：IDataModelScript

管理提供程序实现的单个脚本的 main 接口是 IDataModelScript 接口。 当客户端希望创建新的空白脚本并在 IDataModelScriptProvider 上调用 CreateScript 方法时，将返回实现此接口的组件。

提供程序创建的每个脚本都应位于一个独立的接收器中。 一个脚本不应影响另一个脚本，除非通过数据模型与外部对象进行显式交互。 例如，两个脚本都可以扩展某种类型或概念 (例如：调试器对进程) 的概念。 然后，任一脚本都可以通过外部进程对象访问彼此的字段。

接口定义如下。

C++

```
DECLARE_INTERFACE_(IDataModelScript, IUnknown)
{
    STDMETHOD(GetName)(_Out_ BSTR *scriptName) PURE;
    STDMETHOD(Rename)(_In_ PCWSTR scriptName) PURE;
    STDMETHOD(Populate)(_In_ IStream *contentStream) PURE;
    STDMETHOD(Execute)(_In_ IDataModelScriptClient *client) PURE;
    STDMETHOD(Unlink)() PURE;
    STDMETHOD(IsInvocable)(_Out_ bool *isInvocable) PURE;
    STDMETHOD(InvokeMain)(_In_ IDataModelScriptClient *client) PURE;
}
```

GetName

GetName 方法通过 SysAllocString 函数以分配的字符串的形式返回脚本的名称。 如果脚本尚没有名称，该方法应返回空 BSTR。 在这种情况下，它不应失败。 如果通过调用 Rename 方法显式重命名脚本，则 GetName 方法应返回新分配的名称。

重命名

Rename 方法为脚本分配新名称。 脚本实现负责保存此名称，并在调用 GetName 方法时返回它。 当用户界面选择将脚本另存为新名称时，通常会调用它。 请注意，重命名脚本可能会影响宿主应用程序选择投影脚本内容的位置。

填充

客户端调用 Populate 方法以更改或同步脚本的“内容”。 这是向脚本提供程序发出的脚本代码已更改的通知。 请务必注意，此方法不会导致脚本的执行或更改脚本操作的任何对象。 这只是向脚本提供程序发出通知，告知脚本的内容已更改，以便它可以同步自己的内部状态。

执行

Execute 方法执行由上次成功填充调用所指示的脚本内容，并根据该内容修改调试器的对象模型。 如果语言 (或脚本提供程序) 定义了一个“main函数”（作者希望在单击用户界面中虚构的“执行脚本”按钮时调用的函数），则在执行操作期间不会调用此类“main函数”。 可将 Execute 操作视为仅 (执行初始化和对象模型操作，例如执行根代码和设置扩展点)。

取消链接

Unlink 方法撤消 Execute 操作。 在执行脚本期间建立的任何对象模型操作或扩展点都是撤消的。 执行取消链接操作后，可以通过调用 Execute 重新执行脚本，也可以释放该脚本。

IsInvocable

IsInvocable 方法返回脚本是否可调用 ，即是否具有由其语言或提供程序定义的“main函数”。 从概念上讲，如果用户界面中按下了虚构的“执行脚本”按钮，脚本作者需要调用这种“main函数”。

InvokeMain

如果脚本具有旨在从 UI 调用执行的“main函数”，则它通过 IsInvocable 方法的真实返回来指示此类函数。 然后，用户界面可以调用 InvokeMain 方法，以实际“调用”脚本。 请注意，这不同于 *Execute* ，后者运行所有根代码并将脚本桥接到基础主机的命名空间。

**脚本客户端：IDataModelScriptClient **

托管数据模型的应用程序想要管理脚本并具有用户界面 (围绕此概念的图形或控制台) 实现 IDataModelScriptClient 接口。 在执行或调用期间，此接口将传递给任何脚本提供程序或脚本，以便将错误和事件信息传递回用户界面。

IDataModelScriptClient 接口的定义如下。

C++

```
DECLARE_INTERFACE_(IDataModelScriptClient, IUnknown)
{
   STDMETHOD(ReportError)(_In_ ErrorClass errClass, _In_ HRESULT hrFail,
_In_opt_ PCWSTR message, _In_ ULONG line, _In_ ULONG position) PURE;
}
```

ReportError

如果在脚本执行或调用期间发生错误，脚本提供程序将调用 ReportError 方法以通知用户界面错误。

脚本的主机上下文：IDataModelScriptHostContext

调试主机对数据模型脚本内容的投影方式和位置有一定影响。 预期每个脚本都会要求主机提供一个上下文，以便在其中放置脚本 (桥，例如：可调用的函数对象等...) 。通过调用 IDebugHostScriptHost 上的 CreateContext 方法并获取 IDataModelScriptHostContext 检索此上下文。

IDataModelScriptHostContext 接口的定义如下。

C++

```
DECLARE_INTERFACE_(IDataModelScriptHostContext, IUnknown)
{
   STDMETHOD(NotifyScriptChange)(_In_ IDataModelScript* script, _In_
ScriptChangeKind changeKind) PURE;
   STDMETHOD(GetNamespaceObject)(_COM_Outptr_ IModelObject**
namespaceObject) PURE;
}
```

NotifyScriptChange

在通过对关联上下文上的 NotifyScriptChange 方法进行方法调用时，脚本提供程序要求在发生某些操作时通知调试主机。 此类操作定义为 ScriptChangeKind 枚举的成员

GetNamespaceObject

GetNamespaceObject 方法返回一个 对象，脚本提供程序可以在数据模型和脚本之间放置任何桥。 例如，脚本提供程序可能会将数据模型方法对象 (IModelMethod 接口放入 IModelObject) 其实现调用脚本中相应命名的函数。

新创建的脚本的模板：IDataModelScriptTemplate

想要为新脚本呈现预先填充的内容的脚本提供程序 (例如：帮助用户在调试器用户界面中编写脚本) 可以通过提供一个或多个脚本模板来执行此操作。 此类模板是实现

IDataModelScriptTemplate 接口的组件，并通过脚本提供程序上的 GetDefaultTemplate 方法或 EnumerateTemplates 方法返回。

IDataModelScriptTemplate 接口的定义如下。

C++

```
DECLARE_INTERFACE_(IDataModelScriptTemplate, IUnknown)
{
   STDMETHOD(GetName)(_Out_ BSTR *templateName) PURE;
   STDMETHOD(GetDescription)(_Out_ BSTR *templateDescription) PURE;
   STDMETHOD(GetContent)(_COM_Outptr_ IStream **contentStream) PURE;
}
```

GetName

GetName 方法返回模板的名称。 如果模板没有名称，则此操作可能会失败并 E_NOTIMPL。 如果存在，则单个默认模板 () 不需要具有名称。 所有其他模板都是。 这些名称可以在用户界面中显示为菜单的一部分，以选择要创建的模板。

GetDescription

GetDescription 方法返回模板的说明。 此类说明将在更具描述性的界面中呈现给用户，以帮助用户了解模板的设计目的。 如果模板没有说明，则可能会从此方法返回 E_NOTIMPL。

GetContent

GetContent 方法返回模板的内容 (或代码) 。 如果用户选择从此模板创建新脚本，则会预先填充到编辑窗口中。 该模板负责创建 (并返回) 客户端可以拉取的内容的标准流。

提供程序的模板内容的枚举：IDataModelScriptTemplateEnumerator

脚本提供程序可以提供一个或多个模板，用于将内容预先填充到某些用户界面中新创建的脚本中。 如果提供了这些模板中的任何一个，则脚本提供程序必须对其实现枚举器，该枚举器在调用 EnumerateTemplates 方法时返回。

此类枚举器是 IDataModelScriptTemplateEnumerator 接口的实现，定义如下。

C++

```
DECLARE_INTERFACE_(IDataModelScriptTemplateEnumerator, IUnknown)
{
   STDMETHOD(Reset)() PURE;
   STDMETHOD(GetNext)(_COM_Outptr_ IDataModelScriptTemplate
**templateContent) PURE;
}
```

重置

Reset 方法将枚举器重置为首次创建枚举器时所处的位置，即生成第一个模板之前的位置。

GetNext

GetNext 方法将枚举器移动到下一个模板并返回它。 枚举结束时，枚举器返回 E_BOUNDS。 命中E_BOUNDS标记后，枚举器将继续无限期地生成E_BOUNDS错误，直到发出 Reset 调用。

解析名称的含义：IDataModelNameBinder

数据模型为脚本提供程序提供了一种标准方法，用于确定给定上下文中给定名称的含义，(例如：确定条形图对于将跨各种脚本提供程序运行的 foo.bar) 的含义。 此机制称为名称绑定器，由 IDataModelNameBinder 接口表示。 此类绑定器封装一组规则，这些规则涉及名称如何解析以及如何处理在对象上多次定义名称的冲突解决。 这些规则的一部分包括诸如投影名称 (数据模型添加的名称) 如何根据本机名称解析 (正调试) 语言的类型系统中的本机名称。

为了在脚本提供程序之间提供一定程度的一致性，数据模型的脚本管理器提供了默认名称绑定器。 可以通过调用 IDataModelScriptManager 接口上的 GetDefaultNameBinder 方法获取此默认名称绑定器。 名称绑定器接口定义如下。

C++

```
DECLARE_INTERFACE_(IDataModelNameBinder, IUnknown)
{
   STDMETHOD(BindValue)(_In_ IModelObject* contextObject, _In_ PCWSTR name,
_COM_Errorptr_ IModelObject** value, _COM_Outptr_opt_result_maybenull_
IKeyStore** metadata) PURE;
   STDMETHOD(BindReference)(_In_ IModelObject* contextObject, _In_ PCWSTR
name, _COM_Errorptr_ IModelObject** reference,
_COM_Outptr_opt_result_maybenull_ IKeyStore** metadata) PURE;
   STDMETHOD(EnumerateValues)(_In_ IModelObject* contextObject, _COM_Outptr_
IKeyEnumerator** enumerator) PURE;
   STDMETHOD(EnumerateReferences)(_In_ IModelObject* contextObject,
_COM_Outptr_ IKeyEnumerator** enumerator) PURE;
}
```

BindValue

BindValue 方法根据一组绑定规则对给定对象执行等效的 contextObject.name。 此绑定的结果为 值。 作为一个值，基础脚本提供程序不能使用 值来执行赋回名称。

BindReference

BindReference 方法类似于 BindValue，因为它还根据一组绑定规则对给定对象执行等效的 contextObject.name。 但是，此方法的绑定结果是引用而不是值。 作为引用，脚本提供程序可以利用引用来执行赋回名称。

EnumerateValues

EnumerateValues 方法枚举一组名称和值，这些名称和值将根据 BindValue 方法的规则绑定到对象。 与 IModelObject 上的 EnumerateKeys、EnumerateValues 和类似方法不同，后者可能返回多个具有相同值的名称，（基类、父模型等），此枚举器将仅返回将与 BindValue 和 BindReference 绑定的特定名称集。 名称永远不会重复。 请注意，与调用 IModelObject 方法相比，通过名称绑定器枚举对象的成本要高得多。

EnumerateReferences

EnumerateReferences 方法枚举名称和对它们的引用集，这些名称和引用将根据 BindReference 方法的规则绑定到对象。 与 IModelObject 上的 EnumerateKeys、EnumerateValues 和类似方法不同，后者可能返回多个具有相同值的名称，（基类、父模型等），此枚举器将仅返回将与 BindValue 和 BindReference 绑定的特定名称集。 名称永远不会重复。 请注意，与调用 IModelObject 方法相比，通过名称绑定器枚举对象的成本要高得多。

调试器数据模型 C++ 脚本调试接口

数据模型中脚本提供程序的基础结构还提供了有关调试脚本的概念。 任何希望向调试主机和托管数据模型的调试器应用程序公开调试功能的脚本都可以通过让可调试脚本实现 IDataModelScriptDebug 接口以及 IDataModelScript 接口来实现此目的。 脚本上存在此接口向基础结构表明该接口可调试。

虽然 IDataModelScriptDebug 接口是访问脚本提供程序的调试功能的起点，但它是由一组提供整体调试功能的接口联接起来的。

调试接口包括：

接口	说明
IDataModelScriptDebug	脚本提供程序为使脚本可调试而必须提供的核心接口。 如果脚本可调试，则 IDataModelScript 接口的实现类必须为 IDataModelScriptDebug 的 QueryInterface。
IDataModelScriptDebugClient	希望提供脚本调试功能的用户界面实现 IDataModelScriptDebugClient 接口。 脚本提供程序利用此接口 (来回传递调试信息，例如发生的事件、断点等...)

接口	说明
IDataModelScriptDebugStack	脚本提供程序实现此接口以向脚本调试器公开调用堆栈的概念。
IDataModelScriptDebugStackFrame	脚本提供程序实现此接口以公开调用堆栈中特定堆栈帧的概念。
IDataModelScriptDebugVariableSetEnumerator	脚本提供程序实现此接口以公开一组变量。 此集可以表示函数的参数集、局部变量集或特定范围内的变量集。 确切的含义取决于获取接口的方式。
IDataModelScriptDebugBreakpoint	脚本提供程序实现此接口以公开脚本中特定断点的概念和控制。
IDataModelScriptDebugBreakpointEnumerator	脚本提供程序实现此操作以枚举脚本中当前存在的所有断点 (无论是否) 。

常规管理接口包括：

接口	说明
IDataModelScriptProvider	脚本提供程序必须实现的核心接口。 这是向数据模型管理器的脚本管理器部分注册的接口，以便播发提供程序对特定类型脚本的支持并针对特定文件扩展名注册
IDataModelScript	由提供程序管理的特定脚本的抽象。 加载或正在编辑的每个脚本都有单独的 IDataModelScript 实例
IDataModelScriptClient	脚本提供程序用于将信息传达给用户界面的客户端接口。 脚本提供程序不实现此接口。 托管希望使用脚本提供程序的数据模型的应用程序会使用脚本提供程序。 脚本提供程序将调用脚本客户端的方法来报告状态、错误等...
IDataModelScriptHostContext	脚本提供程序用作脚本内容的容器的主机接口。 除了脚本对调试器应用程序的对象模型执行的操作之外，脚本的内容如何呈现，由特定的调试主机决定。 此接口允许脚本提供程序获取有关放置其内容的位置的信息。
IDataModelScriptTemplate	脚本提供程序可以提供一个或多个模板，用作用户创作脚本的起点。 提供内置编辑器的调试器应用程序可以使用提供程序通过此接口播发的模板内容预填充新脚本。
IDataModelScriptTemplateEnumerator	脚本提供程序实现的枚举器接口，用于播发它支持的所有各种模板。

接口	说明
IDataModelNameBinder	一个名称绑定器 - 一个对象，它可以将上下文中的名称与值相关联。 对于给定的表达式（如“foo.bar”），名称绑定器能够在对象“foo”的上下文中绑定名称“bar”，并生成值或对它的引用。 名称绑定器通常不由脚本提供程序实现;相反，可以从数据模型获取默认绑定程序，并由脚本提供程序使用。

使脚本可调试：IDataModelScriptDebug

任何可调试的脚本都通过实现 IDataModelScript 的同一组件上存在 IDataModelScriptDebug 接口来指示此功能。 调试主机或托管数据模型的调试程序应用程序对此接口的查询指示存在调试功能。

IDataModelScriptDebug 接口定义如下。

C++

```
DECLARE_INTERFACE_(IDataModelScriptDebug, IUnknown)
{
   STDMETHOD_(ScriptDebugState, GetDebugState)() PURE;
   STDMETHOD(GetCurrentPosition)(_Out_ ScriptDebugPosition *currentPosition,
_Out_opt_ ScriptDebugPosition *positionSpanEnd, _Out_opt_ BSTR *lineText)
PURE;
   STDMETHOD(GetStack)(_COM_Outptr_ IDataModelScriptDebugStack **stack)
PURE;
   STDMETHOD(SetBreakpoint)(_In_ ULONG linePosition, _In_ ULONG
columnPosition, _COM_Outptr_ IDataModelScriptDebugBreakpoint **breakpoint)
PURE;
   STDMETHOD(FindBreakpointById)(_In_ ULONG64 breakpointId, _COM_Outptr_
IDataModelScriptDebugBreakpoint **breakpoint) PURE;
   STDMETHOD(EnumerateBreakpoints)(_COM_Outptr_
IDataModelScriptDebugBreakpointEnumerator **breakpointEnum) PURE;
   STDMETHOD(GetEventFilter)(_In_ ScriptDebugEventFilter eventFilter, _Out_
bool *isBreakEnabled) PURE;
   STDMETHOD(SetEventFilter)(_In_ ScriptDebugEventFilter eventFilter, _In_
bool isBreakEnabled) PURE;
   STDMETHOD(StartDebugging)(_In_ IDataModelScriptDebugClient *debugClient)
PURE;
   STDMETHOD(StopDebugging)(_In_ IDataModelScriptDebugClient *debugClient)
PURE;
}
```

GetDebugState

GetDebugState 方法 (返回脚本的当前状态，例如：它是否) 执行。 状态由 ScriptDebugState 枚举中的值定义。

GetCurrentPosition

GetCurrentPosition 的 方法返回脚本中的当前位置。 仅当脚本被分解到调试器中时，才能调用 GetScriptState 将返回 ScriptDebugBreak。 对此方法的任何其他调用都无效，并且将失败。

GetStack

GetStack 方法在中断位置获取当前调用堆栈。 仅当脚本被分解为调试器时，才能调用此方法。

SetBreakpoint

SetBreakpoint 方法在脚本中设置断点。 请注意，实现可以自由调整传入的行和列位置，以前进到适当的代码位置。 通过返回的 IDataModelScriptDebugBreakpoint 接口上的方法调用，可以检索放置断点的实际行号和列号。

FindBreakpointById

通过 SetBreakpoint 方法在脚本中创建的每个断点都分配有一个唯一标识符， (实现) 的 64 位无符号整数。 FindBreakpointById 方法用于从给定标识符获取到断点的接口。

EnumerateBreakpoints

EnumerateBreakpoints 方法返回一个枚举器，该枚举器能够枚举特定脚本中设置的每个断点。

GetEventFilter

GetEventFilter 方法返回是否为特定事件启用“中断事件”。 可能导致“事件中断”的事件由 ScriptDebugEventFilter 枚举的成员描述。

SetEventFilter

SetEventFilter 方法更改由 ScriptDebugEventFilter 枚举成员定义的特定事件的“中断事件”行为。 可以在 GetEventFilter 方法的文档中找到 (可用事件的完整列表，以及此枚举) 的说明。

StartDebugging

StartDebugging 方法“打开”特定脚本的调试器。 启动调试的行为不会主动导致任何执行中断或单步执行。 它只是使脚本可调试，并为客户端提供一组接口，以便与调试接口通信。

StopDebugging

StopDebugging 方法由要停止调试的客户端调用。 在 StartDebugging 成功执行后，可以随时调用此方法， (例如：在中断期间，脚本正在执行，等等...) 。调用会立即停止所有

调试活动，并在调用 StartDebugging 之前将状态重置回 。

调试接口：IDataModelScriptDebugClient

希望围绕脚本调试提供接口的调试主机或调试器应用程序必须通过脚本调试接口上的 StartDebugging 方法为脚本调试器提供 IDataModelScriptDebugClient 接口的实现。

IDataModelScriptDebugClient 是传递调试事件的通信通道，控制从脚本执行引擎传递到调试器接口。 它的定义如下。

```
C++

DECLARE_INTERFACE_(IDataModelScriptDebugClient, IUnknown)
{
   STDMETHOD(NotifyDebugEvent)(_In_ ScriptDebugEventInformation *pEventInfo,
_In_ IDataModelScript *pScript, _In_opt_ IModelObject *pEventDataObject,
_Inout_ ScriptExecutionKind *resumeEventKind) PURE;
}
```

NotifyDebugEvent

每当发生任何中断脚本调试器的事件时，调试代码本身会通过 NotifyDebugEvent 方法调用接口。 此方法是同步方法。 在接口从 事件返回之前，脚本不会继续执行。 脚本调试器的定义很简单：绝对没有需要处理的嵌套事件。 调试事件由称为 ScriptDebugEventInformation 的变体记录定义。 事件信息中的哪些字段有效主要由 DebugEvent 成员定义。 它定义由 ScriptDebugEvent 枚举成员描述的事件类型。

调用堆栈：IDataModelScriptDebugStack

当发生中断脚本调试器的事件时，调试接口需要检索中断位置的调用堆栈。 这是通过 GetStack 方法完成的。 此类堆栈通过 IDataModelScriptDebugStack 表示，如下所述定义。

请注意，整个堆栈可能跨多个脚本和/或多个脚本提供程序。 从对特定脚本调试接口上的 GetStack 方法的单个调用返回的调用堆栈应仅返回该脚本边界内的调用堆栈段。 如果同一提供程序的两个脚本交互，则脚本调试引擎完全有可能检索跨多个脚本上下文的调用堆栈。 GetStack 方法不应返回另一个脚本中的堆栈部分。 相反，如果可以检测到这种情况，则作为脚本边界帧的堆栈帧应通过该堆栈帧上的 IsTransitionPoint 和 GetTransition 方法的实现将自己标记为过渡帧。 调试器接口预期会将现有多个堆栈段的整体堆栈拼凑在一起。

必须以这种方式实现转换，否则调试接口可能会将有关局部变量、参数、断点和其他特定于脚本的构造的查询定向到错误的脚本上下文！ 这将导致调试器接口中出现未定义的行为。

C++

```
DECLARE_INTERFACE_(IDataModelScriptDebugStack, IUnknown)
{
   STDMETHOD_(ULONG64, GetFrameCount)() PURE;
   STDMETHOD(GetStackFrame)(_In_ ULONG64 frameNumber, _COM_Outptr_
IDataModelScriptDebugStackFrame **stackFrame) PURE;
}
```

GetFrameCount

GetFrameCount 方法返回调用堆栈的此段中的堆栈帧数。 如果提供程序可以检测不同脚本上下文中或不同提供程序中的帧，则它应通过在此堆栈段的入口帧上实现 IsTransitionPoint 和 GetTransition 方法向调用方表明这一点。

GetStackFrame

GetStackFrame 从堆栈段获取特定的堆栈帧。 调用堆栈具有从零开始的索引系统：发生中断事件的当前堆栈帧为帧 0。 当前方法的调用方为第 1 帧 (，以此类推) 。

检查中断时的状态：IDataModelScriptDebugStackFrame

在分解为脚本调试器时，可以通过调用 IDataModelScriptDebugStack 接口上的 GetStackFrame 方法来检索调用堆栈的特定帧，该方法表示发生中断的堆栈段。 返回以表示此帧的 IDataModelScriptDebugStackFrame 接口定义如下。

C++

```
DECLARE_INTERFACE_(IDataModelScriptDebugStackFrame, IUnknown)
{
   STDMETHOD(GetName)(_Out_ BSTR *name) PURE;
   STDMETHOD(GetPosition)(_Out_ ScriptDebugPosition *position, _Out_opt_
ScriptDebugPosition *positionSpanEnd, _Out_opt_ BSTR *lineText) PURE;
   STDMETHOD(IsTransitionPoint)(_Out_ bool *isTransitionPoint) PURE;
   STDMETHOD(GetTransition)(_COM_Outptr_ IDataModelScript
**transitionScript, _Out_ bool *isTransitionContiguous) PURE;
   STDMETHOD(Evaluate)(_In_ PCWSTR pwszExpression, _COM_Outptr_ IModelObject
**ppResult) PURE;
   STDMETHOD(EnumerateLocals)(_COM_Outptr_
IDataModelScriptDebugVariableSetEnumerator **variablesEnum) PURE;
   STDMETHOD(EnumerateArguments)(_COM_Outptr_
IDataModelScriptDebugVariableSetEnumerator **variablesEnum) PURE;
}
```

GetName

GetName 方法返回显示名称 (例如：此帧的函数名称) 。 此类名称将显示在调试器界面中向用户显示的堆栈回溯中。

GetPosition

GetPosition 方法返回堆栈帧表示的脚本中的位置。 仅当脚本位于包含此帧的堆栈表示的中断内时，才能调用此方法。 始终返回此帧中的线条和列位置。 如果调试器能够返回脚本中"执行位置"的跨度，则可以在 positionSpanEnd 参数中返回结束位置。 如果调试器无法执行此操作，则范围末尾的行和列值 (如果请求) 应设置为零。

IsTransitionPoint

IDataModelScriptDebugStack 接口表示调用堆栈的一段，该部分包含在一个脚本的上下文中。 如果调试器能够检测从一个脚本到另一个脚本 (或一个脚本提供程序到另一个) 的转换，则它可以通过实现 IsTransitionPoint 方法并根据需要返回 true 或 false 来指示这一点。 进入应用段的脚本的调用堆栈帧应被视为转换点。 所有其他帧都不是。

GetTransition

如果给定的堆栈帧是由 IsTransition 方法确定的过渡点 (请参阅文档了解) 转换点的定义，则 GetTransition 方法将返回有关转换的信息。 具体而言，此方法返回上一个脚本，即调用包含此 IDataModelScriptDebugStackFrame 的堆栈段所表示的脚本的脚本。

评估

Evaluate 方法在调用此方法的 IDataModelScriptDebugStackFrame 接口所表示的堆栈帧的上下文中计算脚本提供程序) (语言的表达式。 表达式计算的结果必须作为 IModelObject 封送出脚本提供程序。 当调试器处于中断状态时，必须能够获取生成的 IModelObject 上的属性和其他构造。

EnumerateLocals

EnumerateLocals 方法返回一 (个由 IDataModelScriptDebugVariableSetEnumerator 接口) 表示的变量集，该变量集由 IDataModelScriptDebugVariableSetEnumerator 接口表示，这些变量在调用此方法的 IDataModelDebugStackFrame 接口所代表的堆栈帧的上下文中。

EnumerateArguments

EnumerateArguments 方法返回一个变量集 (，该变量集由 IDataModelScriptDebugVariableSetEnumerator 接口表示，) 为调用此方法的 IDataModelScriptDebugStackFrame 接口所表示的堆栈帧中调用的函数的所有函数参数。

查看变量：IDataModelScriptDebugVariableSetEnumerator

要调试的脚本中的一组变量 (特定范围内的变量、函数的局部变量、函数的参数等...) 是否由通过 IDataModelScriptDebugVariableSetEnumerator 接口定义的变量集表示：

```
C++
```

```
DECLARE_INTERFACE_(IDataModelScriptDebugVariableSetEnumerator, IUnknown)
{
    STDMETHOD(Reset)() PURE;
    STDMETHOD(GetNext)(_Out_ BSTR *variableName, _COM_Outptr_opt_
IModelObject **variableValue, _COM_Outptr_opt_result_maybenull_ IKeyStore
**variableMetadata) PURE;
}
```

重置

Reset 方法将枚举器的位置重置为创建后紧接的位置，即在集合的第一个元素之前。

GetNext

GetNext 方法将枚举器移动到集中的下一个变量，并返回变量的名称、值以及与它关联的任何元数据。 如果枚举器到达集的末尾，则返回错误E_BOUNDS。 从 GetNext 方法返回E_BOUNDS标记后，在再次调用时，它将继续生成E_BOUNDS，除非进行了干预重置调用。

断点：IDataModelScriptDebugBreakpoint

脚本断点是通过给定脚本的调试接口上的 SetBreakpoint 方法设置的。 此类断点由唯一 ID 和 IDataModelScriptDebugBreakpoint 接口的实现表示，定义如下。

C++

```
DECLARE_INTERFACE_(IDataModelScriptDebugBreakpoint, IUnknown)
{
    STDMETHOD_(ULONG64, GetId)() PURE;
    STDMETHOD_(bool, IsEnabled)() PURE;
    STDMETHOD_(void, Enable)() PURE;
    STDMETHOD_(void, Disable)() PURE;
    STDMETHOD_(void, Remove)() PURE;
    STDMETHOD(GetPosition)(_Out_ ScriptDebugPosition *position, _Out_opt_
ScriptDebugPosition *positionSpanEnd, _Out_opt_ BSTR *lineText) PURE;
}
```

GetId

GetId 方法返回由脚本提供程序的调试引擎分配给断点的唯一标识符。 此标识符在包含脚本的上下文中必须是唯一的。 断点标识符对提供程序可能是唯一的;但是，这不是必需的。

IsEnabled

IsEnabled 方法返回是否启用断点。 禁用的断点仍然存在，并且仍在脚本的断点列表中，它只是暂时“关闭”。 应在已启用状态下创建所有断点。

启用

Enable 方法启用断点。 如果断点已禁用，则调用此方法后“命中断点”将导致中断调试器。

Disable

Disable 方法禁用断点。 在此调用之后，调用此方法后“命中断点”不会进入调试器。 断点虽然仍然存在，但被视为“已关闭”。

删除

Remove 方法从其包含列表中删除断点。 此方法返回后，断点在语义上不再存在。 表示断点的 IDataModelScriptDebugBreakpoint 接口在调用后被视为孤立。 在此调用之后，除了释放它之外，其他任何 (法律) 都做不了的事。

GetPosition

GetPosition 方法返回脚本中断点的位置。 脚本调试器必须在源代码中返回断点所在的行和列。 如果它能够这样做，它还可以通过填写由 positionSpanEnd 参数定义的结束位置来返回由断点表示的源范围。 如果调试器无法生成此范围，并且调用方请求它，则范围结束位置的“行”和“列”字段应填充为零，指示无法提供值。

断点枚举：IDataModelScriptDebugBreakpointEnumerator

如果脚本提供程序支持调试，则它还必须跟踪与每个脚本关联的所有断点，并且能够将这些断点枚举到调试接口。 断点的枚举器是通过给定脚本的调试接口上的 EnumerateBreakpoints 方法获取的，定义如下。

```
C++

DECLARE_INTERFACE_(IDataModelScriptDebugBreakpointEnumerator, IUnknown)
{
   STDMETHOD(Reset)() PURE;
   STDMETHOD(GetNext)(_COM_Outptr_ IDataModelScriptDebugBreakpoint
**breakpoint) PURE;
}
```

重置

Reset 方法将枚举器的位置重置为创建枚举器之后的位置，即第一个枚举断点之前的位置。

GetNext

GetNext 方法将枚举器向前移动到要枚举的下一个断点，并返回该断点的 IDataModelScriptDebugBreakpoint 接口。 如果枚举器已到达枚举的末尾，则返回

E_BOUNDS。 生成E_BOUNDS错误后，对 GetNext 方法的后续调用将继续生成 E_BOUNDS，除非已对 Reset 方法进行了干预调用。

另请参阅

本主题是一系列文章的一部分，该系列介绍了可从 C++ 访问的接口、如何使用接口生成基于 C++ 的调试器扩展，以及如何利用其他数据模型构造 (例如：JavaScript 或 NatVis) 从 C++ 数据模型扩展。

调试器数据模型 C++ 概述

调试器数据模型 C++ 接口

调试器数据模型 C++ 对象

调试器数据模型 C++ 的其他接口

调试器数据模型 C++ 的概念

代码命名空间

项目 • 2024/06/28

> **ⓘ 重要**
>
> 此接口正在积极开发中，并将不断变化。

总结

代码命名空间包含代码和反汇编属性。 它可以创建反汇编器对象，这些对象可以反汇编给定的地址或函数，并提供有关汇编的详细信息以及变量或源代码信息（如有）。

示例

有关如何使用该命名空间和对象的端到端示例，请参阅 GitHub 上的 CodeFlow 示例。

对象方法

展开表

名称	返回类型	Signature	说明
CreateDisassembler	反汇编程序	CreateDisassembler([体系结构])	创建指定体系结构的反汇编程序对象。 体系结构可以是“ARM”、“ARM64”、“X64”或“X86”之一。 如果未指定体系结构，则假定为 X64。
TraceDataFlow	指令集合	TraceDataFlow([地址])	查看指定的*地址*（如未指定地址，则查看当前指令指针）处的指令及其所有源操作数。 此方法会在函数的控制流中逆向查找任何影响跟踪指令源操作数的指令。 **此方法需要加载 CodeFlow.js 示例 中的 CodeFlow 扩展。**

注解

CreateDisassembler 暂时默认为 "X64"，但在某些时候，这一行为会改为在当前线程的指令指针上提取模块的体系结构。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

反汇编程序对象

项目 • 2023/06/15

总结

反汇编程序对象允许对特定体系结构的代码进行反汇编。

对象方法

名称	返回类型	签名	说明
DisassembleBlocks	基本块的集合	DisassembleBlocks (地址)	在 *地址* 处开始反汇编，并返回基本块的 集合 。 此处的反汇编是逐条指令从 *地址* 线性转发的。 由于这不是对函数执行完整的流分析，因此完全有可能跳入此方法返回的块中间。 每个出口点只有一个：然而。
DisassembleInstructions	指令集合	disassembleInstructions (地址)	开始在 *地址*处进行反汇编。
DisassembleFunction	基本块的集合	DisassembleFunction (地址)	假设函数从 *地址*开始，这会对该函数执行完整的流分析。 结果是具有一个入口点和一个退出点的基本块的集合 。
GetRegister	register	GetRegister (regId)	从给定的寄存器 ID 返回一个 register 对象。

注解

如果为反汇编函数提供完整的符号信息，此处提供的反汇编器具有明显更好的反汇编输出，（例如：它将利用地址和操作数大小来确定）接触结构/联合的字段。

给定的反汇编程序实例可能会缓存大量数据，以便提供更好的体验。

基本块对象

项目 • 2023/06/15

总结

基本块是代码区域， (通常) 一个入口点和一个退出点。 反汇编者的 DisassembleBlocks 和 DisassembleFunction 方法都返回基本块的集合。 DisassembleBlocks 方法对基本块执行简单的分析，并可能导致具有多个入口点的块。 DisassembleFunction 将对函数执行完整的流分析，从而生成具有单个入口和单个退出的基本块。

对象属性

名称	说明
StartAddress	基本块的起始地址。
EndAddress	基本块的结束地址。 块由半打开集 [*StartAddress*， *EndAddress*) 定义。
说明	基本块中的指令对象的集合。
InboundControlFlows	此属性仅在基本块上存在，这些块是完整流分析 (的结果，例如：*DisassembleFunction*) 。 它是控制流对象的集合，这些对象描述哪些其他块具有指向此块的入站控制流链接。
OutboundControlFlows	此属性仅在基本块上存在，这些块是完整流分析 (的结果，例如：*DisassembleFunction*) 。 它是控制流对象的集合，用于描述从此块到函数中其他块的出站控制流链接。

指令对象

项目 • 2023/06/15

总结

指令对象描述单个计算机指令，并通过基于指令的反汇编或作为 基本块 对象内容的一部分返回。

对象属性

名称	说明
地址	计算机指令的地址。
属性	说明有关指令的详细信息的指令属性对象。
CodeBytes	表示构成计算机指令的字节的字节数组。
Length	指令在内存中占用的字节数。
LiveVariables	动态变量对象的集合，描述编译器优化器为此特定位置的变量发出的数据。
操作数	描述指令操作数的操作数对象的集合。
SourceInformation	描述计算机指令与更高级别源代码之间的关系的 源信息 对象。
SourceDataFlow	函数中的指令对象的集合，构成计算机指令的源操作数的数据流。 **此方法需要加载 CodeFlow 扩展 。**

指令属性对象

项目 • 2023/06/15

总结

指令对象的 *Attributes* 属性包含指令的某些详细信息的说明。

对象属性

名称	说明
IsBranch	指示指令是否为任何类型的分支指令。
IsConditional	指示指令的结果是否为条件 (例如：条件分支) 。
IsCall	指示指令是否为任何类型的调用指令。
IsReturn	指示指令是否为任何类型的返回指令。

操作数对象

项目 • 2023/06/15

总结

计算机指令的各个操作数由操作数对象描述。 操作数的字符串转换将是整个指令的字符串转换的一部分。

对象属性

名称	说明
属性	操作数特性对象描述操作数的某些方面。
寄存器	寄存器对象的集合，这些对象描述操作数使用的所有寄存器。

操作数属性对象

项目 • 2023/06/15

摘要

机器指令的操作数由操作数属性对象描述。

对象属性

名称	说明
HasImmediate	指示操作数是否具有作为操作数的一部分的即时值。
IsInput	指示操作数是否为指令的数据源， (指令) 的任何操作的输入。
IsOutput	指示操作数是否为指令的数据目标 (指令) 执行的任何操作的输出。
IsMemoryReference	指示操作数是否为内存引用。
IsImmediate	指示操作数是否为即时值。 此类操作数也将 *HasImmediate* 设置为 true。
IsRegister	指示操作数是否只是寄存器。

注册对象

项目 • 2023/06/15

总结

由计算机指令中的操作数使用的寄存器由寄存器对象描述。 寄存器对象的字符串转换是寄存器的名称。

对象属性

名称	说明
ID	寄存器的特定体系结构) 域中 (唯一标识符。

注解

目前，只能通过字符串转换获取寄存器的名称。

动态变量对象

项目 • 2023/06/15

总结

使用完整的符号信息操作时，每个指令都包含有关变量的信息集合，这些变量位于该特定指令及其所在位置。

对象属性

名称	说明
LocationKind	描述变量存储在 (中的位置类型的字符串，例如：“Register”、“RegisterRelative”等...)
Offset	与存储变量的位置的偏移量。 例如，对于存储在 [rbp + 8] 处的变量，偏移量为 8。
注册	一个 register 对象，用于描述存储变量或相对于变量的寄存器。
VariableName	所描述的变量的名称。

源信息对象

项目 • 2023/06/15

总结

计算机指令通过源信息对象与源代码相关。

对象属性

名称	说明
FunctionName	所描述的函数的名称。
FunctionAddress	所描述的函数的基址。
FunctionOffset	所描述位置的 函数的偏移量。

控制流对象

项目 • 2023/06/15

总结

对于完全分析的反汇编，每个都 `basic block` 包含 InboundControlFlows 和 OutboundControlFlows 属性中的一组控制流对象。

对象属性

名称	说明
LinkedBlock	链接另一侧的基本块对象。 如果这是入站控制流，则指具有分支指令的基本块。 如果这是出站控制流，则指的是作为分支指令目标的基本块。
LinkKind	指示哪种类型的控制流导致两个块之间的链接，（例如：“FallThrough”或“Branch”）。
SourceInstruction	控制流链接的源。 这是分支指令或基本块中的最后一个指令。
TargetInstruction	控制流链接的目标。 这是分支目标或基本块的最后一个指令之后的指令，具有下降通过。

Collections 命名空间

项目 • 2023/06/15

总结

Collections 命名空间提供的方法允许创建和操作对象集合 (例如：迭代) 。 API 扩展使用以下属性和方法扩展现有 Collections 命名空间：

对象方法

名称	签名	说明
CreateArray	CreateArray ([value]...)	从传递给 方法的一组参数中创建可索引和可迭代的数组。 此方法可用于创建查询的起点或平展

FileSystem 命名空间

项目 • 2023/06/15

> **ⓘ 重要**
>
> 此接口正在积极开发中，将发生更改。

总结

FileSystem 命名空间提供用于操作文件系统的属性和方法。 这可以从 JavaScript 中用于读取或写入支持调试器扩展所需的文件。

示例

有关如何使用此命名空间和这些对象的简单端到端示例，请在 GitHub 上检查示例 - https://github.com/Microsoft/WinDbg-Samples/tree/master/FileSystem

对象方法

名称	返回类型	签名	说明
CreateFile	文件	CreateFile (路径，[处置])	在指定路径处创建一个新文件，并打开该文件进行写入。 *处置* 可以是“OpenExisting”、“CreateNew”或“CreateAlways”之一。
CreateTempFile	文件	CreateTempFile ()	在 %TEMP% 文件夹中创建新的临时文件，并打开该文件进行写入。
CreateTextReader	文本读取器	CreateTextReader (文件 |path，[encoding])	从给定 的文件 对象或路径创建文本读取器，该对象或路径将读取指定编码的文本。 编码可以是“Ascii”、“Utf8”或“Utf16”之一。 如果未指定，则默认为“Ascii”。
CreateTextWriter	Text Writer — 文本编写器	CreateTextWriter (文件 |path，[encoding])	从给定 的文件 对象或路径创建文本编写器，该对象或路径将写入指定编码的文本。 编码可以是“Ascii”、“Utf8”或“Utf16”之一。 如果未指定，则默认为“Ascii”。
DeleteFile		DeleteFile (路径)	删除指定路径中的文件。

名称	返回类型	签名	说明
FileExists	是或否	FileExists (路径)	对于给定路径中是否存在文件，返回 true 或 false
OpenFile	文件	OpenFile (路径)	在指定路径处打开一个文件以供读取。

对象属性

名称	说明
CurrentDirectory	一个 目录 对象，表示调试器进程的当前工作目录。
TempDirectory	一个 目录 对象，表示调试器进程的 %TEMP% 目录。

目录对象

项目 • 2023/06/15

总结

目录对象表示和操作文件系统上的目录。

对象方法

名称	返回类型	签名	说明
CreateFile	文件	CreateFile (relativePath, [disposition])	使用给定的处置在 目录中创建文件。 处置可以是“OpenExisting”、“CreateNew”或“CreateAlways”之一。
CreateSubDirectory	directory	CreateSubDirectory (名称)	在 目录中创建新的子目录。
删除		Delete ()	如果子目录为空，则删除子目录。
OpenFile	文件	OpenFile (relativePath)	打开一个现有文件，以便从目录中读取。

对象属性

名称	说明
文件	目录中的文件集合。
子	目录中的目录集合。

File 对象

项目 • 2023/03/18

摘要

文件对象用于打开、编辑和操作文件系统上的文件。

对象方法

名称	返回类型	签名	说明
关闭		Close ()	关闭文件。
删除		Delete ()	删除文件
Open		打开 (处置)	使用给定 *的处置*打开文件。 *处置*可以是“OpenExisting”、“CreateNew”或“CreateAlways”之一。
ReadBytes	字节数组	ReadBytes (byteCount)	从文件中读取指定的字节数，并返回这些字节的可索引和可迭代数组。
WriteBytes		WriteBytes (bytes, [byteCount])	将指定的字节写入文件。 如果未提供 *byteCount*，则可以循环访问字节的每个*字节将写入*文件;否则，写入*字节*内的第一个 *byteCount* 字节。

对象属性

名称	说明
分机	包含文件扩展名的字符串。
名称	包含文件名称的字符串。
路径	包含文件的完全限定路径的字符串。
Position	当前读/写光标在文件中的位置。
大小	文件大小（字节）。

注解

如果文件是从垃圾回收环境（如 JavaScript 脚本）中使用的，则应在不再使用时关闭这些文件，而不是等待垃圾回收导致文件关闭。

文本读取器对象

项目 • 2023/06/14

总结

文本读取器对象从文件中读取文本。

对象方法

名称	签名	说明
ReadLine	ReadLine ()	从文件中读取，直到下一行 (或文件结尾标记)，并将读取作为字符串返回。
ReadLineContents	ReadLineContents ()	返回字符串的可迭代集合。 集合中的每个字符串表示文件中的一行。

文本编写器对象

项目 • 2023/06/15

总结

文本编写器对象将给定编码的文本写入文件。

对象方法

名称	签名	说明
写入	写入 (对象)	将给定对象的字符串转换写入文件，而不用换行符。
WriteLine	WriteLine (对象)	将给定对象的字符串转换写入文件，后跟换行符。
WriteContents	WriteContents (对象)	循环访问 *对象* ，并将每个迭代元素的字符串转换写入文件，两者之间没有换行符。
WriteLineContents	WriteLineContents (对象)	循环访问 *对象* ，并将每个迭代元素的字符串转换写入文件，并在每个元素之间加上一条换行符。

术语表

项目 • 2024/03/20

此术语表包含与 Microsoft Windows 调试工具相关的术语和首字母缩略词。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

A

项目 • 2024/02/22

可访问
如果暂停当前目标，则调试会话*可访问*。

实际处理器类型
目标计算机中物理处理器的类型。

另请参阅有效的处理器类型、执行处理器类型。

有关详细信息，请参阅目标信息。

任意异常筛选器
已手动添加到引擎的事件筛选器列表中的异常筛选器。

另请参阅特定异常筛选器。

有关详细信息，请参阅事件筛选器。

B

项目 • 2024/02/14

蓝屏
出现 bug 后显示的蓝色字符模式屏幕检查。

breakpoint
目标或目标操作中的位置，将在触发时引发事件。

有关详细信息，请参阅 Using Breakpoints。

断点 ID
断点的唯一标识符。

有关详细信息，请参阅 Using Breakpoints。

断点类型
用于实现断点的方法。 有两种类型的断点：处理器断点和软件断点。

中断状态
影响调试器引擎在事件发生后执行方式的设置。 中断状态指示事件是否应闯入调试器、向调试器控制台打印通知或被忽略。 中断状态是事件筛选器的一部分。

另 *请参阅处理状态*。

有关详细信息，请参阅控制异常和事件和事件筛选器的主题。

bug 检查
当 Windows 遇到硬件问题、操作所需的数据不一致或其他严重错误时，它会关闭并在蓝色字符模式屏幕上显示错误信息。

此关闭称为 bug 检查、内核错误、系统崩溃、停止错误或偶尔捕获。 屏幕显示本身称为蓝屏或停止屏幕。 屏幕上显示的最重要信息是一个消息代码，它提供有关崩溃的信息;这称为 bug 检查代码或停止代码。

WinDbg 或 KD 可以配置系统，以便在发生此类错误时自动联系调试器。 或者，可以指示系统在出现此类错误时自动重新启动。

有关详细信息，请参阅 Bug 检查（蓝屏）。

bug 检查代码
指示特定类型的 bug 的十六进制代码检查 。

C

项目 • 2024/03/08

C++ 表达式
可由 C++ 表达式计算器计算的表达式。

C 调用堆栈
请参阅调用堆栈。

调用堆栈
每个线程的堆栈帧集，包含表示线程执行的函数调用。 每次进行函数调用时，都会将一个新的堆栈帧推到堆栈的顶部。 当该函数返回时，堆栈帧将从堆栈中弹出。

有时称为或简称为 。

回调对象
请参阅事件回调、输入回调和输出回调。

检查生成

不再为 Windows 提供已检查的版本。 使用驱动程序验证程序和 GFlags 等工具来检查驱动程序代码。

已检查的生成包含额外的错误检查、参数验证和调试信息，这些信息在免费生成中不可用。

尽管检查生成提供了额外的保护，但它使用的内存和磁盘空间比免费生成要多。 系统和驱动程序性能较慢，因为由于参数检查和诊断消息的输出而执行了额外的代码路径，并且使用了一些内核函数的替代实现。

不要将检查生成的 Windows 与在 Windows 驱动程序工具包 (WDK) 的某个检查生成环境中生成的驱动程序混淆。

子符号
包含在另一个符号中的符号。 例如，结构中某个成员的符号是该结构的符号的子级。

client
请参阅客户端对象。

客户端对象
客户端对象用于与调试器引擎交互。 它保持每个客户端的状态，并为调试器引擎 API 中的顶级接口提供实现。

客户端线程
在其中创建客户端对象的线程。 通常，客户端的方法只能从此线程调用。 调试器引擎使用此线程对向客户端注册的回调对象进行所有调用。

代码断点
请参阅软件断点。

崩溃转储文件
一种文件，包含某些内存区域的快照以及与应用程序或操作系统相关的其他数据。 可以存储故障转储文件，然后在以后用于调试应用程序或操作系统。

应用程序崩溃时，Windows 可以创建用户模式崩溃转储文件，当 Windows 自身崩溃时，可以通过特殊的 Windows 例程创建内核模式崩溃转储文件。 有几种不同类型的崩溃转储文件。

当前进程
调试器引擎当前正在控制的进程。 当事件发生时，当前进程被设置为事件进程。

当前目标
调试器引擎当前正在控制的目标。 当事件发生时，当前目标被设置为事件目标。

当前线程
调试器引擎当前正在控制的线程。 当事件发生时，当前线程被设置为事件线程。

D

项目 • 2023/06/15

数据断点
请参阅处理器断点。

DbgEng 扩展
基于 dbgeng.h 头文件中原型的调试器扩展。 这些扩展使用调试器引擎 API 与调试器引擎通信。

调试生成
请参阅已检查的生成。

debuggee
请参阅目标。

调试
使用调试器引擎的完整功能的调试器引擎应用程序。 例如，WinDbg、CDB、NTSD 和 KD 是调试器。

调试器控制台
一个虚构的实体，表示调试器引擎输入的源及其输出的目标。 实际上，调试器引擎仅使用输入和输出回调，并且不了解用于实现它们的内容。

通常，从“调试器命令”窗口接收输入，并将输出发送到同一窗口。 但是，输入和输出回调可以为输出提供许多其他输入源和目标，例如远程连接、脚本文件和日志文件。

调试器引擎
用于操作调试目标的库。 其接口基于 dbgeng.h 文件中的原型。 它由调试器、扩展和调试器引擎应用程序使用。

调试器引擎 API
用于控制调试器引擎行为的强大接口。 它可由调试器、DbgEng 扩展和调试器引擎应用程序使用。 此接口的原型位于 dbgeng.h 头文件中。

调试器引擎应用程序
使用调试器引擎 API 来控制调试器引擎的独立应用程序。

调试器扩展
可在调试器内运行的外部函数。 每个扩展都是从称为调试器扩展 DLL 的模块导出的。 调试器引擎通过在 DLL 中调用调试器扩展的代码来调用调试器扩展。 某些调试器扩展随适用于 Windows 的调试工具一起提供。 可以编写自己的扩展来自动执行任意数量的调试器功能，或自定义调试器可访问的信息的输出。

也称为 ，或简称 。

调试器扩展 DLL
包含调试器扩展的 DLL。 当调试器引擎加载 DLL 时，这些扩展将可用于调试器。

调试器扩展库
请参阅调试器扩展 DLL。

调试客户端
充当代理的调试器引擎的实例，将调试器命令和 I/O 发送到调试服务器。

调试服务器
充当主机的调试器引擎实例，侦听来自调试客户端的连接。

调试会话
调试会话是运行软件调试程序（如 WinDbg、KD 或 CDB）来调试软件组件、系统服务、应用程序或操作系统的实际行为。 还可以针对内存转储文件运行调试会话进行分析。

调试会话在获取 时启动，并持续到所有目标被丢弃为止。

默认异常筛选器
应用于与任何其他异常筛选器不匹配的异常事件的事件筛选器。 默认异常筛选器是特定的异常筛选器。

休眠模式
调试程序正在运行，但没有目标会话或活动会话的状态。

下游存储
符号服务器创建的符号缓存。 通常，此缓存位于本地计算机上，而符号存储位于远程位置。 如果你有一系列符号服务器，则下游存储可以位于符号存储下游的任何计算机上。

转储文件
请参阅故障转储文件。

转储目标
正在调试的故障转储文件。

E

项目 • 2023/06/15

有效处理器类型
调试器处理目标计算机时使用的处理器类型。 有效的处理器类型会影响调试器设置断点、访问寄存器、获取堆栈跟踪以及执行其他特定于处理器的操作的方式。

发动机
请参阅调试器引擎。

事件
调试器引擎监视其目标中发生的一些事件。 其中包括触发的断点、异常、线程和进程创建、模块加载以及内部调试器引擎更改。

事件筛选器
一个规则集合，这些规则会影响在目标中发生事件后调试器引擎的运行方式。 有三种类型的事件筛选器：特定事件筛选器、特定异常筛选器和任意异常筛选器。

事件回调对象
已注册到客户端的 IDebugEventCallbacks 接口的实例。 每当发生事件时，引擎就会通知事件回调。

事件回调
请参阅事件回调对象。

事件进程
发生最后一个事件的进程。

事件目标
发生最后一个事件的目标。

事件线程
发生最后一个事件的线程。

执行处理器类型
当前处理器上下文中的处理器类型。

例外
由于执行特定计算机指令或指示其遇到错误的目标而导致的错误条件。 异常可以是与硬件或软件相关的错误。

异常是 ， 由其 标识。

异常代码
确定异常事件的类型的标识符。

异常筛选器
异常代码指定的异常事件的事件筛选器。

扩展
请参阅调试器扩展。

扩展命令
请参阅调试器扩展。

F

项目 • 2023/06/15

免费内部版本

每个基于 NT 的操作系统存在两个不同版本：

- Windows 的免费版本是操作系统的最终用户版本。 系统和驱动程序是使用完全优化构建的，调试断言被禁用，调试信息从二进制文件中去除。 免费系统和驱动程序更小、更快，并且占用的内存更少。
- Windows 的检查 (或调试) 版本用作测试和调试帮助。 有关详细信息，请参阅已检查的版本。

包含操作系统免费版本的分发媒体没有任何特殊标签 -- 换句话说，包含免费版本的 CD 将仅标有 Windows 版本名称，而不引用生成类型。

第一次机会异常
处理异常的第一次机会。 如果异常未在第一次机会上由任何处理程序处理，则会为处理程序提供第二次机会。

H

项目 • 2023/03/08

主持人
请参阅主机计算机。

处理状态
处理状态指定调试器引擎是否应将异常事件标记为 "已处理"。 处理状态是 "异常筛选器" 的一部分。

另请参阅中断状态。 有关详细信息，请参阅 控制异常和事件 和 事件筛选器

主机计算机
主计算机是运行调试会话的计算机。 所有调试器操作（如执行命令和扩展）和符号加载-在主计算机上执行。

在典型的双系统内核调试会话中，调试器正在连接到正在调试的目标计算机的主计算机上运行。

I

项目 • 2023/06/15

I/O 请求数据包 (IRP)
用于表示 I/O 请求和控制其处理的数据结构。 IRP 结构由标头和一个或多个堆栈位置组成。

图像
Windows 作为用户模式进程或 Windows 内核的一部分加载的可执行文件、DLL 或驱动程序。

另请参阅图像文件。

image 文件
从中加载图像的文件。

隐式进程
在内核模式调试中，该过程用于确定执行虚拟到物理地址转换时要使用的虚拟地址空间。 发生事件时，隐式进程设置为事件进程。

另请参阅隐式线程。

有关详细信息，请参阅 线程和进程。

隐式线程
在内核模式调试中，线程用于确定某些目标寄存器，包括帧偏移量和指令偏移量。 发生事件时，隐式线程设置为事件线程。

无法
当所有目标都执行时，调试会话 *不可访问* 。

初始断点
在调试会话开始附近、重新启动后或目标应用程序重启后自动出现的断点。

有关详细信息，请参阅 使用断点。

输入回调对象
已注册到客户端的 IDebugInputCallbacks 接口的实例。 每当调试器引擎需要输入时，它就会要求输入回调提供输入。

另请参阅输出回调。

有关详细信息，请参阅 使用 AMLI 调试器命令。

输入回调
请参阅输入回调对象。

中断
中断正常命令执行并将控制权转移到中断处理程序的条件。 需要处理器服务的 I/O 设备通常会启动中断。

中断请求级别 (IRQL)
中断的优先级排名。 每个处理器都有一个 IRQL 设置，线程可以提高或降低该设置。 在处理器 IRQL 设置或以下发生的中断将被屏蔽，不会干扰当前操作。 高于处理器 IRQL 设置的中断优先于当前操作。

K

项目 • 2023/06/15

KD 连接服务器
在某些形式的内核模式远程调试期间使用的代理。 它侦听来自智能客户端的连接，并按照这些远程客户端的要求执行内存、处理器或 Windows 操作。

另请参阅调试服务器。

有关详细信息，请参阅 KD 连接服务器 (内核模式) 。

内核
内核是 Windows 操作系统的一部分，用于管理和控制对硬件资源的访问。 它执行线程计划和调度、中断和异常处理以及多处理器同步。

内核错误
请参阅 bug 检查。

内核模式
内核模式代码有权访问系统的任何部分，并且不受限制，就像用户模式代码一样。 它可以访问在用户模式或内核模式下运行的任何其他进程的任何部分。

性能敏感型操作系统组件在内核模式下运行。 这样，它们可以与硬件交互，彼此交互，而无需上下文切换的开销。 所有内核模式组件都受到完全保护，不受在用户模式下运行的应用程序的阻止。 可以按如下方式对它们进行分组：

- 行政。

  它包含基本操作系统组件，例如内存管理、进程和线程管理、安全性、I/O、进程间通信。

- 内核。

  这会执行低级别功能，例如线程计划、中断和异常调度以及多处理器同步。 它还提供一组例程和执行人员用于实现更高级别的语义的基本对象。

- 硬件抽象层 (HAL) 。

  这将处理与硬件的所有直接接口。 因此，它将 Windows 内核、设备驱动程序和 Windows Executive 与特定于平台的硬件差异隔离开来。

- 窗口和图形子系统。

  这将实现图形用户界面 (GUI) 函数。

当某个进程错误地访问另一个应用程序或系统正在使用的内存部分时，内核模式进程缺乏限制会强制 Windows 停止整个系统。 这称为 bug 检查。

驻留在内核模式下的硬件设备或设备驱动程序故障通常是 bug 检查的罪魁祸首。

内核模式目标
请参阅目标计算机。

内核模式调试
一个调试器会话，其中目标在内核模式下运行。

L

项目 • 2023/06/15

实时内核模式调试
使用实时目标进行内核模式调试。

另请参阅实时用户模式调试。

实时目标
当前正在运行 (的目标应用程序或目标计算机，而不是转储目标) 。

实时用户模式调试
使用实时目标进行用户模式调试。

另请参阅实时用户模式调试。

本地上下文
请参阅范围。

本地调试
这指的是调试会话，其中调试器和要调试的应用程序驻留在同一台计算机上。

M

项目 • 2024/03/20

模块
目标进程中的图像。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

N

项目 • 2024/03/20

非分页池
系统内存的一部分，不会分页到磁盘。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

O

项目 • 2023/06/15

输出回调对象
已注册到客户端对象的 IDebugOutputCallbacks 接口的实例。 调试器引擎的所有输出都发送到输出回调。

输出回调
请参阅输出回调对象。

P

项目 • 2023/06/15

页表
将虚拟内存地址映射到物理内存地址的特定于进程的表。

页表条目 (PTE)
页表中的一项。

分页池
可以分页到磁盘的系统内存的一部分。

请注意，此术语不仅指实际上已分页到磁盘的内存，还包括操作系统允许分页的任何内存。

寻呼
一种虚拟内存操作，其中，当物理内存已满时，内存管理器将页从内存传输到磁盘。 当线程访问不在内存中的页时，会发生页 *错误* 。

父符号
包含在其他 *符号* 中的符号，例如，结构包含其成员。

另请参阅 *子符号*。

有关详细信息，请参阅 范围和符号组。

主客户端
已加入当前调试会话的客户端对象

有关详细信息，请参阅 客户端对象。

进程服务器
充当代理的调试器引擎的实例，侦听来自智能客户端的连接，并按照这些远程客户端的请求执行内存、处理器或 Windows 操作。

另请参阅 *调试服务器*。

有关详细信息，请参阅 进程服务器 (用户模式) 和进程服务器和智能客户端。

处理器断点
由处理器实现的断点。 调试器引擎指示目标的处理器插入此断点。

另请参阅软件断点。 另请参阅 *软件断点*。

有关详细信息，请参阅 使用断点。

R

项目 • 2023/06/15

远程调试
远程调试是使用远程连接来执行调试的做法。

由于用户模式调试通常在一台计算机上完成，因此远程用户模式调试通常使用两台计算机。 在此方案中，一台计算机包含目标应用程序和调试服务器，而另一台计算机包含调试客户端。 另一种方法是将目标应用程序和进程服务器放在一台计算机上，另一台计算机上的智能客户端。

由于内核模式调试通常在两台计算机上完成，因此远程内核模式调试需要三台计算机。 目标计算机是正在调试的计算机。 服务器附加到目标;它包含内核模式调试器。 客户端是远程控制会话的计算机。 通常，一台计算机是正在调试的计算机，另一台计算机包含调试服务器，第三台计算机包含调试客户端。

此外，还有其他远程调试方法：使用远程工具、使用 KD 连接服务器或使用中继器。 应选择的方法取决于有问题的计算机的配置和可用的连接。

有关详细信息，请参阅远程调试。

注册
CPU 中非常快速的临时内存位置。

注册上下文
包含所有处理器寄存器的完整处理器状态。

有关详细信息，请参阅 **注册上下文**。

零售版本
请参阅免费版本。

S

项目 • 2023/06/15

范围
定义局部变量的上下文。 范围有三个组件：堆栈帧、当前指令和寄存器上下文。

有时称为 *本地上下文* 或 *词法范围*。

第二次机会异常
处理异常的第二次机会。 仅当第一次机会未处理异常时，才提供此机会。

智能客户端
充当主机的调试器引擎的实例。 智能客户端已连接到进程服务器。 或 KD 连接服务器。

特定异常筛选器
引擎为其具有内置筛选器的异常的事件筛选器。 大多数特定的异常筛选器是指 (异常代码) 标识的特定类型的异常，但默认的异常筛选器也限定为特定的异常筛选器。

特定事件筛选器
不是异常的事件的事件筛选器。 特定事件筛选器在 DEBUG_FILTER_XXX 中列出。

特定筛选器
引擎为其具有内置筛选器的事件的事件筛选器。

软件断点
由调试器引擎临时修改目标的可执行代码实现的断点。 执行代码时会触发断点。 代码修改对调试器或调试器引擎 API 的用户不可见。

堆栈
请参阅调用堆栈。

堆栈帧
调用堆栈中的内存，其中包含单个函数调用的数据。 这包括返回地址、传递给函数的参数和局部变量。

停止代码
请参阅 bug 检查代码。

stop 错误
请参阅 bug 检查。

停止屏幕
请参阅蓝屏。

subregister
包含在另一个寄存器中的寄存器。 当子注册表更改时，包含子注册表的寄存器部分也会更改。

暂停
如果已阻止目标、进程或线程执行，则会暂停该目标、进程或线程。

象征
一个调试信息单元，提供有关调试会话中目标的解释性信息。 符号的示例包括 (局部和全局) 、函数、类型和函数条目的变量。 有关符号的信息可以包括名称、类型（（如果适用））、存储符号的地址或地址，以及任何父符号或子符号。 此信息存储在符号文件中，通常在模块本身中不可用。

当符号文件不可用时，调试器引擎可以合成某些符号，（例如导出的符号），但这些符号通常只提供最少的信息。

符号文件
生成应用程序、库、驱动程序或操作系统时创建的补充文件。 符号文件包含运行二进制文件时实际不需要的数据，但在调试过程中非常有用。

符号组
一组符号，通常表示范围中的所有局部变量。

符号类型
有关符号表示形式的描述性信息，例如其在内存中的布局。 这是编译器用于生成代码以操作符号的信息的一部分。 调试器引擎还使用它来操作符号。 符号类型在符号文件中分布。

合成模块
引擎视为模块的内存区域。 合成模块可能包含合成符号。

合成符号
引擎将其视为符号的内存地址。

系统崩溃
请参阅 bug 检查。

T

项目 • 2023/06/15

目标
目标应用程序、目标计算机或转储目标。

有时称为*调试对象* 。

目标应用程序
正在用户模式下调试的应用程序。

目标计算机
正在内核模式下调试的计算机。

目标 ID
目标的唯一标识符。

目标进程
在用户模式调试中，是作为目标应用程序的一部分的操作系统进程。

在内核模式调试中，表示内核的虚拟进程。

目标线程
在用户模式调试中，目标应用程序中的操作系统线程。

在内核模式调试中，表示处理器的虚拟线程。

线程上下文
切换线程时 Windows 保留的状态。 这类似于寄存器上下文，只是有一些额外的仅限内核的处理器状态是寄存器上下文的一部分，而不是线程上下文的一部分。 在内核模式调试期间，此额外状态可用作寄存器。

有关详细信息，请参阅范围和符号组。

传输层
这控制远程调试期间主机和目标计算机之间的通信。

陷阱
请参阅 bug 检查。

类型
请参阅符号类型。

U

项目 • 2023/06/15

用户模式
应用程序和子系统在 Windows 中以 *用户模式*运行。 在用户模式下运行的进程在其自己的虚拟地址空间中运行。 它们受到限制，无法直接访问系统的许多部分，包括系统硬件、未分配给它们使用的内存以及可能损害系统完整性的系统其他部分。 由于以用户模式运行的进程与系统和其他用户模式进程有效隔离，因此它们不能干扰这些资源。

用户模式进程可按以下类别分组：

- 系统进程

  这些函数执行重要功能，是操作系统不可或缺的一部分。 系统进程包括登录过程和会话管理器进程等项。

- 服务器进程

  这些是操作系统服务，例如事件日志和计划程序。 可以将其配置为在启动时自动启动。

- 环境子系统

  它们用于为应用程序创建完整的操作系统环境。 Windows 提供以下三个环境：Win32、POSIX 和 OS/2。

- 用户应用程序

  有五种类型：Win32、Windows 3.1、Microsoft MS-DOS、POSIX 和 OS/2。

用户模式调试
目标在用户模式下运行的调试器会话。

用户模式目标
请参阅目标应用程序。

V

项目 • 2023/06/15

虚拟进程
在内核模式调试中，调试器引擎将创建一个 *虚拟进程* 来表示目标的内核。

虚拟线程
在内核模式调试中，调试器引擎为目标计算机中的每个处理器创建 *一个虚拟线程* 。

W

项目 • 2023/06/15

WdbgExts API
由调试器引擎为扩展公开的接口。 它包含调试器引擎 API 的一部分功能，只能由扩展使用。

WdbgExts 扩展
基于 wdbgexts.h 头文件中原型的扩展。 这些扩展使用 WdbgExts API。

WOW64
64 位 Windows 中的仿真层，可以运行 32 位 Windows 应用程序。 调试在 64 位 Windows 上运行的 32 位应用程序时，必须区分应用程序本身和 WOW64 子系统。

调试器操作

项目 • 2023/04/14

本节内容：

- WinDbg 功能
- 使用 KD 和 NTKD 进行调试
- 使用 CDB 和 NTSD 进行调试
- 使用调试器命令
- 使用调试器扩展
- 远程调试
- 使用 WinDbg 图形界面 (经典)

使用 Windows 调试器进行调试 - WinDbg

项目 • 2023/04/14

本部分介绍如何使用 WinDbg 调试器执行基本调试任务。

以下主题中提供了详细信息：

- 使用 WinDbg 调试用户模式进程
- 使用 WinDbg 调试 UWP 应用
- 使用 WinDbg 打开转储文件
- 在 WinDbg 中设置符号和可执行映像路径
- 远程 WinDbg 功能
- 在 WinDbg 中配置异常和事件
- 在 WinDbg 中保存日志文件

使用 WinDbg 经典版调试用户模式进程

项目 • 2024/07/08

可以使用 WinDbg 附加到正在运行的进程或生成并附加到新进程。

附加到正在运行的进程

可通过多种方式使用 WinDbg 附加到正在运行的进程。 无论选择哪种方法，都需要进程 ID 或进程名称。 进程 ID 是操作系统分配的数字。 有关如何确定进程 ID 和进程名称的详细信息，请参阅 查找进程 ID。

WinDbg 菜单

当 WinDbg 处于休眠模式时，可以通过从“**文件**”菜单或按 **F6** 选择“**附加到进程**”来附加到正在运行的进程。

在 “**附加到进程** ”对话框中，选择要调试的进程，然后选择“ **确定**”。

命令提示符

在命令提示符窗口中，可以在启动 WinDbg 时附加到正在运行的进程。 使用以下命令之一：

- **windbg -p** *ProcessID*
- **windbg -pn** *ProcessName*

其中 *ProcessID* 是正在运行的进程 ID，或者 *ProcessName* 是正在运行的进程的名称。

有关命令行语法的详细信息，请参阅 **WinDbg 命令行选项**。

调试器命令窗口

如果 WinDbg 已在调试一个或多个进程，则可以在调试器命令窗口中使用 .attach（附加到进程）**命令附加到正在运行的进程。**

除非某些线程被冻结或挂起，否则调试器始终同时启动多个目标进程。

如果 .attach 命令成功，调试器将在下次调试器发出执行命令时附加到指定的进程。 如果在一行中多次使用此命令，则必须像使用此命令一样多次请求调试器执行。

非侵入性附加到正在运行的进程

如果要调试正在运行的进程，并且只干扰其执行，则应以非侵入方式调试进程。

WinDbg 菜单

当 WinDbg 处于休眠模式时，可以通过从“文件”**菜单或按 F6 选择“附加到进程”来非侵入性地调试正在运行**的进程。

出现“附加到进程”对话框时，选中“**非侵入性**”复选框。 然后选择包含所需进程 ID 和名称的行。 （还可以在 中 **输入进程 ID进程 ID** 框。）最后，选择“ **确定**”。

命令提示符

在命令提示符窗口中，启动 WinDbg 时，可以非侵入性地附加到正在运行的进程。 使用以下命令之一：

- **windbg -pv -p** *ProcessID*
- **windbg -pv -pn** *ProcessName*

还有其他几个有用的命令行选项。 有关命令行语法的详细信息，请参阅 **WinDbg 命令行选项**。

调试器命令窗口

如果调试器已处于活动状态，则可以在调试器命令窗口中使用 .attach -v （附加到进程）**命令以非侵入性方式调试正在运行的进程。**

如果调试器已经以侵入性的方式调试了一个或多个进程，则可以使用 **.attach** 命令。 如果 WinDbg 处于休眠状态，则不能使用此命令。

如果 .attach -v 命令成功，调试器将在下次调试器发出执行命令时调试指定的进程。 由于在非侵入性调试期间不允许执行，因此调试器不能一次非侵入性地调试多个进程。 此限制还意味着使用 **.attach -v** 命令可能会使现有入侵调试会话不那么有用。

生成新进程

WinDbg 可以启动用户模式应用程序，然后调试应用程序。 应用程序按名称指定。 调试器还可以自动附加到子进程（原始目标进程启动的其他进程）。

调试器创建的进程（也称为生成进程）的行为与调试器不创建的进程的行为略有不同。

调试器创建的进程不使用标准堆 API，而是使用特殊的调试堆。 可以使用 _NO_DEBUG_HEAP 环境变量 或 **-hd** 命令行选项强制生成进程使用标准堆而不是调试堆。

此外，由于目标应用程序是调试器的子进程，因此它将继承调试器的权限。 此权限可能使目标应用程序能够执行无法执行的某些操作。 例如，目标应用程序可能能够影响受保护的进程。

WinDbg 菜单

当 WinDbg 处于休眠模式时，可以通过从“**文件**”菜单中选择“**打开可执行文件**”或按 Ctrl+E 来生成新进程。

出现“打开可执行文件”对话框时，在“文件名”**框中输入可执行文件**的完整路径，或使用“**查找**”列表选择所需的路径和文件名。

如果要将任何命令行参数用于用户模式应用程序，请在“参数”**框中输入它们。** 如果要从默认目录更改起始目录，请在“开始”**目录框中输入目录路径。** 如果希望 WinDbg 附加到子进程，请选中“ **调试子进程**”**复选框** 。

选择后，选择“ **打开**”。

命令提示符

在命令提示符窗口中，可以在启动 WinDbg 时生成新进程。 使用以下命令：

windbg [**-o**] ProgramName *[Arguments]*

-o 选项会导致调试器附加到子进程。 还有其他几个有用的命令行选项。 有关命令行语法的详细信息，请参阅 **WinDbg 命令行选项**。

调试器命令窗口

如果 WinDbg 已在调试一个或多个进程，则可以在调试器命令窗口中使用 .create（创建进程）**命令创建新进程。**

除非某些线程被冻结或挂起，否则调试器将始终同时启动多个目标进程。

如果 .create 命令成功，调试器将在下次调试器发出执行命令时创建指定的进程。 如果在一行中多次使用此命令，则必须像使用此命令一样多次请求调试器执行。

可以在 .create 之前使用 **.createdir （Set Created Process Directory）** 命令来控制应用程序的起始目录。 可以使用 **.createdir -I** 命令或 **-noinh** 命令行选项来控制目标应用程序是否继承调试器的句柄。

可以使用 .childdbg（调试子进程）**命令激活或停用子进程的**调试。

重新附加到进程

如果调试器停止响应或冻结，可以将新调试器附加到目标进程。 有关如何在此情况下附加调试器的详细信息，请参阅 重新附加到目标应用程序。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用 WinDbg 调试 UWP 应用

项目 • 2024/05/27

可以使用 WinDbg 调试通用 Windows 平台 (UWP) 应用。 这种方法通常用于无法使用 Visual Studio 内置调试器完成调试任务的高级方案。 有关在 Visual Studio 中进行调试模式的详细信息，请参阅在 Visual Studio 中进行调试。

附加到 UWP 应用

附加到 UWP 进程与附加到用户模式进程相同。 例如，在 WinDbg 中，可以选择**从文件附加到进程**菜单或按 F6 键来附加到正在运行的进程。 有关详细信息，请参阅 使用 WinDbg 调试用户模式进程。

UWP 应用不会以与未调试时相同的方式暂停。 要显式挂起/恢复 UWP 应用，则可以使用 .suspendpackage 和 .resumepackage 命令（详情如下）。 有关 UWP 应用使用的进程生命周期管理 (PLM) 的一般信息，请参阅应用生命周期和启动、恢复和后台任务。

启动和调试 UWP 应用

-plmPackage 和 -plmApp 命令行参数指示调试器在调试器下启动一个应用。

控制台

```
windbg.exe -plmPackage <PLMPackageName> -plmApp <ApplicationId>
[<parameters>]
```

由于单个包中可以包含多个应用，因此需要使用 <PLMPackage> 和 <ApplicationId> 两个参数。 这是参数的摘要。

展开表

参数	描述
<PLMPackageName>	应用程序包的名称。 使用 .querypackages 命令列出所有 UWP 应用程序。 不要提供包位置的路径，只提供包名称。
<ApplicationId>	ApplicationId 位于应用程序清单文件中，可使用 .querypackage 或 .querypackages 命令查看，如本主题所述。

	有关应用程序清单文件的详细信息，请参阅应用程序包清单。
[<参数>]	传递给应用的可选参数。 并非所有应用都使用或需要参数。

HelloWorld 示例

为了演示 UWP 调试，本主题将使用创建一个“Hello, world”应用 (XAML) 中描述的 HelloWorld 示例。

要创建一个可行的测试应用，只需完成实验室的第三步。

查找完整包名称和 AppId

使用 .querypackages 命令查找完整的包名称和 AppId。 键入 .querypackages，然后使用 CRTL+F 在输出中搜索应用程序名称，如 HelloWorld。 使用 CTRL+F 查找条目时，将显示包全名，例如 *e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8* 和 *App* 的 AppId。

示例：

dbgcmd

```
0:000>  .querypackages
...
Package Full Name: e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
Package Display Name: HelloWorld
Version: 1.0.0.0
Processor Architecture: x86
Publisher: CN=user1
Publisher Display Name: user1
Install Folder: c:\users\user1\documents\visual studio
2015\Projects\HelloWorld\HelloWorld\bin\x86\Release\AppX
Package State: Unknown
AppId: App
...
```

在清单中查看基本包名称

要进行故障排除，可能需要在 Visual Studio 中查看基础包名称。

要在 Visual Studio 中查找基本包名称，请单击项目资源管理器中的 ApplicationManifest.xml 文件。 基本包名称将显示在打包选项卡下并显示为“包名称”。 默认情况下，包名称为 GUID，例如 *e24caf14-8483-4743-b80c-ca46c28c75df*。

要使用记事本查找基本包名称，请打开 ApplicationManifest.xml 文件并找到 **Identity Name** 标记。

XML

```
  <Identity
    Name="e24caf14-8483-4743-b80c-ca46c28c75df"
    Publisher="CN= User1"
    Version="1.0.0.0" />
```

在清单中查找应用 ID

要在已安装的 UWP 应用的清单文件中查找应用程序 ID，请查找*应用程序 ID* 条目。

例如，hello world 应用的应用程序 ID 是 *App*。

XML

```
<Application Id="App"
      Executable="$targetnametoken$.exe"
      EntryPoint="HelloWorld.App">
```

WinDbg 命令行示例

这是一个使用完整包名称和 AppId 在调试器下加载 HelloWorld 应用的命令行示例。

控制台

```
windbg.exe -plmPackage e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8 -plmApp App
```

在调试器下启动后台任务

后台任务可以通过命令行使用 TaskId 在调试器下显式启动。 为此，请使用 -plmPackage 和 -plmBgTaskId 命令行参数：

控制台

```
windbg.exe -plmPackage <PLMPackageName> -plmBgTaskId <BackgroundTaskId>
```

展开表

参数	描述

<PLMPackageName>	应用程序包的名称。 使用 .querypackages 命令列出所有 UWP 应用程序。 不要提供包位置的路径，只提供包名称。
<BackgroundTaskId>	可使用 .querypackages 命令来查找 BackgroundTaskId，如下所述。 有关应用程序清单文件的详细信息，请参阅应用程序包清单。

这是一个在调试器下加载 SDKSamples.BackgroundTask 代码的示例。

控制台

```
windbg.exe -plmPackage
Microsoft.SDKSamples.BackgroundTask.CPP_1.0.0.0_x64__8wekyb3d8bbwe -
plmBgTaskId {ee4438ee-22db-4cdd-85e4-8ad8a1063523}
```

可以尝试使用背景任务示例代码来熟悉 UWP 调试。 它可在背景任务示例中下载。

使用 .querypackages 命令来查找 BackgroundTaskId。 使用 CTRL-F 查找应用，然后找到 *后台任务 ID* 字段。 后台任务必须正在运行，才能显示相关的后台任务名称和任务 ID。

dbgcmd

```
0:000> .querypackages
...
Package Full Name:
Microsoft.SDKSamples.BackgroundTask.CPP_1.0.0.0_x86__8wekyb3d8bbwe
Package Display Name: BackgroundTask C++ sample
Version: 1.0.0.0
Processor Architecture: x86
Publisher: CN=Microsoft Corporation, O=Microsoft Corporation, L=Redmond,
S=Washington, C=US
Publisher Display Name: Microsoft Corporation
Install Folder: C:\Users\user1\Documents\Visual Studio
2015\Projects\Background_task_sample\C++\Debug\BackgroundTask.Windows\AppX
Package State: Running
AppId: BackgroundTask.App
Background Task Name: SampleBackgroundTask
Background Task Id: {ee4438ee-22db-4cdd-85e4-8ad8a1063523}
...
```

如果知道包全名，就可以使用 .querypackage 来显示*后台任务 ID* 字段。

还可以使用 PLMDebug 的 enumerateBgTasks 选项来查找 BackgroundTaskId。 有关 PMLDebug 实用工具的详细信息，请参阅 **PLMDebug**。

控制台

```
C:\Program Files\Debugging Tools for Windows (x64)>PLMDebug
/enumerateBgTasks
Microsoft.SDKSamples.BackgroundTask.CPP_1.0.0.0_x64__8wekyb3d8bbwe
Package full name is
Microsoft.SDKSamples.BackgroundTask.CPP_1.0.0.0_x64__8wekyb3d8bbwe.
Background Tasks:
SampleBackgroundTask : {C05806B1-9647-4765-9A0F-97182CEA5AAD}

SUCCEEDED
```

使用进程服务器 (DbgSrv) 远程调试 UWP 进程

所有 -plm* 命令都能与 dbgsrv 正常使用。 要使用 dbgsrv 进行调试，请使用 -premote 开关和 dbgsrv 的连接字符串：

控制台

```
windbg.exe -premote npipe:pipe=fdsa,server=localhost -plmPackage e24caf14-
8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8 -plmApp App
```

有关 -premote 选项的详细信息，请参阅进程服务器（用户模式）和进程服务器示例。

UWP 应用命令摘要

本部分概述了 UWP 应用调试器命令

收集包信息

.querypackage

.querypackage 显示 UWP 应用程序的状态。 例如，如果应用正在运行，它可以处于*活动*状态。

dbgcmd

```
.querypackage <PLMPackageName>
```

示例：

dbgcmd

```
0:000> .querypackage e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
Package Full Name: e24caf14-8483-4743-b80c-
```

```
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
Package Display Name: HelloWorld
Version: 1.0.0.0
Processor Architecture: x86
Publisher: CN=user1
Publisher Display Name: user1
Install Folder: c:\users\user1\documents\visual studio
2015\Projects\HelloWorld\HelloWorld\bin\x86\Release\AppX
Package State: Running
AppId: App
Executable: HelloWorld.exe
```

.querypackages

.querypackages 命令会列出所有已安装的 UWP 应用程序及其当前状态。

dbgcmd

```
.querypackages
```

示例：

dbgcmd

```
0:000> .querypackages
...
Package Full Name:
Microsoft.MicrosoftSolitaireCollection_3.9.5250.0_x64__8wekyb3d8bbwe
Package Display Name: Microsoft Solitaire Collection
Version: 3.9.5250.0
Processor Architecture: x64
Publisher: CN=Microsoft Corporation, O=Microsoft Corporation, L=Redmond,
S=Washington, C=US
Publisher Display Name: Microsoft Studios
Install Folder: C:\Program
Files\WindowsApps\Microsoft.MicrosoftSolitaireCollection_3.9.5250.0_x64__8we
kyb3d8bbwe
Package State: Unknown
AppId: App

Package Full Name: e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
Package Display Name: HelloWorld
Version: 1.0.0.0
Processor Architecture: x86
Publisher: CN=user1
Publisher Display Name: user1
Install Folder: c:\users\user1\documents\visual studio
2015\Projects\HelloWorld\HelloWorld\bin\x86\Release\AppX
Package State: Running
AppId: App
```

```
Executable: HelloWorld.exe

Package Full Name:
Microsoft.SDKSamples.BackgroundTask.CPP_1.0.0.0_x86__8wekyb3d8bbwe
Package Display Name: BackgroundTask C++ sample
Version: 1.0.0.0
Processor Architecture: x86
Publisher: CN=Microsoft Corporation, O=Microsoft Corporation, L=Redmond,
S=Washington, C=US
Publisher Display Name: Microsoft Corporation
Install Folder: C:\Users\user1\Documents\Visual Studio
2015\Projects\Background_task_sample\C++\Debug\BackgroundTask.Windows\AppX
Package State: Unknown
AppId: BackgroundTask.App

...
```

启动用于调试的应用

.createpackageapp

.createpackageapp 命令可启用调试并启动 UWP 应用程序。

```
dbgcmd

.createpackageapp <PLMPackageName> <ApplicationId> [<parameters>]
```

此表列出了 .createpackageapp 的参数。

展开表

参数	描述
<PLMPackageName>	应用程序包的名称。 使用 .querypackages 命令列出所有 UWP 应用程序。 不要提供包位置的路径，只提供包名称。
<ApplicationId>	如本主题前面所述，可使用 .querypackage 或 .querypackages 来查找 ApplicationId。 有关应用程序清单文件的详细信息，请参阅应用程序包清单。
[<参数>]	传递给应用程序的可选参数。 并非所有应用程序都需要或使用这些可选参数。

示例：

```
dbgcmd
```

```
.createpackageapp e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8 App
```

启用和禁用调试命令的使用

.enablepackagedebug

使用 .enablepackagedebug 命令可以调试 UWP 应用程序。 在调用任何挂起、恢复或终止函数之前，必须使用 .enablepackagedebug。

请注意，.createpackageapp 命令还可以调试应用程序。

dbgcmd

```
.enablepackagedebug <PLMPackageName>
```

示例：

dbgcmd

```
.enablepackagedebug e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

.disablepackagedebug

.disablepackagedebug 命令会禁用 UWP 应用程序的调试。

dbgcmd

```
.disablepackagedebug <PLMPackageName>
```

示例：

dbgcmd

```
.disablepackagedebug e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

启动和停止应用

请注意，挂起、恢复和终止会影响包中当前运行的所有应用。

.suspendpackage

.suspendpackage 命令可挂起 UWP 应用程序。

```
dbgcmd
```

```
.suspendpackage <PLMPackageName>
```

示例：

```
dbgcmd
```

```
0:024> .suspendpackage e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

.resumepackage

.resumepackage 命令可恢复 UWP 应用程序。

```
dbgcmd
```

```
.resumepackage <PLMPackageName>
```

示例：

```
dbgcmd
```

```
.resumepackage e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

.terminatepackageapp

.terminatepackageapp 命令可终止包中的所有 UWP 应用程序。

```
dbgcmd
```

```
.terminatepackageapp <PLMPackageName>
```

示例：

```
dbgcmd
```

```
.terminatepackageapp e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

后台任务

.activatepackagebgtask

.activatepackagebgtask 命令可启用调试并启动 UWP 后台任务。

```
dbgcmd

 .activatepackagebgtask <PLMPackageName> <bgTaskId>
```

示例：

```
dbgcmd

.activatepackagebgtask
Microsoft.SDKSamples.BackgroundTask.CPP_1.0.0.0_x64__8wekyb3d8bbwe
{C05806B1-9647-4765-9A0F-97182CEA5AAD}
```

用法示例

启动应用时附加调试器

假设有一个名为 HelloWorld 的应用，它位于一个名为 e24caf14-8483-4743-b80c-ca46c28c75df_1.0.0.0_x86__97ghe447vaan8 的包中。 通过显示所有已安装软件包的全名和运行状态来验证软件包是否已安装。 在命令提示符窗口中输入以下命令。 可以使用 CTRL+F 在命令输出中搜索应用名称 HelloWorld。

```
dbgcmd

.querypackages
...

Package Full Name: e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
Package Display Name: HelloWorld
Version: 1.0.0.0
Processor Architecture: x86
Publisher: CN=user1
Publisher Display Name: user1
Install Folder: c:\users\user1\documents\visual studio
2015\Projects\HelloWorld\HelloWorld\bin\x86\Release\AppX
Package State: Unknown
AppId: App

...
```

使用 .createpackageapp 启动并附加到应用。 .createpackageapp 命令还可以调试应用。

```
dbgcmd
```

```
.createpackageapp e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8 App
```

调试结束后，使用 .disablepackagedebug 命令递减包的调试引用计数。

```
dbgcmd

.disablepackagedebug e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

将调试器附加到正在运行的应用

假设要将 WinDbg 附加到已在运行的 MyApp。 在 WinDbg 的**文件**菜单中，选择**附加到进程**。 记下 MyApp 的进程 ID。 假设进程 ID 为 4816。 递增包含 MyApp 的包的调试引用计数。

```
dbgcmd

.enablepackagedebug e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

在 WinDbg 的**附加到进程**对话框中，选择进程 4816，然后单击“确定”。 WinDbg 将附加到 MyApp。

调试结束后，使用 .disablepackagedebug 命令递减包的调试引用计数。

```
dbgcmd

.disablepackagedebug e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

手动暂停和恢复应用

按照以下步骤手动暂停和恢复应用。 首先，递增包含应用的包调试引用计数。

```
dbgcmd

.enablepackagedebug  e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

挂起包。 应用的挂起处理程序会被调用，这对于调试非常有用。

```
dbgcmd
```

```
.suspendpackage e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

调试完成后，继续运行包。

dbgcmd

```
.resumepackage e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

最后，递减包的调试引用计数。

dbgcmd

```
.disablepackagedebug e24caf14-8483-4743-b80c-
ca46c28c75df_1.0.0.0_x86__97ghe447vaan8
```

另请参阅

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用 WinDbg 打开转储文件

项目 • 2024/02/12

可通过多种方式使用 WinDbg 打开故障内存转储文件来调试代码。

WinDbg 菜单

如果 WinDbg 已在运行并且处于休眠模式，可以通过从“**文件**”菜单中选择“**打开故障转储**”或按 Ctrl+D 打开转储文件。 出现“ **打开故障转储** ”对话框时，在“文件名”框中输入故障转储文件的完整路径和 **名称** ，或使用对话框选择正确的路径和文件名。 选择正确的文件后，选择“ **打开**”。

命令提示符

在命令提示符窗口中，可以在启动 WinDbg 时打开转储文件。 请使用以下命令：

```
Console

windbg -y <SymbolPath> -i <ImagePath> -z <DumpFileName>
```

选项 `-v` （即详细模式）也很有用。 有关命令行语法的详细信息，请参阅 WinDbg 命令行选项。

调试器命令窗口

如果 WinDbg 已在内核模式调试会话中，则可以使用 .opendump (打开转储文件) 命令打开转储文件 ，后跟 g (Go) 命令。

在 WinDbg 中设置符号和可执行映像路径

项目 • 2023/06/14

符号路径

符号路径指定符号文件所在的目录。 有关符号和符号文件的详细信息，请参阅符号。

注意 如果连接到 Internet 或公司网络，则访问符号的最有效方法是使用符号服务器。 可以通过在符号路径中使用 srv* 或 symsrv* 字符串来使用符号服务器。 有关访问符号服务器的详细信息，请参阅Windows 调试器的符号路径。

若要控制 WinDbg 中的符号路径，请执行以下操作之一：

- 从“文件”菜单中选择“**符号文件路径**”或按 Ctrl+S。
- 使用 **.sympath (设置符号路径)** 命令。 如果使用符号服务器，. **symfix (设置符号存储路径)** 命令类似于 **.sympath** ，但无需键入。
- 启动调试器时，请使用 **-y** 命令行选项。 请参阅 **WinDbg Command-Line选项**。
- 在启动调试器之前，请使用 _NT_SYMBOL_PATH 并_NT_ALT_SYMBOL_PATH 环境变量 来设置路径。 通过在_NT_ALT_SYMBOL_PATH后追加_NT_SYMBOL_PATH来创建符号路径。 (通常通过_NT_SYMBOL_PATH设置路径。但是，你可能希望在特殊情况下使用_NT_ALT_SYMBOL_PATH来替代这些设置，例如，当你具有共享符号文件的私有版本时。) 如果尝试通过这些环境变量添加无效目录，调试器将忽略此目录。

  注意 如果使用 **-sins** 命令行选项，调试器将忽略符号路径环境变量。 有关详细信息，请参阅 **WinDbg Command-Line选项**。

可执行映像路径

可执行文件是处理器可以运行的二进制文件。 这些文件通常具有.exe、.dll或.sys文件扩展名。 可执行文件也称为模块，尤其是可执行文件被描述为较大应用程序的单元时。 在 Windows 操作系统运行可执行文件之前，它会将其加载到内存中。 内存中可执行文件的副本称为可执行映像或映像。

注意 这些术语有时使用得不精确。 例如，某些文档可能会对磁盘上的实际文件使用“image”。 此外，Windows 内核和 HAL 具有特殊的模块名称。 例如， **nt** 模块对应于 Ntoskrnl.exe 文件。

可执行映像路径指定二进制可执行文件所在的目录。

在大多数情况下，调试器知道可执行文件的位置，因此无需设置此文件的路径。

但是，在某些情况下，需要此路径。 例如，内核模式 小型内存转储 文件不包含在发生停止错误时内存中存在的所有可执行文件，（即崩溃）。 同样，用户模式小型转储文件不包含应用程序二进制文件。 如果设置可执行文件的路径，调试器可以找到这些二进制文件。

调试器的可执行映像路径是由多个目录路径组成的字符串，用分号分隔。 支持相对路径。 但是，除非始终从同一目录启动调试器，否则应在每个路径之前添加驱动程序号或网络共享。 还支持网络共享。 调试器以递归方式搜索可执行映像路径。 也就是说，调试器会搜索此路径中列出的每个目录的子目录。

若要控制 WinDbg 中的可执行映像路径，请执行以下操作之一：

- 从 “**文件** ”菜单中选择“图像 **文件路径**”，或按 Ctrl+I。
- 使用 **.exepath (set Executable Path)** 命令。
- 启动调试器时，请使用 **-i** 命令行选项。 请参阅 **WinDbg Command-Line选项**。
- 在启动调试器之前，请使用 _NT_EXECUTABLE_IMAGE_PATH 环境变量 设置路径。

  注意 如果使用 **-sins** 命令行选项，调试器将忽略可执行映像路径环境变量。 有关详细信息，请参阅 **WinDbg Command-Line选项**。

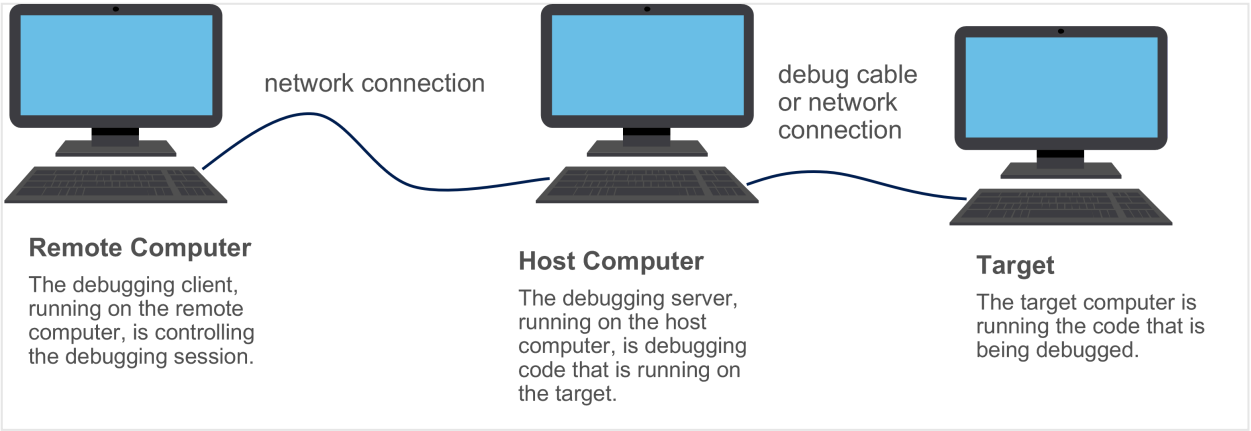
使用 WinDbg 进行远程调试

项目 • 2024/10/31

远程调试涉及两个在两个不同的位置运行的调试器。 执行调试的调试器称为 *调试服务器*。 第二个调试器称为 *调试客户端*，控制来自远程位置的调试会话。 若要建立远程会话，必须先设置调试服务器，然后激活调试客户端。

正在调试的代码可以在运行调试服务器的同一台计算机上运行，也可以运行在单独的计算机上。 如果调试服务器正在执行用户模式调试，则正在调试的进程可以在调试服务器所在的同一台计算机上运行。 如果调试服务器正在执行内核模式调试，则调试的代码通常会在单独的目标计算机上运行。

下图演示了一个远程会话，其中调试服务器在主计算机上运行，正在对在单独的目标计算机上运行的代码执行内核模式调试。



有多种传输协议可用于远程调试连接：TCP、NPIPE、SPIPE、SSL 和 COM 端口。 假设已选择使用 TCP 作为协议，并且已选择将 WinDbg 用作调试客户端和调试服务器。 可以使用以下过程建立远程内核模式调试会话：

1. 在主计算机上，打开 WinDbg，并使用目标计算机建立内核模式调试会话。 （请参阅 使用 WinDbg（经典）进行实时内核模式调试。
2. 通过**从“调试”菜单中选择“中断”**或按 Ctrl-Break 来中断。
3. 在 调试器命令窗口中，输入以下命令。

   .server tcp：port=5005

   请注意，端口号 5005 是任意的。 端口号是你选择的。
4. WinDbg 将响应如下所示的输出。

```
dbgcmd
```

```
Server started.  Client can connect with any of these command lines
0: <debugger> -remote tcp:Port=5005,Server=YourHostComputer
```

5. 在远程计算机上，打开 WinDbg，然后从“文件”**菜单中选择“连接到远程会话”。**

6. 在“连接字符串”**下**，输入以下字符串。

   tcp：Port=5005，Server=*YourHostComputer*

   其中 ， *YourHostComputer* 是运行调试服务器的主机的名称。

   选择“确定”。

使用命令行

作为上一部分中给出的过程的替代方法，可以在命令行中设置远程调试会话。 假设你设置为在主计算机与目标计算机之间通过通道 32 上的 1394 电缆建立内核模式调试会话。 可以使用以下过程建立远程调试会话：

1. 在主计算机上，在命令提示符窗口中输入以下命令。

   windbg -server tcp：port=5005 -k 1394：channel=32

2. 在远程计算机上，在命令提示符窗口中输入以下命令。

   windbg -remote tcp：Port=5005，Server=*YourHostComputer*

   其中 ， *YourHostComputer* 是运行调试服务器的主机的名称。

> ⓘ **重要**
>
> 1394 传输可用于 Windows 10 版本 1607 及更低版本。 它在 Windows 的更高版本中不可用。 应将项目转换为其他传输，例如使用以太网的 KDNET。 有关该传输的详细信息，请参阅**自动设置 KDNET 网络内核调试**。

其他信息

有多种方法可以建立远程调试，而不是本主题中所示的远程调试。 有关在 WinDbg 调试器命令窗口中设置调试服务器的完整信息，请参阅 **.server （创建调试服务器）。** 有关在命令行上启动 WinDbg（并建立远程调试）的完整信息，请参阅 **WinDbg 命令行选项**。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

在 WinDbg 中配置异常和事件

项目 • 2023/06/15

可以将 WinDbg 配置为以特定方式响应指定的异常和事件。 对于每个异常，可以设置中断状态和处理状态。 对于每个事件，可以设置中断状态。

可以通过执行以下操作之一来配置中断状态：

- 从“**调试**”菜单中选择“**事件筛选器**”，在“**事件筛选器**”对话框中的列表中单击所需的事件，然后选择“**启用**”、“**禁用**”、“**输出**”或“**忽略**”。
- 使用 SXE、 SXD、 SXN 或 SXI 命令。

可以通过执行以下操作之一来配置处理状态：

- 从“**调试**”菜单中选择“**事件筛选器**”，在“**事件筛选器**”对话框中的列表中单击所需的事件，然后选择“**已处理**”或“**未处理**”。
- 使用 SXE、 SXD、 SXN 或 SXI 命令。

有关异常和事件的详细讨论，请参阅 控制异常和事件。

在 WinDbg 中保存日志文件

项目 • 2023/10/26

WinDbg 可以写入记录调试会话的日志文件。 此日志文件包含 “调试器命令”窗口的所有内容，包括键入的命令和调试器的响应。

打开新的日志文件

若要打开新的日志文件或覆盖以前的日志文件，请执行下列操作之一：

- 从“**编辑**”菜单中选择“**打开/关闭日志文件**”。
- 在命令提示符窗口中启动 WinDbg 时，请使用 **-logo** 命令行选项。
- 输入 **.logopen (打开日志文件)** 命令。 如果使用 **/t** 选项，则日期和时间将追加到指定的文件名中。 如果使用 **/u** 选项，则日志文件以 Unicode 而不是 ASCII 写入。

追加到现有日志文件

若要将命令窗口文本追加到日志文件，请执行以下操作之一：

- 从“**编辑**”菜单中选择“**打开/关闭日志文件**”。
- 在命令提示符窗口中启动 WinDbg 时，请使用 **-loga** 命令行选项。
- 输入 **.logappend (追加日志文件)** 命令。 如果要追加到 Unicode 日志文件，则必须使用 **/u** 选项。

关闭日志文件

若要关闭打开的日志文件，请执行下列操作之一：

- 从“**编辑**”菜单中选择“**打开/关闭日志文件**”。
- 输入 **.logclose (关闭日志文件)** 命令。

检查日志文件状态

若要确定日志文件状态，请执行下列操作之一：

- 从“**编辑**”菜单中选择“**打开/关闭日志文件**”，并查看是否列出了日志文件。

- 输入 **.logfile (显示日志文件状态)** 命令。

使用 KD 和 NTKD 进行调试

项目 • 2023/06/15

本部分介绍如何使用 KD 和 NTKD 调试器执行基本调试任务。

KD 和 NTKD 在各方面都是相同的，只是 NTKD 在启动时会生成一个新的文本窗口，而 KD 继承调用它的命令提示符窗口。 本节中的说明针对 KD 提供，但它们同样适用于 NTKD。 有关何时使用 KD 或 NTKD 的讨论，请参阅 调试环境。

以下主题中提供了详细信息：

- 使用 KD 打开转储文件
- 使用 KD 进行实时内核模式调试
- 在 KD 中结束调试会话
- 在 KD 中设置符号和可执行映像路径
- 在 KD 中设置断点
- 在 KD 中查看调用堆栈
- 在 KD 中查看和编辑内存
- 在 KD 中查看和编辑寄存器
- 使用 KD 进行远程调试
- 在 KD 中配置异常和事件
- 在 KD 中保存日志文件

使用 KD 打开转储文件

项目 • 2024/07/08

命令提示符

在命令提示符窗口中，可以在启动 KD 时打开转储文件。 使用以下命令。

kd -y *SymbolPath* **-i** *ImagePath* **-z** *DumpFileName*

-v 选项（详细模式）也很有用。 有关命令行语法的详细信息，请参阅 **KD 命令行选项**。

KD 命令行

还可以在调试器运行后打开转储文件，方法是输入 .opendump （打开转储文件） **命令，后跟 g （Go）。**

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用 KD 进行实时内核模式调试

项目 • 2024/10/31

在命令提示符窗口中，启动 KD 时，可以启动实时内核模式调试会话。 输入以下命令之一。

kd [-y *SymbolPath*] -k net：port=*PortNumber*，*key*=*Key*[，target=*TargetIPAddress*|*TargetHostName*]

kd [-y *SymbolPath*] -k usb：targetname=*USBString*

kd [-y *SymbolPath*] -k com：port=*ComPort*，*baud*=*BaudRate*

kd [-y *SymbolPath*] -k com：ipport=*SerialTcpIpPort*，*port*=*SerialIPAddress*

kd [-y *SymbolPath*] -k com：pipe，port=*VMHost*\pipe*PipeName*[，resets=0][，reconnect]

kd [-y *SymbolPath*] -k com：*调制解调器*

kd [-y *SymbolPath*] -kl

kd [-y *SymbolPath*] -k

有关详细信息，请参阅 **KD 命令行选项**。

环境变量

若要通过串行（COM 端口）连接进行调试，可以使用环境变量来指定连接设置。

使用以下变量指定串行连接。

set _NT_DEBUG_PORT = *ComPort*

set _NT_DEBUG_BAUD_RATE = *BaudRate*

有关详细信息，请参阅 内核模式环境变量。

参数

SymbolPath
符号文件所在的目录列表。 列表中的目录用分号分隔。 有关详细信息，请参阅 符号路径。

PortNumber
用于网络调试的端口号。 可以选择从 49152 到 65535 的任意数字。 有关详细信息，请参阅 手动设置网络连接。

钥匙
用于网络调试的加密密钥。 建议使用自动生成的密钥，该密钥由 bcdedit 在配置目标计算机时提供。 有关详细信息，请参阅 手动设置网络连接。

TargetIPAddress
目标计算机的 IPv4 地址。

指定 target= IP 地址时，这会导致调试器通过向目标发送特殊数据包来启动与指定目标计算机的连接，从而导致它尝试与该调试器连接。 调试器会每隔半秒重复向目标发送数据包，尝试连接。 如果连接成功，目标将删除任何现有连接，并仅与此调试器实例通信。 这样，就可以从现有调试连接中控制调试会话。

当目标配置有主机 IP 地址，并且调试器在具有配置的主机 IP 地址的计算机上运行时，无需指定 target= IP 地址参数。 当目标配置有主机 IP 地址时，它将每隔三秒将 OFFER 数据包发送到主机。 如果未指定 target= IP 地址，则 OFFER 数据包允许调试器连接到主机。

有关在目标上配置主机 IP 地址的详细信息，请参阅 “自动 设置 KDNET 网络内核调试”和 “手动设置 KDNET 网络内核调试”。

TargetMachineName
目标电脑的计算机名称。 若要使用计算机名称，网络上的 DNS 系统必须具有与目标电脑的 IP 地址关联的计算机名称。

USBString
USB 连接字符串。 这必须与使用 /targetname 启动选项指定的字符串匹配。 有关详细信息，请参阅设置 USB 3.0 xHCI-DBC 内核模式调试（KDUSB）。

ComPort
COM 端口的名称。 这可以采用格式“com2”或格式为“\\.\com2”，但不应只是数字。 有关详细信息，请参阅 手动设置串行连接。

BaudRate
波特率。 可以是 9600、19200、38400、57600 或 115200。

VMHost
调试虚拟机时， *VMHost* 指定运行虚拟机的物理计算机的名称。 如果虚拟机与内核调试器本身在同一台计算机上运行，请对 *VMHost* 使用单个句点（.）。 有关详细信息，请参阅 设置与虚拟机的连接。

PipeName
虚拟机为调试连接创建的管道的名称。

resets=0
指定在主机和目标同步时，可以向目标发送不限数量的重置数据包。 仅当调试某些类型的虚拟机时，才需要此参数。

重新
使调试器在发生读/写失败时自动断开连接并重新连接管道。 此外，如果在启动调试器时找不到命名管道，则重新连接参数将导致它等待显示此名称的管道。 仅当调试某些类型的虚拟机时，才需要此参数。

-kl
使调试器执行本地内核模式调试。 有关详细信息，请参阅 本地内核模式调试。

SerialTcpIpPort
模拟串行 COM 端口的目标 TCP IP 端口。

SerialIPAddress
模拟串行 COM 端口的目标 TCP IP 地址。

示例

以下批处理文件可用于通过 COM 端口连接设置和启动调试会话。

控制台

```
set _NT_SYMBOL_PATH=d:\mysymbols
set _NT_DEBUG_PORT=com1
set _NT_DEBUG_BAUD_RATE=115200
set _NT_DEBUG_LOG_FILE_OPEN=d:\debuggers\logfile1.log
kd
```

以下命令行可用于启动 WinDbg，而无需任何环境变量。

kd -y d：\mysymbols -k com：port=com2，baud=57600

kd -y d：\mysymbols -k com：port=\.\com2，baud=115200

kd -y d：\mysymbols -k net：port=50000，key=*AutoGeneratedKey*

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

在 KD 中结束调试会话

项目 • 2023/06/15

可通过两种不同的方式退出 KD。

- 在 KD 中发出 **q (Quit)** 命令以保存日志文件、结束调试会话并退出调试器。 目标计算机保持锁定状态。
- 按 **Ctrl+B** ，然后按 Enter 以突然结束调试器。 如果希望目标计算机在调试器结束后继续运行，则必须使用此方法。 由于 CTRL+B 不会删除断点，因此应首先使用以下命令。

  dbgcmd

  ```
  kd>  bc *
  kd>  g
  ```

使用 CTRL+B 退出调试器不会清除内核模式断点，但附加新的内核调试器会清除这些断点。

执行远程调试时， **q** 结束调试会话。 CTRL+B 退出调试器，但使会话保持活动状态。 这种情况使另一个调试器能够连接到会话。

如果 **q (退出)** 命令不起作用，请按 **Ctrl+R** ，然后按主计算机键盘上的 ENTER，然后重试 **q** 命令。 如果此过程不起作用，则必须使用 Ctrl+B 退出调试程序。

在 KD 中设置符号和可执行映像路径

项目 • 2023/06/15

符号路径

符号路径指定符号文件所在的目录。 有关符号和符号文件的详细信息，请参阅符号。

注意 如果连接到 Internet 或公司网络，则访问符号的最有效方法是使用符号服务器。 可以通过在符号路径中使用 srv* 或 symsrv* 字符串来使用符号服务器。 有关访问符号服务器的详细信息，请参阅 Windows 调试器的符号路径。

若要控制 KD 中的符号路径，请执行下列操作之一：

- 输入 **.sympath (Set Symbol Path)** 命令。 如果使用符号服务器， **.symfix (设置符号存储路径)** 命令类似于 **.sympath** ，但可以保存键入内容。
- 启动调试器时，请使用 **-y** 命令行选项。 请参阅 **KD Command-Line 选项**。
- 在启动调试器之前，请使用 _NT_SYMBOL_PATH 并_NT_ALT_SYMBOL_PATH 环境变量 来设置路径。 符号路径是通过在_NT_ALT_SYMBOL_PATH后追加 _NT_SYMBOL_PATH来创建的。 (通常通过_NT_SYMBOL_PATH设置路径。但是，你可能希望在特殊情况下使用_NT_ALT_SYMBOL_PATH来替代这些设置，例如，当你拥有共享符号文件的专用版本时。)

  注意 如果使用 **-sins** 命令行选项，调试器将忽略符号路径环境变量。

可执行映像路径

可执行文件是处理器可以运行的二进制文件。 这些文件通常具有.exe、.dll或.sys文件扩展名。 可执行文件也称为模块，尤其是当可执行文件被描述为较大应用程序的单元时。 在 Windows 操作系统运行可执行文件之前，它会将其加载到内存中。 内存中可执行文件的副本称为可执行映像或映像。

注意 这些术语有时使用得不精确。 例如，某些文档可能会对磁盘上的实际文件使用“映像”。 此外，基于 Windows 的应用程序引用可执行文件名称，该名称通常包括文件扩展名。 但这些应用程序引用模块名称，该名称不包括文件扩展名。 此外，Windows 内核和 HAL 具有特殊的模块名称。 例如， **nt** 模块对应于 Ntoskrnl.exe 文件。

可执行映像路径指定二进制可执行文件所在的目录。

在大多数情况下，调试器知道可执行文件的位置，因此不必设置此文件的路径。

但是，在某些情况下需要此路径。 例如，内核模式 小型内存转储 文件不包含在发生停止错误时内存中存在的所有可执行文件，（即崩溃）。 同样，用户模式微型转储文件不包含应用程序二进制文件。 如果设置可执行文件的路径，调试器可以找到这些二进制文件。

调试器的可执行映像路径是一个由多个目录路径组成的字符串，用分号分隔。 支持相对路径。 但是，除非始终从同一目录启动调试器，否则应在每个路径之前添加驱动程序号或网络共享。 还支持网络共享。 调试器以递归方式搜索可执行映像路径。 也就是说，调试器将搜索此路径中列出的每个目录的子目录。

若要控制 KD 中的可执行映像路径，请执行下列操作之一：

- 输入 **.exepath (设置可执行路径)** 命令。
- 启动调试器时，请使用 **-i** 命令行选项。 请参阅 **KD Command-Line 选项**。
- 在启动调试器之前，请使用 _NT_EXECUTABLE_IMAGE_PATH 环境变量 来设置路径。

  注意 如果使用 **-sins** 命令行选项，调试器将忽略可执行映像路径环境变量。

在 KD 中设置断点

项目 • 2024/03/20

可以通过输入命令来设置、查看和操作 KD 中的断点。 有关命令的列表，请参阅 控制断点的方法。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

在 KD 中查看调用堆栈

项目 • 2023/06/15

调用堆栈是导致程序计数器当前位置的函数调用链。 调用堆栈上的顶部函数是当前函数，下一个函数是调用当前函数的函数，依此而行。 除非更改寄存器上下文，否则显示的调用堆栈基于当前程序计数器。 有关如何更改寄存器上下文的详细信息，请参阅 更改上下文。

在 KD 中，可以通过输入 **k (显示堆栈回溯)** 命令之一来查看调用堆栈。

在 KD 中查看和编辑内存

项目 • 2023/06/15

查看和编辑内存

在 KD 中，可以通过输入其中一个 **“显示内存”** 命令来查看和编辑内存，还可以通过输入其中一个 **“输入值”** 命令来编辑内存。 有关这些命令的详细讨论，请参阅 按虚拟地址访问内存 和 按物理地址访问内存。

查看和编辑变量

在 KD 中，可以通过输入命令来查看和编辑全局变量。 调试器将全局变量的名称解释为虚拟地址。 因此， 通过虚拟地址访问内存 中所述的所有命令都可用于读取或写入全局变量。 有关查看和编辑全局变量的其他信息，请参阅 访问全局变量。

在 KD 中，可以通过输入命令来查看和编辑局部变量。 调试器将局部变量的名称解释为地址。 因此， 可以通过虚拟地址访问内存 中所述的所有命令都可用于读取或写入局部变量。 但是，如果需要向命令指示符号是本地符号，请在符号前面加上美元符号（$），（！），如 中所示 `$!var`。 有关查看和编辑局部变量的其他信息，请参阅 访问局部变量。

在 KD 中查看和编辑寄存器

项目 • 2023/06/15

寄存器是位于 CPU 上的小型易失性内存单元。 许多寄存器专用于特定用途，其他寄存器可供用户模式应用程序使用。 基于 x86 和基于 x64 的处理器具有不同的寄存器集合。 有关每个处理器上的寄存器的详细信息，请参阅 处理器体系结构。

在 KD 中，可以通过输入 **r (Registers)** 命令来查看和编辑寄存器。 可以使用多个选项或使用 **rm (Register Mask)** 命令自定义显示。

每次目标停止时，寄存器也会自动显示。 如果使用 **p (Step)** 或 **t (Trace)** **命令单步执行** 代码，则每个步骤都会显示寄存器。 若要停止此显示，请在使用这些命令时使用 **r** 选项。

在基于 x86 的处理器上， **r** 选项还控制多个称为标志的一位寄存器。 若要更改这些标志，使用的语法与更改常规寄存器时略有不同。 有关这些标志的详细信息和此语法的说明，请参阅 x86 标志。

使用 KD 进行远程调试

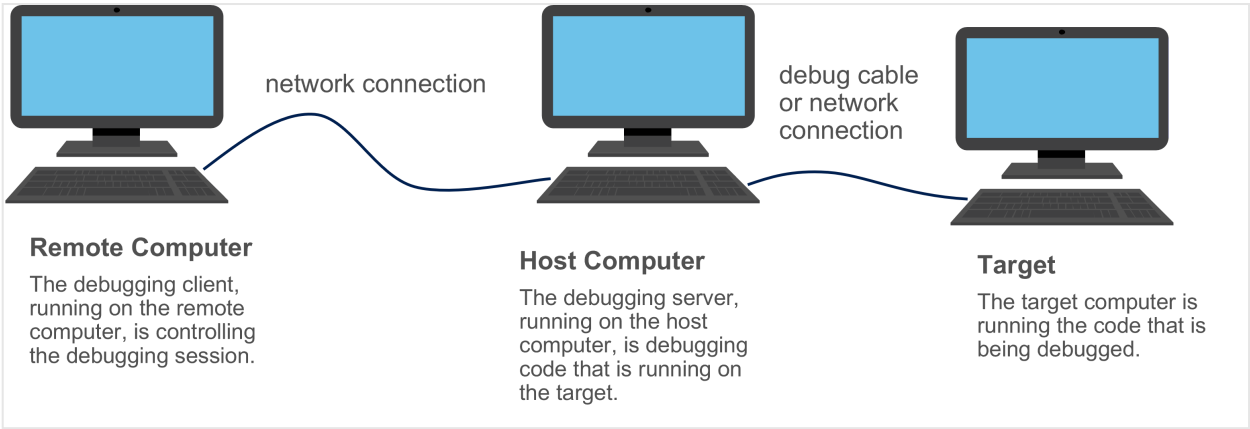
项目 • 2023/11/15

远程调试涉及在两个不同位置运行的两个调试器。 执行调试的调试器称为 *调试服务器*。 第二个调试器（称为 *调试客户端*）从远程位置控制调试会话。 若要建立远程会话，必须先设置调试服务器，然后激活调试客户端。

当你希望让其他人查看你正在电脑上调试的问题时，远程调试非常有用。

正在调试的代码可以在运行调试服务器的同一台计算机上运行，也可能在单独的计算机上运行。 如果调试服务器正在执行用户模式调试，则正在调试的进程可以在与调试服务器相同的计算机上运行。 如果调试服务器正在执行内核模式调试，则所调试的代码通常会在单独的目标计算机上运行。

下图演示了一个远程会话，在该会话中，在主计算机上运行的调试服务器正在对在单独的目标计算机上运行的代码执行内核模式调试。



有多种传输协议可用于远程调试连接：TCP、NPIPE、SPIPE、SSL 和 COM 端口。 假设你已选择使用 TCP 作为协议，并且已选择使用 KD 作为调试客户端和调试服务器。 可以使用以下过程建立远程内核模式调试会话：

1. 在主计算机上，打开 KD 并与目标计算机建立内核模式调试会话。 (请参阅使用 KD.) 执行 Kernel-Mode 调试
2. 按 CRTL-Break 进行中断。
3. 输入以下命令。

   .server tcp：port=5005

   注意 端口号 5005 是任意的。 端口号由你选择。
4. KD 将使用如下所示的输出进行响应。

```
dbgcmd
```

```
Server started.  Client can connect with any of these command lines
0: <debugger> -remote tcp:Port=5005,Server=YourHostComputer
```

5. 在远程计算机上，打开命令提示符窗口，并输入以下命令。

   kd -remote tcp：Port=5005，Server=*YourHostComputer*

   其中， *YourHostComputer* 是运行调试服务器的主机的名称。

其他信息

有关在命令行上启动 KD (和建立远程调试) 的完整信息，请参阅 **KD Command-Line 选项**。

在 KD 中配置异常和事件

项目 • 2023/06/14

可以将 KD 配置为以特定方式响应指定的异常和事件。 对于每个异常，可以设置中断状态和处理状态。 对于每个事件，可以设置中断状态。

可以使用 **SXE**、**SXD**、**SXN** 或 **SXI** 命令配置中断状态或处理状态。

有关异常和事件的详细讨论，请参阅 控制异常和事件。

在 KD 中保存日志文件

项目 • 2023/06/15

KD 可以写入记录调试会话的日志文件。 此日志文件包含键入的所有命令以及调试器的响应。

打开新的日志文件

若要打开新的日志文件或覆盖以前的日志文件，请执行下列操作之一：

- 在启动调试器之前，请设置_NT_DEBUG_LOG_FILE_OPEN 环境变量。
- 启动调试器时，请使用 **-logo** 命令行选项。
- 输入 **.logopen (打开日志文件)** 命令。 如果使用 **/t** 选项，则日期和时间将追加到指定的文件名中。 如果使用 **/u** 选项，则日志文件以 Unicode 而不是 ASCII 写入。

追加到现有日志文件

若要将文本追加到现有日志文件，请执行下列操作之一：

- 在启动调试器之前，请设置 _NT_DEBUG_LOG_FILE_APPEND 环境变量。
- 启动调试器时，请使用 **-loga** 命令行选项。
- 输入 **.logappend (追加日志文件)** 命令。 如果要追加到 Unicode 日志文件，则必须使用 **/u** 选项。

关闭日志文件

若要关闭打开的日志文件，请执行以下操作：

- 输入 **.logclose (关闭日志文件)** 命令。

检查日志文件状态

若要确定日志文件状态，请执行以下操作：

- 输入 **.logfile (显示日志文件状态)** 命令。

使用 CDB 和 NTSD 进行调试

项目 • 2023/06/15

本部分介绍如何使用 Microsoft 控制台调试器 (CDB) 和 Microsoft NT 符号调试器 (NTSD) 执行基本调试任务。

CDB 和 NTSD 在各方面都是相同的，只是 NTSD 在启动时会生成一个新的文本窗口，而 CDB 继承调用它的命令提示符窗口。 本节中的说明适用于 CDB，但它们同样适用于 NTSD。 有关何时使用 CDB 或 NTSD 的讨论，请参阅 调试环境。

以下主题中提供了详细信息：

- 使用 CDB 调试用户模式进程
- 使用 CDB 打开转储文件
- 在 CDB 中结束调试会话
- 在 CDB 中设置符号和可执行映像路径
- 在 CDB 中设置断点
- 在 CDB 中查看调用堆栈
- 在 CDB 中查看和编辑内存
- 在 CDB 中查看和编辑寄存器
- 在 CDB 中配置异常和事件
- 在 CDB 中保存日志文件

使用 CDB 调试用户模式进程

项目 • 2024/07/08

可以使用 CDB 附加到正在运行的进程或生成并附加到新进程。

附加到正在运行的进程

命令提示符

在命令提示符窗口中，可以在启动 CDB 时附加到正在运行的进程。 使用以下命令之一：

- **cdb -p** *ProcessID*
- **cdb -pn** *ProcessName*

其中 *ProcessID* 是正在运行的进程 ID，或者 *ProcessName* 是正在运行的进程的名称。

有关命令行语法的详细信息，请参阅 **CDB 命令行选项**。

CDB 命令窗口

如果调试器已经调试了一个或多个进程，则可以使用 **.attach（附加到进程）** 命令附加到正在运行的进程。

除非某些线程被冻结或挂起，否则调试器始终同时启动多个目标进程。

如果 .attach 命令成功，调试器将在下次调试器发出执行命令时附加到指定的进程。 如果在一行中多次使用此命令，则必须像使用此命令一样多次请求调试器执行。

非侵入性附加到正在运行的进程

如果要调试正在运行的进程，并且只干扰其执行，则应以非侵入方式调试进程。

命令提示符

若要从 CDB 命令行以非侵入方式调试正在运行的进程，请在以下语法中指定 **-pv** 选项、**-p** 选项和进程 ID。

cdb -pv -p *ProcessID*

或者，若要通过指定进程名称以非侵入性方式调试正在运行的进程，请改用以下语法。

cdb -pv -pn *ProcessName*

还有其他几个有用的命令行选项。 有关命令行语法的详细信息，请参阅 **CDB 命令行选项**。

CDB 命令窗口

如果调试器已处于活动状态，则可以通过输入 **.attach -v（附加到进程）** 命令，以非侵入性地调试正在运行的进程。

如果调试器已经以侵入性的方式调试了一个或多个进程，则可以使用 **.attach** 命令。

如果 **.attach -v** 命令成功，调试器将在下次调试器发出执行命令时调试指定的进程。 由于在非侵入性调试期间不允许执行，因此调试器不能一次非侵入性地调试多个进程。 此限制还意味着使用 **.attach -v** 命令可能会使现有入侵调试会话不那么有用。

生成新进程

CDB 可以启动用户模式应用程序，然后调试应用程序。 应用程序按名称指定。 调试器还可以自动附加到子进程（原始目标进程启动的其他进程）。

调试器创建的进程（也称为生成进程）的行为与调试器不创建的进程的行为略有不同。

调试器创建的进程不使用标准堆 API，而是使用特殊的调试堆。 可以使用 _NO_DEBUG_HEAP 环境变量 或 **-hd** 命令行选项强制生成进程使用标准堆而不是调试堆。

此外，由于目标应用程序是调试器的子进程，因此它将继承调试器的权限。 此权限可能使目标应用程序能够执行无法执行的某些操作。 例如，目标应用程序可能能够影响受保护的进程。

在命令提示符窗口中，可以在启动 CDB 时生成新进程。 输入以下命令。

cdb [**-o**] ProgramName *[Arguments]*

-o 选项会导致调试器附加到子进程。 还有其他几个有用的命令行选项。 有关命令行语法的详细信息，请参阅 **CDB 命令行选项**。

如果调试器已在调试一个或多个进程，可以通过输入 **.create （创建进程）** 命令来创建新进程。

除非某些线程被冻结或挂起，否则调试器将始终同时启动多个目标进程。

如果 **.create** 命令成功，调试器将在下次调试器发出执行命令时创建指定的进程。 如果在一行中多次使用此命令，则必须像使用此命令一样多次请求调试器执行。

可以在 .create 之前使用 **.createdir （Set Created Process Directory）** 命令来控制应用程序的起始目录。 可以使用 **.createdir -I** 命令或 **-noinh** 命令行选项来控制目标应用程序是否继承调试器的句柄。

可以使用 .childdbg（调试子进程）**命令激活或停用子进程的**调试。

重新附加到进程

如果调试器停止响应或冻结，可以将新调试器附加到目标进程。 有关如何在此情况下附加调试器的详细信息，请参阅 重新附加到目标应用程序。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用 CDB 打开转储文件

项目 • 2024/07/08

命令提示符

在命令提示符窗口中，启动 CDB 时，可以打开用户模式转储文件。 输入以下命令。

cdb **-y** *SymbolPath* **-i** *ImagePath* **-z** *DumpFileName*

-v 选项（详细模式）也很有用。 有关命令行语法的详细信息，请参阅 **CDB 命令行选项**

CDB 命令行

还可以在调试器运行后打开转储文件，方法是输入 .opendump （打开转储文件） **命令，后跟 g （Go）。** 这样就可以同时调试多个转储文件。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

在 CDB 中结束调试会话

项目 • 2023/10/26

可以通过输入 q (Quit) **命令退出** CDB。 此命令还会关闭要调试的应用程序。

qd (Quit and Detach) 命令将 CDB 从目标应用程序分离，退出调试器，并使目标应用程序保持运行状态。 如果在启动调试器时使用了 **-pd** 命令行选项，则如果会话因任何原因结束，则会发生分离。 (此方法使 **-pd** 在调试不想结束的敏感进程（例如客户端服务器 Run-Time 子系统 (CSRSS) ）时特别有用。）

如果调试器未响应，可以按 CTRL+B ，然后按 ENTER 退出。 此方法是辅助退出机制。 它突然结束调试器，类似于通过任务管理器或关闭窗口结束进程。

若要结束用户模式调试会话，将调试器返回到休眠模式，并关闭目标应用程序，可以使用以下方法：

- 输入 .kill (Kill Process) 命令。

若要结束用户模式调试会话，将调试器返回到休眠模式，并设置目标应用程序再次运行，可以使用以下方法：

- 从 Process) **命令输入** .detach (Detach 命令。 如果要调试多个目标，此命令将从当前目标分离，并继续与剩余目标的调试会话。
- 输入 qd (Quit and Detach) 命令。
- 如果已使用 **-pd** 选项启动调试器，请输入 q (Quit) 命令。

若要结束用户模式调试会话，请将调试器返回到休眠模式，但让目标应用程序保持调试状态，可以使用以下方法：

- 输入 .abandon (**放弃进程**) 命令。

有关重新附加到目标的详细信息，请参阅 重新附加到目标应用程序。

在 CDB 中设置符号和可执行映像路径

项目 • 2023/10/26

符号路径

符号路径指定符号文件所在的目录。 有关符号和符号文件的详细信息，请参阅符号。

注意 如果连接到 Internet 或公司网络，则访问符号的最有效方法是使用符号服务器。 可以通过在符号路径中使用 srv* 或 symsrv* 字符串来使用符号服务器。 有关符号服务器的详细信息，请参阅 Windows 调试器的符号路径。

若要控制 CDB 中的符号路径，请执行以下操作之一：

- 输入 **.sympath (设置符号路径)** 命令。 如果使用符号服务器， **.symfix (设置符号存储路径)** 命令类似于 .sympath，但无需键入。
- 启动调试器时，请使用 **-y** 命令行选项。 请参阅 **CDB Command-Line 选项**。
- 在启动调试器之前，请使用 _NT_SYMBOL_PATH 并_NT_ALT_SYMBOL_PATH 环境变量 来设置路径。 通过在_NT_ALT_SYMBOL_PATH后追加_NT_SYMBOL_PATH来创建符号路径。 (通常通过_NT_SYMBOL_PATH设置路径。但是，你可能希望在特殊情况下使用_NT_ALT_SYMBOL_PATH来替代这些设置，例如，当你拥有共享符号文件的专用版本时。)

  注意 如果使用 **-sins** 命令行选项，调试器将忽略符号路径环境变量。

可执行映像路径

可执行文件是处理器可以运行的二进制文件。 这些文件通常具有 .exe、.dll 或 .sys 文件扩展名。 可执行文件也称为模块，尤其是可执行文件被描述为较大应用程序的单元时。 在 Windows 操作系统运行可执行文件之前，它会将其加载到内存中。 内存中可执行文件的副本称为可执行映像或映像。

注意 这些术语有时使用得不精确。 例如，某些文档可能会对磁盘上的实际文件使用“image”。 此外，Windows 内核和 HAL 具有特殊的模块名称。 例如， **nt** 模块对应于 Ntoskrnl.exe 文件。

可执行映像路径指定二进制可执行文件所在的目录。

在大多数情况下，调试器知道可执行文件的位置，因此无需设置此文件的路径。

但是，在某些情况下，需要此路径。 例如，内核模式 小型内存转储 文件不包含在发生停止错误时内存中存在的所有可执行文件，（即崩溃）。 同样，用户模式小型转储文件不包含应用程序二进制文件。 如果设置可执行文件的路径，调试器可以找到这些二进制文件。

调试器的可执行映像路径是由多个目录路径组成的字符串，用分号分隔。 支持相对路径。 但是，除非始终从同一目录启动调试器，否则应在每个路径之前添加驱动程序号或网络共享。 还支持网络共享。 调试器以递归方式搜索可执行映像路径。 也就是说，调试器会搜索此路径中列出的每个目录的子目录。

若要控制 CDB 中的可执行映像路径，请执行以下操作之一：

- 输入 **.exepath (Set Executable Path)** 命令。
- 启动调试器时，请使用 **-i** 命令行选项。 请参阅 **CDB Command-Line 选项**。
- 在启动调试器之前，请使用 _NT_EXECUTABLE_IMAGE_PATH 环境变量 设置路径。

  注意 如果使用 **-sins** 命令行选项，调试器将忽略可执行映像路径环境变量。

在 CDB 中设置断点

项目 • 2024/03/20

可以通过输入命令在 CDB 中设置、查看和操作断点。 有关命令的列表，请参阅 控制断点的方法。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

在 CDB 中查看调用堆栈

项目 • 2023/10/26

调用堆栈是导致程序计数器当前位置的函数调用链。 调用堆栈上的顶部函数是当前函数，下一个函数是调用当前函数的函数，依此类拔。 除非更改寄存器上下文，否则显示的调用堆栈基于当前程序计数器。 有关如何更改寄存器上下文的详细信息，请参阅 更改上下文。

在 CDB 中，可以通过输入 **k (显示堆栈回溯)** 命令之一来查看调用堆栈。

在 CDB 中查看和编辑内存

项目 • 2023/06/15

查看和编辑内存

在 CDB 中，可以通过输入其中一个 **“显示内存** ”命令来查看和编辑内存，也可以输入其中一个 **Enter Values** 命令来编辑内存。 有关这些命令的详细讨论，请参阅 按虚拟地址访问内存 和 按物理地址访问内存。

查看和编辑变量

在 CDB 中，可以通过输入命令来查看和编辑全局变量。 调试器将全局变量的名称解释为虚拟地址。 因此，可以使用 按虚拟地址访问内存 中所述的所有命令读取或写入全局变量。 有关查看和编辑全局变量的其他信息，请参阅 访问全局变量。

在 CDB 中，可以通过输入命令来查看和编辑局部变量。 调试器将局部变量的名称解释为地址。 因此， 可以使用按虚拟地址访问内存 中所述的所有命令来读取或写入局部变量。 但是，如果需要向命令指示符号是本地符号，请在符号前面加上美元符号（$），感叹号（!），如 中所示 `$!var`。 有关查看和编辑局部变量的其他信息，请参阅 访问局部变量。

在 CDB 中查看和编辑寄存器

项目 • 2023/10/26

寄存器是位于 CPU 上的小型易失性内存单元。 许多寄存器专用于特定用途，其他寄存器可供用户模式应用程序使用。 基于 x86 和基于 x64 的处理器具有可用的寄存器的不同集合。 有关每个处理器上的寄存器的详细信息，请参阅 处理器体系结构。

在 CDB 中，可以通过在“调试器命令”窗口中输入 **r (Registers)** 命令来查看寄存器。 可以使用多个选项或使用 **rm (Register Mask)** 命令自定义显示。

每次目标停止时，也会自动显示寄存器。 如果使用 **p (Step)** 或 **t (Trace)** 命令单步执行代码，则每个步骤都会显示寄存器。 若要停止此显示，请在使用这些命令时使用 **r** 选项。

在基于 x86 的处理器上， **r** 选项还控制多个称为标志的一位寄存器。 若要更改这些标志，请使用与更改常规寄存器时略有不同的语法。 有关这些标志的详细信息和此语法的说明，请参阅 x86 Flags。

在 CDB 中配置异常和事件

项目 • 2023/06/15

可以将 CDB 配置为以特定方式响应指定的异常和事件。 对于每个异常，可以设置中断状态和处理状态。 对于每个事件，可以设置中断状态。

可以通过执行以下操作之一来配置中断状态或处理状态：

- 使用 **SXE**、 **SXD**、 **SXN** 或 **SXI** 命令。
- 在 **CDB 命令行**上使用 **-x**、**-xe**、**-xd**、**-xn** 或 **-xi** 选项。
- 在 Tools.ini 文件中使用 **sxe** 或 **sxd** 关键字 (keyword) 。

有关异常和事件的详细讨论，请参阅 控制异常和事件。

在 CDB 中保存日志文件

项目 • 2023/10/26

CDB 可以编写记录调试会话的日志文件。 此日志文件包含键入的所有命令以及来自调试器的响应。

打开新日志文件

若要打开新的日志文件或覆盖以前的日志文件，请执行下列操作之一：

- 在启动调试器之前，请设置_NT_DEBUG_LOG_FILE_OPEN 环境变量。
- 启动调试器时，请使用 **-logo** 命令行选项。
- 输入 **.logopen (打开日志文件)** 命令。 如果使用 **/t** 选项，日期和时间将追加到指定的文件名中。 如果使用 **/u** 选项，则日志文件以 Unicode 而不是 ASCII 编写。

追加到现有日志文件

若要将命令窗口文本追加到日志文件，请执行下列操作之一：

- 在启动调试器之前，请设置_NT_DEBUG_LOG_FILE_APPEND 环境变量。
- 启动调试器时，请使用 **-loga** 命令行选项。
- 输入 **.logappend (追加日志文件)** 命令。 如果要追加到 Unicode 日志文件，则必须使用 **/u** 选项。

关闭日志文件

若要关闭打开的日志文件，请执行以下操作：

- 输入 **.logclose (关闭日志文件)** 命令。

检查日志文件状态

若要确定日志文件状态，请执行以下操作：

- 输入 **.logfile (显示日志文件状态)** 命令。

本地内核模式调试

项目 • 2023/06/15

适用于 Windows 的调试工具支持 *本地内核调试*。 这是单台计算机上的内核模式调试。 换句话说，调试器在正在调试的同一台计算机上运行。

设置本地Kernel-Mode调试

有关设置本地内核模式调试的信息，请参阅 手动设置本地Kernel-Mode调试单台计算机。

启动调试会话

使用 WinDbg

以管理员身份打开 WinDbg。 在“ **文件** ”菜单上，选择“ **内核调试**”。 在“内核调试”对话框中，打开“ **本地** ”选项卡。选择“ **确定**”。

还可以通过以管理员身份打开命令提示符窗口并输入以下命令来启动 WinDbg 会话：

windbg -kl

使用 KD

以管理员身份打开命令提示符窗口，并输入以下命令：

kd -kl

不可用的命令

并非所有命令都在本地内核调试会话中可用。 通常，不能使用导致目标计算机停止的任何命令，甚至暂时停止，因为无法恢复操作。

具体而言，不能使用以下命令：

- 执行命令，例如 **g (Go)** 、 **p (Step)** 、 **t (跟踪)** 、 **wt (跟踪和监视数据)** 、 **tb (跟踪到下一个分支)** 、 **gh () Go** 和 **gn (Go 未处理异常)**
- 关闭和转储文件命令，例如 **.crash**、 **.dump** 和 **.reboot**
- 断点命令，例如 **bp**、 **bu**、 **ba**、 **bc**、 **bd**、 **be** 和 **bl**

- 注册显示命令，例如 **r** 和变体
- 堆栈跟踪命令，例如 **k** 和变体

如果使用 WinDbg 执行本地内核调试，则所有等效的菜单命令和按钮也不可用。

可用的命令

所有内存输入和输出命令都可用。 可以从用户内存和内核内存中自由读取。 还可以写入内存。 请确保不要写入内核内存的错误部分，因为它可能会损坏数据结构，并经常导致计算机停止响应 (即 *崩溃*) 。

执行本地内核调试的困难

本地内核调试是一项非常精细的操作。 请注意，系统不会损坏或崩溃。

本地内核调试最困难的方面之一是计算机状态在不断变化。 内存被分页和传出，活动进程不断变化，虚拟地址上下文不保持不变。 但是，在这些情况下，可以有效地分析变化缓慢的事物，例如某些设备状态。

内核模式驱动程序和 Windows 操作系统经常使用 **DbgPrint** 和相关函数将消息发送到内核调试器。 在本地内核调试期间，这些消息不会自动显示。 可以使用 **！ dbgprint** 扩展来显示它们。

LiveKD

LiveKD 工具模拟本地内核调试。 此工具会创建内核内存的“快照”转储文件，而不会在进行此快照时实际停止内核。 (因此，快照实际上可能不会显示计算机的单一即时状态。)

LiveKD 不是 Windows 调试工具包的一部分。 可以从 Windows Sysinternals 站点下载 LiveKd 。

控制目标

项目 • 2024/07/08

在用户模式下调试目标应用程序或内核模式下的目标计算机时，目标可以 *运行* 或 *停止*。

当调试器连接到内核模式目标时，调试器将离开目标运行，除非使用 **-b** 命令行选项，否则目标系统已停止响应（即*崩溃*），或者目标系统由于早期内核调试操作而停止。

当调试器启动或连接到用户模式目标时，调试器会立即停止目标，除非使用 **-g** 命令行选项。 有关详细信息，请参阅 初始断点。

当目标正在运行时

目标运行时，大多数调试器操作都不可用。

如果要停止正在运行的目标，可以发出 **Break** 命令。 此命令会导致调试器 *闯入目标*。 也就是说，调试器停止目标，并向调试器提供所有控件。 应用程序可能不会立即中断。 例如，如果所有线程当前都在执行系统代码，或者处于等待操作中，则应用程序仅在控制返回到应用程序代码后中断。

如果正在运行的目标遇到异常、发生某些事件、命中断点或应用程序正常关闭，则目标会*中断调试器*。 此操作会停止目标，并向调试器提供所有控制。 “调试器命令”窗口中会显示一条消息，并描述错误、事件或断点。

目标停止时

若要启动或控制目标的执行，可以执行以下操作：

- 若要使应用程序开始运行，请发出 **Go** 命令。
- 若要一次单步执行应用程序一个指令，请使用 “**单步执行** ”或 “**单步执行** ”命令。 如果函数调用发生， **单步执行** 将进入函数并继续单步执行每个指令。 **Step Over** 将函数调用视为单个步骤。 当调试器处于 程序集模式时，单步执行一次发生一台计算机指令。 当调试器处于 源模式时，单步执行一次发生一个源行。
- 若要完成当前函数并在返回发生时停止，请使用 **Step Out** 或 **Trace 和 Watch** 命令。 **Step Out** 命令将一直持续到当前函数结束。 **跟踪和监视** 将一直持续到当前函数结束，并显示函数调用的摘要。 但是，必须在相关函数的第一个指令上发出 **Trace 和 Watch** 命令。
- 如果发生异常，可以使用 **Go 和异常处理和 Go 和“未处理**异常”命令恢复执行并控制异常的状态。 （有关异常的详细信息，请参阅 控制异常和事件。

- （仅限 WinDbg）如果在反汇编窗口或源窗口中选择一行，然后使用“**运行到光标**”命令，程序将运行，直到遇到所选行。
- （仅限用户模式）若要关闭目标应用程序并从头开始重启它，请使用 “**重启**” 命令。 只能对调试器创建的进程使用此命令。 重启进程后，它会立即中断调试器。
- （仅限 WinDbg）若要关闭目标应用程序并清除调试器，请使用 “**停止调试**” 命令。 此命令使你能够开始调试其他目标。

命令窗体

用于启动或控制目标执行的大多数命令都以文本命令、菜单命令、工具栏按钮和快捷键的形式存在。 作为基本文本命令，可以在 CDB、KD 或 WinDbg 中使用这些命令。 （命令的文本形式经常支持其他选项，例如更改程序计数器的位置或执行固定数量的指令。可以在 WinDbg 中使用菜单命令、工具栏按钮和快捷键。

可以使用以下窗体中的命令。

展开表

命令	WinDbg 命令	WinDbg 快捷键	效果
	调试 | 运行到光标处	F7 CTRL + F10	（仅限 WinDbg）执行，直到它到达光标标记的行。
	调试 | 停止调试	SHIFT + F5	停止所有调试并关闭目标。
（仅限 CDB/KD）Ctrl+C	调试 | 中断	CTRL + BREAK	执行停止，调试器会进入目标。
.restart （重启目标应用程序）	调试 | 重启	CTRL + SHIFT + F5	（仅限用户模式）重启目标应用程序。
g（转到）	调试 | 转到	F5	目标可以自由执行。
gc （从条件断点进行）			在条件断点后恢复执行。
gh （带异常处理的 Go）	调试 | 转到已处理的异常		与 g（Go）相同，但当前异常被视为已处理。

命令	WinDbg 命令	WinDbg 快捷键	效果
gn （未处理异常的 Go）	调试 | 转到未经处理的异常		**与 g （Go）** 相同，但当前异常被视为未经处理的异常。
顾 （上去）	调试 | 跳出	SHIFT + F11	目标在当前函数完成之前执行。
p（步进）	调试 | 逐步运行	F10	目标执行一个指令。 如果此指令是函数调用，该函数将作为单个步骤执行。
pa （地址步骤）			目标执行到到达指定地址为止。 将显示此函数中的所有步骤（但调用函数中的步骤不是）。
pc （下一个呼叫的步骤）			目标执行到下一**个调用** 指令为止。 如果当前指令是 **调用** 指令，则完全执行此调用，直到下一次**调用**继续执行。
pct （下一个调用或返回的步骤）			目标执行到到达 **调用** 指令或 **返回** 指令为止。
ph （下一步分支指令）			目标执行到达到任何类型的分支指令，包括条件分支或无条件分支、调用、返回和系统调用。
pt （下一步返回）			目标执行到到达 **返回** 指令为止。
t （跟踪）	调试 | 跳入	F11 F8	目标执行一个指令。 如果此指令是函数调用，调试器将跟踪该调用。

命令	WinDbg 命令	WinDbg 快捷键	效果
ta （跟踪地址）			目标执行到到达指定地址为止。 将显示此函数和调用函数中的所有步骤。
tb （跟踪到下一分支）			（所有模式（内核模式除外）仅在基于 x86 的系统上）目标执行到下一分支指令为止。
tc （跟踪到下一个调用）			目标执行到下一**个调用** 指令为止。 如果当前指令是 **呼叫** 指令，则会跟踪该指令，直到到达新 **呼叫** 为止。
tct （跟踪到下一次调用或返回）			目标执行到到达 **调用** 指令或 **返回** 指令为止。 如果当前指令是 **呼叫** 指令或 **返回** 指令，则会跟踪该指令，直到到达新的 **呼叫** 或 **返回** 。
th （跟踪到下一分支指令）			目标执行到达到任何类型的分支指令，包括条件分支或无条件分支、调用、返回和系统调用。 如果当前指令是分支指令，则会跟踪该指令，直到到达新的分支指令为止。
tt （跟踪到下一个返回）			目标执行到到达 **返回** 指令为止。 如果当前指令是 **返回** 指令，则会跟踪该指令，直

命令	WinDbg 命令	WinDbg 快捷键	效果
			到到达新的 **返回** 为止。
wt （跟踪和监视数据）			目标执行到整个指定函数完成为止。 然后显示统计信息。

有关如何重启目标计算机的详细信息，请参阅 崩溃和重新启动目标计算机。

命令行选项

如果不希望应用程序在启动或加载时立即停止，请使用 CDB 或 WinDbg 和 **-g** 命令行选项。 有关这种情况的详细信息，请参阅 初始断点。

CDB 和 WinDbg 还支持 **-G** 命令行选项。 如果应用程序正确完成，此选项会导致调试会话结束。

以下命令尝试从头到尾运行应用程序，并且仅当发生错误时，调试器提示才会显示。

控制台

```
cdb -g -G ApplicationName
```

可以使用 **-pt** 命令行选项 设置中断超时。某些问题可能导致目标无法与调试器通信。 如果发出中断命令并且调试器在此时间后无法进入目标，则调试器将显示“中断超时”消息。

此时，调试器停止尝试闯入目标。 相反，调试器会暂停目标，并使你能够检查目标应用程序（但不控制）。

默认超时为 30 秒。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

启用事后调试

项目 • 2024/07/08

用户模式异常处理

异常和断点

最常见的应用程序错误称为异常。 其中包括访问冲突、除以零错误、数值溢出、CLR 异常和许多其他类型的错误。 应用程序还可能导致断点中断。 当 Windows 无法运行应用程序（例如，无法加载必要的模块时）或遇到断点时，会出现这些情况。 断点可由调试器插入代码中，也可以通过 DebugBreak **等**函数调用。

异常处理程序优先级

Windows 根据配置值和哪些调试器处于活动状态，以多种方式处理用户模式错误。 以下序列显示了用于用户模式错误处理的优先级：

1. 如果用户模式调试器当前附加到故障进程，则所有错误都将导致目标进入此调试器。

   只要附加了用户模式调试器，就不会使用其他错误处理方法 -- 即使 **使用 gn （Go With Exception Not Handled）** 命令也是如此。

2. 如果没有附加用户模式调试器，并且执行代码有自己的异常处理例程（例如 **，try -except**），则此异常处理例程将尝试处理错误。

3. 如果未附加用户模式调试器，并且 Windows 具有打开的内核调试连接，并且错误是断点中断，Windows 将尝试联系内核调试器。

   必须在 Windows 启动过程中打开内核调试连接。 如果想要阻止用户模式中断进入内核调试器，可以将 KDbgCtrl 实用工具与 -du **参数一起使用。** 有关如何配置内核调试连接以及如何使用 KDbgCtrl 的详细信息，请参阅 “获取调试设置”。

   在内核调试器中，可以使用 **gh （Go With Exception Handled）** 忽略错误并继续运行目标。 可以使用 **gn（未处理异常）** 绕过内核调试器并转到步骤 4。

4. 如果步骤 1、2 和 3 中的条件不适用，Windows 将激活 AeDebug 注册表值中配置的调试工具。 可以提前选择任何程序作为在此情况下使用的工具。 所选程序称为 *事后调试器*。

5. 如果步骤 1、2 和 3 中的条件不适用，并且未注册事后调试器，Windows 错误报告（WER）会显示一条消息，并提供解决方案（如果有）。 如果注册表中设置了适当

的值，WER 还会写入内存转储文件。 有关详细信息，请参阅 使用 WER 和 收集用户模式转储。

DebugBreak 函数

如果已安装事后调试器，可以通过调用 **DebugBreak** 函数来故意从用户模式应用程序中断调试器。

指定事后图调试器

本部分介绍如何将 WinDbg 等工具配置为事后调试器。 配置后，每当应用程序崩溃时，都会自动启动事后调试器。

事后调试器注册表项

Windows 错误报告（WER）使用 AeDebug 注册表项中设置的值创建事后调试器进程。

HKLM\Software\Microsoft\Windows NT\CurrentVersion\AeDebug

有两个主要注册表值感兴趣，*即调试器和自动*。 *调试器*注册表值指定事后调试器的命令行。 自动注册表值指定是否自动启动事后调试器，或者是否首先显示确认消息框。

调试器（REG_SZ）

此REG_SZ值指定将处理事后调试的调试器。

除非调试器位于默认路径中的目录中，否则必须列出调试器的完整路径。

命令行通过包含 3 个参数的 printf 样式调用从调试器字符串生成。 虽然顺序是固定的，但不需要使用任何或所有可用参数。

DWORD （%ld） - 目标进程的进程 ID。

DWORD （%ld） - 事件句柄重复到事后调试器进程中。 如果事后调试器发出事件信号，WER 将继续执行目标进程，而无需等待事后调试器终止。 仅当问题得到解决时，才应发出该事件信号。 如果事后调试器在未发出事件信号的情况下终止，WER 将继续收集有关目标进程的信息。

void* （%p） - 在目标进程的地址空间中分配的JIT_DEBUG_INFO结构的地址。 该结构包含其他异常信息和上下文。

自动（REG_SZ） 此REG_SZ值始终 **为 0 或 1。**

如果 **自动** 设置为 **0**，则会在启动事后调试过程之前显示确认消息框。

如果 **自动** 设置为 **1**，则立即创建事后调试器。

手动编辑注册表时，请仔细执行此操作，因为注册表的更改不当可能不允许 Windows 启动。

命令行用法示例

许多验尸调试器使用包含 -p 和 -e 开关的命令行来指示参数是 PID 和事件（分别）。 例如，通过 `windbg.exe -I` 创建以下值安装 WinDbg：

控制台

```
Debugger = "<Path>\WinDbg -p %ld -e %ld -g"
Auto = 1
```

使用 WER %ld %p 参数的方式有灵活性。 例如： 不需要指定 WER 参数周围或之间的任何开关。 例如，使用 `procdump.exe -i` 安装 Windows Sysinternals ProcDump 会创建以下值，且 WER %ld %ld %p 参数之间没有切换：

控制台

```
Debugger = "<Path>\procdump.exe" -accepteula -j "c:\Dumps" %ld %ld %p
Auto = 1
```

32 和 64 位调试器

在 64 位平台上，调试器（REG_SZ）和自动（REG_SZ）注册表值分别为 64 位和 32 位应用程序定义。 Windows 上的其他 Windows（WOW）密钥用于存储 32 位应用程序验尸后调试值。

HKLM\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\AeDebug

在 64 位平台上，对 32 位进程使用 32 位事后调试器，对 64 位进程使用 64 位调试器。 这可以避免在 32 位进程中将 64 位调试器集中在 WOW64 线程而不是 32 位线程上。

对于许多事后调试器（包括 Windows 验尸调试器的调试工具）而言，这涉及到运行安装命令两次;一次使用 x86 版本，一次使用 x64 版本。 例如，若要将 WinDbg 用作交互式事后调试器，该 `windbg.exe -I` 命令将针对每个版本运行两次。

64 位安装：

控制台

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\windbg.exe –I
```

这会使用这些值更新注册表项。

reg

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug
Debugger = "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\windbg.exe"
-p %ld -e %ld –g
```

32 位安装：

控制台

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\windbg.exe –I
```

这会使用这些值更新注册表项。

reg

```
HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Windows
NT\CurrentVersion\AeDebug
Debugger = "C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\windbg.exe"
-p %ld -e %ld –g
```

配置事后调试器

Windows 调试工具

Windows 调试器的调试工具支持设置为事后调试器。 install 命令希望以交互方式调试进程。

WinDbg

若要将事后调试器设置为 WinDbg，请运行 `windbg -I`。 `I`（必须大写。此命令在使用后将显示成功或失败消息。 若要同时处理 32 位和 64 位应用程序，请为 64 和 32 个调试器运行命令。

控制台

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\windbg.exe –I
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\windbg.exe –I
```

这是运行时 `windbg -I` 如何配置 AeDebug 注册表项。

控制台

```
Debugger = "<Path>\WinDbg -p %ld -e %ld -g"
Auto = 1
```

在示例中， *<Path>* 是调试器所在的目录。

如前所述，-p 和 -e 参数传递进程 ID 和事件。

-g 将 g （Go） 命令传递给 WinDbg，并继续从当前指令执行。

请注意，传递 g （Go） 命令存在重大问题。 此方法的问题在于，异常通常不会总是重复，因为重启代码时不再存在的暂时性条件。 有关此问题的详细信息，请参阅 **.jdinfo （使用 JIT_DEBUG_INFO）。**

若要避免此问题，请使用 .jdinfo 或 .dump /j。 此方法允许调试器位于相关代码故障的上下文中。 有关详细信息，请参阅 本主题后面的实时（JIT）调试 。

CDB

若要将事后调试器设置为 CDB，请运行 **cdb -iae** （安装 AeDebug） 或 **cdb -iaec** *KeyString* （使用命令安装 AeDebug）。

控制台

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\cdb.exe -iae
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\cdb.exe -iae
```

使用 -iaec 参数时，*KeyString* 指定要追加到命令行末尾的字符串，用于启动事后调试器。 如果 *KeyString* 包含空格，则必须用引号将其引起来。

控制台

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\cdb.exe -iaec
[KeyString]
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\cdb.exe -iaec
[KeyString]
```

此命令在成功时不显示任何内容，如果失败，则显示一条错误消息。

NTSD

若要将事后调试器设置为 NTSD，请运行 **ntsd -iae** （Install AeDebug） 或 **ntsd -iaec** *KeyString* （Install AeDebug with Command）。

控制台

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\ntsd.exe -iae
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\ntsd.exe -iae
```

使用 -iaec 参数时，*KeyString* 指定要追加到命令行末尾的字符串，用于启动事后调试器。 如果 *KeyString* 包含空格，则必须用引号将其引起来。

控制台

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\ntsd.exe -iaec
[KeyString]
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\ntsd.exe -iaec
[KeyString]
```

如果成功，此命令将不显示任何内容，并在失败时显示新控制台窗口的错误。

请注意，由于 -p %ld -e %ld -g 参数始终首先出现在事后调试器的命令行上，因此不应使用 -iaec 开关指定 -server 参数，因为 -server 将不起作用，除非它在命令行上出现第一个。 若要安装包含此参数的事后调试器，必须手动编辑注册表。

Visual Studio JIT 调试器

如果已安装 Visual Studio，vsjitdebugger.exe将注册为事后调试器。 Visual Studio JIT 调试器打算以交互方式调试进程。

控制台

```
Debugger = "C:\WINDOWS\system32\vsjitdebugger.exe" -p %ld -e %ld
```

如果 Visual Studio 已更新或重新安装，则会重新编写此条目，覆盖设置的任何备用值。

Window Sysinternals ProcDump

Windows Sysinternals ProcDump 实用工具也可用于事后转储捕获。 有关使用和下载 ProcDump 的详细信息，请参阅 ProcDump。

与 .dump WinDbg 命令一样，ProcDump 能够以非交互方式捕获崩溃的转储。 捕获可能发生在任何 Windows 系统会话中。

ProcDump 在转储文件捕获完成时退出，然后 WER 报告失败并终止故障进程。

用于 `procdump -i` 安装 procdump 和 -you 为 32 位和 64 位验尸后调试卸载 ProcDump。

控制台

```
<Path>\procdump.exe -i
```

安装和卸载命令输出成功时修改的注册表值，以及失败时的错误。

注册表中的 ProcDump 命令行选项设置为：

控制台

```
Debugger = <Path>\ProcDump.exe -accepteula -j "<DumpFolder>" %ld %ld %p
```

ProcDump 使用所有 3 个参数 - PID、事件和JIT_DEBUG_INFO。 有关JIT_DEBUG_INFO参数的详细信息，请参阅 下面的实时（JIT）调试 。

捕获的转储大小默认为 Mini（process/threads/handles/modules/address space），没有大小选项集、MiniPlus （Mini plus MEM_PRIVATE pages）和 -mp set 或 Full（所有内存 - 等效于 -马 set 的".dump /mA"）。

对于具有足够驱动器空间的系统，建议使用完整（-马）捕获。

将 -马 与 -i 选项一起使用以指定所有内存捕获。 （可选）提供转储文件的路径。

控制台

```
<Path>\procdump.exe -ma -i c:\Dumps
```

对于驱动器空间有限的系统，建议使用 MiniPlus （-mp） 捕获。

控制台

```
<Path>\procdump.exe -mp -i c:\Dumps
```

要将转储文件保存到的文件夹是可选的。 默认值为当前文件夹。 该文件夹应使用等于或优于用于 C：\Windows\Temp 的 ACL 进行保护。有关管理与文件夹相关的安全的详细信息，请参阅 事后调试期间的安全性。

若要卸载 ProcDump 作为事后调试器，并还原以前的设置，请使用 -u （卸载） 选项。

控制台

```
<Path>\procdump.exe -u
```

有关 ProcDump 的其他信息，请参阅 Microsoft Press 发布的 Mark Russinovich 和 Aaron Margosis 的 ProcDump 和 *Windows SysInternals 管理员参考* 。

恰时 （JIT） 调试

将上下文设置为故障应用程序

如前所述，使用 JIT_DEBUG_INFO 参数将上下文设置为导致崩溃的异常是非常可取的。 有关此内容的详细信息，请参阅 **.jdinfo （使用 JIT_DEBUG_INFO）。**

Windows 调试工具

此示例演示如何编辑注册表以运行使用 .jdinfo <地址> 命令显示其他异常信息的初始命令（-c），并将上下文更改为异常的位置（类似于 .ecxr 如何使用 .ecxr 将上下文设置为异常记录）。

控制台

```
Debugger = "<Path>\windbg.exe -p %ld -e %ld -c ".jdinfo 0x%p"
Auto = 1
```

%p 参数是目标进程的地址空间中JIT_DEBUG_INFO结构的地址。 %p 参数预追加了 0x，以便将其解释为十六进制值。 有关详细信息，请参阅 **.jdinfo （使用 JIT_DEBUG_INFO）。**

若要调试 32 和 64 位应用的混合，请配置 32 位和 64 位注册表项（如上所述），将正确的路径设置为 64 位和 32 位WinDbg.exe的位置。

使用 .dump 创建转储文件

若要在发生包含JIT_DEBUG_INFO数据的故障时捕获转储文件，请使用 .dump /j <地址>。

控制台

```
<Path>\windbg.exe -p %ld -e %ld -c ".dump /j %p /u <DumpPath>\AeDebug.dmp;
qd"
```

使用 /u 选项生成唯一文件名，以允许自动创建多个转储文件。 有关选项的详细信息，**请参阅 .dump （创建转储文件）。**

创建的转储将JITDEBUG_INFO数据存储为默认异常上下文。 不使用 .jdinfo 查看异常信息并设置上下文，而是使用 .exr -1 显示异常记录和 .ecxr 设置上下文。 有关详细信息，请参阅 .exr （显示异常记录） **和 .ecxr （显示异常上下文记录）。**

Windows 错误报告 - q/qd

调试会话结束的方式确定Windows 错误报告是否报告了失败。

如果在调试器关闭之前使用 qd 分离调试会话，WER 将报告失败。

如果调试会话使用 q 退出（或者调试器关闭而不分离），WER 将不会报告失败。

追加 *;q* 或 *;qd* 到命令字符串的末尾以调用所需的行为。

例如，若要允许 WER 在 CDB 捕获转储后报告失败，请配置此命令字符串。

控制台

```
<Path>\cdb.exe -p %ld -e %ld -c ".dump /j 0x%p /u c:\Dumps\AeDebug.dmp; qd"
```

此示例允许 WER 在 WinDbg 捕获转储后报告失败。

控制台

```
<Path>\windbg.exe -p %ld -e %ld -c ".dump /j %p /u <DumpPath>\AeDebug.dmp;
qd""
```

安全漏洞

如果正在考虑在与其他人共享的计算机上启用事后调试，请参阅 事后调试期间的安全性。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

打开 Enclave 调试

项目 • 2023/04/14

WinDbg 版本 1.0.1908.30002 及更高版本支持 Open Enclave 调试。 开放式 Enclave 应用程序利用基于硬件的受信任执行环境，也称为 enclave。 Enclave 是受保护的内存区域，为数据和代码执行提供机密性。 它是受信任的执行环境 (TEE) 的实例，通常受硬件保护，例如 Intel Software Guard Extensions (SGX) 。

Open Enclave SDK 是一个与硬件无关的开放源代码库，用于开发利用 enclave 的应用程序。 有关详细信息，请参阅 中的 https://github.com/openenclave/openenclave 项目 GitHub。

有关如何使用 WinDbg 调试 Open Enclave 应用程序的信息，请参阅 使用 WinDbg 调试 ELF Enclave 。

debugrt DLL

Open Enclave 的某些调试功能由作为 enclave 调试目标的一部分加载的 debugrt DLL 提供。 有关详细信息和源代码，请参阅 debugrt 项目 GitHub https://aka.ms/debugrt 。

使用调试器命令程序

项目 • 2024/03/20

本节包括下列主题：

调试器命令程序的元素

控制流标记

调试器命令程序执行

调试器命令程序示例

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

调试器命令程序的元素

项目 • 2023/10/26

*调试器命令程序*是一个由调试器命令和控制流令牌（如 .if、.for 和 .while）组成的小型应用程序。 (有关控制流令牌及其语法的完整列表，请参阅 控制流令牌。)

可以使用大括号 ({ }) 将语句块括在更大的命令块中。 输入每个块时，将计算块中的所有别名。 如果稍后更改命令块中别名的值，则该点后面的命令不会使用新的别名值，除非它们位于从属块中。

不能使用一对大括号创建块。 必须在左大括号前添加控制流标记。 如果只想创建一个块来评估别名，则应在左大括号前使用 **.block** 标记。

调试器命令程序可以使用 用户命名别名或固定名称别名 作为其本地变量。 如果要使用数值或类型化变量，可以使用 **$t***n*个伪寄存器。

仅当用户命名别名不位于其他文本旁边时，才会对其进行计算。 如果要评估其他文本旁边的别名，请使用 **${ } (别名解释器)** 令牌。 此令牌具有可选开关，可用于通过多种方式评估别名。

可以使用两个美元符号将注释添加到调试器命令程序，（**$$ (注释说明符)** ）。 不应在标记与其元素之间插入注释 (，例如大括号或条件) 。

注意 不应使用星号 (*** (注释行说明符)** ）。 由于用星号指定的注释不以分号结尾，因此将忽略程序的其余部分。

通常，应在调试器命令程序中使用 MASM 语法。 当必须使用 C++ 元素（（例如指定结构或类) 的成员）时，可以使用 **@@c++ ( )** 标记切换到该子句的 C++ 语法。

MASM 语法中的 **$scmp**、 **$sicmp**和 **$spat** 字符串运算符特别有用。 有关这些运算符的详细信息，请参阅 MASM 数字和运算符。

控制流标记

项目 • 2024/05/03

可以使用 *控制流令牌* 在调试器命令程序中创建条件执行和执行循环。

控制流令牌的行为类似于 C 和 C++ 中的对应项，但有以下一般例外情况：

- 你必须将每个命令块括起来，这些命令按条件执行或重复执行，即使只有一个这样的命令。 例如，不能在以下命令中省略大括号。

  dbgcmd

  ```
  0:000> .if (ebx>0) { r ebx }
  ```

- 每个条件都必须是一个表达式。 不允许使用命令。 例如，以下示例生成语法错误。

  dbgcmd

  ```
  0:000> .while (r ebx) { .... }
  ```

- 右大括号前的最后一个命令不必后跟分号。

调试器命令程序中支持以下控制流令牌。 有关每个标记的语法的详细信息，请参阅各个参考主题。

- .if 标记的行为类似于 **C 中的 if** 关键字 (keyword)。
- .else 标记的行为类似于 C 中的 **else** 关键字。
- .elsif 标记的行为类似于 C 中的 **else if** 关键字组合。
- .foreach 令牌分析调试器命令、字符串或文本文件的输出。 然后，此令牌接受它查找的每个项，并将其用作指定调试器命令列表的输入。
- .for 标记的行为类似于 **C 中的 for** 关键字 (keyword)，但必须用分号分隔多个递增命令，而不是用逗号分隔。
- .while 令牌的行为类似于 C 中的 **while** 关键字。
- .do 标记的行为类似于 **C 中的 do** 关键字 (keyword)，只是条件前不能使用“while”一词。
- .break 令牌的行为类似于 **C 中的中断**关键字 (keyword)。可以在任何 **.for**、**.while** 或 **.do** 循环中使用此令牌。

- .continue 令牌的行为类似于 C **中的** continue 关键字 (keyword)。可以在任何 .for、.while 或 .do 循环中使用此令牌。
- 如果 **发生错误，** .catch 令牌将阻止程序结束。 .catch 标记后跟包含一个或多个命令的大括号。 如果其中一个命令生成错误，则会显示错误消息，将忽略大括号中的所有剩余命令，并在右大括号后继续执行第一个命令。
- .leave 令牌用于退出 .catch 块。
- .printf 令牌的行为类似于 C 中的 printf 语句。
- . block 令牌不执行任何操作。 应仅使用此令牌来引入块，因为不能仅使用一对大括号创建块。 必须在左大括号之前添加控制流令牌。

！for_each_module、！for_each_frame **和** ！for_each_local **扩展也可用于调试器命令程序。**

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

执行调试器命令程序

项目 • 2023/06/15

可以通过以下方式之一执行调试器命令程序：

- 在 “调试器命令”窗口中 以单个字符串的形式输入所有语句，并用分号分隔各个语句和命令。
- 将脚本文件中的所有语句添加到一行中，并用分号分隔各个语句和命令。 然后，使用使用脚本文件中所述的方法之一运行此 脚本文件。
- 将所有 语句添加到脚本文件中，每个语句位于单独的行上。 （或者，按回车符和分号的任意组合分隔语句。）然后，使用 **$>< (运行脚本文件)** 或 **$$>< (运行脚本文件) 命令运行此脚本文件** 。 这些命令打开指定的脚本文件，将所有回车符替换为分号，并将生成的文本作为单个命令块执行。

调试器命令程序示例

项目 • 2023/10/26

以下部分介绍调试器命令程序。

使用 .foreach 令牌

以下示例使用 .foreach 令牌搜索 5a4d 的 WORD 值。 对于找到的每个 5a4d 值，调试器将显示 8 个 DWORD 值，从找到 5a4d DWORD 的地址开始。

dbgcmd

```
0:000> .foreach (place { s-[1]w 77000000 L?4000000 5a4d }) { dc place L8 }
```

以下示例使用 .foreach 令牌搜索 5a4d 的 WORD 值。 对于找到的每个 5a4d 值，调试器将显示 8 个 DWORD 值，在找到 5a4d DWORD 的地址之前从 4 个字节开始。

dbgcmd

```
0:000> .foreach (place { s-[1]w 77000000 L?4000000 5a4d }) { dc place -0x4
L8 }
```

以下示例显示相同的值。

dbgcmd

```
0:000> .foreach (place { s-[1]w 77000000 L?4000000 5a4d }) { dc ( place -0x4
) L8 }
```

注意 如果要对命令的 *OutCommands* 部分中的变量名称进行操作，必须在变量名称后面添加一个空格。 例如，在之前的示例中，变量 *位置* 和减法运算符之间有一个空格。

-[1] 选项与 **(搜索内存)** 命令一起导致其输出仅包含找到的地址，而不包括在这些地址找到的值。

以下命令显示内存范围（从0x77000000到0x7F000000）的所有模块的详细模块信息。

dbgcmd

```
0:000> .foreach (place { lm1m }) { .if ((${place} >= 0x77000000) & (${place}
<= 0x7f000000)) { lmva place } }
```

1m 选项与 lm (列出加载的模块) 命令会导致其输出仅包含模块的地址，而不是模块的完整说明。

前面的示例使用 ${ } (别名解释器) 令牌来确保替换别名，即使它们位于其他文本旁边。 如果命令不包含此标记，则 **放置** 旁边的左括号将阻止别名替换。 请注意， ${} 令牌适用于 .foreach 和 true 别名中使用的变量。

遍览进程列表

以下示例演练内核模式进程列表，并显示列表中每个条目的可执行名称。

此示例应存储为文本文件，并使用 $$>< (运行脚本文件) 命令执行。 此命令加载整个文件，将所有回车符替换为分号，并执行生成的块。 使用此命令，可以使用多行和缩进编写可读程序，而不必将整个程序挤压到单行上。

此示例演示了以下功能：

- $t 0、$t 1 和 $t 2 伪寄存器用作此程序中的变量。 程序还使用名为 Procc 和 $ImageName 的别名。
- 此程序使用 MASM 表达式计算器。 但是， @@c++ ( ) 令牌出现一次。 此标记导致程序使用 C++ 表达式计算器分析括号中的表达式。 此用法使程序能够直接使用 C++ 结构令牌。
- **?** 标志与 r (Registers) 命令一起使用。 此标志将类型化值分配给伪寄存器 $t 2。

dbgcmd

```
$$  Get process list LIST_ENTRY in $t0.
r $t0 = nt!PsActiveProcessHead

$$  Iterate over all processes in list.
.for (r $t1 = poi(@$t0);
      (@$t1 != 0) & (@$t1 != @$t0);
      r $t1 = poi(@$t1))
{
    r? $t2 = #CONTAINING_RECORD(@$t1, nt!_EPROCESS, ActiveProcessLinks);
    as /x Procc @$t2

 $$  Get image name into $ImageName.
 as /ma $ImageName @@c++(&@$t2->ImageFileName[0])

 .block
    {
        .echo ${$ImageName} at ${Procc}
    }

    ad $ImageName
```

```
    ad Procc
}
```

浏览LDR_DATA_TABLE_ENTRY列表

以下示例演练用户模式LDR_DATA_TABLE_ENTRY列表，并显示每个列表条目的基址和完整路径。

与前面的示例一样，此程序应保存在 文件中，并使用 $$>< (**运行脚本文件**) 命令执行。

此示例演示了以下功能：

- 此程序使用 MASM 表达式计算器。 但是，在两个位置显示 @@c++ ( ) 令牌。 此标记导致程序使用 C++ 表达式计算器分析括号中的表达式。 此用法使程序能够直接使用 C++ 结构令牌。
- **?** 标志与 r (Registers) 命令一起使用。 此标志将类型化值分配给伪寄存器 **$t 0** 和 **$t 1**。 在循环的正文中， **$t 1** 的类型为 **ntdll!** _LDR_DATA_TABLE_ENTRY*，因此程序可以进行直接成员引用。
- 此程序使用用户命名别名 **$Base** 和 **$Mod** 。 美元符号降低了以前在当前调试器会话中使用这些别名的可能性。 美元符号没有必要。 ${/v: } 令牌按字面解释别名，防止在运行脚本之前定义别名时替换别名。 还可以将此令牌与任何块一起使用，以防止在块之前使用别名定义。
- .block 令牌用于添加额外的别名替换步骤。 加载整个脚本时发生一次别名替换，在输入每个块时发生一次别名替换。 如果没有 .block 标记及其大括号， .echo 命令不会接收上一行中分配的 **$Mod** 和 **$Base** 别名的值。

dbgcmd

```
$$ Get module list LIST_ENTRY in $t0.
r? $t0 = &@$peb->Ldr->InLoadOrderModuleList

$$ Iterate over all modules in list.
.for (r? $t1 = *(ntdll!_LDR_DATA_TABLE_ENTRY**)@$t0;
 (@$t1 != 0) & (@$t1 != @$t0);
      r? $t1 = (ntdll!_LDR_DATA_TABLE_ENTRY*)@$t1->InLoadOrderLinks.Flink)
{
    $$ Get base address in $Base.
 as /x ${/v:$Base} @@c++(@$t1->DllBase)

 $$ Get full name into $Mod.
 as /msu ${/v:$Mod} @@c++(&@$t1->FullDllName)

 .block
    {
```

```
        .echo ${$Mod} at ${$Base}
    }

    ad ${/v:$Base}
    ad ${/v:$Mod}
}
```

远程调试

项目 • 2023/03/08

远程用户模式调试涉及两台计算机： *客户端* 和 *服务器*。 服务器是运行要调试的应用程序的计算机。 服务器还运行用户模式调试器或进程服务器。 客户端是远程控制调试会话的计算机。

远程内核模式调试涉及三台计算机： *客户端*、 *服务器*和 *目标计算机*。 目标计算机是要调试的计算机。 服务器是运行内核调试器或 KD 连接服务器的计算机。 客户端是远程控制调试会话的计算机。

选择最佳的远程调试方法

通过调试器进行远程调试

通过内核调试程序控制用户模式调试程序

通过 Remote.exe 进行远程调试

进程服务器（用户模式）

KD 连接服务器（内核模式）

中继器

高级远程调试方案

在工作组计算机上进行远程调试

选择最佳的远程调试方法

项目 • 2024/03/08

执行远程调试的主要方法有两种，还有几种其他方法和大量组合方法。

下面有一些技巧可以帮助你选择最佳方法。

- 通过调试器进行远程调试通常是最佳方法。 如果你只有一台服务器和一个客户端，并且它们可以自由地相互连接，那么客户端和服务器上都安装了相同的调试器二进制文件，并且将操作客户端的调试技术人员将能够与房间里的服务器人员交谈，这是推荐的方法。

  客户端和服务器可以运行任何版本的 Windows。 它们不必运行彼此相同的版本。

  如果客户端无法向服务器发送连接请求，但服务器能够向客户端发送请求，则可以使用 **clicon** 参数通过具有*反向连接*的调试器使用远程调试。

- 通过 remote.exe 远程调试用于远程控制命令提示符窗口。 可用于远程控制 KD、CDB 或 NTSD。 不能与 WinDbg 一起使用。

  如果客户端没有调试器二进制文件的副本，则必须使用 remote.exe 方法。

- 如果调试技术人员无法与使用服务器的房间中的人员交谈，则可以使用**进程服务器**或 **KD 连接服务器**。 所有实际调试工作都是由客户端（称为*智能客户端*）完成的；这就消除了让第二个人出现在服务器本身的需要。

  进程服务器用于用户模式调试；KD 连接服务器用于内核模式调试。 除了这种区别之外，它们的行为方式类似。

  如果运行服务器的计算机无法处理繁重的进程负载，或者运行客户端的技术人员有权访问机密且无法由服务器访问的符号文件或源文件，则此方法也很有用。 但是，此方法不如通过调试器进行远程调试那样快速或高效。 此方法不能用于转储文件调试。

  有关详细信息，请参阅进程服务器（用户模式）和 KD 连接服务器（内核模式）。

- **重复器**是一个轻型代理服务器，用于在两台计算机之间中继数据。 如果通过调试器执行远程调试，或者使用进程服务器，则可以在客户端和服务器之间添加重复器。

  如果客户端和服务器无法直接相互通信，但每次访问外部计算机，但可以各自访问外部计算机，则可能需要使用重复器。 还可以将反向连接与重复器一起使用。 甚至可以连续使用两个重复器，但这很少必须这样做。

  有关详细信息，请参阅重复器。

- 还可以从内核调试器控制 CDB（或 NTSD）。 这是远程调试的另一种形式。 有关详细信息，请参阅从内核调试器控制用户模式调试器。

所有这些方法的变体都是可能的。

可以将多台计算机连接在一起，以利用多种传输方法。 可以创建复杂的传输序列，以考虑技术人员所在的位置、符号所在的位置，以及是否有防火墙阻止特定方向的连接。 示例请参阅高级远程调试方案。

甚至可以在一台计算机上执行远程调试。 例如，启动一个低权限的进程服务器，然后用一个高权限的智能客户端连接到它，这可能会很有用。

通过调试器进行远程调试

项目 • 2023/06/15

直接通过调试器进行远程调试通常是执行远程调试的最佳且最简单的方法。

此方法涉及在不同的位置运行两个调试器。 实际执行调试的调试器称为 *调试服务器*。 从一段距离控制会话的调试器称为 *调试客户端*。

这两台计算机不必运行同一版本的 Windows;它们可以运行任何版本的 Windows。 实际使用的调试器不一定相同;WinDbg 调试客户端可以连接到 CDB 调试服务器，等等。

但是，两台计算机上的调试器二进制文件最好都来自同一版本的 Windows 调试工具包，或者至少两者都来自最新版本。

若要设置此远程会话，请先设置调试服务器，然后激活调试客户端。 任意数量的调试客户端可以连接到调试服务器。 单个调试器可以同时将自身转换为多个调试服务器，以促进不同类型的连接。

但是，单个调试器不能同时是调试客户端和调试服务器。

本节包括：

激活调试服务器

搜索调试服务器

激活调试客户端

客户端和服务器示例

控制远程调试会话

激活调试服务器

项目 • 2024/02/22

有两种方法可用于激活调试服务器。 当在提升的命令提示符窗口中（以管理员身份运行），使用 **-server** 命令行选项启动调试器时，可以将其激活。 也可以在调试器在运行后将其激活。 使用提升的特权启动调试器（以管理员身份运行），然后输入 **.server** 命令。

注意 可以在没有提升的特权的情况下激活调试服务器，并且调试客户端将能够连接到服务器。 但是，除非客户端使用了提升的特权激活，否则客户端将无法发现调试服务器。 有关如何发现调试服务器的信息，请参阅搜索调试服务器。

调试器支持多种传输协议：命名管道 (NPIPE)、TCP、COM 端口、安全管道 (SPIPE) 和安全套接字层 (SSL)。

用于激活调试服务器的常规语法取决于使用的协议。

控制台

```
Debugger -server npipe:pipe=PipeName[,hidden][,password=Password]
[,IcfEnable] [-noio] [Options]

Debugger -server tcp:port=Socket[,hidden][,password=Password][,ipversion=6]
[,IcfEnable] [-noio] [Options]

Debugger -server tcp:port=Socket,clicon=Client[,password=Password]
[,ipversion=6] [-noio] [Options]

Debugger -server com:port=COMPort,baud=BaudRate,channel=COMChannel[,hidden]
[,password=Password] [-noio] [Options]

Debugger -server spipe:proto=Protocol,
{certuser=Cert|machuser=Cert},pipe=PipeName[,hidden][,password=Password] [-
noio] [Options]

Debugger -server ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},port=Socket[,hidden][,password=Password] [-
noio] [Options]

Debugger -server ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},port=Socket,clicon=Client[,password=Password]
[-noio] [Options]
```

另一种激活调试服务器的方法是在已启动调试器后，使用 **.server（创建调试服务器）** 命令。

dbgcmd

```
.server npipe:pipe=PipeName[,hidden][,password=Password][,IcfEnable]

.server tcp:port=Socket[,hidden][,password=Password][,ipversion=6]
[,IcfEnable]

.server tcp:port=Socket,clicon=Client[,password=Password][,ipversion=6]

.server com:port=COMPort,baud=BaudRate,channel=COMChannel[,hidden]
[,password=Password]

.server spipe:proto=Protocol,
{certuser=Cert|machuser=Cert},pipe=PipeName[,hidden][,password=Password]

.server ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},port=Socket[,hidden][,password=Password]

.server ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},port=Socket,clicon=Client[,password=Password]
```

上述命令中的参数具有以下可能的值：

Debugger
可以是 KD、CDB、NTSD 或 WinDbg。

pipe=*PipeName*
使用 NPIPE 或 SPIPE 协议时，*PipeName* 是将用作管道名称的字符串。 每个管道名称都应标识唯一的调试服务器。 如果尝试重用管道名称，将收到一条错误消息。 *PipeName* 不得包含空格或引号。 *PipeName* 可以包含数字 **printf** 样式的格式代码，如 **%x** 或 **%d**。 调试器会将此代码替换为调试器的进程 ID。 第二个此类代码将被替换为调试器的线程 ID。

注意 在运行调试服务器的计算机上，可能需要启用文件和打印机共享。 在控制面板中，导航到“网络和 Internet”>“**网络和共享中心**”>“高级共享设置”。 选择“启用文件和打印机共享”。

port=*Socket*
使用 TCP 或 SSL 协议时，*Socket* 是套接字端口号。

还可以指定用冒号分隔的端口范围。 调试器将检查此范围中的每个端口，以了解是否可用。 如果发现可用端口且未出错，则将创建调试服务器。 调试客户端必须指定要用于连接到服务器的实际端口。 若要确定实际端口，请使用**搜索调试连接服务器**中介绍的任一方法；显示此调试服务器时，该端口后面将紧跟两个用冒号分隔的数字。 第一个数字将是使用的实际端口；可以忽略第二个数字。 例如，如果将端口指定为 **port=51:60**，并且实际上使用了端口 53，则搜索结果将显示“port=53:60”。 （如果使用 **clicon** 参数建立反

向连接，调试客户端可以采用这种方式指定端口范围，而服务器必须指定使用的实际端口。）

clicon=*Client*
使用 TCP 或 SSL 协议并指定 **clicon** 参数时，将打开*反向连接*。 这意味着，调试服务器将尝试连接到调试客户端，而不是让客户端启动联系。 如果防火墙阻止了正常方向的连接，这将非常有用。 *Client* 指定调试客户端所在或将创建调试客户端的计算机的网络名称或 IP 地址。 两个初始反斜杠 (\) 是可选的。

由于服务器正在查找特定客户端，因此如果使用此方法，则无法将多个客户端连接到服务器。 如果连接被拒绝或已断开，则必须重启服务器连接。 当另一个调试器显示所有活动服务器时，反向连接服务器不会显示。

注意 在使用 **clicon** 时，最好在创建调试服务器之前启动调试客户端，但也允许遵循正常顺序（服务器在客户端之前）。

port=*COMPort*
使用 COM 协议时，*COMPort* 指定要使用的 COM 端口。 前缀“COM”是可选的 -- 例如，“com2”和“2”都可接受。

baud=*BaudRate*
使用 COM 协议时，*BaudRate* 指定连接运行时的波特率。 允许使用硬件支持的任何波特率。

channel=*COMChannel*
如果使用 COM 协议，*COMChannel* 指定要用于与调试客户端通信的 COM 通道。 这可以是 0 和 254 之间（含）的任何值。 可以使用不同的通道号将单个 COM 端口用于多个连接。 （这不同于将 COM 端口用于调试电缆，在那种情况下，不能在 COM 端口中使用通道。）

proto=*Protocol*
如果使用 SSL 或 SPIPE 协议，*Protocol*将指定安全通道（S 通道）协议。 这可以是字符串 tls1、pct1、ssl2 或 ssl3 中的任一个。

Cert
如果使用 SSL 或 SPIPE 协议，*Cert* 指定证书。 这可以是证书名称或证书的指纹（证书管理单元提供的十六进制数字字符串）。 如果使用语法 **certuser**=*Cert*，调试器将在系统存储区（默认存储区）中查找证书。 如果使用语法 **machuser**=*Cert*，调试器将在计算机存储区中查找证书。 指定的证书必须支持服务器身份验证。

hidden
当另一个调试器显示所有活动服务器时，阻止服务器显示。

password=*Password*
要求客户端提供指定的密码，以便连接到调试会话。 *Password* 可以是任意字母数字字符串，长度最多为 12 个字符。

警告 将密码与 TCP、NPIPE 或 COM 协议配合使用时，仅提供少量的保护，因为密码未加密。 将密码与 SSL 或 SPIPE 协议配合使用时，将对其进行加密。 如果要建立安全的远程会话，则必须使用 SSL 或 SPIPE 协议。

ipversion=6
（仅适用于 Windows 调试工具 6.6.07 及更低版本）使用 TCP 连接到 Internet 时，强制调试器使用 IP 版本 6，而不是版本 4。 在 Windows Vista 和更高版本中，调试器会尝试自动默认为 IP 版本 6，因此不需要此选项。

-noio
如果使用 **-noio** 选项创建调试服务器，则无法通过服务器本身执行任何输入或输出。 调试器将仅接受调试客户端的输入（以及 **-c** 命令行选项指定的任何初始命令或命令脚本）。 所有输出都将定向到调试客户端。 **-noio** 选项仅适用于 KD、CDB 和 NTSD。 如果将 NTSD 用于服务器，则根本不会创建控制台窗口。

IcfEnable
促使调试器在 Internet 连接防火墙处于活动状态时启用 TCP 或命名管道通信所需的端口连接。 默认情况下，Internet 连接防火墙会禁用这些协议使用的端口。 将 **IcfEnable** 与 TCP 连接配合使用时，调试器会导致 Windows 打开 *Socket* 参数指定的端口。 将 **IcfEnable** 与命名管道连接配合使用时，调试器会促使 Windows 打开用于命名管道的端口（端口 139 和 445）。 连接终止后，调试器不会关闭这些端口。

选项
可以在此处放置任何其他命令行参数。 有关完整列表，请参阅命令行选项。

可以使用 **.server** 命令，使用不同的协议选项启动多台服务器。 这允许不同类型的调试客户端加入会话。

另请参阅

控制远程调试会话

.endsrv（结束调试服务器）

搜索调试服务器

项目 • 2023/04/14

可以将 KD 或 CDB 与 **-QR** 命令行选项一起使用，以获取在网络服务器上运行的可用调试服务器的列表。

此列表可能包括不再存在但未正确关闭的服务器 -- 连接到其中一个服务器会生成错误消息。 该列表还将包括进程服务器和 KD 连接服务器。 在每种情况下都会指示服务器类型。

此配置的语法如下：

```
控制台

Debugger -QR \\Server
```

调试器 可以是 KD 或 CDB -- 无论哪种情况，显示器都相同。 (*server 前面的*\\ 两个反斜杠) 是可选的。

在 WinDbg 中，可以使用“ **连接到远程调试器会话** ”对话框浏览可用服务器的列表。

要使调试服务器可被发现，必须使用提升的权限将其激活。 有关详细信息，请参阅 激活调试服务器。

如果未使用有权访问服务器计算机的帐户登录到客户端计算机，则可能需要提供用户名和密码。 在客户端计算机上的命令提示符窗口中，输入以下命令。

net use ***Server*\ipc$ /user：***UserName* ，其中 *Server* 是服务器计算机的名称，*UserName* 是有权访问服务器计算机的帐户的名称。

出现提示时，输入 *UserName* 的密码。

此命令成功后，可以使用 **-QR** 或 **连接到远程调试器会话**发现 (在服务器计算机上运行的调试服务器) 。

激活调试客户端

项目 • 2024/07/08

激活调试服务器后，可以在另一台计算机上启动调试客户端，并连接到调试会话。

有两种方法可用于启动调试客户端：使用 -remote 命令行选项，或使用 WinDbg 图形界面。

客户端的协议必须与服务器的协议匹配。 用于启动调试客户端的常规语法取决于使用的协议。 存在以下选项：

dbgcmd

```
Debugger -remote npipe:server=Server,pipe=PipeName[,password=Password]

Debugger -remote tcp:server=Server,port=Socket[,password=Password]
[,ipversion=6]

Debugger -remote tcp:clicon=Server,port=Socket[,password=Password]
[,ipversion=6]

Debugger -remote
com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=Password]

Debugger -remote spipe:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,pipe=PipeName[,password=Password
]

Debugger -remote ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,port=Socket[,password=Password]

Debugger -remote ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},clicon=Server,port=Socket[,password=Password]
```

若要使用图形界面连接到远程调试会话，WinDbg 必须处于休眠模式，即必须已在未使用任何命令行参数的情况下启动，或者必须已结束上一个调试会话。 选择“文件 | 连接到远程会话”菜单命令，或按 CTRL+R 快捷键。 出现“连接到远程调试器会话”对话框时，在**连接字符串**文本框中输入以下字符串之一：

dbgcmd

```
npipe:server=Server,pipe=PipeName[,password=Password]

tcp:server=Server,port=Socket[,password=Password][,ipversion=6]

tcp:clicon=Server,port=Socket[,password=Password][,ipversion=6]

com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=Password]
```

```
spipe:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,pipe=PipeName[,password=Password
]

ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,port=Socket[,password=Password]

ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},clicon=Server,port=Socket[,password=Password]
```

或者，可以使用"浏览"按钮找到活动调试服务器。

上述命令中的参数具有以下可能的值：

调试器
这无需与调试客户端使用的调试器相同，即 WinDbg、KD 和 CDB 都可用于通过调试器进行远程调试。

服务器
这是在其中创建调试服务器的计算机的网络名称或 IP 地址。 两个初始反斜杠 (\) 在命令行上是可选的，但在 WinDbg 对话框中不允许使用。

pipe= *PipeName*
如果使用 NPIPE 或 SPIPE 协议，*PipeName* 是创建服务器时赋予管道的名称。

如果未使用有权访问服务器计算机的帐户登录到客户端计算机，则必须提供用户名和密码。 在客户端计算机的命令提示符窗口中输入以下命令。

net use ***Server*\ipc$ /user:***UserName*

其中 *Server* 是服务器计算机的名称，*UserName* 是有权访问服务器计算机的帐户的名称。

出现提示时，输入 *UserName* 的密码。

此命令成功后，可以使用 **-remote** 命令行选项或使用 WinDbg 图形界面激活调试客户端。

注意 可能需要在服务器计算机上启用文件和打印机共享。 在控制面板中，导航到**网络和 Internet**>**网络和共享中心**>**高级共享设置**。 选择**启用文件和打印机共享**。

port= *插座*
如果使用 TCP 或 SSL 协议，*Socket* 是创建服务器时使用的同一套接字端口号。

clicon
指定调试服务器将尝试通过反向连接连接到客户端。 当且仅当服务器使用的是 **clicon**

时，客户端才必须使用 **clicon**。 在大多数情况下，在使用反向连接时，会在调试服务器之前启动调试客户端。

port= *COMPort*
如果使用 COM 协议，*COMPort* 指定要使用的 COM 端口。 前缀“COM”是可选的 -- 例如，“com2”和“2”都可接受。

baud= *BaudRate*
如果使用 COM 协议，*BaudRate* 应与创建服务器时选择的波特率匹配。

channel= *COMChannel*
如果使用 COM 协议，*COMChannel* 应与创建服务器时选择的通道号匹配。

proto= *协议*
如果使用 SSL 或 SPIPE 协议，*Protocol* 应与创建服务器时使用的安全协议匹配。

证书
如果使用 SSL 或 SPIPE 协议，则应使用创建服务器时使用的相同 **certuser=***Cert* 或 **machuser=** *Cert* 参数。

password= *密码*
如果在创建服务器时使用了密码，则必须提供 *Password*，以便创建调试客户端。 它必须与原始密码匹配。 密码是区分大小写的。 如果提供的密码错误，错误消息将指定“错误 0x80004005”。密码长度必须为 12 个字符或更少。

ipversion=6
（仅适用于 Windows 调试工具 6.6.07 及更低版本）使用 TCP 连接到 Internet 时，强制调试器使用 IP 版本 6，而不是版本 4。 在 Windows Vista 和更高版本中，调试器会尝试自动默认为 IP 版本 6，因此不需要此选项。

调试客户端不能使用用于启动新调试会话的命令行选项（如 **-p**），只能由服务器使用。 配置选项（如 **-n**）将适用于客户端或服务器。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

客户端和服务器示例

项目 • 2024/03/08

假设一个人正在名为 *\\BOX17* 的计算机上运行应用程序。 此应用程序有问题，但调试技术人员位于其他站点。

第一个人在 *\\BOX17* 上使用 CDB 设置调试服务器。 目标应用程序的进程 ID 为 122。 选择 TCP 协议，套接字端口号为 1025。 在提升的命令提示符窗口（以管理员身份运行）中输入以下命令启动服务器：

控制台

```
E:\Debugging Tools for Windows> cdb -server tcp:port=1025 -p 122
```

在另一台计算机上，技术人员决定使用 WinDbg 作为调试客户端。 可以通过以下命令启动：

控制台

```
G:\Debugging Tools> windbg -remote tcp:server=BOX17,port=1025
```

以下是另一个示例。 在这种情况下，选择 NPIPE 协议，并使用 CDB 而不是 WinDbg。 第一个用户选择管道名称。 这可以是任何字母数字字符串，在本例中为“MainPipe”。 第一个用户打开提升的命令提示符窗口（以管理员身份运行），并通过输入以下命令启动调试服务器：

控制台

```
E:\Debugging Tools for Windows> cdb -server npipe:pipe=MainPipe -v
winmine.exe
```

技术人员使用无权访问服务器计算机的帐户登录到客户端计算机。 但技术人员知道可以访问服务器计算机的帐户的用户名和密码。 该帐户的用户名为 Contoso。 技术人员输入以下命令：

控制台

```
net use \\BOX17\ipc$ /user:Contoso
```

出现提示时，技术人员输入 Contoso 帐户的密码。

技术人员不确定命名管道的名称，因此他们查询 BOX17 以获取可用的调试服务器。

控制台

```
G:\Debugging Tools> cdb -QR \\BOX17
Servers on \\BOX17:
Debugger Server - npipe:Pipe=MainPipe
Remote Process Server - npipe:Pipe=AnotherPipe
```

显示了两个管道。 但是，只有一个是调试服务器，另一个是进程服务器，我们对此不感兴趣。 因此，**MainPipe** 必须是正确的名称。 技术人员使用以下命令启动调试客户端：

控制台

```
G:\Debugging Tools> cdb -remote npipe:server=BOX17,pipe=MyPipe
```

使用安全服务器

下面是安全服务器的示例。 该服务器使用安全套接字层，采用 TLS1 的 S-Channel 协议。 调试器将在计算机存储中查找证书。 证书由其十六进制指纹指定。

控制台

```
D:\> cdb -server "ssl:proto=tls1,machuser=ab 38 f7 ae 13 20 ac da 05 14 65
60 30 83 7b 83 09 2c d2 34,port=1234" notepad.exe
```

控制远程调试会话

项目 • 2023/10/26

启动远程会话后，可以在调试服务器或调试客户端中输入命令。 如果有多个客户端，则任何客户端都可以输入命令。 按 Enter 后，命令将传输到调试服务器并执行。

每当一个用户输入命令时，所有用户都会看到命令本身及其输出。 如果此命令是从调试客户端发出的，则所有其他用户都将在命令前面看到一个标识，该标识是哪个用户发出了命令。 从调试服务器发出的命令没有此前缀。

一个用户执行命令后，通过 KD 或 CDB 连接的其他用户将看不到新的命令提示符。 另一方面，WinDbg 的用户将在调试器命令窗口的底部面板中连续看到提示，即使调试器引擎正在运行也是如此。 这两者都不应引起恐慌：任何用户都可以随时输入命令，引擎将按照他们收到的顺序执行这些命令。

通过 WinDbg 接口执行的操作也将由调试服务器执行。

用户之间的通信

每当新的调试客户端连接到会话时，所有其他用户都会看到一条消息，指出此客户端已连接。 客户端断开连接时不显示任何消息。

.clients (列出调试客户端) 命令将列出当前连接到调试会话的所有客户端。

.echo (Echo Comment) 命令可用于将消息从一个用户发送到另一个用户。

WinDbg 工作区

当 WinDbg 用作调试客户端时，其工作区将仅保存通过图形界面设置的值。 不会保存通过“调试器命令”窗口所做的更改。 (这保证仅反映本地客户端所做的更改，因为调试器命令窗口将接受来自所有客户端以及调试服务器的输入。)

文件路径

符号路径、可执行映像路径和扩展 DLL 路径都解释为相对于调试服务器上的 Windows 调试工具安装文件夹的文件路径。

将 WinDbg 用作调试客户端时，它也有其自己的 *本地源路径* 。 所有与源相关的命令都将访问本地计算机上的源文件。 因此，必须在将使用源命令的任何客户端或服务器上设置正确的路径。

此多路径系统允许调试客户端使用与源相关的命令，而无需实际与其他客户端或服务器共享源文件。 如果有一个用户有权访问的私有或机密源文件，这很有用。

取消调试服务器

.endsrv (End Debugging Server) 命令可用于终止调试服务器。 如果调试器已建立多个调试服务器，则可以取消其中一些服务器，同时让其他服务器保持运行状态。

终止服务器将阻止将来的任何客户端附加到该服务器。 它不会切断当前通过服务器附加的任何客户端。

退出调试器并终止会话

若要退出一个调试客户端而不终止服务器，必须从该特定客户端发出命令。 如果此客户端是 KD 或 CDB，请使用 Ctrl+B 键退出。 如果使用脚本运行 KD 或 CDB，请使用 .remote_exit (**退出调试客户端**) 。 如果此客户端是 WinDbg，请从“**文件**”菜单中选择“**退出**”。

若要终止整个会话并退出调试服务器，请使用 q (Quit) 命令。 可以从任何服务器或客户端输入此命令，它将终止所有用户的整个会话。

通过内核调试程序控制用户模式调试程序

项目 • 2023/10/26

可以将用户模式调试器的输入和输出重定向到内核调试器。 此重定向使内核调试器能够控制目标计算机上发生的特定用户模式调试会话。

可以使用 KD 或 WinDbg 作为内核调试器。 请注意，WinDbg 的许多熟悉功能在此方案中不可用。 例如，不能使用“局部变量”窗口、“反汇编”窗口或“调用堆栈”窗口，并且不能单步执行源代码。 这是因为 WinDbg 仅充当目标计算机上运行的调试器 (NTSD 或 CDB) 的查看器。

可以使用 CDB 或 NTSD 作为用户模式调试器。 NTSD 是更好的选择，因为它需要最少的资源来自正在调试应用程序的计算机的处理器和操作系统。 事实上，在内核调试器的控制下启动 NTSD 时，不会创建 NTSD 窗口。 使用 NTSD，可以在启动阶段早期通过串行端口执行用户模式调试，并在关闭后期执行。

注意 将用户模式调试器的输出重定向到内核调试器时，不支持 **.shell** 命令。

本节包括下列内容：

- 启动调试会话 介绍了如何开始从内核调试器控制用户模式调试器的会话。
- 切换模式 介绍了涉及的四种不同模式，以及如何在它们之间交替。
- 何时使用此方法 描述了此方法特别有用的方案。
- 将此方法与远程调试相结合 介绍了如何从内核调试器控制用户模式调试器，并同时将其用作调试服务器。 如果用户模式符号位于符号服务器上，则此组合非常有用。

启动调试会话

项目 • 2024/07/08

在有关如何 从内核调试器控制用户模式调试的本文档中， *目标应用程序* 是指正在调试的用户模式应用程序， *目标计算机* 是指包含目标应用程序和 NTSD 或 CDB 进程的计算机， *而主计算机* 引用包含内核调试器的计算机。

若要开始使用此方法，必须执行以下操作。 可以按任一顺序执行步骤 1 和 2。

1. 使用 -d 命令行选项在目标计算机上启动 NTSD 或 CDB。

   例如，可以使用以下语法附加到正在运行的进程。

   ntsd -d [-y *UserSymbolPath*] **-p** *PID*

   或者，可以使用以下语法启动新进程作为目标。

   ntsd -d [-y *UserSymbolPath*] *ApplicationName*

   如果要将其安装为事后调试器，应使用以下语法。

   ntsd -d [-y UserSymbolPath*]*

   有关此步骤的详细信息，请参阅 使用 CDB 调试用户模式进程。

2. 在主计算机上启动 WinDbg 或 KD，就像要调试目标计算机一样，但实际上不会中断目标计算机。 若要使用 WinDbg，请使用以下语法。

   windbg [-y KernelSymbolPath*] [-k ConnectionOptions]*

   有关此步骤的详细信息，请参阅使用 WinDbg 的实时内核模式调试（经典版）。

   注意 如果使用 WinDbg 作为内核调试器，那么在这种情况下，WinDbg 的许多熟悉功能都不可用。 例如，不能使用“局部变量”窗口、“反汇编”窗口或“调用堆栈”窗口，也不能单步执行源代码。 这是因为 WinDbg 仅充当目标计算机上运行的调试器（NTSD 或 CDB）的查看器。

3. 如果尚未设置用户模式符号路径，请从“输入> ”提示符进行设置。 如果尚未设置内核模式符号路径，请从 kd> 提示符中设置它。 有关如何访问这些提示以及如何在模式之间切换的信息，请参阅 切换模式。

如果使用 CDB，则与 CDB 关联的命令提示符窗口在继续调试时将保持锁定状态且不可用。 如果使用 NTSD，即使 NTSD 在目标计算机上具有与之关联的进程 ID，也不会创建其他窗口。

如果要从内核调试器运行用户模式调试器，同时将其用作调试服务器，请参阅 将此方法与远程调试相结合。

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

切换模式

项目 • 2023/10/26

从内核调试器控制用户模式调试时，会遇到四种不同的模式，并且可以通过多种方式在它们之间切换。

注意 在描述此方案时， *目标应用程序* 是指正在调试的用户模式应用程序， *目标计算机* 是指包含目标应用程序和 CDB 或 NTSD 进程的计算机， *而主计算机* 是指包含内核调试器的计算机。

将遇到以下四种模式：

用户模式调试
目标计算机和目标应用程序已冻结。 用户模式调试提示显示在内核调试 器的"调试器命令"窗口中 。 在 WinDbg 中，WinDbg 窗口下面板上的提示会显示"输入>"。 可以在此提示符下输入命令（就像在用户模式调试期间输入命令一样），以分析目标应用程序的状态或使其运行或单步执行。 调试器访问的符号文件、扩展 DLL 和其他文件将是目标计算机上的文件，而不是主计算机上的文件。

目标应用程序执行
目标计算机正在运行，目标应用程序正在运行，调试器正在等待。 此模式与让目标在普通调试中运行相同。

睡眠模式
目标计算机正在运行，但目标应用程序已冻结，并且两个调试器都已冻结。 如果必须在目标计算机上执行某些操作，但不想更改调试会话的状态，则此模式非常有用。

内核模式调试
目标计算机和目标应用程序已冻结。 内核模式调试提示 kd> 显示在内核调试器的"调试器命令"窗口中。 此模式是典型的内核模式调试状态。

会话在用户模式调试模式下开始。 以下操作和事件会导致模式更改：

- 若要从用户模式调试切换到目标应用程序执行，请在提示符处`Input>`使用 **g (Go)** 命令。
- 若要暂时从用户模式调试切换到目标应用程序执行，然后返回到用户模式调试，请使用步骤、跟踪或其他临时执行命令。 有关此类命令的列表，请参阅 控制目标。
- 若要从用户模式调试切换到睡眠模式，请使用 **.sleep (暂停调试器)** 命令。 此命令已计时。 当时间过期时，系统将返回到用户模式调试。

- 若要从用户模式调试切换到内核模式调试，请使用 .breakin (Break to the Kernel Debugger) 命令。 请注意，如果调用进程没有管理员权限， .breakin 可能会失败并出现拒绝访问错误。 在这种情况下，通过发出简短 **的 .sleep** 命令并按 Ctrl+C 切换到 KD。

- 只能在某些环境中从目标应用程序执行切换到用户模式调试。 如果目标计算机运行的是 Microsoft Windows XP 或更高版本的 Windows 操作系统，则可以使用 **！ bpid** 扩展命令。 如果使用 CDB (而不是 NTSD) ，则可以在目标计算机上激活 CDB 窗口，然后按 Ctrl+C。

- 如果目标应用程序命中断点、遇到异常、遇到其他一些受控事件或结束，系统会从目标应用程序执行切换到用户模式调试。 应提前计划此类事件，尤其是在使用 NTSD 时。 有关这些事件的详细信息，请参阅使用断点和控制异常和事件。

- 若要从目标应用程序执行切换到内核模式调试，请在 KD 窗口中按 Ctrl+C，按 Ctrl+BREAK 或单击 WinDbg 窗口中**的“调试”**菜单上的“**中断**”，或按目标计算机键盘上的 SYSRQ 或 Alt+SYSRQ。 (如果内核调试器是 KD，并且你在内核调试器与用户模式调试器通信的同时按 Ctrl+C，则用户模式调试器可能会捕获按 CTRL+C.)

- 如果调试器遇到内核错误，或者如果使用 Breakin.exe 工具，系统会从目标应用程序执行切换到内核模式调试。

- 若要从睡眠模式切换到用户模式调试，请等待睡眠时间过期，使用 -wake **命令行选项**在目标计算机上启动新的 CDB 进程，或使用目标计算机上的其他 CDB 或 NTSD 副本中的 .wake (**唤醒调试器**) 命令。

- 若要退出内核模式调试，请在提示符处`kd>`使用 g (Go) 命令。 此命令返回到用户模式调试或目标应用程序执行 (两者中最近使用的状态) 。

何时使用此方法

项目 • 2023/06/15

在某些情况下， 从内核调试器控制用户模式调试很有用，甚至有必要：

- 当你需要执行用户模式调试时，还需要控制运行用户模式目标的 Windows 内核，或者需要使用某些内核模式调试功能来分析问题。
- 当用户模式目标是 Windows 进程（例如 CSRSS 或 WinLogon）时。 有关如何调试这些目标的详细说明，请参阅 调试 CSRSS 和 调试 WinLogon。
- 用户模式目标是服务应用程序时。 有关如何调试这些目标的详细说明，请参阅 调试服务应用程序。

将此方法与远程调试结合使用

项目 • 2024/06/04

有时从内核调试器控制用户模式调试器并同时使用用户模式调试器作为调试服务器很有用。

例如，当用户模式符号位于符号服务器上时，此配置就非常有用。 在从内核调试器控制用户模式调试器的标准配置中，两个调试器的交互可能会导致同步中的微小失误，而这些失误可能会阻止符号服务器身份验证。 此处所述的更复杂的配置可以避免此问题。

注意，在描述此方案时，*目标应用程序* 是指正在调试的用户模式应用程序，*目标计算机* 是指包含目标应用程序和 CDB 或 NTSD 进程的计算机，而*主计算机*是指包含内核调试器的计算机。

若要使用此技术，必须执行以下操作：

1. 使用 -ddefer 和 -server 命令行选项在目标计算机上启动 NTSD 或 CDB，并指定所需的传输选项。 -server 选项必须是命令行上的第一个参数。

   例如，可以使用以下语法附加到正在运行的进程。

   控制台

   ```
   ntsd -server ServerTransport -ddefer [-y UserSymbolPath] -p PID
   ```

   或者，可以使用以下语法启动新进程作为目标。

   控制台

   ```
   ntsd -server ServerTransport -ddefer [-y UserSymbolPath]
   ApplicationName
   ```

   如果要将其安装为事后调试器，应使用以下语法。 请注意，必须手动编辑注册表才能安装包含 -server 参数的事后调试器；有关详细信息，请参阅启用事后调试。

   控制台

   ```
   ntsd -server ServerTransport -ddefer [-y UserSymbolPath]
   ```

   有关可用传输选项的信息，请参阅**激活调试服务器**。

2. 在主计算机上启动 WinDbg 或 KD，就像要调试目标计算机一样，但实际上不会中断目标计算机。 若要使用 WinDbg，请使用以下语法。

控制台

```
windbg [-y KernelSymbolPath] [-k ConnectionOptions]
```

有关此步骤的详细信息，请参阅使用 WinDbg 的实时内核模式调试（经典版）

。

3. 使用与启动服务器相同的传输选项，将 WinDbg 或 CDB 作为调试客户端启动。 此调试客户端可以在主机或第三台计算机上运行。

控制台

```
cdb -remote ClientTransport
```

有关此步骤的详细信息，请参阅**激活调试客户端**。

4. 调试器运行并且`Input>`提示出现在内核调试器中后，使用**.sleep（暂停调试器）**命令暂停调试器，让目标计算机运行几秒钟。 这为目标计算机提供了处理远程传输协议的时间，从而在用户模式远程服务器和远程客户端之间建立连接。

如果使用 CDB 作为用户模式调试器，则与 CDB 关联的命令提示符窗口将保持锁定，在继续调试时不可用。 如果使用 NTSD，即使 NTSD 在目标计算机上具有与之关联的进程 ID，也不会创建其他窗口。

切换模式主题中描述的四种模式及其之间的切换方法在此组合方案中适用，但存在以下差异：

- 有两种不同的用户模式调试模式。 目标计算机运行时，调试服务器由调试客户端控制，就像在任何其他远程调试会话中一样；这称为*远程控制用户模式调试*。 当内核模式调试器损坏到目标计算机并显示 `Input>` 提示时，用户模式调试器由内核调试器控制；这称为 *内核控制的用户模式调试*。

- 这两种模式永远不会同时可用。 当内核调试器接入目标计算机时，即使用户模式调试器处于活动状态，目标计算机也无法处理远程传输协议，因此用户模式调试器无法通过此连接接收远程输入。

- 如果用户模式符号位于符号服务器上，任何需要访问符号的调试器命令都应在远程控制的用户模式调试模式下发出。

- 若要从内核控制的用户模式调试切换到远程控制的用户模式调试，请使用 **.sleep（暂停调试器）**命令。 当用户模式调试器从睡眠命令唤醒时，它将处于远程控制的用户模式调试模式。

- 若要从远程控制的用户模式调试切换到内核模式调试，请从 `Input>` 提示输入任何命令。 如果此提示不可见，请切换到内核模式调试，然后在 `kd>` 提示符处使用 g (Go) 命令。

在内部，以 -ddefer 启动的用户模式调试器优先于调试客户端的输入，其次才是来自内核调试器的输入。 但是，同时输入之间永远不会发生冲突，因为当内核调试器闯入目标计算机时，远程连接不可用。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

通过 Remote.exe 进行远程调试

项目 • 2023/06/15

通过remote.exe进行远程调试涉及在远程计算机上运行调试器，以及在本地计算机上运行remote.exe工具。

远程计算机和本地计算机可以运行任何 Windows 操作系统。

注意 由于 remote.exe 仅适用于控制台应用程序，因此不能用于远程控制 WinDbg。

本节包括：

Remote.exe 实用工具

启动 Remote.exe 会话

Remote.exe 批处理文件

Remote.exe 实用工具

项目 • 2023/06/14

remote.exe实用工具是一种通用的服务器/客户端工具，可用于在远程计算机上运行命令行程序。

Remote.exe通过命名管道向使用 STDIN 和 STDOUT 进行输入和输出的应用程序提供远程网络访问。 网络上其他计算机上的用户，或者通过直接拨号调制解调器连接进行连接的用户。 可以查看远程会话，也可以自行输入命令。

此实用工具具有大量用途。 例如，开发软件时，可以在计算机上执行其他任务时使用远程计算机的处理器和资源编译代码。 还可以使用 remote.exe 将特定任务的处理要求分发到多台计算机上。

请注意，remote.exe不进行安全授权，并且将允许运行 Remote.exe Client 的任何人连接到Remote.exe服务器。 这会使运行 Remote.exe Server 的帐户对连接的任何用户开放。

启动 Remote.exe 会话

项目 • 2023/10/26

可通过两种方式使用 KD 或 CDB 启动 remote.exe 会话。 只有其中第二种方法适用于 NTSD。

自定义命令提示符窗口

Remote.exe 客户端和 Remote.exe 服务器在命令提示符窗口中运行。

若要准备远程会话，应自定义此窗口以提高其可用性。 打开“命令提示”窗口。 右键单击标题栏，然后选择“ **属性**”。 选择“**布局**”选项卡。转到标题为“屏幕缓冲区大小”的部分，在“**宽度**”框中键入 **90**，并在“**高度**”框中键入**介于 4000 和 9999** 之间的值。 这会在内核调试器上的远程会话中启用滚动条。

如果要更改命令提示符的形状，请更改“Windows 大小”部分的高度和宽度的值。 选择“**选项** ”选项卡。启用 “**编辑选项** ”快速编辑模式和插入模式。 这样，就可以在命令提示符会话中剪切和粘贴信息。 单击“确定”应用更改。 在出现提示时，选择选项以将更改应用到所有将来的会话。

启动 Remote.exe 服务器：First 方法

启动 Remote.exe 服务器的常规语法如下：

控制台

```
remote /s "Command_Line" Unique_Id [/f Foreground_Color] [/b
Background_Color]
```

这可用于在远程计算机上启动 KD 或 CDB，如以下示例所示：

控制台

```
remote /s "KD [options]" MyBrokenBox

remote /s "CDB [options]" MyBrokenApp
```

这会在命令提示符窗口中启动 Remote.exe 服务器，并启动调试器。

不能使用此方法直接启动 NTSD，因为 NTSD 进程在与调用它的窗口不同的窗口中运行。

启动 Remote.exe 服务器：第二种方法

有一种备用方法可以启动 Remote.exe 服务器。 此方法涉及首先启动调试器，然后使用 .remote (Create Remote.exe Server) 命令启动服务器。

由于 **.remote** 命令是在调试器启动后发出的，因此此方法同样适用于 KD、CDB 和 NTSD。

示例如下。 首先，以正常方式启动调试器：

控制台

```
KD [options]
```

调试器运行后，请使用 **.remote** 命令：

dbgcmd

```
.remote MyBrokenBox
```

这会导致 KD 进程也是 ID 为“MyBrokenBox”的 Remote.exe 服务器，这与第一种方法完全一样。

此方法的一个优点是，无需事先决定是否要使用远程调试。 如果使用其中一个控制台调试器进行调试，然后决定希望远程位置的某人接手，则可以使用 **.remote** 命令，然后他们可以连接到会话。

启动 Remote.exe 客户端

启动 Remote.exe 客户端的一般语法如下：

控制台

```
remote /c ServerNetBIOSName Unique_ID [/l Lines_to_Get] [/f
Foreground_Color] [/b Background_Color]
```dbgcmd

For example, if the "MyBrokenBox" session, described above, was started on a
local host computer whose network name was "Server2", you can connect to it
with the command:

```console
remote /c server2 MyBrokenBox
```

网络上具有适当权限的任何人都可以连接到此调试会话，只要他们知道你的计算机名称和会话 ID。

发出命令

命令通过 Remote.exe 客户端发出，并发送到 Remote.exe 服务器。 可以将任何命令输入客户端，就像直接将命令输入调试器一样。

若要退出 Remote.exe 客户端上的 remote.exe 会话，请输入 **@Q** 命令。 这会使 Remote.exe 服务器和调试器保持运行。

若要结束服务器会话，请在 **Remote.exe** 服务器上输入 @K 命令。

Remote.exe 批处理文件

项目 • 2023/06/15

作为使用 remote.exe 进行远程调试的更详细示例，假设在三台计算机内核调试方案中本地主计算机存在以下情况：

- 调试需要通过 COM2 上的 null 调制解调器电缆进行。
- 符号文件位于 c：\winnt\symbols 文件夹中。
- 在 **c：\temp** 中创建名为 debug.log 的日志文件。

日志文件保存调试会话期间在“调试”屏幕上看到的所有内容的副本。 执行调试的人员的所有输入以及目标系统上内核调试器的所有输出都会写入该日志文件。

在本地主机上运行调试会话的示例批处理文件是：

bat

```
set _NT_DEBUG_PORT=com2
set _NT_DEBUG_BAUD_RATE=19200
set _NT_SYMBOL_PATH=c:\winnt\symbols
set _NT_LOG_FILE_OPEN=c:\temp\debug.log
remote /s "KD -v" debug
```

注意 如果此批处理文件与 Remote.exe 不在同一目录中，并且Remote.exe不在系统路径中列出的目录中，则应在此批处理文件中调用Remote.exe时提供实用工具的完整路径。

运行此批处理文件后，使用连接到本地主机的 Windows 计算机的任何人都可以使用以下命令连接到调试会话：

控制台

```
remote /c computername debug
```

其中 *computername* 是本地主机的 NetBIOS 名称。

进程服务器（用户模式）

项目 • 2023/06/14

通过进程服务器进行远程调试涉及在服务器计算机上运行名为 *进程服务器的* 小型应用程序。 然后，在客户端计算机上启动用户模式调试器。 由于此调试器将执行所有实际处理，因此称为 *智能客户端*。

Windows 调试工具包包含名为 DbgSrv (dbgsrv.exe 的进程服务器，) 在用户模式下使用。

这两台计算机不必运行相同版本的 Windows;它们可以运行任何版本的 Windows。 但是，客户端上使用的调试器二进制文件和服务器上使用的 DbgSrv 二进制文件必须来自 Windows 调试工具包的同一版本。 此方法不能用于转储文件调试。

若要设置此远程会话，请先设置进程服务器，然后激活智能客户端。 任意数量的智能客户端都可以通过单个进程服务器运行 -- 这些调试会话将保持独立，不会相互干扰。 如果调试会话结束，进程服务器将继续运行，并可用于新的调试会话。

本部分中的内容

- **激活进程服务器**
- **搜索进程服务器**
- **激活智能客户端**
- 进程服务器示例
- 控制进程服务器会话

激活进程服务器

项目 • 2024/03/01

Windows 调试工具中包含的进程服务器称为 DbgSrv (dbgsrv.exe)。 若要激活进程服务器，请打开提升的命令提示符窗口（以管理员身份运行），然后输入 **dbgsrv** 命令。

注意 可以在没有提升的特权的情况下激活进程服务器，并且调试客户端将能够连接到服务器。 但是，除非客户端使用了提升的特权激活，否则客户端将无法发现进程服务器。 有关如何发现调试服务器的信息，请参阅搜索进程服务器。

DbgSrv 支持多种传输协议：命名管道 (NPIPE)、TCP、COM 端口、安全管道 (SPIPE) 和安全套接字层 (SSL)。

控制台

```
dbgsrv -t npipe:pipe=PipeName[,hidden][,password=Password][,IcfEnable] [[-
sifeo Executable] -c[s] AppCmdLine] [-x | -pc]

dbgsrv -t tcp:port=Socket[,hidden][,password=Password][,ipversion=6]
[,IcfEnable] [[-sifeo Executable] -c[s] AppCmdLine] [-x | -pc]

dbgsrv -t tcp:port=Socket,clicon=Client[,password=Password][,ipversion=6]
[[-sifeo Executable] -c[s] AppCmdLine] [-x | -pc]

dbgsrv -t com:port=COMPort,baud=BaudRate,channel=COMChannel[,hidden]
[,password=Password] [[-sifeo Executable] -c[s] AppCmdLine] [-x | -pc]

dbgsrv -t spipe:proto=Protocol,
{certuser=Cert|machuser=Cert},pipe=PipeName[,hidden][,password=Password] [[-
sifeo Executable] -c[s] AppCmdLine] [-x | -pc]

dbgsrv -t ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},port=Socket[,hidden][,password=Password] [[-
sifeo Executable] -c[s] AppCmdLine] [-x | -pc]

dbgsrv -t ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},port=Socket,clicon=Client[,password=Password]
[[-sifeo Executable] -c[s] AppCmdLine] [-x | -pc]
```

上述命令中的参数具有以下可能的值：

pipe=*PipeName*
使用 NPIPE 或 SPIPE 协议时，*PipeName* 是将用作管道名称的字符串。 每个管道名称都应标识唯一的进程服务器。 如果尝试重用管道名称，将收到一条错误消息。 *PipeName* 不得包含空格或引号。 *PipeName* 可以包含数字 **printf** 样式的格式代码，如 **%x** 或 **%d**。

进程服务器将用 DbgSrv 的进程 ID 替换它。 第二个此类代码将被替换为 DbgSrv 的线程 ID。

注意 在运行进程服务器的计算机上，可能需要启用文件和打印机共享。 在控制面板中，导航到**网络和 Internet**>**网络和共享中心**>**高级共享设置**。 选择**启用文件和打印机共享**。

port=*Socket*
使用 TCP 或 SSL 协议时，*Socket* 是套接字端口号。

还可以指定用冒号分隔的端口范围。 DbgSrv 将检查此范围内的每个端口，以了解是否可用。 如果发现可用端口且未出错，则将创建进程服务器。 智能客户端必须指定要用于连接到服务器的实际端口。 若要确定实际端口，请使用**搜索进程服务器**中介绍的任一方法；显示此进程服务器时，该端口后面将紧跟两个用冒号分隔的数字。 第一个数字将是使用的实际端口；可以忽略第二个数字。 例如，如果将端口指定为 **port=51:60**，并且实际上使用了端口 53，则搜索结果将显示"port=53:60"。 （如果使用 **clicon** 参数建立反向连接，智能客户端可以采用这种方式指定端口范围，而 KD 进程服务器必须指定使用的实际端口。）

clicon=*Client*
使用 TCP 或 SSL 协议并指定 **clicon** 参数时，将打开*反向连接*。 这意味着，进程服务器将尝试连接到智能客户端，而不是让客户端启动联系。 如果防火墙阻止了正常方向的连接，这将非常有用。 *Client* 指定智能客户端所在或将创建智能客户端的计算机的网络名称或 IP 地址。 两个初始反斜杠 (\) 是可选的。

由于进程服务器正在查找特定客户端，因此如果使用此方法，则无法将多个客户端连接到服务器。 如果连接被拒绝或已断开，则必须重启进程服务器。 当有人使用 **-QR** 命令行选项显示所有活动服务器时，反向连接进程服务器将不会显示。

注意 在使用 **clicon** 时，最好在创建进程服务器之前启动智能客户端，但也允许遵循正常顺序（服务器在客户端之前）。

port=*COMPort*
使用 COM 协议时，*COMPort* 指定要使用的 COM 端口。 前缀"COM"是可选的 -- 例如，"com2"和"2"都可接受。

baud=*BaudRate*
使用 COM 协议时，*BaudRate* 指定连接运行时的波特率。 允许使用硬件支持的任何波特率。

channel=*COMChannel*
如果使用 COM 协议，*COMChannel* 指定要用于与调试客户端通信的 COM 通道。 这可以是 0 和 254 之间（含）的任何值。 可以使用不同的通道号将单个 COM 端口用于多个连接。 （这不同于将 COM 端口用于调试电缆，在那种情况下，不能在 COM 端口中使用通道。）

proto=*Protocol*
如果使用 SSL 或 SPIPE 协议，*Protocol*将指定安全通道（S 通道）协议。 这可以是字符串 tls1、pct1、ssl2 或 ssl3 中的任一个。

Cert
如果使用 SSL 或 SPIPE 协议，*Cert* 指定证书。 这可以是证书名称或证书的指纹（证书管理单元提供的十六进制数字字符串）。 如果使用语法 **certuser**=*Cert*，调试器将在系统存储区（默认存储区）中查找证书。 如果使用语法 **machuser**=*Cert*，调试器将在计算机存储区中查找证书。 指定的证书必须支持服务器身份验证。

hidden
当有人使用 **-QR** 命令行选项显示所有活动服务器时，会阻止进程服务器显示。

password=*Password*
要求智能客户端提供指定的密码，以便连接到进程服务器。 *Password* 可以是任意字母数字字符串，长度最多为 12 个字符。

警告 将密码与 TCP、NPIPE 或 COM 协议配合使用时，仅提供少量的保护，因为密码未加密。 将密码与 SSL 或 SPIPE 协议配合使用时，将对其进行加密。 如果要建立安全的远程会话，则必须使用 SSL 或 SPIPE 协议。

ipversion=6
（仅适用于 Windows 调试工具 6.6.07 及更低版本）使用 TCP 连接到 Internet 时，强制调试器使用 IP 版本 6，而不是版本 4。 在 Windows Vista 和更高版本中，调试器会尝试自动默认为 IP 版本 6，因此不需要此选项。

IcfEnable
促使调试器在 Internet 连接防火墙处于活动状态时启用 TCP 或命名管道通信所需的端口连接。 默认情况下，Internet 连接防火墙会禁用这些协议使用的端口。 将 **IcfEnable** 与 TCP 连接配合使用时，调试器会导致 Windows 打开 *Socket* 参数指定的端口。 将 **IcfEnable** 与命名管道连接配合使用时，调试器会促使 Windows 打开用于命名管道的端口（端口 139 和 445）。 连接终止后，调试器不会关闭这些端口。

-sifeo*Executable*
挂起给定映像的映像文件执行选项 (IFEO) 值。 *可执行文件* 应包括可执行文件映像的文件名，包括文件扩展名。 **-sifeo** 选项允许将 DbgSrv 设置为 **-c** 选项创建的映像的 IFEO 调试器，而不会由于 IFEO 设置导致递归调用。 仅当使用 **-c** 时，才能使用此选项。

-c
使 DbgSrv 创建新进程。 可以使用此它来创建要调试的进程。 这类似于从调试器生成新进程，只是在创建该进程时不会对其进行调试。 若要调试此过程，请确定其 PID，并在启动智能客户端时使用 **-p** 选项调试该进程。

s
使新创建的进程立即挂起。 如果使用此选项，建议使用 CDB 作为智能客户端，并使用 **-pb** 命令行选项和 **-p PID** 结合启动智能客户端。 如果在命令行中包含 **-pb** 选项，则进程将在调试器附加到它时恢复；否则可以使用 **~*m** 命令恢复进程。

AppCmdLine
指定要创建的进程的完整命令行。 *AppCmdLine* 可以是 Unicode 或 ASCII 字符串，并且可以包含任何可打印字符。 在 **-c[s]** 参数之后出现的所有文本都用来构成字符串 AppCmdLine。

-x
导致命令行的其余部分被忽略。 如果从可能将不需要的文本附加到其命令行的应用程序启动 DbgSrv，此选项非常有用。

-pc
导致命令行的其余部分被忽略。 如果从可能将不需要的文本附加到其命令行的应用程序启动 DbgSrv，此选项非常有用。 如果 **-pc** 是 DbgSrv 命令行上的最终元素，则会出现语法错误。 除了此限制之外， **-pc** 与 **-x** 相同。

您可以在一台计算机上启动任意数量的进程服务器。 但是，这通常是不必要的，因为一个进程服务器可供任意数量的智能客户端使用（每个客户端都参与不同的调试会话）。

搜索进程服务器

项目 • 2023/04/14

可以将 KD 或 CDB 与 **-QR** 命令行选项结合使用，以获取在网络服务器计算机上运行的可用进程服务器的列表。

此列表可能包含不再存在但未正确关闭的服务器 -- 连接到其中一个服务器会生成错误消息。 该列表还将包括调试服务器和 KD 连接服务器。 在每种情况下，都会指示服务器类型。

此配置的语法如下：

控制台

```
Debugger -QR \\Server
```

调试器 可以是 KD 或 CDB -- 在任一情况下，显示都相同。 (前一个*服务器*) \\ 的两个反斜杠是可选的。

在 WinDbg 中，可以使用" **连接到远程存根服务器** "对话框浏览可用进程服务器的列表。

注意 要使进程服务器可发现，必须使用提升的权限激活它。 有关详细信息，请参阅 激活进程服务器。

注意 如果未使用有权访问服务器计算机的帐户登录到客户端计算机，则可能需要提供用户名和密码。 在客户端计算机的命令提示符窗口中，输入以下命令。

net 使用 ***Server*\ipc$ /user：*UserName* ，其中 *Server* 是服务器计算机的名称，*UserName* 是有权访问服务器计算机的帐户的名称。

出现提示时，输入 *UserName* 的密码。

此命令成功后，可以使用 **-QR** 或 **连接到远程存根**发现) 在服务器计算机上运行的进程服务器 (。

激活智能客户端

项目 • 2024/07/08

激活 DbgSrv 进程服务器后，可以在另一台计算机上创建智能客户端并开始调试会话。

可通过两种方法启动智能客户端：使用 -premote 命令行选项启动 CDB 或 WinDbg，或者使用 WinDbg 图形界面。

智能客户端的协议必须与进程服务器的协议匹配。 用于启动智能客户端的常规语法取决于使用的协议。 存在以下选项：

控制台

```
Debugger -premote npipe:server=Server,pipe=PipeName[,password=Password]
[Options]

Debugger -premote tcp:server=Server,port=Socket[,password=Password]
[,ipversion=6] [Options]

Debugger -premote tcp:clicon=Server,port=Socket[,password=Password]
[,ipversion=6] [Options]

Debugger -premote
com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=Password]
[Options]

Debugger -premote spipe:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,pipe=PipeName[,password=Password
] [Options]

Debugger -premote ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,port=Socket[,password=Password]
[Options]

Debugger -premote ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},clicon=Server,port=Socket[,password=Password]
[Options]
```

若要使用图形界面连接到进程服务器，WinDbg 必须处于休眠模式，即必须已在未使用任何命令行参数的情况下启动，或者必须已结束上一个调试会话。 选择**文件 |连接到远程存根**菜单命令。 出现“**连接到存根服务器**”对话框时，在**连接字符串**文本框中输入以下字符串之一：

dbgcmd

```
npipe:server=Server,pipe=PipeName[,password=Password]

tcp:server=Server,port=Socket[,password=Password][,ipversion=6]
```

```
tcp:clicon=Server,port=Socket[,password=Password][,ipversion=6]

com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=Password]

spipe:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,pipe=PipeName[,password=Password
]

ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,port=Socket[,password=Password]

ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},clicon=Server,port=Socket[,password=Password]
```

或者，可以使用**浏览**按钮找到活动进程服务器。

上述命令中的参数具有以下可能的值：

调试器
这可以是 CDB 或 WinDbg。

服务器
这是在其中创建进程服务器的计算机的网络名称或 IP 地址。 两个初始反斜杠 (\) 在命令行上是可选的，但在 WinDbg 对话框中不允许使用。

pipe= *PipeName*
如果使用 NPIPE 或 SPIPE 协议，*PipeName* 是创建进程服务器时赋予管道的名称。

如果未使用有权访问服务器计算机的帐户登录到客户端计算机，则必须提供用户名和密码。 在客户端计算机的命令提示符窗口中输入以下命令。

net use ***Server*\ipc$ /user:***UserName*

其中 *Server* 是服务器计算机的名称，*UserName* 是有权访问服务器计算机的帐户的名称。

出现提示时，输入 *UserName* 的密码。

此命令成功后，可以使用 **-premote** 命令行选项或使用 WinDbg 图形界面激活调试智能客户端。

注意 可能需要在服务器计算机上启用文件和打印机共享。 在控制面板中，导航到**网络和 Internet**>**网络和共享中心**>**高级共享设置**。 选择**启用文件和打印机共享**。

port= *插座*
如果使用 TCP 或 SSL 协议，*Socket* 是创建进程服务器时使用的同一套接字端口号。

clicon
指定进程服务器将尝试通过反向连接连接到智能客户端。 当且仅当服务器使用的是 **clicon** 时，客户端才必须使用 **clicon**。 在大多数情况下，在使用反向连接时，会在进程服务器之前启动智能客户端。

port= *COMPort*
如果使用 COM 协议，*COMPort* 指定要使用的 COM 端口。 前缀“COM”是可选的 -- 例如，“com2”和“2”都可接受。

baud= *BaudRate*
如果使用 COM 协议，*BaudRate* 应与创建进程服务器时选择的波特率匹配。

channel= *COMChannel*
如果使用 COM 协议，*COMChannel* 应与创建进程服务器时选择的通道号匹配。

proto= *协议*
如果使用 SSL 或 SPIPE 协议，*Protocol* 应与创建进程服务器时使用的安全协议匹配。

Cert
如果使用 SSL 或 SPIPE 协议，则应使用创建进程服务器时使用的相同 **certuser**=*Cert* 或 **machuser**=*Cert* 参数。

password= *密码*
如果在创建进程服务器时使用了密码，则必须提供 *Password*，以便创建调试智能客户端。 它必须与原始密码匹配。 密码是区分大小写的。 如果提供了错误的密码，错误消息将指定“Error 0x80004005”。

ipversion=6
（仅适用于 Windows 调试工具 6.6.07 及更低版本）使用 TCP 连接到 Internet 时，强制调试器使用 IP 版本 6，而不是版本 4。 在 Windows Vista 和更高版本中，调试器会尝试自动默认为 IP 版本 6，因此不需要此选项。

选项
可以在此处放置任何其他命令行参数。 有关完整列表，请参阅命令行选项。 如果使用 CDB，则必须指定要调试的进程。 如果使用 WinDbg，可以在命令行上或通过图形界面指定进程。

由于进程服务器只充当智能客户端的网关，因此，附加*选项*将与在目标应用程序相同的计算机上启动用户模式调试器时使用的选项相同。

如果将 **-premote** 选项与 **.attach（附加到进程）**或 **.create（创建进程）**，一起使用，则参数与上面列出的参数相同。

故障排除

如果您看到此消息：*客户端未使用与服务器相同版本的远程处理协议*，这表明尝试连接的 DbgSrv 版本使用的协议版本与 WinDbg 版本不同。

协议更改的情况并不常见。 发生这种情况时，请确保使您使用的是最新可用版本的 DbgSrv 和 WinDbg（经典版）或 WinDbg 的匹配版本。 有关下载最新版本的信息，请参阅 Windows 调试工具。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

进程服务器示例

项目 • 2023/06/15

假设一个人在名为 \\BOX17 的计算机上运行应用程序。 此应用程序有问题，但调试技术人员位于不同的站点。

第一个人在 \\BOX17 上使用 DbgSrv 设置进程服务器。 目标应用程序的进程 ID 为 122。 选择 TCP 协议，套接字端口号为 1025。 使用以下命令启动服务器：

控制台

```
E:\Debugging Tools for Windows> dbgsrv -t tcp:port=1025
```

在其他计算机上，技术人员使用以下命令将 WinDbg 作为智能客户端启动：

控制台

```
G:\Debugging Tools> windbg -premote tcp:server=BOX17,port=1025 -p 122
```

以下是另一个示例。 在这种情况下，选择 NPIPE 协议，并使用 CDB 而不是 WinDbg。 第一个用户选择管道名称。 这可以是任何字母数字字符串 -- 在此示例中为“AnotherPipe”。 第一个用户 (以管理员身份运行) 打开提升的命令提示符窗口，并通过输入以下命令启动调试服务器：

控制台

```
E:\Debugging Tools for Windows> dbgsrv -t npipe:pipe=AnotherPipe
```

技术人员使用无权访问服务器计算机的帐户登录到客户端计算机。 但技术人员知道有权访问服务器计算机的帐户的用户名和密码。 该帐户的用户名为 Contoso。 技术人员输入以下命令：

控制台

```
net use \\BOX17\ipc$ /user:Contoso
```

出现提示时，技术人员输入 Contoso 帐户的密码。

技术人员不确定命名管道使用的名称，因此他们查询 BOX17 以查找进程服务器：

控制台

```
G:\Debugging Tools> cdb -QR \\BOX17
Servers on \\BOX17:
Debugger Server - npipe:Pipe=MainPipe
Remote Process Server - npipe:Pipe=AnotherPipe
```

显示两个管道。 但是，只有一个是进程服务器 -- 另一个是调试服务器，对此不感兴趣。 因此， **AnotherPipe** 必须是正确的名称。 技术人员输入以下命令以启动智能客户端：

控制台

```
G:\Debugging Tools> cdb -premote npipe:server=BOX17,pipe=AnotherPipe -v
sol.exe
```

有关使用进程服务器的更复杂的示例，请参阅 中间符号。

控制进程服务器会话

项目 • 2023/10/26

启动远程会话后，可以使用智能客户端，就像在单个计算机上调试目标应用程序一样。 所有命令的行为都与这种情况相同，但路径相对于智能客户端的计算机除外。

将 WinDbg 用作智能客户端

将 WinDbg 作为用户模式进程服务器的智能客户端启动后，它将永久附加到进程服务器。 如果调试会话已结束，则 **文件 |附加到进程** 菜单命令或 **.tlist (列出进程 ID)** 命令将显示运行进程服务器的计算机上运行的所有进程。 WinDbg 可以附加到其中任何一个进程。

文件 |无法打开可执行文件]命令。 只有在 WinDbg 命令行中包含新进程时，才能生成新进程。

在这种情况下，WinDbg 将无法在运行它的计算机上调试进程，也无法启动内核调试会话。

结束会话

CDB 或 WinDbg 可以正常退出或结束调试会话。 有关详细信息，请参阅在 WinDbg (经典) 中结束调试会话 。 进程服务器将保持运行状态，并且可以根据需要多次重用。 (它还可用于任意数量的同时调试会话。)

可以从任一计算机终止进程服务器。 若要从智能客户端终止它，请使用 **.endpsrv (End Process Server)** 命令。 若要从运行进程服务器的计算机终止进程服务器，请使用任务管理器结束 dbgsrv.exe 进程。

KD 连接服务器（内核模式）

项目 • 2023/06/15

通过 KD 连接服务器进行内核模式远程调试涉及在服务器上运行名为 *KD 连接服务器的* 小型应用程序。 然后，在客户端上启动内核模式调试器。 由于此调试器将执行所有实际处理，因此称为 *智能客户端*。

Windows 调试工具包包含名为 KdSrv (kdsrv.exe) 的 KD 连接服务器。

这两台计算机不必运行相同版本的 Windows;它们可以运行任何版本的 Windows。 但是，客户端上使用的调试器二进制文件和服务器上使用的 KdSrv 二进制文件必须来自 Windows 调试工具包的同一版本。 此方法不能用于转储文件调试。

若要设置此远程会话，请先设置 KD 连接服务器，然后激活智能客户端。 任意数量的智能客户端都可以通过单个 KD 连接服务器运行，但它们必须分别连接到不同的内核调试会话。

本节包括：

激活 KD 连接服务器

搜索 KD 连接服务器

激活智能客户端（内核模式）

KD 连接服务器示例

控制 KD 连接服务器会话

激活 KD 连接服务器

项目 • 2024/07/08

Windows 调试工具中包含的 KD 连接服务器名为 KdSrv (kdsrv.exe)。 若要激活 KD 连接服务器，请打开提升的命令提示符窗口（以管理员身份运行），然后输入 **kdsrv** 命令。

注意 可以在没有提升的特权的情况下激活 KD 连接服务器，并且调试客户端将能够连接到服务器。 但是，除非客户端使用了提升的特权激活，否则客户端将无法发现 KD 连接服务器。 有关如何发现调试服务器的信息，请参阅**搜索 KD 连接服务器**。

KdSrv 支持多种传输协议：命名管道 (NPIPE)、TCP、COM 端口、安全管道 (SPIPE) 和安全套接字层 (SSL)。

KdSrv 命令行的语法取决于使用的协议。 存在以下选项：

控制台

```
kdsrv -t npipe:pipe=PipeName[,hidden][,password=Password][,IcfEnable]

kdsrv -t tcp:port=Socket[,hidden][,password=Password][,ipversion=6]
[,IcfEnable]

kdsrv -t tcp:port=Socket,clicon=Client[,password=Password][,ipversion=6]

kdsrv -t com:port=COMPort,baud=BaudRate,channel=COMChannel[,hidden]
[,password=Password]

kdsrv -t spipe:proto=Protocol,
{certuser=Cert|machuser=Cert},pipe=PipeName[,hidden][,password=Password]

kdsrv -t ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},port=Socket[,hidden][,password=Password]

kdsrv -t ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},port=Socket,clicon=Client[,password=Password]
```

上述命令中的参数具有以下可能的值：

pipe= *PipeName*
使用 NPIPE 或 SPIPE 协议时，*PipeName* 是将用作管道名称的字符串。 每个管道名称都应标识唯一的进程服务器。 如果尝试重用管道名称，将收到一条错误消息。 *PipeName* 不得包含空格或引号。 *PipeName* 可以包含数字 **printf** 样式的格式代码，如 **%x** 或 **%d**。 这将替换为 KdSrv 的进程 ID。 第二个此类代码将被替换为 KdSrv 的线程 ID。

port= *插座*
使用 TCP 或 SSL 协议时，*Socket* 是套接字端口号。

还可以指定用冒号分隔的端口范围。 KdSrv 将检查此范围内的每个端口，以了解是否可用。 如果发现可用端口且未出错，则将创建 KD 连接服务器。 智能客户端必须指定要用于连接到服务器的实际端口。 若要确定实际端口，请使用**搜索 KD 连接服务器**中介绍的任一方法；显示此 KD 连接服务器时，该端口后面将紧跟两个用冒号分隔的数字。 第一个数字将是使用的实际端口；可以忽略第二个数字。 例如，如果将端口指定为 **port=51:60**，并且实际上使用了端口 53，则搜索结果将显示“port=53:60”。 （如果使用 **clicon** 参数建立反向连接，智能客户端可以采用这种方式指定端口范围，而 KD 连接服务器必须指定使用的实际端口。）

clicon= *客户*
使用 TCP 或 SSL 协议并指定 **clicon** 参数时，将打开*反向连接*。 这意味着，KD 连接服务器将尝试连接到智能客户端，而不是让客户端启动联系。 如果防火墙阻止了正常方向的连接，这将非常有用。 *Client* 指定智能客户端所在或将创建智能客户端的计算机的网络名称或 IP 地址。 两个初始反斜杠 (\) 是可选的。

由于 KD 连接服务器正在查找特定客户端，因此如果使用此方法，则无法将多个客户端连接到服务器。 如果连接被拒绝或已断开，则必须重启进程服务器。 当有人使用 **-QR** 命令行选项显示所有活动服务器时，反向连接 KD 连接服务器将不会显示。

注意 在使用 **clicon** 时，最好在创建 KD 连接服务器之前启动智能客户端，但也允许遵循正常顺序（服务器在客户端之前）。

port= *COMPort*
使用 COM 协议时，*COMPort* 指定要使用的 COM 端口。 前缀“COM”是可选的 -- 例如，“com2”和“2”都可接受。

baud= *BaudRate*
使用 COM 协议时，*BaudRate* 指定连接运行时的波特率。 允许使用硬件支持的任何波特率。

channel= *COMChannel*
如果使用 COM 协议，*COMChannel* 指定要用于与调试客户端通信的 COM 通道。 这可以是 0 和 254 之间（含）的任何值。 可以使用不同的通道号将单个 COM 端口用于多个连接。 （这不同于将 COM 端口用于调试电缆，在那种情况下，不能在 COM 端口中使用通道。）

proto= *协议*
如果使用 SSL 或 SPIPE 协议，*Protocol*将指定安全通道（S 通道）协议。 这可以是字符串 tls1、pct1、ssl2 或 ssl3 中的任一个。

证书
如果使用 SSL 或 SPIPE 协议，*Cert* 指定证书。 这可以是证书名称或证书的指纹（证书管理单元提供的十六进制数字字符串）。 如果使用语法 **certuser=***Cert*，调试器将在系统存

储区（默认存储区）中查找证书。 如果使用语法 **machuser**=*Cert*，调试器将在计算机存储区中查找证书。 指定的证书必须支持服务器身份验证。

隐藏
当有人使用 **-QR** 命令行选项显示所有活动服务器时，会阻止 KD 连接服务器显示。

password= *密码*
要求智能客户端提供指定的密码，以便连接到 KD 连接服务器。 *Password* 可以是任意字母数字字符串，长度最多为 12 个字符。

警告 将密码与 TCP、NPIPE 或 COM 协议配合使用时，仅提供少量的保护，因为密码未加密。 将密码与 SSL 或 SPIPE 协议配合使用时，将对其进行加密。 如果要建立安全的远程会话，则必须使用 SSL 或 SPIPE 协议。

ipversion=6
（仅适用于 Windows 调试工具 6.6.07 及更低版本）使用 TCP 连接到 Internet 时，强制调试器使用 IP 版本 6，而不是版本 4。 在 Windows Vista 和更高版本中，调试器会尝试自动默认为 IP 版本 6，因此不需要此选项。

IcfEnable
促使调试器在 Internet 连接防火墙处于活动状态时启用 TCP 或命名管道通信所需的端口连接。 默认情况下，Internet 连接防火墙会禁用这些协议使用的端口。 将 **IcfEnable** 与 TCP 连接配合使用时，调试器会导致 Windows 打开 *Socket* 参数指定的端口。 将 **IcfEnable** 与命名管道连接配合使用时，调试器会促使 Windows 打开用于命名管道的端口（端口 139 和 445）。 连接终止后，调试器不会关闭这些端口。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

搜索 KD 连接服务器

项目 • 2023/04/14

可以将 KD 或 CDB 与 **-QR** 命令行选项一起使用，以获取在网络服务器上运行的可用 KD 连接服务器的列表。

此列表可能包括不再存在但未正确关闭的服务器 -- 连接到其中一个服务器会生成错误消息。 该列表还将包括调试服务器和用户模式进程服务器。 在每种情况下都会指示服务器类型。

此配置的语法如下：

控制台

```
Debugger -QR \\Server
```

调试器 可以是 KD 或 CDB -- 无论哪种情况，显示器都相同。 (*server 前面的*\\ 两个反斜杠) 是可选的。

在 WinDbg 中，可以使用“ **连接到远程存根服务器** ”对话框浏览可用 KD 连接服务器的列表。

注意 若要发现 KD 连接服务器，必须使用提升的权限将其激活。 有关详细信息，请参阅激活 KD 连接服务器。

注意 如果未使用有权访问服务器计算机的帐户登录到客户端计算机，则可能需要提供用户名和密码。 在客户端计算机上的命令提示符窗口中，输入以下命令。

net use ***Server*ipc$ /user:** *UserName* ，其中 *Server* 是服务器计算机的名称，*UserName* 是有权访问服务器计算机的帐户的名称。

出现提示时，输入 *UserName* 的密码。

此命令成功后，可以使用 **-QR** 或 **连接到远程存根**发现 (在服务器计算机上运行的 KD 连接服务器) 。

激活智能客户端（内核模式）

项目 • 2024/03/01

激活 KD 连接服务器后，可以在另一台计算机上创建智能客户端并开始调试会话。

可通过两种方法启动智能客户端：使用内核协议 **kdsrv** 启动 KD 或 WinDbg，或使用 WinDbg 图形界面。

您需要指定 KD 连接服务器使用的远程传输协议。 还可以指定 KD 连接服务器与目标计算机之间实际内核连接的协议，也可以使用默认值。

用于启动智能客户端的常规语法取决于使用的协议。 存在以下选项：

控制台

```
Debugger -k
kdsrv:server=@{npipe:server=Server,pipe=PipeName[,password=Password]},trans=
@{ConnectType} [Options]

Debugger -k kdsrv:server=@{tcp:server=Server,port=Socket[,password=Password]
[,ipversion=6]},trans=@{ConnectType} [Options]

Debugger -k kdsrv:server=@{tcp:clicon=Server,port=Socket[,password=Password]
[,ipversion=6]},trans=@{ConnectType} [Options]

Debugger -k
kdsrv:server=@{com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=P
assword]},trans=@{ConnectType} [Options]

Debugger -k kdsrv:server=@{spipe:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,pipe=PipeName[,password=Password
]},trans=@{ConnectType} [Options]

Debugger -k kdsrv:server=@{ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,port=Socket[,password=Password]}
,trans=@{ConnectType} [Options]

Debugger -k kdsrv:server=@{ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},clicon=Server,port=Socket[,password=Password]}
,trans=@{ConnectType} [Options]
```

若要使用图形界面连接到 KD 连接服务器，WinDbg 必须处于休眠模式，即必须已在未使用任何命令行参数的情况下启动，或者必须已结束上一个调试会话。 选择**文件 |连接到远程存根**菜单命令。 出现“**连接到存根服务器**”对话框时，在**连接字符串**文本框中输入以下字符串之一：

dbgcmd

```
npipe:server=Server,pipe=PipeName[,password=Password]

tcp:server=Server,port=Socket[,password=Password][,ipversion=6]

tcp:clicon=Server,port=Socket[,password=Password][,ipversion=6]

com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=Password]

spipe:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,pipe=PipeName[,password=Password
]

ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},server=Server,port=Socket[,password=Password]

ssl:proto=Protocol,
{certuser=Cert|machuser=Cert},clicon=Server,port=Socket[,password=Password]
```

或者，可以使用“浏览”按钮找到活动 KD 连接服务器。

上述命令中的参数具有以下可能的值：

Debugger
这可以是 KD 或 WinDbg。

服务器
这是创建 KD 连接服务器的计算机的网络名称或 IP 地址。开头的两个反斜杠 (\) 在命令行是可选的，但在 WinDbg 对话框中是不允许的。

pipe=*PipeName*
如果使用 NPIPE 或 SPIPE 协议，*PipeName* 是创建 KD 连接服务器时赋予管道的名称。

如果未使用有权访问服务器计算机的帐户登录到客户端计算机，则必须提供用户名和密码。 在客户端计算机的命令提示符窗口中输入以下命令。

net use ***Server*\ipc$ /user:***UserName*

其中 *Server* 是服务器计算机的名称，*UserName* 是有权访问服务器计算机的帐户的名称。

出现提示时，输入 *UserName* 的密码。

此命令成功后，可以使用 **-kdsrv** 或使用 WinDbg 图形界面激活智能客户端。

port=*Socket*
如果使用 TCP 或 SSL 协议，*Socket* 是创建 KD 连接服务器时使用的同一套接字端口号。

clicon
指定 KD 连接服务器将尝试通过反向连接连接到智能客户端。 当且仅当服务器使用的是 **clicon** 时，客户端才必须使用 **clicon**。 在大多数情况下，在使用反向连接时，会在 KD 连接服务器之前启动智能客户端。

port=*COMPort*
如果使用 COM 协议，*COMPort* 指定要使用的 COM 端口。 前缀“COM”是可选的 -- 例如，“com2”和“2”都可接受。

baud=*BaudRate*
如果使用 COM 协议，*BaudRate* 应与 KD 连接服务器时选择的波特率匹配。

channel=*COMChannel*
如果使用 COM 协议，*COMChannel* 应与创建KD 连接服务器时选择的通道号匹配。

proto=*Protocol*
如果使用 SSL 或 SPIPE 协议，*Protocol* 应与创建 KD 连接服务器时使用的安全协议匹配。

Cert
如果使用 SSL 或 SPIPE 协议，则应使用创建 KD 连接服务器时使用的相同 **certuser**=*Cert* 或 **machuser**=*Cert* 参数。

password=*Password*
如果在创建 KD 连接服务器时使用了密码，则必须提供 *Password*，以便创建智能客户端。 它必须与原始密码匹配。 密码是区分大小写的。 如果提供了错误的密码，错误消息将指定“Error 0x80004005”。

ipversion=6
（仅适用于 Windows 调试工具 6.6.07 及更低版本）使用 TCP 连接到 Internet 时，强制调试器使用 IP 版本 6，而不是版本 4。 在 Windows Vista 和更高版本中，调试器会尝试自动默认为 IP 版本 6，因此不需要此选项。

trans=@{*ConnectType*}
告知调试器如何连接到目标。 允许以下内核连接协议：

```
dbgcmd

com:port=ComPort,baud=BaudRate
1394:channel=1394Channel[,symlink=1394Protocol]
usb2:targetname=String
com:pipe,port=\\VMHost\pipe\PipeName[,resets=0][,reconnect]
com:modem
```

有关这些协议的信息，请参阅调试设置。 您可以省略这些协议的任何参数。例如，您可以说 **trans=@{com:}**，调试器将默认使用运行 KdSrv 的计算机上的环境变量指定的值。

选项
可以在此处放置任何其他命令行参数。 有关完整列表，请参阅命令行选项。

由于 KD 连接服务器只是充当智能客户端的网关，因此，附加 *选项* 将与在运行 KdSrv 的计算机上启动内核调试器使用的选项相同。 例外情况是指定路径或文件名的任何选项都将作为运行智能客户端的计算机上的路径。

KD 连接服务器示例

项目 • 2023/06/15

假设要调试的计算机所在的站点上不存在调试技术人员。 调试技术人员要求此站点上的某人使用调试电缆将此目标计算机连接到其他计算机。

让另一台计算机位于 IP 地址 127.0.0.42。 调试电缆将此计算机上的 COM1 连接到目标计算机上已启用调试的端口。 KD 连接服务器是使用以下命令启动的：

控制台

```
E:\Debugging Tools for Windows> kdsrv -t tcp:port=1027
```

然后在另一个位置，技术人员使用以下命令启动 WinDbg 作为智能客户端：

控制台

```
G:\Debugging Tools> windbg -k
kdsrv:server=@{tcp:server=127.0.0.42,port=1027},trans=@{com:port=com1,baud=5
7600} -y SymbolPath
```

符号路径将相对于运行智能客户端的计算机。

以下是另一个示例。 在这种情况下，选择 NPIPE 协议，并使用 KD 而不是 WinDbg。 第一个用户选择管道名称。 这可以是任何字母数字字符串 -- 在本示例中为“KernelPipe”。 第一个用户打开提升的命令提示符窗口 (以管理员身份运行) ，并通过输入以下命令启动调试服务器：

控制台

```
E:\Debugging Tools for Windows> set _NT_DEBUG_PORT=com1
E:\Debugging Tools for Windows> kdsrv -t npipe:pipe=KernelPipe
```

技术人员使用无权访问服务器计算机的帐户登录到客户端计算机。 但技术人员知道有权访问服务器计算机的帐户的用户名和密码。 该帐户的用户名为 Contoso。 技术人员输入以下命令：

控制台

```
net use \\BOX17\ipc$ /user:Contoso
```

出现提示时，技术人员输入 Contoso 帐户的密码。

技术人员不确定命名管道使用的名称，因此他们查询 KD 连接服务器的 127.0.0.42：

控制台

```
G:\Debugging Tools> cdb -QR 127.0.0.42
Servers on 127.0.0.42:
Debugger Server - npipe:Pipe=MainPipe
Remote Process Server - npipe:Pipe=AnotherPipe
Remote Kernel Debugger Server - npipe:Pipe=KernelPipe
```

显示三个管道。 但是，只有一个是 KD 连接服务器 -- 其他是调试服务器和用户模式进程服务器。 技术人员输入以下命令可以启动智能客户端：

控制台

```
G:\Debugging Tools> kd -k
kdsrv:server=@{npipe:server=127.0.0.42,pipe=KernelPipe},trans=@{com:baud=576
00} -y SymbolPath
```

请注意，尽管指定了波特率，但端口不是。 这会导致调试器默认为运行 KdSrv 的计算机上由 _NT_DEBUG_PORT 指定的端口。

控制 KD 连接服务器会话

项目 • 2023/10/26

启动远程会话后，可以使用智能客户端，就像从运行 KD 连接服务器的计算机调试目标计算机一样。 所有命令的行为都与在这种情况下相同，只不过路径相对于智能客户端的计算机。

将 WinDbg 用作智能客户端

将 WinDbg 作为 KD 连接服务器的智能客户端启动后，可以使用 **调试 |停止调试** 命令以结束调试会话。 此时，WinDbg 将进入休眠模式，并且不再连接到 KD 连接服务器。 所有后续调试都将在运行 WinDbg 的计算机上完成。 无法使用文件 | 重新附加到 KD 连接服务 **内核调试** - 这只能通过命令行完成。

结束会话

KD 或 WinDbg 可以正常退出或结束调试会话。 请参阅 在 WinDbg (经典) 中结束调试会话

以获取详细信息。 KD 连接服务器将保持运行状态，并且可以根据需要多次重用。 (它还可用于任意数量的同时调试会话。)

KD 连接服务器可以从任一计算机终止。 若要从智能客户端终止它，请使用 **.endpsrv (End Process Server)** 命令。 若要从运行 KD 连接服务器的计算机终止 KD 连接服务器，请使用任务管理器结束 kdsrv.exe 进程。

中继器

项目 • 2023/06/15

*中继器*是一种轻型代理服务器，它在计算机上运行，并在其他两台计算机之间中继数据。 中继器不会以任何方式处理数据。 另外两台计算机几乎没注意到中继器：从他们的角度来看，他们似乎直接相连。

在这两台其他计算机上运行的进程称为 *服务器* 和 *客户端*。 从中继器的角度来看，它们之间没有任何根本区别，只不过在大多数情况下，首先启动服务器，然后启动中继器，最后启动客户端。

Windows 调试工具包包含一个名为 DbEngPrx (dbengprx.exe) 的中继器。

本节包括：

激活中继器

使用中继器

中继器示例

激活中继器

项目 • 2024/03/08

若要激活中继器连接，通常先启动服务器，然后启动中继器，之后再启动客户端。

也可以先启动中继器，然后再启动服务器。 但是，除非使用 **clicon** 参数建立反向连接，否则客户端必须始终最后启动。

步骤 1：启动服务器

服务器可以是调试服务器、进程服务器或 KD 连接服务器。 您可以像平常一样启动此操作，但传输协议设置将用于连接到中继器，而不是客户端。 有关详细信息，请参阅**激活调试服务器**、**激活进程服务器**或**激活 KD 连接 服务器**。

如果在创建服务器时使用密码，则在连接客户端时需要此密码，但在创建中继器时不需要。

如果使用**隐藏** 参数，服务器将照常隐藏。 中继器本身始终处于隐藏状态。

步骤 2：启动中继器

Windows 调试工具中包含的中继器名为 DbEngPrx (dbengprx.exe)。

DbEngPrx 支持以下传输协议：命名管道 (NPIPE)、TCP 和 COM 端口。

如果客户端和服务器使用安全套接字层 (SSL) 协议，则应为中继器使用 TCP 协议。 如果客户端和服务器使用安全管道 (SPIPE) 协议，则应为中继器使用 NPIPE 协议。 中继器将传递它接收的任何数据，它不解释、加密或解密任何信息。 所有加密和解密都将由客户端和服务器完成。

DbEnPrx 命令行的语法如下：

dbengprx [**-p**] **-c***ClientTransport*-**s***ServerTransport*

上述命令中的参数具有以下可能的值：

-p
即使删除了与 DbEngPrx 的所有连接，DbEngPrx 也会继续存在。

ClientTransport
指定要用于连接到服务器的协议设置。 协议应与创建服务器时使用的协议相匹配。 协议语法如下所示：

dbgcmd

```
npipe:server=Server,pipe=PipeName[,password=Password]
tcp:server=Server,port=Socket[,password=Password][,ipversion=6]
tcp:clicon=Server,port=Socket[,password=Password][,ipversion=6]
com:port=COMPort,baud=BaudRate,channel=COMChannel[,password=Password]
```

这些协议参数具有以下含义：

服务器
这是在其中创建服务器的计算机的网络名称或 IP 地址。 两个初始反斜杠 (\) 是可选的。

pipe=*PipeName*
如果使用 NPIPE 或 SPIPE 协议，*PipeName* 是创建服务器时赋予管道的名称。

port=*Socket*
如果使用 TCP 或 SSL 协议，*Socket* 是创建服务器时使用的同一套接字端口号。

clicon
指定服务器将尝试通过反向连接连接到中继器。 当且仅当服务器使用的是 **clicon** 时，*ClientTransport* 才必须使用 **clicon**。 在大多数情况下，当使用反向连接时，中继器会在服务器之前启动。

port=*COMPort*
如果使用 COM 协议，*COMPort* 指定要使用的 COM 端口。 前缀“COM”是可选的 -- 例如，“com2”和“2”都可接受。

baud=*BaudRate*
如果使用 COM 协议，*BaudRate* 应与创建服务器时选择的波特率匹配。

channel=*COMChannel*
如果使用 COM 协议，*COMChannel* 应与创建服务器时选择的通道号匹配。

password=*Password*
如果在创建服务器时使用了密码，则必须提供 *Password*，以便创建调试客户端。 它必须与原始密码匹配。 密码是区分大小写的。 如果提供了错误的密码，错误消息将指定“Error 0x80004005”。

ipversion=6
（仅适用于 Windows 调试工具 6.6.07 及更低版本）使用 TCP 连接到 Internet 时，强制调试器使用 IP 版本 6，而不是版本 4。 在 Windows Vista 和更高版本中，调试器会尝试自动默认为 IP 版本 6，因此不需要此选项。

ServerTransport
指定客户端连接到中继器时将使用的协议设置。 可能的协议语法包括：

