第30章　WinDBG 用法详解

WinDBG 是个非常强大的调试器，它设计了极其丰富的功能来支持各种调试任务，包括用户态调试、内核态调试、调试转储文件、远程调试等等。WinDBG 具有非常大的灵活性和可扩展性，用来满足各种各样的调试需求，比如用户可以自由定义调试事件的处理方式，编写调试扩展模块来定制和补充 WinDBG 的调试功能。

尽管 WinDBG 是个典型的窗口程序，但是它的大多数调试功能还是以手工输入命令的方式来工作的。目前版本的 WinDBG 共提供了 20 多条标准命令，140 多条元命令（Meta-commands），和难以计数的大量扩展命令。学习和灵活使用这些命令是学习 WinDBG 的关键，也是难点。

上一章我们从设计的角度分析了 WinDBG，本章将从使用（用户）的角度介绍 WinDBG。我们先介绍工作空间的概念和用法（第 1 节），然后介绍命令的分类和不同种类的命令提示符（第 2 节）。第 3 节介绍不同的调试模式，也就是如何与不同特征的调试目标建立调试会话。第 4 节介绍上下文的概念和在调试时应该如何切换和控制上下文。第 5 节介绍调试事件和如何定制调试事件的处理方式。从第 6 节到第 9 节我们将分别介绍如何在 WinDBG 中完成典型的调试操作，比如控制调试目标（第 6 节）、设置断点（第 7 节）、观察栈（第 8 节）以及如何观察和修改数据（第 9 节）。

30.1　工作空间

WinDBG 使用工作空间（Workspace）来描述和存储一个调试项目的属性、参数、以及调试器设置等信息。其功能类似于集成开发环境的项目文件。

30.1.1 分类

WinDBG 定义了两种工作空间，一种称为缺省的工作空间（Default Workspace），另一种称为命名的工作空间（Named Workspace）。当没有明确使用某个命名的工作空间时，WinDBG 总是使用缺省的工作空间，因此缺省的工作空间也叫隐含的（implicit）工作空间，命名的工作空间也叫显式的（explicit）工作空间。

WinDBG 安装时就预先创建了一系列缺省的工作空间，分别是：

- 基础工作空间（base workspace），当调试会话尚未建立，WinDBG 处于模糊状态时，它会使用基础工作空间。
- 缺省的内核态工作空间（default kernel-mode workspace），当在 WinDBG 中开始内核调试，但是尚未与调试目标建立起联系时，WinDBG 会缺省使用这个工作空间。
- 缺省的远程调试工作空间（remote default workspace），当通过调试服务器（DbgSrv 或 KdSrv）进行远程调试时，WinDBG 会缺省使用这个工作空间。
- 特定处理器的工作空间（processor-specific workspace），在进行内核调试时，当 WinDBG 与调试目标建立起联系，并知道对方的处理器类型后，WinDBG 会缺省使用其对应处理器类型的工作空间。典型的处理器类型有 x86、AMD64、Itanium 等。
- 缺省的用户态工作空间（default user-mode workspace），当 WinDBG 正在附加到一个用户态进程的过程中时，它会使用这个工作空间。

- 相对于可执行文件的缺省工作空间，当在用户态调试时，一旦 WinDBG 知道了调试目标的可执行文件名后（对于附加到已经运行的进程，是附加到进程后，对于调试新运行的程序，那么是选定程序文件后），它就会使用这个可执行文件所对应的工作空间，如果这个工作空间已经存在，那么它就用存在的，如果不存在，那么就创建一个新的。
- 相对于转储文件（dump file）的工作空间，在分析转储文件时，WinDBG 会为每个转储文件建立和维护一个工作空间。

在通过 WinDBG 文件菜单的 Save workspace as…命令调出的 Save Workspace As 对话框的标题中包含了 WinDBG 当前所使用工作空间的名字。当 WinDBG 切换到一个新的工作空间或者退出前通常也会提示是否要保存工作空间，提示对话框（图 30-8）的标题中包含了工作空间的名字。

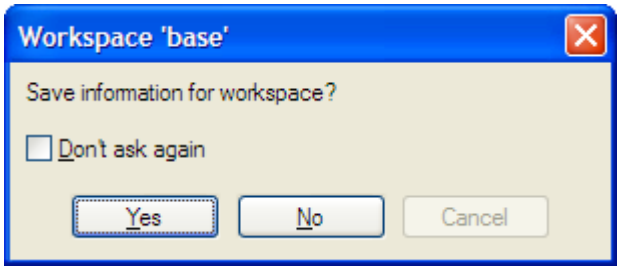


图 30-1 切换或者关闭调试会话时 WinDBG 提示是否要保存工作空间

将当前工作空间另存为一个特定的名字，那么便创建了一个命名的工作空间。

30.1.2 内容

工作空间保存了如下几类信息：

- 调试会话状态：包括断点，打开的源文件，用户定义的别名（alias）等。
- 调试器设置：包括符号文件路径，可执行映像文件路径，源文件路径，用 l+/l-命令设置的源文件选项，日志文件设置，通过启动内核调试对话框设置内核调试连接设置，最近一次打开文件对话框所使用的路径，和输出设置等。
- WinDBG 图形界面信息：包括 WinDBG 窗口的标题，是否自动打开反汇编窗口，缺省字体，WInDBG 窗口在桌面的位置，打开的 WinDBG 子窗口，每个打开窗口的详细信息，包括位置，浮动状态等，命令窗口的设置，是否显示状态条和工具条，寄存器窗口的定制信息，源文件窗口的光标位置，变量观察窗口的变量信息等等。

除了以上信息外，命名的工作空间还可以保存调试会话的状态，对于调试转储文件和调试新创建进程的情况，打开对应的工作空间，WinDBG 可以自动重新开始调试会话。

30.1.3 存储

WinDBG 缺省使用注册表来保存工作空间设置。其路径为：

HKEY_CURRENT_USER\Software\Microsoft\Windbg\Workspaces

在这个键下通常有四个子健 User、Kernel、Dump 和 Explicit（参见图 30-1），前三个子键分别用来保存用户态调试、内核态调试、调试转储文件时使用的缺省工作空间，Explicit 用来保存命名的工作空间。

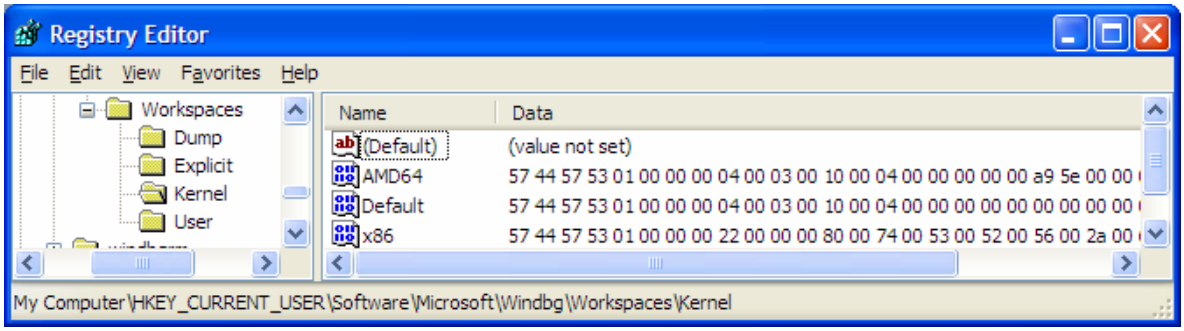


图 30-1 注册表中保存 WinDBG 工作空间的子键

在以上四个子键下的每个键值对应于一个工作空间，键值名是工作空间的名称，键值值就是这个工作空间的二进制数据。

WinDBG 支持使用文件来保存工作空间。使用 File 菜单的 Save Workspace to File 功能就能将当前的工作空间保存为一个.WEW 文件。这个文件是二进制的，其格式与注册表中的数据是一样的。

30.1.4应用

启动 WinDBG 时可以通过-W 开关指定要使用的工作空间名称，也可以通过 File 菜单来打开一个工作空间以显式的加载这个工作空间的设置。

值得说明的是，工作空间中的除了图形界面以外的大多数设置都是以累积方式被应用的。WinDBG 启动时，它便应用缺省的基础设置，而后加载新的工作空间时，WinDBG 只是加载这个新工作空间中的特别内容。

30.1.5删除

通过 WinDBG 的 File 菜单的 Delete Workspaces 可以删除工作空间。另一种更快速的方法就是使用注册表编辑器（regedit）直接删除注册表中上文所述位置的键值。如果要删掉全部，那么就把 Workspaces 键全部删除。可以不用担心缺省的工作空间删掉会影响 WinDBG 运行，下次使用时它会自动为缺省工作空间创建新的键值。

30.1.6主题

WinDBG 程序目录中的 Themes 子目录中包含了四种经过定制的工作空间设置（主要是窗口布局），称为主题（Theme）。每个主题配备了一个.reg 文件和一个.WEW 文件。将.reg 文件导入到注册表或者使用 WinDBG 打开.WEW 文件（Open Workspace in File）便可以应用对应的主题。读者可以以这些主题为基础，经过调整，然后保存为自己所喜欢的设置。

30.2 命令概览

WinDBG 主要是以命令方式工作的，本节我们将概括性的介绍 WinDBG 的三类命令分类：WinDBG 共支持三类命令：标准命令、元命令和扩展命令。

，然后介绍不同形式的命令行提示符。

30.2.1 标准命令

标准命令通常是一两个字符（version 除外）或者符号，用来提供适用于各种调试目标的最基本调试功能。标准命令是不分大小写的。以下是根据功能归纳的当前版本 WinDBG 所支持的所有标准命令：

- 控制调试目标执行，包括恢复运行的 g 命令、跟踪的 t 命令（trace into）和 p 命令（step over）；
- 观察和修改寄存器的 r 命令；
- 观察、编辑和搜索内存数据的 d 命令、e 命令和 s 命令；
- 观察栈的 k 命令；
- 设置和维护断点的 BP（软件断点）、BA（硬件断点）、BL（列出所有断点）和 BC/BD/BE（清除、禁止和重新启用断点）命令；
- 显示和控制线程的～命令；
- 显示进程的|命令；
- 显示表达式的?命令和显示 C++表达式的??命令；
- 用于汇编的 a 命令和用于反汇编的 u 命令；
- 显示段选择子的 dg 命令；
- 执行命令文件的$命令；
- 设置调试事件处理方式的 sx 命令；
- 显示调试器和调试目标版本的 version 命令；
- 检查符号的 x 命令，搜索符号的 ln 命令，和显示模块列表的 lm 命令；
- 结束调试会话的 q 命令。

值得说明的是上面某些单字符的命令代表了一系列以这个字符开始的双字符命令，比如 d 命令后面通常带上第二个字符来指定数据的显示长度，比如 db（按字节显示）、dd（按双字显示）等。

在命令行输入一个问号（？）可以显示出一个标准命令的列表和简单介绍。

30.2.2 元命令

元命令（Meta-Command）用来提供标准命令没有提供的调试功能，与标准命令一样，元命令也是内建在调试器引擎或者 WinDBG 程序文件中的。

所有元命令都以一个点（.）开始，所以元命令也被称为点命令（Dot Command）。

按照功能，可以把元命令分成如下几类：

- 显示或者设置调试会话或者调试器选项，比如用于符号选项的.symopt，用于符号路径的.sympath 和.symfix，用于源程序文件的.srcpath、.srcnoise 和.srcfix，用于扩展命令模块路径的.extpath，用于匹配扩展命令的.extmatch，用于可执行文件的.exepath，设置反汇编选项的.asm，控制表达式评估器的.expr 命令，等等。
- 控制调试会话或者调试目标，如重新开始调试会话的.restart，放弃用户态调试目标（进程）的.abandon，创建新进程的.create 命令和附加到存在进程的.attach 命令，打开转储文件的.opendump，分离调试目标的.detach，用于杀掉进程的.kill 命令，等等。
- 管理扩展命令模块，包括加载模块的.load 命令，卸载用的.unload 命令和.unloadall 命令，显示已经加载模块的.chain 命令等。
- 管理调试器日志文件，如.logfile（显示信息）、.logopen（打开）、.logappend（追加）、.logclose

（关闭）。

- 远程调试，如用于启动 remote.exe 服务的.remote 命令，启动引擎服务器的.server，列出可用服务器的.servers 命令，用于向远程服务器发送文件的.send_file，用于结束远程进程服务器的.endpsrv，用于结束引擎服务器的.endsrv 命令，等等。
- 控制调试器，如让调试器睡眠一段时间的.sleep 命令，唤醒处于睡眠状态的调试器的.wake 命令，启动另一个调试器来调试当前调试器的.dbgdbg 命令，等等。
- 编写命令程序，包括一系列类似 C 语言关键字的命令，如.if、.else、.elsif、.foreach、.do、.while、.continue、.catch、.break、.continue、.leave、.printf、.block 等。我们将在第 10 节介绍命令程序的编写方法。
- 显示或者转储调试目标数据，如产生转储文件的.dump 命令，将原始内存数据写到文件的.writemem 命令，显示调试会话时间的.time 命令，显示线程时间的.ttime 命令，显示任务列表的.tlist（task list）命令，以不同格式显示数据的.fromats 命令，等等。

输入.help 可以列出所有元命令和每个命令的简单说明。

30.2.3扩展命令

扩展命令（Extension Command）用于扩展某一方面的调试功能。与标准命令和元命令是内建在 WinDBG 程序文件中不同，扩展命令是实现在动态加载的扩展模块（DLL）文件中的。

通过 WinDBG 的 SDK，用户可以编写自己的扩展模块和扩展命令。WinDBG 程序包中包含了常用的扩展命令模块。并存放在以下几个子目录中：

- NT4CHK – 用于 Windows NT 4.0 Checked 版本的扩展模块。
- NT4FRE – 用于 Windows NT 4.0 Free 版本的扩展模块。
- W2KCHK – 用于 Windows 2000 Checked 版本的扩展模块。
- W2KFRE –用于 Windows 2000 Free 版本的扩展模块。
- WINXP - 用于 Windows XP 或者更高版本的 Windows 的扩展模块。
- WINEXT – 适用所有 Windows 版本的扩展模块。

表 30-1 列出了 WINEXT 和 WINXP 木立中的的扩展些模块的名称和简单描述。

表 30-1 WinDBG 工具包中的扩展命令模块

扩展模块	路径	描述
ext.dll	WINEXT	适用于各种调试目标的常用扩展命令。
kext.dll	WINEXT	内核态调试时的常用扩展命令。
uext.dll	WINEXT	用户态调试时的常用扩展命令。
logexts.dll	WINEXT	用于监视和记录 API 调用（Windows API Logging Extensions）。
sos.dll	WINEXT	用于调试托管代码和.Net 程序。
ks.dll	WINEXT	用于调试内核流（Kernel Stream）。
wdfkd.dll	WINEXT	调试使用 WDF（Windows Driver Foundation）编写的驱动程序。
acpikd.dll	WINXP	用于 ACPI 调试，追踪调用 ASL 程序的过程，显示 ACPI 对象。
exts.dll	WINXP	关于堆（!heap）、进程/线程结构（!teb/!peb）、安全信息（!token、!sid、!acl）、和应用程序验证（!avrf）等的扩展命令。
kdexts.dll	WINXP	包含了大量用于内核调试的扩展命令。
fltkd.dll	WINXP	用于调试文件系统的过滤驱动程序（FsFilter）。
minipkd.dll	WINXP	用于调试 AIC78xx 小端口（miniport）驱动程序。
ndiskd.dll	WINXP	用于调试网络有关驱动程序。

ntsdexts.dll	WINXP	实现了!handle、!locks、!dp、!dreg（显示注册表）等命令。
rpcexts.dll	WINXP	用于 RPC 调试。
scsikd.dll	WINXP	用于调试 SCSI 有关的驱动程序。
traceprt.dll	WINXP	用于格式化 ETW 信息。
vdmexts.dll	WINXP	调试运行在 VDM 中的 DOS 程序和 WOW 程序。
wow64exts.dll	WINXP	调试运行在 64 位 Windows 系统中的 32 位程序。
wmitrace.dll	WINXP	显示 WMI 追踪有关的数据结构、缓冲区和日志文件。
WudfExt.dll	DDK 中*	用于调试使用 UMDF 编写的用户态驱动程序。

*目前的 WinDBG 工具包尚未包含这个扩展模块。

执行扩展命令时，应该以叹号（!）开始，叹号在英文中被称为 bang，因此扩展命令也被称为 Bang Command。

执行扩展命令的完整格式是：

![扩展模块名].<扩展命令名> [参数]

其中扩展模块名可以省略，如果省略，WinDBG 会自动在已经加载的扩展模块中搜索这个命令的实现。

因为扩展命令是实现在动态加载扩展模块中的，所以执行时需要加载对应的扩展模块。当调试目标被激活时，WinDBG 会根据调试目标的类型和当前的工作空间，自动加载某些扩展模块。此外，也可以使用以下方法来加载扩展模块：

- 使用.load 命令加上扩展模块的完整路径来加载它。
- 使用.loadby 命令加上扩展模块的名称，WinDBG 会自动到当前配置中定义的扩展模块所有路径中搜索匹配的模块。
- 当使用“!扩展模块名.扩展命令名”的方式执行扩展命令时，WinDBG 会自动搜索和加载指定的模块。

使用.chain 命令可以列出当前加载的所有扩展模块。使用.unload 和.unloadall 命令可以卸载指定的或者全部扩展模块。

30.3 用户界面

本节我们将介绍 WinDBG 的基本用户界面特别是命令窗口的用法，包括命令提示符的含义，和输入命令的一些基本常识。

30.3.1 窗口概览

WinDBG 是个典型的 Windows 窗口程序（图 30-1），最外层是框架窗口，框架窗口的用户区上边是菜单和工具条，下边是状态条，中部可以自由摆放不确定数量的工作窗口。

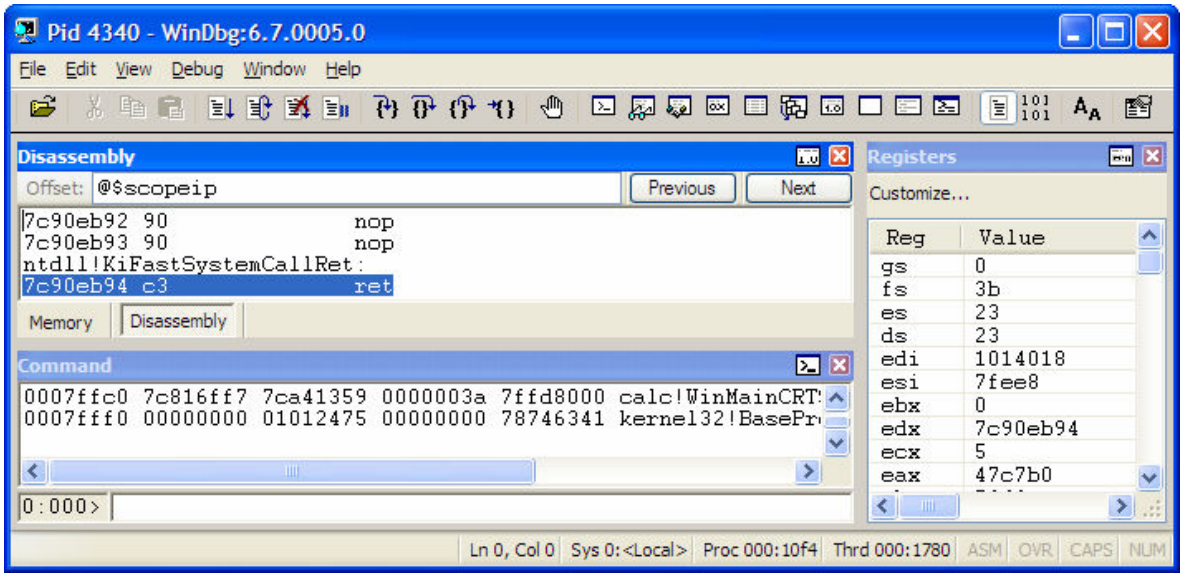


图 30-1 WinDBG 的基本用户界面

图 30-1 中打开了四个常用的子窗口，分别是反汇编窗口、内存窗口、命令窗口和寄存器窗口，其中内存窗口与反汇编窗口共享一个区域，目前没有完全显示出来。通过 View 菜单或者热键还可以打开其它窗口。表 30-1 列出了目前版本的 WinDBG 的所提供能所有类型的工作窗口的名称、热键、和用途。

表 30-1 WinDBG 的工作窗口

名称	热键	用途
Command	Alt+1	输入命令、显示命令结果和调试信息输出
Watch	Alt+2	观察指定全局变量、局部变量和寄存器的信息
Locals	Alt+3	自动显示当前函数的所有局部变量
Registers	Alt+4	观察和修改寄存器的值
Memory	Alt+5	观察和修改内存数据
Call Stack	Alt+6	栈中记录的函数调用序列
Disassembly	Alt+7	反汇编
Scratch Pad	Alt+8	白板，可以用来做调试笔记等
Processes and Threads	Alt+9	显示所有调试目标的列表，包括进程和线程等
Command Browser	Alt+N	执行和浏览命令

图 30-2 显示了图 30-1 中的各个 WinDBG 窗口的窗口属性（句柄、标题、窗口类）以及它们的相互关系。

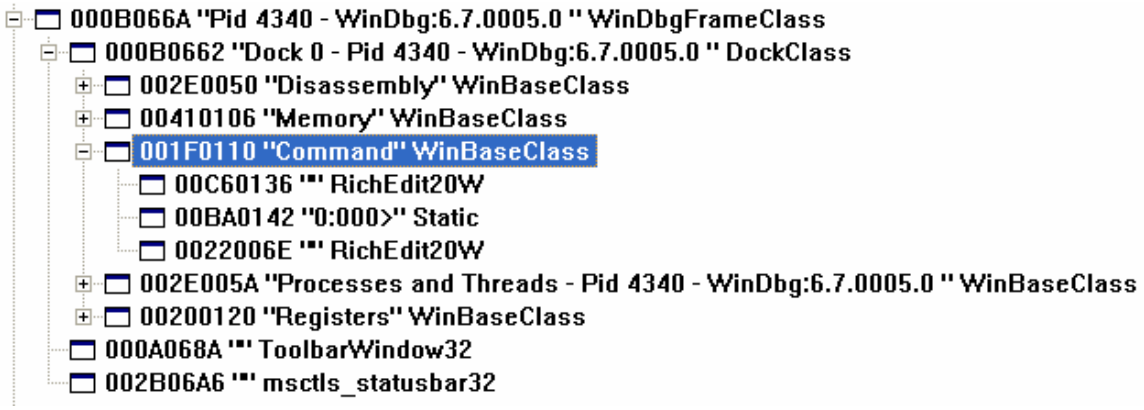


图 30-2 WinDBG 窗口的亲子关系

从图中可以看到，最顶层是窗口类 WinDbgFrameClass 的一个实例，它有三个子窗口，分别是工具条、状态条和一个 DockClass 类型的子窗口 Dock0。Dock0 又有 5 个子窗口，即我们打开的 5 个工作窗口。

WinDBG 支持两种方式来摆放工作窗口，一种是浮动方式（floating），另一种是码放（dock）。对于浮动方式，WinDBG 提供了自动水平平铺、垂直平铺和级联（Cascade）功能。

对于码放方式，所有工作窗口填充在图 30-2 中的 DockClass 窗口中。用户可以使用鼠标来调整子窗口的大小和位置。图 30-1 所示的情况使用的是码放方式。

可以把窗口布局保存到工作空间中，这样下次再打开这个工作空间时，WinDBG 会自动打开上次使用的子窗口并恢复到保存时的状态。

30.3.2 命令窗口和命令提示符

命令窗口是 WinDBG 最主要的人机接口。它主要由上下两个部分组成：上部的信息显示区和下面的命令条。命令条又分为左右两个部分：左边是命令提示符，右边是命令编辑框。

信息显示区是 WinDBG 输出各种调试信息的主要场所，包括命令的执行结果、调试事件、错误信息等。

WinDBG 的命令提示符由一系列文字和大于号两部分组成。因为大于号是固定不变的，所以我们接下来只讨论文字部分。

根据调试目标的类型和调试会话状态的不同，命令提示符的内容也会不同。下面我们分别做详细讨论。

首先，如果调试器处于繁忙状态，那么命令提示符的内容是*BUSY*。繁忙状态通常有三种情况：调试目标处于运行状态，（内核调试时）尚未与调试目标建立起连接，调试器正在加载符号或者处理上一个命令。

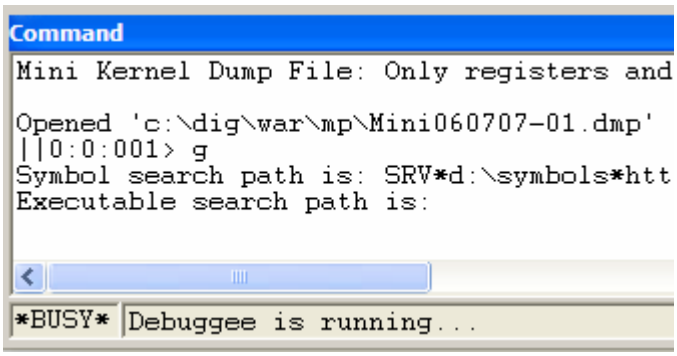


图 30-2

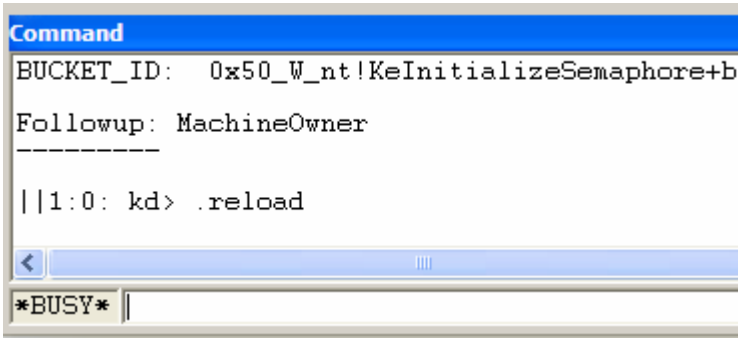


图 30-3

第二，如果目前没有任何调试目标，那么命令提示符是 No Target。

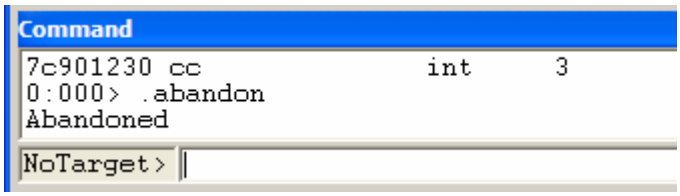


图 30-3 在没有调试目标时的 WinDBG 命令提示符

当 WinDBG 处于空闲状态，也就是等待输入命令时，它的提示符文字是以很简短的方式来描述当前的调试目标，也就是命令的执行目标。考虑到 WinDBG 支持同时调试多个系统中的多个调试目标，因此对于用户态目标，命令提示符的完整形式是：

[||system_index:]process_index:thread_index>

对于双机内核调试时的内核态目标或者内核转储文件目标，命令提示符的完整形式是：

[||system_index:][processor_index:]kd>

对于本地内核态调试，命令提示符的完整形式是：

[||system_index:][processor_index:]lkd>

在以上表示中，system_index 代表系统序号，同一个 Windows 系统中的多个用户态目标属于一个系统，每个内核目标单独属于一个系统；process_index 代表进程序号，processor_index 代表处理器序号；thread_index 线程序号。所有序号都是从 0 开始，全局编排的。当调试目标既有内核态目标，又有用户态目标时，处理器序号与线程序号同等编排，每个内核态目标也被分配一个进程序号。

下面通过一个实验来说明。启动 WinDBG 和一个计算器程序（Calc.exe）。首先将 WinDBg 附加到计算器进程。然后使用.opendump 命令打开一个内核转储文件，需要输入 g 命令执行一次才真正开始建立这个调试目标。最后再使用.create notepad.exe 调试一个新创建的进程（也需要恢复执行一次）。然后打开进程和线程工作窗口，其状态如图 30-5 所示。

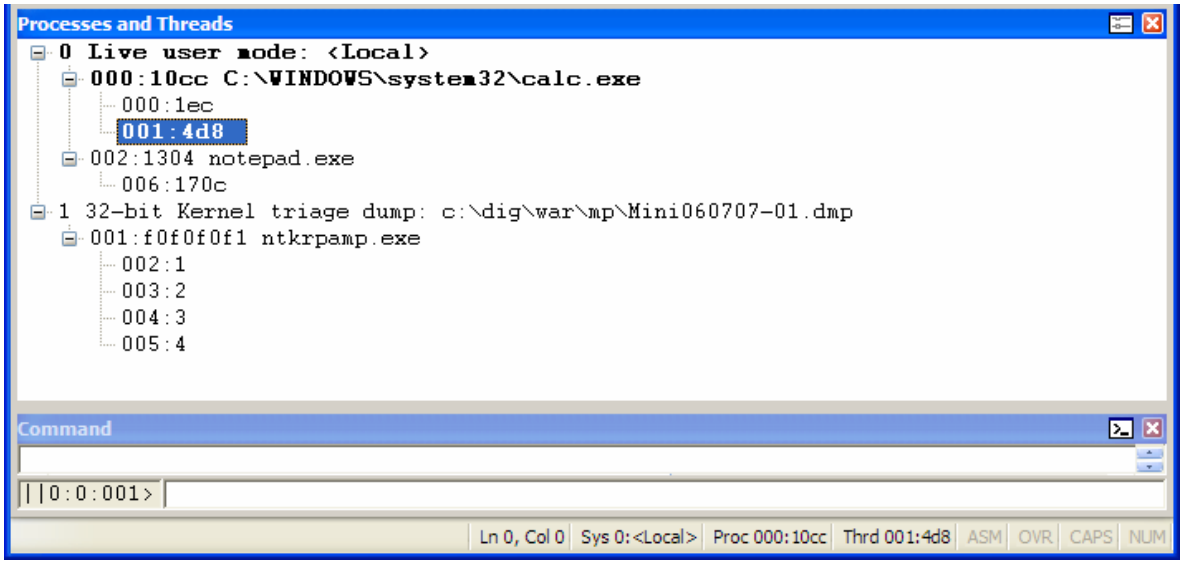


图 30-5 使用 WinDBG 调试多个目标

首先看系统序号的分配，0 号系统代表的是本地的活动用户态调试，1 号系统代表的是内核转储文件。

再看进程号，0 号进程是 Calc 程序，0x10cc 是它的进程 ID。2 号进程是 notepad.exe，0x1304 是它的进程 ID。1 号“进程号”分给了内核转储文件，f0 f0 f0 f1 是个虚拟的进程 ID，如果再打开一个内核转储文件，那么它的虚拟进程号是 f0 f0 f0 f2。因为我们的操作顺序是在创建 notepad 之前打开转储文件，所以转储文件的进程号为 1，另一个为 2。

最后再看线程号和处理器号，0 和 1 分给了 Calc 进程的两个线程。2～5 分给了转储文件的四个处理器，6 号分给了记事本程序的唯一线程。

在 WinDBG 的状态条上，也显示了当前的系统号（Sys 0:<Local>），进程号（Proc 000:10cc）和线程号（001:4d8）。如果切换到转储文件，那么状态条的显示分别为：Sys 1:c:\dig\...；Proc 001:0（进程 ID 显示为 0）；Thrd 002:0（0 代表 0 号 CPU）。

使用||<system_index> s 命令可以切换当前系统。比如对于 30-5 所示的情况，当前系统是 0 号，此时执行||1 s 便可以把当前系统切换到 1 号。使用|<process_index> s 可以切换当前系统中的当前进程。因为进程号是全局编排的，所以可以直接从一个系统的某个进程切换到另一个系统的某个进程。使用~<thread_index> s 可以切换当前线程。只能在一个系统的范围内切换线程。比如，如果当前系统是 0 号，那么执行~6 s 便切换到 Notepad 进程的线程，如果执行~0 s 便切换回计算器进程的 0 号线程。但是如果目前系统是转储文件，那么执行~0 s 或者~6 s 会得到错误：

```
||1:0: kd> ~0 s
                    ^ Extra character error in '~0 s'
||1:0: kd> ~6 s
6 is not a valid processor number
                    ^ Extra character error in '~6 s'
```

30.4 输入和执行命令

本节我们介绍输入和执行 WinDBG 调试命令的一些常识和技巧，包括重复执行、条件执行等。

30.4.1基本要点

首先，大家应该记住以下几个基本要点：

- 直接按回车键可以重复上一条命令。比如在使用 u 命令反汇编时，第一次输入 u 命令后，每次再按回车 WinDBG 便继续反汇编接下来的代码。
- 使用分号（;）号作为分隔符，可以在同一行输入多条命令。
- 按上下方向键可以浏览和选择以前输入过的命令。
- 当命令提示符显示为*BUSY*时，即使命令编辑框可以输入命令（图 30-5），但是这个命令也不会被马上执行，要等 WinDBG 恢复到空闲状态才能执行。
- 输入元命令时应该以点（.）开始，输入扩展命令时应该以叹号（!）开始。
- 使用 Ctrl+Break 来终止一个长时间未完成的命令。如果使用 KD 或则 CDB，那么用 Ctrl+C。

30.4.2注释

WinDBG 支持两种方法在命令中加入注释文字。一种是使用*命令，另一种是使用$$命令。因为二者都是特别的命令，所以使用前应该在前一条命令后加上分号作为分隔。二者的差异是，*之后的所有内容都会被当作注释，而$$后的注释可以用分号结束，然后后面再写其它命令。

例如，在命令编辑框中输入 r eax; $$ address of var_a; r ebx; * var_b; r ecx <will not be

executed>，那么命令信息区显示的执行结果为：

||0:2:006> r eax; $$ address of var_a; r ebx; * var_b; r ecx <will not be executed>

eax=001a1ea4

ebx=7ffdf000

可见，$$ 之后的 r ebx 仍被当作命令执行，而*之后的 r ecx 被当作注释文字。

因为命令的执行结果可以被写入到记录文件中，所以为某些命令加上注释相当于做调试笔记，是一个好的调试习惯。

注释命令的另一个用途就是用在命令程序中，我们将在后面介绍。

30.4.3 别名

WinDBG 支持定义和使用三类别名（Alias）。第一类是所谓的用户命名别名（User-Named Alias），即别名的名称和实体都是用户指定的。第二类是固定名称别名（Fixed-Name Alias），其名称固定为$u0~$u9。第三类是 WinDBG 自动定义的别名。

表 30-1 列出了目前版本的 WinDBG 所定义的自动别名和自动寄存器别名。

别名名称	含义
$ntnsym	NT 内核或者 NT DLL 的符号名，内核态调试，其值为 nt，用户态时为 ntdll。
$ntwsym	在 64 位系统上调试 32 位目标时的 NT 系统 DLL 符号名，可能为 ntdll32 或 ntdll。
$ntsym	与当前调试目标的机器模式匹配的 NT 模块名称。
$CurrentDumpFile	转储文件名称。
$CurrentDumpPath	转储文件路径。
$CurrentDumpArchiveFile	最近加载的 CAB 文件名称。
$CurrentDumpArchivePath	最近加载的 CAB 文件路径。

可以使用.echo 命令来显示某个别名的取值，比如：

||1:0: kd> .echo $ntnsym * 内核态目标

nt

||0:2:006> .echo $ntnsym * 用户态目标

ntdll

可以使用 as 命令来定义或者修改用户命名别名，其基本语法如下：

as 别名名称 别名实体

比如，以下例子为内部命令 version 定义了一个别名 v。

||1:0: kd> as v version

接下来便可以使用 v 来执行 version 命令：

||1:0: kd> v

Windows Server 2003 Kernel Version 3790 (Service Pack 2) MP (4 procs) Free x86 compatible

…

因为别名是在命令真正执行前被替换的，而且 WinDBG 匹配别名的顺序是先搜索固定名称的别名和自动定义的别名，然后再搜索用户定义的别名，所以用户别名名称可以和自动别名或者 WinDBG 的其它命令重复，但是最好不要这样做，以防止混淆。

修改固定别名取值的方法是使用以下格式：

r $.u<0~9>=<别名实体>

例如，以下第一条命令将 u9 定义为 nt!KiServiceTable，第二条命令显示它的内容，第三条是在 dd 命令中使用这个别名。

```
||1:0: kd> r $.u9=nt!KiServiceTable
||1:0: kd> .echo $u9
nt!KiServiceTable
||1:0: kd> dd $u9
80834190    8092023a 8096b71e 8096f9be 8096b750
```

下面我们进一步介绍别名的替换规则。首先，如果用户别名是与命令的其它部分明确分隔开的，那么它可以直接使用这个别名名称，就像上面用 v 来执行 version 命令那样。但如果用户别名是和命令的其它部分连续的，那么必须使用${用户别名}的方式。例如，如果定义 SST 为 nt!KiServiceTable：

```
||1:0: kd> as SST nt!KiServiceTable
```

