UNDOCUMENTED WINDOWS 2000 SECRETS

A PROGRAMMERS COOKBOOK

SVEN B.SCHREIBER

翻译：Kendiv（fcczj@263.net）

译者Blog：http://blog.csdn.net/Kendiv/

整理：F7

目　录

一、Windows 2000 对调试技术的支持

尽管本书中的很多内容都称之为“Undocumented”，但其中的一些内容只能通过挖掘操作系统的代码才能获取。Windows 2000 DDK（Device Driver Kit）提供了一个强大的调试器可以出色的完成这方面的工作。本章将从建立一个完善的调试环境开始介绍。在阅读随后的章节时，你会经常的使用内核调试器来挖掘操作系统内部的各种特性。如果你对内核调试器很是厌烦，或许你需要制作一个自己的调试工具了。因此，本章还将介绍有关 Windows 2000 调试接口的文档化和未文档化的资料，包括微软符号文件（Symbol File）的详细信息。It features two sample libraries with companion applications that list processes, process and system modules, and various kinds of symbol information buried inside the Windows 2000 symbol files。做为一个特殊收获，在本章结束时，你将得到首份有关 PDB（Microsoft Program Database）的公开文档。

1.1、建立一个调试环境

“嗨，我不想调试 Windows 2000 程序。在此之前，我想自己写一个先！”当你读到这个标题时你可能会这样大喊出来。“很对！”我说“这就是你该去做的！”但是为什么你要以建立一个调试环境开始这次旅行呢？答案很简单：调试器是进入系统的后门。当然，这并不是调试器开发人员的主要目的。然而，当你跟踪代码的执行过程或者你的程序意外的玩完时，任何优秀的调试器都须能够告诉你一些有用的系统信息。仅仅报告一个指向 4GB 地址空间某处的 8 位崩溃地址，然后让你独自一人去寻找到底发生了什么，真是无法让人接受。调试器至少应该告诉你最后执行的引发错误的代码是哪个模块中的代码，而且，在理想情况下，它还应该告诉你让你的程序玩完的那个函数的名称。因此，调试器通常必须知道比编程手册还要多的系统信息，你可以利用这些信息来研究系统的内部情况。

Windows 2000 提供了两个调试器：WinDbg.exe（发音很像“WindBag”，译注：WindBag 在俚语中指空话连篇的人）一个 Win32 GUI 程序和 i386kd.exe 一个提供与之等价功能的命令行模式程序。我曾经同时使用过这两个程序，最后确定 i386kd.exe 是最好的一个，因为它有一组非常强大的选项。不过，最近看来 WinDbg.exe 似乎有所改进。不过，本书中的所有例子都是与 i386kd.exe 相关的。就像你猜想的那样，i386 前缀表示目标平台（Intel 386

处理器家族，也包括 Pentium）kd 是 Kernel Debugger（内核调试器）的缩写。Windows 2000 内核调试器是一个非常强大的工具。比如，他知道如何使用 Windows 2000 安装光盘中的符号文件（Symbol files），因此，可以给出系统内存中几乎任何地址的相关符号信息（这非常有价值）。而且，它还可以反编译二进制代码、将内存信息的 16 进制转储数据以多种格式显示，甚至还能显示一些内核关键结构的布局。在调试器的在线帮助中有其命令行接口的详细文档。

1.1.1、准备一次崩溃转储（Crash Dump）

这些都是好消息。坏消息是你在内核调试器顺从你之前，必须做一些准备工作。第一个障碍是调试通常涉及两台独立的计算机（通过线路连接在一起），其中一台运行调试器，另一台用于被调试。然而，如果并不需要实时调试，那么有一个简单的方法，可以不需要第二台机器。例如，如果一个有错误的程序抛出了一个未处理的异常而引发了声名狼藉的 NT 蓝屏死机（Blue Screen Of Death, BSOD），你可以选择保存崩溃前的内存映像到一个文件中，在重新启动后，检查这个崩溃转储（Crash Dump）文件。这项技术通常被叫做 post mortem（事后检查）在拉丁文中，post mortem 意思是“after death”。这种方式是本书首选方法之一。在这里，我们的主要任务是研究系统内存，在大多数情况下，内存数据是来自还在工作的系统或者来自系统崩溃前内存的一个快照（snapshot）都并不重要。然而，一些有趣的信息则需要通过内核模式的驱动程序深入正在工作的系统的内部才能观察到，这一主题被保留在后面的章节中。

一个崩溃转储（Crash Dump）只是简单将当前内存数据写入一个磁盘文件而已。因此，一个完整的崩溃转储（crash dump）文件的大小通常与系统物理内存一样大（事实上，会略微小些）。崩溃转储（Crash dump）是由内核中的一个特殊程序在处理致命错误过程中生成的。然而，这个例程（handler）并不是立即将内存数据写入目标文件中。这是个不错的处理方式，因为在系统崩溃后，磁盘文件系统可能也不能正常工作。因此，内存映像首先被复制到页面文件存储器（page file storage），这是系统内存管理器的一部分。因此，你应该将你的页面文件大小增加到至少两倍于物理内存。两倍？一样大还不够吗？当然—那只够存放崩溃转储（crash dump）。要知道，在启动时，系统会尝试将崩溃转储（crash dump）映像复制到实际的磁盘文件，这意味着，如果系统不能及时的释放映像数据占用的页面文件，它就可能用尽所有的虚拟内存。通常，系统会处理这种情况，它会疯狂的读写磁盘并向你抛

出一个惹人厌的“虚拟内存不足”的警告。只要你预料蓝屏的概率会增大时，将页面文件设置的足够大，这将会为你节省很多时间。

到这儿，你应该打开 Windows 2000 的控制面板，改变如下的设置：

- 增加页面文件到至少两倍物理内存的大小。
- 接下来，配置系统以便当蓝屏发生时生成一个崩溃转储（crash dump）文件。在系统属性对话框里，选择高级页，然后单击启动和恢复按钮，检查写入调试信息选项。你应该在下拉列表中选择完成内存转储选项。在转储文件对话框中输入一个文件名和路径，转储文件将会从页面文件中复制到你指定的这个文件中。

  %SystemRoot%\MEMORY.DMP 是默认设置。

1.1.2、 让系统崩溃

当设置好系统准备一次 crash dump 后，是时候做在 Windows 2000 系统程序员一生中最厌恶的事了：开始让系统崩溃！通常，只要达摩克利斯的剑挂在了你的头顶上（通常是在离产品截止时间还有几个小时的时候）你就会看到恐怖的蓝屏。现在，是你自愿让系统崩溃，但你可能无法找到有问题的软件来完成这项“工作”。来试试 David Solomon 在他的《Inside Windows NT 第二版》中提到的那个优雅的诀窍：

“如何能可靠的产生一个崩溃转储（crash dump）文件？只需要使用 Windows NT 资源工具中的 kill.exe 工具，kill 掉 Win32 子系统进程（csrss.exe）或者 Windows NT 登陆进程（winlogon.exe），你必须有管理员权限”（Solomon [1998],p.23.）

神奇，太神奇了！这个窍门不能在 Windows 2000 上工作！第一感觉，很不走运，但是从另一个角度看，这是个好消息。当你知道使用微软自己正式发布的一个小工具就能如此轻松的破坏系统，你会怎样想？事实上，微软关闭这个安全漏洞非常对。可是，我们现在需要一种方法来使系统崩溃啊。在这一点上，想想那个古老而简单的 NT 规则：“If anything seems to be impossible in the Win32 word, just write a kernel-mode driver, and it will work out all right!”Windows 2000 非常谨慎的管理 Win32 程序。它在应用程序和内核之间构建了一堵墙，任何企图跨越此边界者都会被毫不留情的解决掉。这对于系统的稳定性是个好消息，但是对于编写需要直接与硬件打交道的程序的人来说并不是个好消息。想想 DOS，在那儿任何程序都可以直接触及硬件，在这方面 Windows 2000 有些过分讲究。但

这并不意味着在 Windows 2000 中访问硬件是不可能的。不同的是，这种访问被限制到一个特殊类型的模块—内核模式的驱动程序（Kernel-mode driver）。

我可以告诉你，现在我将简要的介绍一下 Kernel-mode driver 编程技术（这本是第三章的内容）。眼下，这已经足够说明 kernel-mode driver 让系统崩溃是非常容易的事。当驱动程序出错时，Windows 2000 没有提供一种错误恢复机制，这就导致即使无意中试图执行一个不合法的操作也会招来蓝屏。当然，最简单而且危险最小的违规动作就是读取一个无效的内存地址。由于系统显示捕获所有通过空指针进行的内存访问，这是 C 编程中一个常见的错误，读取一个空指针是让系统崩溃的理想动作。示例代码中的 w2k_kill.sys driver 就是这么做的。这是一个非常简单的程序，同时也是出现在本书中的第一个 kernel-mode driver。

列表 1-1 是 w2k_kill.c 的引用部分，展示了引发蓝屏的错误代码。当编写这样无意义的代码时，需要注意内建于 Visual C/C++的优化器可能会抵消你的努力，它会跟踪所有的代码并试图消除其中任何有副作用的部分。在下面的例子中，优化器并未起作用，因为 DriverEntry()坚持将在 0 地址发现的东西作为其返回值。这意味着这个数值将会被存放到 CPU 的 EAX 寄存器中，最简单的方法就是 MOV EAX, [0]指令，这个指令将会抛出我们期待的异常。

```
NTSTATUS DriverEntry ( PDRIVER_OBJECT pDriverObject,
                       PUNICODE_STRING puRegistryPath )
{
    return *((NTSTATUS*)0);  // read through NULL pointer
}
```

列表 1-1 A NULL Pointer Read Operation in Kernel-Mode Crashes the System

w2k_load.exe 程序出现在第三章，用来用来载入并启动 w2k_kill.sys 驱动程序。如果你在精神上做好了 kill 掉你的 Windows 2000 系统，请按照下面的步骤来做：

- 关闭所有应用程序
- 插入本书的附带光盘
- 在开始菜单中选择运行
- 输入 d:\bin\w2k_load w2k_kill.sys，用你 CD-ROM 的盘符替换 d:，然后单击确定

当单击后，w2k_load.exe 将试图加载 w2k_kill.sys 文件（位于光盘的\bin 目录下）。随后 DriverEntry()开始执行，蓝屏出现了，如图 1-3 所示，你会看到当内存数据被转储到页面文件存储器时，屏幕上会有一个计数器从 0 逐渐增加到 100。如果你在启动和恢复对话框中选中了自动重起，当崩溃转储完成后，系统会立即重新启动。当系统进入等待登陆状态后，稍等一会直到硬盘灯不再闪烁。这是因为将崩溃转储数据从页面文件复制到磁盘文件需要一定的时间，特别是你的物理内存很大时。在此时干扰系统，例如，将系统过早的关闭，可能会产生一个无效的崩溃转储文件。

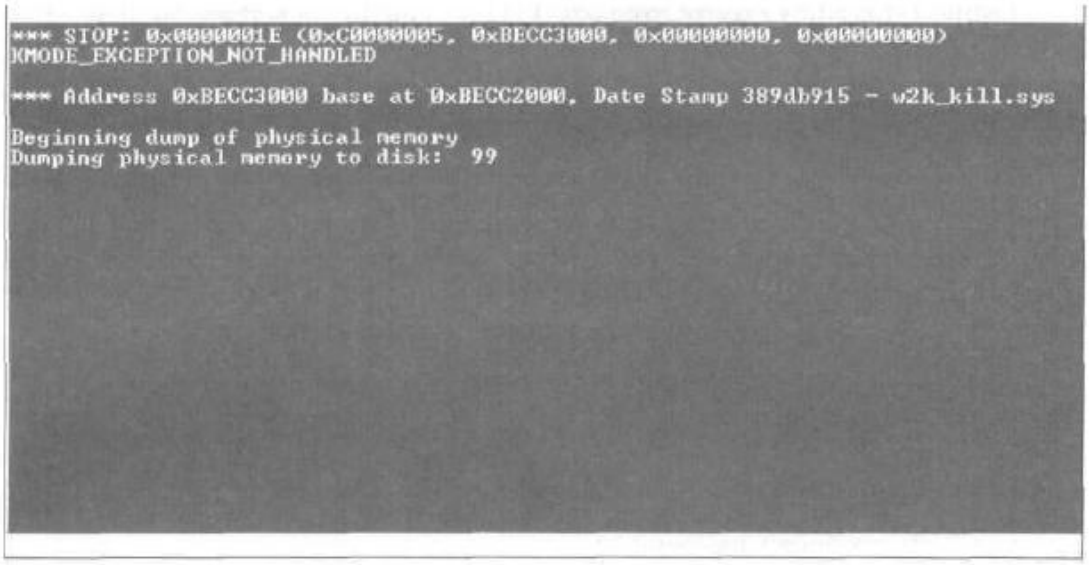


FIGURE 1-3. *Execution of* w2k_kill.sys *Yieldsa Nice Blue Screen*

在图 1-3 中，可以看出系统会显示包含出错代码的模块的名称（w2k_kill.sys），以及引发异常的指令地址（0xBECC3000）。这个地址或许和你的系统的不同，因为它是随硬件配置而变化的。驱动程序的加载地址通常都不确定，和 DLL 的加载地址类似。请记下显示的地址----稍后在安装和配置内核调试器是你还需要它。

一点小的提示：故意让系统崩溃不该是你每天都作的事。尽管有问题的 w2k_kill.sys 本身是无害的，但在它执行的那一刻可能并不那么走运。如果度去空指针发生时另一个线程正在做某些重要的事情，系统可能会在该线程有机会做清理工作之前关闭。比如，在重起之后，活动桌面往往会抱怨发生了一些可恶的事情，它需要进行恢复。因此，在你使系统崩溃之前，应该仔细的检察系统是否影响了重要数据并且保证所有 cache 中的数据都被写入了磁盘。注意，作者和出版商不会对 w2k_kill.sys 驱动程序造成的破坏负责。

译注：达摩克利斯希腊传说中的叙拉古国王狄奥尼西奥斯的朝臣，据传说其被迫坐在上悬宝剑的餐桌旁，宝剑由一根头发系住,以此来暗示君王命运的多危

1.1.3、安装符号文件

重新启动后，你就有了一个 Windows 2000 系统的快照（snapshot），包括一个有问题的 Kernel-mode Driver（读取空指针时被捕获）。观察此快照文件和察看实际系统内存是一样的。当然，这个快照文件和动物死尸一样---不再对外界刺激有所反应，但是你现在不需要担心这些。接下来你需要安装内核调试器需要的符号文件（symbol files），当你分析崩溃文件时，你就会用到了。

MSDN 用户可以在 Windows 2000 Customer Suuport - Diagnostic Tools 光盘上找到这些符号文件。插入光盘，用 IE 打开光盘上的 DBG.HTM 文件，你会看到很多安装选项。如果你运行的是 free build 的 Windows 2000,你最好安装 retail symbols。对于 checked build 版，可以选择安装 debug symbols。安装程序会从 SYMBOLS.CAB 中复制一些.dbg 和.pdb 文件到系统符号文件目录中。默认的系统符号文件目录为：%SystemRoot%\Symbols。%SystemRoot%环境变量代表 Windows 2000 的安装目录。

在起动时，Windows 2000 内核调试器会尝试通过环境变量_NT_SYMBOL_PATH 指示的路径来寻找符号文件，所以最好正确的定义该变量。

译注：

现在可以通过 Symchk.exe 工具来检查和下载最新的符号文件，该工具随 Debugging Tools for Windows 软件包安装。

微软的文档中对于_NT_SYMBOL_PATH 应该指向哪里的说明有些模糊不清。在 DDK 的内核调试一节里提到必须包含符号子目录，即 C:\WINNT\Symbols 或等价目录。而在 SDK 关于 dbghelp.dll 库的文档中，有关符号路径的描述又稍微有些区别：

“该库需要使用符号搜索路径来定位.dll、.exe 或.sys 对应的调试符号(.dbg 文件)。它会在路径后添加\dll、\exe 或\sys。例如，.dll 符号文件位于：C:\WINNT\Symbols\dll，.exe 文件的路径则为：C:\WINNT\Symbols\exe”

。。。。。。

“如果你设置了_NT_SYMBOL_PATH 环境变量，符号管理器按照如下顺序搜索符号文件：

1. *应用程序的当前工作目录*
2. *_NT_SYMBOL_PATH 指示的目录*

3. _NT_ALT_SYMBOL_PATH 指示的目录

4. SYSTMEROOT 指示的目录

”

这样看来把_NT_SYMBOL_PATH 设定为 C:\WINNT 似乎要好于 C:\WINNT\Symbols，为了确定哪种说法是正确的。我试验了这两种方法，很高兴它们都能正常的工作。

1.1.4、配置内核调试器

构建调试环境的最后一步就是安装和配置内核调试器。如果你已经安装了 Windows 2000 DDK，那你可以在\NTDDK\bin 目录中找到调试器。内核调试器的可执行文件名为 i386kd.exe。另一种方法是从 Windows 2000 Customer Support---Diagnostic Tools 光盘中安装。

为了使用前面我们得到的崩溃转储文件，你需要使用 i368kd 的 -Z 选项。示例如下：

i386kd -z C:\WINNT\MEMORY.DMF

成功打开我们的 crash dump 后，你会看到类似图 1-7 所示的东东，kd>提示符会出现，这表示内核调试器已经准备接受命令了。在开始之前，请检查符号搜索路径是否正确。列出的启动信息，表示调试器已经加载了三个扩展 DLL。i386kd.exe 一个强大的特性就是其扩展机制，这允许第三方采用独立的 DLL 来扩展其基本功能。对于这些扩展的命令，要在其前面加上！号以区分内建的命令。

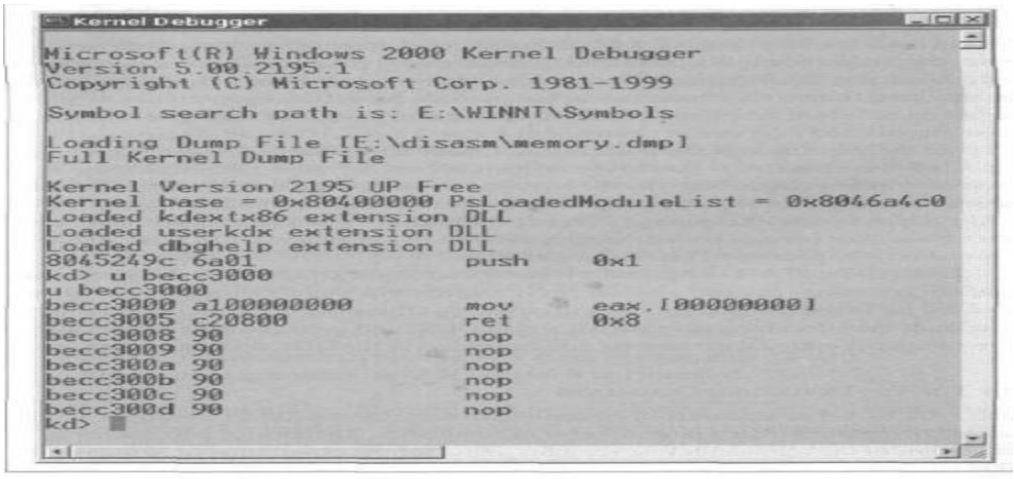


FIGURE 1-7. *Initiating a Kernel Debugger Session*

如图 1-7 所示，我输入了一个内建命令：u becc3000，u 的含义是“反编译（unassembel）”，becc3000 是开始反编译的 16 进制地址。默认情况下，均采用 16 进制，

但你也可也通过命令来改变此默认值。命令为：n 10，此后默认所有数字都是 10 进制表示。你可以使用 0x 前缀来表示这是一个 16 进制数。地址 becc3000 就是 w2k_kill.sys 引起系统崩溃的地方。请使用你在蓝屏时看到的地址。如果一切正确的话，你会看到 mov eax, [00000000]指令，如上图所示。如果没有看到的话，你可能没有使用正确的崩溃转储文件。mov eax, [00000000]指令表示从虚拟地址（也可称作线性地址）0x00000000 读取一个 32 位的数值到 CPU 寄存器 EAX 中，这明显是列表 1-1 中*(NTSTATUS*)0)表达式的实现，等同于读取空指针的操作。没有针对此类错误的异常处理例程，因此，系统在蓝屏上会显示 KMODE_EXCEPTION_NOT_HANDLED，如图 1-3 所示。如果你想知道有关此错误信息更多的东西请参考《The NT Insider》（Open Systems Resources 1999b）。

1.1.5、内核调试器的命令

尽管调试器的命令已经注意了易记性，但有时总是难以回忆起它们。因此，我把它们都整理到了附录 A 中。表 A-1 是其快速参考。这个表是调试器的 help 信息（使用?命令）的整理版。命令所需参数的类型汇总在表 A-2。

前面提到过，内核调试器执行扩展命令时需要一个！号作为开始。只要命令前有个！号，调试器就会到已加载的扩展 DLL 的导出列表中进行查找。如果发现匹配的，就会跳转到相应的 DLL 中。图 1-7 显示了内核调试器加载了 kdextx86.dll、userkdx.dll 和 dbghelp.dll 扩展 DLL。最后一个和 i386kd.exe 位于同一目录；前两个共有四个版本：针对 Windows NT 4.0 的 free 和 checked 版本（对应子目录为：nt4fr 和 nt4chk），针对 Windows 2000 的 free 和 checked 版本。通常，调试器在搜寻扩展命令是会采用一个默认的搜索顺序。然而，你可以改变这个默认设置，只需要在命令前指定模块名称即可，采用.符号作为分隔符。例如，kedxtx86.dll 和 userkdx.dll 都导出了 help 命令，键入!help，在默认情况下，你会得到 kdextx86.dll 的帮助信息。要执行 userdkx.dll 的 help 命令，你必须输入!userkdx.help（或者!userkdx.help -V 如果你想得到更详细的帮助信息的话）。按照此方法，只要你知道规则，你也可以编写自己的扩展命令。在 The NT Insider（Open Systems Resources 1999a）中你能找到很棒的 how-to 文章。不过其针对的是 WinDbg.exe 而不是 i386kd.exe，但两者使用相同的扩展 DLL，大多数信息对于 i386kd.exe 也是有效的。

附录 A 中的表 A-3 和 A-4 分别列出了 kdextx86.dll 和 userkdx.dll 的 help 命令的输出信息。为了便于阅读作了些修改和整编。你会发现这些表中列出的命令多于 Microsoft DDK 文档中的，其中有些命令明显有 DDK 文档未提到的附加参数。

1.1.6、10 大调试命令

表 A-1 到 A-4 列出了内核调试器及其标准扩展 DLL 提供的巨多的命令。因此，我将讨论其中一些常用命令的细节。

u: 反编译机器码

在检查 crash dump 是否正确时，你已经用过了此命令，u 命令有三种格式：

1. u <from> 从地址<from>开始反编译 8 个机器码。
2. u <from> <to> 反编译<from>到<to>之间的所有机器码。
3. u 不提供任何参数时，从上次 u 命令停止的位置开始反编译。

当然，反编译打段代码是十分厌烦的，但如果你只想知道在特定地址发生的事情，那这是最便捷的方法。或许 u 命令最令人感兴趣的特性是它可以解析代码引用到的符号----即使是目标模块没有导出的符号。不过，使用本书光盘中的 Multi-Format Visual Disassembler 反编译完整的 Windows 2000 可执行体将是十分有趣的。在随后章节中，会有关该产品的更多信息。

db，dw 和 dd： Dump Memory BYTEs、WORDs 和 DWORDs

如果你当前感兴趣的内存数据是二进制的，那么调试器的 16 进制转储命令将能完成此任务。根据你对源地址（source address）数据类型的判断，来选择 dd（针对 BYTES）、dw（针对 WORDS）、dd（针对 DWORDS）。

- db 将指定内存范围里的数据显示为两个部分：左边是 16 进制表示（每 2 个 8 bit 一组），右边是对应的 ASCI 码。
- dw 仅按照 16 进制显示（16 bit 一组）
- dw 仅按照 16 进制显示（32 bit 一组）

此组命令可以使用与 u 命令相同的参数。注意，<to>所指示的地址内容，也会被显示出来。如果没有任何参数，将显示接下来的 128 个字节。

x：检查符号

x 命令非常重要。它可以根据已安装的符号文件创建一个列表。典型的使用方式如下：

1. x *!* 显示所有可用符号的模块。在启动后，默认只有 ntoskrnl.exe 的符号是可用的。其他模块的符号可以使用.reload 命令来加载。

2. x <module>!<filter> 显示模块<module>的符号文件中的符号名称，<filter>可以包括通配符?和*。<module>必须属于 x *!*列出的模块名。例如，x nt!*将列出在内核符号文件 ntoskrnl.dbg 中找到的所有符号，x win32k!*将列出 win32k.dbg 提供的符号。如果调试器报告说“Couldn’t resolve ‘X….’ ”，尝试用.reload 再次加载所有的符号文件。

3. x <filter> 显示所有可用符号的一个子集，该子集不匹配<filter>表达式。本质上，这是 x <module>!<filter>的一个变形，在这里<module>!被省略了。

随符号名一起显示的，还有与其相关的虚拟地址。对于函数名，与其对应的就是函数的入口地址。对于变量，就是改变量的基地址。该命令值得的注意的地方是，它可以输出很多内部符号（internal symbols），这些在可执行文件的导出表中都是找不到的。

ln：列出最近的符号

ln 是我最喜欢的一个命令。因为它可以快速且简单的访问已安装的符号文件。算是 x 命令的理想补充。不过后者适用于列出所有系统符号的地址。Ln 命令则用于按照地址或名称查找符号。

- ln <address> 显示<address>指示的地址以及和其前后相邻的地址的符号信息。
- ln <symbol> 将符号名解析为与其对应的虚拟地址。其过程与 ln <address>类似。

像 x 命令一样，调试器知道所有导出的以及一些内部的符号。因此，对于想弄清楚出现在反编译列表或 16 进制转储中的不明指针的确切含义的人有着非常大的帮助。注意，u、db、dw、dd 也会使用符号文件。

!processfield：列出 EPROCESS 的成员

该命令前的!号，意味着它来自于调试器的扩展模块—kdextx86.dll。该命令可显示内核用来代表一个进程的 EPROCESS 结构（该结构并没有正式的说明文档）的成员及其偏移量。

尽管该命令仅列出了成员的偏移量，但你也能很容易的猜出其正确的类型。例如，LockEvent 位于 0x70 处，其下一个成员的偏移量为 0x80。则该成员占用了 16 个字节，这与 KEVENT 结构非常类似。如果你不知道什么是 KEVENT，不要担心，我在第 7 章将会讨论之。

!threadfields：列出 ETHREAD 成员

这是 kdextx86.dll 提供的另一个强大的选项。和!processfields 类似，它列出未文档化的 ETHREAD 结构的成员及其偏移量。内核使用它表示一个线程。参见**示例 1-2**

```
kd> !threadfields
 ETHREAD structure offsets:

    Tcb:                                0x0
    CreateTime:                         0x1b0
    ExitTime:                           0x1b8
    ExitStatus:                         0x1c0
    PostBlockList:                      0x1c4
    TerminationPortList:                0x1cc
    ActiveTimerListLock:                0x1d4
    ActiveTimerListHead:                0x1d8
    Cid:                                0x1e0
    LpcReplySemaphore:                  0x1e8
    LpcReplyMessage:                    0x1fc
    LpcReplyMessageId:                  0x200
    ImpersonationInfo:                  0x208
    IrpList:                            0x20c
    TopLevelIrp:                        0x214
    ReadClusterSize:                    0x21c
    ForwardClusterOnly:                 0x220
    DisablePageFaultClustering:         0x221
    DeadThread:                         0x222
    HasTerminated:                      0x224
    GrantedAccess:                      0x228
    ThreadsProcess:                     0x22c
    StartAddress:                       0x230
    Win32StartAddress:                  0x234
    LpcExitThreadCalled:                0x238
    HardErrorsAreDisabled:              0x239
```

!drivers：列出已加载的 Drivers

kdextx86.dll 真是太棒了。!drivers 列出了当前运行的内核和文件系统模块的详细信息。如果检查 crash dump，该命令会列出系统崩溃那一刻的系统状态。**示例 1-3** 是我机器上输出的摘要。注意，在输出的最后一行，导致 Windows 2000 崩溃的 Driver 的地址为 0xBECC2000，这显然是 w2k_kill.sys 引发蓝屏后显示的地址。

```
kd> !drivers
 kdrivers
Loaded System Driver Summary

Base      Code    Size      Data    Size      Driver Name   Creation Time
80400000  142dc0  (1291 kb)  4d680   (309 kb)  ntoskrnl.exe  Wed Dec 08 00:41:11 1999
80062000   13c40  (  79 kb)  34e0    (  13 kb)      hal.dll  Sun Oct 31 00:48:14 1999
f0810000    1760  (   5 kb)  1000    (   4 kb)  BOOTVID.DLL  Thu Nov 04 02:24:33 1999
f0400000    bdc0  (  47 kb)  22a0    (   8 kb)      pci.sys  Thu Oct 28 01:11:08 1999
f0410000    99c0  (  38 kb)  18e0    (   6 kb)   isapnp.sys  Sat Oct 02 22:00:35 1999
f09c8000     760  (   1 kb)   520    (   1 kb)  intelide.sys  Fri Oct 29 01:20:03 1999
f0680000    42e0  (  16 kb)   e80    (   3 kb)  PCIIDEX.SYS  Thu Oct 28 01:02:19 1999
f0688000    64a0  (  25 kb)   a20    (   2 kb)  MountMgr.sys  Sat Oct 23 00:48:06 1999
bffe3000   192c0  ( 100 kb)  2b00    (  10 kb)    ftdisk.sys  Mon Nov 22 20:36:23 1999
f0900000    12e0  (   4 kb)   640    (   1 kb)  Diskperf.sys  Fri Oct 01 02:30:40 1999
 [...]
bf255000    fc40  (  63 kb)  2120    (   8 kb)    wdmaud.sys  Wed Oct 27 20:40:45 1999
f0670000    9520  (  37 kb)  1f40    (   7 kb)  sysaudio.sys  Mon Oct 25 21:28:14 1999
f094c000    d40   (   3 kb)   860    (   2 kb)    ParVdm.SYS  Tue Sep 28 05:28:16 1999
f0958000    a00   (   2 kb)   480    (   1 kb)   PfModNT.sys  Thu Dec 16 05:14:08 1999
bf0dd000   35520  ( 213 kb)  59e0    (  22 kb)       rv.sys   Tue Nov 30 08:38:21 1999
bf191000    d820  (  54 kb)  1280    (   4 kb)     Cdfs.SYS   Mon Oct 25 21:23:52 1999
bed9a000   11f20  (  71 kb)  2ac0    (  10 kb)    ipsec.sys   Tue Nov 30 08:08:54 1999
beaaf000       0  (   0 kb)     0    (   0 kb)   ATMFD.DLL   Header Paged Out
be9eb000   16f60  (  91 kb)  ccc0    (  51 kb)   kmixer.sys  Wed Nov 10 07:52:30 1999
becc2000     200  (   0 kb)   a00    (   2 kb)  w2k_kill.sys  Sun Feb 06 19:10:29 2000
TOTAL:    79c660  (7793 kb)  15c160  (1392 kb)        0 kb     0 kb
```

EXAMPLE **1-3.** *Displaying Information about System Modules*

译注：

在新的 i386kd.exe（ver: 6.3.0017.0）中，!driver 命令已不被支持。取而代之的是 lm 命令。该命令的一般用法是：lm t n

!sel：检查 Selector 的值

如果没有争议的话，!sel 实现于 kdextx86.dll。它用来显示 16 个连续的 memory selector（按地址升序排列）。你可以反复的使用此命令直到出现“Selector is invalid”。在第 4 章将讨论 Memory Selector，到时我会提供一个示列代码来演示如何在你的程序中 crack selectors。

译注：

在新的调试器中，该命令已不被支持，取而代之的是：dg 命令。其一般性用法为：dg. 注意末尾的.符号。dg 命令最多可列出 256 个 Selector。调试器的 Online Help 中有详细说明

1.1.7、关闭调试器

你可以通过简单的关闭控制台窗口来关闭内核调试器。当然，更好的关闭办法是使用 q 命令，这儿“q”代表着 quit。

1.2、更多的调试工具

在本书的光盘中，你可以找到两个由我的 e-friends 贡献的非常有价值的调试工具。我很高兴他们允许我将完整版本放入我的光盘中。Wayne J. Radburn 的 PE and COFF 文件浏览器（PEView）是本书读者的一个特殊免费工具。Jean-Louis Seigne 的 Multi-Format Visual Disassembler（MFVDasm）一个限时版本。本节将简单介绍这两个工具。

1.2.1、MFVDASM：可视化多格式反编译器

MFVDasm 不仅仅是个汇编列表生成器。事实上，它比汇编代码浏览器增加了多个很不错的导航特性。如图 1-8 所示，那是我使用 MFVDasm 察看 Windows 2000 I/O 管理函数 IoDetachDevice()的截图。图中并没有显示出屏幕上的颜色。例如，所有的函数表以及特定地址的 jumps 和 calls 都被显示为红色。针对其余地址(没有相关导出符号的地址)的 jumps 和 calls 显示为蓝色。引用了从其他模块动态导出的符号的显示为紫色。所有可到达的目标地址（reachable destinations）都加上了下划线，这意味着你可以通过单击他们来滚动代码窗口到达其地址。使用工具栏上的 Back 和 Forward 按钮，你能回顾看过的东东。这很像在 IE 中察看浏览过的网页。

FIGURE 1-8. *MFVDasm Disassembling* ntoskrnl.!o DetachDevice()

在右边，你可以随意选择你想跳到的符号或目标地址。当然，通过单击列头你可以进行排序。在最底层，MFVDasm 提供了 Tab 页来分别显示符号、16 进制转储（HexDump）和重定位（Relocations）。对包含嵌入字符串的代码段进行反编译时 16 进制转储视图会显得很有用。在分析很大的文件如 ntoskrnl.exe 时，MFVDasm 不会阻塞住，和其他流行的反编译工具一样，得到的汇编代码可以保存到文本文件中。

1.2.2、PEView --- PE 和 COFF 文件察看器

尽管 MFVDasm 展示了 PE（Portable Executable）文件的很多内部结构的细节，但其侧重于代码的查看。另一方面，PEView 虽然不能展示比代码段的 16 进制码更进一步的细节，但它能非常详细的显示文件结构的细节。如图 1-9 所示。这是我用 PEView 察看 ntoskrnl.exe 的截图。可以看出 PEView 采用三种形式来显示 ntoskrnl.exe 的多个部分。如果你单击左边的一个叶结点，在右面就会显示与该项相关的所有信息。在图 1-9 种，我选择了 IMAGE_OPTIONAL_HEADER 结构，该结构是 IMAGE_NT_HEADERS 结构的成员之一。