```
dbgcmd
```

```
npipe:pipe=PipeName[,hidden][,password=Password][,IcfEnable]
tcp:port=Socket[,hidden][,password=Password][,IcfEnable]
tcp:port=Socket,clicon=Client[,password=Password]
com:port=COMPort,baud=BaudRate,channel=COMChannel[,hidden]
[,password=Password]
```

这些协议参数具有以下含义：

pipe=*PipeName*
使用 NPIPE 或 SPIPE 协议时，*PipeName* 是将用作管道名称的字符串。 每个管道名称应标识唯一的中继器。 如果尝试重用管道名称，将收到一条错误消息。 *PipeName* 不得包含空格或引号。 *PipeName* 可以包含数字 **printf** 样式的格式代码，如 **%x** 或 **%d**。 中继器会将其替换为 DbEngPrx 的进程 ID。 第二个此类代码将被替换为 DbEngPrx 的线程 ID。

port=*Socket*
使用 TCP 或 SSL 协议时，*Socket* 是套接字端口号。

还可以指定用冒号分隔的端口范围。 DbEngPrx 将检查此范围内的每个端口，以了解是否可用。 如果发现可用端口且未出错，则将创建中继器。 智能客户端必须指定要用于连接到中继器的实际端口。 若要确定实际端口，请搜索中继器；显示此中继器时，端口后跟两个用冒号分隔的数字。 第一个数字将是使用的实际端口；可以忽略第二个数字。 例如，如果将端口指定为 **port=51:60**，并且实际上使用了端口 53，则搜索结果将显示“port=53:60”。 （如果使用 **clicon** 参数建立反向连接，客户端可以采用这种方式指定端口范围，而中继器必须指定使用的实际端口。）

clicon=*Client*
使用 TCP 或 SSL 协议并指定 **clicon** 参数时，将打开*反向连接*。 这意味着，中继器将尝试连接到客户端，而不是让客户端启动联系。 如果防火墙阻止了正常方向的连接，这将非常有用。 *Client* 指定智能客户端所在或将创建客户端的计算机的网络名称或 IP 地址。 两个初始反斜杠 (\) 是可选的。

由于中继器正在查找特定客户端，因此如果使用此方法，则无法将多个客户端连接到中继器。 如果连接被拒绝或已断开，则必须重启中继器。

在使用 **clicon** 时，最好在创建服务器之前启动中继器，但也允许遵循正常顺序（中继器在客户端之前）。

port=*COMPort*
使用 COM 协议时，*COMPort* 指定要使用的 COM 端口。 前缀“COM”是可选的 -- 例如，“com2”和“2”都可接受。 不能在 *ClientTransport* 和 ServerTransport 中使用同一 COM 端口。

baud=*BaudRate*
使用 COM 协议时，*BaudRate* 指定连接运行时的波特率。 允许使用硬件支持的任何波特率。 如果在 *ClientTransport* 和 *ServerTransport* 中都使用 COM 协议，则可以指定不同的波特率，但较慢的速率自然会限制客户端和服务器之间的通信速度。

channel=*COMChannel*
如果使用 COM 协议，*COMChannel* 指定要用于与客户端通信的 COM 通道。 这可以是 0 和 254 之间（含）的任何值。 可以使用不同的通道号将单个 COM 端口用于多个连接。（这不同于将 COM 端口用于调试电缆，在那种情况下，不能在 COM 端口中使用通道。）

hidden
当另一个调试器显示所有活动服务器时，阻止服务器显示。

password=*Password*
要求客户端提供指定的密码，以便连接到调试会话。 *密码*可以是任何字母数字字符串。

IcfEnable
促使调试器在 Internet 连接防火墙处于活动状态时启用 TCP 或命名管道通信所需的端口连接。 默认情况下，Internet 连接防火墙会禁用这些协议使用的端口。 将 **IcfEnable** 与 TCP 连接配合使用时，调试器会导致 Windows 打开 *Socket* 参数指定的端口。 将 **IcfEnable** 与命名管道连接配合使用时，调试器会促使 Windows 打开用于命名管道的端口（端口 139 和 445）。 连接终止后，调试器不会关闭这些端口。

步骤 3：启动客户端

客户端应是调试客户端或智能客户端，与服务器类型相对应。 有关详细信息，请参阅**激活调试客户端**、**激活智能客户端**或**激活智能客户端（内核模式）**。

如果服务器拒绝连接（例如，如果提供的密码有误），则中继器和客户端都将关闭。 必须重启它们才能重新建立与服务器的联系。

使用中继器

项目 • 2023/10/26

中继器连接遵循非常简单的规则：

- 服务器和客户端打算彼此之间的任何通信都通过中继器而不进行任何更改。
- 服务器对传输连接执行的任何操作都会影响中继器（，仅间接影响客户端）。
- 客户端对传输连接执行的任何操作都会影响中继器（，并且仅间接影响服务器）。

这意味着，任何调试命令、调试器输出、控制键和文件访问都将完全像直接连接客户端和服务器一样发生。 中继器对所有这些命令不可见。

终止连接本身的操作将影响中继器。 例如，如果从客户端发出 qq (Quit) 命令，服务器将关闭并将关闭信号发送到传输。 这将导致中继器退出（，除非它以 -p 选项）启动。 再举一例， .clients (List Debugging Clients) 命令将列出客户端的计算机名称，但它将显示用于将服务器与中继器连接的连接协议。

如果服务器关闭，则中继器将自动退出（，除非它是使用 -p 选项）启动的。 当中继器关闭时，这将导致调试客户端退出，尽管智能客户端不会退出。 如果出于某种原因需要直接终止中继器，可以使用任务管理器或 kill.exe 工具。

中继器示例

项目 • 2023/06/15

假设你有三台计算机，即 \\BOXA、\\BOXB 和 \\BOXC，并且你希望将它们分别用作服务器、中继器和客户端。

可以通过以下方式在 \\BOXA 上启动调试服务器，使用进程 122 作为目标：

控制台

```
E:\Debugging Tools for Windows> cdb -server tcp:port=1025,password=wrought -
p 122
```

然后，可以在 \\BOXB 上启动中继器，如下所示：

控制台

```
C:\Misc> dbengprx -c tcp:server=BOXA,port=1025 -s npipe:pipe=MyPipe
```

最后，按以下方式在 \\BOXC 上启动调试客户端：

控制台

```
G:\Debugging Tools> windbg -remote
npipe:server=BOXB,pipe=MyPipe,password=wrought
```

以下是另一个示例。 符号位于远程位置 127.0.0.30。 因此，你决定在目标为 127.0.0.10 的计算机上使用进程服务器。 将中继器置于 127.0.0.20。

你还决定使用反向连接。 因此，首先在 127.0.0.30 上启动客户端：

控制台

```
G:\Debugging Tools> windbg -premote tcp:clicon=127.0.0.20,port=1033
notepad.exe
```

然后在 127.0.0.20 上启动中继器：

控制台

```
C:\Misc> dbengprx -c tcp:clicon=127.0.0.10,port=1025 -s
tcp:port=1033,clicon=127.0.0.10
```

最后，在 127.0.0.10 上启动进程服务器：

控制台

```
E:\Debugging Tools for Windows> dbgsrv -t tcp:port=1025,clicon=127.0.0.20
```

有关使用中继器的更复杂示例，请参阅 两个防火墙。

高级远程调试方案

项目 • 2024/03/20

本节包括：

在多台计算机上调试目标

中间的符号

两个防火墙

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

在多台计算机上调试目标

项目 • 2023/10/26

调试器可以同时调试多个转储文件或实时用户模式应用程序。 有关详细信息，请参阅调试多个目标 。

即使进程位于不同的系统上，也可以调试多个实时目标。 只需在每个系统上启动一个进程服务器，然后将 **-premote** 参数与 **.attach (Attach to Process)** 或 **.create (Create Process)** 配合使用来标识正确的进程服务器。

*当前*或*活动*系统是当前正在调试的系统。 如果在未指定 **-premote** 参数的情况下再次使用 **.attach** 或 **.create** 命令，调试器将附加到当前系统上的进程或创建进程。

有关如何控制此类调试会话的详细信息，请参阅 调试多个目标。

中间的符号

项目 • 2023/06/15

在此方案中，你有三台计算机。 第一个具有目标应用程序，第二个具有符号，第三个具有技术人员。

由于智能客户端在各方面的行为都类似于常规调试器，因此它可以同时用作调试服务器。 这样，就可以将三台计算机与中间的智能客户端链接在一起。

首先，在计算机 \\BOXA 上启动进程服务器：

控制台

```
dbgsrv -t npipe:pipe=FarPipe
```

名为 \\BOXB 的中间计算机使用 **-premote** 和 **-server** 参数启动调试器。 假设目标应用程序的 PID 为 400，符号路径为 G：\MySymbols：

控制台

```
cdb -server npipe:pipe=NearPipe -premote npipe:server=BOXA,pipe=FarPipe -v -
y g:\mysymbols -p 400
```

然后，可以按如下所示启动第三台计算机上的调试客户端：

控制台

```
windbg -remote npipe:server=BOXB,pipe=NearPipe
```

然后，第三台计算机用于控制调试，而第二台计算机是完成实际处理并访问符号的位置。

两个防火墙

项目 • 2023/06/15

在此方案中，需要在生成 A 中的计算机上执行内核调试。技术人员位于大楼 C 中，有权访问该处的符号。 但是，这两个建筑物的防火墙都不允许传入连接。

你需要在中性站点设置中继器-例如，建筑物 B。然后，可以将 A 向外连接到 B，将 C 向外连接到 B。

此方案涉及四台计算机：

- 位于建筑物 A 中的目标计算机。
- 位于建筑物 A 中的本地主机。此计算机将运行 KD 连接服务器。 它将通过调试电缆或 1394 电缆连接到目标计算机，并将向外连接到中继器。 让此计算机的 IP 地址为 127.0.10.10。
- 建筑物 B 中的计算机。这将运行中继器。 使其 IP 地址为 127.0.20.20。
- 技术人员所在的大楼 C 中的计算机。 此计算机将作为智能客户端运行 WinDbg。 使其 IP 地址为 127.0.30.30。

首先，确保目标计算机已配置为进行调试，并且已附加到本地主计算机。 在此示例中，使用 1394 电缆。

其次，在 127.0.20.20 启动中继器：

控制台

```
dbengprx -p -s tcp:port=9001 -c tcp:port=9000,clicon=127.0.10.10
```

第三，在建筑物 A 中的 127.0.10.10 上启动 KD 连接服务器，如下所示：

控制台

```
kdsrv -t tcp:port=9000,clicon=127.0.20.20,password=longjump
```

最后，在大楼 C.中启动智能客户端 127.0.30.30， (这实际上可以在建筑物 A.) 中启动服务器之前或之后完成

控制台

```
windbg -k
kdsrv:server=@{tcp:server=127.0.20.20,port=9001,password=longjump},trans=@{1
394:channel=9} -y SymbolPath
```

Five-Computer方案

如果假设符号位于建筑物 C 中的一台计算机上，但技术人员位于另一台计算机上，则此方案可能更加复杂。

假设 127.0.30.30 与之前一样具有符号，并且其本地名称为 \\BOXC。 可以使用上述相同命令启动智能客户端，但使用附加的 **-server** 参数。 由于没有人会使用此计算机，因此如果使用 KD 而不是 WinDbg，则需要更少的处理时间：

控制台

```
kd -server npipe:pipe=randomname -k
kdsrv:server=@{tcp:server=127.0.20.20,port=9001,password=longjump},trans=@{1
394:channel=9} -y SymbolPath
```

然后，技术人员可以在大楼中的其他位置启动调试客户端，如下所示：

控制台

```
windbg -remote npipe:server=\\BOXC,pipe=randomname
```

请注意，密码必须由链中的第一个非中继器提供， (\\BOXC) 上的智能客户端，而不是由链中的最后一个调试器提供。

使用 WinDbg 图形界面 (经典)

项目 • 2023/04/14

本部分介绍如何使用 WinDbg 调试器执行基本调试任务。

以下主题中提供了详细信息：

使用工作区 - WinDbg (经典)

使用工具栏和状态栏 - WinDbg (经典)

使用 WinDbg (经典) 进行实时Kernel-Mode调试

在 WinDbg (经典) 中结束调试会话

在 WinDbg 中输入调试器命令 (经典)

在 WinDbg 中使用命令浏览器窗口 (经典)

在 WinDbg (经典) 中设置断点

查看 WinDbg 中的调用堆栈 (经典)

WinDbg 中的程序集代码调试 (经典)

WinDbg (经典) 中的源代码调试

查看和编辑 WinDbg (经典) 中的内存

在 WinDbg (经典) 中查看和编辑全局变量

在 WinDbg (经典) 中查看和编辑局部变量

在 WinDbg (经典) 中查看和编辑寄存器

控制 WinDbg 中的进程和线程 (经典)

使用监视窗口 - WinDbg (经典)

使用暂存板 - WinDbg (经典)

使用 WinDbg（经典）的实时内核模式调试

项目 • 2024/10/31

可通过两种方式使用 WinDbg 启动实时内核模式调试会话。

WinDbg 菜单

当 WinDbg 处于休眠模式时，可以通过从“**文件**”菜单或按 Ctrl+K 选择**内核调试**会话来开始内核调试会话。 **出现**“**内核调试**”对话框时，单击相应的选项卡：**NET**、**COM**、**USB** 或**本地**。 每个选项卡指定不同的连接方法。

命令提示符

在命令提示符窗口中，启动 WinDbg 时，可以启动内核模式调试会话。 输入以下命令之一：

windbg [-y *SymbolPath*] -k net：port=*PortNumber*，*key*=*Key*[，target=*TargetIPAddress*|*TargetMachineName*]

windbg [-y *SymbolPath*] -k usb：targetname=*USBString*

windbg [-y *SymbolPath*] -k com：port=*ComPort*，*baud*=*BaudRate*

windbg [-y *SymbolPath*] -k com：ipport=*SerialTcpIpPort*，*port*=*SerialIPAddress*

windbg [-y *SymbolPath*] -k com：pipe，port=*VMHost*\pipe*PipeName*[，resets=0][，reconnect]

windbg [-y *SymbolPath*] -k com：*调制解调器*

windbg [-y *SymbolPath*] -kl

windbg [-y *SymbolPath*] -k

有关详细信息，请参阅 **WinDbg 命令行选项**。

环境变量

若要通过串行（COM 端口）进行调试，可以使用环境变量来指定连接设置。

使用以下变量指定串行连接。

set _NT_DEBUG_PORT = *ComPort*

set _NT_DEBUG_BAUD_RATE = *BaudRate*

有关详细信息，请参阅 内核模式环境变量。

参数

SymbolPath
符号文件所在的目录列表。 列表中的目录用分号分隔。 有关详细信息，请参阅 符号路径。

PortNumber
用于网络调试的端口号。 可以选择从 49152 到 65535 的任意数字。 有关详细信息，请参阅 手动设置网络连接。

钥匙
用于网络调试的加密密钥。 建议使用自动生成的密钥，该密钥由 bcdedit 在配置目标计算机时提供。 有关详细信息，请参阅 手动设置网络连接。

TargetIPAddress
目标计算机的 IPv4 地址。

指定 target= IP 地址时，这会导致调试器通过向目标发送特殊数据包来启动与指定目标计算机的连接，从而导致它尝试与该调试器连接。 调试器会每隔半秒重复向目标发送数据包，尝试连接。 如果连接成功，目标将删除任何现有连接，并仅与此调试器实例通信。 这样，就可以从现有调试连接中控制调试会话。

当目标配置有主机 IP 地址，并且调试器在具有配置的主机 IP 地址的计算机上运行时，无需指定 target= IP 地址参数。 当目标配置有主机 IP 地址时，它将每隔三秒将 OFFER 数据包发送到主机。 如果未指定 target= IP 地址，则 OFFER 数据包允许调试器连接到主机。

有关在目标上配置主机 IP 地址的详细信息，请参阅 “自动 设置 KDNET 网络内核调试”和 “手动设置 KDNET 网络内核调试”。

TargetMachineName
目标电脑的计算机名称。 若要使用计算机名称，网络上的 DNS 系统必须具有与目标电脑的 IP 地址关联的计算机名称。

USBString
USB 连接字符串。 这必须与使用 /targetname 启动选项指定的字符串匹配。 有关详细信息，请参阅设置 USB 3.0 xHCI-DBC 内核模式调试（KDUSB）。

ComPort
COM 端口的名称。 这可以采用格式“com2”或格式为“\\.\com2”，但不应只是数字。 有关详细信息，请参阅 手动设置串行连接。

BaudRate
波特率。 可以是 9600、19200、38400、57600 或 115200。

SerialTcpIpPort
模拟串行 COM 端口的目标 TCP IP 端口。

SerialIPAddress
模拟串行 COM 端口的目标 TCP IP 地址。

VMHost
调试虚拟机时， *VMHost* 指定运行虚拟机的物理计算机的名称。 如果虚拟机与内核调试器本身在同一台计算机上运行，请对 *VMHost* 使用单个句点（.）。 有关详细信息，请参阅 设置与虚拟机的连接。

PipeName
虚拟机为调试连接创建的管道的名称。

resets=0
指定在主机和目标同步时，可以向目标发送不限数量的重置数据包。 仅当调试某些类型的虚拟机时，才需要此参数。

重新
使调试器在发生读/写失败时自动断开连接并重新连接管道。 此外，如果在启动调试器时找不到命名管道，则重新连接参数将导致它等待显示此名称的管道。 仅当调试某些类型的虚拟机时，才需要此参数。

-kl
使调试器执行本地内核模式调试。 有关详细信息，请参阅 本地内核模式调试。

以前的版本参数

> **重要**
>
> 1394 传输可用于 Windows 10 版本 1607 及更低版本。 它在 Windows 的更高版本中不可用。 应将项目转换为其他传输，例如使用以太网的 KDNET。 有关该传输的详细信息，请参阅**自动设置 KDNET 网络内核调试**。

1394Channel
1394 通道编号。 有效通道号是介于 0 和 62 之间的任意整数，包括 0 和 62。

1394Channel 必须与目标计算机使用的数字匹配，但不依赖于适配器上选择的物理 1394 端口。

1394Protocol
要用于 1394 内核连接的连接协议。 这几乎始终可以省略，因为调试器将自动选择正确的协议。 如果想要手动设置此设置，并且目标计算机正在运行 Windows XP， *应将 1394Protocol* 设置为“channel”。 如果目标计算机运行的是 Windows Server 2003 或更高版本， *则应将 1394Protocol* 设置为“instance”。 如果省略，调试器将默认为适用于当前目标计算机的协议。 这只能通过命令行或环境变量指定，而不能通过 WinDbg 图形界面指定。

示例

以下批处理文件可用于通过 COM 端口连接设置和启动调试会话。

控制台

```
set _NT_SYMBOL_PATH=d:\mysymbols
set _NT_DEBUG_PORT=com1
set _NT_DEBUG_BAUD_RATE=115200
set _NT_DEBUG_LOG_FILE_OPEN=d:\debuggers\logfile1.log
windbg -k
```

以下命令行可用于启动 WinDbg，而无需任何环境变量。

windbg -y d：\mysymbols -k com：port=com2，baud=57600

windbg -y d：\mysymbols -k com：port=\.\com2，baud=115200

windbg -y d：\mysymbols -k net：port=50000，key=AutoGeneratedKey

windbg -y d：\mysymbols -k net：port=50000，key=AutoGeneratedKey，target=TargetIPAddress

另请参阅

WinDbg 命令行选项

内核模式环境变量

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

在 WinDbg 中结束调试会话 (经典)

项目 • 2023/11/15

退出 WinDbg

可以通过从“**文件**”菜单中选择“**退出**”或按 Alt+F4 来退出 WinDbg。

如果要执行用户模式调试，这些命令会关闭要调试的应用程序，除非在启动调试器时使用了 -pd 命令行选项。

如果要执行内核模式调试，则目标计算机将保持其当前状态。 这种情况使你能够使目标保持运行或冻结状态。 (如果使目标保持冻结状态，则来自内核调试器的任何将来连接都可以在离开它的位置恢复调试。)

在不退出的情况下结束 User-Mode 会话

若要结束用户模式调试会话，将调试器返回到休眠模式，并关闭目标应用程序，可以使用以下方法：

- 输入 .kill (Kill Process) 命令。
- 输入 q (Quit) 命令 (，除非使用 -pd 选项) 启动调试器。
- 从 “**调试** ”菜单中选择“停止 **调试** ”。
- 按 SHIFT+F5。
- 单击工具栏上的 “**停止调试** ”按钮。

若要结束用户模式调试会话，将调试器返回到休眠模式，并设置目标应用程序再次运行，可以使用以下方法：

- 从 Process) **命令输入** .detach (Detach 命令。 如果要调试多个目标，此命令将从当前目标分离，并继续与剩余目标的调试会话。
- 从 “**调试** ”菜单中选择“分离 **调试** 者”。 如果要调试多个目标，此命令将从所有当前目标中分离。
- 输入 qd (Quit and Detach) 命令。
- 如果已使用 -pd 选项启动调试器，请输入 q (Quit) 命令。

若要结束用户模式调试会话，请将调试器返回到休眠模式，但让目标应用程序保持调试状态，可以使用以下方法：

- 输入 **.abandon (放弃进程)** 命令。

有关重新附加到目标的信息，请参阅 重新附加到目标应用程序。

在不退出的情况下结束 Kernel-Mode 会话

若要结束内核模式调试会话，将调试器返回到休眠模式，并使目标计算机保持冻结状态，可以使用以下方法：

- 输入 **q (Quit)** 命令 (除非使用 **-pd** 选项启动调试器)
- 从 “**调试** ”菜单中选择“停止 **调试** ”。
- 按 SHIFT+F5。
- 单击工具栏上的“ **停止调试 (Shift+F5)** ”按钮。

WinDbg 会话结束时，系统会提示你保存当前会话的工作区，然后 WinDbg 将返回到休眠模式。 此时，可以使用所有起始选项。 也就是说，可以开始调试正在运行的进程、生成新进程、附加到目标计算机、打开故障转储或连接到远程调试会话。

在 WinDbg 中输入调试器命令 (经典)

项目 • 2023/11/15

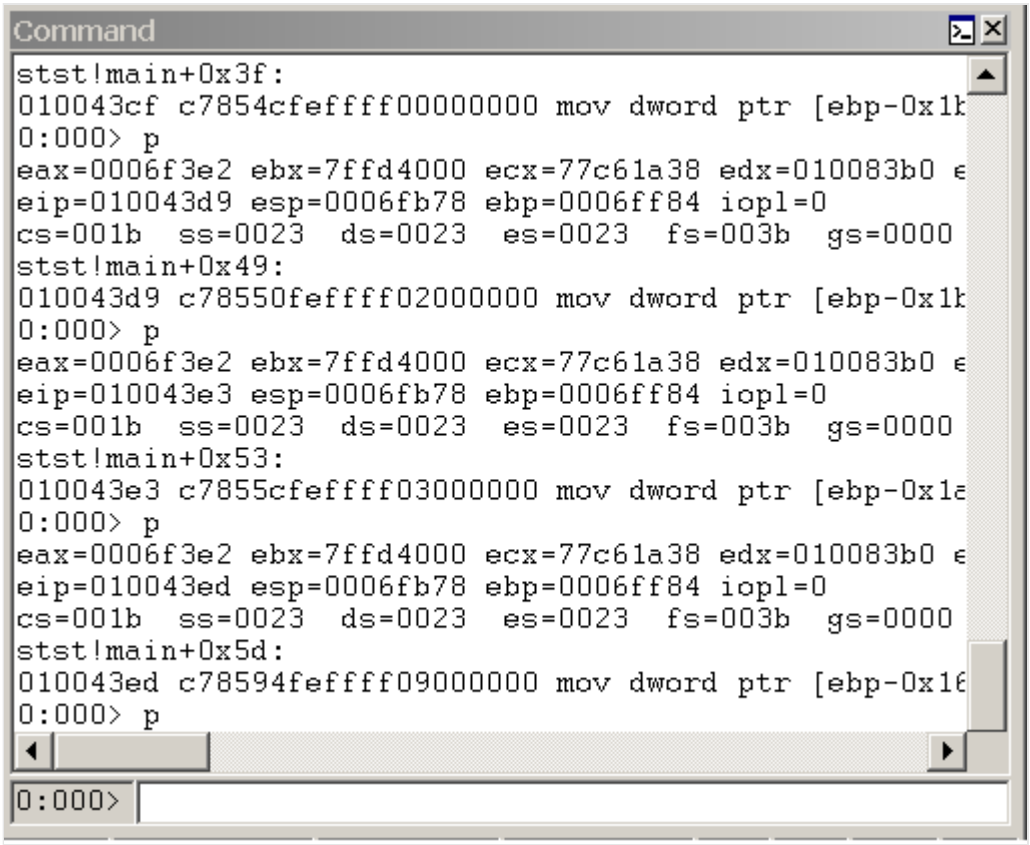
调试器命令窗口是 WinDbg 中的主要调试信息窗口。 可以输入调试器命令并在此窗口中查看命令输出。

此窗口在标题栏中显示“命令”。 但是，本文档始终将此窗口称为“调试器命令窗口”，以避免将其与用于发出 Microsoft MS-DOS 命令的命令提示符窗口混淆。

打开调试器命令窗口

若要打开“调试器命令”窗口，请从“**视图**”菜单中选择“**命令**”。 (还可以按 Alt+1 或选择工具栏上的 “**命令** ”按钮。Alt+SHIFT+1 关闭调试器命令窗口。)

以下屏幕截图显示了调试器命令窗口的示例。



使用调试器命令窗口

调试器命令窗口拆分为两个窗格。 在较小的窗格中键入命令 (命令输入窗格) 窗口底部，并在窗口顶部的大窗格中查看输出。

在命令条目窗格中，使用向上键和向下键滚动浏览命令历史记录。 出现命令时，可以对其进行编辑或按 Enter 运行命令。

“调试器命令”窗口包含一个包含其他命令的快捷菜单。 若要访问此菜单，请选择并按住(或右键单击) 窗口标题栏，或选择窗口右上角附近的图标。 以下列表介绍了一些菜单命令。

- **添加到命令输出** 会向命令输出添加注释，类似于 **编辑 |添加到命令输出** 命令。
- **清除命令输出** 会删除窗口中的所有文本。
- **选择文本颜色并重新着色选择...** 打开一个对话框，使你能够选择要在其中显示调试器命令窗口中所选文本的文本颜色。
- Word**换行**将打开和关闭自动换行状态。 此命令会影响整个窗口，而不仅仅是选择此状态后使用的命令。 由于许多命令和扩展会生成带格式的显示器，因此不建议使用自动换行。
- **标记当前位置** 在命令窗口中的当前光标位置设置标记。 标记的名称是光标右侧的行的内容。
- **转到标记** 会导致窗口滚动，以便包含所选标记的行位于窗口顶部。
- **始终浮动** 会导致窗口保持未停靠状态，即使它被拖到停靠位置。
- **使用框架移动** 会导致在移动 WinDbg 帧时移动窗口，即使窗口已取消停靠也是如此。

在 WinDbg 中使用命令浏览器窗口 (经典)
经典

项目 • 2023/11/15

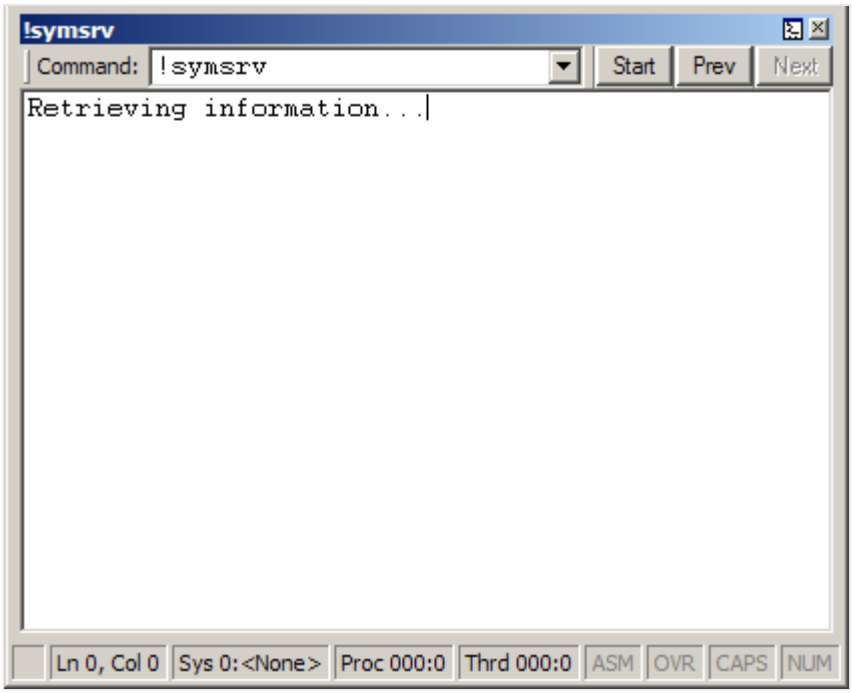
命令浏览器窗口显示并存储调试器命令的文本结果。 此窗口创建一个命令引用，使你无需重新输入命令即可查看特定命令的结果。 “命令浏览器”窗口还提供在存储的命令中的导航，因此与使用 “调试器命令”窗口相比，你可以更快地访问命令。

打开命令浏览器窗口

可以一次打开多个命令浏览器窗口。 若要打开命令浏览器窗口，请从“**视图**”菜单中选择“**命令浏览器**”。 (还可以按 Ctrl+N 或单击工具栏上的 “**命令浏览器** ”按钮。Alt+SHIFT+N 关闭命令浏览器窗口。)

还可以通过在常规的调试器命令窗口中输入 .browse (Display Command in Browser) 来打开命令浏览器窗口。

以下屏幕截图显示了命令浏览器窗口的示例。



使用命令浏览器窗口

在“命令浏览器”窗口中，可以执行以下操作：

- 若要输入命令，请在“命令”框中键入该 **命令** 。

- 若要查看以前输入的命令的结果，请使用“**开始**”、“**上一步**”和“**下一步**”按钮滚动命令列表，或者从“**命令**”菜单中选择上述 20 个命令之一。 若要查找不是上述 20 个命令之一的命令，请使用“ **下一步** ”按钮。

“命令浏览器”窗口有一个包含其他命令的快捷菜单。 若要访问菜单，请右键单击标题栏，或单击窗口右上角附近的图标。 以下列表描述了一些菜单命令。

- **Start**、 **Prev** 和 **Next** 分别将光标移动到命令历史记录的开头或上一个或下一个命令。
- “**添加到最近使用的命令**”将当前命令放入 WinDbg 窗口中“**视图**”菜单的“**最近使用命令**”菜单中。 最近的命令保存在工作区中。
- **工具栏** 可打开和关闭工具栏。
- **移动到新扩展坞** 会关闭命令浏览器窗口，并在新扩展坞中打开它。
- **始终浮动** 会导致窗口保持未停靠状态，即使它被拖动到停靠位置。
- **随帧移动** 会导致移动 WinDbg 框架时窗口移动，即使窗口已取消停靠也是如此。

在命令浏览器窗口中输入的命令由调试器引擎执行，而不是由 WinDbg 用户界面执行。 这意味着无法在命令浏览器窗口中输入 **用户界面命令（如 .cls ）** 。 如果用户界面是远程客户端，则服务器 (而不是客户端) 执行命令。

在命令浏览器窗口中输入的命令会同步执行，因此在完成之前不会显示输出。

命令浏览器窗口保存在 WinDbg 工作区中，但不保存命令历史记录。 工作区中仅保存每个命令浏览器窗口的当前命令。

在 WinDbg (经典) 中设置断点

项目 • 2023/04/14

可以使用 WinDbg 通过多种方式设置、查看和操作断点。

调试器命令窗口

可以通过在调试器命令窗口中输入命令来设置、查看和操作断点。 有关命令的列表，请参阅 控制断点的方法。

WinDbg 菜单

可以通过从“**编辑**”菜单中选择“断**点**”或按 Alt+F9 来打开“**断点**”对话框。 此对话框列出所有断点，您可以使用它禁用、启用或清除现有断点或设置新的断点。

代码窗口

“反汇编”窗口和“源”窗口突出显示设置了断点的行。 已启用的断点为红色，禁用的断点为黄色。

如果在反汇编窗口或源窗口中的特定行上设置光标，则可以按 F9 在该行设置断点。

在 WinDbg 中查看调用堆栈（经典）

项目 • 2024/03/08

调用堆栈是指向程序计数器当前位置的函数调用链。 调用堆栈上最上面的函数是当前函数，下一个函数是调用当前函数的函数，依此类推。 除非更改寄存器上下文，否则显示的调用堆栈基于当前程序计数器。 有关如何更改注册上下文的详细信息，请参阅更改上下文。

在 WinDbg 中，可以通过输入命令或使用“调用”窗口来查看调用堆栈。

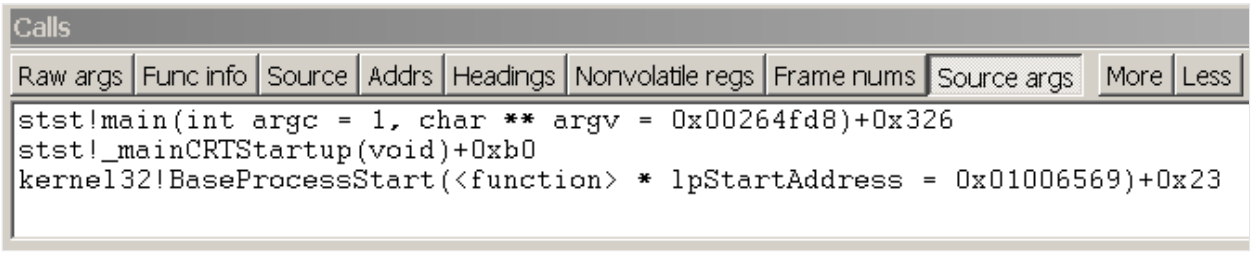
调试器命令窗口

可以通过在调试器命令窗口中输入其中一个 **k（显示堆栈回溯）** 命令来查看调用堆栈。

“调用”窗口

作为 **k** 命令的替代方法，可以在“调用”窗口中查看调用堆栈。 若要打开“调用”窗口，请从**视图**菜单中选择**调用堆栈**。

以下屏幕截图显示了“调用”窗口的示例。



“调用”窗口中的按钮可用于自定义调用堆栈的视图。 要在源窗口或反汇编窗口中移动到相应的调用位置，请双击调用堆栈的一行，或选择一行并按 ENTER 键。 此操作还会将本地上下文更改为所选堆栈帧。 有关运行到此点或从此点开始运行的详细信息，请参阅控制目标。

在用户模式下，堆栈跟踪基于当前线程的堆栈。 有关当前线程堆栈的详细信息，请参阅控制进程和线程。

在内核模式下，堆栈跟踪基于当前的寄存器上下文。 可以将寄存器上下文设置为与特定线程、上下文记录或陷阱帧匹配。 有关设置寄存器上下文的详细信息，请参阅寄存器上下文。

“调用”窗口具有一个工具栏，其中包含多个按钮，还有一个包含其他命令的快捷菜单。 若要访问此菜单，请选择并按住标题栏或选择窗口右上角附近的图标 (

- **原始参数**显示传递给函数的前三个参数。 在基于 x86 的处理器上，此显示器包含传递给函数的前三个参数（“Args to Child”）。
- **函数信息**显示帧指针省略 (FPO) 数据以及有关函数的其他内部信息。 此命令仅在基于 x86 的处理器上可用。
- **源**在函数名称后显示源模块名称和行号（如果调试器具有此信息）。
- **Addrs** 显示各种与帧相关的地址。 在基于 x86 的处理器上，此显示包括堆栈帧（“ChildEBP”）和返回地址（“RetAddr”）的基指针。
- **帧编号**显示帧编号。 帧始终连续编号，从零开始。
- **参数类型**显示有关堆栈中函数预期和接收的参数的详细信息。
- **始终浮动**会使窗口保持未停靠状态，即使将其拖动到停靠位置也是如此。
- **随帧移动**会导致在移动 WinDbg 帧时移动窗口，即使取消停靠该窗口也是如此。

其他信息

有关寄存器上下文和本地上下文的详细信息，请参阅更改上下文。

WinDbg（经典）中的汇编代码调试

项目 • 2024/05/27

在 WinDbg 中，可以通过输入命令或使用“反汇编”窗口来查看汇编代码。

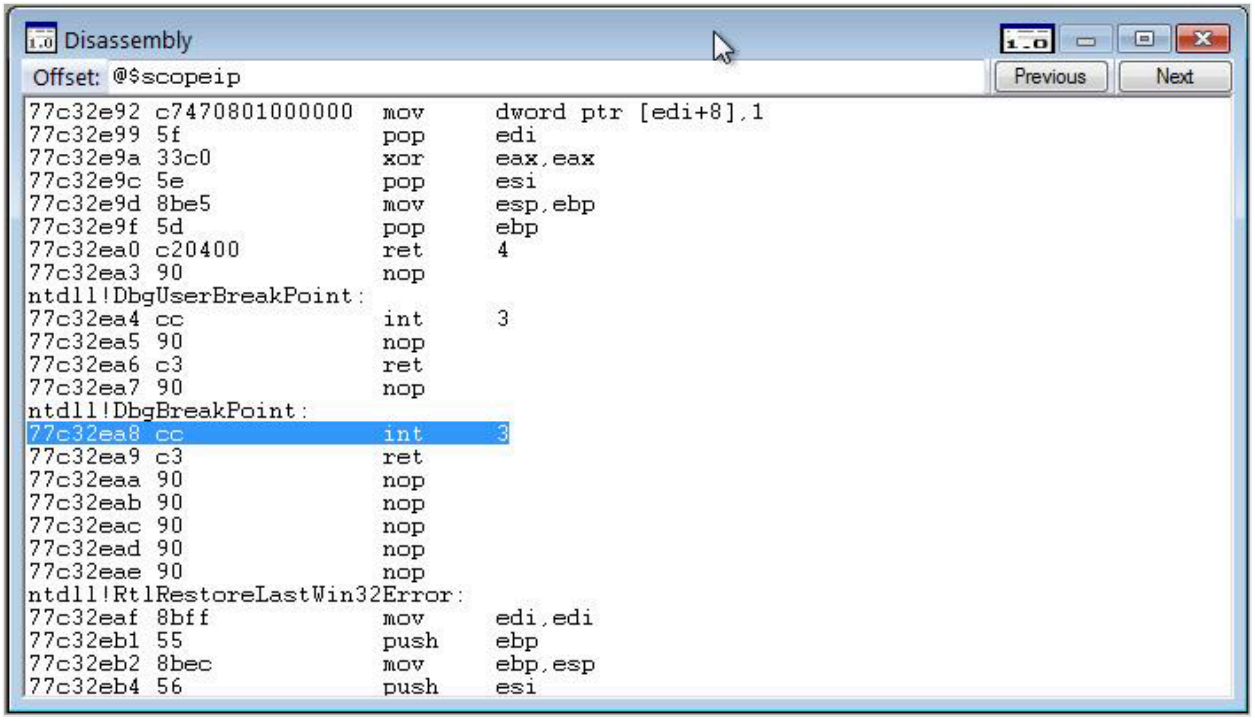
调试器命令窗口

在调试器命令窗口中输入 **u、ub、uu（反汇编）**命令之一即可查看汇编代码。

“反汇编”窗口

要打开或切换到拆卸窗口，请从“视图”菜单中选择“反汇编”。 （也可以按 ALT+7 或选择工具栏上的“反汇编”按钮。ALT+SHIFT+7 将关闭反汇编窗口。）

以下屏幕截图显示了“反汇编”窗口的示例。



调试器会获取一段内存，将其解释为二进制计算机指令，然后进行反汇编，以便生成计算机指令的汇编语言版本。 生成的代码将显示在“反汇编”窗口中。

在“反汇编”窗口中，可以执行以下操作：

- 要对不同部分的内存进行反汇编，请在“偏移量”框中键入要反汇编的内存的地址。（键入地址后可以按 Enter，但并非必须。）在输入地址之前，“反汇编”窗口会显示代码；这些代码可被忽略。

- 要查看内存的其他部分，请选择“上一个”或“下一个”按钮，或者按 PAGE UP 或 PAGE DOWN 键。 这些命令将分别显示前一段或后一段内存的反汇编代码。 按右箭头、左箭头、上箭头和下箭头键可在窗口中导航。 如果使用这些键离开页面，则会出现一个新的页面。

“反汇编”窗口具有一个工具栏，其中包含两个按钮，还有一个包含其他命令的快捷菜单。 要访问此菜单，请选择并按住标题栏或选择窗口右上角附近的图标。 下面列出了一些菜单命令。

- **转到当前地址**可打开“源”窗口，其中包含与“反汇编”窗口中选定行相对应的源文件，并突出显示该行。
- **在当前指令前反汇编**会使当前行位于“反汇编”窗口的中间。 此命令为默认选项。 如果清除此命令，则当前行将显示在“反汇编”窗口的顶部，这样可以节省时间，因为反向反汇编可能很耗时。
- **突出显示当前源代码行的指令**会让与当前源代码行相对应的所有指令都突出显示。 通常情况下，一个源行会对应多个汇编指令。 如果代码经过了优化，这些汇编指令可能不是连续的。 可以使用此命令来查找当前源行中的所有汇编指令。
- **显示每条指令的源行**会显示每个汇编指令对应的源行号。
- **显示每条指令的源文件**会显示与每条汇编指令相对应的源文件名。

另请参阅

有关汇编调试和相关命令的更多信息，以及汇编显示的完整说明，请参阅在汇编模式下调试。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

WinDbg (经典) 中的源代码调试

项目 • 2023/08/10

源路径

源路径指定 C 和 C++ 源文件所在的目录。 有关在调试器中查看源代码的详细信息，请参阅 源路径。

注意 如果连接到公司网络，则访问源文件的最有效方法是使用源服务器。 可以通过在源路径中使用 srv* 字符串来使用源服务器。 有关源服务器的详细信息，请参阅 使用源服务器。

若要控制 WinDbg 中的源路径，请执行下列操作之一：

- 从 “**文件** ”菜单中选择“源文件路径 ” 或按 Ctrl+P。
- 使用 **.srcpath (设置源路径)** 命令。 如果使用源服务器， **.srcfix (使用源服务器)** 稍微简单一些。
- 使用 **.lsrcpath (设置本地源路径)** 命令。 如果使用源服务器， **.lsrcfix (使用本地源服务器)** 稍微简单一些。
- 启动调试器时，请使用 **-srcpath** 或 **-lsrcpath** 命令行选项。 请参阅 **WinDbg Command-Line 选项**。
- 在启动调试器之前，请设置 _NT_SOURCE_PATH 环境变量。

打开和关闭源文件

若要直接打开或关闭源文件，请执行以下操作之一：

- 从 “**文件** ”菜单中选择“打开源文件 ”，或按 Ctrl+O。 还可以使用“**打开源文件**”按钮 (工具栏上) 。

  注意 使用菜单或工具栏按钮打开源文件时，该文件的路径会自动追加到源路径。
- 从“**文件**”菜单中选择“**关闭当前窗口**”。
- 选择“源”窗口一角的“ **关闭** ”按钮。
- 从 “**文件** ”菜单中选择“最近使用 **的文件** ”，打开最近在 WinDbg 中打开的四个源文件之一。

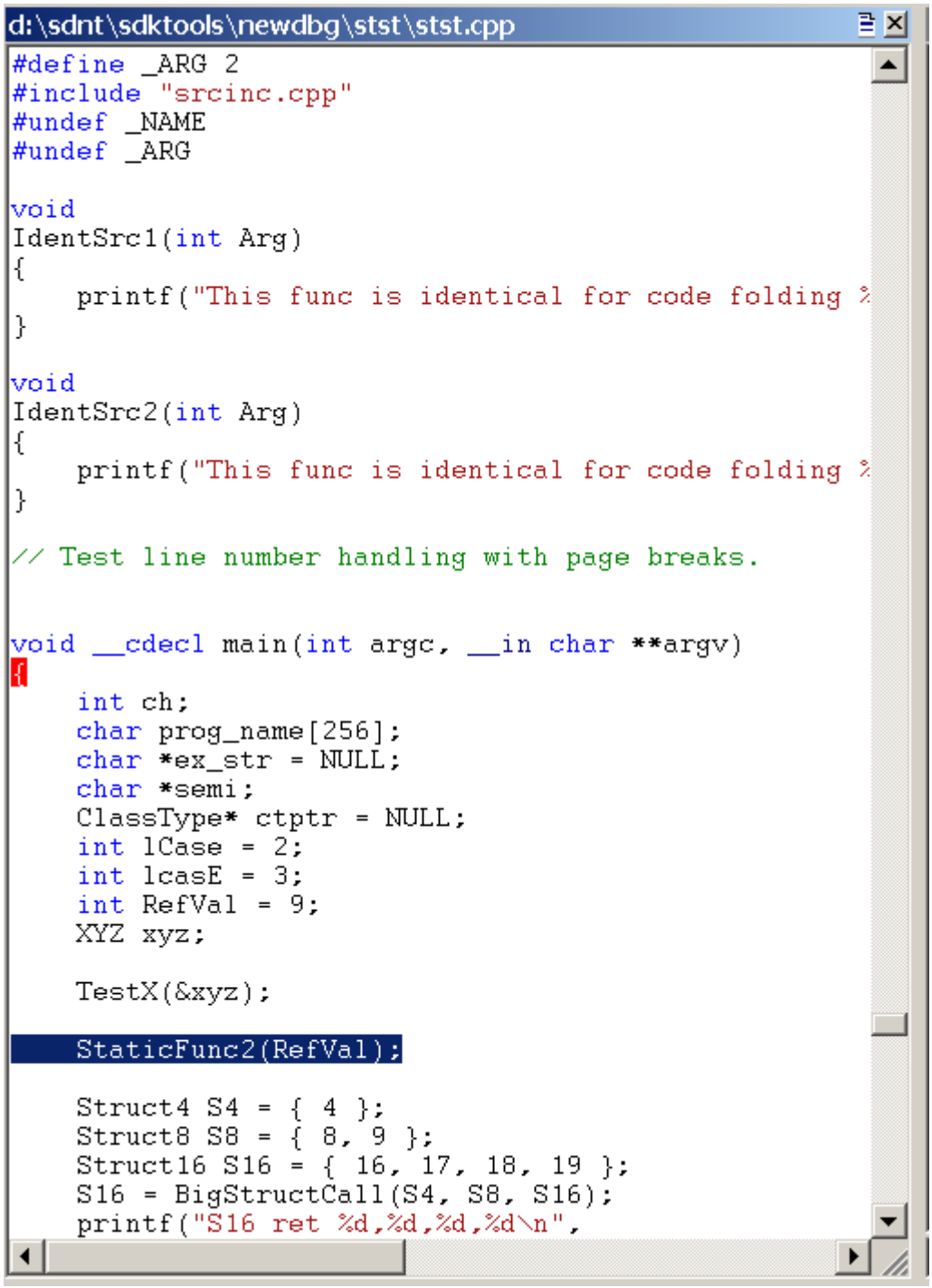
- 输入 .open (Open Source File) 命令。
- 输入 lsf (**加载或卸载源文件**) 命令。

在 WinDbg 中，“源”窗口显示已加载到调试器的源文件。

打开源窗口

调试器在加载新的源文件时会打开一个源窗口。 若要还原或切换到打开的“源”窗口，请转到“ **窗口** ”菜单，然后从菜单底部的窗口列表中选择。

以下屏幕截图显示了“源”窗口的示例。



每个源文件都驻留在其自己的“源”窗口中。 每个“源”窗口的标题是源文件的完整路径。

使用源窗口

每个“源”窗口显示一个源文件的文本。 无法在调试器中编辑源文件。 有关更改字体和选项卡设置的详细信息，请参阅 更改文本属性。

每个“源”窗口都有一个快捷菜单，其中包含其他命令。 若要访问菜单，请选择并按住 (或右键单击) 标题栏，或选择窗口右上角附近显示的图标 (。)。 以下列表描述了一些菜单命令：

- **将指令指针设置为当前行** 会将指令指针的值更改为对应于当前行的指令。 此命令等效于使用 **编辑 |设置“当前指令”** 命令或按 Ctrl+Shift+I。
- **编辑此文件** 将在文本编辑器中打开源文件。 编辑器由 WinDiff 编辑器注册表信息或 WINDBG_INVOKE_EDITOR环境变量的值确定。 例如，假设 WINDBG_INVOKE_EDITOR的值如下所示。

  控制台

  ```
  c:\my\path\myeditor.exe -file %f -line %l
  ```

  在这种情况下，Myeditor.exe 将打开当前源文件的从 1 开始的行号。 %l 选项指示应将行号读取为从 1 开始，而 %f 指示应使用当前源文件。 其他替换可能性包括 %L（指示行号从零开始）和 %p（也可以指示应使用当前源文件）。
- **Evaluate selection** 使用 C++ 表达式计算器计算当前选定的文本。 结果显示在 “调试器命令”窗口中。 如果所选文本包含多行，则会导致语法错误。 此命令等效于使用 **编辑 |“计算选择”** 命令，按 CTRL+SHIFT+V，或使用？？ (**将所选文本作为其参数的计算 C++ 表达式**) 命令。
- **显示所选类型** 显示所选对象的数据类型。 此显示将显示在“调试器命令”窗口中。 如果所选文本包含多个对象，则可能显示语法错误或其他不规则结果。 此命令等效于使用 **编辑 |显示“所选类型”** 命令或按 Ctrl+Shift+Y。
- **打开选择的内存窗口** 将打开一个新的停靠内存窗口，该窗口显示从所选表达式的地址开始的内存。
- **将所选内容添加到“监视”窗口** 会将所选源令牌追加到“监视”窗口。
- **在当前行处反汇编** 会导致与当前行对应的指令显示在 反汇编窗口中。 所选行在“源”窗口和“反汇编”窗口中突出显示，但此命令仅影响显示，指令指针不会更改。 如果在选中此命令时关闭了反汇编窗口，则会打开该窗口。

- **选择源语言** 将显示编程语言列表。 选择用于生成源文件的编程语言，然后选择“ **确定** ”为当前“源”窗口启用基本语法突出显示。 选择“**无**>”<可禁用当前“源”窗口的语法突出显示。

源窗口颜色和悬停评估

如果调试器识别源文件文件扩展名，则“源”窗口以颜色显示某些语法元素。 若要关闭或更改颜色，请执行以下操作：

- 若要在单个窗口中关闭语法颜色，请打开“源”窗口的快捷菜单，选择“**选择源语言**”，然后选择“**无**>”。 <
- 若要关闭所有“源”窗口的语法颜色，请从“**视图**”菜单中选择“**选项**”。 然后清除“**分析源语言**检查框。
- 若要更改语法颜色，请从“**视图**”菜单中选择“**选项**”。 然后，在 “**颜色** ”区域中，选择语法元素，然后选择“ **更改** ”按钮以更改颜色。
- 用于突出显示的解析方法由与源文件的文件扩展名关联的编程语言确定。 若要更改与特定文件扩展名关联的编程语言，请使用“ **源语言的文件扩展名**”**对话框**。 若要打开此对话框，请从“**视图**”菜单中选择“**源语言文件扩展名**”。

突出显示表示当前程序计数器的行。 设置断点的行也会突出显示。

如果选择一个“源”窗口，然后使用鼠标将鼠标悬停在该窗口中的某个符号上，则将计算该符号。 评估与 **dt (显示类型)** 命令生成的评估相同。 若要停用此评估，请从“**视图**”菜单中选择“**选项**”。 然后清除“**悬停检查评估**”框。

其他信息

有关源调试和相关命令的详细信息，请参阅 在源模式下调试。

在 WinDbg (经典) 中查看和编辑内存

项目 • 2023/11/15

在 WinDbg 中，可以通过输入命令或使用“内存”窗口来查看和编辑内存。

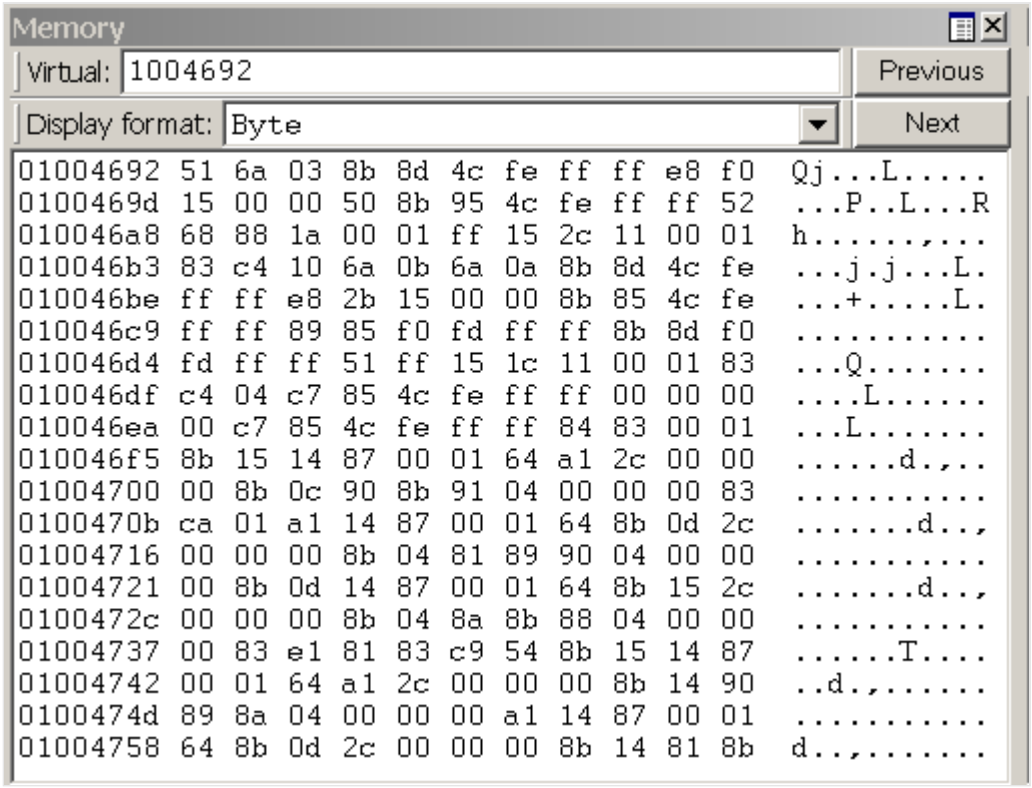
调试器命令窗口

可以通过在“调试器命令”窗口中输入其中一个 **“显示内存”** 命令来查看内存。 可以通过在“调试器命令”窗口中输入 **“输入值”** 命令之一来编辑内存。 有关详细信息，请参阅 按虚拟地址访问内存 和 按物理地址访问内存。

打开“内存”窗口

若要打开“内存”窗口，请从“**视图**”菜单中选择“**内存**”。 (还可以按 Alt+5 或选择 **“内存”** 按钮 (

以下屏幕截图显示了“内存”窗口的示例。



使用内存窗口

“内存”窗口以多个列显示数据。 窗口左侧的列显示每行的起始地址。 其余列从左到右显示请求的信息。 如果在“**显示格式**”菜单中选择“**字节**”，则与这些字节对应的 ASCII 字符将

显示在窗口右侧。

注意 默认情况下，“内存”窗口显示虚拟内存。 此类型的内存是在用户模式下唯一可用的内存类型。 在内核模式下，可以使用“ **内存选项** ”对话框显示物理内存和其他数据空间。 本主题稍后将介绍 “**内存选项** ”对话框。

在“内存”窗口中，可以执行以下操作：

- 若要写入内存，请在“内存”窗口中选择并键入新数据。 只能编辑十六进制数据，不能直接编辑 ASCII 和 Unicode 字符。 键入新信息后，更改将立即生效。
- 若要查看内存的其他部分，请使用“内存”窗口工具栏上的 “**上一** 步”和“ **下一步** ”按钮，或按 PAGE UP 或 PAGE DOWN 键。 这些按钮和键显示紧靠前面或后面的内存部分。 如果请求无效页面，将显示一条错误消息。
- 若要在窗口中导航，请使用向右键、向左键、向上键和向下键。 如果使用这些键移出页面，则会显示一个新页面。 使用这些键之前，应调整内存窗口的大小，使其没有滚动条。 通过此大小调整，可以区分实际页面边缘和窗口截断。
- 若要更改正在查看的内存位置，请在“内存”窗口顶部的地址框中输入新地址。 请注意，“内存”窗口在输入地址时会刷新其显示，因此在键入地址之前可能会收到错误消息。

  注意 输入到框中的地址将解释为当前基数。 如果当前基数不是 16，则应在十六进制地址前面加上 **0x**。 若要更改默认基数，请在“调试器命令”窗口中使用 n (Set Number Base) 命令。 内存窗口本身中的显示不受当前基数的影响。
- 若要更改窗口用于显示内存的数据类型，请使用“内存”窗口工具栏中的“ **显示格式** ”菜单。 支持的数据类型包括短字、双字和四字;短整数、长整数和四整数和无符号整数;10 字节、16 字节、32 位和 64 位实数;ASCII 字符;Unicode 字符;和十六进制字节。 十六进制字节的显示也包括 ASCII 字符。

“内存”窗口有一个工具栏，其中包含两个按钮、一个菜单和一个框，并具有包含其他命令的快捷菜单。 若要访问菜单，请选择并按住 (或右键单击) 标题栏，或选择工具栏上窗口右上角附近的图标。 Alt+SHIFT+5 关闭活动内存窗口。

- (工具栏仅) 使用地址框可以指定新的地址或偏移量。 此框的确切含义取决于你正在查看的内存类型。 例如，如果正在查看虚拟内存，则通过框可以指定新的虚拟地址或偏移量。
- (工具栏仅) **显示格式** 可用于选择新的显示格式。
- (工具栏和菜单) 工具栏) 上的 **上一** 个 (和快捷菜单上的 **上一页** () 导致显示上一部分内存。

- (工具栏和菜单) 工具栏) 上的 “**下一 (** ”和快捷菜单上的 “**下一页** ” (，) 显示下一部分内存。
- (菜单仅) **工具栏** 打开和关闭工具栏。
- (菜单仅) **自动调整列** 可确保“内存”窗口中显示的列数符合“内存”窗口的宽度。
- (菜单仅) **停靠** 或 **取消停靠** 会导致窗口进入或离开停靠状态。
- (菜单仅) **移动到新扩展坞** 会关闭“内存”窗口并在新扩展坞中打开它。
- (菜单仅) **设置为窗口类型的选项卡停靠目标** 会将所选内存窗口设置为其他内存窗口的选项卡停靠目标。 选择一个作为选项卡停靠目标后打开的所有内存窗口将自动与选项卡式集合中的该窗口分组。
- (菜单仅) **始终浮动** 会导致窗口保持未停靠状态，即使它被拖动到停靠位置。
- (菜单仅) **移动框架** 会导致在移动 WinDbg 框架时窗口移动，即使窗口未停靠也是如此。
- (菜单仅) **属性** 将打开 “**内存选项** ”对话框，本主题的下一部分对此进行介绍。
- (菜单仅) **帮助** 打开 Windows 调试工具文档中的本主题。
- (菜单仅) **关闭** 会关闭此窗口。

“内存选项”对话框

在快捷菜单中选择“ **属性**” 时，将显示“ **内存选项** ”对话框。

在内核模式下，有六种内存类型可用作此对话框中的选项卡： **虚拟内存**、 **物理内存**、 **总线数据**、 **控制数据**、 **I/O** (I/O 端口信息) ，以及 **MSR** (模型特定的寄存器信息) 。 选择与要访问的信息对应的选项卡。

在用户模式下，只有“ **虚拟内存** ”选项卡可用。

通过每个选项卡，可以指定要显示的内存：

- 在“ **虚拟内存** ”选项卡的“ **偏移量** ”框中，指定要查看的内存范围开头的地址或偏移量。
- 在“ **物理内存** ”选项卡的“ **偏移量** ”框中，指定要查看的内存范围开头的物理地址。 “内存”窗口只能显示描述的可缓存物理内存。 如果要显示具有其他属性的物理内存，请使用 d* (Display Memory) 命令或 ！ d* 扩展。

- 在“ **总线数据** ”选项卡的“ **总线数据类型** ”菜单中，指定总线数据类型。 然后，使用“**总线编号**”、“ **槽号**”和“ **偏移量** ”框指定要查看的总线数据。
- 在“ **控制数据** ”选项卡中，使用“ **处理器** ”和“ **偏移量** ”文本框指定要查看的控件数据。
- 在“ I/O ”选项卡的“ **接口类型** ”菜单中，指定 I/O 接口类型。 使用“**总线编号**”、“ **地址空间**”和“ **偏移量** ”框指定要查看的数据。
- 在“ MSR ”选项卡的“ MSR ”框中，指定要查看的特定于模型的寄存器。

每个选项卡还包括“ **显示格式** ”菜单。 此菜单的效果与“内存”窗口中的“**显示格式** ”菜单相同。

在“**内存选项**”对话框中选择“**确定**”，使更改生效。

其他信息

有关内存操作和其他内存相关命令的说明的详细信息，请参阅 读取和写入内存。

在 WinDbg (经典) 中查看和编辑全局变量

项目 • 2023/11/15

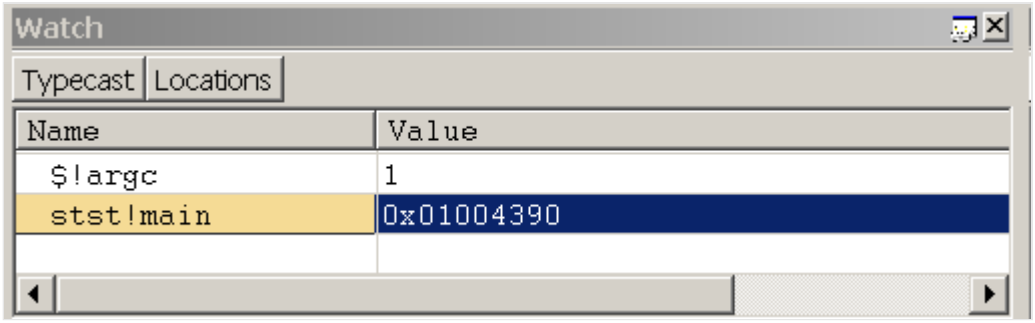
调试器将全局变量的名称解释为虚拟地址。 因此，可以使用 按虚拟地址访问内存 中所述的所有命令来读取或写入全局变量。

此外，可以使用 **？ (Evaluate Expression)** 命令显示与任何符号关联的地址。

在 WinDbg 中，还可以使用“监视”窗口显示和更改全局和局部变量。 “监视”窗口可以显示所需的任何变量列表。 这些变量可以包括来自任何函数的全局变量和局部变量。 “监视”窗口随时显示与当前函数的范围匹配的变量的值。 还可以通过“监视”窗口更改这些变量的值。

若要打开“监视”窗口，请从“**视图**”菜单中选择“**监视**”。 也可以按 Alt+2 或单击工具栏上的“**监视** ”按钮：

以下屏幕截图显示了监视窗口的示例。



监视窗口可以包含四列。 **始终显示“名称”**和“**值**”列，“**类型**”和“**位置**”列是可选的。 若要显示 “**类型**” 和“ **位置** ”列，请分别单击工具栏上的“ Typecast ”和“ **位置**” 按钮。

在 WinDbg (经典) 中查看和编辑局部变量

项目 • 2023/04/14

在 WinDbg 中，可以通过输入命令、使用“局部变量”窗口或使用“监视”窗口来查看局部变量。

调试器命令窗口

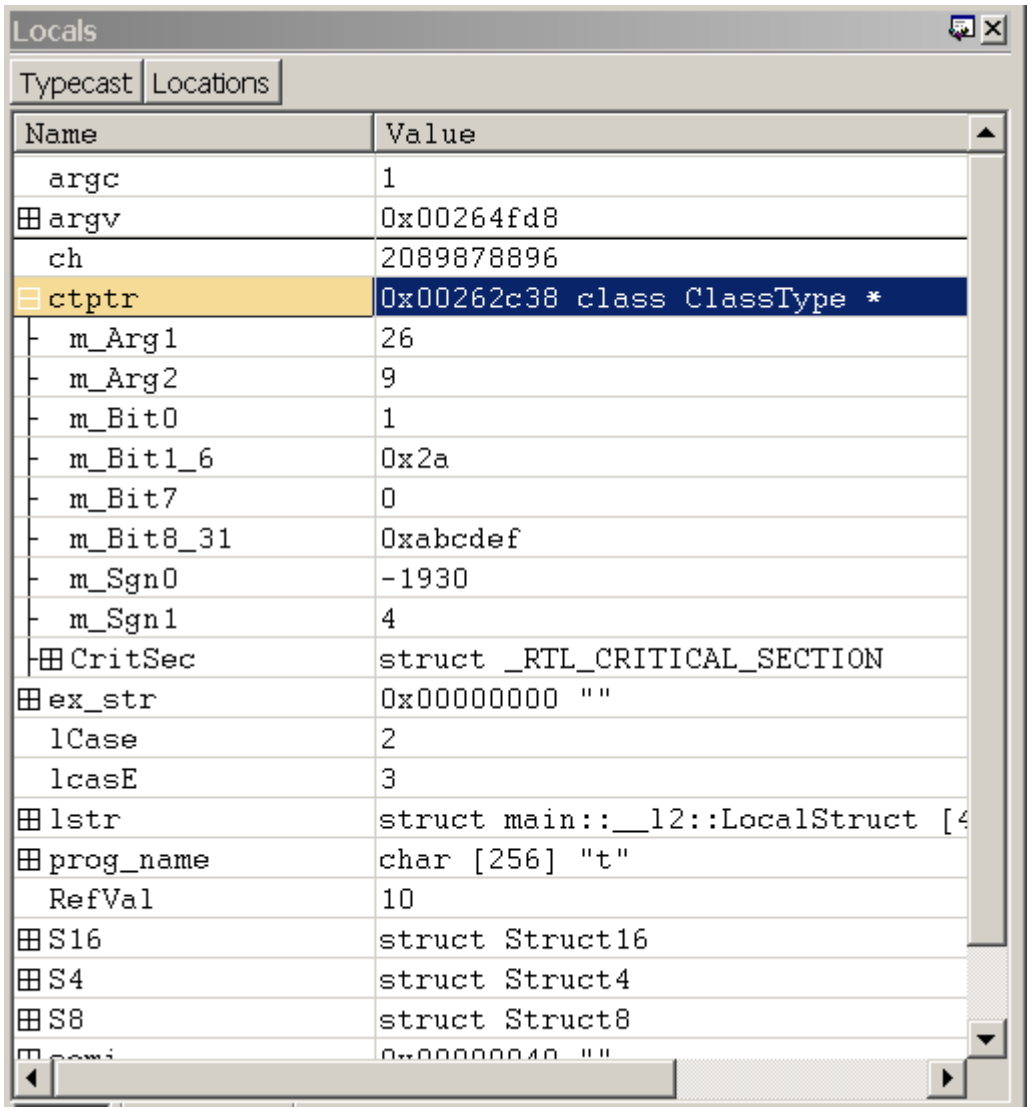
可以通过在“调试器命令”窗口中输入 **dv** 命令或 **dt** 命令来查看局部变量和参数。

打开“局部变量”窗口

“局部变量”窗口显示有关当前范围内所有局部变量的信息。

若要打开或切换到“局部变量”窗口，请在“WinDbg”窗口中的“ **视图** ”菜单上，选择“ **局部变量**”。 （还可以按 Alt+3 或选择 “**局部变量** ”按钮 (。) 工具栏上。 Alt+SHIFT+3 关闭“局部变量”窗口。）

以下屏幕截图显示了“局部变量”窗口的示例。



"局部变量"窗口可以包含四列。 **始终显示"名称"**和"**值**"列，"**类型**"和"**位置**"列是可选的。 若要显示 "**类型**" 和" **位置**" 列，请分别在工具栏上选择" Typecast "和" **位置**" 按钮。

使用"局部变量"窗口

在"局部变量"窗口中，可以执行以下操作：

- " **名称**" 列显示每个局部变量的名称。 如果变量是数据结构，则其名称旁边会显示一个检查框。 若要展开或折叠结构成员的显示，请选择或清除检查框。

- " **值** "列显示每个变量的当前值。
  - 若要为变量输入新值，请双击当前值并键入新值，或编辑旧值。 (剪切、复制和粘贴命令可用于编辑。) 可以键入任何 C++ 表达式。
  - 若要保存新值，请按 Enter。
  - 若要放弃新值，请按 ESC。
  - 如果键入无效值，则按 Enter 时，旧值将再次出现。

  int 类型的整数显示为十进制值;UINT 类型的整数显示在当前基数中。 若要更改当前基数，请在"调试器命令"窗口中使用 n (Set Number Base) 命令。

- 如果 “**类型**” 列显示在“局部变量”窗口中，则 (该列) 显示每个变量的当前数据类型。 每个变量都以适合其自己的数据类型的格式显示。 数据结构在 “**类型** ”列中具有其类型名称。 其他变量类型在此列中显示“输入新类型”。

  如果双击“输入新类型”，可以通过输入新的数据类型来强制转换类型。 此强制转换仅在“局部变量”窗口中更改此变量的当前显示;它不会更改调试器或目标计算机上的任何内容。 此外，如果在 “**值** ”列中输入新值，将根据符号的实际类型（而不是在“**类型** ”列中输入的任何新类型）分析输入的文本。 如果关闭并重新打开“局部变量”窗口，将丢失数据类型更改。

  还可以在 “**类型**” 列中输入扩展命令。 调试器会将符号的地址传递给此扩展，并在当前行下方的一系列可折叠行中显示生成的输出。 例如，如果此行中的符号是线程环境块的有效地址，则可以在“**类型**”列中输入 **！teb**，以在此符号的地址上运行 **！teb** 扩展。

- 如果 “**位置**” 列显示在“局部变量”窗口中， (“位置”列) 显示数据结构中每个成员的偏移量。

- 如果局部变量是包含 Vtable 的类的实例，则 **Name** 列将显示 Vtable，你可以展开 Vtable 以显示函数指针。 如果 Vtable 包含在指向派生实现的基类中，则显示表示法 **_vtcast_Class** 以指示由于派生类而添加的成员。 这些成员的扩展方式类似于派生类类型。

- 局部上下文确定将在“局部变量”窗口中显示哪一组局部变量。 当本地上下文因任何原因而更改时，“局部变量”窗口会自动更新。 默认情况下，本地上下文与程序计数器的当前位置匹配。 有关如何更改本地上下文的详细信息，请参阅本地上下文。

“局部变量”窗口具有一个工具栏，其中包含两个按钮 (**Typecast** 和 **Locations**)，以及一个包含其他命令的快捷菜单。 若要访问菜单，请选择并按住 (或右键单击) 窗口标题栏，或选择窗口右上角附近的图标， ( ) 。 工具栏和菜单包含以下按钮和命令：

- (工具栏和菜单) **Typecast** 打开和关闭 “**类型**” 列的显示。

- (工具栏和菜单) “**位置**” 打开和关闭“ **位置**” 列的显示。

- (菜单仅) **显示 16 位值，因为 Unicode** 在此窗口中显示 Unicode 字符串。 此命令打开和关闭影响“局部变量”窗口、“监视”窗口和调试器命令输出的全局设置。 此命令等效于使用 **.enable_unicode (启用 Unicode 显示)** 命令。

- (菜单仅) **始终在默认基数中显示数字** 会导致整数以默认基数显示，而不是以十进制格式显示它们。 此命令打开和关闭影响“局部变量”窗口、“监视”窗口和调试器命令输出的全局设置。 此命令等效于使用 **.force_radix_output (将 Radix 用于整数)** 命令。

注意“**始终在默认基数中显示数字**”命令不会影响长整数。 长整数以十进制格式显示，除非设置了 .enable_long_status (**启用长整型显示**) 命令。 .enable_long_status 命令会影响“局部变量”窗口、“监视”窗口和调试器命令输出中的显示;在“局部变量”窗口的菜单中，此命令没有等效项。

- (菜单仅) **打开所选值的内存窗口** 会打开一个新的停靠内存窗口，该窗口显示从所选表达式的地址开始的内存。
- (菜单仅) **调用所选内存值的 dt运行 dt (显示类型)** 命令，并将所选符号作为其参数。 结果显示在“调试器命令”窗口中。 **-n** 选项会自动用于区分符号和十六进制地址。 不使用其他选项。 请注意，使用此菜单选择生成的内容与从命令行运行 **dt** 命令时生成的内容相同，但格式略有不同。
- (菜单仅) **工具栏** 打开和关闭工具栏。
- (菜单仅) **停靠** 或 **取消停靠** 会导致窗口进入或离开停靠状态。
- (菜单仅) **移动到新扩展坞** 会关闭“局部变量”窗口，并在新扩展坞中打开它。
- (菜单仅) **设置为窗口类型的选项卡停靠目标** 对“局部变量”窗口不可用。 此选项仅适用于“源”或“内存”窗口。
- (菜单仅) **始终浮动** 会导致窗口保持未停靠状态，即使它被拖动到停靠位置。
- (菜单仅) **移动框架** 会导致在移动 WinDbg 帧时窗口移动，即使窗口已取消停靠也是如此。
- (菜单仅) **帮助** 打开 Windows 调试工具文档中的本主题。
- (菜单仅) **关闭** 会关闭此窗口。

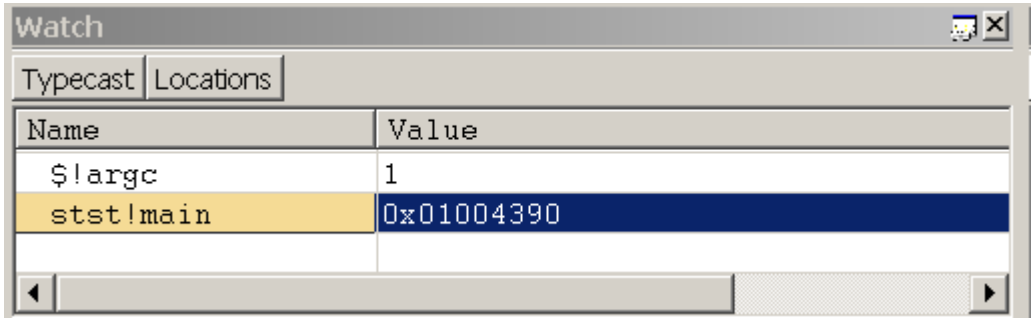
监视窗口

在 WinDbg 中，可以使用“监视”窗口显示和更改局部变量。 “监视”窗口可以显示所需的任何变量列表。 这些变量可以包括来自任何函数的全局变量和局部变量。 “监视”窗口随时显示与当前函数的范围匹配的变量的值。 还可以通过“监视”窗口更改这些变量的值。

与“局部变量”窗口不同，“监视”窗口不受本地上下文更改的影响。 只有当前程序计数器范围内定义的变量才能显示或修改其值。

若要打开“监视”窗口，请从“**视图**”菜单中选择“**监视**”。 还可以按 Alt+2 或选择工具栏上的“**监视**”按钮： 。 Alt+SHIFT+2 关闭“监视”窗口。

以下屏幕截图显示了监视窗口的示例。



监视窗口可以包含四列。 **始终显示“名称”**和“**值**”列，“**类型**”和“**位置**”列是可选的。 若要显示 “**类型**” 和“ **位置**” 列，请分别在工具栏上选择“ Typecast ”和“ **位置**” 按钮。

其他信息

有关控制局部变量的详细信息、使用变量和更改范围的概述以及其他内存相关命令的说明，请参阅 读取和写入内存。

在 WinDbg (经典) 中查看和编辑寄存器

项目 • 2023/10/26

寄存器是位于 CPU 上的小型易失性内存单元。 许多寄存器专用于特定用途，其他寄存器可供用户模式应用程序使用。 基于 x86 和基于 x64 的处理器具有可用的寄存器的不同集合。 有关每个处理器上的寄存器的详细信息，请参阅 处理器体系结构。

在 WinDbg 中，可以通过输入命令、使用“寄存器”窗口或使用“监视窗口”来查看和编辑寄存器。

命令

可以通过在“调试器命令”窗口中输入 **r (Registers)** 命令来查看和编辑寄存器。 可以使用多个选项或使用 **rm (Register Mask)** 命令自定义显示。

每次目标停止时，也会自动显示寄存器。 如果使用 **p (Step)** 或 **t (Trace)** 命令单步执行代码，则每个步骤都会显示寄存器。 若要停止此显示，请在使用这些命令时使用 **r** 选项。

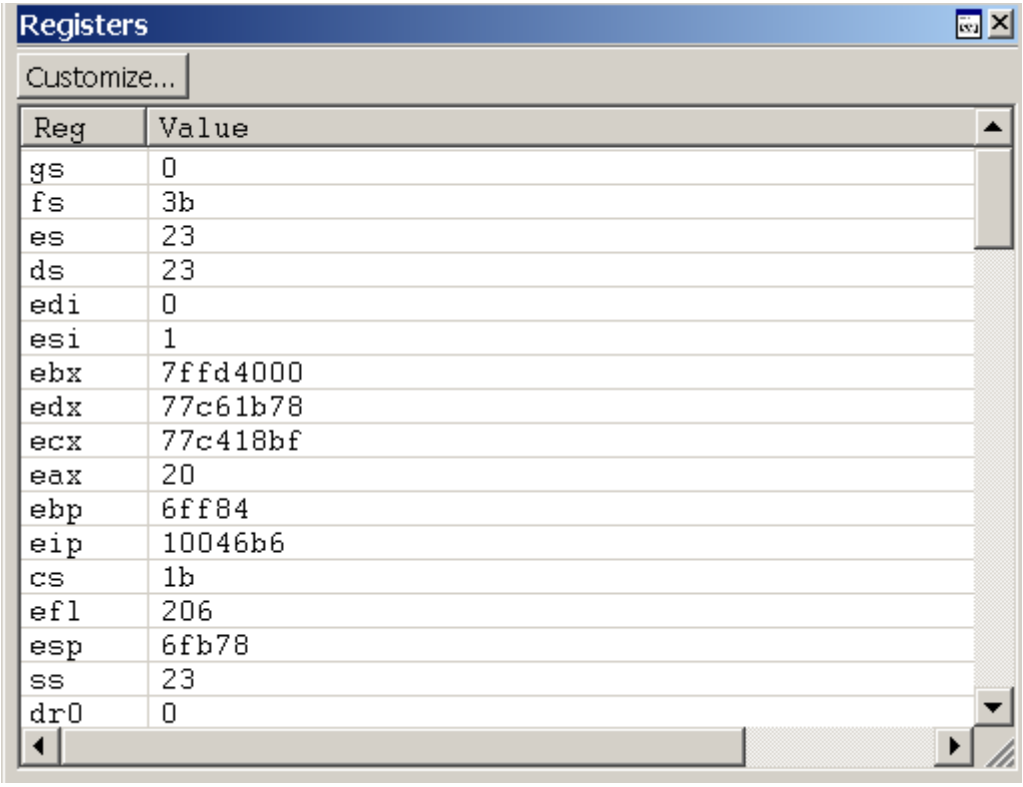
在基于 x86 的处理器上， **r** 选项还控制多个称为标志的一位寄存器。 若要更改这些标志，请使用与更改常规寄存器时略有不同的语法。 有关这些标志的详细信息和此语法的说明，请参阅 x86 Flags。

寄存器窗口

打开“寄存器”窗口

若要打开或切换到“寄存器”窗口，请从“**视图**”菜单中选择“**寄存器**”。 (还可以按 Alt+4 或选择 “**寄存器** ”按钮 (。) 工具栏上。 Alt+Shift+4 关闭寄存器窗口。)

以下屏幕截图显示了“寄存器”窗口的示例。



“寄存器”窗口包含两列。 Reg 列列出目标处理器的所有寄存器。 “ **值**” 列显示每个寄存器的当前值。 此窗口还包含用于打开“**自定义注册列表**”对话框的工具栏上的“自定义”按钮。

使用“寄存器”窗口

在“寄存器”窗口中，可以执行以下操作：

- “ **值**” 列显示每个寄存器的当前值。 最近更改的寄存器的值以红色文本显示。
  - 若要输入新值，请双击 “**值**” 单元格，然后键入新值或编辑旧值。 (剪切、复制和粘贴命令可用于编辑。)
  - 若要保存新值，请按 Enter。
  - 若要放弃新值，请按 ESC。
  - 如果键入无效值，则按 Enter 时，旧值将重新出现。

- 寄存器值显示在当前基数中，并且必须在相同的基数中键入新值。 若要更改当前基数，请在“调试器命令”窗口中使用 n (Set Number Base) 命令。

- 在用户模式下，“寄存器”窗口显示与当前线程关联的寄存器。 有关当前线程的详细信息，请参阅 控制进程和线程。

- 在内核模式下，“寄存器”窗口显示与当前 寄存器上下文关联的寄存器。 可以设置寄存器上下文以匹配特定的线程、上下文记录或陷阱帧。 实际仅显示指定寄存器上下文的最重要寄存器;无法更改其值。

“寄存器”窗口有一个工具栏，其中包含“ **自定义** ”按钮，并具有包含其他命令的快捷菜单。 若要访问菜单，请选择并按住 (右键单击) 标题栏，或选择窗口右上角附近的图标，

(用于) 。 工具栏和菜单包含以下按钮和命令：

- (工具栏和菜单) **“自定义”** 将打开“ **自定义注册列表** ”对话框，本主题的下一节对此进行介绍。
- 仅 (菜单) **工具栏** 打开和关闭工具栏。
- (菜单仅) **停靠** 或 **取消停靠** 会导致窗口进入或离开停靠状态。
- (菜单仅) **移动到新扩展坞** 会关闭“寄存器”窗口并在新扩展坞中打开它。
- (菜单仅) **设置为窗口类型的选项卡停靠目标** 对于“寄存器”窗口不可用。 此选项仅适用于“源”或“内存”窗口。
- (菜单仅) **始终浮动** 会导致窗口保持取消停靠状态，即使它被拖动到停靠位置。
- (菜单仅) **移动框架** 会导致在移动 WinDbg 框架时窗口移动，即使窗口已取消停靠也是如此。
- (菜单“仅) **帮助** ”打开 Windows 调试工具文档中的本主题。
- (菜单仅) **关闭** 关闭此窗口。

“自定义注册列表”对话框

若要更改显示的寄存器列表，请选择“ **自定义** ”按钮。 将显示“ **自定义注册列表** ”对话框。

在此对话框中，可以编辑寄存器列表以更改寄存器的显示顺序。 (实际上无法从列表中删除寄存器;如果这样做，它将在末尾重新出现。) 寄存器名称之间必须有一个空格。

如果选择“**首先检查显示修改的寄存器值**”框，则最近更改其值的寄存器将显示在顶部。

如果选择“**不显示子注册**检查”框，则不会显示子注册。 例如，将显示 **eax** ，但不会显示 **ax**、 **ah** 或 **al**。

选择 **“确定”** 保存更改，或 **选择“取消”** 以放弃更改。

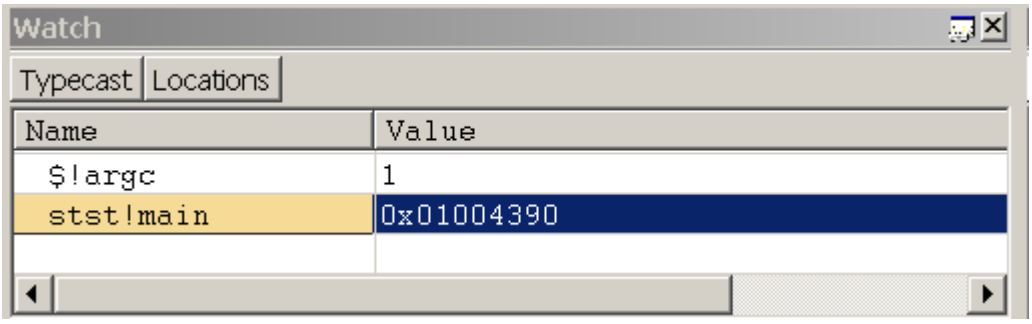
如果要调试具有多种处理器的多处理器计算机，WinDbg 将单独存储每种处理器类型的自定义设置。 通过这种分离，可以同时自定义每个处理器寄存器的显示。

监视窗口

在 WinDbg 中，可以使用“监视”窗口显示寄存器。 不能使用“监视”窗口更改寄存器的值。

若要打开“监视”窗口，请从“**视图**”菜单中选择“**监视**”。 还可以按 Alt+2 或选择工具栏上的“**监视**”按钮： 。 Alt+Shift+2 关闭“监视”窗口。

以下屏幕截图显示了“监视”窗口的示例。



其他信息

有关寄存器上下文和其他上下文设置的详细信息，请参阅 更改上下文。

控制 WinDbg 中的进程和线程 (经典)

项目 • 2023/04/14

在 WinDbg 中，“进程和线程”窗口显示有关正在调试的系统、进程和线程的信息。 此窗口还可用于选择要处于活动状态的新系统、进程和线程。

打开“进程和线程”窗口

若要打开“进程和线程”窗口，请从“**视图**”菜单中选择“**进程和线程**”。 (还可以按 Alt+9 或选择“ **进程和线程** ”按钮 (。在工具栏上) 。 ALT+SHIFT+9 关闭“进程和线程”窗口。)

以下屏幕截图显示了“进程和线程”窗口的示例。

“进程和线程”窗口显示当前正在调试的所有进程的列表。 进程中的线程显示在每个进程下。 如果调试器附加到多个系统，则系统显示在树的顶层，进程从属于这些系统，线程从属于进程。

每个系统列表都包含服务器名称和协议详细信息。 运行调试器的系统标识为 **<Local>**。

每个进程列表包括调试器使用的内部十进制进程索引、十六进制进程 ID 以及与进程关联的应用程序的名称。

每个线程列表包括调试器使用的内部十进制线程索引和十六进制线程 ID。

使用“进程和线程”窗口

在“进程和线程”窗口中，当前或活动系统、进程和线程以粗体类型显示。 若要使新的系统、进程或线程处于活动状态，请在窗口中选择其行。

“进程和线程”窗口具有包含其他命令的快捷菜单。 若要访问菜单，请选择并按住 (或右键单击标题栏)，或选择窗口右上角附近的图标， ( ) 。 以下列表描述了一些菜单命令：

- **移动到新扩展坞** 会关闭“进程和线程”窗口，并在新扩展坞中打开它。
- **始终浮动** 会导致窗口保持未停靠状态，即使它被拖动到停靠位置。
- **随帧移动** 会导致移动 WinDbg 框架时窗口移动，即使窗口已取消停靠也是如此。

其他信息

有关显示或控制系统的其他方法，请参阅 调试多个目标。 有关显示或控制进程和线程的其他方法，请参阅 控制进程和线程。

使用监视窗口 - WinDbg (经典)

项目 • 2023/10/26

“监视”窗口显示有关全局变量、局部变量和寄存器的信息。 可以自定义此窗口以显示要跟踪的项。

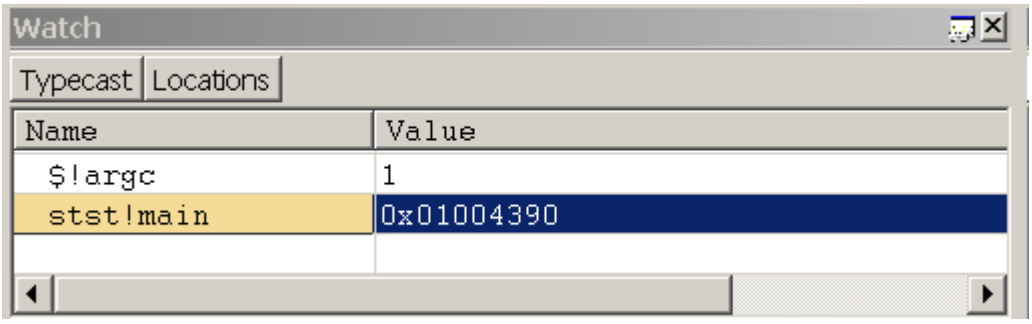
打开监视窗口

若要打开或切换到“监视”窗口，请在 WinDbg 窗口中的“ **视图** ”菜单上，单击“ **监视**”。

还可以按 Alt+2 或单击工具栏上的“**监视 (Alt+2)** ”按钮：“

Alt+SHIFT+2 将关闭“监视”窗口。

以下屏幕截图显示了监视窗口的示例。



监视窗口可以包含四列。 **始终显示“名称”和“值”列，“类型”和“位置”列是可选的。** 若要显示 “**类型**” 和“ **位置** ”列，请分别单击工具栏上的“ Typecast ”和“ **位置**” 按钮。

使用监视窗口 - WinDbg (经典)

在“监视”窗口中，可以执行以下操作：

- 若要将变量添加到“监视”窗口，请选择“ **名称**” 列中的第一个空单元格，键入变量名称，然后按 Enter。 使用感叹号 (**!**) 将模块名称与变量分开。 如果未指定模块，则使用当前模块。 若要在 “**名称**” 字段中输入地址，地址必须以十进制数字 (开头（如有必要），请使用前缀 **0x**) 。

  如果输入的变量名称在当前函数的作用域中定义，则其值将显示在 “**值** ”列中。 如果未定义， “**值**”列将显示“错误：无法获取值”。

  即使未定义变量，将其添加到“监视”窗口也很有用。 如果程序计数器输入在其中定义了此名称的变量的函数，则此时其值将显示在窗口中。

- 若要从“监视”窗口中删除变量，请双击其名称，按 DELETE，然后按 Enter。 还可以将旧名称替换为新名称，方法是双击旧名称，键入新名称，然后按 Enter。

- 如果变量是数据结构，则其名称旁边会显示一个检查框。 若要展开和折叠结构成员的显示，请选择或清除检查框。

- int 类型的整数显示为十进制值;UINT 类型的整数显示在当前基数中。 若要更改当前基数，请在“调试器命令”窗口中使用 n (Set Number Base) 命令。

- 若要更改局部变量的值，请双击其 Value 单元格。 输入新值，或编辑旧值。 (剪切、复制和粘贴命令可用于编辑。) 输入的值可以包含任何 C++ 表达式。 输入新值或编辑旧值后，可以按 Enter 来存储新值，或按 ESC 将其丢弃。 如果提交无效值，则按 Enter 后，旧值将再次出现。

  int 类型的整数显示为十进制值;UINT 类型的整数显示在当前基数中。 若要更改当前基数，请在“调试器命令”窗口中使用 n (Set Number Base) 命令。

- 如果“监视”窗口中显示 “**类型**” 列，则 (“类型”列) 显示每个变量的当前数据类型。 每个变量都以适合其自己的数据类型的格式显示。 数据结构在 “**类型** ”列中具有其类型名称。 其他变量类型在此列中显示“输入新类型”。

  如果双击“输入新类型”，可以通过输入新的数据类型来强制转换类型。 此强制转换仅在“监视”窗口中更改此变量的当前显示;它不会更改调试器或目标计算机上的任何内容。 此外，如果在 “**值** ”列中输入新值，将根据符号的实际类型（而不是在“ **类型** ”列中输入的任何新类型）分析输入的文本。 如果关闭并重新打开“监视”窗口，则会丢失数据类型更改。

  还可以在 “**类型**” 列中输入扩展命令。 调试器会将符号的地址传递给此扩展，并在当前行下方的一系列可折叠行中显示生成的输出。 例如，如果此行中的符号是线程环境块的有效地址，则可以在“**类型**”列中输入 **！ teb**，以在此符号的地址上运行 **！ teb** 扩展。

- 如果 “**位置**” 列显示在“监视”窗口中，则 (该列) 显示数据结构中每个成员的偏移量。

- 除了变量，还可以在“监视”窗口中监视以下项：
  - 寄存 器。 将寄存器添加到“监视”窗口时，在其名称前面加上 at 符号 (@) 。 与变量不同，无法通过“监视”窗口更改寄存器值。
  - 包含函数指针的 Vtable。 当 Vtable 出现在“监视”窗口中时，可以浏览表中的函数条目。 如果 Vtable 包含在指向派生实现的基类中，则显示表示法 **_vtcast_***Class* 以指示由于派生类而添加的成员。 这些成员的扩展方式类似于派生类类型。
  - 扩展函数的返回值，例如 _EFN_GetPoolData。

与“ 局部变量”窗口不同，“监视”窗口不受 对寄存器上下文的更改的影响。 在“监视”窗口中，只能查看和修改在当前程序计数器范围内定义的变量。

如果打开新工作区，则“监视”窗口内容将被丢弃，并替换为新工作区中的内容。

工具栏和快捷菜单

“监视”窗口的工具栏包含两个按钮 (Typecast 和 Locations)，以及一个包含其他命令的快捷菜单。 若要访问菜单，请右键单击窗口的标题栏或单击窗口右上角附近的图标：

工具栏和菜单包含以下按钮和命令：

- (工具栏和菜单) Typecast 打开和关闭 “**类型**” 列的显示。
- (工具栏和菜单) “**位置**” 打开和关闭“ **位置**” 列的显示。
- (菜单仅) **显示 16 位值，因为** Unicode 在此窗口中显示 Unicode 字符串。 此命令打开和关闭影响 “局部变量”窗口、“监视”窗口和调试器命令输出的全局设置。 此命令等效于使用 .enable_unicode (**启用 Unicode 显示**) 命令。
- (菜单仅) **始终在默认基数中显示数字** 会导致整数显示在默认基数中，而不是始终以十进制格式显示它们。 此命令打开和关闭影响“局部变量”窗口、“监视”窗口和调试器命令输出的全局设置。 此命令等效于使用 .force_radix_output (**将 Radix 用于整数**) 命令。 **注意“始终在默认基数中显示数字”**命令不会影响长整数。 长整数以十进制格式显示，除非使用 .enable_long_status (**启用长整型显示**) 命令。 .enable_long_status 命令会影响“局部变量”窗口、“监视”窗口和调试器命令输出中的显示。 在“监视”窗口的菜单中，此命令没有等效项。
- (菜单仅) **打开所选值的内存窗口** 会打开一个新的停靠内存窗口，该窗口显示从所选表达式的地址开始的内存。
- (菜单仅) **调用所选内存值的** dt 运行 dt (**显示类型**) 命令，并将所选符号作为其参数。 结果显示在 “调试器命令”窗口中。 -n 选项会自动用于区分符号和十六进制地址。 不使用其他选项。 请注意，使用此菜单选择生成的内容与从命令行运行 dt 命令时生成的内容相同，但格式略有不同。
- (菜单仅) **工具栏** 打开和关闭工具栏。
- (菜单仅) **停靠** 或 **取消停靠** 会导致窗口进入或离开停靠状态。
- (菜单仅) **移动到新扩展坞** 会关闭“监视”窗口，并在新扩展坞中打开它。
- (菜单仅) **设置为窗口类型的选项卡停靠目标** 对监视窗口不可用。 此选项仅适用于 “源” 或 “内存” 窗口。

- (菜单仅) **始终浮动** 会导致窗口保持未停靠状态，即使它被拖动到停靠位置。
- (菜单仅) **移动框架** 会导致在移动 WinDbg 帧时窗口移动，即使窗口已取消停靠也是如此。
- (菜单仅) **帮助** 打开 Windows 调试工具文档中的本主题。
- (菜单仅) **关闭** 会关闭此窗口。

其他信息

有关控制变量的详细信息和其他内存相关命令的说明，请参阅 读取和写入内存。 有关寄存器及其操作的详细信息，请参阅 查看和编辑 WinDbg (经典) 中的寄存器 。

使用 Scratch Pad - WinDbg (经典) - WinDbg (经典)

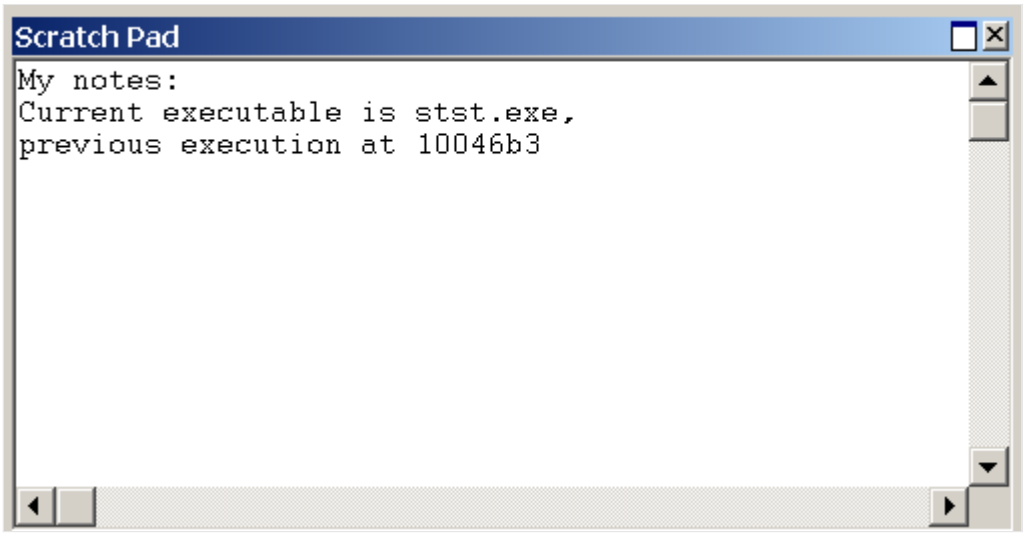
项目 • 2023/11/15

“暂存板”窗口是可在其中键入和保存文本的剪贴板。

打开“暂存板”窗口

若要打开或切换到“暂存板”窗口，请在 WinDbg 窗口中的“ **视图** ”菜单上，单击“ **暂存板**”。 (还可以按 Alt+8 或单击“ **暂存板 (Alt+8)** ”按钮 (

以下屏幕截图显示了“暂存板”窗口的示例。



使用暂存板 - WinDbg (经典) - WinDbg (经典) 窗口

在“暂存板”窗口中，可以执行以下操作：

- 若要在“暂存板”窗口中键入，请单击要添加文本的窗口中，然后开始键入。 还可以使用标准复制和粘贴功能。 Scratch Pad 窗口的内容不会影响调试器的操作。 此窗口的存在只是为了帮助编辑文本。
- 如果关闭“暂存板”窗口，文本将保留，并在重新打开窗口时可用。 还可以通过将文本与文件相关联，从“暂存板”窗口中保存文本。

“暂存板”窗口有一个快捷菜单，其中包含其他命令。 若要访问菜单，请右键单击标题栏或单击右上角附近的图标。 此菜单包含以下命令：

- (菜单仅) “**与文件关联**” 会打开一个对话框，通过该对话框可以选择文本文件。 选择文件后，将清除暂存板中的当前文本，并将其替换为所选文件中的文本。 虽然 Scratch Pad 与此文件相关联，但所有键入 Scratch Pad 的新文本都会保存到该文件中。 可以通过选择“ **结束文件关联** ”短切菜单选项或关闭并重新打开 Scratch Pad 来结束与文件的关联。

- (菜单仅) **结束文件关联** 结束 Scratch Pad 与指定文本文件的关联。 选择此选项之前，Scratch Pad 中的所有文本都保存在 文件中。 关联结束后在 Scratch Pad 中键入的所有文本不再保存在文本文件中。

- **停靠** 或 **取消停靠** 会导致窗口进入或离开停靠状态。

- (菜单仅) **移动到新扩展坞** 会关闭 Scratch Pad 并在新扩展坞中打开它。

- 仅 (菜单) **设置为窗口类型的选项卡停靠目标** 不适用于 Scratch Pad。 此选项仅适用于 “源” 或 “内存” 窗口。

- **始终浮动** 会导致窗口保持未停靠状态，即使它被拖动到停靠位置。

- **随帧移动** 会导致移动 WinDbg 框架时窗口移动，即使窗口已取消停靠也是如此。

- “**帮助**” 在 Windows 调试工具文档中打开本主题。

- **关闭** 将关闭此窗口。

使用工作区 - WinDbg (经典)

项目 • 2023/04/14

退出 WinDbg 时，它会将会话配置保存在 *工作区*中。 工作区使你可以轻松地将设置从一个会话保存到另一个会话。 还可以手动保存或清除工作区，甚至使用工作区保存仍在进行的调试会话。

本节包括下列主题：

创建和打开工作区

工作区内容

使用和自定义 WinDbg 主题 - WinDbg (经典)

创建和打开工作区

项目 • 2024/07/08

WinDbg 有两种类型的工作区： *默认工作区* 和 *命名工作区*。

默认工作区

WinDbg 具有多种不同类型的默认工作区：

- *当 WinDbg 处于休眠状态时，将使用基本工作区。*
- *附加到用户模式进程时使用默认用户模式工作区*（使用 -p**命令行选项**或使用**文件 | 附加到** Process 命令。
- *连接到调试服务器时，将使用远程默认工作区。*
- *当 WinDbg 开始内核模式调试会话时，将使用默认内核模式工作区。*
- WinDbg 附加到目标计算机后，处理器 *特定的工作区* 在内核模式调试期间使用。 对于基于 x86 的处理器和基于 x64 的处理器，有单独的特定于处理器的工作区。

当 WinDbg 创建用于调试的用户模式过程时，将为该可执行文件创建工作区。 每个创建的可执行文件都有自己的工作区。

当 WinDbg 分析转储文件时，将为该转储文件分析会话创建一个工作区。 每个转储文件都有自己的工作区。

开始调试会话时，将加载相应的工作区。 结束调试会话或退出 WinDbg 时，将显示一个对话框，询问是否要保存对当前工作区所做的更改。 如果使用 -QY**命令行选项****启动 WinDbg**，则不会显示此对话框，并且工作区会自动保存。 此外，如果通过 **-Q** 命令行选项启动 WinDbg，则不会显示此对话框，并且不会保存任何更改。

工作区以累积方式加载。 始终首先加载基本工作区。 开始特定的调试操作时，将加载相应的工作区。 因此，大多数调试是在加载两个工作区后完成的。 在加载了三个工作区（基本工作区、默认内核模式工作区和特定于处理器的工作区）之后，内核模式调试已完成。

为了提高效率，如果希望将设置应用于所有 WinDbg 工作，则应将设置保存在较低级别的工作区中。

请注意，调试信息窗口的布局是工作区累积行为的一个例外。 每个窗口的位置、停靠状态和大小仅由打开的最新工作区决定。 此行为包括“监视”窗口的内容以及在每个 “内存”

窗口中查看的位置。 打开新工作区时，调试器命令窗口中的命令历史记录不会清除，但所有其他窗口状态都会重置。

若要访问基本工作区，请启动没有目标的 WinDbg，或在会话完成后单击“调试”**菜单上的**“**停止调试**”。 然后，可以进行基工作区中允许的任何编辑。

命名工作区

还可以提供工作区名称，然后单独保存或加载它们。 加载命名工作区后，将禁用默认工作区的所有自动加载和保存。

命名工作区包含一些默认工作区未提供的其他信息。 有关此附加信息的详细信息，请参阅 工作区内容。

打开、保存和清除工作区

若要控制工作区，可以执行以下操作：

- 使用 -W **命令行选项**打开并加载命名工作区。
- 使用 -WF **命令行选项**从文件打开并加载工作区。
- 使用 -WX **命令行选项**禁用所有自动工作区加载。 只有显式工作区命令才能保存或加载工作区。
- 通过单击“文件”**菜单上的**“打开工作区”或按 Ctrl+W 打开并加载命名工作区。
- 单击“文件”菜单上的“保存工作区”，**保存当前默认工作区或当前命名工作区。**
- 在“文件”**菜单上单击“另存为”，将名称分配给当前工作区**并将其保存。
- 单击“文件”**菜单上的“清除工作区”**，从当前工作区中删除特定项和设置。
- 单击“文件”**菜单上的**“删除工作区”**来删除工作区。**
- 通过单击“文件”菜单上的“文件”**中的“打开工作区”**，从文件打开并加载工作区。
- 单击“文件”菜单上的“**将工作区保存到文件”，将工作区保存到文件。**

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

工作区内容

项目 • 2024/10/31

每个工作区保留有关当前调试会话的以下信息。 此信息是累积应用的，从基本工作区开始，以最近加载的工作区结束。

- 所有中断和处理异常和事件的信息。 有关中断和处理信息的详细信息，请参阅工作区中的断点。
- 所有开放源代码文件。 如果未找到源文件，将显示一条错误消息。 可以单独关闭这些错误消息，也可以使用 **窗口关闭 |关闭所有错误** Windows 命令。
- 所有用户定义别名。

每个工作区保留有关调试器配置设置的以下信息。 此信息是累积应用的，从基本工作区开始，以最近加载的工作区结束。

- 符号路径。
- 可执行映像路径。
- 源路径。 （在远程调试中，保存主源路径和本地源路径。
- 使用 **l+、l- （设置源选项）** 设置的当前源选项。
- 日志文件设置。
- 如果连接是使用图形界面启动的，则 COM 内核连接设置。
- 每个 “**打开** ”对话框中的最新路径（工作区文件和文本文件路径除外，未保存）。
- 当前.enable_unicode、.force_radix_output和.enable_long_status设置。

所有默认工作区和命名工作区都保留有关 WinDbg 图形界面的以下信息。 此信息是累积加载的，从基本工作区开始，以最近加载的工作区结束。

- WinDbg 窗口的标题
- 自动打开反汇编设置
- 默认字体

所有默认工作区和命名工作区都保留有关 WinDbg 图形界面的以下信息。 此信息不会累积应用。 它仅取决于最近加载的工作区。

- 桌面上的 WinDbg 窗口的大小和位置。

- 打开哪些调试信息窗口。
- 每个打开窗口的大小和位置，包括窗口的大小、其浮动状态或停靠状态、它是否与其他窗口选项卡以及其快捷菜单中的所有相关设置。
- “调试器命令”窗口中窗格边界的位置以及该窗口中的换行设置。
- 工具栏和状态栏以及每个调试信息窗口中的各个工具栏是否可见。
- “注册”窗口的自定义。
- “呼叫”窗口、“局部变量”窗口和“监视”窗口中的标志。
- 在“监视”窗口中查看的项目。
- 每个 源窗口中的光标位置。

命名工作区

命名工作区包含未存储在默认工作区中的其他信息。

此附加信息包括有关当前会话状态的信息。 保存命名工作区后，将保存当前会话。 如果稍后打开此工作区，则会自动重启此会话。

只能以这种方式启动内核调试、转储文件调试和生成用户模式进程的调试。 调试器附加到的远程会话和用户模式进程在其工作区中未保存此会话信息。

如果另一个会话已处于活动状态，则无法打开此类命名工作区。

调试客户端和工作区

使用 WinDbg 作为调试客户端时，其工作区仅保存通过图形界面设置的值。 不会保存通过调试器命令窗口所做的更改。 （此限制保证仅反映本地客户端所做的更改，因为调试器命令窗口接受来自所有客户端和调试服务器的输入。有关详细信息，请参阅 控制远程调试会话。

工作区中的断点

此外，断点信息保存在工作区中，包括中断地址和状态。 当会话结束时处于活动状态的断点在启动下一个会话时处于活动状态。 但是，如果尚未加载适当的模块，则其中一些断点可能无法解决。

通过符号表达式、行号、数字地址或在源窗口中使用鼠标指定的断点全部保存在工作区中。 在反汇编或“调用”窗口中使用鼠标指定的断点不会保存在工作区中。

如果要调试多个用户模式进程，则仅保存与进程零关联的断点。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用和自定义 WinDbg 主题 - WinDbg (经典)

项目 • 2023/04/14

主题是预配置的 WinDbg 工作区，其中包含调试信息窗口的有用配置。

任何主题都可以保存为基本工作区。 Windows 调试工具包中的主题作为一组注册表文件提供，（扩展名为 .reg）。 随着累积更多的调试会话，会自动设置各种默认工作区。 这些默认工作区使用基本工作区作为起点。 有关默认工作区的详细信息，请参阅 创建和打开工作区。

为了获得最佳用户体验，建议在启动调试会话之前按照本主题中的说明进行操作。

本节包括下列主题：

加载主题

自定义主题

使用 Windows 调试工具中提供的主题

加载主题

项目 • 2023/06/15

在加载主题之前，建议清除所有工作区数据。 这可以通过三种方式完成：

通过使用 WinDbg 用户界面。 在“ **文件** ”菜单下，选择“ **清除工作区...** ”在弹出窗口中选择“ **全部清除** ”，然后单击“ **确定** ”。

通过删除 **HKCU\Software\Microsoft\Windbg\Workspaces** 下的注册表项。

通过运行命令 **reg，删除 HKCU\Software\Microsoft\Windbg**。

清除所有工作区数据后，请运行其中一个主题。 这些文件以 .reg 文件的形式存储在 Windows 安装调试工具的“主题”目录中。 运行主题会将其设置导入注册表，从而重新定义基本工作区。

加载主题后，可以对其进行更改，使其更符合你的偏好。 有关一些常见选项的更多详细信息，请参阅 自定义主题。

自定义主题

项目 • 2023/04/14

在自定义主题之前，必须先加载该主题。 有关详细信息，请参阅加载主题 。

加载主题后，不使用命令行参数启动 WinDbg。 这会打开基本工作区。 自定义主题有两个常见的焦点区域：设置路径和调整窗口位置。

完成所需的任何调整后，退出 WinDbg 并通过从“**文件**”菜单中选择“**保存工作区**”来保存工作区。 如果要将新设置保存到 .reg 文件，请打开 Regedit 并将 **HKCU\Software\Microsoft\Windbg\Workspaces** 下的注册表项导出到 .reg 文件。

设置路径

通过设置适当的路径，可以确保 WinDbg 能够找到有效调试所需的所有文件。 有三个main路径需要设置：符号路径、源路径和可执行映像路径。

下面是如何设置符号和源路径的示例。 可执行映像路径通常与符号路径相同。

设置符号路径：

```text
SRV*c:\MySymCache*\\CompanySymbolServer\Symbols;SRV*c:\WinSymCache*https://msdl.microsoft.com/download/symbols
```

设置源路径：

```text
SRV*;d:\MySourceRoot
```

调整窗口位置

在使用主题之前，应调整窗口定位，以便 WinDbg 正确处理源文件。 这可确保源窗口知道停靠的位置。

首先在 WinDbg 中打开“源”窗口。 使用为源窗口留出占位符的 Tab 键停靠此窗口。 为了建立正确的关系，在执行此 Tab-停靠操作之前，占位符窗口必须是停靠中最上面的窗口。 现在关闭源窗口，但不关闭占位符窗口。

由于调试信息窗口"记住"其上次停靠操作，因此，执行此过程后，每个源窗口的最后一个停靠操作都与占位符窗口之一相关联。 由于此内存属性，不应关闭任何占位符窗口。 此外，如果选择更改主题的配置，在停靠中重新定位的任何窗口应始终使用占位符文件以 Tab 键停靠。

Windows 调试工具中包含的示例主题是使用以下操作创建的：

将占位符*.c 文件放置到扩展坞中。

将每个窗口类型的 Tab 键停靠在所需占位符窗口上方。

使用 Windows 调试工具中提供的主题

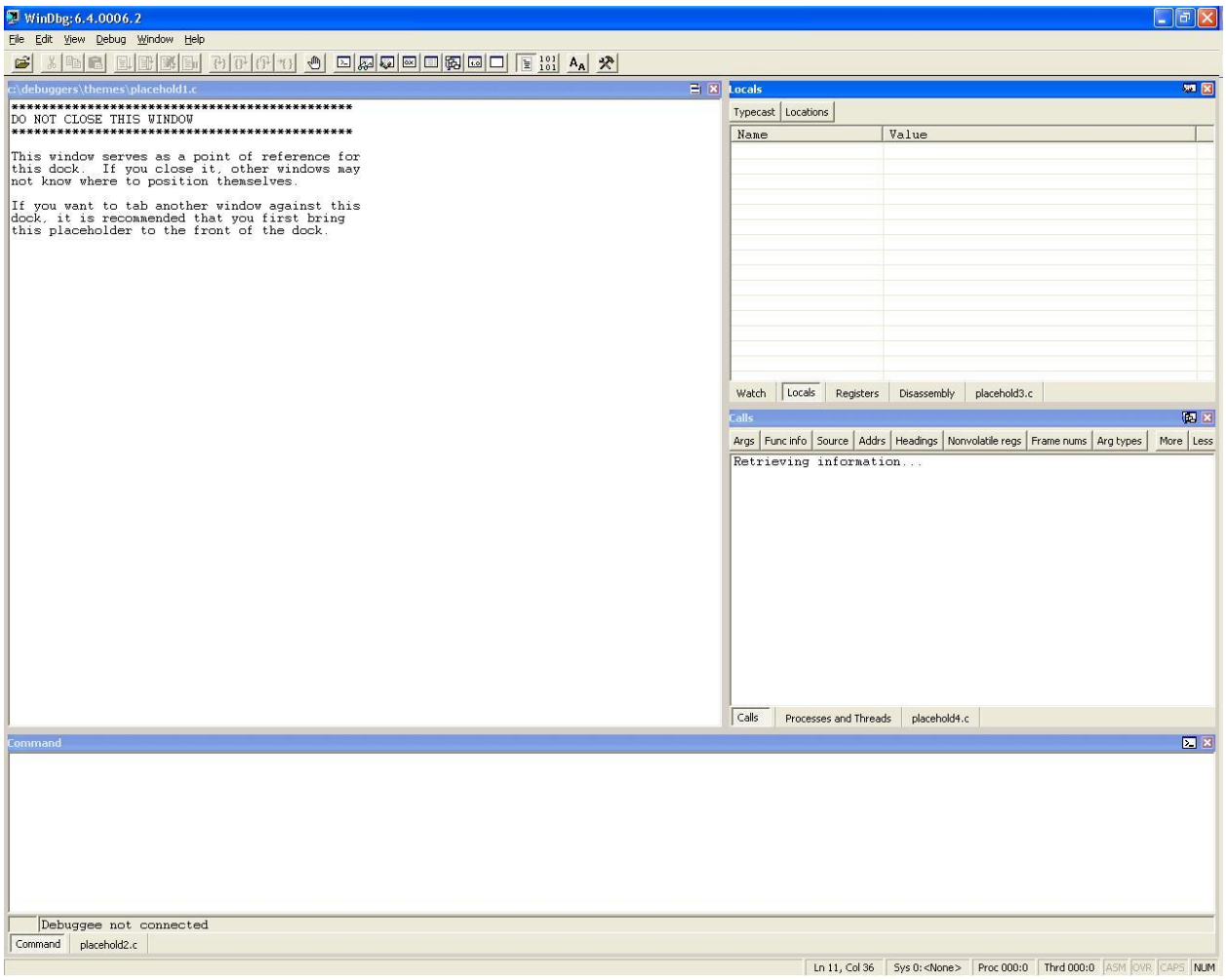
项目 • 2023/06/15

本主题显示 Windows 调试工具中提供的四个主题中每个主题的配置屏幕截图。 这些主题包括 Standard.reg、Standardvs.reg、Srcdisassembly.reg 和 Multimon.reg。

Standard.reg

Standard.reg 主题可用于大多数调试目的。 在此排列方式中，WinDbg 窗口的下三分之一由调试器命令窗口占用。 高三分之二大致分成两半。 左半部分由一个占位符窗口占用，该占位符窗口指示源窗口在选项卡式集合中打开的位置。 右半部分进一步分为垂直的两半。 上半部分包含一个选项卡式集合，其中包括“监视”、“局部变量”、“寄存器”和“反汇编”窗口。 下半部分包含一个选项卡式集合，其中包括“调用”、“进程”和“线程”窗口。

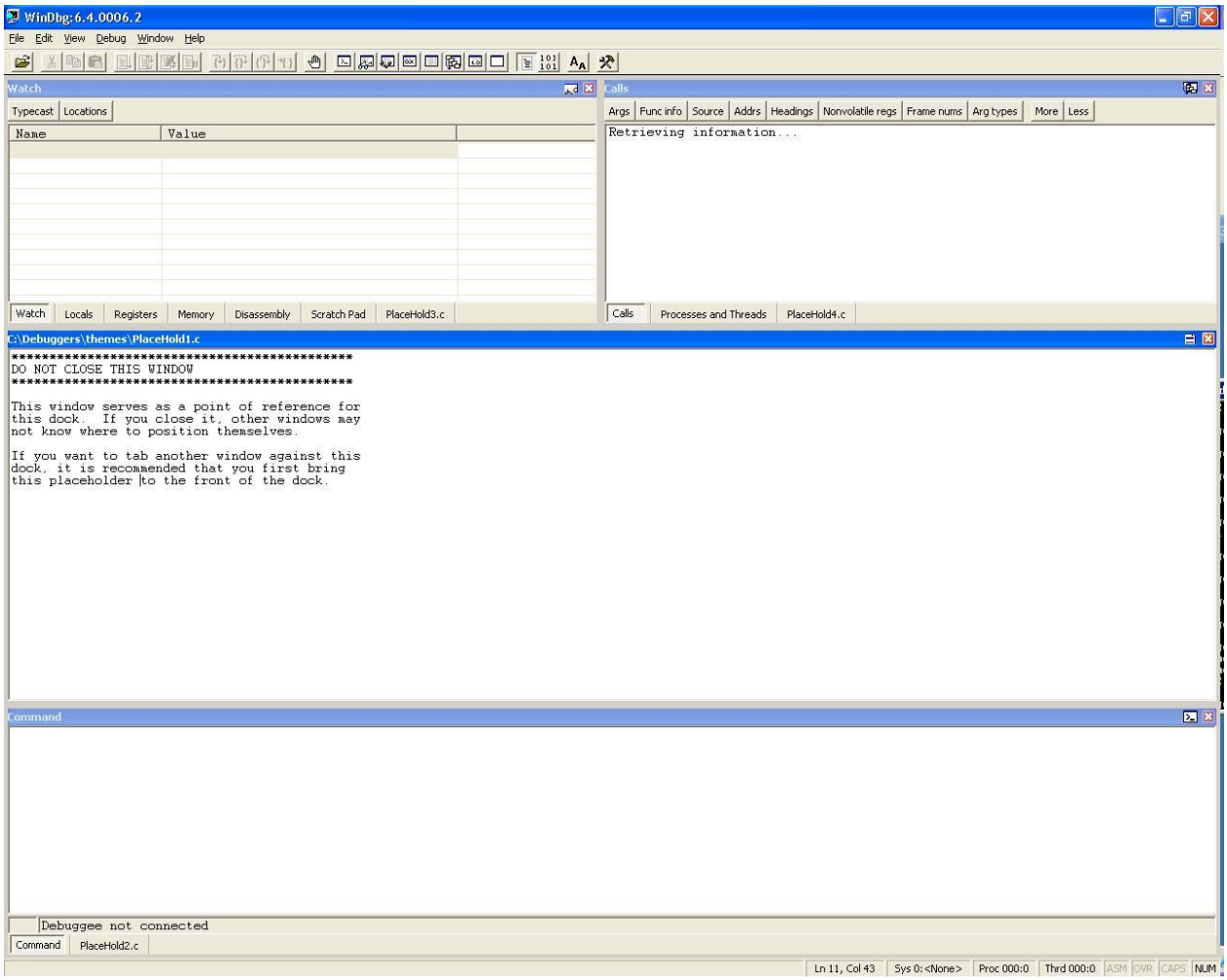
在每个停靠位置中，一个占位符窗口也作为其他窗口的参考点包含在内。 不应关闭占位符窗口，因为关闭它们可能会更改窗口的配置。 此排列方式中的所有窗口都停靠在一起。以下屏幕截图显示了 Standard.reg 主题。



Standardvs.reg

Standardvs.reg 主题可用于大多数调试目的，但在布局上与 Visual Studio 更相似。 在此排列方式中，WinDbg 窗口水平划分为三分之一。 上三分之一被进一步垂直划分为两半。 上半部分的左半部分包含一个选项卡式集合，其中包括 Watch、Locals、Registers、Memory、Disassembly 和 Scratchpad 窗口。 上半部分包含一个选项卡式集合，其中包含“调用”、“进程”和“线程”窗口。 WinDbg 窗口的下三分之一由调试器命令窗口占用。 中间第三个窗口由一个占位符窗口填充，该占位符窗口指示源窗口在选项卡式集合中的打开位置。

在每个停靠位置中，一个占位符窗口也作为其他窗口的参考点包含在内。 不应关闭占位符窗口，因为关闭它们可能会更改窗口的配置。 此排列方式中的所有窗口都停靠在一起。 以下屏幕截图显示了 Standardvs.reg 主题。

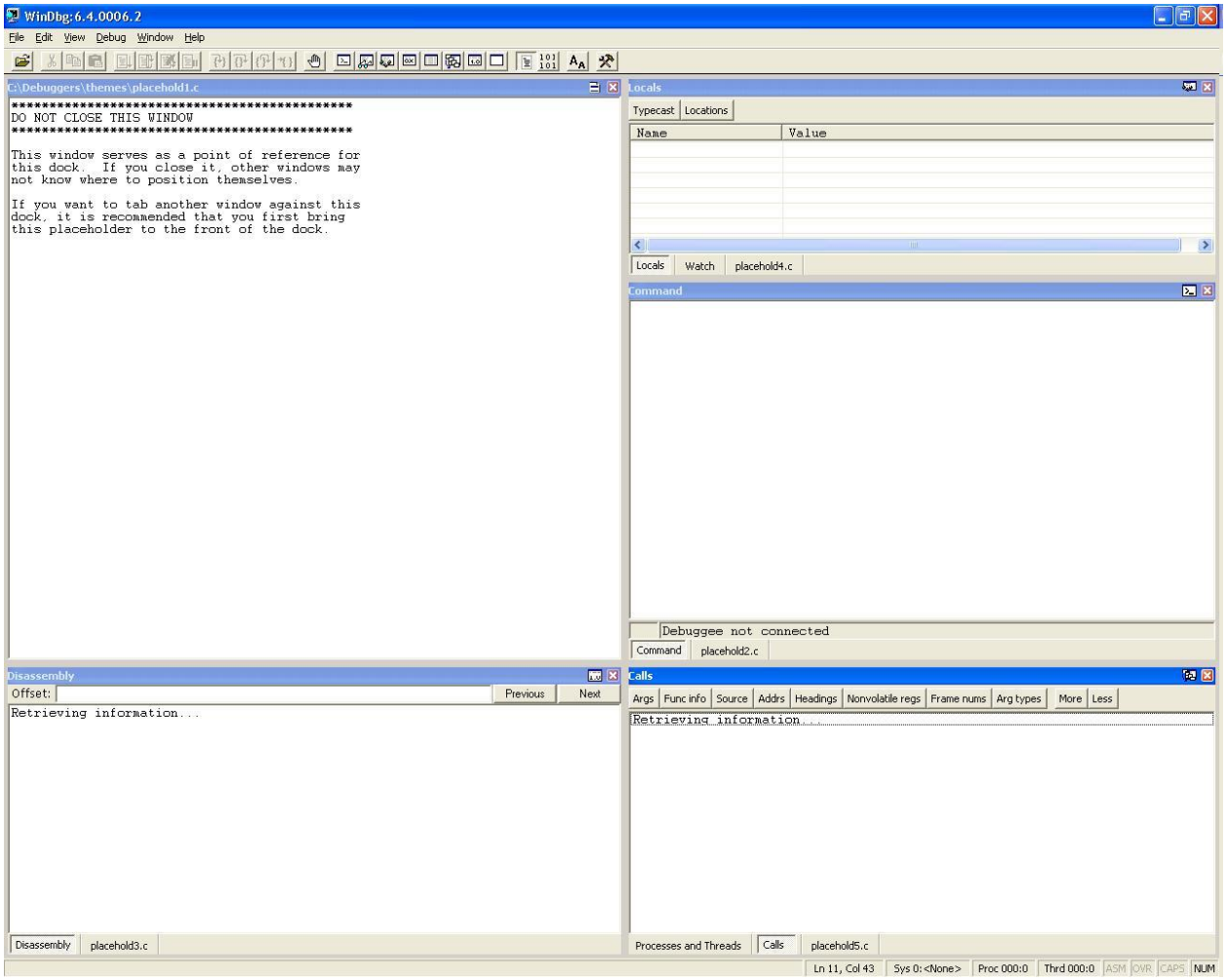


Srcdisassembly.reg

Srcdisassembly.reg 主题包含一个反汇编窗口，用于在程序集模式下进行调试。 在此排列方式中，WinDbg 窗口垂直划分为一半，形成的每一半进一步水平划分为三分之一。 在右半部分，上三个是“局部变量”和“监视”窗口的选项卡式集合，中间三个是“调试器命令”窗口，下三个是“进程”和“线程”和“调用”窗口的选项卡式集合。 在左半部分，上三分之二

由占位符窗口获取，该占位符窗口指示源窗口在选项卡式集合中的打开位置;下三分之一由“反汇编”窗口占用。

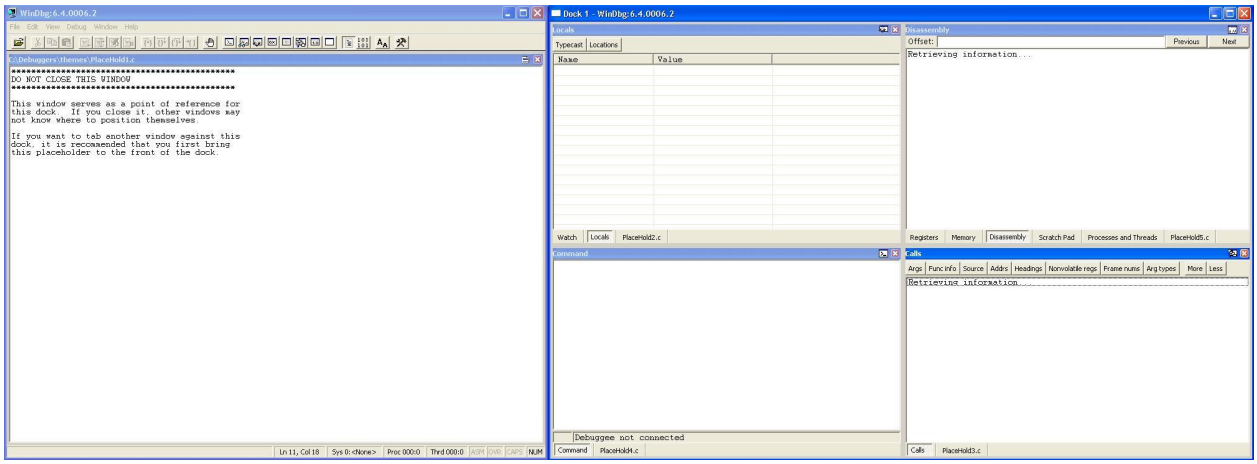
在每个停靠位置中，一个占位符窗口也作为其他窗口的参考点包含在内。 不应关闭占位符窗口，因为关闭它们可能会更改窗口的配置。 此排列方式中的所有窗口都停靠在一起。 以下屏幕截图显示了 Srcdisassembly.reg 主题。



Multimon.reg

Multimon.reg 主题设置为使用多个监视器进行调试。 在此排列方式中，将创建一个新扩展坞，以便可以在一台监视器上查看 WinDbg 窗口，并在另一台监视器上查看新扩展坞。 WinDbg 窗口由一个占位符窗口填充，该占位符窗口指示源窗口在选项卡式集合中的打开位置。 新扩展坞分为四个扩展坞。 左上角包含一个选项卡式集合，其中包括“监视”和“局部变量”窗口。 右上角包含选项卡式集合，其中包含 Registers、Memory、Disassembly、Scratchpad 以及“进程和线程”窗口。 左下角包含“调试器命令”窗口。 右下角包含“调用”窗口。

在每个停靠位置中，一个占位符窗口也作为其他窗口的参考点包含在内。 不应关闭占位符窗口，因为关闭它们可能会更改窗口的配置。 此排列方式中的所有窗口都停靠在一起。 以下屏幕截图显示了 Multimon.reg 主题。


WinDbg:6.4.0006.2
File Edit View Debug Window Help
C:\Debuggers\themes\PlaceHold1.c
****
DO NOT CLOSE THIS WINDOW
****
This window serves as a point of reference for
this dock. If you close it, other windows may
not know where to position themselves.
If you want to tab another window against this
dock, it is reccomended that you first bring
this placeholder to the front of the dock.
Ln 11, Col 18
Sys 0:<None>
Proc 000:0
Thrd 000:0
ASM
OVR
CAPS
NUM
Dock 1 - WinDbg:6.4.0006.2
Locals
Typecast
Locations
Name
Value
Watch
Locals
PlaceHold2.c
Disassembly
Offset:
Previous
Next
Retrieving information...
Registers
Memory
Disassembly
Scratch Pad
Processes and Threads
PlaceHold5.c
Command
Debuggee not connected
Command
PlaceHold4.c
Calls
Args
Func Info
Source
Addrs
Headings
Nonvolatile regs
Frame nums
Arg types
More
Less
Retrieving information...
Calls
PlaceHold3.c

使用工具栏和状态栏 - WinDbg (经典)

项目 • 2023/11/15

*工具栏*显示在菜单栏下方的 WinDbg 窗口顶部附近。 *状态栏*显示在 WinDbg 窗口的底部。

使用工具栏

以下屏幕截图显示了 WinDbg 工具栏。

工具栏按钮具有各种效果。 其中大多数等效于菜单命令。 若要执行与工具栏按钮关联的命令，请单击工具栏按钮。 如果无法使用按钮，则它看起来不可用。

有关每个按钮的详细信息，请参阅 工具栏按钮 - WinDBg (经典) 。

使用状态栏

以下屏幕截图显示了 WinDbg 状态栏。

下表描述了 WinDbg 状态栏的各个部分。

部分	说明
消息	显示来自调试器的消息。
Ln、Col	在活动 “源”窗口中的光标处显示行号和列号。
Sys	显示要调试的系统的内部十进制数，后跟其计算机名称 (或 **<Local>** （如果它与调试器在) 上运行的计算机相同）。
Proc	显示要调试的进程的内部十进制数，后跟其十六进制进程 ID。
Thrd	显示要调试的线程的内部十进制数，后跟其十六进制线程 ID。
ASM	指示 WinDbg 处于程序集模式。 如果 ASM 不可用，则 WinDbg 处于源模式。

部分	说明
OVR	指示 overtype 模式处于活动状态。 如果 OVR 不可用，则插入模式处于活动状态。
帽	指示 CAPS LOCK 处于打开状态。
Num	指示 NUM LOCK 处于打开状态。

隐藏工具栏或状态栏

若要显示或隐藏工具栏，请在“**视图**”菜单上选择或清除“**工具栏**”。 若要显示或隐藏状态栏，请在“**视图**”菜单上选择或清除“**状态栏**”。

如果隐藏工具栏或状态栏，则 WinDbg 显示区域中有更多的空间用于调试信息窗口。

设置窗口标题

可以使用 **.wtitle (Set Window Title)** 命令更改 WinDbg 窗口的标题。

工具栏按钮 - WinDbg (经典)

项目 • 2023/11/15

除断点按钮外，工具栏上的每个按钮都等效于菜单命令。 工具栏上的按钮具有以下效果。

Button	说明
	将源文件作为只读文件打开。 等效于文件 > 开放源代码文件。
	从活动窗口中删除所选文本并将其放在剪贴板上。 等效于编辑 > 剪切。
	将所选文本从活动窗口复制到剪贴板。 编辑 > 副本。
	将剪贴板上的文本粘贴到光标所在的位置。 等效于编辑 > 粘贴。
	开始或恢复执行。 执行将继续，直到到达断点、发生异常或事件、进程结束或调试器中断目标。 等效于 Debug > Go。
	在进程开始时重新启动执行。 等效于调试 > 重启。
	停止执行并永久终止目标进程。 等效于调试 > 停止调试。
	在用户模式下，此按钮停止进程及其线程。 在内核模式下，此按钮会进入目标计算机。 控件将返回到调试器。 此按钮还可用于切断长调试器命令窗口的显示。 等效于调试 > 中断。
	执行单个指令。 如果指令是函数调用，则调试器将进入 函数。 等效于调试 > 单步执行。
	执行单个指令。 如果指令是函数调用，则调试器一步执行整个函数。 等效于调试 > 单步执行。
	执行当前函数的其余部分，并在函数返回完成时中断。 等效于调试 > 单步执行。
	执行从当前指令到活动反汇编窗口或源窗口中标记的指令的所有指令。 等效于调试 > 运行到游标。
	如果活动窗口是“源”或“反汇编”窗口：在当前行处插入断点。 (如果当前行中已设置了断点，此按钮将删除断点。) 否则：打开断点对话框，如编辑 > 断点。
	打开或激活调试器命令窗口。 等效于查看 > 命令。
	打开或激活“监视”窗口。 查看 > 监视。
	打开或激活“局部变量”窗口。 等效于查看 > 局部变量。

Button	说明
	打开或激活“寄存器”窗口。 等效于 View > Registers。
	打开新的“内存”窗口。 等效于查看 > 内存。
	打开或激活“呼叫”窗口。 等效于查看 > 调用堆栈。
	打开或激活反汇编窗口。 等效于查看 > 反汇编。
	打开或激活 Scratch Pad。 等效于 View > Scratch Pad。
	在源模式调试和程序集模式调试之间切换。 等效于选择或清除“调试 > 源模式”。
	使你能够更改调试信息窗口中使用的字体。 等效于视图 > 字体。
	显示“选项”对话框。 等效于视图 > 选项。

另请参阅

调试器命令窗口

键盘快捷方式

项目 • 2024/02/06

可以使用以下键盘快捷方式在窗口之间切换。 有关如何在窗口之间移动的详细信息，请参阅 定位窗口。

展开表

键	效果
Ctrl+Tab	在调试信息窗口之间切换。 通过重复使用此键，可以扫描所有窗口，无论它们是浮动窗口、自行停靠窗口，还是停靠窗口的选项卡式集合的一部分。
Alt+Tab	在当前桌面上的窗口之间切换。 还可以使用此键盘快捷方式在 WinDbg 帧和已创建的任何其他扩展坞之间切换。

可以使用以下键盘快捷方式而不是鼠标来选择菜单命令。

展开表

键	等效菜单
F1	帮助 | 内容
F3	编辑 | 查找下一个
Shift+F3	与编辑相同 |“查找下一步”，只不过搜索是反向执行的。
Alt+F4	文件 | 退出
Ctrl+F4	文件 | 关闭当前窗口
F5	调试 | 转到
Shift+F5	调试 | 停止调试
Ctrl+Shift+F5	调试 | 重启
F6	文件 | 附加到进程
F7	调试 | 运行到光标处
F8	调试 | 跳入
F9	如果活动窗口是 “**源**” 或 “**反汇编** ”窗口：在当前行处插入断点。 如果已在当前行中设置了断点，则此键将删除该断点。 否则：打开 “**断点** ”对话框，如“编辑” |断点。

键	等效菜单
Alt+F9	编辑 | 断点
F10	调试 | 逐步运行
Ctrl+F10	调试 | 运行到光标处
F11	调试 | 跳入
Shift+F11	调试 | 跳出
Alt+1	“调试器命令”窗口，与“视图”相同 |命令。
Alt+Shift+1	关闭 “**命令**” 窗口。
Alt+2	打开“ **监视** ”窗口，与“视图 | ”相同”看。
Alt+Shift+2	关闭“ **监视** ”窗口
Alt+3	局部变量窗口，与视图相同 |当地人。
Alt+Shift+3	关闭“ **局部变量** ”窗口。
Alt+4	打开“ 寄存器”窗口，与“视图 | ”相同”寄存 器。
Alt+Shift+4	关闭“ **寄存器**” 窗口。
Alt+5	内存窗口，与视图相同 |记忆。
Alt+Shift+5	关闭“ **内存**” 窗口。
Alt+6	打开“ 调用”窗口，与“视图 | ”相同”调用堆栈。
Alt+Shift+6	关闭“ **调用**” 窗口
Alt+7	打开 反汇编窗口，与视图相同 |拆卸。
Alt+Shift+7	关闭 **反汇编** 窗口。
Alt+8	打开 “**暂存板**”，与“视图”相同 |划痕板。
Alt+Shift+8	关闭 **暂存板。**
Alt+9	打开“ 进程和线程”窗口，与“视图”相同 |进程和线程。
Alt+Shift+9	关闭“ **进程和线程**” 窗口。
Ctrl+A	编辑 | 全选
Ctrl+C	编辑 | 复制
Ctrl+D	文件 | 打开故障转储

键	等效菜单
Ctrl+E	文件 | 打开可执行文件
Ctrl+F	编辑 | 查找
Ctrl+G	编辑 | 转到地址
Ctrl+I	文件 | 映像文件路径
Ctrl+Shift+I	编辑 | 设置当前指令
Ctrl+K	文件 | 内核调试
Ctrl+L	编辑 | 转到行
Ctrl+O	文件 | 打开源文件
Ctrl+P	文件 | 源文件路径
Ctrl+R	文件 | 连接到远程会话
Ctrl+S	文件 | 符号文件路径
Ctrl+V	编辑 | 粘贴
Ctrl+Shift+V	编辑 | 评估所选内容
Ctrl+W	文件 | 打开工作区
Ctrl+X	编辑 | 剪切
Ctrl+Shift+Y	编辑 | 显示所选类型
Alt+* (数字键盘)	编辑 | 转到当前指令
Shift+Delete	编辑 | 剪切
Shift+Insert	编辑 | 粘贴
Ctrl+Insert	编辑 | 复制
Ctrl+Break	调试 | 中断
Alt+Delete	调试 | 中断

以下快捷键等效于 KD/CDB 控制键。

展开表

键	等效菜单	KD/CDB 控制键
Ctrl+Alt+A	调试 | 内核连接 | 循环波特率	Ctrl+A
Ctrl+Alt+D		Ctrl+D (切换调试信息)
Ctrl+Alt+K	调试 | 内核连接 | 循环初始中断	Ctrl+K
Ctrl+Alt+R	调试 | 内核连接 | 重新同步	Ctrl+R
Ctrl+Alt+V	视图 | 详细输出	Ctrl+V
Ctrl+Alt+W	视图 | 显示版本	Ctrl+W

可以使用以下键盘快捷方式在大多数调试信息窗口中移动插入点 (^) 。

展开表

脱字符号移动	键
左一个字符	向左键
右一个字符	向右键
左Word	Ctrl+向左键
Word右	Ctrl+向右键
向上移动一行	向上键
向下移动一行	向下键
向上翻页	向上翻页
向下翻页	向下翻页
当前行的开头	主页
行尾	结束
文件开头	Ctrl+Home
文件末尾	Ctrl+End

在 “调试器命令”窗口中，向上键和向下键浏览命令历史记录。 可以使用 **Insert** 键打开和关闭插入模式。

使用以下键盘快捷方式选择文本。

展开表

Select	键
当前位置左侧的字符	Shift+向左键
当前位置右侧的字符	Shift+向右键
Word当前位置左侧	Shift+Ctrl+向左键
当前位置右侧的Word	Shift+Ctrl+向右键
当前行	如果插入点位于第 1 列，则为 Shift+向下键
当前位置上方的行	如果插入点位于第 1 列，则为 Shift+向上键
到行尾	Shift+End
到行的开头	Shift+Home
屏幕向上	Shift+Page up
屏幕关闭	Shift+Page down
到文件的开头	Shift+Ctrl+Home
到文件结尾	Shift+Ctrl+End

使用以下键盘快捷方式删除文本。

展开表

删除	密钥
插入点右侧的字符	删除
插入点左侧的字符	Backspace
所选文本	删除

使用帮助文档 - WinDbg (经典)

项目 • 2023/10/26

你正在阅读的 WinDbg 帮助文档是 Windows 调试工具包的文档。

本文档使用随 Microsoft HTML 帮助 (hh.exe) 提供的查看器。 此查看器提供目录和索引，使你能够搜索文档、标记收藏夹或常用主题，以及打印一个或多个主题。

本主题中的以下部分介绍 帮助文档的功能和使用方法。

“内容”选项卡

“ **内容** ”选项卡提供文档目录的可展开视图。

单击节点旁边的加号 (+)，或双击节点的标题以展开或收缩该节点下的目录。

“索引”选项卡

“ **索引** ”选项卡显示本文档中使用的术语的完整索引。 可以输入术语的开头或使用滚动条浏览此索引。

“搜索”选项卡

在“ **搜索** ”选项卡中，可以搜索文档中包含的任何单词或短语。 可以搜索所有文本或将搜索限制为主题标题。

若要搜索包含多个单词的短语，请将短语括在引号中。 可以使用 AND、OR 和 NOT 运算符连接多个字词和短语。 默认运算符为 AND。

请注意，“AND NOT”无效。 若要搜索包含 *x* 而不是 *y* 的主题，请使用“*x* NOT *y*”。 还可以在搜索中使用 NEAR。

还允许使用通配符。 使用问号 (**?**) 表示任何单个字符，使用星号 (*) 表示零个或多个字符。 但是，不能在带引号的字符串中使用通配符。

所有字母和数字均按字面处理，但不允许在搜索中使用某些符号。

搜索返回与指定条件匹配的所有主题的列表。 如果选择“ **搜索以前的结果** ”框，则可以在这些结果中搜索更多字词。

“收藏夹”选项卡

在“ **收藏夹** ”选项卡中，可以保存经常访问的主题的标题。 查看主题时，单击“ **收藏夹** ”选项卡，然后单击“ **添加** ”按钮。

若要重命名收藏夹列表中的页面，请缓慢单击其名称两次，然后重新键入该名称。 若要查看“收藏夹”列表中的某个主题，请双击其名称或单击一次，然后单击“ **显示**”。 若要从收藏夹列表中删除某个主题，请单击该主题一次，然后单击“ **删除**”。

打印主题

若要打印一个或多个主题，请单击“ **内容** ”选项卡中的主题，然后单击工具栏上的“ **打印** ”按钮。 系统将询问您是仅打印所选主题还是该主题及其所有子主题。

还可以通过在视图窗口中右键单击，然后单击快捷菜单上的“ **打印** ”来打印正在查看的主题。 但是，此方法无法打印子主题。

在主题中搜索

若要在正在查看的主题中搜索文本字符串，请按 Ctrl+F，或在“**编辑**”菜单上单击“**在本主题中查找**”。

访问帮助文档

若要打开此帮助文档，请执行下列操作之一：

- 单击“ **开始**”，指向 “**所有程序**”，指向 “**Windows 调试工具**”，然后单击“ **调试帮助**”。
- 打开 Windows 资源管理器，找到 Debugger.chm 文件，然后双击它。
- 在命令提示符下，浏览到包含 Debugger.chm 的文件夹并运行 **hh debugger.chm**。
- 在任何 Windows 调试器中，使用 **.hh (打开 HTML 帮助文件)** 命令。
- 在 WinDbg 中，单击“**帮助**”菜单上的“**内容**”。 此命令打开“帮助”文档并打开“ **内容** ”选项卡。
- 在 WinDbg 中，单击“**帮助**”菜单上的“**索引**”。 此命令打开“帮助”文档并打开“ **索引** ”选项卡。
- 在 WinDbg 中，单击“**帮助**”菜单上的“**搜索**”。 此命令将打开“帮助”文档并打开“ **搜索** ”选项卡。
- WinDbg 中的许多对话框都有 “**帮助**” 按钮。 单击“ **帮助** ”打开此文档并打开相关页面。

Windows 调试工具：Windows 10新增功能 - WinDbg (经典)

项目 • 2023/10/26

更新了 WinDbg 信息

有关 Windows 调试工具的最新消息，请参阅 WinDbg - 发行说明。

Windows 10版本 1703

本部分介绍 Windows 10 版本 1703 中的新调试工具。

- 八个新的 JavaScript 主题，包括 JavaScript 调试器脚本
- 汇报 **dx (Display Debugger 对象模型表达式)** 命令，以包含新的命令功能。
- 新的 **dtx (显示类型 - 扩展调试器对象模型信息)** 命令。
- 新建 **！ioctldecode** 命令。
- 更新了 调试器引擎参考 以包含其他接口和结构。
- 汇报配置 tools.ini，以记录命令行调试器的其他选项。
- 在 Bug 检查代码参考中发布了 75 个以前未记录的停止代码。
- Windows 10主题中新的支持用于网络内核调试的以太网 NIC。

Windows 10，版本 1607

本部分介绍 Windows 10 版本 1607 中的新调试工具。

- 有关 使用 WinDbg 调试 UWP 应用的新主题。
- 汇报 Bug 检查代码参考中的 30 个最常查看的开发人员 bug 检查主题。

Windows 10

- **.settings (设置调试设置)** - 用于在 Debugger.Settings 命名空间中设置、修改、显示、加载和保存设置的新命令。
- **dx (显示 NatVis 表达式)** - 介绍新的 dx 调试器命令，该命令使用 NatVis 扩展模型和 LINQ 支持显示对象信息。
- 在调试器环境中处理 NatVis 可视化文件的新命令。
  - **.nvlist（NatVis 列出）**
  - **.nvload（NatVis 加载）**
  - **.nvload（NatVis 卸载）**

    - **.nvunloadall（NatVis 全部卸载）**
- 蓝牙扩展 (Bthkd.dll)
- 存储内核调试程序扩展
- 新的 Symproxy 信息，包括 SymProxy 自动安装。 此外，还更新了以下主题，以涵盖新的 SymProxy 功能：
    - HTTP 符号存储
    - SymProxy
    - 安装 SymProxy
    - 配置注册表
    - 为 SymProxy 配置 IIS
- **CDB Command-Line 选项** - 更新为包含新的命令行选项。
- **！analyze** - 更新为包含有关将此扩展与 UMDF 2.15 配合使用的信息。
- **！wdfkd.wdfcrashdump** - 更新为包含有关将此扩展与 UMDF 2.15 配合使用的信息
- **！irp** - 已更新。 从 Windows 10命令输出中会显示 IRP 主要代码文本和次要代码文本。
- 使用调试器标记语言 - 更新为描述调试器标记语言 (DML) 中可用的新的选择保留 (或右键单击) 行为。
- 使用 Windows 调试器 (WinDbg) 进行故障转储分析 - 通过 KDNET 进行内存转储的性能有所提高。
- 调试通用驱动程序 - 分步实验室 (回显内核模式) - 新的分步实验室，演示如何使用 WinDbg 调试示例 KMDF 回显驱动程序。

想要下载调试工具?

有关下载调试工具的信息，请参阅 Windows 调试工具。

Windows 调试工具：Windows 8.1 的新增功能

项目 • 2023/10/26

对于Windows 8.1，适用于 Windows 的调试工具包括这些新功能。

- GPIO 扩展
- HID 扩展
- 大多数 内核模式驱动程序框架扩展命令 现在适用于 UMDF 2 和 KMDF。 (某些专门支持 UMDF 2 的命令（例如 **！ wdfkd.wdfumdevstacks**）) 已添加到此集中。
- 分页文件现可与内存转储文件一起包含在 CAB 文件中。 请参阅 包含分页文件和内存转储的 CAB 文件

另请参阅

Windows 调试

适用于 Windows 的调试工具：适用于 Windows 8 的新增功能

项目 • 2024/10/31

从 Windows 8 开始，你可以从 Microsoft visual Studio 中开发、生成和调试内核模式和用户模式组件。 添加了对通过网络连接或 USB 3.0 连接进行调试的支持，并改进了对调试优化代码和内联函数的支持。 有关详细信息，请参阅以下主题：

- 手动设置网络连接
- 设置 USB 3.0 xHCI-DBC 内核模式调试 （KDUSB）
- 调试优化代码和内联函数

包括两组新的调试器扩展命令：

- USB 3.0 扩展
- RCDRKD 扩展

除了 Visual Studio，还可以将 Windows 调试器用于调试 Windows 应用。 适用于 Windows 的调试工具包包括 **PLMDebug.exe**，可用于手动控制暂停、恢复、调试和终止 Windows 应用。

Sos.dll 是用于调试托管代码的组件。 适用于 Windows 包的 Windows 8 调试工具不包括 sos.dll。 有关如何获取sos.dll的信息，请参阅*调试托管代码中的 SOS 调试扩展（sos.dll）*。

另请参阅

Windows 调试

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

适用于Windows 8发行说明的调试工具

项目 • 2023/04/14

Visual Studio 中用于 1394 调试的通道号

如果使用 Microsoft Visual Studio 通过 1394 电缆执行内核模式调试，请将通道设置为介于 1 和 62（含 1 和 62）之间的十进制整数。 首次设置调试时，不要将通道设置为 0。 由于默认通道值为 0，因此软件假定没有更改并且不会更新设置。 如果必须使用通道 0，请先使用备用通道 (1 到 62)，然后切换到通道 0。

默认情况下，内联函数调试处于打开状态

在 Windows 8 中，默认启用内联函数的调试。 命令 **.inline 0** 关闭内联函数调试，命令 **.inline 1** 打开内联函数调试。

网络调试的配置页中的端口号无效

问题： 如果在内核模式网络调试的配置页中输入了无效的端口号，则配置会成功，但主计算机上的调试器无法连接到目标计算机。

解决 方案： 确保端口号有效。 有效的端口号范围为 49152–65535。 此外，你的公司可能对哪些端口可用于网络调试有限制。 若要确保输入有效的端口号，请检查内部 IP 安全策略。

在命令行中使用 .remote 工具

问题： 在命令行中使用 **.remote** 工具会使接口崩溃，因为它使用 npipe 创建旧样式 remote.exe。

解决 方案： 请改用 **.server** 命令。

Visual Studio 的设计功能

- 计算机的自动预配包括用户模式和内核模式引导，无论选择哪一个。 这需要两次重启才能进行预配，并且需要 8-20 分钟。
- 支持一次仅附加一个进程。
- 在 Visual Studio 的 Windows 调试器扩展的调试会话期间，异常使用命令行进行管理。

全局设计功能

- 用户必须在提升的上下文中运行才能安装 USB 3.0 XHCI 筛选器驱动程序。 如果用户未运行提升，PnP 管理器将返回一条错误消息，该错误消息不会通知用户提升是问题。
- 如果启用了内核调试，则当设备仍处于打开状态时，不应从系统中删除用于内核调试的设备。 如果设备被删除，系统将挂起，需要重启。

适用于 Windows 的 Windows 7 调试工具

项目 • 2023/04/14

若要调试在 Windows Vista 或 Windows Server 2008 上运行的代码，请获取适用于 Windows 7 和 .NET Framework 4.0 的 Microsoft Windows 软件开发工具包 (SDK) 中包含的 Windows 7 调试工具包。 如果只想下载适用于 Windows 的调试工具，请安装 SDK，并在安装过程中，选中“ **适用于 Windows 的调试工具** ”框并清除所有其他框。

另请参阅

下载并安装 WinDbg Windows 调试器

调试 Windows Vista

项目 • 2023/06/15

若要使用 WinDbg 调试 Windows Vista，请获取适用于 Windows 7 和 .NET Framework 4.0 的 Microsoft Windows 软件开发工具包 (SDK) 中包含的 Windows 7 调试工具包。

如果只想下载适用于 Windows 的调试工具，请安装 SDK，并在安装过程中选择“**Windows 调试工具**”框并清除所有其他框。

注意在安装 SDK 之前，可能需要卸载 Microsoft Visual C++ 2010 可再发行组件。

Windows Vista (WinDbg、KD、CDB、NTSD) 调试工具

适用于 Windows 的 Windows 7 调试工具可以在基于 x86 或基于 x64 的处理器上运行，并且可以调试在基于 x86 或基于 x64 的处理器上运行的代码。 有时候，调试程序和要调试的代码运行在同一计算机上，但另外一些时候，调试程序和要调试的代码则运行在不同的计算机上。 不管哪一种情况，运行调试程序的计算机称为“主计算机”，被调试的计算机称为“目标计算机”。 当目标计算机运行其中一个操作系统时，请使用适用于 Windows 的 Windows 7 调试工具。

Windows Vista：Windows Server 2008

如果目标计算机运行的是最新版本的 Windows，请获取适用于 Windows 的当前 调试工具。

调试方法

项目 • 2023/06/15

本部分讨论两种类型的调试技术：标准和专用。 标准技术适用于大多数调试方案，示例包括设置断点、检查调用堆栈和查找内存泄漏。 专用技术适用于特定技术或代码类型，示例即插即用调试、内核模式驱动程序框架调试和 RPC 调试。

可以在以下部分了解详细信息。

标准调试方法

专业调试方法

标准调试方法

项目 • 2023/06/15

本部分讨论可应用于不同技术和不同类型的代码的标准调试技术。

本部分中的内容

- 使用断点
- 读取和写入内存
- 使用 !analyze 扩展
- 在启用驱动程序验证程序的情况下处理 Bug 检查
- 非侵入式调试（用户模式）
- 在程序集模式下调试
- 在源模式下调试
- 调试优化的代码和内联函数
- 使用 Windows 调试器调试托管代码
- 使用 Windows 调试器调试 Windows 应用
- 更改上下文
- 控制进程和线程
- 使用调试器标记语言
- 控制异常和事件
- 查找进程 ID
- 调试堆栈溢出
- 手动遍历堆栈
- 调试包含 JScript 帧的堆栈跟踪
- 调试应用程序错误
- 重新附加到目标应用程序
- 使目标计算机崩溃和重新启动目标计算机
- 与目标计算机同步
- 查找内存泄漏
- 调试超时
- 调试已停止的系统
- 调试多个目标
- 跟踪占用大量处理器资源的进程
- 确定对象的 ACL
- 显示关键节
- 调试死锁
- 调试失败的驱动程序卸载
- 读取 Bug 检查回调数据

- 使用 KD 调试用户模式故障
- 映射驱动程序文件
- 来自目标的消息

使用断点 - 调试技术

项目 • 2023/06/15

*断点*是可执行代码中操作系统停止执行并进入调试器的位置。 这使你可以分析目标并发出调试器命令。

本节包括下列主题：

断点控制方法

断点语法

无法解析的断点（bu 断点）

处理器断点（ba 断点）

初始断点

用户空间和系统空间

设置断点造成的风险

断点控制方法

项目 • 2023/06/27

在源模式下) 时，可以按虚拟地址、模块和例程偏移量或源文件和行号 (指定断点的位置。 如果在没有偏移量的例程上放置断点，则输入该例程时将激活该断点。

有多种其他类型的断点：

- 断点可以与某个线程相关联。
- 断点可以在触发地址之前启用固定数量的传递。
- 断点可以在触发时自动发出某些命令。
- 可以在非可执行内存上设置断点，并watch读取或写入该位置。

如果要在用户模式下调试多个进程，则断点集合取决于当前进程。 若要查看或更改进程的断点，必须将进程选为当前进程。 有关当前进程的详细信息，请参阅 控制进程和线程。

用于控制和显示断点的调试器命令

若要控制或显示断点，可以使用以下方法：

- 使用 bl (**断点列表**) 命令列出现有断点及其当前状态。
- 使用 .bpcmds (**显示断点命令**) 命令列出所有断点以及用于创建断点的命令。
- 使用 bp (**设置断点**) 命令设置新的断点。
- 使用 bu (**设置未解析断点**) 命令设置新的断点。 使用 **bu** 设置的断点称为未解析断点;它们具有与使用 **bp** 设置的断点不同的特征。 有关完整的详细信息，请参阅 未解决的断点 (bu 断点) 。
- 使用 bm (**设置符号断点**) 命令在与指定模式匹配的符号上设置新的断点。 如果包含 **/d** 开关，则具有 **bm** 的断点集将与地址 (关联，例如 **bp** 断点) ;如果未包含此开关，它将像 **bu** 断点) 一样 (解析。
- 使用 ba (Break on Access) 命令设置 *处理器断点*，也称为 *数据断点*。 在写入内存位置、读取内存位置、将其作为代码执行或发生内核 I/O 时，可以触发这些断点。 有关完整详细信息，请参阅 处理器断点 (ba 断点) 。
- 使用 bc (**断点清除**) 命令永久删除一个或多个断点。
- 使用 bd (**断点禁用**) 命令暂时禁用一个或多个断点。

- 使用 **be (断点启用)** 命令重新启用一个或多个禁用的断点。
- 使用 **br (断点重新编号)** 命令更改现有断点的 ID。
- 使用 **bs (更新断点命令)** 命令更改与现有断点关联的命令。
- 使用 **bsc (更新条件断点)** 命令更改发生现有条件断点的条件。

在 WinDbg 中，有几个用户界面元素有助于控制和显示断点。 请参阅 在 WinDbg (经典) 中设置断点 。

每个断点都有一个十进制数，称为与之关联的断点 ID。 此数字标识各种命令中的断点。

断点命令

可以在命中断点时自动执行的断点中包含命令。 例如，以下命令在 MyFunction+0x47 处中断，写入转储文件，然后继续执行。

```
dbgcmd

0:000> bu MyFunction+0x47 ".dump c:\mydump.dmp; g"
```

注意 如果要从内核调试器控制用户模式调试器，请不要在断点命令字符串中使用 **g (Go)** 。 串行接口可能无法跟上此命令的运行，并且你将无法重新进入 CDB。 有关这种情况的详细信息，请参阅 从内核调试器控制 User-Mode 调试器。

断点数

在内核模式下，最多可以使用 32 个软件断点。 在用户模式下，可以使用任意数量的软件断点。

支持的处理器断点数取决于目标处理器体系结构。

条件断点

可以设置仅在特定条件下触发的断点。 有关这些断点类型的详细信息，请参阅 设置条件断点。

不明确的断点

在版本 10.0.25310.1001 及更高版本的调试器引擎中，现在支持不明确的断点解析。 不明确的断点允许调试器在某些断点表达式解析到多个位置的情况下设置断点。 有关详细信息，请参阅 不明确的断点解析。

请参阅

使用断点

断点语法

bp、bu、bm（设置断点）

无法解析的断点（bu 断点）

断点语法

项目 • 2024/03/05

通过调试器命令窗口或 WinDbg 图形界面创建断点时，可以使用以下语法元素。

断点中的地址

断点支持多种地址语法，包括虚拟地址、函数偏移量和源行号。 例如，可以使用以下命令中的任何一个设置断点：

dbgcmd

```
0:000> bp 0040108c
0:000> bp main+5c
0:000> bp `source.c:31`
```

有关此语法的详细信息，请参阅数字表达式语法、源行语法和单个命令主题。

方法断点

如果要在 *MyClass* 类的 *MyMethod* 方法上放置断点，可以使用两种不同的语法：

- 在 MASM 表达式语法中，可以通过双冒号或双下划线来指示方法。

  dbgcmd

  ```
  0:000> bp MyClass::MyMethod
  0:000> bp MyClass__MyMethod
  ```

- 在 C++ 表达式语法中，必须通过双冒号来指示方法。

  dbgcmd

  ```
  0:000> bp @@( MyClass::MyMethod )
  ```

如果要使用更复杂的断点命令，应使用 MASM 表达式语法。 有关表达式语法的详细信息，请参阅计算表达式。

使用复杂的 MASM 表达式的断点

若要对复杂函数（包括包含空格的函数以及 C++ 公共类的成员）设置断点，请将表达式括在括号中。 例如，使用 **bp (??MyPublic)** 或 **bp (operator new)**。

更通用的技术是使用 @!"chars" 语法。 这是 MASM 计算器中的一种特殊转义，可用于提供任意文本进行符号解析。 必须以 @!" 这三个符号开头，最后加上引号 (")。 如果没有此语法，则不能在 MASM 计算器中的符号名称中使用空格、尖括号（<、 >）或其他特殊字符。 此语法专用于名称，而不是参数。 模板和重载是需要此引号表示法的符号的主要来源。 还可以使用 @!"chars" 语法设置 **bu** 命令，如下面的代码示例所示。

dbgcmd

```
0:000> bu @!"ExecutableName!std::pair<unsigned
int,std::basic_string<unsigned short,std::char_traits<unsigned
short>,std::allocator<unsigned short> > >::operator="
```

在此示例中，*ExecutableName* 是可执行文件的名称。

此转义语法对 C++ （例如重载运算符）比对 C 更有用，因为 C 函数名称中没有空格（或特殊字符）。 但是，由于 .NET Framework 中大量使用重载，此语法对于许多托管代码也很重要。

若要在 C++ 语法中的任意文本上设置断点，请使用 **bu @@c++ (***text***)** 作为与 C++ 兼容的符号。

脚本中的断点

断点 ID 不必显式引用。 相反，可以使用解析为对应于断点 ID 的整数的数字表达式。 若要指示应将表达式解释为断点，请使用以下语法。

dbgcmd

```
b?[Expression]
```

在此语法中需要方括号，*Expression* 代表解析为与断点 ID 对应的整数的任何数值表达式。

此语法允许调试器脚本以编程方式选择断点。 在以下示例中，断点会根据用户定义的伪寄存器的值而更改。

dbgcmd

```
b?[@$t0]
```

断点伪寄存器

如果要在表达式中引用断点地址，则可以将pseudo-register 与 **$bp** *Number* 语法一起使用，其中 *Number* 是断点 ID。 有关此语法的详细信息，请参阅伪寄存器语法。

不明确的断点解析

在调试器引擎的 10.0.25310.1001 版本及更高版本中，目前支持不明确的断点解析。 不明确的断点允许调试器在某些断点表达式解析为多个位置的特定情况下设置断点。 有关详细信息，请参阅不明确的断点解析。

另请参阅

使用断点

断点语法

bp、bu、bm（设置断点）

无法解析的断点（bu 断点）

无法解析的断点（bu 断点）

项目 • 2024/05/27

如果为未加载的例程名称设置了断点，则该断点称为*延迟*、*虚拟*或*未解析*断点。 （这些术语可以互换使用。）未解析的断点与模块的任何特定加载无关。 每次加载新程序时，都会检查该例程名称。 如果出现此例程，调试器就会计算虚拟断点的实际编码地址，同时启用断点。

如果使用 **bu** 命令设置断点，则该断点将自动被视为未解析。 如果此断点位于已加载的模块中，则该断点仍会启用并正常运行。 但是，如果随后卸载并重新加载模块，则此断点不会消失。 另一方面，使用 **bp** 设置的断点会被立即解析为一个地址。

bp 断点和 **bu** 断点之间存在三个主要差异：

- **bp** 断点位置始终转换为地址。 如果模块更改移动了设置 **bp** 断点的代码，断点仍会保留在相同地址。 另一方面，**bu** 断点仍与使用的符号值（通常是符号加偏移量）相关联，即使符号的地址发生更改，也会跟踪此符号位置。
- 如果在加载的模块中找到 **bp** 断点地址，并且该模块稍后卸载，则会从断点列表中删除断点。 另一方面，**bu** 断点在重复卸载和加载后仍然存在。
- 使用 **bp** 设置的断点不会保存在 WinDbg 工作区中。 使用 **bu** 设置的断点保存在工作区中。

控制地址断点和未解析断点

可以使用 **bp（设置断点）**命令或 **bm（设置符号断点）** 命令（如果包含 **/d** 开关）来创建地址断点。 可以使用 **bu（设置未解析断点）**命令或 **bm** 命令（如果不包含 **/d** 开关）来创建未解析断点。 禁用、启用和修改断点的命令适用于所有类型的断点。 显示断点列表的命令包括所有断点，并指示每个断点的类型。 有关这些命令的列表，请参阅控制断点的方法。

WinDbg 的“断点”对话框会显示所有断点，并用“u”表示未解析的断点。 此对话框可用于修改任何断点。 此对话框中的“命令”文本框可用于创建任何类型的断点；如果省略该类型，则会创建一个未解析的断点。 有关详细信息，请参阅**编辑 | 断点**。 在 WinDbg 的“反汇编窗口”或“源”窗口中使用鼠标设置断点时，调试器将创建一个未解析的断点。

反馈

此页面是否有帮助？ 是 否

处理器断点（ba 断点）

项目 • 2024/05/27

应调试器要求由处理器控制的断点称为*处理器断点*或*数据断点*。 由调试器直接控制的断点称为*软件断点*。

注意尽管术语*数据断点*通常用作*处理器断点*的同义词，但此术语可能会产生误导。 断点有两种基本类型：由处理器控制的处理器断点，以及由调试器控制的软件断点。 处理器断点通常设置在程序数据上，因此被称为“数据断点”，但也可以设置在可执行代码上。 软件断点通常在在可执行代码上设置，但也可以在程序数据上设置。 遗憾的是，调试文献中通常把处理器断点称为“数据断点”，即使它们是在可执行代码上设置的。

处理器断点

处理器断点在访问特定内存位置时触发。 处理器断点有四种类型，与触发断点的内存访问类型相对应：

展开表

断点类型	操作
e（执行）	当处理器从指定地址获取指令时触发。
r（读取/写入）	当处理器读取或写入指定地址的内存时触发。
w（写入）	当处理器写入指定地址的内存时触发。
i (i/o)	在访问指定地址的 I/O 端口时触发。

每个处理器断点都有与之关联的大小。 例如，可以在地址 0x70001008 处设置 **w**（写入）处理器断点，大小为 4 个字节。 这将监视从 0x70001008 到 0x7000100B （含）的地址块。 如果写入此内存块，将触发断点。

在处理器对与指定区域重叠与不完全相同的内存区域执行操作时，可能会发生此情况。 在上一段给出的示例中，包含 0x70001000 至 0x7000100F 范围的单个写入操作，或仅包含 0x70001009 位字节的写入操作，都属于重叠操作。 在这种情况下，是否触发断点取决于处理器。 有关特定处理器如何处理这种情况的详细信息，请查阅处理器架构手册，并查找“调试寄存器”或“调试控制寄存器”。 以一种特定的处理器类型为例，在 x86 处理器上，只要访问范围与断点范围重叠，就会触发读取或写入断点。

同样，如果在地址 0x00401003 上设置了 **e**（执行）断点，然后执行跨越地址 0x00401002 和 0x00401003 的双字节指令，结果将取决于处理器。 同样，有关详细信息，请参阅处理器体系结构手册。

处理器区分用户模式调试器设置的断点和内核模式调试器设置的断点。 用户模式处理器断点不会影响任何内核模式进程。 内核模式处理器断点可能会影响用户模式进程，具体取决于用户模式代码是否使用调试寄存器状态以及是否存在附加的用户模式调试器。

要将当前进程的现有数据断点应用于其他寄存器上下文，请使用 .apply_dbp（将数据断点应用到上下文）命令。

在多处理器计算机上，每个处理器断点都适用于所有处理器。 例如，如果当前处理器是 3，并且使用 `ba e1 MyAddress` 命令在 MyAddress 处设置了断点，那么在该地址执行的任何处理器（不仅仅是处理器 3）都会触发该断点。 软件断点也是如此。

软件断点

软件断点与处理器断点不同，它由调试器控制。 当调试器在某个位置设置软件断点时，它会用断点指令暂时替换该内存位置的内容。 调试器会记住该位置的原始内容，因此如果在调试器中显示该内存，调试器将显示该内存位置的原始内容，而不是中断指令。 当目标进程执行该位置的代码时，中断指令会使进程进入调试器中。 在执行了选择的任何操作后，可以使目标程序恢复执行，并以最初在该位置的指令继续执行。

处理器断点类型的可用性

i (i/o) 选项仅在内核模式调试时可用。

并非所有数据大小都能用于所有处理器断点类型。 允许的大小取决于目标计算机的处理器。 有关详细信息，请参阅 ba（访问时中断）。

软件断点和处理器断点的限制

使用 bp 或 bm /a 命令时，可以指定数据地址而不是程序地址。 但是，即使指定了数据位置，这些命令创建的也是软件断点，而不是处理器断点。 当调试器在某个位置设放置件断点时，它会用断点指令暂时替换该内存位置的内容。 这样不会损坏可执行映像，因为调试器会记住该位置的原始内容，当目标进程试图执行该代码时，调试器可以做出适当的响应。 但是，当软件断点设置在数据位置时，由此产生的覆盖会导致数据损坏。 因此，在数据位置设置软件断点只有在确定该位置只用作可执行代码时才是安全的。

bp、**bu** 和 **bm** 命令通过将处理器指令替换为中断指令来设置软件断点。 因此，这些代码不能用于只读代码或任何其他不能覆盖的代码。 要在此类代码中设置断点，必须使用 ba（访问时断开）和 **e**（执行）选项。

不能在同一地址创建多个处理器断点，这些断点仅在触发断点时自动执行的命令上有所不同。 但是，可以在同一地址创建多个断点，但这些断点的其他限制条件各不相同（例

如，可以使用 ba 命令在同一地址创建多个断点，但 /p、/t、/c 和 /C 选项的值各不相同）。

用户模式进程中的初始断点（通常设置在 main 函数或类似函数上）不能是处理器断点。

支持的处理器断点数取决于目标处理器体系结构。

控制软件断点和处理器断点

可以使用 bp（设置断点）、bm（设置符号断点）和 bu（设置未解析断点）命令来创建软件断点。 处理器断点可用 ba（访问时中断）命令来创建。 禁用、启用和修改断点的命令适用于所有类型的断点。 显示断点列表的命令包括所有断点，并指示每个断点的类型。 有关这些命令的列表，请参阅控制断点的方法。

WinDbg **断点** 对话框显示所有断点，指示表示法为“e”、“r”、“w”或“i'的处理器断点，后跟块的大小。 此对话框可用于修改任何断点。 此对话框中的**命令**文本框可用于创建任何类型的断点。如果需要处理器断点，请以“ba”开始输入。 在 WinDbg 反汇编窗口或源窗口中使用鼠标设置断点时，调试器将创建一个未解析的软件断点。

处理器断点存储在处理器的调试寄存器中。 可以通过手动编辑调试寄存器的值来设置断点，但强烈建议不要这样做。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

初始断点

项目 • 2023/10/26

当调试器启动新的目标应用程序时，在main映像之后自动发生初始断点，并在调用任何 DLL 初始化例程之前加载所有静态链接的 DLL。

当调试器附加到现有用户模式应用程序时，会立即发生初始 断点 。

-g 命令行选项会导致 WinDbg 或 CDB 忽略初始断点。 此时可以自动执行命令。 有关这种情况的详细信息，请参阅 控制异常和事件。

如果要启动新目标并在实际应用程序即将开始执行时中断目标，请不要使用 **-g** 选项。 而是让初始断点发生。 调试器处于活动状态后，在 **main** 或 **winmain** 例程上设置断点，然后使用 **g (Go)** 命令。 然后，所有初始化过程都会运行，当main应用程序即将开始执行时，应用程序将停止。

有关内核模式下的自动断点的详细信息，请参阅 崩溃并重新启动目标计算机。

请参阅

使用断点

断点语法

bp、bu、bm（设置断点）

不明确的断点解析

无法解析的断点（bu 断点）

不明确的断点解析

项目 • 2024/03/05

在调试器引擎的 10.0.25310.1001 版本及更高版本中，目前支持不明确的断点解析。

不明确的断点允许调试器在某些断点表达式解析为多个位置的特定情况下设置断点。 例如，在以下情况时可能会发生这种问题：

- 函数的多个重载。
- 有多个符号与断点表达式匹配。
- 同一个符号名称用于多个位置。
- 符号已内联。
- 在源窗口中具有多个实例的模板函数中设置断点。

启用后，调试器将在给定断点表达式的每个符号匹配上设置断点。 如果满足某些条件，调试器还将过滤符号匹配项。

有关使用断点的一般信息，请参阅使用断点。

启用不明确的断点解析

默认情况下，禁用不明确的断点。 若要在调试器会话中启用，请在 WinDbg 控制台中运行以下命令：

dbgcmd

```
dx
@$debuggerRootNamespace.Debugger.Settings.EngineInitialization.ResolveAmbigu
ousBreakpoints = true;
```

若要确认不明确的断点设置是否已处于活动状态：

dbgcmd

```
0:010> dx
@$debuggerRootNamespace.Debugger.Settings.EngineInitialization.ResolveAmbigu
ousBreakpoints
@$debuggerRootNamespace.Debugger.Settings.EngineInitialization.ResolveAmbigu
ousBreakpoints                 : true
```

有关使用 dx 命令的详细信息，请参阅 dx（显示调试器对象模型表达式）。

若要禁用该功能，请将上述值设置为 `false`。 若要确保设置在会话之间保留，请确保单击`File -> Settings -> Debugger Settings`，然后检查标记为`Persist engine settings across debugger sessions`的框。

用法适用于单个断点

解析不明确的断点表达式仅适用于运行断点命令以在调试器中设置单个断点。 换句话说，使用 `bm` 命令设置多个断点将继续像往常一样工作。 运行启用了此功能的命令将导致单个断点的新断点行为。

有关断点命令的常规信息，请参阅 bp、bu、bm（设置断点）。

分层断点

分层断点表示将不明确的断点表达式解析为多个断点的结果。 如果表达式生成两个或多个将用于设置断点的匹配项，则会创建另一个断点来控制断点设置。 此重写断点（分层断点）可以像正常断点一样启用/禁用/清除和列出，并添加了对它拥有的断点执行相同操作的功能。

例如，如果运行命令 `bp foo!bar`，导致两个匹配项与符号 `bar` 匹配，则将创建一个分层断点来控制这两个匹配项。 如果分层已启用/禁用/清除，则匹配的断点也将如此。

.bpcmds（显示断点命令）将列出可以运行以设置每个断点的断点命令。 分层断点拥有的断点仍将列出一个有效的 bp 命令，该命令将在其地址上设置断点。 分层断点也将在输出中列出，并显示可用于重新创建整组断点而非单个断点的命令。

不明确的符号

如果符号如下，在符号名称上设置断点应会导致以下行为：

- 重载：与符号匹配的每个重载都应具有断点。
- 模板函数：
  - 如果表达式指定了所有模板参数（例如 `bp foo!bar<int>`），则将在模板函数的特定实现上设置断点。
  - 如果表达式未指定类型实现（例如 `bp foo!bar`），则不会设置断点。 在这种情况下，`bm` 应用于在模板函数上设置断点。
  - 调试器不支持部分模板规范，在这种情况下不会设置断点。

- 内联函数：每个内联位置都有一个断点

请注意，当符号表达式包含需要调试器进行更多计算的运算符或偏移量时，将不会设置多个断点。 例如，如果符号 `foo` 解析到多个位置，但表达式 `foo+5` 已计算，调试器将不会尝试解析要设置断点的所有位置。

断点代码示例

给出以下代码片段：

C++

```
class BikeCatalog
{
public:
    void GetNumberOfBikes()
    {
        std::cout << "There are 42 bikes." << std::endl;
    }
    int GetNumberOfBikes(int num)
    {
        std::cout << "There are " << num << " bikes." << std::endl;
        return num;
    }
};
```

调用命令 `bu BikeCatalog::GetNumberOfBikes` 将导致创建两个断点，每个重载一个断点。列出断点将导致以下输出：

dbgcmd

```
0:000> bl
     2 e Disable Clear  <hierarchical breakpoint>     0001 (0001)  0:****
{BikeCatalog!BikeCatalog::GetNumberOfBikes}
         0 e Disable Clear  00007ff6`c6f52200
[C:\BikeCatalog\BikeCatalog.cpp @ 13]     0001 (0001)  0:****
BikeCatalog!BikeCatalog::GetNumberOfBikes
         1 e Disable Clear  00007ff6`c6f522a0
[C:\BikeCatalog\BikeCatalog.cpp @ 9]     0001 (0001)  0:****
BikeCatalog!BikeCatalog::GetNumberOfBikes
```

不明确的源行

如果源行为如下，在源行上设置断点应会导致以下行为：

- 编译器优化函数：如果由于编译器优化导致行被拆分在多个位置，则断点将设置在函数中与指定行对应的最低位置上。
- 内联函数：为每个调用站点设置断点，除非指定的行已作为内联的一部分进行优化。
- 解析为多个位置：如果未满足上述条件，则将为每个地址设置一个断点，条件如下：
  - 如果表达式中有一组与源行匹配的 *N* 地址，并且这些 *N* 地址的子集 *M* 在表达式的源行中没有源行位移，则只有 M 地址将具有断点。
  - 如果 *N* 地址集中没有表达式中源行的源行位移为零的地址，则所有 *N* 地址都将具有断点。

基于符号索引进行筛选

每个符号都应具有唯一的符号索引。 有关符号结构的详细信息，请参阅 SYMBOL_INFO 结构。

如果多个地址的源行位移为零，调试器将使用符号索引来确保过滤重复匹配项。

模板和重载函数的示例

模板函数

在模板函数定义的源行上设置断点后，将导致模板函数的每个实现都产生一个断点。 给定第 19 行`BikeCatalog.cpp`的以下模板函数：

```
C++

template <class T>
void RegisterBike(T id)
{
    std::cout << "Registered bike " << id << std::endl;
}
```

及其用法：

```
C++

catalog.RegisterBike("gravel bike");
catalog.RegisterBike(1234);
```

调用命令 `bp `BikeCatalog.cpp:19`` 将设置两个断点，这些断点解析为稍后在文件中使用的模板函数的实现。 如果用户想要在函数上设置单个断点，则必须在模板函数实现的特

定源行上设置断点，或者在具有适当类型信息（例如 `bp BikeCatalog::RegisterBike<int>`）的模板函数符号上设置断点。

列出断点会导致以下输出：

```
dbgcmd
```

```
0:000> bl
     2 e Disable Clear  <hierarchical breakpoint>     0001 (0001)  0:****
{BikeCatalog!BikeCatalog::RegisterBike&lt;int&gt;}
         0 e Disable Clear  00007ff7`6b691dd0
[C:\BikeCatalog\BikeCatalog.cpp @ 20]     0001 (0001)  0:****
BikeCatalog!BikeCatalog::RegisterBike<int>
         1 e Disable Clear  00007ff7`6b691e60
[C:\BikeCatalog\BikeCatalog.cpp @ 20]     0001 (0001)  0:****
BikeCatalog!BikeCatalog::RegisterBike<char const *>
```

重载函数

在源行上为重载函数的定义设置断点将导致重载函数的该定义上仅产生一个断点。 重用上述代码片段，第一行从第 5 行开始：

```
C++
```

```
class BikeCatalog
{
public:
    void GetNumberOfBikes()
    {
        std::cout << "There are 42 bikes." << std::endl;
    }
    int GetNumberOfBikes(int num)
    {
        std::cout << "There are " << num << " bikes." << std::endl;
        return num;
    }
};
```

调用命令 `bp `BikeCatalog.cpp:9`` 将在行上设置一个断点，以便 `void` 实现 `GetNumberOfBikes`。 列出断点会导致以下输出：

```
dbgcmd
```

```
0:000> bl
     0 e Disable Clear  00007ff7`6b691ec0  [C:\BikeCatalog\BikeCatalog.cpp @
9]     0001 (0001)  0:**** BikeCatalog!BikeCatalog::GetNumberOfBikes
```

内联函数

在内联函数的调用站点的源行上设置断点只会在该特定调用站点上生成一个断点，即使同一函数中存在另一个调用站点也是如此。

多个分层断点

分层断点将拥有其集中的每个断点，除非：

其集中的断点已清除

- 分层断点已清除。
- 创建另一个分层断点，其中包含此分层断点集中的一个断点。

另一种思路是，断点可能只有一个分层断点所有者，最近的断点命令将确定断点列表的状态。

此外，一个分层断点不能拥有另一个分层断点。

包含预先存在的断点

如果断点 *A* 本身存在，然后解析不明确的断点表达式以创建断点 *A*、*B*，则 *A* 将包含在带有 *B* 的新断点集中。

包含分层断点集交集

如果分层断点 *A* 拥有断点 *B*，则 *C* 和不明确的断点表达式被解析为创建断点：

- *B*、*C*、*D*：断点 *B*、*C* 将联接具有断点 *D* 的新分层断点组，并且分层断点 *A* 被清除。
- *C*、*D* 或 *B*、*D*：其中一个断点将联接具有断点 *D* 的新分层断点组，而分层断点 *A* 将继续存在，其余一个断点未加入新组。

另请参阅

使用断点

断点语法

bp、bu、bm（设置断点）

无法解析的断点（bu 断点）

用户空间和系统空间

项目 • 2023/06/15

Windows 为每个用户模式应用程序提供一个虚拟地址块。 这称为该应用程序 *的用户空间* 。 应用程序无法直接访问另一个大型地址块（称为 *系统空间* 或 *内核空间*）。

当 WinDbg 或 CDB 在用户空间中设置 断点 时，此断点在单个进程的用户空间中的指定地址处设置。 在用户模式调试期间，当前进程确定虚拟地址的含义。 有关详细信息，请参阅 控制进程和线程。

在内核模式下，可以使用 **bp**、 **bu** 和 **ba** 命令或"断点"对话框在用户空间中设置 **断点** 。 必须先使用 *进程上下文* 来指定用户模式进程，该进程使用 **.process /i** (或某些内核空间函数上的特定于进程的断点) 将目标切换到正确的 进程上下文。

用户空间中的断点始终与进程上下文在设置断点时处于活动状态的进程相关联。 如果用户模式调试器正在调试此进程，并且内核调试器正在调试运行该进程的计算机，则此断点会中断到用户模式调试器中，即使断点实际上是从内核调试器设置的。 此时，可以从内核调试器中断系统，或者从用户模式调试器使用 **.breakin (Break to the Kernel Debugger)** 命令将控制权转移到内核调试器。

确定用户空间和系统空间的范围

如果需要确定目标计算机上的用户空间和系统空间范围，可以使用内核调试器中的 **dp (Display Memory)** 命令显示 Windows 全局变量 **MmHighestUserAddress**。 此变量包含用户空间顶部的地址。 由于系统空间地址始终高于用户空间地址，因此可以使用此值确定任何给定地址是在用户空间中还是在内核空间中。

例如，在具有 x86 处理器和标准启动参数的 32 位目标计算机上，此命令将显示以下结果：

```
dbgcmd
```

```
kd> dp nt!mmhighestuseraddress L1
81f71864  7ffeffff
```

这表明用户空间的范围从地址0x00000000到0x7FFEFFFF，因此系统空间的范围从0x80000000到在标准 32 位 Windows 安装) 上0xFFFFFFFF的最高可能地址 (。

对于 64 位目标计算机，将发生不同的值。 例如，此命令可能会显示以下内容：

```
dbgcmd
```

```
0: kd> dp nt!mmhighestuseraddress L1
fffff800`038b4010  000007ff`fffeffff
```

这表示用户空间的范围从 0x00000000'00000000 到 0x000007FF'FFFEFFFF。 因此，系统空间包括0x00000800'00000000 以上的所有地址。

有关 Windows 内存管理的详细信息，请参阅 Microsoft *Windows Internals* by David Solomon 和 Mark Russinovich (第 4 版，Microsoft Press，2005) 。

设置断点造成的风险

项目 • 2023/10/26

通过指定内存地址或符号加上偏移量来设置 断点 时，不得将此断点置于指令中间。

例如，请考虑以下反汇编代码。

```
dbgcmd

770000f1 5e              pop     esi
770000f2 5b              pop     ebx
770000f3 c9              leave
770000f4 c21000          ret     0x10
770000f7 837ddc00        cmp     dword ptr [ebp-0x24],0x0
```

前三个指令只有一个字节长。 但是，第四条指令的长度为三个字节。 (它包括字节0x770000F4、0x770000F5和0x770000F6.) 如果要使用 **bp**、 **bu** 或 **ba** 命令在此指令上放置断点，则必须指定0x770000F4地址。

如果使用 **ba** 命令在0x770000F5地址中放置断点，处理器会在该位置放置断点。 但永远不会触发此断点，因为处理器将0x770000F4视为指令的实际地址。

如果使用 **bp** 或 **bu** 命令在 0x770000F5 地址中放置断点，调试器会在该位置写入断点。 但是，由于调试器创建断点的方式，此断点可能会损坏目标：

1. 调试器保存0x770000F5的内容，并使用断点指令覆盖此内存。
2. 如果尝试在调试器中显示此内存，调试器不会显示它已写入的断点指令。 相反，调试器会显示“应该”的内存。 也就是说，调试器会显示原始内存，或者自插入断点以来对该内存所做的任何修改。
3. 如果使用 **BC** 命令删除断点，调试器会将原始内存还原到其正确位置。

在0x770000F5放置断点时，调试器会保存此字节，并在此处写入中断指令。 但是，当应用程序运行时，它会到达0x770000F4地址，并将此地址识别为多字节指令的第一个字节。 然后，处理器尝试将0x770000F4、0x770000F5，以及以后的一些字节合并到单个指令中。 此组合可以创建各种行为，这些行为都不可取。

因此，使用 **bp**、 **bu** 或 **ba** 命令放置断点时，请确保始终将断点放在正确的地址。 如果使用 WinDbg 图形界面添加断点，则无需担心这种情况，因为会自动选择正确的地址。

不明确的断点解析

在调试器引擎的版本 10.0.25310.1001 及更高版本中，现在支持不明确的断点解析。 不明确的断点允许调试器在某些断点表达式解析为多个位置的情况下设置断点。 有关详细信息，请参阅 不明确的断点解析。

请参阅

使用断点

断点语法

bp、bu、bm（设置断点）

不明确的断点解析

无法解析的断点（bu 断点）

WinDbg 和其他 Windows 调试器中的条件断点

项目 • 2024/05/27

仅在满足特定条件才需要中断时，WinDbg 和其他 Windows 调试器中的条件中断点非常有用。

条件断点可使用 **bp（设置断点）**或其他断点命令的"/w"参数来创建。 该命令的基本语法如下：

```
dbgcmd

0:000> bp /w "(Condition)" Address
```

只有当指定条件为 true 时，断点才会中断到调试器。 "w"是"when"的缩写。 条件表达式可以是任何可用于 **dx（显示调试器对象模型表达式）**命令的表达式。 这包括大多数 C++ 样式的表达式，包括比较、算术、指针操作等。 例如，一个基本的条件断点只有在变量大于 20 时才会断开，则可以编写为：

```
dbgcmd

0:000> bp /w "MyVar > 20" `mysource.cpp:143`
```

由于条件使用调试器对象模型进行评估，因此还可以利用 NatVis 支持等功能。 例如，假设 myVec 是一个 `std::vector<int>`，则可以创建如下条件：

```
dbgcmd

0:000> bp /w "myVec.Count() == 4" `mysource.cpp:143`
```

当 myVec 变量包含 4 个元素时，执行 mysource.cpp 的第 143 行将会中断。

除了 natvis，还可以调用 JavaScript 函数。 如果使用 WinDbg 脚本窗口或 **.scriptload（加载脚本）**命令加载了一个脚本，其中包含一个名为"myFunc"的函数，则可以按照如下所示来设置断点：

```
dbgcmd

0:000> bp /w "@$scriptContents.myFunc()" `mysource.cpp:143`
```

有关在调试器中编写 JavaScript 函数和扩展的详细信息，请参阅 JavaScript 调试器脚本

虽然较高级别的表达式通常最为有用，但也可以使用这些表达式来计算寄存器。 例如，可以创建一个仅当堆栈指针达到某个阈值时才触发的断点：

```
dbgcmd
```

```
0:000> bp /w "@esp < 0x6ff9f8" `mysource.cpp:143`
```

旧式条件断点语法

在断点命令使用"/w"参数之前，设置条件断点的推荐方法是使用 **j (Execute If - Else)** 命令或 **.if** 令牌，然后使用 **gc（从条件断点开始）** 命令。 虽然不再推荐使用这些设置条件断点的方法，但它们仍然有效，而其他资料中也会提及这种语法。

使用 **j** 命令创建条件断点的基本语法如下所示：

```
dbgcmd
```

```
0:000> bp Address "j (Condition) 'OptionalCommands'; 'gc' "
```

使用 **.if** 令牌创建条件断点的基本语法如下所示：

```
dbgcmd
```

```
0:000> bp Address ".if (Condition) {OptionalCommands} .else {gc}"
```

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

一直执行到出现指定的状态为止

项目 • 2023/06/27

有多种方法可导致目标执行，直到达到指定状态。

使用断点控制执行

一种方法是使用断点。 当程序计数器到达指定地址时，最简单的断点将停止执行。 更复杂的断点可以：

- 仅在特定线程执行此地址时触发，
- 在触发之前，允许指定数量的传递通过此地址，
- 触发时自动发出指定的命令，或者
- watch非可执行内存中的指定地址，在读取或写入该内存时触发。

有关如何设置和控制断点的详细信息，请参阅 使用断点。

在达到指定状态之前，一种更复杂的执行方法是使用 *条件断点*。 此类断点在某个地址设置，但仅当指定条件成立时才会触发。 有关详细信息，请参阅 设置条件断点。

断点和Pseudo-Registers

在指定所需状态时，使用 *自动伪寄存器*通常很有帮助。 这些变量由调试器控制，可用于引用与目标状态相关的各种值。

例如，以下断点使用 **伪寄存器**$thread ，该寄存器始终等于当前线程的值。 在命令中使用时，它解析为当前线程的值。 通过将 $thread 用作 bp (set Breakpoint) 命令的 /t 参数的参数，可以创建每次由发出 bp 命令时处于活动状态的线程调用 NtOpenFile 时触发的**断**点：

```
dbgcmd
```

```
kd> bp /t @$thread nt!ntopenfile
```

当任何其他线程调用 NtOpenFile 时，不会触发此断点。

有关自动伪寄存器的列表，请参阅 伪寄存器语法。

使用脚本文件控制执行

在达到指定状态之前执行的另一种方法是创建一个以递归方式调用自身的脚本文件，并在每次迭代中测试所需状态。

通常，此脚本文件将包含 .if 和 .else 标记。 可以使用 t (Trace) 等命令执行单个步骤，然后测试有问题的条件。

例如，如果要在 eax 寄存器包含值0x1234之前执行，则可以创建包含以下行的名为 *eaxstep* 的脚本文件：

```
dbgcmd
```

```
.if (@eax == 1234) { .echo 1234 } .else { t "$<eaxstep" }
```

然后从“调试器命令”窗口发出以下命令：

```
dbgcmd
```

```
t "$<eaxstep"
```

此 t 命令将执行单个步骤，然后执行带引号的命令。 此命令恰好是 $< (运行脚本文件) 运行 *eaxstep* 文件。 脚本文件测试 eax 的值，运行 t 命令，然后以递归方式调用自身。 这会一直持续到 eax 寄存器等于0x1234，此时 .echo (Echo Comment) 命令将消息输出到调试器命令窗口，并停止执行。

有关脚本文件的详细信息，请参阅使用脚本文件和使用调试器命令程序。

不明确的断点解析

在版本 10.0.25310.1001 及更高版本的调试器引擎中，现在支持不明确的断点解析。 不明确的断点允许调试器在某些断点表达式解析到多个位置的情况下设置断点。 有关详细信息，请参阅 不明确的断点解析。

请参阅

使用断点

断点语法

bp、bu、bm（设置断点）

不明确的断点解析

无法解析的断点（bu 断点）

读取和写入内存

项目 • 2023/06/15

Windows 调试器可以直接读取和写入内存。 此内存可由地址或变量名称引用。

本节包括下列主题：

按虚拟地址访问内存

按物理地址访问内存

访问全局变量

访问局部变量

通过监视窗口控制变量

按虚拟地址访问内存

项目 • 2024/02/22

若要访问内存地址或地址范围，可以使用多个命令。 Visual Studio 和 WinDbg 提供可用于查看和编辑内存的用户界面元素（以及命令）。 有关详细信息，请参阅在WinDbg (Classic) 中查看和编辑内存。

以下命令可以读取或写入各种格式的内存。 这些格式包括十六进制字节、字词（字词、双字和四进制字词）、整数（短、长、四进制整数和无符号整数）、浮点数（10 字节、16 字节、32 字节和 64 字节实数）和 ASCII 字符。

- **d*（显示内存）**命令显示指定内存地址或范围的内容。
- **e*（输入值）**命令将值写入指定的内存地址。

可以使用以下命令来处理更加专用的数据类型：

- **dt（显示类型）**命令会查找各种数据类型，并显示由正在调试的应用程序创建的数据结构。 此命令具有很多用途，具有许多变体和选项。
- **ds、dS（显示字符串）**命令显示 STRING、ANSI_STRING 或 UNICODE_STRING 数据结构。
- **dl（显示链接列表）**命令跟踪并显示链接列表。
- **d*s（显示字词和符号）**命令会查找可能包含符号信息的双字或四进制字词，然后显示数据和符号信息。
- **!address** 扩展命令显示有关位于特定地址的内存属性的信息。

可以使用以下命令操作内存范围：

- **m（移动内存）**命令将一个内存范围的内容移到另一个内存范围。
- **f（填充内存）**命令将模式写入内存范围，重复该模式，直到范围已满。
- **c（比较内存）**命令比较两个内存范围的内容。
- **s（搜索内存）**命令在内存范围内搜索指定的模式，或搜索内存范围内存在的任何 ASCII 或 Unicode 字符。
- **.holdmem（保留和比较内存）**命令将一个内存范围与另一个内存范围进行比较。

在大多数情况下，这些命令在当前基数中解释其参数。 因此，如果当前基数不是 16，则应在十六进制地址之前添加 **0x**。 但是，这些命令的显示输出通常采用十六进制格式，而

不考虑当前基数。 （有关输出的详细信息，请参阅单个命令主题。）“内存”窗口以十进制格式显示整数和实数，并以十六进制格式显示其他格式。

若要更改默认基数，请使用 **n（设置基数）** 命令。 若要快速将数字从一个基数转换为另一个基数，请使用 **?（计算表达式）** 命令或 **.formats（显示数字格式）** 命令。

执行用户模式调试时，虚拟地址的含义由当前进程确定。 执行内核模式调试时，虚拟地址的含义可由调试器控制。 有关详细信息，请参阅进程上下文。

按物理地址访问内存

项目 • 2024/02/22

若要从物理地址读取，请使用 !db、!dc、!dd、!dp、!du 和 !dw 扩展命令。

若要写入物理地址，请使用 !eb 和 !ed 扩展命令。

fp（填充物理内存）命令将模式写入物理内存范围，重复该模式，直到范围已满。

在内核模式下使用 WinDbg 时，还可以直接从“内存”窗口读取或写入物理内存。

若要在物理内存中搜索一段数据或一个数据范围，请使用 !search 扩展命令。

此外，有关物理地址的详细信息，请参阅将虚拟地址转换为物理地址。

访问全局变量

项目 • 2024/02/22

全局变量的名称存储在编译应用程序时创建的符号文件中。 调试器将全局变量的名称解释为虚拟地址。 接受地址作为参数的任何命令也接受变量的名称。 因此，可以使用通过虚拟地址访问内存中介绍的所有命令来读取或写入全局变量。

此外，可以使用 **？（计算表达式）** 命令显示与任何符号关联的地址。

WinDbg 提供用户界面元素，可以使用这些元素（除了命令）来查看和编辑全局变量。 请参阅在 WinDbg (Classic) 中查看和编辑全局变量。

请看下面的示例。 假设您想要检查 `MyCounter` 全局变量，该变量是一个 32 位整数。 另假设默认基数为 10。

可以获取此变量的地址，然后如下所示显示该变量。

dbgcmd

```
0:000> ? MyCounter
Evaluate expression: 1244892 = 0012fedc
0:000> dd 0x0012fedc L1
0012fedc  00000052
```

第一个命令输出会告知 `MyCounter` 的地址是 0x0012FEDC。 然后可以使用 **d*（显示内存）** 命令在此地址处显示一个双字。 （也可以使用 1244892，这是此地址的十进制版本。但是，大多数 C 程序员更喜欢使用 0x0012FEDC。）第二个命令告知 MyCounter 的值为 0x52（十进制 82）。

也可以在以下命令中执行这些步骤。

dbgcmd

```
0:000> dd MyCounter L1
0012fedc  00000052
```

若要将 `MyCounter` 的值更改为十进制 83，请使用以下命令。

dbgcmd

```
0:000> ed MyCounter 83
```

此示例使用十进制输入，因为该格式对于整数而言更为自然。 但是，d* 命令的输出仍采用十六进制格式。

dbgcmd

```
0:000> dd MyCounter L1 0012fedc  00000053
```

访问局部变量

项目 • 2024/02/22

局部变量如同全局变量，也存储在符号文件中。 与全局变量一样，调试器将其名称解释为地址。 可以采用与全局变量相同的方式读取和写入局部变量。 但是，如果需要向命令指示符号是局部符号，请在符号前面加上一个美元符号 ($) 和一个感叹号 (!)，如 `$!var` 中所示。

Visual Studio 和 WinDbg 提供用户界面元素，可以使用这些元素（除了命令）来查看和编辑局部变量。 有关详细信息，请参阅在 WinDbg (Classic) 中查看和编辑局部变量。

还可以使用以下方法来显示、更改和使用局部变量：

- **dv（显示局部变量）**命令显示所有局部变量的名称和值。
- 利用 **!for_each_local** 扩展，可以针对每个局部变量重复执行单个命令一次。

但是，局部变量和全局变量之间存在一项主要差异。 当应用程序执行时，局部变量的含义取决于程序计数器的位置，因为此类变量的范围仅扩展到定义它们的函数。

调试器根据本地上下文解释局部变量。 默认情况下，此上下文与程序计数器的位置匹配。 但是，调试器可以更改上下文。 有关本地上下文的详细信息，请参阅“本地上下文”。

更改本地上下文时，将立即更新“局部变量”窗口，以反映局部变量的新集合。 **dv** 命令还显示新变量。 然后，所有这些变量名称都由上述内存命令正确解释。 然后，可以读取或写入这些变量。

通过监视窗口控制变量

项目 • 2023/06/15

在 WinDbg 中，还可以使用“监视”窗口显示和更改全局和局部变量。

“监视”窗口可以显示所需的任何变量列表。 这些变量可以包括来自任何函数的全局变量和局部变量。 “监视”窗口随时显示与当前函数的范围匹配的变量的值。 还可以通过“监视”窗口更改这些变量的值。

与“局部变量”窗口不同，“监视”窗口不受本地上下文更改的影响。 只有那些在当前程序计数器范围内定义的变量才能显示或修改其值。

将虚拟地址转换为物理地址

项目 • 2023/06/15

大多数调试器命令使用虚拟地址（而不是物理地址）作为其输入和输出。 但是，有时拥有物理地址可能很有用。

可通过两种方式将虚拟地址转换为物理地址：使用 **！ vtop** 扩展和 **！ pte** 扩展。

有关 Windows 中虚拟地址的概述，请参阅 虚拟地址空间。

使用！ vtop 进行地址转换

假设要调试运行MyApp.exe进程的目标计算机，并且想要调查虚拟地址0x0012F980。 下面是用于 **！ vtop** 扩展以确定相应物理地址的过程。

使用！ vtop 将虚拟地址转换为物理地址

1. 请确保以十六进制格式工作。 如有必要，请使用 N 16 命令设置当前基数。
2. 确定地址的 *字节索引* 。 此数字等于虚拟地址的最低 12 位。 因此，虚拟地址0x0012F980的字节索引为0x980。
3. 使用 **！ process** 扩展确定地址的*目录基*:

   dbgcmd

   ```
   kd> !process 0 0
   **** NT ACTIVE PROCESS DUMP ****
   ....
   PROCESS ff779190  SessionId: 0  Cid: 04fc    Peb: 7ffdf000  ParentCid:
   0394
    DirBase: 098fd000  ObjectTable: e1646b30  TableSize:   8.
       Image: MyApp.exe
   ```

4. 确定目录基 *页的页码* 。 这只是没有三个尾随十六进制零的目录基。 在此示例中，目录基为0x098FD000，因此页帧编号0x098FD。
5. 使用 **！ vtop** 扩展。 此扩展的第一个参数应该是页帧编号。 **！ vtop** 的第二个参数应该是有问题的虚拟地址：

   dbgcmd

   ```
   kd> !vtop 98fd 12f980
   Pdi 0 Pti 12f
   0012f980 09de9000 pfn(09de9)
   ```

最后一行中显示的第二个数字是物理页开头的物理地址。

6. 将字节索引添加到页面开头的地址：0x09DE9000 + 0x980 = 0x09DE9980。 这是所需的物理地址。

可以通过在每个地址显示内存来验证此计算是否正确完成。 ！ d* 扩展在指定的物理地址处显示内存：

dbgcmd

```
kd> !dc 9de9980
# 9de9980 6d206e49 726f6d65 00120079 0012f9f4 In memory.......
# 9de9990 0012f9f8 77e57119 77e8e618 ffffffff .....q.w...w....
# 9de99a0 77e727e0 77f6f13e 77f747e0 ffffffff .'.w>..w.G.w....
# 9de99b0 .....
```

d* (Display Memory) 命令使用虚拟地址作为其参数：

dbgcmd

```
kd> dc 12f980
0012f980  6d206e49 726f6d65 00120079 0012f9f4  In memory.......
0012f990  0012f9f8 77e57119 77e8e618 ffffffff  .....q.w...w....
0012f9a0  77e727e0 77f6f13e 77f747e0 ffffffff  .'.w>..w.G.w....
0012f9b0  .....
```

由于结果相同，这表示物理地址0x09DE9980确实对应于虚拟地址0x0012F980。

使用！ pte 进行地址转换

同样，假设你正在调查属于MyApp.exe进程的虚拟地址0x0012F980。 下面是用于！ pte 扩展以确定相应物理地址的过程：

使用！ pte 将虚拟地址转换为物理地址

1. 请确保以十六进制格式工作。 如有必要，请使用 N 16 命令设置当前基数。

2. 确定地址的 *字节索引* 。 此数字等于虚拟地址的最低 12 位。 因此，虚拟地址0x0012F980的字节索引为0x980。

3. 将 进程上下文 设置为所需的进程：

dbgcmd

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
....
PROCESS ff779190  SessionId: 0  Cid: 04fc    Peb: 7ffdf000  ParentCid:
0394
    DirBase: 098fd000  ObjectTable: e1646b30  TableSize:   8.
    Image: MyApp.exe

kd> .process /p ff779190
Implicit process is now ff779190
.cache forcedecodeuser done
```

4. 将 **！ pte** 扩展与虚拟地址一起使用作为其参数。 这在两列中显示信息。 左列描述此地址 (PDE) 的页面目录条目;右列描述其页表条目 (PTE)：

```
dbgcmd
```

```
kd> !pte 12f980
               VA 0012f980
PDE at   C0300000        PTE at C00004BC
contains 0BA58067      contains 09DE9067
pfn ba58 ---DA--UWV    pfn 9de9 ---DA--UWV
```

5. 查看右列的最后一行。 此时会显示表示法“pfn 9de9”。 0x9DE9编号是此 PTE) (*PFN 的页码* 。 将页帧编号乘以0x1000 (例如，将页码左移 12 位) 。 结果（0x09DE9000）是页面开头的物理地址。

6. 将字节索引添加到页面开头的地址：0x09DE9000 + 0x980 = 0x09DE9980。 这是所需的物理地址。

这与之前方法获取的结果相同。

手动转换地址

尽管 **！ ptov** 和 **pte** 扩展提供了将虚拟地址转换为物理地址的最快方法，但也可以手动进行此转换。 此过程的说明将阐明虚拟内存体系结构的一些细节。

内存结构的大小因处理器和硬件配置而异。 此示例取自未启用物理地址扩展 (PAE) 的 x86 系统。

再次使用 0x0012F980 作为虚拟地址，首先需要手动或使用 **.formats (显示数字格式)** 命令将其转换为二进制地址：

```
dbgcmd
```

```
kd> .formats 12f980
Evaluate expression:
  Hex:     0012f980
  Decimal: 1243520
  Octal:   00004574600
  Binary:  00000000 00010010 11111001 10000000
  Chars:   ....
  Time:    Thu Jan 15 01:25:20 1970
  Float:   low 1.74254e-039 high 0
  Double:  6.14381e-318
```

此虚拟地址是三个字段的组合。 位 0 到 11 是字节索引。 位 12 到 21 是页表索引。 位 22 到 31 是页面目录索引。 分隔字段后，你有：

dbgcmd

```
0x0012F980  =  0y  00000000 00   010010 1111   1001 10000000
```

这会公开虚拟地址的三个部分：

- 页目录索引 = 0y00000000000 = 0x0
- 页表索引 = 0y0100101111 = 0x12F
- 字节索引 = 0y100110000000 = 0x980

然后，系统需要另外三条信息。

- 每个 PTE 的大小。 这是非 PAE x86 系统上的 4 个字节。
- 页面的大小。 这是0x1000个字节。
- PTE_BASE虚拟地址。 在非 PAE 系统上，这是0xC0000000。

使用此数据，可以计算 PTE 本身的地址：

dbgcmd

```
PTE address   =   PTE_BASE
                + (page directory index) * PAGE_SIZE
                + (page table index) * sizeof(MMPTE)
              =   0xc0000000
                + 0x0   * 0x1000
                + 0x12F * 4
              =   0xC00004BC
```

这是 PTE 的地址。 PTE 是 32 位 DWORD。 检查其内容：

dbgcmd

```
kd> dd 0xc00004bc L1
c00004bc  09de9067
```

此 PTE 的值0x09DE9067。 它由两个字段组成：

- PTE 的低 12 位是 *状态标志*。 在这种情况下，这些标志等于 0x067 -- 或二进制 0y0000011100111。 有关状态标志的说明，请参阅 **！ pte** 参考页。
- PTE 的高 20 位等于 PTE (PFN) *页帧数* 。 在这种情况下，PFN 是0x09DE9。

物理页上的第一个物理地址是 PFN 乘以0x1000 (移左 12 位) 。 字节索引是此页上的偏移量。 因此，要查找的物理地址为 0x09DE9000 + 0x980 = 0x09DE9980。 这与之前的方法获取的结果相同。

使用 !analyze 扩展

项目 • 2024/07/08

调试崩溃的目标计算机或应用程序的第一步是使用 **！ analyze** 扩展命令。 此扩展执行大量自动分析。 此分析的结果显示在“调试器命令”窗口中。

应将 **-v** 选项用于完全详细地显示数据。 有关其他选项的详细信息，请参阅 **！ analyze** 参考页。

用户模式！ analyze -v 示例

在此示例中，调试器附加到遇到异常的用户模式应用程序。

dbgcmd

```
0:000> !analyze -v
*******************************************************************************
***
*
*
*                        Exception Analysis
*
*
*
*******************************************************************************
***

Debugger SolutionDb Connection::Open failed 80004005
```

如果连接到 Internet，调试器将尝试访问由Microsoft维护的崩溃解决方案数据库。 在这种情况下，将显示一条错误消息，指示计算机无法访问 Internet 或网站未正常工作。

dbgcmd

```
FAULTING_IP:
ntdll!PropertyLengthAsVariant+73
77f97704 cc               int     3
```

FAULTING_IP字段在故障发生时显示指令指针。

dbgcmd

```
EXCEPTION_RECORD:  ffffffff -- (.exr ffffffffffffffff)
ExceptionAddress: 77f97704 (ntdll!PropertyLengthAsVariant+0x00000073)
   ExceptionCode: 80000003 (Break instruction exception)
```

```
  ExceptionFlags: 00000000
NumberParameters: 3
   Parameter[0]: 00000000
   Parameter[1]: 00010101
   Parameter[2]: ffffffff
```

EXCEPTION_RECORD字段显示此崩溃的异常记录。 也可以使用 .exr（显示异常记录）**命令查看**此信息。

```
dbgcmd

BUGCHECK_STR:  80000003
```

BUGCHECK_STR字段显示异常代码。 该名称是错误分类 - 术语 *bug 检查* 实际上表示内核模式崩溃。 在用户模式调试中，将显示异常代码-在本例中，0x80000003。

```
dbgcmd

DEFAULT_BUCKET_ID:  APPLICATION_FAULT
```

DEFAULT_BUCKET_ID字段显示此失败所属的一般故障类别。

```
dbgcmd

PROCESS_NAME:  MyApp.exe
```

PROCESS_NAME字段指定引发异常的进程的名称。

```
dbgcmd

LAST_CONTROL_TRANSFER:  from 01050963 to 77f97704
```

LAST_CONTROL_TRANSFER字段显示堆栈上的最后一次调用。 在这种情况下，地址处的代码0x01050963在0x77F97704调用函数。 可以将这些地址与 ln（列表最近的符号）**命令一起使用**，以确定这些地址所在的模块和函数。

```
dbgcmd

STACK_TEXT:
0006b9dc 01050963 00000000 0006ba04 000603fd
ntdll!PropertyLengthAsVariant+0x73
0006b9f0 010509af 00000002 0006ba04 77e1a449 MyApp!FatalErrorBox+0x55
[D:\source_files\MyApp\util.c @ 541]
0006da04 01029f4e 01069850 0000034f 01069828 MyApp!ShowAssert+0x47
[D:\source_files\MyApp\util.c @ 579]
0006db6c 010590c3 000e01ea 0006fee4 0006feec MyApp!SelectColor+0x103
```

```
[D:\source_files\MyApp\colors.c @ 849]
0006fe04 77e11d0a 000e01ea 00000111 0000413c MyApp!MainWndProc+0x1322
[D:\source_files\MyApp\MyApp.c @ 1031]
0006fe24 77e11bc8 01057da1 000e01ea 00000111 USER32!UserCallWinProc+0x18
0006feb0 77e172b4 0006fee4 00000001 010518bf
USER32!DispatchMessageWorker+0x2d0
0006febc 010518bf 0006fee4 00000000 01057c5d USER32!DispatchMessageA+0xb
0006fec8 01057c5d 0006fee4 77f82b95 77f83920 MyApp!ProcessQCQPMessage+0x3b
[D:\source_files\MyApp\util.c @ 2212]
0006ff70 01062cbf 00000001 00683ed8 00682b88 MyApp!main+0x1e6
[D:\source_files\MyApp\MyApp.c @ 263]
0006ffc0 77e9ca90 77f82b95 77f83920 7ffdf000 MyApp!mainCRTStartup+0xff
[D:\source_files\MyApp\crtexe.c @ 338]
0006fff0 00000000 01062bc0 00000000 000000c8 KERNEL32!BaseProcessStart+0x3d
```

STACK_TEXT字段显示故障组件的堆栈跟踪。

dbgcmd

```
FOLLOWUP_IP:
MyApp!FatalErrorBox+55
01050963 5e              pop     esi

FOLLOWUP_NAME:  dbg

SYMBOL_NAME:  MyApp!FatalErrorBox+55

MODULE_NAME:  MyApp

IMAGE_NAME:  MyApp.exe

DEBUG_FLR_IMAGE_TIMESTAMP:  383490a9
```

当！ analyze **确定可能导致错误的指令时**，它会在FOLLOWUP_IP字段中显示该指令。 SYMBOL_NAME、MODULE_NAME、IMAGE_NAME和DEBUG_FLR_IMAGE_TIMESTAMP字段显示与此指令对应的符号、模块、图像名称和图像时间戳。

dbgcmd

```
STACK_COMMAND:  .ecxr ; kb
```

STACK_COMMAND字段显示用于获取STACK_TEXT的命令。 可以使用此命令重复此堆栈跟踪显示，或更改它以获取相关的堆栈信息。

dbgcmd

```
BUCKET_ID:  80000003_MyApp!FatalErrorBox+55
```

BUCKET_ID字段显示当前故障所属的特定故障类别。 此类别可帮助调试器确定在分析输出中显示的其他信息。

dbgcmd

```
Followup: dbg
---------
```

有关FOLLOWUP_NAME和后续字段的信息，请参阅 “跟进字段”和“triage.ini文件”。

可能会显示多种其他字段：

- 如果控制权已传输到无效地址，则FAULTING_IP字段将包含此无效地址。 FAILED_INSTRUCTION_ADDRESS字段将显示此地址中的反汇编代码，而不是FOLLOWUP_IP字段，尽管此反汇编可能毫无意义。 在这种情况下，SYMBOL_NAME、MODULE_NAME、IMAGE_NAME和DEBUG_FLR_IMAGE_TIMESTAMP字段将引用此指令的调用方。
- 如果处理器失火，你可能会看到SINGLE_BIT_ERROR、TWO_BIT_ERROR或POSSIBLE_INVALID_CONTROL_TRANSFER字段。
- 如果内存损坏似乎已发生，则CHKIMG_EXTENSION字段将指定应该用于调查的 **!chkimg** 扩展命令。

内核模式！ analyze -v 示例

在此示例中，调试器附加到刚刚崩溃的计算机。

dbgcmd

```
kd> !analyze -v
*******************************************************************************
***
*
*
*                        Bugcheck Analysis
*
*
*
*******************************************************************************
***

DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pagable (or completely invalid) address at
an
interrupt request level (IRQL) that is too high.  This is usually
```

```
caused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
```

显示的第一个元素显示 bug 检查代码和有关此类 bug 检查的信息。 显示的某些文本可能不适用于此特定实例。 有关每个 bug 检查的更多详细信息，请参阅 Bug 检查代码参考部分。

dbgcmd

```
Arguments:
Arg1: 00000004, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000001, value 0 = read operation, 1 = write operation
Arg4: f832035c, address which referenced memory
```

接下来会显示 bug 检查参数。 它们各有说明。 例如，第三个参数为 1，下面的注释会说明写入操作失败。

dbgcmd

```
## Debugging Details:


WRITE_ADDRESS:  00000004 Nonpaged pool

CURRENT_IRQL:  2
```

接下来的几个字段因崩溃的性质而异。 在本例中，我们看到WRITE_ADDRESS和CURRENT_IRQL字段。 这些只是重述 bug 检查参数中显示的信息。 通过将语句“Nonpaged pool”与错误检查文本进行比较，该文本读取了“尝试访问可分页地址（或完全无效）地址”，可以看到地址无效。 本例中无效的地址0x00000004。

dbgcmd

```
FAULTING_IP:
USBPORT!USBPORT_BadRequestFlush+7c
f832035c 894204          mov     [edx+0x4],eax
```

FAULTING_IP字段在故障发生时显示指令指针。

dbgcmd

```
DEFAULT_BUCKET_ID:  DRIVER_FAULT
```

DEFAULT_BUCKET_ID字段显示此失败所属的一般故障类别。

dbgcmd

```
BUGCHECK_STR:  0xD1
```

BUGCHECK_STR字段显示我们已经看到的 bug 检查代码。 在某些情况下，会追加其他会审信息。

dbgcmd

```
TRAP_FRAME:  f8950dfc -- (.trap ffffffff8950dfc)
.trap ffffffff8950dfc
ErrCode = 00000002
eax=81cc86dc ebx=81cc80e0 ecx=81e55688 edx=00000000 esi=81cc8028
edi=8052cf3c
eip=f832035c esp=f8950e70 ebp=f8950e90 iopl=0         nv up ei pl nz ac po
nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00010216
USBPORT!USBPORT_BadRequestFlush+7c:
f832035c 894204          mov     [edx+0x4],eax
ds:0023:00000004=????????
.trap
Resetting default context
```

TRAP_FRAME字段显示此崩溃的陷阱帧。 也可以使用 .trap（显示陷阱帧）**命令查看**此信息。

dbgcmd

```
LAST_CONTROL_TRANSFER:  from f83206e0 to f832035c
```

LAST_CONTROL_TRANSFER字段显示堆栈上的最后一次调用。 在这种情况下，地址处的代码0xF83206E0在0xF832035C调用函数。 可以使用 **ln（列表最近的符号）** 命令来确定这些地址所在的模块和函数。

dbgcmd

```
STACK_TEXT:
f8950e90 f83206e0 024c7262 00000000 f8950edc
USBPORT!USBPORT_BadRequestFlush+0x7c
f8950eb0 804f5561 81cc8644 81cc8028 6d9a2f30
USBPORT!USBPORT_DM_TimerDpc+0x10c
f8950fb4 804f5644 6e4be98e 00000000 ffdff000 nt!KiTimerListExpire+0xf3
f8950fe0 8052c47c 8053cf20 00000000 00002e42 nt!KiTimerExpiration+0xb0
f8950ff4 8052c16a efdefd44 00000000 00000000 nt!KiRetireDpcList+0x31
```

STACK_TEXT字段显示故障组件的堆栈跟踪。

```
dbgcmd
```

```
FOLLOWUP_IP:
USBPORT!USBPORT_BadRequestFlush+7c
f832035c 894204          mov     [edx+0x4],eax
```

FOLLOWUP_IP字段显示可能导致错误的指令的反汇编。

```
dbgcmd
```

```
FOLLOWUP_NAME:  usbtri

SYMBOL_NAME:  USBPORT!USBPORT_BadRequestFlush+7c

MODULE_NAME:  USBPORT

IMAGE_NAME:  USBPORT.SYS

DEBUG_FLR_IMAGE_TIMESTAMP:  3b7d868b
```

SYMBOL_NAME、MODULE_NAME、IMAGE_NAME和DBG_FLR_IMAGE_TIMESTAMP字段显示与此指令对应的符号、模块、图像和图像时间戳（如果有效），或此指令的调用方（如果无效）。

```
dbgcmd
```

```
STACK_COMMAND:  .trap ffffffff8950dfc ; kb
```

STACK_COMMAND字段显示用于获取STACK_TEXT的命令。 可以使用此命令重复此堆栈跟踪显示，或更改它以获取相关的堆栈信息。

```
dbgcmd
```

```
BUCKET_ID:  0xD1_W_USBPORT!USBPORT_BadRequestFlush+7c
```

BUCKET_ID字段显示当前故障所属的特定故障类别。 此类别可帮助调试器确定在分析输出中显示的其他信息。

有关FOLLOWUP_NAME和后续字段的信息，请参阅 “跟进字段”和“triage.ini文件”。

可能会显示多种其他字段：

- 如果控制权已传输到无效地址，则FAULTING_IP字段将包含此无效地址。 FAILED_INSTRUCTION_ADDRESS字段将显示此地址中的反汇编代码，而不是FOLLOWUP_IP字段，尽管此反汇编可能毫无意义。 在这种情况下，

SYMBOL_NAME、MODULE_NAME、IMAGE_NAME和DBG_FLR_IMAGE_TIMESTAMP字段将引用此指令的调用方。

- 如果处理器失火，你可能会看到SINGLE_BIT_ERROR、TWO_BIT_ERROR或POSSIBLE_INVALID_CONTROL_TRANSFER字段。
- 如果内存损坏似乎已发生，则CHKIMG_EXTENSION字段将指定应该用于调查的 **!chkimg** 扩展命令。
- 如果在设备驱动程序的代码中发生了 bug 检查，则其名称可能会在BUGCHECKING_DRIVER字段中显示。

后续字段和triage.ini文件

在用户模式和内核模式下，如果可以确定，显示的“跟进”字段将显示有关当前堆栈帧所有者的信息。 此信息按以下方式确定：

1. **使用！ analyze** 扩展时，调试器从堆栈中的顶部帧开始，并确定它是否对错误负责。 如果不是，将分析下一帧。 此过程将继续执行，直到找到可能处于故障的帧。
2. 调试器尝试在此帧中确定模块和函数的所有者。 如果可以确定所有者，则此帧被视为有故障。
3. 如果无法确定所有者，调试器将传递给下一个堆栈帧，依此推，直到确定所有者（或完全检查堆栈）。 在此搜索中找到其所有者的第一个帧被视为有故障。 如果堆栈已用尽且未找到任何信息，则不会显示“跟进”字段。
4. 故障帧的所有者显示在“跟进”字段中。 如果使用 **！ analyze -v** ，则FOLLOWUP_IP、SYMBOL_NAME、MODULE_NAME、IMAGE_NAME和DBG_FLR_IMAGE_TIMESTAMP字段将引用此帧。

要使后续字段显示有用的信息，必须先创建一个包含模块名称和函数所有者名称的triage.ini文件。

triage.ini文件应标识可能出错的所有模块的所有者。 可以使用信息字符串而不是实际所有者，但此字符串不能包含空格。 如果确定某个模块不会出错，可以省略此模块或指示应跳过该模块。 还可以指定单个函数的所有者，从而为会审过程提供更精细的粒度。

有关triage.ini文件的语法的详细信息，请参阅 指定模块和函数所有者。

其他！ analyze 技术

如果未发生崩溃或异常， **！ analyze** 将显示一个非常短的文本，提供目标的当前状态。 在某些情况下，你可能希望强制分析发生，就像发生崩溃一样。 使用 **！ analyze -f** 完成

此任务。

在用户模式下，如果发生了异常，但你认为基础问题是挂起的线程，请将当前线程设置为正在调查的线程，然后使用 **！ analyze -hang。** 此扩展将执行线程堆栈分析，以确定任何线程是否阻止其他线程。

在内核模式下，如果发生了 bug 检查，但你认为基础问题是挂起线程，请使用 **！ analyze -hang。** 此扩展将调查系统持有的锁并扫描 DPC 队列链，并显示任何挂起线程的指示。 如果认为问题是内核模式资源死锁，请使用 **！ 死锁扩展以及驱动程序验证器的死锁检测** 选项。

还可以自动忽略已知问题。 为此，必须先创建包含已知问题的格式化列表的 XML 文件。 **使用 ！ analyze -c -load** *KnownIssuesFile* 扩展加载此文件。 然后，当发生异常或中断时，请使用 **！ analyze -c** 扩展。 如果异常与某个已知问题匹配，则目标将恢复执行。 如果目标未恢复执行，则可以使用 **！ analyze -v** 来确定问题的原因。

另请参阅

有关其他信息，请参阅这些主题。

!analyze

错误检查代码参考

使用 Windows 调试器 (WinDbg) 进行故障转储分析

使用 WinDbg 分析内核模式转储文件

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

在启用驱动程序验证程序的情况下处理 Bug 检查

项目 • 2023/04/14

驱动程序验证程序 在运行时检测驱动程序错误。 可以使用驱动程序验证程序以及 ! analyze 调试器命令来检测和显示有关驱动程序中的错误的信息。

在 Windows 8 中，驱动程序验证程序已使用新功能（包括 DDI 符合性检查）进行了增强。 下面提供了一个演示 DDI 符合性检查的示例。

使用以下过程进行设置。

1. 在主机和目标计算机之间建立内核模式调试会话。
2. 在目标计算机上安装驱动程序。
3. 在目标计算机上，打开命令提示符窗口并输入命令 **验证程序**。 使用 驱动程序验证程序管理器 为驱动程序启用驱动程序验证程序。
4. 重新启动目标计算机。

当驱动程序验证程序检测到错误时，它会生成一个 bug 检查。 然后，Windows 进入调试器并显示错误的简短说明。 下面是驱动程序验证程序生成 bug 检查 DRIVER_VERIFIER_DETECTED_VIOLATION (C4) 的示例。

```
dbgcmd
```

```
Driver Verifier: Extension abort with Error Code 0x20005
Error String ExAcquireFastMutex should only be called at IRQL <= APC_LEVEL.

*** Fatal System Error: 0x000000c4

(0x0000000000020005,0xFFFFF88000E16F50,0x0000000000000000,0x0000000000000000
)

Break instruction exception - code 80000003 (first chance)

A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.

A fatal system error has occurred.

nt!DbgBreakPointWithStatus:
fffff802`a40ef930 cc              int     3
```

在调试器中，输入 ! analyze -v 以获取错误的详细说明。

```
dbgcmd
```

```
0: kd> !analyze -v
Connected to Windows 8 9200 x64 target at (Thu Oct 11 13:48:31.270 2012 (UTC
- 7:00)), ptr64 TRUE
Loading Kernel Symbols
...............................................................
...............................................................
..................
Loading User Symbols
...............................................
Loading unloaded module list
............
*******************************************************************************
***
*
*
*                        Bugcheck Analysis
*
*
*
*******************************************************************************
***

DRIVER_VERIFIER_DETECTED_VIOLATION (c4)
A device driver attempting to corrupt the system has been caught.  This is
because the driver was specified in the registry as being suspect (by the
administrator) and the kernel has enabled substantial checking of this
driver.
If the driver attempts to corrupt the system, bugchecks 0xC4, 0xC1 and 0xA
will
be among the most commonly seen crashes.
Arguments:
Arg1: 0000000000020005, ID of the 'IrqlExApcLte1' rule that was violated.
Arg2: fffff88000e16f50, A pointer to the string describing the violated rule
condition.
Arg3: 0000000000000000, An optional pointer to the rule state variable(s).
Arg4: 0000000000000000, Reserved (unused)

## Debugging Details:

...

DV_VIOLATED_CONDITION:  ExAcquireFastMutex should only be called at IRQL <=
APC_LEVEL.

DV_MSDN_LINK: !url https://go.microsoft.com/fwlink/p/?linkid=216022

DV_RULE_INFO: !ruleinfo 0x20005

BUGCHECK_STR:  0xc4_IrqlExApcLte1_XDV

DEFAULT_BUCKET_ID:  WIN8_DRIVER_FAULT

PROCESS_NAME:  TiWorker.exe
```

```
CURRENT_IRQL:  9
```

在前面的输出中，可以看到违反的规则 **IrqlExApcLte1** 的名称和说明，并且可以选择指向描述 IrqlExApcLte1 规则 (wdm) 的引用页的链接。 还可以选择调试器命令链接 **！ruleinfo 0x20005**以获取有关规则的信息。 在这种情况下，如果 IRQL) 中断请求级别 (大于 APC_LEVEL，则规则指出不能调用 ExAcquireFastMutex 。 输出显示当前 IRQL 为 9，在 wdm.h 中，可以看到APC_LEVEL的值为 1。 有关 IRQL 的详细信息，请参阅 管理硬件优先级。

！analyze -v 的输出继续执行堆栈跟踪和有关导致错误的代码的信息。 在以下输出中，可以看到MyDriver.sys的 **OnInterrupt** 例程称为 ExAcquireFastMutex。 **OnInterrupt** 是一个在大于 APC_LEVEL 的 IRQL 上运行的中断服务例程，因此此例程调用 ExAcquireFastMutex 是一种冲突。

dbgcmd

```
LAST_CONTROL_TRANSFER:  from fffff802a41f00ea to fffff802a40ef930

STACK_TEXT:
... : nt!DbgBreakPointWithStatus ...
... : nt!KiBugCheckDebugBreak+0x12 ...
... : nt!KeBugCheck2+0x79f ...
... : nt!KeBugCheckEx+0x104 ...
... : VerifierExt!SLIC_abort+0x47 ...
... : VerifierExt!SLIC_ExAcquireFastMutex_entry_irqlexapclte1+0x25 ...
... : VerifierExt!ExAcquireFastMutex_wrapper+0x1a ...
... : nt!ViExAcquireFastMutexCommon+0x1d ...
... : nt!VerifierExAcquireFastMutex+0x1a ...
... : MyDriver!OnInterrupt+0x69 ...
... : nt!KiScanInterruptObjectList+0x6f ...
... : nt!KiChainedDispatch+0x19a ...
...

STACK_COMMAND:  kb

FOLLOWUP_IP:
MyDriver!OnInterrupt+69 ...
fffff880`16306649 4c8d0510040000  lea     r8,[MyDriver! ??
::FNODOBFM::`string' (fffff880`16306a60)]

FAULTING_SOURCE_LINE:  c:\MyDriverwdm03\cpp\MyDriver\pnp.c

FAULTING_SOURCE_FILE:  c:\MyDriverwdm03\cpp\MyDriver\pnp.c

FAULTING_SOURCE_LINE_NUMBER:  26

FAULTING_SOURCE_CODE:
    22:    if(1 == interruptStatus)
    23:    {
```

```
    24:       ...
    25:       ExAcquireFastMutex( &(fdoExt->fastMutex) );
>   26:       ...
    27:       ExReleaseFastMutex( &(fdoExt->fastMutex) );
    28:       ...
    29:       return TRUE;
    30:    }
    31:    else


SYMBOL_STACK_INDEX:  9

SYMBOL_NAME:  MyDriver!OnInterrupt+69

FOLLOWUP_NAME:  ...

MODULE_NAME: MyDriver

IMAGE_NAME:  MyDriver.sys

DEBUG_FLR_IMAGE_TIMESTAMP:  50772f37

BUCKET_ID_FUNC_OFFSET:  69

FAILURE_BUCKET_ID:  0xc4_IrqlExApcLte1_XDV_VRF_MyDriver!OnInterrupt

BUCKET_ID:  0xc4_IrqlExApcLte1_XDV_VRF_MyDriver!OnInterrupt
```

另请参阅

静态驱动程序验证程序

非侵入式调试（用户模式）

项目 • 2024/07/08

如果用户模式应用程序已在运行，调试器可以非侵入性地*调试它*。 使用非侵入性调试，无需执行任意数量的调试操作。 但是，可以将调试器对目标应用程序的干扰降到最低。 如果目标应用程序停止响应，则非侵入性调试非常有用。

在非侵入性调试中，调试器实际上不会附加到目标应用程序。 调试器会挂起所有目标线程，并有权访问目标的内存、寄存器和其他此类信息。 但是，调试器无法控制目标，因此 g（Go）**等**命令不起作用。

如果尝试执行在非侵入性调试期间不允许的命令，将收到一条错误消息，指出"调试器未附加，因此无法监视进程执行。

选择要调试的进程

可以通过进程 ID（PID）或进程名称指定目标应用程序。

如果按名称指定应用程序，则应使用进程的完整名称，包括文件扩展名。 如果两个进程具有相同的名称，则必须改用进程 ID。

有关如何确定进程 ID 和进程名称的详细信息，请参阅 查找进程 ID。

有关启动和停止非侵入性调试会话的信息，请参阅以下主题：

- 使用 WinDbg 调试用户模式进程
- 使用 CDB 调试用户模式进程

CDB 命令行

若要从 CDB 命令行以非侵入方式调试正在运行的进程，请在以下语法中指定 -pv 选项、-p 选项和进程 ID。

cdb -pv -p *ProcessID*

或者，若要通过指定进程名称以非侵入性方式调试正在运行的进程，请改用以下语法。

cdb -pv -pn *ProcessName*

还有其他几个有用的命令行选项。 有关命令行语法的详细信息，请参阅 **CDB 命令行选项**。

WinDbg 命令行

若要从 WinDbg 命令行以非侵入方式调试正在运行的进程，请在以下语法中指定 -pv 选项、-p 选项和进程 ID。

windbg -pv -p *ProcessID*

或者，若要通过指定进程名称以非侵入性方式调试正在运行的进程，请改用以下语法。

windbg -pv -pn *ProcessName*

还有其他几个有用的命令行选项。 有关命令行语法的详细信息，请参阅 **WinDbg 命令行选项**。

WinDbg 菜单

当 WinDbg 处于休眠模式时，可以通过单击“文件”菜单上的“附加到进程”或按 F6 来非侵入性地调试正在运行的进程。

出现“附加到进程”对话框时，选中“非侵入性”复选框。 然后选择包含所需进程 ID 和名称的行。 （还可以在“进程 ID”框中输入进程 ID。最后，单击“确定”。

调试器命令窗口

如果调试器已处于活动状态，则可以在调试器命令窗口中使用 .attach -v （附加到进程）**命令以非侵入性方式调试正在运行的进程。**

如果调试器已经以侵入性的方式调试了一个或多个进程，则可以使用 .attach 命令。 如果 CDB 处于休眠状态，但不能在休眠 WinDbg 中，则可以在 CDB 中使用此命令。

如果 .attach -v 命令成功，调试器将在下次调试器发出执行命令时调试指定的进程。 由于在非侵入性调试期间不允许执行，因此调试器不能一次非侵入性地调试多个进程。 此限制还意味着使用 .attach -v 命令可能会使现有入侵调试会话不那么有用。

启动调试会话

有关如何开始调试会话的详细信息，请参阅 调试器操作。

反馈

此页面是否有帮助? 是 否

在程序集模式下调试

项目 • 2023/04/14

如果应用程序有 C 或 C++ 源文件，则如果在 源模式下进行调试，则可以更强大的使用调试器。

但是，很多时候你无法执行源调试。 可能没有应用程序的源文件。 你可能在调试其他人的代码。 你可能没有使用完整的 .pdb 符号生成可执行文件。 即使可以对应用程序执行源调试，也可能必须跟踪应用程序调用或用于加载应用程序的 Microsoft Windows 例程。

在这些情况下，必须在程序集模式下进行调试。 此外，程序集模式具有许多在源调试中不存在的有用功能。 调试器会在访问内存位置时自动显示内存位置的内容和寄存器，并显示程序计数器的地址。 此显示使程序集调试成为可与源调试一起使用的宝贵工具。

反汇编代码

调试器主要分析二进制可执行代码。 调试器不以原始格式显示此代码，而是 *反汇编* 此代码。 也就是说，调试器将代码从计算机语言转换为汇编语言。

可以通过几种不同的方式显示生成的代码 (称为 *反汇编代码*) ：

- **u (Unassemble)** 命令反汇编并显示计算机语言的指定部分。
- **uf (Unassemble 函数)** 命令反汇编并显示函数。
- **向上 (从物理内存中取消汇编)** 命令反汇编，并显示已存储在物理内存中的指定计算机语言部分。
- **(Unassemble Real Mode BIOS)** 命令反汇编并显示指定的 16 位实模式代码。
- **ux (Unassemble x86 BIOS)** 命令反汇编，并在指定地址显示基于 x86 的 BIOS 代码指令集。
- (WinDbg 仅) 反汇编窗口反汇编，并显示计算机语言的指定部分。 如果在窗口菜单中选择“ **自动打开反汇编** ”命令，则此 **窗口** 将自动处于活动状态。 也可以打开此窗口，方法是在**视图**菜单上选择**反汇编**，按 alt+7，或按**反汇编 (alt+7)** 按钮 (反) WinDbg 工具栏上。

反汇编显示显示在四列中：地址偏移量、二进制代码、汇编语言助记符和汇编语言详细信息。 以下示例演示此显示。

```
dbgcmd
```

```
0040116b    45          inc         ebp
0040116c    fc          cld
0040116d    8945b0      mov         eax,[ebp-0x1c]
```

在表示当前程序计数器的行的右侧，显示正在访问的任何内存位置或寄存器的值。 如果此行包含分支指令，则显示表示法 [br=1] 或 [br=0] 。 此表示法分别指示已采用或未采用的分支。

可以使用 .asm (**更改反汇编选项**) 命令更改反汇编指令的显示方式。

在 WinDbg 的反汇编窗口中，突出显示表示当前程序计数器的行。 还突出显示了设置断点的行。

还可以使用以下命令操作程序集代码：

- # (**搜索反汇编模式**) 命令在内存区域中搜索特定模式。 此命令等效于搜索反汇编显示的四列。
- (Assemble) 命令可以接受程序集指令并将其转换为二进制计算机代码。

程序集模式和源模式

调试器有两种不同的操作模式： *程序集模式* 和 *源模式*。

单步执行应用程序时，单步执行的大小是一行程序集代码或一行源代码，具体取决于模式。

多个命令根据模式创建不同的数据显示。

在 WinDbg 中，在程序集模式下运行或单步执行应用程序时， 反汇编窗口 会自动移动到前台。 在源模式下， “源”窗口 移动到前台。

若要设置模式，可以执行下列操作之一：

- 使用 l+、l- (**设置源选项**) 命令来控制模式。 l-t 命令激活程序集模式。
- (WinDbg 仅) “**调试**”菜单上的“清除**源模式**”命令，导致调试器进入程序集模式。还可以选择**源模式关闭**按钮 ( ) 工具栏上。

在 WinDbg 中，处于程序集模式时， ASM 会显示在状态栏上。

WinDbg 反汇编窗口中的快捷菜单包括 **当前源行命令中的“突出显示”指令** 。 此命令突出显示与当前源行对应的所有指令。 通常，单个源行对应于多个程序集指令。 如果代码已优化，则这些程序集指令可能不是连续的。 使用 “**突出显示当前源行中的指令** ”命令，可以查找从当前源行组装的所有指令。

程序集语言源文件

如果应用程序是使用汇编语言编写的，则调试器生成的反汇编可能与原始代码不完全匹配。 特别是，NO-OPs和注释将不存在。

如果要通过引用原始 .asm 文件来调试代码，则必须使用源模式调试。 可以像 C 或 C++ 源文件一样加载程序集文件。 有关此类调试的详细信息，请参阅 在源模式下调试。

在源模式下调试

项目 • 2023/10/26

如果可以分析代码的源，而不是反汇编的二进制文件，则调试应用程序会更容易。

如果源语言为 C、C++或程序集，WinDbg、CDB 和 KD 可以在调试中使用源代码。

编译要求

若要使用源调试，必须在生成二进制文件时) 编译器或链接器 (.pdb 文件创建符号文件。 这些符号文件显示调试器二进制指令如何对应于源行。

此外，调试器必须能够访问实际的源文件，因为符号文件不包含实际的源文本。

如果可能，编译器和链接器不应优化代码。 如果代码已优化，则源调试和访问本地变量更加困难，有时几乎是不可能的。 如果使用生成实用工具作为编译器和链接器，请将 MSC_OPTIMIZATION 宏设置为 **/Od /Oi** 以避免优化。

查找符号文件和源文件

若要在源模式下进行调试，调试器必须能够找到源文件和符号文件。 有关详细信息，请参阅源路径。

开始源调试

每当调试器具有当前正在调试的线程的适当符号和源文件时，调试器都可以显示源信息。

如果使用调试器启动新的用户模式应用程序，则 Ntdll.dll 加载应用程序时会发生初始中断。 由于调试器无权访问 Ntdll.dll 源文件，因此此时无法访问应用程序的源信息。

若要将程序计数器移动到应用程序的开头，请将入口点处的断点添加到二进制文件。 在“调试器命令”窗口中，键入以下命令。

```
dbgcmd
```

```
bp main
g
```

然后加载应用程序，并在输入**main**函数时停止。 当然， (可以使用任何入口点，而不仅仅是 **main**.)