那么可以以以下三种方式中的任一种来使用这个别名：

```
||1:0: kd> dd SST l4
80834190    8092023a 8096b71e 8096f9be 8096b750
||1:0: kd> dd SST +8 l4
80834198    8096f9be 8096b750 8096f9f8 8096b786
||1:0: kd> dd ${SST}+8 l4
80834198    8096f9be 8096b750 8096f9f8 8096b786
```

但像下面这样便会出错：

```
||1:0: kd> dd SST+8 l4
Couldn't resolve error at 'SST+8 '
```

因为固定别名是长度是确定的，所以使用固定别名时，可以直接使用$u0，不需要大括号，如：

```
||1:0: kd> dd $u9+8 l4
80834198    8096f9be 8096b750 8096f9f8 8096b786
```

使用 al 命令可以列出目前定义的所有用户命名别名。使用 ad 可以删除指定的或者全部（ad *）用户别名。

30.4.4伪寄存器

除了别名，WinDBG 还自动定义了很多伪寄存器（Pseudo-Register）。在命令行和命令文件中都可以使用伪寄存器。WinDBG 会自动将其替换（展开）为合适的值。表 30-5 列出了当前版本的 WinDBG 所定义的伪寄存器。

表 30-5 WinDBG 的伪寄存器

伪寄存器	值的含义
$ea	调试目标所执行上一条指令的有效地址（effective address）。
$ea2	调试目标所执行上一条指令的第二个有效地址。
$exp	表达式评估器所评估的上一条表达式。
$ra	当前函数的返回地址（retrun address）。例如，可以使用 g @$ra 与 gu（go up）具有同样的效果。

$ip	指令指针寄存器。x86 中即 EIP，x64 即 eip。
$eventip	当前调试事件发生时的指令指针。
$previp	上一事件的指令指针。
$relip	与当前事件关联的指令指针，例如按分支跟踪时的分支源地址。
$scopeip	当前上下文（scope）的指令指针。
$exentry	当前进程的入口地址。
$retreg	首要的函数返回值寄存器。x86 架构使用的是 EAX，x64 是 RAX，安腾是 ret0。
$retreg64	64 位格式的首要函数返回值寄存器，x86 中是 edx:eax 寄存器对。
$csp	帧指针。x86 中即 ESP 寄存器，x64 是 RSP，安腾为 BSP。
$p	上一个内存显示命令（d*）所打印的第一个值。
$proc	当前进程的 EPROCESS 结构的地址。
$thread	当前线程的 ETHREAD 结构的地址。
$peb	当前进程的进程环境块（PEB）的地址。
$teb	当前线程的线程环境块（TEB）的地址。
$tpid	拥有当前线程的进程的进程 ID（PID）。
$tid	当前线程的线程 ID。
$bpx	x 号断点的地址。
$frame	当前栈帧的序号。
$dbgtime	当前时间。
$callret	使用.call 命令调用的上一个函数的返回值，或者使用.fnret 命令设置的返回值。
$ptrsize	调试目标所在系统的指针类型宽度。
$pagesize	调试目标所在系统的内存页字节数。

可以直接用上表中的名称来使用伪寄存器，但是更快速的方法是在$前加上一个@符号。这样，WinDBG 就知道@后面是一个伪寄存器，不需要搜索其它符号。

以下是使用伪寄存器的两个例子，我们将在后面给出更多的例子。

```
||0:0:001> ln    @$exentry
(01012475)     calc!WinMainCRTStartup     |  (0101263c)     calc!__CxxFrameHandler
Exact matches:
      calc!WinMainCRTStartup = <no type information>
||0:0:001> ? @$pagesize
Evaluate expression: 4096 = 00001000
```

除了以上 WinDBG 定义的伪寄存器，WinDBG 还为用户准备了 20 个用户伪寄存器（User-Defined Pseudo-Registers），它们的名称是$t0~$t19。可以使用 r 命令来定义用户伪寄存器的实体。

30.4.5循环和条件执行

可以使用 z 命令来循环执行一或多个命令。以下是一个例子：

```
||0:0:001> r ecx=2 *  将 ecx 设置为 2，防止循环太多次
||0:0:001> r ecx=ecx-1; r ecx; z(ecx); recx=ecx+1 *  递减 ecx 直到为 0，然后再递增一次
```

ecx=00000001

redo [1] r ecx=ecx-1; r ecx; z(ecx); recx=ecx+1

ecx=00000000

上面的 redo 行便是循环执行的提示，[1]代表循环次数。执行晚再观察 ecx，其值为 1。概言之，z 命令或循环执行它前面的命令，然后测试自己的条件。循环结束后，再执行 z 命令后的命令。

循环执行的另一种常用方法是使用!for_each_XXX 扩展命令，比如!for_each_frame 命令可以对每个栈帧执行一个操作，!for_each_local 是对每个局部变量。例如以下命令会打印出每个栈帧的每个局部变量：

!for_each_frame !for_each_local dt @#Local

命令 j 可以判断一个条件，然后选择性的执行后面的命令，类似于 C 语言中的 if…else…。其格式为：

j <条件表达式> [Command1>] ; [Command2>]

如果条件成立，那么便执行 Command1，否则便执行 Command2。如果要执行一组命令，那么可以使用单引号，即：

j Expression ['Command1'] ; ['Command2']

例如：

0:001> r ecx; j (ecx<2) 'r ecx';'r eax'

ecx=00000002

eax=7ffdc000

上面的命令是先显示寄存器 ecx 的值，等于 2。而后执行 j 命令，它判断 ecx 是否小于 2，因为不成立，所以便执行后半（else）部分，显示 eax 寄存器的值。因为只有一个命令，所以上面的单引号可以省略，但为了清晰考虑，建议大家总是使用引号来包围两组命令。

以下是一个更复杂一点的例子：

bp `my.cpp:122` "j (poi(MyVar)>5) '.echo MyVar Too Big'; '.echo MyVar Acceptable; gc' "

上面命令的含义是在 my.cpp 的 122 行设置一个断点，当这个断点命中时，WinDBG 自动执行双引号包围的命令，即 j 命令。J 命令判断变量 MyVar 的值是否大于 5，如果是则执行第一对单引号包围的命令（显示 MyVar Too Big，并中断到调试器），如果否则执行第二对单引号包围的命令（显示 MyVar Acceptable，然后立刻恢复目标继续执行）。

调试执行的另一种方法是使用元命令中的.if、.else 和.elseif。比如上面的例子可以表示为：

recx; .if (ecx>2) {r ecx} .else {r eax}

每对大括号中也可以有多个以分号分隔的命令。

30.4.6进程和线程限定符

在很多命令前可以加上进程和线程限定符，用来指定这些命令所适用的进程和线程。表 30-1 列出了限定符的表示方法。

表 30-1 进程和线程限定符

进程限定符	含义	线程限定符	含义
|.	当前进程	~.	当前线程
|#	导致当前调试事件的进程	~#	导致当前调试事件的线程
|*	当前进程的所有进程	~*	当前进程的所有线程

|*Number*	序号为 *Number* 的进程	~*Number*	序号为 *Number* 的线程
|~[*PID*]	进程 ID 等于 *PID* 的进程	~~[*TID*]	线程 ID 等于 *TID* 的线程

例如，可以使用以下命令来显示 0 号线程的寄存器和栈回溯，尽管当前线程是 1 号线程：

0:001> ~0r; ~0k;

以上两条命令可以简写为如下形式:

0:001>~0e r; k;

注意这里，如果没有 e，那么 k 命令便是显示当前线程的栈回溯。

利用~*可以对当前进程的所有线程执行一系列命令，比如以下命令对每个线程分别执行 r 和 k 命令：

0:001> ~*e r; k;

30.4.7记录到文件

可以把输入的命令和命令的执行结果记录一个文本文件中，称为 Log File（日志文件）。可以使用 Edit 菜单的 Open/Close Log File 功能来启用和关闭日志文件，也可以是用.logopen、.logclose、.logfile 三个元命令来打开、关闭和显示日志文件。

本节我们介绍了使用 WinDBG 命令的基本要领，和如何设计比较复杂的命令。我们将在第 13 节介绍 WinDBG 命令程序时讨论如何编写更复杂的命令。

30.5 建立调试会话

建立调试会话是开始调试的必须步骤。在调试会话建立以前，除了少数选项设置和用于建立调试会话的命令可以执行外，其它大多数命令都是被禁止的。只有建立调试会话后，WinDBG 才允许执行其它命令。本节我们将讨论与各种不同调试目标建立调试会话的方法。

30.5.1附加到已经运行的进程

有以下几种方法可以把调试器附加到已经运行的进程。

- 使用 WinDBG 的 File 菜单中的 Attach to a Process 命令，或者按 F6 热键，然后在进程列表中选择要附件的进程。
- 将 WinDBG 设置为 JIT 调试器，这样当应用程序崩溃时，在应用程序错误对话框中选择 Debug 系统便会启动 WinDBG 并将其附加到这个进程。详细情况请参见第 11 章关于应用程序错误和 JIT 调试器的讨论。
- 启动 WinDBG 时通过-p 开关指定要附加的进程 ID，让 WinDBG 启动后便会附加到这个进程。
- 启动 WinDBG 时通过-pn 开关指定要附加进程的程序名，让 WinDBG 启动后便会附加到这个进程。
- 使用.attach 命令，使用这种方法需要现有一个调试会话，然后才能输入这个命令，因此这种方法常用于同时调试多个目标时的情况。

当调试系统服务或者被其它程序自动启动的程序时，通常需要使用上面介绍的方法来建立调试会话。如果要调试的程序可以在 WinDBG 中启动，那么可以使用下面介绍的创建并调试新进程

的方法。

30.5.2 非入侵式调试

非入侵式调试是一种特别的调试用户态进程的方式。使用这种方式，WinDBG 与目标进程没有真正建立调试与被调试的关系，因此不可以执行控制目标程序执行的各种命令，包括单步跟踪、继续执行等。但是可以执行观察栈、内存数据等操作。非入侵式调试的好处是减小调试器对目标进程的干预，最大程度的减少黑森博格效应。

非入侵式调试只适用于附加到已经运行进程的情况。对于图形界面方式附加到一个进程，那么只要选中对话框中的 Noninvasive 复选框。对于使用命令行的情况，只要加上-pv 开关。如果使用.attach 命令，那么只要加上-v 开关。JIT 调试不支持这种方式。

因为没有建立真正的调试关系，所以使用一个 WinDBG 以非入侵方式调试一个进程时，不会影响其它调试器再附加到这个进程进行普通的调试。

因为 Windows NT 和 Windows 2000 这样的系统不支持调试器与调试目标分离（Detach），一旦建立调试关系，那么调试会话终止就会终止被调试进程，所以对于这些系统，以非入侵方式调试有时很有用。

30.5.3 创建并调试新的进程

与将调试器附加到已经运行的进程类似，创建并调试一个新的程序也有很多种方法：

- 使用 WinDBG 的 File 菜单中的 Open Executatble 命令，或者按 Ctrl+E 热键，然后使用图 30-5 所示的打开可执行文件对话框选择要调试的程序文件，并可以可选的指定程序的命令行参数和启动目录。
- 启动 WinDBG 时将要调试的程序文件做为命令行参数传递给 WinDBG。
- 在注册表的 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution 键下，创建一个以要调试程序文件名（不包括路径）命名的子键，然后在这个子键下建立一个名为 Debugger 的 REG_SZ 类型的键值，取值为 WinDBG 程序的完整路径，比如 c:\windbg\windbg.exe。有了这个设置后，再运行要调试的程序时，操作系统就会先启动 WinDBG，并把要执行的程序名和路径传递给它。
- 使用.create 命令，使用这种方法需要现有一个调试会话，然后才能输入这个命令，因此这种方法常用于同时调试多个目标时的情况。

无论使用哪种方法，被调试程序都是作为 WinDBG 的子进程而被创建的，WinDBG 在创建这个子进程时会指定要与这个程序建立调试关系。

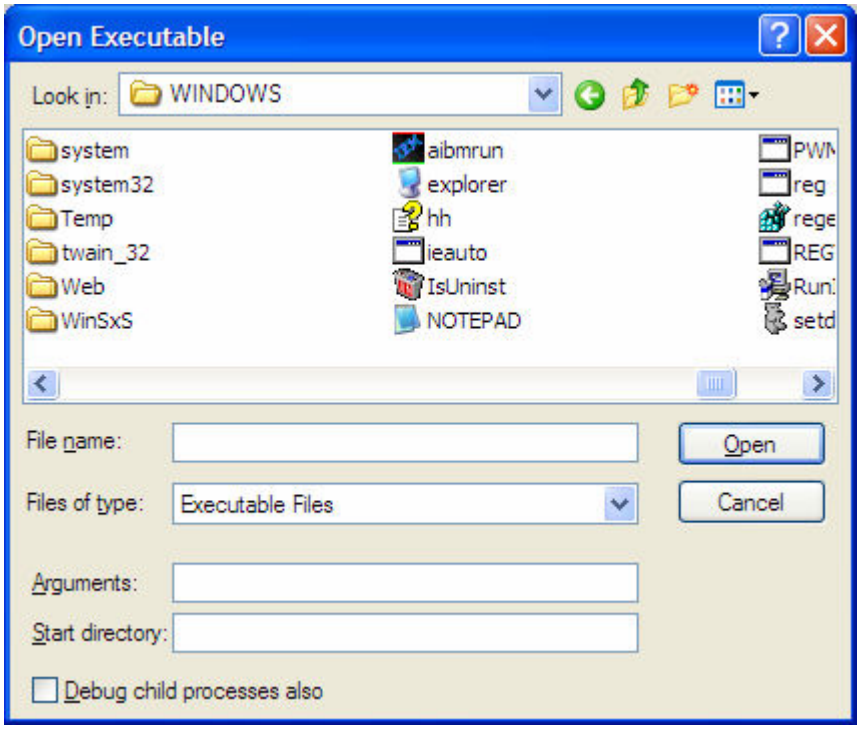


图 30-5 WinDBG 的打开可执行文件对话框

30.5.4 调试内核目标

与用户态调试相比，建立双机内核调试要复杂一些。通常包括以下几个步骤：

第一步，选择两台系统之间的通信方式，目前 WinDBG 支持串行口、1394、USB2（USB 2.0）三种方式。串行口方式需要有一根 Null-Modem 电缆，并且要求主机和目标系统都具备串行口。对于主机端，可以使用“USB 到串口”转接头而产生的串口，对于目标系统，必须具有真正的串口设备和接口。尽管串口方式的通信速度不如 1394 和 USB2，但是串口方式的优点是兼容性好，总是可以稳定的建立连接和进行调试。1394 又称为火线，使用这种方式要求两台系统都具有 1394 端口。1394 的通信速度比串口要高的多，但是这种方法的缺点是不同 1394 通信芯片的兼容性不一样，因此很多时候难以建立调试链接。USB2 方式是一种比较新的方法，它要求目标系统是 Windows Vista 或者更高的版本。因为 USB 通信具有方向性，而且一般个人电脑系统上的 USB 端口都是所谓的上游（upstream）接头，所以 USB2 方式需要有一根特殊的 USB2.0 主机到主机（Host to Host）电缆。另外，内核调试引擎对于 USB 2.0 的控制器有着特别要求，并不是每个 USB2.0 端口都满足这个要求，查阅主板或者芯片组的手册可以知道一个系统的哪个 USB 口支持内核调试。主机端没有这个要求，只有有支持 USB2.0 的接口就可以。选择好通信方式后，就可以将合适的电缆插到两台系统的相应端口上。

第二步，启用目标系统的内核调试引擎。虽然内核调试引擎是内建在 Windows 系统中的，但是缺省是禁止的。在调试前需要先启用它。

如果目标系统是 Vista 之前的版本（Windows 2000、XP 或 NT），那么应该修改 Boot.INI 文件（位于），然后将本来的启动入口（boot entry）复制一份，然后加入调试选项。以下是一个例子：

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
```

multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional" /fastdetect

【以上是文件本来的内容，下面是新增加的】

multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Windows XP Debug" /fastdetect /debug /debugport=COM1 /baudrate=115200

multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Windows XP Debug" /fastdetect /debug /debugport=1394 /channel=22

对于串行口方式，主机和目标系统所使用的 COM 端口号可以不同，比如如果目标机用的是 COM1，那么主机可以用 COM1，也可以用 COM2 等。但是它们的波特率应该是一样的。对于 1394 方式，主机和目标机的 channel 号一定要是一致的。

如果目标系统是 Windows Vista，那么应该使用 BCDEdit 工具来修改它的启动选项，其主要步骤如下：

1) 以管理员方式运行一个命令窗口（cmd）。
2) 执行 bcdedit /copy {current} /d "VistaDebug"复制一个启动入口（boot entry）。
3) 不带任何参数执行 bcdedit 列出所有启动选项，然后找到新复制的选项，将其 GUID 复制下来，然后执行命令（其中的 GUID 应该替换为实际值）：bcdedit /debug {49916baf-0e08-11db-9af4-000bdbd316a0} on
4) 使用 BCDEdit /dbgsettings 命令来观察和切换调试连接设置。例如以下命令将连接方式设置为串行口 2，波特率 115200：bcdedit /dbgsettings serial debugport:2 baudrate:115200

无论是直接修改 Boot.ini 还是使用 BCD（Boot Configuration Data），都需要重新启动系统。

第三步，在主机上启动 WinDBG 和内核调试会话。这可以有几种方法：

- 不带命令行参数直接启动 WinDBG，然后在其 File 菜单选择 Kernel Debug，或者按 Ctrl+K。然后在图 30-6 所示的内核调试对话框中选择与通信电缆和目标机器一致的类型和参数。
- 使用-k 开关和命令行参数来启动 WinDBG。例如，windbg -k com:port=Com1,baud=115200

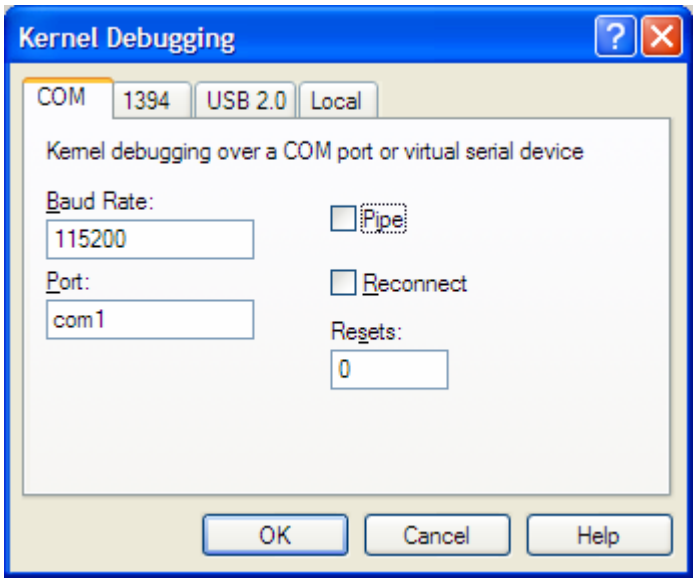


图 30-8 WinDBG 的内核调试对话框

无论是使用上面何种方法，WinDBG 都会进入图 30-9 所示的等待状态，等待来自目标系统的调试数据。

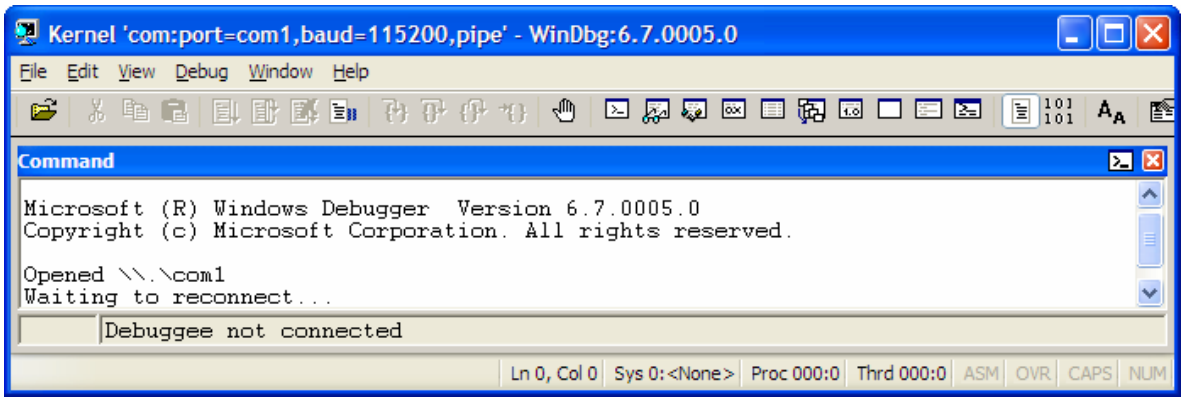


图 30-5 WinDBG 等待与目标系统建立连接

如果是先运行主机上的调试器，然后再以调试选项启动目标系统，那么目标系统在启动早期初始化内核调试引擎时会向主机上的调试器发送信息，使二者建立起调试连接。

如果是后运行调试器，那么可以在调试器上按 Ctrl+Break 来触发 WinDBG 调试器主动向目标系统发送信息，如果发送成功，二者也会开始通信并建立起调试连接。

30.5.5本地内核调试

有以下三种方式启动本地内核调试：

- 运行 WinDBG，然后选择 File 菜单中的 Kernel Debug，然后在图 30-8 所示的对话框中选择 Local。
- 使用-kl 作为命令行参数来启动 WinDBG。
- 执行.attach –k 命令，这种方式需要先有一个调试会话，所以可以用于调试多个目标的情况。

对于 Windows Vista，需要以调试选项启动当前系统才能进行本地内核调试。Windows XP 没有这个要求。Windows 2000 或者更早的 Windows 不支持本地内核调试。

30.5.6调试转储文件

WinDBG 支持以下三种方式来打开一个转储文件：

- 运行 WinDBG，然后选择 File 菜单中的 Open Crash Dump，然后选择要打开的转储文件。
- 使用命令行方式，通过 WinDBG 的-z 开关来指定要打开的转储文件。
- 使用.opendump 命令，这种方式需要先有一个调试会话，所以可以用于调试多个目标的情况。

30.5.7远程调试

WinDBG 工具包支持多种远程调试方式，上一章我们介绍了使用 DbgSrv 和 KdSrv 来作为服务器的方法。本节我们介绍使用服务器和客户端都使用调试器的方法。这里的调试器是指 WinDBG 工具包中的任一调试器，我们只以 WinDBG 为例。

首先服务端（被调试程序运行的系统）和客户端（调试者进行调试的系统）都应该安装相同

版本的 WinDBG 工具包。主机和客户机直接应该有网络连接（局域网或者互联网）或者串行口连接。

在服务端可以以下两种方式之一启动服务器：

- 命令行方式，使用-server 开关来指定连接方式，例如以下命令创建了一个使用命名管道（名称为 advdbg）方式通信的服务器：windbg –server npipe:pipe=advdbg
- 启动 WinDBG 和调试会话，然后执行.server 命令。这种方式需要先建立一个调试会话。比如执行.server npipe:pipe=advdbg，那么 WinDBG 会显示：

  0:001> .server npipe:pipe=advdbg

  Server started. Client can connect with any of these command lines

  0: <debugger> -remote npipe:Pipe=advdbg,Server=SHXPLYZHAN31B

类似的，在客户端也有两种方式：

- 命令行方式，使用-remote 开关来指定连接方式，例如可以使用以下命令与上面创建的服务器建立连接：windbg -remote npipe:server=SHXPLYZHAN31B,pipe=advdbg
- 直接启动 WinDBG，然后选择 File 菜单的 Connect to Remote Session（Ctrl+R），而后在图 30-9 所示的对话框中直接输入连接字符串（npipe:Pipe=advdbg,Server=SHXPLYZHAN31B）或者点击 Browse 按钮，然后输入服务器机器名，然后选择 WinDBG 搜索到的服务器。

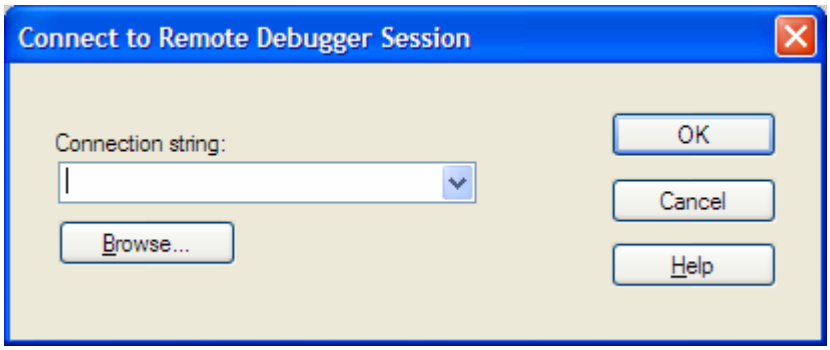


图 30-5 WinDBG 的连接到远程调试会话对话框

成功连接后，命令信息区会显示类似如下的信息：

SHXPLYZHAN31B\yzhan31 (npipe advdbg) connected at Thu Jul 05 18:24:01 2007

之后可以在客户端或者服务端的 WinDBG 中执行各种调试命令，执行结果会同时显示在两个调试器中。

30.6 终止调试会话

WinDBG 提供了多种方式来终止调试会话，本节将分别做简单介绍。

30.6.1 停止调试

当 WinDBG 处于命令模式时，在 WinDBG 的 Debug 下拉菜单中选择 Stop Debugging 可以终止当前调试器中的调试会话，使调试器恢复到赋闲（dormant）状态。如果是在调试活动的用户态目标，那么这一操作也会导致调试目标被终止。如果是在调试活动的内核目标，那么目标系统仍会保持被中断到调试器的状态，还可以重新与其建立连接。

也可以使用标准命令 q 来停止调试，除了具有与这个菜单操作相同的效果外，q 命令还会使

WinDBG 程序退出。

30.6.2分离调试目标

可以使用分离调试目标功能来结束调试器与调试目标的调试关系。具体有两种操作方式，一种是使用 Debug 菜单中的 Detach Debuggee 命令，另一种是输入.detach 命令。

当调试用户态的活动目标时，这一命令会保持目标进程继续运行，与选择 Stop Debugging 不同。但因为这一功能依赖于 Windows XP 才引入的操作系统支持（参见 DebugSetProcessKillOnExit API），所以要求目标系统为 Windows XP 或者更高版本。

当只调试一个目标时，执行这个命令后 WinDBG 就会进入到赋闲状态，如果调试多个目标，那么可以继续调试其它目标。

当调试内核目标时，执行分离操作和上面的 Stop Debugging 效果是一样的。

30.6.3抛弃被调试进程

使用分离调试目标命令时，系统会修改目标进程的进程属性，使其脱离被调试状态成为一个普通的进程。如果想让其保持被调试状态，那么可以使用.abandon 命令来抛弃被调试进程。

0:000> .abandon

Abandoned

这个命令执行后，调试器会恢复到无调试目标状态，命令提示符变为 No Target。被调试进程仍处于挂起状态。简单来说，这个命令只是把调试器的状态恢复了，但是并没有把被调试进程恢复到调试前的状态。

这种情况下，可以使用另一个调试器再附加到被调试进程，但需要在启动调试器的命令行中指定-pe 开关，比如：

Windbg –pe –p 2272

其中 2272 是处于被抛弃状态的进程的进程 ID。

如果没有-pe 开关，那么调试器会附加失败，并报告 DebugPort 不为空。有了-pe 开关后，WinDBG 会知道这是重新附加，不会报告错误，调试器引擎产生一个“人工合成”的异常，使调试器进入命令模式，使调试可以继续。与第一次附加到一个进程不同，重新附加不会收到调试子系统所发送的关于现有模块和线程虚拟历史调试事件。

30.6.4杀死被调试进程

使用.kill 命令可以杀死当前的被调试进程。

0:000> .kill

Terminated. Exit thread and process events will occur.

事实上，这个命令会调用操作系统的 TerminateProcess API 来终止当前线程，其执行过程如下：

0:001> kn

 # ChildEBP RetAddr

00 00f0e6c0 0229aa44 kernel32!TerminateProcess

01 00f0e6d4 020f3e05 dbgeng!LiveUserDebugServices::TerminateProcess+0x14

02 00f0e6f8 0219250e dbgeng!LiveUserTargetInfo::TerminateProcess+0xc5

03 00f0e77c 020f8799 dbgeng!ProcessInfo::Separate+0x2ee
04 00f0e8e4 0210577f dbgeng!ParseSeparateCurrentProcess+0x2e9
05 00f0e8f8 0218758e dbgeng!DotCommand+0x3f
06 00f0e9d8 021889a9 dbgeng!ProcessCommands+0x4be
……

再执行 g 命令，WinDBG 会收到被终止进程的线程退出事件。如果只是在调试一个进程，那么 WinDBG 会结束当前调试会话恢复到赋闲状态。如果同时调试多个进程，那么调试会话不会终止，还可以继续调试其它进程。

30.6.5 调试器异常终止

如果直接关闭调试器进程，那么它所建立的调试会话也会终止，调试会话中如果包含活动的调试目标进程，那么这些进程也会随之终止。

30.6.6 重新运行调试程序

当调试一个调试器创建（Spawn）的进程时，可以使用如下方法之一让这个进程退出，然后再重新开始运行。

执行.restart 命令。

在 Debug 菜单中选择 Restart 或者按 Ctrl+Shift+F5。

如果调试器是附加到一个已经运行的进程，那么 WinDBG 会提示如下信息：

0:000> .restart
Process attaches cannot be restarted. If you want to
restart the process, use !peb to get what command line
to use and other initialization information.

当进行内核态调试时，Restart 命令相当于重新启动调试器然后再建立调试连接。如果要让目标系统重新启动，那么可以使用.reboot 命令。

30.6.7 调试器僵死

如果调试器因为某种原因僵死，但是调试任务尚未完成，此时可以使用上面介绍的-pe 开关启动一个新的调试器进程，然后再终止僵死的进程。

30.7 理解上下文

Windows 是个典型的多任务操作系统，在同一时刻，系统内可能运行着多个登录会话（Logon Session），每个会话中运行着多个进程，每个进程可能有一或多个线程。对于这样的系统，我们在执行操作或者陈述现象时都是基于一定上下文的。在使用 WinDBG 调试时，很多调试命令也都是针对当前当前上下文（context）的。在理解命令的执行结果时，我们也必须结合它的上下文来分析。

根据 Windows 操作系统的特征，WinDBG 定义了如下几种上下文：会话上下文、进程上下文、寄存器上下文、局部（变量）上下文。下面我们分别进行介绍每种上下文的含义和切换方法。

30.7.1会话上下文

从 Windows XP 开始，Windows 支持同时有多个登录会话，每个会话可以有自己的输入、输出设备和桌面，并拥有很多个进程在其中运行。

所谓会话上下文（Session Context）就是当前操作或者陈述所基于的登录会话语境。例如，对于会话 A 的所有进程来说，会话 A 的状态和属性便是它们的会话上下文。使用!session 扩展命令可以显示或者切换会话上下文（Session Context）。

因为 Windows Vista 使用所谓的会话隔离（Session Isolation）方法让所有系统服务运行在会话 0 以增强系统服务的安全性，所以在调试 Vista 目标时，使用!session 命令观察通常可以看到两个会话。而对于 Windows XP 目标，通常只有一个会话，除非使用所谓的快速用户切换（FUS）或者从远程登录。

以下是调试 Vista 目标时的一个例子。

```
0: kd> !session
Sessions on machine: 2
Valid Sessions: 0 1
Current Session 0
```

使用-s 可以设置当前的会话上下文：

```
0: kd> !session -s 1
Sessions on machine: 2
Implicit process is now 848178d8
WARNING: .cache forcedecodeuser is not enabled
Using session 1
```

改变会话后，隐含进程通常也会变化，因此以前缓存的虚拟内存数据不再有效。所以应该将缓存选项中设置强制将虚拟内存翻译为物理内存，即使用 forcedecodeuser 或者 forcedecodeptes 选项。上面命令结果中的警告告诉我们目前还没有启用这个缓存选项。

!sprocess 扩展命令用来列出某个会话的所有进程，比如以下列出的是会话 1 的进程：

```
0: kd> !sprocess 1
Dumping Session 1

_MM_SESSION_SPACE 8642b000
_MMSESSION              8642bd00
PROCESS 848178d8    SessionId: 1    Cid: 01ec       Peb: 7ffdf000    ParentCid: 01d8
        DirBase: 26340000    ObjectTable: 94c75ad0    HandleCount: 205.
        Image: csrss.exe
...
```

以下是会话 0 的进程：

```
0: kd> !sprocess 0
Dumping Session 0

_MM_SESSION_SPACE 8643f000
_MMSESSION              8643fd00
PROCESS 84843d90    SessionId: 0    Cid: 01b0       Peb: 7ffd5000    ParentCid: 01a4
```

DirBase: 27480000 ObjectTable: 8ca57b30 HandleCount: 449.
Image: csrss.exe

…

可见每个会话都有自己的 Windows 子系统服务器进程（CSRSS）。另外，会话管理器进程本身不属于任何一个会话。

在当前的 WinDBG 中，第一次使用!session 命令时，会显示读取当前会话错误：

1: kd> !session
Sessions on machine: 2
Valid Sessions: 0 1
Error in reading current session

在使用!session –s 将某个会话设置为当前会话后，就不再有上面的问题了。

会话上下文只有在内核调试时才有意义，所以以上扩展命令也只有在调试内核目标时才能使用。

30.7.2进程上下文

所谓进程上下文就是指当前操作或者陈述所基于的进程语境。我们知道，Windows 系统中的内核空间是单一的，也就是说所有进程的进程空间中的内核部分是共享的。但是，用户部分是独立的。例如，在典型的 32 位 Windows 系统中，每个进程的进程空间是 4GB，高 2GB 是内核空间，低 2GB 是用户空间。对于高 2GB 的内存地址，它们的指向和取值是相同的。对于低 2GB 空间，其指向和取值是相对于进程的。

在内核调试时，通常只有一个进程的用户空间是可见的。或者说所有用户空间地址和内容都是属于这个进程的。要观察另一个进程的用户空间，就要将进程上下文切换到另一个进程。WinDBG 的.process 命令便是为了满足这个需求而设计的。

例如，以下命令将进程 83f7fc78 设置为隐含进程：

1: kd> .process 83f7fc78
Implicit process is now 83f7fc78

其中 83f7fc78 是进程的 EPROCESS 结构的地址。使用!process 0 0 命令可以列出系统中的所有进程的基本信息，其中包含 EPROCESS 结构的地址。

一个有关的命令是.context，它用来设置或者显示用来翻译用户态地址的页目录基地址（Base of Page Directory）。

例如以下命令显示当前所使用的页目录基地址：

1: kd> .context
User-mode page directory base is 671125c0

页目录基地址是进程的一个重要属性，因此使用.process 设置进程上下文时，它会自动设置合适的页目录基地址。

对于 x86 系统，cr3 寄存器用来存放页目录基地址，而且每个进程的用户空间都是基于一个页目录基地址的，因此.context 命令和.process 命令的效果几乎是一样的。对于安腾系统，一个进程可能使用多个页目录基地址，这时使用.process 命令切换更高效。

因为调试用户态目标时，所有虚拟地址都是相对于当前进程的，不需要切换进程上下文，所以.process 和.context 命令都只能用在内核态调试时。

当在一个调试会话中调试多个用户态目标时，应该使用|<进程号> s 命令来切换。

30.7.3 寄存器上下文

所谓寄存器上下文（Register Context）就是寄存器取值所基于的语境。因为一个 CPU 只有一套寄存器，所以当它轮番执行系统中的多个任务（线程）时，它的寄存器中存放的是当前正在执行线程的寄存器值。对于没有执行的线程，它的寄存器值被保存在内存中，当 CPU 要执行这个任务时，这些寄存器值被从内存加载到物理寄存器中。

在调试时，当我们观察一个线程的寄存器时，这个线程通常是处于挂起状态的，所以我们看到的寄存器都是保存在内存中的寄存器值，而不是此时物理寄存器的值。当我们修改寄存器时，也是修改保存在内存中的寄存器值。

系统在以下几种情况下会将一个线程的寄存器保存到内存中，称为上下文记录（Context Record）。

- 当系统做线程切换时，系统会将要挂起线程的寄存器取值保存起来，这个记录常被称为线程上下文。
- 当发生中断或者异常时，系统会将当时的寄存器取值保存起来，这个记录常被称为异常上下文。

使用.thread 命令可以显示或者设置寄存器上下文所针对的线程。例如以下命令显示当前的隐含线程：

```
1: kd> .thread
Implicit thread is now 83f81950
```

使用!process <所属进程的 EPROCESS 结构地址> f 可以列出一个进程的所有线程，包括的每个线程的 ETHREAD 结构，把 ETHREAD 结构的地址作为.thread 命令的参数便可以将这个线程的上下文设置为新的线程上下文：