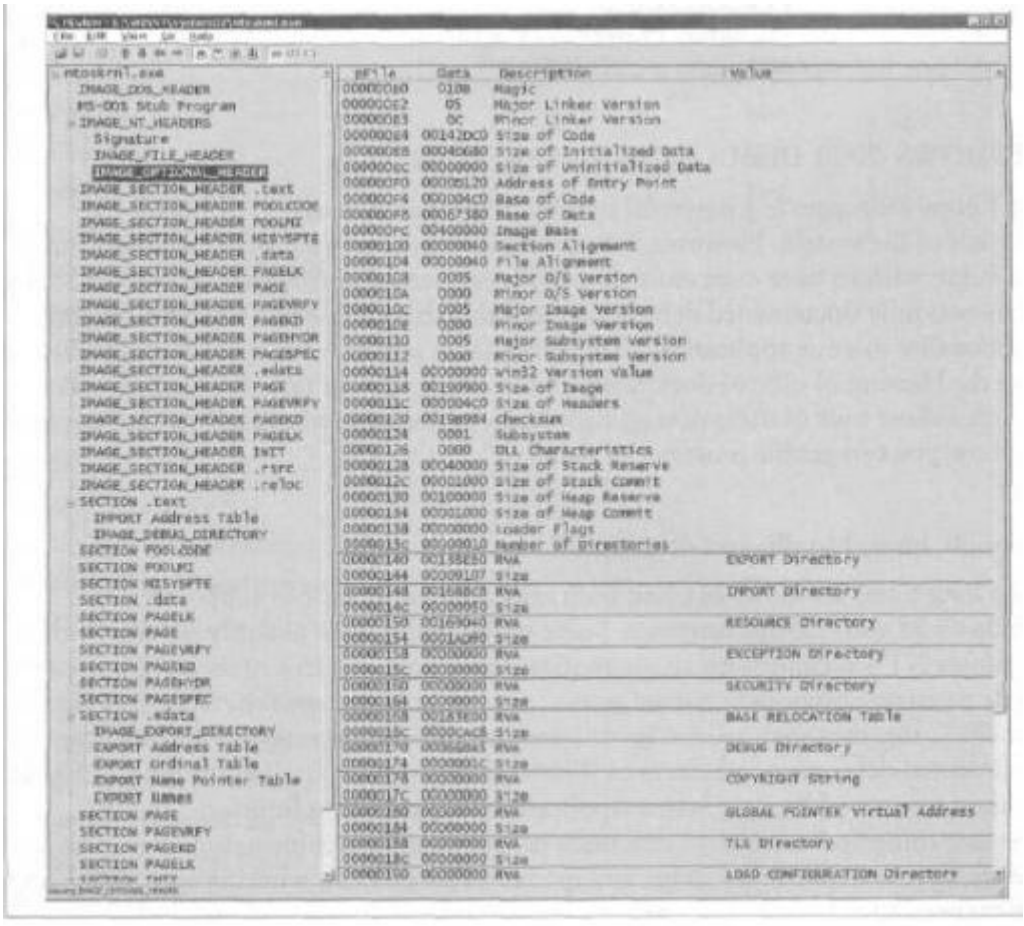


FIGURE 1-9. *PEview Dissecting the PE File Structure of* ntoskrnl.exe

译注：还有两段我没有译，和本书讨论的主题无关，感兴趣的话，去看原文吧。☺

1.2.3、Windows 2000 调试接口

对于喜欢研究系统内核的人来说，内核调试器是一个非常强大的工具。不过，它的界面有些简单。有时你可能希望有更强大的命令。很幸运，Windows 2000 提供了两个完整的调试接口文档，使得你可以在你的程序中加入调试功能。这些接口远算不上豪华（但他们得到了微软官方文档的祝福 ☺）。在本节中，我将带你进行调试接口一日游，向你展示这些文档可以为你做什么以及你如何从这些文档中得到更多东东。

1.2.4、psapi.dll、imagehlp.dll 和 dbghelp.dll

长久以来，Windows NT 由于缺乏对 Windows 95 的 ToolHelp32 接口的支持而备受指责。可能评论家们并不都知道 Windows NT 4.0 提供了其独有的调试接口---内建于系统组件 psapi.dll（随 Win32 SDK 一起发布）中。随此 DLL 一起发布的还有 imagehlp.dll 和 dbghelp.dll 以及针对 NT/2000 的调试接口的官方文档。PSAPI 是 Process Status Application Programming Interface 的首字母缩写，此接口提供了 14 个函数用于获取有关设备驱动程序、进程、进程的内存使用情况和其加载的模块、工作集、内存映射文件的系统信息。psapi.dll 同时支持 ANSI 和 Unicode 字符串。

其余两个调试 DLLs---imagehlp.dll 和 dgbhelp.dll 涵盖了不同的工作范围。二者都导出了相似的函数集合，区别较大的是 imagehlp.dll，它提供了更多函数，但 dbghelp.dll 提供了可重新发布的组件。这意味着微软允许你将 dbghelp.dll 放入你自己的调试程序的安装包中。如果选择使用 imagehlp.dll，你必须获取在目标系统已安装的一个。这两个 DLL 都提供了丰富的函数来分析和维护 PE 文件。二者最显著的特性是能很好的使用符号文件(就是你为内核调试器准备的那些）。为了指导你该选择哪个 DLL，我将这两个 DLL 的所有导出函数汇总到了表 1-1，N/A 表示不支持。

序号	函数名称	ImageHlp.DLL	DbgHelp.DLL
1	BindImage		N/A
2	BindImageEx		N/A

3	CheckSumMappedFile		N/A
4	EnumerateLoadedModules		
5	EnumerateLoadedModules64		
6	ExtensionApiVersion	N/A	
7	FindDebuglnfoFile		
8	FindDebuglnfoFileEx		
9	FindExecutablelmage		
10	FindExecutablelmageEx		
11	FindFilelnSearchPath		
12	GetlmageConfiglnformation		N/A
13	GetlmageUnusedHeaderBytes		N/A
14	GetTimestampForLoadedLibrary		
15	ImageAddCertificate		N/A
16	ImageDirectoryEntryToData		
17	ImageDirectoryEntryToDataEx		
18	ImageEnumerateCertificates		N/A
19	ImageGetCertificateData		N/A
20	ImageGetCertificateHeader		N/A
21	ImageGetDigestStream		N/A
22	ImagehlpApiVersion		
23	ImagehlpApiVersionEx		
24	ImageLoad		N/A
25	ImageNtHeader		
26	ImageRemoveCertificate		N/A
27	ImageRvaToSection		
28	ImageRvaToVa		
29	ImageUnload		N/A
30	MakeSureDirectoryPathExists		
31	MapAndLoad		N/A

32	MapDebuglnformation		
33	MapFileAndCheckSumA		N/A
34	MapFileAndCheckSumW		N/A
35	ReBaselmage		N/A
36	ReBaseImage64		N/A
37	RemovePrivateCvSymbolic		N/A
38	RemovePrivateCvSymbolicEx		N/A
39	RemoveRelocations		N/A
40	SearchTreeForFile		
41	SetlmageConfiglnformation		N/A
42	SplitSymbols		N/A
43	StackWalk		
44	StackWalk64		
45	Sym	N/A	
46	SymCleanup		
47	SymEnumerateModules		
48	SymEnumerateModules64		
49	SymEnumerateSymbols		
50	SymEnumerateSymbols64		
51	SymEnumerateSymbolsW		
52	SymFunctionTableAccess		
53	SymFunctionTa ble Access64		
54	SymGetLineFromAddr		
55	SymGetLineFromAddr64		
56	SymGetLineFromName		
57	SymGetLineFromName64		
58	SymGetLineNext		
59	SymGetLineNext64		
60	SymGetLinePrev		

61	SymGetLinePrev64		
62	SymGetModuleBase		
63	SymGetModuleBase64		
64	SymGetModulelnfo		
65	SymGetModuleInfo64		
66	SymGetModulelnfo Ex		
67	SymGetModulelnfo Ex64		
68	SymGetModulelnfoW		
69	SymGetModulelnfo W64		
70	SymGetOptions		
71	SymGetSearchPath		
72	SymGetSymbolInfo		
73	SymGetSymbolInfo64		
74	SymGetSymFromAddr		
75	SymGetSymFromAddr64		
76	SymGetSymFromName		
77	SymGetSymFromName64		
78	SymGetSymNext		
79	SymGetSymNext64		
80	SymGetSymPrev		
81	SymGetSymPrev64		
82	Symlnitialize		
83	SymLoadModule		
84	SymLoadModule64		
85	SymMatchFileName		
86	SymEnumerateSymbolsW64		
87	SymRegisterCallback		
88	SymRegisterCallback64		
89	SymRegisterFunctionEntryCallback		

90	SymRegisterFunctionEntryCallback64		
91	SymSetOptions		
92	SymSetSearchPath		
93	SymUnDName		
94	SymUnDName64		
95	SymUnloadModule		
96	SymUnloadModule64		
97	TouchFileTimes		N/A
98	UnDecorateSymbolName		
99	UnMapAndLoad		N/A
100	UnmapDebuglnformation		
101	UpdateDebuglnfoFile		N/A
102	UpdateDebuglnfoFileEx		N/A
103	WinDbgExtensionDllInit	N/A	

在本节的示例代码中，我会演示如何使用 psapi.dll 和 imagehlp.dll 完成如下任务：

- 枚举所有内核组件和驱动程序
- 枚举系统当前管理的所有进程
- 枚举加载到进程地址空间的所有模块（modules）
- 枚举一个给定组件的所有符号（如果其符号文件可用的话）

psapi.dll 的接口并不像其设计的那样好。它提供了最小的功能集，尽管它曾试图增加一些便利性。虽然，它能从内核获取一些信息但却扔掉了其中的大多数，只留下很少一部分。

由于 psapi.dll 和 imagehlp.dll 的函数并不是标准 Win32 API 的一部分，它们所需的头文件和导入库不会自动包括在 Visual C/C++工程中。因此，列表 1-2 中列出的四个指示符（directives）应该在你的原文件中出现。第一部分是所需的头文件，剩余部分用于和这两个 DLL 中的导出函数建立动态链接。

```
#include <imagehlp.h>
#include <psapi.h>
```

```
#pragma comment (linker,”/defaultlib:imagehlp.dll”)
#pragma comment (linker,”/defaultlib:psapi.dll”)
```

列表 1-2　增加 psapi.dll 和 imagehlp.dll 到 Visual C/C++工程

译注：

其实，也可以采用静态链接，如下：

```
#pragma comment(lib,”psapi.lib”)
#pragma comment(lib,”imagehlp.lib”)
```

这样，就不需要目标平台必须有这两个 DLL 了。

1.2.5、光盘中的示列代码

在本书的附带光盘中，有两个工程是构建与 psapi.dll 和 imagehlp.dll 之上。其中一个示例工程是 w2k_sym.exe----一个 Windows 2000 符号浏览器，它可以从任意符号文件中提取符号名称（假如你已经安装了的话）。它输出的符号表可以按照名称、地址和大小来排序，同时接受一个采用通配符的过滤器。作为附送功能，w2k_sym.exe 还可列出当前活动的系统模块/驱动程序的名称，运行的进程和每个进程加载的模块。另一个示例工程是调试支持库 w2k_dbg.dll，这个库包含几个便于使用的针对 psapi.dll 和 imagehlp.dll 的外包函数。w2k_sym.exe 完全依赖这个 DLL。这些工程的源代码分别位于光盘的\src\w2k_dbg 和\src\w2k_sym 目录。

表 1-2 列出了 w2k_dbg.dll 用到的函数名称。A./W 列表示对 ANSI 和 Unicode 的支持情况。稍早提示过，psapi.dll 同时支持 ANSI 和 Unicode。不幸的是，imagehlp.dll 和 dbghelp.dll 没有这么聪明,其中几个函数只能接受 ANSI 字符串。这有些烦人,因为 Windows 2000 的调试程序通常不能运行在 Windows 9x 上，所以不该限制使用 Unicode。若将 imagehlp.dll 假如你的工程中，你就必须选择是使用 ANSI 还是来回转化 Unicode 字符串。因为我很讨厌在一个可处理 16 位字符串的系统中使用 8 位的字符串，所以我选择后一种方法。w2k_dbg.dll 导出的所有函数中涉及的字符串默认都是 Unicode。所以，如果你在自己的 Windows 2000 工程中使用这个 DLL 不需要再关心字符大小问题。

另一方面，imagehlp.dll 和 dbghelp.dll 有一个 psapi.dll 没有的特性：他们同样适用于 Win64----让每个开发人员恐惧的 64 位 Windows，这是因为没人知道将 Win32 程序移植到 Win64 有多困难。这些 DLL 导出了 Win64 API 函数，好吧----或许有一天我们会用到他们。

名称	A/W	库
EnumDeviceDrivers		psapi.dll
EnumProcesses		psapi.dll
EnumProcessModules		psapi.dll
GetDeviceDriverFileName	A/W	psapi.dll
GetModuleFileNameEx	A/W	psapi.dll
GetModulelnformation		psapi.dll
ImageLoad	A	imagehlp.dll
ImageUnload		imagehlp.dll
SymCleanup		imagehlp.dll
SymEnumerateSymbols	A/W	imagehlp.dll
Symlnitialize	A	imagehlp.dll
SymLoadModule	A	imagehlp.dll
SymUnloadModule		imagehlp.dll

表 1-2　w2k_dbg.dll 使用的调试函数

我没有深入的探究 psapi.dll 和 imagehlp.dll。本书的焦点在于未文档化的接口，而且在 SDK 中与这两个 DLL 的接口有关的文档还算不错。可是，我并不打算完全绕过它们，因为它们和 Windows 2000 Native API（将在第 2 章讨论）紧密联系在一起。而且，psapi.dll 是证明为什么未文档化的接口比文档化的那个更好的最佳实例。该 DLL 的接口不仅仅只是看上去的简单和笨拙---在某些地方它竟然会返回明显矛盾的数据。如果我不得不编写一个专业的调试工具来出售，我是不会指望这个 DLL 的。Windows 2000 内核提供了强大、通用和更加合适的调试 API 函数。然而，这些几乎都没有文档化。幸运的是，微软提供的许多系统工具都广泛的使用了这些 API，so it has undergone only slight changes across Windows NT versions。是的，如果你使用了这些 API，每当发布了新版的 NT，你就必须修订和小心的测试你的软件，但是它们带来的好处远大于这些障碍。

本章随后的大多数示例代码都来自 w2k_dbg.dll，你可以在光盘的 \src\w2k_dbg\w2k_dbg.c 中发现它们。这个 DLL 封装了多个步骤，以返回更丰富的信息。数据会以合适的大小、链表（包括可选的索引值）返回，以便于对它们进行排序等操作。表 1-3 列出了 w2k_dbg.dll 导出的所有 API 函数。这些函数很多，详细讨论每个函数已经超出了本章的范围，因此我鼓励你去参考 w2k_sym.exe 的源代码（位于光盘\src\w2k_sym\x），来学习它们的典型用法。

表 1-3

函数名称	描　　述
dbgBaseDriver	Return the base address and size of a driver, given its path
dbgBaseModule	Return the base address and size of a DLL module
dbgCrc32Block	Compute the CRC32 of a memory block
dbgCrc32Byte	Bytewise computation of a CRC32
dbgCrc32Start	CRC32 preconditioning
dbgCrc32Stop	CRC32 postconditioning
dbgDriverAdd	Add a driver entry to a list of drivers
dbgDriverAddresses	Return an array of driver addresses (EnumDeviceDrivers ( ) wrapper)
dbgDriverlndex	Create an indexed (and optionally sorted) driver list
dbgDriverList	Create a flat driver list
dbgFileClose	Close a disk file
dbgFileLoad	Load the contents of a disk file to a memory block
dbgFileNew	Create a new disk file
dbgFileOpen	Open an existing disk file
dbgFileRoot	Get the offset of the root token in a file path
dbgFileSave	Save a memory block to a disk file
dbgFileUnload	Free a memory block created by dbgFileLoad ( )
dbglndexCompare	Compare two entries referenced by an index (used by dbgindexsort ( ))

dbglndexCreate	Create a pointer index on an object list
dbglndexCreateEx	Create a sorted pointer index on an object list
dbglndexDestroy	Free the memory used by an index and its associated list
dbglndexDestroyEx	Free the memory used by a two-dimensional index and its associated lists
dbglndexList	Create a flat copy of a list from its index
dbglndexListEx	Create a flat copy of a two-dimensional list from its index
dbglndexReverse	Reverse the order of the list entries referenced by an index
dbglndexSave	Save the memory image of an indexed list to a disk file
dbglndexSaveEx	Save the memory image of a two-dimensional indexed list to a disk file
dbglndexSort	Sort the list entries referenced by an index by address, size, ID, or name
dbgListCreate	Create an empty list
dbgListCreateEx	Create an empty list with reserved space
dbgListDestroy	Free the memory used by a list
dbgListFinish	Terminate a sequentially built list and trim any unused memory
dbgListlndex	Create a pointer index on an object list
dbgListLoad	Create a list from a disk file image
dbgListNext	Update the list header after adding an entry
dbgListResize	Reserve memory for additional list entries
dbgListSave	Save the memory image of a list to a disk file
dbgMemory	Align Round up a byte count to the next 64-bit boundary
dbgMemoryAlignEx	Round up a string character count to the next 64-bit boundary
dbgMemoryBase	Query the internal base address of a heap memory block

dbgMemoryBaseEx	Query the internal base address of an individually tagged heap memory block
dbgMemoryCreate	Allocate a memory block from the heap
dbgMemoryCreateEx	Allocate an individually tagged memory block from the heap
dbgMemoryDestroy	Return a memory block to the heap
dbgMemoryDestroyEx	Return an individually tagged memory block to the heap
dbgMemoryReset	Reset the memory usage statistics
dbgMemoryResize	Change the allocated size of a heap memory block
dbgMemoryResizeEx	Change the allocated size of an individually tagged heap memory block
dbgMemoryStatus	Query the memory usage statistics
dbgMemory	Track Update the memory usage statistics
dbgModuleIndex	Create an indexed (and optionally sorted) process module sub-list
dbgModuleList	Create a flat process module sub-list
dbgPathDriver	Build a default driver path specification
dbgPathFile	Get the offset of the file name token in a file path
dbgPrivilegeDebug	Request the debug privilege for the calling process
dbgPrivilegeSet	Request the specified privilege for the calling process
dbgProcessAdd	Add a process entry to a list of processes
dbgProcessGuess	Guess the default display name of an anonymous system process
dbgProcessIds	Return an array of process IDs (EnumProcesses ( ) wrapper)
dbgProcessIndex	Create an indexed (and optionally sorted) process list
dbgProcessIndexEx	Create a two-dimensional indexed (and optionally sorted) process/module list
dbgProcessList	Create a flat process list

dbgProcessModules	Return a list of process module handles (EnumProcessModules ( )wrapper)
dbgSizeDivide	Divide a byte count by a power of two, optionally rounding up or down
dbgSizeKB	Convert bytes to KB, optionally rounding up or down
dbgSizeMB	Convert bytes to MB, optionally rounding up or down
dbgStringAnsi	Convert a Unicode string to ANSI
dbgStringDay	Get the name of a day given a day-of-week number
dbgStringMatch	Apply a wildcard filter to a string
dbgSymbolCallback	Add a symbol entry to a list of symbols (called by SymEnumerateSymbols ( ))
dbgSymbolIndex	Create an indexed (and optionally sorted) symbol list
dbgSymbolList	Create a flat symbol list
dbgSymbolLoad	Load a module's symbol table
dbgSymbolLookup	Look up a symbol name and optional offset given a memory address
dbgSymbolUnload	Unload a module's symbol table

1.2.6、枚举系统模块和驱动（Drivers）

psapi.dll 可以返回当前内存中的内核模块。这本是非常简单的工作。psapi.dll 的 EnumDeviceDrivers()函数接受一个 PVOID 类型的数组，它将用当前活动的内核驱动模块（active kernel-mode driver）的映像基址（image base address）来填充这个数组，这包括基本的内核模块 ntdll.dll、ntoskrnl.exe、Win32K.sys、hal.dll 和 bootvid.dll。返回值是这些可执行文件映射到的虚拟内存地址（译注，也称作线性地址）。如果你使用内核调试器或其他调试工具检查这些地址的最初几个字节，你将清楚地认出那个有名的 DOS stub 程序，它以著名的 Mark Zbikowski 的首字母大写“MZ”开始，内含一个文本消息--“This program cannot be run in DOS mode”或类似的东西。列表 1-3 展示了一个使用 EnumDeviceDrivers()的简单函数，以及 EnumDeviceDrivers 函数的原型。

```
BOOL WINAPI EnumDeviceDrivers ( PVOID* lpImageBase,
```

```
                                DWORD  cb,
                                PDWORD lpcbNeeded);


PPVOID WINAPI dbgDriverAddresses( PDWORD pdCount )
{
    DWORD dSize;
    DWORD dCount = 0;
    PPVOID ppList = NULL;

    dSize = SIZE_MINIMUM * sizeof( PVOID );

    while ( (ppList = dbgMemoryCreate(dSize)) != NULL )
    {
        if ( EnumDeviceDrivers( ppList, dSize, &dCount) && (dCount < dSize) )
        {
            dCount /= sizeof( PVOID );
            break;
        }
        dCount = 0;
        ppList = dbgMemoryDestroy( ppList );

        if ( (dSize <<= 1) > (SIZE_MAXIMUM * sizeof( PVOID )))
        {
            break;
        }
    }
    if ( pdCount != NULL )
    {
        *pdCount = dCount;
```

```
    }
    return ppList;
}
```

列表 1-3　枚举系统模块地址

EnumDeviceDrivers()期望三个参数：一个数组指针，一个表示输入大小的值以及一个用于输出的类型为 DWORD 的变量。第二个参数指定了传入的数组的字节数，第三个参数表示复制到该数组中的字节数。因此，你必须将返回值除以 sizeof(PVOID)来确定有多少个地址数据复制到了数组中。不幸的是，该函数不能帮助你确定该提供多大的数组，尽管它实际上知道有多少个 Driver 在运行。但它仅仅告诉你返回了多少字节，而且，如果数组太小，它会隐藏多出的字节。因此，你必须使用无聊的 trial-and-error 循环来确定适当的数组大小，就如同**列表 1-3** 所示的那样，只要返回值与数组大小相同就假定还有数据未复制到数组中。在刚开始时，代码中使用了一个合理的最小值--256（由 SIZE_MINIMUM 表示），这通常都足够大了，但是如果不够的话，在开始新的循环时，数组大小会增加为原来的 2 倍，直到获取了所有的指针或者数组大小超过了 65,536。数组使用的内存缓冲区由两个帮助函数 dbgMemoryCreate()和 dbgMemoryDestroy()提供，这两个函数只是 Win32 函数 LocalAlloc 和 LocalFree 的外包而已，这儿就不列出了。

```
BOOL WINAPI EnumDeviceDrivers( PVOID* lpImageBase,
                               DWORD  cb,
                               DWORD* lpcbNeeded)
{
    SYSTEM_MODULE_INFORMATION  smi;
    PSYSTEM_MODULE_INFORMATION psmi;
    DWORD                      dSize, i;
    NTSTATUS                   ns;
    BOOL                       fOk = FALSE;

    ns = NtQuerySystemInformation( SystemModuleInformation,
                                   &smi, sizeof(smi),NULL);
```

```
    if ( (STATUS_SUCCESS == ns) | (STATUS_INFO_LENGTH_MISMATCH == ns) )
    {
        dSize = sizeof(SYSTEM_MODULE_INFORMATION)
                        +  (smi.dCount*sizeof(SYSTEM_MODULE));

        if ( (psmi = LocalAlloc(LEME_FIXED,dSize)) != NULL )
        {
            ns =
NtQuerySystemInformation( SystemModuleInformation,psmi,dSize,NULL );
            if ( ns == STATUS_SUCCESS )
            {
                for( i = 0; (i < psmi->dCount) && (i < cb/sizeof(DWORD));  i++)
                    lpImageBase[i] = psmi->aModules[i].pImageBase;
                *lpcbNeeded = i*sizeof(DWORD);
                fOk = TRUE;
            }
            LocalFree(psmi);
            if ( !fOk )  SetLastError( RtlNtStatusToDosError(ns) );
        }
    }
    else
        SetLastError( RtlNtStatusToDosError(ns) );
    return fOk;
}
```

列表 1-4 EnumDeviceDrivers 函数的示列

列表 1-4 列出了 EnumDeviceDrivers()一种可能的实现方式。注意这并不是来自 psapi.dll 的原始代码。但通过 C 编译器它可以变成等效的二进制代码。为了保持简单干净，我省略了源代码中易分散注意力的细节，比如结构化异常等。在列表 1-4 的中间，你会看到 NtQuerySystemInformation()函数作了很多工作。这是我非常喜欢的 Windows 2000 函数之

一，因为该函数可以访问多种重要的数据结构，如驱动、进程、线程、句柄（handle）和LPC端口列表等等。我的文章“Inside Windows NT Sytem Data”（出版于1999年11月的Dr.Dobb’s Journal）在第一时间提供了有关该函数的内部信息及其搭档函数NtSetSystemInformation()的文档化资料。另外的全面讲述这两个函数的文档可以在Gary Nebbett的《Indispendsable Windows NT/2000 Native API Reference》中找到。

不要过于担心列表1-4列出的EnumDeviceDrivers()函数的实现细节。我增加这些代码片断只是为了例举该函数有趣的一面，这像一根红线贯穿于psapi.dll。在使用SystemModuleInformation标志第二次调用NtQuerySystemInformation()获取了完整的驱动列表后,代码遍历驱动模块数组并将其pImageBase成员复制到调用者提供的指针数组(名为lpImageBase[]）中。这似乎很正确，但除非你不知道NtQuerySystemInformation提供的模块数组所包含的其他信息。这些数据结构都是没有文档化的，但是我现在可以告诉你，这些信息同样是有关模块在内存中的大小、它们的路径和名称、引用计数（load counts）和其他一些标志信息的。甚至文件名在路径中的偏移量也是很容易就能得到的！，EnumDeviceDrivers()残忍的丢掉了所有这些有用的信息,仅仅保留了映像基址(Image Base address）。

所以如果你试图通过返回的指针来获取有关模块的更多信息，则肯定会失败。当你调用GetDeviceDriverFileName()来获取指定映像基址对应的文件路径时，猜猜psapi.dll会怎样做？它会运行与列表1-4类似的代码来获取完整的驱动列表,并遍历该列表来寻找指定的映像基址。如果它找到一个匹配项，就将其路径复制到调用者的缓冲区中。这难道很高效吗？为什么EnumDeviceDrivers不在它首次遍历驱动列表时就复制路径呢？按这样的方式实现此函数并没有多么困难。除去性能问题，这种设计还有另一个潜在的问题：如果在GetDeviceDriverFileName()执行之前指定的模块就已经被卸载了会怎么样呢？该模块的地址将不会出现在第二次获取的驱动列表中，GetDeviceDriverFileName()将会失败。我真不明白微软为什么会发布这样的DLL。

1.2.7、枚举活动进程

psapi.dll的另一个典型工作就是枚举当前系统中运行的进程。为此目的，该DLL提供了EnumProcesses()函数。该函数的工作与EnumDeviceDrivers()十分类似，不过返回的是进程ID而不是虚拟地址了。再次提示，该函数并不会提示缓冲区大小不足，因此我们还

需再次使用 trial-and-error 循环，如列表 1-5 所示，这些代码和列表 1-3 很相似，除了有些不同的符号和类型名称。

一个进程 ID 是一个全局数字标签可在整个系统中唯一标识一个进程。进程和线程 ID 都取自同一个数字池（pool of numbers），从以 0 开始的 Idle 进程，在同一时间，所有运行的进程和线程都不会有相同的 ID。但是，当一个进程结束后，另一个进程可能会再次使用该结束进程或线程的 ID。因此，在 X 时间获取的一个进程 ID 在 Y 时间可能会代表另一个完全不同的进程。也有可能在其使用的那一刻还没有定义或者指定给了某个线程。所以，EnumProcesses()返回一个简单的进程 ID 列表并不能可靠的代表当前系统活动进程的快照。如果考虑该函数的实现方式，这个设计缺陷真是无法原谅。列表 1-6 是 psapi.dll 另一个函数的克隆，大致勾勒出了 EnumProcessees()的基本动作。和 EnumDeviceDrivers()类似，它也依赖 NtQuerySystemInformation()函数，不过在调用时，用 SystemProcessInformation 代替了 SystemModuleInformation。注意列表 1-6 中间的循环，在哪儿 lpidProcess[]数组被来自 SYSTEM_PROCESS_INFORMATION 结构中的数据填充。没什么好惊奇的，该结构也没有文档化。

```
BOOL WINAPI EnumProcesses( DWORD* lpidProcess,
                           DWORD  cb,
                           DWORD* lpcbNeeded);


PDWORD WINAPI dbgProcessIds( PDWORD pdCount )
{
    DWORD dSize;
    DWORD dCount = 0;
    PDWORD pdList = NULL;

    dSize = SIZE_MINIMUM * sizeof( DWORD );

    while ( (pdList = dbgMemoryCreate(dSize)) != NULL )
    {
        if ( EnumProcesses( pdList, dSize, &dCount) && (dCount < dSize) )
```

```
        {
            dCount /= sizeof( DWORD );
            break;
        }
        dCount = 0;
        pdList = dbgMemoryDestroy(pdList);
        if ( (dSize <<= 1) > (SIZE_MXAIMUM*sizeof(DWORD)) )  break;
    }
    if ( pdCount != NULL ) *pdCount = dCount;
    return pdList;
}
```

列表 1-5　枚举进程 ID

在看过 EnumDeviceDrivers()是如何浪费从 NtQuerySystemInformation()返回的数据后，不幸的是，EnumProcesses 也是和其类似的函数，但，事实上，这个函数更糟糕！因为可用的进程信息要远多于驱动模块的信息,因为进程数据之后还包含很多有关系统中每个线程的详细信息。在我写下这段文字时，我的系统正运行着 37 个进程，调用 NtQuerySystemInformation()产了一个 24,488 字节的数据块！而当 EnumProcesses()处理完这些数据后，仅剩下了 148 字节，这些刚好够存放 37 个进程 ID。

尽管 EnumDeviceDirvers()让我有些难过，但 EnumProcesses()却真正伤害了我的心。如果你需要使用未文档化 API 函数的理由，那这两个函数就是最好的证据。如果实际的工作只需一步既可完成，那为什么还要使用如此低效的函数呢？为什么不自己调用 NtQuerySystemInformation()函数自由的获取感兴趣的系统信息？微软提供的许多系统管理工具都依赖于 NtQuerySystemInformation()而不是 psapi.dll，so why settle for less?

```
BOOL WINAPI EnumProcesses( PDWORD lpidProcess,
                           DWORD  cb,
                           PDWORD lpcbNeeded)
{
    PSYSTEM_PROCESS_INFORMATION pspi, pSpiNext;
    DWORD                       dSize, i;
```

```
    NTSTATUS                    ns;
    BOOL                        fOk = FALSE;

    // 0x8000 = 32KB
    for (dSize=0x8000; ((pspi = LocalAlloc(LMEM_FIXED,dSize)) != NULL);
         dSize += 0x8000)
    {
        ns = NtQuerySystemInformation( SystemProcessInformation,pspi,
                                       dSize, NULL);

        if ( STATUS_SUCCESS == ns )
        {
            pSpiNext = pspi;

            for ( i=0; i < cb/sizeof(DWORD); i++ )
            {
                lpidProcess[i] = pspiNext->dUniqueProcessId;

                pSpiNext = (PSYSTEM_PROCESS_INFORMATION)
                           ((BYTE)pSpiNext+pSpiNext->dNext);
            }
            *lpcbNeeded = i * sizeof(DWORD);
            fOk = TRUE;
        }
        LocalFree(pspi);

        if ( fOk || (ns != STATUS_INFO_LENGTH_MISMATCH) )
        {
            if ( !fOk) SetLastError(RtlNtStatusToDosError(ns));
```

```
            break;
        }
        return fOk;
    }
```

列表 1-6　EnumProcesses()函数的示例实现

1.2.8、枚举进程模块

一但你从 EnumProcess()返回的进程列表中发现了你感兴趣的进程 ID，你可能会想知道在此进程的虚拟地址空间中加载了哪些模块。psapi.dll 提供了另一个 API 函数来完成此功能，叫做 EnumProcessModules()。与 EnumDeviceDrivers()和 EnumProcesses()不同，这个函数需要四个参数（参见列表 1-7）。不同于前两个返回系统全局列表的函数，EnumProcessModules()只取回指定进程的列表，因此，增加的那个参数唯一表示一个进程。然而，该函数需要一个进程句柄（HANDLE）来代替进程 ID。为了通过进程 ID 获取其句柄（HANDLE），必须调用 OpenProcess()函数。

```
BOOL WINAPI EnumProcessModule( HNADLE   hProcess,
                               HMODULE* lphModule,
                               DWORD    cb,
                               DWORD*   lpcbNeeded);


PHMODULE WINAPI dbgProcessModules( HANDLE hProcess, PDWORD pdCount)
{
    DWORD    dSize;
    DWORD    dCount = 0;
    PHMODULE phList = NULL;

    if ( hProcess != NULL )
    {
        dSize = SIZE_MINIMUM * sizeof( HMODULE );
```

```
        while ( (phList = dbgMemoryCreate(dSize)) != NULL )
        {
            if ( EnumProcessModules(hProcess,phList,dSize,&dCount))
            {
                if (dCount <= dSize)
                {
                    dCount /= sizeof( HMODULE );
                    break;
                }
            }
            else
            {
                dCount = 0;
            }
            phList = dbgMemoryDestroy(phList);
            if ( !(dSize = dCount) ) break;
        }
    }
    if ( pdCount != NULL) *pdCount = dCount;
    return phList;
}
```

列表 1-7　枚举进程模块

EnumProcessModules()返回指定进程所有模块的句柄的引用。在 Windows 2000 中，一个 HMODULE 只是简单的模块映像基址。在 SDK 头文件 windef.h 中，HMODULE 被定义为 HINSTANCE 的别名，二者都是 HANDLE 类型。严格的来讲 HMODULE 并不是一个句柄。通常，句柄是系统管理的一个表的索引，可通过此表来查找对象属性。系统返回的所有句柄都有一个与特定对象相关的计数器，在一个对象的所有句柄没有返回系统时，该对象不能从内存中被移除。Win32 API 提供了 CloseHandle()函数用于关闭句柄。该函数与 Native API NtClose()等价。有关 HMODULEs 最重要的事情是，这些“handles”不需要关闭。

另一件让人困惑的事是，事实上，模块句柄通常并不被保证是一直有效的。SDK 的 GetModuleHandle()函数文档提示到，在多线程程序中必须更加注意模块句柄，因为一个线程可以通过卸载HMODULE引用的模块而让另一个线程拥有的HMODULE无效。在多任务环境下，一个程序（如调试器）使用另一程序的模块句柄时也许注意这一点。这似乎使 HMODULEs 没有多大用处了，但是，在下面两种情况中，HMODULE 的有效性会保持足够长的时间：

1. 由 LoadLibrary()或 LoadLibraryEx()返回的 HMODULE 在进程调用 FreeLibrary()之前都会一直有效，由于这些函数包含了模块引用计数，所以即使在多线程程序中，这也会阻止模块被意外卸载。

2. 如果 HMODULE 指向的模块会永久的存在，那么它也会一直有效。例如，所有 Windows 2000 内核组件（不包括内核模式的驱动程序）总是被映射到每个进程的相同固定地址上，并且在进程生命期里一直在那里。

