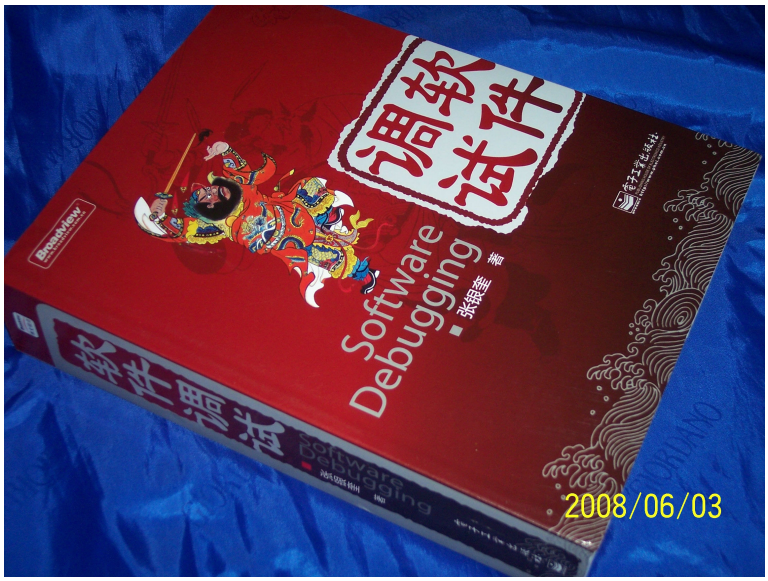
《软件调试》补编

作者：张银奎



2009 年 1 月 12 日

大多数程序员的技术水平不如黑客的主要原因是他们远远不如黑客那样重视和擅于使用调试技术。

前　言

总的来说，今天的软件很糟糕，而且一段时期内还会继续变得更糟。原因有很多，关键的是错误观念大行其道，聊举数例。

“写代码很容易，不需要那么资深的人”，于是乎软件白领变蓝领了，蓝领还是大材小用，“软件民工刚刚好，性价比最高。”

“我从来不调试我的程序，运行一下没有错误就可以了。”这是为什么使用调试器来看时，很多软件有那么多明晃晃的问题。

“XXX 很快呀”，还可以用它来写操作系统呢？！当我启动一个应用程序需要 5 秒钟以上时，看着硬盘的灯疯狂的闪动，我恨不得快点把它从我的系统中删掉。

“抓 BUG 完全是程序员自己的事。”那么项目延迟了还是程序员自己的事么？

“今天的硬盘空间大，CPU 速度快，内存条便宜，软件大一些，多用些资源没关系。”

事实上，没有简单的软件，无论是上层的应用程序，还是下层的驱动或者系统程序。

软件的特征决定了软件天生就需要像绣花那样精工细作。软件是给 CPU 来运行的，让高速的 CPU 在软件编排的指令流上奔跑可谓是系千钧于一发。糟糕的软件在浪费能源、也在浪费时间，如果套用鲁迅先生说的话，那么糟糕的软件每天都在图财害命。

如何才能让软件变得很精细，准确无误呢？不仔细看一看可以做到么？

有些软件可以工作，但是有时会出问题，糟糕的是，出了问题后没有什么办法来寻找原因。在 XXX 做了一年多的开发之后，对此深有体会，一次我向在 XXX 上工作多年的一个朋友询问：“在 XXX 上调试主要靠什么方法呢？”他的回答颇令人回味：“靠想！”

“使劲想，然后加些 PRINT，逐步缩小范围……”

多么好的程序员呀！

老雷

2009 年元月

《软件调试》书友活动获奖名单

整理和发布这份补编的主要目的是赠送给参加“2008《软件调试》以书会友”活动的朋友们。这次活动从 2008 年 6 月 1 日开始，截止日期为 2008 年 12 月 31 日。从 6 月 11 日 Neilshu 第一个参与，到 12 月 31 日的 23 点 47 分 Vito1997 参与，共有 22 位朋友参加了这次活动，收到照片大约 100 幅。

参与这次活动的 22 位朋友是：

Casechen	Neilhsu
Ccl	Nightxie
ckj1234	Pch
Coding	s5689412
Dbgsun	shamexln
Flyingdancex	speedingboy
grant_fei2003@hotmail.com	turboc
hnsy777	Vito1997
KernelPanic	WANGyu
Mabel	xszhou1997
Mybios	yfliu

经过博文视点的周老师和《软件调试》这本书的编辑团队以及作者的认真评比，获奖结果如下：

一等奖一名：Neilhsu

奖品为《奔腾 4 全录：IA32 处理器宗谱》（The Unabridged Pentium 4: IA32 Processor Genealogy）作者签名英文原版

二等奖两名：mybios 和 nightxie

奖品为《深入解析 Windows 操作系统 第 4 版》

三等奖三名：yfliu、WANGyu 和 shamexln

奖品为《Windows 用户态程序高效排错》

所有参加活动的朋友都获得纪念奖，奖品是电子版本的《<软件调试>补编》。

衷心感谢参加这次活动的所有朋友，愿我们的友谊永驻！

目　　录

补编内容 1 错误提示机制之消息框

补编说明：

这一节本来属于《软件调试》第 13 章的第 1 节，旨在介绍消息框这种简单的错误提示机制。凡是做过 Windows 编程的人都知道，消息框用起来很简单，但是大多数人没有仔细思考过它内部是如何工作的，这个问题其实不是很容易说清楚的。

在《软件调试》正式出版前压缩篇幅时，这一节被删除了。主要原因是相对于其它内容，这个内容略显次要，作者也担心有人会提出这样的质疑：“花好几页就写个消息框实在是不值得!”。

13.1 MessageBox

消息对话框（message box）是 Windows 中最常见的即时错误提示方法。利用系统提供的 MessageBox API，弹出一个图形化的消息框对程序员来说真是唾手可得。而且不论是程序本身带有消息循环的 Win32 GUI 程序，还是用户代码中根本没有消息循环的控制台程序，都可以调用这个 API，清单 13-1 所示的代码显示了名为 MsgBox 的控制台程序是如何调用 MessageBox API 的。

清单 13-1 MsgBox 程序的源代码（部分）

```
#include <windows.h>

void main()
{
   MessageBox(NULL, "Simplest way for interactive Instant Error Notification",
      "Instant Error Notification",   MB_OK);
}
```

编写过 Windows 程序消息循环的读者看了清单 13-1 中的代码很可能有个疑问：Windows 窗口都是靠清单 13-2 所示的消息循环来驱动的，但上面的控制台程序根本没有消息循环，消息窗口是如何工作的呢？

清单 13-2 Windows 程序的消息循环

```
   while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0)
   {
      if (bRet == -1)
      {
         // handle the error and possibly exit
      }
      else
      {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
      }
```

```
}
```

使用 WinDBG 跟踪 MessageBox 函数的执行过程，就可以回答上面的问题了。清单 13-3 显示了 MessageBox 函数的内部执行过程。

清单 13-3　MessageBox 函数的执行过程

```
0:000> knL
 # ChildEBP RetAddr
00 0012f990 77d48b04 SharedUserData!SystemCallStub    // 进入内核态执行
01 0012f994 77d48ae7 USER32!NtUserPeekMessage+0xc     // 调用子系统的内核服务
02 0012f9bc 77d48c07 USER32!_PeekMessage+0x72         // 内部函数
03 0012f9e8 77d4e5d9 USER32!PeekMessageW+0xba         // 调用查取消息的 API
04 0012fa30 77d53e2a USER32!DialogBox2+0xe2           // 显示对话框并开始消息循环
05 0012fa58 77d6e6a8 USER32!InternalDialogBox+0xce    // 创建对话框
06 0012fd10 77d6e12b USER32!SoftModalMessageBox+0x72c // 动态产生对话框资源
07 0012fe58 77d6e7ef USER32!MessageBoxWorker+0x267    // 工作函数
08 0012feb0 77d6e8d7 USER32!MessageBoxTimeoutW+0x78   // UNICODE 版本
09 0012fee4 77d6e864 USER32!MessageBoxTimeoutA+0x9a   // 带超时支持的消息框函数
0a 0012ff04 77d6e848 USER32!MessageBoxExA+0x19        // 统一到 MessageBoxEx API
0b 0012ff1c 0040103e USER32!MessageBoxA+0x44          // 调用 MessageBox API
0c 0012ff80 00401199 msgbox!main+0x2e                 // main 函数
0d 0012ffc0 77e8141a msgbox!mainCRTStartup+0xe9       // C 运行库的入口函数
0e 0012fff0 00000000 kernel32!BaseProcessStart+0x23   // 进程的启动函数
```

通过以上函数调用序列，我们可以很清楚地看出 MessageBox API 的执行过程。下面逐步来进行分析。因为 k 命令是照按从（栈）顶到（栈）底的顺序显示的，所以函数调用的关系是下面的调用上面的。最下面的 BaseProcessStart 是系统提供的进程启动代码，普通 Windows 进程的初始线程都是从此开始运行的。接下来是 VC 编译器插入的入口函数，而后是我们代码中的 main 函数。我们在 main 函数中调用了 MessageBox API，因为是 ANSI 程序（非 Unicode），所以链接的是 MessageBoxA（MessageBoxW 是用于 Unicode 程序）。MessageBoxA 内部的没有做任何处理，只是简单地调用另一个 API MessageBoxEx。

13.1.1　MessageBoxEx

MessageBoxEx API 的函数原型只比 MessageBox 多一个参数 wLanguageId，即：

```
int MessageBoxEx( HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption,
                  UINT uType, WORD wLanguageId);
```

其中，hWnd 用来指定父窗口句柄，如果没有父窗口，那么可以将其设置为 NULL。lpText 和 lpCaption 分别是消息框的消息文本和标题。uType 用来定制消息框的行为，我们稍后再讨论。wLanguageId 参数用来指定语言，目前这个参数保留未用，调用时只要指定为 0 即可。因此，MessageBox 和 MessageBoxEx 这两个 API 是等价的。

13.1.2　MessageBoxTimeout

接下来被调用的是 MessageBoxTimeout，它是一个广被使用但却未文档化的 API，其函数原型如下：

```
int MessageBoxTimeout(HWND hWnd, LPCSTR lpText,
   LPCSTR lpCaption, UINT uType, WORD wLanguageId, DWORD dwMilliseconds);
```

与 MessageBoxEx 相比，MessageBoxTimeout 也只是多了最后一个参数，即 dwMilliseconds。这是因为 MessageBoxTimeout 函数具有定时器（timer）功能，当定时器指定的时间到期（timeout）时，它会自动关闭对话框，dwMilliseconds 参数就是用来指定定时器的时间长度的，单位是毫秒（1/1000 秒）。尽管没有文档化，但是 USER32.DLL 输出了 MessageBoxTimeout 函数，因此可以用通过 GetProcAddress 函数获取这个函数的指针来动态调用它，MsgBox 小程序中包含了详细的源代码

（code\chap13\msgbox\msgbox.cpp）。

接下来，MessageBoxTimeoutA 在把所有字符串类型的参数都转为 UNICODE 类型后，调用 MessageBoxTimeoutW。这是自 Windows 2000 以来的典型做法，大多数包含字符串类型参数的 API 都有 UNICODE 和 ANSI 两个版本，一个版本在将参数转换为另一个版本的参数后便交由另一个版本统一处理。

MessageBoxTimeoutW 将所有参数放入一个与 MSDN 中文档化了的 MSGBOXPARAMS 结构非常类似的内部结构中，然后将该结构的指针作为参数调用 MessageBoxWorker 函数。另一个消息框 API MessageBoxIndirect 使用了 MSGBOXPARAMS 结构。

13.1.3 MessageBoxWorker

MessageBoxWorker 做了很多琐碎的参数检查和处理工作，比如，根据 dwStyle（对应于顶层的 uType 参数）和 dwLanguageId 参数确定按钮和加载按钮文字等。在准备好这些信息并将其放到刚才所说的内部结构中后，MessageBoxWorker 便完成任务了，它会把接下来的工作交给 SoftModalMessageBox 函数。

需要说明的是，如果 uType 参数中指定了 MB_SERVICE_NOTIFICATION 或 MB_DEFAULT_DESKTOP_ONLY 标志，那么 MessageBoxWorker 会调用 ServiceMessageBox 函数来处理。ServiceMessageBox 判断，如果对话框应该显示在其他 Station（工作站），那么便调用 WinStationSendMessage 将其转发到其他 Windows Station，如果对话框应该显示在当前 Station，那么便调用 NtRaiseHardError 服务将其发给 Windows 子系统进程（CSRSS）来处理。下一节将详细讨论 NtRaiseHardError 的工作原理。

首先，SoftModalMessageBox 做的工作更加具体，包括计算消息文字所需的长度，按钮和消息窗口的位置及大小等。然后 SoftModalMessageBox 将这些信息放到一个自己动态创建的窗口模板（template）中。大家知道我们在设计对话框时都需要在资源中创建对话框模板，其中包含了对话框的位置、布局、内容等信息。因为我们在调用 MessageBox 函数时没有提供资源模板，所以 SoftModalMessageBox 在这里动态创建了一个。在有了资源模板后，SoftModalMessageBox 将其传递给 InternalDialogBox，让其产生对话框。

13.1.4 InternalDialogBox

接下来的过程就与使用 DialogBox API 产生的模态对话框的过程基本一致了。InternalDialogBox 首先检查参数中的拥有者窗口（owner）句柄是否为空，如果不为空，那么便调用 NtUserEnableWindow 将其禁止，这是为什么在一个程序弹出模态对话框（消息对话框总是模态的）后，父窗口就不响应了的原因。接下来，Internal- DialogBox 调用 InternalCreateDialog 创建对话框窗口，如果创建失败，则恢复父窗口后返回。如果成功，它会调用 NtUserShowWindow 显示消息框，然后调用 DialogBox2。

正如大家所估计的，DialogBox2 内部包含了一个类似清单 13-2 所示的消息循环。该循环反复调用 PeekMessage 来从本线程的消息队列中获取消息，然后处理。这样，消息框窗口便可以与用户交互了。

图 13-1 左侧是 MsgBox 程序在 Windows XP 英文版上运行时弹出的消息框，右侧是在 Windows 2000 中文版上运行时弹出的消息框。显而易见，二者除了窗口风格略有差异外，OK 按钮的文字是与操作系统的语言相一致的，这是因为 MessageBoxWorker 函数是根据

语言 ID 来加载合适的字符串资源的。

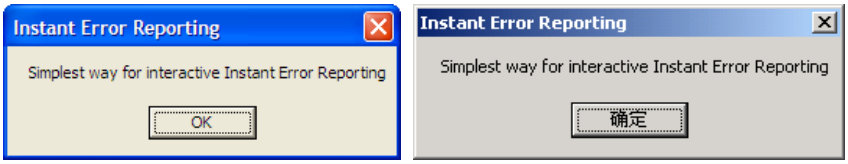


图 13-1　MessageBox API 弹出的消息框

13.1.5　消息框选项（uType）

在对 MessageBox 的工作原理有了较深入的了解后，下面我们回过头来仔细考察它的 uType 参数。uType 参数的类型是无符号的整数（32 位），用来指定控制 MessageBox 外观和行为的各种选项。为了便于使用，每一种选项被定义为一个以 MB_开始的宏（定义在 winuser.h 文件中），如 MB_YESNO 表示消息框中应该包含 YES 和 NO 按钮。迄今为止，已经定义了将近 40 个这样的宏，按照类型被分为 6 个组（group），表 13-1 按类别列出了这些宏的定义和用途。

表 13-1　MessageBox API 的选项标志

宏定义/类	值/类掩码	含义
按钮类（MB_TYPEMASK）	0x0000000FL	用来定义包含的按钮
MB_OK	0x00000000L	显示 OK 按钮
MB_OKCANCEL	0x00000001L	显示 OK 和 Cancel 按钮
MB_ABORTRETRYIGNORE	0x00000002L	显示 Abort、Retry 和 Ignore 按钮
MB_YESNOCANCEL	0x00000003L	显示 Yes、No 和 Cancel 按钮
MB_YESNO	0x00000004L	显示 Yes 和 No 按钮
MB_RETRYCANCEL	0x00000005L	显示 Retry 和 Cancel 按钮
MB_CANCELTRYCONTINUE	0x00000006L	显示 Cancel、Try Again 和 Continue 按钮
图标类（MB_ICONMASK）	0x000000F0L	用来定义包含的图标
MB_ICONHAND	0x00000010L	带有停止符号的图标
MB_ICONQUESTION	0x00000020L	带有问号的图标
MB_ICONEXCLAMATION	0x00000030L	带有惊叹号的图标
MB_ICONASTERISK	0x00000040L	带有字母 i 的图标
MB_USERICON（仅用于 MessageBoxIndirect）	0x00000080L	通过 MSGBOXPARAMS 结构的 lpszIcon 指定图标资源
MB_ICONWARNING	0x00000030L	带有惊叹号的图标
MB_ICONERROR	0x00000010L	带有停止符号（X）的图标
宏定义/类	值/类掩码	含义
MB_ICONINFORMATION	0x00000040L	带有字母 i 的图标
MB_ICONSTOP	0x00000010L	带有停止符号（X）的图标
默认按钮（MB_DEFMASK）	0x00000F00L	定义默认（含初始焦点的按钮）按钮
MB_DEFBUTTON1	0x00000000L	第一个按钮为默认按钮
MB_DEFBUTTON2	0x00000100L	第二个按钮为默认按钮
MB_DEFBUTTON3	0x00000200L	第三个按钮为默认按钮
MB_DEFBUTTON4	0x00000300L	第四个按钮为默认按钮
模态（modality）（MB_MODEMASK）	0x00003000L	指定窗口的模态性
MB_APPLMODAL	0x00000000L	应用程序级模态（见下文）
MB_SYSTEMMODAL	0x00001000L	系统级模态（见下文）
MB_TASKMODAL	0x00002000L	任务级模态（见下文）
杂项（MB_MISCMASK）	0x0000C000L	

MB_HELP	0x00004000L	包含 Help 按钮，当用户按此按钮时，向 hWnd 窗口发送 WM_HELP 消息
MB_NOFOCUS	0x00008000L	内部使用，参见微软知识库 87341 号文章
其他	N/A	
MB_SETFOREGROUND	0x00010000L	对消息框窗口调用 SetForegroundWindow
MB_DEFAULT_DESKTOP_ONLY	0x00020000L	仅在默认桌面显示消息框
MB_TOPMOST	0x00040000L	消息框具有 WS_EX_TOPMOST 属性
MB_RIGHT	0x00080000L	文字右对齐
MB_RTLREADING	0x00100000L	按从右到左的顺序显示标题和消息文字
MB_SERVICE_NOTIFICATION	0x00200000L	供系统服务（system service）程序使用
MB_SERVICE_NOTIFICATION	0x00040000L	用于 NT 4 之前的 Windows 版本
MB_SERVICE_NOTIFICATION_NT3X	0x00040000L	用于 NT 3.51

其中，模态性用来定义消息框弹出后，消息框窗口对用户输入的垄断性，如果模态性为 MB_APPLMODAL，那么，hWnd 参数所指定的父窗口将被禁止，直到消息框关闭后才恢复响应。如果模态性为 MB_SYSTEMMODAL，那么，除了具有 MB_APPLMODAL 的特征外，消息框窗口还会被授予 WS_EX_TOPMOST 属性，也就是成为最上层窗口，这样，如果它不被关闭，就总显示在最顶层，目的是让用户始终看到，但这时用户可以与 hWnd 参数外的其他窗口交互。如果指定 MB_TASKMODAL，即使 hWnd 参数为 NULL，则当前线程的所有顶层窗口也将被禁止，这是为了在不知道顶层窗口句柄时也将其禁止。

13.1.6 返回值

MessageBox API 的返回值是一个整数，如表 13-2 所示。

表 13-2 MessageBox 函数的返回值

宏定义	值	宏定义	值
IDOK	1	IDNO	7
IDCANCEL	2	IDCLOSE	8
IDABORT	3	IDHELP	9
IDRETRY	4	IDTRYAGAIN	10
IDIGNORE	5	IDCONTINUE	11
IDYES	6	IDTIMEOUT	32000

通过以上返回值，可以知道用户对消息框的响应结果，例如 IDYES 代表用户选择了 Yes 按钮，等等。

13.1.7 归纳

前面我们介绍了 MessageBox 函数的工作原理和几个相关的 API 及内部函数。图 13-2 归纳出了这些函数的相互关系。

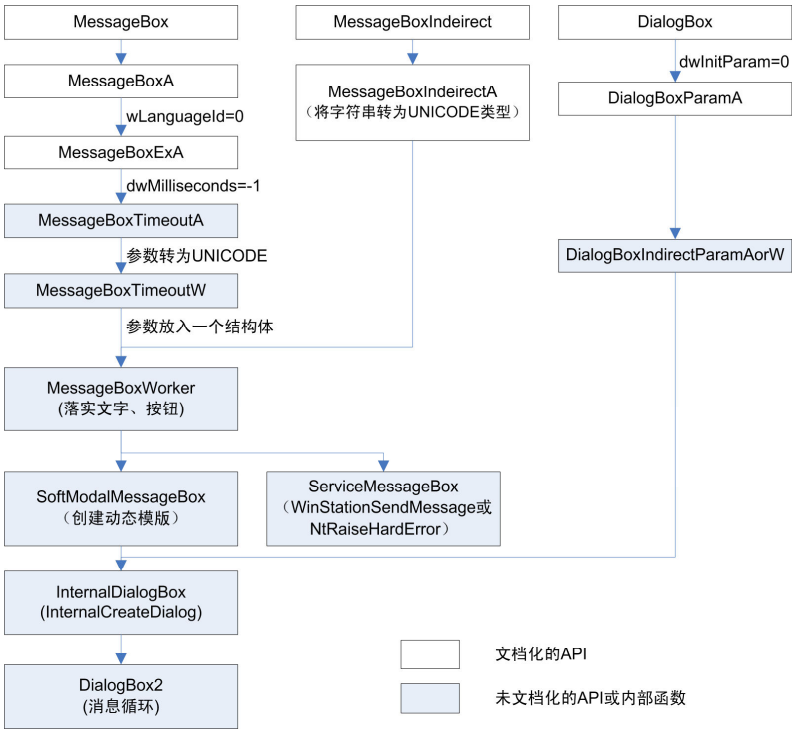


图 13-2　MessageBox 和有关 API 的调用关系

使用 MessageBox API 来实现错误提示的优点是简单易用，但是这种方法存在如下局限。首先，MessageBox 是一个用户态的 API，内核代码无法直接使用。第二，MessageBox 是工作在调用者（通常是错误发生地）的进程和线程上下文中的，如果当前进程/线程的数据结构（比如消息队列）已经由于严重错误而遭到损坏，那么 MessageBox 可能无法工作。第三，对于系统启动关闭等特殊情况，MessageBox 是无法工作的。

补编内容 2 堆检查之实例分析

补编说明：

这一节本来属于《软件调试》第 23 章的第 16 节，也就是最后一节，旨在通过一个真实案例来总结和巩固这一章前面各节的内容。案例涉及动态链接库模块和 EXE 模块，是在同一进程中有多个 C 运行库实例的情况。

在《软件调试》正式出版前压缩篇幅时，这一节被删除了。主要原因是虽然这个例子挺好的，但是毕竟是例子，为了确保原理性的内容，还是把这个例子删了。

23.16 实例分析

前面几节我们介绍了 Win32 堆、CRT 堆，以及它们的调试支持。本节我们通过一个实例来巩固大家的理解。在现实的软件产品中，一个应用程序通常由很多个模块组成，这些模块可能是不同团队或不同公司和组织开发的。这就很可能有多个模块都使用了 CRT 库，但是使用的版本和链接方式是不同的。对于这种一个进程内的多个模块（EXE 和 DLL）都使用了 CRT 堆的情况，它们使用的是多个 CRT 堆还是一个 CRT 堆呢？这个问题的答案主要和链接 CRT 的方式有关。简单地说，如果一个模块（EXE 或 DLL）静态链接 CRT，那么这个模块便会创建和使用自己的 CRT 堆。如果一个模块动态链接 CRT，那么这个模块便与同一进程内动态链接这个 CRT DLL 的所有模块共享一个 CRT 堆。下面以分别使用 VC6 和 VC2005（即 VC8）创建的 Win32 DLL 和 Win32 应用程序为例进行分析。

23.16.1 FaultDll 和 FaultApp

FaultDll 是使用 VC6 创建的标准 Win32 DLL，操作步骤为 File>New>选择 Win32 Dynamic-Link Library 并命名为 FaultDll>选择 A Dll that exports some symbols。FaultApp 是使用 VC6 创建的标准 Win32 应用程序，操作步骤为 File>New>选择 Win32 Application 并命名>选择 A typical “Hello World” application。以上两个项目除了将输出目录设置到同一个目录外，其他选项都是默认的。

在 FaultDll 中我们实现并输出一个简单的类 `CFaultClass`，它有一个公开的成员，类型为 STL（Standard Template Libarary，即标准模板库）中的 `std::string` 类。在 FaultApp 中我们加入一个名为 FaultCase 的函数来使用 `CFaultClass` 类，清单 23-35 给出了相关的

代码。

清单 23-35　CFaultClass 类的定义和使用

```
#include <string>
using namespace std;                    //使用 STL 的命名空间
class FAULTDLL_API CFaultClass          //定义并输出一个 C++类
{
public:
    CFaultClass(void);                  //构造函数，内部没有做任何操作
    string m_cstrMember;                //定义一个字符串成员
};
//以下是 FaultApp.cpp 中的代码
#include "../faultdll.h"                //包含声明 CFaultClass 类的头文件
void FaultCase(HWND hWnd)               //使用 CFaultClass 类的 FaultCase 函数
{
    CFaultClass fc;                     //定义一个实例
    fc.m_cstrMember="AdvDbg.org";       //直接对公开的成员赋值
    MessageBox(hWnd, fc.m_cstrMember.c_str(), "FaultApp",MB_OK);
                                        //显示成员的当前值
}                                       //返回
```

最后在 FaultApp 程序中加入一个菜单项 IDM_FAULT，并在响应这个菜单项命令时调用 FaultCase 函数。

以上代码选摘自一个实际的软件项目，看起来非常简单，似乎也没什么问题，但是实际上它却隐藏着严重的问题，我们先来看调试版本的运行情况。

23.16.2　运行调试版本

编译以上两个项目，会有一个 c4251 警告，我们稍后再讨论它。直接执行（非调试）调试版本的 FaultApp 程序（位于 code\bin\debug 目录），点击 File 菜单的 Triger Fault 项后，会得到图 23-10 所示的断言失败对话框。

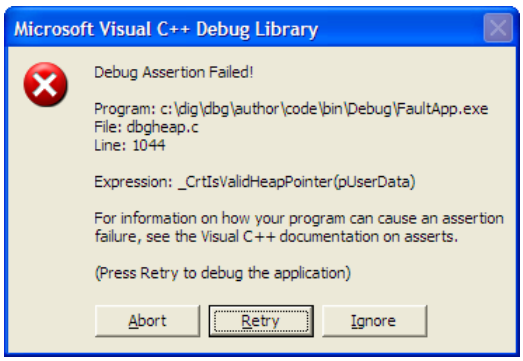


图 23-10　内存检查断言失败对话框

根据对话框中提示的文件名和行号，打开 CRT 堆的源程序文件 dbgheap.c（典型路径为 c:\Program Files\Microsoft Visual Studio\VC98\crt\src），找到 1044 行，可以看到该位置果然有图 23-10 中所描述的断言_CrtIsValidHeapPointer(pUserData)，断言上面有一段注释：

```
/*
 * If this ASSERT fails, a bad pointer has been passed in. It may be
 * totally bogus, or it may have been allocated from another heap.
 * The pointer MUST come from the 'local' heap.
 */
_ASSERTE(_CrtIsValidHeapPointer(pUserData));
```

以上代码所属的函数名处于一个条件编译块，如果定义了用于支持多线程_MT 标志，那么函数名是_free_dbg_lk 函数，会被一个带有锁定支持的_free_dbg 函数所调用，否

则这段代码便会被编译为_free_dbg 函数。因为我们的程序中定义了_MT 标志，所以断言发生在_free_dbg_lk 函数中。注释的意思是如果这个断言失败，那么有人向这个函数传递进了错误的指针，这个指针可能是完全捏造的，也可能是从另一个堆上分配的。要释放的指针一定要来源于本地的堆。看了这个说明后，可以推测出断言失败是因为_free_dbg_lk 函数认为传递给它的 pUserData 参数有问题。这个参数用来指定要释放的堆块。那么是要释放哪个堆块时导致这个断言失败呢？

23.16.3 分析原因

将 WinDBG 设置为 JIT 调试器(执行 WinDBG -I)，然后选择 Retry 按钮进行 JIT 调试。在 WinDBG 与 FaultApp 成功建立调试会话后，从 WinDBG 显示的信息中可以看出报告断言失败的断点指令确实位于_free_dbg_lk 函数中，键入 k 命令观察栈回溯信息（清单 23-36）。

清单 23-36 析构 string 成员（释放内存）的执行过程（摘要）

```
0:000> kpnL //L 代表不显示源文件名
 # ChildEBP RetAddr
00 0012fa84 100045ca FaultDll!_free_dbg_lk+0xc8  //lk 代表具有锁定（Lock）保护
01 0012fa94 1000457e FaultDll!_free_dbg+0x1a     //调试版本的堆块释放函数
02 0012faa4 1000246c FaultDll!free+0xe           //C 的内存释放函数
03 0012fab0 10001dd6 FaultDll!operator delete+0xc  //delete 运算符
04 0012fb0c 100018f2 FaultDll!std::allocator<char>::deallocate+0x26
05 0012fb70 10001497 FaultDll!std::basic_string<char,… >::_Tidy+0x82
06 0012fbcc 10001245 FaultDll!std::basic_string<… >::~basic_string<… >+0x27
07 0012fc24 004014b8 FaultDll!CFaultClass::~CFaultClass+0x25 //析构函数
08 0012fc94 00401628 FaultApp!FaultCase+0x88     //应用程序中使用输出类的函数
09 0012fdb4 7e418724 FaultApp!WndProc+0x118      //窗口过程，以下栈帧省略
```

从上面的栈回溯信息可以看到，问题与 FaultApp 的 FaultCase 函数有关，在该函数释放局部对象 fc 时，引发调用 CFaultClass 的析构函数，后者调用 string 类的析构函数~basic_string 来释放成员 m_cstrMember。~basic_string 函数依次调用_Tidy 方法和内存分配器（allocator）的 deallocate 方法，接下来 Deallocate 方法调用 delete 运算符来释放 string 的缓冲区，从而调用 CRT 堆的堆释放函数 free_dbg。因为 FaultDll 的 CRT 链接选项中包含/MT，即使用多线程支持，所以 free_dbg 对堆锁定后调用 free_dbg_lk 函数执行释放操作。

通过上面的分析我们知道，是 CFaultClass 类的析构函数在析构 m_cstrMember 成员时引发了错误。CFaultClass 类和 FaultCase 函数都很简单，是哪里出现错误了呢？为了搞清楚这个问题，我们结束 JIT 调试会话。然后在 WinDBG 中打开（Open Executable）FaultApp.exe 开始一个新的调试会话。我们的目标是跟踪 FaultCase 函数向 m_cstrMember 成员赋值的过程，观察 m_cstrMember 为其成员分配内存的细节。对 FaultCase 函数设置一个断点（bp FaultCase），恢复程序执行后选择菜单中的 Triger Fault 让该断点命中。为了避免枯燥的单步跟踪，对 RtlAllocateHeap 函数设置一个断点(bp ntdll!RtlAllocateHeap)，因为前面我们讨论过大多数情况 CRT 堆使用的都是系统模式，会调用 Win32 堆的分配函数分配堆块。设好断点，让程序执行，断点果然命中，键入 k 命令观察栈回溯信息（清单 23-37）。

清单 23-37 为 string 成员分配内存的执行过程（摘要）

```
0:000> k
ChildEBP RetAddr
0012f8d8 00408f42 ntdll!RtlAllocateHeap    //Win32 堆的分配函数
0012f8f0 00404752 FaultApp!_heap_alloc_base+0xc2 [malloc.c @ 200]
0012f918 00404559 FaultApp!_heap_alloc_dbg+0x1a2 [dbgheap.c @ 378]
```

```
0012f964 004020fe FaultApp!operator new+0xf [new.cpp @ 24]
0012f9bc 00402048 FaultApp!std::_Allocate+0x2e […\xmemory @ 30]
0012fa1c 00401ed9 FaultApp!std::allocator<char>::allocate+0x28 [...\xmemory]
0012fb04 0040194b FaultApp!std::basic_string<… >::_Grow+0x120 [… @ 568]
0012fbc4 00401849 FaultApp!std::basic_string<…>::assign+0x36 [… @ 138]
0012fc20 00401473 FaultApp!std::basic_string<… >::operator=+0x29 [… @ 67]
0012fc94 004015ce FaultApp!FaultCase+0x53 [C:\... \FaultApp.cpp @ 126]
```

上面的清单显示了向一个 string 对象（m_cstrMember）赋值的完整过程。因为要存储的字符串长度超过了 string 类现有缓冲区的容量，所以 assign 方法调用 Grow 方法增大缓冲区，从而引发了调用 std::_Allocate 和 new 运算符分配内存。

比较清单 23-35 和清单 23-36 中的释放和分配 string 对象缓冲区的过程，我们可以明显地看到，分配过程使用的是静态链接到 FaultApp 模块中的 CRT 函数（注意每个函数名前的模块名），而释放过程使用的是静态链接到 FaultDll 模块中的 CRT 函数。尽管两个模块静态链接的 `string` 类和 CRT 函数的代码应该是相同的（因为我们使用同一个 VC 环境开发），但是我们知道 CRT 库不仅有代码，还有全局变量，比如 CRT 堆的句柄就是记录在名为 `_crtheap` 的全局变量中的。这样一来，如果这两套函数使用了各自的全局变量，那么它们使用的也是各自的堆。这势必造成分配时使用一个堆，释放时使用的是另一个堆，因而导致了上面的问题。使用 WinDBG 观察 FaultDll 和 FaultApp 中的 `_crtheap` 变量，可以看到它们指向的确实是不同的堆：

```
0:000> dd FaultDll!_crtheap l1
10039838  003c0000
0:000> dd FaultApp!_crtheap l1
0042e938  003d0000
```

这个例子告诉我们静态链接到每个模块的中 CRT 是相对独立的，它们各自维护自己的全局变量，创建和使用自己的 CRT 堆。在使用 CRT 堆时应该确保在一个 CRT 堆分配的内存也要在这个 CRT 堆上进行释放。

23.16.4 发布版本

刚才我们分析的是调试版本的情况，调试版的 CRT 堆释放函数在释放前的检查中会发现问题并以断言的形式报告出来，也就是错误的释放动作没有真正执行。那么发布版本的情况如何呢？

为了提高执行速度，发布版本的 CRT 堆函数中不再包含调试版本中的很多检查工作，因此前面的断言不再存在。

事实上，运行发布版本的 FaultApp 时，其结果是不确定的。如果运行完全使用默认选项编译出的发布版本，那么执行没有任何问题。使用 WinDBG 跟踪 FaultCase 函数，可以发现编译器的优化功能将 string 类的赋值运算符做了 inline，将该运算符的函数代码直接插入到了 `FaultCase` 函数中，类似的 `CFaultClass` 的析构函数也被 inline 到 FaultCase 函数中。这便使得内存分配和删除操作都完全是在 FaultCase 函数中发起的。其主要汇编指令如下：

```
call    FaultApp!std::basic_string<…>::_Grow (00401450)    // 分配
rep movs dword ptr es:[edi],dword ptr [esi]                // 赋值
call    FaultApp!operator delete (00401750)                // 释放
```

也就是说，因为编译器对发布版本的优化措施将本来发生在 FaultDll 中的释放操作（清单 23-35 的栈帧#03）移入到了 FaultApp 中。这样分配和释放便都发生在 FaultApp 中，它们使用的都是一个堆，这样确实没有问题了。

下面我们试一下禁止 inline 的情况。在 FaultApp 项目的发布版本属性中将 inline 功能

禁止（Project > Settings > C++ > Optimizations > Inline function expansion > Disable），然后编译并执行 FaultApp 的发布版本（bin\release\faultapp.exe），选择 Triger Fault 菜单执行 FaultCase 函数，执行一两次时并没有什么异常情况发生，但是当执行第 7 次（有随机性）时，应用程序错误对话框弹出来了，错误的详细信息中显示应用程序执行了非法访问，错误代码是 0xC000005。点击调试按钮启动 WinDBG 开始 JIT 调试，WinDBG 显示如下异常现场信息：

```
(1364.848): Access violation - code c0000005 (!!! second chance !!!)
eax=003d3040 ebx=003c0000 ecx=00000000 edx=00000000 esi=003d3038 edi=003d2378...
ntdll!RtlpCoalesceFreeBlocks+0x36e:
7c910f29 8b09          mov    ecx,dword ptr [ecx]  ds:0023:00000000=????????
```

可见，是因为 `RtlpCoalesceFreeBlocks` 函数访问了空指针，试图读取 ECX 指针的内容，但 ECX 的值是 0。使用 kbn 命令显示栈回溯信息（清单 23-38）。

清单 23-38　发布版本中的析构过程

```
0:000> kbn
 # ChildEBP RetAddr  Args to Child
00 0012fc24 7c910d5c 003d08c0 00000000 0012fcdc ntdll!RtlpCoalesceFreeBlocks+0x36e
01 0012fcf8 10002b9b 003c0000 00000000 003d23b8 ntdll!RtlFreeHeap+0x2e9
02 0012fd0c 10001ba2 003d23b8 10001284 003d23b8 FaultDll!free+0x46
03 0012fd14 10001284 003d23b8 00401230 00401216 FaultDll!operator delete+0x9
04 0012fd20 00401216 00401200 003d23b9 0000000a FaultDll!CFaultClass::~CFau…
05 0012fd40 00401304 00990338 00000000 00000000 FaultApp!FaultCase+0x66
06 0012fdfc 7e418724 00990338 00000111 00008003 FaultApp!WndProc+0xd4
```

从栈帧#04 可以了解到异常仍是与 `CFaultClass` 类的析构函数有关。栈帧#3 是在执行 delete 运算符，栈帧#1 是调用 Win32 堆的释放函数，栈帧#0 的 `RtlpCoalesceFreeBlocks` 函数是 Win32 堆中用来合并空闲块的工作函数。仔细观察栈帧#1 中传递给 `RtlFreeHeap` 函数的参数，其中 **003c0000** 是堆句柄，**003d23b8** 是要释放堆块的用户指针。使用!heap 命令列出进程中的所有堆：

```
0:000> !heap
Index   Address  Name      Debugging options enabled …
  6:   003c0000
  7:   003d0000
```

根据我们多次分析 Win32 堆的经验，用户指针 **003d23b8** 显然更像是 7 号堆上的堆块，但现在却是试图从 6 号堆上释放。为了提高运行速度，默认情况下堆的工作函数是假定传递给它的用户指针都是正确的，会根据这个指针的地址计算堆块的起始位置，然后修改堆块的属性，更新空闲链表……。因此当把从一个堆上分配的用户指针和另一个堆的句柄传递给释放函数时，混乱局面便开始了，用户数据和管理数据被张冠李戴，链表指针被错误的指来指去，导致整个堆混乱，即所谓的堆败坏（Heap Corruption）。而且更可怕的是，这样的问题并没有立刻暴露出来，FaultCase 函数运行了 7 次才有异常发生，这种延迟性显然提高了定位错误根源的难度。

23.16.5　回放混乱过程

尽管刚才的试验中，执行多次 FaultCase 函数后问题才显现出来，但事实上第一次执行时就已经导致了问题。为了加深大家的印象，下面我们将跟踪第一次执行错误释放动作时的内部过程。先执行 gflags /i faultapp.exe +0 防止在调试器中运行时系统自动开启 Win32 的调试支持。然后重新启动 WinDBG 调试打开发布版本的 FaultApp.exe（code\bin\realease\），在 FaultDll!free 处设置一个断点。选择 Triger Fault 菜单项触发应用程序执行 FaultCase 函数，点击消息框的 OK 按钮后断点命中，执行 kv 命令观察栈回溯，注意关于 free 函数的那一行：

```
0012fd0c 10001ba2 003d07e0 10001284 003d07e0 FaultDll!free (FPO: [1,0,1])
```

其中 003d07e0 是 free 函数的参数，也就是要释放的用户指针。将这个地址减去 8 便是 _HEAP_ENTRY 结构，因此可以使用 dt _HEAP_ENTRY 003d07e0-8 来显示要释放堆块的信息，注意它的 Size 信息：

```
   +0x000 Size           : 7
```

即这个堆块占用的空间为 7*8=56 个字节，那么下一个堆块的地址应该为 003d07e0-8+0x38=003d0810。使用!heap 0x003d0000 –a 列出 0x003d0000 堆中的所有堆块，可以看到一致的信息：

```
003d07d8: 00088 . 00038 [01] - busy (30)
003d0810: 00038 . 00af8 [00]
```

在 003d1d78 处设置一个数据访问断点，ba w1 003d07d8，然后输入 g 命令让程序继续执行，断点旋即命中，使用 kbn 命令观察栈回溯：

```
00 0012fc20 7c9105c8 003d07d8 0012fcf8 7c910551
ntdll!RtlpInterlockedPushEntrySList+0xe
01 0012fc2c 7c910551 003c07d8 003d07e0 0012fe64 ntdll!RtlpFreeToHeapLookaside+0x22
02 0012fcf8 10002b9b 003c0000 00000000 003d07e0 ntdll!RtlFreeHeap+0x1e9
03 0012fd0c 10001ba2 003d07e0 10001284 003d07e0 FaultDll!free+0x46
```

可将 RtlpFreeToHeapLookaside 函数在调用 RtlpInterlockedPushEntrySList 将堆块 **003d07d8** 加入到堆的空闲块旁视列表中。但是遗憾的是，FaultDll!free 函数调用 RtlFreeHeap 指定了错误的堆，现在是从 0x3c0000 堆上释放属于 0x3d0000 的堆块，所以 RtlpFreeToHeapLookaside 函数的第一个参数中所指定的旁视列表是属于 0x3c0000 堆的。也就是说，0x3d0000 堆上的堆块释放时被记录到了 0x3c0000 堆的空闲堆块旁视列表中。

尽管隐患已经埋下，但是因为所有调试机制目前都被禁止了，所以错误症状还没体现出来，RtlFreeHeap 函数返回的结果是 1，成功。再执行一编 TrigerFault，我们可以看到 0x3d0000 堆上又多了一个堆块（003d1da8）。

```
003d07d8: 00088 . 00038 [01] - busy (30)   //第一次执行 TrigerFault 时分配的堆块
003d0810: 00038 . 00038 [01] - busy (30)   //第二次执行 TrigerFault 时分配的堆块
003d0848: 00038 . 00ac0 [00]               //下一个空闲块
```

可见，堆管理器把刚才与 003d07d8 相邻的空闲块一分为二。事实上如果不出问题，因为第二次执行时请求的块大小与刚才释放的相同，那么堆管理器从旁视列表（“前端堆”）中就可以找到正好满足要求的空闲块，不须要再分配一个。

类似地每执行 FaultCase 函数一次，0x3d0000 堆上会增加一个占用堆块，但是又执行几次后访问异常发生了，栈回溯与 JIT 调试看到的一样。

如果是 FaultDll 与 FaultApp 中的代码交替使用堆，比如每执行一次 FaultCase 函数后都选择 File 菜单的 Not My Fault 项执行一次 FaultDll 中的 fnFaultDll 函数，那么因为 fnFaultDll 函数要分配的空间是 48 个字节，刚好等于 FaultCase 函数中释放的大小，所以 FaultDll 的堆会尝试使用旁视列表中的这个空闲块来满足 fnFaultDll 函数。但因为旁视列表中实际记录的是另一个堆上的堆块，所以我们可能看到 fnFaultDll 函数得到的用户指针很奇怪，使用这个指针所指向的空间可能破坏进程中的其他数据，于是局面更加混乱。

23.16.6 思考

如果在 VC2005 中创建类似的 DLL（FtDllVC8）和应用程序（FtAppVC8），并将相应的代码加入进去，然后运行调试版和发布版本都没有问题。但略加分析，就可以知道这是因为 VC8 默认的链接选项是动态链接 CRT。这样 DLL 和应用程序便共享一个 CRT 堆了，

因此也就没有上面的因为 CRT 堆不同的而导致的问题。事实上，在 VC6 中也可以通过将两个项目的链接选项都改为动态链接解决这个问题。但是这种方法是不值得推荐的，因为问题的核心是违背了哪里分配内存哪里释放的基本原则。一种优雅的解决方法是将 CFaultClass 的 m_cstrMember 成员从公开改为 private 或 protected，然后公开适当的方法来设置和读取这个成员，这样既可以保证对 m_cstrMember 成员的内存分配和释放都发生在这个类所在的模块中，又遵守了面向对象的原则。

最后介绍一下 VC6 编译 CFaultClass 类的 string 成员时给出的 c4251 警告。该警告的完整信息如下：

```
'm_cstrMember' : class 'std::basic_string<…>' needs to have dll-interface to be used by clients of class 'CFaultClass'
```

其意思是成员 m_cstrMember 所属的 basic_string 类须要有 DLL 接口才能被 CFaultClass 类（客户）使用。换句话来说，编译器发现了 CFaultClass 类是以 DLL 形式输出的（类定义中包含__declspec(dllexport)），但是它的公开成员 m_cstrMember 的类 basic_string 没有 DLL 方式的接口，或者说 basic_string 类的声明中没有__declspec(dllexport)，因为我们选择是静态链接 CRT。回顾上面的错误，直接原因就是 FaultApp 中的 basic_string 类和 FaultDll 中的 basic_string 类交替操作一个实例 fc.m_cstrMember。也就是说 basic_string 类的两份代码拷贝交替操作一个类实例。如果 FaultDll 是动态链接 basic_string 类的，或者说 basic_string 类也是以 DLL 形式输出的，那么也不会有本节讨论的问题，这也再次告诉我们要重视编译器的警告信息。

补编内容 4 异常编译

补编说明：

这一部分内容本来属于《软件调试》第 24 章的后半部分，讲的是编译器编译异常处理代码的一些内部细节，包括局部展开、全局展开和对象展开等。写作这些内容的目的是让读者对异常编译有一个既全面又深入的理解。

在请朋友预览第 24 章时，朋友在肯定这一内容的深度的同时，质疑了这一内容的必要性，“又不是一本写编译器的书，没有必要写那么多编译的细节!”

于是在最后一轮压缩篇幅时，这部分内容就被砍掉了。自己当时很觉得可惜，写作这部分内容还是颇花了些时间和心思的。

24.6 栈展开

栈展开是异常处理中比较难理解的一个部分。本节将通过一个实例来介绍栈展开的原因和方法。

24.6.1 SehUnwind

为了说明栈展开的必要性和工作过程，我们编写了一个名为 SehUnwind 的小程序，其主要代码如清单 24-15 所示。

清单 24-15 SehUnwind 程序的主要代码

```
1   EXCEPTION_DISPOSITION
2   __cdecl _uraw_seh_handler( struct _EXCEPTION_RECORD *ExceptionRecord,
3      void * EstablisherFrame, struct _CONTEXT *ContextRecord,
4      void * DispatcherContext )
5   {
6      printf("_raw_seh_handler: code-0x%x, flags-0x%x\n",
7         ExceptionRecord->ExceptionCode,
8         ExceptionRecord->ExceptionFlags);
9
10     return ExceptionContinueSearch;
11  }
12  int FuncFoo(int n)
13  {
14     __asm
15     {
16        push    OFFSET _uraw_seh_handler
17        push    FS:[0]
18        mov     FS:[0],ESP
19
20        xor     edx, edx
21        mov     eax, 100
```

```
22          xor     ecx, ecx   // Zero out ECX
23          idiv    ecx        // Divide EDX:EAX by ECX
24      }
25      printf("Never been here if %x==0\n",n);
26      __asm
27      {
28          mov     eax,[ESP]
29          mov     FS:[0], EAX
30          add     esp, 8
31      }
32      return n;
33  }
34  int main(int argc, char* argv[])
35  {
36      int n=argc;
37      __try
38      {
39          printf("FuncSeh got %x!\n", FuncFoo(n-1));
40      }
41      __except(EXCEPTION_EXECUTE_HANDLER)
42      {
43          n=0x111;
44      }
45
46      printf("Exiting with n=%x\n",n);
47      return n;
48  }
```

在上面的代码中，FuncFoo 函数使用手工方法注册了一个异常处理器，处理函数为_uraw_seh_handler。main 函数在调用 FuncFoo 函数时通过__try{}__except()结构也使用了结构化异常处理，像这种不同层次的函数都使用异常处理的情况在实际应用中是很普遍的。

如果不带任何参数执行 SehUnwind 程序，那么 argc=1，这会使 FuncFoo 中的整除（29 行）操作导致一个除零异常。因为 FuncFoo 自己注册了异常处理器，所以系统会调用它的异常处理函数_uraw_seh_handler。不过_uraw_seh_handler 总是返回 ExceptionContinueSearch，并不处理任何异常（试验目的），这导致系统继续寻找其他的异常处理器，于是会找到 main 函数登记的异常处理器，这个处理器的过滤表达式为常量 EXCEPTION_EXECUTE_HANDLER，也就是处理任何异常。这样，系统便找到了父函数中登记的异常处理器，接下来应该执行这个处理器的异常处理块（第 50 行）。这意味着执行路线将由 FuncFoo 函数的中部（第 23 行）跳转到 main 函数中（第 43 行）。也就是说，由于发生异常 FuncFoo 将由其中部退出，这个出口是编译器在编译期所预料不到的，我们称这样的出口为函数的异常出口（Exception Exit）。异常出口直接导致了如下两个问题。

第一，对于意外退出的函数，由于函数从异常出口退出，异常出口后的代码便不会执行，那么本来放在本来出口路线上的清理代码也得不到执行了。对于 FuncFoo 函数，注销异常处理器（第 26~31 行）和恢复栈的代码将被意外跳过。

第二，对于要处理异常的 main 函数，由于是从 FuncFoo 中突然跳回（而不是正常返回）到 main 函数中执行的，为了保证 main 函数的代码顺利执行，应该将栈帧和寄存器状态恢复成 main 函数所使用的栈帧和寄存器。

栈展开（Stack Unwind）的初衷就是为了解决以上问题，简单来说，就是要为执行异常处理块做好准备，同时也给意外退出的函数做清理工作的机会。对于我们现在讨论的例子，栈展开就是要将目前正在执行 FuncFoo 函数时的栈变成适合返回到 main 函数的第 43 行执行的栈。如果把每个子函数的栈帧看作是栈上的一个个弯曲，那么栈展开就是把这些弯曲拉直，使栈直接恢复到异常处理块所在位置的状态。

24.6.2 全局展开

了解了栈展开的含义和必要性之后，我们继续探索栈展开的过程。仍以 SehUnwind 程序为例，在它执行结束后，打印出的结果如下：

```
_raw_seh_handler: code-0xc0000094, flags-0x0
_raw_seh_handler: code-0xc0000027, flags-0x2
Exiting with n=111
```

其中，第 1 行是发生异常时，系统调用_uraw_seh_handler 函数而输出的，0xc0000094 是除零异常的代码。从第 2 行来看，_uraw_seh_handler 函数显然是又被调用了一次。事实上，这一次就是因为栈展开而调用的，0xc0000027 是专门为栈展开而定义的异常代码 STATUS_UNWIND，标志位 0x2 代表 EH_UNWINDING，即因为栈展开而调用异常处理函数。

如果对_raw_seh_handler 函数设置断点，在除零异常发生后，这个断点会命中两次，第一次命中时的栈调用序列与清单 24-6 一样，只有栈帧 0 的函数名从_raw_seh_handler 变为_uraw_seh_handler。清单 24-16 给出了第二次命中时的栈调用序列。

清单 24-16 全局展开的执行过程

```
# ChildEBP RetAddr
00 0012f738 7c9037bf SehUnwind!_uraw_seh_handler //FuncFoo 函数登记的异常处理函数
01 0012f75c 7c90378b ntdll!ExecuteHandler2+0x26      //在保护块中调用异常处理函数
02 0012fb24 004011dc ntdll!ExecuteHandler+0x24       //参见 24.4.2
03 0012fb4c 0040131b SehUnwind!_global_unwind2+0x18  //全局展开
04 0012fb70 7c9037bf SehUnwind!_except_handler3+0x5f //main 函数登记的异常处理函数
05 0012fb94 7c90378b ntdll!ExecuteHandler2+0x26      //在保护块中调用异常处理函数
06 0012fc44 7c90eafa ntdll!ExecuteHandler+0x24       //参见 24.4.2
07 0012fc44 004010d6 ntdll!KiUserExceptionDispatcher+0xe //用户态分发异常的起点
08 0012ff4c 00401139 SehUnwind!FuncFoo+0x26          //发生除零异常的函数
09 0012ff80 00401448 SehUnwind!main+0x39             //入口函数
0a 0012ffc0 7c816fd7 SehUnwind!mainCRTStartup+0xb4   //编译器插入的入口函数
0b 0012fff0 00000000 kernel32!BaseProcessStart+0x23  //进程的启动函数
```

其中，栈帧#08 对应的 FuncFoo 导致了除零异常，首先在内核态进行分发（参见第 11.3 节），然后回到用户态的 KiUserExceptionDispatcher 继续分发，也就是调用 RtlDispatchException 函数（未显示出来）。栈帧#06～栈帧#04 是在执行 FS:[0]链条中找到的异常处理器，即 main 函数登记的异常处理函数_except_handler3，这是当 RtlDispatchException 函数调用_uraw_seh_handler 函数时，得到 ExceptionContinueSearch 后继续遍历 FS:[0]链条而找到的。_except_handler3 在执行 main 函数中的过滤表达式时得到的结果是 EXCEPTION_EXECUTE_HANDLER，于是它准备执行对应的处理块。在转去执行处理块之前，它先调用_global_unwind2 开始栈展开（栈帧#03）。_global_unwind2 的实现很简单，只须调用另一个重要的 RTL 函数 RtlUnwind，上面的清单没有显示出 RtlUnwind 函数。

```
__global_unwind2(_EXCEPTION_REGISTRATION * pRegistrationFrame)
 {
    _RtlUnwind(pRegistrationFrame, &__ret_label, 0, 0 );
    __ret_label:
 }
```

RtlUnwind 函数的原型为：

```
VOID RtlUnwind (PEXCEPTION_REGISTRATION pTargetFrame, ULONG ulTargetIpAddress,
   PEXCEPTION_RECORD pExceptionRecord, ULONG ulReturnValue);
```

其中，pTargetFrame 指向 FS:[0]链条中同意处理异常的那个异常登记结构，用来指定栈展开的截止栈帧。参数 ulTargetIpAddress 用来指定展开操作结束后要跳转回来的地址，从上面__global_unwind2 函数的代码可以看到，它就是调用 RtlUnwind 语句下

面的一个标号的地址。参数 pExceptionRecord 用来指定异常记录结构，ulReturnValue 可以指定一个放入到 EAX 寄存器中的返回值。RtlUnwind 的实现较为复杂，以下是它执行的主要动作。

第一，定义一个新的异常记录 EXCEPTION_RECORD，将它的异常代码设置为 STATUS_UNWIND，并在 ExceptionFlags 字段中设置 EH_UNWINDING 标志。

第二，调用 RtlpCaptureContext 将当时的线程状态捕捉到一个上下文结构（Context）中，并调整 Esp 字段使其对应的栈不包含本函数的内容。

第三，从 FS:[0]链条的表头开始，依次取出每个异常处理器的登记结构，执行下面两步中的操作。

第四，将异常登记结构中的处理函数作为参数调用 RtlpExecutehandlerFor-Unwind 函数，即：

```
nDisposition = RtlpExecutehandlerForUnwind(
      pExceptionRecord, pExceptionRegitrationRecord,
      &context,  &DispatcherContext, pExceptionRegitrationRecord ->handler );
```

RtlpExecuteHandlerForUnwind 函数与 RtlpExecuteHandlerForException 非常类似，它只是先将处理内嵌异常的函数地址赋给 EDX 寄存器，然后便自然进入（Fall Through）到紧随其后的 ExecuteHandler 函数中。ExecuteHandler 内部会调用 ExecuteHandler2 函数，ExecuteHandler2 内部调用真正的异常处理函数，比如 _uraw_seh_handler。在异常处理函数返回后，标志着这个函数的展开工作结束。

第五，调用 RtlpUnlinkHandler(pExceptionRegitrationRecord)将刚才展开结束的异常处理器从 FS:[0]链条上移除。

第六，取出下一个异常登记结构并回到第 4 步，直到遇到参数 pTargetFrame 所指定的记录，标志着展开操作结束。

第七，调用 ZwContinue(Context, FALSE)恢复到 Context 结构中指定的状态，并继续执行，如果执行成功，程序就“飞”回到 ulTargetIpAddress 参数所指定的地址，即标号__ret_label 所代表的地址，也就是返回到_global_unwind2 函数中。

全局展开结束后，FS:[0]指向的便是声明愿意处理异常的登记结构了。接下来，_except_handler3 函数会将这个结构中记录的 EBP 值（即 _EXCEPTION_ REGISTRATION 结构的_ebp 字段）设置到 EBP 寄存器中。这样栈便恢复到了这个异常处理器所对应的栈帧。对于我们的例子，就是把栈恢复到 main 函数的栈帧，FS:[0]链条也是 main 函数调用 FuncFoo 之前的样子，此时的状态与 main 函数中发生异常需要执行异常处理块是一样的。而后，_except_handler3 函数会调用异常处理块，也就是执行 pRegistrationFrame->scopetable[trylevel].lpfnHandler()。异常处理块结束后便会自然进入（Fall Through）到所属的函数中，不会再返回到_except_ handler3 函数。对于我们的例子，在 main 函数的异常处理块（第 43 行）执行后，便自然地执行后面的第 46 行和第 47 行了。

24.6.3 局部展开（Local Unwind）

那么，在每个异常处理器函数收到展开调用后应该做些什么呢？总的来说，就是完成该处理函数范围内的清理和善后工作。因为 VC 为每个使用 SEH 的函数注册了一个异常处理器，这意味着一个处理器的管理范围是它所负责的那个函数。相对于全局展开需要遍历 FS:[0]链条依次调用多个异常处理器来讲，我们把每个异常处理器收到全局展开调用后

所作的本函数范围内的清理工作叫局部展开。因为 RtlUnwind 会从 FS:[0]链条注销被展开的登记结构，即上面描述的第 5 步，所以局部展开时不需要注销自己的登记结构。

概括来讲，局部展开主要是完成如下两项任务。

第一，遍历范围表（scopetable）中属于被展开范围的各个表项，察看是否有终结处理块（__try{}__finally 结构的 finally 块）需要执行。

第二，对于 C++程序，调用被展开范围内还存活对象调用函数内还存活对象（alive objects）的析构函数。C++标准规定当栈被展开时，异常发生时仍然存活着的对象的析构函数应该被调用。这一操作很多时候被简称为对象展开（Object Unwind）。

我们将在下一节讨论__try{}__finally 结构。现在来看对象展开，如清单 24-17 所示的 SehUwObj 程序。如果不带任何命令行参数执行 SehUwObj，那么 argc = 1，这会让 main 函数使用参数 n = 0 调用 FuncObjUnwind 函数，在第 23 行会发生一个除零异常。这时，第 20 行实例化的 bug0 对象依然有效（存活），而 bug1 对象还没创建。按照 C++标准，当栈展开时应该调用 CBug 的析构函数来析构 bug0 对象。

清单 24-17　演示对象展开的 SehUwObj 程序

```
1   #include "stdafx.h"
2   #include <windows.h>
3
4   class CBug
5   {
6   public:
7       CBug(int n) : m_nIndex(n){ printf("Bug %d constructed\n", m_nIndex);}
8       ~CBug(){ printf("Bug %d deconstructed\n", m_nIndex);};
9   protected:
10      int m_nIndex;
11  };
12  int FuncObjUnwind(int n)
13  {
14      CBug bug0(0);
15      __try
16      {
17          n=1/n;              // n=0 时会触发除零异常
18      }
19      __except(EXCEPTION_CONTINUE_SEARCH)
20      {
21          CBug bug1(1);
22          n=0x122;
23      }
24      return n;
25  }
26
27  int main(int argc, char* argv[])
28  {
29      __try
30      {
31          printf("FuncObjUnwind got %x!\n", FuncObjUnwind(argc-1));
32      }
33      __except(printf("Filter in main\n"), EXCEPTION_EXECUTE_HANDLER)
34      {
35          printf("Handling block in main\n");
36      }
37      return 0;
38  }
```

在 VC6 中编译以上代码，会得到多个 C4509 号警告信息：

```
C:\...\ SehUwObj.cpp(32) : warning C4509: nonstandard extension used: '
FuncObjUnwind' uses SEH and 'bug1' has destructor
```

上面的警告告诉我们，FuncObjUnwind 函数使用了不属于 C++标准的 SEH 扩展（即__try 和__except），而且 bug1 对象有析构函数。

接下来还有一个 C2712 号错误：

```
C:\...\ SehUwObj.cpp(34) : error C2712: Cannot use __try in functions that require
object unwinding
```

这个错误明确地告诉我们，不可以在需要对象展开的函数中使用__try{}__except()扩展结构。察看 MSDN，对以上警告的一种解决建议是使用 C++的 try{}catch()结构。