```
1: kd> .thread 84018d78
Implicit thread is now 84018d78
```

这时再使用观察寄存器和栈命令，WinDBG 会提示命令结果是针对刚刚设置的上下文的：

```
1: kd> r
Last set context:
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
…
1: kd> kv
  *** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr    Args to Child
9ce23be0 818ac9cf 84018d78 818f4820 84018e00 nt!KiSwapContext+0x26 …
```

输入.cxr 或者输入不带参数的.thread 命令可以将线程上下文恢复成以前的情况。

当调试用户态的转储文件时，可以使用.ecxr 命令将转储文件中保存的异常上下文设置为寄存器上下文。

30.7.4 局部（变量）上下文

所谓局部上下文（Local Context），就是指局部变量所基于的语境。局部变量是指定义在函数内部的变量，这些变量的含义与当前的执行位置密切相关。在调试时，调试器缺省显示的是当前函数所对应的局部上下文。因为当前函数和局部变量都是与栈帧密切相关的，所以 WinDBG 调试器通常使用栈帧号来代表局部上下文。

下面通过一个例子来说名如何切换局部上下文，以下是要观察线程（UefWin32 程序）的栈帧列表：

```
0:000> kn
 # ChildEBP RetAddr
00 0012fdb4 7e418724 UefWIn32!WndProc+0xe1 [C:\...\UefWin32.cpp @ 151]
01 0012fde0 7e418806 USER32!InternalCallWinProc+0x28
02 0012fe48 7e4189bd USER32!UserCallWinProcCheckWow+0x150
03 0012fea8 7e4196b7 USER32!DispatchMessageWorker+0x306
04 0012feb8 00401127 USER32!DispatchMessageA+0xf
05 0012ff30 004018e3 UefWIn32!WinMain+0xe7 [C:\...\UefWin32.cpp @ 48]
06 0012ffc0 7c816ff7 UefWIn32!WinMainCRTStartup+0x113 [crt0.c @ 198]
07 0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

使用不带任何参数的.frame 命令可观察当前的局部上下文：

```
0:000> .frame
00 0012fdb4 7e418724 UefWIn32!WndProc+0xe1 [C:\...\UefWin32.cpp @ 151]
```

这说明当前的函数是 WndProc 函数，此时使用 dv 命令可以显示这个函数的参数和局部变量：

```
0:000> dv
0:000> dv
             rt = struct tagRECT
           hWnd = 0x002d0500
        message = 0x111
         wParam = 0x8003
         lParam = 0
        wmEvent = 0
            hdc = 0xcccccccc
             ps = struct tagPAINTSTRUCT
        szHello = char [100] "Hello World!"
           wmId = 32771
```

使用.frame 5 可以将局部上下文切换到栈帧 5，也就是 WinMain 函数：

```
0:000> .frame 5
05 0012ff30 004018b3 UefWIn32!WinMain+0xf3 [C:\...\UefWin32.cpp @ 48]
```

此时可以使用 dv 命令显示 WinMain 函数的参数和局部变量。

```
0:000> dv
      hInstance = 0x00400000
  hPrevInstance = 0x00000000
      lpCmdLine = 0x00141f0d ""
       nCmdShow = 10
    hAccelTable = 0x055d087d
            msg = struct tagMSG
```

值得说明的一点是，因为 VC 编译器缺省将类型符号放在 VCx0.PDB 文件中，而 WinDBG 没有很好的处理这样的类型符号，所以在显示局部变量时，会显示很多 no type information 错误。解决的方法是将符号格式设置为 C7 Compatable（Settings>C++ > General > Debug Info）。上面的结果就是使用这种格式显示的。

最后要说明的是以上介绍的各种上下文的范围是从大到小的，改变大范围的上下文必然会影响小范围的上下文。例如，线程上下文切换后，那么局部变量上下文也一定会变化了。

30.8 调试符号

在第 25 章中，我们详细的介绍了调试符号的概念、种类、产生过程和存储方式。本节我们将讨论如何在调试中合理的使用调试符号，介绍 WinDBG 加载调试符号的有关内容，以及如何设置调试符号有关的选项，以及如何解决常见的调试符号发生错误的情况。

30.8.1 重要意义

调试符号（Debug Symbols）是调试器工作的重要依据，保证调试符号的准确对于调试器的正常工作非常重要。如果缺少调试符号或者调试符号不匹配，那么调试器就可能显示出不准确的结果，有可能导致错误的结论。

为了使大家对这一点有深刻的认识，我们给出一个简单的例子。使用 WinDbg（尚未设置符号路径）附加到一个记事本进程上，然后使用~0 s 切换到 0 号线程，再输入 k 命令显示栈回溯信息，其结果如下：

```
0:000> k
*** ERROR: Module load completed but symbols could not be loaded for C:\WINDOWS\system32\notepad.exe
ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
0007fed8 01002a1b ntdll!KiFastSystemCallRet
0007ff1c 01007511 notepad+0x2a1b
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\WINDOWS\system32\kernel32.dll -
0007ffc0 7c816fd7 notepad+0x7511
0007fff0 00000000 kernel32!RegisterWaitForInputIdle+0x49
```

在上面的结果中，WinDbg 报告了两个错误和一条警告，都与符号有关。第一个错误是告诉我们未能为 notepad.exe 加载符号。接下来的警告告诉我们因为缺少符号文件提供栈展开信息，所以其下各桢的信息可能是错误的。这个警告决不是空穴来风，看到了这样的警告，确实需要提高警惕，应该以怀疑的眼光观察其后的内容。例如最下面一行显示的函数名是 kernel32.dll 中的 RegisterWaitForInputIdle 函数，意思是这个函数是当前线程的起始函数。看这个函数名它怎么会是线程的起始函数呢？键入.symfix c:\symbols 命令设置符号文件的搜索路径（稍后再详细介绍这条命令），然后输入.reload 加载符号，再次输入 k 命令，这次看到的结果是：

```
0:000> k
ChildEBP RetAddr
0007feb8 7e4191ae ntdll!KiFastSystemCallRet
0007fed8 01002a1b USER32!NtUserGetMessage+0xc
0007ff1c 01007511 notepad!WinMain+0xe5
0007ffc0 7c816fd7 notepad!WinMainCRTStartup+0x174
0007fff0 00000000 kernel32!BaseProcessStart+0x23
```

这次的结果没有任何错误和警告而且看到的函数调用关系很符合实际，BaseProcessStart 调用编译

器插入的程序启动函数 WinMainCRTStartup，后者再调用程序的入口函数 WinMain。

30.8.2符号搜索路径

可以指定一个目录列表，让调试器从这些目录中搜索符号文件。这个目录列表被称为符号搜索路径，有时也简称符号路径（symbol path）。

目录列表中可以指定两类位置，一类是普通的磁盘目录，另一类是符号服务器。多个位置之间使用分号分隔。比如以下是一个典型的符号路径：

SRV*d:\symbols*http://msdl.microsoft.com/download/symbols;c:\work\symview\debug;

第一个分号后面定义的是一个本地的目录，前面定义的是符号服务器，我们稍后再详细介绍。

可以有几种方法来设置符号路径：

- 设置环境变量_NT_SYMBOL_PATH 和_NT_ALT_SYMBOL_PATH。
- 启动调试器（WinDBG）时，在命令行参数中通过-y 开关来定义。
- 使用.sympath 命令可以增加、修改、或者显示符号路径。如执行 sympath + c:\folder2 便将 c:\folder2 目录加入到搜索路径中。
- 使用.symfix 命令可以设置符号服务器（详见下文）。
- 使用 WinDBG 的 GUI，通过 File>Symbol File Path 菜单打开 Symbol Search Path 对话框，然后通过图形界面进行设置。

执行不带任何参数的.sympath 命令可以显示当前的符号路径。

30.8.3符号服务器

无论是用户态调试，还是内核态调试，通常涉及到很多个模块。因为不同的模块可能属于不同的开发部门或者公司，比如很多 Windows 系统模块是微软开发的，而且不同的模块可能属于不同的版本，所以要为每个模块都找到正确的符号便不再是一件简单的事。

解决以上问题的一个有效方法是使用符号服务器（Symbol Server）。简单说来，符号服务器就是用来存储调试符号文件的一个大仓库，调试器可以从仓库中读取指定特征（名称、版本等）的符号文件。

图 30-1 画出了 WinDBG 的符号服务器的简单架构图。图中左侧是使用 WinDBG 调试器的工作机，右侧是符号服务器。

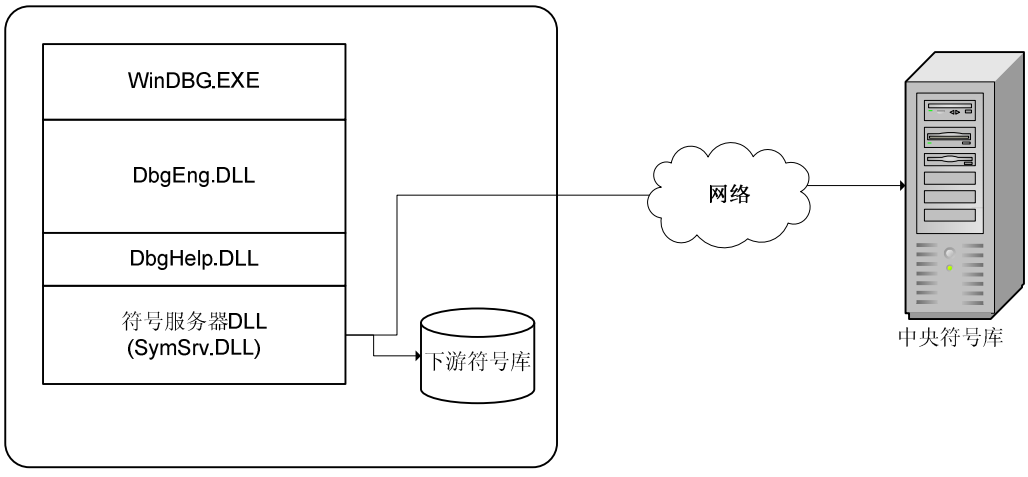


图 30-1 符号服务器架构示意图

在工作机一端，我们画出了 WinDBG 进程中与访问符号有关的各个模块。其中，DbgHelp.DLL 是 Windows 操作系统的符号库模块，WinDBG 通过它读取和解析调试符号。符号服务器 DLL 负责从符号服务器查找、下载和管理符号文件，是 DbgHelp 访问符号服务器的唯一接口。

为了避免重复下载以前已经使用和下载过的符号文件，符号服务器 DLL 通常将下载的文件保存在本地的一个文件夹中，这个文件夹是使用与符号服务器上相似的方式来组织符号文件的，为了相区分，本地的部分叫下游符号仓库（Downstream Store），符号服务器上的叫中央符号仓库（Centralized Store）。当符号服务器 DLL 每次接到 DbgHelp 的请求需要某个符号文件时，符号服务器 DLL 会先在下游仓库中寻找，如果寻找不到才到远程的中央符号仓库去寻找。

DbgHelp 通过所谓的符号服务器 API（Symbol Server API）来调用符号服务器 DLL。符号服务器 DLL 输出这些 API 供 DbgHelp 来调用。WinDBG 开发工具包中的 DbgHelp 帮助文件（sdk\help\dbghelp.chm）详细描述了符号服务器 API 的函数原型和功能。

根据符号服务器 API 的定义，用户也可以编写符号服务器 DLL 来实现自己的符号服务器。只要这个 DLL 正确的实现和输出的符号服务器 API 所定义的函数。

WinDBG 工具包中包含了一个符号服务器 DLL，名为 SymSrv.DLL。

可以通过以下格式来向符号搜索路径中加入符号服务器：

symsrv**ServerDLL**[*DownstreamStore**]*ServerPath*

其中 *ServerDLL* 是符号服务器 DLL 的文件名称，*DownstreamStore* 是下游符号库的位置，*ServerPath* 是符号服务器的 URL 或共享路径，例如以下是两个有效的定义：

symsrv*symsrv.dll*\\mybuilds\mysymbols

symsrv*symsrv.dll*\\localserver\myshare\mycache*http://www.somecompany.com/manysymbols

第一个没有指定下游符号库。

因为大多用户都是使用 WinDBG 工具包中的 SymSrv.DLL 作为符号服务器 DLL，所以可以使用以下简化形式：

srv*[*DownstreamStore**]*ServerPath*

也就是，其中的 srv 相当于 symsrv*SymSrv.DLL*。

30.8.4符号文件的加载过程

下面通过几个例子来说明符号文件的加载过程。清单 30-1 列出了在 WinDBG 调试器中（用户态调试）键入 ld kernel32 命令后，调试器工作线程通过 SymSrv 模块向符号服务器请求符号文件的过程。

清单 30-1 从符号服务器获取符号文件的执行过程

```
0:001> kn 30
 # ChildEBP RetAddr
00 00f1b158 01d1182a WININET!HttpSendRequestW
01 00f1b180 01d11528 symsrv!StoreWinInet::request+0x2a
02 00f1b1b4 01d10ebc symsrv!StoreWinInet::fileinfo+0x18
03 00f1b1c4 01d11023 symsrv!StoreWinInet::open+0x7c
04 00f1b404 01d05127 symsrv!StoreWinInet::find+0xd3
05 00f1b4f8 01d06277 symsrv!cascade+0x87
06 00f1ba48 01d06087 symsrv!SymbolServerByIndexW+0x127
07 00f1bc78 0302dfee symsrv!SymbolServerW+0x77
08 00f1c0b8 03018e7d dbghelp!symsrvGetFile+0x12e
09 00f1cda0 03019ee7 dbghelp!diaLocatePdb+0x33d
```

0a 00f1d01c 030415fe dbghelp!diaGetPdb+0x207
0b 00f1d240 0303fa35 dbghelp!GetDebugData+0x2be
0c 00f1d6e8 0303fcf4 dbghelp!modload+0x305
0d 00f1db68 03037fdd dbghelp!LoadModule+0xb4
0e 00f1dbd4 0303815a dbghelp!SymLoadModuleEx+0x7d
0f 00f1dc00 02185a18 dbghelp!SymLoadModule64+0x2a
10 00f1e900 02187ca8 dbgeng!ParseLoadModules+0x188
11 00f1e9d8 021889a9 dbgeng!ProcessCommands+0xbd8
12 00f1ea1c 020cbec9 dbgeng!ProcessCommandsAndCatch+0x49
13 00f1eeb4 020cc12a dbgeng!Execute+0x2b9
14 00f1eee4 01028553 dbgeng!DebugClient::ExecuteWide+0x6a
15 00f1ef8c 01028a43 windbg!ProcessCommand+0x143
16 00f1ffa0 0102ad06 windbg!ProcessEngineCommands+0xa3
17 00f1ffb4 7c80b6a3 windbg!EngineLoop+0x366
18 00f1ffec 00000000 kernel32!BaseThreadStart+0x37

其中，从 16 号到 10 号栈帧是分发命令的过程。0f 号是调用 DbgHelp 库的 SymLoadModuleEx 函数，后者调用 LoadModule，之后调用 diaGetPdb 发起读取 PDB 文件的过程。diaLocatePdb 函数是搜索 PDB 文件的一个主要函数。当它在符号搜索路径中指定的普通位置中找不到符号文件时，便会调用 symsrvGetFile 函数来使用符号服务器方式。

接下来 symsrvGetFile 函数调用符号搜索路径中定义的符号服务器 DLL 中的 SymbolServer 函数。SymbolServer 函数符号服务器 API 中定义的一个重要函数，它的作用就是向符号服务器请求指定的符号文件，并返回访问这个文件的完整路径。SymbolServer 函数的典型实现是，如果所需要的符号文件已经在下游库中，那么便返回它的全路径，不然的话便向远程查询，如果在远程查找到，那么便将其下载到下游库，然后返回其在下游库的全路径。

SymbolServer 函数的原型如下：

```
BOOL CALLBACK SymbolServer(
   LPCSTR params,   LPCSTR filename,
   PVOID id,   DWORD two,   DWORD three,   LPSTR path);
```

第 1 个参数 params 是符号服务器的设置信息，其典型值为：

```
0:001> du 00f1bc98
00f1bc98   "d:\symbols*http://msdl.microsoft"
00f1bcd8   ".com/download/symbols"
```

第 2 个参数 filename 即符号文件名，如 kernel32.pdb。最后一个参数 path 用来存放符号文件的完整路径。

```
0:001> du 00f1c120
00f1c120   "32.pdb\006D2240474D414087FF801C6"
00f1c160   "4935DDD2\kernel32.pdb"
```

第 3～5 三个参数用来定义符号文件的版本特征。它们的用法根据第 2 个参数中的指定的文件类型而定，如表 30-4 所示。

表 30-4 SymbolServer 函数用来指定文件版本的参数

要加载的文件后缀	参数 id	参数 two	参数 three
.dbg	PE 文件头中定义的映像时间戳（TimeDateStamp）	PE 文件头中定义的映像文件大小（SizeOfImage）	没有使用，为 0

PE 文件（.exe/.dll）	同上	同上	同上
.pdb	PDB 签名	PDB 年龄（Age）	没有使用，为 0

SymbolServer 首先根据上面的特征调用 SymbolServerGetIndexStringW 函数生成一个索引串。比如，以下是笔者机器上的 kernel32.dll 所对应 pdb 文件的索引串：

0:001> du 00f1ba60

00f1ba60 "006D2240474D414087FF801C64935DDD"

00f1baa0 "2"

其中，006D2240474D414087FF801C64935DDD 是 GUID 签名，即 {006D2240-474D-4140-87FF-801C64935DDD}，2 是 PDB 年龄。

接下来 SymbolServer 函数调用 SymbolServerByIndexW 函数取符合指定索引串的符号文件。后者会调用一个名为 cascade 的函数。

Cascade 函数先使用 StoreUNC 类来在下游库中查找。如果找到了便返回完整的路径。

00f1b1c8 01d0bc3f symsrv!StoreUNC::filestatus

00f1b3f4 01d0b33b symsrv!StoreUNC::status+0x6f

00f1b404 01d05127 symsrv!StoreUNC::find+0x2b

00f1b4f8 01d06277 symsrv!cascade+0x87

00f1ba48 01d06087 symsrv!SymbolServerByIndexW+0x127

00f1bc78 0302dfee symsrv!SymbolServerW+0x77

如果在下游库中没有找到匹配的符号文件，那么 cascade 便使用 StoreWinInet 搜索远程的中央符号库，即栈帧 04~00 所示的情况。

以下是 SymbolServer 函数返回时它的 path 参数中所存放的内容：

0:001> du 00f1c100

00f1c100 "c:\dstore\kernel32.pdb\006D22404"

00f1c140 "74D414087FF801C64935DDD2\kernel3"

00f1c180 "2.pdb"

其中 006D2240474D414087FF801C64935DDD2 就是索引串。

使用.reload 命令可以重新加载所有或者指定模块的符号文件。以下是主要的执行步骤：

00f1d488 030380b5 dbghelp!LoadModule+0x501

00f1d4f0 02190a29 dbghelp!SymLoadModuleExW+0x65

00f1e440 022215ed dbgeng!ProcessInfo::AddImage+0xbf9

00f1e8d0 02102822 dbgeng!TargetInfo::Reload+0x1cbd

00f1e8e4 0210577f dbgeng!DotReload+0x22

00f1e8f8 0218758e dbgeng!DotCommand+0x3f

00f1e9d8 021889a9 dbgeng!ProcessCommands+0x4be

因为是元命令，所以 ProcessCommands 先分发给 DotCommand 函数，后者再调用 DotReload，DotReload 交给 TargetInfo 类的 Reload 方法。Reload 方法枚举进程中的各个模块，对于每个模块调用 ProcessInfo 类的 AddImage 方法将其加入到进程信息中。AddImage 方法会调用调试支持库（dbghelp）的 SymLoadModuleExW 方法来加载这个模块的信息，包括符号文件。接下来的过程与清单 30-1 所示的情况非常类似。

除了使用 ld 和.reload 命令直接加载符号文件，某些使用符号的命令也可以触发调试器来加载符号，比如栈回溯命令（k*）和反汇编命令等。

值得说明的是，因为 WinDBG 缺省使用所谓的懒惰式符号加载策略，所以当它接收到模块加载事件时，它通常是不会加载符号的。对于这样的模块，在显示其符号状态时，其信息通常为

deferred（延迟加载）。

30.8.5观察模块信息

可以使用以下方法之一来观察模块信息，包括调试符号情况。

- 使用 lm 命令。
- 使用!lmi 扩展命令。
- 使用 WinDBG 图形界面的模块列表对话框（Debug>Modules）来观察。

我们先介绍 lm 命令。如果不指定任何参数，那么 lm 命令显示一个简单的列表：

```
0:001> lm
start    end        module name
01000000 01093000   windbg     (pdb symbols)          d:\...\windbg.pdb
01400000 015c6000   ext        (deferred)
...
```

其中 start 列和 end 列分别是该模块在进程空间中的起始地址和终止地址。Module name 列是模块名称，接下来的一类是符号状态（symbol status），表 30-1 列出了符号状态列中可能出现的状态信息和它们的含义。如果已经加载符号文件，那么第 4 列是符号文件的完整名称。

表 30-1 符号状态

缩写	含义
deferred	模块已经加载，但是调试器还有试图为其加载符号，会在需要时尝试。
#	符号文件和执行映像文件存在不匹配，比如时间戳、校验和等。
T	时间戳缺失，不可访问，或者等于 0。
C	校验和缺失，不可访问，或者等于 0。
DIA	符号文件是通过 DIA（Debug Interface Access ）方式加载的。
Export	没有发现实际的符号文件，使用映像文件的输出信息（如 DLL 的 Export）作为符号。
M	符号文件和执行映像文件存在不匹配，要么是时间戳或者校验和。但是仍然加载了这样的符号文件。
PERF	执行文件包含性能优化代码，对地址进行简单加减运行可能产生错误结果。
Stripped	调试信息是从映像文件中抽取出来的。
PDB	符号文件是.PDB 格式。
COFF	符号是 COFF 格式（Common Object File Format）。

对于 PDB 格式又分为私有 PDB 文件和公共 PDB 文件，前者通常是编译调试版本时产生的，包含的调试信息更多。后者通常是编译发布版本产生的，很多类型信息已经去除了。

```
0:000> lm
start    end        module name
00400000 0041a000   dbgee    C (private pdb symbols)  C:\...\dbgee.pdb
10200000 10320000   MSVCR80D   (deferred)
77c10000 77c68000   msvcrt     (deferred)
7c800000 7c8f5000   kernel32   (pdb symbols)          d:\...\kernel32.pdb
```

上面的 dbgee.pdb 是私有 PDB 文件，kernel32.pdb 是公共 PDB 文件。

如果要为每个模块显示更丰富的信息，那么可以使用 v 选项：

```
0:001> lm v
```

start end module name
01000000 01093000 windbg (pdb symbols) d:\...\windbg.pdb
Loaded symbol image file: C:\windbg\windbg.exe
Image path: C:\windbg\windbg.exe
Image name: windbg.exe
Timestamp: Thu Mar 29 21:09:08 2007 (460C00C4)
CheckSum: 0008852B
ImageSize: 00093000
File version: 6.7.5.0
Product version: 6.7.5.0
File flags: 8 (Mask 3F) Private
File OS: 40004 NT Win32
File type: 1.0 App
File date: 00000000.00000000
Translations: 0409.04b0
CompanyName: Microsoft Corporation
ProductName: Debugging Tools for Windows(R)
InternalName: windbg.exe
OriginalFilename: windbg.exe
ProductVersion: 6.7.0005.0
FileVersion: 6.7.0005.0 (debuggers(dbg).070215-1229)
FileDescription: Windows GUI symbolic debugger
LegalCopyright: © Microsoft Corporation. All rights reserved.
01400000 015c6000 ext (deferred)
…

如果想控制要显示的模块，那么可以使用如下方法之一：

- 使用 m 开关来指定对模块名的过滤模式，比如 lm m k*显示以模块名 k 开头的模块。
- 使用 M 开关来指定对模块路径的过滤模式。
- 使用 o 开关只显示加载的模块（排除已经卸载的模块）。
- 使用 l 开关只显示已经加载符号的模块。
- 使用 e 开关只显示有符号问题的模块。

也可以使用!lmi 扩展命令可以来观察模块的信息，但是这个命令每次只能观察一个模块。比如以下是对 WinDBG 模块的显示结果：

```
0:001> !lmi windbg
Loaded Module Info: [windbg]
         Module: windbg
   Base Address: 01000000
     Image Name: C:\windbg\windbg.exe
   Machine Type: 332 (I386)
     Time Stamp: 460c00c4 Thu Mar 29 21:09:08 2007
           Size: 93000
       CheckSum: 8852b
Characteristics: 102
Debug Data Dirs: Type  Size     VA  Pointer
```

```
            CODEVIEW 23,c348,b748 RSDS - GUID: {CDA70185-4AB9-4F6F-8B60-FDC14F75FB31}
              Age: 1, Pdb: windbg.pdb
 Image Type: FILE     - Image read successfully from debugger.
             C:\windbg\windbg.exe
Symbol Type: PDB      - Symbols loaded successfully from symbol server.
             d:\symbols\windbg.pdb\CDA701854AB94F6F8B60FDC14F75FB311\windbg.pdb
Load Report: public symbols , not source indexed
```

30.8.6分析符号

可以用标准命令 x（或 X，不分大小写）来分析调试符号，其命令格式如下：

X [选项] 模块名!符号名

其中的模块名和符号名都可以包含通配符，*代表 0 或任意多个字符，?代表任一个单一字符，#代表它前面的字符可以出现任意次，比如 lo#p 表示通配 l 和 p 之间有任意多个 o 的单词，lop，loop，looop，…。如果中间允许多个字符重复，那么可以使用方括号，例如用 m[ai]#n 可以通配 man、min、maan、main、maiain 等。

例如，使用 x ntdll!dbg*可以列出 ntdll 模块的所有以 dbg 开头的符号。

0:000> x ntdll!dbg*

7c95081a ntdll!DbgUiDebugActiveProcess = <no type information>

…

第一列是这个符号的地址，如果符号是函数，那么便是这个函数的入口地址，如果符号是变量，那么便是这个变量的起始地址。等号后面用来显示符号的取值，这通常需要私有符号信息，也就是调试版本的符号文件。因为我们没有 NTDLL 的私有符号信息（类型信息），无法显示其值，所以 WinDBG 就显示<no type information>。

打开调试版本的 dbgee 小程序，然后执行 x dbgee!arg*命令，得到的结果如下：

0:000> x dbgee!arg*

0041718c dbgee!argret = 0

00417184 dbgee!argv = 0x003a2e90

0041717c dbgee!argc = 3

可见等号后面出现了每个变量的取值，argc 是命令行参数的个数，argv 是参数数组的指针。

类似的，模块名中也可以使用通配符，比如 x *!_crtheap 会检查所有模块，看其是否有_crtheap 符号，如果有便显示出来：

0:000> x *!_crtheap

103130d0 MSVCR80D!_crtheap = <no type information>

77c62418 msvcrt!_crtheap = <no type information>

下面我们看一下 x 命令的选项。目前它支持如下选项：

- 控制显示结果的排列顺序，/a 和/A 分别按地址的升序和降序，/n 和/N 分别按名称的升序和降序，/z 和/Z 分别按符号大小（size）的升序和降序。
- 显示符号的数据类型（/t）。
- 显示符号的符号类型和大小(/v),其中符号类型分为 local(局部)、global(全局)、parameter（参数）、function（函数）、或者 unknown（未知）。
- 按符号大小设置过滤条件，其格式为/s <符号大小>。对于函数类符号，其大小是这个函

数在内存中的大小（字节数），对于其它符号，是这个符号的数据类型的大小。

- 控制显示格式，/p 可以省去函数名与括号之间的空格，/q 参数可以启用所谓的引号格式来显示符号名。

下面给出几个例子来说明以上选项。首先我们看/v 选项，在前面的 x dbgee!arg*命令中加入/v：

```
0:000> x /v dbgee!arg*
prv global 0041718c       4 dbgee!argret = 0
prv global 00417184       4 dbgee!argv = 0x003a2e90
prv global 0041717c       4 dbgee!argc = 3
```

现在的显示多了三列，最左边的 prv 代表这个符号属于私有（private）符号信息，如果是公共符号，那么显示为 pub（public）。第二列是符号类型，global 代表全局变量，接下来是这个符号在调试目标中的地址，第 4 列是符号的大小。这里的几个符号的大小都是 4 个字节。

下面再增加/t 选项：

```
0:000> x /v /t dbgee!arg*
prv global 0041718c       4 int dbgee!argret = 0
prv global 00417184       4 unsigned short ** dbgee!argv = 0x003a2e90
prv global 0041717c       4 int dbgee!argc = 3
```

可见，在符号大小后面多了数据类型。Argret 和 argc 的类型都是 int（整数），argv 是 unsigned short **即 wchar_t **，也就是字符串指针数组。因为 argc 是数组的长度，因此我们可以使用 dd 命令得到每个数组元素的值：

```
0:000> dd 0x003a2e90 l3
003a2e90  003a2ea0 003a2f08 003a2f0e
```

因为是 wchar_t **类型，所以每个数组元素应该是一个字符串指针（wchar_t *），于是可以用 du 命令来观察：

```
0:000> du 003a2ea0
003a2ea0  "C:\dig\dbg\author\code\chap28\db"
003a2ee0  "gee\Debug\dbgee.exe"
0:000> du 003a2f08
003a2f08  "/o"
0:000> du 003a2f0e
003a2f0e  "adv.ini"
```

这正是三个命令行参数（对于控制台程序，第一个参数总是程序文件名）。

以下是有无/p 开关的比较：

```
0:000> x /v dbgee!wmain
prv func    00411790    51 dbgee!wmain (int, wchar_t **)
```

不指定/p 时，wmain 与左括号间有一个空格。

```
0:000> x /v /p dbgee!wmain
prv func    00411790    51 dbgee!wmain(int, wchar_t **)
```

指定/p 后这个空格被删除，这主要是为了复制整个函数声明时会更方便些。

以下是使用更多选项的例子：

```
0:000> x /v /q /t /N dbgee!*main*
prv func    00411520     f <function> @!"dbgee!wmainCRTStartup" ()
prv func    00411790    51 <function> @!"dbgee!wmain" ()
prv global 00417194     4 int @!"dbgee!mainret" = 0
```

```
pub global 00418288     0 <NoType> @!"dbgee!_imp____wgetmainargs" = <no type information>
pub global 00411c92     0 <NoType> @!"dbgee!__wgetmainargs" = <no type information>
prv func   00411540   244 <function> @!"dbgee!__tmainCRTStartup" ()
prv global 00417020     4 unsigned int @!"dbgee!__native_dllmain_reason" = 0xffffffff
```

因为使用了/q 参数，所以以上符号名是以@!"模块名!符号名"的格式现实的。另一点值得注意的是因为公开的符号信息不包括类型信息，所以其类型部分显示为<NoType>，符号大小也显示为 0。

30.8.7搜索符号

标准命令 ln（List Nearest Symbols）用来搜索距离指定地址最近的符号。比如：

```
lkd> ln 8053ca11
(8053ca11)   nt!KiSystemService    |  (8053ca85)   nt!KiFastCallEntry2
Exact matches:
    nt!KiSystemService = <no type information>
```

上面的结果显示了地址 8053ca11 附近的两个符号，其中 KiSystemService 精确匹配。

30.8.8设置符号选项

元命令.symopt 用来显示和修改符号选项，其命令格式为：

.symopt[+/- 选项标志]

WinDBG 使用一个 32 位的 DWORD 来记录符号选项，每个位代表一个选项。使用+可以设置这个标志位，-用来移除某个标志位，不带任何参数便显示当前的设置。表 30-5 列出了目前定义的所有标志位。

表 30-5 符号选项的各个标志位

标志位	常量	含义	缺省值
0x1	SYMOPT_CASE_INSENSITIVE	不分大小写	On
0x2	SYMOPT_UNDNAME	显示未装饰的符号名	On
0x4	SYMOPT_DEFERRED_LOADS	延迟加载符号	On
0x8	SYMOPT_NO_CPP	关闭 C++翻译*	Off
0x10	SYMOPT_LOAD_LINES	加载源代码行信息	**
0x20	SYMOPT_OMAP_FIND_NEAREST	对于优化代码，允许使用最相近的符号	On
0x40	SYMOPT_LOAD_ANYTHING	降低匹配符号的挑剔度	Off
0x80	SYMOPT_IGNORE_CVREC	胡略加载映像的 CV 记录	Off
0x100	SYMOPT_NO_UNQUALIFIED_LOADS	禁止符号处理器自动加载模块	Off
0x200	SYMOPT_FAIL_CRITICAL_ERRORS	显示关键错误	On
0x400	SYMOPT_EXACT_SYMBOLS	严格评估所有符号文件	Off
0x800	SYMOPT_ALLOW_ABSOLUTE_SYMBOLS	允许位于内存绝对地址的符号	Off

0x1000	SYMOPT_IGNORE_NT_SYMPATH	胡略环境变量中的符号和映像路径	Off
0x2000	SYMOPT_INCLUDE_32BIT_MODULES	在安腾处理器上模拟 32-位模块	Off
0x4000	SYMOPT_PUBLICS_ONLY	仅使用公共符号	Off
0x8000	SYMOPT_NO_PUBLICS	不使用公共符号	Off
0x10000	SYMOPT_AUTO_PUBLICS	最后使用公共符号***	On
0x20000	SYMOPT_NO_IMAGE_SEARCH	不搜索映像文件	On
0x40000	SYMOPT_SECURE	（内核调试） Secure Mode	Off
0x80000	SYMOPT_NO_PROMPTS	（远程调试）不显示代理服务器的认证对话框	****
0x80000000	SYMOPT_DEBUG	显示符号加载过程	Off

* 使用 C++翻译时，类成员的__或被替换为::。

** 在 KD and CDB 中缺省为 Off，在 WinDbg 中缺省为 On。进行源代码级调试时，必须设置这个选项。

*** 当搜索私有符号失败时再使用公共符号。

**** 在 KD and CDB 中缺省为 On，在 WinDbg 中缺省为 Off。

因为记忆和使用 16 进制的标志位比较困难，所以 WinDBG 提供了扩展命令!sym 来简化某些常用的选项设置。比如!sym noisy 相当于.symopt+0x80000000，即开启所谓的“吵杂”式符号加载，显示符号加载的调试信息。!sym quiet 相当于.symopt-0x80000000，用来关闭这种方式。

0:000> !sym noisy

noisy mode - symbol prompts on

例如以下是 WinDBG 帮助文件中给出的一个重新加载 nt 内核模块时的情景：

```
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
```

经过笔者试验，目前版本的 WinDBG 不再显示如此丰富的信息，只是简要的：

lkd> .reload nt

DBGHELP: nt - public symbols

d:\symbols\ntkrnlpa.pdb\CF7B79A8CE864FCF8ABF248F0B69F4C91\ntkrnlpa.pdb

30.8.9加载不严格匹配的符号文件

在实际工作中，有时要调试的程序只是做了简单的重新构建（rebuild），代码仅有微小的变化或者根本没有变化。这时，如果调试环境中只有旧的符号文件，那么调试器缺省仍会发现符号文件和映像文件不匹配，拒绝加载符号文件。

一种解决方法是使用/i 参数来重新加载缺少符号的程序文件。为了便于发现问题，最好先开启符号加载的“吵杂”模式。

```
0:000> !sym noisy
noisy mode - symbol prompts on
0:000> .reload /i dbgee.exe
SYMSRV:  d:\symbols\dbgee.pdb\75DCAE56ACE24AAE97FDFF4F915565162\dbgee.pdb not found
SYMSRV:
http://msdl.microsoft.com/download/symbols/dbgee.pdb/75DCAE56ACE24AAE97FDFF4F9155651
62/dbgee.pdb not found
DBGHELP: C:\dig\dbg\author\code\chap28\dbgee\Debug\dbgee.pdb - mismatched pdb
DBGHELP: c:\dig\dbg\author\code\chap28\dbgee\Debug\dbgee.pdb - mismatched pdb
DBGHELP: Loaded mismatched pdb for C:\dig\dbg\author\code\chap28\dbgee\Debug\dbgee.exe
*** WARNING: Unable to verify checksum for dbgee.exe
DBGENG:  dbgee.exe has mismatched symbols - type ".hh dbgerr003" for details
DBGHELP: dbgee - private symbols & lines
         C:\dig\dbg\author\code\chap28\dbgee\Debug\dbgee.pdb - unmatched
```

以上信息说明，WinDBG 尽管发现 PDB 文件与要求不完全匹配，但是它还是加载了。使用 lm 命令显示模块列表，也可以看到已经为 dbgee 模块加载了符号：

```
0:000> lm
start    end        module name
00400000 0041a000   dbgee    M (private pdb symbols)  C:\...\dbgee\Debug\dbgee.pdb
…
```