如果应用程序引发异常，它会中断调试器。 此时可以使用源信息。 但是，如果使用 CTRL+C、 CTRL+BREAK 或调试发出中断 |Break 命令，调试器会创建新线程，因此看不到源代码。

到达具有源文件的线程后，可以使用“调试器命令”窗口执行源调试命令。 如果使用 WinDbg，则会显示 “源”窗口 。 如果已通过**单击“文件”菜单上的“打开源文件”打开了“源”**窗口，则 WinDbg 通常会为源创建新窗口。 可以在不影响调试过程的情况下关闭上一个窗口。

WinDbg GUI 中的源调试

如果使用 WinDbg，只要程序计数器位于调试器具有其源信息的代码中，就会立即显示“源”窗口。

WinDbg 为你或 WinDbg 打开的每个源文件显示一个“源”窗口。 有关此窗口的文本属性的详细信息，请参阅 源 Windows。

然后，可以单步执行应用程序或执行到断点或游标。 有关单步执行和跟踪命令的详细信息，请参阅 控制目标。

如果处于源模式，则单步执行应用程序时，相应的“源”窗口将移动到前台。 由于还有在应用程序执行期间调用的 Microsoft Windows 例程，因此当此类调用发生时，调试器可能会将 反汇编窗口 移动到前台 (，因为调试器无权访问这些函数) 的源。 当程序计数器返回到已知的源文件时，相应的“源”窗口将变为活动状态。

在应用程序中移动时，WinDbg 会在“源”窗口和“反汇编”窗口中突出显示你的位置。 还突出显示了设置断点的行。 源代码根据语言分析进行着色。 如果已选择“源”窗口，可以使用鼠标将鼠标悬停在符号上来计算它。 有关这些功能以及如何控制这些功能的详细信息，请参阅 源 Windows。

若要在 WinDbg 中激活源模式，请使用 I+t 命令，单击**调试**菜单上的**源模式**，或单击**源模式上的**按钮。 当源模式处于活动状态时， ASM 指示器在状态栏上显示为不可用。

在源模式下单步执行函数时，可以查看或更改任何局部变量的值。 有关详细信息，请参阅 读取和写入内存。

调试器命令窗口中的源调试

如果使用 CDB，则没有单独的“源”窗口。 但是，在单步执行源时，仍可以查看进度。

在 CDB 中执行源调试之前，必须通过发出 **.lines (切换源行支持)** 命令或通过使用 **-lines 命令行选项**启动调试器来加载源行符号。

如果执行 l+t 命令，则所有程序单步执行一次一个源行。 使用 l-t 一次执行一个程序集指令。 如果使用 WinDbg，此命令的效果与在“**调试**”菜单上选择或清除“**源模式**”或使用工具栏按钮的效果相同。

l+s 命令在提示符处显示当前源行和行号。 如果只想查看行号，请改用 l+l 。

如果使用 l+o 和 l+s，则在单步执行程序时仅显示源行。 程序计数器、反汇编代码和寄存器信息处于隐藏状态。 通过这种显示，可以快速单步执行代码并查看源。

可以使用 lsp (Set Source Line number) 命令指定单步执行应用程序时显示的源行数。

以下命令序列是单步执行源文件的有效方法。

text

```
.lines        enable source line information
bp main       set initial breakpoint
l+t           stepping will be done by source line
l+s           source lines will be displayed at prompt
g             run program until "main" is entered
pr            execute one source line, and toggle register display off
p             execute one source line
```

由于 ENTER 重复上一个命令，因此现在可以使用 Enter 键单步执行应用程序。 每个步骤都会显示源行、内存偏移量和程序集代码。

有关如何解释反汇编显示的详细信息，请参阅 在程序集模式下调试。

显示程序集代码时，正在访问的任何内存位置将显示在行的右端。 可以使用 d* (**显示内存**) 和 e* (**输入值**) 命令查看或更改这些位置的值。

如果必须查看每个程序集指令以确定偏移量或内存信息，请使用 l-t 逐步执行程序集指令，而不是源行。 仍可显示源行信息。 每个源行对应于一个或多个程序集指令。

所有这些命令在 WinDbg 和 CDB 中都可用。 可以使用命令从 WinDbg 的 “调试器命令”窗口 而不是“源”窗口查看源行信息。

源行和偏移量

还可以通过使用表达式计算器来确定对应于特定源行的偏移量来执行源调试。

以下命令显示内存偏移量。

dbgcmd

```
? `[[module!]filename][:linenumber]`
```

如果省略 *文件名*，调试器将搜索对应于当前程序计数器的源文件。

调试器以十进制数形式读取 *亚麻小数* ，除非在它前面添加 **0x** ，而不考虑当前的默认基数。 如果省略 *亚麻*，则表达式的计算结果为对应于源文件的可执行文件的初始地址。

仅当 **.lines** 命令或 - lines 命令行选项已加载源行符号时，CDB 才会理解此语法。

此方法非常通用，因为无论程序计数器指向何处，都可以使用它。 例如，使用此方法，可以使用以下命令提前设置断点。

```
dbgcmd
```

```
bp `source.c:31`
```

有关详细信息，请参阅 源行语法 和使用 断点。

源模式下的单步执行和跟踪

在源模式下调试时，单个源行上可以有多个函数调用。 不能使用 **p** 和 **t** 命令来分隔这些函数调用。

例如，在以下命令中， **t** 命令逐步进入 **GetTickCount** 和 **printf**，而 **p** 命令将逐步执行这两个函数调用。

```
C++
```

```
printf( "%x\n", GetTickCount() );
```

如果要在跟踪到其他调用时单步执行某些调用，请使用 **.step_filter (设置步骤筛选器)** 来指示要单步执行哪些调用。

可以使用 **_step_filter** 筛选出框架函数 (，例如 Microsoft 基础类 (MFC) 或活动模板库 (ATL) 调用) 。

调试优化的代码和内联函数

项目 • 2023/11/15

对于Windows 8，调试器和 Windows 编译器已得到增强，以便可以调试优化的代码和调试内联函数。 调试器显示参数和局部变量，无论它们是存储在寄存器中还是存储在堆栈中。 调试器还会在调用堆栈中显示内联函数。 对于内联函数，调试器显示局部变量，但不显示参数。

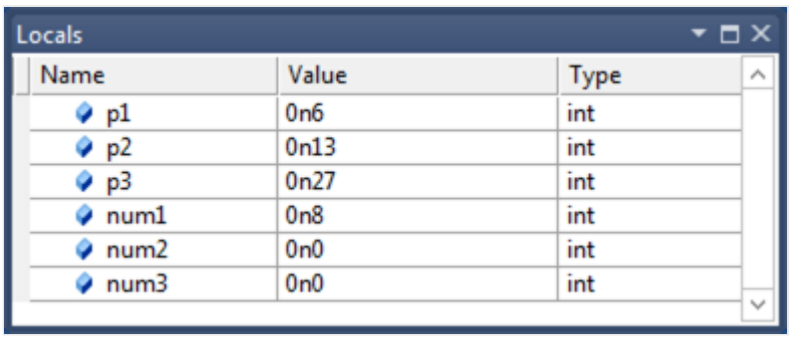
优化代码后，将转换代码以更快地运行并使用更少的内存。 有时，由于删除死代码、合并代码或将函数置于内联中，函数会被删除。 也可以删除局部变量和参数。 许多代码优化会删除不需要或使用的局部变量;其他优化删除循环中的感应变量。 公共子表达式消除将局部变量合并在一起。

Windows 的零售版本已优化。 因此，如果你运行的是 Windows 的零售版本，则拥有一个旨在与优化代码很好地配合使用的调试器尤其有用。 若要使优化代码的调试有效，需要两个主要功能：1) 本地变量的准确显示，2) 调用堆栈上内联函数的显示。

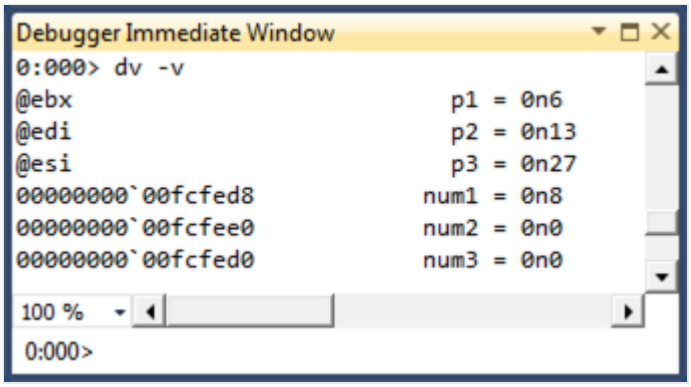
准确显示局部变量和参数

为了便于准确显示局部变量和参数，编译器将有关局部变量和参数的位置的信息记录在符号 (PDB) 文件中。 这些位置记录跟踪变量的存储位置和这些位置有效的特定代码范围。 这些记录不仅有助于跟踪变量) 寄存器或堆栈槽中 (的位置，还有助于跟踪变量的移动。 例如，参数可能首先位于 register RCX 中，但会移动到堆栈槽以释放 RCX，然后在循环中大量使用时移动到注册 R8，然后在代码退出循环时移动到不同的堆栈槽。 Windows 调试器使用 PDB 文件中的丰富位置记录，并使用当前指令指针为局部变量和参数选择适当的位置记录。

Visual Studio 中“局部变量”窗口的屏幕截图显示了优化 64 位应用程序中函数的参数和局部变量。 函数不是内联函数，因此我们看到了参数和局部变量。



可以使用 dv -v 命令查看参数和局部变量的位置。



请注意，即使参数存储在寄存器中，“局部变量”窗口也会正确显示参数。

除了使用基元类型的跟踪变量外，位置还记录本地结构和类的跟踪数据成员。 以下调试器输出显示本地结构。

dbgcmd

```
0:000> dt My1
Local var Type _LocalStruct
   +0x000 i1                : 0n0 (edi)
   +0x004 i2                : 0n1 (rsp+0x94)
   +0x008 i3                : 0n2 (rsp+0x90)
   +0x00c i4                : 0n3 (rsp+0x208)
   +0x010 i5                : 0n4 (r10d)
   +0x014 i6                : 0n7 (rsp+0x200)

0:000> dt My2
Local var @ 0xefa60 Type _IntSum
   +0x000 sum1              : 0n4760 (edx)
   +0x004 sum2              : 0n30772 (ecx)
   +0x008 sum3              : 0n2 (r12d)
   +0x00c sum4              : 0n0
```

下面是有关上述调试器输出的一些观察结果。

- 本地结构 **My1** 说明编译器可以将本地结构数据成员分散到寄存器和非连续堆栈槽。
- 命令 **dt My2** 的输出将与命令 **dt _IntSum 0xefa60** 的输出不同。 不能假定本地结构将占用连续的堆栈内存块。 对于 **My2**，仅 `sum4` 保留在原始堆栈块中;其他三个数据成员将移动到寄存器。
- 某些数据成员可以有多个位置。 例如， **My2.sum2** 有两个位置：一个是注册 ECX (Windows 调试器选择)，另一个是0xefa60+0x4 (原始堆栈槽) 。 基元类型局部变量也可能发生这种情况，Windows 调试器会施加先例启发法来确定要使用的位置。 例如，注册位置始终优先于堆栈位置。

在调用堆栈上显示内联函数

在代码优化期间，某些函数将按行放置。 也就是说，函数的主体直接放置在代码中，就像宏扩展一样。 没有函数调用，也没有返回到调用方。 为了便于显示内联函数，编译器将数据存储在 PDB 文件中，这有助于解码内联函数的代码块 (即，属于内联) 被调用方函数的调用方函数中的代码块序列，以及本地变量 (这些代码块) 的作用域局部变量。 此数据可帮助调试器在堆栈展开过程中包括内联函数。

假设编译了一个应用程序，并强制将名为 的 `func1` 函数内联。

C++

```
__forceinline int func1(int p1, int p2, int p3)
{
   int num1 = 0;
   int num2 = 0;
   int num3 = 0;
   ...
}
```

可以使用 **bm** 命令在 处设置断点 `func1`。

dbgcmd

```
0:000> bm MyApp!func1
  1: 000007f6`8d621088 @!"MyApp!func1" (MyApp!func1 inlined in
MyApp!main+0x88)
0:000> g

Breakpoint 1 hit
MyApp!main+0x88:
000007f6`8d621088 488d0d21110000  lea     rcx,[MyApp!`string'
(000007f6`8d6221b0)]
```

执行 `func1`一个步骤后，可以使用 **k** 命令在调用堆栈上查看 `func1` 。 可以使用 **dv** 命令查看 的 `func1`局部变量。 请注意，局部变量 `num3` 显示为不可用。 由于多种原因，局部变量在优化代码中不可用。 可能是该变量不存在于优化代码中。 可能是变量尚未初始化，或者该变量不再被使用。

dbgcmd

```
0:000> p
MyApp!func1+0x7:
000007f6`8d62108f 8d3c33          lea     edi,[rbx+rsi]

0:000> knL
# Child-SP          RetAddr           Call Site
00 (Inline Function) --------`-------- MyApp!func1+0x7
01 00000000`0050fc90 000007f6`8d6213f3 MyApp!main+0x8f
02 00000000`0050fcf0 000007ff`c6af0f7d MyApp!__tmainCRTStartup+0x10f
```

```
03 00000000`0050fd20 000007ff`c7063d6d KERNEL32!BaseThreadInitThunk+0xd
04 00000000`0050fd50 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

0:000> dv -v
00000000`0050fcb0            num1 = 0n0
00000000`0050fcb4            num2 = 0n0
<unavailable>                num3 = <value unavailable>
```

如果查看堆栈跟踪中的第 1 帧，可以看到函数的 `main` 局部变量。 请注意，其中两个变量存储在寄存器中。

dbgcmd

```
0:000> .frame 1
01 00000000`0050fc90 000007f6`8d6213f3 MyApp!main+0x8f

0:000> dv -v
00000000`0050fd08               c = 0n7
@ebx                            b = 0n13
@esi                            a = 0n6
```

Windows 调试器聚合 PDB 文件中的数据，以查找特定函数已内联放置的所有位置。 可以使用 **x** 命令列出内联函数的所有调用方站点。

dbgcmd

```
0:000> x simple!MoreCalculate
00000000`ff6e1455 simple!MoreCalculate =  (inline caller) simple!wmain+8d
00000000`ff6e1528 simple!MoreCalculate =  (inline caller) simple!wmain+160

0:000> x simple!Calculate
00000000`ff6e141b simple!Calculate =  (inline caller) simple!wmain+53
```

由于 Windows 调试器可以枚举内联函数的所有调用方站点，因此它可以通过计算与调用方站点的偏移量来设置内联函数内的断点。 可以使用 **bm** 命令 (它用于设置与正则表达式模式匹配的断点，) 为内联函数设置断点。

Windows 调试器将为特定内联函数设置的所有断点分组到断点容器中。 可以使用 **be**、**bd**、 **bc** 等命令将断点容器作为一个整体进行操作。 请参阅以下 **bd 3** 和 **bc 3** 命令示例。 还可以操作单个断点。 请参阅下面的 **2** 命令示例。

dbgcmd

```
0:000> bm simple!MoreCalculate
  2: 00000000`ff6e1455 @!"simple!MoreCalculate" (simple!MoreCalculate
inlined in simple!wmain+0x8d)
  4: 00000000`ff6e1528 @!"simple!MoreCalculate" (simple!MoreCalculate
inlined in simple!wmain+0x160)
```

```
0:000> bl
 0 e 00000000`ff6e13c8 [n:\win7\simple\simple.cpp @ 52]    0001 (0001)
0:**** simple!wmain
 3 e <inline function>     0001 (0001)  0:**** {simple!MoreCalculate}
     2 e 00000000`ff6e1455 [n:\win7\simple\simple.cpp @ 58]    0001 (0001)
0:**** simple!wmain+0x8d (inline function simple!MoreCalculate)
     4 e 00000000`ff6e1528 [n:\win7\simple\simple.cpp @ 72]    0001 (0001)
0:**** simple!wmain+0x160 (inline function simple!MoreCalculate)

0:000> bd 3
0:000> be 2

0:000> bl
 0 e 00000000`ff6e13c8 [n:\win7\simple\simple.cpp @ 52]    0001 (0001)
0:**** simple!wmain
 3 d <inline function>     0001 (0001)  0:**** {simple!MoreCalculate}
     2 e 00000000`ff6e1455 [n:\win7\simple\simple.cpp @ 58]    0001 (0001)
0:**** simple!wmain+0x8d (inline function simple!MoreCalculate)
     4 d 00000000`ff6e1528 [n:\win7\simple\simple.cpp @ 72]    0001 (0001)
0:**** simple!wmain+0x160 (inline function simple!MoreCalculate)

0:000> bc 3

0:000> bl
 0 e 00000000`ff6e13c8 [n:\win7\simple\simple.cpp @ 52]    0001 (0001)
0:**** simple!wmain
```

由于内联函数没有显式调用或返回指令，因此源级单步执行对于调试器来说尤其具有挑战性。 例如，如果下一个指令是内联函数) 的一部分，则可以无意中单步执行内联函数 (;或者，由于内联函数的代码块已被编译器) 拆分和移动，因此可以 (多次单步执行并单步执行同一内联函数。 为了保留熟悉的单步执行体验，Windows 调试器为每个代码指令地址维护一个小型概念调用堆栈，并生成一个内部状态机来执行分步、单步执行和分步操作。 这为非内联函数的步进体验提供了相当准确的近似值。

其他信息

注意 可以使用 **.inline 0** 命令禁用内联函数调试。 **.inline 1** 命令启用内联函数调试。 标准调试方法

另请参阅

标准调试方法

使用 Windows 调试器调试托管代码

项目 • 2024/10/16

可以使用 Windows 调试器（WinDbg、CDB 和 NTSD）调试包含托管代码的目标应用程序。 若要调试托管代码，请将 !SOS 调试扩展 (sos.dll) 和数据访问组件 (mscordacwks.dll) 与 CLR 运行时结合使用。

Windows 调试器（如 WinDbg）与 Visual Studio 调试器相互独立。 有关 Windows 调试器与 Visual Studio 调试器之间的区别的信息，请参阅适用于 Windows 的调试工具。

本文提供有关使用 Windows 调试器（WinDbg、CDB、NTSD）调试托管代码（包括 .NET Framework、.NET Core 和 .NET 5+ 应用程序）的说明。

> **备注**
>
> 请确保使用最新版本的 Windows 调试器工具调试 .NET Framework、.NET Core 和 .NET 5+ 应用程序。 此外，请考虑使用 Visual Studio 或 Visual Studio Code 获得集成度更高的调试体验。 WinDbg 更加复杂，设置起来更加麻烦，并且通常在需要其他低级别信息时使用。

托管代码简介

托管代码与 Microsoft .NET 公共语言运行时 (CLR) 一起执行。 在托管代码应用程序中，编译器生成的二进制代码采用与平台无关的 Microsoft 中间语言 (MSIL)。

运行托管代码时，运行时将生成特定于平台的本机代码。 从 MSIL 生成本机代码的过程称为*实时 (JIT) 编译*。 在 JIT 编译器编译特定方法的 MSIL 后，该方法的本机代码将保留在内存中。 稍后调用此方法时，就会执行本机代码，并且不需要 JIT 编译器。

可以使用由各种软件生成者制造的多个编译器来生成托管代码。 具体而言，Microsoft Visual Studio 可以从多种不同的语言（包括 C#、Visual Basic、JScript 和具有托管扩展的 C++）生成托管代码。

每次更新 .NET Framework 时，CLR 都不会更新。 例如，.NET Framework 的 2.0、3.0 和 3.5 版都使用 CLR 的 2.0 版。 有关 .NET 版本的详细信息，请参见 .NET Framework 版本和依赖关系。 有关确定 PC 上的 .NET 版本的信息，请参阅确定安装了哪些 .NET Framework 版本。

调试托管代码

若要使用 !SOS 调试扩展调试托管代码，调试器必须加载各种组件。 用于不同 .NET Core 和原始 .NET Framework 的 !SOS 调试扩展和所需组件。 对于任一项，都将使用数据访问组件 (DAC) (mscordacwks.dll)。

.NET SDK 提供有助于调试 .NET 应用的工具。 有关详细信息，请参阅什么是 .NET SDK? 。

.NET Core

- 对于 .NET Core 或 .NET 5+ 及更高版本，运行时为 `coreclr.dll`。 有关详细信息，请参阅公共语言运行时 (CLR) 概述。
- .NET Core SOS 调试扩展

对于 .NET Core，有一个可用于安装 !sos.dll 的 dotnet CLI 工具。 有关详细信息，请参阅 SOS 安装程序 (dotnet-sos)。

原始 .NET Framework。

- 对于原始 .NET Framework，`clr.dll` 是运行时。
- SOS 调试扩展 (sos.dll)

获取 SOS 调试扩展 (sos.dll)

SOS 调试扩展 (sos.dll) 文件不包含在所有版本的 Windows 调试工具中。 如果 sos.dll 不可用，请参阅在 Windows 上安装 SOS 。

加载 SOS 调试扩展 (sos.dll)

若要调试 .NET Core 和 .NET 5+ 应用程序，需要加载适当的 SOS 调试扩展版本。

例如，对于调试器随附的包含在当前扩展搜索路径中的 !SOS 版本，将使用 **.load** 命令。

```
0:000> .load sos.dll
```

若要验证 SOS 调试扩展加载是否正确，请使用 **.chain** 命令并检查*扩展 DLL 链*。

dbgcmd

```
...
Extension DLL chain:
    C:\Windows\Microsoft.NET\Framework\v4.0.30319\SOS.dll: image 4.8.9275.0,
API 1.0.0, built Wed Aug 28 14:43:27 2024
        [path: C:\Windows\Microsoft.NET\Framework\v4.0.30319\SOS.dll]
    C:\Program Files
```

```
(x86)\dotnet\shared\Microsoft.NETCore.App\8.0.8\coreclr.dll: image
8,0,824,36612 @Commit: 08338fcaa5c9b9a8190abb99222fed12aaba956c, built Tue
Jul 16 11:10:19 2024
        [path: C:\Program Files
(x86)\dotnet\shared\Microsoft.NETCore.App\8.0.8\coreclr.dll]
```

如果调试器版本不包含 sos.dll，则可能需要指定 SOS.dll 文件的完整路径。 通常可以在 .NET Core 或 .NET Framework 安装的运行时目录中找到 SOS.dll 文件。

```
0:000> .load C:\Windows\Microsoft.NET\Framework64\v4.0.30319\sos.dll
```

加载特定版本的 sos.dll

加载 sos.dll 可能很复杂，因为依赖于 sos.dll 用来与 .NET 通信的其他 DLL。 此外，所需的 DLL 版本依赖于正在调试的应用的 .NET 版本，并且计算机上可能存在多个版本的 .NET。

加载依赖 DLL 的正确版本的一种策略是要求调试器使用 sx、sxd、sxe、sxi、sxn、sxr、sx- (Set Exceptions)[../debuggercmds/sx--sxd--sxe--sxi--sxn--sxr--sx---set-exceptions-.md] 命令在第一个 .NET clr 通知 (CLRN) 出现时中断操作。 附加到目标 .NET 应用程序后，将在第一次中断后使用此命令。

```
0:000> sxe CLRN
```

接下来继续执行并等待中断发生。

```
0:000> g
```

中断发生时，禁用 clr 通知中断，因为我们知道已加载 clr.dll（或 coreclr.dll）。

```
0:000> sxd CLRN
```

在此上下文中使用调试器时，使用 **.loadby** 从同一“附近”目录位置加载 !sos。

```
0:000> .loadby sos clr
```

另一个选项是使用 .cordll（控制 CLR 调试）通过提供目标框架位置的路径来加载 CLR 调试 DLL。

dbgcmd

```
0:000> .cordll -ve -u -lp C:\Windows\Microsoft.NET\Framework\v4.0.30319\
CLRDLL: Loaded DLL
C:\Windows\Microsoft.NET\Framework\v4.0.30319\mscordacwks.dll
Automatically loaded SOS Extension
```

```
CLR DLL status: Loaded DLL
C:\Windows\Microsoft.NET\Framework\v4.0.30319\mscordacwks.dll
```

使用 SOS 调试扩展

若要验证 SOS 调试扩展是否正确加载，请输入 .chain 命令。

dbgcmd

```
0:000> .chain
Extension DLL search Path:
    C:\Program Files\Debugging Tools for Windows (x64);...
Extension DLL chain:
    C:\Windows\Microsoft.NET\Framework\v4.0.30319\SOS.dll: image 4.8.9275.0,
API 1.0.0, built Wed Aug 28 14:43:27 2024
        [path: C:\Windows\Microsoft.NET\Framework\v4.0.30319\SOS.dll]
...
```

.NET 符号文件

符号文件对于调试至关重要。 对于 .NET Framework、.NET Core 和 .NET 5+ 应用程序，可以从 Microsoft 的公共符号服务器检索必要的符号文件。 使用以下命令设置符号路径和回显符号加载。

`.symfix`

`!sym noisy`

`.reload`

测试 .NET Core !sos 扩展

若要测试 SOS 调试扩展，请输入 **!sos.help**。

dbgcmd

```
0:000> !sos.help
-------------------------------------------------------------------------------
---
SOS is a debugger extension DLL designed to aid in the debugging of managed
programs. Functions are listed by category, then roughly in order of
importance. Shortcut names for popular functions are listed in parenthesis.
Type "!help <functionname>" for detailed info on that function.

Object Inspection                  Examining code and stacks
-----------------------------      -----------------------------
```

```
DumpObj (do)                        Threads
DumpArray (da)                      ThreadState
DumpStackObjects (dso)              IP2MD
DumpHeap                            U
DumpVC                              DumpStack
GCRoot                              EEStack
ObjSize                             CLRStack
FinalizeQueue                       GCInfo
PrintException (pe)                 EHInfo
TraverseHeap                        BPMD
                                    COMState
```

然后尝试 SOS 调试扩展提供的命令之一。 例如，可以尝试 .NET Core SOS 调试扩展提供的 **!sos.DumpDomain** 或 **!sos.Threads** 命令。

dbgcmd

```
0:000> !sos.DumpDomain
-----------------------------------
System Domain:      7565d980
LowFrequencyHeap:   7565dca4
HighFrequencyHeap:  7565dcf0
StubHeap:           7565dd3c
Stage:              OPEN
Name:               None
-----------------------------------
Shared Domain:      7565d620
LowFrequencyHeap:   7565dca4
HighFrequencyHeap:  7565dcf0
StubHeap:           7565dd3c
Stage:              OPEN
Name:               None
Assembly:           00fa5e78
[C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c56193
4e089\mscorlib.dll]
ClassLoader:        00fa5f40
  Module Name
73571000
C:\WINDOWS\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934
e089\mscorlib.dll

0:000> !sos.Threads
ThreadCount:      2
UnstartedThread:  0
BackgroundThread: 2
PendingThread:    0
DeadThread:       0
Hosted Runtime:   no

Lock
       ID OSID ThreadOBJ    State GC Mode     GC Alloc Context  Domain
Count Apt Exception
   0    1 4538 00f91110     20220 Preemptive  02FE1238:00000000 00f58be0 1
```

```
Ukn
   7    2 250c 00f9da88     21220 Cooperative 00000000:00000000 00f58be0 1
Ukn (Finalizer)
```

测试 .NET Framework !sos 扩展

若要测试 SOS 调试扩展，请输入 **!sos.help**。 然后尝试 SOS 调试扩展提供的命令之一。 例如，可以尝试 **!sos.sostatus** 或 **!sos.threads** 命令。

dbgcmd

```
0:030> !soshelp
crashinfo                                  Displays the crash details that
created the dump.
help, soshelp <command>                    Displays help for a command.
loadsymbols <url>                          Loads symbols for all modules.
logclose <path>                            Disables console file logging.
logging <path>                             Enables/disables internal
diagnostic logging.
logopen <path>                             Enables console file logging.
maddress                                   Displays a breakdown of the
virtual address space.
modules, lm                                Displays the native modules in the
process.
registers, r                               Displays the thread's registers.
runtimes <id>                              Lists the runtimes in the target
or changes the default runtime.
setclrpath <path>                          Sets the path to load coreclr
DAC/DBI files.
setsymbolserver, SetSymbolServer <url>     Enables and sets symbol server
support for symbols and module download.
sosflush                                   Resets the internal cached state.
sosstatus                                  Displays internal status.
threads, setthread <thread>                Lists the threads in the target or
sets the current thread.
```

备注

有时，托管代码应用程序会加载多个版本的 CLR。 在这种情况下，必须指定要加载的 DAC 版本。 有关详细信息，请参阅 **.cordll** 和 Clrver.exe（CLR 版本工具）。

反馈

此页面是否有帮助? 是 否

使用 Windows 调试器调试 Windows 应用

项目 • 2023/06/15

可以使用 Windows 调试器 (WinDbg、CDB 和 NTSD) 来调试 Windows 应用。 使用 **PLMDebug** 工具控制在调试时暂停、恢复和终止 Windows 应用。

若要调试托管代码 Windows 应用，请加载 SOS 调试扩展 (sos.dll) 和数据访问组件 (mscordacwks.dll) 。 有关详细信息，请参阅 使用 Windows 调试器调试托管代码。

Windows 调试器独立于 Visual Studio 调试器。 有关 Windows 调试器和 Visual Studio 调试器之间的区别的信息，请参阅 Windows 调试。

更改上下文

项目 • 2024/03/08

在内核模式调试中，有许多进程、线程，有时还有用户会话同时执行。 因此，诸如“虚拟地址 0x80002000”或“ **eax** 寄存器”之类的短语是不明确的。 必须指定能够理解此类短语的*上下文*。

调试器有五种不同的上下文，可以在调试时进行设置：

1. 会话上下文指示默认用户会话。
2. 进程上下文决定调试器如何解释虚拟地址。
3. *用户模式地址上下文*几乎从不直接设置。 更改进程上下文时，会自动设置此上下文。
4. 寄存器上下文决定调试器如何解释寄存器，并控制堆栈跟踪的结果。 此上下文也称为*线程上下文*，但该术语并不完全准确。 *显式上下文*也是寄存器上下文的类型。 如果指定显式上下文，则使用该上下文而不是当前寄存器上下文。
5. 本地上下文确定调试器如何解释局部变量。 此上下文也称为*作用域*。

会话上下文

多个登录会话可以同时运行。 每个登录会话都有自己的进程。

!session 扩展插件显示所有登录会话或更改当前会话上下文。

会话上下文由 **!sprocess** 和 **!spoolused** 扩展在会话编号输入为“-2”时使用。

当会话上下文发生更改时，进程上下文会自动更改为该会话的活动进程。

进程上下文

每个进程都有自己的页面目录，用于记录虚拟地址映射到物理地址的方式。 当进程中的任何线程正在执行时，Windows 操作系统使用此页目录来解释虚拟地址。

在用户模式调试期间，当前进程确定进程上下文。 调试器命令、扩展和调试信息窗口中使用的虚拟地址是通过使用当前进程的页面目录来解释。

在内核模式调试期间，可以使用 **.process（设置进程上下文）** 命令设置进程上下文。 使用此命令可以选择用于解释虚拟地址的进程的页面目录。 设置进程上下文后，可以在任

何获取地址的命令中使用此上下文。 甚至可以在此地址处设置断点。 通过在 .process 命令中包含 /i 选项以指定侵入调试，还可以使用内核调试器在用户空间中设置断点。

还可以通过在内核空间函数上使用进程特定的断点，从内核调试器设置用户模式断点。 设置战略性断点，等待合适的上下文出现。

*用户模式地址上下文*是进程上下文的一部分。 通常，无需直接设置用户模式地址上下文。 如果设置进程上下文，则用户模式地址上下文会自动更改为进程的相关页表的目录基。

当在内核模式调试期间设置进程上下文时，该进程上下文将被保留，直到另一个 .process 命令更改上下文。 在 .process 或 .context 命令更改之前，用户模式地址上下文也会保留。 当目标计算机执行时，这些上下文不会更改，也不受寄存器上下文或本地上下文更改的影响。

寄存器上下文

每个线程都有自己的寄存器值。 当一个线程正在执行时，这些值被存储在 CPU 寄存器中；而当另一个线程正在执行时，它们被存储在存储器中。

在用户模式调试期间，当前线程通常确定寄存器上下文。 调试器命令、扩展和调试信息窗口中对寄存器的任何引用都将根据当前线程的寄存器进行解释。

在执行用户模式调试时，可以使用以下命令之一将寄存器上下文更改为当前线程以外的值：

.cxr（显示上下文记录）

.ecxr（显示异常上下文记录）

在内核模式调试期间，可以使用各种调试器命令控制寄存器上下文，包括以下命令：

.thread（设置寄存器上下文）

.cxr（显示上下文记录）

.trap（显示陷阱帧）

这些命令不会更改 CPU 寄存器的值。 相反，调试器从内存中的位置检索指定的寄存器上下文。 实际上，调试器只能检索*保存的*寄存器值。 （其他值动态设置，不会保存。 保存的值足以重新创建堆栈跟踪。

设置寄存器上下文后，新寄存器上下文将用于使用寄存器值的任何命令，例如 **k（显示堆栈回溯）**和 **r（寄存器）**。

但是，在调试多处理器计算机时，某些命令允许指定处理器。 （有关此类命令的详细信息，请参阅 多处理器语法。）如果为命令指定处理器，则即使指定的处理器是当前活动的处理器，该命令也会使用指定处理器上活动线程的寄存器上下文，而不是当前寄存器上下文。

此外，如果寄存器上下文与当前处理器模式设置不匹配，则这些命令会产生不正确的或毫无意义的输出。 为了避免输出错误，依赖于寄存器状态的命令会失败，直到更改处理器模式以匹配寄存器上下文。 若要更改处理器模式，请使用 .effmach (Effective Machine) 命令，

更改寄存器上下文还会更改本地上下文。 通过这种方式，寄存器上下文可能会影响局部变量的显示。

如果发生任何应用程序执行、单步执行或跟踪，则会立即重置寄存器上下文以匹配程序计数器的位置。 在用户模式下，如果当前进程或线程发生更改，则寄存器上下文也会重置。

寄存器上下文会影响堆栈跟踪，因为堆栈跟踪从堆栈指针寄存器（x86 处理器上的 esp）指向的位置开始。 如果寄存器上下文设置为无效或不可访问的值，则无法获取堆栈跟踪。

可以使用 .apply_dbp（将数据断点应用到上下文）命令，将处理器断点（数据断点）应用到特定的寄存器上下文。

本地上下文

当程序运行时，局部变量的含义取决于程序计数器的位置，因为此类变量的范围仅扩展到定义它们的函数。

执行用户模式或内核模式调试时，调试器使用当前函数的作用域（堆栈上的当前帧）作为本地上下文。 若要更改此上下文，请使用 .frame（设置本地上下文）命令，或在“调用”窗口中双击所需的帧。

在用户模式调试中，本地上下文始终是当前线程堆栈跟踪中的帧。 在内核模式调试中，本地上下文始终是当前寄存器上下文线程的堆栈跟踪中的一个帧。

对于本地上下文，一次只能使用一个堆栈帧。 无法访问其他帧中的局部变量。

如果发生以下任何事件，则会重置本地上下文：

- 任何程序执行、单步执行或跟踪
- 在任何命令中使用线程分隔符 (~)
- 对寄存器上下文的任何更改

!for_each_frame 扩展使你能够重复执行单个命令，对堆栈中的每一帧执行一次。 此命令更改每个帧的本地上下文，执行指定的命令，然后将本地上下文返回到其原始值。

控制进程和线程

项目 • 2023/06/15

执行用户模式调试时，可以激活、显示、冻结、取消冻结、挂起和取消挂起进程和线程。

*当前*或*活动*进程是当前正在调试的进程。 同样， *当前* 线程或 *活动* 线程是调试器当前正在控制的线程。 许多调试器命令的操作由当前进程和线程的标识决定。 当前进程还确定调试器使用的虚拟地址映射。

调试开始时，当前进程是调试器附加到的进程，或者是导致异常进入调试器的进程。 同样，当前线程是调试器附加到进程时处于活动状态或导致异常的线程。 但是，可以使用调试器更改当前进程和线程，以及冻结或取消冻结单个线程。

在内核模式调试中，进程和线程不受本节中所述的方法控制。 有关如何在内核模式下操作进程和线程的详细信息，请参阅 更改上下文。

显示进程和线程

若要显示进程和线程信息，可以使用以下方法：

- **| (进程状态)** 命令
- **~ (线程状态)** 命令
- 仅) “进程和线程”窗口 (WinDbg

设置当前进程和线程

若要更改当前进程或线程，可以使用以下方法：

- **|s (设置当前进程)** 命令
- **~s (设置当前线程)** 命令
- 仅) “进程和线程”窗口 (WinDbg

冻结和挂起线程

调试器可以通过 *暂停* 线程或 *冻结* 线程来更改线程的执行。 这两个操作具有一些不同的效果。

每个线程都有一个与之关联的 *挂起计数* 。 如果此计数为 1 或更大，则系统不会运行线程。 如果计数为零或更低，则系统在适当时运行线程。

通常，每个线程的挂起计数为零。 当调试器附加到某个进程时，它将该进程中所有线程的挂起计数递增 1。 如果调试器从进程分离，则会将所有挂起计数减 1。 调试器执行进程时，它会暂时将所有挂起计数减一。

可以使用以下方法控制调试器中任何线程的挂起计数：

- **~n (挂起线程)** 命令将指定线程的暂停计数递增 1。
- **~m (恢复线程)** 命令会将指定线程的暂停计数减 1。

这些命令最常见的用途是将特定线程的暂停计数从 1 提高到 2。 当调试器执行或从进程分离时，线程的挂起计数为 1，并且即使进程中的其他线程正在执行，也会保持挂起状态。

即使在执行 非侵入性调试时，也可以挂起线程。

调试器还可以 *冻结* 线程。 此操作类似于以某种方式挂起线程。 但是，“冻结”只是调试器设置。 Windows 操作系统中的任何内容都不会识别到此线程有任何不同之处。

默认情况下，所有线程都是取消冻结的。 当调试器导致进程执行时，冻结的线程不会执行。 但是，如果调试器从进程分离，则所有线程都会解冻。

若要冻结和取消冻结单个线程，可以使用以下方法：

- **~f (冻结线程)** 命令冻结指定的线程。
- **~u (取消冻结线程)** 命令可解冻指定的线程。

在任何情况下，当调试器进入目标进程时，属于目标进程的线程永远不会执行。 仅当调试器执行进程或分离时，线程的挂起计数才会影响线程的行为。 仅当调试器执行进程时，冻结状态才影响线程的行为。

其他命令中的线程和进程

可以在许多其他命令之前添加线程说明符或进程说明符。 有关详细信息，请参阅各个命令主题。

可以在许多命令和扩展命令之前添加 **~e (线程特定的命令)** 限定符。 此限定符会导致针对指定的线程执行命令。 如果要将命令应用于多个线程，此限定符特别有用。 例如，以下命令对正在调试的每个线程重复 **！ gle** extension 命令。

```
dbgcmd
```

```
~*e !gle
```

多个系统

调试器可以同时附加到多个目标。 当这些进程包括转储文件或包含多台计算机上的实时目标时，调试器将为每个操作引用系统、进程和线程。 有关此类调试的详细信息，请参阅 调试多个目标。

使用调试器标记语言

项目 • 2024/07/08

调试器命令可以使用纯文本或增强格式（使用调试器标记语言（DML）提供输出。 使用 DML 增强的输出包括可以单击以执行相关命令的链接。

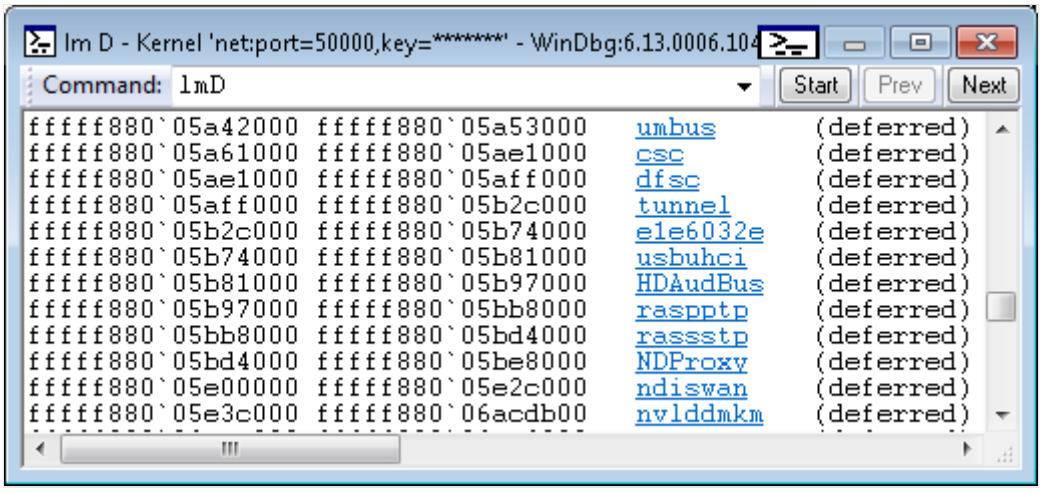
DML 在 Windows 10 及更高版本中可用。

支持 DML 的命令

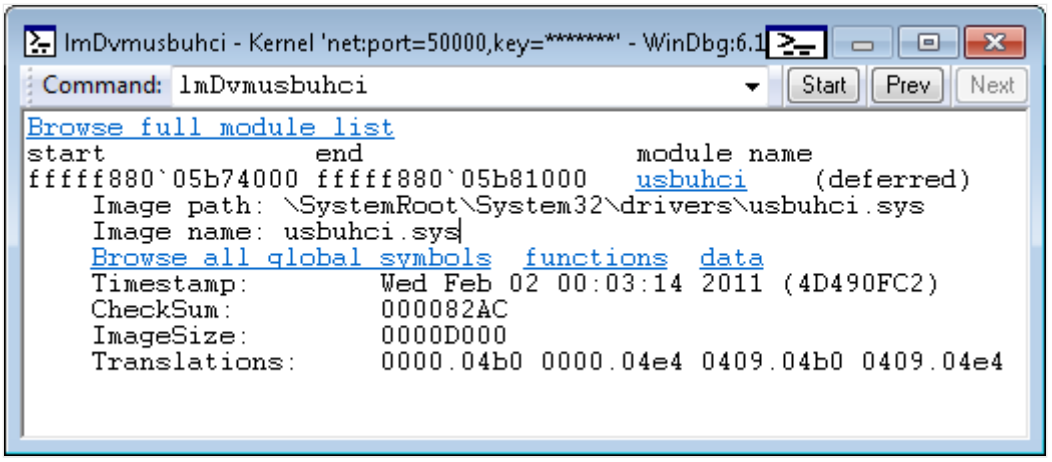
以下命令能够生成 DML 输出：

- .dml_start
- .dml_flow
- ！ dml_proc
- lmD
- **公里**
- .chain /D
- .help /D
- .printf /D

lmD 命令是一个能够提供 DML 输出的命令的示例。 lmD 命令显示已加载模块的列表。 如下图所示，每个模块名称都是一个链接，可以单击该链接获取有关该模块的更多详细信息。



下图显示了单击 **usbuhci** 链接的结果。 输出包含其他链接，可让你浏览 usbuhci 模块的更多详细信息。



打开和关闭 DML

.prefer_dml命令打开或关闭 DML。 启用 DML 时（.prefer_dml 1），能够生成 DML 输出的命令将默认生成 DML 输出。

控制台增强功能

所有 Windows 调试器现在都有支持 DML 分析的命令输出区域。 在 windbg 中，命令窗口支持所有 DML 行为，并显示颜色、字体样式和链接。 控制台调试器 ntsd、cdb 和 kd 仅支持 DML 的颜色属性，仅当在启用了颜色模式的 true 控制台中运行时。 使用重定向的 I/O、ntsd –d 或 remote.exe 会话的调试器不会显示任何颜色。

控制台调试器颜色模式

控制台调试器、ntsd、cdb 和 kd 现在能够在真实控制台中运行时显示彩色输出。 这不是默认设置，它要求通过tools.ini显式启用颜色模式。 tools.ini中的新col_mode <true|false> 标记控制颜色模式设置。 有关使用tools.ini文件的详细信息，请参阅 配置 tools.ini

启用颜色模式后，调试器可以生成彩色输出。 默认情况下，大多数颜色未设置，而是默认为当前控制台颜色。

Windbg 命令浏览器窗口

在 Windows 10 及更高版本中，Windbg 命令浏览器窗口分析和显示 DML。 完全支持链接、 <执行>和外观修改等<>所有标记。

若要使用 WinDbg 中的菜单启动命令浏览器会话，请选择“视图”**和**“**命令浏览器**”。 命令窗口中的 .browse <命令> 将打开新的命令浏览器窗口并执行给定的命令。 有关详细信息，请参阅在 WinDbg 中使用命令浏览器窗口（经典版）。 还可以使用 Ctrl+N 打开新的命令浏览器窗口。

命令浏览器窗口故意模拟 Web 浏览器的行为，其中包含下拉历史记录和上一个/下一个按钮。 历史记录下拉列表仅显示最后 20 个命令，但完整历史记录会保留，因此，可以在命令中返回下拉列表以显示较旧的历史记录。

你可以一次性打开任意数量的命令窗口。 命令窗口保留在工作区中，但仅保存当前命令;历史记录未保留。

WinDbg **视图** 菜单具有" **设置浏览器启动命令** "选项，该选项允许用户为新的浏览器窗口设置首选命令，例如.dml_start。 此命令保存在工作区中。

"视图"菜单上提供了"最近使用的命令"子窗口，用于保存感兴趣的命令。 选择最近的命令会打开具有给定命令的新浏览器。 浏览器窗口的上下文菜单上有一个菜单项，用于将窗口的当前命令添加到最近命令的列表。 最近命令的列表保存在工作区中。

命令浏览器窗口同步执行命令，因此在命令完成之前不会显示输出。 长时间运行的命令在完成之前不会显示任何内容。

链接具有类似于 Web 浏览器中右键单击上下文菜单的右键单击上下文菜单。 可以在新的浏览器窗口中打开链接。 可以将链接的命令复制到剪贴板以供使用。

单击标题栏右上角附近的图标，将命令浏览器窗口设置为自动刷新或手动刷新。 自动刷新浏览器将自动在调试器状态更改上重新运行其命令。 这会使输出保持实时状态，但代价是对所有更改执行命令。 默认情况下，自动刷新处于打开状态。 如果浏览器不需要实时运行，则可以使用窗口的上下文菜单来禁用自动刷新。

由于命令由引擎执行，而不是由用户界面执行，因此用户界面特定的命令（如 **.cls（Clear Screen）**将在命令浏览器窗口中使用时返回语法错误。 这也意味着，当用户界面是远程客户端时，命令将由服务器执行，而不是由客户端执行，命令输出将显示服务器状态。

命令浏览器窗口可以运行任何调试器命令，它不必是生成 DML 的命令。 可以使用浏览器窗口让任意一组命令处于活动状态以供使用。

自定义 DML

DML 定义一组可包含在命令输出中的标记。 一个示例是 <链接> 标记。 可以使用 .dml_start **和 .browse 命令来试验<链接>标记（和其他 DML 标记）。** 命令 **.browse .dml_start** *filepath* 执行存储在 DML 文件中的命令。 输出显示在 命令浏览器窗口中，而不是常规命令窗口。

假设文件 c：\DmlExperiment.txt包含以下行。

```
text
```

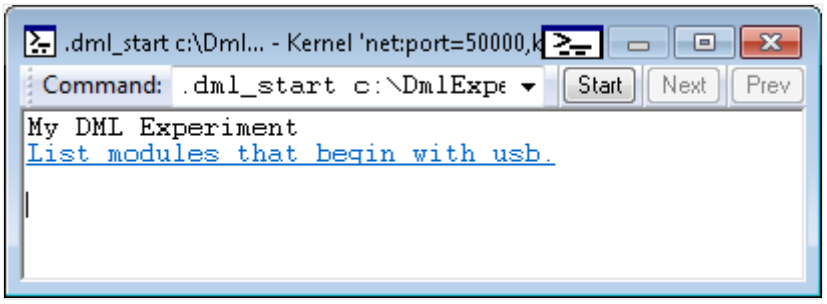
```
My DML Experiment
```

```
<link cmd="lmD musb*">List modules that begin with usb.</link>
```

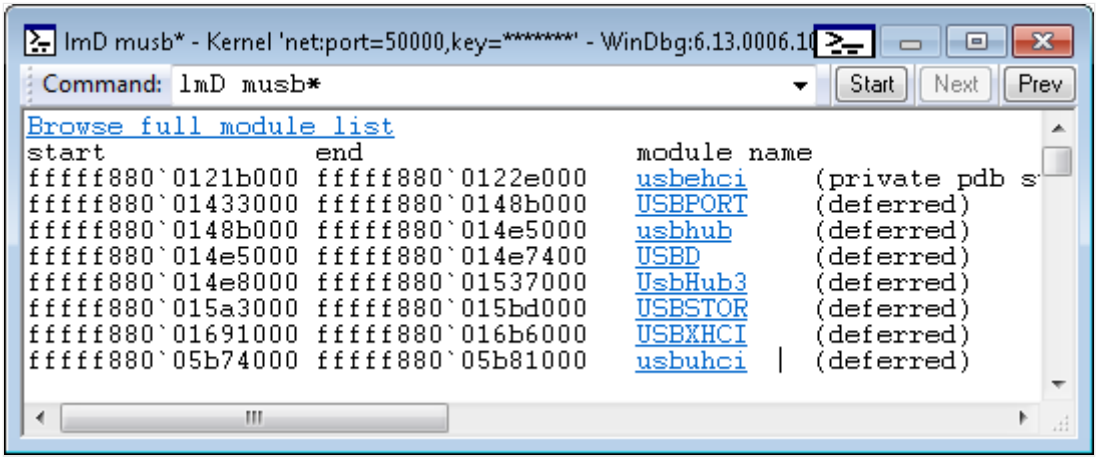
以下命令在“命令浏览器”窗口中显示文本和链接。

```
dbgcmd
```

```
.browse .dml_start c:\Dml_Experiment.txt
```



如果单击 **以 usb** 链接开头的列表模块，则会看到类似于下图的输出。



有关 DML 自定义和 DML 标记的完整列表的彻底讨论，请参阅 使用 DML 自定义调试器输出。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

控制异常和事件

项目 • 2023/04/14

可以通过各种方法捕获和处理用户模式和内核模式应用程序中的异常。 活动调试器、事后调试器或内部错误处理例程都是处理异常的常见方法。

有关这些各种异常处理程序的优先顺序的详细信息，请参阅 启用事后调试。

当 Microsoft Windows 操作系统允许调试器处理异常时，生成异常的应用程序 *会中断调试* 器。 也就是说，应用程序停止，调试器变为活动状态。 然后，调试器可以以某种方式处理异常或分析情况。 然后，调试器可以结束进程或让它继续运行。

如果调试器忽略异常并允许应用程序继续运行，则操作系统会查找其他异常处理程序，就像没有调试器一样。 如果异常得到处理，应用程序将继续运行。 但是，如果异常仍未处理，则调试器会获得第二次处理这种情况的机会。

使用调试器分析异常

当异常或事件闯入调试器时，可以使用调试器检查正在执行的代码和应用程序正在使用的内存。 通过更改某些数量或跳转到应用程序中的不同点，可以消除异常的原因。

可以通过发出 **gh (Go with Exception Handled)** 或 **gn (Go with Exception Not Handled)** 命令来恢复执行。

如果在调试器处理异常的第二次机会中发出 **gn** 命令，应用程序将结束。

内核模式异常

内核模式代码中发生的异常比用户模式异常更严重。 如果未处理内核模式异常，则会发出 bug 检查并停止系统。

与用户模式异常一样，如果内核模式调试器附加到系统，则会在 *bug 检查屏幕* (也称为*蓝屏*) 出现之前通知调试器。 如果未附加调试器，则显示 bug 检查屏幕。 在这种情况下，操作系统可能会创建 故障转储文件。

从调试器控制异常和事件

可以将调试器配置为以特定方式响应指定的异常和事件。

调试器可以为每个异常或事件设置中断状态：

- 该事件在“第一次机会”) 发生 (，就可能导致调试器中断。

- 在其他错误处理程序有机会响应 (“第二次机会”) 后，该事件可能会中断。
- 事件还可以向调试器发送消息，但继续执行。
- 调试器可以忽略 事件。

调试器还可以设置每个异常和事件的处理状态。 调试器可以将事件视为已处理的异常或未经处理的异常。 (当然，实际上不是错误的事件不需要任何处理。)

可以通过执行以下操作之一来控制中断状态和处理状态：

- 在“调试器命令”窗口中使用 **SXE**、**SXD**、**SXN** 或 **SXI** 命令。
- (CDB 和 NTSD) 在**命令行**上使用 **-x**、**-xe**、**-xd**、**-xn** 或 **-xi** 选项。
- (CDB、NTSD 和 KD)Tools.ini文件中使用 **sxe** 或 **sxd** 关键字 (keyword) 。
- (WinDbg 仅) “**调试**”菜单上的“选择**事件筛选器**”打开“**事件筛选器**”对话框，然后选择所需的选项。

SX* 命令、**-x*** 命令行选项和 **sx*** Tools.ini 关键字 (keyword) 通常设置指定事件的中断状态。 可以添加 **-h** 选项，以便改为设置处理状态。

有四个特殊事件代码 (**cc**、 **hc**、 **bpec** 和 **ssec**)，它们始终指定处理状态而不是中断状态。

可以使用 **.lastevent (显示** 最后一个事件) 命令来显示最近的异常或事件。

控制中断状态

设置异常或事件的中断状态时，可以使用以下选项。

命令	状态名称	说明
SXE 或 **-xe**	**休息** **已启用** ()	发生此异常时，目标会立即中断调试器。 此中断发生在激活任何其他错误处理程序之前。 此方法称为 *第一次机会处理*。
SXD 或 **-xd**	**第二次机会休息** (**禁用**)	尽管) 显示消息，但调试器不会针对此类第一次发生异常 (中断。 如果其他错误处理程序无法解决此异常，则执行将停止，目标将中断调试器。 此方法称为 *第二次机会处理*。

命令	状态名称	说明
SXN 或 -xn	**输出** (**通知**)	发生此异常时，目标应用程序根本不会中断调试器。 但是，将显示一条消息，通知用户此异常。
SXI 或 -xi	**忽略**	发生此异常时，目标应用程序不会中断调试器，并且不会显示任何消息。

如果 SX* 设置未预料到异常，则目标应用程序在第二次机会中中断调试器。 本主题的以下“事件定义和默认值”部分列出了事件的默认状态。

若要使用 WinDbg 图形界面设置中断状态，“**调试**”菜单上**的事件筛选器**从“**事件筛选器**”对话框的列表中选择所需的事件，然后选择“**启用**”、“**禁用**”、“**输出**”或“**忽略**”。

控制处理状态

所有事件都被视为未经处理，除非使用 gh (Go with Exception Handled) 命令。

所有异常都被视为未经处理，除非将 sx* 命令与 -h 选项一起使用。

此外， SX* 选项可以为无效句柄、STATUS_BREAKPOINT中断指令和单步异常配置处理状态。 (此配置与其中断配置分开。) 配置中断状态时，这些事件分别命名为 ch、 bpe 和 sse。 配置其处理状态时，这些事件分别命名为 hc、 bpec 和 ssec。 (有关事件的完整列表，请参阅以下“事件定义和默认值”部分。)

可以配置 CTRL+C 事件的处理状态 (cc)，但不能配置其中断状态。 如果应用程序收到 CTRL+C 事件，则应用程序始终会中断调试器。

在 cc、hc、bpec 和 ssec 事件上使用 SX* 命令时，或者在异常时将 SX* 命令与 -h 选项一起使用时，将发生以下操作。

命令	状态名称	说明
SXE	**已处理**	恢复执行时，该事件被视为已处理。
SXD、SXN、SXI	**未处理**	恢复执行时，该事件被视为未处理。

若要使用 WinDbg 图形界面设置处理状态，请在“**调试**”菜单中选择“**事件筛选器**”，从“**事件筛选器**”对话框的列表中选择所需的事件，然后选择“**已处理**”或“**未处理**”。

自动命令

调试器还允许设置在事件或异常导致中断调试器时自动执行的命令。 可以为第一次机会中断设置命令字符串，为第二次机会中断设置命令字符串。 可以使用 SX* 命令或 Debug | **设置这些字符串事件筛选器** 命令。 每个命令字符串可以包含多个用分号分隔的命令。

无论中断状态如何，都会执行这些命令。 也就是说，如果中断状态为“忽略”，则仍会执行该命令。 如果中断状态为“第二次中断”，则会在首次发生异常时执行第一次机会命令，然后再涉及任何其他异常处理程序。 命令字符串可以以执行命令结尾，如 g (Go) 、gh (Go **与异常处理**) ，或 gn (Go **与未处理异常**) 。

事件定义和默认值

可以更改以下异常的中断状态或处理状态。 指示其默认中断状态。

以下异常的默认处理状态始终为“未处理”。 请小心更改此状态。 如果将此状态更改为“已处理”，则此类型的所有第一次和二次机会异常都被视为已处理，并且此配置将绕过所有异常处理例程。

事件代码	**含义**	**默认中断状态**
asrt	断言失败	中断
Av	访问冲突	中断
Dm	数据未对齐	中断
Dz	整数除以零	中断
c000008e	浮点除以零	中断
eh	C++ EH 异常	第二次机会休息
Gp	保护页冲突	中断
第二	非法指令	第二次机会休息
iov	整数溢出	中断
ip	页内 I/O 错误	中断
Isc	无效的系统调用	中断
Isq	无效的锁序列	中断
sbo	堆栈缓冲区溢出	中断

事件代码	含义	默认中断状态
苏联	堆栈溢出	中断
wkd	唤醒调试器	中断
aph	应用程序挂起 如果 Windows 操作系统断定某个进程已停止响应，（即 *挂起*），则会触发此异常。	中断
3c	子应用程序终止	第二次机会休息
chhc	无效句柄	中断
数字	任何编号异常	第二次机会休息

注意可以使用 **ah (断言处理)** 命令替代特定地址的 **asrt** 中断状态。 **ch** 和 **hc** 事件代码引用相同的异常。 控制中断状态时，请使用 **sx* ch**。 控制其处理状态时，请使用 **sx* hc**。

可以更改以下异常的中断状态或处理状态。 指示其默认中断状态。

以下异常的默认处理状态始终为“已处理”。 由于这些异常用于与调试器通信，因此通常不应将其状态更改为“未处理”。 如果调试器忽略这些异常，此状态会导致其他异常处理程序捕获这些异常。

应用程序可以使用 DBG_COMMAND_EXCEPTION (**dbce**) 与调试器通信。 此异常类似于断点，但可以使用 **SX*** 命令在发生此异常时以特定方式做出反应。

事件代码	含义	默认中断状态
dbce	特殊调试器命令异常	忽略
vcpp	特殊 Visual C++ 异常	忽略
wos	WOW64 单步异常	中断
wob	WOW64 断点异常-	中断
Sse ssec	单步异常	中断
Bpe bpec	断点异常	中断

事件代码	含义	默认中断状态
Cce Cc	Ctrl+C 或 CTRL+BREAK 如果目标是控制台应用程序，并且向其传递 CTRL+C 或 CTRL+BREAK，则会触发此异常。	中断

注意 上表中最后三个例外具有两个不同的事件代码。 控制中断状态时，请使用 sse、bpe 和 cce。 控制其处理状态时，请使用 ssec、 bpec 和 cc。

调试托管代码时，以下异常非常有用。

事件代码	含义	默认状态
clr	公共语言运行时异常	第二次机会休息 未处理
clrn	公共语言运行时通知异常	第二次机会休息 已处理

可以更改以下事件的中断状态。 由于这些事件不是例外，因此其处理状态无关紧要。

事件代码	含义	默认中断状态
ser	系统错误	忽略

事件代码	含义	默认中断状态
cpr[：*Process*]	进程创建 设置此事件的中断状态仅适用于用户模式调试。 此事件不会在内核模式下发生。 仅当已通过 -o**命令行选项** 或通过 **.childdbg** (调试子进程) 命令激活 CDB 或 WinDbg 中的子进程的调试时，才能控制此事件。 进程名称可以包含可选的文件扩展名和星号 () *或问号 (？) 作为通配符。 调试器仅记住最新的* ***cpr*** *设置。 不支持单独进程的单独设置。 在* ***cpr*** *和 Process 之间包括冒号或空格。* *如果省略 Process ，则该设置将应用于任何子进程创建。*	忽略
epr[：*Process*]	进程退出 设置此事件的中断状态仅适用于用户模式调试。 此事件不会在内核模式下发生。 仅当已通过 -o**命令行选项** 或通过 **.childdbg** (调试子进程) 命令激活 CDB 或 WinDbg 中的子进程的调试时，才能控制此事件。 进程名称可以包含可选的文件扩展名和星号 () 或问号 (？) 作为通配符。 调试器仅记住最新的 **epr** 设置。 不支持单独进程的单独设置。 在 **epr** 和 *Process* 之间包括冒号或空格。 如果省略 *Process* ，则该设置将应用于任何子进程退出。	忽略
ct	线程创建	忽略
Et	线程退出	忽略

事件代码	含义	默认中断状态
ld[**:** *Module*]	加载模块 如果指定 *Module*，则加载具有此名称的模块时会发生中断。 *模块* 可以指定模块的名称或地址。 如果使用名称， *模块* 可能包含各种通配符和说明符。 (有关语法的详细信息，请参阅字符串通配符语法。) 调试器仅记住最新的 **ld** 设置。 不支持单独模块的单独设置。 在 **ld** 和 *Module* 之间包括冒号或空格。 如果省略 *Module* ，则会在加载任何模块时触发事件。	输出
ud[**:** *Module*]	卸载模块 如果指定 *Module*，则卸载具有此名称的模块或此基址处的模块时，将发生中断。 *模块* 可以指定模块的名称或地址。 如果使用名称， *则 Module* 可以是确切的名称，也可以包含通配符。 如果 *Module* 是确切的名称，则会使用当前调试器模块列表立即将其解析为基址，并将其存储为地址。 如果 *Module* 包含通配符，则会保留模式字符串，以便在发生卸载事件时进行匹配。 在极少数情况下，调试器没有卸载事件的名称信息，并且仅按基址匹配。 因此，如果 *Module* 包含通配符，则调试器无法在此特定卸载情况下执行名称匹配，并在卸载任何模块时中断。 调试器仅记住最新的 **ud** 设置。 不支持单独模块的单独设置。 在 **ud** 和 *Module* 之间包括冒号或空格。 如果省略 *Module* ，则会在加载任何模块时触发事件。	输出

事件代码	含义	默认中断状态
out[：*Output*]	目标应用程序输出 如果指定 *Output*，则仅当收到与指定模式匹配的输出时，才会发生中断。 *输出* 可以包含各种通配符和说明符。 (有关语法的详细信息，请参阅 字符串通配符语法。) 但是， *输出* 不能包含冒号或空格。 匹配时不区分大小写。 在 **out** 和 *Output* 之间包括冒号或空格。	忽略
ibp	初始断点 (此事件在调试会话开始时和重新启动目标计算机后发生。)	**在用户模式下**： 打破。 可以使用 **-g**命令行选项将此状态更改为“忽略”。 **在内核模式下**： 忽略。 可以通过各种方法将此状态更改为“已启用”。 有关如何更改此状态的详细信息，请参阅 崩溃并重新启动目标计算机。
iml	初始模块加载 仅) (内核模式	忽略。 可以通过各种方法将此状态更改为“中断”。 有关如何更改此状态的详细信息，请参阅 崩溃并重新启动目标计算机。

查找进程 ID

项目 • 2023/12/29

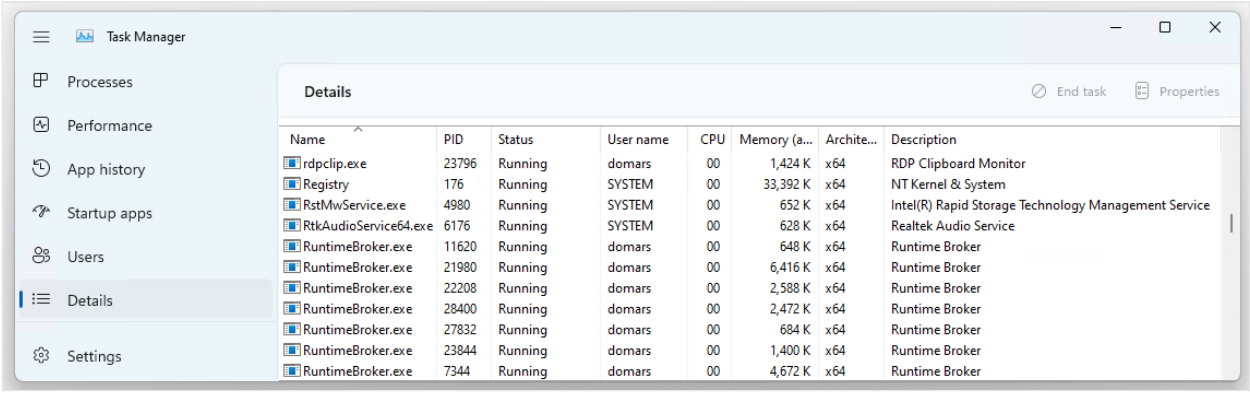
Windows 中运行的每个进程都分配有一个称为进程 ID 的唯一十进制数， (PID) 。 此数字以多种方式使用，例如，在将调试器附加到它时指定进程。

本主题介绍如何使用任务管理器、任务列表 Windows 命令、TList 实用工具、PowerShell Get-Process 命令或调试器确定给定应用的 PID。

任务管理器

可以通过多种方式打开任务管理器，但最简单的方法是选择 Ctrl+Alt+Delete，然后选择“**任务管理器**”。

在 Windows 中，首先单击“ **更多详细信息** ”以展开显示的信息。 在“ **进程** ”选项卡中，选择“ **详细信息** ”以查看 *PID* 列中列出的进程 ID。



单击任意列名进行排序。 可以右键单击进程名称，查看进程的更多选项。

某些内核错误可能会导致任务管理器的图形界面出现延迟。

任务列表命令

使用命令提示符中的内置 Windows **任务列表** 命令显示所有进程、其 PID 和其他各种详细信息。

控制台

```
C:\>tasklist

Image Name                     PID Session Name        Session#    Mem Usage
========================= ======== ================ =========== ============
System Idle Process              0 Services                   0          8 K
System                           4 Services                   0      7,428 K
```

```
Secure System                  104 Services                   0     40,344 K
Registry                       164 Services                   0    146,596 K
smss.exe                       592 Services                   0      1,176 K
csrss.exe                      896 Services                   0      6,224 K
wininit.exe                    980 Services                   0      6,572 K
...
```

使用 `tasklist /?` 显示命令行帮助。

TList 实用工具

任务列表查看器 (TList) （或 tlist.exe）是一个命令行实用工具，用于显示当前在本地计算机上运行的任务列表或用户模式进程列表。 TList 包含在 Windows 调试工具中。 有关如何下载和安装调试工具的信息，请参阅 Windows 调试工具。

如果在 64 位电脑上的默认目录中安装了 Windows 驱动程序工具包，调试工具位于此处：

`C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\`

从命令提示符运行 TList 时，它将在内存中显示具有唯一 PID 编号的所有用户模式进程的列表。 对于每个进程，它都会显示 PID、进程名称，如果进程具有窗口，则显示该窗口的标题。

控制台

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64>tlist -t
System Process (0)
System (4)
  smss.exe (592)
  Memory Compression (3376)
Secure System (104)
Registry (164)
csrss.exe (896)
wininit.exe (980)
  services.exe (660)
    svchost.exe (1232)
      WmiPrvSE.exe (6008)
      dllhost.exe (1748)
      WmiPrvSE.exe (1860)
...
```

有关详细信息，请参阅 TList。

.tlist 调试器命令

如果已在有问题的系统上运行用户模式调试器， **.tlist (列出进程 ID)** 命令将显示该系统上所有 PID 的列表。

PowerShell Get-Process 命令

若要使用自动化脚本，请使用 Get-Process PowerShell 命令。 指定特定的进程名称，以查看该进程的进程 ID。

PowerShell

```
C:\> Get-Process explorer

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
   2520     404   108948     179284   1,702.95    7656   1 explorer
```

有关详细信息，请参阅 Get-Process。

其他资源

若要详细了解 Windows 内部 (包括上下文、线程和进程)，请查看其他资源，例如 Pavel Yosifovich、Mark Russinovich、David Solomon 和 Alex Ionescu 的 *Windows 内部*。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

Windows driver documentation 反馈

Windows driver documentation 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

调试堆栈溢出

项目 • 2024/10/18

堆栈溢出是用户模式线程可能会遇到的错误。 此错误有三个可能的原因：

- 线程使用为其保留的整个堆栈。 这通常是由无限递归引起的。
- 线程无法扩展堆栈，因为页面文件已最大化，因此无法提交其他页面来扩展堆栈。
- 线程无法扩展堆栈，因为系统在用于扩展页面文件的短时间内。

当线程上运行的函数分配局部变量时，变量将放在线程的调用堆栈上。 函数所需的堆栈空间量可以和所有局部变量的大小之和一样大。 但是，编译器通常会执行优化，以减少函数所需的堆栈空间。 例如，如果两个变量位于不同的范围内，则编译器可以为这两个变量使用相同的堆栈内存。 编译器还可以通过优化计算来完全消除某些局部变量。

优化量受生成时应用的编译器设置的影响。 例如，按 /F （设置堆栈大小） - C++编译器选项。

本主题假定大致了解线程、线程块、堆栈和堆等概念。 有关这些基本概念的其他信息，请参阅 *Mark Russinovich 和 David 所罗门Microsoft Windows 内部* 。

在没有符号的情况下调试堆栈溢出

下面是有关如何调试堆栈溢出的示例。 在此示例中，NTSD 与目标应用程序在同一台计算机上运行，并将输出重定向到主计算机上的 KD。 有关详细信息，请参阅从内核调试器控制用户模式调试器。

第一步是查看导致调试器中断的事件：

```
dbgcmd
```

```
0:002> .lastevent
Last event: Exception C00000FD, second chance
```

可以在 ntstatus.h 中查找异常代码0xC00000FD，此异常代码 STATUS_STACK_OVERFLOW，这表示*无法创建堆栈的新防护页*。所有状态代码都列在 2.3.1 NTSTATUS 值中。

还可以使用 ！ error 命令在 Windows 调试器中查找错误。

```
dbgcmd
```

```
0:002> !error 0xC00000FD
Error code: (NTSTATUS) 0xc00000fd (3221225725) - A new guard page for the
stack cannot be created.
```

若要仔细检查堆栈是否溢出，可以使用 k（显示堆栈回溯） 命令：

dbgcmd

```
0:002> k
ChildEBP RetAddr
009fdd0c 71a32520 COMCTL32!_chkstk+0x25
009fde78 77cf8290 COMCTL32!ListView_WndProc+0x4c4
009fde98 77cfd634 USER32!_InternalCallWinProc+0x18
009fdf00 77cd55e9 USER32!UserCallWinProcCheckWow+0x17f
009fdf3c 77cd63b2 USER32!SendMessageWorker+0x4a3
009fdf5c 71a45b30 USER32!SendMessageW+0x44
009fdfec 71a45bb0 COMCTL32!CCSendNotify+0xc0e
009fdffc 71a1d688 COMCTL32!CICustomDrawNotify+0x2a
009fe074 71a1db30 COMCTL32!Header_Draw+0x63
009fe0d0 71a1f196 COMCTL32!Header_OnPaint+0x3f
009fe128 77cf8290 COMCTL32!Header_WndProc+0x4e2
009fe148 77cfd634 USER32!_InternalCallWinProc+0x18
009fe1b0 77cd4490 USER32!UserCallWinProcCheckWow+0x17f
009fe1d8 77cd46c8 USER32!DispatchClientMessage+0x31
009fe200 77f7bb3f USER32!__fnDWORD+0x22
009fe220 77cd445e ntdll!_KiUserCallbackDispatcher+0x13
009fe27c 77cfd634 USER32!DispatchMessageWorker+0x3bc
009fe2e4 009fe4a8 USER32!UserCallWinProcCheckWow+0x17f
00000000 00000000 0x9fe4a8
```

目标线程已分解为 COMCTL32！_chkstk，这表示堆栈问题。 现在，应调查目标进程的堆栈使用情况。 进程有多个线程，但重要的是导致溢出的线程，因此首先使用 ~ （线程状态） 命令标识此线程：

dbgcmd

```
0:002> ~*k

   0  id: 570.574   Suspend: 1 Teb 7ffde000 Unfrozen
   .....

   1  id: 570.590   Suspend: 1 Teb 7ffdd000 Unfrozen
   .....

. 2  id: 570.598   Suspend: 1 Teb 7ffdc000 Unfrozen
ChildEBP RetAddr
 009fdd0c 71a32520 COMCTL32!_chkstk+0x25
.....
```

```
   3  id: 570.760   Suspend: 1 Teb 7ffdb000 Unfrozen
```

现在需要调查线程 2。 此行左侧的句点指示这是当前线程。

堆栈信息包含在0x7FFDC000的 TEB （线程环境块） 中。 列出它的最简单方法是使用 ! teb。

dbgcmd

```
0:000> !teb
TEB at 000000c64b95d000
    ExceptionList:        0000000000000000
    StackBase:            000000c64ba80000
    StackLimit:           000000c64ba6f000
    SubSystemTib:         0000000000000000
    FiberData:            0000000000001e00
    ArbitraryUserPointer: 0000000000000000
    Self:                 000000c64b95d000
    EnvironmentPointer:   0000000000000000
    ClientId:             0000000000003bbc . 0000000000004ba0
    RpcHandle:            0000000000000000
    Tls Storage:          0000027957243530
    PEB Address:          000000c64b95c000
    LastErrorValue:       0
    LastStatusValue:      0
    Count Owned Locks:    0
    HardErrorMode:        0```
```

但是，这要求你具有正确的符号。 更困难的情况是，没有符号，需要使用 dd （显示内存） 命令在该位置显示原始值：

dbgcmd

```
0:002> dd 7ffdc000 L4
7ffdc000   009fdef0 00a00000 009fc000 00000000
```

若要解释这一点，需要查找 TEB 数据结构的定义。 使用 dt Display Type 命令在符号可用的系统上执行此操作。

dbgcmd

```
0:000> dt _TEB
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : Ptr32 Void
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : Ptr32 Void
   +0x02c ThreadLocalStoragePointer : Ptr32 Void
```

```
   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
   +0x034 LastErrorValue   : Uint4B
   +0x038 CountOfOwnedCriticalSections : Uint4B
   +0x03c CsrClientThread  : Ptr32 Void
   +0x040 Win32ThreadInfo  : Ptr32 Void
   +0x044 User32Reserved   : [26] Uint4B
   +0x0ac UserReserved     : [5] Uint4B
   +0x0c0 WOW32Reserved    : Ptr32 Void
...
```

线程数据结构

若要了解有关线程的详细信息，还可以显示有关线程控制块相关结构的信息 ethread 和 kthread。 （请注意，此处显示了 64 位示例。

dbgcmd

```
0:001> dt nt!_ethread
ntdll!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x430 CreateTime       : _LARGE_INTEGER
   +0x438 ExitTime         : _LARGE_INTEGER
   +0x438 KeyedWaitChain   : _LIST_ENTRY
   +0x448 PostBlockList    : _LIST_ENTRY
   +0x448 ForwardLinkShadow : Ptr64 Void
   +0x450 StartAddress     : Ptr64 Void
...
```

dbgcmd

```
0:001> dt nt!_kthread
ntdll!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 SListFaultAddress : Ptr64 Void
   +0x020 QuantumTarget    : Uint8B
   +0x028 InitialStack     : Ptr64 Void
   +0x030 StackLimit       : Ptr64 Void
   +0x038 StackBase        : Ptr64 Void
```

有关 *线程数据结构的详细信息，请参阅 Microsoft Windows 内部* 。

查看_TEB结构的 32 位版本，它指示 TEB 结构中的第二和第三个 DWORD 分别指向堆栈的底部和顶部。 在此示例中，这些地址是0x00A00000和0x009FC000。 （堆栈在内存中向下增长。可以使用 ？ （计算表达式） 命令：

dbgcmd

```
0:002> ? a00000-9fc000
```

```
Evaluate expression: 16384 = 00004000
```

这表明堆栈大小为 16 K。最大堆栈大小存储在字段 **DeallocationStack** 中，这是此 TEB 结构的一部分。 该 `DeallocationStack` 字段指示堆栈的基数。 进行一些计算后，可以确定此字段的偏移量0xE0C。

dbgcmd

```
0:002> dd 7ffdc000+e0c L1
7ffdce0c   009c0000

0:002> ? a00000-9c0000
Evaluate expression: 262144 = 00040000
```

这表明最大堆栈大小为 256 K，这意味着剩余的堆栈空间超过足够的堆栈空间。

此外，此过程看起来很干净 -- 它不是无限递归，也不是通过使用过多的基于堆栈的数据结构超过其堆栈空间。

现在，使用！ vm 扩展命令了解整个系统内存使用情况，了解 KD：

dbgcmd

```
0:002> .breakin
Break instruction exception - code 80000003 (first chance)
ntoskrnl!_DbgBreakPointWithStatus+4:
80148f9c cc               int     3

kd> !vm

*** Virtual Memory Usage ***
        Physical Memory:     16268   (   65072 Kb)
        Page File: \??\C:\pagefile.sys
           Current:    147456Kb Free Space:     65988Kb
           Minimum:     98304Kb Maximum:       196608Kb
        Available Pages:      2299   (    9196 Kb)
        ResAvail Pages:       4579   (   18316 Kb)
        Locked IO Pages:        93   (     372 Kb)
        Free System PTEs:    42754   (  171016 Kb)
        Free NP PTEs:         5402   (   21608 Kb)
        Free Special NP:       348   (    1392 Kb)
        Modified Pages:        757   (    3028 Kb)
        NonPagedPool Usage:    811   (    3244 Kb)
        NonPagedPool Max:     6252   (   25008 Kb)
        PagedPool 0 Usage:    1337   (    5348 Kb)
        PagedPool 1 Usage:     893   (    3572 Kb)
        PagedPool 2 Usage:     362   (    1448 Kb)
        PagedPool Usage:      2592   (   10368 Kb)
        PagedPool Maximum:   13312   (   53248 Kb)
        Shared Commit:        3928   (   15712 Kb)
```

```
        Special Pool:         1040   (    4160 Kb)
        Shared Process:       3641   (   14564 Kb)
        PagedPool Commit:     2592   (   10368 Kb)
        Driver Commit:         887   (    3548 Kb)
        Committed pages:     45882   (  183528 Kb)
        Commit limit:        50570   (  202280 Kb)

        Total Private:       33309   (  133236 Kb)
         .....
```

首先，查看未分页的池使用情况和分页池使用情况。 两者都在限制范围内，因此这不是问题的原因。

接下来，查看提交的页数：183528 202280 年。 这非常接近限制。 虽然此显示未显示此数字完全处于限制，但请记住，在执行用户模式调试时，系统上运行其他进程。 每次执行 NTSD 命令时，这些其他进程也会分配和释放内存。 这意味着，在堆栈溢出发生时，你不知道内存状态到底是什么样子。 鉴于提交的页码与限制的接近程度，因此可以合理地得出结论，页面文件在某个时间点使用，这会导致堆栈溢出。

这种情况并不常见，目标应用程序对此无法真正出错。 如果频繁发生，可能需要考虑为失败的应用程序提高初始堆栈承诺。

分析单个函数调用

了解特定函数调用分配的堆栈空间量也很有用。

为此，请反汇编前几个说明并查找指令 `sub esp` *编号*。 这会移动堆栈指针，从而有效地保留 *本地数据的字节数* 。

以下是一个示例。 首先使用 k 命令查看堆栈。

dbgcmd

```
0:002> k
ChildEBP RetAddr
009fdd0c 71a32520 COMCTL32!_chkstk+0x25
009fde78 77cf8290 COMCTL32!ListView_WndProc+0x4c4
009fde98 77cfd634 USER32!_InternalCallWinProc+0x18
009fdf00 77cd55e9 USER32!UserCallWinProcCheckWow+0x17f
009fdf3c 77cd63b2 USER32!SendMessageWorker+0x4a3
009fdf5c 71a45b30 USER32!SendMessageW+0x44
009fdfec 71a45bb0 COMCTL32!CCSendNotify+0xc0e
009fdffc 71a1d688 COMCTL32!CICustomDrawNotify+0x2a
009fe074 71a1db30 COMCTL32!Header_Draw+0x63
009fe0d0 71a1f196 COMCTL32!Header_OnPaint+0x3f
009fe128 77cf8290 COMCTL32!Header_WndProc+0x4e2
```

然后使用 u、ub、uu （Unassemble） 命令查看该地址的汇编程序代码。

dbgcmd

```
0:002> u COMCTL32!Header_Draw
 COMCTL32!Header_Draw :
71a1d625 55              push    ebp
71a1d626 8bec            mov     ebp,esp
71a1d628 83ec58          sub     esp,0x58
71a1d62b 53              push    ebx
71a1d62c 8b5d08          mov     ebx,[ebp+0x8]
71a1d62f 56              push    esi
71a1d630 57              push    edi
71a1d631 33f6            xor     esi,esi
```

这表明Header_Draw分配的堆栈空间0x58字节。

r （Registers） 命令提供有关寄存器的当前内容的信息，例如 esp。

当符号可用时调试堆栈溢出

符号为内存中存储的项提供标签，并且当可用时，可以更轻松地检查代码。 有关符号的概述，请参阅 “使用符号”。 有关设置符号路径的信息，请参阅 .sympath （设置符号路径）。

若要创建堆栈溢出，可以使用此代码，该代码会继续调用子例程，直到堆栈耗尽为止。

C++

```
// StackOverFlow1.cpp
// This program calls a sub routine using recursion too many times
// This causes a stack overflow
//

#include <iostream>

void Loop2Big()
{
    const char* pszTest = "My Test String";
    for (int LoopCount = 0; LoopCount < 10000000; LoopCount++)
    {
        std::cout << "In big loop \n";
        std::cout << (pszTest), "\n";
        std::cout << "\n";
        Loop2Big();
    }
}


int main()
```

```
{
    std::cout << "Calling Loop to use memory \n";
    Loop2Big();
}
```

在 WinDbg 下编译并运行代码时，它将循环一些次数，然后引发堆栈溢出异常。

dbgcmd

```
(336c.264c): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=0fa90000 edx=00000000 esi=773f1ff4
edi=773f25bc
eip=77491a02 esp=010ffa0c ebp=010ffa38 iopl=0         nv up ei pl zr na pe
nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2b:
77491a02 cc              int     3
0:000> g
(336c.264c): Stack overflow - code c00000fd (first chance)
```

使用！ analyze 命令检查循环是否确实有问题。

dbgcmd

```
...

FAULTING_SOURCE_LINE_NUMBER:  25

FAULTING_SOURCE_CODE:
    21: int main()
    22: {
    23:     std::cout << "Calling Loop to use memory \n";
    24:     Loop2Big();
>   25: }
    26:
```

使用 kb 命令，我们看到循环程序有多个实例，每个实例都使用内存。

dbgcmd

```
0:000> kb
 # ChildEBP RetAddr      Args to Child
...
0e 010049b0 00d855b5     01004b88 00d81023 00ff5000
StackOverFlow1!Loop2Big+0x57 [C:\StackOverFlow1\StackOverFlow1.cpp @ 13]
0f 01004a9c 00d855b5     01004c74 00d81023 00ff5000
StackOverFlow1!Loop2Big+0x85 [C:\StackOverFlow1\StackOverFlow1.cpp @ 17]
10 01004b88 00d855b5     01004d60 00d81023 00ff5000
StackOverFlow1!Loop2Big+0x85 [C:\StackOverFlow1\StackOverFlow1.cpp @ 17]
```

```
11 01004c74 00d855b5     01004e4c 00d81023 00ff5000
StackOverFlow1!Loop2Big+0x85 [C:\StackOverFlow1\StackOverFlow1.cpp @ 17]
12 01004d60 00d855b5     01004f38 00d81023 00ff5000
StackOverFlow1!Loop2Big+0x85 [C:\StackOverFlow1\StackOverFlow1.cpp @ 17]
13 01004e4c 00d855b5     01005024 00d81023 00ff5000
StackOverFlow1!Loop2Big+0x85 [C:\StackOverFlow1\StackOverFlow1.cpp @ 17]
14 01004f38 00d855b5     01005110 00d81023 00ff5000
StackOverFlow1!Loop2Big+0x85 [C:\StackOverFlow1\StackOverFlow1.cpp @ 17]
15 01005024 00d855b5     010051fc 00d81023 00ff5000
StackOverFlow1!Loop2Big+0x85 [C:\StackOverFlow1\StackOverFlow1.cpp @ 17]
16 01005110 00d855b5     010052e8 00d81023 00ff5000
StackOverFlow1!Loop2Big+0x85 [C:\StackOverFlow1\StackOverFlow1.cpp @ 17]
17 010051fc 00d855b5     010053d4 00d81023 00ff5000
StackOverFlow1!Loop2Big+0x85 [C:\StackOverFlow1\StackOverFlow1.cpp @ 17]
18 010052e8 00d855b5     010054c0 00d81023 00ff5000
StackOverFlow1!Loop2Big+0x85 [C:\StackOverFlow1\StackOverFlow1.cpp @ 17]
19 010053d4 00d855b5     010055ac 00d81023 00ff5000
StackOverFlow1!Loop2Big+0x85 [C:\StackOverFlow1\StackOverFlow1.cpp @ 17]
1a 010054c0 00d855b5     01005698 00d81023 00ff5000
StackOverFlow1!Loop2Big+0x85 [C:\StackOverFlow1\StackOverFlow1.cpp @ 17]
1b 010055ac 00d855b5     01005784 00d81023 00ff5000
StackOverFlow1!Loop2Big+0x85 [C:\StackOverFlow1\StackOverFlow1.cpp @ 17]

...
```

如果符号可用， dt _TEB 可用于显示有关线程块的信息。 有关线程内存的详细信息，请参阅 线程堆栈大小。

dbgcmd

```
0:000> dt _TEB
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : Ptr32 Void
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : Ptr32 Void
   +0x02c ThreadLocalStoragePointer : Ptr32 Void
   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
   +0x034 LastErrorValue   : Uint4B
   +0x038 CountOfOwnedCriticalSections : Uint4B
   +0x03c CsrClientThread  : Ptr32 Void
   +0x040 Win32ThreadInfo  : Ptr32 Void
   +0x044 User32Reserved   : [26] Uint4B
   +0x0ac UserReserved     : [5] Uint4B
   +0x0c0 WOW32Reserved    : Ptr32 Void
```

我们还可以使用 显示 StackBase abd StackLimit 的！ teb 命令。

dbgcmd

```
0:000> !teb
TEB at 00ff8000
    ExceptionList:        01004570
    StackBase:            01100000
    StackLimit:           01001000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 00ff8000
    EnvironmentPointer:   00000000
    ClientId:             0000336c . 0000264c
    RpcHandle:            00000000
    Tls Storage:          00ff802c
    PEB Address:          00ff5000
    LastErrorValue:       0
    LastStatusValue:      c00700bb
    Count Owned Locks:    0
    HardErrorMode:        0
```

可以使用此命令计算堆栈大小。

dbgcmd

```
0:000> ?? int(@$teb->NtTib.StackBase) - int(@$teb->NtTib.StackLimit)
int 0n1044480
```

命令摘要

- k（显示堆栈回溯）
- ~（线程状态）
- d、da、db、dc、dd、dD、df、dp、dq、du、dw（显示内存）
- u、ub、uu （Unassemble）
- r（寄存器）
- .sympath（设置符号路径）
- x（检查符号）
- dt （显示类型）
- !analyze
- ！teb

另请参阅

Getting Started with WinDbg (User-Mode)（WinDbg 入门（用户模式））

/F （设置堆栈大小） - C++编译器选项

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

手动遍历堆栈

项目 • 2023/06/15

在某些情况下，堆栈跟踪函数将在调试器中失败。 这可以由对无效地址的调用导致调试器丢失返回地址的位置导致的;或者，你可能遇到无法直接获取堆栈跟踪的堆栈指针;或者可能存在其他调试器问题。 在任何情况下，能够手动遍查堆栈通常很有价值。

基本概念相当简单：转储堆栈指针，找出模块的加载位置，查找可能的函数地址，并通过检查每个可能的堆栈条目是否调用下一个来验证。

在完成示例之前，请务必注意，（**显示堆栈回溯**) 命令的 kb 具有 Intel 系统上的附加功能。 通过执行 kb=[ebp] [eip] [esp]，调试器将分别显示具有基指针、指令指针和堆栈指针的给定值的帧的堆栈跟踪。

对于示例，使用实际提供堆栈跟踪的故障，以便在最后检查结果。

第一步是找出加载位置的模块。 这是使用 **x (检查符号)** 命令完成的，（某些符号出于长度) 的原因进行编辑：

dbgcmd

```
kd> x *!
start    end        module name
77f70000 77fb8000   ntdll      (C:\debug\ntdll.dll,
\\ntstress\symbols\dll\ntdll.DBG)
80010000 80012320   Aha154x    (load from Aha154x.sys deferred)
80013000 8001aa60   SCSIPORT   (load from SCSIPORT.SYS deferred)
8001b000 8001fba0   Scsidisk   (load from Scsidisk.sys deferred)

80100000 801b7b40   NT         (ntoskrnl.exe,
\\ntstress\symbols\exe\ntoskrnl.DBG)
802f0000 8033c000   Ntfs       (load from Ntfs.sys deferred)
80400000 8040c000   hal        (load from hal.dll deferred)
fe4c0000 fe4c38c0   vga        (load from vga.sys deferred)
fe4d0000 fe4d3e60   VIDEOPRT   (load from VIDEOPRT.SYS deferred)
fe4e0000 fe4f0e40   ati        (load from ati.SYS deferred)
fe500000 fe5057a0   Msfs       (load from Msfs.SYS deferred)
fe510000 fe519560   Npfs       (load from Npfs.SYS deferred)

fe520000 fe521f60   ndistapi   (load from ndistapi.sys deferred)
fe530000 fe54ed20   Fastfat    (load from Fastfat.SYS deferred)
fe5603e0 fe575360   NDIS       (NDIS.SYS, \\ntstress\symbols\SYS\NDIS.DBG)
fe580000 fe585920   elnkii     (elnkii.sys,
\\ntstress\symbols\sys\elnkii.DBG)
fe590000 fe59b8a0   ndiswan    (load from ndiswan.sys deferred)
fe5a0000 fe5b7c40   nbf        (load from nbf.sys deferred)
fe5c0000 fe5c1b40   TDI        (load from TDI.SYS deferred)
fe5d0000 fe5dd580   nwlnkipx   (load from nwlnkipx.sys deferred)
```

```
fe5e0000 fe5ee220   nwlnknb   (load from nwlnknb.sys deferred)
fe5f0000 fe5fb320   afd       (load from afd.sys deferred)
fe610000 fe62bf00   tcpip     (load from tcpip.sys deferred)
fe630000 fe648600   netbt     (load from netbt.sys deferred)
fe650000 fe6572a0   netbios   (load from netbios.sys deferred)
fe660000 fe660000   Parport   (load from Parport.SYS deferred)
fe670000 fe670000   Parallel  (load from Parallel.SYS deferred)
fe680000 fe6bcf20   rdr       (rdr.sys, \\ntstress\symbols\sys\rdr.DBG)

fe6c0000 fe6f0920   srv       (load from srv.sys deferred)
```

输出来自较旧版本的 Windows，模块名称在当前版本中会有所不同。

第二步是转储堆栈指针，以查找 **x *!** 命令提供的模块中的地址：

dbgcmd

```
kd> dd esp
fe4cc97c  80136039 00000270 00000000 00000000
fe4cc98c  fe682ae4 801036fe 00000000 fe68f57a
fe4cc99c  fe682a78 ffb5b030 00000000 00000000
fe4cc9ac  ff680e08 801036fe 00000000 00000000
fe4cc9bc  fe6a1198 00000001 fe4cca78 ffae9d98

fe4cc9cc  02000901 fe4cca68 ffb50030 ff680e08
fe4cc9dc  ffa449a8 8011c901 fe4cca78 00000000
fe4cc9ec  80127797 80110008 00000246 fe6a1430

kd> dd
fe4cc9fc  00000270 fe6a10ae 00000270 ffa44abc
fe4cca0c  ffa449a8 ff680e08 fe6b2c04 ff680e08
fe4cca1c  ffa449a8 e12820c8 e1235308 ffa449a8
fe4cca2c  fe685968 ff680e08 e1235308 ffa449a8
fe4cca3c  ffb0ad48 ffb0ad38 00100000 ffb0ad38
fe4cca4c  00000000 ffa44a84 e1235308 0000000a
fe4cca5c  c00000d6 00000000 004ccb28 fe4ccbc4

fe4cca6c  fe680ba4 fe682050 00000000 fe4ccbd4
```

若要确定哪些值可能是函数地址，哪些值是参数或保存的寄存器，首先要考虑的是堆栈上不同类型的信息的外观。 大多数整数的值会较小，这意味着当显示为 DWORD 时，它们将大多为零， (如0x00000270) 。 大多数指向本地地址的指针将靠近堆栈指针 (，如 fe4cca78) 。 状态代码通常以 c (c00000d6) 开头。 Unicode 和 ASCII 字符串可以通过每个字符在 20-7f 范围内进行标识。 (在 KD 中， dc (Display Memory) 命令将在右侧显示字符。) 最重要的是，函数地址将位于 **x *!** 列出的范围内。

请注意，列出的所有模块都在 77f70000 到 8040c000 和 fe4c0000 到 fe6f0920 的范围内。 根据这些范围，上述列表中的可能函数地址为：80136039，801036fe (列出两次，

因此更有可能是参数)， fe682ae4、fe68f57a、fe682a78、fe6a1198、8011c901、80127797、80110008、fe6a1430、fe6a10ae、fe6b2c04、fe685968、fe680ba4 和 fe682050。 为每个地址使用 ln (List Nearest Symbols) 命令调查这些位置：

dbgcmd

```
kd> ln 80136039
(80136039)   NT!_KiServiceExit+0x1e   |  (80136039)   NT!_KiServiceExit2-
0x177
kd> ln fe682ae4
(fe682ae4)   rdr!_RdrSectionInfo+0x2c | (fe682ae4)
rdr!_RdrFcbReferenceLock-0xb4
kd> ln 801036fe
(801036fe)   NT!_KeWaitForSingleObject | (801036fe)
NT!_MmProbeAndLockPages-0x2f8
kd> ln fe68f57a
(fe68f57a)   rdr!_RdrDereferenceDiscardableCode+0xb4
                        (fe68f57a)   rdr!_RdrUninitializeDiscardableCode-
0xa
kd> ln fe682a78
(fe682a78)   rdr!_RdrDiscardableCodeLock | (fe682a78)
rdr!_RdrDiscardableCodeTimeout-0x38

kd> ln fe6a1198
(fe6a1198)   rdr!_SubmitTdiRequest+0xae | (fe6a1198)
rdr!_RdrTdiAssociateAddress-0xc
kd> ln 8011c901
(8011c901)   NT!_KeSuspendThread+0x13 | (8011c901)
NT!_FsRtlCheckLockForReadAccess-0x55
kd> ln 80127797
(80127797)   NT!_ZwCloseObjectAuditAlarm+0x7 | (80127797)
NT!_ZwCompleteConnectPort-0x9
kd> ln 80110008
(80110008)   NT!_KeWaitForMultipleObjects+0x27c | (80110008)
NT!_FsRtlLookupMcbEntry-0x164
kd> ln fe6a1430
(fe6a1430)   rdr!_RdrTdiCloseConnection+0xa | (fe6a1430)
rdr!_RdrDoTdiConnect-0x4

kd> ln fe6a10ae
(fe6a10ae)   rdr!_RdrTdiDisconnect+0x56 | (fe6a10ae)
rdr!_SubmitTdiRequest-0x3c
kd> ln fe6b2c04
(fe6b2c04)   rdr!_CleanupTransportConnection+0x64 |
(fe6b2c04)rdr!_RdrReferenceServer-0x20
kd> ln fe685968
(fe685968)   rdr!_RdrReconnectConnection+0x1b6
                        (fe685968)   rdr!_RdrInvalidateServerConnections-
0x32
kd> ln fe682050
(fe682050)   rdr!__strnicmp+0xaa   |  (fe682050)   rdr!_BackPackSpinLock-
0xa10
```

如前所述，801036fe 不太可能成为堆栈跟踪的一部分，因为它被列出两次。 如果返回地址的偏移量为零，则可以忽略它们，（不能返回到函数）的开头。 根据此信息，堆栈跟踪显示为：

dbgcmd

```
NT!_KiServiceExit+0x1e
rdr!_RdrSectionInfo+0x2c
rdr!_RdrDereferenceDiscardableCode+0xb4
rdr!_SubmitTdiRequest+0xae
NT!_KeSuspendThread+0x13
NT!_ZwCloseObjectAuditAlarm+0x7
NT!_KeWaitForMultipleObjects+0x27c
rdr!_RdrTdiCloseConnection+0xa
rdr!_RdrTdiDisconnect+0x56
rdr!_CleanupTransportConnection+0x64
rdr!_RdrReconnectConnection+0x1b6
rdr!__strnicmp+0xaa
```

若要验证每个符号，请紧接在指定的返回地址之前取消组合，以查看它是否调用了其上方的 函数。 为了缩短长度，将编辑以下内容，（通过试验和错误）找到使用的偏移量：

dbgcmd

```
kd> u 80136039-2 l1      //  looks ok, its a call
NT!_KiServiceExit+0x1c:
80136037 ffd3             call    ebx
kd> u fe682ae4-2 l1      //  paged out (all zeroes) unknown
rdr!_RdrSectionInfo+0x2a:
fe682ae2 0000             add     [eax],al
kd> u fe68f57a-6 l1      //  looks ok, its a call, but not anything above
rdr!_RdrDereferenceDiscardableCode+0xae:
fe68f574 ff15203568fe     call dword ptr
[rdr!__imp__ExReleaseResourceForThreadLite]
kd> u fe682a78-6 l1      //  paged out (all zeroes) unknown

rdr!_DiscCodeInitialized+0x2:
fe682a72 0000             add     [eax],al
kd> u  fe6a1198-5 l1      //  looks good, call to something above
rdr!_SubmitTdiRequest+0xa9:
fe6a1193 e82ee3feff       call  rdr!_RdrDereferenceDiscardableCode
(fe68f4c6)
kd> u 8011c901-2 l1      //  not good, its a jump in the function
NT!_KeSuspendThread+0x11:
8011c8ff 7424             jz      NT!_KeSuspendThread+0x37 (8011c925)
kd> u 80127797-2 l1      //  looks good, an int 2e -> KiServiceExit

NT!_ZwCloseObjectAuditAlarm+0x5:
80127795 cd2e             int     2e
kd> u 80110008-2 l1      //  not good, its a test instruction not a call
NT!_KeWaitForMultipleObjects+0x27a:
```

```
80110006 85c9              test    ecx,ecx
kd> u 80110008-5 l1      //  paged out (all zeroes) unknown
NT!_KeWaitForMultipleObjects+0x277:
80110003 0000              add     [eax],al
kd> u fe6a1430-6 l1      //  looks good its a call to ZwClose...
rdr!_RdrTdiCloseConnection+0x4:
fe6a142a ff15f83468fe      call    dword ptr [rdr!__imp__ZwClose (fe6834f8)]

kd> u fe6a10ae-2 l1      //  paged out (all zeroes) unknown
rdr!_RdrTdiDisconnect+0x54:
fe6a10ac 0000              add     [eax],al
kd> u  fe6b2c04-5 l1      //  looks good, call to something above
rdr!_CleanupTransportConnection+0x5f:
fe6b2bff e854e4feff        call    rdr!_RdrTdiDisconnect (fe6a1058)
kd> u fe685968-5 l1      //  looks good, call to immediately above
rdr!_RdrReconnectConnection+0x1b1:
fe685963 e838d20200        call    rdr!_CleanupTransportConnection (fe6b2ba0)

kd> u fe682050-2 l1      //  paged out (all zeroes) unknown
rdr!__strnicmp+0xa8:
fe68204e 0000              add     [eax],al
```

基于此， **RdrReconnectConnection** 似乎名为 **CleanupTransportConnection**，到 **RdrTdiDisconnect**，到 **ZwCloseObjectAuditAlarm**，到 **KiServiceExit**。 堆栈上的其他函数可能是以前活动堆栈的剩余部分。

在这种情况下，堆栈跟踪正常工作。 下面是用于检查答案的实际堆栈跟踪：

dbgcmd

```
kd> k
ChildEBP RetAddr
fe4cc978 80136039 NT!_NtClose+0xd
fe4cc978 80127797 NT!_KiServiceExit+0x1e

fe4cc9f4 fe6a1430 NT!_ZwCloseObjectAuditAlarm+0x7
fe4cca10 fe6b2c04 rdr!_RdrTdiCloseConnection+0xa
fe4cca28 fe685968 rdr!_CleanupTransportConnection+0x64
fe4cca78 fe688157 rdr!_RdrReconnectConnection+0x1b6
fe4ccbd4 80106b1e rdr!_RdrFsdCreate+0x45b
fe4ccbe8 8014b289 NT!IofCallDriver+0x38
fe4ccc98 8014decd NT!_IopParseDevice+0x693
fe4ccd08 8014d6d2 NT!_ObpLookupObjectName+0x487
fe4ccde4 8014d3ad NT!_ObOpenObjectByName+0xa2
fe4cce90 8016660d NT!_IoCreateFile+0x433
fe4cced0 80136039 NT!_NtCreateFile+0x2d
```

第一个条目是基于堆栈跟踪的当前位置，但除此之外，堆栈一直正确到调用 **RdrReconnectConnection** 的点。 同一过程可用于跟踪整个堆栈。 对于更精确的手动堆

栈遍查方法，需要取消组合每个潜在函数，并按照每次 **推送** 和 **弹出** 操作来标识堆栈上的每个 DWORD。

调试包含 JScript 帧的堆栈跟踪

项目 • 2023/06/15

JScript 堆栈转储创建和使用功能的工作原理是收集 JScript 帧并将其与调试器物理帧拼接在一起。 有时在 x86 平台上，调试器会错误地构造堆栈跟踪。

如果堆栈包含你认为可能不正确的 JScript 帧，请在调试器中输入以下命令。

.stkwalk_force_frame_pointer 1

调试应用程序错误

项目 • 2023/07/25

用户模式应用程序中可能存在各种错误。

最常见的故障类型包括访问冲突、对齐错误、异常、关键节超时 (死锁) 和页内 I/O 错误。

访问冲突和数据类型不一致是最常见的。 当取消引用无效指针时，通常会发生这些错误。 追溯可能位于导致错误的函数上，或者与将无效参数传递给故障函数的早期函数有关。

用户模式异常有许多可能的原因。 如果发生未知异常，请尽可能在 ntstatus.h 或 winerror.h 中找到它。

当一个线程长时间等待关键节时， (或可能的死锁) 会发生严重节超时。 这些操作难以调试，需要对堆栈跟踪进行深入分析。

页内 I/O 错误几乎总是硬件故障。 可以在 ntstatus.h 中双击检查状态代码进行验证。

另请参阅

显示关键节

关键节超时 (用户模式)

重新附加到目标应用程序

项目 • 2023/10/26

如果调试器冻结或停止响应 (即在执行用户模式调试时) *崩溃* ，则可以将新的调试器附加到现有进程。

注意 此方法仅在 Microsoft Windows XP 和更高版本的 Windows 上受支持。 此方法不依赖于调试器是最初创建进程还是附加到现有进程。 此方法不依赖于是否使用了 **-pd** 选项。

若要将调试器重新附加到现有目标应用程序，请执行以下操作：

1. 确定目标应用程序的进程 ID。
2. 启动 CDB 或 WinDbg 的新实例。 使用 **-pe** 命令行选项。

   控制台

   ```
   Debugger -pe -p PID
   ```

   还可以使用其他 命令行选项。

   还可以使用 **.attach (附加到进程)** 命令以及 **-e** 选项，从休眠调试器进行连接。
3. 完成附加后，结束原始调试器进程。
4. 如果进程未正确响应，则其暂停计数可能过高。 可以使用 **~m (“恢复线程)** ”命令来减少暂停计数。 有关挂起计数的详细信息，请参阅 控制进程和线程。

如果原始调试器仍然正常运行，则此方法可能不起作用。 这两个调试器正在争用调试事件，Windows 操作系统不一定将所有调试事件分配给新的调试器。

如果原始调试器在附加新调试器之前已结束，则目标应用程序也会关闭。 (但是，如果调试器附加了 **-pd** 选项，然后正常退出，则目标应用程序将继续运行。在这种情况下，第二个调试器可以在不使用 **-pe** 选项的情况下附加到目标应用程序。)

如果已在调试某个进程，并且想要与进程分离，但使其处于调试状态，则可以使用 **.abandon (放弃进程)** 命令。 此命令后，任何 Windows 调试器都可以使用本主题中所述的过程重新附加到进程。

使目标计算机崩溃和重新启动目标计算机

项目 • 2024/07/08

执行内核调试时，可以通过发出 .crash（**强制系统崩溃**）命令，导致目标计算机停止响应（即*崩溃*或 *bug 检查*）。 此命令立即导致目标计算机停止响应。 如果已启用故障转储，调试器将写入内核模式转储文件。 （有关这些文件的详细信息，请参阅 创建内核模式转储文件。

若要重启目标计算机，请使用 .reboot （**重新启动目标计算机**） 命令。

如果希望目标计算机创建故障转储文件，然后重新启动，则应发出 .crash **命令，然后发出** .reboot 命令。 如果只想重启， **则不需要** .crash 命令。

在启动过程的早期阶段，主机与目标计算机之间的连接将丢失。 调试器不提供有关目标计算机的信息。

连接断开后，调试器会关闭所有符号文件并卸载所有调试器扩展。 此时，如果运行的是 KD 或 CDB，则所有断点都将丢失。 在 WinDbg 中，可以保存当前工作区。 此操作保存所有断点。

如果此时想要结束调试会话，请使用 Ctrl+B 命令（在 KD 中），或在“文件”**菜单上单击**“**退出**”（在 WinDbg 中）。

如果未退出调试器，则完成足够的启动过程后，连接将重新建立。 此时会重新加载符号和扩展。 如果运行的是 WinDbg，则会重新加载内核模式工作区。

可以指示调试器在重启过程中在两次可能的时间自动闯入目标计算机：

- 将第一个内核模块加载到内存中时
- 内核初始化时

若要在第一个内核模块加载时设置自动断点，请使用 **-d** 命令行选项。

还可以在调试器运行后更改中断状态：

- 控制初始模块加载和内核初始化断点，如所有异常和事件。 当发生这些事件或忽略这些事件时，可以进入调试器。 还可以在命中这些断点时自动执行指定的命令。 有关详细信息，请参阅控制异常和事件。
- 在 **KD 中使用** Ctrl+K 快捷键、**WinDbg 中的** Ctrl+Alt+K 快捷键和**调试 |内核连接 |WinDbg 中的**“**循环初始中断**”命令可更改中断状态。 每次使用这些命令时，调试器都会切换三种状态：没有自动中断、内核初始化时中断，在第一个内核模块加载时中断。 此方法不能同时激活两个自动断点。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

与目标计算机同步

项目 • 2023/10/26

有时，在内核模式调试期间，目标计算机停止响应调试器。

在 KD 中，可以按 **CTRL+R (重新同步)** 然后按 Enter 与目标计算机同步。 在 WinDbg 中，使用 **CTRL+Alt+R** 或调试 |内核连接 |重新同步。

这些命令经常还原主机和目标之间的通信。 但是，重新同步可能并不总是成功，尤其是在使用 1394 内核连接时。

.restart (Restart Kernel Connection) 命令提供了一种更强大的重新同步方法。 此命令等效于退出调试器，然后将新的调试器附加到现有会话。 此命令仅在 KD 中可用，在 WinDbg 中不可用。

通过 remote.exe执行远程调试时，**.restart** 命令最有用。 (在此类调试中，可能很难结束并重启 debugger.) 但是，如果通过调试器执行远程调试，则不能从调试客户端使用 **.restart** 。

查找内存泄漏

项目 • 2024/08/31

当进程从分页或非分页池中分配内存，但没有释放内存时，就会发生内存泄漏。 因此，这些有限的内存池会随着时间推移而耗尽，从而导致 Windows 运行速度减慢。 如果内存完全耗尽，可能会导致故障。

本节内容

展开表

文章	说明
使用 perfmon 确定是否存在内存泄漏	介绍了一种在您不确定系统上是否存在内存泄漏时可以使用的技术。
查找内核模式内存泄漏	介绍如何查找由内核模式驱动程序或组件引起的泄漏。
查找用户模式内存泄漏	介绍如何查找由用户模式驱动程序或应用程序引起的泄漏。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用 perfmon 确定是否存在内存泄漏

项目 • 2024/08/31

如果 Windows 性能随着时间的推移而下降，并且您怀疑可能涉及内存泄漏，请使用 Windows 性能监视器调查是否存在内存泄漏。 此过程不会告诉您泄漏的来源是什么，也不会告诉您泄漏是用户模式还是内核模式。

首先启动性能监视器。 若要打开性能监视器，请使用以下过程之一：

- 打开“开始”菜单，搜索“性能监视器”，然后单击结果
- 使用 `Windows Key + R` 键盘快捷方式打开*运行*命令，键入 `perfmon`，并单击“确定”以打开。

添加计数器

打开性能监视器后，将以下计数器添加到主性能监视器图中。 右键单击*监视工具*下的*性能监视器*，然后选择**属性**。

- **内存**-->**Pool Nonpaged Bytes**
- **内存**-->**池分页字节**
- **分页文件**-->**% 使用情况**

设置持续时间以捕获足够的活动

要捕获随时间变化的泄漏图，请将*采样间隔*时间设置为 600 秒，以每十分钟测量一次值。 设置*持续时间*以捕获足够的活动。 例如，若要将其设置为 24 小时，该值为 `60*60*24 = 86,400`。可能还需要将数据记录到文件以供以后检查。

启动应用程序和监视器

启动你认为导致泄漏的应用程序或测试。 允许应用程序或测试在一段时间内不受干扰地运行；在此期间不要使用目标计算机。 泄漏通常很慢，可能需要几个小时才能检测到。 等待几个小时后再判断是否发生了泄漏。

监视性能监视器计数器。 测试启动后，计数器值将迅速变化，内存池值可能需要一些时间才能达到稳定状态。

用户模式内存泄漏始终位于可分页池中，并导致**池分页字节**计数器和页面文件**使用情况**计数器随时间推移稳步增加。 内核模式内存泄漏通常会耗尽非分页池，导致**池非分页字节**

计数器增加，尽管可分页内存也会受到影响。 有时，这些计数器可能会显示误报，因为应用程序正在缓存数据。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

查找内核模式内存泄漏

项目 • 2023/06/14

使用以下技术确定内核模式内存泄漏的原因：

使用 PoolMon 查找内核模式内存泄漏

使用内核调试程序查找内核模式内存泄漏

使用驱动程序验证程序查找内核模式内存泄漏

如果不知道哪个内核模式驱动程序或组件负责泄漏，则应先使用 PoolMon 技术。 此技术显示与内存泄漏关联的池标记;使用此池标记的驱动程序或组件负责泄漏。

如果已确定负责的驱动程序或组件，请使用前面列表中的第二种和第三种方法更具体地确定泄漏的原因。

使用 PoolMon 查找内核模式内存泄漏

项目 • 2024/08/26

如果怀疑存在内核模式内存泄漏，可以使用 PoolMon 工具确定哪个池标记与泄漏相关联。

PoolMon (Poolmon.exe) 按池标记名称监视池内存使用情况。 此工具包含在 Windows 驱动程序工具包 (WDK) 中。 有关详细信息，请参阅 PoolMon。

GFlags 池设置

某些 GFlags 设置（如特殊池）将影响内存池的使用方式。 有关详细信息，请参阅 GFlags 和 配置特殊池。

使用 PoolMon

PoolMon 标头显示总分页和非分页池字节数。 这些列显示每个池标记的池用法。 屏幕每隔几秒钟自动更新一次。 例如：

dbgcmd

```
Memory: 16224K Avail: 4564K PageFlts: 31 InRam Krnl: 684K P: 680K
Commit: 24140K Limit: 24952K Peak: 24932K Pool N: 744K P: 2180K

## Tag   Type  Allocs       Frees        Diff    Bytes       Per Alloc


CM    Paged   1283  ( 0)  1002  ( 0)   281  1377312   ( 0)  4901
Strg  Paged  10385 ( 10)  6658  ( 4)  3727   317952 ( 512)    85
Fat   Paged   6662  ( 8)  4971  ( 6)  1691   174560 ( 128)   103
MmSt  Paged    614  ( 0)   441  ( 0)   173    83456   ( 0)   482
```

PoolMon 具有命令键，这些命令键根据各种条件对输出进行排序。 选择与每个命令关联的字母，以便使用数据。 每个命令需要几秒钟才能正常工作。

排序命令包括：

展开表

命令键	操作
P	将显示的标记限制为非分页池字节和/或分页池字节。 按该顺序重复按 P 循环遍历每个选

命令键	操作
	项。
B	按最大字节使用量对标记进行排序。
M	按最大字节分配数对标记进行排序。
T	按标记名称按字母顺序对标记进行排序。
E	使显示内容在底部包含分页和未分页的总计。
A	按分配大小对标记进行排序。
F	按自由操作对标记进行排序。
S	按分配和释放之间的差异对标记进行排序。
Q	退出 PoolMon。

在 PoolMon 中显示驱动程序名称

可以使用 PoolMon /g 参数显示分配每个池标记的 Windows 组件和常用驱动程序的名称。 如果在具有特定标记的分配中发现问题，此功能可帮助你识别有问题的组件或驱动程序。

组件和驱动程序列在“Mapped_Driver”列中，即显示中最右侧的列。 Mapped_Driver 列的数据来自 pooltag.txt，该文件随 WDK 一起安装。

以下命令演示如何使用 /g 参数添加Mapped_Driver列。

```
poolmon /g "C:\Program Files (x86)\Windows
Kits\10\Debuggers\x64\triage\pooltag.txt"
```

显示特定池

使用 /i 参数显示以特定字符串开头的池标记，例如 *Hid*。

```
poolmon /iHid? /g "C:\Program Files (x86)\Windows
Kits\10\Debuggers\x64\triage\pooltag.txt"
```

```
dbgcmd

 Memory:33473120K Avail:20055132K  PageFlts:     5   InRam Krnl:10444K
P:1843072K
 Commit:15035764K Limit:67027552K Peak:16677444K            Pool N:1023400K
P:1955448K
```

```
 System pool information
 Tag  Type     Allocs            Frees            Diff       Bytes
Per Alloc Mapped_Driver

 HidC Paged      1667 (   0)      1659 (   0)        8         896 (
0)         112 [hidclass.sys - HID Class d
 HidC Nonp      17375 (   0)     17256 (   0)      119       19808 (
0)         166 [hidclass.sys - HID Class d
 HidP Nonp       1014 (   0)       998 (   0)       16        6704 (
0)         419 [hidparse.sys - HID Parser]
```

使用 PoolMon 实用工具查找内存泄漏

下面是使用 PoolMon 实用工具查找内存泄漏的一种方法：

1. 启动 PoolMon。
2. 如果已确定泄漏发生在非分页池中，请选择 P 一次。 如果已确定它发生在分页池中，请选择 P 两次。 如果不知道，请不要选择 P，以便包括这两种类型的池。
3. 选择 B 可按最大字节使用量对显示进行排序。
4. 开始测试。 从屏幕复制输出，例如，拍摄屏幕截图并保存。
5. 每半小时拍摄一次新的屏幕截图。 通过比较屏幕截图，确定哪些标记的字节正在增加。
6. 停止测试并等待几个小时。 确定此时释放了多少标记。

通常，在应用程序达到稳定运行状态后，它会以相同的速率分配内存和可用内存。 如果分配内存的速度比释放内存快，则内存使用量会随着时间的推移而增长。 这通常表示内存泄漏。

解决泄漏问题

确定与泄漏关联的池标记后，你可能已获得有关泄漏的全部需要了解的内容。 如果需要确定分配例程的特定实例导致泄漏，请参阅 使用内核调试器查找内核模式内存泄漏。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用内核调试程序查找内核模式内存泄漏

项目 • 2023/06/14

内核调试器确定内核模式内存泄漏的精确位置。

启用池标记

必须先使用 GFlags 启用池标记。 GFlags 包含在 Windows 调试工具中。 启动 GFlags，选择“**系统注册表**”选项卡，检查“**启用池标记**”框，然后选择“**应用**”。 必须重启 Windows 才能使此设置生效。

在 Windows Server 2003 及更高版本的 Windows 上，始终启用池标记。

确定泄漏的池标记

若要确定哪个池标记与泄漏相关联，通常最简单的方法是使用 PoolMon 工具执行此步骤。 有关详细信息，请参阅 使用 PoolMon 查找Kernel-Mode内存泄漏。

或者，可以使用内核调试器查找与大型池分配关联的标记。 为此，请遵循以下过程：

1. 使用 **.reload (重新加载模块) 命令重新加载所有模块** 。
2. 使用 **！ poolused** 扩展。 包括标志“4”以按分页内存使用对输出进行排序：

   dbgcmd

   ```
   kd> !poolused 4
   Sorting by Paged Pool Consumed

   Pool Used:
               NonPaged            Paged
   Tag    Allocs     Used    Allocs     Used
   Abc         0        0     36405 33930272
   Tron        0        0       552  7863232
   IoN7        0        0     10939   998432
   Gla5        1      128      2222   924352
   Ggb         0        0        22   828384
   ```

3. 确定哪个池标记与最大内存使用率相关联。 在此示例中，使用标记“Abc”的驱动程序使用的内存最多，几乎为 34 MB。 因此，内存泄漏最有可能在此驱动程序中。

查找泄漏

确定与泄漏关联的池标记后，请按照以下步骤查找泄漏本身：

1. 使用 ed (Enter Values) 命令修改全局系统变量 PoolHitTag 的值。 每当使用与值匹配的池标记时，此全局变量会导致调试器中断。

2. 将 PoolHitTag 设置为你怀疑为内存泄漏源的标记。 应指定模块名称“nt”以加快符号解析速度。 标记值必须以 little-endian 格式输入，（即向后）。 由于池标记始终为四个字符，因此此标记实际上是 A-b-c 空间，而不仅仅是 A-b-c。 因此，请使用以下命令：

   ```
   dbgcmd

   kd> ed nt!poolhittag ' cbA'
   ```

3. 若要验证 PoolHitTag 的当前值，请使用 db (Display Memory) 命令：

   ```
   dbgcmd

   kd> db nt!poolhittag L4
   820f2ba4  41 62 63 20           Abc
   ```

4. 每次使用 **标记** Abc 分配或释放池时，调试器都会中断。 每次调试器中断其中一个分配或免费操作时，请使用 kb (**显示堆栈回溯**) 调试器命令查看堆栈跟踪。

使用此过程，可以确定驻留在内存中的哪些代码使用标记 Abc 过度分配池。

若要清除断点，请将 PoolHitTag 设置为零：

```
dbgcmd

kd> ed nt!poolhittag 0
```

如果有多个不同位置分配了此标记的内存，并且这些内存位于已编写的应用程序或驱动程序中，则可以更改源代码，以便对每个分配使用唯一标记。

如果无法重新编译程序，但想要确定代码中导致泄漏的多个可能位置之一，则可以在每个位置取消组合代码，并使用调试器编辑驻留在内存中的此代码，以便每个实例使用不同的（和以前未使用的）池标记。 然后，允许系统运行几分钟或更长的时间。 经过一段时间后，使用调试器再次中断，并使用 ！ poolfind 扩展查找与每个新标记关联的所有池分配。

使用驱动程序验证程序查找内核模式内存泄漏

项目 • 2023/06/15

驱动程序验证程序确定内核模式驱动程序是否正在泄漏内存。

驱动程序验证程序的池跟踪功能监视指定驱动程序进行的内存分配。 卸载驱动程序时，驱动程序验证程序会验证驱动程序所做的所有分配是否已释放。 如果尚未释放驱动程序的某些分配，则会发出 bug 检查，bug 的参数检查指示问题的性质。

当此功能处于活动状态时，请使用驱动程序验证程序管理器图形界面来监视池分配统计信息。 如果内核调试器附加到驱动程序，请使用 **！ verifier 0x3** 扩展来显示分配统计信息。

如果驱动程序使用直接内存访问 (DMA)，则驱动程序验证程序的 DMA 验证功能也有助于查找内存泄漏。 DMA 验证测试许多常见的 DMA 例程误用，包括无法释放常见缓冲区和其他可能导致内存泄漏的错误。 如果在此选项处于活动状态时附加了内核调试器，请使用 **！ dma** 扩展显示分配统计信息。

有关驱动程序验证程序的信息，请参阅 Windows 驱动程序工具包 (WDK) 文档中的驱动程序 验证程序 。

查找用户模式内存泄漏

项目 • 2023/06/15

使用以下技术来确定用户模式内存泄漏的原因：

- 使用性能监视器查找用户模式内存泄漏
- 使用 UMDH 查找用户模式内存泄漏

第一种方法确定哪个进程正在泄漏内存。 在知道涉及哪个进程后，第二种方法可以确定有故障的特定例程。

使用性能监视器查找用户模式内存泄漏

项目 • 2024/08/31

如果您怀疑存在用户模式内存泄漏，但不确定是哪个进程导致了泄漏，请使用性能监控器来测量各个进程的内存使用情况。

以管理员身份运行性能监视器。 右键单击*监视工具*下的*性能监视器*，然后选择**属性**以添加以下计数器：

- “进程”>“**专用字节**”（针对要检查的每个进程）
- “进程”>“**虚拟字节**”（针对要检查的每个进程）

设置*持续时间*以捕获足够的活动。 例如，将更新时间更改为 600 秒，以捕获泄漏随时间的变化图。 你可能还想要将数据记录到文件中，以供以后检查。

专用字节计数器指示进程已分配的内存总量，不包括与其他进程共享的内存量。

虚拟字节数计数器指示进程使用的虚拟地址空间的当前大小。

一些内存泄漏以分配的专用字节增加的形式出现在数据文件中。 其他内存泄漏以虚拟地址空间增加的形式出现。

在确定哪个进程泄漏内存后，使用 UMDH 工具确定出故障的特定例程。 有关详细信息，请参阅使用 UMDH 查找用户模式内存泄漏。

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用 UMDH 查找用户模式内存泄漏

项目 • 2023/06/14

用户模式转储堆 (UMDH) 实用工具与操作系统配合使用，以分析特定进程的 Windows 堆分配。 UMDH 查找特定进程中哪个例程正在泄漏内存。

UMDH 包含在 Windows 调试工具中。 有关完整详细信息，请参阅 UMDH。

准备使用 UMDH

如果尚未确定哪个进程正在泄漏内存，请先执行此操作。 有关详细信息，请参阅使用性能监视器查找User-Mode内存泄漏。

UMDH 日志中最重要的数据是堆分配的堆栈跟踪。 若要确定某个进程是否正在泄漏堆内存，请分析这些堆栈跟踪。

在使用 UMDH 显示堆栈跟踪数据之前，必须使用 GFlags 正确配置系统。 GFlags 包含在 Windows 调试工具中。

以下 GFlags 设置启用 UMDH 堆栈跟踪：

- 在 GFlags 图形界面中，选择“图像文件”选项卡，键入进程名称 (包括文件扩展名)，按 TAB 键，选择“ **创建用户模式堆栈跟踪数据库**”，然后选择“ **应用**”。

  或者，同样，使用以下 GFlags 命令行，其中 *ImageName* 是进程名称 (包括文件扩展名)：

  ```
  dbgcmd

  gflags /i ImageName +ust
  ```

  完成后，使用此命令清除 GFlag 设置。 有关详细信息，请参阅 GFlags 命令。

  ```
  dbgcmd

  gflags /i ImageName -ust
  ```

- 默认情况下，Windows 收集的堆栈跟踪数据量在 x86 处理器上限制为 32 MB，在 x64 处理器上限制为 64 MB。 如果必须增加此数据库的大小，请在 GFlags 图形界面中选择“**图像文件**”选项卡，键入进程名称，按 TAB 键，检查“**堆栈回溯 (Megs)** 检查”框，在关联的文本框中键入以 MB) (值，然后选择“**应用**”。 仅在必要时增加此数

据库，因为它可能会耗尽有限的 Windows 资源。 如果不再需要更大的大小，请将此设置返回到其原始值。

- 如果在“ **系统注册表** ”选项卡上更改了任何标志，则必须重启 Windows 才能使这些更改生效。 如果更改了“ **图像文件** ”选项卡上的任何标志，则必须重启该过程才能使更改生效。 对“ **内核标志** ”选项卡的更改会立即生效，但在下次 Windows 重启时会丢失。

在使用 UMDH 之前，必须有权访问应用程序的正确符号。 UMDH 使用环境变量 _NT_SYMBOL_PATH指定的符号路径。 将此变量设置为包含应用程序的符号的路径。 如果还包括 Windows 符号的路径，则分析可能更完整。 此符号路径的语法与调试器使用的语法相同;有关详细信息，请参阅 符号路径。

例如，如果应用程序的符号位于 C：\MySymbols，并且你希望使用 C：\MyCache 作为下游存储区来使用 Windows 符号的公共 Microsoft 符号存储，则可以使用以下命令设置符号路径：

控制台

```
set
_NT_SYMBOL_PATH=c:\mysymbols;srv*c:\mycache*https://msdl.microsoft.com/downl
oad/symbols
```

此外，为了确保结果准确，必须禁用 BSTR 缓存。 为此，请将 OANOCACHE 环境变量设置为等于 1 (1) 。 在启动要跟踪其分配的应用程序之前进行此设置。

如果需要跟踪服务进行的分配，必须将 OANOCACHE 设置为系统环境变量，然后重启 Windows 才能使此设置生效。

使用 UMDH 检测堆分配的增加

完成这些准备工作后，可以使用 UMDH 捕获有关进程的堆分配的信息。 为此，请遵循以下过程：

1. 确定要调查 的进程 (PID) 的进程 ID 。
2. 使用 UMDH 分析此过程的堆内存分配，并将其保存到日志文件。 将 -p 开关与 PID 配合使用，将 -f 开关用于日志文件的名称。 例如，如果 PID 为 124，并且想要将日志文件命名为Log1.txt，请使用以下命令：

   控制台

   ```
   umdh -p:124 -f:log1.txt
   ```

3. 使用记事本或其他程序打开日志文件。 此文件包含每个堆分配的调用堆栈、通过该调用堆栈进行的分配数以及通过该调用堆栈使用的字节数。

4. 由于要查找内存泄漏，因此单个日志文件的内容不足。 必须比较在不同时间记录的日志文件，以确定哪些分配正在增加。

   UMDH 可以比较两个不同的日志文件，并按各自的分配大小显示更改。 可以使用大于符号 (>) 将结果重定向到第三个文本文件。 可能还需要包含 -d 选项，该选项将字节和分配计数从十六进制转换为十进制。 例如，若要比较Log1.txt和Log2.txt，将比较结果保存到文件LogCompare.txt，请使用以下命令：

   控制台

   ```
   umdh log1.txt log2.txt > logcompare.txt
   ```

5. 打开LogCompare.txt文件。 其内容如下所示：

   text

   ```
   + 5320 ( f110 - 9df0) 3a allocs BackTrace00B53
   Total increase == 5320
   ```

   对于 UMDH 日志文件中标记为“BackTrace”) 的每个调用堆栈 (，在两个日志文件之间进行了比较。 在此示例中，第一个日志文件 (Log1.txt) 记录了为 BackTrace00B53 分配的0x9DF0字节，而第二个日志文件记录0xF110个字节，这意味着在捕获两个日志的时间之间分配了0x5320个额外的字节。 这些字节来自 BackTrace00B53 标识的调用堆栈。

6. 若要确定该回溯中的内容，请打开其中一个原始日志文件 (例如，Log2.txt) 并搜索“BackTrace00B53”。结果类似于以下数据：

   text

   ```
   00005320 bytes in 0x14 allocations (@ 0x00000428) by: BackTrace00B53
   ntdll!RtlDebugAllocateHeap+0x000000FD
   ntdll!RtlAllocateHeapSlowly+0x0000005A
   ntdll!RtlAllocateHeap+0x00000808
   MyApp!_heap_alloc_base+0x00000069
   MyApp!_heap_alloc_dbg+0x000001A2
   MyApp!_nh_malloc_dbg+0x00000023
   MyApp!_nh_malloc+0x00000016
   MyApp!operator new+0x0000000E
   MyApp!DisplayMyGraphics+0x0000001E
   MyApp!main+0x0000002C
   MyApp!mainCRTStartup+0x000000FC
   KERNEL32!BaseProcessStart+0x0000003D
   ```

此 UMDH 输出显示0x5320 (十进制 21280) 从调用堆栈分配的总字节数。 这些字节是从0x14 (十进制 20) 0x428 (1064) 字节的单独分配分配的。

调用堆栈的标识符为“BackTrace00B53”，并显示此堆栈中的调用。 在查看调用堆栈时，可以看到 **DisplayMyGraphics** 例程正在通过 **new** 运算符分配内存，该运算符调用例程 *malloc*，后者使用 Visual C++ 运行时库从堆中获取内存。

确定其中哪一个调用是最后一个在源代码中显式显示的调用。 在这种情况下，可能是 **新** 运算符，因为对 *malloc* 的调用是 **作为 new** 实现的一部分发生的，而不是单独的分配。 因此，**DisplayMyGraphics** 例程中 **new** 运算符的此实例会重复分配未释放的内存。

调试超时

项目 • 2023/06/15

Windows 系统上发生两个main超时：

资源超时 (内核模式)

关键节超时 (用户模式)

在许多情况下，这些问题只是线程释放资源或退出代码部分所花费的时间过长的问题。

在零售系统上，超时值设置得足够高，无法看到中断 (真正的死锁只会挂起) 。 超时值在注册表中的 **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\SessionManager**下设置。 整数值指定每个超时中的秒数。

资源超时

项目 • 2023/06/15

在资源超时期间，等待资源的线程将中断内核调试器，并显示如下消息：

控制台

```
Resource @ 800e99c0
 ActiveCount = 0001  Flags = IsOwnedExclusive sharedWaiter
 NumberOfExclusiveWaiters = 0000
  Thread = 809cd2f0, Count = 01
  Thread = 809ebc50, Count = 01
  Thread = 00000000, Count = 00
  Thread = 00000000, Count = 00
  Thread = 00000000, Count = 00
NT!DbgBreakPoint+0x4:
800cee04: 000000ad callkd
```

持有锁的线程是 (列出的第一个线程，如果它是共享锁)，则为多个线程。 若要检查该线程，请在上一个示例中的线程 ID (809cd2f0 上使用 **！** thread 扩展) 。 这将为拥有资源的线程提供堆栈。 如果它还在等待资源变为可用， **ExpWaitForResourceExclusive** 函数或 **ExpWaitForResourceShared** 函数将位于该线程的堆栈上。

ExpWaitForResource*Xxx* 的第一个参数是等待的锁。 若要了解该资源，请使用 **！ locks <*资源 ID*>** 扩展，这将为你提供另一个线程来检查。

如果到达的线程未等待另一个资源，该线程可能是问题的根源。 有关所有已保留锁的列表，请在 **kd>** 提示符下使用不带参数的 **！ locks** 扩展。

示例

控制台

```
Resource @ fc664ee0                  // Here's the resource lock address

 ActiveCount = 0001  Flags = IsOwnedExclusive ExclusiveWaiter
 NumberOfExclusiveWaiters = 0001
  Thread = ffaf5410, Count = 01                // Here's the owning thread
  Thread = 00000000, Count = 00
ntoskrnl!_DbgBreakPoint:
80131400 cc               int     3

kd> kb                                        // Start with a stack
ChildEBP RetAddr  Args to Child
fcd44980 801154c0 fc664ee0 ffab45d0 00110001 ntoskrnl!_DbgBreakPoint
fcd4499c 80102521 fc664ee0 ffb08ea8 fcd44a4c
```

```
ntoskrnl!_ExpWaitForResource+0x114        // Lock being waited on...

fcd449e8 fc6509fa e12597c8 fef27c08 fee4fca8
ntoskrnl!_ExAcquireResourceExclusiveLite+0xa5
00380020 00000000 00000000 00000000 00000000 nwrdr!_CreateScb+0x2ff

kd> !locks fc664ee0                      // !locks resource address gives
lock info
Resource @ nwrdr!_NwScavengerSpinLock (0xfc664ee0)    Exclusively owned
    Contention Count = 45
    NumberOfExclusiveWaiters = 1
     Threads: ffaf5410-01                  // Owning thread again
1 total locks, 1 locks currently held

kd> !thread ffaf5410                     // Check the owning thread

THREAD ffaf5410  Cid e7.e8  Teb: 7ffde000 WAIT: (Executive) KernelMode Non-
Alertable
    feecf698  SynchronizationEvent
IRP List:
    fef29208: (0006,00b8) Flags: 00000884  Mdl: feed8328
Not impersonating
Owning Process ffaf5690
WaitTime (seconds)      2781250
Context Switch Count    183175
UserTime                  0:00:23.0153
KernelTime                0:01:01.0187
Start Address 0x77f04644
Initial Sp fec6c000 Current Sp fec6b938
Priority 11 BasePriority 7 PriorityDecrement 0 DecrementCount 8

ChildEBP RetAddr  Args to Child
fec6b950 801044fc feecf668 feecf668 00000080 ntoskrnl!KiSwapContext+0x25
fec6b974 fc655976 feecf698 00000000 00000000
ntoskrnl!_KeWaitForSingleObject+0x218
fec6ba5c fc6509fa e1263968 fef29208 feecf668 nwrdr!_ExchangeWithWait+0x38
fec6ba28 fc6533e5 feecf668 e125b3c8 ffafae08 nwrdr!_CreateScb+0x2ff
fec6bac0 fc652f26 feecf668 fec6bae4 fef29208 nwrdr!_CreateRemoteFile+0x2c9
fec6bb6c fc652b14 feecf668 fef29208 fee50b60 nwrdr!_NwCommonCreate+0x3a2

fec6bbac 80107aea fee50b60 fef29208 804052ac nwrdr!_NwFsdCreate+0x56
fec6bbc0 80142792 fef37700 fec6bdbc fee50b28 ntoskrnl!IofCallDriver+0x38
fec6bd10 80145403 fee50b60 00000000 fec6bdbc ntoskrnl!_IopParseDevice+0x6a0
fec6bd7c 80144c0c 00000000 fec6be34 00000040
ntoskrnl!_ObpLookupObjectName+0x479
fec6be5c 80127803 0012dd64 00000000 80127701
ntoskrnl!_ObOpenObjectByName+0xa2
fec6bef4 801385c3 0012dd64 0012dd3c 00000000
ntoskrnl!_NtQueryAttributesFile+0xc1
fec6bef4 77f716ab 0012dd64 0012dd3c 00000000 ntoskrnl!_KiSystemService+0x83

0012dd20 00000000 00000000 00000000 00000000
ntdll!_ZwQueryAttributesFile+0xb
```

关键节超时

项目 • 2023/10/26

当堆栈跟踪在堆栈顶部附近显示例程 **RtlpWaitForCriticalSection** 时，可以识别某些类型的关键节超时。 另一种关键节超时可能是死锁应用程序错误。

与资源超时一样， **！ ntsdexts.locks** 扩展将提供当前持有的锁以及拥有锁的线程的列表。 与资源超时不同，给定的线程 ID 没有立即用处。 这些是系统 ID，不会直接映射到线程编号。

与 **ExpWaitForResource*Xxx*** *一样，锁标识符是 **RtlpWaitForCriticalSection* 的第一个参数。 继续跟踪等待链，直到找到循环或最终线程未等待关键部分超时。

其他信息

有关可显示关键节信息的其他命令和扩展，请参阅 显示关键节。 有关关键部分的详细信息，请参阅 mark Russinovich 和 David Solomon 的 Microsoft Windows SDK 文档和 *Microsoft Windows Internals*。

调试关键超时的示例

首先显示堆栈：

dbgcmd

```
0:024> kb

ChildEBP RetAddr  Args to Child
0569fca4 77f79c78 77f71000 002a6b88 7fffffff ntdll!_DbgBreakPoint
0569fd04 77f71048 5ffa9f9c 5fef0b4b 5ffa9f9c
ntdll!_RtlpWaitForCriticalSection+0x89
0569fd0c 5fef0b4b 5ffa9f9c 002a6b88 002a0019
ntdll!_RtlEnterCriticalSection+0x48
0569fd70 5fedf83f 002a6b88 0569fdc0 0000003e
winsrv!_StreamScrollRegion+0x1f0
0569fd8c 5fedfa5b 002a6b88 00190000 00000000
winsrv!_AdjustCursorPosition+0x8e
0569fdc0 5fedf678 0569ff18 0031c200 0335ee88 winsrv!_DoWriteConsole+0x104

0569fefc 5fe6311b 0569ff18 0569ffd0 00000005 winsrv!_SrvWriteConsole+0x96
0569fff4 00000000 00000000 00000024 00000024
csrsrv!_CsrApiRequestThread+0x4ff
```

现在，使用 **！ ntsdexts.locks** 扩展查找关键部分：

dbgcmd

```
0:024> !locks
CritSec winsrv!_ScrollBufferLock at 5ffa9f9c        5ffa9f9c is the first
one
LockCount          5
RecursionCount     1
OwningThread       88         // here's the owning thread ID
EntryCount         11c
ContentionCount    135
*** Locked

CritSec winsrv!_gcsUserSrv+0 at 5ffa91b4     //second critical section found
below

LockCount          8
RecursionCount     1
OwningThread       6d         // second owning thread
EntryCount         1d6c
ContentionCount    1d47
*** Locked
```

现在，搜索 ID 号0x6D的线程：

dbgcmd

```
0:024> ~
  0  id: 16.15   Teb 7ffdd000 Unfrozen
  1  id: 16.13   Teb 7ffdb000 Unfrozen
  2  id: 16.30   Teb 7ffda000 Unfrozen
  3  id: 16.2f   Teb 7ffd9000 Unfrozen
  4  id: 16.2e   Teb 7ffd8000 Unfrozen
  5  id: 16.6c   Teb 7ff6c000 Unfrozen
  6  id: 16.6d   Teb 7ff68000 Unfrozen    // this thread owns the second
critical section
  7  id: 16.2d   Teb 7ffd7000 Unfrozen
  8  id: 16.33   Teb 7ffd6000 Unfrozen
  9  id: 16.42   Teb 7ff6f000 Unfrozen
 10  id: 16.6f   Teb 7ff6e000 Unfrozen
 11  id: 16.6e   Teb 7ffd5000 Unfrozen
 12  id: 16.52   Teb 7ff6b000 Unfrozen
 13  id: 16.61   Teb 7ff6a000 Unfrozen
 14  id: 16.7e   Teb 7ff69000 Unfrozen
 15  id: 16.43   Teb 7ff67000 Unfrozen
 16  id: 16.89   Teb 7ff50000 Unfrozen
 17  id: 16.95   Teb 7ff65000 Unfrozen
 18  id: 16.90   Teb 7ff64000 Unfrozen
 19  id: 16.71   Teb 7ff63000 Unfrozen
 20  id: 16.bb   Teb 7ff62000 Unfrozen
 21  id: 16.88   Teb 7ff61000 Unfrozen    // this thread owns the first
critical section
 22  id: 16.cd   Teb 7ff5e000 Unfrozen
```

```
 23  id: 16.c1   Teb 7ff5f000 Unfrozen
 24  id: 16.bd   Teb 7ff5d000 Unfrozen
```

线程 21 拥有第一个关键部分。 使该线程成为活动线程，并获取堆栈跟踪：

dbgcmd

```
0:024> ~21s
ntdll!_ZwWaitForSingleObject+0xb:
77f71bfb c20c00          ret     0xc

0:021> kb

ChildEBP RetAddr  Args to Child
0556fc44 77f79c20 00000110 00000000 77fa4700
ntdll!_ZwWaitForSingleObject+0xb
0556fcb0 77f71048 5ffa91b4 5feb4f7e 5ffa91b4
ntdll!_RtlpWaitForCriticalSection+0x31
0556fcb8 5feb4f7e 5ffa91b4 0556fd70 77f71000
ntdll!_RtlEnterCriticalSection+0x48
0556fcf4 5fef0b76 01302005 00000000 fffffff4 winsrv!__ScrollDC+0x14
0556fd70 5fedf83f 002bd880 0556fdc0 00000025
winsrv!_StreamScrollRegion+0x21b
0556fd8c 5fedfa5b 002bd880 00190000 00000000
winsrv!_AdjustCursorPosition+0x8e

0556fdc0 5fedf678 0556ff18 002bdf70 002a4d58 winsrv!_DoWriteConsole+0x104
0556fefc 5fe6311b 0556ff18 0556ffd0 00000005 winsrv!_SrvWriteConsole+0x96
0556fff4 00000000 00000000 00000024 00000024
csrsrv!_CsrApiRequestThread+0x4ff
```

线程 6 拥有第二个关键部分。 同时检查其堆栈：

dbgcmd

```
0:021> ~6s
winsrv!_PtiFromThreadId+0xd:
5fe8429a 394858          cmp     [eax+0x58],ecx
ds:0023:7f504da8=000000f8

0:006> kb

ChildEBP RetAddr  Args to Child
01ecfeb4 5fecd0d7 00000086 00000000 7f5738e0 winsrv!_PtiFromThreadId+0xd
01ecfed0 5feccf62 00000086 01ecfff4 00000113 winsrv!__GetThreadDesktop+0x12
01ecfefc 5fe6311b 01ecff18 01ecffd0 00000005 winsrv!___GetThreadDesktop+0x8b
01ecfff4 00000000 00000000 00000024 00000024
csrsrv!_CsrApiRequestThread+0x4ff
```

线程 21 在其堆栈顶部附近具有 **RtlpWaitForCriticalSection** 。 线程 6 没有。 因此，应进一步调查线程 21 以确定它正在等待的内容。 检查与锁定的线程关联的源代码，以查看锁是否已正确保留和释放。

应用程序验证工具

应用程序验证程序可以截获和包装调用，以检测不正确的锁使用情况。 应用程序验证程序可帮助查找以下问题。

- 使用未初始化的关键部分。
- 线程释放它不拥有的关键部分。
- 重新初始化关键部分。
- 过度释放关键部分。
- 线程处于不应拥有 (ExitThread、TerminateThread、ThreadPool、RPC 线程) 但实际上拥有锁的状态。

有关详细信息，请参阅 应用程序验证程序 - 概述。

另请参阅

有关孤立关键部分的代码示例和示例调试会话，请参阅 Mario Hewardt 和 Daniel Pravat *的高级 Windows 调试* 。

显示关键节

关键节超时 (用户模式)

！ ntsdexts.locks

！ Cs

调试已停止的系统

项目 • 2023/06/15

有时，计算机可以在不实际启动 bug 的情况下停止响应检查。 这种“冻结”可以以各种形式出现：

- 鼠标指针可以移动，但不会影响屏幕上的任何窗口。
- 整个屏幕处于静止状态，鼠标指针不会移动，但在内存和磁盘之间继续分页。
- 屏幕处于静止状态，磁盘处于无提示状态。

如果鼠标指针移动或磁盘有分页，这通常是由于客户端服务器Run-Time子系统 (CSRSS) 出现问题。

如果 NTSD 在 CSRSS 上运行，请按 F12 并转储出每个线程，以查看是否有异常内容。 (有关详细信息，请参阅调试 CSRSS 。)

如果对 CSRSS 的检查没有发现任何信息，那么问题可能毕竟出在内核上。

如果没有鼠标移动或分页，则几乎可以肯定这是一个内核问题。

分析此类内核崩溃通常是一项艰巨的任务。 若要开始，请使用 **CTRL+C**) 或使用 **CTRL+BREAK**) 进入 WinDbg (KD (。 现在可以使用调试器命令来检查情况。

在这种情况下，一些有用的技术包括：

查找失败的进程

调试中断风暴

查找失败的进程

项目 • 2024/07/08

在查找失败的进程之前，请确保处于接受处理器的上下文中。 若要确定接受处理器，请使用 **每个处理器上的！ rf** 扩展，并查找已加载异常处理程序的处理器。 接受处理器的异常处理程序具有除0xFFFFFFFF以外的地址。

例如，由于此处理器上的 NtTib.ExceptionList **的**地址0xFFFFFFFF，因此这不是具有失败进程的处理器：

dbgcmd

```
0: kd> !pcr
PCR Processor 0 @ffdff000
 NtTib.ExceptionList: ffffffff
            NtTib.StackBase: 80470650
           NtTib.StackLimit: 8046d860
         NtTib.SubSystemTib: 00000000
              NtTib.Version: 00000000
          NtTib.UserPointer: 00000000
              NtTib.SelfTib: 00000000

                    SelfPcr: ffdff000
                       Prcb: ffdff120
                       Irql: 00000000
                        IRR: 00000000
                        IDR: ffffffff
              InterruptMode: 00000000
                        IDT: 80036400
                        GDT: 80036000
                        TSS: 80257000

              CurrentThread: 8046c610
                 NextThread: 00000000
                 IdleThread: 8046c610

                  DpcQueue:
```

但是，处理器 1 的结果大相径庭。 在这种情况下，NtTib.ExceptionList **的值**为 **f0823cc0**，而不是0xFFFFFFFF，表示这是发生异常的处理器。

dbgcmd

```
0: kd> ~1
1: kd> !pcr
PCR Processor 1 @81497000
 NtTib.ExceptionList: f0823cc0
            NtTib.StackBase: f0823df0
```

```
           NtTib.StackLimit: f0821000
         NtTib.SubSystemTib: 00000000
              NtTib.Version: 00000000
          NtTib.UserPointer: 00000000
              NtTib.SelfTib: 00000000

                    SelfPcr: 81497000
                       Prcb: 81497120
                       Irql: 00000000
IRR: 00000000
                        IDR: ffffffff
              InterruptMode: 00000000
                        IDT: 8149b0e8
GDT: 8149b908
                        TSS: 81498000

              CurrentThread: 81496d28
                 NextThread: 00000000
                 IdleThread: 81496d28

                  DpcQueue:
```

当处于正确的处理器上下文中时， **！进程** 扩展将显示当前正在运行的进程。

进程转储最有趣的部分包括：

- 时间（高值表示进程可能是罪魁祸首）。
- 句柄计数（这是第一个条目中 ObjectTable **后面的**括号中的数字）。
- 线程状态（许多进程具有多个线程）。 如果当前进程为 *“空闲”*，则可能是因为计算机确实处于空闲状态，或者由于一些异常问题而挂起。

尽管使用 **！process 0 7** 扩展是查找挂起系统上问题的最佳方法，但有时信息过多，无法筛选。 请改用 **！process 0 0** ，然后在 CSRSS 和其他任何可疑进程的进程句柄上使用一个 **！进程** 。

使用 **！process 0 7** 时，许多线程可能会标记为“内核堆栈不驻留”，因为这些堆栈已分页。如果这些页面仍在正在转换的缓存中，则可以在 ！process 0 7 **之前**使用 **.cache 解码器**获取详细信息：

dbgcmd

```
kd> .cache decodeptes
kd> !process 0 7
```

如果可以识别失败的进程，请使用 **！process** <*进程*> **7** 显示进程中每个线程的内核堆栈。 此输出可以识别内核模式下的问题，并显示可疑进程正在调用的内容。

除了 ！ process，以下扩展还有助于确定无响应计算机的原因：

展开表

扩展	效果
！ 准备	按优先级顺序标识已准备好运行的线程。
!kdext*.locks	标识任何保留的资源锁，以防存在零售超时的死锁。
!vm	检查虚拟内存使用情况。
！ poolused	确定一种类型的池分配是否不成比例（需要池标记）。
!memusage	检查物理内存状态。
！ 堆	检查堆的有效性。
!irpfind	在非分页池中搜索活动 IRP。

如果提供的信息未指示异常情况，请尝试在 ntoskrnl 处 **设置断点！KiSwapThread** 用于确定处理器是否停滞在一个进程中，还是仍在计划其他进程。 如果它未卡住，请设置常见函数（如 **NtReadFile**）中的断点，以确定计算机是否停滞在特定代码路径中。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

调试中断风暴

项目 • 2024/05/27

系统停滞最常见的例子之一就是中断风暴。 *中断风暴*被定义为电平触发的中断信号保持在断言状态。

以下事件可能导致中断风暴：

- 一个硬件设备在被设备驱动程序告知之后不会释放它的中断信号。
- 设备驱动程序不会指示其硬件释放中断信号，因为它没有检测到中断是从其硬件发起的。
- 即使中断不是从其硬件启动的，设备驱动程序也会声明中断。 只有当多个设备共享同一 IRQ 时，才会发生这种情况。
- 边缘电平控制寄存器 (ELCR) 设置不正确。
- 边缘和级别中断触发的设备会共享一个 IRQ（例如 COM 端口和 PCI SCSI 控制器）。

本例演示了一种检测和调试中断风暴的方法。

当计算机挂起时，使用内核调试器进行中断。 使用 **!irpfind** 扩展命令查找挂起的 IRP。 然后，使用 **!irp** 扩展获取任何挂起的 IRP 的详细信息。 例如：

dbgcmd

```
kd> !irp 81183468
Irp is active with 2 stacks 2 is current (= 0x811834fc)
 No Mdl Thread 00000000:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
 [  0, 0]   0  0 8145f470 00000000 00000000-00000000
                \Driver\E100B
                        Args: 00000000 00000000 00000000 00000000
>[ 16, 2]   0 e1 8145f470 00000000 8047f744-814187a8 Success Error Cancel
pending
                \Driver\E100B    ntoskrnl!PopCompleteSystemPowerIrp
                        Args: 00000000 00000000 00000002 00000002
```

此示例显示 \driver\e100b 没有返回 **ntoskrnl!PopCompleteSystemPowerIrp** 的 IRP。 它似乎被卡住了，可能正在经历一次中断风暴。

要进行调查，请使用 **kb** 命令来请求堆栈跟踪。 例如：

dbgcmd

```
kd> kb
ChildEBP RetAddr  Args to Child
f714ee68 8046355a 00000001 80068c10 00000030
ntoskrnl!RtlpBreakWithStatusInstruction
f714ee68 80067a4f 00000001 80068c10 00000030
ntoskrnl!KeUpdateSystemTime+0x13e
f714eeec 8046380b 01001010 0000003b f714ef00
halacpi!HalBeginSystemInterrupt+0x83
f714eeec 80463c50 01001010 0000003b f714ef00 ntoskrnl!KiChainedDispatch+0x1b
f714ef78 80067cc2 00000000 00000240 8000017c ntoskrnl!KiDispatchInterrupt
f714ef78 80501cb5 00000000 00000240 8000017c
halacpi!HalpDispatchInterrupt2ndEnt
```

请注意，以 `halacpi!HalBeginSystemInterrupt` 开头的部分是中断调度。 如果使用 **g** 命令并再次中断，则很可能会看到不同的堆栈跟踪，但仍会看到中断调度。 要确定是哪个中断导致了系统停滞，请查看传入 **HalBeginSystemInterrupt** 中的第二个参数（在本例中为 0x3B）。 标准规则是，显示的中断矢量 (0x3B) 是 IRQ 行加上 0x30，因此中断号为 0xB。 运行另一个堆栈跟踪可提供有关哪个设备发出中断服务请求 (ISR) 的详细信息。 在这种情况下，第二个堆栈跟踪的结果如下：

dbgcmd

```
kd> kb
ChildEBP RetAddr  Args to Child
f714ee24 8046355a 00000001 00000010 00000030
ntoskrnl!RtlpBreakWithStatusInstruction
f714ee24 bfe854b9 00000001 00000010 00000030
ntoskrnl!KeUpdateSystemTime+0x13e
f714eed8 f7051796 00000000 80463850 8143ec88 atimpab!AtiInterrupt+0x109
f714eee0 80463850 8143ec88 81444038 8046380b
VIDEOPRT!pVideoPortInterrupt+0x16
f714eef8 80463818 00000202 0000003b 80450bb8
ntoskrnl!KiChainedDispatch2ndLvl+0x28
f714eef8 80463c50 00000202 0000003b 80450bb8 ntoskrnl!KiChainedDispatch+0x28
f714ef78 80067cc2 00000000 00000240 8000017c ntoskrnl!KiDispatchInterrupt
f714ef78 80501cb5 00000000 00000240 8000017c
halacpi!HalpDispatchInterrupt2ndEntry+0x1b
f714f084 8045f744 f714f16c 00020019 f714f148 ntoskrnl!NtCreateKey+0x113
f714f084 8042e487 f714f16c 00020019 f714f148 ntoskrnl!KiSystemService+0xc4
f714f118 804ab556 f714f16c 00020019 f714f148 ntoskrnl!ZwCreateKey+0xb
f714f184 8041f75b f714f1e8 8000017c f714f1d0
ntoskrnl!IopCreateRegistryKeyEx+0x4e
f714f204 804965cd 8145f630 00000000 00000001
ntoskrnl!IopProcessSetInterfaceState+0x93
f714f220 bfee1eb9 8145f630 00000000 8145f5a0
ntoskrnl!IoSetDeviceInterfaceState+0x2b
f714f254 bfedb416 00000004 00000800 0045f570
NDIS!ndisMCommonHaltMiniport+0x1f
f714f268 bfed4ddb bfed0660 811a2708 811a2708 NDIS!ndisPmHaltMiniport+0x9a
f714f288 bfed5146 811a2708 00000004 8145f570 NDIS!ndisSetPower+0x1d1
```

```
f714f2a8 8041c60f 81453a30 811a2708 80475b18 NDIS!ndisPowerDispatch+0x84
f714f2bc 8044cc52 80475b18 811a2708 811a279c ntoskrnl!IopfCallDriver+0x35
f714f2d4 8044cb89 811a279c 811a2708 811a27c0 ntoskrnl!PopPresentIrp+0x62
```

系统目前正在运行显卡的 ISR。 系统将为共享 IRQ 0xB 的每个设备运行 ISR。 如果没有进程请求中断，操作系统将无限期等待，从而请求驱动程序 ISR 来处理中断。 进程也有可能处理中断并停止中断，但如果硬件损坏，则中断可能会被重新断言。

使用 **!arbiter 4** 扩展来确定 IRQ 0xB上的设备。 如果 IRQ 0xB 上只有一个设备，则问题的原因便已找到。 如果不止一个设备共享中断（99% 的情况），则需要通过手动编程 LNK 节点（会对系统状态造成破坏）或移除或禁用硬件来隔离设备。

dbgcmd

```
kd> !arbiter 4
DEVNODE 8149a008 (HTREE\ROOT\0)
  Interrupt Arbiter "RootIRQ" at 80472a20
    Allocated ranges:
      0000000000000000 - 0000000000000000   B   8149acd0
      0000000000000001 - 0000000000000001   B   8149acd0
      0000000000000002 - 0000000000000002   B   8149acd0
      0000000000000003 - 0000000000000003   B   8149acd0
      0000000000000004 - 0000000000000004   B   8149acd0
      0000000000000005 - 0000000000000005   B   8149acd0
      0000000000000006 - 0000000000000006   B   8149acd0
      0000000000000007 - 0000000000000007   B   8149acd0
      0000000000000008 - 0000000000000008   B   8149acd0
      0000000000000009 - 0000000000000009   B   8149acd0
      000000000000000a - 000000000000000a   B   8149acd0
      000000000000000b - 000000000000000b   B   8149acd0
      000000000000000c - 000000000000000c   B   8149acd0
 000000000000000d - 000000000000000d   B   8149acd0
      000000000000000e - 000000000000000e   B   8149acd0
 000000000000000f - 000000000000000f   B   8149acd0
      0000000000000010 - 0000000000000010   B   8149acd0
      0000000000000011 - 0000000000000011   B   8149acd0
      0000000000000012 - 0000000000000012   B   8149acd0
      0000000000000013 - 0000000000000013   B   8149acd0
      0000000000000014 - 0000000000000014   B   8149acd0
      0000000000000015 - 0000000000000015   B   8149acd0
      0000000000000016 - 0000000000000016   B   8149acd0
      0000000000000017 - 0000000000000017   B   8149acd0
      0000000000000018 - 0000000000000018   B   8149acd0
      0000000000000019 - 0000000000000019   B   8149acd0
      000000000000001a - 000000000000001a   B   8149acd0
      000000000000001b - 000000000000001b   B   8149acd0
      000000000000001c - 000000000000001c   B   8149acd0
      000000000000001d - 000000000000001d   B   8149acd0
 000000000000001e - 000000000000001e   B   8149acd0
      000000000000001f - 000000000000001f   B   8149acd0
 0000000000000020 - 0000000000000020   B   8149acd0
      0000000000000021 - 0000000000000021   B   8149acd0
```

```
      0000000000000022 - 0000000000000022   B   8149acd0
      0000000000000023 - 0000000000000023   B   8149acd0
      0000000000000024 - 0000000000000024   B   8149acd0
      0000000000000025 - 0000000000000025   B   8149acd0
      0000000000000026 - 0000000000000026   B   8149acd0
      0000000000000027 - 0000000000000027   B   8149acd0
      0000000000000028 - 0000000000000028   B   8149acd0
      0000000000000029 - 0000000000000029   B   8149acd0
      000000000000002a - 000000000000002a   B   8149acd0
      000000000000002b - 000000000000002b   B   8149acd0
      000000000000002c - 000000000000002c   B   8149acd0
000000000000002d - 000000000000002d   B   8149acd0
      000000000000002e - 000000000000002e   B   8149acd0
000000000000002f - 000000000000002f   B   8149acd0
      0000000000000032 - 0000000000000032   B   8149acd0
      0000000000000039 - 0000000000000039 S     814776d0  (ACPI)
   Possible allocation:
     < none >

   DEVNODE 81476f28 (ACPI_HAL\PNP0C08\0)
     Interrupt Arbiter "ACPI_IRQ" at bfff10e0
       Allocated ranges:
         0000000000000000 - 0000000000000000   B   81495bb0
         0000000000000001 - 0000000000000001       814952b0  (i8042prt)
         0000000000000003 - 0000000000000003 S     81495610  (Serial)
         0000000000000004 - 0000000000000004   B   8149acd0
         0000000000000006 - 0000000000000006       81495730  (fdc)
         0000000000000008 - 0000000000000008       81495a90
         0000000000000009 - 0000000000000009 S     814776d0  (ACPI)
         000000000000000b - 000000000000000b S
           000000000000000b - 000000000000000b S     81453c30  (ds1)
           000000000000000b - 000000000000000b S     81453a30  (E100B)
           000000000000000b - 000000000000000b S     81493c30  (uhcd)
           000000000000000b - 000000000000000b S     8145c390  (atirage3)
         000000000000000c - 000000000000000c       814953d0  (i8042prt)
000000000000000d - 000000000000000d   B   81495850
         000000000000000e - 000000000000000e       8145bb50  (atapi)
000000000000000f - 000000000000000f       8145b970  (atapi)
       Possible allocation:
         < none >
```

在这种情况下，音频、通用串行总线 (USB)、网络接口卡 (NIC) 和视频都会使用相同的 IRQ。

要找出哪个 ISR 拥有中断的所有权，可检查 ISR 的返回值。 只需将 **U** 命令与在 **!arbiter** 显示中给定的地址一起用于对 ISR 进行反汇编，然后在 ISR 的最后一条指令上设置断点（这将是一条“ret”指令）。 请注意，使用 **g <address>** 命令相当于在该地址上设置断点：

```
dbgcmd
```

```
kd> g bfe33e7b
ds1wdm!AdapterIsr+ad:
bfe33e7b c20800           ret     0x8
```

使用 **r** 命令检查寄存器。 请具体查看 EAX 寄存器。 如果以下代码示例中显示的 EAX 寄存器内容不为零，则此 ISR 要求了中断。 否则，中断未被要求，操作系统将调用下一个 ISR。 本例表明显卡没有要求中断：

dbgcmd

```
kd> r
eax=00000000 ebx=813f4ff0 ecx=00000010 edx=ffdff848 esi=8145d168
edi=813f4fc8
eip=bfe33e7b esp=f714eec4 ebp=f714eee0 iopl=0         nv up ei pl zr na po
nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00000246
ds1wdm!AdapterIsr+ad:
bfe33e7b c20800           ret     0x8
```

事实上，在这种情况下，IRQ 0xb 上的任何设备都没有要求中断。 遇到此问题时，还应检查与中断相关联的每个硬件是否真正启用。 对于 PCI，这很简单 - 查看 **!pci** 扩展输出所显示的 CMD 寄存器即可：

dbgcmd

```
kd> !pci 0 0
PCI Bus 0
00:0  8086:7190.03  Cmd[0006:.mb...]  Sts[2210:c....]  Device  Host bridge
01:0  8086:7191.03  Cmd[0107:imb..s]  Sts[0220:.6...]  PciBridge 0->1-1
PCI-PCI bridge
03:0  1073:000c.03  Cmd[0000:......]  Sts[0210:c....]  Device
SubID:1073:000c Audio device
04:0  8086:1229.05  Cmd[0007:imb...]  Sts[0290:c....]  Device
SubID:8086:0008 Ethernet
07:0  8086:7110.02  Cmd[000f:imb...]  Sts[0280:.....]  Device  ISA bridge
07:1  8086:7111.01  Cmd[0005:i.b...]  Sts[0280:.....]  Device  IDE
controller
07:2  8086:7112.01  Cmd[0005:i.b...]  Sts[0280:.....]  Device  USB host
controller
07:3  8086:7113.02  Cmd[0003:im....]  Sts[0280:.....]  Device  Class:6:80:0
```

请注意，音频芯片（标记为“音频设备”）的 CMD 寄存器为零。 这意味着此时音频芯片实际上已被禁用。 这也意味着音频芯片无法响应驱动程序的访问。

在这种情况下，需要手动重新启用音频芯片。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

调试多个目标

项目 • 2024/05/03

可以同时调试多个转储文件或实时用户模式应用程序。 每个目标包含一个或多个进程，每个进程包含一个或多个线程。

这些目标也分组到 *系统中*。 系统是一组目标，这些目标组合在一起以便于识别和操作。 系统的定义如下：

- 每个内核模式或用户模式转储文件都是一个单独的系统。
- 当你在不同的计算机上调试实时用户模式应用程序（通过使用 进程服务器（如 Dbgsrv）时，每个应用程序都是一个单独的系统。
- 在本地计算机上调试实时用户模式应用程序时，应用程序将组合成单个系统。

当前或*活动*系统是当前正在调试的系统。

获取多个目标

第一个目标以通常的方式获取。

可以使用 .attach（附加到进程）或 **.create （创建进程）命令调试其他实时用户模式应用程序**，后跟 **g （Go）** 命令。

可以使用 .opendump （Open Dump File） **命令调试其他转储文件**，后跟 **g （Go）** 命令。 还可以在启动调试器时打开多个转储文件。 若要打开多个转储文件，请在命令中包含多个 -z 开关，每个开关后跟不同的文件名。

即使进程位于不同的系统上，也可以使用上述命令。 必须在每个系统上启动进程服务器，然后将 -premote 参数与 .attach **或 .create 配合使用**来标识正确的进程服务器。 如果再次使用 **.attach** 或 **.create** 命令而不指定 -premote 参数，则调试器会附加到当前系统上的进程或创建进程。

操作系统和目标

调试开始时，当前系统是调试器最近附加到的系统。 如果发生异常，则当前系统将切换到发生此异常的系统。

若要关闭一个目标并继续调试其他目标，请使用 **.kill （Kill Process）** 命令。 可以使用 **.detach（从进程分离）** 命令或 WinDbg 的 **调试 |请改为分离 Debuggee** 菜单命令。 这些命令将调试器与目标分离，但使目标保持运行状态。

若要控制多个系统的调试，可以使用以下方法：

- **||（系统状态）** 命令显示有关一个或多个系统的信息
- **|s （设置当前系统）** 命令使你能够选择当前系统
- （仅限 WinDbg）“ 进程和线程”窗口 使你能够显示或选择系统、进程和线程

通过使用这些命令来选择当前系统，并使用标准命令 选择当前进程和线程，可以确定显示内存和寄存器的命令的上下文。

但是，不能单独执行这些进程。 **g （Go）** 命令始终导致所有目标一起执行。

注意 在一起调试实时目标和转储目标时，存在一些复杂情况，因为对于每种类型的调试，命令的行为方式不同。 例如，如果在当前系统是转储文件时使用 **g（执行）**命令，调试器将开始执行，但无法中断回调试器，因为中断命令无法识别为对转储文件调试有效。

示例

若要同时处理三个转储文件，可以使用 -z 选项在 WinDbg 启动时加载它们。

控制台

```
windbg -z c:\notepad.dmp -z c:\paint.dmp -z c:\calc.dmp
```

有关详细信息，请参阅 WinDbg 命令行选项。 还可以使用 .opendump 和 **g （Go）** 命令在调试器中加载其他转储文件。

使用 ||（系统状态） 用于确认所有三个系统都存在命令。

dbgcmd

```
||0:0:007> ||
.  0 User mini dump: c:\notepad.dmp
   1 User mini dump: C:\paint.dmp
   2 User mini dump: c:\calc.dmp
```

使用 g （Go） 命令完成转储文件的加载。

dbgcmd

```
||0:0:007> g

************* Path validation summary **************
Response                         Time (ms)     Location
```

```
Deferred                                        srv*
Symbol search path is: srv*
Executable search path is:
Windows 10 Version 15063 MP (4 procs) Free x64
Product: WinNt, suite: SingleUserTS
15063.0.amd64fre.rs2_release.170317-1834
Machine Name:
Debug session time: Fri Jun  9 15:52:04.000 2017 (UTC - 7:00)
System Uptime: not available
Process Uptime: 0 days 0:03:44.000
.....................................................................
This dump file has a breakpoint exception stored in it.
The stored exception information can be accessed via .ecxr.
ntdll!DbgBreakPoint:
00007ff8`aada8d70 cc              int     3
```

然后使用 ||s （设置当前系统） 命令，将当前系统设置为系统 1，然后显示当前系统。

dbgcmd

```
||1:1:017> ||1s
||1:1:017> ||
   0 User mini dump: c:\notepad.dmp
.  1 User mini dump: c:\paint.dmp
   2 User mini dump: c:\calc.dmp
```

查看当前转储文件时，可以使用 .detach 命令。

dbgcmd

```
||1:1:017> .detach
ntdll!DbgBreakPoint:
00007ff8`aada8d70 cc              int     3
Detached
||0:0:007> ||
.  0 User mini dump: c:\notepad.dmp
   2 User mini dump: c:\calc.dmp
```

资源

有关调试的其他信息，请参阅以下资源。

书籍

- 《高级 Windows 调试》，作者：Mario Hewardt 和 Daniel Pravat
- 在 Windows 调试中：Tarik Soulami 在 Windows 中调试和跟踪策略的实用指南

- 由帕维尔·约西夫维奇、亚历克斯·伊恩斯库、马克·鲁西诺维奇和大卫·所罗门的 Windows 内部

视频

碎片整理工具秀 WinDbg 第 13-29 集：</shows/defrag-tools/>

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

跟踪占用大量处理器资源的进程

项目 • 2023/06/15

如果一个应用程序消耗 (“hogging”) 所有处理器的注意力，其他进程最终将“饥饿”并无法运行。

使用以下过程更正此类 bug。

调试使用所有 CPU 周期的应用程序

1. **确定导致此问题的应用程序：** 使用 **任务管理器** 或 **Perfmon** 查找使用处理器周期 99% 或 100% 的进程。 这也可能会告诉你有问题的线程。

2. 将 WinDbg、KD 或 CDB 附加到此过程。

3. **确定导致问题的线程：** 闯入有问题的应用程序。 使用 **！ runaway 3** 扩展获取所有 CPU 时间的“快照”。 使用 **g (Go)** 并等待几秒钟。 然后闯入并再次使用 **！ 失控 3** 。

```
dbgcmd
```

```
0:002> !runaway 3
 User Mode Time
 Thread    Time
 4e0        0:12:16.0312
 268        0:00:00.0000
 22c        0:00:00.0000
 Kernel Mode Time
 Thread    Time
 4e0        0:00:05.0312
 268        0:00:00.0000
 22c        0:00:00.0000

0:002> g

0:001> !runaway 3
 User Mode Time
 Thread    Time
 4e0        0:12:37.0609
 3d4        0:00:00.0000
 22c        0:00:00.0000
 Kernel Mode Time
 Thread    Time
 4e0        0:00:07.0421
 3d4        0:00:00.0000
 22c        0:00:00.0000
```

比较两组数字，查找其用户模式时间或内核模式时间增加最多的线程。 因为 **！runaway** 按 CPU 时间降序排序，因此违规线程通常是列表顶部的线程。 在这种情况下，线程0x4E0导致问题。

4. 使用 **~ (线程状态)** 和 **~s (设置当前线程)** 命令将此线程设为当前线程：

```
dbgcmd
```

```
0:001> ~
   0  Id: 3f4.3d4 Suspend: 1 Teb: 7ffde000 Unfrozen
.  1  Id: 3f4.22c Suspend: 1 Teb: 7ffdd000 Unfrozen
 2  Id: 3f4.4e0 Suspend: 1 Teb: 7ffdc000 Unfrozen

0:001> ~2s
```

5. 使用 **kb (显示堆栈回溯)** 获取此线程的堆栈跟踪：

```
dbgcmd
```

```
0:002> kb
FramePtr  RetAddr   Param1   Param2   Param3   Function Name
0b4ffc74  77f6c600  000000c8.00000000 77fa5ad0
BuggyProgram!CreateMsgFile+0x1b
0b4ffce4  01836060  0184f440 00000001 0b4ffe20
BuggyProgram!OpenDestFileStream+0xb3
0b4ffd20  01843eba  02b5b920 00000102 02b1e0e0
BuggyProgram!SaveMsgToDestFolder+0xb3
0b4ffe20  01855924  0b4ffef0 00145970 0b4ffef0
BuggyProgram!DispatchToConn+0xa4
0b4ffe5c  77e112e6  01843e16 0b4ffef0 0b4fff34
RPCRT4!DispatchToStubInC+0x34
0b4ffeb0  77e11215  0b4ffef0 00000000 0b4fff34 RPCRT4!?
DispatchToStubWorker@RPC_INTERFACE@@AAEJPAU_RPC_MESSAGE@@IPAJ@Z+0xb0
0b4ffed0  77e1a3b1  0b4ffef0 00000000 0b4fff34 RPCRT4!?
DispatchToStub@RPC_INTERFACE@@QAEJPAU_RPC_MESSAGE@Z+0x41
0b4fff40  77e181e4  02b1e0b0 00000074 0b4fff90 RPCRT4!?
ReceiveOriginalCall@OSF_SCONNECTION@Z+0x14b
0b4fff60  77e1a5df  02b1e0b0 00000074 00149210 RPCRT4!?
DispatchPacket@OSF_SCONNECTION@+0x91
0b4fff90  77e1ac1c  77e15eaf 00149210 0b4fffec RPCRT4!?
ReceiveLotsaCalls@OSF_ADDRESS@@QAEXXZ+0x76
```

6. 在当前运行的函数的返回地址上设置断点。 在这种情况下，返回地址在第一行显示为0x77F6C600。 返回地址等效于 BuggyProgram！ 第二行中显示的函数偏移量 (**BuggyProgram！OpenDestFileStream+0xB3**) 。 如果应用程序没有可用的符号，则可能不会显示函数名称。 使用符号或十六进制地址，使用 **g (Go)** 命令执行，直到到达此返回地址：

```
dbgcmd
```

```
0:002> g BuggyProgram!OpenDestFileStream+0xb3
```

7. 如果命中此断点，请重复此过程。 例如，假设命中了此断点。 应执行以下步骤：

dbgcmd

```
0:002> kb
FramePtr  RetAddr   Param1   Param2   Param3   Function Name
0b4ffce4  01836060  0184f440 00000001 0b4ffe20
BuggyProgram!OpenDestFileStream+0xb3
0b4ffd20  01843eba  02b5b920 00000102 02b1e0e0
BuggyProgram!SaveMsgToDestFolder+0xb3
0b4ffe20  01855924  0b4ffef0 00145970 0b4ffef0
BuggyProgram!DispatchToConn+0xa4
0b4ffe5c  77e112e6  01843e16 0b4ffef0 0b4fff34
RPCRT4!DispatchToStubInC+0x34
0b4ffeb0  77e11215  0b4ffef0 00000000 0b4fff34 RPCRT4!?
DispatchToStubWorker@RPC_INTERFACE@@AAEJPAU_RPC_MESSAGE@@IPAJ@Z+0xb0
0b4ffed0  77e1a3b1  0b4ffef0 00000000 0b4fff34 RPCRT4!?
DispatchToStub@RPC_INTERFACE@@QAEJPAU_RPC_MESSAGE@Z+0x41
0b4fff40  77e181e4  02b1e0b0 00000074 0b4fff90 RPCRT4!?
ReceiveOriginalCall@OSF_SCONNECTION@Z+0x14b
0b4fff60  77e1a5df  02b1e0b0 00000074 00149210 RPCRT4!?
DispatchPacket@OSF_SCONNECTION@+0x91
0b4fff90  77e1ac1c  77e15eaf 00149210 0b4fffec RPCRT4!?
ReceiveLotsaCalls@OSF_ADDRESS@@QAEXXZ+0x76

0:002> g BuggyProgram!SaveMsgToDestFolder+0xb3
```

如果命中，请继续：

dbgcmd

```
0:002> kb
FramePtr  RetAddr   Param1   Param2   Param3   Function Name
0b4ffd20  01843eba  02b5b920 00000102 02b1e0e0
BuggyProgram!SaveMsgToDestFolder+0xb3
0b4ffe20  01855924  0b4ffef0 00145970 0b4ffef0
BuggyProgram!DispatchToConn+0xa4
0b4ffe5c  77e112e6  01843e16 0b4ffef0 0b4fff34
RPCRT4!DispatchToStubInC+0x34
0b4ffeb0  77e11215  0b4ffef0 00000000 0b4fff34 RPCRT4!?
DispatchToStubWorker@RPC_INTERFACE@@AAEJPAU_RPC_MESSAGE@@IPAJ@Z+0xb0
0b4ffed0  77e1a3b1  0b4ffef0 00000000 0b4fff34 RPCRT4!?
DispatchToStub@RPC_INTERFACE@@QAEJPAU_RPC_MESSAGE@Z+0x41
0b4fff40  77e181e4  02b1e0b0 00000074 0b4fff90 RPCRT4!?
ReceiveOriginalCall@OSF_SCONNECTION@Z+0x14b
0b4fff60  77e1a5df  02b1e0b0 00000074 00149210 RPCRT4!?
DispatchPacket@OSF_SCONNECTION@+0x91
0b4fff90  77e1ac1c  77e15eaf 00149210 0b4fffec RPCRT4!?
ReceiveLotsaCalls@OSF_ADDRESS@@QAEXXZ+0x76
```

```
0:002> g BuggyProgram!DispatchToConn+0xa4
```

8. 最后，你将找到一个未命中的断点。 在这种情况下，应假定最后一个 **g** 命令设置运行的目标，并且目标未中断。 这意味着 **SaveMsgToDestFolder ()** 函数永远不会返回。

9. 再次闯入线程并在 **BuggyProgram 上设置断点！SaveMsgToDestFolder+0xB3** 与 **bp (Set Breakpoint)** 命令。 然后重复使用 **g** 命令。 如果此断点立即命中，无论你已执行目标多少次，很可能已识别出有问题的函数：

```
dbgcmd
```

```
0:002> bp BuggyProgram!SaveMsgToDestFolder+0xb3

0:002> g

0:002> g
```

10. 使用 **p (Step)** 命令继续执行函数，直到确定指令循环序列所在的位置。 然后，可以分析应用程序的源代码，以确定旋转线程的原因。 原因通常是在 **while**、 **do-while**、 **goto** 或 **for** 循环的逻辑中出现问题。

确定对象的 ACL

项目 • 2024/01/10

可以使用调试器检查对象的 ACL) (访问控制列表。

如果要执行内核调试，则可以使用以下方法。 若要在执行用户模式调试时使用它，需要将控件重定向到内核调试器。 有关详细信息，请参阅从内核调试器控制 User-Mode 调试器 。

首先，使用具有相关对象名称的 **！ object** 调试器扩展：

dbgcmd

```
kd> !object \BaseNamedObjects\AgentToWkssvcEvent
Object: ffbb8a98  Type: (80e30e70) Event
    ObjectHeader: ffbb8a80
    HandleCount: 2  PointerCount: 3
    Directory Object: e14824a0  Name: AgentToWkssvcEvent
```

这表明对象标头具有地址0xFFBB8A80。 将 **dt (Display Type)** 命令用于此地址和 **nt!_OBJECT_HEADER** 结构名称：

dbgcmd

```
kd> dt nt!_OBJECT_HEADER ffbb8a80
   +0x000 PointerCount     : 3
   +0x004 HandleCount      : 2
   +0x004 NextToFree       : 0x00000002
 +0x008 Type             : 0x80e30e70
   +0x00c NameInfoOffset   : 0x10 '
 +0x00d HandleInfoOffset : 0 '
   +0x00e QuotaInfoOffset  : 0 '
   +0x00f Flags            : 0x20 ' '
   +0x010 ObjectCreateInfo : 0x8016b460
   +0x010 QuotaBlockCharged : 0x8016b460
   +0x014 SecurityDescriptor : 0xe11f08b6
   +0x018 Body             : _QUAD
```

安全描述符指针值显示为0xE11F08B6。 此值的最低 3 位表示超出此结构开头的偏移量，因此应忽略它们。 换句话说，SECURITY_DESCRIPTOR结构实际上从0xE11F08B6 &~0x7 开始。 在此地址上使用 **！ sd** 扩展：

dbgcmd

```
kd> !sd e11f08b0
->Revision: 0x1
```

```
->Sbz1    : 0x0
->Control : 0x8004
            SE_DACL_PRESENT
 SE_SELF_RELATIVE
->Owner   : S-1-5-32-544
->Group   : S-1-5-18
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x44
->Dacl    : ->AceCount   : 0x2
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x14
->Dacl    : ->Ace[0]: ->Mask : 0x001f0003
->Dacl    : ->Ace[0]: ->SID: S-1-5-18

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x18
->Dacl    : ->Ace[1]: ->Mask : 0x00120001
->Dacl    : ->Ace[1]: ->SID: S-1-5-32-544

->Sacl    :  is NULL
```

这将显示此对象的安全信息。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

Windows driver documentation 反馈

Windows driver documentation 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

显示关键节

项目 • 2023/10/26

关键部分可以通过各种不同的方法在用户模式下显示。 每个字段的确切含义取决于所使用的 Microsoft Windows 版本。

显示关键部分

关键部分可由 **！ ntsdexts.locks** 扩展、 **！ critsec** 扩展、 **！ cs** 扩展和 **dt (Display Type)** 命令显示。

！ ntsdexts.locks 扩展显示与当前进程关联的关键部分的列表。 如果使用 **-v** 选项，则显示所有关键部分。 以下是示例：

dbgcmd

```
0:000> !locks

CritSec ntdll!FastPebLock+0 at 77FC49E0
LockCount          0
RecursionCount     1
OwningThread       c78
EntryCount         0
ContentionCount    0
*** Locked

....
Scanned 37 critical sections
```

如果知道要显示的关键部分的地址，则可以使用 **！ critsec** 扩展。 这将显示与 **！ ntsdexts.locks** 相同的信息集合。 例如：

dbgcmd

```
0:000> !critsec 77fc49e0

CritSec ntdll!FastPebLock+0 at 77FC49E0
LockCount          0
RecursionCount     1
OwningThread       c78
EntryCount         0
ContentionCount    0
*** Locked
```

！ cs 扩展可以基于其地址显示关键节，搜索关键节的地址范围，甚至显示与每个关键节关联的堆栈跟踪。 其中一些功能需要完整的 Windows 符号才能正常工作。 如果应用程序验证程序处于活动状态，则 ！ cs -t 可用于显示关键节树。 有关详细信息和示例，请参阅 ！ cs 参考页。

！ cs 显示的信息与 ！ ntsdexts.locks 和 ！ critsec 显示的信息略有不同。 例如：

dbgcmd

```
## 0:000> !cs 77fc49e0

Critical section   = 0x77fc49e0 (ntdll!FastPebLock+0x0)
DebugInfo          = 0x77fc3e00
LOCKED
LockCount          = 0x0
OwningThread       = 0x00000c78
RecursionCount     = 0x1
LockSemaphore      = 0x0
SpinCount          = 0x00000000
```

dt (Display Type) 命令可用于显示RTL_CRITICAL_SECTION结构的文本内容。 例如：

dbgcmd

```
0:000> dt RTL_CRITICAL_SECTION 77fc49e0
   +0x000 DebugInfo        : 0x77fc3e00
   +0x004 LockCount        : 0
   +0x008 RecursionCount   : 1
   +0x00c OwningThread     : 0x00000c78
   +0x010 LockSemaphore    : (null)
   +0x014 SpinCount        : 0
```

解释 Windows XP 和 Windows 2000 中的“关键节”字段

关键节结构最重要的字段如下所示：

- 在 Microsoft Windows 2000 和 Windows XP 中， **LockCount** 字段指示任何线程为此关键部分调用 **EnterCriticalSection** 例程的次数减一次。 对于已解锁的关键部分，此字段从 -1 开始。 **每次调用 EnterCriticalSection** 都会递增此值;**每次调用 LeaveCriticalSection** 都会递减它。 例如，如果 **LockCount** 为 5，则锁定此关键部分，一个线程已获取它，另外五个线程正在等待此锁。
- **RecursionCount** 字段指示拥有线程为此关键部分调用 **EnterCriticalSection** 的次数。

- **EntryCount** 字段指示除拥有线程以外的线程为此关键部分调用 **EnterCriticalSection** 的次数。

新初始化的关键部分如下所示：

```
dbgcmd

0:000> !critsec 433e60
CritSec mymodule!cs+0 at 00433E60
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    0
```

调试器将“NOT LOCKED”显示为 **LockCount** 的值。 此字段对于已解锁的关键部分的实际值为 -1。 可以使用 **dt (Display Type)** 命令对此进行验证：

```
dbgcmd

0:000> dt RTL_CRITICAL_SECTION 433e60
   +0x000 DebugInfo        : 0x77fcec80
   +0x004 LockCount        : -1
   +0x008 RecursionCount   : 0
   +0x00c OwningThread     : (null)
   +0x010 LockSemaphore    : (null)
   +0x014 SpinCount        : 0
```

当第一个线程调用 **EnterCriticalSection** 例程时，关键节的 **LockCount**、**RecursionCount**、 **EntryCount** 和 **ContentionCount** 字段全部递增一个， **而 OwningThread** 将成为调用方线程 ID。 **EntryCount** 和 **ContentionCount** 永远不会递减。 例如：

```
dbgcmd

0:000> !critsec 433e60
CritSec mymodule!cs+0 at 00433E60
LockCount          0
RecursionCount     1
OwningThread       4d0
EntryCount         0
ContentionCount    0
```

此时，可能会发生四个不同的事情。

1. 拥有线程再次调用 **EnterCriticalSection** 。 这会递增 **LockCount** 和 **RecursionCount**。 **EntryCount** 不递增。

```
dbgcmd

0:000> !critsec 433e60
CritSec mymodule!cs+0 at 00433E60
LockCount          1
RecursionCount     2
OwningThread       4d0
EntryCount         0
ContentionCount    0
```

2. 另一个线程调用 **EnterCriticalSection**。 这会递增 **LockCount** 和 **EntryCount**。 **RecursionCount** 不递增。

```
dbgcmd

0:000> !critsec 433e60
CritSec mymodule!cs+0 at 00433E60
LockCount          1
RecursionCount     1
OwningThread       4d0
EntryCount         1
ContentionCount    1
```

3. 拥有的线程调用 **LeaveCriticalSection**。 这会将 **LockCount** (减为 -1)， **将 RecursionCount** (减为 0)，并将 **OwningThread** 重置为 0。

```
dbgcmd

0:000> !critsec 433e60
CritSec mymodule!cs+0 at 00433E60
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    0
```

4. 另一个线程调用 **LeaveCriticalSection**。 这会生成与调用 **LeaveCriticalSection** 的线程相同的结果-它将 **LockCount** (减为 -1)， recursionCount (减为 0)，并将 **OwningThread** 重置为 0。

当任何线程调用 **LeaveCriticalSection** 时，Windows 会递减 **LockCount** 和 **RecursionCount**。 此功能具有好方面和坏方面。 它允许设备驱动程序在一个线程上进入关键部分，并将关键部分保留在另一个线程上。 但是，它也有可能在错误的线程上意外调用 **LeaveCriticalSection** ，或者调用 **LeaveCriticalSection** 太多次，导致 **LockCount** 达到低于 -1 的值。 这会损坏关键部分，并导致所有线程无限期地等待关键部分。

解释 Windows Server 2003 SP1 及更高版本中的关键节字段

在 Microsoft Windows Server 2003 Service Pack 1 及更高版本的 Windows 中，**LockCount** 字段分析如下：

- 最低位显示锁定状态。 如果此位为 0，则锁定关键部分;如果为 1，则不锁定关键部分。
- 下一位显示是否已为此锁唤醒线程。 如果此位为 0，则表示已为此锁唤醒线程;如果为 1，则表示尚未唤醒任何线程。
- 剩余的位是等待锁的线程数的补码。

例如，假设 **LockCount** 为 -22。 可按以下方式确定最低位：

```
dbgcmd

0:009> ? 0x1 & (-0n22)
Evaluate expression: 0 = 00000000
```

可按以下方式确定下一个最低位：

```
dbgcmd

0:009> ? (0x2 & (-0n22)) >> 1
Evaluate expression: 1 = 00000001
```

可以按以下方式确定剩余位的补数：

```
dbgcmd

0:009> ? ((-1) - (-0n22)) >> 2
Evaluate expression: 5 = 00000005
```

在此示例中，第一个位为 0，因此关键部分被锁定。 第二个位为 1，因此尚未为此锁唤醒任何线程。 剩余位的补数为 5，因此有 5 个线程等待此锁。

其他信息

有关如何调试关键节超时的信息，请参阅 关键节超时。 有关关键部分的一般信息，请参阅 Microsoft Windows SDK、Windows 驱动程序工具包 (WDK) 或 Microsoft *Windows Internals*（由 Mark Russinovich 和 David Solomon 撰写）。

调试死锁

项目 • 2023/10/26

当线程需要对代码或其他资源进行独占访问时，它会请求 *锁定*。 如果可以，Windows 会通过将此锁分配给线程来做出响应。 此时，系统中的任何其他内容都无法访问锁定的代码。 这种情况一直发生，并且是任何编写良好的多线程应用程序的正常部分。 尽管特定代码段一次只能有一个锁，但多个代码段可以各自有自己的锁。

当两个或多个线程以不兼容的顺序请求对两个或多个资源的锁定时，会出现 *死锁* 。 例如，假设线程 1 已获取资源 A 上的锁，然后请求访问资源 B。同时，线程二已获取资源 B 上的锁，然后请求访问资源 A。在放弃其他线程的锁之前，两个线程都无法继续，因此，两个线程都无法继续。

当多个线程（通常是单个应用程序）阻止彼此访问同一资源时，会出现用户模式死锁。 但是，多个应用程序的多个线程也可能阻止彼此访问全局/共享资源，例如全局事件或信号量。

当多个线程从同一进程或不同进程（，）阻止彼此访问同一内核资源时，就会发生内核模式死锁。

用于调试死锁的过程取决于死锁是在用户模式还是在内核模式下发生。

调试 User-Mode 死锁

在用户模式下发生死锁时，请使用以下过程对其进行调试：

1. 发出 **！ ntsdexts.locks** 扩展。 在用户模式下，只需在调试器提示符处键入 **！ locks**；假定为 **ntsdexts** 前缀。
2. 此扩展显示与当前进程关联的所有关键部分，以及拥有线程的 ID 和每个关键节的锁计数。 如果关键节的锁计数为零，则它不会锁定。 使用 **~ (线程状态)** 命令查看有关拥有其他关键部分的线程的信息。
3. 为每个线程使用 **kb (显示堆栈回溯)** 命令来确定它们是否正在等待其他关键部分。
4. 使用这些 **kb** 命令的输出，可以找到死锁：两个线程分别等待另一个线程持有的锁。 在极少数情况下，死锁可能是由两个以上的线程以循环模式持有锁引起的，但大多数死锁只涉及两个线程。

下面是此过程的插图。 从 **！ ntdexts.locks** 扩展开始：

```
dbgcmd
```

```
0:006>  !locks
CritSec ftpsvc2!g_csServiceEntryLock+0 at 6833dd68
LockCount          0
RecursionCount     1
OwningThread       a7
EntryCount         0
ContentionCount    0
*** Locked

CritSec isatq!AtqActiveContextList+a8 at 68629100
LockCount          2
RecursionCount     1
OwningThread       a3
EntryCount         2
ContentionCount    2
*** Locked

CritSec +24e750 at 24e750
LockCount          6
RecursionCount     1
OwningThread       a9
EntryCount         6
ContentionCount    6
*** Locked
```

显示的第一个关键部分没有锁，因此可以忽略。

显示的第二个关键部分的锁计数为 2，因此可能是死锁的原因。 拥有线程的线程 ID 为 0xA3。

可以通过列出具有 **~ (线程状态)** 命令的所有线程并查找具有此 ID 的线程来查找此线程：

dbgcmd

```
0:006>  ~
   0  Id: 1364.1330 Suspend: 1 Teb: 7ffdf000 Unfrozen
   1  Id: 1364.17e0 Suspend: 1 Teb: 7ffde000 Unfrozen
   2  Id: 1364.135c Suspend: 1 Teb: 7ffdd000 Unfrozen
   3  Id: 1364.1790 Suspend: 1 Teb: 7ffdc000 Unfrozen
   4  Id: 1364.a3 Suspend: 1 Teb: 7ffdb000 Unfrozen
   5  Id: 1364.1278 Suspend: 1 Teb: 7ffda000 Unfrozen
.  6  Id: 1364.a9 Suspend: 1 Teb: 7ffd9000 Unfrozen
   7  Id: 1364.111c Suspend: 1 Teb: 7ffd8000 Unfrozen
   8  Id: 1364.1588 Suspend: 1 Teb: 7ffd7000 Unfrozen
```

在此显示中，第一项是调试器的内部线程号。 字段 (`Id` 的第二项) 包含两个用小数点分隔的十六进制数。 小数点前的数字是进程 ID;小数点后面的数字是线程 ID。 在此示例中，可以看到线程 ID 0xA3对应于线程编号 4。

然后，**使用 kb (Display Stack Backtrace)** 命令显示对应于线程编号 4 的堆栈：

dbgcmd

```
0:006>  ~4 kb
  4  id: 97.a3   Suspend: 0 Teb 7ffd9000 Unfrozen
ChildEBP RetAddr  Args to Child
014cfe64 77f6cc7b 00000460 00000000 00000000 ntdll!NtWaitForSingleObject+0xb
014cfed8 77f67456 0024e750 6833adb8 0024e750
ntdll!RtlpWaitForCriticalSection+0xaa
014cfee0 6833adb8 0024e750 80000000 01f21cb8
ntdll!RtlEnterCriticalSection+0x46
014cfef4 6833ad8f 01f21cb8 000a41f0 014cff20
ftpsvc2!DereferenceUserDataAndKill+0x24
014cff04 6833324a 01f21cb8 00000000 00000079
ftpsvc2!ProcessUserAsyncIoCompletion+0x2a
014cff20 68627260 01f21e0c 00000000 00000079
ftpsvc2!ProcessAtqCompletion+0x32
014cff40 686249a5 000a41f0 00000001 686290e8 isatq!I_TimeOutContext+0x87
014cff5c 68621ea7 00000000 00000001 0000001e
isatq!AtqProcessTimeoutOfRequests_33+0x4f
014cff70 68621e66 68629148 000ad1b8 686230c0 isatq!I_AtqTimeOutWorker+0x30
014cff7c 686230c0 00000000 00000001 000c000a
isatq!I_AtqTimeoutCompletion+0x38
014cffb8 77f04f2c 00000000 00000001 000c000a isatq!SchedulerThread_297+0x2f
00000001 000003e6 00000000 00000001 000c000a kernel32!BaseThreadStart+0x51
```

请注意，此线程具有对 WaitForCriticalSection 函数的调用，这意味着它不仅具有锁，还在等待被其他内容锁定的代码。 通过查看 **对** WaitForCriticalSection 的调用的第一个参数，我们可以找出正在等待的关键部分。 这是 **Args to Child** 下的第一个地址：“24e750”。 因此，此线程正在等待地址0x24E750的关键部分。 这是前面使用的 **！ locks** 扩展列出的第三个关键部分。

换句话说，拥有第二个关键节的线程 4 正在等待第三个关键节。 现在，请将注意力转向第三个关键部分，该部分也已锁定。 拥有线程具有线程 ID 0xA9。 返回到之前看到的命令的 ~ 输出，请注意，具有此 ID 的线程是线程编号 6。 显示此线程的堆栈回溯：

dbgcmd

```
0:006>  ~6 kb
ChildEBP RetAddr  Args to Child
0155fe38 77f6cc7b 00000414 00000000 00000000 ntdll!NtWaitForSingleObject+0xb
0155feac 77f67456 68629100 6862142e 68629100
ntdll!RtlpWaitForCriticalSection+0xaa
0155feb4 6862142e 68629100 0009f238 686222e1
ntdll!RtlEnterCriticalSection+0x46
0155fec0 686222e1 0009f25c 00000001 0009f238
isatq!ATQ_CONTEXT_LISTHEAD__RemoveFromList
0155fed0 68621412 0009f238 686213d1 0009f238
isatq!ATQ_CONTEXT__CleanupAndRelease+0x30
```

```
0155fed8 686213d1 0009f238 00000001 01f26bcc
isatq!AtqpReuseOrFreeContext+0x3f
0155fee8 683331f7 0009f238 00000001 01f26bf0 isatq!AtqFreeContext+0x36
0155fefc 6833984b ffffffff 00000000 00000000
ftpsvc2!ASYNC_IO_CONNECTION__SetNewSocket
0155ff18 6833adcd 77f05154 01f26a58 00000000 ftpsvc2!USER_DATA__Cleanup+0x47
0155ff28 6833ad8f 01f26a58 000a3410 0155ff54
ftpsvc2!DereferenceUserDataAndKill+0x39
0155ff38 6833324a 01f26a58 00000000 00000040
ftpsvc2!ProcessUserAsyncIoCompletion+0x2a
0155ff54 686211eb 01f26bac 00000000 00000040
ftpsvc2!ProcessAtqCompletion+0x32
0155ff88 68622676 000a3464 00000000 000a3414 isatq!AtqpProcessContext+0xa7
0155ffb8 77f04f2c abcdef01 ffffffff 000ad1b0 isatq!AtqPoolThread+0x32
0155ffec 00000000 68622644 abcdef01 00000000 kernel32!BaseThreadStart+0x51
```

此线程也正在等待释放关键部分。 在这种情况下，它正在等待0x68629100的关键部分。 这是 **前面由！ locks** 扩展生成的列表中的第二个关键部分。

这是死锁。 拥有第二个关键节的线程 4 正在等待第三个关键节。 拥有第三个关键节的线程 6 正在等待第二个关键节。

确认此死锁的性质后，可以使用常用的调试技术来分析线程 4 和 6。

调试 Kernel-Mode 死锁

有几个调试器扩展可用于在内核模式下调试死锁：

- **！ kdexts.locks** 扩展显示有关内核资源上保留的所有锁以及持有这些锁的线程的信息。 （在内核模式下，只需在调试器提示符下键入 **！ locks** ;假定 **kdexts** 前缀。）
- **！ qlocks** 扩展显示所有排队的旋转锁的状态。
- **！ wdfkd.wdfspinlock** 扩展显示有关 Kernel-Mode Driver Framework (KMDF) spin-lock 对象的信息。
- **！ 死锁**扩展与驱动程序验证程序结合使用，以检测代码中可能导致死锁的不一致使用锁。

在内核模式下发生死锁时，请使用 **！ kdexts.locks** 扩展列出线程当前获取的所有锁。

通常，可以通过查找一个未执行的线程来查明死锁，该线程持有执行线程所需的资源上的独占锁。 大多数锁都是共享的。

调试失败的驱动程序卸载

项目 • 2023/06/15

如果存在对 DeviceObject **或** DriverObject 的引用泄漏， **驱动程序**将不会卸载。 这是驱动程序卸载失败的常见原因。

除了 IoCreateDevice，还有几个函数引用 DriverObject 和 DeviceObject。 如果不遵循使用函数的准则，最终会泄漏引用。

下面是如何调试此问题的示例。 尽管本示例使用 DeviceObject ，但此方法适用于所有对象。

修复无法卸载的驱动程序

1. 在驱动程序调用 IoCreateDevice 后立即放置断点。 获取 DeviceObject 地址。
2. 使用此对象地址上的 **！ object** 扩展查找对象标头：

   dbgcmd

   ```
   kd> !object 81a578c0
   Object: 81a578c0  Type: (81bd0e70) Device
       ObjectHeader: 81a578a8
       HandleCount: 0  PointerCount: 3
       Directory Object: e1001208  Name: Serial0
   ```

   ObjectHeader 中的第一个变量是*指针计数*或*引用计数*。
3. 使用 ObjectHeader 的地址在指针计数上放置写入断点：

   dbgcmd

   ```
   kd> ba w4 81a578a8 "k;g"
   ```
4. 使用 **g (Go)** 。 调试器将生成日志。
5. 查找不匹配的引用/取消引用对 -- 具体而言，是缺少的取消引用。 (请注意，ObReferenceObject 在 kernel.)

读取 Bug 检查回调数据

项目 • 2023/10/26

许多驱动程序*检查回调例程提供 bug*。 当 Windows 检查发出 bug 时，它会在关闭系统之前调用这些例程。 这些例程可以指定并写入称为*回调数据和辅助回调数据的*内存区域。

BugCheckCallback 使用 KBUGCHECK_CALLBACK_ROUTINE
此例程写入的数据将成为 *回调数据的*一部分。 故障转储文件不包含数据。

BugCheckSecondaryDumpDataCallback 使用 KBUGCHECK_REASON_CALLBACK_ROUTINE
此例程写入的数据将成为 *辅助回调数据的*一部分。 数据包含在故障转储文件中。

BugCheckAddPagesCallback 使用 KBUGCHECK_REASON_CALLBACK_ROUTINE
此例程指定的页将成为 *回调数据的*一部分。 这些页中的数据包含在故障转储文件中。

调试器可用的回调和辅助回调数据量取决于几个因素：

- 如果要对崩溃的系统执行实时调试，则已由 BugCheckCallback 写入或由 BugCheckAddPagesCallback 指定的回调数据将可用。 辅助回调数据将不可用，因为它未存储在任何固定的内存位置。
- 如果要调试完整内存转储或内核内存转储， 则 BugCheckAddPagesCallback 指定的回调数据以及 BugCheckSecondaryDumpDataCallback 写入的 辅助回调数据将可用。 BugCheckCallback 写入的回调数据将不可用。
- 如果要调试小型内存转储，则回调数据将不可用。 辅助回调数据将可用。

有关这些不同 转储文件大小的 更多详细信息，请参阅 Kernel-Mode 转储文件的品种。

显示回调数据

若要显示 bug 检查回调数据，可以使用 **！bugdump** 扩展。

没有任何参数， **！bugdump** 将显示所有回调的数据。

若要查看一个特定回调例程的数据，请使用 **！bugdump***组件*，其中 *Component* 是注册该例程时传递给 **KeRegisterBugCheckCallback** 的相同参数。

显示辅助回调数据

有两种方法可用于显示辅助回调数据。 可以使用 **.enumtag** 命令，也可以编写自己的调试器扩展。

辅助回调数据的每个块都由 GUID 标记标识。 此标记由传递给 BugCheckSecondaryDumpDataCallback **的 (KBUGCHECK_SECONDARY_DUMP_DATA) ReasonSpecificData** 参数的 **Guid** 字段指定。

.enumtag (枚举辅助回调数据) 命令不是非常精确的工具。 它显示每个辅助数据块，显示标记，然后以十六进制和 ASCII 格式显示数据。 通常，它只用于确定哪些标记实际用于辅助数据块。

若要以更实用的方式使用此数据，建议编写自己的调试器扩展。 此扩展必须调用 dbgeng.h 头文件中的方法。 有关详细信息，请参阅 编写新的调试器扩展。

如果知道辅助数据块的 GUID 标记，扩展应使用 **IDebugDataSpaces3：：ReadTagged** 方法来访问数据。 其原型如下所示：

```
C++

STDMETHOD(ReadTagged)(
    THIS_
    IN LPGUID Tag,
    IN ULONG Offset,
    OUT OPTIONAL PVOID Buffer,
    IN ULONG BufferSize,
    OUT OPTIONAL PULONG TotalSize
    ) PURE;
```

下面是如何使用此方法的示例：

```
C++

UCHAR RawData[MY_DATA_SIZE];
GUID MyGuid = .... ;

Success = DataSpaces->ReadTagged(  &MyGuid,  0,  RawData,
                                   sizeof(RawData),  NULL);
```

如果提供的 *BufferSize* 太小， **ReadTagged** 将成功，但只会将请求的字节数写入 *Buffer*。 如果指定的 *BufferSize* 太大， **ReadTagged** 将成功，但只会将实际块大小写入 *Buffer*。 如果为 *TotalSize* 提供指针， **ReadTagged** 将使用该指针返回实际块的大小。 如果无法访问块， **ReadTagged** 将返回失败状态代码。

如果两个块具有相同的 GUID 标记，将返回第一个匹配块，第二个块将无法访问。

如果不确定块的 GUID 标记，可以使用 **IDebugDataSpaces3：：StartEnumTagged**、**IDebugDataSpaces3：：GetNextTagged** 和 **IDebugDataSpaces3：：EndEnumTagged** 方法来枚举标记的块。 其原型如下所示：

```
C++
```

```
STDMETHOD(StartEnumTagged)(
    THIS_
    OUT PULONG64 Handle
    ) PURE;

STDMETHOD(GetNextTagged)(
    THIS_
    IN ULONG64 Handle,
    OUT LPGUID Tag,
    OUT PULONG Size
    ) PURE;

STDMETHOD(EndEnumTagged)(
    THIS_
    IN ULONG64 Handle
    ) PURE;
```

调试回调例程

还可以调试回调例程本身。 回调例程中的断点的工作方式与任何其他断点一样。

如果回调例程导致第二个 bug 检查，则将首先处理此新的 bug 检查。 但是，Windows 不会重复停止进程的某些部分，例如，它不会写入第二个故障转储文件。 蓝屏上显示的停止代码将是代码检查第二个 bug。 如果附加了内核调试器，通常会显示有关这两个 bug 检查的消息。

使用 KD 调试用户模式故障

项目 • 2023/10/26

若要正确调试用户模式故障，需要 CDB 或 WinDbg。 但是，有时用户模式异常会中断 KD，因为不存在用户模式调试器。 有时，在调试内核模式问题时，监视特定用户模式进程正在执行的操作也很有帮助。

默认情况下，内核调试器尝试加载与 **k**、 **u** 或 **ln** 命令 (指定地址匹配的第一个用户模式符号) 。

遗憾的是，用户模式符号通常未在符号路径中指定，或者第一个符号不是正确的符号。 如果符号不存在，请将它们复制到符号路径，或使用 **.sympath (设置符号路径)** 命令指向完整的符号树，然后使用 **.reload (Reload Module)** 命令。 如果加载了错误的符号，可以通过执行 **.reload <binary.ext>** 来显式加载符号。

大多数 Windows DLL 已重定基，因此它们在不同的地址加载，但也有例外。 视频适配器是最常见的异常。 有数十个视频适配器都在同一基址加载，因此 KD 几乎总是会找到 ati.dll (第一个视频符号，按字母顺序) 。 对于视频，还加载了一个 .sys 文件，可以使用 **lm** 命令进行标识。 有了这些信息，你可以发出 **.reload** 来获取正确的视频 DLL。 有时调试器也会变得混乱，重新加载特定符号将有助于提供正确的堆栈。 取消组合函数以查看符号是否正确。

与视频 DLL 类似，几乎所有可执行文件都在同一地址加载，因此 KD 将报告访问权限。 如果在访问中看到堆栈跟踪，请执行 **！ 进程** ，然后执行给定的可执行文件名称的 **.reload** 。 如果可执行文件在符号路径中没有符号，请将其复制到该处，然后再次执行 **.reload** 。

有时，即使完整的符号树位于符号路径中，KD 或 WinDbg 也难以加载正确的用户模式符号。 在这种情况下，ntdll.dll 和 kernel32.dll 是所需的两个最常见的符号。 从 KD 调试 CSRSS 时，winsrv.dll 和 csrsrv.dll 也是要加载的常见 DLL。

使目标计算机崩溃和重新启动目标计算机

项目 • 2024/07/08

执行内核调试时，可以通过发出 .crash（**强制系统崩溃**）命令，导致目标计算机停止响应（即*崩溃*或 *bug 检查*）。 此命令立即导致目标计算机停止响应。 如果已启用故障转储，调试器将写入内核模式转储文件。 （有关这些文件的详细信息，请参阅 创建内核模式转储文件。

若要重启目标计算机，请使用 .reboot （**重新启动目标计算机**） 命令。

如果希望目标计算机创建故障转储文件，然后重新启动，则应发出 .crash **命令，然后发出** .reboot 命令。 如果只想重启， **则不需要** .crash 命令。

在启动过程的早期阶段，主机与目标计算机之间的连接将丢失。 调试器不提供有关目标计算机的信息。

连接断开后，调试器会关闭所有符号文件并卸载所有调试器扩展。 此时，如果运行的是 KD 或 CDB，则所有断点都将丢失。 在 WinDbg 中，可以保存当前工作区。 此操作保存所有断点。

如果此时想要结束调试会话，请使用 Ctrl+B 命令（在 KD 中），或在“文件”**菜单上单击**“**退出**”（在 WinDbg 中）。

如果未退出调试器，则完成足够的启动过程后，连接将重新建立。 此时会重新加载符号和扩展。 如果运行的是 WinDbg，则会重新加载内核模式工作区。

可以指示调试器在重启过程中在两次可能的时间自动闯入目标计算机：

- 将第一个内核模块加载到内存中时
- 内核初始化时

若要在第一个内核模块加载时设置自动断点，请使用 **-d** 命令行选项。

还可以在调试器运行后更改中断状态：

- 控制初始模块加载和内核初始化断点，如所有异常和事件。 当发生这些事件或忽略这些事件时，可以进入调试器。 还可以在命中这些断点时自动执行指定的命令。 有关详细信息，请参阅控制异常和事件。
- 在 **KD 中使用** Ctrl+K 快捷键、**WinDbg 中的** Ctrl+Alt+K 快捷键和**调试 |内核连接 |WinDbg 中的**“**循环初始中断**”命令可更改中断状态。 每次使用这些命令时，调试器都会切换三种状态：没有自动中断、内核初始化时中断，在第一个内核模块加载时中断。 此方法不能同时激活两个自动断点。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

映射驱动程序文件

项目 • 2024/07/19

替换驱动程序文件可能很困难。 通常，必须引导到 Microsoft Windows *安全内部版本*，替换驱动程序二进制文件，然后再次引导。

使用映射文件时存在替代方法。 你可以使用此映射方法来替换任何内核模式驱动程序（包括显示驱动程序）、任何 Windows 子系统驱动程序或任何其他内核模式模块。 为简单起见，本主题中将这些文件称为*驱动程序*，即使你可以将此方法用于任何内核模式模块也不例外。

每当将 WinDbg 或 KD 附加为内核调试器时，你都可以使用此方法。 你还可以对引导驱动程序使用此方法，但难度更大。 有关如何将此方法用于引导驱动程序的详细信息，请参阅“替换引导驱动程序”。

若要使用驱动程序替换映射来替换驱动程序文件，请执行以下操作：

1. 创建*驱动程序替换映射文件*。 此文件是一个文本文件，其中列出了目标计算机上的驱动程序及其主机上的替换驱动程序。 你可以替换任意数量的驱动程序。 例如，你可以在主机的 d:\Map_Files 目录中创建一个名为 Mymap.ini 的文件，其中包含以下信息。

   ```text
   map
   \Systemroot\system32\drivers\videoprt.sys
   \\myserver\myshare\new_drivers\videoprt.sys
   ```

   有关此文件的语法的详细信息，请参阅驱动程序替换映射文件格式。

2. 设置与目标计算机的内核调试连接，并在主机上启动内核调试器（KD 或 WinDbg）。 （您不必实际中断目标计算机。）

3. 通过执行以下其中一项操作来加载驱动程序替换映射文件：

   - 在启动内核调试器之前设置 _NT_KD_FILES 环境变量。

     ```
     D:\Debugging Tools for Windows> set
     _NT_KD_FILES=d:\Map_Files\mymap.ini
     D:\Debugging Tools for Windows> kd
     ```

- 启动内核调试器后，使用 **.kdfiles（设置驱动程序替换映射）**命令。

  控制台

  ```
  D:\Debugging Tools for Windows> kd
  kd> .kdfiles d:\Map_Files\mymap.ini
  KD file associations loaded from 'd:\Map_Files\mymap.ini'
  ```

  你还可以使用 **.kdfiles** 命令来显示当前驱动程序替换映射文件或删除驱动程序替换映射。 如果不使用此命令，则映射将一直保留，直到退出调试器为止。

完成此过程后，驱动程序替换映射将生效。

每当目标计算机即将加载驱动程序时，它总会查询内核调试器以确定是否已映射此驱动程序。 如果已映射此驱动程序，则会通过内核连接发送替换文件，并将此文件复制到旧驱动程序文件。 然后加载新驱动程序。

驱动程序替换映射文件格式

驱动程序替换映射文件中的三个行表示每个驱动程序文件替换。

- 第一行包括单词“map”。
- 第二行指定目标计算机上的旧驱动程序的路径和文件名。
- 第三行指定新驱动程序的完整路径。 此驱动程序可以位于主机或某个其他服务器上。

你可以任意多次重复此信息模式。

路径和文件名不区分大小写，并且实际驱动程序文件名可能不同。 目标计算机即将加载该驱动程序时，第三行上指定的文件将复制到第二行中指定的文件。

Kdfiles 将尝试匹配服务控制管理器 (SCM) 数据库中存储的文件名。 SCM 数据库中的名称与传递给 MmLoadSystemImage 的名称完全相同。

在 Windows 10 及更高版本的调试工具中，驱动程序映射适用于动态匹配驱动程序名称并确定正确的路径。 不需要指定完整路径，文件扩展名是可选项。 你可以使用这些条目中的任何一个来匹配 NT 文件系统驱动程序。

- ntfs
- NTFS
- ntfs.sys
- windows\system32\drivers\ntfs.sys

你可以使用这些条目中的任何一个来匹配 NT 内核驱动程序。

- ntoskrnl
- NTOSKRNL
- ntoskrnl.sys
- windows\system32\drivers\ntoskrnl.sys

映射文件可以包含空白行，并且可以包含以数字符号开头的注释行 (#)。 但是，在文件中出现“map”后，接下来的两行必须是旧驱动程序和新驱动程序。 空行和注释行无法分解三行映射图块。

以下示例显示了驱动程序替换映射文件。

text

```
# Use the # for comments like this one
map
\Systemroot\system32\drivers\videoprt.sys
e:\MyNewDriver\binaries\videoprt.sys
map
\Systemroot\system32\mydriver.sys
\\myserver\myshare\new_drivers\mydriver0031.sys

# This is replacing a beep driver
map
\??\c:\windows\system32\beep.sys
\\myserver\myshare\new_drivers\new_beep.sys
```

驱动程序替换映射文件必须是文本文件，但可以使用任何文件名和文件扩展名（.ini、.txt、.map 等）。

其他说明

发生驱动程序替换时，内核调试器中会显示一条消息。

如果使用 **CTRL+D**（在 KD 中）或 Ctrl+Alt+D（在 WinDbg 中），则你会看到有关替换请求的详细信息。 如果不确定列出的名称是否与 SCM 数据库中的名称匹配，则此信息可能非常有用。

你可以启用 bcdedit bootdebug 选项来查看早期引导信息，这些信息可用于替换内核、hal 或引导驱动程序。

控制台

```
bcdedit -bootdebug on
```

有关更多信息，请参见 BCDEdit 选项参考。

如果内核调试器退出，则不再发生驱动程序替换。 但是，已替换的任何驱动程序都不会还原成其旧的二进制文件，因为实际上覆盖了驱动程序文件。

此驱动程序替换功能会自动绕过 Windows 文件保护 (WFP)。

你无需重启目标计算机。 无论目标计算机是否已重启，目标计算机加载驱动程序时，都会发生驱动程序替换。 当然，大多数驱动程序都在引导过程中加载，因此实际上，你应在加载映射文件后重启目标计算机。

如果定义了 _NT_KD_FILES 变量，则会在启动内核调试器时读取指定的驱动程序替换映射文件。 如果发出 **.kdfiles** 命令，则会立即读取指定的文件。 此时，调试器会验证文件是否具有基本映射/行/行格式。 但是，在替换发生之前，不会验证实际路径和文件名。

读取映射文件后，调试器会存储其内容。 如果在此时后更改此文件，则更改不起作用（除非重新发出 **.kdfiles** 命令）。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

来自目标的消息

项目 • 2023/06/15

目标应用程序或目标计算机可以使用各种例程，有条件地或无条件地将消息发送到调试器或中断调试器。 内核模式目标还可以测试当前是否附加了调试器。

本节包括：

- 突入调试器
- 将输出发送到调试器
- 读取和筛选调试消息
- 确定是否已附加调试器

突入调试器

项目 • 2024/03/01

用户模式和内核模式代码使用不同的例程进入调试器。

用户模式中断例程

中断例程会导致当前进程中发生异常，以便调用线程可以向与调用进程关联的调试器发出信号。

若要从用户模式程序进入调试器，请使用 DebugBreak 函数。 其原型如下：

```
C++

VOID DebugBreak(VOID);
```

当用户模式程序调用 **DebugBreak** 时，将发生以下可能的操作：

1. 如果附加了用户模式调试器，程序将进入调试器。 这意味着程序将暂停，调试器将激活。
2. 如果未附加用户模式调试器，但在启动时启用了内核模式调试，则整个计算机将进入内核调试器。 如果未附加内核调试器，计算机将冻结并等待内核调试器。
3. 如果未附加用户模式调试器，且未启用内核模式调试，程序将终止并出现未经处理的异常，事后分析（实时）调试器将会激活。 有关详细信息，请参阅启用调事后调试。

内核模式中断例程

当内核模式程序进入调试器时，整个操作系统将冻结，直到内核调试器允许执行恢复为止。 如果不存在内核调试器，则会将其视为 Bug 检查。

DbgBreakPoint 例程在内核模式代码中工作，但与其他 **DebugBreak**用户模式例程类似。

DbgBreakPointWithStatus 例程也会导致中断，但它还会向调试器发送 32 位状态代码。

在检查生成环境中编译时，KdBreakPoint 和 KdBreakPointWithStatus 例程分别与 **DbgBreakPoint** 和 **DbgBreakPointWithStatus** 相同。 在免费生成环境中编译时，它们不起作用。

内核模式条件中断例程

两个条件中断例程可用于内核模式代码。 这些例程测试逻辑表达式。 如果表达式为 false，则执行将停止，调试器变为活动状态。

- ASSERT 宏测试逻辑表达式。 如果表达式为 false，则执行将停止，调试器变为活动状态。 调试器中会显示失败的表达式及其在程序中的位置。
- ASSERTMSG 宏与 ASSERT 相同，只不过它允许向调试器发送其他消息。

ASSERT 和 ASSERTMSG 仅在检查生成环境中编译时处于活动状态。 在免费生成环境中编译时，它们不起作用。

将输出发送到调试器

项目 • 2023/06/15

用户模式和内核模式代码使用不同的例程将输出发送到调试器。

User-Mode输出例程

OutputDebugString 例程将以 null 结尾的字符串发送到调用进程的调试器。 在用户模式驱动程序中， **OutputDebugString** 在“调试器命令”窗口中显示字符串。 如果调试器未运行，则此例程不起作用。 **OutputDebugString** 不支持 **printf** 格式字符串的变量参数。

此例程的原型如下所示：

```
C++

VOID OutputDebugString(
   LPCTSTR lpOutputString
   );
```

有关此例程的完整文档，请参阅 与调试器通信。

Kernel-Mode输出例程

DbgPrint 例程在调试器窗口中显示输出。 此例程支持基本的 **printf** 格式参数。 只有内核模式驱动程序才能调用 **DbgPrint**。

DbgPrintEx 例程类似于 **DbgPrint**，但它允许你“标记”消息。 运行调试器时，只能允许发送具有特定标记的消息。 这样，你只能查看你感兴趣的那些消息。 有关详细信息，请参阅 读取和筛选调试消息。

在检查的生成环境中编译时， KdPrint 和 KdPrintEx 宏分别与 **DbgPrint** 和 **DbgPrintEx** 相同。 在自由生成环境中编译时，它们不起作用。

读取和筛选调试消息

项目 • 2023/06/15

DbgPrintEx、vDbgPrintEx、vDbgPrintExWithPrefix 和 KdPrintEx 例程在指定的条件下向内核调试器发送消息。 此过程使你能够筛选出低优先级消息。

> **备注**
>
> 在 Microsoft Windows Server 2003 和早期版本的 Windows 中， DbgPrint 和 KdPrint 例程无条件地向内核调试器发送消息。 在 Windows Vista 和更高版本的 Windows 中，这些例程按条件发送消息，如 **DbgPrintEx** 和 **KdPrintEx**。 无论使用哪个版本的 Windows，都应使用 **DbgPrintEx**、 **vDbgPrintEx**、**vDbgPrintExWithPrefix** 和 **KdPrintEx**，因为这些例程使你能够控制发送消息的条件。

筛选调试消息

1. 对于要发送到调试器的每条消息，请在驱动程序代码中使用 **DbgPrintEx**、**vDbgPrintEx**、 **vDbgPrintExWithPrefix** 或 **KdPrintEx** 。 将相应的组件名称传递给 *ComponentId* 参数，并将值传递给反映此消息的严重性或性质的 *Level* 参数。 消息本身通过使用与 **printf** 相同的语法传递给 *Format* 和 *arguments* 参数。

2. 设置相应 *组件筛选器掩码*的值。 每个组件都有不同的掩码。 掩码值指示显示该组件的消息中的哪一个。 可以使用注册表编辑器或在内存中使用内核调试器在注册表中设置组件筛选器掩码。

3. 将内核调试器附加到计算机。 每次驱动程序将消息传递给 **DbgPrintEx**、**vDbgPrintEx**、 **vDbgPrintExWithPrefix** 或 **KdPrintEx** 时，传递给 *ComponentId* 和 *Level* 的值都会与相应的组件筛选器掩码的值进行比较。 如果这些值满足特定条件，则会将消息发送到内核调试器并显示。 否则，不会发送任何消息。

> **备注**
>
> 此页上对 **DbgPrintEx** 的所有引用同样适用于 **KdPrintEx**、 **vDbgPrintEx** 和 **vDbgPrintExWithPrefix**。

标识组件名称

每个组件都有一个单独的筛选器掩码。 这允许调试器单独为每个组件配置筛选器。

根据上下文，以不同的方式引用每个组件。 在 **DbgPrintEx** 的 *ComponentId* 参数中，组件名称的前缀为“DPFLTR_”，后缀为“_ID”。 在注册表中，组件筛选器掩码与组件本身同名。 在调试器中，组件筛选器掩码的前缀为“Kd_”，后缀为“_Mask”。

Microsoft Windows 驱动程序工具包 (WDK) 标头 dpfilter.h 中 (DPFLTR_*XXXX*_ID格式) 的所有组件名称的完整列表。 其中大多数组件名称是为 Windows 和 Microsoft 编写的驱动程序保留的。

有六个组件名称保留给独立硬件供应商。 若要避免将驱动程序的输出与 Windows 组件的输出混合，应使用以下组件名称之一：

组件名称	**驱动程序类型**
IHVVIDEO	视频驱动程序
IHVAUDIO	音频驱动程序
IHVNETWORK	网络驱动程序
IHVSTREAMING	内核流式处理驱动程序
IHVBUS	总线驱动程序
IHVDRIVER	任何其他类型的驱动程序

例如，如果要编写视频驱动程序，请使用 DPFLTR_IHVVIDEO_ID 作为 **DbgPrintEx** 的 *ComponentId* 参数，在注册表中使用值名称 **IHVVIDEO**，并在调试器中引用 **Kd_IHVVIDEO_Mask**。

DbgPrint 和 **KdPrint** 发送的所有消息都与 **DEFAULT** 组件相关联。

选择正确的级别

DbgPrintEx 例程的 *Level* 参数的类型为 DWORD。 它用于确定 *重要性位字段*。 *Level* 参数与此位字段之间的连接取决于 *Level* 的大小：

- 如果 *Level* 等于介于 0 和 31（含）之间的数字，则解释为位移。 重要性位字段设置为值 1 << *级别*。 因此，为 *Level* 选择介于 0 和 31 之间的值会生成一个恰好设置了一个位的位字段。 如果 *Level* 为 0，则位字段等效于 0x00000001;如果 *Level* 为 31，则位字段等效于0x80000000。
- 如果 *Level* 是介于 32 和 0xFFFFFFFF（含）之间的数字，则重要性位字段设置为 *Level* 本身的值。

因此，如果要将位字段设置为0x00004000，则可以将 *Level* 指定为 0x00004000 或仅指定为 14。 请注意，此系统无法提供某些位字段值，包括完全为零的位字段。

以下常量可用于设置 *Level* 的值。 它们在 Microsoft Windows 驱动程序工具包 (WDK) 标头 dpfilter.h 和 Windows SDK 标头 ntrtl.h 中定义：

```
C++

#define   DPFLTR_ERROR_LEVEL     0
#define   DPFLTR_WARNING_LEVEL   1
#define   DPFLTR_TRACE_LEVEL     2
#define   DPFLTR_INFO_LEVEL      3
#define   DPFLTR_MASK    0x80000000
```

使用 *Level* 参数的一种简单方法是始终使用介于 0 和 31 之间的值 -- 使用位 0、1、2、3 以及DPFLTR_*XXXX*_LEVEL给出的含义，并使用其他位表示你选择的任何值。

使用 *Level* 参数的另一种简单方法是始终使用显式位字段。 如果选择此方法，则可能希望将值与位字段DPFLTR_MASK OR;这可确保不会意外使用小于 32 的值。

为了使驱动程序与 Windows 使用消息级别的方式兼容，只应在发生严重错误时设置重要性位字段的最低位 (0x1) 。 如果使用的 *级别* 值小于 32，则对应于 DPFLTR_ERROR_LEVEL。 如果设置了此位，则每当有人将内核调试器附加到运行驱动程序的计算机时，都会看到你的消息。

应在适当情况下使用警告、跟踪和信息级别。 其他位可以自由用于你认为有用的任何目的。 这允许你拥有各种可以选择性地查看或隐藏的消息类型。

DbgPrint 和 **KdPrint** 发送的所有消息的行为类似于*级别*等于 DPFLTR_INFO_LEVEL 的 **DbgPrintEx** 和 **KdPrintEx** 消息。 换句话说，这些消息设置了其重要性位字段的第三位。

设置组件筛选器掩码

有两种方法可以设置组件筛选器掩码：

- 可以在HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Debug Print Filter **的注册表项中** 访问组件筛选器掩码。 使用注册表编辑器创建或打开此密钥。 在此键下，使用所需组件的名称（大写）创建一个值。 将其设置为要用作组件筛选器掩码的 DWORD 值。

- 如果内核调试器处于活动状态，它可以通过取消引用存储在符号 **Kd_*XXXX*_Mask** 中的地址来访问组件筛选器掩码值，其中 *XXXX* 是所需的组件名称。 可以使用 **dd (Display DWORD)** 命令在 WinDbg 或 KD 中显示此掩码的值，或者使用 **ed (Enter**

DWORD) 命令输入新的组件筛选器掩码。 如果存在符号歧义的危险，建议将此符号指定为 **nt！Kd_*XXXX*_Mask**。

存储在注册表中的筛选器掩码在启动期间生效。 调试器创建的筛选器掩码会立即生效，并一直持续到 Windows 重新启动。 调试器可以重写注册表中设置的值，但如果系统重新启动，组件筛选器掩码将返回到注册表中指定的值。

还有一个名为 **WIN2000** 的系统级掩码。 默认情况下，这等于0x1，但可以像所有其他组件一样通过注册表或调试器对其进行更改。 执行筛选时，每个组件筛选器掩码都是使用 **WIN2000** 掩码的第一个 ORed。 具体而言，这意味着从未指定掩码的组件默认0x1。

显示消息的条件

在内核模式代码中调用 **DbgPrintEx** 时，Windows 会将 *Level* 指定的消息重要性位字段与 *ComponentId* 指定的组件的筛选器掩码进行比较。

> ⓘ **备注**
>
> 回想一下，当 *Level* 参数介于 0 和 31 之间时，重要性位字段等于 1 <<*Level*。 但是，当 *Level* 参数为 32 或更高时，重要性位字段仅等于 *Level*。

Windows 对重要性位字段和组件筛选器掩码执行 AND 操作。 如果结果为非零，则会将消息发送到调试器。

调试筛选器示例

假设在上次启动之前，你在 “**调试打印筛选器**” 键中创建了以下值：

- **IHVVIDEO**，其值等于 DWORD 0x2
- **IHVBUS**，等于 DWORD 0x7FF

现在，在内核调试器中发出以下命令：

```
dbgcmd
```

```
kd> ed Kd_IHVVIDEO_Mask 0x8
kd> ed Kd_IHVAUDIO_Mask 0x7
```

此时， **IHVVIDEO** 组件的筛选器掩码为 0x8， **IHVAUDIO** 组件的筛选器掩码为 0x7，**IHVBUS** 组件的筛选器掩码为 0x7FF。

但是，由于这些掩码自动使用 **WIN2000** 系统范围的掩码 (通常等于 0x1)， **因此 IHVVIDEO** 掩码实际上等于0x9。 事实上，尚未设置筛选器掩码的组件 (例如 **IHVSTREAMING** 或 **DEFAULT**) 将具有0x1的筛选器掩码。

现在，假设以下函数调用发生在各种驱动程序中：

```
C++

DbgPrintEx( DPFLTR_IHVVIDEO_ID,  DPFLTR_INFO_LEVEL,   "First message.\n");
DbgPrintEx( DPFLTR_IHVAUDIO_ID,  7,                    "Second message.\n");
DbgPrintEx( DPFLTR_IHVBUS_ID,    DPFLTR_MASK | 0x10,  "Third message.\n");
DbgPrint( "Fourth message.\n");
```

第一条消息的 *Level* 参数等于 DPFLTR_INFO_LEVEL，即 3。 由于这小于 32，因此被视为位移，导致重要性位字段0x8。 然后，使用 0x9 的有效 **IHVVIDEO** 组件筛选器掩码对此值进行 AND，从而得出非零结果。 因此，第一条消息将传输到调试器。

第二条消息的 *Level* 参数等于 7。 同样，这被视为位移，导致重要性位字段0x80。 然后使用0x7的 **IHVAUDIO** 组件筛选器掩码进行 AND，结果为零。 因此，不会传输第二条消息。

第三条消息的 *Level* 参数等于 DPFLTR_MASK |0x10。 此值大于 31，因此重要性位字段设置为等于 *Level* 的值，换言之，设置为0x80000010。 然后使用 0x7FF 的 **IHVBUS** 组件筛选器掩码进行 AND，从而得出非零结果。 因此，第三条消息将传输到调试器。

第四条消息传递到 **DbgPrint** ，而不是 **DbgPrintEx**。 在 Windows Server 2003 和早期版本的 Windows 中，始终传输传递给此例程的消息。 在 Windows Vista 和更高版本的 Windows 中，传递给此例程的消息始终具有默认筛选器。 重要性位字段等于 1 << DPFLTR_INFO_LEVEL，这是0x00000008。 此例程的组件为 **DEFAULT**。 由于尚未设置 **DEFAULT** 组件筛选器掩码，因此其值为 0x1。 当使用重要性位字段进行 AND 时，结果为零。 因此，不会传输第四条消息。

DbgPrint 缓冲区和调试器

当 **DbgPrint**、 **DbgPrintEx**、 **vDbgPrintEx**、 **vDbgPrintExWithPrefix**、 **KdPrint** 或 **KdPrintEx** 例程将消息传输到调试器时，格式化字符串将发送到 **DbgPrint** 缓冲区。 除非使用 GFlags 的 **Buffer DbgPrint 输出** 选项禁用了此显示，否则此缓冲区的内容将立即显示在“调试器命令”窗口中。

在本地内核调试期间以及禁用此显示的任何其他时间，只能使用 ！ dbgprint 扩展命令查看 DbgPrint 缓冲区的内容。

对 **DbgPrint**、 **DbgPrintEx**、 **vDbgPrintEx**、 **vDbgPrintExWithPrefix**、 **KdPrint** 或 **KdPrintEx** 的任何一次调用仅传输 512 字节的信息。 超过 512 字节的任何输出都将丢

失。 DbgPrint 缓冲区本身在 Windows 的免费版本中最多可以保存 4 KB 的数据，在已检查的 Windows 版本中最多可以容纳 32 KB 的数据。 在 Windows Server 2003 及更高版本的 Windows 上，可以使用 KDbgCtrl 工具来更改 DbgPrint 缓冲区的大小。 此工具是 Windows 调试工具的一部分。

> **备注**
>
> Windows 10 版本 1803 之前的旧版 Windows 上提供已检查的版本。 使用驱动程序验证器和 GFlags 等工具在更高版本的 Windows 中检查驱动程序代码。

如果消息由于 *ComponentId* 和 *Level* 值而被筛选掉，则不会通过调试连接传输该消息。 因此，在调试器中无法显示此消息。

确定是否已附加调试器

项目 • 2023/06/15

内核模式代码可以通过使用以下变量和例程来确定内核调试的状态：

- KD_DEBUGGER_ENABLED全局内核变量指示是否启用内核调试。
- KD_DEBUGGER_NOT_PRESENT全局内核变量指示当前是否附加了内核调试器。
- KdRefreshDebuggerNotPresent 例程刷新 KD_DEBUGGER_NOT_PRESENT 的值。

专业调试方法

项目 • 2023/03/08

本部分介绍适用于特定技术和代码模块类型的调试技术。

可以在以下主题中了解详细信息。

- 调试 Arm64
- Windows 运行时调试
- 内核模式驱动程序框架调试
- 用户模式驱动程序框架调试
- 调试托管代码
- 调试设备节点和设备堆栈
- 调试即插即用和电源问题
- 使用 KD 调试用户模式故障
- 调试设备安装包中的辅助安装程序
- 调试双重引导计算机
- 调试 Windows 安装程序和 OS 加载程序
- 调试 CSRSS
- 调试 WinLogon
- 调试 BIOS 代码
- 指定模块和函数所有者

在 Arm64 上进行调试

项目 • 2024/01/10

本文介绍在 ARM 处理器上调试Windows 10。 有关 Arm 上Windows 10的常规信息，请参阅 arm64 上的Windows 10桌面。

通常，调试用户模式应用的开发人员应使用与目标应用的体系结构匹配的调试器版本。 使用 Arm64 版本的 WinDbg 调试用户模式 Arm64 应用程序，并使用 Arm 版本的 WinDbg 调试用户模式 ARM32 应用程序。 使用 x86 版本的 WinDbg 调试在 Arm64 处理器上运行的用户模式 x86 应用程序。

在需要调试系统代码（如 WOW64 或 CHPE）的极少数情况下，可以使用 Arm64 版本的 WinDbg。 如果要从另一台计算机调试 Arm64 内核，请使用与另一台计算机的体系结构匹配的 WinDbg 版本。

获取适用于 Windows 的 Arm 调试工具

可以通过下载 Windows SDK (版本 10.0.16299 或更高版本) 来获取适用于 Arm64 的调试工具。 在安装过程中，选择“ *Windows 调试工具* ”框。

调试工具位于 `Debuggers` 工具包安装目录的 文件夹中。 x86 工具位于 下 `Debuggers\x86`，ARM32 工具位于 下 `Debuggers\Arm`，Arm64 工具位于 下 `Debuggers\Arm64`。

调试 Arm64 代码

调试 Arm64 代码需要 Arm64 WinDbg。 调试体验类似于在 x86 Windows 上使用 x86 WinDbg 调试 x86 应用程序，但存在以下差异。

- 有 32 个常规用途寄存器 - x0 到 x28 和 fp、lr、sp。
- 程序计数器寄存器（pc）不是常规用途寄存器。
- 所有常规用途寄存器和电脑寄存器的宽度均为 64 位。
- 最多两个用于执行的活动数据断点和两个用于读/写内存的活动数据断点。 有关详细信息，请参阅 处理器断点。

调试 x86 用户模式代码

在需要使用 Arm64 WinDbg 调试 x86 用户模式代码的极少数情况下，可以使用以下 WinDbg 命令在上下文之间切换：

- .effmach x86：切换到 并查看 x86 上下文，模拟使用 x86 WinDbg 的效果。
- .effmach arm64：切换到 并查看 Arm64 上下文
- .effmach chpe：切换到 并查看 CHPE 上下文。

有关 .effmach 的详细信息，请参阅 .effmach (Effective Machine) 。

在用户模式下调试 x86 应用时，无论使用的是哪个 WinDbg 版本，请注意以下注意事项。

- 如果线程未主动调试 (例如单步执行，遇到断点) ，未报告异常，也没有在系统调用中，则寄存器上下文可能不是最新的。
- 模拟器在内部生成数据未对齐、非法指令、页内 I/O 错误异常，并处理生成的异常。 使用 WinDbg 时，请考虑将这些 *异常配置为*" 调试/事件筛选器..." 菜单项来访问更新。
- 如果在用户模式下使用 Arm64 WinDbg，则不支持跨 x86 & CHPE 函数边界的单步执行。 若要解决此问题，请在目标代码上设置断点。

有关 ARM64 和 WOW64 的一般信息，请参阅 64 位 Windows 编程指南中的 运行 32 位应用程序 。

有关调试 WOW64 下运行的应用程序的信息，请参阅 调试 WOW64。

在 Visual Studio 中进行调试

有关在 Visual Studio 中调试 Arm 的信息，请参阅 远程调试。

另请参阅

- 使用 WDK 构建 Arm64 驱动程序
- 讨论始终连接的电脑的 Windows 社区站立

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

Windows driver documentation 反馈

Windows driver documentation 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

Windows 运行时调试

项目 • 2023/06/15

可以使用以下调试器扩展来调试使用Windows 运行时定义的数据类型的代码。

- !hstring
- !hstring2
- !winrterr

内核模式驱动程序框架调试

项目 • 2023/10/26

Wdfkd.dll 扩展库中包含 Kernel-Mode Driver Framework (KMDF) 的调试扩展。

可以使用 Wdfkd.dll 扩展库包含的扩展命令来调试使用 KMDF 的驱动程序。

有关 Wdfkd.dll 中扩展命令的说明，请参阅 内核模式驱动程序框架扩展 (Wdfkd.dll) 。

这些扩展可以在 Microsoft Windows XP 和更高版本的操作系统上使用。 某些扩展具有其他限制;这些限制在各个参考页上进行了说明。

注意 在创建新的 KMDF 或 UMDF 驱动程序时，必须选择一个不多于 32 个字符的驱动程序名称。 此长度限制在 wdfglobals.h 中定义。 如果驱动程序名称超过最大长度，驱动程序将无法加载。

若要使用此扩展库，必须将库加载到调试器中。 有关如何将扩展库加载到调试器的信息，请参阅 加载调试器扩展 DLL。

用户模式驱动程序框架调试

项目 • 2023/06/14

有关如何调试 User-Mode Driver Framework (UMDF) 驱动程序的概述，包括有关如何启动此类调试会话的信息，请参阅 如何启用 UMDF 驱动程序的调试。

UMDF 调试扩展

User-Mode驱动程序框架 (UMDF) 调试扩展在扩展模块Wudfext.dll中实现。 可以使用这些扩展来调试使用 UMDF 的驱动程序。

有关 Wudfext.dll 中扩展命令的完整说明，请参阅 用户模式驱动程序框架扩展 (Wudfext.dll) 。

某些扩展对所需的 Windows 版本或 UMDF 版本有其他限制;这些限制记录在各个参考页上。

注意 在创建新的 KMDF 或 UMDF 驱动程序时，必须选择一个不多于 32 个字符的驱动程序名称。 此长度限制在 wdfglobals.h 中定义。 如果驱动程序名称超过最大长度，驱动程序将无法加载。

若要使用此扩展库，必须将该库加载到调试器中。 有关如何将扩展库加载到调试器中的信息，请参阅 加载调试器扩展 DLL。

调试设备节点和设备堆栈

项目 • 2023/03/08

可以使用以下命令调试设备节点和设备堆栈。

- !devnode
- !devstack
- !devobj
- !drvobj

调试即插即用和电源问题

项目 • 2024/03/20

以下扩展可用于调试即插即用和电源问题。

- !pnpevent
- !pocaps
- !popolicy

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用 KD 调试用户模式故障

项目 • 2023/10/26

若要正确调试用户模式故障，需要 CDB 或 WinDbg。 但是，有时用户模式异常会中断 KD，因为不存在用户模式调试器。 有时，在调试内核模式问题时，监视特定用户模式进程正在执行的操作也很有帮助。

默认情况下，内核调试器尝试加载与 **k**、 **u** 或 **ln** 命令 (指定地址匹配的第一个用户模式符号) 。

遗憾的是，用户模式符号通常未在符号路径中指定，或者第一个符号不是正确的符号。 如果符号不存在，请将它们复制到符号路径，或使用 **.sympath (设置符号路径)** 命令指向完整的符号树，然后使用 **.reload (Reload Module)** 命令。 如果加载了错误的符号，可以通过执行 **.reload <binary.ext>** 来显式加载符号。

大多数 Windows DLL 已重定基，因此它们在不同的地址加载，但也有例外。 视频适配器是最常见的异常。 有数十个视频适配器都在同一基址加载，因此 KD 几乎总是会找到 ati.dll (第一个视频符号，按字母顺序) 。 对于视频，还加载了一个 .sys 文件，可以使用 **lm** 命令进行标识。 有了这些信息，你可以发出 **.reload** 来获取正确的视频 DLL。 有时调试器也会变得混乱，重新加载特定符号将有助于提供正确的堆栈。 取消组合函数以查看符号是否正确。

与视频 DLL 类似，几乎所有可执行文件都在同一地址加载，因此 KD 将报告访问权限。 如果在访问中看到堆栈跟踪，请执行 **！ 进程** ，然后执行给定的可执行文件名称的 **.reload** 。 如果可执行文件在符号路径中没有符号，请将其复制到该处，然后再次执行 **.reload** 。

有时，即使完整的符号树位于符号路径中，KD 或 WinDbg 也难以加载正确的用户模式符号。 在这种情况下，ntdll.dll 和 kernel32.dll 是所需的两个最常见的符号。 从 KD 调试 CSRSS 时，winsrv.dll 和 csrsrv.dll 也是要加载的常见 DLL。

调试设备安装包中的辅助安装程序

项目 • 2023/10/26

某些硬件设备安装包包括称为 *共同安装程序的* DLL 文件，这些文件有助于安装设备。

不能以与其他模块相同的方式调试共同安装程序。 这是因为加载共同安装程序的独特方式，并且许多安装方案会自动发生，而不会为开发人员提供进入正在运行的进程的机会。

可以通过以编程方式安装设备来解决此问题。 将调试器附加到安装设备的应用程序允许访问共同安装程序本身。 实现此目的的最简单方法是使用 Windows 驱动程序工具包 (WDK) 中包含的 DevCon 工具安装或重新安装设备。 然后，可以使用 WinDbg 调试共同安装程序。

使用以下过程完成此任务。 此过程假定你已为使用共同安装程序的设备开发了一个工作驱动程序安装包。 它还假定你拥有 WDK 的最新副本。 有关开发驱动程序、驱动程序安装包和驱动程序安装共同安装程序的信息，请参阅 WDK 文档。

使用 DevCon 和 WinDbg 调试共同安装程序

1. 插入硬件设备。
2. 取消“ **查找新硬件** ”向导。
3. 启动 WinDbg。
4. 从 WinDbg **的“文件”菜单中选择“打开可执行文件”**。
5. 在“ **打开可执行文件** ”对话框中，执行以下操作：
   a. 在文件选择文本框中，选择 DevCon 工具 (Devcon.exe) 。 为此，请浏览到 WDK 安装文件夹，打开子目录工具，打开子目录 devcon，然后打开与计算机的处理器体系结构匹配的子目录，然后选择“Devcon.exe”。 在 Devcon.exe 上仅单击一次，并且尚未按 “**打开**”。
   b. 在“ **参数** ”文本框中，输入以下文本，其中 *INFFile* 是设备安装信息 (INF) 文件的文件名， *HardwareID* 是设备的硬件 ID：

      ```
      text

      update INFFile HardwareID
      ```

   c. 在 “**开始目录** ”文本框中，输入设备安装包的路径。
   d. 单击 “**打开**” 。

6. 调试过程将开始，WinDbg 将在 DevCon 安装驱动程序之前进入 DevCon 进程。

7. 配置调试器，使其在加载时进入共同安装程序进程。 可以通过以下任一方法执行此操作：

   - 在“调试器命令”窗口中，使用 **sxe (设置异常)** 命令，后跟 **ld:** ，然后使用共同安装程序的文件名（不包括文件扩展名）。 冒号后应没有空格 例如，如果共同安装程序的名称 mycoinst.dll，请使用以下命令：

     ```
     dbgcmd

     sxe ld:mycoinst
     ```

   - 从 WinDbg 的“**调试**”菜单中选择“**事件筛选器**”。 在“ **事件筛选器** ”对话框中，选择“ **加载模块**”。 在“ **执行**”下，选择“ **已启用**”。 在 “**继续**”下，选择“ **未处理**”。 单击“ **参数** ”按钮，然后在文本框中输入共同安装程序的文件名（例如，不包括文件扩展名 (为 mycoinst.dll) 。 单击 “**确定** ”，然后单击“ **关闭**”。

8. 按 F5 或在“调试器命令”窗口中输入 **g (Go)** 命令以继续执行。

9. 加载共同安装程序时，执行将中断到调试器中。 此时，可以设置所需的任何其他断点。

此过程的限制

在某些情况下，在 DevCon 下运行设备安装包可能会导致与 PnP 安装的行为略有不同，因为安全令牌等不同。 如果尝试调试共同安装程序中的特定问题，则如果涉及 DevCon，则此问题可能不会复制。 因此，在使用此方法之前，应使用 DevCon 在未附加调试器的情况下安装驱动程序，以验证 PnP 和 DevCon 方案中是否存在此问题。

如果问题在 DevCon 启动安装时消失，则必须在不使用 DevCon 的情况下调试共同安装程序。 执行此操作的一种方法是将 TList 工具与 **/m** 选项配合使用，以确定哪个进程正在加载共同安装程序模块，然后将调试器附加到该进程。

调试双重引导计算机

项目 • 2023/06/15

当备用操作系统未在双启动计算机上启动时，应如何响应？

首先，检查启动选项指向其他操作系统的正确路径。 有关详细信息，请参阅 Getting Set Up for Debugging 。

在 x86 计算机上，还应验证是否存在boosect.ini。 此文件包含其他操作系统的启动记录。 若要取消隐藏此文件，请使用 **attrib -r -s -h boosect.ini** 命令。

调试 Windows 安装程序和 OS 加载程序

项目 • 2023/06/15

在启动序列期间进行调试

有关在启动时调试计算机的信息，请参阅以下主题。

- **BCDEdit /bootdebug**
- BCD 启动选项参考

调试 CSRSS

项目 • 2023/10/26

客户端服务器 Run-Time 子系统 (CSRSS) 是控制 Windows 环境底层的用户模式进程。

> **备注**
>
> 从 Windows 10 开始，CSRSS 是一个受保护的进程，只能在内核模式下进行调试。

有关受保护进程的一般信息，以及有关 Windows 保护的关键代码（如 wininit 和 csrss）的其他详细信息，请参阅由 Pavel Yosifovich、Mark E. Russinovich、David A. Solomon 和 Alex Ionescu 撰写的 *Windows Internals* 。

显示 CSRSS 进程信息

若要检查 CSRSS，可以使用内核调试获取一些信息。

使用 **!** process 扩展显示与 csrss.exe 关联的进程的相关信息。

dbgcmd

```
0: kd> !process 0 0 csrss.exe
PROCESS ffffe381a583b080
    SessionId: 0  Cid: 027c    Peb: e0c93ef000  ParentCid: 0270
    DirBase: 115478000  ObjectTable: ffffaa87786b67c0  HandleCount: 722.
    Image: csrss.exe

PROCESS ffffe381a68ab140
    SessionId: 1  Cid: 02f4    Peb: 186a447000  ParentCid: 02dc
    DirBase: 143c0e000  ObjectTable: ffffaa87786b5200  HandleCount: 445.
    Image: csrss.exe
```

获取其中一个关联的进程，并使用 .process (Set Process Context) 命令将上下文设置为该位置。

dbgcmd

```
0: kd> .process /r /p ffffe381a583b080
Implicit process is now ffffe381`a583b080
Loading User Symbols
```

现在，使用 dt (Display Type) 命令直接显示进程结构：

dbgcmd

```
0: kd> dt csrss!_csr_process
   +0x000 ClientId         : _CLIENT_ID
   +0x010 ListLink         : _LIST_ENTRY
   +0x020 ThreadList       : _LIST_ENTRY
   +0x030 NtSession        : Ptr64 _CSR_NT_SESSION
   +0x038 ClientPort       : Ptr64 Void
   +0x040 ClientViewBase   : Ptr64 Char
   +0x048 ClientViewBounds : Ptr64 Char
   +0x050 ProcessHandle    : Ptr64 Void
   +0x058 SequenceNumber   : Uint4B
   +0x05c Flags            : Uint4B
   +0x060 DebugFlags       : Uint4B
   +0x064 ReferenceCount   : Int4B
   +0x068 ProcessGroupId   : Uint4B
   +0x06c ProcessGroupSequence : Uint4B
   +0x070 LastMessageSequence : Uint4B
   +0x074 NumOutstandingMessages : Uint4B
   +0x078 ShutdownLevel    : Uint4B
   +0x07c ShutdownFlags    : Uint4B
   +0x080 Luid             : _LUID
   +0x088 ServerDllPerProcessData : [1] Ptr64 Void
```

! peb 扩展可用于显示有关进程环境块 (PEB) 的其他信息。

调试 WinLogon

项目 • 2023/06/15

WinLogon 是一个用户模式进程，用于处理交互式用户登录和注销的任务，并处理 CTRL+ALT+DELETE 的所有实例。

从内核调试器控制 NTSD

调试 WinLogon 的最简单方法是使用 NTSD 并从内核调试器控制它。

启用 WinLogon 调试

由于需要将用户模式调试器输出重定向到内核调试器，因此需要设置内核调试连接。 请参阅 设置调试。

若要将调试器附加到 WinLogon，必须完成注册表，以便从启动进程开始调试。 若要设置 WinLogon 调试，请将 **HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\WinLogon.EXE\Debugger** 设置为：

控制台

```
ntsd -d -x -g
```

-d 选项将控制权传递给内核调试器。 **-x** 选项会导致调试器将访问冲突捕获为第二次机会异常。 **-g** 选项会导致 WinLogon 进程在附件之后运行。 如果要在Winlogon.exe开始前开始调试 (例如，如果要设置初始断点)，请不要添加 **-g** 。

此外，应将 **winlogon.exe** 键下的 GlobalFlag 值设置为REG_DWORD“0x000400F0”。 这会设置堆检查和FLG_ENABLE_KDEBUG_SYMBOL_LOAD。 但是，由于第二个标志仅影响内核调试器，因此在启动调试器之前，还必须将符号复制到目标计算机。

注册表更改需要重新启动才能生效。

执行调试

下次重新启动后，调试器将自动进入 WinLogon。

有关如何继续的说明，请参阅 从内核调试器控制User-Mode调试器 。

你必须将符号路径设置为主计算机上的某个位置或网络上的某个其他位置。 调试 WinLogon 时，目标计算机上的网络身份验证将无法正常工作。

调试 BIOS 代码

项目 • 2023/06/15

BIOS 代码不是从标准程序集代码生成的，因此需要不同的调试技术。

在基于 x86 的处理器上，BIOS 使用 16 位代码。 若要反汇编此代码，请使用 **ux (Unassemble x86 BIOS)** 命令。 若要显示有关 Intel 多处理器规范 (MPS) 的信息，请使用 **！ mps** 扩展。

如果要调试 ACPI BIOS 代码，上述命令不起作用，因为 ACPI BIOS 是用 ACPI 计算机语言 (AML) 编写的。 若要反汇编此代码，应使用 **！ amli u**。 有关此类调试的详细信息，请参阅 ACPI 调试。

指定模块和函数所有者

项目 • 2023/10/26

！analyze 和 **！owner** 扩展使用名为 triage.ini 的文件来确定调试器遇到的符号的所有者。

使用这些扩展时，函数或模块所有者的标识会显示在单词“Followup”后面。

triage.ini 文件是驻留在 Windows 调试工具安装的 \triage 子目录中的文本文件。 示例 triage.ini 文件包含在 Windows 调试工具包中。

警告 如果在当前版本的同一目录中安装更新版本的 Windows 调试工具，它将覆盖该目录中的所有文件，包括 triage.ini。 修改或替换示例 triage.ini 文件后，将其副本保存到其他目录。 重新安装调试器后，可以复制默认版本上保存的 triage.ini。

triage.ini 文件格式

尽管 triage.ini 文件旨在帮助你确定已进入调试器的函数的所有者，但此文件中的“owner”字符串可以是任何可能对调试有帮助的内容。 字符串可以是编写或维护代码的人员的姓名。 或者，字符串可以是有关在模块或函数中发生错误时可以执行的操作的简短说明。

此文件中的每一行都有以下语法。

dbgcmd

```
Module[!Function]=Owner
```

只能在模块或函数名称的末尾添加星号 (*) 。 如果它出现在其他位置，则将其解释为文本字符。

不能在所有者字符串中添加空格。 如果所有者字符串中确实存在空格，则忽略它们。

有关语法选项的详细信息，请参阅 特殊 triage.ini 语法。

以下示例演示一个示例 triage.ini 文件。

ini

```
module1=Person1
module2!functionA=Person2
module2!functionB=Person3
module2!funct*=Person4
module2!*=Person5
```

```
module3!singleFunction=Person6
mod*!functionC=Person7
```

Triage.ini 和 ！ owner

将模块或函数名称传递给 **！ owner** 扩展时，调试器将显示单词“Followup”，后跟模块或函数的所有者的名称。

以下示例使用上一个示例 triage.ini 文件。

```
dbgcmd
```

```
0:000> !owner module2!functionB
Followup:  Person3
```

根据该文件，“Person3”拥有 **module2！functionB**，“Person4”拥有 **module2！funct***。 *这两个字符串都匹配传递给 **！ owner** 的参数，因此使用更完整的匹配。*

Triage.ini 和 ！ analyze

*使用 **！ analyze** 扩展时，调试器将查看堆栈中的顶部故障帧，并尝试确定此帧中模块和函数的所有者。 如果调试器可以确定所有者，则会显示所有者信息。*

如果调试器无法确定所有者，调试器将传递到下一个堆栈帧，依此推，直到调试器确定所有者或完全检查堆栈为止。

如果调试器可以确定所有者，则会在单词“Followup”后面显示所有者名称。 如果调试器搜索整个堆栈而未找到任何信息，则不会显示任何名称。

以下示例使用本主题前面提供的示例 triage.ini 文件。

*假设堆栈上的第一个帧是 **MyModule！ someFunction**。 调试器在 triage.ini 文件中找不到 **MyModule** 。 接下来，它继续到堆栈上的第二个帧。*

*假设第二个帧是 **module3！ anotherFunction**。 调试器确实看到 **module3** 的条目，但此模块中 **没有与另一个Function** 匹配项。 接下来，调试器继续到第三帧。*

*假设第三个帧是 **module2！ functionC**。 调试器首先查找完全匹配项，但此类匹配不存在。 然后，调试器会剪裁函数名称，并在 triage.ini 中发现 **module2！ funct***。 此匹配将结束搜索，因为调试器确定所有者为“Person4”。*

然后，调试器会显示类似于以下示例的输出。

```
dbgcmd
```

```
0:000> !analyze
*******************************************************************************
***
*
*
*                        Exception Analysis
*
*
*
*******************************************************************************
***

Use !analyze -v to get detailed debugging information.