如果在项目属性中不启用 C++异常处理（Project > Settings > C++ > 选取 C++ Language > 不选中 Enable Exception Handling），那么没有上面的编译错误，但警告信息还在。执行编译好的程序，得到的结果如下：

```
Bug 0 constructed.
Filter in main.
Handling block in main.
```

这个结果说明，由于执行第 17 行时发生了异常，程序仿佛从第 17 行（FuncObjUnwind 函数）直接飞到了第 33 行（main 函数）。FuncObjUnwind 函数中第 17 行后的代码和 main 函数中尚未执行完的 printf 函数都因为发生异常而被跳过，Bug0 对象也没能被析构，这是不符合 C++标准的，这正是编译器发出警告信息和错误信息的原因。

总结以上分析可以得出结论，VC 的__try{}__except()结构不支持 C++标准所规定的对象展开。这主要是因为对象展开需要记录对象的生存范围和析构函数地址等与 C++语言有关的信息，而__try{}__except()结构是一种既可以用于 C 语言，也可以用于 C++语言的结构，如果加入大量对 C++的支持，那么必然会影响使用 C 语言编写的大量系统代码和驱动程序代码的性能。因此，VC 编译器的做法是如果要支持对象展开，那么就使用 C++的 try{}catch()结构。

事实上，__try{}__except()结构的异常处理函数_except_handler3 会调用一个名为_local_unwind2 的函数执行局部展开动作。该函数的原型如下：

```
void _local_unwind2(_EXCEPTION_RECORD * frame, DWORD trylevel)
```

因为不支持对象展开，_local_unwind2 的实现比较简单，它只是遍历 scopetable，寻找其中的__try{}__finally 块（特征为 lpfnFilter 字段为空），找到后执行对应的 lpfnHandler 函数（即 finally 块），循环的结束条件是遍历了整个 scopetable 或到达参数 trylevel 所指定的块。我们将在下一节详细介绍以上过程。

24.7 __try{}__finally 结构

在对__try{}__except()结构有了比较深入的理解后，我们来看一下 Windows 结构化异常处理中的终结处理，也就是__try{}__finally()结构。我们依然以一个简单的控制台程序 SehFinally 为例（见清单 24-18）。

清单 24-18 SehFinally 程序的源代码

```
1   #include "stdafx.h"
2   #include <windows.h>
3
4   int SehFinally(int n)
5   {
6       __try
7       {
8          n=1/n;
9       }__finally
10      {
11         printf("Finally block is executed.\n");
12      }
13      return n;
14  }
15  int main(int argc, char* argv[])
16  {
17      __try // TryMain
18      {
```

```
19          printf("SehFinally got %d\n", SehFinally(argc-1));
20      }__except(printf("Filter expression in main is evaluated.\n"),
21          EXCEPTION_EXECUTE_HANDLER)
22      {
23          printf("Exception handling block is executed.\n");
24      }
25      return 0;
26  }
```

观察 SehFinally 函数的汇编指令（见清单 24-19），很容易看出编译器使用的编译方法与编译__try{}__except 结构非常类似。事实上，VC 使用统一的数据结构（scope_entry）和处理函数（_except_handler3）来处理__try{}__except 结构和__try{}__finally 结构。区分这两种结构的方法也非常直观，那就是__try{}__finally 结构所对应的 scope_entry 的 lpfnFilter 字段（过滤表达式）为空，也就是说，编译器是把__try{}__finally 结构看作过滤表达式为 NULL 的特殊__try{}__except 结构来编译的。具体来说，像把异常处理块编译成函数形式一样，终结块也被编译为一种函数形式，我们不妨将其称为终结块函数。在清单 24-20 中，第 29～32 行便是 SehFinally 函数的终结块函数。

清单 24-19　SehFinally 函数（发布版本）的反汇编结果

```
1   00401000 55            push    ebp             ;保护父函数的栈帧基地址
2   00401001 8bec          mov     ebp,esp         ;建立本函数的栈帧基地址
3   00401003 6aff          push    0FFFFFFFFh      ;分配并初始化 trylevel
4   00401005 68e0704000  push offset SehFinally!KERNEL32…+0x30 (004070e0)
5   0040100a 682c124000  push    offset SehFinally!_except_handler3 (0040122c)
6   0040100f 64a100000000    mov     eax,dword ptr fs:[00000000h]
7   00401015 50            push    eax             ;将以前的登记结构地址压入栈
8   00401016 64892500000000  mov     dword ptr fs:[0],esp ;登记动态建立的登记结构
9   0040101d 83ec08          sub     esp,8
10  00401020 53            push    ebx             ;保存需要保持原值的寄存器
11  00401021 56            push    esi             ;保存需要保持原值的寄存器
12  00401022 57            push    edi             ;保存需要保持原值的寄存器
13  00401023 c745fc00000000  mov     dword ptr [ebp-4],0;修改 trylevel
14  0040102a b801000000      mov     eax,1           ;准备被除数
15  0040102f 99            cdq     ;
16  00401030 f77d08          idiv    eax,dword ptr [ebp+8]  ;整除
17  00401033 894508          mov     dword ptr [ebp+8],eax  ;商赋给参数 1
18  00401036 c745fcffffffff  mov     dword ptr [ebp-4],0FFFFFFFFh ;离开保护块
19  0040103d e814000000 call SehFinally!SehFinally+0x56(00401056);调用终结块函数
20  00401042 8b4508        mov     eax,dword ptr [ebp+8]   ;准备返回值
21  00401045 8b4df0     mov  ecx,dword ptr [ebp-10h];取保存的前一个登记结构地址
22  00401048 64890d00000000  mov     dword ptr fs:[0],ecx ;恢复
23  0040104f 5f            pop     edi             ;恢复需要保持原值的寄存器
24  00401050 5e            pop     esi             ;恢复需要保持原值的寄存器
25  00401051 5b            pop     ebx             ;恢复需要保持原值的寄存器
26  00401052 8be5          mov     esp,ebp         ;恢复栈指针
27  00401054 5d            pop     ebp             ;恢复父函数的栈帧基地址
28  00401055 c3            ret                     ;返回，以下是终结块函数
29  00401056 6830804000    push    offset SehFinally!`string' (00408030)
30  0040105b e8a0000000    call    SehFinally!printf (00401100)
31  00401060 83c404        add     esp,4           ;释放调用 printf 函数压入的参数
32  00401063 c3            ret                     ;返回
```

因为无论保护块内是否发生异常，终结块都应该被执行，所以在函数正常执行路线上也会有对终结块函数的调用，例如第 19 行就是这样的调用。

那么，当有异常发生时，终结块函数是如何被调用的呢？在终结块函数的入口处（第 29 行）设置一个断点：

```
bp 00401056
```

然后执行程序，会有异常通知发给调试器，按 F5 继续，于是断点命中，观察栈序列会发现与正常执行 SehFinally 函数没什么两样。这种效果正是栈展开过程所追求的目标，也就

是当执行终结处理块或其他局部展开代码时，都将栈设置成这些代码所在函数的情况。观察程序的屏幕输出，可以看到屏幕上已经输出了一行信息：

```
Filter expression in main is evaluated.
```

可见，此时 main 函数中的过滤表达式已经被执行过了，它同意处理异常，系统才执行栈展开，于是被展开函数中的终结块被调用。单步跟踪第 29～32 行的指令，并回到上一级函数，此时看到的是_NLG_Return2 标号，它其实是局部展开函数_local_unwind2 的后半部分。在_local_unwind2 中调用 lpfnHandler 函数的下面便是_NLG_Return2 标号，继续执行会跳转到_local_unwind2 函数中的遍历范围表的起始处。至此，我们印证了是_local_unwind2 函数调用终结块函数。

从_local_unwind2 退出后是_except_handler3 函数，它是第 6 行指令注册的异常处理函数。继续跟踪_except_handler3 函数，直到它返回到 ExecuteHandler2 函数，此时再观察栈调用序列，便可以看到与清单 24-17 中的栈帧#01～#0b 几乎一样的栈回溯了。

清单 24-20 显示了_local_unwind2 函数的伪代码，其中包含了执行当前函数范围内的各个终结块的过程。

清单 24-20　_local_unwind2 函数的伪代码

```
1   void _local_unwind2(_EXCEPTION_REGISTRATION * pRegFrame,
2                       DWORD dwEndTrylevel)
3   {
4      DWORD dwPrevTrylevel=0x0FFFFFFFE;
5
6      push    pRegFrame                       //本代码块是要注册一个内嵌的异常处理器
7      push    0FFFFFFFEh                      //一个特殊的 trylevel 值
8      push    offset SehFinally!_global_unwind2+0x20 (00401154)//压入范围表地址
9      push    dword ptr fs:[0]                //旧的异常处理登记结构
10     mov     dword ptr fs:[0],esp            //登记到 FS:[0]中
11
12     while (pRegFrame->trylevel!= TRYLEVEL_NONE &&
13          pRegFrame->trylevel!= dwEndTrylevel)
14     {
15       pCurScopeEntry = pRegFrame->scopetable[pRegFrame->trylevel];
16       dwPrevTrylevel = pCurScopeEntry->previousTryLevel;
17       pRegFrame-> trylevel= dwPrevTrylevel;
18       if (!pCurScopeEntry ->lpfnFilter)   //判断是__try{}__finally 结构
19       {
20          pCurScopeEntry ->lpfnHandler();  //执行终结块
21  _NLG_Return2:
22       }
23     }
24     pop     dword ptr fs:[0]                //注销内嵌的异常处理器
25  }
```

参数中的 dwEndTrylevel 用来作为循环结束标志，即当循环到这个编号的__try 结构时停止遍历。对于本节的例子，__except_handler3 函数是以–1（TRYLEVEL_ NONE）为参数来调用的，含义是遍历指定栈帧中的所有__try 结构。

24.8　C++的 try{}catch 结构

在理解了操作系统的 SEH 工作机制和 VC 的__try{}__except()结构后，接下来要讨论的是 C++语言的异常处理，即 C++的 try{}catch 结构。为了与 SEH 相区别，C++的异常处理通常被简称为 CppEH（C plus plus Exception Handling）。

24.8.1 C++的异常处理

C++标准定义了 3 个与异常有关的关键字：try、catch 和 throw。其典型结构为：

```
try {
   // 被保护块
   throw [expression]
}
catch (exception-declaration) {
   // 处理符合声明类型的异常处理块
}
[[catch (exception-declaration) {
   //处理符合声明类型的异常处理块
} ] . . . ]
```

也就是使用 try 关键字和一对大括号将要保护的代码包围起来，其后可以有一个或多个由 catch 关键字开始的 catch 块。每个 catch 块由异常声明（exception-declaration）和异常处理两部分组成。异常声明的作用类似于过滤表达式，只不过这里是使用类型来进行匹配。异常声明的一个特例是省略号（...），意思是与所有异常都匹配。例如，在清单 24-21 所示的 CppEH 程序中，CppEH 函数的 try 块后共有 3 个 catch 块，前两个分别捕捉字符串类型和整数类型的异常，最后一个用于捕捉所有其他类型的异常。

清单 24-21 CppEH 程序的源代码

```
1   #include "stdafx.h"
2   class C{
3   public:
4      C(int n,int a){no=n;age=a;}
5      ~C(){printf("Object %d is destroyed [%d]\n",no,age);};
6      int no,age;
7   };
8   int CppEH(int n)
9   {                                        //EHRec = -1
10     C o(-1,100);                          //定义-1 号类实例
11     try                                   //Try0       // EHRec = 0
12     {                                     // EHRec = 1
13        C o0(0,o.age*n);                   //定义 0 号类实例
14        try              //Try1           // EHRec = 2
15        {                                  // EHRec = 3
16           C o1(1,o0.age/n);               //定义 1 号类实例
17           throw o1;                       //此语句前 EHRec 被设置为 4
18        }
19        catch(C e)                         // 捕捉 C 类型的异常
20        {
21           printf("Class C %d captured\n",e.no);
22        }
23        o.age=o0.age;                      //此语句前 EHRec 被设置为 2
24     }                                     // EHRec = 1
25     catch(...)                            // 捕捉所有类型的异常
26     {
27        printf("Async exception captured\n");
28     }
29     return o.age;                         // EHRec = 0
30  }
31  int main(int argc, char* argv[])
32  {
33     printf("Exception Handling in C++ [%d]!\n", CppEH(argc-1));
34     return 0;
35  }
36
```

随便跟一个参数执行调试版本的 CppEh.EXE 程序，此时 argc = 2，所以第 16 行的除法操作可以顺利进行，于是执行第 17 行的 throw 语句，即抛出一个 C++类类型的异常，这个异常会被第一个 catch 块捕捉到，其执行结果如清单 24-22 所示。

清单 24-22 带一个参数执行 CppEH 程序

```
c:\dig\dbg\author\code\bin\Debug>cppeh 888 //带一个参数执行调试版本的 CppEH 程序
```

```
Object 1 is destroyed [100]                 //执行 Catch 块前栈展开时析构对象 o1
Class C 1 is captured                       //执行第 19～22 行的 catch 块
Object 1 is destroyed [100]                 //Catch 块中的用户代码执行后析构对象 e
Object 1 is destroyed [100]                 //析构抛出的异常对象，参见第 24.9 节
Object 0 is destroyed [100]                 //对象 o0 被析构
Object -1 is destroyed [100]                //-1 号对象被析构
Exception Handling in C++ [100]!            //main 函数打印的消息
```

如果不带任何参数执行调试版本的 CppEh.EXE，因为 argc = 1，所以第 16 行会导致一个除零异常，对于 C++程序来说，这是个非语言级的异常，相对于使用 throw 语句抛出的 C++异常，又被称为异步异常（Asynchronous Exception）或结构化异常（Structured Exception）。普通 catch 块的异常声明只匹配 C++异常，对于除零这样的异步异常，只有第二个 catch 块有可能捕捉到。之所以说有可能，是因为这与编译器的设置有关，直接执行使用默认选项编译出的调试版本 CppEH 程序（不带参数），程序的执行结果为：

```
c:\dig\dbg\author\code\bin\Debug>cppeh       //不带参数执行调试版本的 CppEH 程序
Object 0 is destroyed [0]                   //对象 o0 被析构
Async exception captured                    //执行第 25～28 行的 catch 块
Object -1 is destroyed [100]                //对象 o-1 被析构
Exception Handling in C++ [100]!            // main 函数打印的消息
```

也就是说，使用默认选项编译的调试版本中的 catch(…)块捕捉到了除零异常。对于发布版本的 CppEH.EXE，带参数执行的结果是一样的，如果不带参数执行，那么会导致应用程序错误，启动 JIT 调试，这说明默认发布版本的 CppEH.EXE 的 catch(…)块没有捕捉到除零异常。

事实上，VC 编译器（VC6 和 VC2005 都有）有一个 C++异常有关的重要选项：

```
/EH{s|a}[c][-]
```

其中 s 代表只捕捉同步异常而且外部（extern）C 函数会抛出异常；a 代表捕捉同步和异步异常，c 代表外部（extern）C 函数不抛出异常。编译器对发布版本的默认设置为 EHsc，即只捕捉非 C 函数的同步异常，这样编译器就可以假定只有 throw 语句和函数调用语句才会发生异常，如果某些对象的生命期没有跨越 throw 语句和函数调用，那么在构建对象展开记录时就可以不考虑这些对象，从而减小编译出的目标代码大小，因此 EHsc 是最经济的设置。

如果在发布版本的项目属性中加入/EHa 开关（Settings → C++选 Customize，然后在编辑框中输入），重新编译，然后再执行，则 catch(…)块便也可以捕捉除零异常了。

24.8.2 C++异常处理的编译

了解了 C++异常处理的基本用法后，我们来看一下 VC 编译器是如何编译包含 try{}catch 结构的，仍以 CppEH 函数为例。在讨论之前要说明的一点是，C++标准定义了 C++异常处理的基本行为，但是并没有规定编译器应该如何编译有关的代码。因此不同编译器的实现机制会有所不同，我们以 VC 编译器为例。清单 24-23 给出了 CppEH 函数（发布版本）开头处的汇编代码。

清单 24-23 CppEH 函数的序言（发布版本）

```
1   CppEH!CppEH:
2   00401000 55              push    ebp
3   00401001 8bec            mov     ebp,esp
4   00401003 6aff            push    0FFFFFFFFh
5   00401005 68d8784000      push    offset CppEH!CloseHandle+0x22 (004078d8)
6   0040100a 64a100000000    mov     eax,dword ptr fs:[00000000h]
7   00401010 50              push    eax
8   00401011 64892500000000  mov     dword ptr fs:[0],esp
9   00401018 83ec24          sub     esp,24h
```

容易看出，以上代码与前面几节介绍使用__try{}__except()结构的情况很类似，但有以下两点不同。

第一，在压入处理函数的地址之前，压入的是 EHRec 变量的初始值–1，没有压入范围表指针（scopetable）。这是因为 try{}catch 结构不使用我们前面介绍的范围表结构，而是使用我们下面要介绍的 cxx_function_descr 和 tryblock_info 等结构。EHRec 变量的作用与__try{}__except 结构中使用的 trylevel 类似，但是其变化规则有所不同。这样一来，try{}catch 结构登记在 FS:[0]链条上的数据结构便与__try{}__except()所使用的_EXCEPTION_REGISTRATION 结构有所不同，MSVCR80 的调试符号中将这个结构称为 EHRegistrationNode，详情如下：

```
0:000> dt EHRegistrationNode                                  //CppEH 使用的登记结构
   +0x000 pNext          : Ptr32 EHRegistrationNode           //指向下一个节点
   +0x004 frameHandler    : Ptr32 Void                        //处理函数
   +0x008 state          : Int4B                              //即 EHRec
```

第二，对于__try{}__except()结构，第 5 行压入的是_except_handler3 这样的函数，而现在压入的显然不再是_except_handler3 函数。也就是说，使用 C++异常处理的函数所注册的异常处理函数与使用__try{}__except()结构时的处理函数是不同的。那么这个函数是什么？不妨反汇编其地址（004078d8）来看一下。

清单 24-24　编译器（VC6）动态产生的异常处理函数

```
0:000> u 004078d8
CppEH!CloseHandle+0x22:
004078d8 b820864000     mov     eax,offset CppEH!_TI1H+0x10 (00408620)
004078dd e9a299ffff     jmp     CppEH!__CxxFrameHandler (00401284)
```

可见这个函数只是一个过度，它将某个地址赋给 EAX 寄存器后便跳转到__CxxFrameHandler 函数。所以可以说这段代码只是__CxxFrameHandler 函数的一个特别入口，我们将这一小段代码称为 C++异常处理函数的入口片段，简称 CppEH 入口。CppEH 入口是编译器动态产生的，因为没有为其输出专门的符号，所以上面清单中（第 5 行）使用了邻近的符号（CppEH!CloseHandle）来做参照物。

事实上，CppEH 入口就是为了传递赋给 EAX 寄存器的那个地址值（00408620），或者说以这种特别的方式来传递参数。这个地址指向的是一个名为__cxx_function_ descr 的数据结构，称为 C++函数描述符，其定义如下：

```
typdedef struct __cxx_function_descr{
      UINT  magic                    //结构签名，固定为 0x19930520
      UINT  unwind_count             //unwind_table 数组所包含的元素个数
      unwind_info *  unwind_table    //用于描述展开信息的展开表
      UINT  tryblock_count           // tryblock 数组所包含的元素个数
      tryblock_info *  tryblock      //用于描述 try{}catch 结构的 Try 块表
      UINT  unknown [3]              //
} cxx_function_descr;
```

其中，magic 为一个固定的整数，始终为 0x19930520，当使用 throw 关键字抛出 C++异常时，异常参数信息 ExceptionInformation[0]中设置的也是这个值（见后文）。unwind_count 字段描述的是下面的 unwind_table 数组所包含的元素个数，unwind_table 的每个元素是一个 unwind_info，用于描述栈展开时所需的信息。tryblock_count 是下面的 tryblock 数组的元素个数，也就是函数内所包含的 Try 块的个数。使用 dd 命令加上 CppEH 入口代码中的地址便可以观察 CppEH 函数的描述结构：

```
0:000> dd 00408620
00408620  19930520 00000007 00408640 00000002
00408630  00408678 00000000 00000000 00000000
```

其中，19930520 是 magic 字段，00000007 代表 00408640 处有 7 个 unwind_info 结

构，后面的 2 代表这个函数中共有两个 Try 块，它们的描述位于 00408678。unwind_info 结构的定义如下：

```
typedef struct __unwind_info
{
    int    prev;              //这个展开任务执行后要执行的下一个展开处理器的 EHRec
    void (*handler)();        //执行这个展开任务的函数，即展开处理器（unwind handler）
} unwind_info;
```

对于 CppEH 函数，unwind_count = 7，unwind_table 数组的地址是 00408640，使用 dd 命令可以观察原始的数据：

```
0:000> dd 00408640 le
00408640  ffffffff 004078c0 00000000 00000000
00408650  00000001 004078c8 00000002 00000000
00408660  00000003 004078d0 00000002 00000000
00408670  00000000 00000000
```

其中第 2、4 列是 prev 字段，第 3、5 列是 handler 字段。可以看到某些 handler 字段为空，这是预留在这，没有真正使用。非零的 handler 字段值代表栈展开时要调用的函数地址。对其其中的 004078c0 进行反汇编：

```
0:000> u 004078c0
004078c0 8d4de0        lea     ecx,[ebp-20h]
004078c3 e91898ffff    jmp     CppEH!C::~C (004010e0)
```

可见这也是很短的一个转发性代码，共有两行汇编，第 1 行是将一个局部变量的地址赋给 ECX 寄存器，然后便跳转到类 C 的析构函数中，因为 this 调用协议是使用 ECX 寄存器来传递 this 指针的，所以这显然是将对象指针赋给 ECX，然后便调用这个对象的析构函数。对 004078c8 和 004078d0 进行反汇编，看到的结果非常类似，因此可以想象到这 3 个代码片段是与 CppEH 函数中的 3 个对象一一对应。

描述 try{}catch 布局的 tryblock_info 结构的定义如下：

```
typedef struct __tryblock_info
{
    int   start_level;                    //这个 Try 块起点的 EHRec 级别
    int   end_level;                      //这个 Try 块的终点的 EHRec 级别
    int   catch_level;                    //Catch 块的初始 EHRec 级别
    int   catchblock_count;               //catchblock 数组的元素个数
    const catchblock_info *catchblock;    //描述 Catch 块的数组
} tryblock_info;
```

在 CppEH 函数中有两个 Try 块，因此 tryblock 所指向的地址处有两个 tryblock_info，使用 dd 命令可以观察这两个结构：

```
0:000> dd 00408678 la
00408678  00000003 00000004 00000005 00000001
00408688  004086a0 00000001 00000005 00000006
00408698  00000001 004086b0
```

每个 tryblock_info 结构的长度是 20 个字节，即 5 个 DWORD 长，因此我们一共显示了 10 个 DWORD，前 5 个描述的是内层的 Try 块（第 14～18 行，我们将其称为 Try1），后 5 个描述的是外层的 Try 块（第 11～24 行，即 Try0）。

为了支持对象展开，编译器在描述 C++异常结构时，使用一个名为 EHRec 的内部变量来标记不同的区域。编译器产生的代码是使用 EBP-4 来索引 EHRec 变量，因此观察汇编代码中对局部变量 EBP-4 的赋值指令就可以看到编译器是如何设置 EHRec 的。对于 CppEH 函数，我们将 EHRec 的设置情况标记在清单 24-22 的注释中。

catchblock_count 字段用来记录 Try 块所拥有的 Catch 块数量，catchblock 数组用来描述每个 Catch 块的信息，每个元素是一个 catchblock_info 结构：

```
typedef struct __catchblock_info
{
```

```
    UINT           flags;               //标志
    const type_info *type_info;         //这个 Ctach 块要捕捉的类型
    int            offset;              //用来复制异常对象的栈偏移
    void          (*handler)();         //Catch 块处理函数，即 Ctach 内的代码
} catchblock_info;
```

其中 flags 字段可以包含一个或多个以下标志：

```
#define TYPE_FLAG_CONST       1         //常数
#define TYPE_FLAG_VOLATILE    2         //指定 volatile 特征
#define TYPE_FLAG_REFERENCE   8         //引用
```

type_info 指向一个类型信息，用于描述这个 Catch 块所要捕捉的异常类型，offset 字段用于指定一个相对栈帧基地址（EBP）的偏移值，异常分发函数会将异常对象复制到这个栈地址中，也可以理解为这就是 catch 关键字后的异常声明表达式中的那个异常变量（e）的偏移地址；handler 即 catch 块的异常处理代码的起始地址。

对于 CppEH 函数中的 Try0 块，它有一个 Ctach 块，根据前面的内存显示，它的地址是 004086b0，使用 dd 命令可以看到 catchblock_info 结构的各个字段值：

```
0:000> dd 004086b0
004086b0  00000000 00000000 00000000 00401098
```

可见，flags、type_info 和 offset 字段都为零，这是因为这个 Catch 块的表达式是…，即捕捉所有异常，不需要类型匹配和复制异常对象。00401098 是 Ctach 块的代码，使用 u 命令可以看到它确实与第 27 行的源代码相对应。

以下是 Try1 块的 Catch 块描述：

```
0:000> dd 004086a0
004086a0  00000000 00409040 fffffffc8 0040106f
```

其中 00409040 是 type_info 字段，fffffc8（-56）是 offset 字段，0040106f 是 handler 字段。type_info 结构的定义如下：

```
typedef struct __type_info
{
  const vtable_ptr *vtable;     //type_info 类的虚拟方法表指针
  char  *name;                  //类型名称
  char  mangled[32];            //可变长度的类型标志串
} type_info;
```

其中，name 字段用来指向类型的名称，它是按照需要而分配的，所以通常为 0，mangled 字段用来存放类型标志串，它是编译器按照名称修饰（参见 25.1 节）规则产生的类型名称，其长度是可变的，以 0 结束。以刚才观察的 Ctach 块捕捉的类型为例，它的 type_info 地址是 00409040，其内容为：

```
0:000> dd 00409040
00409040  004080f0 00000000 56413f2e 00404043
```

当异常发生寻找处理异常的 Catch 块时，会先比较 mangled 字段，匹配后再比较 __catchblock_info 结构中的 flags 字段，如果也匹配，则准备调用__catchblock_ info 结构中的 hanlder 函数。

图 24-4 归纳了上面介绍的这些数据结构和它们之间的关系。其中 cxx_ function_desc 是核心，它的 unwind_table 字段指向描述展开信息的 unwind_info 数组，tryblock 字段指向描述 Try 块布局的 tryblock_info 数组。一个 tryblock_info 结构描述一个 Try 块，它的 catchblock 字段指向描述 Catch 块的 catchblock_info 数组。

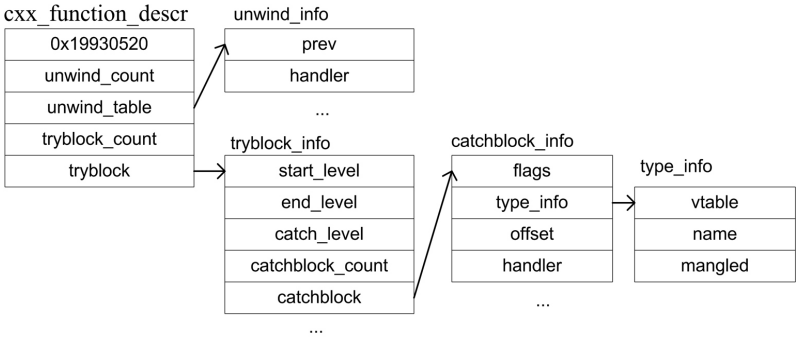


图 24-4　描述 try{}catch 结构的数据结构

从数量的角度来讲，一个使用了 try{}catch 结构的函数有一个 cxx_function_ desc 结构，函数中有几个 Try 块就有几个 tryblock_info 结构，同样，函数中共有多少个 Catch 块就有多少个 catchblock_info 结构，它们是与自己所属的 Try 块关联在一起的。

24.9　编译 throw 语句

上一节我们介绍了 VC 编译器编译 C++语言的 try{}catch 结构的方法，本节将继续介绍编译器是如何编译 C++的 throw 语句的。我们仍以上一节的 CppEH 程序（清单 24-22）为例，在函数 CppEH 中，第 17 行代码使用 throw 关键字抛出了一个类型为类 C（Class C）的异常。

清单 24-25 显示了 throw o1 语句（清单 24-22 的第 17 行）所对应的汇编代码。

清单 24-25　throw o1 语句所对应的汇编代码

```
1    004010a8 8b4dd8          mov     ecx,dword ptr [ebp-28h]
2    004010ab 894dc8          mov     dword ptr [ebp-38h],ecx
3    004010ae 8b55dc          mov     edx,dword ptr [ebp-24h]
4    004010b1 8955cc          mov     dword ptr [ebp-34h],edx
5    004010b4 6898664200      push    offset CppEH!_TI1?AVC (00426698)
6    004010b9 8d45c8          lea     eax,[ebp-38h]
7    004010bc 50              push    eax
8    004010bd e8be080000      call    CppEH!_CxxThrowException (00401980)
```

其中 ebp-28h 即对象 o1 的地址，因此第 1～4 行是把 o1 的两个成员 n 和 age 的值赋给栈上的临时对象（位于 ebp-38h），也就是要抛出的对象。第 5 行是将 CppEH!_TI1?AVC 的地址压入栈，它是编译器为类 C 产生的异常类型描述，是一个 cxx_exception_type 结构：

```
typedef struct __cxx_exception_type
{
  UINT               flags;                    //类型标志
  void                (*destructor)();         //析构函数
  cxx_exc_custom_handler  custom_handler;      //异常的定制处理器（Custom Handler）
  const cxx_type_info_table *type_info_table; //类型列表
} cxx_exception_type;
```

使用 dd 命令显示本例中的异常类型：

```
0:000> dd 00408610
00408610  00000000 004010f0 00000000 00408608
```

其中，00000000 是 flags 字段，004010f0 是析构函数的地址，即 C::~C()，00408608 是 type_info_table 字段，它是一个 cxx_type_info_table 结构：

```
typedef struct __cxx_type_info_table
{
    UINT           count;                //info 数组所包含的元素个数
    const cxx_type_info *info[3];        //类型信息，可变长度
} cxx_type_info_table;
```

尽管 info 字段声明包含 3 个元素，但其实际元素个数应根据 count 字段来确定，例如，对于前面的 cxx_exception_type 结构中类型那个列表：

```
0:000> dd 00408608
00408608  00000001 004085e8 00000000 004010f0
```

这说明这个类型列表中包含一个类型，其地址为 004085e8，该地址又是一个 cxx_type_info 结构：

```
typedef struct __cxx_type_info
{
    UINT           flags;               //标志，见下文
    const type_info *type_info;         //C++类的类型信息
    int       this_ptr_offset;          //基类的 this 指针偏移
    int       vbase_descr;              //虚拟基类的描述
    int       vbase_offset              //虚拟基类的 this 指针偏移
    unsigned int    size;               //对象大小
    cxx_copy_ctor   copy_ctor;          //C++复制构造函数（Copy Constructor）
} cxx_type_info;
```

使用 0:000> dd 004085e8

```
0:000> dd 004085e8 l7
004085e8  00000000 00409040 00000000 ffffffff
004085f8  00000000 00000008 00000000
```

其中 flags 字段可以包含如下标志：

```
#define CLASS_IS_SIMPLE_TYPE         1
#define CLASS_HAS_VIRTUAL_BASE_CLASS  4
```

00409040 是类型信息，与我们前面观察 catchblock 数组时看到的内层 catch 块的类型值是相同的，因为第二个 catch 块捕捉的和这里抛出的都是 C++类。

清单 24-25 的第 7 行是将栈上的临时对象 ebp-38h 的地址压入栈，第 8 行是调用用于产生异常的_CxxThrowException 函数，清单 24-26 给出了这个函数的伪代码。

清单 24-26　_CxxThrowException 函数的伪代码

```
1    void _CxxThrowException( void *object, const cxx_exception_type *type )
2    {
3        DWORD args[3];
4
5        args[0] = CXX_FRAME_MAGIC;
6        args[1] = (DWORD)object;
7        args[2] = (DWORD)type;
           RaiseException( CXX_EXCEPTION, EH_NONCONTINUABLE, 3, args );
8    }
```

显而易见，_CxxThrowException 函数内部先将一个常量（CXX_FRAME_ MAGIC）、要抛出的对象和描述异常的 cxx_exception_type 类型放入一个数组中，然后便调用 RaiseException API，调用时将异常代码指定为常量 CXX_EXCEPTION，即 0xe06d7363，对应的 ASCII 代码为.msc，因此，使用 throw 关键字抛出的异常都具有这个异常代码。另一点值得注意的是，在 RaiseException 的第二个参数中指定了 EH_NONCONTINUABLE 标志，这意味着 C++异常是不可以恢复继续执行的。

RaiseException API 是由 Kernel32.DLL 输出的一个标准 API，它内部会调用 RtlRaiseException，产生一个 EXCEPTION_RECORD 结构，并将 args 数组放入到 EXCEPTION_RECORD 结构的 ExceptionInformation 字段中作为异常记录的额外信息，然后便调用 NtRaiseExcaption 系统服务，进入内核态。到内核态后，其分发和处理流程就与 CPU 产生的硬件异常基本一致了。在分发的过程中，可以从异常代码及 EXCEPTION_RECORD 结构的 ExceptionInformation 字段中识别出一个异常是否是 C++异常：

- ExceptionInformation[0]字段总是等于常量 CXX_FRAME_MAGIC，即 0x19930520。
- ExceptionInformation[1]字段是 throw 语句所抛出的异常对象的地址。
- ExceptionInformation[2]字段是 throw 语句所抛出的异常对象的类型指针（cxx_exception_type 结构）。

归纳一下，throw 语句会将要抛出的 C++对象复制到一个临时对象中，然后将指向这个临时对象的指针和描述该对象类型的 cxx_exception_type 指针作为参数来调用 _CxxThrowException 函数。在分发异常时异常分发函数会从异常的额外信息 ExceptionInformation[2]中取出对象类型信息以寻找匹配的 Catch 块，找到后先执行栈展开，然后将 ExceptionInformation[1]中取出异常对象指针并将其复制给 Catch 块声明表达式中的变量（e），而后调用 Catch 块中的处理代码，当 Catch 块中的代码执行后，throw 语句抛出的临时对象会被析构。从这个分析我们知道，执行 Catch 块的方法与执行 except 块是有很大不同的，except 块是不返回的，因此异常分发函数是在做好所有分发和清理工作后执行 except 块，except 块中的代码执行好后便自然的执行它后面的代码了（参见清单 24-10），而 Catch 块中的代码执行好后它会返回到异常分发函数中。

补编内容 5 调试符号详解

补编说明：

这一部分内容本来属于《软件调试》第 25 章的后半部分，讲的是调试符号，根据符号的类型（Tag），逐一介绍每一种符号，包括符号的属性和示例。

写作这一内容时，我觉得这部分内容对于深刻理解软件调试和调试器的工作原理是非常有帮助的，可以解除很多疑惑。

但是在最后一轮压缩篇幅时，因为这部分内容相对独立，砍起来效果比较明显，于是就删除了。

本节（25.8 节，即正式出版的最后一节）介绍了符号文件中的 5 种数据表，以及表中所保存的数据对象。其中最重要是符号表，包含的记录也最多，从下一节开始，我们将分几节介绍符号表中的各种符号。

25.9 EXE 和 Compiland 符号

本节将介绍描述可执行文件的 SymTagExe 符号和描述编译素材（Compiland）的 SymTagCompiland、SymTagCompilandEnv、SymTagCompilandDetail 符号。

25.9.1 SymTagExe[1]

每个 PDB 文件都会包含一个 SymTagExe 类型的符号，简称 EXE 符号。EXE 符号用来描述这个 PDB 文件所对应的可执行文件（EXE、DLL、SYS 等）的信息，它是文件内所有其他符号的祖先，也是唯一没有父符号的符号。调用 IDiaSession 接口的 get_globalScope 方法可以得到 EXE 符号的 IDiaSymbol 指针。表 25-12 显示了 NameDeco.PDB（VC6 产生的调试版本符号文件）文件的 EXE 符号的各个属性值。

表 25-12 NameDeco.PDB 文件的 SymTagExe 符号

方法/属性	值
get_age	0
get_guid	{45DD54E4-0000-0000-0000-000000000000}
get_isCTypes	0
get_isStripped	0

续表

方法/属性	值
get_machineType	0
get_name	NameDeco
get_signature	1172133092
get_symbolsFileName	C:\...\code\chap25\NameDeco\Debug\NameDeco.pdb

表中第一列是 `IDiaSymbol` 接口的方法名，代表了 EXE 符号的一种属性，第二列是通过调用这个方法读取到的属性值。以下是对各个属性的解释说明。

- 年龄（Age）属性代表了这个 PDB 文件自创建以来的版本序号。因为 PDB 文件是支持递增或部分修改的，所以如果没有做过 Clean 或者 Rebuild All，大多时候 VC 都是在现有的 PDB 文件基础上做修改。另外，当我们在 VC 的集成环境中调试时，我们可以对被调试的程序作一些小的改动然后继续调试（即所谓的 Edit and Continue，简称 EnC），这时，VC 只是编译受影响的模块，并对 PDB 文件作局部更新。每次更新 PDB 后，Age 属性会被递增 1。
- GUID 属性用来代表一个符号文件的全局 ID。每创建一个新的 PDB 文件时（如 Rebuild All），链接器会生成一个新的 GUID 给该文件。因此同一个项目不同版本的 PDB 文件，其 GUID 可能是不同的。VC6 产生的符号文件不包含真正的 GUID，所以返回的是利用时间戳（Time Stamp）模拟生成的，其中的 45DD54E4 是下面的 Signature 值 1172133092 的十六进制。
- isCTypes 用来说明这个符号文件是否包含 C（语言）类型。
- isStripped 表示是否从这个文件剥离出了私有符号。
- MachineType 表示符号文件以及它对应的可执行文件的 CPU 类型。枚举类型 CV_CPU_TYPE_e 定义了各种 CPU 类型，0 表示英特尔的 8080 CPU。
- Name 属性是符号文件的主文件名（不含后缀），通常这也是符号文件所对应的目标文件的名称。
- Signature 属性是 PDB 文件创建时的时间戳。因此它的稳定性与 GUID 属性是一致的，EnC 时不会改变，但是 Rebuild All 时会改变。使用 Sig2Time 小程序（code\chap25\Sig2Time）可以把 Signature 中的数值转换为时间，如 1172133092 对应的时间是 2007 年 2 月 22 日 11:31:32。
- symbolsFileName 属性是当前 PDB 文件的完整路径。

调试器通常使用 GUID 值和 Age 值共同来标识一个符号文件，并以此为依据来寻找与一个可执行文件相匹配的符号文件。对于不包含 GUID 的符号文件会使用 Signature 来产生一个 GUID。WinDBG 的符号管理器也是使用这两个值的组合作为子目录名来存储同一个可执行文件的多个符号文件的。以 Beep.sys 为例，它的多个 PDB 文件是以如下规则存放在多个子目录中的：

```
beep.pdb\GUID+Age\beep.pdb
```

例如，以下是两个版本的 Beep.PDB 的完整路径：

```
D:\symbols\beep.pdb\65DC45B439164E4C9DEFF20E161DC74C1\beep.pdb
D:\symbols\beep.pdb\380E3FD31\beep.pdb
```

对于第一个，65DC45B439164E4C9DEFF20E161DC74C 来自于该符号文件的 GUID 值，紧跟其后的 1 是 Age 属性的值。第二个应该是较早的编译器产生的符号文件，380E3FD3 应该是 Signature 值，使用 Sig2Time 程序可以将其转换为时间：Thu Oct 21 02:18:59 1999。

25.9.2 SymTagCompiland[2]

Compiland 是编译方面的一个术语，用来泛指编译过程中所使用的用来产生目标文件的各种“素材”文件。包括各种源程序文件（.c、cpp、.rc 等）、中间目标文件（.obj、.res 等）和依赖的库文件（.lib、.dll 等）。举例来说，以下是驱动程序 Beep.PDB 中所描述的 5 个 Compiland：

```
{Compiland}[1]obj\i386\beep.obj(0)
{Compiland}[2]obj\i386\beep.res(0)
{Compiland}[3]ntoskrnl.exe(0)
{Compiland}[4]HAL.dll(0)
{Compiland}[5]* Linker *(0)
```

其中方括号中的数字是这个 Compiland 的 ID，圆扩号中的数字是这个 Compiland 的子符号的数目，因为这是个 Free 版本的公开符号文件，所以这些 Compiland 都没有子符号（扩号中都是 0）。表 25-13 列出了通过调用 `IDiaSymbol` 接口的方法（第 1 列）读取到的 Compiland 符号的属性。

表 25-13 Compiland 符号示例

方法/属性	值
get_editAndContinueEnabled	1
get_lexicalParentId	1224
get_libraryName	c:\dig\dbg\author\code\chap25\HiWorld\debug\BaseClass.obj
get_name	.\BaseClass.cpp
get_sourceFileName	0

因为调试符号是以关系数据库所惯用的表格形式存储的，所以从存储结构上来看，各个符号之间都是并列（平行）关系。但是为了体现出符号之间的附属和关联关系，除了 EXE 符号外，其他每个符号都有一个父词条 ID（Lexical Parent ID）属性，用来标识这个符号的词典编撰意义上的“父”符号。有了父词条 ID，本来平行存储的各个符号在逻辑上便有了父子关系，形成逻辑上的树状结构，根节点是 EXE 符号，其下一代便是很多个 Compiland 符号，每个 Compiland 符号又有很多子符号（见图 25-12）。为了节约篇幅，下文列出的符号属性中大多省略了父词条 ID 属性。

从图 25-12 中可以看到，每个 Compiland 符号下又包含了很多个子符号，比如描述环境信息的 SymTagCompilandEnv 符号，描述函数的函数符号等。

图 25-12 PDB 符号的树形逻辑结构

25.9.3 SymTagCompilandEnv[4]

`SymTagCompilandEnv` 类型的符号用来描述它所属的 Compiland 符号的环境信息。比

如图 25-12 中的 BaseClass.obj 符号有 6 个 SymTagCompilandEnv 类型的子符号，分别描述这个 Compiland 的某一方面信息。以下是常见 CompilandEnv 符号的名称、用途和典型值：

Obj 环境串，目标文件信息。Obj 和 res 类型的 Compiland 通常有一个 obj 环境串，Dll 类型的 Compiland 通常有很多个。比如 BaseClass.obj 有一个 obj 环境串，其值为目标文件的全路径。

Cwd 环境串，当前工作目录（Current Working Directory）。比如 BaseClass.obj 的 Cwd 环境串的值为 c:\...\chap25\HiWorld，即项目目录。

Cl 环境串，编译器的推动器程序（Compiler Driver，参见 20.3.2 节）的文件名称和路径，通常只有 C/C++源文件的 Compiland 才有这个子符号，比如 BaseClass.obj 的 Cl 环境串为 C:\Program Files\Microsoft Visual Studio 8\VC\bin\cl.exe。

Cmd 环境串，编译选项，即编译所属的 Compiland 时使用的参数，比如 BaseClass.obj 的 Cmd 环境串为 -Od -DWIN32 -D_DEBUG -D_WINDOWS -D_UNICODE -DUNICODE -Gm-EHs-EHc-RTC1-MDd-Yustdafx.h-Fpc:\...\chap25\HiWorld\Debug\HiWorld.pch-Foc:\...\chap25\HiWorld\Debug\-Fdc:\...\chap25\HiWorld\Debug\vc80.pdb-W3-c

Src 环境串，源程序文件。如 BaseClass.obj 的 Src 环境串为.\BaseClass.cpp。

Pdb 环境串，VCx0 符号文件。VCx0 符号文件是在 VC 集成环境中调试时所使用的符号文件，x 是 VC 编译器的主版本号。如 BaseClass.obj 的 Pdb 环境串为 c:\dig\...\Debug\vc80.pdb。

CompilandEnv 符号通常没有子符号。

25.9.4 SymCompilandDetail[3]

SymTagCompilandDetail 类型的符号用来描述它所属的 Compiland 符号的详细信息，包括相关的编译器和链接器名称版本等。表 25-14 显示了用来描述 BaseClass.obj 的 CompilandDetail 符号的各个属性。

表 25-14 SymTagCompilandDetail 类型的符号示例

方法/属性	值	简介
get_backEndBuild	50727	编译器后端的 Build 号
get_backEndMajor	14	编译器后端的主版本号
get_backEndMinor	0	编译器后端的小版本号
get_compilerName	Microsoft (R) Optimizing Compiler	编译器名称
get_editAndContinueEnabled	1	是否启用 EnC
get_frontEndBuild	50727	编译器前端的 Build 号
get_frontEndMajor	14	编译器前端的主版本号
get_frontEndMinor	0	编译器前端的小版本号
get_hasDebugInfo	1	是否包含调试信息
get_hasManagedCode	0	是否包含托管代码
get_hasSecurityChecks	1	是否使用/GS 编译
get_isCVTCIL	0	是否从公共中间语言（CIL）转化而来
get_isDataAligned	1	用户定义数据类型（UDT）是否内存对齐
get_isHotpatchable	0	是否使用/hotpatch 编译
get_isMSILNetmodule	0	是否包含微软中间语言的.Net 模块
get_language	CPP[1]	源程序语言，1 代表 C++
get_platform	Pentium III[7]	编译时选择的目标平台（CPU）

对于 DLL 类型的 Compiland，它的 CompilandDetail 符号的语言属性为 LINK，CompilerName 属性为链接器的名字，如：Microsoft (R) LINK。SymTagCompiland- Detail

类型的符号通常没有名称，也没有子符号。

25.10 类型符号

本节将介绍 PDB 文件中用于描述数据类型的各种符号。首先从描述基本数据类型的 SymTagBaseType 开始。

25.10.1 SymTagBaseType[16]

SymTagBaseType 符号用来描述程序语言定义的基本数据类型。CVCONST.H 中的 BasicType 枚举类型定义了用来表示基本类型的各个常量。

```
enum BasicType{ btNoType = 0,       btVoid = 1,      btChar = 2, btWChar = 3,
   btInt = 6,       btUInt = 7,     btFloat = 8,     btBCD = 9, btBool = 10,
   btLong = 13,     btULong = 14,   btCurrency = 25,  btDate = 26,
   btVariant = 27, btComplex = 28, btBit = 29, btBSTR = 30,btHresult = 31};
```

除了符号 ID 和类型外，每个 SymBaseType 符号通常还有以下几种属性。

- Base Type：即基本类型，值为 BasicType 枚举常量中的一个。
- Length：数据类型的长度，例如 void 类型的长度为 0。
- 父词条 ID（lexicalParentId）：通常为所在符号文件的 EXE 符号的 ID。
- 内存对齐（UnalignedType）：该类型是否内存对齐。
- 是否常量（ConstType）：即声明该类型时是否将其声明为常数（constant）。
- 易变性（VolatileType）：即声明该类型时是否附加了 volatile 关键字。

对于一种基本类型，符号文件中可能包含多个符号，对应不同的内存特征（const/volatile）。使用 SymView 工具打开 HiWorld_RES.PDB，左侧选择 SymTag 页，然后选中 BaseType[16]条目，便可以看到这个文件中所包含的基本类型符号。使用这种方法也可以观察其他类型的符号。

25.10.2 SymTagUDT[11]

SymTagUDT 符号用来描述用户（程序员）定义的数据类型（User Defined Type），包括结构、类和联合。数组和枚举类型分别由 SymTagArrayType 和 SymTagEnum 来描述。

一个 SymTagUDT 符号通常会有多个子符号，每个子符号描述它的一个成员或者方法，SymTagUDT 符号本身用来描述 UDT 的概括性信息。表 25-15 列出了描述 tagPOINT 结构和 CBaseClass 类的 UDT 符号的各种属性值和简单说明。

表 23-15 UDT 符号示例

属性	tagPOINT	CBaseClass	说明
get_constructor	0	1	是否有构造函数或析构函数
get_constType	0	0	是否定义为 const
get_hasAssignmentOperator	0	1	是否有赋值运算符
get_hasCastOperator	0	0	是否有类型转换（cast）运算符
get_hasNestedTypes	0	1	是否包含嵌套类型
get_length	0x8	0x21c	长度（字节数）
get_name	tagPOINT	CBaseClass	名称
get_nested	0	0	该类型是否是嵌套类型
get_overloadedOperator	0	0	是否有运算符重载
get_packed	0	0	是否被紧缩（成员紧密相连*）

属性	tagPOINT	CBaseClass	说明
get_scoped	0	0	是否出现在非全局域
get_udtKind	struct[0]	class[1]	UDT 子类
get_unalignedType	0	0	是否没有内存对齐
get_virtualTableShapeId	3295	4382	虚表符号的 ID
get_volatileType	0	0	是否定义为 volatile

*编译器默认会为结构中的每个成员进行自动的内存对齐，以提高内存访问效率。因为内存对齐，不同的成员间可能有一定的空隙。所谓紧缩就是在定义类型时通过#pragma pack 告诉编译器不要自动内存对齐，使各成员紧密相连。

知道了一个 UDT 符号的 ID 后，可以使用 `IDiaSession` 接口的 `findChildren` 方法寻找它的子符号，以便找到它的成员和方法。例如符号 tagPOINT 符号的子符号有 2 个，如表 25-16 所示。

表 25-16　tagPOINT 符号的子符号

ID	偏移	名称	可访问性	SymTag	DataKind	位置类型	父词条	类型 ID
3298	0	x	Public[3]	Data[7]	Member[7]	ThisRel[4]	1226	3299
3300	4	y	Public[3]	Data[7]	Member[7]	ThisRel[4]	1226	3299

最后一列的类型 ID（#3299）代表描述这个成员类型的符号 ID。寻找该符号，是一个 long 类型的基本类型符号，这正好与 tagPOINT 结构的成员类型相符。值得说明的是，这两个子符号的父词条 ID 并不是 tagPOINT 符号的 ID（#1276），而是 EXE 符号的 ID。这是因为，PDB 文件中除了有词典编撰意义上的父子关系外，还有类型角度的父子关系，`findChildren` 方法寻找的是类型定义角度的子符号。

类似的，表 25-17 是使用 `findChildren` 方法搜索到的 CBaseClass 符号（#3291）的子符号。其中，#3301 和#3303 是 CBaseClass 类的虚拟方法表（VTable），#3304、#3306 和#3316 是 CBaseClass 的数据成员，其他是类的方法。

表 25-17　CBaseClass 符号的子符号（部分）

ID	节	偏移	RVA	长度	名称	Tag	父词条	类型
3301		0				25	1226	3302
3303		0				25	1226	3302
3304		8			m_nPrivate	7	1226	3305
3306		12			m_szName	7	1226	3307
3308					CBaseClass	5	1226	3309
300	2		0x11550	0x43	CBaseClass::CBaseClass	5	292	3310
304	2		0x115a0	0x33	CBaseClass::~CBaseClass	5	292	3310
3311					Run	5	1226	3312
149	2		0x11cd0	0x30	CBaseClass::GetName	5	12	3314
3315					f	5	1226	3310
3316		532			__EventingCS	7	1226	3288
3322					operator=	5	1226	3323
3324					__vecDelDtor	5	1226	3325

父词条列中的 292 和 12 都是 Compiland 符号，分别是 BaseClass.obj 和 HiWorld.obj。SymTag 列中的 25、7 和 5 分别代表 VTable 符号、数据类符号和函数类符号，我们将在后面作详细介绍。

25.10.3　SymTagBaseClass[18]

SymTagBaseClass 符号用来描述派生类所属的基类。例如在 HiWorld 项目中，CClassFoo 是从 CBaseClass 派生而来的。

```
class CClassFoo : public CBaseClass
```

为了描述这一继承关系，CClassFoo 符号（UDT）会有一个 SymTagBaseClass 类型的

子符号。表 25-18 列出了这个符号的属性。

表 25-18　描述基类信息的 BaseClass 符号

属性	值	说明
get_access	public[3]	继承关系的可访问性
get_constructor	1	有构造函数或析构函数
get_constType	0	无 Const 特征
get_hasAssignmentOperator	1	有赋值运算符
get_hasCastOperator	0	没有类型转换运算符
get_hasNestedTypes	1	有嵌套类型
get_indirectVirtualBaseClass	0	不是间接虚拟基类
get_length	0x21c	长度
get_name	CBaseClass	名称
get_nested	0	不是嵌套类型
get_offset	0	该基类的子对象的偏移
get_overloadedOperator	0	没有重载运算符
get_packed	0	没有被紧缩
get_scoped	0	没有出现在非全局域
get_typeId	2912	类型 ID
get_udtKind	class[1]	UDT 类型
get_unalignedType	0	不是内存未对齐类型
get_virtualBaseClass	0	不是虚拟基类
get_virtualTableShapeId	4382	虚拟方法表形态符号的 ID
get_volatileType	0	非 volatile 类型

值得说明的是，虚拟基类（Virtual Base Class）和虚拟类是两个不同的概念，尽管 CBaseClass 包含纯虚方法，是个虚拟类，但对 CClassFoo 来说，它并不是虚拟基类，因为在声明继承关系时并没有指定 `virtual` 关键字。对于使用如下方法声明的 CClassNoo 类来说，CBaseClass 才是它的虚拟基类。

```
class CClassNoo :   virtual public CBaseClass
```

观察 CClassNoo 符号的基类子符号，可以看到 `get_virtualBaseClass` 方法返回 1。并且可以得到如下属性：`virtualBaseDispIndex`，值为 1；`virtualBasePointer-Offset`，值为 4；`virtualBaseTableType`，值为 3338，即虚拟基类表的类型符号。

25.10.4　SymTagEnum[12]

SymTagEnum 类符号用来描述枚举类型（enum）。下面以 excpt.h 中定义的 _EXCEPTION_DISPOSITION 枚举类型为例进行简要说明。使用 SymView 工具打开 HiWorld.PDB，然后在左侧的 SymTag 树视图中选择 Enum[12]，SymView 便会列出这个 PDB 文件中的枚举类型符号，从中可以找到名为_EXCEPTION_DISPOSITION 的符号，其 Base Type 属性为 int[6]，类型 ID（1227）指向的也是基本类型 int，其长度属性为 4，与 int 型相同，父词条 ID 指向的是 EXE 符号，其自身的符号 ID 为 1297。

使用 SymView 的搜索功能寻找 1297 符号的子符号（Symbols by Parent ID），可以搜索到这个枚举类型的所有成员（见表 25-19）。

表 25-19　枚举类型的子符号

ID	名称	值	Data Kind	Tag	父词条	类型 ID
3400	ExceptionContinueExecution	0	Constant[9]	7	1226	1227
3401	ExceptionContinueSearch	1	Constant[9]	7	1226	1227
3402	ExceptionNestedException	2	Constant[9]	7	1226	1227

3403	ExceptionCollidedUnwind	3	Constant[9]	7	1226	1227

SymTag 为 7，代表这些子符号都是数据类型的符号，Data Kind 为 9，代表是常量。类型 ID 1227 指向的是 int 基本类型。

25.10.5 SymTagPointerType[14]

SymTagPointerType 符号用来描述指针类型。例如，表 25-20 列出了无符号长整型指针（PULONG）符号的各个属性。

表 25-20 指针类型符号

属性	值	描述
get_constType	0	是否为 constant
get_length	0x4	长度
get_reference	0	是否是引用
get_type	5003	指向的类型，这里显示的是类型 ID
get_typeId	5003	类型 ID
get_unalignedType	0	是否未内存对齐
get_volatileType	0	是否为 volatile

其中 `get_type` 返回的符号描述就是基本类型 ULONG（unsigned long）。概言之，指针符号本身所包含的信息很少，其中最重要的内容就是它所指向的类型的 ID，通过这个 ID 可以得到进一步的信息。

25.10.6 SymTagArrayType

SymTagArrayType 符号用来描述数组类型。数组符号的信息描述方式与指针符号很类似，其本身只是记录数组的概括性信息，其元素类型和索引类型都是由其他符号来描述的。表 25-21 列出了描述全局数组变量（TCHAR szWindowClass[100]）的数组类型符号的属性。

表 25-21 数组类型符号的属性

属性	值	描述
get_arrayIndexTypeId	4704	描述数组索引类型的符号 ID
get_constType	0	是否为 const
get_count	100	数组的元素个数
get_length	200	数组的长度，字节数
get_typeId	4283	描述数组元素类型的符号 ID
get_unalignedType	0	是否未内存对齐
get_volatileType	0	是否为 volatile

符号 4283 描述的是基本数据类型 `wchar_t`。这个数组类型符号也可以用来描述同样维数、类型和长度的其他数组变量。事实上在 HiWorld 程序中，另一个全局数组 szTitle（TCHAR szTitle[100]）的声明和 `szWindowClass` 是一样的。所以这两个数据符号使用的是类型符号。但数据符号是两个，因为它们的名称和内存位置不同。

25.10.7 SymTagTypedef[17]

`SymTagTypedef` 类型的符号用来描述使用 `typedef` 关键字定义的类型别名。例如 NT_TIB 是_NT_TIB 的别名，INT 是 int 的别名，TEXTMETRICW 是 tagTEXTMETRICW 的别名，PINPUT 是指向 tagINPUT 结构的指针类型的别名等等。

Typedef 符号的名称属性就是 typedef 所定义的别名名称，Type ID 属性是描述原本类

型的符号 ID。

除了以上类型符号外，SymTagVTable 和 SymTagVTableShape 符号用来描述类的虚表，SymTagFriend 用来描述 C++中的友元信息，SymTagCustomType 符号用来描述编译器厂商定制的与编译器相关的类型，SymTagDimension 用来描述 FORTRAN 数组的维度信息（上下边界和维数——Rank），SymTagManagedType 符号用来描述使用 C#等语言开发的托管类型。因为篇幅关系，我们就不一一介绍这些类型符号了。描述函数类型和参数类型的 SymTagFunctionType 和 SymTagFunctionArgType 将在下一节介绍。

25.11 函数符号

函数是软件的重要组成部分，也是软件调试的重要目标。本节将介绍描述函数类型的 SymTagFunctionType 符号、描述函数参数类型的 SymTagFunctionArgType 符号、描述函数实例（Instance）的 SymTagFunction 符号、描述函数的调试起点和终点的 SymTagFunctionStart 和 SymTagFunctionEnd 符号，以及描述标号的 SymTagLabel 符号。

25.11.1 SymTagFunctionType[13]

函数类型（SymTagFunctionType）符号用来描述一个函数的原型，包括返回值类型、调用协议和参数信息等，这些特征有时也被称为函数签名（Function Signature）。表 25-22 列出了 CBaseClass 类的 Setup 方法所使用的函数类型。

表 25-22 Setup 方法的函数类型符号

属性	值	说明
get_callingConvention	CV_CALL_THISCALL[11]	调用协议，this 协议
get_classParentId	1257	所属类的类型符号，即 CBaseClass
get_count	2	参数个数
get_objectPointerType	5704	对象指针（this）的类型符号
get_thisAdjust	0	this 指针调整值
get_typeId	3778	返回值的类型符号

Setup 方法的原型是：int Setup（LPCTSTR szName），也就是它只有一个参数，但是我们知道调用类的（非静态）方法时总要隐含的传递 this 指针，因此尽管 Setup 方法的声明中只有 1 个参数，但是在实际调用时，参数的个数是 2，所以上表中的参数个数是实际调用参数的个数。

25.11.2 SymTagFunctionArgType[13]

如果一个函数的声明中包含参数，那么可以通过寻找它的函数类型符号的子符号找到描述参数的参数类型符号，即 SymTagFunctionArgType 符号。例如使用 SymView 工具搜索 Setup 函数符号（#3265）的子符号，可以找到一条结果，就是描述参数 szName 的符号，它的主要属性如下：

```
get_lexicalParentId    1255    父词条 ID，即 EXE 符号
get_symIndexId         3440    符号 ID
get_typeId             1263    类型符号的 ID
```

其中的类型符号 ID（1263）代表是指针类型的符号，它的基本类是 wchar_t，这与参数 szName 的类型 LPCTSTR 完全匹配。

25.11.3 SymTagFunction [5]

SymTagFunction 符号用来描述一个函数或者类方法实例。我们仍然以 CBaseClass 的 Setup 方法为例来进行介绍，表 25-23 给出了 Setup 函数符号的属性。

表 25-23 Setup 方法的函数符号

属性	值	说明
get_access	public[3]	可访问性，public 方法
get_addressOffset	0x620	地址偏移
get_addressSection	2	所在的节
get_classParentId	3266	所属类的类型符号 ID
get_customCallingConvention	0	没有使用定制的调用协议
get_farReturn	0	没有包含 far return
get_hasAlloca	0	方法中没有调用 alloca*
get_hasEH	0	没有使用非托管的异常处理
get_hasEHa	0	没有使用/EHa 编译
get_hasInlAsm	0	没有使用嵌入式汇编
get_hasLongJump	0	没有长跳转（long jump）**
get_hasSecurityChecks	0	没有使用安全检查（cookie）
get_hasSEH	0	没有使用结构化异常处理（SEH）
get_hasSetJump	0	没有使用 setjump**
get_InlSpec	0	没有被标记为 inline
get_interruptReturn	0	是否包含中断返回指令，如 iret
get_intro	1	是引入 virtual 的方法***
get_isNaked	0	是否具有 naked 属性，该属性告诉编译器不要加入序言和结语
get_isStatic	0	不是静态方法
get_length	0xc3	函数的长度
get_lexicalParentId	316	父词条 ID，即 BaseClass.obj
get_locationType	static[1]	位置属性
get_name	CBaseClass::Setup	名称
get_noInline	0	没有 noinline 标记
get_noReturn	0	没有 noreturn 标记
get_noStackOrdering	0	安全检查（GS）时可以栈定序
get_notReached	0	不具有“never reached”特征
get_optimizedCodeDebugInfo	0	不属于包含调试信息的优化代码
get_pure	0	不是纯虚函数
get_relativeVirtualAddress	71200	函数入口相对于模块起点的地址
get_typeId	1261	函数类型符号的 ID
get_undecoratedName	****	未修饰过的名称
get_virtual	1	是虚函数
get_virtualAddress	0x11620	函数入口的虚拟地址
get_virtualBaseOffset	4	这个虚函数在虚函数表中的偏移