不幸地是，这些情况并不适用于 EnumProcessModules()函数返回的模块句柄，至少通常不行。复制到调用者提供的缓冲区中的 HMODULE，在获取进程快照那一刻其所表示映像基址是有效的。稍后，进程可能调用 FreeLibrary()来释放一个或多个模块，并将其从内存中移除，此时它们的句柄将无效，随后进程很有可能立即调用 LoadLibrary()加载了另一个 DLL，而此新模块恰好映射到了前面释放的地址上。这看上去是不是很熟悉？是的，同样的问题也存在于 EnumDeviceDrivers()的指针数组和 EnumProcesses()函数的 ID 数组。不过，这些问题是可以避免的。psapid.dll 通过调用未文档化的 API 函数来完成数据收集工作后，考虑这些数据的完整性，可返回一个完整的请求对象的快照，其中应包括所有感兴趣的属性信息。这样就没有必要在稍后调用另一个函数来获取附加的信息了。我的观点是，psapi.dll 的设计过于简单，因为它忽略了数据的完整性，这也是我不会将此 DLL 作为一个专业调试工具的基础的原因。

与 EnumDeviceDrivers()和 EnunProcesses()函数相比 EnumProcessModules()函数算是个好公民了，因为如果调用者提供的缓冲区不能放下全部的输出数据，它会准确地提示有多少字节没有复制。注意列表 1-7 没有包括一个循环，在那里缓冲区会不断增大直到足够的大。然而，仍然需要 trial-and-error 循环，因为在下一次调用时，EnumProcessModules 报告的所需大小可能已经无效了（如果指定进程在两次调用之间又加载了新的模块）。因此，列表 1-7 中的代码将不断枚举模块直到 EnumProcessModules()报告需要的缓冲区等于或小于实际可用大小，或者出现了错误。

我不想描述 EnumProcessModules()的等价函数，因为该函数要比 EnumDeviceDrivers 和 EnumProcesses 稍微复杂些，它涉及几个未文档化的数据结构。基本上，它还是通过调用 NtQuerySystemInformation()函数（当然，该函数也没有文档化）来获取目标进程环境块（PEB）的地址，通过该地址可获取一个模块信息链表。因为不管是 PEB 还是这个链表在调用进程的地址空间都是无法直接使用的，EnumProcessModules 调用 Win32 API ReadProcessMemory()（该函数有文档记载）来遍历目标进程的地址空间。顺便说一下，PEB 结构的布局将在第 7 章讨论，在附录 C 中，可以找到该结构的定义。

1.2.9、调整进程特权

回忆一下稍早讨论过的有关 EnumProcessModules 所需的进程句柄。通常，你首先得到的是进程 ID---可能是 EnumProcesses 返回的进程 ID 集中的一个。Win32 API OpenProcess()可通过进程 ID 来获取其句柄。这个函数期望一个访问标志符作为其第一个参数。假定进程 ID 存放在一个 DWORD 类型的变量 dId 中，你以最大访问权限来调用 OpenProcess，如下：

```
OpenProcess(PROCESS_ALL_ACCESS，FALSE，dId)
```

以获取该进程的句柄，此时你会收到一个针对几个低 ID 进程的错误代码。这并不是 bug---这是安全特性！这些进程都是保持系统活动的系统服务。一个普通用户进程不允许执行针对系统服务的所有操作。例如，允许所有进程都可以杀死系统中其余进程并不是个好主意。如果一个程序意外终止了一个系统服务，那么整个系统都将崩溃。因此，一个进程只有拥有确切的访问权限才会有适当的特权。

由于多种原因，调试器必须拥有大量的权限来完成他的工作。改变进程的特权可通过以下三个简单的基本步骤：

1．首先，必须打开进程的访问令牌（access token），使用 advapi32.dll 中的函数 OpenProcessToken()。

2．如果上一步正确完成，接下来就是准备 TOKEN_PRIVILEGES 结构，该结构包含有关要请求的特权的信息。这个工作需要 advapi32.dll 中的另一个函数 LookupPrivilegeValue()的帮助。特权通过名称来指定。SDK 文档 winnt.h 定义了 27 中特权名称和其对应的符号名称。例如，调试权限的符号名称为：SE_DEBUG_NAME，该名称和字符串“SeDebugPrivilege”等效。

3．如果上一步正确完成，就可以使用进程的令牌句柄（Token Handle）来调用AdjustTokenPrivileges()函数以初始化TOKEN_PRIVILEGES结构。该函数也是advapi32.dll导出的。

如果OpenProcessToken()调用成功，要记得关闭其返回的令牌句柄（Token Handle）。w2k_dbg.dll包含一个dbgPrivilegeSet()函数，该函数合并了这几个步骤，下面的列表1-8列出了该函数和w2k_dbg.dll中的另一个函数：dbgPrivilegeDebug()。此函数是dbgPrivilegeSet()的一个外包函数，为了便于设定调试特权。顺便说一下，Windows NT Server资源工具集中的kill.exe也使用了同样的技巧。Kill.exe需要调试特权来剔除内存中饿死的服务（starved services）。这是NT Server管理员不可缺少的一个工具，这对于重起一个挂掉的系统服务十分有用，而且可以避免不必要的系统重启。对于使用IIS（Internet Information Server）的人，在他们的紧急工具箱中可能都有这个工具，以便重起偶尔挂掉的inerinfo.exe。

```
BOOL WINAPI dbgPrivilegeSet(PWORD pwName)
{
    HANDLE           hToken;
    TOKEN_PRIVILEGES tp;
    BOOL             fOk = FALSE;

    if ( (pwName != NULL) &&
        OpenProcessToken(GetCurrentProcess(),
                         TOKEN_ADJUST_PRIVILEGES,
                         &hToken) )
    {
        if ( LookupPrivilegeValue(NULL,pwName,&tp,Privileges->Luid) )
        {
            tp.Privileges->Attributes = SE_PRIVILEGE_ENABLED;
            tp.PrivilegeCount         = 1;

            fOk = AdjustTokenPrivileges(hToken,FALSE,&tp,0,NULL,NULL)
```

```
                    && (GetLastError() == ERROR_SUCCESS);
        }
        CloseHandle(hToken);
    }
    return fOk;
}
//--------------------------------------------------------------------------
-------
BOOL WINAPI dbgPrivilegeDebug(void)
{
    return dbgPrivilegeSet(SE_DEBGU_NAME);
}
```

列表 1-8 Requesting a Privilege for a Process

1.2.10、枚举符号

毫不留情的批判完 psapi.dll 后，现在是说些好话的时候了。psapi.dll 可能比较失败，但另一方面，imagehlp.dll 却十分完美！ 我曾深入研究这个设计精巧的软件，以寻找有关 Windows 2000 符号文件的内部结构的更多信息。最终，世界著名的“Windows 外科专家” Matt Pietrek 在三年前发表的一篇文章让我确信：至少到现在还绝对没有必要了解符号文件的格式，因为 imagehlp.dll 可以很容易的分析它们。但现在这个神话被 SymEnumerateSymbols()打破了。该函数的原形在**列表 1-9** 中列出。就目前来说，我已经学到了 Windows NT 4.0 和 Windows 2000 的符号文件的大多数基本的内部细节，因此我不再需要 imagehlp.dll 了。我会在下一节覆盖这部分内容。

hProcess 参数通常是调用进程的句柄,因此可以使用 GetCurrentProcess()的返回值。注意：GetCurrentProcess()不会返回实际的进程句柄。它实际返回一个常量—0xFFFFFFFF，我们称之为伪句柄，它可以被所有期望一个进程句柄的函数使用。0xFFFFFFFE 是另一个用来代表当前线程的伪句柄，由 GetCurrentThread()返回。

BaseOfDll 被定义为 DWORD，尽管其实际类型为 HMODULE 或 HINSTANCE。我猜测微软选择这个数据类型是为了表明该值并不需要必须是一个有效的 HMODULE，尽管它大多数情况下都是。SymEnumeraterSymbols()计算所有与该值相关的所有可列举符号的基地址。该函数完

全可以查询一个没有被加载到任何进程地址空间中的 DLL 的符号，所以 BaseOfDll 可以指定为任意值。

```
BOOL IMAGEAPI SymEnumerateSymbols(

      HANDLE     hProcess,

      DWORD      BaseOfDll,

      PSYM_ENUMSYMBOLS_CALLBACK  CallBack,

      PVOID      UserContext);

     typedef BOOL (CALLBACK *PSYM_ENUMSYMBOLS_CALLBACK)

     (PTSTR  SymbolName,

     DWORD   SymbolAddress,

     DWORD   SymbolSize,

     PVOID   UserContext);
```

列表 1-9 SymEnumerateSymbols()和它的 CallBack 函数

CallBack 参数是一个指向用户定义的 callback 函数的指针，该指针函数针对每个符号来调用。**列表 1-9** 给出了该参数的相关信息。这个 CallBack 函数接受一个以 0 结尾的符号名字符串，符号的基地址将作为 SymEnumerateSymbols 的 BaseOfDll 参数，the estimated size of the item tagged by the symbol.SymbolName 定义为 PTSTR，这意味着其实际类型依赖于调用的是 SymEnumerateSymbols 的 ANSI 版还是 Unicode 版。SDK 文档明确规定 SymbolSize 是一个最佳猜测值（best-guess value）或者为 0。我发现 SymbolAddress 最好为 0，UserContext 是一个指针，调用者用它来跟踪记录枚举顺序。例如，它可能指向一块用来存放符号信息的内存块。该指针也被传递给 CallBack 函数，作为该函数的 UserContext 参数。CallBack 函数可在任何时候取消枚举操作，并返回 FALSE。该操作发生的典型情况是发生了无法恢复的错误或者调用者收到他消息，表示其需要等待的时。

```
PDBG_LIST WINAPI dbgSymbolList( PWORD pwPath, PVOID pBase)
```

```
{

    PLOADED_IMAGE pli;

    HANDLE        hProcess = GetCurrentProcess();

    PDBG_LIST     pdl = NULL;

    if ( NULL != pwPath && SymInitialize(hProcess,NULL,FALSE) )

    {

        if ( (pli = dbgSymbolLoad(pwPath,pBase,hProcess)) != NULL )

        {

            if ( (pdl = dbgListCreate()) != NULL )

            {

                SymEnumerateSymbols(hProcess,(DWORD_PTR)pBase,

                                    dbgSymbolCallback,fcpdl);

            }

            dbgSymbolUnload(pli,pBase,hProcess);

        }

        SymCleanup(hProcess);

    }

    return dbgListFinish(pdl);

}
```

列表 1-10 创建一个符号链表

列表 1-10 示范了一个使用 SymEnumerateSymbols()的典型程序，其源代码来自 w2k_dbg.dll。该程序枚举指定模块的符号，在此之前必须完成如下步骤：

1. 在做任何事之前，必须先调用 SymInitialize()来初始化符号句柄。列表 1-11 给出了该函数的原型和另一个在此讨论的函数。hProcess 参数可以是系统中任何活动进程的句柄。调试器使用该参数来标识目标进程。SymInitialize()分配的资源必须调用 SymCleanup()进行释放。

2. 要获取有关模块的精确信息，最好调用 ImageLoad()。注意，该函数由 imagehlp.dll 导出，dbghelp.dll 没有此函数。ImageLoad()返回一个 LOADED_IMAGE 结构的指针，该结构包含被加载模块的详细信息（参见**列表 1-11**）。该结构必须使用 ImageUnload()进行释放。

3. 在调用 SymEnumerateSymbols()之前的最后一步就是执行 SymLoadModule()。如果 ImageLoad()在此之前已被调用，那么将其返回的 LOADED_IMAGE 结构中的 SizeOfImage 和 hFile 作为参数传递给 SymLoadModule()。否则,你就必须将 hFile 设置为 NULL，并将 SizeOfImage 设为 0。在此种情况下，SymLoadModule()将试图从符号文件中获取映像的大小,但这不能保证精确。稍后须调用 SymUnloadModule()来释放符号表。

```
BOOL IMAGEAPI SymInitialize(HANDLE hProcess,
                            PSTR UserSearchPath,
                            BOOL fInvadeProcess);
                             BOOL IMAGEAPI SymCleanup(HANDLE hProcess);
                             DWORD IMAGEAPI SymLoadModule(HANDLE hProcess,
                               HANDLE hFile,
                               PSTR    ImageName,
                               PSTR    ModuleName,
```

```
                                DWORD  BaseOfDll,
                                DWORD  SizeOfDll);
 BOOL IMAGEAPI SymUnloadModule(HANDLE hProcess,DWORD BaseOfDll);
 BOOL IMAGEAPI ImageUnload(PLOADED_IMAGE LoadedImage);
 typedef struct _LOADED_IMAGE
{
    PSTR                    ModuleName;
    HANDLE                  hFile;
    PUCHAR                  MappedAddress;
    PIMAGE_NT_HEADERS       FileHeader;
    PIMAGE_SECTION_HEADER   LastRvaSection;
    ULONG                   NumberOfSections;
    PIMAGE_SECTION_HEADER   Sections;
    ULONG                   Characteristices;
    BOOLEAN                 fSystemImage;
    BOOLEAN                 fDOSImage;
    LIST_ENTRY              Links;
    ULONG                   SizeOfImage;
} LOADED_IAMGE, *PLOADED_IMAGE;
```

列表 1-11　imagehlp.dll 的一些函数原型

在**列表 1-10** 中，对 SymInitialize()、SymEnumerateSymbols()和 SymCleanup()的调用很清晰。这里请先忽略 dbgListCreate()和 dbgListFinish()调用，这两个函数涉及到 w2k_dbg.dll,主要是用来帮助在内存中建立对象链表。前面提到的 imagehlp.dll 中的函数，隐含于 w2k_dbg.dll 中的 dbgSymbolLoad()和 dbgSymbolUnload()中，请参考**列表 1-12** 中。注意，dbgSymbolLoad()使用 dbgStringAnsi()将模块路径字符串从 Unicode 转化为 ANSI，这是因为 imagehlp.dll 没有导出一个支持 Unicode 的 ImageLoad()。

```
PLOADED_IMAGE WINAPI dbgSymbolLoad (PWORD  pwPath,
                                    PVOID  pBase,
                                    HANDLE hProcess)
    {
    WORD           awPath [MAX_PATH];
    PBYTE          pbPath;
    DWORD          dPath;
    PLOADED_IMAGE pli = NULL;
    if ((pbPath = dbgStringAnsi (pwPath, NULL)) != NULL)
        {
        if (((pli = ImageLoad (pbPath, NULL)) == NULL)         &&
            (dPath = dbgPathDriver (pwPath, awPath, MAX_PATH)) &&
            (dPath < MAX_PATH))
            {
            dbgMemoryDestroy (pbPath);
```

```
            if ((pbPath = dbgStringAnsi (awPath, NULL)) != NULL)
                {
                pli = ImageLoad (pbPath, NULL);
                }
            }
        if ((pli != NULL)
            &&
            (!SymLoadModule (hProcess, pli->hFile, pbPath, NULL,
                             (DWORD_PTR) pBase, pli->SizeOfImage)))
            {
            ImageUnload (pli);
            pli = NULL;
            }
        dbgMemoryDestroy (pbPath);
        }
    return pli;
    }
// ---------------------------------------------------------------
PLOADED_IMAGE WINAPI dbgSymbolUnload (PLOADED_IMAGE pli,
                                      PVOID         pBase,
```

```
                                        HANDLE        hProcess)

    {

    if (pli != NULL)

        {

        SymUnloadModule (hProcess, (DWORD_PTR) pBase);

        ImageUnload     (pli);

        }

    return NULL;

    }

// ------------------------------------------------------------------

PDBG_LIST WINAPI dbgSymbolList (PWORD pwPath,

                                PVOID pBase)

    {

    PLOADED_IMAGE pli;

    HANDLE        hProcess = GetCurrentProcess ();

    PDBG_LIST     pdl      = NULL;

    if ((pwPath != NULL) &&

        SymInitialize (hProcess, NULL, FALSE))

        {

        if ((pli = dbgSymbolLoad (pwPath, pBase, hProcess)) != NULL)
```

```
            {
            if ((pdl = dbgListCreate ()) != NULL)
                {
                SymEnumerateSymbols (hProcess, (DWORD_PTR) pBase,
                                     dbgSymbolCallback, &pdl);
                }
            dbgSymbolUnload (pli, pBase, hProcess);
            }
        SymCleanup (hProcess);
        }
    return dbgListFinish (pdl);
    }
```

列表 1-12. 加载/卸载符号信息

即使仅提供模块名，不提供任何路径信息，ImageLoad()也可定位指定的模块。不过，对于\winnt\system32\drivers 目录下内核驱动程序，该函数将调用失败。因为它该目录并不是系统搜索路径的一部份。此时，dbgSymbolLoad()将调用 dbgPathDriver()函数，然后再尝试调用 LoadImage()。如果路径中仅包含一个文件名，那么 dbgPathDriver()将简单的在指定路径前增加一个”driver\”前缀。如果这两次 ImageLoad()调用中有任意一个能返回一个有效的 LOADED_IMAGE 指针，则 dbgSymbolLoad()将加载模块的符号表（使用 SymLoadModule()函数),并返回 LOADED_IMAGE 结构(如果成功的话),至此,dbgSymbolLoad()的任务就完成了。与之很相似的是 dbgSymbolUnload()，这个函数的价值不大，它完成卸载符号表，和销毁 LOADED_IMAGE 结构的工作。

在**列表 1-10** 中，SymEnumerateSymbols()被指示使用 dbgSymbolCallback()函数来进行回调（callback），dbgSymbolCallback()是 w2k_dbg.dll 中的一个导出函数。我没有给出 SymEnumerateSymbols()的源代码，因为它是 imagehlp.dll 中的一个函数。该函数仅使用它接收到的符号信息（参见**列表 1-9** 中的 PSYM_ENUMSYMBOLS_CALLBACK 的定义）并将此符号信息保存到 UserContext 指向的内存块中，UserContext 指向的内存块由调用者分配。尽管 w2k_dbg.dll 提供的链表、索引和排序函数等特性仅对其自己有用，但它们也属于本书的范畴。如果你需要更多的信息，请参考 w2k_dbg.dll 和 w2k_sym.exe 的源代码。

1.3、Windows 2000 符号浏览器

w2k_sym.exe 是 w2k_dbg.dll 的一个客户端程序，它运行于 Win32 控制台下。如果不使用任何参数调用它，它将显示如**示例 1-5** 所示的内容。w2k_sym.exe 可识别多个命令行选项，它根据这些选项来确定它该执行什么样的操作。四个基本的选项是：/p（列出进程）、/m（列出进程模块）、/d（列出驱动和系统模块）、或者你要查看符号信息的模块的全路径。通过使用排序、过滤等显示模式选项可以修改程序的默认显示方式。例如，如果你打算查看 ntoskrnl.exe 的所有符号，并按照名称进行排序，可使用命令：`w2k_sym /n /v ntoskrnl.exe`。/n 选项表示按名称进行排序，/v 选项表示显示详细的信息，否则，将只显示一些汇总信息。

```
// w2k_sym.exe

// SBS Windows 2000 Symbol Browser V1.00

// 08-27-2000 Sven B. Schreiber

// sbs@orgon.com

Usage: w2k_sym { <mode> [ /f | /F <filter> ] <operation> }

<mode> is a series of options for the next <operation>:

        /a : sort by address

        /s : sort by size

        /i : sort by ID (process/module lists only)

        /n : sort by name
```

```
        /c : sort by name (case-sensitive)

        /r : reverse order

        /l : load  checkpoint file (see below)

        /w : write checkpoint file (see below)

        /e : display end address instead of size

        /v : verbose mode

/f <filter> applies a case-insensitive search pattern.

/F <filter> works analogous, but case-sensitive.

In <filter>, the wildcards * and ? are allowed.

<operation> is one of the following:

        /p : display processes      - checkpoint: processes.dbgl

        /m : display modules        - checkpoint: modules.dbgl

        /d : display drivers        - checkpoint: drivers.dbgl

    <file> : display <file> symbols - checkpoint: symbols.dbgl

<file> is a file name, a relative path, or a fully qualified path.

Checkpoint files are loaded from and written to the current directory.

A checkpoint is an on-disk image of a DBG_LIST structure (see w2k_dbg.h).
```

示例 1-5. w2k_sym.exe 的帮助信息

作为一个附加功能，w2k_sym.exe 允许读取/写入检查点文件。一个检查点文件只是一对一的将对象列表写入到磁盘文件中。你可以使用检查点来保存系统的当前状态以便以后进行比对。检查点文件中包含一个 CRC32 字段，在加载检查点文件时将使用该字段检查文件

内容的有效性。w2k_sym.exe 在当前目录下维护 4 个检查点，分别对应前面提到的四个选项，即进程、模块、驱动和符号列表。

1.3.1、深入微软符号文件

微软提供了一个标准的接口来访问 Windows 2000 的符号文件（Symbol file），该接口隐藏了这些文件的内部格式。有时，你可能想直接读取这些符号文件，以获取进一步的控制权。在这一节里，我将向你展示符号文件.pdb 和.dbg 中的数据的结构形式，并提供了一个 DLL 和一个示例性的客户端程序来使你可以查找和浏览符号文件的内部信息。是的，这将是另一个符号查看程序，不过不要担心，这将是一个全新的程序，和我们先前讨论的将完全不同。

1.3.2、符号的编码方式

符号名称在微软符号文件中的存储格式叫做编码格式（原始的符号都使用前缀/后缀进行了一定的修饰），这意味着这些增加了前缀和后缀的符号名将包含更多有关类型和如何使用符号的信息。**表 1-4** 列出了最常见的编码方式。由 C 代码生成的符号通常有一个下划线或者一个@符号，这与采用的调用约定有关。@表示这是一个__fastcall 函数，下划线表示__stdcall 和__cdecl 函数。__fastcall 和__stdcall 约定将参数堆栈的清理工作留给了被调用函数，同时这种类型的函数还表示参数所占字节数将由调用者放入堆栈中。这些信息被增加到符号化名称上，由@符号分隔开来。此种情况下，全局符号将按照__cdecl 函数对待，这意味着它们的符号将由一个下划线开始但结尾处没有参数堆栈的相关信息。

表 1-4. 符号编码方式的分类

示　例	描　　述
symbol	未修饰的符号（可能定义于 ASM 模块中）
_symbol	__cdecl 函数或全局变量
_symbol@N	__stdcall 函数（其参数占用 N 字节）
@ symbol@N	__fastcall 函数（其参数占用 N 字节）
_imp_symbol	__cdecl 函数或变量的 import thunk
_imp_symbol@N	__stdcall 函数（其参数占用 N 字节）的 import thunk
imp@symbol@N	__fastcall 函数（其参数占用 N 字节）的 import thunk

?symbol	内嵌参数类型信息的 C++符号
__@@_PchSym_symbol	PCH 符号

符号名称中的_imp__或_imp_@前缀表示这是一个 import thunk，import thunk 是指一个函数指针或位于其他模块中的变量。Import thunk 可以在运行时更容易的动态链接到其他模块的导出符号上,而不需要关心目标模块的实际加载地址。当一个模块加载到内存之后，载入机制将修改 Thunk 指针使其指向实际的入口点地址。Import thunk 的优势在于对导入的函数或者变量只需修改一次，所有对此外部符号的引用都将导向该符号的 Thunk。应该注意的是 Import thunk 并不是必须的。这取决于编译器是想使用 thunks 来使所需的修改最小化，还是不使用 thunks 来最小化内存使用率（使用 thunks 将占用一定的内存空间）。如表 1-4 所示，相同的前缀/后缀规则也可应用于本地的导入符号，但前缀为__imp_的 import thunks 除外（注意，该前缀有两个下划线）。

通过先前示范的w2k_sym.exe可很容易得发现Imagehlp.dll的在对符号进行解码（undecoration）时存在问题，只所以w2k_sym.exe可以证明这一点是因为它实际上使用的是imagehlp.dll中的API（通过w2k_dbg.dll）。如果你使用命令：`w2k_sym /v /n /f __* ntoskrnl.exe`，来要求w2k_sym.exe显示有两个下划线的符号（按名称排序），你会发现有些地方和**示例 1-6** 并不相同。在列表顶部堆积的大量字符让人非常困惑。在内核调试器中使用`ln 8047F798`得到结果却是：ntoskrnl!__，这更让人气愤。地址 8047F798 处的符号的原始修饰名实际居然是__@@_PchSym_@00@UmgUkirezgvUmlhUlyUfkUlyqUrDIGUlykOlyq@ob,看起来似乎是imagehlp.dll简单的丢弃了除两个下划线外的所有符号。

```
     #  ADDRESS      SIZE NAME

  6870  8047F798        4 __
  6871  80480B8C       14 __
  6872  8047E724        4 __
  6873  80471FE0        4 __
  6874  804733B8       28 __
  6875  804721D0       20 __
  6876  804759A4        4 __
  6877  80480004       1C __
  6878  8047DA8C       14 _
  6879: 8047238C        4 __
  6880  8047B6D4        4 __
  6881  804755D4        4 __
  6882: 80471700        4 __decimal_point
  6883  80471704        4 __decimal_point_length
  6884  80471FC0        8 __fastflag
```

更好一些的示例命令是：`w2k_sym /v /n /f _imp_* ntoskrnl.exe`，该命令显示所有以_imp_开头的符号，这些符号就是 ntoskrnl.exe 的 import thunks。示例 1-7 给出部分符号。这里再次出现了与示例 1-6 类似的情况，列表开始都是一大段非常晦涩难懂的名字，而内核调试器也无能为力（内核调试器给出的这些地址对应的符号名与列表中的相同）。如果我告诉你地址 0x804005A4 处的原始符号名其实是__imp_@ExReleaseFastMutex@4，你会怎么想呢？很显然，有一个下划线迷路了，第一个@之后的整个字符串都丢失了。看起来 imagehlp.dll 中的解码算法在处理@符号时存在着问题。这种奇怪行为的原因是：@并不只是__fastcall 函数名的前缀，它也是__fastcall 和__stdcall 函数末尾用来分隔堆栈大小的分隔符，显然，解码算法应该能查找第一个下划线和@符号，错误的假定末尾的剩余部分指定了参数在调用者堆栈中所占的字节数。因此，较长的 PCH 符号被从两个下划线之后截断了，__fastcall 的 import thunk 被简化为_imp_。在这两种情况下，第一个下划线将被移除并且第一个@以及其后的字符都将被丢掉。

```
    #  ADDRESS     SIZE NAME

6761:  804005A4       4 _ mp_
6762:  80400584       4 _ mp_
6763:  80400594       4 _ mp_
6764:  80400524       4 _ mp_
6765:  8040059C       4 _ mp_
6766:  80400534       4 _ mp_
6767:  80400590       4 _ mp_
6768:  804004EC       4 _ mp_
6769:  80400554       4 _ mp_
6770:  80400598       4 _ mp_
6771:  80400520       4 _ mp__HalAllocateAdapterChannel
6772:  804004CC       4 _ mp__HalAllocateCommonBuffer
6773:  804004E8       4 _ mp__HalAllProcessorsStarted
```

译注：

在 Windows XP SP2 上，我测试的结果是 ntoskrnl.exe 并没有导出以双下划线为前缀的符号，ntdll.dll 倒是导出了一些。（2005-5-16）

如示例 1-6、1-7 所示的问题，在 Windows XP SP2 上并未发现。

上面两个例子所示的问题可能会让你失去耐心，你可能会喊道：“Hey，我还是以自己的方式来干吧！”，但存在的问题是几乎没有文档记录过有关微软符号文件格式的内部信息，而某些符号信息----特别是 PDB 文件的结构—则一点文档也没有。甚至在微软基本知识库中居然包含如下内容：

“Program Database 文件的格式（即.PDB 文件的格式）没有提供相应的文档。这些信息是微软私有的。”（微软 2000d）

这么看来开发自己的符号信息分析器似乎必定要失败了。别担心，我将告诉你 PDB 文件的结构到底是什么样的。但在开始之前，我们需要深入研究一下.dbg 文件，因为它是我们整个故事的开始。

1.3.2、.dbg 文件的内部结构

Windows NT 4.0 组件的符号化信息均保存在扩展名为.dbg 的文件中。如果假设符号文件所在的根目录为：d:\winnt\symbols，则组件 filename.ext 的符号文件的全路径就是：d:\winnt\symbols\filename.dbg。例如，内核符号可以在文件 d:\winnt\symbols\exe\ntoskrnl.dbg 中找到。Windows 2000 也使用.dbg 文件。不过，在 Windows 2000 下符号化的信息被移动到了独立的.pdb 文件中。因此，每个 Windows 2000 组件在符号文件根目录下都有一个相关的 ext\filename.dbg 和一个附加的 ext\filename.pdb 文件，除此之外，Windows NT 4.0 和 Windows 2000 的.dbg 文件的内容仍是相同的。

幸运的是，有关.dbg 文件的内部信息的文档还是有一小部分的。Win32 SDK 头文件 winnt.h 提供了核心的常量和类型定义，在 MSDN 中也有一些有关.dbg 文件格式的很有帮助的文章。其中最具有启发性的是 Matt Pietrek 在 1999 年 3 月的 MSJ（Microsoft Systems Journal，MSJ，现在该命位 MSDN Magazine）的“Under the Hood”专栏中发表的文章。基本上，一个.dbg 文件包含一个文件头和数据节。这两部分的大小并不固定，并且将来可能会进一步划分。文件头中包含四个主要的分段（subsections）：

1. 一个 IMAGE_SEPARATE_DEBUG_HEADER 结构，该结构以两个标志性字符“DI”开始。（见**列表 1-13**）
2. 一个 IMAGE_SECTION_HEADER 类型的数组，该数组中的每个结构都位于对应组件的 PE 文件中。该数组的大小由 IMAGE_SEPARATE_DEBUG_HEADER 的 NumberOfSections 成员指定。

3. 一组以零结尾的ANSI字符串（每个ANSI字符串占8个字节），这些字符串均是导出符号的解码格式（undecorated form）。

IMAGE_SEPARATE_DEBUG_HEADER的ExportedNameSize 成员指出了一共有多少个字符串。如果模块没有导出任何符号，ExportedNameSize 将为 0，并且该分段也将不存在。

4. 一个 IMAGE_DEBUG_DIRECTORY 类型的数组，这些结构用来描述随后部分的格式以及它们的位置。IMAGE_SEPARATE_DEBUG_HEADER 的 DebugDirectorySize 成员给出了该数组的大小。

```
#define IMAGE_SEPARATE_DEBUG_SIGNATURE 0x4944 // "DI"
typedef struct _IMAGE_SEPARATE_DEBUG_HEADER
{
    WORD Signature;
    WORD Flags ;
    WORD Machine;
    WORD Characteristics;
    DWORD TimeDateStamp;
    DWORD Checksum;
    DWORD ImageBase;
    DWORD SizeOf Image;
    DWORD NurnberOf Sections ;
    DWORD ExportedNamesSize ;
    DWORD DebugDirectorySize;
    DWORD SectionAlignment;
    DWORD Reserved [2 ];
}
IMAGE_SEPARATE_DEBUG_HEADER, *PIMAGE_SEPARATE_DEBUG_HEADER;
//------------------------------------------------------------
#define IMAGE_SIZEOF_SHORT_NAME 8
typedef struct _IMAGE_SECTION_HEADER
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME] ;
    union
```

```
    (
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
        DWORD VirtualAddress;
        DWORD SizeOf RawData;
        DWORD PointerToRawData;
        DWORD PointerToRelocations;
        DWORD PointerToLinenumbers ;
        WORD NumberOf Relocations;
        WORD NumberOf Linenumbers ;
        DWORD Characteristics;
    }
IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
#define IMAGE_DEBUG_TYPE_UNKNOWN            0
#define IMAGE_DEBUG_TYPE_COFF               1
#define IMAGE_DEBUG_TYPE_CODEVIEW           2
#define IMAGE_DEBUG_TYPE_FPO                3
#define IMAGE_DEBUG_TYPE_MISC               4
#define IMAGE_DEBUG_TYPE_EXCEPTION          5
#define IMAGE_DEBUG_TYPE_FIXUP              6
#define IMAGE_DEBUG_TYPE_OMAP_TO_SRC        7
#define IMAGE_DEBUG_TYPE_OMAP_FROM_SRC 8
#define IMAGE_DEBUG_TYPE_BORLAND            9
#define IMAGE_DEBUG_TYPE_RESERVED10         10
#define IMAGE_DEBUG_TYPE_CLSID              11
//-------------------------------------------------------
typedef struct _IMAGE_DEBUG_DIRECTORY
{
```

```
    DWORD Characteristics;
    DWORD TimeDateStamp;
    WORD MajorVersion;
    WORD MinorVersion;
    DWORD Type;
    DWORD SizeOfData;
    DWORD AddressOfRawData;
    DWORD PointerToRawData;
}
IMAGE_DEBUG_DIRECTORY , * PIMAGE_DEBUG_DIRECTORY ;
```

列表 1-13. .dbg 文件的文件头结构

由于文件头中的表头分段的大小不能确定，因此它们在.dbg 文件中的绝对位置必须通过它们前面的分段的大小来计算出来。一个.dbg 文件分析器通常采用如下算法：

- IMAGE_SEPARATE_DEBUG_HEADER 结构总是位于文件的开始位置。
- 第一个 IMAGE_SECTION_HEADER 结构紧随 IMAGE_SEPARATE_DEBUG_HEADER 结构之后，因此总是可以在文件偏移量为 0x30 的位置找到该结构。
- 将 IMAGE_SECTION_HEADER 结构的大小与该结构的个数相乘然后加上第一个 IMAGE_SECTION_HEADER 结构在文件中的偏移量就可得到第一个导出符号的偏移量。即第一个导出字符串的位置是：0x30+(NumberOfSections*0x28)。
- 通过将 ExportedNameSize 与导出符号分段的偏移量相加即可得到第一个 IMAGE_DEBUG_DIRECTORY 结构的位置。
- 通过 IMAGE_DEBUG_DIRECTORY 项可确定.dbg 文件中剩余数据项的偏移量。PointerToRaw 和 SizeOfData 成员分别指出了相关数据块的偏移量和大小。

列表 1-13 给出了 IMAGE_DEBUG_TYPE_*结构的定义。这些结构反映了.dbg 文件中所包含的多种数据格式。不过，Windows NT 4.0 的符号文件通常仅包含这些结构中的四个：IMAGE_DEBUG_TYPE_COFF、IMAGE_DEBUG_TYPE_CODEVIEW、

IMAGE_DEBUG_TYPE_FPO 和 IMAGE_DEBUG_TYPE_MISC。Windows 2000 的.dbg 文件通常会增加 IMAGE_DEBUG_TYPE_OMAP_TO_SRC、IMAGE_DEBUG_TYPE_OMAP_FROM_SRC 以及一个未文档化的类型 ID 为 0x1000 的结构。如果你仅对解析或浏览符号感兴趣，那么你只需要了解目录项结构：IMAGE_DEBUG_TYPE_CODEVIEW、IMAGE_DEBUG_TYPE_OMAP_TO_SRC 和 IMAGE_DEBUG_TYPE_OMAP_FROM_SRC。