Probably caused by : module2 ( module2!functionC+15a )

Followup: Person4
---------
```

更完整的匹配优先于较短的匹配。 但是，模块名称匹配始终优先于函数名称匹配。 如果 ***module2！funct**** *未在此 triage.ini 文件中，则调试器会选择* ***module2！**** *作为匹配项。 如果同时删除* ***module2！funct**** *和* ***module2！**** *，则会选择* ***mod*！functionC*** *。*

特殊 triage.ini 语法

如果省略感叹号和函数名称或在模块名称后添加 ***！****，则会指示该模块中的所有函数。 如果此模块中的函数也单独指定，则更精确的规范优先。*

如果使用"default"作为模块名称或函数名称，则它等效于通配符。 例如， ***nt！**** *与* ***nt！default*** *相同，* ***默认值*** *与* ****！**** *相同。*

如果进行了匹配，但" ***忽略*** *"一词出现在等号 (=) 右侧，则调试器将继续转到堆栈中的下一帧。*

可以在所有者姓名之前添加 ***last_*** *或* ***maybe_*** *。 运行* ***！analyze*** *时，此前缀会降低所有者的优先级。 调试器选择堆栈上较低的明确匹配项，而选择堆栈上较高的* ***maybe_*** *匹配项。 调试器还会选择堆栈上较低的* ***maybe_*** *匹配项，而选择堆栈上较高的* ***last_*** *匹配项。*

示例 triage.ini

适用于 Windows 的调试工具包中包含示例 triage.ini 模板。 可以将所需的任何模块和函数的所有者添加到此文件。 如果希望没有全局默认值，请删除此文件开头的 ***default=MachineOwner*** *行。*

RPC 调试

项目 • 2023/03/08

本节包括：

RPC 调试概述

启用 RPC 状态信息

显示 RPC 状态信息

常用 RPC 调试方法

RPC 状态信息的内部结构

RPC 调试概述

项目 • 2023/06/14

Microsoft 远程过程调用 (RPC) 可以轻松地跨越进程和计算机边界并携带数据。 此网络编程标准是 Microsoft Windows 网络功能强大的原因之一。

但是，由于 RPC 对单个进程隐藏网络调用，因此它会遮盖计算机之间交互的详细信息。 这样就很难确定线程为何正在执行它们正在执行的操作，或者无法执行它们应该执行的操作。 因此，调试和排查 RPC 错误可能很困难。 此外，绝大多数看似 RPC 错误的问题实际上是配置问题、网络连接问题或其他组件问题。

Windows 调试工具包含名为 DbgRpc 的工具，以及与 RPC 相关的调试器扩展。 这些可用于分析 Windows XP 和更高版本的 Windows 上的各种 RPC 问题。

可以将这些 Windows 版本配置为保存 RPC 运行时状态信息。 可以保存不同数量的状态信息;这允许你获取所需的信息，而不会给计算机带来重大负担。 有关详细信息，请参阅启用 RPC 状态信息 。

然后，可以通过调试器或 DbgRpc 工具访问此信息。 在每种情况下，都有可用的查询集合。 有关详细信息，请参阅显示 RPC 状态信息 。

在许多情况下，可以使用 常见 RPC 调试技术中概述的技术来排查问题。

如果要探索存储此信息的机制，或者想要设计自己的状态信息分析技术，请参阅 RPC 状态信息内部。

启用 RPC 状态信息

项目 • 2023/06/15

可以收集两个不同级别的 RPC 运行时状态信息： **服务器** 信息和 **完整** 信息。 必须先启用此信息收集，然后调试器或 DbgRpc 才能用于分析状态信息。

只有 Windows XP 和更高版本的 Windows 支持收集 RPC 状态信息。

收集 **服务器** 状态信息非常轻量。 每次 RPC 调用大约需要 100 个计算机指令，因此即使在性能测试期间，也无法检测到负载。 但是，收集此信息会占用每个 RPC 服务器) 大约 4KB 的内存 (，因此不建议在已遇到内存压力的计算机上使用内存。 **服务器** 信息包括有关终结点、线程、连接对象和服务器调用 (SCALL) 对象的数据。 这足以调试大多数 RPC 问题。

收集 **完整** 状态信息更重。 它包括 **服务器** 级别收集的所有信息，此外，还包括客户端调用 (CCALL) 对象。 通常不需要**完整的**状态信息。

若要允许在单个计算机上收集状态信息，请运行 组策略 编辑器 (Gpedit.msc) 。 在“本地计算机策略”下，导航到 “**计算机配置/管理模板/系统/远程过程调用**”。 在此节点下，你将看到 “**维护 RPC 故障排除状态信息**” 项。 编辑其属性时，将看到五种可能的状态：

没有
不会维护任何状态信息。 除非计算机遇到内存压力，否则不建议这样做。

服务器
将收集**服务器**状态信息。 这是单台计算机上的推荐设置。

全
将收集**完整的**状态信息。

Auto1
在 RAM 小于 64 MB 的计算机上，这与 **None** 相同。 在 RAM 至少为 64 MB 的计算机上，这与 **服务器**相同。

Auto2
在运行 RAM 少于 128 MB 的 Windows Server 2003 的计算机上，或任何 Windows XP 计算机上，这与 **None** 相同。 在具有至少 128 MB RAM 的 Windows Server 2003 计算机上，这与 **服务器**相同。

这是默认设置。

如果要在一组联网计算机上同时设置这些级别，请使用组策略编辑器将计算机策略推出到首选计算机集。 策略引擎将负责将所需的设置传播到首选计算机集。 在这种情况下，

Auto1 和 **Auto2** 级别特别有用，因为每台计算机上的操作系统和 RAM 量可能会有所不同。

如果网络包含运行早于 Windows XP 的 Windows 版本的计算机，则这些计算机上的设置将被忽略。

显示 RPC 状态信息

项目 • 2023/06/15

所有 RPC 运行时状态信息都包含在单元格中。 单元格是可以单独查看和更新的最小信息单位。 DbgRpc 工具和 RPC 调试器扩展都允许查看任何给定单元格的内容或运行高级查询。

RPC Run-Time中的每个键对象都将保留一个或多个有关其状态的信息单元格。 每个单元格都有一个单元格 ID。 当对象引用另一个对象时，它通过指定该对象的单元格 ID 来执行此操作。

RPC Run-Time可以维护相关信息的关键对象包括终结点、线程、连接对象、服务器调用 (SCALL) 对象和客户端调用 (CCALL) 对象。 服务器调用对象通常简称为 *调用对象*。

无论使用的是 DbgRpc 工具还是 RPC 调试器扩展，RPC 状态信息查询都生成相同的信息。 以下部分介绍了如何在每辆车中使用查询：

使用 RPC 调试器扩展

使用 DbgRpc 工具

最基本的查询仅显示单个单元格：

获取 RPC 单元信息

还可以使用以下高级查询：

获取 RPC 终结点信息

获取 RPC 线程信息

获取 RPC 调用信息

获取 RPC 客户端调用信息

使用 RPC 调试器扩展

项目 • 2023/04/14

从 Rpcexts.dll 导出各种 RPC 调试器扩展。

用于显示 RPC 状态信息的 RPC 扩展将仅在用户模式下运行。 可以从 CDB (或 NTSD) 或用户模式 WinDbg 使用它们。

用户模式调试器必须具有目标应用程序，但目标与 RPC 扩展无关。 如果调试器尚未运行，只需使用不感兴趣的目标 (启动它，例如 **windbg 记事本** 或 **cdb winmine**) 。 然后，在 CDB 或调试中使用 **CTRL+C**|**中断** WinDbg 以停止目标并访问调试器命令窗口。

如果需要分析远程计算机的 RPC 状态信息，则应在需要分析的计算机上启动用户模式调试器，然后使用 远程调试。

在压力环境中或调试器恰好正在运行时，通过调试器访问 RPC 状态信息特别有用。

使用 DbgRpc 工具

项目 • 2023/06/15

DbgRpc 工具 (Dbgrpc.exe) 位于 Windows 调试工具安装的根目录中，必须在命令提示符窗口中启动。 双击图标不会启动此工具。

命令提示符窗口必须在具有本地计算机上的管理权限或域管理权限的帐户下运行。

DbgRpc 不调用任何系统服务 (，例如 LSASS) 。 这样，即使系统服务崩溃，只要内核仍在运行，它也能用于调试。

在远程计算机上使用 DbgRpc

DbgRpc 还可用于检查远程计算机中的信息。 若要使此功能正常工作，远程计算机必须能够接受远程连接并验证远程用户身份。 如果远程计算机的 RPCSS (RPC Endpoint Mapper) 服务崩溃，DbgRpc 将无法正常工作。 需要远程计算机上的管理或域管理权限。

-s 命令行选项用于指定服务器名称，**-p** 参数用于指定传输协议。 TCP 和命名管道协议均可用。 TCP 是建议的协议;它几乎应该在每种情况下都起作用。

以下是示例：

控制台

```
G:\>dbgrpc -s MyServer -p ncacn_ip_tcp -l -P 1e8 -L 0.1
Getting remote cell info ...
Endpoint
Status: Active
Protocol Sequence: LRPC
Endpoint name: OLE18
```

DbgRpc 命令行

有关完整命令语法的说明，请参阅 DbgRpc Command-Line **选项**。

获取 RPC 单元信息

项目 • 2023/06/15

使用 -l 开关时，**！ rpcexts.getdbgcell** 扩展或 DbgRpc 会显示详细的单元格信息。

必须指定包含首选单元格的进程的进程 ID 以及单元格编号。

在以下示例中，进程 ID 为 0x278，单元格号为 0000.0002：

控制台

```
D:\wmsg>dbgrpc -l -P 278 -L 0.2
Getting cell info ...
Thread
Status: Dispatched
Thread ID: 0x1A4 (420)
Last update time (in seconds since boot):470.25 (0x1D6.19)
```

有关可选参数的详细信息，请参阅 **DbgRpc Command-Line 选项**。

有关使用 RPC 调试器扩展的类似示例，请参阅 **！ rpcexts.getdbgcell**。

获取 RPC 终结点信息

项目 • 2023/10/26

终结点信息由 **！ rpcexts.getendpointinfo** 扩展显示，或使用 **-e** 开关时由 DbgRpc 显示。

如果指定了终结点编号，则会显示有关该终结点的信息。 如果省略它，则会显示系统上所有进程的终结点。

以下示例显示所有终结点。 这通常是获取可用作其他命令参数的进程 ID 和单元格号的有用方法：

控制台

```
D:\wmsg>dbgrpc -e
Searching for endpoint info ...
## PID  CELL ID   ST PROTSEQ        ENDPOINT
-------------------------------------------------------
00a8 0000.0001 01            NMP \PIPE\InitShutdown
00a8 0000.0003 01            NMP \PIPE\SfcApi
00a8 0000.0004 01            NMP \PIPE\ProfMapApi
00a8 0000.0007 01            NMP \pipe\winlogonrpc
00a8 0000.0008 01           LRPC OLE5
00c4 0000.0001 01           LRPC ntsvcs
00c4 0000.0003 01            NMP \PIPE\ntsvcs
00c4 0000.0008 01            NMP \PIPE\scerpc
00d0 0000.0001 01            NMP \PIPE\lsass
00d0 0000.0004 01            NMP \pipe\WMIEP_d0
00d0 0000.000b 01            NMP \PIPE\POLICYAGENT
00d0 0000.000c 01           LRPC policyagent
0170 0000.0001 01           LRPC epmapper
0170 0000.0003 01            TCP 135
0170 0000.0005 01            SPX 34280
0170 0000.0006 01             NB 135
0170 0000.0007 01             NB 135
0170 0000.000b 01            NMP \pipe\epmapper
01b8 0000.0001 01            NMP \pipe\spoolss
01b8 0000.0003 01           LRPC spoolss
01b8 0000.0007 01           LRPC OLE7
00ec 0000.0001 01           LRPC OLE2
00ec 0000.0003 01           LRPC senssvc
00ec 0000.0007 01            NMP \pipe\tapsrv
00ec 0000.0008 01           LRPC tapsrvlpc
00ec 0000.000c 01            NMP \PIPE\ROUTER
00ec 0000.0010 01            NMP \pipe\WMIEP_ec
0214 0000.0001 01            NMP \PIPE\winreg
022c 0000.0001 01           LRPC LRPC0000022c.00000001
022c 0000.0003 01            TCP 1058
022c 0000.0005 01            SPX 24576
```

```
022c 0000.0006 01            NMP \PIPE\atsvc
02a8 0000.0001 01           LRPC OLE3
0370 0000.0001 01           LRPC OLE9
0278 0000.0001 01            TCP 1120
030c 0000.0001 01           LRPC OLE12
```

有关可选参数的详细信息，请参阅 DbgRpc Command-Line **选项**。

有关使用 RPC 调试器扩展的类似示例，请参阅 **！** rpcexts.getendpointinfo。

获取 RPC 线程信息

项目 • 2023/10/26

线程信息由 **！ rpcexts.getthreadinfo** 扩展显示，或使用 **-t** 开关时由 DbgRpc 显示。

必须指定进程的 PID。 还可以在该进程中指定线程。 如果省略线程，将显示该进程中的所有线程。

在以下示例中，进程 ID 为0x278，线程 ID 被省略：

控制台

```
D:\wmsg>dbgrpc -t -P 278
Searching for thread info ...
## PID  CELL ID   ST TID      LASTTIME
-----------------------------------
0278 0000.0002 01 000001a4 00072c09
0278 0000.0005 03 0000031c 00072bf5
```

有关可选参数的详细信息，请参阅 DbgRpc Command-Line **选项**。

有关使用 RPC 调试器扩展的类似示例，请参阅 **！ rpcexts.getthreadinfo**。

获取 RPC 调用信息

项目 • 2023/10/26

服务器端调用 (SCALL) 信息由 **！ rpcexts.getcallinfo** 扩展显示，或使用 **-c** 开关时由 DbgRpc 显示。

允许使用四个可选参数。 其中三个 （ *CallID*、 *IfStart* 和 *ProcNum* ） 标识 RPC 用于跟踪其调用的信息。 第四个参数 *ProcessID* 是拥有调用的服务器进程的 PID。 应提供任何已知参数以缩小搜索范围。

如果未提供任何参数，将显示系统中的所有已知 SCALLS。 下面是此长显示的示例：

控制台

```
D:\wmsg>dbgrpc -c
Searching for call info ...
## PID  CELL ID   ST PNO IFSTART  TIDNUMBER CALLFLAG CALLID   LASTTIME
CONN/CLN
-------------------------------------------------------------------------
00c4 0000.0002 00 00f 82273fdc 0000.0007 00000001 00000002 0003595d
0000.0010
00c4 0000.0006 00 009 367abb81 0000.0015 00000001 0000004d 000185bd
0000.0005
00c4 0000.000a 00 007 367abb81 0000.002d 00000001 0000009f 00014672
0000.0009
00c4 0000.000c 00 007 367abb81 0000.002d 00000001 00000083 000122e3
0000.000b
00c4 0000.000d 00 03b 8d9f4e40 0000.002d 00000001 000000f7 0001aba5
0000.0020
00c4 0000.000e 00 03b 8d9f4e40 0000.0026 00000001 00000002 00023056
0000.0021
00c4 0000.000f 00 008 82273fdc 0000.001e 00000009 baadf00d 000366b4
00ec.03bc
00c4 0000.0012 00 00d 8d9f4e40 0000.0004 00000001 00000051 0000a334
0000.0011
00c4 0000.0014 00 000 367abb81 0000.0015 00000001 0000004c 0002db53
0000.0013
00c4 0000.0017 00 007 367abb81 0000.0015 00000001 00000006 0000d102
0000.0016
00c4 0000.0019 00 007 367abb81 0000.0004 00000001 00000006 0000f09e
0000.0018
00c4 0000.001b 00 009 65a93890 0000.0007 00000001 0000012e 00630f65
0000.001a
00c4 0000.001e 00 026 8d9f4e40 0000.0015 00000001 0000037d 0005e579
0000.002c
00c4 0000.001f 00 008 82273fdc 0000.0033 00000009 baadf00d 000145b3
00c4.02f8
00c4 0000.0023 00 000 367abb81 0000.0004 00000001 0000007e 000372f3
0000.0022
```

```
00c4 0000.0025 00 03b 8d9f4e40 0000.0026 00000001 0000000b 000122e3
0000.0024
00c4 0000.0027 00 000 367abb81 0000.002d 00000001 0000000b 00012e27
0000.0028
00c4 0000.002a 00 008 82273fdc 0000.0033 00000009 baadf00d 0001245f
022c.0290
00c4 0000.002f 00 007 367abb81 0000.0026 00000001 0000000a 0002983c
0000.002e
00c4 0000.0031 00 004 3ba0ffc0 0000.0026 00000001 00000007 0005c439
0000.001c
00c4 0000.0032 00 00b 82273fdc 0000.0039 00000009 baadf00d 00687db6
00d0.01d4
00c4 0000.0036 00 007 367abb81 0000.0030 00000001 00000065 0003a5e1
0000.0035
00c4 0000.0037 00 00d 8d9f4e40 0000.0015 00000001 0000033f 000376fa
0000.002b
00c4 0000.0038 00 008 8d9f4e40 0000.0015 00000001 00000803 0018485c
0000.003b
00c4 0000.003c 00 00b 82273fdc 0000.0034 00000009 baadf00d 0001f956
00a8.0244
00c4 0000.003d 00 008 82273fdc 0000.0034 00000009 baadf00d 0001ff02
01b8.037c
0170 0000.0009 00 002 e60c73e6 0000.0013 00000009 baadf00d 0005a371
00ec.031c
0170 0000.000a 00 002 0b0a6584 0000.0002 00000009 baadf00d 000126ae
00c4.0130
0170 0000.000c 00 002 0b0a6584 0000.0010 00000009 baadf00d 00012bc4
022c.0290
0170 0000.000d 00 003 00000136 0000.001b 00000009 baadf00d 0005ba71
00ec.0310
0170 0000.000e 00 000 412f241e 0000.0002 00000009 baadf00d 00012f21
02a8.029c
0170 0000.0010 00 003 00000136 0000.0013 00000009 00000003 000341da
0370.0060
0170 0000.0011 00 006 e60c73e6 0000.001b 00000009 baadf00d 000f1d00
0370.0328
0170 0000.0017 00 002 0b0a6584 0000.001b 00000009 baadf00d 0006c803
0278.0184
0170 0000.001a 00 004 00000136 0000.0012 00000001 baadf00d 00038e9b
00ec.0348
00ec 0000.0006 00 009 00000134 0000.0011 00000009 baadf00d 000b233f
0170.0244
00ec 0000.000b 00 001 2f5f6520 0000.001c 00000009 baadf00d 00035510
00ec.0334
00ec 0000.000e 00 001 629b9f66 0000.0014 00000009 baadf00d 00035813
00ec.01c4
00ec 0000.0012 00 000 629b9f66 0000.0014 00000009 baadf00d 00026cc6
00a8.0164
00ec 0000.001b 00 001 2f5f6520 0000.0004 00000001 baadf00d 000352c1
00ec.03a8
02a8 0000.0004 00 009 00000134 0000.0002 00000009 baadf00d 0009a540
0170.0244
0370 0000.0006 00 003 00000134 0000.0005 0000000b baadf00d 0002e7cd
00ec.0350
0370 0000.0008 00 009 00000134 0000.0007 0000000b 01cee9e4 000838fa
```

```
0170.0244
0278 0000.0004 02 000 19bb5061 0000.0002 00000001 00000001 00072c09
0000.0003
```

有关可选参数的详细信息，请参阅 **DbgRpc Command-Line 选项**。

有关使用 RPC 调试器扩展的类似示例，请参阅 **！ rpcexts.getcallinfo**。

获取 RPC 客户端调用信息

项目 • 2023/10/26

客户端调用 (CCALL) 调用信息由 **！ rpcexts.getclientcallinfo** 扩展显示，或使用 **-a** 开关时由 DbgRpc 显示。

允许使用四个可选参数。 其中三个（ *CallID*、 *IfStart* 和 *ProcNum* ）是 RPC 用来跟踪其调用的标识信息。 第四个参数 *ProcessID* 是拥有调用的进程 PID。 应提供任何已知参数以缩小搜索范围。

如果未提供任何参数，将显示系统中的所有已知 CCALLS。 下面是此 (可能较长) 显示的示例：

控制台

```
D:\wmsg>dbgrpc -a
Searching for call info ...
## PID  CELL ID   PNO  IFSTART  TIDNUMBER CALLID   LASTTIME PS CLTNUMBER
ENDPOINT
--------------------------------------------------------------------------------
--
0390 0000.0001 0000 19bb5061 0000.0000 00000001 00072bff 07 0000.0002 1120
```

有关可选参数的详细信息，请参阅 **DbgRpc Command-Line 选项**。

有关使用 RPC 调试器扩展的类似示例，请参阅 **！ rpcexts.getclientcallinfo**。

注意 仅当正在收集 **完整** 状态信息时，才会收集有关客户端调用对象的信息。 有关详细信息，请参阅启用 RPC 状态信息 。

常用 RPC 调试方法

项目 • 2023/06/15

本部分介绍四个与 RPC 相关的常见问题。 RPC 状态信息可用于排查这些问题。

DbgRpc 工具或 RPC 调试器扩展可用于这四种情况中的任何一种。

分析调用阻塞问题

跟踪服务器进程中的争用

检查线程阻塞问题

识别服务器线程中的调用方

分析调用阻塞问题

项目 • 2024/03/01

当进程在直接或间接地执行 RPC 调用时，如果持有关键节或资源，就会出现常见问题。 在这种情况下，RPC 调用将转到另一个进程或计算机，并调度到管理器例程（服务器例程），然后就会挂起或耗时过长。 这会导致原始调用方遇到关键节超时。

通过调试器进行检查时，RPC 位于拥有关键节的线程堆栈之上，但不清楚它正在等待的内容。

下面是此类堆栈的一个示例。 可以有很多变化。

dbgcmd

```
0:002> ~1k
ChildEBP RetAddr
0068fba0 77e9e8eb ntdll!ZwWaitForSingleObject+0xb
0068fbc8 4efeff73 KERNEL32!WaitForSingleObjectEx+0x5a
0068fbe8 4eff0012 RPCRT4!UTIL_WaitForSyncIO+0x21
0068fc0c 4efe6e2b RPCRT4!UTIL_GetOverlappedResultEx+0x44
0068fc44 4ef973bf RPCRT4!WS_SyncRecv+0x12a
0068fc68 4ef98d5a RPCRT4!OSF_CCONNECTION__TransSendReceive+0xcb
0068fce4 4ef9b682 RPCRT4!OSF_CCONNECTION__SendFragment+0x297
0068fd38 4ef9a5a8 RPCRT4!OSF_CCALL__SendNextFragment+0x272
0068fd88 4ef9a9cb RPCRT4!OSF_CCALL__FastSendReceive+0x165
0068fda8 4ef9a7f8 RPCRT4!OSF_CCALL__SendReceiveHelper+0xed
0068fdd4 4ef946a7 RPCRT4!OSF_CCALL__SendReceive+0x37
0068fdf0 4efd56b3 RPCRT4!I_RpcSendReceive+0xc4
0068fe08 01002850 RPCRT4!NdrSendReceive+0x4f
0068ff40 01001f32 rtclnt+0x2850
0068ffb4 77e92ca8 rtclnt+0x1f32
0068ffec 00000000 KERNEL32!CreateFileA+0x11b
```

下面介绍如何解决此问题。

逐步排除调用阻塞问题

1. 确保调试器正在调试拥有阻塞单元的进程。 （这是包含疑似在 RPC 中挂起的客户端线程的进程。）

2. 获取此线程的堆栈指针。 堆栈将类似于前述示例中所示的堆栈。 在此示例中，堆栈指针为 0x0068FBA0。

3. 获取此线程的调用信息。 为此，请使用带有线程堆栈指针的 !rpcexts.rpcreadstack 扩展作为其参数，如下所示：

dbgcmd

```
0:001> !rpcexts.rpcreadstack 68fba0
    CallID: 1
    IfStart: 19bb5061
    ProcNum: 0
            Protocol Sequence:      "ncacn_ip_tcp"  (Address: 00692ED8)
            NetworkAddress: ""      (Address: 00692F38)
            Endpoint:       "1120"  (Address: 00693988)
```

此处显示的信息将允许您跟踪调用。

4. 网络地址为空，表示本机。 终结点为 1120。 需要确定托管此终结点的进程。 未完成此操作，可以将此终结点编号传递给 !rpcexts.getendpointinfo 扩展，如下所示：

dbgcmd

```
    0:001> !rpcexts.getendpointinfo 1120
    Searching for endpoint info ...
    PID  CELL ID   ST PROTSEQ        ENDPOINT
    -----------------------------------------
    0278 0000.0001 01            TCP 1120
```

5. 从上述信息可以看出，该进程 0x278 包含此终结点。 可以使用 !rpcexts.getcallinfo 扩展来确定此进程是否知道有关此调用的任何内容。 此扩展需要四个参数：*CallID*、*IfStart* 和 *ProcNum*（可在步骤 3 中找到），以及 0x278 的 *ProcessID*。

dbgcmd

```
    0:001> !rpcexts.getcallinfo 1 19bb5061 0 278
    Searching for call info ...
    PID  CELL ID   ST PNO IFSTART  TIDNUMBER CALLFLAG CALLID   LASTTIME
CONN/CLN
    ------------------------------------------------------------------------
----
    0278 0000.0004 02 000 19bb5061 0000.0002 00000001 00000001 00072c09
0000.0003
```

6. 步骤 5 中的信息非常有用，但有些简略。 第二列中的单元格 ID 为 0000.0004。 如果将进程 ID 和此单元格号传递给 !rpcexts.getdbgcell 扩展，您将看到此单元格的更清晰显示：

dbgcmd

```
    0:001> !rpcexts.getdbgcell 278 0.4
    Getting cell info ...
```

```
    Call
    Status: Dispatched
    Procedure Number: 0
    Interface UUID start (first DWORD only): 19BB5061
    Call ID: 0x1 (1)
    Servicing thread identifier: 0x0.2
    Call Flags: cached
    Last update time (in seconds since boot):470.25 (0x1D6.19)
    Owning connection identifier: 0x0.3
```

这表明调用处于"已调度"状态，意味着已离开 RPC 运行时。 上次更新时间为 470.25。 可以使用 !rpcexts.rpctime 扩展了解当前时间：

dbgcmd

```
    0:001> !rpcexts.rpctime
    Current time is: 6003, 422
```

这表明，该通话的最后一次联系大约为 5533 秒之前，即大约 92 分钟前。 因此，这一定是调用阻塞。

7. 在将调试器附加到服务器进程之前的最后一步，可以使用服务线程标识符隔离当前应为调用提供服务的线程。 这是另一个单元格号；它在步骤 6 中显示为"0x0.2"。 可以按如下所示使用它：

dbgcmd

```
    0:001> !rpcexts.getdbgcell 278 0.2
    Getting cell info ...
    Thread
    Status: Dispatched
    Thread ID: 0x1A4 (420)
    Last update time (in seconds since boot):470.25 (0x1D6.19)
```

现在，您知道当前正在进程 0x278 中查找线程 0x1A4。

线程可能正在进行另一个 RPC 调用。 如有必要，可以通过重复此过程来跟踪此调用。

注意此过程演示在已知客户端线程的情况下如何查找服务器线程。 有关反向技术的示例，请参阅从服务器线程识别调用方。

跟踪服务器进程中的争用

项目 • 2023/06/15

为了为传入请求提供服务，RPC 将维护一组工作线程。 理想情况下，线程数会很小。 但是，这种理想情况仅在仔细优化服务器管理器例程的实验室环境中出现。 在实际情况下，线程数将因服务器工作负荷而异，但线程数可以是 1 到 50。

如果工作线程数超过 50，则服务器进程中可能存在过多争用。 导致这种情况的常见原因是错误地使用堆、内存压力或通过单个关键部分序列化服务器中的大多数活动。

若要查看给定服务器进程中的线程数，请使用 **！ rpcexts.getthreadinfo** 扩展，或将 DbgRpc 与 **-t** 开关配合使用。 在以下示例中提供进程 ID (，0xC4)：

控制台

```
D:\wmsg>dbgrpc -t -P c4
Searching for thread info ...
## PID  CELL ID   ST TID      LASTTIME
-----------------------------------
00c4 0000.0004 03 0000011c 000f164f
00c4 0000.0007 03 00000120 008a6290
00c4 0000.0015 03 0000018c 008a6236
00c4 0000.0026 03 00000264 0005c443
00c4 0000.002d 03 00000268 000265bb
00c4 0000.0030 03 0000026c 000f1d32
00c4 0000.0034 03 00000388 007251e9
```

在这种情况下，只有七个工作线程，这是合理的。

如果线程超过 100 个，则应将调试器附加到此进程，并调查原因。

注意 远程运行 **dbgrpc -t** 等查询对服务器和网络来说成本高昂。 如果在脚本中使用此查询，应确保此命令不会过于频繁地运行。

检查线程阻塞问题

项目 • 2024/03/04

RPC 需要其工作线程可用才能正常执行。 一个常见的问题是，同一进程中的某些组件在保存某个全局关键部分（例如，加载程序锁或堆锁）时会发生死锁。 这将导致许多线程挂起，很可能包括一些 RPC 工作线程。

如果发生这种情况，RPC 服务器将不会对外部世界做出响应。 RPC 调用它将返回 RPC_S_SERVER_UNAVAILABLE 或 RPC_S_SERVER_TOO_BUSY。

如果有故障的驱动程序阻止 IRP 完成并访问 RPC 服务器，也可能导致类似的问题。

如果怀疑其中一个问题可能发生，请将 DbgRpc 与 **-t** 开关一起使用（或使用 **!rpcexts.getthreadinfo** 扩展）。 进程 ID 应用作参数。 在以下示例中，假设进程 ID 为 0xC4：

dbgcmd

```
D:\wmsg>dbgrpc -t -P c4
Searching for thread info ...
## PID  CELL ID   ST TID      LASTTIME
-----------------------------------
00c4 0000.0004 03 0000011c 000f164f
00c4 0000.0007 03 00000120 008a6290
00c4 0000.0015 03 0000018c 008a6236
00c4 0000.0026 03 00000264 0005c443
00c4 0000.002d 03 00000268 000265bb
00c4 0000.0030 03 0000026c 000f1d32
00c4 0000.0034 03 00000388 007251e9
```

TID 列为每个线程提供线程 ID。 LASTTIME 列包含每个线程上次更改状态的时间戳。

每当服务器收到请求时，至少有一个线程将更改状态，并且其时间戳将更新。 因此，如果向服务器发出 RPC 请求，但请求失败，但时间戳都没有更改，则表明该请求实际上没有达到 RPC 运行时。 应调查原因。

识别服务器线程中的调用方

项目 • 2024/02/06

可以确定给定 RPC 调用的内容，即使你拥有的唯一信息是服务调用的服务器线程。

这非常有用 -- 例如，找出谁向 RPC 调用传递了无效参数。

根据此特定调用使用的协议序列，可以获取不同程度的详细信息。 某些协议（（例如 NetBios））根本没有此信息。

从服务器线程标识调用方

1. 启动以服务器线程为目标的用户模式调试器。

2. 使用 **| (进程状态)** 命令获取进程 ID：

   ```
   dbgcmd

   0:001> |
     0     id: 3d4 name: rtsvr.exe
   ```

3. 使用 **！ rpcexts.getcallinfo** 扩展获取此进程中的活动调用。 (有关语法的说明，请参阅参考页。) 需要提供0x3D4的进程 ID：

   ```
   dbgcmd

   0:001> !rpcexts.getcallinfo 0 0 FFFF 3d4
   Searching for call info ...
   PID  CELL ID   ST PNO IFSTART  THRDCELL  CALLFLAG CALLID   LASTTIME
   CONN/CLN
   ------------------------------------------------------------------------
   -----
   03d4 0000.0004 02 000 19bb5061 0000.0002 00000001 00000001 00a1aced
   0000.0003
   ```

   查找状态为 02 或 01 的呼叫 (已调度或活动) 。 在此示例中，进程只有一个调用。 如果有更多，则必须将 **！ rpcexts.getdbgcell** 扩展与 THRDCELL 列中的单元格号一起使用。 这将允许你检查线程 ID，以便确定你感兴趣的调用。

4. 知道你感兴趣的呼叫后，请查看 CONN/CLN 列中的单元格编号。 这是连接对象的单元格 ID。 在本例中，单元格编号为 0000.0003。 将此单元格号和进程 ID 传递给 **！ rpcexts.getdbgcell**：

   ```
   dbgcmd
   ```

```
0:001> !rpcexts.getdbgcell 3d4 0.3
Getting cell info ...
Connection
Connection flags: Exclusive
Authentication Level: Default
Authentication Service: None
Last Transmit Fragment Size: 24 (0x6F56D)
Endpoint for the connection: 0x0.1
Last send time (in seconds since boot):10595.565 (0x2963.235)
Last receive time (in seconds since boot):10595.565 (0x2963.235)
Getting endpoint info ...
Process object for caller is 0xFF9DF5F0
```

此扩展将显示有关此连接的客户端的所有信息。 实际信息的数量会有所不同，具体取决于所使用的传输。

在此示例中，本地命名管道用作传输，并显示调用方的进程对象地址。 如果附加内核调试器 (或启动本地内核调试器) ，则可以使用 **！ process** 扩展来解释此进程地址。

如果使用 LRPC 作为传输，则会显示调用方的进程 ID 和线程 ID。

如果将 TCP 用作传输，则会显示调用方 IP 地址。

如果将远程命名管道用作传输，则不会提供任何信息。

注意 前面的示例演示了如果知道服务器线程，如何查找客户端线程。 有关反向技术的示例，请参阅 分析卡住呼叫问题。

RPC 状态信息的内部结构

项目 • 2023/06/15

本部分提供了 RPC 运行时收集的状态信息的内部结构的详细信息。

所有 RPC 运行时状态信息都包含在单元格中。 单元格是可以单独查看和更新的最小信息单位。

RPC Run-Time中的每个键对象都将保留一个或多个有关其状态的信息单元格。 每个单元格都有一个单元格 ID。 当对象引用另一个对象时，它通过指定该对象的单元格 ID 来执行此操作。 RPC Run-Time可以维护相关信息的关键对象包括终结点、线程、连接对象、服务器调用 (SCALL) 对象和客户端调用 (CCALL) 对象。

RPC 服务器运行时，RPC Run-Time使用一个或多个工作线程侦听一组终结点。 每当将数据传输到服务器时，线程会选取数据并确定传入请求是什么。 如果请求是创建连接，则创建 Connection 对象，然后为连接上的所有调用提供服务。 当对连接进行 RPC 调用时，Connection 对象实例化与客户端调用 (CCALL) 对象对应的服务器调用 (SCALL) 对象。 然后，此服务器调用对象处理此特定调用。

当 RPC 客户端运行时，每次调用时，RPC Run-Time都会创建一个客户端调用对象。 此客户端调用对象包含有关此特定调用的信息。

终结点单元格

从 RPC 运行时的角度来看，终结点是可以联系特定服务器的入口点。 终结点始终与给定的 RPC 传输相关联。 终结点状态信息用于将客户端调用与服务器上的特定进程相关联。

终结点单元格中的字段为：

ProtseqType
此终结点的协议序列。

地位
状态值： *已分配*、 *活动*或 *非活动*。 大多数终结点处于活动状态。 创建过程开始时，终结点已 *分配* 状态，但尚未完成。 如果终结点不再处于 *非活动状态* ，（例如，）卸载协议时。

EndpointName
终结点名称的前 28 个字符。

线程单元格

服务器线程是可供 RPC) 使用的标准 Win32 线程 (工作线程。

线程单元格中的字段为：

地位
状态值： *正在处理*、 *已调度*、 *已分配*或 *空闲*。 *处理*线程是Run-Time中正在处理信息的线程。 *已调度*线程已将名为) 的 (调度到服务器提供的管理器例程 (通常只称为*服务器例程*) 。 已缓存 *已分配* 的线程。 *空闲*线程可用于服务请求。

LastUpdateTime
启动后 () 上次更新信息的时间（以毫秒为单位）。

TID
此线程的线程 ID。 尝试与调试器中的线程列表关联时，这很有用。

连接对象单元格

连接对象单元格中的字段为：

标志
标志值包括*独占非独占/*、*身份验证级别*和*身份验证服务*。

LastTransmitFragmentSize
通过连接传输的最后一个片段的大小。

端点
从中选取此连接的终结点的单元格 ID。

LastSendTime
上次在连接上发送数据的时间。

LastReceiveTime
上次在连接上接收数据的时间。

服务器调用对象单元格

服务器调用 (SCALL) 对象单元格中的字段为：

地位
状态值： *已分配*、 *活动*或 *已调度*。 *分配的*调用处于非活动状态并已缓存。 当调用 *处于活动状态*时，RPC Run-Time正在处理与此调用相关的信息。 *调度*调用时，管理器例程 (服务器例程) 已调用，尚未返回。

ProcNum
此调用) netmon 捕获文件中的过程编号 (操作号。 RPC Run-Time通过按 IDL 文件中的位置对各个例程进行编号来标识接口中的单个例程。 接口中的第一个例程为数字 0，第二个例程为数字 1，依此依序。

InterfaceUUIDStart
接口 UUID 的第一个 DWORD。

ServicingTID
为此调用提供服务的线程的单元格 ID。 如果调用未 *处于活动状态* 或 *未调度*，则包含过时的信息。

CallFlags
这些标志值指示这是否是独占连接中的缓存调用、是否为异步调用、是否为管道调用，以及这是 LRPC 还是 OSF 调用。

LastUpdateTime
启动后 () 上次更新调用对象状态信息的时间（以毫秒为单位）。

Pid
调用方的进程 ID。 仅对 LRPC 调用有效。

TID
调用方线程 ID。 仅对 LRPC 调用有效。

客户端调用对象单元格

客户端呼叫 (CCALL) 对象分为两个单元格，因为有关客户端调用的信息太大，无法容纳在一个单元格中。 第一个单元格称为 *“客户端呼叫信息”*，第二个单元格称为 *“呼叫目标信息”*。 大多数工具都会一起显示信息，因此无需区分它们。

除非收集完整状态信息，否则不会维护有关客户端调用的信息。 此规则有一个例外：即使仅收集服务器状态信息，也会保留有关服务器调用中客户端调用的信息。 这允许跟踪跨多个跃点的调用。

“客户端呼叫信息”单元格中的字段为：

ProcNum
netmon 捕获文件中的过程编号 (操作编号，) 所调用的方法。 RPC Run-Time通过按 IDL 文件中的位置对各个例程进行编号来标识接口中的单个例程。 接口中的第一个例程为数字 0，第二个例程为数字 1，依此依序。

ServicingThread
进行此调用的线程的单元格 ID。

IfStart
进行调用的接口 UUID 的第一个 DWORD。

端点
进行调用的服务器上的终结点的前 12 个字符。

“呼叫目标信息”单元格中的字段为：

ProtocolSequence
此调用的协议序列。

LastUpdateTime
启动后的时间 (毫秒，) 更新有关客户端调用或调用目标的信息。

TargetServer
进行调用的服务器名称的前 24 个字符。

ACPI 调试

项目 • 2024/03/20

本节包括：

AMLI 调试器

其他 ACPI 调试扩展

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

AMLI 调试器

项目 • 2023/03/08

本节包括：

AMLI 调试器简介

设置 AML 调试会话

基本 AML 调试

使用 AMLI 调试器扩展

使用 AMLI 调试器命令

AML 调试示例

AMLI 调试器简介

项目 • 2023/06/15

调试标准内核模式代码与调试 ACPI (高级配置和电源接口) BIOS 之间存在显著差异。

Windows 及其驱动程序由为特定处理器编译的二进制计算机代码组成，而 ACPI BIOS 的核心不在计算机代码中。 相反，它存储为 ACPI 计算机语言 (AML)，并由 Microsoft AML 解释器在运行时进行处理。

Microsoft AMLI 调试器是一组可以调试 AML 代码的特殊调试工具。

在 Windows 10 之前的 Windows 版本中，版本 1803 检查了 Windows ACPI 驱动程序 (Acpi.sys) 的内部版本。 这种情况不再存在，因为不再提供已检查的内部版本。

AMLI 调试器完全可识别 64 位。 无论目标计算机或主计算机使用什么处理器，AMLI 调试器都将正常运行。

设置 AML 调试会话

项目 • 2023/06/15

AMLI 调试器代码包含在 Acpi.sys 中。 若要完全执行 AML 调试，必须在目标计算机上安装此驱动程序，它通常就是这样。

若要在免费版本上激活中断调试器，请使用 **！ amli set dbgbrkon** 命令，该命令是 AMLI 调试器扩展的一部分。 有关详细信息，请参阅 **！ amli set**。

另请参阅

AMLI 调试器

基本 AML 调试

项目 • 2024/07/08

AMLI 调试器支持两种的专用命令：*AMLI 调试器扩展*和 *AMLI 调试器命令*。

执行 AML 调试时，应仔细区分“调试器命令”窗口出现的两种不同提示：

- 看到 **kd>** 提示符时，说明您在控制内核调试器。 所有标准内核调试器命令和扩展都可用。 此外，AMLI 调试器扩展也可用。 这些扩展的语法为 **！amli** *命令*。 AMLI 调试器命令在此模式下不可用。

- 当看到 **AMLI(? for help)->** 提示时，表明您正在控制 AMLI 调试器。 （使用 WinDbg 时，此提示将显示在调试器命令窗口的顶部窗格中，同时**输入>** 提示将显示在底部窗格中。）在此提示下，可以输入任何 AMLI 调试器命令。 还可以输入任何 AMLI 调试器扩展；这些扩展不应将 **!amli** 作为前缀。 标准内核调试命令在此模式下不可用。

- 如果看不到任何提示，表示目标计算机正在运行。

在任何调试会话开始时，应使用 **!amli set** 扩展设置 AMLI 调试器选项。 **verboseon**、**traceon** 和 **errbrkon** 选项也非常有用。 应考虑激活 **spewon** 选项。 有关详细信息，请参阅扩展参考页。

AMLI 调试器有多种激活方式：

- 如果遇到 AML 代码中的断点，ACPI 将进入 AMLI 调试器。

- 如果 AML 代码（如 **int 3**）中出现严重错误或异常，ACPI 将进入 AMLI 调试器。

- 如果设置了 **errbrkon** 选项，任何 AML 错误都会导致 ACPI 进入 AMLI 调试器。

- 如果要故意进入 AMLI 调试器，请使用 **!amli debugger**扩展，然后使用 **g (Go)** 命令。 下次由解释器执行任何 AML 代码时，AMLI 调试器将接管。

在 AMLI 调试器提示符下，可以键入 **q** 以返回到内核调试器，或键入 **g** 以恢复正常执行。

以下扩展对 AML 调试特别有用：

- **!amli dns** 扩展显示特定对象的 ACPI 命名空间、属于该对象的命名空间树，甚至显示整个命名空间树。 此命令在确定特定命名空间对象是什么时特别有用，无论是方法、字段单元、设备还是其他类型的对象。

- **!amli find** 扩展采用任何命名空间对象的名称并返回其完整路径。

- **!amli u** 扩展取消汇编 AML 代码。
- **!amli lc** 扩展显示有关所有活动 ACPI 上下文的简短信息。
- **!amli r** 扩展显示有关解释器的当前上下文的详细信息。 这在检测到错误后出现 AMLI 调试器提示时非常有用。
- 断点可以在 AML 代码中设置和控制。 使用 **!amli bp** 设置断点，使用**!amli bc** 清除断点，使用**!amli bd** 禁用断点，使用**!amli be** 重新启用断点，使用**!amli bl** 列出所有断点。
- AMLI 调试器可以通过 AML 代码运行、单步执行和跟踪。 使用 **run**、**p** 和 **t** 命令执行这些操作。

有关扩展和命令的完整列表，请参阅使用 AMLI 调试器扩展 和使用 AMLI 调试器命令。

另请参阅

AMLI 调试器

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用 AMLI 调试器扩展

项目 • 2023/10/26

AMLI 调试器扩展命令包含在扩展模块 Kdexts.dll 中，并使用以下语法：

```
dbgcmd
```

```
kd> !amli command [parameters]
```

与任何扩展模块一样，加载扩展模块后，可以省略 **acpikd** 前缀。

如果在 AMLI 调试器提示符处，只需输入 *命令名称而不* 带 **！ amli** 前缀，即可执行其中任何扩展命令：

```
dbgcmd
```

```
AMLI(? for help)-> command [parameters]
```

在此提示符下，（**！ amli 调试器** 命令不可用，因为它) 毫无意义。 此外，在此提示符下，help 命令（**？** ）将显示所有 AMLI 调试器扩展和命令，而 **！ amli?** 扩展仅显示有关实际扩展的帮助。

操作	**扩展命令**
显示帮助	!amli ?
设置 AML 断点	!amli bp
列出 AML 断点	!amli bl
禁用 AML 断点	!amli bd
启用 AML 断点	!amli be
清除 AML 断点	!amli bc
输入 AMLI 调试器	!amli debugger
显示事件日志	!amli dl
清除事件日志	!amli cl
显示堆	!amli dh
显示数据对象	!amli do
显示堆栈	!amli ds

操作	扩展命令
显示命名空间对象	!amli dns
查找命名空间对象	!amli find
显示最近的方法	!amli ln
列出所有上下文	!amli lc
显示上下文信息	!amli r
取消组合 AML 代码	!amli u
设置 AMLI 调试器选项	!amli set

另请参阅

AMLI 调试器

使用 AMLI 调试器命令

项目 • 2024/07/08

可以从 AMLI 调试器提示符发出以下命令。

展开表

常规类别	特定操作	AMLI 调试器命令
控制调试器	继续执行中断到内核调试器	**gq**
控制 AML 执行	将方法单步执行 AML 代码跟踪到 AML 代码中	**运行pt**
控制跟踪模式设置	配置跟踪模式	**trace**
通知命名空间对象	通知命名空间对象	**通知**
显示对象计数表	显示对象计数表	**dc**
访问内存	显示数据显示数据字节显示数据字词显示数据 DWORD 显示数据字符串编辑内存	**ddbdwddda e**
访问端口	从端口读取 Word 的字节从端口读取 DWORD 从端口写入字节到端口写入 Word 到端口写入 DWORD 到端口	**iwidoow od**
显示帮助	显示帮助	**?**

控制调试器

这些命令退出 AMLI 调试器。 **g** 命令将恢复目标计算机的正常执行，**q** 命令将冻结目标计算机并进入内核调试器。

g

q

控制 AML 执行

这些命令允许你运行或单步执行 AML 方法。 **run** 命令将在指定点开始执行。 **p** 和 **t** 命令允许一次单步执行一个指令。 如果遇到函数调用， **p** 命令会将函数视为单个步骤，而 **t** 命令一次跟踪到新函数中一个指令。

run MethodName ***[ArgumentList]***

运行 *CodeAddress* [***ArgumentList***]

p

t

MethodName
指定方法的完整路径和名称。 执行将从此方法的内存位置的开头开始。

CodeAddress
指定要开始执行的地址。

ArgumentList
指定要传递给方法的参数的列表。 每个参数必须是一个整数。 应使用空格分隔多个参数。

控制跟踪模式设置

跟踪命令控制 AML 解释器的跟踪模式设置。 如果此命令没有参数使用，则会显示当前跟踪模式设置。

trace [trigon|trigoff] [level=*Level*] [**add**=*TPStrings*] [**zap**=*TPNumbers]*

三角形
激活跟踪触发器模式。

trigoff
停用跟踪触发器模式。

Level
指定跟踪级别的新设置。

TPStrings
指定要添加的一个或多个触发器点。 每个触发器点都按名称指定。 多个触发器点字符串应用逗号分隔。

TPNumbers
指定要删除的一个或多个触发器点。 每个触发器点都按数字指定。 多个触发点数应用逗号分隔。 若要查看触发器点编号的列表，请使用不带参数的 **跟踪** 命令。

通知命名空间对象

notify 命令将通知发送到 ACPI 命名空间对象。 通知将放置在指定对象的队列中。

notify *ObjectName 值*

notify *ObjectAddress 值*

ObjectName
指定要通知的对象的完整命名空间路径。

ObjectAddress
指定要通知的对象地址。

值
指定通知值。

显示对象计数表

dc 命令显示内存对象计数表。

dc

访问内存

内存访问命令允许读取和写入内存。 读取内存时，可以使用 db**、dw、dd 或 da 命令选择内存单位**的大小。 简单的 **d** 命令在最近选择的单元中显示内存。 如果这是使用的第一个显示命令，则使用字节单位。

如果未指定地址或方法，则显示将开始上一个显示命令结束的位置。

这些命令与标准内核调试器内存命令的效果相同;它们在 AMLI 调试器中重复，以便轻松访问。

d[b|w|d|a] [[**l**=*Length*] [*方法* | [%%]*地址*]]

e [%%]*Address Datalist*

b
指定数据应以字节单位显示。

w
指定数据应以单词（16 位）单位显示。

d
指定数据应以 DWORD（32 位）单位显示。

a
指定数据应显示为字符串。 数据显示为 ASCII 字符。 当读取 NULL 字符或显示长度*字符*

时，显示将终止。

时长
指定要显示的字节数。 *长度* 必须是十六进制数字（没有 **0x** 前缀）。 如果 *省略 Length* ，则默认显示大小为0x80字节。

方法
指定方法的完整路径和名称。 显示将从此方法的内存位置的开头开始。

Address
指定读取或写入开始的内存地址。 如果地址的前缀为两个百分点（%%），则会将其解释为物理地址。 否则，将其解释为虚拟地址。

DataList
指定要写入内存的数据。 列表中的每个项可以是十六进制字节或字符串。 使用字符串时，必须用引号括起来。 多个项应用空格分隔。

访问端口

端口命令允许你从数据端口发送输出或接收输入。 i 和 o 命令传输单个字节，**iw** 和 **ow** 命令传输单词（16 位），ID 和 **od** 命令传输 DWORDS （32 位）。

这些命令与标准内核调试器端口命令的效果相同;它们在 AMLI 调试器中重复，以便轻松访问。

i *端口*

iw *端口*

id *端口*

o *Port DataForPort*

ow *Port DataForPort*

od *Port DataForPort*

端口
指定要访问的端口的地址。 端口大小必须与所选命令匹配。

DataForPort
指定要写入端口的数据。 此数据的大小必须与所选命令匹配。

显示帮助

此命令显示 AMLI 调试器命令的帮助文本。

? [*命令*]

命令
指定要为其显示帮助的命令。 如果省略此项，将显示所有 AMLI 调试器命令和 AMLI 调试器扩展的列表。

另请参阅

AMLI 调试器

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

AML 调试示例

项目 • 2024/02/14

下面是演示如何开始使用 AML 调试的示例。

调查冻结的计算机

如果目标计算机已冻结，并且你怀疑这可能是 ACPI 问题，请首先使用 **！ amli lc** 扩展显示所有活动上下文：

dbgcmd

```
kd> !amli lc
*Ctxt=ffffffff8128d000, ThID=ffffffff81277880, Flgs=----R----,
pbOp=ffffffff8124206c, Obj=\_SB.PCI0.ISA0.FDC0._CRS
```

如果未显示任何上下文，则错误可能与 ACPI 无关。

如果显示上下文，请查找用星号标记的上下文。 这是 *当前上下文* （当前正在由解释器执行）。

在此示例中，目标计算机在 32 位处理器上运行 Windows。 因此，所有地址都转换为 64 位，在高 32 位中生成无端的 FFFFFFFF。 缩写 **pbOp** 指示指令指针（“指向二进制操作代码的指针”）。 **Obj** 字段提供在 ACPI 表中显示的方法的完整路径和名称。 有关标志的说明，请参阅 ！ amli lc。

可以使用 **！ amli u** 命令来反汇编_CRS方法，如下所示：

dbgcmd

```
kd> !amli u \_SB.PCI0.ISA0.FDC0._CRS

ffffffff80e4a535 : CreateDWordFieldCRES, 0x76, RAMT)
ffffffff80e4a540 : CreateDWordField(CRES, 0x82, PCIT)
ffffffff80e4a54b : Add(MLEN(), 0x100000, RAMT)
ffffffff80e4a559 : Subtract(0xffe00000, RAMT, PCIT)
ffffffff80e4a567 : Return(CRES)
```

闯入 AMLI 调试器

！ amli 调试器命令会导致 AML 解释器下次执行任何 AML 代码时闯入 AMLI 调试器。

出现 AMLI 调试器提示后，可以使用任何 AMLI 调试器命令。 还可以使用 **！amli** 扩展命令，而无需为它们添加前缀“！amli”：

dbgcmd

```
kd> !amli debugger
kd> g

AMLI(? for help)-> find _crs
\_SB.LNKA._CRS
\_SB.LNKB._CRS
\_SB.LNKC._CRS
\_SB.LNKD._CRS
\_SB.PCI0._CRS
\_SB.PCI0.LPC.NCP._CRS
\_SB.PCI0.LPC.PIC._CRS
\_SB.PCI0.LPC.TIME._CRS
\_SB.PCI0.LPC.IDMA._CRS
\_SB.PCI0.LPC.RTC._CRS
\_SB.PCI0.LPC.SPKR._CRS
\_SB.PCI0.LPC.FHUB._CRS
\_SB.PCI0.SBD1._CRS
\_SB.PCI0.SBD2._CRS
\_SB.MBRD._CRS

AMLI(? for help)-> u \_SB.PCI0._CRS

ffffffff80e4a535 : CreateDWordFieldCRES, 0x76, RAMT)
ffffffff80e4a540 : CreateDWordField(CRES, 0x82, PCIT)
ffffffff80e4a54b : Add(MLEN(), 0x100000, RAMT)
ffffffff80e4a559 : Subtract(0xffe00000, RAMT, PCIT)
ffffffff80e4a567 : Return(CRES)
```

使用断点

在以下示例中，在执行方法_BST之前，将进入 AMLI 调试器。

即使已找到_BST对象，也应验证它是否确实是一种方法。 可以使用 **！amli dns** 扩展执行此操作。

dbgcmd

```
kd> !amli dns /s \_sb.pci0.isa.bat1._bst

ACPI Name Space: \_SB.PCI0.ISA.BAT1._BST (c29c2044)
Method(_BST:Flags=0x0,CodeBuff=c29c20a5,Len=103)
```

现在可以使用 **！amli bp** 命令来放置断点：

dbgcmd

```
kd> !amli bp \_sb.pci0.isa.bat1._bst
```

可能还需要在方法中放置断点。 可以使用 **！ amli u** 命令来反汇编_BST，然后在其中一个步骤中放置断点：

dbgcmd

```
kd> !amli u _sb.pci0.isa.bat1._bst

ffffffffc29c20a5: Acquire(\_SB_.PCI0.ISA_.EC0_.MUT1, 0xffff)
ffffffffc29c20c0: Store("CMBatt - _BST.BAT1", Debug)
ffffffffc29c20d7: \_SB_.PCI0.ISA_.EC0_.CPOL()
ffffffffc29c20ee: Release(\_SB_.PCI0.ISA_.EC0_.MUT1)
ffffffffc29c2107: Return(PBST)

kd> !amli bp c29c20ee
```

响应触发的断点

在以下示例中，_WAK的方法正在运行，然后遇到断点：

dbgcmd

```
Running \_WAK method
Hit Breakpoint 0.
```

使用！ amli ln 扩展查看当前程序计数器最近的方法。 以下示例以 32 位格式显示地址：

dbgcmd

```
kd> !amli ln
c29accf5: \_WAK
```

！ amli lc 扩展显示所有活动上下文：

dbgcmd

```
kd> !amli lc
 Ctxt=c18b6000, ThID=00000000, Flgs=A-QC-W----, pbOp=c29bf8fe,
Obj=\_SB.PCI0.ISA.EC0._Q09
*Ctxt=c18b4000, ThID=c15a6618, Flgs=----R-----, pbOp=c29accf5, Obj=\_WAK
```

这表明活动上下文与方法_Q09和_WAK相关联。 当前上下文_WAK。

现在可以使用 **！ amli r** 命令显示有关当前上下文的更多详细信息。 从中可以看到有用的线程和堆栈信息，以及传递给_WAK和本地数据对象的参数。

dbgcmd

```
kd> !amli r
Context=c18b4000*, Queue=00000000, ResList=00000000
ThreadID=c15a6618, Flags=00000010
StackTop=c18b5eec, UsedStackSize=276 bytes, FreeStackSize=7636 bytes
LocalHeap=c18b40c0, CurrentHeap=c18b40c0, UsedHeapSize=88 bytes
Object=\_WAK, Scope=\_WAK, ObjectOwner=c18b4108, SyncLevel=0
AsyncCallBack=ff06b5d0, CallBackData=0, CallBackContext=c99efddc

MethodObject=\_WAK
c18b40e4: Arg0=Integer(:Value=0x00000001[1])
c18b5f3c: Local0=Unknown()
c18b5f54: Local1=Unknown()
c18b5f6c: Local2=Unknown()
c18b5f84: Local3=Unknown()
c18b5f9c: Local4=Unknown()
c18b5fb4: Local5=Unknown()
c18b5fcc: Local6=Unknown()
c18b5fe4: Local7=Unknown()
c18b4040: RetObj=Unknown()
```

跟踪、单步执行和运行 AML 代码

如果要通过代码进行跟踪，可以使用！ amli 设置**扩展打开完整跟踪信息**，如下所示：

dbgcmd

```
kd> !amli set spewon verboseon traceon
```

现在，可以逐行执行 AML 代码，并逐行观看代码执行。 **p** 命令会逐步执行任何函数调用。 **t** 命令将单步执行函数调用。

dbgcmd

```
AMLI(? for help)-> p

c29bfcb7: Store(\_SB_.PCI0.ISA_.ACAD.CHAC(SEL0=0x10e1)
c29c17b1: {
c29c17b1: |
Store(LGreater(And(Arg0=0x10e1,0xf0,)=0xe0,0x80)=0xffffffff,Local0)=0xffffff
ff

AMLI(? for help)-> p

c29c17bb: |
```

```
If(LNot(LEqual(Local0=0xffffffff,ACP_=0xffffffff)=0xffffffff)=0x0)
c29c17ce: | {
c29c17ce: | | Return(Zero)
c29c17d0: | }
c29c17d0: },Local1)=0x0

AMLI(? for help)-> t

c29bfcd4: Store(\_SB_.PCI0.ISA_.BAT1.CHBP(SEL0=0x10e1)
c29c293d: {
c29c293d: | Store("CMBatt - CHBP.BAT1",Debug)String(:Str="CMBatt -
CHBP.BAT1")="CMBatt - CHBP.BAT1"
```

如果选择，还可以从 AMLI 调试器中运行方法。 例如，可以通过运行其控制方法_STA来评估 LNKA 设备的状态：

dbgcmd

```
AMLI(? for help)-> run \_sb.lnka._sta
PCI OpRegion Access on region c29b2268 device c29b2120

\_SB.LNKA._STA completed successfully with object data:
Integer(:Value=0x0000000b[11])
```

另请参阅

AMLI 调试器

其他 ACPI 调试扩展

项目 • 2023/10/26

以下扩展命令可用于调试高级配置和电源接口 (ACPI) BIOS 的问题：

- **！acpicache** 显示硬件应用程序层缓存的所有 ACPI 表 (HAL)
- **！acpiinf** 显示有关 ACPI 配置的信息
- **！acpiirqarb** 显示 ACPI IRQ 仲裁程序结构的内容
- **！facs** 显示固件 ACPI 控制结构
- **！fadt** 显示固定的 ACPI 说明表
- **！mapic** 显示 ACPI 多 APIC 表
- **！nsobj** 显示 ACPI 命名空间对象
- **！nstree** 显示 ACPI 命名空间树的一部分
- **！rsdt** 显示 ACPI 根系统说明表

有关 ACPI 相关扩展的完整列表，请参阅 **！acpikd.help**。

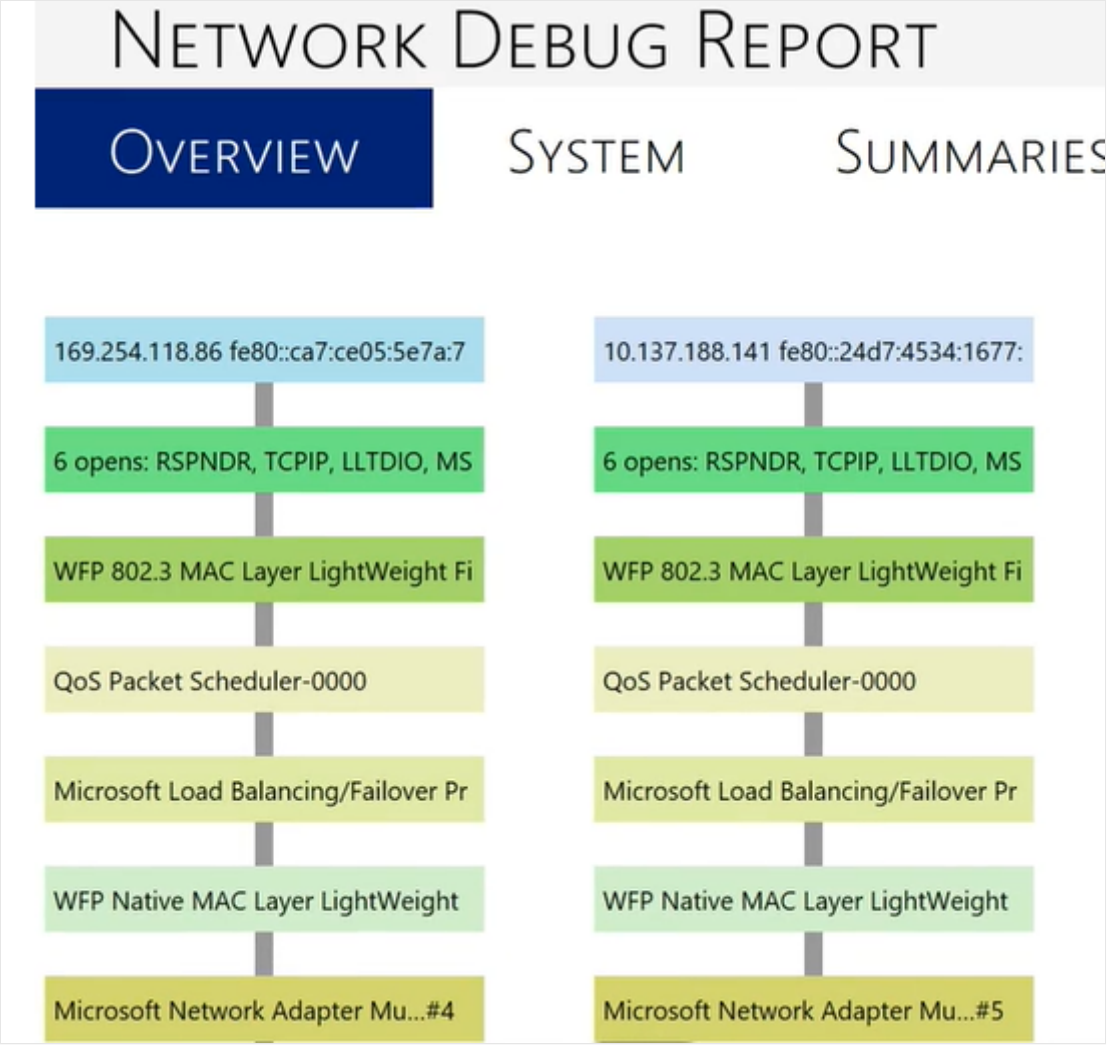
有关 **！amli***xxx* 扩展的详细信息，请参阅 AMLI 调试器。

NDIS 调试概述

项目 • 2023/11/15

用于调试网络驱动程序的两个主要工具是调试跟踪和网络驱动程序接口规范 (NDIS) 扩展。 有关调试跟踪的详细信息，请参阅 启用 NDIS 调试跟踪。 有关 NDIS 调试扩展的详细信息，请参阅 NDIS 扩展，其中提供了扩展模块 Ndiskd.dll 中找到的扩展命令的完整列表。

使用 ndiskd.netreport 命令生成显示当前适配器和协议的可视报表。



然后， ndiskd.netadapter 内核调试器命令是开始调查当前驱动程序集的好位置。

dbgconsole

```
1: kd> !ndiskd.netadapter
    Driver             NetAdapter          Name
    ffffdf8015a98380   ffffdf8015aa11a0    Microsoft ISATAP Adapter #2
    ffffdf801418d650   ffffdf80140c71a0    Microsoft Kernel Debug Network
Adapter
```

用于调试网络驱动程序的另一个工具是常规调试扩展的集合，这对于获取调试信息非常有用。 例如，输入 ！ stacks 2 ndis！ 将显示以 **ndis！** 开头的堆栈中的所有线程。 此信息可用于调试挂起和停止。 有关 WinDbg 入门的一般信息，请参阅 Windows 调试入门。

驱动程序验证程序

用于测试 NDIS 驱动程序的另一个有用工具是 NDIS 验证程序。 有关详细信息，请参阅 NDIS 驱动程序 和 静态驱动程序验证程序的规则。

NDIS 调试资源

碎片整理工具节目的第 175 集涵盖了 NDIS 调试 - 碎片整理工具 #175 - 调试网络堆栈。

NDIS 团队博客存档可在 NDIS 博客中找到。

NDIS Bug 检查

还有一个特定于 NDIS 的 bug 检查代码、bug 检查 0x7C (BUGCODE_NDIS_DRIVER) 。 有关其参数的完整列表，请参阅 bug 检查0x7C。

与 NDIS 行为错误相关的常见 bug 检查是 bug 检查0xD1：DRIVER_IRQL_NOT_LESS_OR_EQUAL，它可能是由驱动程序代码本身引起的。 这很可能是 bug 或内存损坏，最终表现为错误的指针。

另一个常见问题是 bug 检查0x9F：DRIVER_POWER_STATE_FAILURE。

所有 bug 检查的第一步是找到一个良好的转储文件，将其加载到 Windows 调试器中，并使用 ！ analyze 命令。 有关详细信息，请参阅 使用 ！ analyze 扩展。

启用 NDIS 调试跟踪

项目 • 2023/06/15

NDIS 调试跟踪是调试 NDIS 驱动程序的主要方法。 设置 NDIS 调试跟踪时，实际上是使用 NDIS 启用一个或多个级别的 DbgPrint 语句。 生成的信息足以调试大多数网络驱动程序问题。

通过设置注册表值启用 NDIS 调试跟踪

可以通过编辑注册表，在各种 NDIS 组件中启用不同级别的调试跟踪。 通常，应将以下条目和值添加到 **HKLM\SYSTEM\CurrentControlSet\Services\NDIS\Parameters** 注册表项：

text

```
"DebugLevel"=dword:00000000
"DebugSystems"=dword:000030F3
"DebugBreakPoint"=dword:00000001
```

DebugBreakPoint、**DebugLevel** 和 **DebugSystems** 可接受以下值：

DebugBreakPoint
控制 NDIS 驱动程序是否自动进入调试器。 如果此值设置为 1，则当驱动程序进入 Ndis.sys **DriverEntry** 函数时，NDIS 将中断到调试器中。

DebugLevel
使用 **DebugSystems** 值选择 NDIS 组件中的调试跟踪级别或数量。 以下值指定可以选择的级别：

级别	说明	值
DBG_LEVEL_INFO	所有可用的调试信息。 这是最高级别的跟踪。	0x00000000
DBG_LEVEL_LOG	日志信息。	0x00000800
DBG_LEVEL_WARN	警告。	0x00001000
DBG_LEVEL_ERR	“错误”。	0x00002000
DBG_LEVEL_FATAL	致命错误，这可能导致操作系统崩溃。 这是最低级别的跟踪。	0x00003000

DebugSystems

为指定的 NDIS 组件启用调试跟踪。 这对应于使用 **！ ndiskd.dbgsystems** 扩展。 以下值指定可以选择的 NDIS 组件：

组件	说明	值
DBG_COMP_INIT	处理适配器初始化。	0x00000001
DBG_COMP_CONFIG	处理适配器配置。	0x00000002
DBG_COMP_SEND	处理通过网络发送数据。	0x00000004
DBG_COMP_RECV	处理从网络接收数据。	0x00000008
DBG_COMP_PROTOCOL	处理协议操作。	0x00000010
DBG_COMP_BIND	处理绑定操作。	0x00000020
DBG_COMP_BUSINFO	处理总线查询。	0x00000040
DBG_COMP_REG	处理注册表操作。	0x00000080
DBG_COMP_MEMORY	处理内存管理。	0x00000100
DBG_COMP_FILTER	处理筛选器操作。	0x00000200
DBG_COMP_REQUEST	处理请求。	0x00000400
DBG_COMP_WORK_ITEM	处理工作项操作。	0x00000800
DBG_COMP_PNP	处理即插即用操作。	0x00001000
DBG_COMP_PM	处理电源管理操作。	0x00002000
DBG_COMP_OPENREF	处理打开引用对象的操作。	0x00004000
DBG_COMP_LOCKS	处理锁定操作。	0x00008000
DBG_COMP_RESET	处理重置操作。	0x00010000
DBG_COMP_WMI	处理 Windows Management Instrumentation 操作。	0x00020000
DBG_COMP_CO	处理 NDIS Connection-Oriented。	0x00040000
DBG_COMP_REF	处理引用操作。	0x00080000
DBG_COMP_ALL	处理所有 NDIS 组件。	0xFFFFFFFF

可以选择多个 NDIS 组件。 如果选择了多个组件，请将数据值与 OR 运算符合并。 例如，若要选择DBG_COMP_PNP、DBG_COMP_PM、DBG_COMP_INIT和

DBG_COMP_CONFIG，需要组合相应的值 (0x1000、0x2000、0x1和0x2) 来获取值 0x3003，然后在注册表中对其进行设置：

```text
"DebugSystems"=dword:00003003
```

每当更改调试跟踪的注册表值时，都必须重新启动计算机才能使新设置生效。

内核流调试

项目 • 2023/03/08

本节包括：

内核流调试概述

分析视频流停滞问题

分析捕获停滞问题

实时本地调试

无法加载的模块的图形分析

使用 !ks.graph

内核流调试概述

项目 • 2023/10/26

可以在扩展模块 Ks.dll 中找到内核流调试扩展。

可以使用 Ks.dll 来帮助调试 KS、AVStream、端口类和流类驱动程序。

有关 Ks.dll 中扩展命令的完整列表，请参阅 内核流式处理扩展 (Ks.dll) 。

如果希望将这些扩展与 Windows 2000 配合使用，必须将 Ks.dll 从 kdexts 目录复制到 w2kfre 目录。

分析视频流停滞问题

项目 • 2024/02/14

本部分介绍如何分析视频流停止。 引入独立硬件供应商（IHV）驱动程序 bug 后，所有示例输出都是从 AVStream 微型驱动程序生成的。

确定原因

调试处理停止

使用日志记录跟踪重要事件

解释 bug 检查0xCB

确定视频流停滞的原因

项目 • 2023/06/15

视频流停止有两个基本原因：

- **一个挂起。** 驱动程序未释放用户模式线程或内核模式线程。
- **停止。** 这是流式处理路径中的组件出现问题的结果。 一些可能的原因包括：
  - 捕获驱动程序未完成数据包。 在这种情况下，驱动程序组件或硬件可能是停止的根源。
  - 捕获驱动程序没有要完成的数据包。 在这种情况下，缓冲区可能会在编解码器或其他下游组件中停止。

如果可以重现问题，请此时附加调试器以确定实际原因。

确定问题是否为挂起

1. 将用户模式调试器附加到应用程序，并查找阻止的用户模式线程。
2. 确定应用程序是否响应。 图形可以暂停吗？ 图形是否可以停止？ 如果图形停止并重启，流式处理是否会重启？
3. 如果应用程序无响应，请尝试使用任务管理器结束任务。 如果此操作失败，则存在内核模式挂起。

确定问题是否为停滞

1. 确定样本在图中的位置。 这可以在本地或在内核模式调试会话中完成。
2. 确定样本是否在下游流动。 如果可以在 GraphEdit 中重现 bug，请在图形中放置一个中间筛选器以显示示例。
3. 确定是否正在调用处理例程。 这可以通过附加内核模式调试器并在此例程中设置断点来完成。

调试处理停滞问题

项目 • 2023/10/26

首先查找相关引脚。 在假设情况下，相关视频捕获引脚的地址为 **8160DDE0**，因此我们对此地址使用 **！ ks.dump** 扩展命令来获取更多详细信息：

dbgcmd

```
kd> !ks.dump 8160DDE0 7
Pin object 8160DDE0 [CKsPin = 8160DD50]
    DeviceState    KSSTATE_RUN
    ClientState    KSSTATE_RUN
    CKsPin object 8160DD50 [KSPIN = 8160DDE0]
        State                    KSSTATE_RUN
        Processing Mutex         8160DFD0 is not held
        And Gate &               8160DF88
        And Gate Count           1
```

首先，确定引脚是否处于适当的状态，以及处理互斥体是否由另一个线程持有。 在这种情况下，引脚 **状态KSSTATE_RUN，**并且未保留处理互斥体，因此我们接下来使用 **！ ks.dumpqueue** 扩展来确定是否有可用的帧：

dbgcmd

```
kd> !ks.dumpqueue 8160DDE0 7
Queue 8172D5D8:
 Frames Received  : 763
    Frames Waiting   : 5
...<this part of display not shown>...
Queue 8172D5D8:
 Frame Header 81B77E60:
        Irp = 816EE008
        Refcount = 1
    Frame Header 81A568D0:
        Irp = 816DE008
        Refcount = 0
    Frame Header 81844ED8:
        Irp = FFA0F650
        Refcount = 0
    Frame Header 8174B0B0:
        Irp = FFABB460
        Refcount = 0
    Leading Edge:
        Stream Pointer 8183EA58 [Public 8183EA90]:
            Frame Header = 81B77E60
...<this part of display not shown>...
```

在 **！ ks.dumpqueue** 输出的上述部分显示中，我们看到有五个帧正在等待或可用。 这些帧是领先还是落后？ 在 **！ ks.dumpqueue** 显示中，帧始终从旧到新列出。 前导边缘的帧标头与列出的第一个帧（最早的帧）的帧标头匹配。 因此，所有可用帧都领先于前缘。

如果情况并非如此，并且所有帧都位于前边缘后面，并且由于克隆指针而具有引用计数，则问题很可能源于硬件或驱动程序的硬件编程。 确保硬件正在向缓冲区完成发出信号 (检查 中断和 DPC)，并确定驱动程序在缓冲区完成时删除克隆（例如）） 来适当响应 (这些通知。

如示例中所示，如果所有帧都领先于前导边缘，则问题几乎肯定是软件问题。 可以通过查看引脚的 And 门获取更多信息。

解释 And 门

引脚和门控制处理。 如果门计数为 1，则可能会进行处理。 使用 **！ ks.dump** 扩展获取 And 门的当前状态：

dbgcmd

```
kd> !ks.dump 8160DDE0 7
Pin object 8160DDE0 [CKsPin = 8160DD50]
    DeviceState    KSSTATE_RUN
    ClientState    KSSTATE_RUN
    CKsPin object 8160DD50 [KSPIN = 8160DDE0]
        State                   KSSTATE_RUN
        Processing Mutex        8160DFD0 is not held
        And Gate &              8160DF88
        And Gate Count          1
```

由于入口计数为 1，因此 And 门处于打开状态。 在这种情况下，请调查导致处理停止的以下潜在原因：

- 进程调度错误地返回STATUS_PENDING。
- 数据可用性案例缺少 KsPinAttemptProcessing 调用。

使用日志记录跟踪重要事件

项目 • 2023/06/15

通常，数据仅通过触发事件、微型驱动程序的处理和缓冲区完成向下游移动。 若要查明挂起或停止的原因，请使用以下命令：

- 检查是否存在不匹配 **的 KsGate*Xxx*** 调用。
- 检查是否省略了 **Ks*Xxx*AttemptProcessing** 调用。
- 查找与触发事件相关的代码中的问题，包括引用问题流的 pin 标志或调用 **KsPinAttemptProcessing 的代码。**
- 在与处理调度相关的代码中查找问题，尤其是在它排队到硬件以及创建克隆指针的位置。
- 在代码中查找与驱动程序的延迟过程调用相关的问题，（DPC），尤其是在缓冲区已完成或对 KsStreamPointerDelete 进行任何调用时。
- 在流的启动代码中查找问题。

收集此信息的最有效方法是记录受影响区域中的所有内容，包括处理、缓冲区获取（（例如克隆和编程硬件）、缓冲区释放（（如删除克隆））以及任何门操作。 此信息大多高度依赖于计时，需要基于内存的日志记录或 ETW。

若要维护基于内存的滚动日志，请使用以下代码：

C++

```
typedef struct _LOGENTRY {
    ULONG Tag;
    ULONG Arg[3];
} LOGENTRY, *PLOGENTRY;
#define LOGSIZE 2048
LONG g_LogCount;
LOGENTRY g_Log [LOGSIZE];
#define LOG(tag,arg1,arg2,arg3) do { \
    LONG i = InterlockedIncrement (&g_LogCount) % LOGSIZE; \
    g_Log [i].Tag = tag; \
    g_Log [i].Arg [0] = (ULONG)(arg1); \
    g_Log [i].Arg [1] = (ULONG)(arg2); \
    g_Log [i].Arg [2] = (ULONG)(arg3); \
} while (0)
```

然后，使用简单的“dc g_Log”在调试器中查看 **g_Log** 数组的内容。

以下示例使用上述基于内存的方案来确定处理停止的原因。 输出来自 graphedt 中的 AVStream 流式处理方案。 记录了以下微型驱动程序事件：

缩写	说明
Strt	当微型驱动程序首先从微型驱动程序的 *"开始"调度* 中排队设备缓冲区时，会发生此事件。
中国<	此事件在微型驱动程序 *的进程调度开始时* 发生。
AddB	当微型驱动程序从其 *进程* 调度中将缓冲区排队到设备时，会发生此事件。
DPC<	此事件发生在微型驱动程序的 *CallOnDPC* 的开头。 它指示缓冲区完成。
Atmp	当微型驱动程序从 DPC 内部调用 **KsPinAttemptProcessing** 时，会发生此事件。
Dele	当微型驱动程序从 DPC 内部调用 以删除克隆流指针时，会发生此事件。

日志摘录如下：

text

```
f9494b80  3c435044 816e2c90 00000000 00000000  DPC<.,n.........
f9494b90  656c6544 816e2c90 81750260 00000000  Dele.,n.`.u.....
f9494ba0  706d7441 816e2c90 ffa4d418 00000000  Atmp.,n.........
f9494bb0  3c637250 819c1f00 00000000 00000000  Prc<............
f9494bc0  42646441 819c1f00 ffa2eb08 00000000  AddB............
f9494bd0  3c435044 816e2c90 00000000 00000000  DPC<.,n.........
f9494be0  656c6544 816e2c90 ffa80348 00000000  Dele.,n.H.......
f9494bf0  706d7441 816e2c90 ffa4d418 00000000  Atmp.,n.........
f9494c00  3c637250 819c1f00 00000000 00000000  Prc<............
f9494c10  42646441 819c1f00 ffa3d9b8 00000000  AddB............
```

第一个日志摘录代表正常的流式处理状态。 在第一行中，调用微型驱动程序的 *CallOnDPC* 以完成缓冲区 (*DPC<*) 。 (*Dele*) 中删除缓冲区，如果队列中有任何未处理的缓冲区 (*Atmp*) ，则调用 **KsPinAttemptProcessing** 向前移动前边缘。 在这种情况下，通过调用 *中国<*) 的进程调度 (可以看出。 (*AddB*) 将更多缓冲区添加到队列中，整个方案将重复执行。

下一个摘录包括停止发生前日志中的最后一个条目。

text

```
f949b430  3c435044 816e2c90 00000000 00000000  DPC<.,n.........
f949b440  656c6544 816e2c90 ffac4de8 00000000  Dele.,n..M......
f949b450  706d7441 816e2c90 ffa4d418 00000000  Atmp.,n.........
f949b460  3c435044 816e2c90 00000000 00000000  DPC<.,n.........
f949b470  656c6544 816e2c90 816ffc80 00000000  Dele.,n...o.....
f949b480  706d7441 816e2c90 ffa4d418 00000000  Atmp.,n.........
f949b490  3c435044 816e2c90 00000000 00000000  DPC<.,n.........
f949b4a0  656c6544 816e2c90 ffa80348 00000000  Dele.,n.H.......
f949b4b0  706d7441 816e2c90 ffa4d418 00000000  Atmp.,n.........
f949b4c0  3c435044 816e2c90 00000000 00000000  DPC<.,n.........
f949b4d0  656c6544 816e2c90 8174e1c0 00000000  Dele.,n...t.....
f949b4e0  706d7441 816e2c90 ffa4d418 00000000  Atmp.,n.........
```

在此示例中，（*DPC*<）的重复实例指示多个缓冲区正在完成，但队列中没有未处理的缓冲区，因此不会调用进程调度，（没有 *Prc*<）表示。 事实上，队列中所有已处理的缓冲区都已完成，这显然是在可以添加任何新的未处理的缓冲区之前完成的。 由于应用程序已在（运行，因此不会）调用 *Start* ，并且不会调用 *CallOnDPC*（因为没有准备好）完成的已处理缓冲区，因此任何新缓冲区显然在前缘之前累积，等待处理，且没有任何启动处理。

问题是已设置KSPIN_FLAG_DO_NOT_INITIATE_PROCESSING标志。 设置此标志后，只能通过调用 *Start* 或 *CallOnDPC* 进行处理。 如果未设置此标志，则每当将新缓冲区添加到队列时，都会启动处理。

解释 Bug 检查 0xCB

项目 • 2023/10/26

与调试视频流停止相关的最常见 bug 检查代码是 bug 检查0xCB (DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS) 。 有关其参数的详细列表，请参阅 **bug 检查0xCB**。

bug 检查发生时显示的消息将指向 Ks.sys 作为原因。

dbgcmd

```
Use !analyze -v to get detailed debugging information.
BugCheck CB, {f90c6ae0, f9949215, 81861788, 26}
Probably caused by : ks.sys ( ks!KsProbeStreamIrp+333 )
```

建议使用 **！ analyze -v** 获取更多详细信息。

dbgcmd

```
kd> !analyze -v
DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS (cb)
Caused by a driver not cleaning up completely after an I/O.
When possible, the guilty driver's name (Unicode string) is printed on
the bugcheck screen and saved in KiBugCheckDriver.
Arguments:
Arg3: 81861788, A pointer to the MDL containing the locked pages.
```

现在，使用 **！ search** 扩展查找与 MDL 指针关联的虚拟地址。

dbgcmd

```
kd> !search 81861788
Searching PFNs in range 00000001 - 0000FF76 for [FFFFFFFF81861788 -
FFFFFFFF81861788]

Pfn      Offset   Hit      Va       Pte
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
000008A7 00000B0C 81861788 808A7B0C C020229C
00000A04 00000224 16000001 80A04224 C0202810
...
00001732 000009B4 81861788 817329B4 C0205CC8
```

对于找到的每个虚拟地址 (VA) ，请查找 IRP 签名。 为此，将 **dd** 命令与 VA 减去一个 DWORD。

dbgcmd

```
kd> dd 808A7B0C-4 l4
808a7b08  f9949215 81861788 00000026 00000000
kd> $ Not an Irp
kd> dd 80A04224-4 l4
80a04220  00000000 00000000 00000000 00000000
kd> $ Not an Irp
kd> dd 817329B4-4 l4
817329b0  01900006 81861788 00000070 ffa59220
kd> $ Matches signature
```

找到具有 IRP 签名的 VA 后，请使用 **！ irp** 扩展找出此 IRP 上挂起的驱动程序。

dbgcmd

```
kd> !irp 817329b0 7
Irp is active with 2 stacks 2 is current (= 0x81732a44)
 Mdl = 81861788 System buffer = ffa59220 Thread 00000000:  Irp stack trace.
>[  e, 0]   1  1 81a883c8 81ae6158 00000000-00000000    pending
 \Driver\TESTCAP
                       Args: 00000070 00000000 002f4017 00000000
```

在这种情况下，\Driver\TESTCAP 可能是导致 bug 检查的原因。

分析捕获停滞问题

项目 • 2024/02/15

下面是模拟捕获停止的人工创建的场景。 这是一个特别有价值的方案，因为类似的情况经常发生在压力测试中。 方案如下所示：

Windows 录制组件 Sndrec32 正在从主捕获设备录制，在本例中为 Creative SBLive 波形设备。 在一段时间内，它通常记录：但是，图形在 8.50 秒时停止，因为我们显式导致 portcls 无法完成捕获 IRP，以便进行此测试。

应用程序显示正在运行，但流位置未推进。 位置在 8.50 秒处停止。

由于此计算机上的主捕获设备是 PCI 声音卡，因此首先使用 **！ ks.pciaudio** 命令尝试确定起点。 使用标志值 1 请求显示所有正在运行的流：

dbgcmd

```
kd> !pciaudio 1

1 Audio FDOs found:
 Functional Device 8121c030 [\Driver\emu10k]
        Wave Cyclic Streams:
            Pin 812567c0 RUN [emu10k1m!CMiniportWaveCyclicStreamSBLive
ff9ec7f8]
```

在这种情况下，只有一台 PCI 音频设备，它由 Intel emu10k 驱动程序（\Driver\emu10k）提供服务。 此驱动程序当前有一个正在运行的流（0x812567C0）。 现在可以使用 **！ ks.graph** 查看内核图。 将级别*和标志都设置为* 7，以获取停止时的最大详细信息：

dbgcmd

```
kd> !graph 812567c0 7 7
Attempting a graph build on 812567c0...  Please be patient...
Graph With Starting Point 812567c0:
"emu10k" Filter ff9ebb98, Child Factories 5
    Output Factory 0:
        Pin 812567c0 (File 811c6630, -> "splitter" 811df960) Irps(q/p) = 8,
0
 Queued: 81255418 811df008 81252008 81255280 81250b30 ffa1fe70 81252e70
ffa01d98
```

以上显示了工厂 0 的详细信息。 emu10k 输出引脚0x812567C0连接到拆分器输入引脚0x811DF960。 有八个 IRP 排队到 emu10k 的输出引脚。 ！ ks.graph **的**输出继续如下：

dbgcmd

```
"splitter" Filter ffb18890, Child Factories 2
    Output Factory 0:
        Pin 811df430 (File ffa55f90) Irps(q/p) = 10, 0
 Queued: ffadd008 ffa73b00 ffa1e998 811de310 ffa54370 ffaaf008 811dee70
81250e70 811de580 811de8c0
```

有 10 个 IRP 排队，用于拆分器输出引脚。

dbgcmd

```
    Input Factory 1:
        Pin 811df960 (File 81187820, <- "emu10k" 812567c0) Irps(q/p) = 0, 8
            Pending: 81255418 811df008 81252008 81255280 81250b30 ffa1fe70
81252e70 ffa01d98
```

拆分器输入引脚没有排队的 IRP;但是，它正在等待 emu10k 中的 8 个进入队列。

dbgcmd

```
Analyzing a Hung Graph From 812567c0:

Suspect Filters (For a Hung Graph):
 "emu10k" Filter ff9ebb98 or class "PortCls Wave Cyclic" is suspect.
        Reasons For This Analysis:
            - No critical pin has less than 8 queued Irps
            - Downstream "splitter" pin 811df960 is starved
        Irps to check:
 81255418 811df008 81252008 81255280 81250b30 ffa1fe70 81252e70 ffa01d98
```

从此信息中，分析器建议 emu10k 或 WaveCyclic 可能处于故障。 它还提供可疑 IRP 的列表;这些是排队到 emu10k 的输入引脚的 IRP。 如果任一 IRP 完成，拆分器将复制数据并完成 IRP，图形将进行。 出于某种原因，emu10k 未完成这些捕获 Irps。

实时本地调试

项目 • 2023/10/26

在 Microsoft Windows XP 及更高版本的操作系统中，可以通过使用 **-kl** 命令行选项启动内核调试器 (KD) 或 WinDbg 来执行本地内核调试：

控制台

```
kd [-y SymbolPath] -kl
```

或

控制台

```
windbg [-y SymbolPath] -kl
```

在 Windows Vista 及更高版本中，本地内核调试要求使用 **/debug** 选项启动计算机。 以管理员身份打开命令提示符窗口，并输入 **bcdedit /debug on**。 重新启动计算机。

> **ⓘ 重要**
>
> 在使用 BCDEdit 更改启动信息之前，可能需要暂时挂起测试电脑上的 Windows 安全功能，例如 BitLocker 和安全启动。 测试完成后重新启用这些安全功能，并在禁用安全功能时适当地管理测试电脑。

在 Windows Vista 及更高版本中，本地内核调试要求以管理员身份运行调试器。

实时本地调试对于调试在附加调试器时难以重现的问题非常有用;但是，需要了解时间敏感信息（包括数据包、IRP 和 SRB 数据）的任何内容都不太可能起作用，除非问题是挂起或停滞。

执行本地调试时，请考虑以下变量：

- **总体状态。** 例如，流是否正在运行？ 流是否已暂停？
- **数据包计数。** 例如，是否有 IRP 排队到流？
- **数据包计数更改。** 流是否在移动？
- **数据包列表中的更改。**
- **挂起的内核线程。**

请考虑其中最后一个。

检查 LKD 中的挂起线程

首先，使用 **！ process 0 0** 扩展标识包含挂起线程的进程。 然后，再次发出 **！ process** 以获取有关该线程的详细信息：

dbgcmd

```
lkd> !process 816a550 7
        THREAD 81705da8  Cid 0b5c.0b60  Teb: 7ffde000 Win32Thread: e1b2d890
WAIT: (Suspended)
        IRP List:
            816c9ad8: (0006,0190) Flags: 00000030  Mdl: 00000000
        Start Address kernel32!BaseProcessStartThunk (0x77e5c650)
        Win32 Start Address 0x0101c9be
        Stack Init f50bf000 Current f50bea74 Base f50bf000 Limit f50b9000
Call 0
        Priority 10 BasePriority 8 PriorityDecrement 0
```

不显示线程，但堆栈地址为 。 对堆栈上的当前地址使用 **dds** (或 **ddq**) 命令会生成一个用于进一步调查的起点，因为它指定要调用哪个进程。

dbgcmd

```
lkd> dds f50bea74
f50bea74  f50bebc4
f50bea78  00000000
f50bea7c  80805795 nt!KiSwapContext+0x25
f50beab4  8080ece0 nt!KeWaitForSingleObject
f50beabc  f942afda ks!CKsQueue::CancelAllIrps+0x14
f50bead8  f94406c4 ks!CKsQueue::SetDeviceState+0x170
f50beb00  f943f6f1 ks!CKsPipeSection::DistributeDeviceStateChange+0x1d
f50beb24  f943fb1e ks!CKsPipeSection::SetDeviceState+0xb2
```

无法加载的模块的图形分析

项目 • 2023/10/26

本部分介绍在使用可卸载模块（如 KMixer）时可能会影响你的方案。

加载后，扩展模块在第一个命令使用时初始化。 初始化时，扩展模块会检查是否加载了每个模块以及是否具有正确的符号。 如果卸载了任何单个模块或加载了不正确的符号，则扩展将禁用处理该模块的标识、转储等的库扩展。 在这种情况下，需要手动重新启用禁用的模块。

如果在启动时加载扩展，则可能会出现上述情况。 具体而言，如果加载 Ks.dll 然后发出 .reboot 命令，则可能会遇到这种情况。 或者，如果在启动和加载 Ks.dll 时中断调试器，则可能发生此情况。

在以下示例中，我们将从 Telex USB 麦克风捕获两个流 (sndrec32) 。 在 **拆分器上中断！FilterProcess** 并在拆分器的筛选器上运行 **！ ks.graph** 会生成：

dbgcmd

```
kd> !graph ffa0c6d4 7
Attempting a graph build on ffa0c6d4...  Please be patient...
Graph With Starting Point ffa0c6d4:
"usbaudio" Filter ffaaa768, Child Factories 2
    Output Factory 0:
        Pin ffb1caf0 (File 811deeb8, -> "splitter" ffa8b008) Irps(q/p) = 7,
1
"splitter" Filter ffa0c660, Child Factories 2
    Output Factory 0:
        Pin 81250008 (File ffb10028) Irps(q/p) = 3, 0
        Pin 811df9c0 (File ffaaf2f0) Irps(q/p) = 3, 0
    Input Factory 1:
        Pin ffa8b008 (File ffb26d68, <- "usbaudio" ffb1caf0) Irps(q/p) = 1,
7
```

在此示例中，KMixer 已加载并连接到拆分器，但 Kmixer 未显示在图中。 有 IRP 排队到拆分器输出引脚，但 **！ ks.graph** 命令无法回溯和发现 KMixer。 发出 **！ ks.libexts details** 命令以进一步调查：

dbgcmd

```
kd> !libexts details
## LibExt Details:
------------------------------------------------
LibExt "portcls!" :
    Status :  ACTIVE
    This is the port class library extension to the KS DLL.  It supports
    dumping wave cyclic, wave pci, irp streams, and several other upper
```

```
    level structures.
    Commands Exported: pciaudio
    Help : pchelp
    Hooks : dump dumpqueue dumpcircuit conv(file) conv(device) graph
LibExt "STREAM!" :
    Status :  ACTIVE
    This is the stream class library extension to the KS DLL.  It supports
    dumping device extensions, filters, streams, and SRBs.
    Hooks : dump enumdevobj graph
LibExt "kmixer!" :
    Status :  INACTIVE
    This is the KMIXER extension to the KS DLL.  It supports
    virtually nothing at this point!
    Hooks : dump graph
```

根据上述输出，当前禁用扩展的 KMixer 部分 (状态：非活动) 。 由于扩展模块首次在未加载 KMixer 的上下文中使用，因此 Ks.dll 禁用了扩展的 KMixer 部分，以防止对卸载的模块进行耗时的引用。

若要显式启用 KMixer，可以将 **！ ks.libexts** 与 **enable** 参数配合使用，如下所示：

dbgcmd

```
kd> !libexts enable kmixer
LibExt "kmixer" has been enabled.

kd> !graph ffa0c6d4 7
Attempting a graph build on ffa0c6d4...  Please be patient...
Graph With Starting Point ffa0c6d4:
"usbaudio" Filter ffaaa768, Child Factories 2
    Output Factory 0:
        Pin ffb1caf0 (File 811deeb8, -> "splitter" ffa8b008) Irps(q/p) = 7,
1
"splitter" Filter ffa0c660, Child Factories 2
    Output Factory 0:
        Pin 81250008 (File ffb10028, -> "kmixer" 8123c000) Irps(q/p) = 3, 0
        Pin 811df9c0 (File ffaaf2f0, -> "kmixer" 81236000) Irps(q/p) = 3, 0
    Input Factory 1:
        Pin ffa8b008 (File ffb26d68, <- "usbaudio" ffb1caf0) Irps(q/p) = 1,
7
"kmixer" Filter ffa65b70, Child Factories 4
    Input Factory 2:
        Pin 81236000 (File ffaaf7d0, <- "splitter" 811df9c0) Irps(q/p) = 0,
0
    Output Factory 3:
        Pin 81252d00 (File 811df1d8) Irps(q/p) = 10, 0
"kmixer" Filter ffb03808, Child Factories 4
    Input Factory 2:
        Pin 8123c000 (File ffb10130, <- "splitter" 81250008) Irps(q/p) = 0,
0
```

```
    Output Factory 3:
        Pin ffa1e9c0 (File 81253468) Irps(q/p) = 10, 0
```

KMixer 现在在捕获图中按预期显示。

使用 !ks.graph

项目 • 2023/10/26

！ ks.graph 命令是内核流式处理扩展模块中功能最强大的扩展命令之一。 此命令显示任何给定起点在内核模式下的整个图形的图片。

在运行 **！ ks.graph** 之前，可能需要启用所有能够处于活动状态的库扩展。 为此，请发出 **！ ks.libexts enableall** 命令。 **！ ks.graph** 的输出将是内核模式图的文本说明，按拓扑顺序排序。 以下是示例：

dbgcmd

```
kd> !graph ffa0c6d4 7
Attempting a graph build on ffa0c6d4...  Please be patient...
Graph With Starting Point ffa0c6d4:
"usbaudio" Filter ffaaa768, Child Factories 2
    Output Factory 0:
        Pin ffb1caf0 (File 811deeb8, -> "splitter" ffa8b008) Irps(q/p) = 7,
1
```

此示例显示一个捕获图，其中两个 Sndrec32.exe 从 Telex USB 麦克风捕获。 在上一节)中，每条单独的记录都以 usbaudio 的名称 (开头，并显示筛选器上 2) 的子引脚工厂的筛选器地址 ((0xFFAAA768) 和数量。

在每个条目下，枚举每个工厂，并列出每个引脚实例 (0xFFB1CAF0) 的地址、对应于每个实例的文件对象 (0x811DEEB8) 、连接的方向、该连接的目标以及目标引脚 (0xFFA8B008) 的地址。 还显示每个引脚的排队 (7 个) 和挂起的 IRP (1 个) 。

具有向前方向符号 (>) 的连接指示引脚是输出引脚并连接到输入引脚。 另一方面，具有反向方向符号 (<-) 的连接是输入引脚，并显示连接的起源。 输出将继续，如下所示：

dbgcmd

```
"splitter" Filter ffa0c660, Child Factories 2
    Output Factory 0:
        Pin 81250008 (File ffb10028, -> "kmixer" 8123c000) Irps(q/p) = 3, 0
        Pin 811df9c0 (File ffaaf2f0, -> "kmixer" 81236000) Irps(q/p) = 3, 0
    Input Factory 1:
        Pin ffa8b008 (File ffb26d68, <- "usbaudio" ffb1caf0) Irps(q/p) = 1,
7
"kmixer" Filter ffa65b70, Child Factories 4
    Input Factory 2:
        Pin 81236000 (File ffaaf7d0, <- "splitter" 811df9c0) Irps(q/p) = 0,
0
    Output Factory 3:
        Pin 81252d00 (File 811df1d8) Irps(q/p) = 10, 0
"kmixer" Filter ffb03808, Child Factories 4
```

```
    Input Factory 2:
        Pin 8123c000 (File ffb10130, <- "splitter" 81250008) Irps(q/p) = 0,
0
    Output Factory 3:
        Pin ffa1e9c0 (File 81253468) Irps(q/p) = 10, 0
```

若要遵循图形，请使用以下过程：

若要遵循此图：：

1. 查找感兴趣的图钉。 请考虑0xFFB1CAF0，usbaudio 的输出引脚 (工厂 0) 。
2. 找到连接的引脚。 在此示例中，这是拆分器引脚0xFFA8B008。
3. 查看连接方向，然后以可视方式移动，查找筛选器名称。 (请记住，列表按拓扑排序。) 在此示例中，右箭头指示我们需要在列表中查看此条目下方才能找到相应的图钉。 拆分器筛选器0xFFA0C660紧接在下方。
4. 在筛选器引脚实例列表中查找目标引脚地址。 在本例中，此地址0xFFA8B008。

“遵循此图： ”命令还可用于分析任何给定起点的已停止的图形。 为此，请在 *Flags* 参数中指定 4：

dbgcmd

```
kd> !graph 812567c0 7 4

Attempting a graph build on 812567c0...  Please be patient...

Graph With Starting Point 812567c0:

"emu10k" Filter ff9ebb98, Child Factories 5
    Output Factory 0:
        Pin 812567c0 (File 811c6630, -> "splitter" 811df960) Irps(q/p) = 8,
0
"splitter" Filter ffb18890, Child Factories 2
    Output Factory 0:
        Pin 811df430 (File ffa55f90) Irps(q/p) = 10, 0
    Input Factory 1:
        Pin 811df960 (File 81187820, <- "emu10k" 812567c0) Irps(q/p) = 0, 8

Analyzing a Hung Graph From 812567c0:

Suspect Filters (For a Hung Graph):
    "emu10k" Filter ff9ebb98 or class "PortCls Wave Cyclic" is suspect.
        Reasons For This Analysis:
            - No critical pin has less than 8 queued Irps
            - Downstream "splitter" pin 811df960 is starved
        Irps to check:
            81255418 811df008 81252008 81255280 81250b30 ffa1fe70 81252e70
ffa01d98
```

```
NOTE: The above is based on heuristic analysis.  It is not designed to be a
      replacement for an actual developer looking at this particular hang!
The
      filters listed as suspects may or may not be the actual cause of the
      stall!
```

对于此类输出，请查看“嫌疑人”列表。 这些可疑筛选器是处于关系图中进度的关键路径中的筛选器。 根据分析器为停止生成的原因从该点开始调试。

注意 此功能只应用于已停止的图形！ 分析器无法知道图形处于此状态的时间。 闯入调试器并将正在运行的图形分析为已停止的图形仍会显示可疑筛选器。

SCSI 微型端口调试

项目 • 2023/03/08

本节包括：

SCSI 微型端口调试概述

用于调试 SCSI 微型端口驱动程序的扩展

用于 SCSI 微型端口调试的 Bug 检查

分析已停止的驱动程序和超时

用于分析可再现问题的重要断点

SCSI 微型端口调试概述

项目 • 2023/06/15

SCSI 验证

可以在两个扩展模块中找到小型计算机系统接口 (SCSI) 调试扩展：Scsikd.dll 和 Minipkd.dll。 有关这些模块中最重要的扩展命令的概述，请参阅 用于调试 SCSI 微型端口驱动程序的扩展。 有关完整列表，请参阅 SCSI 微型端口扩展。

SCSIkd.dll扩展命令可用于任何版本的 Windows。 Minipkd.dll扩展命令只能在 Windows XP 和更高版本的 Windows 中使用。 Minipkd.dll 中的命令仅适用于使用 SCSI 端口驱动程序的微型端口驱动程序。

若要测试 SCSI 微型端口驱动程序，请使用驱动程序验证程序的 SCSI 验证功能。 有关驱动程序验证程序的信息，请参阅 Windows 驱动程序工具包 (WDK) 文档中的驱动程序 验证程序 。

用于调试 SCSI 微型端口驱动程序的扩展

项目 • 2023/10/26

调试 SCSI 微型端口驱动程序时，可能会发现以下调试器扩展很有用。 首先列出常规调试器扩展，然后列出特定于 SCSI 微型端口调试的扩展。

!devobj

！devobj 扩展显示有关DEVICE_OBJECT的详细信息。 如果 “**当前 Irp** ”字段为非空，则这可能是由等待映射寄存器的 SCSI 驱动程序引起的。

以下是示例：

dbgcmd

```
0: kd> !devobj 8633da70
Device object (8633da70) is for:
 adpu160m1 \Driver\adpu160m DriverObject 8633eeb8
Current Irp 860ef008 RefCount 0 Type 00000004 Flags 00000050
Dacl e129871c DevExt 8633db28 DevObjExt 8633dfd0
ExtensionFlags (0000000000)
AttachedTo (Lower) 863b2978 \Driver\PCI
Device queue is not busy.
```

!errlog

！errlog 扩展显示 I/O 系统错误日志中任何挂起条目的内容。

！对象

！object 扩展显示有关系统对象的信息。 此扩展显示所有 SCSI 设备。

例如：

dbgcmd

```
0: kd> !object \device\scsi
Object: e12a2520  Type: (863d12c8) Directory
    ObjectHeader: e12a2508
    HandleCount: 0  PointerCount: 9
    Directory Object: e1001100  Name: Scsi

    Hash Address  Type          Name
    ---- -------  ----          ----
     04  86352040 Device        adpu160m1Port3Path0Target6Lun0
     11  86353040 Device        adpu160m1Port3Path0Target1Lun0
     13  86334a70 Device        lp6nds351
     22  862e6040 Device        adpu160m1Port3Path0Target0Lun0
     24  8633da70 Device        adpu160m1
```

```
      25  86376040 Device        adpu160m2
      34  862e5040 Device        adpu160m1Port3Path0Target2Lun0
```

!pcr

！ pcr 扩展显示有关处理器控制区域的详细信息，（处理器上的 PCR）。 该信息包括 DPC 队列中的项，这些项可能很有用。 调试已停止的驱动程序或超时时。

!minipkd.help

！ minipkd.help 扩展显示所有 Minipkd.dll 扩展命令的列表。

如果出现类似于以下内容的错误消息，则表明符号路径不正确，并且未指向 Scsiport.sys 符号的正确版本。

dbgcmd

```
minipkd error (0) <path> ... \minipkd\minipkd.c @ line 435
```

.sympath (Set Symbol Path) 命令可用于显示当前路径和更改路径。 .reload (Reload Module) 命令将从当前路径重新加载符号。

！ minipkd.adapter 适配器

！ minipkd.adapter 扩展显示有关指定适配器的详细信息。 **可以通过查看！minipkd.adapters** 显示中的 **DevExt** 字段来找到适配器。

!minipkd.adapters

！ minipkd.adapters 扩展显示与 Windows 标识的 SCSI 端口驱动程序一起使用的所有适配器，以及与每个适配器关联的单个设备。

以下是示例：

dbgcmd

```
0: kd> !minipkd.adapters
Adapter \Driver\lp6nds35     DO 86334a70         DevExt 86334b28
Adapter \Driver\adpu160m     DO 8633da70         DevExt 8633db28
 LUN 862e60f8 @(0,0,0) c ev     pnp(00/ff) pow(0,0) DevObj 862e6040
 LUN 863530f8 @(0,1,0) c ev p d pnp(00/ff) pow(0,0) DevObj 86353040
 LUN 862e50f8 @(0,2,0) c ev     pnp(00/ff) pow(0,0) DevObj 862e5040
 LUN 863520f8 @(0,6,0)   ev     pnp(00/ff) pow(0,0) DevObj 86352040
Adapter \Driver\adpu160m     DO 86376040         DevExt 863760f8
```

类似于以下内容的错误消息指示符号路径不正确且未指向正确版本的 Scsiport.sys 符号，或者 Windows 未识别任何使用 SCSI 端口驱动程序的适配器：

dbgcmd

```
minipkd error (0) <path> ... \minipkd\minipkd.c @ line 435
```

如果 **！ minipkd.help** 扩展命令成功返回帮助信息，则 SCSI 端口符号是正确的。

！ minipkd.exports 适配器

！ minipkd.exports 扩展显示指定适配器的微型端口导出的地址。

！ minipkd.lun {LUN |Device}

！ minipkd.lun 扩展显示有关指定逻辑单元扩展 (LUN) 的详细信息。 可以通过指定 LUN 的地址 (可通过查看 **！ minipkd.adapters** 显示) 中的 **LUN** 字段来指定 LUN，也可以按其物理设备对象 (指定，该对象可在 **！ minipkd.adapters** 显示) 的 **DevObj** 字段中找到。

！ minipkd.portconfig PortConfig

！ minipkd.portconfig 扩展显示有关指定PORT_CONFIGURATION_DATA的详细信息。 PortConfig 可在 **！ minipkd.adapter** 显示的"**端口配置信息**"字段中找到。

！ minipkd.req {Adapter |Device}

！ minipkd.req 扩展显示有关指定适配器或 LUN 设备上所有当前活动请求的信息。

！ minipkd.srb SRB

！ minipkd.srb 扩展显示有关指定 SCSI 请求块的详细信息 (SRB) 。 SRB 由地址指定。 可以在 **！ minipkd.req** 命令输出的 **SRB** 字段中找到所有当前活动请求的地址。

！ scsikd.classext [Device [Level]]

！ scsikd.classext 扩展显示有关指定类的详细信息即插即用 (PnP) 设备或所有此类设备的列表。 *设备*是类 PnP 设备的设备对象或设备扩展。 如果省略 *Device* ，则显示所有类 PnP 扩展的列表。

以下是示例：

dbgcmd

```
0: kd> !scsikd.classext

 ' !scsikd.classext 8633e3f0 '   (             ) "IBM     " / "DDYS-T09170M
" / "S93E" / "        XBY45906"
 ' !scsikd.classext 86347b48 '   (paging device) "IBM     " / "DDYS-T09170M
" / "S80D" / "        VDA60491"
  ' !scsikd.classext 86347360 '   (             ) "UNISYS  " /
"003451ST34573WC " / "5786" / "HN0220750000181300L6"
  ' !scsikd.classext 861d1898 '   (             ) "" / "MATSHITA CD-ROM CR-
177" / "7T03" / ""

 usage: !classext <class fdo> <level [0-2]>
```

! scsikd.scsiext 设备

! scsikd.scsiext 扩展显示有关指定 SCSI 端口扩展的详细信息。 *设备*可以是适配器或 LUN 的设备对象或设备扩展。

下面是一些示例：

dbgcmd

```
0: kd> !scsikd.scsiext 86353040
Common Extension:
   < ..omitted.. >
Logical Unit Extension:
  Address (3, 0, 1, 0) Claimed  Enumerated Visible
  LuFlags (0x00000000):
  Retry 0x00          Key 0x008889ff
  Lock 0x00000000  Pause 0x00000000   CurrentLock: 0x00000000
  HwLuExt 0x862e6f00  Adapter 0x8633db28  Timeout 0x0000000a
  NextLun 0x00000000  ReadyLun 0x00000000
  Pending 0x00000000  Busy 0x00000000     Untagged 0x00000000
  < ..omitted.. >
Request list @0x86353200:
       Tick count is 2526
       SrbData 8615d700  Srb 8611f4fc  Irp 8611f2b8   Key 60197  <1s
       SrbData 85e72868  Srb 86100c3c Irp 861009f8   Key e29dc7  <1s

0: kd> !scsikd.scsiext 8633da70
Common Extension:
   < ..omitted.. >
Adapter Extension:
  Port 3     IsPnp VirtualSlot HasInterrupt
  LowerPdo 0x84f9fb68   HwDevExt 0x84634004   Active Requests 0x00000000
  MaxBus 0x03   MaxTarget 0x40   MaxLun 0x08
  Port Flags (0x00001000): PD_DISCONNECT_RUNNING
  NonCacheExt 0x850d4000  IoBase 0xd80f0000   Int 0x23  < ..omitted.. >
```

! scsikd.srbdata Address

! scsikd.srbdata 扩展显示有关指定SRB_DATA跟踪块的详细信息。

用于 SCSI 微型端口调试的错误检查

项目 • 2024/03/04

调试 SCSI 微型端口驱动程序的过程中，主要会出现两个错误检查：错误检查 0x77 (KERNEL_STACK_INPAGE_ERROR) 和错误检查 0x7A (KERNEL_DATA_INPAGE_ERROR)。 有关其参数的完整详细信息，请参阅**错误检查 0x77** 和**错误检查 0x7A**。

这些错误检查中的每一个都表示发生了分页错误。 这些错误检查有三个主要原因：

- 由于特定设备上的超时或适配器上没有活动而导致总线完全重置
- 选择超时
- 控制器错误

若要确定失败的确切原因，请首先使用 **!scsikd.classext** 扩展，该扩展显示有关最近失败的请求的信息，包括 SRB 状态、SCSI 状态和请求的感知数据。

dbgcmd

```
kd> !scsikd.classext 816e96b0
Storage class device 816e96b0 with extension at 816e9768

Classpnp Internal Information at 817b4008

    Failed requests:

           Srb    Scsi
    Opcode Status Status Sense Code  Sector   Time  Stamp
    ------ ------ ------ ---------- -------- ------------
      2a     0a     02    03 0c 00  0000abcd 23:01:07.453  Retried
      28     0a     02    03 04 00  0000abcd 23:01:07.984  Retried

dt classpnp!_CLASS_PRIVATE_FDO_DATA 817b4008 -

...
```

在前面的示例中，操作码 0x2A 表示写入操作，0x28 表示读取操作。 示例中的 SCSI 状态为 02，指示检查条件。 感知代码提供更多错误信息。

与往常一样，微型端口驱动程序开发人员负责将错误代码从其硬件关联到 SRB 状态代码。 通常，超时与 SRB 0x0A（用于选择超时的代码）相关联。 SRB 0x0e 通常与总线完全重置相关，但也可能与控制器错误相关。

分析已停止的驱动程序和超时

项目 • 2024/02/15

调试 SCSI 微型端口驱动程序时，挂起和超时的三个最常见原因是：

- SCSI 微型端口 DPC 未运行
- SCSI 微型端口无法请求下一个请求
- SCSI 微型端口未完成请求，通常是因为它正在等待映射寄存器。

如果怀疑 SCSI 微型端口 DPC 未运行，请使用 **！ rf** 显示当前处理器的 DPC 队列。 如果 SCSI 端口 DPC 例程位于 DPC 队列中，请将断点置于此例程上，以确定是否调用此例程。 否则，在每个设备上使用 ！ scsikd.scsiext。 请考虑 ！ scsikd.scsiext **扩展的**以下示例输出：

dbgcmd

```
0: kd> !scsikd.scsiext 86353040
Common Extension:
   < ..omitted.. >
Logical Unit Extension:
  Address (3, 0, 1, 0) Claimed  Enumerated Visible
  LuFlags (0x00000000):
  Retry 0x00          Key 0x008889ff
  Lock 0x00000000  Pause 0x00000000   CurrentLock: 0x00000000
  HwLuExt 0x862e6f00  Adapter 0x8633db28  Timeout 0x0000000a
  NextLun 0x00000000  ReadyLun 0x00000000
  Pending 0x00000000  Busy 0x00000000     Untagged 0x00000000

  . . .

Request list @0x86353200:
      Tick count is 2526
      SrbData 8615d700  Srb 8611f4fc  Irp 8611f2b8   Key 60197  <1s
      SrbData 85e72868  Srb 86100c3c Irp 861009f8   Key e29dc7  <1s
```

如果超时槽为 -1 且未标记的槽为非零，或者超时槽为非零且显示有请求，则微型端口无法请求下一个请求。

或者，如果重试槽和繁忙槽为非零，则 SCSI 微型端口可能无法完成请求，因为它正在等待映射寄存器。 同样，如果未标记的和挂起的槽是非零的，则 SCSI 微型端口可能正在等待映射寄存器。 在任一情况下，SCSI 请求块（SRB）的地址是繁忙槽中的地址，以及未完成的请求的地址。 有关 SRB 的详细信息，请使用带有此地址的 **！ minipkd.srb** 扩展作为输入。

！ devobj 扩展确定 SCSI 微型端口是否正在等待映射寄存器。 使用发出请求的设备的设备对象地址作为对 ！ devobj **的**输入。 如果当前 IRQ 为非零，则 SCSI 微型端口很可能正在等待映射寄存器。

用于分析可再现问题的重要断点

项目 • 2023/10/26

调试 SCSI 微型端口驱动程序时，有三个例程可用于设置断点：

- **scsiport！scsiportnotification**
- **scsiport！spstartiosynchronized**
- **微型端口！HwStartIo**

将请求发送到微型端口后立即调用例程 **scsiport！scsiportnotification** 。 因此，如果在 **scsiport！scsiportnotification** 中设置断点，然后使用 kb 3 运行堆栈回溯，则可以确定微型端口是否正在接收和完成请求。 如果第一个参数为零，则请求已完成。 如果第一个参数为非零值，则第三个参数是未完成的 SCSI 请求块 (SRB) 的地址，可以使用 **！minipkd.srb** 扩展进一步分析情况。

在 **scsiport！spstartiosynchronized** 或微型端口中放置断点 **！HwStartIo** 将在向微型端口发送请求之前导致中断。

即插即用调试

项目 • 2023/03/08

本节包括：

用于调试即插即用驱动程序的扩展

确定设备的状态

设备节点状态标志

设备管理器问题代码

检查资源冲突

用于调试即插即用驱动程序的扩展

项目 • 2023/06/15

调试即插即用驱动程序时，可能会发现以下调试器扩展很有用。

!arbiter

显示当前系统资源仲裁程序。 仲裁程序是由总线驱动程序公开的一段代码，用于仲裁资源请求，并尝试解决该总线上连接的设备之间的资源冲突。

!cmreslist

显示指定设备对象的CM_RESOURCE_LIST。

必须知道 CM 资源列表的地址。

以下是示例：

dbgcmd

```
kd> !cmreslist 0xe12576e8

CmResourceList at 0xe12576e8  Version 0.0  Interface 0x1  Bus #0
  Entry 0 - Port (0x1) Device Exclusive (0x1)
    Flags (0x01) - PORT_MEMORY PORT_IO
    Range starts at 0x3f8 for 0x8 bytes
  Entry 1 - Interrupt (0x2) Shared (0x3)
    Flags (0x01) - LATCHED
    Level 0x4, Vector 0x4, Affinity 0xffffffff
```

这表明具有此 CM 资源列表的设备正在使用 I/O 范围 3F8-3FF 和 IRQ 4。

！Dcs

此扩展已过时 -- 其功能已被 ！ pci 包含。 请参阅本节后面的！ pci 100 示例。

!devext

显示各种设备的总线特定设备扩展信息。

!devnode

显示设备树中有关节点的信息。

设备节点 0 (零) 是设备树的根。

以下是示例：

dbgcmd

```
0: kd> !devnode 0fffffa8003634af0
DevNode 0fffffa8003634af0 for PDO 0fffffa8003658590
  Parent 0fffffa8003604010   Sibling 0fffffa80036508e0   Child 0000000000
  InstancePath is "ROOT\SYSTEM\0000"
  ServiceName is "swenum"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  StateHistory[09] = DeviceNodeEnumerateCompletion (0x30d)
  StateHistory[08] = DeviceNodeEnumeratePending (0x30c)
  StateHistory[07] = DeviceNodeStarted (0x308)
  StateHistory[06] = DeviceNodeStartPostWork (0x307)
  StateHistory[05] = DeviceNodeStartCompletion (0x306)
  StateHistory[04] = DeviceNodeStartPending (0x305)
  ...
  Flags (0x6c000131)  DNF_MADEUP, DNF_ENUMERATED,
                      DNF_IDS_QUERIED, DNF_NO_RESOURCE_REQUIRED,
                      DNF_NO_LOWER_DEVICE_FILTERS,
DNF_NO_LOWER_CLASS_FILTERS,
                      DNF_NO_UPPER_DEVICE_FILTERS,
DNF_NO_UPPER_CLASS_FILTERS
  UserFlags (0x00000008)  DNUF_NOT_DISABLEABLE
  DisableableDepends = 1 (including self)
```

!devobj

显示有关DEVICE_OBJECT的详细信息。

以下是示例：

dbgcmd

```
kd> !devobj 0xff0d4af0

Device object (ff0d4af0) is for:
 00252d \Driver\PnpManager DriverObject ff0d9030
Current Irp 00000000 RefCount 0 Type 00000004 Flags 00001040AttachedDev
ff0b59e0

DevExt ff0d4ba8 DevNode ff0d4a08
Device queue is not busy.
```

！司机

不再支持！ drivers 命令。 请改用 lm t n 命令。

!drvobj

显示有关DRIVER_OBJECT的详细信息。

列出指定驱动程序创建的所有设备对象。

以下是示例：

dbgcmd

```
kd> !drvobj serial

Driver object (ff0ba630) is for:
 \Driver\Serial
Driver Extension List: (id , addr)

Device Object list:
ffba3040  ff0b4040  ff0b59e0  ff0b5040
```

!ecb、!ecd、!ecw

(x86 目标计算机仅) 将一系列值写入 PCI 配置空间。

ib、iw、id

从 I/O 端口读取数据。

这三个命令可用于确定某个 I/O 范围是否由正在调试的驱动程序以外的设备声明。 端口处的 0xFF 字节值表示端口未使用。

!ioreslist

显示指定的IO_RESOURCE_REQUIREMENTS_LIST。

！ Irp

显示有关 IRP 的信息。

!irpfind

显示有关当前在目标系统中分配的所有 IRP 的信息，或有关其字段与指定搜索条件匹配的 IRP 的信息。

！ Pci

(x86 目标计算机仅) 显示 PCI 总线及其连接的任何设备的当前状态。 它还可以显示 PCI 配置空间。

以下示例在主总线上显示设备：

dbgcmd

```
kd> !pci
PCI Bus 0
00:0  8086:1237.02  Cmd[0106:.mb..s]  Sts[2280:.....]  Device  Host bridge
0d:0  8086:7000.01  Cmd[0007:imb...]  Sts[0280:.....]  Device  ISA bridge
0d:1  8086:7010.00  Cmd[0005:i.b...]  Sts[0280:.....]  Device  IDE
controller
0e:0  1011:0021.01  Cmd[0107:imb..s]  Sts[0280:.....]  PciBridge 0->1-1
PCI-PCI
 bridge
10:0  5333:8811.43  Cmd[0023:im.v..]  Sts[0200:.....]  Device  VGA
```

```
compatible controller



The following example displays the devices for the secondary bus, with
verbose output:

kd> !pci 1 1

PCI Bus 1
08:0  10b7:5900.00  Cmd[0107:imb..s]  Sts[0200:.....]  Device  Ethernet
      cf8:80014000  IntPin:1  IntLine:f  Rom:fa000000  cis:0  cap:0
      IO[0]:fce1

09:0  9004:8178.00  Cmd[0117:imb..s]  Sts[0280:.....]  Device  SCSI
controller
      cf8:80014800  IntPin:1  IntLine:f  Rom:fa000000  cis:0  cap:0
      IO[0]:f801       MEM[1]:f9fff000

0b:0  9004:5800.10  Cmd[0116:.mb..s]  Sts[0200:.....]  Device
SubID:9004:8940
1394 host controller
      cf8:80015800  IntPin:1  IntLine:e  Rom:fa000000  cis:0  cap:0
      MEM[0]:f9ffec00
```

以下示例显示 SCSI 控制器 (总线 1、设备 9、函数 0) 的 PCI 配置空间:

dbgcmd

```
kd> !pci 100 1 9 0
00: 9004    ;VendorID=9004
02: 8178    ;DeviceID=8178
04: 0117
;Command=SERREnable,MemWriteEnable,BusInitiate,MemSpaceEnable,IOSpac
eEnable
06: 0280    ;Status=FB2BCapable,DEVSELTiming:1
08: 00      ;RevisionID=00
09: 00      ;ProgIF=00 (SCSI bus controller)
0a: 00      ;SubClass=00
0b: 01      ;BaseClass=01 (Mass storage controller)
0c: 08      ;CacheLineSize=Burst8DW
0d: 20      ;LatencyTimer=20
0e: 00      ;HeaderType=00
0f: 00      ;BIST=00
10: 0000f801;BAR0=0000f801
14: f9fff000;BAR1=f9fff000
18: 00000000;BAR2=00000000
1c: 00000000;BAR3=00000000
20: 00000000;BAR4=00000000
24: 00000000;BAR5=00000000
28: 00000000;CBCISPtr=00000000
2c: 0000    ;SubSysVenID=0000
2e: 0000    ;SubSysID=0000
```

```
30: fa000000;ROMBAR=fa000000
34: 00000000;Reserved=00000000
38: 00000000;Reserved=00000000
3c: 0f      ;IntLine=0f
3d: 01      ;IntPin=01
3e: 08      ;MinGnt=08
3f: 08      ;MaxLat=08
40: 00001580,00001580,00000000,00000000,00000000,00000000,00000000,00000000
60: 00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000
80: 00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000
a0: 00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000
c0: 00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000
e0: 00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000
```

!pcitree

显示有关 PCI 设备对象的信息，包括子 PCI 总线和 CardBus 总线，以及连接到它们的设备。

!pnpevent

显示 PnP 设备事件队列。

!rellist

显示 PnP 关系列表以及任何相关的CM_RESOURCE_LIST和IO_RESOURCE_LIST结构。

确定设备的状态

项目 • 2023/06/15

若要显示整个设备树（从根开始），请使用 **！ devnode 0 1**：

dbgcmd

```
kd> !devnode 0 1
```

通过检查公开它的驱动程序或总线来标识要搜索的开发节点。

devnode 状态标志描述设备的状态。 有关详细信息，请参阅 设备节点状态标志。

在"调试即插即用驱动程序扩展"部分中的 **！ devnode** 命令示例中，swenum 设备是由 PnP 管理器创建的，因此设置了DNF_MADEUP (0x00000001) 标志。

以下示例演示由 PCI 总线创建的设备。 此设备未设置DNF_MADEUP标志。

dbgcmd

```
0: kd> !devnode 0xfffffa8004483490
DevNode 0xfffffa8004483490 for PDO 0xfffffa800448d060
  Parent 0xfffffa80036766d0   Sibling 0xfffffa8004482010   Child
0xfffffa80058ad720
  InstancePath is
"PCI\VEN_8086&DEV_293C&SUBSYS_2819103C&REV_02\3&21436425&0&D7"
  ServiceName is "usbehci"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  StateHistory[09] = DeviceNodeEnumerateCompletion (0x30d)
  StateHistory[08] = DeviceNodeEnumeratePending (0x30c)
  StateHistory[07] = DeviceNodeStarted (0x308)
  StateHistory[06] = DeviceNodeStartPostWork (0x307)
  ...
  Flags (0x6c0000f0)  DNF_ENUMERATED, DNF_IDS_QUERIED,
                      DNF_HAS_BOOT_CONFIG, DNF_BOOT_CONFIG_RESERVED,
                      DNF_NO_LOWER_DEVICE_FILTERS,
DNF_NO_LOWER_CLASS_FILTERS,
                      DNF_NO_UPPER_DEVICE_FILTERS,
DNF_NO_UPPER_CLASS_FILTERS
  CapabilityFlags (0x00002000)  WakeFromD3
```

示例

1. 资源不足设备的开发节点：

dbgcmd

```
kd> !devnode 0xff0d06e8 6

DevNode 0xff0d06e8 for PDO 0xff0d07d0 at level 0x3
  Parent 0xff0d1408   Sibling 0000000000   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xfffffff0
  InstancePath is "ISAPNP\SUP2171\00000067"
  ServiceName is "Modem"
  TargetDeviceNotify List - f 0xff0d074c  b 0xff0d074c
  Flags (.........)  DNF_PROCESSED, DNF_ENUMERATED,
                     DNF_INSUFFICIENT_RESOURCES, DNF_ADDED,
                     DNF_HAS_BOOT_CONFIG
                     Unknown flags 0x40000000

  IoResList at 0xe133e7a8 : Interface 0x1  Bus 0  Slot 0
    Alternative 0 (Version 1.1)
      Preferred Descriptor 0 - NonArbitrated/ConfigData (0x80) Shared (0x3)
        Flags (0000) -
        Data:              : 0x0 0x61004d 0x680063
      Preferred Descriptor 1 - Port (0x1) Undetermined Sharing (0)
        Flags (0x11) - PORT_IO 16_BIT_DECODE
        0x000008 byte range with alignment 0x000001
        2f8 - 0x2ff
      Preferred Descriptor 2 - Interrupt (0x2) Shared (0x3)
        Flags (0x01) - LATCHED
        0x3 - 0x3
```

请注意，开发节点没有 CM 资源列表，因为它未启动且未使用资源，尽管它已请求资源。

2. 请注意，对于旧版驱动程序，此开发节点中没有存储任何资源。

dbgcmd

```
kd> !devnode 0xff0d1648 6

DevNode 0xff0d1648 for PDO 0xff0d22d0 at level 0x2
  Parent 0xff0d2e28   Sibling 0xff0d1588   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xffffffff
  InstancePath is "PCI\VEN_102B&DEV_0519\0&60"
  ServiceName is "mga_mil"
  TargetDeviceNotify List - f 0xff0d16ac  b 0xff0d16ac
  Flags (0x6000500b)  DNF_PROCESSED, DNF_STARTED,
                      DNF_ENUMERATED, DNF_ADDED,
                      DNF_LEGACY_DRIVER, DNF_HAS_BOOT_CONFIG
                      Unknown flags 0x40000000
```

可以为以下类型的设备检索驱动程序的设备对象列表：

dbgcmd

```
kd> !drvobj mga_mil

Driver object (ff0bbc10) is for:
 \Driver\mga_mil
Driver Extension List: (id , addr)

Device Object list:
ff0bb900
```

然后，可以转储此设备对象的数据：

dbgcmd

```
kd> !devobj ff0bb900

Device object (ff0bb900) is for:
 Video0 \Driver\mga_mil DriverObject ff0bbc10
Current Irp 00000000 RefCount 1 Type 00000023 Flags 0000204c
DevExt ff0bb9b8 DevNode ff0bb808
Device queue is not busy.
```

最后，可以转储设备对象引用的开发节点。 此开发节点未在设备树中链接。 它表示用于为旧设备声明资源的“伪开发节点”。 请注意DNF_RESOURCE_REPORTED标志，指示设备是报告检测到的设备。

dbgcmd

```
kd> !devnode ff0bb808 6

DevNode 0xff0bb808 for PDO 0xff0bb900 at level 0xffffffff
  Parent 0xff0daf48   Sibling 0000000000   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xffffffff
  TargetDeviceNotify List - f 0xff0bb86c  b 0xff0bb86c
  Flags (0x00000400)  DNF_RESOURCE_REPORTED
  CmResourceList at 0xe12474e8  Version 0.0  Interface 0x5  Bus #0
    Entry 0 - Port (0x1) Shared (0x3)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x3c0 for 0x10 bytes
    Entry 1 - Port (0x1) Shared (0x3)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x3d4 for 0x8 bytes
    Entry 2 - Port (0x1) Shared (0x3)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x3de for 0x2 bytes
    Entry 3 - Memory (0x3) Device Exclusive (0x1)
      Flags (0000) - READ_WRITE
      Range starts at 0x0000000040000000 for 0x4000 bytes
    Entry 4 - Memory (0x3) Device Exclusive (0x1)
      Flags (0000) - READ_WRITE
      Range starts at 0x0000000040800000 for 0x800000 bytes
```

设备节点状态标志

项目 • 2023/06/15

设备节点状态标志描述设备的状态。

最重要的标志是：

DNF_MADEUP (0x00000001)
设备已创建并归 PnP 管理器所有。 它不是由总线驱动程序创建的。

DNF_DUPLICATE (0x00000002)
设备节点是另一个枚举设备节点的副本。

DNF_HAL_NODE (0x00000004)
设备节点是由硬件抽象层 (HAL) 创建的根节点。

DNF_REENUMERATE (0x00000008)
需要重新枚举设备。

DNF_ENUMERATED (0x00000010)
设备的 PDO 由其父级公开。

DNF_IDS_QUERIED (0x00000020)
操作系统应向设备驱动程序发送IRP_MN_QUERY_ID请求。

DNF_HAS_BOOT_CONFIG (0x00000040)
设备具有 BIOS 分配的资源。 设备被视为伪启动设备，需要参与重新平衡。

DNF_BOOT_CONFIG_RESERVED (0x00000080)
设备的启动资源是保留的。

DNF_NO_RESOURCE_REQUIRED (0x00000100)
设备不需要资源。

DNF_RESOURCE_REQUIREMENTS_NEED_FILTERED (0x00000200)
设备的资源要求列表是筛选列表。

DNF_RESOURCE_REQUIREMENTS_CHANGED (0x00000400)
设备的资源要求列表已更改。

DNF_NON_STOPPED_REBALANCE (0x00000800)
可以使用新资源重启设备，而不会停止。

DNF_LEGACY_DRIVER (0x00001000)
设备的控制驱动程序是非 PnP 驱动程序。

DNF_HAS_PROBLEM (0x00002000)
设备有问题，将被移除。

DNF_HAS_PRIVATE_PROBLEM (0x00004000)
设备报告PNP_DEVICE_FAILED而不报告
PNP_DEVICE_RESOURCE_REQUIREMENTS_CHANGED。

DNF_HARDWARE_VERIFICATION (0x00008000)
设备节点具有硬件验证。

DNF_DEVICE_GONE (0x00010000)
设备的 PDO 不再在IRP_QUERY_RELATIONS请求中返回。

DNF_LEGACY_RESOURCE_DEVICENODE (0x00020000)
为旧资源分配创建了设备节点。

DNF_NEEDS_REBALANCE (0x00040000)
设备节点已触发重新平衡。

DNF_LOCKED_FOR_EJECT (0x00080000)
设备正在弹出或与正在弹出的设备相关。

DNF_DRIVER_BLOCKED (0x00100000)
已阻止设备节点的一个或多个驱动程序加载。

DNF_CHILD_WITH_INVALID_ID (0x00200000)
设备节点的一个或多个子级具有无效的 ID。

DNF_ASYNC_START_NOT_SUPPORTED (0x00400000)
设备不支持异步启动。

DNF_ASYNC_ENUMERATION_NOT_SUPPORTED (0x00800000)
设备不支持异步枚举。

DNF_LOCKED_FOR_REBALANCE (0x01000000)
设备已锁定以重新平衡。

DNF_UNINSTALLED (0x02000000)
设备正在进行IRP_MN_QUERY_REMOVE_DEVICE请求。

DNF_NO_LOWER_DEVICE_FILTERS (0x04000000)
设备没有较低设备筛选器类型的注册表项。

DNF_NO_LOWER_CLASS_FILTERS (0x08000000)
设备没有较低类筛选器类型的注册表项。

DNF_NO_SERVICE (0x10000000)
设备的服务 没有注册表项。

DNF_NO_UPPER_DEVICE_FILTERS (0x20000000)
设备没有上层设备筛选器类型的注册表项。

DNF_NO_UPPER_CLASS_FILTERS (0x40000000)
设备没有上类筛选器类型的注册表项。

DNF_WAITING_FOR_FDO (0x80000000)
设备的枚举正在等待，直到驱动程序附加其 FDO。

设备管理器问题代码

项目 • 2023/06/15

当设备出现问题时，设备管理器使用黄色感叹号 (！) 标记设备。 问题代码的格式为 CM_PROB_*XXX* ，并在头文件 cfg.h 中定义。 此处介绍了最重要的内容，以及它们与 设备节点状态标志的映射。 有关更全面的列表，请参阅设备管理器错误消息。

代码 1 (CM_PROB_NOT_CONFIGURED)
指示设备未安装且以前未安装。 (对应于 DNF_NOT_CONFIGURED.)

代码 10 (CM_PROB_FAILED_START)
指示设备由于某种原因未启动，但 I/O 管理器尝试使用一组资源启动设备。 (对应于 DNF_START_FAILED.)

代码 12 (CM_PROB_NORMAL_CONFLICT)
指示没有足够的资源来启动此设备。 (对应于 DNF_INSUFFICIENT_RESOURCES.)

代码 14 (CM_PROB_NEED_RESTART)
指示用户模式重新配置了设备，需要重新启动才能使更改生效。 (对应于 DNF_NEED_RESTART.)

代码 18 (CM_PROB_REINSTALL)
指示设备需要安装并且以前已安装。 (对应于 DNF_REINSTALL.)

代码 21 (CM_PROB_WILL_BE_REMOVED)
指示用户模式卸载了此设备。 (对应于 DNF_WILL_BE_REMOVED.)

代码 22 (CM_PROB_DISABLED)
指示设备已禁用。 (对应于 DNF_DISABLED.)

代码 28 (CM_PROB_FAILED_INSTALL)
指示安装失败，并且没有为此设备选择驱动程序，尽管内核没有报告 (问题，也没有 DNF_XXX匹配此问题) 。 此问题可能是由于板载系统设备 (ISA 计时器、ISA RTC、RAM 内存等) 尚未包含 INF 文件的结果。

代码 31 (CM_PROB_FAILED_ADD)
指示未添加设备。 失败的原因可能包括：驱动程序的 **AddDevice** 例程返回错误，或者注册表中没有为设备列出的服务。 (对应于 DNF_ADD_FAILED.)

检查资源冲突

项目 • 2024/03/08

本部分讨论可用于检测资源冲突的方法。

第一种方法涉及转储仲裁程序数据。 以下示例检查 I/O 范围的仲裁程序数据：

dbgcmd

```
kd> !arbiter 1

DEVNODE ff0daf48
  Port Arbiter "RootPort" at 8045b920
    Allocated ranges:
      10 - 1f S         Owner ff0d6b30
      22 - 3f S         Owner ff0d6b30
      44 - 47 S         Owner ff0d6b30
      4c - 6f S         Owner ff0d6b30
      72 - 7f S         Owner ff0d6b30
      90 - 91 S         Owner ff0d6b30
      93 - 9f S         Owner ff0d6b30
      a2 - bf S         Owner ff0d6b30
      d0 - ef S         Owner ff0d6b30
      100 - 2f7 S       Owner ff0d6b30
      300 - cf7 S       Owner ff0d6b30
      d00 - ffff S      Owner ff0d6b30
    Possible allocation:
      < none >

    DEVNODE ff0d2e28 (PCI_HAL\PNP0A03\0)
      Port Arbiter "PCI I/O Port (b=00)" at e122c2c8
        Allocated ranges:
          0 - f         Owner 00000000
          20 - 21       Owner 00000000
          40 - 43       Owner 00000000
          48 - 4b       Owner 00000000
          60 - 60       Owner ff0d4030
          64 - 64       Owner ff0d4030
          70 - 71       Owner 00000000
          80 - 8f       Owner 00000000
          92 - 92       Owner 00000000
          a0 - a1       Owner 00000000
          c0 - cf       Owner 00000000
          f0 - ff       Owner 00000000
          170 - 177     Owner ff0cf030
          1ce - 1cf S   Owner ff040040
          2f8 - 2ff     Owner 00000000
          376 - 376     Owner ff0cf030
          378 - 37f     Owner ff0d4e70
          3b0 - 3bb S   Owner ff040040
 3c0 - 3cf S   Owner ff0bb900
```

```
          3c0 - 3df S   Owner ff040040
          3d4 - 3db S   Owner ff0bb900
          3de - 3df S   Owner ff0bb900
 3ec - 3ef     Owner ff0d0b50 (This device conflicts with another device,
see below)
          3f2 - 3f5     Owner ff0d4770
          3f7 - 3f7 S   Owner ff0d4770
          3f8 - 3ff     Owner ff0d4af0
          778 - 77b     Owner ff0d4e70
          cf8 - cff     Owner 00000000
          1000 - 10ff           Owner ff0d1030
          1400 - 140f           Owner ff0d1d30
          1410 - 141f           Owner ff0d1890
          10000 - ffffffffffffffff      Owner 00000000
        Possible allocation:
          < none >
```

请注意，有两个仲裁程序：一个位于设备树的根目录中，另一个位于 PCI_HAL。 另请注意，PCI 仲裁程序声明并预分配其仲裁的设备的范围（0xD000-0xFFFF，稍后由 PCI 仲裁程序为其设备进行子分配）。 “所有者”字段指示拥有该范围的设备对象。 “所有者”的值为零，指示范围不在总线上。 例如，对于 PCI 桥，不会传递的所有范围都将分配给 **NULL**。

在以下示例中，PCI 桥通过 I/O 0xD000-0xDFFFF，因此其仲裁程序将包含以下两个范围：

text

```
0-CFFFF            Owner 00000000
E0000-FFFFFFFFFFFFFFFF   Owner 00000000
```

FFFFFFFFFFFFFFFF 是因为所有仲裁资源都被视为 64 位范围。

示例：

dbgcmd

```
kd> !devobj ff0bb900

Device object (ff0bb900) is for:
 Video0 \Driver\mga_mil DriverObject ff0bbc10
Current Irp 00000000 RefCount 1 Type 00000023 Flags 0000204c
DevExt ff0bb9b8 DevNode ff0bb808
Device queue is not busy.
kd> !devnode ff0bb808

DevNode 0xff0bb808 for PDO 0xff0bb900 at level 0xffffffff
  Parent 0xff0daf48   Sibling 0000000000   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xffffffff
  TargetDeviceNotify List - f 0xff0bb86c  b 0xff0bb86c
```

```
  Flags (0x00000400)  DNF_RESOURCE_REPORTED
kd> !devnode ff0bb808 6

DevNode 0xff0bb808 for PDO 0xff0bb900 at level 0xffffffff
  Parent 0xff0daf48   Sibling 0000000000   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xffffffff
  TargetDeviceNotify List - f 0xff0bb86c  b 0xff0bb86c
  Flags (0x00000400)  DNF_RESOURCE_REPORTED
  CmResourceList at 0xe12474e8  Version 0.0  Interface 0x5  Bus #0
    Entry 0 - Port (0x1) Shared (0x3)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x3c0 for 0x10 bytes
    Entry 1 - Port (0x1) Shared (0x3)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x3d4 for 0x8 bytes
    Entry 2 - Port (0x1) Shared (0x3)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x3de for 0x2 bytes
    Entry 3 - Memory (0x3) Device Exclusive (0x1)
      Flags (0000) - READ_WRITE
      Range starts at 0x0000000040000000 for 0x4000 bytes
    Entry 4 - Memory (0x3) Device Exclusive (0x1)
      Flags (0000) - READ_WRITE
      Range starts at 0x0000000040800000 for 0x800000 bytes
```

如示例所示，此操作检索到拥有范围 3c0-3cf 的传统视频卡。 同一设备对象在其拥有的其他范围（3de-3df 和 3d4-3dc）附近列出。 使用相同的跟踪方法，3f8-3ff 的范围确定为串行端口使用的范围。

转换中断需要类似的方法：

dbgcmd

```
kd> !arbiter 4

DEVNODE ff0daf48
  Interrupt Arbiter "RootIRQ" at 8045bae0
    Allocated ranges:
      31 - 31           Owner ff0d4030
      34 - 34 S         Owner ff0d4af0
      36 - 36           Owner ff0d4770
      3b - 3b S         Owner ff0d1030
      3b - 3b S         Owner ff0d1d30
      3c - 3c           Owner ff0d3c70
 3f - 3f           Owner ff0cf030
    Possible allocation:
      < none >
```

请注意，中断只有一个仲裁程序：根仲裁程序。

例如，将中断 3F 转换为 IRQ。 首先转储设备对象，然后转储 devnode：

dbgcmd

```
kd> !devobj ff0cf030

Device object (ff0cf030) is for:
 IdeFdoff0d0398Channel1 \Driver\IntelIde DriverObject ff0d0530
Current Irp 00000000 RefCount 0 Type 00000004 Flags 00001040AttachedDev
ff0cd030

DevExt ff0cf0e8 DevNode ff0cfe88
Device queue is not busy.
kd> !devnode ff0cfe88 6

DevNode 0xff0cfe88 for PDO 0xff0cf030 at level 0x3
  Parent 0xff0d1348   Sibling 0000000000   Child 0xff0c84a8
  InterfaceType 0xffffffff  Bus Number 0xfffffff0
  InstancePath is "PCIIDE\IDEChannel\1&1"
  ServiceName is "atapi"
  TargetDeviceNotify List - f 0xff0cfeec  b 0xff0cfeec
  Flags (0x6000120b)  DNF_PROCESSED, DNF_STARTED,
                      DNF_ENUMERATED, DNF_RESOURCE_ASSIGNED,
                      DNF_ADDED, DNF_HAS_BOOT_CONFIG
                      Unknown flags 0x40000000
  CmResourceList at 0xe12321c8  Version 0.0  Interface 0x1  Bus #0
    Entry 0 - Port (0x1) Device Exclusive (0x1)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x170 for 0x8 bytes
    Entry 1 - Port (0x1) Device Exclusive (0x1)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x376 for 0x1 bytes
    Entry 2 - Interrupt (0x2) Device Exclusive (0x1)
      Flags (0x01) - LATCHED
      Level 0xf, Vector 0xf, Affinity 0xffffffff

  IoResList at 0xe12363c8 : Interface 0x1  Bus 0  Slot 0
  Reserved Values = {0x0002e0d0, 0x00920092, 0xe1235508}
    Alternative 0 (Version 1.1)
      Preferred Descriptor 0 - NonArbitrated/ConfigData (0x80) Shared (0x3)
        Flags (0000) -
        Data:              : 0x1 0x61004d 0x680063
      Preferred Descriptor 1 - Port (0x1) Device Exclusive (0x1)
        Flags (0x01) - PORT_IO
        0x000008 byte range with alignment 0x000001
        170 - 0x177
      Preferred Descriptor 2 - Port (0x1) Device Exclusive (0x1)
        Flags (0x01) - PORT_IO
        0x000001 byte range with alignment 0x000001
        376 - 0x376
      Preferred Descriptor 3 - Interrupt (0x2) Device Exclusive (0x1)
        Flags (0x01) - LATCHED
        0xf - 0xf
    Alternative 1 (Version 1.1)
      Preferred Descriptor 0 - Port (0x1) Device Exclusive (0x1)
        Flags (0x01) - PORT_IO
```

```
        0x000008 byte range with alignment 0x000001
        170 - 0x177
      Preferred Descriptor 1 - Port (0x1) Device Exclusive (0x1)
        Flags (0x01) - PORT_IO
        0x000001 byte range with alignment 0x000001
        376 - 0x376
      Preferred Descriptor 2 - Interrupt (0x2) Device Exclusive (0x1)
        Flags (0x01) - LATCHED
        0xf - 0xf
```

例如，从 **devnode** 开始，尝试确定是否存在导致此设备无法启动的资源冲突：

dbgcmd

```
kd> !devnode 0xff0d4bc8 6

DevNode 0xff0d4bc8 for PDO 0xff0d4cb0 at level 0
  Parent 0xff0daf48   Sibling 0xff0d4a08   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xffffffff
  InstancePath is "Root\*PNP0501\1_0_17_2_0_0"
  ServiceName is "Serial"
  TargetDeviceNotify List - f 0xff0d4c2c  b 0xff0d4c2c
  Flags (0x60001129)  DNF_PROCESSED, DNF_ENUMERATED,
                      DNF_MADEUP, DNF_INSUFFICIENT_RESOURCES,
                      DNF_ADDED, DNF_HAS_BOOT_CONFIG
                      Unknown flags 0x40000000

  IoResList at 0xe1251e28 : Interface 0x1  Bus 0  Slot 0
  Reserved Values = {0x0043005c, 0x006e006f, 0x00720074}
    Alternative 0 (Version 1.1)
      Preferred Descriptor 0 - NonArbitrated/ConfigData (0x80) Undetermined
Shar
ing (0)
        Flags (0000) -
        Data:              : 0xc000 0x0 0x0
      Preferred Descriptor 1 - Port (0x1) Undetermined Sharing (0)
        Flags (0x05) - PORT_IO 10_BIT_DECODE
        0x000008 byte range with alignment 0x000001
        3e8 - 0x3ef
      Preferred Descriptor 2 - Interrupt (0x2) Shared (0x3)
        Flags (0x01) - LATCHED
        0x5 - 0x5
    Alternative 1 (Version 1.1)
      Preferred Descriptor 0 - NonArbitrated/ConfigData (0x80) Undetermined
Shar
ing (0)
        Flags (0000) -
        Data:              : 0xc000 0x0 0x0
      Preferred Descriptor 1 - Port (0x1) Undetermined Sharing (0)
        Flags (0x05) - PORT_IO 10_BIT_DECODE
        0x000008 byte range with alignment 0x000008
        3e8 - 0x3ef
      Preferred Descriptor 2 - Interrupt (0x2) Shared (0x3)
```

```
        Flags (0x01) - LATCHED
        0x5 - 0x5
```

首先，假设这是一个 I/O 冲突，并转储仲裁程序（请参阅前面的示例）。 结果表明，范围 0x3EC-0x3EF 由 0xFF0D0B50 所有，这与串行设备的资源请求重叠。 接下来，为该范围的所有者转储设备对象，然后为所有者转储 devnode：

dbgcmd

```
kd> !devobj ff0d0b50

Device object (ff0d0b50) is for:
 Resource00413e \Driver\isapnp DriverObject ff0d0e10
Current Irp 00000000 RefCount 0 Type 00000004 Flags 00001040
DevExt ff0d0c08 DevNode ff0d0a68
Device queue is not busy.
kd> !devnode ff0d0a68 6

DevNode 0xff0d0a68 for PDO 0xff0d0b50 at level 0xffffffff
  Parent 0xff0daf48   Sibling 0000000000   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xffffffff
Duplicate PDO 0xff0d0e10  TargetDeviceNotify List - f 0xff0d0acc  b
0xff0d0acc
  Flags (0x00000421)  DNF_PROCESSED, DNF_MADEUP,
                      DNF_RESOURCE_REPORTED
  CmResourceList at 0xe1233628  Version 0.0  Interface 0x1  Bus #0
    Entry 0 - Port (0x1) Device Exclusive (0x1)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0x3ec for 0x4 bytes
```

这是一个“伪 devnode”，对应于 ISAPNP 驱动程序为其读取数据端口分配的范围。

若要确定 PnP 管理器在尝试启动设备时分配给特定设备的资源，请执行以下操作：

1. 在驱动程序收到 IRP_MN_START_DEVICE 请求时调用的例程上放置断点。 还可以在驱动程序的调度例程上放置断点（如果知道其名称）。 在这两种情况下，都应加载驱动程序及其符号。 这可能需要设置初始断点。

   例如，对于 PCMCIA，可以在 **pcmcia!pcmciastartpccard** 上设置断点。 使用此特定例程的优点在于，第二个参数是 CM 资源列表，可以使用 **!cmreslist** 进行转储（取消步骤 3）。 请参阅以下 PCMCIA 示例。

2. 确定需要关注的设备后，转储设备对象（如果尚未这样做），然后使用 CM 资源列表转储 devnode。 检查已将哪些资源分配给设备。 还可以检查这些资源是否是 I/O 资源列表的子集。 然后键入 **g** 或通过该过程执行单个步骤，并确定设备是否已启动以及分配了哪些资源。 如果向设备提供了一组启动资源，但未能启动，则驱动程序

可能无法正常运行（例如，如果设备错误地声明可以使用一组实际无法使用的资源）。

示例：

dbgcmd

```
ntoskrnl!IopStartDevice:
80420212 55              push    ebp
kd> kb
ChildEBP RetAddr  Args to Child
f64138c0 8048b640 ff0ce870 ff0d7c08 ff0cde88 ntoskrnl!IopStartDevice
(!devobj ff0ce870)
f64138f0 8048d8e7 ff0cde88 f6413978 ff0cddc8
ntoskrnl!IopStartAndEnumerateDevice
+0x1a
f6413900 8048de7f ff0cde88 f6413978 ff0cf448
ntoskrnl!IopProcessStartDevicesWork
er+0x43
f6413910 8048d8d5 ff0cf448 8048d8a4 f6413978
ntoskrnl!IopForAllChildDeviceNodes+
0x1f
f6413924 8048de7f ff0cf448 f6413978 ff0d3f48
ntoskrnl!IopProcessStartDevicesWork
er+0x31
f6413934 8048d8d5 ff0d3f48 8048d8a4 f6413978
ntoskrnl!IopForAllChildDeviceNodes+
0x1f
f6413948 8048d893 ff0d3f48 f6413978 e12052e8
ntoskrnl!IopProcessStartDevicesWork
er+0x31
f641395c 804f6f1b ff0d7c08 f6413978 8045c520
ntoskrnl!IopProcessStartDevices+0x1
f
f64139d0 804f5cc1 80088000 f6413aec 8045bba0
ntoskrnl!IopInitializeBootDrivers+0
x2f9
f6413b24 804f4db3 80088000 00000001 00000000 ntoskrnl!IoInitSystem+0x3a6
f6413da8 80447610 80088000 00000000 00000000
ntoskrnl!Phase1Initialization+0x6a3

f6413ddc 8045375a 804f4710 80088000 00000000
ntoskrnl!PspSystemThreadStartup+0x5
4
00000000 00000000 00000000 00000000 00000000 ntoskrnl!KiThreadStartup+0x16
kd> !devobj ff0ce870

Device object (ff0ce870) is for:
 NTPNP_PCI0002 \Driver\PCI DriverObject ff0ceef0
Current Irp 00000000 RefCount 0 Type 00000022 Flags 00001040AttachedDev
ff0cb9e0

DevExt ff0ce928 DevNode ff0cde88
```

```
Device queue is not busy.
kd> !devnode ff0cde88 6

DevNode 0xff0cde88 for PDO 0xff0ce870 at level 0x2
  Parent 0xff0cf448   Sibling 0xff0cddc8   Child 0000000000
  InterfaceType 0xffffffff  Bus Number 0xffffffff
  InstancePath is "PCI\VEN_8086&DEV_7010\0&69"
  ServiceName is "intelide"
  TargetDeviceNotify List - f 0xff0cdeec  b 0xff0cdeec
  Flags (0x00001209)  DNF_PROCESSED, DNF_ENUMERATED,
                      DNF_RESOURCE_ASSIGNED, DNF_ADDED
 (note that the device is not yet started)
  CmResourceList at 0xe120fce8  Version 0.0  Interface 0x5  Bus #0
    Entry 0 - Port (0x1) Device Exclusive (0x1)
      Flags (0x01) - PORT_MEMORY PORT_IO
      Range starts at 0xfff0 for 0x10 bytes (these are the resources used:
0xfff0-0xffff)
    Entry 1 - DevicePrivate (0x81) Device Exclusive (0x1)
      Flags (0000) -
      Data - {0x00000001, 0x00000004, 0000000000}

  IoResList at 0xe120df88 : Interface 0x5  Bus 0  Slot 0x2d
    Alternative 0 (Version 1.1)
      Descriptor 0 - Port (0x1) Device Exclusive (0x1)
        Flags (0x01) - PORT_IO
        0x000010 byte range with alignment 0x000010
 0 - 0xffff     (it could have used any 16 bytes that are 16-byte aligned
between 0 and 0xffff)
      Descriptor 1 - DevicePrivate (0x81) Device Exclusive (0x1)
        Flags (0000) -
        Data:              : 0x1 0x4 0x0
```

PCMCIA 的示例：

dbgcmd

```
kd> bp pcmcia!pcmciastartpccard
Loading symbols for 0x8039d000       pcmcia.sys ->   pcmcia.sys
kd> kb
Loading symbols for 0x80241000         ndis.sys ->   ndis.sys
ChildEBP RetAddr  Args to Child
f6413814 803a7cbd ff0d0c30 e11d8808 ff0d0c30 pcmcia!PcmciaStartPcCard
f6413838 803a3798 ff0d0c30 ff0d1500 ff0d1588
pcmcia!PcmciaPdoPnpDispatch+0x169
f6413848 80418641 ff0d0c30 ff0d1588 00000000 pcmcia!PcmciaDispatch+0x3a
f641385c 802455cf ff0d1614 ff0d1638 00040000 ntoskrnl!IofCallDriver+0x35
f641387c 802497cf ff0d1588 ff0d0c30 ff0c8210
NDIS!ndisPassIrpDownTheStack+0x3b
f64138ac 80418641 ff0c8210 ff0d161c ff0d1640 NDIS!ndisPnPDispatch+0x1f9
f64138c0 8048de68 ff0c3508 ff0d16a8 00000000 ntoskrnl!IofCallDriver+0x35
f64138d4 8041ff5e ff0c8210 f64138f8 ff0c3508
ntoskrnl!IopAsynchronousCall+0x90
f6413920 8048b4ae ff0d0c30 ff0e0a68 ff0d16a8 ntoskrnl!IopStartDevice+0x76
```

```
f6413950 8048d707 ff0d16a8 f64139fc 00000000
ntoskrnl!IopStartAndEnumerateDevice+0x1a
f6413960 8048dc9f ff0d16a8 f64139fc ff0d1e88
ntoskrnl!IopProcessStartDevicesWorker+0x43
f6413970 8048d6f5 ff0d1e88 8048d6c4 f64139fc
ntoskrnl!IopForAllChildDeviceNodes+0x1f
f6413984 8048dc9f ff0d1e88 f64139fc ff0d3268
ntoskrnl!IopProcessStartDevicesWorker+0x31
f6413994 8048d6f5 ff0d3268 8048d6c4 f64139fc
ntoskrnl!IopForAllChildDeviceNodes+0x1f
f64139a8 8048dc9f ff0d3268 f64139fc ff0d7b28
ntoskrnl!IopProcessStartDevicesWorker+0x31
f64139b8 8048d6f5 ff0d7b28 8048d6c4 f64139fc
ntoskrnl!IopForAllChildDeviceNodes+0x1f
f64139cc 8048d6b3 ff0d7b28 f64139fc 80087000
ntoskrnl!IopProcessStartDevicesWorker+0x31
f64139e0 804f6c97 ff0e0a68 f64139fc 8045c140
ntoskrnl!IopProcessStartDevices+0x1f
f6413a30 804f5601 000001e0 80087000 00000000
ntoskrnl!IopInitializeSystemDrivers+0x5b
f6413b7c 804f4820 80087000 00000001 00000000 ntoskrnl!IoInitSystem+0x3fe
kd> !cmreslist e11d8808

CmResourceList at 0xe11d8808  Version 0.0  Interface 0x1  Bus #0
  Entry 0 - Interrupt (0x2) Device Exclusive (0x1)
    Flags (0x01) - LATCHED
    Level 0x9, Vector 0x9, Affinity 0xffffffff
  Entry 1 - Port (0x1) Device Exclusive (0x1)
    Flags (0x01) - PORT_MEMORY PORT_IO
    Range starts at 0xdfe0 for 0x20 bytes (started with IRQ 9, IO dfe0-dfff)
  Entry 2 - DevicePrivate (0x81) Device Exclusive (0x1)
    Flags (0000) -
    Data - {0x00010120, 0000000000, 0000000000}

kd> g
```

调试服务应用程序

项目 • 2023/06/15

服务（也称为 *Windows 服务*）是一种用户模式进程，设计为由 Windows 启动，无需人工交互。 它在系统启动时自动启动，或者由使用 Win32 API 中包含的服务功能的应用程序启动。 用户还可以通过“服务”控制面板实用工具启动服务。 每个服务都必须符合服务控制管理器 (SCM) 的接口规则。

每个服务由三个元素组成： *服务应用程序*、 *服务控制程序和*服务控制管理器本身。 尽管服务应用程序有时 (错误) 称为“服务”，但它实际上是构成服务的三个组件之一。 服务应用程序几乎可以包含任何类型的用户模式代码。 服务控制程序控制服务应用程序何时启动和停止。 服务控制管理器是 Windows 的一部分。

以下部分介绍如何调试服务应用程序：

选择最佳方法

准备调试服务应用程序

自动调试服务应用程序

手动调试服务应用程序

有关服务、服务应用程序和服务控制管理器的概述，请参阅 *Microsoft Windows 内部：Microsoft Windows Server 2003*、*Windows XP 和 Windows 2000* 由 David A. Solomon 和 Mark E. Russinovich (第 4 版，Microsoft Press，2005) 。

选择最佳方法

项目 • 2024/03/04

调试服务应用程序有几种不同的方法。 为了选择正确的方法，必须首先做出两个选择：调试器附加到服务应用程序的时间和要使用的调试配置。

调试器可以在三个阶段附加到服务应用程序：

- 服务启动的最初阶段。 调试器在服务启动时自动启动。 如果要调试服务的初始化代码，请选择此选项。
- 服务第一次遇到异常时。 当发生异常或崩溃或服务应用程序调用 **DebugBreak** 函数时，调试器会自动启动。 如果希望调试器在遇到问题时出现，而不是在遇到问题之前出现，请选择此选项。
- 服务正常运行后。 可以随时手动将调试器附加到已在运行的服务。 如果不想提前为调试做准备，请选择此选项。

可以选择三种调试配置：

- 本地调试。 单个调试器，与服务在同一台计算机上运行。
- 远程调试。 与服务在同一台计算机上运行的用户模式调试器，由运行在第二台计算机上的调试客户端控制。
- 内核控制的用户模式调试。 与服务在同一台计算机上运行的用户模式调试器，由第二台计算机上的内核调试器控制。

如果服务在 Windows 2000、Windows XP 或 Windows Server 2003 上运行，则可以将这三个附加选项中的任何一个与这三个调试配置选项中的任意一个组合使用。

如果服务在 Windows Vista 或更高版本的 Windows 上运行，则对如何组合这些选项有一个限制。 如果要从服务启动之初或遇到异常时开始进行调试，则必须使用远程调试或内核控制的用户模式调试。

换句话说，在Windows Vista及更高版本上，除非计划在服务已经运行后手动附加调试器，否则无法使用本地调试。 此限制是由于以下事实造成的：在这些版本的 Windows 中，服务在会话 0 中运行，并且自动启动并附加到服务的调试器也在会话 0 中，并且在运行服务的计算机上没有用户界面。

准备调试服务应用程序

项目 • 2023/06/15

本主题列出了在调试服务应用程序之前可能需要的所有准备步骤。 方案中需要执行哪些步骤取决于你选择的附加选项和你选择的调试配置。 有关这些选项的列表，请参阅 选择最佳方法。

本主题中所述的每个准备步骤都指定了需要它的条件。 这些步骤可以按任意顺序完成。

启用初始化代码的调试

如果计划从执行开始调试服务应用程序（包括其初始化代码），则需要执行此准备步骤。

找到或创建以下注册表项，其中 *ProgramName* 是服务应用程序的可执行文件的名称：

```text
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\ProgramName
```

ProgramName 应包含文件扩展名，但不包括路径。 例如， *ProgramName* 可能 Myservice.exe或Thisservice.dll。

在此注册表项下，创建名为 **Debugger** 的字符串数据值。 应将此字符串的值设置为要附加到服务应用程序的调试器的完整路径和文件名。

- 如果计划在本地调试，请使用如下所示的字符串：

  ```
  控制台
  c:\Debuggers\windbg.exe
  ```

  如果运行的是 Windows Vista 或更高版本的 Windows，请不要选择此选项。

- 如果打算使用远程调试，请使用 -noio 选项指定 NTSD。 这会导致 NTSD 在没有自己的任何控制台的情况下运行，只能通过远程连接访问。 例如：

  ```
  控制台
  c:\Debuggers\ntsd.exe -server ServerTransport -noio -y SymbolPath
  ```

  如果调试会话在 Windows 完全加载之前开始，则可能无法从远程共享访问符号;在这种情况下，必须使用本地符号。 *ServerTransport* 必须指定由 Windows 内核实现

的传输协议，而不与用户模式服务（如 TCP 或 NPIPE）交互。 有关 *ServerTransport* 的语法，请参阅 **激活调试服务器**。

- 如果计划从内核模式调试器控制用户模式调试器，请使用 -d 选项指定 NTSD。 例如：

  控制台

  ```
  c:\Debuggers\ntsd.exe -d -y SymbolPath
  ```

  如果计划使用此方法，并且将从符号服务器访问用户模式符号，则应将此方法与远程调试相结合。 在这种情况下，请使用 -ddefer 选项指定 NTSD。 选择由 Windows 内核实现且不与用户模式服务（如 TCP 或 NPIPE）交互的传输协议。 例如：

  控制台

  ```
  c:\Debuggers\ntsd.exe -server ServerTransport -ddefer -y SymbolPath
  ```

  有关详细信息，请参阅 从内核调试器控制 User-Mode 调试器。

完成此注册表编辑后，每当启动或重启具有此名称的服务时，将启动调试器。

使服务应用程序能够中断调试器

如果希望服务应用程序在崩溃或遇到异常时中断调试器，则需要执行此准备步骤。 如果希望服务应用程序通过调用 **DebugBreak** 函数来中断调试器，则还需要执行此步骤。

注意 如果已启用初始化代码调试 (“启用初始化代码调试”) 子节中所述的步骤，则应跳过此步骤。 启用初始化代码调试后，调试器会在启动时附加到服务应用程序，这会导致所有崩溃、异常和对 **DebugBreak** 的调用路由到调试器，而无需进行其他准备。

此准备步骤涉及将所选调试器注册为事后调试器。 这是通过使用调试器命令行上的 -iae 或 -iaec 选项完成的。 建议使用以下命令，但如果要更改它们，请参阅 启用事后调试中的语法详细信息。

- 如果计划在本地调试，请使用如下命令：

  控制台

  ```
  windbg -iae
  ```

  如果运行的是 Windows Vista 或更高版本的 Windows，请不要选择此选项。

- 如果打算使用远程调试，请使用 -noio 选项指定 NTSD。 这会导致 NTSD 在没有自己的任何控制台的情况下运行，只能通过远程连接访问。 若要安装包含 -server 参数的事后调试器，必须手动编辑注册表;有关详细信息，请参阅 启用事后调试。 例如，**AeDebug** 键的**调试器**值可能如下所示：

  控制台

  ```
  ntsd -server npipe:pipe=myproc%x -noio -p %ld -e %ld -g -y SymbolPath
  ```

  在管道规范中， **%x** 令牌将替换为启动调试器的进程的进程 ID。 这可以保证，如果多个进程启动事后调试器，则每个进程都具有唯一的管道名称。 如果调试会话在 Windows 完全加载之前开始，则可能无法从远程共享访问符号;在这种情况下，必须使用本地符号。 *ServerTransport* 必须指定由 Windows 内核实现的传输协议，而不与用户模式服务（如 TCP 或 NPIPE）交互。 有关 *ServerTransport* 的语法，请参阅 **激活调试服务器**。

- 如果计划从内核模式调试器控制用户模式调试器，请使用 -d 选项指定 NTSD。 例如：

  控制台

  ```
  ntsd -iaec -d -y SymbolPath
  ```

  如果选择此方法并打算从符号服务器访问用户模式符号，则应将此方法与远程调试相结合。 在这种情况下，请使用 -ddefer 选项指定 NTSD。 选择由 Windows 内核实现且不与用户模式服务（如 TCP 或 NPIPE）交互的传输协议。 若要安装包含 -server 参数的事后调试器，必须手动编辑注册表;有关详细信息，请参阅 启用事后调试。 例如，**AeDebug** 键的**调试器**值可能如下所示：

  控制台

  ```
  ntsd -server npipe:pipe=myproc%x -ddefer -p %ld -e %ld -g -y SymbolPath
  ```

  有关详细信息，请参阅 从内核调试器控制 User-Mode 调试器。

发出其中一个命令时，将注册事后调试器。 每当任何用户模式程序（包括服务应用程序）遇到异常或运行 **DebugBreak** 函数时，都会启动此调试器。

调整服务应用程序超时

如果计划在服务启动时或遇到异常) 时自动 (启动调试器，则需要执行此准备步骤。

找到以下注册表项：

```text
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control
```

在此键下，找到或创建名为 **ServicesPipeTimeout** 的 DWORD 数据值。 将此条目设置为希望服务在超时之前等待的时间量（以毫秒为单位）。例如，值 60，000 是 1 分钟，而值 86，400，000 是 24 小时。 如果未设置此注册表值，则默认超时约为 30 秒。

此值的意义在于时钟在启动每个服务时开始运行，当达到超时值时，附加到服务的任何调试器都将终止。 因此，所选值应长于服务启动和调试会话完成之间的总时间。

此设置适用于在注册表编辑完成后启动或重启的每个服务。 如果某些服务崩溃或挂起，并且此设置仍然有效，则 Windows 不会检测到该问题。 因此，应仅在调试时使用此设置，并在调试完成后将注册表项返回到其原始值。

隔离服务

有时，多个服务组合在单个服务主机 (Svchost) 进程中。 如果要调试此类服务，必须先将其隔离到单独的 Svchost 进程中。

有三种方法可用于隔离服务。 Microsoft 建议将服务移动到其自己的组方法，如下所示。 (更改服务类型和复制 SvcHost 二进制) 的替代方法可以临时用于调试，但由于它们更改了服务的运行方式，因此它们不如第一种方法可靠。

首选方法：将服务移动到其自己的组

1. 发出以下服务配置工具 (Sc.exe) 命令，其中 *ServiceName* 是服务的名称：

   ```控制台
   sc qc ServiceName
   ```

   这会显示服务的当前配置值。 感兴趣的值为 BINARY_PATH_NAME，它指定用于启动服务控制程序的命令行。 在此方案中，由于服务尚未隔离，因此此命令行包括目录路径、Svchost.exe和一些 SvcHost 参数（包括 -k 开关），后跟组名。 例如，它可能如下所示：

   ```控制台
   %SystemRoot%\System32\svchost.exe -k LocalServiceNoNetwork
   ```

   请记住此路径和组名称;它们在步骤 5 和 6 中使用。

2. 找到以下注册表项：

```
text

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\SvcHost
```

创建具有唯一名称的新REG_MULTI_SZ值 (例如 **TempGrp**) 。

3. 将此新值设置为要隔离的服务的名称。 请勿包含任何目录路径或文件扩展名。 例如，可以将新值 **TempGrp** 设置为等于 **MyService**。

4. 在同一注册表项下，创建一个与步骤 2 中使用的名称相同的新密钥。 例如：

```
text

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\SvcHost\
TempGrp
```

现在，SvcHost 键包含一个具有新名称的值，并且还有一个同名的从属键。

5. 查找与步骤 1 中找到的组同名的 SvcHost 密钥下的另一个密钥。 如果存在此类键，请检查其中的所有值，并在步骤 4 中创建的新密钥中创建它们的副本。

例如，旧密钥可能命名为：

```
text

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\SvcHost\
LocalServiceNoNetwork
```

它可能包含 **CoInitializeSecurityParam**、 **AuthenticationCapabilities** 等值。 将转到新创建的密钥：

```
text

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\SvcHost\
TempGrp
```

并在 其中创建名称、类型和数据与旧键中的值相同。

如果旧密钥不存在，则无需创建新密钥。

6. 使用以下服务配置工具命令修改步骤 1 中找到的路径：

```
控制台
```

```
sc config ServiceName binPath= "RevisedPath"
```

在此命令中， *ServiceName* 是服务的名称， *而 RevisedPath* 是你为 BINARY_PATH_NAME提供的新值。 对于 *RevisedPath*，请使用与步骤 1 中显示的路径完全相同的路径，包括该行上显示的所有选项，只进行一次更改：将 -k 开关后面的 参数替换为在步骤 2 中创建的新注册表值的名称。 将 *RevisedPath* 括在引号中。 需要等号后的空格。

例如，命令可能如下所示：

控制台

```
sc config MyService binPath= "%SystemRoot%\System32\svchost.exe -k
TempGrp"
```

可能需要再次使用 **sc qc** 命令来查看所做的更改。

这些设置将在下次启动服务时生效。 若要清除旧服务的影响，建议重启 Windows，而不只是重启服务。

完成调试后，如果要将此服务返回到共享服务主机，请再次使用 **sc config** 命令将二进制路径返回到其原始值，并删除创建的新注册表项和值。

替代方法：更改服务类型

1. 发出以下服务配置工具 (Sc.exe) 命令，其中 *ServiceName* 是服务的名称：

   控制台

   ```
   sc config ServiceName type= own
   ```

   需要等号后的空格。

2. 使用以下命令重启服务：

   控制台

   ```
   net stop ServiceName
   net start ServiceName
   ```

不建议使用此替代方法，因为它可以更改服务的行为。 如果确实使用此方法，请在完成调试后使用以下命令还原正常行为：

控制台

```
sc config ServiceName type= share
```

替代方法：复制 SvcHost 二进制文件

1. Svchost.exe可执行文件位于 Windows 的 system32 目录中。 创建此文件的副本，将其命名为svhost2.exe，并将其放置在 system32 目录中。

2. 发出以下服务配置工具 (Sc.exe) 命令，其中 *ServiceName* 是服务的名称：

   控制台

   ```
   sc qc ServiceName
   ```

   此命令显示服务的当前配置值。 感兴趣的值为 BINARY_PATH_NAME，它指定用于启动服务控制程序的命令行。 在这种情况下，由于服务尚未隔离，因此此命令行将包含目录路径、Svchost.exe以及一些 SvcHost 参数。 例如，它可能如下所示：

   控制台

   ```
   %SystemRoot%\System32\svchost.exe -k LocalServiceNoNetwork
   ```

3. 若要修改此路径，请发出以下命令：

   控制台

   ```
   sc config ServiceName binPath= "RevisedPath"
   ```

   在此命令中， *ServiceName* 是服务的名称， *而 RevisedPath* 是你为BINARY_PATH_NAME提供的新值。 对于 *RevisedPath*，请使用与步骤 2 中显示的路径完全相同的路径，包括该行上显示的所有选项，仅进行一项更改：将 Svchost.exe 替换为 Svchost2.exe。 将 *RevisedPath* 括在引号中。 需要等号后的空格。

   例如，命令可能如下所示：

   控制台

   ```
   sc config MyService binPath= "%SystemRoot%\System32\svchost2.exe -k
   LocalServiceNoNetwork"
   ```

   可以再次使用 **sc qc** 命令查看所做的更改。

4. 使用以下命令重启服务：

   控制台

```
net stop ServiceName
net start ServiceName
```

不建议使用此替代方法，因为它可以更改服务的行为。 如果确实使用此方法，请在完成调试后使用 **sc config** 命令将路径更改回其原始值。

自动调试服务应用程序

项目 • 2023/10/26

服务应用程序启动时，可以自动启动调试器。 或者，当服务应用程序遇到异常或执行 **DebugBreak** 命令时，它可以自动启动。 如果选择了其中一种方法，本主题将介绍如何继续操作。 如果不确定要选择哪种方法，请参阅 选择最佳方法。

然后执行以下步骤：

1. 执行以下准备步骤之一：

   - 如果计划从一开始就调试服务应用程序，包括其初始化代码，请按照 启用初始化代码调试中所述的过程进行操作。 或者，如果希望服务应用程序在调试器崩溃或遇到异常时闯入调试器，请按照 启用服务应用程序以中断调试器中所述的过程进行操作。
   - 若要确保服务应用程序允许调试器正确运行，请执行 调整服务应用程序超时中所述的过程。
   - 如果服务与单个 SvcHost 进程中的其他服务结合使用，请执行隔离服务中所述的过程。

2. 如果服务已在运行，则必须重新启动该服务，这些更改才能生效。 建议重启 Windows 本身，以删除正在运行的服务的任何影响。 如果不想重启 Windows，请使用以下命令，其中 *ServiceName* 是服务的名称：

   控制台

   ```
   net stop ServiceName
   net start ServiceName
   ```

3. 如果选择调试服务应用程序的初始化代码，则服务启动时，将启动调试器并附加到服务应用程序。

   如果选择允许调试器由异常触发，则服务应用程序将正常执行，直到遇到异常或执行 **DebugBreak** 函数。 此时，调试器将启动并附加到服务应用程序。

4. 下一步取决于在步骤 1 中指定的调试器命令行：

   - 如果指定了不带任何远程处理选项的调试器，则会启动此调试程序，并且其窗口变为可见。
   - 如果使用 -server 和 -noio 选项指定 NTSD，则 NTSD 将在没有控制台窗口的情况下启动。 然后，可以通过使用 -remote 参数启动任何用户模式调试器，从另一台计算机连接到调试会话。 有关说明，请参阅 **激活调试客户端**。

- 如果使用 -d 选项指定了 NTSD，则 NTSD 将在没有控制台窗口的情况下启动。 然后，可以使用在另一台计算机上运行的内核调试器连接到调试会话。 有关说明，请参阅 从内核调试器控制 User-Mode 调试器。
- 如果使用 -ddefer 和 -server 选项指定 NTSD，则 NTSD 将在没有控制台窗口的情况下启动。 然后，可以使用内核调试器和用户模式远程调试器连接到调试会话，该调试器在与服务 (不同的计算机上运行，但可能与彼此) 相同的计算机。 有关说明，请参阅 将此方法与远程调试相结合。

5. 调试器启动时，服务在初始进程断点、异常或 **DebugBreak** 命令处暂停。 这使你可以检查服务应用程序的当前状态、设置断点，并做出任何其他所需的配置选择。

6. 使用 **g (Go)** 或其他执行命令继续执行服务应用程序。

另请参阅

DebugBreak 函数

手动调试服务应用程序

项目 • 2023/06/15

启动服务应用程序后手动附加到服务应用程序与调试任何正在运行的用户模式进程非常类似。

使用带 **/s 选项的**TList 工具可显示每个正在运行的进程 (PID) 以及每个进程中处于活动状态的服务的进程 ID。

如果要调试的服务应用程序与单个进程中的其他服务组合在一起，则必须在调试之前将其隔离。 为此，请执行隔离服务中所述的过程。 在此过程结束时，重启服务。

若要确定服务的新 PID，请发出以下服务配置工具 (Sc.exe) 命令，其中 *ServiceName* 是服务的名称：

控制台

```
sc queryex ServiceName
```

现在，使用此服务应用程序作为目标启动 WinDbg 或 CDB。 有三种方法可以做到这一点：使用 -p 选项指定 PID、使用 -pn 选项指定可执行文件名称（如果可执行文件名称）是唯一的 () ，或者使用 -psn 选项指定服务名称。

例如，如果进程SpoolSv.exe的 PID 为 651，并且包含名为 *Spooler 的服务*，则以下三个命令等效：

控制台

```
windbg -p 651 [AdditionalOptions]
windbg -pn spoolsv.exe [AdditionalOptions]
windbg -psn spooler [AdditionalOptions]
```

调试器启动后，像在任何其他用户模式调试会话中一样继续。

Windows 调试的符号

项目 • 2024/08/26

符号文件包含大量的数据，这些数据在运行二进制文件时实际上并不需要，但在调试过程中很有用。

符号可以包括名称、类型（如果适用）、存储地址或寄存器以及任何父符号或子符号。 符号示例包括变量名（本地和全局）、函数以及模块的任何入口点。

调试器从位于本地文件系统的或从远程符号服务器加载的符号文件中获取其有关符号的信息。 使用符号服务器时，调试器将自动使用正确的符号文件版本来匹配目标中的模块。

可以通过 Internet 从公共符号服务器获取 Windows 调试程序的符号（WinDbg、KD、CDB、NTSD）。

> **提示**
>
> 可以使用 **.symfix（设置符号存储路径）** 命令自动加载符号，前提是在调试器运行时可以访问 Internet。 然后使用 **.reload（重新加载模块）** 命令加载符号。

如果要执行用户模式调试，则需要目标应用程序的符号。 如果要执行内核模式调试，则需要要调试的驱动程序的符号以及 Windows 公共符号。

这些主题介绍如何在调试会话期间访问符号以及如何控制调试程序的符号选项和符号匹配。

Microsoft 公共符号服务器

Windows 符号程序包

Windows 调试器的符号路径

这些主题介绍符号的含义以及对可移植 PDB 符号的 WinDbg 支持。

符号和符号文件

公共和专用符号

可移植 PDB 符号

有关使用符号的其他详细信息，请参阅以下页面。

使用符号服务器

SymSrv 的高级用法

防火墙和代理服务器

符号语法和符号匹配

自定义符号存储和符号服务器

调试时的符号问题

如果你只是想要配置调试器，使其能够为你自己的程序和 Windows 访问符号，你可能会发现，阅读简要介绍的主题符号路径和 Microsoft 公共符号服务器会更快地实现这一目的。 使用 Use !sym noisy 命令，在加载符号时显示更多详细信息，以解决符号问题。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Microsoft 公共符号服务器

项目 • 2024/05/31

服务器状态：无已知问题✅
Microsoft 公共符号服务器已全面运行。

向 windbgfb@microsoft.com 报告任何已知问题。

Microsoft 符号服务器使 Windows 调试程序符号公开可用。

如何访问

可以通过多种不同方式在符号路径中直接引用公共符号服务器，如 Windows 调试器的符号路径中所述。 例如，要设置 _NT_SYMBOL_PATH 环境值，请使用此命令。

控制台

```
set
_NT_SYMBOL_PATH=srv*DownstreamStore*https://msdl.microsoft.com/download/symbols
```

DownstreamStore 必须在本地计算机或网络上指定一个将要用于缓存符号的目录。 此下游存储可存放调试器访问过的符号。 大多数从未被访问过的符号仍保留在 Microsoft 的符号库中。 此存储过程可让下游存储空间保持相对较小，并让符号服务器能够快速工作，每个文件只需下载一次。

要避免键入如此长的符号路径，请使用 .symfix（设置符号存储路径）命令。 以下命令将公共符号存储追加到现有的符号路径：

dbgcmd

```
.symfix+ C:\MySymbols
```

如果省略了本地符号缓存位置，则使用调试器安装目录下的 sym 子目录。

使用 .sympath（设置符号存储路径）命令显示完整的符号路径。 以下示例展示了如何使用 symfix 创建本地符号缓存并使用 Microsoft http 符号服务器。

dbgcmd

```
0: kd> .symfix c:\MyCache
0: kd> .sympath
Symbol search path is: srv*
```

```
Expanded Symbol search path is:
cache*c:\MyCache;SRV*https://msdl.microsoft.com/download/symbols
```

有关使用符号的详细信息，请参阅使用符号服务器和 Windows 调试器的符号路径。

公共符号服务器仅支持 TLS 1.2+ 的 https 连接。

Microsoft 许可条款 - Microsoft 符号服务器

Microsoft 通过 Microsoft 符号服务器来提供某些符号、二进制代码和其他可执行文件。 这些资源用于使用 Microsoft 软件调试和测试用户的软件。 未经授权不得使用它们。 请参阅 Microsoft 许可条款 - Microsoft 符号服务器。

另请参阅

Windows 调试器的符号路径

符号和符号文件

.symfix（设置符号存储路径）

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

用于调试的 Windows 符号包

项目 • 2023/12/29

符号文件使调试代码更加容易。 获取 Windows 符号的最简单方法是使用 Microsoft 公共符号服务器。 符号服务器根据需要向调试工具提供符号。 从符号服务器下载符号文件后，会将其缓存在本地计算机上以便快速访问。

符号包弃用

> ⓘ **重要**
>
> 我们不再发布适用于 Windows 的脱机符号包。
>
> 随着我们发布 Windows 更新的节奏，我们通过此页面上的包发布的 Windows 调试符号很快就会过时。 通过将联机 ***Microsoft 公共符号服务器*** 转换为基于 Azure 的符号存储，我们对它进行了重大改进。 所有 Windows 版本和更新的符号都位于该处。 有关详细信息，请参阅此 ***博客文章***。
>
> 有关如何检索未连接到 Internet 的计算机的符号的信息，请参阅 ***将清单文件与 SymChk 配合使用***。

符号资源和反馈

若要了解有关使用符号和调试的详细信息，请参阅 符号和符号文件。

有关调试问题的帮助，请参阅 调试资源。

我们有兴趣了解你对符号的反馈。 邮件建议或 bug 报告给 windbgfb@microsoft.com。 此地址不提供技术支持，但你的反馈将帮助我们规划符号的未来更改，并使它们将来对你更有用。

另请参阅

- 下载 Windows 调试工具
- 下载 Windows 驱动程序工具包 (WDK)
- 下载 Windows 评估和部署工具包 (Windows ADK)
- 下载 Windows HLK、HCK 或徽标工具包

- 下载Windows Insider Preview版本

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

Windows driver documentation 反馈

Windows driver documentation 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

Windows 调试器的符号路径

项目 • 2024/07/08

符号路径指定 Windows 调试器（如 WinDbg、KD、CDB 和 NTST）查找符号文件的位置。 有关符号和符号文件的详细信息，请参阅符号。

某些编译器（包括 Microsoft Visual Studio）将符号文件放在与二进制文件相同的目录中。 符号文件和选中的二进制文件包含路径和文件名信息，使调试器能够自动查找符号文件。 如果在生成可执行文件的计算机上调试用户模式进程，并且符号文件位于其原始位置，则调试器可以在不设置符号路径的情况下找到符号文件。

在大多数其他情况下，需要将符号路径设置为指向符号文件位置。

> **提示**
>
> 使用 ***.symfix*** 设置公共Microsoft公共符号服务器的默认路径，这些服务器在很多情况下效果良好。

符号路径语法

调试器的符号路径是一个字符串，其中包含由分号分隔的多个目录路径。 例如 `C:\Dir1;C:\Dir2\DirA;C:\Dir2\DirB`。

支持相对路径。 但是，应在每个路径之前添加驱动器号或网络共享，除非始终从同一目录中启动调试器。 还支持网络共享。

对于符号路径中的每个目录，调试器将在三个目录中查找。 例如，如果符号路径包含 `C:\Dir1` 并且调试器正在查找 DLL 的符号信息，则调试器将按顺序查找以下目录中的符号信息：

- `C:\Dir1\symbols\dll`
- `C:\Dir1\dll`
- `C:\Dir1`

然后，调试器对符号路径中的每个目录重复此过程。 最后，调试器在当前目录中查找，然后在当前目录中附加 `..\dll`。 调试器根据调试的二进制文件追加`..\dll` 或`..\exe` `..\sys`追加。

符号文件具有日期和时间戳。 调试器始终查找与它正在调试的二进制文件上的时间戳匹配的符号。 无需担心调试器使用它在此序列中首先找到的错误符号。 有关符号文件不可用时响应的详细信息，请参阅 匹配符号名称。

设置符号路径的一种方法是输入 .sympath 命令。 有关设置符号路径的其他方法，请参阅本主题后面的“控制符号路径 ”。

在本地缓存符号

应在本地缓存符号。 在本地缓存符号的一种方法是在符号路径中包含 `cache*;` 或 `cache*localsymbolcache;*`。

如果在符号路径中包含字符串 `cache*;`，则从该字符串右侧的任何元素加载的符号都存储在本地计算机的默认符号缓存目录中。 例如，以下命令告知调试器从命名 `\\someshare` 的网络共享获取符号，并在本地计算机上的默认位置缓存符号。

```
dbgcmd
```

```
.sympath cache*;\\someshare
```

如果在符号路径中包含字符串 `cache*localsymbolcache;`，则从该字符串右侧出现的任何元素加载的符号都存储在 localsymbolcache 目录中。

例如，以下命令通知调试器从网络共享 `\\someshare` 中获取符号，并将这些符号缓存在 `c:\MySymbols` 目录中。

```
dbgcmd
```

```
.sympath cache*C:\MySymbols;\\someshare
```

使用符号服务器：srv*

如果连接到 Internet 或公司网络，访问符号的最有效方法是使用符号服务器（如公共 Microsoft公共符号服务器 ）。 可以使用符号路径中的以下字符串之一来使用符号服务器。

- 字符串 `srv*`

  如果在符号路径中包含字符串 `srv*`，调试器将使用符号服务器从默认符号存储中获取符号。 例如，以下命令告知调试器从默认符号存储中获取符号。 这些符号不会在本地计算机上缓存。

  ```
  dbgcmd
  ```

  ```
  .sympath srv*
  ```

- 字符串`srv*symbolstore`

  如果在符号路径中包含字符串 `srv*symbolstore` ，调试器将使用符号服务器从 *符号存储*中获取符号。 例如，以下命令告知调试器从 Microsoft符号服务器 存储中获取符号。 这些符号不会在本地计算机上缓存。

  dbgcmd

  ```
  .sympath srv*https://msdl.microsoft.com/download/symbols
  ```

- 字符串`srv*localsymbolcache*symbolstore`

  如果将字符串 `srv*localcache*symbolstore` 包含在符号路径中，调试器将使用符号服务器从 *符号存储* 获取符号，并将其缓存在 *localcache* 目录中。 例如，以下命令告知调试器从Microsoft符号服务器获取符号，`https://msdl.microsoft.com/download/symbols` 并在中 `c:\MyServerSymbols`缓存符号。

  dbgcmd

  ```
  .sympath
  srv*C:\MyServerSymbols*https://msdl.microsoft.com/download/symbols
  ```

如果计算机上具有手动放置符号的目录，请不要使用该目录作为从符号服务器获取的符号的缓存。 相反，请使用两个不同的目录。 例如，可以手动将符号放在 `c:\MyRegularSymbols` 中，然后指定用 `c:\MyServerSymbols` 缓存从服务器获取的符号。 下面的示例演示如何在符号路径中指定这两个目录。

dbgcmd

```
.sympath
C:\MyRegularSymbols;srv*C:\MyServerSymbols*https://msdl.microsoft.com/downlo
ad/symbols
```

有关符号服务器和符号存储的详细信息，请参阅 自定义符号存储和符号服务器。

合并缓存* 和 srv*

如果在符号路径中包含字符串 `cache*;`，则从该字符串右侧的任何元素加载的符号都存储在本地计算机的默认符号缓存目录中。 例如，以下命令告知调试器从 Microsoft符号服务器 存储中获取符号，并将其缓存在默认符号缓存目录中。

dbgcmd

```
.sympath cache*;srv*https://msdl.microsoft.com/download/symbols
```

如果在符号路径中包含字符串 `cache*localsymbolcache;`，则从该字符串右侧出现的任何元素加载的符号都存储在 localsymbolcache 目录中。

例如，以下命令告知调试器从 Microsoft符号服务器 存储和缓存目录中的 `c:\MySymbols` 符号。

dbgcmd

```
.sympath cache*C:\MySymbols;srv*https://msdl.microsoft.com/download/symbols
```

使用 AgeStore 减小缓存大小

可以使用 AgeStore 工具删除早于指定日期的缓存文件，或删除足够的旧文件，以便生成的缓存大小小于指定数量。 如果下游存储太大，则缓存文件的清理非常有用。

延迟符号加载

调试器的默认行为是使用*延迟符号加载*，也称为*延迟符号加载*。 这种加载意味着在需要符号之前不会加载符号。

更改符号路径（例如，通过使用 .sympath 命令）时，所有带有导出符号的已加载模块都会延迟重新加载。

如果新路径不再包含用于加载 PDB 符号的原始路径，则具有完整 PDB 符号 的模块符号将延迟重新加载。 如果新路径仍包含 PDB 符号文件的原始路径，则不会延迟重新加载这些符号。

可以使用 -s 命令行选项关闭 CDB 和 KD 中的延迟符号加载。 还可以通过使用 ld load symbols 命令或使用 .reload 模块命令 和选项来 `/f` 强制加载符号。

Azure DevOps Services Artifacts

符号服务器可用于 Azure DevOps Services 中的 Azure Artifacts。 若要了解如何在 WinDbg 中使用 Azure Artifacts，请参阅 使用 WinDbg 中的符号进行调试。 有关 Azure 生成的符号的一般信息，请参阅 “符号”概述。

控制符号路径

若要控制符号路径，可以选择以下方法之一：

- 使用 .symfix 设置符号存储路径命令设置公共Microsoft符号服务器的默认路径，这些服务器在很多情况下效果良好。 若要设置本地缓存，只需键入 `.symfix C:\MyCache`。
- 使用 .sympath 命令显示、设置、更改或追加到路径。
- 在启动调试器之前，请使用 `_NT_SYMBOL_PATH` 和 `_NT_ALT_SYMBOL_PATH` 环境变量设置路径。 符号路径是通过在 `_NT_ALT_SYMBOL_PATH` 后附加 `_NT_SYMBOL_PATH` 创建的。 通常，路径通过 `_NT_SYMBOL_PATH`. 但是，你可能想要在特殊情况下使用这些 `_NT_ALT_SYMBOL_PATH` 设置，例如，如果你有共享符号文件的专用版本。 如果尝试通过这些环境变量添加无效目录，调试器将忽略此目录。
- 启动调试器时，请使用 -y 命令行选项 设置路径。
- 仅在 WinDbg 中，可以使用文件 |符号文件路径命令 或按 `CTRL+S` 显示、设置、更改或追加到路径。

如果使用 -sins 命令行选项，调试器将忽略符号路径环境变量。

故障排除

使用 **！ sym 干扰**或 *-n* **WinDbg 命令行选项**在加载符号时显示其他详细信息。 有关其他故障排除策略，请参阅 “验证符号”。

另请参阅

符号

高级 SymSrv 使用

.sympath 命令

.symfix

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用符号服务器

项目 • 2024/11/14

符号服务器使调试器能够自动从符号存储（符号文件的索引集合）检索正确的符号文件，而无需用户知道产品名称、版本或内部版本号。 Windows 调试工具包括所需的 dll 符号服务器 *SymSrv*。 有关使用符号服务器的信息，请参阅 Microsoft公共符号。

将 SymSrv 与调试器配合使用

SymSrv 可与 WinDbg、KD、NTSD 或 CDB 配合使用。

若要将此符号服务器用于调试器，只需在符号路径中包含文本 **srv*** 即可。 例如：

控制台

```
set _NT_SYMBOL_PATH = srv*DownstreamStore*SymbolStoreLocation
```

其中，*DownstreamStore* 指定将用于缓存单个符号文件的本地目录或网络共享，SymbolStoreLocation 是符号存储的位置，格式为 *\\server\share* 或 Internet 地址。 有关更多语法选项，请参阅 高级 SymSrv 使用。

Microsoft有一个网站，使 Windows 符号公开可用。 可以通过以下方式直接在符号路径中引用此站点：

控制台

```
set
_NT_SYMBOL_PATH=srv*DownstreamStore*https://msdl.microsoft.com/download/symb
ols
```

其中， *下游存储* 再次指定将用于缓存单个符号文件的本地目录或网络共享。 有关详细信息，请参阅 Microsoft公共符号。

如果计划创建符号存储、配置用于 Web（HTTP）访问的符号存储，或编写自己的符号服务器或符号存储，请参阅 自定义符号存储和符号服务器。

使用 AgeStore 减少缓存大小

调试会话结束后，SymSrv 下载的任何符号文件都将保留在硬盘驱动器上。 若要控制符号缓存的大小，AgeStore 工具可用于删除早于指定日期的缓存文件，或减少缓存小于指定大小的内容。 有关详细信息，请参阅 AgeStore。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

SymSrv 的高级用法

项目 • 2024/03/01

SymSrv 可以从中央符号存储中传送符号文件。 此存储可以包含任意数量的符号文件，对应任意数量的程序或操作系统。 存储还可以包含二进制文件（在调试微型转储时非常有用）。

存储可以包含实际符号和二进制文件，也可以只包含指向符号文件的指针。 如果存储包含指针，SymSrv 将直接从文件源检索实际文件。

SymSrv 还可用于将大型符号存储区分隔为适合专用调试任务的较小子集。

最后，SymSrv 可以使用操作系统提供的登录信息从 HTTP 或 HTTPS 源获取符号文件。 SymSrv 支持受智能卡、证书以及常规登录名和密码保护的 HTTPS 站点。 有关详细信息，请参阅 HTTP 符号存储。

设置符号路径

若要使用此符号服务器，symsrv.dll 必须与调试器安装在同一目录下。 可以按以下代码所示设置符号路径：

控制台

```
set _NT_SYMBOL_PATH = symsrv*ServerDLL*DownstreamStore*\\Server\Share

set _NT_SYMBOL_PATH = symsrv*ServerDLL*\\Server\Share

set _NT_SYMBOL_PATH = srv*DownstreamStore*\\Server\Share

set _NT_SYMBOL_PATH = srv*\\Server\Share
```

此语法的各个部分说明如下：

symsrv
此关键字必须始终首先显示。 它向调试器表明，此项是符号服务器，而不仅仅是普通的符号目录。

ServerDLL
指定符号服务器 DLL 的名称。 如果使用 SymSrv 符号服务器，则始终为 symsrv.dll。

srv
这是 **symsrv*symsrv.dll** 的简写。

DownstreamStore
指定下游存储。 这是用于缓存单个符号文件的本地目录或网络共享。

可以指定多个下游存储，用星号分隔。 此页下方的**级联下游存储**中解释了多个下游存储。

如果在通常指定下游存储的行中包含两个星号，则使用默认的下游存储。 此存储将位于主目录的符号子目录中。 主目录默认为调试器安装目录；这可以通过使用 !homedir 扩展或设置DBGHELP_HOMEDIR 环境变量来更改。

如果 *DownstreamStore* 指定不存在的目录，SymStore 将尝试创建它。

如果省略了 *DownstreamStore* 参数并且不包含额外的星号，换句话说，如果使用仅带一个星号的 **srv** 或仅带有两个星号的 **symsrv**，则不会创建任何下游存储。 调试器将直接从服务器加载所有符号文件，而不在本地缓存它们。

注意： 如果要从 HTTP 或 HTTPS 站点访问符号，或者符号存储使用压缩文件，则始终使用下游存储。 如果未指定下游存储，则会在主目录的符号子目录中创建一个。

\\Server\Share
指定远程符号存储的服务器和共享。

如果使用下游存储，调试器将首先在此存储中查找符号文件。 如果未找到符号文件，调试器将从指定的 *服务器*和*共享*中找到符号文件，然后在下游存储中缓存此文件的副本。 该文件将复制到 *DownstreamStore* 下树中的子目录，该子目录对应于其在 *\\Server\Share* 下树中的位置。

符号服务器不必是符号路径中唯一的条目。 如果符号路径由多个条目组成，则无论命名的是符号服务器还是实际目录，调试器都会按顺序（从左到右）检查所需符号文件的每个条目。

下面是一些示例。 若要将 SymSrv 用作符号服务器并在 \\mybuilds\mysymbols 上存储符号，请设置以下符号路径：

控制台

```
set _NT_SYMBOL_PATH= symsrv*symsrv.dll*\\mybuilds\mysymbols
```

若要设置符号路径以便调试器将符号文件从 \\mybuilds\mysymbols 上的符号存储复制到本地目录 c:\localsymbols，请使用：

控制台

```
set _NT_SYMBOL_PATH=symsrv*symsrv.dll*c:\localsymbols*\\mybuilds\mysymbols
```

若要设置符号路径以便调试器将符号文件从 HTTPS 站点复制到 `https://www.company.com/manysymbols` 本地网络目录 \\localserver\mycache，请使用：

控制台

```
set
_NT_SYMBOL_PATH=symsrv*symsrv.dll*\\localserver\myshare\mycache*https://www.
company.com/manysymbols
```

最后一个示例也可以缩短：

控制台

```
set
_NT_SYMBOL_PATH=srv*\\localserver\myshare\mycache*https://www.company.com/ma
nysymbols
```

此外，符号路径可以包含多个目录或符号服务器，用分号分隔。 这样就可以从多个位置（甚至多个符号服务器）查找符号。 如果二进制文件的符号文件不匹配，则调试器无法使用符号服务器找到它，因为它仅检查确切参数。 但是，调试器可能会使用传统符号路径找到名称正确的不匹配符号文件，并成功加载它。 尽管文件在技术上不是正确的符号文件，但它可能提供有用的信息。

删除缓存

如果使用 *DownstreamStore* 作为缓存，可以随时删除此目录以节省磁盘空间。

可以拥有一个庞大的符号存储区，其中包含许多不同程序或 Windows 版本的符号文件。 如果升级目标计算机上使用的 Windows 版本，则缓存的符号文件将全部与早期版本匹配。 这些缓存的文件不会再使用，因此这可能是删除缓存的好时机。

级联下游存储

可以指定任意数量的下游存储，用星号分隔。 这些存储称为*级联符号存储*。

初始 `srv*` 或 `symsrv*ServerDLL*` 之后，每个后续标记表示符号位置。 首先检查最左边的标记。 空标记（由一行中的两个星号或字符串末尾的星号表示）表示默认的下游存储。

下面是一个符号路径示例，该路径使用两个下游存储来保存正在访问的主符号存储中的信息。 这些可以称为主存储、中级存储和本地缓存：

控制台

```
srv*c:\localcache*\\interim\store*https://msdl.microsoft.com/download/symbol
s
```

在此方案中，SymSrv 将首先在 c:\localcache 中查找符号文件。 如果在那里找到它，它将返回一个路径。 如果未在那里找到，它将在 \\interim\store 中查找。 如果在那里找到符号文件，SymSrv 会将其复制到 c:\localcache 并返回路径。 如果未在那里找到，SymSrv 将在 Microsoft 公共符号存储中https://msdl.microsoft.com/download/symbols 查找；如果找到该文件，SymSrv 会将该文件复制到 \\interim\store 和 c:\localcache。

使用以下路径可以获取类似的行为：

控制台

```
srv**\\interim\store*https://internetsite
```

在这种情况下，本地缓存是默认的下游存储，主存储是 Internet 站点。 指定了 \\interim\store 的中级存储区在其他两个存储之间使用。

当 SymSrv 处理包含级联存储的路径时，它将跳过无法读取或写入的任何存储。 因此，如果共享出现故障，它会将文件从缺失存储区复制到下游存储区，而不会出现任何错误。 此错误的一个很好的副作用是，只要主存储不可写，用户就可以指定多个主存储来提供单个下游存储流。

从主存储中检索压缩符号文件时，它将以压缩形式存储在任何中级存储中。 该文件将在路径的最底部存储区中解压缩。

使用 HTTP 和 SMB 符号服务器路径

如前所述，链接（或级联）是指在符号路径中每个“*”分隔符之间进行的复制。 符号按从左到右的顺序搜索。 每次未命中时，将查询下一个（上游）符号服务器，直到找到该文件。

如果找到该文件，则会将文件从（上游）符号服务器复制到上一个（下游）符号服务器。 对于每个（下游）符号服务器，都会重复此操作。 通过这种方式，（共享）下游符号服务器将由使用符号服务器的所有客户端的集体努力进行填充。

即使可以在没有 SRV* 前缀的情况下使用链接的 UNC 路径，但我们建议指定 SRV* 以便使用 symsrv.dll 的高级错误处理。

在路径中包含 HTTP 符号服务器时，只能指定一个（每个链），并且它必须位于路径的末尾（因为它无法写入以用作缓存）。 如果基于 HTTP 的符号存储位于存储列表的中间或左侧，则无法将任何找到的文件复制到其中，并且链将断开。 此外，由于符号处理程序无法从网站打开文件，因此基于 HTTP 的存储不应是列表最左侧或唯一的存储。 如果

SymSrv 曾出现过此符号路径，它将尝试通过将文件复制到默认的下游存储并将其从该位置打开来恢复，无论符号路径中是否指示默认下游存储。

仅当使用 SRV* 前缀（由 symsrv.dll 符号处理程序实现）时，才支持 HTTP。

HTTP 和 SMB 共享符号服务器方案示例

常见的仅 UNC 部署涉及托管所有文件的中央办公室 (\\MainOffice\Symbols)、缓存子集的分支机构 (\\BranchOfficeA\Symbols) 和缓存其引用的文件桌面 (C:\Symbols)。

控制台

```
srv*C:\Symbols*\\BranchOfficeA\Symbols*\\MainOffice\Symbols
```

当 SMB 共享是主要（上游）符号存储时，需要读取。

控制台

```
srv*C:\Symbols*\\MachineName\Symbols
```

当 SMB 共享是中间（下游）符号存储时，需要读取/更改。 客户端会将文件从主符号存储复制到 SMB 共享，然后从 SMB 共享复制到本地文件夹。

控制台

```
srv*C:\Symbols*\\MachineName\Symbols*https://msdl.microsoft.com/download/symbols
srv*C:\Symbols*\\MachineName\Symbols*\\MainOffice\Symbols
```

当 SMB 共享是 SymProxy 部署中的中间（下游）符号存储时，只需要读取。 SymProxy ISAPI 筛选器将执行写入操作，而不是客户端。

控制台

```
srv*C:\Symbols*\\MachineName\Symbols*https://SymProxyName/Symbols
```

多个 HTTP 和 SMB 共享符号服务器缓存方案

可以指定多个符号服务器链和缓存位置，用分号“;”分隔。 如果符号位于第一个链中，则不会遍历第二个链。 如果符号不在第一个链中，将遍历第二个链；如果符号位于第二个链中，则会将其缓存在指定位置。 如果符号在第一个链中指定的主符号服务器上不可用，此方法将允许正常使用主符号服务器，并且仅使用辅助服务器。

控制台

```
srv*C:\Symbols*\\Machine1\Symbols*https://SymProxyName/Symbols;srv*C:\WebSym
bols*https://msdl.microsoft.com/download/symbols
```

cache**localsymbolcache*

创建符号本地缓存的另一种方法是使用符号路径中的 `cache*localsymbolcache` 字符串。 这不是符号服务器元素的一部分，而是符号路径中的单独元素。 调试器将使用指定的目录 *localsymbolcache* 来存储从此字符串右侧的符号路径中出现的任何元素加载的任何符号。 这允许您对从任何位置下载的符号使用本地缓存，而不仅仅是符号服务器下载的符号。

例如，以下符号路径不会缓存从 *someshare* 获取的符号。 它将使用 c:\mysymbols 缓存从 *anothershare* 获取的符号，因为以 *anothershare* 开头的元素显示在 **cache*c:\mysymbols** 元素右侧。 它还将使用 c:\mysymbols 来缓存从 Microsoft 公共符号存储中获取的符号，因为符号服务器使用的是通常语法（**srv** 包含两个或多个星号）。 此外，如果随后使用 **.sympath+** 命令将其他位置添加到此路径，这些新元素也将被缓存，因为它们将会追加到路径的右侧。

控制台

```
_NT_SYMBOL_PATH=\\someshare\that\cachestar\ignores;srv*c:\mysymbols*https://
msdl.microsoft.com/download/symbols;cache*c:\mysymbols;\\anothershare\that\g
ets\cached
```

How SymSrv Locates Files

SymSrv 创建所需符号文件的完全限定 UNC 路径。 此路径以 _NT_SYMBOL_PATH 环境变量中记录的符号存储路径开始。 **SymbolServer** 例程用于标识所需文件的名称；此名称作为目录名称追加到路径中。 然后追加另一个目录名称，由传递给 *SymbolServer* 的 *ID*、*2* 和 **3** 个参数的串联组成。 如果其中任何值为零，则省略这些值。

在结果目录中搜索符号文件或符号存储指针文件。

如果此搜索成功，**SymbolServer** 会将路径传递给调用方并返回 **TRUE**。 如果未找到该文件，则 **SymbolServer** 返回 **FALSE**。

使用 AgeStore 减少缓存大小

AgeStore 工具可用于删除早于指定日期的缓存文件，或将缓存内容减少到指定大小以下。 如果下游存储太大，这可能很有用。 有关详细信息，请参阅 AgeStore。

防火墙和代理服务器

项目 • 2023/10/26

如果使用 SymSrv 访问符号，并且计算机位于使用代理服务器的网络上，或者符号存储位于防火墙之外，则可能需要进行身份验证才能进行数据传输。

SymSrv 收到身份验证请求时，调试器可以显示身份验证请求或自动拒绝请求，具体取决于其配置方式。

SymSrv 具有对代理服务器的集成支持。 它可以使用默认代理服务器 SymProxy，也可以使用你选择的另一个代理服务器。

身份验证请求

可以将调试器配置为允许身份验证请求。 当防火墙或代理服务器请求授权时，将显示一个对话框。 必须输入某种信息 (通常为用户名和密码)，调试器才能下载符号。 如果输入的信息不正确，对话框会重新显示。 如果选择“ **取消** ”按钮，对话框将消失，并且不会传输符号信息。

如果将调试器配置为拒绝所有身份验证请求，则不会显示任何对话框，并且如果需要身份验证，则不会传输符号。

如果拒绝身份验证请求，或者调试器自动拒绝身份验证请求，SymSrv 将不会进一步尝试联系符号存储。 如果要续订联系人，则必须重新启动调试会话或使用 **！ symsrv close**。

注意 如果使用 KD 或 CDB，则身份验证对话框可能会出现在打开的窗口后面。 如果发生这种情况，可能需要移动或最小化某些窗口才能找到此对话框。

在 WinDbg 中，默认情况下允许身份验证请求。 在 KD 和 CDB 中，默认情况下会自动拒绝身份验证请求。

若要允许身份验证请求，请使用 **！ sym 提示** 符或 **.symopt-0x80000**。 若要拒绝所有请求，请使用 **！ sym 提示关闭** 或 **.symopt+0x80000**。 若要显示当前设置，请使用 **！ sym**。

对身份验证权限状态进行任何更改后，必须使用 **.reload (重新加载模块)** 。

选择代理服务器

若要为 Windows 选择默认代理服务器，请在 控制面板 中打开“**Internet 选项**”，选择“**连接**”选项卡，然后选择“**LAN 设置**”按钮。 然后，可以输入代理服务器名称和端口号，或选

择“ **高级** ”来配置多个代理服务器。 有关更多详细信息，请参阅 Internet Explorer 的帮助文件。

若要选择要使用的特定代理服务器，请将_NT_SYMBOL_PROXY环境变量设置为等于代理服务器的名称或 IP，后跟冒号和端口号。 例如：

控制台

```
set _NT_SYMBOL_PROXY=myproxyserver:80
```

以这种方式选择代理服务器时，任何使用 SymSrv 的 Windows 调试器都将使用该代理服务器来访问符号服务器。 它还将由使用 DbgHelp 作为符号处理程序的任何其他调试工具使用。 此设置不会影响其他任何程序。

符号和符号文件

项目 • 2023/06/14

链接应用程序、库、驱动程序或操作系统时，创建.exe和.dll文件的链接器也会创建一些称为 *符号文件的其他文件*。

符号文件包含大量的数据，这些数据在运行二进制文件时实际上并不需要，但在调试过程中很有用。

通常，符号文件可能包含：

- 全局变量
- 局部变量
- 函数名称及其入口点的地址
- FPO) 记录 (框架指针省略
- 源行号

其中每个项单独称为 *一个符号*。 例如，单个符号文件 Myprogram.pdb 可能包含数百个符号，包括全局变量和函数名称和数百个局部变量。 通常，软件公司会发布每个符号文件的两个版本：一个包含 *公共符号* 和 *私有符号的完整符号*文件，以及一个仅包含公共符号的简化 (剥离) 文件。 有关详细信息，请参阅 公共符号和专用符号。

调试时，必须确保调试器可以访问与要调试的目标关联的符号文件。 实时调试和调试故障转储文件都需要符号。 必须为要调试的代码获取正确的符号，并将这些符号加载到调试器中。

Windows 符号

Windows 将其符号保存在扩展名为 .pdb 的文件中。

编译器和链接器控制符号格式。 Visual C++ 链接器将所有符号放入 .pdb 文件中。

Windows 操作系统内置有两个版本。 *免费生成* (或*零售版本*) 具有相对较小的二进制文件，且*检查的生成* (或*调试版本*) 具有较大的二进制文件，代码本身具有更多的调试符号。 已检查的版本在 Windows 10 版本 1803 之前的较旧版本的 Windows 上可用。 每个生成都有自己的符号文件。 在 Windows 上调试目标时，必须使用与目标上的 Windows 内部版本匹配的符号文件。

下表列出了标准 Windows 符号树中存在的几个目录：

目录	包含 的符号文件
Acm	Microsoft Audio Compression Manager 文件
COM	可执行文件 (.com)
CPL	控制面板程序
DLL	动态链接库文件 (.dll)
DRV	驱动程序文件 (.drv)
EXE	可执行文件 (.exe)
Scr	屏幕保护程序文件
SYS	驱动程序文件 (.sys)

公共和专用符号

项目 • 2023/10/26

当链接器生成全尺寸的 .pdb 或 .dbg 符号文件时，它包含两个不同的信息集合：*私有符号数据*和*公共符号表*。 这些集合在包含的项列表和存储的关于每个项的信息方面有所不同。

私有符号数据包括以下项：

- 函数
- 全局变量
- 局部变量
- 有关用户定义的结构、类和数据类型的信息
- 源文件的名称和该文件中对应于每个二进制指令的行号

公共符号表包含的项较少：

- 函数 (，声明为 **静态**) 的函数除外
- 指定为 **外部** (的全局变量，以及跨多个对象文件可见的任何其他全局变量)

作为一般规则，公共符号表只包含可从一个源文件到另一个源文件访问的项。 仅在一个对象文件中可见的项（例如 **静态** 函数、仅在单个源文件中的全局变量和局部变量）不包含在公共符号表中。

这两个数据集合在针对每个项包含的信息方面也有所不同。 对于 *私有* 符号数据中包含的每个项，通常包含以下信息：

- 项的名称
- 虚拟内存中项的地址
- 每个变量、结构和函数的数据类型
- 每个函数的参数类型和名称
- 每个局部变量的范围
- 与每个源文件中的每一行关联的符号
- 帧指针省略 (FPO) 用于访问堆栈的每个函数的记录

另一方面， *公共* 符号表仅存储有关其中包含的每个项的以下信息：

- 项的名称。
- 项在其模块的虚拟内存空间中的地址。 对于函数，这是其入口点的地址。
- 框架指针省略 (FPO) 每个函数的记录。
- 可能包含称为修饰的符号前缀/后缀。

可以通过两种方式将公共符号数据视为私有符号数据的子集：它包含较短的项列表，并且还包含有关每个项的较少信息。 例如，公共符号数据根本不包括局部变量。

每个局部变量仅包含在专用符号数据中，其地址、数据类型和范围。 另一方面，函数同时包含在私有符号数据和公共符号表中，但尽管私有符号数据包括函数名称、地址、FPO 记录、输入参数名称和类型和输出类型，但公共符号表仅包括函数名称、地址和 FPO 记录。

私有符号数据和公共符号表之间还有一个区别。 公共符号表中的许多项的名称都 *带有* 前缀和/或后缀。 这些修饰由 C 编译器、C++ 编译器和 MASM 汇编程序添加。 典型的前缀包括一系列下划线或字符串 **__imp_** (指定导入的函数) 。 典型的后缀包括一个或多个 at 符号 (@) 后跟地址或其他标识字符串。 链接器使用这些修饰来消除符号的歧义，因为函数名称或全局变量名称可能在不同的模块之间重复。 这些修饰是公共符号表是私有符号数据的子集的一般规则的例外。

完整符号文件和去除符号文件

*完整的符号文件*同时包含私有符号数据和公共符号表。 此类文件有时称为 *专用符号文件*，但此名称具有误导性，因为此类文件同时包含私有符号和公共符号。

*去除符号文件*是仅包含公共符号表的较小文件，或者在某些情况下，仅包含公共符号表的子集。 此文件有时称为 *公共符号文件*。

创建完整和去除符号文件

如果使用 Visual Studio 生成二进制文件，则可以创建完整符号文件或带去符号文件。 有关构建剥离符号的信息，请参阅 /PDBSTRIPPED (Strip Private Symbols) 。

使用 BinPlace 工具，可以从完整的符号文件创建去除的符号文件。 当使用最常见的 BinPlace 选项 (**-a -x -s -n**) 时，去除的符号文件将放在 **-s** 开关后列出的目录中，而完整的符号文件将放在 **-n** 开关后列出的目录中。 当 BinPlace 去除符号文件时，会为该文件的去除版本和完整版本提供相同的签名和其他标识信息。 这允许使用任一版本进行调试。 有关 BinPlace 的详细信息，请参阅 BinPlace。

使用 PDBCopy 工具，可以通过删除私有符号数据，从完整的符号文件创建去除的符号文件。 PDBCopy 还可以删除公共符号表的指定子集。 有关详细信息，请参阅 PDBCopy。

使用 SymChk 工具，可以确定符号文件是否包含专用符号。 有关详细信息，请参阅 SymChk。

在调试器中查看公共符号和专用符号

可以使用 WinDbg、KD 或 CDB 查看符号。 当其中一个调试器有权访问完整的符号文件时，它同时具有私有符号数据中列出的信息和公共符号表中列出的信息。 公共符号数据包含符号修饰。

访问专用符号时，始终使用私有符号数据，因为这些符号不包含在公共符号表中。 这些符号永远不会修饰。

.symopt (Set Symbol Options) 命令可用于控制确定调试器如何使用公共符号和专用符号的符号选项。 例如，此命令打开符号调试信息。

```
dbgcmd

.symopt+ 0x80000000
```

以下选项更改了在调试器中使用公共符号和专用符号的方式。

- 打开 “SYMOPT_UNDNAME ”选项时，显示公共符号的名称时不包括修饰。 此外，搜索符号时，会忽略修饰。 当此选项处于关闭状态时，显示公共符号时会显示修饰，并在搜索中使用修饰。 任何情况下都不会修饰私有符号。 默认情况下，此选项在所有调试器中都处于打开状态。

- 打开 “SYMOPT_PUBLICS_ONLY ”选项时，将忽略私有符号数据，并且仅使用公共符号表。 默认情况下，此选项在所有调试器中处于关闭状态。

- 打开 “SYMOPT_NO_PUBLICS ”选项时，将忽略公共符号表，搜索和符号信息仅使用专用符号数据。 默认情况下，此选项在所有调试器中处于关闭状态。

- 当 SYMOPT_AUTO_PUBLICS 选项处于 (并且SYMOPT_PUBLICS_ONLY和SYMOPT_NO_PUBLICS都关闭) 时，将在私有符号数据中执行第一个符号搜索。 如果找到所需的符号，则搜索将终止。 如果没有，则搜索公共符号表。 由于公共符号表在私有数据中包含符号的子集，因此这通常会导致忽略公共符号表。

- 当SYMOPT_PUBLICS_ONLY、SYMOPT_NO_PUBLICS和SYMOPT_AUTO_PUBLICS选项全部关闭时，每次需要符号时，都会搜索私有符号数据和公共符号表。 但是，当在这两个位置找到匹配项时，将使用私有符号数据中的匹配项。 因此，此实例中的行为与打开 SYMOPT_AUTO_PUBLICS 时的行为相同，只是使用 SYMOPT_AUTO_PUBLICS 可能会导致符号搜索的发生速度略快一些。

以下示例使用命令 **x (检查符号)** 三次。 第一次使用默认符号选项，因此信息取自私有符号数据。 请注意，这包括有关数组 **typeString** 的地址、大小和数据类型的信息。 接下来，使用命令 .symopt+ 4000，导致调试器忽略专用符号数据。 当 **x** 命令再次运行时，将使用公共符号表;这次没有 **用于 typeString** 的大小和数据类型信息。 最后，使用命令 .symopt-2，这会导致调试器包含修饰。 最后一次运行 **x** 命令时，将显示函数名称的修饰版本 **（_typingString）**。

dbgcmd

```
0:000> x /t /d *!*typingstring*
00434420 char [128] TimeTest!typingString = char [128] ""

0:000> .symopt+ 4000

0:000> x /t /d *!*typingstring*
00434420 <NoType> TimeTest!typingString = <no type information>

0:000> .symopt- 2

0:000> x /t /d *!*typingstring*
00434420 <NoType> TimeTest!_typingString = <no type information>
```

使用 DBH 工具查看公共符号和专用符号

查看符号的另一种方法是使用 DBH 工具。 使用 `/?` 选项显示帮助选项。

dbgcmd

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64>dbh /?
dbh dbghelp shell
usage: dbh [-n] [-c] [-d] [-?] [-??] [-p] [targetmodule] [command]
       [-n]             display noisy symbol spew
       [-d]             use decorated publics
       [-p:XXXX]        attaches to process ID XXXX
       [-s:SSSS]        set symbol path to SSSS
       [-c]             callbacks return false
       [targetmodule]   load symbols for specified module
       [command]        execute command and exit
       [-?]             display these usage instructions
       [-??]            display detailed usage instructions
```

使用 Tlist 等工具列出进程 ID，使用 -p 选项附加到现有进程。

dbgcmd

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64>dbh -p:4308
```

DBH 使用与调试器相同的符号选项。 与调试器一样，DBH 默认保留 SYMOPT_PUBLICS_ONLY 并 SYMOPT_NO_PUBLICS 关闭，默认情况下会打开 SYMOPT_UNDNAME 和 SYMOPT_AUTO_PUBLICS 。 这些默认值可由命令行选项或 DBH 命令替代。

以下示例使用 DBH 命令 **addr 414fe0 三** 次。 首次使用默认符号选项，因此信息取自私有符号数据。 请注意，这包括有关函数 **fgets** 的地址、大小和数据类型的信息。 接下来，使用命令 symopt +4000，这会导致 DBH 忽略专用符号数据。 当 **再次运行 addr 414fe0** 时，将使用公共符号表;这一次，函数 **fgets** 没有大小和数据类型信息。 最后，使用命令 symopt -2，这会导致 DBH 包含修饰。 当最后一次运行 **addr 414fe0** 时，将显示函数名称 **_fgets**的修饰版本。

dbgcmd

```
pid:4308 mod:TimeTest[400000]: addr 414fe0

fgets
   name : fgets
   addr :   414fe0
   size : 113
  flags : 0
   type : 7e
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagFunction (5)
  index : 7d

pid:4308 mod:TimeTest[400000]: symopt +4000

Symbol Options: 0x10c13
Symbol Options: 0x14c13

pid:4308 mod:TimeTest[400000]: addr 414fe0

fgets
   name : fgets
   addr :   414fe0
   size : 0
  flags : 0
   type : 0
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagPublicSymbol (a)
  index : 7f

pid:4308 mod:TimeTest[400000]: symopt -2
```

```
Symbol Options: 0x14c13
Symbol Options: 0x14c11

pid:4308 mod:TimeTest[400000]: addr 414fe0

_fgets
   name : _fgets
   addr :   414fe0
   size : 0
  flags : 0
   type : 0
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagPublicSymbol (a)
  index : 7f
```

其他信息

有关符号的其他信息，请参阅 符号语法和符号匹配、 符号选项、 符号状态缩写 和 延迟符号加载。

可移植 PDB 符号

项目 • 2023/10/26

从 Windows 调试器版本 1.0.2007.01003 开始，支持可移植 PDB 符号。 可移植符号可用于向使用符号的所有常用调试器命令提供信息，例如 x (检查符号) 、 dt (Display Type) 和 dx (Display Debugger 对象模型表达式) 。 有关可移植 PDB 格式的常规信息，请参阅 GitHub 上的 可移植 PDB 。

可移植 PDB (Program Database) 格式

可移植 PDB (Program Database) 格式描述了公共语言基础结构 (CLI 编译器生成的调试信息的编码，) 语言并由调试器和其他工具使用。 该格式基于 ECMA-335 分区 II 元数据标准。 它使用相同的物理表和流布局和编码来扩展其架构。

ECMA-335-II 第 24 章介绍了数据的物理布局，可移植 PDB 调试元数据格式不会对基本结构进行任何更改。 有关 ECMA-335 的详细信息，请参阅 标准 ECMA-335 公共语言基础结构 。

有关可移植 PDB 格式的完整信息，请参阅 可移植 PDB v1.0：格式规范 。

用于读取可移植 PDB 文件的代码示例

有关读取可移植 PDB 文件的代码示例，请参阅 GitHub 上的 Microsoft.DiaSymReader.PortablePdb 。

可移植 PDB 的此读取器实现 DiaSymReader 接口，例如 ISymUnmanagedReader 和 ISymUnmanagedBinder。 有关这些 .NET 接口的详细信息，请参阅 诊断符号存储 (非托管 API 参考) 。

另请参阅

符号和符号文件

公共和专用符号

符号语法和符号匹配

项目 • 2023/10/26

符号允许你直接操作正在调试的程序使用的令牌。 例如，可以使用命令 **bp** main在函数 **main**设置断点，或使用命令 **dd MyInt L1** 显示整数变量 MyInt。

在许多情况下，符号可用作调试器命令中的参数。 大多数数值参数都支持此操作，某些文本参数也支持此操作。 除了符号语法的一般规则外，还有适用于上述每种情况的符号语法规则。

常规符号语法规则

符号名称由一个或多个字符组成，但始终以字母、下划线 (_) 、问号 (**?**) 或美元符号 ($) 开头。

符号名称可由模块名称限定。 感叹号 (**!**) 将模块名称与符号（（例如 **mymodule!** **main**）） 分隔开来。 如果未使用模块名称，则符号仍可使用感叹号作为前缀。 使用没有模块名称的感叹号可能特别有用，即使对于局部变量，也可以向调试器命令指示参数是名称而不是十六进制数。 例如，变量 **fade** 将由 dt (Display Type) 命令读取为地址，除非它以感叹号为前缀或使用 -n 选项。 但是，若要指定符号是本地符号，请在符号前面加上美元符号（$），感叹号（！），就像 **在 $!** **lime** 中一样。

符号名称完全不区分大小写。 这意味着调试器无法正确理解程序中是否存在 **myInt** 和 **MyInt** ;引用其中一个命令的任何命令都可以访问另一个命令，而不管该命令的大小写方式如何。

数值表达式中的符号语法

调试器理解两种不同类型的表达式：Microsoft 宏汇编程序 (MASM) 表达式和 C++ 表达式。 就符号而言，这两种语法形式的不同之处如下：

- 在 MASM 表达式中，每个符号都解释为一个地址。 根据符号所引用的内容，它将是全局变量、局部变量、函数、段、模块或任何其他已识别标签的地址。
- 在 C++ 表达式中，每个符号都根据其类型进行解释。 根据符号所引用的内容，它可能被解释为整数、数据结构、函数指针或任何其他数据类型。 与 C++ 数据类型（（如未修改的模块名称））不对应的符号将导致语法错误。

有关何时以及如何使用每种类型的语法的说明，请参阅 计算表达式。

如果使用 MASM 表达式语法，任何可解释为十六进制数或寄存器（（例如 **BadFeed**、**ebX**））的符号都应始终以感叹号作为前缀。 这可确保调试器将其识别为符号。

ss (Set Symbol Suffix) 命令可用于设置符号后缀。 这会指示调试器自动将“A”或“W”追加到它无法找到的任何符号名称。

许多 Win32 例程同时存在于 ASCII 和 Unicode 版本中。 这些例程的名称末尾通常分别追加一个“A”或“W”。 在搜索这些符号时，使用符号后缀将有助于调试器。

默认情况下，后缀匹配不处于活动状态。

文本表达式中的符号语法

符号可用于某些命令的文本参数-例如， **bm (设置断点)** 和 **x (检查符号)** 。

这些文本参数支持各种通配符和说明符。 有关详细信息，请参阅字符串通配符语法 。 除了标准字符串通配符之外，用于指定符号的文本表达式还可以以前导下划线作为前缀。 将此与符号匹配时，调试器会将此视为任何数量的下划线，甚至为零。

在文本表达式中匹配符号时，不使用符号后缀。

延迟符号加载

项目 • 2024/07/08

默认情况下，加载目标模块时不会实际加载符号信息。 相反，调试器会根据需要加载符号。 这称为 *延迟符号加载* 或 *延迟符号加载*。 启用此选项后，每当遇到无法识别的符号时，调试器将加载符号。

更改符号路径时，例如使用 **.sympath（设置符号路径）** 命令，所有加载的带导出符号的模块都会延迟重新加载。 如果新路径不再包含用于加载 PDB 符号的原始路径，则具有完整 PDB 符号的模块符号将延迟重新加载。 如果新路径仍然包含 PDB 符号文件的原始路径，则不会延迟重新加载这些符号。

禁用延迟符号加载时，进程启动速度可能会慢得多，因为每当加载模块时都会读取所有符号。

在 WinDbg 中，可以使用“调试”**菜单上的“解析未限定符号”选项**，为没有模块前缀的符号修改延迟符号加载行为。

可以使用 ld （加载符号） **命令或 .reload （重载模块） 命令和 /f 选项替代延迟的符号加载。** 这些强制立即加载指定的符号，尽管会延迟加载其他符号。

默认情况下，已启用延迟符号加载。 在 CDB 和 KD 中 **，** **-s** 命令行选项 将关闭此选项。 也可以使用 tools.ini 文件中的 *LazyLoad* 变量在 CDB 中将其关闭。 调试器运行后，可以使用 .symopt+0x4 或 **.symopt-0x4 分别打开或关闭**此选项。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

避免调试器搜索不需要的符号

项目 • 2024/02/14

上次更新时间：

- 2007 年 5 月 27 日

调试驱动程序时，你到达了一个有趣的断点，只是让调试器暂停很长一段时间，同时它会尝试加载你不拥有的驱动程序的符号，甚至对手头调试任务并不重要。 这是怎么回事？

默认情况下，调试器会根据需要加载符号。 （这称为延迟符号加载或延迟符号加载。）每当调试器执行调用符号显示命令时，调试器都查找符号。 如果在当前上下文中设置了无效的监视变量（例如当前堆栈帧中不存在的函数参数或局部变量），则在上下文发生更改时，在断点上可能会发生这种情况。 如果只是错误键入符号名称或执行无效的调试器命令，调试器将开始查找匹配的符号，则也可能发生这种情况。

为什么这有时需要这么久？ 这取决于符号名称是限定还是不限定。 限定的符号名称前面有包含符号的模块的名称，例如 myModule！myVar。 未限定的符号名称未指定模块名称，例如 myOtherVar。

对于限定名称，调试器将在指定的模块中查找符号，如果模块尚未加载，则加载模块（假设模块存在并包含符号）。 这种情况相当快。

对于非限定名称，调试器不“知道”哪个模块包含符号，因此它必须查找所有这些模块。 调试器首先检查符号的所有已加载模块，然后，如果它与任何加载模块中的符号不匹配，调试器将继续搜索，方法是加载所有卸载的模块，从下游存储开始，以符号服务器结尾（如果使用一个模块）。 显然，这可能需要很多时间。

如何防止自动加载非限定符号

SYMOPT_NO_UNQUALIFIED_LOADS选项在搜索未限定符号时禁用或启用调试器自动加载模块。 设置 **SYMOPT_NO_UNQUALIFIED_LOADS** 并且调试器尝试匹配未限定的符号时，它只搜索已加载的模块，并在与符号不匹配时停止搜索，而不是加载卸载的模块以继续其搜索。 此选项不会影响搜索限定名称。

默认情况下，SYMOPT_NO_UNQUALIFIED_LOADS处于关闭状态。 若要激活此选项，请使用 **-snul** 命令行选项，或者在调试器运行时，分别使用 **.symopt+0x100** 或 **.symopt-0x100** 打开或关闭该选项。

若要查看SYMOPT_NO_UNQUALIFIED_LOADS**的效果**，请尝试以下试验：

1. 使用 -n **命令行选项激活干扰符号加载（SYMOPT_DEBUG），或者，如果调试器正在运行，请使用 .symopt+0x80000000 或！ sym 干扰调试器扩展命令。**

SYMOPT_DEBUG指示调试器显示有关其搜索符号的信息，例如加载每个模块的名称;如果调试器找不到文件，则显示错误消息。

2. 指示调试器评估不存在的符号（例如，类型 **为？ asdasdasd**）。 调试器在搜索不存在的符号时应报告大量错误。
3. **使用 .symopt+0x100 激活SYMOPT_NO_UNQUALIFIED_LOADS。**
4. 重复步骤 2。 调试器应仅搜索加载的模块来查找不存在的符号，并且应更快地完成任务。
5. 若要禁用 **SYMOPT_DEBUG**，请使用 **.symopt-0x80000000 或！ sym quiet** 调试器扩展命令。

许多选项可用于控制调试器加载和使用符号的方式。 有关符号选项的完整列表以及如何使用它们，请参阅 Windows 调试工具提供的联机文档中的“设置符号选项”。 最新版本的 Windows 调试工具包可从 Web 免费下载，也可以从 Windows DDK、平台 SDK 或客户支持诊断 CD 安装包。

该怎么办?

- 若要加快符号搜索速度，请尽可能在断点和调试器命令中使用限定的名称。 如果想要查看已知模块中的符号，请使用模块名称对其进行限定;如果不知道符号的位置，请使用不限定的名称。 对于局部变量和函数参数，请使用 **$** 模块名称（例如 *$！MyVar*。
- 若要诊断符号加载缓慢的原因，请使用 -n **命令行选项激活干扰符号加载（SYMOPT_DEBUG），或者，如果调试器已在运行，请使用 .symopt+0x80000000 或！ sym 干扰调试器扩展命令。**
- 若要防止调试程序在卸载的模块中搜索符号，请使用 -snul 命令行选项激活 **SYMOPT_NO_UNQUALIFIED_LOADS**，或者，如果调试器已在运行，请使用 **.symopt+0x100。**
- 若要显式加载调试会话所需的模块，请使用 .reload **或 ld 等**调试器命令。

另请参阅

下载 WDK

Windows 调试工具

Windows 调试入门

符号选项

项目 • 2023/06/15

有许多选项可用于控制符号的加载和使用方式。 可以通过多种方式设置这些选项。

下表列出了这些符号选项：

标志	选项名称	调试器中的默认值	DBH 中的默认值
0x1	SYMOPT_CASE_INSENSITIVE	启用	开
0x2	SYMOPT_UNDNAME	启用	开
0x4	SYMOPT_DEFERRED_LOADS	开	关闭
0x8	SYMOPT_NO_CPP	关闭	关闭
0x10	SYMOPT_LOAD_LINES	在 KD 和 CDB 中关闭 在 WinDbg 中打开	开
0x20	SYMOPT_OMAP_FIND_NEAREST	开	关闭
0x40	SYMOPT_LOAD_ANYTHING	关闭	关闭
0x80	SYMOPT_IGNORE_CVREC	关闭	关闭
0x100	SYMOPT_NO_UNQUALIFIED_LOADS	关闭	关闭
0x200	SYMOPT_FAIL_CRITICAL_ERRORS	开	关闭
0x400	SYMOPT_EXACT_SYMBOLS	关闭	启用
0x800	SYMOPT_ALLOW_ABSOLUTE_SYMBOLS	关闭	启用
0x1000	SYMOPT_IGNORE_NT_SYMPATH	关闭	关闭
0x2000	SYMOPT_INCLUDE_32BIT_MODULES	关闭	关闭
0x4000	SYMOPT_PUBLICS_ONLY	关闭	关闭
0x8000	SYMOPT_NO_PUBLICS	关闭	关闭
0x10000	SYMOPT_AUTO_PUBLICS	启用	开
0x20000	SYMOPT_NO_IMAGE_SEARCH	开	关闭

标志	选项名称	调试器中的默认值	DBH 中的默认值
0x40000	SYMOPT_SECURE	关闭	关闭
0x80000	SYMOPT_NO_PROMPTS	在 KD 和 CDB 中启用 在 WinDbg 中关闭	关
0x80000000	SYMOPT_DEBUG	关闭	关闭

更改符号选项设置

.sympt (设置符号选项) 命令可用于更改或显示符号选项设置。 此外，许多命令行参数和命令可用于更改这些设置;这些列在单独的 SYMOPT_*XXX* 节中。

还可以使用 **-sflags**命令行选项一次性控制所有设置。 此选项后面可以跟一个十进制数，也可以后跟一个以 **0x** 为前缀的十六进制数。 建议使用十六进制，因为符号标志以这种方式正确对齐。 使用此方法时请谨慎，因为它设置整个位域，并会替代所有符号处理程序默认值。 例如， **-sflags 0x401** 不仅会打开SYMOPT_EXACT_SYMBOLS和SYMOPT_CASE_INSENSITIVE，而且还会关闭默认情况下打开的所有其他选项！

当启动这些程序时，总标志位的默认值在 WinDbg 中0x30237，在 CDB 和 KD 中0xB0227，在 DBH 工具中0x10C13。

SYMOPT_CASE_INSENSITIVE

此符号选项会导致所有对符号名称的搜索不区分大小写。

默认情况下，此选项在所有调试器中处于打开状态。 调试器运行后，可以分别使用 **.symopt+0x1** 或 .symopt-0x1 打开或关闭它。

默认情况下，此选项在 DBH 中处于打开状态。 运行 DBH 后，可以分别使用 symopt +1 或 symopt -1 打开或关闭它。

SYMOPT_UNDNAME

此符号选项会导致公共符号名称在显示时取消修饰，并导致对符号名称的搜索忽略符号修饰。 无论此选项是否处于活动状态，都不会修饰私有符号名称。 有关符号名称修饰的信息，请参阅 公共符号和专用符号。

默认情况下，此选项在所有调试器中处于打开状态。 调试器运行后，可以分别使用 **.symopt+0x2** 或 .symopt-0x2打开或关闭调试器。

默认情况下，此选项在 DBH 中处于打开状态。 如果使用 -d 命令行选项，则会关闭它。 运行 DBH 后，可以分别使用 symopt +2 或 symopt -2 打开或关闭它。

SYMOPT_DEFERRED_LOADS

此符号选项称为 *延迟符号加载* 或 *延迟符号加载*。 当它处于活动状态时，在加载目标模块时不会实际加载符号。 相反，调试器会根据需要加载符号。 有关详细信息，请参阅延迟符号加载 。

默认情况下，此选项在所有调试器中处于打开状态。 在 CDB 和 KD 中，-s 命令行选项将关闭此选项。 也可以使用 tools.ini文件中的 **LazyLoad** 变量在 CDB 中将其关闭。 调试器运行后，可以分别使用 **.symopt+0x4** 或 .symopt-0x4打开或关闭此选项。

默认情况下，此选项在 DBH 中处于关闭状态。 运行 DBH 后，可以分别使用 symopt +4 或 symopt -4 打开或关闭它。

SYMOPT_NO_CPP

此符号选项关闭 C++ 转换。 设置此符号选项后，在所有符号中将 **替换为__** ： 。

默认情况下，此选项在所有调试器中处于关闭状态。 可以使用 -snc 命令行选项激活它。 调试器运行后，可以分别使用 **.symopt+0x8** 或 .symopt-0x8 将其打开或关闭。

默认情况下，此选项在 DBH 中处于关闭状态。 运行 DBH 后，可以分别使用 symopt +8 或 symopt -8 打开或关闭它。

SYMOPT_LOAD_LINES

此符号选项允许从源文件中读取行号信息。 必须启用此选项，源调试才能正常工作。

在 KD 和 CDB 中，此选项默认处于关闭状态;在 WinDbg 中，此选项默认处于打开状态。 在 CDB 和 KD 中，-lines 命令行选项将打开此选项。 调试器运行后，可以分别使用 **.symopt+0x10** 或 .symopt-0x10 打开或关闭调试器。 也可以使用 **.lines (切换源行支持)** 命令打开和关闭它。

默认情况下，此选项在 DBH 中处于打开状态。 运行 DBH 后，可以分别使用 symopt +10 或 symopt -10 将其打开或关闭。

SYMOPT_OMAP_FIND_NEAREST

如果代码已优化，并且预期位置没有符号，则此选项会导致改用最近的符号。

默认情况下，此选项在所有调试器中处于打开状态。 调试器运行后，可以分别使用 **.symopt+0x20** 或 .symopt-0x20 打开或关闭它。

默认情况下，此选项在 DBH 中处于打开状态。 运行 DBH 后，可以分别使用 symopt +20 或 symopt -20 将其打开或关闭。

SYMOPT_LOAD_ANYTHING

当尝试匹配符号时，此符号选项可降低符号处理程序的挑剔度。

默认情况下，此选项在所有调试器中处于关闭状态。 调试器运行后，可以分别使用 **.symopt+0x40** 或 .symopt-0x40打开或关闭调试器。

默认情况下，此选项在 DBH 中处于关闭状态。 运行 DBH 后，可以分别使用 symopt +40 或 symopt -40 打开或关闭它。

SYMOPT_IGNORE_CVREC

此符号选项会导致符号处理程序在搜索符号时忽略已加载图像标头中的 CV 记录。

默认情况下，此选项在所有调试器中处于关闭状态。 可以使用 -sicv 命令行选项激活它。 调试器运行后，可以分别使用 **.symopt+0x80** 或 .symopt-0x80打开或关闭它。

默认情况下，此选项在 DBH 中处于关闭状态。 运行 DBH 后，可以分别使用 symopt +80 或 symopt -80 将其打开或关闭。

SYMOPT_NO_UNQUALIFIED_LOADS

此符号选项禁用符号处理程序自动加载模块。 当设置此选项并且调试器尝试匹配符号时，它将仅搜索已加载的模块。

此选项可用于防止符号名称错误。 通常，键入错误的符号将导致调试器在搜索所有卸载的符号文件时暂停。 当此选项处于活动状态时，在加载的模块中找不到键入错误的符号，然后搜索将终止。

默认情况下，此选项在所有调试器中处于关闭状态。 可以使用 -snul 命令行选项激活它。 调试器运行后，可以分别使用 **.symopt+0x100** 或 .symopt-0x100 打开或关闭调试器。

默认情况下，此选项在 DBH 中处于关闭状态。 运行 DBH 后，可以分别使用 symopt +100 或 symopt -100 打开或关闭它。

SYMOPT_FAIL_CRITICAL_ERRORS

此符号选项会导致禁止显示文件访问错误对话框。

如果此选项处于关闭状态，则符号加载期间遇到的文件访问错误（如“驱动器未就绪”）将导致出现对话框。 如果此选项处于打开状态，则会取消这些框，并且所有访问错误都会收到“失败”响应。

默认情况下，此选项在所有调试器中处于打开状态。 可以使用 -sdce 命令行选项将其停用。 调试器运行后，可以分别使用 **.symopt+0x200** 或 .symopt-0x200 打开或关闭调试器。

默认情况下，此选项在 DBH 中处于关闭状态。 运行 DBH 后，可以分别使用 symopt +200 或 symopt -200 将其打开或关闭。

SYMOPT_EXACT_SYMBOLS

此符号选项使调试器对所有符号文件执行严格评估。

启用此选项时，即使符号文件与符号处理程序的期望之间的细微差异也会导致忽略符号。

默认情况下，此选项在所有调试器中处于关闭状态。 可以使用 -ses 命令行选项激活它。 调试器运行后，可以分别使用 **.symopt+0x400** 或 .symopt-0x400打开或关闭它。

-failinc 命令行选项还会打开SYMOPT_EXACT_SYMBOLS。 此外，如果要调试用户模式小型转储或内核模式微型转储，-failinc 将阻止调试器加载无法映射其映像的任何模块。

默认情况下，此选项在 DBH 中处于打开状态。 运行 DBH 后，可以分别使用 symopt +400 或 symopt -400 将其打开或关闭。

SYMOPT_ALLOW_ABSOLUTE_SYMBOLS

此符号选项允许 DbgHelp 读取存储在内存中绝对地址的符号。 在大多数情况下，不需要此选项。

默认情况下，此选项在所有调试器中处于关闭状态。 调试器运行后，可以分别使用 **.symopt+0x800** 或 .symopt-0x800 将其打开或关闭。

默认情况下，此选项在 DBH 中处于打开状态。 运行 DBH 后，可以分别使用 symopt +800 或 symopt -800 打开或关闭它。

SYMOPT_IGNORE_NT_SYMPATH

此符号选项会导致调试器忽略符号路径和可执行映像路径的环境变量设置。

默认情况下，此选项在所有调试器中处于关闭状态。 可以使用 -sins 命令行选项激活它。 但是，调试器运行后，它不能由 **.symopt** 控制，因为环境变量仅在启动时读取。

默认情况下，此选项在 DBH 中处于关闭状态，DBH 在所有情况下都会忽略此选项。

SYMOPT_PUBLICS_ONLY

此符号选项会导致 DbgHelp 忽略私有符号数据，并仅搜索公共符号表以获取符号信息。 这在添加对这些类型的支持之前模拟 DbgHelp 的行为。 请参阅 公共和专用符号。

默认情况下，此选项在所有调试器中处于关闭状态。 调试器运行后，可以分别使用 **.symopt+0x4000** 或 .symopt-0x4000 将其打开或关闭。

默认情况下，此选项在 DBH 中处于关闭状态。 如果使用 -d 命令行选项，则会打开它。 运行 DBH 后，可以分别使用 symopt +4000 或 symopt -4000 将其打开或关闭。

SYMOPT_NO_PUBLICS

此符号选项可防止 DbgHelp 搜索公共符号表。 这可以加快符号枚举和符号搜索的速度。 如果只关心搜索速度，则SYMOPT_AUTO_PUBLICS选项通常优于此选项。 有关公共符号表的信息，请参阅 公共符号和专用符号。

默认情况下，此选项在所有调试器中处于关闭状态。 调试器运行后，可以分别使用 **.symopt+0x8000** 或 .symopt-0x8000 将其打开或关闭。

默认情况下，此选项在 DBH 中处于关闭状态。 运行 DBH 后，可以分别使用 symopt +8000 或 symopt -8000 将其打开或关闭。

SYMOPT_AUTO_PUBLICS

此符号选项导致 DbgHelp 仅作为最后手段搜索 .pdb 文件中的公共符号表。 如果在搜索私有符号数据时找到任何匹配项，则不会搜索公共符号。 这可以提高符号搜索速度。

默认情况下，此选项在所有调试器中都处于打开状态。 可以使用 -sup 命令行选项将其停用。 调试器运行后，可以分别使用 **.symopt+0x10000** 或 .symopt-0x10000 将其打开或关闭。

默认情况下，此选项在 DBH 中处于打开状态。 如果使用 -d 命令行选项，则会将其关闭。 运行 DBH 后，可以分别使用 symopt +10000 或 symopt -10000 将其打开或关闭。

SYMOPT_NO_IMAGE_SEARCH

此符号选项可防止 DbgHelp 在加载符号时在磁盘上搜索映像副本。

默认情况下，此选项在所有调试器中都处于打开状态。 调试器运行后，可以分别使用 **.symopt+0x20000** 或 .symopt-0x20000 将其打开或关闭。

默认情况下，此选项在 DBH 中处于关闭状态。 运行 DBH 后，可以分别使用 symopt +20000 或 symopt -20000 打开或关闭它。

SYMOPT_SECURE

(内核模式仅) 此符号选项指示 安全模式 是否处于活动状态。

在所有调试器中，安全模式默认处于关闭状态。 可以使用 -secure 命令行选项激活它。 如果调试器正在运行，处于休眠模式，并且尚未建立任何调试服务器，则可以使用 **.symopt+0x40000** 或 **.secure (激活安全模式) 启用安全模式** 。

默认情况下，此选项在 DBH 中处于关闭状态。 运行 DBH 后，可以分别使用 symopt +40000 或 symopt -40000 将其打开或关闭。

激活安全模式后，永远无法将其关闭。

SYMOPT_NO_PROMPTS

此符号选项禁止来自代理服务器的身份验证对话框。 这可能会导致 SymSrv 无法访问 Internet 上的符号存储。

有关详细信息，请参阅 防火墙和代理服务器。

在 KD 和 CDB 中，此选项默认处于打开状态;在 WinDbg 中，此选项默认处于关闭状态。 调试器运行后，可以分别使用 **.symopt+0x80000 或** . symopt-0x80000 打开或关闭调试器，然后使用 **.reload (Module)** 命令。 还可以使用 **！ sym 提示关闭** 和 **！ sym prompts** 扩展命令打开和关闭它，后跟 **.reload (Module)** 命令。

默认情况下，此选项在 DBH 中处于关闭状态。 运行 DBH 后，可以分别使用 symopt +80000 或 symopt -80000 打开或关闭它。

-SYMOPT_DEBUG

此符号选项启用 *干扰符号加载*。 这会指示调试器显示有关搜索符号的信息。

每个符号文件的名称将在加载时显示。 如果调试器无法加载符号文件，它将显示一条错误消息。 .pdb 文件的错误消息将以文本方式显示。 .dbg 文件的错误消息将以错误代码的形式显示;winerror.h 文件中介绍了这些代码。

如果仅加载图像文件以恢复符号标头信息，则也会显示此信息。

默认情况下，此选项在所有调试器中处于关闭状态。 可以使用 -n 命令行选项激活它。 调试器运行后，可以分别使用 **.symopt+0x80000000** 或 .symopt-0x80000000将其打开或关闭。 还可以使用 **！ sym noisy** 和 **！ sym quiet** 扩展命令来打开和关闭它。

注意 此选项不应与干扰 *源* 加载混淆 -- 由 **.srcnoisy (干扰源加载)** 命令控制。

默认情况下，此选项在 DBH 中处于关闭状态。 可以使用 -n 命令行选项激活它。 运行 DBH 后，可以分别使用 symopt +80000000 或 symopt -80000000 将其打开或关闭。 还可以通过使用详细打开和详细关闭命令来打开和关闭它。

符号状态缩写

项目 • 2023/10/26

可以使用 lm (List Loaded Modules) 命令、 ！ lmi 扩展名或 WinDbg 的 **Debug | 来确定符号文件类型及其加载状态"模块** "菜单命令。

其中每个都显示有关加载的模块及其符号的信息。

这些命令生成的显示器中使用以下缩写：

缩写	含义
递 延	模块已加载，但调试器未尝试加载符号。 符号将在需要时加载。 有关详细信息，请参阅延迟符号加载 。
#	符号文件和可执行文件的时间戳或校验和不匹配。
T	时间戳缺失、不可访问或等于零。
C	校验和缺失、不可访问或等于零。
直径	符号文件是通过调试接口访问 (DIA) 加载的。
导出	未找到实际的符号文件，因此从二进制文件的导出表中提取了符号信息。
M	符号文件和可执行文件的时间戳或校验和不匹配。 但是，由于符号选项设置，符号文件已加载。
PERF	此二进制文件包含 性能优化的代码。 标准地址算术可能无法生成正确的结果。
剥夺	已从映像文件中删除调试信息。
Pdb	符号采用 .pdb 格式。
COFF	符号采用通用对象文件格式 (COFF) 符号格式。

调试时的符号问题

项目 • 2023/06/15

符号无效或缺失是调试器问题的最常见原因之一。 当你看到某种问题时，你需要找出你是否有符号问题。

在某些情况下，解决方案涉及获取正确的符号文件。 在其他情况下，只需重新配置调试器即可识别已有的符号文件。 但是，如果无法获取正确的符号文件，则需要解决此问题并以更有限的方式调试目标。

本节包括：

验证符号

匹配符号名称

读取移出页面的标头中的符号

将 PEB 移出页面时映射符号

不使用符号调试用户模式进程

调试性能优化的代码

验证符号

项目 • 2023/04/14

符号问题可能以多种方式出现。 也许堆栈跟踪显示不正确的信息，或者无法识别堆栈中函数的名称。 或者，调试器命令可能无法理解模块、函数、变量、结构或数据类型的名称。

如果怀疑调试器未正确加载符号，可以执行几个步骤来调查此问题。

首先，使用 **lm (列出加载的模块)** 命令显示包含符号信息的已加载模块的列表。 此命令最有用的形式如下：

```
dbgcmd

0:000> lml
```

如果使用 WinDbg， **则调试 |“模块** ”菜单命令也允许你查看此信息。

请特别注意在这些显示器中看到的任何备注或缩写。 有关这些内容的解释，请参阅 符号状态缩写。

如果没有看到正确的符号文件，首先要做的是检查符号路径：

```
dbgcmd

0:000> .sympath
Current Symbol Path is: d:\MyInstallation\i386\symbols\retail
```

如果符号路径错误，请修复它。 如果使用内核调试器，请确保本地 %WINDIR% 不在符号路径中。

然后使用 **.reload (重新加载模块) 命令重新加载** 符号：

```
dbgcmd

0:000> .reload ModuleName
```

如果符号路径正确，则应激活 *干扰模式* ，以便可以查看 **dbghelp** 正在加载的符号文件。 然后重新加载模块。 有关如何激活干扰模式的信息，请参阅 设置符号选项 。

下面是 Microsoft Windows 符号的“干扰”重载的示例：

```
dbgcmd
```

```
kd> !sym noisy
kd> .reload nt
 1: Kernel Version 2081 MP Checked
 2: Kernel base = 0x80400000 PsLoadedModuleList = 0x80506fa0
 3: DBGHELP: FindExecutableImageEx-> Looking for
D:\MyInstallation\i386\ntkrnlmp.exe...mismatched timestamp
 4: DBGHELP: No image file available for ntkrnlmp.exe
 5: DBGHELP: FindDebugInfoFileEx-> Looking for
 6: d:\MyInstallation\i386\symbols\retail\symbols\exe\ntkrnlmp.dbg... no
file
 7: DBGHELP: FindDebugInfoFileEx-> Looking for
 8: d:\MyInstallation\i386\symbols\retail\symbols\exe\ntkrnlmp.pdb... no
file
 9: DBGHELP: FindDebugInfoFileEx-> Looking for
d:\MyInstallation\i386\symbols\retail\exe\ntkrnlmp.dbg... OK
10: DBGHELP: LocatePDB-> Looking for
d:\MyInstallation\i386\symbols\retail\exe\ntkrnlmp.pdb... OK
11: *** WARNING: symbols checksum and timestamp is wrong 0x0036a4ea
0x00361a83 for ntkrnlmp.exe
```

符号处理程序首先查找与它尝试加载 (第三行和第四行) 模块匹配的图像。 图像本身并非始终是必需的，但如果存在不正确的图像，符号处理程序通常会失败。 这些行显示调试器在 **D:\MyInstallation\i386\ntkrnlmp.exe**找到图像，但时间戳不匹配。 由于时间戳不匹配，因此搜索将继续。 接下来，调试器查找与加载的映像匹配的 .dbg 文件和 .pdb 文件。 这些位于第 6 行到第 10 行。 第 11 行指示即使加载了符号，图像的时间戳也与 (即符号错误) 不匹配。

如果符号搜索遇到灾难性故障，则会看到以下格式的消息：

dbgcmd

```
ImgHlpFindDebugInfo(00000000, module.dll, c:\MyDir;c:\SomeDir, 0823345, 0)
failed
```

这可能是由文件系统故障、网络错误和 .dbg 文件损坏等项引起的。

诊断符号加载错误

在干扰模式下，调试器在无法加载符号文件时可能会输出错误代码。 .dbg 文件的错误代码列在 winerror.h 中。 .pdb 错误代码来自另一个来源，最常见的错误以纯英文文本打印。

winerror.h 中 .dbg 文件的一些常见错误代码如下：

0xB
ERROR_BAD_FORMAT

0x3
ERROR_PATH_NOT_FOUND

0x35
ERROR_BAD_NETPATH

由于网络错误，符号文件可能无法加载。 如果看到ERROR_BAD_FORMAT或ERROR_BAD_NETPATH并且正在从网络上的另一台计算机加载符号，请尝试将符号文件复制到主计算机，并将其路径放入符号路径中。 然后尝试重新加载符号。

验证搜索路径和符号

让“c：\MyDir;c：\SomeDir”表示符号路径。 应在何处查找调试信息？

如果二进制文件已去除调试信息（例如 Windows 的免费版本），请先在以下位置查找 .dbg 文件：

```
dbgcmd
```

```
c:\MyDir\symbols\exe\ntoskrnl.dbg
c:\SomeDir\symbols\exe\ntoskrnl.dbg
c:\MyDir\exe\ntoskrnl.dbg
c:\SomeDir\exe\ntoskrnl.dbg
c:\MyDir\ntoskrnl.dbg
c:\SomeDir\ntoskrnl.dbg
current-working-directory\ntoskrnl.dbg
```

接下来，在以下位置查找 .pdb 文件：

```
dbgcmd
```

```
c:\MyDir\symbols\exe\ntoskrnl.pdb
c:\MyDir\exe\ntoskrnl.pdb
c:\MyDir\ntoskrnl.pdb
c:\SomeDir\symbols\exe\ntoskrnl.pdb
c:\SomeDir\exe\ntoskrnl.pdb
c:\SomeDir\ntoskrnl.pdb
current-working-directory\ntoskrnl.pdb
```

请注意，在搜索 .dbg 文件时，调试器会相互搜索 MyDir 和 SomeDir 目录，但在 .pdb 搜索中，它不会。

Windows XP 和更高版本的 Windows 不使用任何 .dbg 符号文件。 有关详细信息，请参阅符号和符号文件 。

不匹配的内部版本

在经常更新的计算机上调试故障时，最常见的问题之一是不同版本的符号不匹配。 此问题的三个常见原因是：指向错误生成的符号、使用没有相应符号的私有生成的二进制文件，以及使用多处理器计算机上的单处理器硬件抽象级别 (HAL) 和内核符号。 前两个只是匹配二进制文件和符号的问题：可以通过将 hal*.dbg 和 ntkrnlmp.dbg 重命名为 hal.dbg 和 ntoskrnl.dbg 来更正第三个。

若要了解目标计算机上安装的 Windows 版本，请使用 **vertarget (显示目标计算机版本)** 命令：

dbgcmd

```
kd> vertarget
Windows XP Kernel Version 2505 UP Free x86 compatible
Built by: 2505.main.010626-1514
Kernel base = 0x804d0000 PsLoadedModuleList = 0x80548748
Debug session time: Mon Jul 02 14:41:11 2001
System Uptime: 0 days 0:04:53
```

测试符号

测试符号更加困难。 它涉及到在调试器上验证堆栈跟踪，并查看调试输出是否正确。 下面是一个可尝试的示例：

dbgcmd

```
kd> u videoprt!videoportfindadapter2
Loading symbols for 0xf2860000     videoprt.sys ->   videoprt.sys

VIDEOPRT!VideoPortFindAdapter2:
f2856f42 55              push    ebp
f2856f43 8bec            mov     ebp,esp
f2856f45 81ecb8010000    sub     esp,0x1b8
f2856f4b 8b4518          mov     eax,[ebp+0x18]
f2856f4e 53              push    ebx
f2856f4f 8365f400        and     dword ptr [ebp-0xc],0x
f2856f53 8065ff00        and     byte ptr [ebp-0x1],0x0
f2856f57 56              push    esi
```

u 命令在 videoprt.sys 中取消组合 videoportfindadapter 字符串。 符号在调试器上是正确的，因为常见的堆栈命令（如 **push** 和 **mov** ）显示在堆栈上。 大多数函数首先使用基本指针 (ebp) 或堆栈指针 (esp) 进行添加、子或推送操作。

当符号无法正常工作时，这通常是显而易见的。 Glintmp.sys在此示例中没有符号，因为 **Glintmp** 旁边未列出函数：

dbgcmd

```
kd> kb
Loading symbols for 0xf28d0000     videoprt.sys ->   videoprt.sys
Loading symbols for 0xf9cdd000      glintmp.sys ->   glintmp.sys
*** ERROR: Symbols could not be loaded for glintmp.sys
ChildEBP RetAddr  Args to Child
f29bf1b0 8045b5fa 00000001 0000a100 00000030
ntoskrnl!RtlpBreakWithStatusInstruction
f29bf1b0 8044904e 00000001 0000a100 00000030
ntoskrnl!KeUpdateSystemTime+0x13e
f29bf234 f28d1955 f9b7d000 ffafb2dc f9b7d000
ntoskrnl!READ_REGISTER_ULONG+0x6
f29bf248 f9cde411 f9b7d000 f29bf2b0 f9ba0060
VIDEOPRT!VideoPortReadRegisterUlong+0x27
00000002 00000000 00000000 00000000 00000000 glintMP+0x1411 [No function
listed.]
```

为此堆栈跟踪加载了错误的生成符号。 请注意，前两个调用没有列出函数。 此堆栈跟踪类似于win32k.sys绘制矩形的问题：

dbgcmd

```
1: kd>
1: kd> kb                       [Local        9:50 AM]
Loading symbols for 0xf22b0000       agpcpq.sys ->   agpcpq.sys
*** WARNING: symbols checksum is wrong 0x0000735a 0x00000000 for agpcpq.sys
*** ERROR: Symbols could not be loaded for agpcpq.sys
Loading symbols for 0xa0000000       win32k.sys ->   win32k.sys
*** WARNING: symbols checksum is wrong 0x00191a41 0x001995a9 for win32k.sys
ChildEBP RetAddr  Args to Child
be682b18 f22b372b 82707128 f21c1ffc 826a70f8 agpCPQ+0x125b [No function
listed.]
be682b4c a0140dd4 826a72f0 e11410a8 a0139605 agpCPQ+0x372b [No function
listed.]
be682b80 a00f5646 e1145100 e1cee560 e1cee560 win32k!vPatCpyRect1_6x6+0x20b
00000001 00000000 00000000 00000000 00000000
win32k!RemoteRedrawRectangle+0x32
```

下面是正确的堆栈跟踪。 问题在于AGP440.sys。 堆栈跟踪上出现的第一个项通常出错。 请注意，win32k.sys矩形错误消失：

dbgcmd

```
1: kd> kb                       [Local        9:49 AM]
ChildEBP RetAddr  Args to Child
be682b18 f22b372b 82707128 f21c1ffc 826a70f8 agpCPQ!AgpReleaseMemory+0x88
be682b30 f20a385c 82703638 e183ec68 00000000
agpCPQ!AgpInterfaceReleaseMemory+0x8b
be682b4c a0140dd4 826a72f0 e11410a8 a0139605
```

```
VIDEOPRT!AgpReleasePhysical+0x44
be682b58 a0139605 e1cee560 e11410a8 a00e5f0a win32k!OsAGPFree+0x14
be682b64 a00e5f0a e1cee560 e11410a8 e1cee560 win32k!AGPFree+0xd
be682b80 a00f5646 e1145100 e1cee560 e1cee560 win32k!HeapVidMemFini+0x49
be682b9c a00f5c20 e1cee008 e1cee008 be682c0c win32k!vDdDisableDriver+0x3a
be682bac a00da510 e1cee008 00000000 be682c0c
win32k!vDdDisableDirectDraw+0x2d
be682bc4 a00da787 00000000 e1843df8 e1843de8
win32k!PDEVOBJ__vDisableSurface+0x27
be682bec a00d59fb 00000000 e1843de8 00000000
win32k!PDEVOBJ__vUnreferencePdev+0x204
be682c04 a00d7421 e1cee008 82566a98 00000001 win32k!DrvDestroyMDEV+0x30
be682ce0 a00a9e7f e1843e10 e184a008 00000000
win32k!DrvChangeDisplaySettings+0x8b3
be682d20 a008b543 00000000 00000000 00000000
win32k!xxxUserChangeDisplaySettings+0x106
be682d48 8045d119 00000000 00000000 00000000
win32k!NtUserChangeDisplaySettings+0x48
be682d48 77e63660 00000000 00000000 00000000 ntkrnlmp!KiSystemService+0xc9
```

有用的命令和扩展

以下命令和扩展可用于跟踪符号问题：

lm (列出加载的模块)

列出所有模块，并提供这些模块中所有符号的加载状态。

! dh image-header-base

显示从 *image-header-base* 开始的已加载图像的标头信息。

.reload /n

重新加载所有内核符号。

.reload [image-name]

(CDB 或 WinDbg 仅) 重新加载映像 *映像名称*的符号。 如果未指定 *图像名称*，则为所有图像重新加载符号。 (更改符号路径后，必须重新加载符号。)

! sym noisy

为符号加载启用详细模式。 这可用于获取有关模块加载的信息。 有关详细信息，请参阅设置符号选项 。

.sympath [new-symbol-path]

设置新的符号路径，或显示当前符号路径。 有关详细信息，请参阅符号路径 。

如果内核符号正确，但未获得完整的堆栈，则以下命令可能也很有用：

X*!

这将列出当前加载了符号的模块。 如果内核符号正确，这很有用。

.reload /user

这会尝试重新加载所有用户模式符号。 如果在一个进程运行时加载了符号，并且稍后在另一个进程中发生了中断，则在执行内核调试时需要执行此操作。 在这种情况下，除非执行此命令，否则不会加载新进程中的用户模式符号。

X wdmaud！*start*

这将仅列出名称包含“start”字符串的 **wdmaud** 模块中的符号。 这有一个优点，即它强制重新加载 **wdmaud** 中的所有符号，但只显示包含“start”符号的符号。 (这意味着列表较短，但由于其中总是有一些带有“开始”的符号，因此将进行一些验证，以确定是否发生了加载。)

验证符号的另一种有用方法是取消组合代码。 大多数函数首先使用基本指针 (**ebp**) 或堆栈指针 (**esp** 或 **sp**) 进行添加、子或推送操作。 尝试取消组合 (**U 函数**) 堆栈上的某些函数， (偏移零) 以验证符号。

网络和端口问题

符号文件以及连接到调试器时会出现问题。 遇到问题时，请记住以下几点：

- 确定调试电缆在测试系统上连接到哪个 COM 端口。
- 检查测试系统的boot.ini设置。 查找 **/debug** 开关，检查波特率和 COM 端口设置。
- 如果符号文件是通过网络访问的，则网络问题可能会干扰调试。
- .dll和.sys具有相同名称的文件 (，例如， mga64.sys 和 mga64.dll) 如果它们未分离到符号树的正确目录中，则调试器会混淆。
- 内核调试器并不总是喜欢将生成符号文件替换为私有符号文件。 双检查符号路径，对行为不当的符号执行 **.reload***FileName*。 **！dlls** 命令有时很有用。

问题和误解

问： 我已成功加载符号，但堆栈似乎出错了。 调试器是否损坏?

答： 不一定。 问题的最可能原因是符号不正确。 完成本部分中概述的步骤以确定是否已加载有效符号。 不要假设因为某些操作有效，因此具有有效的符号。 例如，你很可能能够执行 **dd nt！ntbuildnumber** 或 **u nt！具有错误符号的 KeInitializeProcess** 。 使用上述过程验证它们是否正确。

问： 调试器是否仍使用不正确的符号?

答： 是和否。 通常，你可以避开不严格匹配的符号。 例如，以前 Windows 版本中的符号通常在某些情况下起作用，但对于何时工作以及何时不起作用没有规则。

问： 我在内核调试器中已停止，我想查看用户模式进程的符号。 我能做到吗？

答： 主要。 此方案的支持很差，因为内核调试器没有保留足够的信息来跟踪每个进程的模块负载，但有一个合理的解决方法。 若要加载用户模式模块的符号，请执行 **.reload -user** 命令。 这将加载当前上下文的用户模式模块。

问： 以下消息是什么意思？

```
*** WARNING: symbols checksum and timestamp is wrong 0x0036d6bf 0x0036ab55 for ntkrnlmp.exe
```

答： 这意味着ntkrnlmp.exe的符号错误。

匹配符号名称

项目 • 2023/06/15

在某些情况下，符号的实际名称将替换为另一种形式，这可能会导致符号匹配问题。 在公共符号和专用符号之间更改或对文件使用 MS-DOS 兼容性 8.3 短名称时，这种情况最常发生。

公共符号与专用符号匹配

在公共符号和私有符号之间切换有时可能会导致符号匹配问题。 通常，公共符号和相应的私有符号具有相同的名称，具有不同的符号修饰。 但在某些情况下，它们可能具有完全不同的名称。 在这种情况下，可能需要显式引用这两个名称。 例如，可以设置两个断点：一个在公共符号上，另一个在私有符号上。 有关更多详细信息，请参阅 公共符号和专用符号。

MS-DOS 兼容性 8.3 短名称符号匹配

有时会为具有很长名称的文件提供自动生成的 MS-DOS 兼容性 8.3 短名称。 根据用于创建符号文件和调试的工具和选项，存储在图像的调试记录中的文件名可以是长名称或这些短名称之一。 如果使用短名称，这可能会导致符号匹配问题，因为分配的短名称依赖于系统。

例如，假设有两个文件，Longfilename1.pdb 和 Longfilename2.pdb。 如果它们位于同一目录中，则其 MS-DOS 兼容性 8.3 名称为 Longfi~1.pdb，另一个名称为 Longfi~2.pdb。 如果它们不放在同一目录中，则它们都是 Longfi~1.pdb。 因此，如果不小心复制了关联的 .pdb 文件，短文件名可能会更改，从而导致符号匹配问题。 有关更多详细信息，请参阅 文件系统引用和符号文件。

读取移出页面的标头中的符号

项目 • 2023/10/26

内核调试器必须读取每个已加载模块映像的标头，以便知道哪些符号对应于该模块。

如果将模块的标头分页到磁盘，调试器将不会加载此模块的符号。 如果对调试过程至关重要的模块发生这种情况，则可能是一个关键问题。

以下过程可用于解决此问题。

获取分页的页眉的符号

1. 创建内核本身的第二个副本。 将其放在网络共享上可能最容易。
2. 将此共享的根目录追加到符号路径。 有关更改 符号路径 的方法，请参阅符号路径。
3. 使用 **.reload (重载模块)** 命令。
4. 使用 **！ sym 干扰扩展** 命令查看更详细的输出。 如果使用此方法，你将能够看到哪些符号是从目标计算机上的模块映像加载的，以及哪些符号是从内核模块的副本加载的。

必须谨慎使用此方法，因为调试器无法验证文件副本是否与原始副本实际匹配。 因此，网络共享上使用的 Windows 版本与目标计算机上所用的版本匹配至关重要。

此方法仅用于内核模式调试。 操作系统能够在用户模式调试 (期间所需的任何标头中分页，除非存放分页文件的磁盘已卸载或) 无法访问。

下面是使用此方法的示例：

dbgcmd

```
kd> .reload
Connected to Windows XP 2268 x86 compatible target, ptr64 FALSE
Loading Kernel Symbols
..........Unable to read image header for dmload.sys at fe0be000 - NTSTATUS
0xC0000001
..........Unable to read image header for dmboot.sys at fda93000 - NTSTATUS
0xC0000001
.....................................Unable to read image header for fdc.sys
at fdfc2000 - NTSTATUS 0xC0000001
...Unable to read image header for flpydisk.sys at fde4a000 - NTSTATUS
0xC0000001
.Unable to read image header for Fs_Rec.SYS at fe0c8000 - NTSTATUS
0xC0000001
```

```
.Unable to read image header for Null.SYS at fe2c4000 - NTSTATUS 0xC0000001
...................Unable to read image header for win32k.sys at a0000000 -
NTSTATUS 0xC0000001
..Unable to read image header for dxg.sys at a0194000 - NTSTATUS 0xC0000001
.......Unable to read image header for ati2draa.dll at a01a4000 - NTSTATUS
0xC0000001
..Unable to read image header for ParVdm.SYS at fe116000 - NTSTATUS
0xC0000001
.......
Loading unloaded module list
..............
Loading User Symbols
Unable to retrieve the PEB address. This is usually caused
by being in the wrong process context or by paging
```

请注意，许多图像具有不可访问的标头。 检查以下示例中 (其中一个文件中的符号，fs_rec.sys)：

dbgcmd

```
kd> x fs_rec!*
*** ERROR: Module load completed but symbols could not be loaded for
fs_rec.sys
```

这些标头显然是分页的。因此，需要将正确的图像添加到符号路径：

dbgcmd

```
kd> .sympath+ \\myserver\myshare\symbols\x86fre\symbols
Symbol search path is:
symsrv*symsrv.dll*c:\localcache*https://msdl.microsoft.com/download/symbols;
\\myserver\myshare\symbols\x86fre\symbols

kd> .reload
Connected to Windows XP 2268 x86 compatible target, ptr64 FALSE
Loading Kernel Symbols
..........Unable to read image header for dmload.sys at fe0be000 - NTSTATUS
0xC0000001
..........Unable to read image header for dmboot.sys at fda93000 - NTSTATUS
0xC0000001
.....................................Unable to read image header for fdc.sys
at fdfc2000 - NTSTATUS 0xC0000001
...Unable to read image header for flpydisk.sys at fde4a000 - NTSTATUS
0xC0000001
.Unable to read image header for Fs_Rec.SYS at fe0c8000 - NTSTATUS
0xC0000001
.Unable to read image header for Null.SYS at fe2c4000 - NTSTATUS 0xC0000001
...................Unable to read image header for win32k.sys at a0000000 -
NTSTATUS 0xC0000001
..Unable to read image header for dxg.sys at a0194000 - NTSTATUS 0xC0000001
.......Unable to read image header for ati2draa.dll at a01a4000 - NTSTATUS
```

```
0xC0000001
..Unable to read image header for ParVdm.SYS at fe116000 - NTSTATUS
0xC0000001
.......
Loading unloaded module list
.............
Loading User Symbols
Unable to retrieve the PEB address. This is usually caused
by being in the wrong process context or by paging
```

出现相同的警告，但符号本身现在可以访问：

dbgcmd

```
kd> x fs_Rec!*
fe0c8358  Fs_Rec!_imp___allmul
fe0c8310  Fs_Rec!_imp__IoCreateDevice
fe0c835c  Fs_Rec!_imp___allshr
........
fe0c8360  Fs_Rec!ntoskrnl_NULL_THUNK_DATA
fe0c832c  Fs_Rec!_imp__KeSetEvent
fe0c9570  Fs_Rec!_NULL_IMPORT_DESCRIPTOR
```

将 PEB 移出页面时映射符号

项目 • 2023/10/26

若要加载符号，调试器会查看操作系统加载的模块列表。 指向用户模式模块列表的指针是存储在进程环境块中的项之一， (PEB) 。

为了回收内存，内存管理器可能会分页出用户模式数据，为其他进程或内核模式组件提供空间。 分页的用户模式数据可能包括 PEB 数据结构。 如果没有此数据结构，调试器无法确定要为哪些映像加载符号。

注意 这仅影响用户模式模块的符号文件。 内核模式模块和符号不受影响，因为它们在不同的列表中进行跟踪。

假设用户模式模块已映射到当前进程，并且你想要修复该模块的符号。 在模块的虚拟地址范围内查找任何地址。 例如，假设一个模块映射到包含地址 7f78e9e000F 的虚拟地址范围。 输入以下命令。

dbgcmd

```
3: kd> !vad 7f78e9e000F 1
```

命令输出显示有关模块的虚拟地址描述符 (VAD) 的信息。 命令输出还包括可用于加载模块符号的 Reload 命令字符串。 重载命令字符串包括记事本模块的起始地址 (000007f7'8e9e0000) 和大小 (32000) 。

dbgcmd

```
VAD @ fffffa80056fb960
...
Reload command: .reload notepad.exe=000007f7`8e9e0000,32000
```

若要加载符号，请输入重载命令字符串中提供的命令。

dbgcmd

```
.reload notepad.exe=000007f7`8e9e0000,32000
```

下面是使用略有不同的技术的另一个示例。 该示例演示如何在 PEB 分页时使用 **！ vad** 扩展映射符号。基本思路是查找相关 DLL 的起始地址和大小，以便可以使用 **.reload** 命令加载必要的符号。 假设当前进程的地址0xE0000126'01BA0AF0，并且你想要修复它的符号。 首先，使用 **！ process** 命令获取虚拟地址描述符 (VAD) 根地址：

dbgcmd

```
kd> !process e000012601ba0af0 1
PROCESS e000012601ba0af0
    SessionId: 2  Cid: 0b50    Peb: 6fbfffde000  ParentCid: 0efc
    DirBase: 079e8461  ObjectTable: e000000600fbceb0  HandleCount: 360.
    Image: explorer.exe
    VadRoot e000012601a35e70 Vads 201 Clone 0 Private 917. Modified 2198.
Locked 0.
...
```

然后使用 **！ vad** 扩展列出与进程关联的 VAD 树。 标记为“EXECUTE_WRITECOPY”的 VAD 属于代码模块。

dbgcmd

```
kd> !vad e000012601a35e70
VAD     level      start      end    commit
...
e0000126019f9790 ( 6)      3fff0    3fff7        -1 Private      READONLY
e000012601be1080 ( 7)   37d9bd30 37d9bd3e         2 Mapped  Exe
EXECUTE_WRITECOPY  <-- these are DLLs
e000012600acd970 ( 5)   37d9bec0 37d9bece         2 Mapped  Exe
EXECUTE_WRITECOPY
e000012601a5cba0 ( 7)   37d9c910 37d9c924         2 Mapped  Exe
EXECUTE_WRITECOPY
...
```

然后再次使用 **！ vad** 扩展查找保存所需 DLL 的分页内存的起始地址和大小。 这确认你已找到正确的 DLL：

dbgcmd

```
kd> !vad e000012601be1080 1

VAD @ e000012601be1080
  Start VPN:      37d9bd30  End VPN: 37d9bd3e  Control Area:
e00001260197b8d0
  First ProtoPte: e0000006013e00a0  Last PTE fffffffffffffffc  Commit Charge
2 (2.)
  Secured.Flink          0  Blink           0  Banked/Extend:        0
  File Offset   0
   ImageMap ViewShare EXECUTE_WRITECOPY

...
        File: \Windows\System32\ExplorerFrame.dll
```

“启动 VPN”字段（在本例中为0x37D9BD30）指示起始虚拟页码。 必须将其乘以页面大小，从而将其转换为实际地址。 可以使用 **？ (Evaluate Expression)** 命令将此值乘以

0x2000，这是示例基于 Itanium 的计算机的页面大小。

dbgcmd

```
kd> ? 37d9bd3e*2000
Evaluate expression: 7676040298496 = 000006fb`37a7c000
```

然后，可以将范围的大小转换为字节：

dbgcmd

```
kd> ? 37d9bd3e-37d9bd30+1 <--   computes the number of pages
Evaluate expression: 15 = 00000000`0000000f
kd> ? f*2000
Evaluate expression: 122880 = 00000000`0001e000
```

因此，ExplorerFrame.dll 从地址0x000006Fb'37A7C000 开始，0x1E000个字节大。 可以使用以下代码加载其符号：

dbgcmd

```
kd> .reload /f ExplorerFrame.dll=6fb`37a7c000,1e000
```

不使用符号调试用户模式进程

项目 • 2023/06/15

在针对用户模式故障启动调试器之前，在出错的计算机上具有符号非常重要。 但是，有时调试器在没有符号的情况下启动。 如果问题很容易重现，则只需复制符号并重新运行。 但是，如果问题可能不会再次出现，仍然可以从故障中收集一些信息：

1. 若要弄清楚地址的含义，需要一台与错误匹配的计算机。 它应具有同一平台 (x86 或 x64)，并使用相同版本的 Windows 进行加载。

2. 配置计算机后，将用户模式符号和要调试的二进制文件复制到新计算机上。

3. 在无符号计算机上启动 CDB 或 WinDbg。

4. 如果不知道哪个应用程序在无符号计算机上失败，请发出 **| (进程状态)** 命令。 如果这未提供名称，请在无符号计算机上进入 KD，然后执行 **！ process 0 0**，查找 CDB 命令提供的进程 ID。

5. 如果设置了两个调试器（一个具有未命中错误的符号，另一个已命中错误，但没有符号），则会在无符号计算机上发出 **k (Display Stack Backtrace)** 命令。

6. 在具有符号的计算机上，针对无符号堆栈上给出的每个地址发出 **u (Unassemble)** 命令。 这将为你提供无符号计算机上错误的堆栈跟踪。

7. 通过查看堆栈跟踪，可以看到调用中涉及的模块和函数名称。

调试性能优化的代码

项目 • 2024/06/11

Microsoft 采用某些技术来重新排列编译和链接的代码，从而以更高效的效率执行。 这些技术以训练场景为基础，针对内存层次结构对组件进行优化。

由此产生的优化减少了分页（和页面错误），并提高了代码和数据之间的空间局部性。 它解决了因原始代码定位不当而带来的关键性能瓶颈问题。 经过这种优化的组件，其代码或函数中的数据块可能会被移动到二进制文件的不同位置。

在经过这些技术优化的模块中，代码和数据块所在的内存地址往往与正常编译和链接后的地址不同。 此外，函数可能被分割成许多非连续的块，以便最常用的代码路径可以在同一页面上彼此靠近。

因此，函数（或任何符号）加上偏移量并不一定具有与非优化代码相同的含义。

调试性能优化的代码

进行调试时，在已加载符号的任何模块上使用 !lmi 扩展命令，即可查看模块是否进行了性能优化：

dbgcmd

```
0:000> !lmi ntdll
Loaded Module Info: [ntdll]
         Module: ntdll
   Base Address: 77f80000
     Image Name: ntdll.dll
   Machine Type: 332 (I386)
     Time Stamp: 394193d2 Fri Jun 09 18:03:14 2000
       CheckSum: 861b1
Characteristics: 230e stripped perf
Debug Data Dirs: Type Size     VA  Pointer
                 MISC  110,     0,   76c00 [Data not mapped]
     Image Type: DBG      - Image read successfully from symbol server.
                 c:\symbols\dll\ntdll.dbg
    Symbol Type: DIA PDB  - Symbols loaded successfully from symbol server.
                 c:\symbols\dll\ntdll.pdb
```

在此输出中，请注意“特征”行上的术语 **perf**。 这表明该性能优化已应用于 ntdll.dll。

调试器能够理解没有偏移的函数或其他符号；这样就可以顺利地在函数或其他标签上设置断点。 但是，反汇编操作的输出结果可能会令人困惑，因为反汇编会反映出优化器所做的更改。

由于调试器会尽量贴近原始代码，因此可能会看到一些有趣的结果。 处理性能优化代码的经验法则很简单，就是不能在优化代码上执行可靠的地址运算。

以下是示例：

dbgcmd

```
kd> bl
 0 e f8640ca6     0001 (0001) tcpip!IPTransmit
 1 e f8672660     0001 (0001) tcpip!IPFragment

kd> u f864b4cb
tcpip!IPTransmit+e48:
f864b4cb f3a4              rep     movsb
f864b4cd 8b75cc            mov     esi,[ebp-0x34]
f864b4d0 8b4d10            mov     ecx,[ebp+0x10]
f864b4d3 8b7da4            mov     edi,[ebp-0x5c]
f864b4d6 8bc6              mov     eax,esi
f864b4d8 6a10              push    0x10
f864b4da 034114            add     eax,[ecx+0x14]
f864b4dd 57                push    edi
```

从断点列表中可以看到，**IPTransmit** 的地址是 0xF8640CA6。

如果将该函数中位于 0xF864B4CB 处的一段代码反汇编，输出结果表明该处已超过函数起始处 0xE48 字节。 但是，如果从该地址减去函数的基数，则实际偏移量似乎是 0xA825。

现在的情况是：调试器确实显示了对从 0xF864B4CB 开始的二进制指令的反汇编。 但调试器不是通过简单的减法计算偏移量，而是尽可能显示优化前原始代码中函数入口的偏移量。 该值为 0xE48。

另一方面，如果尝试查看 **IPTransmit**+0xE48，则会看到如下内容：

dbgcmd

```
kd> u tcpip!iptransmit+e48
tcpip!ARPTransmit+d8:
f8641aee 0856ff            or      [esi-0x1],dl
f8641af1 75fc              jnz     tcpip!ARPTransmit+0xd9 (f8641aef)
f8641af3 57                push    edi
f8641af4 e828eeffff        call    tcpip!ARPSendData (f8640921)
f8641af9 5f                pop     edi
f8641afa 5e                pop     esi
f8641afb 5b                pop     ebx
f8641afc c9                leave
```

这里发生的情况是，调试器识别出符号 **IPTransmit** 等同于地址 0xF8640CA6，并且命令解析器执行简单的加法运算，发现 0xF8640CA6 + 0xE48 = 0xF8641AEE。 然后，此地址

将作为 **u（取消汇编）**命令的参数。 但在对该位置进行分析后，调试器就会发现这并不等于 **IPTransmit** 加上 0xE48 的偏移量。 事实上，它根本不是此函数的一部分。 相反，它对应于函数 **ARPTransmit** 加上 0xD8 的偏移量。

之所以发生这种情况，是因为性能优化无法通过地址运算来逆转。 虽然调试器可以获取地址并推导出其原始符号和偏移量，但它没有足够的信息来获取符号和偏移量并将其转换为正确的地址。 因此，在这种情况下，反汇编并不会起作用。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

自定义符号存储和符号服务器

项目 • 2023/12/29

正确设置符号进行调试可能是一项很有挑战性的任务，尤其是对于内核调试而言。 这通常需要知道计算机上所有产品的名称和版本。 调试器必须能够找到与产品版本和服务包对应的每个符号文件。

这可能会导致符号路径极长，路径中包含一长串的目录。 为了简化协调符号文件的这些困难，可以将符号文件收集到 *符号存储*中，然后由 *符号服务器*访问。

*符号存储*是符号文件、索引和工具的集合，管理员可使用该工具添加和删除文件。 符号存储还可以包含可执行图像文件。

根据唯一参数（如时间戳和图像大小）为文件编制索引。 符号存储还可以保存可使用符号服务器提取的可执行图像文件。 Windows 调试工具包含名为 SymStore 的符号存储创建工具。

调试器使用符号 *服务器访问符号*存储中的文件。 Windows 调试工具包括符号存储创建工具 SymStore 和符号服务器 *SymSrv*。 它还包括一个工具 SymProxy，用于在网络上设置 HTTP 符号存储，以充当调试器可能需要访问的所有符号存储的代理。

符号服务器使调试器能够自动从符号存储检索正确的符号文件，而无需用户知道产品名称、版本或内部版本号。 Windows 调试工具包含一个名为 *SymSrv 的*符号服务器。 通过在符号路径中包含特定文本字符串来激活符号服务器。 每次调试程序需要为新加载的模块加载符号时，都会调用符号服务器来查找相应的符号文件。 有关使用符号服务器的信息，请参阅 Microsoft 公共符号。

如果要对符号搜索使用与 SymSrv 提供的方法不同的方法，则可以开发自己的符号服务器 DLL。 有关实现此类符号服务器的详细信息，请参阅 其他符号服务器。

本节包括：

HTTP 符号存储

文件共享 (SMB) 调试器符号服务器

符号存储文件夹树

其他符号存储

安装 Windows 符号文件

Windows 更新的脱机符号

如果不设置自己的符号存储，而只是想要使用公共 Microsoft 符号存储，请参阅 Microsoft 公共符号。

符号实用工具

以下实用工具可以在较大的软件开发项目中分发和控制符号。

展开表

实用程序	说明
SymProxy	用于将基于 HTTP 的符号存储配置为充当客户端计算机和其他符号存储之间的代理。
SymStore	SymStore (symstore.exe) 工具用于创建符号存储。
AgeStore	AgeStore 工具 (agestore.exe) 根据上次访问日期删除目录或目录树中的文件。
DBH	DBH 工具 (dbh.exe) 是一种命令行工具，用于显示有关符号文件内容的信息。
PDBCopy	PDBCopy 工具 (pdbcopy.exe) 是一种命令行工具，用于从符号文件中删除专用符号信息。
SymChk	SymChk (Microsoft 符号检查器工具) Symchk.exe 是一种将可执行文件与符号文件进行比较的程序。

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

Windows driver documentation 反馈

Windows driver documentation 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

HTTP 符号存储

项目 • 2024/05/27

通过使用 symsrv.dll（随调试器一起提供）支持的 SRV 协议，可以使用 HTTP（而不只是 UNC/SMB）来访问符号存储。

当防火墙不允许客户端和服务器之间使用 SMB 时，通常使用 HTTP 来代替 SMB。 生产环境和实验室环境就是一个很好的例子。

由于其只读性质，HTTP 符号服务器不能成为符号路径链中的下游存储。 符号服务器代理（ISAPI 筛选器）可绕过这一限制。 SymProxy 会使用预先配置的上游符号存储将丢失的文件下载到服务器的文件系统中。 筛选器会将文件下载到文件系统，允许 IIS 将文件下载到客户端，从而恢复符号存储链的概念。 有关详细信息，请参阅 SymProxy。

将 IIS 配置为符号存储相对简单，因为符号文件只是用作静态文件。 唯一的非默认设置是 MIME 类型的配置，以允许以二进制流的形式下载符号文件。 这可以通过在符号文件夹的虚拟目录中使用“*”通配符来实现。

要通过 Internet 访问符号存储，必须配置包含符号文件和 Internet Information Services (IIS) 的目录。

注意由于 IIS 配置为提供符号文件的方式，因此不建议将同一服务器实例用于任何其他目的。 通常情况下，符号服务器所需的安全设置对于其他用途（例如面向外部的商务服务器）并无意义。 请确保此处所述的示例配置对环境具有意义，并根据具体需要进行适当调整。

创建符号目录

首先选择要用作符号存储的目录。 在示例中，我们将此目录称为 c:\symstore，网络服务器的名称为 \SymMachineName。

有关如何填充符号存储的详细信息，请参阅 SymStore 和符号存储文件夹树。

配置 IIS

必须配置 Internet Information Services (IIS)，通过创建虚拟目录和配置 MIME 类型为符号提供服务。 完成上述操作后，即可选择身份验证方法。

创建虚拟目录

1. 打开“Internet Information Services (IIS)管理器”。

2. 导航到**网站**。

3. 右键单击**默认网站**或正在使用的网站名称，然后选择**添加虚拟目录...**。

4. 为**别名**键入**符号**，然后单击**下一步**。

   为便于管理，建议文件夹、共享和虚拟目录都使用相同的名称。

5. 在**路径**中输入 **c:\SymStore**，然后单击**下一步**。

6. 单击“确定”完成添加虚拟目录。

为服务器执行一次子目录配置过程。 请注意，这是一个全局设置，会影响未托管在网站根文件夹中的应用程序。

子目录配置

1. 导航到[**计算机**]。

2. 打开**配置编辑器**。

3. 导航到**系统 ApplicationHost/sites**。

4. 展开 **virtualDirectoryDefaults**。

5. 将 **allowSubDirConfig** 设置为 *False*。

为服务器执行一次此过程。 请注意，这是一个全局设置，会影响未托管在网站根文件夹中的应用程序。

可选择让符号文件可浏览

1. 导航到[**计算机**] | **站点** | [**网站**] | **符号**。

2. 双击中心窗格中的“目录浏览”。

3. 在右侧窗格中单击“启用”。

下载内容的 MIME 类型需要设置为 application/octet-stream，这样 IIS 才能传送所有符号文件。

配置 MIME 类型

1. 右键单击“符号”虚拟目录，然后选择“属性”。

2. 选择“HTTP 标头”。

3. 单击 **MIME 类型**。

4. 单击**新建**。

5. 对于“扩展”，请键入 *。

6. 对于“MIME 类型”，请键入 **application/octet-stream**。

7. 要退出“MIME 类型”对话框，请单击“确定”。

8. 若要退出**符号属性**，请单击**确定**。

可以编辑 web.config 文件，为符号配置 MIME 类型。 此方法可清除继承的 MIME 类型，并添加一个万能通配符 * MIME 类型。 当 MIME 类型在某些 IIS 配置中被继承时，可能需要使用此方法。

使用 web.config 配置 MIME 类型

1. 编辑 web.config 文件，如此处所示。