*alloca 是从栈上分配内存的 C 标准函数。

**setjump 可以把栈环境保存起来，以便以后可以使用 longjump 跳回到这个状态。Setjump 和 longjump 一起可以实现跨函数的跳转。C 中的异常处理使用这种方法来执行异常处理和恢复代码。

***如果父类和子类中都有相同的虚方法，那么最早将该方法声明为 virtual 的那个方法就是所谓的引入 virtual 的方法（Introducing Virtual Function）。

****内容为 public: virtual int __thiscall CBaseClass::Setup(wchar_t const *)，使用 get_undecoratedNameEx 方法可以取得不同形式的修饰名。

25.11.4 SymTagFunctionStart[21]

SymTagFunctionStart 符号用来描述源代码调试时函数的可调试起点。举例来说，当我们在源程序中对函数入口设置断点时，调试器实际上会把断点设置在函数序言之后的某一位置上，FunctionStart 符号便是用来描述这一位置的。

通过 SymView 的 Compiland 视图，我们可以观察函数的 FunctionStart 符号，表 25-24 列出了 Setup 方法的 FunctionStart 符号的主要属性。

表 25-24　Setup 方法的 FunctionStart 符号

属性	值	说明
get_addressOffset	0x643	调试起点的偏移地址
get_addressSection	2	调试起点所在的节号
get_locationType	static[1]	位置类型
get_relativeVirtualAddress	0x11643	调试起点相对于模块起点的地址
get_virtualAddress	0x11643	调试起点的虚拟地址

从上表可以看出，FunctionStart 符号所定义的位置是 0x11643，这与 VC2005 调试器实际设置断点的位置（0x00411643）是一致的，因为我们使用 SymView 工具观察的值没有算模块基地址。

25.11.5　SymTagFunctionEnd[22]

与 FunctionStart 符号的功能类似，SymTagFunctionEnd 符号用来描述源代码级调试时函数的调试结束位置。还是以 Setup 方法为例，它的 FunctionEnd 符号所定义的偏移地址是 0x6cd，对应的是设置返回值的下一条语句：

```
004116C8  mov       eax,1
   return TRUE;
004116CD  pop       edi
```

FunctionStart 符号和 FunctionEnd 符号主要是供源代码级调试使用的，在汇编窗口调试时可以把断点设置在起始点之前，也可以跟踪到截止点之后。

25.11.6　SymTagLabel[9]

SymTagLabel 符号用来描述程序中的标号（Label）。因为标号实际上记录的就是某段代码的地址，所以标号既可以是跳转语句的目标，也可以被当作函数来调用。表 25-25 列出了一个典型标号符号的各种属性，这个符号描述的是 HiWorld 程序中的 TAG_EXIT 标号，位于 LabelTest 函数中。

表 25-25　标号符号的属性

属性	值	说明
get_addressOffset	0xa019	节内偏移
get_addressSection	1	所属节
get_locationType	static[1]	位置类型
get_name	TAG_EXIT	名称
get_relativeVirtualAddress	0xb019	标号的 RVA
get_virtualAddress	0xb019	标号的虚拟地址

其中的父词条 ID（#1786）代表的是所在函数的符号，因为这个标号是函数内标号。对于函数外标号，父符号是它所在的 Compiland。

25.12　数据符号

数据符号（SymTagData[7]）用来描述程序中的常量和各种变量，包括局部变量、全局变量、类的成员和参数。我们先来看数据符号的公共属性，然后再分别介绍各种数据符号。

25.12.1 公共属性

除了符号 ID 和 SymTag，数据符号通常还具有如下属性。

名称：即变量或常量的名称。

类型：用来描述变量类型的类型符号。

取值：常量的取值，该属性是一个 VARIANT 结构，可以表示各种类型的常量。对于变量，应该从其存储空间（内存、栈或寄存器）读取它的值。

数据种类（Data Kind）：数据的种类，其值为表 25-26 所列出的 DataKind 枚举常量之一。

位置类型（Location Type）：数据存放的位置类型，其值为表 25-27 所列出的 LocationType 枚举常量之一。

表 25-26 描述数据符号种类的 DataKind 枚举类型

常量	值	说明
DataIsUnknown	0	未知种类
DataIsLocal	1	局部变量
DataIsStaticLocal	2	静态局部变量
DataIsParam	3	参数
DataIsObjectPtr	4	对象指针，即 this 指针
DataIsFileStatic	5	文件作用域内（File-scoped）的静态变量
DataIsGlobal	6	全局变量
DataIsMember	7	成员变量
DataIsStaticMember	8	静态变量
DataIsConstant	9	常量

这里说明什么是文件作用域内（File-scoped）的静态变量，简单来说，就是带有 `static` 关键字的定义在函数之外的变量。如果没有加 `static`，那么它就是一个普通的全局变量。加了 static 之后，它的作用域便被限定在它所在的源文件（linkage）中，这样的好处是所在文件中的所有函数可以使用它，而且可以防止它的值被其他文件内的函数所修改。这种变量主要用在 c 程序中，C++中因为有了对象封装技术，较少使用这种变量了。例如 crtexe.c 中，将记录命令行参数的 `argc` 和 `argv` 变量定义为文件作用域内的静态变量：

```
// All the below variables are made static global for this file. …
static int argc;   /* three standard arguments to main */
static _TSCHAR **argv;
```

枚举类型 LocationType 定义了数据的位置特征，包括存贮位置所在空间，位置偏移所使用的参照物等（表 25-27）。

表 25-27 描述数据符号位置类型的 LocationType 枚举类型

常量	值	说明
LocIsNull	0	没有位置信息
LocIsStatic	1	位置是静态的
LocIsTLS	2	位于线程局部存储区（Thread Local Storage）中
LocIsRegRel	3	位置信息是相对于寄存器的
LocIsThisRel	4	位置信息是相对于对象指针（this）的
LocIsEnregistered	5	对应的变量被寄存器化了，位置信息是寄存器号
LocIsBitField	6	位置信息是二进制位域
LocIsSlot	7	位置信息是中间语言（MSIL）的 slot
LocIsIlRel	8	位置信息是中间语言（IL）相关的
LocInMetaData	9	位于元数据内
LocIsConstant	10	常量
LocTypeMax	11	本枚举类型所定义的位置类型总数

25.12.2 全局数据符号

使用 SymView 工具打开一个 PDB 文件（HiWorld.PDB），然后在 SymTag 视图中选择 Data[7]，SymView 便会显示出 PDB 文件中的所有全局常量和变量符号，包括文件作用域内的静态变量和真正的全局变量，我们把这些符号统称为全局数据符号（见表 25-28）。

表 25-28 全局数据符号示例

属性	szWindowClass	envp	PowerDeviceD0
get_addressOffset	0x1f8	0x3e4	N/A
get_addressSection	4	4	N/A
get_dataKind	Global[6]	File Static[5]	Constant[9]
get_lexicalParentId	1255	1255	1255
get_locationType	static[1]	static[1]	Constant[10]
get_name	szWindowClass	envp	PowerDeviceD0
get_relativeVirtualAddress	107000	107492	N/A
get_typeId	3382	3865	3292
get_virtualAddress	0x1a1f8	0x1a3e4	N/A
get_value	N/A	N/A	1

表 25-28 列出了 HiWorld.PDB 中的 3 个全局数据符号的属性值。这 3 个数据符号分别描述了全局变量 `szWindowClass`、文件作用域内的静态变量 `envp` 和常量 `PowerDeviceD0`。对于前两者，它们有详细的地址属性，包括节、偏移、虚拟地址和 RVA，调试器就可以通过这些信息读到它们的当前值，不该使用 `get_value` 方法。对于 `PowerDeviceD0` 常量，它没有地址信息，使用 `get_value` 方法可以读到它的值（1）。

`typeId` 属性的值代表了数据的类型符号，依次是数组类型、指针类型和枚举类型 _DEVICE_POWER_STATE。

25.12.3 参数符号

通过 SymView 的 Compiland 视图，浏览 Compiland 下的函数，然后展开一个有参数的函数，便可以看到它的参数符号。知道了一个函数符号的 ID 后，也可以使用 SymView 的“Symbols by Parent ID”搜索功能找到这个函数的参数符号。表 25-29 列出了 Setup 方法的参数符号的属性值。

表 25-29 参数符号示例

属性	This	lpszName	说明
get_dataKind	Object Ptr[4]	Param[3]	数据种类
get_lexicalParentId	332	332	父词条 ID，即所在的 Compiland
get_locationType	RegRel[3]	RegRel[3]	位置类型，相对于寄存器
get_name	this	lpszName	名称
get_offset	-8	8	偏移
get_registerId	22	22	寄存器 ID
get_typeId	3826	3827	类型 ID

类型 ID 代表了参数的类型符号，例如，上面两个类型符号都是指针符号（SymTag-PointerType），分别指向 CBaseClass 类（UDT）和 `wchar_t` 类型。位置类型中的 RegRel（Register Relative）代表偏移信息是相对于寄存器的，Register ID 中的 22 代表的是 EBP 寄存器（CVCONST.H 中的 `CV_HREG_e` 枚举定义了各个寄存器的 ID）。大家知道，C++方法使用的是 this 调用协议，this 指针是通过 ECX 寄存器来传递的，其他参数是使用栈来传递的，这样一来，可以理解 EBP + 8 就是第一个参数 `lpszName` 的地址，那么，为什么 EBP - 8 是 this 参数的地址呢？这又是编译器对调试的一个特别支持。因为寄存器的值经常变化，所以为了使得隐含的 `this` 参数也可以被追溯，编译器会在栈帧中分配空间并生成代码将其保存

到栈中，保存的位置就是 ebp-8。观察 Setup 方法的汇编代码，可以看到在函数的序言部分有一条 MOV 指令（地址为 0x00411640 ）来做这个工作，即：

```
0041162C  push        ecx
0041162D  lea         edi,[ebp-0DCh]
00411633  mov         ecx,37h
00411638  mov         eax,0CCCCCCCCh
0041163D  rep stos    dword ptr es:[edi] 0041163F  pop       ecx
00411640  mov         dword ptr [ebp-8],ecx
```

在发布版本中，为了提高性能，编译器通常不会加入这个支持，这时针对 this 符号调用 `get_offset` 方法不再返回 S_OK，它的位置类型也变为 Enregistered，寄存器 ID 值为 18，代表 ECX。

25.12.4 局部变量符号

根据分配方式，局部变量又可分为分配在静态变量区内的静态局部变量，分配在栈上的局部变量和分配在寄存器中的寄存器变量。以 HiWorld 项目的 HiWorld.cpp 文件中的 FuncTest 函数为例，`dwEntryCount` 是静态的局部变量，`cf` 和 `szMsg` 是分配在栈上的普通局部变量，循环变量 i 是比较典型的可能被分配在寄存器中的变量，因为它的大小可以被寄存器所容纳，而且访问它的频率可能比较高，放入寄存器中有利于提高性能。

使用 SymView 工具打开调试版本的 HiWorld.PDB 文件，在 Compiland 视图浏览到 HiWorld.obj 下的 FuncTest 函数，可以看到它共有 9 个数据符号。其中前 4 个是编译器的变量检查功能自动加入的静态局部变量(参见 22.11 节)，hWnd 是参数，其他 4 个是 FuncTest 函数中的 4 个局部变量（表 25-30）。

表 25-30　局部变量符号示例

属性	cf	szMsg	i	dwEntryCount
get_addressOffset	N/A	N/A	N/A	0x90
get_addressSection	N/A	N/A	N/A	4
get_dataKind	Local[1]	Local[1]	Local[1]	Static Local[2]
get_lexicalParentId	155	155	166	155
get_locationType	RegRel[3]	RegRel[3]	RegRel[3]	static[1]
get_name	cf	szMsg	i	dwEntryCount
get_offset	-560	-1088	-1100	N/A
get_relativeVirtualAddress	N/A	N/A	N/A	0x1a090
get_registerId	22	22	22	N/A
get_typeId	1333	1335	1327	1334
get_virtualAddress	N/A	N/A	N/A	0x1a090

从上表可以看出，静态变量具有虚拟地址、节和地址偏移等信息，因此可以从内存中读到它的值，即使不在执行这个函数，也可以使用调试器读取它的值，从这个意义上来说，它与全局变量很类似。其他 3 个局部变量都是分配在栈上的，它的偏移属性是相对于栈帧的基地址寄存器（EBP）。所有变量符号都有一个 typeId 属性用来查询它的类型。

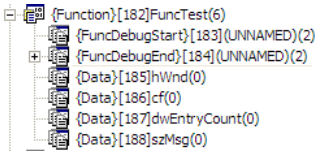


图 25-13　FuncTest 函数（发布版本）的调试符号

图 25-13 显示的是发布版本的 FuncTest 函数的子符号情况，从图中可以看到，用于变量检查功能的静态变量在发布版本中不见了，另外，循环变量 i 也被优化掉而分配到寄存器中了。

25.13 Thunk 及其符号

当 CPU 在不同字长（16 位和 32 位）、不同空间（内核空间和用户空间）或不同模块之间（DLL）的代码之间跳转（执行转移）时，往往需要有一些代码来完成衔接、映射，以及转换翻译等工作，人们给这样的代码片段起了个很特别的名字，叫做 Thunk。关于 Thunk 一词的来源有很多种说法，有人说它是 Think 的一种特别的过去式（类似于 drink > drunk），有人说这个词来源于早期计算机在运算时所发出的声音，也有人说它就是“THe-fUNCtion”的缩写。不管怎样，大家只要知道 Thunk 就是用来实现某些特别的函数调用和执行转移而设计的一小段代码。下面我们先分别介绍几种常见的 Thunk，然后介绍描述 Thunk 的符号。

25.13.1 DLL 技术中的 Thunk

Thunk 的一种典型应用就是在动态连接库（Dynamic Link Library，DLL）技术中实现动态绑定和跨模块调用。让我们通过一个例子来理解 DLL 技术的简要原理和 Thunk 在其中的作用。

在使用了 MessageBox API 的 HiWorld 程序中，我们可以看到如下代码：

```
0:001> u HiWorld!MessageBoxW
HiWorld!MessageBoxW:
00412538 ff2500b44100    jmp     dword ptr [HiWorld!_imp__MessageBoxW (0041b400)]
```

其中的 `HiWorld!_imp__MessageBoxW` 可以理解成是一个全局变量，使用 `dd` 命令可以观察到它的值：

```
0:001> dd HiWorld!_imp__MessageBoxW l1
0041b400  77d9610a
```

使用 `ln` 命令可以看到 `77d9610a` 是 `User32.DLL` 中的 `MessageBoxW` 函数的地址：

```
0:001> ln 77d9610a
(77d9610a)   USER32!MessageBoxW   |  (77d96158)   USER32!SetSysColors
```

位于 HiWorld 中的 MessageBoxW 是一个典型的 Thunk，它的作用是跳转到全局变量 `HiWorld!_imp__MessageBoxW` 所指定的地址。

25.13.2 实现不同字长模块间调用的 Thunk

Thunk 的另一种典型应用就是在不同字长的模块间进行函数调用以完成翻译和转换任务。例如在 Windows 9x 中，为了支持旧的 16 位软件，系统允许从 32 位的应用程序（EXE）中调用 16 位模块（DLL）中的函数。但因为 32 位代码使用的数据类型和寄存器与 16 位代码有很大差异，所以在调用前，必须将函数参数翻译为 16 位代码所使用的数据类型，然后将线程的栈切换为供 16 位代码所使用的栈，当函数返回时，再将 16 位函数的返回值翻译成 32 位。为了简化这一过程，Windows 9x 设计了一种称为 Flat Thunk 的机制，使用一对 DLL（一个 32 位，一个 16 位）来执行从 32 位代码到 16 位代码的函数调用，并且提供了一个专门的编译工具（Thunk Compiler）来帮助生成这对 DLL。主要步骤是先使用这个工具根据 Thunk 脚本产生合适的汇编代码，然后使用汇编编译器（ml.exe）将汇编代码分别编译成 16 位和 32 位的目标文件（obj），最后再分别生成 16 位和 32 位的 DLL。在这两个 DLL 中用来做翻译和转换工作的代码，便被称为 Thunk。今天的 Windows 操作系统已经不再支持 16 位的模块，因此 Flat Thunk 技术已经过时，只有个别的 API（例如 ThunkConnect32）还残留在 MSDN 中。

25.13.3 启动线程的 Thunk

在 kernel32.dll 中我们可以看到一段名为 BaseProcessStartThunk 的代码，其汇编指令如下：

```
0:001> u kernel32!BaseProcessStartThunk
kernel32!BaseProcessStartThunk:
7c810665 33ed            xor     ebp,ebp
7c810667 50              push    eax
7c810668 6a00            push    0
7c81066a e945690000      jmp     kernel32!BaseProcessStart (7c816fb4)
```

这段代码将 EBP 寄存器清 0，然后向栈中压入两个参数，便无条件地跳转到 BaseProcessStart 函数（永远不会再返回到这段代码）。事实上，这段代码就是每个进程的初始线程开始在用户态执行的起点。其中 EAX 的值是进程的启动函数地址，即登记在 PE 文件头结构中的入口地址。BaseProcessStartThunk 是一个典型的 Thunk，它做一点简单的操作后便跳转到真正执行任务的目标函数。那么为什么要使用这个 Thunk 呢？因为系统在创建进程的初始线程时，是不方便向线程的栈中压入内容的，所以便只是将进程的启动函数放入到线程上下文结构（CONTEXT）的 EAX 寄存器中，这样，当用户线程开始执行时，便需要一段简单代码来调用使用栈来接收参数的 BaseProcessStart 函数。当然，这段代码的另一个作用就是将 EBP 寄存器清零，这为回溯栈帧设置了一个很好的终结点。例如，我们在调试器中显示线程的栈回溯信息时，其最后一个栈帧指针的内容总是 0。

```
0012ffc0 7c816fd7 01a3f6f0 0000008c 7ffd8000 HiWorld!wWinMainCRTStartup+0xd
0012fff0 00000000 0041128f 00000000 78746341 kernel32!BaseProcessStart+0x23
0:000> dd 0012fff0 l1          // 显示最末栈帧基地址处的值
0012fff0  00000000             // 0 代表这是线程的最后一个栈帧
```

与 BaseProcessStartThunk 类似的还有 BaseThreadStartThunk，它将 EBP 寄存器清零并将 EBX、EAX 和 0 压入到栈中后便跳转到 BaseThreadStart 函数。

25.13.4 Thunk 分类

在 NTDLL.DLL 中，我们也可以看到 Thunk 代码，比如 LdrInitializeThunk（目标函数为 LdrInitialize），在 ATL 中也使用了 Thunk，因为篇幅关系，我们不一一叙述。DIA SDK 中的 THUNK_ORDINAL 枚举类型将 Thunk 归纳为表 25-31 所列出的 7 种类型。

表 25-31 Thunk 分类

常量	值	说明
THUNK_ORDINAL_NOTYPE	0	普通的 Thunk
THUNK_ORDINAL_ADJUSTOR	1	用于调整 this 指针的 Thunk
THUNK_ORDINAL_VCALL	2	用于调用虚函数的 Thunk
THUNK_ORDINAL_PCODE	3	用于调用 P-Code 的 Thunk
THUNK_ORDINAL_LOAD	4	加载地址并跳转到这个地址
THUNK_ORDINAL_TRAMP_INCREMENTAL	5	增量性的 Trampoline Thunk
THUNK_ORDINAL_TRAMP_BRANCHISLAND	6	分支性的 Trampoline Thunk

其中 P-Code 是 Packed Code 的缩写，它是一种比.NET 技术更早的中间代码技术。其基本思想是通过使用中间代码（P-Code）来缩减可执行文件的大小，当执行时，链接到可执行文件中的一个小的引擎将 P-Code 解释为机器码来执行。VB 支持将程序编译为 P-Code。

Trampoline Thunk 用于将函数调用从一个空间弹到另一个空间，比如从内核空间到用户空间或反之。

25.13.5 Thunk 符号

Thunk 符号（SymTagThunk[27]）用来描述程序中的 Thunk。表 25-32 列出了用来描述我们前面介绍的 HiWorld!MessageBoxW 的 Thunk 符号的各种属性。

表 25-32 Thunk 符号示例

属性	值	说明
get_addressOffset	0x1538	Thunk 的地址偏移
get_addressSection	2	Thunk 的节地址
get_length	0x6	Thunk 的代码长度，6 个字节
get_lexicalParentId	449	父词条 ID，代表的是 User32.DLL
get_locationType	static[1]	位置类型
get_name	MessageBoxW	名称
get_relativeVirtualAddress	0x12538	Thunk 的 RVA
get_thunkOrdinal	standard thunk[0]	Thunk 类型
get_virtualAddress	0x12538	Thunk 过程的虚拟地址

从 PDB 的编纂结构角度来讲，Thunk 符号是 Compiland 符号的子符号。比如，表 25-32 所描述的 MessageBoxW 符号的父词条 ID（449）代表的是描述 User32.DLL 的 Compiland 符号。

补编内容 6 调试器标准

补编说明：

这一节本来属于《软件调试》第 28 章的最后一节，讲的是 Java 调试器的标准。

《软件调试》的出发点是写软件调试的一般原理的，但是为了不流于空泛，始终是选择真实的产品（软件和硬件）为例来讲的。但是因为众所周知的事实，无论是 CPU 还是核心软件，如果只在主流的产品中选择，那么选择的余地很有限。

我在多年前做过几年的 Java 开发，对 Java 还算熟悉，于是写作了这一内容，写作时很觉得这一内容巩固调试器原理和了解 Java 都听不错。特别是可以证明，原理是相通的，不论是 Java 调试器，还是 C/C++调试器，它们内部其实很像。

在最后一轮压缩篇幅时，这一节被删除了，主要原因是这一内容是属于“锦上添花（姑且用这个词吧）”类型，删除后也不会影响整个内容的完整性。

28.9 JPDA 标准

Java 是一种流行的动态语言，使用 Java 语言开发的程序先编译为字节码，然后由 Java 虚拟机（JVM）按照 JIT 编译的方式来执行。Java 程序可以在各种装有 Java 运行环境（Java Runtine）的系统中运行，具有非常好的跨平台特征，因此被广泛应用到企业应用、网络服务、网站开发、移动和嵌入式设备等领域。本节我们介绍用于调试 Java 程序的调试器标准——JPDA。我们介绍的版本是 Java SE 6。

28.9.1 JPDA 概貌

JPDA 的全称是 Java 平台调试器架构（Java Platform Debugger Architecture），它由图 28-13 所示的 3 个部分组成，即：

1. Java 调试器接口（Java Debug Interface），简称 JDI，这是一套供调试器或者性能分析工具使用的 Java API，用来访问目标程序的内部状态和调用 Java 虚拟机的各种调试功能。JDI 工作在调试器进程中，负责与调试器的其他部分进行交互。
2. Java 虚拟机工具接口（JVM Tool Interface），简称 JVM TI，它是 JVM 对外提供调试服务的标准接口，它工作在被调试的 Java 进程中，负责与 Java 虚拟机进行交互收集

调试信息并接收和处理来自 JDI 的命令请求。

3. Java Debug Wire Protocol ，简称 JDWP，这是 JVM TI 与 JDI 之间进行通信的协议，二者通过这个协议交换信息。

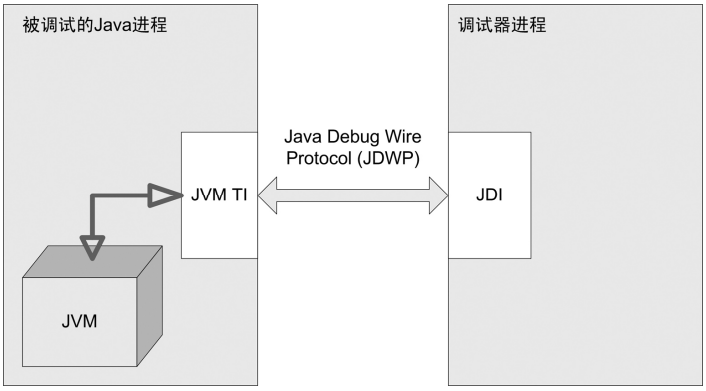


图 28-13　Java 平台调试器架构（JPDA）

从用户的角度来看，位于调试器进程中的部分（包括 JDI）常被称为前端（Front End），被调试器进程中负责支持调试器前端工作的部分常被称为后端（Back End）。下面我们对以上 3 个部分进行分别介绍。

28.9.2　JDI

JDI 是一套纯粹的 Java API 库，用来简化使用 Java 语言来开发 Java 调试器，它封装了通过 JDWP 与 JVM TI 通信的过程，使得调试器开发者只要调用这些简单易用的 Java API 就能开发出强大的调试器。尽管调试器开发者可以直接使用 JDWP 或 JVM TI，但是使用 JDI 是推荐的方法。JDI 它主要包含表 28-7 所列出的一些 Java 包（package）。

表 28-7　JDI 的各个包

包名	功能
com.sun.jdi	JDI 的核心包，定义了位于目标进程内的类型、数据和虚拟机本身的本地镜像（mirror）。利用这些镜像调试器可以像访问本地那样访问目标进程内的数据、类型和虚拟机状态
com.sun.jdi.connect	调试器进程的 JVM 与目标进程的 JVM 之间的连接
com.sun.jdi.connect.spi	包含了一系列接口和类，用于开发新的传输服务（TransportService）
com.sun.jdi.event	JDI 事件和事件处理
com.sun.jdi.request	向调试器后端发送请求，请求的种类见下文

通过 com.sun.jdi.request 包中的类，调试器可以向目标进程中的调试器后端发出请求来订阅调试事件，表 28-8 列出了发送请求的接口名称和所对应的事件通知。

表 28-8　JDI 中用于发送调试请求的接口

接口	请求
AccessWatchpointRequest	当目标进程中的指定字段被访问时得到通知
BreakpointRequest	当目标虚拟机执行到指定位置时得到通知
ClassPrepareRequest	当目标虚拟机准备指定的类时得到通知

续表

接口	请求
ClassUnloadRequest	当指定的类被从目标虚拟机中卸载时得到通知
ExceptionRequest	当目标虚拟机中发生异常时得到通知
MethodEntryRequest	当目标虚拟机中的方法被调用（进入方法）时得到通知
MethodExitRequest	当目标虚拟机中的方法返回（从方法中出来）时得到通知
ModificationWatchpointRequest	当一个字段被设置时得到通知
MonitorContendedEnteredRequest	当目标进程中的线程等待到了存在竞争的监视对象（monitor object）后得到通知
MonitorContendedEnterRequest	当目标进程中的线程开始等待一个已经被其他线程得到的监视对象时得到通知
MonitorWaitedRequest	当目标进程中的线程结束等待监视对象时得到通知
MonitorWaitRequest	当目标进程中的线程即将等待监视对象时得到通知
StepRequest	当目标虚拟机中发生单步执行时得到通知
ThreadDeathRequest	当目标虚拟机中的线程终止时得到通知
ThreadStartRequest	当目标虚拟机中的线程启动时得到通知
VMDeathRequest	当目标虚拟机终止时得到通知
WatchpointRequest	定义一个被监视的字段

以上接口均派生自 EventRequest 接口，因此可以使用 EventRequestManager 类来统一管理和发送，当请求所对应的条件满足时，一个对应的 Event 对象会被放入到事件队列（EventQueue）中，然后调试器可以从队列中取出这个事件。

28.9.3 JVM TI

与 JDI 是一个 100%使用 Java 语言开发的 Java 库不同，Java 虚拟机工具接口（JVM Tool Interface，简称 JVM TI）是一个本地的（native）编程接口。通过这个接口，工具软件可以观察 Java 虚拟机（JVM）中所执行程序的状态并控制它的执行，以实现调试、Profiling、监控、线程分析、覆盖率分析（Coverage Analysis）等目标。

JVM TI 是 JDK 5.0 所引入的，它取代了本来用于 Profiling 的 JVMPI（Java Virutal Machine Profiler Interface）接口和用于调试的 JVMDI（Java Virtual Machine Debug Interface）接口。

从架构角度来看，JVM TI 是 JVM 为工具程序所提供的一个本地接口。使用这个接口的客户模块被称为主体（Agent）。主体可以在进程内，也可以在进程外。进程内主体通常是以动态链接库的形式存在的。可以使用任何支持 C 语言调用规范的本地语言开发主体，JVM TI 的数据结构和函数定义在 jvmti.h 文件中。在图 28-13 所示的架构中，调试器使用的是 JVM TI 的进程外接口（Out-of-process Interface）。

JVM TI 共提供了几十个函数，分为 21 个组，表 28-9 列出了这些函数的名称和功能。为了节约篇幅，除了组名占一整行外，表格的每一行列出了两个函数。

表 28-9 JVM TI 的函数

函数名称	功能	函数名称	功能
内存管理（Memory Management）			
Allocate	分配内存	Deallocate	释放内存
线程（Thread）			
GetThreadState	取线程状态	GetCurrentThread	取当前线程的结构
GetAllThreads	取所有线程	SuspendThread	挂起线程
SuspendThreadList	挂起列表中的线程	ResumeThread	恢复线程

ResumeThreadList	恢复列表中的线程	StopThread	停止线程
InterruptThread	中断线程	GetThreadInfo	读取线程信息
GetOwnedMonitorInfo	取当线程所拥有的监视对象信息	GetOwnedMonitorStackDepthInfo	取线程所拥有的监视对象信息和锁定这些对象的栈帧深度
GetCurrentContendedMonitor	取指定线程等待进入的竞争性监视对象	RunAgentThread	启动主体线程
SetThreadLocalStorage	设置线程局部存储的指针	GetThreadLocalStorage	读取线程局部存储的指针
线程组（Thread Group）			
GetTopThreadGroups	读取 VM 中的顶层线程组	GetThreadGroupInfo	取线程组的信息
GetThreadGroupChildren	取线程组的活动线程和子组		
栈帧（Stack Frame）			
GetStackTrace	读取指定线程的栈帧信息	GetAllStackTraces	读取所有存活线程的栈帧信息
GetThreadListStack-Traces	读取指定线程列表中各个线程栈帧信息	GetFrameCount	读取指定线程栈的栈帧数
PopFrame	弹出指定线程的当前栈帧	GetFrameLocation	读取当前执行位置
NotifyFramePop	当指定栈帧弹出时产生 FramePop 事件		
强制提前返回（Force Early Return）			
ForceEarlyReturnObject	强制返回结果为指定对象或派生对象的方法提早返回	ForceEarlyReturnInt	强制返回结果为整数的方法提早返回
ForceEarlyReturnLong	强制返回结果为长整数的方法提早返回	ForceEarlyReturnFloat	强制返回结果为浮点数的方法提早返回
ForceEarlyReturnDouble	强制返回结果为双精度浮点数的方法提早返回	ForceEarlyReturnVoid	强制无返回结果的方法提早返回
堆（Heap）			
FollowReferences	遍历对象引用	IterateThroughHeap	发起遍历堆中的所有对象
GetTag	读取与指定对象关联的 Tag	SetTag	设置与指定对象关联的 Tag
GetObjectsWithTags	读取与指定 Tag 关联的所有对象	ForceGarbageCollection	强制内存回收（GC）
1.0 堆（Heap 1.0）			
IterateOverObjectsReachableFromObject	从指定对象遍历可到达的对象	IterateOverReachable-Objects	从根遍历对象
IterateOverHeap	遍历堆	IterateOverInstancesOfClass	遍历类的实例
局部变量（Local Variable）			
GetLocalVariableObject	读取指定对象类型的局部变量的取值	GetLocalVariableInt	读取 int/short/char/byte/boolean 型局部变量的值
GetLocalVariableLong	读 Long 型局部变量的值	GetLocalVariableFloat	读取浮点类型局部变量的值
GetLocalVariableDouble	读取双精度类型局部变量的值	SetLocalVariableObject	设置指定对象类型的局部变量的取值
SetLocalVariableInt	读取 int 等类型局部变量的值	SetLocalVariableLong	设置 Long 型局部变量的值
SetLocalVariableFloat	设置浮点类型局部变量的值	SetLocalVariable-Double	设置双精度类型局部变量的值

断点（Breakpoint）			
SetBreakpoint	设置断点	ClearBreakpoint	清除断点
观察字段（WatchedField）			
SetFieldAccessWatch	当指定类的指定字段被访问时产生 Field-Access 事件	ClearFieldAccessWatch	清除字段访问监视
SetFieldModification-Watch	当指定类的指定字段被修改时产生 Field-Modification 事件	ClearFieldModification Watch	清除字段修改监视
类（Class）			
GetLoadedClasses	读取 VM 中已经加载的类	GetClassLoaderClasses	读取指定类加载器所加载的类
GetClassSignature	读取类的签名	GetClassStatus	读取类的状态
GetSourceFileName	读取类的源文件名	GetClassModifiers	读取类的访问标志
GetClassMethods	读取类的方法	GetClassFields	读取类的字段
GetImplementedInter-faces	读取类的 super-interface	GetClassVersion-Numbers	读取类的版本号
GetConstantPool	读取类的 constant pool 的原始数据	IsInterface	是否是接口
IsArrayClass	是否是数组类	IsModifiableClass	类是否可以修改
GetClassLoader	读取指定类的类加载器	GetSourceDebug-Extension	读取类的调试扩展信息
RetransformClasses	更新已经加载类的字节码	RedefineClasses	更新类的字节码
对象（Object）			
GetObjectSize	读取对象大下	GetObjectHashCode	读取对象的哈希代码
GetObjectMonitorUsage	读取监视对象的使用情况		
字段（Field）			
GetFieldName	读取字段名称	GetFieldDeclaringClass	读取字段的声明类
GetFieldModifiers	读取字段的访问标志	IsFieldSynthetic	是否是编译器产生的假造字段
方法（Method）			
GetMethodName	读取方法名称	GetMethodDeclaring-Class	读取方法的声明类
GetMethodModifiers	读取方法的访问标志	GetMaxLocals	取局部变量槽的数量
GetArgumentsSize	取参数大小	GetLineNumberTable	取行号表
GetMethodLocation	读取方法的位置	GetLocalVariableTable	取局部变量表
GetBytecodes	读取字节码	IsMethodNative	是否是本地方法
IsMethodSynthetic	是否是编译器加入的假造方法	IsMethodObsolete	是否是过时的方法
SetNativeMethodPrefix	设置本地方法的前缀	SetNativeMethod-Prefixes	设置本地方法的多个前缀
原始的监视器（Raw Monitor）			
CreateRawMonitor	创建一个原始监视器	DestroyRawMonitor	销毁一个原始监视器
RawMonitorEnter	获取独自拥有权	RawMonitorExit	释放独自拥有权
RawMonitorWait	等待监视器的通知	RawMonitorNotify	通知等待监视器的单一线程
RawMonitorNotifyAll	通知等待监视器的所有线程		
JNI 函数介入（JNI Function Interception）			
SetJNIFunctionTable	设置 JNI 函数表	GetJNIFunctionTable	读取 JNI 函数表

事件管理（EventManagement）			
SetEventCallbacks	设置事件回调函数	SetEventNotification-Mode	设置事件通知模式
GenerateEvents	产生事件		
扩展机制（Extension Mechanism）			
GetExtensionFunctions	读取扩展函数	GetExtensionEvents	读取扩展事件
SetExtensionEventCall-back	设置扩展事件的回调函数		
JVM TI 的功能（Capability）			
GetPotentialCapabilities	读取当前环境所支持的潜在可选功能	AddCapabilities	增加功能
RelinquishCapabilities	放弃指定的功能	GetCapabilities	读取当前环境所支持的可选功能
计时器（Timers）			
GetCurrentThreadCPUTimerInformation	读取当前线程所用CPU 时间的信息	GetCurrentThreadCPU-Time	读取当前线程所使用的 CPU 时间
GetThreadCPUTimerInformation	读取指定线程所用CPU 时间的信息	GetThreadCPUTime	读取指定线程所使用的 CPU 时间
GetTimerInformation	读取计时器信息	GetTime	读取时间
GetAvailableProcessors	读取 JVM 可用的CPU 数量		
类加载器搜索（Class Loader Search）			
AddToBootstrapClass-LoaderSearch	增加Bootstrap类加载器的搜索路径	AddToSystemClass-LoaderSearch	增加系统类加载器的搜索路径
系统属性（System Properties）			
GetSystemProperties	读取系统属性	GetSystemProperty	读取单个系统属性
SetSystemProperty	设置系统属性		
其他			
GetPhase	读取 VM 执行的当前所处阶段	DisposeEnvironment	关闭与 JVM TI 的连接
SetEnvironmentLocal-Storage	设置环境信息	GetEnvironmentLocal-Storage	读取环境信息
GetVersionNumber	读取 JVM TI 的版本	GetErrorName	读取错误代码的符号名
SetVerboseFlag	控制输出信息的种类	GetJLocationFormat	读取位置的表示方法

以上我们列出了 JVM TI 的当前版本（JDK6）所定义的所有函数。我们之所以列出这些函数，是因为这些函数反映了实现调试功能所需的底层支持，理解这些函数有利于理解调试功能的工作原理。特别是以上函数也反映了 Java 语言的很多重要特征，比如 JVM、类、类加载器，以及用于同步的监视对象等。

28.9.4 JDWP

简单来说，Java Debug Wire Protocol（JDWP）是一个通信协议，它定义了被调试进程（Debuggee）与调试器进程通信的格式和方法。JDWP 允许调试器进程和被调试程序工作在不同的机器上，以支持远程调试。

JDWP 中的所有数据通信是基于数据包的，它定义了两种基本的包类型：命令包和回复包。每个包都由固定长度的头结构（Header）和可变长度的数据组成。命令包的头格式为。

- 长度（4 bytes）：整个包的长度，包括这个字段。

- ID（4 bytes）：包的唯一 ID。
- 标志（1 byte）：标志位，0x80 位如果为 1，则代表是回复包。
- 命令集合（1 byte）：命令所属的命令集合。
- 命令（1 byte）：命令。

回复包的头格式为：

- 长度（4 bytes）：整个包的长度，包括这个字段。
- ID（4 bytes）：包的唯一 ID。
- 标志（1 byte）：标志位。
- 错误代码（2 bytes）：0 代表成功，非零代表发生错误。

调试器进程通过命令包向被调试进程中的调试器后端发送命令，调试器后端收到命令后通过回复包发送处理结果。命令包中的命令集合字段和命令字段指定了要执行的命令，目前 JDWP 定义了 18 个命令集合，共 100 多条命令。JDWP 的文档详细定义了每个命令包的格式和用法，因为篇幅关系，在此从略。

JDK 中包含了使用 JPDA 的 3 个例子程序，分别是用来显示踪迹信息的 Trace 程序，一个简单的命令行调试器 JDB 和一个简单的 GUI 调试器 Javadt。感兴趣的读者可以下载 JDK 并阅读它们的源代码。

补编内容 7 WinDBG 内幕

补编说明：

这一节本来属于《软件调试》第 29 章的后半部分，讲的是 WinDBG 调试器实现不同类型的调试会话的方式，分内核调试和远程用户态调试两个部分。

考虑到第 30 章还会介绍有关的主题，在最后一轮压缩篇幅时，这一节被删除了。

29.8 内核调试

前面两节我们以活动的用户态调试目标为例，介绍了调试会话和 WinDBG 接收处理命令的过程。本节我们将把这些内容推广到内核调试的情况，严格来说，是通过通信电缆进行双机内核调试的情况。

29.8.1 建立内核调试会话

对于用户态调试目标，无论是启动新进程开始调试还是附加到一个已经存在的进程，一旦选定了程序文件或者进程 ID，那么调试器就立刻与调试目标建立起了调试会话。内核调试与此略有不同，当我们选择内核调试（WinDBG 的 File>Kernel Debug…）并指定通信方式（COM、1394 或 USB）后，虽然调试器会开始启动调试会话并等待与调试目标建立连接，但是直到与调试目标建立起通信连接并检测到目标的基本信息后，内核调试会话才真正建立。为了便于理解，我们将内核调试会话的建立过程分为两个阶段：启动阶段和连接阶段。

举例来说，我们可以在一台没有连接串行数据线的计算机上，选择开始内核调试（File>Kernel Debug，选择 COM），点击确定后，WinDBG 就会创建调试会话线程，然后调用 StartSession 开始调试会话，其执行过程如清单 29-8 所示。

清单 29-8　开始内核调试会话

```
0:001> kn
 # ChildEBP RetAddr
00 00dffce8 020c5dec dbgeng!ConnLiveKernelTargetInfo::ConnLiveKernelTargetInfo
01 00dffd38 020bf86d dbgeng!LiveKernelInitialize+0x6c
02 00dffd5c 0102a532 dbgeng!DebugClient::AttachKernelWide+0x7d
03 00dfffa4 0102a9bb WinDBG!StartSession+0x5f2
04 00dfffb4 7c80b6a3 WinDBG!EngineLoop+0x1b
05 00dfffec 00000000 kernel32!BaseThreadStart+0x37
```

其中，2 号栈帧是调用 DebugClient 类的 AttachKernel 方法，其原型如下：

```
HRESULT  IDebugClient5::AttachKernelWide(
   IN ULONG  Flags,    IN OPTIONAL PCWSTR  ConnectOptions);
```

其中 Flags 可以为 DEBUG_ATTACH_KERNEL_CONNECTION（0）、DEBUG_ATTACH_LOCAL_KERNEL（1）和 DEBUG_ATTACH_EXDI_DRIVER（2）三个值之一。因为我们选择的是通过串口进行双机调试，所以 Flags 参数为 0，ConnectOptions 参数的值如下：

```
0:001> du 010662e4 // 可以从栈帧#02 的参数中得到这个地址
010662e4  "com:port=com1,baud=115200"
```

AttachKernel 函数根据指定的参数调用 LiveKernelInitialize 来初始化调试活动内核目标所需的调试器引擎对象，0 号栈帧显示在执行 ConnLiveKernelTargetInfo 类的构造函数，也就是构建 ConnLiveKernelTargetInfo 类的实例，这个构造好的实例会保存在 g_Target 全局变量中。AttachKernel 返回后，StartSession 做了些公共的初始化工作后，便也返回了。至此，内核调试会话的启动阶段结束，调试会话线程开始等待与调试目标建立连接。

我们知道内核调试引擎（Kernel Debug Engine，简称 KD）是 Windows 操作系统内核的一部分，它的功能就是调试器一起工作实现内核调试。因此，从通信的角度来看，在进行双机内核调试时，通信的一方是主机上的调试器，另一方是目标系统上的内核调试引擎，即 KD。

当 WinDBG 收到 KD 的第一个数据包后，ConnLiveKernelTargetInfo 类的 WaitForEvent 方法会调用 ProcessStateChange 方法，后者会调用 NotifyDebuggee- Activation 方法，这与用户调试态调试时在处理进程创建事件前调用 NotifyDebuggee- Activation 的情况类似。清单 29-9 显示了其执行过程。

清单 29-9　建立连接的过程

```
0:001> kn                                                          //调试会话线程
 # ChildEBP RetAddr
00 00d0fa20 7c90e9c0 ntdll!KiFastSystemCallRet                     //调用内核服务
01 00d0fa24 7c8025cb ntdll!ZwWaitForSingleObject+0xc               //残根函数
02 00d0fa88 020c275a kernel32!WaitForSingleObjectEx+0xa8           //等待同步对象
03 00d0faa8 01055bdc dbgeng!DebugClient::DispatchCallbacks+0x4a
04 00d0fab8 0102a7e1 WinDBG!EngSwitchWorkspace+0x9c                //切换工作空间
05 00d0fad0 01027378 WinDBG!SessionActive+0x171                    //激活会话
06 00d0fadc 020b71ea WinDBG!EventCallbacks::SessionStatus+0x28
07 00d0faf0 020b38ec dbgeng!SessionStatusApcData::Dispatch+0x2a
08 00d0fb28 020b3b4f dbgeng!ApcDispatch+0x4c
09 00d0fb78 020b7181 dbgeng!SendEvent+0xcf
0a 00d0fb98 02130ef5 dbgeng!NotifySessionStatus+0x21
0b 00d0fbc4 0213466b dbgeng!NotifyDebuggeeActivation+0x55
0c 00d0fef4 02133eb1 dbgeng!ConnLiveKernelTargetInfo::ProcessStateChange+0x1bb
0d 00d0ff10 020ceacf dbgeng!ConnLiveKernelTargetInfo::WaitForEvent+0xe1
0e 00d0ff34 020cee9e dbgeng!WaitForAnyTarget+0x5f                  //等待目标
0f 00d0ff80 020cf110 dbgeng!RawWaitForEvent+0x2ae
10 00d0ff98 0102aadf dbgeng!DebugClient::WaitForEvent+0xb0         //等待事件
11 00d0ffb4 7c80b6a3 WinDBG!EngineLoop+0x13f                       //调试循环
12 00d0ffec 00000000 kernel32!BaseThreadStart+0x37
```

其中，栈帧#0b~#05 是向调试器中注册的回调对象通知会话状态（Session Status）改变。 栈帧#04 是当调试器检测到了调试目标的基本信息后，要切换到与其相匹配的工作空间。在切换到新的工作空间前，WinDBG 通常会显示图 29-19 所示的对话框，询问用户是否要保存当前使用的默认工作空间（'Kernel default'）。

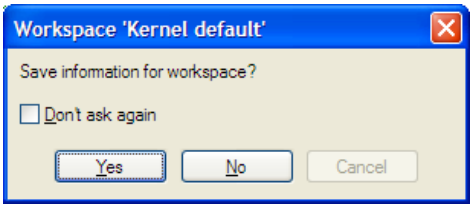


图 29-19　WinDBG 在收到内核调试目标激活消息后显示的工作空间对话框

值得注意的是，3 号栈帧调用了 DispatchCallbacks 方法，这会使当前线程（调试会话线程）进入等待状态。因此，当 WinDBG 显示图 29-8 所示的对话框时，调试器的调试会话线程是处于等待状态的，而且此时目标系统也是处于冻结状态的。只有这个对话框被关闭后，UI 线程才会调用 FinishCommand 命令，然后 UpdateEngine，再调用 ExitDispatch 方法，增加信号量的计数，使调试会话进程被唤醒，清单 29-18 显示了这一过程。

清单 29-10　应用内核调试工作空间

```
0:000> kn
 # ChildEBP RetAddr
00 0006cd00 0102af20 dbgeng!DebugClient::ExitDispatch    //通知调试会话线程
01 0006cd10 010279d5 WinDBG!UpdateEngine+0x30            //更新调试器引擎
02 0006cd18 01027acf WinDBG!FinishCommand+0x15
03 0006cd3c 01027b89 WinDBG!AddStringCommand+0xef
04 0006cd5c 01054d11 WinDBG!AddStringMultiCommand+0xa9   //执行命令恢复目标执行
05 0006d218 01055901 WinDBG!Workspace::Apply+0x5c1       //应用新的工作空间
06 0006d240 01055b03 WinDBG!UiSwitchWorkspace+0xd1       //切换工作空间
07 0006d260 0103d539 WinDBG!UiDelayedSwitchWorkspace+0x33//迟后的工作空间切换
08 0006ddf4 7e418724 WinDBG!FrameWndProc+0x1e09          //窗口过程
…                                                        //省略其他栈帧
```

调试会话线程等待到信号量后，EngineLoop 再次调用 DebugClient 类的 WaitForEvent 方法等待下一个调试事件。至此内核调试会话的连接阶段结束，调试会话完全建立。

29.8.2　等待调试事件

清单 29-11 显示了使用串行电缆进行双机内核调试时，WinDBG 的调试会话线程等待调试事件的过程。

清单 29-11　等待内核调试事件

```
0:001> kn
 # ChildEBP RetAddr
00 00e0fb68 022919be kernel32!ReadFile                   //从 COM 口读取数据
01 00e0fcd4 022a83cc dbgeng!ComPortRead+0x22e            //调试器引擎的 COM 函数
02 00e0fd00 022a7782 dbgeng!KdComConnection::Read+0x6c   //串行通信连接层
03 00e0fd20 022a91be dbgeng!KdConnection::ReadAll+0x22
04 00e0fd90 020dd497 dbgeng!KdComConnection::Synchronize+0x16e
05 00e0fdbc 020dd624 dbgeng!DbgKdTransport::Synchronize+0xc7
06 00e0fddc 020ddff3 dbgeng!DbgKdTransport::ReadPacketContents+0x84
07 00e0fe50 02133f5b dbgeng!DbgKdTransport::WaitForPacket+0x133
08 00e0feec 02133e38 dbgeng!ConnLiveKernelTargetInfo::WaitStateChange+0x8b
09 00e0ff10 020ceacf dbgeng!ConnLiveKernelTargetInfo::WaitForEvent+0x68
0a 00e0ff34 020cee9e dbgeng!WaitForAnyTarget+0x5f        //轮番等待所有调试目标
0b 00e0ff80 020cf110 dbgeng!RawWaitForEvent+0x2ae
0c 00e0ff98 0102aadf dbgeng!DebugClient::WaitForEvent+0xb0
0d 00e0ffb4 7c80b6a3 WinDBG!EngineLoop+0x13f             //调试会话循环
0e 00e0ffec 00000000 kernel32!BaseThreadStart+0x37       //会话线程的启动函数
```

可以看出，从栈帧#0e 到栈帧#0a 与用户态的情况（清单 29-2）完全相同，这得益于重构后的 WinDBG 使用了 C++的多态性，顶层可以用统一的代码来处理不同类型的调试目标。栈帧#07 是调用 DbgKdTransport 类的 WaitForPacket 方法等待来自目标系统的数据包，而后传输层调用连接层，即 KdComConnection 类的方法。KdComConnection 会按照

一定的时间间隔反复读取 COM 口，监视是否有数据来临。

29.8.3 执行命令

在进行内核调试时，大多数命令都需要位于目标系统的内核调试引擎（KD）的帮助，只有少数命令可以完全在本地执行。对于需要调试引擎协助的命令，调试器需要通过第 18 章介绍的内核调试协议与 KD 进行通信。清单 29-12 显示了 WinDBG 的调试会话线程在执行寄存器命令（r）时向 KD 发送数据包的过程。

清单 29-12 调试会话线程执行 r 命令时向 KD 发送请求的过程

```
0:001> kn 50
 # ChildEBP RetAddr
00 00d0e334 022a77d6 dbgeng!KdComConnection::Write          //写数据
01 00d0e358 022a8ee0 dbgeng!KdConnection::WriteAll+0x26     //连接层的基类方法
02 00d0e384 020dd8aa dbgeng!KdComConnection::WritePacketContents+0x80
03 00d0e3c8 020de6b5 dbgeng!DbgKdTransport::WritePacketContents+0x7a
04 00d0e448 020b250a dbgeng!DbgKdTransport::WriteDataPacket+0x1c5
05 00d0e46c 020deb14 dbgeng!DbgKdTransport::WritePacket+0x2a
06 00d0e4a8 020debf2 dbgeng!DbgKdTransport::SendReceivePacket+0x44
07 00d0e4d4 020d72df dbgeng!DbgKdTransport::SendReceiveManip+0x42//发送访问类 API
08 00d0e548 020f058e dbgeng!ConnLiveKernelTargetInfo::ReadControl+0x7f
09 00d0e574 0214e3d7 dbgeng!TargetInfo::GetTargetSpecialRegisters+0x3e
0a 00d0e590 021723e1 dbgeng!X86MachineInfo::KdGetContextState+0xd7
0b 00d0e5a8 02150f68 dbgeng!MachineInfo::GetContextState+0x121
0c 00d0e6e0 02236afd dbgeng!X86MachineInfo::OutputAll+0x28
0d 00d0e824 02199468 dbgeng!OutCurInfo+0x25d
0e 00d0e8bc 02186362 dbgeng!ParseRegCmd+0x1b8               //解析 r 命令
0f 00d0e8fc 02188348 dbgeng!WrapParseRegCmd+0x92            //r 命令的入口
10 00d0e9d8 021889a9 dbgeng!ProcessCommands+0x1278          //分发命令
11 00d0ea1c 020cbec9 dbgeng!ProcessCommandsAndCatch+0x49
12 00d0eeb4 020cc12a dbgeng!Execute+0x2b9
13 00d0eee4 01028553 dbgeng!DebugClient::ExecuteWide+0x6a   //执行命令的接口函数
14 00d0ef8c 01028a43 WinDBG!ProcessCommand+0x143
15 00d0ffa0 0102ad06 WinDBG!ProcessEngineCommands+0xa3
16 00d0ffb4 7c80b6a3 WinDBG!EngineLoop+0x366                //调试会话循环
17 00d0ffec 00000000 kernel32!BaseThreadStart+0x37
```

其中栈帧#0b 到#17 与调试 x86 架构的用户目标时是完全一样的。栈帧#08 是 ConnLiveKernelTargetInfo 类的 ReadControl 方法通过传输层向目标系统的 KD 发送请求。请求发送后，SendReceivePacket 方法调用 WaitForPacket 读取 KD 的回复包（清单 29-13）。

清单 29-13 WinDBG 的调试会话线程读取 KD 回复的过程（部分）

```
0:001> kn
 # ChildEBP RetAddr
00 00d0e35c 022a7782 dbgeng!KdComConnection::Read        //从 COM 口读取数据
01 00d0e37c 022a93f8 dbgeng!KdConnection::ReadAll+0x22
02 00d0e3a0 022a87ba dbgeng!KdComConnection::ReadPacketLeader+0x38//读导引字节
03 00d0e3c8 020dd6a8 dbgeng!KdComConnection::ReadPacketContents+0x6a
04 00d0e400 020ddff3 dbgeng!DbgKdTransport::ReadPacketContents+0x108
05 00d0e474 020deb29 dbgeng!DbgKdTransport::WaitForPacket+0x133
06 00d0e4a8 020debf2 dbgeng!DbgKdTransport::SendReceivePacket+0x59
07 00d0e4d4 020d72df dbgeng!DbgKdTransport::SendReceiveManip+0x42
08 00d0e548 020f058e dbgeng!ConnLiveKernelTargetInfo::ReadControl+0x7f
09 00d0e574 0214e3d7 dbgeng!TargetInfo::GetTargetSpecialRegisters+0x3e
```

因为与 KD 通信需要花费较多时间，所以 WinDBG 会将某些命令的执行结果保存起来，如果下次再执行同样的命令时，就不必再从 KD 那里读取。举例来说，当我们连续多次执行 kv 命令或者 r 命令时，会感觉到后面几次的速度明显加快，因为它们使用了缓存的数据。为了保证数据的一致性。每次退出命令状态时，WinDBG 会清除缓存的数据。

29.8.4 将调试目标中断到调试器

如果目标系统处于运行状态，那么可以通过中断（Break）命令将其中断到调试器。其工作过程如下：

在通过 WinDBG 的界面发出中断命令（菜单或者 Ctrl+Break）后，UI 线程会调用 DebugClient 类的 SetInterrupt 方法，并将第一个参数设置为 DEBUG_INTERRUPT_ ACTIVE（0）。这会导致 SetInterrupt 方法通过全局变量 g_Target 调用调试目标对象的 RequestBreakIn 方法。对于内核调试，g_Target 指向的是 ConnLiveKernelTargetInfo 类的实例，因此这个类的 RequestBreakIn 方法会被调用，这个方法的实现非常简单，只是将对象的一个成员变量（偏移为 0x1F7）设置为 1。这些操作发生在 UI 线程中。此时调试会话线程通常是在等待目标系统的调试事件，也就是在执行 DbgKdTransport 类的 ReadPacketContents 方法。这个方法在反复等待目标系统的通信包期间，每次循环时都会检查目标对象的 0x1F7 成员变量，如果发现其值等于 1，那么就调用 WriteBreakInPacket 方法向目标系统发送中断命令，其过程如清单 29-14 所示。

清单 29-14 内核调试会话线程向目标系统发送中断命令

```
0:001> kn
 # ChildEBP RetAddr                                          //调试会话线程
00 00d0fd78 020e04c3 dbgeng!KdComConnection::Write          //写 COM 口
01 00d0fd98 020de3c2 dbgeng!DbgKdTransport::Write+0x33      //写 Break 命令
02 00d0fdbc 020dd5ed dbgeng!DbgKdTransport::WriteBreakInPacket+0x32
03 00d0fddc 020ddff3 dbgeng!DbgKdTransport::ReadPacketContents+0x4d
04 00d0fe50 02133f5b dbgeng!DbgKdTransport::WaitForPacket+0x133
05 00d0feec 02133e38 dbgeng!ConnLiveKernelTargetInfo::WaitStateChange+0x8b
06 00d0ff10 020ceacf dbgeng!ConnLiveKernelTargetInfo::WaitForEvent+0x68
07 00d0ff34 020cee9e dbgeng!WaitForAnyTarget+0x5f
08 00d0ff80 020cf110 dbgeng!RawWaitForEvent+0x2ae          //等待调试事件
09 00d0ff98 0102aadf dbgeng!DebugClient::WaitForEvent+0xb0
0a 00d0ffb4 7c80b6a3 WinDBG!EngineLoop+0x13f               //调试会话循环
0b 00d0ffec 00000000 kernel32!BaseThreadStart+0x37
```

如 18 章所介绍的，目标系统在每次更新系统时间（KeUpdateSystemTime）时会调用内核调试引擎的 KdPollBreakIn 函数，检查是否有中断命令，如果有，则准备中断到内核调试器。

29.8.5 本地内核调试

WinDBG 将本地内核调试看作是双机内核调试的特例，调试引擎中的 LocalLiveKernelTargetInfo 类用来描述本地内核目标。因此当建立本地内核调试会话时，LiveKernelInitialize 方法创建的是 LocalLiveKernelTargetInfo 类的实例。

因为调试器与调试目标在同一个系统中，所以本地内核调试的通信过程比双机调试简单得多。事实上，调试器就是通过 NtSystemDebugControl 内核服务来与调试目标进行通信的。清单 29-15 显示了 WinDBG 的调试会话线程执行显示内存命令的过程。

清单 29-15 本地内核调试时执行内存显示命令的过程

```
0:002> kn
 # ChildEBP RetAddr
00 00e0e128 0222395f ntdll!ZwSystemDebugControl+0xa        //调用内核服务
01 00e0e168 020d821f dbgeng!LocalLiveKernelTargetInfo::DebugControl+0xaf
02 00e0e1a8 0217a9b2 dbgeng!LocalLiveKernelTargetInfo::ReadVirtual+0xbf
03 00e0e470 0217ae72 dbgeng!DumpValues::Dump+0x552
04 00e0e48c 0217d26d dbgeng!DumpValues::ParseAndDump+0x72 //解析命令
05 00e0e900 02187883 dbgeng!ParseDumpCommand+0xa0d         //内存显示命令的入口
06 00e0e9d8 021889a9 dbgeng!ProcessCommands+0x7b3          //分发命令
07 00e0ea1c 020cbec9 dbgeng!ProcessCommandsAndCatch+0x49
```

```
08 00e0eeb4 020cc12a dbgeng!Execute+0x2b9
09 00e0eee4 01028553 dbgeng!DebugClient::ExecuteWide+0x6a //执行命令的接口函数
0a 00e0ef8c 01028a43 WinDBG!ProcessCommand+0x143
0b 00e0ffa0 0102ad06 WinDBG!ProcessEngineCommands+0xa3
0c 00e0ffb4 7c80b6a3 WinDBG!EngineLoop+0x366              //调试会话循环
0d 00e0ffec 00000000 kernel32!BaseThreadStart+0x37
```

可以看到，03 号到 0d 号栈帧与双机内核调试是一样的。栈帧 02 是 Dump 方法调用的 LocalLiveKernelTargetInfo 类的 ReadVirtual 方法，后者再调用 DebugControl 方法，这里仍然使用了 C++的多态性。DebugControl 方法只是对系统服务 ZwSystemDebug- Control 的封装。ZwSystemDebugControl 通过系统调用机制调用内核中的 NtSystemDebugControl 方法。

29.9 远程用户态调试

WinDBG 工具包提供了多种方式进行远程调试，本节将讨论通过进程服务器（Process Server）来进行远程用户态调试的基本原理和实现方法。

29.9.1 基本模型

图 29-20 显示了通过进程服务器调试位于另一个系统中的用户态程序时的基本模型。左侧的系统是被调试程序所运行的系统，称为目标系统。因为它是调试服务的提供者，所以目标系统有时也被称为服务器系统。相对而言，右侧运行调试器的系统称为客户系统（Client）。