其中的 M 表示符号文件和执行映像文件存在不匹配。

除了使用带有/i 开关的.reload 命令，也可以通过设置符号选项 SYMOPT_LOAD_ANYTHING（0x40）来让调试器加载不严格匹配的符号文件。

```
0:000> .symopt+0x40
Symbol options are 0x30277:
  0x00000001 - SYMOPT_CASE_INSENSITIVE
  0x00000002 - SYMOPT_UNDNAME
  0x00000004 - SYMOPT_DEFERRED_LOADS
  0x00000010 - SYMOPT_LOAD_LINES
  0x00000020 - SYMOPT_OMAP_FIND_NEAREST
  0x00000040 - SYMOPT_LOAD_ANYTHING
  0x00000200 - SYMOPT_FAIL_CRITICAL_ERRORS
  0x00010000 - SYMOPT_AUTO_PUBLICS
  0x00020000 - SYMOPT_NO_IMAGE_SEARCH
```

本节使用比较大的篇幅详细介绍了调试符号有关的 WinDBG 命令。熟练使用这些命令对于调

试非常重要，希望读者能够在实际调试中进一步体会和应用。

30.9 事件处理

从某种程度上来说，Windows 的调试模型是事件驱动的。整个调试过程就是围绕调试事件的产生、发送、接收和处理为线索而展开的。调试目标是调试事件的发生源，介绍和处理调试事件是调试器的核心任务，调试子系统负责将调试事件发送给调试器，并执行调试器的处理结果。

我们在第 9 章介绍了调试事件，本节我们将从调试器的角度来介绍调试事件。

30.9.1 调试事件与异常的关系

简单说来，异常是调试事件的一种。Windows 定义了 9 类调试事件，分别用以下 9 个常量来表示：EXCEPTION_DEBUG_EVENT（1）、CREATE_THREAD_DEBUG_EVENT（2）、CREATE_PROCESS_DEBUG_EVENT（3）、EXIT_THREAD_DEBUG_EVENT（4）、EXIT_PROCESS_DEBUG_EVENT（5）、LOAD_DLL_DEBUG_EVENT（6）、UNLOAD_DLL_DEBUG_EVENT（7）、OUTPUT_DEBUG_STRING_EVENT（8）、RIP_EVENT（9）。其中异常事件的代码为 EXCEPTION_DEBUG_EVENT（1）。

因为有很多种异常，所以异常事件又根据异常代码分为很多个子类。其它事件都比较单纯，不再包含子类。常见的异常子类有：

- Win32 异常，这是 Windows 操作系统所定义的异常，其中主要是 CPU 产生的异常，典型的有非法访问、除零等。这类异常的异常代码定义在 ntstatus.h 中。
- Visual C++异常，这是 Visual C++编译器的 throw 关键字所抛出的异常，throw 关键字调用 RaiseException API 产生异常。所有这类异常的异常代码都是 0xe06d7363（.msc）。
- 托管异常，这是.Net 程序使用托管方法抛出的异常。所有这类异常的异常代码都是 0xe0636f6d（.com）。
- 其它异常，包括用户程序直接调用 RaiseException API 抛出的异常，以及其它 C++编译器抛出的异常等。

除了以上 9 类调试事件，为了复用事件处理机制，调试器定义了某些专门供调试使用的事件，比如 WinDBG 定义了用于唤醒处于睡眠状态的调试器的 Wake Debugger 事件。我们把这类事件通常为调试器事件。

30.9.2 两轮机会

我们在第 10 章介绍异常管理时，曾经详细讨论过 Windows 操作系统异常的过程。其中最重要的一点就是每个异常，Windows 会最多给于两轮处理机会。对于每一轮机会 Windows 都会先分发给调试器（如果存在），然后再寻找异常处理器（VEH、SEH 等）。这样看来，对于每个异常，调试器都可能收到两次通知，或者说有两次处理机会，每次处理后调试器都应该向系统返回一个结果，说明它是否处理了这个异常。

对于第一轮异常处理机会，调试器通常是返回没有处理异常，然后让系统继续分发，交给程序中的异常处理器来处理。对于第二轮机会，如果调试器不处理，那么系统便会采取终极措施：如果异常发生在应用程序中，那么便立刻终止应用程序；如果发生在内核代码，那么便蓝屏停止系统。所以对于第二轮处理机会，调试器通常是返回已经处理，这样让程序恢复执行，通常是又

导致异常，又重新分发异常，如此循环。

考虑到异常事件是最主要的一类调试事件，所以调试器通常把其事件也纳入到同一个框架下来管理。但是必须清楚的是，异常以外的调试事件（比如进程创建）通常是一次性的，没有两次通知。

30.9.3 定制事件处理方式

大多数调试器都允许用户来定制处理调试事件的方式，WinDBG 也如此。因为异常事件最多有两轮处理机会，而且对于每一轮机会都需要决定如下两个问题：

- 当收到事件通知后是否中断给用户，即所谓的中断状态（break status）。中断给用户的含义就是让调试器进入到命令状态，启动交互式的诊断界面。
- 返回给系统的处理结果，是返回已经处理（handled），还是没有处理（not handled），即所谓的处理状态（handling status），有时也称为继续状态（continue status），因为涉及到程序如何继续（执行）。

所以，对于每个异常事件都有四个选项：

- 第一轮机会是否中断给用户；
- 第一轮机会的处理结果；
- 第二轮机会是否中断给用户；
- 第二轮机会的处理结果。

为了允许用户设置以上选项，不同调试器设计的界面外观上有所不同。图 30-1 所示的是 Visual Studio 2005（VS8）的异常过滤对话框，对于每个异常，它只提供了两个选项，分别称为 Thrown 和 User-unhandled。前者的含义是对于第一轮机会是否中断给用户，后者的含义是对于第二轮机会是否中断给用户。也就是说，VS8 允许用户配置两轮机会的中断选项，但是没有让用户配置继续选项。

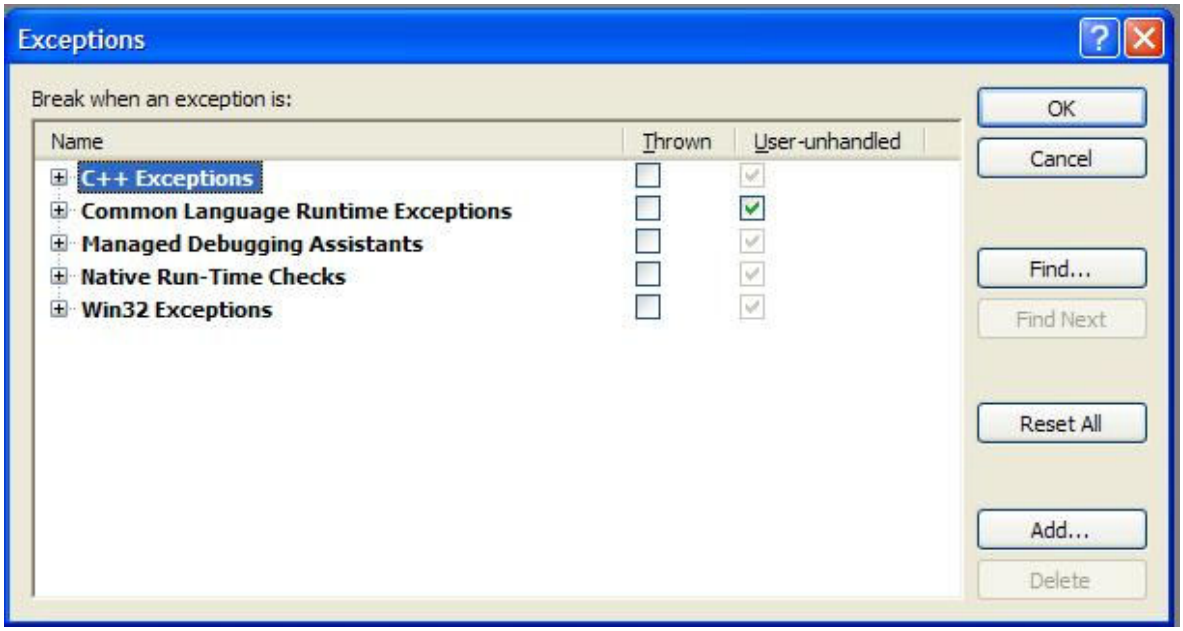


图 30-1 Visual Studio 2005 的异常处理选项对话框

Visual Studio .Net 2002 和 2003（VS7）的配置对话框（图 30-2）虽然界面上看起来有较大差异，但实质是一样的。VS7 的界面更直观和好理解一些。

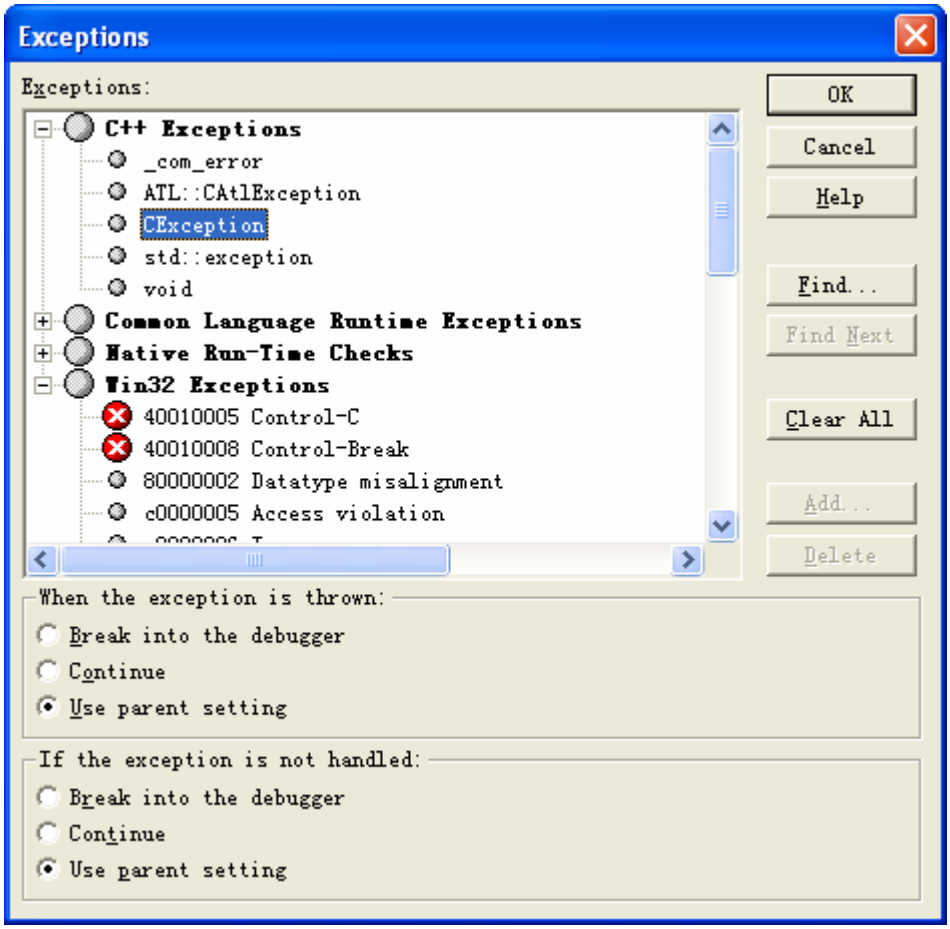


图 30-2 VS7 的异常处理选项对话框

VC6 调试器的异常处理选项对话框（图 30-3）的外观又有所不同，但其配置的实质仍然是一样的，也就是决定两轮机会是否都中断给用户（stop always），还是只有第二轮时中断（stop if not handled）。

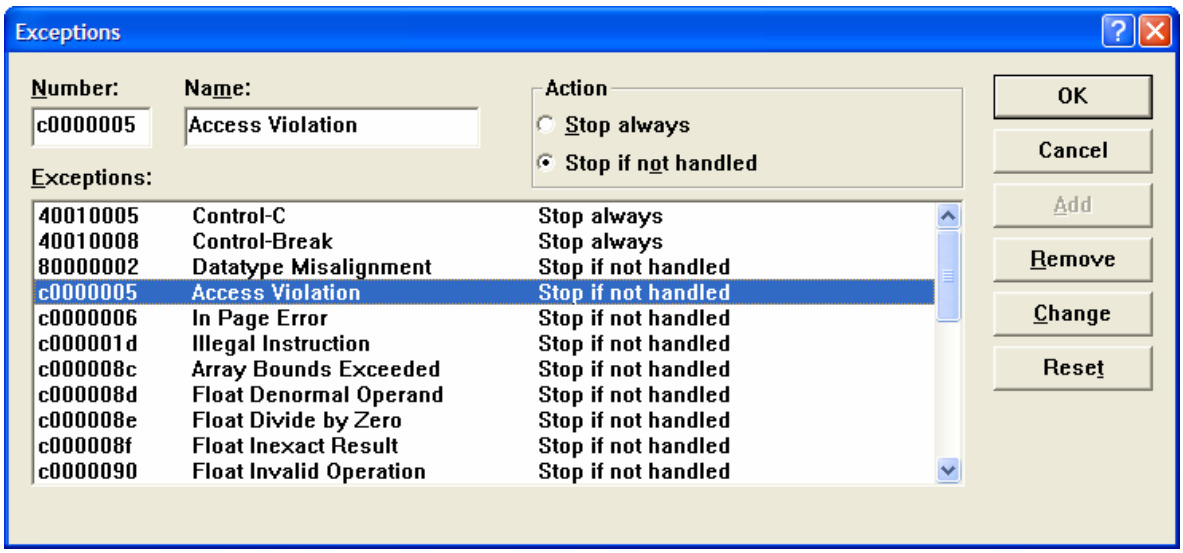


图 30-3 VC6 调试器的异常处理选项对话框

图 30-4 是 WinDBG 的异常处理选项对话框。与 VS 调试器各个版本的对话框变化很大不同，WinDBG 自从 2.0 版本开始这个对话框的界面便是这样的。

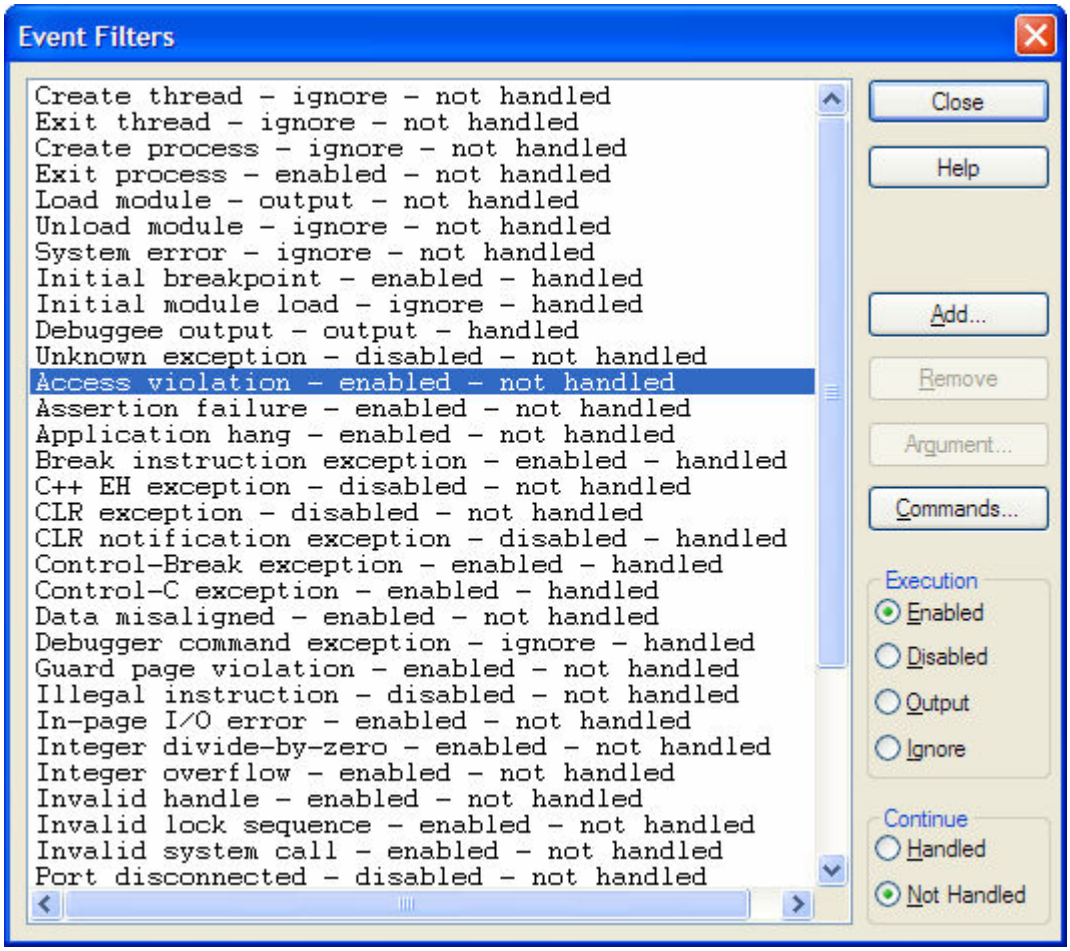


图 30-4 WinDBG 的异常处理选项对话框

首先，Execution 组的四个单选按钮的含义如下：

- Enabled – 第一轮机会便中断给用户，第二轮也中断。
- Disabled – 第二轮机会时中断给用户，第一轮不中断。
- Output – 输出信息通知用户。
- Ignore – 胡略这个事件。

Continue 组的两个单选按钮是配置事件的处理状态，是返回给系统已经处理异常，还是没有处理异常。对于异常来说，它是针对第一轮处理机会的。

所以对于 WinDBG 的配置对话框允许设置上面四个选项中的三个，对于第二轮异常的返回值，它默认返回已经处理。在调试时，如果需要返回没有处理，那么可以使用 gn 命令。

另外，WinDBG 允许为每个事件定义命令。点击 Commands 按钮便弹出一个对话框，允许用户输入一系列关联的命令，对于异常事件，可以为每一轮机会输入一系列命令。

除了使用图形界面，也可以使用命令来配置异常处理选项：

sx{e|d|i|n} [-c "Cmd1"] [-c2 "Cmd2"] [-h] {Exception|Event|*}

其中的 e|d|i|n 对应于图形界面的 Enabled、Disabled、Output 和 Ignore，-c 和-c2 分别用来定义第一轮机会和第二轮机会的关联命令，Exception|Event 用来指定这条命令所针对的异常或者事件。WinDBG 为常见的异常和事件定义了一个简单的代码，比如非法访问异常的代码是 av，除零异常的代码是 dz，线程退出的代码是 et 等等。

如果指定了-h 开关，那么这条命令就是用来设置处理状态（handling status），而不是中断状态。此时，sxe 命令设置的处理状态是 Handled，其它三条命令都是设置为 Not Handled。

使用 sxr 命令可以将所有事件处理选项恢复为缺省值。

直接输入 sx 命令可以列出各个事件的代码和目前的处理选项。

30.9.4GH 和 GN 命令

当因为发生异常而中断到调试器中时，如果使用 g 命令恢复调试目标执行，那么调试器将使用上面介绍的配置方法中的设置而决定返回给系统的处理状态。如果调试人员希望返回与设置不同的状态，那么可以使用 GH 或者 GN 命令。前者用来返回已经处理（Handled），后者用来返回没有处理（Not Handled）。

30.9.5实验

下面通过一个实验来加深大家的理解。我们使用第 28 章所用的 dbgee 小程序作为调试目标，它的主要代码如下：

```
1 int _tmain(int argc, _TCHAR* argv[])
2 {
3       if(argc==1)
4       {
5             *(int *)0=1;
6             printf("test\n");
7       }
8    return 0;
9 }
```

启动 WinDBG 然后通过 Open Executables 打开 dbgee.exe。WinDBG 成功创建进程和创建调试会话后，会因为初始断点和中断给用户。命令窗口显示如下信息：

```
(9fc.db0): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffde000 ecx=00000004 edx=00000010 esi=00251f48 edi=00251eb4
…
ntdll!DbgBreakPoint:
7c901230 cc                    int       3
```

其中 80000003 是由于断点指令（INT 3）异常的异常代码，括号中的 first chance 代表这是第一轮处理机会。这说明对于断点异常，WinDBG 收到第一轮通知时，便中断给用户。观察图 30-4 所示的对话框，可以看到这个异常（Break instruction exception）的处理选项是 enabled – handled，即第一轮机会便中断，返回已经处理这个异常。

然后依次执行如下步骤：

1. 输入 sxe av 命令设置对于访问异常（av）第一轮便中断。
2. 输入 sxd –h av 命令设置访问异常的处理状态是不处理。
3. 输入 sx 命令，确保关于访问异常的设置是如下内容：

   av - Access violation - break - not handled
4. 输入 g 命令让调试目标执行。
5. 因为源代码第 5 行的空指针访问，所以程序会产生非法访问异常。调试收到调试事件后，检查异常处理设置，发现这个异常的设置是 Enable（第一轮便中断给用户），于是进入命令模式，并显示如下内容：

   (1574.14f8): Access violation - code c0000005 (first chance)

First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=cccccccc ebx=7ffdf000 ecx=00000000 edx=00000001 esi=0142f774 edi=0012ff68
…
dbgee!wmain+0x24:
004113b4 c7050000000001000000 mov dword ptr ds:[0],1 ds:0023:00000000=????????
第一行的 c0000005 是非法访问异常的异常代码，(first chance)代表这是第一轮处理机会。第二、三行提示我们系统还没有执行程序中的异常处理器，对于某些程序来说，异常可能是故意抛出的，可能属于期望的情况。

6. 执行 gh 命令，让调试器返回已经处理了这个异常。这时尽管系统会中断异常分发（因为调试器声称已经处理了异常），恢复调试目标继续执行，但由于异常条件仍在，所以还有产生异常，于是再次分发。WinDBG 重复执行上面一步，并显示同样的信息。
7. 执行 g 命令。因为对这个异常的设置是 Not Handled（第一轮异常的处理结果是没有处理），所以系统会继续分发这个异常，寻找各种异常处理器（VEH、SHE 等）。因为上面的代码没有任何异常处理器，所以系统会执行缺省的异常处理器（参见第 10 章），执行系统的 UnhandledExceptionFilter 函数（位于 kernel32.dll 中）。UnhandledExceptionFilter 函数会判断当前程序是否在被调试，如果不在调试，那么便启动应用程序错误对话框，通知用户终止程序。如果在被调试，那么 UnhandledExceptionFilter 便返回 EXCEPTION_CONTINUE_SEARCH，这会导致系统继续分发这个异常，即进入异常的第二轮分发。对于第二轮机会，系统仍然是先分给调试器。对于第二轮机会，WinDBG 总会中断到调试器，并显示如下信息：

(1574.14f8): Access violation - code c0000005 (!!! second chance !!!)
eax=cccccccc ebx=7ffdf000 ecx=00000000 edx=00000001 esi=0142f774 edi=0012ff68
…
dbgee!wmain+0x24:
004113b4 c7050000000001000000 mov dword ptr ds:[0],1 ds:0023:00000000=????????

8. 输入 g 命令让目标继续执行。对于第 2 轮机会，WinDBG 总是默认使用已经处理作为处理状态返回给系统。这会导致系统恢复执行目标，重新产生异常，WinDBG 又得到第一轮机会，显示第 5 步中的信息。
9. 输入 g，得到第二论处理机会后，输入 gn 强制调试器告诉系统没有处理第 2 轮异常，这时会发现调试目标突然消失，因为系统将其强制终止。

本节介绍了调试事件的有关内容，这些内容是我们异常系列内容的最后一部分。理解这部分内容需要前面的基础，建议读者在阅读本节时遇到不清楚的内容便返回到前面的章节，复习一下前面的内容。

30.10 控制调试目标

有些软件问题通过观察症状然后审查代码就可以发现根源并找到解决方案，但更多的问题是需要跟踪程序的执行过程才能摸清来龙去脉发现症结所在的。自由控制程序的停止和运行是设计调试器的最基本目标，也是调试器的威力所在。所谓交互式调试，其主要内涵就是可以与被调试程序互动，可以让其停下来接受观察，观察好了可以让其继续运行一段，然后再停下来观察……

控制调试目标（被调试程序）是调试器的一个核心任务。其宗旨就是使调试目标始终处于调试器的控制之下，让调试人员可以可以随心所欲的控制程序的执行状态。

WinDBG 提供了强大的机制和丰富的命令来控制调试目标，本节我们将介绍这些命令和有关的使用技巧。

30.10.1初始断点

当调试一个新创建的进程（用户态目标）时，为了让调试人员可以尽早的分析调试目标，Windows 操作系统的进程加载器加入了特别的调试支持：在完成最基本的用户态初始后，系统的进程初始化函数就会主动执行断点指令，触发断点，让调试目标中断到调试器中。这个断点被称为初始断点（Initial Breakpoint）。

举例来说，当我们使用 WinDBG 打开（Open Executable）Dbgee.exe 程序时，很快 WinDBG 的便显示出如下信息：

```
CommandLine: C:\dig\dbg\author\code\chap28\dbgee\Debug\dbgee.exe
Symbol search path is: SRV*d:\symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 0041a000      dbgee.exe
ModLoad: 7c900000 7c9b0000      ntdll.dll
ModLoad: 7c800000 7c8f5000      C:\WINDOWS\system32\kernel32.dll
ModLoad: 10200000 10320000      C:\WINDOWS\WinSxS\...\MSVCR80D.dll
ModLoad: 77c10000 77c68000      C:\WINDOWS\system32\msvcrt.dll
(1360.1740): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffdb000 ecx=00000004 edx=00000010 esi=00251f48 edi=00251eb4
eip=7c901230 esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c901230 cc               int       3
```

以上信息表明调试目标因为执行了断点指令（INT 3）而中断。执行 kn 命令观察栈调用：

```
0:000> kn
 # ChildEBP RetAddr
00 0012fb1c 7c93edc0 ntdll!DbgBreakPoint
01 0012fc94 7c921639 ntdll!LdrpInitializeProcess+0xffa
02 0012fd1c 7c90eac7 ntdll!_LdrpInitialize+0x183
03 00000000 00000000 ntdll!KiUserApcDispatcher+0x7
```

可见，NTDLL 的 LdrpInitializeProcess 函数调用了 DbgBreakPoint，后者包含了断点指令。

我们知道，当创建一个新进程时，很多早期的创建工作（创建进程对象、进程空间、建立初始线程、通知子系统等）都是在父进程的环境下完成的。初始线程真正在新进程环境下执行是从内核态的 KiThreadStartup 开始的。KiThreadStartup 将线程的 IRQL（中断级别）降到 APC 级别后调用 PspUserThreadStartup 来为线程在用户态执行做准备。

因为 PspUserThreadStartup 仍然是在内核态的，为了可以执行用户态的加载工作，它初始化了一个对用户态代码的异步过程调用（APC），并插入到 APC 队列中，这个 APC 是调用 NTDLL.DLL 中的 LdrpInitialize 函数。

因此可以说，LdrpInitialize 函数是一个新进程的初始线程开始在用户态执行的最早代码。LdrpInitialize 在初始化加载器和读取执行选项后，调用 LdrpInitializeProcess 函数。

LdrpInitializeProcess 函数的一个主要任务就是加载 EXE 文件所依赖的动态链接库。在加载每个 DLL 后，LdrpInitializeProcess 检查当前进程是否在被调试（PEB 的 BeingDebugged 字段），如

果是，则调用 DbgBreakPoint 通知调试器。注意此时尚未调用每个 DLL 的 DllMain 函数。

当 LdrpInitialize 执行完毕后，KiUserApcDispatcher 调用 ZwContinue 返回到内核态的 PspUserThreadStartup 函数中。接下来 PspUserThreadStartup 函数把现成的 IRQL 降低到 0（PASSIVE）。而后，系统开始执行已经放在线程上下文中的进程启动函数 BaseProcessStart，后者调用程序的入口函数使用程序开始运行。

当将 WinDBG 附加到一个已经运行的进程时，WinDBG 默认也会通过在目标进程创建一个远程进程来触发一个初始断点。但这个断点是发生在新创建的线程上下文中，其栈调用通常为：

```
0:001> kn
 # ChildEBP RetAddr
00 00cdffc8 7c9507a8 ntdll!DbgBreakPoint
01 00cdfff4 00000000 ntdll!DbgUiRemoteBreakin+0x2d
```

值得注意的是，这个线程并不是目标进程的本来线程，它是调试器创建的。当我们恢复目标执行时，这个线程也会立刻结束。

在 WinDBG 的命令行中加入-g 开关，可以让其胡略或者不发起初始断点，也就是对于调试新进程的时候当接收到初始断点事件时，不中断给用户；当附加到已经创建的进程时，不再发起远程线程来触发断点。

另外，要说明的是初始断点并不是调试器可以得到的最早控制机会，进程创建事件和某些模块加载事件都要比初始断点的时间还要早。

如果要跟踪和分析程序的或者 DLL 的入口函数，那么在初始断点时对入口函数设置断点是个合适的时机。

30.10.2俘获调试目标

有以下几种方法可以把运行的调试目标俘获（Acquire）到调试器中：

- 在调试器界面中选择中断命令（Debug > Break）或者使用 Ctrl+Break 热键。
- 对于有窗口界面的程序，将被调试程序窗口切换到前台，然后按 F12 热键（参见第 9 章）。
- 如果已经设置了断点，或者在代码中加入了特殊的异常代码，那么便执行相应的操作，让程序触发断点或者异常，使其中断到调试器。

因为第一种方法使用最多，所以我们介绍一下它的工作细节。清单 30-5 显示了 WinDBG 的 UI 线程收到 Break 命令后的工作过程。

清单 30-5 调试器 UI 线程处理 Break 命令的过程

```
0:000> kn
 # ChildEBP RetAddr
00 0006ce54 7c93401e ntdll!ZwCreateThread
01 0006d1bc 7c9507ff ntdll!RtlCreateUserThread+0xdc
02 0006d1fc 7c85a383 ntdll!DbgUiIssueRemoteBreakin+0x26
03 0006d208 0229c11b kernel32!DebugBreakProcess+0xd
04 0006d230 02225a4c dbgeng!LiveUserDebugServices::RequestBreakIn+0x1b
05 0006d24c 020c7126 dbgeng!LiveUserTargetInfo::RequestBreakIn+0x5c
06 0006d258 0103cb21 dbgeng!DebugClient::SetInterrupt+0xa6
07 0006ddf4 7e418724 windbg!FrameWndProc+0x13f1
08 0006de20 7e418806 USER32!InternalCallWinProc+0x28
```

```
09 0006de88 7e4189bd USER32!UserCallWinProcCheckWow+0x150
0a 0006dee8 7e418a00 USER32!DispatchMessageWorker+0x306
0b 0006def8 0104d062 USER32!DispatchMessageW+0xf
0c 0006df14 0104d0d4 windbg!ProcessNonDlgMessage+0x2a2
0d 0006df3c 01052895 windbg!ProcessPendingMessages+0x64
0e 0006ff7c 01056046 windbg!wmain+0x205
0f 0006ffc0 7c816ff7 windbg!_initterm_e+0x163
10 0006fff0 00000000 kernel32!BaseProcessStart+0x23
```

依照函数调用的先后顺序（从下至上），栈帧 7 是窗口的过程函数，它收到 Break 命令后通过全局变量 g_DbgClient 调用 SetInterrupt 方法，这个方法的用途就是让调试器进入命令状态，以便用户可以开始执行各种分析和诊断工作，其函数原型如下：

```
HRESULT   IDebugControl::SetInterrupt(
    IN ULONG   Flags );
```

其中 Flags 参数可以包含如下标志

- DEBUG_INTERRUPT_PASSIVE（1），向调试引擎注册用户希望中断到命令模式，但是不强制。其函数内部是将 dbgeng!g_UserInterruptCount 加 1，将 dbgeng!g_EngStatus 设置为 0x1005。
- DEBUG_INTERRUPT_EXIT（2），设置 dbgeng!g_EngStatus 的 0x800 位，让调试器引擎取消等待调试事件，强制返回。通常这会导致调试器没有中断调试目标就进入到命令模式，因为没有合适的进程和线程上下文，所以命令提示符会包含多个问号（图 30-8），此时大多数涉及调试目标的命令都无法执行。
- DEBUG_INTERRUPT_ACTIVE（0），判断全局变量 dbgeng!g_CmdState 如果当前调试器没有处于命令状态，那么要求调试目标中断到调试器以进入命令状态，如果调试器已经在命令状态，那么只是简单的递增 dbgeng!g_UserInterruptCount 变量。

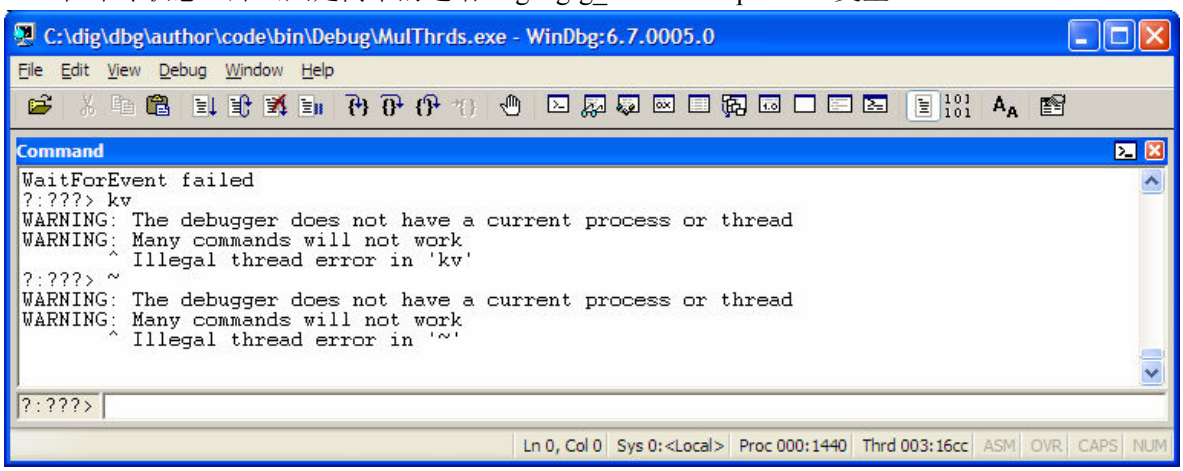


图 30-8 调试器被强制进入命令模式

在清单 30-5 中，UI 线程是使用参数 0 来调用 SetInterrupt 方法的，也就是要求调试目标中断到调试器中。

栈帧 5 和 4 分别是调用调试目标类（LiveUserTargetInfo）和调试服务类（LiveUserDebugServices）的 RequestBreakIn 方法，将中断请求层层下传。栈帧 3 是调用操作系统的调试 API，DebugBreakProcess。

```
BOOL DebugBreakProcess(  HANDLE Process );
```

栈帧 2～0 显示了 DebugBreakProcess API 的内部工作过程，栈帧 2 是调用 NTDLL 中的 DbgUiIssueRemoteBreakin，栈帧 1 和 0 是调用线程创建函数在调试目标进程中创建远程线程。这

个远程线程的线程函数是 NTDLL 中的 DbgUiRemoteBreakin 函数，它内部的代码很少，只是简单的调用 DbgBreakPoint API 来执行断点指令，产生一个断点异常。

当目标进程中断到调试器中时，当前线程就是调试器创建的这个远程线程，使用 k 命令可以看到它的执行过程：

```
0:004> kn
 # ChildEBP RetAddr
00 0106ffc8 7c9507a8 ntdll!DbgBreakPoint
01 0106fff4 00000000 ntdll!DbgUiRemoteBreakin+0x2d
```

上面讲的是正常情况，也就是创建一个远程线程，这个远程线程一执行就产生一个断点异常，触发调试目标中断到调试器中。

但是以上过程有时可能失败，比如远程线程创建后还没有执行断点指令就被挂起了。这时 WinDBG 就收不到断点事件了。WinDBG 等待一段时间后，会显示如下提示信息：

```
Break-in sent, waiting 30 seconds...
```

再等待 30 秒后，WinDBG 会“人工合成”（ Synthesize）一个代码为 0x80000007 的异常事件，合成这个事件的调试引擎函数名为 SynthesizeWakeEvent，调用过程如下：

```
0:002> kn
 # ChildEBP RetAddr
00 015dff10 020ceacf dbgeng!SynthesizeWakeEvent+0x4b
01 015dff34 020cee9e dbgeng!WaitForAnyTarget+0x5f
02 015dff80 020cf110 dbgeng!RawWaitForEvent+0x2ae
03 015dff98 0102aadf dbgeng!DebugClient::WaitForEvent+0xb0
04 015dffb4 7c80b6a3 windbg!EngineLoop+0x13f
05 015dffec 00000000 kernel32!BaseThreadStart+0x37
```