本书的 CD 中包含一个示例 DLL----w2k_img.dll，该 DLL 用于解析.dbg 和.pdb 文件并导出了多个用于开发内核调试工具的重要函数。可在本书的\src\w2k_img 目录下找到该 DLL 的源代码。w2k_img.dll 的一个重要属性是：它所有 Win32 平台上都可以运行。这不只包括 Windows 2000、Windows NT 4.0 还包括 Windows 9x。像所有 Win32 世界中的好市民一样，这个 DLL 的每个函数为支持 ANSI 和 Unicode 字符串均提供了独立的接口。默认情况下，客户端使用 ANSI 版的函数。如果应用程序的源文件中包含了#define UICODE，那么将选择 Unicode 版的函数。运行于 Win32 平台上的程序最好选择 ANSI 版的函数。针对 Windows NT/2000 开发的程序则可选择 Unicode 版的函数以获取更好的性能。

在本书 CD 中还包含一个名为“SBS Windows 2000 CodeView Decompiler”的示例程序，可在 CD 的\src\w2k_cv 目录下找到该程序的 Visual C/C++项目文件。该程序是一个简单的用于分析.dbg 和.pdb 文件，并在 Windows 控制台中显示它们的内容。在阅读本节时，你可以使用该程序来查看我们正在讨论的这些数据结构。w2k_cv.exe 大量使用了 w2k_img.dll 中的 API 函数。

列表 1-14 给出了 w2k_img.h 中定义的一个最基本的数据结构---IMG_DBG，该结构是由.dbg 文件头中的前两个分段串联而成，也就是说，该结构由一个大小固定的基本表头和一组 PE 节的表头构成。给定 PE 节表头的数目就可通过 IMG_DBG__()宏计算出该结构的实际大小。这一大小还确定了导出符号节（exported-names subsections）在文件中的偏移量。

W2k_img.dll 中有几个函数需要一个指向已初始化的 IMG_DBG 结构的指针。imgDbgLoad()函数可分配一个 IMG_DBG 结构，并对该结构进行适当的初始化（该函数会用指定的.dbg 文件的内容填充该结构）。imgDbgLoad()会对数据进行严格的完整性检查以确定指定的.dbg 文件是有效和完整的。imgDbgLoad()函数返回的 IMG_DBG 结构可传递给多个分析函数，通过这些分析函数我们可得到一些经常使用的线性地址。例如，imgDbgExports()函数可计算出导出符号节（该节紧随 IMAGE_SECTION_HEADER 数组之后）的线性地址。

该函数还可通过扫描整个导出符号节来统计有效符号名的个数，并可通过 pdcount 参数来返回统计的结果（可选）。

```
typedef struct _IMG_DBG
{
    IMAGE_SEPARATE_DEBUG_HEADER    Header;
    IMAGE_SECTION_HEADER                aSections[];
}
IMG_DBG, *PIMG_DBG, **PPIMG_DBG;
#define IMG_DBG_ sizeof(IMG_DBG)
#define IMG_DBG__(_n) (IMG_DBG_+((_n)*IMAGE_SECTION_HEADER_))
#define IMG_DBG_DATA(_p,_d) \
        ((PVOID)((PBYTE)(_p) + (_d)->PointerToRawData))
```

列表 1-14. IMG_DBG 结构以及相关的宏定义

```
PVOID WINAPI imgDbgLoadA (PBYTE    pbPath,
                          PDWORD pdSize)
    {
    DWORD dOffset = (pdSize != NULL ? *pdSize : 0);
    DWORD dSize     = dOffset;
    PBYTE pbData    = imgFileLoadA (pbPath, &dSize);
    if ((pbData != NULL) &&
        (!imgDbgVerify ((PIMG_DBG) (pbData + dOffset), dSize)))
        {
        pbData = imgMemoryDestroy (pbData);
        }
    if (pdSize != NULL) *pdSize = dSize;
```

```
    return pbData;
    }
// ------------------------------------------------------------
PVOID WINAPI imgDbgLoadW (PWORD    pwPath,
                          PDWORD pdSize)
    {
    DWORD dOffset = (pdSize != NULL ? *pdSize : 0);
    DWORD dSize   = dOffset;
    PBYTE pbData  = imgFileLoadW (pwPath, &dSize);
    if ((pbData != NULL) &&
        (!imgDbgVerify ((PIMG_DBG) (pbData + dOffset), dSize)))
        {
        pbData = imgMemoryDestroy (pbData);
        }
    if (pdSize != NULL) *pdSize = dSize;
    return pbData;
    }
// ------------------------------------------------------------
PBYTE WINAPI imgDbgExports (PIMG_DBG pid,
                            PDWORD   pdCount)
    {
    DWORD i, j;
    DWORD dCount     = 0;
    PBYTE pbExports = NULL;
    if (pid != NULL)
        {
        pbExports = (PBYTE) pid->aSections
                  + (pid->Header.NumberOfSections
                     * IMAGE_SECTION_HEADER_);
```

```
        for (i = 0; i < pid->Header.ExportedNamesSize; i = j)
            {
            if (!pbExports [j = i]) break;
            while ((j < pid->Header.ExportedNamesSize) &&
                    pbExports [j++]);
            if ((j > i) && (!pbExports [j-1])) dCount++;
            }
        }
    if (pdCount != NULL) *pdCount = dCount;
    return pbExports;
    }
```

列表 1-15. imgDbgLoad()和 imgDbgExports()函数

列表 1-16 定义了两个可根据 ID（这里的 ID 形如：IMAGE_DEBGU_TYPE_*）来定位相应的调试目录项（debug directory entry）的 API 函数。imgDbgDirectories()返回 IMAGE_DEBUG_DIRECTORY 数组的基地址，imgDbgDirectory()返回指向给定 ID 所对应的第一个目录项的指针，如果不存在这样的目录项，则返回 NULL。

```
PIMAGE_DEBUG_DIRECTORY WINAPI imgDbgDirectories (PIMG_DBG pid,
                                                 PDWORD     pdCount)
    {
    DWORD                       dCount = 0;
    PIMAGE_DEBUG_DIRECTORY pidd     = NULL;
    if (pid != NULL)
        {
        pidd     = (PIMAGE_DEBUG_DIRECTORY)
                   ((PBYTE) pid
                    + IMG_DBG__ (pid->Header.NumberOfSections)
                    + pid->Header.ExportedNamesSize);
        dCount = pid->Header.DebugDirectorySize
                   / IMAGE_DEBUG_DIRECTORY_;
```

```
        }
    if (pdCount != NULL) *pdCount = dCount;
    return pidd;
    }
// ----------------------------------------------------------------
PIMAGE_DEBUG_DIRECTORY WINAPI imgDbgDirectory (PIMG_DBG pid,
                                                DWORD       dType)
    {
    DWORD                       dCount, i;
    PIMAGE_DEBUG_DIRECTORY pidd = NULL;
    if ((pidd = imgDbgDirectories (pid, &dCount)) != NULL)
        {
        for (i = 0; i < dCount; i++, pidd++)
            {
            if (pidd->Type == dType) break;
            }
        if (i == dCount) pidd = NULL;
        }
    return pidd;
    }
```

列表 1-16. imgDbgDirectories()和 imgDbgDirectory() API 函数

imgDbgDirectories()函数可用来查找.dbg 文件中的 CodeView 数据。**列表 1-17** 中的 imgDbgCv()函数完成了这一任务。imgDbgCv()函数使用 IMAGE_DEBUG_TYPE_CODEVIEW 调用 imgDbgDirectories()，并使用 IMG_DBG_DATA()宏将 IMAGE_DEBUG_DIRECTORY 项提供的偏移量转化为绝对线性地址。该宏只是简单的将 IMG_DBG 结构的基地址与给定的偏移量相加然后再将结果转型（typecast）为 PVOID 类型的指针。如果 pdSize 参数不为 NULL，则 imgDbgCv()将 CodeView 子节的大小保存到该参数中。接下来我们将讨论 CodeView 数据的内部结构。

针对其他数据子节（data subsection）的函数非常类似。**列表 1-18** 给出了imgDbgOmapToSrc()和 imgDbgOmapFromSrc()函数以及它们使用的 OMAP_TO_SRC 和 OMAP_FROM_SRC 结构。稍后，我们将使用这些结构来计算位于 CodeView 子节中的符号的线性地址。因为 OMAP 数据结构是一个长度固定的数组，所以这两个 API 函数并不返回子节的大小，而是计算数组中的项数。该项数将被保存到*pdcount 参数中（如果该参数不为 NULL 的话）。

```
PCV_DATA WINAPI imgDbgCv (PIMG_DBG pid,
                                PDWORD     pdSize)
    {
    PIMAGE_DEBUG_DIRECTORY pidd;
    DWORD                  dSize = 0;
    PCV_DATA               pcd     = NULL;
    if ((pidd = imgDbgDirectory (pid, IMAGE_DEBUG_TYPE_CODEVIEW))
        != NULL)
        {
        pcd     = IMG_DBG_DATA (pid, pidd);
        dSize = pidd->SizeOfData;
        }
    if (pdSize != NULL) *pdSize = dSize;
    return pcd;
    }
```

列表 1-17. imgDbgCv()函数

```
typedef struct _OMAP_TO_SRC
    {
    DWORD dTarget;
    DWORD dSource;
    }
    OMAP_TO_SRC, *POMAP_TO_SRC, **PPOMAP_TO_SRC;
#define OMAP_TO_SRC_ sizeof (OMAP_TO_SRC)
```

```
// -------------------------------------------------------------
typedef struct _OMAP_FROM_SRC
    {
    DWORD dSource;
    DWORD dTarget;
    }
    OMAP_FROM_SRC, *POMAP_FROM_SRC, **PPOMAP_FROM_SRC;
#define OMAP_FROM_SRC_ sizeof (OMAP_FROM_SRC)
POMAP_TO_SRC WINAPI imgDbgOmapToSrc (PIMG_DBG pid,
                                     PDWORD    pdCount)
    {
    PIMAGE_DEBUG_DIRECTORY pidd;
    DWORD                  dCount = 0;
    POMAP_TO_SRC           pots   = NULL;
    if ((pidd = imgDbgDirectory (pid,
                                 IMAGE_DEBUG_TYPE_OMAP_TO_SRC))
        != NULL)
        {
        pots   = IMG_DBG_DATA (pid, pidd);
        dCount = pidd->SizeOfData / OMAP_TO_SRC_;
        }
    if (pdCount != NULL) *pdCount = dCount;
    return pots;
    }
// -------------------------------------------------------------
POMAP_FROM_SRC WINAPI imgDbgOmapFromSrc (PIMG_DBG pid,
                                         PDWORD    pdCount)
    {
    PIMAGE_DEBUG_DIRECTORY pidd;
    DWORD                  dCount = 0;
```

```
    POMAP_FROM_SRC              pofs     = NULL;
    if ((pidd = imgDbgDirectory (pid,
                                        IMAGE_DEBUG_TYPE_OMAP_FROM_SRC))
        != NULL)
        {
        pofs     = IMG_DBG_DATA (pid, pidd);
        dCount = pidd->SizeOfData / OMAP_FROM_SRC_;
        }
    if (pdCount != NULL) *pdCount = dCount;
    return pofs;
    }
```

列表 1-18. imgDbgOmapToSrc()和 imgDbgOmapFromSrc()函数

1.3.3、CodeView 子节（CodeView Subsections）

CodeView 是微软自己的调试信息格式。随着微软 C/C++编译器和链接器的发展它也经历了很多变化。有些 CodeView 版本之间的区别非常明显。不过，所有版本的 CodeView 在它们的开始位置都有一个 32 位的签名来唯一标识其采用的数据格式。Windows NT 4.0 符号文件使用 NB09 格式，该格式由 CodeView 4.10 引入。Windows 2000 的符号文件包含的则是 NB10 格式的 CodeView 数据，这一格式只是引入了独立.pdb 文件，这一点我在前面已经提及过。

NB09 版的 CodeView 中的数据被进一步细化为一个目录及其下属的目录项。就像 Matt Pietrek 在 MSJ 所发表的文章中所指出的那样， 对于.dbg 文件，大多数基本的 CodeView 结构在 SDK 的一组示例性头文件中都有定义。如果你安装了 SDK 示例，你会在\Program Files\Microsoft Platform SDK\Samples\SdkTools\Image\Include 目录下发现一组非常有趣的文件。你解析 CodeView 所需的文件名为：cvexefmt.h 和 cvinfo.h。很不幸的是，这些文件已经很长时间没有更新了，这些文件的日期还停留在 09-07-1994。这些文件最引人注意的是定义在 cvexefmt.h 中结构体的名字，这些名字都以 OMF 开始，OMF 表示 Object Module Format。OMF 是 16 位 DOS 和 Windows 的.obj、.lib 文件使用的一种标准文件格式。从微软的 32 位

开发工具开始，这一格式由通用对象文件格式（Common Object File Format，COFF）代替了。

尽管今天看来原始的 OMF 格式已经过时了，但它仍被公认为是一种灵活的文件格式。它的一个目标就是尽可能的减少对内存和磁盘空间的使用量。另一个非常重要的属性是：即使应用程序并不完全了解此种格式的所有部分，也可以成功的解析这个格式的文件。基本的 OMF 数据结构是一种带有标识的记录，开始处的标识字节给出了记录中所包含的数据的类型。这种设计使得 OMF 读取器可以在记录之间灵活的移动，以选出它们感兴趣的记录。微软的 CodeView 格式就采用了这种设计方案，cvexetmt.h 中 CodeView 结构名称的 OMF 前缀可以说明这一点。尽管 CodeView 记录和原始的 OMF 记录一样仅包含了很少的数据，但它仍保留了此种格式的基本特性：不需要理解记录中的所有内容，就可以读取此种格式。

列表 1-19 给出了多个基本的 CodeView 结构，它们都来自 w2k_img.h。其中的一些定义和 cvexefmt.h 和 cvinfo.h 中的定义类似，但它们可以满足 w2k_img.dll 的所有需求。在所有 CodeView 数据中都出现了 CV_HEADER 结构，但格式的版本除外。签名是一个 32 位的格式版本 ID，它类似于 CV_SIGNATURE_NB09 或者 CV_SIGNATURE_NB10。lOffset 成员给出了 CodeView 目录相对于表头地址的偏移量。在 Windows NT 4.0 的 NB09 格式的符号文件中，这一偏移量似乎总是 8，这表示紧随表头之后的就是 CodeView 目录结构。Windows 2000 符号文件中这一偏移量则为 0。在稍后我们将详细讨论这一格式。

```
#define CV_SIGNATURE_NB      'BN'
#define CV_SIGNATURE_NB09 '90BN'
#define CV_SIGNATURE_NB10 '01BN'
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
typedef union _CV_SIGNATURE
      {
      WORD    wMagic;         // 'BN'
      DWORD dVersion;      // 'xxBN'
      BYTE    abText [4]; // "NBxx"
      }
      CV_SIGNATURE, *PCV_SIGNATURE, **PPCV_SIGNATURE;
#define CV_SIGNATURE_ sizeof (CV_SIGNATURE)
```

```
// -------------------------------------------------------------
typedef struct _CV_HEADER
	{
	CV_SIGNATURE Signature;
	LONG		lOffset;
	}
	CV_HEADER, *PCV_HEADER, **PPCV_HEADER;
#define CV_HEADER_ sizeof (CV_HEADER)
// -------------------------------------------------------------
typedef struct _CV_DIRECTORY
	{
	WORD	wSize;		// in bytes, including this member
	WORD	wEntrySize; // in bytes
	DWORD dEntries;
	LONG	lOffset;
	DWORD dFlags;
	}
	CV_DIRECTORY, *PCV_DIRECTORY, **PPCV_DIRECTORY;
#define CV_DIRECTORY_ sizeof (CV_DIRECTORY)
// -------------------------------------------------------------
#define sstModule		0x0120 // CV_MODULE
#define sstGlobalPub	0x012A // CV_PUBSYM
#define sstSegMap		0x012D // SV_SEGMAP
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
typedef struct _CV_ENTRY
	{
	WORD	wSubSectionType;	// sst*
	WORD	wModuleIndex;		// -1 if not applicable
	LONG	lSubSectionOffset; // relative to CV_HEADER
```

```
    DWORD dSubSectionSize;      // in bytes, not including padding
    }
    CV_ENTRY, *PCV_ENTRY, **PPCV_ENTRY;
#define CV_ENTRY_ sizeof (CV_ENTRY)
// -------------------------------------------------------------------
typedef struct _CV_NB09 // CodeView 4.10
    {
    CV_HEADER       Header;
    CV_DIRECTORY Directory;
    CV_ENTRY        Entries [];
    }
    CV_NB09, *PCV_NB09, **PPCV_NB09;
#define CV_NB09_ sizeof (CV_NB09)
// -------------------------------------------------------------------
typedef struct _CV_NB10 // PDB reference
    {
    CV_HEADER       Header;
    DWORD           dSignature;     // seconds since 01-01-1970
    DWORD           dAge;           // 1++
    BYTE            abPdbName []; // zero-terminated
    }
    CV_NB10, *PCV_NB10, **PPCV_NB10;
#define CV_NB10_ sizeof (CV_NB10)
```

列表 1-19. CodeView 的数据结构

NB09 版的 CodeView 目录由一个 CV_DIRECTORY 结构和紧随其后的一个 CV_ENTRY 数组构成。**列表 1-19** 中的 CV_NB09 结构反映了这种结构特点。CV_NB09 结构包含 CodeView 表头、目录和一个 Entry 数组。CV_DIRECTORY 结构的 dEntries 成员给出了 Entries[]数组的大小。数组中的每个 CV_ENTRY 都指向 CodeView 的一个子节，该子节的类型由 CV_ENTRY 结构的 wSubSectionType 成员给出。Cvexefmt.h 中定义了至少 21 种子节类型。

不过，Windows NT 4.0 仅使用其中的 3 个：sstModule（0x0120）、sstGlobalPub（0x012A）和 sstSegMap（0x012D）。通常你会在符号文件发现多个 sstModule 类型的子节，而 sstGlobalPub 和 sstSegMap 类型的子节仅有一个（在一个符号文件中）。就像它们的名字所暗示的，sstGlobalPub 表示在该类型的子节中，我们可以找到对应模块的公开的全局符号信息。

列表 1-20 中给出的 imgCvEntry()函数可以方便的按类型查找 CodeView 目录项。该函数的 pc09 参数指向一个 CV_NB09 结构，这意味着，该参数将指向.dbg 文件中签名为 NB09 的 CodeView 数据块。dType 参数用于指定 CodeView 子节的类型 ID（形如 sst*），dIndex 参数用于表示在多个类型相同的子节中选择哪个子节的实例。因此，仅当 dType 为 sstModule 时，dIndex 才能被设置为一个非 0 值。

```
PCV_ENTRY WINAPI imgCvEntry (PCV_NB09 pc09,
                             DWORD    dType,
                             DWORD    dIndex)
    {
    DWORD       i, j;
    PCV_ENTRY pce = NULL;
    if ((pc09 != NULL) &&
        (pc09->Header.Signature.dVersion == CV_SIGNATURE_NB09))
        {
        for (i = j = 0; i < pc09->Directory.dEntries; i++)
            {
            if ((pc09->Entries [i].wSubSectionType == dType) &&
                (j++ == dIndex))
                {
                pce = pc09->Entries + i;
                break;
                }
            }
        }
    return pce;
```

```
    }
// ------------------------------------------------------------
PCV_PUBSYM WINAPI imgCvSymbols (PCV_NB09 pc09,
                                PDWORD     pdCount,
                                PDWORD     pdSize)
    {
    PCV_ENTRY    pce;
    PCV_PUBSYM pcp1;
    DWORD        i;
    DWORD        dCount = 0;
    DWORD        dSize    = 0;
    PCV_PUBSYM pcp       = NULL;
    if ((pce = imgCvEntry (pc09, sstGlobalPub, 0)) != NULL)
        {
        pcp = CV_PUBSYM_DATA ((PBYTE) pc09
                              + pce->lSubSectionOffset);
        dSize = pce->dSubSectionSize;
        for (i    = 0; dSize - i >= CV_PUBSYM_;
              i += CV_PUBSYM_SIZE (pcp1))
            {
            pcp1 = (PCV_PUBSYM) ((PBYTE) pcp + i);
            if (dSize - i < CV_PUBSYM_SIZE (pcp1)) break;
            if (pcp1->Header.wRecordType == CV_PUB32) dCount++;
            }
        }
    if (pdCount != NULL) *pdCount = dCount;
    if (pdSize    != NULL) *pdSize    = dSize;
    return pcp;
    }
```

列表 1-20. imgCvEntry()和 imgCvSymbols()函数

1.3.4、CodeView 符号

列表 1-20 底部的 imgCvSymbols()函数将返回一个指向首个 CodeView 符号记录的指针。sstGlobalPub 子节包含一个长度确定的 CV_SYMHASH 表头，紧随其后的是一组长度可变的 CV_PUBSYM 记录。**列表 1-21** 给出了这两个结构的定义。首先，imgCvSymbols()调用 imgCvEntry()来找出 CV_ENTRY，该结构的 wSubSectionType 成员将被设置为 sstGlobalPub。如果 imgCvEntry()返回一个 CV_ENTRY 结构，则将使用**列表 1-4** 底部的 CV_PUBSYM_DATA()宏来跳过前导的 CV_SYMHASH 结构。最后，imgCvSymbols()通过遍历 CV_PUBSYM 记录列表来统计符号的个数，并使用 CV_PUBSYM_SIZE()宏（参见**列表 1-21**）计算每个记录的大小。

CV_PUBSYM 列表和 OMF 对象文件的内容有些相似。前面已经提到过，一个 OMF 数据流由长度可变的记录组成，每个记录的开始位置的第一个字节为标志，紧随其后的一个 WORD 存放的是该记录的大小。CV_PUBSYM 记录与之类似。CV_PUBSYM 记录的开始位置是一个 OMF_HEADER 结构，该结构由 wRecordSize 和 wRecordType 成员构成。可看出这 OMF 非常相似，不同之处是标志字节之后的存放长度的 WORD 被扩展为 16 位。CV_PUBSYM 结构的最后一部分是符号名，该符号名采用的是 PASCAL 格式，这一格式是 OMF 记录常用的格式。PASCAL 格式的字符串的第一个字节用来记录该字符串的长度，其后的 8 位用来存储字符。和 C 风格的字符串不同，它并不以 0 表示结束。在符号名结束之后，CV_PUBSYM 还将占有其后的 16 个位，这样做是为了到达 32 位边界。这 16 位由 OMF_HEADER 结构中的 wRecordSize 成员使用。要特别注意的是，wRecordSize 给出的 CV_PUBSYM 结构的大小，并不包括 wRecordSize 自己占用的空间。这也是**列表 1-21** 中的 CV_PUBSYM_SIZE()宏会在 wRecordSize 之上再加上 sizeof(WORD)，以获取整个记录体的大小。

```
typedef struct _CV_SYMHASH
    {
    WORD    wSymbolHashIndex;
    WORD    wAddressHashIndex;
    DWORD dSymbolInfoSize;
```

```
    DWORD dSymbolHashSize;
    DWORD dAddressHashSize;
    }
    CV_SYMHASH, *PCV_SYMHASH, **PPCV_SYMHASH;
#define CV_SYMHASH_ sizeof (CV_SYMHASH)
// --------------------------------------------------------------
typedef struct _OMF_HEADER
    {
    WORD wRecordSize; // in bytes, not including this member
    WORD wRecordType;
    }
    OMF_HEADER, *POMF_HEADER, **PPOMF_HEADER;
#define OMF_HEADER_ sizeof (OMF_HEADER)
// --------------------------------------------------------------
typedef struct _OMF_NAME
    {
    BYTE bLength;        // in bytes, not including this member
    BYTE abName [];
    }
    OMF_NAME, *POMF_NAME, **PPOMF_NAME;
#define OMF_NAME_ sizeof (OMF_NAME)
#define S_PUB32   0x0203
#define S_ALIGN   0x0402
#define CV_PUB32 S_PUB32
// --------------------------------------------------------------
typedef struct _CV_PUBSYM
    {
    OMF_HEADER Header;
    DWORD          dOffset;
```

```
    WORD            wSegment;       // 1-based section index
    WORD            wTypeIndex;     // 0
    OMF_NAME        Name;           // zero-padded to next DWORD
    }
    CV_PUBSYM, *PCV_PUBSYM, **PPCV_PUBSYM;
#define CV_PUBSYM_ sizeof (CV_PUBSYM)
#define CV_PUBSYM_DATA(_p) \
        ((PCV_PUBSYM) ((PBYTE) (_p) + CV_SYMHASH_))
#define CV_PUBSYM_SIZE(_p) \
        ((DWORD) (_p)->Header.wRecordSize + sizeof (WORD))
#define CV_PUBSYM_NEXT(_p) \
        ((PCV_PUBSYM) ((PBYTE) (_p) + CV_PUBSYM_SIZE (_p)))
```

列表 1-21. CV_SYMHASH 和 CV_PUBSYM 结构

如果你要扫描 CV_PUBSYM 流，一般情况下，你会遇到两种类型的记录：S_PUB32(0x0203)或者 S_ALIGN(0x0402)。最后一种可以安全的忽略它，因为它仅仅是为了满足 32 位平台的字节对其方式。S_PUB32 记录中则包含实际的符号信息。除符号名之外，记录中的 wSegment 和 dOffset 成员也非常有用。wSegment 给出了 PE 文件中包含该符号的 Section 的索引。使用该索引的负数可以作为 IMAGE_SECTION_HEADER 数组（此数组位于.dbg 文件的开始位置）的索引。dOffset 是符号名相对于 PE Section 的偏移量。这里的符号名指的是与函数的入口点或者全局变量的基地址相关的符号。通常，将 dOffset 与对应 IMAGE_SECTION_HEADER 结构的虚拟地址相加即可得到该符号相对于模块基地址的偏移量。不过，如果.dbg 文件还包含 IMAGE_DEBUG_TYPE_OMAP_TO_SRC 和 IMAGE_DEBUG_TYPE_OMAP_FROM_SRC 子节，那么 dOffset 必须经过一个附加的转换层。OMAP 表的使用方式将在介绍完 PDB 文件的格式后再来讨论。

符号在 CodeView 的 sstGlobalPub 子节中的顺序有些随机。我还不清楚它的深层含义。不过，我可以肯定地说，符号并不是按照 section 的序号、偏移量或名字的顺序来存放的。不要依赖对符号存放顺序的假设，你必须自己完成符号记录的排序。w2k_img.dll 示例库提供了三个默认的符号顺序：按照地址、按照名字（大小写敏感）和按照名字（忽略大小写）。

1.3.5、 .pdb 文件的内部结构

在安装完 Windows 20000 符号文件之后，你会发现最明显的不同是每个模块都有与其相关的两个符号文件：一个扩展名为.dbg，新增加的那个文件的扩展名为.pdb。粗略察看一下.pdb 文件，会发现在其起始位置存放的是这样一个字符串“Microsoft C/C++ program database 2.00”。可以看出 PDB 是 Program Database 的首字母缩写。在 MSDN 中或 Internet 上搜索一下有关 PDB 内部结构的信息，你会发现没有任何有用的信息，唯一例外的是，在微软的基础知识文章中，微软申明此种格式是它有的（Microsoft Corporation, 2000d）。就连 Windows 的老大 Matt Pietrek 也承认：

“PDB 符号表的格式并没有公开的文档。（就连我也不知道其确切的格式，唯一知道的是，它会随着 Visual C++的更新而更新。）”（Pietrek 1997a）

或许，pdb 格式会随着 Visual C/C++一起更新，不过针对当前版本的 Windows 2000 我可以确切的告诉你 PDB 符号文件的结构。这或许是首次公开的 PDB 格式文档。但首先，还是让我们检查一下.dbg 和.pdb 文件是如何链接到一起的。

Windows 2000 的.dbg 文件的一个显著特性是：它们包含的数据很少，几乎可以忽略它们的 CodeView 子节。**示例 1-8** 给出了 ntsokrnl.exe 的.dbg 文件所包含的整个 CodeView 数据，只有区区 32 字节。这些数据是由 w2k_dump.exe 工具获取的，可在本书 CD 的 \src\w2k_dump 目录下找到该工具的源代码。通常，子节总是以一个 CV_HEADER 结构开始，该结构中包含 CodeView 的版本标识。这一次，该版本标识是 NB10。MSDN（Microsoft 2000a）没能告诉我们有关这个特殊版本的更多信息：

“NB10，可执行文件的这一标识表示，其调试信息保存在独立的 PDB 文件中。相应的格式还有 NB09 或 NB11。”（MSDN Library—April 2000\Specifications\Technologies and Languages\Visual C++ 5.0 Symbolic Debug Information Specification\Debug Information Format）

我并不知道 NB11 格式的内部细节，不过 PDB 格式和前面讨论的 NB09 格式一样几乎什么也没有。第一句话很明确的说明了为什么 NB10 数据块是如此的小。所有相关的信息都被移到了独立的文件中了，因此这个 CodeView 子节的主要作用就是提供指向实际数据的链接。如示**例 1-8** 所暗示的，在 ntoskrnl.pdb 文件中一定可以找到实际的符号信息。

如果你对**示例 1-8** 中剩余数据的作用非常好奇，那么**列表 1-22** 或许可以满足你的好奇心。CV_HEADER 结构是自解释的。其后的两个成员的偏移量分别为：0x8 和 0xC，它们的名字分别为：dSignature 和 dAge，在.dbg 和.pdb 文件链接的过程中它们将扮演重要角色。dSignature 是一个 32 位的 UNIX 风格的时间戳，它保存了调试信息构建的日期和时间（自 01-01-1970 以来逝去的秒数）。w2k_img.dll 提供了两个函数：imgTimeUnpack()和 imgTimePack()用来将 dSignature 和 Windows 风格的时间格式进行相互的转化。我还不是非常清楚 dAge 成员的确切含义。目前知道的是：dAge 成员的初始值为 1，每次修改 PDB 数据后其值就会增一。dSignature 和 dAge 共同构成一个 64 位的 ID，调试器可以使用它来验证给定的 PDB 文件是否与它引用的.dbg 文件相匹配。PDB 文件在它的一个数据流中包含着两个值的一个副本，因此调试器可以拒绝处理不相匹配的.dbg/.pdb 文件。

无论你何时遇到格式未知的数据结构，你应该做的第一件事就是使用十六进制 Dump 浏览器察看这些结构。本书附带的 w2k_dump.exe 可很好的完成这一工作。通过检查 Windows 2000 PDB 文件，如 ntoskrnl.pdb 或 ntfs.pdb，你会发现这些文件拥有如下一些共同特性：

- 这些文件似乎都被划分为多个大小固定的块，一般情况下，每个块的大小为 0x400 字节。
- 某些块包含一长串 1，但偶而会被一小段连续的 0 打断。
- 文件中的信息并不必须是连续的。有时，数据会在块的边界处突然结束，但又会在文件的其它地方继续开始。
- 有些数据块会在文件中反复出现。

```
typedef struct _CV_NB10   // PDB reference

{

    CV_HEADER  Header;

    DWORD      dSignature;   // seconds since 01-01-1970

    DWORD      dAge;         // 1++

    BYTE       abPdbName[];  // zero-terminated

}
```

```
CV_NB10, *PCV_NB10, **PPCV_NB10;

#define CV_NB10_ sizeof(CV_NB10)
```

列表 1-22.　CodeView 的 NB10 子节

最终弄清这些复合文件的典型特点花费了我不少时间。复合文件是将一个小型文件系统打包到一个单一文件中。“文件系统”这一修饰词可很好的解释上面得到的观察结果：

- 一个文件系统会将磁盘细化为大小固定的扇区，一组扇区又构成一个文件（此文件件大小可变）。由扇区构成的文件可位于磁盘的任何位置上，并不要求必须是连续的。文件/扇区的对应关系定义在文件目录中。
- 一个复合文件将一个原始磁盘文件细化为大小固定的页，一组页构成一个流（stream），并且流的大小可变。由页构成的复合文件可位于原始文件中的任何位置，这些页并不必须是连续的。流和页的对应关系定义在流目录中。

很显然，文件系统中的格式和复合文件格式差不多是一一对应的，只需简单的将“扇区”替换为“页”，将“文件”替换为“流（Stream）”。对照文件系统可以很好的解释为什么 PDB 文件是按大小固定的块组织起来的，同时还解释了为什么这些块并不一定都是连续的。不过，一页中几乎都是二进制 1 的块又代表什么呢？实际上，这种类型的数据在文件系统中是很常见的。为了跟踪磁盘上已用和还未使用的扇区，很多文件系统都维护了一个二进制位的分配数组，数组中的每个二进制位对应文件系统中的一个扇区（或一簇扇区）。如果一个扇区未使用，其对应的二进制位就将被设置为 1。当文件系统为文件分配空间时，它就会扫描这个分配位数组，以找出未使用的扇区。在将扇区加入到文件中后，文件系统就将对应得分配位设为 0。复合文件的页和流也采用了相似的处理方式。一长串的二进制 1 代表还未使用的页，二进制 0 表示对应的页已分配给某个流。

现在唯一的问题就是为什么有些数据块会在 PDB 文件中反复出现。同样的事情也出现在磁盘的扇区上。当文件系统中的一个文件被多次重写时，每个写操作可能会使用不同的扇区来存放数据。因此，磁盘上某些空扇区中可能会包含旧数据的副本。这在文件系统中不算是什么问题。如果扇区在分配数组中标识为未使用，那么该扇区上有什么数据就无所谓了。这样的扇区很快就会在另一个文件中被使用，其原有内容将被新的数据覆盖掉。对应文件系统的这一特性，我们再来看复合文件，这意味着我们观察到的那些重复的页应该是修改留下的副本。可以安全地忽略它们；我们唯一需要关心的就是那些在流目录（stream directory）中被引用到的页。