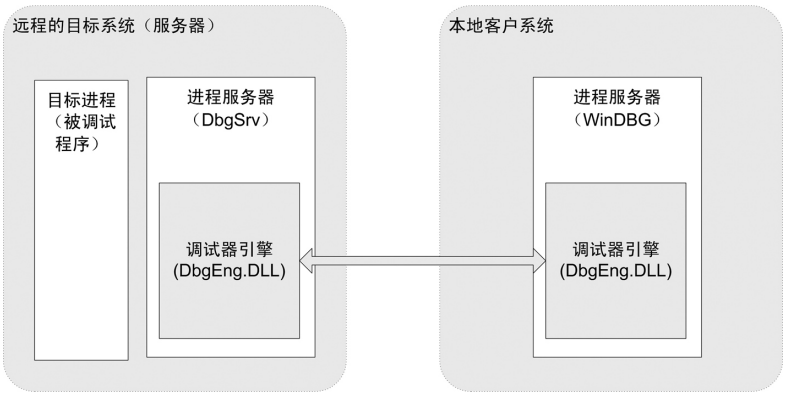


图 29-20 通过进程服务器（DbgSrv）进行远程用户态调试

目标系统需要运行进程服务器程序 DbgSrv.exe，它位于 WinDBG 的程序目录中。可以在目标系统中安装完整的 WinDBG 工具包，也可以直接将 WinDBG 的程序目录复制到目标系统。

目标系统与客户系统之间的通信方式可以有以下几种：命名管道（NPIPE）、TCP、COM 口、安全的管道（Secure Pipe，简称 SPIPE）和 SSL（Secure Sockets Layer）。其中 SPIPE 和 SSL 需要两个系统中的操作系统都至少是 Windows 2000。

29.9.2 进程服务器

在目标系统中，启动一个命令行窗口，切换到 DbgSrv.exe 文件所在的目录，然后键入如下命令：

```
C:\WinDBG>dbgsrv -t tcp:port=1022 -c notepad.exe
```

这条命令的含义是启动进程服务器，让其使用 TCP 协议监听 1025 号端口，同时创建

notepad.exe 进程（要调试的程序）。

此时在目标系统中运行一个用于分析 DbgSrv 的调试器，将其附加到 DbgSrv 进程，然后将 DbgSrv 中断到调试器。可以发现除了用于中断到调试器的 2 号线程外，DbgSrv 还有两个线程。0 号线程是 UI 线程，也是这个进程的初始线程，1 号线程是监听调试器连接请求的监听线程。

事实上，DbgSrv 启动后，它的主函数（main）在分析命令行参数后便调用 DebugClient 类的 StartProcessServerWide 方法启动进程服务器，这个方法的原型如下：

```
HRESULT IDebugClient5::StartProcessServerWide(
   IN ULONG  Flags, IN PCWSTR  Options, IN PVOID  Reserved);
```

其中 Flags 参数用于指定调试目标的类型，必须为 DEBUG_CLASS_USER_WINDOWS(2)，Options 用来指定与调试器的连接字符串，它的值就是命令行中-t 开关后的参数，观察其值为：

```
0:000> du 007a1118
007a1118  "tcp:port=1025"
```

StartProcessServerWide 函数会调用 DbgRpcCreateServer，后者调用 DbgRpcInitializeTransport 创建传输层对象实例，因为我们指定的是 TCP 连接，所以创建的是 DbgRpcTcpTransport 类的实例。而后 DbgRpcCreateServer 函数调用新创建的传输层对象的 CreateServer 方法。CreateServer 再调用 CreateServer- Socket 方法，后者调用操作系统的 Socket API WSASocket 创建通信套接字。在创建好通信套接字之后，DbgRpcCreateServer 方法调用 CreateThread API 来创建一个新的线程，线程的函数为 dbgeng!DbgRpcServerThread。这个线程用来监听来自客户机器的连接请求。

StartProcessServerWide 方法返回后，DbgSrv 的主函数根据-c 参数指定的命令行来创建新的进程，但是并没有与其建立调试关系。在以上任务完成后，0 号线程的任务基本完成，调用 WaitForProcessServerEnd 方法等待结束命令。

29.9.3 连接进程服务器

在客户系统中，使用如下命令行启动 WinDBG：

```
c:\WinDBG>WinDBG -premote tcp:server=<DbgSrv 进程所在的机器名>,port=1022
```

然后选择 File 菜单的 Attach to a process…命令，这时目标系统的 DbgSrv 会命中我们预先设置的 CreateThread 断点。这是因为，WinDBG 从命令行参数中知道是远程调试，所以需要从远程获得供调试的进程列表，于是开始与目标系统的进程服务器建立连接。在目标机器上，DbgSrv 的监听线程接收到连接请求后，会调用 CreateThread API 创建一个新的工作线程来与客户机上的 WinDBG 通信，我们称其为服务线程。这个新线程的入口函数为 DBGENG 模块中的 DbgRpcClientThread 函数。处理完一个连接后，监听线程再次调用 AcceptConnection 方法等待新的连接请求。当再有新的连接请求（客户）时，DbgSrv 的监听线程会再创建一个新的服务线程。也就是说，DbgSrv 会为每个客户创建一个不同的服务线程。因此，一个 DbgSrv 进程可以为多个 WinDBG 服务。

29.9.4 服务循环

WinDBG 使用 RPC（Remote Procedure Call）机制来调用服务器进程中的调试器引擎函数。下面我们简要讨论其过程。服务线程启动后，便进入一个工作循环等待来自客户端的数据，如清单 29-16 所示。

清单 29-16　DbgSrv 的服务线程在等待客户数据

```
0:002> kn
 # ChildEBP RetAddr
00 00a7fdd8 7c90e9c0 ntdll!KiFastSystemCallRet            //调用系统服务
01 00a7fddc 7c8025cb ntdll!ZwWaitForSingleObject+0xc      //残根函数
02 00a7fe40 7c802532 kernel32!WaitForSingleObjectEx+0xa8
03 00a7fe54 7c831568 kernel32!WaitForSingleObject+0x12    //等待同步对象
04 00a7fe68 71a6b083 kernel32!GetOverlappedResult+0x30
05 00a7feac 71ac0d59 mswsock!WSPGetOverlappedResult+0x62
06 00a7fed8 0228ac57 WS2_32!WSAGetOverlappedResult+0x56
07 00a7ff0c 02286a65 dbgeng!DbgRpcTcpTransport::Read+0xa7//传输层的读数据方法
08 00a7ff60 022882d8 dbgeng!DbgRpcReceiveCalls+0x55       //接收远程调用
09 00a7ffb4 7c80b6a3 dbgeng!DbgRpcClientThread+0xa8
0a 00a7ffec 00000000 kernel32!BaseThreadStart+0x37        //线程的启动函数
```

当服务线程收到一个完整的 RPC 数据包后，它先调用 DbgRpcGetStub 函数读取要调用的函数指针，这个函数指针通常是调试器引擎中的 SFN_IXXX 函数。在确认读取到的函数指针不为空后，服务线程便就调用这个函数，然后再把函数的执行结果发送给客户端的 WinDBG。完成一次服务后，服务线程再调用 DbgRpcReceiveCalls 来等待新的调用，如此循环直到结束。

29.9.5　建立调试会话

当我们在 WinDBG 中选择 Notepad 进程并按确定后，和调试本地的应用程序一样，WinDBG 会创建一个新的调试会话线程并调用 StartSession 函数，其详细过程如清单 29-17 所示。

清单 29-17　附加到远程的应用程序

```
0:001> kn
 # ChildEBP RetAddr
00 00f0fad8 7c90e9c0 ntdll!KiFastSystemCallRet              //系统调用
01 00f0fadc 7c8025cb ntdll!ZwWaitForSingleObject+0xc
02 00f0fb40 7c802532 kernel32!WaitForSingleObjectEx+0xa8
03 00f0fb54 7c831568 kernel32!WaitForSingleObject+0x12      //等待同步对象
04 00f0fb68 71a6b083 kernel32!GetOverlappedResult+0x30
05 00f0fbac 71ac0d59 mswsock!WSPGetOverlappedResult+0x62
06 00f0fbd8 0228ac57 WS2_32!WSAGetOverlappedResult+0x56
07 00f0fc0c 02286a65 dbgeng!DbgRpcTcpTransport::Read+0xa7   //传输层的读数据方法
08 00f0fc60 02286f8e dbgeng!DbgRpcReceiveCalls+0x55
09 00f0fc80 0229e83d dbgeng!DbgRpcConnection::SendReceive+0x10e //发送并接收应答
0a 00f0fccc 020f3a31 dbgeng!ProxyIUserDebugServicesN::AttachProcess+0x11d
0b 00f0fcfc 020c1344 dbgeng!LiveUserTargetInfo::StartAttachProcess+0xd1
0c 00f0fd40 0102a385 dbgeng!DebugClient::CreateProcessAndAttach2Wide+0x104
0d 00f0ffa4 0102a9bb WinDBG!StartSession+0x445              //开始调试会话
0e 00f0ffb4 7c80b6a3 WinDBG!EngineLoop+0x1b                 //调试会话循环
0f 00f0ffec 00000000 kernel32!BaseThreadStart+0x37
```

在上面的清单中，从栈帧#0b 到#0f 与调试本地对的应用程序是完全一样的。差异是从栈帧#0a 开始的，在调试本地的应用程序时，LiveUserTargetInfo 用的是 LiveUserDebugServices 类，而这里 LiveUserTargetInfo 用的是 ProxyIUserDebugServicesN 类。ProxyIUserDebugServicesN 类与 LiveUserDebugServices 具有相同的接口，所以 LiveUserTargetInfo 类可以不关心二者的差异。

ProxyIUserDebugServicesN 类将 AttachProcess 调用通过 DbgRpcConnection 发送给远程的进程服务器，然后等待它的回复（栈帧 0～8）。

清单 29-18 显示的是目标机器上的服务线程收到调用 AttachProcess 函数的请求后，在 DbgSrv 进程中执行这个请求的过程。

清单 29-18　服务线程执行附加动作时的函数调用过程

```
0:002> kn
 # ChildEBP RetAddr
00 00a7fe9c 0229a146 ntdll!NtDebugActiveProcess          //系统调用
01 00a7feb8 0229a2b0 dbgeng!LiveUserDebugServices::CreateDebugActiveProcess…
02 00a7fed4 022a47ee dbgeng!LiveUserDebugServices::AttachProcess+0xb0
03 00a7ff04 02286c27 dbgeng!SFN_IUserDebugServicesN_AttachProcess+0xbe
04 00a7ff60 022882d8 dbgeng!DbgRpcReceiveCalls+0x217     //接收调用
05 00a7ffb4 7c80b6a3 dbgeng!DbgRpcClientThread+0xa8      //服务线程
06 00a7ffec 00000000 kernel32!BaseThreadStart+0x37
```

如果将清单 29-18 的 0 到 2 号栈帧放在清单 29-17 的#0b 到#0f 号栈帧之上，那么合并起来的函数栈调用序列恰好与本地调试时的情况相同。因此可以把这样的远程调试功能看作是利用 RPC 机制将调试器引擎的功能分布在两台机器上，而分割的边界是在调试服务层，即 IUserDebugServices 接口。远程调试时，客户机上使用 ProxyIUserDebugServicesN 类远程调用服务进程中的 SFN_IUserDebugServicesN_XXX 系列函数，后者再调用真正的调试服务（LiveUserDebugServices）。这使得远程调试时，调试目标类（LiveUserTargetInfo）可以使用统一的方式来处理本地调试和远程调试。

类似的，等待调试事件和执行用户输入的调试命令的过程也是利用 RPC 机制分布在两台机器上，不再赘述。

29.9.6 比较

在经典调试架构中，使用传输层来隔离本地调试和远程调试的差异性，即使是本地调试也要使用一个简单的本地调试传输层 TLLoc.DLL。在重构后的调试器引擎架构中，使用统一的 IDebugService 接口来统一本地调试和远程调试。这样的好处是本地调试时不再需要形式上的传输层。经典调试模型设计时没有考虑使用 C++的多态机制，而后者设计时就想到了要发挥 C++语言和 COM 接口等技术，这是导致以上差异的原因。

本节介绍了通过进程服务器进行远程调试的实现方法。与 DbgSrv.exe 相对应，WinDBG 工具包中还有一个名为 KdSrv.exe 的工具，KdSrv 用于支持远程内核调试，它的工作原理与 DbgSrv 非常类似，本书不再详细讨论。

补编内容 8 WMI

补编说明：

这一章本来是《软件调试》第 3 篇中的一章，是操作系统的调试支持中的一部分。

写作这一章的原因有三个，一是 WMI 体现了软件的可配置性和可管理性，它是软件行业中标准化工作做的最好的一个典型。而可配置行和可管理性都与软件调试有着密切的关系。第二个原因是 WMI 作为系统的一种重要机制，遍布在系统的各个部分，内核、驱动、服务、应用程序、日志文件、管理终端等等，因此，理解这一内容对于了解整个系统，提高综合能力很有用。第三个原因是，WMI 各个部件之间的协作模型是设计的很不错的软件架构，使用了 RPC 机制，调试 RPC 是用户态调试中较难的任务，这一章中以背景知识的方式介绍了一部分 RPC 的基础知识。

这一章是唯一被整章删除的内容，也是最早删除的内容，删除的原因是担心被质疑跑题。

因为这部分内容被删除的较早，所以没有做过仔细的审查，还处于草稿的状态。

CHAPTER

第 31 章

WMI

Windows 是个庞大的系统，如何了结系统中各个部件的运行状况并对它们进行管理和维护是个重要而复杂的问题。如果每个部件都提供一个管理程序，那么不仅会导致很多的重复开发工作，而且也会导致用户要学习各种不同的程序和界面。更好的做法是操纵系统实现并提供一套统一的机制和框架，其它部件只要按照一定的规范实现与自身逻辑密切相关的部分。WMI（Windows Management Instrumentation）就是对这一套机制的统称。

WMI 提供了一套标准化的机制来管理本地及远程的 Windows 系统，包括操作系统自身的各个部件以及系统中运行的各种应用软件，只要它们提供了 WMI 支持。WMI 最早出现在 NT4 的 SP4 中，并成为其后的所有 Windows 操作系统的必不可少的一个部分。在今天的 Windows 系统中，很容易就可以看到 WMI 的身影，比如计算机管理（Computer Management），事件查看器，系统服务管理（Services Console）等。

WMI 是个很大的话题，全面的介绍可能需要一本书的篇幅，这显然超出了本书的范围。所以我们的策略还是从调试角度来了解 WMI 的要点，以达到如下两个目的：

- 熟悉现有的 WMI 设施，以便可以把它们应用到实际问题中，辅助调试。
- 在我们的软件产品中，加入 WMI 支持，利用 WMI 增强产品的可调试性。

我们将在第 27 章介绍如何如何在软件开发中使用 WMI，本章我们将着重介绍 WMI 的架构和工作原理。

31.1 WBEM 简介

WMI 是基于 DMTF（Distributed Management Task Force）组织制定的 WBEM 系列标准实现的。DMTF 是一家旨在建立和推行计算机系统管理有关的标准国际组织，其成员有包括英特尔、微软 Dell、IBM 等在内的众多著名企业。WBEM 的全称是 Web Based Enterprise Management（基于网络的企业管理）。下面我们先介绍一些 WBEM 有关的背景知识。

WBEM 起始于 1996 年，其目的是发起制定一套标准来统一企业内计算资源的管理方法，以减少管理的复杂性和费用，降低总体拥有成本（TCO）。相对于用于网络管理的 SNMP（Simple Network Management Protocol ）和用于桌面系统管理的 DMI（Desktop Management Interface）的标准，WBEM 的宗旨是提供一个单一的，可共享的模型（a single, shared model）来收集信息和实施管理。因此，严格说来 WBEM 本身是一个倡议，但是今天也经常把因为该倡议而制定的一系列标准泛称为 WBEM 标准。

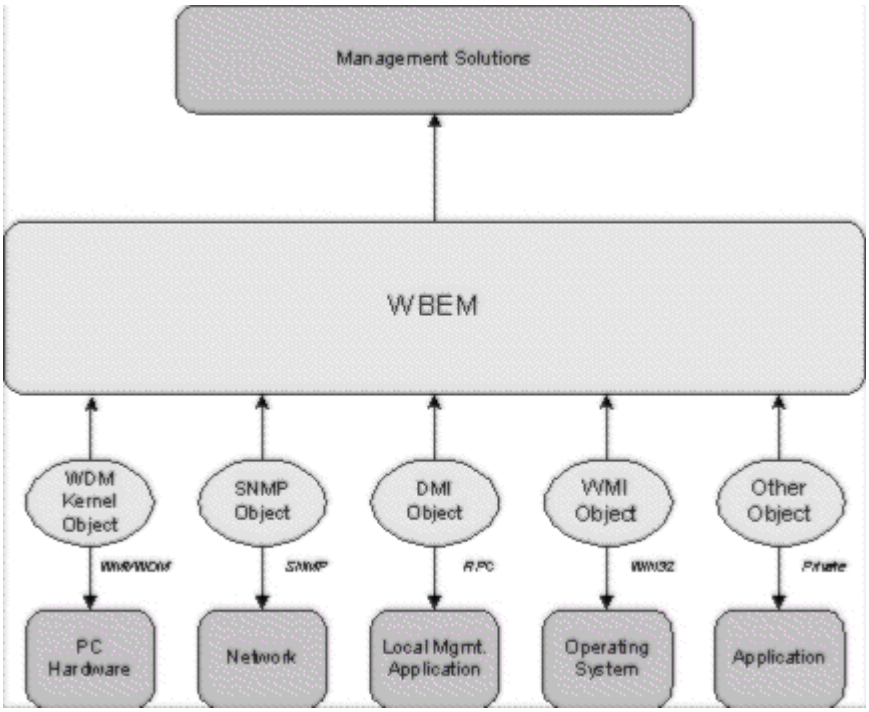


图 31-1　WBEM 旨在通过一个单一的可共享的模型来交流信息和实施管理

WBEM 主要由以下几个部分组成：

- 公共信息模型 (Common Information Model，简称 CIM)规约 ：定义了实现企业网络管理（WBEM）的基本原则和方法。其核心是如何使用 CIM 为被管理对象（managed objects，简称受管对象）建模。CIM 是一种语言无关的面向对象编程模型，它使用类来描述管理对象。与 C++的类类似，CIM 的类也可以包含属性和行为，可以相互继承。受管对象格式 (Managed Object Format) 语言是表达 CIM 模型的程序语言之一。MOF 是基于 IDL（Interface Definition Language）的，熟悉 COM 编程的读者应该知道 IDL 是描述 COM 接口的一种主要方法。MOF 有它独有的语法，但使用 DMTF 提供的 DTD（Document Type Definition）可将 MOF 文件转化为 XML 文件。使用 CIM 和 MOF，我们便可以使用面向对象设计方法来对管理对象进行描述和建模。
- CIM Schema：作为 WBEM 模型库的一个部分，DMTF 建立了核心模型（Core Model）和公共模型（Common Model）用于描述具有普遍意义的概念和对象，统称为 CIM Schema。其它开发者可以使用这些模型来提高设计和开发速度。
- CIM 查询语言：用于从基于 CIM 建设的管理系统中提取数据的查询（query）语言。
- CIM 的 XML 表示（Representation of CIM in XML）：如何使用 XML 表示 SIM 模型。

从 DMTF 网站（http://www.dmtf.org）可以下载以上标准和模型的最新版本。

31.2　CIM 和 MOF

CIM（Common Information Model）是一种层次化的面向对象的建模方法，是 WBEM（WMI）中定义和描述受管对象的基本标准。CIM 既可以描述物理的对象，也可以描述逻辑对象。

《CIM 基础规约》（COMMON INFORMATION MODEL (CIM) INFRASTRUCTURE SPECIFICATION）是 CIM 标准的根本文件，也是了解 CIM 标准的最好资料。笔者写作本内容时，该文档的最新版本是 2.3。下面就以该版本为例，介绍 CIM 的核心内容。

31.2.1 类和 Schema

CIM 将被管理环境看作是由一系列相互联系的系统组成的，每个系统又包含很多个分立的对象。CIM 使用类来描述物理或者逻辑对象。

CIM 将对模型的正式定义叫做 Schema。每个 Schema 通常定义了模型内的一系列类和类之间的关系。可以把 Schema 理解为类库，人们可以使用建立好的 Schema 来设计新的模型。因此，CIM 文档将 Schema 称为是用来建造管理平台的积木（building block）。

Schema 是各种建模技术中很常用的一个术语，其基本含义就是对问题域的模型描述，比如数据库设计中的 Schema，和 XML Schema 等。

CIM 规定所有完整的类名应该是以 Schema 名开始，并使用下划线与类名分隔。比如，CIM_ManagedSystemElement（CIM 的基类），CIM_ComputerSystem，CIM_SystemComponent 是 CIM Schema 中的几个类。

根据所描述对象的普遍性，CIM 将 Schema 分为如下三个层次：

- CIM Core Schema：适用于所有管理域。
- CIM Common Schema：适用于特定管理域，不依赖于特定的技术和实现。
- Extension Schema：适用于特定技术。与特定的环境（如操作系统）相关。

CIM Core Schema 和 CIM Common Schema 被统称为 CIM Schema。从设计和开发的角度来看，可以把 CIM Schema 理解为 CIM 标准已经定义好的类库。在设计 Extension Schema 时可以从这些定义好的类派生新的类，以提高建模的速度。

从 DMTF 网站，可以下载包含所有 CIM Schema 详细定义的文件。随着技术的发展，CIM Schema 的定义也在不断扩充。目前的版本包含的 Schema 有 CIM_Core，CIM_Application，CIM_Database，CIM_Device，CIM_Event，CIM_Interop，CIM_IPsecPolicy，CIM_Metrics，CIM_Network，CIM_Physical，CIM_Policy，CIM_Security，CIM_Support，CIM_System 和 CIM_User。这些 Schema 中的类最终都是以 CIM 作为 Schema 名的。

CIM 中定义了一套以 UML（Unified Modeling Language）规范为基础的图形语言来定义 CIM 模型，称为 Meta Schema。Meta Schema 的绝大多数表达方法都与 UML 相同。比如，用一个包含类名的矩形来表示类，矩形内可以包含类的属性和方法，使用不同样式的连线表示类之间的关系。但略微不同的是，Meta Schema 还使用线的颜色来方便阅读，表示关联关系的的线通常使用红颜色，表示继承关系的线用蓝颜色，表示聚合关系（aggregation）的线用绿色，UML 中并没有这些约定。

图 31-2 显示了 CIM 核心模型（Core Model Schema）中的几个重要类的 Meta Schema

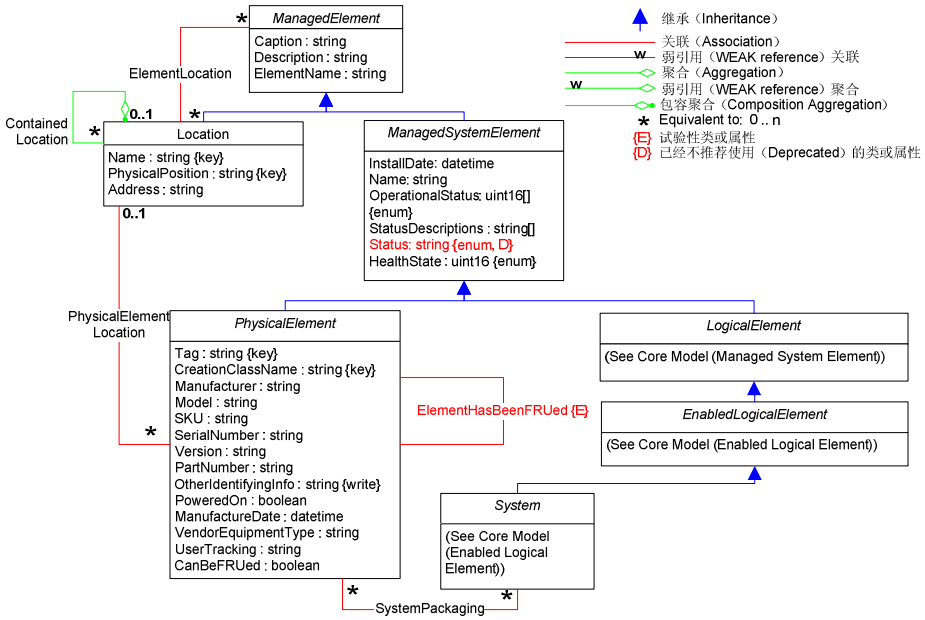


图 31-2　CIM 核心模型中的几个重要类的 Meta Schema 表示

表示。因为该图的所有类都是在 CIM Schema 内，所以类名中省略了 CIM 字样。图中最上面的 ManagedElement（CIM_ManagedElement）是 CIM 中的最高基类。它的一个极其重要的派生类是 ManagedSystemElement 类，所以的系统要素都是从这个类派生出的，在 WMI 的实现中，ManagedSystemElement 是最高基类。下一层次的两个重要类便是 PhysicalElement 和 LogcalElement，分别用来描述物理元素和逻辑元素的基本属性（这两个类没有方法）。

31.2.2　MOF

MOF(Managed Object Format) 是使用文字形式来描述 CIM 模型的程序语言。MOF 文件的主要内容是对类、属性、方法、和实例声明的文字表达。

学习 MOF 的一种简单方法就是阅读 CIM Schema 中已经定义好的各个类，可以从 DMTF 网站（http://www.dmtf.org/standards/cim/）下载包含所有类定义的 MOF 文件压缩包。

例如，打开 CIM_ManagedSystemElement.mof（解压后的 Core 目录下）文件，就可以看到使用 MOF 定义的 ManagedSystemElement 类（清单 31-1，为了节约篇幅，笔者删除了部分描述和空行）。

清单 31-1　使用 MOF 语法定义的 ManagedSystemElement 类

```
// ==================================================================
//  CIM_ManagedSystemElement
// ==================================================================
   [Abstract, Version ( "2.8.0" ), Description (
      "CIM_ManagedSystemElement is the base class for the System "
      "Element hierarchy. Any distinguishable component of a System "
      "is a candidate for inclusion in this class. [删除多行]")]
class CIM_ManagedSystemElement : CIM_ManagedElement {
     [Description (
        "A datetime value indicating when the object was installed. "
        "A lack of a value does not indicate that the object is not "
        "installed."),
      MappingStrings { "MIF.DMTF|ComponentID|001.5" }]
   datetime InstallDate;
     [Description (
        "The Name property defines the label by which the object is "
        "known. When subclassed, the Name property can be overridden "
```

```
        "to be a Key property."),
      MaxLen ( 1024 )]
   string Name;
   [删除多行]
};
```

观察上面的清单，熟悉面向对象编程的读者可以很容易看懂其中的绝大部分内容。比如类和继承关系声明都与 C++完全一样。

```
class CIM_ManagedSystemElement : CIM_ManagedElement {
```

对于某些看不懂的内容只要查阅 CIM 文档（CIM Infrastructure Specification）就可以了，比如 InstallDate 属性上面的描述中的 MappingStrings { "MIF.DMTF|ComponentID|001.5"}的含义。MappingStrings 是 MOF 中的一种修饰符（qualifier）。MOF 中可以使用修饰符对类和属性进行修饰或限定。MappingStrings 修饰符的作用是将 CIM 中的属性与 MIF（Management Information Format）中的属性关联起来。

31.2.3 WMI CIM Studio

WMI CIM Studio 是微软的 WMI Tools 工具包中的一个工具，通过它可以浏览系统中的 CIM 类和对象并执行各种操作，是学习 CIM 和解决 WMI 有关问题的一个重要助手。

WMI 工具曾经是 WMI SDK 的一部分，但现在 WMI SDK 被集成到 Platform SDK 中。WMI 工具可以单独从微软的网站下载。你只要在搜索 WMI Administrative Tools 便可以找到下载链接，然后下载一个名为 WMITools.exe 的安装文件。安装后，开始菜单中会被加入一个名为 WMI Tools 的程序组。其中包含了如下几个工具：

- WMI CIM Studio：观察编辑 CIM 库中的类、属性、修饰符和实例；运行选中的方法；产生和编译 MOF 文件。
- WMI Object Browser（对象浏览器）：观察 CIM 对象，编辑属性值和修饰符（qualifiers），运行类的方法。
- WMI Event Registration Tool：WMI 事件注册工具，配置事件消耗器，创建或观察事件消耗器实例。
- WMI Event Viewer：WMI 事件观察器，显示所有注册消耗器（consumer）实例的事件。

除了 WMI Event Viewer 外，另外三个工具都是以 OCX 控件形式在浏览器中运行的，如果浏览器禁止了 OCX 控件运行，那么必须选择 Allow Blocked Content，它们才能工作。

下面我们先来看一下 CIM Studio，启动后，会出现图 31-3 所示的选择要连接到的命名空间对话框。

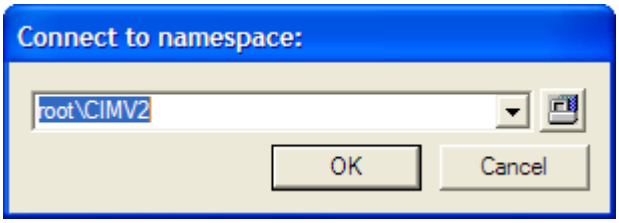


图 31-3 CIM Studio 的连接对话框

命名空间（namespace）定义了对象的生存范围（scope）和可见范围，是 CIM 中组织类和管理信息的一个逻辑单位。如果使用数据库的术语来理解，那么一个命名空间对应于一个数据库（类好似表，属性好似字段）。一个 WBEM 系统中可以有多个命名空间。表 31-1 列出了典型的 Windows XP 系统中存在的命名空间和简单描述。

表 31-1　Windows XP 系统中常见的命名空间

命名空间（Namespace）	描述
Root	根
CIMV2	CIM
CIMV2\Applications	某些应用程序（如 IE）
Default	默认的命名空间
Directory\LDAP	Lightweight Directory Access Protocol
Microsoft\HomeNet	家庭网络
Microsoft\SqlServer	SQL Server 数据库服务器
MSAPPS11	Office 程序
Policy	系统策略有关的数据
RSOP	安全有关的管理信息
WMI	WDM 提供器
<namespace>\MS_XXX	语言（locale）有关的信息，例如 MS_409 是英语有关的信息

可以在连接对话框的编辑框中输入要连接到的命名空间，也可以点击下拉框旁边的按钮浏览要连接的命名空间（图 31-4）。

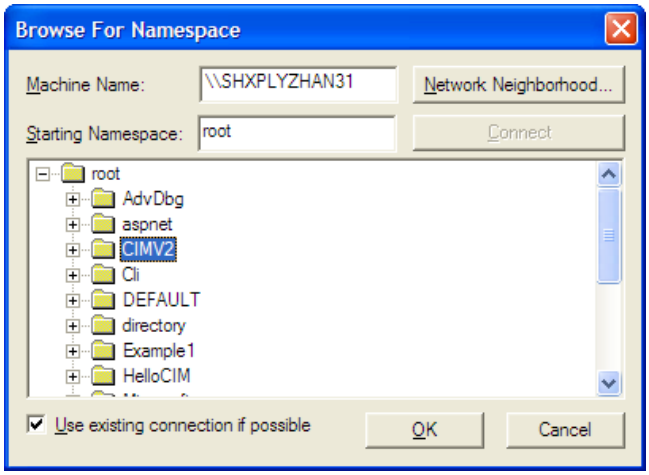


图 31-4　浏览命名空间

连接对话框的默认值是 root\CIMV2，这是 CIM 类所在的命名空间，也包含了微软的设计的从 CIM 类派生出的一些类（以 Win32 或 MSFT 为 Schema 名）。图 31-5 显示了 CIM Studio 的主界面。

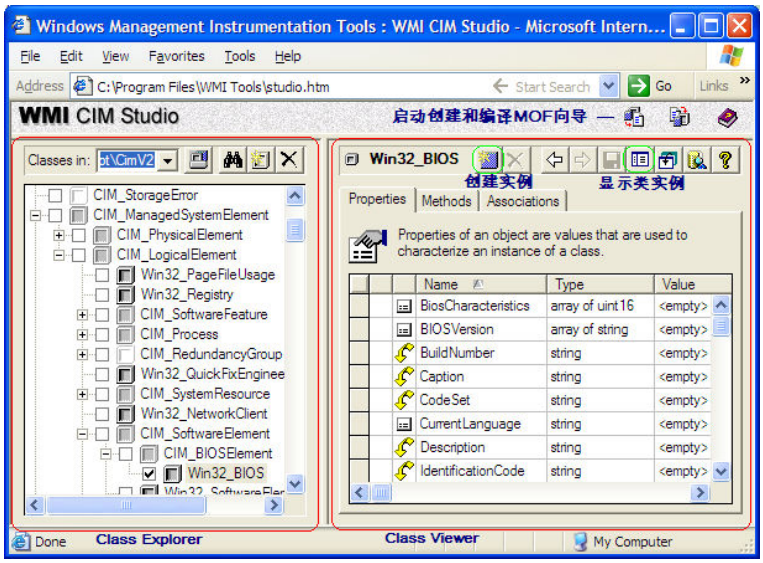


图 31-5　CIM Studio

左侧窗口区域被称为类浏览器（Class Explorer），用来查找各个类，树控件很好的表现了各个类之间的继承关系。比如图中选中的 Win32_BIOS 类是从一个各个类派生而来的：

CIM_SoftwareElement:CIM_LogicalElement:CIM_ManagedSystemElement

可以通过快捷菜单或者命名空间名称旁的按钮来查找或者删除类。

右侧窗口区域被称为类观察器（Class Viewer），用来显示左侧选中类的详细信息，包括属性、方法、关联。通过按钮区域的按钮（从左至右）可以创建类的实例、删除类实例、左右切换试图、保存编辑内容、显示类的所有实例、设置显示选项、执行 WQL（WMI Query Language，稍候介绍）查询、或者显示当前类的描述信息。

类观察器上面的三个按钮可以分别用来启动创建 MOF 向导、编译 MOF 向导和 CIM Studio 的帮助文件。

31.2.4　定义自己的类

尽管 CIM Schema 中已经可以数百个类，但是对于某个具体的管理任务，大多时候还是需要根据具体对象和问题定义自己的类和 Schema。下面便通过一个简单的例子来演示如何使用 MOF 语言设计新的类。

优盘（USB Disk）是近两三年流行起来的一种常见移动存储设备。在 CIM 中定义了 CIM_USBDevice 类来描述 USB 设备，但是没有设计 USB Disk 类。 于是我们很自然的想到可以从 CIM_USBDevice 类派生出一个新的类来描述优盘设备。清单 31-2 所示的 MOF 代码实现了这一设想。

清单 31-2　优盘（USB Disk）设备类

```
1   // AdvDbg_UsbDisk.MOF
2   // A sample used to demonstrate inheritance from CIM class.
3
4   #pragma classflags("forceupdate")
5   #pragma namespace ("\\\\.\\Root\\CIMV2")
6   class AdvDbg_UsbDisk:CIM_USBDevice
7   {
8       [write (true), Description("The OS this disk can boot to."): ToSubClass]
9       string BootableOS;
10      [read(true), Description("Capacity of this disk in bytes."): ToSubClass ]
11      uint32 Capacity;
12      [read, key, MaxLen(256), Override("DeviceID"): ToSubClass]
13      string DeviceID;
14      [Description("Format the disk, all data will be lost.")]
15      boolean Format([in] boolean quick);
16  };
17  instance of AdvDbg_UsbDisk
18  {
19      DeviceId = "USB_ADVDBG2006";
20      Name = "USB Disk for AdvDbg";
21      Caption = "USB Disk";
22      BootableOS = "DOS70";
23      Capacity = 1288888;
24  };
```

下面对上面代码中可能有些难以理解的地方作些说明。首先看第 4 行，与 C/C++程序一样，pragma 代表这一行是通知编译器的编译器指令（compiler directive）。classflags("forceupdate")的作用是如果强制更新这个类的定义，即使存在有冲突的子类。第 5 行是用来指定命名空间。

第 6 行声明了一个新的类 AdvDbg_UsbDisk，“:CIM_USBDevice”表示继承

CIM_USBDevice 类。7 到 16 行是类定义，9、11 和 13 行各定义了一个属性，15 行定义了一个方法，8、10、12 和 14 行是修饰符（Property Qualifier）行。在 CIM 中，在定义类以及类的属性和方法时都可以使用修饰符。修饰符以一对方括号包围起来，如果有多个，那么以逗号相分隔。每个修饰符通常包括名称和取值两个部分，取值通常放在括号中，如果没有括号则表示使用默认值（如 12 行中的 read）。以第 8 行为例，其中包含了两个修饰符：

- write(true)表示该属性可以被消耗器（consumer）所修改。
- Description("The OS this disk can boot to.") : ToSubClass 是更常见的一种修饰符，其值是对属性的描述和说明，描述信息也会作为类定义的一部分编译并存储到 CIM 库（CIM Repository）中。': ToSubClass'被称为 Flavor，其含义是该修饰符会自动被应用到子类，如果加上了 Restricted Flavor，那么该修饰符仅在当前类有效。除了 ToSubClass 和 Restricted，CIM 定义的 Flavor 还有 EnableOverride（允许该修饰符被子类覆盖），DisableOverride（禁止该修饰符被子类覆盖）和 Translatable（该修饰符的值是否可以用多种语言（locale）表示）。因为每个修饰符默认的 Flavor 中就包含 ToSubClass，所以这一行是否包含': ToSubClass'是等价的。

下面看一下第 12 行，其中包含了 4 个修饰符，key 的含义是将所修饰的属性作为键（key）属性，这好比是数据表中的键字段。CIM 使用 Key 属性的值来判断实例的等价性。Override("DeviceID")表示覆盖基类中的 DeviceID 属性，在基类中 DeviceID 不是键属性，这是重新定义的原因。MaxLen(256)表示属性的最大长度是 256 个字符。

第 15 行定义了一个方法，MOF 中区别方法和属性的唯一办法是看是否有小括号。

第 17 到 24 行定义了 AdvDbg_UsbDisk 类的一个实例。"instance of" 是关键字，后面应该是一个非抽象类的名字。抽象类的表准是类的修饰符中包含了 abstract 修饰符。19 到 23 行为这个实例指定了属性值，值得注意的是 Name 和 Caption 都是基类中定义的属性。

那么如何编译这个 MOF 文件呢？只要使用 mofcomp 程序就可以了。Mofcomp.exe 是个命令行程序，位于 c:\<WINDOWS 根目录>\system32\wbem 目录中。打开一个命令行窗口，将该路径加到 PATH 环境变量中后，转到 MOF 文件所在的目录，然后只要输入 mofcomp advdbg_usbdisk.mof 就可以了，在笔者的机器上，其输出如下：

```
c:\dbg\author\code\chap31\mof>mofcomp advdbg_usbdisk.mof
Microsoft (R) 32-bit MOF Compiler Version 5.1.2600.2180
Copyright (c) Microsoft Corp. 1997-2001. All rights reserved.
Parsing MOF file: advdbg_usbdisk.mof
MOF file has been successfully parsed
Storing data in the repository...
Done!
```

说是编译器，其实 mofcomp 不仅对 mof 文件进行解析和检查，如果没有错误，mofcomp 还会将该类定义加到 CIM 库中，上面最后两行的提示就是 mofcomp 在向 CIM 库存储数据。

以上操作成功后，再次打开 CIM Studio，连接到 root\CimV2 命名空间，就可以查找到刚刚定义的 AdvDbg_UsbDisk 类了，点击右侧的显示类实例按钮还可以看到我们定义对象实例（图 31-6）。

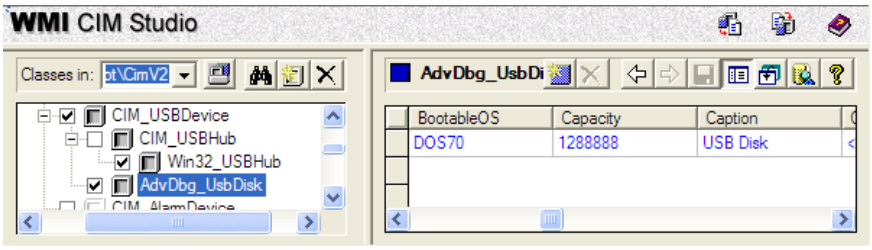


图 31-6 观察我们自己定义的 AdvDbg_UsbDisk 类和实例

31.3 WMI 的架构和基础构件

WMI 是 WBEM 标准在 Windows 系统中的应用和实现。对于今天的 Windows 系统，它已经成为作为操作系统的一个基本部件，为系统中的其它部件提供 WBEM 支持和服务。下面我们先来看一下 WMI 的基本架构。

31.3.1 WMI 的架构

从架构角度来看，整个 WMI 系统由以下几个部分组成（参见图 31-1）：

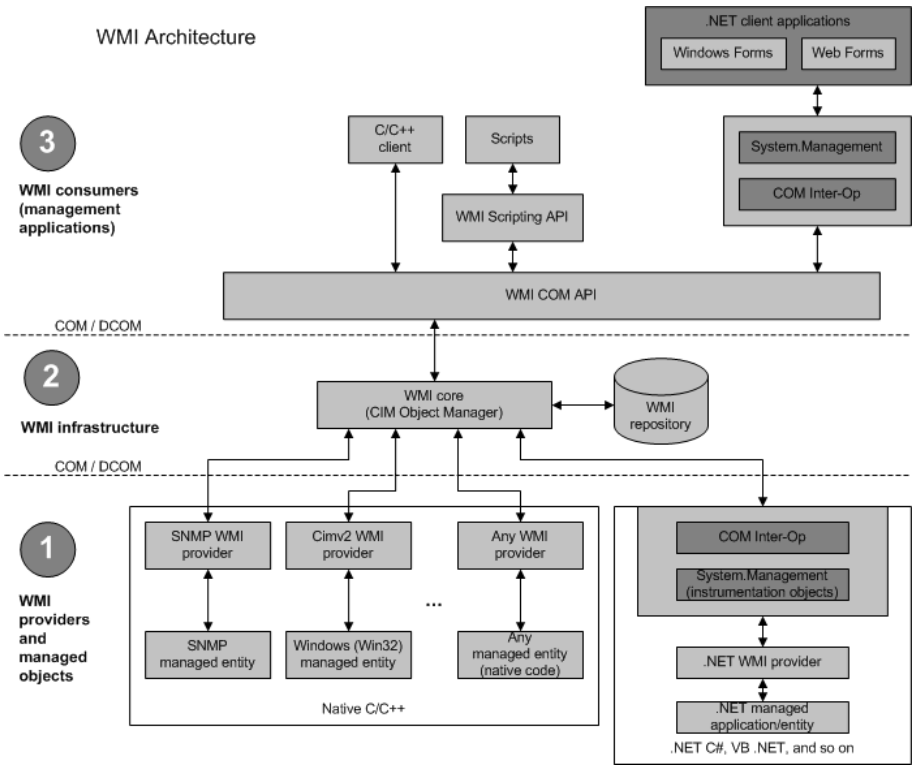


图 31-7 WMI 架构

受管对象（Managed Objects）：即要管理的目标对象,使用 WMI 的目的就是获得这些对象的信息或者配置它们的行为。

- WMI 提供器(WMI Providers)：按照 WMI 标准编写的软件组件，它代表受管对象与 WMI 管理器交互，向其提供数据或者执行其下达的操作。WMI 提供器隐藏了各种受管对象的差异，使 WMI 管理器可以以统一的方式查询和管理受管对象。
- WMI 基础构件（WMI Infrastructure）：包括存储对象信息的数据库和实现 WMI 核心功能的对象管理器。因为 WMI 使用 CIM（Common Information Model）标准来描述和管理受管对象。因此，WMI 的数据库和对象管理器被命名为 CIM 数据仓库（CIM Repository）和 CIM 对象管理器（CIM Object Manager，简称 CIMOM）。
- WMI 编程接口（API）：WMI 提供了几种形式的 API 接口，以方便不同类型的 WMI 应用使用 WMI 功能，比如供 C/C++程序调用的函数形式（DLL+Lib+头文件），供 VB 和脚本语言调用 ActiveX 控件形式，和通过 ODBC 访问的数据库形式（ODBC Adaptor）。
- WMI 应用程序（WMI Applications）：即通过 WMI API 使用 WMI 服务的各种工具和

应用程序。比如 Windows 中的事件查看器程序，以及各种实用 WMI 的 Windows 脚本。因为从数据流向角度看，WMI 应用程序是消耗 WMI 提供器所提供的信息的，所以有时又被称为 WMI 消耗器（WMI Consumer）。

从上面的架构图可以看出，WMI 是个由 WMI 提供器、WMI 消耗器和 WMI 基础构件组成的一个复杂系统。用户是使用 WMI 应用程序（消耗器）来管理他们所关心的各种目标对象（受管对象），WMI 基础构件好似一个枢纽为消耗器和提供器的信息交流提供通道。换句话来讲，是 WMI 基础构件搭建了一个平台，使 WMI 应用程序可以利用这个平台找到它要找的信息。下面我们就分别介绍一下支撑起这个平台的各个 WMI 基础构件。

31.3.2 WMI 的工作目录和文件

默认情况下，WMI 的工作目录位于 Windows 系统目录下的 system32\wbem 下（参见图 31-8）。WBEM 目录下保存了很多 WMI 的程序文件（EXE 和 DLL），还有 COM 类型库文件（TLB）和一部分 CIM 类定义文件（MOF 和 MFL）。

WBEM 目录下还包含了几个子目录，Logs 子目录是用来存储 WMI 日志文件的，AutoRecovery 子目录保存了可以自动恢复的类的 MOF 文件的备份。另一个重要的子目录就是 Repository，它是存储 WMI 数据仓库文件的地方。

WMI 数据仓库文件

WMI 将类和对象等信息存储在 WMI 数据仓库中。实现 WMI 数据仓库的文件被保存在系统目录下的 wbem\Repository\FS 目录下，如图 31-8 所示。

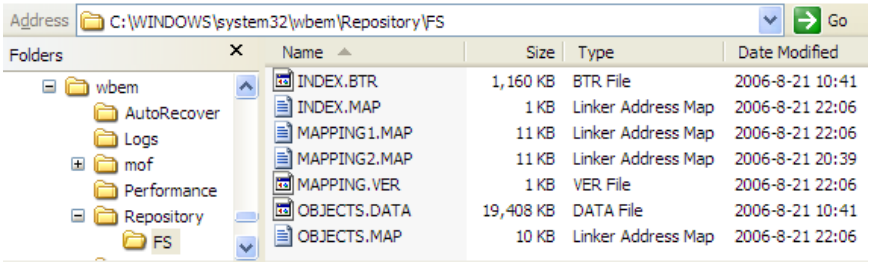


图 31-8 组成 WMI 数据仓库的各个文件

其中最重要的文件是 Objects.DATA，用于存放数据，其它几个是索引和映射文件。可以使用 winmgmt 程序对 WMI 数据仓库进行备份。比如输入如下命令，便可以将 WMI 数据仓库的所有数据备份到一个文件中。

```
winmgmt /backup c:\windows\system32\wbem\aug_bakup.data
```

使用/restore 开关可以恢复 WMI 数据仓库。键入 winmgmt /？可以得到简单的帮助信息。

system32\wbem 目录下的 repdrvfs.dll 是管理和维护 WMI 数据仓库的主要模块，我们将在下面详细介绍。

WMI 程序文件

了解 WMI 的程序文件有助于从文件层次理解 WMI 的架构层次和模块组织。WMI 的大多数程序文件都位于 Windows 系统目录下的 system32\wbem 下。表 31-2 列出了大多数 WMI 程序文件的名称和主要功能。

表 31-2 WMI 的程序文件

文件名	简介
wbemcore.dll	WMI 的核心模块，包括 CIM 对象管理器等重要基础设施。
cimwin32.dll	WMI 的 Win32 提供器，内部包含了很多重要 Win32 类的实现。
wmipiprt.dll	IP 路由事件（IP Route Event）提供器。
wmipdskq.dll	磁盘配额（Disk Quota Volume）提供器。
wmipcima.dll	WBEM Framework Instance Provider CIMA
Wbemprox.dll	WBEM 代理，供 WMI 应用程序连接 WMI 服务，包含了 IWbemLocator 接口的实现（Clocator 类）。
Wbemperf.dll	性能计数器（NT5 Base Perf）提供器。
Wmipicmp.dll	Ping 提供器，ICMP 是 Internet Control Message Protocol 的缩写。
Stdprov.dll	PerfMon 和注册表提供器。
Wbemdisp.dll	包含了供脚本语言使用的各种 ActiveX 控件的实现。
Wmiprov.dll	WDM 提供器（实例、事件和 HiPerf）。
Wmiutils.dll	解析和执行 WQL 查询的 COM 组件。
Wbemads.dll	ADSI 扩展。
Wmicookr.dll	WMI 高性能计数器数据加工期（Cooker）。
Msiprov.dll	MSI（MS Installer）提供器。
Wbemcntl.dll	WMISnapin 组件，即配置 WMI 的 MMC（Microsoft Management Console）插件。
Repdrvfs.dll	包含了管理 CIM 对象数据仓库的各个类，参见下文。
Scrcons.exe	供脚本（Active Scripting）使用的事件消耗器提供器。
Fastprox.dll	包含了用于进程间调用和 RPC 通信的类和函数，又称为 Microsoft WBEM Call Context。
Wbemcons.dll	命令行的事件消耗器提供器。
Wmitimep.dll	当前时间提供器。
Esscli.dll	事件子系统的过滤器列集代理（filter marshaling proxy）。
Wbemess.dll	WMI 的事件子系统。
Fwdprov.dll	转寄（Forwarding）事件提供器和转寄事件消耗器提供器。
Wbemcons.dll	日志文件（Log File）和 NT 事件日志事件消耗器提供器。
Wmimsg.dll	消息服务，RPC 消息收发器（Receiver 和 Sender）。
Wbemess.dll	新的事件子系统。
Ntevt.dll	WMI 事件日志（Eventlog）事件提供器。 Event Provider
tmplprov.dll	模版（Template）提供器。
trnsprov.dll	Microsoft WBEM Transient Instance Provider
Unsecapp.exe	非安全套间（Unsecured Apartment）进程，用于需要穿过防火墙的 RPC 通信时向 MMC 或客户程序返回调用结果。
updprov.dll	Microsoft WBEM Updating Consumer Provider
viewprov.dll	Microsoft WBEM View Provider
policman.dll	安全策略状态提供器。
wmiprvsd.dll	WMI 提供器子系统 DLL。
wmiprvse.exe	WMI 提供器子系统的宿主进程。
wmidcprv.dll	提供器子系统的非耦合（Decoupled）提供器注册和事件管理。

31.3.3 CIM 对象管理器

CIM 对象管理器（CIM Object Manager，简称 CIMOM）是 WMI 的核心部件。它负责管理和维护系统中的类和对象，也是 WMI 管理程序（消耗器）和 WMI 提供器之间进行交互的桥梁。从进程的角度看，CIMOM 是工作在 WMI 服务器进程中的一系列动态链接库，它们利用 COM/DCOM 技术相互协作。对外也是以 COM 接口的形式公开它们的服务。

CIM 标准中没有规定 CIMOM 该如何实现，微软也没有公开过 WMI 的 CIMOM 的内部实现方法。所有的文档中都是将其模糊的看作一个整体，称之为 CIMOM。但是因为 CIMOM 是 WMI 的核心，了解 CIMOM 是了解 WMI 的捷径。处于这一考虑，笔者对 CIMOM 做了很多探索，仅供读者参考。必须说明的是，这些内容没有得到微软的认可和确认，也会因为 Windows 的版本不同而有所不同。

CWbemClass 类

WBEMCORE.DLL 中的 CWbemClass 类是描述和管理 CIM 类对象的一个内部类。包括存取类的名称、属性、方法、修饰符，产生类的实例，克隆（Clone）和派生类等。表 31-3 列出了 CWbemClass 类的一些重要方法和属性。

表 31-3　CRepository 类的部分方法和属性

方法或属性	描述
GetClassNameW	取类的名称。
GetPropertyCount	取属性个数。
BeginMethodEnumeration，NextMethod 和 EndMethodEnumeration	用于便利类的所有方法，分别是开始枚举类的方法，取下一个，和结束枚举。
Clone 和 CloneEx	复制类。
GetProperty，GetMethod	取类的属性和方法。
PutMethod	
SetPropQualifier 和 SetMethodQualifier	设置属性和方法的修饰符。
SpawnInstance 和 SpawnKeyedInstance	产生当前类的实例。

CWbemInstance 类

WBEMCORE.DLL 中的 CWbemInstance 类是描述和管理 CIM 类实例的一个内部类。包括读取实例的类名（class name）、修改或读取实例的属性值、复制实例数据等。

MSDN 中公开的 IWbemClassObject 接口定义了操作 WMI 类和实例的基本方法，通过该接口，WMI 应用程序可以访问相应的 WMI 类或实例。可以认为 CWbemClass 类和 CWbemInstance 类为实现这一接口的方法而提供的支持类。

CRepository 类

REPDRVFS.DLL 中的 CRepository 类是对 WMI 数据仓库（CIM Repository）的抽象，它是管理和维护 WMI 数据仓库的一个最重要的类。它实现了一系列方法来完成有关 WMI 数据仓库的各种操作，包括初始化、读取、锁定、解锁、关闭、备份、恢复等。表 31-3 列出了 CRepository 类的一些重要方法和属性。

表 31-3　CRepository 类的部分方法和属性

方法或属性	描述
Initialize	初始化 WMI 数据仓库
GetRepositoryVersions	读取 WMI 数据仓库的版本，保存在全局变量 repdrvfs!g_dwCurrentRepositoryVersion 中，对于 XP SP2，为 6。
GetRepositoryDirectory	从注册表中（"SOFTWARE\Microsoft\ WBEM\CIMOM"）读取 WMI 数据仓库文件的路径，默认为 %SystemRoot%\system32\WBEM\Repository。
GetNamespaceHandle	获取某个命名空间的句柄。这是从仓库中读取数据的主要途径。
GetStatistics	读取统计信息。
Backup 和 Restore	备份和恢复数据仓库。

方法或属性	描述
Logon	登陆数据仓库。
Shutdown	关闭 WMI 数据仓库。
LockRepository 和 UnlockRepository	锁定和解除锁定。
ReadOperationNotification 和 WriteOperationNotification	当有读/写数据仓库的操作发生时，此方法会被调用，以产生事件通知。
FlushCache	冲转缓存区，即将缓存在内存中的数据写入文件。
m_ulReadCount 和 m_ulWriteCount	读/写次数计数。
m_threadCount	工作线程数。

在 WBEMCORE.DLL 中也有个 CRepository 类，它是对 CIM 数据仓库的顶层抽象，对于需要底层操作的任务，它仍需转给 REPDRVFS.DLL 中的 CRepository 类来完成。

CNamespaceHandle 类

REPDRVFS.DLL 中的 CNamespaceHandle 类是对 CIM 命名空间（CIM Namespace）物理特性的抽象，它封装了关于命名空间的各种底层操作，包括初始化命名空间和向命名空间中增加删除类、实例和关系等。表 31-4 列出了 CNamespaceHandle 类主要方法和简单说明。

表 31-4　CNamespaceHandle 类的主要方法

方法或属性	描述
Initialize 和 Initialize2	初始化命名空间。
DeleteInstance	删除实例。
PutInstance	加入实例。
DeleteDerivedClasses	删除派生类。
EraseClassRelationships	删除类关系。
GetInstanceByKey	根据键值取实例。
FireEvent	激发事件。
DeleteObjectByPath	根据路径删除对象。
PutObject	保存或加入对象。
PutClass	保存或加入类。
DeleteClass	删除类。
DeleteClassInstances	删除类实例。
EnumerateClasses	枚举命名空间中所包含的类。
ExecQuery	查询功能的总入口函数，预处理后产生一个纤程（Fiber）任务（CreateFiberForTask）。而后再在纤程分发给真正的查询函数（ExecInstanceQuery 或 ExecClassQuery）。
ExecInstanceQuery	查询实例。
GetObjectW	读取对象。
ExecClassQuery	查询类。

CSession 类

REPDRVFS.DLL 中的 CSession 类是外界与 WMI 数据仓库对话的主要媒介。通常，数据仓库的使用者只要调用 CSession 类封装好的方法，而不必关心数据仓库内部的操作细节。CSession 类接收到调用后通常再转发给内部的其它类来真正完成各种操作。

清单 31-3 显示了一个典型的查询操作的执行过程，wbemcore.dll 中的 CRepository 的

ExecQuery 方法调用 repdrvfs.dll 中的 CSession 类的 ExecQuery 方法。而后 CSession 类的 ExecQuery 再分发给 CNamespaceHandle 类的 ExecQuery 方法。

清单 31-3 一个典型的查询操作的执行过程

```
0:022> k
ChildEBP RetAddr
011cfbdc 752146c9 repdrvfs!CNamespaceHandle::ExecQuery
011cfc1c 762d11e0 repdrvfs!CSession::ExecQuery+0xb6
011cfc7c 762d167e wbemcore!CRepository::ExecQuery+0xb5
011cfcd8 762ddadc wbemcore!CRepository::GetRefClasses+0x8d
```

表 31-5 CSession 类的主要方法

方法或属性	描述
ExecQuery	执行查询，通常是调用 CNamespaceHandle 的 ExecQuery。
BeginWriteTransaction 和 BeginReadTransaction	启动读/写事务。
RenameObject	重命名对象。
PutObject	保存对象。
CommitTransaction	提交事务。
AddObject 和 DeleteObject	加入和删除对象。
GetObjectByPath	根据路径取对象。
GetObjectW	取对象。
Enumerate	枚举。

CWbemNamespace 类

WBEMCORE.DLL 中的 CWbemNamespace 类是对 CIM 命名空间的逻辑抽象。它封装了命名空间的各种行为和针对空间内类或对象的操作，包括访问和管理命名空间中的对象、执行各种查询操作、事件通知、和安全控制等。CWbemNamespace 是外部访问 WMI 类和对象的主要途径。CWbemNamespace 类包含了 100 多个方法，可以说是 WMI 对象管理器中处于核心地位一个类。表 31-6 列出了 CWbemNamespace 类的部分方法和简要说明。

表 31-6 CWbemNamespace 类的主要方法

方法或属性	描述
UniversalConnect	连接命名空间。
ExecNotificationQuery	执行查询，通常是调用 CNamespaceHandle 的 ExecQuery。
InitializeSD	初始化安全描述符。
PutInstance 和 PutInstanceAsync	创建或更新实例，以 Async 结尾的是异步调用。
PutClass 和 PutClassAsync	创建或更新类，以 Async 结尾的是异步调用。
GetObjectByFullPath	根据路径取得对象。
DeleteObject 、 DeleteClass 和 DeleteInstance	删除对象、类和实例。
CreateNamespace	创建命名空间。
EnsureSecurity、PutAceList	安全检查，加入 ACE（Access Control Entry 访问控制表项）。
GetObjectW	读取对象。
ExecQuery、ExecSyncQuery 和 ExecQueryAsync	执行查询，Sync 代表同步，Async 代表异步。

CCoreServices 类

从客户服务器（Client/Server）模型的角度来看，WMI 应用程序（WMI 消耗器）利用 WMI 管理受管对象，因此 WMI 应用程序是客户，WMI 命名空间中的类和对象为其提供

了管理服务，是服务（Service）。广义来说，WMI 服务是指 WMI 的基础构件和 WMI 提供器所构成的整体。狭义来说，WMI 服务是实现了 IWbemServices 接口的 COM 组件。MSDN 文档公开了 IWbemServices 接口的定义。所有 WMI 提供器和其它服务提供者都应该实现这一接口。CCoreServices 类是位于 WBEMCORE.DLL 中的一个 WMI 内部类，它的主要职责就是管理实现了 IWbemServices 接口的各种 WMI 服务，包括初始化系统内的子系统和内部服务、创建服务实例和管理事件等。表 31-7 列出了 CCoreServices 类的主要方法和属性。

表 31-7　CCoreServices 类的主要方法和属性

方法或属性	描述
IsProviderSubsystemEnabled	通过查询注册表键"SOFTWARE\Microsoft\WBEM\CIMOM"中的"Enable Provider Subsystem"键值，判断提供器子系统是否启用。
GetServices2	取得 WMI 服务组件的实例，第二个参数是使用路径表示的服务，如\\.\root\directory\LDAP，第三个参数是用户名，参见下文。
DeliverIntrinsicEvent 和 DeliverExtrinsicEvent	投递内部和外部事件。
m_pEssOld 和 m_pEssNew	指向事件子系统（Event Subsystem）的指针。
g_pSvc	指向本类全局实例的指针。
StartEventDelivery 和 StopEventDelivery	启动和停止事件投递。
SetCounter、IncrementCounter 和 DecrementCounter	设置、递增和递减用于记录服务实例个数的计数器。
GetObjFactory	取得对象工厂。
RegisterWriteHook 和 UnregisterWriteHook	注册和注销写挂钩。
InitRefresherMgr	初始化刷新器（refresher）管理器，记录在 m_pFetchRefrMgr 成员中。
CreatePathParser	创建路径解析器。
GetProviderSubsystem	取提供器子系统指针，记录在 m_pProvSS 成员变量中。
CreateFinalizer	创建终结器（Finalizer）。
m_pProvSS	提供器之系统指针。
CreateQueryParser	创建查询解析器。
GetRepositoryDriver	取 CIM 仓库驱动，已经不用，返回错误。
GetSystemClass 和 GetSystemObjects	取系统类和对象，支持 WMI 基础架构的类被称为 WMI 系统类，相应的，其实例被称为系统对象。

清单 31-4 显示了当 WMI 应用程序中连接一个命名空间时，WMI 服务器进程内部的执行过程：CWbemLevel1Login::LoginUser 调用 CCoreServices::GetServices2，然后再调用 CWbemNamespace 类的连接函数，最后创建命名空间的实例。