这个合成事件触发调试器的事件等待函数返回，并开始处理这个事件，这会引发 dbgeng!SuspendExecution 函数被调用，这个函数会依次挂起调试目标中的所有线程，使调试目标被中断。最后调试器进入到命令模式中并显示如下信息：

```
(1440.1554): Wake debugger - code 80000007 (first chance)
```

另两种方式的工作原理于第一种类似，都是使调试目标因为发生调试事件而被调试子系统挂起，调试子系统把调试事间发送给调试器，后者收到调试事件后进入命令模式。

30.10.3 单步执行

单步执行是最早的调试机制之一，可以追溯到小型机时代，例如 PDP 系列小型机的控制面板上就有一个按钮可以切换到单步执行模式，让处理器每次执行一条指令。通过单步执行可以更好的理解程序的执行流程和数据的变化情况。但是在软件日益庞大的今天，单步跟踪整个程序很多情况下都是不现实的。可行的方法是只让最关心的代码单步执行，让不关心的部分全速执行。WinDBG 调试器为了实现这一目标提供了丰富的单步跟踪命令，让调试人员可以迅速的抵达要跟踪的位置，或者快速跳过不关心的部分。

首先，根据根据当前是否处于源代码模式（Source Mode），单步跟踪分为源代码级的单步和汇编指令一级的单步。选中 Debug 菜单的 Source Code 菜单项或者执行 l+t 命令可以进入源代码模式，反选 Source Code 菜单项或者执行 l-t 命令可以退出源代码模式，进入汇编模式。

在汇编模式，每次单步执行执行一条汇编指令。在源代码模式时，每次单步执行执行源代码的一行。

如果当前的指令或者代码行包含函数调用，那么有两种选择，一种是跟踪进入要调用的函数，另一种是胡略要调用函数的执行过程，让其执行完毕后再停下来。前一种方式通常称为单步进入（Step Into），后一种方式称为单步越过（Step Over）。在 WinDBG 中，使用单步（Step）一词来指代前一种方式，对应的命令是 p；使用跟踪（Trace）一词指代后一种方式，对应的命令是 t。

如果当前行不包含函数调用，那么 p 命令和 t 命令的作用是一样的。

从实现的角度来讲，汇编一级的单步执行（p）、单步进入（Trace Into）和针对非函数调用执令执行 t 命令都是依赖 CPU 的陷阱机制来实现的，对于 x86 CPU，也就是设置标志寄存器的 T 标志。

举例来说，当我们在 WinDBG 中发出 t 命令后，工作线程会解析此命令（ParseStepTrace），然后调用 SetExecStepTrace 和 SetNextStepTraceState 函数，这两个函数会将用户命令转化并设置给内部对象、变量以及线程的上下文结构中。其过程如清单 30-1 所示。

清单 30-1 解析和执行单步命令的过程

```
0:004> kn
 # ChildEBP RetAddr
00 0144e720 021f51a3 dbgeng!SetNextStepTraceState
01 0144e760 021f44dd dbgeng!SetExecStepTrace+0x293
02 0144e8f0 02187feb dbgeng!ParseStepTrace+0x58d
03 0144e9d8 021889a9 dbgeng!ProcessCommands+0xf1b
04 0144ea1c 020cbec9 dbgeng!ProcessCommandsAndCatch+0x49
05 0144eeb4 020cc12a dbgeng!Execute+0x2b9
06 0144eee4 01028553 dbgeng!DebugClient::ExecuteWide+0x6a
07 0144ef8c 01028a43 windbg!ProcessCommand+0x143
08 0144ffa0 0102ad06 windbg!ProcessEngineCommands+0xa3
09 0144ffb4 7c80b6a3 windbg!EngineLoop+0x366
0a 0144ffec 00000000 kernel32!BaseThreadStart+0x37
```

在设置完毕后，调试引擎便报告此命令执行完毕，于是函数层层返回。直到 ProcessEngineCommands 返回到 EngineLoop。EngineLoop 继续执行，调用 DebugClient::WaitForEvent 等待下一个调试事件，在等待前先要恢复目标运行，在恢复目标执行时（ResumeExecution），调试引擎会将线程上下文通过 SetThreadContext API 设置给系统。

清单 30-2 通过设置线程上下文设置单步标志

```
0:004> kn
 # ChildEBP RetAddr
00 0144f350 0229ba85 kernel32!SetThreadContext
01 0144f368 020f2de9 dbgeng!LiveUserDebugServices::SetContext+0x45
02 0144f38c 020f0bf8 dbgeng!LiveUserTargetInfo::SetTargetContext+0x59
03 0144fe24 021244fe dbgeng!TargetInfo::SetContext+0xb8
04 0144fe40 0217257c dbgeng!MachineInfo::UdSetContext+0x3e
05 0144fe54 0221bbfa dbgeng!MachineInfo::SetContext+0x15c
06 0144fe6c 0221c82d dbgeng!TargetInfo::ChangeRegContext+0xca
07 0144fe84 021311f8 dbgeng!TargetInfo::PrepareForExecution+0x1d
08 0144fe94 0213148f dbgeng!PrepareOrFlushPerExecution+0x38
09 0144fea4 02132098 dbgeng!ResumeExecution+0x2f
0a 0144ff28 02130fe8 dbgeng!PrepareForExecution+0x618
0b 0144ff3c 020cec09 dbgeng!PrepareForWait+0x28
```

```
0c 0144ff80 020cf110 dbgeng!RawWaitForEvent+0x19
0d 0144ff98 0102aadf dbgeng!DebugClient::WaitForEvent+0xb0
0e 0144ffb4 7c80b6a3 windbg!EngineLoop+0x13f
0f 0144ffec 00000000 kernel32!BaseThreadStart+0x37
```

以下是传递给 SetThreadContext 函数的参数，它是一个 CONTEX 结构：

```
0:004> dt _CONTEXT 01473e00
ntdll!_CONTEXT
   +0x000 ContextFlags     : 0x1003f
…
   +0x0b8 Eip              : 0x7c93edc5
   +0x0bc SegCs            : 0x1b
   +0x0c0 EFlags           : 0x302
   +0x0c4 Esp              : 0x12fb24
   +0x0c8 SegSs            : 0x23
   +0x0cc ExtendedRegisters : [512]  "???"
```

其中 EFlags 就是标志寄存器字段，将它的值翻译为二进制：

```
0:004> .formats 0x302
Evaluate expression:
  …
  Binary:  00000000 00000000 00000011 00000010
```

位 1 是保留位，永远为 1，位 8 就是跟踪标志（Trap Flag），1 代表单步执行。另一个 1，位 9 的含义是启用中断（Interrupt Enable Flag），即 IF 位。

对 dbgeng!LiveUserTargetInfo::ProcessDebugEvent 设置一个断点，当期命中时，观察它的参数可以分析发生的调试事件类型。

```
0:004> kv
ChildEBP RetAddr  Args to Child
0144fda8 02136581 0144fe50 00000000 00000002
dbgeng!LiveUserTargetInfo::ProcessDebugEvent
```

其中 **0144fe50** 是 DEBUG_EVENT 结构，通过 dd 命令显示其值：

```
0:004> dd 0144fe50
0144fe50   00000001 000013f4 000015d8 00000000
0144fe60   80000004 00000000 00000000 00000000
0144fe70   7c903912 00000000 00000000 00000000
```

其中，00000001 是事件代码，即 EXCEPTION_DEBUG_EVENT，000013f4 是目标程序的进程 ID，000015d8 是目标程序的线程 ID，即触发调试事件的进程和线程。80000004 是异常代码，即单步异常。

如果当前指令是 CALL 指令，而且执行 p 命令，那么 WinDBG 需要一步执行好要调用的函数，即单步越过，这时尽管命令的执行过程仍然与清单 30-1 和 30-2 所示的基本一致，但是设置的线程上下文中的 EFlags 值为：

```
+0x0c0 EFlags           : 0x246
```

也就是 TF 标志没有设置。当 ProcessDebugEvent 处的断点命中时，发生的异常代码是 80000003，也就是断点异常，不再是单步异常。

```
0:004> dd 0144fe50
0144fe50   00000001 000013f4 000015d8 00000000
```

0144fe60 80000003 00000000 00000000 00000000

这充分说明，单步越过命令是通过在下一条指令处设置一个软件断点来实现的。

源代码一级的单步执行是通过多次设置陷阱标志，也就是多次汇编一级的单步执行而实现的。因为篇幅关系，不再详细讨论，感兴趣的读者可以使用上面的方法自己来探索。

可以通过向 p 和 t 命令附带参数来使用它们的附加功能：

p|t [r] [= StartAddress] [Count] ["Command"]

其中 r 的用处是禁止自动显示寄存器内容，缺省情况下每次单步执行后，WinDBG 会自动显示各个寄存器的值，如果不想显示，则使用 r 开关，r 与命令之间的空格可有可无。

缺省情况下，调试器总是让目标程序从当前位置开始单步执行，但是也可以通过等号（=）来指定一个新的起始地址，让程序从这个地址开始执行。需要注意的是，如果指定的地址超出了当前的函数，那么用于跳过了调整栈的代码，那么栈会失去平衡，目标程序很快会出现严重错误。所以使用这个功能时应该特别慎重。

可选的参数[Count]用来指定要单步执行的次数。如果 Count 大于 1，那么执行号一次单步并更新显示后，WinDBG 会再发送一次单步命令，直到达到指定的次数。例如：t 2 会单步执行两次，每次执行一条指令（汇编模式）或者源代码的一行（源代码模式）。

["Command"]参数用来指定每次单步执行后要执行的命令。例如：p “kb”会在单步执行后自动执行 kb 命令。

30.10.4 单步执行到指定地址

WinDBG 提供了 pa 和 ta 命令用来执行到指定的代码地址，其命令格式为：

pa|ta [r] [= StartAddress] StopAddress

其中 pa 是 Step to Address 的缩写，即单步执行到 StopAddress 参数所代表地址处的指令，如果中间有函数调用，那么不进入所调用的函数。在执行过程中，WinDBG 会显示程序执行的每一步，其效果相当于反复的执行 p 命令。

Ta 命令与 pa 非常相似，只不过遇到函数调用时会进入到函数中，而不是越过。这和 t 命令与 p 命令的差异是一样的。

因为伪寄存器$ra 总是代表当前函数的返回地址（return address），因此可以使用使用 pa 或者 ta 命令加上@$ra 来“步出”当前函数，也就是从当前位置反复单步直到返回到上一级函数。其效果相当于我们下面要介绍的 gu 命令（执行到上一层函数）。

如果在到达目标地址前遇到断点，那么 WinDBG 会报告断点，这个命令也就从此被中断。

如果在到达目标地址前程序发生异常，那么 pa 或者 ta 命令也可能被中断。举例来说，假设当前的程序指针（EIP）等于 004113b4，即执行标记位于如下指令上：

004113b4 c7050000000001000000 mov dword ptr ds:[0],1 ds:0023:00000000=????????

004113be 8bf4 mov esi,esp

004113c0 683c564100 push offset dbgee+0x1563c (0041563c)

如果这时发出 par 004113c0 命令（r 告诉调试器不要显示寄存器，以减少显示内容节约篇幅），那么其执行结果如下：

0:000> par 004113c0

(13f4.15d8): Access violation - code c0000005 (first chance)

ntdll!KiUserExceptionDispatcher+0x4:

7c90eaf0 8b1c24 mov ebx,dword ptr [esp] ss:0023:0012fbac=0012fbb4

【省略 20 行】

ntdll!KiUserExceptionDispatcher+0x24:
7c90eb10 e83df7ffff call ntdll!NtRaiseException (7c90e252)
(13f4.15d8): Access violation - code c0000005 (!!! second chance !!!)
dbgee+0x113b4:
004113b4 c7050000000001000000 mov dword ptr ds:[0],1 ds:0023:00000000=????????

接到 par 命令后，WinDBG 设置标志寄存器的 TF 位，然后恢复目标程序执行，但因为所在指令向地址 0 写数据 1，所以会导致访问异常。因此执行结果的第一行便是 WinDBG 接收到异常的第一轮机会（first chance）而显示的信息。因为对第一轮机会 WinDBG 的处理方式是立刻返回不处理，因此系统继续分发异常，即交给用户态的 KiUserExceptionDispatcher 开始寻找异常处理代码，因为被调试的程序（dbgee.exe）没有异常处理器，而且程序在被调试，所以 KiUserExceptionDispatcher 通过 NtRaiseException 触发系统开始第二轮异常分发，第二轮分发，也是先给调试器来处理，WinDBG 收到事件后显示出倒数第三行的通知信息。因为对于第二轮机会，调试器的默认处理方式是中断给用户，因此 WinDBG 进入到命令方式，停下来等待用户处理。至此，pa 命令也执行完毕。

30.10.5 单步执行到下一个函数调用

与 pa 和 ta 命令类似，pc 和 tc 命令用来单步执行到下一个函数调用指令（CALL）。

pc|tc [r] [= StartAddress] [Count]

pc 或 tc 命令都是让调试目标从当前地址或者 StartAddress 指定的地址恢复执行，直到遇到函数调用指令时停下来。Count 参数用来指定遇到的函数调用指令个数，缺省为 1。

这两个命令的差异依然与 p 指令的和 t 指令的差异一样，如果当前所在指令是函数调用指令，那么使用 tc 时会单步进入所调用的函数，如果使用 pc 命令那么会一次执行完这个函数。如果当前指令不是函数调用指令，那么 pc 与 tc 是等价的。

从实现角度来看，WinDBG 依然是反复单步执行，每次收到调试事件后，判断下一条指令是否是函数调用指令，如果不是就重新设置单步标志，然后继续执行，如果是那么就停下来，命令执行完毕。当处理 CALL 指令时（当前指令是 CALL 或者 Count 参数大于 1），pc 命令需要使用在下一条指令设置断点的方法。

30.10.6 单步执行到下一分支

在第 2 篇介绍 CPU 一级的调试支持时，我们介绍了 CPU 的分支记录和监视功能。利用这一功能可以实现分支到分支的单步执行，即一次执行到下一个分支指令。WinDBG 的 tb 命令就是利用这一机制而实现的。

因为使用了 CPU 的硬件支持，所以 tb 命令与 tc 和 pc 这样的反复多次单步执行不同，它是让设置好标志寄存器和 MSR 寄存器后，便让目标程序恢复运行，然后当 CPU 执行到分支指令时，报告异常停下来。从这个意义上来说，tb 命令要比 tc 和 pc 更高效。

Tb 命令的语法与 tc 和 pc 一样：

tb [r] [= StartAddress] [Count]

在 x86 平台上这个命令只能用户内核调试，对安腾系统和 x64 系统，这个命令既可以用在内核调试，也可以用在用户态调试。

30.10.7继续运行

WinDBG 提供了几个命令来让调试目标恢复继续运行。最常用的就是 g（go）命令。热键 F5 和 Debug 菜单的 Go 菜单项对应的就是 g 命令。G 命令的一般形式为：

g[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]

其中的 StartAddress 用来指定恢复执行的起始地址，缺省为当前位置。BreakAddress 用来指定一个断点地址，BreakCommands 用来指定断点命中后所执行的命令。开关 a 只有在当使用 BreakAddress 设置断点时才有用，使用 a 将断点设置为硬件断点，如果没有 a，则设置为软件断点。

如果不带任何参数，那么 g 命令就是恢复目标运行，直到它遇到断点或者发生异常等调试事件时才能再被中断到调试器中。

如果指定了断点地址（BreakAddress），那么 WinDBG 会设置一个隐藏的断点，然后恢复目标执行，当执行到这个断点时，WinDBG 中断并自动删除这个断点。当使用汇编或者源代码窗口时，可以使用“运行到光标处”（Run to Cursor）命令（Ctrl+F10 或者菜单 Debug > Run to Cursor）来执行到光标所在位置。这其实就是使用 g 命令加断点地址来实现的。当发出这个命令时，命令信息区会显示出对应的 g 命令，如 0:000> g 0x`4113c0。

对于因为异常而中断到调试器的情况，当恢复目标运行时调试器需要回复系统是否处理了这个异常，如果是使用 g 命令，那么 WinDBG 会使用关于所发生异常的配置信息来决定回复内容。为了提高灵活性，用户可以使用 gn 或者 gh 命令来指定要回复给系统的异常处理决定，如果要回复已经处理，那么就是用 gh（Go with Exception Handled）命令；否则就使用 gn（Go with Exception Not Handled）命令。这两条命令的语法和其它特征与 g 命令完全一样。

除了以上介绍的 g、gh 和 gn 命令，gu（go up）命令用来执行到上一级函数，即执行完当前函数，返回到上一级函数。另外 gc 命令用在使用条件断点的情况，我们将在下一节介绍。Gu 和 gc 命令都没有参数。

30.10.8追踪并监视

如果我们想了解一个函数的执行路径和它调用了哪些其它函数，每个函数包含了多少条指令，但我们又不想一步步的跟踪执行，那么可以使用 wt 命令让它帮我跟踪执行并生成一份报告给我们。

下面通过一个例子来解释 wt 命令的用法。清单 30-8 是当执行位置在 dbgeng!TargetInfo::SetContext 函数的起始处（第一条指令）时执行 wt 命令所得到的结果。

清单 30-8 wt 命令产生的关于 TargetInfo::SetContext 函数的追踪报告

```
0:001> wt
Tracing dbgeng!TargetInfo::SetContext to return address 021244fe
   30     0 [  0] dbgeng!TargetInfo::SetContext
   10     0 [  1]   dbgeng!LiveUserTargetInfo::SetTargetContext
   14     0 [  2]     dbgeng!ProcessInfo::CanModifyContexts
   31    14 [  1]   dbgeng!LiveUserTargetInfo::SetTargetContext
   17     0 [  2]     dbgeng!LiveUserDebugServices::SetContext
    6     0 [  3]       kernel32!SetThreadContext
    1     0 [  4]         ntdll!NtSetContextThread
    2     0 [  4]         ntdll!ZwSetContextThread
```

```
    2     0 [  5]           ntdll!KiFastSystemCall
    1     0 [  4]         ntdll!ZwSetContextThread
   12     6 [  3]       kernel32!SetThreadContext
   23    18 [  2]     dbgeng!LiveUserDebugServices::SetContext
   34    55 [  1]   dbgeng!LiveUserTargetInfo::SetTargetContext
   33    89 [  0] dbgeng!TargetInfo::SetContext

122 instructions were executed in 121 events (0 from other threads)

Function Name                                  Invocations MinInst MaxInst AvgInst
dbgeng!LiveUserDebugServices::SetContext               1      23      23      23
dbgeng!LiveUserTargetInfo::SetTargetContext            1      34      34      34
dbgeng!ProcessInfo::CanModifyContexts                  1      14      14      14
dbgeng!TargetInfo::SetContext                          1      33      33      33
kernel32!SetThreadContext                              1      12      12      12
ntdll!KiFastSystemCall                                 1       2       2       2
ntdll!NtSetContextThread                               1       1       1       1
ntdll!ZwSetContextThread                               2       1       2       1

1 system call was executed

Calls  System Call
    1  ntdll!KiFastSystemCall

eax=00000000 ebx=00000000 ecx=00000001 edx=ffffffff esi=015b2008 edi=000d5dc8
eip=021244fe esp=014cfe3c ebp=014cfe40 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
dbgeng!MachineInfo::UdSetContext+0x3e:
021244fe 8be5            mov     esp,ebp
```

可以把 wt 命令的结果分成 6 个部分。第一部分是标题（第 2 行），显示了所追踪的函数名，和追踪的结束地址，即函数的返回地址。第二部分是详细的执行情况表，包含如下 4 列：

第 1 列为指令数，当这一行所描述的函数是父函数的入口部分，那么这一列的数字就是这个函数从入口进入到下一行所对应的函数的入口（也就是执行好 CALL 指令）所执行的指令数，比如第一行的数字 30 代表从 SetContext 函数的入口到进入第二行中的 SetTargetContext 函数共执行了 30 条指令。如果所在行描述的是函数的出口，那么第 1 列的数字是本次调用这个函数一共执行的指令数，例如 16 行的数字 33 表示本次执行 TargetInfo::SetContext 函数共执行了 33 条指令（不包括调用其它函数）。如果所在行描述的函数没有调用其它函数（既是入口又是出口），比如第 5 行，那么就是总共执行这个函数的指令数。

第 2 列用来显示本行所对应的函数调用其它函数时所执行的总指令数。以第 6 行为例，14 表示 SetTargetContext 函数调用其它函数（即 CanModifyContexts）执行了 14 条指令。

第 3 列用来表示函数调用深度。被追踪的函数的深度为 0（第一行和最后一行），每进入一个函数深度加一，每返回一次，深度减一。

第 4 列为函数名称，名称前的缩进用来表示调用深度。

Wt 结果的第三部分（第 18 行）是执行归纳，本次共执行了 122 条指令，共处理了 121 次调试事件。值得注意的是，总指令数并不是第一列所有数字的直接求和，而应该是只汇总函数出口行的数字，即我们加下划线的数字。

Wt 结果的第四部分（20 到 28 行）是函数指令数列表，对于执行过的每个函数，分别显示出调用它的次数（Invocations），每次调用时执行的最少指令数（MinInst），最多指令数（MaxInst），和平均指令数（AvgInst）。因为本例中，大多数函数都是调用一次，所以后三列的数字大多相同。

Wt 结果的第五部分（30 到 33 行）是系统调用情况。

第六部分（35 到 39 行）是追踪执行完成后的寄存器状态和当前程序指针位置，显示的是函数返回到上一级函数后即将执行的下一条指令。

值得说明的是，上面关于第 4 层和第 5 层函数的显示（9 到 12 行）是不准确的。对于 NtSetContextThread 反汇编，可以看到其汇编代码如下：

```
ntdll!NtSetContextThread:
7c90e4f2 b8d5000000      mov     eax,0D5h
7c90e4f7 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
7c90e4fc ff12            call    dword ptr [edx]
7c90e4fe c20800          ret     8
```

从以上代码可以看出，NtSetContextThread 执行 3 条指令后进入 SystemCallStub 变量所指向的函数，因此第 9 行第 1 列应该是 3。

另外，观察 SystemCallStub 变量的值：

```
0:001> dd SharedUserData!SystemCallStub l1
7ffe0300   7c90eb8b
```

看起所对应的函数：

```
0:001> ln 7c90eb8b
(7c90eb8b)    ntdll!KiFastSystemCall    |    (7c90eb94)    ntdll!KiFastSystemCallRet
Exact matches:
    ntdll!KiFastSystemCall = <no type information>
```

这意味着，NtSetContextThread 是直接调用 KiFastSystemCall 的，没有经过第 10 行所显示的 ZwSetContextThread。

那么 WinDBG 为什么会认为调用了 ZwSetContextThread 函数了呢？笔者认为这是由于地址 7c90e4f2 有两个准确匹配的符号：

```
0:001> ln 7c90e4f2
(7c90e4f2)    ntdll!NtSetContextThread    |    (7c90e507)    ntdll!ZwSetDebugFilterState
Exact matches:
    ntdll!NtSetContextThread = <no type information>
    ntdll!ZwSetContextThread = <no type information>
```

当单步执行完 NtSetContextThread 函数的第一条指令后，WinDBG 搜索下一条指令所邻近的函数，即与地址 7c90e4f7 的最近函数：

```
0:001> ln 7c90e4f7
(7c90e4f2)    ntdll!ZwSetContextThread+0x5    |    (7c90e507)    ntdll!ZwSetDebugFilterState
```

问题就出在这里，搜索结果竟然是 ZwSetContextThread，而不是 NtSetContextThread。这会让 WinDBG 误以为已经退出了 NtSetContextThread，并且进入了 ZwSetContextThread 函数，于是它把前者的指令数统计为 1，并把接下来的指令算给 ZwSetContextThread 函数。

纠正以上错误，9 到 12 行应该为如下内容：

```
3      0 [  4]          ntdll!NtSetContextThread
```

```
2      0 [  5]            ntdll!KiFastSystemCall
4      2 [  4]          ntdll!NtSetContextThread
```

第 28 行的显示也是不必要的，其指令数应该归于 NtSetContextThread 函数。不过以上错误不影响总指令数，因为指令数是通过单步事件的次数统计出来的。

如果不是在函数开始处执行 wt 命令，那么它的效果相当于 p 命令，只是单步执行一次。

30.10.9程序指针飞跃

在前面介绍的单步命令和 g 命令中，都可以指定起始执行地址。如果指定了起始地址，那么 WinDBG 便会把这个地址设置到线程上下文中的程序指针寄存器中。这样一来，当目标程序恢复执行时，那么系统就会把这个地址放入到真正的程序指针寄存器（EIP）中，于是 CPU 也就从这个地址开始执行了。这意味着，程序一下子“飞跃”到这个地址。这个飞跃不是目标程序的代码所定义的，而是我们通过调试器来操纵的。我们把这种特殊的跳转称为“程序指针飞跃”。

在某些情况下程序指针飞跃可以帮助我们调试。比如在调试时，如果我们现在不想执行某个函数调用，那么就可以通过程序指针飞跃来“飞过”这个函数。或者有时我们想跳过一条导致异常的指令，那么也可以从其下一条地址恢复执行而绕过它。但是我们必须知道，这种飞跃是很危险的，如果所跨跃的代码包含栈操作，那么很容易导致栈不平衡，从而是目标程序无法继续运行。

除了通过在恢复程序执行的这些命令中指定起始地址外，也可以通过修改寄存器命令（r）来直接修改程序指针寄存器来实现程序指针飞跃。

30.10.10　归纳

本节比较详细的介绍了用来控制目标程序执行的 WinDBG 命令，表 30-5 将这些命令列在了一起。

表 30-5 控制目标程序执行的命令一览

命令	含义	说明
p	Step	单步，一次执行完函数调用
t	Trace	追踪，如果遇到函数调用，则进入被调用函数
pa	Step to Address	单步到指定地址，不进入子函数
ta	Trace to Address	追踪到指定地址，进入子函数
pc	Step to next Call	单步执行到下一个函数调用（CALL 指令）
tc	Trace to next Call	追踪执行到下一个函数调用（CALL 指令）
tb	Trace to next branch	追踪执行到下一条分支指令
g	Go	恢复运行
gh	Go Handled	恢复运行，告诉系统已经处理异常
gn	Go Not handled	恢复运行，告诉系统没有处理异常
gu	Go Up	执行到本函数返回

以上命令除了 tb 命令在 x86 系统中只能在内核调试时使用，其它命令都是既可以在用户态调试使用，又可以在内核态调试时使用，但是调试目标不能是转储文件，而必须是活动目标。

除了以上标准命令，WinDBG 还设计了一些元命令和扩展命令来辅助以上命令的使用，比如在连续单步跟踪时，如果觉得界面更新太频繁，映像速度并且叫人眼花缭乱，那么可以使

用.suspend_ui 命令暂时停止刷新信息窗口。

30.11 使用断点

断点是软件调试的最常用技术之一。设置断点看似简单，但是如何把断点运用的恰到好处还必须认真学习不同类型的断点的特征，并仔细琢磨目标程序的执行流程，然后才能在合理的位置设下断点，并且这个断点在合适的时机命中。如果断点设置的位置不好，那么它要么频繁命中，不断的中断到调试器，然后再恢复执行，浪费大量时间，要么根本不命中，让人空等一场。有人把设置断点称为“埋断点”，仿佛就像“埋地雷”，我觉得非常形象生动。埋地雷要讲究位置，埋伏和期望被踩上（命中）的时间，埋断点也一样。要设置一个好的断点既要考虑空间性（断点位置），又要考虑时间性（命中时机），并非易事。

为了支持以上目标，WinDBG 提供了多种断点机制，让满足不同的调试需要，下面我们分别进行介绍。

30.11.1 软件断点

我们在前面很多章节中提到了软件断点。简单来说软件断点就是通过将指定位置的指令替换为断点指令（INT 3）来设置的断点。

WinDBG 设计了三条命令来设置软件断点，分别是 bp、bu 和 bm。其中 bp 是基本的而且最常用的，其命令格式如下：

bp[ID] [Options] [Address [Passes]] ["CommandString"]

其中的 ID 用来指定断点编号，如果不指定，那么 WinDBG 会自动从 0 开始编排。Options 用来指定选项，我们稍后再讨论。Address 用来指定断点的地址，如果不指定，那么默认使用当前程序指针所代表的地址。Passes 用来指定经过这个断点而停下来的次数，其缺省值为 1，也就是经过一次就中断给用户，如果这个值大于 1，那么每次 WinDBG 收到经过这个断点地址的事件时，便会把这个计数递减 1，然后判断其值是否等于 0，如果大于 0，那么便直接让程序恢复执行，直到等于 0 时才进入命令模式中断给用户看。"CommandString"用来指定一系列命令，当断点中断时 WinDBG 会自动执行这些命令。应该使用双引号将命令包围起来，多个命令用分号分隔。

例如以下 bp 命令对 C 运行库的 printf 函数的入口偏移 3 的地址设置断点，当第二次经过这个位置时中断给用户，并自动执行 kv 和 da poi(ebp+8)命令。

bp MSVCR80D!printf+3 2 "kv;da poi(ebp+8)"

其中 kv 命令用来显示函数调用序列，da poi(ebp+8)用来显示 printf 的第一个参数所指定的字符串。之所以要对入口偏移 3 的位置设置断点，而不是入口，是因为要等入口处的栈帧建立代码执行好后，ebp+8 才能指向第一个参数。

```
MSVCR80D!printf:
1022e3f0 55              push    ebp
1022e3f1 8bec            mov     ebp,esp
1022e3f3 6afe            push    0FFFFFFFEh
```

Bu 命令用来设置一个延迟的以后再求解的断点，用于对尚未加载模块中的代码设置断点。当指定的模块被加载时，WinDBG 会真正落实（解决）这个断点。所以 bu 命令对于调试动态加载模块的入口函数或者初始化代码特别有用。例如，在调试即插即用的驱动程序时，因为驱动程序是由操作系统的 I/O 管理器动态加载的，当我们发现它加载后，它的入口函数（DriverEntry）和初始化代码已经执行完了。这时就可以使用 bu 命令来在这个驱动加载前就设置一个断点，即 bu

MyDriver!DriverEntry。

Bm 命令用来设置一批断点，相当于帮我们自动执行很多次 bp 或者 bu 命令。比如以下命令对于 msvcr80d 模块中的所有 print 开头的函数设置断点：

```
0:000> bm msvcr80d!print*
  2: 1022e3f0 @!"MSVCR80D!printf"
  3: 1022e590 @!"MSVCR80D!printf_s"
```

因为 bm 命令在设置断点前需要确认匹配的符号对应的是代码而不是数据，所以使用 bm 命令时要求目标模块的调试符号有类型信息，能够判断出一个符号的类型。这通常需要所谓的私有符号文件，也就是调试版本的符号文件。对于公共的符号文件，尽管我们可以看到函数符号存在，比如 ntdll 中有多个以 DbgPrint 开头的符号：

```
0:000> x ntdll!DbgPrint*
7c95881a ntdll!DbgPrintReturnControlC = <no type information>
7c9303f0 ntdll!DbgPrint = <no type information>
7c91eb25 ntdll!DbgPrintEx = <no type information>
```

但是当我们对其使用 bm 命令时，bm 命令会显示错误信息，并没有设置断点：

```
0:000> bm ntdll!DbgPrint*
No matching code symbols found, no breakpoints set.
If you are using public symbols, switch to full or export symbols.
```

解决这个问题的一种简单方法是使用/a 开关，这个开关告诉 bm 命令不管符号对应的位置是数据还是代码都设置断点，这样做是有风险的，建议只有在确信所有符号都是函数时才使用。

另一种更可靠的解决方式是按最后一句提示的建议使用完全的符号或者函数输出符号。完全符号不去讲了。下面我们看使用 DLL 文件的输出信息，输出信息尽管包含的符号较少，但是可以判断出是代码还是数据。为了让 WinDBG 胡略 PDB 文件而使用 DLL 输出信息，我们将符号路径设置为一个没有符号的路径（.sympath .），然后使用.reload 命令刷新模块和符号。而后再使用 x 命令检查符号，触发 WinDBG 使用输出符号：

```
0:000> x ntdll!dbgprint*
7c91eb25 ntdll!DbgPrintEx (<no parameter info>)
7c9303f0 ntdll!DbgPrint (<no parameter info>)
7c95881a ntdll!DbgPrintReturnControlC (<no parameter info>)
```

可见此时的符号信息不再抱怨没有符号信息（no type information），尽管还是缺参数信息。此时再使用 bm 命令：

```
0:000> bm ntdll!dbgprint*
  4: 7c91eb25 @!"ntdll!DbgPrintEx"
  5: 7c9303f0 @!"ntdll!DbgPrint"
  6: 7c95881a @!"ntdll!DbgPrintReturnControlC"
```

就能够成功设置一批断点了。

Bm 和 bu 命令的格式与 bp 类似：

```
bu[ID] [Options] [Address [Passes]] ["CommandString"]
bm [Options] SymbolPattern [Passes] ["CommandString"]
```

其中的 Options 可以为以下内容：

/1 如果指定此选项，那么这个断点命中一次后便会便自动从断点列表中删除。这种断点被称为一次命中断点。

/p 这个开关只能用在内核调试中，/p 后跟一个进程的 EPROCESS 结构，作用是只有当前进程是指定进程时才触发这个断点。

/t 与/p 开关类似，只能用在内核调试中，用来指定一个 ETHREAD 结构，作用是只有在执行指定的线程访问断点地址时才触发断点。

/c 和/C 这两个开关后面可以带一个数字，用来指定中断给用户的最大函数调用深度和最小函数调用深度。举例来说，使用命令 bp /c5 msvcr80d!printf 设置的断点只有当函数调用深度浅于 5 时才中断给用户。

以上命令选项对于下面介绍的硬件断点命令也是适用的。

30.11.2 硬件断点

硬件断点就是通过 CPU 的硬件寄存器设置的断点。硬件断点具有数量限制，但是可以实现软件断点不具有的功能，比如监视数据访问和 I/O 访问等（详见第 3 章）。

WinDBG 的 ba 命令用来设置硬件断点，其格式如下：

ba[ID] Access Size [Options] [Address [Passes]] ["CommandString"]

其中 ID 用来指定断点的序号，这 bp 命令一样，通常不需要指定序号，WinDBG 会自动编排。Access 用来指定触发断点的访问方式，可以为以下几个字母之一：

e – 当从指定地址读取和执行指令时，触发断点。这种断点又称为访问代码硬件断点。从效果上来看，这种断点与软件断点的效果是类似的，都是针对执行代码时触发断点。但是硬件断点的好处是不需要做指令替换和恢复。

r - 当从指定地址读取和写入数据时，触发断点。

w – 当向指定地址写数据时触发断点。通过 r 和 w 选项设置的断点又称为访问数据断点。

i – 当向指定的地址执行输入输出访问（I/O）时触发断点，比如对于 x86 架构，当执行 IN 或者 OUT 指令时。这种断点又称为访问 I/O 断点。

Size 参数用来指定访问的长度，对于访问代码硬件断点，它的值应该为 1。对于其它硬件断点，允许的长度值因为平台的不同而不同，对于 x86 系统，可以为 1、2、4 三种值，分别代表对指定地址的 1 字节访问、字访问和双字访问。对于 x64 系统可以为 1、2、4、8（四字访问）四种值。对于安腾系统，可以 1 到 0x80000000 间任何 2 的次方值。如果对指定地址的实际访问长度大于断点定义的访问长度，断点仍会命中。举例来说，对于如下命令设置的断点：

ba r1 0041717c

那么对内存地址 0041717c 的一字节访问、字访问、双字访问（读写）都会触发这个断点。

Address 参数用来指定断点的地址。需要注意的是，地址值一定是按照 Size 参数的值做内存对齐的。比如如果 Size 中是 4，那么地址值应该是按 4 字节做内存对齐的，也就是它的值应该是 4 的整数倍。

Passes 参数和 CommandString 参数的用法与设置软件断点命令中的一样。

硬件断点是设置在 CPU 的调试寄存器中的，在 x86 CPU 中就是 DR0～DR7。比如当我们设置上面的断点并让调试目标恢复执行，然后再将其中断到调试器时，可以看到 dr0 和 dr7 的寄存器内容为：

0:000> r dr0,dr6,dr7

dr0=0041717c dr6=ffff0ff0 dr7=00030501

dr0 就是断点的地址值，dr6 是断点的状态寄存器，dr7 是断点的控制寄存器，详见第 4 章关于这些寄存器的介绍。

因为硬件断点要占用 CPU 的调试寄存器，所以硬件断点的数量是很有限的。但是因为在恢复目标执行时才会把断点落实到上下文的寄存器中，所以在发出 ba 命令时，即使超出了硬件断点的数量限制，WinDBG 也不会报告错误。只有当恢复目标执行时，它才报告错误：

0:000> g

Too many data breakpoints for thread 0
bp8 at 00417180 failed
Too many data breakpoints for thread 0
bp9 at 00417184 failed
WaitForEvent failed