现在已经介绍完了 PDB 文件的基本结构，接下来我们将检查构成 PDB 文件的那些基本的数据块。**列表 1-23** 给出了 PDB 头部的布局。在 PDB_HEADER 的开始位置有一个文件字符串给出了当前 PDB 的版本标识。该标识字符串以 EOF 字符（ASCII 码为 0x1A）结束。在其后还有一个附加的数字：0x0000474A，如果将该数字解释为字符串的话，则为：”JG\0\0”。或许这代表 PDB 格式的最初设计者吧。嵌入的 EOF 字符有一个很好的作用：如果普通用户在控制台窗口中使用 `type ntoskrnl.pdb`，那么将不会显示其后的数据，显示出来的信息只是：Microsoft C/C + + program database 2.00。Windows 2000 所有的符号文件都是 PDB 2.00 版。显然，曾经存在过 PDB 1.00 格式，而且其结构似乎与现在的有很大不同。

```
#define PDB_SIGNATURE_200 \

        "Microsoft C/C++ program database 2.00\r\n\x1AJG\0"

#define PDB_SIGNATURE_TEXT 40

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

typedef struct _PDB_SIGNATURE

    {

    BYTE abSignature [PDB_SIGNATURE_TEXT+4]; // PDB_SIGNATURE_nnn

    }

    PDB_SIGNATURE, *PPDB_SIGNATURE, **PPPDB_SIGNATURE;

#define PDB_SIGNATURE_ sizeof (PDB_SIGNATURE)

// ------------------------------------------------------------

#define PDB_STREAM_FREE -1

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

typedef struct _PDB_STREAM
```

```
    {
    DWORD dStreamSize;   // in bytes, -1 = free stream
    PWORD pwStreamPages; // array of page numbers
    }
    PDB_STREAM, *PPDB_STREAM, **PPPDB_STREAM;
#define PDB_STREAM_ sizeof (PDB_STREAM)
// ------------------------------------------------------------
#define PDB_STREAM_MASK 0x0000FFFF
#define PDB_STREAM_MAX  (PDB_STREAM_MASK+1)
#define PDB_STREAM_DIRECTORY 0
#define PDB_STREAM_PDB       1
#define PDB_STREAM_TPI       2
#define PDB_STREAM_DBI       3
#define PDB_STREAM_PUBSYM    7
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
typedef struct _PDB_ROOT
    {
    WORD       wCount;      // < PDB_STREAM_MAX
    WORD       wReserved;   // 0
    PDB_STREAM aStreams []; // stream #0 reserved for stream table
```

```
    }

    PDB_ROOT, *PPDB_ROOT, **PPPDB_ROOT;

#define PDB_ROOT_ sizeof (PDB_ROOT)

#define PDB_PAGES(_r) \

        ((PWORD) ((PBYTE) (_r) \

                  + PDB_ROOT_ \

                  + ((DWORD) (_r)->wCount * PDB_STREAM_)))

// ------------------------------------------------------------

#define PDB_PAGE_SIZE_1K   0x0400 // bytes per page

#define PDB_PAGE_SIZE_2K   0x0800

#define PDB_PAGE_SIZE_4K   0x1000

#define PDB_PAGE_SHIFT_1K  10     // log2 (PDB_PAGE_SIZE_*)

#define PDB_PAGE_SHIFT_2K  11

#define PDB_PAGE_SHIFT_4K  12

#define PDB_PAGE_COUNT_1K  0xFFFF // page number < PDB_PAGE_COUNT_*

#define PDB_PAGE_COUNT_2K  0xFFFF

#define PDB_PAGE_COUNT_4K  0x7FFF

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

typedef struct _PDB_HEADER

    {
```

```
    PDB_SIGNATURE Signature;      // PDB_SIGNATURE_200

    DWORD         dPageSize;      // 0x0400, 0x0800, 0x1000

    WORD          wStartPage;     // 0x0009, 0x0005, 0x0002

    WORD          wFilePages;     // file size / dPageSize

    PDB_STREAM    RootStream;     // stream directory

    WORD          awRootPages [];  // pages containing PDB_ROOT

    }

    PDB_HEADER, *PPDB_HEADER, **PPPDB_HEADER;

#define PDB_HEADER_ sizeof (PDB_HEADER)
```

列表 1-23.　PDB_HEADER 结构

在标识字符串之后偏移量为 0x2C 处有一个名为 dPageSize 的 DWORD 类型的值，它代表的是复合文件中每个页所占的字节数。合法的值可以是：0x0400（1KB）、0x800（2KB）和 0x1000(4KB)。wFilePages 成员记录了 PDB 文件使用的页的总数。将 wFilePages 与 dPageSize 相乘即可得到该 PDB 文件的大小。wStartPage 是一个从零开始的页码，它指向第一个数据页。该页的字节偏移量可由该页的页码乘以每页的大小得到。通常的值为：页号为 9 的 1KB 页（字节偏移量为 0x2400），页号为 5 的 2KB 页（字节偏移量为 0x2800）或者页号为 2 的 4KB 页（字节偏移量为 0x2000）。在 PDB_HEADER 和第一个数据页之间的空间保留给分配位数组，并总是从第二个页开始。这意味着，如果页大小为 1 或 2KB，则 PDB 文件使用 0x10000（64K）个分配位，每位对应 0x2000 字节（8KB）的页，如果页大小为 4KB，则使用 0x8000（32K）个分配位，每位对应 0x1000 字节（4KB）的页。以此类推，这意味着，在页大小为 1KB 的情况下，PDB 文件可容纳 64MB 数据，在页大小为 2KB 或 4KB 的情况下，PDB 文件可容纳 128MB 数据。

PDB_HEADER 最后的 RootStream 和 wRootPages[]成员记录了 PDB 文件中流目录的位置。就像前面提到的，PDB 文件是由一组长度可变的流构成的，这些流中才包含有实际的数据。流的位置及其内容是由一个单一的流目录管理的。流目录自身也存储在一个流中。我称这个特殊的流为“Root Stream”。Root Stream 中保存着流目录（该流目录可能位于 PDB 文件的任何位置）。PDB_HEADER 的 Rootstream 和 wRootPages[]成员提供了 Root Stream 的位置和大小。PDB_STREAM 子结构的 dStreamSize 成员给出了流目录占用的页的数目，这些页的首地址保存在 wRootPages[]数组中，这些页包含实际的数据

现在让我们用一个小例子来说明这一点。**示例 1-9** 给出了 ntoskrnl.pdb 的 PDB_HEADER 的十六进制 Dump 的部分内容。这里引用到的值由下划线标识出来。显然，这个 PDB 文件使用的页的大小为 0x400 字节（1KB），一共使用了 0x02D1（721）个页，这样该文件的大小则为 0xB4400（十进制 738,304）。使用 dir 命令可验证这个大小是正确的。Root Stream 的大小为 0x5B0 字节（1456 字节），由于每个页的大小为 0x400 字节（1KB），则意味着 wRootPages[]数组中包含两项，分别位于偏移量为 0x3C 和 0x3E 处。数组中的两项内容都是页码，需要将此页码与页大小相乘才能得到对应的字节偏移量。此处，该字节偏移量为：0xB2000 和 0xB2800。

上面最后一行给出的计算结果是 ntoskrnl.pdb 文件的流目录所占用的两组文件页的首地址，其范围分别为：0xB2000----0xB23FF 和 0xB2800----0xB29AF。**示例 1-10** 给出了这些范围的部分内容。

流目录由两个部分构成：一个 PDB_ROOT 结构的文件头部分，该结构定义在列表 1-24 中，另一部分是由 16 位页码构成的数组。PDB_ROOT 结构中的 wCount 成员记录了保存在 PDB 文件中的流的数目。aStream[]数组包含多个 PDB_STREAM 结构（参见**列表 1-23**），每个 PDB_STREAM 结构代表一个流，紧随 aStream[]数组之后在就是页码数组。在**示例 1-10** 中，流的个数为 8，对应的偏移量为 0xB2000，该位置已用下划线标识出。随后的 8 个 PDB_STREAM 结构分别给出了这 8 个流的大小：0x5B0、0x3A、0x38、0x402A9、0x0、0x4004、0x19EB4 和 0x4DF3C。这些值也都以下划线标识出。在 1KB 页模式下，流的大小为：0x2、0x1、0x1、0x101、0x0、0x11、0x68 和 0x138，这样可计算出这些流总共占用了 0x2B6 个页。在 PDB_STREAM 数组之后，第一个以下划线标识出来的值是页码列表中的第一个页码。这里每个页码占用 2 个字节，这里需要考虑的是，页目录被属于其他部分的一个页截断了，故页目录随后的偏移量为应为：0xB2044+0x400+(0x2B6*2)=0xB29B0，**示例 1-10** 很好的展示了这一点。

```
#define PDB_STREAM_DIRECTORY 0

#define PDB_STREAM_PDB       1
```

```
#define PDB_STREAM_TPI       2

#define PDB_STREAM_DBI       3

#define PDB_STREAM_PUBSYM    7

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

typedef struct _PDB_ROOT

    {

    WORD       wCount;      // < PDB_STREAM_MAX

    WORD       wReserved;   // 0

    PDB_STREAM aStreams []; // stream #0 reserved for stream table

    }

    PDB_ROOT, *PPDB_ROOT, **PPPDB_ROOT;

#define PDB_ROOT_ sizeof (PDB_ROOT)

#define PDB_PAGES(_r) \

        ((PWORD) ((PBYTE) (_r) \

                  + PDB_ROOT_ \

                  + ((DWORD) (_r)->wCount * PDB_STREAM_)))
```

列表 1-24. PDB 的流目录结构

要找到给定的流所对应的页码需要一定的技巧，因为页目录除了流的大小之外，没有提供任何信息。如果你对 3 号流感兴趣，那么你必须计算流 1 和流 2 所占用的页的数目，以获取 3 号流在页码数组中的起始索引。一旦定位了指定流的页码列表，读取流中的数据就很简单了。只需要遍历页码列表，将列表中的每个页码和每页的大小相乘，就可获得此页码对应页的文件偏移量，然后从该偏移量处开始读取页的内容，反复如此，直到到达流的结束处，

就可读取整个流的内容了。猛地一看，解析一个 PDB 文件似乎非常费劲。但从另一个角度看却十分简单-----因为这要比解析一个.dbg 文件简单的多。PDB 格式的这种清晰的随机访问机制，将读取一个流的任务简化为读取连续的大小固定的页。这种优雅的数据访问机制让我很是吃惊。

当更新一个已存在的 PDB 文件时，PDB 格式的优势就非常明显了。将使用连续的结构体的数据插入到一个文件中，意味着将移动大量的原有数据。PDB 文件从文件系统借鉴来的随机访问架构允许以最小的开销完成删除或插入数据的操作，就像文件系统中的文件可以很容易的修改一样。当一个流在增大时，只需改动流目录或则收缩页的边界。这种非常重要的特性大为提高了 PDB 文件更新的灵活性。微软在基本知识库中正式提供这样一片文章：“信息：PDB 和 DBG 文件-----它们是什么以及它们是如何工作的”：

“.PDB 扩展了“Program database”架构。此种文件用来存放调式信息，这种格式随 Visual C++ 1.0 一起引入。在将来，.PDB 文件还将包含其它的项目状态信息。格式改变的一个重要动机是为了允许程序调试版的增量链接，第一次改变随 Visual C++ 2.0 引入。”（Microsoft Corporation 2000e）

现在 PDB 文件的内部结构已经很清晰了，下一个问题是如何识别这些流的具体内容。在检查完 PDB 文件的各个方面后，我得出这样一个结论：每一种流都用于特定的目的。第一个流似乎总是包含一个流目录，第二个流包含用于验证该 PDB 文件是否与其关联的.dbg 文件相匹配的信息。例如，该流中包含的 dSignature 和 dAge 成员应该和 NB10 CodeView 节中的对应成员一致，如**列表 1-22** 所示。第八个流是本章最感兴趣一个，因为该流中包含我们要探索的 CodeView 的符号信息。其余流的作用我还不是很清楚，这是将来要研究得一个方向。

我没有提供 PDB 读取程序的示例代码，因为这已超出本章的范围。替代的，我鼓励你深入研究一下本书 CD 中的 w2k_img.c 和 w2k_img.h。重点是 imgPdb*()函数和 PDB_*类型的数据。顺便说一下，本书 CD 中包含一个完整的 PDB 流读取程序的完整源代码。你已经用过这个程序了，它就是 w2k_dump.exe。前面的一些例子就是我用这个工具生成的。这个简单的控制台程序使用+p 命令选项来允许解析 PDB 流。如果指定的文件不是一个有效的 PDB 文件，那么该程序将仅进行连续的十六进制 Dump。可在本书 CD 的\src\w2k_dump 目录下找到 w2k_dump.exe 的 Visual C/C++项目文件。

二、The Windows 2000 Native API

本章对于 Windows 2000 Native API 的讨论，主要集中在这些 API 和系统模块之间的关系，将重点介绍 Windows 2000 采用的中断机制。Windows 2000 利用此机制将对内核服务的请求从用户模式向内核模式传递。另外，Win32K 接口和一些与 Native API 相关的主要运行时库也会被提及，同时还将介绍一些经常使用的数据类型。

有关 Windows 2000 架构的详细讨论已经很多。许多有关 Windows NT 的讨论同样适用于 Windows 2000。《Inside Windows NT》（Custer 1993, Solomon 1998）的第一、二版都是有关此方面的好书，同样的还有《Inside Windows 2000》（Solomon and Russinovich 2000）。

2.1、NT*()和 Zw*()函数集

有关 Windows 2000 架构的一个有趣的事实是：它模拟了多个操作系统。Windows 2000 内置三个子系统来支持 Win32、POSIX 和 OS/2 应用程序。Win32 子系统是最流行的一个，因此它更多的被开发人员和操作系统所关照。在 Windows 9x 中，Win32 接口实际上是作为整个系统的基础结构来实现的。但是，Windows 2000 的设计却有很大不同。尽管 Win32 子系统包含一个名为 kernel32.dll 的系统模块，但这并不是实际的操作系统内核。它仅仅是 Win32 子系统的一个基本组件。在很多编程书籍中，Windows NT/2000 的软件开发被简化为与 Win32 API 打交道的工作，NT 平台暴露出的一个隐藏的事实是存在另一个更为基础的调用接口：Native API。相信编写 kernel-mode driver 或 file system driver 的开发人员已经对 Native API 非常熟悉了，因为 kernel-mode 模块位于更低的系统层，在那里子系统是不可见的。然而，你并不需要到驱动程序一层才能访问此接口----即使一个普通的 Win32 应用程序也可在任何时候调用 Native API。这没什么技术上的限制----仅仅是微软不支持此种应用程序开发模式而已。因此，有关此话题的信息并不是很多，neither SDK nor the DDK make the Native API available to Win32 Application.

2.1.1、未文档化的级别

本书中的多数东西都来自被称为未文档化的信息。这通常意味着微软没有公开发布这些信息。然而，未文档化也存在几个级别，这是因为可能被公布的有关庞大的操作系统（如 Windows 2000）的信息非常的多。我个人的系统分类如下：

- **正式文档**：这些信息来自微软出版的书、文件或者开发工具。大多数重要信息来自 SDK、DDK 和 MSDN。
- **半文档化的（Semidocumented）**：尽管不是正式文档，但这些信息还是可以从微软正式发布的文件中挖掘出来的。例如，Windows 2000 的很多函数和结构体并没有在 SDK 或 DDK 文档中提到，但出现在一些头文件或示列程序中。以 Windows 2000 为例，很多重要的半文档化信息都源自头文件 ntddk.h 和 ntdef.h，这两个文件都是 DDK 的一部分。
- **未文档化，但并没有隐藏**：这些信息不能在任何官方文档和开发文档中找到，但其中的一部分对调试工具是可用的。可执行文件或符号文件中的所有符号化信息都属于这一部分。最好的例子是内核调试器的!processfields 和!threadfields 命令，这两个命令会给出两个未文档化的结构：EPROCESS 和 ETHREAD 的成员名称及其偏移量。
- **完全未文档化的**：微软很好的隐藏了某些信息，要获得它们只能通过逆向工程和推理。此类信息包含很多实现细节的信息，没有人认为 Windows 2000 开发人员需要关注它们，但是这些信息对于系统开发人员和开发调试软件的人来说却非常宝贵。挖掘系统内部的信息是非常困难的，但同样是非常有趣的。

本书讨论的 Windows 2000 的内部细节覆盖了上述系统分类的后三个。

2.1.2、系统服务分配器（System Service Dispatcher）

Win32 子系统和 Native API 之间的关系可以由 Win32 核心模块与 Windows 2000 内核模块之间的依赖关系很好的解释。图 2-1 展示了模块间的依赖关系，方框表示系统模块，箭头表示模块间的依赖关系。如果一个箭头从模块 A 指向模块 B，这表示 A 依赖于 B，即，模块 A 调用 B 中的函数。模块由双向箭头连接，表示二者之间相互依赖。在图 2-1 中，模块：

user32.dll、advapi32.dll、gdi32.dll、rpcrt4.dll 以及 kernel32.dll 实现了基本的 Win32 API。当然，还有其他的 DLL（如 version.dll、shell32.dll 和 comctl32.dll）也为 Win32 API 提供支持，为了更清晰些，我省略了它们。**图 2-1** 表现出的一个特性非常有趣，所有的 Win32 API 调用最后都转移到了 ntdll.dll，而 ntdll.dll 又将其转移到了 ntoskrnl.exe。

Ntdll.dll 是一个操作系统组件，它为 Native API 准确地提供服务，ntdll.dll 是 Native API 在用户模式下的前端。Native API 真正的接口在 ntoskrnl.exe 中实现。从其文件名可以猜出它就是 NT 操作系统内核。事实上，内核模式驱动程序对系统服务的请求多数时候都会进入该模块。Ntdll.dll 的主要任务就是为运行于用户模式的程序提供一个确定的内核函数的子集，这其中就包括 Win32 子系统 DLLs。在**图 2-1** 中，从 ntdll.dll 指向 ntoskrnl.exe 的箭头旁标注的 INT 2eh 表示 Windows 2000 使用此中断将 CPU 特权级从用户模式切换到内核模式。开发内核（kernel-mode）模式程序的人员认为用户模式的代码是具有攻击性的、充满错误的和危险的。因此，必须让这些代码远离内核函数。而通过在调用 API 的过程中将特权级别从用户模式切换到内核模式是一种可控制这些问题的方式。调用程序从来不可能触及内核，它只能察看它们。

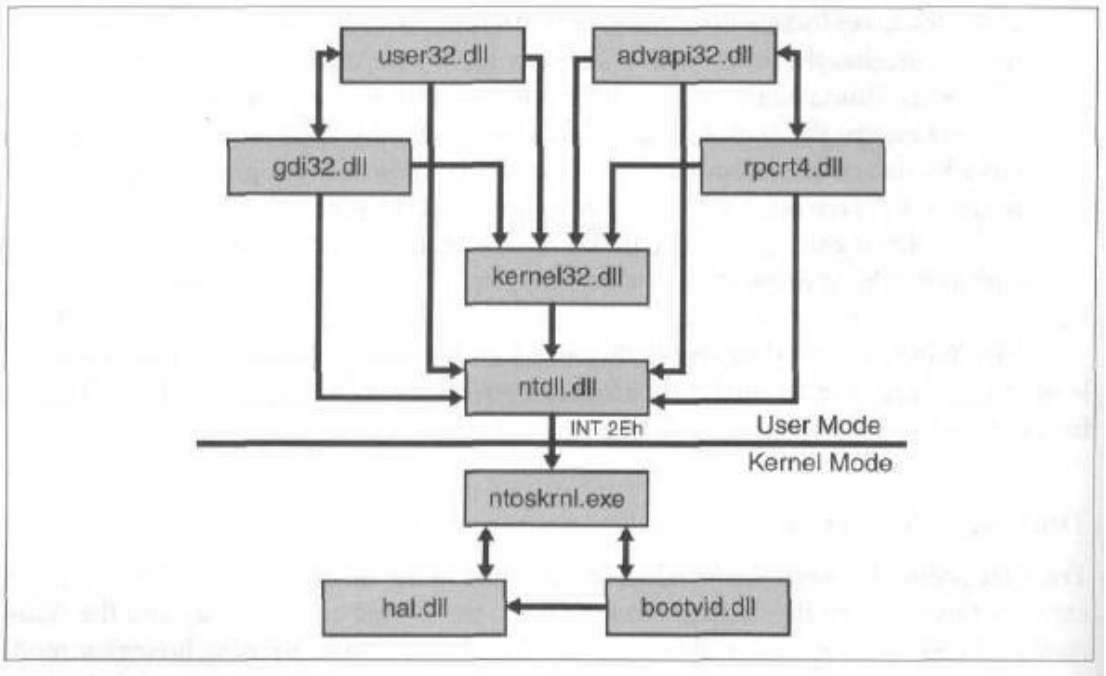


FIGURE 2-1. *System Module Dependencies*

例如，由 kernel32.dll 导出的 Win32 API 函数 DeviceIoControl()最终会调用由 ntdll.dll 导出的 NtDeviceIoControlFile()。通过反编译该函数会发现此函数令人惊讶的实现方式—

它是如此的简单！**示列 2-1** 展示了这些。首先，CPU 寄存器 EAX 被装入了一个“魔术”数字 0x38，这是一个分派 ID。接下来，寄存器 EDX 被设置指向堆栈中的某处，其地址为堆栈指针 ESP 加上 4，因此，EDX 将指向堆栈中返回地址的后面，该返回地址在进入 NtDeviceIoControlFile()时将被立即保存下来。显而易见，EDX 指向的位置是用来临时存放传递进来的参数的。接下来的指令是一个简单的 INT 2eh，该指令将跳转到中断描述符表（Interrupt Descriptor Table,IDT）的 0x2e 位置上存放的中断处理例程（interrupt handler）中。这看上去是不是很熟悉？事实上，这有些像 DOS 下的 INT 21h 调用。然而，Windows 2000 的 INT 2eh 接口要远比一个简单的 API 调用有用，分配器（dispatcher）利用它从用户模式进入内核模式。请注意，这种模式切换方式是 x86 处理器特有的。在 Alpha 平台上，有不同的方式来实现此种功能。

```
NtDeviceIoControlFile:
        mov    eax, 38h
        lea    edx, [esp+4]
        int    2Eh
        ret    28h
```

示列 2-1. ntdll.NtDeviceIoControlFile()的实现方式

Windows 2000 Native API 由 248 个函数组成，这些函数都采用上述方式进入内核。与 Windows NT 4.0 相比多出了 37 个。你很容易在 ntdll.dll 的导出列表中通过 Nt 前缀来认出它们。Ntdll.dll 总共导出了 249 个这样的符号。多出的那个函数是 NtCurrentTeb()，该函数是一个纯粹的用户模式函数，它无需进入内核。**附录 B** 中的**表 B-1** 列出了所有可用的 Native API。该表同时还指出那个函数是由 ntoskrnl.exe 导出的。令人奇怪的是，在处于内核模式的模块中，只能调用 Native API 的一个子集。另一方面，ntoskrnl.exe 导出了两个 ntdll.dll 没有提供的 Nt*符号(指以 Nt 开头的符号)：NtBuildNumber 和 NtGlobalFlag。这两个符号都没有指向函数的入口地址，而是指向 ntoskrnl.exe 中的变量。驱动模块（driver module）可以使用 C 编译器的 extern 关键字来导入这些变量。Window 2000 采用此种方式导出了很多变量，稍后我将给出一个示例代码来使用其中的几个。

你可能会奇怪为什么**表 B-1**（位于附录 B 中）分别为 ntdll.dll 和 ntoskrnl.exe 提供了两列，其名称分别为：ntdll.Nt*、ntdll.Zw*和 ntoskrnl.Nt*、ntoskrnl.Zw*。原因是，

这两个模块导出了两组相互关联的 Native API 符号。在**表 B-1**（位于附录 B 中）的最左列给出了所有名字中包含 Nt 前缀的符号。另一个集合包含相似的名字，不过由 Zw 前缀代替了 Nt。反编译 ndll.dll 可看出每对符号都指向相同的代码。这看起来似乎是浪费内存。然而，如果你反编译 ntoskrnl.exe，你就会发现 Nt*符号指向实际的代码而 Zw*指向 INT 2eh stubs（如**示列 2-1** 列出的）。这意味着 Zw*函数集合将从用户模式转入内核模式，而 Nt*符号直接指向的代码会在模式切换后被执行。

表 B-1（位于附录 B 中）中有两件事需要特别注意。首先，NtCurrentTeb()函数没有对应的 Zw*函数。这不是什么大问题，因为 ntdll.dll 以相似的方式导出 Nt*和 Zw*函数。其次，ntoskrnl.exe 不再一贯的成对的导出 Nt/Zw 函数。其中的一些仅以 Nt*或 Zw*的形式出现。我不知道为什么会这样，我猜测 ntoskrnl.exe 仅导出了在 Windows 2000 DDK 中有文档记录的函数以及其它系统模块必须的那些函数。注意，保留的 Native API 函数仍然实现于 ntoskrnl.exe 的内部。这些函数并没有公开的进入点，但可通过 INT 2eh 到达他们。

2.1.3、服务描述符表（The Service Descriptor Tables）

从**示例 2-1** 给出的反编译代码可看出，INT 2eh 随同传入 CPU 寄存器 EAX 和 EDX 的两个参数一起被调用。我已经提到过 EAX 中的“魔术”数字是一个分派 ID。除 NtCurrentTeb()之外的所有 Native API 都采用此种方式，处理 INT 2eh 的代码必须确定每个调用将被分配到那个函数。这就是提供分派 ID 的原因。位于 ntoskrnl.exe 中的中断处理例程将 EAX 中的数值作为一个索引来查询一个特定的表。这个表被称作系统服务表（System Service Table, SST）该表对应的 C 结构体---SYSTEM_SERVICE_TABLE 的定义在**列表 2-1** 中给出。在该列表中还包含 SERVICE_DESCRIPTOR_TABLE 结构的定义，该结构共有四个 SST 类型的数组，其中的前两个用于特定目的。

尽管上述的两个表是系统基本的数据类型，但他们在 Windows 2000 DDK 中并没有相应的文档记载，本书中出现的许多代码片断都包含未文档化的数据类型和函数。因此，不能保证这些信息是完全真实可信的。所有符号化的信息，如结构名称、结构成员和参数都是如此。在创建这些符号时，我试图使用适当的名称，这些名称基于从已知符号的一个很小的子集（包括从符号文件中得到的那些）中得出的命名方案。然而，在很多场合这种启发式方法并不成功。只有在原始的代码中包含所有的信息，但我无法得到它们。实际上，我并不打算阅读这

些源代码，因为这需要和微软签订一个 NDA（Non-Disclosure Agreement,，不可泄漏协议），由于该 NDA 的限制，将很难写出一本有关非文档化信息的书。

```
typedef NTSTATUS (NTAPI*NTPROC)();
typedef NTPROC* PNTPROC;
#define NTPROC_ sizeof(NTPROC)

typedef struct _SYSTEM_SERVICE_TABLE
{
    PNTPROC  ServiceTable;   // array of entry points
    PDOWRD  CounterTable;   // array of usage counters
    DWORD   ServiceLimit;    // number of table entries
    PBYTE    ArgumentTable;  // array of byte counts
}
SYSTEM_SERVICE_TABLE,
*PSYSTEM_SERVICE_TABLE,
**PPSYSTEM_SERVICE_TABLE;

//-------------------------------------------------------------------------
-----------------------------

typedef struct _SERVICE_DESCRIPTOR_TABLE
{
    SYSTEM_SERVICE_TABLE ntoskrnl;  // ntoskrnl.exe ( native api )
    SYSTEM_SERVICE_TABLE win32k;    // win32k.sys (gdi/user support)
    SYSTEM_SERVICE_TABLE Table3;    // not used
    SYSTEM_SERVICE_TABLE Table4;    // not used
}
SYSTEM_DESCRIPTOR_TABLE,
*PSYSTEM_DESCRIPTOR_TABLE,
```

```
**PPSYSTEM_DESCRIPTOR_TABLE;
```

列表 2-1 系统服务描述符表的结构定义

现在，回到 SDT（Service Descriptor Table）的秘密上来。从**列表 2-1** 给出的该结构的定义可看出该结构的头两个数组保留给了 ntoskrnl.exe 和 Win32 子系统（位于 win32k.sys）中的内核模式（kernel-mode）部分。来自 gdi32.dll 和 user32.dll 的调用都通过 Win32k 的系统服务表（SST）进行分派。Ntolkrnl.exe 导出了一个指针（符号为 KeServiceDescriptorTable）指向其主服务描述符表（Main SDT）。内核还维护了一个替代的 SDT，其名称为：KeServiceDescriptorTableShadow，但这个 SDT 并没有被导出。从处于内核模式的模块中访问主服务描述符表（SDT）非常容易，你只需要两个 C 指令，如**列表 2-2** 所示。首先是由 extern 关键字修饰的变量说明，这告诉链接器该变量并不包含在此模块中，而且不需要在链接时解析相应的符号名称。当该模块被加载到进程的地址空间后，针对该符号的引用才会动态连接到相应的模块中。**列表 2-2** 中第二个 C 指令就是这样的一个引用。将类型为 PSERVER_DESCRIPTOR_TABLE 的变量赋值为 KeServiceDescriptorTable 时，就会和 ntoskrnl.exe 建立一个动态连接。这很像调用一个 DLL 中的 API 函数。

```
// Import SDT pointer
extern PSERVICE_DESCRIPTOR_TABLE KeServiceDescriptorTable;


// Create SDT reference
PSERVICE_DESCRIPTOR_TABLE psdt = KeServiceDescriptorTable;
```

列表 2-2　访问系统服务描述符表

SDT 中的每个 SST 的 ServiceTable 成员都是一个指针，指向一个由函数指针构成的数组，此函数指针的类型为：NTPROC，这为 Native API 提供了占位符，这种方式和在 Win32 编程中使用的 PROC 类型很相似。NTPROC 的定义在前面的**列表 2-1** 中给出。Native API 函数通常返回一个 NTSTATUS 类型的代码并且使用 NTAPI 调用方式，NTAPI 实际上就是_stdcall。ServiceLimit 成员保存在 ServieTable 数组中发现的入口地址的个数。在 Windows 2000 中，其默认值为 248。ArgumentTable 成员是一个 BTYE 类型的数组，它和 ServiceTable 所指的数组一一对应，并给出其中每个函数指针所需的参数在调用者的堆栈中的字节数。此信息随 EDX 寄存器提供的指针一起使用。当内核从调用者的堆栈中复制参数到自己的堆栈时就需要这些信息。CounterTable 成员在 Windows 2000 的 Free Build 版中不被使用。在 Debug Build

版中，该成员指向一个 DWORD 类型的数组，作为每个函数的使用计数器（usage counters）。This information can be used for profiling purposes.

使用 Windows 2000 的内核调试器可方便的显示 SDT 中的内容。如果你还没有设置好这个有用的程序，那请参考第一章。在**示列 2-2** 中，我首次使用了 dd KeServiceDescriptorTable 命令。调试器会将此公开符号解析为 0x8046AB80，同时显示该地址之后的 32 个 DWORD 的 16 进制转储。不过仅有前面的四行才是有意义的，它们分别对应**列表 2-1** 中的四个 SDT 成员。为了更清晰些，它们都将以黑体显示。如果你仔细观察，你会发现第五行与第一行十分相像，这是另一个 SDT 吗？这是测试内核调试器的 ln 命令的好机会。在示列 2-2 中，在显示完 KeServiceDescriptorTable 的十六进制 dump 之后，我输入 ln 8046abc0 命令。显然，调试器知道地址 0x8046abc0，它将此地址转化为对应的符号 KeServiceDescriptorTableShadow 可以看出，这是内核维护的第二个 SDT。二者之间的显著区别是：第二个 SDT 包含 Win32k.sys 的入口地址。这两个表的的第三和第四个成员都是空的。Ntoskrnl.exe 提供了一个函数 KeAddSystemServiceTabel()来填充这两个成员。

注意，我截断了 ln 命令的输出信息，仅保留了基本的信息。

从地址 0x8046ab88 开始，是 KeServiceDescriptorTable 的十六进制转储，在那儿可以找到 ServiceLimit 成员，可看到其值为 0xF8（十进制 248），这和我们预期的一样。ServiceTable 和 ArgumentTable 的值分别指向地址 0x804704d8 和 0x804708bc。用 ln 命令察看着两个地址，可得到其符号：KiServiceTable 和 KiArgumentTable。这两个符号都没有从 ntoskrnl.exe 中导出，但是调试器可通过察看 Windows 2000 的符号文件识别它们。ln 命令还可应用到 Win32k SST 指针上，针对其 ServiceTable 和 ArgumentTable 成员，调试器分别给出了其对应的符号 w32pServiceTable 和 W32pArgumenTable。这两个符号都来自 Win32k.sys 的符号文件。如果调试器无法解析这些地址，可使用**.reload** 命令强制重新加载所有可用符号文件，然后再进行解析。

示例 2-2 的剩余部分是 KiServiceTable 和 KiArgumentTable 最前面的 128 个字节的十六进制转储。到目前为止，如果我说的有关 Native API 的东西都是正确的，那么 NtClose()函数的地址应位于 KiServiceTable 数组的第 24 个位置上，其地址为 0x80470538。在该地址处，可发现其值为 0x8044c422，在 dd KiServiceTable 的输出中，该地址以黑体标记。用 ln 察看 0x8044c422，会看到其对应的符号正是 NtClose()。

```
kd> dd KeServiceDescriptorTable
```

```
8046ab80  804704d8 00000000 000000f8 804708bc
8046ab90  00000000 00000000 00000000 00000000
8046aba0  00000000 00000000 00000000 00000000
8046abb0  00000000 00000000 00000000 00000000
8046abc0  804704d8 00000000 000000f8 804708bc
8046abd0  a0186bc0 00000000 0000027f a0187840
8046abe0  00000000 00000000 00000000 00000000
8046abf0  00000000 00000000 00000000 00000000

kd> ln 8046abc0
(8046abc0)   nt!KeServiceDescriptorTableShadow

kd> ln 804704d8
(804704d8)   nt!KiServiceTable

kd> ln 804708bc
(804708bc)   nt!KiArgumentTable

kd> ln a0186bc0
(a0186bc0)   win32k!W32pServiceTable

kd> ln a0187840
(a0187840)   win32k!W32pArgumentTable

kd> dd KiServiceTable
804704d8  804ab3bf 804ae86b 804bdef3 8050b034
804704e8  804c11f4 80459214 8050c2ff 8050c33f
804704f8  804b581c 80508874 8049860a 804fc7e2
80470508  804955f7 8049c8a6 80448472 804a8d50
```

```
80470518  804b6bfb 804f0cef 804fcb95 8040189a
80470528  804d06cb 80418f66 804f69d4 8049e0cc
80470538  8044c422 80496f58 804ab849 804aa9da
80470548  80465250 804f4bd5 8049bc80 804ca7a5

kd> db KiArgumentTable
804708bc  18 20 2c 2c 40 2c 40 44-0c 18 18 08 04 04 0c 10  . ,,@,@D........
804708cc  18 08 08 0c 08 08 04 04-04 0c 04 20 08 0c 14 0c  ........... ....
804708dc  2c 10 0c 1c 20 10 38 10-14 20 24 1c 14 10 20 10  ,... .8.. $... .
804708ec  34 14 08 04 04 04 0c 08-28 04 1c 18 18 18 08 18  4.......(.......
804708fc  0c 08 0c 04 10 00 0c 10-28 08 08 10 00 1c 04 08  ........(.......
8047090c  0c 04 10 00 08 04 08 0c-28 10 04 0c 0c 28 24 28  ........(....($(
8047091c  30 0c 0c 0c 18 0c 0c 0c-0c 30 10 0c 0c 0c 0c 10  0........0......
8047092c  10 0c 0c 14 0c 14 18 14-08 14 08 08 04 2c 1c 24  .............,.$

kd> ln 8044c422
(8044c422)   nt!NtClose
```

示例 2-2　检查服务描述符表

译注：

在 Windows XP 中，KeServiceDescriptorTable 和 KeServiceDescriptorTableShadow 和 Windows 2000 有所区别。在 XP 中，后者位于前者的前面，而在 W2K 中，后者位于前者的后面。

2.1.4、INT 2eh 系统服务处理例程（System Service Handler）

隐藏在内核模式中的 INT 2eh 中断处理例程为 KiSystemService()。再强调一次，这是一个内部符号，ntoskrnl.exe 并没有导出该符号，不过，它却包含在 Windows 2000 的符号