清单 31-4　WMI 应用程序连接命名空间的内部过程

```
0:081> k
ChildEBP RetAddr
0353f3a0 762ea4a9 wbemcore!CWbemNamespace::CreateInstance+0x56
0353f3b4 762ea548 wbemcore!CWbemNamespace::UniversalConnect+0x47
0353f400 762da5ee wbemcore!CWbemNamespace::PathBasedConnect+0x3c
0353f434 762fbcd7 wbemcore!CCoreServices::GetServices2+0x2b
0353f4a8 762fc0ad wbemcore!CWbemLevel1Login::LoginUser+0x1cf
0353f548 762fc12f wbemcore!CWbemLevel1Login::ConnectorLogin+0x2fc
0353f56c 77e79dc9 wbemcore!CWbemLevel1Login::NTLMLogin+0x21
[以下是 RPC 工作函数，省略]
```

31.3.4 WMI 服务进程

WMI 服务是以进程外服务的形式提供的，WMI 应用程序通过 WMI API 调用位于 WMI 服务进程中的 WMI 服务。因为 WMI 服务都是以 COM/DCOM 形式封装好的，所以调用 WMI 服务的过程是典型的调用 EXE 中的 COM/DCOM 服务器的过程。

WMI 服务进程是 WMI 服务组件的宿主（host）进程，在 Windows 9x 系统中，WMI 服务进程是以单独的可执行文件（Winmgmt.exe）形式运行的。在 Windows NT 和 2000 系统中，WMI 服务进程是以 Windows 系统服务的形式自动启动和运行的，服务的可执行文件也是 Winmgmt.exe。在 Windows XP 系统中，WMI 服务进程也是以系统服务的形式运行的，不过服务的可执行文件是 SVCHOST.EXE。

SVCHOST.EXE

SVCHOST.EXE 是 Windows 系统中的多个服务的共享宿主进程。在典型的 Windows XP 系统中，通常有三个或更多的 SVCHOST 进程的实例在运行，承载着多个不同的系统服务，典型的有 WMI 服务，RPC 服务等。通常每个 SVCHOST 进程实例负责一组服务，以下注册表表键下定义了各个组的名称和每组所包含的服务：

```
HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\Svchost
```

图 31-9 显示了在笔者使用的 Windows XP SP2 系统上，使用 SVCHOST 作为宿主进程的各组服务。图中右侧的每个键值定义一个组，所以共有 8 个组。键值名称即组名，键值数据包含改组内的服务的服务名（Service Name）。每个服务在如下表键下都有个子键，定义了该服务的详细信息。

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\
```

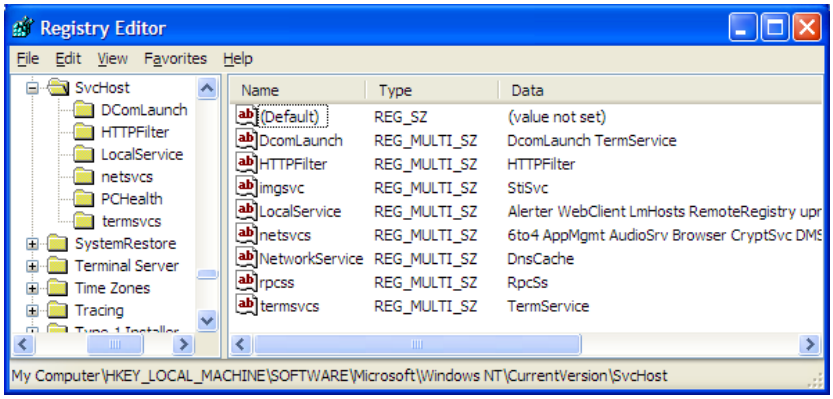


图 31-9　使用 SVCHOST 作为宿主进程的系统服务

每个 SVCHOST 进程负责一组服务，所以如果以上 8 组服务全部启动，那么在任务管理器中就会看到 8 个 SVCHOST 进程，但因为很多服务都是按需要自动启动的，所有有些服务可能并不是每次都启动。使用 tasklist /SVC 命令可以列出系统中的所有进程中所包含的系统服务。从中可以看到 SVCHOST 进程的每个实例，和这个实例中所承载的各个服务（清单 31-5）。

清单 31-5　使用 tasklist /SVC 命令观察进程中所包含的系统服务

```
C:\>tasklist /SVC
Image Name                     PID Services
========================= ====== ============================================
System Idle Process              0 N/A
System                           4 N/A
smss.exe                       876 N/A
```

```
csrss.exe                   1340 N/A
winlogon.exe                1368 N/A
services.exe                1412 Eventlog, PlugPlay
lsass.exe                   1424 Netlogon, PolicyAgent, ProtectedStorage,
                                 SamSs
ibmpmsvc.exe                1584 IBMPMSVC
ati2evxx.exe                1612 Ati HotKey Poller
svchost.exe                 1624 DcomLaunch, TermService
svchost.exe                 1700 RpcSs
svchost.exe                 1896 AudioSrv, CryptSvc, Dhcp, ERSvc,
                                 EventSystem, helpsvc, Irmon, lanmanserver,
                                 lanmanworkstation, Netman, Nla, RasMan,
                                 Schedule, seclogon, SENS, ShellHWDetection,
                                 srservice, TapiSrv, Themes, TrkWks, W32Time,
                                 winmgmt, WZCSVC
S24EvMon.exe                1932 S24EventMonitor
svchost.exe                  272 Dnscache
svchost.exe                  524 LmHosts, RemoteRegistry, SSDPSRV, WebClient
spoolsv.exe                  836 Spooler
[后面的省略]
```

列表的第一列是进程的映像文件名称，第二列是进程 ID，第三类是该进程所包含的系统服务，N/A（Not Applicable）该进程内没有系统服务。使用 tlist 工具（参见第 22 章）也可以看到类似的信息。

使用 SVCHOST 作为宿主进程的系统服务需要将自己的真正服务模块实现在 DLL 中，并通过注册表中该服务的 Parameters 表键将 DLL 文件名称（ServiceDll）和入口函数（ServiceMain）告诉给 SVCHOST 进程。比如 WMI 服务的注册表设置如图 31-10 所示。

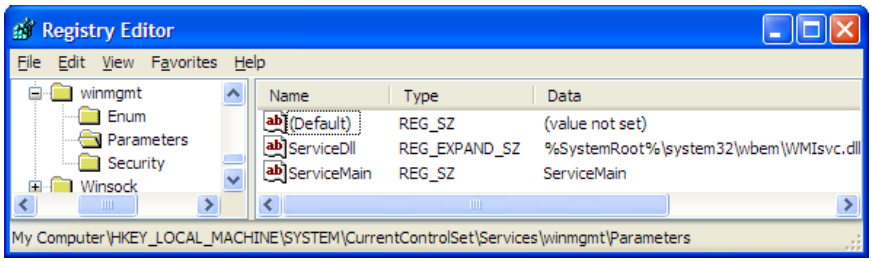


图 31-10　WMI 服务的注册表设置

要说明的是，WMI 服务的服务名是 winmgmt，并非 WMI。注册表中却是存在名为 WMI 的服务，但它是 WMI 的驱动程序扩展（Driver Extensions），用于支持内核态的驱动程序实现 WMI 有关的功能。从图 31-10 可以看到，WMI 服务的服务模块是 %SystemRoot%\system32\wbem\WMIsvc.dll，即 WMISvc.DLL，服务的主函数（入口函数）是 ServiceMain，使用 Depends 工具观察 WMISvc.DLL 文件，可以看到 ServiceMain 函数是 WMISvc.DLL 的一个导出函数。因此可想而知，当启动 WMI 服务时，SVCHOST 进程会根据 ServiceDll 指定的路径加载 WMISvc.DLL，然后动态取得并执行 ServiceMain 键值指定的服务入口函数。ServiceMain 函数内部会初始化 WMI 数据仓库和 CIM 对象管理器等基础构件，然后启动一系列工作线程开始提供服务。

31.3.5　WMI 服务的请求和处理过程

WMI 应用程序利用 DCOM 技术来使用 WMI 服务进程内的 WMI 服务。DCOM 是分布式组件模型的简称，是对 COM 技术的扩展，目的是使不同计算机上的 COM 对象可以相互通信。DCOM 协议又被称为对象 RPC（Object Remote Procedure Call），是基于标准 RPC 协议而制定的。ORPC 规约（specification）定义了如何跨计算机创建、表示、使用和维护 COM 对象以及如何调用对象的方法。COM/DCOM 运行库封装了使用 RPC 通信的细

节，使程序员可以像使用本地 COM 组件一样来使用 DCOM 组件。

为了降低设计 WMI 应用程序的难度，Windows 提供了一系列本地化的组件来进一步简化调用 WMI 服务的过程，这些组件的实现在 WBEMPROX.DLL 中。

发起请求

清单 31-6 是 WbemClient 程序（WMI 应用程序，SDK 的一个例子，位于 Samples\SysMgmt\WMI\VC\SimpleClient）通过 IWbemLocator 接口的 ConnectServer 方法连接某个命名空间时的执行过程（函数调用序列）。

清单 31-6 通过 IWbemLocator 接口的 ConnectServer 方法连接 WMI 服务的执行过程

```
0012e32c 77ef3eac RPCRT4!NdrProxySendReceive
[省略三行 RPCRT4 内部的调用]
0012e79c 77520a71 ole32!CRpcResolver::CreateInstance+0x13d
0012e9e8 7752cddf ole32!CClientContextActivator::CreateInstance+0xfa
0012ea28 7752cc24 ole32!ActivationPropertiesIn::DelegateCreateInstance+0xf7
0012f1d8 774ffaba ole32!ICoCreateInstanceEx+0x3c9
0012f200 774ffa89 ole32!CComActivator::DoCreateInstance+0x28
0012f224 74ef18c1 ole32!CoCreateInstanceEx+0x1e
0012f258 74ef186e wbemprox!CDCOMTrans::DoActualCCI+0x3d
0012f29c 74ef15db wbemprox!CDCOMTrans::DoCCI+0x12d
0012f358 74ef17e4 wbemprox!CDCOMTrans::DoActualConnection+0x25c
0012f384 74ef1ee1 wbemprox!CDCOMTrans::DoConnection+0x25
0012f3c4 00415752 wbemprox!CLocator::ConnectServer+0x7c
0012f45c 782aca70 WbemClient!CWbemClientDlg::OnConnect+0xf2
```

从下而上，我们可以清楚的看到，当我们点击 WbemClient 程序的连接按钮后，其 OnConnect 方法创建并使用 WbemLocator 对象试图连接指定的命名空间。

```
pIWbemLocator->ConnectServer(pNamespace,  // path of the namespace to connect
                         NULL,            // using current account for simplicity
                         NULL,            // using current password for simplicity
                         0L,              // locale
                         0L,              // securityFlags
                         NULL,            // authority (domain for NTLM)
                         NULL,            // context
                         &m_pIWbemServices)
```

wbemprox.dll 中的 CLocator 类是对 IWbemLocator 接口的实现。接下来，CLocator 类调用另一个内部类 CDCOMTrans 真正开始连接。CoCreateInstanceEx 是创建对象实例的一个重要 API。它既支持创建本地对象，也支持创建指定机器上的远程对象。需要说明的是，即使 WMI 应用程序就是调用同一台机器上的 WMI 服务，系统也会使用统一的 RPC 机制进行处理。最后，RPCRT4!NdrProxySendReceive 函数将数据消息发送给服务器并等待回复。NDR 是 Network Data Representation 的缩写，即网络数据表示，DCOM 和 RPC 底层负责数据列集（marshaling）和网络通信的一系列函数和类通常被称为 NDR 引擎。

顺便介绍一个小窍门，因为 NDR 的收发函数是 RPC 和 DCOM 调用的一条必经之路，所以通过对这些函数设置断点可以截获 RPC 和 DCOM 调用，然后可以打印函数调用序列或者使用 WinDbg 的关于 RPC 扩展命令（输入!rpcexts.help 显示帮助）显示 MIDL_STUB_MESSAGE（!rpcstub）、RPC_MESSAGE（!rpcmsg）等结构。

受理服务

DCOM 和 RPC 机制会将 WMI 应用程序发起的服务调用转发给 WMI 服务器进程中的相应函数（组件的方法）。对于枚举和查询这样的请求，WMI 会将该请求放入一个对列，然后由现有的或新启动的工作线程来处理这个请求。清单 31-7 显示了 WMI 服务进程（SVCHOST 进程）内接收枚举实例（enuerate instance）请求并将其放入对列的执行过程。

清单 31-7　WMI 服务进程接收枚举实例请求并将其放入对列的执行过程

```
0:064> k
ChildEBP RetAddr
0158f418 762cec2f wbemcore!CCoreQueue::PlaceRequestInQueue+0xba
0158f4a4 762f7463 wbemcore!CCoreQueue::Enqueue+0x1cf
0158f4e4 762ee2a0 wbemcore!ConfigMgr::EnqueueRequest+0x77
0158f518 762eabeb wbemcore!CWbemNamespace::_CreateInstanceEnumAsync+0x19f
0158f560 77e79dc9 wbemcore!CWbemNamespace::CreateInstanceEnum+0xae
[以下是 RPC 工作函数，省略]
```

对于连接命名空间这样的请求，WMI 会立即处理，前面的清单 31-5 显示了其内部过程。

WMI 的工作线程会依次处理被放在对列中的请求任务，位于 WBEMCORE.DLL 中的 CCoreQueue 是专门用于管理和维护 WMI 请求对列的一个内部类，它提供了很多方法用于完成对对列的操作，包括加入请求（Enqueue、ExecSubRequest）、执行请求（Execute）、创建新工作线程（DoesNeedNewThread 和 CreateNewThread）。清单 31-8 显示了一个 WMI 工作线程执行队列中的查询请求的过程。

清单 31-8　WMI 工作线程执行队列中的查询请求的过程

```
0:022> k
ChildEBP RetAddr
011cfbdc 752146c9 repdrvfs!CNamespaceHandle::ExecQuery
011cfc1c 762d11e0 repdrvfs!CSession::ExecQuery+0xb6
011cfc7c 762d167e wbemcore!CRepository::ExecQuery+0xb5
011cfcd8 762ddadc wbemcore!CRepository::GetRefClasses+0x8d
011cfcf4 762e124d wbemcore!CAssocQuery::Db_GetRefClasses+0x20
011cfd58 762e2328 wbemcore!CAssocQuery::BuildMasterAssocClassList+0x71
011cfd9c 762e25a8 wbemcore!CAssocQuery::ExecNormalQuery+0x6b
011cfdec 76303336 wbemcore!CAssocQuery::Execute+0x178
011cfe8c 762fc769 wbemcore!CQueryEngine::ExecQuery+0x2a1
011cfea8 762cef24 wbemcore!CAsyncReq_ExecQueryAsync::Execute+0x19
011cfed4 762ced4e wbemcore!CCoreQueue::pExecute+0x3c
011cff04 762f25cb wbemcore!CCoreQueue::Execute+0x18
011cff4c 762cee89 wbemcore!CWbemQueue::Execute+0xf6
011cff80 762cf0f9 wbemcore!CCoreQueue::ThreadMain+0x111
011cffb4 7c80b50b wbemcore!CCoreQueue::_ThreadEntryRescue+0x56
011cffec 00000000 kernel32!BaseThreadStart+0x37
```

31.4　WMI 提供器

从广义来讲，凡是为 WMI 应用程序（WMI 消耗器）提供管理数据或执行操作的 WMI 组件都属于 WMI 提供器，包括 CIM 中使用 MOF 编写的各个类以及它们的实例。但是很多时候，WMI 提供器是特指通过 COM API 与 WMI 核心部件交互而提供管理服务（尤其是动态信息）的 WMI 组件。从 COM 接口的角度来讲，WMI 提供器就是实现了 WMI 对象管理器所规定接口（例如 IWbemServices 和 IWbemProviderInit）的 COM 组件。

从提供器所提供内容的属性来看，WMI 提供器主要提供以下三类内容（功能）：

- 类：定义了（包含了）新的类，这样的提供器又叫类提供器。
- 实例：定义了类的实例，这样的提供器又叫实例提供器。
- 事件：定义了新的事件，这样的提供器又叫事件提供器。
- 方法：主要是实现 IWbemServices 接口的 ExecMethodAsync 方法，即执行类的方法。这样的提供器又叫方法提供器。
- 属性：实现了 IWbemPropertyProvider，为某个 WMI 类提供和设置属性数据。这样的提供器又叫属性提供器。
- 事件消耗器：定义了永久的事件接受器，这样的提供器又叫事件消耗器提供器。

- 一个提供器通常至少提供以上六种功能中的一种，当然也可以同时提供多种功能，例如纯粹的方法提供器很少，通常是与类和实例提供器实现在一起。

31.4.1 Windows 系统的 WMI 提供器

为了提供简单一致的可管理性，Windows 操作系统本身和微软的很多产品都内嵌了对 WMI 的支持，配备了 WMI 提供器可以被管理程序通过 WMI 来访问和管理。表 31-8 列出了 Windows 系统中常见的 WMI 提供器和简要描述。

表 31-8 Windows 系统中常用的 WMI 提供器

WMI 提供器	管理目标	类、实例或事件*
内核追踪提供器	内核追踪（kernel tracing）事件，参见 17 章关于的介绍。	Win32_ProcessTrace，Win32_ThreadTrace，Win32_ModuleLoadTrace 等
活动目录提供器	活动目录（Active Directory）中的对象。	[WMI\LDAP]：DS_LDAP_Class_Containment，RootDSE
BizTalk 提供器	BizTalk 服务器。	Win32_PerfRawData 及其派生类。
性能计数器提供器	原始的性能计数器数据。	
加工后计数器提供器	计算好的（Cooked）计数器数据。	Win32_PerfFormattedData 及其派生类
Perfmon 提供器	性能监视数据，不是高性能的计数器。建议使用上面两种。	实例提供器。
DFS 提供器	Distributed File System (DFS) 。	Win32_DFSNode，Win32_DFSTarget，Win32_DFSNodeTarget
DNS 提供器	Domain Name System (DNS) 资源记录（resource records，简称 RR）以及 DNS 服务。	
Disk Quota 提供器	每个用户可以存储在 NTFS 文件系统中的最大数据量（数据配额）。	Win32_DiskQuota，Win32_QuotaSetting，Win32_VolumeQuotaSetting
Event Log 提供器	事件日志（Event Log）数据。	Win32_NTEventlogFile Win32_NTLogEvent Win32_NTLogEventLog
IIS 提供器	IIS（Internet Information Services）服务。	IIsWebServerSetting
IP Route 提供器	网络路由信息。	Win32_IP4RouteTable，Win32_IP4PersistedRouteTable，Win32_ActiveRoute，Win32_IP4RouteTableEvent
Job Object 提供器	命名的作业（Job）内核对象	Win32_NamedJobObjectStatistics Win32_NamedJobObjectProcess Win32_NamedJobObjectLimit Win32_NamedJobObjectSecLimit
Ping 提供器	PING 命令得到的状态信息。	Win32_PingStatus
Policy 提供器	组策略。	[\root\policy]：MSFT_Providers，MSFT_Rule，MSFT_SomFilter
电源管理事件提供器	电源管理事件。	事件提供器（MS_Power_Management_Event_Provider）

WMI 提供器	管理目标	类、实例或事件*
Security 提供器	安全设置。	Win32_AccountSID， Win32_Ace，Win32_SID Win32_LogicalFileAccess ，Win32_LogicalFileAuditing ，Win32_Trustee 等。
Session 提供器	网络会话和连接（network sessions and connections）。	Win32_ServerSession Win32_ServerConnection
Shadow Copy 提供器	共享资源（文件夹）的 Shadow 复制。	Win32_ShadowProvider Win32_ShadowCopy Win32_ShadowContext
SNMP 提供器	MIB （ Management Information Base）中定义的 Simple Network Management Protocol (SNMP)对象。	
Storage Volume 提供器	存储卷（storage volume）	Win32_Volume Win32_DefragAnalysis
System Registry 提供器	系统注册表。	[root\DEFAULT]: StdRegProv ，RegistryKeyChangeEvent 等。"RegProv" ，"RegistryEventProvider" ，"RegPropProv"
终端服务提供器	Terminal Services。	Win32_Terminal ，Win32_TerminalService ，Win32_TerminalServiceSetting 等。
Trustmon 提供器	域（domain）之间的信赖关系。	[root\microsoftactivedirectory]：Microsoft_TrustProvider ，Microsoft_DomainTrustStatus ，Microsoft_LocalDomainInfo
WDM 提供器	符合 Windows Driver Model (WDM)规范的设备驱动程序。	[root\wmi]：
Win32 提供器	Windows 系统。	以 Win32_开头的众多类。
Windows Installer 提供器	Windows Installer（MSI）及与其兼容的应用程序。	Win32_Product ，Win32_SoftwareElement ，Win32_SoftwareFeature 等。
Windows 产品激活提供器	Windows 产品激活（ Windows Product Activation ，简称 WPA） 管理。	Win32_ComputerSystemWindows-ProductActivationSetting Win32_Proxy Win32_WindowsProductActivation

*方括号中为命名空间路径。

31.4.2 编写新的 WMI 提供器

可以把开发 WMI 提供器的过程分成两个主要任务，一是编写提供服务的 COM 组件，二是向 CIM 对象管理器注册。

编写 WMI 提供器的 COM 组件

编写 WMI 提供器 COM 组件的过程和编写普通 COM 组件很类似。设计普通 COM 组件时，我们首先要考虑的一个问题就是该组件要实现什么样的接口（interface）。因为接口

决定了组件的主要功能和被调用方式。

IWbemServices 接口是 WMI 对外提供服务的重要窗口，WMI 应用程序（消耗器）成功连接到 WMI 的某个命名空间后，WMI 对象管理器返回给 WMI 应用程序的最重要信息便是一个 IWbemServices 类型的对象指针。因为 IWbemServices 接口具有非常好的通用性，所以很多 WMI 组件都实现了这个接口，比如 WMI 命名空间（CWbemNameSpace）。WMI 中的通信（方法调用）也经常使用 IWbemServices 类型。IWbemServices 接口也是大多数 WMI 提供器要实现的首要接口。在 WMI 提供器五种类型中，有三种（类提供器、实例提供器和方法提供器）都是以 IWbemServices 接口为核心的。表 31-9 列出了 IWbemServices 接口的主要方法。

表 31-9　IWbemServices 接口的主要方法

方法	描述
OpenNamespace	打开指定的子命名空间。
CancelAsyncCall	取消一个正在执行的异步调用。
QueryObjectSink	Allows a caller to obtain a notification handler sink.
GetObject	Retrieves an object—an instance or class definition.
GetObjectAsync	Asynchronously retrieves an object—an instance or class definition.
PutClass	Creates or updates a class definition.
PutClassAsync	Asynchronously creates or updates a class definition.
DeleteClass	Deletes a class.
DeleteClassAsync	Deletes a class and receives confirmation asynchronously.
CreateClassEnum	Creates a class enumerator.
CreateClassEnumAsync	Creates a class enumerator that executes asynchronously.
PutInstance	Creates or updates an instance of a specific class.
PutInstanceAsync	Asynchronously creates or updates an instance of a specific class.
DeleteInstance	Deletes a specific instance of a class.
DeleteInstanceAsync	Deletes an instance and provides confirmation asynchronously.
CreateInstanceEnum	Creates an instance enumerator.
CreateInstanceEnumAsync	Creates an instance enumerator that executes asynchronously.
ExecQuery	Executes a query to retrieve classes or instances.
ExecQueryAsync	Executes a query to retrieve classes or instances asynchronously.
ExecNotificationQuery	Executes a query to receive events.
ExecNotificationQueryAsync	Executes a query to receive events asynchronously.
ExecMethod	Executes an object method.
ExecMethodAsync	Executes an object method asynchronously.

IWbemProviderInit 接口的 Initialize 方法是 WMI 提供器所属的命名空间对象对其进行初始化的主要途径，通过该方法，命名空间对象让提供器得到初始化的机会，并将必要的信息传递给提供器，其中包括指向自身的一个指针，提供器可以通过该指针反过来调用 CIM 对象管理器的方法。

```
HRESULT Initialize(
  LPWSTR wszUser,
  LONG lFlags,
  LPWSTR wszNamespace,
  LPWSTR wszLocale,
  IWbemServices* pNamespace,
  IWbemContext* pCtx,
```

```
  IWbemProviderInitSink* pInitSink
);
```

因此，大多数 WMI 提供器也通常会实现 IWbemProviderInit 接口。

如果使用 C++语言来，那么 WMI 类提供器、实例提供器和方法提供器的核心类的典型定义就是：

```
class CInstOrClassOrMethodProvider : public IWbemServices, public IWbemProviderInit
```

接下来的任务就是实现 IWbemProviderInit 和 IWbemServices 接口所定义的各个方法了。我们暂时跳过这一内容，留给第 20 章结合实际任务进行讨论。

事件提供器要实现的主要接口是 IWbemEventProvider，通常还实现 IWbemEventProvider 和 IWbemEventProviderSecurity 分别用于初始化和安全控制。

```
class CMyEventProvider : public IWbemEventProvider,
 public IWbemProviderInit, public IWbemEventProviderSecurity
```

IWbemEventProviderSecurity 接口只有一个方法 AccessCheck（不包括从 IUnknown 继承的方法），是用来检查请求订阅事件的事件消耗器程序是否可以接受该事件。

属性提供器要实现的首要接口是 IWbemPropertyProvider，该接口有两个方法 GetProperty 和 PutProperty。

```
class CPropPro : public IWbemPropertyProvider
```

事件消耗器提供器要实现的主要接口是 IWbemEventConsumerProvider，该接口只有一个方法，即 FindConsumer。

```
HRESULT FindConsumer(
  IWbemClassObject* pLogicalConsumer,
  IWbemUnboundObjectSink** ppConsumer
);
```

通过该方法，WMI 将一个逻辑消耗器对象传递给事件消耗器提供器，事件消耗器提供器应该返回一个事件接插器对象（event sink object）供事件类触发事件时使用。

实现好主要的提供器类后，还有一个工作就是要实现类工厂，或者说实现并导出 DllGetClassObject 方法。通常每个 DLL 形式的 COM 组件都会导出 DllGetClassObject 方法，目的是当系统中有客户程序要使用该 COM 组件时，COM 库函数会调用 DllGetClassObject 方法让 COM 组件创建指定接口的对象实例。关于如何编写类工厂的进一步细节超出了本书的范围，WMI SDK（Pltform SDK）对于每种 WMI 提供器都给出了一个完整的例子，位于 Samples\SysMgmt\WMI\VC\目录下。感兴趣的读者可以参考，在此不再详述。

注册 WMI 提供器

编写好的 WMI 提供器组件和其它 COM 组件一样可以注册到 Windows 系统中，通常是通过 regsvr32 命令将组件的类 ID、接口的 GUID 及服务模块等信息注册到注册表中。但对于 WMI 提供器来说，只注册为 COM 服务器还不够，还需要向 WMI 的对象管理器注册，这样 WMI 应用程序才可以通过 WMI 枚举到这个提供器所提供的类或实例等。

向 WMI 注册提供器的简单方法就是先编写一个 MOF 文件，然后使用 mofcomp 命令执行注册操作。其大体步骤如下：

1. 创建一个 MOF 文件，然后指定要注册到的目标命名空间。

```
#pragma namespace("\\\\.\\ROOT\\MyNamespace")
```

以上指令将当前命名空间设定为本地机器的\root\MyNamespace。

2. 创建一个__Win32Provider 类的实例。__Win32Provider 是一个内部类，用于描述提供器对象。可以把创建一个__Win32Provider 实例的过程理解成是在 CIM 对象数据库的__Win32Provider 数据表中增加一行。每个属性的值就是这一行对应列的值。

```
instance of __Win32Provider as "$Reg ;
{
Name = "RegProv";
CLSID = "{fe9af5c0-d3b6-11ce-a5b6-00aa00680c3f}";
HostingModel = "NetworkServiceHost:LocalServiceHost"
};
```

以上语句实质上会在提供器表中增加一行新的记录，该行 Name 列、CLSID 列和 hostingMode 列的内容分别等于指定的值。Name 即该提供器的名称，不需解释。CLSID 是该提供器的 COM 组件的 GUID。COM 注册时，该 GUID 会被写入注册表。WMI 就是通过提供器的 GUID 找到它，然后借助 COM 技术加载对应的模块和创建提供器对象。HostingModel 用来指定提供器的宿主（host）特征，即 WMI 应该以什么样的环境来加载和运行提供器，包括以什么帐号来加载它以及将其加载到哪个进程。目前 WMI 支持如下几种 HostingModel：

- SelfHost：提供器运行在自己的本地 COM 服务器进程（EXE）中。
- LocalSystemHost 、LocalSystemHostOrSelfHost 、NetworkServiceHost 和 LocalServiceHost：如果提供器是以进程内 COM 服务器的方式实现的，那么便运行在共享的提供器宿主进程（wmiprvse.exe）中。否则提供器便是运行在自己的本地 COM 服务器进程（EXE）中。这几个选项的差异是宿主进程是以哪个帐号运行的，LocalSystem 的权限最高，应该仅在当提供器需要访问特权信息时才使用。NetworkService 和 LocalService 的权限都是受限的（limited），如果提供器需要访问远程的机器，那么推荐使用 NetworkService，如果所有操作都是在本地机器上完成，那么推荐使用 LocalService。
- Decoupled:Com：提供器运行在隔离的 WMI 客户进程中。

需要说明的是，HostingModel 设置只适用于 Windows 2000 只后的 Windows，对于 Windows 2000 来说，说有进程内提供器（DLL 形式的提供器）都运行在 winmgmt 进程内。

3. 创建__ProviderRegistration 类或其派生类的实例。WMI 提供了六个__ProviderRegistration 类__ClassProviderRegistration、__InstanceProvider- Registration、__EventProviderRegistration 、__EventConsumerProviderRegistration 、__MethodProviderRegistration、__PropertyProviderRegistration 分别用来注册 6 种类型的提供器。如果一个提供器实现了多个角色，那么便要分别创建多个实例。比如如下语句为 StdRegProv 提供器注册了实例提供器和方法提供器两种身份。

```
instance of __InstanceProviderRegistration
{
   provider = "$Reg";
   SupportsDelete = FALSE;
   SupportsEnumeration = TRUE;
   SupportsGet = TRUE;
   SupportsPut = TRUE;
};
instance of __MethodProviderRegistration
{
   provider = "$Reg";
};
```

4. 将以上内容保存到一个 MOF 文件中并放在一个安全的位置。然后使用 mofcomp 命令编译并执行以上语句。如果一切顺利，那么提供器便顺利注册了。

5. 接下来可以使用我们前面介绍的 WMI CIM Studio 在第一步指定的命名空间中寻找刚注册的提供器（类和实例），检验其是否工作正常。要提醒大家的是，如果遇到 0x80041013 错误，那么其含义是 Provider load failure，即加载提供其失败，这时首先应该检查提供器的 COM 组件是否成功注册以及对应的模块文件是否存在。

31.4.3 WMI 提供器进程

所谓 WMI 提供器进程就是指 WMI 提供器所处的进程，因为少量的 EXE 形式的提供器运行在自己的进程中，不需要 WMI 为其提供宿主进程，所以很多时候（包括本书），WMI 提供器进程是用来泛指承载（host）DLL 形式的 WMI 提供器的进程。

在 Windows 2000 中，WMI 的服务进程（winmgmt.exe）同时承担提供器进程的角色。这样的做的问题是如果某个提供器中的代码发生了错误，那么可能导致整个进程崩溃。Windows XP 对此做了改进，使用 wmiprvse.exe 进程来加载和运行提供器模块。wmiprvse.exe 进程的个数是不确定的，WMI 服务进程会根据需要动态创建 wmiprvse.exe 进程，当不需要时，wmiprvse.exe 进程会退出。所以在任务列表中，有时我们可能看不到 wmiprvse.exe 进程，但是如果运行一个使用 WMI 服务的程序，那么就可以看到这个进程了，过一回可能又会发现这个进程不见了。wmiprvse.exe 进程和 WMI 服务进程之间也是使用 RPC 机制进行通行。清单 31-9 的函数调用序列显示了当执行枚举 InstProvSamp 类实例的脚本时，wmiprvse.exe 进程内的 WMI 提供器类（CInstPro）的方法被调用的过程。

清单 31-9 运行在 wmiprvse.exe 进程内的 InstProvSamp 提供器类被远程调用的典型过程

```
instprov!CreateInst+0x10
instprov!CInstPro::CreateInstanceEnumAsync+0xd3
wmiprvse!CInterceptor_IWbemSyncProvider::Helper_CreateInstanceEnumAsync+0x152
wmiprvse!CInterceptor_IWbemSyncProvider::CreateInstanceEnumAsync+0x70
RPCRT4!CheckVerificationTrailer+0x75
RPCRT4!NdrStubCall2+0x215
RPCRT4!CStdStubBuffer_Invoke+0x82
FastProx!CBaseStublet::Invoke+0x22
ole32!SyncStubInvoke+0x33
ole32!StubInvoke+0xa7
[以下省略多行]
```

如此看来，在 Windows XP，WMI 应程序和 WMI 提供器之间的通信大多要经历两次基于 RPC 机制的进程间通信（参见图 31-11），RPC 通信为分布式管理提供了支持，有着很高的灵活性，但是这也是有代价的，要进行比较复杂的安全检查和数据整理工

图 31-11 WMI 应程序和提供器间的通信过程（Windows XP）

作，因此比一般的本地进程间通信速度要慢很多。另外，因为提供器进程是动态创建的，创建和初始化该进程也需要一些时间。这些因素导致执行到调用 WMI 服务的代码时，有时可以感觉到明显的时间延迟。

31.5 WMI 应用程序

WMI 应用程序通常是指利用 WMI 服务提供各种管理功能的软件工具。WMI 应用程序使通过 WMI 技术构建的管理服务为用户所用、发挥价值。从消耗和提供的角度来看，WMI 应用程序消耗（Consume）WMI 服务提供器提供的信息，因此属于消耗器。

简单来说，Windows 提供了以下四种方式供不同类型的应用程序使用 WMI 服务：

1. COM/DCOM 接口，C/C++程序可以通过这些接口与 WMI 的核心组件通信并调用所需的服务。
2. ActiveX 控件，各种脚本程序可以通过进一步封装过的 ActiveX 控件来调用 WMI 服务。
3. ODBC 适配器（ODBC Adaptor），通过该适配器，可以像访问数据库那样访问 WMI 中的信息。
4. .Net 框架中的 System.Management 类库，.Net 程序可以通过该类库中的各个类使用 WMI 服务。

下面我们分别进行介绍。

31.5.1 通过 COM/DCOM 接口使用 WMI 服务

使用 C/C++调用 WMI 服务是其前面介绍的四种方式中最复杂的一种，因为需要在应用程序中自己初始化 COM 库、创建组件实例、并完成琐碎的清理工作。但是这种方式的优点是灵活性高，执行速度也会快一些。

使用 C/C++调用 WMI 服务的过程归纳为以下几个步骤。

初始化 COM 库

因为 WMI 服务都是以 COM 组件的形式建立的，所以访问 WMI 服务的过程实际上就是创建和使用 COM 组件的过程。在使用 COM API 创建 COM 对象之前，必须进行初始化，可以使用 CoInitializeEx 函数或 OleInitialize 函数来初始化 COM 库。

初始化进程安全属性

因为很多 WMI 服务都定义了安全控制，所以通常还需要调用 CoInitializeSecurity 来设置当前进程的默认安全属性。

```
hres =  CoInitializeSecurity(
   NULL,
   -1,                              // COM negotiates service
   NULL,                            // Authentication services
   NULL,                            // Reserved
   RPC_C_AUTHN_LEVEL_DEFAULT,       // Default authentication
   RPC_C_IMP_LEVEL_IMPERSONATE,     // Default Impersonation
   NULL,                            // Authentication info
   EOAC_NONE,                       // Additional capabilities
   NULL                             // Reserved
   );
```

连接 WMI 服务

在使用 WMI 服务（类或提供器）前，必须先与该服务所在的命名空间建立连接，目的是获得一个实现了 IWbemServices 接口的 WBEM 服务实例。可以使用 IWbemLocator 接口的 ConnectServer 方法来实现这一目的。这先需要创建一个实现了 IWbemLocator 接口

的组件。

```
IWbemLocator *pIWbemLocator = NULL;
   if(CoCreateInstance(CLSID_WbemLocator,
      NULL, CLSCTX_INPROC_SERVER,
      IID_IWbemLocator, (LPVOID *) &pIWbemLocator) == S_OK)
   {
      // Using the locator, connect to CIMOM in the given namespace.
      if(pIWbemLocator->ConnectServer(pNamespace,
                           NULL,       //using current account for simplicity
                           NULL,       //using current password for simplicity
                           0L,         // locale
                           0L,         // securityFlags
                           NULL,       // authority (domain for NTLM)
                           NULL,       // context
                           &m_pIWbemServices) != S_OK)
      {
         // error handling
      }
      pIWbemLocator->Release();
   }
```

设置 WMI 连接的安全等级（security level）

因为 WMI 应用程序是通过 WMI 组件的代理（proxy）来访问进程外的 WMI 服务的，所以应该设置合适的认证信息供代理进行 RPC 调用时使用。这可以通过调用 CoSetProxyBlanket API 来完成。

```
   hres = CoSetProxyBlanket(
      pSvc,                           // Indicates the proxy to set
      RPC_C_AUTHN_WINNT,              // RPC_C_AUTHN_xxx
      RPC_C_AUTHZ_NONE,               // RPC_C_AUTHZ_xxx
      NULL,                           // Server principal name
      RPC_C_AUTHN_LEVEL_CALL,         // RPC_C_AUTHN_LEVEL_xxx
      RPC_C_IMP_LEVEL_IMPERSONATE,    // RPC_C_IMP_LEVEL_xxx
      NULL,                           // client identity
      EOAC_NONE                       // proxy capabilities
   );
```

执行应用逻辑

在成功连接 WMI 服务并完成代理安全设置后，便可以调用 IWbemServices 接口中的方法来执行应用程序自身的逻辑了。 比如通过 CreateInstanceEnum 方法创建一个实例枚举（IEnumWbemClassObject）对象，然后枚举出某个类的所有实例。再比如，可以通过 GetObject 方法取得命名空间中的类对象，然后创建类的实例（SpawnInstance），读取类的属性，执行类的方法（ExecMethod）等。概而言之，就是利用 WMI 服务暴露出的 COM 接口来访问和操作 WMI 对象。

WMI SDK（现在是 Platform SDK 的一部分）的 Samples\Sysmgmt\VC 目录下给出了几个使用 C++编写的 WMI 应用程序的例子，MSDN 文档中也给出了一些例子，大家可以参考，在此从略。

清理工作

包括释放创建的对象（Release），调用 CoUninitialize 清理 COM/DCOM 库所分配的资源等。

31.5.2 WMI 脚本

使用 WMI 服务的更简单和更常用方法是编写脚本。因为 Windows 提供了可以访问 WMI 的 ActiveX 控件，所以任何支持 ActiveX 控件的脚本语言都可以使用 WMI 服务，比

如 VBScript、Microsoft Jscript 和 Perl。可以执行 WMI 脚本的常见环境（解释器）有 Windows Script Host (WSH)、ASP（Active Server Pages）网页（IIS 服务器）和 IE 浏览器等。

下面先举几个例子让大家认识一下 WMI 脚本。将清单 31-10 所示的使用 VBScript 编写的脚本保存为一个 process_list.vbs 文件。然后在命令行下执行 cscript process_list.vbs，便可以列出当前系统中所运行的所有进程。

清单 31-10 process_list.vbs

```
for each Process in GetObject("winmgmts:{impersonationLevel=impersonate}")._
 InstancesOf("Win32_process")
   WScript.Echo Process.Name
Next
```

每次执行清单 31-11 所示的 cerate_process 脚本都会启动一个记事本程序，通过这个脚本我们演示了如何执行一个带有输入参数的方法。

清单 31-11 create_process.vbs

```
Set objWMIService = GetObject("winmgmts:{impersonationLevel=impersonate}")
Set objW32Process = objWMIService.Get("Win32_Process")
Set objInParam = objW32Process.Methods_("Create")._
   inParameters.SpawnInstance_()

' Add the input parameters.
objInParam.Properties_.Item("CommandLine") =  "notepad.exe"

' Execute the method and obtain the return status.
Set objOutParams = objWMIService.ExecMethod("Win32_Process", "Create", objInParam)

Wscript.Echo "Out Parameters: "
Wscript.echo "ProcessId: " & objOutParams.ProcessId
Wscript.echo "ReturnValue: " & objOutParams.ReturnValue
```

最后再看看如何监听 WMI 事件，清单 31-12 所示的脚本注册接收记事本（notepad.exe）进程启动（Win32_ProcessStartTrace）事件。

清单 31-12 pmon.vbs

```
Set EventSource = GetObject("winmgmts:{impersonationLevel=impersonate}")._
 ExecNotificationQuery("select * from Win32_ProcessStartTrace"&_
 " where ProcessName='notepad.exe'")

While true
   Set objEvent = EventSource.NextEvent
   WScript.Echo "NotePad starts with PID = "& objEvent.ProcessID
Wend
```

31.5.3 WQL

SQL（结构化查询语言）是数据库软件中广泛使用的一种数据查询语言，利用它可以定义数据库对象（这部分被称为 Data definition Language，简称 DDL），操作数据库中的数据（这部分被称为 Data Manipulation Language, 简称 DML），以及控制数据库的安全（这部分被称为 Data Contorl Language，简称 DCL）。SQL 简单易懂，功能强大，被几乎所有关系数据库管理系统（RDBMS）所采用。用于不同数据库系统的 SQL 的大多是对 ANSI 标准定义（SQL-92）的扩展，基本语法是相同的，但是某些方面会有所差异，比如微软 SQL Server 数据库所使用的 T-SQL。那么可以不可以使用 SQL 来查询 WMI 中的数据呢？可以，这便是用于 WMI 的 SQL，被称为 WQL。

WQL（WMI Query Language）是标准 SQL 的一个子集，并加入了少量的变化。目前，WQL 只实现了 SQL 语言中的 DDL 部分（数据操纵）所定义的一部分功能，主要是提取数据。

WMI 的很多内部类都对执行 WQL 查询提供了强大的支持。通过 IWbemServices 接口的 ExecQuery 方法便可以提交 WQL 查询。因此只要先取得或创建一个实现了 IWbemServices 接口的对象，然后就可以通过它执行查询了。与数据库查询都是相对于当前数据库类似，WQL 查询都是相对于当前命名空间的。因此要先连接到一个命名空间，成功连接到命名空间就会得到一个实现了 IWbemServices 接口的对象（参考前面对 ConnectServer 方法的介绍），恰好就可以使用它执行查询了。事实上，连接命名空间会导致 WMI 创建一个 CWbemNamespace 实例（参见清单），该实例便是实现了 IWbemServices 接口的组件对象。清单 31-13 所示的脚本演示了如何使用 VbScript 来连接一个命名空间，取得 WbemService 对象，然后执行 WQL 查询的过程。

清单 31-13 使用 VBScript 执行 WQL 查询

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:\\" & strComputer & "\root\CIMV2")
Set colItems = objWMIService.ExecQuery( _
    "SELECT * FROM Win32_NTLogEvent",,48)
For Each objItem in colItems
    Wscript.Echo "-----------------------------------"
    Wscript.Echo "Win32_NTLogEvent instance"
    Wscript.Echo "Message: " & objItem.Message
Next
```

容易想到，通过 COM API 也可以得到 WbemService 对象然后执行 WQL 查询，Windows XP 自带的 WbemTest 工具（wbemtest.exe）就是一个这样的小程序。启动后连接到一个命名空间，然后点击 Query 按钮就可以执行 WQL 查询了（图 31-12），下面我们便利用这个工具带领大家通过几个试验学习 WQL。

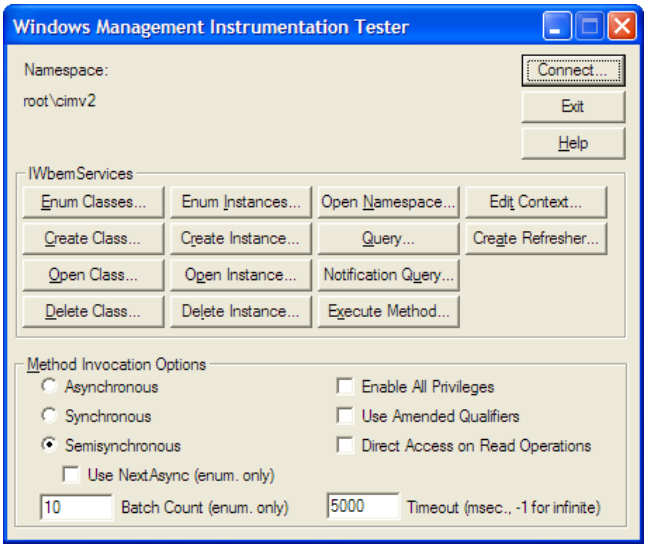


图 31-12 WbemTest 工具

首先我们来看一下最常用的 SELECT 语句，在查询对话框的编辑区输入如下查询，然后点击 Apply 按钮开始执行（参见图 31-13）。

```
select * from meta_class where __class like "%Win%"
```

这个查询的作用是列出当前命名空间中所有类名里包含 Win 字样的所有类。可以把 meta_class 理解为包含了当前命名空间（相当于数据库）内所有类描述的一张内部数据表。__class 是 WQL 的一个关键字，可以把理解为 meta_calss 数据表中的一个内部列。

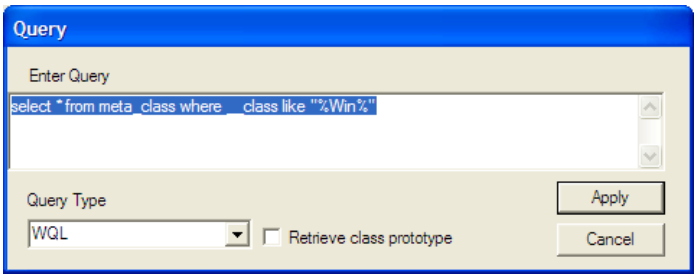


图 31-13　使用 WbemTest 工具执行 WQL 查询

下面再来看一下 WQL 特有的 ASSOCIATORS OF 和 REFERENCES OF 语句，它们都是用来提取与指定实例存在关联关系的实例（association instances）。但不同的是，ASSOCIATORS OF 得到的实例是都是关联关系的端点（endpoint），而 REFERENCES OF 语句得到的是中继（intervening）。举例来说，首先输入并执行如下语句：

```
ASSOCIATORS OF {Win32_LogicalDisk.DeviceID="C:"}
```

在笔者机器上得到的结果如下：

```
Win32_Directory.Name="C:\\"
Win32_QuotaSetting.VolumePath=”C:\\”
Win32_DiskPartition.DeviceID="Disk #0, Partition #0"
Win32_ComputerSystem.Name="ADVDBG2"
```

以上每行代表一个对象实例，其格式为“类名.键名=键值”，即这个类的键值等于指定值的实例与源实例存在关联关系。

接下来输入并执行以下语句，看看使用 REFERENCES OF 语句的结果。

```
REFERENCES OF {Win32_LogicalDisk.DeviceID="C:"}
```

在笔者机器上得到的结果如下：

```
Win32_VolumeQuotaSetting.Element="\\\\SHXPLYZHAN31\\root\\cimv2:Win32_LogicalDis
k.DeviceID=\"C:\"",Setting="\\\\SHXPLYZHAN31\\root\\cimv2:Win32_QuotaSetting.Vol
umePath=\"C:\\\\\\""
[以下多行省略]
```

使用我们前面介绍的 CIM Sutdio 工具可以观察到 Win32_LogicalDisk、Win32_VolumeQuotaSetting 和 Win32_QuotaSetting 这三个类的关联关系。从图中可以看到 Win32_QuotaSetting 是该关联关系的端点，而 Win32_VolumeQuotaSetting 类是联系着两个类的一个中继类。这与前面的执行结果正好符合，ASSOCIATORS OF 的结果包含了 Win32_QuotaSetting 的实例，因为它是端点关联实例；REFERENCES OF 的结果包含了 Win32_VolumeQuotaSetting 的实例，因为它是中间的引用类。

从 Win32_VolumeQuotaSetting 类的定义中也可以看到它的“纽带特征”，Element 属性指向 Win32_LogicalDisk，Setting 属性指向 Win32_QuotaSetting。

```
[dynamic: ToInstance, provider("DskQuotaProvider"): ToInstance, Locale(1033):
ToInstance, UUID("FA452BCE-5B4F-4A56-BF52-7C4533984706"): ToInstance]
class Win32_VolumeQuotaSetting : CIM_ElementSetting
{
   [read: ToSubClass, key, Override("Element")] Win32_LogicalDisk ref Element = NULL;
   [read: ToSubClass, key, Override("Setting")] Win32_QuotaSetting ref Setting =
NULL;
};
```

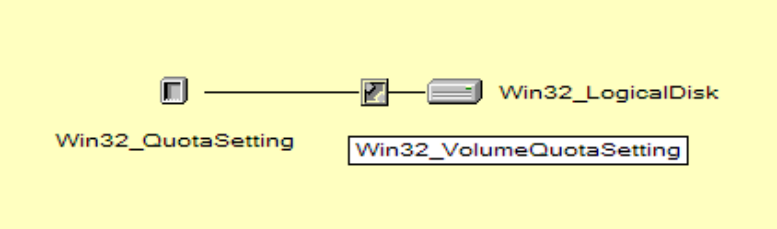


图 31-14　Win32_LogicalDisk 和 Win32_QuotaSetting 类的关联关系

31.5.4　WMI 代码生成器

从微软网站可以免费下载一个名为 WMI Code Writer 的工具，使用该工具可以交互式的生成各种 WMI 代码，比如浏览命名空间、执行查询、执行方法和接收事件（参见图 31-15）。

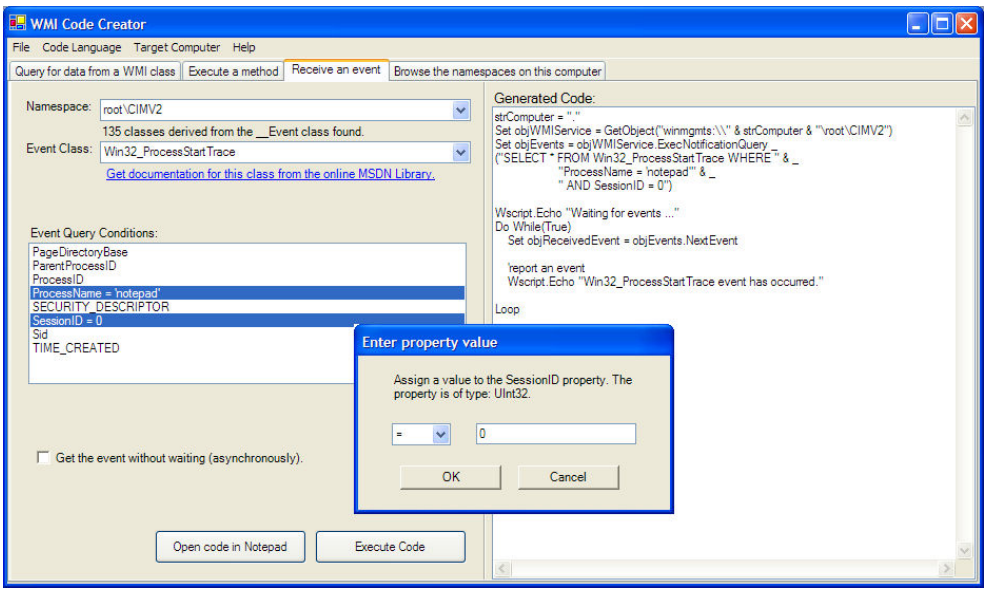


图 31-15　WMI 代码生成器

WMI Code Writer 支持生成三种语言的代码：VBScript、VB.Net 和 C#。清单 31-13 中的 VBScript 脚本就是该工具自动生成的。WMI Code Writer 是使用 C#语言编写的，下载的压缩包中包含了完整的源代码，感兴趣的读者可以仔细读一读。

31.5.5　WMI ODBC 适配器

ODBC（Open Database Connectivity）是数据库领域中一种广泛应用的调用层（call level）接口，通过 ODBC，应用程序可以使用同一套 API 来访问各种格式的数据库，只要为其安装了 ODBC 驱动程序。WMI 的 CIM 数据仓库也是一种数据库，因此只要为其配备了驱动程序，那么各种数据库应用程序就可以使用 ODBC API 来访问 WMI 中的信息了，而不必使用 COM/DCOM。WMI ODBC 适配器（Adapter）就是按照这一思想设计的，通过 WMI ODBC 适配器，应用程序可以像访问数据库一样访问 CIM 数据仓库。从 ODBC 架构角度来看，WMI ODBC 适配器启动的就是 ODBC 驱动程序的作用。

在 Windows XP 和 2000 的默认安装中都不含 WMI ODBC 适配器，但可以在安装光盘的如下目录中找到它的安装程序。

```
VALUEADD\MSFT\MGMT\WBEMODBC
```

运行以上目录中的 wbemodbc.msi 后，从控制面板的 ODBC 数据源管理对话框中就可以看到 WBEM ODBC 驱动程序了（如图 31-16）。

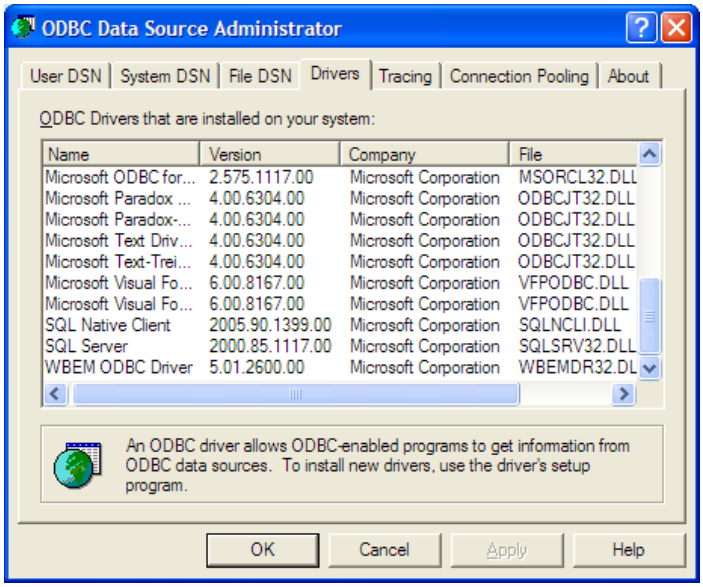


图 31-16　WBEM ODBC 驱动程序

有了 ODBC 驱动程序，就可以通过 DSN（Data Source Name）或者连接字符串（Connection String）来访问 CIM 数据了。其细节请参看 MSDN 中关于 ODBC 编程的部分。需要说明的是，通过 WMI ODBC 适配器所访问到的 CIM 信息是只读的，而且不支持 Unicode。

31.5.6　在.Net 程序中使用 WMI

前面我们讲过，通过 COM API 来访问 WMI 服务是最灵活高效的方式，因为 WMI 本身就是使用 COM 技术构建的。利用 COM Interop 技术，.Net 程序可以访问和使用 COM 组件，COM 组件也可以像使用其它 COM 组件那样访问.Net 语言编写的组件。因此，.Net 程序完全可以通过 COM Interop 来使用 WMI（WMI Consumer），也可以成为 WMI 的提供器。为了使.Net 程序可以更简单使用 WMI，.Net Framework 提供了一个类库来封装 COM Interop 的细节，让程序员只要通过这些托管类（managed class）就可以使用 WMI。这些类主要位于 System.Management 和 System.Management. Instrumentation 两个 namespace 中，前者主要供使用 WMI（WMI 消耗器）的程序使用，后者供向 WMI 提供事件或数据（WMI 提供器）的程序使用，它们都被包含在 System.Management.dll 内。

清单 31-14 给出了一个使用 C#语言编写的小程序，它可以通过 WMI 列出当前系统中运行着的所有进程。\code\chap31\c#\WmiClientCS\目录下包含了完整的项目文件。因为使用了 System.Management 类，所以应该向项目中加入对 System.Management.dll 的引用（Add Reference）。

清单 31-14 在.Net 程序中使用 WMI 的简单示例

```
1  using System;
2  using System.Management;
3
4  namespace WmiClientCS
5  {
6     class Program
7     {
8        static void Main(string[] args)
9        {
```

```
10          string searchQuery = "SELECT * FROM Win32_Process";
11          ManagementObjectSearcher searchPrinters =
12               new ManagementObjectSearcher(searchQuery);
13          ManagementObjectCollection printerCollection = searchPrinters.Get();
14          foreach (ManagementObject printer in printerCollection)
15          {
16             Console.WriteLine(printer.Properties["Name"].Value.ToString());
17          }
18       }
19     }
20 }
```

31.6 调试 WMI

WMI 是个复杂的系统，种类繁多的 WMI 应用程序和 WMI 提供器丰富了 WMI 的功能，但同时也增加了整个系统的复杂度。在一个典型的 WMI 系统中，各种 COM/DCOM 组件在多个进程间相互调用，RPC 通信和异步调用非常普遍，这些因素都为调试 WMI 有关的问题增加了难度。好在 WMI 在设计时便考虑到了调试问题，内建了很多调试支持：

1. WMI 的所有核心模块都具有日志记录功能，可以将重要的操作和异常情况记录到日志文件中。
2. WMI 内部包含了很多专门用于调试的类，包括用于对各种操作计数的 Msft_WmiProvider_Counters 类，专门用于产生调试事件的 MSFT_WmiSelfEvent 类和它的数十个派生类。
3. 尽管 WMI 自身的基础模块和类很多，而且微软的文档中没有介绍这些内部类，但是借助调试符号，我们还是可以比较容易的理解出每个类的作用，各个类之间的关系等。从函数和属性的名称也基本可以推测出它的含义，从而可以设置断点或跟踪执行。这为追踪某些复杂的 WMI 问题提供了条件。

   下面我们对 1 和 2 两点进行详细介绍。

31.6.1 WMI 日志文件

默认情况下，WMI 的日志文件被保存在%windir%\system32\wbem\logs 目录下。因为不同功能的 WMI 模块会使用不同的的日志文件，所以在以上目录中，通常可以看到十几个文件，表 31-10 列出了这些文件的名称和用途。

表 31-10 WMI 日志文件的名称和简要介绍

文件名	描述
Dsprovider.log	Trace information and error messages for the Directory Services Provider.
Framework.log	Trace information and error messages for the provider framework and the Win32 Provider.
Mofcomp.log	Compilation details from the MOF compiler.
Ntevt.log	Trace messages from the Event Log Provider.
Setup.log	Report on those MOF files that failed to load during the setup process. However, the error that caused the failure is not reported. You must review the Mofcomp.log file to determine the reason for the failure. After the error has been corrected, you can recompile the MOF file (using mofcomp) with the -autorecover switch.
Viewprovider.log	Trace information from the View Provider. This provider requires that you set a bit value for the following registry key. HKEY_LOCAL_MACHINE \SOFTWARE\Microsoft\WBEM\ PROVIDERS\Logging\ViewProvider\Level
Wbemcore.log	Wide spectrum of trace messages.
Wbemess.log	Log entries related to events.
Wbemprox.log	Trace information for the WMI proxy server.

Winmgmt.log	Trace information that is typically not used for diagnostics.
Wmiadap.log	Error messages related to the AutoDiscoveryAutoPurge (ADAP) process.
Wmiprov.log	Management data and events from WMI-enabled Windows Driver Model (WDM) drivers.