上面的信息的含义是数据断点个数太多，WaitForEvent 调用失败，这时 WinDBG 会返回到命令模式。

最后要说明的一点是，当初始断点命中时，尚不能设置硬件断点，如果设置，那么会得到如下错误：

0:000> ba r1 kernel32!BasepCurrentTopLevelFilter
^ Unable to set breakpoint error
The system resets thread contexts after the process
breakpoint so hardware breakpoints cannot be set.
Go to the executable's entry point and set it then.
'ba r1 kernel32!BasepCurrentTopLevelFilter'

30.11.3 条件断点

在调试时，如果要分析的代码或者变量被多次执行和访问，那么对它设置的断点就会反复命中，而我们可能只关心特定条件时的命中，而不关心其它情况。为了避免断点反复命中而浪费时间，可以使用条件断点。当断点发生时，让调试器检查一个条件，对于不关心的情况，立刻恢复目标执行，只有用户关心的情况发生时才中断给用户。

我们前面介绍的软件断点命令和硬件断点命令都支持在断点中指定一系列命令。可以通过编写这些命令来设置条件断点。常见的方式有以下两种，一种是使用 j 命令：

bp|bu|bm|ba Address "j (Condition) 'OptionalCommands'; 'gc' "

另一种是使用.if 命令：

bp|bu|bm|ba Address ".if (Condition) {OptionalCommands} .else {gc}"

其中 Condition 用来定义用户关心的情况，OptionalCommands 用来定义关心的情况发生，中断给用户时所执行的命令。

举例来说，以下是一个典型的条件断点命令：

bp dbgee!wmain "j (poi(argc)>1) 'dd argc l1;du poi(poi(argv)+4)';'gc'"

这个命令对 dbgee 程序的 wmain 函数设置一个条件断点，只有当命令行参数的个数（argc）大于 1 时，才中断给用户，中断时执行 dd argc l1 和 du poi(poi(argv)+4)命令，显示出 argc 参数的值，和第一个命令行参数（即 argv[1]，argv[0]是程序文件）的字符串内容。

事实上，无论后面是否有条件命令，对于以上 bp 命令 WinDBG 都是在 dbgee!wmain 函数的入口处设置一个软件断点。当 CPU 执行到这个位置时，也总是触发断点事件，并报告给调试器。调试器收到断点事件后，会找到这个断点所附带的命令串，然后执行它的命令串，也就是双引号中的内容。于是 WinDBG 执行 j 命令，判断小括号中的条件，如果条件不满足，那么就执行分号后的 gc 命令，直接恢复调试目标执行。如果条件满足，那么则执行单引号中的命令，然后进入命令模式中断给用户。

如果使用.if 命令，那么上面的命令可以写成：

bp dbgee!wmain ".if (poi(argc)>1) {dd argc l1;du poi(poi(argv)+4)} .else {gc}"

下面解释一下上面命中的 poi 的含义。这是由于 WinDBG 缺省使用 MASM 语法的表达式评估器，在 MASM 语法中，argc 代表一个地址，要取它的值就需要使用 poi 操作符。Poi 的含义是

从指定地址取指针长度的数据（Pointer-sized data），类似的还有 by、wo、dwo、qwo 分别表示从指定地址取一个字节（Byte）、一个字（Word）、一个双字（Double-word）和一个四字（Quad-word）。关于 MASM 表达式的更多内容，请读者参考 WinDBG 帮助文件中关于 MASM Numbers and Operators 的介绍。

可以使用以下命令将表达式评估器切换为 C++评估器：

0:000> .expr /s c++

Current expression evaluator: C++ - C++ source expressions

此时以上命令可以表示为：

bp dbgee!wmain "j (argc>1) '? argc;?? argv[1]';'gc'"

但也许因为目前版本的 WinDBG 存在 BUG，以上命令在执行时会发生 Overflow 错误。

如果当前是 MASM 表达式评估器，可以使用@@前导符来嵌入 C++表达式，也就是可以把上面的断点命令写为：

bp dbgee!wmain "j @@(argc>1) 'dv argc; du @@(argv[1])';'gc'"

最后我们再给出一个针对函数参数设置条件断点的例子。比如，我们想了解 IO 管理器的 IoGetDeviceProperty 函数的工作细节，这个函数的原型如下：

```
NTSTATUS
  IoGetDeviceProperty(
    IN PDEVICE_OBJECT    DeviceObject,
    IN DEVICE_REGISTRY_PROPERTY    DeviceProperty,
    IN ULONG    BufferLength,
    OUT PVOID    PropertyBuffer,
    OUT PULONG    ResultLength
    );
```

其中第二个参数是个枚举型的常量，用来指定要查询的设备属性，比如 DevicePropertyBusNumber（14），用来查询设备的总线号。

如果直接对 IoGetDeviceProperty 函数设置断点，那么它会频繁命中，而我们实际上只关心第二个参数等于某个值，比如 14 的情况。

这时便可以这样设置断点：

```
0: kd> bp nt!IoGetDeviceProperty+0x5 ".if poi(@ebp+0xc) = 0xe  {} .else {.echo Entered IoGetDeviceProperty with ;dd (@ebp+0xc) l1; gc}"
```

首先，为什么将第一个参数（断点地址）设置为 nt!IoGetDeviceProperty+0x5，而不是 nt!IoGetDeviceProperty 呢？这是为了让建立栈帧的代码执行好后再中断，以便后面的条件表达式可以引用到栈帧中的参数。

```
nt!IoGetDeviceProperty:
8059ecf5 8bff            mov     edi,edi
8059ecf7 55              push    ebp
8059ecf8 8bec            mov     ebp,esp
8059ecfa 83ec64          sub     esp,64h
```

其中 poi(@ebp+0xc) 用来表示第二个参数的取值。

运行后，当这个函数被调用，但第二个参数不等于 14 时，WinDBG 会显示类似如下的信息：

```
breakpoint 4 redefined
0: kd> g
Entered IoGetDeviceProperty with
```

f7ca1a00 00000007
Entered IoGetDeviceProperty with
f7ca1a00 00000007

但第二个参数等于 14 时，WinDBG 会中断到命令模式，让我们做进一步的分析。

30.11.4地址表达方法

可以使用以下三种方法来指定断点命令中的地址参数。

- 使用模块名加函数符号的方式，比如 bp dbgee!wmain 代表对 dbgee 模块中的 wmain 函数设置断点。也可以在符号后增加一个地址偏移，比如 bp dbgee!wmain+3。
- 直接使用内存地址，比如 bp 00411390。
- 如果是使用完全的调试符号，调试符号中包含源代码行信息，那么可以使用如下形式：`[[Module!]Filename][:LineNumber]`，其中 Module 为模块名，Filename 为源程序文件名，LineNumber 为行号。整个表达式应该使用两个重音符号（`）包围起来，注意是重音符号，而不是单引号（’）。比如以下命令 bp `dbgee!dbgee.cpp:16`对于 dbgee.cpp 的 16 行设置断点，其中的 dbgee!可以省略。
- 对于 C++的类方法，也可以使用类名双冒号（::）或者双下划线（__）来连接类名和方法名，比如：bp MyClass__MyMethod，bp MyClass::MyMethod 或者 bp @@(MyClass::MyMethod)。

如果是使用前两种方法设置软件断点，那么应该确保断点地址指向的是指令的起始处，而不是一条指令的中部。如果指向一条指令的中部，那么在落实这个断点时，调试器就会把这条指令的中间字节替换为断点指令。这样，当 CPU 执行到这个位置时，CPU 会因为这里是一条多字节指令，把原来的指令和断点指令放在一起来解码，这会导致难以预知的结果。

30.11.5设置针对线程的断点

多于多线程程序，如果有多个线程都会调用某个函数，那么有时我们可能只希望在某个线程调用这个函数时才中断到调试器。

为了满足这一需要，WinDBG 的软件断点设置命令都支持在命令前增加线程限定符，即使用~加线程号，然后再是 bp 等命令和其它参数。例如，以下命令对 MSVCR80D!printf 设置一个断点，这个断点是线程相关的，只有当 0 号线程执行到这个函数时才会中断给用户：

~0 bp MSVCR80D!printf

以上方法适用于用户态调试的情况，对于内核态调试可以使用我们前面介绍的/p 和/t 选项来指定断点的进程上下文和线程上下文。

30.11.6管理断点

使用 bl 命令可以列出当前已经设置的所有断点，例如：

```
0:000> bl
 0 e [c:\...\dbgee.cpp @ 16]     0001 (0001)  0:**** dbgee!wmain+0x73
 1 d [c:\...\dbgee.cpp @ 8]     0001 (0001)  0:**** dbgee!wmain "kv"
     Call stack shallower than: 00000005
```

```
 2 e 7c8843ac r 1 0001 (0001)  0:**** kernel32!BasepCurrentTopLevelFilter
 3 e [f:\rtm\...\printf.c @ 49]    0002 (0002)  0:~000 MSVCR80D!printf
 4 e 7c91eb28      0001 (0001)  0:**** ntdll!DbgPrintEx+0x3 "kv"
```

以上命令结果中，第一列是断电的序号。第二列是断点的状态，e 代表启用（enable）、d 代表暂时禁止使用（disable），对于使用 bu 命令设置的断点，还可能 e 或者 d 后可能跟有字母 u，代表断点尚未求解落实（unresolved）。第三列是断点的地址，和设置断点时指定地址一样，有多种表示方法，可以为源文件加行号（断点 0、1、3），或者内存地址（断点 2、4）。对于数据断点（断点 2），地址后跟有访问方式（r）和访问长度（1），我们把它们连同地址看作一列。

第四列是断点触发剩余次数，即还要经过（pass）这个断点多少次才会中断给用户。第五列是断点的初始计数，也就是设置断点是 Passes 参数所指定的值，缺省为 1。

第六列是断点所关联的进程和线程，冒号前是进程号，冒号后是线程号。****代表这个断点是针对 所有线程的。

第七列是断点地址的符号表示。

如果断点有关联的命令，那么显示在第七列之后，例如上面断点 1 的 kv 命令。断点 1 下面的信息是因为这个断点使用了/c5 选项。

命令 bc、bd、be 分别用来删除、禁止和启用断点，它们的格式都是：

bc|bd|be 断点号

其中断点号可以使用*来通配所有断点，使用-来表示一个范围，或者使用逗号来指定多个断点号。例如以下命令都是有效的：

bd 0-2,4

禁止 0、1、2、和 4 号断点。

be *

启用所有断点。

可以使用 br 命令对改变某个断点的编号。例如当 3 号断点删除后，可以使用 br 4 3 将 4 号断点的编号改为 3 号。

30.12 观察栈

今天我们所使用的计算机系统都是基于栈架构的，栈是函数调用和程序运行的基础，栈中记录了软件运行的丰富信息。因此，观察和分析栈是软件调试的一个基本手段。在第 22 章我们详细讨论了栈的布局、栈帧的建立和变量分配等内容，本节我们将介绍如何利用 WinDBG 的命令来观察和分析栈。

30.12.1 显示栈回溯

因为函数调用指令（CALL）会将函数的返回地址记录在栈上，因此通过从栈顶向下遍历每个栈栈帧，然后找到其中的函数返回地址，便可以追溯出函数调用过程，这个过程被称为栈回溯（Stack Backtrace）。

WinDBG 的 k 系列命令就是用来帮助我们进行栈回溯的。之所以叫 k 系列命令是因为 WinDBG 提供了以 k 开头的多个命令来做栈回溯。这些命令的基本功能是一样的，但是现实的信息格式各有侧重点。

我们先来看基本的 k 命令。以下是当 dbgee 程序的 main 函数的断点命中时，执行 k 命令后的结果。

清单 30-9 使用 k 命令进行栈回溯

```
0:000> k
ChildEBP RetAddr
0012ff68 00411ad6 dbgee!wmain [c:\dig\dbg\author\code\chap28\dbgee\dbgee.cpp @ 8]
0012ffb8 0041191d dbgee!__tmainCRTStartup+0x1a6 [f:\...\crtexe.c @ 583]
0012ffc0 7c816ff7 dbgee!wmainCRTStartup+0xd [f:\...\crtexe.c @ 403]
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

其中的每一行代表栈上的一个栈帧，也就是一个函数。最上面一行表示的是当前正在执行（中断到调试器时）的函数。每个函数下面的一行是这个函数的父函数。因此，整个函数调用关系是下面的函数调用上面的。例如第 1 行描述的是 wmain 函数和它的栈帧，第 2 行是调用 wmain 函数的__tmainCRTStartup 函数，第 3 行是 wmainCRTStartup 函数，最下面是进程启动函数 BaseProcessStart。

横向来看，第一列是栈帧的基地址，因为 x86 架构中通常使用 EBP 寄存器来记录栈帧的基地址，所以这一行的标题叫 ChildEBP，意思是子栈帧（子函数）的基地址寄存器（EBP）值。第二列是函数的返回地址，这个地址是父函数中的指令地址，通常就是调用本行函数的那条 CALL 指令的下一条指令的地址。第三列是函数名以及执行位置，其中执行位置表示的是程序指针指向的位置（对于正在执行的函数）或者返回到这个函数时将执行的指令地址（对于父函数）。例如第一行的 dbgee!wmain 表示当前的程序指针指向的是这个函数的第一条指令。第 2 行的 dbgee!__tmainCRTStartup+0x1a6 表示的是 wmain 返回后将执行的指令地址，这个值其实是通过寻找距离第 1 行的返回地址值最近的符号而得到的。

```
0:000> ln 00411ad6
f:\rtm\vctools\crt_bld\self_x86\crt\src\crtexe.c(583)+0x19
(00411800)   dbgee!__tmainCRTStartup+0x1a6   |  (00411ae0)   dbgee!NtCurrentTeb
```

第 3 列之后是源文件信息（如果有）。如果不想看到这个信息可以在 k 命令后加上 L（需要大写）选项。

K 命令显示了函数名信息，但是没有显示每个函数的参数。命令 kb 可以显示放在栈上的前三个参数。例如以下是在与清单 30-9 同样的条件下执行 kb L 命令的结果（使用 L 开关是为了屏蔽掉与上文相同的源文件信息，以节约篇幅）。

清单 30-9 使用 kb 命令进行栈回溯

```
0:000> kb L
ChildEBP RetAddr  Args to Child
0012ff68 00411ad6 00000002 003a2e90 003a5c20 dbgee!wmain
0012ffb8 0041191d 0012fff0 7c816ff7 0175f6f2 dbgee!__tmainCRTStartup+0x1a6
0012ffc0 7c816ff7 0175f6f2 0175f77a 7ffd8000 dbgee!wmainCRTStartup+0xd
0012fff0 00000000 0041107d 00000000 78746341 kernel32!BaseProcessStart+0x23
```

易见，前两列以及最后一列的内容与 k 命令的结果是一样的。所不同的是中间三列，这三列被称为是子函数的参数（Args to Child）。不管函数的实际参数个数是多少，这里总是显示三个，而且其位置是固定的，第一个参数即 EBP+8，第二个就是 EBP+C，第三个是 EBP+0x10，如果要观察第四个参数，那么可以使用 dd EBP+0x14，依此类推。尽管通常说这三列是函数的前三个参数，事实上这是不准确的，严格来说，这只是放在栈上的前三个参数。对于使用快速调用协议（FASTCALL）的函数来说，某些参数是用寄存器来传递的，因此栈上的前三个参数很可能并不是真正的前三个参数顺序。换句话来说，调试器只是把栈帧上用来传递参数的三个内存位置的值显示出来，至于它们到底对应的是哪个参数，那么应该参考函数的原型。Kp 命令会根据符号文件中的函数原型信息来帮助我们自动做这件事（清单 30-11）。

清单 30-11 使用 kp 命令进行栈回溯

```
0:000> kp L
ChildEBP RetAddr
0012ff68 00411ad6 dbgee!wmain(int argc = 2, wchar_t ** argv = 0x003a2e90)
0012ffb8 0041191d dbgee!__tmainCRTStartup(void)+0x1a6
0012ffc0 7c816ff7 dbgee!wmainCRTStartup(void)+0xd
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

kp 命令把参数和参数值都以函数原型的格式显示出来，显然更易于理解。但是仅当有完全的调试符号（私有符号）时才能做到这一点。在没有私有符号的情况下，kp 命令不会显示任何参数，这也是使用 kb 命令的客观原因。

```
0:000> kp
ChildEBP RetAddr
0012fb1c 7c93edc0 ntdll!DbgBreakPoint
0012fc94 7c921639 ntdll!LdrpInitializeProcess+0xffa
0012fd1c 7c90eac7 ntdll!_LdrpInitialize+0x183
00000000 00000000 ntdll!KiUserApcDispatcher+0x7
```

kv 命令可以在 kb 命令的基础上增加显示 FPO（栈指针省略）信息和调用协议（清单 30-12）。

```
0:000> kv L
ChildEBP RetAddr  Args to Child
0012ff68 00411ad6 … dbgee!wmain+0x3 (FPO: [Non-Fpo]) (CONV: cdecl)
0012ffb8 0041191d … dbgee!__tmainCRTStartup+0x1a6 (FPO: [Non-Fpo]) (CONV: cdecl)
0012ffc0 7c816ff7 … dbgee!wmainCRTStartup+0xd (FPO: [Non-Fpo]) (CONV: cdecl)
0012fff0 00000000 … kernel32!BaseProcessStart+0x23 (FPO: [Non-Fpo])
```

可以看到前六列与 kb 命令都是一样的（为了排版，我们省略了参数）。第 7 列和第 8 列分别是 FPO 信息和调用协议，前者以 FPO:开始，后者以 CONV:开始。

因为上面的各个函数都没有使用帧指针省略，所以显示的都是(FPO: [Non-Fpo])，意思是帧中没有 FPO 数据。

除了以上命令，还有 kn 命令，它会在每行前显示栈帧的序号。另外可以在所有 k 命令中指定 f 选项，有了这个选项后，那么会显示每两个相邻栈帧的内存距离，即栈帧基地址的差值。

清单 30-11 在 k 命令中使用 f 和 L 选项

```
0:000> kn f L
 #   Memory  ChildEBP RetAddr
00           0012ff68 00411ad6 dbgee!wmain+0x3
01        50 0012ffb8 0041191d dbgee!__tmainCRTStartup+0x1a6
02         8 0012ffc0 7c816ff7 dbgee!wmainCRTStartup+0xd
03        30 0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

其中，第一列是栈帧序号，当前正在执行函数的栈帧为 0 号，依次类推。第二列是相邻栈帧的基地址差值，某一行的这个数值越大，说明这个函数使用的栈空间越多。

30.12.2 观察栈变量

大多数局部变量是分配在栈上的（详见 22 章）。观察某个函数的栈帧所对应的内存，就可以看到这个函数的位于栈上的局部变量的当前值。但是直接观察原始内存，其可读性较差。WinDBG 的 dv 命令可以帮助我们以更友好的方式显示这些变量。

仍然以前面的 dbgee 小程序为例，当位于 main 函数入口的断点命中时，执行 dv 命令，其结果如下：

```
0:000> dv /i/t/V
prv param  0012ff70 @ebp+0x08 int argc = 1
prv param  0012ff74 @ebp+0x0c wchar_t ** argv = 0x003a2e90
prv local  0012fd4c @ebp-0x21c wchar_t [260] szBuffer = wchar_t [260] "□뫫???"
prv local  0012ff5c @ebp-0x0c int nRet = 0
```

其中第一列 prv 是 private 的缩写，表示这个信息是利用私有符号产生的。第二列是变量的类型，param 表示函数参数，local 表示局部变量。第三列是变量的内存起始地址，这个地址是属于当前函数栈帧的。第 4 列是使用栈帧基地址（EBP）所表示的变量起始地址，因为栈是向低地址方向生长的，参数位于 EBP 的正偏移方向，局部变量位于 EBP 的负偏移方向。对于上面的数据，当前函数的栈帧基地址是 0012ff68（清单 30-11），即 EBP=0012ff68，第一个参数的位置是 EBP+8，即 0012ff70，第二个参数是 EBP+0xC，即 0012ff74。

第五列是变量类型，第六列是变量名称，而后是等号和变量取值。值得说明的是，因为现在还在 main 函数的入口处，建立 main 函数栈帧的代码尚未运行，所以 szBuffer 变量的值还是随机的，当建立栈帧的代码运行后，它会被断点指令（0xCC）填充。Dv 命令能够在这个时候显示出以上信息，是因为其内部利用 ESP 寄存器的值调整了 EBP 的值。

如果要观察父函数的局部变量，那么可以使用.frame 命令加上父函数的帧号将局部上下文切换到那个栈帧，然后再使用 dv 命令。

例如，从清单 30-11 可以看到__tmainCRTStartup 函数的帧号为 1，所以可以发出如下命令：

```
0:000> .frame 1
01 0012ffb8 0041191d dbgee!__tmainCRTStartup+0x1a6 [f:\rtm\...\crt\src\crtexe.c @ 583]
```

此时再用 dv 命令就可以看到__tmainCRTStartup 函数的栈变量了：

```
0:000> dv /i /t /V
prv local  0012ff94 @ebp-0x24 void * lock_free = 0x00000000
prv local  0012ff98 @ebp-0x20 void * fiberid = 0x00130000
prv local  0012ff9c @ebp-0x1c int nested = 0
```

使用 dv 命令显示局部变量需要有私有符号信息，对于没有私有符号的模块，dv 命令是无法工作的。比如栈帧 3 是 kernel32 的 BaseProcessStart 函数，我们先把局部上下文切换到这个函数：

```
0:000> .frame 03
03 0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

然后再执行 dv 命令：

```
0:000> dv /i /t /V
Unable to enumerate locals, HRESULT 0x80004005
Private symbols (symbols.pri) are required for locals.
```

Type ".hh dbgerr005" for details.

可见 dv 命令失败了，原因是缺少 kernel32 模块的私有调试符号。对于这种情况只能用手工方法来观察局部变量。这通常有两种方法。第一种是直接使用内存观察窗口或者内存显示命令（下一节介绍）浏览当前栈帧的内存区域。根据 EBP 的值和栈帧布局知识来寻找局部变量。对于字符串类型的变量，有时候可以根据变量内容找到它的起始位置。

第二种方法是根据汇编指令中对局部变量的引用，来得到局部变量的地址，然后再观察这个地址的内容。对于没有使用 FPO 的函数，局部变量大都是通过 EBP-XXX 的方式来引用的。

例如以下就是 main 函数中使用 nRet 局部变量的汇编语句：

```
004113c8 c745f400000000    mov         dword ptr [ebp-0Ch],0
```

对于 VC7 或者更高版本的 VC 编译器，局部变量至少是从偏移-C 开始的，因为 EBP-4 到 EBP 存放的是安全 Cookie 值，EBP-8 到 EBP-4 存放的是保护安全 Cookie 的屏障字段（即 0xCCCCCCCC）。另外，在 32 位系统中，栈空间是以 4 字节为单位分配的，所以 0xC 是可能的最小局部变量偏移地址，这个局部变量可以是个整数或者比整数还短的类型。

例如以下 main 函数栈帧附近内存的原始内容：

```
0:000> dd ebp-10
0012ff58    cccccccc 00000000 cccccccc 04b69550
0012ff68    0012ffb8 00411ad6 00000001 003a2e90
```

其中，EBP（0012ff68）处的数据（0012ffb8）是父函数的 EBP 值，EBP+4 是 main 函数的返回地址（00411ad6），EBP-4 处的值（04b69550）是安全 Cookie，EBP-8 是 Cookie 屏障，EBP-0xC 就是局部变量 nRet。

再例如，以下是访问局部变量的另一段典型代码：

```
0041146e 8d85e4fdffff    lea     eax,[ebp-21Ch]
00411474 50              push    eax
00411475 ff15cc824100    call    dword ptr [dbgee!_imp__wprintf (004182cc)]
```

第一句是把变量的地址（指针）放入到 EAX 寄存器，第二句是作为参数压入到栈中，第三句是调用字符串打印或者处理函数。由此推断 ebp-21Ch 一定是个字符串类型的局部变量，因此可以使用 du 命令观察它的值：

```
0:000> du ebp-21Ch
0012fd4c    "Arg [0]: C:\dig\dbg\author\code\"
0012fd8c    "chap28\dbgee\Debug\dbgee.exe"
```

这正是 main 函数中 szBuffer 变量的值。

扩展命令!for_each_local 用于枚举当前栈帧的所有局部变量，并可以对每个变量执行一系列命令，在这些命令中可以使用别名@#Local 来指代每个变量。例如可以使用以下命令显示每个局部变量的类型和值：

```
0:000> !for_each_local dt @#Local
Local var @ 0x12ff70 Type int
2
Local var @ 0x12ff74 Type wchar_t**
……
```

类似的扩展命令!for_each_frame 用来遍历所有栈帧，比如以下命令可以显示当前现成的每个栈帧的局部变量：

```
0:000> !for_each_frame dv
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
00 0012ff68 00411ad6 dbgee!wmain+0x2f [c:\...\dbgee.cpp @ 15]
                argc = 2
                argv = 0x003a2e90
……
```

其效果相对于先使用.frame 命令切换到每个栈帧，然后执行 dv 命令。

30.12.3 手工回溯栈

有时因为缓冲区溢出或者其它错误，可能导致栈帧信息被破坏，以致于以上介绍的栈回溯命

令无法正常工作。这时可能不得不使用手工方法来分析和回溯栈。

以下是栈回溯的基本原理和步骤：

1) 根据程序的当前执行位置，得到程序指针寄存器的值，然后查找符号得到当前所在的函数。将得到的函数名记录下来，作为栈回溯报告的第一行第 3 列。
2) 寻找当前函数的栈帧基准地址。如果 EBP 或者 ESP 寄存器的内容没有被破坏，那么就可以使用它们的值。EBP 是栈帧的基准地址，ESP 是栈的当前栈顶地址。如果 EBP 或者 ESP 寄存器的内容不可信，那么可以根据当前线程的线程信息块中的栈信息找到栈的基地址和界限，这可以通过 WinDBG 的!teb 命令来获得。得到的 EBP 值便是栈回溯报告中的 ChildEBP 值，将其记录在当前行的第一列。
3) 寻找当前函数的返回地址，这个信息通常被保存在 EBP+4 位置。如果 EBP 的值不确定，那么可以从栈顶依次判断每个 DWORD 值（32 位系统，64 位系统应该看每个 QWORD），看其是否“像”函数返回地址，有效的用户态函数返回地址应该是小于 0x80000000，至少大于 0x1000，EXE 模块的缺省加载位置是 0x400000，DLL 的缺省加载位置通常高一些，所以典型的函数返回地址在 0x400000 到 0x80000000 间。可以使用 ln 命令来搜索一个数值附近的符号，如果它距离某个符号很近，那么就有可能。WinDBG 的 dds 命令（64 位系统为 dqs）可以帮助我们自动的处理指定内存范围内的所有 DWORD 值，逐一搜索每个值对应的符号。这样的得到的返回地址就是栈回溯报告中的第 2 列。这个地址的函数名就是下一行的第 3 列。
4) 寻找父函数的栈帧基地址。如果 EBP 寄存器的值可信，那么 EBP 值所指向的就是父函数的栈帧基地址（不考虑 FPO 情况），也就是 EBP 值所对应地址处的值就是父函数的 EBP 值。如果不能使用 EBP 值，那么可以使用第 3 不找到的函数返回地址位置推测当前函数的 EBP。因为 EBP+4 是函数返回地址的位置，所以函数返回地址位置-4 就是 EBP 的位置。这个 EBP 值就是父函数的 ChildEBP 值，将其记录在父函数的第 1 列。
5) 回到第 3 步，继续寻找父函数的父函数。直到 ChildEBP 值等于 0，表示已经到达当前线程的最后一帧。

下面通过一个实例来说明。我们就是 dbgee 程序为例，当前处于 main 函数中，我们的目标是手工生成类似 k 命令的栈回溯报告。

首先使用 ln 命令寻找当前函数的函数名。

```
0:000> ln eip
c:\dig\dbg\author\code\chap28\dbgee\dbgee.cpp(25)
(004113a0)   dbgee!wmain+0xcc   |  (00411540)   dbgee!wprintf
```

我们先考虑所有寄存器都没有被损坏的情况，也就是此时的 EBP 值就是当前函数的 ChildEBP。EBP+4 的值就是函数返回地址：

```
0:000> r ebp
ebp=0012ff68
0:000> dd ebp+4 l1
0012ff6c  00411ad6
```

把以上结果放在一起，加上标题行便得到回溯报告的第一部分：

```
ChildEBP RetAddr
0012ff68 00411ad6 dbgee!wmain+0xcc
```

接下来使用 ln 命令寻找父函数的函数名：

```
0:000> ln 00411ad6
f:\rtm\vctools\crt_bld\self_x86\crt\src\crtexe.c(583)+0x19
(00411930)   dbgee!__tmainCRTStartup+0x1a6   |  (00411c10)   dbgee!NtCurrentTeb
```

根据当前的 EBP 值得到父函数的 EBP 值：

0:000> dd 0012ff68 l1

0012ff68 0012ffb8

即 0012ffb8，父函数 EBP 的偏移+4 处是父函数的返回地址。

0:000> dd 0012ffb8+4 l1

0012ffbc 0041191d

将以上信息放在一起，便得到第二行。

0012ffb8 0041191d dbgee!__tmainCRTStartup+0x1a6

使用同样方法可以得到第三行和第四行：

0012ffc0 7c816ff7 dbgee!wmainCRTStartup+0xd

0012fff0 00000000 kernel32!BaseProcessStart+0x23

此时再显示 EBP 指针的值

0:000> dd 0012fff0 l1

0012fff0 00000000

0 代表已经到达最后一帧，至此栈回溯结束，将以上三部分信息放在一起便得到一份完整的栈回溯报告，与 k 命令的输出是完全的一致的。

下面再考虑异常的情况，假设 EBP 和 ESP 寄存器的值已经不可信。此时可以使用!teb 命令得到栈的内存位置：

0:000> !teb

TEB at 7ffdf000

    ExceptionList: 0012ffa8

    StackBase: 00130000

    StackLimit: 00126000

    ...

以上信息信息，栈的起始地址是 0x00130000，边界是 0x00126000，即目前的栈空间是 0x00126000 到 0x00130000。

栈是从基地址向低地址方向生长的。因此我们先使用 dds 命令来显示栈底附近区域的内容(清单 30-12)。

清单 30-12 使用 dds 命令显示和分析栈

```
0:000> dds 0x00130000-0n168 0x00130000
0012ff58  cccccccc
0012ff5c  00000000
0012ff60  cccccccc
0012ff64  6370c10b
0012ff68  0012ffb8
0012ff6c  00411ad6 dbgee!__tmainCRTStartup+0x1a6 [f:\rtm\...\crtexe.c @ 583]
0012ff70  00000002
【省略 13 行】
0012ffa8  0012ffe0
0012ffac  00411082 dbgee!ILT+125(__except_handler4)
0012ffb0  63235433
0012ffb4  00000000
0012ffb8  0012ffc0
0012ffbc  0041191d dbgee!wmainCRTStartup+0xd [f:\rtm\...\crtexe.c @ 403]
```

```
16  0012ffc0  0012fff0
17  0012ffc4  7c816ff7 kernel32!BaseProcessStart+0x23
18  0012ffc8  015ff6f2
19  0012ffcc  015ff77e
20  0012ffd0  7ffdd000
21  0012ffd4  e5434c00
22  0012ffd8  0012ffc8
23  0012ffdc  88425020
24  0012ffe0  ffffffff
25  0012ffe4  7c839a30 kernel32!_except_handler3
26  0012ffe8  7c817000 kernel32!`string'+0x98
27  0012ffec  00000000
28  0012fff0  00000000
29  0012fff4  00000000
30  0012fff8  0041107d dbgee!ILT+120(_wmainCRTStartup)
31  0012fffc  00000000
32  00130000  78746341
```

以上显示的第一列是栈内存的地址，第二列是该地址的取值，第三列是取值对应的符号。双斜线后是我们的注释。

首先我们选取包含符号的各个行，也就是 7、11、15、17、25、26、和 30 行，然后排除明显不是函数的部分 26 行（用于异常处理的范围表），11 和 25 行是 SHE 处理器的异常处理函数，30 行登记在 PE 文件中的入口函数，它是进程启动函数的参数。把剩下来的各个行连同它的上一行挑选出来放在一起：

```
0012ff68  0012ffb8
0012ff6c  00411ad6 dbgee!__tmainCRTStartup+0x1a6 [f:\rtm\...\crtexe.c @ 583]
0012ffb8  0012ffc0
0012ffbc  0041191d dbgee!wmainCRTStartup+0xd [f:\rtm\...\crtexe.c @ 403]
0012ffc0  0012fff0
0012ffc4  7c816ff7 kernel32!BaseProcessStart+0x23
```

有符号那一行的第二列是这个符号的地址，也就是它的子函数的返回地址。根据函数返回地址-4 是 EBP，那么有符号那一行的上一行就是子函数的 EBP 值（第一列）和父函数的 EBP 值（第二列）。上面加粗显示的都是把 EBP 值，容易可以看到它们的链接关系：子函数的 EBP 指向父函数的 EBP。

把上面的显示略作调整：把偶数行的第二列（函数返回地址）移到上一行，第三列移到下一行，然后删除偶数行：

```
0012ff68  00411ad6
0012ffb8  0041191d dbgee!__tmainCRTStartup+0x1a6 [f:\rtm\...\crtexe.c @ 583]
0012ffc0  7c816ff7 dbgee!wmainCRTStartup+0xd [f:\rtm\...\crtexe.c @ 403]
0012fff0           kernel32!BaseProcessStart+0x23
```

然后再根据当前程序指针位置找到第一行的函数名，并将最后一行的第二列填写为 0 便可以了。

30.13 分析内存

内存是软件工作的舞台，程序的代码必须先加载到内存中才可以被 CPU 所执行，程序的数据（变量）也主要是分配在内存中的（极少数变量分配在寄存器中）。除了外界因素外，程序的行为就是由它的代码和数据所决定的，程序的内存状态，决定了它的外在行为。很多时候，软件调试的目标就是为了搞清楚某个行为的内在根源，即所谓的根本原因（Root Cause）。观察和分析内存是寻找根本原因的最有效方式之一。

内存空间是通过地址来标识和引用的。内存地址有多种，常用的有物理内存地址和虚拟内存地址。本节中的内存地址除非特别说明外都是指虚拟内存地址。

30.13.1 显示内存区域

WinDBG 的 d 系列命令用来显示指定内存区域的数据内容。这些命令的格式为：

d{a|b|c|d|D|f|p|q|u|w|W} [Options] [Range]

dy{b|d} [Options] [Range]

d [Options] [Range]

其中大括号中的字母（区分大小写）用来指定数据的显示方式，a 表示 ASCII 码，b 表示字节和 ASCII 码，c 表示 DWORD 和 ASCII 码，d 表示 DWORD，D 表示双精度浮点数，f 表示单精度浮点数，p 表示按指针宽度显示，q 表示四字（8 字节），u 表示 UNICODE 字符，w 表示字，W 表示字和 ASCII 码，yb 表示二进制和字节，yd 表示二进制和双字。

Range 参数用来指定要显示的内存范围。可以有以下几种表示方法。

第一种方法是起始地址加空格加终止地址，比如 dd 0012fd9c 0012fda8 命令以双字格式显示从 0012fd9c 开始到 0012fda8 结束的 16 字节内存数据：

```
0:000> dd 0012fd9c 0012fda8
0012fd9c    cccccccc cccccccc cccccccc cccccccc
```

第二种方法是起始地址加空格加 L（或者 l）和对象个数，比如上面的命令可以等价的写为：dd 0012fd9c L4。

第三种方式是终止地址加空格加 L（或者 l）加负号和对象个数。使用这种方式可以把上面的命令写为：dd 0012fdac L-4。

注意理解上面的对象个数（L 后的数字），它是“数据单元”的个数，而不是字节数。对于 dd 它的单位是 DWORD，对于 db，它的含义是 4 个字节，例如：

```
0:000> db 0012fd9c l4
0012fd9c    cc cc cc cc                                                    ....
```

如果省略数据显示格式而直接执行 d 命令，那么它将采用最近使用过的数据显示格式。

30.13.2 显示字符串

可以以 0 结尾的简单字符串，可以使用 da 或者 du 命令来显它的内容，前者用于使用单字节字符集的字符串，后者用于采用 UNICODE 字符集的字符串。当遇到字符串末尾的 0 时，会自动停止显示。例如：

```
0:000> du 003a2e9c
```

003a2e9c "C:\dig\dbg\author\code\chap28\db"
003a2edc "gee\Debug\dbgee.exe"

如果使用 da 命令显示 UNICODE 字符集的字符串，如果字符串的内容是英文字母，那么通常只能显示第一个字符：

```
0:000> da 003a2e9c
003a2e9c  "C"
```

有些字符串是使用数据结构来表示的，常用的结构有 UNICODE_STRING 结构和 STRING 结构，它们的定义分别为：

```
0:000> dt _UNICODE_STRING
ntdll!_UNICODE_STRING
   +0x000 Length           : Uint2B
   +0x002 MaximumLength    : Uint2B
   +0x004 Buffer           : Ptr32 Uint2B
0:000> dt _STRING
ntdll!_STRING
   +0x000 Length           : Uint2B
   +0x002 MaximumLength    : Uint2B
   +0x004 Buffer           : Ptr32 Char
```

对于着两种结构的字符串，可以分别使用 dS（S 大写）（UNICODE_STRING 结构）和 ds 命令（STRING 结构）来显示。对于后者，也可以使用!str 扩张命令来显示。

30.13.3 显示数据类型

WinDBG 的 dt 命令用来显示数据类型以及按照类型来显示数据。Dt 的含义是 Dump symbolic Type information。Dt 是个比较复杂的命令，下面我们按照用法分别来介绍。

首先，可以使用 dt 来显示一个数据类型（数据结构）。比如上文我们用 dt 命令显示了 _UNICODE_STRING 结构的定义。这种用法的典型格式是：

dt [模块名!]类型名

其中模块名部分可以省略，如果省略，那么调试器会自动搜索所有模块。类型名即程序中定义数据结构或者通过 typedef 定义的类姓名。类型名中可以包含通配符，比如以下命令会列出 NTDLL 模块中的所有类型

```
0:000> dt ntdll!*
            ntdll!LIST_ENTRY64
......
```

如果类型名是确定的类型，那么 dt 便会显示这个类型的定义，如果类型中还包含子类型，那么可以用-b 开关来递归式显示所有子类型，也可以使用-r 开关来指定显示深度。-r0 表示不显示子类型，-r1 表示显示 1 级子类型，依此类推，例如：

```
0:000> dt -r1 _TEB
ntdll!_TEB
   +0x000 NtTib             : _NT_TIB
      +0x000 ExceptionList     : Ptr32 _EXCEPTION_REGISTRATION_RECORD
      +0x004 StackBase         : Ptr32 Void
      +0x008 StackLimit        : Ptr32 Void
```

.....