文件中。因此，内核调试器可以正确的解析该符号。从本质上来看，KiSystemService()将执行如下操作：

1.从当前线程的控制块（thread’s control block）中检索 SDT 指针。

2.通过测试 EAX 寄存器中的分派 ID 的第 12、14 位来确定使用 SDT 中的那个 SST（SDT 中有四个 SST)。如果分派 ID 位于 0x0000-0x0FFF，将选择 ntoskrnl 表；位于 0x1000-0x1FFF 则选择 Win32k 表。0x2000-0x2FFF 和 0x3000-0x3FFF 由 SDT 的 Table3 和 Table4 保留。如果分配 ID 超过了 0x3FFF，在分派前多余的位将被屏蔽掉。

3.通过检查分派 ID 的 0 到 11 位来确定该 ID 在所选 SST 中对应的 ServiceLimit 成员。如果 ID 超出了范围，将返回错误代码：STATUS_INVALID_SYSTEM_SERVICE。在一个未使用的 SST 中，ServiceLimit 成员始终是 0，从而为所有可能的分派 ID 产生一个错误代码。

4.通过检查 EDX 中保存的参数堆栈指针，来取得 MmUserProbeAddress 的值。这是由 ntoskrnl.exe 导出的一个公开变量。参数指针通常会与 0x7FFF0000 进行比较。如果没有低于该地址，那么将返回 STATUS_ACCESS_VIOLATION。

5.根据在 SST 的 ArgumentTable 中查找到的参数堆栈的字节数，将所有函数参数从调用者堆栈中复制到当前的内核堆栈中。

6.在从服务调用(Service Call)中返回后，将控制权传递给内部函数 KiServiceExit()

非常有趣的是 INT 2eh 中断处理例程并不使用全局 SDT（即 KeServiceDescriptorTable），而是使用线程专属的指针替代之。显然，每个线程可以拥有不同的 SDT。在线程初始化时，KeInitializeThread()会将 KeServiceDescriptorTable 的指针写入线程控制块（Thread Control Block）中。不过，此默认值在稍后可能会改变，如改为指向 KeServiceDescriptorTableShadow。

2.2、Win32 内核模式接口（Win32 Kernel-mode Interface）

从前面对 SDT 的讨论，可看出存在着与 Native API 相关的第二个主内核模式接口(main Kernel-mode Interface）。该接口将 Win32 子系统的图形设备接口（Graphics Device Interface, GDI)、窗口管理器(即 User 模块)连接至内核组件---Win32K(即 Win32k.sys).，

该组件随同 Windows NT 4.0 引入。引入该组件是为了克服 Win32 图形引擎固有的性能限制（由于 Windows NT 子系统的最初设计导致）。在 Windows NT 3.x 中，Win32 子系统采用的是客户-服务器模式（Client-Server model），这样就必须从用户模式切换到内核模式才能进行内核调用（Kernel Involved）。通过将图形引擎的绝大部分移至内核组件---Win32k.sys，从而避免了大部分因内核切换导致的性能损失。

2.2.1、Win32K 分派 ID（Win32K Dispatch IDs）

现在该介绍 Win32k.sys 了，也是该更新**图 2-1** 的时候了。**图 2-2** 基于**图 2-1**，但在 ntoskrnl.exe 左面加入了 Win32k.sys。同时我还加入了从 GDI32.DLL 和 USER32.DLL 指向 Win32k.sys 的箭头。当然，这不是百分之百正确，因为这些模块中的 INT 2eh 调用实际上指向 ntoskrnl.exe，在 ntoskrnl.exe 中才有该中断的处理例程。然而，调用最后还是由 Win32k.sys 管理，这也是箭头这样指的原因。

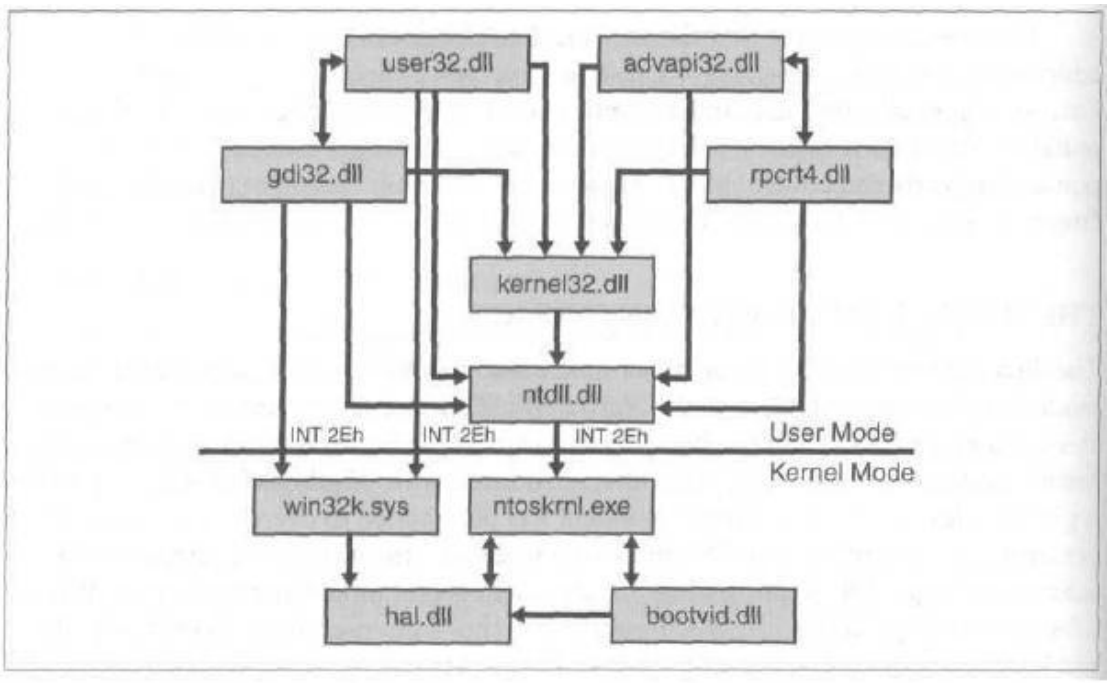


FIGURE 2-2. *System Module Dependencies, including* win32k.sys

稍早提到过，Win32K 接口同样基于 INT 2eh 分派器（INT 2eh Dispatcher），这与 Native API 非常相似。仅有的区别在于 Win32K 使用另一区段的分派 ID。尽管与所有 Native API 调用相关的分派 ID 都位于 0x0000----0x0FFF，而 Win32K 分派 ID 位于 0x1000---0x1FFF 之

间。如**图 2-2** 所示，Win32K 的主要客户端是 GDI32.DLL 和 USER32.DLL。因此，通过反编译这些模块（指 gdi32.dll 和 user32.dll）可能会找到与 Win32K 分派 ID 相关的符号化名称。通过反编译可发现在这些模块（gdi32.dll 和 user32.dll）的导出节（export sections）中仅包含 INT 2eh 调用的一个很小的子集，看来是时候再次使用内核调试器了。如**示例 2-3** 所示，我通过使用 dd W32pServiceTable 命令，来确定 Win32k.sys 的符号是可用的，在此之前请先使用.reload 命令以加载所有可用符号文件。

```
kd> .reload
.reload
Loading Kernel Symbols...
Unable to read image header for fdc.sys at f0798000 - status c0000001
Unable to read image header for ATMFD.DLL at beaaf000 - status c0000001
Loading User Symbols...
Unable to read selector for PCR for Processor 0
PPEB is NULL (Addr= 0000018c)

kd> dd W32pServiceTable
dd W32pServiceTable
a01859f0  a01077f0 a011f59e a000788a a01141e1
a0185a00  a0121264 a0107e05 a01084df a010520b
a0185a10  a0120a6f a008c9eb a00befa2 a007cb5c
a0185a20  a0085c9b a001e4e7 a0120fd1 a0122d19
a0185a30  a0085d0c a0122e73 a0027671 a006d1f0
a0185a40  a0043fe0 a009baeb a007eb9b a009eb05
a0185a50  a0043392 a007c14f a01229cc a0027470
a0185a60  a001ad09 a00af751 a004e9f5 a004ef53

kd> ln a01077f0
ln a01077f0
(a00b316e)   win32k!NtGdiAbortDoc   |  (a00ba173)     win32k!IsRectEmpty
```

EXAMPLE 2-3. *Examination of the Win32K System Services*

在**示例 2-3** 的最后三行中，我使用 ln 命令显示与 W32pServiceTable 的第一个入口地址相关的符号。显然，可看到分派 ID 为 0 的 Win32K 函数为 NtGdiAbortDoc()。你可以针对所有 639 个 ID 来重复此过程，但是最好能自动进行符号的查找。现在，我已经为你完成了这项工作，所有分派 ID 对应的符号名称都收录在**附录 B** 的**表 B-2** 中。符号从 gdi32.dll 和 user32.dll 映射到 win32k.sys 十分简单：GDI 符号可通过在其前面添加 NtGdi 前缀就可转换为 Win32K 符号，USER 符号则添加 NtUser 前缀。然而，有一少部分例外。例如，如果一个 GDI 符号以 Gdi 开始，那么其前缀就减少为 Nt，这可能是为了避免出现 NtGdiGdi 这样的字符序列。在其他的一些例子中，字符的大小写会有些不同（比如 EnableEUDC()转化后则变成了 NtGdiEnableEudc()），或者用符号名称尾部的 W 来表示没有对应的 Unicode 函数（如，CopyAcceleratorTableW()转化后成为 NtUserCopyAcceleratorTable()）。

提供 Win32K API 的详细文档需要很大的努力。这些函数几乎是 Native API 的三倍。或许某天有人会为这些 API 编写一本不错的参考手册，就像 Gary Nebbett 编写的 Native API 手册。不过，在本书范围内，有关这些 API 的信息已经足够了。

2.3、Windows 2000 运行时库

Nt*()和 Zw*()函数构成了 Native API 的基本部分，但并不是主要部分，还有一部分代码位于 ntdll.dll 中。该 DLL 至少导出了 1179 个符号。其中的 249 和 248 个分别属于 Nt*()和 Zw*()函数集，剩余的 682 个函数并不通过 INT 2eh 中断进行调用。显然，这一大组函数并不依赖 Windows 2000 内核。那提供它们的目的何在呢？让我们继续往下看。

2.3.1、C 运行时库

如果你研究过位于 ntdll.dll 导出节（export section）的符号，你会发现很多在 C 程序员看来很熟悉的小写的函数名称。这些都是众所周知的名子，如 memcpy()、sprintf()和 qsort()，这些 C 运行时库中的函数都合并到了 ntdll.dll 中。对于 ntoskrnl.exe 也是如此，它同样提供了一组与 C 运行时函数十分相像的函数，虽然这两组函数并不相同。**附录 B 的表 B-3** 列出了这两组函数，并指出了每个函数分别属于哪个模块。

你可以简单的将 ntdll.lib（来自 Windows 2000 DDK）添加到导入库列表（链接器在解析符号期间将扫描该列表）中，就可以链接到这些函数。如果你更喜欢对话框，你可以选择 Visual C/C++的工程菜单中的 Settings 子菜单,然后单击 Linke 页,选择 Category General，然后将 ntdll.dll 添加到 Object/Library 模块列表中。还有一种方法：在源文件中，添加如下的内容：

```
#pragma comment(linker,”/defaultlib:ntdll.lib”)
```

这同样有效，好处是，其他开发人员可以使用 Visual C/C++的默认设置来 rebuild 你的工程。

反编译这些与 C 运行时函数类似的函数（来自 ntdll.dll 和 ntoskrnl.exe），会发现 ntdll.dll 并不依赖于 ntoskrnl.exe，这和 ndll.dll 中的 Native API 不一样。事实上，这两个模块分别实现了这些函数。本节出现的其他函数也是如此。注意，表 B-3 中的一些函数并不使用其导出的名称。例如，如果在内核模式的驱动程序中针对一个 64 位的

LARGE_INTEGER 使用移位操作符<<和>>，编译器和链接器会自动导入 ntoskrnl.exe 的 _allshr()和_allshl()。

2.3.2、扩展的运行时函数

随同标准的 C 运行时函数，Windows 2000 还提供了一组扩展的运行时函数。在次强调，ntdll.dll 和 ntoskrnl.exe 分别实现了它们。并且其中有些函数是重叠的。这些扩展函数的名字都有一个共同的前缀 Rtl（for Runtime Library）。**附录 B** 的**表 B-4** 列出了所有这些扩展函数。Windows 2000 提供的这些运行时函数还包含用于普通任务的助手函数（helper function），这些任务都超过了 C 运行时函数的能力范围。例如，其中的某些用于管理安全性，另一些用于操作 Windows 2000 特有的数据结构，还有一些对内存管理提供支持。很难理解为什么微软仅在 Windows 2000 DDK 中提供了其中 115 个函数的文档，而扔掉了其余 406 个非常有用的函数。

2.3.3、浮点模拟器（The Floating-Point Emulator）

让我用 ntdll.dll 提供的另一组函数集合来结束这次 API 函数汇展。**表 2-1** 列出了这些函数的名称，这些名称可能对于汇编程序员有些眼熟。去了名称前的__e 前缀，你就会得到 i386 系列 CPU 中的 FPU（Floating-Point Unit）汇编助记符。事实上，从**表 2-1** 中列出的函数来看，ntdll.dll 包含了一个完整的浮点模拟器。这再次证明了这个 DLL 是一个庞大的代码仓库，这吸引了众多的 System Spelunker 去反编译它。

表 2-1. ntdll.dll 的浮点模拟器接口

函数名称			
_eCommonExceptions	_eFIST32	_eFLD64	_eFSTP32
_eEnulatorInit	_eFISTP16	_eFLD80	_eFSTp64
_eF2XM1	_eFISTP32	_eFLDCW	_eFSTP80
_eFABS	_eFISTP64	_eFLDENV	_eFSTSW
_eFADD32	_eFISUB16	_eFLDL2E	_eFSUB32
_eFADD64	_eFISUB32	_eFLDLN2	_eFSUB64

_eFADDPreg	_eFISUBR16	_eFLDPI	_eFSUBPreg
_eFADDreg	_eFISUBR32	_eFLDZ	_eFSUBR32
_eFADDtop	_eFLDI	_eFMUL32	_eFSUBR64
_eFCHS	_eFIDIVR16	_eFMUL64	_eFSUBreg
_eFCOM	_eFIDIVR32	_eFMULPreg	_eFSUBRPreg
_eFCOM32	_eFILD16	_eFMULreg	_eFSUBRreg
_eCOM64	_eFILD32	_eFMULtop	_eFSUBRtop
_eFCOMP	_eFILD64	_eFPATAN	_eFSUBtop
_eFCOMP32	_eFIMUL16	_eFPREm	_eFTST
_eFCOMP64	_eFIMUL32	_eFPREM1	_eFUCOM
_eFCOMPP	_eFINCSTP	_eFPTAN	_eFUCOMP
_eFCOS	_eFINIT	_eFRNDINT	_eFUCOMPP
_eFDECSTP	_eFIST16	_eFRSTOR	_eFXAM
_eFIDIVR16	_eFIST32	_eFSAVE	_eFXCH
_eFIDIVR32	_eFISTP16	_eFSCALE	_eFXTRACT
_eFILD16	_eFISTP32	_eFSIN	_eFYL2X
_eFILD32	_eFISTP64	_eFSQRT	_eFYL2XP1
_eFILD64	_eFISUB16	_eFST	_eGetStatusWord
_eFIMUL16	_eFISUB32	_eFST32	NPXEMULATORTABLE
_eFIMUL32	_eFISUBR16	_eFST64	RestoreEm87Context
_eFINCSTP	_eFISUBR32	_eFSTCW	SaveEm87Context
_eFINIT	_eFLD16	_eFSTENV	
_eFIST16	_eFLD32	_eFSTP	

有关浮点指令集的更多信息，请参考Intel 80386 CPU的原始文档。可以从Intel官方网站：http://developer.intel.com/design/pentium/manuals/来下载PDF格式的Pentium手册。讲解这些机器码指令集的手册是：*Intel Architecture SoftWare Developer’s Manual. Volume 2: Instruction Set Reference*（Intel 1999b）。

2.3.4、其它的 API 函数

除**附录 B** 和**表 2-1** 列出的函数外，ntdll.dll 和 ntoskrnl.exe 还为多个内核组件导出了为数众多的函数。为了避免更长的表格，我这里仅列出可用函数的名称前缀及其所属类别（**表 2-2**）。

表 2-2　函数名前缀及其所属分类

前缀	ntdll.dll	ntoskrnl.exe	分类
_e		N/A	浮点模拟器
Cc		N/A	Cache 管理器
Csr			Client-Server 运行时库
Dbg	N/A		调试支持
Ex	N/A		执行支持（Executive Support）
FsRtl	N/A		文件系统运行时库
Hal	N/A		硬件抽象层调度器
Inbv	N/A		系统初始化/VGA 启动驱动（bootvid.dll）
Init	N/A		系统初始化
Interlocked	N/A		处理线程安全的变量
Io	N/A		I/O 管理器
Kd	N/A		内核调试支持
Ke	N/A		内核例程
Ki			内核中断例程
Ldr			映像加载器
Lpc	N/A		本地过程调用（LPC）设备
Lsa	N/A		本地安全授权
Mm	N/A		内存管理器
Nls			National Language Support (NLS)
Nt			NT Native API

Ob	N/A		对象管理器
Pfx			前缀处理
Po	N/A		电源管理器
Ps	N/A		进程支持
READ_REGISTER_	N/A		从寄存器地址中读取
Rtl			Windows 2000 运行时库
Se	N/A		安全处理
WRITE_REGISTER_	N/A		向寄存器地址中写入
Zw			另一组 Native API
<other>			帮助函数和 C 运行时库

很多内核函数都使用统一的命名规则----PrefixOperationObject()。例如，NtQueryInformationFile()函数属于 Native API，这是因为其 Nt 前缀，而且该函数显然针对一个文件对象执行了 QueryInformation 操作。但并不是所有函数都遵循这一规则，不过绝大多数都是如此。因此，可以很容易的通过函数的名称猜测其功能。

2.4、经常使用的数据类型

当编写与 Windows 2000 内核有关的软件时---不管是和用户模式的 ntdll.dll 还是和内核模式的 ntoskrnl.exe，你都必须处理几个基本的数据类型，而这些数据类型在 Win32 世界里非常少见。它们中的多数都会在本书中反复出现。下面的章节将介绍使用频率最高的数据类型。

2.4.1、整型

一般说来，整数类型有多个不同的变体。Win32 SDK 的头文件和 SDK 文档使用了其专有的术语，这些术语很容易和 C/C++的基本类型以及一些派生类型相混淆。**表 2-3** 列出了这些整数类型，以及它们之间的等价关系。在“MASM”列中，给出了微软宏汇编语言（MASM）使用的类型名称。Win32 SDK 为 C/C++的基本数据类型定义了对应的 BYTE、WORD、DWORD 别名。“别名 1”和“别名 2”两列包含其经常使用的别名。例如，WCHAR 代表基础的 Unicode 字

符类型。最后一列“有符号的”，列出了对应的有符号类型的常见别名。一定要记住 ANSI 字符类型 CHAR 是有符号的，而 Unicode 类型 WCHAR 是无符号的。当编译器将表达式或计算中的这些类型转换为整数类型时，这种不一致性将导致意外的错误。

表 2-3 最后一行的 MASM 的 TBYTE 类型（读做“10-byte”）是一个 80 位的浮点数，用于高精度的浮点运算操作。Microsoft Visual C/C++没有为 Win32 程序员提供对应的数据类型。需要注意的是，MASM 的 TBYTE 和 Win32 的 TBYTE（读做“text byte”）没有任何关系，后者只是一个用于转换的宏，根据源文件中是否有`#define UNICODE`而分别对应 CHAR 或 WCHAR。

表 2-3.　等价的整数类型

位数	MASM	基本类型	别名 1	别名 2	有符号的
8	BTYE	unsigned char	UCHAR		CHAR
16	WORD	unsigned short	USHORT	WCHAR	SHORT
32	DWORD	unsigned long	ULONG		LONG
32	DWORD	unsigned int	UINT		INT
64	QWORD	unsigned __int64	ULONGLONG	DWORDLONG	LONGLONG
80	TBYTE	N/A			

由于在 32 位编程环境中较难处理 64 位整数，Windows 2000 通常不提供 64 位的基本类型，如__int64 或其派生类型。替代的，DDK 头文件 ntdef.h 中定义了一个精巧的 union 结构，可以将一个 64 位数解释为一对 32 位数或一个完整的 64 位数，参见**列表 2-3** 给出了 LARGE_INTEGER 和 ULARGE_INTEGER 类型定义。该类型可分别表示有符号和无符号的整数。通过使用 LONGLONG/ULONGLONG（针对 64 位的 QuadPart 成员）或者 LONG/ULONG（针对 32 位的 HighPart 成员）来控制有无符号。

```
typedef union _LARGE_INTEGER
{
    struct
    {
        ULONG LowPart;
        LONG  HighPart;
    }
```

```
    LONGLONG QuadPart;
} LARGE_INTEGER,*PLARGE_INTEGER;


typedef union _ULARGE_INTEGER
{
    struct
    {
        ULONG LowPart;
        ULONG HighPat;
    }
    ULONGLONG QuadPat;
} ULARGE_INTEGER,*PULARGE_INTEGER;
```

列表 2-3.　LARGE_INTEGER 和 ULARGE_INTEGER

2.4.2、字符串

在 Win32 程序设计中，常使用 PSTR 和 PWSTR 来分别代替 ANSI 和 Unicode 字符串。PSTR 被定义为 CHAR*，PWSTR 则定义为 WCHAR*（参见表 2-3）。通过源文件中是否出现#define UNICODE 指示符，附加的 PTSTR 类型分别对应 PSTR 或 PWSTR，这样就可通过单一的源文件来维护应用程序的 ANSI 和 Unicode 版本。基本上,这些字符串都是简单的指向以零结尾的 CHAR 或 WCHAR 类型的数组。如果你常和 Windows 2000 内核打交道，你将必须处理一种很不同的字符串表示法。最常见的类型是 UNICODE_STRING，这是一个第三方类型，**列表 2-4** 给出了它的定义。

```
typedef struct _UNICODE_STRING
{
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
} UNICODE_STRING,*PUNICODE_STRING;
```

```
typedef struct _STRING
{
    USHORT Length;
    USHORT MaximumLength;
    PCHAR  Buffer;
} STRING, *PSTRING;

typedef STRING ANSI_STRING, *PANSI_STRING;
typedef STRING OEM_STRING, *POEM_STRING;
```

列表 2-4.　字符串类型

Length 成员给出了当前字符串的字节数（注意，不是字符个数），MaximumLength 成员指出 Buffer 所指向内存块的大小，实际的字符串数据将保存在该内存块中。注意，MaximumLength 也是字节数。由于 Unicode 字符宽度为 16 位，所有其长度总是字符个数的两倍。通常，Buffer 指向的字符串都是以零结尾的。然而，有些内核模块可能仅依赖字符串的长度值，而不考虑结尾的 0 字符，这种情况下要小心处理。

Windows 2000 的 ANSI 字符串叫做 STRING，如**列表 2-4** 中所示。为了方便，nedef.h 分别定义了 ANSI_STRING 和 OEM_STRING 来代表使用不同代码页的 8 位字符串（ANSI 默认代码页为 1252；OEM 默认代码页为 437）。不过，Windows 2000 内核使用的主要字符串类型还是 UNICODE_STRING。你可能偶尔会碰到 8 位字符串。

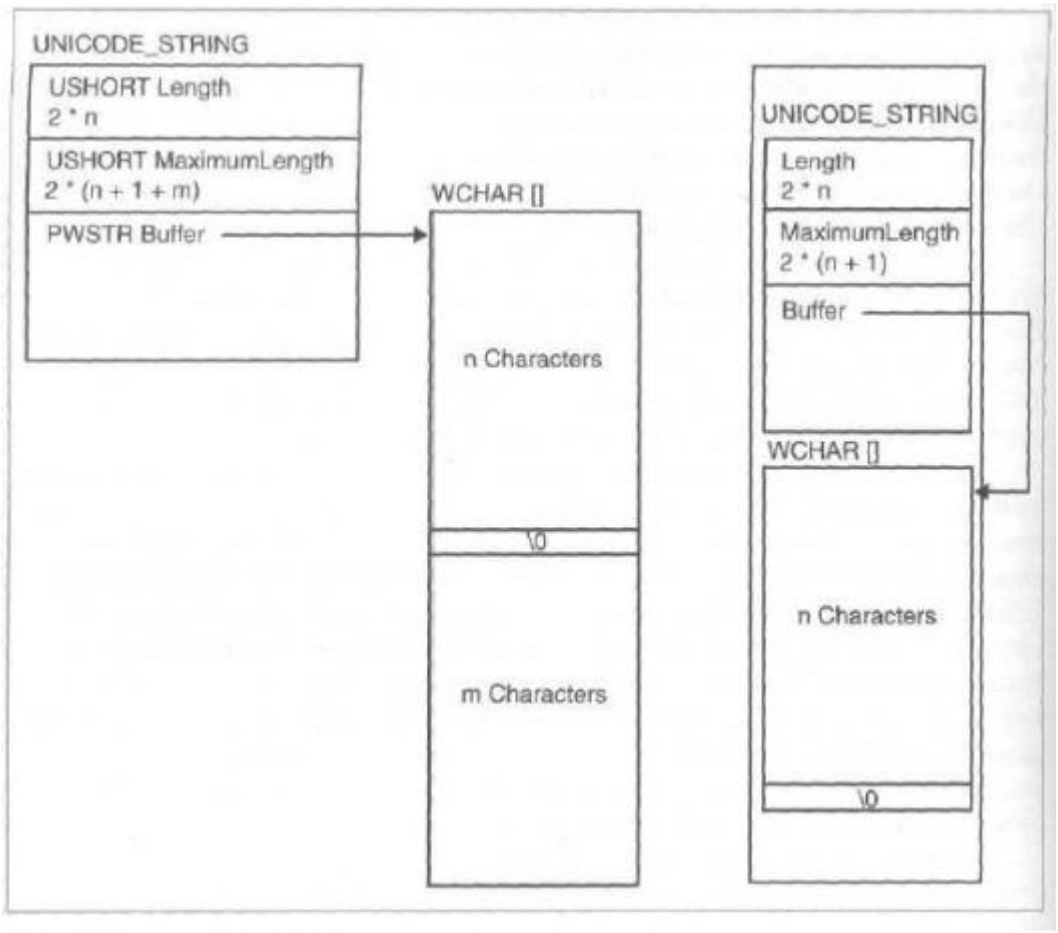


FIGURE 2-3. *Examples of* UNICODE_STRINGs

在图 2-3 中，我给出了两个典型的 UNICODE_STRING 示例。左面的那个包含两个独立的内存块：一个 UNICODE_STRING 结构和一个 16 位 PWCHAR 类型的 Unicode 字符数组。这或许是在 Windows 2000 数据类型中最常见的字符串类型。右边的是一种频繁出现的特殊类型，在此种类型中，UNICODE_STRING 和 PWCHAR 数组位于同一个内存块中。有些内核函数，包括 Native API 内部使用的一些函数，都在连续的内存块中保存其返回的结构化的系统信息。如果数据中包含字符串，它们通常都存储在嵌入式的 UNICODE_STRING 中，如图 2-3 右面所示。例如，NtQuerySystemInformation()函数就频繁使用了这种特殊的字符串类型。

这些字符串结构不许要手工维护，ntdll.dll 和 ntoskrnl.exe 导出了一组丰富的运行时 API 函数，如 RtlCreateUnicodeString()、RtlInitUnicodeString()、RtlCopyUnicodeString()等。通常，STRING 和 ANSI_STRING 也有对应的等价函数。这些函数中的大多数在 DDK 中都有文档记录，但其中有些没有。不过，很容易猜出这些未文档化的字符串函数的功能及其需要的参数。使用 UNICODE_STRING、STRING 的好处是，可以隐示的指定 Buffer 可容纳的字符串的大小。如果你给一个函数传递了一个 UNICODE_STRING 类型的

字符串，而该函数需要适当改变该字符串的值，而这可能会增加该字符串的长度，那这个函数只需要简单的检查 MaximumLength 成员就可确定是否有足够的空间来存放结果。

2.4.3、结构体

个别的几个内核 API 函数期望其处理的对象有一个合适的 OBJECT_ATTRIBUTES 结构，**列表 2-5** 给出了该结构的定义。例如，NtOpenFile()函数没有 PWSTR 或 PUNICODE_STRING 参数用来指定要打开的文件的路径。替代的，OBJECT_ATTRIBUTES 结构中的 ObjectName 成员给出了该路径。通常，设置该结构很容易。除 ObjectName 外，还需要设置 Length 和 Attributes 成员。Length 必须设置为：sizeof(OBJECT_ATTRIBUTES)，Attributes 是一组来自 ntdef.h 的 OBJ_*常量。例如，如果你对象名称不区分大小写的话，Attributes 应设置为 OBJ_CASE_INSENSITIVE。当然，ObjectName 成员是一个 UNICODE_STRING 指针，并不是通常的 PWSTR。剩余的成员只要不使用，都可设置为 NULL。

```
typedef struct  _OBJECT_ATTRIBUTES
{
    ULONG               Length;
    HANDLE              RootDirectory;
    PUNICODE_STRING    ObjectName;
    ULONG               Attributes;
    PVOID                SecurityDescriptor;
    PVOID                SecurityQualityOfService;
} OBJECT_ATTRIBUTES, *POBJECT_ATTRIBUTES;
```

列表 2-5. OBJECT_ATTRIBUTES 结构

OBJECT_ATTRIBUTES 结构仅描述函数使用的数据的细节，**列表 2-6** 给出的 IO_STATUS_BLOCK 结构则用于记录对用户所提交的操作的处理结果。该结构很简单---Staus 成员存放一个 NTSTATUS 类型的代码，其值可能是 STATUS_SUCCESS 或定义于 ntstatus.h 中的所有可能的错误代码。Information 成员在操作成功的情况下，提供与操作相关的附加数据。比如，如果函数返回一个数据块，该成员将被设置为该数据块的大小。

```
typedef struct  _IO_STRATUS_BLOCK
```

```
{
    NTSTATUS  Status;
    ULONG     Information;
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

列表 2-6.　IO_STATUS_BLOCK 结构

另一个常见的 Windows 2000 数据类型是 LIST_ENTRY 结构，列表 2-7 给出了该结构的定义。内核使用该结构将所有对象维护在一个双向链表中。一个对象分属多个链表是很常见的，Flink 成员是一个向前链接，指向下一个 LIST_ENTRY 结构，Blink 成员则是一个向后链接，指向前一个 LIST_ENTRY 结构。通常情况下，这些链表都成环形，也就是说，最后一个 Flink 指向链表中的第一个 LIST_ENTRY 结构，而第一个 Blink 指向最后一个。这样就很容易双向遍历该链表。如果一个程序要遍历整个链表，它需要保存第一个 LIST_ENTRY 结构的地址，以判断是否已遍历了整个链表。如果链表仅包含一个 LIST_ENTRY 结构，那么该 LIST_ENTRY 结构必须引用其自身，也就是说，Flink 和 Blink 都指向其自己。

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

列表 2-7.　LIST_ENTRY 结构

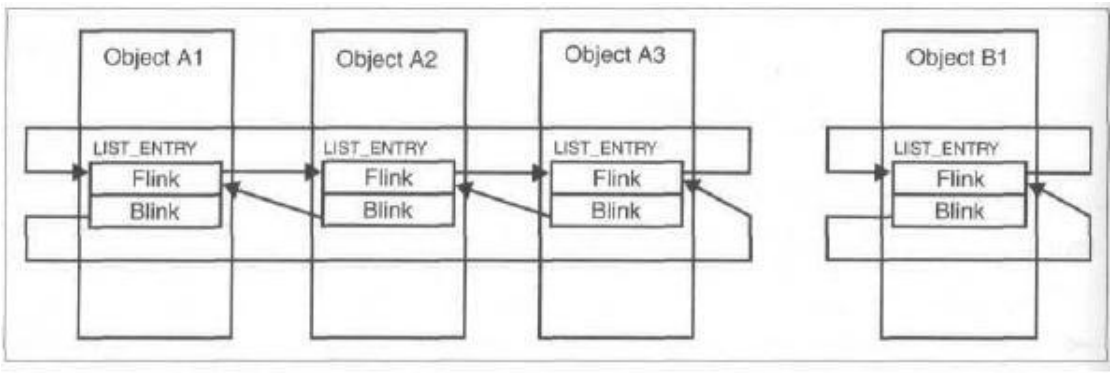


FIGURE 2-4.　*Examples of Doubly Linked Lists*

图 2-4 展示了对象链表各成员间的关系。对象 A1、A2、A3 属于同一链表。注意，A3 的 Flink 指向 A1，A1 的 Blink 指向 A3。最右边的对象 B1 仅有一个成员，因此，其 Flink 和 Blink 都指向相同的地址---即对象 B1 的地址。典型的双向链表的例子是进程和线程链表。内部变量 PsActiveProcessHead 就是一个 LIST_ENTRY 结构，位于 ntoskrnl.exe 的.data 节

中。该变量指向系统进程列表的首部（通过其 Blink 指针）。你可以在内核调试器中使用 dd PsActiveProcessHead 来获取该链表的首部，然后通过其 Flink 和 Blink 指针遍历整个链表（仍使用 dd 命令）。当然，这种探测 Windows 进程的方法非常繁琐，但这可使你深入的观察基本的系统结构。Windows 2000 Native API 提供了更便利的方法来枚举进程，如 NtQuerySystemInformation()函数。

```
typedef struct _CLIENT_ID
{
    HANDLE  UniqueProcess;
    HANDLE  UniqueThread;
} CLIENT_ID, *PCLIENT_ID;
```

列表 2-8.　CLIENT_ID 结构

处理进程和线程的 API 函数，如：NtOpenProcess()和 NtOpenThread()，使用**列表 2-8** 给出的 CLIENT_ID 结构来和特定的进程、线程相关联。尽管其类型为 HANDLE，实际上，从严格的意义上来讲 UniqueProcess 和 UniqueThread 成员并不是句柄（Handle），它们都是整数型的进程 ID 和线程 ID。即标准 Win32 函数 GetCurrentProcessId()和 GetCurrentThreadId()返回的 DWORD 类型的数值。

Windows 2000 执行体（Executive）还使用 CLIENT_ID 结构在全局范围内标识唯一的线程。例如，如果你使用内核调试器的!thread 命令来显示当前线程参数，就会在输出的第一行看到类似“Cid ppp.ttt”的显示，其中“ppp”就是 CLIENT_ID 的 UniqueProcess 成员，而“ttt”则代表 UniqueThread，如下所示。注意，我用黑体标出的地方。

```
kd> !thread
THREAD 83a51ba8  Cid 0a5c.0e64  Teb: 7ffdd000 Win32Thread: e14f4eb0 RUNNING on
processor 0
Not impersonating
DeviceMap              e20fb208
Owning Process          83a14708
Wait Start TickCount      906512         Elapsed Ticks: 68570
Context Switch Count     266             LargeStack
UserTime               00:00:00.0312
```