WMI 的日志功能是可以配置的，WMI 核心模块的日志选项保存在如下注册表表键下：

```
HKEY_LOCAL_MACHINE \SOFTWARE\Microsoft\WBEM\CIMOM\
```

可以配置的选项如表 31-11 所示。

表 31-11　WMI 核心模块的日志选项

键值名称	类型	功能和默认值
Logging	REG_SZ	记录级别，可以为 0（不记录），1（仅记录错误），2（详细记录）三个级别。默认为 2。
Logging Directory	REG_SZ	日志文件路径，默认为%windir%\system32\wbem\logs。
Log File Max Size	REG_SZ	日志文件最大长度（字节数），默认为 65536 字节。

各个 WMI 提供器的日志选项保存在 PROVIDERS\Logging 子键下的以提供器名称命名的表键下，比如 NTEVT 提供器的日志配置选项存储在如下位置：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WBEM\PROVIDERS\Logging\NTEVT
```

对于提供器，可以设置表 31-12 所列出的 4 种选项。

表 31-12　WMI 提供器的配置选项

键值名称	类型	功能
File	REG_SZ	日志文件的完整路径和文件名。比如 C:\WINDOWS\system32\WBEM\Logs\\NTEVT.log
Level	REG_DWORD	用来定义调试信息输出的 32 位的掩码（bit mask），与提供器的实现有关。
MaxFileSize	REG_DWORD	日志文件最大长度（字节数），默认为 65536 字节。
Type	REG_SZ	日志输出的类型，可以设为“File”（写到文件）和“Debugger”（输出到调试器）两种。

31.6.2　WMI 的计数器类

WMI 内部专门设计了一些类用来收集状态和调试信息，我们先来看一下用于对各种操作计数的 Msft_WmiProvider_Counters 类

Msft_WmiProvider_Counters 类的作用是对各种常见的函数调用进行计数。这个类没有方法，只有 24 个属性，全都是 64 位的无符号整数（uint64），用作计数器（Counter）。表 31-13 列出了这些属性的名称和它们各自要记录的操作（函数调用）。

表 31-13　WMI 的计数器（Msft_WmiProvider_Counters 类的成员）

属性	记录的调用
ProviderOperation_AccessCheck	IWbemEventProviderSecurity::AccessCheck
ProviderOperation_CancelQuery	IWbemEventProviderQuerySink::CancelQuery
ProviderOperation_CreateClassEnumAsync	IWbemServices::CreateClassEnumAsync
ProviderOperation_CreateInstanceEnumAsync	IWbemServices::CreateInstanceEnumAsync
ProviderOperation_CreateRefreshableEnum	IWbemHiPerfProvider::CreateRefreshableEnum
ProviderOperation_CreateRefreshableObject	IWbemHiPerfProvider::CreateRefreshableObject
ProviderOperation_CreateRefresher	IWbemHiPerfProvider::CreateRefresher
ProviderOperation_DeleteClassAsync	IWbemServices::DeleteClassAsync
ProviderOperation_DeleteInstanceAsync	IWbemServices::DeleteInstanceAsync
ProviderOperation_ExecMethodAsync	IWbemServices::ExecMethodAsync

属性	记录的调用
ProviderOperation_ExecQueryAsync	IWbemServices::ExecQueryAsync
ProviderOperation_FindConsumer	IWbemEventConsumerProvider::FindConsumer
ProviderOperation_GetObjectAsync	IWbemServices::GetObjectAsync
ProviderOperation_GetObjects	IWbemHiPerfProvider::GetObjects
ProviderOperation_GetProperty	IWbemPropertyProvider::GetProperty
ProviderOperation_NewQuery	IWbemEventProviderQuerySink::NewQuery
ProviderOperation_ProvideEvents	IWbemEventProvider::ProvideEvents
ProviderOperation_PutClassAsync	TIWbemServices::PutClassAsync
ProviderOperation_PutInstanceAsync	IWbemServices::PutInstanceAsync
ProviderOperation_PutProperty	IWbemPropertyProvider::PutProperty
ProviderOperation_QueryInstances	IWbemHiPerfProvider::QueryInstances
ProviderOperation_SetRegistrationObject	未实现
ProviderOperation_StopRefreshing	TIWbemHiPerfProvider::StopRefreshing
ProviderOperation_ValidateSubscription	IWbemEventProviderSecurity::AccessCheck

WMI 已经为 Msft_WmiProvider_Counters 类定义了一个实例，位于\root\CIMV2 命名空间中，因此只要使用清单 31-15 所示的 VBScript 就可以得到各个计数器的值了。

清单 31-15 列出所有 WMI 计数器值的脚本

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:\\" & strComputer & "\root\CIMV2")
Set colItems = objWMIService.ExecQuery( _
   "SELECT * FROM Msft_WmiProvider_Counters",,48)
For Each objItem in colItems
   Wscript.Echo "-----------------------------------"
   Wscript.Echo "Msft_WmiProvider_Counters instance"
   Wscript.Echo "-----------------------------------"
Wscript.Echo "ProviderOperation_AccessCheck: " &
objItem.ProviderOperation_AccessCheck
' 排版时省略多行，完整脚本位于 chap31\script\counters.vbs
Next
```

31.6.3 WMI 的故障诊断类

WMI 定义了很多用于汇报重要 WMI 事件的事件类，供调试和故障诊断使用。这些类被统称为故障诊断类（troubleshooting class），它们都派生自一个共同的基类——MSFT_WmiSelfEvent 类。根据事件来源，这些类被划分为三个组，分别被称为 WMI 服务事件诊断类、提供器事件诊断类和事件消耗器提供器诊断类。

WMI 服务事件诊断类

WMI 服务事件诊断类用于报告 WMI 内部服务模块中的发生的重要事件，比如创建线程池（MSFT_WmiThreadPoolCreated）、激活和解除事件过滤器（MSFT_WmiFilterActivated 和 MSFT_WmiFilterDeactivated）、订阅事件和取消订阅（MSFT_WmiRegisterNotificationEvent 和 MSFT_WmiCancelNotificationSink）等等。所有服务事件诊断类都派生自 MSFT_WmiEssEvent 类，ESS 的含义是事件子系统（Event Sub-system）。因此可以通过订阅 MSFT_WmiEssEvent 类来接受所有服务诊断类事件。

提供器事件诊断类

WMI 调试的最常见任务就是调试 WMI 提供器，很多 WMI 故障都与某个（些）提供器有关。因此 WMI 内建了 30 几个事件类来报告各种与提供器有关的操作，包括安全检查、

执行和取消查询、加载 COM 服务器、存取对象等等。而且对于大多数操作，都设计了两个事件类，分别报告该操作执行前和执行后的状态，如对于取消查询操作，就有两个事件类，MSFT_WmiProvider_CancelQuery_Pre 和 MSFT_WmiProvider_CancelQuery_Post。前者会在提供器的 IWbemEventProviderQuerySink::CancelQuery 被调用前报告，后者在提供器的方法被执行后报告。

因为篇幅有限，我们无法一一介绍每个提供器事件类，大家在使用时可以参阅 SDK 文档。唠届下面我们通过一个试验来介绍 MSFT_WmiProvider_LoadOperationFailure- Event 类。

MSFT_WmiProvider_LoadOperationFailureEvent 类定义了激活或初始化提供器失败事件。我们可以使用清单 31-16 所示的脚本订阅并监听此事件。

清单 31-16 订阅并监听此加载提供器失败事件（LoadFailure.vbs）

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:\\" & strComputer & "\root\CIMV2")
Set objEvents = objWMIService.ExecNotificationQuery _
("SELECT * FROM Msft_WmiProvider_LoadOperationFailureEvent")

Wscript.Echo "Waiting for events ..."
Do While(True)
    Set objReceivedEvent = objEvents.NextEvent

    'report an event
    Wscript.Echo "-----------------------------------------------------------"
    Wscript.Echo "Msft_WmiProvider_LoadOperationFailureEvent event has occurred."
    Wscript.Echo "ServerName: " & objReceivedEvent.ServerName
    Wscript.Echo "InProcServer: " & objReceivedEvent.InProcServer
    Wscript.Echo "InProcServerPath: " & objReceivedEvent.InProcServerPath
    Wscript.Echo "ThreadingModel: " & objReceivedEvent.ThreadingModel
```

```
    Wscript.Echo "ResultCode: " & objReceivedEvent.ResultCode
Loop
```

可以通过如下实验来感受该事件的产生条件和作用。

1. 编译 WMI SDK 中的 ClassProv 项目，chap31\ClassProv 目录下有该示例的一个副本。
2. 开启一个命令行窗口，转到编译好的 ClassProv.DLL 文件所在目录，键入 regsvr32 classprov.dll 对其进行注册。
3. 转到项目文件所在目录，确认存在 ClassProv.MOF 文件，然后键入 mofcomp classprov.mof 向 CIM 对象管理器注册此提供器。
4. 运行 WbemTest 工具，连接到\root\default 命名空间，然后选择 Open Class 按钮并键入 ClassProvSamp，确认该类已经存在。
5. 停止 WMI 服务（请在测试机器上实验，以免导致意外损失），可以在服务面板操作，也可以在命令行键入 net stop winmgmt。
6. 将刚刚注册的 ClassProv.DLL 文件删除。
7. 将命令行窗口的当目录切换到 chap31\script\，然后键入 cscript loadfailure.vbs 执行该脚本。如果成功，应该看到如下屏幕输出。

```
c:\dig\dbg\author\code\chap31\script>cscript loadfailure.vbs
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. All rights reserved.

Waiting for events ...
```

8. 重复第 4 步的打开 ClassProvSamp 类操作，因为 ClassProvSamp 类的提供器模块已经仔第 6 步被删除，所以 Wbemtest 会显示出图 31-17 所示的错误对话框。

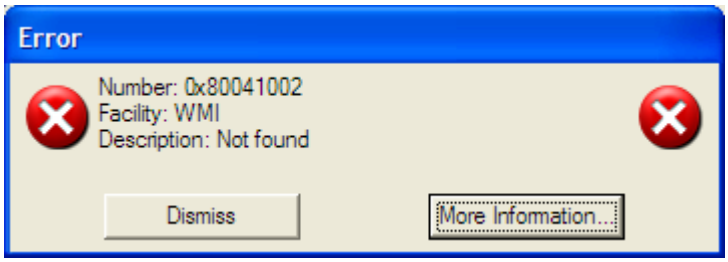


图 31-17　加载提供器失败对话框

9. 此时观察运行脚本的命令行窗口，应该看到类似如下的输出：

```
-----------------------------------------------------------
Msft_WmiProvider_LoadOperationFailureEvent event has occurred.
ServerName: WINMGMT Sample Class Provider
InProcServer: True
InProcServerPath: c:\dig\dbg\author\code\chap31\ClassProv\Debug\classprov.dll
ThreadingModel: 1
ResultCode: -2147024770
```

结果码-2147024770 对应的 16 进制值为 0x8007007E，使用 Error Lookup 工具可以查到其含义为 “The specified module could not be found.”，即没有找到指定的模块。

事件消耗器提供器诊断类

前面我们介绍过，事件消耗器提供器的作用是判断应该使用哪一个永久的（permanent）事件消耗器来处理给定的事件，也就是将物力消耗器映射到逻辑消耗器。与普通 WMI 提供器一样，事件消耗器提供器也是以 COM 服务器的形式工作的，WMI 会根据需要动态加载这些提供器。为了调试事件消耗器提供器，WMI 设计了四个事件类来

报告事件消耗器提供器的加载（MSFT_WmiConsumerProviderLoaded）、卸载（MSFT_WmiConsumerProviderLoaded）以及成功激活（MSFT_WmiConsumer-ProviderSinkLoaded）和解除（MSFT_WmiConsumerProviderSinkUnloaded）事件接插点（sink）对象。这四个事件类都派生自 MSFT_WmiProviderEvent。

31.6 本章总结

本章花较大的篇幅比较全面的介绍 Windows 系统中应用最广泛的管理（administration）机制——WMI。我们的主要目的有三：

第一、尽管 WMI 主要是一种管理机制，由于它所建立的访问受管对象的强大设施和丰富工具也为调试受管对象提供了宝贵的资源。Windows 系统的很多重要部件和很多系统服务都支持 WMI（参见 31.4 节），使用 WMI 可以探测这些部件和服务的内部信息，接收事件，或者执行操作。

第二、从软件开发的角度来看，支持 WMI 有助于提高软件的可维护性和可调试性（Debuggability）。本章介绍 WMI 的工作原理也是为本书第 5 部分讨论软件的可调试性奠定基础。

第三、WMI 是 Windows 操作系统的一个重要部分，学习和深入理解 WMI 是全面理解 Windows 系统的一门必修课。熟悉 WMI 的基础部件和提供器，熟练使用 WMI 脚本和工具可以丰富我们的知识库、有助于提高调试技能。

补编内容 9 CPU 异常逐一描述

补编说明：

这一章本来是《软件调试》的第一个附录，描述的是 CPU 的每个异常。

《软件调试》的一条主线就是异常，全书按从底层到顶层的顺序反复介绍了异常。CPU 的异常（有时也叫硬件异常）是很多程序员感觉陌生的，因此《软件调试》的第 2 篇对此做了比较全面的介绍。这个附录是配合第 2 篇内容的，更详细的介绍了每一种 CPU 异常。

在邻近出版进行编辑时，这一附录被删除了，目的是为了控制篇幅，删除的原因主要是这一内容的难度比较大，一般的程序员可能望而却步，不会仔细阅读。

附 录 C

CPU 异常详解

本书第 3 章介绍了 IA-32 CPU 的异常，并通过表 3-2 列出了 IA-32 处理器迄今为止所定义的所有异常。本附录将以向量号为顺序逐一介绍每一种异常，详细描述它的以下属性：

- 向量号：该异常的向量号。
- 异常类型：即该异常属于错误类异常、陷阱类异常还是中止类异常。
- 相关处理器：最早引入该异常的处理器，以及其它处理器对该异常的实现情况。
- 描述：该异常的产生原因、用途等信息。
- 错误代码：CPU 在产生某些异常时，会向栈中压入一个 32 位的错误代码。
- 保存的程序指针：CPU 在产生异常时会向栈中压入程序指针。
- 程序状态变化：该异常对程序状态的影响，是否可以安全的恢复重新执行。

本附录的内容主要参考了 IA-32 手册和 Tom Shanley 所著的《The Unabridged Pentium 4》一书。

C.1 除零异常（#DE）

向量号：0

异常类型：错误（Fault）

引入该异常的处理器：8088 最早引入该异常，其后的 8086、80286 以及所有 IA-32 处理器都实现了该异常。

描述：CPU 在执行 DIV（无符号除法）或 IDIV（带符号除法）指令时如果检测到以下情况则会产生该异常：

- 除数为 0。
- 商值太大，无法用目标运算符表示出来。

错误代码：CPU 不会向栈中放入错误代码。

保存的程序指针：栈中保存的 CS 和 EIP 值指向的是导致该异常的指令。

程序状态变化：该异常不会改变程序状态，CPU 会将处理器状态恢复到执行该指令之前的状态。所以当错误情况被纠正后（比如除数改为非 0）可以安全的重新执行。

C.2 调试异常（#DB）

向量号：1

异常类型：陷阱（Trap）或错误（Fault），异常处理例程可以根据 DR6 和其它调试寄存器的内容判断，参见第 4 章的表 4-2。

引入该异常的处理器：8088 最早引入该异常，其后的 8086、80286 以及所有 IA-32 处理器都实现了该异常。

描述：此异常表示 CPU 已将检测一或多个调试事件的触发条件被满足。

错误代码：CPU 不会向栈中放入错误代码。

保存的程序指针：对于错误类异常，栈中保存的 CS 和 EIP 值指向的是导致该异常的指令。

对于陷阱类异常，栈中保存的 CS 和 EIP 值指向的是导致异常指令执行完接下来该执行的指令。注意，CS 和 EIP 指向的不一定是地址相邻的下一条指令，比如如果在执行 JMP 指令时发生了陷阱异常，那么 CS：EIP 指向的是 JMP 指令的目标地址。

程序状态变化：错误类异常不会改变程序状态，CPU 会将处理器状态恢复到执行该指令之前的状态。所以程序可以安全的从异常处理例程返回并正常执行。

陷阱类调试异常伴随有程序状态变化，因为在异常产生前正在执行指令或任务切换会执行完毕。不过，程序状态不会被破坏，可以继续安全的执行。

C.3 不可屏蔽中断（NMI）

向量号：2

异常类型：不属于异常，属于中断。

引入该异常的处理器：8088 最早引入该中断（不是异常），其后的 8086、80286 以及所有 IA-32 处理器都实现了该中断。

描述：硬件中断，是由于外部硬件通过 CPU 的 NMI 管脚发出中断请求或者系统的 I/O APIC 向 CPU 内的本地 APIC 发送了 NMI 中断消息。

错误代码：N/A（不适用）

保存的程序指针：栈中保存的 CS 和 EIP 值指向的是响应中断前接下来要执行的那条指令。

程序状态变化：CPU 总是在指令边界（instruction boundary）响应 NMI 中断，也就是当 CPU 接收到 NMI 时如果有指令正在执行，那么它会执行完正在执行的指令。因此，当中断处理程序处理结束后可以安全的返回到被中断的程序或任务继续执行。

C.4 断点异常（#BP）

向量号：3

异常类型：陷阱（Trap）

引入该异常的处理器：8088 最早引入该异常，其后的 8086、80286 以及所有 IA-32 处理器都实现了该异常。

描述：当 CPU 执行 INT 3 指令时会产生此异常。INT n（n=3）指令也会导致该异常，但是二者的机器码不同，功能也有微小差异。

INT 3 指令是常用的软件断点的基础，当用户在调试器中设置断点时，调试器会用 INT 3 指令替换掉断点位置的那条指令的第一个字节。参见本章第 2 节。

错误代码：CPU 不会向栈中放入错误代码。

保存的程序指针：栈中保存的 CS 和 EIP 值指向的是 INT 3 指令后面要执行的那条指令。

程序状态变化：尽管栈中的 CS 和 EIP 指向的是下一条指令，但是因为 INT 3 指令不会影响任何寄存器或内存数据，程序状态并没有实质性变化，所以调试器可以先将被 INT 3 替换掉的指令首字节恢复回来，再将 EIP 指针减一，然后恢复程序执行。程序恢复执行后，执行的是被 INT 3 替换的那条指令。

C.5 溢出异常（#OF）

向量号：4

异常类型：陷阱（Trap）

引入该异常的处理器：8088 最早引入该异常，其后的 8086、80286 以及所有 IA-32 处理器都实现了该异常。

描述：当 CPU 执行 INTO 指令时，如果标志寄存器的 OF 标志为 1（即 EFlags[OF]=1），那么便会产生此异常。

一些算术指令（如 ADD 和 SUB）使用 OF（Overflow Flag）标志表示在进行带符号整数计算时发生溢出（太大的整数或太小的负数，目标操作数无法容纳），使用 CF（Carry Flag）位表示在进行无符号整数计算时发生溢出（有进位或借位）。在进行带符号算术计算时，程序可以直接测试 OF 标志判断是否有溢出发生，也可以使用 INTO 指令。INTO 指令的优点是如果有溢出发生，CPU 会自动调用异常处理程序来进行处理。

错误代码：无

保存的程序指针：栈中保存的 CS 和 EIP 值指向的是 INTO 指令后面要执行的那条指令。

程序状态变化：尽管栈中的 CS 和 EIP 指向的是下一条指令，但是因为 INTO 指令不会影响任何寄存器或内存数据，程序状态并没有实质性变化，所以可以安全的恢复执行原来的程序。

C.6 数组越界异常（#BR）

向量号：5

异常类型：错误（Fault）

引入该异常的处理器：80286 最早引入该异常，其后的所有 IA-32 处理器都实现了该异常。

描述：当 CPU 执行 BOUND 指令时，如果数组的索引值不在指定的数组边界（bound）内，那么便会产生本异常（BOUND Range Exceeded Exception）。

BOUND 指令需要两个操作数，第一个操作数是装有数组索引值的寄存器，第二个操作数一个指向数组高低边界的内存位置（下边界在前）。如果将数组的高低边界都存贮在数组本身的前面，那么只要从数组开始处偏移一个常量就可以得到数组边界的位置，因为数组地址通常已经在寄存器中，这样便避免了分别取数组上下边界有效地址的总线操作。

但对于 Windows 这样的操作系统环境，因为所有异常都首先交给操作系统的异常处理程序来处理，然后如果异常来自用户态，那么再分发给用户态的应用程序。这个过程所花费的代价远远超过了 BOUND 指令可以节约的开销，所以 BOUND 指令在今天的软件产品中应用并不多。

错误代码：无

保存的程序指针：栈中保存的 CS 和 EIP 值指向的是导致异常的 BOUND 指令。

程序状态变化：该异常不会伴有程序状态变化，BOUND 指令的操作数也不会被修改。从异常处理例程返回后会重新执行 BOUND 指令。因此如果错误情况没有消除，那么会陷入“重新执行 BOUND 指令导致异常，异常返回再重新执行 BOUND 指令”的“死”循环。

C.7 非法操作码异常（#UD）

向量号：6

异常类型：错误（Fault）

引入该异常的处理器：80286 最早引入该异常，其后的所有 IA-32 处理器都实现了该异常，而且导致该异常的情况也在逐渐增加。

描述：以下情况会导致此异常：

- 试图执行无效或保留的操作码（OpCode）。一个例外是，尽管 D6 和 F1 是 IA-32 架构保留的未定义操作码，但它们不会导致异常。
- 指令的操作数类型与操作码不匹配，比如 LEA（Load Effective Address）指令的源操作数不是内存地址（偏移部分）。
- 在不支持 MMX 技术或 SSE/SSE2/SSE3 扩展的处理器上试图执行 MMX 或 SSE/SSE2/SSE3 指令。执行这些指令前应该先使用 CPUID 指令判断处理器是否支持 MMX（EDX 的位 23）、SSE（EDX 的位 25）/SSE2（EDX 的位 26）/SSE3（ECX 的位 0）。
- 当 CR4 的 OSFXSR 位（位 9）为 0 时，试图执行 SSE/SSE2/SSE3 指令。OSFXSR 位的含义是操作系统对 FXSAVE 和 FXRSTOR 指令的支持。但此规则不包括以下 SSE/SSE2/SSE3 指令：MASKMOVQ、MOVNTQ、MOVNTI、PREFETCHh、SFENCE、LFENCE、MFENCE 和 CLFLUSH；或 64 位版本的 PAVGB、PAVGW、PEXTRW、PINSRW、PMAXSW、PMAXUB、PMINSW、PMINUB、PMOVMSKB、PMULHUW、PSADBW、PSHUFW、PADDQ 和 PSUBQ。
- 当 CR4 的 OSXMMEXCPT 位（位 10）为 0 时，试图执行导致 SIMD 浮点异常的 SSE/SSE2/SSE3 指令。OSXMMEXCPT 位的含义是操作系统对非屏蔽 SIMD 浮点异常（Unmasked SIMD Floating-Point Exception，简称#XF）的支持。操作系统设置此位以表示已经准备好处理#XF 异常的例程。如果此位没被设置，则处理器会产生#UD 异常。
- 执行了 UD2 指令。值得注意的是即使是 UD2 指令导致了本异常，栈中保存的指令指针仍然是指向 UD2 指令。
- 在不可以锁定的指令前或可以锁定但目标操作数不是内存地址的指令前使用了 LOCK 前缀。
- 试图在实模式或虚拟 8086 模式执行 LLDT、SLDT、LTR、STR、LSL、LAR、VERR、VERW 或 ARPL 指令。
- 不在 SMM 模式（系统管理模式）下执行 RSM（从 SMM 模式返回）指令。

- 还要说明的一定是，对于奔腾 4 和 P6 系列这些支持乱序执行的处理器，只有在可能导致本异常的指令被回收（retire）时才会真正产生异常，因此投机执行了错误的分支并不会导致不该发生的异常。类似的，对于奔腾以及更早的 IA-32 处理器，预取（prefetching）和初步解码也不会产生异常。

错误代码：无

保存的程序指针：栈中保存的 CS 和 EIP 值指向的是导致本异常的那条指令。

程序状态变化：该异常不会伴有程序状态变化，因为无效的指令并没有被执行。

C.8 设备不可用异常（#NM）

向量号：7

异常类型：错误（Fault）

引入该异常的处理器：80286 最早引入该异常，当时的名字叫无可用数学单元（No Math Unit Available），其后的所有 IA-32 处理器都实现了该异常。

描述：以下情况会导致设备不可用异常（device-not-available）：

- 当 CR0 寄存器的 EM 标志为 1 时执行 x87 FPU 浮点指令。处理器使用 EM（Emulation）位表示其内部没有 x87 FPU 浮点单元。系统软件检测到此标志后，应当设置相应的处理例程，这样当有 x87 FPU 浮点指令执行时，CPU 便会产生本异常，并交由异常处理例程处理（可以进一步调用浮点指令仿真程序）。
- 当 CR0 寄存器的 MP（Monitor Coprocessor）和 TS（Task Switch）标志为 1 时（不管 EM 位是 0 或 1）执行 WAIT/FWAIT[1]指令。
- 当 CR0 寄存器的 TS 标志为 1 且 EM 标志为 0 时执行 x87 FPU、MMX 或 SSE/SSE2/SSE3 指令，但 MOVNTI、PAUSE、PREFETCHh、SFENCE、LFENCE、MFENCE 和 CLFLUSH 除外。
- CPU 在每次任务切换时都会设置 TS 标志，并在执行 x87 FPU、MMX 和 SSE/SSE2/SSE3 指令检查此标志，如果以上条件满足便在执行这些指令前产生此异常，目的是让异常处理例程保存 x87 FPU、MMX 和 SSE/SSE2/SSE3 的环境信息（context）。
- MP 标志和 TS 标志共同作用决定 WAIT/FWAIT 指令会不会导致#NM 异常。如果 MP 标志为 1，那么当 TS 为 1 时执行 WAIT/FWAIT 指令时便会导致本异常，让异常处理例程有机会在执行 WAIT 会 FWAIT 指令前保存 x87 FPU 的环境信息（context）。

错误代码：无

保存的程序指针：栈中保存的 CS 和 EIP 值指向的是导致本异常的那条指令。

程序状态变化：该异常不会伴有程序状态变化，因为导致异常的指令没有被执行。

C.9 双重错误异常（#DF）

向量号：8

异常类型：中止（Abort）

引入该异常的处理器：80286 最早引入该异常，其后所有 IA-32 处理器都实现了该异常。

[1] FWAIT 只是 WAIT 指令的另一个助记符，二者的机器码都是 9B，因此是完全等价的。

描述：表明当 CPU 在调用前一个异常处理例程时检测到第二个异常。正常情况下，当 CPU 在调用一个异常处理例程过程中又检测另一个异常时，它可以顺次的（serially）进行处理，但是如果处理器无法顺次处理它们，那么便会产生双重错误异常（Double Fault Exception）。为了判定对于那些错误组合应该产生双重错误，处理器将异常分为三个类型：良性（benign）异常、有益（contributory）异常和页错误（page fault）。

表 C-1　异常的另一种分类

类型	向量号码	描述
良性异常和中断	1	调试
	2	NMI 中断
	3	断点
	4	溢出
	5	数组越界
	6	无效操作码
	7	设备不可用
	9	协处理器段溢出（overrun）
	16	浮点错误
	17	对齐检查
	18	机器检查
	18	SIMD 浮点
	所有	INT n
	所有	INTR（来自 INTR 管脚的外部中断请求）
有益异常	0	除零
	10	无效 TSS
	11	段不存在
	12	栈错误
	13	一般保护（GP）异常
页错误	14	页错误异常

表 C-2 列出了哪些组合会导致双错异常。

表 C-2　导致双错异常的组合

第一次异常	第二次异常		
	良性类异常	有益类异常	页错误
良性类异常	顺次处理	顺次处理	顺次处理
有益类异常	顺次处理	产生双重异常	顺次处理
页错误	顺次处理	产生双重异常	产生双重异常

需要注意的是，表 2-12 列出的并非是所有会产生双重错误异常的情况，当 CPU 在试图将控制权移交给错误处理例程时，任何再发生的错误都会导致双重错误异常。

如果 CPU 在将控制权移交给双重错误异常的处理例程时又有异常发生，那么 CPU 机会进入关机模式（shutdown mode）。关机模式类似于执行 HLT（HALT，停止）指令后处理器所处于的状态。在此状态下，CPU 停止执行任何指令直到：

- 有不可屏蔽中断请求发生；
- 有 SMI（系统管理中断）请求发生；
- 硬复位（hardware reset）；
- 对 CPU 的 INIT#管脚置低电平（assertion）。

当进入关机模式时，CPU 会通过一个特别的总线周期/事务（bus cycle/transaction）输出关机消息（Shutdown message）以通知前端总线上的系统部件（芯片组）。作为响应，系

统部件可以采取如下措施：

- 打开指示灯通知给用户；
- 产生一个 NMI 中断，让 CPU 苏醒并执行 NMI 处理例程，NMI 处理例程可以从芯片组那里获得状态信息，判断出 CPU 处在关机模式，记录下诊断信息，然后命令芯片组向设置以下信号之一：
  - RESET#，让 CPU 硬复位；
  - INIT#，让 CPU 软复位；
  - SMI#，让 CPU 转入系统管理模式。

当 CPU 在执行 NMI 中断处理例程或在系统管理模式下发生关机，那么就只能通过硬复位使 CPU 重启。

错误代码：0。对于双重错误异常，CPU 总是向栈中压入错误码 0。

保存的程序指针：栈中保存的 CS 和 EIP 值的指向未定义。

程序状态变化：双重错误异常属于中止类异常，该异常发生后的处理器状态没有明确定义，所以被中断的程序不可以恢复执行。

双重错误异常处理例程的可以做的事情就是收集可能的环境信息供将来诊断使用。如果当异常处理机制的机器状态被破坏时有双重错误异常发生，那么处理器将无法调用相应的处理例程，这是 CPU 只能重启。

C.10 协处理器段溢出异常

向量号：9

异常类型：中止（Abort）

引入该异常的处理器：286 处理器最早引入该异常， 386 以及 486 的早期版本（在将数学协处理器集成进 CPU 以前，如 486SX）实现了该异常。从 486DX 开始后的 IA-32 处理器不再使用此异常。

描述：该异常表示 CPU（如 386）在向 FPU（数学协处理器，如 387）操作数的中间部分时检测到了页错误或段不存在异常。从 486DX 开始，这种情况会被当作一般保护异常（#GP）报告。

错误代码：无

保存的程序指针：栈中保存的 CS 和 EIP 值指向的是导致该异常的那条指令。

程序状态变化：该异常发生后的处理器状态没有明确定义，所以被中断的程序不可以恢复执行。

C.11 无效 TSS 异常（#TS）

向量号：10

异常类型：错误（Fault）

引入该异常的处理器：286 最早引入该异常，其后的所有 IA-32 处理器都实现了该异常。

描述：该异常表示 CPU 在进行任务切换或者执行使用 TSS（任务状态段）信息的指令

时检测到了一个与 TSS 有关的错误。表 C-3 列出了各种错误情况和该情况导致异常时错误代码中的索引域内容。总体说来，错误情况是由于以下内容违反了保护模式规则：

- TSS 描述符；
- TSS 指向的 LDT（局部描述符表）；
- TSS 中引用的任何其它段。

表 C-3　导致无效 TSS 的错误情况

无效情况	错误码索引域
TSS 段的限制小于 67H（32 位的 TSS）或 2CH（16 位 TSS）	TSS 段选择子索引
通过 IRET 切换任务时，TSS 段选择子的 TI（table index）标志值代表的是 LDT（应该为 GDT）。	TSS 段选择子索引
通过 IRET 切换任务时，TSS 段选择子的索引值超出描述符表的限制。	TSS 段选择子索引
通过 IRET 切换任务时，TSS 描述符的 busy 标志值代表的是非活动任务。	TSS 段选择子索引
通过 IRET 切换任务时，试图加载 backlink[2] limit 时出错。	TSS 段选择子索引
通过 IRET 切换任务时，backlink 指向的是空选择子。	TSS 段选择子索引
通过 IRET 切换任务时，backlink 选择的 TSS 描述符的 busy 标志不“繁忙”（not busy）。[3]	TSS 段选择子索引
新的 TSS 描述符超出了 GDT 的限制。	TSS 段选择子索引
新的 TSS 描述符是不可写的。	TSS 段选择子索引
向旧的 TSS 存储时遇到错误	TSS 段选择子索引
通过跳转或 IRET 进行任务切换时旧的 TSS 描述符不可以写	TSS 段选择子索引
通过调用（call）或异常进行任务切换时新 TSS 的 backlink 不可以写	TSS 段选择子索引
试图锁定新的 TSS 时，新 TSS 的选择子为空	TSS 段选择子索引
试图锁定新的 TSS 时发现新 TSS 选择子的 TI 位为 1（1 代表 LDT，应该为 0 代表 GDT）	TSS 段选择子索引
试图锁定新的 TSS 时发现新 TSS 描述符不是可用的 TSS 描述符	TSS 段选择子索引
LDT 或 GDT 不存在	LDT 段选择子索引
栈段选择子（stack segment selector）超出描述符表限制	栈段选择子索引
栈段选择子为空	栈段选择子索引
栈段描述符的类型不是数据段	栈段选择子索引
栈段不可以写	栈段选择子索引
栈段 DPL!=CPL	栈段选择子索引
栈段选择子 RPL!=CPL	栈段选择子索引
代码段选择子（code segment selector）超出描述符表限制	代码段选择子索引
代码段选择子为空	代码段选择子索引
代码段描述符的类型不是代码段	代码段选择子索引
对于非相容（nonconforming）代码段[4]，DPL!=CPL	代码段选择子索引
对于相容（conforming）代码段[5]，DPL 大于 CPL	代码段选择子索引
数据段选择子（data segment selector）超出描述符表限制	数据段选择子索引
数据段描述符的类型不是可读的代码或数据类型	数据段选择子索引

[2] TSS 中指向前一个任务的 TSS 的段选择子，简称 backlink。

[3] 繁忙的任务是指正在运行的任务或者被挂起（suspended）的任务。从 IRET 切换回去的任务是被打断的任务，其 Busy 标志应该为繁忙。Busy 标志是为了防止任务切换陷入死循环。通常，当切换到的目标任务（新任务）的繁忙标志已经设置时，会产生一般保护异常。但是如果该次任务切换是由 IRET 指令发起的那么便不会产生异常。

[4] 非相容代码段的代码只能被 CPL（Current Previlege Level）与其 DPL（Descriptor Previlege Level）相等的程序所调用，也就是只有同等优先级的代码才可以调用或跳转到非相容代码段的代码。

[5] 相容代码段的代码可以被 CPL（Current Previlege Level）与其 DPL（Descriptor Previlege Level）相等或更小的程序所调用，也就是同等优先级或更低优先级的代码可以调用或跳转到相容代码段的代码。

无效情况	错误码索引域
数据段描述符的类型是非相容代码类型而且 RPL>DPL	数据段选择子索引
数据段描述符的类型是非相容代码类型而且 CPL>DPL	数据段选择子索引
对于 LTR 指令，TSS 段选子为空	TSS 段选择子索引
对于 LTR 指令，TSS 段选子的 TI 标志为 1	TSS 段选择子索引
TSS 段描述符超出 GDT 段限制	TSS 段选择子索引
TSS 段或 upper 描述符不是可用的 TSS 类型	TSS 段选择子索引
IA-32e 模式下，TSS 段描述符是 286 TSS 类型	TSS 段选择子索引
TSS 段 upper 描述符不是正确的类型	TSS 段选择子索引
TSS 段描述符包含不规范的（non-canonical）基址（base）	TSS 段选择子索引
试图从 TSS 加载 SS 选择子或 ESP 时超出限制	TSS 段选择子索引

CPU 可能按照原任务的上下文产生无效 TSS 异常，也可按照新任务的上下文产生该异常。如果处理器已经完成对新 TSS 的检验，那么就在新任务的上下中产生异常，否则处理器会在原任务的上下文中产生异常。

无效 TSS 异常的处理例程必须是通过任务门调用的任务。在 TSS 有问题的任务中处理该异常可能因处理器的状态不一致而出问题。

错误代码：错误码包含了导致异常的段描述符的段选择子索引和段描述符表的种类（IDT/GDT）。如果 EXT 标志被置位，那么表示该异常是由外部事件（相对于正在运行的程序）导致的，例如当有外部硬件中断发生时，使用任务门调用外部中断服务例程过程中目标 TSS 无效。

保存的程序指针：如果是在任务切换完成前检测到异常，那么栈中保存的 CS 和 EIP 值指向的是请求（invoke）任务切换的那条指令。如果是在任务切换完成后检测到异常，那么栈中保存的 CS 和 EIP 值指向的是新任务的第一条指令。

程序状态变化：如果无效 TSS 异常发生在任务移交点（即执行权移交给新任务那一点）之前，那么程序状态没有变化。如果发生在移交点（新的段选择子的段描述符信息已经被加载到段寄存器中）之后，那么处理器会在产生异常前从新 TSS 加载所有状态信息。在任务切换过程中，处理器首先从 TSS 中装载所有段寄存器，然后再检查其内容的有效性。如果在检查中发现无效 TSS 异常，那么处理器不再继续检查还没有检查的段寄存器。因此无效 TSS 异常处理例程如果使用各个段寄存器（CS、DS、ES、FS、GS 和 SS）中的段选择子，那么有可能再次导致无效 TSS 异常。英特尔推荐使用一个单独的任务来处理无效 TSS 异常，那么当从异常处理例程切换回被打断的任务时，CPU 在从 TSS 加载该任务时会检查各个段寄存器。

C.12 段不存在异常（#NP）

向量号：11

异常类型：错误（Fault）

引入该异常的处理器：286 处理器最早引入该异常，其后的所有 IA-32 处理器都实现了该异常。

描述：表示段或门描述符的存在（present）标志为 0。以下操作可能导致处理器产生该异常：

- 试图加载 CS、DS、ES、FS、GS 寄存器。如果在加载 SS 寄存器时检测到存在位为 0，那么会导致栈错误异常（#SS）。

- 使用 LLDT 指令加载 LDTR 寄存器。如果在任务切换过程中加载 LDTR 检测到 LDT 不存在那么会导致无效 TSS 异常（#TS）。
- 当执行 LTR 指令时 TSS 被标记为不存在。
- 试图使用一个被标记为段不存在（其它属性都有效）的门描述符或 TSS。

操作系统可以使用段不存在异常实现段一级的虚拟内存。当要访问一个不在物理内存中的段时，CPU 产生段不存在异常并调用异常处理程序，异常处理程序可以把该段从虚拟内存中加载到物理内存中，然后返回继续执行。

在门描述符中，因为门并不对应段，所以“不存在”并不表示段不存在。操作系统可以使用该标志表示其它含义。

在报告或处理有益类（contributory）异常和页错误异常时如果引用了不存在的段，那么处理器会报告双重错误异常（#DF）而不是段不存在异常（#NP）。

错误代码：错误码中包含了导致错误的段描述符的段选择子索引。如果 EXT 标志为 1，那么该异常是由于以下情况之一导致的：

- 外部事件（不可屏蔽中断或 INTR）导致了异常，随后引用了不存在的段。
- 良性（benign）类异常随后引用了不存在的段。

如果错误码描述的是 IDT 表项，那么 IDT 标志为 1。这类错误是由于被响应中断的向量值指向的 IDT 表项是不存在的中断门。

保存的程序指针：栈中保存的 CS 和 EIP 值指向的通常是导致该异常的那条指令。如果该异常是发生在任务切换过程中加载新任务的段描述符，那么 CS 和 EIP 寄存器指向的新任务的第一条指令。如果是在访问门描述符时发生该异常，那么 CS 和 EIP 寄存器指向的请求访问的那条指令（比如引用调用门的 CALL 指令）。

程序状态变化：如果段不存在异常是因为加载 CS、DS、ES、FS 或 GS 寄存器而发生的，那么程序状态不会改变[6]，因为寄存器内容并没有改变。对于此种异常，异常处理程序可以在将缺少的段加载到内存并设置段描述的存在标志后恢复运行原来的程序。

如果在访问门描述符时发生了段不存在异常，不会伴有程序状态变化，异常处理程序只要设置好存在标志就可以恢复程序运行。

当段不存在异常发生在任务切换过程中，如果发生时刻是在把控制权移交给新任务那一点之前，那么程序状态没有变化。如果发生在移交点之后，那么处理器会在产生异常前从新的 TSS 加载所有状态信息。

C.13 栈错误异常（#SS）

向量号：12

异常类型：错误（Fault）

引入该异常的处理器：80286 最早引入该异常，其后的所有 IA-32 处理器都实现了该异常。

描述：该异常表示 CPU 检测到与栈有关的以下情况之一：

[6] IA-32 手册此处有误，原句为”a program state does accompany the exception because the register is not loaded”，应该是遗漏了“not”。

在执行某个引用 SS 寄存器的操作时 CPU 检测到段界违例（limit violation）。导致段界违例的操作有以下两类：明确或隐含使用 SS 寄存器的指令，如 MOV AX, [BP+6]或 MOV AX, SS:[EAX+6]；栈操作指令，如 POP、PUSH、CALL、RET、IRET、ENTER 和 LEAVE，当使用 ENTER 指令来为一个过程建立栈桢（stack frame）时，如果没有足够的栈空间来分配局部变量，那么便会产生此异常。

当加载 SS 寄存器时，CPU 检测到栈段不存在（not present stack segment）。这种情况可能发生在执行任务切换的过程中，目标优先级不同的 CALL 指令，返回到不同的优先级，LSL 指令（Load Segment Limit，即加载段界），目标为 SS 寄存器的 MOV 或 POP 指令。

栈错误异常是可能被恢复的，对于第一种情况可以通过扩展段界的方式来修正这个错误；对于第二种情况只要把缺少的栈段加载到内存。

错误代码：对于栈段不存在（第二类情况）和不同优先级间的调用导致的栈溢出（第一类情况中的跨优先级情况），错误码中包含了导致异常的那个段的段选择子。异常处理例程可以通过该选择子找到该段的段描述符并判断其存在标志。

对于通常情况（发生在一个已经使用了的栈）的栈界违例，错误码为 0。

保存的程序指针：栈中保存的 CS 和 EIP 值通常指向的是导致该异常的那条指令。但如果是在任务切换时由于试图加载一个不存在的栈段而导致异常，那么 CS 和 EIP 指向的是新任务的第一条指令。

程序状态变化：如果栈错误异常是在任务切换过程中发生的，那么它可能发生在把控制权移交给新任务那一点之前，也可能发生在巴控制权移交给新任务之后。如果是发生在之前，那么程序状态没有改变。如果是发生在之后，那么处理器会在产生异常前继续从新的 TSS 加载所有状态信息（而且不再做边界、类型和存在性检查）。所以异常处理例程不能假定使用 CS、SS、DS、ES、FS 和 GS 段寄存器中的段选择子不会再导致异常。异常处理例程应该认真检查每个段寄存器后再恢复运行原来的程序，否则可能再次发生错误使问题更难定位和解决。

对于其它情况的栈错误异常，不会伴有程序状态变化，因为 CPU 在报告异常前会恢复到执行触发异常的指令前的状态。

C.14 一般性保护异常（#GP）

向量号：13

异常类型：错误（Fault）

引入该异常的处理器：

80286 最早引入该异常，其后的所有 IA-32 处理器都实现了该异常。

描述：CPU 的保护模式旨在提供一种多个应用程序在系统软件的监管下共享资源同时运行的多任务环境，首先要保护系统软件的安全，使其不会受到其它软件的有意或无意破坏，其次就是要保护系统中运行着的每个任务不受其它任务的破坏。为了实现这两种保护，CPU 制定了一系列保护性规则（protection rule），并要求系统中运行的所有程序都要遵守这些规则。如果有程序违反了保护性规则（称为 protection violation），那么 CPU 便会通过异常的方式汇报给系统软件，系统软件根据情况采取措施，必要时会强行关闭“违例”的程序（参见 3.6 节）。为了报告监测到的“违例”情况，CPU 定义了几种异常。比如当程序中的指针指向了无效的内存地址时（比如空指针），那么 CPU 便会通过页错误异常（#PF）

将这一情况报告给系统软件。再比如，当一个程序试图向栈中压入过多数据，将导致超出栈边界（limit）时，那么 CPU 便会通过栈错误异常（#SS）将该情况报告给操作系统。对于 Windows 操作系统来说，如果该情况发生在用户模式，那么操作系统会尝试分配更多的栈空间然后恢复执行被中断的指令，如果该情况发生在内核模式，那么操作系统便会。但更多的情况下，CPU 会通过一般性保护异常（#GP——General Protection Exception）来报告违反保护模式规则的情况。按照 IA-32 手册的说法，凡是不属于（已定义的）其它异常的保护性违例都会作为一般性保护异常来报告。目前会导致一般性保护异常的情况有：

- 当访问 CS、DS、ES、FS 或 GS 段时超出段边界。
- 当引用一个描述符表时超出段边界（不包括发生在任务切换或栈切换过程中的情况，因为这两种情况会分别报告无效 TSS 异常和栈错误异常）。
- 向不可执行的段移交执行权。
- 向代码段或只读数据段写。
- 读仅可以执行（execute-only）的代码段。
- 向 SS 寄存器中加载指向只读段的段选择子（不包括在切换任务时从 TSS 中读出的段选择子，该情况会以无效 TSS 异常的形式汇报）。
- 向 SS、DS、ES、FS 或 GS 寄存器中加载指向系统段的段选择子。
- 向 DS、ES、FS 或 GS 寄存器中加载指向只可执行（execute-only）代码段的段选择子。
- 向 SS 寄存器中加载指向可执行段的段选择子或空段选择子[7]（null segement selector）。
- 向 CS 寄存器中加载指向数据段的段选择子或空段选择子。
- 当 DS、ES、FS 或 GS 寄存器中包含空段选择子时使用它们访问内存。
- 当使用指向 TSS 的 CALL 或 JMP 指令进行任务切换时，目标任务忙（busy）。
- 在使用非 IRET（non-IRET）指令切换任务时使用的段选择子指向了当前 LDT（局部描述符表）的 TSS。TSS 段只能位于 GDT（全局描述符表）中。如果在使用 IRET 指令切换任务时检测到这种情况，那么 CPU 会报告无效 TSS 异常。
- 违反任何权限规则（privilege rules），参见 2.3 节。
- 指令超出长度限制（最长 15 字节），这种情况只可能发生在指令前放置了过多的前缀（否则便会导致无效指令异常）。
- 试图将 CR0 寄存器加载为 PG 标志为 1（paging enabled）PE 标志为 0（protection disabled），也就是没有启用保护模式时启用内存页支持。因该先启用保护模式（PE 位置 1）或同时将 PE 和 PG 位置 1。
- 试图将 CR0 寄存器加载为 NW（Not Write-through）标志为 1 但 CD（Cache Disable）标志为 0。
- 中断或异常所引用的中断描述符表（IDT）表项不是中断门、陷阱门或任务门。
- 企图从虚拟 8086 模式通过中断或陷阱门访问所在代码段的 DPL 大于 0 的中断或陷阱处理例程。
- 企图向 CR4 寄存器的保留位写 1。
- 当 CPL（当前权限级别）不等于 0 时执行特权指令。
- 写 MSR 寄存器的保留位。

[7] GDT 表的第一个表项是保留不用的，指向这个表项的段选择子（也就是 TI 标志为 0，索引也为 0）被称为空段选择子。当向 CS 或 SS 寄存器加载空段选择子，CPU 会立刻报告一般性保护异常。向其它段寄存器加载空段选择子时，CPU 不会报告异常，但如果使用这些寄存器访问内存时，CPU 便会报告异常。

- 访问包含空段选择子的门。
- 执行 INT n 指令时 CPL 大于（数值上）引用的中断门、陷阱门或任务门的 DPL。
- 调用门、中断门或陷阱门中的段选择子指向的不是代码段。
- LLDT 指令中的段选择子操作符是局部类型（TI 位为 1）或者没有指向 LDT 类型的段描述符。
- LTR 指令中的段选择子操作符是局部类型（TI 位为 1）或者指向的 TSS 不可用。
- 调用、跳转或返回的目标代码段选择子为空。
- 当 CR4 寄存器的 PAE 和/或 PSE 位为 1 时 CPU 检测到页目录指针表（page-directory-pointer-table）表项的保留位被置位 1。当写 CR0、CR3 或 CR4 控制寄存器时导致重新加载页目录指针表表项时，CPU 会检查这些位。
- 试图向 MXCSR 寄存器的保留位写非零的值。
- 执行要求按 16 字节对齐的 SSE/SSE2/SSE3 指令访问 128 位的内存区域时边界地址没有按 16 字节对齐。这一规则也适用于堆栈段。

错误代码：处理器会向异常处理例程的堆栈压入一个错误码。如果错误是在加载段描述符时检测到的，那么错误码会包含指向这个代码段描述符的段选择子或者这个描述符对应的 IDT 向量号。CPU 可能从以下来源提取错误码中的段选择子：指令操作数；参数所指定的门（gate）的选择子；参与任务切换的 TSS 的选择子；IDT 向量号。

保存的程序指针：栈中保存的 CS 和 EIP 值指向的是产生异常的那条指令。

程序状态变化：通常一般性保护异常不会伴有程序状态变化，因为无效的指令或操作没有被执行（严格说是 CPU 会恢复到导致异常的指令被执行前的状态）。因此，异常处理程序可以纠正导致一般性保护错误的情况，然后恢复被中断的任务或程序。

当一般性保护异常是在任务切换过程中发生的，那么它可能发生在把控制权移交给新任务那一点之前，也可能发生在巴控制权移交给新任务之后。如果是发生在之前，那么程序状态没有改变。如果是发生在之后，那么处理器会在产生异常前继续从新的 TSS 加载所有状态信息（而且不再做边界、类型和存在性检查）。所以异常处理例程不能假定使用 CS、SS、DS、ES、FS 和 GS 段寄存器中的段选择子不会再导致异常。异常处理例程应该认真检查每个段寄存器后再恢复运行原来的程序，否则可能再次发生错误使问题更难定位和解决。

如果异常发生在试图调用中断处理例程的过程中，那么被中断的程序可以重新运行，但是中断可能丢失了。

C.15 页错误异常（#PF）

向量号：14

异常类型：错误（Fault）

引入该异常的处理器：80386 最早引入该异常，其后所有 IA-32 处理器都实现了该异常。

描述：表示在启用了页机制（paging）（也就是 CR0 寄存器的 PG 标志为 1）的情况下，CPU 在将线性地址翻译为物理地址时检测到了如下情况：

- 进行地址翻译所需的页表（page-table）或页目录（page-directory）表项的 P（Present）标志为 0。P 标志为 0 表示包含操作数的页表或页不在物理内存中。

- 当前程序没有足够的权限访问指定的页（比如，用户模式下的函数访问管理模式使用的页）。
- 运行在用户模式下的代码试图写只读属性的内存页。从 486 开始，如果 CR0 寄存器的 WP（Write Protect）标志为 1，那么运行在管理模式的代码如果写只读属性的用户态内存页也会触发页错误异常。
- 取指到（fetch instruction to）一个线性内存地址时，该地址被翻译为物理地址后所在的内存页设置了禁止执行（execute-disable）标志。
- 页目录表项的一或多个保留位被设为 1。参见下面对 RSVD 错误码标志的描述。

错误代码：处理器会向异常处理例程的堆栈压入一个特殊格式的错误码。该错误码的格式（参见图 C-1）不同于其它异常所使用的格式（图 3-2）。

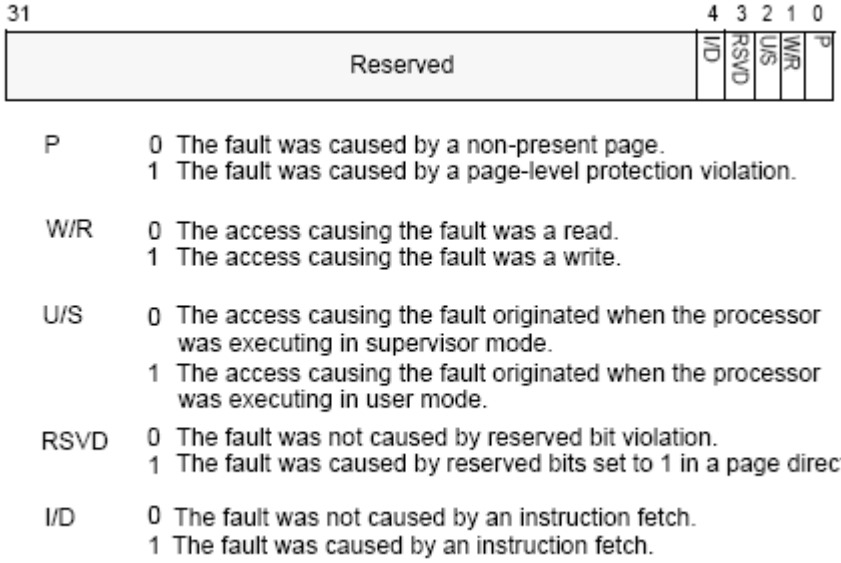


图 C-1　页错误异常的错误码

- P 位如果为 0 则表示该异常是由于页不存在所导致的，如果为 1 则表示该异常是因为违反访问权限或使用保留位所导致的。
- W/R 位表示导致该异常的内存访问是读（0）还是写（1）操作。
- U/S 位表示异常发生时处理器是在用户模式（1）还是内核模式（0）下执行。
- RSVD 位为 1 表示当 CR4 寄存器的 PSE 或 PAE 为 1 时，CPU 检测到页目录的 1 或多个保留位中包含 1。如果 RSVD 位为 0，那么该异常不是由于设置保留位而导致的。
- I/D 位表示该异产是否是由于取指操作导致的（1 是，0 否）。如果 CPU 不支持禁止执行位或者没有启用禁止执行位，那么 I/D 位保留。

除了错误码，CPU 还提供了另一个信息供页错误异常处理例程分析异常原因。在产生异常前，CPU 会把导致异常的 32 位线性地址加载到 CR2 寄存器中。异常处理例程可以使用该地址来定位对应的页目录和页表表项。在执行页错误异常例程时还有可能再发生页错误，而且每次发生页错误异常时，CPU 都会更新 CR2 寄存器，为了防止前一次异常的线性地址被第二次异常的线性地址所覆盖，异常处理程序应该在发生第二次异常前将 CR2 寄存器的内容保存起来。

如果页错误异常是由于违反页面级保护规则[8]而导致的，那么当错误发生时，CPU 会

[8] 从内存管理角度来看，可以把保护机制分为段级（主要通过段描述符定义的）和页级（主要通过页面属

设置页目录表项的访问标志。但是页表表项访问标志的规则是与处理器型号有关的。

保存的程序指针：通常栈中保存的 CS 和 EIP 值指向的是产生异常的那条指令。如果实在任务切换过程中发生的页错误异常，那么 CS 和 EIP 值指向的可能是新任务的第一条指令。

程序状态变化：通常不会导致程序状态变化因为导致错误的指令没有被执行。异常处理例程可以纠正错误（比如将不在物理内存中的页面加载到内存中，这便是虚拟内存的基本工作原理），然后恢复被中断的程序。

但是如果实在任务切换的过程中发生了页错误异常，那么程序状态可能变化。在切换任务时，以下任何操作都可能导致页错误异常：

- 将原来任务的状态（寄存器等）写到该任务的 TSS（任务状态段）中。
- 从 GDT 表中读取信任务的 TSS 的段描述符。
- 读新任务的 TSS。
- 读新任务中的段选择子指向的段描述符。
- 读新任务的 LDT（局部描述符表）以验证保存在新任务的 TSS 中的段寄存器。

对于最后两种情况，异常会发生在新任务的上下文中，栈中保存的 CS 和 EIP 值指向的是新任务的第一条指令，不是导致任务切换的那条指令。

如果页错误异常是在任务切换过程中发生的，那么它可能发生在把控制权移交给新任务那一点之前，也可能发生在巴控制权移交给新任务之后。如果是发生在之前，那么程序状态没有改变。如果是发生在之后，那么处理器会在产生异常前继续从新的 TSS 加载所有状态信息（而且不再做边界、类型和存在性检查）。所以异常处理例程不能假定使用 CS、SS、DS、ES、FS 和 GS 段寄存器中的段选择子不会再导致异常。异常处理例程应该认真检查每个段寄存器后再恢复运行原来的程序，否则可能再次发生错误使问题更难定位和解决。

C.16 x87 FPU 浮点错误异常（#MF）

向量号：16

异常类型：错误（Fault）

引入该异常的处理器：80286 最早引入该异常，其后的所有 IA-32 处理器都实现了该异常。

描述：该异常表示 x87 FPU（Floating-Point Unit）检测到当一个浮点错误。CR0 寄存器的 NE 标志位为 1 时允许产生该异常（NE 标志为 0 的情况见下文）。

x87 FPU 可以检测并报告 6 种浮点错误情况：

- 无效运算（#I）：又包括栈溢出和下溢（underflow）（#IS）；以及无效的算术运算（#IA）。
- 除零（#Z）。
- 非规格化（denormalized）的操作数（#D）。
- 数值溢出（#O）。
- 数值下溢（#U）。

性定义的）保护。

- 结果不精确（#P），P 代表精度（Precision）。

以上的每种错误情况对应于 x87 FPU 的一种异常类型。对每一种异常类型，x87 FPU 的状态寄存器和控制寄存器分别提供了一个标志和一个屏蔽位。如果 x87 FPU 检测到一个浮点错误，而且控制寄存器中该异常类型的屏蔽位为 1，那么 x87 FPU 会自动处理该异常：产生一个预先定义（缺省的）的回应然后继续执行。对于大多数浮点应用，缺省响应可以提供合理的结果。

如果异常类型对应的屏蔽位为 0，而且 CR0 寄存器的 NE（Numeric Error）标志位为 1，那么 x87 FPU 会采取如下动作：

1. 在 FPU 状态寄存器中设置必要的标志。

2. 如果遇到等待型 x87 FPU 指令或 WAIT/FWAIT 指令，便在执行该指令前产生一个内部信号使 CPU 产生一个浮点错误异常。这里要说明的是如果程序中一直没有这样的指令，那么该异常便不会被报告。

CR0 的 NE 标志为 0 可以使 x87 FPU 使用 PC 兼容方式（PC-Style，又叫 MS-DOS 兼容方式）来报告浮点错误：当 IGNNE#管脚的信号有效（asserted）时，忽略所有 x87 浮点错误；当 IGNNE#管脚信号无效（deasserted）时，如果有未屏蔽的（unmasked）x87 FPU 错误发生，那么处理器会在遇到下一个等待浮点指令（WAIT 或 FWAIT）时置起（assert）FERR#管脚信号并进入冻结状态(停止指令执行)等待中断发生。在 PC 兼容系统中，FERR#管脚信号会被送到级联的可编程中断控制器（PIC）芯片的 IRQ13 输入端。在 FERR#信号的作用下，PIC 会产生中断使 CPU 进入处理浮点错误的中断处理例程。

在执行等待型 x87 FPU 指令或 WAIT/FWAIT 指令之前，x87 FPU 会检查是否有等待报告（pending）的浮点错误异常（上面的第二步）。对于非等待型（non-waiting）x87 FPU 指令(包括 FNINIT、FNCLEX、FNSTSW、FNSTSW AX、FNSTCW、FNSTENV 和 FNSAVE)，x87 FPU 不会检查是否有等待报告的异常。在执行浮点状态管理指令（FXSAVE 和 FXRSTOR）时，x87 FPU 也会忽略等待报告的异常。

错误代码：无，x87 FPU 的状态寄存器提供了错误信息

保存的程序指针：栈中保存的 CS 和 EIP 值指向的是产生 x87 FPU 浮点错误异常时将要执行的 WAIT/FWAIT 或浮点指令。不是 x87 FPU 检测到导致浮点错误的那条指令。导致浮点错误的指令的地址被存储在 x87 FPU 的指令指针寄存器中。

程序状态变化：程序状态通常会变化，因为浮点错误异常发生后会被延迟到遇到下一个等待型浮点指令或 WAIT/FWAIT 指令时汇报和处理。不过，因为 x87 FPU 保存了足够多的信息，所以大多时候还是可以从错误中恢复，如果需要可以重新执行导致错误的指令。

如果程序中有非 x87 FPU 指令依赖 x87 FPU 指令的结果，那么可以在该指令前插入一条 WAIT/FWAIT 指令以强迫检查是否有等待报告的浮点异常。

C.17 对齐检查异常（#AC）

向量号：17

异常类型：错误（Fault）

引入该异常的处理器：486 处理器最早引入该异常，其后所有 IA-32 处理器都实现了该异常。

描述：32 位处理器通常具有 32 根数据线（比如 386 和 486），因此一次可以传递 4 个字节（32 位）。从奔腾处理器开始，IA-32 处理器的数据线从 32 根增加到 64 根，一次可以传递 8 个字节（64 位）。为了提高访问效率和简化电路设计，32 位数据总线的 CPU 总是从能够被 4 整除的地址访问内存（每次 4 个字节）。64 位数据总线的 CPU 总是从可以被 8 整除的地址访问内存（每次 4 个字节）。也就是说 CPU 的地址线总是选通（呈现）是 4 或 8 整数倍的地址。例如，奔腾 4 CPU 的地址线是 A[35:3]，共 33 根，代表高 33 位地址（低 3 位省略），可以最多寻址 64G 物理内存。也就是说，这样的地址线已经不能表示出不是 8 的整数倍的地址。当要访问非 8 整数倍的地址，那么就选通与其最接近的 8 整数倍地址。举例来说，如果要读取地址 4 开始的 4 个字节，那么 CPU 便通过地址线输出地址 0，并通过字节选中信号 BE#（Byte Enable）指定需要的是高 4 字节。如果要读取地址 7 开始的 4 个字节，那么就要向输出地址 0，取最高字节，然后再输出地址 8，取低 3 字节，最后再通过移位操作将两次得到的数据合在一起。从这两个例子看到，同样是读取 4 个字节，前一种情况只要读取一次，后一种情况则要读取两次。分析原因，前一种情况要读取的 4 字节数据的起始地址是可以被 4 整除的，这样的地址被称为是按 4 字节边界对齐的。而后一种情况要读取的数据的起始地址是不可以被要读取的字节数整除，这样的地址被称为是没有对齐的。

从上面的分析我们看到 CPU 访问满足内存对齐要求的数据可以大大提高性能。另外，对于某些数据区，CPU 要求其起始地址一定要是内存对齐的。内存对齐异常正是为了强制这些要求而设计的。下表列出了各种数据类型或数据的内存对齐要求。

表 C-4　内存对齐要求

数据/数据类型	地址必须可以被整除
Word（字）	2
Doubleword（双字）	4
单精度浮点数（32 位）	4
双精度浮点数（64 位）	8
扩展双精度浮点数（double extended-precision floating-point）（80 位）	8
Quadword（4 字）	8
Double quadword	16
段选择子	2
32 位长指针	2
48 位长指针	4
32 位指针	4
GDTR、IDTR、LDTR 或任务寄存器（TR）的内容	4
FSTENV/FLDENV 保存区域	4 或 2*
FSAVE/FRSTOR 保存区域	4 或 2*
Bit String	4 或 2*

*依赖于操作数长度（size）。

值得注意的是仅当满足以下条件时，CPU 才会启用内存对齐检查：

- CR0 寄存器的 AM（Alignment Mask）标志为 1。
- EFLAGS 寄存器的 AC（Alignment Check）标志为 1。
- CPU 处于保护模式或虚拟 8086 模式，并且 CPL（当前权限级别）为 3。

也就是说只有当 CPU 在用户模式下操作时才会进行内存对齐检查。缺省指向 0 特权级的内存引用（比如加载段描述符）不会导致对齐检查，即使该操作是由于用户模式下的内存引用所导致的。

如果满足了内存对齐检查的条件，而且 CPU 检测到违反表 2-8 规定的情况，那么 CPU 便会产生内存对齐检查异常（#AC），但有一个例外，对于 128 位的数据类型没有按 16 字节边界对齐的情况，CPU 会以一般性保护异常（#GP）的形式报告。

错误代码：无

保存的程序指针：栈中保存的 CS 和 EIP 值指向的是产生异常的那条指令。

程序状态变化：程序状态并没有变化，CPU 报告异常前会恢复到导致异常的指令被执行前的状态，所以当异常处理程序纠正了错误情况后可以安全的恢复执行原来的程序。

C.18 机器检查异常（#MC）

向量号：18

异常类型：中止（Abort）

引入该异常的处理器：奔腾处理器最早引入该异常，其后所有 IA-32 处理器都实现了该异常。

描述：该异常表示 CPU 检测到一个内部错误或总线错误，或者系统的外部主体（agent，比如内存控制器 MCH）检测到总线错误。外部主体检测到的错误是通过专门的 CPU 管脚通知 CPU 的。奔腾 CPU 使用的是 BUSCHK#管脚，奔腾 4、至强、和 P6 系列处理器使用的是 BINIT#和 MCERR#管脚。

机器检查异常的具体工作方式是与处理器型号有关的。尤其是奔腾处理器和其后的 P6 及奔腾 4 处理器在这方面有较大的差异，详见 2.9 节。

CR4 寄存器的 MCE 标志用来启用机器检查机制。

错误代码：无，但是专门为机器检查机制设计的 MSR 寄存器提供了错误信息。

保存的程序指针：对于奔腾 4 和至强处理器，扩展的机器检查状态寄存器中保存了当 CPU 检测到机器检查异常时的状态信息，包括通用寄存器、标志寄存器 EFLAGS、EIP 等，而且这些信息是与发生的机器检查异常直接相关的。

对于 P6 系列处理器，如果 MCG_STATUS_MSR 寄存器的 EIPV 标志为 1，那么保存的 CS 和 EIP 值是与导致机器检查异常的错误直接有关的，如果该标志为 0，那么保存的指令指针可能与发生的错误不相关。

对于奔腾处理器，CS 和 EIP 寄存器的值可能与发生的错误相关，也可能不相关。

程序状态变化：对于奔腾 4、至强、P6 系列和奔腾处理器，机器检查异常总会伴有程序状态变化，而且机器检查异常是以中止类异常报告的。当中止类异常发生后，异常处理程序可以收集各种信息供调试使用，但是通常不能恢复程序继续运行。

如果没有启用机器检查机制，那么机器检查错误会导致 CPU 进入关机状态。

C.19 SIMD 浮点异常（#XF）

向量号：19

异常类型：错误（Fault）

引入该异常的处理器：奔腾 III 处理器最早引入该异常，其后所有 IA-32 处理器都实现了该异常。

描述：该异常表示 CPU 在执行 SSE/SSE2/SSE3 SIMD 运算时检测到错误。SIMD 是 Single Instruction Multiple Data，即单指令多数据的缩写，SIMD 是 MMX（MultiMedia Extensions，即多媒体扩展）技术的主要内容，其核心思想是使用 64 位的 MMX 寄存器，一次对多个整数进行算术运算。SSE 是 Streaming SIMD Extension 的缩写，即 SIMD 流扩展。SSE 将 SIMD 技术推广到可以对单精度浮点数进行单指令多数据计算。

与 x87 FPU 浮点异常很类似，SIMD 浮点异常也被划分成 6 个子类：

- 无效运算（#I）。
- 除零（#Z）。
- 非规格化（denormal）的操作数（#D）。
- 数值溢出（#O）。
- 数值下溢（#U）。
- 结果不精确（#P），P 代表精度（Precision）。

在以上六类异常情况中，无效操作、除零和非规格化的操作数属于计算前异常，也就是算当 CPU 检测到错误情况使还没有进行任何算术计算。数值溢出、数值下溢和结果不精确属于计算后异常。

对于以上六类情况的每一种，MXCSR 寄存器配备了一个屏蔽位和一个标志位分别用于屏蔽该类异常和报告该类异常发生。

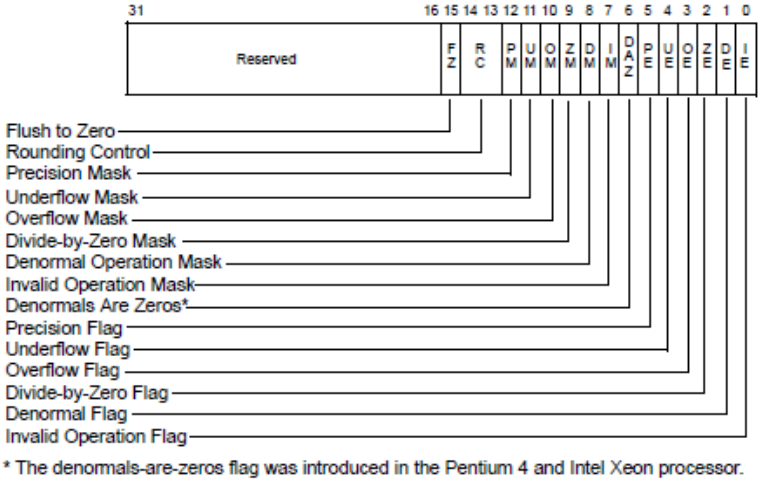


图 C-2　MXCSR 控制和状态寄存器

当 CPU 检测到以上异常情况时，如果该类异常被屏蔽（MXCSR 寄存器的对应屏蔽位为 1），那么 CPU 便会自动处理该异常，产生一个合理的结果，然后让程序继续运行。如果该类异常没有被屏蔽（MXCSR 寄存器的对应屏蔽位为 0），那么 CPU 会通过检查 CR4 寄存器的 OSXMMEXCPT 标志来判断操作系统是否支持 SIMD 浮点异常，如果 OSXMMEXCPT 标志为 1（表示操作系统支持 SIMD 异常），那么 CPU 便会产生一个 SIMD 浮点异常，然后执行该异常对应的异常处理例程。如果 OSXMMEXCPT 标志为 0（表示操作系统不支持 SIMD 异常），那么 CPU 会产生一个无效操作码（#UD）异常。

值得注意的是，与 x87 FPU 浮点异常的延迟处理策略不同，SIMD 浮点异常是被立即汇报的。因此 WAIT/FWAIT 或其它 SSE/SSE2/SSE3 指令遇到等待报告的 SIMD 浮点异常的情况是不会发生的。另一种情况是，当一个 SIMD 浮点异常发生时该类异常是被屏蔽的，这时 CPU 会设置 MXCSR 寄存器中对应的标志位，但不会报告该异常，而后如果取消屏蔽该

类异常，那么取消屏蔽时并不会导致 CPU 汇报这个前面发生的异常。

当 SSE/SSE2/SSE3 SIMD 浮点指令对一组操作数（包含 2 或 4 个子操作数）进行计算时，CPU 可能检测到多个 SIMD 浮点异常的情况。如果一个子操作数的异常情况不超过一个，那么 CPU 会为设置所有异常情况所对应的标志位。例如，为一个子操作数的无效异常设置异常标志位不会妨碍为另一个子操作数的除零异常设置对应的标志位。但是如果一个子操作数被检测到几个异常情况，那么 CPU 会根据表 C-5 所示的先后顺序只报告一种情况。表 C-5 所定义的顺序会使 CPU 汇报高优先级的错误情况，忽略低优先级的情况。

表 C-5　SIMD 浮点异常优先级

优先级	描述
1（最高）	因为 SNaN 操作数导致的无效运算异常，或者对任何 NaN 操作数进行最大值、最小值、或某些比较和转化运算
2	QNaN（并非异常，但是处理 QNaN 操作数比低优先级的异常更优先，比如 QNaN 除零会导致 QNaN，而不是除零异常）
3	任何上面没有提到的无效运算异常和除零异常*
4	非规格化操作数异常*
5	数值溢出或下溢异常，可能与结果不精确异常同时发生*
6（最低）	结果不精确异常

* 如果被屏蔽，那么指令会继续执行，可能会有第优先级的异常发生。

错误代码：无，可以通过 MXCSR 寄存器判断异常的进一步信息。

保存的程序指针：保存的 CS 和 EIP 值指向的是导致 SIMD 浮点异常的那条 SSE/SSE2/SSE3 指令。也就是从中检测到错误情况的那条指令。

程序状态变化：不会导致程序状态变化，因为处理器检测到错误情况后会立即报告异常（除非被屏蔽）。异常处理程序可以通过检查 MXCSR 寄存器和其它信息判断并纠正错误情况，然后恢复程序继续运行。

补编内容 10 《软件调试》导读

补编说明：

《软件调试》出版后，不少反馈是嫌这本书太厚了，不好读。当然不好读的原因也有书中的内容比较难懂。

其实作者写作这本书时，是很有一个让“略懂计算机的人就能读懂”的远大理想的。

为了鼓励一下大家的阅读兴趣，我在高端调试网站（http://advdbg.org）上写了一系列导读性的文章，至今已经完成下面四篇：
《软件调试》导读之提纲挈领
《软件调试》导读之绪论篇
《软件调试》导读之 CPU 篇
《软件调试》导读之操作系统篇
也把这个内容放入这个补编中吧，希望能有人觉得有点帮助。

《软件调试》导读之提纲挈领

拙作《软件调试》出版两个月了，有热心读者建议我讲些阅读这本书的方法。有读者愿意读自己的书，当然是好事，再说读者是客户，他们的意见就是命令，不能怠慢。粗略思考一番，计划先为《软件调试》的每一篇写一个导读短文。总为开篇，今日先谈谈《软件调试》这本书的篇章结构，用软件的术语就是架构，用写作的术语也就是提纲。

从最初的书名说起

早在 2003 年，我就萌生了写一本关于软件调试的书的念头。但是软件调试是个大话题，有很多东西可以写，必须选择好一个角度才能写出一本好书来。于是我开始搜索当时已经有的书，无论是美国出的，还是英国出的，一共找到了十来本。而后，逐一了解了已有的这些书，归纳了它们的主要内容和特色。

2004 年下半年，第一个版本的规划初步成型了，书名叫 Advanced System Debugging（《高级系统调试》）（简称 ASD）。针对的目标问题是系统级的调试任务，简单理解，就是在系统范围找 BUG，是与模块范围内的常规调试相对而言的。在当时的规划书中，我特意从以下四个方面比较了系统调试和常规调试的不同：

- Scope
  - System wide vs. Module/Product wide
- Addressed Issues
  - Application or OS Hang/Crash vs. Feature Failure
- Source Code
  - Not depends on source code vs. Based on source code
- Time Frame
  - System Debugging is more relevant with issues near or after product deployment

并强调系统调试需要不同的工具，并且更具挑战性：

- Addressed issues are more serious
  - Hang, Crash, Halt, auto Restart, etc.
- Locate Issues in system wide
  - Any component, including hardware, maybe the root cause
- Without source code and complete document
- Very broad knowledge is needed
  - OS, Hardware, Firmware, etc.

当时确定的主要内容有以下几个部分：

（一）调试基础

- How debugger works?
  - Break, stack trace, memory view, and variable watch
- CPU & OS support to Debugging
  - Post Mortem (JIT) debug, Attach Debugger
- General debug mechanism
  - Dump, asserts, event log, and debug outputs
- Debugging in software engineering

（二）异常

- Understand software and hardware exceptions;
- Exception handling mechanism;
- What if exceptions uncaught in user mode and kernel mode;
- Frequent exceptions.

（三）方法学和工具

- Advanced Inspecting
  - View system components inside;
  - Examine binary (program) files and process;
- Advanced Monitoring/Spying
  - Monitor system activities and kernel objects;
  - Exploring OS boot process;
- Advanced Tracing
  - Interrupt an application, driver, and OS;
  - Skills for WinDbg

（四）蓝屏（BSOD）

- Why BSOD?
- Interpret BSOD
  - Illustrate Stop Code one by one
- Enable and analyze memory dump
- Trace BSOD by WinDbg
  - Kernel debugging
- Advanced topics about BSOD
  - Crash handler
  - More serious issues than BSOD

（五）调试实践

- User mode debugging practice

- Dr. Watson, MiniDump
- Kernel mode debugging practice
- Kernel debug using COM, 1394 and Virtual PC
- Debug driver issues
- Debug ACPI issues
- Resolve tough S3/S1 issues by actual sample

现在回过头来看第一版计划，可以看到，其中包含了大多数后来要写的内容。而且这种从整个计算机系统的角度来着眼的思想一直保持到最后。

2005 年时的选题列选单

2005 年年初，开始和电子出版社协商出版计划。我开始进一步细化写作内容和篇章结构。于是第一个版本的章节计划产生了，下面是从当时的选题列选单中摘录下来的：

软件调试是软件开发及维护中最重要且最富有挑战性的工作之一，大多数软件工程师都认为他们 50%以上的工作时间是用在软件调试上的。但是软件调试无论是在软件工程实践中还是在学术界至今都还没有得到应有的重视，少数效率低下的调试方法仍在普遍使用，如何提高软件调试的效率和增强软件的可调试性还很少得到关注。特别是，纵观浩如烟海的计算机图书世界，目前还找不到一本系统全面阐述软件调试理论和实践的作品。本书正是出于这种考虑，力争填补软件领域和国内外出版界的一大空白。本书本着深入全面和理论与实践并重的原则，多方位的向读者展现软件调试的原理、方法和技巧。全书分为 4 篇，18 章。基础篇（1~4 章）除了介绍基本的概念和术语（第 1 章）外，系统的阐述了 CPU（第 2 章）、操作系统（第 3 章）和编译器（第 4 章）是如何支持软件调试的。开发篇（5~7 章）开创性的提出如何在软件工程的各个环节中，尤其是设计阶段，融入软件调试策略，提高软件的可调试性，以从根本上降低软件调试的复杂度，提高调试效率（第 5 章）。该篇不仅全面归纳比较了常用的提高软件可调试性的方法（第 6 章），还提出了一些新的模型和实现，有很强的实践参考价值（第 7 章）。工具篇（8~12 章）在对各类调试工具进行概括性介绍（第 8 章）后，深入的解析了三类常用调试工具的原理和用法以及一批经典工具。第 9 章在介绍微软的著名内核调试器 WinDbg 的同时，深入地揭示了内核调试的原理（目前还没有一本书包含此内容）。第 10 章通过深入解析微软.Net 调试器的源代码，介绍了目前流行的中间/脚本语言（比如.Net 和 Java）调试的原理。第 11 章以介绍 JTAG 原理为线索，探索了极富挑战性的嵌入式调试领域，介绍了如何调试常见的嵌入式系统（xScale 系统, ARM 系统等）。第 12 章把近百个经典、小巧、免费的调试工具归纳为几类，对它们做了个大检阅（这些工具是附带光盘工具箱的一部分）。实践篇（13~18 章）首先归纳了被国内外专家普遍认可的一些调试规则和方法（第 13 章），然后结合真实的案例，介绍了解决几类难度较大的调试问题的方法和技巧。第 14 章介绍了远程调试、RPC 调试等用户态调试任务。第 15 章在介绍非常热门的 Windows 内核/驱动程序调试的同时，还向读者揭示了如何探索 Windows 内核的一些技巧。第 16 章全面的探讨了著名的蓝屏崩溃问题。第 17 章介绍了如何在没有源代码和文档的情况下定位系统故障。第 18 章以最富挑战性的 ACPI 问题为例，探讨了如何利用前面介绍的方法和工具解决软件、硬件、固件相结合的棘手问题，并总结全书。

归纳一下，首先当时把要写的内容分为如下四篇：基础篇、开发篇、工具篇和实践篇。另外，将书名从 ASD 改为《软件调试》。现在看来，这一版本的架构与 AWD（Advanced

Windows Debugging）颇有相通之处，特别是基础篇和实践篇与 AWD 的第一篇和第二篇是一个思路。

软件调试的最初 300 页就是按照以上架构来写作的，写的是上面规划中的第 2 章和第 3 章。

重构

动笔后，更深的感觉到写书难。本来计划两个月完成的第 2 章，写到 2005 年年底也没完成。又因为春节的大块时间用来探索 Windows 调试子系统，所以 2006 年 3 月才完成第 2 章的第一稿。2006 年 8 月完成了第 3 章的初稿。这两章完成后，一个明显的问题是这两章的篇幅都很长，第 2 章有 100 页，第 3 章有 210 多页。这两章的长度让我觉得很不称心，我觉得一章太长，不易于阅读，也不方便编辑和排版。记得当时我还特意找了几本书，在 Windows Internals 中，有接近和超过 100 页的两章，第 3 章系统机制（98 页），和第 7 章内存管理（110 页）。

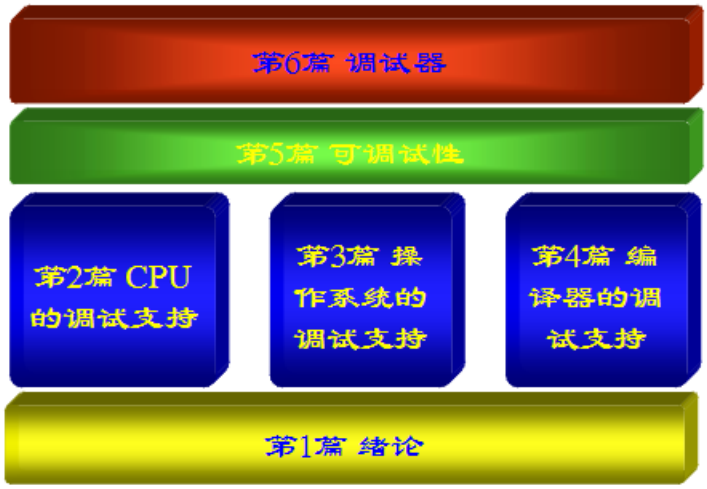
一边写作，一边思考了几周后，我终于下定决心重新组织结构。重构的一个指导思想是更侧重调试原理，将工程实践所需的基础知识和技能融入到原理中去，不再面向问题组织篇章和“就事论事”。 根据这一思想，做了如下几个大的改动：

- 将原来的 2，3，4 章升级为篇，以便更好的组织这些原理性和基础性的内容。
- 砍去实践篇，以便使这本书具有更好的通用性。因为实践篇中本来的内容还是面向问题的，深入讨论其中的每个问题域都可以单独写一本书，如果把这些内容放在同一本书中写，那么很难深入，喜欢一个问题域的读者通常也不需要深入了解另外的问题域。
- 在工具篇中，只集中讨论调试器，去掉本来安排的 10～12 章。

根据以上策略调整后，新的架构便是今天的样子，全书分为 6 篇，30 章。

目前的架构

下图中画出了 2006 年重构后的篇章结构，也就是目前使用的架构。



在新的架构中，2、3、4 篇是全书的核心，它们所描述的对象也恰好是计算机系统中的三个核心：即硬件核心 CPU，软件核心操作系统和生产软件的核心工具编译器。从调试的角度来看，这三个核心所提供的调试支持是支撑软件调试大厦的三块基石。或者说，上层的很多调试技术都是这三个核心所提供支持的应用。因此，了解这些调试支持是了解

调试技术的关键。

另外，从从事计算机相关工作的技术人员（本书读者）的角度来看，了解这三个核心也是至关重要的，对于提高软件开发能力、调试能力和对计算机系统的认知力都非常有益。从这个因素出发，第 2、3、4 篇的开头一章，即第 2 章，第 8 章和第 20 章，都是介绍这篇所描述核心的基础知识。

2～4 篇的总页数为 755 页，占全书篇幅的 70%。因此阅读和消化这三篇的内容是读懂这本书的主要任务。把这三篇内容搞懂了，就掌握了软件调试技术的基础和核心，同时也可以把对 CPU、操作系统、编译器这三大核心的理解提高到一个新的水准。以后，我会分别介绍每一篇的构思，给出一些阅读建议。但读到这里，读者应该清楚，全书的重心在 2-4 篇。在重心之下，是第 1 篇，即绪论，这是其它 5 篇完成后，最后写的一篇，目的是把读者带进门。

第 5 篇可调试性，尽管很短，只有短短的两章，总共 52 页，但是它的“思想地位”非常高。高效调试是学习和研究软件调试技术的初衷。如何实现高效调试呢？答案是调试过程中涉及的每一个部件都要大力支援、积极配合，至少不要抵触和反抗。2-4 篇介绍了计算机系统中的三大核心的调试支持，但如果被调试程序不配合，那么调试效率也会大打折扣。因此第 5 篇的第一个目的是探讨被调试软件的可调试性，介绍在软件设计和开发过程中就要考虑可调试性，未雨绸缪。第 5 篇的另一个目的是彰显可调试性这一主题，任何精心设计的系统都应该考虑可调试性，对于 CPU、操作系统、编译器这样的基础设施，不仅要考虑自身的可调试性，还要考虑如何支持应用软件的可调试性。

如果说，第 1 篇是全书内容的第一轮循环，2-5 篇是第二轮循环，那么第 6 篇便是第 3 轮循环，它以调试器为视角，在介绍调试器的实现方法和使用方法的同时，将前面 5 篇的内容“复习”了一遍。

调试器是软件调试的最核心工具，是每个软件高手必备的武器，深谙调试器兵法是《软件调试》这本书的核心目标，有人可能想，为什么不直接按照调试器来组织内容呢？我的确考虑过在第 1 篇就详细介绍调试器。但是后来没有这么做，原因有二。第一，对于今天的大多数调试器来说，它是与系统密切耦合的，夸张一点说，它只不过是底层调试功能的一个用户接口。因此即使把一个调试器的所有代码都分析一遍，那么还有很多东西搞不清楚。以 Windows 系统的用户态调试为例，调试器是建立在调试 API 之上的（《软件调试》第 18 章），很多调试功能不过是一个 API 调用就进入到系统代码了。以内核调试为例，WinDBG 不过是内核调试引擎（KD）的一个客户端（《软件调试》第 18 章）。第二，调试器有很多种，如果正面围绕调试展开，那么选择哪种调试器呢？如果选 WinDBG，那么整本书就变为《WinDBG 调试器 XXX》，这是我不希望的，不符合写作这本书的一般性原则。

在现在的架构中，调试器被安排在最后一篇，并不是降低它的地位，而是让其坐享前面的基础。对于读者，有了前面的基础再理解调试器会觉得很自然，有水到渠成之感。对于作者，写作这一篇时，也非常轻松，不时感觉到前面发散的内容在这里收敛了。另外，把调试器安排在其它 5 篇的上面也与它的“接口”身份更吻合。

归纳一下，《软件调试》的架构经历了三个版本。第一个版本侧重系统调试，书名为 ASD。第二个版本将写作范围扩大，书名推而广之为《软件调试》，内容划分为基础、开发、工具四篇。第三个版本提高第二版本中基础篇的位置，将重心明确在一般原理和具有共性的知识技巧上，缩减开发篇、工具和实践篇的内容。纵观这三个版本，版本 1 到版本

2 是扩张，版本 2 到版本 3 是精选和淘汰，前两个版本中的核心内容被细化，非原理性和适用面狭窄的内容被去掉了。特别是实践篇被去除了，其中的有些内容被融入到其它篇中，比如蓝屏崩溃被放入到第 3 篇，纳入到错误提示机制中讨论；栈溢出和内存泄漏被放入到第 4 篇，与编译器的有关支持一起讨论。而目标读者较窄的 RPC 调试和 ACPI 调试只好放弃了。放弃这些内容的一个长远规划是，在完成《软件调试》后，分专题写一系列实战性的短篇，自称为调试战役系列。概而言之，《软件调试》的着眼点是软件调试的一般原理，其目标是为所有喜欢调试技术的读者打下一个宽广而且坚实的基础，这个基础对于做调试是有用的，对于做开发也是有用的，对于解决迫在眉睫的问题有用，对于长远的职业发展也有用。为了实现通用性，那么只能多写具有共性的基础内容，舍弃细枝末节和具体问题，让读者掌握这些基础后，自己来举一反三，也就是常说的“授之以渔”。希望读者阅读《软件调试》时，能想起作者的这一良苦用心，这将有助于您理解书中的内容。

《软件调试》导读之绪论篇

《软件调试》的第 1 篇名为“绪论”，只包含一章，共 26 页，是全书最短的一篇。

第一篇要实现的几个目标是：

1）介绍基本概念和术语，为后面各篇打基础和做铺垫。比如，1.1.1 节给出了 Bug 和 Debug 的定义，1.6 节详细的讨论了 BUG，1.4 节介绍了常见的软件调试型态。

2）浏览本书要介绍的主要调试技术，比如，1.5 节分 10 个专题（三级小节）浏览了本后后面要深入讨论重要调试技术。

3）突显软件调试技术的关键特征和重要性，1.2 节（基本特征）和 1.7 节（与软件工程的关系）都是服务于这个目的的。

读者在阅读第 1 篇时，建议认真阅读以下几个内容：

1.1.2 节（P5）中的软件调试基本过程。这一内容虽然很基本，但是很重要。熟悉和遵循这个基本过程是对调试高手的最起码要求。如果不遵循这个基本过程，有时候就会白白浪费很多时间。比如，有些人没有在自己的调试环境下重现问题就开始跟踪代码，跟踪了几个小时后才发现原来问题根本不会在这个系统中发生，或者跟踪的版本就不对。

1.2 节中的软件调试基本特征。在写作这一内容时，我列出了很多个特征，然后进行筛选和提炼，最终归纳为三大特征：难度大、难以估计完成时间和广泛的关联性。理解软件调试的这三个特征对于学习软件调试和在实践中解决软件问题都很重要。因为软件调试技术与其它技术有着广泛的关联性，所以学习软件调试时必须本着海纳百川的心态去博学，而不能期望要用什么就学什么。因为“难以估计完成时间”，所以大家在工程实践中，面对一个软件调试问题时，不能轻易“拍胸脯”，不到有十足把握时，不能下断言。

1.3 节中的软件调试简要历史。两年前，某本杂志在做软件调试专题时，想找人写篇文章介绍软件调试的历史，找到我，我说深知这个内容不好写，而且当时没有时间，便推迟了，编辑很有信心的去找别人，但是始终没有找到。的确，还没有人仔细为软件调试编写历史，很多调试技术是何时出现的还没有定论。考虑到了解历史的重要性，《软件调试》有意做了很多努力，1.2 节介绍了断点、跟踪和分支监视的历史，28 章介绍了调试器的历史，第 29 章介绍了 WinDBG 的发展历史，另外，在书中的其它章节中，也尽可能给出所介绍技术或者工具的时间信息。《软件调试》的这些历史性内容在其他书中是很少见的。阅读这些内容，有利于更深入的理解调试技术。

从写作时间角度来看，第一篇是在后面 5 篇大局已定后才写的，不过在考虑篇章结构时早已经预留好这一篇。

概而言之，第一篇是很容易理解和阅读的，对于初学者来说，应该完整阅读全篇的内容，特别是 1.1 节和 1.5 节。对于其它读者，那么建议仔细阅读 1.2（特征）、1.3（历史）、1.6（对“错误与缺欠”的思考）和 1.7 节（关联性），浏览其它各节。

《软件调试》导读之 CPU 篇

《软件调试》的第 2 篇是 CPU 的调试支持，由第 2～7 章组成，共有 136 页，是全书的第一个核心部分。写作和阅读这一篇的主要目标有如下几个：

10. 介绍大多数软件工程师需要补充的 CPU 基础。
11. CPU 对软件调试核心功能的支持。
12. CPU 对软件调试扩展功能的支持。
13. CPU 中用于调试系统故障和自身问题的设施。
14. 现代 CPU 和集成芯片所使用的硬件调试方案。

针对以上目标，第 2、3 章是满足目标 1 的，4～7 章依次是满足另外四个目标的。下面对各部分的重点内容分别略作介绍。

一、介绍一个调试高手应该掌握的 CPU 层的基础知识。第 2 章和第 3 章是专门服务于这一目的的。调试好比行医看病，病人是计算机系统，要能看懂这个系统的毛病然后再对其施以治疗或者手术，那么必须了解其五丈六腑的结构，血脉流通的路线，生息运转的机理。要做到这一点，深刻理解计算机系统中硬件部分的核心——CPU——很重要。有人说，CPU 是重要，但有什么必要在一本《软件调试》的书中写这个呢？调试高手还需要数学基础和语文基础呢，怎么不开两章讲讲呢？这一拮问不是没有道理，因此作者考虑到这一点，慎重选择了要讲的内容，并严格控制了篇幅。入选的内容要符合三个条件：一是够重要，二是够常用，三是与调试密切相关。于是，《软件调试》最后选择如下一些内容：

指令集的概念（2.1 节）：有几次面试年轻的程序员时，我询问：“你熟悉哪种 CPU 架构和指令集呢？”不止一次，有人不能理解这个问题。“CPU 还有很多种（架构）吗？”x86 太成功了！指令是 CPU 的语言，理解指令集是为构筑软件知识大厦打下一块不可少的基石。

IA-32 处理器（2.2 节）：这也是一个备受胡略和误解的重要概念。有人说都进入 64 位时代了，还学 IA-32 干吗？殊不知，今天大多数 PC 使用的 x64 架构只是原有 32 位架构中的一种新的操作模式（e.g. IA-32e）。要理解 x64，还需要先理解 32 位的情况。或者说，如果有扎实的 32 位基础，那么可以很容易理解 64 位，反方向则很难走通。因为长达三十多年的广泛应用（从 1985 年 80386 的推出算起），IA-32 架构对计算机的影响太深入了，甚至超出了 CPU，影响到了系统总线和外设的设计。2.2 节使用 4 页半的篇幅全面浏览了已经推出的每一代 IA-32 CPU，从 386 到今天的 Core 2 系列。

CPU 的操作模式（2.3 节）：经常遇到的重要概念，扼要介绍。

寄存器（2.4 节）：将 IA CPU 的所有寄存器分为五类做了介绍：通用寄存器、标志寄存器、MSR 寄存器、控制寄存器和其它寄存器。除了介绍概念外，这一小节还有一个目的是用作“调试手册”，我在调试时，时常还翻到这几页内容，查寄存器的位定义。

保护模式（2.5-2.7 节）：这几乎是我做面试时必问的一个概念。深刻理解这个概念，才能深刻理解今天的计算机系统在如何求解计算机领域的一个永恒课题——多任务。保护

数据是保护模式的一个主要任务，虚拟内存是目前实现这一任务的主要方法。页机制是今天所有的商业操作系统都依赖的一种机制。理解虚拟内存和页机制对于软件调试也非常重要。因为如果搞不清楚这些概念，那么就会被虚拟地址、线性地址、物理地址、IO 地址这些概念所搞晕。也不容易建立起计算机世界的空间概念——内存空间、进程空间等。

中断和异常（第 3 章）：这两个概念有联系，又不同。有时又被混用，所以不少程序员对其模棱两可。IA-32 架构将异常分为三种，这更是很多人闻所未闻。记得有个陌生的青年写 EMAIL 给我，对《软件调试》76 页的说法“当 CPU 产生异常时，其程序指针是指向导致异常的下一条指令的”提出质疑：

“CPU 产生异常时，其程序指针不是指向原来那条指令吗？”

我回信解释说“因为异常不同，异常发生时程序指针的取值是不同的”，他更加困惑：

“我还是不能理解，难道异常还有两种吗？”

我知道了他迷惑的根本原因，把表 3-1“异常分类”发给了他，这下他彻底清楚了。是啊，如此重要的概念，我们的大学教育（包括计算机科学的研究生）里根本没有，也不能完全怪个人。

另外，3.4 节的“中断和异常优先级”是写给高水平读者的较难内容。

二、CPU 对软件调试核心功能的根本支持，即第 4 章。这是全书的核心内容之一，深入介绍了断点指令、调试寄存器和支持单步执行的陷阱标志。这三大支持是构筑今日调试技术的三大基石，很多至关重要的调试功能都是建立在这三个基础之上的，包括设置断点、变量监视、各种各样的跟踪执行、调试信息输出、内核调试等等。这一章共有 32 页，读者应该认真阅读每一页。4.4 节的实模式调试器例析也值得仔细读，最好是把 Tim Paterson 先生所写的汇编代码打印出来对照阅读（链接为：http://www.patersontech.com/dos/Docs/Mon_86_1.4a.pdf），同时又可以学习汇编和欣赏大师的“手笔”。

三、CPU 对软件调试扩展功能的硬件支持，即第 5 章。第 4 章介绍的根本支持是从 80386 开始就定形了的东西。软件在不断发展，调试支持明显跟不上速度。因此今天的调试技术很多时候很乏力。第 5 章介绍的分支记录和性能监视机制可以说是从奔腾开始的新一代处理器引入的最重要调试支持。它是今天的大多数软件分析（profiling）和性能调优（performance tune）工具所依赖的基础设施。

四、机器检查机制（第 6 章）。这是奔腾 CPU 引入的一个旨在报告硬件问题的错误记录和报告机制。这一机制对很多软件工程师可能都很陌生。了解这一机制，不仅可以学到这样一个 CPU 层的基础知识，而且有助于建立“可调试性”的思想，也就是第 5 篇重点讨论的东西。

五、硬件调试方案（第 7 章）。软件问题是千变万化的，甚至可以说，很难找到两个负责的软件问题是一模一样的。因此，不同的调试工具和调试技术都有它的适用范围，不是无所不能的。比如有些问题，就适合用用户态调试会话来跟踪，有些问题使用内核调试比较合适，有些问题只使用单纯的软件调试器可能根本不行。这时就要使用硬件调试工具来帮忙。JTAG 是 CPU 和其它大规模集成芯片普遍使用的测试和调试方案，广泛用于调试芯片自身、系统软件、固件和特殊复杂的软件问题。从某种程度上讲，基于 JTAG 技术的硬件工具只是为调试器访问调试目标提供了一种“硬件化”的通信方法。在大多数情况下，还是要依赖软件调试器这样的软件工具来实现各种调试。或者说，大多时候是把这种调试方式统一到纯软件的调试方案架构中，这样调试者就可以使用熟悉的调试功能来实现调

试。比如，INTEL 的 ITP 工具就以 COM 组件方式为 WinDBG 提供了服务，使 WinDBG 可以通过 ITP 工具进行内核调试，参见 http://advdbg.com/blogs/advdbg_system/articles/903.aspx。

最后想提一下 2.8 节——系统概貌。这一节用一页的篇幅介绍了今天 PC 系统的硬件架构，也就是图 2-13。在脑海中能有这样一幅图对于理解计算机系统很有好处。这里介绍了一系列常用的术语，比如南桥、北桥、总线、芯片组等。CPU、北桥和南桥是目前主板上的三颗最重要芯片。即将推向市场的下一代芯片组将把北桥的功能整合到 CPU 和南桥中，即所谓的“双芯片架构”，不过这仍不影响基本的概念，比如内存控制器（memory controller）依然存在，只不过是移到 CPU 内部。

总体来说，第 2 篇的内容都比较好理解，篇幅也不是很长。稍微用点毅力就可以将其通读一遍。希望读者看过后能对 CPU 的认识有一个明显的提高，使自己的硬件基础更扎实些。当然重中之重是调试支持，后面几篇还会接着讲.....

《软件调试》导读之操作系统篇

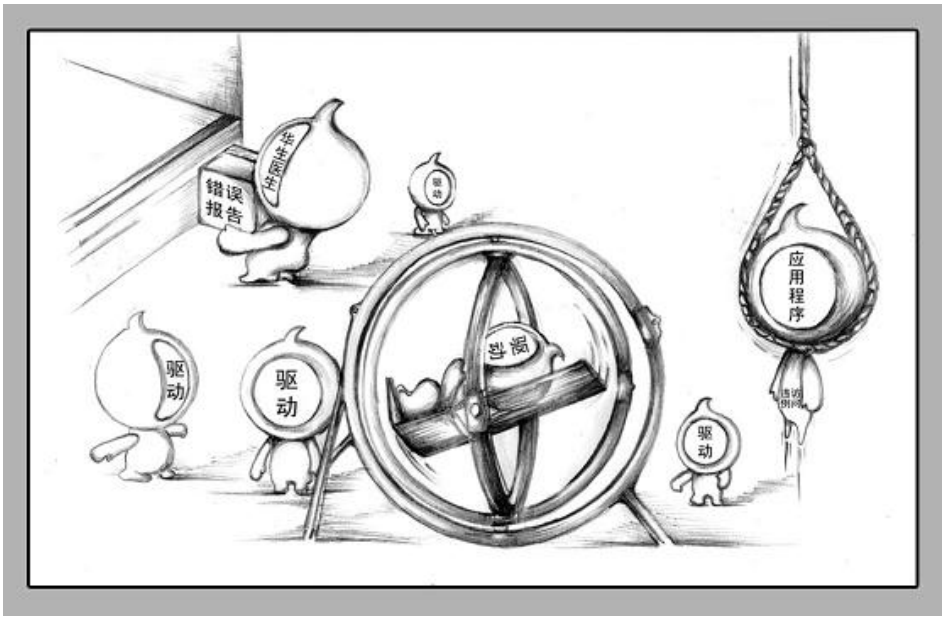
在今天的计算机架构中，操作系统是整个系统的统治者，它指挥着系统中的软硬件。如果拿人类社会来类比，那么操作系统好比是国家机器，应用软件是公民，操作系统的各个执行体是国家机构。从这个角度来看，操作系统与应用软件之间是统治与被统治的关系。

对于最终用户来说，他们需要的是应用软件，大多用户说不清什么是操作系统，只知道没有它不行。用户之所以肯掏出钱买操作系统是因为有了它才能跑自己需要的应用软件。从这个角度来说，应用软件是操作系统从用户那里拿到钱的资本，应用软件是前台唱戏的主角，操作系统是藏在后面的支持者。

概而言之，应用软件与操作系统之间既有统治关系，又有共生关系，应用软件对操作系统来说可谓是“船可载舟，亦能覆舟”。理解了这层意义之后，对于一个操作系统来说，它存活的关键就是要能运行尽可能多的应用软件，而且这些软件最好是用户离不开的，最好是在其它操作系统上无法运行的，最好是不断更新的，最好是高质量的，而且最好这样的软件会层出不穷、源源不断的涌出来.....

或者说，对一个操作系统来说，光把自己的代码弄好还远远不够，还要有能力跑缤纷多彩的应用软件，有魄力为应用软件搭建一个宽广伟岸的运行平台，有魅力吸引软件开发商（ISV）前仆后继的为其开发应用软件。深刻理解这一点不容易，做到这一点就更不容易了。

要有高质量的应用软件不容易，为了做到这一点，操作系统需要提供基础设施来支持。有点像很多好的公司都为员工准备健身房一样，操作系统也要给应用软件建设好磨练筋骨的地方，让它们身强体壮，身手敏捷，能适应各种复杂甚至恶劣的运行环境。



当然对于不守纪律的人，也要有纪律来惩罚，吊起来上刑（图中右侧那个）。对于生病了人，应该有医生帮助它治疗。

让 ISV 愿意在一个系统上做软件开发不是件容易的事，吸引一个 ISV 可能很简单，但是吸引成千上万的 ISV 就不容易了。软件开发不是请客吃饭，一个软件开发团队的日常开销不是小数目。还活着 ISV 大多懂得不能在螺蛳壳里做道场的道理。就像选择开发区做投资要考察它的基础设施一样，要在一个系统上做开发，也应该衡量下这个系统的基础设施怎么样。对于开发区来说，交通、水、电、煤气、网络等等都是头等重要的基础设施。而对一个操作系统来说，API、开发工具和调试设施可谓是直接影响软件开发效率的关键基础设施。

《软件调试》将操作系统的调试设施分为如下几类：

支持调试器的系统设施：因为调试器是软件调试的核心工具，所以支持调试器是操作系统支持调试的首要任务。在 DOS 这样的单任务系统中，调试器可以直接使用硬件中的调试设施，因此基本不需要操作系统的支持，但是在一个多任务的操作系统中，调试器可能也同时运行很多个，这就要求操作系统必须来统一管理和协调调试资源。另外，多任务环境下的诸多保护机制使得让操作系统来实现必要的“调试器功能”是最合适的，比如收集和分发调试事件。从运行模式角度来看，多任务环境下的软件有的运行在搞特权的内核模式下，有的运行在低特权的用户模式下。调试这两种不同模式下的软件有着很多不同，因此通常使用不同的调试器，即内核态调试器（第 18 章）和用户态调试器（第 9 章和第 10 章）。

异常处理：处理异常是软件开发中一个老生常谈的话题。也是很多程序员觉得难以理解和棘手的问题。操作系统能不能减轻程序员这方面的负担呢？如果能又是如何做的呢？在这个问题上，不同操作系统的做法有挺大的不同。举例来说，同样是 C++语言规范中的 try{}catch()结构，在某些系统下就可以捕捉到 CPU 产生的异常（有时称异步异常），而在某些系统上就不能捕捉。《软件调试》的第 11 章详细介绍了 Windows 操作系统中管理和分发异常的方法。如果应用软件自己没有处理异常又怎么样呢？第 12 章介绍了未处理异常的处置方法。很多人对 Windows 下异常处理机制的一个困惑是搞不清楚所谓的第一轮（First Chance）和第二轮（Last Chance）。看过上面两章后，可以把这个问题彻底搞清楚。

错误通知机制：指实时的向用户通报错误信息（第 13 章），通过对话框、声音、闪动窗口等。

错误报告机制：软件是要给客户用的，而且发布到客户手里的软件也会出问题。从 BUG 的成本曲线（《软件调试》图 1-9，P23）来看，解决发布后的 BUG 的成本是最高的。如何降低这个成本呢？普遍认可的一种方法就是让位于客户那里的软件为自己产生一份“生病”报告，整理，然后借助互联网发送出来（第 14 章）。

错误记录机制：指永久记录软件的运行过程，特别是遇到异常和错误时的情况（第 15 章）。

事件追踪机制：错误记录机制通常不适合频繁的输出，如果频繁输出那么不仅会明显影响系统的性能，而且可能导致记录文件很大，难以检索有用的信息。而事件追踪机制就是针对这一需求而设计的，因为是使用专门的内存缓冲区，而且具有动态开启机制，所以它能够承受频繁的信息输出，而且开销不大（第 16 章）。

验证机制：根据 BUG 的成本曲线，越早发现问题越好，但是做到早发现问题并不容易。一种方法就是一个更严格的标准进行测试，让被测试软件在更苛刻的条件下运行，故意为其设置障碍来考验它。如何实施这些考验呢？操作系统的验证机制就是满足这一需要的（第 19 章）。

硬件错误管理机制：有些严重的崩溃和挂住是与硬件有关的，但是却找不到进一步的信息来定位到根源，以便排除或者在以后的产品中改进。PCIe 总线标准中制定了报告错误的通用机制，CPU 中也有机器检查设施（第 6 章），但是这些硬件设施是需要软件，特别是系统软件的配合才能发挥作用的（第 17 章）。

打印调试信息：打印调试信息（Output debug info 或者 Print）是一种简单易用的辅助调试方法。这种方法的不足就是效率比较低，不仅运行效率低，而且增加和减少需要重新编译，时间成本很高（10.7 节）。

崩溃和转储机制：转储是最古老的调试方法之一，简单说就是把内存数据“拍张照片”保存下来。在 Jack Dennis 为《软件调试》写的《历史回眸》短文中就提到了这种方法，当年是先使用一个工具程序将内存中的数据显示到 CRT 上，然后用照相机拍下 CRT 上的内容，最后再使用胶卷阅读器来阅读冲洗出来的胶片。这可真是给内存“拍照”（12.9 节和 13.3 节）。

为了让读者对以上调试设施有一个全面的理解，《软件调试》使用了三分之一的篇幅进行介绍，分 12 章，总页数达 376 页之多。这样的篇幅也使这一篇成为全书六篇中最长的一篇。为什么花这么大篇幅呢？

1）第 9、10、18 章分别介绍的是用户态和内核态调试模型，是理解调试器的基础，因此理当不惜笔墨。这三章分别是 34 页、46 页和 52 页，加起来为 132 页，占这一篇的三分之一。

2）第 11 章和 12 章介绍异常分发和未处理异常，这些内容不仅与调试器有着密切关系，而且是本书的“异常”主题的核心内容。这两章一共有 86 页。

3）其它 7 章肩负着介绍上面提到的其它辅助调试设施的责任，一共用了 158 页，平均每章 22 页。介绍这些内容一则是对调试有全面的理解，在软件工程中使用这些设施，另外也可以帮助大家更好的理解操作系统，提高综合调试能力。

另一个问题是以什么方式来介绍这些调试设施。是以一个具体的操作系统为例详细介绍，还是以抽象理论为主，偶尔举例。《软件调试》采用的是前一种方法，而且选择的是 Windows。为什么选择 Windows 呢？主要原因是它的广泛性。为什么没有选择 LINUX 呢？主要原因是它在调试方面还有很多足。很多资深的 LINUX 也不讳言 LINUX 在调试方面的缺欠，直至今天，LINUX 下最普遍使用的调试方法依然是 PRINT。事实上，选择 LINUX 来写，会好写很多，毕竟有现成的源代码可以读。

尽管书中多次明显提到选择 Windows 只是将其作为一个操作系统实例来介绍操作系统对软件调试的支持，但是还有一些人无法理解。不过，在买了书的读者中，还是理解的人多，其中也有一些专业做 LINUX 的工程师或者讲师。

上面介绍了这一篇的写作目的、主要内容和结构布局，下面再推荐一下阅读的顺序。对于初级读者，建议先泛读第 9、10 两章之外的所有章节，然后仔细读第 11 章和 12 章，再后则读第 13 章中的《硬错误和蓝屏》。对于有一定调试基础的中级读者，可以根据自己的兴趣仔细读感兴趣的章节。对于高级读者，可以先读第 9、10 和 18 章，然后浏览其它章节，遇到感兴趣的仔细阅读。

[12 月 31 晨略作修改，增加插图]

补编内容 11 “调试之剑”专栏之启动系列

补编说明：

大约从 2005 年开始，我陆续在《程序员》杂志上发表一些关于软件调试的文章，最早的一个系列是“CPU 的调试支持”，而后又写了几期后便中断了。2008 年 9 月，《软件调试》出版后，《程序员》杂志建议我继续以前的调试专栏，并命名为调试之剑。

新的“调试之剑”专栏开始后，我写的第一个系列便是“系统启动系列”，已经发表了下面四篇文章：

举步维艰——如何调试显示器点亮前的故障

权利移交——如何调试引导过程中的故障

步步为营——如何调试操作系统加载阶段的故障

百废待兴——如何调试内核初始化阶段的故障

以上文章发表后，虽然收到几封读者的来信（通过编辑），总的来说反响甚是冷淡。可能大多数人都提不起来兴趣学习这些没用的东西。

举步维艰——如何调试显示器点亮前的故障

显示器是个人计算机（PC）系统中必不可少的输出设备，它是计算机向用户传递信息的首要媒介。用户也正是通过显示器来观察计算机所作的“工作”，与其交流。离开了显示器，我们便很难知道计算机在干什么。因为这个原因，在计算机系统启动的早期，要做的一个重要任务就是初始化显示系统以便可以通过显示器输出信息，俗称点亮显示器。

对于今天的大多数个人计算机，从用户按下电源按钮到显示器被点亮通常在一秒钟左右。对人类而言，这是一个稍纵即逝的时间。但对计算机系统和 CPU 而言，这一秒钟要完成很多任务。如果中间遇到障碍，那么便可能停滞不前，出现显示器迟迟没有被点亮的现象。今天我们就由浅入深的谈一谈遇到这种情况时该如何处理。考虑到笔记本系统的差异性较大，我们将以典型的台式机系统（即所谓的 IBM 兼容 PC）为例。为了辅助记忆，我们不妨套用一下我国中医使用的“望闻问切”方法。

望——不要闹笑话

首先，应该“望一望”主机和显示器的电源是否都插上了，它们的指示灯是否正常，它们之间的连线是不是连接妥当。这样做的目的是在“大动干戈”之前做好基本的检查，防止费了很多力气最终才发现是插头松了这样的低级问题，闹出笑话。不过这些检查靠常识就足够，没有什么技术含量，我们不去赘述。

闻——听声识原委

中医中的“闻”既包含用耳朵听，也包含用鼻子闻——嗅。这两种途径对我们也都适用。

我们先来谈如何靠听来了解计算机系统的病在哪里。尽管今天的个人计算机主要是靠声卡（或者集成在芯片组中集成音频设备）来播放声音的，但是在个人计算机诞生之初并没有声卡，甚至到了上世纪九十年代初笔者购买电脑时，典型的 PC 系统仍没有声卡，原因是价格很贵。在声卡出现之前，图 1 所示的扬声器是 PC 系统上的主要发声设备。

图 1 位于机箱上的 PC 喇叭

图 1 中的照片是从一台大约购于 2000 年的旧电脑中拍摄的。那时声卡设备便比较普及，PC 喇叭的用途变得越来越少，为了节约成本，今天的 PC 系统通常用一个位于主板上的小蜂鸣器（Beeper）来代替 PC 喇叭（图 2），二者虽然外观有很大的不同，但是工作原理是完全一样的。因此我们仍使用统一的名字来称呼它们。

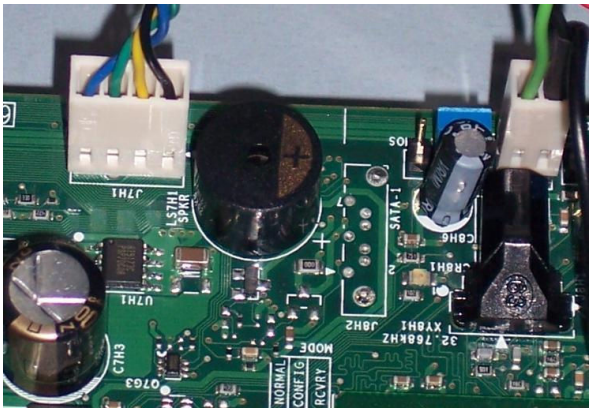


图 2 位于主板上的蜂鸣器

对程序员来说，使用 PC 喇叭的方法非常简单，只要将 I/O 端口 0x61 的最低两位都写为 1 便可以让 PC 喇叭开始鸣叫；将最低两位中的某一位置为 0 便可以让它停止鸣叫。通过一个小实验可以很方便的感受一下。使用 WinDBG 启动一个本地内核调试会话，然后使用端口输出命令来读写 0x61 端口，这样便可以开关 PC 喇叭。具体来讲，首先使用 ib 命令读出端口 0x61 的当前内容：