如果不想显示整个结构，而只显示某些字段，那么可以在类型名后使用-ny 开关附加字段搜索选项，比如以下命令只显示 TEB 结构的 LastError 开始的字段：

```
0:000> dt _TEB -ny LastError
ntdll!_TEB
   +0x034 LastErrorValue : Uint4B
```

Dt 命令的第二种用法是在上一种方法的基础上增加内存地址，让 dt 按照类型显示指定地址的变量。例如，以下命令使用_PEB 结构来显示内存地址 7ffdd000 出的数据：

```
0:000> dt _PEB 7ffdd000
ntdll!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged     : 0x1 ''
.....
```

这时仍可以使用前面介绍的-r 和-y 开关。

Dt 命令的第三种用法是显示类型的实例，包括全局变量、静态变量和函数。比如以下命令显示 dbgee 程序的 g_szGlobal 全局变量：

```
0:000> dt dbgee!g_szGlobal
[14]  "A global var."
```

因为函数是一种特殊的类型实例，所以也可以使用它来枚举函数符号，这样使用 dt 命令实际上与 x 命令的功能很类似，比如：

```
0:000> dt dbgee!*wmain*
004113a0  dbgee!wmain
00411910  dbgee!wmainCRTStartup
```

如果指定确定的某个函数，那么 dt 会显示它的参数和取值：

```
0:000> dt dbgee!wmain
wmain  int (
    int argc = 2,
    wchar_t** argv = 003a2e90 )
```

dt 命令的另一种用法是使用-l 开关指定一个指向下一链表元素的字段，让 dt 命令按指定的类型依次显示每个链表元素，我们稍后再详细讨论这种用法。

30.13.4搜索内存

一个 32 位 Windows 程序的进程空间是 4GB，用户空间有 2GB 或者 3GB，一个典型的 Windows 程序实际使用的内存空间通常在几百 KB 到几十 MB 之间。即使是几百 KB，手工在这么大的内存范围内寻找某个内容也是很困难的。这时可以借助 WinDBG 的 s 命令来帮忙。

S 命令有三种使用方法，我们按照又简单到复杂的顺序依次介绍。

第一种用法是在指定的内存范围内搜索任何 ASCII 字符或者 UNICODE 字符串，其格式如下：

s -[[Flags]]sa|su Range

其中 Range 用来指定内存范围，其写法与前面在 d 命令中介绍的一样，sa 用搜索 ASCII 字符串，su 用来搜索 UNICODE 字符串。[Flags]可以用来指定搜索选项，比如可以用 l 加一个整数来指定字符串的最小长度，使用 s 将搜索结果保存起来，然后使用 r 再在保存的结果中搜索。

例如，以下命令搜索 nt!PsInitialSystemProcess 变量所指向地址开始的 512 个字节范围内任何长度不小于 5 的 ASCII 字符串：

```
lkd> s-[l5]sa poi(nt!PsInitialSystemProcess) l200
8a672764  "System"
```

其结果表示在内存地址 8a672764 处找到了"System"字符串。事实上这正是初始系统进程的名字，也就是_EPROCESS 结构的 ImageFileName 字段。

```
lkd> dt nt!_EPROCESS -y Ima
   +0x174 ImageFileName : [16] UChar
```

第二种用法是在指定内存地址范围内搜索与指定对象相同类型的对象，这里的对象是指包含虚拟函数表的使用面向对象语言（如 C++）编写的类（Class）对象。其格式为：

```
s -[[Flags]]v Range Object
```

例如，在我们编写的 MfcHello 程序中，CMfcHelloDlg 类定义了五个 CButton 类的实例 m_Button1～m_Button5。通过观察我们知道 m_Button1 对象的地址是 0x12fe4c，CMfcHelloDlg 类实例的地址是 0x12fc30，因此我们可以输入以下命令来搜索与 m_Button1 同类型的其它对象：

```
0:000> s-v 0x12fc30 l1000 0x12fe4c
```

但是在 6.7.5.1 版本的 WinDBG 中 s 命令的这种用法似乎存在错误（bug），它总是返回以下错误信息：

```
Object '0x12fe4c' has no vtables
```

S 命令的第三种用法是在指定范围内搜索某一内容模式，其语法格式为：

```
s [-[[Flags]]Type] Range Pattern
```

其中类型表示搜索内容的数据类型（宽度），它决定了匹配搜索内容的方式，可以为字母 b（字节）、w（字）、d（双字）、q（四字）、a（ASCII 字符串）或者 u（Unicode 字符串）之一，如果不指定类型，那么默认类型为 b，即按字节搜索指定的内容。Pattern 参数用来指定要搜索的内容，可以用空格分隔依次要搜索的数值，比如：

```
0:000> s-w 0x400000 l2a000 41 64 76 44 62 67
0041b954  0041 0064 0076 0044 0062 0067 0000 0000  A.d.v.D.b.g.....
```

因为要搜索的内容可以表示为 ASCII 码，因此以上命令也可以等价表示为如下形式：

```
0:000> s-w 0x400000 l2a000 'A' 'd' 'v' 'D' 'b' 'g'
0041b954  0041 0064 0076 0044 0062 0067 0000 0000  A.d.v.D.b.g.....
```

或者按字符串搜索，需要用双引号来包围要搜索的内容，比如：

```
0:000> s-u 0x400000 l2a000 "AdvDbg"
0041b954  0041 0064 0076 0044 0062 0067 0000 0000  A.d.v.D.b.g.....
```

以下是搜索双字的一个例子：

```
0:000> s-d 12fe4c l20 782e35fc
0012fe4c  782e35fc 00000001 00000000 00000000  .5.x............
```

30.13.5修改内存

命令 e 用来修改指定内置地址或者区域的内容，我们仍然按用法来介绍。

第一种是按字符串编辑，其命令格式为：

```
e{a|u|za|zu} Address "String"
```

其中 Address 是要修改内存的起始地址。za 代表以 0 结尾的 ASCII 字符串，zu 代表以 0 结尾的 Unicode 字符串，a 和 u 分别代表不是以 0 结尾的 ASCII 和 Unicode 字符串。

仍然以 MfcHello 小程序为例，在 CMfcHelloDlg 类中我们定义了一个 TCHAR m_szBuffer[MAX_PATH]成员，在构造函数中将其初始化为"AdvDbg"，使用 CMfcHelloDlg 实例的地址作为开始地址搜索这个内容，我们可以找到它的地址：

0:000> s-u 0x0012fa20 l200 "AdvDbg"

0012fc94　0041 0064 0076 0044 0062 0067 0000 cccc　A.d.v.D.b.g.....

从上面的命令结果知道从内存地址 0012fc94 开始存放着 Unicode 字符串"AdvDbg"，它是以 0 结尾的，0 之后是初始化时填充的 CC。现在可以输入如下命令修改这个地址的内容：

0:000> ezu 12fc94 "DbgAdv"

使用内存观察命令观察，可以看到修改成功了：

0:000> dw 12fc94

0012fc94　0044 0062 0067 0041 0064 0076 0000 cccc

0012fca4　cccc cccc cccc cccc cccc cccc cccc cccc

……

将上面命令中的 ezu 换成 eu 得到的结果也是一样的，但是如果新的串比原来的串长，这时再使用 eu 命令就可能导致原来的字符串失去结尾的 0，而无法正常显示：

0:000> eu 12fc94 "Advanced Debugging"

0:000> du 12fc94

0012fc94　"Advanced Debugging 쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘"

0012fcd4　"쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘쵘"

……

这时如果恢复程序运行，那么可能导致程序错误。只要使用 ezu 编辑一下，就恢复正常了：

0:000> ezu 12fc94 "Advanced Debugging"

0:000> du 12fc94

0012fc94　"Advanced Debugging"

也就是说，ezu 命令或保证在编辑好的串末尾加 0，而 eu 不会。类似的，ea 和 eza 的差异也是这样。

第二种用法是以数值方式编辑，其格式为：

e{b|d|D|f|p|q|w} Address [Values]

其中，大括号中的字母用来表示要修改数据的类型，也决定了修改内存的方式。Address 用来指定要改内存的起始地址。Values 用来指定新的值，其表示方法与前面搜索内存中的方法相同。Values 参数的多少决定了要修改内存的长度，例如以下命令将 0x12fc94 开始的 5 个 WORD 都改为 0x41（字符 A）：

0:000> ew 12fc94 41 41 41 41 41

显示修改后的内存：

0:000> du 12fc94

0012fc94　"AAAAAced Debugging"

如果在命令中没有指定 Values 参数，那么 WinDBG 会以交互式的方式来让用户输入，命令提示符会改变为“Input>”（图 30-5）。

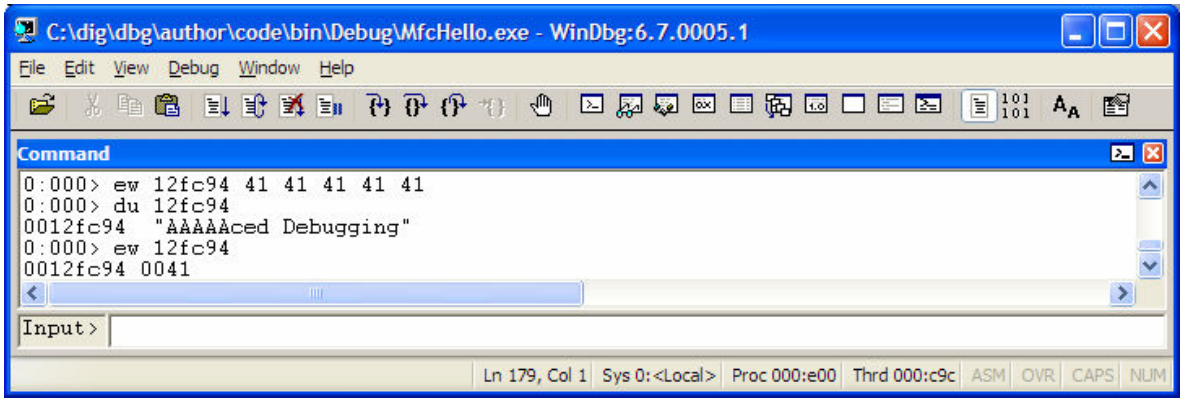


图 30-5 交互式的修改内存

根据命令中指定的编辑类型，WinDBG 会先显示出要编辑的内存地址（12fc94），和当前的取值（0041），然后等待用户输入。此时可以输入新的值，然后按回车提交输入值。如果想保留当前值，那么可以按空格然后再按回车。如果想停止输入，那么直接按回车键。

30.13.6 使用物理内存地址

前面讲的命令中，使用的都是虚拟地址，如果要显示和修改物理地址那么需要使用扩展命令!d{b|c|d|p|q|u|w}和!e{b|d}。只有在内核调试时才可以使用这些命令。

30.13.7 观察内存属性

使用扩展命令!address 可以显示某一个内存地址（区域）的特征信息，它的基本格式是：

!address [Address]

其中 Address 是要观察的内存地址，比如，以下是关于 m_szBuffer 变量所在内存的信息：

```
0:000> !address 12fc94

        00030000 : 00126000 - 0000a000
                            Type     00020000 MEM_PRIVATE
                            Protect  00000004 PAGE_READWRITE
                            State    00001000 MEM_COMMIT
                            Usage    RegionUsageStack
                            Pid.Tid  e00.c9c
```

其中第一行的含义是，指定的内存地址属于一个从 00030000 开始的较大内存区（region）中以 00126000 开始的较小内存区，这个内存区的大小是 0000a000。

第二行是内存类型，可以为 MEM_IMAGE、MEM_MAPPED、或者 MEM_PRIVATE，分别代表从执行映像文件映射的内存，从其它文件映射的内存，和私有内存（不是从文件映射的，也不是与其它进程共享的）。

第三行是内存的页属性，可以为 PAGE_READONLY、PAGE_READWRITE、PAGE_READONLY、PAGE_EXECUTE、PAGE_GUARD 等值，参考 MSDN 中关于 VirtualAlloc API 的说明。

第四行是内存的状态，可以为 MEM_COMMIT、MEM_RESERVE、和 MEM_FREE 之一，分别代表已经提交的内存、保留的内存和标志为释放的内存。

第 5 行是内存区的用途，可以为 RegionUsageIsVAD（虚拟地址描述符 VAD）、RegionUsageFree

（空闲）、RegionUsageImage（执行映像的映射）、RegionUsageStack（栈）、RegionUsageTeb（线程环境块）、RegionUsageHeap（堆）、RegionUsagePageHeap（页堆，参见 23 章）、RegionUsagePeb（进程环境块）、RegionUsageProcessParametrs（进程参数）、RegionUsageEnvironmentBlock（环境块）之一。

第 6 行的内容会因为用途的不同而有所不同，因为上面的内存地址属于栈，所以第六行显示的这个栈所属的进程 ID 和线程 ID。

以下是另一个例子，它显示的是字符串常量”AdvDbg”所在地址的信息：

```
0:000> !address 41b954
    00400000 : 0041b000 - 00004000
                        Type        01000000 MEM_IMAGE
                        Protect     00000002 PAGE_READONLY
                        State       00001000 MEM_COMMIT
                        Usage       RegionUsageImage
                        FullPath MfcHello.exe
```

因为常量是作为映像文件的一部分而映射到内存中的，所以可以看到它的用途是 RegionUsageImage，最后一行的信息是映像文件的名称，内存页属性是只读的。

如果不指定地址参数，那么 WinDBG 会显示当前进程（或者内核空间——对于内核调试）中的所有内存区域信息。例如以下是关于 MfcHello 进程的显示结果（省略很多行）：

```
0:000> !address
    00000000 : 00000000 - 00010000 【这是所谓的空指针区，注意其保护属性为 NOACCESS】
                    Type      00000000
                    Protect   00000001 PAGE_NOACCESS
                    State     00010000 MEM_FREE
                    Usage     RegionUsageFree
    00010000 : 00010000 - 00002000
                    Type      00020000 MEM_PRIVATE
                    Protect   00000004 PAGE_READWRITE
                    State     00001000 MEM_COMMIT
                    Usage     RegionUsageEnvironmentBlock
...
    00030000 : 00030000 - 000f5000 【这是为栈分配的内存区，又分为三个小的区】
                    Type      00020000 MEM_PRIVATE
                    State     00002000 MEM_RESERVE 【这个区是保留区】
                    Usage     RegionUsageStack
                    Pid.Tid   e00.c9c
                00125000 - 00001000
                    Type      00020000 MEM_PRIVATE
                    Protect   00000104 PAGE_READWRITE | PAGE_GUARD
                    State     00001000 MEM_COMMIT 【这是所谓的保护页面】
                    Usage     RegionUsageStack
                    Pid.Tid   e00.c9c
                00126000 - 0000a000
                    Type      00020000 MEM_PRIVATE
                    Protect   00000004 PAGE_READWRITE
```

```
                    State     00001000 MEM_COMMIT 【已经提交的使用中部分】
                    Usage     RegionUsageStack
                    Pid.Tid   e00.c9c
…… 【以下是分别按用途、类型和状态所作的统计报告】
------------------- Usage SUMMARY --------------------------
    TotSize (      KB)   Pct(Tots) Pct(Busy)   Usage
     e2b000 (   14508) : 00.69%    47.54%    : RegionUsageIsVAD
   7e222000 ( 2066568) : 98.54%    00.00%    : RegionUsageFree
     c3e000 (   12536) : 00.60%    41.07%    : RegionUsageImage
     100000 (    1024) : 00.05%    03.36%    : RegionUsageStack
       1000 (       4) : 00.00%    00.01%    : RegionUsageTeb
     260000 (    2432) : 00.12%    07.97%    : RegionUsageHeap
          0 (       0) : 00.00%    00.00%    : RegionUsagePageHeap
       1000 (       4) : 00.00%    00.01%    : RegionUsagePeb
       1000 (       4) : 00.00%    00.01%    : RegionUsageProcessParametrs
       2000 (       8) : 00.00%    00.03%    : RegionUsageEnvironmentBlock
       Tot: 7fff0000 (2097088 KB) Busy: 01dce000 (30520 KB)
【以上第一列总字节数，第二列是十进制表示的 KB 字节数，第三列是占总内存的百分比，第四列是占繁忙状态的百分比，第五列是用途】
------------------- Type SUMMARY --------------------------
    TotSize (      KB)   Pct(Tots)  Usage
   7e222000 ( 2066568) : 98.54%    : <free>
     c3e000 (   12536) : 00.60%    : MEM_IMAGE
     ca8000 (   12960) : 00.62%    : MEM_MAPPED
     4e8000 (    5024) : 00.24%    : MEM_PRIVATE

------------------- State SUMMARY --------------------------
    TotSize (      KB)   Pct(Tots)  Usage
    1170000 (   17856) : 00.85%    : MEM_COMMIT
   7e222000 ( 2066568) : 98.54%    : MEM_FREE
     c5e000 (   12664) : 00.60%    : MEM_RESERVE

Largest free region: Base 10320000 - Size 27be0000 (651136 KB)
```

除了!address 命令，还可以使用!vprot 命令来显示一个内存地址的属性，它的显示结果与!address 类似。比如，

```
0:001> !vprot 12fc94
BaseAddress:          0012f000
AllocationBase:       00030000
AllocationProtect: 00000004   PAGE_READWRITE
RegionSize:           00001000
State:                00001000   MEM_COMMIT
Protect:              00000004   PAGE_READWRITE
Type:                 00020000   MEM_PRIVATE
```

也可以使用扩展命令!vadump 来显示当前进程的所有虚拟地址，与不带参数的!address 命令类似，比如：

```
0:001> !vadump -v
BaseAddress:          00000000
AllocationBase:       00000000
RegionSize:           00010000
State:                00010000    MEM_FREE
Protect:              00000001    PAGE_NOACCESS
…….
```

除了以上介绍的命令，在内核调试中，还可以使用!pte 命令来显示指定地址所属的页表表项（PTE）和页目录表项（PDE）。比如：

```
lkd> !pte 801544f4
                VA 801544f4
PDE at 00000000C0602000     PTE at 00000000C0400AA0
contains 0000000000741163  contains 0000000000154163
pfn 741 -G-DA--KWEV     pfn 154 -G-DA—KWEV
```

其中，第三行显示的分别是 PDE 和 PTE 的虚拟地址。第四行是 PDE 和 PTE 的内容，也即：

```
lkd> dd C0602000 l1
c0602000    00741163
lkd> dd C0400AA0 l1
c0400aa0    00154163
```

第五行是将第四行的内容按高 20 位和低 12 位分解为 Page Frame Numer（PFN）和页属性，。PFN 用来转换物理地址，其规则是先将其（PFN）乘以 0x1000（页大小，也就是左移 12 位），然后再加上虚拟地址的页内偏移，即后 12 位。对于上面的地址，其物理地址为：

154×0x1000+4F4=1544F4

分别使用!dd 命令和 dd 观察内存内容，可以验证我们的转换结果是正确的：

```
lkd> !dd 1544f4 l4
#    1544f4 004e0024 00630069 00720061 00670061
lkd> dd 801544f4 l4
801544f4    004e0024 00630069 00720061 00670061
```

页属性中的 G 代表全局（Global），D 代表数据，A 代表访问过（Accessed），K 代表这是内核态拥有的内存页，W 代表可以写，E 代表可以执行，V 代表有效（Valid），即对应的内存页已经在物理内存中。

关于虚拟内存和内存保护的更详细介绍，请参阅第 3 章。

30.14 遍历链表

链表是非常常用的一种数据架构，Windows 操作系统的很多重要数据是以链表方式组织的。在任一用户态调试会话中执行 dt ntdll!*List*可以看到很多链表类型：

```
0:000> dt ntdll!*List*
        ntdll!LIST_ENTRY64
        ntdll!LIST_ENTRY32
        ntdll!_LIST_ENTRY
```

…

在内核调试会话中执行 dt nt!*List*看到类型会更多。

30.14.1 结构定义

Windows 中主要使用两种链表，一种是双向链表，每个节点是一个 LIST_ENTRY 结构：

```
typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;
    struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER PRLIST_ENTRY;
```

另一种是单向链表，每个节点是一个 SINGLE_LIST_ENTRY 结构：

```
typedef struct _SINGLE_LIST_ENTRY {
     struct _SINGLE_LIST_ENTRY *Next;
} SINGLE_LIST_ENTRY, *PSINGLE_LIST_ENTRY;
```

30.14.2 双向链表示例

下面我们通过几个例子来说明，我们先给出一个双向链表的例子。第 10 章我们曾经介绍过，Windows 内核使用 EPROCESS 结构来记录每个 Windows 进程。为了可以方便的遍历所有进程，在 EPROCESS 结构中定义了一个 ActiveProcessLinks 字段，它便是一个 LIST_ENTRY 结构：

```
lkd> dt _EPROCESS -y ActiveProcess
nt!_EPROCESS
    +0x088 ActiveProcessLinks : _LIST_ENTRY
```

因为系统内所有进程的 EPROCESS 结构是通过 ActiveProcessLinks 字段相互链接的，所以只要遍历这个双向链表就可以枚举出系统内的所有进程。因为全局变量 PsInitialSystemProcess 记录了初始的系统进程的 EPROCESS 结构的地址，所以我们就可以使用如下 dt 命令来遍历所有进程：

```
lkd> dt nt!_EPROCESS -l ActiveProcessLinks.Flink -y Ima -yoi Uni poi(PsInitialSystemProcess)
ActiveProcessLinks.Flink at 0x8a6725f0
-------------------------------------------
UniqueProcessId : 0x00000004
ActiveProcessLinks :    [ 0x8a20ae28 - 0x805596b8 ]
ImageFileName : [16]    "System"

ActiveProcessLinks.Flink at 0x8a20ada0
-------------------------------------------
UniqueProcessId : 0x000005a4
ActiveProcessLinks :    [ 0x8a271370 - 0x8a672678 ]
ImageFileName : [16]    "smss.exe"
……
```

需要注意几点，第一，以上命令中 poi(PsInitialSystemProcess)是作为地址参数传递给 dt 命令的，这个地址指向的是系统进程的_EPROCESS 结构，而不是 LIST_ENTRY 结构。使用!process 扩展命令观察 System 进程可以确认这一点：

```
lkd> !process 4 0
```

```
Searching for Process with Cid == 4
PROCESS 8a6725f0    SessionId: none    Cid: 0004       Peb: 00000000    ParentCid: 0000
    DirBase: 00730000    ObjectTable: e1000ca0    HandleCount: 654.
    Image: System
lkd> dd PsInitialSystemProcess l1
805597b4    8a6725f0
```

因此，尽管 dt 命令的结果中提示 ActiveProcessLinks.Flink at 0x8a6725f0，但 at 后的地址并不一定是 LIST_ENTRY 结构。

第一，-l 开关后的字段参数的作用是帮助 dt 命令寻找下一个 LIST_ENTRY 结构的地址，它可以使用“子结构字段.子结构字段.字段”的形式，比如上面指定的是 ActiveProcessLinks.Flink。尽管 Flink 字段指向的是下一个类型结构中的 ActiveProcessLinks 字段地址，而不是 EPROCESS 结构的地址，但我们不用担心，dt 命令会帮我们自动将 ActiveProcessLinks 字段地址减去这个字段在结构中的偏移而得到 EPROCESS 结构的地址。观察上面的命令结果，第一个 EPROCESS 结构的地址是 0x8a6725f0，它的 ActiveProcessLinks.Flink 的内容是 0x8a20ae28。在显示下一个 EPROCESS 结构时，dt 命令提示它的地址为 0x8a20ada0，可见此时 dt 已经将 0x8a20ae28 做了调整，调整的幅度恰好是 ActiveProcessLinks 字段的偏移即 0x88。

```
lkd> ?0x8a20ae28-0x88
Evaluate expression: -1977569888 = 8a20ada0
```

30.14.3 单向链表示例

下面我们再给出一个单向链表的例子，我们就以 TEB 结构中的异常处理器等级列表为例。

```
0:000> dt -r2 _TEB
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
      +0x000 ExceptionList    : Ptr32 _EXCEPTION_REGISTRATION_RECORD
         +0x000 Next             : Ptr32 _EXCEPTION_REGISTRATION_RECORD
         +0x004 Handler          : Ptr32           _EXCEPTION_DISPOSITION
      +0x004 StackBase        : Ptr32 Void
...
```

其中的 ExceptionList 字段是一个 EXCEPTION_REGISTRATION_RECORD 结构，用来记录异常处理器（详见 24 章），它的 Next 字段指向下一个 EXCEPTION_REGISTRATION_RECORD 结构，因此这是一个典型的单向链表结构。

使用~命令列出当前进程的所有线程：

```
0:000> ~
.  0  Id: 1700.164 Suspend: 1 Teb: 7ffdf000 Unfrozen
```

取出其中的 Teb 结构地址 7ffdf000，因为 NtTib 在 TEB 结构的开始处，而 ExceptionList 又是在 NtTib 结构的开始处，所以 TEB 结构的地址也就是 ExceptionList 字段的地址。于是可以使用如下 dt 命令来显示出当前线程的所有异常处理器记录：

```
0:000> dt _EXCEPTION_REGISTRATION_RECORD -l Next 7ffdf000
dbgee!_EXCEPTION_REGISTRATION_RECORD
Next at 0x7ffdf000
-----------------------------------------
   +0x000 Next             : 0x0012ffa8 _EXCEPTION_REGISTRATION_RECORD
```

```
   +0x004 Handler          : 0x00130000      _EXCEPTION_DISPOSITION  +130000

Next at 0x12ffa8
-----------------------------------------
   +0x000 Next             : 0x0012ffe0 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x00411082      _EXCEPTION_DISPOSITION
dbgee!ILT+125(__except_handler4)+0
……
```

30.14.4Dl 命令

除了 dt 命令，也可以使用 dl 命令来遍历链表，其格式为：

dl[b] Address MaxCount Size

其中 Address 是链表的起始地址，它应该指向 LIST_ENTRY 或者 SINGLE_LIST_ENTRY 结构。MaxCount 用来指定要显示的最多节点数。Size 用来指定链表节点的结构长度，或者说希望显示的结构长度，是以指针长度为单位的。

下面举个例子来说明，内核变量 PsActiveProcessHead 是指向进程链表的起始地址的，也就是说它是指向初始系统进程的 EPROCESS 结构中的 LIST_ENTRY 结构的（ActiveProcessLinks 字段）。显示这个指针的值：

```
lkd> dd nt!PsActiveProcessHead l1
805596b8    8a672678
```

将它的值 8a672678 减去 ActiveProcessLinks 字段在 EPROCESS 结构中的偏移 0x88，得到 8a6725f0，这正是 PsInitialSystemProcess 变量的值：

```
lkd> dd PsInitialSystemProcess l1
805597b4    8a6725f0
```

了解以上信息后，我们就可以将 PsActiveProcessHead 作为参数来用 dl 命令来遍历进程链表：

```
lkd> dl nt!PsActiveProcessHead 1000
805596b8    8a672678 87976460 00000001 adc4fa6c
8a672678    8a20ae28 805596b8 00000000 00000000
8a20ae28    8a271370 8a672678 00000280 00001884
8a271370    8a234a60 8a20ae28 000024a0 0002e7fc
……
```

以上结果中，每一行显示一个链表节点，对应一个进程。第一列是 LIST_ENTRY 结构的地址，也就是各个进程的 EPROCESS 结构的 ActiveProcessLinks 字段的地址。第二列是 ActiveProcessLinks 字段的 Flink 子字段的值，第三列是 Blink 子字段值。第四列和第五列 EPROCESS 结构中 ActiveProcessLinks 字段后字段的值。

```
lkd> dt _EPROCESS
nt!_EPROCESS
   ......
      +0x088 ActiveProcessLinks : _LIST_ENTRY
      +0x090 QuotaUsage         : [3] Uint4B
      +0x09c QuotaPeak          : [3] Uint4B
   ……
```

也就是说第四列是 QuotaUsage 字段的值，第五列是 QuotaPeak 字段的值。因为我们没有指定结构长度参数，所以 dl 命令使用默认值显示 4 个指针长度的值，或者说它默认链表节点的结构长度为 4 个指针长度。

从上面结果我们还看到，因为 dl 命令是假定 LIST_ENTRY 结构是定义在链表表项结构的头部的，所以它在显示完 LIST_ENTRY 结构后，便显示与其相邻地址的其它数据。但这种假设有时是不成立的，比如 EPROCESS 结构就把 LIST_ENTRY（ActiveProcessLinks 字段）定义在中间。对于这种情况，我们需要把 dl 命令显示的 ActiveProcessLinks 字段地址减去它的偏移才能得到 EPROCESS 结构的地址。比如：

```
lkd> dt _EPROCESS -y Image 8a672678-0x88
nt!_EPROCESS
    +0x174 ImageFileName : [16]  "System"
lkd> dt _EPROCESS -y Image 8a20ae28-0x88
nt!_EPROCESS
    +0x174 ImageFileName : [16]  "smss.exe"
```

可见 dl 命令尽管简单，但是它的功能比较有限。

30.14.5!list 命令

也可以使用扩展命令!list 来遍历链表，显示链表的每个元素或者对其执行其它命令。!list 命令的格式为：

```
!list -t [Module!]Type.Field -x "Commands" [-a "Arguments"] [Options] StartAddress
!list "-t [Module!]Type.Field -x \"Commands\" [-a \"Arguments\"] [Options] StartAddress"
```

例如，可以使用如下命令来枚举系统内的进程：

```
lkd> !list "-t _EPROCESS.ActiveProcessLinks.Flink -e -x \"dd @$extret l4; dt _EPROCESS @$extret -y Image\" poi(nt!PsInitialSystemProcess)"
dd @$extret l4; dt _EPROCESS @$extret -y Image
8a6725f0  001b0003 00000000 8a6725f8 8a6725f8
nt!_EPROCESS
    +0x174 ImageFileName : [16]  "System"

dd @$extret l4; dt _EPROCESS @$extret -y Image
8a20ada0  001b0003 00000000 8a20ada8 8a20ada8
nt!_EPROCESS
    +0x174 ImageFileName : [16]  "smss.exe"
    .......
```

命令中的$extret 是!list 命令所定义的伪寄存器，用来代表每个链表节点的地址。上面的命令可以分解为如下几个部分：

- 使用-t 开关指定的数据类型和它的链接字段。这里的数据类型是指链表所链接的那个数据结构，在本例中也就是 EPROCESS 结构，为了与它所包含的 LIST_ENTRY 结构相区别，我们将这个较大的数据结构称为**外层结构**（outer structure）。链接字段是供!list 命令寻找下一个元素的。外部结构和链表字段使用点相分隔，与编程语言中的表示很类似，显得也很自然。这与 dt 命令将类型放在前面，然后使用-l 来指定链接字段不同。
- 使用-x 来指定对每个节点所执行的命令，比如我们在上面的例子中指定的命令是：dd @$extret l4; dt _EPROCESS @$extret -y Image

- 使用-e 或或者-m 来指定执行选项，-e 的含义是显示（echo）针对每个节点所执行的命令。-m 加一个数字用来限制最多显示的节点数。

总结一下，dt、dl、和!list 三个命令都可以遍历链表结构，dl 命令最简单，只要向其提供链表表头的地址。!list 命令最复杂，它可以对每个节点执行一系列命令。!list 与 dt 命令的地址参数指定的都是外层结构的地址，而不是 LIST_ENTRY 结构的地址，而 dl 命令的地址参数必须是 LIST_ENTRY 结构（或者 SINGLE_LIST_ENTRY）的地址。

30.15 调用目标程序的函数

本节我们介绍如何使用 WinDBG 的.call 元命令来从调试器中调用被调试程序中的函数。我们结合一个例子来谈。

30.15.1 调用示例

我们使用 WinDBG 打开 dbgee 小程序（调试版本），当 main 函数入口的断点命中时，输入如下命令：

0:000> .call MSVCR80D!wprintf(argv[0])

因为 WinDBG 总是使用 C++表达式评估器来解析这个命令中的函数参数，所以这里我们使用了 C/C++的语法（argv[0]）来引用 dbgee 程序的命令行参数。

发出以上命令后，WinDBG 提示如下信息：

Thread is set up for call, 'g' will execute.

WARNING: This can have serious side-effects,

including deadlocks and corruption of the debuggee.