```
KernelTime              00:00:00.0015
。。。。。。。。。。。。。。。。。。。。
```

2.5、Native API 的接口

对于内核模式的驱动程序，使用 Native API 的接口非常平常，就像在用户模式下的程序中调用 Win32 API 一样。Windows 2000 DDK 提供的头文件和库包含了所有在调用 ntoskrnl.exe 导出的 Native API 时所需的信息。而另一方面，Win32 SDK 几乎不支持在程序中调用 ntdll.dll 导出的 Native API。我说“几乎不”是因为 Win32 SDK 实际上提供了一个重要的东西：导入库 ntdll.lib，该文件位于\Program Files\Microsoft Platfrom SDK\Lib 目录中。如果没有这个库，将很难调用 ntdll.dll 导出的函数。

译注：

你需要安装 Windows 2000 DDK 才能获得 ntdll.lib

可以到 http://www.microsoft.com/msdownload/platformsdk/sdkupdate/ 下载最新的 SDK

2.5.1、将 NTDLL.DLL 导入库添加到工程中

在你能成功的编译和链接在用户模式下使用 ntdll.dll 导出函数的代码之前，你必须考虑如下的四个重点：

- SDK 的头文件中，没有包含这些函数的原型。
- SDK 文件中缺少这些函数使用的几个基本的数据类型。
- SDK 和 DDK 头文件并不兼容，你不能将#include <ntddk.h>加入你的 Win32 C 源代码文件中。
- ntdll.lib 并没有加入 Visual C/C++默认的导入库列表中

最后一个问题很容易解决，只需要编辑工程的设置属性，或者将如下内容加入你的源代码中，`#pragma comment(linker,”defaultlib:ntdll.lib”)`，像在前面的 Windows 2000 运行时库一节解释的那样，这会在编译时，将 ntdll.dll 加入链接器的`/defaultlib`设置中。解决缺失的定义比较困难。因为不可能将 SDK 和 DDK 头文件整合到 C 程序中，最简易的解决

方法是写一格自定义的头文件，在该头文件中包含所有调用 ntdll.dll 导出函数必须的定义。幸运的是，你不需要开始这项工作了，在本书光盘的\src\common\include 目录下的 w2k_def.h 文件包含了你所需要的所有基本信息。该头文件将在第六、七两章中扮演重要角色。因为它被设计为可同时兼容用户模式和内核模式的工程，在用户模式代码中，你必须在#include <w2k_def.h>之前插入#define _USER_MODE_，以加入仅出现在 DDK 中的一些定义。

有关 Native API 编程的很多详细信息都已经出版，目前看来，针对 Windows 2000 平台的好书是 Gary Nebbett’s 的《Windows NT/2000 Native API Reference》。该书提供的示例程序较少，但它覆盖了 Windows NT/2000 平台上的所有 Native API，还包括这些函数需要的数据结构定义以及其他必须的一些结构定义。

将在第六章介绍的 w2k_call.dll 示例库，演示了 w2k_def.h 的典型用法。第六章还将讨论另一种在用户模式进入 Windows 2000 内核的方法，此种方法不受限于 Native API。事实上，这种技巧也可用于 ntoskrnl.exe，对于所有加载到内核空间的模块，只要它们导出了函数或者可以和.dbg 或.pdb 符号文件相匹配都可以使用此方法。如你所见，在本书剩余章节中还有很多有趣的信息。但是，在我们到达那儿之前，我们会继续讨论一些基本的概念和技术。

三、编写内核模式驱动程序

在下一章中，我们会经常访问那些仅在内核模式下才有效的系统资源。大量的示例代码都被设计为内核驱动例程（Kernel-mode driver routine）。因此，需要有关开发此种软件的基本知识。因为我不能假定所有读者都有这方面的经验，我会在此简要地介绍一下内核模式驱动程序编程，不过这仅集中在如何使用驱动开发向导（在本书光盘上）。

本章还将讨论 Windows 2000 服务控制管理器（Service Control Manager，SCM）的基本知识，这包括 SCM 如何允许在运行时加载、控制和卸载驱动程序，resulting in wonderfully short change-build-test turnaround cycles。本章的题目或许会让人有些误解，驱动一词通常与控制硬件的底层软件相关。事实上，很多内核程序员每天都在做这些事情。不过，Windows 2000 的驱动程序分层模式允许做比这更多的事情。内核驱动程序可以完成任意复杂的任务，若不考虑它们运行于更高的 CPU 特权级别上而且使用不同的开发接口，那它们很像用户模式下的 DLL。在此，我们将使用这种强大的开发技术来侦测 Windows 2000 的内部秘密，使用内核驱动程序就像驾驶从狭小的用户模式飞往 Windows 2000 内核的太空飞船。

3.1、创建一个驱动程序的骨架

即使长时间开发 Win32 应用程序和库的开发人员，在首次编写内核驱动程序时，也会感觉像是一个绝对的初学者。这是因为，内核模式下的代码运行在一个完全不同的操作系统环境中。Win32 开发人员的工作仅局限在属于 Windows 2000 Win32 子系统的几个系统组件上。其他开发人员可能编写 POSI 或 OS/2 应用程序，Windows 2000 的附加子系统为它们提供支持。感谢子系统这个概念，Windows 2000 就像一个变色龙---它可通过这些子系统（前面提及的）导出不同的应用程序开发接口来模拟不同的操作系统。与此相反，内核模式的代码可以看到“真实”的 Windows 2000 操作系统。它们使用的接口可以称之为“最终边界”。当然，这并不是说，内核模式完全摆脱了子系统。在第二章中，我们看到 win32k.sys 就是 Win32 GUI 和窗口管理器在内核模式下的分支，将它们放在内核是出于性能考虑。然而，win32k.sys 导出的 API 函数集合中只有一小部分出现在了 gdi32.dll 和 user32.dll 中，这也意味着只

有这一小部分函数可以作为 Win32 API 函数来使用，因此，Win32K 决不只是 Win32 踏入内核世界的一脚，实际上，应把它看作是一个高性能的内核模式的图形引擎。

3.1.1、Windows 2000 DDK（Device Driver Kit）

由于内核模式下的编程使用了不同的系统接口，在 Win32 编程中经常使用的头文件和库都将无法在内核模式下使用。针对 Win32 开发，微软提供了 Platform Software Development Kit（SDK）。而与内核模式的驱动开发相关的是，Windows 2000 Device Driver Kit（DDK）。随文档一起，DDK 还提供了特殊的头文件和导入库，这些都是 Windows 2000 内核模块必须的接口。安装完 DDK 之后，接下来你应该打开 Visual C/C++，把 DDK 的路径加入到编译器和链接器的目录列表中。在主菜单中选择 Tools→Options，然后单击 Directories。在目录选择下拉列表中选择 Include files，然后将 DDK 的适当路径加入，如图 3-1 所示。默认情况下，DDK 将安装到\NTDDK 目录下，included 文件位于\NTDDK\inc 子目录中。需要注意的是，请将新添加的路径置于原有路径的上方，这样就会使用新的头文件或者库。

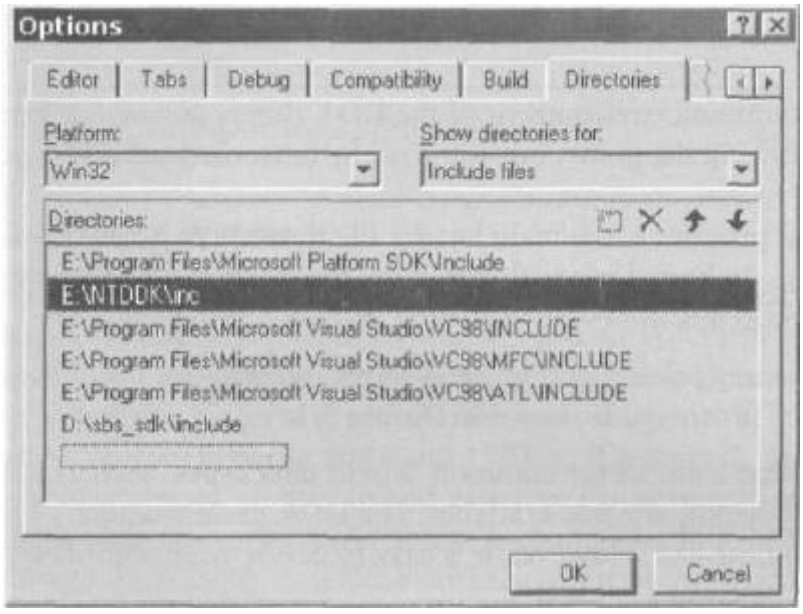


图 3-1 添加 DDK 头文件路径

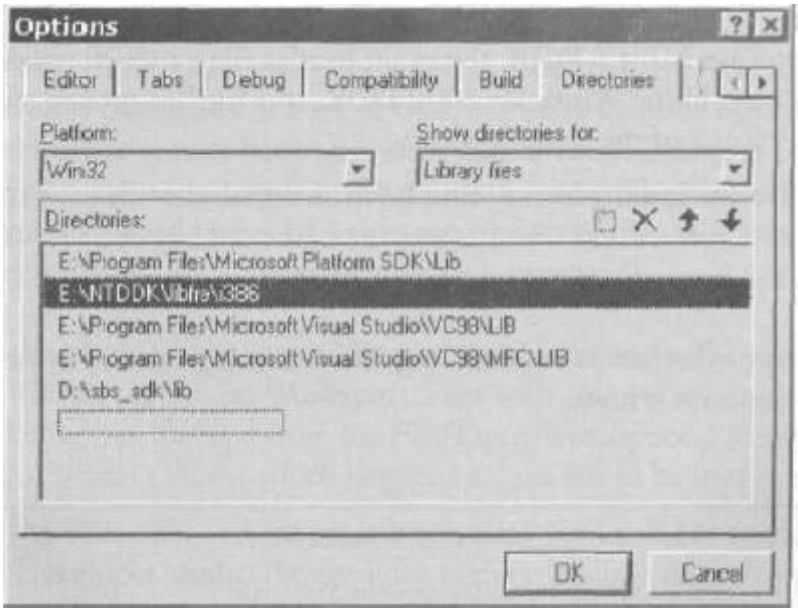


图 3-2 添加 DDK 导入库路径

在添加完 DDK 头文件路径后，用同样的方法添加导入库的路径。DDK 包含两组导入库，一组叫做 free（release）builds，另一组叫做 checked（debug）builds。其对应的目录为：\NTDDK\libfre\i386 和\NTDDK\libchk\i386，参见**图 3-2**。

DDK 开发环境与 Win32 模式有所不同，下面给出二者之间的一些明显区别：

- 对于 Win32 程序员来说，主要的头文件是 windows.h，对于内核模式代码来说，应使用 ntddk.h 替代之。
- 主进入点函数叫做 DirverEntry()，而不再是 WinMain()或 main()。**列表 3-1** 给出了它们的原型。
- 不能再使用一些常见的 Win32 数据类型，如 BYTE、WORD 和 DWORD。DDK 使用 UCHAR、USHORT、ULONG 等。

不过，很容易就能定义你自己喜欢的类型，**列表 3-2** 给出了这样的一个示例。

```
NTSTATUS DriverEntry ( PDRIVER_OBJECT pDriverObject,
                       PUNICODE_STRING pusRegistryPath);
```

列表 3-1　DriverEntry 函数的原型

```
typedef  UCHAR   BYTE, *PBYTE;
typedef  USHORT  WORD, *PWORD;
typedef  ULONG   DWORD, *PDWORD;
```

列表 3-2　定义常见的 Win32 数据类型

此外，还需要注意 Windows NT 4.0 和 Windows 2000 所使用的 DDK 之间的差别，有三点不同需要注意，如下：

- 默认情况下，Windows NT 4.0 DDK 的主目录叫做\DDK，而 Windows 2000 DDK 叫做\NTDDK
- 在 Windows NT 4.0 DDK 中，主要的头文件 ntddk.h 位于主目录之下。而在 Windows 2000 DDK 中，该文件被移到了\NTDDK\DDK 子目录下。
- 导入库的路径也发生了变化：lib\i386\free 变成了 libfre\i386，lib\i386\checked 变成了 libchk\i386。

我不知道微软的这种改变有什么实际意义，不过为了生活，我们还是需要了解其变化☺。

3.1.2、可定制的驱动程序向导

开发内核驱动程序的主要困难在于 Visual C/C++没有提供此种类型的工程向导。幸运的是，MSDN 里有一系列不错的关于 Windows NT 内核驱动开发的文章，是 Ruediger R. Asche. 在 1994 至 1995 年编写的。其中的两篇文章（Asche 1995a，1995b）详细说明了如何在 Visual C/C++中加入自定义的驱动程序向导，这些文章给了我很大的帮助，尽管原始向导的输出文件不能满足我的所有需求，但这是一个很好的开始。我提供的内核驱动向导将基于 Ruediger Asche 的原始向导产生的输出文件。

我提供的驱动向导的所有源代码位于本书光盘的\src\w2k_wiz 目录。通过阅读这些代码，你会发现它实际的标题“SBS Windows 2000 Code Wizard”。事实上，这是一个一般性的 Windows 2000 程序骨架生成器，该生成器可以产生多种类型的程序，包括 Win32 DLL 和应用程序。不过，光盘中的配置文件针对内核驱动开发做了一定的修改。基本上来说，我提供的向导是一个文件转换器，它读取一组文件，然后按照一些简单的规则将它们进行转换，最后将结果写入另一组文件中。输入文件是模板，输出文件是 C 工程文件。通过修改模板文件，该向导可以变成一个 DLL 向导等等。必须提供 7 个模板文件（如果丢失了某一个，会产生错误）：

- 扩展名为.tw 的文件是 workspace 模板，此种文件将会被保存为 Visual Studio 的工程文件.dsw。
- 扩展名为.tp 的文件是工程模板，此种文件将被保存为.dsp 文件。工程文件由于之关联的 workspace 文件引用，工程文件还包含生成工程的所有配置选项。

- 扩展名为.tc、.th、.tr 和.td 的文件都是 C 代码文件，这些文件最后会变成相应的.c、.h、.rc 和.def 文件。
- 扩展名为.ti 的是 icon 文件，该文件会被直接保存为.ico 文件。

这七个文件是一个新工程所必需的。.def 文件以一种较老风格的方法从 DLL 中导出 API 函数，不过我更喜欢__declspec(dllexport)方式。因为驱动程序通常不导出函数，所以我省略了.td 模板，导致的结果是，在开始时，向导会报告一个错误。我还省略了资源脚本和 icon 文件，不过经验告诉我，最好提供它们。采用的转换规则也非常简单，仅包含一个很短的字符串替换列表。在扫描模板文件时，转换器查找以%号开始的转义符。当它找到后，会根据%后的字符来决定执行什么样的动作。**表 3-1** 列出了验证过的转义符。

TABLE 3-1. *The Wizard's String Substitution Rules*

INPUT	OUTPUT
%n	Project name (original notation)
%N	Project name (uppercase notation)
%s	Fully qualified path of the *w2k_wiz.ini* file
%d	Current day (always two digits)
%m	Current month (always two digits)
%y	Current year (always four digits)
%t	Default project description, as defined in *w2k_wiz.ini*
%c	Author's company name, as defined in *w2k_wiz.ini*
%a	Author's name, as defined in *w2k_wiz.ini*
%e	Author's email address, as defined in *w2k_wiz.ini*
%p	Default ProgID prefix, as defined in *w2k_wiz.ini*
%i	DDK header file path, as defined in *w2k_wiz.ini*
%l	DDK import library path (release configuration), as defined in *w2k_wiz.ini*
%L	DDK import library path (debug configuration), as defined in *w2k_wiz.ini*
%%	% (escapement for a single percent character)
%<other>	Copied unchanged to the output file

表 3-1 中有几处需参考配置文件---w2k_wiz.ini。**示例 3-1** 给出了其默认设置。在使用向导之前，你应该将光盘\src\w2k_wiz\release 目录下的 w2k_wiz.exe、w2k_wiz.ini 和所有的 w2k_wiz.t*模板文件复制到你的硬盘上，然后编辑配置文件，将对应内容改为你自己的设置。你还需要修改 Include、Free 和 Checked，使其和你的 DDK 安装相匹配。如果你使用 Visual C/C++ 6.0，可以不改变 Root 的值。如果不，则将其设为你存放工程文件的根目录。如果以一个反斜线结尾，它将作为默认值。在**示例 3-1** 中，其键值为：

HKEY_CURRENT_USER\SoftWare\Microsofto\DevStudio\6.0\Directories，而 WorkspaceDir 用来存放基本的工作目录。

键入 w2k_wiz MyDriver 来执行该向导，它会当前目录下创建名为 MyDriver 的工程目录，该目录将存放向导生成的 MyDriver.dsw、MyDriver.dsp、MyDriver.c、MyDriver.h、MyDriver.rc 和 MyDriver.ico 文件。如果你指定了具体的路径，则会在你指定的路径下创建该目录。另一个合法的命令选项是星号，如：w2k_wiz *MyDriver。在此种情况下，向导不会在当前目录下创建工程目录，而是去查找 Visual C/C++维护的默认的工程根目录，即 w2k_wiz.ini 中的 Root 所指向的位置。

```
; w2k_wiz.ini
; 08-27-2000 Sven B. Schreiber
; sbs@orgon.com

[Settings]
Text    = <SBS Windows 2000 Code Wizard Project>
Company = <MyCompany>
Author  = <MyName>
Email   = <my@email>
Prefix   = <MyPrefix>
Include  = E:\NTDDK\inc
Free    = E:\NTDDK\libfre\i386
Checked = E:\NTDDK\libchk\i386
Root =
HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\Directories\WorkspaceDir
```

示例 3-1.　向导支持的自定义选项

3.1.3、运行驱动向导

现在，来试试这个驱动向导。**示例 3-2** 给出了在 Windows 2000 控制台下执行 w2k_wiz *TestDrv 后的输出。这将在 Visual C/C++默认的工程根目录下创建一个名为 TestDrv 的工程目录。

```
D:\>w2k_wiz *TestDrv

// w2k_wiz.exe
// SBS Windows 2000 Code Wizard V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Project D:\Program Files\DevStudio\MyProjects\TestDrv\

Loading D:\etc32\w2k_wiz.tc ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.c ... OK

Loading D:\etc32\w2k_wiz.td ... ERROR

Loading D:\etc32\w2k_wiz.th ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.h ... OK

Loading D:\etc32\w2k_wiz.ti ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.ico ... OK

Loading D:\etc32\w2k_wiz.tp ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.dsp ... OK

Loading D:\etc32\w2k_wiz.tr ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.rc ... OK

Loading D:\etc32\w2k_wiz.tw ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.dsw ... OK
```

EXAMPLE 3-2. *Running the Windows 2000 Code Wizard*

显然，除了将.td 模板转换为.def 时出了错，其余转换都成功的完成了。因为该向导生成的驱动程序骨架不需要.def 文件，所以不需要提供.td 模板文件。现在，用 Visual C/C++打开一个新的 WorkSpace，然后你会发现一个名为 TestDrv 的新目录，该目录中包含一个名为 TestDrv.dsw 的 WorkSpace 文件。该文件可以被正确的打开。接下来，你因该为生成项目选择活动的配置信息。驱动向导生成的.dsp 文件提供了如下两个可用配置：

1. Win2k Kernel-mode Driver(debug)
2. Win2k Kernel-mode Driver(release)

默认情况下，将使用 debug 配置来生成项目，但是你可在任何时候从 Visual C/C++菜单 Build/Set Active Configuration 来选择不同的项目配置。最后，你要将光盘中的 \src\common\include\DrvInfo.h 复制到你自己的头文件目录中。在打开 TestDrv.rc 时，应使用文本模式来打开（如**图 3-3** 所示），这是因为该文件使用了来自 DrvInfo.h 中的复杂的宏定义，这些宏会导致资源编辑器异常退出。这个错误从 Visual C/C++ 5.0 开始，在我印象中，一直没有被改正过。和编辑器不同，资源编译器（Resource Compiler）可以正常的处理这些宏。

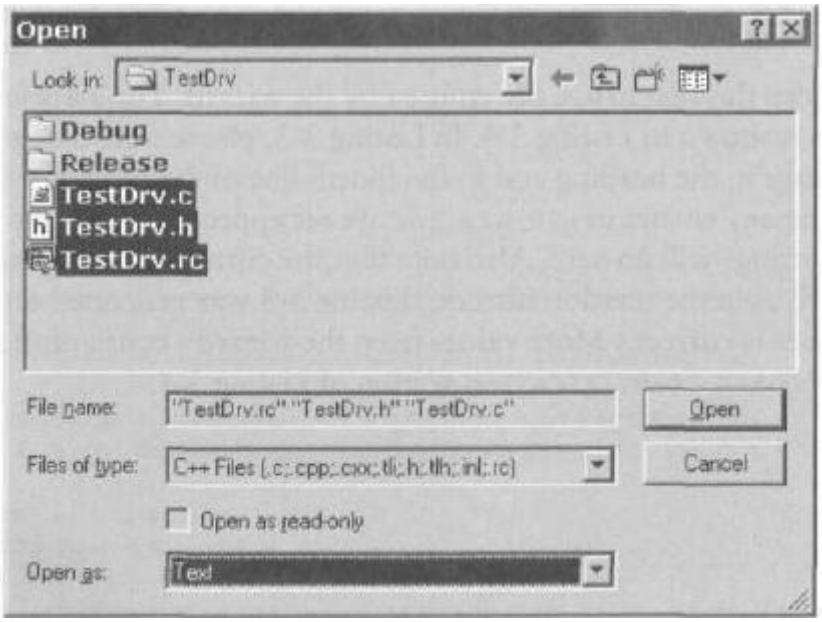


图 3-3. 以文本模式打开 TestDrv.c、TestDrv.h 和 TestDrv.rc

现在，已经为第一次编译做好了所有准备。在示例 3-3 中，我通过选择 Build/Rebuild 菜单来建立 Driver 的 Release 版，看起来一切都正常。顺便说一下，头两行末尾的省略号表示我截断了 Build 命令的输出。

链接器会在 Debug 或 Release 目录下创建了一个名为 TestDrv.sys 的可执行文件，这依赖于你的生成配置。Test Driver 的 Release 版大小为 5.5KB，其 Debug 版为 8KB。你可以使用本书光盘中的 MFVDasm 或 PEView 来验证 TestDrv.sys 是否包含有效的代码和数据。

```
Deleting intermediate files and output files for project 'TestDrv - Win2K ...
      ●- Configuration: TestDrv - Win2K kernel-mode driver (release) ...
Compiling resources...
Compiling...
TestDrv.c
Linking...

TestDrv.sys - 0 error(s), 0 warning(s)
```

EXAMPLE 3-3. *Building the Release Version of the Test Driver*

3.1.4、深入驱动程序的骨架

列表 3-3 展示了向导生成的 TestDrv.c。与之相关的头文件 TestDrv.h 在**列表 3-4** 中。在**列表 3-3** 中，请注意标题处的<MyName>和<MyCompany>标志。如果 w2k_wiz.ini 中的作者和公司名称正确，那你自己的名字和相应的公司名称将会替代它们。

```
// TestDrv.c
// 08-07-2000 <MyName>
```

```
// Copyright @2005 <MyCompany>


#define  _TESTDRV_SYS_
#include <ntddk.h>
#include "TestDrv.h"


// ================================================================
// DISCLAIMER
// ================================================================


/*


This software is provided "as is" and any express or implied
warranties, including, but not limited to, the implied warranties of
merchantability and fitness for a particular purpose are disclaimed.
In no event shall the author <MyName> be liable for any
direct, indirect, incidental, special, exemplary, or consequential
damages (including, but not limited to, procurement of substitute
goods or services; loss of use, data, or profits; or business
interruption) however caused and on any theory of liability,
whether in contract, strict liability, or tort (including negligence
or otherwise) arising in any way out of the use of this software,
even if advised of the possibility of such damage.


*/


// ================================================================
// REVISION HISTORY
// ================================================================
```

```
/*

08-07-2000 V1.00 Original version.

*/

// ================================================================
// GLOBAL DATA
// ================================================================

PRESET_UNICODE_STRING (usDeviceName,       CSTRING (DRV_DEVICE));
PRESET_UNICODE_STRING (usSymbolicLinkName, CSTRING (DRV_LINK  ));

PDEVICE_OBJECT  gpDeviceObject  = NULL;
PDEVICE_CONTEXT gpDeviceContext = NULL;

// ================================================================
// DISCARDABLE FUNCTIONS
// ================================================================

NTSTATUS DriverInitialize (PDRIVER_OBJECT  pDriverObject,
                           PUNICODE_STRING pusRegistryPath);

NTSTATUS DriverEntry      (PDRIVER_OBJECT  pDriverObject,
                           PUNICODE_STRING pusRegistryPath);

// ----------------------------------------------------------------
```

```
#ifdef ALLOC_PRAGMA


#pragma alloc_text (INIT, DriverInitialize)

#pragma alloc_text (INIT, DriverEntry)


#endif


// ================================================================

// DEVICE REQUEST HANDLER

// ================================================================


NTSTATUS DeviceDispatcher (PDEVICE_CONTEXT pDeviceContext,

                           PIRP            pIrp)

    {

    PIO_STACK_LOCATION pisl;

    DWORD              dInfo = 0;

    NTSTATUS           ns    = STATUS_NOT_IMPLEMENTED;


    pisl = IoGetCurrentIrpStackLocation (pIrp);


    switch (pisl->MajorFunction)

        {

        case IRP_MJ_CREATE:

        case IRP_MJ_CLEANUP:

        case IRP_MJ_CLOSE:

            {

            ns = STATUS_SUCCESS;

            break;

            }
```

```
        }
    pIrp->IoStatus.Status      = ns;
    pIrp->IoStatus.Information = dInfo;

    IoCompleteRequest (pIrp, IO_NO_INCREMENT);
    return ns;
    }

// ====================================================================
// DRIVER REQUEST HANDLER
// ====================================================================

NTSTATUS DriverDispatcher (PDEVICE_OBJECT pDeviceObject,
                           PIRP           pIrp)
    {
    return (pDeviceObject == gpDeviceObject
            ? DeviceDispatcher (gpDeviceContext, pIrp)
            : STATUS_INVALID_PARAMETER_1);
    }

// --------------------------------------------------------------------

void DriverUnload (PDRIVER_OBJECT pDriverObject)
    {
    IoDeleteSymbolicLink (&usSymbolicLinkName);
    IoDeleteDevice       (gpDeviceObject);
    return;
    }
```

```
// ============================================================

// DRIVER INITIALIZATION

// ============================================================


NTSTATUS DriverInitialize (PDRIVER_OBJECT  pDriverObject,

                           PUNICODE_STRING pusRegistryPath)

    {

    PDEVICE_OBJECT pDeviceObject = NULL;

    NTSTATUS       ns = STATUS_DEVICE_CONFIGURATION_ERROR;


    if ((ns = IoCreateDevice (pDriverObject, DEVICE_CONTEXT_,

                              &usDeviceName, FILE_DEVICE_CUSTOM,

                              0, FALSE, &pDeviceObject))

        == STATUS_SUCCESS)

        {

        if ((ns = IoCreateSymbolicLink (&usSymbolicLinkName,

                                        &usDeviceName))

            == STATUS_SUCCESS)

            {

            gpDeviceObject  = pDeviceObject;

            gpDeviceContext = pDeviceObject->DeviceExtension;


            gpDeviceContext->pDriverObject = pDriverObject;

            gpDeviceContext->pDeviceObject = pDeviceObject;

            }

        else

            {

            IoDeleteDevice (pDeviceObject);

            }
```

```
        }
    return ns;
    }

// ------------------------------------------------------------

NTSTATUS DriverEntry (PDRIVER_OBJECT  pDriverObject,
                      PUNICODE_STRING pusRegistryPath)
    {
    PDRIVER_DISPATCH *ppdd;
    NTSTATUS          ns = STATUS_DEVICE_CONFIGURATION_ERROR;

    if ((ns = DriverInitialize (pDriverObject, pusRegistryPath))
        == STATUS_SUCCESS)
        {
        ppdd = pDriverObject->MajorFunction;

        ppdd [IRP_MJ_CREATE                   ] =
        ppdd [IRP_MJ_CREATE_NAMED_PIPE        ] =
        ppdd [IRP_MJ_CLOSE                    ] =
        ppdd [IRP_MJ_READ                     ] =
        ppdd [IRP_MJ_WRITE                    ] =
        ppdd [IRP_MJ_QUERY_INFORMATION        ] =
        ppdd [IRP_MJ_SET_INFORMATION          ] =
        ppdd [IRP_MJ_QUERY_EA                 ] =
        ppdd [IRP_MJ_SET_EA                   ] =
        ppdd [IRP_MJ_FLUSH_BUFFERS            ] =
        ppdd [IRP_MJ_QUERY_VOLUME_INFORMATION] =
        ppdd [IRP_MJ_SET_VOLUME_INFORMATION  ] =
```

```
        ppdd [IRP_MJ_DIRECTORY_CONTROL       ] =
        ppdd [IRP_MJ_FILE_SYSTEM_CONTROL     ] =
        ppdd [IRP_MJ_DEVICE_CONTROL          ] =
        ppdd [IRP_MJ_INTERNAL_DEVICE_CONTROL ] =
        ppdd [IRP_MJ_SHUTDOWN                ] =
        ppdd [IRP_MJ_LOCK_CONTROL            ] =
        ppdd [IRP_MJ_CLEANUP                 ] =
        ppdd [IRP_MJ_CREATE_MAILSLOT         ] =
        ppdd [IRP_MJ_QUERY_SECURITY          ] =
        ppdd [IRP_MJ_SET_SECURITY            ] =
        ppdd [IRP_MJ_POWER                   ] =
        ppdd [IRP_MJ_SYSTEM_CONTROL          ] =
        ppdd [IRP_MJ_DEVICE_CHANGE           ] =
        ppdd [IRP_MJ_QUERY_QUOTA             ] =
        ppdd [IRP_MJ_SET_QUOTA               ] =
        ppdd [IRP_MJ_PNP                     ] = DriverDispatcher;
        pDriverObject->DriverUnload            = DriverUnload;
        }
    return ns;
    }


// ====================================================================
// END OF PROGRAM
// ====================================================================
```

列表 3-3. 驱动程序骨架的源代码

```
// TestDrv.h
// 08-07-2000 <MyName>
// Copyright @2005 <MyCompany>
```

```
// ================================================================
// PROGRAM IDENTIFICATION
// ================================================================

#define DRV_BUILD           1
#define DRV_VERSION_HIGH    1
#define DRV_VERSION_LOW     0

// ----------------------------------------------------------------

#define DRV_DAY             07
#define DRV_MONTH           02
#define DRV_YEAR            2005

// ----------------------------------------------------------------
// Customize these settings by editing the configuration file
// D:\etc32\w2k_wiz.ini

#define DRV_MODULE          TestDrv
#define DRV_NAME            <SBS Windows 2000 Code Wizard Project>
#define DRV_COMPANY         <MyCompany>
#define DRV_AUTHOR          <MyName>
#define DRV_EMAIL           <my@email>
#define DRV_PREFIX          <MyPrefix>

// ================================================================
// HEADER FILES
// ================================================================
```

```
#include "drvinfo.h"        // defines more DRV_* items

////////////////////////////////////////////////////////////////////
#ifndef _RC_PASS_
////////////////////////////////////////////////////////////////////

// ================================================================
// CONSTANTS
// ================================================================

#define FILE_DEVICE_CUSTOM      0x8000

// ================================================================
// STRUCTURES
// ================================================================

typedef struct _DEVICE_CONTEXT
    {
    PDRIVER_OBJECT pDriverObject;
    PDEVICE_OBJECT pDeviceObject;
    }
    DEVICE_CONTEXT, *PDEVICE_CONTEXT, **PPDEVICE_CONTEXT;

#define DEVICE_CONTEXT_ sizeof (DEVICE_CONTEXT)

////////////////////////////////////////////////////////////////////
#endif // #ifndef _RC_PASS_
////////////////////////////////////////////////////////////////////
```

```
// ============================================================

// END OF FILE

// ============================================================
```

列表 3-4. 驱动程序骨架的头文件

作为补充这里给出 DrvInfo.h 的内容：

```
// ____________________________________________________________

//

//                           DrvInfo.h

//                Driver Info Definitions V1.00

//                06-02-2000 Sven B. Schreiber

//                        sbs@orgon.com

// ____________________________________________________________


#ifndef _DRVINFO_H_

#define _DRVINFO_H_


// ============================================================

// DISCLAIMER

// ============================================================


/*


This software is provided "as is" and any express or implied

warranties, including, but not limited to, the implied warranties of

merchantability and fitness for a particular purpose are disclaimed.

In no event shall the author Sven B. Schreiber be liable for any

direct, indirect, incidental, special, exemplary, or consequential

damages (including, but not limited to, procurement of substitute

goods or services; loss of use, data, or profits; or business
```

```
interruption) however caused and on any theory of liability,
whether in contract, strict liability, or tort (including negligence
or otherwise) arising in any way out of the use of this software,
even if advised of the possibility of such damage.

*/

// ====================================================================
// REVISION HISTORY
// ====================================================================

/*

05-26-2000 V1.00 Original version (SBS).