```
lkd> ib 0x61
00000061: 30
```

然后，把读到值的低三位置为 1，使用 ob 命令输出（执行前做好心理准备，叫声可能很刺耳）：

```
lkd> ob 0x61 30|3
```

此时读取这个端口的内容，可以看到端口值的低两位都为 1。听得不耐烦了吧，那么赶紧执行下面的命令将其停止：

```
lkd> ob 0x61 30
```

事实上，端口 0x61 的位 0 的含义是启用 PC 系统中的可编程时钟计数器（Programmable Interval-Timer/Counter，通常称为 8253/8254）的 2 号通道（共有三个，分别为 0、1 和 2）使其输出一定频率的方波脉冲，刚才听到的鸣叫声正是这个方波输出给 PC 喇叭而发出的。端口 0x61 的位 1 相当于给时钟控制器的输出加一个开关，或者说加了个与门（图 3）。

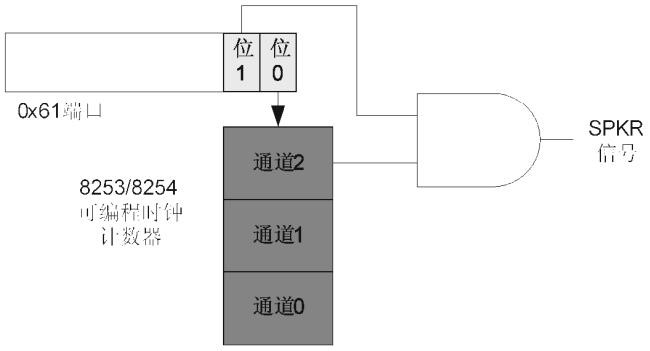


图 3　PC 喇叭的工作原理示意图

时钟控制器的输出频率是可以改变的，以变换的频率来驱动 PC 喇叭可以使其发出一些简单的“旋律”。

因为大多数 PC 系统都有 PC 喇叭，而且对软件来说，使用 PC 喇叭输出声音非常简单，所以计算机系统的设计者们很自然地想到了在 PC 启动早期使用蜂鸣器来报告系统遇到的错误情况。虽然播放复杂的声音很困难，但是可以使用蜂鸣的次数或者每次蜂鸣的长短不同来代表不同的含义。因为这种错误信息是通过 PC 喇叭以鸣叫的方式报告的，所以通常称为蜂鸣代码（Beep Code）。

当按下 PC 机的电源按钮后，首先运行的是固化在系统主板上的固件程序（firmware），通常称为 POST 程序，POST 是 Power On Self Testing 的缩写，含义是上电自检。不同的 POST 程序（固件），定义蜂鸣代码的方式也有所不同。表 1 中列出的是英特尔主板通常使用的蜂鸣代码。

表 1　英特尔主板所使用的蜂鸣代码

蜂鸣代码	含义
鸣叫 1 声	DRAM 刷新失败
鸣叫 2 声	校验电路失败
鸣叫 3 声	基础 64K 内存失败，可能是没有插内存条或者内存条松动
鸣叫 4 声	系统时钟失败
鸣叫 5 声	处理器（CPU）失败
鸣叫 6 声	键盘控制器失败
鸣叫 7 声	CPU 产生异常
鸣叫 8 声	显卡不存在，或者显卡上的显存失败
鸣叫 9 声	固件存储器中内容的校验和与固件中记录的不一样
鸣叫 10 声	读写 CMOS 中的数据失败或者其内容有误
鸣叫 11 声	高速缓存失败

通常，在固件厂商的网站或者产品手册中可以查找到蜂鸣代码的含义。例如通过以下链接可以访问到英特尔主板/固件的蜂鸣代码含义：http://www.intel.com/support/motherboards/desktop/sb/cs-010249.htm

在以下网页中列出了其它几种常见固件的蜂鸣代码定义：http://www.computerhope.com/beep.htm

关于 PC 喇叭，还有两点需要说明。第一点是，有些固件在正常完成基本的启动动作后会鸣叫一声，这并不是报告错误，而是报告好消息。第二点是台式机的 PC 喇叭通常是不受静音控制的，而笔记本电脑的 PC 喇叭是受静音控制的，因此在诊断笔记本电脑时应该调整音量按钮取消静音，这样才可能听到蜂鸣代码。

下面再谈一下闻的另一种含义——“嗅”，也就是闻味道。当出现显示器无法点亮这样的故障时，确实可能是某些硬件损坏了，比如电容被击穿和短路等。因此在每次开机和调试时，不妨用鼻子闻一闻，如果闻到烧焦味道，那么应该立刻切断电源；如果有其它事情需要离开，那么也该先给系统断电。

问——黑暗中交谈

在显示器被点亮前，系统通常还不能接收键盘和鼠标输入，这时该如何询问它呢？一种简单的方法是改变系统的配置或者调换系统的部件，然后通过聆听蜂鸣代码或者观察它的其它反应来感知计算机的“回答”，以便收集更多的信息。举例来说，有一个故障系统，按下电源后很久，显示器仍不亮，也听不到任何蜂鸣声音。这时，可以先切断电源，拔下所有内存条，然后再上电开机，如果听到三声鸣叫，那么便说明系统已经执行到内存检查部分，这可以初步证明 CPU 是正常的，系统的主板也是可以工作的。

切——接收自举码

中医中的切是指把脉，也就是通过感受患者的脉搏来了解健康状况。那么如何能感受计算机系统的脉搏并从中提取出它的生命信息呢？PC 系统的开拓者们真的设计出了一种方法。简单来说，就是将一种名为“上电自检卡（POST Card）”的标准 PC 卡插到系统的扩展槽中，让这块卡“切”入到目标系统中来监听系统总线上的活动，接收上面的数据。上电自检卡通常是 PCI 接口的，也有 ISA 接口的。图 4 中的照片便是一个 PCI 接口的上电自检卡。

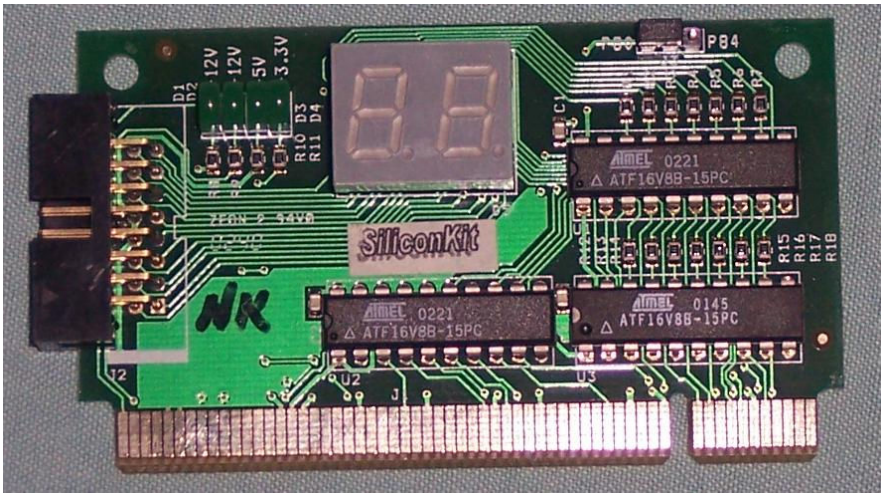


图 4　PCI 接口的上电自检卡

为了支持调试，POST 程序在执行的过程中，会将代表一定含义的 POST 代码发送到 0x80 端口。系统硬件会将发送到这个端口的数据发送到 PCI 总线上，于是上电自检卡便可以从总线上读取到 POST 代码，然后显示出来。POST 程序会使用不同的 POST 代码代表不同的含义，有些代表错误号，有些代表进展到了哪个阶段。通常可以在产品的技术文档中查找到 POST 代码的含义，然后根据这个含义来了解故障的原因。

透视和跟踪

使用上面介绍的四类方法，通常可以定位出导致故障的部件或者粗略的原因，对于普通的测试或者维修目的，做到这一步也就可以满足要求了。那么对于需要修正故障或者想深入研究的开发人员该如何进一步分析出精确的故障位置呢？如果是软件错误，那么能不能分析出是哪段程序或者哪条指令出错了呢？

要做到这一点，比较有效的方法是使用调试器。因为这个时候系统还在初始化阶段，纯软件的调试器还不能工作，所以这时需要硬件调试器，也就是《软件调试》第 7 章介绍的基于 JTAG 技术的 ITP/XDP 调试器或者同类的硬件工具。

使用硬件调试器可以单步跟踪执行 POST 程序；可以设置断点，包括代码断点（执行到指定地址的代码时中断）、内存访问断点（访问指定的内存地址时中断）和 IO 访问断点（访问指定的 IO 地址时中断）；也可以在发生重要事件时中断下来，比如进入或者退出系统管理模式（SMM）时中断、进入或者退出低功耗状态时中断、或者系统复位后立刻中断等。举例来说，在将系统复位事件的处理方式设置为中断（break）并重启系统后，CPU 复位后一开始执行便会中断到调试器中。观察此时的寄存器值（图 5），代码段寄存器 cs=f000，程序指针寄存器 eip=0000fff0。因为这时 CPU 工作在实模式下，所以目前要执行代码的物理地址是 0xf000 x 16 + 0xfff0 = 0xffff0，这正是 PC 标准中定义的 CPU 复位后开始执行的程序地址，PC 系统的硬件保证这个地址会指向位于主板上的 POST 程序。因此可以毫不夸张的说，以这种方式中断到调试器中可以得到“最早的”调试机会，从 CPU 复位后执行的第一条指令开始跟踪调试。

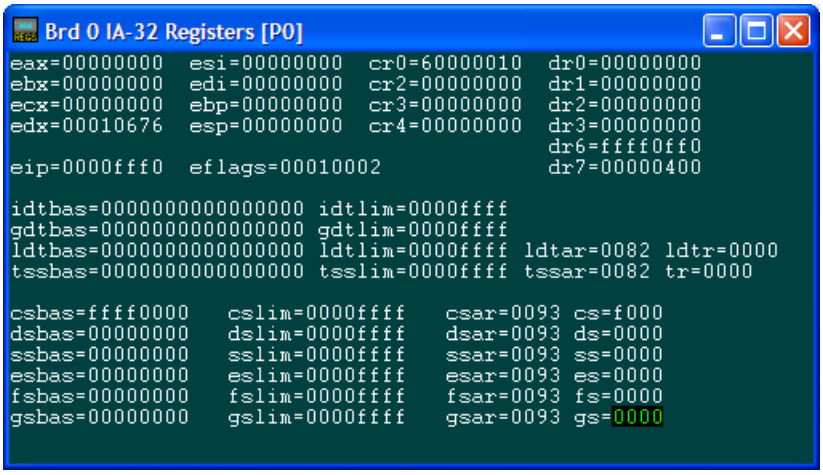


图 5 CPU 复位后的寄存器值

接下来，使用断点功能对端口 0x80 设置一个 IO 断点，然后恢复 CPU 执行：