```
XML
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <directoryBrowse enabled="true" />
        <staticContent>
            <clear />
            <mimeMap fileExtension=".*"
mimeType="application/octet-stream" />
        </staticContent>
    </system.webServer>
</configuration>
```

2. 重启 IIS。

现在 IIS 已准备就绪，可以从符号存储提供各种类型的符号文件。

配置身份验证

可以将 IIS 配置为使用“集成 Windows 身份验证”，这样客户端（例如 windbg.exe）就可以自动对 IIS 进行身份验证，而无需提示最终用户提供凭据。

注意仅在 IIS 上配置 Windows 身份验证，以控制对符号服务器的访问（如果这样适合具体环境）。 如果环境需要，还有其他安全选项可用于进一步控制对 IIS 的访问。

将身份验证方法配置为匿名

1. 启动 **Internet Information Services (IIS) 管理器**。

2. 导航到**[计算机] | 站点 | [网站] | 符号。**

3. 在中间窗格中双击**身份验证。**

4. 在“身份验证和访问控制”下，单击“编辑”。

5. 右键单击“Windows 身份验证”，然后选择“启用”。

6. 对于所有其他身份验证提供程序，右键单击每个提供程序并选择“禁用”。

7. 单击“确定”完成身份验证配置。

如果未列出窗口身份验证，请使用“打开和关闭 Windows 功能”来启用该功能。 在每个版本的 Windows 中，该功能的位置都不尽相同。 在 Windows 8.1/Windows 2012 R2 中，它位于“Internet Information Services 下 | 万维网服务 | 安全”下。

禁用 Kerberos 支持

在连接 IIS 时，SymSrv.dll 不支持 Kerberos 身份验证。 因此，必须在 IIS 中禁用 Kerberos 身份验证，并将 NTLM 设置为唯一的 Windows 身份验证协议。

注意仅当适合环境时，才会禁用 Kerberos 安全性。

使用 appcmd.exe 禁用 Kerberos 支持

1. 打开“命令提示符”窗口

2. 要禁用 Kerberos 并强制使用 NTLM，请使用此命令：

```
控制台

appcmd.exe set config -
section:system.webServer/security/authentication/windowsAuthentication
/+"providers.[value='NTLM']" /commit:apphost
```

3. 要返回启用 Kerberos 的默认值，请使用以下命令：

```
控制台

appcmd.exe set config -
section:system.webServer/security/authentication/windowsAuthentication
/+"providers.[value='Negotiate,NTLM']" /commit:apphost
```

配置 SymSrv 客户端身份验证提示

当 SymSrv 收到身份验证请求时，调试器可以显示身份验证对话框，也可以自动拒绝请求，具体取决于配置方式。 可以使用 !sym prompts on|off 来配置此行为。 例如，要打开提示，请使用此命令。

dbgcmd

```
!sym prompts on
```

要检查当前设置，请使用此命令。

dbgcmd

```
!sym prompts
```

有关详细信息，请参阅 !sym 和防火墙和代理服务器。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

SMB) 调试器符号服务器 (文件共享

项目 • 2023/12/20

若要运行 SMB 符号服务器，请创建文件共享并分配权限，向用户或组授予对文件共享的访问权限。

> ⓘ **备注**
>
> 在当前版本的 Windows 中，可能需要配置或启用特定版本的 SMB。 有关详细信息，请参阅 ***如何在 Windows 中检测、启用和禁用 SMBv1、SMBv2 和 SMBv3***。

创建 SMB 文件共享符号存储

使用 文件资源管理器 或计算机管理创建文件共享并分配权限。

以下部分中的步骤假定符号位于 D：\SymStore\Symbols 中。

文件资源浏览器

使用 文件资源管理器分配文件共享权限：

1. 打开文件资源管理器。
2. 选择并按住 (或右键单击) **D：\SymStore\Symbols** 文件夹，然后选择 **属性**。
3. 选择“ **共享** ”选项卡。
4. 选择“ **高级共享**”。
5. 在 “**高级共享**”中，选中“ **共享此文件夹**” 复选框，然后选择“ **权限**”。
6. 在 “**共享权限**”中，选择“ **所有人**”，然后选择“ **删除**”。
7. 选择“ **添加** ”并输入要访问文件共享的用户或组。
8. 对于添加的每个用户或组，选择“ **允许** ”以分配“完全控制”、“更改”或“读取”权限。
9. 依次选择“应用”、“确定” 。
10. 选择“确定”，然后选择“关闭”。

篆福恨瞶

使用计算机管理分配文件共享权限：

1. 选择并按住 (或右键单击“) ”**开始** “，然后选择“ **计算机管理**”。
2. 在控制台树中，选择“ **系统工具**>”“**共享文件夹**>**共享**”。
3. 选择并按住 (或右键单击“) ”，然后选择“ **新建**>**共享**”。
4. 在 “**创建共享文件夹向导**”中，选择“ **下一步**”。
5. 对于 “**文件夹路径**”，输入 **D：\SymStore\Symbols**，然后选择“ **下一步**”。
6. 选择“**下一页**”。
7. 在 “**共享文件夹权限**”**中**，选择“ **自定义权限**”，然后选择“ **自定义**”。
8. 在 “**共享权限**”中，选择“ **所有人**”，然后选择“ **删除**”。
9. 选择“ **添加** ”并输入要访问文件共享的用户或组。
10. 对于添加的每个用户或组，选择“ **允许** ”以分配“完全控制”、“更改”或“读取”权限。
11. 依次选择“应用”、“确定” 。
12. 选择“ **完成**” 两次。

测试 SMB 文件共享

将调试器配置为使用此符号路径：

```
srv*C:\Symbols*\\MachineName\Symbols
```

若要查看调试器中引用的 PDB 的位置，请使用 `lm` (list modules) 命令。 PDB 的路径都应以 `C:\Symbols`开头。

若要查看从 \\MachineName\Symbols 文件服务器到 C：\Symbols 的符号和图像下载日志，请运行 `!sym noisy` 和 `.reload /f` 。

设置文件共享符号路径

若要将调试器的符号路径 (`.sympath`) 配置为使用文件共享，可以使用多个选项。 符号路径的语法确定符号文件是否在本地缓存以及缓存位置。

直接文件共享 (不使用本地缓存)：

```text
srv*\\MachineName\Symbols
```

将文件共享的文件本地缓存到特定本地文件夹， (例如 C：\Symbols)：

```text
srv*C:\Symbols*\\MachineName\Symbols
```

将文件共享的文件本地缓存到 %DBGHELP_HOMEDIR%\Sym 文件夹：

```text
srv**\\MachineName\Symbols
```

此示例中的第二个“*”表示默认本地服务器缓存。 有关设置符号路径和使用本地缓存的详细信息，请参阅 Windows 调试器的符号路径。

`DBGHELP\_HOMEDIR`如果未设置变量， `DBGHELP\_HOMEDIR`则默认为调试器可执行文件文件夹 (例如 C：\Program Files\Windows Kits\10.0\Debuggers\x86) 和缓存发生在 C：\Program Files\Windows Kits\10.0\Debuggers\x86\Sym 中。

另请参阅

符号存储文件夹树

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

Windows driver documentation 反馈

Windows driver documentation 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

符号存储文件夹树

项目 • 2023/12/29

支持 SMB 和 HTTP 请求的符号存储是驻留在本地磁盘上的文件夹树。

为了简化管理，子文件夹名称 (例如符号) 也可以用作文件共享名称和虚拟目录名称。 如果要添加新的符号存储区，则会在 D：\SymStore 下创建一个新的子文件夹，并将创建一个新的文件共享和该名称的虚拟目录，以便向客户端公开存储。

应仔细选择文件夹树的位置以及磁盘的文件系统。 从内部) 生成服务器和 Internet 缓存文件时，符号存储可能会获得非常大 ((TB) 。 文件夹树应位于能够进行大量读取和少量写入的磁盘上。 文件系统可能会影响性能 - ReFS 的性能可能优于 NTFS，应针对大型部署进行调查。 同样，到服务器的网络速度应该足够快，以便处理来自客户端的负载，以及上游符号存储的负载，以检索缓存填充的符号。

符号存储 Single-Tier 或 Two-Tier 结构

通常，文件放置在单个层目录结构中，其中缓存的每个文件名都有一个子目录。 在每个文件名文件夹下，将创建其他文件夹来存储文件的每个版本。 树将具有以下结构：

控制台

```
D:\SymStore\Symbols\ntdll.dll\...\
D:\SymStore\Symbols\ntdll.pdb\...\
D:\SymStore\Symbols\kernel32.dll\...\
D:\SymStore\Symbols\kernel32.pdb\...\
```

如果要存储大量文件，则可以在符号存储的根目录中使用两层结构。 文件名的前 2 个字母用作中间文件夹名称。

若要使用双层结构，请将名为 index2.txt 的文件放在 D：\SymStore\Symbols 的根目录中。 文件的内容不重要。 如果存在此文件，symsrv.dll 将使用此结构从两层树创建和使用文件：

控制台

```
D:\SymStore\Symbols\nt\ntdll.dll\...\
D:\SymStore\Symbols\nt\ntdll.pdb\...\
D:\SymStore\Symbols\ke\kernel32.dll\...\
D:\SymStore\Symbols\ke\kernel32.pdb\...\
```

如果要在填充符号存储区后转换结构，请使用调试器文件夹中的 convertstore.exe 应用程序。 若要允许该工具正常工作，请在根文件夹中创建名为 000Admin 的文件夹。

convertstore.exe 需要此文件夹，以便它可以控制符号存储的锁定。

另请参阅

HTTP 符号存储

文件共享 (SMB) 调试器符号服务器

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

Windows driver documentation 反馈

Windows driver documentation 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

其他符号存储

项目 • 2023/06/15

可以编写自己的符号存储创建程序，而不是使用 SymStore。

由于 SymStore 事务都记录在 CSV 格式的文本文件中，因此可以利用任何现有的 SymStore 日志文件在自己的数据库程序中使用。

如果计划使用 Windows 调试工具包随附的 SymSrv 程序，建议也使用 SymStore。 这两个程序的汇报将始终一起发布，因此它们的版本将始终匹配。

其他符号服务器 DLL

项目 • 2023/06/14

如果要使用不同的方法进行符号搜索，可以提供自己的符号服务器 DLL，而不是使用 SymSrv。

设置符号路径

实现 SymSrv 以外的符号服务器时，调试器的符号路径的设置方式与使用 SymSrv 的方式相同。 有关符号路径语法的说明，请参阅 Microsoft 公共符号和高级 SymSrv 用法。 唯一需要做出的更改是将字符串 **symsrv.dll** 替换为自己的符号服务器 DLL 的名称。

如果需要，可以在参数中随意使用不同的语法，以指示使用不同技术，例如 UNC 路径、SQL 数据库标识符或 Internet 规范。

实现自己的符号服务器

一种方法是使用与 DbgHelp 通信的代码来查找符号。 每当 DbgHelp 需要新加载的模块的符号时，它都会调用符号服务器来查找相应的符号文件。 符号服务器根据唯一参数（如时间戳或图像大小）查找每个文件。 服务器返回所请求文件的验证路径。

不得更改符号服务器返回的实际符号文件名。 DbgHelp 将符号文件的名称存储在多个位置。 因此，服务器必须返回与请求符号时指定的同名的文件。 需要此限制才能确保符号加载期间显示的符号名称是程序员将识别的符号名称。

对多个符号服务器的限制

DbgHelp 一次仅支持使用一个符号服务器。 符号路径可以包含同一符号服务器 DLL 的多个实例，但不能包含两个不同的符号服务器 DLL。 这并不是什么限制，因为你仍然可以在符号路径中包含符号服务器的多个实例，每个实例都指向不同的符号存储。 但是，如果要在两个不同的符号服务器 DLL 之间切换，则必须每次更改符号路径。

安装自定义符号服务器 DLL

符号服务器安装的详细信息将取决于你的情况。 你可能希望设置一个安装过程，以复制符号服务器 DLL 并自动设置_NT_SYMBOL_PATH环境变量。

根据服务器中使用的技术，可能还需要安装或访问符号数据本身。

可移植 PDB (程序数据库)

可移植 PDB (程序数据库) 格式描述了公共语言基础结构 (CLI 编译器生成的调试信息) 语言以及调试器和其他工具使用的调试信息的编码。 格式基于 ECMA-335 分区 II 元数据标准。 它支持在不同的操作系统和平台上进行操作。 有关详细信息，请参阅 可移植 PDB 符号。

自定义符号存储和符号服务器

有关创建自定义符号服务器的信息，请参阅 自定义符号存储和符号服务器。

Windows 更新的脱机符号

项目 • 2023/06/15

本主题介绍如何使用Windows 更新的脱机符号。 它描述了一个过程，该过程可用于在无法访问 Microsoft 符号服务器的计算机上解码Windows 更新日志。

如果你经常需要这样做，你应该看看设置符号代理服务器是否适用于网络配置。 有关详细信息，请参阅 SymProxy。

以下所有选项都要求你有一台计算机可以连接到 Microsoft 的符号服务器，并且能够将文件复制到具有日志的计算机或从具有日志的计算机复制文件。 无权访问符号服务器的计算机将称为 *脱机* 计算机，而具有访问权限的计算机将称为 *联机* 计算机。

我们建议为每个 OS 内部版本使用单个联机计算机，以便 WU 符号缓存将逐月生成并包含来自多个更新版本的 WU 符号。

如果有权访问与脱机计算机完全相同的修补程序级别的联机计算机，则有两个选项：

- 选项 1：将 ETL 事件日志复制到联机计算机
- 选项 2：将符号复制到脱机计算机

通过在 `winver` 两台计算机上运行 或 `ver` 来验证联机和脱机电脑的版本级别相同。

控制台

```
C:\>ver

Microsoft Windows [Version 10.0.17134.167]
```

如果无法访问具有相同版本的联机计算机，则需要执行一些额外的步骤来创建 SymChk 清单文件，本主题稍后在 选项 3：创建 SymChk 清单文件中进行了介绍。

选项 1：将 ETL 事件日志复制到联机计算机

1. 将所有 WindowsUpdate ETL 文件从 `C:\Windows\logs\WindowsUpdate\` 复制到联机计算机。
2. 在联机计算机上，打开 PowerShell 提示符并运行以下 Get-WindowsUpdateLog PowerShell 命令。

   PowerShell

   ```
   Get-WindowsUpdateLog -ETLPath <path to ETLs>
   ```

这将下载日志分析所需的符号。

选项 2：将符号复制到脱机计算机

1. 在联机计算机上，打开 PowerShell 提示符并运行“Get-WindowsUpdateLog”。 这会缓存日志分析所需的符号。
2. 将 %temp%\WindowsUpdateLog\SymCache 中的所有文件从联机计算机复制到 `%temp%\WindowsUpdateLog\SymCache` 脱机计算机上。
3. 在脱机计算机上，打开 PowerShell 提示符并运行“Get-WindowsUpdateLog”来分析日志。

选项 3：创建 SymChk 清单文件

1. 在脱机计算机上，按照将 清单文件与 SymChk 配合使用 中的步骤操作，在 system32 目录中为这些文件创建清单：

   控制台

   ```
   storewuauth.dll
   wuapi.dll
   wuauclt.exe
   wuaueng.dll
   wuautoappupdate.dll
   wuuhext.dll
   wuuhmobile.dll
   ```

2. 将清单复制到联机计算机。
3. 使用清单文件，使用 SymChk 将符号本地下载到联机电脑。
4. 将传递给 SymChk 的文件夹和符号复制到脱机电脑上的 %temp%\WindowsUpdateLog\SymCache。
5. 在脱机计算机上，打开 PowerShell 提示符并运行“Get-WindowsUpdateLog”来分析日志。

另请参阅

使用符号服务器。

符号路径

调试时的符号问题

在旧版 Windows 中安装 Windows 符号文件

项目 • 2023/06/15

在调试 Windows 内核、驱动程序或应用之前，需要访问正确的符号文件。 获取 Windows 符号的官方方法是使用 Microsoft 符号服务器。 符号服务器根据需要向调试工具提供符号。 从符号服务器下载符号文件后，该文件将缓存在本地计算机上以便快速访问。

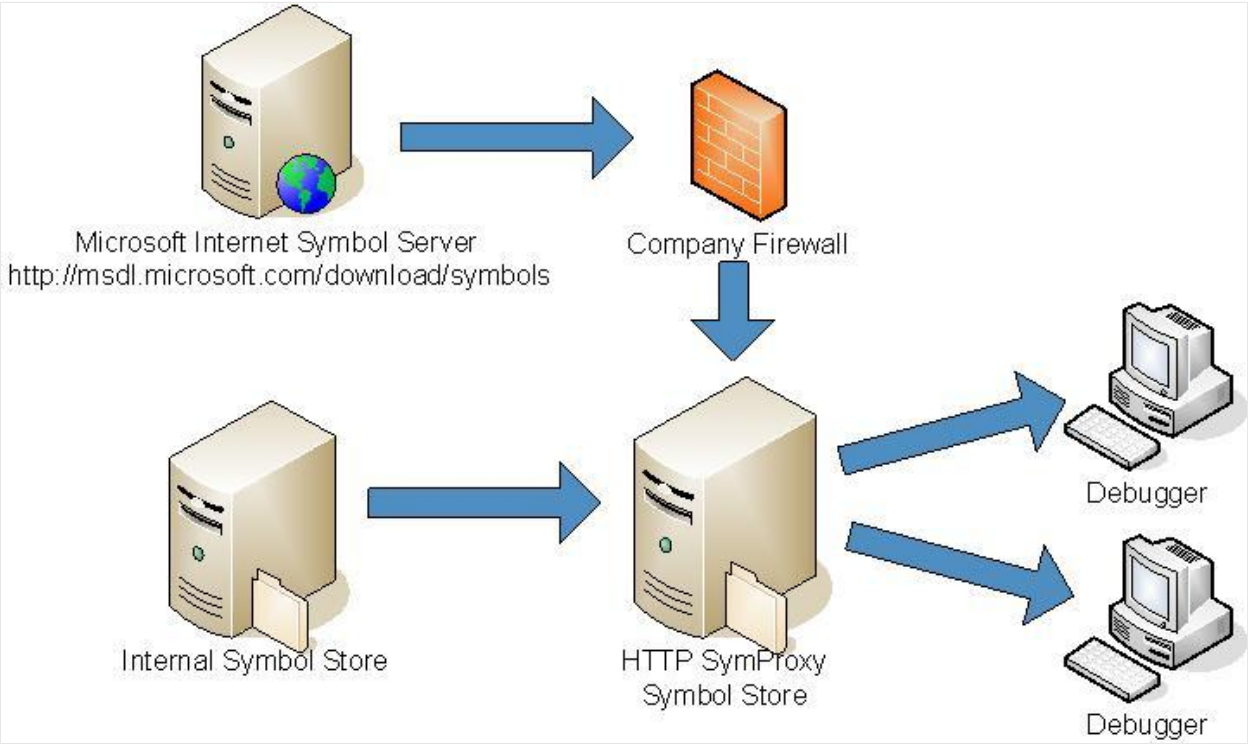
只需使用 **.symfix (设置符号存储路径)** 命令，即可连接到 Microsoft 符号服务器。 有关完整详细信息，请参阅 Microsoft 公共符号。

> **ⓘ 重要**
>
> 我们不再发布适用于 Windows 的脱机符号包。 更快的 Windows 更新节奏意味着 Windows 调试符号很快就会过时。 我们对联机 **Microsoft Symbol Server** 进行了重大改进，其中提供了所有 Windows 版本和更新的符号。 有关此内容的详细信息，请参阅此 **博客文章**。
>
> 有关如何检索未连接到 Internet 的计算机的符号的信息，请参阅 **将清单文件与 SymChk 配合使用**。

SymProxy

项目 • 2023/11/15

可以将基于 HTTP 的符号存储配置为充当客户端计算机和其他符号存储之间的代理。 实现通过名为 SymProxy (Symproxy.dll) 的 ISAPI) 筛选器 (Internet 服务器应用程序编程接口。 SymProxy 服务器可用作 Internet 或公司网络中其他源的网关计算机。 下图显示了一个示例 SymProxy 配置。



SymProxy 在许多情况下很有用。 例如：

- 你正在实验室环境中调试许多系统，其中计算机未连接到公司网络，但符号存储在网络中，必须使用集成 Windows 身份验证 (IWA) 进行访问。
- 企业计算环境包括一个防火墙，该防火墙阻止正在调试的计算机访问 Internet，并且必须从 Internet 网站获取符号。
- 你希望为公司中的所有用户提供单个符号路径，以便他们无需知道或关心符号所在的位置，并且无需用户干预即可添加新的符号存储区。
- 你有一个远程站点，该站点在物理上远离公司的其他资源，并且网络访问速度缓慢。 此系统可用于获取符号并将其缓存到远程站点。

若要安装 SymProxy，必须手动将文件复制到正确的位置，配置注册表，选择网络安全凭据，并将 Internet Information Services (IIS) 。 若要确保正确配置 HTTP 符号存储，请参阅 HTTP 符号存储。

多符号服务器性能注意事项

每个虚拟目录可以与多个 (上游) 符号存储相关联。 单独查询每个符号存储区。 为了获得性能，应在 Internet HTTP 服务器之前处理本地 SMB 服务器。 与调试器符号路径不同，可以在 SymProxy 符号路径中指定多个 HTTP 符号存储。 每个虚拟目录最多支持 10 个条目。

SymProxy 符号路径

SymProxy 将) 符号路径值定义的 (注册表拆分为各个条目，并使用每个条目生成基于 SRV* 的符号路径来检索文件。 它使用虚拟目录的文件夹作为每个查询中的下游存储 - 实际上，将上游存储合并到单个下游符号存储。

SymProxy 使用的) 符号路径生成的 (等效于：

```
dbgcmd

SRV*<Virtual Directory Folder>*<SymbolPath Entry #N>
```

在此示例中，一个 UNC 路径和两个 HTTP 路径与虚拟目录相关联，以合并来自公司符号服务器、Microsoft 和第三方 (Contoso) 的符号。 SymProxy SymbolPath 的设置如下所示：

```
控制台

\\MainOffice\Symbols;https://msdl.microsoft.com/download/symbols;
https://symbols.contoso.com/symbols
```

首先使用) 符号路径生成的 (查询 Main Office 符号文件共享：

```
dbgcmd

SRV*D:\SymStore\Symbols*\\MainOffice\Symbols
```

如果未找到符号文件，将使用) 符号路径生成的 (查询 Microsoft 符号存储：

```
dbgcmd

SRV*D:\SymStore\Symbols*https://msdl.microsoft.com/download/symbols
```

如果仍未找到该文件，将使用生成的 (查询 Contoso 符号存储 `(https://symbols.contoso.com/symbols)` ，) 符号路径：

dbgcmd

```
SRV*D:\SymStore\Symbols*https://symbols.contoso.com/symbols
```

本节包括：

安装 SymProxy

配置注册表

选择网络安全凭据

为 SymProxy 配置 IIS

设置排除列表

处理不可用的符号存储

处理文件指针

缓存获取的符号文件

安装 SymProxy

项目 • 2023/06/15

安装任务摘要

下面总结了安装和配置 SymProxy 的任务。

- 需要将 SymProxy 文件复制到 IIS 的 %WINDIR%\system32\inetsrv 文件夹。 下面讨论了此任务。
- 需要为 SymProxy 配置注册表。 有关详细信息，请参阅 配置注册表。
- 清单需要注册为性能计数器和 ETW 事件，并且需要配置事件日志。
- 需要配置 IIS。 有关详细信息，请参阅 选择网络安全凭据 和 为 SymProxy 配置 IIS。

可以使用 Install.cmd 文件自动执行这些步骤。 有关详细信息，请参阅 SymProxy 自动安装。

将 SymProxy 文件复制到 IIS

SymProxy 文件包含在 Windows 驱动程序工具包的调试器目录中。 例如，这是Windows 10工具包的 64 位文件的位置。 C：\Program Files (x86) \Windows Kits\10\Debuggers\x64\symproxy。

若要在服务器上安装 SymProxy，请将 symproxy.dll、symsrv.dll 和 symproxy.man 复制到 %WINDIR%\system32\inetsrv。

为了防止访问 Microsoft Symbol Store 时可能出现的问题，请创建名为 %WINDIR%\system32\inetsrv\symsrv.yes 的空白文件。 此文件的内容并不重要。 当 symsrv.yes 文件存在时，它会自动接受 Microsoft 公共符号存储的 EULA。

请注意，通常随 IIS 和 Windows 服务器一起安装的证书（如“Baltimore CyberTrust Root”）用于与上游提供程序的 HTTPS/TLS 通信，并且它们需要位于运行 SymProxy 的计算机上的受信任根存储中。 有关排查 SSL 问题的一般信息，请参阅 排查 (服务器证书) SSL 相关问题 。

配置注册表

项目 • 2023/06/15

SymProxy 将其设置存储在此注册表项中。

reg

```
HKLM/Software/Microsoft/Symbol Server Proxy
```

此注册表项控制从中查找要存储在网站中的符号的位置、日志记录级别以及 SymProxy 是否通过与网络的直接连接进行操作。 可以通过运行 SymProxy 注册工具 (Symproxy.reg) Windows 调试工具来创建此密钥。 在命令提示符下键入 **symproxy.reg** ，或在 Windows 资源管理器中双击它。

这会为设置添加以“x”为前缀的条目，以便禁用它们。 若要启用某个设置，请从所需设置的前面删除“x”。

reg

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Symbol Server Proxy]
"Available Settings"="Remove the 'x' prefix to use the setting"
"xMissAgeTimeout"=dword:00015180
"xMissAgeCheck"=dword:00000e10
"xMissTimeout"=dword:00000e10
"xNoCache"=dword:00000001
"xNoFilePointers"=dword:00000001
"xNoInternetProxy"=dword:00000001
"xNoLongerIndexedAuthoritive"=dword:00000001
"xNoUncompress"=dword:00000001
"xRequestTimeout"=dword:0000019
"xRetryAppHang"=dword:0000002
"xUriFilter"=dword:00000FF
"xUriTiers"=dword:0000001
```

symproxy.reg 注册表文件假定虚拟目录名称为 Symbols，并将符号路径配置为使用 Microsoft 公共符号服务器。

reg

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Symbol Server Proxy\Web Directories]

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Symbol Server Proxy\Web
Directories\Symbols]
"SymbolPath"="https://msdl.microsoft.com/download/symbols"
```

symproxy.reg 中的事件日志记录条目在本主题的事件日志部分进行了介绍。

```reg
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\EventLog\Application\Microsoft-Windows-SymProxy]
"ProviderGuid"="{0876099c-a903-47ff-af14-52035bb479ef}"
"EventMessageFile"=hex(2):25,00,53,00,79,00,73,00,74,00,65,00,6d,00,52,00,6f,\

00,6f,00,74,00,25,00,5c,00,73,00,79,00,73,00,74,00,65,00,6d,00,33,00,32,00,\

5c,00,69,00,6e,00,65,00,74,00,73,00,72,00,76,00,5c,00,53,00,79,00,6d,00,50,\
  00,72,00,6f,00,78,00,79,00,2e,00,64,00,6c,00,6c,00,00,00
"TypesSupported"=dword:00000007
```

本主题讨论了 symproxy.reg 中的 Web 目录条目。

Web 目录

对于用作符号存储的 IIS 中生成的每个虚拟目录，必须在以下注册表项的 **Web 目录** 子项下设置注册表项。

```reg
HKLM/Software/Microsoft/Symbol Server Proxy
```

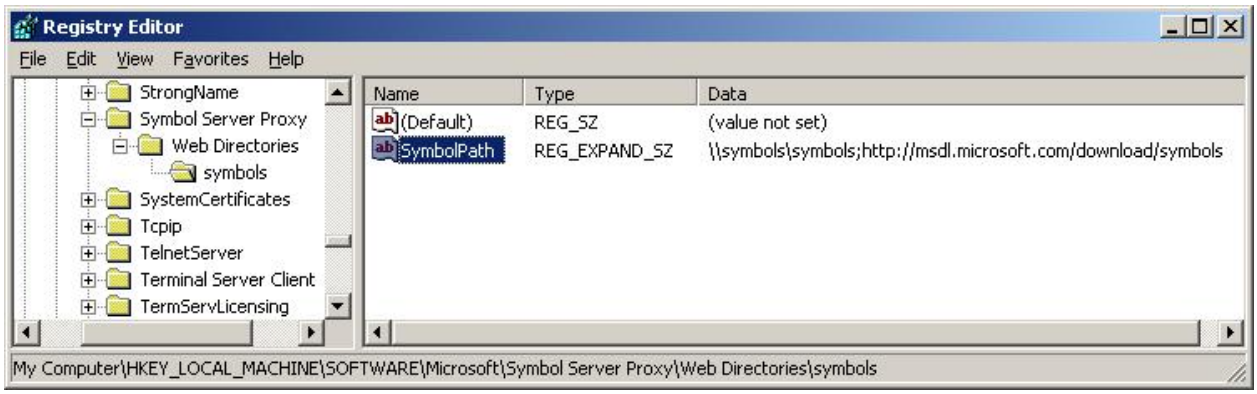
编辑符号存储虚拟目录的注册表项

- 编辑 **SymbolPath** 的内容，以包含 SymProxy 符号存储使用的所有符号存储。 如果正在使用多个符号存储，请使用分号分隔它们。 每个值最多支持 10 个存储。 HTTP 路径必须包含 **https:// 前缀**，UNC 路径必须包含 前缀 \\ 。

例如，如果其中一个虚拟目录称为 Symbols，并且它访问的符号存储位于 UNC 存储 \\symbols\symbols 和 HTTP 存储 https://msdl.microsoft.com/download/symbols 中，则创建以下注册表项。

```reg
HKLM/Software/Microsoft/Symbol Server Proxy/Web Directories/Symbols
```

创建此键后，将其 **SymbolPath** 编辑为 \\symbols\symbols;https://msdl.microsoft.com/download/symbols 。 可以在注册表编辑器的以下屏幕截图中看到这一点。



在此示例中，SymProxy 首先搜索 \\symbols\symbols 中的符号。 如果找不到文件，则将使用 Microsoft 符号存储。

- 在 Web 目录下与虚拟目录名称匹配的每个键中，都需要创建一个名为 SymbolPath 的REG_SZ。 值包含将用于填充 SymProxy 符号存储的所有上游符号存储。
- 最多支持 10 个条目。
- 用分号分隔条目。
- UNC 路径需要包含“\\”前缀
- HTTP 路径需要包含“https://”前缀
- 将值从“最低”到“最贵”进行排序。
  - 你需要在计算中平衡使用性能目标与服务器和数据通信成本。
  - 通常，将本地 SMB/HTTP 服务器置于 Internet HTTP 服务器之前。

SymProxy 性能计数器

SymProxy 可以通过名为 SymProxy 的提供程序发出性能计数器。

若要启用性能计数器支持，请在管理员命令窗口中注册 symproxy 清单文件：

控制台

```
C:\> lodctr.exe /m:%WINDIR%\system32\inetsrv\symproxy.man
```

若要禁用性能计数器支持，请注销清单：

控制台

```
C:\> unlodctr.exe /m:%WINDIR%\system32\inetsrv\symproxy.man
```

适用于 Windows 的 SymProxy 事件跟踪

SymProxy 可以通过名为 Microsoft-Windows-SymProxy 的提供程序创建 ETW 事件。

控制台

```
C:\> logman query providers | findstr SymProxy
Microsoft-Windows-SymProxy {0876099C-A903-47FF-AF14-52035BB479EF}
```

若要启用 ETW 支持，请注册清单文件：

控制台

```
C:\> wevtutil.exe install-manifest %WINDIR%\system32\inetsrv\symproxy.man
```

若要禁用 ETW 支持，请注销清单文件：

控制台

```
C:\> wevtutil.exe uninstall-manifest %WINDIR%\system32\inetsrv\symproxy.man
```

事件日志

如果配置了 ETW，事件将作为事件记录在事件日志中的*"应用程序和服务日志\Microsoft\Windows\SymProxy"*下的*"操作和分析"*通道中。

若要正确查看事件日志条目的消息，需要将 symproxy.reg 文件的事件日志区域添加到注册表：

reg

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\EventLog\Application\M
icrosoft-Windows-SymProxy]
"ProviderGuid"="{0876099c-a903-47ff-af14-52035bb479ef}"
"EventMessageFile"=hex(2):25,00,53,00,79,00,73,00,74,00,65,00,6d,00,52,00,6f
,\

00,6f,00,74,00,25,00,5c,00,73,00,79,00,73,00,74,00,65,00,6d,00,33,00,32,00,\

5c,00,69,00,6e,00,65,00,74,00,73,00,72,00,76,00,5c,00,53,00,79,00,6d,00,50,\
  00,72,00,6f,00,78,00,79,00,2e,00,64,00,6c,00,6c,00,00,00
"TypesSupported"=dword:00000007
```

SymProxy 事件

SymProxy 记录以下事件：

事件 ID	说明	Channel
1	ISAPI 筛选器的开头	管理员
2	停止 ISAPI 筛选器	管理员
3	ISAPI 筛选器的配置	管理员
4	缺少缓存统计信息	管理员
10	URL 请求 - 本地缓存命中	可运行
11	URL 请求 - 本地缓存未命中	可运行
20	通过 SymSrv 下载符号	可运行
30	缺少关键符号	管理员
31	缺少关键图像	管理员
40	Symsrv - 找不到路径	管理员
41	Symsrv - 找不到文件	管理员
42	SymSrv - 拒绝访问	管理员
43	Symsrv - 路径太长	管理员
49	SymSrv - 错误代码	管理员
90	锁争用	可运行
100	常规关键消息	分析
101	常规错误消息	分析
102	常规警告消息	分析
103	常规信息性消息	分析
104	常规分析消息	分析
105	常规调试消息	调试

符号服务器代理配置

SymProxy 将其配置设置存储在以下注册表项区域中：

```
reg
```

```
HKLM/Software/Microsoft/Symbol Server Proxy
```

从此位置，SymProxy 获取其全局设置和上游符号存储的符号路径。

可以通过在前面所述的自定义的 symproxy.reg 文件中合并来创建此密钥。

符号服务器代理密钥

符号服务器代理注册表项支持以下全局设置 (所有REG_DWORD) 。 可以通过回收应用程序池实时应用设置。 将创建新的w3wp.exe进程，并读取新值。 完成对旧w3wp.exe进程的所有挂起请求后，旧w3wp.exe进程将结束。 默认情况下，IIS 每 1，740 分钟 (29 小时) 回收w3wp.exe进程。

REG_DWORD	说明
NoInternetProxy	作为服务运行时，SymSrv.dll使用 WinHTTP 而不是 WinInet 发出 HTTP 请求。 因此，可能需要设置 HTTP 代理设置，以便服务可以访问外部网络资源。 可以使用 netsh 程序执行此操作。 键入“netsh.exe winhttp -？ ” 以获取说明。 默认情况下，SymProxy 使用指定的 HTTP 代理。 如果未配置 HTTP 代理，SymProxy 将使用虚拟代理。 这样就可以安全地访问 Intranet 中的 HTTP 站点。 作为一个副作用，这会阻止 SymProxy 直接连接到不安全的站点。 • 未指定 - (默认) 禁用 - 使用代理 • 0 – 已禁用 • 1+ - 已启用
NoFilePointers	默认情况下，对于不存在的符号，SymProxy 将在本地缓存) (查找请求的文件旁边的 file.ptr 文件。 如果找到，它将返回 file.ptr 文件指定的位置。 仅当 SymStore.exe填充本地缓存时，才需要此功能。 • 未指定 - (默认) 禁用 - 搜索/读取 file.ptr 文件 • 0 – 已禁用 • 1+ - 已启用
NoUncompress	默认情况下，SymProxy 将在将文件返回到调用方之前解压缩下载的符号。 这会减少客户端的 CPU，但会增加 I/O。 • 未指定 - (默认) 禁用 - 发生解压缩 • 0 – 已禁用

	• 1+ - 已启用
NoCache	默认情况下，SymProxy 会将下载的符号缓存到由虚拟目录的路径定义的本地文件系统。 • 未指定 - (默认) 禁用 - 发生缓存 • 0 – 已禁用 • 1+ - 已启用
MissTimeout	超时期限（以秒为单位），其中缺失符号报告为缺失，而无需重新查询上游符号服务器。 未命中与基于 UTC 的时间相关联。 对文件的后续请求将立即拒绝 N 秒。 N 秒后对文件的第一个请求会导致重新查询上游符号存储。 成功后，将返回符号文件并删除未命中。 失败时，未命中数将移动到当前 (UTC) 开始新的超时期限的时间。 使用"未命中缓存"性能计数器监视未命中数。 • 未指定 - (默认) 300 秒/5 分钟 • 0 – 功能已禁用 • N – 超时持续 N 秒
MissAgeCheck	错过年龄检查之间的时间段。 扫描 Miss 缓存并删除早于 MissAgeTimeout 秒的记录。 当前统计信息使用事件 ID 4 保存到事件日志中。 • 未指定 - (默认) 3600 秒/1 小时 • 0 – 功能已禁用 • N – 检查之间的时间段（以 N 秒为单位）
MissAgeTimeout	清除每个 Miss 缓存条目的超时。 由于在整个时间段内对条目的任何请求均无效，因此会清除该条目。 • 未指定 - (默认) 86400 秒/1 天 • 0 – 功能已禁用 • N – 条目超时（以 N 秒为单位）
NoLongerIndexedAuthoritive	启用后，NoLongerIndexed 的 file.ptr 响应将被视为跨所有符号存储的权威。

	使用此方法可避免 (对不为文件编制索引的服务器调用不必要的) 调用。 • 未指定 - (默认) 禁用 • 0 – 已禁用 • 1+ - 已启用
RetryAppHang	启用对上游 HTTP 符号存储的重试。 这等效于 SymSrv SSRVOPT_RETRY_APP_HANG (0x80000000) 选项。 通过上游 HTTP 符号存储的“Symbol-Agent-Status” HTTP 响应标头收到 0x80070512/HRESULT_FROM_WIN32 (ERROR_APP_HANG) 错误代码时，套接字将保持打开状态，GET 将重复到“N”次。 SymProxy 合并对同一 URI 的多个请求。 当任何挂起的请求达到 25 秒时，SymProxy 将通过“Symbol-Agent-Status”HTTP 响应标头将0x80070512返回到调用方，但在后台继续操作。 客户端应在 SymSrv 中启用 SSRVOPT_RETRY_APP_HANG选项，以便支持 (额外的) 长请求 - 实质上是链接重试上游。 SymProxy 将响应超时默认为 25 秒，以便在套接字关闭之前做出响应， (默认) SymSrv 用于 HTTP 符号存储的 30 秒超时。 更高版本的 SymSrv 通过 mSec) 中的“Symbol-Agent-Receive-Timeout”HTTP 请求标头 (发送其配置的 HTTP 超时值。 如果提供此 HTTP 请求标头，则使用此值而不是 25 秒默认值。 • 未指定 - (默认) 禁用 • 0 – 已禁用 • N – 重试尝试
UriFilter	启用 URI 筛选。 这等效于 SymSrv SSRVOPT_URI_FILTER (0x20000000) 选项。 URI 筛选可减少**对所有**上游符号存储的请求。 这些值是位掩码。 HTTP - SSRVURI_HTTP_MASK (0x0F) • 0x01 - SSRVURI_HTTP_NORMAL - 例如 http://symbols/.../foo.pdb • 0x02 - SSRVURI_HTTP_COMPRESSED - 例如 http://symbols/.../foo.pd_

	• 0x04 - SSRVURI_HTTP_FILEPTR - 例如 http://symbols/.../file.ptr UNC - SSRVURI_UNC_MASK (0xF0) • 0x10 - SSRVURI_UNC_NORMAL - 例如 \\MyServer\Symbols\...\foo.pdb • 0x20 - SSRVURI_UNC_COMPRESSED - 例如 \\MyServer\Symbols\...\foo.pd_ • 0x40 - SSRVURI_UNC_FILEPTR - 例如 \\MyServer\Symbols\...\file.ptr
UriTiers	启用 URI 层。 这等效于 SymSrv SSRVOPT_URI_TIERS (0x40000000) 选项。 URI 层强制**所有**上游符号存储使用的层架构。 如果未设置，则需要其他请求来确定架构。 根目录中存在“index2.txt”表示有 2 层布局。 1 层存储的格式为：/widget.dll/<index>/widget.dll|widget.dl_|file.ptr 2 层存储的格式为：**/wi**/widget.dll/<index>/widget.dll|widget.dl_|file.ptr • 未指定 - (默认) 禁用 • 0 – 已禁用 • 1 - 1 层符号存储 • 2 – 2 层符号存储

访问外部网络资源

当 SymSrv 与 SymProxy 一起使用时，它将作为服务运行，并使用 WinHTTP API 通过 HTTP 连接访问符号。 这与用于此目的的 WinInet 的通常行为不同。

因此，可能需要设置 HTTP 代理设置，以便此服务可以访问外部网络资源。 使用以下方法之一配置这些设置：

- 使用 Netsh 工具 (netsh.exe) 。 有关说明，请在命令提示符窗口中键入以下内容：

```
控制台

netsh winhttp -?
```

SymProxy 的默认行为是使用 ProxyCfg 或 Netsh 指定的任何 HTTP 代理。 如果未配置 HTTP 代理，SymProxy 将使用虚拟代理来允许访问 Intranet 中的安全 HTTP 站点。 作为

一个副作用，此方法可防止 SymProxy 使用与外部 Internet 的直接连接。 如果希望允许 SymProxy 使用与 Internet 的直接连接进行操作，请在注册表的**符号服务器代理服务器**密钥中创建名为 **NoInternetProxy** 的REG_DWORD值。 将 **NoInternetProxy** 的值设置为 1，并验证 ProxyCfg 是否未指示 HTTP 代理。

选择网络安全凭据

项目 • 2024/03/04

符号代理服务器必须从安全上下文中运行，并具有访问计划使用的符号存储的适当权限。 如果从外部 Web 存储（例如，https://msdl.microsoft.com/download/symbols ）获取符号，则符号代理服务器必须从任何防火墙外部访问 Web。 如果从网络上的其他计算机获取文件，符号代理服务器必须具有从这些位置读取文件的适当权限。 两种可能的选择是将符号代理服务器设置为**网络服务**帐户进行身份验证，或者创建一个与其他用户帐户一起在 Active Directory 域服务中管理的用户帐户。

注意 最好将此帐户的权限限制为仅读取文件并将其复制到 c:\symstore 所需的权限。 此限制可防止访问 HTTP 存储的客户端损坏系统。

注意 确保此处提供的选项在你的环境中有意义。 不同的组织有不同的安全需求和要求。 修改此处概述的过程以支持组织的安全要求。

作为网络服务进行身份验证

网络服务帐户内置于 Windows 中，因此不需要创建新帐户的额外步骤。 在本示例中，我们将在名为 *corp* 的域上配置符号代理服务器的计算机命名为 *SymMachineName*。

必须将外部符号存储或 Internet 代理配置为允许此计算机的**网络服务**帐户（计算机帐户）成功进行身份验证。 可通过两种方式实现此目的：

- 允许访问外部存储或 Internet 代理上的**经过身份验证的用户**组。
- 允许访问计算机帐户 *corp\SymMachineName$*。 此选项更安全，因为它仅限制对符号代理服务器的“网络服务”帐户的访问。

以域用户身份进行身份验证

在此示例中，我们假定用户帐户在名为 *corp* 的域中命名为 *SymProxyUser*。

将用户帐户添加到 IIS_USRS 组

1. 从**管理工具**中，打开**计算机管理**。
2. 展开“本地用户和组”。
3. 单击“组”。
4. 在中间窗格中双击 **IIS_USRS**，然后选择**属性**。

5. 在**成员**部分下，单击**添加**。
6. 在标记为**输入要选择的对象名称**的窗格中键入 *corp\SymProxyUser*。
7. 若要退出**选择用户、计算机或组**对话框，请单击**确定**。
8. 若要退出 **IIS_USRS 属性**，请单击**确定**。
9. 关闭“计算机管理”控制台。

设置 IIS 以使用帐户

1. 从**管理工具**中打开 **Internet Information Services (IIS) 管理器**。
2. 展开**网站**。
3. 右键单击**默认网站**，然后选择**属性**。
4. 单击“目录安全性”选项卡。
5. 在**身份验证和访问控制**部分，单击**编辑...**。
6. 请确保选中*启用匿名访问*。
7. 输入有权访问远程符号服务器存储的帐户的凭据 (“corp\SymProxyUser”)，然后单击**确定**。
8. 在系统询问时重新输入密码，然后单击**确定**。
9. 若要退出**默认网站属性**，请单击**确定**。
10. 你可能会看到*继承替代*对话框。 如果是，请选择要将其应用于哪个虚拟目录。

使用 IIS_WPG 组作为域用户进行身份验证

在本例中，在名为 *corp* 的域上，用户帐户名为 *SymProxyUser*。 若要对此用户帐户进行身份验证，必须将其添加到 **IIS_WPG** 组中。

将用户帐户添加到 IIS_WPG 组

1. 从**管理工具**中，打开**计算机管理**。
2. 展开“本地用户和组”。
3. 单击“组”。
4. 在右侧窗格中，双击 **IIS_WPG**。

5. 单击“添加” 。

6. 在标记为**输入要选择的对象名称**的窗格中键入 *corp\SymProxyUser*。

7. 若要退出**选择用户、计算机或组**对话框，请单击**确定**。

8. 若要退出 **IIS_WPG 属性**，请单击**确定**。

9. 关闭“计算机管理”控制台。

为 SymProxy 配置 IIS

项目 • 2023/06/15

必须将 Internet Information Services (IIS) 配置为使用 SymProxy 作为 Internet 服务器应用程序编程接口 (ISAPI) 筛选器。 此外，必须设置权限，以便 IIS 可以获取符号。

有关自动执行此过程的信息和设置摘要，请参阅 SymProxy 自动安装。

确认示例安全设置适合你的环境，并修改以遵守特定于组织的任何其他安全要求。

配置选项将因正在运行的特定 IIS 版本而异。 有关 IIS 的详细信息，请参阅 IIS Web 服务器概述。

配置应用程序池

1. 打开“Internet Information Services (IIS)管理器” 。
2. 展开左侧具有计算机名称的条目，并找到 “**应用程序池**”。
3. 右键单击“ **应用程序池** ”，然后选择“ **添加应用程序池**”。
4. 对于 “**名称**” ，请键入 *SymProxy 应用池*。
5. 在 “**.Net CLR 版本** ”下，选择“ *无托管代码*”
6. 单击“ **确定** ”创建应用程序池。
7. 接下来，右键单击新应用程序池的条目，然后选择 “**高级设置...**”。
8. 在 “**进程模型**”下，你将看到 “**标识**”。 单击右侧标记为“...”的按钮。
    a. 如果要作为网络服务进行身份验证，请选择“**应用程序池标识**”**的**“**内置帐户**”，然后选择“**网络服务**”，然后单击“**确定**”。
    b. 如果要以域用户身份进行身份验证，请选择“ **自定义帐户** ”，然后单击“ **设置** ”按钮。 键入有权访问远程符号服务器存储的帐户的凭据 (例如 *corp\SymProxyUser*) ，然后单击“ **确定**”。
9. 单击“ **确定** ”退出 “**应用程序池标识** ”对话框。
10. 单击“ **确定** ”退出 “**高级设置**” 对话框。

示例虚拟目录配置

1. 展开 “**站点**”。
2. 右键单击“**默认网站**”，然后选择“**添加虚拟目录**”

3. 使用 **符号等名称** ，并将其映射到所选位置。

4. 右键单击已创建的 **Symbols** 虚拟目录，然后选择 “**添加应用程序**”。

5. 在 “**应用程序池** ”下拉菜单中，选择“ **SymProxy 应用池** ”，然后单击“ **确定**”。

配置 ISAPI 筛选器

1. 确认 ISAPI 选项已安装在 IIS 中。

2. 单击 “**默认网站**”。

3. 双击“ **ISAPI 筛选器**”。

4. 右键单击“ **名称** ”列下的中心窗格，然后选择“单击 **添加**”。

5. 对于 “**筛选器名称**”， 请键入 **SymProxy** 或其他一些有意义的名称。

6. 对于 **可执行** 类型 ， *c:\windows\system32\inetsrv\symproxy.dll*。

7. 若要退出“ **筛选属性** ”对话框，请单击“ **确定**”。

8. 若要退出 “**默认网站属性**”， 请单击“ **确定**”。

配置 MIME 类型

下载内容的 MIME 类型需要设置为 application/octet-stream，以允许 IIS 传递所有符号文件。

1. 右键单击 “**符号**” 虚拟目录。

2. 单击“ **MIME 类型**”。

3. 单击“添加”。

4. 对于 “**扩展**”，请键入 。*

5. 对于 **MIME 类型**，请键入 **application/octet-stream**。

6. 若要退出 “**MIME 类型** ”对话框，请单击“ **确定**”。

使用 web.config 配置 MIME 类型

可以编辑web.config文件，为符号配置 MIME 类型。 此方法清除继承的 MIME 类型，并添加一个全部捕获的野生卡 * MIME 类型。 在某些 IIS 配置中继承 MIME 类型时，可能需要此方法。

1. 编辑web.config文件，如下所示。

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <directoryBrowse enabled="true" />
        <staticContent>
            <clear />
            <mimeMap fileExtension=".*"
mimeType="application/octet-stream" />
        </staticContent>
    </system.webServer>
</configuration>
```

其他配置

所需的步骤是 IIS 符号服务器和符号配置的一部分。 有关其他设置注意事项的信息，请参阅这些主题。

HTTP 符号存储

缓存获取的符号文件

SymProxy 自动安装

设置排除列表

项目 • 2023/06/15

在某些环境中，你可能会发现自己调试的系统加载了大量模块，而你无法获取符号。 如果你有由第三方供应商调用的代码，则通常就是这种情况。 这可能会导致大量尝试查找符号失败，这非常耗时，并会堵塞网络资源。 若要缓解这种情况，可以使用 *排除列表* 来指定应从搜索中排除的符号。 此功能存在于客户端调试器中，但你也可以将 SymProxy 筛选器配置为使用其自己的排除列表，并防止此类网络活动最有可能占用资源。

排除列表由要阻止其处理的文件的名称组成。 文件名可以包含通配符。 例如：

控制台

```
dbghelp.pdb
symsrv.*
mso*
```

可以通过两种方式实现列表。 第一个位于.ini文件中，%WINDIR%\system32\inetsrv\Symsrv.ini。 名为“排除”的部分应包含列表：

控制台

```
[exclusions]
dbghelp.pdb
symsrv.*
mso*
```

或者，可以将排除项存储在注册表中。 创建名为 的密钥

text

```
HKLM\Software\Microsoft\Symbol Server\Exclusions
```

将文件名列表存储为此键中 (REG_SZ) 字符串值。 字符串值的名称充当要排除的文件名。 字符串值的内容可用作描述文件被排除原因的注释。

SymProxy 每半小时从排除列表中读取一次，因此无需重启 Web 服务即可看到更改生效。 将文件添加到注册表中的列表或.ini文件，然后稍等片刻，以使用排除项。

注意 SymProxy 不支持同时使用Symsrv.ini和注册表。 如果.ini文件存在，则使用该文件。 否则，将检查注册表。

处理不可用的符号存储

项目 • 2023/06/15

如果 SymSrv 配置为从中获取文件的符号存储之一处于关闭状态或不可用，则每个文件请求的客户端可能会等待很长时间。 从 SymProxy 调用 SymSrv 时，可以通过设置 SymSrv 停止尝试访问有问题的存储来避免大部分等待。 使用此功能时，SymSrv 在设置的时间间隔内从同一存储区遇到指定超时数后，将停止尝试在一段设定的时间段内使用该存储。 这些变量的值可以通过 .ini 文件或注册表进行控制。

使用 .ini 文件控制符号存储访问

1. 在 %WINDIR%\system32\inetsrv\Symsrv.ini中创建名为 **timeouts 的部分。**
2. 将值 **触发器**、 **计数**和 **中断** 添加到此部分。

触发器指示超时watch的时间（以分钟为单位）。 **Count** 指示在 **触发** 期间要查找的超时数。 “**停电** ”表示达到阈值后禁用存储的时间长度（以分钟为单位）。

例如，建议使用以下设置：

```ini
[timeouts]
trigger=10
count=5
blackout=15
```

在此示例中，如果在 10 分钟内遇到 5 次超时，则会关闭存储访问。 完成 15 分钟的停电后，将重新激活存储。

使用注册表控制符号存储访问

1. 创建名为 的密钥

   ```text
   HKLM\ Software\Microsoft\Symbol Server\Timeouts
   ```

2. 向此键添加三个REG_DWORD值 **触发器**、 **计数**和 **停电** 。 像在 .ini 文件中一样设置这些值。

无论是使用注册表还是.ini文件，如果任何触发器、计数或停电值设置为 0，或者如果任何键或值不存在，则禁用此功能。

SymSrv 的此功能目前仅在作为服务运行时可用。 这意味着此功能的唯一实际应用是从 SymProxy 调用它时。

处理文件指针

项目 • 2023/06/15

UNC 符号存储支持将实际文件放置在单独的位置，客户端代码通过文件指针查找文件的位置。 这些指针使用带 **/p** 选项的 SymStore 在符号存储中生成。 仅当文件指针指向客户端可直接访问的 UNC 位置时，其他基于 HTTP 的符号存储才支持此处理。 将 SymProxy 加载到 Web 服务器时，文件指针处理会自动增强。 客户端不再需要能够直接访问目标文件，因为 SymProxy 通过 HTTP 接口为这些文件提供服务。

由于此功能是自动应用的，因此存在一个选项来关闭此功能，以防必须使用代理为某些文件提供服务，并为其他文件提供常规文件指针实现。 为此，请在 **HKLM\Software\Microsoft\Symbol Server** 中创建名为“NoFilePointerHandler”的 REG_DWORD。 将此值设置为 1 (或 0 以外的任何值，) 关闭 SymProxy 中的内部文件指针处理程序。

缓存获取的符号文件

项目 • 2024/03/08

通常，SymProxy 会将其获取的文件缓存在 Internet 信息服务 (IIS) 中指定为相关网站虚拟根的目录中。 然后，IIS 使该文件可用于客户端调试器。 由于调试器无法直接从 HTTP 打开文件，因此它将文件复制到由符号路径指定的本地缓存：

```
text

srv*c:\localcache*https://server/symbols
```

在本例中，客户端调试器将文件复制到 c:\localcache。 在这种情况下，文件会复制两次：一次由 SymProxy 复制到网站的虚拟根目录，另一次由调试器复制到其本地缓存。

可以避免第二次复制操作并加快处理速度。 要执行此操作，必须首先将网站的虚拟根目录共享为调试器可以访问的 UNC 路径。 例如，此路径名为 \\server\symbols。 然后，必须删除 MIME 类型的 IIS 配置：

删除 MIME 类型的 IIS 配置

1. 从**管理工具**中打开 **Internet Information Services (IIS) 管理器。**
2. 展开**网站。**
3. 右键单击**默认网站。**
4. 右键单击**符号**虚拟目录，然后选择**属性。**
5. 单击 **HTTP 标头**选项卡。
6. 单击 **MIME 类型。**
7. 选择标记为**已注册 MIME 类型**的列表框中的所有类型。
8. 单击“移除”。
9. 要退出 **MIME 类型**对话框，请单击**确定。**
10. 若要退出**符号属性**，请单击**确定。**

这导致 IIS 将**找不到的文件**返回到网站上所有事务的调试客户端。 但是，这并不阻止 SymProxy 使用该文件填充虚拟根。

删除 MIME 类型的 IIS 配置后，使用以下命令将调试器客户端配置为首先在 HTTP 存储和映射到存储的虚拟根的共享中查找符号：

text

```
srv**https://server/symbols;srv*\\server\symbols
```

在前面的示例中，符号路径的第一个元素（srv**https://server/symbols) 表示要从 HTTP 存储中获取文件，并将其作为本地缓存复制到默认符号存储区。 指定的缓存并不重要，因为从未从 HTTP 存储收到任何文件。 失败后，它会尝试从存储的虚拟根的实际位置 (srv*\\server\symbols) 获取文件。 此尝试成功，因为文件被复制到该位置，这是前一个路径处理的副作用。

SymProxy 自动安装

项目 • 2023/04/14

这些步骤与下面的 Install.cmd 脚本一起有助于自动将 SymProxy 安装到默认 IIS 安装。 可能需要根据环境的特定需求调整这些步骤。

1. 创建 D：\SymStore\Symbols 文件夹。
    - 向所有人授予读取权限
    - 向 SymProxy 应用池用户帐户授予读取/写入权限 (域\用户)
2. 将 D：\SymStore\Symbols 共享为符号。
    - 向所有人授予读取 (或更具体的)
3. (（可选）) 在 D：\SymStore\Symbols 中创建名为 index2.txt 的空文件。
4. (（可选）) 创建名为 %WINDIR%\system32\inetsrv\symsrv.yes 的空文件。 这接受 Microsoft 公共符号存储的 EULA。
5. 确定 Install.cmd 的参数并运行它。
6. 使用创建的服务器名称配置客户端符号路径。

    dbgcmd

    ```
    SRV*\\MachineName\Symbols*https://MachineName/Symbols
    ```

Install.cmd 脚本需要 3 个参数：

- 虚拟目录路径 (例如 D：\SymStore\Symbols)
- 应用程序池) 的用户名 (
- 应用程序池) 的密码 (

若要清除 MIME 类型继承，需要一个 XML 文件来驱动关联的 AppCmd.exe 命令。 将下面所示的 staticContentClear.xml 文件放在 Install.cmd 脚本所在的文件夹中以实现此结果。

Install.Cmd 参数用法示例：

控制台

```
Install.cmd D:\SymStore\Symbols CONTOSO\SymProxyService Pa$$word
```

Install.cmd

bat

```
@echo off

SET VirDirectory=%1
SET UserName=%2
SET Password=%3

::
::  SymProxy dll installation.
::

copy symproxy.dll %windir%\system32\inetsrv
copy symproxy.man %windir%\system32\inetsrv
copy symsrv.dll %windir%\system32\inetsrv

lodctr.exe /m:%windir%\system32\inetsrv\symproxy.man
wevtutil.exe install-manifest %windir%\System32\inetsrv\symproxy.man
regedit.exe /s symproxy.reg

::
::  Web server Configuraiton
::

IF not exist %VirDirectory% mkdir %VirDirectory%

rem Make the 'Default Web Site'
%windir%\system32\inetsrv\appcmd.exe add site -site.name:"Default Web Site"
-bindings:"http/*:80:" -physicalPath:C:\inetpub\wwwroot

rem Enabled Directory Browsing on the 'Default Web Site'
%windir%\system32\inetsrv\appcmd.exe set config "Default Web Site" -
section:system.webServer/directoryBrowse /enabled:"True"

rem Make the 'SymProxy App Pool'
%windir%\system32\inetsrv\appcmd.exe add apppool -
apppool.name:SymProxyAppPool -managedRuntimeVersion:
%windir%\system32\inetsrv\appcmd.exe set apppool -
apppool.name:SymProxyAppPool -processModel.identityType:SpecificUser -
processModel.userName:%UserName% -processModel.password:%Password%

rem Make the 'Symbols' Virtual Directory and assign the 'SymProxy App Pool'
%windir%\system32\inetsrv\appcmd.exe add app -site.name:"Default Web Site" -
path:/Symbols -physicalpath:%VirDirectory%
%windir%\system32\inetsrv\appcmd.exe set app -app.name:"Default Web
Site/Symbols" -applicationPool:SymProxyAppPool

rem Disable 'web.config' for folders under virtual directories in the
'Default Web Site'
%windir%\system32\inetsrv\appcmd.exe set config -
section:system.applicationHost/sites "/[name='Default Web
```

```
Site'].virtualDirectoryDefaults.allowSubDirConfig:false

rem Add the 'SymProxy ISAPI Filter'
%windir%\system32\inetsrv\appcmd.exe set config -
section:system.webServer/isapiFilters /+"
[name='SymProxy',path='%windir%\system32\inetsrv\SymProxy.dll',enabled='True
']

rem Clear the MIME Types on the 'Default Web Site'
%windir%\system32\inetsrv\appcmd.exe set config -in "Default Web Site" <
staticContentClear.xml

rem Add * to the MIME Types of the 'Default Web Site'
%windir%\system32\inetsrv\appcmd.exe set config "Default Web Site" -
section:staticContent /+"[fileExtension='.*',mimeType='application/octet-
stream']"
```

staticContentClear.xml

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<appcmd>
    <CONFIG CONFIG.SECTION="system.webServer/staticContent"
path="MACHINE/WEBROOT/APPHOST">
        <system.webServer-staticContent>
            <clear />
        </system.webServer-staticContent>
    </CONFIG>
</appcmd>
```

测试 SymProxy 安装

系统现在应该已准备好获取和提供文件。 若要测试它，请首先通过运行iisreset.exe重启 IISAdmin 服务。 这将使用当前 IIS 和 SymProxy 配置重新加载 ISAPI 筛选器。

将调试器配置为使用此符号路径：

dbgcmd

```
srv*\\MachineName\Symbols*https://MachineName/Symbols
```

如果启用 *MissTimeout* (默认情况下将其设置为 300 秒)， 则运行 .reload /f 命令两次应该会导致第二次执行速度要快得多。

若要查看被引用的 PDB 的位置，请使用 lm (list modules) 命令。 PDB 的路径应全部以 \\MachineName\Symbols 开头。

如果在网站上启用了目录浏览，请浏览到 https://MachineName/Symbols 以查看缓存的文件。

打开性能监视器并查看符号代理计数器。

打开事件查看器并查看 Microsoft\Windows\SymProxy 事件。

另请参阅

安装 SymProxy

SymStore

项目 • 2023/04/14

SymStore (symstore.exe) 工具用于创建符号存储。 它包含在 Windows 调试工具中。 有关详细信息，请参阅 Debugging Tools for Windows （Windows 调试工具）。

SymStore 以一种格式存储符号，使调试器能够根据 .dbg 或可执行文件) 的图像 (的时间戳和大小，或 .pdb 文件) 的签名和年龄 (来查找符号。 符号存储比传统符号存储格式的优势在于，所有符号都可以存储在同一服务器上，并由调试器检索，而无需事先知道哪个产品包含相应的符号。

请注意，多个版本的 .pdb 符号文件 (例如，) 公共和专用版本不能存储在同一服务器上，因为它们都包含相同的签名和期限。

本节包括：

SymStore 事务

文件系统参考和符号文件

符号存储格式

SymStore 事务

项目 • 2023/06/15

每次对 SymStore 的调用都记录为一个事务。 有两种类型的事务：添加和删除。

创建符号存储时，将在服务器的根目录下创建一个名为“000admin”的目录。 000admin 目录包含每个事务的一个文件，以及server.txt和history.txt日志文件。 server.txt 文件包含服务器上当前所有事务的列表。 history.txt文件包含所有事务的按时间顺序排列的历史记录。

每次 SymStore 存储或删除符号文件时，都会创建一个新的事务编号。 然后，在 000admin 中创建名为此事务编号的文件。 此文件包含在此事务期间已添加到符号存储的所有文件或指针的列表。 如果删除事务，SymStore 将读取其事务文件，以确定应删除的文件和指针。

add 和 **del** 选项指定是执行添加或删除事务。 将 **/p** 选项与添加操作一起指定要添加指针; 省略 **/p** 选项将指定要添加实际符号文件。

还可以在两个单独的阶段中创建符号存储。 在第一阶段，将 SymStore 与 **/x** 选项一起使用来创建索引文件。 第二阶段，将 SymStore 与 **/y** 选项配合使用，根据索引文件中的信息创建文件或指针的实际存储。

由于各种原因，这可以是一种有用的技术。 例如，只要索引文件仍然存在，这样就可以在存储以某种方式丢失时轻松重新创建符号存储。 或者，包含符号文件的计算机与要在其中创建符号存储的计算机的网络连接速度缓慢。 在这种情况下，可以在符号文件所在的计算机上创建索引文件，将索引文件传输到第二台计算机，然后在第二台计算机上创建存储。

有关所有 SymStore 参数的完整列表，请参阅 SymStore Command-Line选项。

注意 SymStore 不支持来自多个用户的并发事务。 建议将一个用户指定为符号存储的“管理员”，并负责所有 **添加** 和 **del** 事务。

事务示例

下面是 SymStore 将 Windows 2000 内部版本 2195 的符号指针添加到 \\MyDir\symsrv 的两个示例：

控制台

```
symstore add /r /p /f \\BuildServer\BuildShare\2195free\symbols\*.* /s
\\MyDir\symsrv /t "Windows 2000" /v "Build 2195 x86 free" /c "Sample add"
```

```
symstore add /r /p /f \\BuildServer\BuildShare\2195free\symbols\*.* /s
\\MyDir\symsrv /t "Windows 2000" /v "Build 2195 x86 checked" /c "Sample add"
```

在以下示例中，SymStore 将 \\largeapp\appserver\bins 中的应用程序项目的实际符号文件添加到 \\MyDir\symsrv：

控制台

```
symstore add /r /f \\largeapp\appserver\bins\*.* /s \\MyDir\symsrv /t "Large
Application" /v "Build 432" /c "Sample add"
```

下面是如何使用索引文件的示例。 首先，SymStore 基于 \\largeapp\appserver\bins\ 中的符号文件集合创建索引文件。 在这种情况下，索引文件放置在第三台计算机 \\hubserver\hubshare 上。 使用 **/g** 选项指定文件前缀“\\largeapp\appserver”将来可能会更改：

控制台

```
symstore add /r /p /g \\largeapp\appserver /f \\largeapp\appserver\bins\*.*
/x \\hubserver\hubshare\myindex.txt
```

现在，假设将所有符号文件移出计算机 \\largeapp\appserver，并将其放在 \\myarchive\appserver 上。 然后，可以从索引文件\\hubserver\hubshare\myindex.txt创建符号存储本身，如下所示：

控制台

```
symstore add /y \\hubserver\hubshare\myindex.txt /g \\myarchive\appserver /s
\\MyDir\symsrv /p /t "Large Application" /v "Build 432" /c "Sample Add from
Index"
```

最后，下面是 SymStore 删除由上一个事务添加的文件的示例。 请参阅下面的“server.txt和history.txt文件”部分，了解如何在这种情况下确定事务 ID (，0000000096) 。

控制台

```
symstore del /i 0000000096 /s \\MyDir\symsrv
```

server.txt和history.txt文件

添加事务时，会向server.txt添加多个信息项，并history.txt将来的查找功能。 下面是添加事务server.txt和history.txt中的行示例：

```
```

```text
0000000096,add,ptr,10/09/99,00:08:32,Windows Vista SP 1,x86 fre 1.156c-RTM-2,Added from \\mybuilds\symbols,
```

这是逗号分隔的行。 这些字段的说明如下：

字段	说明
0000000096	事务 ID 号，由 SymStore 创建。
添加	事务的类型。 此字段可以是 **add** 或 **del**。
ptr	是否添加了文件或指针。 此字段可以是 **file** 或 **ptr**。
10/09/99	事务发生日期。
00:08:32	事务开始的时间。
Windows Vista SP 1	产品。
x86 fre	版本 (可选) 。
添加自	注释 (可选)
未使用	(保留以供以后使用。)

下面是事务文件0000000096的一些示例行。 每行记录目录以及添加到目录的文件或指针的位置。

```text
canon800.dbg\35d9fd51b000,\\mybuilds\symbols\sp4\dll\canon800.dbg
canonlbp.dbg\35d9fd521c000,\\mybuilds\symbols\sp4\dll\canonlbp.dbg
certadm.dbg\352bf2f48000,\\mybuilds\symbols\sp4\dll\certadm.dbg
certcli.dbg\352bf2f1b000,\\mybuilds\symbols\sp4\dll\certcli.dbg
certcrpt.dbg\352bf04911000,\\mybuilds\symbols\sp4\dll\certcrpt.dbg
certenc.dbg\352bf2f7f000,\\mybuilds\symbols\sp4\dll\certenc.dbg
```

如果使用 **del** 事务撤消原始 **添加** 事务，将从server.txt中删除这些行，并将以下行添加到history.txt：

```text
0000000105,del,0000000096
```

删除事务的字段描述如下。

字段	说明
0000000105	事务 ID 号，由 SymStore 创建。
del	事务的类型。 此字段可以是 **add** 或 **del**。
0000000096	已删除的事务。

文件系统参考和符号文件

项目 • 2023/06/15

磁盘上的文件可以具有长文件名和自动生成的缩写 MS-DOS 兼容 8.3 短文件名。 将符号文件添加到符号存储后，如果符号文件包含任何缩写的 MS-DOS 8.3 文件名，则有可能在调试期间无法访问该符号文件中的符号。

工具创建符号文件时，符号文件调试记录中记录的文件名的版本取决于工具及其运行方式。 如果符号文件具有缩写的 MS-DOS 8.3 文件名，而不是记录中嵌入的实际文件名，则调试时加载符号可能会遇到问题，因为缩写的文件名因系统而异。 如果出现此问题，则调试期间可能无法访问这些符号文件的内容。 用户应尽可能避免在创建符号文件时使用缩写文件路径名称。 无意中使用缩写文件名的一些方法是将缩写文件路径名称用于源文件、 *包含* 目录或包含的库文件。

有关详细信息，请参阅 匹配符号名称。

符号存储格式

项目 • 2023/06/14

SymStore 使用文件系统本身作为数据库。 它创建一个大型目录树，其目录名称基于符号文件时间戳、签名、年龄和其他数据等内容。

例如，将多个不同的 acpi.dbgs 添加到服务器后，目录可能如下所示：

控制台

```
Directory of \\mybuilds\symsrv\acpi.dbg
10/06/1999  05:46p      <DIR>          .
10/06/1999  05:46p      <DIR>          ..
10/04/1999  01:54p      <DIR>          37cdb03962040
10/04/1999  01:49p      <DIR>          37cdb04027740
10/04/1999  12:56p      <DIR>          37e3eb1c62060
10/04/1999  12:51p      <DIR>          37e3ebcc27760
10/04/1999  12:45p      <DIR>          37ed151662060
10/04/1999  12:39p      <DIR>          37ed15dd27760
10/04/1999  11:33a      <DIR>          37f03ce962020
10/04/1999  11:21a      <DIR>          37f03cf7277c0
10/06/1999  05:38p      <DIR>          37fa7f00277e0
10/06/1999  05:46p      <DIR>          37fa7f01620a0
```

在此示例中，acpi.dbg 符号文件的查找路径可能如下所示：\\mybuilds\symsrv\acpi.dbg\37cdb03962040。

查找目录中可能存在三个文件：

1. acpi.dbg（如果存储了文件）
2. file.ptr，其中包含实际符号文件的路径（如果存储了指针）
3. refs.ptr，其中包含 acpi.dbg 的所有当前位置的列表，以及当前添加到符号存储中的此时间戳和图像大小

显示 \\mybuilds\symsrv\acpi.dbg\37cdb03962040 的目录列表会提供以下内容：

控制台

```
10/04/1999  01:54p                52 file.ptr
10/04/1999  01:54p                67 refs.ptr
```

文件 file.ptr 包含文本字符串“\\mybuilds\symbols\x86\2128.chk\symbols\sys\acpi.dbg”。由于此目录中没有名为 acpi.dbg 的文件，调试器将尝试在

\\mybuilds\symbols\x86\2128.chk\symbols\sys\acpi.dbg 中查找该文件。

refs.ptr 的内容仅由 SymStore 使用，而不能由调试器使用。 此文件包含此目录中发生的所有事务的记录。 refs.ptr 中的示例行可能是：

text

```
0000000026,ptr,\\mybuilds\symbols\x86\2128.chk\symbols\sys\acpi.dbg
```

这表明已添加指向 \\mybuilds\symbols\x86\2128.chk\symbols\sys\acpi.dbg 的指针，事务为“0000000026”。

某些符号文件在各种产品或内部版本或特定产品中保持不变。 其中一个示例是 Windows 2000 文件 msvcrt.pdb。 \\mybuilds\symsrv\msvcrt.pdb 的目录列表显示，只有两个版本的 msvcrt.pdb 已添加到符号服务器：

控制台

```
Directory of \\mybuilds\symsrv\msvcrt.pdb
10/06/1999  05:37p      <DIR>          .
10/06/1999  05:37p      <DIR>          ..
10/04/1999  11:19a      <DIR>          37a8f40e2
10/06/1999  05:37p      <DIR>          37f2c2272
```

但是，\\mybuilds\symsrv\msvcrt.pdb\37a8f40e2 的目录列表显示 refs.ptr 中有几个指针。

控制台

```
Directory of \\mybuilds\symsrv\msvcrt.pdb\37a8f40e2
10/05/1999  02:50p              54     file.ptr
10/05/1999  02:50p           2,039     refs.ptr
```

\\mybuilds\symsrv\msvcrt.pdb\37a8f40e2\refs.ptr 的内容如下：

text

```
0000000001,ptr,\\mybuilds\symbols\x86\2137\symbols\dll\msvcrt.pdb
0000000002,ptr,\\mybuilds\symbols\x86\2137.chk\symbols\dll\msvcrt.pdb
0000000003,ptr,\\mybuilds\symbols\x86\2138\symbols\dll\msvcrt.pdb
0000000004,ptr,\\mybuilds\symbols\x86\2138.chk\symbols\dll\msvcrt.pdb
0000000005,ptr,\\mybuilds\symbols\x86\2139\symbols\dll\msvcrt.pdb
0000000006,ptr,\\mybuilds\symbols\x86\2139.chk\symbols\dll\msvcrt.pdb
0000000007,ptr,\\mybuilds\symbols\x86\2140\symbols\dll\msvcrt.pdb
0000000008,ptr,\\mybuilds\symbols\x86\2140.chk\symbols\dll\msvcrt.pdb
0000000009,ptr,\\mybuilds\symbols\x86\2136\symbols\dll\msvcrt.pdb
0000000010,ptr,\\mybuilds\symbols\x86\2136.chk\symbols\dll\msvcrt.pdb
```

```
0000000011,ptr,\\mybuilds\symbols\x86\2135\symbols\dll\msvcrt.pdb
0000000012,ptr,\\mybuilds\symbols\x86\2135.chk\symbols\dll\msvcrt.pdb
0000000013,ptr,\\mybuilds\symbols\x86\2134\symbols\dll\msvcrt.pdb
0000000014,ptr,\\mybuilds\symbols\x86\2134.chk\symbols\dll\msvcrt.pdb
0000000015,ptr,\\mybuilds\symbols\x86\2133\symbols\dll\msvcrt.pdb
0000000016,ptr,\\mybuilds\symbols\x86\2133.chk\symbols\dll\msvcrt.pdb
0000000017,ptr,\\mybuilds\symbols\x86\2132\symbols\dll\msvcrt.pdb
0000000018,ptr,\\mybuilds\symbols\x86\2132.chk\symbols\dll\msvcrt.pdb
0000000019,ptr,\\mybuilds\symbols\x86\2131\symbols\dll\msvcrt.pdb
0000000020,ptr,\\mybuilds\symbols\x86\2131.chk\symbols\dll\msvcrt.pdb
0000000021,ptr,\\mybuilds\symbols\x86\2130\symbols\dll\msvcrt.pdb
0000000022,ptr,\\mybuilds\symbols\x86\2130.chk\symbols\dll\msvcrt.pdb
0000000023,ptr,\\mybuilds\symbols\x86\2129\symbols\dll\msvcrt.pdb
0000000024,ptr,\\mybuilds\symbols\x86\2129.chk\symbols\dll\msvcrt.pdb
0000000025,ptr,\\mybuilds\symbols\x86\2128\symbols\dll\msvcrt.pdb
0000000026,ptr,\\mybuilds\symbols\x86\2128.chk\symbols\dll\msvcrt.pdb
0000000027,ptr,\\mybuilds\symbols\x86\2141\symbols\dll\msvcrt.pdb
0000000028,ptr,\\mybuilds\symbols\x86\2141.chk\symbols\dll\msvcrt.pdb
0000000029,ptr,\\mybuilds\symbols\x86\2142\symbols\dll\msvcrt.pdb
0000000030,ptr,\\mybuilds\symbols\x86\2142.chk\symbols\dll\msvcrt.pdb
```

这表明同一 msvcrt.pdb 用于存储在 \\mybuilds\symsrv 上的 Windows 2000 符号的多个内部版本。

下面是包含文件和指针添加混合的目录示例：

控制台

```
Directory of E:\symsrv\dbghelp.dbg\38039ff439000
10/12/1999  01:54p         141,232     dbghelp.dbg
10/13/1999  04:57p              49     file.ptr
10/13/1999  04:57p             306     refs.ptr
```

在本例中，refs.ptr 包含以下内容：

text

```
0000000043,file,e:\binaries\symbols\retail\dll\dbghelp.dbg
0000000044,file,f:\binaries\symbols\retail\dll\dbghelp.dbg
0000000045,file,g:\binaries\symbols\retail\dll\dbghelp.dbg
0000000046,ptr,\\MyDir\bin\symbols\retail\dll\dbghelp.dbg
0000000047,ptr,\\foo2\bin\symbols\retail\dll\dbghelp.dbg
```

因此，事务 43、44 和 45 将同一文件添加到服务器，事务 46 和 47 添加了指针。 如果删除事务 43、44 和 45，则将从目录中删除文件 dbghelp.dbg。 然后，该目录将包含以下内容：

控制台

```
Directory of e:\symsrv\dbghelp.dbg\38039ff439000
10/13/1999  05:01p                    49 file.ptr
10/13/1999  05:01p                   130 refs.ptr
```

现在，file.ptr 包含“\\foo2\bin\symbols\retail\dll\dbghelp.dbg”，refs.ptr 包含

text

```
0000000046,ptr,\\MyDir\bin\symbols\retail\dll\dbghelp.dbg
0000000047,ptr,\\foo2\bin\symbols\retail\dll\dbghelp.dbg
```

每当 refs.ptr 中的最后一个条目是指针时，文件 file.ptr 将存在并包含关联文件的路径。每当 refs.ptr 中的最后一个条目是文件时，此目录中将不存在 file.ptr。 因此，删除 refs.ptr 中最后一个条目的任何删除操作都可能导致创建、删除或更改 file.ptr。

AgeStore

项目 • 2024/02/15

AgeStore 工具 (agestore.exe) 根据上次访问日期删除目录或目录树中的文件。 为了节省磁盘空间，此程序特别适用于从符号服务器使用的下游存储中删除旧文件。

本节包括：

使用 AgeStore

AgeStore 命令行选项

使用 AgeStore

项目 • 2023/06/15

AgeStore 是一种工具，用于根据文件上次访问日期删除目录或目录树中的文件。 它的主要用途是从符号服务器或源服务器使用的下游存储中删除旧文件，以节省磁盘空间。 它还可用作常规文件删除工具。

AgeStore 可以删除目标目录) (单个 *目录中* 的所有文件，也可以删除目标 *树*) (树中的所有目录中的所有文件。 -s 选项指示要以整个树为目标。

有三种方法可以指定要删除目标目录或目标树中的哪些文件。 agestore -date=Month-Day-Year 命令删除在指定日期之前上次访问的所有文件。 agestore -days=NumberOfDays 命令删除上次访问时间超过指定天数的所有文件。 agestore -size=SizeRemaining 命令删除目标目录或目标树中的所有文件，从最近访问最少的文件开始，直到剩余文件的总大小小于或等于 *SizeRemaining*。

例如，以下命令删除 C：\MyDir 中上次在 2008 年 1 月 7 日之前访问的所有文件：

控制台

```
agestore c:\mydir -date=01-07-2008
```

以下命令删除从属于 C：\symbols\downstreamstore 的目录树中三十天前访问的所有文件：

控制台

```
agestore c:\symbols\downstreamstore -days=30 -s
```

以下命令删除从 C：\symbols\downstreamstore 的目录树中的文件，从访问时间最长的文件开始，直到此树中所有文件的总大小小于或等于 50，000 字节：

控制台

```
agestore c:\symbols\downstreamstore -size=50000 -s
```

-l 选项会导致 AgeStore 不删除任何文件，而只是列出在没有此选项的情况下将删除的所有文件。 在使用任何 AgeStore 命令之前，应运行添加了 -l 选项的预期命令，以验证它是否将删除你打算删除的文件。

有关完整的命令行语法，请参阅 **AgeStore Command-Line选项**。

在 Windows Vista 及更高版本上运行 AgeStore

由于 AgeStore 根据上次访问文件的时间删除文件，因此仅当文件系统存储“上次访问时间” (LAT) 数据时，它才能成功运行。 在 NTFS 文件系统中，可以启用或禁用 LAT 数据存储。 如果禁用，AgeStore 将不会运行，但将显示以下错误消息：

控制台

```
Last-Access-Time support is disabled on this computer.
Please read the documentation for more details.
```

在 Windows Vista 及更高版本的 Windows 中，默认情况下禁用 LAT 数据存储，因此除非你首先启用此数据，否则 AgeStore 将不会运行。

在 Windows Vista 及更高版本的 Windows 中，可以使用 FSUtil (Fsutil.exe) 工具来启用 LAT 数据的收集。 在命令提示符窗口中，发出以下命令：

控制台

```
fsutil behavior set disablelastaccess 0
```

若要禁用 LAT 数据收集，请使用以下命令：

控制台

```
fsutil behavior set disablelastaccess 1
```

这些更改在 Windows 下次重启后生效。

FAT32 文件系统始终存储 LAT 信息 (尽管仅存储日期而不是时间) 。 因此，AgeStore 适用于 FAT32 文件系统。 但是，由于禁用 NTFS LAT 时，AgeStore 不会运行，因此即使文件系统为 FAT32，也必须启用 NTFS LAT。

AgeStore 命令行选项

项目 • 2024/03/01

AgeStore 命令行使用以下语法。 参数可以按任意顺序包含。

控制台

```
agestore [PathSpec] -date=Month-Day-Year Options

agestore [PathSpec] -days=NumberOfDays Options

agestore [PathSpec] -size=SizeRemaining Options

agestore [PathSpec] -size Options

agestore -?
```

参数

PathSpec
指定要删除文件的目标目录。 如果使用 -s 选项，*PathSpec* 会指定要删除文件的目标树的根目录。 *PathSpec* 可以是本地计算机上的目录的绝对路径或相对路径，也可以是 UNC 路径。 如果 *PathSpec* 包含空格，则必须用引号将其引起来。 如果省略 *PathSpec*，*AgeStore* 将使用当前工作目录。

-date=*Month-Day-Year*
指定删除文件的截止日期。 在指定日期之前最后一次访问的所有文件都将被删除。 日期必须以“月-日-年”的格式指定，月和日之间、日与年之间应使用连字符。 *月份*和*日期*中的前导零是可选的。 年份可以指定为两位数或四位数。 因此，可以将日期 2008 年 1 月 5 日表示为 01-05-2008 或 1-5-08。

-days=*NumberOfDays*
指定删除文件的截止日期和时间。 在指定天数之前最后一次访问的所有文件都将被删除。 *NumberOfDays* 指定 24 小时天数的整数。 例如，如果在 2008 年 2 月 17 日下午 6:00 使用说明符 -days=3，则将被删除 2008 年 2 月 14 日下午 6:00 之前最后访问的所有文件。

-size=*SizeRemaining*
指定删除后应保留文件的总大小（以字节为单位）。 使用此开关时，AgeStore 将从最近访问的文件开始删除目标目录或目标树中的文件，直到剩余文件的总大小不高于 *SizeRemaining*。 使用 **-s** 选项时，AgeStore 将面向整个目录树，而且*SizeRemaining* 指定删除后应保留在整个目录树中的文件总大小。

-size
导致 AgeStore 列出目标目录或目标树中所有文件的总大小。 不删除任何文件。

Options 以下选项的任意组合。

-l
使 AgeStore 不删除任何文件，而只是列出在不使用 **-l** 选项的情况下运行此相同命令时将删除的所有文件。

-s
使 AgeStore 将从属 *PathSpec* 的整个目录树视为目标。 如果未使用 **-s** 选项，*PathSpec* 指定的目录将成为待删除文件的目标目录。 使用 **-s** 选项时，*PathSpec* 指定的目录及其下方的所有子目录将成为待删除文件的目标树。

-k
使 AgeStore 保留任何空子目录。 如果未使用此选项，则 AgeStore 会在命令运行后删除完全为空的目标目录。 如果在不使用 **-k** 选项的情况下使用 **-s** 选项，则当 AgeStore 完成文件删除后，目标目录树中的所有空目录都将被删除，即使根目录本身也变为空。 如果在 AgeStore 运行之前此树中有一些目录已经为空，则 AgeStore 也会删除这些目录。 但是，如果 AgeStore 命令不会导致文件删除（例如，如果 **-size**=*SizeRemaining* 参数指定的大小大于目标树中所有文件的总大小），则不会删除空目录。 如果未使用 **-s** 选项，则永远不会删除空目录，并且忽略 **-k** 选项。

-q
静默模式。 如果不包含此选项，AgeStore 会列出所有被删除的文件。

-y
禁止显示 **(y/n)** 提示符。 如果不使用此选项，AgeStore 会在删除任何文件之前提示您“您确定吗？”。

-?
显示 AgeStore 命令行的帮助文本。

其他信息

有关 AgeStore 工具的详细信息，请参阅使用 AgeStore。

DBH

项目 • 2024/06/19

DBH 工具 (dbh.exe) 是一种命令行工具，用于显示有关符号文件内容的信息。

DBH 通过方便的命令行接口公开 DbgHelp API （dbghelp.dll） 的功能。 因此，随着 DbgHelp 的更新，其行为可能会更改。 有关详细信息，请参阅 调试帮助库。

本节包括：

使用 DBH

其他 DBH 示例

DBH 命令行选项

DBH 命令

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用 DBH

项目 • 2024/06/19

DBH 是一种命令行工具，用于公开 DbgHelp API（dbghelp.dll）中的许多函数。 它可以显示有关符号文件内容的信息、显示文件中符号的特定详细信息，以及搜索文件中匹配各种条件的符号。 有关详细信息，请参阅 调试帮助库。

DBH 提供的功能类似于 WinDbg、KD 和 CDB 中通过 x（检查符号）**等**命令提供的功能。

在交互模式下运行 DBH

使用简单的命令行启动 DBH，在该命令行上指定要调查其符号的目标模块。 目标模块可以是 EXE 程序或 PDB 符号文件。 还可以指定要调查的进程 ID（PID）。 有关完整语法，请参阅 **DBH 命令行选项** 。

DBH 启动时，它会加载指定模块的符号，然后提示你键入各种命令。 有关可用命令的列表，请参阅 DBH 命令 。

例如，以下序列通过指定进程 ID 为 4672 的目标进程来启动 DBH，然后在 DBH 提示符处执行 **枚举** 命令以显示匹配特定模式的符号，然后执行 **q** 命令退出 DBH：

控制台

```
C:\> dbh -p:4672
            400000 : TimeTest
          77820000 : ntdll
          77740000 : kernel32

pid:4672 mod:TimeTest[400000]: enum TimeTest!ma*

 index            address     name
     1             42cc56 :   main
     3             415810 :   malloc
     5             415450 :   mainCRTStartup

pid:4672 mod:TimeTest[400000]: q

goodbye
```

在批处理模式下运行 DBH

如果只想运行单个 DBH 命令，可以在命令行末尾指定它。 这会导致 DBH 启动、加载指定的模块、运行指定的命令，然后退出。

例如，上一个示例可以替换为单个命令行：

控制台

```
C:\> dbh -p:4672 enum TimeTest!ma*
           400000 : TimeTest
         77820000 : ntdll
         77740000 : kernel32

index            address     name
    1             42cc56 :   main
    3             415810 :   malloc
    5             415450 :   mainCRTStartup
```

这种运行 DBH 的方法称为 *批处理模式*，因为它可以轻松地在批处理文件中使用。 此版本的命令行也可以后跟一个管道（|），该管道将 DBH 输出重定向到另一个程序。

指定目标

DBH 可以通过三种方式选择目标：按正在运行的进程 ID、可执行文件的名称或符号文件的名称。 例如，如果当前运行MyProg.exe只有一个实例，且进程 ID 为 1234，则以下命令几乎等效：

控制台

```
C:\> dbh -v -p:1234
C:\> dbh -v c:\mydir\myprog.exe
C:\> dbh -v c:\mydir\myprog.pdb
```

这些命令之间的一个区别是，通过指定进程 ID 启动 DBH 时，DBH 将使用此过程使用的实际虚拟地址。 通过指定可执行文件名称或符号文件名来启动 DBH 时，DBH 假定模块的基址是标准值（例如，0x01000000）。 然后，**可以使用 base** 命令指定实际的基址，从而移动模块中所有符号的地址。

DBH 不会以调试器的方式附加到目标进程。 DBH 不能导致进程开始或结束，也不能改变进程运行方式。 要使 DBH 通过其进程 ID 附加到进程，必须运行目标进程，但一旦启动 DBH，目标进程就可以终止，DBH 将继续访问其符号。

修饰符号和未装饰符号

默认情况下，DBH 在显示和搜索符号时使用未编码的符号名称。 如果关闭 SYMOPT_UNDNAME 符号选项，或在 DBH 命令行中包含 -d 选项，则会包含修饰。

有关符号修饰的信息，请参阅 公共符号和专用符号。

退出 DBH

若要退出 DBH，请在 DBH 提示符处使用 **q** 命令。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

其他 DBH 示例

项目 • 2024/02/15

下面是可在 DBH 提示符下发出的命令的其他示例。

显示专用符号和公共符号

如果目标是完整的符号文件，则每个公共符号都会在文件中出现两次：在公共符号表中，以及专用符号数据中。 公共符号表中的副本通常包含各种修饰（前缀和后缀）。 有关详细信息，请参阅 公共符号和专用符号。

DBH 可以从专用符号数据、公共符号表（不含修饰）以及带有修饰的公共符号表中显示有关此符号的信息。 下面是一个示例，其中所有三个都显示，每次使用命令 **addr 414fe0** 。

首次在此示例中显示此命令时，DBH 使用默认符号选项，因此生成的信息来自专用符号数据。 请注意，此信息包括函数 **fgets** 的地址、大小和数据类型。 然后，将使用命令符号 +4000，这将打开SYMOPT_PUBLICS_ONLY选项。 这会导致 DBH 忽略专用符号数据，因此 **，当 addr 414fe0** 命令第二次运行时，DBH 使用公共符号表，并且函数 **fget** 不显示大小或数据类型信息。 最后，使用命令符号 -2，关闭SYMOPT_UNDNAME选项，并导致 DBH 包含修饰。 **当 addr 414fe0** 在此最后一次运行时，它会显示函数名称的修饰版本，**_fgets。**

dbgcmd

```
pid:4308 mod:TimeTest[400000]: addr 414fe0

fgets
   name : fgets
   addr :   414fe0
   size : 113
  flags : 0
   type : 7e
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagFunction (5)
  index : 7d

pid:4308 mod:TimeTest[400000]: symopt +4000

Symbol Options: 0x10c13
Symbol Options: 0x14c13

pid:4308 mod:TimeTest[400000]: addr 414fe0
```

```
fgets
   name : fgets
   addr :   414fe0
   size : 0
  flags : 0
   type : 0
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagPublicSymbol (a)
  index : 7f

pid:4308 mod:TimeTest[400000]: symopt -2

Symbol Options: 0x14c13
Symbol Options: 0x14c11

pid:4308 mod:TimeTest[400000]: addr 414fe0

_fgets
   name : _fgets
   addr :   414fe0
   size : 0
  flags : 0
   type : 0
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagPublicSymbol (a)
  index : 7f
```

如果使用了 -d 命令行选项，则结果将从头开始显示修饰的公共名称。

确定特定符号的修饰

DBH 可以确定特定符号上的修饰。 当与需要指定符号的程序（如 PDBCopy）一起使用时，这非常有用。

例如，假设你知道符号文件 mysymbols.pdb 包含一个未编码的名称为 **MyFunction1** 的符号。 若要查找修饰的名称，请使用以下过程。

首先，在没有 -d 命令行选项的情况下启动 DBH，然后使用 symopt +4000 命令，以便所有信息都来自公共符号表：

```
控制台
```

```
C:\> dbh c:\mydir\mysymbols.pdb

mysymbols [1000000]: symopt +4000

Symbol Options: 0x10c13
Symbol Options: 0x14c13
```

接下来，使用 **name** 命令或 **枚举** 命令显示所需符号的地址：

```
dbgcmd
```

```
mysymbols [1000000]: enum myfunction1

 index            address     name
   2ab            102cb4e :   MyFunction1
```

现在，使用 symopt -2 使符号修饰可见，然后将 addr **命令与此符号的地址一**起使用：

```
dbgcmd
```

```
mysymbols [1000000]: symopt -2

Symbol Options: 0x14c13
Symbol Options: 0x14c11

mysymbols [1000000]: addr 102cb4e

_MyFunction1@4
   name : _InterlockedIncrement@4
   addr :  102cb4e
   size : 0
  flags : 0
   type : 0
modbase :  1000000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagPublicSymbol (a)
  index : 2ab
```

这显示符号的修饰名称_MyFunction1@4。

解码符号修饰

undec 命令可用于显示 C++ 符号修饰的含义。 在以下示例中，附加到 ？？ _C@_03GGCAPAJC@Sep？$AA@ 进行解码以指示它是字符串：

```
```

dbgcmd

```
dbh: undec ??_C@_03GGCAPAJC@Sep?$AA@

??_C@_03GGCAPAJC@Sep?$AA@ =
`string'
```

以下示例对附加到三个函数名称的修饰进行解码，并显示其原型：

dbgcmd

```
dbh: undec ?gcontext@@3_KA

?gcontext@@3_KA =
unsigned __int64 gcontext


dbh: undec ?pathcpy@@YGXPAGPBG1K@Z

?pathcpy@@YGXPAGPBG1K@Z =
void __stdcall pathcpy(unsigned short *,unsigned short const *,unsigned
short const *,unsigned long)


dbh: undec ?_set_new_handler@@YAP6AHI@ZP6AHI@Z@Z

?_set_new_handler@@YAP6AHI@ZP6AHI@Z@Z =
int (__cdecl*__cdecl _set_new_handler(int (__cdecl*)(unsigned int)))
(unsigned int)
```

undec 命令不显示有关初始下划线、前缀__**imp**_或尾随"address"@修饰的信息，这些修饰通常附加到函数名称。

可以将 **undec** 命令与任何字符串一起使用，而不仅仅是当前加载模块中的符号的名称。

按地址对符号列表进行排序

如果只是想要按地址顺序排序的符号列表，则可以在批处理模式下运行 DBH，并将结果通过管道传递给 **排序** 命令。 地址值通常在每个行的第 18 列中开始，因此以下命令按地址对结果进行排序：

dbgcmd

```
dbh -p:4672 enum mymodule!* | sort /+18
```

显示源行信息

使用完整符号文件时，DBH 可以显示源行信息。 这不需要访问任何源文件，因为此信息存储在符号文件中。

此处， **行** 命令显示对应于指定源行的二进制指令的十六进制地址，并显示与该行关联的符号。 （在此示例中，没有与该行关联的符号。

dbgcmd

```
dbh [1000000]: line myprogram.cpp#767

   file : e:\mydirectory\src\myprogram.cpp
   line : 767
   addr :  1006191
    key : 0000000000000000
disp : 0
```

在这里， **srclines** 命令显示与指定源行关联的对象文件：

dbgcmd

```
dbh [1000000]: srclines myprogram.cpp 767

0x1006191: e:\mydirectory\objchk\amd64\myprogram.obj
line 767 e:\mydirectory\src\myprogram.cpp
```

请注意，srclines 的**输出类似于 ln（列表最近的符号）调试器命令的**输出。

显示数据类型

type 命令可用于显示有关数据类型的信息。 此处显示有关 CMDPROC 类型的数据：

dbgcmd

```
dbh [1000000]: type CMDPROC

   name : CMDPROC
   addr :        0
   size : 8
  flags : 0
   type : c
modbase :  1000000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagTypedef (11)
  index : c
```

“tag”后面列出的值指定此数据类型的性质。 在这种情况下， **SymTagTypedef** 指示此类型是使用 **typedef** 语句定义的。

使用虚构符号

add 命令可以将虚构符号添加到加载的模块。 实际符号文件未更改;只会更改 DBH 内存中该文件的图像。

如果希望临时替代哪些符号与给定地址范围关联，则 **add** 命令非常有用。 在以下示例中，与 MyModule！main **关联的**地址范围的一部分被虚构符号 **MyModule！magic** 重写。

下面是在添加虚构符号之前模块的显示方式。 请注意， **主** 函数从0x0042CC56开始，大小0x42B。 因此，**当 addr** 命令与地址0x0042CD10一起使用时，它将此地址识别为位于主**函数边界**内的地址：

dbgcmd

```
pid:6040 mod:MyModule[400000]: enum timetest!ma*

 index            address     name
     1             42cc56 :   main
     3             415810 :   malloc
     5             415450 :   mainCRTStartup

pid:6040 mod:MyModule[400000]: addr 42cc56

main
   name : main
   addr :   42cc56
   size : 42b
  flags : 0
   type : 2
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagFunction (5)
  index : 1

pid:6040 mod:MyModule[400000]: addr 42cd10

main+ba
   name : main
   addr :   42cc56
   size : 42b
  flags : 0
   type : 2
modbase :   400000
  value :        0
```

```
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagFunction (5)
  index : 1
```

现在，符号 **magic** 将添加到地址0x0042CD00，大小0x10字节。 **使用枚举**命令时，将设置索引中的高位，显示这是虚构符号：

dbgcmd

```
pid:6040 mod:MyModule[400000]: add magic 42cd00 10


pid:6040 mod:MyModule[400000]: enum timetest!ma*

 index            address     name
     1             42cc56 :   main
     3             415810 :   malloc
     5             415450 :   mainCRTStartup
  80000001             42cd00 :   magic
```

使用 addr 命令时，它会查找其范围包括指定地址的任何符号。 由于此搜索以指定地址开头并向后运行，因此地址0x004CD10现在与 **magic** 相关联。 另一方面，地址0x004CD40仍然与主地址相关联，**因为它位于魔术符号的范围**之外。 另请注意，标记 **SymTagCustom** 指示虚构符号：

dbgcmd

```
pid:6040 mod:MyModule[400000]: addr 42cd10

magic+10
   name : magic
   addr :   42cd00
   size : 10
  flags : 1000
   type : 0
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagCustom (1a)
  index : 80000001

pid:6040 mod:MyModule[400000]: addr 42cd40

main+ea
   name : main
   addr :   42cc56
   size : 42b
  flags : 0
```

```
   type : 2
modbase :   400000
  value :        0
    reg : 0
  scope : SymTagNull (0)
    tag : SymTagFunction (5)
  index : 1
```

最后， **del** 命令可以删除符号 **magic**，将所有符号返回到其原始范围：

dbgcmd

```
pid:6040 mod:MyModule[400000]: del magic


pid:6040 mod:MyModule[400000]: enum timetest!ma*

 index            address     name
     1             42cc56 :   main
     3             415810 :   malloc
     5             415450 :   mainCRTStartup
```

DBH 命令行选项

项目 • 2024/07/08

DBH 命令行使用以下语法。

控制台

```
dbh [Options] -p:PID [Command]

dbh [Options] ExecutableName [Command]

dbh [Options] SymbolFileName [Command]

dbh -?

dbh -??
```

参数

-p:*PID*
指定要加载其符号的进程的进程 ID。

ExecutableName
指定要加载其符号的可执行文件，包括文件扩展名（通常为.exe或.sys）。 应包含相对目录路径或绝对目录路径;如果未包含路径，则假定当前工作目录。 如果未在此位置找到指定的文件，DBH 将使用 **SymLoadModuleEx** 搜索它。

SymbolFileName
指定要加载其符号的符号文件，包括文件扩展名 （.pdb 或 .dbg）。 应包含相对目录路径或绝对目录路径;如果未包含路径，则假定当前工作目录。

Options 以下选项的任意组合。

-d
导致在显示符号和搜索符号时使用修饰的名称。 使用此选项时，SYMOPT_PUBLICS_ONLY 处于打开状态，同时关闭SYMOPT_UNDNAME和SYMOPT_AUTO_PUBLICS。 这相当于在 DBH 运行时发出命令 symopt +4000，后跟 symopt -10002。

-s：*Path*
将符号路径设置为指定的 *Path* 值。

-n
打开 *干扰符号加载*。 将显示有关搜索符号的其他信息。 每个符号文件的名称在加载时显

示。 如果调试器无法加载符号文件，则会显示错误消息。 .pdb 文件的错误消息以文本显示。 .dbg 文件的错误消息采用错误代码的形式，如 winerror.h 文件中所述。 并非所有这些消息都很有用，但其中一些消息可能有助于分析找不到或匹配符号文件的原因。 如果仅加载图像文件以恢复符号标头信息，也会显示该文件。

命令
导致 DBH 运行、执行指定的 *命令*，然后退出。 有关可能命令的列表，请参阅 DBH 命令。

-?
显示 DBH 命令行的帮助文本。

-??
显示 DBH 命令行的帮助文本，并显示所有 DBH 命令的列表。

其他信息

有关 DBH 工具的详细信息，请参阅 “使用 DBH”。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

DBH 命令

项目 • 2024/07/08

在 DBH 命令行中，可以使用各种命令来分析符号和符号文件。

下表列出了控制 DBH 选项和执行其他基本任务的命令。

展开表

命令	效果
verbose [**on**|**off**]	打开或关闭详细模式。 如果没有参数，则显示当前详细模式设置。
同情 [*路径*]	设置符号搜索路径。 如果没有参数，则显示当前符号搜索路径。
symopt *选项* **symopt** +*Options* **symopt** -*Options* **symopt**	设置符号选项。 没有+或-，选项*的值*将替换当前符号选项。 如果 + 或 - 已使用， *选项* 指定要添加或删除的选项;必须有一个空格之前 + 或 - 之后没有空格。 如果没有参数，将显示当前符号选项。 启动 DBH 时，所有符号选项的默认值 0x10C13。 有关可用选项的列表，请参阅 "设置符号选项"。
help	显示 DBH 命令的帮助文本。
退出	退出 DBH 程序。

下表列出了加载、卸载和重新设置目标模块的命令。 如果在命令行上指定进程 ID 来启动 DBH，则无法使用这些命令。

展开表

命令	效果
加载*文件*	加载指定的模块。 *文件* 应指定可执行文件或符号文件的路径、文件名和文件扩展名。
卸载	卸载当前模块。
基*址*	将默认基址设置为指定值。 所有符号地址都将相对于此基址确定。

下表列出了搜索文件和显示目录信息的命令。

展开表

命令	效果
findexe *文件路径*	使用 **FindExecutableImage** 例程在指定路径中找到指定的可执行文件。
finddbg *文件路径*	在指定的路径中找到指定的 .dbg 文件。 包括 .dbg 扩展是可选的。
dir *文件路径*	使用 EnumDirTree **例程查找指定路径或此路径下的任何子目录中的**指定文件。
srchtree *路径文件*	使用 SearchTreeForFile **例程查找指定路径或此路径下的任何子目录中的**指定文件。 此命令与 **dir** 相同，不同之处在于参数是反向的。
ffpath *文件*	查找当前符号路径中的指定文件。

下表列出了分析模块列表并控制默认模块的命令。 默认模块及其基址显示在 DBH 提示符上。

展开表

命令	效果
mod *地址*	将默认模块更改为具有指定基址的模块。
刷新	刷新模块列表。
omap	显示模块 OMAP 结构。
epmod *PID*	枚举为指定进程加载的所有模块。 *PID* 指定所需进程的进程 ID。
info	显示有关当前加载的模块的信息。
obj *Mask*	列出与与指定模式匹配的默认模块关联的所有对象文件。 *掩码* 可能包含各种通配符和说明符;有关详细信息，请参阅 字符串通配符语法 。
src *掩码*	列出与与指定模式匹配的默认模块关联的所有源文件。 *掩码* 可能包含各种通配符和说明符;有关详细信息，请参阅 字符串通配符语法 。
enummod	枚举所有加载的模块。 除非 DBH 在没有目标的情况下运行，否则始终至少有一个模块，在这种情况下，没有任何模块。

下表列出了显示和搜索符号的命令。

展开表

命令	效果
枚举*模块！象征*	枚举与指定模块和符号匹配的所有符号。 *模块* 指定要搜索的模块（没有文件扩展名）。 *符号* 指定符号必须包含的模式。 模块和符号可以包含各种通配符和说明符;有关详细信息，请参阅 字符串通配符语法。
enumaddr *地址*	枚举与指定地址关联的所有符号。
addr *Address*	显示与指定地址关联的符号的详细信息。
名称 [*Module***!**]*象征*	显示有关指定符号的详细信息。 可以包含可选的 *模块* 说明符。 不应使用通配符，因为如果多个符号与模式匹配， **则名称** 仅显示其中第一个符号。
下一步 [*Module***!**]*象征* **下一个***地址*	显示指定符号或地址后下一个符号的详细信息。 如果符号是按名称指定的，则可以包含可选的 *模块* 说明符，但不应使用通配符。
上一页 [*Module***!**]*象征* **上一个***地址*	显示有关指定符号或地址前面第一个符号的详细信息。 如果符号是按名称指定的，则可以包含可选的 *模块* 说明符，但不应使用通配符。
line *File#LineNum*	显示与指定源行关联的二进制指令的十六进制地址，以及与此行关联的任何符号。 此外，将当前行号设置为等于指定的行号。 *文件*指定源文件的名称，*LineNum* 指定该文件中的行号;应用数字符号（）。 #
srclines *File LineNum*	显示与指定源行关联的对象文件，以及与此行关联的二进制指令的十六进制地址。 不更改当前行号。 *文件* 指定源文件的名称， *LineNum* 指定该文件中的行号;应用空格分隔这些行号。
laddr *地址*	显示对应于位于指定地址的符号的源文件和行号。
linenext	递增当前行号，并显示有关新行号的信息。
lineprev	递减当前行号，并显示有关新行号的信息。
局部变量*函数* [*掩码*]	显示指定函数中包含的所有局部变量。 如果 *包含 Mask* ，则仅显示与指定模式匹配的局部变量;有关详细信息，请参阅 字符串通配符语法 。
typeName	显示有关指定数据类型的详细信息。 *TypeName* 指定数据类型的名称（例如 WSTRING）。 如果没有类型名称与此值匹配，将显示任何匹配符

命令	效果
	号。 与大多数 DBH 命令参数不同， *TypeName* 区分大小写。
elines [Source *[Obj]*]	枚举与指定的源掩码和对象掩码匹配的所有源行。 *源* 指定源文件的名称，包括绝对路径和文件扩展名。 *Obj* 指定对象文件的名称，包括相对路径和文件扩展名。 Source 和 Obj *可能包含各种通配符和说明符;有关详细信息，请参阅字符串通配符语法。* 如果省略参数，则这相当于使用星号 （） *通配符。 如果不想指定路径信息，请使用 /strong 为文件名添加<* **前缀以指示通配符路径。** >
索引*值*	显示有关具有指定索引值的符号的详细信息。
范围*地址* **范围** [*Module***!**]*象征*	显示有关指定符号的父级的详细信息。 该符号可以按地址或名称指定。
srch [**mask**=*Symbol*] [**index**=*Index*] [**tag**=*Tag*] [addr=***Address***] [**globals**]	搜索与指定掩码匹配的所有符号。 *符号* 指定符号名称。 它不应包含模块名称，但它可能包含通配符和说明符;有关详细信息，请参阅 字符串通配符语法 。 *索引* 指定要用作搜索父级的符号的十六进制地址。 *标记* 指定必须与符号匹配的十六进制符号类型分类器 （**SymTag***Xxx*） 值。 *地址* 指定符号的地址。 如果 **包含全局符号** ，则仅显示全局符号。
uw *地址*	显示指定地址处函数的展开信息。
dtag	显示所有符号类型分类器（**SymTag***Xxx*）值。
etypes	枚举所有数据类型。
dump	显示目标文件中所有符号信息的完整列表。

下表列出了与符号服务器和符号存储相关的命令。

展开表

命令	效果
home [*Path*]	设置 SymSrv 和 SrcSrv 用于默认下游存储的主目录。 如果符号路径包含对使用默认下游存储的符号服务器的引用，则 **主目录的符号** 子目录将用于下游存储。 如果没有参数， **home** 将显示当前主目录。
srvpath *路径*	测试指定的路径是否为符号存储的路径。

命令	效果
srvind *文件*	查找与指定文件相对应的符号服务器索引。 符号服务器索引是基于文件内容的唯一值，无论它是否实际已添加到任何符号存储中。 *文件* 应指定所需文件的文件名和绝对路径。
fii *文件*	显示指定二进制文件的符号服务器索引及其关联的文件。
getfile *文件索引*	显示具有指定名称和符号服务器索引的文件。 *文件* 指定所需文件的名称;这不应包含其路径。 *索引* 指定所需文件的符号服务器索引。 DBH 使用 **SymFindFileInPath** 例程在当前符号路径下搜索具有此名称和此索引的文件的树。
sup *Path File1 File2*	根据参数的值将文件存储在符号存储中。 *路径* 指定符号存储的目录路径。 *File1* 和 *File2* 用于创建增量值，而增量值又用于确定要存储的文件。
storeadd *文件存储*	将指定文件添加到指定的符号存储区。 *存储* 应是符号存储的根路径。

下表列出了应用于真实符号和虚构符号的 DBH 命令。

展开表

命令	效果
undec *Name*	显示附加到指定符号名称的修饰的含义。 *名称* 可以是任何字符串;它不需要对应于当前加载的符号。 如果 *Name* 包含C++修饰，则会显示这些修饰的含义。
添加*名称地址大小*	将指定的虚构符号添加到 DBH 中加载的符号列表中。 *名称*指定要添加的符号的名称，*Address* 指定要添加的十六进制地址，并指定其十六进制大小（以字节为单位）。 这在后面的 DBH 命令中被视为任何其他符号，直到 DBH 会话以退出或卸载结束，**或者直到使用 del 删除虚构符号为止。**不会更改实际的目标符号文件。
del *Name* **del** *Address*	删除之前使用 **add** 命令添加的虚构符号。 可以按名称或地址指定符号。 这不能用于删除实际符号。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

PDBCopy

项目 • 2023/06/15

PDBCopy 工具 (pdbcopy.exe) 是一种命令行工具，用于从符号文件中删除专用符号信息。 它还可以从公共符号表中删除所选信息。

本节包括：

使用 PDBCopy

选择要删除的公共符号

PDBCopy 命令行选项

使用 PDBCopy

项目 • 2023/06/14

PDBCopy 是一种命令行工具，用于从完整符号文件创建去除符号文件。 换句话说，它采用同时包含私有符号数据和公共符号表的符号文件，并创建仅包含公共符号表的文件的副本。 根据使用的 PDBCopy 选项，去除符号文件包含整个公共符号表或公共符号表的指定子集。

有关 PDBCopy 在 WDK 中的位置，请参阅 Windows 调试工具中包含的工具中的**安装目录。**

PDBCopy 适用于文件扩展名为 .pdb) 的任何 PDB 格式符号文件 (，但不适用于较旧格式 (.dbg) 符号文件。

有关公共符号表和私有符号数据的说明，请参阅 公共符号和专用符号。

删除私有符号

如果要创建包含所有公共符号且不包含私有符号的剥离符号文件，请使用 PDBCopy 和三个参数：原始符号文件的路径和名称、新符号文件的路径和名称以及 -p 选项。

例如，以下命令创建一个名为 publicsymbols.pdb 的新文件，该文件包含与 mysymbols.pdb 相同的公共符号表，但不包含私有符号数据：

pdbcopy mysymbols.pdb publicsymbols.pdb -p

如果 mysymbols.pdb 恰好已经是一个去除的符号文件，则新文件和旧文件的符号内容将相同。

发出此命令后，应将新文件移动到新位置，并将其重命名为符号文件的原始名称 (在此示例中为 mysymbols.pdb) ，因为大多数调试程序和符号提取程序会根据特定文件名查找符号。 或者，只要指定了不同的目录，就可以对 PDBCopy 命令行上的输入文件和输出文件使用相同的文件名：

pdbcopy c：\dir1\mysymbols.pdb c：\dir2\mysymbols.pdb -p

注意 在运行 PDBCopy 之前，目标文件不应存在。 如果存在具有此名称的文件，则可能会出现各种错误。

删除私有符号和选定的公共符号

如果不仅要删除私有符号数据，还要减少公共符号表中的信息量，可以使用 -f 选项指定要删除的公共符号的列表。

以下示例演示了此过程：

1. 确定要删除的符号的全名（包括修饰）。 如果不确定修饰符号名称，可以使用 DBH 工具来确定它们。 有关详细信息，请参阅确定特定符号的修饰。 在此示例中，假设要删除的符号的修饰名称 **_myGlobal1** 且 **_myGlobal2**。

2. 创建包含要删除的符号列表的文本文件。 此文件中的每一行都应包含一个符号的名称，包括修饰，但不包括模块名称。 在此示例中，文件将包含以下两行：

   ```
   text

   _myGlobal1
   _myGlobal2
   ```

   可以为文件指定你选择的任何名称。 假设将此文件命名listfile.txt，并将其放在目录 C：\Temp 中。

3. 使用以下 PDBCopy 命令行：

   ```
   控制台

   pdbcopy OldPDB NewPDB -p -f:@TextFile
   ```

   其中， *OldPDB* 和 *NewPDB* 是原始符号文件和新符号文件， *TextFile* 是在步骤 2 中创建的文件。 -f 选项指示要删除某些公共符号，和号（@）指示这些符号列在指定的文本文件中。

   在当前示例中，命令如下所示：

   ```
   控制台

   pdbcopy c:\dir1\mysymbols.pdb c:\dir3\mysymbols.pdb -p -
   f:@c:\temp\listfile.txt
   ```

   这会创建一个新的符号文件 C：\dir3\mysymbols.pdb，该文件不包含任何私有符号，并且不包含listfile.txt中列出的两个全局变量。

如本示例所示，PDBCopy 的 -f 选项会删除公共符号的特定列表。 @）的和号 (指示这些符号在文本文件中列出。 另一种方法是列出命令行上的所有符号，使用不带和号的 -f 选项。 因此，以下命令行等效于上述过程中的示例：

pdbcopy c：\dir1\mysymbols.pdb c：\dir3\mysymbols.pdb -p -f：_myGlobal1 -f：_myGlobal2

除非只想删除一两个符号，否则使用文本文件比在命令行上列出它们更简单。

如果要从 .pdb 文件中删除大多数公共符号，则 -F 选项是最简单的方法。 虽然 -f 选项要求列出要删除的公共符号，但 -F 选项要求列出不希望删除的公共符号。 将删除所有其他(公共符号以及) 的所有私有符号。 -F 选项支持与 -f 选项相同的两个语法选项：-F： 后跟要保留的符号的名称，或 -F：@ 后跟包含要保留的符号列表的文本文件的名称。 在任一情况下，都必须使用修饰符号名称。

例如，以下命令删除所有私有符号和几乎所有公共符号，只保留 **_myFunction5** 和 **_myGlobal7**符号：

pdbcopy c：\dir1\mysymbols.pdb c：\dir3\mysymbols.pdb -p -F：_myFunction5 -F：_myGlobal7

如果在一行中合并了 -f 选项的多个实例，则会删除所有指定的符号。 如果在一行中合并了 -F 选项的多个实例，则保留所有指定的符号，并删除所有其他符号。 不能将 -f 与 -F 组合在一起。

如果没有 -p 选项，则不能使用 -f 和 -F 选项，这将删除所有私有符号数据。 即使原始文件不包含私有符号，你仍必须包括 -p 选项 (，尽管在这种情况下它不起作用) 。

-F 选项不能用于防止删除专用符号数据。 如果将此选项与未包含在公共符号表中的符号一起使用，PDBCopy 将忽略它。

mspdb*.dll 文件

PDBCopy 必须访问 Mspdb80.dll 文件或 Mspdb60.dll 文件才能运行。 默认情况下，PDBCopy 使用 Mspdb80.dll，这是 Visual Studio .NET 2002 和更高版本的 Visual Studio 所使用的版本。 如果符号是使用 Visual Studio 6.0 或更早版本生成的，则可以指定 -vc6 命令行选项，以便 PDBCopy 改用 Mspdb60.dll，尽管这不是必需的。 即使不使用 -vc6 选项，PDBCopy 也会查找相应的文件。 可以在安装 Visual Studio、平台 SDK 或 Windows 驱动程序工具包 (WDK) 中找到这些文件。

在运行 PDBCopy 之前，请确保计算机可以访问正确版本的 mspdb*.dll 文件，并确保其位置是命令路径的一部分。 如果不是，则应使用 **path** 命令将此位置添加到命令路径。

文件签名和 SymSrv 索引

每个符号文件都有一个唯一标识它的固定签名。 SymSrv 使用签名为文件生成唯一的“索引值”。 如果两个文件具有不同的内容或不同的创建时间，它们还将具有不同的签名和不同的 SymSrv 索引值。

使用 PDBCopy 创建的文件是唯一索引值规则的例外。 PDBCopy 创建新的符号文件时，其签名和 SymSrv 索引值与旧文件相同。 此功能允许将一个文件替换为另一个文件，而无需更改符号相关工具的行为。

如果希望新文件具有不同的签名和 SymSrv 索引，请使用 -s 选项。 在大多数情况下，你不希望使用此选项，因为 PDBCopy 最常见的用法是创建可替换旧文件的更改符号文件，而不会导致不匹配。 新签名可能会导致 SymSrv 为新文件分配与旧文件不同的索引值，从而阻止新文件正确替换旧文件。

有关完整的命令行语法，请参阅 **PDBCopy Command-Line 选项**。

选择要删除的公共符号

项目 • 2024/03/04

PDBCopy 提供 -f 和 -F 选项，以便可以从剥离的符号文件中删除任意一组公共符号，只保留访问群体为了执行调试而需要访问的那些符号。

PDBCopy 的常见用途是创建特殊版本的符号文件，供 Microsoft 在其联机崩溃分析 (OCA) 程序中使用。 OCA 可以将某些函数指定为*惰性*，这意味着如果在堆栈跟踪上找到该函数，则忽略该函数。 如果一个函数只是一个包装器或不执行重要计算的“传递”函数，那么它通常会被声明为惰性函数。 如果在故障分析中在堆栈上发现了这样的函数，则可以假设该函数本身没有故障，并且它最多传递了从堆栈早期例程接收的无效或损坏的数据。 通过忽略此类函数，OCA 可以更好地确定错误或损坏的实际原因。

当然，任何想要声明为“惰性”的函数都需要包含在 OCA 使用的符号文件的公共符号表中。 然而，正如下面的示例所示，这些并不是唯一需要包含的函数。

假设你编写 Windows 驱动程序，并使用 PDBCopy 从其符号文件中删除所有公共符号，但 **FunctionOne** 和 **FunctionSix** 这两个惰性函数除外。 你的期望是，如果在崩溃后在堆栈上找到 **FunctionOne** 或 **FunctionSix**，则 OCA 将忽略它们。 如果驱动程序的任何其他部分位于堆栈上，Microsoft 将提供相应的内存地址，并且你可以使用该地址调试驱动程序。

然而，让我们假设你的驱动程序占用内存如下布局：

展开表

Address	内存内容
0x1000	模块的基址
0x2000	**FunctionOne** 的开头
0x203F	**FunctionOne** 的结尾
0x3000	**FunctionSix** 的开头
0x305F	**FunctionSix** 的结尾
0x7FFF	内存中模块的末尾

如果调试器在堆栈上找到地址，则选择具有下一个较低地址的符号。 由于公共符号表包含每个符号的地址，但没有大小信息，因此调试器无法知道地址是否实际位于任何特定符号的边界内。

因此，如果在地址 0x2031 发生错误，Microsoft OCA 运行的调试器将正确地识别故障位于 **FunctionOne** 内。 由于这是一个惰性函数，调试器将继续遍查堆栈以找出崩溃的原因。

但是，如果在 0x2052 处发生故障，调试器仍然将此地址与 **FunctionOne** 匹配，即使它超出了此函数的实际末尾 (0x203F)。

因此，必须在剥离的符号文件中不仅包括要公开的函数，还包括紧跟在这些函数之后的符号。 在本例中，你想要公开 **FunctionOne**、**FunctionTwo**、**FunctionSix** 和 **FunctionSeven**：

展开表

Address	内存内容
0x1000	模块的基址
0x2000	**FunctionOne** 的开头
0x203F	**FunctionOne** 的结尾
0x2040	**FunctionTwo** 的开头
0x3000	**FunctionSix** 的开头
0x305F	**FunctionSix** 的结尾
0x3060	**FunctionSeven** 的开头
0x7FFF	内存中模块的末尾

如果将这四个函数全部包含在剥离的符号文件中，则 Microsoft OCA 分析不会错误地将地址 0x2052 视为 **FunctionOne** 的一部分。 在本例中，假设此地址是 **FunctionTwo** 的一部分，但这并不重要，因为你尚未向 OCA 注册 **FunctionTwo** 作为惰性函数。 重要的是，地址 0x2052 被识别为不属于惰性函数，因此 OCA 会将其识别为驱动程序中的一个有意义的故障，并可以通知你该故障。

如果不希望公布每个惰性函数后面的函数的名称，可以在每个惰性函数之后的代码中插入不重要的函数，以便将这些函数的名称包含在公共符号文件中。 请确保验证这些添加的函数是否确实遵循二进制文件地址空间中的惰性函数，因为一些优化例程可能会改变这一点，甚至完全删除一些函数。

PDBCopy 命令行选项

项目 • 2024/07/08

PDBCopy 命令行使用以下语法。 参数可以按任意顺序包含。

```
dbgcmd

pdbcopy OldPDB NewPDB [Options]

pdbcopy OldPDB NewPDB -p [-f:Symbol] [-f:@TextFile] [Options]

pdbcopy OldPDB NewPDB -p [-F:Symbol] [-F:@TextFile] [Options]

pdbcopy InputPDB BackupPDB -CVE-2018-1037 [autofix|verbose]

pdbcopy /?
```

参数

OldPDB
指定要读取的原始符号文件的路径和文件名，包括 .pdb 文件扩展名。 *OldPDB* 可能包含本地计算机上的目录的绝对路径或相对路径，或者包含 UNC 路径。 如果未指定路径，则使用当前工作目录。 如果 *OldPDB* 包含空格，则必须用引号括起来。

NewPDB
指定要创建的新符号文件的路径和文件名，包括 .pdb 文件扩展名。 *NewPDB* 可能包含本地计算机上的目录的绝对路径或相对路径，或者包含 UNC 路径。 此路径必须已存在;PDBCopy 不会创建新目录。 如果未指定路径，则使用当前工作目录。 如果 *NewPDB* 包含空格，则必须用引号将其括起来。 指定的文件不应已经存在;如果存在，则新文件可能未写入，或写入错误。

-p
导致 PDBCopy 从新符号文件中删除专用符号数据。 如果旧符号文件不包含专用符号，则此选项不起作用。 如果省略此选项，PDBCopy 将创建一个与原始文件相同的符号内容的新文件。

-f: *Symbol*
导致 PDBCopy 从新符号文件中删除指定的公共符号。 *符号* 必须指定要删除的符号的名称，包括任何符号名称修饰（例如初始下划线），但不包括模块名称。 此选项需要 -p 选项。 如果使用多个 **-f** 或 **-f:** @ 参数，PDBCopy 将从新符号文件中删除所有指定的符号。

-f： @*TextFile*
导致 PDBCopy 从新符号文件中删除指定文本文件中列出的公共符号。 *TextFile* 指定此文件的文件名和路径（绝对或相对）。 此文件可以列出任意数量的符号的名称，每行一个符号，包括任何符号名称修饰（例如初始下划线），但不包括模块名称。 此选项需要 -p 选项。

-F： *Symbol*
导致 PDBCopy 从新符号文件中删除所有公共符号和专用符号，但指定的公共符号除外。 *符号* 必须指定要保留的符号的名称，包括任何符号名称修饰（例如，初始下划线），但不包括模块名称。 此选项需要 -p 选项。 如果使用多个 **-F 或 -F：** @ 参数，则所有指定的符号都将保留在新符号文件中。

-F： @*TextFile*
导致 PDBCopy 从新符号文件中删除所有公共符号和专用符号，但指定文本文件中列出的公共符号除外。 *TextFile* 指定此文件的文件名和路径（绝对或相对）。 此文件可以列出任意数量的符号的名称，每行一个符号，包括任何符号名称修饰（例如初始下划线），但不包括模块名称。 此选项需要 -p 选项。

Options 以下选项的任意组合。 这些选项区分大小写。

-s
使新符号文件具有与旧文件不同的签名。 通常不应使用 -s 选项，因为新签名可能会导致 SymSrv 为新文件分配与旧文件不同的索引值，从而阻止新文件正确替换旧文件。

-vc6
导致 PDBCopy 使用mspdb60.dll而不是mspdb80.dll。 此选项从不需要，因为 PDBCopy 会自动查找正确的 mspdb*.dll版本。 默认情况下，PDBCopy 使用 mspdb80.dll，这是 Visual Studio .NET 2002 和更高版本的 Visual Studio 使用的版本。 如果符号是使用 Visual Studio 6.0 或早期版本生成的，则可以指定此命令行选项，以便 PDBCopy 改用 mspdb60.dll。 但是，这不是必需的，因为即使不使用此选项，PDBCopy 也会查找相应的文件。 无论使用哪种版本的 mspdb*.dll，都必须位于启动 PDBCopy 的命令提示符窗口的可执行路径中。

-CVE-2018-1037

报告 InputPDBFile 是否具有 CVE-2018-1037 中所述的问题，并选择性地修正问题。 有关详细信息和详细的使用情况信息，请参阅 KB# 4131751 - PDBCopy 工具 。

-?
显示 PDBCopy 命令行的帮助文本。

其他信息

有关 PDBCopy 工具的详细信息，请参阅 使用 PDBCopy。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

SymChk

项目 • 2023/10/23

SymChk (Microsoft 符号检查器工具) **Symchk.exe**是一个将可执行文件与符号文件进行比较以验证是否可用的正确符号的程序。

本节包括：

使用 SymChk

SymChk 命令行选项

使用 SymChk

项目 • 2023/10/23

SymChk 的基本语法如下：

控制台

```
symchk [/r] FileNames /s SymbolPath
```

FileNames 指定需要其符号的一个或多个程序文件。 如果 *FileNames* 是一个目录，并且使用了 **/r** 标志，则以递归方式浏览此目录，SymChk 将尝试在此目录树中查找所有程序文件的符号。 *SymbolPath* 指定 SymChk 搜索符号的位置。

还有更多命令行选项。 有关完整列表，请参阅 SymChk Command-Line 选项。

获取 symchk

与其他调试工具一样，Symchk 作为调试器的一部分提供。 有关详细信息，请参阅 Debugging Tools for Windows（Windows 调试工具）。

安装调试工具后，symchk 将在此目录中用于 64 位 Windows。

C:\Program Files (x86)\Windows Kits\10\Debuggers\x64

示例用法

指定的符号路径可以包含任意数量的本地目录、UNC 目录或符号服务器。 本地目录和 UNC 目录不以递归方式搜索。 仅搜索指定的目录和基于可执行文件的扩展的子目录。 例如，查询

控制台

```
symchk thisdriver.sys /s G:\symbols
```

将搜索 G：\mysymbols 和 G：\mysymbols\sys。

可以通过使用以下任一语法作为符号路径的一部分来指定符号服务器：

控制台

```
srv*DownstreamStore*\\Server\Share
```

```
srv*\\Server\Share
```

这与在调试器的符号路径中使用符号服务器非常相似。 有关此内容的详细信息，请参阅使用符号服务器和符号存储。

如果指定了下游存储，SymChk 将复制符号服务器找到的所有有效符号文件，并将其放置在下游存储中。 仅将完全匹配的符号文件复制到下游。

SymChk 始终在查询符号服务器之前搜索下游存储。 因此，当其他人维护符号存储时，应小心使用下游存储。 如果运行 SymChk 一次并找到符号文件，则会将这些符号文件复制到下游存储。 如果在符号存储上更改或删除这些文件后再次运行 SymChk，SymChk 将不会注意到这一事实，因为它会在下游存储中找到它正在查找的内容，而不再查找。

注意 SymChk 始终使用 SymSrv (Symsrv.dll) 作为其符号服务器 DLL。 另一方面，如果符号服务器 DLL 可用，调试器可以选择符号服务器 DLL 而不是 SymSrv。 (SymSrv 是 Windows 调试工具包中包含的符号服务器。)

使用 SymChk 确定符号是私有符号还是公共符号

若要确定符号文件是私有还是公共，请使用 **/v** 参数，以便 SymChk 显示详细输出。 假设 MyApp.exe 和 MyApp.pdb 位于 c：\sym 文件夹中。 输入此命令：

控制台

```
symchk /v C:\sym\MyApp.exe /s C:\sym**
```

如果 MyApp.pdb 包含专用符号，则 SymChk 的输出如下所示。

控制台

```
[SYMCHK] Searching for symbols to c:\sym\MyApp.exe in path c:\sym
...
DBGHELP: MyApp - private symbols & lines
        c:\sym\MyApp.pdb
...
SYMCHK: FAILED files = 0
SYMCHK: PASSED + IGNORED files = 1
```

如果 MyApp.pdb 仅包含公共符号，则 SymChk 的输出如下所示。

控制台

```
[SYMCHK] Searching for symbols to c:\sym\MyApp.exe in path c:\sym
...
DBGHELP: MyApp - public symbols
```

```
        c:\sym\MyApp.pdb
...
SYMCHK: FAILED files = 0
SYMCHK: PASSED + IGNORED files = 1
```

若要限制搜索，使其仅查找公共符号文件，请将 **s** 选项与 **/s** 参数 (**/ss**) 。 如果 MyApp.pdb 仅包含公共符号，则以下命令将查找匹配项。 如果 MyApp.pdb 包含私有符号，则找不到匹配项。

控制台

```
symchk /v C:\sym\MyApp.exe /s C:\sym
```

有关详细信息，请参阅 公共符号和专用符号。

示例

下面是一些示例。 以下命令搜索程序 Myapp.exe 的符号：

控制台

```
E:\debuggers> symchk F:\myapp.exe /s F:\symbols\applications

SYMCHK: Myapp.exe           FAILED  - Myapp.pdb is missing

SYMCHK: FAILED files = 1
SYMCHK: PASSED + IGNORED files = 0
```

可以使用其他符号路径重试：

控制台

```
E:\debuggers> symchk F:\myapp.exe /s F:\symbols\newdirectory

SYMCHK: FAILED files = 0
SYMCHK: PASSED + IGNORED files = 1
```

这次搜索成功。 如果未使用详细选项，SymChk 将仅列出找不到符号的文件。 因此，在此示例中未列出任何文件。 可以判断搜索成功，因为“已通过”类别中列出了一个文件，“失败”类别中没有文件。

如果程序文件不包含可执行代码，则忽略它。 许多资源文件属于此类型。

如果希望查看所有程序文件的文件名，可以使用 **/v** 选项生成详细输出：

控制台

```
E:\debuggers> symchk /v F:\myapp.exe /s F:\symbols\newdirectory

SYMCHK: MyApp.exe           PASSED

SYMCHK: FAILED files = 0
SYMCHK: PASSED + IGNORED files = 1
```

以下命令在符号服务器中搜索大量 Windows 符号。 可能存在多种错误消息：

控制台

```
E:\debuggers> symchk /r C:\windows\system32 /s srv*\\manysymbols\windows

SYMCHK: msisam11.dll         FAILED  - MSISAM11.pdb is missing
SYMCHK: msuni11.dll          FAILED  - msuni11link.pdb is missing
SYMCHK: msdxm.ocx            FAILED  - Image is split correctly, but
msdxm.dbg i
s missing
SYMCHK: expsrv.dll           FAILED  - Checksum doesn't match with
expsrv.DBG
SYMCHK: imeshare.dll         FAILED  - imeshare.opt.pdb is missing
SYMCHK: ir32_32.dll          FAILED  - Built with no debugging information
SYMCHK: author.dll           FAILED  - rpctest.pdb is missing
SYMCHK: msvcrt40.dll         FAILED  - Built with no debugging information
......
SYMCHK: FAILED files = 211
SYMCHK: PASSED + IGNORED files = 4809
```

另请参阅

SymChk 命令行选项

使用符号服务器和符号存储

将清单文件与 SymChk 配合使用

项目 • 2023/05/09

在某些情况下，可能需要检索独立计算机上的文件的符号;也就是说，不在任何网络上或位于没有符号存储的网络上的计算机。 在这种情况下，可以使用以下过程检索符号。

1. 使用 **/om** 参数运行 SymChk 以创建描述要检索符号的文件的清单文件。
2. 将清单文件移动到具有符号存储的网络。
3. 使用 **/im** 参数运行 SymChk 以检索清单文件中描述的文件的符号。
4. 将符号文件移回独立计算机。

示例

假设yourApp.exe在独立计算机上运行。 以下命令创建一个清单文件，该文件描述调试yourApp.exe过程所需的所有符号。

dbgcmd

```
C:\>SymChk /om c:\Manifest\man.txt /ie yourApp.exe

SYMCHK: FAILED files = 0
SYMCHK: PASSED + IGNORED files = 28
```

现在，假设已将清单文件移动到网络上有权访问符号存储的其他计算机。 以下命令检索清单文件中描述的符号，并将其放在 mySymbols 文件夹中。

dbgcmd

```
C:\>SymChk /im c:\FolderOnOtherComputer\man.txt /s
srv*c:\mysymbols*\\aServer\symbols

SYMCHK: myApp.exe             ERROR - Unable to download file. Error
reported was 2
. . .
SYMCHK: FAILED files = 28
SYMCHK: PASSED + IGNORED files = 28
```

现在，可以将符号移动到独立计算机，并将它们用于调试。

SymChk 命令行选项

项目 • 2024/07/08

SymChk 使用以下语法：

```
dbgcmd
```

```
symchk [/r] [/v | /q ] FileNames /s[Opts] SymbolPath Options

symchk [/r] [/v | /q ] /ie ExeFile /s[Opts] SymbolPath Options

symchk [/r] [/v | /q ] /id DumpFile /s[Opts] SymbolPath Options

symchk [/r] [/v | /q ] /ih HotFixFile /s[Opts] SymbolPath Options

symchk [/r] [/v | /q ] /ip ProcessID /s[Opts] SymbolPath Options

symchk [/r] [/v | /q ] /it TextFileList /s[Opts] SymbolPath Options

symchk [/r] [/v | /q ] /om Manifest FileNames

symchk [/v | /q ] /im ManifestList /s[Opts] SymbolPath Options

symchk [/v | /q ] /om Manifest /ie ExeFile

symchk [/v | /q ] /om Manifest /id DumpFile

symchk [/v | /q ] /om Manifest /ih HotFixFile

symchk [/v | /q ] /om Manifest /ip ProcessFile

symchk [/v | /q ] /om Manifest /it TextFileList
```

参数

/r
如果 *文件* 指定目录， **则 /r** 选项会导致 SymChk 以递归方式在此目录下搜索程序文件的所有子目录。

/v
显示详细信息。 这包括调查其符号的每个程序文件的文件名，以及它是否传递、失败或被忽略。

/q
启用静默模式。 所有输出将被禁止（除非 **包含 /ot** 选项）。

文件名
指定要检查其符号的程序文件。 允许绝对路径、相对路径和 UNC 路径。 允许使用星号（*） 通配符。 如果 *FileNames 以斜杠结尾，则将其设置为目录名称，并检查该目录中的所有文件。 如果 *FileNames* 包含空格，则必须用引号引起来。

/ie *ExeFile*
指定当前正在执行的程序的名称。 将检查此程序的符号。 *ExeFile* 必须包含文件名和文件扩展名（通常.exe），但不包含路径信息。 如果具有相同名称的两个不同的可执行文件，则不建议使用此选项。 *ExeFile* 可以指定任何程序，包括内核模式驱动程序。 如果 *ExeFile* 是单个星号（*），SymChk 将检查所有正在运行的进程（包括驱动程序）的符号。

/id *DumpFile*
指定内存转储文件。 将检查此转储文件的符号。

/ih *HotFixFile*
指定自解压缩修补程序 CAB 文件。

/ip *ProcessID*
指定当前正在执行的程序的进程 ID。 将检查此程序的符号。 *ProcessID* 必须指定为十进制数。 支持两个特殊通配符：

- 如果 *ProcessID* 为零（ **0** ），SymChk 将检查所有正在运行的驱动程序的符号。
- 如果 *ProcessID* 是单个星号（*），SymChk 将检查所有正在运行的进程（包括驱动程序）的符号。

/它 *TextFileList*
指定包含程序文件列表的文本文件。 将检查所有这些程序的符号。 *TextFileList* 必须指定一个文件（按相对文件、绝对路径或 UNC 路径），但不包含通配符）;如果它包含空格，则应用引号括起来。 在此文件中，每行都指示程序文件（按相对路径、绝对路径或 UNC 路径），并允许使用星号通配符 （*）。 但是，使用此通配符的任何行都必须使用相对路径。

如果此文件中的一行包含空格，则应用引号括起来。 此文件中的分号是注释字符 -- 将忽略分号和行尾之间的所有内容。

/我 *ManifestList*
指定命令的输入是以前使用 **/om** 参数创建的清单文件。 清单文件包含有关检索符号的文件的信息。 有关使用清单文件的详细信息，请参阅 将清单文件与 SymChk 配合使用。

/om *清单*
指定创建清单文件。 清单文件包含一组有关稍后使用 **/im** 参数检索符号的文件集的信息。

/s[*Opts*] *SymbolPath*
指定包含符号的目录。 允许绝对路径、相对路径和 UNC 路径。 可以指定任意数量的目录 - 多个目录应用分号分隔。 如果 *SymbolPath* 包含空格，则必须用引号引起来。 如果要在此路径中指定符号服务器，应使用以下语法之一：

```
dbgcmd
```

```
srv*DownstreamStore*\\Server\Share
srv*\\Server\Share
```

不建议省略 **/s**[*Opts*] *SymbolPath* 参数，但如果省略，SymChk 将使用以下默认路径指向公共符号存储：

```
dbgcmd
```

```
srv*%SystemRoot%\symbols*https://msdl.microsoft.com/download/symbols
```

以下任意数量的选项都可以遵循 **/s**。 /s **和以下选项之间**不能有空格：

e
SymChk 将单独检查每个路径，而不是一次性检查所有路径。

u
下游存储将更新。 如果符号路径包括下游存储区，则符号存储将搜索符号文件。 只会更新 SymChk 正在检查的符号存储。

p
强制检查专用符号。 公共符号将被视为不匹配。 p 选项表示 **e** 和 **u**，不能与 s **一起使用。**

s
强制检查公共（拆分）符号。 私有符号将被视为不匹配。 该选项意味着 **e** 和 **u**，不能与 p **一起使用。**

r
展开指定路径中的所有非符号服务器元素，以便对路径进行深入搜索。 注意：此选项可能会生成调试器内不会发生的匹配项，因为它修改指定的符号路径。

可用选项 分为多个类。 每个选项类控制一组不同的功能。

输出选项。 可以指定以下任意数量的选项。 这些选项只能使用 **/o** 一次来缩写 -- 例如，**/oi /oe** 可以编写为 **/oie**。

展开表

选项	效果
/oe	输出将包括单个错误。 仅当使用 /q **时**，此选项才有用，因为如果未激活静默模式，则会自动显示各个错误。
/op	输出将列出每个传递的文件。 （默认情况下，SymChk 仅显示测试失败的文件。
/oi	输出将列出被忽略的每个文件。 （默认情况下，SymChk 仅显示测试失败的文件。
/od	输出将包括完整详细信息。 与 **/oe /op /oi** 相同。
/ot	输出将包括结果总计。 仅当使用 /q **时**，此选项才有用，因为如果未激活静默模式，将自动显示这些总计。
/ob	二进制文件的完整路径将包含在所有输出消息中。
/操作系统	符号的完整路径将包含在所有输出消息中。
/oc *Dir*	SymChk 将在目录 *Dir* 中创建一个传统符号树，其中包含选中的所有符号文件的列表。
/ov	SymChk 还将打印已检查二进制文件的版本信息。
/ol *文件*	除了发送到标准外的消息，还编写一个文件，其中包含所有二进制文件及其传递符号检查的逗号分隔列表。

DBG 文件选项。 这些选项控制 SymChk 如何检查 *.dbg* 符号文件。 只能指定以下选项之一。

展开表

选项	效果
/ds	SymChk 将验证 .dbg 信息是否已从可执行文件中剥离，并且仅显示在 .dbg 文件中，并且可执行文件是否指向 .dbg 文件。 如果程序没有 .dbg 符号文件生成，则此选项不起作用。 这是默认情况。
/德	SymChk 将验证 .dbg 信息是否未从可执行文件中剥离，并且可执行文件未指向 .dbg 文件。 如

选项	效果
	果程序没有 .dbg 符号文件生成，则此选项不起作用。
/dn	SymChk 将验证映像中不存在 .dbg 信息，并且该映像未指向 .dbg 文件。

PDB 文件选项。 这些选项控制 SymChk 检查 .pdb 符号文件的方式。 只能指定以下选项之一。

展开表

选项	效果
/pf	SymChk 不检查 .pdb 文件的内容 - 它只是验证文件是否存在并匹配二进制文件。 这是默认情况。
/ps	SymChk 将验证 .pdb 文件是否已去除源行、数据类型和全局信息。
/铂	SymChk 将验证 .pdb 文件是否包含数据类型信息。

筛选选项。 这些选项控制在 SymChk 检查进程或转储文件时如何执行模块筛选。 只能指定以下选项之一。

展开表

选项	效果
/fm *模块*	SymChk 将仅检查与指定模块关联的转储文件或进程。 *模块* 必须包含完整文件名，但不得包含目录路径的任何部分。

符号检查选项。 可以指定以下任意数量的选项。

展开表

选项	效果
/cs	SymChk 不会验证 CodeView 数据是否存在。 （默认情况下，验证 CodeView 数据是否存在。
/cc	当 SymChk 检查修补程序 CAB 文件时，它不会在 cab 中查找符号。 （默认情况下，SymChk 将在 cab 中以及提供的符号路径中查找符号。

选项	效果
/ea *文件*	SymChk 不会验证指定文件中列出的程序的符号。 这允许你否决某些程序，否则将被验证。 *文件* 必须指定一个文件（按相对文件、绝对路径或 UNC 路径），但不包含通配符）;如果该文件包含空格，则应用引号引起来。 在 File 中，每行都指示程序文件（按相对路径、绝对路径或 UNC 路径）;不允许使用通配符。 如果此文件中的一行包含空格，则应用引号括起来。 此文件中的分号是注释字符 -- 将忽略分号和行尾之间的所有内容。 如果使用符号服务器，这些程序的符号将不会复制到下游存储。
/ee *文件*	禁止显示指定文件中所列程序的错误消息。 “成功”和“忽略”消息将照常显示，符号文件将照常复制到下游存储。 File 的格式*及其内容的格式与 /ea 文件的格式*相同。

其他信息

有关 SymChk 的详细信息，请参阅 使用 SymChk。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用 WinDbg 分析故障转储文件

项目 • 2024/02/06

你可以使用 WinDbg 和其他 Windows 调试器分析故障转储文件。

> **备注**
>
> 此内容适用于开发人员。 如果你是 Microsoft 客户并且计算机显示蓝屏错误代码，请参阅 **蓝屏错误疑难解答** 。

本部分包括以下文章：

内核模式转储文件

用户模式转储文件

从转储文件中提取信息

在内存转储中包含分页文件的 CAB 文件

内核模式转储文件

项目 • 2024/02/07

发生内核模式错误时，Microsoft Windows 默认显示蓝屏，检查数据检查。 但是，可以从多种替代行为中进行选择：

- 可以联系内核调试器，如 WinDbg 或 KD。
- 系统可以自动重新启动。
- 可以写入内存转储文件。
- 可以写入内存转储文件，系统随后可以自动重新启动。

创建和分析内核模式内存转储文件。

有三种不同 种类的内核模式转储文件。 但是，任何转储文件都不会像附加到已失败的系统的实时内核调试器那样有用且通用。

- 创建内核模式转储文件
- 启用 Kernel-Mode 转储文件
- 分析内核模式转储文件
- 生成内核或完整的故障转储

内核模式转储文件的种类

项目 • 2023/10/26

内核模式故障转储文件有五个设置：

完整内存转储

内核内存转储

小规模内存转储

自动内存转储

活动内存转储

这些转储文件之间的差异是大小之一。 *完整内存转储*最大，包含的信息最多，包括一些 User-Mode 内存。 *活动内存转储*稍小一些，但包含用于大多数用途的类似信息。 *内核内存转储*较小，通常省略 User-Mode 内存，*小内存转储*大小仅为 64 KB。

如果选择“*自动内存转储*”，则转储文件与内核内存转储相同，但 Windows 在设置系统分页文件的大小方面具有更大的灵活性。

较大的文件的优点是，由于这些文件包含更多信息，因此更有可能帮助你找到崩溃的原因。

较小的文件的优点是更小，写入速度更快。 速度通常很有价值：如果运行的是服务器，你可能希望服务器在崩溃后尽快重新启动，并且直到写入转储文件后才会重新启动。

创建完整内存转储或内核内存转储后，可以从较大的转储文件创建小型内存转储文件。 有关详细信息，请参阅 **.dump (创建转储文件)** 命令。

注意 可以通过分析内核模式转储文件来获取大量信息。 但是，没有内核模式转储文件可以提供与直接使用内核调试器直接调试崩溃一样多的信息。

另请参阅

内核模式转储文件

启用内核模式转储文件

完整内存转储

项目 • 2023/04/14

*完整内存转储*是最大的内核模式转储文件。 此文件包括 Windows 使用的所有物理内存。 默认情况下，完整内存转储不包括平台固件使用的物理内存。

从 Windows 8 开始，可以注册在完整内存转储期间调用的 *BugCheckAddPagesCallback* 例程。 *BugCheckAddPagesCallback* 例程可以指定要添加到转储文件中的特定于驱动程序的数据。 例如，此附加数据可能包括未映射到虚拟内存中的系统地址范围，但包含可以帮助你调试驱动程序的信息的物理页。 *BugCheckAddPagesCallback* 例程可能会将未映射或映射到虚拟内存中用户模式地址的任何驱动程序拥有的物理页添加到转储文件。

此转储文件要求启动驱动器上的页面文件至少与main系统内存一样大;它应该能够保存大小等于整个 RAM 加 1 MB 的文件。

默认情况下，完整内存转储文件写入 %SystemRoot%\Memory.dmp。

如果出现第二个 bug 检查，并且创建了另一个完整内存转储 (或内核内存转储)，则会覆盖上一个文件。

另请参阅

内核模式转储文件的种类

内核内存转储

项目 • 2023/04/14

*内核内存转储*包含崩溃时内核使用的所有内存。

此类转储文件明显小于完整内存转储。 通常，转储文件的大小约为系统上物理内存的三分之一。 此数量会有很大差异，具体取决于你的情况。

此转储文件不包括未分配的内存或任何分配给用户模式应用程序的内存。 它仅包括分配给 Windows 内核和硬件抽象层 (HAL) 的内存，以及分配给内核模式驱动程序和其他内核模式程序的内存。

在大多数情况下，此故障转储最有用。 它明显小于完整内存转储，但它只省略了那些不太可能参与崩溃的内存部分。

由于此类转储文件不包含在崩溃时驻留在内存中的任何用户模式可执行文件的图像，因此，如果这些可执行文件变得重要，可能还需要设置可执行映像路径。

默认情况下，内核内存转储文件写入 %SystemRoot%\Memory.dmp。

如果出现第二个 bug 检查，并且创建了另一个内核内存转储 (或完整内存转储)，则会覆盖上一个文件。

若要在调试内核内存转储时禁止显示缺少页的错误消息，请使用 .ignore_missing_pages 命令。

另请参阅

内核模式转储文件的种类

小规模内存转储

项目 • 2023/04/14

*小型内存转储*比其他两种内核模式故障转储文件小得多。 它的大小正好为 64 KB，只需要启动驱动器上的 64 KB 页文件空间。

此转储文件包括以下内容：

- 该 bug 检查消息和参数以及其他蓝屏数据。
- 已崩溃的处理器 (PRCB) 的处理器上下文。
- 进程信息和内核上下文 (崩溃进程的 EPROCESS) 。
- 崩溃线程的线程信息和内核上下文 (ETHREAD) 。
- 崩溃线程的内核模式调用堆栈。 如果此长度超过 16 KB，则仅包含最顶层的 16 KB。
- 已加载驱动程序的列表。

在 Windows XP 和更高版本的 Windows 中，还包括以下项：

- 已加载模块和卸载模块的列表。
- 调试器数据块。 它包含有关系统的基本调试信息。
- Windows 标识为在调试失败时有用的任何其他内存页。 这包括发生崩溃时寄存器指向的数据页，以及故障组件专门请求的其他页。
- (Windows Server 2003 及更高版本) Windows SKU，例如，“Professional”或“Server”。

当空间受到极大限制时，这种转储文件可能很有用。 但是，由于包含的信息有限，因此分析此文件可能无法发现不是直接由线程在崩溃时执行的错误。

由于此类转储文件不包含在崩溃时驻留在内存中的任何可执行文件的映像，因此如果这些可执行文件变得重要，可能还需要设置可执行映像路径。

如果出现第二个 bug 检查，并且创建了第二个小型内存转储文件，则将保留上一个文件。 将为每个附加文件指定一个不同的名称，其中包含在文件名中编码的崩溃日期。 例如，mini022900-01.dmp 是 2000 年 2 月 29 日生成的第一个内存转储文件。 所有小型内存转储文件的列表保存在 %SystemRoot%\Minidump 目录中。

另请参阅

内核模式转储文件的种类

自动内存转储

项目 • 2024/03/01

*自动内存转储*包含与内核内存转储相同的信息。 两者之间的差异不在于转储文件本身，而是 Windows 设置系统分页文件的大小的方式。

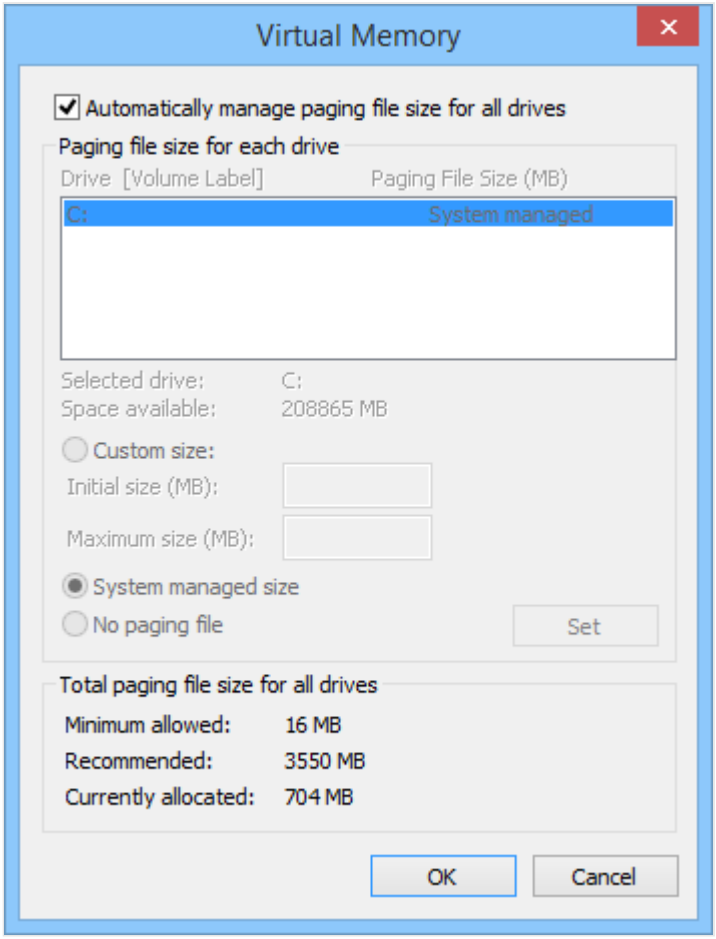
如果系统分页文件大小设置为 **系统托管大小**，并且内核模式故障转储设置为**自动内存转储**，则 Windows 可以将分页文件的大小设置为小于 RAM 的大小。 在这种情况下，Windows 会将分页文件的大小设置地足够大，以确保大部分时间都可以捕获内核内存转储。

如果计算机崩溃并且分页文件不够大，无法捕获内核内存转储，Windows 会将分页文件的大小至少增加到 RAM 的大小。 此事件的时间记录在注册表中：

HKLM\SYSTEM\CurrentControlSet\Control\CrashControl\LastCrashTime

增加的分页文件大小会保持 4 周，然后返回到较小的大小。 如果要在 4 周前返回到较小的分页文件，可以删除注册表项。

若要查看分页文件设置，请前往**控制面板**>**系统和安全**>**系统**>**高级系统设置**。 在**性能**下选择**设置**。 在**高级**选项卡上的**虚拟内存**下，选择**更改**。 在“虚拟内存”对话框中，可以看到分页文件设置。



自动内存转储文件默认写入 %SystemRoot%\Memory.dmp。

自动内存转储可在 Windows 8 及更高版本中使用。

请注意，若要在调试自动内存转储时禁止显示缺少的页面错误消息，请使用 .ignore_missing_pages 命令。

另请参阅

内核模式转储文件的种类

内核模式转储文件

创建内核模式转储文件

活动内存转储

项目 • 2024/03/01

*活动内存转储*与完全内存转储类似，但它会过滤掉可能与主机故障诊断无关的页面。 由于这种过滤操作，它通常比完整的内存转储要小得多。

此转储文件包含分配给用户模式应用程序的任何内存。 它还包括分配给 Windows 内核和硬件抽象层 (HAL) 的内存，以及分配给内核模式驱动程序和其他内核模式程序的内存。 转储包括映射到内核或用户空中对调试有用的活动页面，以及选定的页面文件支持的转换、备用和修改页面，例如使用 VirtualAlloc 或页面文件支持的分区分配的内存。 活动转储不包括空闲列表和清零列表上的页面、文件缓存、来宾 VM 页面以及在调试期间不太可能有用的各种其他类型的内存。

当 Windows 托管虚拟机 (VM) 时，活动内存转储特别有用。 在进行完整的内存转储时，每个 VM 的内容都会包括在内。 当多个 VM 运行时，这可能会导致主机系统上使用大量内存。 很多时候，感兴趣的代码活动位于父主机操作系统中，而不是子 VM。 活动内存转储会过滤与所有子 VM 关联的内存。

活动内存转储文件默认写入 %SystemRoot%\Memory.dmp。

活动内存转储可在 Windows 10 及更高版本中使用。

请注意，若要在调试活动内存转储时禁止显示缺少的页面错误消息，请使用 .ignore_missing_pages 命令。

另请参阅

内核模式转储文件的种类

创建内核模式转储文件

项目 • 2023/06/15

可通过三种方式创建内核模式转储文件：

1. 可以从控制面板启用转储文件，然后系统可能会自行崩溃。
2. 可以从控制面板启用转储文件，然后强制系统崩溃。
3. 可以使用调试器创建转储文件，而不会使系统崩溃。

本节包括：

启用内核模式转储文件

强制系统崩溃

未发生系统崩溃的情况下创建转储文件

验证是否已创建内核模式转储文件

使用 NMI 开关

还可以使用 NMI 开关创建故障转储文件。 请与硬件供应商联系，确定计算机是否具有此开关。

本文档不介绍 NMI 开关的用法。

启用 Kernel-Mode 转储文件

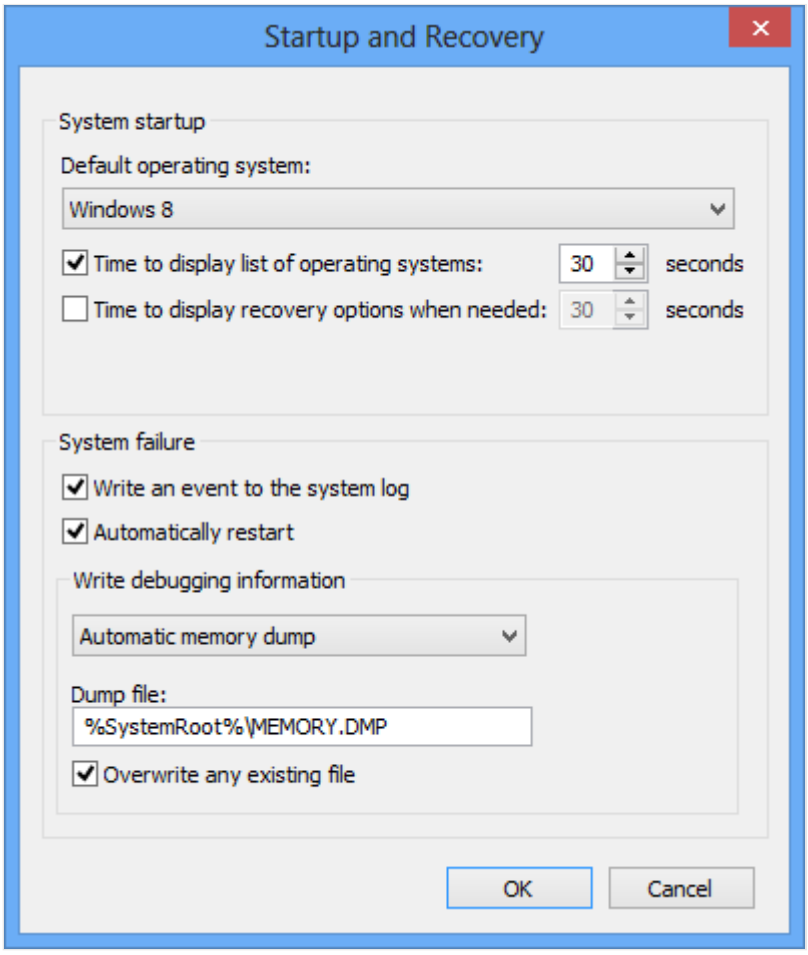
项目 • 2023/12/21

在系统崩溃期间，Windows 故障转储设置确定是否将创建转储文件，如果是，则确定转储文件的大小。

Windows 控制面板控制内核模式故障转储设置。 只有系统管理员才能修改这些设置。

若要更改这些设置，请转到**控制面板>系统和安全>系统**。 选择“ **高级系统设置**”。 在 “**启动和恢复**”下，选择 “**设置**”。

你将看到以下对话框：



在 “**写入调试信息**”下，可以指定内核模式转储文件设置。 对于任何给定的故障，只能创建一个转储文件。 有关不同 转储文件 设置的说明，请参阅各种 Kernel-Mode 转储文件。

还可以选择或取消选择“ **将事件写入系统日志**” 和“ **自动重启** ”选项。

选择的设置将应用于系统崩溃创建的任何内核模式转储文件，无论系统崩溃是意外的还是由调试器引起的。 有关导致故意崩溃的详细信息，请参阅 强制系统 崩溃。

但是，这些设置不会影响 .dump 命令创建的转储文件。 有关使用此命令的详细信息，请参阅创建没有系统崩溃的转储文件 。

另请参阅

内核模式转储文件

内核模式转储文件的种类

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

Windows driver documentation 反馈

Windows driver documentation 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

强制系统崩溃

项目 • 2023/09/13

启用内核模式转储文件后，大多数系统崩溃应会导致写入崩溃文件并显示蓝屏。

但是，有时系统会冻结，而不会实际启动内核崩溃。 此类冻结的可能症状包括：

- 鼠标指针移动，但无法执行任何操作。
- 所有视频都已冻结，鼠标指针不会移动，但分页继续。
- 对鼠标或键盘没有任何响应，也不使用磁盘。

如果有经验丰富的调试技术人员，他们可以附加内核调试器并分析问题。 有关出现这种情况时要查找的一些提示，请参阅 调试停止的系统。

但是，如果没有技术人员，你可能希望创建一个内核模式转储文件并将其发送给非现场技术人员。 此转储文件可用于分析错误的原因。

有三种方法故意导致系统崩溃：

通过调试器强制系统崩溃

通过键盘强制系统崩溃

使用电源按钮强制系统崩溃

通过调试器强制系统崩溃

项目 • 2023/10/26

如果 KD 或 WinDbg 正在执行内核模式调试，则可能会强制发生系统崩溃。 这是通过在命令提示符处输入 **.crash (强制系统崩溃)** 命令来完成的。 (如果目标计算机未立即崩溃，请使用 **g (Go)** 命令执行此操作。)

发出此命令时，系统会调用 **KeBugCheck** 并发出 **bug 检查 0xE2** (MANUALLY_INITIATED_CRASH) 。 除非已禁用故障转储，否则此时会写入故障转储文件。

写入故障转储文件后，主计算机上的内核调试器将收到警报，并可用于主动调试崩溃的目标。

通过键盘强制系统崩溃

项目 • 2024/09/25

以下类型的键盘可能会直接导致系统崩溃：

- **i8042prt 端口上连接的 PS/2 键盘** Windows 2000 及更高版本的 Windows 操作系统中提供此功能。
- **USB 键盘**
  Windows Vista 及更高版本的 Windows 操作系统中提供此功能。
- **Hyper-V 键盘**
  Windows 10 版本 1903 及更高版本的 Windows 操作系统中提供此功能。

配置

配置以下设置以使用键盘实现系统崩溃：

1. 如果希望编写故障转储文件，则必须启用此类转储文件。 选择路径和文件名，然后选择转储文件的大小。 有关详细信息，请参阅启用内核模式转储文件。
2. 对于 PS/2 键盘，你必须在注册表中启用键盘启动的崩溃。 在注册表项 `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\i8042prt\Parameters` 中，创建一个名为 `CrashOnCtrlScroll` 的值，并将其设置为等于 0x01 的 `REG_DWORD` 值。
3. 对于 USB 键盘，你必须在注册表中启用键盘启动的崩溃。 在注册表项 `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\kbdhid\Parameters` 中，创建一个名为 `CrashOnCtrlScroll` 的值，并将其设置为等于 0x01 的 `REG_DWORD` 值。
4. 对于 Hyper-V 键盘，你必须在注册表中启用键盘启动的崩溃。 在注册表项 `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\hyperkbd\Parameters` 中，创建一个名为 `CrashOnCtrlScroll` 的值，并将其设置为等于 0x01 的 `REG_DWORD` 值。

某些笔记本电脑将 PS/2 驱动程序用于内置键盘，并且还支持外部 HID 键盘。 对于这些系统，请考虑同时创建 USB 和 PS/2 注册表项以允许使用任一键盘。

你必须重新启动系统才能使这些设置生效。

重启完成后，可以使用以下热键序列启动键盘崩溃：按住最右侧的 Ctrl 键，然后按 Scroll Lock 键两次。

然后，系统会调用 `KeBugCheck` 并发出 错误检查：0xE2 MANUALLY_INITIATED_CRASH。 除非禁用了故障转储，否则会写入故障转储文件。

如果内核调试器附加到了崩溃的计算机，则计算机将在写入故障转储文件后进入内核调试器。

定义备用键盘快捷方式以强制系统从键盘崩溃

你可以在以下注册表子项下面配置备用值，以便键盘快捷序列生成内存转储文件：

- 对于 PS/2 键盘：

  `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\i8042prt\crashdump`

- 对于 USB 键盘：

  `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\kbdhid\crashdump`

- 对于 Hyper-V 键盘：

  `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\hyperkbd\crashdump`

你必须在这些子项下面创建以下注册表 `REG_DWORD` 值：

Dump1Keys

此 `Dump1Keys` 注册表值是要使用的第一个热键的位映射。 例如，可以将第一个热键设置为最左侧的 SHIFT 键，而不是使用最右侧的 CTRL 键来启动热键序列。

下表描述了第一个热键的十六进制值。

展开表

值	键盘快捷序列中使用的第一个键
0x01	最右侧的 SHIFT 键
0x02	最右侧的 CTRL 键
0x04	最右侧的 ALT 键
0x10	最左侧的 SHIFT 键
0x20	最左侧的 CTRL 键
0x40	最左侧的 ALT 键

你可以给 `Dump1Keys` 分配一个值，该值启用一个或多个键作为键盘快捷序列中使用的第一个键。 例如，给 `Dump1Keys` 分配一个 0x11 值，以将最右侧和最左侧的 SHIFT 键都定

义为键盘快捷序列中的第一个键。

Dump2Key

此 `Dump2Key` 注册表值是目标计算机的键盘布局扫描代码表中的索引。 请参阅驱动程序中的实际表：

```
C++

const UCHAR keyToScanTbl[134] = {
        0x00,0x29,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,
        0x0A,0x0B,0x0C,0x0D,0x7D,0x0E,0x0F,0x10,0x11,0x12,
        0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,0x00,
        0x3A,0x1E,0x1F,0x20,0x21,0x22,0x23,0x24,0x25,0x26,
        0x27,0x28,0x2B,0x1C,0x2A,0x00,0x2C,0x2D,0x2E,0x2F,
        0x30,0x31,0x32,0x33,0x34,0x35,0x73,0x36,0x1D,0x00,
        0x38,0x39,0xB8,0x00,0x9D,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0xD2,0xD3,0x00,0x00,0xCB,
        0xC7,0xCF,0x00,0xC8,0xD0,0xC9,0xD1,0x00,0x00,0xCD,
        0x45,0x47,0x4B,0x4F,0x00,0xB5,0x48,0x4C,0x50,0x52,
        0x37,0x49,0x4D,0x51,0x53,0x4A,0x4E,0x00,0x9C,0x00,
        0x01,0x00,0x3B,0x3C,0x3D,0x3E,0x3F,0x40,0x41,0x42,
        0x43,0x44,0x57,0x58,0x00,0x46,0x00,0x00,0x00,0x00,
        0x00,0x7B,0x79,0x70 };
```

索引 124 (sysreq) 是一种特殊情况，因为 84 键键盘具有不同的扫描代码。

如果定义备用键盘快捷方式以强制系统从 USB 或 PS/2 键盘崩溃，则必须将 `CrashOnCtrlScroll` 注册表值设置为 0，或者将其从注册表中删除。

示例：在此方案中，笔记本电脑使用 PS2 键盘驱动程序，并且附加了外部 HID 键盘。 设置这两个值可提供从任一键盘触发手动系统崩溃的功能。 设置以下注册表项时，可以按住最右侧的控制键并按两次空格键来强制手动系统崩溃。

```
reg

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\i8042prt\crashdump]
"Dump1Keys"=dword:00000002
"Dump2Key"=dword:0000003d

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\kbdhid\crashdump]
"Dump1Keys"=dword:00000002
"Dump2Key"=dword:0000003d
```

限制

系统冻结致使键盘快捷序列不起作用是有可能发生的，但很少会出现这种情况。 即使在 **CTRL+ALT+DELETE** 不起作用的许多情况下，也可以使用键盘快捷方式序列启动崩溃。

如果计算机在高中断请求级别 (IRQL) 停止响应，则从键盘强制系统崩溃不起作用。 之所以存在此限制，是因为 **Kbdhid.sys** 驱动程序允许内存转储进程运行，其运行 IRQL 低于 **i8042prt.sys** 驱动程序。

另请参阅

错误检查 0xE2：MANUALLY_INITIATED_CRASH

!analyze -v

使用 WinDbg 分析内核模式转储文件

错误检查 0x161：LIVE_SYSTEM_DUMP

生成内核或完整的故障转储

内核模式转储文件的种类。

ⓘ **注意：**作者在 AI 的帮助下创作了此文章。 了解详细信息

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用电源按钮强制系统崩溃

项目 • 2023/12/12

bug 检查0x1C8：当在显示的注册表项中设置以下注册表值时，可以通过长按电源按钮来强制 MANUALLY_INITIATED_POWER_BUTTON_HOLD手动系统崩溃：

reg

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power]
"PowerButtonBugcheck"=dword:00000001
```

若要启用手动系统崩溃电源按钮保持功能，请从提升的命令提示符运行以下命令：

dos

```
REG ADD HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power /v PowerButtonBugCheck /t
REG_DWORD /d 0x1 /f
```

如果此注册表 *值不存在*， 则必须重新启动系统才能使此更改生效。

如果此注册表值 *确实* 存在并且值已更改，则 *无需* 重新启动系统即可使更改生效。

当电源按钮按住 7 秒，但在 UEFI 重置发生 10 秒之前释放时，会出现 bug 检查。

触发该功能时，不会显示常规 bug 检查蓝屏或绿屏。 相反，你将看到一个显示 *"请释放电源"按钮的黑屏。我们只需要几秒钟的时间关闭* 和完成百分比指标。

看到此屏幕后，释放电源按钮非常重要。 如果继续按住电源按钮较长时间，可能会触发固件硬关闭超时，通常为 10 秒左右。 如果发生这种情况，则转储可能无法完成，并且可能已损坏。

在某些系统（如 ARM64 笔记本电脑）上，固件计时器较短，因此必须在 7 秒后立即释放电源按钮，以防止硬断电中断转储的完成。

此功能在 Windows 10 1809 /Windows Server 2019 及更新版本中可用。

Bug 检查0x11C8：MANUALLY_INITIATED_POWER_BUTTON_HOLD_LIVE_DUMP

还可以使用长电源按钮保持 (LPBH) 创建实时转储，而不是重启电脑。 有关实时转储的一般信息，请参阅内核实时转储代码参考。

若要启用电源按钮保持实时转储功能，请设置以下注册表项下的 *PowerButtonLiveDump* 值：

reg

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power]
"PowerButtonLiveDump"=dword:00000001
```

若要启用电源按钮实时转储功能，请从提升的命令提示符运行以下命令：

```dos
REG ADD HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power /v PowerButtonLiveDump /t REG_DWORD /d 0x1 /f
```

如果此 *注册表项不存在*， 则必须重新启动系统才能使此更改生效。

如果此注册表项 *确实存在*且值已更改，则 *无需* 重新启动系统即可使更改生效。

此功能在 Windows 10 版本 22000 及更新版本中可用。

PowerButtonLiveDump 自定义

可以按如下所示自定义 PowerButtonLiveDump 注册表值。

展开表

名称	长度（以位为单位）	从开始偏移	含义	默认值
已启用	1	0	设置后，将启用电源按钮实时转储。	不可用
UserPages	1	1	设置后， 将请求捕获实时转储中的用户模式内存。	0
HypervisorPages	1	2	设置后， 将请求捕获实时转储中的虚拟机监控程序页面。	0
保留	1	3	预留	N/A
TimeoutInSec	4	4	指定捕获实时转储文件的超时值（以秒为单位）。 只有 2 到 6 (包括 2 和 6) 的值才有效。 如果指定了任何其他值，将使用默认超时值 5 秒。	5
预留	24	8	预留	N/A

例如，从提升的命令提示符运行以下命令将启用包含 UserPages 和 TimeoutInSec 为 2 的 PowerButtonLiveDump 功能。 当电源按钮按住 2 秒时，内核将请求捕获完整实时转储文件。

```dos
REG ADD HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power /v PowerButtonLiveDump /t REG_DWORD /d 0x23 /f
```

PowerButtonBugcheck 和 PowerButtonLiveDump 优先级

如果在同一设备上同时配置并启用了 PowerButtonBugcheck 和 PowerButtonLiveDump，则调用时将发生 PowerButtonBugcheck，而 PowerButtonLiveDump 将不会发生。

长电源按钮保持 (LPBH) 的设备支持

若要支持长电源按钮保持，设备需要：

- 基于 常规用途 I/O (GPIO) 电源按钮
- 用于将电源事件路由到 Windows Power Manager 的固件
- 在注册表中启用的 bug 检查功能

另请参阅

Bug 检查0x11C8：MANUALLY_INITIATED_POWER_BUTTON_HOLD_LIVE_DUMP

Show 内部 - bugcheck 0x1C8 MANUALLY_INITIATED_POWER_BUTTON_HOLD

ACPI 按钮设备

在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。 有关详细信息，请参阅参与者指南。

Windows driver documentation 反馈

Windows driver documentation 是一个开放源代码项目。 选择一个链接以提供反馈：

提出文档问题

提供产品反馈

未发生系统崩溃的情况下创建转储文件

项目 • 2023/06/15

如果 KD 或 WinDbg 正在执行内核模式调试，则可能会导致写入内核模式转储文件，而不会使目标计算机崩溃。

此转储文件可以是完整内存转储或小型内存转储。 控制面板设置与此操作无关。

而由系统崩溃导致的转储文件将写入已崩溃的计算机，而此转储文件将写入主计算机。

有关详细信息，请参阅 **.dump (创建转储文件)** 命令。

验证是否已创建内核模式转储文件

项目 • 2023/06/14

如果计算机已中断调试器，但不确定故障转储文件是否已成功写入，请执行以下命令：

```
dbgcmd
```

```
dd nt!IopFinalCrashDumpStatus L1
```

这将显示 **IopFinalCrashDumpStatus** 变量的值。

如果此值等于零，则表示该过程成功。 如果等于 -1 (0xFFFFFFFF)，则表示转储过程尚未启动。

任何其他值都是指示转储过程中出错的状态代码。

分析内核模式转储文件

项目 • 2024/03/01

本节包括：

使用 KD 分析内核模式转储文件

使用 WinDbg 分析内核模式转储文件

安装符号文件

无论您使用哪种工具，都需要访问生成转储文件的 Windows 版本的符号文件。 您选择用于分析转储文件的调试器将使用这些文件。 有关使用符号服务器的信息，请参阅 Microsoft 公共符号。

DumpExam

DumpExam 工具已过时。 故障转储文件分析中不再需要它。

使用 KD 分析内核模式转储文件

项目 • 2024/07/08

内核模式内存转储文件可由 KD 分析。 创建转储文件的处理器或 Windows 版本不必与运行 KD 的平台一致。

启动 KD

若要分析转储文件，请使用 **-z** 命令行选项启动 KD：

kd **-y** *SymbolPath* **-i** *ImagePath* **-z** *DumpFileName*

-v 选项（详细模式）也很有用。 有关选项的完整列表，请参阅 **KD 命令行选项**。

也可以使用 .opendump （打开转储文件）命令在调试器运行之后打开转储文件，后跟 **g (Go)**。

可以同时调试多个转储文件。 这可以通过在命令行上加入多个 **-z** 开关（每个开关后跟一个不同的文件名），或使用 **.opendump** 添加额外的转储文件作为调试器目标来完成。 有关如何控制多目标会话的信息，请参阅调试多个目标。

转储文件通常以扩展名.dmp或 .mdmp 结尾。 可以将网络共享或通用命名约定 (UNC) 文件名用于内存转储文件。

将转储文件打包到 CAB 文件中也很常见。 如果在 **-z** 选项后指定文件名（包括 .cab 扩展名），或者作为 **.opendump**命令的参数，调试器可以直接从 CAB 读取转储文件。 但是，如果单个 CAB 中存储了多个转储文件，调试器将只能读取其中一个。 调试器不会从 CAB 读取任何其他文件，即使它们是符号文件或与转储文件关联的其他文件。

分析转储文件

如果要分析内核内存转储或小型内存转储，则可能需要将可执行映像路径设置为指向在崩溃时可能已加载到内存中的任何可执行文件。

转储文件分析类似于实时调试会话的分析。 有关哪些命令可用于在内核模式下调试转储文件的详细信息，请参阅调试器命令参考章节。

在大多数情况下，应首先使用 !analyze。 此扩展命令对转储文件执行自动分析，通常会产生大量有用的信息。

.bugcheck（显示 Bug 检查数据） 显示 Bug 检查代码及其参数。 有关特定错误的信息，请在 Bug 检查代码参考中查找此 Bug 检查。

以下调试器扩展对于分析内核模式故障转储特别有用：

!lm

!kdext*.locks

!memusage

!vm

!errlog

!process 0 0

!process 0 7

有关可用于从转储文件读取特定类型信息的技术，请参阅 从转储文件中提取信息。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

使用 WinDbg 分析内核模式转储文件

项目 • 2024/03/08

可以使用 WinDbg 分析内核模式内存转储文件。

启动 WinDbg

转储文件通常以扩展名 *.dmp* 或 *.mdmp* 结尾。 可以将网络共享或通用命名约定文件名用于内存转储文件。 用于创建转储文件的处理器或 Windows 版本不需要与运行 KD 的平台匹配。

若要分析转储文件，请启动 WinDbg 并将 **-z** 命令行选项包括在内：

控制台

```
windbg -y <SymbolPath> -i <ImagePath> -z <DumpFileName>
```

-v 选项（详细模式）也很有用。 有关选项的完整列表，请参阅 WinDbg 命令行选项。

如果 WinDbg 已在休眠模式下运行，请通过选择 **File | Open Crash Dump** 菜单命令或按 **Ctrl+D** 打开故障转储。 出现**打开故障转储**对话框时，在**文件名中**输入故障转储文件的完整路径和名称，或使用对话框选择路径和文件名。 指定文件后，选择**打开**。

或在调试器运行后使用 .opendump （打开转储文件）命令打开转储文件，后跟 g (Go) 命令。

您可以同时调试多个转储文件。 在命令行中包含多个 **-z** 开关，每个开关后跟不同的文件名，或运行 .opendump 以添加其他转储文件作为调试器目标。 有关如何控制多目标会话的详细信息，请参阅调试多个目标。

转储文件可以打包到 CAB 文件中。 如果在 **-z** 选项后指定文件名（包括 *.cab* 文件扩展名），或将其作为 .opendump 命令的参数，调试器将直接读取转储文件。

如果单个 CAB 文件中存储了多个转储文件，调试器将只读取其中一个转储文件。 即使存在符号文件或与转储文件关联的其他文件，调试器也不会从 CAB 读取任何其他文件。

分析转储文件

若要分析内核内存转储或小型内存转储，可能需要设置可执行映像路径，指向崩溃期间内存中的可执行文件。

转储文件分析类似于实时调试会话的分析。 有关可用于在内核模式下调试转储文件的命令详情，请参阅调试器命令参考章节。

在大多数情况下，使用 !analyze 开始。 此扩展命令对转储文件执行自动分析，这通常提供有用的信息。

.bugcheck（显示 Bug 检查数据）命令显示 Bug 检查代码及其参数。 有关特定错误的信息，请参阅 Bug 检查代码参考。

以下调试器扩展对于分析内核模式故障转储特别有用：

- lm
- !kdext*.locks
- !memusage
- !vm
- !errlog
- !process 0 0
- !process 0 7

有关从转储文件读取特定类型信息的技术，请参阅 从转储文件中提取信息。

另请参阅

开始使用 WinDbg（内核模式）

调试器操作

调试技术

下载并安装 WinDbg Windows 调试器

!analyze

用户模式转储文件

项目 • 2024/11/25

本文概述了用户模式转储文件，以及如何使用它们来帮助解决错误和崩溃。

有关分析转储文件的信息，请参阅分析用户模式转储文件。

用户模式转储文件的类型

可以使用多种类型的用户模式崩溃转储文件。 不同类型的转储文件分为两类：

- 完整用户模式转储
- 小型转储

通过分析转储文件，可以获得大量信息。 但是，任何转储文件都无法提供通过使用调试器调试崩溃所获取的信息量。

完整用户模式转储

*完整用户模式转储*是基本用户模式转储文件。 完整用户模式转储文件包括：

- 进程的整个内存空间。
- 程序的可执行映像。
- 句柄表。
- 其他有助于调试器重建转储发生时正在使用的内存的信息。

可以将一个完整用户模式转储文件*收缩*为一个迷你转储中。 若要收缩完整用户模式转储文件，请先在调试器中加载转储文件。 然后，使用 .dump（创建转储文件）命令以小型转储格式保存新的转储文件。

尽管名称如此，但最大的小型转储文件包含的信息比完整用户模式转储文件更多。 例如，`.dump /mf` 和 `.dump /ma` 命令创建的文件比 `.dump /f` 命令更大、更完整。

在用户模式下，`.dump /m`[*MiniOptions*] 通常是最佳选择。 使用此开关创建的转储文件的大小可能从非常小到非常大不等。 通过指定正确的 *MiniOptions* 开关，可以准确控制包含的信息。

小型转储

小型转储文件的大小和内容因转储的程序以及执行转储的应用程序和所选的选项而异。 有时，一个小型转储文件的大小适中，包含了完整的内存和句柄表。 其他情况下，小型

转储文件要小得多。 例如，小型转储文件可能仅包含有关单个线程的信息，也可能只包含堆栈中引用的模块的信息。

术语*小型转储*具有误导性，因为最大的小型转储文件包含的信息比完整用户模式转储文件更多。 例如，`.dump /mf` 或 `.dump /ma` 创建的文件比 `.dump /f` 更大、更完整。 因此，建议使用 `.dump /m`[*MiniOptions*] 而不是 `.dump /f` 来创建所有用户模式转储文件。

如果使用调试器创建小型转储文件，则可以选择要包含的信息。 `.dump /m` 命令包括有关组成目标进程的已加载模块的基本信息、线程信息和堆栈信息。 可以使用下表中所述的任何开关选项修改基本命令：

展开表

`.dump` 选项	对转储文件的影响
`/ma`	创建包含所有可选添加的小型转储。 `/ma` 选项等同于 `/mfFhut`。 它会在小型转储中添加完整内存数据、处理数据、未加载模块信息、基本内存信息和线程时间信息。
`/mf`	将完整内存数据添加到小型转储。 目标应用程序拥有的所有可访问提交页面都将包括在内。
`/mF`	将所有基本内存信息添加到小型转储。 此开关会向小型转储添加包含所有基本内存信息的流，而不仅仅是有效内存的相关信息。 调试器可以使用这些信息，在调试小型转储时重建进程的完整虚拟内存布局。
`/mh`	将与目标应用程序相关的句柄数据添加到小型转储。
`/mu`	将未加载的模块信息添加到小型转储。 此选项仅适用于 Windows Server 2003 和更高版本的 Windows。
`/mt`	将更多线程信息添加到小型转储。 线程信息包括线程时间，在调试小型转储时，可以通过 .ttime（显示线程时间）来显示线程时间。
`/mi`	将辅助内存添加到小型转储。 *辅助内存*是堆栈或后备存储上指针引用的任何内存，以及围绕此地址的一小块区域。
`/mp`	将进程环境块和线程环境块数据添加到小型转储。 如果需要访问有关应用程序进程和线程的 Windows 系统信息，这些信息非常有用。
`/mw`	将所有提交的读写专用页面添加到小型转储。
`/md`	将可执行映像中的所有读写数据段添加到小型转储。
`/mc`	在映像中添加代码节。
`/mr`	从堆栈的小型转储部分删除并存储不用于重新创建堆栈跟踪的内存。 局部变量和其他数据类型值也会被删除。 此选项不会使小型转储变小（未使用的内存部分只是被清零），但如果要保护其他应用程序的隐私，它会很有用。

`.dump` 选项	对转储文件的影响
`/mR`	从小型转储中删除完整的模块路径。 仅包含模块*名称*。 如果要保护用户目录结构的隐私，此选项非常有用。

可以组合使用这些开关选项。 例如，使用命令 `.dump /mfiu` 创建一个中等大小的小型转储，其中包含已卸载的内存和辅助内存。 使用命令 `.dump /mrR` 创建一个小型转储，用于删除某些用户的信息。 有关完整语法的详细信息，请参阅 .dump（创建转储文件）。

用于创建转储文件的工具

可以使用多种不同的工具来创建用户模式转储文件：

- ProcDump
- WinDbg 和 CDB

ProcDump

ProcDump 是一个命令行实用工具，可用于监视应用程序的 CPU 峰值，并在峰值期间生成崩溃转储。 管理员或开发人员可以使用崩溃转储文件来确定峰值的原因。 ProcDump 还包括对挂起窗口的监视（通过使用与 Windows 和任务管理器使用的窗口挂起相同的定义）和未经处理的异常。 可以使用 ProcDump 根据系统性能计数器的值生成转储。 ProcDump 还可用作可嵌入到其他脚本中的常规进程转储实用工具。

有关使用 Sysinternals ProcDump 实用工具创建用户模式转储文件的信息，请参阅 ProcDump。

WinDbg 和 CDB

控制台调试器 (CDB) 和 Windows 调试器 (WinDbg) 是 Windows 软件开发工具包和 Windows 驱动程序工具包中包含的调试工具。 请参阅下载并安装 WinDbg Windows 调试器中的安装选项。

可以使用 CDB 或 WinDbg 以多种方式创建用户模式转储文件：

- 自动创建转储文件。
- 调试时创建转储文件。
- 收缩现有转储文件。

有关这些工具的详细信息，请参阅 Windows 调试入门指南和使用 CDB 进行调试。

自动创建转储文件

发生应用程序错误时，Windows 可能会根据事后调试设置以多种方式之一做出响应。 如果这些设置指示调试工具创建转储文件，则会创建用户模式内存转储文件。 有关详细信息，请参阅启用事后调试。

调试时创建转储文件

当 CDB 或 WinDbg 调试用户模式应用程序时，还可以使用 .dump（创建转储文件）命令创建转储文件。

此命令不会导致目标应用程序终止。 通过选择特定的命令选项，可以创建一个小型转储文件，其中包含所需的信息量。

收缩现有转储文件

可以使用 CDB 或 WinDbg 收缩转储文件。 若要收缩转储文件，请开始调试现有的转储文件。 然后，使用 `.dump` 命令创建较小的转储文件。

时间旅行调试

调试用户模式应用程序的另一个选项是时间旅行调试 (TTD)。 TTD 是一种工具，可以在进程运行时使用它来记录进程。 可以重播调试器会话的录制，以查找错误。 可以轻松地转到录制的不同部分，了解导致错误的情况以及如何解决问题。

TTD 比崩溃转储文件具有显著的优势，崩溃转储文件通常会丢失导致失败的代码执行。 在代码执行中向后移动的能力对于确定根本原因非常有用。

有关详细信息，请参阅时间旅行调试概述。

另请参阅

- 了解如何分析用户模式转储文件。
- 获取从转储文件中提取信息的提示。

反馈

此页面是否有帮助? 是 否

分析用户模式转储文件

项目 • 2024/03/01

本主题包括以下内容：

- 使用 WinDbg 分析用户模式转储文件
- 使用 CDB 分析用户模式转储文件

使用 WinDbg 分析用户模式转储文件

WinDbg 可以分析用户模式内存转储文件。 创建转储文件的处理器或 Windows 版本不必与运行 WinDbg 的平台一致。

安装符号文件

在分析内存转储文件之前，需要访问生成转储文件的 Windows 版本的符号文件。 您选择用于分析转储文件的调试器将使用这些文件。 有关使用符号服务器的信息，请参阅 Microsoft 公共符号。

还需要安装用户模式进程的所有符号文件（应用程序或系统服务），这会使系统生成转储文件。 如果此代码是由您编写的，那么在编译和链接代码时应该已生成了符号文件。 如果这是商业代码，请在产品 CD-ROM 上查看，或联系软件制造商获取这些特定符号文件。

启动 WinDbg

若要分析转储文件，请使用 **-z** 命令行选项启动 WinDbg：

windbg **-y***SymbolPath* **-i***ImagePath* **-z***DumpFileName*

-v 选项（详细模式）也很有用。 有关选项的完整列表，请参阅 **WinDbg 命令行选项**。

如果 WinDbg 已在运行并且处于休眠模式，则可以通过选择 **File | Open Crash Dump** 菜单命令或按 Ctrl+D 快捷键打开故障转储。 出现“**打开故障转储**”对话框时，在“**文件名**”文本框中输入故障转储文件的完整路径和名称，或使用对话框选择正确的路径和文件名。 选择正确的文件后，选择**打开**。

也可以使用 .opendump （打开转储文件）命令在调试器运行之后打开转储文件，后跟 **g (Go)**。

可以同时调试多个转储文件。 这可以通过在命令行上加入多个 **-z** 开关（每个开关后跟一个不同的文件名），或使用 **.opendump** 添加额外的转储文件作为调试器目标来完成。

有关如何控制多目标会话的信息，请参阅调试多个目标。

转储文件通常以扩展名.dmp或 .mdmp 结尾。 可以将网络共享或通用命名约定 (UNC) 文件名用于内存转储文件。

将转储文件打包到 CAB 文件中也很常见。 如果在 **-z** 选项后指定文件名（包括 .cab 扩展名），或者作为 **.opendump**命令的参数，调试器可以直接从 CAB 读取转储文件。 但是，如果单个 CAB 中存储了多个转储文件，调试器将只能读取其中一个。 调试器不会从 CAB 读取任何其他文件，即使它们是符号文件或与转储文件关联的可执行文件。

分析完整用户转储文件

完整用户转储文件分析类似于实时调试会话的分析。 有关哪些命令可用于在用户模式下调试转储文件的详细信息，请参阅调试器命令参考章节。

分析小型转储文件

分析用户模式小型转储文件的方式与完全用户转储相同。 但是，由于保留的内存量要少得多，可以执行的操作会受到更多限制。 尝试访问内存超出小型转储文件所保留内存的命令将无法正常工作。

其他技术

有关可用于从转储文件读取特定类型信息的技术，请参阅 从转储文件中提取信息。

使用 CDB 分析用户模式转储文件

CDB 可以分析用户模式内存转储文件。 创建转储文件的处理器或 Windows 版本不必与运行 CDB 的平台一致。

安装符号文件

在分析内存转储文件之前，需要访问生成转储文件的 Windows 版本的符号文件。 您选择用于分析转储文件的调试器将使用这些文件。 有关使用符号服务器的信息，请参阅 Microsoft 公共符号。

还需要安装用户模式进程的所有符号文件（应用程序或系统服务），这会使系统生成转储文件。 如果此代码是由您编写的，那么在编译和链接代码时应该已生成了符号文件。 如果这是商业代码，请在产品 CD-ROM 上查看，或联系软件制造商获取这些特定符号文件。

正在启动 CDB

若要分析转储文件，请使用 **-z** 命令行选项启动 CDB：

cdb -y*SymbolPath* **-i***ImagePath* **-z***DumpFileName*

-v 选项（详细模式）也很有用。 有关选项的完整列表，请参阅 **CDB 命令行选项**。

也可以使用 .opendump （打开转储文件）命令在调试器运行之后打开转储文件，后跟 **g (Go)**。 这样就可以同时调试多个转储文件。

可以同时调试多个转储文件。 这可以通过在命令行上加入多个 **-z** 开关（每个开关后跟一个不同的文件名），或使用 **.opendump** 添加额外的转储文件作为调试器目标来完成。 有关如何控制多目标会话的信息，请参阅调试多个目标。

转储文件通常以扩展名.dmp或 .mdmp 结尾。 可以将网络共享或通用命名约定 (UNC) 文件名用于内存转储文件。

将转储文件打包到 CAB 文件中也很常见。 如果在 **-z** 选项后指定文件名（包括 .cab 扩展名），或者作为 **.opendump**命令的参数，调试器可以直接从 CAB 读取转储文件。 但是，如果单个 CAB 中存储了多个转储文件，调试器将只能读取其中一个。 调试器不会从 CAB 读取任何其他文件，即使它们是符号文件或与转储文件关联的可执行文件。

分析完整用户转储文件

完整用户转储文件分析类似于实时调试会话的分析。 有关哪些命令可用于在用户模式下调试转储文件的详细信息，请参阅调试器命令参考章节。

分析小型转储文件

分析用户模式小型转储文件的方式与完全用户转储相同。 但是，由于保留的内存量要少得多，可以执行的操作会受到更多限制。 尝试访问内存超出小型转储文件所保留内存的命令将无法正常工作。

其他技术

有关可用于从转储文件读取特定类型信息的技术，请参阅 从转储文件中提取信息。

从转储文件中提取信息

项目 • 2024/11/06

某些信息（例如目标计算机的名称）可在实时调试期间轻松获取。 还可以调试转储文件以确定信息。 本文介绍如何从转储文件中提取信息。

有关使用转储文件的一般信息，请参阅：

- 分析用户模式转储文件
- 分析内核模式转储文件

在内核模式转储文件中查找计算机名称

如果需要确定创建故障转储的计算机的名称，请使用 ！ peb 扩展。 在其输出中查找 COMPUTERNAME 的值。

在内核模式转储文件中查找 IP 地址

若要确定创建故障转储的计算机的 IP 地址，请查找显示某些发送或接收网络活动的线程堆栈。 打开其中一个发送或接收数据包。 IP 地址在该数据包中可见。

在用户模式转储文件中查找进程 ID

若要从用户模式转储文件确定目标应用程序的进程 ID，请使用 |（进程状态） 命令。 此命令显示写入转储时正在调试的所有进程。 标记为句点 （.） 的进程是当前进程。 其进程 ID 在 ID：**表示法之后的**十六进制中提供。

将 WER 集成到应用程序中

Windows 错误报告（WER） 信息可以集成到应用程序中，以提供其他故障转储信息。 有关详细信息，请参阅 使用 WER。

另请参阅

- 高级驱动程序调试 - 2005 [336 KB] [PPT]
- 下载 Windows 驱动程序工具包 (WDK)
- 驱动程序调试基础知识 [WinHEC 2007; 633 KB] [PPT]

- 如何读取 Windows 创建的小型内存转储文件（如果发生崩溃）
- 资源定义语句
- Windows 错误报告
- VERSIONINFO 资源

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

包含页面文件和内存转储的 CAB 文件

项目 • 2024/03/08

内存转储文件可以与分页文件一起放置在 Cabinet (CAB) 文件中。 当 Windows 调试器分析内存转储文件时，它可以使用分页文件来显示完整视图内存，包括创建转储文件时分页的内存。

假设一个名为 MyCab.cab 的 CAB 文件包含以下文件：

Memory.dmp Cabmanifest.xml Pagefile.sys 还假设 Cabmanifest.xml 如下所示：

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<WatsonPageFileManifest>
  <Pagefiles>
    <Pagefile Name="pagefile.sys"></Pagefile>
  </Pagefiles>
</WatsonPageFileManifest>
```

可以通过输入下列命令之一来打开 CAB 文件：

- **windbg /z MyCab.cab**
- **kd /z MyCab.cab**

调试器读取 Cabmanifest.xml，以获取要包含在调试会话中的页面文件列表。 在此示例中，只有一个分页文件。 调试程序将分页文件转换为可在调试会话期间使用的目标信息文件 (TIF) 文件。 因为调试器可以访问 TIF，所以它可以显示在创建转储文件时调出的内存。

无论 CAB 文件中有多少个分页文件，调试器都只使用 Cabmanifest.xml 中列出的分页文件。 下面是一个 CAB 清单文件的示例，它列出了三个分页文件。

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<WatsonPageFileManifest>
  <Pagefiles>
    <Pagefile Name="pagefile1.sys"></Pagefile>
    <Pagefile Name="pagefile2.sys"></Pagefile>
    <Pagefile Name="swapfile.sys"></Pagefile>
  </Pagefiles>
</WatsonPageFileManifest>
```

在 Cabmanifest.xml 中，分页文件的列出顺序必须与 Windows 使用它们的顺序相同。 也就是说，它们必须按照在注册表中出现的顺序列出。

放入 CAB 文件中的内存转储文件必须是完整的内存转储。 可以使用控制面板将 Windows 配置为在发生崩溃时创建完整的内存转储。 例如，在 Windows 8 中，可以转到**控制面板>系统和安全>系统>高级系统设置>启动和恢复**。 作为使用控制面板的替代方法，可以将此注册表项的值设置为 1。

HKLM\SYSTEM\CurrentControlSet\Control\CrashControl\CrashDumpEnabled

从 Windows 8.1 开始，可以将 Windows 配置为在 Windows 重启时保留分页文件的内容。

若要指定希望在 Windows 重启时保存分页文件，请将此注册表项的值设置为 1。

HKLM\SYSTEM\CurrentControlSet\Control\CrashControl\SavePageFileContents

错误检查（蓝屏）

项目 • 2024/10/16

注意 如果你是 IT 专业人员或支持代理，请参阅本文以了解其他信息，即先对“蓝屏”或“停止错误问题”进行故障排除，然后再联系 Microsoft 支持部门。 如果你是一个客户，你的系统显示了带有错误检查代码的蓝屏，请参阅蓝屏错误疑难解答。

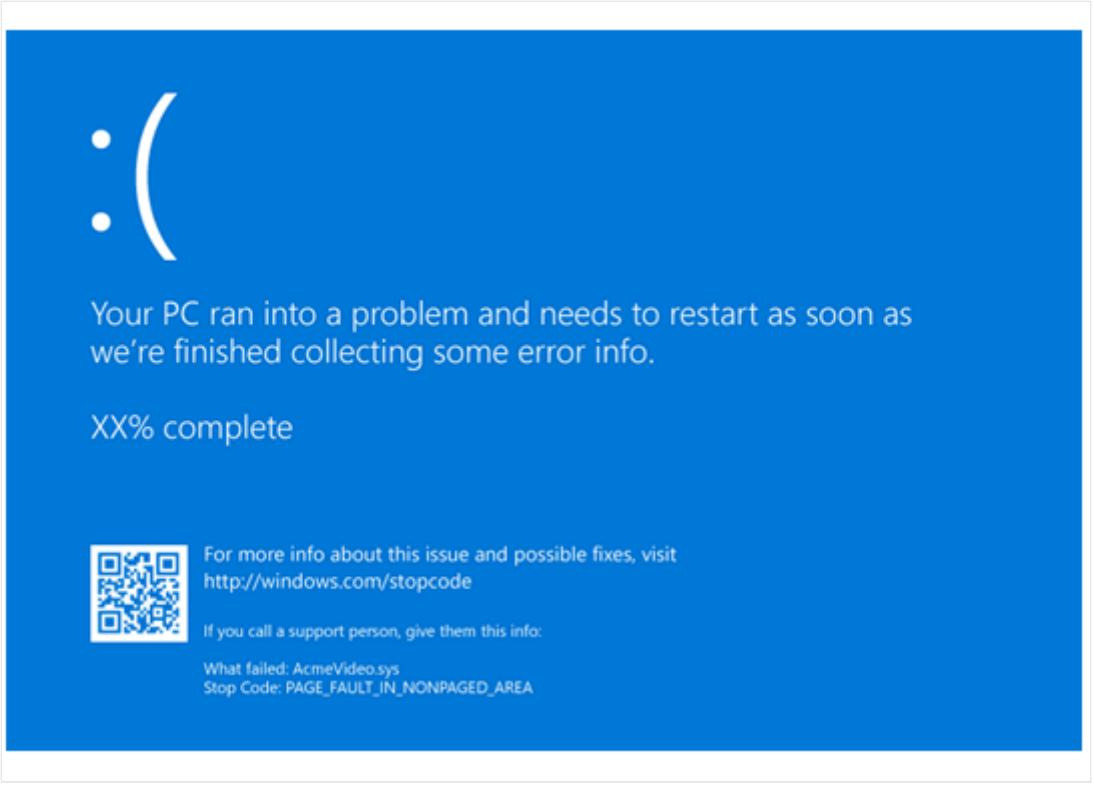
错误检查系统停止

当 Microsoft Windows 遇到危及系统安全操作的情况时，系统将停止。 此条件称为*错误检查*。 它也称为*系统崩溃*、*内核错误*或 *停止错误*。

可能发生的情况示例包括：

- 如果允许操作系统在操作系统完整性受到破坏后继续运行，可能会损坏数据或危及系统的安全性。
- 如果在系统上启用了故障转储，则会创建故障转储文件。
- 如果内核调试器已附加且处于活动状态，则系统会中断，以便使用调试器调查崩溃。
- 如果未附加调试器，将显示一个蓝色文本屏幕，其中包含错误相关信息。 此屏幕称为*蓝屏*、*错误检查屏幕*或*停止屏幕*。

如果您使用的是 Windows 预览体验成员版本，文本会显示在绿色背景上。 蓝屏的确切外观取决于错误原因。 以下示例演示了可能出现的蓝屏类型：



显示停止代码，例如 PAGE_FAULT_IN_NONPAGED_AREA。 当它可用时，还会显示所执行代码的模块名称，例如 **AcmeVideo.sys**。

如果已写入内核模式转储文件，则会在写入转储时以百分比完成倒计时的方式显示。

存在与错误检查代码引用中列出的每个停止代码相关的停止代码十六进制值。

错误检查（蓝屏）的一般提示

如果计算机停止工作并显示蓝屏，则表示计算机已突然关闭以防止数据丢失。 可能是硬件设备、其驱动程序或相关软件导致此错误。 若要了解有关显示的信息的详细信息（如故障驱动程序名称），请参阅分析错误检查蓝屏数据。

有关 Windows 错误检查代码的一般故障排除，请按照以下建议操作：

- 如果最近添加了新的设备驱动程序或系统服务，请尝试删除或更新它们。 尝试确定系统中导致新错误检查代码出现的原因。
- 在**设备管理器**中查看是否有任何设备标有感叹号 (!)。 查看驱动程序属性中显示的事件日志，以查找任何故障驱动程序。 请尝试更新相关驱动程序。
- 如果最近向系统添加了硬件，请尝试删除或替换它。 也可以与制造商联系，查看是否有可用的修补程序。
- 你可尝试运行系统制造商提供的硬件诊断。

- 在事件查看器中检查系统日志以获取其他错误消息，这可能有助于发现导致错误的设备或驱动程序。 有关详细信息，请参阅打开事件查看器。 在系统日志中查找与蓝屏同时出现的严重错误。

- 运行 Windows 内存诊断工具以测试内存。 在控制面板搜索框中键入**内存**，然后选择**诊断计算机内存问题**。运行测试后，使用事件查看器查看系统日志下的结果。 查找“内存诊断结果”条目以查看结果。

- 确保安装的任何新硬件都与已安装的 Windows 版本兼容。 例如，可以在 Windows 10 规范 中获取所需硬件的相关信息。

- 运行病毒检测程序。 病毒可感染针对 Windows 格式化的所有类型的硬盘，由此导致的磁盘损坏可能会生成系统错误检查代码。 使用病毒检测程序检查主启动记录是否存在感染。

- 使用扫描磁盘实用工具确认没有文件系统错误。 选择并按住（或右键单击）要扫描的驱动器，然后选择**属性**>**工具**>**立即检查**。

- 使用系统文件检查器工具修复丢失或损坏的系统文件。 系统文件检查器是 Windows 中的一个实用工具，供用户扫描 Windows 系统文件中的损坏情况并恢复损坏的文件。 使用以下命令运行系统文件检查器工具 (SFC.exe)。

  控制台

  ```
  SFC /scannow
  ```

  有关详细信息，请参阅使用系统文件检查器工具修复丢失或损坏的系统文件 。

- 确认硬盘驱动器上有足够的可用空间。 操作系统和某些应用程序需要足够的可用空间来创建交换文件和执行其他功能。 确切的要求因系统配置而异，但最好有 10% 到 15% 的可用空间。

- 验证系统是否已安装最新的 Service Pack。 若要检测系统上安装了哪个 Service Pack（如有），请选择**开始**，然后选择**运行**并输入 **winver**，然后选择 Enter。 “**关于 Windows**”对话框显示 Windows 版本号和 Service Pack 的版本号（如果已安装）。

- 请与制造商联系，查看是否有更新的系统 BIOS 或 UEFI 固件。

- 请确保电脑的所有扩展板都正确安置，且所有电缆都良好连接。

- **使用安全模式**

  删除或禁用组件时，请考虑使用安全模式。 在 Windows 启动过程中，使用安全模式只会加载所需的最低驱动程序和系统服务。

1. 若要进入安全模式，请转到设置并选择**更新和安全**。
2. 选择**恢复**>**高级启动**以启动进入维护模式。
3. 在生成的菜单中，选择**疑难解答**>**高级选项**>**启动设置**>**重启**。
4. Windows 重启进入“**启动设置**”界面后，选择选项 4、5 或 6 以启动进入安全模式。

在启动时按下功能键（例如 F8）可以进入安全模式。 有关具体的启动选项，请参阅制造商提供的信息。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

分析错误检查蓝屏数据

项目 • 2024/10/16

> **备注**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

> **备注**
>
> 如果您是 IT 专业人员或支持代理方，请参阅**停止错误或蓝屏错误问题高级疑难解答**了解更多信息。

收集停止代码参数

每个错误检查代码都有四个提供信息的关联参数。 错误检查代码引用中介绍了每个停止代码的参数。

可通过多种方式收集四个停止代码参数。

- 在事件查看器中检查 Windows 系统日志。 错误检查的事件属性将列出四个停止代码参数。 有关详细信息，请参阅打开事件查看器。
- 加载生成的转储文件，并将 !analyze 命令与附加的调试器一起使用。 有关详细信息，请参阅使用 WinDbg 分析内核模式转储文件。
- 将内核调试器附加到故障电脑。 出现停止代码时，调试器输出将包含停止代码十六进制值后的四个参数。

```
dbgcmd

*******************************************************************************
*
*
*                        Bugcheck Analysis
*
*
*
*******************************************************************************

Use !analyze -v to get detailed debugging information.
```

```
BugCheck 9F, {3, ffffe000f38c06a0, fffff803c596cad0, ffffe000f46a1010}

Implicit thread is now ffffe000`f4ca3040
Probably caused by : hidusb.sys
```

错误检查符号名称

DRIVER_POWER_STATE_FAILURE 是错误检查符号名称，与错误检查代码 9F 相关联。 错误检查代码引用中列出了与错误检查符号名称相关的停止代码十六进制值。

从调试器中读取错误检查信息

如果附加了调试器并且已在电脑上启用调试，则检查错误将导致目标计算机进入调试器。 在这种情况下，蓝屏可能不会立即显示。 此崩溃的完整详细信息将发送到调试器，并显示在调试器窗口中。 若要再次查看此信息，请使用 .bugcheck（显示错误检查数据）命令或 !analyze 扩展命令。 有关启用调试的信息，请参阅 WinDbg 入门指南（内核模式）。

内核调试和故障转储分析

当其他故障排除技术失败或问题反复出现时，内核调试特别有用。 请记住在错误消息的错误检查信息部分捕捉确切的文本。 若要隔离复杂问题并制定可行的解决方法，记录导致故障的确切操作非常有用。

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

您还可以在导致此停止代码的代码中设置断点，并尝试单步执行故障代码。

有关详细信息，请参阅以下文章：

使用 Windows 调试器 (WinDbg) 进行故障转储分析

使用 WinDbg 分析内核模式转储文件

使用 !analyze 扩展 和 !analyze

碎片整理工具显示

使用驱动程序验证程序收集信息

据估计，大约四分之三的蓝屏是由故障驱动程序引起的。 驱动程序验证程序是一个实时运行的工具，用于检查驱动程序的行为。 例如，驱动程序验证程序检查内存资源（如内

存池）的使用。 如果在执行驱动程序代码时发现错误，它会主动创建一个异常，以允许进一步检查该部分驱动程序代码。 驱动程序验证程序管理器内置于 Windows 中，可在所有 Windows PC 上使用。 若要启动驱动程序验证程序管理器，请在命令提示下输入 **Verifier**。 你可以配置要验证的驱动程序。 验证驱动程序的代码在运行时会增加开销，因此请尝试验证尽可能少的驱动程序。 有关详细信息，请参阅驱动程序验证程序。

给软件工程师的提示

当您编写的代码出现错误检查时，应使用内核调试器分析问题，然后修复代码中的 Bug。 有关完整详细信息，请参阅错误检查代码参考章节中的各个错误检查代码。

但是，您也可能遇到并非由自己的代码引起的错误检查。 在这种情况下，您可能无法解决问题的实际原因，因此您的目标应该是绕过问题。 如果可能，请隔离并删除发生故障的硬件或软件组件。

许多问题可以通过基本故障排除过程来解决，例如验证说明、重新安装关键组件和验证文件日期。 此外，事件查看器、Sysinternals 诊断工具和网络监视工具可能会隔离并解决这些问题。

另请参阅

- 错误检查代码参考
- 使用 Windows 调试器 (WinDbg) 进行故障转储分析
- 使用 WinDbg 分析内核模式转储文件
- 使用 !analyze 扩展 和 !analyze

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

错误检查代码参考

项目 • 2024/08/23

本部分包含常见错误检查代码的说明，这些代码显示在蓝色错误检查屏幕上。 本部分还介绍如何使用 Windows 调试程序中的 **!analyze** 扩展来显示有关错误检查代码的信息。

> **备注**
>
> 本主题是面向程序员的。 如果你是一个客户，你的系统显示了带有错误检查代码的蓝屏，请参阅**蓝屏错误疑难解答**。

有关错误检查系统停止错误的常规信息，请参阅错误检查（蓝屏）。

如果你是 IT 专业人员，请参阅本文以了解其他信息，即先对“蓝屏”或“停止错误问题”进行故障排除，然后再联系 Microsoft 支持部门。

使用 WinDbg 显示终止代码信息

如果此主题中没有某个错误检查代码的说明信息，请在 Windows 调试程序 (WinDbg) 中使用以下语法（内核模式）和 **!analyze** 扩展，并将 `<code>` 替换为某个错误检查代码：

```
!analyze -show <code>
```

输入此命令会使 WinDbg 显示指定的错误检查代码的相关信息。 如果默认数字基数不是 16，则为 `<code>` 添加前缀 **0x**。

向 !analyze 命令提供终止代码参数，以显示任何可用的参数信息。 例如，若要显示有关参数 1 值为 0x3 的 错误检查 0x9F：DRIVER_POWER_STATE_FAILURE 的信息，请使用 `!analyze -show 0x9F 0x3`，如下所示。

dbgcmd

```
1: kd> !analyze -show 0x9F 0x3
DRIVER_POWER_STATE_FAILURE (9f)
A driver has failed to complete a power IRP within a specific time.
Arguments:
Arg1: 0000000000000003, A device object has been blocking an Irp for too
long a time
Arg2: 0000000000000000, Physical Device Object of the stack
Arg3: 0000000000000000, nt!_TRIAGE_9F_POWER on Win7 and higher, otherwise
the Functional Device Object of the stack
Arg4: 0000000000000000, The blocked IRP
```

要下载 WinDbg，请参阅 适用于 Windows 的调试工具。 若要了解有关 WinDbg 开发工具的详细信息，请参阅 Windows 调试入门。

错误检查转储文件

进行错误检查时，可能会有一个转储文件，其中包含有关终止代码出现时的内存内容的其他信息。 若要了解故障期间的内存内容，需要了解处理器内存寄存器和程序集。

有关详细信息，请参阅：

- 使用 WinDbg 分析内核模式转储文件
- !analyze
- 处理器体系结构

实时转储

实时转储会停止代码不会重置 OS，但允许在操作系统可继续的异常情况下捕获内存信息。 此处未列出实时转储停止代码。 有关实时转储的信息，请参阅错误检查代码参考 - 实时转储。

错误检查代码

下表提供了错误检查代码的链接。

展开表

代码	名称
0x00000001	APC_INDEX_MISMATCH
0x00000002	DEVICE_QUEUE_NOT_BUSY
0x00000003	INVALID_AFFINITY_SET
0x00000004	INVALID_DATA_ACCESS_TRAP
0x00000005	INVALID_PROCESS_ATTACH_ATTEMPT
0x00000006	INVALID_PROCESS_DETACH_ATTEMPT
0x00000007	INVALID_SOFTWARE_INTERRUPT
0x00000008	IRQL_NOT_DISPATCH_LEVEL

代码	名称
0x00000009	IRQL_NOT_GREATER_OR_EQUAL
0x0000000A	IRQL_NOT_LESS_OR_EQUAL
0x0000000B	NO_EXCEPTION_HANDLING_SUPPORT
0x0000000C	MAXIMUM_WAIT_OBJECTS_EXCEEDED
0x0000000D	MUTEX_LEVEL_NUMBER_VIOLATION
0x0000000E	NO_USER_MODE_CONTEXT
0x0000000F	SPIN_LOCK_ALREADY_OWNED
0x00000010	SPIN_LOCK_NOT_OWNED
0x00000011	THREAD_NOT_MUTEX_OWNER
0x00000012	TRAP_CAUSE_UNKNOWN
0x00000013	EMPTY_THREAD_REAPER_LIST
0x00000014	CREATE_DELETE_LOCK_NOT_LOCKED
0x00000015	LAST_CHANCE_CALLED_FROM_KMODE
0x00000016	CID_HANDLE_CREATION
0x00000017	CID_HANDLE_DELETION
0x00000018	REFERENCE_BY_POINTER
0x00000019	BAD_POOL_HEADER
0x0000001A	MEMORY_MANAGEMENT
0x0000001B	PFN_SHARE_COUNT
0x0000001C	PFN_REFERENCE_COUNT
0x0000001D	NO_SPIN_LOCK_AVAILABLE
0x0000001E	KMODE_EXCEPTION_NOT_HANDLED
0x0000001F	SHARED_RESOURCE_CONV_ERROR
0x00000020	KERNEL_APC_PENDING_DURING_EXIT
0x00000021	QUOTA_UNDERFLOW
0x00000022	FILE_SYSTEM

代码	名称
0x00000023	FAT_FILE_SYSTEM
0x00000024	NTFS_FILE_SYSTEM
0x00000025	NPFS_FILE_SYSTEM
0x00000026	CDFS_FILE_SYSTEM
0x00000027	RDR_FILE_SYSTEM
0x00000028	CORRUPT_ACCESS_TOKEN
0x00000029	SECURITY_SYSTEM
0x0000002A	INCONSISTENT_IRP
0x0000002B	PANIC_STACK_SWITCH
0x0000002C	PORT_DRIVER_INTERNAL
0x0000002D	SCSI_DISK_DRIVER_INTERNAL
0x0000002E	DATA_BUS_ERROR
0x0000002F	INSTRUCTION_BUS_ERROR
0x00000030	SET_OF_INVALID_CONTEXT
0x00000031	PHASE0_INITIALIZATION_FAILED
0x00000032	PHASE1_INITIALIZATION_FAILED
0x00000033	UNEXPECTED_INITIALIZATION_CALL
0x00000034	CACHE_MANAGER
0x00000035	NO_MORE_IRP_STACK_LOCATIONS
0x00000036	DEVICE_REFERENCE_COUNT_NOT_ZERO
0x00000037	FLOPPY_INTERNAL_ERROR
0x00000038	SERIAL_DRIVER_INTERNAL
0x00000039	SYSTEM_EXIT_OWNED_MUTEX
0x0000003A	SYSTEM_UNWIND_PREVIOUS_USER
0x0000003B	SYSTEM_SERVICE_EXCEPTION
0x0000003C	INTERRUPT_UNWIND_ATTEMPTED

代码	名称
0x0000003D	INTERRUPT_EXCEPTION_NOT_HANDLED
0x0000003E	MULTIPROCESSOR_CONFIGURATION_NOT_SUPPORTED
0x0000003F	NO_MORE_SYSTEM_PTES
0x00000040	TARGET_MDL_TOO_SMALL
0x00000041	MUST_SUCCEED_POOL_EMPTY
0x00000042	ATDISK_DRIVER_INTERNAL
0x00000043	NO_SUCH_PARTITION
0x00000044	MULTIPLE_IRP_COMPLETE_REQUESTS
0x00000045	INSUFFICIENT_SYSTEM_MAP_REGS
0x00000046	DEREF_UNKNOWN_LOGON_SESSION
0x00000047	REF_UNKNOWN_LOGON_SESSION
0x00000048	CANCEL_STATE_IN_COMPLETED_IRP
0x00000049	PAGE_FAULT_WITH_INTERRUPTS_OFF
0x0000004A	IRQL_GT_ZERO_AT_SYSTEM_SERVICE
0x0000004B	STREAMS_INTERNAL_ERROR
0x0000004C	FATAL_UNHANDLED_HARD_ERROR
0x0000004D	NO_PAGES_AVAILABLE
0x0000004E	PFN_LIST_CORRUPT
0x0000004F	NDIS_INTERNAL_ERROR
0x00000050	PAGE_FAULT_IN_NONPAGED_AREA
0x00000051	REGISTRY_ERROR
0x00000052	MAILSLOT_FILE_SYSTEM
0x00000053	NO_BOOT_DEVICE
0x00000054	LM_SERVER_INTERNAL_ERROR
0x00000055	DATA_COHERENCY_EXCEPTION
0x00000056	INSTRUCTION_COHERENCY_EXCEPTION

代码	名称
0x00000057	XNS_INTERNAL_ERROR
0x00000058	FTDISK_INTERNAL_ERROR
0x00000059	PINBALL_FILE_SYSTEM
0x0000005A	CRITICAL_SERVICE_FAILED
0x0000005B	SET_ENV_VAR_FAILED
0x0000005C	HAL_INITIALIZATION_FAILED
0x0000005D	UNSUPPORTED_PROCESSOR
0x0000005E	OBJECT_INITIALIZATION_FAILED
0x0000005F	SECURITY_INITIALIZATION_FAILED
0x00000060	PROCESS_INITIALIZATION_FAILED
0x00000061	HAL1_INITIALIZATION_FAILED
0x00000062	OBJECT1_INITIALIZATION_FAILED
0x00000063	SECURITY1_INITIALIZATION_FAILED
0x00000064	SYMBOLIC_INITIALIZATION_FAILED
0x00000065	MEMORY1_INITIALIZATION_FAILED
0x00000066	CACHE_INITIALIZATION_FAILED
0x00000067	CONFIG_INITIALIZATION_FAILED
0x00000068	FILE_INITIALIZATION_FAILED
0x00000069	IO1_INITIALIZATION_FAILED
0x0000006A	LPC_INITIALIZATION_FAILED
0x0000006B	PROCESS1_INITIALIZATION_FAILED
0x0000006C	REFMON_INITIALIZATION_FAILED
0x0000006D	SESSION1_INITIALIZATION_FAILED
0x0000006E	SESSION2_INITIALIZATION_FAILED
0x0000006F	SESSION3_INITIALIZATION_FAILED
0x00000070	SESSION4_INITIALIZATION_FAILED

代码	名称
0x00000071	SESSION5_INITIALIZATION_FAILED
0x00000072	ASSIGN_DRIVE_LETTERS_FAILED
0x00000073	CONFIG_LIST_FAILED
0x00000074	BAD_SYSTEM_CONFIG_INFO
0x00000075	CANNOT_WRITE_CONFIGURATION
0x00000076	PROCESS_HAS_LOCKED_PAGES
0x00000077	KERNEL_STACK_INPAGE_ERROR
0x00000078	PHASE0_EXCEPTION
0x00000079	MISMATCHED_HAL
0x0000007A	KERNEL_DATA_INPAGE_ERROR
0x0000007B	INACCESSIBLE_BOOT_DEVICE
0x0000007C	BUGCODE_NDIS_DRIVER
0x0000007D	INSTALL_MORE_MEMORY
0x0000007E	SYSTEM_THREAD_EXCEPTION_NOT_HANDLED
0x0000007F	UNEXPECTED_KERNEL_MODE_TRAP
0x00000080	NMI_HARDWARE_FAILURE
0x00000081	SPIN_LOCK_INIT_FAILURE
0x00000082	DFS_FILE_SYSTEM
0x00000085	SETUP_FAILURE
0x0000008B	MBR_CHECKSUM_MISMATCH
0x0000008E	KERNEL_MODE_EXCEPTION_NOT_HANDLED
0x0000008F	PP0_INITIALIZATION_FAILED
0x00000090	PP1_INITIALIZATION_FAILED
0x00000092	UP_DRIVER_ON_MP_SYSTEM
0x00000093	INVALID_KERNEL_HANDLE
0x00000094	KERNEL_STACK_LOCKED_AT_EXIT

代码	名称
0x00000096	INVALID_WORK_QUEUE_ITEM
0x00000097	BOUND_IMAGE_UNSUPPORTED
0x00000098	END_OF_NT_EVALUATION_PERIOD
0x00000099	INVALID_REGION_OR_SEGMENT
0x0000009A	SYSTEM_LICENSE_VIOLATION
0x0000009B	UDFS_FILE_SYSTEM
0x0000009C	MACHINE_CHECK_EXCEPTION
0x0000009E	USER_MODE_HEALTH_MONITOR
0x0000009F	DRIVER_POWER_STATE_FAILURE
0x000000A0	INTERNAL_POWER_ERROR
0x000000A1	PCI_BUS_DRIVER_INTERNAL
0x000000A2	MEMORY_IMAGE_CORRUPT
0x000000A3	ACPI_DRIVER_INTERNAL
0x000000A4	CNSS_FILE_SYSTEM_FILTER
0x000000A5	ACPI_BIOS_ERROR
0x000000A7	BAD_EXHANDLE
0x000000AC	HAL_MEMORY_ALLOCATION
0x000000AD	VIDEO_DRIVER_DEBUG_REPORT_REQUEST
0x000000B1	BGI_DETECTED_VIOLATION
0x000000B4	VIDEO_DRIVER_INIT_FAILURE
0x000000B8	ATTEMPTED_SWITCH_FROM_DPC
0x000000B9	CHIPSET_DETECTED_ERROR
0x000000BA	SESSION_HAS_VALID_VIEWS_ON_EXIT
0x000000BB	NETWORK_BOOT_INITIALIZATION_FAILED
0x000000BC	NETWORK_BOOT_DUPLICATE_ADDRESS
0x000000BD	INVALID_HIBERNATED_STATE

代码	名称
0x000000BE	ATTEMPTED_WRITE_TO_READONLY_MEMORY
0x000000BF	MUTEX_ALREADY_OWNED
0x000000C1	SPECIAL_POOL_DETECTED_MEMORY_CORRUPTION
0x000000C2	BAD_POOL_CALLER
0x000000C4	DRIVER_VERIFIER_DETECTED_VIOLATION
0x000000C5	DRIVER_CORRUPTED_EXPOOL
0x000000C6	DRIVER_CAUGHT_MODIFYING_FREED_POOL
0x000000C7	TIMER_OR_DPC_INVALID
0x000000C8	IRQL_UNEXPECTED_VALUE
0x000000C9	DRIVER_VERIFIER_IOMANAGER_VIOLATION
0x000000CA	PNP_DETECTED_FATAL_ERROR
0x000000CB	DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS
0x000000CC	PAGE_FAULT_IN_FREED_SPECIAL_POOL
0x000000CD	PAGE_FAULT_BEYOND_END_OF_ALLOCATION
0x000000CE	DRIVER_UNLOADED_WITHOUT_CANCELLING_PENDING_OPERATIONS
0x000000CF	TERMINAL_SERVER_DRIVER_MADE_INCORRECT_MEMORY_REFERENCE
0x000000D0	DRIVER_CORRUPTED_MMPOOL
0x000000D1	DRIVER_IRQL_NOT_LESS_OR_EQUAL
0x000000D2	BUGCODE_ID_DRIVER
0x000000D3	DRIVER_PORTION_MUST_BE_NONPAGED
0x000000D4	SYSTEM_SCAN_AT_RAISED_IRQL_CAUGHT_IMPROPER_DRIVER_UNLOAD
0x000000D5	DRIVER_PAGE_FAULT_IN_FREED_SPECIAL_POOL
0x000000D6	DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION
0x000000D7	DRIVER_UNMAPPING_INVALID_VIEW
0x000000D8	DRIVER_USED_EXCESSIVE_PTES
0x000000D9	LOCKED_PAGES_TRACKER_CORRUPTION

代码	名称
0x000000DA	SYSTEM_PTE_MISUSE
0x000000DB	DRIVER_CORRUPTED_SYSPTES
0x000000DC	DRIVER_INVALID_STACK_ACCESS
0x000000DE	POOL_CORRUPTION_IN_FILE_AREA
0x000000DF	IMPERSONATING_WORKER_THREAD
0x000000E0	ACPI_BIOS_FATAL_ERROR
0x000000E1	WORKER_THREAD_RETURNED_AT_BAD_IRQL
0x000000E2	MANUALLY_INITIATED_CRASH
0x000000E3	RESOURCE_NOT_OWNED
0x000000E4	WORKER_INVALID
0x000000E6	DRIVER_VERIFIER_DMA_VIOLATION
0x000000E7	INVALID_FLOATING_POINT_STATE
0x000000E8	INVALID_CANCEL_OF_FILE_OPEN
0x000000E9	ACTIVE_EX_WORKER_THREAD_TERMINATION
0x000000EA	THREAD_STUCK_IN_DEVICE_DRIVER
0x000000EB	DIRTY_MAPPED_PAGES_CONGESTION
0x000000EC	SESSION_HAS_VALID_SPECIAL_POOL_ON_EXIT
0x000000ED	UNMOUNTABLE_BOOT_VOLUME
0x000000EF	CRITICAL_PROCESS_DIED
0x000000F0	STORAGE_MINIPORT_ERROR
0x000000F1	SCSI_VERIFIER_DETECTED_VIOLATION
0x000000F2	HARDWARE_INTERRUPT_STORM
0x000000F3	DISORDERLY_SHUTDOWN
0x000000F4	CRITICAL_OBJECT_TERMINATION
0x000000F5	FLTMGR_FILE_SYSTEM
0x000000F6	PCI_VERIFIER_DETECTED_VIOLATION

代码	名称
0x000000F7	DRIVER_OVERRAN_STACK_BUFFER
0x000000F8	RAMDISK_BOOT_INITIALIZATION_FAILED
0x000000F9	DRIVER_RETURNED_STATUS_REPARSE_FOR_VOLUME_OPEN
0x000000FA	HTTP_DRIVER_CORRUPTED
0x000000FC	ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY
0x000000FD	DIRTY_NOWRITE_PAGES_CONGESTION
0x000000FE	BUGCODE_USB_DRIVER
0x000000FF	RESERVE_QUEUE_OVERFLOW
0x00000100	LOADER_BLOCK_MISMATCH
0x00000101	CLOCK_WATCHDOG_TIMEOUT
0x00000102	DPC_WATCHDOG_TIMEOUT
0x00000103	MUP_FILE_SYSTEM
0x00000104	AGP_INVALID_ACCESS
0x00000105	AGP_GART_CORRUPTION
0x00000106	AGP_ILLEGALLY_REPROGRAMMED
0x00000108	THIRD_PARTY_FILE_SYSTEM_FAILURE
0x00000109	CRITICAL_STRUCTURE_CORRUPTION
0x0000010A	APP_TAGGING_INITIALIZATION_FAILED
0x0000010C	FSRTL_EXTRA_CREATE_PARAMETER_VIOLATION
0x0000010D	WDF_VIOLATION
0x0000010E	VIDEO_MEMORY_MANAGEMENT_INTERNAL
0x0000010F	RESOURCE_MANAGER_EXCEPTION_NOT_HANDLED
0x00000111	RECURSIVE_NMI
0x00000112	MSRPC_STATE_VIOLATION
0x00000113	VIDEO_DXGKRNL_FATAL_ERROR
0x00000114	VIDEO_SHADOW_DRIVER_FATAL_ERROR

代码	名称
0x00000115	AGP_INTERNAL
0x00000116	VIDEO_TDR_FAILURE
0x00000117	VIDEO_TDR_TIMEOUT_DETECTED
0x00000119	VIDEO_SCHEDULER_INTERNAL_ERROR
0x0000011A	EM_INITIALIZATION_FAILURE
0x0000011B	DRIVER_RETURNED_HOLDING_CANCEL_LOCK
0x0000011C	ATTEMPTED_WRITE_TO_CM_PROTECTED_STORAGE
0x0000011D	EVENT_TRACING_FATAL_ERROR
0x0000011E	TOO_MANY_RECURSIVE_FAULTS
0x0000011F	INVALID_DRIVER_HANDLE
0x00000120	BITLOCKER_FATAL_ERROR
0x00000121	DRIVER_VIOLATION
0x00000122	WHEA_INTERNAL_ERROR
0x00000123	CRYPTO_SELF_TEST_FAILURE
0x00000124	WHEA_UNCORRECTABLE_ERROR
0x00000125	NMR_INVALID_STATE
0x00000126	NETIO_INVALID_POOL_CALLER
0x00000127	PAGE_NOT_ZERO
0x00000128	WORKER_THREAD_RETURNED_WITH_BAD_IO_PRIORITY
0x00000129	WORKER_THREAD_RETURNED_WITH_BAD_PAGING_IO_PRIORITY
0x0000012A	MUI_NO_VALID_SYSTEM_LANGUAGE
0x0000012B	FAULTY_HARDWARE_CORRUPTED_PAGE
0x0000012C	EXFAT_FILE_SYSTEM
0x0000012D	VOLSNAP_OVERLAPPED_TABLE_ACCESS
0x0000012E	INVALID_MDL_RANGE
0x0000012F	VHD_BOOT_INITIALIZATION_FAILED

代码	名称
0x00000130	DYNAMIC_ADD_PROCESSOR_MISMATCH
0x00000131	INVALID_EXTENDED_PROCESSOR_STATE
0x00000132	RESOURCE_OWNER_POINTER_INVALID
0x00000133	DPC_WATCHDOG_VIOLATION
0x00000134	DRIVE_EXTENDER
0x00000135	REGISTRY_FILTER_DRIVER_EXCEPTION
0x00000136	VHD_BOOT_HOST_VOLUME_NOT_ENOUGH_SPACE
0x00000137	WIN32K_HANDLE_MANAGER
0x00000138	GPIO_CONTROLLER_DRIVER_ERROR
0x00000139	KERNEL_SECURITY_CHECK_FAILURE
0x0000013A	KERNEL_MODE_HEAP_CORRUPTION
0x0000013B	PASSIVE_INTERRUPT_ERROR
0x0000013C	INVALID_IO_BOOST_STATE
0x0000013D	CRITICAL_INITIALIZATION_FAILURE
0x00000140	STORAGE_DEVICE_ABNORMALITY_DETECTED
0x00000143	PROCESSOR_DRIVER_INTERNAL
0x00000144	BUGCODE_USB3_DRIVER
0x00000145	SECURE_BOOT_VIOLATION
0x00000147	ABNORMAL_RESET_DETECTED
0x00000149	REFS_FILE_SYSTEM
0x0000014A	KERNEL_WMI_INTERNAL
0x0000014B	SOC_SUBSYSTEM_FAILURE
0x0000014C	FATAL_ABNORMAL_RESET_ERROR
0x0000014D	EXCEPTION_SCOPE_INVALID
0x0000014E	SOC_CRITICAL_DEVICE_REMOVED
0x0000014F	PDC_WATCHDOG_TIMEOUT

代码	名称
0x00000150	TCPIP_AOAC_NIC_ACTIVE_REFERENCE_LEAK
0x00000151	UNSUPPORTED_INSTRUCTION_MODE
0x00000152	INVALID_PUSH_LOCK_FLAGS
0x00000153	KERNEL_LOCK_ENTRY_LEAKED_ON_THREAD_TERMINATION
0x00000154	UNEXPECTED_STORE_EXCEPTION
0x00000155	OS_DATA_TAMPERING
0x00000157	KERNEL_THREAD_PRIORITY_FLOOR_VIOLATION
0x00000158	ILLEGAL_IOMMU_PAGE_FAULT
0x00000159	HAL_ILLEGAL_IOMMU_PAGE_FAULT
0x0000015A	SDBUS_INTERNAL_ERROR
0x0000015B	WORKER_THREAD_RETURNED_WITH_SYSTEM_PAGE_PRIORITY_ACTIVE
0x00000160	WIN32K_ATOMIC_CHECK_FAILURE
0x00000162	KERNEL_AUTO_BOOST_INVALID_LOCK_RELEASE
0x00000163	WORKER_THREAD_TEST_CONDITION
0x00000164	WIN32K_CRITICAL_FAILURE
0x0000016C	INVALID_RUNDOWN_PROTECTION_FLAGS
0x0000016D	INVALID_SLOT_ALLOCATOR_FLAGS
0x0000016E	ERESOURCE_INVALID_RELEASE
0x00000170	CLUSTER_CSV_CLUSSVC_DISCONNECT_WATCHDOG
0x00000171	CRYPTO_LIBRARY_INTERNAL_ERROR
0x00000173	COREMSGCALL_INTERNAL_ERROR
0x00000174	COREMSG_INTERNAL_ERROR
0x00000178	ELAM_DRIVER_DETECTED_FATAL_ERROR
0x0000017B	PROFILER_CONFIGURATION_ILLEGAL
0x0000017E	MICROCODE_REVISION_MISMATCH
0x00000187	VIDEO_DWMINIT_TIMEOUT_FALLBACK_BDD

代码	名称
0x00000189	BAD_OBJECT_HEADER
0x0000018B	SECURE_KERNEL_ERROR
0x0000018C	HYPERGUARD_VIOLATION
0x0000018D	SECURE_FAULT_UNHANDLED
0x0000018E	KERNEL_PARTITION_REFERENCE_VIOLATION
0x00000191	PF_DETECTED_CORRUPTION
0x00000192	KERNEL_AUTO_BOOST_LOCK_ACQUISITION_WITH_RAISED_IRQL
0x00000196	LOADER_ROLLBACK_DETECTED
0x00000197	WIN32K_SECURITY_FAILURE
0x00000199	KERNEL_STORAGE_SLOT_IN_USE
0x0000019A	WORKER_THREAD_RETURNED_WHILE_ATTACHED_TO_SILO
0x0000019B	TTM_FATAL_ERROR
0x0000019C	WIN32K_POWER_WATCHDOG_TIMEOUT
0x000001A0	TTM_WATCHDOG_TIMEOUT
0x000001A2	WIN32K_CALLOUT_WATCHDOG_BUGCHECK
0x000001AA	EXCEPTION_ON_INVALID_STACK
0x000001AB	UNWIND_ON_INVALID_STACK
0x000001C6	FAST_ERESOURCE_PRECONDITION_VIOLATION
0x000001C7	STORE_DATA_STRUCTURE_CORRUPTION
0x000001C8	MANUALLY_INITIATED_POWER_BUTTON_HOLD
0x000001CA	SYNTHETIC_WATCHDOG_TIMEOUT
0x000001CB	INVALID_SILO_DETACH
0x000001CD	INVALID_CALLBACK_STACK_ADDRESS
0x000001CE	INVALID_KERNEL_STACK_ADDRESS
0x000001CF	HARDWARE_WATCHDOG_TIMEOUT
0x000001D0	CPI_FIRMWARE_WATCHDOG_TIMEOUT

代码	名称
0x000001D2	WORKER_THREAD_INVALID_STATE
0x000001D3	WFP_INVALID_OPERATION
0x000001D5	DRIVER_PNP_WATCHDOG
0x000001D6	WORKER_THREAD_RETURNED_WITH_NON_DEFAULT_WORKLOAD_CLASS
0x000001D7	EFS_FATAL_ERROR
0x000001D8	UCMUCSI_FAILURE
0x000001D9	HAL_IOMMU_INTERNAL_ERROR
0x000001DA	HAL_BLOCKED_PROCESSOR_INTERNAL_ERROR
0x000001DB	IPI_WATCHDOG_TIMEOUT
0x000001DC	DMA_COMMON_BUFFER_VECTOR_ERROR
0x000001DD	BUGCODE_MBBADAPTER_DRIVER
0x000001DE	BUGCODE_WIFIADAPTER_DRIVER
0x000001DF	PROCESSOR_START_TIMEOUT
0x000001E4	VIDEO_DXGKRNL_SYSMM_FATAL_ERROR
0x000001E9	ILLEGAL_ATS_INITIALIZATION
0x000001EA	SECURE_PCI_CONFIG_SPACE_ACCESS_VIOLATION
0x000001EB	DAM_WATCHDOG_TIMEOUT
0x000001ED	HANDLE_ERROR_ON_CRITICAL_THREAD
0x00000356	XBOX_ERACTRL_CS_TIMEOUT
0x00000BFE	BC_BLUETOOTH_VERIFIER_FAULT
0x00000BFF	BC_BTHMINI_VERIFIER_FAULT
0x00020001	HYPERVISOR_ERROR
0x1000007E	SYSTEM_THREAD_EXCEPTION_NOT_HANDLED_M
0x1000007F	UNEXPECTED_KERNEL_MODE_TRAP_M
0x1000008E	KERNEL_MODE_EXCEPTION_NOT_HANDLED_M
0x100000EA	THREAD_STUCK_IN_DEVICE_DRIVER_M

代码	名称
0x4000008A	THREAD_TERMINATE_HELD_MUTEX
0xC0000218	STATUS_CANNOT_LOAD_REGISTRY_FILE
0xC000021A	WINLOGON_FATAL_ERROR
0xC0000221	STATUS_IMAGE_CHECKSUM_MISMATCH
0xDEADDEAD	MANUALLY_INITIATED_CRASH1

另请参阅

错误检查（蓝屏）

分析错误检查蓝屏数据

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

错误检查 0x1：APC_INDEX_MISMATCH

项目 • 2024/10/16

APC_INDEX_MISMATCH 错误检查的值为 0x00000001。 错误检查表明异步过程调用 (APC) 状态索引不匹配。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是 Microsoft 客户，并且你的计算机显示蓝屏错误代码，请参阅**对蓝屏错误进行故障排除**。

APC_INDEX_MISMATCH 参数

展开表

参数	说明
1	系统函数（系统调用）或辅助角色例程的地址。
2	当前线程的 **ApcStateIndex** 字段的值。
3	当前线程的 **CombinedApcDisable** 字段的值。 此字段由两个单独的 16 位字段组成：（*线程*>**SpecialApcDisable**<< 16）| *线程*> **KernelApcDisable**。
4	调用类型： 0 - 系统调用 1 - 辅助角色例程

原因

此错误检查的最常见原因是文件系统或驱动程序禁用和重新启用 APC 的调用顺序不匹配。 关键数据项是 *Thread*>**CombinedApcDisable** 字段。 **CombinedApcDisable** 字段由两个单独的 16 位字段组成：**SpecialApcDisable** 和 **KernelApcDisable**。 任一字段的负值表示驱动程序已分别禁用特殊或普通 APC，而没有重新启用它们。 正值表示驱动程序启用特殊或普通 APC 的次数过多。

解决方法

可以使用 WinDbg 或基本故障排除技术解决此问题。

使用 WinDbg 进行调试

!analyze 调试程序扩展显示有关错误检查的信息，并有助于确定根本原因。

可以使用 !apc 扩展显示一个或多个 APC 内容。

还可以在此停止代码之前的代码中设置断点，并尝试单步前进到故障代码中。

有关使用 WinDbg 的详细信息，请参阅使用 Windows 调试程序 (WinDbg) 进行故障转储分析。

不使用 WinDbg 进行调试

如果您不具备使用 Windows 调试器来解决此问题的能力，那么：

- 在事件查看器中检查系统日志以获取更多错误消息，这可能有助于识别导致此错误检查的设备或驱动程序。
- 如果在错误检查消息中标识了驱动程序，请禁用该驱动程序，或与制造商联系以获取驱动程序更新。
- 确保安装的任何新硬件都与已安装的 Windows 版本兼容。 例如，可以在 Windows 10 规范 中获取所需硬件的相关信息。

有关一般故障排除详细信息，请参阅分析错误检查蓝屏数据。

注解

此错误检查是内核中内部错误导致的结果。 退出系统调用时会出现此错误。 此错误检查的可能原因是文件系统或驱动程序在进入或离开受保护区域或关键区域时的系统调用顺序不匹配。 例如，对 KeEnterCriticalRegion 的每次调用都必须具有对 KeLeaveCriticalRegion 的匹配调用。

如果您正在开发驱动程序，可以使用静态驱动程序验证程序（Windows 驱动程序工具包中提供的静态分析工具）在交付驱动程序之前检测代码中的问题。 使用 CriticalRegions 规则运行静态驱动程序验证程序，以验证源代码是否按正确的顺序使用这些系统调用。

另请参阅

错误检查代码参考

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

错误检查 0x2：DEVICE_QUEUE_NOT_BUSY

项目 • 2024/10/16

DEVICE_QUEUE_NOT_BUSY 错误检查的值为 0x00000002。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

错误检查 0x3：INVALID_AFFINITY_SET

项目 • 2024/02/22

INVALID_AFFINITY_SET 错误检查的值为 0x00000003。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x4：INVALID_DATA_ACCESS_TRAP

项目 • 2024/02/22

INVALID_DATA_ACCESS_TRAP 错误检查的值为 0x00000004。 它表示数据访问陷阱不正确。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

Bug 检查 0x5：INVALID_PROCESS_ATTACH_ATTEMPT

项目 • 2024/02/20

INVALID_PROCESS_ATTACH_ATTEMPT Bug 检查的值为 0x00000005。 这通常表示线程在不允许的情况下附加到进程。 例如，如果在线程已附加到进程（这是非法的），或者线程从处于附加状态的某些函数调用中返回（无效）时调用 **KeAttachProcess**，可能会出现此 Bug 检查

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INVALID_PROCESS_ATTACH_ATTEMPT 参数

展开表

参数	说明
1	指向目标进程的调度程序对象的指针；如果线程已附加，则为指向原始进程的对象的指针。
2	指向当前线程当前附加到的进程调度程序对象的指针。
3	线程的 APC 状态索引的值。
4	非零值表示 DPC 正在当前处理器上运行。

注解

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

如果驱动程序调用 KeAttachProcess 函数并且线程已附加到另一个进程，则会发生此 Bug 检查。 最好使用 KeStackAttachProcess 函数。 如果当前线程已附加到另一个进程，**KeStackAttachProcess** 函数则会在将当前线程附加到新进程之前保存当前 APC 状态。 错

误地调用 **KeStackAttachProcess** 也会导致此 Bug 检查，例如，如果 DPC 正在当前处理器上运行。

有关此区域的一般信息，请参阅使用 Windows 内核模式进程和线程管理器以及内核调度程序对象简介。

Bug 检查 0x6：INVALID_PROCESS_DETACH_ATTEMPT

项目 • 2024/02/21

INVALID_PROCESS_DETACH_ATTEMPT Bug 检查的值为 0x00000006。

此 Bug 检查很少出现。 此 Bug 检查可能是由于调用 KeStackAttachProcess 例程，随后在驱动程序的 KeUnstackDetachProcess 回调函数实现中调用 PLOAD_IMAGE_NOTIFY_ROUTINE 引起的。 回调在加载图像的进程的线程中运行。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x7：INVALID_SOFTWARE_INTERRUPT

项目 • 2024/02/20

INVALID_SOFTWARE_INTERRUPT Bug 检查的值为 0x00000007。

此 Bug 检查很少出现。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x8：IRQL_NOT_DISPATCH_LEVEL

项目 • 2024/02/20

IRQL_NOT_DISPATCH_LEVEL Bug 检查的值为 0x00000008。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

Bug 检查 0x9：IRQL_NOT_GREATER_OR_EQUAL

项目 • 2024/02/20

IRQL_NOT_GREATER_OR_EQUAL Bug 检查的值为 0x00000009。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

错误检查 0xA：IRQL_NOT_LESS_OR_EQUAL

项目 • 2024/10/16

IRQL_NOT_LESS_OR_EQUAL 错误检查的值为 0x0000000A。 此错误检查表明在提高中断请求级别 (IRQL) 时，Microsoft Windows 或内核模式驱动程序访问了无效地址的分页内存。 原因通常是指针错误或分页性问题。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

IRQL_NOT_LESS_OR_EQUAL 参数

展开表

参数	说明
1	无法访问的虚拟内存地址。 使用此地址上的 !pool 查看它是否为分页池。 在收集有关失败的信息时，可能有用的其他命令是 !pte、!address 和 ln （列出最近符号）。
2	失败时的 IRQL。 **值**: 2 - 发生故障时，IRQL 为 DISPATCH_LEVEL。
3	描述导致错误的操作的位域。 请注意，位 3 仅在支持此报告级别的芯片集上可用。 **位 0 值**: 0 - 读取操作 1 - 写入操作 **位 3 值**: 0 - 非执行操作 1 - 执行操作 **位 0 和位 3 组合值**: 0x0 - 尝试从参数 1 中的地址读取时出错 0x1 - 尝试写入参数 1 中的地址时出错

参数	说明
	0x8 - 尝试从参数 1 中的地址执行代码时出错 此值源于： • 调用在 DISPATCH_LEVEL 时无法在 DISPATCH_LEVEL 调用的函数。 • 忘记释放自旋锁。 • 在代码必须为不可分页时将其标记为可分页。 例如，如果代码获取了自旋锁，或在延迟过程调用中被调用。
4	发生故障时的指令指针。 使用此地址上的 ln（列出最近符号） 命令查看函数名称。

原因

此错误检查是由于使用不当地址的内核模式设备驱动程序引起的。

此错误检查表示在中断请求级别 (IRQL) 升高时尝试访问无效地址。 原因是内存指针错误或设备驱动程序代码的分页性问题。

可用于对导致错误检查的编码错误类型进行分类的一般准则如下所示：

- 如果参数 1 小于0x1000，则问题可能是 NULL 指针的取消引用。
- 如果 !pool 报告参数 1 是分页池（或其他类型的可分页内存），则 IRQL 太高，无法访问此数据。 在较低的 IRQL 上运行，或分配非分页池中的数据。
- 如果参数 3 表明错误检查试图执行可分页代码，则 IRQL 太高，无法调用此函数。 在较低的 IRQL 上运行，或者不将代码标记为可分页。
- 这可能是由于释放后使用或位翻转而导致的指针错误。 使用 !pte、!address 和 ln（列出最近符号）调查参数 1 的有效性。

解决方法

如果内核调试器可用，请获取堆栈跟踪。 首先运行 !analyze 调试器扩展以显示有关错误检查的信息。 **!analyze** 扩展有助于确定根本原因。 接下来，输入其中一个 k* （显示堆栈回溯） 命令以查看调用堆栈。

收集信息

检查驱动程序的名称是否列在蓝屏上。

在事件查看器中检查系统日志以获取其他错误消息，这可能有助于发现导致错误的设备或驱动程序。 在系统日志中查找与蓝屏同时出现的严重错误。

驱动程序验证程序

驱动程序验证程序是一个实时运行的工具，用于检查驱动程序的行为。 例如，驱动程序验证程序检查内存资源（如内存池）的使用。 如果在执行驱动程序代码时标识错误，它会主动创建一个异常，以允许进一步检查该部分驱动程序代码。 驱动程序验证程序管理器内置于 Windows 中，可在所有 Windows 电脑上使用。

若要启动驱动程序验证程序管理器，请在命令提示符下键入 **verifier**。 你可以配置要验证的驱动程序。 验证驱动程序的代码在运行时会增加开销，因此请尝试验证尽可能少的驱动程序。 有关详细信息，请参阅驱动程序验证程序。

下方代码显示了一个调试示例：

dbgcmd

```
kd> .bugcheck       [Lists bug check data.]
Bugcheck code 0000000a
Arguments 00000000 0000001c 00000000 00000000

kd> kb [Lists the stack trace.]
ChildEBP RetAddr  Args to Child
8013ed5c 801263ba 00000000 00000000 e12ab000 NT!_DbgBreakPoint
8013eecc 801389ee 0000000a 00000000 0000001c NT!_KeBugCheckEx+0x194
8013eecc 00000000 0000000a 00000000 0000001c NT!_KiTrap0E+0x256
8013ed5c 801263ba 00000000 00000000 e12ab000
8013ef64 00000246 fe551aa1 ff690268 00000002 NT!_KeBugCheckEx+0x194

kd> kv [Lists the trap frames.]
ChildEBP RetAddr  Args to Child
8013ed5c 801263ba 00000000 00000000 e12ab000 NT!_DbgBreakPoint (FPO:
[0,0,0])
8013eecc 801389ee 0000000a 00000000 0000001c NT!_KeBugCheckEx+0x194
8013eecc 00000000 0000000a 00000000 0000001c NT!_KiTrap0E+0x256 (FPO: [0,0]
TrapFrame @ 8013eee8)
8013ed5c 801263ba 00000000 00000000 e12ab000
8013ef64 00000246 fe551aa1 ff690268 00000002 NT!_KeBugCheckEx+0x194

kd> .trap 8013eee8 [Gets the registers for the trap frame at the time of the
fault.]
eax=dec80201 ebx=ffdff420 ecx=8013c71c edx=000003f8 esi=00000000
edi=87038e10
eip=00000000 esp=8013ef5c ebp=8013ef64 iopl=0         nv up ei pl nz na pe
nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00010202
```

```
ErrCode = 00000000
00000000 ?????????????    [The current instruction pointer is NULL.]

kd> kb       [Gives the stack trace before the fault.]
ChildEBP RetAddr  Args to Child
8013ef68 fe551aa1 ff690268 00000002 fe5620d2 NT!_DbgBreakPoint
8013ef74 fe5620d2 fe5620da ff690268 80404690
NDIS!_EthFilterIndicateReceiveComplete+0x31
8013ef64 00000246 fe551aa1 ff690268 00000002
elnkii!_ElnkiiRcvInterruptDpc+0x1d0
```

注解

在安装有故障的设备驱动程序、系统服务或 BIOS 后，通常会出现产生此错误检查的错误。

如果在升级到较新版本的 Windows 时遇到错误检查 0xA，该错误可能是由与新版本不兼容的设备驱动程序、系统服务、病毒扫描程序或备份工具引起的。

解决硬件故障问题：如果系统中最近添加了新硬件，请将其删除并查看错误是否再次出现。 如果现有硬件出现故障，请卸下或更换故障部件。 运行系统制造商提供的硬件诊断程序。 有关这些过程的详细信息，请参阅计算机的用户手册。

解决故障系统服务问题：禁用服务并确认该措施否可以解决错误。 如果可以，请联系系统服务的制造商以了解可能的更新。 如果在系统启动期间发生错误，请调查 Windows 修复选项。 有关详细信息，请参阅 Windows 10 中的恢复选项。

解决防病毒软件问题：禁用程序并确认该措施是否可以解决错误。 如果是，请与程序制造商联系，了解可能的更新。

有关对错误检查进行故障排除的一般信息，请参阅分析错误检查蓝屏数据。

另请参阅

错误检查代码参考

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Bug 检查 0xB：NO_EXCEPTION_HANDLING_SUPPORT

项目 • 2024/02/20

NO_EXCEPTION_HANDLING_SUPPORT Bug 检查的值为 0x0000000B。

此 Bug 检查很少出现。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

另请参阅

Bug 检查代码参考

错误检查 0xC：MAXIMUM_WAIT_OBJECTS_EXCEEDED

项目 • 2024/03/04

MAXIMUM_WAIT_OBJECTS_EXCEEDED 错误检查的值为 0x0000000C。 这表示当前线程超过了允许的等待对象数。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

MAXIMUM_WAIT_OBJECTS_EXCEEDED 参数

无

原因

此错误检查是由于不正确使用 **KeWaitForMultipleObjects** 或 **FsRtlCancellableWaitForMultipleObjects** 造成的。

调用方将指针传递到此例程的 *WaitBlockArray* 参数中的缓冲区。 系统将使用此缓冲区跟踪等待对象。

如果提供了缓冲区，则 *Count* 参数不能超过 MAXIMUM_WAIT_OBJECTS。 如果未提供缓冲区，则 *Count* 参数不能超过 THREAD_WAIT_OBJECTS。

如果 *Count* 的值超过允许的值，则会发出此错误检查。

错误检查 0xD：MUTEX_LEVEL_NUMBER_VIOLATION

项目 • 2024/03/04

MUTEX_LEVEL_NUMBER_VIOLATION 错误检查的值为 0x0000000D。

这种错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

错误检查 0xE：NO_USER_MODE_CONTEXT

项目 • 2024/03/04

NO_USER_MODE_CONTEXT 错误检查的值为 0x0000000E。

在启动系统线程的过程中，如果控制从初始线程过程返回，则会发生错误检查。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0xF：SPIN_LOCK_ALREADY_OWNED

项目 • 2024/03/04

SPIN_LOCK_ALREADY_OWNED 错误检查的值为 0x0000000F。 这表示在已拥有自旋锁的情况下启动了对自旋锁的请求。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SPIN_LOCK_ALREADY_OWNED 参数

无

原因

通常，此错误是由对自旋锁的递归请求引起的。 如果启动了类似于自旋锁的递归请求，也可能发生这种情况，例如，当线程获取了自旋锁，然后该线程调用了一个函数，该函数也试图获取自旋锁。 在这种情况下，获取自旋锁的第二次尝试不会被阻止，因为这样做会导致不可恢复的死锁。 如果对多个处理器进行调用，则一个处理器将被阻止，直到另一个处理器释放锁为止。

在没有显式递归的情况下，当所有线程和所有自旋锁都分配了 IRQL 时，也可能发生此错误。 自旋锁 IRQL 始终大于或等于 DPC 级别，但对于线程来说并非如此。 但是，持有自旋锁的线程必须维护大于或等于自旋锁的 IRQL。 将线程 IRQL 降低到其持有的自旋锁的 IRQL 级别以下，可以在处理器上调度另一个线程。 然后，此新线程会尝试获取相同的自旋锁。

解决方法

确保没有递归地获取锁。 并且，对于持有自旋锁的线程，确保没有将线程的 IRQL 降低到低于它所持有的自旋锁的 IRQL 的级别。

bug 检查0x10：SPIN_LOCK_NOT_OWNED

项目 • 2024/02/14

SPIN_LOCK_NOT_OWNED bug 检查的值为 0x00000010。

此 bug 检查很少出现。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

Bug 检查 0x11: THREAD_NOT_MUTEX_OWNER

项目 • 2024/03/01

THREAD_NOT_MUTEX_OWNER Bug 检查的值为 0x00000011。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

错误检查 0x12：TRAP_CAUSE_UNKNOWN

项目 • 2024/10/16

The TRAP_CAUSE_UNKNOWN 错误检查的值为 0x00000012。 这表示发生了未知异常。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

TRAP_CAUSE_UNKNOWN 参数

展开表

参数	说明
1	TRAP_CAUSE_UNKNOWN 的类型 **VALUES** 1 - 意外中断。 （参数 2 – 中断向量） 2 - 未知浮点异常。 3 - 启用和断言的状态位（请参阅处理器定义）。
2	依赖于 Arg1
3	预留
4	Reserved

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

开始时，使用 **k、kb、kc、kd、kp、kP、kv（显示堆栈回溯）**命令检查堆栈跟踪。 可以指定处理器编号以检查所有处理器上的堆栈。

您还可以在导致此停止代码的代码中设置断点，并尝试单步执行故障代码。

!idt 扩展可用于显示指定中断调度表 (IDT) 的中断服务例程 (ISR)。

调试中断 Storm 中介绍的一些技术可用于解决意外中断。

有关使用故障转储的一般信息，请参阅使用 Windows 调试器进行故障转储分析 (WinDbg)。

如果您不具备使用 Windows 调试器来解决此问题的能力，则可以使用一些基本的故障排除技术。

- 在事件查看器中检查系统日志以获取其他错误消息，这可能有助于识别导致此错误检查的设备或驱动程序。
- 如果在错误检查消息中标识了驱动程序，请禁用该驱动程序，或与制造商联系以获取驱动程序更新。
- 确保安装的任何新硬件都与已安装的 Windows 版本兼容。 例如，可以在 Windows 10 规范 中获取所需硬件的相关信息。
- 有关其他常规故障排除信息，请参阅**分析错误检查蓝屏数据**。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

bug 检查 0x13：EMPTY_THREAD_REAPER_LIST

项目 • 2024/02/21

EMPTY_THREAD_REAPER_LIST bug 检查的值为 0x00000013。

Windows 当前未使用此 bug 检查。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

bug 检查 0x14：CREATE_DELETE_LOCK_NOT_LOCKED

项目 • 2024/02/23

CREATE_DELETE_LOCK_NOT_LOCKED bug 检查的值为 0x00000014。

此 bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

bug 检查 0x15：LAST_CHANCE_CALLED_FROM_KMODE

项目 • 2024/02/23

LAST_CHANCE_CALLED_FROM_KMODE bug 检查的值为 0x00000015。

此 bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

bug 检查 0x16：CID_HANDLE_CREATION

项目 • 2024/02/21

CID_HANDLE_CREATION bug 检查的值为 0x00000016。

此 bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

bug 检查 0x17：CID_HANDLE_DELETION

项目 • 2024/02/21

CID_HANDLE_DELETION bug 检查的值为 0x00000017。

此 bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

错误检查 0x18：REFERENCE_BY_POINTER

项目 • 2024/10/16

REFERENCE_BY_POINTER 错误检查的值为 0x00000018。 这表示对象的引用计数对于对象的当前状态是非法的。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅蓝屏错误疑难解答。