第一行信息表示 WinDBG 已经为函数调用做好了准备（稍后讨论），输入 g 命令将执行这个函数。因为是在目标程序中执行目标程序的函数，所以 WinDBG 需要恢复目标执行来执行这个函数。后两行信息警告我们这样调用目标程序中的函数可能有严重的副作用，包括导致被调试程序死锁和被破坏。通常我们应该调用比较单纯的函数，这个函数不依赖太多其它函数，它执行完某个操作就立刻返回。

我们输入 g 命令来恢复目标运行，当目标程序立刻又中断到调试器中，并显示：

.call returns:

int 51

以上信息的含义是函数调用的返回值是整数 51（int 51）。同时，函数返回值还被放在伪寄存器$callret 中，可以使用 r 命令来观察：

0:000> r $callret

$callret=00000033

对于多线程程序，为了减小副作用，可以使用~. g 命令来只恢复调用线程。

30.15.2 工作原理

简单来说，.call 命令所执行的主要步骤是：

1) 在栈上插入一小段代码，这段代码被用作要调用函数的父函数，当被调用函数返回时返回到这段代码中，其代码内容就是中断指令，使函数返回后便立刻触发一个断点再中断到调试器中。
2) 在栈上建立一个新的栈帧模拟调用要调用的函数，包括压入参数，压入函数返回地址，即上面的那段代码。
3) 修改寄存器，使程序指针寄存器指向要调用函数的起始地址，使得一恢复当前线程执行便执行这个函数。

分析上面的例子，在执行.call 命令前的栈调用情况为：

```
0:000> kn
 # ChildEBP RetAddr
00 0012ff68 00411ad6 dbgee!wmain [c:\dig\dbg\author\code\chap28\dbgee\dbgee.cpp @ 11]
01 0012ffb8 0041191d dbgee!__tmainCRTStartup+0x1a6 [f:\rtm\...\crtexe.c @ 583]
02 0012ffc0 7c816ff7 dbgee!wmainCRTStartup+0xd [f:\rtm\...\crtexe.c @ 403]
03 0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

寄存器内容为：

```
0:000> r
eax=003a5c20 ebx=7ffd4000 ecx=003a2e90 edx=00000001 esi=0125f774 edi=0125f6f2
eip=004113a0 esp=0012ff6c ebp=0012ffb8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
dbgee!wmain:
004113a0 55              push    ebp
```

记下以上内容后，执行.call 命令，然后再观察栈调用情况，变为：

```
0:000> kn
 # ChildEBP RetAddr
00 0012ff5c 0012ff68 MSVCR80D!wprintf [f:\rtm\vctools\crt_bld\self_x86\crt\src\wprintf.c @ 49]
WARNING: Frame IP not in any known module. Following frames may be wrong.
01 0012ffb8 0041191d 0x12ff68
02 0012ffc0 7c816ff7 dbgee!wmainCRTStartup+0xd [f:\rtm\...\crtexe.c @ 403]
03 0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

可见此时，WinDBG 已经模拟好了在执行 wprintf 函数的假象。第三行的警告信息是说 wprintf 函数的返回地址 0012ff68 不属于任何模块，这是因为这个返回地址是指向栈上那小段代码的，当然不属于任何模块，执行 u 命令可以看到它的内容：

```
0:000> u 0012ff68
0012ff68 cc              int     3
0012ff69 ebfd            jmp     0012ff68
0012ff6b cc              int     3
0012ff6c d6              ???
0012ff6d 1a4100          sbb     al,byte ptr [ecx]
…….
```

从前面的寄存器信息我们知道执行.call 命令前的栈顶指针为 0012ff6c，所以那小段代码其实就是 0012ff68 到 0012ff6b 的三条指令（占四个字节）。

此时的寄存器内容为：

```
0:000> r
```

eax=003a5c20 ebx=7ffd4000 ecx=003a2e90 edx=00000001 esi=0125f774 edi=0125f6f2

eip=1029ed60 esp=0012ff60 ebp=0012ffb8 iopl=0 nv up ei pl nz na po nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202

MSVCR80D!wprintf:

1029ed60 55 push ebp

可见程序指针寄存器已经被修改为 1029ed60，即 wprintf 函数的地址。另外栈顶指针寄存器 ESP 也变了，因为需要为 wprintf 函数压入返回地址和参数。使用 dd 显示栈上的原始数据：

0:000> dd 0012ff60

0012ff60 0012ff68 003a2e98 ccfdebcc 00411ad6

0012ff70 00000001 003a2e90 003a5c20 a87c6bd4

其中 0012ff68 是函数返回地址，即那小段代码的地址。003a2e98 是参数 argv[0]：

0:000> dd poi(argv) l1

003a2e90 003a2e98

第三个 DWORD，ccfdebcc 就是那小段代码的机器码。00411ad6 是 main 函数的返回地址，再后面是 main 函数的参数。

通过以上分析我们便很容易理解.call 命令的工作原理了，简单来说，WinDBG 就是在当前线程的栈上模拟出函数调用的线程，将参数、返回地址和寄存器等准备好，然后恢复目标执行，目标线程恢复后便执行这个要调用的函数。

30.15.3 限制条件和常见错误

首先，.call 命令支持在调试用户态活动目标时使用，不能用在内核态调试和调试转储文件的情况。

第二个条件是必须有被调用函数的私有符号，也就是包含类型信息的函数符号。举例来说，对于如下函数：

0:000> x dbgee!wprint*

004113f6 dbgee!wprintf = <no type information>

等号后信息表明 WinDBG 没有这个符号的类型信息。如果对这个函数执行.call 命令，那么 WinDBG 便会提示错误：

0:000> .call dbgee!wprintf(argv[0])

^ Symbol not a function in '.call dbgee!wprintf(argv[0])'

如果有符号类型，那么检查符号时为类似如下的显示：

0:000> x msvcr80d!wprintf

1029ed60 MSVCR80D!wprintf (wchar_t *)

另外，每个线程一次只能调用一个函数，只有当这次调用结束后，才能再调用另一个。否则便会得到如下错误：

0:000> .call MSVCR80D!wprintf(argv[0])

^ Thread already has call in progress error in '.call MSVCR80D!wprintf(argv[0])'

但可以使用.call /C 来清除当前线程中的函数调用。

30.16 命令程序

类似于 DOS 或者 Windows 控制台中的批处理命令文件和脚本文件，WinDBG 也支持把一系列调试器命令放在一个文件中，然后以文件的形式提交给调试器来执行，这样的文件被称为调试器命令程序（Debugger Command Program），简称命令程序。在命令程序中，除了可以使用 WinDBG 的标准命令、元命令和扩展命令外，还可以使用专门用来控制执行流程的所谓流程控制符号，使用别名和各种伪寄存器来充当变量。下面我们分别来介绍。

30.16.1 流程控制符号

模仿 C/C++语言中的流程控制关键字，WinDBG 定义了一系列元命令和扩展来实现流程控制，统称为流程控制符号（Control Flow Token）。简要列举如下：

- 用作分支和判断的.if、.else 和.elseif。
- 用作循环的.do、.while、!for_each_module、!for_each_frame 和!for_each_local。以及用在循环体中的.break 和.continue。
- 捕捉异常的.catch 和从.catch 块中退出的.leave。
- 定义代码块的.block。因为大括号（{}）已经在别名和很多命令中有用途，所以不可以单独使用大括号来定义代码块。

以上符号的用法大多与 C/C++语言中的响应关键字非常类似，我们不做详细说明，稍后我们会通过几个例子来演示它们的用法。

30.16.2 变量

编写程序总离不开变量，在 WinDBG 命令程序中可以使用如下几种变量：

- 自动的伪寄存器，即 WinDBG 调试器内部已经定义好的一系列寄存器，比如$peb、$ip 等。这些寄存器不需要定义和初始化，系统会将其对应到它所代表的值，可以直接使用。
- 用户赋值的伪寄存器，共有 20 个$t0~$t19。这些伪寄存器的缺省类型是整数。可以使用 r 命令为其赋值，但也可以使用 r?命令让其自动获取所赋参数的类型。比如 r $t1 = 7 是将 $t1 赋值为整数 7；r? $t2 = &@$peb->Ldr 是让 $t2 获取 Ldr 字段的类型 _PEB_LDR_DATA，&符号用来取地址，含义与 C++中相同，如果有&符号，那么$t2 的值为 Ldr 字段的内存地址，比如如果 PEB 结构的地址为 7ffdf000，那么$t2 的值为 7ffdf00c，因为 Ldr 字段的偏移为 0xC，如果不带&符号，那么$t2 的值就是 Ldr 字段的内容。
- 用户定义的别名，可以通过 as 命令来定义别名，然后使用，不用时使用 ad 命令删除。
- 自动别名，如$ntsym、$CurrentDumpFile 等。
- 固定名称的别名，共有 10 个，分别为$u0~$u9。定义固定名称别名的等价量时应该在 u 前加一个点（.），如 r $.u5="dd esp; g"。

如果使用的表达式评估器为 MASM，那么引用伪寄存器的方法是有两种，一种是在$符号前加@符号，另一种是不加@符号，但如果使用 C++表达式评估器，那么一定要加@符号。引用（解释）别名的典型方式是使用${ }。

30.16.3命令程序示例

下面通过例子来说名命令程序的编写方法。清单 30-2 列出了一个典型的命令程序，它可以按照加载顺序列出当前进程中的所有模块。

清单 30-2 命令程序示例

```
1   $$ Get module list LIST_ENTRY in $t0.
2   r? $t0 = &@$peb->Ldr->InLoadOrderModuleList
3
4   $$ Iterate over all modules in list.
5   .for (r? $t1 = *(ntdll!_LDR_DATA_TABLE_ENTRY**)@$t0;
6           (@$t1 != 0) & (@$t1 != @$t0);
7           r? $t1 = (ntdll!_LDR_DATA_TABLE_ENTRY*)@$t1->InLoadOrderLinks.Flink)
8   {
9       $$ Get base address in $Base.
10      as /x ${/v:$Base} @@c++(@$t1->DllBase)
11
12      $$ Get full name into $Mod.
13      as /msu ${/v:$Mod} @@c++(&@$t1->FullDllName)
14
15      .block
16      {
17          .echo ${$Mod} at ${$Base}
18      }
19
20      ad ${/v:$Base}
21      ad ${/v:$Mod}
22  }
```

在以上清单中，1、4、9、12 行都是注释行，尽管*也可以开始一个注释行，但是在命令程序中通常需要使用$$，其原因是命令程序执行时会变自动合并为一行，换行符被替换为分号，而*注释符是注释整行，$$是注释到分号为止。

第二行是把当前进程_PEB 结构（使用伪寄存器$peb 代表）的 Ldr 子结构的 InLoadOrderModuleList 字段的地址赋给$t0 伪寄存器，并使之自动获得 InLoadOrderModuleList 字段的类型_LIST_ENTRY。

```
0:000> dt -r2 _PEB 7ffdf000
ntdll!_PEB
…
   +0x00c Ldr              : 0x00251ea0 _PEB_LDR_DATA
      +0x000 Length           : 0x28
      +0x004 Initialized      : 0x1 ''
      +0x008 SsHandle         : (null)
      +0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x251ee0 - 0x252220 ]
         +0x000 Flink            : 0x00251ee0 _LIST_ENTRY [ 0x251f48 - 0x251eac ]
         +0x004 Blink            : 0x00252220 _LIST_ENTRY [ 0x251eac - 0x252160 ]
```

也就是 Ldr 指针的值是 0x00251ea0，这就是_PEB_LDR_DATA 结构的地址，因为 InLoadOrderModuleList 字段在结构中的偏移是 0xC，所以$t0 的值应该为 00251eac，使用?命令显示，果然如此：

```
0:000> r? $t0 = &@$peb->Ldr->InLoadOrderModuleList
0:000> ? @$t0
Evaluate expression: 2432684 = 00251eac
```

第 5～7 行是开始一个 for 循环，与 C++中的 for 语句类似，第 5 是给循环变量（伪寄存器$t1）赋初值，也就是让$t1 等于$t0 所代表地址处的值，也就是_LIST_ENTRY 结构的 FLink 字段的内容 00251ee0。并且把这个内容转换为 ntdll!_LDR_DATA_TABLE_ENTRY*结构。在 WinDBG 中分别执行这几条命令，可以验证以上分析：

```
0:000> r? $t1 = *(ntdll!_LDR_DATA_TABLE_ENTRY**)@$t0;
0:000> ? $t1
Evaluate expression: 2432736 = 00251ee0
0:000> dd 00251eac l1
00251eac  00251ee0
```

第 6 句是循环条件，即 Flink 的值（$t1）不为空，并且 Flink 不等于$t0 所代表的起始节点地址。这也是遍历链表的典型判断方法。第 7 句是更新循环变量，为下一轮循环做准备。

第 8 到 22 行是循环体，也就是显示$t1 所指向的_LDR_DATA_TABLE_ENTRY 结构的内容。我们先使用 dt 命令观察一下这个结构：

```
0:000> dt ntdll!_LDR_DATA_TABLE_ENTRY 00251ee0
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x251f48 - 0x251eac ]
   +0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x251f50 - 0x251eb4 ]
   +0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x018 DllBase          : 0x00400000
   +0x01c EntryPoint       : 0x0041107d
   +0x020 SizeOfImage      : 0x1a000
   +0x024 FullDllName      : _UNICODE_STRING "C:\dig\dbg\...\Debug\dbgee.exe"
   +0x02c BaseDllName      : _UNICODE_STRING "dbgee.exe"
   +0x034 Flags            : 0x5000
   +0x038 LoadCount        : 0xffff
   +0x03a TlsIndex         : 0
   +0x03c HashLinks        : _LIST_ENTRY [ 0x7c97c1d8 - 0x7c97c1d8 ]
   +0x03c SectionPointer   : 0x7c97c1d8
   +0x040 CheckSum         : 0x7c97c1d8
   +0x044 TimeDateStamp    : 0x46a2b5b8
   +0x044 LoadedImports    : 0x46a2b5b8
   +0x048 EntryPointActivationContext : (null)
   +0x04c PatchInformation : (null)
```

可见以上结构包含了模块的大多数重要信息，包括基地址，入口点（即编译器插入的入口函数，对于.exe 模块通常为_wmainCRTStartup），模块大小、名称等等。

第 10 行是定义一个用户命名的别名$Base，用其表示模块的基地址。

as /x ${/v:$Base} @@c++(@$t1->DllBase)

其中的@@c++用来强制使用 C++表达式评估器，/x 是使这个别名取后面表达式的 64 位值，/v 用来阻止别名替换，不管其是否已经定义，因为这句尚是在定义别名阶段，省略亦可，加上更

稳妥。

第 13 行是定义另一个别名 Mod，使其的值等于 FullDllName 字段的地址。

as /msu ${/v:$Mod} @@c++(&@$t1->FullDllName)

其中的/msu 用来使别名 Mod 等价于后面地址处的 UNICODE_STRING，因为它要求后面是一个地址，所以小括号中的取地址符号&不能省略，否则便会产生如下错误：

```
0:000> as /msu ${/v:$Mod} @@c++(@$t1->FullDllName)
Type conflict error at '@@c++(@$t1->FullDllName)'
```

此时使用 al 命令显示别名列表，可以看到 Mod 和 Base 别名和它们的值：

```
0:000> al
  Alias                 Value
 -------              -------
 $Base                  0x400000
 $Mod                    C:\dig\dbg\author\code\chap28\dbgee\Debug\dbgee.exe
…
```

第 15 到 18 行定义了一个块，尽管其中只有一行命令，这个块定义仍是必要的，它起到的作用是强制评估块中的所有别名，如果省略 15、16 和 18 行，那么执行对应的命令文件得到的结果为：

```
0:000> $$><c:\dig\dbg\author\code\chap30\lm_bad.dbg
${$Mod} at 0x400000
${$Mod} at 0x400000
${$Mod} at 0x400000
${$Mod} at 0x400000
${$Mod} at 0x400000
```

第 17 行的是用.echo 命令显示别名$Mod 和$Base 的值。

第 20 和 21 行是删除别名定义，然后开始下一轮循环。

30.16.4执行命令程序

将清单 30-2 所示的内容保存为一个文件，然后便可以在 WinDBG 中通过如下命令来执行它：

```
$><c:\dig\dbg\author\code\chap30\lm.dbg
```

以下是在调试 dbgee 程序的调试会话中的执行结果：

```
C:\dig\dbg\author\code\chap28\dbgee\Debug\dbgee.exe at 0x400000
C:\WINDOWS\system32\ntdll.dll at 0x7c900000
C:\WINDOWS\system32\kernel32.dll at 0x7c800000
C:\WINDOWS\WinSxS\x86_Microsoft.VC80…-ww_f75eb16c\MSVCR80D.dll at 0x10200000
C:\WINDOWS\system32\msvcrt.dll at 0x77c10000
```

$><的含义是让 WinDBG 读取后面的文件，并将其中的内容浓缩成一个单一的命令，然后执行，浓缩时，WinDBG 会自动把换行符替换为分号。

如果不希望 WinDBG 进行浓缩处理那么可以将$><换为$>，这时 WinDBG 每次从文件中读取一行，然后执行。对于我们上面的 lm.dbg 文件，这样执行会有错误：

```
0:000> $<c:\dig\dbg\author\code\chap30\lm.dbg
0:000> $$ Get module list LIST_ENTRY in $t0.
0:000> r? $t0 = &@$peb->Ldr->InLoadOrderModuleList
0:000>
```

```
0:000> $$ Iterate over all modules in list.
0:000> .for (r? $t1 = *(ntdll!_LDR_DATA_TABLE_ENTRY**)@$t0;
                ^ Syntax error in '.for (r? $t1 = *(ntdll!_LDR_DATA_TABLE_ENTRY**)@$t0;'
```

从以上显示结果可看到，lm.dbg 文件中的第 1 到 4 行还是可以顺利执行的，但是到第 5 行时由于这一行并没有包含完整的.for 命令所以出错了。

也可以使用$$><或者$$<来执行一个命令文件，它们与$><和$<的差异就是允许在文件名前有空格，并允许使用双引号包围文件名。

如果要为命令文件指定参数，那么可以使用如下形式：

```
$$>a< Filename arg1 arg2 arg3 ... argn
```

在命令程序中可以像使用别名那样使用参数，比如${$arg1}。

30.17 控制进程和线程

很多软件是由多个进程所构成的一个系统，每个进程内可能还包含着多个线程。随着多核处理器的出现和迅速发展，越来越多的软件开始考虑如何通过并行化提高软件的执行速度。而并行化的一个基本方式就是多线程，也就是将本来在一个线程中串行执行的任务分解成多个任务放到多个线程中并行执行。

就像编写多线程程序比编写单线程程序复杂一样，调试多线程程序通常也要比调试单线程程序难度更大。本节我们将先介绍调试多线程程序和同时调试多个进程所需掌握的基本知识和要领。我们首先介绍控制线程执行的主要命令，然后介绍如何同时调试多个进程。

30.17.1 MulThrds 程序

为了便于说明后面要介绍的内容，我们特意编写一个可以动态创建线程的小程序，名为 MulThrds（Multi-Threads 之意）。它的界面如图 30-8 所示。

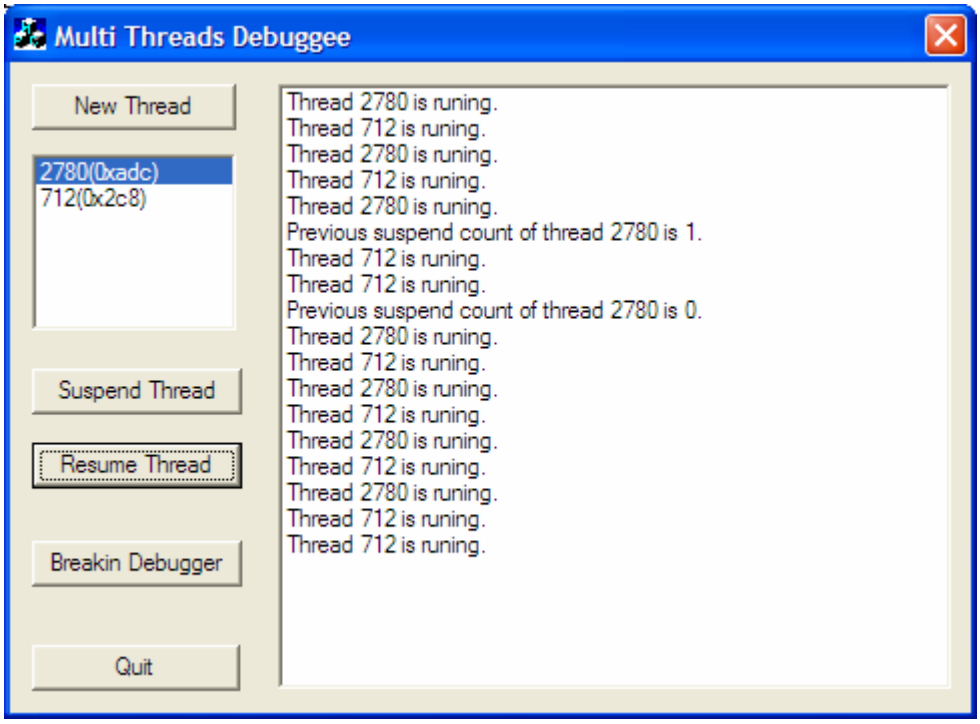


图 30-8 MulThrds 程序的界面

起初运行时，MulThrds 是单线程的，只有用户接口（UI）线程。每次点击 New Thread 按钮时，UI 线程会创建一个新的工作线程，工作线程的任务就是没秒钟向右侧的列表框中加一条消息，显示自己的 ID 和说明自己在运行。每个工作线程的 ID 会被放入到 New Thread 按钮下面的的列表框中。在这个列表框中选中一个线程 ID 后，再点击下面的 Suspend Thread 按钮会触发 UI 线程调用 SuspendThread API 来增加选中线程的挂起计数（Suspend Count），点击下面的 Resume Thread 按钮会触发调用 ResumeThread API 来减少挂起计数。挂起计数是线程的一个基本属性，当一个线程的挂起计数大于 0 时，那么这个线程处于被挂起状态（Suspended），不会被执行。只有当挂起计数降低到 0 时，它才会被执行。因此当选中一个线程第一次点挂起按钮时，对应的线程会被挂起而停止运行，当点恢复按钮时，会恢复其执行。

Breakin Debugger 按钮会触发 UI 线程调用 DebugBreak API，因此当这个程序处于被调试状态时按这个按钮会将其中断到调试器中。

30.17.2 控制线程执行

通常，当被调试程序被中断到调试器中时，它的所有线程都是被挂起的；当恢复执行时，也是恢复所有线程都开始执行。但是在调试时，也可以根据调试任务的需要而单独恢复某个或者某些线程。WinDBG 提供了几种方法。

第一种方法是通过增加线程的挂起计数来禁止线程被恢复运行。当调试目标中断到调试器中时，WinDBG 会对所有线程依次调用 SuspendThread API，当恢复程序执行时，再对所有线程调用 ResumeThread API。

因此当我们在调试器中观察线程时，每个线程的挂起计数通常是 1。以 MulThrds 程序为例，点击两次 New Thread 按钮后，点击 Breakin Debugger 使其中断到调试器中，使用～命令列出所有线程：

```
0:000> ~
.  0  Id: 1440.1554 Suspend: 1 Teb: 7ffde000 Unfrozen
   1  Id: 1440.2c8 Suspend: 1 Teb: 7ffdd000 Unfrozen
   2  Id: 1440.adc Suspend: 1 Teb: 7ffdc000 Unfrozen
```

第一列是线程序号，Id：后面是进程 ID 和线程 ID，之后便是挂起计数，而后是线程环境块的 Teb 的地址。最后一列是线程的冻结状态，我们稍后再详细介绍。

此时可以使用~*Thread* n 命令来增加 Thread 参数所指定的线程。比如输入如下命令可以增加 1 号线程的挂起计数：

```
0:000> ~1 n
```

此时再观察线程状态，可以看到 1 号线程的挂起计数变成了 2。

```
0:000> ~1
   1  Id: 1440.2c8 Suspend: 2 Teb: 7ffdd000 Unfrozen
```

此时输入 g 命令恢复目标程序执行，恢复执行后，我们可以看到只有一个工作线程在活动了，也就是说有一个工作线程没有恢复执行。

与~n 命令相对，~m 命令用来减少线程的挂起计数。比如~1 m 可以将 1 号线程的挂起计数减 1。当线程的挂起计数降低到-1 后，不可再降低。

从实现角度来看，~n 命令就是调用 SuspendThread，而~m 命令就是调用 ResumeThread API。以下是执行~n 命令的过程：

```
0:002> kn
 # ChildEBP RetAddr
```

00 015de3fc 0229b769 kernel32!SuspendThread

01 015de418 020f321d dbgeng!LiveUserDebugServices::SuspendThreads+0x39

02 015de860 02228427 dbgeng!LiveUserTargetInfo::SuspendThreads+0x2fd

03 015de8fc 021874e2 dbgeng!ParseThreadCmds+0x697

04 015de9d8 021889a9 dbgeng!ProcessCommands+0x412

05 015dea1c 020cbec9 dbgeng!ProcessCommandsAndCatch+0x49

06 015deeb4 020cc12a dbgeng!Execute+0x2b9

07 015deee4 01028553 dbgeng!DebugClient::ExecuteWide+0x6a

08 015def8c 01028a43 windbg!ProcessCommand+0x143

09 015dffa0 0102ad06 windbg!ProcessEngineCommands+0xa3

0a 015dffb4 7c80b6a3 windbg!EngineLoop+0x366

0b 015dffec 00000000 kernel32!BaseThreadStart+0x37

终上，通过~n 和~m 命令来改变线程的挂起计数可以帮助我们选择性的让某些线程执行，而某些不执行。

控制线程执行的第二种方法是使用~f 和~u 命令，前者用来冻结（Freeze）一个线程，后者用来解冻（Unfreeze）。当一个线程处于冻结状态时，恢复目标执行时这个线程不会恢复执行。

例如，以下命令可以冻结 1 号线程：

0:000> ~1 f

观察线程状态，可以看到其冻结状态为 Frozen（被冻结）。

0:000> ~1

1 Id: 1440.2c8 Suspend: 1 Teb: 7ffdd000 Frozen

输入 g 命令恢复目标执行，WinDBG 会提示有一个线程在冻结状态。

0:000> g

System 0: 1 of 3 threads are frozen

程序恢复执行后，从 UI 上看，也可以发现 1 号线程没有恢复执行。

从实现角度来看，与~n 和~m 命令是调用操作系统的 API 来改变线程的系统属性（挂起计数）不同，~f 和~u 命令完全是改变调试器的线程设置。当执行~f 和~u 命令时，调试器引擎内部都是调用 ThreadInfo 类的 ChangeFreeze 方法，修改线程的属性信息。通常，当恢复程序执行时，调试器会对所有线程依次调用 ResumeThread API。而一旦某个线程被设置为冻结状态，那么 WinDBG 便不再对其调用 ResumeThread API，因此这个线程也就不会恢复执行。从操作系统的角度来看这个线程是处于被挂起状态。但值得提醒的是，如果使用~m 命令将一个冻结线程的挂起计数降为 0，然后恢复目标程序执行，那么这个线程是会恢复运行的。

控制线程执行的第二种方法就是在恢复执行的命令前通过线程限定符和线程号码只恢复执行指定的线程。比如，不管目标程序有多少个线程，命令~0 g 都只是恢复 0 号线程执行。

0:000> ~0 g

System 0: 2 of 3 threads are frozen

从实现角度来讲，WinDBG 在执行~0 g 命令时就是只对 0 号线程调用 ResumeThread API，对其它线程不调用，使它们处于挂起状态。

值得说明的是，当这样只恢复一个线程执行后，如果试图使用 Ctrl+Break（或者菜单 Debug>Break）来俘获调试目标时，会显示如下信息：

System 0: 2 of 4 threads were frozen

Create thread 3:16fc 【要看到此行信息需要将线程创建事件的处理方式改为 Output】

System 0: 3 of 4 threads are frozen

Break-in sent, waiting 30 seconds...

我们知道，WinDBG 执行 Ctrl+Break 的方法是在目标进程中创建一个远程线程（对于 XP 或更高版本的 Windows 使用 DebugBreakProcess API），然后让这个线程执行一个 DebugBreak 中断到调试器中。因此上面的第 1、2 行信息就是 WinDBG 在收到这个新线程创建的调试事件时所输出的信息。在 MulThrds 进程中本来有三个线程，我们只恢复 0 号执行，所以本来是 3 个中的 2 个被冻结，这里新创建了一个，因此变成 4 个中的 2 个被冻结。第 3 行信息是当 WinDBG 处理好新线程创建事件让目标继续执行时而显示的，因为当前的调试器状态是只恢复 0 号线程执行，因此这里不会对新创建的线程调用 ResumeThread，所以它也被冻结了。第 4 行信息是 Ctrl+Break 命令的输出，它告诉我们 Break-in 信号（线程）已经发出去了，等待 30 秒。事实上显示这条消息时，WinDBG 就已经等待一会了。

再过 30 秒后，调试目标因为调试引擎模拟的异常事件而被调试器挂起，调试器显示如下信息：

WARNING: Break-in timed out, suspending.

This is usually caused by another thread holding the loader lock

(1440.1554): Wake debugger - code 80000007 (first chance)

System 0: 3 of 4 threads were frozen

……

上面第一行信息警告我们这次 Ctrl+Break 命令的执行方法与平常并不相同，发送的 Break-in 信号（线程）超出规定时间也没有回复（断点事件），因此这次不是开远程线程触发断点中断的，而是使用调试器强行挂起目标进程的方法。

类似的，也可以在单步跟踪命令前加上线程限定符，这样保证只有指定的线程会被单步执行，其它线程不会执行。例如以下命令是让 0 号线程单步执行一次：

0:000> ~0 t

System 0: 2 of 3 threads are frozen

System 0: 2 of 3 threads were frozen

第 2 行信息是当恢复目标执行时调试引擎给出的提示，告诉我们目标进程共有 3 个线程，其中两个是被冻结的。也就是说，调试器不会对这两个线程调用 ResumeThread，因此它们不会恢复执行。其中的 System 0 是系统编号，当同时调试多个系统上多个进程时这个信息很有用。

第 3 行信息是单步执行后，调试器收到单步事件后准备进入命令时打印出的，告诉我们调试目标在上次执行时 3 个线程中的 2 个是被冻结着的。注意这两条提示信息很类似，区别的主要方法就是后一句是使用过去时态的。

30.17.3 多进程调试

WinDBG 支持使用一个调试器来调试多个进程，这些进程可以在一个系统（操作系统）上，也可以在多个系统上。我们先介绍同时调试与调试器处于同一个系统中的多个进程的情况。

我们继续使用上面调试 MulThrds 程序的调试会话。我们可以通过.attach 命令把记事本进程（进程 ID 为 2788）加入到调试会话中：

0:003> .attach 0n2788

Attach will occur on next execution

提示信息告诉我们，调试器已经做了必要的登记，但是真正的附加动作需要等下次恢复调试目标执行时发生。事实上，此时记事本程序已经被挂起了。

执行任意一个恢复目标执行的命令，比如 g，让目标恢复执行。这时 WinDBG 会提示有悬而未决的附加操作：

*** wait with pending attach

很快，WinDBG 会进入命令模式，新附加的进程中断到调试器中。WinDBG 的命令提示符为类似 1:004 的样子，表示当前上下文是 1 号进程的 4 号线程。

使用~命令可以列出当前进程的所有线程：

```
1:004> ~
   3  Id: ae4.8d4 Suspend: 1 Teb: 7ffdf000 Unfrozen
.  4  Id: ae4.c30 Suspend: 1 Teb: 7ffde000 Unfrozen
```

因为线程是全局编号的，0～2 号分给了 MulThrds 进程的三个线程。

至此，当前调试器已经和两个进程建立了调试关系。我们可以使用前面介绍的调试命令来分析和控制这两个进程。比如可以单独让 4 号进程恢复执行：

```
1:003> ~3 g
System 0: 4 of 5 threads are frozen
```

这时，只有 Notepad 的 UI 线程是恢复执行的，Notepad 的另一个线程和 MulThrds 进程的所有线程都是被挂起的。

也可以利用我们前面介绍的线程控制命令来控制哪些线程执行，哪些不执行，这在调试存在相互协作的多个进程时可能非常有价值。

利用|<进程号> s 命令可以切换当前进程，使用~<线程号> s 可以切换当前线程。

如果要调试位于多个系统中的进程，那么可以在.attach 命令通过-premote 开关来指定另一个系统的位置和通信方式。类似的.create 命令也支持-premote 开关。-premote 参数的写法与建立远程调试相同，请参阅 30.5.7 节。

30.18 本章总结

本章比较详细的介绍了 WinDBG 调试器的基本用法，覆盖了各种常用的功能和调试器命令。WinDBG 是个多用途的调试器，可以用作内核态调试器、用户态调试器，也可以用来分析应用程序和系统的转储文件。因为篇幅限制，我们没有按照内核态调试、用户态调试、和调试转储文件的分类方法来做分别介绍，而是抽取这些调试类型中的共同任务，介绍其中的一般知识和要领。

WinDBG 的帮助文件是学习 WinDBG 的一个宝贵资源，它详细的介绍了 WinDBG 的全部功能和命令。在制定本章的写作计划时，作者的目标是弥补帮助文件的不足，对数百万字的帮助信息进行归纳和浓缩，使读者可以在较短的时间内了解到 WinDBG 的全貌。另外，我们选取了帮助文件中介绍较少或者较难理解的内容，比如遍历链表（14 节）、事件处理（第 9 节）、调试上下文（第 7 节）等。本章的目标是帮助大家更好的使用帮助文件，而不是替代它。建议大家在阅读本章时和日常调试时都经常打开帮助文件，慢慢加深和细化对每个命令和功能的理解。

除了 WinDBG 的帮助文件，WinDBG 工具包中还包含了以下几个文档：

- kernel_debugging_tutorial.doc – 位于 WinDBG 的根目录中，详细的介绍了如何使用 WinDBG 进行内核调试。
- symhttp.doc - 位于 symproxy 子目录中，介绍了如何建立符号服务器。
- srcsrv.doc – 位于 sdk\srcsrv 子目录中，详细的介绍了源文件服务器的概况和如何建立和配置源文件服务器。
- dml.doc - 位于 WinDBG 的根目录中，介绍了 DML（Debugger Markup Language）的用途和编写方法。DML 是一种标记语言，用于标记 WinDBG 或者扩展命令的信息输出。
- themes.doc – 位于 themes 目录中，介绍了主题（theme）的概念（一个主题代表一套特定风格的界面布局和工作空间配置）和如何加载和使用该目录中的四套主题配置。
- tools.doc -位于 WinDBG 的根目录中，介绍了 PDBCopy 和 dbh（DbgHelp Shell）两个

WinDBG 附带的命令行工具的用法。前者主要用来复制调试符号，后者用来调用调试辅助库（DbgHelp.dll）的各种功能，包括处理模块和调试符号。

- pooltag.txt – 位于 triage 目录中，包含了 Windows 内核模块和驱动程序所使用的内存分配标记（Pool Tag）。当启用了 Windows 操作系统的内存池标记（Pool Tagging）功能后（Windows Server 2003 开始永久启用，之前的 Windows 需要用 GFlags 来启用），每个内存块可以有一个分配标记来标识它的使用者。用于显示内存池使用情况的扩展命令!poolused 就是使用这个文件来查找每个分配标记所对应的模块。

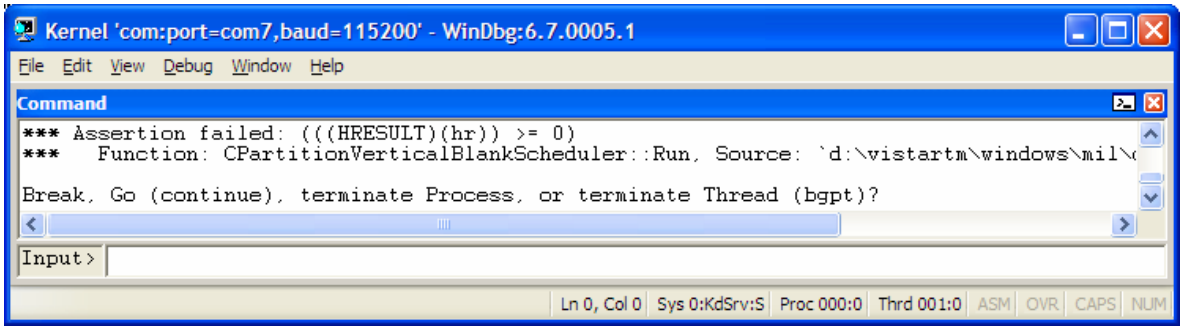
互联网上可以搜索到更多关于 WinDBG 的文档、讨论和博客，在此不一一列举。

参考文献

1. WinDBG 的帮助文件（debugger.chm）
2. WinDBG SDK 中的 Debug Help Library 文档 （位于 WinDBG\sdk\help 目录中的 dbghelp.chm）
3. WinDBG SDK 中的 dbgeng.h 文件

READ: Received Data packet with unmatched ID = 80800800, acking
Throttle 0x10 write to 0x1
PacketType=0, ByteCount=6, PacketId=10000,
READ: Received Data packet with unmatched ID = 10000, acking
Throttle 0x10 write to 0x1
PacketType=7, ByteCount=f4, PacketId=80800800,
READ: Received Data packet with unmatched ID = 80800800, acking
Throttle 0x10 write to 0x1
PacketType=0, ByteCount=6, PacketId=10000,
READ: Received Data packet with unmatched ID = 10000, acking
Throttle 0x10 write to 0x1
PacketType=7, ByteCount=f4, PacketId=80800800,
READ: Received Data packet with unmatched ID = 80800800, acking
Throttle 0x10 write to 0x1
PacketType=0, ByteCount=6, PacketId=10000,
READ: Received Data packet with unmatched ID = 10000, acking
Throttle 0x10 write to 0x1

Waiting to reconnect...
Will request initial breakpoint at next boot.
Will breakin on first symbol load at next boot.



SYNCTARGET: Timeout.
Throttle 0x10 write to 0x1
SYNCTARGET: Timeout.
Throttle 0x10 write to 0x1
Throttle 0x10 write to 0x1
SYNCTARGET: Received KD_RESET ACK packet.
SYNCTARGET: Target synchronized successfully...
 Done.
READ: Wait for type 7 packet
Attempting to get initial breakpoint.
Send Break in ...
        PacketType=7, ByteCount=112, PacketId=80800000,
READ: Received Type 7 data packet with id = 80800000 successfully.

kd> g
Assertion failure - code c0000420 (first chance)
nt!KeUpdateRunTime+0x248:
81867a71 cd2c   int   2Ch
kd> g
Continuing an assertion failure can result in the debuggee
being terminated (bugchecking for kernel debuggees).
If you want to ignore this assertion, use 'ahi'.
If you want to force continuation, use 'gh' or 'gn'.
kd> g
Continuing an assertion failure can result in the debuggee
being terminated (bugchecking for kernel debuggees).
If you want to ignore this assertion, use 'ahi'.
If you want to force continuation, use 'gh' or 'gn'.
kd> ahi
nt!KeUpdateRunTime+0x248 (81867a71) - ignore

```
kd> g
0:001> |
.  0id: c10  attach   name: C:\WINDOWS\system32\notepad.exe
```