```
[P0]>go
```

结果，这个断点很快便命中了，调试器显示：

```
[ Debug Register break at 0x0010:00000000fffffeca in task 00000 ]
[[P1] BreakAll break at 0xf000:0000000000000000 ]
```

因为系统中的 CPU 是双核的，所以第 1 行显示的是 0 号 CPU（P0）的中断位置，第 2 行显示的是 1 号 CPU 的中断位置，其程序指针寄存器的值为 0，还没有开始工作。使用反汇编指令可以观察断点附近的指令：

```
[P0]>asm $-2 length 10
0x0010:00000000fffffec8   e680        out 0x80, al
0x0010:00000000fffffeca   e971f9ffff   jmp $-0x0000068a  ;a=fffff840
```

```
0x0010:00000000fffffecf   680000e4ff   push 0xffe40000
0x0010:00000000fffffed4   68c4faffff   push 0xfffffac4
…
```

可见，中断前执行的正是向 0x80 号端口输出的指令。观察一下 al 寄存器，它的值为 1，看一下上电自检卡上显示的数字也是 1，正好吻合。

归纳

今天，我们介绍了一种比较特殊的调试任务。之所以介绍这个内容，除了让大家了解上面介绍的调试方法外，还有两个目的。一是学习 PC 系统的设计者们以不同方式支持调试的聪明才智和重视调试的职业精神，他们在打印信息或者显示文字等方法不可行的情况下，设计出了蜂鸣代码和 POST 代码这样的调试机制，这些机制看似简陋，但是却可以传递出来自系统第一线的直接信息，实践证明这些信息可以大大提高调试的效率。二是希望能提高大家对计算机硬件和整个系统的兴趣，程序员的主要目标是编写软件，但是对硬件和系统的深刻理解对于程序员的长远发展是有非常有意义的。

下一期的问题：

一台安装 Windows 的计算机系统开机后显示因为系统文件丢失而无法进入系统，对于这样的问题有哪些方法来调试和解决？

权利移交——如何调试引导过程中的故障

上一期我们讨论了如何调试显示器点亮前的故障，在文章中我们提到，CPU 复位（Reset）后，首先执行的是固化在主板上的 POST 程序（图 1）。POST 程序的核心任务是检测系统中的硬件设备，并对它们做基本的检查和初始化，并根据需要给它们分配系统资源（中断、内存和 IO 空间等）。POST 程序成功执行后，系统接下来要做的一个重要任务便是寻找和加载操作系统（OS）。对于不同的计算机系统和不同的使用需求，需要加载的操作系统可能位于不同的地点。最常见的情况是操作系统位于硬盘（Hard Disk）上，但是也可能位于光盘、优盘、软盘或者网络上。

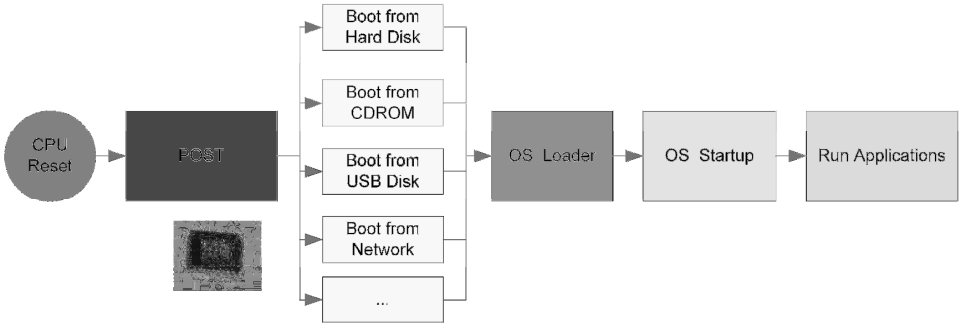


图 1 计算机的启动过程

通常把寻找和加载操作系统的过程叫做引导（Boot 或者 Bootstrap），也就是图一中的黄色方框。本期我们就谈谈引导有关的问题，介绍如何分析和调试的这个过程中可能发生的故障。

BBS——BIOS 引导规约

考虑到引导过程涉及到来自不同厂商生产的不同部件之间的协作，因此需要一个标准来定义每个部件的职责和各个部件之间交互的的方法，在这种背景下，英特尔、Phoenix 和康柏公司在 1996 年联合发布了 BIOS 引导规约（BIOS Boot Specification），简称 BBS（图 2）。尽管十几年已经过去了，但是这个规约中的大多数内容至今仍被使用着。本文中使用的很多术语和数据结构都来自这个规约。在互联网上搜索 BIOS Boot Specification，可以下载到 BSS 的电子版本。

图 2 BIOS 引导规约

IPL 表格

BBS 把系统中可以引导和加载 OS 的设备成为初始程序加载设备(Initial Program Load Device)，简称 IPL 设备。BIOS 会在内存中维护一个 IPL 表格，每一行（表项）描述一个 IPL 设备。表 1 列出了 IPL 表的各个列（表项的字段）的用途和详细情况。

表 1　IPL 表项的各个字段

名称	偏移	长度（字节）	描述
deviceType	00h	2	设备标号，参见下文
statusFlags	02h	2	状态标志
bootHandler	04h	4	发起引导的代码的地址
descString	08h	4	指向一个以零结束的 ASC 字符串
expansion	0ch	4	保留，等于 0

其中的 deviceType 字段用来记录代表引导设备编号的数字，01h 代表软盘，02h 代表硬盘，03h 代表光盘，04h 代表 PCMCIA 设备，05h 代表 USB 设备，06h 代表网络，07h..7fh 和 81..feh 保留，80h 代表以 BEV 方式启动的设备(我们稍后详细讨论)。接下来的 statusFlags 字段用来记录它所描述的引导设备的状态信息，使用不同的二进制位代表不同的状态，图 2 画出了各个位域，Old Position 位域（bit 3..0）代表上次引导时这个表项在 IPL 表中的索引，Enabled 位域（位 8）用来启用（1）或禁止（0）这个表项，Failed 位域（位 9）为 1 代表已经尝试过使用该表项而且得到了失败的结果，Media Present 位域（位 11..10）的典型用途是描述驱动器是否有可引导的媒介（光盘、磁盘），0 代表没有，1 代表未知，2 代表有媒介，而且看起来可以引导。

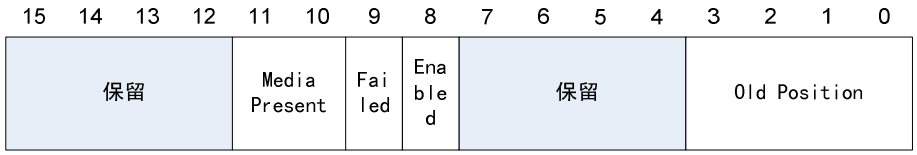


图 3 IPL 表的状态标志字段（statusFlag）的位定义

从编程的角度来看，可以使用下面的数据结构来描述 IPL 表的表项：

```
struct ipl_entry {
  Bit16u deviceType;
  Bit16u statusFlags;
  Bit32u bootHandler;
  Bit32u descString;
  Bit32u expansion;
};
```

在 EFI（Extensible Firmware Interface）中，使用 BBS_TABLE 结构来描述 IPL 设备，希望了解其具体定义的读者可以到 http://www.uefi.org/下载详细文档。

引导设备分类

BBS 将引导设备划分为以下三种类型：

- BAID – 即 BIOS 知道的 IPL 设备（BIOS Aware IPL Device），也就是说 BIOS 中已经为这样的设备准备了支持引导的代码。第一个软驱、第一个硬盘、ATAPI 接口的光驱等都属于这一类型。
- 传统设备 – 是指带有 Option ROM（见下文）但没有 PnP 扩展头的标准 ISA 设备。例如已经过时的通过 ISA 卡连接到系统中的 SCSI 硬盘控制器。
- PnP 设备 – 是指符合 PnP BIOS 规约（Plug and Play BIOS Specification）的即插即用设备。

因为第二类设备已经很少见，所以我们重点介绍一下从另两类设备引导的方法。

从即插即用（PnP）设备引导

这需要先了解什么是 Option ROM。所谓 Option ROM 就是在位于 PCI 或者 ISA 设备上的只读存储器，因为这个存储器不是总线标准规定一定要实现的，所以叫 Option ROM（可选实现的 ROM）。Option ROM 里面通常存放着用于初始化该设备的数据和代码。显卡和网卡等设备上通常带有 Option ROM。

PnP BIOS 规约详细定义了 Option ROM 的格式。简单来说，在它的开始处，总是一个固定结构的头结构，称为 PnP Option ROM Header，为了行文方便，我们将其简称为 PORH。在 PORH 的偏移 18h 和 1Ah 处可以指向另外两个结构，分别称为 PCI 数据结构和 PnP 扩展头结构（PnP Expansion Header），我们将其简称为 PEH。PEH 中有一个起到链表作用的 Next 字段（偏移 06h，长度为 WORD）用来描述下一个扩展结构的偏移。

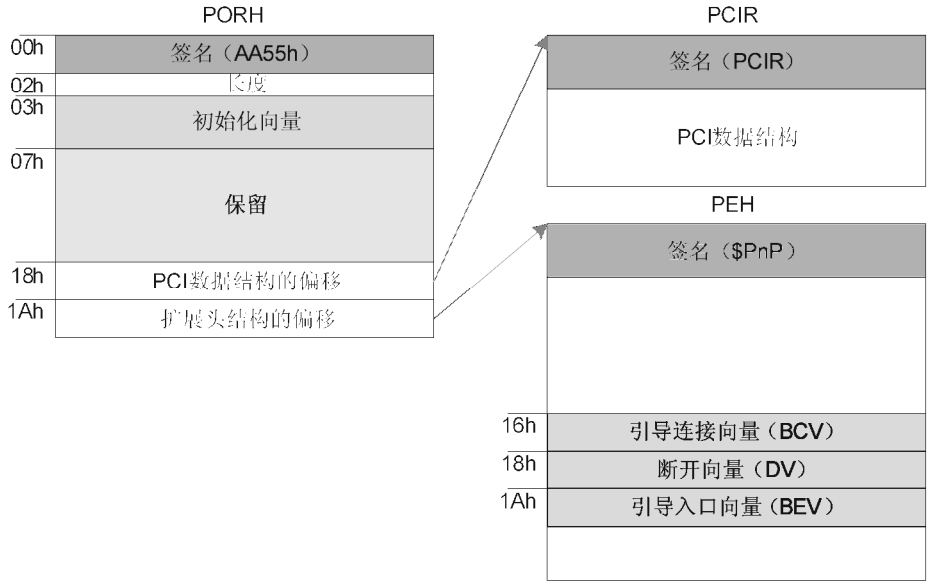


图 4　PnP 设备 Option ROM 中的头结构

首先，所有的 Option ROM 的头两个字节都是 0xAA55，因此在调试时可以通过这个签名来搜索或者辨别 Option ROM 的头结构。另外，PC 标准规定，0xC0000 到 0xEFFFF 这段物理内存地址空间是供 Option ROM 使用的。

在图 4 中，以黄颜色标出的向量字段与引导有着比较密切的关系，下面分别作简单介绍：

- **初始化向量** – 系统固件在引导前会通过远调用执行这个地址所指向的代码，这就是通常所说的执行 Option ROM。Option ROM 得到执行后，除了做初始化工作外，如果该设备希望支持引导，那么可以通过改写（Hook）系统的 INT 13h（用于读写磁盘的软中断）和输入设备来实现，上面提到过的传统 SCSI 硬盘就是这样做的。对于 PnP 设备，应该使用下面的 BCV 或者 BEV 方法。
- **引导连接向量（Boot Connect Vector）** - 这个向量可以指向 Option ROM 中的一段代码（通过相对于 Option ROM 起始处的偏移），当这段代码被 BIOS 调用后，它可以根据需要改写（Hook）INT 13h。
- **引导入口向量（Boot Entry Vector）** – 用来指向可以加载操作系统的代码的入口，当系统准备从这个设备引导时，那么会执行这个向量所指向的代码。下面介绍的从网卡通过 PXE 方式启动就是使用的这种方法。

图 5 所示的是网卡设备的 Option ROM 内容，第 1 列是内存物理地址，后面四列是这第一地址起始的 16 字节数据（以 DWORD 格式显示，每 4 字节一组）。图中第一个黄颜色方框包围起来的 32 字节是 PORH 结构，它的 0x1A 偏移处的值 0x60 代表的是 PEH 结构的偏移，因此下面的方框包围起来的便是这个扩展结构。

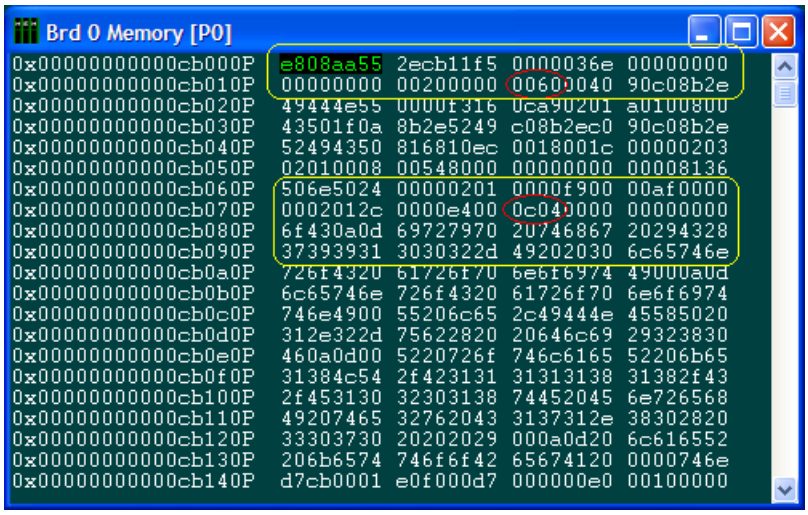


图 5 观察 PnP 设备的引导入口向量

根据图 4，在 PEH 结构的 0x1A 处的两个字节便是 BEV 向量，也就是 0x0c04。因此，当在 BIOS 中选择从这个网卡引导时，BIOS 在做好引导准备工作后，便会通过远调用来执行 0xcb00:0c04 处的代码。在调试时，如果对这个地址设置断点，那么便会命中。

INT 19h 和 INT 18h

BBS 还定义了两个软中断来支持引导，它们分别是发起引导的 INT 19h 和使用某一设备引导失败后恢复重新引导的 INT 18h。

下面列出的是 BBS 中给出的 INT 19h 的伪代码。

```
IPLcount = current number of BAIDs and BEV devices at this boot.
FOR (i = 0; i < IPLcount; ++i)
   currentIPL = IPL Priority[i].
   Use currentIPL to index the IPL Table entry.
   Do a far call to the entry's boot handler or BEV.
   IF (control returns via RETF, or an INT 18h)
       Clean up the stack if necessary.
   ENDIF
Execute an INT 18h instruction.
```

其中，第 5 行的远调用便是把执行权交给了用于引导当前 IPL 设备的过程，如果这个调用成功，那么便永远不会返回。

下面是 INT 18h 的伪代码。

```
Reset stack.
IF (all IPL devices have been attempted)
   Print an error message that no O/S was found.
   Wait for a key stroke.
   Execute the INT 19h instruction.
ELSE
   Determine which IPL device failed to boot.
   Jump to a label in the INT 19h handler to try the next IPL device.
ENDIF
```

需要说明的是，上面的伪代码完全是示意性的，实际的 BIOS 实现会更复杂而且可能有所不同。

使用 Bochs 调试引导过程

除了可以使用 ITP 这样的硬件调试器来调试引导过程外，某些情况下，也可以使用虚拟机来调试。具体来说就是把要调试的固件（BIOS 或者 EFI）文件配置到虚拟机中，然后利用虚拟机管理软件的调试功能来调试。例如，Bochs 虚拟机便具有这样的功能。Bochs 目前是一个开源的项目，可以从它的网站 http://bochs.sourceforge.net/上下载安装文件和源代码。

图 6 中的屏幕截图便是使用 Bochs 调试的场景，大的窗口是虚拟机，重叠在大窗口上的小窗口是 Bochs 的控制台，在里面可以输入各种调试命令。图中显示的是设置在 INT 19h 入口处（0xf000:e6f2）的断点命中时的状态。

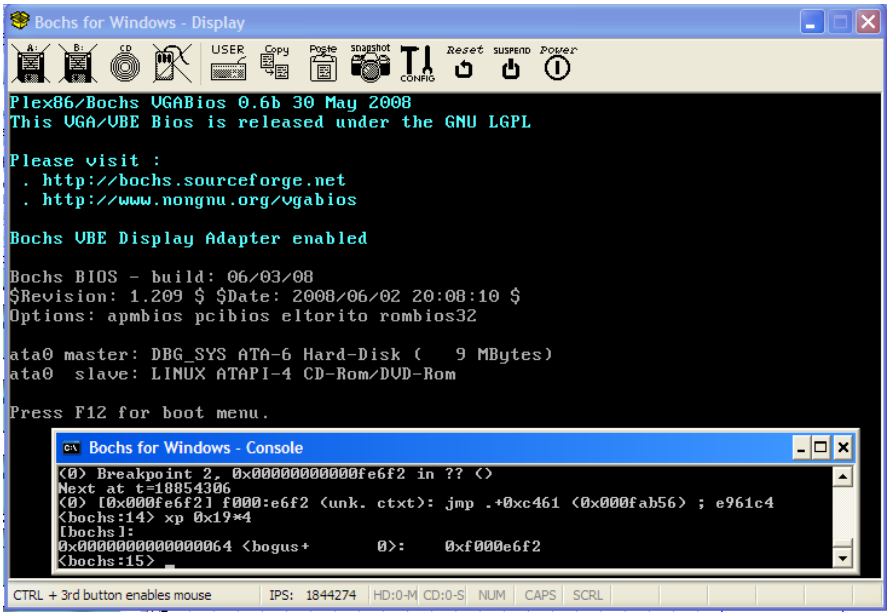


图 6 使用 Bochs 调试引导过程

使用 xp 0x19*4 可以显示中断向量表中 INT 19h 所对应的内容，即 0xf000e6f2，其中高 16 位是段地址，低 16 位是偏移。值得说明的是，大多数 BIOS 中的 INT 19h 的入口地址都与此相同。知道了地址后，就可以使用 pb 0xfe6f2 来设置断点，其中 0xfe6f2 是 0xf000:e6f2 这个实模式地址对应的物理地址，其换算方法是把 0xf000 左移 4 个二进制位（相当于在十六进制数的末尾加一个 0），然后加上偏移。

顺便说一下，在 Bochs 项目中，实现了一个简单的 BIOS，其主要代码都位于 rombios.c 文件，通过下面的链接可以访问到这个文件：http://bochs.sourceforge.net/cgi-bin/lxr/source/bios/rombios.c 想学习 BIOS 的读者，可以仔细读一下这个文件，这是深刻理解 BIOS 的很有效方法。

0x7c00——新的起点

对于大多数时候使用从 BAID 设备引导，BIOS 中的支持函数会从设备（磁盘）的约定位置读取引导扇区，存放到内存中 0x0000:7c00 这个位置，然后把控制权转交过去。转交时会通过 DL 寄存器传递一个参数，这个参数用来指定磁盘号码，00 代表 A 盘，0x80

代表 C 盘。接下来的引导代码在通过 INT 13h 来访问磁盘时，应该使用这个参数来指定要访问的磁盘。

因为从磁盘引导时，BIOS 一定会把控制权移交到 0x7c00 这个地址，所以在调试时可以在这个位置设置断点，开始分析和跟踪。表 2 列出了其它一些固定的 BIOS 入口地址。

表 2　BIOS 兼容入口点

地址	用途
0xf000:e05b	POST 入口点
0xf000:e2c3	不可屏蔽中断（NMI）处理函数入口点
0xf000:e3fe	INT 13h 硬盘服务入口点
0xf000:e401	硬盘参数表
0xf000:e6f2	INT 19h（引导加载服务）入口点
0xf000:e6f5	配置数据表
0xf000:e739	INT 14h 入口点
0xf000:e82e	INT 16h 入口点
0xf000:e987	INT 09h 入口点
0xf000:ec59	INT 13h 软盘服务入口点
0xf000:ef57	INT 0Eh（Diskette Hardware ISR）入口点
0xf000:efc7	软盘控制器参数表
0xf000:efd2	INT 17h（打印机服务）入口点
0xf000:f065	INT 10h（显示服务）入口点
0xf000:f0a4	MDA/CGA 显示参数表 (INT 1Dh)
0xf000:f841	INT 12h（内存大小服务）入口点
0xf000:f84d	INT 11h 入口点
0xf000:f859	INT 15h（系统服务）入口点
0xf000:fa6e	低 128 个字符的图形模式字体
0xf000:fe6e	INT 1Ah（时间服务）入口点
0xf000:fea5	INT 08h（System Timer ISR）入口点
0xf000:fef3	POST 用这个值来初始化中断向量表
0xf000:ff53	只包含 IRET 指令的 dummy 中断处理过程
0xf000:ff54	INT 05h（屏幕打印服务）的入口点
0xf000:fff0	CPU 复位后的执行起点
0xf000:fff5	构建日期，按 MM/DD/YY 格式，共 8 个字符
0xf000:fffe	系统型号

另外，地址 0x0040:0000 开始的 257 个字节是所谓的 BIOS 数据区（BIOS Data Area），简称 BDA，里面按固定格式存放了 BIOS 向后面的引导程序和操作系统移交的信息。

下一期的问题：

一台 PC 系统开机后显示 Windows could not start because of a general computer hardware configuration problem.，对于这样的问题有哪些方法来调试和解决？（注：上期的问题留到下一期给出答案）

步步为营——如何调试操作系统加载阶段的故障

上一期我们介绍了系统固件（BIOS）寻找不同类型的引导设备的方法，描述了固件向引导设备移交执行权的过程。对于从硬盘引导，首先接受控制权的是位于硬盘的 0 面 0 道 0 扇区中的主引导记录（Main Boot Record），简称 MBR。MBR 一共有 512 个字节，起始处为长度不超过 446 字节的代码，然后是 64 个字节长的分区表，最后两个字节固定是 0x55 和 oxAA。MBR 中的代码会在分区表中寻找活动的分区，找到后，它会使用 INT 13h 将活动分区的引导扇区（Boot Sector）加载到内存中，加载成功后，将执行权移交过去。按照惯例，引导扇区也应该被加载到 0x7C00 这个内存位置，所以 MBR 代码通常会先把自己复制到 0x600 开始的 512 个字节，以便给引导扇区腾出位置。也正是因为这个原因，当使用虚拟机或者 ITP 调试时，如果在 0x7C00 处设置断点，那么这个断点通常会命中两次。引导扇区的内容是和操作系统相关的，在安装操作系统时，操作系统的安装程序会设置好引导扇区的内容。引导的职责通常是加载操作系统的加载程序（OS Loader）。OS Loader 得到控制权后，再进一步加载操作系统的内核和其它程序。本期我们就以 Windows Vista 操作系统为例谈一谈 OS Loader 的工作过程以及如何调试这一阶段的问题。

切换工作模式

我们知道，对于 x86 CPU 来说，不管它是否支持 32 位或 64 位，在它复位后都是处于 16 位的实地址模式。在 BIOS 阶段，CPU 可能被切换到保护模式，但是在 BIOS 把控制权移交给主引导记录前，它必须将 CPU 恢复回实模式，这是一直保持下来的传统。对于使用 EFI 固件的系统，固件可以在保护模式下把控制权移交给操作系统的加载程序。但本文仍旧讨论传统的方式。

因为实模式下的每个段最大只有 64K，而且只能直接访问 1MB 的内存，这个空间是无法容纳今天的主流操作系统的核心文件的，所以 OS Loader 首先要做的一件事就是把 CPU 切换到可以访问更大空间的保护模式。

在切换到保护模式前，应该先建立好全局描述符表（GDT）和中断描述符表（IDT）。通常在 OS Loader 阶段不会开启 CPU 的分页机制（Paging），而且描述符表中的每个段的基地址通常都设置为 0，界限设置为 0xFFFFFFFF，这样便可以在程序中自由访问 4GB 的地址空间，而且线性地址的值就等于物理地址的值，这里使用内存空间的方法就是所谓的平坦模式（Flat Model）。以 Windows Vista 操作系统为例，它的引导管理器程序 BootMgr.EXE 内部既有 16 位代码又有 32 位代码，16 位代码先执行，在验证文件的完好后，会切换到保护模式，并把内嵌的 32 位程序映射到 0x400000 开始的内存区，然后把控制权移交给 32 位代码的起始函数 BmMain。此时观察 CR0 寄存器，可以看到代表保护模式的位 0 已经为 1。

```
kd> r cr0
```

```
cr0=00000013
```

但是代表分页机制的位 31 为 0，说明没有启用分页。观察代码段和数据段的段描述符：

```
kd> dg cs
                              P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0020 00000000 ffffffff Code RE Ac 0 Bg Pg P  Nl 00000c9b
kd> dg ds
                              P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0030 00000000 ffffffff Data RW Ac 0 Bg Pg P  Nl 00000c93
```

可见，它们的基地址都是 0，边界都是 0xFFFFFFFF，这正是平坦模式的典型特征。分别使用 dd 命令和!dd（观察物理地址）观察同一个地址值：

```
kd> dd idtr l4
0001f080  00500390 00008f00 002073b0 00448e00
kd> !dd idtr l4
#   1f080 00500390 00008f00 002073b0 00448e00
```

显示的内容是一样的，这说明线性地址与它所对应的物理地址的值是相等的。

休眠（Hibernation）支持

在执行 BlImgQueryCodeIntegrityBootOptions 函数和 BmFwVerifySelfIntegrity 函数对自身的完整性做进一步检查后，BootMgr 会调用 BmResumeFromHibernate 检查是否需要从休眠（Hibernation）中恢复，如果需要，那么它会加载 WinResume.exe，并把控制权移交给它。

显示启动菜单

BootMgr 会从系统的引导配置数据（Boot Configuration Data，简称 BCD）中读取启动设置信息，如果有多个启动选项，那么它会显示出启动菜单。清单 1 中的栈回溯显示的便是 BootMgr 在显示启动菜单后等待用户选择时的状态。

清单 1 等待用户选择启动项

```
kd> kn
 # ChildEBP RetAddr
00 00061e34 00432655 bootmgr!DbgBreakPoint
01 00061e44 00431c24 bootmgr!BlXmlConsole::getInput+0xe
02 00061e90 00402e8f bootmgr!OsxmlBrowser::browse+0xe0
03 00061e98 00402b5e bootmgr!BmDisplayGetBootMenuStatus+0x13
04 00061f10 004017ce bootmgr!BmDisplayBootMenu+0x174
05 00061f6c 00401278 bootmgr!BmpGetSelectedBootEntry+0xf8
06 00061ff0 00020a9a bootmgr!BmMain+0x278
WARNING: Frame IP not in any known module. Following frames may be wrong.
07 00000000 f000ff53 0x20a9a
08 00000000 00000000 0xf000ff53
```

栈帧 6 中的 BmMain 便是 BootMgr 的 32 位代码的入口函数，栈帧 4 中的 BmDisplayBootMenu 是显示启动菜单的函数，栈帧 7 和 8 是在实模式中执行时的痕迹。

执行用户选择的启动项

当用户选择一个启动选项后，BootMgr 会调用 函数来准备引导对应的操作系统。如果系统上有 Windows XP 或者更老的 Windows，而且用户选择了这些项，那么 BootMgr 会加载 NTLDR 来启动它们。如果用户选择的是 Windows Vista 的启动项，那么 BootMgr 会寻找和加载 WinLoad.exe，如果没有找到或者在检查文件的完整性时发现问题，那么 BootMgr 会显示出图 1 所示的错误界面。

在成功加载 WinLoad.exe 后，BootMgr 会为其做一系列其它准备，包括启用新的 GDT 和 IDT，然后调用平台相关的控制权移交函数把执行权移交给 WinLoad。在 x86 平台中，完成这一任务的是 Archx86TransferTo32BitApplicationAsm 函数。至此，BootMgr 完成使命，WinLoad 开始工作。

加载系统核心文件

WinLoad 的主要任务是把操作系统内核加载到内存，并为它做好“登基”的准备。它首先要做的一件事就是进一步改善运行环境，启用 CPU 的分页机制。然后初始化自己的支持库，如果启用了引导调试支持（稍后介绍），那么它会初始化调试引擎。

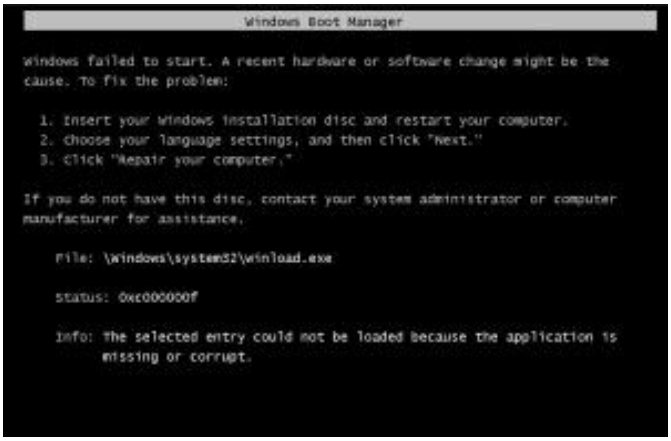


图 1 加载 WinLoad.exe 失败时的错误提示

接下来 WinLoad 会读取启动参数，决定是否显示高级启动菜单，高级菜单中含有以安全模式启动等选项，也叫 Windows Error Recovery 菜单。如果用户按了 F8 或者上次没有正常关机，那么 WinLoad 便会显示高级启动菜单。

接下来要做的一个重要工作是读取和加载注册表的 System Hive，因为其中包含了更多的系统运行参数，负责这项工作的是 OslpLoadSystemHive 函数。

做好以上工作后，WinLoad 开始它的核心任务，那就是加载操作系统的内核文件和引导类型的设备驱动程序。它首先加载的是 NTOSKRNL.EXE，这个文件包含了 Windows 操作系统的内核和执行体。此时真正的磁盘和文件系统驱动程序还没有加载进来，所以 WinLoad 是使用它自己的文件访问函数来读取文件的。例如，FileIoOpen 函数便是用来打开一个文件的，

如果 FileIoOpen 打开文件失败，那么调用它的 BlpFileOpen 函数会返回错误码 0C000000Dh，否则返回 0 代表成功。

其中，PSHED.DLL 用于支持 WHEA（Windows Hardware Error Architecture）（《软件调试》第 17 章有详细介绍），HAL.DLL 是硬件抽象层模块，BOOTVID.DLL 用于引导期间和发生蓝屏时的显示，KDCOM.DLL 用于支持内核调试，CLFS.SYS 是支持日志的内核模块，CI.DLL 是用于检查模块的完整性的（Code Integrity）。

加载好系统模块后，WinLoad 还需要加载引导类型（Boot Type）的设备驱动程序，在安装驱动程序时，每个驱动程序都会指定启动类型（Start Type），这个设置决定了驱动程序的加载时机，指定为引导类型的驱动程序是最先被加载的。

接下来加载的是硬件抽象层模块 HAL.DLL，支持调试的 KDCOM.DLL 和它们的依赖模块。使用 Depends 工具可以观察一个 PE 模块所依赖的其它模块，例如，图 2 显示出了内核文件 NTOSKRNL.EXE 所依赖的其它模块。

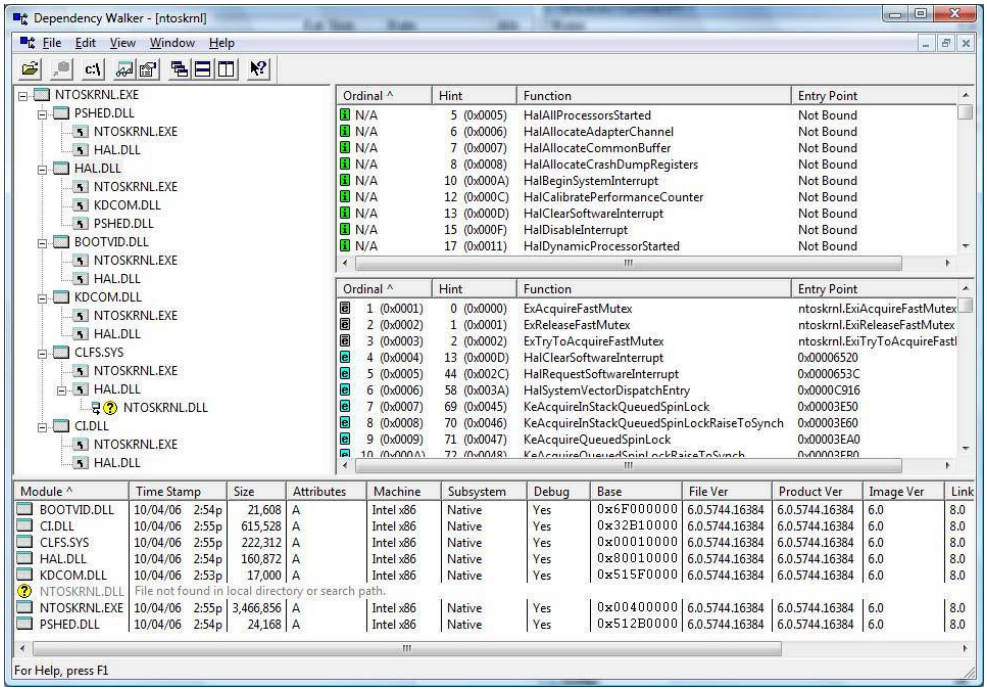


图 2 使用 DEPENDS 工具观察 NTOSKRNL.EXE 所依赖的其它模块

如果在加载以上程序模块或者注册表的过程中找不到需要的文件或者在检查文件的完整性时发现异常，那么 WinLoad 便会提示错误而停止继续加载，我们在 08 年第 11 期中提到的问题便是与此有关的。当遇到这样的问题时，可以使用安装光盘引导，然后恢复丢失或者被破坏的文件。

完成模块加载后，WinLoad 开始准备把执行权移交给内核，包括为内核准备新的 GDT 和 IDT（OslArchpKernelSetupPhase0）和建立内存映射（OslBuildKernelMemoryMap）等。所有准备工作做完后，WinLoad 调用 OslArchTransferToKernel 函数把供内核使用的 GDT 和 IDT 地址加载到 CPU 中，然后调用内核的入口函数，正式把控制权移交个内核。

启用调试选项

Windows Vista 的 BootMgr 和 WinLoad 程序内部都集成了调试引擎，不管是 Checked 版本还是 Free 版本，对于 Free 版本，默认是禁止的，使用时需要开启，具体做法如下：

如果要启用 BootMgr 中的调试引擎，那么应该在一个具有管理员权限的控制台窗口中

执行如下命令：

```
bcdedit /set {bootmgr} bootdebug on
bcdedit /set {bootmgr} debugtype serial
bcdedit /set {bootmgr} debugport 1
bcdedit /set {bootmgr} baudrate 115200
```

以上命令是使用串行口作为主机和目标机之间的通信方式，如果使用其它方式，那么应该设置对应的参数。

如果要启用 WinLoad 程序中的调试引擎，那么应该先找到它所对应的引导项的 GUID 值，然后执行如下命令：

```
bcdedit /set {GUID} bootdebug on
```

启用调试引擎并连接通信电缆后，在主机端运行 WinDBG 工具，便可以进行调试了，栈回溯、访问内存、访问寄存器等内核调试命令都可以像普通内核调试一样使用。

Windows Vista 之前的情况

在 Vista 之前，NTLDR 是 Windows 操作系统的加载程序。因为只有 Checked 版本的 NTLDR 才支持调试，所以如果要调试加载阶段的问题，应该先将 NTLDR 替换为 Checked 版本。DDK 中通常包含有 Checked 版本的 NTLDR 程序。记住，在替换前，应该先去除 NTLDR 文件的系统、隐藏和只读属性，在更换后，要加上这些属性，否则的话引导扇区中的代码会报告 NTLDR is missing 错误，无法继续启动。

除了加载内核和引导类型的驱动程序外，NTLDR 会调用 NTDETECT.COM 来做基本的硬件检查并搜集硬件信息。NTDETECT 会把搜集到的信息存放到注册表中。如果找不到 NTDETECT.COM，那么通常会直接重启，如果 NTDETECT 发现系统缺少必须的硬件或固件支持，比如 ACPI 支持，那么会显示因为硬件配置问题而无法启动，也就是我们上一期所提问的问题。对于这样的问题，可以尝试更改 BIOS 选项来解决，或者通过调试 NTLDR 来进一步定位错误原因。

恢复缺失文件

可以使用如下方法之一来尝试恢复丢失或者损坏的系统文件：

15. 启动时按 F8，调出高级启动菜单，尝试选择 Last Known Good Configuration（LKG）。
16. 启动时按 F8，在高级启动菜单中选择安全模式（Safe Mode），如果成功启动后，那么可以尝试执行 CHKDSK 命令检查和修复磁盘，或者从安装光盘中恢复缺失的文件。
17. 使用 Windows 安装光盘引导，并记入到恢复控制台（Recovery Console）界面。对于 Windows XP，在安装程序的主界面中按 R 键进入文本界面的恢复控制台，进入时输入管理员密码。对于 Windows Vista，从安装光盘启动后，可以进入图形界面的系统恢复向导（图 3）。如果是 MBR 或者引导分区损坏，那么 Windows XP 的恢复控制台中提供了 FIXMBR 和 FIXBOOT 命令。而 Vista 的恢复向导中包含了自动修复功能。

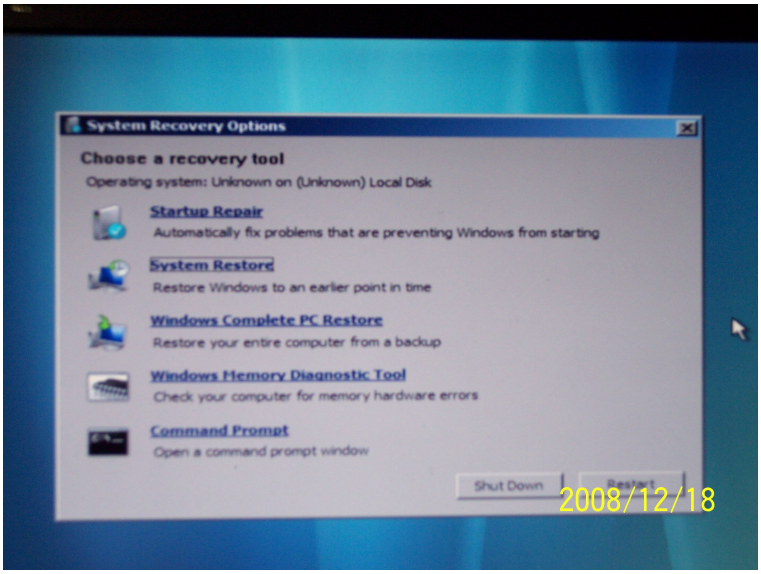


图 3 Windows Vista 安装光盘上的系统恢复程序

18. 如果系统硬盘的个数或者有所变化，那么可能是因为分区编号变化而导致系统无法找到文件，这时可以考虑恢复旧的磁盘和分区配置，或者启动到恢复控制台来修改系统的启动配置文件，对于 Vista，需要修改 BCD，对于 Vista 之前的系统，也就是修改 BOOT.INI 文件。

对于第 11 期的问题，天津的黄小非读者给出了非常好的答案，他的来信中给出了多种方法，包括使用控制台，使用 Windows Preinstallation Environment（WinPE）以及修改 BOOT.INI。其实 Vista 的恢复界面就是运行在 WinPE 中的。从黄小非的来信中，我们可以看出他的实践经验很丰富。

下一期的问题：

系统启动后很快出现蓝屏，其中含有 STOP 0x0000007B INACCESSABLE_BOOT_DEVICE，哪些原因会导致这样的问题，该如何来解决？

百废待兴——如何调试内核初始化阶段的故障

上一期我们介绍了加载操作系统的过程。简单来说，负责加载操作系统的加载程序（OS Loader）会把系统内核模块、内核模块的依赖模块、以及引导类型的驱动程序加载到内存中，并为内核开始执行准备好基本的执行环境。这些工作做好后，加载程序会把执行权移交给内核模块的入口函数，于是操作系统的内核模块就开始执行了。在今天的软件架构中，操作系统承担着统一管理系统软硬件资源的任务，可以说是整个系统的统帅。内核模块是操作系统的核心部分，像任务调度、中断处理、输入输出等核心功能就是实现在内核模块中的。因此，内核模块开始执行，标志着“漫长的”启动过程进入到了一个新的阶段，系统的统帅走马上任了。虽然前面已经做了很多准备工作，但是对于一个典型的多任务操作系统来说，要建设出一个可以运行各种应用程序的多任务环境来，还有很多事情要做，可谓是百废待兴。本期我们仍以 Windows 操作系统为例谈一谈系统内核和执行体初始化（简称内核初始化）的过程以及如何调试这一阶段的问题。

入口函数

Windows 程序的入口函数地址是登记在可执行文件的头结构中的，也就是 IMAGE_OPTIONAL_HEADER 结构的 AddressOfEntryPoint 字段。内核文件的入口函数也是如此。通过下面几个步骤就可以使用 WinDBG 观察到内核文件的入口函数。先启动 WinDBG，并开始一个本地内核调试对话，使用 lm nt 命令列出内核文件的基本信息：

```
lkd> lm a nt
start    end      module name
804d7000 806cdc80 nt (pdb symbols) d:\symbols\ntkrnlpa.pdb\C…\ntkrnlpa.pdb
```

其中 804d7000 就是内核模块在内存中的起始地址。起始处是一个所谓的 DOS 头：

```
dt nt!_IMAGE_DOS_HEADER 804d7000
   +0x000 e_magic         : 0x5a4d
…
   +0x03c e_lfanew        : 232
```

其中 e_lfanew 字段的值代表的是新的 NT 类型可执行文件的头结构的起始偏移地址。

```
lkd> dt nt!_IMAGE_NT_HEADERS 804d7000+0n232
   +0x000 Signature       : 0x4550
   +0x004 FileHeader      : _IMAGE_FILE_HEADER
   +0x018 OptionalHeader  : _IMAGE_OPTIONAL_HEADER
```

现在可以知道 804d7000+0n232+18 处便是_IMAGE_OPTIONAL_HEADER 结构，于是可以使用下面的命令来显示出 AddressOfEntryPoint 字段的值：

```
lkd> dt _IMAGE_OPTIONAL_HEADER -y Add* 804d7000+0n232+18
nt!_IMAGE_OPTIONAL_HEADER
   +0x010 AddressOfEntryPoint : 0x1b6f5c
```

上面显示的 AddressOfEntryPoint 字段的值 0x1b6f5c 便代表着内核文件的入口函数在模块中的偏移，加上模块的基地址便可以得到入口函数的线性地址，使用 ln 命令查找这个地址对应的符号：

```
lkd> ln 0x1b6f5c+804d7000
(8068df5c)   nt!KiSystemStartup   |  (8068e244)   nt!KiSetCR0Bits
Exact matches:
   nt!KiSystemStartup = <no type information>
```

这表明入口地址处的函数名为 KiSystemStartup，实际上，它就是 NT 系统 Windows 操作系统的内核文件一直使用的入口函数。

上面我们介绍的是使用类型显示命令一步步观察，当然也可以使用扩展命令!dh 一下子显示出以上信息：

```
lkd> !dh 804d7000 -f
File Type: EXECUTABLE IMAGE
…
  1B6F5C address of entry point
```

当 OS Loader（NTLDR 或 WinLoad）调用 KiSystemStartup 时，它会将启动选项以一个名为 LOADER_PARAMETER_BLOCK 的数据结构传递给 KiSystemStartup 函数。Windows Vista 的内核符号文件包含了这个结构的符号，因此在对 Windows Vista 做内核调试时可以观察到这个结构的详细定义。

内核初始化

KiSystemStartup 函数开始执行后，它首先会进一步完善基本的执行环境，包括建立和初始化处理器控制结构（PCR）、建立任务状态段（TSS）、设置用户调用内核服务的 MSR 寄存器等。在这些基本的准备工作完成后，接下来的过程可以分为图 1 所示的左右两个部分。左侧为发生在初始的启动进程中的过程，这个初始的进程就是启动后的 Idle 进程。右侧为发生在系统进程（System）中的所谓的执行体阶段 1 初始化过程。

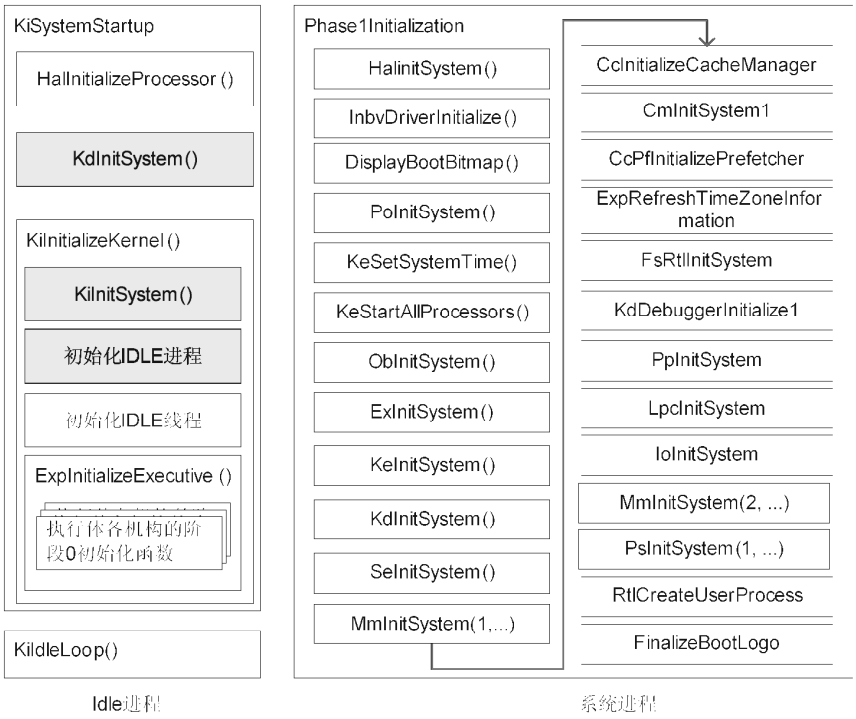


图 1 Windows 启动过程概览

首先我们来看 KiSystemStartup 函数的执行过程，它所做的主要工作有：

一、调用 HalInitializeProcessor()初始化 CPU。

二、调用 KdInitSystem 初始化内核调试引擎，我们稍后将详细介绍这个函数。

三、调用 KiInitializeKernel 开始内核初始化，这个函数会调用 KiInitSystem 来初始化系统的全局数据结构，调用 KeInitializeProcess 创建并初始化 Idle 进程，调用 KeInitializeThread 初始化 Idle 线程。

对于多 CPU 的系统，每个 CPU 都会执行 KiInitializeKernel 函数，但只有第一个 CPU 会执行其中的所有初始化工作，包括全局性的初始化，其它 CPU 会只执行 CPU 相关的部分。比如只有 0 号 CPU 会调用和执行 KiInitSystem，初始化 Idle 进程的工作也只有 0 号 CPU 执行，因为只需要一个 Idle 进程，但是因为每个 CPU 都需要一个 Idle 线程，所以每个 CPU 都会执行初始化 Idle 线程的代码。KiInitializeKernel 函数使用参数来了解当前的 CPU 号。全局变量 KeNumberProcessors 标志着系统中的 CPU 个数，其初始值为 0，因此当 0 号 CPU 执行 KiSystemStartup 函数时，KeNumberProcessors 的值刚好是当前的 CPU 号。当第二个 CPU 开始运行时，这个全局变量会被递增 1，因此 KiSystemStartup 函数仍然可以从这个全局变量了解到 CPU 号，依此类推，直到所有 CPU 都开始运行。ExpInitializeExecutive 函数的第一个参数也是 CPU 号，在这个函数中也有很多代码是根据 CPU 号来决定是否执行的。

执行体的阶段 0 初始化

在 KiInitializeKernel 函数结束基本的内核初始化后，它会调用 ExpInitializeExecutive() 开始初始化执行体。如果把操作系统看作是一个国家机器，那么执行体便是这个国家的各个行政机构。典型的执行体部件有进程管理器、对象管理器、内存管理器、IO 管理器等等。考虑到各个执行体之间可能有相互依赖关系，所以每个执行体会有两次初始化的机会，第一次通常是做基本的初始化，第二次做可能依赖其它执行体的动作。通常前者叫阶段 0 初始化，后者叫阶段 1 初始化。

ExpInitializeExecutive 的主要任务是依次调用各个执行体的阶段 0 初始化函数，包括调用 MmInitSystem 构建页表和内存管理器的基本数据结构，调用 ObInitSystem 建立名称空间，调用 SeInitSystem 初始化 token 对象，调用 PsInitSystem 对进程管理器做阶段 0 初始化（稍后详细说明），调用 PpInitSystem 让即插即用管理器初始化设备链表。

下面我们仔细看一下进程管理器的阶段 0 初始化，它所做的主要动作有：

- 定义进程和线程对象类型。
- 建立记录系统中所有进程的链表结构，并使用 PsActiveProcessHead 全局变量指向这个链表。此后 WinDBG 的!process 命令才能工作。
- 为初始的进程创建一个进程对象（PsIdleProcess），并命名为 Idle。
- 创建系统进程和线程，并将 Phase1Initialization 函数作为线程的起始地址。

注意上面的最后一步，因为它衔接着系统启动的下一个阶段，即执行体的阶段 1 初始化。但是这里并没有直接调用阶段 1 的初始化函数，而是将它作为新创建系统线程的入口函数。此时由于当前的 IRQL 很高，所以这个线程还得不到执行。在 KiInitializeKernel 函数返回后，KiSystemStartup 函数将当前 CPU 的中断请求级别（IRQL）降低到 DISPATCH_LEVEL，然后跳转到 KiIdleLoop()，退化为 Idle 进程中的第一个 Idle 线程。当再有时钟中断发生时，内核调度线程时，便会调度执行刚刚创建的系统线程，于是阶段 1 初始化便可以继续了。

执行体的阶段 1 初始化

阶段 1 初始化占据了系统启动的大多数时间，其主要任务就是调用执行体各机构的阶段 1 初始化函数。有些执行体部件使用同一个函数作为阶段 0 和阶段 1 初始化函数，使用参数来区分。图 1 列出了这一阶段所调用的主要函数，简要说明其中几个：

- 调用 KeStartAllProcessors()初始化所有 CPU。这个函数会构建并初始化好一个处理器状态结构，然后调用硬件抽象层的 HalStartNextProcessor 函数将这个结构赋给一个新的 CPU。新的 CPU 仍然是从 KiSystemStartup 开始执行。
- 再次调用 KdInitSystem 函数，并且调用 KdDebuggerInitialize1 来初始化内核调试通信扩展 DLL（KDCOM.DLL 等）。
- 调用 IO 管理器的阶段 1 初始化函数 IoInitSystem 做设备枚举和驱动加载工作，需要花很长的时间。

在这一阶段结束前，会创建第一个使用映像文件创建的进程，即会话管理器进程（SMSS.EXE）。会话管理器进程会初始化 Windows 子系统，创建 Windows 子系统进程和登录进程（WinLogon.EXE），我们以后再介绍。

0x7B 蓝屏

上面介绍的过程不总是一帆风顺的。如果遇到意外，那么系统通常会以蓝屏形式报告错误。比如图 2 所示的 0x7B 蓝屏就是发生在内核和执行体初始化期间的（我们上一期的问题）。

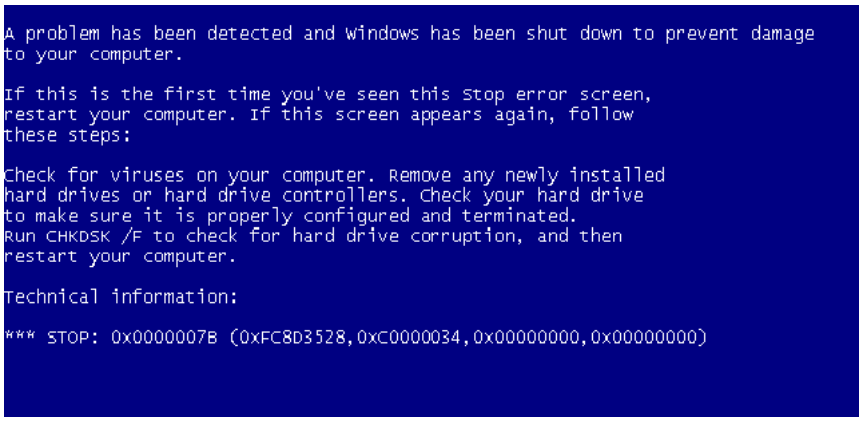


图 2　0x7B 蓝屏

注意这个蓝屏的下方没有转储有关的信息（稍后你就会明白原因了）。

那么应该如何寻找这个蓝屏的故障原因呢？

首先可以根据蓝屏的停止代码 0x7B 查阅 WinDBG 的帮助文件或者 MSDN 了解它的含义。于是我们知道，0x7B 是 INACCESSIBLE_BOOT_DEVICE 错误的代码，其含义是不可访问的引导设备。意思是系统在读或者写引导设备时出错了，进一步来说，也就是在访问包含有系统文件的磁盘分区时出问题了。

访问系统分区怎么会出问题呢？操作系统加载程序刚刚不是还读过系统分区来加载系统文件了的，现在怎么不能访问了呢？磁盘设备在这两个时间点之间损坏的概率很低，

因此，主要的原因还是因为访问的方式不同了。操作系统加载程序是使用简单的方式来访问磁盘的，而操作系统内核开始运行后，开始改用更为强大的驱动程序来访问磁盘，而这里恰恰是常出问题的地方。对于典型的 IDE 硬盘，需要使用 ATAPI.SYS 这个驱动程序来进行访问。那么 ATAPI 这个驱动是谁来加载的呢？让内核自己来加载，肯定不行，因为内核是依赖它来访问磁盘的，正所谓“自己的刀刃削不了自己的刀把”。那么应该由谁来加载呢？OS Loader，也就是 NTLDR 或者 WinLoad。它们怎么知道要加载这个驱动呢？是根据注册表。图 2 显示了注册表中 ATAPI 驱动程序的各个键值。其中的 Start 键值等于 0 代表是引导类型，Group 键值标志着这个驱动属于 SCSI miniport 这个组。OS Loader 看到 Start 键值为 0 后，就会将这个驱动程序加载到内存中。我们不妨把以这种方式加载的驱动程序称为第一批加载的驱动程序。

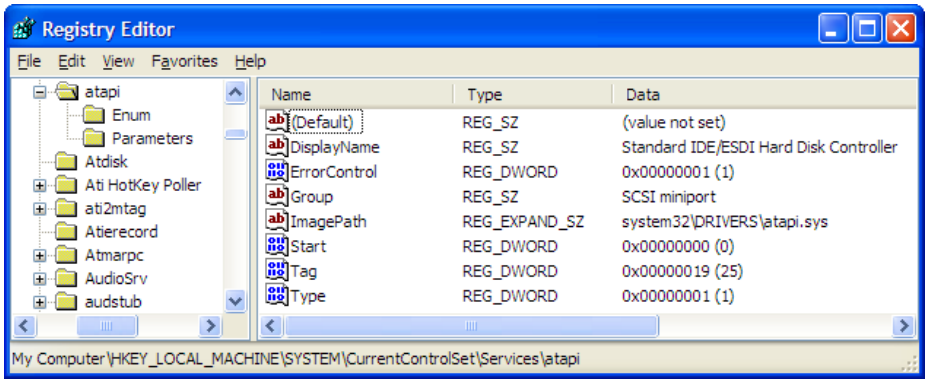


图 3　ATAPI 驱动程序的注册表键值

如果按 F8 通过高级选项菜单中的某一项启动，那么 NTLDR 会显示出它加载的第一批驱动程序的清单（图 4）。

在上面的清单中，没有 ATAPI.SYS，这正是问题所在。事实上笔者就是将 Start 值改为 3 来“制造”出这个蓝屏的（读者一定不要草率模仿，以免丢失数据）。

```
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\system32\c_437.nls
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\system32\l_intl.nls
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\FONTS\vgaoem.fon
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\AppPatch\drvmain.sdb
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\system32\DRIVERS\ACPI.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\system32\DRIVERS\WMILIB.SYS
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\system32\DRIVERS\pci.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\system32\DRIVERS\isapnp.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\system32\DRIVERS\intelide.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\system32\DRIVERS\PCIIDEX.SYS
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\System32\Drivers\MountMgr.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\system32\DRIVERS\ftdisk.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\System32\drivers\dmload.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\System32\drivers\dmio.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\System32\Drivers\PartMgr.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\System32\Drivers\VolSnap.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\system32\DRIVERS\disk.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\system32\DRIVERS\CLASSPNP.SYS
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\system32\DRIVERS\fltMgr.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\system32\DRIVERS\sr.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\System32\Drivers\KSecDD.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\System32\Drivers\Ntfs.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\System32\Drivers\NDIS.sys
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS\System32\Drivers\Mup.sys
```

图 4　第一批加载的驱动程序清单

除了观察访问磁盘的关键驱动程序是否加载，还可以使用内核调试来做进一步的分析。如果目标系统事先没有启用内核调试，那么可以在引导初期按 F8 调出高级引导菜单，

然后选择 Debug。这时系统通常会使用串行口 2（COM2）以波特率 19200 来启用内核调试引擎（参见《软件调试》18.3.3 P478）。然后使用一根串口通信电缆将目标机器与调试主机相连接（主机不一定要使用 COM2）。

成功建立调试会话后，在出现蓝屏前，调试器便会收到通知：

```
*** Fatal System Error: 0x0000007b
                       (0xFC8D3528,0xC0000034,0x00000000,0x00000000)
```

此时观察栈回溯，便可以看到发生蓝屏的过程：

```
kd> kn
 # ChildEBP RetAddr
00 fc8d3090 805378e7 nt!RtlpBreakWithStatusInstruction
01 fc8d30dc 805383be nt!KiBugCheckDebugBreak+0x19
02 fc8d34bc 805389ae nt!KeBugCheck2+0x574
03 fc8d34dc 806bdc94 nt!KeBugCheckEx+0x1b
04 fc8d3644 806ae093 nt!IopMarkBootPartition+0x113
05 fc8d3694 806a4714 nt!IopInitializeBootDrivers+0x4ba
06 fc8d383c 806a5ab0 nt!IoInitSystem+0x712
07 fc8d3dac 80582fed nt!Phase1Initialization+0x9b5
08 fc8d3ddc 804ff477 nt!PspSystemThreadStartup+0x34
09 00000000 00000000 nt!KiThreadStartup+0x16
```

这个栈回溯表明这个系统线程正在做执行体的阶段 1 初始化。目前在执行的是 IO 管理器的 IoInitSystem 函数。后者又调用 IopInitializeBootDrivers 来初始化第一批加载的驱动程序。IopInitializeBootDrivers 又调用 IopMarkBootPartition 来把引导设备标识上引导标记。在做标记前，IopMarkBootPartition 需要打开引导设备，获得它的对象指针。但是打开这个设备时失败了，于是 IopMarkBootPartition 调用 KeBugCheckEx 发起蓝屏，报告错误。

蓝屏停止码的第一个参数是引导设备的路径，使用 dS 命令可以显示其内容：

```
kd> dS fc8d3528
e13fa810  "\ArcName\multi(0)disk(0)rdisk(0)"
e13fa850  "partition(1)"
```

蓝屏停止码的第二个参数是 IopMarkBootPartition 调用 ZwOpenFile 打开引导设备失败的返回值。使用!error 命令可以显示其含义：

```
kd> !error 0xC0000034
Error code: (NTSTATUS) 0xc0000034 (3221225524) - Object Name not found.
```

也就是没有这样的设备对象存在，无法打开，这是因为没有加载 ATAPI 驱动。

观察系统中的进程列表，可以看到系统中目前只有 System 进程和 IDLE 进程。

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 812917f8  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00039000  ObjectTable: e1000b98  HandleCount:  34.
    Image: System
```

使用 lm 观察模块列表，可以看到与图 4 中一致的结果。也就是说，目前系统中还没有加载普通的驱动程序，必须等到引导类型的驱动程序初始化结束后，也就是访问磁盘和文件系统的第一批驱动程序准备好了后，才可能加载其它驱动程序。

对于上面分析的例子，原因是由于注册表异常而没有加载必要的 ATAPI.SYS。知道了原因后，对于 Windows Vista 可以使用我们上一期介绍的用安装光盘引导到恢复控制台，然后将注册表中的 Start 键值改回到 0 系统便恢复正常了。对于 Windows XP，可以借助 ERD Commander 等工具来引导和修复。

在上一期的读者来信中，天津的黄小非先生给出了很全面的分析，把导致问题的可能原因归纳为病毒破坏、驱动程序故障和硬件故障三种情况，归纳的很好。关于如何定位原因，他提到了使用转储文件（DUMP），也是有帮助的。但因为默认的小型转储文件包含的信息有限，所以我们在上文中重点介绍了利用双机内核调试来跟踪和分析活动的目标。

因为建立内核调试会话的详细步骤很容易找到，所以我们没有详细描述，感觉有困难的朋友可以参考 WinDBG 帮助文件中 Kernel-Mode Setup 一节，有《软件调试》一书的朋友可以看第 18 章的前三节。黄小非在来信中还对我们以后要讨论的内容提出了很好的建议，我们会认真考虑这些建议，在此深表感谢。

下一期的问题：

一台装有 Windows 的系统输入用户名和密码后桌面一闪便自动 Log Off 了，再尝试登录，现象一样，始终无法进入到正常的桌面状态，哪些原因会导致这样的问题，该如何来解决？