REFERENCE_BY_POINTER 参数

展开表

参数	说明
1	正在降低其引用计数的对象的对象类型。
2	正在降低其引用计数的对象。
3	预留
4	预留

原因

对象的引用计数对于对象的当前状态是非法的。 每当驱动程序使用指向对象的指针时，驱动程序都会调用内核例程，以将对象的引用计数加一。 使用指针完成驱动程序后，驱动程序将调用另一个内核例程，以将引用计数减一。

驱动程序必须与对增加引用计数（*引用*）和减少（*取消引用*）的例程的调用匹配。 此错误检查是由对象的引用计数不一致引起的。 通常，不一致是由驱动程序导致的，该驱动程序会减少对象的引用计数过多次，从而发出额外的调用来取消引用对象。 可能会发生此错误检查的原因是对象的引用计数在对象仍有打开句柄时将变为零。 当对象的引用计数下降到零以下时，也可能会发生这种情况，而无论对象是否有打开的句柄。

解决方法

确保驱动程序匹配会调用增加和减少对象的引用计数的例程。 确保驱动程序不会对取消引用对象的例程进行额外的调用（请参阅参数 2）。

可以使用调试器来帮助分析此问题。 有关详细信息，请参阅使用 Windows 调试程序 (WinDbg) 进行故障转储分析。 **!analyze** 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

若要查找对象的句柄和指针计数，请使用 **!object** 调试器命令。

kd> !object *地址*

其中 *地址* 是参数 2 中给定对象的地址。

您还可以在导致此停止代码的代码中设置断点，并尝试单步执行故障代码。

如果你不具备使用 Windows 调试器来解决此问题的能力，则可以使用一些基本的故障排除技术。

- 在事件查看器中检查系统日志以获取其他错误消息，这可能有助于识别导致此错误检查的设备或驱动程序。
- 如果在错误检查消息中标识了驱动程序，请禁用该驱动程序，或与制造商联系以获取驱动程序更新。
- 确保安装的任何新硬件都与已安装的 Windows 版本兼容。 例如，可以在 Windows 10 规范 中获取所需硬件的相关信息。
- 有关其他常规故障排除信息，请参阅**分析错误检查蓝屏数据**。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

错误检查 0x19：BAD_POOL_HEADER

项目 • 2024/10/16

BAD_POOL_HEADER 错误检查的值为 0x00000019。 这表示池标头已损坏。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

BAD_POOL_HEADER 参数

参数 1 表示违规类型。 其他参数的含义取决于参数 1 的值。

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x2	正在检查的池条目	池块的大小	0	特殊池模式检查失败。 （所有者可能损坏了池块。）
0x3	正在检查的池条目	读回 **flink** 自由列表值	读回 **blink** 自由列表值	池自由列表已损坏。 （在正常运行的列表中，参数 2、3 和 4 的值应相同。）
0x5	其中一个池条目	Reserved	另一个池条目	一对相邻池条目具有相互矛盾的标头。 其中至少有一个损坏。
0x6	一个错误计算的条目	Reserved	导致错误计算的错误条目	池块标头以前的大小太大。
0x7	0	保留	错误池条目	池块标头大小已损坏。
0x8	0	保留	错误池条目	池块标头大小为零。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x9	一个错误计算的条目	Reserved	导致错误计算的错误条目	池块标头大小已损坏（太大）。
0xA	应找到的池条目	Reserved	应包含池条目的页面的虚拟地址	池块标头大小已损坏。
0xD、0xE、0xF、0x23、0x24、0x25	Reserved	预留	保留	释放后已修改释放块的池标头。这通常不是释放块的先前所有者的错误；相反，这通常是（但并非总是）由于释放块之前释放的块已溢出。
0x20	应找到的池条目	下一个池条目	Reserved	池块标头大小已损坏。
0X21	正在释放的池指针	为池块分配的字节数	在池块后找到的损坏值	正在释放池块后的数据已损坏。通常，这意味着使用者（调用堆栈）已溢出块。
0X22	正在释放的地址	Reserved	保留	正在释放的地址没有跟踪条目。这通常是因为调用堆栈尝试释放已释放或从未分配以开始的指针。

原因

当前请求时池已损坏。

这可能是由于调用方造成的。

解决方法

必须使用内核调试器来演练内部池链接，才能找出问题可能的原因。

然后，可以对可疑池标记使用特殊池，或者在可疑驱动程序上使用驱动程序验证程序"特殊池"选项。 !analyze 扩展可能有助于查明可疑驱动程序，但这通常不是池损坏者的情

况。

使用**分析错误检查蓝屏数据**中所述的步骤收集停止代码参数。 使用停止代码参数来确定要跟踪的特定类型的代码行为。

驱动程序验证程序

驱动程序验证程序是一个实时运行的工具，用于检查驱动程序的行为。 如果它在驱动程序代码的执行中发现错误，则会主动创建一个异常，以允许进一步检查驱动程序代码的该部分。 驱动程序验证程序管理器内置于 Windows 中，可在所有 Windows PC 上使用。 若要启动驱动程序验证程序管理器，请在命令提示下键入“验证程序”。 你可以配置要验证的驱动程序。 验证驱动程序的代码在运行时会增加开销，因此请尝试验证尽可能少的驱动程序。 有关详细信息，请参阅驱动程序验证程序。

Windows 内存诊断

如果此错误检查不一致，则可能与物理内存故障相关。

运行 Windows 内存诊断工具以测试内存。 在控制面板搜索框中键入“内存”，然后选择**诊断计算机内存问题**。运行测试后，使用事件查看器查看系统日志下的结果。 查找“内存诊断结果”条目以查看结果。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

错误检查 0x1A：MEMORY_MANAGEMENT

项目 • 2024/02/22

MEMORY_MANAGEMENT 错误检查的值为 0x0000001A。 错误检查表明发生了严重的内存管理错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是 Microsoft 客户，并且你的计算机显示蓝屏错误代码，请参阅**对蓝屏错误进行故障排除**。

MEMORY_MANAGEMENT 参数

参数 1 标识完全冲突。

展开表

参数 1	错误消息的原因
0x31	映像重新定位修复表或代码流已损坏。 该错误可能是一个硬件错误。
0x3f	由于循环冗余校验 (CRC) 错误，一个页内操作失败。 参数 2 包含页面文件偏移量。 参数 3 包含页 CRC 值。 参数 4 包含预期的 CRC 值。
0x403	页表和页帧编号 (PFN) 不同步。此错误可能是一个硬件错误，尤其是在参数 3 和 4 仅相差一位时。
0x404	在删除系统页的过程中，发现 PFN 与当前页表条目 (PTE) 指针不一致。 参数 2 是预期的 PTE。 参数 3 是 PTE 内容。 参数 4 是 PFN 的 PTE。
0x411	PTE 已损坏。 参数 2 是 PTE 的地址。
0x1010	调用方正在解锁当前未锁定的可分页部分。 （此部分要么从未锁定，要么已被解锁两次。）
0x1233	驱动程序试图映射未锁定的物理内存页。 此操作是非法的，因为页面的内容或属性可能随时更改。 代码中的一个错误导致了映射调用。 参数 2 是驱动程序尝试映射的物理页的 PFN。
0x1234	调用方正在尝试锁定一个不存在的可分页部分。
0x1235	调用方尝试用无效映射保护 MDL。

参数 1	错误消息的原因
0x1236	调用方指定了包含未锁定（或无效）物理页的 MDL。 参数 2 包含指向 MDL 的指针。 参数 3 包含指向无效 PFN 的指针。 参数 4 包含无效的 PFN 值。
0x1240	调用方为非常驻虚拟地址范围建立 MDL 是非法的。 参数 2 是 MDL。 参数 3 是 PTE 指针。
0x3300	在写入操作期间，引用的虚拟地址被错误地标记为“写入时复制”。 参数 2 是 `FaultingAddress` 的值。 参数 3 是 PTE 内容。 参数 4表 示虚拟地址空间类型。
0x3451	已调出的内核线程堆栈的 PTE 已损坏。
0x3453	由于未完成的引用，无法删除已退出进程的所有页表页。 此错误通常表示进程页表结构已损坏。
0x3470	缓存的内核堆栈在空闲列表中时已损坏。 此内存损坏表明问题很严重，调用堆栈可能是受害者，也可能是罪魁祸首。 参数 2 是虚拟地址。 参数 3 是虚拟地址 Cookie。
0x4477	驱动程序尝试写入系统进程的用户空间中未分配的地址。 参数 2 包含尝试写入的地址。
0x5100	分配位图已损坏。 内存管理器即将覆盖已在使用的虚拟地址。
0x5305	调用方正在指定要释放的无效池地址（参数 2）。 参数 2 是正在评估的虚拟地址。 参数 3 是区域大小。
0x6001	内存存储组件的专用内存范围已损坏，因此无法访问。 参数 2 是返回的状态。 参数 3 是存储的专用内存范围内的虚拟地址。 参数 4 是 MDL。
0x8886 0x8887	（Windows 7 及更高版本）。 备用列表中的两个页面本应具有相同的页面优先级值，但却没有相同的页面优先级值。 参数 4 中捕获了不同的值。
0x8888 0x8889	内部内存管理结构已损坏。
0x888A	内部内存管理结构（可能是 PTE 或 PFN）已损坏。
0x9696	遇到一个 PFN（参数 2），其链接已损坏，不再与其顶层进程相连。 此错误表明 PFN 结构已损坏。
0x15000	调用方要么提供了错误的地址，要么在错误的进程上下文中调用了此例程。 这两个操作都是非法的，因为我们不能取消保护由于这个错误而无法找到的范围。 参数 2 是正在评估的虚拟地址。
0x15001	取消保护以前已保护的内存的过程中发生错误。 如果调用方在错误的进程上下文中错误地调用了 `MmUnsecureVirtualMemory`，则可能发生此错误。
0x41202	在确定非零 PTE 的页面保护过程中，确定 PTE 已损坏。 参数 2 是 PTE 指针。 参数 3 是 PTE 内容。 参数 4 是虚拟地址描述符。

参数 1	错误消息的原因
0x41286	调用方试图释放无效的池地址。
0x41785	工作集列表已损坏。
0x41287	保持工作集同步时出现非法页错误。 参数 2 包含引用的虚拟地址。
0x41790	页表页已损坏。 在 64 位版本的 Windows 上，参数 2 包含损坏页表页的 PFN 地址。 在 32 位版本的 Windows 上，参数 2 包含指向已使用 PTE 数的指针，参数 3 包含已使用 PTE 数。
0x41792	检测到损坏的 PTE。 参数 2 包含 PTE 的地址。 参数 3 和 4 包含 PTE 的低和高部分。
0x61941	分页层次结构已损坏。 参数 2 是指向导致故障的虚拟地址的指针。
0x61948	在递减 I/O 空间区域的引用计数过程中，无法找到该区域的记帐节点。 通常，此错误意味着参数范围从未锁定或已经解锁。 参数 2 是基本 I/O 帧。 参数 3 是区域内的页数。 参数 4 是找不到节点的特定 I/O 帧。
0x61949	`IoPageFrameNode` 值为 null。 参数 2 为 `PageFrameIndex`。
0x6194A	减少正在取消映射的 I/O 空间物理页上的引用计数时出错。 正在取消引用当前未被引用的条目。 参数 2 和 3 描述了调用方正在取消映射的 I/O 空间范围。 参数 4 是应引用的 I/O 空间物理页，但尚未引用该页。
0x03030308	要删除（或截断）的范围已被加载程序使用。 它无法被安全移除，因此系统必须发出停止代码。 参数 2 为 `HighestPhysicalPage`。

这些参数 1 值用于以前的 Windows 版本。

展开表

参数 1	错误消息的原因
0x777	调用方正在解锁当前未锁定的系统缓存地址。 （该地址要么从未映射过，要么被解锁了两次。）
0x778	系统正在使用最后一个系统缓存视图地址，而不是保留它。
0x780 0x781	映射参数系统缓存视图的 PTE 已损坏。
0x1000	`MmGetSystemAddressForMdl*` 的调用方试图将完全缓存的物理页面映射为非缓存页面。 该操作会导致硬件转换缓冲区条目冲突，因此已被操作系统拒绝。 由于调用方在请求的内存描述符列表 (MDL) 中指定了“故障时进行错误检查”，因此系统发出了一个错误检查。
0x1241	在调用生成 MDL 期间，MDL 的虚拟地址意外异步取消映射。 参数 2 是 MDL。 参

参数 1	错误消息的原因
	数 3 是 PTE 指针。
0x5003	工作集空闲列表已损坏。 这可能是硬件错误。
0x5200	可用池列表上的页已损坏。 此错误可能是驱动程序中的 write-after-free 错误或上一页溢出造成的。 参数 2 包含可用池块的地址。 参数 4 包含预期位于该地址的值。 参数 3 包含找到的实际值。
0x8884 0x8885	（Windows 7 及更高版本）。 备用列表中的两个页面本应具有相同的页面优先级值，但却没有相同的页面优先级值。 参数 4 中捕获了不同的值。
0x41201	在查询虚拟地址的过程中，发现 PFN 与当前 PTE 指针不一致。 参数 2 是对应的 PTE。 参数 3 是 PTE 内容。 参数 4 是虚拟地址描述符。
0x41283	PTE 中编码的工作集索引已损坏。
0x41284	PTE 或工作集列表已损坏。
0x41793	页表页已损坏。 参数 2 包含指向上次处理的 PTE 的指针。 参数 3 包含找到的非零 PTE 数。 参数 4 包含页表中非零 PTE 的预期数量。 此内存参数已被弃用，在 Windows 10 1803 版本之后不再可用。
0x61940	PDE 意外失效。
0x61946	正在创建的 MDL 存在缺陷。 此错误几乎总是意味着驱动程序调用 `MmProbeAndLockPages` 存在问题。 通常情况下，在被要求处理分页读取时，驱动程序会尝试创建写入 MDL。
0x03030303	启动加载器已损坏。 （此值仅适用于 Intel Itanium 计算机。）

解决方法

!analyze 调试扩展会显示有关错误检查的信息。 调试扩展中的信息可能有助于确定根本原因。

运行 Windows 内存诊断工具来检查影响物理内存模块的问题可能也会有所帮助。

另请参阅

错误检查代码参考

错误检查 0x1B：PFN_SHARE_COUNT

项目 • 2024/10/16

PFN_SHARE_COUNT 错误检查的值为 0x0000001B。

Windows 操作系统内不再使用此错误检查代码。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

另请参阅

错误检查代码参考

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

错误检查 0x1C：PFN_REFERENCE_COUNT

项目 • 2024/02/26

PFN_REFERENCE_COUNT 错误检查的值为 0x0000001C。 这表示检测到引用计数错误。 它可能是由计数器溢出、下溢或对象被释放后又被使用引起的。 检查堆栈以确定故障。 注意：此错误检查代码用于报告多种类型的引用计数错误，不一定与内存管理器页码 (PFN) 有关。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PARAMETERS

展开表

参数	说明
1	未使用
2	未使用
3	未使用
4	未使用

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x1D：NO_SPIN_LOCK_AVAILABLE

项目 • 2024/02/26

NO_SPIN_LOCK_AVAILABLE 错误检查的值为 0x0000001D。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

错误检查 0x1E：KMODE_EXCEPTION_NOT_HANDLED

项目 • 2024/02/22

KMODE_EXCEPTION_NOT_HANDLED 错误检查的值为 0x0000001E。 错误检查表明内核模式应用程序产生了错误处理程序未捕获到的异常。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是 Microsoft 客户，并且你的计算机显示蓝屏错误代码，请参阅**对蓝屏错误进行故障排除**。

KMODE_EXCEPTION_NOT_HANDLED 参数

展开表

参数	说明
1	未处理的异常代码。
2	发生异常的地址。
3	异常记录的异常信息参数 0。
4	异常记录的异常信息参数 0。

原因

若要对此错误检查进行解释，必须确定生成的具体异常。

常见的异常代码包括：

- 0x80000002: STATUS_DATATYPE_MISALIGNMENT

  遇到未对齐的数据引用。

- 0x80000003: STATUS_BREAKPOINT

  没有内核调试器连接到系统时，遇到断点或 ASSERT。

- 0xC0000005: STATUS_ACCESS_VIOLATION

出现内存访问冲突。 （Bug 检查的参数 4 是驱动程序尝试访问的地址。）

有关异常代码的完整列表，请参阅 NTSTATUS 值。 异常代码在 ntstatus.h（Windows 驱动程序工具包中的标头文件）中定义。 有关详细信息，请参阅 Windows 驱动程序工具包中的头文件。

注解

如果不具备调试此问题的能力，可以使用蓝屏数据中描述的一些基本故障排除技巧。 如果在错误检查消息中标识了驱动程序，请禁用该驱动程序，或与制造商联系以获取驱动程序更新。

硬件不兼容

确保安装的任何新硬件都与已安装的 Windows 版本兼容。 例如，可以从 Windows 10 规格 中获取所需硬件的相关信息。

设备驱动程序或系统服务存在故障

设备驱动程序或系统服务故障可能会导致此错误。 硬件问题（如 BIOS 不兼容、内存冲突和 IRQ 冲突）也可能会产生此错误。

如果错误检查消息中按名称列出了某个驱动程序，请禁用或删除该驱动程序。 禁用或删除最近添加的所有驱动程序或服务。 如果在启动过程中发生错误，并且系统分区格式化为 NTFS 文件系统，则可以使用安全模式禁用设备管理器中的驱动程序。

在事件查看器中检查系统日志以获取更多错误消息，这可能有助于识别导致 Bug 检查 0x1E 的设备或驱动程序。 同时运行系统制造商提供的硬件诊断程序，尤其是内存扫描程序。 有关这些故障排除步骤的详细信息，请参阅计算机的用户手册。

出现此消息的错误可能会发生在 Windows 安装过程中首次重启后或安装完成后。 可能导致错误的原因是系统 BIOS 不兼容。 可以通过升级系统 BIOS 版本来解决 BIOS 问题。

解决方法

在调试此问题时，可能会发现很难获得堆栈跟踪。 异常地址（参数 2）应标识导致问题的驱动程序或函数。

异常代码 0x80000003 表示命中了硬编码断点或断言，但系统是使用 `/NODEBUG` 开关启动的。 此问题应该很少发生。 如果重复发生，请确保内核调试器已连接，并且已使用 `/DEBUG` 开关启动系统。

如果出现异常代码 0x80000002，陷阱帧会提供更多信息。

未知原因

如果异常的具体原因未知，可考虑使用以下步骤来获取堆栈跟踪。

> **备注**
>
> 此步骤假定你可以找到 `NT!PspUnhandledExceptionInSystemThread`。 但是，在某些情况下，例如在访问违规崩溃时，你将无法找到 `NT!PspUnhandledExceptionInSystemThread`。 在这种情况下，请查找 `ntoskrnl!KiDispatchException`。 传递给此函数的第三个参数是陷阱帧地址。 使用包含此地址的 **.trap（显示陷阱帧）** 命令将寄存器上下文设置为正确值。 然后就可以进行堆栈跟踪并发出其他命令。

获取堆栈跟踪

如果正常的堆栈跟踪程序失败，则获取堆栈跟踪：

1. 使用 kb（显示堆栈回溯）命令来显示堆栈回溯中的参数。 查找对用 `NT!PspUnhandledExceptionInSystemThread` 的调用。 （如果未列出此函数，请参阅前面的注释。）

2. `NT!PspUnhandledExceptionInSystemThread` 的第一个参数是指向一个结构的指针。 该指针包含指向 `except` 语句的指针：

   ```
   C++

   typedef struct _EXCEPTION_POINTERS {
       PEXCEPTION_RECORD ExceptionRecord;
       PCONTEXT ContextRecord;
       } EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;

   ULONG PspUnhandledExceptionInSystemThread(
       IN PEXCEPTION_POINTERS ExceptionPointers
       )
   ```

   对该地址使用 dd（显示内存）命令以显示所需的数据。

3. 第一个检索值是异常记录。 对于异常记录，请使用 .exr（显示例外记录）命令。

   第二个值是上下文记录。 对于上下文记录，请使用 .cxr（显示上下文记录）命令。

4. 在执行 `.cxr` 命令后，使用 `kb` 命令显示基于上下文记录信息的堆栈跟踪。 该堆栈跟踪指明了发生未处理异常的调用堆栈。

示例错误检查

下面的示例显示了 x86 处理器上的错误检查 0x1E：

dbgcmd

```
kd> .bugcheck                 get the bug check data
Bugcheck code 0000001e
Arguments c0000005 8013cd0a 00000000 0362cffff

kd> kb                        start with a stack trace
FramePtr  RetAddr   Param1   Param2   Param3   Function Name
8013ed5c  801263ba  00000000 00000000 fe40cb00 NT!_DbgBreakPoint
8013eecc  8013313c  0000001e c0000005 8013cd0a NT!_KeBugCheckEx+0x194
fe40cad0  8013318e  fe40caf8 801359ff fe40cb00
NT!PspUnhandledExceptionInSystemThread+0x18
fe40cad8  801359ff  fe40cb00 00000000 fe40cb00
NT!PspSystemThreadStartup+0x4a
fe40cf7c  8013cb8e  fe43a44c ff6ce388 00000000 NT!_except_handler3+0x47
00000000  00000000  00000000 00000000 00000000 NT!KiThreadStartup+0xe

kd> dd fe40caf8 L2            dump EXCEPTION_POINTERS structure
0xFE40CAF8  fe40cd88 fe40cbc4                  ..@...@.

kd> .exr fe40cd88             first DWORD is the exception record
Exception Record @ FE40CD88:
   ExceptionCode: c0000005
  ExceptionFlags: 00000000
  Chained Record: 00000000
ExceptionAddress: 8013cd0a
NumberParameters: 00000002
   Parameter[0]: 00000000
   Parameter[1]: 0362cfff

kd> .cxr fe40cbc4             second DWORD is the context record
CtxFlags: 00010017
eax=00087000 ebx=00000000 ecx=03ff0000 edx=ff63d000 esi=0362cfff
edi=036b3fff
eip=8013cd0a esp=fe40ce50 ebp=fe40cef8 iopl=0         nv dn ei pl nz ac po
cy
vip=0    vif=0
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00010617
0x8013cd0a  f3a4            rep movsb

kd> kb                        kb gives stack for context record
ChildEBP RetAddr  Args to Child
fe40ce54 80402e09 ff6c4000 ff63d000 03ff0000 NT!_RtlMoveMemory@12+0x3e
fe40ce68 80403c18 ffbc0c28 ff6ce008 ff6c4000 HAL!_HalpCopyBufferMap@20+0x49
```

```
fe40ce9c fe43b1e4 ff6cef90 ffbc0c28 ff6ce009
HAL!_IoFlushAdapterBuffers@24+0x148
fe40ceb8 fe4385b4 ff6ce388 6cd00800 ffbc0c28
QIC117!_kdi_FlushDMABuffers@20+0x28
fe40cef8 fe439894 ff6cd008 ffb6c820 fe40cf4c
QIC117!_cqd_CmdReadWrite@8+0x26e
fe40cf18 fe437d92 ff6cd008 ffb6c820 ff6e4e50 QIC117!_cqd_DispatchFRB@8+0x210
fe40cf30 fe43a4f5 ff6cd008 ffb6c820 00000000 QIC117!_cqd_ProcessFRB@8+0x134
fe40cf4c 80133184 ff6ce388 00000000 00000000 QIC117!_kdi_ThreadRun@4+0xa9
fe40cf7c 8013cb8e fe43a44c ff6ce388 00000000
NT!_PspSystemThreadStartup@8+0x40
```

错误检查 0x1F：SHARED_RESOURCE_CONV_ERROR

项目 • 2024/02/22

SHARED_RESOURCE_CONV_ERROR 错误检查的值为 0x0000001F。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

错误检查 0x20：KERNEL_APC_PENDING_DURING_EXIT

项目 • 2024/02/26

KERNEL_APC_PENDING_DURING_EXIT 错误检查的值为 0x00000020。 这表明当线程退出时，异步过程调用 (APC) 仍处于待处理状态。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

KERNEL_APC_PENDING_DURING_EXIT 参数

展开表

参数	说明
1	在退出过程中发现的待处理 APC 的地址
2	线程的 APC 禁用计数
3	当前 IRQL
4	Reserved

原因

关键数据项是线程的 APC 禁用计数（参数 2）。 如果计数不为零，则它将表明问题的来源。

每次驱动程序调用 **KeEnterCriticalRegion**、**FsRtlEnterFileSystem** 或获取互斥体时，APC 禁用计数都会递减。

每次驱动程序调用 **KeLeaveCriticalRegion**、**KeReleaseMutex** 或 **FsRtlExitFileSystem** 时，APC 禁用计数都会递增。

由于这些调用应始终成对进行，因此当线程退出时，APC 禁用计数应为零。 负值表示驱动程序已禁用 APC 调用，而未重新启用它们。 正值表示情况恰好相反。

如果看到此错误，请对计算机上安装的所有驱动程序（尤其是不常见或非标准的驱动程序）保持高度怀疑。

当前 IRQL（参数 3）应为零。 如果不是，则可能是驱动程序的取消例程以较高的 IRQL 返回，从而导致了错误检查。 在这种情况下，请仔细记下崩溃时正在运行的程序（以及正在关闭的程序），并记下崩溃时安装的所有驱动程序。 在这种情况下，原因通常是驱动程序中存在严重的错误。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x21：QUOTA_UNDERFLOW

项目 • 2024/02/27

QUOTA_UNDERFLOW 错误检查的值为 0x00000021。 这表明配额收费处理不当，返回给某个块的配额多于先前收取费用的配额。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

QUOTA_UNDERFLOW 参数

展开表

参数	说明
1	最初收费的流程（如有）。
2	配额类型。 有关所有可能的配额类型值列表，请参阅 Windows 驱动程序工具包 (WDK) 中的头文件 Ps.h。
3	初始收取的配额返还量。
4	未返还的剩余配额量。

错误检查 0x22：FILE_SYSTEM

项目 • 2024/02/22

FILE_SYSTEM 错误检查的值为 0x00000022。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x23：FAT_FILE_SYSTEM

项目 • 2024/02/22

FAT_FILE_SYSTEM 错误检查的值为 0x00000023。 这表明 FAT 文件系统出现了问题。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

FAT_FILE_SYSTEM 参数

展开表

参数	说明
1	指定源文件和行号信息。 高 16 位（“0x”后面的前四个十六进制数字）按标识符号标识源文件。 低 16 位标识文件中出现 Bug 检查的源行。
2	如果 **FatExceptionFilter** 位于堆栈上，则此参数指定异常记录的地址。
3	如果 **FatExceptionFilter** 位于堆栈上，则此参数指定上下文记录的地址。
4	Reserved

原因

造成此错误检查的一个可能原因是磁盘损坏。 磁盘上的文件系统损坏或坏块（扇区）可能会引发此错误。 损坏的 SCSI 和 IDE 驱动程序也会对系统读写磁盘的能力产生不利影响，从而导致错误。

另一个可能的原因是非分页池内存耗尽。 如果非分页池内存完全耗尽，此错误可能会导致系统停止运行。 但是，在索引过程中，如果可用非分页池内存量非常低，则另一个需要非分页池内存的内核模式驱动程序也可能会触发此错误。

解决方法

若要调试此问题：请使用参数 3 的 **.cxr（显示上下文记录）**命令，然后使用 **kb（显示堆栈回溯跟踪）。**。

若要解决磁盘损坏问题：请检查事件查看器查看来自 SCSI 和 FASTFAT（系统日志）或 Autochk（应用程序日志）的错误消息，这些错误消息可能有助于查明导致错误的设备或驱动程序。 尝试禁用任何持续监视系统的病毒扫描程序、备份程序或磁盘碎片整理程序工具。 你还应运行系统制造商提供的硬件诊断。 有关这些过程的详细信息，请参阅计算机的用户手册。 运行 **Chkdsk /f /r**，检测并解决任何文件系统结构损坏问题。 在系统分区上开始磁盘扫描之前，必须重启系统。

若要解决非分页池内存耗尽问题： 向计算机添加新的物理内存。 这将增加内核可用的非分页池内存的数量。

错误检查 0x24：NTFS_FILE_SYSTEM

项目 • 2024/02/26

NTFS_FILE_SYSTEM 错误检查的值为 0x00000024。 这表明 ntfs.sys 出现了问题，该驱动文件允许系统读写 NTFS 驱动器。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

NTFS_FILE_SYSTEM 参数

展开表

参数	说明
1	指定源文件和行号信息。 高 16 位（“0x”后面的前四个十六进制数字）按标识符号标识源文件。 低 16 位标识文件中出现 Bug 检查的源行。
2	如果 **NtfsExceptionFilter** 位于堆栈上，则此参数指定异常记录的地址。
3	如果 **NtfsExceptionFilter** 位于堆栈上，则此参数指定上下文记录的地址。
4	Reserved

原因

造成此错误检查的一个可能原因是磁盘损坏。 硬盘上的 NTFS 文件系统损坏或坏块（扇区）可能会引发此错误。 损坏的硬盘 (SATA/IDE) 驱动程序也会对系统读写磁盘的能力产生不利影响，从而导致错误。

解决方法

若要调试此问题：请使用参数 3 的 **.cxr（显示上下文记录）**命令，然后使用 **kb（显示堆栈回溯跟踪）**。。

要解决磁盘损坏问题，请执行以下操作：

- 检查事件查看器，查看系统日志中是否出现与硬盘驱动器有关的错误信息，这些信息可能有助于确定导致错误的设备或驱动程序。
- 尝试禁用任何持续监视系统的病毒扫描程序、备份程序或磁盘碎片整理程序工具。
- 您还应该运行系统制造商提供的与存储子系统有关的硬件诊断程序。
- 使用扫描磁盘实用工具确认没有文件系统错误。 选择并按住（或右键单击）要扫描的驱动器，然后选择“属性”。 选择**工具**。 选择“立即检查”按钮。
- 确认硬盘驱动器上有足够的可用空间。 操作系统和某些应用程序需要足够的可用空间来创建交换文件和用于其他功能。 确切的要求因系统配置而异，但通常最好有 10% 到 15% 的可用空间。
- 使用系统文件检查器工具修复丢失或损坏的系统文件。 系统文件检查器是 Windows 中的一个实用工具，供用户扫描 Windows 系统文件中的损坏情况并恢复损坏的文件。 使用以下命令运行系统文件检查器工具 (SFC.exe)。

  控制台

  ```
  SFC /scannow
  ```

  有关详细信息，请参阅使用系统文件检查器工具修复丢失或损坏的系统文件。
- **驱动程序验证程序**

  驱动程序验证程序是一个实时运行的工具，用于检查驱动程序的行为。 如果它在驱动程序代码的执行中发现错误，则会主动创建一个异常，以允许进一步检查驱动程序代码的该部分。 驱动程序验证程序管理器内置于 Windows 中，可在所有 Windows PC 上使用。 若要启动驱动程序验证程序管理器，请在命令提示下键入“验证程序”。 你可以配置要验证的驱动程序。 验证驱动程序的代码在运行时会增加开销，因此请尝试验证尽可能少的驱动程序。 有关详细信息，请参阅驱动程序验证程序。

过去，导致该停止代码的另一个可能原因是非分页池内存耗尽。 如果非分页池内存完全耗尽，此错误可能会导致系统停止运行。 但是，在索引过程中，如果可用非分页池内存量非常低，则另一个需要非分页池内存的内核模式驱动程序也可能会触发此错误。

错误检查 0x25：NPFS_FILE_SYSTEM

项目 • 2024/02/26

NPFS_FILE_SYSTEM 错误检查的值为 0x00000025。 这表明 NPFS 文件系统出现了问题。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

NPFS_FILE_SYSTEM 参数

展开表

参数	说明
1	指定源文件和行号信息。 高 16 位（“0x”后面的前四个十六进制数字）按标识符号标识源文件。 低 16 位标识文件中出现 Bug 检查的源行。
2	Reserved
3	Reserved
4	Reserved

原因

造成此错误检查的一个可能原因是非分页池内存耗尽。 如果非分页池内存完全耗尽，此错误可能会导致系统停止运行。 但是，在索引过程中，如果可用非分页池内存量非常低，则另一个需要非分页池内存的内核模式驱动程序也可能会触发此错误。

解决方法

若要解决非分页池内存耗尽问题： 向计算机添加新的物理内存。 这将增加内核可用的非分页池内存的数量。

错误检查 0x26：CDFS_FILE_SYSTEM

项目 • 2024/02/22

CDFS_FILE_SYSTEM 错误检查的值为 0x00000026。 这表明 CD 文件系统出现了问题。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

CDFS_FILE_SYSTEM 参数

展开表

参数	说明
1	指定源文件和行号信息。 高 16 位（“0x”后面的前四个十六进制数字）按标识符号标识源文件。 低 16 位标识文件中出现 Bug 检查的源行。
2	如果 **CdExceptionFilter** 位于堆栈上，则此参数指定异常记录的地址。
3	如果 **CdExceptionFilter** 位于堆栈上，则此参数指定上下文记录的地址。
4	Reserved

原因

造成此错误检查的一个可能原因是磁盘损坏。 磁盘上的文件系统损坏或坏块（扇区）可能会引发此错误。 损坏的 SCSI 和 IDE 驱动程序也会对系统读写磁盘的能力产生不利影响，从而导致错误。

另一个可能的原因是非分页池内存耗尽。 如果非分页池内存完全耗尽，此错误可能会导致系统停止运行。 但是，在索引过程中，如果可用非分页池内存量非常低，则另一个需要非分页池内存的内核模式驱动程序也可能会触发此错误。

解决方法

若要调试此问题：请使用参数 3 的 **.cxr（显示上下文记录）**命令，然后使用 **kb（显示堆栈回溯跟踪）。**。

若要解决磁盘损坏问题：请检查事件查看器查看来自 SCSI 和 FASTFAT（系统日志）或 Autochk（应用程序日志）的错误消息，这些错误消息可能有助于查明导致错误的设备或驱动程序。 尝试禁用任何持续监视系统的病毒扫描程序、备份程序或磁盘碎片整理程序工具。 你还应运行系统制造商提供的硬件诊断。 有关这些过程的详细信息，请参阅计算机的用户手册。 运行 **Chkdsk /f /r**，检测并解决任何文件系统结构损坏问题。 在系统分区上开始磁盘扫描之前，必须重启系统。

若要解决非分页池内存耗尽问题： 向计算机添加新的物理内存。 这将增加内核可用的非分页池内存的数量。

错误检查 0x27：RDR_FILE_SYSTEM

项目 • 2024/02/26

RDR_FILE_SYSTEM 错误检查的值为 0x00000027。 此错误检查表明 SMB 重定向文件系统中出现了问题。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

RDR_FILE_SYSTEM 参数

展开表

参数	说明
1	高 16 位（“0x”后面的前四个十六进制数字）确定问题类型。 可能的值包括： 0xCA550000 RDBSS_BUG_CHECK_CACHESUP 0xC1EE0000 RDBSS_BUG_CHECK_CLEANUP 0xC10E0000 RDBSS_BUG_CHECK_CLOSE 0xBAAD0000 RDBSS_BUG_CHECK_NTEXCEPT
2	如果 **RxExceptionFilter** 位于堆栈上，则此参数指定异常记录的地址。
3	如果 **RxExceptionFilter** 位于堆栈上，则此参数指定上下文记录的地址。
4	Reserved

原因

造成此错误检查的一个可能原因是非分页池内存耗尽。 如果非分页池内存完全耗尽，此错误可能会导致系统停止运行。 但是，在索引过程中，如果可用非分页池内存量非常低，则另一个需要非分页池内存的内核模式驱动程序也可能会触发此错误。

解决方法

若要调试此问题：请使用参数 3 的 **.cxr（显示上下文记录）**命令，然后使用 **kb（显示堆栈回溯跟踪）**。。

若要解决非分页池内存耗尽问题： 向计算机添加新的物理内存。 这将增加内核可用的非分页池内存的数量。

错误检查 0x28：CORRUPT_ACCESS_TOKEN

项目 • 2024/02/26

CORRUPT_ACCESS_TOKEN 错误检查的值为 0x00000028。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x29：SECURITY_SYSTEM

项目 • 2024/02/26

SECURITY_SYSTEM 错误检查的值为 0x00000029。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x2A：INCONSISTENT_IRP

项目 • 2024/02/22

INCONSISTENT_IRP 错误检查的值为 0x0000002A。 这表示发现 IRP 包含不一致的信息。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅蓝屏错误疑难解答。

INCONSISTENT_IRP 参数

展开表

参数	说明
1	被发现不一致的 IRP 地址
2	Reserved
3	Reserved
4	Reserved

原因

发现一个 IRP 处于不一致状态。 通常这意味着 IRP 的某些字段与 IRP 的剩余状态不一致。 例如，一个 IRP 正在完成，但仍被标记为正在排队加入驱动程序的设备队列中。

注解

此错误检查代码目前并未在系统中使用，而是用于调试目的。

错误检查 0x2B：PANIC_STACK_SWITCH

项目 • 2024/02/26

PANIC_STACK_SWITCH 错误检查的值为 0x0000002B。 这表示内核模式堆栈已溢出。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PANIC_STACK_SWITCH 参数

展开表

参数	说明
1	陷阱帧
2	Reserved
3	Reserved
4	Reserved

原因

此错误通常出现在内核模式驱动程序使用了过多堆栈空间时。 当内核中发生严重数据损坏时，也可能会出现这种情况。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x2C：PORT_DRIVER_INTERNAL

项目 • 2024/02/22

PORT_DRIVER_INTERNAL 错误检查的值为 0x0000002C。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x2D：SCSI_DISK_DRIVER_INTERNAL

项目 • 2024/02/22

The SCSI_DISK_DRIVER_INTERNAL 错误检查的值为 0x0000002D。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x2E：DATA_BUS_ERROR

项目 • 2024/02/27

DATA_BUS_ERROR 错误检查的值为 0x0000002E。 这通常表示检测到系统内存中的奇偶校验错误。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DATA_BUS_ERROR 参数

展开表

参数	说明
1	导致故障的虚拟地址
2	导致故障的物理地址
3	处理器状态寄存器 (PSR)
4	故障指令寄存器 (FIR)

原因

该错误几乎总是由硬件问题引起的，如配置问题、硬件缺陷或硬件不兼容。

导致此错误的最常见硬件问题是 RAM 损坏、二级 (L2) RAM 缓存错误或视频 RAM 错误。 硬盘损坏也会导致此错误。

当设备驱动程序尝试访问 0x8*xxxxxxx* 范围内不存在的地址（换句话说，没有物理地址映射）时，也会导致该错误检查。

解决方法

解决硬件问题：如果系统中最近添加了新硬件，请将其删除并查看错误是否再次出现。

如果现有硬件出现故障，请卸下或更换故障部件。 你应该运行系统制造商提供的硬件诊断程序，以确定哪个硬件组件出现了故障。 有关这些过程的详细信息，请参阅计算机的

用户手册。 检查计算机中的所有适配器卡是否已正确安置。 使用电子用品商店出售的墨迹橡皮擦或电气触点处理剂，确保适配器卡接触点干净无尘。

如果问题发生在新安装的系统上，请检查 BIOS、SCSI 控制器或网卡是否有更新。 这类更新通常可在硬件制造商的网站或公告板系统 (BBS) 上获得。

如果在安装新的或更新的设备驱动程序后出现错误，则应删除或更换驱动程序。 在这种情况下，如果错误发生在启动过程中，并且系统分区已格式化为 NTFS，则可以使用安全模式重命名或删除有问题的驱动程序。

如果在安全模式下将驱动程序用作系统启动过程的一部分，则需要使用恢复控制台启动计算机以访问文件。

有关可能有助于确定导致错误的设备或驱动程序的其他错误信息，请查看事件查看器中的系统日志。 在 BIOS 中禁用内存缓存或隐藏也可能解决此错误。 此外，使用任何可检查硬盘主引导记录的最新商用病毒扫描软件来检查系统是否存在病毒。 所有 Windows 文件系统都可能被病毒感染。

解决硬盘损坏问题：在系统分区上运行 **Chkdsk /f /r**。 在磁盘扫描开始前，必须重新启动系统。 如果由于错误而无法启动系统，请使用恢复控制台并运行 **Chkdsk /r**。

警告如果系统分区是用文件分配表 (FAT) 文件系统格式化的，那么在使用 Scandisk 或其他基于 Microsoft MS-DOS 的硬盘工具从 MS-DOS 验证硬盘的完整性时，Windows 使用的长文件名可能会被损坏。 请务必使用与 Windows 版本相匹配的 Chkdsk 版本。

错误检查 0x2F：INSTRUCTION_BUS_ERROR

项目 • 2024/02/22

INSTRUCTION_BUS_ERROR 错误检查的值为 0x0000002F。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x30：SET_OF_INVALID_CONTEXT

项目 • 2024/02/22

SET_OF_INVALID_CONTEXT 错误检查的值为 0x00000030。 这表明陷阱帧中的堆栈指针值无效。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SET_OF_INVALID_CONTEXT 参数

展开表

参数	说明
1	新的堆栈指针
2	旧的堆栈指针
3	陷阱帧地址
4	0

原因

当某些例程试图将陷阱帧中的堆栈指针设置为比当前堆栈指针值更低的值时，就会出现此错误检查。

如果为捕获到此错误，就会导致内核运行时堆栈指针指向不再有效的堆栈。

错误检查 0x31: PHASE0_INITIALIZATION_FAILED

项目 • 2024/02/22

PHASE0_INITIALIZATION_FAILED Bug 检查的值为 0x00000031。 这表明系统初始化失败。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PHASE0_INITIALIZATION_FAILED 参数

无

原因

系统初始化在很早的阶段就已失败。

解决方法

分析此问题需要使用调试器。

错误检查 0x32：PHASE1_INITIALIZATION_FAILED

项目 • 2024/02/22

PHASE1_INITIALIZATION_FAILED Bug 检查的值为 0x00000032。 这表明系统初始化失败。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PHASE1_INITIALIZATION_FAILED 参数

展开表

参数	说明
1	说明系统初始化失败原因的 NT 状态代码
2	Reserved
3	Reserved
4	Reserved

错误检查 0x33：UNEXPECTED_INITIALIZATION_CALL

项目 • 2024/02/22

UNEXPECTED_INITIALIZATION_CALL 错误检查的值为 0x00000033。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

错误检查 0x34：CACHE_MANAGER

项目 • 2024/02/22

CACHE_MANAGER 错误检查的值为 0x00000034。 这表明文件系统的缓存管理器出现了问题。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

CACHE_MANAGER 参数

展开表

参数	说明
1	指定源文件和行号信息。 高 16 位（“0x”后面的前四个十六进制数字）按标识符号标识源文件。 低 16 位标识文件中出现 Bug 检查的源行。
2	Reserved
3	Reserved
4	Reserved

原因

造成此错误检查的一个可能原因是非分页池内存耗尽。 如果非分页池内存完全耗尽，此错误可能会导致系统停止运行。 但是，在索引过程中，如果可用非分页池内存量非常低，则另一个需要非分页池内存的内核模式驱动程序也可能会触发此错误。

解决方法

若要解决非分页池内存耗尽问题： 向计算机添加新的物理内存。 这将增加内核可用的非分页池内存的数量。

错误检查 0x35：NO_MORE_IRP_STACK_LOCATIONS

项目 • 2024/02/22

NO_MORE_IRP_STACK_LOCATIONS 错误检查的值为 0x00000035。 当 **IoCallDriver** 数据包不再有剩余的堆栈位置时，就会出现此错误检查。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

NO_MORE_IRP_STACK_LOCATIONS 参数

展开表

参数	说明
1	IRP 的地址
2	Reserved
3	Reserved
4	Reserved

原因

高级驱动程序试图通过 **IoCallDriver** 接口调用较低级别的驱动程序，但数据包中没有更多堆栈位置。 这会阻止较低级别的驱动程序访问其参数。

这是一个灾难性的情况，因为较高级别的驱动程序在运行时，就像它已为较低级别的驱动程序填充了参数（根据需要）。 但由于后一个驱动程序没有堆栈位置，前一个驱动程序实际上已经注销了数据包的末尾。 这意味着其他一些内存也已损坏。

错误检查 0x36：DEVICE_REFERENCE_COUNT_NOT_ZERO

项目 • 2024/02/26

DEVICE_REFERENCE_COUNT_NOT_ZERO 错误检查的值为 0x00000036。 这表示驱动程序尝试删除仍具有正引用计数的设备对象。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DEVICE_REFERENCE_COUNT_NOT_ZERO 参数

展开表

参数	说明
1	设备对象的地址。
2	Reserved
3	Reserved
4	Reserved

原因

设备驱动程序试图从系统中删除其中一个设备对象，但该对象的引用计数并非为零。

这意味着该设备仍有未处理的引用。 （引用计数表示无法删除此设备对象的原因数量。）

这是调用设备驱动程序中的一个错误。

错误检查 0x37：FLOPPY_INTERNAL_ERROR

项目 • 2024/02/22

FLOPPY_INTERNAL_ERROR 错误检查的值为 0x00000037。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

错误检查 0x38：SERIAL_DRIVER_INTERNAL

项目 • 2024/02/26

SERIAL_DRIVER_INTERNAL 错误检查的值为 0x00000038。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

错误检查 0x39：SYSTEM_EXIT_OWNED_MUTEX

项目 • 2024/02/22

SYSTEM_EXIT_OWNED_MUTEX 错误检查的值为 0x00000039。 这表明工作例程返回时没有释放它所拥有的 Mutex 对象。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅蓝屏错误疑难解答。

SYSTEM_EXIT_OWNED_MUTEX 参数

展开表

参数	说明
1	导致错误的工作例程的地址。
2	传递给工作例程的参数。
3	工作项的地址。
4	保留。

原因

工作例程返回时仍拥有一个 Mutex 对象。 当前工作线程将继续运行其他无关的工作项，Mutex 永远不会被释放。

解决方法

分析此问题需要使用调试器。 要查找导致错误的驱动程序，请使用 ln（列出最接近符号）调试器命令：

kd> ln 地址

其中地址是参数 1 中给定的工作例程。

错误检查 0x3A：SYSTEM_UNWIND_PREVIOUS_USER

项目 • 2024/02/26

SYSTEM_UNWIND_PREVIOUS_USER 错误检查的值为 0x0000003A。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

错误检查 0x3B：SYSTEM_SERVICE_EXCEPTION

项目 • 2024/02/26

SYSTEM_SERVICE_EXCEPTION 错误检查具有 0x0000003B 值。 这表示在执行从非特权代码转换为特权代码的例程时发生了异常。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SYSTEM_SERVICE_EXCEPTION 参数

展开表

参数	说明
1	导致错误检查的异常。
2	导致错误检查的指令的地址
3	导致错误检查的异常的上下文记录的地址
4	0（未使用）

原因

此停止代码指示执行代码存在异常，其下方的线程是系统线程。

出现这种情况的原因可能是取消引用了 NULL 指针或访问了随机的错误地址。 而这又可能是内存被过早释放或数据结构损坏造成的。

NTSTATUS 值中介绍了在参数 1 中返回的异常信息。 异常代码在 ntstatus.h（Windows 驱动程序工具包提供的标头文件）中定义。 （有关详细信息，请参阅 Windows 驱动程序工具包中的标头文件）。

常见的异常代码包括：

- 0x80000003: STATUS_BREAKPOINT

没有内核调试器连接到系统时，遇到断点或 ASSERT。

- 0xC0000005: STATUS_ACCESS_VIOLATION

  出现内存访问冲突。

解决方法

若要确定具体原因并创建代码修复，需要编程经验和故障模块源代码的访问权限。

若要调试此问题，请使用参数 3 的 .cxr（显示上下文记录） 命令，然后使用 kb（显示堆栈回溯跟踪）。 还可以在此停止代码之前的代码中设置断点，并尝试单步前进到故障代码中。 使用 u、ubuu (unassemble) 命令查看程序集程序代码。

!analyze 调试程序扩展显示有关错误检查的信息，有助于确定根本原因。 下面的示例是 !analyze 的输出。

dbgcmd

```
SYSTEM_SERVICE_EXCEPTION (3b)
An exception happened while executing a system service routine.
Arguments:
Arg1: 00000000c0000005, Exception code that caused the bugcheck
Arg2: fffff802328375b0, Address of the instruction which caused the bugcheck
Arg3: ffff9c0a746c2330, Address of the context record for the exception that
caused the bugcheck
Arg4: 0000000000000000, zero.
...
```

有关 WinDbg 和 !analyze 的详细信息，请参阅下列主题：

- 使用 !analyze 扩展
- 使用 WinDbg 分析内核模式转储文件

有关 Windows 内存使用情况的详细信息，请参阅由 Pavel Yosifovich、Mark E. Russinovich、David A. Solomon 和 Alex Ionescu 撰写的 Windows Internals 第 7 版第 1 部分。

标识驱动程序

如果能够识别出导致错误的驱动程序，则会将其名称打印在蓝屏上，并存储在内存中的 (PUNICODESTRING) **KiBugCheckDriver** 位置。 可以使用一个调试器命令 dx（显示调试器对象模型表达式）来显示此内容：`dx KiBugCheckDriver`。

dbgcmd

```
kd> dx KiBugCheckDriver
KiBugCheckDriver                 : 0xffffe10b9991e3e8 : "nvlddmkm.sys"
[Type: _UNICODE_STRING *]
```

使用 !error 扩展显示参数 1 中的异常代码的相关信息。 以下是 !error 的输出实例。

dbgcmd

```
2: kd> !error 00000000c0000005
Error code: (NTSTATUS) 0xc0000005 (3221225477) - The instruction at 0x%p
referenced memory at 0x%p. The memory could not be %s.
```

请查看 WinDbg 的“堆栈文本”输出以获取故障发生时运行的内容的线索。 如果有多个转储文件可用，请比较信息以查找堆栈中的通用代码。 使用调试器命令，如使用 kb（显示堆栈回溯）来调查错误代码。

使用以下命令列出在内存中加载的模块：lm t n

使用 !memusage 检查系统内存的一般状态。 还可以使用命令 !pte 和 !pool 来检查内存的特定区域。

过去，此错误已链接到过度使用分页池，如果用户模式图形驱动程序交叉将错误数据传递到内核代码，则可能导致出现这种情况。 如果你怀疑就是这种情况，请使用驱动程序验证器中的池选项收集其他信息。

驱动程序验证程序

驱动程序验证程序是一个实时运行的工具，用于检查驱动程序的行为。 例如，驱动程序验证程序检查内存资源（如内存池）的使用。 如果在执行驱动程序代码时标识错误，它会主动创建一个异常，以允许进一步检查该部分驱动程序代码。 驱动程序验证程序管理器内置于 Windows 中，可在所有 Windows 电脑上使用。

若要启动驱动程序验证程序管理器，请在命令提示下输入“验证程序”。 你可以配置要验证的驱动程序。 验证驱动程序的代码在运行时会增加开销，因此请尝试验证尽可能少的驱动程序。 有关详细信息，请参阅驱动程序验证程序。

注解

有关 Windows 错误检查代码的一般故障排除，请按照以下建议操作：

- 如果最近添加了新的设备驱动程序或系统服务，请尝试删除或更新它们。 尝试确定系统中导致新错误检查代码出现的原因。

- 查看设备管理器，了解是否有任何设备标记为惊叹号 (!)，这表示存在问题。 查看任何故障设备驱动程序的属性中显示的事件日志。 尝试更新相关驱动程序。
- 检查事件查看器中的系统日志，以获取可能有助于查明导致错误的设备或驱动程序的其他错误消息。 在系统日志中查找与蓝屏同时出现的严重错误。
- 如果最近向系统添加了硬件，请尝试删除或替换它。 或与制造商联系，查看是否有可用的修补程序。

有关其他常规疑难解答信息，请参阅蓝屏数据。

另请参阅

使用 Windows 调试器 (WinDbg) 进行故障转储分析

使用 WinDbg 分析内核模式转储文件

错误检查代码参考

错误检查 0x3C：INTERRUPT_UNWIND_ATTEMPTED

项目 • 2024/02/26

INTERRUPT_UNWIND_ATTEMPTED 错误检查的值为 0x0000003C。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

错误检查 0x3D：INTERRUPT_EXCEPTION_NOT_HANDLED

项目 • 2024/02/22

INTERRUPT_EXCEPTION_NOT_HANDLED 错误检查的值为 0x0000003D。 这表明内核中断对象中断管理的异常处理程序无法处理生成的异常。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INTERRUPT_EXCEPTION_NOT_HANDLED 参数

展开表

参数	说明
1	异常记录（可用时）
2	上下文记录（可用时）
3	0
4	0

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x3E：MULTIPROCESSOR_CONFIGURATION_NOT_SUPPORTED

项目 • 2024/02/22

MULTIPROCESSOR_CONFIGURATION_NOT_SUPPORTED 错误检查的值为 0x0000003E。 这表明系统有多个处理器，但它们彼此之间不对称。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

MULTIPROCESSOR_CONFIGURATION_NOT_SUPPORTED 参数

无

原因

要保持对称，所有处理器必须具有相同的类型和级别。 该系统包含不同类型的处理器（例如一个 Pentium 处理器和一个 80486 处理器）。

错误检查 0x3F：NO_MORE_SYSTEM_PTES

项目 • 2024/02/26

NO_MORE_SYSTEM_PTES 错误检查的值为 0x0000003F。 这是系统执行过多 I/O 操作的结果。 这导致了系统页表条目 (PTE) 碎片化。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

NO_MORE_SYSTEM_PTES 参数

展开表

参数	说明
1	**0**：系统扩展 PTE 类型 **1**：非分页池扩展 PTE 类型
2	内存请求的大小
3	免费系统 PTE 总数
4	系统 PTE 总数

原因

几乎在所有情况下，系统实际上都没有 PTE。 相反，驱动程序请求了一个大内存块，但没有足够大的连续内存块来满足该请求。

视频驱动程序通常会分配大量必须成功的内核内存。 某些备份程序执行相同的操作。

解决方法

可能的解决方法：修改注册表以增加系统 PTE 总数。 如果仍无济于事，请删除最近安装的任何软件，尤其是备份实用程序或磁盘密集型应用程序。

调试问题：可使用以下方法来调试错误检查 0x3F。

首先，获取堆栈跟踪，然后使用 **!sysptes 3** 扩展命令。

然后将 **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\TrackPtes** 设置为等于 DWORD 1，然后重启。 这将导致系统保存堆栈跟踪。

这样就可以显示有关 PTE 所有者的更详细信息。 例如：

dbgcmd

```
0: kd> !sysptes 4

0x2c47 System PTEs allocated to mapping locked pages

VA       MDL     PageCount  Caller/CallersCaller
f0e5db48 eb6ceef0        1
ntkrpamp!MmMapLockedPages+0x15/ntkrpamp!IopfCallDriver+0x35
f0c3fe48 eb634bf0        1
netbt!NbtTdiAssociateConnection+0x1f/netbt!DelayedNbtProcessConnect+0x17c
f0db38e8 eb65b880        1
mrxsmb!SmbMmAllocateSessionEntry+0x89/mrxsmb!SmbCepInitializeExchange+0xda
f8312568 eb6df880        1
rdbss!RxCreateFromNetRoot+0x3d7/rdbss!RxCreateFromNetRoot+0x93
f8363908 eb685880        1
mrxsmb!SmbMmAllocateSessionEntry+0x89/mrxsmb!SmbCepInitializeExchange+0xda
f0c54248 eb640880        1
rdbss!RxCreateFromNetRoot+0x3d7/rdbss!RxCreateFromNetRoot+0x93
f0ddf448 eb5f3160        1
mrxsmb!MrxSmbUnalignedDirEntryCopyTail+0x387/mrxsmb!MRxSmbCoreInformation+0x
36
f150bc08 eb6367b0        1
mrxsmb!MrxSmbUnalignedDirEntryCopyTail+0x387/mrxsmb!MRxSmbCoreInformation+0x
36
f1392308 eb6fba70        1
netbt!NbtTdiOpenAddress+0x1fb/netbt!DelayedNbtProcessConnect+0x17c
eb1bee64 edac5000      200
VIDEOPRT!pVideoPortGetDeviceBase+0x118/VIDEOPRT!VideoPortMapMemory+0x45
f139b5a8 edd4b000       12
rdbss!FsRtlCopyWrite2+0x34/rdbss!RxDriverEntry+0x149
eb41f400 ede92000       20
VIDEOPRT!pVideoPortGetDeviceBase+0x139/VIDEOPRT!VideoPortGetDeviceBase+0x1b
eb41f198 edf2a000       20
NDIS!NdisReadNetworkAddress+0x3a/NDIS!NdisFreeSharedMemory+0x58
eb41f1e4 eb110000       10
VIDEOPRT!pVideoPortGetDeviceBase+0x139/VIDEOPRT!VideoPortGetDeviceBase+0x1b
......
```

如果在设置 **TrackPtes** 注册表值后系统再次耗尽 PTE，则会发出**错误检查 0xD8** (DRIVER_USED_EXCESSIVE_PTES)，而不是 0x3F。 同时还会显示导致该错误的驱动程序名

称。

错误检查 0x40：TARGET_MDL_TOO_SMALL

项目 • 2024/02/26

TARGET_MDL_TOO_SMALL 错误检查的值为 0x00000040。 这表明驱动程序未正确使用 **IoBuildPartialMdl**。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

TARGET_MDL_TOO_SMALL 参数

无

原因

这是一个驱动程序错误。 驱动程序调用了 **IoBuildPartialMdl** 函数，并将一个 MDL 传递给它以映射源 MDL 的一部分，但目标 MDL 的大小不足以映射请求的整个地址范围。

解决方法

源 MDL 和目标 MDL 以及要映射的地址范围长度是 **IoBuildPartialMdl** 函数的第一、第二和第四个参数。 因此，在调试过程中对该特定函数进行堆栈跟踪可能会有所帮助。 请确保代码正确计算了目标 MDL 所需的大小，以满足传递给此函数的地址范围长度。

错误检查 0x41: MUST_SUCCEED_POOL_EMPTY

项目 • 2024/02/22

MUST_SUCCEED_POOL_EMPTY 错误检查的值为 0x00000041。 这表示内核模式线程请求了太多必须成功的池。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

MUST_SUCCEED_POOL_EMPTY 参数

展开表

参数	说明
1	无法满足的请求的大小
2	从非分页池中使用的页数
3	非分页池中大于 PAGE_SIZE 的请求数
4	可用页数

原因

任何驱动程序都不得请求必须成功的池。

如果必须成功请求无法被满足，就会发出错误检查。

解决方法

替换或重写提出请求的驱动程序。 驱动程序不应请求必须成功的池。 相反，它应该请求正常池，并正常处理池暂时为空的情况。

kb（显示堆栈回溯）命令将显示导致错误的驱动程序。

此外，也有可能是第二个组件耗尽了必须成功的池。 要确定情况是否如此，请先使用 **kb** 命令。 然后使用 !vm 1 来显示总池使用情况，使用 !poolused 2 来显示每个标记的非分页池使用情况，并使用 **!poolused 4** 来显示每个标记的分页池使用情况。 与使用最多池的标签相关联的组件可能是问题的根源。

错误检查 0x42：ATDISK_DRIVER_INTERNAL

项目 • 2024/02/22

ATDISK_DRIVER_INTERNAL 错误检查的值为 0x00000042。

此错误检查很少出现。

> ⓘ **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x43：NO_SUCH_PARTITION

项目 • 2024/02/22

NO_SUCH_PARTITION 错误检查的值为 0x00000043。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x44：MULTIPLE_IRP_COMPLETE_REQUESTS

项目 • 2024/02/20

MULTIPLE_IRP_COMPLETE_REQUESTS Bug 检查的值为 0x00000044。 这表示驱动程序已试图请求完成一个已完成的 IRP。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

MULTIPLE_IRP_COMPLETE_REQUESTS 参数

展开表

参数	说明
1	IRP 的地址
2	Reserved
3	Reserved
4	Reserved

原因

驱动程序已调用 **IoCompleteRequest** 来要求完成 IRP，但数据包已完成。

解决方法

这是一个很难发现的 bug，因为最简单的情况（驱动程序两次试图完成自己的数据包）通常不是问题的根源。 更有可能的是，两个单独的驱动程序都认为自己拥有该数据包，并且每个驱动程序都试图完成它。 第一个请求成功，第二个请求失败，因此出现了这个 Bug 检查。

由于第一个驱动程序的踪迹已被第二个驱动程序覆盖，因此很难跟踪系统中的哪些驱动程序导致了错误。 但是，可以通过检查每个堆栈位置中的设备对象字段来找到当前请求的

驱动程序堆栈。

Bug 检查 0x45：INSUFFICIENT_SYSTEM_MAP_REGS

项目 • 2024/02/20

INSUFFICIENT_SYSTEM_MAP_REGS Bug 检查的值为 0x00000045。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x46：DEREF_UNKNOWN_LOGON_SESSION

项目 • 2024/02/20

DEREF_UNKNOWN_LOGON_SESSION Bug 检查的值为 0x00000046。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x47：REF_UNKNOWN_LOGON_SESSION

项目 • 2024/02/20

REF_UNKNOWN_LOGON_SESSION Bug 检查的值为 0x00000047。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x48：CANCEL_STATE_IN_COMPLETED_IRP

项目 • 2024/02/21

CANCEL_STATE_IN_COMPLETED_IRP Bug 检查的值为 0x00000048。 这表明 I/O 请求数据包 (IRP) 已完成，随后被取消。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

CANCEL_STATE_IN_COMPLETED_IRP 参数

展开表

参数	说明
1	指向 IRP 的指针
2	驱动程序设置的 Cancel 例程
3	Reserved
4	Reserved

原因

具有 *Cancel* 例程集的 IRP 已正常完成，无需取消。 但在完成后，驱动程序调用 IRP 的 *Cancel* 例程。

这可能是由于驱动程序完成了 IRP，然后尝试取消它造成的。

这也可能是由于两个驱动程序各自试图以不正确的方式访问同一个 IRP 造成的。

解决方法

Cancel 例程参数可用于确定哪个驱动程序或堆栈导致了 Bug 检查。

Bug 检查 0x49：PAGE_FAULT_WITH_INTERRUPTS_OFF

项目 • 2024/02/20

PAGE_FAULT_WITH_INTERRUPTS_OFF Bug 检查的值为 0x00000049。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

Bug 检查 0x4A：IRQL_GT_ZERO_AT_SYSTEM_SERVICE

项目 • 2024/02/23

IRQL_GT_ZERO_AT_SYSTEM_SERVICE Bug 检查的值为 0x0000004A。 这表示当线程的 IRQL 仍高于PASSIVE_LEVEL时，线程正从系统调用返回到用户模式。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

IRQL_GT_ZERO_AT_SYSTEM_SERVICE 参数

展开表

参数	说明
1	系统函数的地址（系统调用例程）
2	当前 IRQL
3	0
4	0

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

Bug 检查 0x4B：STREAMS_INTERNAL_ERROR

项目 • 2024/02/21

STREAMS_INTERNAL_ERROR Bug 检查的值为 0x0000004B。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x4C：FATAL_UNHANDLED_HARD_ERROR

项目 • 2024/02/20

FATAL_UNHANDLED_HARD_ERROR Bug 检查的值为 0x0000004C。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x4D：NO_PAGES_AVAILABLE

项目 • 2024/02/20

NO_PAGES_AVAILABLE Bug 检查的值为 0x0000004D。 这表示没有可用页面来继续操作。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

NO_PAGES_AVAILABLE 参数

展开表

参数	说明
1	脏页总数
2	发送至页面文件的脏页数
3	发生 Bug 检查时可用的非分页缓冲池的大小
4	最近修改的写入错误状态。

原因

若要查看常规内存统计信息，请使用 !vm 3 扩展。

此 Bug 检查可能由以下任意原因造成：

- 驱动程序已阻止、死锁已修改或映射的页面编写器。 此类示例包括互斥死锁或对文件系统驱动程序或筛选器驱动程序中分页内存的访问。 这表明存在驱动程序 bug。

  如果参数 1 或参数 2 较大，则有可能出现这种情况。 使用 !vm 3。

- 存储驱动程序未处理请求。 此类示例包括滞留队列和非响应驱动器。 这表明存在驱动程序 bug。

  如果参数 1 或参数 2 较大，则有可能出现这种情况。 使用 !vm 8，后跟 !process 0 7。

- 高优先级实时线程导致平衡集管理器无法从工作集中剪裁页面，或导致修改后的页面编写器无法将其写出。这表明创建此线程的组件中存在 bug。

  这种情况很难分析。 请尝试使用 **!ready**。 还可以尝试 **!process 0 7** 列出所有线程，并查看是否有任何线程累积过多的内核时间及其当前优先级。 此类进程可能已阻止内存管理线程提供可用页面。

- 没有足够的池可供存储堆栈来写出修改的页面。 这表明存在驱动程序 bug。

  如果参数 3 较小，则有可能出现这种情况。 使用 **!vm** 和 **!poolused 2**。

如果找不到问题，请尝试从一开始就使用附加的内核调试器启动，并监视情况。

Bug 检查：0x4E PFN_LIST_CORRUPT

项目 • 2024/02/20

PFN_LIST_CORRUPT Bug 检查的值为 0x0000004E。 这表示页框号 (PFN) 列表已损坏。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅蓝屏错误疑难解答。

PFN_LIST_CORRUPT 参数

参数 1 表示违规类型。 其他参数的含义取决于*参数 1* 的值。

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x01	ListHead 值已损坏	可用页数	0	列表标头已损坏。
0x02	列表中正在删除的条目	最高物理页码	要删除的条目的引用计数	列表条目已损坏。
0x06	页框号	原型 PTE	PTE 内容	硬件 PTE 和/或原型 PTE 数据结构已损坏。 这可能是硬件单个位错误、DMA 传输中断等造成的。
0x07	页框号	当前共享计数	0	驱动程序已解锁某个页面的次数超过锁定页面的次数。
0x8D	状态不一致的页框号	0	0	空闲页列表已损坏。 此错误代码很可能表明存在硬件问题。
0x8F	新页码	旧页码	0	空闲或归零页列表头已损坏。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x99	页框号	当前页面状态	0	页表条目 (PTE) 或 PFN 已损坏。
0x9A	页框号	当前页面状态	要删除的条目的引用计数	驱动程序尝试释放仍处于 IO 锁定状态的页面。

原因

此错误通常是由于驱动程序传递了错误的内存描述符列表造成的。 例如，驱动程序可能已使用同一列表调用 **MmUnlockPages** 两次。

如果内核调试器可用，请检查堆栈跟踪：**!analyze** 调试扩展显示有关 Bug 检查的信息，并且有助于确定根本原因，然后输入其中一个 **k（显示堆栈回溯）**命令以查看调用堆栈。

另请参阅

!analyze

Bug 检查代码参考

Bug 检查 0x4F: NDIS_INTERNAL_ERROR

项目 • 2024/02/20

NDIS_INTERNAL_ERROR Bug 检查的值为 0x0000004F。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

错误检查 0x50：PAGE_FAULT_IN_NONPAGED_AREA

项目 • 2024/10/16

PAGE_FAULT_IN_NONPAGED_AREA 错误检查的值为 0x00000050。 这表明引用了无效的系统内存。 通常，内存地址错误或内存地址指向已释放的内存。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PAGE_FAULT_IN_NONPAGED_AREA 参数

展开表

参数	说明
1	引用的内存地址
2	*Windows 1507 (TH1) 版本 - x64 后* **0**：读取操作 **2**：写入操作 **10**：执行操作 *Windows 1507 (TH1) 版本 - x86 后* **0**：读取操作 **2**：写入操作 **10**：执行操作 *Windows 1507 (TH1) 版本之后 - Arm* **0**：读取操作 **1**：写入操作 **8**：执行操作 *Windows 1507 (TH1) 版本 x64 / x86 前*

参数	说明
	0： 读取操作 **1：** 写入操作
3	引用内存的地址（如果已知）
4	页面错误类型 0x0 - NONPAGED_BUGCHECK_FREED_PTE - 引用的地址位于标记为空闲的页面表条目上。 0x2 - NONPAGED_BUGCHECK_NOT_PRESENT_PAGE_TABLE 引用的地址没有有效的活动页表条目。 0x03-NONPAGED_BUGCHECK_WRONG_SESSION - 尝试在没有会话的进程的上下文中引用会话空间地址。 通常，这意味着调用方在未首先正确获取对正确进程的对象引用并附加到该进程的情况下，不正确地尝试访问会话地址。 此错误检查和子类型最后用于 Windows 10 RS3 中。 在 Windows 10 RS4 及更高版本中，此错误改为 0x02 (NONPAGED_BUGCHECK_NOT_PRESENT_PAGE_TABLE)。 NONPAGED_BUGCHECK_VA_NOT_CANONICAL 0x04 - 尝试引用非规范（非法）虚拟地址（参数 1）。 调用方不会尝试访问此地址。 0xF - NONPAGED_BUGCHECK_USER_VA_ACCESS_INCONSISTENT - 内核模式代码尝试访问用户模式虚拟地址，但这种访问是不允许的。

如果能够识别出导致错误的驱动程序，则其名称将打印在蓝屏上，并存储在内存中的 (PUNICODESTRING) **KiBugCheckDriver 位置**。 可以使用以下调试程序 dx 命令来显示它 - `dx KiBugCheckDriver`。

原因

安装了错误的系统服务或错误的驱动程序代码都可能造成错误检查 0x50。 防病毒软件和损坏的 NTFS 卷也可能触发此错误。

安装有故障的硬件之后，或者安装的硬件出现故障时（通常与 RAM 缺陷有关，包括主存储器、二级 RAM 缓存或视频 RAM 缺陷），也可能出现此错误。

注解

事件日志：在事件查看器中检查系统日志中的其他错误消息，这些消息可能有助于查明导致错误的设备或驱动程序。 有关详细信息，请参阅打开事件查看器。 在系统日志中查找与蓝屏同时出现的严重错误。

解决有故障的驱动程序：检查驱动程序的名称是否已列在蓝屏上或存在于事件日志中。 请与驱动程序供应商联系，查看是否有更新的驱动程序可用。

解决故障系统服务问题：禁用服务并确认这是否可以解决错误。 如果可以，请联系系统服务的制造商以了解可能的更新。 如果在系统启动期间发生错误，请调查 Windows 修复选项。 有关详细信息，请参阅 Windows 10 中的恢复选项。

解决防病毒软件问题：禁用程序并确认这是否可以解决错误。 如果是，请与程序制造商联系，了解可能的更新。

解决损坏的 NTFS 卷问题：运行 **Chkdsk /f /r** 来检测和修复磁盘错误。 在系统分区上开始磁盘扫描之前，必须重启系统。 请与硬盘驱动器系统制造商联系，找到他们为硬盘驱动子系统提供的任何诊断工具。

Windows 内存诊断：运行 Windows 内存诊断工具，以测试物理内存。 选择“开始”按钮，再选择“控制面板”。 在搜索框中键入“内存”，然后选择“*诊断计算机内存问题*”。运行测试后，使用事件查看器查看系统日志下的结果。 查找“内存诊断结果”条目以查看结果。

解决硬件故障问题：如果系统中最近添加了新硬件，请将其删除并查看错误是否再次出现。 如果现有硬件出现故障，请卸下或更换故障部件。 你应运行系统制造商提供的硬件诊断。 有关这些过程的详细信息，请参阅计算机的用户手册。

有关常规蓝屏故障排除信息，请参阅**分析错误检查蓝屏数据**和停止错误或蓝屏错误问题的高级疑难解答。

解决方法

若要确定具体原因并创建代码修复，需要编程经验和故障模块源代码的访问权限。

通常，引用的地址在释放的内存中，或者是无效的。 这不能通过“try-except”处理程序保护，只能由探测或类似的编程技术保护。 有关文件系统驱动程序中缓冲区处理和探测的相关信息，请参阅缓冲区处理。 有关驱动程序开发的最佳做法以及驱动程序开发人员常犯错误的相关信息，请参阅 Surface Team Driver Development Best Practices。

通过 -v 详细选项使用 **!analyze** 调试扩展，以显示有关错误检查的信息，从而确定根本原因。

```
dbgcmd
```

```
2: kd> !analyze -v
*******************************************************************************
***
*
*
*                        Bugcheck Analysis
*
*
*
*******************************************************************************
***

PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced.  This cannot be protected by try-
except.
Typically the address is just plain bad or it is pointing at freed memory.
Arguments:
Arg1: ffffffff00000090, memory referenced.
Arg2: 0000000000000000, value 0 = read operation, 1 = write operation.
Arg3: fffff80240d322f9, If non-zero, the instruction address which
referenced the bad memory
    address.
Arg4: 000000000000000c, (reserved)
```

在本例中，参数 2 表示在读取内存区域时发生错误检查。

查看所有 !analyze 输出，以了解进行错误检查时发生的情况。 检查 MODULE_NAME: 和 FAULTING_MODULE: 以查看引用无效系统内存涉及的代码。

请查看“堆栈文本”以获取故障发生时运行的内容的线索。 如果有多个转储文件可用，请比较信息以查找堆栈中的通用代码。

使用 !analyze 输出中提供的 .trap 命令以设置上下文。

dbgcmd

```
TRAP_FRAME:  fffff98112e8b3d0 -- (.trap 0xfffff98112e8b3d0)
```

使用调试器命令，例如使用 **kb（显示堆栈回溯）**来调查错误代码。

使用 `lm t n` 列出内存中加载的模块。

使用 d、da、db、dc、dd、dD、df、dp、dq、du、dw（显示内存）命令调查参数 1 和参数 3 引用的内存区域。

dbgcmd

```
2: kd> db ffffffff00000090
ffffffff`00000090  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
```

```
????????????????
fffffff`000000a0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
????????????????
fffffff`000000b0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
????????????????
fffffff`000000c0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
????????????????
fffffff`000000d0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
????????????????
fffffff`000000e0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
????????????????
fffffff`000000f0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
????????????????
fffffff`00000100  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
????????????????
```

在本例中，参数 1 中的内存区域中似乎没有数据，该参数是试图读取的内存区域。

使用 !address 命令查看参数 3，该参数是引用错误内存的指令的地址。

dbgcmd

```
2: kd> !address fffff80240d322f9
Usage:                  Module
Base Address:           fffff802`40ca8000
End Address:            fffff802`415fb000
Region Size:            00000000`00953000
VA Type:                BootLoaded
Module name:            ntoskrnl.exe
Module path:            [\SystemRoot\system32\ntoskrnl.exe]
```

使用参数 3 的 u、ub、uu（取消汇编）反汇编检查引用错误内存的对象。 有关 X64 处理器和汇编语言的详细信息，请参阅 X64 处理器。

dbgcmd

```
2: kd> u fffff80240d322f9
nt!RtlSubtreePredecessor+0x9:
fffff802`40d322f9 488b4810        mov     rcx,qword ptr [rax+10h]
fffff802`40d322fd eb07            jmp     nt!RtlSubtreePredecessor+0x16
(fffff802`40d32306)
fffff802`40d322ff 488bc1          mov     rax,rcx
fffff802`40d32302 488b4910        mov     rcx,qword ptr [rcx+10h]
fffff802`40d32306 4885c9          test    rcx,rcx
fffff802`40d32309 75f4            jne     nt!RtlSubtreePredecessor+0xf
(fffff802`40d322ff)
fffff802`40d3230b c3              ret
fffff802`40d3230c c3              ret
```

使用 `ub` 从给定地址向后分解。

使用 r（寄存器）命令检查在检查系统错误时执行的操作。

dbgcmd

```
2: kd> r
Last set context:
rax=ffffffff00000080 rbx=0000000000000000 rcx=ffffa68337cb7028
rdx=7a107838c48dfc00 rsi=0000000000000000 rdi=0000000000000000
rip=fffff80240d322f9 rsp=ffff840c96510958 rbp=ffffffffffffffff
 r8=ffffffffffffffff  r9=0000000000000000 r10=7fffffffffffffc
r11=ffff840c96510a10 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei ng nz na pe nc
cs=0010  ss=0018  ds=0000  es=0000  fs=0000  gs=0000
efl=00010282
nt!RtlSubtreePredecessor+0x9:
fffff802`40d322f9 488b4810        mov     rcx,qword ptr [rax+10h]
ds:ffffffff`00000090=????????????????
```

在这种情况下，`fffff80240d322f9` 位于指令指针寄存器 rip 中。

`!pte` 和 `!pool` 命令也可用于检查内存。

使用 `!memusage` 检查系统内存的一般状态。

有关 Windows 内存使用情况的详细信息，请参阅由 Pavel Yosifovich、Mark E. Russinovich、David A. Solomon 和 Alex Ionescu 撰写的 Windows Internals 第 7 版第 1 部分。

驱动程序验证程序

驱动程序验证程序是一个实时运行的工具，用于检查驱动程序的行为。 例如，驱动程序验证程序检查内存资源（如内存池）的使用。 如果在执行驱动程序代码时发现错误，它会主动创建一个异常，以允许进一步检查该部分驱动程序代码。 驱动程序验证程序管理器内置于 Windows 中，可在所有 Windows PC 上使用。 使用驱动程序验证程序跟踪失败的具体原因。

若要启动驱动程序验证程序管理器，请在命令提示下键入“验证程序”。 你可以配置要验证的驱动程序。 验证驱动程序的代码在运行时会增加开销，因此请尝试验证尽可能少的驱动程序。 如果识别出故障驱动程序，请选择它。 有关详细信息，请参阅驱动程序验证程序。

另请参阅

错误检查代码参考

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Bug 检查 0x51：REGISTRY_ERROR

项目 • 2024/02/20

REGISTRY_ERROR Bug 检查的值为 0x00000051。 这表示发生了严重的注册表错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

REGISTRY_ERROR 参数

展开表

参数	说明
1	保留
2	Reserved
3	指向 hive 的指针（如果可用）
4	如果 hive 损坏，则为 **HvCheckHive**的返回代码（如果可用）

原因

注册表出错了。 如果内核调试器可用，则获取堆栈跟踪：**!analyze** 调试扩展显示有关 Bug 检查的信息，并且非常有助于确定根本原因，然后输入其中一个 **k（显示堆栈回溯）**命令以查看调用堆栈。

此错误可能表明注册表在尝试读取其中一个文件时遇到了 I/O 错误。 这可能是硬件问题或文件系统损坏造成的。

这也可能是由于刷新操作失败造成的，刷新操作仅在安全系统中使用，并且仅在遇到资源限制时使用。

Bug 检查 0x52：MAILSLOT_FILE_SYSTEM

项目 • 2024/02/20

MAILSLOT_FILE_SYSTEM Bug 检查的值为 0x00000052。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x53：NO_BOOT_DEVICE

项目 • 2024/02/21

NO_BOOT_DEVICE Bug 检查的值为 0x00000053。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x54：LM_SERVER_INTERNAL_ERROR

项目 • 2024/02/20

LM_SERVER_INTERNAL_ERROR Bug 检查的值为 0x00000054。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x55：DATA_COHERENCY_EXCEPTION

项目 • 2024/02/20

DATA_COHERENCY_EXCEPTION Bug 检查的值为 0x00000055。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

Bug 检查 0x56：INSTRUCTION_COHERENCY_EXCEPTION

项目 • 2024/02/20

INSTRUCTION_COHERENCY_EXCEPTION Bug 检查的值为 0x00000056。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

Bug 检查 0x57：XNS_INTERNAL_ERROR

项目 • 2024/02/21

XNS_INTERNAL_ERROR Bug 检查的值为 0x00000057。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x58：FTDISK_INTERNAL_ERROR

项目 • 2024/02/21

FTDISK_INTERNAL_ERROR Bug 检查的值为 0x00000058。 如果系统从镜像分区的错误副本启动，则会出现此问题。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

FTDISK_INTERNAL_ERROR 参数

无

原因

配置单元表明镜像有效，但事实并非如此。 配置单元实际上应该指向阴影分区。

这几乎总是由被恢复的主分区造成的。

解决方法

从阴影分区重启系统。

Bug 检查 0x59：PINBALL_FILE_SYSTEM

项目 • 2024/02/20

PINBALL_FILE_SYSTEM Bug 检查的值为 0x00000059。 这表明 Pinball 文件系统出现了问题。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PINBALL_FILE_SYSTEM 参数

展开表

参数	说明
1	指定源文件和行号信息。 高 16 位（“0x”后面的前四个十六进制数字）按标识符号标识源文件。 低 16 位标识文件中出现 Bug 检查的源行。
2	Reserved
3	Reserved
4	Reserved

原因

造成此 Bug 检查的一个可能原因是非分页池内存耗尽。 如果非分页池内存完全耗尽，此错误可能会导致系统停止运行。 但是，在索引过程中，如果可用非分页池内存量非常低，则另一个需要非分页池内存的内核模式驱动程序也可能会触发此错误。

解决方法

若要解决非分页池内存耗尽问题： 向计算机添加新的物理内存。 这将增加内核可用的非分页池内存的数量。

Bug 检查 0x5A：CRITICAL_SERVICE_FAILED

项目 • 2024/02/20

CRITICAL_SERVICE_FAILED Bug 检查的值为 0x0000005A。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

Bug 检查 0x5B：SET_ENV_VAR_FAILED

项目 • 2024/02/20

The SET_ENV_VAR_FAILED Bug 检查的值为 0x0000005B。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

错误检查 0x5C：HAL_INITIALIZATION_FAILED

项目 • 2024/10/16

HAL_INITIALIZATION_FAILED 错误检查的值为 0x0000005C。

这表明 HAL 初始化失败。 硬件抽象层是 Windows 启动时早期初始化的低级别代码。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

另请参阅

错误检查代码参考

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Bug 检查 0x5D：UNSUPPORTED_PROCESSOR

项目 • 2024/02/20

UNSUPPORTED_PROCESSOR Bug 检查的值为 0x0000005D。 这表明计算机正尝试在不受支持的处理器上运行 Windows。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

UNSUPPORTED_PROCESSOR 参数

无

原因

Windows 需要比当前使用的处理器更高级别的处理器。

Bug 检查 0x5E: OBJECT_INITIALIZATION_FAILED

项目 • 2024/02/20

OBJECT_INITIALIZATION_FAILED Bug 检查的值为 0x0000005E。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x5F：SECURITY_INITIALIZATION_FAILED

项目 • 2024/02/20

The SECURITY_INITIALIZATION_FAILED Bug 检查的值为 0x0000005F。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x60：PROCESS_INITIALIZATION_FAILED

项目 • 2024/02/20

PROCESS_INITIALIZATION_FAILED Bug 检查的值为 0x00000060。

此 Bug 检查很少出现。

> ⓘ **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

Bug 检查 0x61: HAL1_INITIALIZATION_FAILED

项目 • 2024/02/20

HAL1_INITIALIZATION_FAILED Bug 检查的值为 0x00000061。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x62：OBJECT1_INITIALIZATION_FAILED

项目 • 2024/02/20

OBJECT1_INITIALIZATION_FAILED Bug 检查的值为 0x00000062。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x63：SECURITY1_INITIALIZATION_FAILED

项目 • 2024/02/20

SECURITY1_INITIALIZATION_FAILED Bug 检查的值为 0x00000063。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x64：SYMBOLIC_INITIALIZATION_FAILED

项目 • 2024/02/21

SYMBOLIC_INITIALIZATION_FAILED Bug 检查的值为 0x00000064。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x65：MEMORY1_INITIALIZATION_FAILED

项目 • 2024/02/20

MEMORY1_INITIALIZATION_FAILED Bug 检查的值为 0x00000065。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x66：CACHE_INITIALIZATION_FAILED

项目 • 2024/02/20

CACHE_INITIALIZATION_FAILED Bug 检查的值为 0x00000066。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x67：CONFIG_INITIALIZATION_FAILED

项目 • 2024/02/20

CONFIG_INITIALIZATION_FAILED Bug 检查的值为 0x00000067。 此 Bug 检查表明注册表配置失败。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

CONFIG_INITIALIZATION_FAILED 参数

展开表

参数	说明
1	保留
2	位置选择器
3	NT 状态代码
4	Reserved

原因

注册表无法分配包含注册表文件所需的池。 这种情况应该永远不会发生，因为寄存器在系统初始化时足够早地分配此池，从而应该有足够的分页池可用。

Bug 检查 0x68：FILE_INITIALIZATION_FAILED

项目 • 2024/02/20

FILE_INITIALIZATION_FAILED Bug 检查的值为 0x00000068。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x69：IO1_INITIALIZATION_FAILED

项目 • 2024/02/20

IO1_INITIALIZATION_FAILED Bug 检查的值为 0x00000069。 此 Bug 检查表明 I/O 系统的初始化由于某种原因失败。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

IO1_INITIALIZATION_FAILED 参数

无

原因

可用于分析此错误的信息非常少。

最有可能的情况是，安装例程未正确安装系统，或者用户重新配置了系统。

Bug 检查 0x6A：LPC_INITIALIZATION_FAILED

项目 • 2024/02/21

LPC_INITIALIZATION_FAILED Bug 检查的值为 0x0000006A。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x6B：PROCESS1_INITIALIZATION_FAILED

项目 • 2024/02/20

PROCESS1_INITIALIZATION_FAILED Bug 检查的值为 0x0000006B。 此 Bug 检查表明 Microsoft Windows 操作系统的初始化失败。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PROCESS1_INITIALIZATION_FAILED 参数

蓝屏上显示以下参数。

展开表

参数	说明
1	导致失败的 NT 状态代码
2	Reserved
3	Reserved
4	Reserved

原因

磁盘子系统的任何部分都可能导致PROCESS1_INITIALIZATION_FAILED Bug 检查，包括损坏的磁盘、损坏或错误的电缆、在同一链上混合不同的 ATA 类型设备，或因硬件重新生成而不可用的驱动器。

此 Bug 检查也可能是由于启动分区中丢失的文件或用户在“**驱动程序**”选项卡中意外禁用的驱动程序引起的。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

Bug 检查 0x6C：REFMON_INITIALIZATION_FAILED

项目 • 2024/02/21

REFMON_INITIALIZATION_FAILED Bug 检查的值为 0x0000006C。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x6D：SESSION1_INITIALIZATION_FAILED

项目 • 2024/02/20

SESSION1_INITIALIZATION_FAILED Bug 检查的值为 0x0000006D。 此 Bug 检查表明 Microsoft Windows 操作系统的初始化失败。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SESSION1_INITIALIZATION_FAILED 参数

展开表

参数	说明
1	导致初始化失败的 NT 状态代码
2	0
3	0
4	0

Bug 检查 0x6E: SESSION2_INITIALIZATION_FAILED

项目 • 2024/02/20

SESSION2_INITIALIZATION_FAILED Bug 检查的值为 0x0000006E。 此 Bug 检查表明 Microsoft Windows 操作系统的初始化失败。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SESSION2_INITIALIZATION_FAILED 参数

展开表

参数	说明
1	导致 Windows 操作系统得出初始化失败结论的 NT 状态代码
2	0
3	0
4	0

Bug 检查 0x6F：SESSION3_INITIALIZATION_FAILED

项目 • 2024/02/20

SESSION3_INITIALIZATION_FAILED Bug 检查的值为 0x0000006F。 此 Bug 检查表明 Microsoft Windows 操作系统的初始化失败。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SESSION3_INITIALIZATION_FAILED 参数

展开表

参数	说明
1	导致 Windows 操作系统得出初始化失败结论的 NT 状态代码
2	0
3	0
4	0

Bug 检查 0x70：SESSION4_INITIALIZATION_FAILED

项目 • 2024/02/20

SESSION4_INITIALIZATION_FAILED Bug 检查的值为 0x00000070。 此 Bug 检查表明 Microsoft Windows 操作系统的初始化失败。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SESSION4_INITIALIZATION_FAILED 参数

展开表

参数	说明
1	导致 Windows 操作系统得出初始化失败结论的 NT 状态代码
2	0
3	0
4	0

Bug 检查 0x71: SESSION5_INITIALIZATION_FAILED

项目 • 2024/02/20

SESSION5_INITIALIZATION_FAILED Bug 检查的值为 0x00000071。 此 Bug 检查表明 Microsoft Windows 操作系统的初始化失败。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SESSION5_INITIALIZATION_FAILED 参数

展开表

参数	说明
1	导致 Windows 操作系统得出初始化失败结论的 NT 状态代码
2	0
3	0
4	0

Bug 检查 0x72：ASSIGN_DRIVE_LETTERS_FAILED

项目 • 2024/02/20

ASSIGN_DRIVE_LETTERS_FAILED Bug 检查的值为 0x00000072。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x73：CONFIG_LIST_FAILED

项目 • 2024/02/20

CONFIG_LIST_FAILED Bug 检查的值为 0x00000073。 此 Bug 检查表明顶级注册表项之一（也称为核心系统配置单元）无法在注册表树中链接。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

CONFIG_LIST_FAILED 参数

展开表

参数	说明
1	1
2	导致 Windows 操作系统认为加载配置单元失败的 NT 状态代码
3	配置单元列表中配置单元的索引
4	指向包含配置单元文件名的 UNICODE_STRING 结构的指针

原因

无法链接的注册表配置单元可能是 SAM、SECURITY、SOFTWARE 或 DEFAULT。 配置单元是有效的，因为它已成功加载。

检查参数 2，了解无法在注册表树中链接配置单元的原因。 导致此错误的一个常见原因是 Windows 操作系统的系统驱动器磁盘空间不足。 （在这种情况下，该参数为 0xC000017D，STATUS_NO_LOG_SPACE。）另一个常见问题是尝试分配池失败。 （在这种情况下，参数 2 为 0xC000009A，STATUS_INSUFFICIENT_RESOURCES。）您必须调查其他状态代码。

错误检查 0x74：BAD_SYSTEM_CONFIG_INFO

项目 • 2024/10/16

BAD_SYSTEM_CONFIG_INFO 错误检查的值为 0x00000074。 此错误检查表明注册表中存在错误。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

BAD_SYSTEM_CONFIG_INFO 参数

展开表

参数	说明
1	保留
2	预留
3	预留
4	NT 状态值/代码（如果可用）

原因

如果 SYSTEM 配置单元损坏，则会进行 BAD_SYSTEM_CONFIG_INFO 错误检查。 但是，这种损坏不太可能发生，因为启动加载程序在其加载配置单元时会检查配置单元是否损坏。

如果缺少某些关键注册表项和值，也可能会出现此错误检查。 如果用户手动编辑了注册表，或者应用程序或服务损坏了注册表，则可能会缺少键和值。

查找参数 4 中返回的 NT 状态值可提供更多信息，请参阅 NTSTATUS 值列表。

解决方法

检查 Windows 系统事件日志，查看是否存在任何与注册表相关的错误事件。 如果存在，请查看事件是否列出了发生错误的配置单元或特定键。

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

dbgcmd

```
BAD_SYSTEM_CONFIG_INFO (74)
Can indicate that the SYSTEM hive loaded by the osloader/NTLDR
was corrupt.  This is unlikely, since the osloader will check
a hive to make sure it isn't corrupt after loading it.
It can also indicate that some critical registry keys and values
are not present.  (i.e. somebody used regedt32 to delete something
that they shouldn't have)  Booting from LastKnownGood may fix
the problem, but if someone is persistent enough in mucking with
the registry they will need to reinstall or use the Emergency
Repair Disk.
Arguments:
Arg1: 0000000000000002, (reserved)
Arg2: ffffd481054b49f0, (reserved)
Arg3: 0000000000000004, (reserved)
Arg4: ffffffffc000014c, usually the NT status code.
```

查看 !analyze 返回的所有信息，了解故障情况。

使用 !error 扩展显示有关参数 4 中 NTSTATUS 值的相关信息。

dbgcmd

```
2: kd> !ERROR ffffffffc000014c
Error code: (NTSTATUS) 0xc000014c (3221225804) - {The Registry Is Corrupt}
The structure of one of the files that contains Registry data is corrupt, or
the image of the file in memory is corrupt, or the file could not be
recovered because the alternate copy or log was absent or corrupt.
```

使用 !reg 扩展显示注册表的相关信息，例如注册表中存在的配置单元。

dbgcmd

```
!reg hivelist

---------------------------------------------------------------------------
---------------------------------------------------------------------------
|     HiveAddr     |Stable Length|    Stable Map    |Volatile Length|
Volatile Map    |MappedViews|PinnedViews|U(Cnt)|     BaseBlock     |
FileName
---------------------------------------------------------------------------
---------------------------------------------------------------------------
| ffff95077ea24000 |       1000  | ffff95077ea24588 |          0    |
```

```
0000000000000000  |     0| ffff95077ea31000  | <NONAME>
| ffff95077ea3e000 |    12d3000  | ffff95077ea49000 |      21000    |
ffff95077ea3e800  |     0| ffff95077ea40000  | SYSTEM
| ffff95077ea8f000 |      53000  | ffff95077ea8f588 |       9000    |
ffff95077ea8f800  |     0| ffff95077ea91000  | <NONAME>
| ffff9507821c8000 |       7000  | ffff9507821c8588 |          0    |
0000000000000000  |     0| ffff9507821cc000  |
kVolume2\EFI\Microsoft\Boot\BCD
| ffff95077f6ae000 |    685c000  | ffff95077f737000 |       6000    |
ffff95077f6ae800  |     0| ffff95077f6b6000  |
emRoot\System32\Config\SOFTWARE
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

使用 !reg openkeys 命令查看已打开的注册表项。

dbgcmd

```
2: kd> !reg openkeys

Hive: \REGISTRY\MACHINE\SYSTEM
================================================================================
===============
Index 0:     00000000 kcb=ffffd805e303c728 cell=00000020 f=002c0100
\REGISTRY\MACHINE\SYSTEM
Index 1:     db67f96d kcb=ffffd805e416ed18 cell=00bd0b40 f=00200080
\REGISTRY\MACHINE\SYSTEM\WPA\8DEC0AF1-0341-4B93-85CD-72606C2DF94C-7P-374
Index 3:     db67ee93 kcb=ffffd805e30c5ab8 cell=00bc1550 f=00200080
\REGISTRY\MACHINE\SYSTEM\WPA\8DEC0AF1-0341-4B93-85CD-72606C2DF94C-7P-161
Index 4:     f9909d96 kcb=ffffd805e44bd268 cell=00bf8f50 f=00200000
\REGISTRY\MACHINE\SYSTEM\CONTROLSET001\CONTROL\POWER\PROFILE\EVENTS\
{54533251-82BE-4824-96C1-47B60B740D00}\{8BC6262C-C026-411D-AE3B-
7E2F70811A13}
Index 5:     e9dd6ce5 kcb=ffffd805e4180e48 cell=00812970 f=00200000
\REGISTRY\MACHINE\SYSTEM\DRIVERDATABASE

...
```

注解

有关确定蓝屏原因的一般信息，请参阅分析错误检查蓝屏数据。

最好确保有足够的硬盘驱动器或 SSD 存储空间，以便操作系统正常运行。

系统文件检查器工具可以查找 Windows 中的损坏情况。 有关详细信息，请参阅使用系统文件检查器工具修复丢失或损坏的系统文件。

尝试启动到安全模式，然后正常重启操作系统。 如果重启不能解决问题，则说明注册表损坏范围太大。 尝试下列步骤。

- 如果有系统还原点，请尝试还原到较早的还原点。
- 重置电脑。
- 使用安装介质还原或重置电脑。
- 使用安装介质重新安装 Windows。

有关详细信息，请参阅 Windows 10 中的恢复选项 。

这篇支持文章讨论该错误检查代码：Error 0x74: Bad_system_config_info

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Bug 检查 0x75：CANNOT_WRITE_CONFIGURATION

项目 • 2024/02/21

CANNOT_WRITE_CONFIGURATION Bug 检查的值为 0x00000075。 此 Bug 检查表明 SYSTEM 注册表配置单元文件无法转换为映射文件。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

CANNOT_WRITE_CONFIGURATION 参数

展开表

参数	说明
1	1
2	导致 Windows 操作系统认为转换配置单元失败的 NT 状态代码
3	Reserved
4	Reserved

原因

如果系统超出池并且 Windows 操作系统无法重新打开配置单元，则通常会发生 CANNOT_WRITE_CONFIGURATION Bug 检查。

此 Bug 检查几乎永远不会发生，因为配置单元文件的转换在系统初始化期间发生地足够早，因此应该有足够的池可用。

Bug 检查 0x76：PROCESS_HAS_LOCKED_PAGES

项目 • 2024/02/20

PROCESS_HAS_LOCKED_PAGES Bug 检查的值为 0x00000076。 此 Bug 检查表明驱动程序在 I/O 操作后未能释放锁定的页面，或者它试图解锁已解锁的页面。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PROCESS_HAS_LOCKED_PAGES 参数

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x00	指向进程对象的指针	锁定的页数	指向驱动程序堆栈的指针（如果已启用）。 否则此参数为零。	被终止的进程已锁定内存页。 在进程终止之前，驱动程序必须解锁进程中可能锁定的任何内存。
0x01	驱动程序指定的 MDL	进程的当前锁定中内存页数	指向该进程的驱动程序堆栈的指针（如果已启用）。 否则此参数为零。	驱动程序正试图解锁未锁定的进程内存页。

原因

驱动程序未能解锁锁定的页面（参数 1 的值为 0x0），或者驱动程序正在试图解锁尚未锁定或已解锁的页面（参数 1 的值为 0x1）。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

如果参数 1 的值为 0x0

首先使用 !search 在整个物理内存中的当前进程指针上扩展。 此扩展可能会找到至少一个指向当前进程的内存描述符列表 (MDL)。 接下来，在找到的每个 MDL 上使用 **!search** 来获取指向当前进程的 I/O 请求数据包 (IRP)。 通过该 IRP，可以识别哪个驱动程序正在泄露页面。

否则，可以通过编辑注册表来检测导致错误的驱动程序：

1. 在 **\\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management** 注册表项中，创建或编辑 **TrackLockedPages** 值，然后将其设置为等于 DWORD 1。
2. 重新启动计算机。

然后，系统会保存堆栈跟踪，以便轻松识别导致问题的驱动程序。 如果驱动程序再次导致相同的错误，则会发出 **Bug 检查 0xCB** (DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS)，并在蓝屏上显示导致此错误的驱动程序名称，同时将其存储在内存中的位置 (PUNICODE_STRING) **KiBugCheckDriver** 中。

如果参数 1 的值为 0x1

检查锁定和解锁内存的驱动程序源代码，并尝试内存在未锁定的情况下被解锁的实例。

Bug 检查 0x77：KERNEL_STACK_INPAGE_ERROR

项目 • 2024/02/20

KERNEL_STACK_INPAGE_ERROR Bug 检查的值为 0x00000077。 此 Bug 检查表明无法将分页文件中请求的内核数据页读入内存。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

KERNEL_STACK_INPAGE_ERROR 参数

消息中列出的四个参数有两种可能的含义。

如果第一个参数为 0、1 或 2，则参数的含义如下。

展开表

参数	说明
1	**0**：内核数据页是从页面缓存中检索的。 **1**：页面是从磁盘中检索的。 **2**：页面是从磁盘中检索的，存储堆栈返回 SUCCESS，但 **Status.Information** 不等于 PAGE_SIZE。
2	出现在堆栈中签名所在位置的值。
3	0
4	内核堆栈上签名的地址

如果第一个参数是 0、1 或 2 以外的任意值，则参数的含义如下。

展开表

参数	说明
1	状态代码

参数	说明
2	I/O 状态代码
3	页文件编号
4	页文件中的偏移量

原因

如果第一个参数为 0 或 1，则表示未找到内核堆栈中的堆栈签名。 此错误可能是由有缺陷的硬件引起的，例如 RAM 错误。

如果第一个参数为 2，驱动程序堆栈返回页面读取的不一致状态。 例如，即使驱动程序堆栈未读取整个页面，也返回了成功状态。

如果第一个参数是 0、1 或 2 以外的任意值，则第一个参数的值是驱动程序堆栈在尝试检索内核数据页后返回的 NTSTATUS 错误代码。 可以通过 I/O 状态代码（第二个参数）确定该错误的确切原因。 一些常见状态代码包括以下内容：

- 0xC000009A 或 STATUS_INSUFFICIENT_RESOURCES 表示缺少非分页池资源。 此状态代码表示存储堆栈中存在驱动程序错误。 （无论软件资源是否可用，存储堆栈都应该始终能够检索此数据。）
- 0xC000009C 或 STATUS_DEVICE_DATA_ERROR 表示硬盘上有坏块（扇区）。
- 0xC000009D 或 STATUS_DEVICE_NOT_CONNECTED 表示布线、端接有有缺陷或松动，或者控制器看不到硬盘驱动器。
- 0xC000016A 或 STATUS_DISK_OPERATION_FAILED 表示硬盘上有坏块（扇区）。
- 0xC0000185 或 STATUS_IO_DEVICE_ERROR 表示 SCSI 设备上的端接不当或布线有缺陷，或者两台设备尝试使用相同的 IRQ。

这些状态代码是具有特定原因的最常见状态代码。 有关可能返回的其他状态代码的详细信息，请参阅 Microsoft Windows 驱动程序工具包 (WDK) 中的 Ntstatus.h 文件。

病毒感染也会导致这种 Bug 检查。

解决方法

解决坏块问题：如果出错后可以重新启动计算机，Autochk 会自动运行并尝试映射坏扇区，以防止其再被使用。

如果 Autochk 未扫描硬盘中的错误，可以手动启动磁盘扫描程序。 在系统分区上运行 **Chkdsk /f /r**。 在磁盘扫描开始前，必须重新启动计算机。 如果由于错误而无法启动系统，请使用恢复控制台并运行 **Chkdsk /r**。

警告 如果系统分区是用 FAT 文件系统进行了格式化，在使用 Scandisk 或其他基于 MS-DOS 的硬盘工具通过 MS-DOS 验证硬盘驱动器的完整性时，Windows 操作系统使用的长文件名可能会被损坏。 请始终使用与 Windows 操作系统版本匹配的 Chkdsk 版本。

解决有缺陷的硬件问题：如果 I/O 状态为 0xC0000185 并且分页文件位于 SCSI 磁盘上，请检查磁盘布线和 SCSI 终端问题。

解决 RAM 故障问题：运行系统制造商提供的硬件诊断程序，尤其是内存扫描程序。 有关这些过程的详细信息，请参阅计算机的用户手册。

检查计算机中的所有适配器卡是否已正确安置。 使用电子用品商店出售的墨迹橡皮擦或电气触点处理剂，确保适配器卡接触点干净无尘。

在事件查看器中检查系统日志以获取更多错误消息，这可能有助于识别导致错误的设备。 也可以禁用 BIOS 的内存缓存以尝试解决此错误。

确保安装了最新的 Windows Service Pack。

如果上述步骤无法解决错误，请将系统主板送到维修机构进行诊断测试。 主板上的裂缝、划痕或有缺陷的组件都可能会导致此错误。

解决感染病毒：使用任何可检查硬盘主启动记录的最新商业病毒扫描软件，检查计算机是否存在病毒。 所有 Windows 文件系统都可能被病毒感染。

另请参阅

Bug 检查 0x7A (KERNEL_DATA_INPAGE_ERROR)

Bug 检查 0x78：PHASE0_EXCEPTION

项目 • 2024/02/20

PHASE0_EXCEPTION Bug 检查的值为 0x00000078。

此 Bug 检查在 HAL 初始化期间遇到意外中断时发生。 如果在启动设置中设置了 **/break** 参数，但未启用内核调试，则会发生此中断。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x79：MISMATCHED_HAL

项目 • 2024/02/20

MISMATCHED_HAL Bug 检查的值为 0x00000079。 此 Bug 检查表明硬件抽象层 (HAL) 修订级别或配置与内核或计算机的版本不匹配。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

MISMATCHED_HAL 参数

参数 1 表示不匹配的类型。

展开表

参数 1	参数 2	参数 3	参数 4	原因。
0x1	Ntoskrnl.exe 的主要处理器控制块 (PRCB) 级别。	Hal.dll 的主要 PRCB 级别。	Reserved	PRCB 发布级别不匹配。 （部分内容已过期。）
0x2	Ntoskrnl.exe 的生成类型。	Hal.dll 的生成类型。	Reserved	生成类型不匹配。
0x3	加载程序参数扩展的大小。	加载程序参数扩展的主要版本。	加载程序参数扩展的次要版本。	加载程序 (ntldr) 和 HAL 版本不匹配。

当参数 1 等于 0x2 时，将使用以下生成类型代码：

- 0：支持多处理器的免费生成
- 1：支持多处理器的检查生成
- 2：单处理器免费生成
- 3：单处理器检查生成

原因

当用户手动更新 Ntoskrnl.exe 或 Hal.dll 时，通常会发生 MISMATCHED_HAL Bug 检查。

该错误还可能表明这两个文件之一已过期。 或者计算机可能错误地安装了多处理器 HAL 和单处理器内核，反之亦然。

Ntoskrnl.exe 内核文件适用于单处理器系统，Ntkrnlmp.exe 适用于多处理器系统。 但是，这些文件名与安装介质上的文件相对应。安装 Windows 操作系统后，无论使用哪个源文件，该文件都会重命名为 Ntoskrnl.exe。 安装后，HAL 文件也使用名称Hal.dll，但在较旧版本的 Windows 上的安装介质上有几个可能的 HAL 文件。

解决方法

使用产品 CD 或 Windows 安装程序磁盘重启计算机。 在欢迎屏幕上，按 F10 键启动恢复控制台。 使用 **Copy** 命令将正确的 HAL 或内核文件从原始 CD 复制到硬盘上的相应文件夹中。 “**复制**” 命令会检测要复制的文件是否为 Microsoft 压缩文件格式。 如果是，它会自动扩展目标驱动器上复制的文件。

Bug 检查 0x7A：KERNEL_DATA_INPAGE_ERROR

项目 • 2024/02/21

KERNEL_STACK_INPAGE_ERROR Bug 检查的值为 0x0000007A。 此 Bug 检查表明无法将分页文件中请求的内核数据页读入内存。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

参数

消息中列出的四个参数可能具有三种含义。

如果第一个参数为 1、2 或 3，第三个参数为 0，则参数的定义如下。

展开表

参数	说明
1	持有的锁类型（1、2 或 3）
2	错误状态（通常是 I/O 状态代码）
3	如果锁类型为 1：当前进程 如果锁类型为 2 或 3：0
4	无法分页到内存中的虚拟地址

如果第一个参数为 3或 4（第三个参数不为 0），则参数的定义如下。

展开表

参数	说明
1	持有的锁类型（3 或 4）
2	错误状态（通常是 I/O 状态代码）
3	InPageSupport 结构的地址

参数	说明
4	故障地址

否则，参数定义如下。

展开表

参数	说明
1	页表项 (PTE) 的地址
2	错误状态（通常是 I/O 状态代码）
3	PTE 内容
4	故障地址

原因

通常情况下，可以通过错误状态（参数 2）确定 KERNEL_DATA_INPAGE_ERROR Bug 检查的原因。 这些是 NTSTATUS 值。

一些常见的状态代码包括：

- 0xC000009A 或 STATUS_INSUFFICIENT_RESOURCES 表示缺少非分页池资源。
- 0xC000009C 或 STATUS_DEVICE_DATA_ERROR 通常表示硬盘上有坏块（扇区）。
- 0xC000009D 或 STATUS_DEVICE_NOT_CONNECTED 表示布线、端接有有缺陷或松动，或者控制器看不到硬盘。
- 0xC000016A 或 STATUS_DISK_OPERATION_FAILED 表示硬盘上有坏块（扇区）。
- 0xC0000185 或 STATUS_IO_DEVICE_ERROR 表示 SCSI 设备上的端接不当或布线有缺陷，或者两台设备尝试使用相同的 IRQ。
- 0xC000000E 或 STATUS_NO_SUCH_DEVICE 表示硬件故障或驱动器配置不正确。 检查电缆，并使用驱动器制造商提供的诊断实用工具检查驱动器。 如果使用的是较旧的 PATA (IDE) 驱动器，此状态代码可能表示主/从驱动器配置不正确。

这些状态代码是具有特定原因的最常见状态代码。 有关特定 Windows 版本可能返回的其他状态代码的详细信息，请参阅 `Ntstatus.h` Microsoft Windows 驱动程序工具包 (WDK) 中的文件。 OSR 第三方提供了将 NTSTATUS 映射到 Win32 错误代码的 PDF 文档 NTSTATUS 到 Win32 错误代码的映射

导致此错误消息的另一个常见原因是存储硬件有缺陷或 RAM 内存故障。

病毒感染也会导致这种 Bug 检查。

解决方法

解决坏块问题： 0xC000009C 或 0xC000016A 的 I/O 状态代码通常表示由于存在坏块（扇区）而无法从磁盘读取数据。 如果出错后可以重新启动计算机，Autochk 会自动运行并尝试映射坏扇区，以防止其再被使用。

如果 Autochk 未扫描硬盘中的错误，可以手动启动磁盘扫描程序。 在系统分区上运行 `Chkdsk /f /r` 。 在磁盘扫描开始前，必须重新启动计算机。 如果由于错误而无法启动计算机，请使用恢复控制台并运行 `Chkdsk /r`。

解决 RAM 故障问题：运行系统制造商提供的硬件诊断程序，尤其是内存扫描程序。 有关这些过程的详细信息，请参阅计算机的用户手册。

解决有缺陷的硬件问题：如果 I/O 状态为 C0000185 并且分页文件位于较旧的 SCSI 磁盘上，检查磁盘布线和 SCSI 终端问题。

解决感染病毒：使用任何可检查硬盘主启动记录的最新商业病毒扫描软件，检查计算机是否存在病毒。 Windows 文件系统有可能感染病毒。

检查系统事件查看器获取错误消息，这可能有助于识别导致错误的设备。 尤其应检查 Bug 检查之前发生的错误。

确保已安装最新的 Windows 更新。

找到并运行任何可用的电脑诊断测试软件。 主板上的裂缝、划痕或有缺陷的组件都可能会导致此 Bug 检查。

对于带卡槽的电脑，检查计算机中的所有适配器卡是否都已正确安置。 使用电子用品商店出售的墨迹橡皮擦或电气触点处理剂，确保适配器卡接触点干净无尘。

另请参阅

Bug 检查 0x77：KERNEL_STACK_INPAGE_ERROR

Bug 检查 0x7b：INACCESIBLE_BOOT_DEVICE

项目 • 2024/02/20

INACCESSIBLE_BOOT_DEVICE Bug 检查的值为 0x0000007B。 此 Bug 检查表明 Microsoft Windows 操作系统在启动期间失去了对系统分区的访问权限。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

参数

展开表

参数	说明
1	UNICODE_STRING 结构的地址，或无法装载的设备对象的地址
2	0
3	0
4	0

若要确定参数 1 的含义，请查看它指向的数据。 如果此地址的第一个单词 (USHORT) 为偶数，则参数 1 是 Unicode 字符串的开头。 如果此地址的第一个单词 (USHORT) 是 0x3，则参数 1 是设备对象的第一个字段（类型）。

- 如果此参数指向设备对象，则表示本应读取启动设备的文件系统初始化失败或没有将启动设备上的数据识别为文件系统结构。 在这种情况下，指定的设备对象是无法装载的对象。
- 如果此参数指向 Unicode 字符串，则必须读取此地址的前 8 个字节。 这些字节构成 UNICODE_STRING 结构。 以下示例演示了如何定义该结构：

```
C++

USHORT Length;
USHORT MaximumLength;
```

```
PWSTR Buffer;
```

“**长度**”字段提供字符串的实际长度。 **缓冲区**字段指向字符串的开头。 **缓冲区**始终至少为 0x80000000。

该字符串包含尝试启动的设备的高级 RISC 计算 (ARC) 规范名称。 ARC 名称是识别 ARC 环境中设备的通用方法。

原因

INACCESSIBLE_BOOT_DEVICE Bug 检查经常因为启动设备故障而发生。 在 I/O 系统初始化期间，启动设备驱动程序可能无法初始化启动设备，通常会是硬盘。

文件系统初始化可能已失败，因为它无法识别启动设备上的数据。 重新划分系统分区、更改 BIOS 配置或安装磁盘控制器也可能导致此错误。

磁盘硬件不兼容可能会导致此错误。 如果错误发生在系统的初始设置中，则系统可能已安装在不受支持的磁盘控制器上。 某些磁盘控制器要求 Windows 启动时存在其他驱动程序。

当存储硬件出现故障并且无法响应来自 Windows 的请求时，可能会出现此错误。

解决方法

该错误总是在系统启动时发生。 此错误经常发生在建立调试器连接之前，因此可能很困难调试。 操作系统可能无法访问，错误日志可能为空，因为操作系统尚未启动到足以启动这些子系统的程度。 以下部分介绍了这两种情况下的解决方法，即无法启动 Windows 和可以启动 Windows。

如果无法启动 Windows

如果收到此停止代码，并且 Windows 无法启动进入操作系统，请尝试以下解决方法：

- **还原最近的任何硬件更改。**

  删除最近添加的所有硬件，尤其是硬盘驱动器或控制器，以查看错误是否已解决。 如果有问题的硬件是硬盘驱动器，则磁盘固件版本可能与您的 Windows 操作系统版本不兼容。 请联系制造商获取更新信息。 如果删除了另一个硬件，且错误已经解决，则可能存在 IRQ 或 I/O 端口冲突。 根据制造商的说明重新配置新设备。

  如果最近对 UEFI (BIOS) 设置进行了更改，例如在 UEFI 中将控制器模式从传统模式更改为 AHCI，请恢复这些更改。 有关详细信息，请参阅 高级主机控制器接口 。

- **检查存储设备兼容性。**

  确认所有硬盘驱动程序、硬盘控制器和任何其他存储适配器都与已安装的 Windows 版本兼容。 例如，可以在 Windows 10 规范 中获取兼容性的相关信息。

- **更新 UEFI (BIOS) 和固件。**

  检查系统 UEFI (BIOS) 和存储控制器固件更新的可用性。

- **使用 Windows Media Creation 工具创建可启动的 U 盘或 DVD。**

  使用另一台计算机上的媒体创建工具创建可启动的 U 盘或 DVD。 使用此工具通过选择安装文件或从 USB 启动来执行全新安装。

  有关详细信息，请参阅获取 Windows 10 。

  可能需要禁用功能，或更改 UEFI (BIOS) 菜单中的启动顺序优先级，以从 USB、FDD (FlashDiskDrive) 或 DVD 启动，而不是 HDD。

  常见的启动菜单键

  启动菜单键因制造商而异。 这些是常用键。 查看电脑文档以确定使用的启动键。

  常用的启动菜单键包括：
  F12
  ESC
  F9
  F10
  F8

  常见的 UEFI (BIOS) 设置键

  UEFI (BIOS) 设置键因制造商而异。 这些是常用键。 查看电脑文档以确定使用的设置键。

  常用的 UEFI (BIOS) 设置键包括：
  ESC
  DEL
  F2

如果可以启动 Windows

如果收到此停止代码并可以启动 Windows，请尝试以下解决方法：

- **启动到安全模式，然后正常启动。**

启动进入安全模式后，系统会加载一组核心存储驱动程序，以便再次访问系统。 完成以下步骤以启动进入安全模式：

1. 在“**设置**”中，选择“**更新和安全**”。
2. 选择“**恢复**”>“**高级启动**”以启动进入维护模式。
3. 在生成的菜单中，选择“**高级选项**”>“**故障排除**”>“**启动设置**”>“**重启**”。
4. Windows 重启进入“**启动设置**”界面后，选择选项 4、5 或 6 以启动进入安全模式。

在安全模式下加载 Windows 后，重启电脑以查看是否加载了正确的存储驱动程序，以及是否识别了存储设备。

在启动时按下功能键（例如 F8）也可以进入安全模式。 有关具体的启动选项，请参阅系统制造商提供的信息。

- 使用扫描磁盘实用工具确认没有文件系统错误。 选择并按住（或右键单击）要扫描的驱动器，然后选择“**属性**”>“**工具**”>“**立即检查**”。

- 运行病毒检测程序。 病毒可感染针对 Windows 格式化的所有类型的硬盘，由此导致的磁盘损坏可能会生成系统 Bug 检查代码。 确保病毒检测程序检查主启动记录是否存在感染。

- 对于 IDE 设备，仅将板载 IDE 端口定义为主端口。 此外还要检查每个 IDE 设备的**主/从/独立**设置是否正确。 请尝试删除除硬盘以外的所有 IDE 设备。 最后，在事件查看器中检查系统日志以获取其他错误消息，这可能有助于识别导致错误的设备或驱动程序。

- 确认硬盘驱动器上有足够的可用空间。 操作系统和某些应用程序需要足够的可用空间来创建交换文件和执行其他功能。 确切的要求因系统配置而异，但最好有 10% 到 15% 的可用空间。

- 在**设备管理器**中查看是否有任何设备标有感叹号 (!)。 查看驱动程序属性中显示的事件日志，以查找故障驱动程序。 请尝试更新相关驱动程序。

- 在事件查看器中检查系统日志以获取其他错误消息，这可能有助于发现导致错误的设备或驱动程序。 有关详细信息，请参阅打开事件查看器。 在系统日志中查找与蓝屏同时出现的严重错误。

- 你可尝试运行系统制造商提供的硬件诊断。

- 使用系统文件检查器工具修复丢失或损坏的系统文件。 系统文件检查器是 Windows 中的一个实用工具，供用户扫描 Windows 系统文件中的损坏情况并恢复损坏的文件。 使用以下命令运行系统文件检查器工具 (SFC.exe)。

```
控制台
```

```
SFC /scannow
```

有关详细信息，请参阅使用系统文件检查器工具修复丢失或损坏的系统文件。

- 自动修复后，在“**选择选项**”屏幕上，选择>“**高级选项**”>“**系统还原**”。 该选项可将电脑还原到一个较早的时间点，称为系统还原点。 安装新应用、驱动程序、更新或手动创建还原点时，会生成还原点。 请选择出现错误之前的还原点。
- 使用内核调试器附加到系统，并按照备注中的描述进一步分析故障。

注解

调查存储系统配置。
若要缩小原因范围，请尽可能多了解安装 Windows 的启动设备。 例如，可以调查以下项目：

- 找出启动设备连接到的控制器类型，例如 SATA 或 IDE。 如果可以启动系统，则可以使用设备管理器检查控制器和磁盘驱动程序属性。 可以看到关联的驱动程序文件和错误事件。
- 指示是否有其他设备连接到启动设备所在的同一控制器，例如 SSD 或 DVD。
- 请注意驱动器上使用的文件系统，通常是 NTFS。

Windows 调试器

若要使用内核调试器分析此错误，请在调试器中运行 lm（List loaded modules，列出加载的模块） 命令，以查看加载了哪些模块以尝试隔离特定驱动程序。 验证是否已加载以下驱动程序。

disk

```
dbgcmd
```

```
0: kd> lm m disk
Browse full module list
start             end                 module name
fffff806`bd0b0000 fffff806`bd0cd000   disk       (deferred)
```

partmgr

```
dbgcmd
```

```
0: kd> lm m partmgr
Browse full module list
start             end                 module name
fffff806`bc5a0000 fffff806`bc5c1000   partmgr    (deferred)
```

NTFS

```
dbgcmd
```

```
0: kd> lm m ntfs
Browse full module list
start             end                 module name
fffff806`bd3f0000 fffff806`bd607000   NTFS       (deferred)
```

classpnp

```
dbgcmd
```

```
0: kd> lm m classpnp
Browse full module list
start             end                 module name
fffff806`bd0d0000 fffff806`bd131000   CLASSPNP   (deferred)
```

PCI

```
dbgcmd
```

```
0: kd> lm m pci
Browse full module list
start             end                 module name
fffff806`bc440000 fffff806`bc494000   pci        (deferred)
```

确保已加载控制器驱动程序。 例如，对于 SATA RAID 控制器，驱动程序可能是 *iaStorA.Sys* 驱动程序，也可能是 *EhStorClass* 驱动程序。

```
dbgcmd
```

```
0: kd> lm m EhStorClass
Browse full module list
start             end                 module name
fffff806`bcbb0000 fffff806`bcbcb000   EhStorClass   (deferred)
```

可能存在包含“stor”的驱动程序，如 storahci。

```
dbgcmd
```

```
0: kd> lm m stor*
Browse full module list
start             end                 module name
fffff806`bcb00000 fffff806`bcb23000   storahci   (deferred)
fffff806`bcb30000 fffff806`bcbaa000   storport   (deferred)
fffff806`c0770000 fffff806`c0788000   storqosflt   (deferred)
```

使用附加调试器启动。

如果可以在连接调试器的情况下启动目标系统，则在出现 Bug 检查时发出 !devnode 0 1。 可以看到哪个设备缺少驱动程序或未启动，而未启动的原因可能很明显。

其中一个原因可能是即插即用无法将资源分配给启动设备。 可以通过查找服务条目来验证此限制。 如果状态标志包括 DNF_INSUFFICIENT_RESOURCES，或者不包含 DNF_STARTED 或 DNF_ENUMERATED，则可能已经找到问题。 尝试 `!devnode 0 1 storahci` 节省一些时间，而不是转储整个设备树。

dbgcmd

```
0: kd> !devnode 0 1 storahci
Dumping IopRootDeviceNode (= 0xffffb9053d94d850)
DevNode 0xffffb9053e8dea50 for PDO 0xffffb9053e8da060
  InstancePath is
"PCI\VEN_8086&DEV_3B22&SUBSYS_304A103C&REV_05\3&21436425&0&FA"
  ServiceName is "storahci"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0xffffb9053e88db30 for PDO 0xffffb9053e890060
    InstancePath is "SCSI\Disk&Ven_&Prod_ST3500418AS\4&23d99fa2&0&000000"
    ServiceName is "disk"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0xffffb9053e88d850 for PDO 0xffffb9053e88e060
    InstancePath is "SCSI\CdRom&Ven_hp&Prod_DVD-
RAM_GH60L\4&23d99fa2&0&010000"
    ServiceName is "cdrom"
    TargetDeviceNotify List - f 0xffffdf0ae9bbb0e0  b 0xffffdf0aea874710
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
```

Bug 检查 0x7C：BUGCODE_NDIS_DRIVER

项目 • 2024/02/21

BUGCODE_NDIS_DRIVER Bug 检查的值为 0x0000007C。 此 Bug 检查表明操作系统检测到网络驱动程序中错误。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅蓝屏错误疑难解答。

BUGCODE_NDIS_DRIVER 参数

参数 1 表示违规类型。 其他参数的含义取决于参数 1 的值。 如果参数的值为“0”，则表示未使用此参数。

展开表

参数 1	参数 1 值和错误原因	参数 2
0x01	NDIS_BUGCHECK_ALLOCATE_SHARED_MEM_HIGH_IRQL 一个驱动程序在高 IRQL 下调用了 NdisMAllocateSharedMemory。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。
0x02	NDIS_BUGCHECK_SHARED_MEM_CORRUPTION 在调用 NdisMAllocateSharedMemory 时，NDIS 检测到以前分配的共享内存页已损坏。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。
0x03	NDIS_BUGCHECK_FREE_INVALID_SHARED_MEM 一个微型端口驱动程序调用了 NdisMFreeSharedMemory(Async)，其共享内存地址已被释放。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。
0x04	NDIS_BUGCHECK_UNLOAD_DRIVER_INVALID_PARAMETER 调用 AddDevice 时使用的驱动程序不在 NDIS 注册的驱动程序列表中。 仅在特殊检测的 NDIS 上启用。	NDIS_M_DRIVER_BLOCK 的地址
0x05	NDIS_BUGCHECK_RECVD_PACKET_IN_USE_BAD_STACK_LOCATION 以太网驱动程序指示它使用协议堆栈当前正在使用的数据包描述符收到了数据包。 通过检查堆栈数据包位置捕获。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。
0x06	NDIS_BUGCHECK_RECVD_PACKET_IN_USE_BAD_REF_COUNT 以太网驱动程序指示它使用协议堆栈当前正在使用的数据包描述符收到了数据包。 通过检查数据包引用计数捕获。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。
0x07	FDDI 驱动程序指示它使用协议堆栈当前正在使用的数据包描述符收到了数据包。 通过检查引用计数捕获。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。
0x08	NDIS_BUGCHECK_HALT_WITHOUT_INTERRUPT_DEREGISTER 微型端口驱动程序在停止过程中未取消注册其中断。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。
0x09	NDIS_BUGCHECK_HALT_WITHOUT_CANCEL_TIMER 微型端口驱动程序在未成功取消其所有计时器的情况下停止。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。
0x0A	NDIS_BUGCHECK_DRIVER_UNLOAD_UNEXPECTED 微型端口驱动程序过早卸载。	NDIS_M_DRIVER_BLOCK 的地址
0x0B	NDIS_BUGCHECK_INIT_FAILED_WITHOUT_INTERRUPT_DEREGISTER 微型端口驱动程序在未取消注册中断的情况下初始化失败。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。

参数 1	参数 1 值和错误原因	参数 2	
0x0C	NDIS_BUGCHECK_INIT_FAILED_WITHOUT_CANCEL_TIMER 微型端口驱动程序在未成功取消其所有计时器的情况下初始化失败。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。	
0x0D	NDIS_BUGCHECK_HALT_INIT_WITHOUT_INTERRUPT_DEREGISTER 微型端口驱动程序在停止过程中未取消注册其中断。 在微型端口驱动程序从初始化处理程序返回成功后，从初始化例程调用了停止。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。	
0x0E	NDIS_BUGCHECK_HALT_INIT_WITHOUT_CANCEL_TIMER 微型端口驱动程序在未成功取消其所有计时器的情况下停止。 在微型端口驱动程序从初始化处理程序返回成功后，从初始化例程调用了停止。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。	
0x0F	NDIS_BUGCHECK_RESET_COMPLETE_UNEXPECTED 一个微型端口驱动程序调用了 NdisMResetComplete，没有任何挂起的重置请求。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。	
0x10	NDIS_BUGCHECK_PM_INIT_FAILED_NO_INT_DEREGISTER 从低功率状态恢复后，微型端口驱动程序在没有取消注册其中断的情况下初始化失败。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。	
0x11	NDIS_BUGCHECK_PM_INIT_FAILED_NO_CANCEL_TIMER 从低功率状态恢复后，微型端口驱动程序在未成功取消其所有计时器的情况下初始化失败。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。	
0x12	NDIS_BUGCHECK_NFILTER_RECVD_PACKET_BAD_REF_COUNT 微型端口驱动程序指示它使用协议堆栈当前正在使用的数据包描述符收到了数据包。 通过检查数据包引用计数捕获。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。	
0x13	NDIS_BUGCHECK_TFILTER_RECVD_PACKET_BAD_REF_COUNT 令牌环微型端口驱动程序指示它使用协议堆栈当前正在使用的数据包描述符收到了数据包。 通过检查数据包引用计数捕获。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。	
0x14	NDIS_BUGCHECK_WAIT_EVENT_HIGH_IRQL NDIS 驱动程序在非法 IRQL 下调用了 NdisWaitEvent	实际 IRQL	
0x15	NDIS_BUGCHECK_INVALID_NDIS5_CALL 微型端口驱动程序调用了为旧版驱动程序保留的 API。 驱动程序应仅调用 NDIS 6.x API。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。	
0x16	NDIS_BUGCHECK_INVALID_OPEN_IN_BIND_CONTEXT 协议驱动程序在绑定过程中未正确打开适配器。	特定协议的地址。 请使用此地址运行 !ndiskd.protocol 获取详细信息。	
0x17	NDIS_BUGCHECK_IFPROVIDER_DEREGISTER_UNEXPECTED 接口提供程序在未先删除其所有接口的情况下调用了 NdisIfDeregisterProvider。	接口提供程序句柄的地址。 请使用此地址运行 !ndiskd.ifprovider 获取详细信息。	
0x1B	NDIS_BUGCHECK_IF_STACK_TABLE_LOOP 驱动程序试图将接口添加到 ifStackTable，但这样做会导致循环。 ifStackTable 不能有循环。 运行 !ndiskd.ifstacktable 查看当前表（调用 NdisIfAddIfStackEntry）。	添加到表中的 HigherLayerIfIndex	
0x1C	NDIS_BUGCHECK_MINIPORT_FAILED_OID_WHICH_MUST_SUCCEED 微型端口驱动程序未成功处理一个不得失败的 OID 请求。 这样做会泄漏内存或其他资源。	特定微型端口适配器块的地址。 请使用此地址运行 !ndiskd.netadapter 获取详细信息。	

参数 1	参数 1 值和错误原因	参数 2
0x1D	NDIS_BUGCHECK_OID_REQUEST_INVALID_BUFFER 微型端口驱动程序或筛选器驱动程序非法完成了 OID 请求。 检查 BytesWritten 是否不大于缓冲区的整个长度。	特定微型端口适配器或筛选器模块块的地址。 请使用此地址运行 !ndiskd.netadapter 或 !ndiskd.filter 获取详细信息。
0x1E	NDIS_BUGCHECK_REFCOUNT_IMBALANCE NDIS 检测到内部 refcount 中有错误。 这可能是由引用下溢（取消引用多于引用）或标记不匹配引起的。	0
0x1F	NDIS_BUGCHECK_ILLEGAL_MINIPORT_STATE_TRANSITION 微型端口驱动程序非法完成了状态转换。	失败的项。 可能的值： 1. NDIS_BUGCHECK_ILLEGAL_MINIPORT_STATE_TRANSITION_PAUSE_COMPLETE 微型端口调用了 **NdisMPauseComplete**，但没有挂起的暂停操作。 2. NDIS_BUGCHECK_ILLEGAL_MINIPORT_STATE_TRANSITION_RESTART_COMPLETE 微型端口调用了 **NdisMRestartComplete**，但没有挂起的重启操作。
0x20	NDIS_BUGCHECK_STATUS_INDICATION_INVALID_BUFFER 微型端口驱动程序或筛选器驱动程序指示非法 NDIS_STATUS_INDICATION。	状态指示的类型。 请使用此代码运行 !ndiskd.help 获取详细信息。
0x21	NDIS_BUGCHECK_INVALID_OBJECT_HEADER 驱动程序创建了无效的 NDIS_OBJECT_HEADER。	指示非法状态指示的驱动程序句柄。 使用此句柄运行 !ndiskd.minidriver 或 !ndiskd.filterdriver，了解详细信息。
0x22	NDIS_BUGCHECK_ILLEGAL_NET_PNP_EVENT 微型端口驱动程序或筛选器驱动程序指示非法 NET_PNP_EVENT_NOTIFICATION。	指示非法状态指示的驱动程序句柄。 使用此句柄运行 !ndiskd.minidriver 或 !ndiskd.filterdriver，了解详细信息。
0x23	NDIS_BUGCHECK_PD_ERROR 在 Packet Direct 数据路径中检测到错误。	bugcheck 的子类型。 可能的值： 1. NDIS_BUGCHECK_PD_ERROR_EC_THREAD_MISMATCH 调用 API 的线程有误。 2. NDIS_BUGCHECK_PD_ERROR_ILLEGAL_ARM_BY_CLIENT PD 客户端试图在非法状态下武装提供程序。 3. NDIS_BUGCHECK_PD_ERROR_ILLEGAL_ARM_NOTIFICATION PD 提供程序在未布防的情况下非法触发了排空通知。 4. NDIS_BUGCHECK_PD_ERROR_ILLEGAL_ARM_NOTIFICATION_VIA_ISR PD 提供程序在未布防的情况下非法触发了 ISR 排空通知。 5. NDIS_BUGCHECK_PD_ERROR_ILLEGAL_THUNK_BY_LWF 筛选器驱动程序试图干扰 Packet Direct 数据路径。 6. NDIS_BUGCHECK_PD_ERROR_ILLEGAL_BM_GROUP_REQUEST PD 提供程序非法试图将自己从缓冲区管理器组中删除。 7. NDIS_BUGCHECK_PD_ERROR_ILLEGAL_PD_BUFFER_SETUP PD 缓冲区设置请求格式错误。
0x24	NDIS_BUGCHECK_UNEXPECTED_FAILURE 内部操作意外失败。 这可能是 NDIS.SYS 本身的 Bug。	操作失败。 可能的值： 0x01 : NDIS_BUGCHECK_UNEXPECTED_FAILURE_KEWAITFORSINGLEOBJECT 对 KeWaitForSingleObject 的调用失败。
0x25	NDIS_BUGCHECK_WATCHDOG 管理网络堆栈的尝试耗时过长。 当 NDIS 调用其他驱动程序时，NDIS 会启动监视器计时器，以确保调用及时完成。 如果调用时间过长，NDIS 会注入 Bug 检查。	操作耗时过长。 可能的值： - 0x01 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_PAUSE 暂停协议驱动程序时出现超时。 - 0x02 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_NETPNPEVENT

参数 1	参数 1 值和错误原因	参数 2	
	这可能是由简单的死锁造成的。 使用“!stacks 2 ndis”或类似命令查看是否有可疑线程。 请特别注意 NDIS_WATCHDOG_TRIAGE_BLOCK中的 PrimaryThread。 这可能是 NBL 丢失造成的，在这种情况下 !ndiskd.pendingnbls 可能会有所帮助。 检查使用 !ndiskd.oid 卡住的 OID。	向协议驱动程序传送 NET_PNP_EVENT_NOTIFICATION 时出现超时。 • 0x03 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_STATUS_INDICATION 向协议驱动程序传送状态指示时出现超时。 • 0x04 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_UNBIND 取消绑定协议驱动程序时出现超时。 • 0x11 : NDIS_BUGCHECK_WATCHDOG_FILTER_PAUSE 暂停筛选器驱动程序时出现超时。 • 0x12 : NDIS_BUGCHECK_WATCHDOG_FILTER_NETPNPEVENT 向筛选器驱动程序传送 NET_PNP_EVENT_NOTIFICATION 时出现超时。 • 0x13 : NDIS_BUGCHECK_WATCHDOG_FILTER_STATUS_INDICATION 向筛选器驱动程序传送状态指示时出现超时。 • 0x14 : NDIS_BUGCHECK_WATCHDOG_FILTER_DETACH 拆离筛选器驱动程序时出现超时。 • 0x21 : NDIS_BUGCHECK_WATCHDOG_MINIPORT_PAUSE 暂停微型端口适配器时出现超时。 • 0x22 : NDIS_BUGCHECK_WATCHDOG_MINIPORT_HALT 停止微型端口适配器时出现超时。 • 0x23 : NDIS_BUGCHECK_WATCHDOG_MINIPORT_OID 向微型端口适配器传送 OID 请求时超时。 • 0x24 : NDIS_BUGCHECK_WATCHDOG_FILTER_OID 向筛选器驱动程序传送 OID 请求时出现超时。 • 0x25 : NDIS_BUGCHECK_WATCHDOG_MINIPORT_IDLE 闲置微型端口适配器时出现超时。 • 0x26 : NDIS_BUGCHECK_WATCHDOG_CANCEL_IDLE 取消微型端口适配器上的空闲请求时出现超时。	
0x26	NDIS_BUGCHECK_INVALID_OID_COMPLETION 微型端口驱动程序试图完成当前未在该微型端口驱动程序上挂起的 OID 请求。 这可能是由于驱动程序尝试多次完成同一请求导致的。	导致 bugcheck 的微型端口驱动程序句柄。 请使用此句柄运行 !ndiskd.minidriver，获取详细信息。	
0x27	NDIS_BUGCHECK_LEAKED_NBL 驱动程序泄露了 NET_BUFFER_LIST 结构。 请通过 !ndiskd.pendingnbls 查看此驱动程序上仍在挂起的任何 NBL。	检测到泄漏的位置。 可能的值： • 0x01：NBL 跟踪器检测到泄漏。 当前正在取消注册或取消绑定的驱动程序是最有可能的原因。 查看 bugchecking 线程的调用堆栈。 驱动程序仍持有活动 NBL 时，不得取消绑定或取消注册。	

原因

参数 1 表示 BUGCODE_NDIS_DRIVER Bug 检查的具体原因。

注解

BUGCODE_NDIS_DRIVER Bug 检查表示网络驱动程序中的问题。 此问题通常是由 NDIS 微型端口驱动程序造成的。 可以使用 !ndiskd.netadapter获取 NDIS 微型端口驱动程序的完整列表。 可以使用 !ndiskd.netreport 查看网络堆栈的全貌。

此Bug 检查代码仅出现在 Microsoft Windows Server 2003 及更高版本的 Windows 系统中。 在 Windows 2000 和 Windows XP 中，相应的代码是 Bug 检查 0xD2 (BUGCODE_ID_DRIVER)。

Bug 检查 0x7D：INSTALL_MORE_MEMORY

项目 • 2024/02/21

INSTALL_MORE_MEMORY Bug 检查的值为 0x0000007D。 此 Bug 检查表明没有足够的内存来启动 Microsoft Windows 操作系统。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INSTALL_MORE_MEMORY 参数

展开表

参数	说明
1	找到的物理页数
2	最低物理页
3	最高物理页
4	0

原因

Windows 操作系统内存不足，无法完成启动过程。

解决方法

安装更多内存。

错误检查 0x7E：SYSTEM_THREAD_EXCEPTION_NOT_HANDLED

项目 • 2024/10/16

SYSTEM_THREAD_EXCEPTION_NOT_HANDLED 错误检查的值为 0x0000007E。 此错误检查表明系统线程产生了错误处理程序未捕获的异常。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SYSTEM_THREAD_EXCEPTION_NOT_HANDLED 参数

展开表

参数	说明
1	未处理的异常代码。
2	发生异常的地址。
3	异常记录的地址。
4	上下文记录的地址。

原因

此错误检查表明系统线程产生了错误处理程序未捕获的异常。 若要对其进行解释，必须确定生成的具体异常。

常见的异常代码如下：

- 0x80000002：STATUS_DATATYPE_MISALIGNMENT 表示遇到未对齐的数据引用。
- 0x80000003：STATUS_BREAKPOINT 表示在未将内核调试器附加到系统时遇到断点或 ASSERT。

- 0xC0000005：STATUS_ACCESS_VIOLATION 表示发生了内存访问冲突。

有关异常代码的完整列表，请参阅 NTSTATUS 值。 异常代码在 ntstatus.h（Windows 驱动程序工具包提供的标头文件）中定义。 有关详细信息，请参阅 Windows 驱动程序工具包中的头文件。

解决方法

如果计划调试此问题，异常地址（参数 2）应标识导致此问题的驱动程序或函数。

如果错误检查消息中按名称列出了某个驱动程序，请禁用或删除该驱动程序。 如果问题缩小到单个驱动程序，请在代码中设置断点并单步前进，以定位故障并深入了解导致崩溃的事件。

!analyze 调试程序扩展显示有关错误检查的信息，有助于确定根本原因。

可以使用 !thread 扩展以及 dds、dps 和 dqs（显示字词和符号） 命令完成更多分析。 当 WinDbg 报告"可能由：ntkrnlmp.exe 引起"时，此技术是合理的。

如果出现异常代码 0x80000003，则表示命中了硬编码断点或断言，但系统是使用 **/NODEBUG** 开关启动的。 此问题应该不会经常发生。 如果重复发生，请确保内核调试器已连接，并且已使用 **/DEBUG** 开关启动系统。

如果出现异常代码 0x80000002，陷阱帧会提供更多信息。

有关 WinDbg 和 **!analyze** 的详细信息，请参阅：

- 使用 WinDbg 分析故障转储文件
- 使用 WinDbg 分析内核模式转储文件
- 使用 !analyze 扩展 和 !analyze

注解

如果您不具备使用 Windows 调试器来解决此问题的能力，则应使用一些基本的故障排除技术：

- 在事件查看器中检查系统日志以获取更多错误消息，这可能有助于识别导致错误检查 0x7E 的设备或驱动程序。
- 如果在错误检查消息中标识了驱动程序，请禁用该驱动程序，或与制造商联系以获取驱动程序更新。

- 请咨询硬件供应商，了解是否有任何 ACPI 或其他固件更新。 硬件问题（如系统不兼容、内存冲突和 IRQ 冲突）也可能导致此错误。
- 禁用 BIOS 的内存缓存/重影，尝试解决错误。 还可以运行系统制造商提供的硬件诊断程序。
- 确保安装的任何新硬件都与已安装的 Windows 版本兼容。 例如，可以在 Windows 10 规范 中获取所需硬件的相关信息。

有关一般故障排除详细信息，请参阅分析错误检查蓝屏数据。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Bug 检查 0x7F：UNEXPECTED_KERNEL_MODE_TRAP

项目 • 2024/02/20

UNEXPECTED_KERNEL_MODE_TRAP Bug 检查的值为 0x0000007F。 此 Bug 检查表明 Intel CPU 生成了陷阱，而内核未能捕获此陷阱。

此陷阱可以是以下任何一种：

- *绑定陷阱*，这是不允许内核捕获的陷阱。
- *双重故障*，这是处理较早的错误时发生的错误，它总是会导致系统故障。

> **ⓘ 重要**
>
> 本主题是面向程序员的。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

参数 1

蓝屏上出现的第一个参数指定了陷阱编号。

最常见的陷阱编号是以下代码：

展开表

参数	陷阱代码	说明
0x00000000	除以零误差	表示执行 DIV 指令并且除数为零。 内存损坏、其他硬件问题或软件故障都可能会导致此错误。
0x00000004	溢出	当处理器在设置溢出 (OF) 标志时执行对中断处理程序的调用时发生。
0x00000005	边界检查错误	表示处理器在执行 BOUND 指令时发现操作数超出了指定的限制。 BOUND 指令确保有符号数组索引在特定范围内。
0x00000006	操作码无效	表明处理器尝试执行无效指令。 当指令指针损坏并指向错误的位置时，通常会发生此错误。 此错误的最常见原因是硬件内存损坏。
0x00000008	双故障	表明在调用先前异常的处理程序时发生异常。 通常，这两个异常是串行处理的。 有多个异常无法串行处理，因此处理器发出双重故障信号。

出现双重故障的常见原因有两个：

- 第一个原因是内核堆栈溢出。 当防护页被命中并且内核尝试推送陷阱帧时，会发生此溢出。 由于没有剩余堆栈，因此堆栈溢出会导致双重故障。 如果认为这种情况已发生，请使用 !thread 扩展来确定堆栈限制，然后使用具有较大值（例如 kb 100）的 kb（显示堆栈回溯） 命令来显示完整堆栈。
- 第二个常见原因是硬件问题。

不常见的陷阱代码包括以下值：

- 0x00000001：系统调试器调用 (DEBUG)
- 0x00000003：调试器断点 (INT3)
- 0x00000007：没有协处理器存在的硬件协处理器指令 (NXP_NOT_AVAILABLE)
- 0x0000000A：已损坏的任务状态段 (INVALID_TSS)
- 0x0000000B：访问不存在的内存段 (SEGMENT_NOT_PRESENT)
- 0x0000000C：超出堆栈限制的内存访问 (STACK_FAULT)
- 0x0000000D：其他异常未涵盖的异常，与应用程序访问冲突相关的保护错误 (GP_FAULT)
- 0x0000000F：保留陷阱异常 (RESERVED_TRAP)
- 0x00000010：硬件协处理器异常 (NPX_ERROR))
- 0x00000011：对齐检查异常 (ALIGNMENT_CHECK)

有关其他陷阱编号，请参阅要排除故障的处理器的 Intel 处理器体系结构手册。

原因

Bug 检查 0x7F 通常在安装了故障或不匹配的硬件（尤其是内存）或安装的硬件出现故障后发生。

当内核堆栈溢出时，可能会出现双重故障。 如果多个驱动程序附加到同一堆栈，则会发生此溢出。 例如，如果两个文件系统筛选器驱动程序附加到同一堆栈，然后文件系统递归回来，堆栈就会溢出。

调试

始终以 !analyze扩展开始，并使用 -v 选项，即 verbose。 检查输出和故障代码。 查找多个转储中重复出现的趋势。

尝试 **!analyze** 后，请使用 kv （显示堆栈回溯） 调试器命令。

- 如果 **kv** 显示任务门，请在冒号前使用 .tss （显示任务状态段）命令。
- 如果 **kv** 显示陷阱帧，请使用 .trap（显示陷阱帧）命令设置帧的格式。
- 否则，在相应的帧上使用 .trap（显示陷阱帧）命令。 在基于 x86 的平台上，此帧与过程 **NT!KiTrap** 相关联。

使用其中一个命令后，再次使用 **kv** 来显示新堆栈。

疑难解答

硬件

如果您最近为计算机添加了硬件，请将其删除后查看错误是否再次出现。 如果现有硬件出现故障，请卸下或更换故障部件。 运行系统制造商提供的硬件诊断程序，以确定哪个硬件组件出现故障。

内存故障或不匹配可能会导致此 Bug 检查。 使用 Windows 中的内存诊断程序测试所有系统内存。

确保所有硬盘驱动器和硬盘驱动器控制器都与已安装的 Windows 版本兼容。

系统主板可能存在问题，例如组件有划痕或缺陷。 电源故障也可能导致问题。

超频 是将 CPU 设置为以高于额定规格的速度运行，这可能会导致此错误。 如果已对遇到错误的计算机进行超频，请将 CPU 恢复到默认时钟速度设置。 如果此选项可用，请禁用 BIOS 的内存缓存，以尝试解决问题。

软件

在事件查看器中检查**系统日志**以获取其他错误消息，这可能有助于识别导致错误的设备或驱动程序。

检查硬件制造商提供的 ACPI/BIOS、硬盘驱动程序控制器或网卡更新的可用性。

如果安装新的或更新的设备驱动程序后出错，请删除或更换驱动程序。 在这种情况下，如果在启动序列时发生错误，请使用安全模式重命名或删除错误的驱动程序。 如果在安全模式下将驱动程序用作系统启动过程的一部分，请使用恢复控制台启动计算机以访问文件。 尝试使用*"最近一次的正确配置"*选项。 一次只添加一个驱动程序或服务时，此选项最有效。

如果在升级到新版本的 Windows 操作系统时遇到此错误，则此错误可能是由于软件不兼容造成的。 示例包括设备驱动程序、系统服务、病毒扫描程序或备份工具。 如果可能，请在升级之前删除所有第三方设备驱动程序和系统服务，并禁用所有病毒扫描程序。 请联系软件制造商获取这些工具的更新。 此外，请确保已安装最新的 Windows 更新。

另请参阅

- !analyze

- Bug 检查代码参考

Bug 检查 0x80：NMI_HARDWARE_FAILURE

项目 • 2024/02/20

NMI_HARDWARE_FAILURE Bug 检查的值为 0x00000080。 此 Bug 检查表明发生了硬件故障。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

NMI_HARDWARE_FAILURE 参数

无

原因

各种硬件故障可能会导致 NMI_HARDWARE_FAILURE Bug 检查。 确切原因难以确定。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。 删除最近安装的所有硬件或驱动程序。 确保所有内存模块的类型相同。

Bug 检查 0x81: SPIN_LOCK_INIT_FAILURE

项目 • 2024/02/20

SPIN_LOCK_INIT_FAILURE Bug 检查的值为 0x00000081。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x82：DFS_FILE_SYSTEM

项目 • 2024/02/23

DFS_FILE_SYSTEM Bug 检查的值为 0x00000082。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x85：SETUP_FAILURE

项目 • 2024/02/21

The SETUP_FAILURE Bug 检查的值为 0x00000085。 此 Bug 检查表明在安装过程中发生了致命错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SETUP_FAILURE 参数

参数 1 表示违规类型。 未使用参数 4。 其他参数的含义取决于参数 1 的值。

展开表

参数 1	参数 2	参数 3	原因
0x0	0	0	OEM HAL 字体不是有效的 .fon 格式文件，因此安装程序无法显示文本。 此原因表明启动软盘或 CD 上的 Vga*xxx*.fon 已损坏。
0x1	精确的视频初始化失败： \device\video0 的 **0: NtCreateFile** **1:** IOCTL_VIDEO_QUERY_NUM_AVAIL_MODES **2:** IOCTL_VIDEO_QUERY_AVAIL_MODES **3：** 所需的视频模式不受支持。 此值表明内部安装错误。 **4：** IOCTL_VIDEO_SET_CURRENT_MODE（无法设置视频模式） **5:** IOCTL_VIDEO_MAP_VIDEO_MEMORY	NT API 调用的状态代码（如果适用）	视频初始化失败。 此故障可能表明包含 Vga.sys（或适用于计算机的其他视频驱动程序）的磁盘已损坏，或者计算机的视频硬件无法与 Microsoft Windows 操作系统通信。

参数 1	参数 2	参数 3	原因
	6: IOCTL_VIDEO_LOAD_AND_SET_FONT		
0x2	0	0	内存不足。
0x3	精确的键盘初始化失败： \device\KeyboardClass0 的 **0:NtCreateFile** 失败。 （安装程序未找到连接到计算机的键盘。） **1:** 无法加载键盘布局 DLL。 （安装程序无法加载键盘布局文件。此故障表明 CD 或软盘缺少文件，例如美国版本的 Kbdus.dll 或本地化版本的其他布局 DLL。）	0	键盘初始化失败。 此故障可能表明包含键盘驱动程序（I8042prt.sys 或 Kbdclass.sys）的磁盘已损坏，或者计算机的键盘硬件无法与 Windows 通信。 此故障也可能意味着无法加载键盘布局 DLL。
0x4	0	0	安装程序无法解析从其启动的设备的 ARC 设备路径名称。 此错误是内部安装错误。
0x5	预留	预留	分区健全性检查失败。 此错误表明磁盘驱动程序中存在 Bug。

Bug 检查 0x8B：MBR_CHECKSUM_MISMATCH

项目 • 2024/02/20

MBR_CHECKSUM_MISMATCH Bug 检查的值为 0x0000008B。 此 Bug 检查表明 MBR 校验和中出现了不匹配。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

MBR_CHECKSUM_MISMATCH 参数

展开表

参数	说明
1	来自 MBR 的磁盘签名
2	操作系统加载程序计算的 MBR 校验和
3	系统计算的 MBR 校验和
4	Reserved

原因

当 Microsoft Windows 操作系统计算的 MBR 校验和与加载程序传入的校验和不匹配时，启动过程中将发生 MBR_CHECKSUM_MISMATCH Bug 检查。

此错误通常表示存在病毒。

解决方法

病毒有许多形式，并非所有病毒都可以检测到。 通常，只有最近升级过的病毒扫描程序才能检测到较新的病毒。 应使用包含病毒扫描程序的受写保护磁盘启动，并尝试清除感染。

Bug 检查 0x8E：KERNEL_MODE_EXCEPTION_NOT_HANDLED

项目 • 2024/02/20

KERNEL_MODE_EXCEPTION_NOT_HANDLED Bug 检查的值为 0x0000008E。 此 Bug 检查表明内核模式应用程序产生了错误处理程序未捕获到的异常。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

KERNEL_MODE_EXCEPTION_NOT_HANDLED 参数

展开表

参数	说明
1	未处理的异常代码
2	发生异常的地址
3	陷阱帧
4	Reserved

原因

KERNEL_MODE_EXCEPTION_NOT_HANDLED Bug 检查是一种非常常见的 Bug 检查。 若要对其进行解释，必须确定生成的具体异常。

常见的异常代码如下：

- 0x80000002：STATUS_DATATYPE_MISALIGNMENT 表示遇到未对齐的数据引用。
- 0x80000003：STATUS_BREAKPOINT 表示在未将内核调试器附加到系统时遇到断点或 ASSERT。

- 0xC0000005：STATUS_ACCESS_VIOLATION 表示发生了内存访问冲突。

有关异常代码的完整列表，请参阅 Microsoft Windows 驱动程序工具包 (WDK) inc 目录中的 Ntstatus.h 文件。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。 如果您不具备调试此问题的能力，则应使用一些基本的故障排除技术：

- 确保有足够的磁盘空间。
- 如果在 bug 检查消息中标识了驱动程序，请禁用该驱动程序，或与制造商联系以获取驱动程序更新。
- 请尝试更改视频适配器。
- 请与硬件供应商联系以获取 BIOS 更新。
- 禁用 BIOS 内存选项，例如缓存或阴影。

如果您计划调试此问题，可能会发现很难获取堆栈跟踪。 参数 2（异常地址）应标识导致此问题的驱动程序或函数。

如果出现异常代码 0x80000003，则表示命中了硬编码断点或断言，但电脑是使用 **/NODEBUG** 开关启动的。 此问题很少发生。 如果重复发生，请确保内核调试器已连接，并且已使用 **/DEBUG** 开关启动电脑。

如果出现异常代码 0x80000002，陷阱帧会提供更多信息。

如果不知道异常的具体原因，请考虑以下各项：

- 硬件不兼容 确保安装的任何新硬件都与已安装的 Windows 版本兼容。
- 设备驱动程序或系统服务存在故障。 设备驱动程序或系统服务故障可能是导致此错误的原因。 硬件问题（如 BIOS 不兼容、内存冲突和 IRQ 冲突）也可能会产生此错误。

如果 Bug 检查消息按名称列出了某个驱动程序，请禁用或删除该驱动程序。 此外，禁用或删除最近添加的任何驱动程序或服务。 如果在启动过程中发生错误，并且系统分区使用 NTFS 文件系统进行了格式化，则有可能可以使用安全模式重命名或删除有故障的驱动程序。 如果在安全模式下将驱动程序用作系统启动过程的一部分，则必须使用恢复控制台启动计算机以访问文件。

如果问题与 Win32k.sys 有关，错误来源可能是第三方远程控制程序。 如果安装了此类软件，可以使用恢复控制台启动系统，然后删除有问题的系统服务文件，从而删除该服务。

在事件查看器中检查系统日志以获取其他错误消息，这可能有助于识别导致 bug 检查 0x8E 的设备或驱动程序。 可以禁用 BIOS 的内存缓存来尝试解决错误。 您还应该运行硬件诊断程序，尤其是系统制造商提供的内存扫描程序。 有关这些过程的详细信息，请参阅计算机的用户手册。

出现此消息的错误可能发生在 Windows 安装过程中首次重启后或安装完成后。 造成错误的可能原因是安装磁盘空间不足和系统 BIOS 不兼容。 对于 Windows 安装过程中与磁盘空间不足相关的问题，请减少目标硬盘驱动器上的文件数。 检查并删除任何不需要的临时文件、Internet 缓存文件、应用程序备份文件和包含磁盘扫描中保存的文件片段的 .chk 文件。 还可以使用可用空间更大的其他硬盘驱动器进行安装。

Bug 检查 0x8F：PP0_INITIALIZATION_FAILED

项目 • 2024/02/20

PP0_INITIALIZATION_FAILED Bug 检查的值为 0x0000008F。 此 Bug 检查表明无法初始化即插即用 (PnP) 管理器。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PP0_INITIALIZATION_FAILED 参数

无

原因

在内核模式 PnP 管理器的阶段 0 初始化期间发生错误。

Bug 检查 0x90：PP1_INITIALIZATION_FAILED

项目 • 2024/02/21

PP1_INITIALIZATION_FAILED Bug 检查的值为 0x00000090。 此 Bug 检查表明无法初始化即插即用 (PnP) 管理器。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PP1_INITIALIZATION_FAILED 参数

无

原因

在内核模式 PnP 管理器的阶段 1 初始化期间发生错误。

阶段 1 完成了大部分初始化工作，包括设置注册表文件和其他环境设置，以供驱动程序在后续 I/O 初始化期间调用。

Bug 检查 0x92：UP_DRIVER_ON_MP_SYSTEM

项目 • 2024/02/20

UP_DRIVER_ON_MP_SYSTEM Bug 检查的值为 0x00000092。 此 Bug 检查表明在多处理器系统上加载了仅单处理器驱动程序。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

UP_DRIVER_ON_MP_SYSTEM 参数

展开表

参数	说明
1	驱动程序的基址
2	Reserved
3	Reserved
4	Reserved

原因

已加载编译为仅在单处理器计算机上工作的驱动程序，但 Microsoft Windows 操作系统在具有多个活动处理器的多处理器系统上运行。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

Bug 检查 0x93：INVALID_KERNEL_HANDLE

项目 • 2024/02/20

INVALID_KERNEL_HANDLE Bug 检查的值为 0x00000093。 此 Bug 检查表明无效或受保护的句柄已传递给 **NtClose**。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INVALID_KERNEL_HANDLE 参数

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
调用 NtClose 时使用的句柄	0	0	0	受保护的句柄已关闭。
调用 NtClose 时使用的句柄	1	0	0	关闭或引用了无效句柄。
引用的句柄	句柄表	0	1	引用无效的内核句柄时发生错误，并启用了错误的句柄检测。

原因

INVALID_KERNEL_HANDLE Bug 检查表明某些内核代码（例如服务器、重定向程序或其他驱动程序）尝试关闭无效的句柄或受保护的句柄。

如果参数 4 的值为 1，则表示引用无效的内核句柄时发生错误，并启用了错误的句柄检测。

如果内核代码试图关闭或引用不是有效句柄的句柄，则会出现此消息。 只有传递给 NtClose 的无效或受保护句柄才会导致此 Bug检查，除非启用了错误的句柄检测。

Bug 检查 0x94：KERNEL_STACK_LOCKED_AT_EXIT

项目 • 2024/02/20

KERNEL_STACK_LOCKED_AT_EXIT Bug 检查的值为 0x00000094。 此 Bug 检查表明线程在内核堆栈标记为不可交换时退出

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

KERNEL_STACK_LOCKED_AT_EXIT 参数

无

Bug 检查 0x96：INVALID_WORK_QUEUE_ITEM

项目 • 2024/02/20

INVALID_WORK_QUEUE_ITEM Bug 检查的值为 0x00000096。 此 Bug 检查表明已删除包含 NULL 指针的队列条目。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INVALID_WORK_QUEUE_ITEM 参数

展开表

参数	说明
1	flink 或 blink 字段为 NULL 的队列条目的地址。
2	被引用的队列地址。 通常，此队列是 ExWorkerQueue。
3	ExWorkerQueue 数组的基址。 （此地址可帮助您确定相关队列是否确实是 ExWorkerQueue。 如果队列是 ExWorkerQueue，则此参数的偏移量将隔离队列。）
4	假设队列是 ExWorkerQueue，则此值是工作项有效时会被调用的工作例程的地址。 （您可以使用此地址来隔离滥用工作队列的驱动程序。）

原因

当 KeRemoveQueu 删除flink 或 blink 字段为 NULL 的队列条目时，会出现 INVALID_WORK_QUEUE_ITEM Bug 检查。

任何队列滥用都可能导致此错误。 但发生此错误的通常原因是工作线程工作项被滥用。

队列上的条目只能在列表中插入一次。 从一个项目队列中删除时，其 flink 字段被设置为 NULL。 然后，当第二次删除此项目时，会发生此 Bug 检查。

在大多数情况下，被引用的队列是 **ExWorkerQueue**（执行辅助角色队列）。 为了有助于识别导致错误的驱动程序，参数 4 显示工作线程例程的地址，如果此工作项有效，则会调用该例程。 但是，如果所引用的队列不是 **ExWorkerQueue**，则此参数没有用处。

Bug 检查 0x97：BOUND_IMAGE_UNSUPPORTED

项目 • 2024/02/20

BOUND_IMAGE_UNSUPPORTED Bug 检查的值为 0x00000097。

此 Bug 检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x98：END_OF_NT_EVALUATION_PERIOD

项目 • 2024/02/20

END_OF_NT_EVALUATION_PERIOD Bug 检查的值为 0x00000098。 此 Bug 检查表明 Microsoft Windows 操作系统的试用期已结束。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

END_OF_NT_EVALUATION_PERIOD 参数

展开表

参数	说明
1	产品到期日期的低位 32 位
2	产品到期日期的高序 32 位
3	Reserved
4	Reserved

原因

您安装的 Windows 操作系统是具有到期日期的评估单元。 试用期结束。

Bug 检查 0x99：INVALID_REGION_OR_SEGMENT

项目 • 2024/02/21

The INVALID_REGION_OR_SEGMENT Bug 检查的值为 0x00000099。 此 Bug 检查表明使用无效的参数集调用了 **ExInitializeRegion** 或 **ExInterlockedExtendRegion**。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INVALID_REGION_OR_SEGMENT 参数

无

Bug 检查 0x9A：SYSTEM_LICENSE_VIOLATION

项目 • 2024/02/21

The SYSTEM_LICENSE_VIOLATION Bug 检查的值为 0x0000009A。 此 Bug 检查表明违反了软件许可协议。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SYSTEM_LICENSE_VIOLATION 参数

参数 1 表示违规类型。 其他参数的含义取决于参数 1 的值。

展开表

参数 1	参数 2	参数 3	参数 4	原因
0x00	**0**：产品应为 WinNT **1**：产品应为 LanmanNT 或 ServerNT	部分序列号	产品选项中产品类型的前两个字符	已尝试离线更改产品类型。
0x01	来源 1 的注册评估时间	部分序列号	备用源的注册评估时间	已尝试对 Microsoft Windows 评估单位时间段进行离线更改。
0x02	与打开失败相关的状态代码	0	0	无法打开安装密钥。
0x03	与密钥查找失败相关的状态代码	0	0	安装密钥中的 SetupType 或 SetupInProgress 值缺失，因此无法检测到安装模式。

参数 1	参数 2	参数 3	参数 4	原因
0x04	与密钥查找失败相关的状态代码	0	0	安装程序密钥中的 SystemPrefix 值缺失。
0x05	（请参阅安装代码）	在许可处理器中发现无效值	官方许可的处理器数量	已尝试对许可处理器数量进行离线更改。
0x06	与打开失败相关的状态代码	0	0	ProductOptions 密钥无法打开。
0x07	与读取失败相关的状态代码	0	0	ProductType 值无法读取。
0x08	与更改通知失败相关的状态代码	0	0	ProductOptions 的更改通知失败。
0x09	与更改通知失败相关的状态代码	0	0	SystemPrefix 的更改通知失败。
0x0A	0	0	0	NTW 系统已转换为 NTS 系统。
0x0B	与更改失败相关的状态代码	0	0	安装密钥的引用失败。
0x0C	与更改失败相关的状态代码	0	0	产品选项密钥的引用失败。
0x0D	与失败相关的状态代码	0	0	尝试在工作线程中打开 ProductOptions 失败。
0x0F	与失败相关的状态代码	0	0	尝试打开安装密钥失败。
0x10	与失败相关的状态代码	**0**：设置值失败 **1**：更改通知失败	0	安装密钥工作线程中发生故障。
0x11	与失败相关的状态代码	**0**：设置值失败 **1**：更改通知失败	0	产品选项密钥工作线程中发生故障。
0x12	与失败相关的状态代码	0	0	无法打开套件的 LicenseInfoSuites 密钥。

参数 1	参数 2	参数 3	参数 4	原因
0x13	与失败相关的状态代码	0	0	无法查询套件的 **LicenseInfoSuites** 密钥。
0x14	内存分配的大小	0	0	无法分配内存。
0x15	与失败相关的状态代码	Reserved	0	无法重置套件密钥的 **ConcurrentLimit** 值。
0x16	与失败相关的状态代码	0	0	无法打开套件产品的许可证密钥。
0x17	与失败相关的状态代码	0	0	无法重置 **套件产品的 ConcurrentLimit** 值。
0x18	与打开失败相关的状态代码	Reserved	0	无法启动 **LicenseInfoSuites** 的更改通知。
0x19	0	0	0	套件在必须是 PDC 的系统上运行。
0x1A	与失败相关的状态代码	0	0	枚举套件时发生故障。
0x1B	0	0	0	尝试更改策略缓存。

原因

Microsoft Windows 操作系统检测到违反软件许可协议的行为。

用户可能试图更改脱机系统的产品类型或更改 Windows 评估单元的试用期。 有关具体冲突的更多信息，请参阅参数列表。

Bug 检查：0x9B UDFS_FILE_SYSTEM

项目 • 2024/02/20

UDFS_FILE_SYSTEM Bug 检查的值为 0x0000009B。 此 Bug 检查表明 UDF 文件系统中出现了问题。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

UDFS_FILE_SYSTEM 参数

展开表

参数	说明
1	源文件和行号信息。 高 16 位（“0x”后面的前四个十六进制数字）按标识符号标识源文件。 低 16 位标识文件中出现 Bug 检查的源行。
2	如果 **UdfExceptionFilter** 位于堆栈上，则此参数指定异常记录的地址。
3	如果 **UdfExceptionFilter** 位于堆栈上，则此参数指定上下文记录的地址。
4	保留。

原因

UDFS_FILE_SYSTEM Bug 检查可能会导致磁盘损坏。 磁盘上的文件系统损坏或坏块（扇区）可能会引发此错误。 损坏的 SCSI 和 IDE 驱动程序也会对系统读写磁盘的能力产生不利影响，并导致错误。

如果非分页池内存已满，也可能出现此 Bug 检查。 如果非分页池内存已满，此错误可能会导致系统停止运行。 但是，在索引过程中，如果可用非分页池内存量非常低，则另一个需要非分页池内存的内核模式驱动程序也可能会触发此错误。

解决方法

若要调试此问题：请使用参数 3 的 **.cxr（显示上下文记录）**命令，然后使用 **kb（显示堆栈回溯跟踪）。**。

若要解决磁盘损坏问题：请检查事件查看器查看来自 SCSI 和 FASTFAT（系统日志）或 Autochk（应用程序日志）的错误消息，这些错误消息可能有助于识别导致错误的设备或驱动程序。 禁用任何持续监视系统的病毒扫描程序、备份应用程序或磁盘碎片整理程序工具。 还应该运行系统制造商提供的硬件诊断程序。 有关这些过程的详细信息，请参阅计算机的用户手册。 运行 **Chkdsk /f /r**，检测并解决任何文件系统结构损坏问题。 在系统分区上开始磁盘扫描之前，必须重启系统。

若要解决非分页池内存耗尽问题： 向计算机添加新的物理内存。 此内存会增加内核可用的非分页池内存数量。

Bug 检查 0x9C：MACHINE_CHECK_EXCEPTION

项目 • 2024/02/20

MACHINE_CHECK_EXCEPTION Bug 检查的值为 0x0000009C。 此 Bug 检查表明发生了致命的计算机检查异常。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

MACHINE_CHECK_EXCEPTION 参数

消息中列出的四个参数具有不同的含义，具体取决于处理器类型。

如果处理器基于较早的 x86 体系结构，并且具有计算机检查异常 (MCE) 功能，但不具有计算机检查体系结构 (MCA) 功能（例如 Intel Processorum 处理器），则参数的含义如下。

展开表

参数	说明
1	P5_MC_TYPE 机器服务报告 (MSR) 的低 32 位
2	MCA_EXCEPTION 结构的地址
3	P5_MC_ADDR MSR 的高 32 位
4	P5_MC_ADDR MSR 的低 32 位

如果处理器基于基于较新的 x86 体系结构，并且具有 MCA 功能和 MCE 功能（例如，系列 6 或更高版本的任何 Intel 处理器，如 Processorum Pro、Pentum IV 或 Xeon），或者如果处理器是基于 x64 的处理器，则参数的含义如下。

展开表

参数	说明
1	银行编号

参数	说明
2	MCA_EXCEPTION 结构的地址
3	出现错误的 MCA 寄存器的 MCi_STATUS MSR 高 32 位
4	出现错误的 MCA 寄存器的 MCi_STATUS MSR 低 32 位

注解

此 Bug 检查仅在以下情况下发生。

- WHEA 尚未完全初始化。
- 会合的所有处理器在其寄存器中都没有错误。

对于其他情况，在 Windows Vista 和更高版本的操作系统中，此 Bug 检查已替换为 **Bug 检查 0x124：WHEA_UNCORRECTABLE_ERROR**。

有关计算机检查体系结构 (MCA) 的详细信息，请参阅 Intel 或 AMD 网站。

Bug 检查 0x9E: USER_MODE_HEALTH_MONITOR

项目 • 2024/02/20

The USER_MODE_HEALTH_MONITOR Bug 检查的值为 0x0000009E。 此 Bug 检查表明一个或多个关键用户模式组件未能满足运行状况检查。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

USER_MODE_HEALTH_MONITOR 参数

展开表

参数	说明
1	在配置的超时时间中未能满足运行状况检查的进程
2	运行状况监视超时，以秒为单位
3	监视程序源。 结合进程地址有助于识别已创建此监视器的子组件。 其值如下所列。
4	Reserved

VALUES

```
0  : WatchdogSourceDefault
       Source was not specified
1  : WatchdogSourceRhsCleanup
       Monitors that RHS (Resource Hosting Subsystem) process goes away when
       terminating on graceful exit
2  : WatchdogSourceRhsResourceDeadlockBugcheckNow
       RHS was asked to immediately bugcheck machine
       on resource deadlock
3  : WatchdogSourceRhsExceptionFromResource
       Resource has leaked unhandled exception from an entry point,
       RHS is terminating and this watchdog monitors that
       process will go away
4  : WatchdogSourceRhsUnhandledException
       Unhandled exception in RHS.
```

```
      RHS is terminating and this watchdog monitors that
      process will go away
5  : WatchdogSourceRhsResourceDeadlock
      Monitors that RHS process goes away when
      terminating on resource deadlock
6  : WatchdogSourceRhsResourceTypeDeadlock
      Monitors that RHS process goes away when
      terminating on resource type deadlock
7  : WatchdogSourceClussvcUnhandledException
      Unhandled exception in clussvc.
      clussvc is terminating and this watchdog monitors that
      process will go away
8  : WatchdogSourceClussvcBugcheckMessageRecieved
      Another cluster node has sent message asking to bugcheck this node.
9  : WatchdogSourceClussvcWatchdogBugcheck
      User mode watchdog has expired and created netft watchdog
      to bugchecked the node.
       0xA : WatchdogSourceClussvcIsAlive
      Cluster service sends heartbeat to netft every 500 millseconds.
      By default, netft expects at least 1 heartbeat per second.
      If this watchdog was triggered that means clussvc is not getting
      CPU to send heartbeats.
      0x65 : WatchdogSourceRhsResourceDeadlockPhysicalDisk
       A subclass of WatchdogSourceRhsResourceDeadlock.
      0x66 : WatchdogSourceRhsResourceDeadlockStoragePool
       A subclass of WatchdogSourceRhsResourceDeadlock.
      0x67 : WatchdogSourceRhsResourceDeadlockFileServer
       A subclass of WatchdogSourceRhsResourceDeadlock.
      0x68 : WatchdogSourceRhsResourceDeadlockSODAFileServer
       A subclass of WatchdogSourceRhsResourceDeadlock.
      0x69 : WatchdogSourceRhsResourceDeadlockStorageReplica
       A subclass of WatchdogSourceRhsResourceDeadlock.
      0x6A : WatchdogSourceRhsResourceDeadlockStorageQOS
       A subclass of WatchdogSourceRhsResourceDeadlock.
      0x6B : WatchdogSourceRhsResourceDeadlockStorageNFSV2
       A subclass of WatchdogSourceRhsResourceDeadlock.
      0xC9 : WatchdogSourceRhsResourceTypeDeadlockPhysicalDisk
       A subclass of WatchdogSourceRhsResourceTypeDeadlock.
      0xCA : WatchdogSourceRhsResourceTypeDeadlockStoragePool
       A subclass of WatchdogSourceRhsResourceTypeDeadlock.
      0xCB : WatchdogSourceRhsResourceTypeDeadlockFileServer
       A subclass of WatchdogSourceRhsResourceTypeDeadlock.
      0xCC : WatchdogSourceRhsResourceTypeDeadlockSODAFileServer
       A subclass of WatchdogSourceRhsResourceTypeDeadlock.
      0xCD : WatchdogSourceRhsResourceTypeDeadlockStorageReplica
       A subclass of WatchdogSourceRhsResourceTypeDeadlock.
      0xCE : WatchdogSourceRhsResourceTypeDeadlockStorageQOS
       A subclass of WatchdogSourceRhsResourceTypeDeadlock.
      0xCF : WatchdogSourceRhsResourceTypeDeadlockStorageNFSV2
       A subclass of WatchdogSourceRhsResourceTypeDeadlock.
```

原因

硬件机制（如监视器计时器）可以检测到基本内核服务未执行。 但是，资源不足问题（包括内存泄漏、锁争用和计划优先级配置错误）可能会阻止关键的用户模式组件，而不会阻止延迟的过程调用 (DPC) 或耗尽非分页池。

内核组件可以通过定期监视关键应用程序将监视器计时器功能扩展到用户模式。 此 Bug 检查表明用户模式运行状况检查失败，从而阻止了正常关闭。 此 Bug 检查通过重启或启用应用程序故障转移到其他服务器来恢复关键服务。

与所有 Bug 检查一样，使用系统事件日志及时查找停止代码之前的事件。 应检查日志中立即执行 Bug 检查的事件，了解可能的原因。

另请参阅

使用 Windows 错误报告排查故障转移群集问题

故障转移群集系统日志事件

Bug 检查 0x1C9 USER_MODE_HEALTH_MONITOR_LIVEDUMP

Bug 检查代码参考

Bug 检查 0x9F：DRIVER_POWER_STATE_FAILURE

项目 • 2024/02/20

DRIVER_POWER_STATE_FAILURE Bug 检查的值为 0x0000009F。 此 Bug 检查表明驱动程序处于不一致或无效的电源状态。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_POWER_STATE_FAILURE 参数

参数 1 表示违规类型。

展开表

参数 1	参数 2	参数 3	参数 4	原因
0x1	设备对象	预留	预留	正在释放的设备对象仍具有尚未完成的未完成电源请求。
0x2	目标设备的设备对象（如果可用）	设备对象	驱动程序对象（如果可用）	设备对象完成了系统电源状态请求的 I/O 请求数据包 (IRP)，但它未调用 PoStartNextPowerIrp。
0x3	堆栈的物理设备对象 (PDO)	nt!_TRIAGE_9F_POWER。	被阻塞的 IRP	设备对象阻塞 IRP 的时间过长。
0x4	超时值（以秒为单位）。	当前持有即插即用 (PnP) 锁的线程。	nt!TRIAGE_9F_PNP。	电源状态转换超时，等待与 PnP 子系统同步。
0x5	堆栈的物理设备对象	POP_FX_DEVICE 对象	已保留 - 0	设备未能在要求的时间内完成定向电源转换。
0x6	POP_FX_DEVICE 对象	表明这是定向电源关闭 (1) 或电源启动 (0) 完	已保留 - 0	设备未成功完成其定向电源转换回调。

参数 1	参数 2	参数 3	参数 4	原因
		成。		
0x500	Reserved	目标设备的设备对象（如果可用）	设备对象	设备对象完成了系统电源状态请求的 IRP，但它未调用 PoStartNextPowerIrp。

原因

有关可能原因的说明，请参阅"参数"部分对每个代码的描述。 常见原因包括：

- 设备对象释放 w/ 未完成的未完成电源请求
- 电源状态转换超时
- 阻塞 IRP 的设备对象
- 已完成 IRP 但未调用 PoStartNextPowerIrp

解决方法

若要确定具体原因并创建代码修复，需要编程经验和故障模块源代码的访问权限。

参数 1 等于 0x3 时调试 Bug 检查 0x9F

- 在内核调试器中，使用 **!analyze -v** 命令执行初始 Bug 检查分析。 详细分析显示 **nt!TRIAGE_9F_POWER** 结构的地址，该结构位于 Arg3 中。

dbgcmd

```
kd>!analyze -v
*******************************************************************************
*
*
*                        Bugcheck Analysis
*
*
*
*******************************************************************************

    DRIVER_POWER_STATE_FAILURE (9f)
    A driver has failed to complete a power IRP within a specific time.
    Arguments:
    Arg1: 0000000000000003, A device object has been blocking an Irp for too
long a time
    Arg2: fffffa8007b13440, Physical Device Object of the stack
```

```
    Arg3: fffff8000386c3d8, nt!_TRIAGE_9F_POWER on Win7 and higher,
otherwise the Functional Device Object of the stack
    Arg4: fffffa800ab61bd0, The blocked IRP
```

如果能够识别出导致错误的驱动程序，则会将其名称打印在蓝屏上，并存储在内存中的 (PUNICODESTRING) **KiBugCheckDriver** 位置。 可以使用一个调试器命令 dx（显示调试器对象模型表达式）来显示此内容：`dx KiBugCheckDriver`。

The nt!TRIAGE_9F_POWER 结构提供了额外的 Bug 检查信息，可以帮助您确定此 Bug 检查的原因。 该结构可以提供所有未完成的电源 IRP 的列表、所有电源 IRP 工作线程的列表，以及指向延迟系统辅助角色队列的指针。

- 使用 **dt（显示类型）**命令并使用 Arg3 中地址指定 nt!TRIAGE_9F_POWER 结构。

dbgcmd

```
    0: kd> dt nt!_TRIAGE_9F_POWER fffff8000386c3d8
       +0x000 Signature        : 0x8000
       +0x002 Revision         : 1
       +0x008 IrpList          : 0xfffff800`01c78bd0 _LIST_ENTRY [
0xfffffa80`09f43620 - 0xfffffa80`0ad00170 ]
       +0x010 ThreadList       : 0xfffff800`01c78520 _LIST_ENTRY [
0xfffff880`009cdb98 - 0xfffff880`181f2b98 ]
       +0x018 DelayedWorkQueue : 0xfffff800`01c6d2d8 _TRIAGE_EX_WORK_QUEUE
```

dt（显示类型）命令显示结构。 可以使用各种调试器命令跟踪 LIST_ENTRY 字段，检查未完成的 IRP 和电源 IRP 工作线程的列表。

- **使用 !irp** 命令检查被阻塞的 IRP。 此 IRP 的地址位于 Arg4 中。

dbgcmd

```
    0: kd> !irp fffffa800ab61bd0
    Irp is active with 7 stacks 6 is current (= 0xfffffa800ab61e08)
     No Mdl: No System Buffer: Thread 00000000:  Irp stack trace.
         cmd  flg cl Device   File     Completion-Context
     [N/A(0), N/A(0)]
                0  0 00000000 00000000 00000000-00000000

                Args: 00000000 00000000 00000000 00000000
     [N/A(0), N/A(0)]
                0  0 00000000 00000000 00000000-00000000

                Args: 00000000 00000000 00000000 00000000
     [N/A(0), N/A(0)]
                0  0 00000000 00000000 00000000-00000000

                Args: 00000000 00000000 00000000 00000000
     [N/A(0), N/A(0)]
```

```
                0  0 00000000 00000000 00000000-00000000

                Args: 00000000 00000000 00000000 00000000
     [N/A(0), N/A(0)]
                0  0 00000000 00000000 00000000-00000000

                Args: 00000000 00000000 00000000 00000000
    >[IRP_MJ_POWER(16), IRP_MN_SET_POWER(2)]
                0 e1 fffffa800783f060 00000000 00000000-00000000    pending
               \Driver\HidUsb
                Args: 00016600 00000001 00000004 00000006
     [N/A(0), N/A(0)]
                0  0 00000000 00000000 00000000-fffffa800ad00170

                Args: 00000000 00000000 00000000 00000000
```

- 使用 !devstack 命令和 Arg2 中的 PDO 地址，显示与故障驱动程序相关的信息。

dbgcmd

```
    0: kd> !devstack fffffa8007b13440
      !DevObj           !DrvObj            !DevExt           ObjectName
      fffffa800783f060  \Driver\HidUsb     fffffa800783f1b0  InfoMask field
not found for _OBJECT_HEADER at fffffa800783f030

    > fffffa8007b13440  \Driver\usbhub     fffffa8007b13590  Cannot read
info offset from nt!ObpInfoMaskToOffset

    !DevNode fffffa8007ac8a00 :
      DeviceInst is "USB\VID_04D8&PID_0033\5&46fa7b7&0&1"
      ServiceName is "HidUsb"
```

- 使用 !poaction 命令显示处理电源操作的线程和任何分配的电源 IRP。

dbgcmd

```
    3: kd> !poaction
    PopAction: fffff801332f3fe0
      State..........: 0 - Idle
      Updates........: 0
      Action.........: None
      Lightest State.: Unspecified
      Flags..........: 10000003 QueryApps|UIAllowed
      Irp minor......: ??
      System State...: Unspecified
      Hiber Context..: 0000000000000000

    Allocated power irps (PopIrpList - fffff801332f44f0)
      IRP: ffffe0001d53d8f0 (wait-wake/S0), PDO: ffffe00013cae060
      IRP: ffffe0001049a5d0 (wait-wake/S0), PDO: ffffe00012d42050
      IRP: ffffe00013d07420 (set/D3,), PDO: ffffe00012daf840, CURRENT:
ffffe00012dd5040
```

```
      IRP: ffffe0001e5ac5d0 (wait-wake/S0), PDO: ffffe00013d33060
      IRP: ffffe0001ed3e420 (wait-wake/S0), PDO: ffffe00013c96060
      IRP: ffffe000195fe010 (wait-wake/S0), PDO: ffffe00012d32050

    Irp worker threads (PopIrpThreadList - fffff801332f3100)
      THREAD: ffffe0000ef5d040 (static)
      THREAD: ffffe0000ef5e040 (static), IRP: ffffe00013d07420, DEVICE:
ffffe00012dd5040

    PopAction: fffff801332f3fe0
      State..........: 0 - Idle
      Updates........: 0
      Action.........: None
      Lightest State.: Unspecified
      Flags..........: 10000003 QueryApps|UIAllowed
      Irp minor......: ??
      System State...: Unspecified
      Hiber Context..: 0000000000000000

    Allocated power irps (PopIrpList - fffff801332f44f0)
      IRP: ffffe0001d53d8f0 (wait-wake/S0), PDO: ffffe00013cae060
      IRP: ffffe0001049a5d0 (wait-wake/S0), PDO: ffffe00012d42050
      IRP: ffffe00013d07420 (set/D3,), PDO: ffffe00012daf840, CURRENT:
ffffe00012dd5040
      IRP: ffffe0001e5ac5d0 (wait-wake/S0), PDO: ffffe00013d33060
      IRP: ffffe0001ed3e420 (wait-wake/S0), PDO: ffffe00013c96060
      IRP: ffffe000195fe010 (wait-wake/S0), PDO: ffffe00012d32050

    Irp worker threads (PopIrpThreadList - fffff801332f3100)
      THREAD: ffffe0000ef5d040 (static)
      THREAD: ffffe0000ef5e040 (static), IRP: ffffe00013d07420, DEVICE:
ffffe00012dd5040
```

- 如果使用的是 KMDF 驱动程序，请使用 Windows 驱动程序框架扩展 (!wdfkd) 收集更多信息。

  使用 **!wdfkd.wdflogdump**< 您的驱动程序名称>，查看 KMDF 是否在等待您 ACK 任何挂起的请求。

  使用 **!wdfkd.wdfdevicequeuess**< 您的 WDFDEVICE>，检查所有未完成请求及其当前状态。

- 使用 **!stacks** 扩展检查每个线程的状态，并查找可能保持电源状态转换的线程。

- 为了帮助您确定错误原因，请考虑以下问题：
  - 物理设备对象 (PDO) 驱动程序 (Arg2) 的特征是什么？
  - 能否找到受阻线程？ 使用 **!thread** 调试器命令检查线程时，线程包含哪些内容？
  - 是否存在与线程相关的 IO 阻止了它的运行？ 堆栈上有哪些符号？
  - 检查受阻电源 IRP 时，你会注意到什么？
  - 电源 IRP 的 PnP 次要函数代码是什么？

参数 1 等于 0x4 时调试 Bug 检查 0x9F

- 在内核调试器中，使用 **!analyze -v** 命令执行初始 Bug 检查分析。 详细分析显示 **nt!TRIAGE_9F_PN** 结构的地址，该结构位于参数 4 (arg4) 中。

dbgcmd

```
    kd> !analyze -v

*******************************************************************************
***
    *
*
    *                        Bugcheck Analysis
*
    *
*

*******************************************************************************
***

    DRIVER_POWER_STATE_FAILURE (9f)
    A driver has failed to complete a power IRP within a specific time
(usually 10 minutes).
    Arguments:
    Arg1: 00000004, The power transition timed out waiting to synchronize
with the Pnp
             subsystem.
    Arg2: 00000258, Timeout in seconds.
    Arg3: 84e01a70, The thread currently holding on to the Pnp lock.
    Arg4: 82931b24, nt!TRIAGE_9F_PNP on Win7
```

nt!TRIAGE_9F_PNP 结构提供了额外的 Bug 检查信息，可帮助您确定错误的原因。 nt!TRIAGE_9F_PNP 结构提供指向包含已调度（但未完成）PnP IRP 列表的结构的指针，并提供指向延迟的系统辅助角色队列的指针。

- 使用 **dt （显示类型）** 命令并指定 **nt!TRIAGE_9F_PNP** 结构和在 Arg4 中找到的地址。

dbgcmd

```
    kd> dt nt!TRIAGE_9F_PNP 82931b24
       +0x000 Signature        : 0x8001
       +0x002 Revision         : 1
       +0x004 CompletionQueue  : 0x82970e20
_TRIAGE_PNP_DEVICE_COMPLETION_QUEUE
       +0x008 DelayedWorkQueue : 0x829455bc _TRIAGE_EX_WORK_QUEUE
```

dt（显示类型）命令显示结构。 可以使用调试器命令跟踪 LIST_ENTRY 字段，检查未完成的 PnP IRP 列表。

为了帮助您确定错误原因，请考虑以下问题：

- 是否有与线程相关的 IRP？
- CompletionQueue 中是否有 IO？
- 堆栈上有哪些符号？
- 请参阅上文参数 0x3 下介绍的其他技术。

注解

如果您不具备使用上述技术来调试此问题的能力，可以使用一些基本的故障排除技术。

- 如果最近添加了新的设备驱动程序或系统服务，请尝试删除或更新它们。 尝试确定系统中导致新 Bug 检查代码出现的原因。
- 在**设备管理器**中查看是否有任何设备标有感叹号 (!)。 查看驱动程序属性中显示的事件日志，以查找任何故障驱动程序。 请尝试更新相关驱动程序。
- 检查事件查看器中的系统日志，以获取可能有助于查明导致错误的设备或驱动程序的其他错误消息。 有关详细信息，请参阅打开事件查看器。 在系统日志中查找与蓝屏同时出现的严重错误。
- 要尝试找出原因，请使用控制面板、电源选项暂时禁用省电功能。 某些驱动程序问题与系统休眠的各种状态以及电源的挂起和恢复有关。
- 如果最近向系统添加了硬件，请尝试删除或替换它。 或与制造商联系，查看是否有可用的修补程序。
- 你可尝试运行系统制造商提供的硬件诊断。
- 请与制造商联系，查看是否有更新的系统 ACPI/BIOS 或其他固件。

另请参阅

使用 Windows 调试器 (WinDbg) 进行故障转储分析

使用 WinDbg 分析内核模式转储文件

Bug 检查代码参考

Bug 检查 0xA0：INTERNAL_POWER_ERROR

项目 • 2024/06/28

INTERNAL_POWER_ERROR Bug 检查的值为 0x000000A0。 此 Bug 检查表明电源策略管理器遇到致命错误。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INTERNAL_POWER_ERROR 参数

参数 1 表示违规类型。 其他参数的含义取决于参数 1 的值。

展开表

参数 1	参数 2	参数 3	参数 4	原因
0x1	**1**：设备已超过其最大引用计数数量。 **2、3 或 4**：排队等候的浪涌电源 IRP 过多。 **5**：电源 IRP 已发送到被动级别设备对象。 **6**：系统未能分配必要的电源 IRP。	如果参数 2 的值为 1，则为允许的最大引用数。 如果参数 2 的值为 2、3 或 4，则为允许的最大挂起 IRP 数。 如果参数 2 的值为 6，则为目标设备对象。	如果参数 2 的值为 6，则表明这是系统 (0x0) 或设备 (0x1) 电源 IRP。	处理电源 I/O 请求数据包 (IRP) 时出错。
0x2	Reserved	预留	保留	尝试处理电源事件时发生内部故障。 有关详细信息，请参阅参数 1 等于 0x2 时调试 Bug 检查 0xA0。
0x3	预期校验和	实际校验和	失败的行号	休眠上下文页的校验和与其预期的校验和不匹配。
0x4	预期校验和	实际校验和	失败的行号	即将写入休眠文件的页面的校验和与预期校验和不匹配。
0x5	Reserved	预留	保留	未知的关闭代码已发送到系统关闭处理程序。
0x7	Reserved	预留	保留	发生了未经处理的异常。 有关详细信息，请参阅参数 1 等于 0x7 时调试 Bug 检查 0xA0。
0x8	此参数始终设置为 0x100。	设备对象	POWER_CHANNEL_SUMMARY	处理系统电源事件时发生致命错误。
0x9	状态代码	镜像阶段	Reserved	准备休眠文件时发生致命错误。
0xA	**0**：恢复后立即请求进行 Bug 检查。 **1**：在启用所有不可分页的设备后，在恢复期间请求了一个 Bug 检查。 **2**：在启用所有设备后，在恢复期间请求了一个 Bug 检查。	Reserved	保留	出于调试目的唤醒时，请求进行 Bug 检查。
0xB	休眠文件的大小。	空间耗尽前的休眠进度 **0**: HIBERFILE_PROGRESS_FREE_MAP	参数 2 为 4 时，剩余内存范围的大小。	休眠文件太小。

参数 1	参数 2	参数 3	参数 4	原因
		1: HIBERFILE_PROGRESS_RESUME_CONTEXT 2: HIBERFILE_PROGRESS_PROCESSOR_STATE 3: HIBERFILE_PROGRESS_SECURE_RANGES 4: HIBERFILE_PROGRESS_MEMORY_RANGES 5: HIBERFILE_PROGRESS_TABLE_PAGES 6: HIBERFILE_PROGRESS_MEMORY_IMAGE		
0xC	状态代码	转储堆栈上下文	Reserved	转储堆栈未能初始化。
0xD	正在转换的系统电源状态。	最近到达的睡眠检查点。	指向 POP_POWER_ACTION 结构的指针。	系统未能及时完成电源转换。
0xF	正在转换的系统电源状态。	最近到达的睡眠检查点。	指向当前处理请求的线程的指针。	系统未能及时完成电源转换。
0xF0	正在转换的系统电源状态。	最近到达的睡眠检查点。	指向当前处理请求的线程的指针。	系统未能及时完成（暂停）电源转换。
0xF1	正在转换的系统电源状态。	最近到达的睡眠检查点。	指向当前处理请求的线程的指针。	系统未能及时完成（恢复）电源转换。

展开表

参数 1	参数 2	参数 3	参数 4	原因
0x101	Reserved	异常指针。	Reserved	处理系统电源事件时发生未经处理的异常。 有关详细信息，请参阅参数 1 等于 0x101 时调试 Bug 检查 0xA0。
0x102	Reserved	DUMP_INITIALIZATION_CONTEXT	POP_HIBER_CONTEXT	休眠工作缓冲区大小未进行页面对齐。
0x103	Reserved	POP_HIBER_CONTEXT	Reserved	在休眠过程中，所有工作页都未能被考虑在内。
0x104	Reserved	POP_HIBER_CONTEXT	Reserved	在内部内存结构被锁定时，尝试映射内部休眠内存。
0x105	Reserved	POP_HIBER_CONTEXT	Reserved	尝试使用不受支持的内存类型标志映射内部休眠内存。
0x106	Reserved	内存描述符列表 (MDL)	Reserved	在休眠过程中创建了内存描述符列表，描述了未分页对齐的内存。
0x107	Reserved	POP_HIBER_CONTEXT	PO_MEMORY_RANGE_ARRAY	内部休眠数据结构中出现数据不匹配。
0x108	Reserved	POP_HIBER_CONTEXT	Reserved	磁盘子系统未能正确写入部分休眠文件。
0x109	Reserved	预期校验和	实际校验和	处理器状态数据的校验和与其预期校验和不匹配。
0x10A	Reserved	POP_HIBER_CONTEXT	NTSTATUS 失败代码	磁盘子系统未能正确读取或写入部分休眠文件。
0x10B	Reserved	当前休眠进度	Reserved	试图使用 PoSetHiberRange API 在错误的时间标记休眠启动阶段的页面。

参数 1	参数 2	参数 3	参数 4	原因
0x10C	Reserved	提供给 API 的标志	要标记的长度	使用无效参数调用了 PoSetHiberRange API。
0x10D	Reserved	POP_HIBER_CONTEXT	NTSTATUS 失败代码	提供恢复数据时，安全内核子系统失败。
0x10E	Reserved	校验和不正确	以前的磁盘读取校验和	从休眠文件读取时，磁盘子系统返回损坏数据。
0x10F	当前系统睡眠检查点。	内部错误的类型。 0：在分页禁用时但在 Po 禁用所有处理器上的中断之前写入检查点。 1：非 0 的 CPU 尝试在系统睡眠的中断禁用阶段写入检查点。 2：系统中的另一段代码正在执行 EFI 运行时服务。	Reserved	在检查指向系统睡眠进度时发生内部错误。
0x110	Reserved	预留	保留	系统未能禁用系统睡眠状态，但为确保数据完整性必须这样做。
0x111	Reserved	预留	保留	驱动程序已表明用户存在，并且用户已启用调试选项来捕获调用堆栈。
0x200	Reserved	DEVICE_OBJECT	DEVICE_OBJECT_POWER_EXTENSION	正在检查未知设备类型的空闲状态。
0x300	Reserved	DEVICE_OBJECT	IRP	从电池电源 IRP 返回未知状态。
0x301	Reserved	DEVICE_OBJECT	IRP	电池已进入未知状态。
0x400	Reserved	IO_STACK_LOCATION	DEVICE_OBJECT	设备已超过其最大引用计数数量。
0x401	Reserved	挂起的 IRP 列表	DEVICE_OBJECT	排队等候的浪涌电源 IRP 过多。
0x402	Reserved	挂起的 IRP 列表	DEVICE_OBJECT	排队等候的浪涌电源 IRP 过多。
0x403	Reserved	挂起的 IRP 列表	DEVICE_OBJECT	排队等候的浪涌电源 IRP 过多。
0x404	Reserved	IO_STACK_LOCATION	DEVICE_OBJECT	电源 IRP 已发送到被动级别设备对象。
0x500	Reserved	IRP	DEVICE_OBJECT	从热电 IRP 返回未知状态。

展开表

参数 1	参数 2	参数 3	参数 4	原因
0x600	DEVICE_OBJECT PDO	Reserved	保留	驱动程序试图使用 Power Runtime Framework 进行重复注册。
0x601	POP_FX_DEVICE 设备	PEP_DEVICE_REGISTER PEP	Reserved	没有 Power Engine 插件接受设备注册。
0x602	DEVICE_NODE 设备节点	睡眠计数	Reserved	设备节点睡眠计数与其激活计数不匹配。
0x603	POP_FX_PLUGIN	工作请求类型	Reserved	Power Engine 插件发出了无效的工作请求。
0x605	通知 ID	POP_FX_PLUGIN	Reserved	Power Engine 插件未能接受强制性设备电源管

参数 1	参数 2	参数 3	参数 4	原因
				理通知。
0x606	POP_FX_COMPONENT	POP_FX_COMPONENT_FLAGS	组件的新条件	当资源已处于活动状态（或空闲）状态时，Power Engine 插件尝试将关键系统资源组件转换为活动（或空闲）条件。
0x607	POP_FX_DEVICE	NTSTATUS	Reserved	当要求成功时，获取运行时电源管理框架设备删除锁定失败。
0x608	POP_FX_COMPONENT	POP_FX_COMPONENT_FLAGS	Reserved	驱动程序试图在没有先前的活动请求的情况下将组件转换为空闲状态。
0x609	POP_FX_PLUGIN	POP_FX_DEVICE	重复的请求类型 **0**: DevicePowerRequired **1**: DevicePowerNotRequired	Power Engine 插件已请求“需要设备电源”或“不需要设备电源”，没有提出相反类型的干预请求。
0x610	POP_FX_PLUGIN	POP_FX_DEVICE	Reserved	Power Engine 插件已请求“不需要设备电源”，而以前的“需要设备电源”请求尚未处理完成。
0x611	POP_FX_PLUGIN	POP_FX_DEVICE	组件索引无效	Power Engine 插件已请求对无效组件执行操作。
0x612	POP_FX_PLUGIN PowerEnginePlugin	Reserved	保留	Power Engine 插件已请求在设备通知的上下文中执行其他工作，其中 PO 未为该请求提供任何缓冲区。
0x613	POP_FX_DEVICE	组件索引	操作 **0**：无需完整的设备电源 **1**：报告设备已打开 **2**：完全空闲条件	当没有此类未完成的请求挂起时，驱动程序试图完成请求。
0x614	POP_FX_DEVICE	组件索引	非法参数 **0**： IRQL 中使用的 PO_FX_FLAG_BLOCKING >= DISPATCH_LEVEL **1**：指定了 PO_FX_FLAG_BLOCKING 和 PO_FX_FLAG_ASYNC_ONLY **2**：组件索引无效	驱动程序请求对具有非法参数的组件执行活动/空闲转换。
0x615	POP_FX_PLUGIN	POP_FX_COMPONENT	非法操作 **0**：组件未处于空闲状态 0 **1**：组件已处于活动状态 **2**：没有未完成的激活请求 **3**：未完成的空闲状态转换	Power Engine 插件非法指示组件激活完成。
0x616	POP_FX_PLUGIN	POP_FX_COMPONENT	非法操作 **0**：无效的空闲状态 **1**：组件已处于请求状态	Power Engine 插件已非法请求组件空闲状态转换。

参数 1	参数 2	参数 3	参数 4	原因
			2：请求了非零空闲状态，未通过空闲状态 0	
0x617	POP_FX_PLUGIN PowerEnginePlugin	UNICODE_STRING DeviceId	PEP_DEVICE_REGISTER PEP 注册	处理设备注册通知时，Power Engine 插件已返回无效的接受类型。
0x618	POP_FX_WORK_ORDER_WATCHDOG_INFO WorkOrder	Reserved	保留	运行时电源工作线程已阻塞太久。
0x619	POP_FX_DEVICE 设备	组件索引	实际负责的子设备的 NULL 或 DEVICE_NODE	设备长时间阻止进入最深的运行时空闲电源状态。
0x61A	POP_FX_PLUGIN Power Engine Plugin	POP_FX_DEVICE 设备	Reserved	Power Engine 插件提供了有关组件性能状态信息的无效信息。
0x61B	POP_FX_DEVICE 设备	组件索引	Reserved	驱动程序在注册设备性能状态之前已发出性能状态请求。
0x61C	POP_FX_DEVICE 设备	组件索引	无效参数 值： 0：PerfChangesCount 超过为此组件注册的性能状态集数	驱动程序已发出具有无效参数的性能状态请求。
0x61D	POP_FX_DEVICE 设备	组件索引	未完成的请求上下文	驱动程序在以前的请求未完成时发出了性能状态请求。
0x61E	Reserved	预留	保留	启用自动转换时，Power Engine 插件试图在调试器设备上执行关键转换。
0x61F	POP_FX_DEVICE 设备	协调空闲状态索引	Reserved	Power Engine 插件试图为非平台范围状态的协调空闲状态启用自动调试器转换。
0x620	POP_FX_DEVICE 设备	协调空闲状态索引	Reserved	Power Engine 插件尝试为非平台范围状态的协调空闲状态注册 D 状态依赖项。
0x621	POP_FX_DEVICE 设备	组件索引	协调空闲状态索引	Power Engine 插件尝试为非平台范围状态的协调空闲状态注册 F 状态依赖项。
0x622	父 POP_FX_COMPONENT	子 POP_FX_COMPONENT	Reserved	驱动程序尝试从具有未完成依赖项的 PoFx 取消注册。
0x666	PPOP_PEP_ACTIVITY	新活动类型 **0**: DevicePowerOn **1**: ComponentIdleStateChange **2**: ComponentActivating **3**: ComponentActive **4**: DevicePowerOff **5**: DeviceSuspend	冲突的活动类型 **0**: DevicePowerOn **1**: ComponentIdleStateChange **2**: ComponentActivating **3**: ComponentActive **4**: DevicePowerOff **5**: DeviceSuspend	默认 Power Engine 插件试图触发与另一个活动冲突的新活动。
0x667	POP_PEP_ACTIVITY	活动类型 **0**: DevicePowerOn	POP_PEP_ACTIVITY_STATUS	默认 Power Engine 插件试图完成未运行的活动。

参数 1	参数 2	参数 3	参数 4	原因
		1: ComponentIdleStateChange **2**: ComponentActivating **3**: ComponentActive **4**: DevicePowerOff **5**: DeviceSuspend		
0x668	正在更新引用计数的 PPPM_COORDINATED_STATE。	此函数观察到的无效引用计数值。	正在更新的平台空闲状态的掩码。	默认 Power Engine 插件试图删除以前未受约束的平台空闲状态约束。
0x669	正在更新引用计数的 PPPM_COORDINATED_STATE。	此函数观察到的无效引用计数值。	Reserved	默认 Power Engine 插件在尝试以独占方式通知 PoFx 平台空闲状态的可用性时，遇到内部一致性错误。

展开表

参数 1	参数 2	参数 3	参数 4	原因
0x680	NTSTATUS 失败代码。	Reserved	保留	运行时 Power Framework ACPI 表，因为该表缺失或 通常是因为 BIOS 错误。
0x700	PEPHANDLE	PEP_PPM_IDLE_SELECT	Reserved	Power Engine 插件指定了无 闲依赖项。
0x701	挂起处理器的所选空闲状态的索引	挂起处理器的 PRCB 地址	挂起处理器的索引	处理器无法在分配的间隔内 换。 这表示指定的处理器
0x702	处理器的所选空闲状态的索引	处理器的空闲同步状态	挂起处理器的 PRCB 地址	处理器从不可中断的状态唤 系统通过 PEP 发起显式唤醒 PPM 空闲同步）。
0x703	PEPHANDLE	PEP_PPM_QUERY_PLATFORM_STATE	Reserved	Power Engine 插件在查询平 间指定了无效的处理器空闲
0x704	Reserved	预留	保留	协调的空闲状态转换没有及
0x705	PEPHANDLE	通知	标识非法更改字段的四个字符标记。 内核调试器中的解码标记，其中包含：.formats 标记，标记包含在< >中。	Power Engine 插件已更改 区中的只读字段。
0x706	通知	标识包含非法值的字段的四个字符标记。 内核调试器中的解码标记，其中包含：.formats 标记，标记包含在< >中。	非法值或存在非法值的数组索引	Power Engine 插件在传递至 的某个字段中返回了非法值
0x800	CS 当前状态	Reserved	保留	当系统处于连接待机状态时 打开。
0x801	显示状态更改原因	更新显示状态的会话 ID	Reserved	发生了无效的显示状态转换
0x802	导致显示关闭的 POWER_MONITOR_REQUEST_REASON	如果启用了 Power Event Processor，则为 1，否则为 0。	指向 POP_PDC_IDLE_PHA标准版 _WATCHDOG_CONTEXT 全局的指针。	PDC 系统空闲阶段 (NoCsP 到新式待机的时间超过预期
0x900	指向负责任\的电源设置回调的指针	调用电源设置回调之前的 IRQL	从电源设置回调返回后的 IRQL	注册的电源设置回调返回了 IRQL。 这表示回调更改了 回之前未还原原始 IRQL。
0x901	DEVICE_OBJECT	IRP	线程的 APC 禁用计数	处理电源 IRP 时，驱动程序 核 APC。
0x4001	KE 错误子代码。	Reserved	保留	(INTERNAL_POWER_ERRO 在电源操作过程中，内核执

参数 1	参数 2	参数 3	参数 4	原因
	值： 0x100： (INTERNAL_POWER_ERROR_KE_PROCESSOR_ON_TIMED_OUT) 固件启动处理器的时间过长。 0x101： (INTERNAL_POWER_ERROR_KE_INVALID_INTERRUPT_TARGET) 指定的中断目标无效。 0x102：(INTERNAL_POWER_ERROR_KE_标准版 TDESTINATION_FAILED) 未能更改中断线的目标位置。 0x103：(INTERNAL_POWER_ERROR_KE_IPI_REQUEST_FAILED) 在重定向中断时未能发出 PI。 0x104： (INTERNAL_POWER_ERROR_KE_ARCH_NOT_SUPPORTED) 不受 支持的处理器体系结构。			故障。
0xAA64	错误代码	PSCI 函数 ID 正在进行中	可选的内部上下文相关 数据	AArm64 Power State Coor Interface (PSCI) 函数遇到不 错误。

解决方法

一般说明

在上表中，多个参数是指向结构的指针。 例如，如果参数 2 被列为 DEVICE_OBJECT，则参数 2 是指向 DEVICE_OBJECT 结构的指针。 某些结构在 wdm.h 中定义，该结构包含在 Windows 驱动程序工具包中。 例如，以下结构在 wdm.h 中定义。

- EXCEPTION_POINTERS
- DEVICE_OBJECT
- IO_STACK_LOCATION
- PEP_DEVICE_REGISTER

上表中显示的某些结构未在任何公共头文件中定义。 可以使用 dt 调试程序命令查看这些结构的定义。 以下示例演示如何使用 **dt** 命令查看 **DEVICE_OBJECT_POWER_EXTENSION** 结构。

dbgcmd

```
3: kd> dt nt!DEVICE_OBJECT_POWER_EXTENSION
   +0x000 IdleCount        : Uint4B
   +0x004 BusyCount        : Uint4B
   +0x008 BusyReference    : Uint4B
   +0x00c TotalBusyCount   : Uint4B
   +0x010 ConservationIdleTime : Uint4B
   +0x014 PerformanceIdleTime : Uint4B
   +0x018 DeviceObject     : Ptr64 _DEVICE_OBJECT
   +0x020 IdleList         : _LIST_ENTRY
   +0x030 IdleType         : _POP_DEVICE_IDLE_TYPE
   +0x034 IdleState        : _DEVICE_POWER_STATE
   +0x038 CurrentState     : _DEVICE_POWER_STATE
   +0x040 Volume           : _LIST_ENTRY
   +0x050 Specific         : <unnamed-tag>
```

以下过程将帮助你调试此 Bug 检查的某些实例。

参数 1 等于 0x2 时调试 Bug 检查 0xA0

1. 检查调用堆栈。 查找 **ntoskrnl！PopExceptionFilter** 函数。 此函数的第一个参数包含以下代码。

   C++

   ```
   (error_code << 16) | _LINE_
   ```

   如果调用方为 **PopExceptionFilter**，则此函数的第一个参数的类型为 PEXCEPTION_POINTERS。 请注意此参数的值。

2. 使用 dt **（显示类型）** 命令并指定在上一步中找到的值作为参数。

```
dbgcmd

dt nt!_EXCEPTION_POINTERS argument
```

此命令显示结构。 请注意上下文记录的地址。

3. 使用 **.cxr（显示上下文记录）**命令并指定在上一步中找到的上下文记录作为*记录*。

```
dbgcmd

.cxr record
```

此命令将寄存器上下文设置为正确的值。

4. 使用各种命令分析错误的来源。 以 **kb 开头（显示堆栈回溯）。**

参数 1 等于 0x7 时调试 Bug 检查 0xA0

1. 检查调用堆栈。 查找 **ntoskrnl！PopExceptionFilter** 函数。 此函数的第一个参数的类型为 PEXCEPTION_POINTERS。 请注意此参数的值。

2. 使用 **dt （显示类型）** 命令并指定在上一步中找到的值作为*参数*。

```
dbgcmd

dt nt!_EXCEPTION_POINTERS argument
```

此命令显示结构。 请注意上下文记录的地址。

3. 使用 **.cxr（显示上下文记录）**命令并指定在上一步中找到的上下文记录作为*记录*。

```
dbgcmd

.cxr record
```

此命令将寄存器上下文设置为正确的值。

4. 使用各种命令分析错误的来源。 以 **kb 开头（显示堆栈回溯）。**

参数 1 等于 0x101 时调试 Bug 检查 0xA0

1. 使用 **dt （显示类型）** 命令并将参数 3 的值指定为*参数*。

```
dbgcmd

dt nt!_EXCEPTION_POINTERS argument
```

此命令显示结构。 请注意上下文记录的地址。

2. 使用 **.cxr（显示上下文记录）**命令并指定在上一步中找到的上下文记录作为*记录*。

```
dbgcmd

.cxr record
```

此命令将寄存器上下文设置为正确的值。

3. 使用各种命令分析错误的来源。 以 **kb 开头（显示堆栈回溯）。**

另请参阅

错误检查代码参考

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Bug 检查 0xA1: PCI_BUS_DRIVER_INTERNAL

项目 • 2024/02/21

PCI_BUS_DRIVER_INTERNAL Bug 检查的值为 0x000000A1。 此 Bug 检查表明 PCI 总线驱动程序检测到其内部结构中存在不一致问题，无法继续运行。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PCI_BUS_DRIVER_INTERNAL 参数

无

另请参阅

Bug 检查代码参考

Bug 检查 0xA2：MEMORY_IMAGE_CORRUPT

项目 • 2024/02/23

MEMORY_IMAGE_CORRUPT Bug 检查的值为 0x000000A2。 此 Bug 检查表明检测到内存中可执行文件的映像出现损坏。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

MEMORY_IMAGE_CORRUPT 参数

参数 1 表示违规类型。 其他参数的含义取决于参数 1 的值。

展开表

参数 1	参数 2	参数 3	参数 4	原因
0x02	**如果参数 3 为零**：表页中失败的页码 **如果参数 3 为非零**：包含失败页运行索引的页码	零，或未能与运行匹配的索引	0	发生表页检查失败。
0x03	范围的起始物理页码	范围的长度（以页为单位）	包含此运行的表页的页码	列出的内存范围的校验和不正确。

原因

内存范围的循环冗余校验 (CRC) 检查失败。

在系统唤醒操作中，可能会检查内存的各个区域来防范内存故障。

另请参阅

Bug 检查代码参考

Bug 检查 0xA3：ACPI_DRIVER_INTERNAL

项目 • 2024/02/20

ACPI_DRIVER_INTERNAL Bug 检查的值为 0x000000A3。 此 Bug 检查表明 ACPI 驱动程序检测到内部不一致。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

ACPI_DRIVER_INTERNAL 参数

展开表

参数	说明
1	保留
2	Reserved
3	Reserved
4	Reserved

原因

ACPI 驱动程序中的不一致非常严重，继续运行会导致严重问题。

造成此问题的一个可能根源原因是 BIOS 错误。

另请参阅

Bug 检查代码参考

Bug 检查 0xA4：CNSS_FILE_SYSTEM_FILTER

项目 • 2024/02/20

CNSS_FILE_SYSTEM_FILTER Bug 检查的值为 0x000000A4。 此 Bug 检查表明 CNSS 文件系统筛选器中出现了问题。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

CNSS_FILE_SYSTEM_FILTER 参数

展开表

参数	说明
1	指定源文件和行号信息。 高 16 位（“0x”后面的前四个十六进制数字）按标识符号标识源文件。 低 16 位标识文件中出现 Bug 检查的源行。
2	Reserved
3	Reserved
4	Reserved

原因

因为非分页池内存已满，可能会发生 CNSS_FILE_SYSTEM_FILTER Bug 检查。 如果非分页池内存已满，此错误可能会导致系统停止运行。 但是，在索引过程中，如果可用非分页池内存量非常低，则另一个需要非分页池内存的内核模式驱动程序也可能会触发此错误。

解决方法

若要解决非分页池内存耗尽问题： 向计算机添加新的物理内存。 该内存会增加内核可用的非分页池内存的数量。

另请参阅

Bug 检查代码参考

Bug 检查 0xA5：ACPI_BIOS_ERROR

项目 • 2024/02/20

ACPI_BIOS_ERROR Bug 检查的值为 0x000000A5。 此 Bug 检查表明计算机的高级配置和电源接口 (ACPI) BIOS 不完全符合 ACPI 规范。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

ACPI_BIOS_ERROR 参数

参数 1 标识不兼容的类型。 其他参数的含义取决于参数 1 的值。

如果 BIOS 不兼容与即插即用 (PnP) 或电源管理相关，则使用以下参数。

展开表

参数 1	参数 2	参数 3	参数 4	原因
0x01	ACPI 的 **deviceExtension**	ACPI 的 **ResourceList**	**0**：找不到资源列表 **1**：列表中未找到 IRQ 资源	ACPI 无法在启动 ACPI 时交给它的资源中找到系统控制中断 (SCI) 向量。
0x02				（请参阅本页后面的表格）
0x03	正在运行的 ACPI 对象	解释器的返回值	控件方法的名称（采用 ULONG 格式）	ACPI 在创建设备扩展来表示 ACPI 命名空间时尝试运行控制方法，但此控制方法失败了。
0x04	_PRW 所属的 ACPI 扩展	指向方法的指针	返回 **的 DataType** （请参阅 Amli.h）	ACPI 评估了一个 _PRW，并期望找到一个整数作为包元素。
0x05	_PRW 所属的 ACPI 扩展	Aointer 到 _PRW	_PRW 中的元素数量	ACPI 评估了 _PRW，返回的包未能包含至少两

参数 1	参数 2	参数 3	参数 4	原因
				个元素。 ACPI 规范要求两个元素始终存在于 _PRW 中。
0x06	_PRx 所属的 ACPI 扩展	指向 _PRx 的指针	指向要查找的对象的名称的指针	ACPI 尝试查找命名对象，但找不到该对象。
0x07	方法所属的 ACPI 扩展	指向方法的指针	返回 **的 DataType** （请参阅 Amli.h）	ACPI 评估了一种方法，并期望在返回中接收缓冲区。 但是，该方法返回了一些其他数据类型。
0x08	方法所属的 ACPI 扩展	指向方法的指针	返回 **的 DataType** （请参阅 Amli.h）	ACPI 评估了一个方法，并期望在返回中接收整数。 但是，该方法返回了一些其他数据类型。
0x09	方法所属的 ACPI 扩展	指向方法的指针	返回 **的 DataType** （请参阅 Amli.h）	ACPI 评估了一种方法，并期望在返回中接收数据包。 但是，该方法返回了一些其他数据类型。
0x0A	方法所属的 ACPI 扩展	指向方法的指针	返回 **的 DataType** （请参阅 Amli.h）	ACPI 评估了一种方法，并期望在返回中接收字符串。 但是，该方法返回了一些其他数据类型。
0x0B	_EJD 所属的 ACPI 扩展	解释器返回的状态	ACPI 尝试查找的对象名称	ACPI 找不到 _EJD 字符串引用的对象。
0x0C	ACPI 为其找到了扩展坞设备的 ACPI 扩展	指向 _EJD 方法的指针	**0**：BIOS 未声明系统为停靠状态 **1**：扩展坞设备的重复设备扩展	ACPI 为扩展坞支持提供了错误或不充分的信息。
0x0D	ACPI 需要对象的 ACPI 扩展	ACPI 查找的方法的 (ULONG) 名称	**0**：基本情况 **1**：冲突	ACPI 在命名空间中找不到所需的

参数 1	参数 2	参数 3	参数 4	原因
				或对象。如果没有 _HID 或 _ADR，则使用此 Bug 检查代码。
0x0E	ACPI 需要对象的 NS **PowerResource**	ACPI 查找的方法的 (ULONG) 名称	0：基本情况	ACPI 在电源资源（或“设备”以外的实体）的命名空间中找不到所需方法或对象。 如果电源资源没有 _ON、_OFF 或 _STA，则使用此 Bug 检查 代码。
0x0F	ACPI 正在分析的当前缓冲区	缓冲区的标记	缓冲区的指定长度	ACPI 无法分析资源描述符。
0x10				（请参阅本页后面的表格）
0x11				（请参阅本页后面的表格）
0x14	ACPI 正在分析的当前缓冲区	缓冲区的标记	指向包含缓冲区的 ULONGLONG 长度的变量的指针	ACPI 无法分析资源描述符。 长度超过 MAXULONG。
0x15	ACPI 计算机语言 (AML) 上下文	**1**：未能加载表 **2**：找不到参数路径字符串对象 **3**：未能将参数数据插入参数路径字符串对象 **4**：系统内存不足	NT 状态代码	尝试加载表时，ACPI 出现致命错误。
0x16	指向父 NSOBJ 的指针	指向非法子 ACPI 命名空间对象的指针	Reserved	ACPI 在处理 xSDT 时出现致命错误。 对象被声明为不能具有子级的父级的子级。

如果发生中断路由失败或不兼容，则使用以下参数。

展开表

参数 1	参数 2	参数 3	参数 4	原因
0x2001	**InterruptModel**（整数）	解释器的返回值	指向控制方法的指针	ACPI 试图评估 PIC 控制方法，但失败了。
0x10001	指向设备对象的指针	指向设备对象父级的指针	指向 _PRT 对象的指针 （请参阅以下注释部分）	ACPI 尝试执行中断路由，但失败了。
0x10002	指向设备对象的指针	指向 ACPI 正在查找但找不到的字符串名称的指针	指向 _PRT 对象的指针 （请参阅以下注释部分）	ACPI 找不到 _PRT 中引用的链接节点。
0x10003	指向设备对象的指针	设备 ID 或函数编号。 此 DWORD 编码如下：位5:0 是 PCI 设备编号，位 8:6 是 PCI 函数编号	指向 _PRT 对象的指针 （请参阅以下注释部分）	ACPI 在设备的 _PRT 包中找不到映射。
0x10005	指向 _PRT 对象的指针 （请参阅以下注释部分）	指向当前 _PRT 元素的指针。 （此指针是 _PRT 中的索引。	设备 ID 或函数编号。 此 DWORD 编码如下：位5:0 是 PCI 函数编号，位 31:16 是 PCI 设备编号	ACPI 在 _PRT 中找到了函数 ID 不是所有 F 的条目。 （_PRT 条目的泛型格式是指定设备编号，但函数编号不是。）
0x10006	指向链接节点的指针。 （此设备缺少 _DIS 方法。）	0	0	ACPI 找到了链接节点，但无法禁用该节点。 （必须禁用链接节点才能重新编程。）
0x10007	找不到的向量	0	0	_PRT 包含对 I/O APIC 条目的 MAPIC 表中未描述的向量的引用。

参数 1	参数 2	参数 3	参数 4	原因
0x10008	无效的中断级别。	0	0	ACPI SCI 中断级别无效。
0x10009	0	0	0	无法找到固定的 ACPI 说明表 (FADT)。
0x1000A	0	0	0	无法找到根系统说明指针 (RSDP) 或扩展系统说明表 (XSDT)
0x1000B	ACPI 表签名	指向 ACPI 表的指针	0	ACPI 表的长度与表修订不一致。
0x1000C	修订 ID	功能索引	0	用于中断返回的格式错误数据的 _DSM 方法
0x1000D	设备的 ACPI 扩展	值 0：指定 _PRW 时无可唤醒的中断，并且至少有一个 GPIO 中断 值 1：由于存在可唤醒的中断，_PRW 应指定 0xffffffff 的 GpeInfo 值	0	不支持使用 GPE 和 GPIO 中断的设备。
0x1000E	验证函数返回的状态。	指向 ACPI 命名空间路径 UNICODE_STRING 的指针。	指向与 SDEV 进行比较的资源列表的指针。	安全设备的 SDEV 资源与其相应的 _CRS 或 _PRS 条目不匹配。

如果出现其他故障或不兼容，则使用以下参数。

展开表

参数 1	参数 2	参数 3	参数 4	原因
0x20000	固定表中的 I/O 端口	0	0	固定 ACPI 说明表中的 PM_TMR_BLK 条目未指向工作 ACPI 计时器块。

该表介绍了使用以下参数时的内存使用问题。

展开表

参数 1	参数 2	参数 3	参数 4	原因
0x1000	内存区域物理地址的高位部分。	内存区域物理地址的低位部分。	被映射的内存长度。	ACPI 在处理内存操作区域时出现严重错误。 内存操作区域尝试映射分配给操作系统使用的内存。

如果参数 1 等于 **0x02**，ACPI BIOS 无法处理 PCI 根总线的资源列表。 在这种情况下，参数 3 指定确切的问题，其余参数的定义如下。

展开表

参数 2	参数 3	参数 4	原因
PCI 总线的 ACPI 扩展	0x0	指向 QUERY_RESOURCES IRP 的指针	ACPI 无法将 BIOS 的资源列表转换为正确的格式。 这可能表示 BIOS 列表编码过程中出现错误。
PCI 总线的 ACPI 扩展	0x1	指向 QUERY_RESOURCE_REQUIREMENTS IRP 的指针	ACPI 无法将 BIOS 的资源列表转换为正确的格式。 这可能表示 BIOS 列表编码过程中出现错误。
PCI 总线的 ACPI 扩展	0x2	0	ACPI 发现一个空的资源列表。
PCI 总线的 ACPI 扩展	0x3	指向 PNP CRS 描述符的指针	ACPI 在 CRS 中找不到当前总线编号。
PCI 总线的 ACPI 扩展	指向 PCI 资源列表的指针	指向 E820 内存表的指针	PCI 声明解码的资源列表与 E820 BIOS 接口报告的内存区域列表重叠。 （这种冲突是绝不允许的。）

如果参数 1 等于 **0x10**，ACPI BIOS 无法正确确定系统到设备状态的映射。 在这种情况下，参数 3 指定确切的问题，其余参数的定义如下。

展开表

参数 2	参数 3	参数 4	原因
需要映射的 ACPI 扩展	0x0	DEVICE_POWER_STATE（这是“x+1”）	_PRx 已映射回不支持的 S 状态。
需要映射的 ACPI 扩展	0x1	无法映射的 SYSTEM_POWER_STATE	ACPI 找不到要与 S 状态相关的 D 状态。
需要映射的 ACPI 扩展	0x2	无法映射的 SYSTEM_POWER_STATE	设备声称能够在系统处于此 S 状态时唤醒系统，但实际上系统不支持此 S 状态。

如果参数 1 等于 **0x11**，则系统无法进入 ACPI 模式。 在这种情况下，参数 2 指定确切的问题，其余参数的定义如下。

展开表

参数 2	参数 3	参数 4	原因
0x0	0	0	系统无法初始化 AML 解释器。
0x1	0	0	系统找不到 RSDT。
0x2	0	0	系统无法分配关键驱动程序结构。
0x3	0	0	系统无法装载 RSDT。
0x4	0	0	系统无法装载 DDBs。
0x5	0	0	系统无法连接中断向量。
0x6	0	0	在 PM1 控制寄存器中，SCI_EN 永远不会被设置。
0x7	指向具有错误校验和的表的指针	Creator 修订	表校验和不正确。
0x8	指向 ACPI 无法加载的表的指针	Creator 修订	ACPI 无法加载 DDB。
0x9	FADT 版本	0	不受支持的固件版本。

参数 2	参数 3	参数 4	原因
0xA	0	0	系统找不到 MADT。
0xB	0	0	系统在 MADT 中找不到任何有效的本地 SAPIC 结构。

原因

参数 1 的值表示出现错误。

解决方法

如果要调试此错误，请使用 **!analyze -v** 扩展。 此扩展显示所有相关数据（设备扩展、nsobjects 或任何适合特定错误的内容）。

如果未执行调试，则此错误表示必须获取新的 BIOS。 请联系供应商或访问 Internet 获取新的 BIOS。

如果无法获取更新的 BIOS，或者最新的 BIOS 仍不符合 ACPI 标准，则可以在文本模式设置期间关闭 ACPI 模式。 若要关闭 ACPI 模式，请在系统提示安装存储驱动程序时按 F7 键。 系统不会通知您 F7 键已按下，但它以无提示方式禁用 ACPI 并允许您继续安装。

注解

PCI 路由表 (_PRT) 是 ACPI BIOS 对象，用于指定如何将所有 PCI 设备连接到中断控制器。 具有多个 PCI 总线的计算机可能有多个 _PRTs。

使用 **!acpikd.nsobj** 扩展并将 _PRT 对象的地址作为其参数，可以在调试器中显示 _PRT 对象。

另请参阅

Bug 检查代码参考

Bug 检查 0xA7：BAD_EXHANDLE

项目 • 2024/02/20

BAD_EXHANDLE Bug 检查的值为 0x000000A7。 此 Bug 检查表明内核模式句柄表检测到句柄表条目状态不一致。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

BAD_EXHANDLE 参数

无

另请参阅

Bug 检查代码参考

Bug 检查 0xAC：HAL_MEMORY_ALLOCATION

项目 • 2024/02/20

HAL_MEMORY_ALLOCATION Bug 检查的值为 0x000000AC。 此 Bug 检查表明硬件抽象层 (HAL) 无法获取足够的内存。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

HAL_MEMORY_ALLOCATION 参数

展开表

参数	说明
1	分配大小
2	0
3	指向包含文件名的字符串的指针。
4	Reserved

原因

HAL 无法获取系统关键要求的非分页内存池。

这些关键内存分配是在系统初始化的早期进行的，预计不会进行 HAL_MEMORY_ALLOCATION Bug 检查。 此 Bug 检查可能表示其他一些严重错误，例如池损坏或大量消耗。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

另请参阅

Bug 检查代码参考

Bug 检查 0xAD：VIDEO_DRIVER_DEBUG_REPORT_REQUEST

项目 • 2024/02/20

VIDEO_DRIVER_DEBUG_REPORT_REQUEST Bug 检查的值为 0x000000AD。 此 Bug 检查表明视频端口在运行时代表视频驱动程序创建了一个非致命的小型转储。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

VIDEO_DRIVER_DEBUG_REPORT_REQUEST 参数

展开表

参数	说明
1	特定于驱动程序
2	特定于驱动程序
3	特定于驱动程序
4	自启动以来请求的所有报告数

注解

视频端口在运行时代表视频驱动程序创建了一个非致命的小型转储，因为视频驱动程序请求了调试报告。

VIDEO_DRIVER_DEBUG_REPORT_REQUEST Bug 检查只能由创建小型转储引起，而不是创建完整转储或内核转储。

另请参阅

Bug 检查代码参考

Bug 检查 0xB1: BGI_DETECTED_VIOLATION

项目 • 2024/02/20

BGI_DETECTED_VIOLATION Bug 检查的值为 0x000000B1。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

BGI_DETECTED_VIOLATION 参数

无

另请参阅

Bug 检查代码参考

Bug 检查 0xB4：VIDEO_DRIVER_INIT_FAILURE

项目 • 2024/02/21

VIDEO_DRIVER_INIT_FAILURE Bug 检查的值为 0x000000B4。 这表明 Windows 无法进入图形模式。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

VIDEO_DRIVER_INIT_FAILURE 参数

无

原因

系统无法进入图形模式，因为没有能够启动的显示驱动程序。

当没有视频微型端口驱动程序能够成功加载时，通常会发生这种情况。

另请参阅

Bug 检查代码参考

Bug 检查 0xB8：ATTEMPTED_SWITCH_FROM_DPC

项目 • 2024/02/20

ATTEMPTED_SWITCH_FROM_DPC Bug 检查的值为 0x000000B8。 这表明，延迟的过程调用 (DPC) 例程尝试了非法操作。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

ATTEMPTED_SWITCH_FROM_DPC 参数

展开表

参数	说明
1	导致失败的原始线程
2	新线程
3	原始线程的堆栈地址
4	Reserved

原因

尝试从 DPC 例程执行等待操作、附加进程或暂停。 这是非法操作。

解决方法

堆栈跟踪将指向导致错误的原始 DPC 例程中代码。

另请参阅

Bug 检查代码参考

错误检查：0xB9 CHIPSET_DETECTED_ERROR

项目 • 2024/03/04

CHIPSET_DETECTED_ERROR 错误检查的值为 0x000000B9。

这种错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并非常有助于确定根本原因。

另请参阅

错误检查代码参考

错误检查 0xBA：SESSION_HAS_VALID_VIEWS_ON_EXIT

项目 • 2024/03/08

SESSION_HAS_VALID_VIEWS_ON_EXIT 错误检查的值为 0x000000BA。 这表示卸载会话时，会话驱动程序仍具有映射视图。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SESSION_HAS_VALID_VIEWS_ON_EXIT 参数

展开表

参数	说明
1	会话 ID
2	正在泄漏的映射视图数
3	此会话的映射视图表的地址
4	此会话的映射视图表的大小

原因

此错误是由会话驱动程序在卸载会话之前未取消映射其映射视图引起的。 这表示 win32k.sys、atmfd.dll、rdpdd.dll 或视频驱动程序中存在错误。

另请参阅

错误检查代码参考

错误检查 0xBB：NETWORK_BOOT_INITIALIZATION_FAILED

项目 • 2024/03/08

NETWORK_BOOT_INITIALIZATION_FAILED 错误检查的值为 0x000000BB。 这表示 Windows 未能成功从网络启动。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

NETWORK_BOOT_INITIALIZATION_FAILED 参数

展开表

参数	说明
1	网络初始化失败的部分。 可能的值为： **1**：更新注册表时失败。 **2**：启动网络堆栈时失败。 Windows 将 IOCTL 发送到重定向程序和数据报接收器，然后等待重定向程序准备就绪。 如果在某个时间段内未准备就绪，则会发出此错误。 **3**：向 TCP 发送 DHCP IOCTL 时失败。 这就是 Windows 通知传输其 IP 地址的方式。
2	故障状态
3	Reserved
4	Reserved

原因

此错误是在 Windows 从网络启动时导致的，并且在 I/O 初始化过程中某个关键功能失败。

另请参阅

错误检查代码参考

错误检查 0xBC：NETWORK_BOOT_DUPLICATE_ADDRESS

项目 • 2024/03/04

NETWORK_BOOT_DUPLICATE_ADDRESS 错误检查的值为 0x000000BC。 这表示在从网络启动时向该计算机分配了重复的 IP 地址。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

NETWORK_BOOT_DUPLICATE_ADDRESS 参数

展开表

参数	说明
1	IP 地址，显示为 DWORD。 形式为 *aa.bb.cc.dd* 的地址将显示为 0xDDCCBBAA。
2	另一台计算机的硬件地址。 （有关以太网连接，请参阅以下说明。）
3	另一台计算机的硬件地址。 （有关以太网连接，请参阅以下说明。）
4	另一台计算机的硬件地址。 （对于以太网连接，这将为零。）

注意 当参数 4 等于零时，表示以太网连接。 在这种情况下，MAC 地址将存储在参数 2 和参数 3 中。 形式为 *aa-bb-cc-dd-ee-ff* 的以太网 MAC 地址将导致参数 2 等于 0xAABBCCDD，参数 3 等于 0xEEFF0000。

原因

此错误表明，当 TCP/IP 为其 IP 地址发送 ARP 时，它从另一台计算机得到了指示重复 IP 地址的响应。

当 Windows 从网络启动时，这是一个致命的错误。

另请参阅

错误检查代码参考

错误检查：0xBD INVALID_HIBERNATED_STATE

项目 • 2024/03/08

INVALID_HIBERNATED_STATE 错误检查的值为 0x000000BD。 这表示休眠的内存映像与当前硬件配置不匹配。 当系统从休眠状态恢复并发现硬件在系统休眠时发生更改时，就会发生此错误检查。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INVALID_HIBERNATED_STATE 参数

展开表

参数	说明
1	硬件无效。 1：安装的处理器数量少于休眠前 参数 2 中的值：休眠前的处理器数 参数 3 中的值：休眠后的处理器数
2	每个参数 1
3	每个参数 1
4	Reserved

错误检查 0xBE：ATTEMPTED_WRITE_TO_READONLY_MEMORY

项目 • 2024/03/04

ATTEMPTED_WRITE_TO_READONLY_MEMORY 错误检查的值为 0x000000BE。 如果驱动程序试图写入只读内存段，则会发出此消息。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

ATTEMPTED_WRITE_TO_READONLY_MEMORY 参数

展开表

参数	说明
1	尝试写入的虚拟地址
2	PTE 内容
3	Reserved
4	Reserved

如果能够识别出导致错误的驱动程序，则其名称将打印在蓝屏上，并存储在内存中的 (PUNICODESTRING) **KiBugCheckDriver 位置。**

注解

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查：0xBF MUTEX_ALREADY_OWNED

项目 • 2024/03/08

MUTEX_ALREADY_OWNED 错误检查的值为 0x000000BF。 这表示线程试图获取其已拥有的互斥体的所有权。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

MUTEX_ALREADY_OWNED 参数

展开表

参数	说明
1	互斥体的地址
2	导致错误的线程
3	0
4	Reserved

错误检查 0xC1: SPECIAL_POOL_DETECTED_MEMORY_CORRUPTION

项目 • 2024/03/04

SPECIAL_POOL_DETECTED_MEMORY_CORRUPTION 错误检查的值为 0x000000C1。 这表示驱动程序写入了特殊池的无效部分。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SPECIAL_POOL_DETECTED_MEMORY_CORRUPTION 参数

参数 4 表示违规类型。

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
驱动程序试图释放的地址	Reserved	0	0x20	驱动程序试图释放未分配的池。
驱动程序试图释放的地址	请求的字节数	计算的字节数（实际提供给调用方）	0x21， 0x22	驱动程序试图释放错误的地址。
驱动程序试图释放的地址	位损坏的地址	Reserved	0x23	驱动程序释放了地址，但同一页内的邻近字节已损坏。
驱动程序试图释放的地址	位损坏的地址	Reserved	0x24	驱动程序释放了地址，但分配结束后出现的字节已被覆盖。
当前 IRQL	池类型	字节数	0x30	驱动程序试图在不正确的 IRQL 中

参数 1	参数 2	参数 3	参数 4	错误消息的原因
				分配池。
当前 IRQL	池类型	驱动程序试图释放的地址	0x31	驱动程序试图在不正确的 IRQL 中释放池。
驱动程序试图释放的地址	一个位损坏的地址	Reserved	0x32	驱动程序释放了地址，但同一页内的邻近字节有一个位错误。

_POOL_TYPE 代码在 ntddk.h 中枚举。 其中，0 表示非分页池，1 表示分页池。

原因

驱动程序已写入到特殊池的无效部分。

解决方法

获取当前线程的回溯。 此回溯通常会揭示错误的来源。

有关特殊池的信息，请参阅 Windows 驱动程序工具包的驱动程序验证程序部分。

错误检查 0xC2：BAD_POOL_CALLER

项目 • 2024/03/04

BAD_POOL_CALLER 错误检查的值为 0x000000C2。 这表示当前线程正在发出错误的池请求。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

BAD_POOL_CALLER 参数

参数 1 表示违规类型。

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x00	0	池类型	池标记	当前线程请求零字节池分配。
0x01， 0x02， 0x04	指向池标头的指针	池标头内容的第一部分	0	池标头已损坏。
0x06	Reserved	指向池标头的指针	池标头内容	当前线程试图释放已释放的池。
0x07	Reserved	池标头内容	被释放的池块的地址	当前线程试图释放已释放的池。
0x08	当前 IRQL	池类型	分配的大小（以字节为单位）	当前线程试图在无效的 IRQL 处分配池。
0x09	当前 IRQL	池类型	池地址	当前线程试图在无效的 IRQL 处释放池。
0x0A	池地址	分配器的标记	试图释放中使用的标记	当前线程试图使用错误的标记释放池内存。 （内存可能属于另一个组件。）

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x0B， 0x0C， 或 0x0D	池地址	池分配的标记	配额进程指针错误	当前线程试图释放已损坏的池分配的配额。
0x40	开始地址	系统地址空间的开始	0	当前线程试图释放用户模式地址处的内核池。
0x41	开始地址	物理页帧	最高物理页帧	当前线程试图释放未分配的非分页池地址。
0x42 或 0x43	正在释放的地址	0	0	当前线程试图释放从未在任何池中的虚拟地址。
0x44	开始地址	Reserved	0	当前线程试图释放未分配的非分页池地址。
0x46	开始地址	0	0	当前线程试图释放无效的池地址。
0x47	开始地址	物理页帧	最高物理页帧	当前线程试图释放未分配的非分页池地址。
0x48	开始地址	预留	预留	当前线程试图释放未分配的分页池地址。
0x50	开始地址	从分页池开始的起始偏移量（以页为单位）	分页池的大小（以字节为单位）	当前线程试图释放未分配的分页池地址。
0x60	开始地址	0	0	当前线程试图释放无效的连续内存地址。 （**MmFreeContiguousMemory** 的调用方正在传递错误的指针。）
0x99	正在释放的地址	0	0	当前线程试图释放具有无效地址的池。 （此代码也可能表示池标头已损坏。）
0x9A	池类型	请求的字节数	池标记	当前线程将分配请求标记为 MUST_SUCCEED。 （不再支持此池类型。）

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x9B	池类型	请求的字节数	调用方的地址	当前线程尝试分配标记为 0 的池 （这无法跟踪，并可能损坏现有的标记表。）
0x9C	池类型	请求的字节数	调用方的地址	当前线程尝试分配标记为“BIG”的池： （这无法跟踪，并可能损坏现有的标记表。）
0x9D	使用的池标记不正确	池类型	调用方的地址	当前线程试图用不包含任何字母或数字的标记分配池。 使用此类标记会使跟踪池问题变得困难。
0x41286	预留	预留	从分页池开始的起始偏移量（以页为单位）	当前线程试图在分配过程中释放分页池地址。

_POOL_TYPE 代码在 Ntddk.h 中枚举。 具体而言，0 表示非分页池，1 表示分页池。

原因

当前线程发出了无效的池请求。 通常，这处于错误的 IRQL 级别，或者两次释放相同的内存分配等。

解决方法

在启用内存池选项的情况下激活驱动程序验证程序，以获取有关这些错误的详细信息并查找故障驱动程序。

驱动程序验证程序

驱动程序验证程序是一个实时运行的工具，用于检查驱动程序的行为。 如果它在驱动程序代码的执行中发现错误，则会主动创建一个异常，以允许进一步检查驱动程序代码的该部分。 驱动程序验证程序管理器内置于 Windows 中，可在所有 Windows PC 上使用。 若要启动驱动程序验证程序管理器，请在命令提示下键入“验证程序”。 你可以配置要验证的驱动程序。 验证驱动程序的代码在运行时会增加开销，因此请尝试验证尽可能少的驱动程序。 有关详细信息，请参阅驱动程序验证程序。

Windows 内存诊断

具体而言，对于内存池损坏的情况，请运行 Windows 内存诊断工具，尝试将物理内存作为原因隔离开来。 在控制面板搜索框中键入“内存”，然后选择**诊断计算机内存问题**。运行测试后，使用事件查看器查看系统日志下的结果。 查找“内存诊断结果”条目以查看结果。

错误检查 0xC4：DRIVER_VERIFIER_DETECTED_VIOLATION

项目 • 2024/03/04

DRIVER_VERIFIER_DETECTED_VIOLATION 错误检查的值为 0x000000C4。 这是驱动程序验证程序发现的严重错误的常规错误检查代码。 有关详细信息，请参阅启用驱动程序验证程序时处理错误检查。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_VERIFIER_DETECTED_VIOLATION 参数

参数 1 可确定违规类型。 其余参数的含义随参数 1 的值而变化。 参数值如下表所示。

注意 如果无法查看该表中的所有 5 列，请尝试以下操作：

- 将浏览器窗口扩展到完整大小。
- 将光标放在表中，并使用箭头键向左和向右滚动。

0x00 到 0x70

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x00	当前 IRQL	池类型	字节数	驱动程序请求零字节池分配。
0x01	当前 IRQL	池类型	分配的大小（以字节为单位）	驱动程序尝试使用 IRQL > APC_LEVEL 分配分页内存。
0x02	当前 IRQL	池类型	分配的大小（以字节为单位）	驱动程序尝试使用 IRQL > DISPATCH_LEVEL 分配非分页内存。
0x03				调用方正在尝试分配多个必须成功池的页面，但一个页是该 API 允许的最大值。
0x10	错误地址	0	0	驱动程序试图释放一个没有从分配调用返回的地址。
0x11	当前 IRQL	池类型	池地址	驱动程序尝试使用 IRQL > APC_LEVEL 释放分页池。
0x12	当前 IRQL	池类型	池地址	驱动程序尝试使用 IRQL > DISPATCH_LEVEL 释放非分页池。
0x13 或 0x14	预留	指向池标头的指针	池标头内容	驱动程序尝试释放已释放的内存池。
0x15	计时器条目	池类型	池地址被释放	调用方尝试释放的池包含一个活动计时器。
0x16	预留	池地址	0	驱动程序尝试在错误的地址释放池，或者驱动程序将无效参数传递给内存例程。
0X17	资源条目	池类型	池地址被释放	调用方尝试释放的池包含一个活动 ERESOURCE。
0x30	当前 IRQL	请求的 IRQL	0	驱动程序将无效参数传递给 KeRaiseIrql。 （该参数的值低于当前 IRQL，或者高于 HIGH_LEVEL。这可能是使用未初始化参数的结果。）
0x31	当前 IRQL	请求的 IRQL	0：新 IRQL 错误 1：新 IRQL 在 DPC 例程中无效	驱动程序将无效参数传递给 KeLowerIrql。 （该参数的值高于当前 IRQL，或者高于 HIGH_LEVEL。这可能是使用未初始化参数的结果。）
0x32	当前 IRQL	自旋锁地址	0	驱动程序在除 DISPATCH_LEVEL 以外的 IRQL 上调用 KeReleaseSpinLock。 （这可能是由于自旋锁的双重释放。）

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x33	当前 IRQL	快速互斥体地址	0	驱动程序尝试使用 IRQL > APC_LEVEL 获取快速互斥体。
0x34	当前 IRQL	线程 APC 禁用计数	快速互斥体地址	驱动程序尝试在除 APC_LEVEL 以外的 IRQL 上释放快速互斥体。
0x35	当前 IRQL	自旋锁地址	旧 IRQL	内核释放了 IRQL 不等于 DISPATCH_LEVEL 的自旋锁。
0x36	当前 IRQL	自旋锁编号	旧 IRQL	内核释放了 IRQL 不等于 DISPATCH_LEVEL 的排队自旋锁。
0x37	当前 IRQL	线程 APC 禁用计数	资源	驱动程序试图获取资源，但 APC 未被禁用。
0x38	当前 IRQL	线程 APC 禁用计数	资源	驱动程序试图释放资源，但 APC 未被禁用。
0x39	当前 IRQL	线程 APC 禁用计数	Mutex	驱动程序试图获取一个“不安全”的互斥体，IRQL 在条目上不等于 APC_LEVEL。
0x3A	当前 IRQL	线程 APC 禁用计数	Mutex	驱动程序试图释放一个“不安全”的互斥体，IRQL 在条目上不等于 APC_LEVEL。
0x3B	当前 IRQL	要等待的对象	超时参数	正在以 DISPATCH_LEVEL 或更高级别调用 KeWaitXxx 例程。
0x3C	传递给例程的句柄	对象类型	0	调用 ObReferenceObjectByHandle 的驱动程序的句柄不正确。
0x3D	0	0	错误资源的地址	驱动程序向 ExAcquireResourceExclusive 传递了一个错误（未对齐）资源。
0x3E	0	0	0	对于当前不在临界区的线程，驱动程序调用 KeLeaveCriticalRegion。
0x3F	对象地址	新对象引用计数。 -1：取消引用案例 1：引用案例	0	驱动程序将 ObReferenceObject 应用于引用计数为 0 的对象，或者将 ObDereferenceObject 应用于引用计数为 0 的对象。
0x40	当前 IRQL	自旋锁地址	0	驱动程序使用 IRQL < DISPATCH_LEVEL 调用 KeAcquireSpinLockAtDpcLevel。
0x41	当前 IRQL	自旋锁地址	0	驱动程序使用 IRQL < DISPATCH_LEVEL 调用 KeReleaseSpinLockFromDpcLevel。
0x42	当前 IRQL	自旋锁地址	0	驱动程序使用 IRQL > DISPATCH_LEVEL 调用 KeAcquireSpinLock。
0x51	分配的基址	超出分配的引用的地址	收费字节数	驱动程序在写入超过分配结束后试图释放内存。 只有当驱动程序验证程序的“池跟踪”选项处于活动状态时，才会使用此参数进行错误检查。
0x52	分配的基址	哈希条目	收费字节数	驱动程序在写入超过分配结束后试图释放内存。 只有当驱动程序验证程序的“池跟踪”选项处于活动状态时，才会使用此参数进行错误检查。
0x53	分配的基址	标头	预留	驱动程序在写入超过分配结束后试图释放内存。 只有当驱动程序验证程序的“池跟踪”选项处于活动状态时，才会使用此参数进行错误检查。
0x54	分配的基址	预留	池哈希大小	驱动程序在写入超过分配结束后试图释放内存。 只有当驱动程序验证程序的“池跟踪”选项处于活动状态时，才会使用此参数进行错误检查。
0x59	分配的基址	Listindex	预留	驱动程序在写入超过分配结束后试图释放内存。 只有当驱动程序验证程序的“池跟踪”选项处于活动状态时，才会使用此参数进行错误检查。
0x60	从分页池分配的字节数	从非分页池分配的字节数	未释放的分配总数	驱动程序在没有首先释放池分配的情况下正在卸载。 只有当驱动程序验证程序的“池跟踪”选项处于活动状态时，才会使用此参数进行错误检查。
0x61	从分页池分配的字节数	从非分页池分配的字节数	未释放的分配总数	驱动线程试图在卸载驱动程序时分配池内存。 只有当驱动程序验证程序的“池跟踪”选项处于活动状态时，才会使用此参数进行错误检查。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x62	驱动程序的名称	预留	未释放的分配总数，包括分页池和非分页池	驱动程序在没有首先释放池分配的情况下正在卸载。 只有当驱动程序验证程序的“池跟踪”选项处于活动状态时，才会使用此参数进行错误检查。 输入 !verifier 3 drivername.sys，获取有关导致错误检查的泄露分配的信息。
0x6F	MDL 地址	物理页被锁定	系统中最高的物理页面	对不在 PFN 数据库中的页面调用 MmProbeAndLockPages。 这通常是一个驱动程序调用这个例程来锁定自己的专用双端口 RAM。 这不仅是不必要的，而且还会损坏具有非连续物理 RAM 的计算机上的内存。

0x70 到 0x91

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x70	当前 IRQL	MDL 地址	访问模式	驱动程序使用 IRQL > DISPATCH_LEVEL 调用 MmProbeAndLockPages。
0x71	当前 IRQL	MDL 地址	进程地址	驱动程序使用 IRQL > DISPATCH_LEVEL 调用 MmProbeAndLockProcessPages。
0x72	当前 IRQL	MDL 地址	进程地址	驱动程序使用 IRQL > DISPATCH_LEVEL 调用 MmProbeAndLockSelectedPages。
0x73	当前 IRQL	在 32 位 Windows 中：物理地址的低 32 位 在 64 位 Windows 中：64 位物理地址	字节数	驱动程序使用 IRQL > DISPATCH_LEVEL 调用 MmMapIoSpace。
0x74	当前 IRQL	MDL 地址	访问模式	驱动程序在内核模式下使用 IRQL > DISPATCH_LEVEL 调用 MmMapLockedPages。
0x75	当前 IRQL	MDL 地址	访问模式	驱动程序在用户模式下使用 IRQL > APC_LEVEL 调用 MmMapLockedPages。
0x76	当前 IRQL	MDL 地址	访问模式	驱动程序在内核模式下使用 IRQL > DISPATCH_LEVEL 调用 MmMapLockedPagesSpecifyCache。
0x77	当前 IRQL	MDL 地址	访问模式	驱动程序在用户模式下使用 IRQL > APC_LEVEL 调用 MmMapLockedPagesSpecifyCache。
0x78	当前 IRQL	MDL 地址	0	驱动程序使用 IRQL > DISPATCH_LEVEL 调用 MmUnlockPages。
0x79	当前 IRQL	正在取消映射虚拟地址	MDL 地址	驱动程序在内核模式下使用 IRQL > DISPATCH_LEVEL 调用 MmUnmapLockedPages。
0x7A	当前 IRQL	正在取消映射虚拟地址	MDL 地址	驱动程序在用户模式下使用 IRQL > APC_LEVEL 调用 MmUnmapLockedPages。
0x7B	当前 IRQL	正在取消映射虚拟地址	字节数	驱动程序使用 IRQL > APC_LEVEL 调用 MmUnmapIoSpace。
0x7C	MDL 地址	MDL 标志	0	驱动程序调用 MmUnlockPages，并传递了一个 MDL，其页面从未成功锁定。
0x7D	MDL 地址	MDL 标志	0	驱动程序调用 MmUnlockPages，并传递了一个 MDL，其页面来自非分页池。 （这些永远不应该解锁。）
0x7E	当前 IRQL	DISPATCH_LEVEL	0	驱动程序使用 IRQL > DISPATCH_LEVEL 调用 MmAllocatePagesForMdl、MmAllocatePagesForMdlEx 或 MmFreePagesFromMdl。
0x7F	当前 IRQL	MDL 地址	MDL 标志	驱动程序调用 BuildMdlForNonPagedPool，并传递了一个 MDL，其页面来自分页池。
0x80	当前 IRQL	事件地址	0	驱动程序使用 IRQL > DISPATCH_LEVEL 调用 KeSetEvent。
0x81	MDL 地址	MDL 标志	0	驱动程序调用 MmMapLockedPages。 （应使用 MmMapLockedPagesSpecifyCache，并将 BugCheckOnFailure 参

参数 1	参数 2	参数 3	参数 4	错误消息的原因
				数设置为 FALSE。）
0x82	MDL 地址	MDL 标志	0	驱动程序调用 MmMapLockedPagesSpecifyCache，BugCheckOnFailure 参数等于 TRUE。 （此参数应设置为 FALSE。）
0x83	要映射的物理地址范围的开始	要映射的字节数	未锁定的第一个页帧编号	驱动程序调用 MmMapIoSpace ，但没有锁定 MDL 页面。 在进行此调用之前，由要映射的物理地址范围表示的物理页必须已锁定。
0x85	MDL 地址	要映射的页数	未锁定的第一个页帧编号	驱动程序调用 MmMapLockedPages，但没有锁定 MDL 页面。
0x89	MDL 地址	指向 MDL 中非内存页的指针	MDL 中的非内存页码	MDL 没有标记为“I/O”，但它包含非内存页地址。
0x91	预留	预留	预留	驱动程序使用操作系统不支持的方法切换堆栈。 扩展内核模式堆栈的唯一支持的方法是使用 KeExpandKernelStackAndCallout。

0xA0 到 0x140

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0xA0	指向发出读取或写入请求的 IRP 的指针	下层设备的设备对象	检测到错误的扇区编号	在硬盘上检测到循环冗余检查 (CRC) 错误。 只有当驱动程序验证程序的磁盘完整性检查选项处于活动状态时，才会使用此参数进行错误检查。
0xA1	发出读取或写入请求的 IRP 的副本。（实际 IRP 已完成）。	下层设备的设备对象	检测到错误的扇区编号	在扇区上检测到 CRC 错误（异步）。 只有当驱动程序验证程序的磁盘完整性检查选项处于活动状态时，才会使用此参数进行错误检查。
0xA2	发出读取或写入请求的 IRP，或此 IRP 的副本	下层设备的设备对象	检测到错误的扇区编号	CRCDISK 校验和副本不匹配。 这可能是分页错误。 只有当驱动程序验证程序的磁盘完整性检查选项处于活动状态时，才会使用此参数进行错误检查。
0xB0	MDL 地址	MDL 标志	MDL 标志不正确	驱动程序为 MDL 调用 MmProbeAndLockPages，标志不正确。 例如，驱动程序将 MmBuildMdlForNonPagedPool 创建的 MDL 传递给 MmProbeAndLockPages。
0xB1	MDL 地址	MDL 标志	MDL 标志不正确	驱动程序为 MDL 调用 MmProbeAndLockProcessPages，标志不正确。 例如，驱动程序将 MmBuildMdlForNonPagedPool 创建的 MDL 传递给 MmProbeAndLockProcessPages。
0xB2	MDL 地址	MDL 标志	MDL 标志不正确	驱动程序为 MDL 调用 MmMapLockedPages，标志不正确。 例如，驱动程序传递的 MDL 已映射到系统地址，或者未锁定到 MmMapLockedPages。
0xB3	MDL 地址	MDL 标志	缺少 MDL 标志（至少应有一个）	驱动程序为 MDL 调用 MmMapLockedPages，标志不正确。 例如，驱动程序传递了一个未锁定到 MmMapLockedPages 的 MDL。
0xB4	MDL 地址	MDL 标志	意外的部分 MDL 标志	驱动程序为部分 MDL 调用了 MmUnlockPages。 部分 MDL 由 IoBuildPartialMdl 创建。
0xB5	MDL 地址	MDL 标志	意外的部分 MDL 标志	MmUnmapLockedPages 在部分 MDL 上调用（使用 IoBuildPartialMdl 创建）。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0xB6	MDL 地址	MDL 标志	缺少 MDL 标志	在未映射到系统地址的 MDL 上调用 MmUnmapLockedPages。
0xB7	损坏的物理页数。	第一个损坏的物理页。	最后一个损坏的物理页。	系统 BIOS 在休眠转换期间损坏了低物理内存。
0xB8	MDL 地址	MDL 标志	预留	MDL 描述的页面仍会映射。 驱动程序必须先取消映射页面，然后才能调用 IoFreeMdl。
0xB9	正在取消映射地址。	MDL 地址。	预留	使用错误的用户空间地址调用 MmUnmapLockedPages。
0xC0	IRP 的地址	0	预留	驱动程序调用 IoCallDriver，并禁用中断。
0xC1	驱动程序调度例程的地址	预留	预留	在中断被禁用的情况下返回了驱动程序调度例程。
0xC2	0	0	0	中断被禁用后，驱动程序调用快速 I/O 调度例程。
0xC3	驱动程序快速 I/O 调度例程的地址	预留	预留	在中断被禁用的情况下，返回了驱动程序快速 I/O 调度例程。
0xC5	驱动程序调度例程的地址	当前线程的 APC 禁用计数	在调用驱动程序调度例程之前，线程的 APC 禁用计数	驱动程序调度例程已更改线程的 APC 禁用计数。 每次驱动程序调用 KeEnterCriticalRegion、FsRtlEnterFileSystem 或获取互斥体时，APC 禁用计数都会递减。 每次驱动程序调用 KeLeaveCriticalRegion、KeReleaseMutex 或 FsRtlExitFileSystem 时，APC 禁用计数都会递增。 由于这些调用应始终成对进行，因此当线程退出时，APC 禁用计数应为零。 负值表示驱动程序已禁用 APC 调用，而未重新启用它们。 正值表示情况恰好相反。
0xC6	驱动程序快速 I/O 调度例程的地址	当前线程的 APC 禁用计数	在调用快速 I/O 驱动程序调度例程之前，线程的 APC 禁用计数	驱动程序快速 I/O 调度例程已更改线程的 APC 禁用计数。 每次驱动程序调用 KeEnterCriticalRegion、FsRtlEnterFileSystem 或获取互斥体时，APC 禁用计数都会递减。 每次驱动程序调用 KeLeaveCriticalRegion、KeReleaseMutex 或 FsRtlExitFileSystem 时，APC 禁用计数都会递增。 由于这些调用应始终成对进行，因此当线程退出时，APC 禁用计数应为零。 负值表示驱动程序已禁用 APC 调用，而未重新启用它们。 正值表示情况恰好相反。
0xCA	后备列表的地址	预留	预留	驱动程序已尝试重新初始化后备列表。
0xCB	后备列表的地址	预留	预留	驱动程序已尝试删除未初始化的后备列表。
0xCC	后备列表的地址	池分配的起始地址	池分配的大小	驱动程序已尝试释放包含活动后备列表的池分配。
0xCD	后备列表的地址	调用方指定的块大小	支持的最小块大小	驱动程序试图创建一个分配块大小太小的后备列表。
0xD0	ERESOURCE 结构的地址	预留	预留	驱动程序尝试重新初始化 ERESOURCE 结构。
0xD1	ERESOURCE 结构的地址	预留	预留	驱动程序试图删除未初始化的 ERESOURCE 结构。
0xD2	ERESOURCE 结构的地址	池分配的起始地址	池分配的大小	驱动程序试图释放包含活动 ERESOURCE 结构的池分配。
0xD5	由驱动程序的检查生成版本创建的 IO_REMOVE_LOCK 结构的地址	当前 IoReleaseRemoveLock 标记	预留	当前的 IoReleaseRemoveLock 标记与以前的 IoAcquireRemoveLock 标记不匹配。 如果调用 IoReleaseRemoveLock 的驱动程序不在检查生成中，参数 2 是驱动程序验证程序代表驱动程序创建的影子 IO_REMOVE_LOCK 结构的地址。 在这种情况下，驱动程序使用的 IO_REMOVE_LOCK 结构的地址根本不使用，因为驱动程序验证程序正在替换所有删除锁 API 的锁定地址。 只有当驱动程序验证程序的“I/O 验证”选项处于活动状态时，才会使用此参数进行错误检查。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0xD6	由驱动程序的检查生成版本创建的 IO_REMOVE_LOCK 结构的地址	标记与以前的 IoAcquireRemoveLock 标记不匹配	以前的 IoAcquireRemoveLock 标记	当前的 IoReleaseRemoveLockAndWait 标记与以前的 IoAcquireRemoveLock 标记不匹配。 如果调用 IoReleaseRemoveLock 的驱动程序不在检查生成中，参数 2 是驱动程序验证程序代表驱动程序创建的影子 IO_REMOVE_LOCK 结构的地址。 在这种情况下，驱动程序使用的 IO_REMOVE_LOCK 结构的地址根本不使用，因为驱动程序验证程序正在替换所有删除锁 API 的锁定地址。 只有当驱动程序验证程序的“I/O 验证”选项处于活动状态时，才会使用此参数进行错误检查。
0xD7	驱动程序验证程序内部使用的已检查的构建删除锁结构的地址	由驱动程序指定的删除锁结构的地址	预留	即使删除锁调用 IoReleaseRemoveLockAndWait 后，也无法重新初始化，因为其他线程可能仍在使用该锁（通过调用 IoAcquireRemoveLock）。 驱动程序应在其设备扩展内分配删除锁，并对其进行一次初始化。 锁将与设备扩展一起删除。
0xDA	驱动程序的起始地址	驱动程序内的 WMI 回调地址	预留	试图卸载尚未注销其 WMI 回调函数的驱动程序。
0xDB	设备对象的地址	预留	预留	试图删除未从 WMI 中取消注册的设备对象。
0xDC	预留	预留	预留	指定了无效的 RegHandle 值作为函数 EtwUnregister 的参数。
0xDD	调用 EtwRegister 的地址	卸载驱动程序的起始地址	对于 Windows 8 及更高版本，此参数是 ETW RegHandle 值。	尝试在不调用 EtwUnregister 的情况下卸载驱动程序。
0xDF	同步对象地址	0	0	同步对象位于会话地址空间中。 不允许在会话地址空间中使用同步对象，因为它们可以从另一个会话或没有会话虚拟地址空间的系统线程进行操作。
0xE0	用作参数的用户模式地址	用作参数的地址范围的大小（以字节为单位）	预留	调用了一个操作系统内核函数，该函数指定了一个用户模式地址作为参数。
0xE1	同步对象的地址	预留	预留	发现同步对象的地址无效或可分页。
0xE2	IRP 的地址	IRP 中存在的用户模式地址	预留	发现 Irp->RequestorMode 设置为 KernelMode 的 IRP 的成员之一为用户模式地址。
0xE3	API 调用的地址	在 API 中用作参数的用户模式地址	预留	驱动程序用用户模式地址作为参数调用内核模式 ZwXxx 例程。
0xE4	API 调用的地址	格式错误的 UNICODE_STRING 结构的地址	预留	驱动程序用格式错误的 UNICODE_STRING 结构作为参数调用内核模式 ZwXxx 例程。
0xE5	当前 IRQL	预留	预留	在不正确的 IRQL 上调用了内核 API。
0xE6	进行 Zw API 调用的驱动程序内部的地址	当前 IRQL	特殊内核 APC。	内核 Zw API 未在 IRQL = PASSIVE_LEVEL调用，并且启用了特殊内核 APC。
0xEA	当前 IRQL	线程的 APC 禁用计数	推送锁的地址	在启用 APC时，驱动程序试图获取推送锁。
0xEB	当前 IRQL	线程的 APC 禁用计数	推送锁的地址	在启用 APC时，驱动程序试图释放推送锁。
0xF0	目标缓冲区的地址	源缓冲区的地址	要复制的字节数	名为 memcpy 函数的驱动程序，其中包含重叠的源缓冲区和目标缓冲区。
0xF5	NULL 句柄的地址	对象类型	预留	驱动程序将 NULL 句柄传递给 ObReferenceObjectByHandle。
0xF6	正在引用的句柄值	当前进程的地址	执行错误引用的驱动程序内的地址	驱动程序将用户模式句柄引用为内核模式。
0xF7	调用方指定的句柄值	调用方指定的对象类型	调用方指定的 AccessMode	驱动程序正试图在系统进程的上下文中对内核句柄进行用户模式引用。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0xFA	完成例程地址。	调用完成例程之前的 IRQL 值	调用完成例程后的当前 IRQL 值	IRP 完成例程返回的 IRQL 与调用该例程的 IRQL 不同。
0xFB	完成例程地址	当前线程的 APC 禁用计数	线程调用 IRP 完成例程之前的 APC 禁用计数	线程的 APC 禁用计数由驱动程序的 IRP 完成例程更改。 每次驱动程序调用 KeEnterCriticalRegion、FsRtlEnterFileSystem 或获取互斥体时，APC 禁用计数都会递减。 每次驱动程序调用 KeLeaveCriticalRegion、KeReleaseMutex 或 FsRtlExitFileSystem 时，APC 禁用计数都会递增。 由于这些调用应始终成对进行，因此当线程退出时，APC 禁用计数应为零。 负值表示驱动程序已禁用 APC 调用，而未重新启用它们。 正值表示情况恰好相反。
0xFC	进行错误 API 调用的驱动程序内部的地址。	提供的 ApcContext 值。	预留	使用不支持的 ApcContext 值调用 ZwNotifyChangeKey（从内核模式）。

0x105 到 0x140

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x105	IRP 的地址	0	0	驱动程序使用 ExFreePool 而不是 IoFreeIrp 来释放 IRP。
0x10A	0	0	0	驱动程序试图向空闲进程对池配额收费。
0x10B	0	0	0	驱动程序试图从 DPC 例程对池配额收费。 这不正确，因为当前进程上下文未定义。
0x110	中断服务例程的地址	执行 ISR 之前保存的扩展上下文的地址	扩展上下文的地址在执行 ISR 后保存	驱动程序的中断服务例程 (ISR) 已损坏扩展线程上下文。
0x111	中断服务例程的地址	执行 ISR 之前的 IRQL	执行 ISR 之后的 IRQL	中断服务例程返回已更改的 IRQL。
0x115	负责关闭的线程的地址，该线程可能已死锁。	0	0	驱动程序验证程序检测到系统运行时间超过 20 分钟，并且关闭未完成。
0x11A	当前 IRQL	0	0	驱动程序在 IRQL > APC_LEVEL 上调用 KeEnterCriticalRegion。
0x11B	当前 IRQL	0	0	驱动程序在 IRQL > APC_LEVEL 上调用 KeLeaveCriticalRegion。
0x120	IRQL 值的地址	要等待的对象地址	超时值的地址	线程在 IRQL > DISPATCH_LEVEL 等待。 KeWaitForSingleObject 或 KeWaitForMultipleObjects 的调用方必须在 IRQL <= DISPATCH_LEVEL 运行。
0x121	IRQL 值的地址	要等待的对象地址	超时值的地址	线程在 IRQL 等于 DISPATCH_LEVEL 处等待，并且 Timeout 为 NULL。 KeWaitForSingleObject 或 KeWaitForMultipleObjects 的调用方在 IRQL <= DISPATCH_LEVEL 运行。 如果为 Timeout 提供了 NULL 指针，则调用线程将保持等待状态，直到用信号通知 Object 为止。
0x122	IRQL 值的地址	要等待的对象地址	超时值的地址	线程在 DISPATCH_LEVEL 等待，超时值不等于零 (0)。 如果 Timeout != 0，则 KeWaitForSingleObject 或 KeWaitForMultipleObjects 的调用方必须在 IRQL <= APC_LEVEL 运行。
0x123	要等待的对象地址	0	0	KeWaitForSingleObject 或 KeWaitForMultipleObjects 的调用方将等待指定为 UserMode，但对象在内核堆栈上。
0x130	工作项的地址	0	0	工作项位于会话地址空间中。 不允许在会话地址空间中使用工作项，因为它们可以从另一个会话或没有会话虚拟地址空间的系统线程进行操作。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x131	工作项的地址	0	0	工作项位于可分页内存中。 工作项必须在不可分页内存中，因为内核在 DISPATCH_LEVEL 中使用它们。
0x135	IRP 的地址	从 IoCancelIrp 调用到完成此 IRP 之间允许的毫秒数	0	取消的 IRP 未在预期时间内完成，驱动程序完成已取消的 IRP 花费的时间超过预期。
0x13A	被释放的池块的地址	错误值	错误值的地址	驱动程序已调用 ExFreePool，驱动程序验证程序在用于跟踪池使用情况的内部值之一中检测到错误。
0x13B	被释放的池块的地址	错误值的地址	指向错误内存页的指针的地址	驱动程序已调用 ExFreePool，驱动程序验证程序在用于跟踪池使用情况的内部值之一中检测到错误。
0x13C	被释放的池块的地址	错误值	错误值的地址	驱动程序已调用 ExFreePool，驱动程序验证程序在用于跟踪池使用情况的内部值之一中检测到错误。
0x13D	被释放的池块的地址	错误值的地址	预期的正确值	驱动程序已调用 ExFreePool，驱动程序验证程序在用于跟踪池使用情况的内部值之一中检测到错误。
0x13E	调用方指定的池块地址	驱动程序验证程序跟踪的池块地址	指向驱动程序验证程序跟踪的池块地址的指针	ExFreePool 调用方指定的池块地址与驱动程序验证程序跟踪的地址不同。
0x13F	被释放的池块的地址	正在释放的字节数	指向驱动程序验证程序跟踪的字节数的指针	调用 ExFreePool 时释放的内存字节数与驱动程序验证程序跟踪的字节数不同。
0x140	当前 IRQL	MDL 地址	与此 MDL 关联的虚拟地址	非锁定 MDL 是由可分页内存或可交易内存生成的。
0x141	驱动程序请求分配的最高物理地址	要分配的字节数	0	驱动程序明确请求 4GB 以下的物理内存。

0x1000 到 0x100B - 死锁

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x1000	资源的地址	预留	预留	自死锁：当前线程尝试以递归方式和独占方式获取它只拥有共享的资源。 只有当驱动程序验证程序的“死锁检测”选项处于活动状态时，才会使用此参数进行错误检查。
0x1001	导致死锁的最终原因的资源的地址	预留	预留	死锁：发现锁层次结构冲突。 只有当驱动程序验证程序的“死锁检测”选项处于活动状态时，才会使用此参数进行错误检查。 （使用 !deadlock 扩展了解详细信息。）
0x1002	资源的地址	预留	预留	未初始化的资源：在未首先初始化的情况下获取了资源。 只有当驱动程序验证程序的“死锁检测”选项处于活动状态时，才会使用此参数进行错误检查。
0x1003	死锁释放的资源的地址	应首先释放的资源的地址	预留	意外释放：资源的释放顺序不正确。 只有当驱动程序验证程序的“死锁检测”选项处于活动状态时，才会使用此参数进行错误检查。
0x1004	资源的地址	获取资源的线程的地址	当前线程的地址	意外线程：错误的线程释放资源。 只有当驱动程序验证程序的“死锁检测”选项处于活动状态时，才会使用此参数进行错误检查。
0x1005	资源的地址	预留	预留	多次初始化：对资源进行多次初始化。 只有当驱动程序验证程序的“死锁检测”选项处于活动状态时，才会使用此参数进行错误检查。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x1007	资源的地址	预留	预留	未获取的资源：资源在被获取之前就被释放了。 只有当驱动程序验证程序的"死锁检测"选项处于活动状态时，才会使用此参数进行错误检查。
0x1008	锁定地址	预留	预留	驱动程序试图通过使用与此锁类型不匹配的 API 获取锁。
0x1009	锁定地址	预留	预留	驱动程序试图通过使用与此锁类型不匹配的 API 释放锁。
0x100A	所有者线程地址	预留	终止的线程拥有锁。	
0x100B	锁定地址	所有者线程地址	预留	已删除的锁仍归线程所有。
0x1010	向其发出写入 IRP 的设备对象。	IRP 的地址。	MDL 描述的缓冲区的系统空间虚拟地址。	修改了 Write Irp 的不变 MDL 缓冲区内容。
0x1011	向其发出写入 IRP 的设备对象。	IRP 的地址。	MDL 描述的缓冲区的系统空间虚拟地址。	Read Irp 的不变 MDL 缓冲区内容在调度期间被修改，或者缓冲区由伪页支持。
0x1012	指向描述冲突的字符串的指针。	此损坏涉及的数据（如果未使用，则为 0）。	此损坏涉及的数据（如果未使用，则为 0）。	验证程序扩展状态存储检测到损坏。
0x1013	指向驱动程序对象的指针。	指向捕获的原始 I/O 回调的指针。	预留（未使用）。	验证程序在捕获的原始 I/O 回调中检测到内部损坏。

0x2000 到 0x2005 - 代码完整性问题

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x2000	检测到错误的驱动程序代码中的地址。	池类型。	池标记（如果提供）。	代码完整性问题：调用方指定了可执行池类型。 （预期：NonPagedPoolNx）
0x2001	检测到错误的驱动程序代码中的地址。	页面保护 (WIN32_PROTECTION_MASK)。	0	代码完整性问题：调用方指定了可执行页面保护。 （预期：清除 PAGE_EXECUTE* 位）
0x2002	检测到错误的驱动程序代码中的地址。	页面优先级（MM_PAGE_PRIORITY 逻辑上与 MdlMapping*为 OR）。	0	代码完整性问题：调用方指定了可执行的 MDL 映射。 （预期： MdlMappingNoExecute）
0x2003	映像文件名（Unicode 字符串）。	节标头的地址。	节名称（UTF-8 编码字符串）。	代码完整性问题：映像包含可执行和可写部分。
0x2004	映像文件名（Unicode 字符串）。	节标头的地址。	节名称（UTF-8 编码字符串）。	代码完整性问题：映像包含未对齐页面的部分。
0x2005	映像文件名（Unicode 字符串）。	IAT 目录。	节名称（UTF-8 编码字符串）。	代码完整性问题：映像包含位于可执行部分的 IAT。

0xA001 到 0xA00D - VM 交换机问题

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0xA001	指向 NetBufferList 对象的指针	指向虚拟交换机对象的指针（如果为非 Null）	预留（未使用）	VM 交换机：必须设置调用方提供的 NetBufferList 的 SourceHandle。 请参阅 AllocateNetBufferListForwardingContext 例程。
0xA002	指向 NetBufferList 对象的指针	指向虚拟交换机对象的指针（如果为非 Null）。	预留（未使用）	VM 交换机：调用方提供的 NetBufferList 的转发详细信息不是零。 请参阅 AllocateNetBufferListForwardingContext 例程。
0xA003	指向 NetBufferList 对象的指针	指向虚拟交换机对象的指针（如果为非 Null）。	预留（未使用）	VM 交换机：调用方提供的 NetBufferList 的数据包标头或路由上下文为 NULL。 请参阅可扩展交换机数据路径的分组管理指南。
0xA004	无效端口的 ID	NIC 索引	指向虚拟交换机对象的指针（如果为非 Null）。	VM 交换机：调用方指定了无效的端口和 NIC 索引组合。 请参阅 Hyper-V 可扩展交换机端口和网络适配器状态。
0xA005	指向 NetBufferList 对象的指针	指向目标列表的指针。	指向虚拟交换机对象的指针（如果为非 Null）。	VM 交换机：调用方提供了无效的目标。 请参阅 AddNetBufferListDestination 和 UpdateNetBufferListDestinations。
0xA006	指向 NetBufferList 对象的指针	指向虚拟交换机对象的指针（如果为非 Null）。	预留（未使用）	VM 交换机：调用方提供了无效的源 NIC 或端口对象。 请参阅 Hyper-V 可扩展交换机端口和网络适配器状态。
0xA007	指向 NetBufferList 对象的指针	指向虚拟交换机对象的指针（如果为非 Null）。	预留（未使用）	VM 交换机：调用方提供了无效的目标列表。 请参阅 AddNetBufferListDestination 和 UpdateNetBufferListDestinations。
0xA008	父 NIC 对象	NIC 索引	指向虚拟交换机对象的指针（如果为非 Null）。	VM 交换机：在不允许时尝试引用 NIC。 请参阅 Hyper-V 可扩展交换机端口和网络适配器状态。
0xA009	引用的端口	指向虚拟交换机对象的指针（如果为非 Null）	预留（未使用）	VM 交换机：尝试在不允许时引用端口。 请参阅 Hyper-V 可扩展交换机端口和网络适配器状态。
0xA00A	指向 NetBufferList 对象的指针	ContextTypeInfo 对象	预留（未使用）	VM 交换机：已设置故障上下文。 请参阅 SetNetBufferListSwitchContext。
0xA00B	指向 NetBufferList 对象的指针	NDIS_SWITCH_REPORT_FILTERED_NBL_FLAGS_*	指向虚拟交换机对象的指针（如果为非 Null）	VM 交换机：为已删除的 NetBufferList 提供的方向无效。 请参阅 ReportFilteredNetBufferLists。
0xA00C	指向 NetBufferList 对象的指针	发送标志值	指向虚拟交换机对象的指针（如果为非 Null）	VM 交换机：当设置 NDIS_SEND_FLAGS_SWITCH_SINGLE_SOURCE 标志时，NetBufferList 链具有多个源端口。 请参阅 Hyper-V 可扩展交换机发送和接收标志。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0xA00D	指向 NetBufferList 对象的指针	指向虚拟交换机上下文的指针	指向虚拟交换机对象的指针（如果为非 Null）	VM 交换机：设置 NDIS_RECEIVE_FLAGS_SWITCH_DESTINATION_GROUP 标志时，链中的一个或多个 NetBufferLists 目标无效。 请参阅 Hyper-V 可扩展交换机发送和接收标志。
0xA00E	指向 NetBufferLists 对象的指针。	指向虚拟交换机上下文的指针。	指向虚拟交换机对象的指针（如果为非 Null）。	VM 交换机：设置 VMS_NBL_ROUTING_CONTEXT_FLAG_NO_WNV_PROCESSING 标志时尝试通过 WNV 完成 NetBufferList。

0x00020002 到 0x00020022 - DDI 合规性规则冲突

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x00020002	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlApcLte。 该规则指定只有当 IRQL <= APC_LEVEL 时，驱动程序才能调用 ObGetObjectSecurity 和 ObReleaseObjectSecurity。
0x00020003	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlDispatch。 IrqlDispatch 规则指定，仅当 IRQL = DISPATCH_LEVEL 时，驱动程序才必须调用某些例程
0x00020004	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlExAllocatePool。 IrqlExAllocatePool 规则指定驱动程序仅在 IRQL<=DISPATCH_LEVEL 时调用 ExAllocatePoolWithTag 和 ExAllocatePoolWithTagPriority。
0x00020005	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlExApcLte1。 IrqlExApcLte1 规则指定驱动程序仅在 IRQL <= APC_LEVEL 时才调用 ExAcquireFastMutex 和 ExTryToAcquireFastMutex。
0x00020006	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlExApcLte2。 IrqlExApcLte2 规则指定驱动程序仅在 IRQL <= APC_LEVEL 时调用某些例程。
0x00020007	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlExApcLte3。 IrqlExApcLte3 规则指定，仅当 IRQL <= APC_LEVEL 时，驱动程序才必须调用某些执行支持例程。
0x00020008	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlExPassive。 IrqlExPassive 规则指定，仅当 IRQL = PASSIVE_LEVEL 时，驱动程序才必须调用某些执行支持例程。
0x00020009	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlIoApcLte。 IrqlIoApcLte 规则指定，仅当 IRQL <= APC_LEVEL 时，驱动程序才必须调用某些 I/O 管理器例程。
0x0002000A	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlIoPassive1。 IrqlIoPassive1 规则指定，仅当 IRQL = PASSIVE_LEVEL 时，驱动程序才必须调用某些 I/O 管理器例程。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x0002000B	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlIoPassive2。 IrqlIoPassive2 规则指定，仅当 IRQL = PASSIVE_LEVEL 时，驱动程序才必须调用某些 I/O 管理器例程。
0x0002000C	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlIoPassive3。 IrqlIoPassive3 规则指定，仅当 IRQL = PASSIVE_LEVEL 时，驱动程序才必须调用某些 I/O 管理器例程。
0x0002000D	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlIoPassive4。 IrqlIoPassive4 规则指定，仅当 IRQL = PASSIVE_LEVEL 时，驱动程序才必须调用某些 I/O 管理器例程。
0x0002000E	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlIoPassive5。 IrqlIoPassive5 规则指定，仅当 IRQL = PASSIVE_LEVEL 时，驱动程序才必须调用某些 I/O 管理器例程。
0x0002000F	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlKeApcLte1。 IrqlKeApcLte1 规则指定，仅当 IRQL <= APC_LEVEL 时，驱动程序才必须调用某些内核例程。
0x00020010	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlKeApcLte2。 IrqlKeApcLte2 规则指定，仅当 IRQL <= APC_LEVEL 时，驱动程序才必须调用某些内核例程。
0x00020011	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlKeDispatchLte。 IrqlKeDispatchLte 规则指定，仅当 IRQL <= DISPATCH_LEVEL 时，驱动程序才必须调用某些内核例程。
0x00020015	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlKeReleaseSpinLock。 IrqlKeReleaseSpinLock 规则指定，仅当 IRQL = DISPATCH_LEVEL 时，驱动程序才必须调用 KeReleaseSpinLock。
0x00020016	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlKeSetEvent。 IrqlKeSetEvent 规则指定，当 Wait 设置为 FALSE 时，仅在 IRQL <= DISPATCH_LEVEL 处调用 KeSetEvent 例程；当 Wait 设置为 TRUE 时，仅在 IRQL <= APC_LEVEL 处调用例程。
0x00020019	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlMmApcLte。 IrqlMmApcLte 规则指定，仅当 IRQL <= APC_LEVEL 时，驱动程序才必须调用某些内存管理器例程。
0x0002001A	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlMmDispatch。 IrqlMmDispatch 规则指定，仅当 IRQL = DISPATCH_LEVEL 时，驱动程序才必须调用 MmFreeContiguousMemory。
0x0002001B	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlObPassive。 IrqlObPassive 规则指定，仅当 IRQL = PASSIVE_LEVEL 时，驱动程序才必须调用 ObReferenceObjectByHandle。
0x0002001C	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlPsPassive。 IrqlPsPassive 规则指定，仅当 IRQL = PASSIVE_LEVEL 时，驱动程序才必须调用某些进程和线程管理器例程。
0x0002001D	指向描述违反规则条件	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 IrqlReturn。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
	的字符串的指针。			
0x0002001E	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlRtlPassive。 IrqlRtlPassive 规则指定，仅当 IRQL = PASSIVE_LEVEL 时，驱动程序才必须调用 RtlDeleteRegistryValue。
0x0002001F	指向描述违反规则条件的字符串的指针。	指向规则状态变量的可选指针。	预留	驱动程序违反了 DDI 合规性规则 IrqlZwPassive。 IrqlZwPassive 规则指定，仅当 IRQL = PASSIVE_LEVEL，驱动程序才必须调用 ZwClose。
0x00020022	指向描述违反规则条件的字符串的指针。	预留（未使用）	预留（未使用）	驱动程序违反了 DDI 合规性规则 IrqlIoDispatch。
0x00020023	指向描述违反规则条件的字符串的指针。	预留（未使用）。	预留（未使用）。	驱动程序违反了 DDI 合规性规则 IrqlIoRtlZwPassive。 IrqlIoRtlZwPassive 规则指定，仅当 IRQL = PASSIVE_LEVEL，驱动程序才调用规则中列出的 DDI。
0x00020024	指向描述违反规则条件的字符串的指针。	预留（未使用）。	预留（未使用）。	驱动程序违反了 DDI 合规性规则 IrqlNtifsApcPassive。 IrqlNtifsApcPassive 规则指定，仅当驱动程序在 IRQL = PASSIVE_LEVEL 或 IRQL <= APC_LEVEL 执行时，才调用规则中列出的 DDI。
0x00020025	指向描述违反规则条件的字符串的指针。	预留（未使用）。	预留（未使用）。	驱动程序违反了 Microsoft 内部 DDI 合规性规则 IrqlKeMore。

0x00040003 到 0x00043006 - DDI 合规性规则冲突

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x00040003	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 CriticalRegions。
0x00040006	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 QueuedSpinLock。
0x00040007	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 QueuedSpinLockRelease。
0x00040009	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 SpinLock。
0x0004000A	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 SpinlockRelease。
0x0004000E	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 GuardedRegions。
0x0004100B	指向描述违反规则条件的字符串的指针。	预留（未使用）	预留（未使用）	驱动程序违反了 DDI 合规性规则 RequestedPowerIrp。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x0004100F	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 IoSetCompletionExCompleteIrp。
0x00043006	指向描述违反规则条件的字符串的指针。	预留（未使用）	预留（未使用）	驱动程序违反了 DDI 合规性规则 PnpRemove。

0x00081001 到 0x00082005 - AVStream 驱动程序合规性规则冲突

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x00081001	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 KsDeviceMutex。
0x00081002	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 KsStreamPointerClone。
0x00081003	指向描述违反规则条件的字符串的指针。	预留（未使用）	预留（未使用）	驱动程序违反了 DDI 合规性规则 KsStreamPointerLock。
0x00081004	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 KsStreamPointerUnlock。
0x00081005	指向描述违反规则条件的字符串的指针。	预留（未使用）	预留（未使用）	驱动程序违反了 DDI 合规性规则 KsCallbackReturn。
0x00081006	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 KsIrqlDeviceCallbacks。
0x00081007	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 KsIrqlFilterCallbacks。
0x00081008	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 KsIrqlPinCallbacks。
0x00081009	指向描述违反规则条件的字符串的指针。	预留（未使用）	预留（未使用）	驱动程序违反了 DDI 合规性规则 KsIrqlDDIs。
0x0008100A	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 KsFilterMutex。
0x0008100B	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 KsProcessingMutex。
0x0008100C	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 KsInvalidStreamPointer。
0x00082001	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 KsTimedPinSetDeviceState。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x00082002	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 KsTimedDeviceCallbacks。
0x00082003	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 KsTimedFilterCallbacks。
0x00082004	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 KsTimedPinCallbacks。
0x00082005	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 KsTimedProcessingMutex。

0x00091001 到 0x0009400C - NDIS DDI 合规性规则冲突

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x00091001	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 NdisOidComplete。
0x00091002	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 NdisOidDoubleComplete。
0x0009100E	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 DDI 合规性规则 NdisOidDoubleRequest。
0x00092003	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 NDIS/WIFI 验证规则 NdisTimedOidComplete。
0x0009200D	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 NDIS/WIFI 验证规则 NdisTimedDataSend。
0x0009200F	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 NDIS/WIFI 验证规则 NdisTimedDataHang。
0x00092010	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 NDIS/WIFI 验证规则 NdisFilterTimedPauseComplete。
0x00092011	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 NDIS/WIFI 验证规则 NdisFilterTimedDataSend。
0x00092012	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 NDIS/WIFI 验证规则 NdisFilterTimedDataReceive。
0x00093004	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 NDIS/WIFI 验证规则 WlanAssociation。
0x00093005	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 NDIS/WIFI 验证规则 WlanConnectionRoaming。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x00093006	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 NDIS/WIFI 验证规则 WlanDisassociation。
0x00093101	指向描述违反规则条件的字符串的指针。	预留（未使用）	预留（未使用）	驱动程序违反了 NDIS/WIFI 验证规则 WlanAssert。
0x00094007	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 NDIS/WIFI 验证规则 WlanTimedAssociation。
0x00094008	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 NDIS/WIFI 验证规则 WlanTimedConnectionRoaming。
0x00094009	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 NDIS/WIFI 验证规则 WlanTimedConnectRequest。
0x0009400B	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 NDIS/WIFI 验证规则 WlanTimedLinkQuality。
0x0009400C	指向描述违反规则条件的字符串的指针。	内部规则状态的地址（第二个参数为 !ruleinfo）。	补充状态的地址（第三个参数为 !ruleinfo）。	驱动程序违反了 NDIS/WIFI 验证规则 WlanTimedScan。

原因

有关原因的说明，请参阅“参数”章节中每个代码的说明。 可以使用 **!analyze -v** 扩展获取更多信息。

解决方法

只有当驱动程序验证程序被指示监视一个或多个驱动程序时，才能进行此错误检查。 如果不打算使用驱动程序验证程序，则应将其停用。 还可以考虑删除导致此问题的驱动程序。

如果你是驱动程序编写者，请使用通过此错误检查获得的信息来修复代码中的错误。

有关驱动程序验证程序的完整详细信息，请参阅驱动程序验证程序。

注解

_POOL_TYPE 代码在 Ntddk.h 中枚举。 其中，**0**（零）表示非分页池，**1**（一）表示分页池。

*（Windows 8 及更高版本的 Windows）*如果 DDI 合规性检查导致错误检查，请在驱动程序源代码上运行静态驱动程序验证程序，并指定导致错误检查的 DDI 合规性规则（由参数 1 值标识）。 静态驱动程序验证程序可以帮助在源代码中找到问题的原因。

另请参阅

在启用驱动程序验证程序的情况下处理错误检查

错误检查 0xC5：DRIVER_CORRUPTED_EXPOOL

项目 • 2024/03/11

DRIVER_CORRUPTED_EXPOOL 错误检查的值为 0x000000C5。 这表示系统试图访问进程 IRQL 过高的无效内存。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_CORRUPTED_EXPOOL 参数

展开表

参数	说明
1	引用的内存
2	引用时的 IRQL
3	**0**: 读取 **1**: 写入
4	引用内存的地址

原因

当 IRQL 过高时，内核试图访问可分页内存（或者可能完全无效的内存）。 几乎可以肯定，这个问题的最终原因是驱动程序损坏了系统池。

在大多数情况下，如果驱动程序破坏了较小的分配（小于 PAGE_SIZE），则会导致此错误检查。 较大的分配会导致**错误检查 0xD0** (DRIVER_CORRUPTED_MMPOOL)。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。 如果最近安装了任何新软件，请检查是否安装正确。 请访问制造商的网站，查看更新的驱动程序。

若要调试此错误，请使用驱动程序验证程序的特殊池选项。 如果无法显示导致错误的驱动程序，请使用 Global Flags 实用工具启用特殊的逐池标记。

有关特殊池的信息，请参阅 Windows 驱动程序工具包的驱动程序验证程序部分。

错误检查 0xC6：DRIVER_CAUGHT_MODIFYING_FREED_POOL

项目 • 2024/03/04

DRIVER_CAUGHT_MODIFYING_FREED_POOL 错误检查的值为 0x000000C6。 这表示驱动程序试图访问已释放的内存池。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_CAUGHT_MODIFYING_FREED_POOL 参数

展开表

参数	说明
1	引用的内存
2	**0**: 读取 **1**: 写入
3	**0**: 内核模式 **1**: 用户模式
4	Reserved

注解

故障组件将显示在当前内核堆栈中。 应更换或调试此驱动程序。

错误检查 0xC7：TIMER_OR_DPC_INVALID

项目 • 2024/03/04

TIMER_OR_DPC_INVALID 错误检查的值为 0x000000C7。 如果在内存中不允许使用内核计时器或延迟的过程调用 (DPC)，就会发生此错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

TIMER_OR_DPC_INVALID 参数

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x0	计时器对象的地址	正在检查的内存范围的起始位置	正在检查的内存范围的结束	在不允许使用计时器对象的内存块中找到计时器对象。 。
0x1	DPC 对象的地址	正在检查的内存范围的起始位置	正在检查的内存范围的结束	在不允许使用 DPC 对象的内存块中找到 DPC 对象。
0x2	DPC 例程的地址	正在检查的内存范围的起始位置	正在检查的内存范围的结束	在不允许使用 DPC 对象的内存块中找到 DPC 例程。
0x3	DPC 对象的地址	处理器编号	系统中的处理器数	DPC 对象的处理器编号不正确。
0x4	DPC 例程的地址	内核调用 DPC 例程之前，线程的 APC 禁用计数	调用 DPC 例程之后，线程的 APC 禁用计数	在执行 DPC 例程期间，线程的 APC 禁用计数发生了更改。 每次驱动程序调用 **KeEnterCriticalRegion**、**FsRtlEnterFileSystem** 或获取互斥体时，APC 禁用计数都会递减。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
				每次驱动程序调用 **KeLeaveCriticalRegion**、**KeReleaseMutex** 或 **FsRtlExitFileSystem** 时，APC 禁用计数都会递增。
0x5	DPC 例程的地址	内核调用 DPC 例程之前，线程的 APC 禁用计数	调用 DPC 例程之后，线程的 APC 禁用计数	线程的 APC 禁用计数在执行计时器 DPC 例程期间发生了更改。 每次驱动程序调用 **KeEnterCriticalRegion**、**FsRtlEnterFileSystem** 或获取互斥体时，APC 禁用计数都会递减。 每次驱动程序调用 **KeLeaveCriticalRegion**、**KeReleaseMutex** 或 **FsRtlExitFileSystem** 时，APC 禁用计数都会递增。

原因

这种情况通常是由驱动程序在释放其所在的内存之前未能取消计时器或 DPC 引起的。

解决方法

如果你是驱动程序编写者，请使用通过此错误检查获得的信息来修复代码中的错误。

如果你是系统管理员，如果问题仍然存在，则应卸载驱动程序。

错误检查 0xC8：IRQL_UNEXPECTED_VALUE

项目 • 2024/03/04

IRQL_UNEXPECTED_VALUE 错误检查的值为 0x000000C8。 这表示处理器的 IRQL 目前不应是什么。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

IRQL_UNEXPECTED_VALUE 参数

展开表

参数	说明
1	以下位计算的值： （当前 IRQL << 16）|（预期 IRQL << 8）| UniqueValue
2 - 取决于 UniqueValue	如果 UniqueValue 为 0 或 1：APC->KernelRoutine。 如果 UniqueValue 为 2：标注例程 如果 UniqueValue 为 3：中断的 ServiceRoutine 如果 UniqueValue 为 0xfe：1（如果禁用 APC）
3 - 取决于 UniqueValue	如果 UniqueValue 为 0 或 1：APC 如果 UniqueValue 为 2：标注的参数 如果 UniqueValue 为 3：KINTERRUPT
4 - 取决于 UniqueValue	如果 UniqueValue 为 0 或 1：APC->NormalRoutine

原因

此错误通常是由设备驱动程序或其他较低级别的程序在一段时间内更改了 IRQL，但在该段时间结束时没有恢复原始 IRQL 引起的。 例如，例程可能获取了自旋锁，但未能将其释放。

!analyze 调试扩展显示有关错误检查的信息，并非常有助于确定根本原因。

另请参阅

使用 Windows 调试器 (WinDbg) 进行故障转储分析

使用 WinDbg 分析内核模式转储文件

错误检查 0xC9：DRIVER_VERIFIER_IOMANAGER_VIOLATION

项目 • 2024/03/11

DRIVER_VERIFIER_IOMANAGER_VIOLATION 错误检查的值为 0x000000C9。 这是所有驱动程序验证程序 **I/O 验证**违规的错误检查代码。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_VERIFIER_IOMANAGER_VIOLATION 参数

当驱动程序验证程序处于活动状态并且选择了 **I/O 验证**时，各种 I/O 冲突将导致发出此错误检查。 参数 1 可确定违规类型。

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x01	正在释放的 IRP 地址	0	0	驱动程序试图释放类型不是 IO_TYPE_IRP 的对象。
0x02	正在释放的 IRP 地址	0	0	驱动程序试图释放仍与线程关联的 IRP。
0x03	要发送的 IRP 地址	0	0	驱动程序传递给 IoCallDriver 的 IRP 类型不等于 IRP_TYPE。
0x04	设备对象的地址	0	0	驱动程序向 IoCallDriver 传递了一个无效的设备对象。
0x05	与有问题的驱动程序关联的设备对象的地址	IoCallDriver 之前的 IRQL	IoCallDriver 之后的 IRQL	在调用驱动程序调度例程的过程中，IRQL 发生了变化。
0x06	IRP 状态	IRP 的地址正在填写中	0	驱动程序调用 IoCompleteRequest，状态标记

参数 1	参数 2	参数 3	参数 4	错误消息的含义 -1）。
0x07	取消例程的地址	IRP 的地址正在填写中	0	驱动程序调用 **IoCompleteRequest**，而其取消例程仍在设置中。
0x08	设备对象的地址	IRP 主要函数代码	异常状态代码	驱动程序向 **IoBuildAsynchronousFsdRequest** 传递了一个无效的缓冲区。
0x09	设备对象的地址	I/O 控制代码	异常状态代码	驱动程序向 **IoBuildDeviceIoControlRequest** 传递了一个无效的缓冲区。
0x10	当前 IRQL	预留	预留	在 DISPATCH_LEVEL 之上调用了 IoCallDriver。
0x11	驱动程序快速 I/O 调度例程地址	调用驱动程序调度例程之前的 IRQL	当前 IRQL	在 DISPATCH_LEVEL 之上调用了 IoCallDriver。
0x12	驱动程序调度例程地址	调用驱动程序调度例程之前的 IRQL	当前 IRQL	在 DISPATCH_LEVEL 之上调用了 IoCallDriver。
0x0A	设备对象的地址	0	0	驱动程序向 **IoInitializeTimer** 传递了一个具有已初始化计时器的设备对象。
0x0C	I/O 状态块的地址	0	0	驱动程序将 I/O 状态块传递给 IRP，但该块是在已经展开超过该点的堆栈上分配的。
0x0D	用户事件对象的地址	0	0	驱动程序向 IRP 传递了一个用户事件，但该事件是在已经展开超过该点的堆栈上分配的。
0x0E	当前 IRQL	IRP 的地址	0	驱动程序使用 IRQL > DISPATCH_LEVEL 调用 **IoCompleteRequest**。
0x0F	IRP 发送到的设备对象的地址	指向 IRP 的指针	指向文件对象的指针	驱动程序发送了一个创建请求，其中包含一个已关闭或已取消打开的文件对象。

除了上表中提到的错误外，还有许多 **I/O 验证**错误会导致驱动程序验证程序停止系统，但这些错误实际上不是错误检查。

这些错误导致消息显示在蓝屏、崩溃转储文件中和内核调试器中。 这些消息在每个位置的显示方式都不同。 当出现这些错误时，十六进制错误检查代码 0xC9 和错误检查字符串

DRIVER_VERIFIER_IOMANAGER_VIOLATION 不会出现在蓝屏或调试器中，尽管它们会出现在崩溃转储文件中。

在蓝屏上，将显示以下数据：

- 消息 **IO 系统验证错误。**
- 消息 **WDM DRIVER ERROR***XXX*，其中 *XXX* 是表示特定错误的十六进制代码。 （有关 I/O 错误代码及其含义的列表，请参阅下表。）
- 导致错误的驱动程序的名称。
- 通常，驱动程序代码中检测到错误的地址（参数 2）。

如果已启用内核模式崩溃转储，则崩溃转储文件中将显示以下信息：

- 消息**错误检查 0xC9 (DRIVER_VERIFIER_IOMANAGER_VIOLATION)。**
- 十六进制 I/O 错误代码。 （有关 I/O 错误代码及其含义的列表，请参阅下表。）
- 通常，驱动程序代码中检测到错误的地址（参数 2）。

如果内核调试器附加到导致此冲突的系统，则会将以下信息发送到调试器：

- 消息 **WDM DRIVER ERROR** 以及对错误严重性的评估。
- 导致错误的驱动程序的名称。
- 说明此错误的原因的描述性字符串。 通常传递其他信息，例如指向 IRP 的指针或指向设备对象或 IRQL 信息的指针。 （有关这些描述性字符串的列表以及指定了哪些附加信息，请参阅下表。）
- 要求进一步操作的查询。 可能的响应为 **b**（中断）、**i**（忽略）、**z**（清除）、**r**（移除）或 **d**（禁用）。 指示操作系统继续，可以让你看到如果没有发生此错误，“接下来”会发生什么。 当然，这通常会导致其他错误检查。 “zap”选项实际上会删除导致发现此错误的断点。

注意 不能以这种方式忽略任何其他错误检查。 只有此类 **I/O 验证** 错误可以忽略，但是这些错误也只能在附加了内核调试器的情况下被忽略。

下表列出了可能出现的 **I/O 验证**错误。

展开表

I/O 错误代码	Severity	错误消息的原因
0x200	未知	此代码涵盖所有未知 **I/O 验证**错误。

I/O 错误代码	Severity	错误消息的原因
0x201	严重错误	一个设备正在删除它自己，而驱动程序堆栈中它下面还有另一个设备。 这可能是因为调用方忘记先调用 **IoDetachDevice**，或者较低的驱动程序可能错误地删除了自己。 参数2 - 检测到错误的驱动程序代码中的地址。
0x202	严重错误	驱动程序试图从未附加到任何内容的设备对象分离。 如果在同一设备对象上调用了两次分离，则可能会发生这种情况。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - 设备对象地址。
0x203	严重错误	驱动程序调用了 **IoCallDriver**，但没有将 IRP 中的取消例程设置为 **NULL**。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x204	严重错误	调用方已将 **NULL** 作为设备对象传入。 这是严重错误。 参数 2 - 保留 参数 3 - 保留
0x205	严重错误	调用方正在转发当前在其下面排队的 IRP。 此驱动程序中返回 STATUS_PENDING 的代码处理 IRP 似乎已损坏。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x206	严重错误	调用方错误地转发了 IRP（控制字段未归零）。 驱动程序应使用 **IoCopyCurrentIrpStackLocationToNext** 或 **IoSkipCurrentIrpStackLocation**。 参数 2 - 保留

I/O 错误代码	Severity	错误消息的原因
		参数 3 - 保留
0x207	严重错误	调用方手动复制了堆栈，并无意中复制了上层的完成例程。 驱动程序应使用 **IoCopyCurrentIrpStackLocationToNext**。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x208	严重错误	此 IRP 即将用完堆栈位置。 可能有人从另一个堆栈转发了此 IRP。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x209	严重错误	调用方正在完成当前在其下面排队的 IRP。 此驱动程序中返回 STATUS_PENDING 的代码处理 IRP 似乎已损坏。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x20A	严重错误	**IoFreeIrp** 的调用方正在释放仍在使用的 IRP。 参数 2 - 保留 参数 3 - 保留
0x20B	严重错误	**IoFreeIrp** 的调用方正在释放仍在使用的 IRP。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x20C	严重错误	**IoFreeIrp** 的调用方正在释放仍在针对线程排队的 IRP。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。

I/O 错误代码	Severity	错误消息的原因
0x20D	严重错误	**IoInitializeIrp** 的调用方传递了一个由 **IoAllocateIrp** 分配的 IRP。 这既是非法行为，也是不必要的行为，并已导致配额泄漏。 如果回收了此 IRP，请查看 **IoReuseIrp** 的文档。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x20E	非严重错误	PNP IRP 的状态无效。 （任何 PNP IRP 都必须将其状态初始化为 STATUS_NOT_SUPPORTED。） 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x20F	非严重错误	Power IRP 的状态无效。 （任何 Power IRP 都必须将其状态初始化为 STATUS_NOT_SUPPORTED。） 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x210	非严重错误	WMI IRP 的状态无效。 （任何 WMI IRP 都必须将其状态初始化为 STATUS_NOT_SUPPORTED。） 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x211	非严重错误	调用方在跳过堆栈中的设备对象时转发了一个 IRP。 调用方可能会将 IRP 发送到 PDO，而不是发送到 **IoAttachDeviceToDeviceStack**返回的设备。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x212	非严重错误	调用方已经销毁了 IRP 的堆栈，或者没有正确地复制 IRP 的堆栈。

I/O 错误代码	Severity	错误消息的原因
		参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x213	非严重错误	调用方更改了它不理解的 IRP 的状态字段。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x214	非严重错误	调用方更改了它不理解的 IRP 的信息字段。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x215	非严重错误	IRP_MJ_PNP 的非成功非 STATUS_NOT_SUPPORTED IRP 状态正在向下传递堆栈。 必须完成失败的 PNP IRP。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x216	非严重错误	以前设置的 IRP_MJ_PNP 状态已转换为 STATUS_NOT_SUPPORTE。 此故障状态保留供操作系统使用。 驱动程序不能使具有此值的 PnP IRP 失败。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x217	非严重错误	驱动程序未处理所需的 IRP。 驱动程序必须更新 IRP 的状态，以指示是否已对其进行了处理。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。

I/O 错误代码	Severity	错误消息的原因
0x218	非严重错误	驱动程序已响应为堆栈中其他位置的其他设备对象保留的 IRP。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x219	非严重错误	IRP_MJ_POWER 的非成功非 STATUS_NOT_SUPPORTED IRP 状态正在向下传递堆栈。 必须完成失败的 POWER IRP。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x21A	非严重错误	以前设置 IRP_MJ_POWER 状态已转换为 STATUS_NOT_SUPPORTED。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x21B	非严重错误	驱动程序已返回可疑状态。 这可能是由于驱动程序中存在未初始化的变量错误。 参数 3 - IRP 地址。
0x21C	警告	调用方已复制 IRP 堆栈，但未设置完成例程。 效率低下 -- 请改用 **IoSkipCurrentIrpStackLocation**。 参数 2 - 保留 参数 3 - 保留
0x21D	严重错误	IRP 调度处理程序在接收到删除 IRP 后，没有正确地从其下面的堆栈分离。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - 设备对象地址。
0x21E	严重错误	IRP 调度处理程序在接收到删除 IRP 时未正确删除其设备对象。

I/O 错误代码	Severity	错误消息的原因
		参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - 设备对象地址。
0x21F	非严重错误	驱动程序尚未为所需的 IRP 主函数填写调度例程。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x220	非严重错误	IRP_MJ_SYSTEM_CONTROL 已由 ProviderId 以外的其他人完成。 该 IRP 应提前完成，或者应予以传递。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - ProviderId。
0x221	严重错误	PDO 的 IRP 调度处理程序已删除其设备对象，但在总线关系查询中未报告硬件丢失。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - 设备对象地址。
0x222	严重错误	当 PDO 仍然处于活动状态时，总线筛选器的 IRP 调度处理程序在收到删除 IRP 时已分离。 总线筛选器必须在 **FastIoDetach** 回调中清理。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - 设备对象地址。
0x223	严重错误	总线筛选器的 IRP 调度处理程序已删除其设备对象，但 PDO 仍然存在。 总线筛选器必须在 **FastIoDetach** 回调中清理。

I/O 错误代码	Severity	错误消息的原因
		参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - 设备对象地址。
0x224	严重错误	IRP 调度处理程序返回的状态与 IRP 的 **IoStatus.Status** 不一致。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - 预期状态代码。
0x225	非严重错误	IRP 调度处理程序返回了非法状态 (0xFFFFFFFF)。 这可能是由于未初始化的堆栈变量造成的。 若要调试此错误，请使用具有指定地址的 **ln（列出最近符）** 命令。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - 状态代码。
0x226	严重错误	IRP 调度处理程序返回时没有传递或完成此 IRP，或者有人忘记返回 STATUS_PENDING。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x227	严重错误	IRP 完成例程是可分页代码。 （绝对不允许。） 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x228	非严重错误	如果传递给驱动程序的 IRP 中设置了 **PendingReturned** 字段，则驱动程序的完成例程没有将 IRP 标记为挂起。 这可能会

I/O 错误代码	Severity	错误消息的原因
		导致 Windows 挂起，尤其是当堆栈返回错误时。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x229	严重错误	已经为 IRP 设置了一个取消例程，该 IRP 当前正由堆栈中较低的驱动程序处理，可能会破坏取消例程。 参数 2 - 保留。 参数 3 - 保留。
0x22A	非严重错误	物理设备对象 (PDO) 尚未响应所需的 IRP。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x22B	非严重错误	物理设备对象 (PDO) 忘记使用 **TargetDeviceRelation** 查询的 PDO 填写设备关系列表。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x22C	严重错误	实现 **TargetDeviceRelation** 查询的代码未在 PDO 上调用 **ObReferenceObject**。 参数 2 - 保留。 参数 3 - 保留。
0x22D	非严重错误	调用方完成了一个它不理解的 IRP_MJ_PNP，而不是向下传递。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x22E	非严重错误	调用方已成功完成 IRP_MJ_PNP，而不是将其向下传递。

I/O 错误代码	Severity	错误消息的原因
		参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x22F	非严重错误	调用方已完成未更改的 IRP_MJ_PNP（而不是向下传递 IRP），或者非 PDO 使用 STATUS_NOT_SUPPORTED 的非法值使 IRP 失败。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x230	非严重错误	调用方完成了一个它不理解的 IRP_MJ_POWER，而不是将其传递下去。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x231	严重错误	调用方已成功完成 IRP_MJ_POWER，而不是将其向下传递。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x232	非严重错误	调用方已完成未更改的 IRP_MJ_POWER（而不是向下传递 IRP），或者非 PDO 使用 STATUS_NOT_SUPPORTED 的非法值使 IRP 失败。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x233	非严重错误	查询功能 IRP 中查询功能结构的版本字段未正确初始化。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x234	非严重错误	查询功能 IRP 中查询功能结构的大小字段未正确初始化。

I/O 错误代码	Severity	错误消息的原因
		参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x235	非严重错误	查询功能 IRP 中查询功能结构的地址字段未正确初始化为 -1。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x236	非严重错误	查询功能 IRP 中查询功能结构的 UI 编号字段未正确初始化为 -1。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x237	严重错误	一个驱动程序发送了一个 IRP，该 IRP 仅限于系统使用。 参数 2 - 保留。 参数 3 - 保留。
0x238	警告	**IoInitializeIrp** 的调用方传递了一个由 **IoAllocateIrp** 分配的 IRP。 这既是非法行为，也是不必要的行为，并且会对正常使用中的性能产生负面影响。 如果回收了此 IRP，请参阅 Windows 驱动程序工具包中的 **IoReuseIrp**。 参数 2 - 保留。 参数 3 - 保留。
0x239	警告	**IoCompleteRequest** 的调用方正在完成一个 IRP，该 IRP 从未通过调用转发到 **IoCallDriver** 或 **PoCallDriver**。 这可能是一个错误。 参数 2 - 保留。 参数 3 - 保留。
0x23A	严重错误	一个驱动程序在一个 IRQL 转发了一个 IRP，这对于这个主要代码来说是非法的。

I/O 错误代码	Severity	错误消息的原因
		参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x23B	非严重错误	调用方更改了它不理解的 IRP 的状态字段。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。

下表列出了可能出现的其他 I/O **验证**错误。 当**增强型 I/O 验证**被激活时，会出现这些错误。 有关详细信息，请参阅增强型 I/O 验证。

展开表

I/O 错误代码	Severity	错误消息的原因
0x23C	严重错误	驱动程序已完成 IRP，但未将 IRP 中的取消例程设置为 **NULL**。 参数 2 - 保留。 参数 3 - 保留。
0x23D	非严重错误	驱动程序已返回 STATUS_PENDING，但未通过调用 **IoMarkIrpPending** 将 IRP 标记为挂起。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - 状态代码。
0x23E	非严重错误	驱动程序已将 IRP 标记为挂起，但未返回 STATUS_PENDING。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - 状态代码。

I/O 错误代码	Severity	错误消息的原因
0x23F	严重错误	驱动程序没有从其附加的堆栈中继承 DO_POWER_PAGABLE 位。 参数 2 - 保留。 参数 3 - 保留。
0x240	严重错误	驱动程序正试图删除一个设备对象，该对象已通过之前对 **IoDeleteDevice** 的调用删除。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 2 - 保留。 参数 3 - 保留。
0x241	严重错误	驱动程序在意外删除 IRP 期间分离了其设备对象。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - 设备对象地址。
0x242	严重错误	驱动程序在意外删除 IRP 期间删除了其设备对象。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - 设备对象地址。
0x243	严重错误	驱动程序无法清除 **AddDevice** 末尾的 DO_DEVICE_INITIALIZING 标志。 参数 2 - 保留。 参数 3 - 保留。 参数 4 -
0x244	严重错误	驱动程序未从其所连接的设备对象复制 DO_BUFFERED_IO 或 DO_DIRECT_IO 标志。 参数 2 - 保留。

I/O 错误代码	Severity	错误消息的原因
		参数 3 - 保留。
0x245	严重错误	驱动程序已设置 DO_BUFFERED_IO 和 DO_DIRECT_IO 标志。 这些标记互相排斥。 参数 2 - 保留。 参数 3 - 保留。
0x246	严重错误	驱动程序无法从附加到的设备对象复制 **DeviceType** 字段。 参数 2 - 保留。 参数 3 - 保留。
0x247	严重错误 参数 2 - 保留。 参数 3 - 保留。	某个驱动程序未通过 IRP，而该 IRP 不合法。
0x248	严重错误	驱动程序已将非 PDO 的设备对象添加到设备关系查询中。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - 设备对象地址。
0x249	非严重错误	驱动程序已枚举两个子 PDO，它们返回相同的设备 ID。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - 第一个设备对象地址。 参数 4 - 第二个设备对象地址。
0x24A	严重错误	驱动程序错误地调用了 IRQL 不等于 PASSIVE_LEVEL 的文件 I/O 函数。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - 保留。

I/O 错误代码	Severity	错误消息的原因
		参数 4 - 保留。
0x24B	严重错误	驱动程序已完成 **TargetDeviceRelation** 类型的 IRP_MN_QUERY_DEVICE_RELATIONS 请求，但未正确填写请求或将 IRP 转发到基础硬件堆栈。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - 设备对象地址。
0x24C	非严重错误	驱动程序已返回 STATUS_PENDING，但未通过调用 **IoMarkIrpPending** 将 IRP 标记为挂起。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - 状态代码。
0x24D	严重错误	驱动程序已将无效的设备对象传递给需要 PDO 的函数。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - 设备对象地址。
0x300	非严重错误	驱动程序已返回可疑状态。 这可能是由于驱动程序中存在未初始化的变量错误。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - 可疑状态代码。
0x301	非严重错误	驱动程序已在 IRQL > DISPATCH_LEVEL 转发了 IRP。 参数2 - 检测到错误的驱动程序代码中的地址。

I/O 错误代码	Severity	错误消息的原因
		参数 3 - IRP 地址。 参数 4 - IRQL 值不正确
0x302	非严重错误	驱动程序已在 IRQL > = APC_LEVEL 转发 IRP。 I/O 管理器将需要对 APC 进行排队以完成此请求。 APC 将无法运行，因为调用方已在 APC 级别，因此调用方可能会死锁。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。 参数 4 - IRQL 值不正确。
0x306	非严重错误	驱动程序正在完成 IRP_MJ_PNP（主要）和 IRP_MN_REMOVE_DEVICE（次要）请求，请求中包含故障状态代码。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x307	非严重错误	驱动程序发出了一个 I/O 请求，其中包含一个已经发出信号的事件，并接收到 STATUS_PENDING 响应。 这可能会导致在 I/O 完成之前展开。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x310	非严重错误	驱动程序正在重新初始化仍在使用的 IRP。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x311	非严重错误	驱动程序正在重新初始化使用 IoMakeAssociatedIrp、IoBuildAsynchronousFsdRequest、IoBuildSynchronousFsdRequest、

I/O 错误代码	Severity	错误消息的原因
		IoBuildDeviceIoControlRequest 创建的 IRP。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。
0x312	非严重错误	调用方为 IRP 状态信息字段提供的值大于系统缓冲区的输出部分。 参数2 - 检测到错误的驱动程序代码中的地址。 参数 3 - IRP 地址。

原因

有关原因的说明，请参阅“参数”章节中每个代码的说明。

解决方法

只有当驱动程序验证程序被指示监视一个或多个驱动程序时，才能进行此错误检查。 如果不打算使用驱动程序验证程序，则应将其停用。 有关详细信息，请参阅驱动程序验证程序中的“如何控制驱动程序验证程序”。 还可以考虑更新或删除导致此问题的驱动程序。

如果你是驱动程序开发人员，请使用通过此错误检查获得的信息来修复代码中的错误。

有关驱动程序验证程序的完整详细信息，请参阅驱动程序验证程序。

错误检查 0xCA：PNP_DETECTED_FATAL_ERROR

项目 • 2024/03/04

PNP_DETECTED_FATAL_ERROR 错误检查的值为 0x000000CA。 这表示即插即用管理器遇到严重错误，可能是由于有问题的即插即用驱动程序造成的。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PNP_DETECTED_FATAL_ERROR 参数

参数 1 可确定违规类型。

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x1	新报告的 PDO 的地址	已复制的旧 PDO 的地址	Reserved	**重复的 PDO**：驱动程序的特定实例枚举了多个具有相同设备 ID 和唯一 ID 的 PDO。
0x2	声称的 PDO 的地址	驱动程序对象的地址	Reserved	**无效 PDO**：需要 PDO 的 API 已使用随机内存、FDO 或未初始化的 PDO 调用。 （未初始化的 PDO 是指未通过 **QueryDeviceRelation** 或 **QueryBusRelations** 返回即插即用的 PDO。）
0x3	被查询 PDO 的地址	ID 缓冲区地址	**1**：DeviceID **2**：UniqueID **3**：HardwareID **4**：CompatibleID	**无效 ID**：枚举器返回的 ID 包含非法字符或未正确终止。 （ID 只能包含 0x20 - 0x2B 和 0x2D - 0x7F 范围内的字符。）

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x4	设置了 DOE_DELETE_PENDING 的 PDO 地址	预留	预留	**已删除 PDO 的枚举无效：**枚举器返回了一个 PDO，该 PDO 以前使用 **IoDeleteDevice** 删除过。
0x5	PDO 的地址	预留	预留	**在 devnode 树中链接时释放的 PDO：**当 devnode 仍在树中链接的时候，PDO 上的对象管理器引用计数降至零。 （这通常表示在查询 IRP 中返回 PDO 时，驱动程序没有添加引用。）
0x8	堆栈返回无效总线关系的 PDO 的地址	作为总线关系返回的 PDO 总数	在其中找到第一个 **NULL** PDO 的索引（从零开始）	**NULL 指针作为总线关系返回：**总线上存在的一个或多个设备是 **NULL** PDO。
0x9	传递的连接类型	预留	预留	**传递给 IoDisconnectInterruptEx 的连接类型无效：**驱动程序已将无效的连接类型传递给 **IoDisconnectInterruptEx**。传递到此例程的连接类型必须与相应的对 **IoConnectInterruptEx** 的成功调用所返回的连接类型相匹配。
0xA	驱动程序对象	从驱动程序回调返回后的 IRQL	从驱动程序回调返回后的组合 APC 禁用计数	**通知回调行为不正确：**驱动程序无法在即插即用通知中保留 IRQL 或组合 APC 禁用计数。
0xB	相关 PDO	删除关系	Reserved	**已删除的 PDO 报告为关系：**正在删除的设备的一个删除关系已被删除。

错误检查 0xCB：DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS

项目 • 2024/03/04

DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS 错误检查的值为 0x000000CB。 这表示驱动程序或 I/O 管理器在 I/O 操作后未能释放锁定的页面。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS 参数

展开表

参数	说明
1	内部锁定跟踪结构的地址。
2	0（预留）
3	包含锁定页的 MDL 的地址。
4	锁定页数

有关内存描述符列表的更多信息，请参阅以下主题：

- 使用 MDLs
- MDL 结构 (wdm.h)

有关锁定内存的详细信息，请参阅 锁定可分页代码或数据。

若要调查此错误检查的原因，请使用 !lockedpages 调试器扩展显示当前进程的所有锁定 MDL。

注解

只有当注册表值等于 DWORD 1 时，才会发出此错误检查。

\\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\TrackLockedPages

如果未设置此值，系统将发出信息量较小的**错误检查 0x76** (PROCESS_HAS_LOCKED_PAGES)。

当启用池跟踪选项时，驱动程序验证程序也会发出此错误检查。 有关详细信息，请参阅池跟踪。

错误检查 0xCC：PAGE_FAULT_IN_FREED_SPECIAL_POOL

项目 • 2024/03/04

PAGE_FAULT_IN_FREED_SPECIAL_POOL 错误检查的值为 0x000000CC。 这表示系统引用了先前释放的内存。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PAGE_FAULT_IN_FREED_SPECIAL_POOL 参数

展开表

参数	说明
1	引用的内存地址
2	**0:** 读取 **1:** 写入
3	引用内存的地址（如果已知）
4	Reserved

如果能够识别出导致错误的驱动程序，则其名称将打印在蓝屏上，并存储在内存中的 (PUNICODESTRING) **KiBugCheckDriver 位置。**

原因

驱动程序验证程序专用池选项已捕获系统访问先前释放的内存。 这通常表明存在系统驱动程序同步问题。

有关特殊池的信息，请参阅 Windows 驱动程序工具包的驱动程序验证程序部分。

注解

这不能由 **try - except** 处理程序保护，只能由探测保护。

错误检查 0xCD：PAGE_FAULT_BEYOND_END_OF_ALLOCATION

项目 • 2024/03/04

PAGE_FAULT_BEYOND_END_OF_ALLOCATION 错误检查的值为 0x000000CD。 这表示系统访问的内存超出了某个驱动程序池分配的上限。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PAGE_FAULT_BEYOND_END_OF_ALLOCATION 参数

展开表

参数	说明
1	引用的内存地址
2	**0**: 读取 **1**: 写入
3	引用内存的地址（如果已知）
4	Reserved

如果能够识别出导致错误的驱动程序，则其名称将打印在蓝屏上，并存储在内存中的 (PUNICODESTRING) **KiBugCheckDriver 位置。**

原因

驱动程序从特殊池中分配了 *n* 个字节内存。 随后，系统从该池引用了超过 *n* 个字节。 这通常表明存在系统驱动程序同步问题。

有关特殊池的信息，请参阅 Windows 驱动程序工具包的驱动程序验证程序部分。

注解

这不能由 try - except 处理程序保护，只能由探测保护。

错误检查 0xCE：DRIVER_UNLOADED_WITHOUT_CANCELLING_PENDING_OPERATIONS

项目 • 2024/03/08

DRIVER_UNLOADED_WITHOUT_CANCELLING_PENDING_OPERATIONS 错误检查的值为 0x000000CE。 这表示驱动程序未能在卸载前取消挂起的操作。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_UNLOADED_WITHOUT_CANCELLING_PENDING_OPERATIONS 参数

展开表

参数	说明
1	引用的内存地址
2	**0:** 读取 **1:** 写入
3	引用内存的地址（如果已知）
4	Reserved

如果能够识别出导致错误的驱动程序，则其名称将打印在蓝屏上，并存储在内存中的 (PUNICODESTRING) **KiBugCheckDriver 位置**。

原因

此驱动程序无法在卸载前取消后备列表、DPC、工作线程或其他类似项目。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0xCF：TERMINAL_SERVER_DRIVER_MADE_INCORRECT_MEMORY_REFERENCE

项目 • 2024/03/04

TERMINAL_SERVER_DRIVER_MADE_INCORRECT_MEMORY_REFERENCE 错误检查的值为 0x000000CF。 这表示驱动程序被错误地移植到终端服务器。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

TERMINAL_SERVER_DRIVER_MADE_INCORRECT_MEMORY_REFERENCE 参数

展开表

参数	说明
1	引用的内存地址
2	**0:** 读取 **1:** 写入
3	引用内存的地址（如果已知）
4	Reserved

如果能够识别出导致错误的驱动程序，则其名称将打印在蓝屏上，并存储在内存中的 (PUNICODESTRING) **KiBugCheckDriver 位置。**

原因

驱动程序从系统进程上下文中引用会话空间地址。 这可能是由于驱动程序将项目排队到系统工作线程所致。

此驱动程序需要符合终端服务器的内存管理规则。

错误检查 0xD0：DRIVER_CORRUPTED_MMPOOL

项目 • 2024/03/04

DRIVER_CORRUPTED_MMPOOL 错误检查的值为 0x000000D0。 这表示系统试图访问进程 IRQL 过高的无效内存。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_CORRUPTED_MMPOOL 参数

展开表

参数	说明
1	引用的内存
2	引用时的 IRQL
3	**0**: 读取 **1**: 写入
4	引用内存的地址

原因

当 IRQL 过高时，内核试图访问可分页内存（或者可能完全无效的内存）。 几乎可以肯定，这个问题的最终原因是驱动程序损坏了系统池。

在大多数情况下，如果驱动程序破坏了较大的分配（PAGE_SIZE 或更大），则会导致此错误检查。 较小的分配会导致**错误检查 0xC5** (DRIVER_CORRUPTED_EXPOOL)。

解决方法

如果最近安装了任何新软件，请检查是否安装正确。 请访问制造商的网站，查看更新的驱动程序。

若要调试此错误，请使用驱动程序验证程序的特殊池选项。 如果无法显示导致错误的驱动程序，请使用 Global Flags 实用工具启用特殊的逐池标记。

有关特殊池的信息，请参阅 Windows 驱动程序工具包的驱动程序验证程序部分。

另一种方法是打开 **\\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management** 注册表项。 在此键中，创建或编辑 **ProtectNonPagedPool** 值，并将其设置为等于 DWORD 1。 然后重新启动。 然后，系统将取消映射所有释放的非分页池。 这将防止驱动程序损坏池。 （但是，这并不能保护池不受 DMA 硬件的影响。）

错误检查 0xD1：DRIVER_IRQL_NOT_LESS_OR_EQUAL

项目 • 2024/03/04

DRIVER_IRQL_NOT_LESS_OR_EQUAL 错误检查的值为 0x000000D1。 这表示内核模式驱动程序在进程 IRQL 过高时试图访问可分页内存。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_IRQL_NOT_LESS_OR_EQUAL 参数

展开表

参数	说明
1	引用的内存。
2	引用时的 IRQL。
3	• 0 - 读取 • 1 - 写入 • 2 - 执行 • 8 - 执行
4	引用内存的地址。 在此地址上使用 ln（列出最近的符号）可以查看函数的名称。

原因

若要确定原因，需要 Windows 调试器、编程体验以及访问故障模块的源代码。

通常，发生此错误时，驱动程序尝试访问可分页（或完全无效）的地址，而中断请求级别 (IRQL) 太高。 这可能是由于：

- 在 DISPATCH_LEVEL 或更高级别执行时，取消引用错误指针（例如 NULL 或释放的指针）。

- 在 DISPATCH_LEVEL 或更高级别访问可分页数据。
- 在 DISPATCH_LEVEL 或更高级别执行可分页代码。

如果能够识别出导致错误的驱动程序，则会将其名称打印在蓝屏上，并存储在内存中的 (PUNICODESTRING) **KiBugCheckDriver** 位置。 可以使用一个调试器命令 **dx（显示调试器对象模型表达式）**来显示此内容：**dx KiBugCheckDriver**。

此错误检查通常是由使用了不正确内存地址的驱动程序引起的。

页面错误可能的原因包括以下事件：

- 该函数被标记为可分页，并且在提升的 IRQL（包括获取锁）下运行。
- 对另一个驱动程序中的函数进行了函数调用，该驱动程序已卸载。
- 函数是通过使用无效指针的函数指针调用的。

有关 Windows IRQL 的详细信息，请参阅由 Pavel Yosifovich、Mark E. Russinovich、David A. Solomon 和 Alex Ionescu 编写的 Windows 内部资料第 7 版第 1 部分。

解决方法

如果问题是由正在开发的驱动程序引起的，请确保在错误检查时正在执行的函数是：

- 未标记为可分页
- 不调用任何其他可以调出的内联函数。

!analyze 调试程序扩展显示有关错误检查的信息，有助于确定根本原因。 下面的示例是 !analyze 的输出。

dbgcmd

```
DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pageable (or completely invalid) address at
an
interrupt request level (IRQL) that is too high.  This is usually
caused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
Arguments:
Arg1: fffff808add27150, memory referenced
Arg2: 0000000000000002, IRQL
Arg3: 0000000000000000, value 0 = read operation, 1 = write operation
Arg4: fffff808adc386a6, address which referenced memory
```

如果能够识别出导致错误的驱动程序，则会将其名称打印在蓝屏上，并存储在内存中的 (PUNICODESTRING) **KiBugCheckDriver** 位置。 可以使用一个调试器命令 dx（显示调试

器对象模型表达式）来显示此内容：`dx KiBugCheckDriver`。

```
dbgcmd
```

```
0: kd> dx KiBugCheckDriver
KiBugCheckDriver                 : 0xffffc6092de892c8 : "Wdf01000.sys"
[Type: _UNICODE_STRING *]
```

如果转储文件中有陷阱帧，请使用 .trap 命令将上下文设置为提供的地址。

若要开始调试这种类型的错误检查，请使用 k、kb、kc、kd、kp、kP、kv（显示堆栈回溯）命令检查堆栈跟踪。

在调试器中，运行 !irql 命令用于在调试器中断之前显示目标计算机上处理器的 IRQL 信息。 例如：

```
dbgcmd
```

```
0: kd> !irql
Debugger saved IRQL for processor 0x0 -- 2 (DISPATCH_LEVEL)
```

在这种类型的错误检查的大多数情况下，问题不在于 IRQL 级别，而在于正在访问的内存。

由于此错误检查通常是由使用了不正确内存地址的驱动程序引起的，因此请使用参数 1、3 和 4 进行进一步调查。

将 ln（列出最近符号）与参数 4 一起使用以查看调用的函数的名称。 还要检查 !analyze 输出，以确定错误代码是否已识别。

在参数 1 地址上使用 !pool 查看是否为分页池。 使用 !address 和高级 !pte 命令以了解有关此内存区域的更多信息。

使用显示内存命令检查参数 1 中命令中引用的内存。

使用 u、ub、uu (unassemble) 命令查看引用参数 4 中内存的地址中的代码。

使用 `lm t n` 命令列出内存中加载的模块。 使用 !memusage 并检查系统内存的一般状态。

驱动程序验证程序

驱动程序验证程序是一个实时运行的工具，用于检查驱动程序的行为。 例如，驱动程序验证程序检查内存资源（如内存池）的使用。 如果在执行驱动程序代码时标识错误，它

会主动创建一个异常，以允许进一步检查该部分驱动程序代码。 驱动程序验证程序管理器内置于 Windows 中，可在所有 Windows 电脑上使用。

若要启动驱动程序验证程序管理器，请在命令提示符下键入 **verifier**。 你可以配置要验证的驱动程序。 验证驱动程序的代码在运行时会增加开销，因此请尝试验证尽可能少的驱动程序。 有关详细信息，请参阅驱动程序验证程序。

注解

如果你不具备使用 Windows 调试器来解决此问题的能力，则可以使用一些基本的故障排除技术。

- 在事件查看器中检查系统日志以获取其他错误消息，这可能有助于识别导致此错误检查的设备或驱动程序。
- 如果在错误检查消息中标识了驱动程序，请禁用该驱动程序，或与制造商联系以获取驱动程序更新。
- 确保安装的任何新硬件都与已安装的 Windows 版本兼容。 例如，可以在 Windows 10 规范 中获取所需硬件的相关信息。

有关其他常规疑难解答信息，请参阅蓝屏数据。

错误检查 0xD2：BUGCODE_ID_DRIVER

项目 • 2024/03/04

BUGCODE_ID_DRIVER 错误检查的值为 0x000000D2。 这表示 NDIS 驱动程序出现问题。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

BUGCODE_ID_DRIVER 参数

展开表

参数 1	参数 2	参数 3	参数 4	消息和原因
微型端口块的地址	请求的字节数	0	1	**在引发的 IRQL 上分配共享内存。** 驱动程序使用 IRQL >= DISPATCH_LEVEL 调用 **NdisMAllocateSharedMemory**。
微型端口块的地址	提交到 **NdisMResetComplete** 的 *Status* 值	提交到 **NdisMResetComplete** 的 *AddressingReset* 值	0	**在没有挂起时完成重置。** 驱动程序调用 **NdisMResetComplete**，但没有重置挂起。
微型端口块的地址	包含被释放的地址的内存页	共享内存签名的地址	正在释放虚拟地址	**释放未分配的共享内存。** 驱动程序调用 **NdisMFreeSharedMemory** 或 **NdisMFreeSharedMemoryAsync**，其地址不在 NDIS 共享内存中。
微型端口块的地址	数据包阵列中错误包含的数据包的地址	数据包阵列的地址	阵列中的数据包数	**表示数据包不属于它。** 微型口的数据包阵列已损坏。

参数 1	参数 2	参数 3	参数 4	消息和原因
MiniBlock 的地址	驱动程序对象的地址	0	0	**NdisAddDevice: AddDevice** 用不在 **NdisMiniDriverList** 中的 **MiniBlock** 调用。
MiniBlock 的地址	MiniBlock 的引用计数	0	0	**NdisMUnload: MiniBlock** 正在卸载，但它仍在 **NdisMiniDriverList** 上。
微型端口块的地址	内存页	包装器上下文	共享内存签名的地址	**覆盖过去分配的共享内存。** 正在写入的地址不在 NDIS 共享内存中。

在以下错误检查实例中，参数的含义取决于消息和参数 4 的值。

展开表

参数 1	参数 2	参数 3	参数 4	消息和原因
微型端口块的地址	微型端口中断的地址	微型端口计时器队列的地址	1	**卸载而不取消注册中断。** 微型端口驱动程序在未取消注册中断的情况下初始化失败。
微型端口块的地址	微型端口计时器队列的地址	微型端口中断的地址	2	**卸载而不取消注册中断。** 微型端口驱动程序在停止过程中未取消注册其中断。
微型端口块的地址	微型端口中断的地址	微型端口计时器队列的地址	1	**卸载而不注销计时器。** 微型端口驱动程序在未成功取消其所有计时器的情况下初始化失败。

参数 1	参数 2	参数 3	参数 4	消息和原因
微型端口块的地址	微型端口计时器队列的地址	微型端口中断的地址	2	**卸载而不注销计时器。** 微型端口驱动程序已停止，但未成功取消其所有计时器。

注解

此错误检查代码仅出现在 Windows 2000 和 Windows XP 上。 在 Windows Server 2003 及更高版本中，相应的代码是**错误检查 0x7C** (BUGCODE_NDIS_DRIVER)。

在 Windows 的检查版本中，只有此错误检查的**在引发的 IRQL 上分配共享内存**和**在没有挂起时完成重置**实例会发生。 错误检查 0xD2 的所有其他实例都替换为 ASSERTs。 有关详细信息，请参阅突入调试器。

> **备注**
>
> Windows 10 版本 1803 之前的旧版 Windows 上提供已检查的版本。 使用诸如驱动程序验证程序和 GFlags 之类的工具在更高版本的 Windows 中检查驱动程序代码。

错误检查 0xD3：DRIVER_PORTION_MUST_BE_NONPAGED

项目 • 2024/03/04

DRIVER_PORTION_MUST_BE_NONPAGED 错误检查的值为 0x000000D3。 这表示系统试图访问进程 IRQL 过高的可分页内存。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_PORTION_MUST_BE_NONPAGED 参数

展开表

参数	说明
1	引用的内存
2	引用时的 IRQL
3	**0**: 读取 **1**: 写入
4	引用内存的地址

如果能够识别出导致错误的驱动程序，则其名称将打印在蓝屏上，并存储在内存中的 (PUNICODESTRING) **KiBugCheckDriver 位置**。

原因

此错误检查通常是由驱动程序错误地将自己的代码或数据标记为可分页引起的。

解决方法

若要开始调试，请使用内核调试器获取堆栈跟踪：**!analyze** 调试扩展显示有关错误检查的信息，有助于确定根本原因，然后使用 **kb（显示堆栈回溯）**命令获取堆栈跟踪。

错误检查 0xD4：SYSTEM_SCAN_AT_RAISED_IRQL_CAUGHT_IMPROPER_DRIVER_UNLOAD

项目 • 2024/03/04

SYSTEM_SCAN_AT_RAISED_IRQL_CAUGHT_IMPROPER_DRIVER_UNLOAD 错误检查的值为 0x000000D4。 这表示驱动程序在卸载之前未取消挂起的操作。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SYSTEM_SCAN_AT_RAISED_IRQL_CAUGHT_IMPROPER_DRIVER_UNLOAD 参数

展开表

参数	说明
1	引用的内存
2	引用时的 IRQL
3	**0:** 读取 **1:** 写入
4	引用内存的地址

如果能够识别出导致错误的驱动程序，则其名称将打印在蓝屏上，并存储在内存中的 (PUNICODESTRING) **KiBugCheckDriver 位置**。

原因

此驱动程序无法在卸载前取消后备列表、DPC、工作线程或其他类似项目。 随后，系统试图访问驱动程序在升高的 IRQL 处的先前位置。

解决方法

若要开始调试，请使用内核调试器获取堆栈跟踪：**!analyze** 调试扩展显示有关错误检查的信息，有助于确定根本原因，然后使用 **kb（显示堆栈回溯）**命令获取堆栈跟踪。 如果已识别出导致错误的驱动程序，请激活驱动程序验证程序并尝试复制此错误。

有关 驱动程序验证程序的完整详细信息，请参阅 Windows 驱动程序工具包。

错误检查 0xD5：DRIVER_PAGE_FAULT_IN_FREED_SPECIAL_POOL

项目 • 2024/03/04

DRIVER_PAGE_FAULT_IN_FREED_SPECIAL_POOL 错误检查的值为 0x000000D5。 这表示驱动程序引用了先前释放的内存。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_PAGE_FAULT_IN_FREED_SPECIAL_POOL 参数

展开表

参数	说明
1	引用的内存地址
2	**0**: 读取 **1**: 写入
3	引用内存的地址（如果已知）
4	Reserved

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。 如果能够识别出导致错误的驱动程序，则其名称将打印在蓝屏上，并存储在内存中的 (PUNICODESTRING) **KiBugCheckDriver 位置**。

原因

驱动程序验证程序**特殊池**选项已捕获到驱动程序正在访问先前释放的内存。

有关特殊池的信息，请参阅 Windows 驱动程序工具包的驱动程序验证程序部分。

注解

这不能由 try - except 处理程序保护，只能由探测保护。

错误检查 0xD6：DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION

项目 • 2024/03/04

DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION 错误检查的值为 0x000000D6。 这表示驱动程序访问的内存超出了其池分配的上限。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION 参数

展开表

参数	说明
1	引用的内存地址
2	**0**: 读取 **1**: 写入
3	引用内存的地址（如果已知）
4	Reserved

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。 如果能够识别出导致错误的驱动程序，则其名称将打印在蓝屏上，并存储在内存中的 (PUNICODESTRING) *KiBugCheckDriver 位置*。

原因

驱动程序分配了 *n* 个字节的内存，却引用了超过 *n* 个字节的内存。 驱动程序验证程序**特殊池**选项检测到此冲突。

有关特殊池的信息，请参阅 Windows 驱动程序工具包的驱动程序验证程序部分。

注解

这不能由 **try - except** 处理程序保护，只能由探测保护。

错误检查 0xD7：DRIVER_UNMAPPING_INVALID_VIEW

项目 • 2024/03/08

DRIVER_UNMAPPING_INVALID_VIEW 错误检查的值为 0x000000D7。 这表示驱动程序正在尝试取消映射未映射的地址。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_UNMAPPING_INVALID_VIEW 参数

展开表

参数	说明
1	要取消映射的虚拟地址
2	**1**：正在取消映射视图 **2**：正在提交视图
3	0
4	0

注解

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。 使用 kb （显示堆栈回溯）命令获取堆栈跟踪：可以从堆栈跟踪中确定导致错误的驱动程序。

错误检查 0xD8：DRIVER_USED_EXCESSIVE_PTES

项目 • 2024/03/04

DRIVER_USED_EXCESSIVE_PTES 错误检查的值为 0x000000D8。 这表示没有其他系统页表条目 (PTE)。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_USED_EXCESSIVE_PTES 参数

展开表

参数	说明
1	指向导致错误（Unicode 字符串）或零的驱动程序名称的指针
2	导致错误的驱动程序使用的 PTE 数（如果参数 1 为非零）
3	免费系统 PTE 总数
4	系统 PTE 总数

如果能够识别出导致错误的驱动程序，则其名称将打印在蓝屏上，并存储在内存中的 (PUNICODESTRING) **KiBugCheckDriver 位置**。

原因

这通常是由驱动程序未正确清理内存使用引起的。 参数 1 显示使用 PTE 最多的驱动程序。 调用堆栈将显示实际导致错误检查的驱动程序。

解决方法

两个驱动程序可能都需要修复。 可能还需要增加系统 PTE 的总数。

错误检查 0xD9：LOCKED_PAGES_TRACKER_CORRUPTION

项目 • 2024/03/04

LOCKED_PAGES_TRACKER_CORRUPTION 错误检查的值为 0x000000D9。 这表示内部锁定的页面跟踪结构已损坏。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

LOCKED_PAGES_TRACKER_CORRUPTION 参数

参数 1 表示违规类型。 其他参数的含义取决于参数 1 的值。

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x01	内部锁定跟踪结构的地址	内存描述符列表的地址	为当前进程锁定的页数	MDL 在同一进程列表中插入两次。
0x02	内部锁定跟踪结构的地址	内存描述符列表的地址	为当前进程锁定的页数	MDL 在系统范围的列表中插入两次。
0x03	找到的第一个内部跟踪结构的地址	内部锁定跟踪结构的地址	内存描述符列表的地址	释放时，在进程列表中发现了 MDL 两次。
0x04	内部锁定跟踪结构的地址	内存描述符列表的地址	0	删除 MDL 后，在系统范围的列表中找到该 MDL。

原因

参数 1 的值表示出现错误。

错误检查 0xDA：SYSTEM_PTE_MISUSE

项目 • 2024/03/04

SYSTEM_PTE_MISUSE 错误检查的值为 0x000000DA。 这表示页表项 (PTE) 例程的使用方式不正确。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SYSTEM_PTE_MISUSE 参数

参数 1 表示违规类型。 其他参数的含义取决于参数 1 的值。

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x01	内部锁定跟踪结构的地址	内存描述符列表的地址	重复的内部锁定跟踪结构的地址	正在释放的映射重复。
0x02	内部锁定跟踪结构的地址	系统期望释放的映射数	驱动程序请求释放的映射数	正在释放的映射数不正确。
0x03	找到的第一个内部跟踪结构的地址	系统期望释放的映射地址	驱动程序请求释放的映射地址	正在释放的映射地址不正确。
0x04	内部锁定跟踪结构的地址	在 MDL 中，系统期望的页帧编号应该是第一个	当前在 MDL 中位于第一个的页帧编号	映射的 MDL 的第一页在映射 MDL 后发生了更改。
0x05	找到的第一个内部跟踪结构的地址	系统期望释放的虚拟地址	驱动程序请求释放的虚拟地址	自映射 MDL 以来，正在释放的 MDL 中的起始虚拟地址已更改。
0x06	驱动程序指定的 MDL	驱动程序指定的虚拟地址	要释放的映射数（由驱动程序指定）	正在释放的 MDL 从未映射（或当前未映射）。
0x07	初始映射	映射的数量	预留	（仅限 Windows 2000）映射范围被双重分配。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x08	初始映射	调用方正在释放的映射数	系统认为应释放的映射数	（仅限 Windows 2000）调用方要求释放错误数量的映射。
0x09	初始映射	调用方正在释放的映射数	系统认为已释放的映射索引	（仅限 Windows 2000）调用方要求释放多个映射，但至少有一个映射未分配。
0x0A	**1**：驱动程序在 MDL 中请求“故障时进行错误检查”。 **0**：驱动程序未在 MDL 中请求“故障时进行错误检查”。	调用方正在分配的映射数	请求的映射池的类型	（仅限 Windows 2000）调用方要求分配零映射。
0x0B	损坏的映射	调用方正在分配的映射数	请求的映射池的类型	（仅限 Windows 2000）进行此分配时，映射列表已损坏。 损坏的映射位于可能的最低映射地址之下。
0x0C	损坏的映射	调用方正在分配的映射数	请求的映射池的类型	（仅限 Windows 2000）进行此分配时，映射列表已损坏。 损坏的映射位于可能的最低映射地址之上。
0x0D	初始映射	调用方正在释放的映射数	映射池的类型	（仅限 Windows 2000）调用方正在尝试释放零映射。
0x0E	初始映射	调用方正在释放的映射数	映射池的类型	（仅限 Windows 2000）调用方正在尝试释放映射，但防护映射已被覆盖。
0x0F	不存在的映射	调用方试图释放的映射数	正在释放的映射池的类型	（仅限 Windows 2000）调用方正在尝试释放一个不存在的映射。 不存在的映射位于可能的最低映射地址之下。
0x10	不存在的映射	调用方试图释放的映射数	正在释放的映射池的类型	（仅限 Windows 2000）调用方正在尝试释放一个不存在的映射。 不存在的映射位于可能的最高映射地址之上。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x11	不存在的映射	调用方试图释放的映射数	正在释放的映射池的类型	（仅限 Windows 2000）调用方正在尝试释放一个不存在的映射。 不存在的映射位于映射地址空间的底部。
0x100	所请求的映射数	调用方标识标记	调用此例程调用方的例程的地址	调用方请求了 0 个映射。
0x101	第一个映射地址	调用方标识标记	所有者的标识标记	调用方正在尝试释放它不拥有的映射地址范围。
0x102	第一个映射地址	调用方标识标记	预留	调用方试图释放的映射地址空间显然是空的。
0x103	无效映射的地址	调用方标识标记	映射地址空间中的映射数	调用方试图释放的映射地址空间仍被保留。 MmUnmap**预留**Mapping 必须在 MmFreeMappingAddress 之前调用。
0x104	第一个映射地址	调用方标识标记	所有者的标识标记	调用方正试图将 MDL 映射到它不拥有的映射地址空间。
0x105	第一个映射地址	调用方标识标记	预留	调用方正试图将 MDL 映射到无效的映射地址空间。 调用方很可能指定了一个无效的地址。
0x107	第一个映射地址	非空映射的地址	最后一个映射地址	调用方正试图将 MDL 映射到未正确保留的映射地址空间。 调用方应在调用 MmMapLockedPagesWith**预留**Mapping 之前调用 MmUnmap**预留**Mapping
0x108	第一个映射地址	调用方标识标记	所有者的标识标记	调用方正试图取消映射它不拥有的锁定映射地址空间。
0x109	第一个映射地址	调用方标识标记	预留	调用方正试图取消映射一个显然为空的锁定虚拟地址空间。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x10A	第一个映射地址	锁定的映射地址空间中的映射数	要取消映射的映射数	调用方正试图取消映射比锁定的映射地址空间中实际存在的映射更多的映射。
0x10B	第一个映射地址	调用方标识标记	要取消映射的映射数	调用方正试图取消映射当前未映射的锁定虚拟地址空间的一部分。
0x10C	第一个映射地址	调用方标识标记	要取消映射的映射数	调用方没有取消映射整个锁定的映射地址空间。
0x200	第一个映射地址	0	0	调用方正试图保留不包含任何映射的映射地址空间。
0x201 0x202	要保留的第一个映射地址	已保留的映射的地址	要保留的映射数	调用方试图保留的映射之一已被保留。
0x300	要释放的第一个映射地址	0	0	调用方正试图释放不包含任何映射的映射地址空间。
0x301	映射的地址	0	0	调用方试图释放不允许释放的映射。
0x302	调用方试图释放的地址。	预留	预留	调用方正试图释放当前未映射的系统地址。
0x303	第一个映射地址	要释放的映射数	0	调用方正试图释放未保留的映射地址范围。
0x304	第一个映射地址	要释放的映射数	0	调用方正试图释放从不同分配中间开始的映射地址范围。
0x305	第一个映射地址	调用方试图释放的映射数	应释放的映射数	调用方试图释放错误数量的映射。
0x306	第一个映射地址	空闲映射地址	要释放的映射数	调用方试图释放的其中一个映射已经是空闲的。
0x400	I/O 空间映射的基址	要释放的页数	0	调用者试图释放系统不知道的 I/O 空间映射。

原因

参数 1 的值表示出现错误。

堆栈跟踪将标识导致错误的驱动程序。

错误检查 0xDB：DRIVER_CORRUPTED_SYSPTES

项目 • 2024/03/04

DRIVER_CORRUPTED_SYSPTES 错误检查的值为 0x000000DB。 这表示试图在无效的 IRQL 中操作内存，可能是由于系统 PTE 损坏。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_CORRUPTED_SYSPTES 参数

展开表

参数	说明
1	引用的内存
2	IRQL
3	**0:** 读取 **1:** 写入
4	引用内存的代码中的地址

原因

驱动程序试图访问 IRQL 过高的可分页（或完全无效）内存。 此错误检查几乎总是由损坏的系统 PTE 的驱动程序引起的。

解决方法

如果发生此错误检查，则可以通过编辑注册表来检测罪魁祸首。 在 **\\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management** 注册表项中，创建或编辑 **TrackPtes** 值，然后将其设置为等于 DWORD 3。 然后重新启动。 然后，系统将保存堆栈跟踪，如果驱动程序提交相

同的错误，系统将发出**错误检查 0xDA** (SYSTEM_PTE_MISUSE)。 然后，堆栈跟踪将标识导致错误的驱动程序。

错误检查：0xDC DRIVER_INVALID_STACK_ACCESS

项目 • 2024/03/04

DRIVER_INVALID_STACK_ACCESS 错误检查的值为 0x000000DC。 这表示驱动程序访问了位于堆栈线程的堆栈指针下方的堆栈地址。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_INVALID_STACK_ACCESS 参数

无

注解

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查：0xDE POOL_CORRUPTION_IN_FILE_AREA

项目 • 2024/03/04

POOL_CORRUPTION_IN_FILE_AREA 错误检查的值为 0x000000DE。 这表示驱动程序的池内存已损坏，该内存用于存放指向磁盘的页面。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

POOL_CORRUPTION_IN_FILE_AREA 参数

无

原因

当内存管理器取消引用文件时，它发现池内存中存在此损坏。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查：0xDF IMPERSONATING_WORKER_THREAD

项目 • 2024/03/04

IMPERSONATING_WORKER_THREAD 错误检查的值为 0x000000DF。 这表示工作项在完成之前没有禁用模拟。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

IMPERSONATING_WORKER_THREAD 参数

展开表

参数	说明
1	导致此错误的工作程序例程
2	传递给此工作程序例程的参数
3	指向工作项的指针
4	Reserved

原因

工作线程正在模拟另一个进程，在返回之前未能禁用模拟。

错误检查 0xE0：ACPI_BIOS_FATAL_ERROR

项目 • 2024/03/08

ACPI_BIOS_FATAL_ERROR 错误检查的值为 0x000000E0。 这表示某个计算机组件出现故障。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

ACPI_BIOS_FATAL_ERROR 参数

此错误检查的参数由 BIOS 发布，而不是由 Windows 发布。 它们只能由硬件供应商进行解释。

原因

计算机的 BIOS 报告系统中的某个组件故障严重，Windows 无法运行。 BIOS 指示除了发出错误检查之外别无选择。

解决方法

可以通过运行计算机随附的诊断磁盘或工具来确定哪个组件有故障。

如果没有此工具，则必须联系系统供应商并报告此错误消息。 他们将帮助解决此硬件问题。 这使 Windows 能够运行。

Microsoft 无法解决此错误。 只有硬件供应商才有资格对其进行分析。

错误检查 0xE1: WORKER_THREAD_RETURNED_AT_BAD_IRQL

项目 • 2024/03/08

WORKER_THREAD_RETURNED_AT_BAD_IRQL 错误检查的值为 0x000000E1。 这表示工作线程已完成并返回 IRQL >= DISPATCH_LEVEL。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WORKER_THREAD_RETURNED_AT_BAD_IRQL 参数

展开表

参数	说明
1	工作例程的地址
2	工作线程返回的 IRQL
3	工作项参数
4	工作项地址

原因

工作线程已完成并返回 IRQL >= DISPATCH_LEVEL。

解决方法

要查找导致错误的驱动程序，请使用 **ln（列出最接近符号）**调试器命令：

```
dbgcmd
```

```
kd> ln address
```

其中 *address* 是参数 1 中给出的工作例程地址。

错误检查：0xE2 MANUALLY_INITIATED_CRASH

项目 • 2024/03/04

MANUALLY_INITIATED_CRASH 错误检查的值为 0x000000E2。 这表示用户有意从内核调试器或键盘启动崩溃转储。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

MANUALLY_INITIATED_CRASH 参数

无

注解

有关手动启动的故障转储的详细信息，请参阅强制系统崩溃。

错误检查：0xE3 RESOURCE_NOT_OWNED

项目 • 2024/03/04

RESOURCE_NOT_OWNED 错误检查的值为 0x000000E3。 这表示线程试图释放它不拥有的资源。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

RESOURCE_NOT_OWNED 参数

展开表

参数	说明
1	资源地址
2	线程地址
3	所有者表的地址（如果存在）
4	Reserved

错误检查 0xE4：WORKER_INVALID

项目 • 2024/03/04

WORKER_INVALID 错误检查的值为 0x000000E4。 这表示不应包含执行工作项的内存中确实包含这样的项，或者当前活动的工作项已排队。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WORKER_INVALID 参数

参数 1 指示代码位置。

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x0	工作项的地址	池块开始	池块结束	释放了一个活动的工作项。
0x1	工作项的地址	队列编号	0	一个活动的工作项已排入队列。
0x2	工作项的地址	I/O 工作例程的地址	0	已释放排队的 I/O 工作项。
0x3	工作项的地址	无效对象的地址	0	试图用无效对象初始化 I/O 工作项。
0x5	工作项的地址	队列编号	NUMA 节点为目标，如果搜索了所有节点，则为 -1。	在初始化工作队列之前，尝试将工作项排队。
0x6	工作项的地址	队列编号	0	提供的队列类型无效。
0x7	工作项的地址	队列编号	0	试图将具有无效工作例程地址的工作项排队。

原因

这通常是由驱动程序释放仍包含执行工作项的内存引起的。

错误检查 0xE6：DRIVER_VERIFIER_DMA_VIOLATION

项目 • 2024/03/04

DRIVER_VERIFIER_DMA_VIOLATION 错误检查的值为 0x000000E6。 这是所有驱动程序验证程序 DMA **验证**违规的错误检查代码。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

> **ⓘ 备注**
>
> 未启用驱动程序验证程序时，可以观察到 E6 主要错误检查代码。 如果在未启用驱动程序验证程序的情况下遇到此代码，请参阅 DMA **验证**页了解详细信息。

DRIVER_VERIFIER_DMA_VIOLATION 参数

参数 1 是唯一感兴趣的参数。 此参数标识完全冲突。 如果附加了调试器，调试器中会显示一条信息性消息。

展开表

参数 1	错误原因和调试器消息
0x00 - 其他 DMA 错误。	该代码表示参数 2 所指示的两种错误： 0x1 - 驱动程序试图将过多的字节刷新到映射寄存器文件的末尾。 参数 3 - MDL 中剩余的字节数。 参数 4 - 请求刷新的剩余字节数。 0x2 - Windows 已用完连续映射寄存器。 参数 3 - 需要映射寄存器。 参数 4 - 连续映射寄存器的数量。

参数 1	错误原因和调试器消息
0x01	性能计数器已减少。 将显示计数器的旧值和新值。
0x02	性能计数器增长过快。 计数器值显示在调试器中。
0x03	驱动程序释放了过多的 DMA 通用缓冲区。 通常这意味着它两次释放相同的缓冲区。 参数 2 - 释放的额外通用缓冲区数。
0x04	驱动程序释放了过多的 DMA 适配器通道。 通常这意味着它两次释放同一适配器通道。 参数 2 - 释放的额外适配器通道数。
0x05	驱动程序释放了过多的 DMA 映射寄存器。 通常这意味着它两次释放同一映射寄存器。 参数 2 - 释放的额外映射寄存器数。
0x06	驱动程序释放了过多的 DMA 散点/聚集列表。 通常这意味着它两次释放相同的散点/聚集列表。 参数 2 - 分配的散点聚集列表。 参数 3 - 释放散点聚集列表。
0x07	驱动程序试图在没有首先释放所有公共缓冲区的情况下释放适配器。 参数 2 - 指向 DMA 适配器的指针。 参数 3 - 未完成的公共缓冲区的数量。 参数 4 - 指向相应的内部验证程序数据的指针。
0x08	驱动程序试图释放适配器，但没有首先释放所有适配器通道、公共缓冲区或分散/聚集列表。 参数 2 - 指向 DMA 适配器的指针。 参数 3 - 未完成的适配器通道数。 参数 4 - 指向相应的内部验证程序数据的指针。
0x09	驱动程序试图在未首先释放所有映射寄存器的情况下释放适配器。 参数 2 - 指向 DMA 适配器的指针。

参数 1	错误原因和调试器消息
	参数 3 - 未完成的映射寄存器数。 参数 4 - 指向相应的内部验证程序数据的指针。
0x0A	驱动程序试图释放适配器，但没有首先释放其所有散点/聚集列表。 参数 2 - 指向 DMA 适配器的指针。 参数 3 - 未完成的散点聚集列表数。 参数 4 - 指向相应的内部验证程序数据的指针。
0x0B	驱动程序同时分配了过多的适配器通道（每个适配器只允许一个适配器通道）。 参数 2 - 未完成的适配器通道。
0x0C	驱动程序试图同时分配过多的映射寄存器。 参数 2 - 必需的映射寄存器。 参数 3 - 最大映射寄存器数。
0x0D	驱动程序未刷新其适配器缓冲区。 参数 2 - 映射的字节数。 参数 3 - 一次可映射的最大字节数。
0x0E	驱动程序尝试在没有锁定缓冲区的情况下进行 DMA 传输。 有问题的缓冲区位于分页内存中。 参数 2 - DMA 缓冲区 MDL 的地址。
0x0F	驱动程序或硬件在其分配的 DMA 缓冲区之外写入。 参数 2 是冲突代码。 0x01：DMA 缓冲区之前的标记已被修改。所需标记为 DmaVrfy0。 参数 3 - 缓冲区长度。 参数 4 - 缓冲区启动。 0x02：DMA 缓冲区之前的标记已被修改。 所需标记为 DmaVrfy0。 参数 3 - 缓冲区长度。 参数 4 - 缓冲区启动。 0x03：可用映射寄存器被覆盖。

参数 1	错误原因和调试器消息
	参数 3 - 损坏地址。 所需填充模式为 0x0F。 0x04：缓冲区之前的填充被错误地修改。 参数 3 - 缓冲区启动。 所需填充为 0x0F。 参数 4 - 损坏地址。 0x05：缓冲区之后的填充被错误地修改。 参数 3 - 缓冲区启动。 参数 4 - 损坏地址。 所需填充模式为 0x0F。
0x10	驱动程序试图释放其映射寄存器，而有些寄存器仍在映射中。 参数 2 - 仍映射的寄存器数。
0x11	驱动程序对适配器有太多未完成的引用计数。 参数 2 - 引用计数。 参数 3 - 指向 DMA 适配器的指针。 参数 4 - 指向相应的内部验证程序数据的指针。
0x13	驱动程序以不正确的 IRQL 调用 DMA 例程。 参数 2 是冲突代码。 0x01：当前的 IRQL 与预期不同。 参数 3 - 预期 IRQL。 参数 4 - 当前 IRQL。 0x02：当前 IRQL 高于预期。 参数 3 - 预期的最大 IRQL。 参数 4 - 当前 IRQL。
0x14	驱动程序以不正确的 IRQL 调用 DMA 例程。
0x15	驱动程序试图分配过多的映射寄存器。 参数 2 - 分配的映射寄存器。 参数 3 - 最大映射寄存器数。
0x16	驱动程序试图刷新未映射的缓冲区。 参数 2 - 映射寄存器的系统虚拟空间中的地址。 参数 3 - 指向相应的内部验证程序数据的指针。

参数 1	错误原因和调试器消息
0x18	驱动程序使用一个已释放但已不存在的适配器尝试 DMA 操作。 参数 2 - 指向 DMA 适配器的指针。 参数 3 - 指向相应的内部验证程序数据的指针。
0x19	驱动程序向 HAL 例程传递了一个空 DMA_ADAPTER 值。
0x1B	驱动程序将地址和 MDL 传递给 HAL 例程。 但是，此地址不在此 MDL 的范围内。 参数 2 - 超出 MDL 边界的虚拟地址。 参数 3 - MDL。
0x1D	驱动程序试图映射已映射的地址范围。 参数 2 - 映射开始的缓冲区。 参数 3 - 映射结束的缓冲区。 参数 4 - 缓冲区中已映射的系统地址。
0x1E	驱动程序调用 **HalGetAdapter**。 此函数已过时 -- 必须改用 **IoGetDmaAdapter**。
0x1F	无效的 DMA 缓冲区。 驱动程序引用了无效的系统地址，要么是在第一个 MDL 之前，要么是在第一个 MDR 结束之后，要么是使用了一个比 MDL 缓冲区更长的传输长度，并且在 MDL 内跨越了一个页面边界。参数 2 是违规代码。 0x01：虚拟缓冲区地址位于第一个 MDL 之前。 参数 3 - DMA 缓冲区开始的虚拟地址。 参数 4 - 指向描述 DMA 缓冲区的第一个 MDL 的指针。 0x02：虚拟地址位于第一个 MDL 之后。 参数 3 - DMA 缓冲区开始的虚拟地址。 参数 4 - 指向描述 DMA 缓冲区的第一个 MDL 的指针。 0x03：额外的传输长度越过页面边界。 参数 3 - 指向描述 DMA 缓冲区的 MDL 的指针。

参数 1	错误原因和调试器消息
	参数 4 - DMA 传输的长度。 0x04：DMA 缓冲区的虚拟地址未与缓存对齐。 参数 3 - DMA 缓冲区开始的虚拟地址。 参数 4 - 指向描述 DMA 缓冲区的 MDL 的指针。 0x05：DMA 缓冲区长度未与缓存对齐。 参数 3 - DMA 缓冲区的长度。 参数 4 - 指向描述 DMA 缓冲区的 MDL 的指针。
0x20	驱动程序试图刷新尚未映射的映射寄存器。 参数 2 - 映射寄存器基。 参数 3 - DMA 缓冲区开始的 VA。 参数 4 - 指向用于描述 DMA 缓冲区的 MDL 的指针。
0x21	驱动程序试图映射一个零长度的缓冲区进行传输。 参数 2 - 指向相应的内部验证程序数据的指针。
0x22	DMA 缓冲区未映射到系统 VA 中。 参数 2 - MDL
0x23	无法刷新尚未完成或取消的通道。 参数 2 - 冲突代码。 值：0x00：非法通道刷新 参数 3 - 控制器 ID。 参数 4 - 通道号。
0x24	缓冲区不足，无法满足请求的长度。 参数 2 - 未计入的长度。
0x25	未知设备说明版本。
0x26	IOMMU 检测到 DMA 冲突。

参数 1	错误原因和调试器消息
	参数 2 - 故障设备的设备对象。 参数 3 - 故障信息（通常是物理地址故障）。 参数 4 - 故障类型（特定于硬件）。

原因

有关原因的说明，请参阅“参数”章节中每个代码的说明。

解决方法

只有当驱动程序验证程序被指示监视一个或多个驱动程序时，才能进行此错误检查。 如果不打算使用驱动程序验证程序，则应将其停用。 还可以考虑删除导致此问题的驱动程序。

如果你是驱动程序编写者，请使用通过此错误检查获得的信息来修复代码中的错误。

有关驱动程序验证程序的详细信息，请参阅驱动程序验证程序。

错误检查 0xE7：INVALID_FLOATING_POINT_STATE

项目 • 2024/03/04

INVALID_FLOATING_POINT_STATE 错误检查的值为 0x000000E7。 这表示线程保存的浮点状态无效。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INVALID_FLOATING_POINT_STATE 参数

参数 1 指示哪些有效性检查失败。 未使用参数 4。 其他参数的含义取决于参数 1 的值。

展开表

参数 1	参数 2	参数 3	错误消息的原因
0x0	标志字段	0	保存的上下文标志字段无效。 未设置 FLOAT_SAVE_VALID，或某些预留位为非零。
0x1	保存的 IRQL	当前 IRQL	当前处理器的 IRQL 与保存浮点上下文时不同。
0x2	拥有此浮点上下文的线程的已保存地址	当前线程	保存的上下文不属于当前线程。

原因

还原线程以前保存的浮点状态时，发现该状态无效。

参数 1 指示哪些有效性检查失败。

错误检查：0xE8 INVALID_CANCEL_OF_FILE_OPEN

项目 • 2024/03/04

INVALID_CANCEL_OF_FILE_OPEN 错误检查的值为 0x000000E8。 这表示向 IoCancelFileOpen 传递了一个无效的文件对象。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅蓝屏错误疑难解答。

INVALID_CANCEL_OF_FILE_OPEN 参数

展开表

参数	说明
1	传递给 IoCancelFileOpen 的文件对象
2	传递给 IoCancelFileOpen 的设备对象
3	Reserved
4	Reserved

原因

传递给 IoCancelFileOpen 的文件对象无效。 它应具有一个引用。 名为 IoCancelFileOpen 的驱动程序出现故障。

错误检查：0xE9 ACTIVE_EX_WORKER_THREAD_TERMINATION

项目 • 2024/03/08

ACTIVE_EX_WORKER_THREAD_TERMINATION 错误检查的值为 0x000000E9。 这表示活动的执行工作线程正在终止。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

ACTIVE_EX_WORKER_THREAD_TERMINATION 参数

展开表

参数	说明
1	正在退出的 ETHREAD
2	预留
3	预留
4	预留

原因

执行工作线程在未执行工作线程耗尽代码的情况下被终止。 这是禁止的；排队到 ExWorkerQueue 的工作项不得终止其线程。

堆栈跟踪应指示原因。

错误检查 0xEA：THREAD_STUCK_IN_DEVICE_DRIVER

项目 • 2024/03/04

THREAD_STUCK_IN_DEVICE_DRIVER 错误检查的值为 0x000000EA。 这表示设备驱动程序中的线程在无休止地旋转。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

THREAD_STUCK_IN_DEVICE_DRIVER 参数

展开表

参数	说明
1	指向停滞线程对象的指针
2	指向 DEFERRED_WATCHDOG 对象的指针
3	指向有问题的驱动程序名称的指针
4	**在内核调试器中：** 命中“拦截”错误 0xEA 次数 **在蓝屏上：** 1

原因

设备驱动程序在无限循环中旋转，最有可能等待硬件处于空闲状态。

这通常表明硬件本身有问题，或者设备驱动程序对硬件编程不正确。 通常情况下，这是由于视频卡或显示器驱动程序不佳造成的。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

将 **.thread（设置寄存器上下文）**命令与参数 1 一起使用。 然后使用 **kb（显示堆栈回溯）**查找线程停滞的位置。

如果内核调试器已连接并在 Windows 检测到超时条件时运行。 则会调用 **DbgBreakPoint**，而不是 **KeBugCheckEx**。 将向调试器输出详细消息。 有关详细信息，请参阅将输出发送到调试器。

此消息将包括错误检查参数。 由于没有发出实际的错误检查，因此 **.bugcheck（显示错误检查数据）**命令将无效。 还可以通过在 32 位系统上使用 **dd watchdog!g_WdBugCheckData L5**" 从 Watchdog 的全局变量检索四个参数，或者在 64 位系统上使用 **dq watchdog!g_WdBugCheckData L5**"。

以这样的交互式方式调试此错误可帮助找到有问题的线程，在其中设置断点，然后使用 **g (Go)** 返回到旋转代码以进一步调试。

在多处理器计算机（操作系统版本 3790 或更早版本）上，如果旋转线程因硬件中断而中断，并且在错误检查时 ISR 或 DPC 例程正在运行，则可能会超时。 这是因为超时的工作项可以同时在第二个 CPU 上交付和处理。 如果发生这种情况，则必须更深入地查看有问题线程的堆栈，以确定导致超时的旋转代码。 使用 **dds（显示字词和符号）**命令执行此操作。

错误检查 0xEB：DIRTY_MAPPED_PAGES_CONGESTION

项目 • 2024/03/04

DIRTY_MAPPED_PAGES_CONGESTION 错误检查的值为 0x000000EB。 这表示没有可用页面来继续操作。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DIRTY_MAPPED_PAGES_CONGESTION 参数

展开表

参数	说明
1	脏页总数
2	发送至页面文件的脏页数
3	仅限 Windows Server 2003：在错误检查时可用的非分页池的大小（以页为单位） Windows Vista 及更高版本：预留
4	仅限 Windows Server 2003：当前滞留的转换页面数 Windows Vista 及更高版本：最近修改的写入错误状态

原因

文件系统驱动程序堆栈已死锁，大多数修改后的页面都指向文件系统。 由于文件系统不可操作，因此系统崩溃，因为修改后的页面都不能在不丢失数据的情况下重复使用。 堆栈中的任何文件系统或筛选器驱动程序都可能出现故障。

若要查看常规内存统计信息，请使用 **!vm 3** 扩展。

此错误检查可能由以下任意原因造成：

- 驱动程序已阻止、死锁已修改或映射的页面编写器。 此类示例包括互斥死锁或对文件系统驱动程序或筛选器驱动程序中分页内存的访问。 这表明存在驱动程序 bug。

  如果参数 1 或参数 2 较大，则有可能出现这种情况。 使用 **!vm 3**。

- 存储驱动程序未处理请求。 此类示例包括滞留队列和非响应驱动器。 这表明存在驱动程序 bug。

  如果参数 1 或参数 2 较大，则有可能出现这种情况。 使用 **!process 0 7**。

- 仅限 Windows Server 2003：没有足够的池可用于存储堆栈，无法写出修改过的页。 这表明存在驱动程序 bug。

  如果参数 3 较小，则有可能出现这种情况。 使用 **!vm** 和 **!poolused 2**。

错误检查：0xEC SESSION_HAS_VALID_SPECIAL_POOL_ON_EXIT

项目 • 2024/03/04

SESSION_HAS_VALID_SPECIAL_POOL_ON_EXIT 错误检查的值为 0x000000EC。 此错误检查表明在会话驱动程序仍保留内存时发生了会话卸载。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SESSION_HAS_VALID_SPECIAL_POOL_ON_EXIT 参数

展开表

参数	说明
1	会话 ID
2	正在泄漏的特殊池页数
3	Reserved
4	Reserved

原因

此错误是由会话驱动程序在卸载会话之前未释放其特殊池分配引起的。 这表示 win32k.sys、atmfd.dll、rdpdd.dll 或视频驱动程序中存在错误。

错误检查 0xED：UNMOUNTABLE_BOOT_VOLUME

项目 • 2024/03/08

UNMOUNTABLE_BOOT_VOLUME 错误检查的值为 0x000000ED。 这表示 I/O 子系统试图装载启动卷，但失败了。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

UNMOUNTABLE_BOOT_VOLUME 参数

展开表

参数	说明
1	启动卷的设备对象
2	文件系统中的状态代码，用于描述装载卷失败的原因
3	Reserved
4	Reserved

解决方法

如果要调试此错误，请使用 !analyze -v 扩展。 此扩展显示与错误相关的特定于数据的错误。

此错误检查通常与操作系统引导存储设备（如硬盘驱动器）的故障有关。 若要尝试验证文件系统并恢复启动记录，以下故障排除步骤可能会有所帮助。

1. 在 Windows 10 中，使用故障排除>高级选项>启动修复。 可能需要创建可启动的恢复媒体，并从 U 盘或 DVD 启动，以运行 Windows 恢复环境。
2. 在 Windows 恢复环境中的命令提示符下，使用 CHKDSK /r 尝试修复文件系统。
3. 使用 bootrec 命令修复主记录和启动记录。

如果这些步骤不成功，则可能是硬盘驱动器出现故障。 一些硬盘驱动器供应商提供的诊断工具可能有助于确认硬件故障。

错误检查 0xEF：CRITICAL_PROCESS_DIED

项目 • 2024/03/08

`CRITICAL_PROCESS_DIED` 错误检查的值为 0x000000EF。 此检查指示关键系统进程已终止。 如果系统终止，关键进程会强制系统进行错误检查。 当进程状态损坏或破坏时，会进行此检查。 当损坏或破坏发生时，由于这些进程对 Windows 操作至关重要，因此当操作系统完整性存在问题时，会进行系统错误检查。

内置 Windows 关键系统服务包括 csrss.exe、wininit.exe、logonui.exe、smss.exe、services.exe、conhost.exe 和 winlogon.exe。

开发人员还可以创建服务并将其恢复选项设置为**重新启动计算机**。 有关详细信息，请参阅设置在服务出现故障时执行的恢复操作。

> **重要**
>
> 本主题是面向程序员的。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

CRITICAL_PROCESS_DIED 参数

展开表

参数	说明
1	进程对象
2	如果为 0，则进程已终止。 如果为 1，则线程已终止。
3	Reserved
4	Reserved

解决方法

确定此问题的原因通常需要使用调试器来收集其他信息。 应检查多个转储文件，以查看此停止代码是否具有类似的特征，例如在停止代码出现时运行的代码。

有关详细信息，请参阅使用 Windows 调试器 (WinDbg) 进行崩溃转储分析、使用 !analyze 扩展和 !analyze。

在许多情况下，还会在系统错误检查之前创建用户转储。 通常，当用户转储可用时，应首先检查该转储，以找出问题的根本原因。 从内核转储调试用户模式代码存在限制，包括分页/缺失数据。 有关详细信息，请参阅用户模式转储文件。

请考虑使用事件日志来查看是否存在导致此停止代码的错误。 如果存在，可以使用这些错误来检查特定的服务或其他要调查的代码。

获取有关相关代码的信息后，在执行此代码之前，在相关代码中设置断点。 从这里，向前单步浏览代码，查看用于控制代码流的关键变量的值。 仔细检查代码的此区域，查找虚假假设或其他错误。

使用错误检查的第二个参数来确定是正在终止的进程还是线程导致了错误检查。

如果是进程，请使用 !process 命令在失败点之前和之后显示有关进程的信息，以查找异常行为。 进程资源管理器实用工具可以收集有关父子关系和正在运行的进程的常规信息。

如果是线程，请考虑使用 !thread 命令显示有关线程的信息。 有关内核模式下线程的信息，请参阅更改上下文。

有关线程、进程和 Windows 保护的关键代码（如 wininit 和 csrss）的其他细节的一般信息，请参阅 Pavel Yosifovich、Mark E. Russinovich、David A. Solomon 和 Alex Ionescu 编写的 *Windows 内部资料*。

常规疑难解答技巧

如果无法使用调试器，这些常规故障排除提示可能会有所帮助。

- 如果最近向系统添加了硬件，请尝试删除或更换该硬件。 也可以与制造商联系，查看是否有可用的修补程序。
- 如果最近添加了新的设备驱动程序或系统服务，请尝试删除或更新它们。 尝试确定系统中导致新错误检查代码出现的原因。
- 在事件查看器中检查系统日志以获取其他错误消息，这可能有助于发现导致错误的设备或驱动程序。 有关详细信息，请参阅打开事件查看器。 在系统日志中查找与蓝屏同时出现的严重错误。
- 请与制造商联系，查看是否有更新的系统 BIOS 或固件。
- 可尝试运行系统制造商提供的硬件诊断。

- 确保安装的任何新硬件都与已安装的 Windows 版本兼容。 例如，可以在 Windows 10 规范 中获取所需硬件的相关信息。
- 运行病毒检测程序。 病毒可以感染为 Windows 格式化的所有类型的硬盘。 由此产生的磁盘损坏可能会生成系统错误检查代码。 确保病毒检测程序检查主启动记录是否存在感染。
- 使用**系统文件检查器**工具修复丢失或损坏的系统文件。 **系统文件检查器**是 Windows 中的一个实用工具，供用户扫描 Windows 系统文件中的损坏情况并恢复损坏的文件。 使用以下命令运行**系统文件检查器**工具 (SFC.exe)。

  控制台

  ```
  SFC /scannow
  ```

  有关详细信息，请参阅使用系统文件检查器工具修复丢失或损坏的系统文件 。
- 在**设备管理器**中查看是否有任何设备标有感叹号 (!)。 查看驱动程序属性中显示的事件日志，以查找任何故障驱动程序。 请尝试更新相关驱动程序。

另请参阅

- 使用 Windows 调试器 (WinDbg) 进行故障转储分析
- 使用 WinDbg 分析内核模式转储文件

错误检查 0xF0：STORAGE_MINIPORT_ERROR

项目 • 2024/03/04

STORAGE_MINIPORT_ERROR 错误检查的值为 0x00000F0。 它表示存储微型端口驱动程序无法完成 SRB 请求。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅蓝屏错误疑难解答。

STORAGE_MINIPORT_ERROR 参数

展开表

参数	说明
1	错误代码。 请参阅下面的值。
2	请参阅下面的值。
3	请参阅下面的值。
4	请参阅下面的值。

值

text

```
1: Miniport failed to complete a SRB request even after successful reset
operation.
    2 - Driver name unicode string address
    3 - SRB address
    4 - Storport unit device object

2: Miniport failed to complete a SRB request even after successful abort
operation for the SRB.
    2 - Driver name unicode string address
    3 - Abort SRB address
    4 - SRB that was being aborted

3: Miniport failed to complete a request within a given timeout.
    2 - Driver name unicode string address
```

```
    3 - SRB address
    4 - Timeout of the request

4: Miniport failed to complete a request for a crypto operation. This can
occur if it is trying to enable an encryption key on an ICE (Inline Crypto
Engine) enabled UFS host.
    2 - Driver name unicode string address
    3 - The STOR_CRYPTO_OPERATION_TYPE for this failure, typically
StorCryptoOperationInsertKey
    4 - Reserved
```

原因

存储微型端口驱动程序中的一个错误使 SRB 请求无法完成。 有关特定类型的故障，请参阅上面列出的错误代码值。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

参数 2 中返回的驱动程序名称应指向有问题的驱动程序。

另请参阅

错误检查代码参考

Storport 与 Storport 微型端口驱动程序交互的接口

错误检查 0xF1：SCSI_VERIFIER_DETECTED_VIOLATION

项目 • 2024/03/04

SCSI_VERIFIER_DETECTED_VIOLATION 错误检查的值为 0x000000F1。 这是所有驱动程序验证程序 **SCSI 验证**违规的错误检查代码。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SCSI_VERIFIER_DETECTED_VIOLATION 参数

参数 1 可确定违规类型。

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x1000	传递的第一个参数	传递的第二个参数	Reserved	微型端口驱动程序向 **ScsiPortInitialize** 传递了错误的参数。
0x1001	延迟（以微秒为单位）	预留	预留	微型端口驱动程序调用了 **ScsiPortStallExecution**，并指定了大于 0.1 秒的延迟，导致处理器停滞时间过长。
0x1002	例程的地址耗时太长	微型端口的 HW_DEVICE_EXTENSION 的地址	例程的持续时间（以微秒为单位）	端口驱动程序调用的微型端口例程执行时间超过 0.5 秒。 （对于大多数例程，0.5 秒是限制。但是，**HwInitialize** 例程允许 5 秒，而 **FindAdapter** 例程则不允许。）
0x1003	微型端口的 HW_DEVICE_EXTENSION 的地址	SRB 的地址	Reserved	微型端口驱动程序不止一次地完成了一个请求。
0x1004	SRB 的地址	微型端口的 HW_DEVICE_EXTENSION 的地址	Reserved	微型端口驱动程序完成了一个 SRB 状态无效的请求。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x1005	微型端口的 HW_DEVICE_EXTENSION 的地址	LOGICAL_UNIT_EXTENSION 的地址	Reserved	微型端口驱动程序调用 **ScsiPortNotification** 请求 **NextLuRequest**，但未标记的请求仍处于活动状态。
0x1006	微型端口的 HW_DEVICE_EXTENSION 的地址	虚拟地址无效	Reserved	微型端口驱动程序向 **ScsiPortGetPhysicalAddress** 传递了无效的虚拟地址。 （这通常意味着提供的地址没有映射到公共缓冲区。）
0x1007	ADAPTER_EXTENSION 的地址	微型端口的 HW_DEVICE_EXTENSION 的地址	Reserved	总线的重置保留期已结束，但微型端口驱动程序仍有未完成的请求。
0x2001	延迟（以微秒为单位）	预留	预留	Storport 微型端口驱动程序调用了 StorPortStallExecution，并指定了超过 0.1 秒的延迟，导致处理器停滞时间过长。
0x2002	预留	预留	预留	StorPortGetUncachedExtension 未从微型端口驱动程序的 HwStorFindAdapter 例程调用。 **StorPortGetUncachedExtension** 例程只能从微型端口驱动程序的 **HwStorFindAdapter** 例程调用，并且只能用于总线主适配器。 在调用 **StorPortGetUncachedExtension** 之前，Storport 微型端口驱动程序必须设置 HW_INITIALIZATION_DATA (Storport) 结构的 **SrbExtensionSize**。
0x2003	预留	预留	预留	已将无效地址传递给 StorPortGetDeviceBase 例程。 **StorPortGetDeviceBase** 例程仅支持系统即插即用 (PnP) 管理器分配给驱动程序的地址。
0x2004	预留	预留	预留	Storport 微型端口驱动程序多次完成同一 I/O 请求。
0x2005	预留	预留	预留	Storport 微型端口驱动程序将无效的虚拟地址传递给 **StorPortRead***xxx* 或 **StorPortWrite***xxx* 例程之一。 这通常意味着提供的地址没有映射

参数 1	参数 2	参数 3	参数 4	错误消息的原因
				到公共缓冲区。 指定的*寄存器*或*端口*必须位于 StorPortGetDeviceBase 例程返回的映射内存空间范围内。

原因

有关原因的说明，请参阅“参数”章节中每个代码的说明。

解决方法

只有当驱动程序验证程序被指示监视一个或多个驱动程序时，才能进行此错误检查。 如果不打算使用驱动程序验证程序，则应将其停用。 还可以考虑删除导致此问题的驱动程序。

如果你是驱动程序编写者，请使用通过此错误检查获得的信息来修复代码中的错误。

驱动程序验证程序 **SCSI 验证**选项仅在 Windows XP 及更高版本中可用。 驱动程序验证程序 **Storport 验证**选项仅在 Windows 7 及更高版本中可用。 有关驱动程序验证程序的完整详细信息，请参阅 Windows 驱动程序工具包。

错误检查 0xF2：HARDWARE_INTERRUPT_STORM

项目 • 2024/03/11

HARDWARE_INTERRUPT_STORM 错误检查的值为 0x000000F2。 这表示内核检测到了中断风暴。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

HARDWARE_INTERRUPT_STORM 参数

展开表

参数	说明
1	连接到风暴中断向量的 ISR（或链中的第一个 ISR）的地址
2	ISR 上下文值
3	风暴中断向量的中断对象地址
4	如果 ISR 未链接，则为 0x1；如果 ISR 已链接，则为 0x2

原因

此错误检查表示内核已检测到中断风暴。 中断风暴被定义为电平触发的中断信号保持在断言状态。 这对系统造成严重影响，因为系统会硬挂起或“总线锁定”。

发生这种情况的原因如下：

- 一个硬件在被设备驱动程序告知之后不会释放它的中断信号。
- 设备驱动程序不会指示其硬件释放中断信号，因为它不认为中断是从其硬件发起的。
- 即使中断不是从其硬件启动的，设备驱动程序也会声明中断。 请注意，只有当多个设备共享同一 IRQ 时，才会发生这种情况。
- ELCR（边缘电平控制寄存器）设置不正确。
- 边缘和电平中断触发设备共享一个 IRQ。

所有这些情况都会立即硬挂起系统。 不会硬挂起系统，而是启动此错误检查，因为在许多情况下，可以识别罪魁祸首。

当发生错误检查时，屏幕上会显示包含风暴 IRQ 的 ISR（中断服务例程）的模块。 下面是将看到的内容的示例：

控制台

```
*** STOP: 0x000000F2 (0xFCA7C55C, 0x817B9B28, 0x817D2AA0, 0x00000002)
An interrupt storm has caused the system to hang.
*** Address FCA7C55C base at FCA72000, Datestamp 3A72BDEF - ACPI.sys
```

如果第四个参数是 0x00000001，则指向的模块很可能是罪魁祸首。 驱动程序已损坏，或者硬件出现故障。

如果第四个参数是 0x00000002，则指向的模块是链中的第一个 ISR，并且永远不能保证是罪魁祸首。

解决方法

反复经历此错误检查的用户应尝试通过查找与模块驱动程序所在的 IRQ（在本例中，与 ACPI 使用的 IRQ 相同）位于同一 IRQ 上的设备来隔离问题。

错误检查 0xF3：DISORDERLY_SHUTDOWN

项目 • 2024/03/04

DISORDERLY_SHUTDOWN 错误检查的值为 0x000000F3。 这表明 Windows 由于内存不足而无法关闭。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DISORDERLY_SHUTDOWN 参数

展开表

参数	说明
1	脏页总数
2	发送至页面文件的脏页数
3	仅限 Windows Server 2003：在错误检查时可用的非分页池的大小（以页为单位） Windows Vista 及更高版本：保留
4	仅限 Windows Server 2003：当前关闭阶段 Windows Vista 及更高版本：最近修改的写入错误状态

原因

Windows 试图关闭，但没有可用的页面来继续操作。

因为应用程序没有被终止，驱动程序也没有被卸载，所以即使在修改后的写入程序终止后，它们仍然可以继续访问页面。 这会导致系统耗尽页面，因为可以使用页面文件。

错误检查 0xF4：CRITICAL_OBJECT_TERMINATION

项目 • 2024/03/04

CRITICAL_OBJECT_TERMINATION 错误检查的值为 0x000000F4。 这表示对系统操作至关重要的进程或线程意外退出或已终止。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

CRITICAL_OBJECT_TERMINATION 参数

展开表

参数	说明
1	终止对象类型： **0x3**：进程 **0x6**：线程
2	终止对象
3	进程映像文件名
4	指向包含解释性消息的 ASCII 字符串的指针

原因

系统运行需要多个进程和线程。 当它们因任何原因终止时，系统将无法正常运行。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0xF5：FLTMGR_FILE_SYSTEM

项目 • 2024/03/04

FLTMGR_FILE_SYSTEM 错误检查的值为 0x000000F5。 这表示筛选器管理器中发生了不可恢复的失败。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅蓝屏错误疑难解答。

FLTMGR_FILE_SYSTEM 参数

参数 1 表示违规类型。 其他参数的含义取决于参数 1 的值。

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x66	指向操作的回调数据结构的指针。	0	0	微型筛选器从操作前回调返回 FLT_PREOP_SUCCESS_WITH_CALLBACK 或 FLT_PREOP_SYNCHRONIZE，但没有注册相应的操作后回调。
0x67	指向操作的回调数据结构的指针。	0	操作的错误 NTSTATUS 代码	内部对象空间不足，系统无法分配新空间。
0x68	预留	FLT_FILE_NAME_INFORMATIONN 结构的地址	预留	FLT_FILE_NAME_INFORMATION 结构被取消引用过多次。
0x6A	文件的文件对	0	0	无法取消文件打开或文件创建请求，因为已经为该文件创建了一个或多个句柄。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
	象指针。			
0x6B	帧 ID	0	线程	BACKPOCKET IRPCTRL 状态无效。
0x6C	帧 ID	BackPocket 列表	线程	BACKPOCKETED IRPCTR 的嵌套 PageFaults 太多。
0x6D	微型筛选器的上下文结构的地址	CONTEXT_NODE 结构的地址	0	上下文结构被取消引用的次数太多。这意味着，筛选器管理器的 CONTEXT_NODE 结构上的引用计数变为零，而该结构仍附加到其关联的对象。
0x6E	微型筛选器的上下文结构的地址	CONTEXT_NODE 结构的地址	0	上下文结构在释放后被引用。

原因

参数 1 的值指示了问题的原因。 请参阅“参数”部分中的表。

解决方法

如果参数 1 等于 **0x66**，则可以通过验证微型筛选器驱动程序是否为此操作注册了操作后回调来调试此问题。 可以在回调数据结构中找到当前操作。 （请参阅参数 2。）使用 **!fltkd.cbd** 调试器扩展。

如果参数 1 等于 **0x67**，则应验证系统中某个位置是否存在无分页池泄漏。

如果参数 1 等于 **0x6A**，请确保微型筛选器驱动程序在微型筛选器处理此操作的任何时候都没有引用此文件对象（请参阅参数 2）以获取句柄。

如果参数 1 等于**0x6B**或**0x6C**，则发生了不可恢复的内部状态错误，这将导致操作系统进行错误检查。

如果参数 1 等于 **0x6D**，请确保微型筛选器驱动程序不会为给定上下文调用 **FltReleaseContext** 太多次（请参阅参数 2）。

如果参数 1 等于 0x6E，请确保微型筛选器驱动程序在删除给定上下文后不会调用 **FltReferenceContext**（请参阅参数 2）。

错误检查 0xF6：PCI_VERIFIER_DETECTED_VIOLATION

项目 • 2024/03/08

PCI_VERIFIER_DETECTED_VIOLATION 错误检查的值为 0x000000F6。 这表示 BIOS 或 PCI 驱动程序正在验证的其他设备中发生错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PCI_VERIFIER_DETECTED_VIOLATION 参数

参数 1 是唯一需要关注的参数；这标识了检测到的故障的性质。

展开表

参数 1	错误消息的原因
0x01	在对接事件期间，BIOS 对活动网桥进行了重新编程。
0x02	PMCSR 寄存器未在规范规定的时间内更新。
0x03	驱动程序已写入 PCI 设备配置空间的 Windows 控制部分。

原因

PCI 驱动程序在正在验证的设备或 BIOS 中检测到错误。

错误检查 0xF7：DRIVER_OVERRAN_STACK_BUFFER

项目 • 2024/03/04

DRIVER_OVERRAN_STACK_BUFFER 错误检查的值为 0x000000F7。 这表示驱动程序已溢出基于堆栈的缓冲区。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_OVERRAN_STACK_BUFFER 参数

展开表

参数	说明
1	堆栈中的实际安全检查 Cookie
2	预期的安全检查 Cookie
3	预期安全检查 cookie 的位补充
4	0

原因

驱动程序覆盖基于堆栈的缓冲区（或局部变量）的方式会覆盖函数的返回地址，并在函数返回时跳回到任意地址。

这是典型的“缓冲区溢出”黑客攻击。 系统已关闭，以防止恶意用户完全控制它。

解决方法

使用 **kb（显示堆栈回溯）** 命令获取堆栈跟踪。

在缓冲区溢出处理程序和错误检查调用之前，堆栈上的最后一个例程是覆盖其局部变量的例程。

错误检查 0xF8：RAMDISK_BOOT_INITIALIZATION_FAILED

项目 • 2024/03/04

RAMDISK_BOOT_INITIALIZATION_FAILED 错误检查的值为 0x000000F8。 这表示在尝试从 RAM 磁盘启动时发生初始化故障。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

RAMDISK_BOOT_INITIALIZATION_FAILED 参数

展开表

参数	说明
1	指示故障原因。 **1**：在加载程序内存列表中找不到 LoaderXIPRom 描述符。 **2**：无法打开 RAM 磁盘驱动程序（ramdisk.sys 或 \Device\Ramdisk）。 **3**：FSCTL_CREATE_RAM_DISK 失败。 **4**：无法从二进制 GUID 创建 GUID 字符串。 **5**：无法创建指向 RAM 磁盘设备的符号链接。
2	NTSTATUS 代码
3	0
4	0

错误检查 0xF9：DRIVER_RETURNED_STATUS_REPARSE_FOR_VOLUME_OPEN

项目 • 2024/03/04

DRIVER_RETURNED_STATUS_REPARSE_FOR_VOLUME_OPEN 错误检查的值为 0x000000F9。 这表示驱动程序针对 IRP_MJ_CREATE 请求返回了 STATUS_REPARSE，但没有尾随名称。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_RETURNED_STATUS_REPARSE_FOR_VOLUME_OPEN 参数

展开表

参数	说明
1	已打开的设备对象
2	向其发出 IRP_MJ_CREATE 请求的设备对象
3	包含文件新名称的 Unicode 字符串的地址（要重新分析）
4	驱动程序为 IRP_MJ_CREATE 请求返回的信息

注解

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

STATUS_REPARSE 应仅针对带有尾随名称的 IRP_MJ_CREATE 请求返回，因为这表明驱动程序支持名称空间。

有关使用文件系统驱动程序的详细信息，请参阅文件系统驱动程序设计指南。 有关 IRP_MJ_CREATE 请求的信息，请参阅 IRP_MJ_CREATE (IFS)。

错误检查 0xFA：HTTP_DRIVER_CORRUPTED

项目 • 2024/03/04

HTTP_DRIVER_CORRUPTED 错误检查的值为 0x000000FA。 这表示 HTTP 内核驱动程序 (Http.sys) 已达到损坏状态且无法恢复。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

HTTP_DRIVER_CORRUPTED 参数

参数 1 标识 HTTP 内核驱动程序的确切状态。

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x1	工作项的地址	包含工作项检查的文件的名称	文件中的工作项检查行号	工作项无效。 这最终将导致线程池损坏和访问冲突。

错误检查 0xFC：ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY

项目 • 2024/03/04

ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY 错误检查的值为 0x000000FC。 这表示试图执行不可执行的内存。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY 参数

展开表

参数	说明
1	试图执行的虚拟地址
2	页面表项 (PTE) 的内容
3	Reserved
4	Reserved

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。 如果可能，尝试执行非可执行内存的驱动程序名称的 Unicode 字符串将显示在错误检查屏幕上，并且也会保存在 KiBugCheckDriver 中。 否则，通常可以通过运行堆栈跟踪，然后查看当前指令指针来找到有问题的驱动程序。

错误检查 0xFD：DIRTY_NOWRITE_PAGES_CONGESTION

项目 • 2024/03/08

DIRTY_NOWRITE_PAGES_CONGESTION 错误检查的值为 0x000000FD。 这表示没有可用的页面来继续基本的系统操作。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DIRTY_NOWRITE_PAGES_CONGESTION 参数

展开表

参数	说明
1	脏页总数
2	不可写脏页数
3	Reserved
4	最近修改的写入错误状态

原因

这种错误检查通常是因为拥有修改后的不可写页面的组件在将相关文件标记为“不可写入”以进行内存管理后，未能写出这些页面。 这表明存在驱动程序 bug。

解决方法

有关导致问题的驱动程序的详细信息，请使用 !vm 3 扩展，后跟 !memusage 1。

错误检查 0xFE：BUGCODE_USB_DRIVER

项目 • 2024/03/04

BUGCODE_USB_DRIVER 错误检查的值为 0x000000FE。 这表示通用串行总线 (USB) 驱动程序中发生了错误。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

BUGCODE_USB_DRIVER 参数

四个错误检查参数显示在错误检查停止屏幕上，可使用 ! 分析。 参数 1 可确定违规类型。

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x1	预留	预留	预留	USB 堆栈中发生了内部错误。
0x2	挂起的 IRP 的地址	传入的 IRP 的地址	导致错误的 USB 请求块 (URB) 的地址	USB 客户端驱动程序已提交一个 URB，该 URB 仍附加到总线驱动程序中挂起的另一个 IRP。
0x3	预留	预留	预留	USB 微型端口驱动程序已生成错误检查。 这通常发生在硬件故障的响应中。
0x4	IRP 的地址	URB 的地址	Reserved	调用方已提交一个 IRP，该 IRP 已在 USB 总线驱动程序中挂起。
0x5	主机控制器的设备扩展指针	PCI 供应商，控制器的产品 ID	指向终结点数据结构的指针	由于在硬件数据结构中发现错误的物理地址，导致硬件故障。
0x6	对象地址	预期签名	Reserved	内部数据结构（对象）已损坏。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x7	指向 usbport.sys 调试日志的指针	消息字符串	文件名	有关详细信息，请参阅提供的消息字符串。
0x8	1	预留	预留	预留
	2	设备对象	IRP	集线器驱动程序收到 IRP，该驱动程序不需要或尚未注册。
	3	预留	预留	预留
	4	如果参数 3 不为 NULL，则为 PDO。 如果参数 3 为 NULL，则为上下文。	上下文或 NULL	严重 PDO 陷阱
	5	预留	预留	预留
	6	超时代码。 请参阅下表。	超时代码上下文：端口数据	严重超时

如果参数 1 的值为 8，参数 2 的值为 6，则参数 3 是超时代码。 下表提供了超时代码的可能值。

展开表

超时代码	含义
0	非严重超时
1	无法恢复挂起的端口。
2	在挂起端口之前，等待由客户端驱动程序启动的重置完成时超时。
3	在挂起端口之前等待端口完成恢复时超时。
4	在挂起端口之前等待端口更改状态机被禁用时超时。
5	等待挂起端口请求完成时超时。
6	等待端口更改状态机被禁用时超时。
7	等待端口更改状态机关闭时超时。
8	等待集线器从选择性挂起恢复时超时。

超时代码	含义
9	在系统挂起之前，等待集线器从选择性挂起恢复时超时。
10	等待端口更改状态机变为空闲时超时。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0xFF：RESERVE_QUEUE_OVERFLOW

项目 • 2024/03/04

RESERVE_QUEUE_OVERFLOW 错误检查的值为 0x000000FF。 这表示试图将新项目插入保留队列，导致队列溢出。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

RESERVE_QUEUE_OVERFLOW 参数

展开表

参数	说明
1	保留队列的地址
2	保留队列的大小
3	0
4	0

解决方法

!analyze 调试扩展显示有关错误检查的信息，并非常有助于确定根本原因。

Bug 检查 0x100：LOADER_BLOCK_MISMATCH

项目 • 2024/03/01

LOADER_BLOCK_MISMATCH Bug 检查的值为 0x00000100。 这表明加载程序块无效，或者它与正在加载的系统不匹配。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

LOADER_BLOCK_MISMATCH 参数

展开表

参数	说明
1	3
2	加载程序黑色扩展的大小。
3	加载程序块的主要版本
4	加载程序块的次要版本

bug 检查 0x101: CLOCK_WATCHDOG_TIMEOUT

项目 • 2024/02/14

CLOCK_WATCHDOG_TIMEOUT bug 检查的值为 0x00000101。 此 bug 检查指示在多处理器系统中，辅助处理器上的预期时钟中断未在分配的时间间隔内收到。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

CLOCK_WATCHDOG_TIMEOUT参数

展开表

参数	说明
1	时钟中断超时间隔，以名义时钟刻度为单位。
2	0
3	无响应处理器的处理器控制块（PRCB）的地址。
4	挂起处理器的索引。

原因

指定的处理器未处理中断。 通常，当处理器无响应或死锁时，检查发生此 bug。

解决方法

以下调试器命令可用于调查处理器状态、IRQL 级别和正在运行的代码，以尝试确定哪些代码段不允许向前执行。

!analyze

k， kb， kc， kd， kp， kP， kv （显示堆栈回溯）

！ Pcr

! prcb

! irql

另请参阅

Bug 检查代码参考

bug 检查0x102：DPC_WATCHDOG_TIMEOUT

项目 • 2024/02/14

DPC_WATCHDOG_TIMEOUT bug 检查的值为 0x00000102。 这表示未在分配的时间间隔内执行 DPC 监视器例程。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

DPC_WATCHDOG_TIMEOUT参数

展开表

参数	说明
1	DPC 监视器超时间隔（以名义时钟刻度为单位）。
2	挂起处理器的 PRCB 地址。
3	Reserved
4	Reserved

原因

此 bug 检查通常意味着 ISR 在低于时钟级别和调度级别的 IRQL 上挂起，或者 DPC 例程挂起在指定的处理器上。

例如，对于 StorPort 微型端口驱动程序，StorPort.sys处理在DISPATCH_LEVEL运行的例程中的 I/O 完成，并串行调用刚刚完成的所有 IRP 的 I/O 完成例程。 如果 I/O 完成例程唱或一起花费太多时间，键盘和/或鼠标可能会停止响应。 Windows DPC 监视器计时器例程也有可能决定 StorPort 例程花费的时间过长。

解决方法

存储堆栈中的内核驱动程序可以通过高效编码驱动程序的 I/O 完成例程来降低问题的可能性。 如果仍然无法在足够时间内在完成例程中执行所有必要的处理，则例程可以为 I/O 工作创建工作元素，将元素排入工作队列并返回STATUS_MORE_PROCESSING_REQUIRED; 然后，驱动程序的工作线程应找到工作元素，执行工作，并为 IRP 执行 IoCallerDriver，以确保 IRP 的进一步 I/O 处理。

Bug 检查 0x103：MUP_FILE_SYSTEM

项目 • 2024/02/14

MUP_FILE_SYSTEM bug 检查 的值为 0x00000103。 此 bug 检查指示多个 UNC 提供程序（MUP）遇到无效或意外的数据。 因此，MUP 无法将远程文件系统请求传送到网络重定向程序（通用命名约定（UNC）提供程序。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

MUP_FILE_SYSTEM参数

参数 1 标识冲突的类型。

展开表

参数 1	参数 2	参数 3	参数 4	错误原因
0x1	挂起的 IRP 的地址。	找不到其文件上下文的文件对象的地址。	设备对象的地址。	MUP 找不到与文件对象对应的文件上下文。 这通常表示 MUP 正在看到文件对象的 I/O 请求，MUP 未看到相应的 IRP_MJ_CREATE请求。 此 bug 检查的可能原因是筛选器驱动程序错误。
0x2	预期文件上下文的地址。	实际从文件对象检索的地址。	Reserved	已知存在文件对象的文件上下文，但不是预期的（例如，它可能是 **NULL**）。
0x3	IRP 上下文的地址。	IRP 完成状态代码。	完成 IRP 的 UNC 提供程序的驱动程序对象（可能为 **NULL**）。	IRP 完成状态意外或无效。 此 bug 检查仅在使用“已检查的 Windows 版本”时发生，并且仅应由附加到旧网络重定向程序的文件系统筛选器驱动程序引起。 在 Windows 10 版本 1803 之前，已检查的版本

参数 1	参数 2	参数 3	参数 4	错误原因
				在旧版本的 Windows 上可用。 旧重定向程序使用 **FsRtlRegisterUncProvider** 注册 MUP。 此 bug 检查检测到返回在 IRP_MJ_CLEANUP或 IRP_MJ_CLO标准版请求中未STATUS_SUCCESS NTSTATUS 的筛选器驱动程序。
0x4	IRP 的地址	文件对象的地址	文件对象的文件上下文	在文件对象的创建请求完成之前，对文件对象启动 I/O 操作。

注解

对于它处理的所有文件对象，MUP 基于每个文件对象维护上下文信息。

Bug 检查 0x104：AGP_INVALID_ACCESS

项目 • 2024/03/01

AGP_INVALID_ACCESS Bug 检查的值为 0x00000104。 这表示 GPU 写入了以前未提交的一系列加速图形端口 (AGP) 内存。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅蓝屏错误疑难解答。

AGP_INVALID_ACCESS 参数

展开表

参数	说明
1	AGP 验证程序页中的偏移量（在 ULONG 中）到第一个损坏的 ULONG 数据
2	0
3	0
4	0

原因

通常，此 Bug 检查是由未签名或未正确测试的视频驱动程序引起的。 它也可能由旧的 BIOS 引起。

解决方法

检查显示驱动程序和计算机 BIOS 更新。

Bug 检查 0x105：AGP_GART_CORRUPTION

项目 • 2024/02/14

AGP_GART_CORRUPTION bug 检查的值为 0x00000105。 这表示图形光圈重新映射表（GART） 已损坏。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

AGP_GART_CORRUPTION参数

展开表

参数	说明
1	GART 的基址（虚拟）
2	发生损坏的 GART 的偏移量
3	GART 缓存的基址（虚拟）（GART 的副本）
4	0

原因

此 bug 检查通常是由驱动程序不正确的直接内存访问（DMA）引起的。

解决方法

为任何未签名的驱动程序启用驱动程序验证程序。 删除它们或逐个禁用它们，直到识别出erring 驱动程序。

Bug 检查 0x106：AGP_ILLEGALLY_REPROGRAMMED

项目 • 2024/03/01

AGP_ILLEGALLY_REPROGRAMMED Bug 检查的值为 0x00000106。 这表示加速图形端口 (AGP) 硬件已由未经授权的代理重新编程。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

AGP_ILLEGALLY_REPROGRAMMED 参数

展开表

参数	说明
1	最初编程的 AGP 命令寄存器值
2	当前命令寄存器值
3	0
4	0

原因

此 Bug 检查通常是由未签名或未正确测试的视频驱动程序引起的。

解决方法

查看视频制造商的网站以了解更新的显示驱动程序或使用 VGA 模式。

Bug 检查 0x108：THIRD_PARTY_FILE_SYSTEM_FAILURE

项目 • 2024/03/01

THIRD_PARTY_FILE_SYSTEM_FAILURE Bug 检查的值为 0x00000108。 这表示第三方文件系统或文件系统过滤器中出现了不可恢复的问题。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

THIRD_PARTY_FILE_SYSTEM_FAILURE 参数

展开表

参数	说明
1	标识失败的文件系统。 可能的值包括： **1:** Polyserve (Psfs.sys)
2	异常记录的地址。
3	上下文记录的地址。
4	保留。

原因

造成此 Bug 检查的一个可能原因是磁盘损坏。 硬盘上第三方文件系统损坏或坏块（扇区）可能会引发此错误。 损坏的 SCSI 和 IDE 驱动程序也会对 Windows 操作系统读写磁盘的能力产生不利影响，从而导致错误。

另一个可能的原因是非分页池内存耗尽。 如果非分页池完全耗尽，此错误可能会导致系统停止运行。

解决方法

若要调试此问题：请使用参数 3 的 **.cxr（显示上下文记录）**命令，然后使用 **kb（显示堆栈回溯跟踪）。**。

若要解决磁盘损坏问题：请检查事件查看器以查看系统中的 SCSI、IDE 或其他磁盘控制器的错误消息，这些错误消息可能有助于查明导致错误的设备或驱动程序。 尝试禁用任何持续监视系统的病毒扫描程序、备份程序或磁盘碎片整理程序工具。 您还应该运行文件系统或文件系统过滤器制造商提供的硬件诊断程序。

若要解决非分页池内存耗尽问题： 向计算机添加新的物理内存。 这将增加内核可用的非分页池内存的数量。

错误检查 0x109：CRITICAL_STRUCTURE_CORRUPTION

项目 • 2024/10/16

CRITICAL_STRUCTURE_CORRUPTION 错误检查的值为 0x00000109。 这表示内核检测到关键内核代码或数据损坏。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

CRITICAL_STRUCTURE_CORRUPTION 参数

展开表

参数	说明
1	保留
2	预留
3	预留
4	损坏区域的类型。 （请参阅本页后面的表格）

参数 4 的值指示损坏区域的类型。

展开表

参数 4	损坏区域的类型、损坏类型或导致损坏的操作类型
0x0	通用数据区域
0x1	函数修改
0x2	处理器中断调度表 (IDT)
0x3	处理器全局描述符表 (GDT)
0x4	类型 1 进程列表损坏

参数 4	损坏区域的类型、损坏类型或导致损坏的操作类型
0x5	类型 2 进程列表损坏
0x6	调试例程修改
0x7	关键 MSR 修改
0x8	对象类型
0x9	处理器 IVT
0xA	修改系统服务函数
0xB	通用会话数据区域
0xC	修改会话函数或 .pdata
0xD	修改导入表
0xE	修改会话导入表
0xF	Ps Win32 标注修改
0x10	调试开关例程修改
0x11	IRP 分配器修改
0x12	驱动程序调用调度程序修改
0x13	IRP 完成调度程序修改
0x14	IRP 释放器修改
0x15	处理器控制寄存器
0x16	关键浮点控制寄存器修改
0x17	本地 APIC 修改
0x18	内核通知标注修改
0x19	加载的模块列表修改
0x1A	类型 3 进程列表损坏
0x1B	类型 4 进程列表损坏
0x1C	驱动程序对象损坏
0x1D	执行回调对象修改
0x1E	修改模块填充

参数 4	损坏区域的类型、损坏类型或导致损坏的操作类型
0x1F	修改受保护的进程
0x20	通用数据区域
0x21	页面哈希不匹配
0x22	会话页哈希不匹配
0x23	加载配置目录修改
0x24	倒排函数表修改
0x25	会话配置修改
0x26	扩展处理器控制寄存器
0x27	类型 1 池损坏
0x28	类型 2 池损坏
0x29	类型 3 池损坏
0x101	常规池损坏
0x102	修改 win32k.sys

原因

此错误检查通常有三个不同的原因：

1. 驱动程序无意或故意修改了关键内核代码或数据。 Microsoft Windows Server 2003 Service Pack 1 (SP1) 和更高版本的 Windows（基于 x64 的计算机）不允许修补内核，除非通过授权的 Microsoft 源热修补程序。

2. 开发人员尝试使用启动系统时未附加的内核调试器设置正常的内核断点。 仅当在启动时附加调试器时，才能设置正常的断点 (**bp**)。 可以随时设置处理器断点 (**ba**)。

3. 发生硬件损坏。 例如，内核代码或数据可能已经存储在失败的内存中。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

开始时，使用 **k、kb、kc、kd、kp、kP、kv（显示堆栈回溯）**命令检查堆栈跟踪。 可以指定处理器编号以检查所有处理器上的堆栈。

您还可以在导致此停止代码的代码中设置断点，并尝试单步执行故障代码。

有关详细信息，请参阅以下主题：

使用 Windows 调试器 (WinDbg) 进行故障转储分析

如果您不具备使用 Windows 调试器来解决此问题的能力，则可以使用一些基本的故障排除技术。

- 在事件查看器中检查系统日志以获取其他错误消息，这可能有助于识别导致此错误检查的设备或驱动程序。
- 如果在错误检查消息中标识了驱动程序，请禁用该驱动程序，或与制造商联系以获取驱动程序更新。
- 运行 Windows 内存诊断工具以测试内存。 在控制面板搜索框中键入“内存”，然后选择**诊断计算机内存问题**。运行测试后，使用事件查看器查看系统日志下的结果。查找“内存诊断结果”条目以查看结果。
- 你可尝试运行系统制造商提供的硬件诊断。
- 确保安装的任何新硬件都与已安装的 Windows 版本兼容。 例如，可以在 Windows 10 规范 中获取所需硬件的相关信息。
- 有关其他常规故障排除信息，请参阅**分析错误检查蓝屏数据**。

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Bug 检查 0x10A：APP_TAGGING_INITIALIZATION_FAILED

项目 • 2024/03/01

APP_TAGGING_INITIALIZATION_FAILED Bug 检查的值为 0x0000010A。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x10C：FSRTL_EXTRA_CREATE_PARAMETER_VIOLATION

项目 • 2024/03/01

FSRTL_EXTRA_CREATE_PARAMETER_VIOLATION Bug 检查的值为 0x0000010C。 这表示在文件系统运行时库 (FsRtl) 额外创建参数 (ECP) 包中检测到冲突。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

FSRTL_EXTRA_CREATE_PARAMETER_VIOLATION 参数

展开表

参数	说明
1	违规的类型。 （有关详细信息，请参阅本页后面的表格）。
2	0
3	ECP 的地址。
4	ECP 列表的起始地址。

参数 1 的值表示违规类型。

展开表

参数 1	违规类型
0x1	由于指针错误或内存损坏，ECP 签名无效。
0x2	ECP 已设置未定义的标志。
0x3	ECP 不是由 FsRtl 分配的。

参数 1	违规类型
0x4	ECP 设置了一些标志，这些标志对于创建调用方传递的参数来说是非法的。
0x5	ECP 已损坏；其大小小于标头大小。
0x6	正在释放的 ECP 具有非空列表指针；它可能仍然是 ECP 列表的一部分。
0x11	由于指针错误或内存损坏，ECP 列表签名无效。
0x12	ECP 列表设置了未定义的标志。
0x13	ECP 列表不是由 FsRtl 分配的。
0x14	ECP 列表设置了一些标志，这些标志对于创建调用方传递的参数列表来说是非法的。
0x15	创建调用方传递的 ECP 列表为空。

Bug 检查 0x10D：WDF_VIOLATION

项目 • 2024/02/14

WDF_VIOLATION bug 检查的值为 0x0000010D。 这表示内核模式驱动程序框架（KMDF）检测到 Windows 在基于框架的驱动程序中发现了错误。

> **重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 “排查蓝屏错误”。

WDF_VIOLATION参数

参数 1 指示 bug 检查的特定错误代码。 保留参数 4。

展开表

参数 1	参数 2	参数 3	错误消息的原因
0x1	指向 WDF_POWER_ROUTINE_TIMED_OUT_DATA 结构的指针	Reserved	基于框架的驱动程序在电源操作期间超时。 这通常意味着设备堆栈未设置 DO_POWER_PAGABLE位，驱动程序在分页设备堆栈关闭后尝试了可分页操作。
0x2	预留	预留	正在尝试获取当前正在保留的锁。
0x3	WDFREQUEST 句柄	这两个缓冲区上保留的未完成引用数	Windows 驱动程序框架验证程序遇到致命错误。 具体而言，I/O 请求已完成，但框架请求对象无法删除，因为对输入缓冲区、输出缓冲区或两者都有未完成的引用。
0x4	Reserved	调用方地址	NULL 参数传递给需要非 NULL 值的函数。
0x5	传入的句柄值	Reserved	错误类型的框架对象句柄传递给框架对象方法。
0x6			请参阅下表。
0x7	框架对象的句柄	Reserved	驱动程序尝试通过调用 WdfObjectDereference 删除句柄而不是

参数 1	参数 2	参数 3	错误消息的原因
			调用 **WdfObjectDelete** 来错误删除框架对象。
0x8	DMA 事务对象的句柄	Reserved	操作发生在 DMA 事务对象上，但状态不正确。
0x9			当前未使用。
0xA	指向WDF_QUEUE_FATAL_ERROR_DATA结构的指针	Reserved	处理当前在队列中的请求时发生致命错误。
0xB			请参阅下表。
0xC	WDFDEVICE 句柄	指向新 PnP IRP 的指针	驱动程序正在处理另一个更改状态的 PnP IRP 时到达新的状态更改 PnP IRP。
0xD	WDFDEVICE 句柄	指向电源 IRP 的指针	设备电源策略所有者收到了未请求的电源 IRP。 可能有多个电源策略所有者，但只允许一个。 KMDF 驱动程序可以通过调用 **WdfDeviceInitSetPowerPolicyOwnership** 来更改电源策略所有权。
0xE	调用事件回调函数的 IRQL。	在其中返回事件回调函数的 IRQL。	事件回调函数未在调用它的同一 IRQL 处返回。 回调函数直接或间接地更改了 IRQL（例如，通过获取一个旋转锁，将 IRQL 提升为DISPATCH_LEVEL，但不释放旋转锁）。
0xF	事件回调函数的地址。	Reserved	事件回调函数输入了关键区域，但在返回之前它未离开关键区域。

参数 1 等于 0x6

如果参数 1 等于0x6，则处理 WDF 请求时出现致命错误。 在这种情况下，参数 2 进一步指定已根据枚举WDF_REQUEST_FATAL_ERROR定义的致命错误的类型。

展开表

参数 2	参数 3	错误消息的原因
0x1	IRP 的地址	没有更多 I/O 堆栈位置可用

参数 2	参数 3	错误消息的原因
		于设置基础 IRP 的格式。
0x2	WDF 请求句柄值	尝试格式化不包含 IRP 的框架请求对象。
0x3	WDF 请求句柄值	驱动程序尝试发送已发送到 I/O 目标的框架请求。
0x4	指向 WDR_REQUEST_FATAL_ERROR_INFORMATION_LENGTH_MISMATCH_DATA 结构的指针，其中包含指向 IRP 的指针、WDF 请求句柄值、IRP 主函数和尝试写入的字节数	驱动程序已完成框架请求，但已向输出缓冲区写入的字节数超过了 IRP 中指定的字节数。

参数 1 等于 0xB

如果参数 1 等于0xB，则尝试获取或释放锁无效。 在这种情况下，参数 3 进一步指定已发生的错误。

展开表

参数 2	参数 3	错误消息的原因
句柄值	0x0	传递给 **WdfObjectAcquireLock** 或 **WdfObjectReleaseLock** 的句柄表示不支持同步锁的对象。
WDF 旋转锁控点	0x1	旋转锁正由未获取它的线程释放。

原因

有关原因的说明，请参阅“参数”部分中每个代码的说明。

解决方法

！ analyze 调试扩展显示有关 bug 的信息检查，有助于收集信息，例如故障代码模块。

通常，WDF 转储文件将提供有关导致此 bug 的驱动程序检查的进一步信息。 使用此命令查看日志文件。

dbgcmd

```
kd> !wdfkd.wdflogdump <WDF_Driver_Name>
```

如果参数 1 等于 **0x2**，请检查调用方堆栈以确定有问题的锁。

如果参数 1 等于 **0x3**，驱动程序的内核模式驱动程序框架错误日志将包含有关未完成引用的详细信息。

如果参数 1 等于**0x4，请使用*参数 3 的值作为参数 3* 的 ln 调试器命令来确定哪个函数需要非 NULL** 参数。

如果参数 1 等于**0x7，请使用！ wdfkd.wdfhandle*Parameter 2* 扩展命令来确定**句柄类型。

如果参数 1 等于 **0xA**，则WDF_QUEUE_FATAL_ERROR_DATA结构将指示有问题的请求或队列句柄。 它还将指示 NTSTATUS（如果未STATUS_SUCCESS），如果可用。

Bug 检查 0x10E：VIDEO_MEMORY_MANAGEMENT_INTERNAL

项目 • 2024/02/15

VIDEO_MEMORY_MANAGEMENT_INTERNAL bug 检查 的值为 0x0000010E。 这表示视频内存管理器遇到了无法从中恢复的条件。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 “**排查蓝屏错误**”。

VIDEO_MEMORY_MANAGEMENT_INTERNAL参数

参数 1 描述视频内存错误的类型。 必须单独检查不在此表中显示的参数 1 的值。

展开表

参数 1	错误消息的原因
0x1	尝试旋转非旋转范围。
0x2	尝试销毁非空进程堆。
0x3	尝试从光圈段取消映射失败。
0x4	必成功路径中的轮换失败。
0x5	延迟命令失败。
0x6	尝试重新分配已取消其驱逐的分配的资源。
0x7	尝试延迟免费使用无效。
0x8	拆分直接内存访问（DMA）缓冲区包含无效引用。
0x9	尝试逐出分配失败。
0xA	尝试使用固定分配无效。

参数 1	错误消息的原因
0xB	驱动程序从 *BuildPagingBuffer* 返回了无效的错误代码。
0xC	在段中检测到资源泄漏。
0xD	段被不当使用。
0xE	尝试将分配映射到光圈段失败。
0xF	驱动程序从 *AcquireSwizzlingRange* 返回了无效的错误代码。

展开表

参数 1	错误消息的原因
0x10	驱动程序从 *ReleaseSwizzlingRange* 返回了无效的错误代码。
0x11	尝试使用光圈段无效。
0x12	驱动程序溢出提供的 DMA 缓冲区。
0x13	驱动程序溢出了提供的专用数据缓冲区。
0x14	尝试清除所有段失败。
0x15	尝试释放仍处于旋转状态的虚拟地址描述符（VAD）
0x16	驱动程序违反了保证的 DMA 缓冲区模型协定。
0x17	发生意外的系统命令失败。
0x18	尝试释放固定分配的资源失败。
0x19	驱动程序无法修补 DMA 缓冲区。
0x1A	已释放共享分配的所有者。
0x1B	试图释放仍在使用的孔径范围。
0x1C	VidMm 正在尝试从帧缓冲区中旋转分配，但 VA 不会在预期的位置旋转。
0x1D	VidMm 正在尝试使用已取消映射的分页缓冲区。
0x1E	VidMm 正在尝试从错误的进程上下文执行操作。

参数 1	错误消息的原因
0x1F	VidMm 正在尝试使用低于当前标记的围栏标记分配。

展开表

参数 1	错误消息的原因
0x20	VidMm 正在尝试操作分配假定为空闲，但不是。
0x21	VidMm 正在尝试在准备或未刷新的数据外刷新分页缓冲区，在准备开始时在分页缓冲区中找到。
0x22	VidMm 正在尝试将 VA 旋转到无效的范围。
0x23	计划程序在等待完成之前唤醒线程。
0x24	正在销毁分配，该分配具有对其后盾存储的未完成引用...物理内存将泄漏。
0x25	GPU 试图在光圈的未定义区域上写入。
0x26	VIDMM_LOCAL_ALLOC已从其所有者以外的进程关闭。
0x27	当前拆分点未重新编程VIDMM_ALLOC。
0x28	引用全局分配时发生意外异常。
0x29	在操作 VIDMM_ALLOC DMA 引用计数时检测到溢出或下溢。
0x2A	VidMm 正在尝试释放对当前显示分配的最后一个引用。
0x2B	VidMm 正在尝试释放无效的 Cpu 主机光圈页面范围。
0x2C	VidMm 正在尝试将页面范围映射到以前已映射的 Cpu 主机光圈。 最佳大小写内存泄漏。
0x2D	调用 DdiMapCpuHostAperture 失败，但预期成功。
0x2E	调用 DdiUnmapCpuHostAperture 失败，但预期成功。
0x2F	报告的范围大小与为数组分配的元素数不匹配。

展开表

参数 1	错误消息的原因
0x30	GPU 虚拟地址操作期间发生错误。
0x31	正在删除的分页队列仍在由 VidMm 工作线程处理。
0x32	正在删除的分页队列仍包含正在运行的数据包。
0x33	设备正在销毁，但仍有驻留在它的分配。
0x34	堆分配已收到与当前状态不兼容的状态转换事件。
0x35	分页请求在默认分页队列中失败。
0x36	分页请求在以前标记为不可恢复的分页数据包或设备恢复时失败，预计后续调用会成功。
0x37	VidMm 无法在 TDR 期间锁定分配页。
0x38	VidMm 正在释放仍包含引用它的分页数据包的分配。
0x39	VidMm 正在将设备置于无限期的惩罚框，但它有分页数据包。
0x3A	VidMm 工作线程正在运行已挂起的分页队列。
0x3B	在 D3 转换期间，内存仍会旋转到帧缓冲区。
0x3C	内存仍分配或映射到 CPU 主机光圈。 这表示在 D3 转换期间，内存仍可能旋转到帧缓冲区。
0x3D	操作期间指定了无效的段组。
0x3E	未能获取VIDMM_ALLOC运行保护。
0x3F	在移动或碎片整理操作期间恢复计划程序设备与惩罚框状态冲突。 这意味着我们将恢复内存尚无法访问的设备的计划程序。

展开表

参数 1	错误消息的原因
0x40	尝试在另一个操作正在进行时启动准备括号。
0x41	删除具有非零驻留或适配器计数的

参数 1	错误消息的原因
	VIDMM_CROSSADAPTER_ALLOC。
0x42	删除具有负驻留或适配器计数的 VIDMM_CROSSADAPTER_ALLOC。
0x43	保存或还原保留的帧缓冲区内容时，我们无法映射节对象中的至少一页才能向前推进。
0x44	内存预算记账最终在流入中。

原因

此 bug 检查通常是由视频驱动程序行为不当引起的。

解决方法

如果问题仍然存在，检查 Windows 更新更新的视频驱动程序。

Bug 检查 0x10F：RESOURCE_MANAGER_EXCEPTION_NOT_HANDLED

项目 • 2024/02/15

RESOURCE_MANAGER_EXCEPTION_NOT_HANDLED bug 检查的值为 0x0000010F。 这表示内核事务管理器检测到内核模式资源管理器已引发异常，以响应直接回调。 资源管理器处于意外且不可恢复的状态。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

RESOURCE_MANAGER_EXCEPTION_NOT_HANDLED 参数

展开表

参数	说明
1	异常记录的地址
2	上下文记录的地址
3	异常代码的地址
4	资源管理器的地址

Bug 检查 0x111：RECURSIVE_NMI

项目 • 2024/02/14

RECURSIVE_NMI bug 检查 的值为 0x00000111。 此 bug 检查指示在以前的 NMI 正在进行时发生不可屏蔽中断 （NMI）。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

注解

当系统管理中断（SMI）代码中出现错误，并且 SMI 中断 NMI 并启用中断时，会发生此检查 bug。 然后，执行将继续启用 NMIs，另一个 NMI 会中断正在进行的 NMI。

Bug 检查 0x112：MSRPC_STATE_VIOLATION

项目 • 2024/02/14

MSRPC_STATE_VIOLATION bug 检查的值为 0x00000112。 这表示Msrpc.sys驱动程序已启动 bug 检查。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

MSRPC_STATE_VIOLATION参数

参数 1 和 2 是唯一感兴趣的参数。 参数 1 指示状态冲突类型;参数 2 的值由参数 1 的值确定。

展开表

参数 1	参数 2	错误消息的原因
0x01	异常代码	调用方继续出现不可连续异常。
0x02	错误	高级本地过程调用 （ALPC） 返回了无效错误。
0x03	与服务器的会话	调用方在仍在使用期间卸载了 Microsoft 远程过程调用 （MSRPC） 驱动程序。 打开的绑定句柄可能保持不变。
0x04和 0x05	与服务器的会话	从 ALPC 收到无效的 close 命令。
0x06	绑定句柄	尝试再次绑定远程过程调用 （RPC） 处理。
0x07	绑定句柄	尝试对未绑定的绑定句柄执行操作。
0x08	绑定句柄	尝试对已绑定的绑定句柄设置

参数 1	参数 2	错误消息的原因
		安全信息。
0x09	绑定句柄	尝试对已绑定的绑定句柄设置选项。
0x0A	调用对象	尝试取消无效的异步远程过程调用。
0x0B	调用对象	尝试在不需要时推送异步管道调用。
0x0C和 0x0E	管道对象	尝试在异步管道上推送，而无需等待通知。
0x0F	管道对象	第二次尝试同步终止管道。
0x15	最接近错误的对象	发生 RPC 内部错误。
0x16	Reserved	两个因果排序的调用是按 RPC 无法强制执行的顺序发出的。
0x17	调用对象	在完成呼叫之前，服务器管理器例程未取消订阅通知。
0x18	异步句柄	异步句柄上的操作无效。

原因

此 bug 检查最常见的原因是，Msrpc.sys驱动程序的调用方违反了此类调用的状态语义。

错误检查 0x113：VIDEO_DXGKRNL_FATAL_ERROR

项目 • 2024/10/16

VIDEO_DXGKRNL_FATAL_ERROR 错误检查的值为 0x00000113。 这表明 Microsoft DirectX 图形内核子系统检测到违规。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Bug 检查 0x114：VIDEO_SHADOW_DRIVER_FATAL_ERROR

项目 • 2024/02/14

VIDEO_SHADOW_DRIVER_FATAL_ERROR bug 检查的值为 0x00000114。 这表示影子驱动程序检测到冲突。

此 bug 检查很少出现。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

Bug 检查 0x115：AGP_INTERNAL

项目 • 2024/03/01

AGP_INTERNAL Bug 检查的值为 0x00000115。 这表示加速图形端口 (AGP) 驱动程序检测到违规。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

错误检查 0x116：VIDEO_TDR_FAILURE

项目 • 2024/10/16

VIDEO_TDR_FAILURE 错误检查的值为 0x00000116。 此错误检查表明重置显示驱动程序并从超时恢复的尝试失败。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

VIDEO_TDR_FAILURE 参数

展开表

参数	说明
1	指向内部 TDR 恢复上下文的指针（如果可用）。
2	指向负责的设备驱动程序模块（例如所有者标记）的指针。
3	上次失败操作的错误代码（如果可用）。
4	内部上下文相关数据（如果可用）。

原因

如果在处理最终用户命令或操作时系统出现完全冻结或挂起，图形中会出现常见的稳定性问题。 通常，GPU 忙于处理密集型图形操作，通常是在游戏过程中。 没有屏幕更新发生，用户假定其系统已冻结。 用户通常等待几秒钟，然后按电源按钮重新启动系统。 Windows 会尝试检测这些有问题的挂起情况，并动态恢复响应式桌面。

检测和恢复过程称为超时检测和恢复 (TDR)。 默认超时值为 2 秒。 在视频卡的 TDR 进程中，操作系统的 GPU 计划程序调用显示微型端口驱动程序的 DxgkDdiResetFromTimeout 函数来重新初始化驱动程序并重置 GPU。

在此过程中，操作系统告知驱动程序不要访问硬件或内存，并提供较短的时间来完成为当前正在运行的线程。 如果线程未在超时内完成，则系统错误检查结果为 0x116 VIDEO_TDR_FAILURE。 有关详细信息，请参阅线程同步和 TDR。

如果短时间内发生多个 TDR 事件，系统还可以使用 VIDEO_TDR_FAILURE 进行错误检查。 默认值为一分钟内超过五个 TDR。

如果恢复过程成功，将显示一条消息来表明“显示驱动程序停止响应并已恢复”。

有关详细信息，请参阅超时检测和恢复 (TDR)、TDR 注册表项和 Windows 8 及更高版本中的 TDR 更改。

解决方法

GPU 向显示器显示图形所需的时间比允许的时间要长。 以下一种或多种原因可能会导致此行为：

- 您可能需要为显示驱动程序安装最新更新，以便它正确支持 TDR 进程。
- 影响显卡正常运行能力的硬件问题，包括：
  - 超频组件，例如主板
  - 组件兼容性和设置不正确（尤其是内存配置和计时）
  - 系统冷却不足
  - 系统电源不足
  - 有缺陷的部件（内存模块、主板等）
- 视觉效果或后台运行的程序过多可能会降低电脑的速度，因此显卡无法根据需要做出响应。

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

dbgcmd

```
1: kd> !analyze -v
*******************************************************************************
***
*
*
*                        Bugcheck Analysis
*
*
*
*******************************************************************************
***

VIDEO_TDR_FAILURE (116)
Attempt to reset the display driver and recover from timeout failed.
Arguments:
Arg1: ffffe000c2c404c0, Optional pointer to internal TDR recovery context
(TDR_RECOVERY_CONTEXT).
Arg2: fffff8016470c14c, The pointer into responsible device driver module
(e.g. owner tag).
Arg3: ffffffffc000009a, Optional error code (NTSTATUS) of the last failed
operation.
```

```
Arg4: 0000000000000004, Optional internal context dependent data.

...
```

还会显示故障模块名称。

dbgcmd

```
MODULE_NAME: nvlddmkm

IMAGE_NAME:  nvlddmkm.sys
```

可以使用 lm (List Loaded Modules) 命令显示有关故障驱动程序的信息，包括时间戳。

dbgcmd

```
1: kd> lmvm nvlddmkm
Browse full module list
start             end                 module name
fffff801`63ec0000 fffff801`649a7000   nvlddmkm T (no symbols)
    Loaded symbol image file: nvlddmkm.sys
    Image path: \SystemRoot\system32\DRIVERS\nvlddmkm.sys
    Image name: nvlddmkm.sys
    Browse all global symbols  functions  data
    Timestamp:        Wed Jul  8 15:43:44 2015 (559DA7A0)
    CheckSum:         00AA7491
    ImageSize:        00AE7000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4
```

参数 1 包含指向 TDR_RECOVERY_CONTEXT 的指针。 如 !analyze 输出中所示，如果有关联代码的符号，可以使用 `dt` 命令来显示此数据。

dbgcmd

```
1: kd> dt dxgkrnl!_TDR_RECOVERY_CONTEXT ffffe000c2c404c0
   +0x000 Signature        : 0x52445476
   +0x008 pState           : 0xffffe000`c2b12a40 ??
   +0x010 TimeoutReason    : 9 ( TdrEngineTimeoutPromotedToAdapterReset )
   +0x018 Tick             : _ULARGE_INTEGER 0xb2
   +0x020 pAdapter         : 0xffffe000`c2a89010 DXGADAPTER
   +0x028 pVidSchContext   : (null)
   +0x030 GPUTimeoutData   : _TDR_RECOVERY_GPU_DATA
   +0x048 CrtcTimeoutData  : _TDR_RECOVERY_CONTEXT::<unnamed-type-
CrtcTimeoutData>
   +0x050 pProcessName     : (null)
   +0x058 DbgOwnerTag      : 0xfffff801`6470c14c
   +0x060 PrivateDbgInfo   : _TDR_DEBUG_REPORT_PRIVATE_INFO
   +0xb00 pDbgReport       : 0xffffe000`c2c3f750 _WD_DEBUG_REPORT
   +0xb08 pDbgBuffer       : 0xffffc000`bd000000 Void
   +0xb10 DbgBufferSize    : 0x37515
```

```
   +0xb18 pDumpBufferHelper : (null)
   +0xb20 pDbgInfoExtension : 0xffffc000`ba7e47a0
_DXGKARG_COLLECTDBGINFO_EXT
   +0xb28 pDbgBufferUpdatePrivateInfo : 0xffffc000`bd000140 Void
   +0xb30 ReferenceCount   : 0n1
   +0xb38 pResetCompletedEvent : (null)
```

参数 2 包含指向负责的设备驱动程序模块（例如所有者标记）的指针。

dbgcmd

```
1: kd> ub fffff8016470c14c
nvlddmkm+0x84c132:
fffff801`6470c132 cc              int     3
fffff801`6470c133 cc              int     3
fffff801`6470c134 48ff254d2deaff  jmp     qword ptr [nvlddmkm+0x6eee88
(fffff801`645aee88)]
fffff801`6470c13b cc              int     3
fffff801`6470c13c 48ff252d2eeaff  jmp     qword ptr [nvlddmkm+0x6eef70
(fffff801`645aef70)]
fffff801`6470c143 cc              int     3
fffff801`6470c144 48ff257d2deaff  jmp     qword ptr [nvlddmkm+0x6eeec8
(fffff801`645aeec8)]
fffff801`6470c14b cc              int     3
```

你可能希望使用 k、kb、kc、kd、kp、kP、kv（显示堆栈回溯）命令检查堆栈跟踪。

dbgcmd

```
1: kd> k
 # Child-SP          RetAddr           Call Site
00 ffffd001`7d53d918 fffff801`61ba2b4c nt!KeBugCheckEx
[d:\th\minkernel\ntos\ke\amd64\procstat.asm @ 122]
01 ffffd001`7d53d920 fffff801`61b8da0e dxgkrnl!TdrBugcheckOnTimeout+0xec
[d:\th\windows\core\dxkernel\dxgkrnl\core\dxgtdr.cxx @ 2731]
02 ffffd001`7d53d960 fffff801`61b8dd7f dxgkrnl!ADAPTER_RENDER::Reset+0x15e
[d:\th\windows\core\dxkernel\dxgkrnl\core\adapter.cxx @ 19443]
03 ffffd001`7d53d990 fffff801`61ba2385 dxgkrnl!DXGADAPTER::Reset+0x177
[d:\th\windows\core\dxkernel\dxgkrnl\core\adapter.cxx @ 19316]
04 ffffd001`7d53d9e0 fffff801`63c5fba7 dxgkrnl!TdrResetFromTimeout+0x15
[d:\th\windows\core\dxkernel\dxgkrnl\core\dxgtdr.cxx @ 2554]
05 ffffd001`7d53da10 fffff801`63c47e5d dxgmms1!VidSchiRecoverFromTDR+0x11b
[d:\th\windows\core\dxkernel\dxgkrnl\dxgmms1\vidsch\vidscher.cxx @ 1055]
06 ffffd001`7d53dbc0 fffff801`aa55c698 dxgmms1!VidSchiWorkerThread+0x8d
[d:\th\windows\core\dxkernel\dxgkrnl\dxgmms1\vidsch\vidschi.cxx @ 426]
07 ffffd001`7d53dc00 fffff801`aa5c9306 nt!PspSystemThreadStartup+0x58
[d:\th\minkernel\ntos\ps\psexec.c @ 6845]
08 ffffd001`7d53dc60 00000000`00000000 nt!KxStartSystemThread+0x16
[d:\th\minkernel\ntos\ke\amd64\threadbg.asm @ 80]
```

如果可以一致地重现停止代码，还可以在导致此停止代码的代码中设置断点，并尝试单步进入错误代码。

有关详细信息，请参阅 使用 WinDbg 分析故障转储文件。

如果您不具备使用 Windows 调试器来解决此问题的能力，则可以使用一些基本的故障排除技术。

- 在事件查看器中检查系统日志以获取其他错误消息，这可能有助于识别导致此错误检查的设备或驱动程序。
- 如果在错误检查消息中标识了驱动程序，请禁用该驱动程序，或与制造商联系以获取驱动程序更新。
- 验证所有与图形相关的软件（如 DirectX 和 OpenGL）是否都是最新的，并且所有图形密集型应用程序（如游戏）都已完全修补。
- 确保安装的任何新硬件都与已安装的 Windows 版本兼容。 例如，可以在 Windows 10 规范 中获取所需硬件的相关信息。
- 运行 Windows 内存诊断工具以测试内存。 在控制面板搜索框中键入**内存**，然后选择**诊断计算机内存问题**。运行测试后，使用事件查看器查看系统日志下的结果。 查找“内存诊断结果”条目以查看结果。
- 你可尝试运行系统制造商提供的硬件诊断。
- 使用安全模式

  请考虑使用安全模式来帮助隔离此问题。 在 Windows 启动过程中，使用安全模式只会加载所需的最低驱动程序和系统服务。

  1. 若要进入安全模式，请转到设置中的**更新和安全**。
  2. 选择**恢复**>**高级启动**以启动进入维护模式。
  3. 在生成的菜单中，选择**故障排除**>**高级选项**>**启动设置**>**重启**。
  4. Windows 重启进入“**启动设置**”界面后，选择选项 4、5 或 6 以启动进入安全模式。

  在启动时按下功能键（例如 F8）可以进入安全模式。 有关具体的启动选项，请参阅制造商提供的信息。

有关一般故障排除信息，请参阅分析错误检查蓝屏数据。

注解

有关硬件设备在实现 TDR 时必须满足的要求的信息，请参阅 Windows 硬件实验室工具包文档。 例如，TDR2 - 标准双设备测试图形。

另请参阅

错误检查代码参考

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Bug 检查 0x117：VIDEO_TDR_TIMEOUT_DETECTED

项目 • 2024/02/14

VIDEO_TDR_TIMEOUT_DETECTED实时转储的值为0x00000117。 这表示显示驱动程序无法及时响应。

（此代码永远不能用于真正的 bug 检查;它用于标识实时转储。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

VIDEO_TDR_TIMEOUT_DETECTED参数

展开表

参数	说明
1	指向内部 TDR 恢复上下文的指针（如果可用）。
2	指向负责任的设备驱动程序模块（例如所有者标记）的指针。
3	特定于辅助驱动程序的存储桶密钥。
4	内部上下文相关数据（如果可用）。

原因

当系统在处理最终用户命令或操作时出现完全冻结或挂起时，图形中会出现常见的稳定性问题。 通常，GPU 正忙于处理密集的图形操作，通常在游戏期间。 不会发生屏幕更新，用户假定其系统已冻结。 用户通常等待几秒钟，然后按电源按钮重新启动系统。 Windows 会尝试检测这些有问题的挂起情况，并动态恢复响应式桌面。

检测和恢复过程称为超时检测和恢复（TDR）。 默认超时为 2 秒。 在视频卡的 TDR 进程中，操作系统的 GPU 计划程序调用显示微型端口驱动程序的 *DxgkDdiResetFromTimeout* 函数以重新初始化驱动程序并重置 GPU。

如果恢复过程成功，将显示一条消息，指示“显示驱动程序已停止响应并已恢复”。

有关详细信息，请参阅 Windows 8 中的超时检测和恢复（TDR）、TDR 注册表项和 TDR 更改，这些更改位于超时检测和恢复中（TDR）

解决方法

GPU 花费的时间比允许向监视器显示图形所需的时间要长。 以下一种或多种原因可能会导致此行为：

- 可能需要安装显示驱动程序的最新更新，以便它正确支持 TDR 过程。
- 影响视频卡正常运行的硬件问题，包括：
  - 加时制组件，如主板
  - 组件兼容性和设置不正确（尤其是内存配置和计时）
  - 系统冷却不足
  - 系统电源不足
  - 有缺陷的部分（内存模块、主板等）
- 在后台运行的视觉效果或过多的程序可能会降低电脑的速度，以便视频卡无法根据需要做出响应。

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

dbgcmd

```
3: kd> !analyze -v
*******************************************************************************
***
*
*
*                        Bugcheck Analysis
*
*
*
*******************************************************************************
***

VIDEO_TDR_TIMEOUT_DETECTED (117)
The display driver failed to respond in timely fashion.
(This code can never be used for a real bug check; it is used to identify
live dumps.)
Arguments:
Arg1: 8975d500, Optional pointer to internal TDR recovery context
(TDR_RECOVERY_CONTEXT).
Arg2: 9a02381e, The pointer into responsible device driver module (e.g owner
tag).
Arg3: 00000000, The secondary driver specific bucketing key.
Arg4: 00000000, Optional internal context dependent data.
```

```
...
```

此外，还会显示故障模块名称

dbgcmd

```
MODULE_NAME: atikmpag

IMAGE_NAME:  atikmpag.sys
```

可以使用 lmv 命令显示有关故障驱动程序的信息，包括时间戳。

dbgcmd

```
3: kd> lmvm atikmpag
Browse full module list
start    end        module name
9a01a000 9a09a000   atikmpag T (no symbols)
    Loaded symbol image file: atikmpag.sys
    Image path: atikmpag.sys
    Image name: atikmpag.sys
    Browse all global symbols  functions  data
    Timestamp:        Fri Dec  6 12:20:32 2013 (52A23190)
    CheckSum:         0007E58A
    ImageSize:        00080000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4
```

参数 1 包含指向TDR_RECOVERY_CONTEXT的指针。

dbgcmd

```
3: kd> dt dxgkrnl!_TDR_RECOVERY_CONTEXT fffffa8010041010
   +0x000 Signature        : ??
   +0x004 pState           : ????
   +0x008 TimeoutReason    : ??
   +0x010 Tick             : _ULARGE_INTEGER
   +0x018 pAdapter         : ????
   +0x01c pVidSchContext   : ????
   +0x020 GPUTimeoutData   : _TDR_RECOVERY_GPU_DATA
   +0x038 CrtcTimeoutData  : _TDR_RECOVERY_CONTEXT::<unnamed-type-
CrtcTimeoutData>
   +0x040 DbgOwnerTag      : ??
   +0x048 PrivateDbgInfo   : _TDR_DEBUG_REPORT_PRIVATE_INFO
   +0xae0 pDbgReport       : ????
   +0xae4 pDbgBuffer       : ????
   +0xae8 DbgBufferSize    : ??
   +0xaec pDumpBufferHelper : ????
   +0xaf0 pDbgInfoExtension : ????
   +0xaf4 pDbgBufferUpdatePrivateInfo : ????
```

```
   +0xaf8 ReferenceCount   : ??
Memory read error 10041b08
```

参数 2 包含指向负责任的设备驱动程序模块（例如所有者标记）的指针。

dbgcmd

```
BUGCHECK_P2: ffffffff9a02381e
```

你可能希望使用 **k、kb、kc、kd、kp、kP、kv（显示堆栈回溯）** 命令检查堆栈跟踪。

dbgcmd

```
3: kd> k
 # ChildEBP RetAddr
00 81d9ace0 976e605e dxgkrnl!TdrUpdateDbgReport+0x93
[d:\blue_gdr\windows\core\dxkernel\dxgkrnl\core\dxgtdr.cxx @ 944]
01 81d9acfc 976ddead dxgkrnl!TdrCollectDbgInfoStage2+0x195
[d:\blue_gdr\windows\core\dxkernel\dxgkrnl\core\dxgtdr.cxx @ 1759]
02 81d9ad24 976e664f dxgkrnl!DXGADAPTER::Reset+0x23f
[d:\blue_gdr\windows\core\dxkernel\dxgkrnl\core\adapter.cxx @ 14972]
03 81d9ad3c 977be9e0 dxgkrnl!TdrResetFromTimeout+0x16
[d:\blue_gdr\windows\core\dxkernel\dxgkrnl\core\dxgtdr.cxx @ 2465]
04 81d9ad50 977b7518 dxgmms1!VidSchiRecoverFromTDR+0x13
[d:\blue_gdr\windows\core\dxkernel\dxgkrnl\dxgmms1\vidsch\vidscher.cxx @
1018]
05 (Inline) -------- dxgmms1!VidSchiRun_PriorityTable+0xfa71
06 81d9ad70 812c01d4 dxgmms1!VidSchiWorkerThread+0xfaf2
[d:\blue_gdr\windows\core\dxkernel\dxgkrnl\dxgmms1\vidsch\vidschi.cxx @ 424]
07 81d9adb0 81325fb1 nt!PspSystemThreadStartup+0x58
[d:\blue_gdr\minkernel\ntos\ps\psexec.c @ 5884]
08 81d9adbc 00000000 nt!KiThreadStartup+0x15
[d:\blue_gdr\minkernel\ntos\ke\i386\threadbg.asm @ 81]
```

如果可以一致地重现停止代码，还可以在代码中设置断点，然后尝试向前单步执行故障代码。

有关详细信息，请参阅以下主题：

使用 Windows 调试器（WinDbg）进行故障转储分析

如果你没有能力使用 Windows 调试器来解决此问题，则可以使用一些基本的故障排除技术。

- 检查系统登录事件查看器是否有其他错误消息，这些错误消息可能有助于识别导致此 bug 的设备或驱动程序检查。

- 如果在 bug 检查 消息中标识驱动程序，请禁用驱动程序或检查制造商进行驱动程序更新。
- 验证 DirectX 和 OpenGL 等所有图形相关软件是否都是最新的，并且所有图形密集型应用程序（如游戏）都已完全修补。
- 确认安装的任何新硬件都与已安装的 Windows 版本兼容。 例如，可以在 Windows 10 规范中 获取有关所需硬件的信息。
- **使用保险箱模式**

  请考虑使用 保险箱 模式来帮助隔离此问题。 使用 保险箱 模式只会在 Windows 启动时加载所需的最低驱动程序和系统服务。 若要进入保险箱模式，请在设置中使用 **更新和安全性**。 选择 “**恢复**高级>**启动** ”以启动到维护模式。 在生成的菜单中，选择 “**故障排除**->**高级选项**->**启动**”**设置** ->**Restart**。 Windows 重启到“**启动设置**”屏幕后，选择选项 4、5 或 6 以启动到保险箱模式。

  保险箱模式可以通过在启动时按函数键（例如 F8）来使用。 有关特定启动选项，请参阅制造商的信息。
- 运行 Windows 内存诊断工具以测试内存。 在控制面板搜索框中，键入“内存”，然后选择“ **诊断计算机的内存问题**”。运行测试后，使用事件查看器查看系统日志下的结果。 查找“内存诊断结果”条目以查看结果。
- 你可尝试运行系统制造商提供的硬件诊断。
- 有关其他常规故障排除信息，请参阅 **蓝屏数据**。

注解

硬件认证要求

有关硬件设备在实现 TDR 时必须满足的要求的信息，请参阅 Device.Graphics 上的 *WHCK 文档..TDRResiliency*。

Bug 检查 0x119：VIDEO_SCHEDULER_INTERNAL_ERROR

项目 • 2024/02/14

VIDEO_SCHEDULER_INTERNAL_ERROR bug 检查的值为 0x00000119。 此 bug 检查指示视频计划程序检测到严重冲突。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

VIDEO_SCHEDULER_INTERNAL_ERROR参数

参数 1 是唯一感兴趣的参数。 它标识确切的冲突。

展开表

参数 1	错误原因
0x1	驱动程序报告了无效的围栏 ID。 （DRIVER_REPORTED_INVALID_FENCE_ID）
0x2	提交命令后驱动程序失败。 （DRIVER_FAILED_SUBMIT_COMMAND）
0x3	修补命令缓冲区时驱动程序失败。 （DRIVER_FAILED_PATCH_COMMAND）
0x4	驱动程序报告了无效的翻转功能。 （DRIVER_INVALID_FLIPQUEUE_LENGTH）
0x5	驱动程序失败了系统或分页命令。 （DRIVER_FAULTED_SYSTEM_COMMAND）
0x6	驱动程序报告 NULL PhysicalAdapterMask，以便在多适配器 GPU 上引发中断。 （DRIVER_INVALID_ADAPTER_MASK）
0x7	驱动程序报表仅在呈现适配器上显示 VSync。 （REPORT_VSYNC_ON_RENDER_ONLY_ADAPTER）
0x8	导致重置的驱动程序节点没有相应的位集。 （INVALID_NODE_MASK）
0x9	取消命令上的驱动程序失败。 （FAILED_CANCEL_COMMAND）
0xA	驱动程序报告了无效的范围中止的围栏。 （REPORTED_INVALID_ABORTED_FENCE）
0xB	驱动程序未能通过 SetVidPnSourceAddressWithMultiPlaneOverlay 命令。 （FAILED_标准版TVIDPNSOURCEMPO_COMMAND）

参数 1	错误原因
0xC	驱动程序 PageFaultFlags 指示硬件故障严重。 （FATAL_PAGE_FAULT）
0xD	驱动程序报告了 DMA 页面错误中断中的 PageFaultFlags 错误。 （INVALID_DMA_FAULT_PARAMETERS）
0xE	驱动程序报告了大于 GPU 上最大覆盖平面的平面索引的 VSync。 （REPORT_VSYNC_PLANE_OUT_OF_RANGE）
0xF	未使用。 （FAILED_POSTMPOPRE标准版NT_COMMAND）
0x10	驱动程序在引擎重置后和 OS 恢复引擎之前报告了意外中断。 （UNEXPECTED_INTERRUPT_AFTER_RE标准版T）
0x11	驱动程序报告 OS 尚未生成的挂起上下文已完成围栏值。 （INCORRECT_SUSPEND_FENCE）
0x12	驱动程序无法更新本机围栏监视值。 （FAILED_UPDATE_MONITORED_VALUE）
0x13	驱动程序无法更新本机围栏当前值。 （FAILED_UPDATE_CURRENT_VALUE）
0x400	这是内部 OS 状态错误，通常是由内存损坏或硬件损坏引起的。
0xE00	OS 内存不足，用于预分配的数据包来处理被动翻转请求。
0x1000	这是内部 OS 状态错误，通常是由内存损坏或硬件损坏引起的。
0xA000	这是内部 OS 状态错误，通常是由内存损坏或硬件损坏引起的。
0x10000	这是内部 OS 状态错误，通常是由内存损坏或硬件损坏引起的。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

如果！ analyze **输出中列出的**故障模块是视频驱动程序，请参阅供应商提供的该视频驱动程序是否有更新。

另请参阅

- 处理命令和 DMA 缓冲区
- 提交命令缓冲区
- 提供围栏标识符
- 视频内存管理和 GPU 计划

- 直接翻转视频内存

Bug 检查 0x11A：EM_INITIALIZATION_FAILURE

项目 • 2024/03/01

EM_INITIALIZATION_FAILURE bug 检查的值为 0x0000011A。

此错误检查很少出现。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

Bug 检查 0x11B：DRIVER_RETURNED_HOLDING_CANCEL_LOCK

项目 • 2024/03/01

DRIVER_RETURNED_HOLDING_CANCEL_LOCK Bug 检查的值为 0x0000011B。 此 Bug 检查表明驱动程序已从持有全局取消锁的*取消*例程返回。 这会导致后面的所有取消调用失败，并导致死锁或其他 Bug 检查。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_RETURNED_HOLDING_CANCEL_LOCK 参数

展开表

参数	说明
1	已取消的 IRP 的地址（可能无效）。
2	*取消*例程的地址。

注解

取消旋转锁应由*取消*例程释放。

驱动程序调用 IoCancelIrpIoCancelIrp 函数来取消单个 I/O 请求数据包 (IRP)。 此函数获取取消旋转锁，在 IRP 中设置取消标志，然后在指定例程时调用 IRP 中相应字段指定的*取消*例程。 *取消*例程应释放取消旋转锁。 如果没有*取消*例程，则会释放取消旋转锁。

bug 检查0x11C：ATTEMPTED_WRITE_TO_CM_PROTECTED_STORAGE

项目 • 2024/02/14

ATTEMPTED_WRITE_TO_CM_PROTECTED_STORAGE bug 检查的值为 0x0000011C。 此 bug 检查指示尝试写入配置管理器的只读受保护存储。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

ATTEMPTED_WRITE_TO_CM_PROTECTED_STORAGE 参数

展开表

参数	说明
1	尝试写入的虚拟地址
2	PTE 内容
3	Reserved
4	Reserved

注解

如果可能，尝试写入操作的驱动程序的名称将打印为 bug 检查 屏幕上的 Unicode 字符串，然后保存在 KiBugCheckDriver 中。

Bug 检查 0x11D：EVENT_TRACING_FATAL_ERROR

项目 • 2024/02/14

EVENT_TRACING_FATAL_ERROR bug 检查 的值为 0x0000011D。 此 bug 检查指示事件跟踪子系统遇到意外的致命错误。

> ⓘ **重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

解决方法

在内核调试器中，**使用！ analyze -v** 命令执行初始 bug 检查分析。 参数 1 将列出 bug 的子类型检查。

0x01：无法初始化安全性。

0x02：无法初始化处理器。

0x03：内核模式注册损坏。

0x04：注销中的句柄无效。

0x05：EventWrite 调用中的数据溢出。

0x06：事件已丢失。

0x07：跟踪缓冲区损坏。

0x08：无法为 ETW LoggerContext 分配缓存感知的运行保护。 参数 2 将包含记录器 ID。

0x09：ETW GuidEntry 的引用计数对于对象的当前状态是非法的。 参数 2 将包含指向 ETW_GUID_ENTRY的指针。

另请参阅

使用 Windows 调试器（WinDbg）进行故障转储分析

使用 WinDbg 分析内核模式转储文件

Bug 检查 0x11E: TOO_MANY_RECURSIVE_FAULTS

项目 • 2024/03/01

TOO_MANY_RECURSIVE_FAULTS Bug 检查的值为 0x0000011E。 这表示文件系统在资源不足的情况下导致过多的递归错误而无法处理。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

TOO_MANY_RECURSIVE_FAULTS 参数

无

Bug 检查 0x11F：INVALID_DRIVER_HANDLE

项目 • 2024/03/01

INVALID_DRIVER_HANDLE Bug 检查的值为 0x0000011F。 这表示在插入驱动程序对象和引用句柄之间，有人关闭了驱动程序的初始句柄。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INVALID_DRIVER_HANDLE 参数

展开表

参数	说明
1	驱动程序对象的句柄值。
2	尝试引用对象时返回的状态。
3	PDRIVER_OBJECT 的地址。
4	Reserved

Bug 检查：0x120 BITLOCKER_FATAL_ERROR

项目 • 2024/03/01

BITLOCKER_FATAL_ERROR Bug 检查的值为 0x00000120。 这表明 BitLocker 驱动器加密遇到无法恢复的问题。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

BITLOCKER_FATAL_ERROR 参数

展开表

参数	说明
1	问题类型
2	Reserved
3	Reserved
4	Reserved

Bug 检查 0x121：DRIVER_VIOLATION

项目 • 2024/03/01

DRIVER_VIOLATION Bug 检查的值为 0x00000121。 此 Bug 检查表明驱动程序导致了违规。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_VIOLATION 参数

参数 1 表示违规类型。

展开表

参数 1	参数 2	参数 3	参数 4	原因
0x1	当前 IRQL。	所需的 IRQL。	Reserved	驱动程序调用了只能在特定 IRQL 中调用的函数。
0x2	当前 IRQL。	允许的最大 IRQL。	Reserved	驱动程序调用函数的 IRQL 大于函数可调用的 IRQL。

注解

使用内核调试器并查看调用堆栈来确定导致违规的驱动程序的名称：**!analyze** 调试扩展显示有关 Bug 检查的信息，并有助于确定根本原因，然后输入其中一个 **k（显示堆栈回溯）**命令以查看调用堆栈。

Bug 检查 0x122：WHEA_INTERNAL_ERROR

项目 • 2024/02/14

WHEA_INTERNAL_ERROR bug 检查的值为 0x00000122。 此 bug 检查指示 Windows 硬件错误体系结构（WHEA）中发生了内部错误。 错误可能是供应商提供的特定于平台的硬件错误驱动程序（PSHED）插件的实现、错误记录的固件实现或错误注入的固件实现导致的。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

WHEA_INTERNAL_ERROR参数

展开表

参数 1	参数 2	参数 3	参数 4	错误原因
0x1	内存大小	错误源计数	0	未能为硬件错误源表中的所有错误源分配足够的内存。
0x2	处理器数目	0	0	未能为每个处理器分配足够的内存，以便为 WHEA 信息块分配足够的内存。
0x5	Status	阶段（bug 的初始化阶段检查）	0	WHEA 无法为错误源分配足够的内存，或者错误源枚举失败。
0x6	Status	阶段	错误源类型	未能在参数 3 指定的阶段初始化错误源（参数 4）。
0x7	Status	0	0	未能分配足够的内存。

参数 1	参数 2	参数 3	参数 4	错误原因
0x8	错误源数	0	0	无法为所有错误源描述符分配足够的内存。
0x9	错误源类型	源 ID	0	WHEA 从无效的错误源收到未更正的错误源。
0xA	错误源类型	源 ID	0	未能为未更正的错误分配错误记录。
0xB	错误源类型	源 ID	0	未能为未更正的错误填充错误记录。

注解

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

如果参数 1 等于 0x6、0x9、0xA 或 0xB，则其他参数之一包含错误源类型。 下表提供了错误源类型的可能值。

展开表

值	说明
0x00	计算机检查异常
0x01	更正的计算机检查
0x02	更正了平台错误
0x03	不可屏蔽的中断
0x04	PCI 快速错误
0x05	其他类型的错误源/泛型
0x06	IA64 INIT 错误源
0x07	BOOT 错误源
0x08	基于 SCI 的泛型错误源
0x09	Itanium 计算机检查中止

值	说明
0x0A	Itanium 计算机检查
0x0B	Itanium 更正了平台错误

Bug 检查 0x123：CRYPTO_SELF_TEST_FAILURE

项目 • 2024/03/01

The CRYPTO_SELF_TEST_FAILURE Bug 检查的值为 0x00000123。 这表示加密子系统在启动期间未能通过强制算法自测试。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

CRYPTO_SELF_TEST_FAILURE 参数

无

错误检查 0x124：WHEA_UNCORRECTABLE_ERROR

项目 • 2024/10/16

WHEA_UNCORRECTABLE_ERROR 错误检查的值为 0x00000124，表明发生了致命的硬件错误。 此错误检查使用 Windows 硬件错误体系结构 (WHEA) 提供的错误数据。

若要识别错误的特定原因，通常需要了解 WHEA_ERROR_RECORD 结构。 有关详细信息，请参阅本文的备注部分。

> **重要**
>
> 本主题是面向程序员的。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

参数

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x0	WHEA_ERROR_RECORD 结构的地址	出现错误的 MCA 寄存器的 MCi_STATUS MSR 高 32 位	出现错误的 MCA 寄存器的 MCi_STATUS MSR 低 32 位	发生计算机检查异常。 如果处理器基于 x64 体系结构，或者具有 MCA 功能的 x86 体系结构（例如 Intel Processorum Pro、Pentum IV 或 Xeon），则这些参数说明适用。
0x1	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	发生经更正的计算机检查异常。
0x2	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	出现已更正的平台错误。
0x3	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	发生了不可屏蔽的中断 (NMI) 错误。
0x4	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	出现无法更正的 PCI Express 错误。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x5	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	发生通用硬件错误。
0x6	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	出现初始化错误。
0x7	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	出现 BOOT 错误。
0x8	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	发生了可扩展的一致接口 (SCI) 一般错误。
0x9	WHEA_ERROR_RECORD 结构的地址	SAL 日志的长度（以字节为单位）	SAL 的地址	发生不可更正的基于 Itanium 的计算机检查中止错误。
0xA	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	发生已更正的基于 Itanium 的计算机检查错误。
0xB	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	已更正的 Itanium 平台错误。
0xC	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	其他类型的错误源 v2。
0xD	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	基于 SCI 的 GHESv2（ACPI 通用硬件错误源）。
0xE	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	BMC（主板管理控制器）错误信息。
0xF	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	ARS PMEM（地址范围清理持性内存）错误源。
0x10	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	设备驱动程序错误源。
0x11	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	Arm 同步外部中止。
0x12	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	Arm SError 中断。

原因

此错误检查通常与物理硬件故障相关。 它可能与热量有关，也可能是硬件和内存缺陷，甚至是开始发生故障或已经发生故障的处理器造成的。 如果已启用过时钟，请尝试禁用它。 确认所有冷却系统（如风扇）都正常运行。 运行系统诊断以确认系统内存没有故障。 驱动程序导致硬件无法通过错误检查的可能性较小，但也有可能。

若要了解常规错误检查故障排除的更多信息，请参阅分析错误检查蓝屏数据。

注解

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

- 参数 1 标识报告错误的错误源的类型。
- 参数 2 保存描述错误条件的 WHEA_ERROR_RECORD 结构的地址。

发生硬件错误时，WHEA 会创建错误记录来存储与硬件错误条件相关的错误信息。 每个错误记录都由 WHEA_ERROR_RECORD 结构描述。 Windows 内核将错误记录包含在它为响应错误而引发的 Windows 事件跟踪 (ETW) 硬件错误事件中，以便错误记录保存在系统事件日志中。 WHEA 使用的错误记录格式基于统一可扩展固件接口 (UEFI) 规范版本 2.2 附录 N 中所述通用平台错误记录。 有关详细信息，请参阅 WHEA_ERROR_RECORD 和 Windows 硬件错误体系结构 (WHEA)。

可以使用 !errrec address 通过参数 2 中提供的地址显示 WHEA_ERROR_RECORD 结构。 !whea 和 !errpkt 扩展可用于显示其他 WHEA 信息。

有关详细信息，请参阅以下文章：

使用 Windows 调试器 (WinDbg) 进行故障转储分析

使用 WinDbg 分析内核模式转储文件

使用 !analyze 扩展 和 !analyze

Windows Vista 之前的 Windows 版本不支持此错误检查。 而是通过错误检查 0x9C 报告计算机检查异常。

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

Bug 检查 0x125：NMR_INVALID_STATE

项目 • 2024/03/01

NMR_INVALID_STATE Bug 检查的值为 0x00000125。 这表示 NMR（网络模块注册器）检测到无效状态。 状态类型请参阅参数 1。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

NMR_INVALID_STATE 参数

展开表

参数	说明
1	bugcheck 的子类型。 0x0：计算机检查异常 参数 2 - WHEA_ERROR_RECORD 结构的地址。 参数 3 - MCi_STATUS 值的 32 位高阶。 参数 4 - MCi_STATUS 值的 32 位低阶。 0x1：更正了计算机检查 参数 2 - WHEA_ERROR_RECORD 结构的地址。 0x2：更正了平台错误 参数 2 - WHEA_ERROR_RECORD 结构的地址。 0x3：不可屏蔽的中断 参数 2 - WHEA_ERROR_RECORD 结构的地址。 0x4：PCI 快速错误 参数 2 - WHEA_ERROR_RECORD 结构的地址。 0x5：常规错误 参数 2 - WHEA_ERROR_RECORD 结构的地址。 0x6：INIT 错误 参数 2 - WHEA_ERROR_RECORD 结构的地址。 0x7：BOOT 错误

参数	说明
	参数 2 - WHEA_ERROR_RECORD 结构的地址。 0x8：SCI 泛型错误 参数 2 - WHEA_ERROR_RECORD 结构的地址。 0x9：Itanium 计算机检查中止 参数 2 - WHEA_ERROR_RECORD 结构的地址。 参数 3 - 以字节为单位的 SAL 日志长度 参数 4 - SAL 日志的地址。 0xa：Itanium 更正了计算机检查 参数 2 - WHEA_ERROR_RECORD 结构的地址。 0xb：Itanium 更正了平台错误 参数 2 - WHEA_ERROR_RECORD 结构的地址。
2	指向 NMI 句柄的指针
3	指向预期类型的指针（如果可用）
4	Reserved

Bug 检查 0x126：NETIO_INVALID_POOL_CALLER

项目 • 2024/03/01

NETIO_INVALID_POOL_CALLER Bug 检查的值为 0x00000126。 这表示对 netio 托管内存池（例如 FSB 和 MDL）发出了无效的池请求。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

NETIO_INVALID_POOL_CALLER 参数

展开表

参数	说明
1	bugcheck 的子类型。 0x1：池无效。 池处于无效状态。 参数 2 - 指向内存块或 MDL 的指针。 参数 3 - 指向页面的指针。 参数 4 - 指向 CPU 池的指针。 0x2：MDL 无效。 MDL 处于无效状态。 参数 2 - 指向 MDL 的指针。 参数 3 - 指向 CPU 池的指针。 参数 4 - 指向池标头的指针。
2	请参阅参数 1
3	请参阅参数 1
4	请参阅参数 1

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

Bug 检查 0x127：PAGE_NOT_ZERO

项目 • 2024/03/08

PAGE_NOT_ZERO Bug 检查的值为 0x00000127。 此 Bug 检查表明应该填充零的页面没有填充。 此 Bug 检查可能是由于硬件错误或操作系统的特权组件在释放页面后修改了页面。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PAGE_NOT_ZERO 参数

展开表

参数	说明
1	映射损坏页面的虚拟地址
2	物理页码
3	零（保留）
4	零（保留）

bug 检查0x128：WORKER_THREAD_RETURNED_WITH_BAD_IO_PRIORITY

项目 • 2024/02/14

WORKER_THREAD_RETURNED_WITH_BAD_IO_PRIORITY bug 检查的值为 0x00000128。 这表示被调用的辅助角色例程错误地修改了工作线程 IOPriority。

> **重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

WORKER_THREAD_RETURNED_WITH_BAD_IO_PRIORITY 参数

展开表

参数	说明
1	工作线程例程的地址（使用此 **地址上的 ln（列出最近的符号）** 命令查找有问题的驱动程序）
2	当前 IoPrioirity 值
3	Workitem 参数
4	Workitem 地址

Bug 检查 0x129：WORKER_THREAD_RETURNED_WITH_BAD_PAGING_IO_PRIORITY

项目 • 2024/03/01

WORKER_THREAD_RETURNED_WITH_BAD_PAGING_IO_PRIORITY Bug 检查的值为 0x00000129。 这表示被调用的辅助角色例程错误地修改了工作线程分页 IOPriority。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WORKER_THREAD_RETURNED_WITH_BAD_PAGING_IO_PRIORITY 参数

展开表

参数	说明
1	工作例程的地址 使用此地址上的 **ln（列出最近符号）**命令查找有问题的驱动程序。
2	当前分页 IoPrioirity 值
3	Workitem 参数
4	Workitem 地址

bug 检查0x12A：MUI_NO_VALID_SYSTEM_LANGUAGE

项目 • 2024/02/14

MUI_NO_VALID_SYSTEM_LANGUAGE bug 检查的值为 0x0000012A。 这表示 Windows 未找到系统默认 UI 语言的任何已安装许可语言包。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

MUI_NO_VALID_SYSTEM_LANGUAGE参数

展开表

参数	说明
1	bug 的子类型检查 0x1：Windows 在第一阶段初始化期间未找到任何已安装的语言包。 参数 2 - 描述失败原因的 NT 状态代码。 0x2：在内核缓存创建期间，Windows 未找到系统默认 UI 语言的任何已安装许可语言包。 参数 2 - 描述失败原因的 NT 状态代码。 。
2	请参阅参数 1
3	Reserved
4	Reserved

Bug 检查 0x12B：FAULTY_HARDWARE_CORRUPTED_PAGE

项目 • 2024/03/01

FAULTY_HARDWARE_CORRUPTED_PAGE Bug 检查的值为 0x0000012B。 此 Bug 检查表示 Windows 内存管理器检测到损坏。 该损坏只能由使用物理寻址访问内存的组件导致。

> **ⓘ 重要**
>
> 本主题是面向程序员的。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

FAULTY_HARDWARE_CORRUPTED_PAGE 参数

在两种情况下，内存管理器会使用两组不同的参数引发 FAULTY_HARDWARE_CORRUPTED_PAGE Bug 检查。

如果参数 3 和 4 均为零，则 Bug 检查表示内存管理器在内存管理器预期为零的页面上检测到单位错误。

如果参数 3 和 4 为非零，则由于物理内存损坏而无法解压缩页面，压缩存储管理器会引发 Bug 检查。

内存管理器页不为零错误参数

此 Bug 检查表示在此页中发现了单位错误。 此错误是硬件内存错误。

展开表

参数	说明
1	映射损坏页面的虚拟地址
2	物理页码
3	Zero
4	Zero

压缩存储管理器错误参数

此 Bug 检查表示发生了存储管理器内存错误。 这可能是身份验证失败、CRC 失败或解压缩失败。

展开表

参数	说明
1	FailStatus - 表示失败的类型
2	正在读取的页面的压缩大小
3	源缓冲区
4	目标缓冲区

原因

只有在物理内存访问导致内存损坏时，可能会出现此错误检查。 物理内存损坏的原因包括：

- RAM 硬件缺陷。
- 驱动程序或设备通过错误的 DMA 操作或相关 MDL 不正确地修改物理页。
- 硬件设备或固件破坏内存造成的损坏，如固件在电源转换时非法修改物理页。

> **备注**
>
> 压缩存储管理器可以检测损坏是否是由单位错误引起的，并自动更正此条件，而不会引发 Bug 检查。 如果损坏不是由单位错误引起的，则压缩存储管理器将报告此 Bug 检查。

有关 Windows 内存管理器和内存压缩的详细信息，请参阅 Windows Internals 第 7 版第 1 部分。

解决方法

要调查此错误检查是否由 RAM 硬件缺陷引起，请运行 Windows 内存诊断工具。 在控制面板搜索框中键入**内存**，然后选择**诊断计算机内存问题**。运行测试后，使用事件查看器查看系统日志下的结果。 选择**内存诊断结果**条目以查看结果。

另请参阅

- 错误检查代码参考
- Windows 内核模式内存管理器

Bug 检查 0x12C：EXFAT_FILE_SYSTEM

项目 • 2024/02/21

EXFAT_FILE_SYSTEM bug 检查的值为 0x0000012C。 此 bug 检查指示扩展文件分配表 (exFAT) 文件系统中出现了问题。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

EXFAT_FILE_SYSTEM 参数

展开表

参数	说明
1	指定源文件和行号信息。 高 16 位（“0x”后面的前四个十六进制数字）按标识符号确定源文件。 低 16 位确定文件中出现 bug 的源行检查。
2	如果 **FppExceptionFilter** 位于堆栈上，则此参数指定异常记录的地址。
3	如果 **FppExceptionFilter** 位于堆栈上，则此参数指定上下文记录的地址。
4	保留。

原因

此 bug 检查是由文件系统作为最后手段造成的，当其内部会计处于不受支持的状态并且继续导致数据丢失的风险很大。 当磁盘结构损坏、磁盘扇区损坏或内存分配失败时，文件系统永远不会导致此 bug 检查。 坏扇区可能会导致错误检查，例如，当页面错误发生在内核代码或数据中并且内存管理器无法读取页面时。 但是，对于此 bug 检查，文件系统不是原因。

解决方法

若要调试此问题：将 **.cxr（显示上下文记录）**命令与参数 3 一起使用，然后使用 **kb（显示堆栈回溯）**。

bug 检查 0x12D：VOLSNAP_OVERLAPPED_TABLE_ACCESS

项目 • 2024/02/23

VOLSNAP_OVERLAPPED_TABLE_ACCESS bug 检查的值为 0x0000012D。 这表示 volsnap 试图从两个不同的线程访问公共表，这可能会导致表损坏并最终损坏该表。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

VOLSNAP_OVERLAPPED_TABLE_ACCESS 参数

无

bug 检查 0x12E：INVALID_MDL_RANGE

项目 • 2024/02/21

INVALID_MDL_RANGE bug 检查的值为 0x0000012E。 这表示驱动程序已调用 IoBuildPartialMdl() 函数，并向其传递 MDL 以映射源 MDL 的一部分，但指定的虚拟地址范围不在源 MDL 中的范围之外。 这通常是驱动程序 bug。

源和目标 MDL 以及要映射的地址范围长度是 IoBuildPartialMdl() 函数的参数，即 ;)。

C++

```
IoBuildPartialMdl(
        IN PMDL SourceMdl,
        IN OUT PMDL TargetMdl,
        IN PVOID VirtualAddress,
        IN ULONG Length
```

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅蓝屏错误疑难解答。

INVALID_MDL_RANGE 参数

展开表

参数	说明
1	SourceMdl
2	TargetMdl
3	VirtualAddress
4	长度

bug 检查 0x12F：VHD_BOOT_INITIALIZATION_FAILED

项目 • 2024/02/23

VHD_BOOT_INITIALIZATION_FAILED bug 检查的值为 0x0000012F。 这表示尝试从 VHD 启动时出现初始化失败。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

VHD_BOOT_INITIALIZATION_FAILED 参数

展开表

参数	说明
1	失败的操作 1：无法从启动设备中提取 VHD 信息。 2：等待 VHD 父设备达到表面时超时。 3：VHD 路径字符串内存分配错误。 4：VHD 路径构造失败。 5：VHD 启动设备装载失败。 6：禁用睡眠状态失败。 7：VHD 信息内存分配错误。 8：VHD 信息构造失败。
2	NT 状态代码
3	Reserved
4	Reserved

bug 检查 0x130：DYNAMIC_ADD_PROCESSOR_MISMATCH

项目 • 2024/02/23

DYNAMIC_ADD_PROCESSOR_MISMATCH bug 检查的值为 0x00000130。 此 bug 检查指示添加到系统的新处理器与当前配置不兼容。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DYNAMIC_ADD_PROCESSOR_MISMATCH 参数

无

bug 检查 0x131: INVALID_EXTENDED_PROCESSOR_STATE

项目 • 2024/02/23

INVALID_EXTENDED_PROCESSOR_STATE bug 检查的值为 0x00000131。 这表示保存或还原扩展处理器状态时检测到参数组合无效。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INVALID_EXTENDED_PROCESSOR_STATE 参数

参数 1 指示哪些有效性检查失败。

展开表

参数	说明
1	0 - 传递功能掩码无效或未启用扩展处理器状态。
2	如果启用了扩展状态，则为非零。
3	功能掩码的低 32 位。
4	功能掩码的高 32 位。

展开表

参数	说明
1	1 - 在 IRQL 上尝试保存或还原扩展状态高于 DISPATCH_LEVEL。
2	IRQL
3	Reserved
4	Reserved

展开表

参数	说明
1	2 - 以前保存的状态为等于或更高的级别。
2	保存的级别。
3	当前级别。
4	Reserved

展开表

参数	说明
1	3 - 以前保存的状态适用于其他线程。
2	保存的线程。
3	当前线程。
4	Reserved

展开表

参数	说明
1	4 - 以前保存的状态适用于不同的级别。
2	保存的级别。
3	当前级别。
4	Reserved

bug 检查 0x132：RESOURCE_OWNER_POINTER_INVALID

项目 • 2024/02/23

RESOURCE_OWNER_POINTER_INVALID bug 检查的值为 0x00000132。 这表示提供了无效的资源所有者指针。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

RESOURCE_OWNER_POINTER_INVALID 参数

展开表

参数	说明
1	资源
2	Resource->OwnerTable
3	CurrentThread
4	OwnerPointer

错误检查 0x133：DPC_WATCHDOG_VIOLATION

项目 • 2024/06/11

DPC_WATCHDOG_VIOLATION 错误检查的值为 0x00000133。 此错误检查指示 DPC 监视器已执行，要么因为它检测到单个长时间运行的延迟过程调用 (DPC)，要么是因为系统长时间在 DISPATCH_LEVEL 或更高版本的中断请求级别 (IRQL) 上花费了时间。

参数 1 的值指示单个 DPC 是否超过超时，或者系统是否在 IRQL DISPATCH_LEVEL 或更高版本中累积花费了长时间。 DPC 的运行时间不应超过 100 微秒，ISR 的运行时间不应超过 25 微秒，但系统上的实际超时值设置要高得多。

有关 DPC 的详细信息，请参阅 Pavel Yosifovich、Mark E. Russinovich、David A. Solomon 和 Alex Ionescu 编写的 DPC 对象简介和 Windows 内部资料第 7 版第 1 部分。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DPC_WATCHDOG_VIOLATION 参数

参数 1 表示违规类型。 其他参数的含义取决于*参数 1* 的值。

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0	DPC 时间计数（以刻度为单位）	DPC 时间分配（以刻度为单位）。	强制转换为 nt!DPC_WATCHDOG_GLOBAL_TRIAGE_BLOCK，其中包含有关此单个 DPC 超时的其他信息	单个 DPC 或 ISR 超过了其时间分配。 通常可以使用堆栈跟踪来标识有问题的组件。
1	监督器时段	强制转换为 nt!DPC_WATCHDOG_GLOBAL_TRIAGE_BLOCK，其中包含有关此单个 DPC 超时的其他信息	Reserved	系统在 IRQL DISPATCH_LEVEL 或更高版本上累计花费了较长的时间。 通常可以使用堆栈跟踪来标识有问题的组件。

原因

要确定原因，需要 Windows 调试器、编程体验以及访问故障模块的源代码。

有关详细信息，请参阅以下主题：

使用 Windows 调试器 (WinDbg) 进行故障转储分析

使用 WinDbg 分析内核模式转储文件

使用 !analyze 扩展 和 !analyze

有关 Windows DPC 的详细信息，请参阅由 Pavel Yosifovich、Mark E. Russinovich、David A. Solomon 和 Alex Ionescu 撰写的 Windows Internals 第 7 版第 1 部分。

示例 1

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

参数 1 = 0

在此示例中，时钟周期计数 501 超过了 500 的 DPC 时间分配。 映像名称指示在发生错误检查时正在执行此代码。

dbgcmd

```
0: kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

DPC_WATCHDOG_VIOLATION (133)
The DPC watchdog detected a prolonged run time at an IRQL of DISPATCH_LEVEL
or above.
Arguments:
Arg1: 0000000000000000, A single DPC or ISR exceeded its time allotment. The offending
    component can usually be identified with a stack trace.
Arg2: 0000000000000501, The DPC time count (in ticks).
Arg3: 0000000000000500, The DPC time allotment (in ticks).
Arg4: 0000000000000000

...

IMAGE_NAME:  BthA2DP.sys
...
```

使用以下调试器命令收集有关失败的详细信息，参数为 0：

k（显示堆栈回溯），查看停止代码发生时正在运行的代码。

你可能想要使用 u、ub、uu (Unassemble) 命令更深入地了解正在运行的代码的具体细节。

!pcr 扩展显示特定处理器上的处理器控制区域 (PCR) 的当前状态。 输出中将是 Prcb 的地址

dbgcmd

```
0: kd> !pcr
KPCR for Processor 0 at fffff8035f5a4000:
    Major 1 Minor 1
    NtTib.ExceptionList: fffff80368e77fb0
        NtTib.StackBase: fffff80368e76000
```

```
       NtTib.StackLimit: 0000000000000000
     NtTib.SubSystemTib: fffff8035f5a4000
          NtTib.Version: 000000005f5a4180
      NtTib.UserPointer: fffff8035f5a4870
          NtTib.SelfTib: 000000b6d3086000

                SelfPcr: 0000000000000000
                   Prcb: fffff8035f5a4180
                   Irql: 0000000000000000
                    IRR: 0000000000000000
                    IDR: 0000000000000000
          InterruptMode: 0000000000000000
                    IDT: 0000000000000000
                    GDT: 0000000000000000
                    TSS: 0000000000000000

          CurrentThread: fffff80364926a00
             NextThread: ffffe40b77c12040
             IdleThread: fffff80364926a00
```

可以使用 **dt（显示类型）**命令显示有关 DPC 和 DPC 监视器的其他信息。 对于地址，请使用 !pcr 输出中列出的 Prcb：

dbgcmd

```
dt nt!_KPRCB fffff80309974180 Dpc*
```

dbgcmd

```
0: kd> dt nt!DPC_WATCHDOG_GLOBAL_TRIAGE_BLOCK fffff803648fa320
   +0x000 Signature        : 0xaebecede
   +0x004 Revision         : 1
   +0x006 Size             : 0x10
   +0x008 DpcWatchdogProfileOffset : 0x84a8
   +0x00c DpcWatchdogProfileLength : 0x8200
```

示例 2

参数 1 = 1

对于参数 1，代码可能不会在代码的冒犯区域中停止。 在这种情况下，一种方法是使用事件跟踪来尝试跟踪哪个驱动程序超出了正常执行持续时间。

使用 !analyze 调试扩展显示有关错误检查的信息。

dbgcmd

```
0: kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

DPC_WATCHDOG_VIOLATION (133)
The DPC watchdog detected a prolonged run time at an IRQL of DISPATCH_LEVEL
or above.
Arguments:
```

```
Arg1: 0000000000000001, The system cumulatively spent an extended period of time at
    DISPATCH_LEVEL or above. The offending component can usually be
    identified with a stack trace.
Arg2: 0000000000001e00, The watchdog period.
Arg3: fffff803648fa320, cast to nt!DPC_WATCHDOG_GLOBAL_TRIAGE_BLOCK, which contains
    additional information regarding the cumulative timeout
Arg4: 0000000000000000
```

强制转换 nt!DPC_WATCHDOG_GLOBAL_TRIAGE_BLOCK 的地址，以显示相关信息。

dbgcmd

```
0: kd> dt nt!DPC_WATCHDOG_GLOBAL_TRIAGE_BLOCK fffff803648fa320
   +0x000 Signature        : 0xaebecede
   +0x004 Revision         : 1
   +0x006 Size             : 0x10
   +0x008 DpcWatchdogProfileOffset : 0x84a8
   +0x00c DpcWatchdogProfileLength : 0x8200
```

使用 !dpcs 命令显示排队的 DPC。

dbgcmd

```
3: kd> !dpcs
CPU Type      KDPC       Function
 0: Normal  : 0xfffff8035f5ac290 0xfffff80363e15630 nt!PpmPerfAction
Failed to read DPC at 0xffffe40b77190dd8
 0: Threaded: 0xfffff8035f5ac3d8 0xfffff80363f27d70 nt!KiDpcWatchdog
```

解决方法

要确定具体原因并创建代码修复，需要编程经验和故障模块源代码的访问权限。

注解

一般情况下，此停止代码是由在某些条件下错误驱动程序代码导致的，该代码未在分配的时间范围内完成其工作。

如果你不具备使用 Windows 调试器来解决此问题的能力，则应使用一些基本的故障排除技术。

- 如果在错误中标识了驱动程序检查消息，要隔离问题，请禁用驱动程序。 请与制造商联系以获取驱动程序更新。
- 在事件查看器中检查系统日志以获取其他错误消息，这可能有助于识别导致错误检查 0x133 的设备或驱动程序。
- 确保安装的任何新硬件都与已安装的 Windows 版本兼容。 例如，对于 Windows 10，可以在 Windows 10 规范 中获取有关所需硬件的信息。
- 有关其他常规故障排除信息，请参阅**分析错误检查蓝屏数据**。

另请参阅

使用 Windows 调试器 (WinDbg) 进行故障转储分析

使用 WinDbg 分析内核模式转储文件

错误检查代码参考

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

bug 检查 0x134：DRIVE_EXTENDER

项目 • 2024/02/21

DRIVE_EXTENDER bug 检查的值为 0x00000134。 这表示驱动器扩展程序组件经历了严重的内部错误，可防止持续系统操作。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVE_EXTENDER 参数

无

bug 检查 0x135：REGISTRY_FILTER_DRIVER_EXCEPTION

项目 • 2024/02/23

REGISTRY_FILTER_DRIVER_EXCEPTION bug 检查的值为 0x00000135。 此 bug 检查是由注册表筛选驱动程序中未经处理的异常引起的。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

REGISTRY_FILTER_DRIVER_EXCEPTION 参数

展开表

参数	说明
1	异常代码
2	导致 bug 检查的异常的上下文记录的地址
3	驱动程序的回调例程地址
4	Reserved

原因

此 bug 检查指示注册表筛选驱动程序在其通知例程中未处理异常。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。 使用第三个参数标识有问题的驱动程序。

bug 检查 0x136：VHD_BOOT_HOST_VOLUME_NOT_ENOUGH_SPACE

项目 • 2024/02/23

VHD_BOOT_HOST_VOLUME_NOT_ENOUGH_SPACE bug 检查的值为 0x00000136。 这表示尝试从 VHD 启动时出现初始化失败。 托管 VHD 的卷没有足够的可用空间来扩展 VHD。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

VHD_BOOT_HOST_VOLUME_NOT_ENOUGH_SPACE 参数

展开表

参数	说明
1	0：无法将 VHD 文件扩展到完整大小。
2	NT 状态代码
3	Reserved
4	Reserved

bug 检查 0x137：WIN32K_HANDLE_MANAGER

项目 • 2024/02/21

WIN32K_HANDLE_MANAGER bug 检查的值为 0x00000137。 这表示 win32k/ntuser 句柄管理器检测到致命错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WIN32K_HANDLE_MANAGER 参数

展开表

参数	说明
1	保留
2	对象的地址（如果可用）
3	Reserved
4	Reserved

在某些情况下，可能会返回此处未列出的其他参数。

bug 检查 0x138：GPIO_CONTROLLER_DRIVER_ERROR

项目 • 2024/02/23

GPIO_CONTROLLER_DRIVER_ERROR bug 检查的值为 0x00000138。 此 bug 检查指示 GPIO 类扩展驱动程序遇到致命错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

GPIO_CONTROLLER_DRIVER_ERROR 参数

参数 1 表示违规类型。 其他参数的含义取决于*参数 1* 的值。

展开表

参数 1	参数 2	参数 3	参数 4	错误原因
1	GSIV	预留	预留	未注册管理特定 GSIV 的 GPIO 控制器。
2	上下文值	预留	预留	客户端驱动程序为锁定或解锁请求指定了无效上下文。
3	指示是否请求关键转换。	指示该库是否由于非关键转换而已处于 F1 中。	指示该库是否由于关键转换而已处于 F1 中。	PoFx 要求 GPIO 控制器通过不适当的 F1 电源状态和/或关键转换发送库。
4	指示是否请求关键转换。	指示该库是否由于非关键转换而处于 F1 中。	指示该库是否由于关键转换而处于 F1 中。	PoFx 要求 GPIO 控制器通过不适当的 F0 电源状态和/或关键转换发送库。
5	NTSTATUS	GPIO 设备扩展	GPIO 中断参数	On-Soc GPIO 中断操作失败。

参数 1	参数 2	参数 3	参数 4	错误原因
6	NTSTATUS	GPIO 设备扩展	GPIO IO 参数	Soc GPIO IO 操作失败。
7	修订 ID	功能索引	Reserved	_DSM 方法返回格式不正确的数据。

错误检查 0x139：KERNEL_SECURITY_CHECK_FAILURE

项目 • 2024/10/16

KERNEL_SECURITY_CHECK_FAILURE 错误检查的值为 0x00000139。 此错误检查指示内核检测到关键数据结构的损坏。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

错误检查 0x139 KERNEL_SECURITY_CHECK_FAILURE 参数

展开表

参数	说明
1	损坏的类型。 有关详细信息，请参阅下表。
2	导致错误检查的异常的陷阱帧的地址
3	导致错误检查的异常的异常记录的地址
4	Reserved

下表对参数 1 的可能值进行了说明。

展开表

参数 1	说明
0	基于堆栈的缓冲区已溢出（旧版 /GS 违规）。
1	VTGuard 检测代码检测到使用非法虚拟函数表的尝试。 通常，C++ 对象已损坏，然后尝试使用损坏对象的**此**指针进行虚拟方法调用。
2	堆栈 Cookie 检测代码检测到基于堆栈的缓冲区溢出（/GS 违规）。

参数 1	说明
3	LIST_ENTRY 已损坏（例如，双重删除）。 有关详细信息，请参阅以下“原因”部分。
4	Reserved
5	将无效参数传递给一个将无效参数视为致命参数的函数。
6	堆栈 Cookie 安全 Cookie 未由加载程序正确初始化。 这可能是因为生成驱动程序仅在 Windows 8 上运行，并尝试在早期版本的 Windows 上加载驱动程序映像。 若要避免此问题，必须生成要在早期版本的 Windows 上运行的驱动程序。
7	请求了致命程序退出。
8	编译器插入的数组边界检查检测到非法数组索引操作。
9	对 **RtlQueryRegistryValues** 的调用是在无 RTL_QUERY_REGISTRY_TYPECHECK 的情况下指定 RTL_QUERY_REGISTRY_DIRECT，并且目标值不在受信任的系统配置单元中。
10	间接呼叫临界检查检测到无效的控制传输。
11	写入临界检查检测到无效的内存写入。
12	尝试切换到无效的纤程上下文。
13	已尝试分配无效的寄存器上下文。
14	对象的引用计数无效。
18	尝试切换到无效的 jmp_buf 上下文。
19	对只读数据进行了不安全的修改。
20	加密自我测试失败。
21	检测到无效的异常链。
22	发生加密库错误。
23	在 DllMain 中进行了无效的调用。
24	检测到无效的映像基址。
25	保护延迟加载导入时遇到无法恢复的失败。
26	已调用不安全的扩展。
27	调用了已弃用的服务。
28	检测到超出边界的缓冲区访问。

参数 1	说明
29	RTL_BALANCED_NODE RBTree 条目已损坏。
37	调用了超出范围的开关跳转项。
38	longjmp 尝试到无效的目标。
39	无法将导出禁止调用目标作为有效的呼叫目标。

原因

使用参数 1 表和转储文件，可以缩小此类型许多错误检查的原因。

LIST_ENTRY 损坏可能难以跟踪，并且此错误检查表明已将不一致引入到一个双重链接的列表中（在向列表中添加或删除单个列表条目元素时检测到）。 遗憾的是，在损坏发生时不一定检测到不一致，因此可能需要一些侦探工作来确定根本原因。

列表项损坏的常见原因包括：

- 驱动程序损坏了内核同步对象，例如 KEVENT（例如，在线程仍在等待同一 KEVENT 时初始化 KEVENT，或者允许基于堆栈的 KEVENT 在使用该 KEVENT 时超出范围）。 这种类型的错误检查通常发生在 nt!Ke* 或 nt!Ki* 代码。 当线程完成等待同步对象或代码尝试将同步对象置于信号状态时，可能会发生这种情况。 通常，要发出信号的同步对象是已损坏的同步对象。 有时，具有特殊池的驱动程序验证程序可以帮助跟踪罪魁祸首（如果损坏的同步对象位于已释放的池块中）。
- 驱动程序已损坏定期 KTIMER。 这种类型的错误检查通常发生在 nt!Ke* 或 nt!Ki* 代码，并涉及向计时器发出信号，或者插入或删除计时器表中的计时器。 正在操作的计时器可能是损坏的计时器，但可能需要使用 **!timer** 检查计时器表（或手动遍历计时器列表链接），以确定哪些计时器已损坏。 有时，具有特殊池的驱动程序验证程序可以帮助跟踪罪魁祸首（如果损坏的 KTIMER 位于已释放的池块中）。
- 驱动程序管理不良好的内部 LIST_ENTRY 样式链接列表。 典型的示例是在同一列表条目上调用 **RemoveEntryList** 两次，而无需重新插入两个 **RemoveEntryList** 调用之间的列表项。 其他变体可行，例如将条目插入到同一列表中。
- 驱动程序释放了包含 LIST_ENTRY 的数据结构，而不会从其相应的列表中删除数据结构，导致在旧池块重复使用后检查列表后检测到损坏。
- 驱动程序以并发方式使用 LIST_ENTRY 样式的列表，而无需进行适当的同步，从而对列表进行撕裂更新。

在大多数情况下，可以通过向前和向后走链接列表（**dl** 和 **dlb** 命令可用于此目的）和比较结果来确定损坏的数据结构。 如果列表在向前和向后走之间不一致，通常是损坏的位

置。 由于链接列表更新操作可以修改相邻元素的列表链接，因此应仔细查看损坏的列表条目的邻居，因为它们可能是基础罪魁祸首。

由于许多系统组件在内部利用 LIST_ENTRY 列表，因此使用系统 API 的驱动程序进行各种类型的资源管理不当可能会导致系统管理的链接列表损坏。

解决方法

确定此问题的原因通常需要使用调试器来收集其他信息。 应检查多个转储文件，以查看此停止代码是否具有类似的特征，例如在停止代码出现时运行的代码。

有关详细信息，请参阅使用 Windows 调试器 (WinDbg) 进行崩溃转储分析、使用 !analyze 扩展和 !analyze。

使用事件日志查看是否存在导致此停止代码的更高级别事件。

这些常规故障排除提示可能会有所帮助。

- 如果最近向系统添加了硬件，请尝试删除或替换它。 或与制造商联系，查看是否有可用的修补程序。
- 如果最近添加了新的设备驱动程序或系统服务，请尝试删除或更新它们。 尝试确定系统中导致新错误检查代码出现的原因。
- 检查事件查看器中的系统日志，以获取可能有助于查明导致错误的设备或驱动程序的其他错误消息。 有关详细信息，请参阅打开事件查看器。 在系统日志中查找与蓝屏同时出现的严重错误。
- 在**设备管理器**中查看是否有任何设备标有感叹号 (!)。 查看驱动程序属性中显示的事件日志，以查找任何故障驱动程序。 请尝试更新相关驱动程序。
- 运行病毒检测程序。 病毒可感染针对 Windows 格式化的所有类型的硬盘，由此导致的磁盘损坏可能会生成系统错误检查代码。 确保病毒检测程序检查主启动记录是否存在感染。
- 有关其他常规故障排除信息，请参阅**分析错误检查蓝屏数据**。

另请参阅

使用 Windows 调试器 (WinDbg) 进行故障转储分析

使用 WinDbg 分析内核模式转储文件

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

错误检查 0x13A：KERNEL_MODE_HEAP_CORRUPTION

项目 • 2024/10/16

KERNEL_MODE_HEAP_CORRUPTION 错误检查的值为 0x0000013A。 此错误检查表明内核模式堆管理器在堆中检测到损坏。

> **重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

KERNEL_MODE_HEAP_CORRUPTION 参数

展开表

参数	说明
1	检测到的损坏类型 - 请参阅以下列表
2	报告损坏的堆的地址
3	检测到损坏的地址
4	Reserved

参数 1 - 堆损坏的类型

0x3：检测到损坏的条目标头。

0x4：检测到多个损坏的条目标头。

0x5：检测到大型分配中的条目标头损坏。

0x6：检测到损坏，其功能与缓冲区溢出一致。

0x7：检测到损坏，其功能与缓冲区欠载一致。

0x8：将空闲块传递给仅对繁忙块有效的操作。

0x9：为当前操作指定了无效的参数。

0xA：检测到无效的分配类型。

0xB：检测到损坏，其功能与释放后使用错误一致。

0xC：为当前操作指定了错误的堆。

0xD：检测到损坏的可用列表。

0xE：堆在空闲列表以外的列表中检测到列表损坏。

0xF：将空闲块传递给仅对繁忙块有效的操作。

0x10：堆在当前操作期间检测到无效的内部状态。 这通常是缓冲区溢出的结果。

0x11：堆在当前操作期间检测到无效的内部状态。 这通常是缓冲区溢出的结果。

0x12：堆在当前操作期间检测到无效的内部状态。 这通常是缓冲区溢出的结果。

0x13：向堆 API 传递了 NULL 堆句柄。 查看调用堆栈，以确定向堆提供错误句柄的原因。

0x14：请求的堆分配大于当前分配限制。

0x15：在执行提交请求的过程中，确定该请求将超过当前的提交限制。

0x16：在检查给定 VA Manager 分配的大小的过程中，确定查询无效。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

!heap 扩展显示堆使用信息、控制堆管理器中的断点、检测泄漏的堆块、搜索堆块或显示页堆信息。

有关对蓝屏错误检查进行故障排除的一般信息，请参阅分析错误检查蓝屏数据。

另请参阅

错误检查代码参考

错误检查（蓝屏）

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

bug 检查 0x13B：PASSIVE_INTERRUPT_ERROR

项目 • 2024/02/23

PASSIVE_INTERRUPT_ERROR bug 检查的值为 0x0000013B。 这表示内核检测到被动级别中断的问题。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PASSIVE_INTERRUPT_ERROR 参数

展开表

参数	说明
1	检测到的错误类型 0x1：驱动程序尝试获取中断自旋锁，但传入了被动级别的中断对象。
2	被动级别中断的 KINTERRUPT 对象的地址。
3	Reserved
4	Reserved

bug 检查 0x13C: INVALID_IO_BOOST_STATE

项目 • 2024/02/23

INVALID_IO_BOOST_STATE bug 检查的值为 0x0000013C。 这表示线程退出时 I/O 提升状态无效。 当线程退出时，这应为零。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INVALID_IO_BOOST_STATE 参数

展开表

参数	说明
1	指向具有无效提升状态的线程的指针
2	当前提升状态或限制计数
3	Reserved
4	Reserved

bug 检查 0x13D：CRITICAL_INITIALIZATION_FAILURE

项目 • 2024/02/23

CRITICAL_INITIALIZATION_FAILURE bug 检查的值为 0x0000013D。 这表示早期内核初始化失败。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

CRITICAL_INITIALIZATION_FAILURE 参数

展开表

参数	说明
1	保留
2	Reserved
3	Reserved
4	Reserved

bug 检查 0x140：STORAGE_DEVICE_ABNORMALITY_DETECTED

项目 • 2024/02/23

STORAGE_DEVICE_ABNORMALITY_DETECTED bug 检查的值为 0x00000140。 这表示存储驱动程序堆栈遇到响应性冲突、超出阈值或其他响应失败的速率。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

STORAGE_DEVICE_ABNORMALITY_DETECTED 参数

无

bug 检查 0x143：PROCESSOR_DRIVER_INTERNAL

项目 • 2024/02/21

PROCESSOR_DRIVER_INTERNAL bug 检查的值为 0x00000143。 这表示处理器电源管理 (PPM) 驱动程序遇到致命错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PROCESSOR_DRIVER_INTERNAL 参数

展开表

参数	说明
1	1 - Power Engine Plugin(PEP) 无法接受所需的通知
2	PEP 运行时通知类型
3	指向通知消息的指针
4	指向发出通知的处理器设备上下文 (FDO_DATA) 的指针

展开表

参数	说明
1	2 - Power Engine Plugin (PEP) 返回无效的处理器空闲状态
2	无效状态的类型 0x0：PEP 请求的处理器过多，无法协调空闲状态 参数 3 - 请求参与协调空闲转换的处理器数 参数 4 - 指向处理器设备上下文的指针 (FDO_DATA) 0x1：PEP 请求处理器处于无效空闲状态 参数 3 - 已请求空闲状态索引 参数 4 - 指向处理器设备上下文 (FDO_DATA) 的指针，对应于无效

参数	说明
	空闲状态 0x2：PEP 请求平台处于无效空闲状态 参数 3 - 已请求平台空闲状态索引 参数 4 - 指向处理器设备上下文 (FDO_DATA) 的指针，对应于无效空闲状态
3	请参阅参数 2
4	请参阅参数 2

原因

处理器驱动程序检测到一个不相和的条件，这促使它出现 bug 检查。 这种情况可能发生在处理器空闲和性能状态更改执行期间，这可能涉及内核、HAL 和 Power Engine Plugin (PEP) 等其他实体。 bug 中的信息检查将有助于确定违反了处理器驱动程序在处理其他实体时做出的哪些假设。 根本原因可能在于其他实体，转储文件可能会显示更多信息，以确定 bug 的原因检查。

Bug 检查 0x144：BUGCODE_USB3_DRIVER

项目 • 2024/02/23

BUGCODE_USB3_DRIVER bug 检查的值为 0x00000144。 这是用于所有 USB 3 bug 检查的代码。 参数 1 指定 USB 3 bug 检查的类型，其他参数的含义取决于参数 1。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

BUGCODE_USB3_DRIVER 参数

展开表

参数 1	参数 2	参数 3	参数 4	错误原因
0x1	可选。 指向用于重新发送 URB 的 IRP 的指针	指向 URB 的指针	指向客户端驱动程序的设备对象的指针	客户端驱动程序使用了以前发送到核心堆栈的 URB。
0x2	指向启动设备的物理设备对象 (PDO) 的指针	预留	预留	启动或分页设备重新枚举失败。
0x3	可选。 指向用于发送 URB 的 IRP 的指针	指向损坏的 URB 的指针	指向客户端驱动程序的设备对象的指针	客户端驱动程序将损坏的 URB 发送到核心堆栈。 之所以发生这种情况，是因为客户端驱动程序未使用 **USBD_*xxx*UrbAllocate** 分配 URB，或者客户端驱动程序为 URB 执行了缓冲区不足。
0x800	发送 Open Static Streams 请求的 IRQL	指向 Open Static Streams IRP 的指针	指向客户端驱动程序的设备	在 IRQL > PASSIVE LEVEL 发送了 Open Static Streams 请求。

参数 1	参数 2	参数 3	参数 4	错误原因
			对象的指针	
0x801	指向 Open Static Streams IRP 的指针	指向 Open Static Streams URB 的指针	指向客户端驱动程序的设备对象的指针	客户端驱动程序在查询流功能之前尝试打开静态流。 在客户端驱动程序成功查询流功能之前，客户端驱动程序无法打开静态流。 有关详细信息，请参阅“备注”。
0x802	客户端驱动程序尝试打开的静态流数	向客户端驱动程序授予的静态流数	指向客户端驱动程序的设备对象的指针	客户端驱动程序尝试打开无效数量的静态流。 流数不能为 0，并且不能大于在查询 USB 功能调用中返回到客户端驱动程序的值。
0x803	指向 Open Static Streams IRP 的指针	指向 Open Static Streams URB 的指针	指向客户端驱动程序的设备对象的指针	客户端驱动程序尝试为已打开静态流的终结点打开静态流。 在打开静态流之前，客户端驱动程序必须关闭以前打开的静态流。
0x804	泄漏的句柄上下文。 运行 **!usbanalyze -v** 以获取有关泄漏的句柄和 URB 的信息。 必须为客户端驱动程序启用驱动程序验证程序。	传递给 USBD_CreateHandle 的设备对象。	Reserved	客户端驱动程序忘记关闭之前使用 USBD_CreateHandle 创建的句柄，或忘记释放分配的 URB。
0x805	Close Static Streams URB 的 WDFREQUEST 句柄	指向 Close Static Streams URB 的指针	指向客户端驱动程序的设备对象的指针	客户端驱动程序发送了处于无效状态的 Close Static Streams URB（例如，处理 D0 Exit 后）。
0x806	指向 IRP 的指针	指向 URB 的指针	指向客户端驱动程序的设备对象的指针	客户端驱动程序在查询链接的 MDL 功能之前尝试发送链接的 MDL。 客户端驱动程序在成功查询链接的 MDL 功能之前，无法发送链接的 MDL。 有

参数 1	参数 2	参数 3	参数 4	错误原因
				关详细信息，请参阅“备注”。
0x807	指向链接的 MDL 的 指针	指向 URB 的指针	指向客户端驱动程序的设备对象的指针（如果可用）	客户端驱动程序将 URB 发送到核心堆栈，其传输缓冲区长度超过传入的 MDL 的字节计数（由 MmGetMdlByteCount 返回）。 有关详细信息，请参阅“备注”。
0x1001	XHCI_LIVEDUMP_CONTEXT	预留	预留	xHCI 控制器断言 HSE 位，指示主机系统错误。
0x1002	XHCI_LIVEDUMP_CONTEXT	预留	预留	xHCI 控制器断言 HCE 位，指示主机控制器错误。
0x1003	XHCI_LIVEDUMP_CONTEXT	预留	预留	xHCI 停止终结点命令返回未经处理的完成代码。
0x1004	XHCI_LIVEDUMP_CONTEXT	预留	预留	xHCI 终结点状态在发出 xHCI 终结点停止命令后收到上下文状态错误。
0x1005	XHCI_LIVEDUMP_CONTEXT	预留	预留	尝试清除控制终结点上的停止期间，设置取消排队指针失败。
0x1006	XHCI_LIVEDUMP_CONTEXT	预留	预留	在尝试清除控制终结点上的停止期间重置 EP 失败。
0x1007	XHCI_LIVEDUMP_CONTEXT	预留	预留	重置恢复期间 xHCI 控制器重置失败。
0x1008	XHCI_LIVEDUMP_CONTEXT	预留	预留	重置恢复期间 xHCI 控制器重启失败。
0x1009	XHCI_LIVEDUMP_CONTEXT	预留	预留	命令超时中止后，xHCI 控制器命令无法完成。
0x100A	XHCI_LIVEDUMP_CONTEXT	预留	预留	在尝试在终结点停止完成后设置取消排队

参数 1	参数 2	参数 3	参数 4	错误原因
				指针时，设置取消排队指针失败。
0x100B	XHCI_LIVEDUMP_CONTEXT	预留	预留	重置恢复期间 xHCI 控制器停止失败。
0x100C	XHCI_LIVEDUMP_CONTEXT	预留	预留	不支持 xHCI 控制器中的固件。 除非更新固件，否则不会在此控制器上加载 xHCI 驱动程序。
0x100D	XHCI_LIVEDUMP_CONTEXT	预留	预留	检测到控制器在物理上被删除。
0x100E	XHCI_LIVEDUMP_CONTEXT	预留	预留	驱动程序在启用了流的终结点上检测到错误。
0x100F	XHCI_LIVEDUMP_CONTEXT	预留	预留	xHCI 控制器中的固件已过时。 xHCI 驱动程序将继续使用此控制器，但可能会遇到一些问题。 建议使用固件更新。
0x1010	XHCI_LIVEDUMP_CONTEXT	预留	预留	传输事件 TRB 使用未经处理的完成代码完成。
0x1011	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器报告事件环已满。 当发生这种情况时，控制器也已知会删除事件。
0x1012	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器已按顺序完成命令。
0x1013	XHCI_LIVEDUMP_CONTEXT	预留	预留	命令中止完成后，控制器报告的命令环取消排队指针不正确。
0x1014	XHCI_LIVEDUMP_CONTEXT	预留	预留	启用插槽完成后，控制器会提供一个错误的插槽 ID。
0x1015	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器使用 BSR1 执行 SetAddress 命令失败。 这是意外行为。

参数 1	参数 2	参数 3	参数 4	错误原因
0x1016	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器在 usbdevice 重置期间未能启用插槽。 这是意外行为。
0x1017	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器执行终结点配置命令失败，我们在其中取消配置终结点。 这是意外行为。
0x1018	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器禁用插槽命令失败。 这是意外行为。
0x1019	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器执行 USB 设备重置命令失败。 这是意外行为。
0x101A	XHCI_LIVEDUMP_CONTEXT	预留	预留	终结点重置后，“设置取消排队指针”命令失败。
0x101B	XHCI_LIVEDUMP_CONTEXT	预留	预留	xHCI 重置终结点命令返回未经处理的完成代码。
0x101C	XHCI_LIVEDUMP_CONTEXT	预留	预留	xHCI 的 D0Entry 失败。
0x101D	XHCI_LIVEDUMP_CONTEXT	预留	预留	在请求取消期间使用“配置终结点”命令而不是“设置取消排队指针”时，暂时删除和添加流终结点（作为两个命令）失败。
0x101E	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器指示未在控制器上挂起的传输完成。 EventData == 1（取消引用传输事件 TRB 的指针会导致 bug 检查）
0x101F	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器指示未在控制器上挂起的传输完成。 EventData == 0（传输事件 TRB 中的逻辑地址不匹配）

参数 1	参数 2	参数 3	参数 4	错误原因
0x1020	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器指示未在控制器上挂起的传输完成。 EventData == 0（传输事件 TRB 中的逻辑地址不匹配）传输事件 TRB 可能是冗余的（指向最近完成的请求附近的某个位置）。
0x1021	XHCI_LIVEDUMP_CONTEXT	预留	预留	使用“配置终结点”命令重置未停止的终结点时，暂时删除和添加流终结点（作为两个命令）失败。
0x1022	XHCI_LIVEDUMP_CONTEXT	预留	预留	删除并添加同一终结点（作为一个命令）失败。
0x3000	USBHUB3_LIVEDUMP_CONTEXT	预留	预留	中心驱动程序成功重置了异常的中心。
0x3001	USBHUB3_LIVEDUMP_CONTEXT	预留	预留	中心驱动程序未能成功重置异常的中心。
0x3002	USBHUB3_LIVEDUMP_CONTEXT	预留	预留	中心驱动程序禁用了非函数 SuperSpeed 中心。
0x3003	USBHUB3_LIVEDUMP_CONTEXT	预留	预留	USB 设备枚举失败。

注解

若要查询 USB 功能，客户端驱动程序必须调用 WdfUsbTargetDeviceQueryUsbCapability 或 USBD_QueryUsbCapability

若要发送链接的 MDL，客户端驱动程序必须调用 USBD_QueryUsbCapability，并使用 URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER_USING_CHAINED_MDL 或 URB_FUNCTION_ISOCH_TRANSFER_USING_CHAINED_MDL。

另请参阅

通用串行总线 (USB)

内核实时转储代码引用

Bug 检查代码参考

bug 检查 0x145：SECURE_BOOT_VIOLATION

项目 • 2024/02/21

SECURE_BOOT_VIOLATION bug 检查的值为 0x00000145。 这表示由于策略无效或所需操作未完成，无法启动安全启动策略强制执行。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SECURE_BOOT_VIOLATION 参数

展开表

参数	说明
1	故障的状态代码。
2	安全启动策略的地址。
3	安全启动策略的大小。
4	Reserved

另请参阅

Bug 检查代码参考

bug 检查 0x147：ABNORMAL_RESET_DETECTED

项目 • 2024/02/23

ABNORMAL_RESET_DETECTED bug 检查的值为 0x00000147。 这表示 Windows 进行了异常重置。 未保存上下文或异常记录，并且未调用 bug 检查回调。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

ABNORMAL_RESET_DETECTED 参数

无

另请参阅

Bug 检查代码参考

错误检查 0x14B：SOC_SUBSYSTEM_FAILURE

项目 • 2024/10/16

SOC_SUBSYSTEM_FAILURE 错误检查的值为 0x0000014B。 这表示芯片上的系统 (SoC) 子系统遇到不可恢复的错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

错误检查 0x14B SOC_SUBSYSTEM_FAILURE 参数

展开表

参数	说明
1	SOC_SUBSYSTEM_FAILURE_DETAILS 结构的地址。
2	保留。
3	保留。
4	可选。 供应商提供的数据块的地址。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

dbgcmd

```
2: kd> !analyze -v
*******************************************************************************
***
*
*
*                        Bugcheck Analysis
*
*
*
```

```
*******************************************************************************
***

SOC_SUBSYSTEM_FAILURE (14b)
A SOC subsystem has experienced an unrecoverable critical fault.
Arguments:
Arg1: 9aa8d630, nt!SOC_SUBSYSTEM_FAILURE_DETAILS
Arg2: 00000000, Reserved
Arg3: 00000000, Reserved
Arg4: a126c000, (Optional) address to vendor supplied general purpose data
block.
```

使用提供的 nt!SOC_SUBSYSTEM_FAILURE_DETAILS 结构，使用 dt 命令和 Arg1 提供的地址转储故障数据。

dbgcmd

```
2: kd> dt nt!SOC_SUBSYSTEM_FAILURE_DETAILS 9aa8d630
   +0x000 SubsysType       : 1 ( SOC_SUBSYS_AUDIO_DSP )
   +0x008 FirmwareVersion  : 0
   +0x010 HardwareVersion  : 0
   +0x018 UnifiedFailureRegionSize : 0x24
   +0x01c UnifiedFailureRegion : [1]  "F"
```

与 SoC 供应商合作进一步分析数据，包括提供常规用途数据块的可选供应商。

你可能希望使用 **k、kb、kc、kd、kp、kP、kv（显示堆栈回溯）**命令检查堆栈跟踪。 可以指定处理器编号以检查所有处理器上的堆栈。

您还可以在导致此停止代码的代码中设置断点，并尝试单步执行故障代码。

有关详细信息，请参阅以下主题：

使用 Windows 调试器 (WinDbg) 进行故障转储分析

如果您不具备使用 Windows 调试器来解决此问题的能力，则可以使用一些基本的故障排除技术。

- 在事件查看器中检查系统日志以获取其他错误消息，这可能有助于识别导致此错误检查的设备或驱动程序。
- 如果在错误检查消息中标识了驱动程序，请禁用该驱动程序，或与制造商联系以获取驱动程序更新。
- 你可尝试运行系统制造商提供的硬件诊断。
- 有关其他常规故障排除信息，请参阅**分析错误检查蓝屏数据**。

要求

展开表

最低受支持的客户端	Windows 8
支持的最低服务器版本	Windows Server 2012

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

bug 检查 0x149：REFS_FILE_SYSTEM

项目 • 2024/02/21

REFS_FILE_SYSTEM Bug 检查的值为 0x00000149。 这表示发生了文件系统错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

REFS_FILE_SYSTEM 参数

展开表

参数	说明
1	__LINE__
2	ExceptionRecord
3	ContextRecord
4	ExceptionRecord->ExceptionAddress

解决方法

如果在堆栈上看到 RefsExceptionFilter，则第二个和第三个参数是异常记录和上下文记录。 对第 2 个参数执行 **.exr** 以查看异常信息，然后对第 3 个参数执行 **.cxr**，再执行 **kb**，以获取更丰富的堆栈跟踪。

另请参阅

Bug 检查代码参考

bug 检查 0x14A：KERNEL_WMI_INTERNAL

项目 • 2024/02/23

KERNEL_WMI_INTERNAL bug 检查的值为 0x0000014A。 这表示内部内核 WMI 子系统遇到致命错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

KERNEL_WMI_INTERNAL 参数

展开表

参数	说明
1	0：内核 WMI 条目引用计数从 0 递增。 参数 2：指向内核 WMI 条目的指针。 1：提前删除了内核 WMI 数据源。 参数 2：指向内核 WMI 数据源的指针。
2	请参阅参数 1
3	Reserved
4	Reserved

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

bug 检查 0x14C：FATAL_ABNORMAL_RESET_ERROR

项目 • 2024/02/21

FATAL_ABNORMAL_RESET_ERROR bug 检查的值为 0x0000014C。 这表示发生不可恢复的系统错误或系统已异常重置。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

FATAL_ABNORMAL_RESET_ERROR 参数

无

原因

系统遇到意外错误并重新启动。 可能导致此错误的问题包括：应用程序或辅助处理器中的硬件监视器计时器，指示系统挂起、用户因挂起启动的密钥序列、破坏或默认 bug 中的故障检查路径。 缓存可能不会刷新，生成的完整内存转储可能不包含当前线程上下文。

标记 {E1D08891-D5A3-45F9-B811-AD711DFB2607} 的辅助数据包含其他“Blackbox”数据。 使用 **.enumtag（枚举辅助回调数据）**查看数据。

bug 检查 0x14D：EXCEPTION_SCOPE_INVALID

项目 • 2024/02/23

EXCEPTION_SCOPE_INVALID bug 检查的值为 0x0000014D。 这表示检测到异常调度的内部不一致。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

EXCEPTION_SCOPE_INVALID 参数

展开表

参数	说明
1	保留
2	Reserved
3	Reserved
4	Reserved

bug 检查 0x14E：SOC_CRITICAL_DEVICE_REMOVED

项目 • 2024/02/23

SOC_CRITICAL_DEVICE_REMOVED bug 检查的值为 0x0000014E。 这表示关键 SOC 设备意外被删除或失败。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SOC_CRITICAL_DEVICE_REMOVED 参数

展开表

参数	说明
1	可用时，指示已删除的设备 ID（4 个字符打包的代码）
2	Reserved
3	Reserved
4	Reserved

bug 检查 0x14F：PDC_WATCHDOG_TIMEOUT

项目 • 2024/02/23

PDC_WATCHDOG_TIMEOUT bug 检查的值为 0x0000014F。 这表示系统组件无法在分配的时间段内做出响应，从而阻止系统退出已连接待机状态。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PDC_WATCHDOG_TIMEOUT 参数

展开表

参数	说明
1	已挂起组件的客户端 ID。
2	已挂起组件的客户端类型。 0x1：通知客户端无法响应。 参数 3：指向通知客户端 (PDC_NOTIFICATION_CLIENT) 的指针。 参数 4：指向 pdc!PDC_14F_TRIAGE 结构的指针。 0x2：复原客户端无法响应。 参数 3：指向复原客户端 (PDC_RESILIENCY_CLIENT) 的指针。 参数 4：指向 pdc!PDC_14F_TRIAGE 结构的指针。 0x3：激活器客户端保留引用时间过长。 参数 3：指向激活客户端 (pdc!_PDC_ACTIVATOR_CLIENT) 的指针。 参数 4：指向 pdc!PDC_14F_TRIAGE 结构的指针。 0x100：Win32k 没有及时完成 monitor-on 请求。 参数 3：此请求的最新 POWER_MONITOR_REQUEST_REASON 值。 参数 4：指示启动请求时所用的内部路径的值。
3	请参阅参数 2

参数	说明
4	请参阅参数 2

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

bug 检查 0x150：TCPIP_AOAC_NIC_ACTIVE_REFERENCE_LEAK

项目 • 2024/02/26

TCPIP_AOAC_NIC_ACTIVE_REFERENCE_LEAK bug 检查的值为 0x00000150。 这表示在完全清空发送队列时，应释放 NIC 活动引用。

> ⓘ **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

TCPIP_AOAC_NIC_ACTIVE_REFERENCE_LEAK 参数

无

另请参阅

Bug 检查代码参考

bug 检查 0x151: UNSUPPORTED_INSTRUCTION_MODE

项目 • 2024/02/23

UNSUPPORTED_INSTRUCTION_MODE bug 检查的值为 0x00000151。 这表示已尝试使用不受支持的处理器指令模式执行代码（例如，执行经典 Arm 指令而不是 ThumbV2 指令）。 这是不允许的。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

UNSUPPORTED_INSTRUCTION_MODE 参数

展开表

参数	说明
1	检测到问题时的程序计数器。
2	陷阱框架
3	Reserved
4	Reserved

另请参阅

Bug 检查代码参考

bug 检查 0x152：INVALID_PUSH_LOCK_FLAGS

项目 • 2024/02/23

INVALID_PUSH_LOCK_FLAGS bug 检查的值为 0x00000152。 这表示提供给其中一个推送锁定 API 的标志无效。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INVALID_PUSH_LOCK_FLAGS 参数

展开表

参数	说明
1	调用方提供的标志无效
2	推送锁定的地址
3	Reserved
4	Reserved

另请参阅

Bug 检查代码参考

bug 检查 0x153：KERNEL_LOCK_ENTRY_LEAKED_ON_THREAD_TERMINATION

项目 • 2024/02/23

KERNEL_LOCK_ENTRY_LEAKED_ON_THREAD_TERMINATION bug 检查的值为 0x00000153。 这表示线程在释放其所有 AutoBoost 锁定条目之前已终止。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

KERNEL_LOCK_ENTRY_LEAKED_ON_THREAD_TERMINATION 参数

展开表

参数	说明
1	线程的地址
2	未释放的条目的地址
3	指示条目状态的状态代码 0x1：锁定指针不为 NULL 0x2：设置了线程指针保留位 0x3：线程指针已损坏 0x4：条目保留剩余的 IO 或 CPU 提升
4	Reserved

原因

这通常是在线程从未释放它之前获取的锁定（例如，依靠另一个线程来释放锁定），或者线程未提供一组一致的标志来锁定包 API 时，这通常会导致此问题。

另请参阅

Bug 检查代码参考

bug 检查 0x154：UNEXPECTED_STORE_EXCEPTION

项目 • 2024/02/23

UNEXPECTED_STORE_EXCEPTION bug 检查的值为 0x00000154。 此 bug 检查指示内核内存存储组件捕获了意外的异常。

> **ⓘ 重要**
>
> 本主题是面向程序员的。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

UNEXPECTED_STORE_EXCEPTION 参数

展开表

参数	说明
1	指向存储上下文或数据管理器的指针
2	异常信息
3	Reserved
4	Reserved

解决方法

若要确定问题的原因，请使用调试器收集其他信息。 检查多个转储文件以查看此停止代码是否具有类似的特征，例如在出现停止代码时运行的相同代码。 有关详细信息，请参阅使用 Windows 调试器 (WinDbg) 进行崩溃转储分析、使用 !analyze 扩展和 !analyze。

获取有关源代码的信息后，在执行此代码之前，在相关代码中设置断点。 向前单步浏览代码，查看用于控制代码流的关键变量的值。 仔细检查代码的此区域，查找虚假假设或其他错误。

故障排除提示

如果无法使用导致问题的基础代码，这些故障排除提示可能会有所帮助。

- 在事件查看器中检查系统日志以获取其他错误消息，这有助于识别导致错误的设备或驱动程序。 若要打开事件查看器，请选择键盘快捷键 Win+R，输入 `eventvwr.msc`，然后按 Enter 键。 在系统日志中查找与蓝屏同时出现的严重错误。

- 选择“开始”，在搜索框中输入“**Windows 内存诊断**”，然后按 Enter 键。 选择是重启计算机并立即运行该工具，还是将该工具计划在下一次重启时运行。 Windows 内存诊断在计算机重启后自动运行，并自动执行标准内存测试。 若要运行扩展测试，请按 F1 键并使用向上键和向下键将测试组合设置为“扩展”。 按 F10 键应用所需的设置并恢复测试。

- 在设备管理器中查看是否有任何设备标有感叹号 (!)。 查看驱动程序属性中显示的事件日志，以查找任何故障驱动程序。 请尝试更新相关驱动程序。

- 使用系统文件检查器工具修复丢失或损坏的系统文件。 系统文件检查器是 Windows 中的一个实用工具，供用户扫描 Windows 系统文件中的损坏情况并恢复损坏的文件。 使用 `SFC /scannow` 命令运行系统文件检查器工具。 有关详细信息，请参阅使用系统文件检查器工具修复丢失或损坏的系统文件。

另请参阅

bug 检查代码参考

bug 检查 0x155：OS_DATA_TAMPERING

项目 • 2024/02/21

OS_DATA_TAMPERING bug 检查的值为 0x00000155。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

OS_DATA_TAMPERING 参数

无

另请参阅

Bug 检查代码参考

错误检查 0x157：KERNEL_THREAD_PRIORITY_FLOOR_VIOLATION

项目 • 2024/11/05

KERNEL_THREAD_PRIORITY_FLOOR_VIOLATION bug 检查的值为 0x00000157。 这表示在特定线程的优先级下尝试了非法操作。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

KERNEL_THREAD_PRIORITY_FLOOR_VIOLATION 参数

展开表

参数	说明
1	线程的地址
2	目标优先级值
3	指示违规性质的状态代码 0x1：目标优先级超流的优先级计数器 0x2：目标优先级下流的优先级计数器 0x3：目标优先级值非法
4	预留

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

bug 检查 0x158：ILLEGAL_IOMMU_PAGE_FAULT

项目 • 2024/02/23

ILLEGAL_IOMMU_PAGE_FAULT bug 检查的值为 0x00000158。 这表示 IOMMU 为无效 ASID 传递了页面故障数据包。 这是不安全的，因为 ASID 可能已被重复使用。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

ILLEGAL_IOMMU_PAGE_FAULT 参数

展开表

参数	说明
1	ASID 无效。
2	当前正在使用的 ASID 数。
3	使用此 ASID 的过程。
4	ASID 的引用计数。

bug 检查 0x159：HAL_ILLEGAL_IOMMU_PAGE_FAULT

项目 • 2024/02/21

HAL_ILLEGAL_IOMMU_PAGE_FAULT bug 检查的值为 0x00000159。 这表示 IOMMU 已针对正在释放的 ASID 提供页面故障。 驱动程序负责在此时间点之前完成任何飞行请求检查指示系统中的驱动程序未执行此操作。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

HAL_ILLEGAL_IOMMU_PAGE_FAULT 参数

展开表

参数	说明
1	IOMMU 供应商消除歧义
2	指向故障数据包的指针
3	供应商特定的故障数据包数据
4	供应商特定的故障数据包数据

展开表

参数	说明
1	IOMMU 供应商消除歧义 = 0x3xxx。
2	Status
3	PASID
4	DirectoryBase

bug 检查 0x15A：SDBUS_INTERNAL_ERROR

项目 • 2024/02/23

SDBUS_INTERNAL_ERROR bug 检查的值为 0x0000015A。 这表示 SD 附加设备上发生了不可恢复的硬件故障。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SDBUS_INTERNAL_ERROR 参数

展开表

参数	说明
1	指向导致故障的内部 SD 工作数据包的指针
2	指向控制器套接字信息的指针
3	指向发送到总线驱动程序的 SD 请求数据包的指针
4	Reserved

bug 检查 0x15B：WORKER_THREAD_RETURNED_WITH_SYSTEM_PAGE_PRIORITY_ACTIVE

项目 • 2024/02/23

WORKER_THREAD_RETURNED_WITH_SYSTEM_PAGE_PRIORITY_ACTIVE bug 检查的值为 0x0000015B。 这表示工作线程的系统页面优先级被调用的工作例程泄露。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅蓝屏错误疑难解答。

WORKER_THREAD_RETURNED_WITH_SYSTEM_PAGE_PRIORITY_ACTIVE 参数

展开表

参数	说明
1	工作例程的地址（在此地址上执行 ln 以查找有问题的驱动程序）
2	当前系统页面优先级值
3	WorkItem 参数
4	WorkItem 地址

bug 检查 0x160：WIN32K_ATOMIC_CHECK_FAILURE

项目 • 2024/02/23

WIN32K_ATOMIC_CHECK_FAILURE bug 检查的值为 0x00000160。 这表示 Win32k 函数违反了 ATOMICCHECK。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WIN32K_ATOMIC_CHECK_FAILURE 参数

展开表

参数	说明
1	堆栈上当前位于 ATOMIC 操作内的函数计数
2	Reserved
3	Reserved
4	Reserved

bug 检查 0x162：KERNEL_AUTO_BOOST_INVALID_LOCK_RELEASE

项目 • 2024/02/23

KERNEL_AUTO_BOOST_INVALID_LOCK_RELEASE bug 检查的值为 0x00000162。 这表示 AutoBoost 跟踪的锁定是由一个不拥有锁定的线程释放的。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

KERNEL_AUTO_BOOST_INVALID_LOCK_RELEASE 参数

展开表

参数	说明
1	线程的地址
2	锁定地址
3	线程的会话 ID
4	Reserved

原因

当某些线程代表另一个线程（启用 AutoBoost 跟踪时不合法）或某些线程尝试释放它不再拥有的锁定时，通常会造成这种情况。

错误检查 0x163：WORKER_THREAD_TEST_CONDITION

项目 • 2024/10/16

WORKER_THREAD_TEST_CONDITION bug 检查的值为 0x00000163。 这表示内核工作线程的测试引发了故障。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WORKER_THREAD_TEST_CONDITION 参数

展开表

参数	说明
1	活动测试标志
2	与触发故障的测试对应的标志
3	预留
4	预留

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

bug 检查 0x164：WIN32K_CRITICAL_FAILURE

项目 • 2024/02/23

WIN32K_CRITICAL_FAILURE bug 检查的值为 0x00000164。 这表示 Win32k 遇到严重故障。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WIN32K_CRITICAL_FAILURE 参数

展开表

参数	说明
1	1 - 故障的类型。 0x1：REGION_VALIDATION_FAILURE - 区域超出表面边界。 2 - 指向 DC 的指针 3 - 指向 SURFACE 的指针 4 - 指向 REGION 的指针 0x2：OPERATOR_NEW_USED - 运算符“new”用于分配内存。 2 - 保留 3 - 保留 4 - 保留 0x3：CRITICAL_APISET_EXTENSIONS_MISSING - 关键扩展 APISET API 缺失。 2 - wchar_t* 到缺失函数的名称 3 - 保留 4 - 保留

参数	说明
	0x4：GDI_SPRITE_SURFACE_INVALID_DELETE - 正在删除 GDI 子画面的形状，而不删除子画面。 2 - SURFACE 的句柄 3 - 对 SURFACE 的引用计数 4 - SURFACE 所有者的 PID 0x5： POINTER_DEVICE_EXCLUSIVE_OPEN_FAILED - 无法打开指针设备。 2 - 设备的 UNICODE_STRING 3 - 保留 4 - 保留 0x8：PUBLIC_DC_INVALID_PRIVATE_MEMBER - 公共 DC 具有指向特定进程拥有的对象的指针。 2 - 指向 DC 的指针 3 - 拥有对象的进程 ID 4 - 保留 0xA：TTFD_INVOKE_ILLEGAL_ID - TTFD 中使用的函数表索引无效。 2 - 保留 3 - 保留 4 - 保留 0xB：OTFD_INVOKE_ILLEGAL_ID - ATMFD 中使用的函数表索引无效。 2 - 保留 3 - 保留 4 - 保留 0xC：GFPE_INVOKE_ILLEGAL_ID - PALETTE 中使用的函数表索引无效。 2 - 指向 PALETTE 的指针 3 - 索引无效 4 - 最大有效索引 + 1 0x10：USER_SAS_REGISTRATION_FAILED - SAS 密钥注册失败。

参数	说明
	2 - vkey 3- 修饰符 4 - 标志
2	请参阅参数 1
3	请参阅参数 1
4	请参阅参数 1

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

bug 检查 0x16C：INVALID_RUNDOWN_PROTECTION_FLAGS

项目 • 2024/02/23

INVALID_RUNDOWN_PROTECTION_FLAGS bug 检查的值为 0x0000016C。 这表示提供给其中一个断开保护 API 的标志无效。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INVALID_RUNDOWN_PROTECTION_FLAGS 参数

展开表

参数	说明
1	调用方提供的标志无效
2	断开引用的地址
3	Reserved
4	Reserved

bug 检查 0x16D：INVALID_SLOT_ALLOCATOR_FLAGS

项目 • 2024/02/23

INVALID_SLOT_ALLOCATOR_FLAGS bug 检查的值为 0x0000016D。 这表示提供给其中一个插槽分配器 API 的标志无效。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INVALID_SLOT_ALLOCATOR_FLAGS 参数

展开表

参数	说明
1	调用方提供的标志无效
2	Reserved
3	Reserved
4	Reserved

bug 检查 0x16E: ERESOURCE_INVALID_RELEASE

项目 • 2024/02/23

ERESOURCE_INVALID_RELEASE bug 检查的值为 0x0000016E。 这表示提供给 ExReleaseResourceForThreadLite 的目标线程指针无效。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

ERESOURCE_INVALID_RELEASE 参数

展开表

参数	说明
1	正在释放的资源
2	当前线程
3	传入的目标线程不正确
4	Reserved

原因

如果 API 客户端跳过了对 ExSetOwnerPointerEx 的调用（如果预期使用跨线程发布），或者调用方意外传入了 ExGetCurrentResourceThread 提供的值，则会出现此 bug 检查。

Bug 检查 0x170：CLUSTER_CSV_CLUSSVC_DISCONNECT_WATCHDOG

项目 • 2024/02/23

CLUSTER_CSV_CLUSSVC_DISCONNECT_WATCHDOG bug 检查的值为 0x00000170。 这表示群集断开连接未向前推进。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

CLUSTER_CSV_CLUSSVC_DISCONNECT_WATCHDOG 参数

展开表

参数	说明
1	正在处理群集断开连接的线程的 ID。
2	超时（以毫秒为单位）。
3	Reserved
4	Reserved

原因

群集断开连接未向前推进。

解决方法

另请参阅

使用实时转储排查挂起问题（博客）

Bug 检查代码参考

Bug 检查 0x171: CRYPTO_LIBRARY_INTERNAL_ERROR

项目 • 2024/02/23

CRYPTO_LIBRARY_INTERNAL_ERROR bug 检查的值为 0x00000171。 它指示加密库中出现内部错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

CRYPTO_LIBRARY_INTERNAL_ERROR 参数

展开表

参数	说明
1	故障的 ID。
2	保留。
3	保留。
4	保留。

原因

此 bug 检查指示加密库遇到本不应发生的异常，并且该库没有向调用方发出错误信号的安全方法。 这可能是主动攻击的症状。

另请参阅

Bug 检查代码参考

加密 API：下一代

bug 检查 0x173：COREMSGCALL_INTERNAL_ERROR

项目 • 2024/02/23

PFN_SHARE_COUNT bug 检查的值为 0x00000173。 这指示 CoreMessageCall 检测到不可恢复的错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

COREMSGCALL_INTERNAL_ERROR 参数

展开表

参数	说明
1	故障的类型。
2	保留。
3	保留。
4	保留。

原因

CoreMessageCall 检测到不可恢复的错误。

另请参阅

Bug 检查代码参考

bug 检查 0x174：COREMSG_INTERNAL_ERROR

项目 • 2024/02/23

COREMSG_INTERNAL_ERROR bug 检查的值为 0x00000174。 这指示 CoreMessaging 检测到不可恢复的错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

COREMSG_INTERNAL_ERROR 参数

展开表

参数	说明
1	故障的类型。
2	保留。
3	保留。
4	保留。

原因

CoreMessaging 检测到不可恢复的错误。

另请参阅

Bug 检查代码参考

bug 检查 0x178：ELAM_DRIVER_DETECTED_FATAL_ERROR

项目 • 2024/02/23

ELAM_DRIVER_DETECTED_FATAL_ERROR bug 检查的值为 0x00000178。 这表示 ELAM 驱动程序检测到致命错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

ELAM_DRIVER_DETECTED_FATAL_ERROR 参数

展开表

参数	说明
1	故障的类型。 0x0：无法吊销 TPM 证明 2 - 指向正在检查的驱动程序的 BDCB_IMAGE_INFORMATION 结构的指针 3 - TBS_RESULT 故障代码 0x10000：ELAM 供应商定义的故障 2 -（可选）ELAM 供应商提供的值 3 -（可选）ELAM 供应商提供的值
2	请参阅参数 1
3	请参阅参数 1
4	（可选）ELAM 供应商提供的常规用途数据块

bug 检查 0x17B：PROFILER_CONFIGURATION_ILLEGAL

项目 • 2024/02/23

PROFILER_CONFIGURATION_ILLEGAL bug 检查的值为 0x0000017B。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PROFILER_CONFIGURATION_ILLEGAL 参数

无

bug 检查 0x17E：MICROCODE_REVISION_MISMATCH

项目 • 2024/02/23

MICROCODE_REVISION_MISMATCH bug 检查的值为 0x0000017E。 这表示多处理器配置中的一个或多个处理器加载的微代码不一致。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

MICROCODE_REVISION_MISMATCH 参数

展开表

参数	说明
1	不匹配的处理器的处理器 CPUID 签名值。
2	处理器的预期微代码修订。
3	处理器的实际报告的微代码修订。
4	不匹配处理器的处理器索引。

原因

多处理器配置中的一个或多个处理器加载的微代码不一致。 此 bug 检查指示出现故障的系统固件错误地将微代码更新应用于主机配置中的一部分处理器。 系统固件必须以统一的方式对所有处理器应用微代码更新。

另请参阅

Bug 检查代码参考

bug 检查 0x189：BAD_OBJECT_HEADER

项目 • 2024/02/23

BAD_OBJECT_HEADER bug 检查的值为 0x00000189。 这表示 OBJECT_HEADER 已损坏。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

BAD_OBJECT_HEADER 参数

展开表

参数	说明
1	指向错误的 OBJECT_HEADER 的指针
2	基于 OBJECT_HEADER 中的 TypeIndex 指向生成的 OBJECT_TYPE 的指针
3	损坏的类型。 0x0：类型索引已损坏 0x1：对象安全描述符无效
4	Reserved

bug 检查 0x18B：SECURE_KERNEL_ERROR

项目 • 2024/02/26

SECURE_KERNEL_ERROR bug 检查的值为 0x0000018B。 这表示安全内核遇到致命错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SECURE_KERNEL_ERROR 参数

展开表

参数	说明
1	保留
2	Reserved
3	Reserved
4	Reserved

bug 检查 0x18C：HYPERGUARD_VIOLATION

项目 • 2024/02/23

HYPERGUARD_VIOLATION bug 检查的值为 0x0000018C。 这表示内核检测到关键内核代码或数据已损坏。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

> **ⓘ 备注**
>
> 此 bug 代码保留仅供 Hyperguard 使用。
> 它不是供其他组件在数据损坏方案中使用的常规用途 bug 代码。
> 相反，为组件定义唯一的 bug 代码。
> 不要在组件中使用此 bug 代码。

HYPERGUARD_VIOLATION 参数

展开表

参数	说明
1	损坏区域的类型 - 下面列出的值。
2	故障类型相关信息。
3	保留。
4	保留。

损坏区域的类型

1001：通用数据区域

1002：页面哈希不匹配

1004：处理器 IDT

1005：处理器 GDT

1007：调试例程修改

1008：动态代码区域

1009：通用可共享数据区域

100a：虚拟机监控程序覆盖区域

100b：处理器模式配置错误

100c：扩展处理器控制寄存器

100d：安全内存区域

100e：已加载模块

100f：处理器状态区域

1010：内核 CFG 位图

1011：虚拟地址 0 页

1012：交替反转函数表

1013：按需页面验证失败

1016：安全映像区域

1017：内核虚拟地址保护不一致

1101：内部上下文损坏

1102：IDTR 修改

1103：GDTR 修改

原因

当内核检测到关键内核代码或数据已损坏时，将生成此 bug检查。 损坏通常有三个原因：

1. 驱动程序无意或故意修改了关键内核代码或数据。
2. 开发人员尝试使用启动系统时未附加的内核调试器设置正常的内核断点。 仅当在启动时附加调试器时，才能设置正常的断点“bp”。 可以随时设置硬件断点“ba”。

3. 发生硬件损坏，例如 RAM 保存内核代码或数据失败。

另请参阅

Bug 检查代码参考

bug 检查 0x18D：SECURE_FAULT_UNHANDLED

项目 • 2024/06/04

SECURE_FAULT_UNHANDLED bug 检查的值为 0x0000018D。

此 bug 检查无法处理安全内核引发的安全故障。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SECURE_FAULT_UNHANDLED 参数

展开表

参数	说明
1	安全错误代码位掩码 - 以下值。
2	安全故障 VA（仅适用于某些安全故障类型）。
3	异常记录。
4	上下文记录。

安全错误代码位掩码

```text
      0x1 : KSECURE_FAULT_SLAT_NX
          A no-execute fault occurred due to SLAT page protections.
      0x2 : KSECURE_FALT_SLAT_READ
          A read fault occurred due to SLAT page protections.
      0x4 : KSECURE_FAULT_SLAT_WRITE
          A write fault occurred due to SLAT page protections.
      0x8 : KSECURE_FAULT_DOUBLE_FAULT
          A secure fault occurred before the prior secure fault had been
dismissed by the kernel.
```

原因

无法处理安全内核引发的安全故障。

另请参阅

错误检查代码参考

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

bug 检查 0x18E：KERNEL_PARTITION_REFERENCE_VIOLATION

项目 • 2024/02/21

KERNEL_PARTITION_REFERENCE_VIOLATION bug 检查的值为 0x0000018E。

此错误指示未正确取消引用分区。 当内核模式驱动程序无法正确取消引用分区对象时，通常会发生这种情况。 当内核中发生严重数据损坏时，也可能会发生这种情况。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

KERNEL_PARTITION_REFERENCE_VIOLATION 参数

蓝屏上显示以下参数。

参数 1 指示故障的类型。 其他参数的含义取决于参数 1 的值。

展开表

参数 1	参数 2	参数 3	参数 4
0x0：正在删除具有非零硬引用计数的分区。	指向分区的指针。	未完成的硬引用数。	Reserved
0x1：正在删除系统分区	指向分区的指针。	预留	预留
0x2：正在删除具有未完成的前工作队列项的分区。	指向分区的指针。	指向具有未完成项的前工作队列的指针。	Reserved

bug 检查 0x191: PF_DETECTED_CORRUPTION

项目 • 2024/02/26

PF_DETECTED_CORRUPTION bug 检查的值为 0x00000191。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PF_DETECTED_CORRUPTION 参数

展开表

参数	说明
1	保留
2	Reserved
3	Reserved
4	Reserved

bug 检查 0x192：KERNEL_AUTO_BOOST_LOCK_ACQUISITION_WITH_RAISED_IRQL

项目 • 2024/02/23

KERNEL_AUTO_BOOST_LOCK_ACQUISITION_WITH_RAISED_IRQL bug 检查的值为 0x00000192。 这表示在 DISPATCH_LEVEL 或更高级别执行时已获取 AutoBoost 跟踪的锁定。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

KERNEL_AUTO_BOOST_LOCK_ACQUISITION_WITH_RAISED_IRQL 参数

展开表

参数	说明
1	线程的地址
2	锁定地址
3	已获取锁定的 IRQL
4	Reserved

原因

调用方无法在 APC_LEVEL 以上锁定时阻止，因为锁定可能由中断的线程独占持有，这会导致死锁。

bug 检查 0x196：LOADER_ROLLBACK_DETECTED

项目 • 2024/02/23

LOADER_ROLLBACK_DETECTED bug 检查的值为 0x00000196。 这表示 OS 加载程序的版本与操作系统不匹配。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

LOADER_ROLLBACK_DETECTED 参数

展开表

参数	说明
1	加载程序安全版本
2	OS 安全版本
3	Reserved
4	Reserved

bug 检查 0x197：WIN32K_SECURITY_FAILURE

项目 • 2024/02/21

WIN32K_SECURITY_FAILURE bug 检查的值为 0x00000197。 这表示在 win32k 中检测到安全故障。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WIN32K_SECURITY_FAILURE 参数

展开表

参数	说明
1	失败类型 0x1：对象句柄条目未重新指向对象。 2 - 指向对象类型的指针 3 - 指向对象句柄条目的指针 4 - 预期对象
2	请参阅参数 1
3	请参阅参数 1
4	请参阅参数 1

bug 检查 0x199：KERNEL_STORAGE_SLOT_IN_USE

项目 • 2024/02/23

KERNEL_STORAGE_SLOT_IN_USE bug 检查的值为 0x00000199。 这表示无法释放存储插槽，因为有一个对象在使用存储插槽。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

KERNEL_STORAGE_SLOT_IN_USE 参数

展开表

参数	说明
1	存储阵列的地址
2	Reserved
3	Reserved
4	Reserved

bug 检查 0x19A：WORKER_THREAD_RETURNED_WHILE_ATTACHED_TO_SILO

项目 • 2024/02/23

WORKER_THREAD_RETURNED_WHILE_ATTACHED_TO_SILO bug 检查的值为 0x0000019A。 这表示工作线程已附加到孤岛，且在返回之前未分离。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WORKER_THREAD_RETURNED_WHILE_ATTACHED_TO_SILO 参数

展开表

参数	说明
1	工作例程的地址
2	Workitem 参数
3	Workitem 地址
4	Reserved

原因

若要调查，请在参数 1 上使用“输入（列出最近符号）”命令来帮助识别行为不正确的驱动程序。

bug 检查 0x19B：TTM_FATAL_ERROR

项目 • 2024/02/23

TTM_FATAL_ERROR bug 检查的值为 0x0000019B。 这表示终端拓扑管理器遇到致命错误。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

TTM_FATAL_ERROR 参数

展开表

参数	说明
1	失败类型 0x1：无法生成终端对象。 2 - 故障的 NT 状态代码 3 - 保留 4 - 保留
2	请参阅参数 1
3	请参阅参数 1
4	请参阅参数 1

错误检查 0x19C：WIN32K_POWER_WATCHDOG_TIMEOUT

项目 • 2024/02/22

The WIN32K_POWER_WATCHDOG_TIMEOUT 错误检查的值为 0x0000019C。 这表明 Win32k 没有及时打开监视器。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WIN32K_POWER_WATCHDOG_TIMEOUT 参数

展开表

参数	说明
1	失败类型 (win32kbase!POWER_WATCHDOG_TYPE) `0x10 : The power request queue is not making progress 2 - Pointer to the thread processing power requests, if any 3 - Pointer to the win32k user lock 4 - Pointer to the power request (win32kbase!PPOWERREQUEST) being processed, if any 0x20 : Calling PO to set power state 2 - Pointer to the power request worker thread 3 - Reserved 4 - Reserved 0x30 : Calling GDI to power on 2 - Pointer to the power request worker thread 3 - Reserved 4 - Reserved 0x40 : Calling DWM to render 2 - Pointer to the power request worker thread 3 - Reserved 4 - Reserved 0x50 : Calling monitor driver to power on 2 - Pointer to the power request worker thread 3 - Reserved 4 - Reserved`
2	请参阅参数 1
3	请参阅参数 1
4	请参阅参数 1

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x1A2：WIN32K_CALLOUT_WATCHDOG_BUGCHECK

项目 • 2024/02/22

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WIN32K_CALLOUT_WATCHDOG_BUGCHECK 实时转储的值为 0x000001A2。 这表明 Win32k 的标注没有及时返回。

WIN32K_CALLOUT_WATCHDOG_BUGCHECK 参数

蓝屏上显示以下参数。

展开表

参数	说明
1	从 Win32k 标注返回线程阻止提示。
2	保留。
3	保留。
4	保留。

另请参阅

Bug 检查代码参考

错误检查 0x1AA：EXCEPTION_ON_INVALID_STACK

项目 • 2024/02/22

EXCEPTION_ON_INVALID_STACK 错误检查的值为 0x000001AA。 此 BugCheck 表明异常调度越过了无效的内核堆栈。 这可能表明内核堆栈指针在异常调度或展开期间已损坏（例如，由于帧指针的堆栈损坏），或者驱动程序正在一个不合法的内核堆栈上执行。

> **ⓘ 重要**
>
> 本主题是面向程序员的。 如果您是在使用计算机时收到此错误代码的客户，请参阅**蓝屏错误疑难解答**。

EXCEPTION_ON_INVALID_STACK 参数

展开表

参数	说明
1	指向当前堆栈的指针。
2	堆栈限制类型，如 NormalStackLimits (3)。 由于堆栈无效，这代表了内核对计算机状态下应激活的内核堆栈类型的最佳估计。 堆栈限制类型： • 0x0 - 错误检查堆栈（如果堆栈限制是在错误检查期间计算的，则可以使用任何堆栈） • 0x1 - 处理器 DPC 堆栈 • 0x2 - KeExpandKernelStackAndCallout(Ex) 堆栈 • 0x3 - 正常内核线程栈 • 0x4 - 线程上下文交换期间的内核线程栈（哪个线程处于活动状态尚不明确） • 0x5 - win32k 内核/用户标注堆栈 • 0x6 - 处理器 ISR 堆栈 • 0x7 - 内核调试器堆栈（内核调试器处理 KD I/O 时的任何堆栈） • 0x8 - 处理器 NMI 处理堆栈 • 0x9 - 处理器计算机检查处理堆栈 • 0xA - 处理器异常堆栈（用于调度某些引发的 IRQL 异常）
3	指向上下文记录的指针，表示在遇到无效堆栈时正在展开（或为异常调度）的上下文。

参数	说明
4	提供一条表示正在调度的活动异常的异常记录。

原因

尝试访问无效堆栈。 由于内核堆栈的大小有限，因此开发人员需要谨慎跟踪其限制，例如在使用它来复制显存块时。 有关 Windows 内核堆栈的信息，请参阅使用内核堆栈。

解决方法

使用完整的内核转储或附加的调试器，以下命令可能有助于收集信息并跟踪错误访问内存的代码。

首先使用 !analyze 命令来收集信息，特别是错误检查参数。 如果有，还可检查故障源行和模块名称。

dbgcmd

```
Arguments:
Arg1: 00000018940ffbe8
Arg2: 0000000000000003
Arg3: ffffe301c8db2900
Arg4: ffffdc0e9ee665d8
```

使用分析输出中提供的 .trap 命令链接，将上下文设置为陷阱帧。

dbgcmd

```
2: kd> .trap 0xffffdc0e9ee66680
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=003f8b813f20b6e0 rbx=0000000000000000 rcx=ee7defdd9c530000
rdx=ffffcb81660ea078 rsi=0000000000000000 rdi=0000000000000000
rip=fffff8002b7f8933 rsp=ffffdc0e9ee66810 rbp=ffffcb81511c3010
 r8=0000000000000001  r9=0000000000004014 r10=ffffdc0e9ee66910
r11=0000000000000000 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl zr na po nc
dxgmms2!RemoveHeadList+0xd [inlined in
dxgmms2!VidSchiSignalRegisteredSyncObjects+0x3f]:
fffff800`2b7f8933 48395808        cmp     qword ptr [rax+8],rbx
ds:003f8b81`3f20b6e8=????????????????
```

使用 !vm 命令检查内存使用情况，例如查看正在使用的内核堆栈内存量。

dbgcmd

```
0: kd> !vm

Physical Memory:          1541186 (    6164744 Kb)
Available Pages:           470550 (    1882200 Kb)
ResAvail Pages:           1279680 (    5118720 Kb)

...

Kernel Stacks:              13686 (      54744 Kb)
```

在本例中，所使用的内存量小于可用内存量。

使用 !thread 命令来收集有关正在运行的内容的信息。 在本例中，视频计划程序工作线程似乎正在运行。

dbgcmd

```
2: kd> !thread
THREAD ffffcb816348b040  Cid 0c58.4a1c  Teb: 0000000000000000 Win32Thread:
0000000000000000 RUNNING on processor 2
Not impersonating
DeviceMap                 ffff840f38c04170
Owning Process            ffffcb81648980c0       Image:
YourPhone.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      34501403       Ticks: 0
Context Switch Count      43             IdealProcessor: 3
UserTime                  00:00:00.000
KernelTime                00:00:00.015
Win32 Start Address 0x00007fff34656d00
Stack Init ffffdc0e9ee675b0 Current ffffdc0e9ee66610
Base ffffdc0e9ee68000 Limit ffffdc0e9ee61000 Call 0000000000000000
Priority 8 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
...
```

然后使用 kb (Display Stack Backtrace) 和 f 选项来显示堆栈和内存使用情况，以查看是否存在大内存用户。

dbgcmd

```
2: kd> kf
...
02        40 ffffdc0e`9ee66850 fffff800`2b7f8801
dxgmms2!VidSchiMarkDeviceAsError+0x4d
...
```

如果代码的某一部分看起来可疑，请使用 u, ub, uu (Unassemble) 命令来检查相关的汇编语言代码。

dbgcmd

```
2: kd> u ffffdc0e`9ee66850 l10
ffffdc0e`9ee66850 1030            adc     byte ptr [rax],dh
ffffdc0e`9ee66852 1c51            sbb     al,51h
ffffdc0e`9ee66854 81cbffffc068    or      ebx,68C0FFFFh
ffffdc0e`9ee6685a e69e            out     9Eh,al
ffffdc0e`9ee6685c 0e              ???
ffffdc0e`9ee6685d dcff            fdiv    st(7),st
ffffdc0e`9ee6685f ff00            inc     dword ptr [rax]
ffffdc0e`9ee66861 0000            add     byte ptr [rax],al
ffffdc0e`9ee66863 0000            add     byte ptr [rax],al
ffffdc0e`9ee66865 0000            add     byte ptr [rax],al
ffffdc0e`9ee66867 000e            add     byte ptr [rsi],cl
ffffdc0e`9ee66869 0000            add     byte ptr [rax],al
ffffdc0e`9ee6686b 0000            add     byte ptr [rax],al
ffffdc0e`9ee6686d 0000            add     byte ptr [rax],al
ffffdc0e`9ee6686f 0010            add     byte ptr [rax],dl
ffffdc0e`9ee66871 301c51          xor     byte ptr [rcx+rdx*2],bl
```

使用 .cxr (Display Context Record) 命令，利用 !analyze 提供的参数 3 值来显示上下文记录。

dbgcmd

```
2: kd> .cxr ffffe301c8db2900
rax=003f8b813f20b6e0 rbx=ffffcb813f607650 rcx=ee7defdd9c530000
rdx=ffffcb81660ea078 rsi=0000000000000000 rdi=ffffcb81511c30a8
rip=fffff8002b7f8933 rsp=ffffdc0e9ee66810 rbp=ffffcb81511c3010
 r8=0000000000000001  r9=0000000000004014 r10=ffffdc0e9ee66910
r11=0000000000000000 r12=ffffdc0e9ee66910 r13=ffffcb814019c000
r14=0000000000000000 r15=ffffdc0e9ee66910
iopl=0         nv up ei pl zr na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b
efl=00050246
dxgmms2!RemoveHeadList+0xd [inlined in
dxgmms2!VidSchiSignalRegisteredSyncObjects+0x3f]:
fffff800`2b7f8933 48395808        cmp     qword ptr [rax+8],rbx
ds:002b:003f8b81`3f20b6e8=????????????????
```

使用 .exr (Display Exception Record 命令，利用 !analyze 提供的参数 4 值来显示例外记录。

dbgcmd

```
2: kd> .exr ffffdc0e9ee665d8
ExceptionAddress: fffff8002b7f8933
(dxgmms2!RemoveHeadList+0x000000000000000d)
   ExceptionCode: c0000005 (Access violation)
  ExceptionFlags: 00000000
NumberParameters: 2
   Parameter[0]: 0000000000000000
   Parameter[1]: ffffffffffffffff
Attempt to read from address ffffffffffffffff
```

异常记录显示试图从 `ffffffffffffffff` 的地址读取数据，这将是一个需要进一步调查的方面。

另请参阅

错误检查代码参考

错误检查 0x1AB：UNWIND_ON_INVALID_STACK

项目 • 2024/02/26

UNWIND_ON_INVALID_STACK 错误检查的值为 0x000001AB。 它表明在试图访问有效内核堆栈范围之外的内存。 具体而言，此 BugCheck 表明堆栈展开越过了无效的内核堆栈。 这可能表明内核堆栈指针在异常调度或展开期间已损坏（例如，由于帧指针的堆栈损坏），或者驱动程序正在一个不合法的内核堆栈上执行。

发生无效访问时，异常记录不可用。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅蓝屏错误疑难解答。

UNWIND_ON_INVALID_STACK 参数

展开表

参数	说明
1	指向当前堆栈的指针。
2	堆栈限制类型，如 NormalStackLimits (3)。 由于堆栈无效，这代表了内核对计算机状态下应激活的内核堆栈类型的最佳估计。 堆栈限制类型： • 0x0 - 错误检查堆栈（如果堆栈限制是在错误检查期间计算的，则可以使用任何堆栈） • 0x1 - 处理器 DPC 堆栈 • 0x2 - KeExpandKernelStackAndCallout(Ex) 堆栈 • 0x3 - 正常内核线程栈 • 0x4 - 线程上下文交换期间的内核线程栈（哪个线程处于活动状态尚不明确） • 0x5 - win32k 内核/用户标注堆栈 • 0x6 - 处理器 ISR 堆栈 • 0x7 - 内核调试器堆栈（内核调试器处理 KD I/O 时的任何堆栈） • 0x8 - 处理器 NMI 处理堆栈 • 0x9 - 处理器计算机检查处理堆栈 • 0xA - 处理器异常堆栈（用于调度某些引发的 IRQL 异常）

参数	说明
3	指向上下文记录的指针，表示在遇到无效堆栈时正在展开（或为异常调度）的上下文。
4	ExceptionRecord - 为 UNWIND_ON_INVALID_STACK 保留并始终为 0。

原因

尝试访问无效堆栈。 由于内核堆栈的大小有限，因此开发人员需要谨慎跟踪其限制，例如在使用它来复制显存块时。 有关 Windows 内核堆栈的信息，请参阅使用内核堆栈。

解决方法

使用完整的内核转储或附加的调试器，以下命令可能有助于收集信息并跟踪错误访问内存的代码。

首先使用 !analyze 命令来收集信息，特别是错误检查参数。 如果有，还可检查故障源行和模块名称。

dbgcmd

```
Arguments:
Arg1: 89344350fffff607
Arg2: 0000000000000003
Arg3: fffff607893436c4
Arg4: fffff60789343ea8
```

使用分析输出中提供的 .trap 命令链接，将上下文设置为陷阱帧。

dbgcmd

```
TRAP_FRAME:  fffff60789343f50 -- (.trap 0xfffff60789343f50)
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=fffff607893441e8 rbx=0000000000000000 rcx=0000000010000004
rdx=0000000000000002 rsi=0000000000000000 rdi=0000000000000000
rip=fffff8026dc296cf rsp=fffff607893440e8 rbp=fffff60789344350
 r8=fffff8028e7a08b2  r9=0000000000000008 r10=fffff8029e9c3980
r11=fffff607893440f8 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
```

使用 !thread 命令来收集有关正在运行的内容的信息。 在本例中，视频计划程序工作线程似乎正在运行。

dbgcmd

```
2: kd> !thread
THREAD ffff8f8e9af25080  Cid 0004.0378  Teb: 0000000000000000 Win32Thread:
0000000000000000 RUNNING on processor 2
Not impersonating
DeviceMap                 ffffd601dbe63e30
Owning Process            ffff8f8e99ab4040       Image:         System
Attached Process          N/A            Image:         N/A
Wait Start TickCount      14361          Ticks: 0
Context Switch Count      64607          IdealProcessor: 1
UserTime                  00:00:00.000
KernelTime                00:00:06.046
Win32 Start Address dxgmms2!VidSchiWorkerThread (0xfffff8027a70d100)
Stack Init fffff60789344c70 Current fffff607893445c0
Base fffff60789345000 Limit fffff6078933f000 Call 0000000000000000
Priority 16 BasePriority 16 PriorityDecrement 0 IoPriority 2 PagePriority 5
...
```

然后使用 kb (Display Stack Backtrace) 和 f 选项来显示堆栈和内存使用情况，以查看是否存在大内存用户。

dbgcmd

```
2: kd> kf
...
02       198 fffff607`89344460 fffff802`8e6b41d5     amdkmdag+0x2308b2
03       120 fffff607`89344580 fffff802`8e59eb35     amdkmdag+0x1441d5
04        30 fffff607`893445b0 fffff802`8e62b5e8     amdkmdag+0x2eb35
05        c0 fffff607`89344670 fffff802`8e623f6c     amdkmdag+0xbb5e8
...
```

如果代码的某一部分看起来可疑，请使用 u, ub, uu (Unassemble) 命令来检查相关的汇编语言代码。

dbgcmd

```
2: kd> u fffff607`893442c8 l10
fffff607`893442c8 d04234          rol     byte ptr [rdx+34h],1
fffff607`893442cb 8907            mov     dword ptr [rdi],eax
fffff607`893442cd f6ff            idiv    bh
fffff607`893442cf ff01            inc     dword ptr [rcx]
fffff607`893442d1 17              ???
fffff607`893442d2 c4              ???
fffff607`893442d3 9f              lahf
fffff607`893442d4 8e8fffff0060    mov     cs,word ptr [rdi+6000FFFFh]
fffff607`893442da 5a              pop     rdx
fffff607`893442db 9f              lahf
fffff607`893442dc 8e8fffff0000    mov     cs,word ptr [rdi+0FFFFh]
fffff607`893442e2 0000            add     byte ptr [rax],al
fffff607`893442e4 0000            add     byte ptr [rax],al
```

```
fffff607`893442e6 0000            add     byte ptr [rax],al
fffff607`893442e8 7527            jne     fffff607`89344311
fffff607`893442ea 6e              outs    dx,byte ptr [rsi]
```

使用 .cxr (Display Context Record) 命令，利用 !analyze 提供的参数 3 值来显示上下文记录。

dbgcmd

```
.cxr fffff607893436c4
```

使用 !vm 命令检查内存使用情况，例如查看正在使用的内核堆栈内存量。

dbgcmd

```
0: kd> !vm

Physical Memory:          1541186 (    6164744 Kb)
Available Pages:           470550 (    1882200 Kb)
ResAvail Pages:           1279680 (    5118720 Kb)

...

Kernel Stacks:              13686 (      54744 Kb)
```

使用 !stacks 命令和 2 个参数来查看有关堆栈的信息。 运行该命令可能需要一些时间。 检查输出中是否有重复的执行受阻模式，这些模式可能指向需要进一步调查的领域。

另请参阅

错误检查代码参考

错误检查 0x1C6：FAST_ERESOURCE_PRECONDITION_VIOLATION

项目 • 2024/02/22

FAST_ERESOURCE_PRECONDITION_VIOLATION 错误检查的值为 0x000001C6。 它表明当前线程正在执行对快速资源例程的无效调用。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

FAST_ERESOURCE_PRECONDITION_VIOLATION 参数

展开表

参数	说明
1	违规类型。 请参阅下面的值。
2	请参阅下面的值。
3	请参阅下面的值。
4	请参阅下面的值。

违规类型

```text
            0x0 : The Irql of the caller was greater than the maximum
allowed
                  Irql for the routine.
                2 - Irql of the caller.
                3 - Maximum allowed Irql of the routine.
            0x1 : The caller specified an invalid (i.e. uninitialized) owner
                  entry.
                2 - Pointer to the owner entry.
            0x2 : The caller specified an owner entry that was already
                  associated with a lock acquisition.
```

```
                2 - Pointer to the owner entry.
                3 - Pointer to the resource to which the owner entry is
already
                  associated.
            0x3 : The caller passed a legacy resource to a fast resource
                  routine.
                2 - Pointer to the resource.
            0x4 : The caller specified a resource that has outstanding lock
                  acquisitions.
                2 - Pointer to the resource.
            0x5 : The caller was executing inside of a DPC.
            0x6 : The caller was executing inside of a Special Kernel APC.
            0x7 : The caller did not ensure Normal Kernel APCs were
disabled.
            0x8 : The caller specified an owner entry that was not
associated
                  with a lock acquisition of the specified resource.
                2 - Pointer to the resource specified.
                3 - Pointer to the owner entry.
                4 - Pointer to the resource with which the owner entry is
                  associated.
            0x9 : The caller specified an owner entry that was not
associated
                  with the calling thread.
                2 - Pointer to the owner entry.
                3 - Pointer to the thread with which the owner entry is
                  associated.
            0xa : The caller specified an owner entry which has been
disowned.
                2 - Pointer to the owner entry.
            0xb : The caller specified an owner entry with a different
                  acquisition type than the caller indicated.
                2 - Pointer to the owner entry.
                3 - High-16 bits indicate the acquisition type of the owner
                  entry. Bottom-16 bits indicate the acquisition type
                  specified. 1 = Shared, 0 = Exclusive.
            0xc : The caller specified an owner entry that was not
associated
                  with a lock acquisition of the specified resource.
                2 - Pointer to the resource specified.
                3 - Pointer to the owner entry.
                4 - Pointer to the resource with which the owner entry is
                  associated.
            0xd : The caller specified an owner entry that has not been
                  disowned.
                2 - Pointer to the owner entry.
            0xe : The caller passed a fast resource to a legacy routine that
                  does not support fast resources.
                2 - Pointer to the resource.
            0xf : The caller passed a fast resource to a legacy routine that
                  supports fast resources, but the fast resource was not
                  initialized with EX_FAST_RESOURCE_ENABLE_LEGACY_APIS.
                2 - Pointer to the resource.
            0x10 : The caller passed invalid flags to
ExInitializeFastResource.
```

```
                2 - Pointer to the resource specified.
                3 - The flags specified.
            0x11 : The caller passed a thread other than the current thread
to
                   ExReleaseResourceForThreadLite.
                2 - Pointer to the resource specified.
                3 - The thread specified.
            0x12 : The caller attempted to disown a resource that had been
                   recursively acquired exclusive by the current thread.
                2 - Pointer to the resource specified.
            0x13 : The caller attempted to convert a resource acquisition
while
                   the calling thread had outstanding recursive acquisitions
of
                   the resource.
                2 - Pointer to the resource specified.
            0x14 : A thread exited with outstanding lock acquisitions.
                2 - The thread.
                3 - Pointer to one of the outstanding owner entries.
            0x15 : A thread exited with outstanding disowned lock
acquisitions.
                2 - The thread.
                3 - Pointer to one of the outstanding owner entries.
            0x16 : A call to ExConvertExclusiveToSharedLite was made by a
thread
                   that did not hold the specified resource exclusive.
                2 - Pointer to the resource.
                3 - Pointer to the thread.
```

原因

当前线程正在执行对快速资源例程的无效调用。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

另请参阅

错误检查代码参考

错误检查 0x1C7：STORE_DATA_STRUCTURE_CORRUPTION

项目 • 2024/02/22

STORE_DATA_STRUCTURE_CORRUPTION 错误检查的值为 0x000001C7。 这表明存储组件检测到其数据结构损坏。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅蓝屏错误疑难解答。

STORE_DATA_STRUCTURE_CORRUPTION 参数

展开表

参数	说明
1	损坏 ID。 请参阅下面的值。
2	请参阅下面的值。
3	请参阅下面的值。
4	请参阅下面的值。

损坏 ID

text

```
 0x0 : A chunk heap buffer's hash doesn't match.
    2 - Chunk heap buffer whose hash didn't match.
    3 - Expected buffer hash.
    4 - Page frame number of the corrupted page.

 0x1 : An unhandled exception occurred on the store thread and a chunk heap
buffer's hash doesn't match, which is likely the source of the exception.
    2 - Chunk heap buffer whose hash didn't match.
    3 - Expected buffer hash.
    4 - Page frame number of the corrupted page.

 0x2 : Page data appears corrupt during a read and the corresponding page
record's heap buffer hash doesn't match.
    2 - Chunk heap buffer whose hash didn't match containing the page record
```

```
of the data being read.
    3 - Expected buffer hash.
    4 - Page frame number of the corrupted page.

 0x3 : Page data appears corrupt during a read and the corresponding page
record has changed since the start of the read operation.
    2 - Pointer to the page location information snapped from the page
record that was found when the read was initiated.
    3 - Pointer to the page record currently in the page tree for the same
page key.
    4 - Reserved.
```

原因

存储组件检测到其数据结构损坏。

由于物理内存访问导致内存损坏，可能会出现此错误检查。 物理内存损坏的原因包括：

1. RAM 硬件缺陷
2. 驱动程序或设备通过错误的 DMA 操作或相关 MDL 不正确地修改物理页。
3. 硬件设备或固件破坏内存造成的损坏，如固件在电源转换时非法修改物理页。

有关 Windows 内存管理器的详细信息，请参阅由 Pavel Yosifovich、Mark E. Russinovich、David A. Solomon 和 Alex Ionescu 撰写的 Windows Internals 第 7 版第 1 部分。

解决方法

Windows 内存诊断工具

要调查此错误检查是否由 RAM 硬件缺陷引起，请运行 Windows 内存诊断工具。 在控制面板搜索框中键入“内存”，然后选择“诊断计算机内存问题”。运行测试后，使用事件查看器查看系统日志下的结果。 查找“内存诊断结果”条目以查看结果。

另请参阅

Bug 检查代码参考

Windows 内核模式内存管理器

错误检查 0x1C8：MANUALLY_INITIATED_POWER_BUTTON_HOLD

项目 • 2024/02/22

MANUALLY_INITIATED_POWER_BUTTON_HOLD 的值为 0x000001C8。

系统被配置为在用户按住电源按钮一定时间后启动错误检查。 这是一种诊断错误检查，用于在长时间按住电源按钮即将硬重置系统时捕获转储。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

请注意，在进行错误检查时，显示的不是标准的“蓝屏”，而是带有以下文字的黑色背景和完成百分比指示符：

“请松开电源按钮。 我们只需要几秒钟就能关闭。”

MANUALLY_INITIATED_POWER_BUTTON_HOLD 参数

蓝屏上显示以下参数。

展开表

参数	说明
1	按住电源按钮的时间（毫秒）。
2	指向 nt!_POP_POWER_BUTTON_TRIAGE_BLOCK 的指针。
3	保留。
4	保留。

另请参阅

使用电源按钮强制系统崩溃

内部展示 - 错误检查 0x1C8 MANUALLY_INITIATED_POWER_BUTTON_HOLD

ACPI 按钮设备

错误检查 0x1CA：SYNTHETIC_WATCHDOG_TIMEOUT

项目 • 2024/02/26

SYNTHETIC_WATCHDOG_TIMEOUT 错误检查的值为 0x000001CA。 系统范围的监视程序已过期。 这表示系统已挂起，没有处理计时器时钟周期。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SYNTHETIC_WATCHDOG_TIMEOUT 参数

展开表

参数	说明
1	自监视程序上次重置以来的时间（中断时间）。
2	当前中断时间。
3	当前 QPC 时间戳。
4	时钟处理器的索引。

原因

系统范围的监视程序已过期。 这表示系统已挂起，没有处理计时器时钟周期。

解决方法

!analyze 调试程序扩展显示有关 bug 检查的信息，有助于确定根本原因。 有关 WinDbg 和 **!analyze** 的更多信息，请参阅使用 !analyze 扩展和 !analyze。

另请参阅

Bug 检查代码参考

错误检查 0x1CB：INVALID_SILO_DETACH

项目 • 2024/02/26

INVALID_SILO_DETACH 错误检查的值为 0x000001CB。 它表示在退出之前，线程无法从接收器分离。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INVALID_SILO_DETACH 参数

展开表

参数	说明
1	指向附加线程的指针。
2	以前附加的接收器。
3	指向线程进程的指针。
4	保留。

原因

在退出之前，线程无法从接收器分离。

另请参阅

Bug 检查代码参考

错误检查 0x1CD：INVALID_CALLBACK_STACK_ADDRESS

项目 • 2024/02/22

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INVALID_CALLBACK_STACK_ADDRESS 错误检查的值为 0x000001CD。 回调堆栈是非法的用户模式地址。

INVALID_CALLBACK_STACK_ADDRESS 参数

无

错误检查 0x1CE：INVALID_KERNEL_STACK_ADDRESS

项目 • 2024/02/22

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

INVALID_KERNEL_STACK_ADDRESS 错误检查的值为 0x000001CE。 在上下文切换过程中遇到无效的初始内核堆栈地址。

INVALID_KERNEL_STACK_ADDRESS 参数

无

错误检查 0x1CF：HARDWARE_WATCHDOG_TIMEOUT

项目 • 2024/02/26

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

HARDWARE_WATCHDOG_TIMEOUT 错误检查的值为 0x000001CF。 这表示系统已挂起，没有处理计时器时钟周期。

HARDWARE_WATCHDOG_TIMEOUT 参数

展开表

参数	说明
1	自监视程序上次重置以来的时间（中断时间）。
2	当前中断时间。
3	当前 QPC 时间戳。
4	时钟处理器的索引。

错误检查 0x1D0：ACPI_FIRMWARE_WATCHDOG_TIMEOUT

项目 • 2024/02/22

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

ACPI_FIRMWARE_WATCHDOG_TIMEOUT 错误检查的值为 0x000001D0。

ACPI 驱动程序未能在预期分配时间内完成操作。

ACPI_FIRMWARE_WATCHDOG_TIMEOUT 参数

蓝屏上显示以下参数。

展开表

参数	说明
1	指向 AMLI 上下文的指针
2	指向 Aml 上下文的 Unicode 名称的指针
3	指向 ACPI 设备扩展的指针。
4	指向 ACPI 会审块的指针。

错误检查 0x1D2：WORKER_THREAD_INVALID_STATE

项目 • 2024/02/22

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WORKER_THREAD_INVALID_STATE 错误检查的值为 0x000001D2。

此错误表示执行工作线程处于无效状态。

WORKER_THREAD_INVALID_STATE 参数

展开表

参数	说明
1	故障类型
2	工作线程的地址
3	Reserved
4	Reserved

参数 1 值

0x0：正在终止的工作线程有未完成的 I/O

2 - 工作线程的地址

3 - 保留

4 - 保留

错误检查 0x1D3：WFP_INVALID_OPERATION

项目 • 2024/02/22

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WFP_INVALID_OPERATION 错误检查的值为 0x000001D3。 这表示 Windows 筛选平台标注执行了无效操作。

WFP_INVALID_OPERATION 参数

展开表

参数	说明
1	bugcheck 的子类型。
2	Reserved
3	Reserved
4	Reserved

参数 1 值

0x1：标注注入了具有多个 NET_BUFFERS 入站的 NBL。

2 - 保留。

3 - 指向 NBL 的指针。

4 - 保留。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并非常有助于确定根本原因。

错误检查 0x1D5：DRIVER_PNP_WATCHDOG

项目 • 2024/02/22

DRIVER_PNP_WATCHDOG 错误检查的值为 0x000001D5。 这表明驱动程序未能在特定时间内完成 PnP 操作。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DRIVER_PNP_WATCHDOG 参数

展开表

参数	说明
1	与 devnode 关联的服务的前几个字符。
2	在 Win10 RS4 及更高版本中指向 nt!TRIAGE_PNP_WATCHDOG 的指针。
3	负责 PnP 监视程序的线程。
4	监视程序布防后的毫秒数。

原因

这表明驱动程序未能在特定时间内完成 PnP 操作。 !analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

另请参阅

错误检查代码参考

错误检查 0x1D6：WORKER_THREAD_RETURNED_WITH_NON_DEFAULT_WORKLOAD_CLASS

项目 • 2024/02/26

WORKER_THREAD_RETURNED_WITH_NON_DEFAULT_WORKLOAD_CLASS 错误检查的值为 0x000001D6。 它表示工作线程更改了其工作负荷类，并且在返回前没有还原。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WORKER_THREAD_RETURNED_WITH_NON_DEFAULT_WORKLOAD_CLASS 参数

展开表

参数	说明
1	辅助角色例程的地址（使用 ln 查找负责任的驱动程序）
2	当前工作负荷类值。
3	工作项参数。
4	工作项地址。

原因

一个工作线程更改了其工作负荷类，但在返回前没有恢复。

另请参阅

Bug 检查代码参考

错误检查 0x1D7：EFS_FATAL_ERROR

项目 • 2024/02/26

EFS_FATAL_ERROR 错误检查的值为 0x000001D7。 它表明发生了 EFS 错误条件，无法在不丢失数据或数据损坏的情况下进行处理。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

EFS_FATAL_ERROR 参数

展开表

参数	说明
1	错误检查子类：**01** - 预卸载失败。
2	操作的 NTSTATUS 返回代码。
3	发生故障时的当前 IRP。
4	发生故障时的文件加密上下文。

原因

发生了 EFS 错误条件，无法在不丢失数据或数据损坏的情况下进行处理。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

检查参数 2 NTSTATUS 字段以尝试确定未返回 NT_SUCCESS 的原因。 对于调用加密预卸载的文件系统，这是唯一允许的预期值。

使用调试程序 !IRP 命令来检查参数 3 是否可能与 IRP 代码或其他问题相冲突。

另请参阅

错误检查代码参考

错误检查 0x1D8：UCMUCSI_FAILURE

项目 • 2024/02/22

UCMUCSI_FAILURE 错误检查的值为 0x000001D8。 它表示 UCSI 类扩展遇到了错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

UCMUCSI_FAILURE 参数

展开表

参数	说明
1	故障类型。 值：**0x0**：UCSI 命令超时，因为固件没有及时响应命令。 **0x1**：由于客户端驱动程序返回失败或固件返回错误代码，UCSI 命令执行失败。
2	UCSI 命令值。
3	如果非零，则是指向附加信息的指针（使用 `dt UcmUcsiCx!UCMUCSICX_TRIAGE`）。
4	保留。

原因

UcmUcsi 驱动程序遇到错误。 驱动程序找到了触发系统崩溃而非实时转储的设置。

解决方法

如果 UCSI 固件没有响应并且 UCSI 命令的 UcmUcsiCx 超时，或者 UCSI 固件在响应 UcmUcsiCx 发送的关键 UCSI 命令时显示错误，UCSI 命令通常会失败。

有关此故障原因的详细信息，请运行 `!rcdrkd.rcdrlogdump UcmUcsiCx`。

有关分析此错误检查的详细信息，请参阅此博客文章 - 调试 UCSI 固件故障。

另请参阅

Bug 检查代码参考

错误检查 0x1D9：HAL_IOMMU_INTERNAL_ERROR

项目 • 2024/02/22

HAL_IOMMU_INTERNAL_ERROR 错误检查的值为 0x000001D9。 它表示在 HAL IOMMU 库中检测到内部错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

HAL_IOMMU_INTERNAL_ERROR 参数

展开表

参数	说明
1	表示操作失败。 请参阅下面的值。
2	请参阅下面的值。
3	请参阅下面的值。
4	请参阅下面的值。

失败操作值

text

```
0x00 : Failed to delete IOMMU domain
     Parameter 2 - Status
     Parameter 3 - Pointer to the IOMMU domain object

0x01 : Failed to unmap pages from IOMMU domain
     Parameter 2 - Status
     Parameter 3 - Pointer to the IOMMU domain object
     Parameter 4 - Logical address

0x02 : Failed to leave IOMMU domain
     Parameter 2 - Status
     Parameter 3 - Pointer to the IOMMU domain object
```

原因

在 HAL IOMMU 库中检测到内部错误。

另请参阅

Bug 检查代码参考

错误检查 0x1DA：HAL_BLOCKED_PROCESSOR_INTERNAL_ERROR

项目 • 2024/02/22

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

HAL_BLOCKED_PROCESSOR_INTERNAL_ERROR 的值为 0x000001DA。 它表示在阻止的处理器库中检测到内部错误。

HAL_BLOCKED_PROCESSOR_INTERNAL_ERROR 参数

蓝屏上显示以下参数。

展开表

参数	说明
1	故障类型 - 请参阅下文
2	请参阅下文。
3	请参阅下文。
4	请参阅下文。

参数 1 值

plaintext

```
0x01 : Library initialization failure
    2 - NT status code
0x02 : Processor start failure
    2 - Processor index
    3 - APIC ID
0x03 : PPM package ID query failure
```

```
            2 - Processor index
        0x04 : PPM operation failure
            2 - Operation
                0x01 : MSR read
                0x02 : MSR write
                0x03 : I/O port read
                0x04 : I/O port write
                0x05 : Idle state registration
            3 - Processor index
            4 - PPM mailbox
        0x05 : A blocked processor has encountered a fatal exception.
            2 - Processor index
            3 - Vector number
            4 - Trap frame
        0x06 : PPM operation timeout
            2 - Operation
                0x01 : MSR read
                0x02 : MSR write
                0x03 : I/O port read
                0x04 : I/O port write
                0x05 : Idle state registration
            3 - Processor index
            4 - PPM mailbox
```

错误检查 0x1DB：IPI_WATCHDOG_TIMEOUT

项目 • 2024/02/29

IPI_WATCHDOG_TIMEOUT 错误检查的值为 0x000001DB。 它表示处理器卡在 IPI 循环中的时间超过了允许的时间。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

IPI_WATCHDOG_TIMEOUT 参数

展开表

参数	说明
1	表示 QPC 频率。
2	表示当前 QPC。
3	表示基线 QPC。
4	保留。

原因

处理器卡在 IPI 循环中的时间超过允许时间。

另请参阅

Bug 检查代码参考

错误检查 0x1DC：DMA_COMMON_BUFFER_VECTOR_ERROR

项目 • 2024/02/26

DMA_COMMON_BUFFER_VECTOR_ERROR 错误检查的值为 0x000001DC。 它表示驱动程序滥用了 DMA 矢量通用缓冲区 API。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DMA_COMMON_BUFFER_VECTOR_ERROR 参数

展开表

参数	说明
1	指示故障的类型。 请参阅下面的值。
2	请参阅下面的值。
3	请参阅下面的值。
4	请参阅下面的值。

故障类型

```text
0x01 : Wrong IRQL
    2 - Current IRQL.

x02 : Vector not empty.
    2 - Index of remaining buffer.
    3 - Virtual Address of remaining buffer.
    4 - Logical address of remaining buffer.

0x03 : Index out of bounds.
    2 - Number of available entries.
    3 - Index requested.
```

```
0x04 : Index freed.
    2 - Index requested.

0x05 : Common buffer leaked.
```

原因

驱动程序滥用了 DMA 矢量通用缓冲区 API。 !analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

另请参阅

错误检查代码参考

错误检查 0x1DD：BUGCODE_MBBADAPTER_DRIVER

项目 • 2024/02/26

BUGCODE_MBBADAPTER_DRIVER 错误检查的值为 0x000001DD。 这表示操作系统遇到了由 MBBCx 管理的网络驱动程序引起的错误。 MBBCx 以基于 KMDF 的 MBB 客户端驱动程序的形式为 MBB 设备提供移动宽带 (MBB) 媒体特定功能。 有关详细信息，请参阅移动宽带 (MBB) WDF 类扩展 (MBBCx) 简介。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

BUGCODE_MBBADAPTER_DRIVER 参数

展开表

参数	说明
1	以下故障代码之一 0 - FailureCode_CorruptedPrivateGlobals 1 - FailureCode_IrqlIsNotPassive 2 - FailureCode_IrqlNotLessOrEqualDispatch 3 - FailureCode_InvalidStructTypeSize 4 - FailureCode_InvalidPowerCapabilities 5 - FailureCode_InvalidWakeReason 6 - FailureCode_DisarmNotInProgress 7 - FailureCode_WdfObjectCreateFailed
2	Reserved
3	Reserved
4	Reserved

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

参数 1 说明了违规类型。 查看调用堆栈以确定行为异常的驱动程序。

另请参阅

移动宽带 (MBB) WDF 类扩展 (MBBCx) 的简介

Bug 检查代码参考

错误检查 0x1DE：BUGCODE_WIFIADAPTER_DRIVER

项目 • 2024/02/22

BUGCODE_WIFIADAPTER_DRIVER 错误检查的值为 0x000001DE。 这表示操作系统遇到了由 WiFiCx 管理的网络驱动程序引起的错误。 Wi-Fi WDF 类扩展 (WiFiCx) 支持基于 KMDF 的 Wi-Fi 设备客户端驱动程序。 有关详细信息，请参阅 Wi-Fi WDF 类扩展 (WiFiCx) 简介。

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

BUGCODE_WIFIADAPTER_DRIVER 参数

展开表

参数	说明
1	此处列出的 WiFiCx BugCheck 子代码值。 0x0 - WiFiCx 专用全局已损坏。 0x1：一个驱动程序以非法 IRQL 调用了需要被动 IRQL 的 WiFiCx API。 2 - 当前 IRQL。 0x2：驱动程序调用了要求 IRQL 小于或等于非法 IRQL 的 WiFiCx API。 2 - 当前 IRQL。 0x3：向 WiFiCx API 传递了一个大小无效的结构。 2 - 驱动程序传递的结构大小。 3 - OS 的 WiFiCx 版本使用的结构大小。 请注意，这并不要求与传递的结构完全匹配，但 WiFiCx 旧版本的结构大小应 <= 于新版本的结构大小。 0x4：驱动程序报告了无效唤醒原因。 2 - 报告的唤醒原因。

参数	说明
	0x5：驱动程序向 WiFiCx API 传递了无效或损坏的 WIFI_POWER_OFFLOAD_LIST。 2 - 指向 WIFI_POWER_OFFLOAD_LIST 的指针。 0x6：驱动程序向 WiFiCx API 传递了无效或损坏的 WIFI_WAKE_SOURCE_LIST。 2 - 指向 WIFI_WAKE_SOURCE_LIST 的指针。 0x7：驱动程序向 WiFiCx API 传递了无效或未初始化的 WIFI_INTERFACE_CONTEXT。 2 - 指向传递的 WIFI_INTERFACE_CONTEXT 的指针。 0x8：驱动程序将带有无效 WIFI_ADAPTER_TX_DEMUX 的 WIFI_ADAPTER_TX_DEMUX_TYPE 传递给 WiFiCx API。 2 - 指向传递的 WIFI_ADAPTER_TX_DEMUX 的指针。 3 - 传递的 WIFI_ADAPTER_TX_DEMUX 的 WIFI_ADAPTER_TX_DEMUX_TYPE。 0x9：驱动程序向 WiFiCx API 传递了范围无效的 WIFI_ADAPTER_TX_DEMUX。 2 - 指向传递的 WIFI_ADAPTER_TX_DEMUX 的指针。 3 - 传递的 WIFI_ADAPTER_TX_DEMUX 的范围。 0xA：驱动程序报告的唤醒时间无效。 唤醒原因应在解除唤醒回调中报告。 0xB：驱动程序在 WiFiCx 结构中使用了一个已弃用的字段。 2 - 指向表示已弃用字段的字符串的指针。
2	依赖于参数 1
3	依赖于参数 1
4	Reserved

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

参数 1 说明了违规类型。 查看调用堆栈以确定行为异常的驱动程序。

另请参阅

Wi-Fi WDF 类扩展 (WiFiCx) 简介

Bug 检查代码参考

错误检查 0x1DF：PROCESSOR_START_TIMEOUT

项目 • 2024/02/22

PROCESSOR_START_TIMEOUT 错误检查的值为 0x000001DF。 这表明处理器未能在允许的时间内启动。 处理器启动发生在操作系统初始化的早期。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PROCESSOR_START_TIMEOUT 参数

展开表

参数	说明
1	处理器状态的虚拟地址。
2	保留。
3	NT 处理器编号。
4	处理器的本地单元 ID。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

另请参阅

错误检查代码参考

错误检查 0x1E4：VIDEO_DXGKRNL_SYSMM_FATAL_ERROR

项目 • 2024/02/26

VIDEO_DXGKRNL_SYSMM_FATAL_ERROR 错误检查的值为 0x000001E4。 这表明 Microsoft DirectX 图形内核系统内存管理器检测到违规。

展开表

参数	说明
1	错误检查的子代码 0x1：物理对象类型无效 0x2：启用了无效原因的 IOMMU 0x3：IOMMU 因无效原因而被禁用 0x4：ADL 是针对非锁定内存生成的 0x5：存在 ADL 时内存被解锁 0x6：适配器对象被泄露 0x7：ADL 已泄漏 0x8：逻辑块池已泄漏 0x9：逻辑块池分配器已泄漏 0xA：域重新附加失败
2	Reserved
3	Reserved
4	Reserved

重要

这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅蓝屏错误疑难解答。

错误检查 0x1E9：ILLEGAL_ATS_INITIALIZATION

项目 • 2024/02/22

ILLEGAL_ATS_INITIALIZATION 错误检查的值为 0x000001E9。 这表明驱动程序试图在已启用共享虚拟内存 (SVM) 的设备上非法启用地址转换服务 (ATS)。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

ILLEGAL_ATS_INITIALIZATION 参数

展开表

参数	说明
1	物理设备对象。
2	Reserved
3	Reserved
4	Reserved

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x1EA：SECURE_PCI_CONFIG_SPACE_ACCESS_VIOLATION

项目 • 2024/02/22

SECURE_PCI_CONFIG_SPACE_ACCESS_VIOLATION 错误检查的值为 0x000001EA。 这表明，由于启用了安全 PCI，禁止通过直接映射 PCI MCFG 范围从 VTL0 访问 PCI 配置空间区域。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

SECURE_PCI_CONFIG_SPACE_ACCESS_VIOLATION 参数

展开表

参数	说明
1	导致了异常的操作码。
2	导致异常的设备的 RID。
3	配置空间访问偏移量。
4	导致异常的指令地址。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x1EB：DAM_WATCHDOG_TIMEOUT

项目 • 2024/02/22

DAM_WATCHDOG_TIMEOUT 错误检查的值为 0x000001EB。 这表明，在设备从新式待机状态恢复后，桌面活动审查器 (DAM) 无法在分配的时间段内解冻非免责用户会话进程。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

DAM_WATCHDOG_TIMEOUT 参数

展开表

参数	说明
1	指向 DAM 用户会话延迟上下文的指针。
2	Reserved
3	Reserved
4	Reserved

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

错误检查 0x1ED：HANDLE_ERROR_ON_CRITICAL_THREAD

项目 • 2024/02/22

HANDLE_ERROR_ON_CRITICAL_THREAD 无误检查的值为 0x000001ED。 这表明在系统关键线程的内核模式下检测到无效句柄访问问题。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

HANDLE_ERROR_ON_CRITICAL_THREAD 参数

展开表

参数	说明
1	保留
2	Reserved
3	Reserved
4	Reserved

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

另请参阅

错误检查代码参考

错误检查 0x356：XBOX_ERACTRL_CS_TIMEOUT

项目 • 2024/02/26

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

XBOX_ERACTRL_CS_TIMEOUT 错误检查的值为 0x00000356。

eractrl.sys 驱动程序无法在长时间连接后将 Xbox 主机转换到已连接待机状态或从连接待机状态转换。 eractrl.sys 驱动程序用于启动支持 VM 的 KM 驱动程序，并帮助管理 VM 的生命周期。

XBOX_ERACTRL_CS_TIMEOUT 参数

展开表

参数	说明
1	CS 退出
2	Reserved
3	Reserved
4	Reserved

错误检查 0xBFE：BC_BLUETOOTH_VERIFIER_FAULT

项目 • 2024/03/04

BC_BLUETOOTH_VERIFIER_FAULT 错误检查的值为 0x00000BFE。 这表示驱动程序已导致冲突。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

BC_BLUETOOTH_VERIFIER_FAULT 参数

展开表

参数	说明
1	蓝牙验证程序错误的子类型。 `0x1 : An attempt was made to submit a Bluetooth Request Block that is already in use 2 - Brb pointer 3 - Reserved 4 - Reserved 0x2 : An attempt was made to free a Bluetooth Request Block that is in use 2 - Brb pointer 3 - Reserved 4 - Reserved 0x3 : An attempt was made to allocate or initialize an invalid BRB type 2 - Brb pointer 3 - pdo extension (if available) 4 - Reserved 0x4 : Invalid Bluetooth Request Block pointer was submitted 2 - Brb pointer 3 - Reserved 4 - Reserved 0x5 : A Bluetooth Request Block with an invalid size was submitted 2 - Brb pointer 3 - Actual Size 4 - Expected Size 0x6 : The IOCTL_BTH_GET_DEVICE_INFO was called with invalid parameters 2 - Reserved 3 - Reserved 4 - Reserved 0x7 : BRB_L2CA_UNREGISTER_SERVER was submitted with an invalid server handle 2 - Server handle 3 - Reserved 4 - Reserved 0x8 : BRB_L2CA_CLOSE_CHANNEL was submitted with an invalid channel handle 2 - Brb pointer 3 - Channel handle 4 - Reserved 0x9 : BRB_SCO_UNREGISTER_SERVER was submitted with`

参数	说明
	`an invalid server handle 2 - Server handle 3 - Reserved 4 - Reserved`
2	请参阅参数 1
3	请参阅参数 1
4	请参阅参数 1

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。 参数 1 说明了违规类型。 查看调用堆栈以确定行为异常的驱动程序。

错误检查 0xBFF：BC_BTHMINI_VERIFIER_FAULT

项目 • 2024/03/04

BC_BTHMINI_VERIFIER_FAULT 错误检查的值为 0x00000BFF。 这表示蓝牙微型端口可扩展驱动程序验证程序已捕获冲突。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

BC_BTHMINI_VERIFIER_FAULT 参数

展开表

参数	说明
1	蓝牙验证程序错误的子类型。 `0x1 : An attempt was made to return a packet with type that mis-matched its original request. 2 - Returned packet type 3 - Expected packet type 4 - Reserved 0x2 : An attempt was made to return an unexpected status code and caused the packet to be discarded. 2 - Unexpected return status 3 - Reserved 4 - Reserved 0x3 : Incorrect output buffer size was returned to indicate number of bytes written by the lower transport driver. 2 - Unexpected buffer size 3 - Expected buffer size 4 - Reserved`
2	请参阅参数 1
3	请参阅参数 1
4	请参阅参数 1

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。 参数 1 说明了违规类型。 查看调用堆栈以确定行为异常的驱动程序。

错误检查 0x20001: HYPERVISOR_ERROR

项目 • 2024/02/26

HYPERVISOR_ERROR 错误检查的值为 0x00020001。 这表示虚拟机监控程序遇到致命错误。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

HYPERVISOR_ERROR 参数

展开表

参数	说明
1	保留
2	Reserved
3	Reserved
4	Reserved

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

Bug 检查 0x1000007E: SYSTEM_THREAD_EXCEPTION_NOT_HANDLED_M

项目 • 2024/03/01

SYSTEM_THREAD_EXCEPTION_NOT_HANDLED_M Bug 检查的值为 0x1000007E。 这表明系统线程产生了错误处理程序未捕获的异常。

Bug 检查 0x1000007E 的含义和参数与 **Bug 检查 0x7E** (SYSTEM_THREAD_EXCEPTION_NOT_HANDLED) 相同。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

bug 检查0x1000007F：UNEXPECTED_KERNEL_MODE_TRAP_M

项目 • 2024/02/14

UNEXPECTED_KERNEL_MODE_TRAP_M bug 检查的值为 0x1000007F。 这表示 Intel CPU 生成了陷阱，内核未能捕获此陷阱。

Bug 检查 0x1000007F的含义和参数与 **bug 检查 0x7F**（UNEXPECTED_KERNEL_MODE_TRAP） 相同。

> **ⓘ 重要**
>
> 本文面向程序员。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅 **“排查蓝屏错误”**。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

Bug 检查 0x1000008E: KERNEL_MODE_EXCEPTION_NOT_HANDLED_M

项目 • 2024/03/01

KERNEL_MODE_EXCEPTION_NOT_HANDLED_M 检查的值为 0x1000008E。 这表示内核模式程序生成了错误处理程序未捕获的异常。

Bug 检查 0x1000008E 的含义和参数与 **Bug 检查 0x8E** (KERNEL_MODE_EXCEPTION_NOT_HANDLED) 相同。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

Bug 检查 0x100000EA：THREAD_STUCK_IN_DEVICE_DRIVER_M

项目 • 2024/03/01

THREAD_STUCK_IN_DEVICE_DRIVER_M Bug 检查的值为 0x100000EA。 这表示设备驱动程序中的线程在无休止地旋转。

Bug 检查 0x100000EA 的含义和参数与 **Bug 检查 0xEA** (THREAD_STUCK_IN_DEVICE_DRIVER) 相同。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

错误检查 0x4000008A：THREAD_TERMINATE_HELD_MUTEX

项目 • 2024/02/22

THREAD_TERMINATE_HELD_MUTEX 错误检查的值为 0x4000008A。 这表明驱动程序在一个线程上获取了一个 Mutex，而该线程在释放 Mutex 之前就已退出。 造成这种情况的原因可能是驱动程序返回用户模式时没有释放 Mutex，或者是驱动程序获取 Mutex 后发生异常，从而导致正在运行的线程被终止。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

THREAD_TERMINATE_HELD_MUTEX 参数

展开表

参数	说明
1	拥有 KMUTEX 的 KTHREAD 的地址。
2	拥有的 KMUTEX 的地址。
3	Reserved
4	Reserved

原因

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

要进行调查，请查看调用堆栈。 如果堆栈上有一个驱动程序直接被系统异常处理例程和线程终止例程所跟踪，那么该驱动程序就存在问题并需要进行修复，以免在持有内核 Mutex 时导致未处理异常。 如果堆栈只显示正常的线程终止代码，而没有牵涉到驱动程序，请运行 !pool 或在 Mutex 地址（参数 2）上使用**ln（列出最近符号）**，并查看是否能找到其所有者。 此错误几乎肯定会出现在该 Mutex 所有者的代码中。

错误检查 0xC0000218：STATUS_CANNOT_LOAD_REGISTRY_FILE

项目 • 2024/03/04

STATUS_CANNOT_LOAD_REGISTRY_FILE 错误检查的值为 0xC0000218。 这表示无法加载注册表文件。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

STATUS_CANNOT_LOAD_REGISTRY_FILE 参数

展开表

参数	说明
1	无法加载的注册表配置单元的名称的地址。
2	零（保留）
3	零（保留）
4	零（保留）

此错误检查显示一条描述性文本消息。 损坏文件的名称将作为消息的一部分显示。

原因

如果无法加载必要的注册表配置单元文件，就会发生此错误。 通常，这意味着文件已损坏或缺失。

在极少数情况下，此错误可能是由损坏内存中注册表映像的驱动程序或该区域中的内存错误引起的。

解决方法

请尝试使用操作系统提供的启动恢复机制（例如，启动修复、恢复控制台或紧急恢复磁盘）。 如果问题是注册表文件遗失或损坏，通常可以解决问题。

错误检查 0xC000021A：WINLOGON_FATAL_ERROR

项目 • 2024/03/04

WINLOGON_FATAL_ERROR 错误检查的值为 0xC000021A。 这意味着 Winlogon 进程意外终止。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WINLOGON_FATAL_ERROR 参数

展开表

参数	说明
1	标识问题的字符串
2	错误代码
3	预留
4	预留

原因

当用户模式子系统（如 WinLogon 或客户端服务器运行时子系统 (CSRSS)）受到致命威胁，安全性无法再得到保证时，就会发生此错误。 作为响应，操作系统切换到内核模式。 没有 WinLogon 或 CSRSS，Microsoft Windows 无法运行。 因此，这是为数不多的用户模式服务故障会导致系统关闭的情况之一。

不匹配的系统文件也可能导致此错误。 如果已从备份还原硬盘，则可能会出现这种不匹配。 某些备份程序可能会跳过还原它们确定正在使用的系统文件。

解决方法

在这种情况下，运行内核调试器可能没有用处，因为实际错误发生在用户模式进程中。

解决用户模式设备驱动程序、系统服务或第三方应用程序中的错误

由于错误检查 0xC000021A 发生在用户模式进程中，最常见的罪魁祸首是第三方应用程序。 如果错误发生在安装新的或更新的设备驱动程序、系统服务或第三方应用程序之后，则应删除或禁用新软件以找出原因。 请与软件制造商联系以获取可能的更新。

解决不匹配的系统文件问题

如果最近从备份中还原了硬盘，请检查制造商是否提供了备份/恢复程序的更新版本。

- 查看最近安装的应用程序。 要执行此操作，请导航到控制面板中的“卸载或更改程序”，然后按安装日期对已安装的应用程序进行排序。
- 检查事件查看器中的系统日志，以获取可能有助于查明导致错误的设备或驱动程序的其他错误消息。 有关详细信息，请参阅打开事件查看器。 在系统日志中查找与蓝屏同时出现的严重错误。
- 检查事件查看器中的系统日志，以获取可能有助于查明导致错误的设备或驱动程序的其他错误消息。 在系统日志中查找与蓝屏同时出现的严重错误。

注解

- 使用系统文件检查器工具修复丢失或损坏的系统文件。 系统文件检查器是 Windows 中的一个实用工具，供用户扫描 Windows 系统文件中的损坏情况并恢复损坏的文件。 使用以下命令运行系统文件检查器工具 (SFC.exe)。

  控制台

  ```
  SFC /scannow
  ```

  有关详细信息，请参阅使用系统文件检查器工具修复丢失或损坏的系统文件。
- 运行病毒检测程序。 病毒可以感染为 Windows 格式化的所有类型的硬盘，由此导致的磁盘损坏可能会生成系统错误检查代码。 确保病毒检测程序检查主启动记录是否存在感染。
- 请验证系统是否安装了最新的更新。 要检测系统上安装的版本，请选择**开始**，选择**运行**，键入 **winver**，然后按 ENTER。 **关于 Windows**对话框显示 Windows 版本号（以及服务包的版本号，如果已安装）。

使用安全模式

请考虑使用安全模式来隔离元素以进行故障排除，并在必要时使用 Windows。 在 Windows 启动过程中，使用安全模式只会加载所需的最低驱动程序和系统服务。

若要进入安全模式，请使用设置中的**更新和安全**。 选择**恢复**>**高级启动**以启动进入维护模式。 在生成的菜单中，选择**故障排除**>**高级选项**>**启动设置**>**重启**。 Windows 重启并显示**启动设置**界面后，选择选项 4、5 或 6 以启动进入安全模式。

在启动时按下功能键（例如 F8）也可以进入安全模式。 有关具体的启动选项，请参阅计算机制造商提供的信息。

错误检查 0xC0000221: STATUS_IMAGE_CHECKSUM_MISMATCH

项目 • 2024/03/04

STATUS_IMAGE_CHECKSUM_MISMATCH 错误检查的值为 0xC0000221。 错误检查表明驱动程序或系统 DLL 已损坏。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是 Microsoft 客户，并且你的计算机显示蓝屏错误代码，请参阅**对蓝屏错误进行故障排除**。

STATUS_IMAGE_CHECKSUM_MISMATCH 参数

无。

原因

此错误检查是由驱动程序或其他系统文件中的严重错误引起的。 文件头校验和与预期的校验和不匹配。

此错误检查也可能是由文件 I/O 路径中的硬件故障引起的，例如磁盘错误、RAM 故障或页面文件损坏。

解决方法

要解决此错误检查，请尝试以下选项：

- 如果可以登录到系统，请在事件查看器中查看“系统日志”以获取更多错误消息，这些错误消息可能有助于识别导致错误的设备或驱动程序。
- 使用系统文件检查器工具修复丢失或损坏的系统文件。 系统文件检查器 (SFC.exe) 是一种 Windows 实用工具，可用于扫描和还原损坏的 Windows 系统文件。

  使用以下命令运行系统文件检查器：

  控制台

  ```
  SFC /scannow
  ```

有关详细信息，请参阅使用系统文件检查器工具修复丢失或损坏的系统文件。

- 使用 Windows 磁盘扫描实用工具检查文件系统错误。 在文件资源管理器中，按住或右键单击要扫描的驱动器，然后选择**属性**。 选择**工具**选项卡。在**错误检查**下，选择**检查**。
- 检查硬盘驱动器上是否有足够的可用空间。 操作系统和某些应用程序需要足够的可用空间来创建交换文件和完成其他功能。 确切的要求因系统配置而异，但通常最好有 10% 到 15% 的可用空间。
- 磁盘错误可能会导致文件损坏。 运行 `Chkdsk /f /r` 以检测和解决文件系统中的任何结构性损坏。 在系统分区上开始磁盘扫描之前，必须重启系统。
- 请与制造商联系，查看是否有更新的系统 BIOS 或固件。 禁用 BIOS 或其他固件设置的内存缓存也可能会解决此错误。
- 可以在现有的 Windows 副本上运行就地安装或恢复。 就地安装或恢复会保留所有注册表设置和配置信息，但会替换所有系统文件。
- 运行病毒检测程序。 病毒可以感染为 Windows 格式化的所有类型的硬盘。 由此产生的磁盘损坏可能会生成系统错误检查代码。 确保病毒检测程序检查主启动记录是否存在感染。
- 如果这些方法失败，请重新安装 Windows，然后从备份还原系统。

另请参阅

错误检查代码参考

错误检查 0xDEADDEAD：MANUALLY_INITIATED_CRASH1

项目 • 2024/03/08

MANUALLY_INITIATED_CRASH1 错误检查的值为 0xDEADDEAD。 这表示发生了手动启动的崩溃。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果你是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

MANUALLY_INITIATED_CRASH1 参数

展开表

参数	说明
1	保留
2	预留
3	预留
4	预留

内核实时转储代码引用

项目 • 2024/10/16

本部分包含可能会出现的常见内核实时转储代码的说明。 实时转储会不重置 OS，但允许在操作系统可继续的异常情况下捕获内存信息。

> **备注**
>
> 本主题是面向程序员的。 如果你是一个客户，你的系统显示了带有错误检查代码的蓝屏，请参阅**蓝屏错误疑难解答**。

内核实时转储与错误检查的比较

使用传统错误检查时，PC 将重置，并且用户的工作将中断。 内核实时转储的目标是收集数据以对异常情况进行故障排除，但允许 OS 继续操作。 与“非致命”但影响严重的失败和挂起的错误检查相比，这可以减少停机时间。 如果可以将 OS 恢复到已知的良好状态，则使用内核实时转储。 例如，子系统的硬件重置（如视频/显示、USB3 或 Wi-Fi）可以允许这些系统恢复到已知的良好状态，同时对用户的影响降到最低。

内核实时转储创建内核内存的一致性快照，并将其保存到转储文件以供将来分析。 为了最大限度地减少对性能的影响，使用内存复制技术在短时间内创建转储文件。 此外，还会限制实时转储的收集，从而最大程度地减少对用户的影响。

内核实时转储适用于一类问题，即需要花费很长时间，但并未出现任何技术故障的问题。 启动操作时，可以初始化监视器计时器。 如果监视器在预期时间内在操作完成之前过期，则可以执行系统的实时转储。 然后，可以通过遍历该操作的调用堆栈和相关等待链来分析转储，以调查它未在预期的时间范围内完成的原因。

当某些操作失败，代码所有者记录失败的原因并能确定原因时，系统日志很有用。 使用监视器计时器的实时转储尝试捕获预期和记录外的故障路径。 但与每个故障一样，系统日志可能会标识其他问题，这些原因可能提供有关失败的特定根本原因的线索。

内核实时转储文件内容

与常规转储文件类似，实时转储文件可能包含（具有次要数据）的小型转储以及完整的内核转储，后者中也可能包含与活动转储类似的用户模式内存。 有关转储文件内容的常规信息，请参阅多种内核模式转储文件。 某些实时转储只会尝试捕获小型转储，因为它们旨在捕获特定的硬件相关数据，而另一些则可能尝试捕获更大的内核实时转储。

为了性能、文件大小以及转储捕获的可靠性，某些信息不包括在内，例如来自备用列表和文件缓存的页面。

实时转储文件通常包含内存页面，例如：

- KdDebuggerBlock
- 加载的模块列表

对于每个处理器，内核转储中会捕获以下信息：

- KiProcessorBlock
- PRCBs
- 当前堆栈
- 当前页面目录表
- KI_USER_SHARED_DATA
- NTOS 内核映像
- HAL 映像

内核转储中的其他信息可能包括：

- 线程/内存状态
- 内存中日志记录

某些实时转储可能包含用户模式进程页面。

某些实时转储可能包含其他特定于域的数据，例如 USB 故障种特定于 USB 的数据。

部分内核实时转储文件

当实时转储无法可靠地捕获所有预期的内存页面时，可能会生成部分内核实时转储文件。 将通过捕获包含在其他页面之前生成有效转储所需的重要数据的页面，筛选在部分转储中捕获的信息并确定其优先级。 例如，当实时转储包括用户页面时，内核页面优先于用户页面。 在某些情况下，没有足够的资源可用于捕获所有预期的可选内存页面，因此转储文件中可能缺少内存。 WinDbg 调试器仍应识别转储文件，但在尝试转储内存时可能会显示错误。 如果调试器在尝试转储位于某个地址的内存时显示错误，则可以使用 !pte 扩展来检查地址的 PTE 是否有效。 这可以帮助确定是内存地址确实无效，还是页面有效只是在转储文件中不可用。

分析实时转储文件

发生实时转储时，可以使用用于其他内存转储文件的技术来分析转储文件。 若要了解故障期间的内存内容，需要了解处理器内存寄存器和程序集编程。

有关详细信息，请参阅：

- 使用 WinDbg 分析内核模式转储文件
- !analyze
- 处理器体系结构

使用 WinDbg 显示实时转储终止代码信息

如果此主题中没有某个实时转储代码的说明信息，请在 Windows 调试程序 (WinDbg) 中使用以下语法（内核模式）和 **!analyze** 扩展，并将 `<code>` 替换为某个错误检查代码：

```
!analyze -show <code>
```

输入此命令会使 WinDbg 显示指定的实时转储代码的相关信息。 如果默认数字基数不是 16，则为 `<code>` 添加前缀 **0x**。

向 !analyze 命令提供实时转储代码参数，以显示任何可用的参数信息。 例如，若要显示有关参数 1 值为 0x3003 的 错误检查 0x144 BUGCODE_USB3_DRIVER 的信息，请使用 `!analyze -show 0x144 0x3003`，如下所示。

dbgcmd

```
0: kd> !analyze -show 0x144 0x3003
BUGCODE_USB3_DRIVER (144)
This bugcheck usually happens when the USB3 core stack detects an invalid
operation being performed by a USB client. This bugcheck may also occur
due to hardware failure on a USB Boot Device.
Arguments:
Arg1: 0000000000003003, USB3_WER_BUGCODE_USBHUB3_DEVICE_ENUMERATION_FAILURE
    A USB device failed enumeration.
Arg2: 0000000000000000, USBHUB3_LIVEDUMP_CONTEXT
Arg3: 0000000000000000, 0
Arg4: 0000000000000000, 0
```

要下载 WinDbg，请参阅 适用于 Windows 的调试工具。 若要了解有关 WinDbg 开发工具的详细信息，请参阅 Windows 调试入门。

实时转储文件位置

实时转储默认存储在“C:\WINDOWS\LiveKernelReports”目录中。

完全转储：`%systemroot%\LiveKernelReports\*.dmp`

小型转储：`%systemroot%\LiveKernelReports\<ComponentName>\*.dmp`

目录结构用于存储不同组件的实时转储。

```dos
NDIS
PDCRevocation
PoW32kWatchdog
USBHUB3
WATCHDOG
```

实时转储注册表项

有关系统生成的实时内核报表的配置选项的详细信息，请参阅 WER 设置。

使用 PowerShell 手动触发实时转储

1. 打开和管理员 PowerShell 提示符。
2. 使用 Get-StorageSubSystem PowerShell 命令获取 StorageSubSystem 易记名称。

```PowerShell
C:\> Get-StorageSubSystem
FriendlyName                    HealthStatus OperationalStatus
------------                    ------------ -----------------
Windows Storage on 10-2411-PC   Healthy      OK
```

3. 使用 Get-StorageDiagnosticInfo 生成上述子系统的实时转储（以及其他诊断日志）。 有关详细信息，请参阅 Get-StorageDiagnosticInfo。

```PowerShell
C:\> Get-StorageDiagnosticInfo -StorageSubSystemFriendlyName "Windows
Storage on 10-2411-PC" -IncludeLiveDump -DestinationPath
C:\destinationfolder
```

4. 输出将指示正在生成请求的信息。

```PowerShell
Gathering storage subsystem diagnostic information
Running
```

```
[ooooooooooo
]
```

5. 转储将位于 `[DestinationPath]\localhost` 内。

PowerShell

```
C:\> dir C:\destinationfolder\localhost\*.dmp
  Directory: C:\destinationfolder\localhost
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         5/5/2016   1:08 PM      867135488 LiveDump.dmp
```

6. 使用调试程序在转储文件上运行 !analyze 将指示这是 LIVE_SYSTEM_DUMP (161) 的实时转储代码。

内核实时转储代码

下表提供了内核实时转储代码的链接。

展开表

代码	名称
0x000000AB	SESSION_HAS_VALID_POOL_ON_EXIT
0x00000117	VIDEO_TDR_TIMEOUT_DETECTED
0x00000141	VIDEO_ENGINE_TIMEOUT_DETECTED
0x00000142	VIDEO_TDR_APPLICATION_BLOCKED
0x00000156	WINSOCK_DETECTED_HUNG_CLOSESOCKET_LIVEDUMP
0x0000015C	PDC_WATCHDOG_TIMEOUT_LIVEDUMP
0x0000015D	SOC_SUBSYSTEM_FAILURE_LIVEDUMP
0x0000015E	BUGCODE_NDIS_DRIVER_LIVE_DUMP
0x0000015F	CONNECTED_STANDBY_WATCHDOG_TIMEOUT_LIVEDUMP
0x00000161	LIVE_SYSTEM_DUMP
0x00000165	CLUSTER_CSV_STATUS_IO_TIMEOUT_LIVEDUMP
0x00000166	CLUSTER_RESOURCE_CALL_TIMEOUT_LIVEDUMP

代码	名称
0x00000167	CLUSTER_CSV_SNAPSHOT_DEVICE_INFO_TIMEOUT_LIVEDUMP
0x00000168	CLUSTER_CSV_STATE_TRANSITION_TIMEOUT_LIVEDUMP
0x00000169	CLUSTER_CSV_VOLUME_ARRIVAL_LIVEDUMP
0x0000016A	CLUSTER_CSV_VOLUME_REMOVAL_LIVEDUMP
0x0000016B	CLUSTER_CSV_CLUSTER_WATCHDOG_LIVEDUMP
0x0000016F	CLUSTER_CSV_STATE_TRANSITION_INTERVAL_TIMEOUT_LIVEDUMP
0x00000175	PREVIOUS_FATAL_ABNORMAL_RESET_ERROR
0x00000179	CLUSTER_CLUSPORT_STATUS_IO_TIMEOUT_LIVEDUMP
0x0000017C	PDC_LOCK_WATCHDOG_LIVEDUMP
0x0000017D	PDC_UNEXPECTED_REVOCATION_LIVEDUMP
0x00000187	VIDEO_DWMINIT_TIMEOUT_FALLBACK_BDD
0x00000188	CLUSTER_CSVFS_LIVEDUMP
0x00000190	WIN32K_CRITICAL_FAILURE_LIVEDUMP
0x00000193	VIDEO_DXGKRNL_LIVEDUMP
0x00000195	SMB_SERVER_LIVEDUMP
0x00000198	UFX_LIVEDUMP
0x0000019D	CLUSTER_SVHDX_LIVEDUMP
0x000001A1	WIN32K_CALLOUT_WATCHDOG_LIVEDUMP
0x000001A3	CALL_HAS_NOT_RETURNED_WATCHDOG_TIMEOUT_LIVEDUMP
0x000001A4	DRIPS_SW_HW_DIVERGENCE_LIVEDUMP
0x000001A5	USB_DRIPS_BLOCKER_SURPRISE_REMOVAL_LIVEDUMP
0x000001A6	BLUETOOTH_ERROR_RECOVERY_LIVEDUMP
0x000001A7	SMB_REDIRECTOR_LIVEDUMP
0x000001A8	VIDEO_DXGKRNL_BLACK_SCREEN_LIVEDUMP
0x000001A9	DIRECTED_FX_TRANSITION_LIVEDUMP
0x000001B0	VIDEO_MINIPORT_FAILED_LIVEDUMP

代码	名称
0x000001B8	VIDEO_MINIPORT_BLACK_SCREEN_LIVEDUMP
0x000001C4	DRIVER_VERIFIER_DETECTED_VIOLATION_LIVEDUMP
0x000001C5	IO_THREADPOOL_DEADLOCK_LIVEDUMP
0x000001C9	USER_MODE_HEALTH_MONITOR_LIVEDUMP
0x000001CC	EXRESOURCE_TIMEOUT_LIVEDUMP
0x000001D1	TELEMETRY_ASSERTS_LIVEDUMP
0x000001D4	UCMUCSI_LIVEDUMP
0x000001E1	DEVICE_DIAGNOSTIC_LOG_LIVEDUMP
0x000001F5	APPLICATION_HANG_KERNEL_LIVEDUMP
0x000021C8	MANUALLY_INITIATED_BLACKSCREEN_HOTKEY_LIVE_DUMP

这些停止代码可用于实时转储或设备错误检查。

展开表

代码	名称
0x00000124	WHEA_UNCORRECTABLE_ERROR
0x00000144	BUGCODE_USB3_DRIVER
0x00000164	WIN32K_CRITICAL_FAILURE

另请参阅

错误检查代码参考

!analyze

错误检查（蓝屏）

分析错误检查蓝屏数据

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

任务管理器实时内存转储

项目 • 2023/09/18

概述

任务管理器可用于创建实时内核内存转储。 这是使用任务管理器创建特定进程的内存转储的现有功能的补充。

实时内核内存转储包含 (内核内存的一致快照（可选）其他类型的内存)，并将其保存到转储文件。 与其他手动生成内核内存转储的方法不同，此方法不会导致系统崩溃。

有权访问相应符号文件和源代码的程序员可以分析转储文件，以检查系统状态并诊断问题。

创建实时内核内存转储的选项在 *系统* 进程上可用，类似于在用户模式进程中提供创建内存转储文件的选项。

*完整实时内核内存转储*包含活动内核内存，可选包含虚拟机监控程序内存和用户模式内存。 用于捕获虚拟机监控程序和用户页面的选项可用于完整的实时内核转储。 或者， *内核堆栈内存转储* 是一个较小的文件，仅限于内核处理器状态和所有内核线程堆栈。

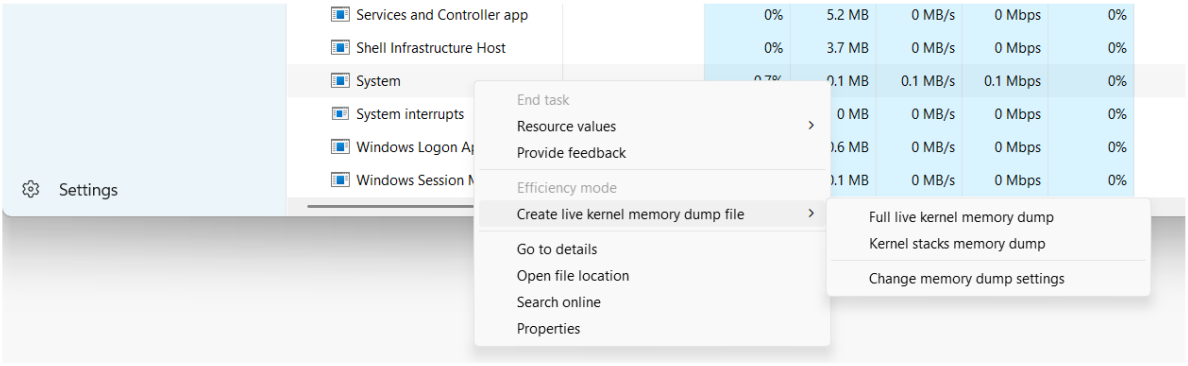
有关实时内核内存转储的一般信息，请参阅 内核实时内存转储代码参考。

任务管理器实时转储功能于 2023 年初发布，并首次在 Canary Channel (内部版本 25276 及更高版本中的 Windows Insider Preview 内部版本) 和 Dev Channel (内部版本 23419 及更高版本) ,2023 年 7 月在 Windows OS 内部版本 22621.1992 及更高版本中提供。

使用任务管理器创建系统的实时内核内存转储

若要使用任务管理器捕获实时内核内存转储，请完成以下步骤。

1. 则启动 Windows 任务管理器。
2. 导航到 “**进程** ”或 “**详细信息**”。
3. 找到 *系统* 进程。
4. 右键单击并选择“ **创建实时内核内存转储文件**”。
5. 从下拉菜单中，选择“ **完整实时内核内存转储** ”或“ **内核堆栈内存转储**”。

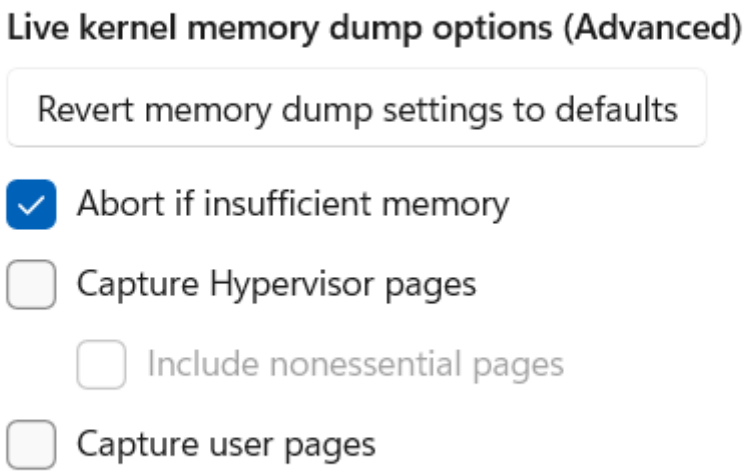


为用户模式进程创建内存转储

任务管理器中使用类似的过程来创建进程的内存转储。 突出显示所需的用户模式进程，然后右键单击并选择“ **创建内存转储文件**”。 有关用户模式转储文件的详细信息，请参阅用户模式转储文件。

实时内核内存转储文件高级选项

实时内核内存转储文件选项在任务管理器设置下可用。



顶部按钮会将实时内核内存转储设置还原为默认值。

选择“ *如果内存压力“选项被选择“中止* ”，如果认为内存可用性不足，实时转储进程将停止。 这是默认设置，可最大程度地减少捕获实时内核转储对系统响应能力的潜在影响。

内核实时内存转储文件设置为内存转储中要包含的信息提供了多个选项。

- *捕获虚拟机监控程序内存页*(带或不带不一切)
- *捕获用户页面*

向转储文件添加其他信息会增加其大小，并在记录内存转储时使用其他内存。

捕获虚拟机监控程序内存页

选择“*捕获虚拟机监控程序内存页*”选项，捕获虚拟机监控程序用于支持 Hyper-V 和虚拟机的内存区域。 有关详细信息，请参阅 Windows 上的 Hyper-V。

可以包含或不包括非任何不切的虚拟机监控程序内存页。

捕获用户页面

如果要排查的问题需要用户模式内存，请启用“*捕获用户页*”。

有关 Windows 内存和页面使用情况的一般信息，请参阅 由 Pavel Yosifovich、Alex Ionescu、Mark Russinovich 和 David Solomon 的 Windows Internals。

实时内存转储文件位置

内存转储完成后，将显示一个对话框，提供内存转储 `.dmp` 文件的位置。 单击“*打开文件位置*”打开文件夹。

实时内核内存转储

默认情况下，实时内核内存转储存储在此处。

`%LocalAppData%\Microsoft\Windows\TaskManager\LiveKernelDumps`

%LocalAppData% 通常为 `C:\Users\<YourUserName>\AppData\Local\`

实时用户模式内存转储

实时用户模式内存转储文件存储在 %localappdata%\Temp 目录中，该目录通常位于 Users 目录中。

`C:\Users\<YourUserName>\AppData\Local\Temp`

对实时内存转储进行故障排除

如果实时内存转储返回错误，检查错误消息了解详细信息。 例如：

- 任务管理器需要以管理员级别用户身份运行。
- 遇到超时问题时，请在几分钟后再次尝试转储。
- 等待任何请求的转储完成，然后再进行任何其他内存转储。
- 实时内核内存转储创建可能成功，但可能不包含内存的完整内容。 捕获暂时需要足够的可用物理内存来保存要写入转储文件的内存副本。 关闭不需要的应用程序或禁用 hyper-V 和用户模式内存页的捕获可能会增加可以保存到转储文件的内存量。

分析实时内存转储文件

发生实时内存转储时，可以使用用于其他内存转储文件的相同技术来分析转储文件。 若要了解故障期间内存的内容，通常需要了解处理器内存寄存器和程序集编程。 此外，访问失败的源代码将使开发人员能够修复该问题。

有关详细信息，请参阅：

- 使用 WinDbg 分析内核模式转储文件
- !analyze
- 内核实时内存转储代码参考

Bug 检查代码：0x161 - LIVE_SYSTEM_DUMP

任务管理器实时内存转储的 bug 检查代码是 bug 检查0x161：LIVE_SYSTEM_DUMP。

另请参阅

- Kernel-Mode 转储文件的品种。
- Bug 检查 0x161：LIVE_SYSTEM_DUMP
- 生成内核或完整的故障转储

Bug 检查 0xAB：SESSION_HAS_VALID_POOL_ON_EXIT

项目 • 2024/02/20

The SESSION_HAS_VALID_POOL_ON_EXIT 实时转储的值为 0x000000AB。 此 Bug 检查表明在会话驱动程序仍保留内存时发生了会话卸载。

（此代码永远不能用于真正的 Bug 检查；它用于标识实时转储。）

SESSION_HAS_VALID_POOL_ON_EXIT 参数

展开表

参数	说明
1	会话 ID。
2	正在泄漏的分页池字节数。
3	正在泄漏的非分页池字节数。
4	正在泄漏的分页分配和非分页分配的总数。（非分页分配的数量位于此单词的上半部分，分页分配的的数量位于此单词的下半部分。）

原因

因为会话驱动程序在会话卸载之前不会释放其池分配，所以发生 The SESSION_HAS_VALID_POOL_ON_EXIT 检查发生。 此 Bug 检查可能表明 Win32k.sys、Atmfd.dll、Rdpdd.dll 或视频或其他驱动程序中存在 Bug。

注解

!analyze 调试扩展显示有关 Bug 检查的信息，并非常有助于确定根本原因。

另请参阅

Bug 检查代码参考

错误检查 0x124：WHEA_UNCORRECTABLE_ERROR

项目 • 2024/10/16

WHEA_UNCORRECTABLE_ERROR 错误检查的值为 0x00000124，表明发生了致命的硬件错误。 此错误检查使用 Windows 硬件错误体系结构 (WHEA) 提供的错误数据。

若要识别错误的特定原因，通常需要了解 WHEA_ERROR_RECORD 结构。 有关详细信息，请参阅本文的备注部分。

> **重要**
>
> 本主题是面向程序员的。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

参数

展开表

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x0	WHEA_ERROR_RECORD 结构的地址	出现错误的 MCA 寄存器的 MCi_STATUS MSR 高 32 位	出现错误的 MCA 寄存器的 MCi_STATUS MSR 低 32 位	发生计算机检查异常。 如果处理器基于 x64 体系结构，或者具有 MCA 功能的 x86 体系结构（例如 Intel Processorum Pro、Pentum IV 或 Xeon），则这些参数说明适用。
0x1	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	发生经更正的计算机检查异常。
0x2	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	出现已更正的平台错误。
0x3	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	发生了不可屏蔽的中断 (NMI) 错误。
0x4	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	出现无法更正的 PCI Express 错误。

参数 1	参数 2	参数 3	参数 4	错误消息的原因
0x5	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	发生通用硬件错误。
0x6	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	出现初始化错误。
0x7	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	出现 BOOT 错误。
0x8	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	发生了可扩展的一致接口 (SCI) 一般错误。
0x9	WHEA_ERROR_RECORD 结构的地址	SAL 日志的长度（以字节为单位）	SAL 的地址	发生不可更正的基于 Itanium 的计算机检查中止错误。
0xA	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	发生已更正的基于 Itanium 的计算机检查错误。
0xB	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	已更正的 Itanium 平台错误。
0xC	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	其他类型的错误源 v2。
0xD	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	基于 SCI 的 GHESv2（ACPI 通用硬件错误源）。
0xE	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	BMC（主板管理控制器）错误信息。
0xF	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	ARS PMEM（地址范围清理持性内存）错误源。
0x10	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	设备驱动程序错误源。
0x11	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	Arm 同步外部中止。
0x12	WHEA_ERROR_RECORD 结构的地址	Reserved	保留	Arm SError 中断。

原因

此错误检查通常与物理硬件故障相关。 它可能与热量有关，也可能是硬件和内存缺陷，甚至是开始发生故障或已经发生故障的处理器造成的。 如果已启用过时钟，请尝试禁用它。 确认所有冷却系统（如风扇）都正常运行。 运行系统诊断以确认系统内存没有故障。 驱动程序导致硬件无法通过错误检查的可能性较小，但也有可能。

若要了解常规错误检查故障排除的更多信息，请参阅分析错误检查蓝屏数据。

注解

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

- 参数 1 标识报告错误的错误源的类型。
- 参数 2 保存描述错误条件的 WHEA_ERROR_RECORD 结构的地址。

发生硬件错误时，WHEA 会创建错误记录来存储与硬件错误条件相关的错误信息。 每个错误记录都由 WHEA_ERROR_RECORD 结构描述。 Windows 内核将错误记录包含在它为响应错误而引发的 Windows 事件跟踪 (ETW) 硬件错误事件中，以便错误记录保存在系统事件日志中。 WHEA 使用的错误记录格式基于统一可扩展固件接口 (UEFI) 规范版本 2.2 附录 N 中所述通用平台错误记录。 有关详细信息，请参阅 WHEA_ERROR_RECORD 和 Windows 硬件错误体系结构 (WHEA)。

可以使用 !errrec address 通过参数 2 中提供的地址显示 WHEA_ERROR_RECORD 结构。 !whea 和 !errpkt 扩展可用于显示其他 WHEA 信息。

有关详细信息，请参阅以下文章：

使用 Windows 调试器 (WinDbg) 进行故障转储分析

使用 WinDbg 分析内核模式转储文件

使用 !analyze 扩展 和 !analyze

Windows Vista 之前的 Windows 版本不支持此错误检查。 而是通过错误检查 0x9C 报告计算机检查异常。

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

bug 检查 0x141: VIDEO_ENGINE_TIMEOUT_DETECTED

项目 • 2024/02/21

VIDEO_ENGINE_TIMEOUT_DETECTED 实时转储的值为 0x00000141。 这表示其中一个显示引擎无法及时响应。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

VIDEO_ENGINE_TIMEOUT_DETECTED 参数

展开表

参数	说明
1	指向内部 TDR 恢复上下文 (TDR_RECOVERY_CONTEXT) 的可选指针。
2	指向负责任的设备驱动程序模块（例如所有者标记）的指针。
3	特定于辅助驱动程序的存储桶密钥。
4	可选的内部上下文相关数据。

注解

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

标记 {270A33FD-3DA6-460D-BA89-3C1BAE21E39B} 的辅助数据包含其他 TDR 相关数据。 使用 .enumtag 查看数据。

bug 检查 0x142：VIDEO_TDR_APPLICATION_BLOCKED

项目 • 2024/02/23

VIDEO_TDR_APPLICATION_BLOCKED 实时转储的值为 0x00000142。 这表示应用程序已被阻止访问图形硬件。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

VIDEO_TDR_APPLICATION_BLOCKED 参数

展开表

参数	说明
1	指向内部 TDR 恢复上下文 (TDR_RECOVERY_CONTEXT) 的可选指针。
2	指向负责任的设备驱动程序模块（例如所有者标记）的指针。
3	特定于辅助驱动程序的存储桶密钥。
4	被阻止访问 GPU 的进程的 ID。

注解

!analyze 调试扩展显示有关 bug 检查的信息，并非常有助于确定根本原因。

标记 {270A33FD-3DA6-460D-BA89-3C1BAE21E39B} 的辅助数据包含其他 TDR 相关数据。 使用 **.enumtag（枚举辅助回调数据）**查看数据。

Bug 检查 0x144：BUGCODE_USB3_DRIVER

项目 • 2024/02/23

BUGCODE_USB3_DRIVER bug 检查的值为 0x00000144。 这是用于所有 USB 3 bug 检查的代码。 参数 1 指定 USB 3 bug 检查的类型，其他参数的含义取决于参数 1。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

BUGCODE_USB3_DRIVER 参数

展开表

参数 1	参数 2	参数 3	参数 4	错误原因
0x1	可选。 指向用于重新发送 URB 的 IRP 的指针	指向 URB 的指针	指向客户端驱动程序的设备对象的指针	客户端驱动程序使用了以前发送到核心堆栈的 URB。
0x2	指向启动设备的物理设备对象 (PDO) 的指针	预留	预留	启动或分页设备重新枚举失败。
0x3	可选。 指向用于发送 URB 的 IRP 的指针	指向损坏的 URB 的指针	指向客户端驱动程序的设备对象的指针	客户端驱动程序将损坏的 URB 发送到核心堆栈。 之所以发生这种情况，是因为客户端驱动程序未使用 **USBD_*xxx*UrbAllocate** 分配 URB，或者客户端驱动程序为 URB 执行了缓冲区不足。
0x800	发送 Open Static Streams 请求的 IRQL	指向 Open Static Streams IRP 的指针	指向客户端驱动程序的设备	在 IRQL > PASSIVE LEVEL 发送了 Open Static Streams 请求。

参数 1	参数 2	参数 3	参数 4	错误原因
			对象的指针	
0x801	指向 Open Static Streams IRP 的指针	指向 Open Static Streams URB 的指针	指向客户端驱动程序的设备对象的指针	客户端驱动程序在查询流功能之前尝试打开静态流。 在客户端驱动程序成功查询流功能之前，客户端驱动程序无法打开静态流。 有关详细信息，请参阅“备注”。
0x802	客户端驱动程序尝试打开的静态流数	向客户端驱动程序授予的静态流数	指向客户端驱动程序的设备对象的指针	客户端驱动程序尝试打开无效数量的静态流。 流数不能为 0，并且不能大于在查询 USB 功能调用中返回到客户端驱动程序的值。
0x803	指向 Open Static Streams IRP 的指针	指向 Open Static Streams URB 的指针	指向客户端驱动程序的设备对象的指针	客户端驱动程序尝试为已打开静态流的终结点打开静态流。 在打开静态流之前，客户端驱动程序必须关闭以前打开的静态流。
0x804	泄漏的句柄上下文。 运行 **!usbanalyze -v** 以获取有关泄漏的句柄和 URB 的信息。 必须为客户端驱动程序启用驱动程序验证程序。	传递给 USBD_CreateHandle 的设备对象。	Reserved	客户端驱动程序忘记关闭之前使用 USBD_CreateHandle 创建的句柄，或忘记释放分配的 URB。
0x805	Close Static Streams URB 的 WDFREQUEST 句柄	指向 Close Static Streams URB 的指针	指向客户端驱动程序的设备对象的指针	客户端驱动程序发送了处于无效状态的 Close Static Streams URB（例如，处理 D0 Exit 后）。
0x806	指向 IRP 的指针	指向 URB 的指针	指向客户端驱动程序的设备对象的指针	客户端驱动程序在查询链接的 MDL 功能之前尝试发送链接的 MDL。 客户端驱动程序在成功查询链接的 MDL 功能之前，无法发送链接的 MDL。 有

参数 1	参数 2	参数 3	参数 4	错误原因
				关详细信息，请参阅“备注”。
0x807	指向链接的 MDL 的 指针	指向 URB 的指针	指向客户端驱动程序的设备对象的指针（如果可用）	客户端驱动程序将 URB 发送到核心堆栈，其传输缓冲区长度超过传入的 MDL 的字节计数（由 MmGetMdlByteCount 返回）。 有关详细信息，请参阅“备注”。
0x1001	XHCI_LIVEDUMP_CONTEXT	预留	预留	xHCI 控制器断言 HSE 位，指示主机系统错误。
0x1002	XHCI_LIVEDUMP_CONTEXT	预留	预留	xHCI 控制器断言 HCE 位，指示主机控制器错误。
0x1003	XHCI_LIVEDUMP_CONTEXT	预留	预留	xHCI 停止终结点命令返回未经处理的完成代码。
0x1004	XHCI_LIVEDUMP_CONTEXT	预留	预留	xHCI 终结点状态在发出 xHCI 终结点停止命令后收到上下文状态错误。
0x1005	XHCI_LIVEDUMP_CONTEXT	预留	预留	尝试清除控制终结点上的停止期间，设置取消排队指针失败。
0x1006	XHCI_LIVEDUMP_CONTEXT	预留	预留	在尝试清除控制终结点上的停止期间重置 EP 失败。
0x1007	XHCI_LIVEDUMP_CONTEXT	预留	预留	重置恢复期间 xHCI 控制器重置失败。
0x1008	XHCI_LIVEDUMP_CONTEXT	预留	预留	重置恢复期间 xHCI 控制器重启失败。
0x1009	XHCI_LIVEDUMP_CONTEXT	预留	预留	命令超时中止后，xHCI 控制器命令无法完成。
0x100A	XHCI_LIVEDUMP_CONTEXT	预留	预留	在尝试在终结点停止完成后设置取消排队

参数 1	参数 2	参数 3	参数 4	错误原因
				指针时，设置取消排队指针失败。
0x100B	XHCI_LIVEDUMP_CONTEXT	预留	预留	重置恢复期间 xHCI 控制器停止失败。
0x100C	XHCI_LIVEDUMP_CONTEXT	预留	预留	不支持 xHCI 控制器中的固件。 除非更新固件，否则不会在此控制器上加载 xHCI 驱动程序。
0x100D	XHCI_LIVEDUMP_CONTEXT	预留	预留	检测到控制器在物理上被删除。
0x100E	XHCI_LIVEDUMP_CONTEXT	预留	预留	驱动程序在启用了流的终结点上检测到错误。
0x100F	XHCI_LIVEDUMP_CONTEXT	预留	预留	xHCI 控制器中的固件已过时。 xHCI 驱动程序将继续使用此控制器，但可能会遇到一些问题。 建议使用固件更新。
0x1010	XHCI_LIVEDUMP_CONTEXT	预留	预留	传输事件 TRB 使用未经处理的完成代码完成。
0x1011	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器报告事件环已满。 当发生这种情况时，控制器也已知会删除事件。
0x1012	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器已按顺序完成命令。
0x1013	XHCI_LIVEDUMP_CONTEXT	预留	预留	命令中止完成后，控制器报告的命令环取消排队指针不正确。
0x1014	XHCI_LIVEDUMP_CONTEXT	预留	预留	启用插槽完成后，控制器会提供一个错误的插槽 ID。
0x1015	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器使用 BSR1 执行 SetAddress 命令失败。 这是意外行为。

参数 1	参数 2	参数 3	参数 4	错误原因
0x1016	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器在 usbdevice 重置期间未能启用插槽。 这是意外行为。
0x1017	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器执行终结点配置命令失败，我们在其中取消配置终结点。 这是意外行为。
0x1018	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器禁用插槽命令失败。 这是意外行为。
0x1019	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器执行 USB 设备重置命令失败。 这是意外行为。
0x101A	XHCI_LIVEDUMP_CONTEXT	预留	预留	终结点重置后，“设置取消排队指针”命令失败。
0x101B	XHCI_LIVEDUMP_CONTEXT	预留	预留	xHCI 重置终结点命令返回未经处理的完成代码。
0x101C	XHCI_LIVEDUMP_CONTEXT	预留	预留	xHCI 的 D0Entry 失败。
0x101D	XHCI_LIVEDUMP_CONTEXT	预留	预留	在请求取消期间使用“配置终结点”命令而不是“设置取消排队指针”时，暂时删除和添加流终结点（作为两个命令）失败。
0x101E	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器指示未在控制器上挂起的传输完成。 EventData == 1（取消引用传输事件 TRB 的指针会导致 bug 检查）
0x101F	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器指示未在控制器上挂起的传输完成。 EventData == 0（传输事件 TRB 中的逻辑地址不匹配）

参数 1	参数 2	参数 3	参数 4	错误原因
0x1020	XHCI_LIVEDUMP_CONTEXT	预留	预留	控制器指示未在控制器上挂起的传输完成。 EventData == 0（传输事件 TRB 中的逻辑地址不匹配）传输事件 TRB 可能是冗余的（指向最近完成的请求附近的某个位置）。
0x1021	XHCI_LIVEDUMP_CONTEXT	预留	预留	使用“配置终结点”命令重置未停止的终结点时，暂时删除和添加流终结点（作为两个命令）失败。
0x1022	XHCI_LIVEDUMP_CONTEXT	预留	预留	删除并添加同一终结点（作为一个命令）失败。
0x3000	USBHUB3_LIVEDUMP_CONTEXT	预留	预留	中心驱动程序成功重置了异常的中心。
0x3001	USBHUB3_LIVEDUMP_CONTEXT	预留	预留	中心驱动程序未能成功重置异常的中心。
0x3002	USBHUB3_LIVEDUMP_CONTEXT	预留	预留	中心驱动程序禁用了非函数 SuperSpeed 中心。
0x3003	USBHUB3_LIVEDUMP_CONTEXT	预留	预留	USB 设备枚举失败。

注解

若要查询 USB 功能，客户端驱动程序必须调用 WdfUsbTargetDeviceQueryUsbCapability 或 USBD_QueryUsbCapability

若要发送链接的 MDL，客户端驱动程序必须调用 USBD_QueryUsbCapability，并使用 URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER_USING_CHAINED_MDL 或 URB_FUNCTION_ISOCH_TRANSFER_USING_CHAINED_MDL。

另请参阅

通用串行总线 (USB)

内核实时转储代码引用

Bug 检查代码参考

bug 检查 0x156：WINSOCK_DETECTED_HUNG_CLOSESOCKET_LIVEDUMP

项目 • 2024/02/21

WINSOCK_DETECTED_HUNG_CLOSESOCKET_LIVEDUMP 实时转储的值为 0x00000156。 这表示 Winsock 检测到挂起的传输终结点关闭请求。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

WINSOCK_DETECTED_HUNG_CLOSESOCKET_LIVEDUMP 参数

展开表

参数	说明
1	AFD 终结点指针 (!afdkd.endp <ptr>) !afdkd 调试器扩展仅适用于硬件开发合作伙伴。
2	传输终结点类型 0x1：UDP 数据报 0x2：RAW 数据报 0x3：TCP 侦听器 0x4：TCP 终结点
3	数据报终结点的缓冲发送字节数
4	afd!NETIO_SUPER_TRIAGE_BLOCK

原因

处理 closesocket 请求时，Winsock 检测到挂起的传输终结点关闭请求。 系统生成了实时转储进行分析，然后 closesocket 请求已完成，无需等待完成挂起的传输终结点关闭请求。

bug 检查 0x15C：PDC_WATCHDOG_TIMEOUT_LIVEDUMP

项目 • 2024/02/23

PDC_WATCHDOG_TIMEOUT_LIVEDUMP 实时转储的值为 0x0000015C。 这表示系统组件无法在分配的时间段内响应，从而阻止系统进入或退出已连接待机状态。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

PDC_WATCHDOG_TIMEOUT_LIVEDUMP 参数

展开表

参数	说明
1	已挂起组件的客户端 ID。
2	已挂起组件的客户端类型。 0x1：通知客户端无法响应。 3 - 指向通知客户端 (pdc!_PDC_NOTIFICATION_CLIENT) 的指针。 4 - 指向 pdc!PDC_14F_TRIAGE 结构的指针。 0x2：复原客户端无法响应。 3 - 指向复原客户端 (pdc!_PDC_RESILIENCY_CLIENT) 的指针。 4 - 指向 pdc!PDC_14F_TRIAGE 结构的指针。 0x3：激活器客户端保留引用时间过长 3 - 指向激活客户端 (pdc!_PDC_ACTIVATOR_CLIENT) 的指针。 4 - 指向 pdc!PDC_14F_TRIAGE 结构的指针。
3	请参阅参数 2
4	请参阅参数 2

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x15D：SOC_SUBSYSTEM_FAILURE_LIVEDUMP

项目 • 2024/10/16

SOC_SUBSYSTEM_FAILURE_LIVEDUMP 实时转储的值为 0x0000015D。 这表示芯片上的系统 (SoC) 子系统遇到严重故障并捕获了实时内核转储。 在这种情况下，SoC 子系统不会生成错误检查。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

错误检查 0x14B SOC_SUBSYSTEM_FAILURE 参数

展开表

参数	说明
1	SOC_SUBSYSTEM_FAILURE_DETAILS 结构的地址。
2	保留。
3	保留。
4	可选。 供应商提供的数据块的地址。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

使用提供的 nt!SOC_SUBSYSTEM_FAILURE_DETAILS 结构，使用 dt 命令和 Arg1 提供的地址转储故障数据。

dbgcmd

```
2: kd> dt nt!SOC_SUBSYSTEM_FAILURE_DETAILS 9aa8d630
   +0x000 SubsysType       : 1 ( SOC_SUBSYS_AUDIO_DSP )
   +0x008 FirmwareVersion  : 0
   +0x010 HardwareVersion  : 0
   +0x018 UnifiedFailureRegionSize : 0x24
   +0x01c UnifiedFailureRegion : [1]  "F"
```

与 SoC 供应商合作进一步分析数据，包括提供常规用途数据块的可选供应商。

你可能希望使用 **k、kb、kc、kd、kp、kP、kv（显示堆栈回溯）**命令检查堆栈跟踪。 可以指定处理器编号以检查所有处理器上的堆栈。

您还可以在导致此停止代码的代码中设置断点，并尝试单步执行故障代码。

有关详细信息，请参阅以下主题：

使用 Windows 调试器 (WinDbg) 进行故障转储分析

如果您不具备使用 Windows 调试器来解决此问题的能力，则可以使用一些基本的故障排除技术。

- 在事件查看器中检查系统日志以获取其他错误消息，这可能有助于识别导致此错误检查的设备或驱动程序。
- 如果在错误检查消息中标识了驱动程序，请禁用该驱动程序，或与制造商联系以获取驱动程序更新。
- 你可尝试运行系统制造商提供的硬件诊断。
- 有关其他常规故障排除信息，请参阅**分析错误检查蓝屏数据**。

要求

展开表

最低受支持的客户端	Windows 8
支持的最低服务器版本	Windows Server 2012

另请参阅

内核实时转储代码引用

错误检查代码参考

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

bug 检查 0x15E：BUGCODE_NDIS_DRIVER_LIVE_DUMP

项目 • 2024/02/21

BUGCODE_NDIS_DRIVER_LIVE_DUMP 实时转储的值为 0x0000015E。 此 bug 代码指示 NDIS 已捕获实时内核转储。 在这种情况下，NDIS 不会生成 bug 检查。

BUGCODE_NDIS_DRIVER 参数

参数 1 表示违规类型。 其他参数的含义取决于参数 1 的值。 如果参数的值为“0”，则表示未使用此参数。

展开表

参数 1	参数 1 值和错误原因	参数 2	参数 3
0x01	NDIS_BUGCHECK_MINIPORT_FATAL_ERROR 微型端口驱动程序遇到致命错误并请求重新枚举。	微型端口块的地址。 请使用此地址运行 !ndiskd.minidriver 获取详细信息。	微型端口的物理设备对象 (PDO) 的地址
0x25	NDIS_BUGCHECK_WATCHDOG 管理网络堆栈的尝试耗时过长。 当 NDIS 调用其他驱动程序时，NDIS 会启动监视器计时器，以确保调用及时完成。 如果调用时间过长，NDIS 会注入 Bug 检查。 这可能是由简单的死锁造成的。 使用“!stacks 2 ndis”或类似命令查看是否有可疑线程。 请特别注意 NDIS_WATCHDOG_TRIAGE_BLOCK中的 PrimaryThread。 这可能是 NBL 丢失造成的，在这种情况下 !ndiskd.pendingnbls 可能会有所帮助。 检查使用 !ndiskd.oid 卡住的 OID。	操作耗时过长。 可能的值： • 0x01 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_PAUSE 暂停协议驱动程序时出现超时。 • 0x02 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_NETPNPEVENT 向协议驱动程序传送 NET_PNP_EVENT_NOTIFICATION 时出现超时。 • 0x03 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_STATUS_INDICATION 向协议驱动程序传送状态指示时出现超时。 • 0x04 : NDIS_BUGCHECK_WATCHDOG_PROTOCOL_UNBIND 取消绑定协议驱动程序时出现超时。 • 0x11 : NDIS_BUGCHECK_WATCHDOG_FILTER_PAUSE 暂停筛选器驱动程序时出现超时。 • 0x12 : NDIS_BUGCHECK_WATCHDOG_FILTER_NETPNPEVENT 向筛选器驱动程序传送 NET_PNP_EVENT_NOTIFICATION 时出现超时。 • 0x13 : NDIS_BUGCHECK_WATCHDOG_FILTER_STATUS_INDICATION 向筛选器驱动程序传送状态指示时出现超时。 • 0x14 : NDIS_BUGCHECK_WATCHDOG_FILTER_DETACH 拆离筛选器驱动程序时出现超时。 • 0x21 : NDIS_BUGCHECK_WATCHDOG_MINIPORT_PAUSE 暂停微型端口适配器时出现超时。 • 0x22 : NDIS_BUGCHECK_WATCHDOG_MINIPORT_HALT 停止微型端口适配器时出现超时。 • 0x23 : NDIS_BUGCHECK_WATCHDOG_MINIPORT_OID 向微型端口适配器传送 OID 请求时超时。 • 0x24 : NDIS_BUGCHECK_WATCHDOG_FILTER_OID 向筛选器驱动程序传送 OID 请求时出现超时。	强制转换为 ndis!NDIS_WATCHDOG_TRIAGE_BLOCK。有用的字段： • **StartTime** 显示 KeQueryInterruptTime 返回的操作启动时间，以 100ns 为单位。 • **TimeoutMilliseconds** 显示在触发此 bugcheck 之前，NDIS 至少等待了多长时间。 • **TargetObject** 是 NDIS 正在等待的协议、筛选器模块或微型端口适配器的句柄。 请使用此句柄运行 !ndiskd.protocol、!ndiskd.filter 或 !ndiskd.netadapter，获取更多信息。 • **PrimaryThread** 是 NDIS 启动操作的线程。 这通常是第一个查找位置，尽管如果操作是异步处理的，线程可能已去往其他位置。

参数 1	参数 1 值和错误原因	参数 2	参数 3
		• 0x25 : NDIS_BUGCHECK_WATCHDOG_MINIPORT_IDLE 闲置微型端口适配器时出现超时。 • 0x26 : NDIS_BUGCHECK_WATCHDOG_CANCEL_IDLE 取消微型端口适配器上的空闲请求时出现超时。	
0x30	NDIS_BUGCHECK_STUCK_NBL 微型端口驱动程序一段时间没有将 NBL 返回回堆栈。	微型端口块的地址。 请使用此地址运行 !ndiskd.minidriver 获取详细信息。	0

原因

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。 参数 1 表示 BUGCODE_NDIS_DRIVER_LIVE_DUMP bug 检查的具体原因。

注解

NDIS 已检测到另一个网络驱动程序中的严重问题并从中恢复。 尽管系统未停止，但此问题以后可能会导致连接问题或致命 bug 检查。

此 bug 代码仅在 Windows 8.1 及更高版本的 Windows 中发生。

bug 检查 0x15F：CONNECTED_STANDBY_WATCHDOG_TIMEOUT_LIVEDUMP

项目 • 2024/02/23

CONNECTED_STANDBY_WATCHDOG_TIMEOUT_LIVEDUMP 实时转储的值为 0x0000015F。 这表示已连接待机监视器超时。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

CONNECTED_STANDBY_WATCHDOG_TIMEOUT_LIVEDUMP 参数

展开表

参数	说明
1	CS 监视器子代码 0x1：DRIPS 监视器超时。 系统处于已连接待机的复原阶段，没有激活器处于活动状态，并且没有设备约束未得到满足太长时间且没有进入 DRIPS（最深层运行时空闲平台状态）。 2 - 指向其他信息的指针 (nt!POP_DRIPS_WATCHDOG_METRICS) 3 - 非 DRIPS 持续时间（以毫秒为单位） 4 - 保留 0x2：DRIPS 监视器设备约束超时。 系统处于已连接待机的复原阶段，并且没有进入 DRIPS（最深层运行时空闲平台状态），原因是没有激活器处于活动状态且某个设备约束未得到满足。 2 - nt!TRIAGE_POP_FX_DEVICE 设备 3 - 组件索引 4 - 保留 0x3：DRIPS 监视程序预否决超时。 由于活动 PEP 预否决、没有设备约束未得到满足且没有激活器处于活动状态，系统长时间处于已连接待机的复原阶段，并且没有进入 DRIPS（最深层运行时空闲平台状态）。 2 - 否决代码 3 - 指向否决名称字符串 (PWSTR) 的指针 4 - 指向 PEP PPM 回调的指针 0x4：深度睡眠监视器 2- 指标

参数	说明
	3 -NonDeepSleepDurationMs 4 - 保留 0x5：深度睡眠电源设置监视器 2- 指标 3 -NonDeepSleepDurationMs 4 - 保留
2	请参阅参数 1
3	请参阅参数 1
4	请参阅参数 1

原因

此计算机正在表现出可降低屏幕关闭电池使用时间的行为。 这通常是由 CPU 活动、设备活动或处于空闲状态不足的设备引起的。

另请参阅

内核实时转储代码引用

Bug 检查代码参考

bug 检查 0x161：LIVE_SYSTEM_DUMP

项目 • 2024/02/23

LIVE_SYSTEM_DUMP 实时转储的值为 0x00000161。 这表示系统管理员请求收集实时系统内存转储。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

LIVE_SYSTEM_DUMP 参数

展开表

参数	说明
1	0x005461736b6d6772，这是文本字符串“Taskmgr”的十六进制编码。
2	Reserved
3	Reserved
4	Reserved

注解

从 Windows 22H2 版本 25276 开始，任务管理器可用于创建实时系统内存转储。 有关详细信息，请参阅任务管理器实时内存转储。

另请参阅

任务管理器实时内存转储

内核实时转储代码引用

bug 检查 0x164：WIN32K_CRITICAL_FAILURE

项目 • 2024/02/23

WIN32K_CRITICAL_FAILURE bug 检查的值为 0x00000164。 这表示 Win32k 遇到严重故障。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

WIN32K_CRITICAL_FAILURE 参数

展开表

参数	说明
1	1 - 故障的类型。 0x1：REGION_VALIDATION_FAILURE - 区域超出表面边界。 2 - 指向 DC 的指针 3 - 指向 SURFACE 的指针 4 - 指向 REGION 的指针 0x2：OPERATOR_NEW_USED - 运算符“new”用于分配内存。 2 - 保留 3 - 保留 4 - 保留 0x3：CRITICAL_APISET_EXTENSIONS_MISSING - 关键扩展 APISET API 缺失。 2 - wchar_t* 到缺失函数的名称 3 - 保留 4 - 保留

参数	说明
	0x4：GDI_SPRITE_SURFACE_INVALID_DELETE - 正在删除 GDI 子画面的形状，而不删除子画面。 2 - SURFACE 的句柄 3 - 对 SURFACE 的引用计数 4 - SURFACE 所有者的 PID 0x5： POINTER_DEVICE_EXCLUSIVE_OPEN_FAILED - 无法打开指针设备。 2 - 设备的 UNICODE_STRING 3 - 保留 4 - 保留 0x8：PUBLIC_DC_INVALID_PRIVATE_MEMBER - 公共 DC 具有指向特定进程拥有的对象的指针。 2 - 指向 DC 的指针 3 - 拥有对象的进程 ID 4 - 保留 0xA：TTFD_INVOKE_ILLEGAL_ID - TTFD 中使用的函数表索引无效。 2 - 保留 3 - 保留 4 - 保留 0xB：OTFD_INVOKE_ILLEGAL_ID - ATMFD 中使用的函数表索引无效。 2 - 保留 3 - 保留 4 - 保留 0xC：GFPE_INVOKE_ILLEGAL_ID - PALETTE 中使用的函数表索引无效。 2 - 指向 PALETTE 的指针 3 - 索引无效 4 - 最大有效索引 + 1 0x10：USER_SAS_REGISTRATION_FAILED - SAS 密钥注册失败。

参数	说明
	2 - vkey 3- 修饰符 4 - 标志
2	请参阅参数 1
3	请参阅参数 1
4	请参阅参数 1

解决方法

!analyze 调试扩展显示有关 bug 检查的信息，并有助于确定根本原因。

bug 检查 0x165：CLUSTER_CSV_STATUS_IO_TIMEOUT_LIVEDUMP

项目 • 2024/02/23

CLUSTER_CSV_STATUS_IO_TIMEOUT_LIVEDUMP 实时转储的值为 0x00000165。 这表示非协调节点上的 SMB 客户端抱怨协调节点上的 IO 花费的时间过长，并且所有具有 STATUS_IO_TIMEOUT 的 IO 都失败。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

CLUSTER_CSV_STATUS_IO_TIMEOUT_LIVEDUMP 参数

展开表

参数	说明
1	可选群集服务 PID。
2	观察到 STATUS_IO_TIMEOUT 的节点的群集节点 ID。
3	保留。
4	保留。

原因

非协调节点上的 SMB 客户端抱怨协调节点上的 IO 花费的时间过长，并且所有具有 STATUS_IO_TIMEOUT 的 IO 都失败。

转储的辅助数据流中提供了其他信息。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储，包括群集共享卷遥测。）

解决方法

另请参阅

使用实时转储排查挂起问题（博客）

内核实时转储代码引用

Bug 检查代码参考

bug 检查 0x166：CLUSTER_RESOURCE_CALL_TIMEOUT_LIVEDUMP

项目 • 2024/02/21

CLUSTER_RESOURCE_CALL_TIMEOUT_LIVEDUMP 实时转储的值为 0x00000166。 这表示群集资源调用花费的时间超过配置的超时时间。

CLUSTER_RESOURCE_CALL_TIMEOUT_LIVEDUMP 参数

展开表

参数	说明
1	资源主机监视器 PID。
2	处理资源调用的线程的 TID。
3	资源调用类型 - 下面列出。
4	子代码。 参数 3 等于 8 时，此参数包含群集资源控制代码。 参数 3 等于 9 时，此参数包含群集资源类型控制代码。

资源调用类型

1 OPEN

2 CLOSE

3 ONLINE

4 OFFLINE

5 TERMINATE

6 ARBITRATE

7 RELEASE

8 RESOURCE CONTROL

9 RESOURCE TYPE CONTROL

10 LOOKS ALIVE

11 IS ALIVE

12 FAILURE NOTIFICATION

13 SHUTDOWN PROCESS

14 CANCEL

原因

群集资源调用花费的时间比配置的超时时间长。 系统生成了实时转储，用于分析延迟。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

解决方法

另请参阅

使用实时转储排查挂起问题（博客）

内核实时转储代码引用

Bug 检查代码参考

bug 检查 0x167：CLUSTER_CSV_SNAPSHOT_DEVICE_INFO_TIMEOUT_LIVEDUMP

项目 • 2024/02/23

CLUSTER_CSV_SNAPSHOT_DEVICE_INFO_TIMEOUT_LIVEDUMP 实时转储的值为 0x00000167。 这表示对 volsnap 的群集服务调用查询快照信息花费的时间过长。

CLUSTER_CSV_SNAPSHOT_DEVICE_INFO_TIMEOUT_LIVEDUMP 参数

展开表

参数	说明
1	群集服务 PID。
2	处理 volsnap 查询的线程的 TID。
3	如果在 CSV 卷处于活动状态时超时，此参数的值为 1；如果即使 CSV 卷关闭也会超时，则为 2。
4	保留。

原因

对 volsnap 的群集服务调用查询快照信息花费的时间过长。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

解决方法

另请参阅

使用实时转储排查挂起问题（博客）

内核实时转储代码引用

Bug 检查代码参考

bug 检查 0x168：CLUSTER_CSV_STATE_TRANSITION_TIMEOUT_LIVEDUMP

项目 • 2024/02/21

CLUSTER_CSV_STATE_TRANSITION_TIMEOUT_LIVEDUMP 实时转储的值为 0x00000168。 这表示群集共享卷状态转换花费的时间太长。

CLUSTER_CSV_STATE_TRANSITION_TIMEOUT_LIVEDUMP 参数

展开表

参数	说明
1	群集服务 PID。
2	CSV 目标状态 ID - 下面列出。
3	Reserved
4	Reserved

CSV 目标状态 ID

0 正在等待卷转换为 Init 状态。

1 正在等待卷转换为 Paused 状态。

2 正在等待卷转换为 Draining 状态。

3 正在等待卷转换为 Set-Down-Level 状态。

4 正在等待卷转换为 Active 状态。

原因

群集共享卷状态转换花费的时间太长。 系统生成了实时转储，用于分析延迟。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

解决方法

另请参阅

使用实时转储排查挂起问题（博客）

Bug 检查代码参考

bug 检查 0x169：CLUSTER_CSV_VOLUME_ARRIVAL_LIVEDUMP

项目 • 2024/02/21

CLUSTER_CSV_VOLUME_ARRIVAL_LIVEDUMP 实时转储的值为 0x00000169。 这表明群集共享卷管理器被要求创建新的卷设备对象，并且卷尚未及时到达。

CLUSTER_CSV_VOLUME_ARRIVAL_LIVEDUMP 参数

展开表

参数	说明
1	群集服务 PID。
2	保留。
3	保留。
4	保留。

原因

群集共享卷管理器被要求创建新的卷设备对象，并且卷尚未及时到达。

系统生成了实时转储，用于分析延迟。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

解决方法

另请参阅

使用实时转储排查挂起问题（博客）

内核实时转储代码引用

Bug 检查代码参考

bug 检查 0x16A：CLUSTER_CSV_VOLUME_REMOVAL_LIVEDUMP

项目 • 2024/02/23

CLUSTER_CSV_VOLUME_REMOVAL_LIVEDUMP 实时转储的值为 0x0000016A。 这表示群集共享卷管理器卷删除请求已超时。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

CLUSTER_CSV_VOLUME_REMOVAL_LIVEDUMP 参数

展开表

参数	说明
1	群集服务 PID
2	保留。
3	保留。
4	保留。

原因

群集共享卷管理器卷删除请求已超时。

系统生成了实时转储，用于分析延迟。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

解决方法

另请参阅

使用实时转储排查挂起问题（博客）

内核实时转储代码引用

Bug 检查代码参考

bug 检查 0x16B：CLUSTER_CSV_CLUSTER_WATCHDOG_LIVEDUMP

项目 • 2024/02/23

CLUSTER_CSV_CLUSTER_WATCHDOG_LIVEDUMP 实时转储的值为 0x0000016B。 这表示群集服务用户模式监视器检测到线程在很长一段时间内未向前推进。

CLUSTER_CSV_CLUSTER_WATCHDOG_LIVEDUMP 参数

展开表

参数	说明
1	群集服务 PID。
2	停滞的线程的 ID。
3	保留。
4	保留。

原因

群集服务用户模式监视器检测到线程在很长一段时间内未向前推进。

系统生成了实时转储，用于分析延迟。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

解决方法

另请参阅

使用实时转储排查挂起问题（博客）

内核实时转储代码引用

Bug 检查代码参考

bug 检查 0x16F：CLUSTER_CSV_STATE_TRANSITION_INTERVAL_TIMEOUT_LIVEDUMP

项目 • 2024/02/21

CLUSTER_CSV_STATE_TRANSITION_INTERVAL_TIMEOUT_LIVEDUMP 实时转储的值为 0x0000016F。 这表示群集共享卷的下一状态转换请求尚未到达。

CLUSTER_CSV_STATE_TRANSITION_INTERVAL_TIMEOUT_LIVEDUMP 参数

展开表

参数	说明
1	群集服务 PID。
2	CSV 目标状态 ID - 下面列出。
3	Reserved
4	Reserved

CSV 目标状态 ID

0 正在等待卷转换为 Init 状态。

1 正在等待卷转换为 Paused 状态。

2 正在等待卷转换为 Draining 状态。

3 正在等待卷转换为 Set-Down-Level 状态。

4 正在等待卷转换为 Active 状态。

原因

群集共享卷下一个状态转换请求尚未到达。

系统生成了实时转储，用于分析延迟。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

解决方法

另请参阅

使用实时转储排查挂起问题（博客）

内核实时转储代码引用

Bug 检查代码参考

bug 检查 0x175：PREVIOUS_FATAL_ABNORMAL_RESET_ERROR

项目 • 2024/02/23

PREVIOUS_FATAL_ABNORMAL_RESET_ERROR 实时转储的值为 0x00000175。 这表示发生了不可恢复的系统错误，或者系统在 Windows 手机设备上异常重置。 系统生成了实时转储，用于从上一错误收集设备崩溃数据。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

> **重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

PREVIOUS_FATAL_ABNORMAL_RESET_ERROR 参数

展开表

参数	说明
1	保留
2	Reserved
3	Reserved
4	Reserved

原因

Windows 手机设备上的系统遇到意外错误并重启。 可能导致此错误的问题包括：应用程序或辅助处理器中的硬件监视器计时器，指示系统挂起、用户因挂起启动的密钥序列等。

bug 检查 0x179：CLUSTER_CLUSPORT_STATUS_IO_TIMEOUT_LIVEDUMP

项目 • 2024/02/21

CLUSTER_CLUSPORT_STATUS_IO_TIMEOUT_LIVEDUMP 实时转储的值为 0x00000179。 这表示发起程序节点上的 SMB 客户端抱怨目标节点的 IO 花费的时间过长，并且所有具有 STATUS_IO_TIMEOUT 的 IO 都失败。

CLUSTER_CLUSPORT_STATUS_IO_TIMEOUT_LIVEDUMP 参数

展开表

参数	说明
1	遇到超时的 IRP。
2	保留。
3	保留。
4	保留。

原因

发起程序节点上的 SMB 客户端抱怨目标节点上的 IO 花费的时间过长，并且所有具有 STATUS_IO_TIMEOUT 的 IO 都失败。

为了响应这一点，群集发起程序捕获发起程序节点上的实时转储，以便我们可以分析 IO 花费的时间。

转储的辅助数据流中提供了其他信息。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

解决方法

另请参阅

使用实时转储排查挂起问题（博客）

内核实时转储代码引用

Bug 检查代码参考

bug 检查 17C：PDC_LOCK_WATCHDOG_LIVEDUMP

项目 • 2024/02/23

PDC_LOCK_WATCHDOG_LIVEDUMP 实时转储的值为 0x0000017C。 这表示线程一直持有 PDC 锁定太久。

PDC_LOCK_WATCHDOG_LIVEDUMP 参数

展开表

参数	说明
1	持有 PDC 锁定的线程。
2	锁定监视器超时（以毫秒为单位）。
3	保留。
4	保留。

原因

线程一直持有 PDC 锁定太久。 已创建一个实时转储以提供信息供调查。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

解决方法

使用调试器 !thread 命令显示持有参数 1 中提供的锁定的线程。 分析该代码以确定它为何将锁定保留超过超时期限。

另请参阅

!thread

内核实时转储代码引用

Bug 检查代码参考

bug 检查 0x17D：PDC_UNEXPECTED_REVOCATION_LIVEDUMP

项目 • 2024/02/26

PDC_UNEXPECTED_REVOCATION_LIVEDUMP 实时转储的值为 0x0000017D。 它表示已将激活器意外吊销。

PDC_UNEXPECTED_REVOCATION_LIVEDUMP 参数

展开表

参数	说明
1	已吊销的激活器的客户端 ID。
2	已吊销的激活器客户端。
3	已吊销的激活实例。
4	pdc!_PDC_CLIENT_PROCESS_INFO

原因

激活器已意外吊销。

已创建一个实时转储以提供信息供调查。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

另请参阅

内核实时转储代码引用

Bug 检查代码参考

bug 检查 0x187：VIDEO_DWMINIT_TIMEOUT_FALLBACK_BDD

项目 • 2024/02/23

VIDEO_DWMINIT_TIMEOUT_FALLBACK_BDD 实时转储的值为 0x00000187。 这表明视频回退到 BDD，而不是使用 IHV 驱动程序。 这始终生成实时转储。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

VIDEO_DWMINIT_TIMEOUT_FALLBACK_BDD 参数

展开表

参数	说明
1	原因代码。 0x1：DWM 无法在重试后初始化、停止显示适配器并回退到 BDD
2	Reserved
3	Reserved
4	Reserved

bug 检查 0x188：CLUSTER_CSVFS_LIVEDUMP

项目 • 2024/02/23

CLUSTER_CSVFS_LIVEDUMP 实时转储的值为 0x00000188。 这表示 CSVFS 已启动此实时转储，以帮助调试不一致的状态。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

CLUSTER_CSVFS_LIVEDUMP 参数

展开表

参数	说明
1	原因代码 0x1：将 oplock 上的缓存清除降级为 none 失败 2 - CSVFS!_SCB 的地址 0x2：从 none 升级 oplock 时的缓存清除失败 2 - CSVFS!_SCB 的地址 0x3：设置清除失败模式下的缓存清除 2 - CSVFS!_SCB 的地址 0x4：将 oplock 降级为 none 时的缓存刷新失败 2 - CSVFS!_SCB 的地址
2	请参阅参数 1
3	Reserved
4	Reserved

原因

第一个参数包含当 CSVFS 检测到当前状态可能导致数据损坏或其他不一致的原因代码时，它将使用此状态代码生成实时转储。 Parameter1 包含指向此实时转储创建方案的代码。 应在原因代码的上下文中解释其他参数。

另请参阅

内核实时转储代码引用

Bug 检查代码参考

bug 检查 0x190：WIN32K_CRITICAL_FAILURE_LIVEDUMP

项目 • 2024/02/26

WIN32K_CRITICAL_FAILURE_LIVEDUMP 实时转储的值为 0x00000190。 这表示 Win32k 遇到严重故障。 已捕获实时转储以收集调试信息。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

WIN32K_CRITICAL_FAILURE_LIVEDUMP 参数

展开表

参数	说明
1	故障的类型 0x1：REGION_VALIDATION_FAILURE - 区域超出表面边界。 2- 指向 DC 的指针 3 - 指向 SURFACE 的指针 4 - 指向 REGION 的指针 0x2：OPERATOR_NEW_USED - 运算符“new”用于分配内存。 2 - 保留 3 - 保留 4 - 保留
2	请参阅参数 1
3	请参阅参数 1
4	请参阅参数 1

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x193：VIDEO_DXGKRNL_LIVEDUMP

项目 • 2024/10/16

VIDEO_DXGKRNL_LIVEDUMP 实时转储的值为 0x00000193。 这表示发生 dxgkrnl 触发的实时转储。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

VIDEO_DXGKRNL_LIVEDUMP 参数

展开表

参数	说明
1	原因代码 0x100 内部
2	预留
3	预留
4	Reserved

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

另请参阅

内核实时转储代码引用

错误检查代码参考

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

bug 检查 0x195：SMB_SERVER_LIVEDUMP

项目 • 2024/02/21

SMB_SERVER_LIVEDUMP 实时转储的值 0x00000195。 这表示 SMB 服务器检测到问题，并捕获了内核转储来收集调试信息。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

SMB_SERVER_LIVEDUMP 参数

展开表

参数	说明
1	0x1：I/O 无法在合理的时间内完成。 2 - 指向 I/O SRV2_WORK_ITEM 的指针
2	请参阅参数 1
3	Reserved
4	Reserved

另请参阅

内核实时转储代码引用

Bug 检查代码参考

bug 检查 0x198：UFX_LIVEDUMP

项目 • 2024/02/21

UFX_LIVEDUMP 实时转储的值为 0x00000198。 这表示发生了 UFX 实时转储。

（此代码永远不能用于真正的 bug 检查；它用于标识实时转储。）

UFX_LIVEDUMP 参数

展开表

参数	说明
1	失败类型 0x1：类驱动程序无法激活总线。 2 - 枚举子 PDO 3 的掩码 - 激活的子 PDO 4 的掩码 - 保留
2	请参阅参数 1
3	请参阅参数 1
4	请参阅参数 1

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x19D：CLUSTER_SVHDX_LIVEDUMP

项目 • 2024/02/22

CLUSTER_SVHDX_LIVEDUMP 实时转储的值为 0x0000019D。 这表示 SVHDX 已启动此实时转储，以帮助调试不一致的状态。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

CLUSTER_SVHDX_LIVEDUMP 参数

展开表

参数	说明
1	原因代码 0x1：装载共享虚拟磁盘失败 2 - Svhdxflt!_SVHDX_VIRTUALDISK_CONTEXT 3 的地址 - nt!_FILE_OBJECT 4 的地址 - NTSTATUS
2	请参阅参数 1
3	请参阅参数 1
4	请参阅参数 1

原因

当 SVHDX 检测到当前状态可能会导致某种不一致时，它会生成带有此状态代码的实时转储。 Parameter1 包含指向此实时转储创建方案的代码。 应在原因代码的上下文中解释其他参数。

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x1A0：TTM_WATCHDOG_TIMEOUT

项目 • 2024/02/22

TTM_WATCHDOG_TIMEOUT 错误检查的值为 0x000001A0。 这表明终端拓扑管理器检测到在配置的超时时间内，某些设备特定操作没有完成。

> **ⓘ 重要**
>
> 这篇文章适合程序员阅读。 如果您是在使用计算机时收到蓝屏错误代码的客户，请参阅**蓝屏错误疑难解答**。

TTM_WATCHDOG_TIMEOUT 参数

展开表

参数	说明
1	失败类型 - 下面列出的值。
2	指向设备的指针。
3	指向工作线程的指针。
4	指向标注例程的指针。

失败类型

0x1：终端的设备分配没有进展。

0x2：设备的关闭回调没有进展。

0x3：设备的 set-input-mode 回调没有进展。

0x4：设备的 set-display-state 回调没有进展。

0x5：设置设备的 built-in panel 状态没有进展。

0x6：更新设备的主要显示可见状态没有进展。

原因

终端拓扑管理器检测到在配置的超时时间内，某些设备特定操作没有完成。

另请参阅

Bug 检查代码参考

错误检查 0x1A1：WIN32K_CALLOUT_WATCHDOG_LIVEDUMP

项目 • 2024/02/22

The WIN32K_CALLOUT_WATCHDOG_LIVEDUMP 实时转储的值为 0x000001A1。 Win32k 的标注没有及时返回。

WIN32K_CALLOUT_WATCHDOG_LIVEDUMP 参数

蓝屏上显示以下参数。

展开表

参数	说明
1	从 Win32k 标注返回线程阻止提示。
2	保留。
3	保留。
4	保留。

原因

Win32k 的标注没有及时返回。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

注解

有关使用线程的一般信息，请参阅：

控制进程和线程

Windows 内核模式进程和线程管理器

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x1A3：CALL_HAS_NOT_RETURNED_WATCHDOG_TIMEOUT_LIVEDUMP

项目 • 2024/02/22

调用未在超时期限内返回。

CALL_HAS_NOT_RETURNED_WATCHDOG_TIMEOUT_LIVEDUMP 实时转储的值为 0x000001A3。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

CALL_HAS_NOT_RETURNED_WATCHDOG_TIMEOUT_LIVEDUMP 参数

蓝屏上显示以下参数。

展开表

参数	说明
1	调用未及时返回的线程的进程。
2	其调用未及时返回的线程。
3	超时（以毫秒为单位）。
4	dt nt!_PO_CALL_HAS_NOT_RETURNED_WATCHDOG

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x1A4：DRIPS_SW_HW_DIVERGENCE_LIVEDUMP

项目 • 2024/02/22

DRIPS_SW_HW_DIVERGENCE_LIVEDUMP 实时转储的值为 0x000001A4。

软件和硬件 DRIPS 分歧超过默认/编程阈值时间。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

DRIPS_SW_HW_DIVERGENCE_LIVEDUMP 参数

蓝屏上显示以下参数。

展开表

参数	说明
1	在软件 DRIPS 中花费的时间（微秒）。
2	在硬件 DRIPS 中花费的时间（微秒）。
3	保留。
4	保留。

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x1A5：USB_DRIPS_BLOCKER_SURPRISE_REMOVAL_LIVEDUMP

项目 • 2024/02/22

USB_DRIPS_BLOCKER_SURPRISE_REMOVAL_LIVEDUMP 实时转储的值为 0x000001A5。 它表示 USB 设备因阻塞 DRIPS 而将被意外移除。

USB_DRIPS_BLOCKER_SURPRISE_REMOVAL_LIVEDUMP 参数

展开表

参数	说明
1	阻止设备的 PDO。
2	保留。
3	保留。
4	保留。

原因

在新式待机过程中，一个 USB 设备阻碍了顶层控制器关闭电源，因此会被意外移除。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x1A6：BLUETOOTH_ERROR_RECOVERY_LIVEDUMP

项目 • 2024/02/22

BLUETOOTH_ERROR_RECOVERY_LIVEDUMP 实时转储的值为 0x000001A6。 它表明蓝牙无线电驱动程序 (bthport.sys) 启动了错误恢复，试图从无法补救的内部状况中恢复和重置无线电。

BLUETOOTH_ERROR_RECOVERY_LIVEDUMP 参数

展开表

参数	说明
1	蓝牙无线电（设备）的 PDO。
2	保留。
3	保留。
4	保留。

原因

蓝牙无线电驱动程序 (bthport.sys) 启动了错误恢复，试图从无法补救的内部状况中恢复和重置无线电。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x1A7：SMB_REDIRECTOR_LIVEDUMP

项目 • 2024/02/22

SMB_REDIRECTOR_LIVEDUMP 实时转储的值为 0x000001A7。 它表明 SMB 重定向器检测到了一个问题，并捕获了内核转储以收集调试信息。

SMB_REDIRECTOR_LIVEDUMP 参数

展开表

参数	说明
1	原因代码 - 请参阅下面的值。
2	请参阅下面的值。
3	保留。
4	保留。

SMB 重定向器检测到了一个问题，并捕获了内核转储以收集调试信息。

原因代码

```text
0x1 : An I/O failed to complete in a reasonable amount of time.
    2 - Pointer to the connection object.
    3 - Reserved.
    4 - Reserved.
```

原因

SMB 重定向器检测到了一个问题，并捕获了内核转储以收集调试信息。

只有设置了以下注册表值，才会生成带有该错误检查代码的实时转储。

```registry
HKLM\System\CurrentControlSet\Services\Lanmanworkstation\Parameters [DWORD]
LiveDumpFilter = 1
```

如果设置了此注册表键值并且 RDR 在 IO 上超时，则会发生实时转储。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x1A8：VIDEO_DXGKRNL_BLACK_SCREEN_LIVEDUMP

项目 • 2024/04/15

VIDEO_DXGKRNL_BLACK_SCREEN_LIVEDUMP 实时转储的值为 0x000001A8。 它表示发生了用户启动的 DXGKRNL 实时转储黑屏情况。

VIDEO_DXGKRNL_BLACK_SCREEN_LIVEDUMP 参数

展开表

参数	说明
1	触发 DXGKRNL 黑屏实时转储的源 - 如下所列。
2	保留。
3	保留。
4	保留。

源值

展开表

源	说明
0x1	黑屏热键生成 DXGKRNL 黑屏实时转储
0x2	卷组合键生成 DXGKRNL 黑屏实时转储
0x4	内部生成的 DXGKRNL 黑屏实时转储
0x8	长按电源按钮 (LPBH) 生成 DXGKRNL 黑屏实时转储
0x9	保留用于内部测试
0xA	连接物理显示器但系统无法执行设置为至少在 1 条路径中启用的模式时自动触发

原因

用户为黑屏方案启动了 DXGKRNL 实时转储。 有关触发的实时转储源，请参阅参数 1 的值。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

另请参阅

内核实时转储代码引用

错误检查代码参考

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

错误检查 0x1A9：DIRECTED_FX_TRANSITION_LIVEDUMP

项目 • 2024/02/22

DIRECTED_FX_TRANSITION_LIVEDUMP 实时转储的值为 0x000001A9。 它表示设备的导向式 Fx 转换调用失败。 有关导向式 Fx 的详细信息，请参阅导向式电源管理框架简介。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

DIRECTED_FX_TRANSITION_LIVEDUMP 参数

展开表

参数	说明
1	指向 POP_FX_DEVICE 对象实例的指针。
2	如果此转换为断电，则为 1；如果此转换为上电，则为 0。
3	保留。
4	保留。

原因

设备的导向式 Fx 转换调用失败。

另请参阅

内核实时转储代码引用

错误检查代码参考

错误检查 0x1B0：VIDEO_MINIPORT_FAILED_LIVEDUMP

项目 • 2024/02/22

VIDEO_MINIPORT_FAILED_LIVEDUMP 实时转储的值为 0x000001B0。

VIDEO_MINIPORT_FAILED_LIVEDUMP 参数

展开表

参数	说明
1	原因代码。 值：0x1：*添加设备失败。* 0x2：*启动设备失败。*
2	NTSTATUS
3	保留。
4	保留。

原因

DXGKRNL 检测到一个问题，并捕获了实时转储以收集调试信息。 当视频微型端口驱动程序发生故障时，dxgkrnl 会触发这些 livedumps。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x1B8：VIDEO_MINIPORT_BLACK_SCREEN_LIVE DUMP

项目 • 2024/02/22

VIDEO_MINIPORT_BLACK_SCREEN_LIVEDUMP 实时转储的值为 0x000001B8。 它表示发生了用户启动的 MINIPORT 实时转储黑屏情况。

VIDEO_MINIPORT_BLACK_SCREEN_LIVEDUMP 参数

展开表

参数	说明
1	触发 MINIPORT 黑屏实时转储的源 - 如下所列。
2	保留。
3	保留。
4	保留。

源值

0x1：黑屏热键生成 MINIPORT 黑屏实时转储

0x2：生成 MINIPORT 黑屏实时转储的卷组合键

0x4：内部生成的 MINIPORT 黑屏实时转储

0x8：长按电源按钮 (LPBH) 生成 MINIPORT 黑屏实时转储

原因

用户为黑屏方案启动了 MINIPORT 实时转储。 有关触发的实时转储源，请参阅参数 1 的值。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x1C4：DRIVER_VERIFIER_DETECTED_VIOLATION_LIVEDUMP

项目 • 2024/02/22

DRIVER_VERIFIER_DETECTED_VIOLATION_LIVEDUMP 实时转储的值为 0x000001C4。 这表明检测到一个设备驱动程序试图破坏系统。 这是因为该驱动程序在注册表中被（管理员）指定为可疑驱动程序，并且内核启用了对该驱动程序的实质性检查。 有关详细信息，请参阅驱动程序验证程序。

DRIVER_VERIFIER_DETECTED_VIOLATION_LIVEDUMP 参数

展开表

参数	说明
1	驱动程序冲突的子类。 请参阅下面的值。
2	请参阅下面的值。
3	请参阅下面的值。
4	请参阅下面的值。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

值

values

```
0x00081001: ID of the 'KsDeviceMutex' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00081002: ID of the 'KsStreamPointerClone' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
```

```
!ruleinfo).

0x00081003: ID of the 'KsStreamPointerLock' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00081004: ID of the 'KsStreamPointerUnlock' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00081005: ID of the 'KsCallbackReturn' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00081006: ID of the 'KsIrqlDeviceCallbacks' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00081007: ID of the 'KsIrqlFilterCallbacks' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00081008: ID of the 'KsIrqlPinCallbacks' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00081009: ID of the 'KsIrqlDDIs' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0008100A: ID of the 'KsFilterMutex' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
```

```
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x0008100B: ID of the 'KsProcessingMutex' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00082001: ID of the 'KsTimedPinSetDeviceState' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00082002: ID of the 'KsTimedDeviceCallbacks' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00082003: ID of the 'KsTimedFilterCallbacks' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00082004: ID of the 'KsTimedPinCallbacks' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00082005: ID of the 'KsTimedProcessingMutex' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00071001: ID of the 'PcIrqlDDIs' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
```

```
    Parameter 4 - Reserved

0x00071003: ID of the 'PcIrqlIport' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00071004: ID of the 'PcUnmapAllocatedPages' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00071005: ID of the 'PcAllocatedPages' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00071006: ID of the 'PcRegisterAdapterPower' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00071007: ID of the 'PcAddAdapterDevice' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00071008: ID of the 'PcPropertyRequest' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00071009: ID of the 'PcAllocateAndMapPages' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0007100A: ID of the 'PcPoRequestPowerIrp' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved
```

```
0x00072001: ID of the 'PcTimedWaveRtStreamSetState' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00020002: ID of the 'IrqlApcLte' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020003: ID of the 'IrqlDispatch' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020004: ID of the 'IrqlExAllocatePool' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020005: ID of the 'IrqlExApcLte1' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020006: ID of the 'IrqlExApcLte2' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020007: ID of the 'IrqlExApcLte3' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020008: ID of the 'IrqlExPassive' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020009: ID of the 'IrqlIoApcLte' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved
```

```
0x0002000A: ID of the 'IrqlIoPassive1' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002000B: ID of the 'IrqlIoPassive2' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002000C: ID of the 'IrqlIoPassive3' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002000D: ID of the 'IrqlIoPassive4' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002000E: ID of the 'IrqlIoPassive5' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002000F: ID of the 'IrqlKeApcLte1' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020010: ID of the 'IrqlKeApcLte2' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020011: ID of the 'IrqlKeDispatchLte' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020015: ID of the 'IrqlKeReleaseSpinLock' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020016: ID of the 'IrqlKeSetEvent' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
```

```
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020019: ID of the 'IrqlMmApcLte' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002001A: ID of the 'IrqlMmDispatch' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002001B: ID of the 'IrqlObPassive' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002001C: ID of the 'IrqlPsPassive' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002001D: ID of the 'IrqlReturn' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x0002001E: ID of the 'IrqlRtlPassive' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0002001F: ID of the 'IrqlZwPassive' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00020022: ID of the 'IrqlIoDispatch' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00040003: ID of the 'CriticalRegions' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
```

```
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00040006: ID of the 'QueuedSpinLock' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00040007: ID of the 'QueuedSpinLockRelease' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00040009: ID of the 'SpinLock' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x0004000A: ID of the 'SpinlockRelease' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x0004000E: ID of the 'GuardedRegions' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x0004100B: ID of the 'RequestedPowerIrp' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x0004100F: ID of the 'IoSetCompletionExCompleteIrp' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
```

```
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00043006: ID of the 'PnpRemove' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved

0x00091001: ID of the 'NdisOidComplete' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00091002: ID of the 'NdisOidDoubleComplete' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x0009100E: ID of the 'NdisOidDoubleRequest' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00092003: ID of the 'NdisTimedOidComplete' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x0009200D: ID of the 'NdisTimedDataSend' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x0009200F: ID of the 'NdisTimedDataHang' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
```

```
!ruleinfo).

0x00092010: ID of the 'NdisFilterTimedPauseComplete' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00092011: ID of the 'NdisFilterTimedDataSend' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00092012: ID of the 'NdisFilterTimedDataReceive' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00093004: ID of the 'WlanAssociation' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00093005: ID of the 'WlanConnectionRoaming' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00093006: ID of the 'WlanDisassociation' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00093101: ID of the 'WlanAssert' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Reserved
    Parameter 4 - Reserved
```

```
0x00094007: ID of the 'WlanTimedAssociation' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00094008: ID of the 'WlanTimedConnectionRoaming' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x00094009: ID of the 'WlanTimedConnectRequest' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x0009400B: ID of the 'WlanTimedLinkQuality' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).

0x0009400C: ID of the 'WlanTimedScan' rule that was violated.
    Parameter 2 - A pointer to the string describing the violated rule
condition.
    Parameter 3 - Address of internal rule state (second argument to
!ruleinfo).
    Parameter 4 - Address of supplemental states (third argument to
!ruleinfo).
```

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x1C5：IO_THREADPOOL_DEADLOCK_LIVEDUMP

项目 • 2024/02/22

IO_THREADPOOL_DEADLOCK_LIVEDUMP 实时转储的值为 0x000001C5。 这表明内核模式线程池遇到了死锁情况。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

IO_THREADPOOL_DEADLOCK_LIVEDUMP 参数

展开表

参数	说明
1	池编号。 0x0 : ExPoolUntrusted
2	指向 PEX_WORK_QUEUE 的指针
3	Reserved
4	Reserved

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x1C9：USER_MODE_HEALTH_MONITOR_LIVEDUMP

项目 • 2024/02/22

USER_MODE_HEALTH_MONITOR_LIVEDUMP 实时转储的值为 0x000001C9。 它表明一个或多个关键用户模式组件未能满足运行状况检查。

USER_MODE_HEALTH_MONITOR_LIVEDUMP 参数

展开表

参数	说明
1	未能在配置超时内完成运行状况检查的进程。
2	运行状况监控超时（秒）。
3	监控程序源。 结合流程地址，这有助于识别源。 请参见下面可能的值。 这些值会与 USER_MODE_HEALTH_MONITOR 共享。
4	保留。

监控程序源值

```
0  : WatchdogSourceDefault
     Source was not specified
1  : WatchdogSourceRhsCleanup
     Monitors that RHS process goes away when
     terminating on graceful exit
2  : WatchdogSourceRhsResourceDeadlockBugcheckNow
     RHS was asked to immediately bugcheck machine
      on resource deadlock
3  : WatchdogSourceRhsExceptionFromResource
      Resource has leaked unhandled exception from an entry point,
      RHS is terminating and this watchdog monitors that
      process will go away
4  : WatchdogSourceRhsUnhandledException
      Unhandled exception in RHS.
      RHS is terminating and this watchdog monitors that
      process will go away
5  : WatchdogSourceRhsResourceDeadlock
```

```
      Monitors that RHS process goes away when
      terminating on resource deadlock
6  : WatchdogSourceRhsResourceTypeDeadlock
      Monitors that RHS process goes away when
      terminating on resource type deadlock
7  : WatchdogSourceClussvcUnhandledException
      Unhandled exception in clussvc.
      clussvc is terminating and this watchdog monitors that
      process will go away
8  : WatchdogSourceClussvcBugcheckMessageRecieved
      Another cluster node has sent message asking to bugcheck this node.
9  : WatchdogSourceClussvcWatchdogBugcheck
      User mode watchdog has expired and created netft watchdog
      to bugchecked the node.
       0xA : WatchdogSourceClussvcIsAlive
      Cluster service sends heartbeat to netft every 500 millseconds.
      By default, netft expects at least 1 heartbeat per second.
      If this watchdog was triggered that means clussvc is not getting
      CPU to send heartbeats.
      0x65 : WatchdogSourceRhsResourceDeadlockPhysicalDisk
       A subclass of WatchdogSourceRhsResourceDeadlock.
      0x66 : WatchdogSourceRhsResourceDeadlockStoragePool
       A subclass of WatchdogSourceRhsResourceDeadlock.
      0x67 : WatchdogSourceRhsResourceDeadlockFileServer
       A subclass of WatchdogSourceRhsResourceDeadlock.
      0x68 : WatchdogSourceRhsResourceDeadlockSODAFileServer
       A subclass of WatchdogSourceRhsResourceDeadlock.
      0x69 : WatchdogSourceRhsResourceDeadlockStorageReplica
       A subclass of WatchdogSourceRhsResourceDeadlock.
      0x6A : WatchdogSourceRhsResourceDeadlockStorageQOS
       A subclass of WatchdogSourceRhsResourceDeadlock.
      0x6B : WatchdogSourceRhsResourceDeadlockStorageNFSV2
       A subclass of WatchdogSourceRhsResourceDeadlock.
      0xC9 : WatchdogSourceRhsResourceTypeDeadlockPhysicalDisk
       A subclass of WatchdogSourceRhsResourceTypeDeadlock.
      0xCA : WatchdogSourceRhsResourceTypeDeadlockStoragePool
       A subclass of WatchdogSourceRhsResourceTypeDeadlock.
      0xCB : WatchdogSourceRhsResourceTypeDeadlockFileServer
       A subclass of WatchdogSourceRhsResourceTypeDeadlock.
      0xCC : WatchdogSourceRhsResourceTypeDeadlockSODAFileServer
       A subclass of WatchdogSourceRhsResourceTypeDeadlock.
      0xCD : WatchdogSourceRhsResourceTypeDeadlockStorageReplica
       A subclass of WatchdogSourceRhsResourceTypeDeadlock.
      0xCE : WatchdogSourceRhsResourceTypeDeadlockStorageQOS
       A subclass of WatchdogSourceRhsResourceTypeDeadlock.
      0xCF : WatchdogSourceRhsResourceTypeDeadlockStorageNFSV2
       A subclass of WatchdogSourceRhsResourceTypeDeadlock.
```

原因

一个或多个关键用户模式组件未能通过运行状况检查。

硬件机制（如监视器计时器）可以检测到基本内核服务未执行。 然而，资源不足问题（包括内存泄露、锁竞争和调度优先级配置错误）可能会阻止关键的用户模式组件，而不会阻止 DPC 或耗尽非分页池。

内核组件可以通过定期监控关键应用程序将监视器计时器功能扩展到用户模式。 此实时转储表示用户模式运行状况检查失败，我们将尝试终止此应用程序，并继续监控终止是否及时完成。 如果终止未及时完成，则会对机器进行错误检查，通过重启和/或允许应用程序故障转移到其他服务器来恢复关键服务。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

另请参阅

使用 Windows 错误报告排查故障转移群集问题

故障转移群集系统日志事件

错误检查 0x1C9 USER_MODE_HEALTH_MONITOR

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x1CC：EXRESOURCE_TIMEOUT_LIVEDUMP

项目 • 2024/02/26

EXRESOURCE_TIMEOUT_LIVEDUMP 实时转储的值为 0x000001CC。

内核 ERESOURCE 已超时。这可能表明出现了死锁情况或严重争用，从而导致性能问题。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

EXRESOURCE_TIMEOUT_LIVEDUMP 参数

蓝屏上显示以下参数。

展开表

参数	说明
1	已超时的 ERESOURCE。
2	检测到超时的线程
3	ERESOURCE 争用计数
4	配置的超时（秒）

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x1D1：TELEMETRY_ASSERTS_LIVEDUMP

项目 • 2024/02/22

TELEMETRY_ASSERTS_LIVEDUMP 实时转储的值为 0x000001D1。

遥测断言验证失败。

此代码永远不能用于真正的 Bug 检查；它用于标识包括设备遥测在内的实时转储。

要排除此问题，请检查调用堆栈以查看 MICROSOFT_TELEMETRY_ASSERT（表达式）中的表达式为何无效。

TELEMETRY_ASSERTS_LIVEDUMP 参数

展开表

参数	说明
1	Rva
2	ModuleName
3	TimeStamp
4	SizeOfImage

另请参阅

内核实时转储代码引用

Bug 检查代码参考

错误检查 0x1D4：UCMUCSI_LIVEDUMP

项目 • 2024/02/22

UCMUCSI_LIVEDUMP 实时转储的值为 0x000001D4。 这表明 UcmUcsi 类扩展遇到了错误。 例如，这可能是因为 UCSI 命令超时，或 UCSI 命令由于客户端驱动程序返回失败而导致执行失败。

UcmUcsiCx.sys 是包含的 UCSI 类扩展。 有关详细信息，请参阅 USB-C 型连接器系统软件接口 (UCSI) 驱动程序。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

UCMUCSI_LIVEDUMP 参数

展开表

参数	说明
1	故障类型 - 请参阅以下值
2	UCSI 命令值。
3	如果非零，则是指向附加信息的指针。 使用 `dt UcmUcsiCx!UCMUCSICX_TRIAGE` 进行显示。
4	保留。

故障类型

0x0：UCSI 命令超时，因为固件没有及时响应命令。

0x1：由于客户端驱动程序返回失败或固件返回错误代码，UCSI 命令执行失败。

另请参阅

USB 团队博客 - 调试 UCSI 固件故障

通用串行总线 (USB)

内核实时转储代码引用

错误检查 0x1E1：DEVICE_DIAGNOSTIC_LOG_LIVEDUMP

项目 • 2024/02/26

DEVICE_DIAGNOSTIC_LOG_LIVEDUMP 的值为 0x000001E1。 这表示设备遇到错误并生成诊断数据。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

展开表

参数	说明
1	设备分类。 0x1：存储设备。 2 - 存储问题类型。 0x1：设备死机。 3 - IEEE 组织 ID。 4 - 保留
2	请参阅上面的说明。
3	请参阅上面的说明。
4	请参阅上面的说明。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

另请参阅

内核实时转储代码引用

错误检查代码参考

错误检查 0x1F5：APPLICATION_HANG_KERNEL_LIVEDUMP

项目 • 2024/02/26

APPLICATION_HANG_KERNEL_LIVEDUMP 的值为 0x000001F5。 这表明应用程序在尝试终止时挂起。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

APPLICATION_HANG_KERNEL_LIVEDUMP 参数

展开表

参数	说明
1	保留
2	Reserved
3	Reserved
4	Reserved

原因

在完成应用程序挂起报告（如使用 WerReportHang 函数创建的报告）后，调用了 TerminateProcess，但目标进程未能终止，这可能是因为线程在内核模式下挂起。

此实时内核会审转储可能包含挂起进程的内核线程状态。 内核转储只包含 WER 无法终止的进程的线程，而且只包含内核内存，而不包含用户内存。

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

另请参阅

内核实时转储代码引用

错误检查代码参考

错误检查 0x21C8：MANUALLY_INITIATED_BLACKSCREEN_HOTKEY_LIVE_DUMP

项目 • 2024/05/03

MANUALLY_INITIATED_BLACKSCREEN_HOTKEY_LIVE_DUMP 实时转储的值为 0x000021C8。 这表明系统配置为捕捉实时转储，并且用户按下了黑屏热键 (CTRL + SHIFT + WIN + B)。 实时转储可能有助于确定用户观察到的黑屏的根本原因。

（此代码永远不能用于真正的错误检查；它用于标识实时转储。）

MANUALLY_INITIATED_BLACKSCREEN_HOTKEY_LIVE_DUI 参数

展开表

参数	说明
1	保留
2	预留
3	预留
4	Reserved

解决方法

!analyze 调试扩展显示有关错误检查的信息，并有助于确定根本原因。

另请参阅

内核实时转储代码引用

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

调试器参考

项目 • 2024/03/20

此参考部分包括：

命令行选项

环境变量

调试器命令

调试器相关的 API

调试器错误和警告消息

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

命令行选项

项目 • 2023/03/08

本部分中的内容

- CDB 命令行选项
- KD 命令行选项
- WinDbg 命令行选项
- DbgSrv 命令行选项
- KdSrv 命令行选项
- DbEngPrx 命令行选项
- KDbgCtrl 命令行选项
- DbgRpc 命令行选项
- SymStore 命令行选项

CDB 命令行选项

项目 • 2024/03/08

第一次使用 CDB 或 NTSD 的用户应该从使用 CDB 和 NTSD 进行调试一节开始。

CDB 命令行使用以下语法：

dbgcmd

```
cdb  [ -server ServerTransport | -remote ClientTransport ]
[ -premote SmartClientTransport ] [-log{a|au|o|ou} LogFile]
[-2] [-d] [-ddefer] [-g] [-G] [-hd] [-lines] [-myob] [-bonc]
[-n] [-o] [-s] [-v] [-w] [-cf "filename"] [-cfr "filename"] [-c "command"]
[-robp] [-r BreakErrorLevel]  [-t PrintErrorLevel]
[ -x{e|d|n|i} Exception ] [-x] [-clines lines]
[-i ImagePath]  [-y SymbolPath] [-srcpath SourcePath]
[-aExtension] [-failinc] [-noio] [-noinh] [-noshell] [-nosqm]
[-sdce] [-ses] [-sicv] [-sins] [-snc] [-snul] [-zp PageFile]
[-sup] [-sflags 0xNumber] [-ee {masm|c++}]
[-e Event] [-pb] [-pd] [-pe] [-pr] [-pt Seconds] [-pv]
[ -- | -p PID | -pn Name | -psn ServiceName | -z DumpFile | executable ]
[-cimp] [-isd] [-kqm] [-pvr] [-version] [-vf] [-vf:<opts>] [-netsyms:
{yes|no}]

cdb -iae

cdb -iaec KeyString

cdb -iu KeyString

cdb -QR Server

cdb -wake pid

cdb -?
```

NTSD 命令行语法与 CDB 的语法相同：

dbgcmd

```
ntsd  [ -server ServerTransport | -remote ClientTransport ]
[ -premote SmartClientTransport ] [-log{a|au|o|ou} LogFile]
[-2] [-d] [-ddefer] [-g] [-G] [-hd] [-lines] [-myob] [-bonc]
[-n] [-o] [-s] [-v] [-w] [-cf "filename"] [-cfr "filename"] [-c "command"]
[-robp] [-r BreakErrorLevel]  [-t PrintErrorLevel]
[ -x{e|d|n|i} Exception ] [-x] [-clines lines]
[-i ImagePath]  [-y SymbolPath] [-srcpath SourcePath]
[-aExtension] [-failinc] [-noio] [-noinh] [-noshell] [-nosqm]
[-sdce] [-ses] [-sicv] [-sins] [-snc] [-snul] [-zp PageFile]
[-sup] [-sflags 0xNumber] [-ee {masm|c++}]
```

```
[-e Event] [-pb] [-pd] [-pe] [-pr] [-pt Seconds] [-pv]
[ -- | -p PID | -pn Name | -psn ServiceName | -z DumpFile | executable ]
[-cimp] [-isd] [-kqm] [-pvr] [-version] [-vf] [-vf:<opts>] [-netsyms:
{yes|no}]

ntsd -iae

ntsd -iaec KeyString

ntsd -iu KeyString

ntsd -QR Server

ntsd -wake PID

ntsd -?
```

NTSD 和 CDB 之间的唯一区别是，NTSD 生成一个新的控制台窗口，而 CDB 继承调用它的窗口。 由于 **start** 命令还可用于生成新的控制台窗口，因此以下两个构造将给出相同的结果：

dbgcmd

```
start cdb [parameters]
ntsd [parameters]
```

以下是 CDB 和 NTSD 命令行选项的说明。 只有 **-remote**、**-server**、**-g** 和 **-G** 选项区分大小写。 初始连字符可以替换为正斜杠 (/)。 不采用任何其他参数的选项可以连接，因此 **cdb -o -d -G -g winmine** 可以编写为 **cdb -odGg winmine**。

如果使用 **-remote** 或 **-server** 选项，则该选项必须出现在命令行上的任何其他选项之前。 如果指定了*可执行文件*，则它必须最后出现在命令行上；*可执行文件*名称之后的任何文本都将作为其自己的命令行参数传递给可执行程序。

参数

-server*ServerTransport*
创建可由其他调试器访问的调试服务器。 有关可能的 *ServerTransport* 值的说明，请参阅**激活调试服务器**。 使用此参数时，它必须是命令行上的第一个参数。

-remote*ClientTransport*
创建调试客户端，并连接到已在运行的调试服务器。 有关可能的 *ClientTransport* 值的说明，请参阅**激活调试客户端**。 使用此参数时，它必须是命令行上的第一个参数。

-premote*SmartClientTransport*
创建智能客户端，并连接到已在运行的进程服务器。 有关可能的 *SmartClientTransport*

值的说明，请参阅**激活智能客户端**。

-2
如果目标应用程序是*控制台应用程序*，则此选项会使其位于新的控制台窗口中。 （默认情况下，目标控制台应用程序与 CDB 或 NTSD 共享窗口。）

--
调试客户端服务器运行时子系统 (CSRSS)。 有关详细信息，请参阅调试 CSRSS。

-a*Extension*
设置默认扩展 DLL。 默认值为 *userexts*。 "a"后面不能有空格，并且不得包含 .dll 扩展名。 有关详细信息以及设置此默认值的其他方法，请参阅加载调试器扩展 DLL。

-bonc
如果指定了此选项，则调试器将在会话开始后立即进入目标。 当连接到当前可能未进入到目标的调试服务器时，这一点尤其有用。

-c "*command*"
指定要在启动时运行的初始调试器命令。 此命令必须用引号括起来。 多个命令可以用分号分隔。 （如果你有一个长命令列表，则将它们放在脚本中可能更容易，然后将 **-c** 选项与 **$<、 $><、$><、$$><（运行脚本文件）**命令一起使用。）

如果要启动调试客户端，此命令必须适用于调试服务器。 不允许使用特定于客户端的命令，如 **.lsrcpath**。

-cf "*filename*"
指定脚本文件的路径和名称。 调试器一启动，就会立即执行此脚本文件。 如果 *filename* 包含空格，则必须用引号将其引起来。 如果省略路径，则假定为当前目录。 如果不使用 **-cf** 选项，则当前目录中的文件 ntsd.ini 将用作脚本文件。 如果文件不存在，则不会发生错误。 有关详细信息，请参阅使用脚本文件。

-cfr "*filename*"
指定脚本文件的路径和名称。 调试器一启动，以及目标重新启动时，都会立即执行此脚本文件。 如果 *filename* 包含空格，则必须用引号将其引起来。 如果省略路径，则假定为当前目录。 如果文件不存在，则不会发生错误。 有关详细信息，请参阅使用脚本文件。

-cimp
指示 CDB/NTSD 从 DbgSrv 隐式命令行开始，而不是从要运行的显式进程开始。 此选项是 dbgsrv -pc 的客户端。

-clines*lines*
设置在远程调试期间可以访问的命令历史记录中的大致命令数。 有关详细信息以及更改此数字的其他方法，请参阅使用调试器命令。

-d
将此调试器的控制权传递给内核调试器。 如果正在调试 CSRSS，则此控件重定向始终处于活动状态，即使未指定 **-d** 也是如此。 （在远程调试期间不能使用此选项，请改用 **-ddefer**。）有关详细信息，请参阅从内核调试器控制用户模式调试器。 此选项不能与 **-ddefer** 选项或 **-noio** 选项结合使用。

注意 如果使用 WinDbg 作为内核调试器，那么在这种情况下，WinDbg 的许多熟悉功能都不可用。 例如，不能使用“局部变量”窗口、“反汇编”窗口或“调用堆栈”窗口，也不能单步执行源代码。 这是因为 WinDbg 仅充当目标计算机上运行的调试器（NTSD 或 CDB）的查看器。

-ddefer
将此调试器的控制权传递给内核调试器，除非调试客户端已连接。 （这是 **-d** 的变体，可从调试服务器使用。）有关详细信息，请参阅从内核调试器控制用户模式调试器。 此选项不能与 **-d** 选项或 **-noio** 选项结合使用。

-e*Event*
向调试器发出指定事件已发生的信号。 此选项仅在以编程方式启动调试器时使用。

-ee {**masm**|**c++**}
设置默认表达式计算器。 如果指定 **masm**，将使用 MASM 表达式语法。 如果指定 **c++**，将使用 C++ 表达式语法。 如果省略 **-ee** 选项，则 MASM 表达式语法将用作默认值。 有关详细信息，请参阅计算表达式。

-failinc
使调试器忽略任何可疑的符号。 在调试用户模式或内核模式的小型转储文件时，此选项还将阻止调试器加载任何映像无法映射的模块。 有关详细信息和其他控制方法，请参阅 SYMOPT_EXACT_SYMBOLS。

-g
忽略目标应用程序中的初始断点。 此选项将导致目标应用程序在启动或 CDB 连接到它之后继续运行，除非设置了另一个断点。 有关详细信息，请参阅初始断点。

-G
忽略进程终止时的最后一个断点。 默认情况下，CDB 在映像运行过程中停止。 此选项将导致 CDB 在子级终止时立即退出。 这与输入命令 **sxd epr** 的效果相同。 有关详细信息，请参阅控制异常和事件。

-hd
指定不应使用调试堆。 有关详细信息，请参阅使用 CDB 调试用户模式进程。

-i*ImagePath*
指定生成错误的可执行文件的位置。 如果路径包含空格，则应将其括在引号中。

-iae
安装 CDB 作为事后调试器。 有关详细信息，请参阅启用事后调试。

如果此操作成功，则不显示任何消息；如果失败，将显示一条错误消息。

-iae 参数不得用于任何其他参数。 此命令实际上不会启动 CDB。

-iaec*KeyString*
安装 CDB 作为事后调试器。 *KeyString* 的内容将追加到 **AeDebug** 注册表项的末尾。 如果 *KeyString* 包含空格，则必须用引号将其引起来。 有关详细信息，请参阅启用事后调试。

如果此操作成功，则不显示任何消息；如果失败，将显示一条错误消息。

-iaec 参数不得用于任何其他参数。 此命令实际上不会启动 CDB。

-isd
为任何进程创建打开 CREATE_IGNORE_SYSTEM_DEFAULT 标志。

-iu*KeyString*
将调试器远程处理注册为 URL 类型，以便用户可以使用 URL 自动启动调试器远程客户端。 *KeyString* 格式为 `remdbgeng://RemotingOption`。 *RemotingOption* 是一个字符串，用于定义**激活调试客户端**主题中定义的传输协议。 如果此操作成功，则不显示任何消息；如果失败，将显示一条错误消息。

-iu 参数不得用于任何其他参数。 此命令实际上不会启动 CDB。

-kqm
以静默模式启动 CDB/NTSD。

-lines
启用源行调试。 如果省略此选项，则必须使用 **.lines（切换源行支持）** 命令，然后才能允许进行源调试。 有关控制此操作的其他方法，请参阅 SYMOPT_LOAD_LINES。

-log{**a**|**au**|**o**|**ou**} *LogFile*
开始将信息记录到日志文件。 如果指定的文件已存在，如果使用 **-logo**，它将被覆盖；如果使用 -loga，输出将附加到文件中。 **-logau** 和 **-logou** 选项分别运行类似于 **-loga** 和 **-logo**，只是日志文件是 Unicode 文件。 有关详细信息，请参阅在 CDB 中保留日志文件。

-myob
如果 dbghelp.dll 的版本不匹配，调试器将继续运行。 （如果没有 **-myob** 开关，这将被视为严重错误。）

-n
干扰符号加载：启用符号处理程序的详细输出。 有关详细信息和其他控制方法，请参阅 SYMOPT_DEBUG。

-netsyms {yes|no}
允许或禁止从网络路径加载符号。

-noinh
防止调试器创建的进程从调试器继承句柄。 有关其他控制方法，请参阅使用 CDB 调试用户模式进程。

-noio
阻止调试服务器用于输入或输出。 仅接受调试客户端的输入（以及 **-c** 命令行选项指定的任何初始命令或命令脚本）。

所有输出都将定向到调试客户端。 如果将 NTSD 用于服务器，则根本不会创建控制台窗口。 有关详细信息，请参阅**激活调试服务器**。 此选项不能与 **-d** 选项或 **-ddefer** 选项结合使用。

-noshell
禁止所有 **.shell** 命令。 只要调试器正在运行，即使新的调试会话已经开始，这种禁止也将持续。 有关详细信息，以及禁用 **.shell** 命令的其他方法，请参阅使用 Shell 命令。

-nosqm
禁用遥测数据收集和上传。

-o
调试目标应用程序启动的所有进程（子进程）。 默认情况下，由正在调试的进程创建的进程将正常运行。 有关其他控制方法，请参阅使用 CDB 调试用户模式进程。

-p*PID*
指定要调试的十进制进程 ID。 这用于调试已在运行的进程。 有关详细信息，请参阅使用 CDB 调试用户模式进程。

-pb
防止调试器在附加到目标进程时请求初始进入。 如果应用程序已挂起，或者想要避免在目标中创建进入线程，这非常有用。

-pd
使目标应用程序在调试会话结束时不终止。 有关详细信息，请参阅在 CDB 中结束调试会话。

-pe
指示目标应用程序已经在调试中。 有关详细信息，请参阅重新附加到目标应用程序。

-pn*Name*
指定要调试的进程的名称。 （此名称必须唯一。）用于调试已在运行的进程。

-pr
促使调试器在附加到目标进程时开始运行目标进程。 如果应用程序已挂起，并且希望应用程序恢复执行，这非常有用。

-psn*ServiceName*
指定要调试的进程中包含的服务的名称。 这用于调试已在运行的进程。

-pt*Seconds*
指定中断超时值（以秒为单位）。 默认为 30。 有关详细信息，请参阅控制目标。

-pv
指定调试器应无干扰地附加到目标进程。 有关详细信息，请参阅非侵入性调试（用户模式）。

-pvr
工作方式类似于 **-pv**，只是目标进程未挂起。

-QR*Server*
列出在指定网络服务器上运行的所有调试服务器。 *Server* 前面的双反斜杠 (\) 为可选。 有关详细信息，请参阅**搜索调试服务器**。

-QR 参数不能用于任何其他参数。 此命令实际上不会启动 CDB。

-r*BreakErrorLevel*
指定将导致目标进入到调试器的错误级别。 这是一个十进制数，等于 0、1、2 或 3。 可能的值如下所示：

展开表

值	返回的常量	含义
0	NONE	不要因任何错误而中断。
1	ERROR	在 ERROR 级别的调试事件时中断。
2	MINORERROR	在 MINORERROR 和 ERROR 级别的调试事件时中断。
3	WARNING	在 WARNING、MINORERROR 和 ERROR 级别调试事件时中断。

此错误级别仅在 Microsoft Windows 的已检查版本中有意义。 默认值为 1。 Windows 10 版本 1803 之前的旧版 Windows 上提供已检查的版本。

-robp
这允许 CDB 在只读内存页上设置断点。 （默认情况下，这样的操作会失败。）

-s
禁用延迟符号加载。 这会减慢进程启动速度。 有关详细信息和其他控制方法，请参阅 SYMOPT_DEFERRED_LOADS。

-sdce
使调试器在符号加载期间显示**文件访问错误**对话框。 有关详细信息以及其他控制方法，请参阅 SYMOPT_FAIL_CRITICAL_ERRORS。

-ses
使调试器对所有符号文件执行严格评估，并忽略任何有问题的符号。 有关详细信息和其他控制方法，请参阅 SYMOPT_EXACT_SYMBOLS。

-sflags 0x*Number*
一次设置所有符号处理程序选项。 *数字*应是以 **0x** 为前缀的十六进制数字，允许使用不带 **0x** 的十进制，但符号选项是二进制标志，因此建议使用十六进制。 应谨慎使用此选项，因为它将覆盖所有符号处理程序的默认值。 有关详细信息，请参阅设置符号选项。

-sicv
使符号处理程序忽略 CV 记录。 有关详细信息和其他控制方法，请参阅 SYMOPT_IGNORE_CVREC。

-sins
使调试器忽略符号路径和可执行映像路径环境变量。 有关详细信息，请参阅 SYMOPT_IGNORE_NT_SYMPATH。

-snc
使调试器关闭 C++ 转换。 有关详细信息和其他控制方法，请参阅 SYMOPT_NO_CPP。

-snul
禁用非限定名称的自动符号加载。 有关详细信息和其他控制方法，请参阅 SYMOPT_NO_UNQUALIFIED_LOADS。

-srcpath*SourcePath*
指定源文件搜索路径。 用分号 (;) 分隔多个路径。 如果路径包含空格，则应将其括在引号中。 有关详细信息以及更改此路径的其他方法，请参阅源路径。

-sup
使符号处理程序在每次符号搜索期间搜索公共符号表。 有关详细信息和其他控制方法，请参阅 SYMOPT_AUTO_PUBLICS。

-t*PrintErrorLevel*
指定将导致调试器显示错误消息的错误级别。 这是一个十进制数，等于 0、1、2 或 3。 可能的值如下所示：

展开表

值	返回的常量	含义
0	NONE	不显示任何错误。
1	ERROR	显示 ERROR 级别调试事件。
2	MINORERROR	显示 MINORERROR 和 ERROR 级别调试事件。
3	WARNING	显示 WARNING、MINORERROR 和 ERROR 级别调试事件。

此错误级别仅在 Microsoft Windows 的已检查版本中有意义。 Windows 10 版本 1803 之前的旧版 Windows 上提供已检查的版本。 默认值为 1。

-v
启用调试器的详细输出。

-version
输出调试器版本字符串。

-vf
启用默认 ApplicationVerifier 设置。

-vf:*<opts>*
启用给定的 ApplicationVerifier 设置。

-w
指定在单独的 VDM 中调试 16 位应用程序。

-wake*PID*
使进程 ID 由 *PID* 指定的用户模式调试器的睡眠模式结束。 在睡眠模式下，必须在目标计算机上发出此命令。 有关详细信息，请参阅从内核调试器控制用户模式调试器。

-wake 参数不应与任何其他参数一起使用。 此命令实际上不会启动 CDB。

-x{e|d|n|i} *Exception*
控制指定事件发生时调试器的行为。 *Exception* 可以是异常编号或事件代码。 可以多次指

定此选项来控制不同的事件。 有关详细信息以及控制这些设置的其他方法，请参阅控制异常和事件。

-x
禁用访问冲突异常时第一次中断。 第二次发生访问冲突将中断调试器。 此设置与 **-xd av** 相同。

-y*SymbolPath*
指定符号搜索路径。 用分号 (;) 分隔多个路径。 如果路径包含空格，则应将其括在引号中。 有关更改此路径的详细信息和其他方法，请参阅符号路径。

-z*DumpFile*
指定要调试的崩溃转储文件的名称。 如果路径和文件名包含空格，则必须用引号括起来。 通过包含多个 **-z** 选项，可以同时打开多个转储文件，每个选项后面都有一个不同的 *DumpFile* 值。 有关详细信息，请参阅分析用户模式转储文件。

-zp*PageFile*
指定修改后的页面文件的名称。 如果要调试转储文件，并且想要使用 **.pagein（内存页）** 命令，这非常有用。 不能将 **-zp** 与标准 Windows 页面文件一起使用，只能使用经过特殊修改的页面文件。

executable
指定可执行进程的命令行。 这用于启动新进程并对其进行调试。 这必须是命令行上的最后一项。 可执行文件名称之后的所有文本都作为其参数字符串传递给可执行文件。

-?
显示命令行帮助文本。

当从**开始 | 运行**或从命令提示符窗口启动调试器时，请在应用程序的文件名后指定目标应用程序的参数。 例如：

```
dbgcmd
```

```
cdb myexe arg1arg2
```

KD 命令行选项

项目 • 2024/07/08

KD 的第一次用户应从“使用 KD 和 NTKD 调试”部分开始。

KD 命令行使用以下语法。

dbgcmd

```
kd [ -server ServerTransport | -remote ClientTransport ]
   [-b | -x] [-d] [-bonc] [-m] [-myob] [-lines] [-n] [-r] [-s]
   [-v] [-clines lines] [-failinc] [-noio] [-noshell]
   [-secure] [-sdce] [-ses] [-sicv] [-sins] [-snc] [-snul]
   [-sup] [-sflags 0xNumber] [-log{a|au|o|ou} LogFile]
   [-aExtension] [-zp PageFile]
   [-i ImagePath] [-y SymbolPath]  [-srcpath SourcePath]
   [-k ConnectType | -kl | -kqm | -kx ExdiOptions] [-ee {masm|c++}]
   [-z DumpFile] [-cf "filename"] [-cfr "filename"] [-c "command"]
   [-t PrintErrorLevel] [-version]

kd -iu KeyString

kd -QR Server

kd -wake PID

kd -?
```

以下 KD 命令行选项的说明。 **只有 -remote** 和 **-server** 选项区分大小写。 初始连字符可以替换为正斜杠 (/)。 不采用任何其他参数的选项可以连接 -- 因此 **kd -r -n -v** 可以编写为 **kd -rnv**。

如果使用 **-remote** 或 **-server** 选项，则该选项必须出现在命令行上的任何其他选项之前。

参数

-服务器 *ServerTransport*
创建可由其他调试器访问的调试服务器。 有关可能的 ServerTransport *的说明，请参阅**激活调试服务器**。* 使用此参数时，它必须是命令行上的第一个参数。

-远程 *ClientTransport*
创建调试客户端，并连接到已在运行的调试服务器。 有关可能的 *ClientTransport* 值的说明，请参阅**激活调试客户端**。 使用此参数时，它必须是命令行上的第一个参数。

-**一个** *外延*
设置默认扩展 DLL。 默认值为kdextx86.dll或kdexts.dll。 “a”之后不得有空格，并且不能包含.dll文件扩展名。 有关详细信息以及设置此默认值的其他方法，请参阅加载调试器扩展 DLL。

-b
此选项不再受支持。

-bonc
如果指定了此选项，则调试器将在会话开始后立即进入目标。 当连接到当前可能未进入到目标的调试服务器时，这一点尤其有用。

-c *"command"*
指定要在启动时运行的初始调试器命令。 此命令必须用引号括起来。 多个命令可以用分号分隔。 （如果你有一个长命令列表，则将它们放在脚本中可能更容易，然后将 -c 选项与 $<、$、$><><、$><$（运行脚本文件）**命令一起使用。**

如果要启动调试客户端，此命令必须适用于调试服务器。 不允许使用特定于客户端的命令（如 **.lsrcpath**）。

-cf *"filename"*
指定脚本文件的路径和名称。 调试器一启动，就会立即执行此脚本文件。 如果 *filename* 包含空格，则必须用引号将其引起来。 如果省略路径，则假定为当前目录。 如果不使用 -cf 选项，则当前目录中的文件 ntsd.ini 将用作脚本文件。 如果文件不存在，则不会发生错误。 有关详细信息，请参阅使用脚本文件。

-cfr *"filename"*
指定脚本文件的路径和名称。 调试器一启动，以及目标重新启动时，都会立即执行此脚本文件。 如果 *filename* 包含空格，则必须用引号将其引起来。 如果省略路径，则假定为当前目录。 如果文件不存在，则不会发生错误。 有关详细信息，请参阅使用脚本文件。

-clines *线*
设置在远程调试期间可以访问的命令历史记录中的大致命令数。 有关详细信息以及更改此数字的其他方法，请参阅使用调试器命令。

-d
重新启动后，调试器会在加载内核模块后立即进入目标计算机。 （此中断早于 -b **选项的**中断。有关更改此状态的其他方法的详细信息，请参阅故障和重新启动目标计算机。

-ee {**masm**|**c++**}
设置默认表达式计算器。 如果指定 **masm**，将使用 MASM 表达式语法。 如果指定 **c++**，将使用 C++ 表达式语法。 如果省略 **-ee** 选项，则 MASM 表达式语法将用作默认值。 有关详细信息，请参阅计算表达式。

-failinc
使调试器忽略任何可疑的符号。 在调试用户模式或内核模式的小型转储文件时，此选项还将阻止调试器加载任何映像无法映射的模块。 有关详细信息和其他控制方法，请参阅 SYMOPT_EXACT_SYMBOLS。

-我 *ImagePath*
指定生成错误的可执行文件的位置。 如果路径包含空格，则应将其括在引号中。

-iu *KeyString*
将调试器远程处理注册为 URL 类型，以便用户可以使用 URL 自动启动调试器远程客户端。 *KeyString* 格式为 `remdbgeng://RemotingOption`。 *RemotingOption* 是一个字符串，用于定义**激活调试客户端**主题中定义的传输协议。 如果此操作成功，则不显示任何消息；如果失败，将显示一条错误消息。

-iu 参数不得用于任何其他参数。 此命令实际上不会启动 KD。

-k *ConnectType*
告知调试器如何连接到目标。 有关详细信息，请参阅 使用 KD 和 NTKD 进行调试。

-kl
在调试器所在的同一台计算机上启动内核调试会话。

-kqm
以安静模式启动 KD。

-kx *ExdiOptions*
使用 EXDI 驱动程序启动内核调试会话。 本文档未介绍 EXDI 驱动程序。 如果你有硬件探测或硬件模拟器的 EXDI 接口，请联系Microsoft以获取调试信息。

-线
启用源行调试。 如果省略此选项，则必须使用 **.lines（切换源行支持）** 命令，然后才能允许进行源调试。 有关控制此操作的其他方法，请参阅 SYMOPT_LOAD_LINES。

-log{**a**|**au**|**o**|**ou**} *LogFile*
开始将信息记录到日志文件。 如果 *LogFile* 已存在，则使用 -logo **时**将被覆盖;如果使用 **-loga**，则输出将追加到文件。 **-logau** 和 **-logou** 选项分别运行类似于 **-loga** 和 **-logo**，只是日志文件是 Unicode 文件。 有关详细信息，请参阅 在 KD 中保留日志文件。

-m
指示串行端口已连接到调制解调器。 指示调试器监视运营商检测信号。

-myob
如果 dbghelp.dll 的版本不匹配，调试器将继续运行。 （如果没有 **-myob** 开关，这将被视为严重错误。）

此选项的次要效果是禁止显示闯入目标计算机时通常显示的警告。

-n
干扰符号加载：启用符号处理程序的详细输出。 有关详细信息和其他控制方法，请参阅 SYMOPT_DEBUG。

-noio
阻止调试服务器用于输入或输出。 仅接受调试客户端的输入（以及 **-c** 命令行选项指定的任何初始命令或命令脚本）。

所有输出都将定向到调试客户端。 有关详细信息，请参阅**激活调试服务器**。

-noshell
禁止所有 **.shell** 命令。 只要调试器正在运行，即使新的调试会话已经开始，这种禁止也将持续。 有关详细信息，以及禁用 shell 命令的其他方法，请参阅 “使用 Shell 命令”。

-QR *服务器*
列出在指定网络服务器上运行的所有调试服务器。 双反斜杠 （\\） 前面的 *服务器* 是可选的。 有关详细信息，请参阅**搜索调试服务器**。

-QR 参数不得用于任何其他参数。 此命令实际上不会启动 KD。

-r
显示寄存器。

-s
禁用延迟符号加载。 这会减慢进程启动速度。 有关详细信息和其他控制方法，请参阅 SYMOPT_DEFERRED_LOADS。

-sdce
使调试器在符号加载期间显示**文件访问错误**对话框。 有关详细信息以及其他控制方法，请参阅 SYMOPT_FAIL_CRITICAL_ERRORS。

-安全
激活安全模式。

-ses
使调试器对所有符号文件执行严格评估，并忽略任何有问题的符号。 有关详细信息和其他控制方法，请参阅 SYMOPT_EXACT_SYMBOLS。

-sflags 0x*数*
一次设置所有符号处理程序选项。 *数字*应是以 **0x** 为前缀的十六进制数字，允许使用不带 **0x** 的十进制，但符号选项是二进制标志，因此建议使用十六进制。 应谨慎使用此选项，因为它将覆盖所有符号处理程序的默认值。 有关详细信息，请参阅设置符号选项。

-sicv
使符号处理程序忽略 CV 记录。 有关详细信息和其他控制方法，请参阅 SYMOPT_IGNORE_CVREC。

-罪
使调试器忽略符号路径和可执行映像路径环境变量。 有关详细信息，请参阅 SYMOPT_IGNORE_NT_SYMPATH。

-snc
使调试器关闭 C++ 转换。 有关详细信息和其他控制方法，请参阅 SYMOPT_NO_CPP。

-snul
禁用非限定名称的自动符号加载。 有关详细信息和其他控制方法，请参阅 SYMOPT_NO_UNQUALIFIED_LOADS。

-srcpath *SourcePath*
指定源文件搜索路径。 用分号分隔多个路径（;）。 如果路径包含空格，则应将其括在引号中。 有关详细信息以及更改此路径的其他方法，请参阅源路径。

-支持
使符号处理程序在每次符号搜索期间搜索公共符号表。 有关详细信息和其他控制方法，请参阅 SYMOPT_AUTO_PUBLICS。

-t *PrintErrorLevel*
指定将导致调试器显示错误消息的错误级别。 这是一个十进制数，等于 0、1、2 或 3。 下面描述了这些值：

展开表

值	返回的常量	含义
0	NONE	不显示任何错误。
1	ERROR	显示 ERROR 级别调试事件。
2	MINORERROR	显示 MINORERROR 和 ERROR 级别调试事件。
3	WARNING	显示 WARNING、MINORERROR 和 ERROR 级别调试事件。

此错误级别仅在 Microsoft Windows 的已检查版本中有意义。 默认值是 1秒。 Windows 10 版本 1803 之前的旧版 Windows 上提供已检查的版本。

-v
为加载、延迟加载和卸载生成详细消息。

-版本
输出调试器版本字符串。

-唤醒 *PID*
使进程 ID 由 *PID* 指定的用户模式调试器的睡眠模式结束。 在睡眠模式下，必须在目标计算机上发出此命令。 有关详细信息，请参阅从内核调试器控制用户模式调试器。

-wake 参数不得与任何其他参数一起使用。 此命令实际上不会启动 KD。

-x
导致调试器在首次发生异常时中断，而不是让导致异常处理的应用程序或模块中断。（与 -b **相同**，但初始 **eb nt 除外！ NtGlobalFlag 9;g** command.）

-y *SymbolPath*
指定符号搜索路径。 用分号分隔多个路径（;）。 如果路径包含空格，则应将其括在引号中。 有关更改此路径的详细信息和其他方法，请参阅符号路径。

-z *DumpFile*
指定要调试的崩溃转储文件的名称。 如果路径和文件名包含空格，则必须用引号括起来。 通过包含多个 **-z** 选项，可以同时打开多个转储文件，每个选项后面都有一个不同的 *DumpFile* 值。 有关详细信息，请参阅 使用 KD 分析内核模式转储文件。

-zp *PageFile*
指定修改后的页面文件的名称。 如果要调试转储文件，并且想要使用 **.pagein（内存页）** 命令，这非常有用。 不能将 -zp **与标准 Windows 页面文件一**起使用，只能使用经过特殊修改的页面文件。

-?
显示命令行帮助文本。

KD 将自动检测运行目标的平台。 无需在 KD 命令行上指定目标。 旧语法（使用名称 *I386KD* 或 *IA64KD*）已过时。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

WinDbg 命令行选项

项目 • 2024/07/08

WinDbg 的第一次用户应从“使用 WinDbg（经典）调试”部分开始。

WinDbg 命令行使用以下语法：

dbgsyntax

```
windbg [ -server ServerTransport | -remote ClientTransport ] [-lsrcpath ]
   [ -premote SmartClientTransport ] [-?] [-ee {masm|c++}]
   [-clines lines] [-b] [-d] [-aExtension]
   [-failinc] [-g] [-G] [-hd] [-j] [-n] [-noshell] [-o] [-
openPrivateDumpByHandle Handle]
   [-Q | -QY] [-QS | -QSY] [-robp] [-secure] [-ses] [-sdce]
   [-sicv] [-sins] [-snc] [-snul] [-sup] [-sflags 0xNumber]
   [-T Title] [-v] [-log{o|a} LogFile] [-noinh]
   [-i ImagePath] [-y SymbolPath] [-srcpath SourcePath]
   [-k [ConnectType] | -kl | -kx ExdiOptions] [-c "command"]
   [-pb] [-pd] [-pe] [-pr] [-pt Seconds] [-pv]
   [-W Workspace] [-WF Filename] [-WX] [-zp PageFile]
   [ -p PID | -pn Name | -psn ServiceName | -z DumpFile | executable ]

windbg -I[S]

windbg -IU KeyString

windbg -IA[S]
```

WinDbg 命令行选项的说明如下。 除 -j **之外**，所有命令行选项都区分大小写。 初始连字符可以替换为正斜杠 (/)。

如果使用 **-remote** 或 **-server** 选项，则该选项必须出现在命令行上的任何其他选项之前。 如果指定了*可执行文件*，则它必须最后出现在命令行上；*可执行文件*名称之后的任何文本都将作为其自己的命令行参数传递给可执行程序。

参数

-服务器 *ServerTransport*
创建可由其他调试器访问的调试服务器。 有关可能的 *ServerTransport* 值的说明，请参阅**激活调试服务器**。 使用此参数时，它必须是命令行上的第一个参数。

-远程 *ClientTransport*
创建调试客户端，并连接到已在运行的调试服务器。 有关可能的 *ClientTransport* 值的说明，请参阅**激活调试客户端**。 使用此参数时，它必须是命令行上的第一个参数。

-premote *SmartClientTransport*
创建智能客户端，并连接到已在运行的进程服务器。 有关可能的 *SmartClientTransport* 值的说明，请参阅**激活智能客户端**。

-一个 *外延*
设置默认扩展 DLL。 默认值为kdextx86.dll或kdexts.dll。 “a”之后不得有空格，并且不能包含.dll文件扩展名。 有关详细信息以及设置此默认值的其他方法，请参阅加载调试器扩展 DLL。

-b
不再支持此选项。

-c *" command "*
指定要在启动时运行的初始调试器命令。 此命令必须用引号引起来。 多个命令可以用分号分隔。 （如果你有一个长命令列表，则将它们放在脚本中可能更容易，然后将 -c 选项与 $<、$、$><><、$><$（运行脚本文件）**命令一起使用**。

如果要启动调试客户端，此命令必须适用于调试服务器。 不允许使用特定于客户端的命令（如 .lsrcpath）。

-clines *线*
设置在远程调试期间可以访问的命令历史记录中的大致命令数。 有关详细信息以及更改此数字的其他方法，请参阅使用调试器命令。

-d
（仅限内核模式）重新启动后，调试器会在加载内核模块后立即进入目标计算机。 （此中断早于 -b **选项的**中断。有关更改此状态的其他方法的详细信息，请参阅故障和重新启动目标计算机。

-ee {**masm**|**c++**}
设置默认表达式计算器。 如果指定 **masm**，将使用 MASM 表达式语法。 如果指定 **c++**，将使用 C++ 表达式语法。 如果省略 **-ee** 选项，则 MASM 表达式语法将用作默认值。 有关详细信息，请参阅计算表达式。

-failinc
使调试器忽略任何可疑的符号。 在调试用户模式或内核模式的小型转储文件时，此选项还将阻止调试器加载任何映像无法映射的模块。 有关详细信息和其他控制方法，请参阅 SYMOPT_EXACT_SYMBOLS。

-g
（仅限用户模式）忽略目标应用程序中的初始断点。 此选项将导致目标应用程序在启动或 WinDbg 附加到它后继续运行，除非设置了另一个断点。 有关详细信息，请参阅初始断点。

-G
（仅限用户模式）忽略进程终止时的最终断点。 通常，调试会话在映像运行过程中结束。 此选项将导致调试会话在子级终止时立即结束。 这与输入命令 **sxd epr** 的效果相同。 有关详细信息，请参阅控制异常和事件。

-硬盘
（仅限用户模式）指定不应使用调试堆。

-I[**S**]
将 WinDbg 安装为事后调试器。 有关详细信息，请参阅启用事后调试。

尝试此操作后，将显示成功或失败消息。 如果 **包含 S**，则此过程在成功时以无提示方式完成;仅显示失败消息。

-I 参数不得用于任何其他参数。 此命令实际上不会启动 WinDbg，尽管 WinDbg 窗口可能暂时出现。

-IA[**S**]
将 WinDbg 与注册表中的文件扩展名.dmp、.mdmp 和 .wew 相关联。 尝试此操作后，将显示成功或失败消息。 如果 **包含 S**，则此过程在成功时以无提示方式完成;仅显示失败消息。 建立此关联后，选择并按住（或右键单击）具有其中一个扩展名的文件将启动 WinDbg。

-IA 参数不得与任何其他参数一起使用。 此命令实际上不会启动 WinDbg，尽管 WinDbg 窗口可能暂时出现。

-IU *KeyString*
将调试器远程处理注册为 URL 类型，以便用户可以使用 URL 自动启动调试器远程客户端。 *KeyString* 格式为 `remdbgeng://RemotingOption`。 *RemotingOption* 是一个字符串，用于定义**激活调试客户端**主题中定义的传输协议。 如果此操作成功，则不显示任何消息；如果失败，将显示一条错误消息。

-IU 参数不得与任何其他参数一起使用。 虽然 WinDbg 窗口可能暂时出现，但此命令实际上不会启动 WinDbg。

-我 *ImagePath*
指定生成错误的可执行文件的位置。 如果路径包含空格，则应将其括在引号中。

-j
允许日记。

-k [*ConnectType*]
（仅限内核模式）启动内核调试会话。 有关详细信息，请参阅使用 WinDbg 的实时内核模式调试（经典版）。 如果使用 **-k** 而不遵循任何 *ConnectType* 选项，则它必须是命令行上的最后一个条目。

-kl
（仅限内核模式）在调试器所在的同一台计算机上启动内核调试会话。

-kx *ExdiOptions*
（仅限内核模式）使用 EXDI 驱动程序启动内核调试会话。 有关 EXDI 的详细信息，请参阅 配置 EXDI 调试器传输。

-log{**o**|**a**} *LogFile*
开始将信息记录到日志文件。 如果指定的日志文件已存在，则使用 -logo **时**将覆盖该文件。 如果使用 **loga** ，输出将追加到文件。 有关详细信息，请参阅 在 WinDbg 中保留日志文件。

-lsrcpath
设置远程客户端的本地源路径。 此选项必须遵循 **命令行上的** -remote 。

-n
干扰符号加载：启用符号处理程序的详细输出。 有关详细信息和其他控制方法，请参阅 SYMOPT_DEBUG。

-noinh
（仅限用户模式）防止调试器创建的进程从调试器继承句柄。 有关控制此内容的其他方法，请参阅 使用 WinDbg 调试用户模式进程。

-noprio
阻止任何优先级更改。 此参数将阻止 WinDbg 在活动期间优先处理 CPU 时间。

-noshell
禁止所有 **.shell** 命令。 只要调试器正在运行，即使新的调试会话已经开始，这种禁止也将持续。 有关详细信息，以及禁用 shell 命令的其他方法，请参阅 “使用 Shell 命令”。

-o
（仅限用户模式）调试目标应用程序启动的所有进程（子进程）。 默认情况下，由正在调试的进程创建的进程将正常运行。

-openPrivateDumpByHandle *处理*
指定要调试的故障转储文件的句柄。

-p *PID*
指定要调试的十进制进程 ID。 这用于调试已在运行的进程。

-铅
（仅限用户模式）防止调试器在附加到目标进程时请求初始中断。 如果应用程序已挂起，或者想要避免在目标中创建进入线程，这非常有用。

-pd
（仅限用户模式）导致目标应用程序不会在调试会话结束时终止。 有关详细信息，请参阅 在 WinDbg 中 结束调试会话。

-体育
（仅限用户模式）指示目标应用程序已被调试。 有关详细信息，请参阅重新附加到目标应用程序。

-pn *名字*
指定要调试的进程的名称。 （此名称必须唯一。）用于调试已在运行的进程。

-公关
（仅限用户模式）使调试器在附加到目标进程时启动运行的目标进程。 如果应用程序已挂起，并且希望应用程序恢复执行，这非常有用。

-psn *ServiceName*
指定要调试的进程中包含的服务的名称。 这用于调试已在运行的进程。

-铂 *秒*
指定中断超时（以秒为单位）。 默认为 30。 有关详细信息，请参阅控制目标。

-光伏
（仅限用户模式）指定调试器应以非侵入方式附加到目标进程。 有关详细信息，请参阅非侵入性调试（用户模式）。

-Q
禁止显示“保存工作区？”对话框。 不会自动保存工作区。 有关详细信息，请参阅“使用工作区 - WinDbg”（经典）。

-QS
禁止显示“重新加载源？”对话框。 源文件不会自动重新加载。

-QSY
禁止显示“重新加载源？”对话框，并自动重新加载源文件。

-QY
禁止显示“保存工作区？”对话框，并自动保存工作区。 有关详细信息，请参阅“使用工作区 - WinDbg”（经典）。

-robp
这允许 CDB 在只读内存页上设置断点。 （默认情况下，这样的操作会失败。）

-sdce
导致调试器在符号加载期间显示 **文件访问错误消息** 。 有关详细信息以及其他控制方法，请参阅 SYMOPT_FAIL_CRITICAL_ERRORS。

-安全
激活安全模式。

-ses
使调试器对所有符号文件执行严格评估，并忽略任何有问题的符号。 有关详细信息和其他控制方法，请参阅 SYMOPT_EXACT_SYMBOLS。

-sflags 0x *数*
一次设置所有符号处理程序选项。 *数字*应是以 0x 为前缀的十六进制数字，允许使用不带 0x 的十进制，但符号选项是二进制标志，因此建议使用十六进制。 应谨慎使用此选项，因为它将覆盖所有符号处理程序的默认值。 有关详细信息，请参阅设置符号选项。

-sicv
使符号处理程序忽略 CV 记录。 有关详细信息和其他控制方法，请参阅 SYMOPT_IGNORE_CVREC。

-罪
使调试器忽略符号路径和可执行映像路径环境变量。 有关详细信息，请参阅 SYMOPT_IGNORE_NT_SYMPATH。

-snc
使调试器关闭 C++ 转换。 有关详细信息和其他控制方法，请参阅 SYMOPT_NO_CPP。

-snul
禁用非限定名称的自动符号加载。 有关详细信息和其他控制方法，请参阅 SYMOPT_NO_UNQUALIFIED_LOADS。

-srcpath *SourcePath*
指定源文件搜索路径。 用分号分隔多个路径（;）。 如果路径包含空格，则应将其括在引号中。 有关详细信息以及更改此路径的其他方法，请参阅源路径。

-支持
使符号处理程序在每次符号搜索期间搜索公共符号表。 有关详细信息和其他控制方法，请参阅 SYMOPT_AUTO_PUBLICS。

-T *标题*
设置 WinDbg 窗口标题。

-v
启用调试器的详细输出。

-W *工作*
加载给定的命名工作区。 如果工作区名称包含空格，请用引号将其括起来。 如果不存在此名称的工作区，则会选择使用此名称创建新工作区或放弃加载尝试。 有关详细信息，请参阅使用工作区 - WinDbg （经典版）。

-WF *文件名*
从给定文件加载工作区。 *文件名* 应包含文件和扩展名（通常是 .wew）。 如果工作区名称包含空格，请用引号将其括起来。 如果不存在具有此名称的工作区文件，则会选择创建具有此名称的新工作区文件或放弃加载尝试。 有关详细信息，请参阅使用工作区 - WinDbg （经典版）。

-WX
禁用自动工作区加载。 有关详细信息，请参阅使用工作区 - WinDbg （经典版）。

-y *SymbolPath*
指定符号搜索路径。 用分号分隔多个路径（;）。 如果路径包含空格，则应将其括在引号中。 有关更改此路径的详细信息和其他方法，请参阅符号路径。

-z *DumpFile*
指定要调试的崩溃转储文件的名称。 如果路径和文件名包含空格，则必须用引号括起来。 通过包含多个 **-z** 选项，可以同时打开多个转储文件，每个选项后面都有一个不同的 *DumpFile* 值。 有关详细信息，请参阅使用 WinDbg 分析用户模式转储文件或分析内核模式转储文件。

-zp *PageFile*
指定修改后的页面文件的名称。 如果要调试转储文件，并且想要使用 **.pagein（内存页）** 命令，这非常有用。 不能将 **-zp** 与标准 Windows 页面文件一起使用，只能使用经过特殊修改的页面文件。

可执行
指定可执行进程的命令行。 这用于启动新进程并对其进行调试。 这必须是命令行上的最后一项。 可执行文件名称之后的所有文本都作为其参数字符串传递给可执行文件。 有关详细信息，请参阅 使用 WinDbg 调试用户模式进程。

-?
弹出此 HTML 帮助窗口。

从命令行运行调试器时，请在应用程序文件名后为目标应用程序指定参数。 例如：

dbgcmd

```
windbg myexe arg1 arg2
```

反馈

此页面是否有帮助? 是 否

DbgSrv 命令行选项

项目 • 2024/07/08

DbgSrv 命令行使用以下语法。

控制台

```
dbgsrv -t ServerTransport [-sifeo image.ext] -c[s] AppCmdLine [-x | -pc]

dbgsrv -?
```

所有选项都区分大小写。

参数

-t *ServerTransport*
指定传输协议。 有关每种情况下 ServerTransport 的可能协议和语法*的列表，请参阅"**激活进程服务器**"*。

-sifeo *可执行*
挂起给定映像的映像文件执行选项 (IFEO) 值。 *可执行文件* 应包括可执行文件映像的文件名，包括文件扩展名。 -sifeo 选项允许将 DbgSrv 设置为 -c 选项创建的映像的 IFEO 调试器，而不会由于 IFEO 设置导致递归调用。 仅当使用 -c 时，才能使用此选项。

-c
使 DbgSrv 创建新进程。 可以使用此它来创建要调试的进程。 这类似于从调试器生成新进程，只是创建此过程时不会对其进行调试。 若要调试此过程，请确定其 PID，并在启动智能客户端时使用 **-p** 选项调试该进程。

s
使新创建的进程立即挂起。 如果使用此选项，建议使用 CDB 作为智能客户端，并使用 -pb 命令行选项和 -p PID 结合启动智能客户端。 如果在命令行中包含 -pb 选项，则调试器附加到该选项时，该过程将恢复;否则，可以使用 ~*m **命令恢复该过程**。

AppCmdLine
指定要创建的进程的完整命令行。 *AppCmdLine* 可以是 Unicode 或 ASCII 字符串，并且可以包含任何可打印字符。 在 -c[s] **参数之后显示的所有文本都将形成字符串** ***AppCmdLine。***

-x
导致命令行的其余部分被忽略。 如果从可能将不需要的文本附加到其命令行的应用程序启动 DbgSrv，此选项非常有用。

-个人电脑
导致命令行的其余部分用作“隐式命令行”作为挂起的进程创建。 如果调试器使用“-cimp”连接到此进程服务器，则使用此命令行。 例如，运行 `dbgsrv -t <ServerTransport> -pc notepad.exe`，然后运行 `ntsd -premote <Transport> -cimp` 将导致 ntsd 连接到 dbgsrv 并启动notepad.exe

-?
显示一个消息框，其中包含 DbgSrv 命令行的帮助文本。

有关使用 DbgSrv 的信息，请参阅进程服务器（用户模式）。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

KdSrv 命令行选项

项目 • 2024/07/08

KdSrv 命令行使用以下语法。

```
控制台

kdsrv -t ServerTransport
```

参数

-t *ServerTransport*
指定传输协议。 有关每种情况下 ServerTransport 的可能协议和语法*的列表，请参阅**激活 KD 连接服务器**。*

如果没有参数键入 **kdsrv**，将显示一个包含帮助文本的消息框。

有关使用 KdSrv 的信息，请参阅 KD 连接服务器（内核模式）。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

DbEngPrx 命令行选项

项目 • 2024/07/08

DbEngPrx 命令行使用以下语法。

```
dbgcmd

dbengprx [-p] -c ClientTransport -s ServerTransport

dbengprx -?
```

参数

-p
即使删除了与 DbEngPrx 的所有连接，DbEngPrx 也会继续存在。

-c *ClientTransport*
指定要用于连接到服务器的协议设置。 协议应与创建服务器时使用的协议相匹配。 有关详细信息，请参阅 **激活重复程序**。

-s *ServerTransport*
指定客户端连接到中继器时将使用的协议设置。 有关详细信息，请参阅 **激活重复程序**。

-?
显示一个消息框，其中包含 DbEngPrx 命令行的帮助文本。

有关使用 DbEngPrx 的信息，请参阅 Repeaters。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

KDbgCtrl 命令行选项

项目 • 2024/07/25

KDbgCtrl 命令行使用以下语法：

```
dbgcmd
```

```
kdbgctrl [-e|-d|-c] [-ea|-da|-ca] [-eu|-du|-cu] [-eb|-db|-cb] [-sdb Size | -cdb]

kdbgctrl -cx

kdbgctrl -td ProcessID File

kdbgctrl -sd {active|automatic|full|kernel|mini}

kdbgctrl -?
```

参数

-e
启用完整内核调试。

-d
禁用完整内核调试。

-c
检查是否启用了完整内核调试。 如果启用了“完整内核调试”，则显示 true；如果禁用了“完整内核调试”，则显示 false。

-ea
启用自动内核调试。

-da
禁用自动内核调试。

-ca
检查是否启用了自动内核调试。 如果启用了“自动内核调试”，则显示 true；如果禁用了“自动内核调试”，则显示 false。

-eu
启用用户模式错误处理。

-du
禁用用户模式错误处理。

-cu
检查是否启用了用户模式错误处理。 如果启用了“用户模式错误处理”，则显示 true；如果禁用了“用户模式错误处理”，则显示 false。

-eb
启用阻止内核调试。

-db
禁用阻止内核调试

-cb
检查是否阻止了内核调试。 如果内核调试被阻止，则显示 true；如果未阻止内核调试，则显示 false。

-sdb *Size*
设置 DbgPrint 缓冲区的大小。 如果 *Size* 以 **0x** 为前缀，则会将其解释为十六进制数字。 如果前缀为 **0**（零），则会将其解释为八进制。 否则，会将其解释为十进制。

-cdb
显示 DbgPrint 缓冲区的当前大小（以字节为单位）。

-cx
确定当前的“完整内核调试”设置，并返回适当的值。 此选项不能与其他选项组合，也不显示任何输出。 它设计用于批处理文件中，可在其中测试 KDbgCtrl 程序的返回值。 可能的返回值如下：

展开表

值	含义
0x10001	已启用完整内核调试。
0x10002	已禁用完整内核调试。
任何其他值	出现了错误。 KDbgCtrl 无法确定完整内核调试的当前状态。

-td *ProcessID File*
获取内核会审转储文件。 输入转储文件的进程 ID 和名称。

-sd {active|automatic|full|kernel|mini}
设置系统崩溃时要收集的转储类型，并重新加载崩溃转储堆栈。 有关转储类型的详细信息，请参阅多种内核模式转储文件。

-?

显示 KDbgCtrl 的命令行帮助。

其他信息

有关所有 KDbgCtrl 设置的说明，请参阅使用 KDbgCtrl。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

DbgRpc 命令行选项

项目 • 2024/07/08

DbgRpc 命令行必须始终包含 -l、-e、-t、-c 或 -a 开关之一。 这些开关后面的选项取决于使用的开关。 -s、-p 和 -r 选项可用于任何其他选项。

控制台

```
 dbgrpc [-s Server -p ProtSeq] [-r Radix] -l -P ProcessID -L CellID1.CellID2

dbgrpc [-s Server -p ProtSeq] [-r Radix] -e [-E EndpointName]

dbgrpc [-s Server -p ProtSeq] [-r Radix] -t -P ProcessID [-T ThreadID]

dbgrpc [-s Server -p ProtSeq] [-r Radix] [-c|-a] [-C CallID] [-I IfStart] [-
N ProcNum] [-P ProcessID]

dbgrpc -?
```

参数

-s *服务器*
允许 DbgRpc 从远程计算机查看信息。 服务器名称不应前面有斜杠标记。 有关远程使用 DbgRpc 的详细信息，请参阅 “使用 DbgRpc 工具”。

-p *ProtSeq*
指定要使用的远程传输。 ProtSeq ***的可能值为*ncacn_ip_tcp**（TCP 协议）和**ncacn_np**（命名管道协议）。 建议使用 TCP 协议。 有关远程使用 DbgRpc 的详细信息，请参阅 “使用 DbgRpc 工具”。

-r *基数*
指定要用于命令参数的弧度。 默认值为 base 16。 **如果使用 -r** 参数，则应先将其放在行上，因为它只影响在自身之后列出的参数。 它不会影响 DbgRpc 工具的输出。

-l
显示指定单元格的 RPC 状态信息。 有关示例，请参阅 “获取 RPC 单元格信息”。

ProcessID
指定进程的进程 ID（PID）。 **使用 -l** 选项时，应是服务器包含所需单元格的进程。 **使用 -t** 选项时，应是包含所需线程的进程。 **使用 -c** 或 **-a** 选项时，此参数是可选的;它应该是拥有要显示的调用的服务器进程。

CellID1.CellID2
指定要显示的单元数。

-e
在系统的 RPC 状态信息中搜索终结点信息。 有关示例，请参阅 “获取 RPC 终结点信息”。

EndpointName
指定要显示的终结点数。 如果省略，将显示系统上所有进程的终结点。

-t
在系统的 RPC 状态信息中搜索线程信息。 有关示例，请参阅 “获取 RPC 线程信息”。

ThreadID
指定要显示的线程的线程 ID。 如果省略，则会显示指定进程中的所有线程。

-c
在系统的 RPC 状态信息中搜索服务器端调用（SCALL）信息。 有关示例，请参阅 “获取 RPC 调用信息”。

-一个
在系统的 RPC 状态信息中搜索客户端调用（CCALL）信息。 有关示例，请参阅 获取 RPC 客户端调用信息。 此选项需要完整的 RPC 状态信息。

CallID
指定调用 ID。 此参数是可选的;仅当想要显示与特定 *CallID* 值匹配的调用时，才包含它。

IfStart
指定调用时接口的通用唯一标识符（UUID）的第一个 DWORD。 此参数是可选的;仅当想要显示与特定 *IfStart* 值匹配的调用时，才包含它。

ProcNum
指定此调用的过程编号。 （RPC 运行时通过按 IDL 文件中的位置对接口中的单个例程进行编号来标识各个例程 -- 接口中的第一个例程为 0、第二个 1 等。此参数是可选的;仅当想要显示与特定 *ProcNum* 值匹配的调用时，才包含它。

其他信息

有关调试 Microsoft 远程过程调用 (RPC) 的详细信息，请参阅 RPC 调试。

反馈

此页面是否有帮助? 是 否

SymStore 命令行选项

项目 • 2024/07/08

SymStore 事务支持以下语法形式。 第一个参数是 **添加**、 **查询** 或 **del**。使用 **/?** 显示可用选项。

dbgcmd

```
symstore add [/r] [/p [/l] [-:MSG Message] [-:REL] [-:NOREFS]] /f File /s
Store /t Product [/v Version] [/o] [/c Comment] [/d LogFile] [/compress]

symstore add [/r] [/p [/l] [-:REL] [-:NOREFS]] /g Share /f File /x IndexFile
[/a] [/o] [/d LogFile]

symstore add /y IndexFile /g Share /s Store [/p [-:MSG Message] [-:REL]
[-:NOREFS]] /t Product [/v Version] [/o] [/c Comment] [/d LogFile]
[/compress]

symstore query [/r] /f File /s Store [/o] [/d LogFile]

symstore del /i ID /s Store [/o] [/d LogFile]

symstore /?
```

参数

/f *文件*
指定要添加的文件或目录的网络路径。

/g *共享*
指定最初存储符号文件的服务器和共享。 与 /f 一起使用**时，*Share* 应与文件*说明符的*开头**相同。 与 /y 一起使用**时，*共享*应是原始符号文件的位置（而不是索引**文件）。 这允许您稍后更改文件路径的这一部分，以防您将符号文件移动到其他服务器并共享。

/s *商店*
指定符号存储的根目录。

/m *前缀*
使 SymStore 倾向于在存储文件或更新指针时使用以前缀*开头*的路径中的符号。 此选项不能与 /x **选项一起使用。**

/h { **PUB** |**PRI** }
使 SymStore 更倾向于使用公共符号（如果指定 PUB），或者在存储或更新符号时使用专用符号（如果指定 PRI）。 此选项对二进制文件没有影响。

/我 *ID*
指定事务 ID 字符串。

/p
使 SymStore 存储一个指向文件的指针，而不是存储文件本身。

/l
允许文件*指定的*文件位于本地目录中，而不是网络路径。 （仅当同时使用 /f **和 /p 时**，才能使用此选项。

-：味精 *消息*
将指定的 *消息* 添加到每个文件。 （此选项只能在使用 /p **时**使用。）

-：相对
允许文件指针中的路径为相对路径。 此选项表示 /l 选项。 （此选项只能在使用 /p **时**使用。）

-：NOREFS
省略为所存储的文件和指针创建引用指针文件。 只有使用此选项创建了要更改的存储时，则此选项才在符号存储的初始创建期间有效。 （此选项只能在使用 /p **时**使用。）

/r
使 SymStore 以递归方式添加文件或目录。

/t *产品*
指定产品的名称。

/v *版本*
指定产品的版本。

/c *评论*
指定事务的注释。

/d *LogFile*
指定要用于命令输出的日志文件。 如果未包含，则事务信息和其他输出将发送到 **stdout**。

/o
导致 SymStore 显示详细输出。

/x *IndexFile*
使 SymStore 不存储实际符号文件。 相反，SymStore 会在 *IndexFile* 中记录信息，以使 SymStore 稍后能够访问符号文件。

/一个
使 SymStore 将新索引信息追加到现有索引文件。 （此选项仅用于 **/x** 选项。）

/y *IndexFile*
使 SymStore 从使用 **/x** 创建的文件读取数据。

/彝 *IndexFile*
将带有事务 ID 的注释追加到使用 /x 选项创建的索引文件的末尾。

/z { **PUB** |**PRI** }
使 SymStore 只为指定的符号类型编制索引。 如果 **指定了 PUB**，则只会为已去除完整源信息的符号编制索引。 如果 **指定 PRI**，则只会为包含完整源信息的符号编制索引。 SymStore 将始终为二进制符号编制索引。

/compress [CAB|ZIP]
使 SymStore 创建复制到符号存储区的每个文件的压缩版本，而不是使用文件的未压缩副本。 此选项仅在存储文件而不是指针时有效，因此在使用 /p **选项时**不能使用。 有两个可选参数“CAB”和“ZIP”，用于指定要使用的压缩类型。 如果未指定 CAB 压缩，则默认为 CAB 压缩。

/?
显示 SymStore 命令的帮助文本。

其他信息

有关 SymStore 的详细信息，请参阅 使用符号服务器和符号存储。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

环境变量

项目 • 2023/06/15

此参考部分包括：

常规环境变量

内核模式环境变量

有关使用环境变量进行调试的信息，请参阅 Getting Set Up for Debugging。

常规环境变量

项目 • 2023/06/15

下表列出了可在用户模式和内核模式调试中使用的环境变量。

变量	含义
_NT_DEBUGGER_EXTENSION_PATH = *路径*	指定调试器将首先搜索扩展 DLL 的路径。 *路径* 可以包含驱动器号，后跟冒号 (：)。 使用分号分隔多个目录 (;)。 有关详细信息，请参阅 加载调试器扩展 DLL。
_NT_EXECUTABLE_IMAGE_PATH = *Path*	指定包含二进制可执行文件的路径。 *路径* 可以包含驱动器号，后跟冒号 (：)。 使用分号分隔多个目录 (;)。在某些情况下使用此路径，例如用户模式小型转储文件。 有关详细信息，请参阅 在 WinDbg 中设置符号和可执行映像路径。
_NT_SOURCE_PATH = *路径*	指定包含目标的源文件的路径。 *路径* 可以包含驱动器号，后跟冒号 (：)。 使用分号分隔多个目录 (;)。 有关详细信息，以及更改此路径的其他方法，请参阅 源路径。
_NT_SYMBOL_PATH = *路径*	指定包含符号文件的目录树的根。 *路径* 可以包含驱动器号，后跟冒号 (：)。 使用分号分隔多个目录 (;)。 有关详细信息，以及更改此路径的其他方法，请参阅 符号路径。
_NT_ALT_SYMBOL_PATH = *路径*	指定在_NT_SYMBOL_PATH之前搜索的备用符号路径。 这对于保留符号文件的私有版本很有用。 *路径* 可以包含驱动器号，后跟冒号 (：)。使用分号分隔多个目录 (;)。 有关详细信息，请参阅 符号路径。
_NT_SYMBOL_PROXY = *Proxy*：*Port*	指定要由 SymSrv 使用的代理服务器。 有关详细信息，请参阅 防火墙和代理服务器。
_NT_DEBUG_HISTORY_SIZE = *数字*	指定在远程调试期间可以访问的命令历史记录中的命令数。 由于命令的长度不同，因此可用的行数可能与 *Number* 不完全匹配。 有关详细信息，以及更改此数字的其他方法，请参阅 使用调试器命令。
_NT_DEBUG_LOG_FILE_OPEN = *文件名*	(CDB 和 KD 仅) 指定调试器应向其发送输出的日志文件。
_NT_DEBUG_LOG_FILE_APPEND = *文件名*	(CDB 和 KD 仅) 指定调试器应向其追加输出的日志文件。

变量	含义
_NT_EXPR_EVAL = {**masm** | **c++**}	指定默认表达式计算器。 如果指定 **了 masm**，则将使用 MASM 表达式语法。 如果指定 **了 c++**，则将使用 C++ 表达式语法。 默认使用 MASM 表达式语法。 有关详细信息，请参阅计算表达式 。
_NO_DEBUG_HEAP	指定调试堆不应用于用户模式调试。
DBGENG_NO_DEBUG_PRIVILEGE	防止调试器生成的进程继承 SeDebugPrivilege。
DBGENG_NO_BUGCHECK_ANALYSIS	阻止自动错误检查分析。
DBGHELP_HOMEDIR	指定 SymSrv 和 SrcSrv 使用的默认下游存储的根路径。 *路径* 可以包含驱动器号，后跟冒号 (：) 。 使用分号分隔多个目录 (;) 。
SRCSRV_INI_FILE	指定 SrcSrv 使用的配置文件的路径和名称。 默认情况下，路径是 Windows 调试工具安装目录的 srcsrv 子目录，文件名Srcsrv.ini。 有关详细信息，请参阅源索引 编制。

内核模式环境变量

项目 • 2024/10/31

下表列出了仅在内核模式调试中使用的环境变量。

展开表

变量	含义
_NT_DEBUG_PORT = *ComPort*	指定要在内核连接中使用的 COM 端口。 有关详细信息，请参阅 “获取调试设置”。
_NT_DEBUG_BAUD_RATE = *BaudRate*	指定要通过 COM 端口连接使用的波特率。
KDQUIET =*Anything*	如果定义了 KDQUIET，调试器将以静默模式*运行*。 静默模式涉及三个不同的效果： 1.每次加载或卸载扩展 DLL 时，调试器都不会显示消息。 2. **r （Registers）** 命令不再需要其语法中的等号。 3. 调试器在闯入目标计算机时不会显示警告消息。 还可以使用 **sq （Set Quiet Mode）** 命令控制静默模式。
_NT_DEBUG_CACHE_SIZE = *大小*	指定最大内核调试缓存大小（以字节为单位）。 此缓存保存主机从串行连接接收的数据。 默认值为 1,024,000。
_NT_DEBUG_OPTIONS = *选项*	指定以下两个值之一： NOEXTWARNING 告知调试器在找不到扩展命令时不输出警告。 NOVERSIONCHECK 告知调试器不要检查调试器扩展的版本。 可以使用“设置内核选项”命令修改或显示**这些选项**。
_NT_KD_FILES = *MapFile*	指定驱动程序替换映射文件。 有关详细信息以及控制驱动程序替换的其他方法，请参阅 映射驱动程序文件。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

控制键

项目 • 2023/06/15

参考的这一部分讨论可在调试器中使用的各种控制键。

这些控制键在 KD 中工作，在某些情况下适用于 CDB。 其中许多在 WinDbg (中使用 CTRL+ALT 键，而不仅仅是 CTRL) 。

WinDbg 还使用 Ctrl 键、ALT 键和 F 键作为快捷键来切换菜单选项。 有关 快捷键 的含义列表，请参阅快捷键。

CTRL+\（调试当前调试器）

项目 • 2023/10/26

CTRL+ 键启动 CDB 的新实例;此新调试器将当前调试器作为其目标。

```
dbgcmd
```

```
CTRL+\  ENTER
```

环境

调试器	CDB、NTSD、KD
模式	用户模式、内核模式
目标	实时、故障转储
平台	all

注解

这相当于通过 **remote.exe** 实用工具启动新的 CDB，并使用它调试你正在运行的调试器。

Ctrl+ 类似于 **.dbgdbg (调试当前调试器)** 命令。

CTRL+ALT+\（调试当前调试器）

项目 • 2023/10/26

Ctrl+Alt+ 键启动 CDB 的新实例;此新调试器将当前调试器作为其目标。

```
dbgcmd
```

```
CTRL+ALT+\
```

环境

调试器	WinDbg
模式	用户模式、内核模式
目标	实时转储、故障转储
平台	all

注解

这相当于通过 **remote.exe** 实用工具启动新的 CDB，并使用它调试已运行的调试器。

CTRL+Alt+ 类似于 **.dbgdbg (调试当前调试器)** 命令，但 **CTRL+Alt+** 的优点是，在没有可用的调试器命令行时可以使用它。

CTRL+A（切换波特率）

项目 • 2023/09/03

Ctrl+A 键切换内核调试连接中使用的波特率。

KD 语法

```
dbgcmd

CTRL+A   ENTER
```

WinDbg 语法

```
dbgcmd

CTRL+ALT+A
```

环境

调试器	仅限 KD 和 WinDbg
模式	仅限内核模式
目标	仅实时调试
平台	all

备注

这将循环访问内核调试连接的所有可用波特率。

支持的波特率为 19200、38400、57600 和 115200。 每次使用此控制键时，都将选择下一个波特率。 如果波特率已为 115200，则降至 19200。

在 WinDbg 中，也可以通过选择 “**调试|**”**来实现此目的内核连接 |周期波特率。**

实际上不能使用此控制键更改调试的波特率。 主计算机和目标计算机的波特率必须匹配，并且目标计算机的波特率不能在不重新启动的情况下更改。 因此，如果两台计算机尝试以不同的速率通信，则只需切换波特率。 在这种情况下，必须更改主计算机的波特率以匹配目标计算机的波特率。

CTRL+B（退出本地调试器）

项目 • 2023/10/26

CTRL+B 键会导致调试器突然终止。 这不会结束远程调试会话。

```
dbgcmd
```

```
CTRL+B   ENTER
```

环境

调试器	仅限 CDB 和 KD
模式	用户模式、内核模式
目标	实时转储、故障转储
平台	all

注解

在 CDB 中，q (Quit) 命令应用于退出。 仅当调试器没有响应时，才应使用 CTRL+B。

在 KD 中， **q** 命令将结束调试会话并使目标计算机保持锁定状态。 如果需要保留调试会话 (以便新的调试器可以) 连接到它，或者如果需要使目标计算机保持运行，则应使用 CTRL+B。

在 WinDbg 中，等效的命令是 **File |退出** 或 Alt+F4。

CTRL+C（中断）

项目 • 2023/09/03

CTRL+C 键会中断调试器，停止目标应用程序或目标计算机，并取消调试器命令。

CDB 语法

```
dbgcmd

CTRL+C
```

KD 语法

```
dbgcmd

CTRL+C
```

目标计算机语法

```
dbgcmd

SYSRQ
ALT+SYSRQ
F12
```

环境

调试器	仅限 CDB 和 KD
模式	用户模式、内核模式
目标	实时、故障转储
平台	all

其他信息

有关发出此命令的其他方法和相关命令的概述，请参阅 控制目标。

备注

使用 CDB 时：

在用户模式下，CTRL+C 键会导致目标应用程序进入调试器。 目标应用程序冻结，调试器变为活动状态，并且可以输入调试器命令。

如果调试器已处于活动状态，则 Ctrl+C 不会影响目标应用程序。 但是，它可用于终止调试器命令。 例如，如果已请求长显示器，但不想再看到它，CTRL+C 将结束显示并返回到调试器命令提示符。

使用 CDB 执行远程调试时，可以在主计算机的键盘上按 Ctrl+C。 如果要对目标计算机的键盘发出中断，请在 x86 计算机上使用 Ctrl+C。

在调试的应用程序正忙时，可以使用 F12 键获取命令提示符。 在目标应用程序的窗口之一上设置焦点，然后按目标计算机上的 F12 键。

使用 KD 时：

在内核模式下，Ctrl+C 键会导致目标计算机进入调试器。 这会锁定目标计算机并唤醒调试器。

调试仍在运行的系统时，必须按主机键盘上的 Ctrl+C 才能获取初始命令提示符。

如果调试器已处于活动状态，则 Ctrl+C 不会影响目标计算机。 但是，可以使用它来终止调试器命令。 例如，如果已请求长显示器，但不想再看到它，CTRL+C 将结束显示并返回到调试器命令提示符。

当调试器命令生成长显示器或目标计算机正忙时，还可以使用 Ctrl+C 获取命令提示符。 调试 x86 计算机时，可以在主机或目标键盘上按下它。

增强型键盘上的 SYSRQ (或 ALT+SYSRQ) 类似。 它可在任何处理器上通过主机或目标键盘工作。 但是，仅当之前至少按一次 Ctrl+C 获取提示时，它才有效。

可以通过编辑注册表来禁用 SYSRQ 密钥。 在注册表项中

```text
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\i8042prt\Parameters
```

创建名为 **BreakOnSysRq** 的值，并将其设置为 DWORD 0x0。 然后重新启动。 完成此操作后，按目标计算机的键盘上的 SYSRQ 键不会中断到内核调试器中。

使用 CDB 调试 KD 时：

如果要使用 CDB 调试 KD，则主机调试器将截获 CTRL+C (CDB) 。 若要闯入目标调试器 (KD) ，应改用 **CTRL+F** 。

注意 请注意，在 WinDbg 中，Ctrl+C 是用于从窗口中复制文本的 快捷键 。 若要在 WinDbg 中发出 break 命令，请使用 **Ctrl+BREAK** 或选择“调试 ” |从菜单中中断。

CTRL+D（切换调试信息）

项目 • 2023/09/03

CTRL+D 键可打开和关闭调试器内部信息流。 这用于在调试器无法正常通信的情况下重启目标计算机与主计算机之间的通信。

KD 语法

```
dbgcmd

CTRL+D   ENTER
```

WinDbg 语法

```
dbgcmd

CTRL+ALT+D
```

环境

调试器	仅限 KD 和 WinDbg
模式	仅限内核模式
目标	仅实时调试
平台	all

备注

打开此选项后，调试器将输出有关主计算机与目标计算机之间的通信的信息。

此键可用于测试调试器是否已崩溃。 如果怀疑调试器或目标已冻结，请使用此密钥。 如果看到正在发送数据包，则表示目标仍在工作。 如果看到超时消息，则表示目标未响应。 如果根本没有消息，则表示调试器已崩溃。

如果目标未响应，请使用 Ctrl+R Enter Ctrl+C。 如果超时消息继续出现，则表示目标 (崩溃，或者调试器) 配置错误。

这对于调试 KD 调试器本身也很有用。

CTRL+F（中断到 KD）

项目 • 2023/04/14

Ctrl+F 键取消命令或中断调试器。 （使用 CDB 调试 KD 本身时，此控制键特别有用。）

dbgcmd

```
CTRL+F   ENTER
```

环境

调试器	CDB、KD
模式	用户模式、内核模式
目标	实时、故障转储
平台	all

注解

在典型情况下，CTRL+F 与 KD 和 CDB 中的 CTRL+C 以及 **调试 | 中的标准中断命令相同** (WinDbg.) 中的 Break 或 CTRL+BREAK

但是，如果使用 CDB 调试 KD，则这两个密钥的工作方式不同。 CTRL+C 将被主机调试器截获 (CDB) ，而 CTRL+F 将由目标调试器截获 (KD) 。

另请参阅

.breakin（突入内核调试程序）

CTRL+K（更改重新启动后的中断状态）

项目 • 2023/09/03

Ctrl+K 键更改调试器自动进入目标计算机的条件。

KD 语法

```
dbgcmd

CTRL+K   ENTER
```

WinDbg 语法

```
dbgcmd

CTRL+ALT+K
```

环境

调试器	仅限 KD 和 WinDbg
模式	仅限内核模式
目标	仅实时调试
平台	all

其他信息

有关相关命令的概述和重新启动过程如何影响调试器的说明，请参阅 崩溃和重新启动目标计算机。

备注

此控制键会导致内核调试器循环运行以下三种状态：

无中断
在此状态下，调试器不会闯入目标计算机，除非按 Ctrl+C。

重新启动时中断
在此状态下，调试器将在内核初始化后中断到重新启动的目标计算机中。 这等效于使用 -

b命令行选项启动 KD 或 WinDbg。

第一个模块加载时中断
在此状态下，在加载第一个内核模块后，调试器将中断到重新启动的目标计算机中。（这将导致中断早于 **重新启动时中断** 选项。）这等效于使用 **-d**命令行选项启动 KD 或 WinDbg。

使用 Ctrl+K 时，将显示新的中断状态。

在 WinDbg 中，也可以通过选择 “**调试** ” **| 来实现此目的内核连接 |循环初始中断。**

CTRL+P（调试当前调试器）

项目 • 2024/03/20

CTRL+P 命令已过时，已由 Ctrl+\ 和 Ctrl+Alt+\替换。

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

CTRL+R（重新同步）

项目 • 2023/06/15

CTRL+R 键与目标计算机同步。

KD 语法

```
dbgcmd

CTRL+R   ENTER
```

WinDbg 语法

```
dbgcmd

CTRL+ALT+R
```

环境

调试器	仅限 KD 和 WinDbg
模式	仅限内核模式
目标	仅实时调试
平台	all

其他信息

有关与目标重新建立联系的其他方法，请参阅 与目标计算机同步。

注解

这会尝试将主计算机与目标计算机同步。 如果目标未响应，请使用此键。

如果使用 1394 内核连接，重新同步可能并不总是成功。

CTRL+V（切换详细模式）

项目 • 2023/09/03

CTRL+V 键可打开和关闭详细模式。

CDB/KD 语法

```
dbgcmd

CTRL+V   ENTER
```

WinDbg 语法

```
dbgcmd

CTRL+ALT+V
```

环境

调试器	CDB、KD、WinDbg
模式	用户模式、内核模式
目标	实时转储、故障转储
平台	all

备注

启用详细模式后，某些显示命令 (，例如寄存器转储) 生成更详细的输出。 将显示发送到调试器的每个 MODULE LOAD 操作。 每次操作系统加载驱动程序或 DLL 时，调试器都会收到通知。

在 WinDbg 中，也可以通过选择“查看 |”来实现此目的 **详细输出。**

CTRL+W（显示调试器版本）

项目 • 2023/10/26

CTRL+W 键显示调试器和所有已加载扩展 DLL 的版本信息。

CDB/KD 语法

```
dbgcmd

CTRL+W  ENTER
```

WinDbg 语法

```
dbgcmd

CTRL+ALT+W
```

环境

调试器	CDB、KD、WinDbg
模式	用户模式、内核模式
目标	实时转储、故障转储
平台	all

注解

这与 **显示调试器版本) 命令 (** 版本相同，只不过后者也显示 Windows 操作系统版本。

在 WinDbg 中，也可以通过选择"查看 |"来实现此目的 **显示版本**。

另请参阅

version（显示调试器版本）

vertarget（显示目标计算机版本）

调试器相关的 API

项目 • 2024/03/20

此参考部分包括：

符号和图像文件 API

dbgeng.h 标头文件

wdbgexts.h 标头文件

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

符号和图像文件 API

项目 • 2023/06/15

有关使用 DbgHelp 接口处理符号的详细信息，请参阅：

调试帮助库

DbgHelp 参考

有关设置自定义符号服务器的信息，请参阅 自定义符号存储和符号服务器。

有关使用可移植 PDB (Program Database) 格式符号的信息，请参阅 可移植 PDB 符号。

有关使用 ImageHlp 接口处理 PE 文件的信息，请参阅：

图像帮助库

ImageHlp 参考

dbgeng.h 标头文件

项目 • 2023/06/15

dbgeng.h 头文件中的例程用于编写 DbgEng 扩展。

有关 dbgeng.h 头文件的详细信息以及有关如何编写调试器扩展的信息，请参阅 编写 DbgEng 扩展 和使用 调试器引擎 API。

另请参阅

调试器相关的 API

wdbgexts.h 标头文件

项目 • 2023/06/15

wdbgexts.h 头文件中的例程用于编写 DbgEng 扩展和 WdbgExts 扩展。

有关 wdbgexts.h 头文件的详细信息以及有关如何编写调试器扩展的信息，请参阅 编写 WdbgExts 扩展。

调试器错误和警告消息

项目 • 2024/03/20

本参考部分介绍调试器可以显示的一些错误和警告消息。

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

dbgerr001:PEB 已移出页面

项目 • 2023/06/15

调试器错误 **dbgerr001** 显示消息“PEB 已分页”。此错误表示无法访问进程环境块 (PEB) 。

若要加载符号，调试器将查看操作系统加载的模块列表。 指向用户模式模块列表的指针是 PEB 中存储的项之一。

为了回收内存，内存管理器可能会分页出用户模式数据，以便为其他进程或内核模式组件提供空间。

发生此错误时，表示调试器已检测到此进程的 PEB 数据结构不再可读。 最有可能的是，已分页到磁盘。

如果没有此数据结构，调试器将无法确定应为其加载符号的图像。

注意 这仅影响用户模式模块的符号文件。 内核模式模块和符号不受影响，因为它们在不同的列表中进行跟踪。

dbgerr002:错误的指针

项目 • 2023/06/15

调试器错误 **dbgerr002** 显示消息“指针错误”。此错误表示检索符号文件时出现问题。

符号服务器已为文件编制索引，但会重定向到另一个位置以查找该文件。 此其他位置无法访问任何文件。

此问题的两个常见原因是：

- 该路径是 UNC 路径，并且包含此服务器的计算机不可用。
- 路径指示已删除的目录或文件。

如果符号存储是使用 SymStore 创建的，则可以通过检查 file.ptr 找到此文件的完整路径。 有关详细信息，请参阅 使用 SymStore。

dbgerr003:符号不匹配

项目 • 2023/06/15

调试器错误 **dbgerr003** 显示消息“*文件* 具有不匹配的符号”。此错误表示 DbgHelp 找到了 File 表示的符号文件，但符号文件不是正确的版本，或者 DbgHelp 无法确认符号文件是正确的版本。

尽管出现此错误，但调试器可能会加载指定的符号文件，具体取决于路径中的其他要求。

dbgerr004:转储文件中不存在页面

项目 • 2023/10/26

调试器错误 **dbgerr004** 显示消息“转储文件中不存在页 *码* ”。此错误指示调试器需要一个未包含在正在调试的转储文件中的内存页。

指定的 *编号* 是对应于原始页物理内存中位置的 PFN (页帧编号) 。

若要禁止显示此错误消息，请使用 **.ignore_missing_pages 1** 命令。 此命令允许继续调试，但不显示此错误消息。

内核模式内存转储有三种大小，较小的内存转储不包括所有内存页。 有关详细信息，请参阅 各种 Kernel-Mode 转储文件。

用户模式内存转储也有多种大小。 有关详细信息， 请参阅用户模式转储文件 。

dbgerr005:需要专用符号

项目 • 2023/06/15

调试器错误 **dbgerr005** 显示消息“局部变量需要私有符号 (symbols.pri) ”。此错误表示调试器无法执行操作，因为不存在专用符号。

在内核模式调试期间，调试器需要 Microsoft Windows 的符号。 在用户模式调试期间，调试器需要目标应用程序的符号，并且通常还需要 Windows 的符号。

即使是最简陋的调试，也需要一些基本符号，例如函数名称和全局变量。 这些称为 *公共符号*。 尽管在更深入的调试会话中，数据结构名称、仅在一个对象文件中可见的全局变量、局部变量和行号信息等符号并非始终是必需的。 这些符号称为 *专用符号*。

许多软件制造商（包括 Microsoft）都会生成其符号文件的两个版本。 发布给其客户的版本仅包含公共符号。 内部使用的版本同时包含公共符号和私有符号。

大多数调试操作都可以单独使用公共符号来执行。 但某些操作（例如显示局部变量）需要私有符号。 尝试执行此类操作且私有符号不可用时，将显示此错误消息。

看到此消息时，通常最好只是继续调试。 无法获取的信息对于正确调试目标可能并不重要。

堆栈展开信息不可用

项目 • 2023/06/15

当调试器检查调用堆栈时，它可能会显示消息“堆栈展开信息不可用。 以下帧可能出错。”

此警告指示调试器不确定此消息后列出的调用堆栈中的帧是否正确。

为了跟踪调用堆栈，调试器会检查堆栈并分析已使用该堆栈的函数。 这样，它就可以标识构成调用堆栈的返回地址链。 但是，此过程需要包含使用堆栈的函数的每个模块的符号信息。

如果此符号信息不可用，则强制调试器对返回地址的帧做出最佳猜测。 显示此警告信息以指示在此点之后调用堆栈的不确定性质。

没有可用的标头信息

项目 • 2023/06/15

调试器通过检查相关模块的标头来标识正确的符号。 如果这些模块标头已分页，则调试器 (，符号服务器) 找不到正确的符号。 发生这种情况时，符号错误消息中会显示“没有可用的标头信息”。

有关如何在模块标头分页时调试目标的信息，请参阅 从Paged-Out标头读取符号。