*/

// ====================================================================
// BASIC TYPES
// ====================================================================

typedef UCHAR               BYTE,   *PBYTE,   **PPBYTE;
typedef USHORT              WORD,   *PWORD,   **PPWORD;
typedef ULONG               DWORD,  *PDWORD,  **PPDWORD;
typedef unsigned __int64    QWORD,  *PQWORD,  **PPQWORD;
typedef int                 BOOL,   *PBOOL,   **PPBOOL;
typedef void                                  **PPVOID;

// --------------------------------------------------------------------
```

```
#define BYTE_                sizeof (BYTE)
#define WORD_                sizeof (WORD)
#define DWORD_               sizeof (DWORD)
#define QWORD_               sizeof (QWORD)
#define BOOL_                sizeof (BOOL)
#define PVOID_               sizeof (PVOID)
#define HANDLE_              sizeof (HANDLE)
#define PHYSICAL_ADDRESS_    sizeof (PHYSICAL_ADDRESS)

// ============================================================
// MACROS
// ============================================================

#define _DRV_DEVICE(_name)  \\Device\\     ## _name
#define _DRV_LINK(_name)    \\DosDevices\\ ## _name
#define _DRV_PATH(_name)    \\\\.\\        ## _name

// ------------------------------------------------------------

#define _CSTRING(_text) #_text
#define CSTRING(_text) _CSTRING (_text)

#define _USTRING(_text) L##_text
#define USTRING(_text) _USTRING (_text)

#define PRESET_UNICODE_STRING(_symbol,_buffer) \
        UNICODE_STRING _symbol = \
            { \
```

```
            sizeof (USTRING (_buffer)) - sizeof (WORD), \
            sizeof (USTRING (_buffer)), \
            USTRING (_buffer) \
            };

// ------------------------------------------------------------

#if DRV_VERSION_LOW < 10

#define _DRV_V2(_a,_b)       _a ## .0 ## _b
#define _DRV_V2X(_a,_b) V ## _a ## .0 ## _b

#else // #if DRV_VERSION_LOW < 10

#define _DRV_V2(_a,_b)       _a ## .  ## _b
#define _DRV_V2X(_a,_b) V ## _a ## .  ## _b

#endif // #if DRV_VERSION_LOW < 10 #else

#define DRV_V2(_a,_b)  _DRV_V2(_a,_b)
#define DRV_V2X(_a,_b) _DRV_V2X(_a,_b)

// ------------------------------------------------------------

#define _DRV_V4(_a,_b,_c) _a ## . ## _b ## .0. ## _c
#define DRV_V4(_a,_b,_c) _DRV_V4(_a,_b,_c)

// ------------------------------------------------------------
```

```
#define DRV_V               DRV_V2X (DRV_VERSION_HIGH, \
                                     DRV_VERSION_LOW)

#define DRV_VERSION         DRV_V2  (DRV_VERSION_HIGH, \
                                     DRV_VERSION_LOW)

#define DRV_VERSION_QUAD    DRV_V4  (DRV_VERSION_HIGH, \
                                     DRV_VERSION_LOW,   \
                                     DRV_BUILD)

#define DRV_VERSION_BINARY          ((DRV_VERSION_HIGH * 100) \
                                    + DRV_VERSION_LOW)

// ================================================================
// DRIVER INFORMATION
// ================================================================

#define DRV_ID                  DRV_PREFIX.DRV_MODULE
#define DRV_ID_VERSION          DRV_ID.DRV_VERSION_HIGH
#define DRV_FILENAME            DRV_MODULE.DRV_EXTENSION
#define DRV_CAPTION             DRV_NAME DRV_V
#define DRV_COMMENT             DRV_DATE DRV_AUTHOR

// ----------------------------------------------------------------

#define DRV_DEVICE              _DRV_DEVICE (DRV_MODULE)
#define DRV_LINK                _DRV_LINK   (DRV_MODULE)
#define DRV_PATH                _DRV_PATH   (DRV_MODULE)
#define DRV_EXTENSION           sys
```

```
// --------------------------------------------------------------------

#define DRV_CLASS               DRV_MODULE.DRV_VERSION_QUAD
#define DRV_ICON                DRV_MODULE.Icon

// --------------------------------------------------------------------

#define DRV_COPYRIGHT           Copyright \xA9 DRV_YEAR
#define DRV_COPYRIGHT_EX        DRV_COPYRIGHT DRV_COMPANY

// --------------------------------------------------------------------

#define DRV_DATE_US             DRV_MONTH-DRV_DAY-DRV_YEAR
#define DRV_DATE_GERMAN         DRV_DAY.DRV_MONTH.DRV_YEAR
#define DRV_DATE                DRV_DATE_US

// ====================================================================
// NT4 COMPATIBILITY
// ====================================================================

#ifndef IRP_MJ_QUERY_POWER
#define IRP_MJ_QUERY_POWER      0x16
#endif

#ifndef IRP_MJ_SET_POWER
#define IRP_MJ_SET_POWER        0x17
#endif
```

```
#ifndef IRP_MJ_PNP_POWER

#define IRP_MJ_PNP_POWER        0x1B

#endif


////////////////////////////////////////////////////////////////////////

#ifdef _RC_PASS_

////////////////////////////////////////////////////////////////////////


// ====================================================================

// HEADER FILES

// ====================================================================


#include <winver.h>


// ====================================================================

// VERSION INFO

// ====================================================================


#define DRV_RC_VERSION \

VS_VERSION_INFO VERSIONINFO \

\

FILEVERSION    DRV_VERSION_HIGH, DRV_VERSION_LOW, 0, DRV_BUILD \

PRODUCTVERSION DRV_VERSION_HIGH, DRV_VERSION_LOW, 0, DRV_BUILD \

FILEFLAGSMASK  VS_FFI_FILEFLAGSMASK \

FILEFLAGS      0 \

FILEOS         VOS_NT \

FILETYPE       VFT_DRV \

FILESUBTYPE    VFT2_UNKNOWN \

  { \
```

```
  BLOCK "StringFileInfo" \
    { \
    BLOCK "040904B0" \
      { \
      VALUE "OriginalFilename", CSTRING (DRV_FILENAME\0) \
      VALUE "InternalName",     CSTRING (DRV_MODULE\0) \
      VALUE "ProductName",      CSTRING (DRV_NAME\0) \
      VALUE "FileDescription",  CSTRING (DRV_CAPTION\0) \
      VALUE "CompanyName",      CSTRING (DRV_COMPANY\0) \
      VALUE "ProductVersion",   CSTRING (DRV_VERSION_QUAD\0) \
      VALUE "FileVersion",      CSTRING (DRV_VERSION_QUAD\0) \
      VALUE "LegalCopyright",   CSTRING (DRV_COPYRIGHT_EX\0) \
      VALUE "Comments",         CSTRING (DRV_COMMENT\0) \
      } \
    } \
  BLOCK "VarFileInfo" \
      { \
      VALUE "Translation", 0x0409, 0x04B0 \
      } \
  }


// ================================================================
// RESOURCES
// ================================================================


#define DRV_RC_ICON DRV_ICON ICON DRV_MODULE.ico


/////////////////////////////////////////////////////////////////////
#endif // #ifdef _RC_PASS_
```

```
//////////////////////////////////////////////////////////////////////


#endif // #ifndef _DRVINFO_H_


// ====================================================================
// END OF FILE
// ====================================================================
```

列表 3-3 和**列表 3-4** 给出的驱动程序的 C 代码中包含了几乎所有 Kernel-mode Driver 都需要的基本代码。我会尽量使该驱动向导有更好的可定制性。你可以自由的更改向导提供的模板文件。对于想保留原有代码的人，下面的章节会为你提供该向导的一些内部细节的简要介绍。

该向导生成的驱动模块的进入点是 DriverEntry()。像所有的 Windows 2000 模块的进入点一样，这个名字并不时必须的。你可以使用任何你喜欢的名称，但是你必须告诉链接器你所使用的进入点名称，通过链接器的命令行选项/entry 可以做到这一点。对于前面提及的 TestDriver,向导已经很好的完成了这项工作。在 w2k_wiz.tp 模板或生成的 TestDrv.dsp 文件中，你会在链接器的命令行中找到/entry:”DriverEntry@8”这样的字符串。@8 后缀表示 DriverEntry()接受 8 个字节的参数（这些参数位于栈中），这和**列表 3-1** 提供的 DriverEntry()的原型一致：两个指针参数，每个占据 32 个位，共使用 64 个二进制位，即 8 字节。

DriverEntry()做的第一件事是调用 DriverInitialize()，该函数将创建一个设备对象（Device Object）和该对象的一个符号链接（Symbolic link），在稍后你可能在用户模式的程序中使用该符号链接来与设备通讯。要想找到 IoCreateDevice()和 IoCreateSymbolicLink()所使用的名字就有些许的困难，因为它们都是依赖 DrvzInfo.h（位于本书光盘的\src\common\include 目录）中的宏定义。如果你想更多的了解这个技巧，请参考 TestDrv.h（前面的**列表 3-4** 已列出）中的 PROGRAM IDENTIFICATION 一节，并跟踪形如 DRV_*的定义,它们以多种方式成组的出现在 DrvInfo.h 中。例如,一个完整的 VERSIONINFO 资源就是由多个小的宏构成的。在别处，还定义了 DRV_DEVICE 和 DRV_LINK 常量，在这里，它们分别等价于\Device\TestDrv 和\DosDevice\TestDrv。注意，很多内核 API 函数，如 IoCreateDevice()和 IoCreateSymbolickLinke()不接受一个以零结尾的字符串，仅支持一

个特殊的结构体---UNICODE_STRING，该结构在第二章已经介绍过，**列表 3-5** 再次给出了该结构的定义。定义于 DrvInfo.h 中的宏----PRESET_UNICODE_STRING（应用于 TestDrv.c 的 GLOBAL DATA Section）从一个简单的 Unicode 字符串常量创建出一个静态的 UNICODE_STRING 结构。这是针对 UNICODE_STRING 结构的一个方便的速记符号。

在成功的创建完设备对象及其符号链接后，DriverInitialize()将设备对象指针和设备上下文（Device Context）的指针保存在一个静态全局变量中。Device Context 是设备的一个私有结构，该结构可以有任意的大小和结构。本书提供的驱动程序骨架附带了一个简单的 DEVICE_CONTEXT 结构，该结构定义于 TestDrv.h 中。该结构仅包含分别指向设备和设备驱动程序对象的两个指针。你可以扩展该结构来保存设备驱动程序所特有的数据。系统针对驱动程序接收到的每个 I/O 请求包（I/O Request Packet，IRP）提供相应的 Device Context。

在 DriverInitialize()成功完成并返回后，DriverEntry()将建立一个重要的数组，该数组由系统传入，并做为驱动程序对象结构----pDriverObject 的一部分。该数组为驱动程序期望的所有 IRP 提供空间，同时 DriverEntry()还为所有希望得到控制的 IRP 写入对应的 CallBack 函数的指针。本书提供的驱动程序骨架遵循此种设计，它保存了一个 DriverDispatcher()指针，并提供了可存放 28 个 IRP 的空间，如**表 3-2** 所示。稍后，DriverDispatcher()将决定需要注意那些类型的 IRP，并针对所有不感兴趣的 IRP 返回 STATUS_NOT_IMPLEMENTED。需要注意的是，Windows NT 和 Windows 2000 的 IRP 处理例程数组的布局有一些微妙的差别。在**表 3-2** 中，这种差别以黑体标识出来。

译注：

在 Windows NT 中，大多数的 I/O 请求都是用 I/O 请求包（IRP）来表示的。在多数情况下，I/O 请求包可以从一个 I/O 系统组件转移到另一组件。这种设计允许单个应用程序线程并行的管理多个 I/O 请求。IRP 是一种数据结构，包含描述一个 I/O 请求的完整信息。具体的细节，请参考《Inside Windows 2000》的第 9 章 I/O System

```
typedef struct  _UNICODE_STRING
{
    WORD    Length;
    WORD    MaximumLength;
    PWORD   Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
```

列表 3-5.　一个普遍存在的 Windows 2000 结构：UNICODE_STRING

元素	Windows NT 4.0	Windows 2000
0x00	IRP_MJ_CREATE	IRP_MJ_CREATE
0x01	IRP_MJ_CREATE_NAMED_PIPE	IRP_MJ_CREATE_NAMED_PIPE
0x02	IRP_MJ_CLOSE	IRP_MJ_CLOSE
0x03	IRP_MJ_READ	IRP_MJ_READ
0x04	IRP_MJ_WRITE	IRP_MJ_WRITE
0x05	IRP_MJ_QUERY_INFORMATION	IRP_MJ_QUERY_INFORMATION
0x06	IRP_MJ_SET_INFORMATION	IRP_MJ_SET_INFORMATION
0x07	IRP_MJ_QUERY_EA	IRP_MJ_QUERY_EA
0x08	IRP_MJ_SET_EA	IRP_MJ_SET_EA
0x09	IRP_MJ_FLUSH_BUFFERS	IRP_MJ_FLUSH_BUFFERS
0x0A	IRP_MJ_QUERY_VOLUME_INFORMATION	IRP_MJ_QUERY_VOLUME_INFORAMTION
0x0B	IRP_MJ_SET_VOLUME_INFORMATION	IRP_MJ_SET_VOLUME_INFORMATION
0x0C	IRP_MJ_DIRECTORY_CONTROL	IRP_MJ_DIRECTORY_CONTROL
0x0D	IRP_MJ_FILE_SYSTEM_CONTROL	IRP_MJ_FILE_SYSTEM_CONTROL
0x0E	IRP_MJ_DEVICE_CONTROL	IRP_MJ_DEVICE_CONTROL
0x0F	IRP_MJ_INTERNAL_DEVICE_CONTROL	IRP_MJ_INTERNAL_DEVICE_CONTROL
0x10	IRP_MJ_SHUTDOWN	IRP_MJ_SHUTDOWN
0x11	IRP_MJ_LOCK_CONTROL	IRP_MJ_LOCK_CONTROL
0x12	IRP_MJ_CLEANUP	IRP_MJ_CLEANUP
0x13	IRP_MJ_CREATE_MAILSLOT	IRP_MJ_CREATE_MAILSLOT
0x14	IRP_MJ_QUERY_SECURITY	IRP_MJ_QUERY_SECURITY
0x15	IRP_MJ_SET_SECURITY	IRP_MJ_SET_SECURITY
0x16	IRP_MJ_QUERY_POWER	IRP_MJ_POWER
0x17	IRP_MJ_SET_POWER	IRP_MJ_SYSTEM_CONTROL
0x18	IRP_MJ_DEVICE_CHANGE	IRP_MJ_DEVICE_CHANGE
0x19	IRP_MJ_QUERY_QUOTA	IRP_MJ_QUERY_QUOTA
0x1A	IRP_MJ_SET_QUOTA	IRP_MJ_SET_QUOTA

0x1B	IRP_MJ_PNP_POWER	IRP_MJ_PNP

表 3-2. 数组中的每个 I/O 请求包的比较

在 IRP 数组建立好之后，DriverEntry()将自己的 CallBack 函数----DriverUnload()写入驱动程序对象结构中，这将允许在运行时卸载该驱动程序。DriverUnload()函数只是简单的销毁由 DriverInitialize()创建的所有对象（即设备对象和其符号链接）。在此之后，就可安全的将驱动程序从系统中移除。

每当一个模块要求驱动程序做出相应时，就会调用 DriverDispatcher()函数。因为，驱动程序能够处理多个设备，Dispatcher 首先检查那个设备应该响应该请求。本书提供的驱动程序骨架仅维护了一个设备，因此，仅需要在初始化时检查从 IoCreateDevice()接受到的设备对象指针是否一致。如果一致，DriverDispatcher()将接收到的 IRP 向前传递，给之前的 DriverDispatcher()函数，随之传递的还有 DriverInitialize()准备好的 Device Context。当你扩展该驱动骨架以管理多个设备驱动程序时，你可能需要为每个设备编写独立的 IRP dispatcher。**列表 3-3** 中的 DeviceDispatcher()函数只是一个示意性的实现，它仅能识别三种常见的请求：IRP_MJ_CREATE、IRP_MJ_CLEANUP 和 IRP_MJ_CLOSE，并通过返回 STATUS_SUCCESS 来表示以处理该请求。这是使设备能够正常打开、关闭的最小实现方式，对于其他的请求都将返回一个 STATUS_NOT_IMPLEMENTED。

你可能想知道在**列表 3-3** 的 DISCARDABLE FUNCTIONS 一节中出现的`#pragma alloc_text` 的目的。#pragma 指示符是将命令送往编译器和链接器的有力手段。`alloc_text` 命令表示将指定函数的代码写入可执行文件的非默认 section 中。默认情况下，所有程序代码都位于.text section。然而，指示符`#pragma alloc_text(INIT,DriverEntry)`将使 DriverEntry()的代码保存在一个新的 section----INIT 中。驱动加载器可以识别这种指定的 section，并在初始化之后丢掉该 section。DriverEntry()和它的帮助函数 DriverInitialize()仅在驱动程序启动时会被调用一次；因此，当它们完成自己的工作后，就可安全的将它们从内存中移除。

现在驱动程序骨架就只剩下了资源脚本---TestDrv.rc，如**列表 3-6** 所示。该文件没有太大价值，因为它仅引用了来自 DrvInfo.h 的宏----DRV_RC_VERSION 以及向导提供的几个数据项来创建一个 VERSIONINFO 资源，另一个宏---DRV_RC_ICON 则等同于将 TestDrv.ico 加入 TesetDrv.sys 的 Resource Section 中的 ICON 声明语句。

```
// TestDrv.rc
// 08-07-2000 <MyName>
// Copyright ?2000 <MyCompany>

#define  _RC_PASS_
#define  _TESTDRV_SYS_
#include "TestDrv.h"

// ====================================================================
// STANDARD RESOURCES
// ====================================================================

DRV_RC_VERSION
DRV_RC_ICON

// ====================================================================
// END OF FILE
// ====================================================================
```

列表 3-6. 驱动程序骨架的资源脚本

3.1.5、设备 I/O 控制

就像在本章开头的简介中提到的，在本书中，我们不会构建某一具体硬件的驱动程序。替代的是，我们将利用功能强大的内核驱动程序来研究 Windows 2000 的秘密。从实际结果来看，驱动程序的强大之处在于它们能在 CPU 的最高特权级别上运行。这意味着内核驱动可以访问所有的系统资源，可以读取所有的内存空间，而且也被允许执行 CPU 的特权指令，如，读取 CPU 控制寄存器的当前值等。而处于用户模式下的程序如果试图从内核空间中读取一个字节或者试图执行像 MOV EAX,CR3 这样的汇编指令都会被立即终止掉。不过，这种强大的底线是驱动程序的一个很小的错误就会让整个系统崩溃。即使是非常小的错误发生，也会让系统蓝屏，因此开发内核程序的人员必须比 Win32 应用程序或 DLL 的开发人员更加仔细的处理

错误。还记得我们在第一章里使用的导致系统蓝屏的 Windows 2000 Killer device driver 吗？它所作的一切只是触及了虚拟内存地址 0x00000000，然后就---Boom！！！你应该意识到在开发内核驱动时，你会比以往更频繁的重启你的机器。

在随后章节中，我给出的驱动程序代码将采用称为设备 I/O 控制（IOCTL）的技术，以允许用户模式下的代码实现一定程序的“远程控制”。如果应用程序需要访问在用户模式下无法触及的系统资源，那内核驱动程序将可很好的完成此项工作，而 IOCTL 则是联系二者的桥梁。事实上，IOCTL 并不是 Windows 2000 采用的新技术。即使旧的操作系统---DOS 2.11 也具有这种能力，0x44 函数及其子函数构成了 DOS 的 IOCTL。基本上，IOCTL 是通过控制通路和设备通讯的一中手段，控制通路在逻辑上独立于数据通路。想象一个硬盘设备通过其主数据通路传递磁盘扇区中的内容。如果客户想获取当前设备使用的媒体信息，它就必须使用另一个不同的通路。例如，DOS 函数 0x44，其子函数 0x0d、0x66 构成了 DOS 的 IOCTL，调用这些函数就可读取磁盘的 32 位连续数据（参考 Brown and Kyle 1991，1993）。

设备 I/O 控制根据要控制的设备，可以有多种实现方式。就其一般形式来说，IOCTL 有如下几类：

- 客户端通过一个特殊的进入点来控制设备。在 DOS 中，这个进入点为 INT 21h、函数号 0x44。在 Windows 2000 中，则通过 Kernel32.dll 导出的 Win32 函数 DeviceIoControl()。
- 客户端通过提供设备的唯一标识符、控制代码以及一个存放输入数据的缓冲区、一个存放输出数据的缓冲区来调用 IOCTL 的进入点。对于 Windows 2000，设备标识符是成功打开的设备的句柄（HANDLE）。
- 控制代码用于告诉目标设备的 IOCTL 分派器（dispatcher），客户端请求的是哪一个控制函数。
- 输入缓冲区中可包含任意地附加数据，设备可能需要这些数据来完成客户所请求的操作。
- 客户所请求的操作产生的任何数据，都会保存在客户端提供的输出缓冲区中。
- IOCTL 操作的整体结果通过返回给客户端的状态代码来表示

很显然这是一种强大的通用机制，这种机制可以适用于很大范围的控制请求。例如，应用程序在访问系统内核所占用的内存空间时会被禁止，这是因为当程序触及该内存空间时会立即抛出一个异常，不过程序可以通过加载一个内核驱动程序来完成此项工作，这样就可避免出现异常。交互的两个模块都需遵循 IOCTL 协议来管理数据的传输。例如，程序可能通过

给驱动程序发送控制代码 0x80002000 来读取内存或发送 0x80002001 来向内存中写入数据。对于读取请求，IOCTL 输入缓冲区或许要提供基地址和要读取的字节数。内核驱动程序能获取这些请求并通过控制代码来判断是读取操作还是写入操作。对于读取请求，内核驱动程序会将请求的内存范围内的数据复制到调用者提供的输出缓冲区中，如果输出缓冲区足够容纳这些数据，则返回成功代码。对于写入请求，驱动程序会将输入缓冲区中的数据复制到指定的内存中（该内存的起始位置也由输入缓冲区指定）。在第四章，我将提供一个 Memory Spy 的示列代码。

现在，可以看出 IOCTL 是 Win32 应用程序的一种后门，通过 IOCTL，程序可以执行几乎所有的操作，而在此之前，这些操作仅允许特权模块执行。当然，这需要首先编写一个特权级的模块，但是，一旦你拥有一个运行于系统中的 Spy 模块，一切就变得很简单了。本书的两个目标是：详细展示如何编写内核模式的驱动程序以及一个可以完成很多让人惊异的事的驱动程序的示例代码。

3.1.6、Windows 2000 的 Killer Device

在开始更高级的驱动程序工程之前，让我们先看看一个非常简单的驱动程序。在第一章中，我介绍了 Windows 2000 的 Killer Device----w2k_kill.sys，它被设计为引发一个良性的系统崩溃。这个驱动程序并不需要**示例 3-3** 中的大多数代码，因为它在有机会收到第一个 I/O 请求包之前就会使系统崩溃。**示例 3-7** 给出了它的实现代码。这里没有给出 w2k_kill.h 文件，因为它不不包含任何我们感兴趣的代码。

示列 3-7 中的代码没有在 DriverEntry()中执行初始化操作，因为系统会在 DriverEntry()返回前就崩溃，所以没有必要进行这些额外的工作。

```
#define  _W2K_KILL_SYS_
#include <ddk\ntddk.h>
#include "w2k_kill.h"
// ============================================================
// DISCARDABLE FUNCTIONS
// ============================================================

NTSTATUS DriverEntry (PDRIVER_OBJECT  pDriverObject,
```

```
                        PUNICODE_STRING pusRegistryPath);


#ifdef ALLOC_PRAGMA
#pragma alloc_text (INIT, DriverEntry)
#endif


// ====================================================================
// DRIVER INITIALIZATION
// ====================================================================


NTSTATUS DriverEntry (PDRIVER_OBJECT  pDriverObject,
                        PUNICODE_STRING pusRegistryPath)
    {
    return *((NTSTATUS *) 0);
    }


// ====================================================================
// END OF PROGRAM
// ====================================================================
```

示列 3-7.　一个小巧的系统崩溃者

3.2、加载/卸载驱动程序

在完成一个内核驱动程序之后，你可能会想立即执行它。怎么做呢？典型的做法是，在系统启动时加载驱动程序并执行之。但这是不是就意味着我们每次更新驱动程序后，都必须重新启动系统呢？很幸运,这并不是必须的。Windows 2000 的一个特色就是提供了一个 Win32 接口以允许在运行时加载或卸载驱动程序。这是由服务控制管理器（Service Control Manager，SCM）完成的，下面的将详细介绍它的用法。

3.2.1、服务控制管理器

“服务控制管理器”这个名字容易让人误解，因为它暗示该组件仅用于服务的管理。服务（Service）是 Windows 2000 的一类非常强大的模块，它们在后台运行配套的程序，并且不需要用户交互（也就是说没有常见的用户界面或者控制台）。换句话说，一个服务就是一个始终运行于系统中的 Win32 进程，即使没有用户登陆进来也如此。尽管开发服务是一个令人兴奋的话题，但它并不属于本书的范畴。想进一步了解服务的开发，请阅读 Windows Developer’s Journal（WDJ）（Tomlinson 1996a）中 Paula Tomlinson 提供的非常不错的教程，以及随后在她的 WDJ 专栏----Understanding NT 中发表的有关服务的论文。

SC 管理器（即服务控制管理器）可以控制服务和驱动程序。为了简单起见，我在这里使用“服务”一词来代表 SC 管理器控制的所有对象，这包括严格意义上的服务和内核驱动程。SC 的接口对于 Win32 程序是可用的，它由 Win32 子系统组件----advapi32.dll 提供，这个 DLL 还提供了很多有趣的 API 函数。**表 3-3** 给出了用于加载、控制和卸载服务的 API 函数的名称，同时还给出了简单的描述。在你可以加载或访问任何服务之前，你必须获取 SC 管理器的句柄（通过调用 OpenSCManager()），在随后的讨论中，该句柄将被称为：管理器句柄。CreateService()和 OpenService()都需要此句柄，而这些函数返回的句柄将被称为:服务句柄。这种类型的句柄可以传递给需要引用一个服务的函数，如 ControlService()、DeleteService()和 StartService()。这两种类型的 SC 句柄都通过 CloseServiceHandle()函数来释放。

名　称	描　　述
CloseServiceHandle	关闭来自 OpenSCManager()、CreateService()或 OpenService()的句柄
ControlService	停止、暂停、继续、查询或通知已加载的服务/驱动程序
CreateService	加载一个服务/驱动程序
DeleteService	卸载一个服务/驱动程序
OpenSCManager	获取 SC 管理器的句柄
OpenService	获取一个已加载的服务/驱动程序的句柄
QueryServiceStatus	查询一个服务/驱动程序的属性和当前状态
StartService	启动一个已加载的服务/驱动程序

表 3-3.　基本的服务控制函数

加载和运行一个服务需要执行的典型操作步骤：

- 调用 OpenSCManager()以获取一个管理器句柄
- 调用 CreateService()来向系统中添加一个服务
- 调用 StartService()来运行一个服务
- 调用 CloseServiceHandle()来释放管理器或服务句柄

要确保当一个错误发生时，要回滚到最后一个成功的调用，然后再开始。例如，你在调用 StartService()时 SC 管理器报告了一个错误，你就需要调用 DeleteService()。否则，服务将保持在一个非预期的状态。另一个使用 SC 管理器 API 易犯的错误是，必须为 CreateService()函数提供可执行文件的全路径名，否则，如果该函数在当前目录中没有找到可执行文件的话，就会失败。因此，你应该使用 Win32 函数---GetFullPathName()来规格化传递给 CreateService()的所有文件名，除非可以保证它们已经是全路径的。

3.2.2、高层的驱动程序管理函数

为了更容易的和 SC 管理器进行交互，本书附带的 CD 提供了多个更高级的外包函数，这些函数屏蔽了原有的一些不方便的特殊要求。这些函数是本书提供的庞大的 Windows 2000 工具库（位于随书 CD 中的\src\w2k_lib）中的一部分。w2k_lib.dll 导出的所有函数都有一个全局的名字前缀 w2k，服务和驱动程序管理函数都使用 w2kService 前缀。**列表 3-8** 给出了本书提供的工具库中实现的加载、控制和卸载服务/驱动程序的函数的细节。

```
// ========================================================================
// SERVICE/DRIVER MANAGEMENT
// ========================================================================


SC_HANDLE WINAPI w2kServiceConnect (void)
    {
    return OpenSCManager (NULL, NULL, SC_MANAGER_ALL_ACCESS);
    }


// ------------------------------------------------------------------------
```

```
SC_HANDLE WINAPI w2kServiceDisconnect (SC_HANDLE hManager)
    {
    if (hManager != NULL) CloseServiceHandle (hManager);
    return NULL;
    }

// ------------------------------------------------------------

SC_HANDLE WINAPI w2kServiceManager (SC_HANDLE  hManager,
                                    PSC_HANDLE phManager,
                                    BOOL       fOpen)
    {
    SC_HANDLE hManager1 = NULL;

    if (phManager != NULL)
        {
        if (fOpen)
            {
            if (hManager == NULL)
                {
                *phManager = w2kServiceConnect ();
                }
            else
                {
                *phManager = hManager;
                }
            }
        else
```

```
            {
            if (hManager == NULL)
                {
                *phManager = w2kServiceDisconnect (*phManager);
                }
            }
        hManager1 = *phManager;
        }
    return hManager1;
    }

// ------------------------------------------------------------

SC_HANDLE WINAPI w2kServiceOpen (SC_HANDLE hManager,
                                 PWORD     pwName)
    {
    SC_HANDLE hManager1;
    SC_HANDLE hService = NULL;

    w2kServiceManager (hManager, &hManager1, TRUE);

    if ((hManager1 != NULL) && (pwName != NULL))
        {
        hService = OpenService (hManager1, pwName,
                                SERVICE_ALL_ACCESS);
        }
    w2kServiceManager (hManager, &hManager1, FALSE);
    return hService;
    }
```

```
// ------------------------------------------------------------------

BOOL WINAPI w2kServiceClose (SC_HANDLE hService)
    {
    return (hService != NULL) && CloseServiceHandle (hService);
    }

// ------------------------------------------------------------------

BOOL WINAPI w2kServiceAdd (SC_HANDLE hManager,
                           PWORD     pwName,
                           PWORD     pwInfo,
                           PWORD     pwPath)
    {
    SC_HANDLE hManager1, hService;
    PWORD     pwFile;
    WORD      awPath [MAX_PATH];
    DWORD     n;
    BOOL      fOk = FALSE;

    w2kServiceManager (hManager, &hManager1, TRUE);

    if ((hManager1 != NULL) && (pwName != NULL) &&
        (pwInfo    != NULL) && (pwPath != NULL) &&
        (n = GetFullPathName (pwPath, MAX_PATH, awPath, &pwFile)) &&
        (n < MAX_PATH))
        {
        if ((hService = CreateService (hManager1, pwName, pwInfo,
```

```
                                        SERVICE_ALL_ACCESS,
                                        SERVICE_KERNEL_DRIVER,
                                        SERVICE_DEMAND_START,
                                        SERVICE_ERROR_NORMAL,
                                        awPath, NULL, NULL,
                                        NULL, NULL, NULL))
            != NULL)
            {
            w2kServiceClose (hService);
            fOk = TRUE;
            }
        else
            {
            fOk = (GetLastError () ==
                   ERROR_SERVICE_EXISTS);
            }
        }
    w2kServiceManager (hManager, &hManager1, FALSE);
    return fOk;
    }

// ------------------------------------------------------------

BOOL WINAPI w2kServiceRemove (SC_HANDLE hManager,
                              PWORD     pwName)
    {
    SC_HANDLE hService;
    BOOL      fOk = FALSE;
```

```
    if ((hService = w2kServiceOpen (hManager, pwName)) != NULL)
        {
        if (DeleteService (hService))
            {
            fOk = TRUE;
            }
        else
            {
            fOk = (GetLastError () ==
                    ERROR_SERVICE_MARKED_FOR_DELETE);
            }
        w2kServiceClose (hService);
        }
    return fOk;
    }

// ------------------------------------------------------------------

BOOL WINAPI w2kServiceStart (SC_HANDLE hManager,
                             PWORD     pwName)
    {
    SC_HANDLE hService;
    BOOL      fOk = FALSE;

    if ((hService = w2kServiceOpen (hManager, pwName)) != NULL)
        {
        if (StartService (hService, 1, &pwName))
            {
            fOk = TRUE;
```

```
            }
        else
            {
            fOk = (GetLastError () ==
                    ERROR_SERVICE_ALREADY_RUNNING);
            }
        w2kServiceClose (hService);
        }
    return fOk;
    }

// ------------------------------------------------------------

BOOL WINAPI w2kServiceControl (SC_HANDLE hManager,
                               PWORD     pwName,
                               DWORD     dControl)
    {
    SC_HANDLE      hService;
    SERVICE_STATUS ServiceStatus;
    BOOL           fOk = FALSE;

    if ((hService = w2kServiceOpen (hManager, pwName)) != NULL)
        {
        if (QueryServiceStatus (hService, &ServiceStatus))
            {
            switch (ServiceStatus.dwCurrentState)
                {
                case SERVICE_STOP_PENDING:
                case SERVICE_STOPPED:
```

```
                    {
                    fOk = (dControl == SERVICE_CONTROL_STOP);
                    break;
                    }
                case SERVICE_PAUSE_PENDING:
                case SERVICE_PAUSED:
                    {
                    fOk = (dControl == SERVICE_CONTROL_PAUSE);
                    break;
                    }
                case SERVICE_START_PENDING:
                case SERVICE_CONTINUE_PENDING:
                case SERVICE_RUNNING:
                    {
                    fOk = (dControl == SERVICE_CONTROL_CONTINUE);
                    break;
                    }
                }
            }
        fOk = fOk ||
              ControlService (hService, dControl, &ServiceStatus);

        w2kServiceClose (hService);
        }
    return fOk;
    }

// --------------------------------------------------------------
```

```
BOOL WINAPI w2kServiceStop (SC_HANDLE hManager,
                            PWORD     pwName)
    {
    return w2kServiceControl (hManager, pwName,
                              SERVICE_CONTROL_STOP);
    }

// ------------------------------------------------------------------

BOOL WINAPI w2kServicePause (SC_HANDLE hManager,
                             PWORD     pwName)
    {
    return w2kServiceControl (hManager, pwName,
                              SERVICE_CONTROL_PAUSE);
    }

// ------------------------------------------------------------------

BOOL WINAPI w2kServiceContinue (SC_HANDLE hManager,
                                PWORD     pwName)
    {
    return w2kServiceControl (hManager, pwName,
                              SERVICE_CONTROL_CONTINUE);
    }

// ------------------------------------------------------------------

SC_HANDLE WINAPI w2kServiceLoad (PWORD pwName,
                                 PWORD pwInfo,
```

```
                                    PWORD pwPath,
                                    BOOL  fStart)
    {
    BOOL      f0k;
    SC_HANDLE hManager = NULL;


    if ((hManager = w2kServiceConnect ()) != NULL)
        {
        f0k = w2kServiceAdd (hManager, pwName, pwInfo, pwPath);


        if (f0k && fStart)
            {
            if (!(f0k = w2kServiceStart (hManager, pwName)))
                {
                w2kServiceRemove (hManager, pwName);
                }
            }
        if (!f0k)
            {
            hManager = w2kServiceDisconnect (hManager);
            }
        }
    return hManager;
    }


// ------------------------------------------------------------------


SC_HANDLE WINAPI w2kServiceLoadEx (PWORD pwPath,
                                   BOOL  fStart)
```

```
{
PVS_VERSIONDATA pvvd;
PWORD           pwPath1, pwInfo;
WORD            awName [MAX_PATH];
DWORD           dName, dExtension;
SC_HANDLE       hManager = NULL;

if (pwPath != NULL)
    {
    dName = w2kPathName (pwPath, &dExtension);

    lstrcpyn (awName, pwPath + dName,
              min (MAX_PATH, dExtension - dName + 1));

    pwPath1 = w2kPathEvaluate (pwPath,  NULL);
    pvvd    = w2kVersionData  (pwPath1, -1);

    pwInfo  = ((pvvd != NULL) && pvvd->awFileDescription [0]
               ? pvvd->awFileDescription
               : awName);

    hManager = w2kServiceLoad (awName, pwInfo, pwPath1, fStart);

    w2kMemoryDestroy (pvvd);
    w2kMemoryDestroy (pwPath1);
    }
return hManager;
}
```

```
// ------------------------------------------------------------


BOOL WINAPI w2kServiceUnload (PWORD     pwName,
                              SC_HANDLE hManager)
    {
    SC_HANDLE hManager1 = hManager;
    BOOL      fOk       = FALSE;


    if (pwName != NULL)
        {
        if (hManager1 == NULL)
            {
            hManager1 = w2kServiceConnect ();
            }
        if (hManager1 != NULL)
            {
            w2kServiceStop (hManager1, pwName);
            fOk = w2kServiceRemove (hManager1, pwName);
            }
        }
    w2kServiceDisconnect (hManager1);
    return fOk;
    }


// ------------------------------------------------------------


BOOL WINAPI w2kServiceUnloadEx (PWORD     pwPath,
                                SC_HANDLE hManager)
    {
```

```
    DWORD dName, dExtension;
    WORD  awName [MAX_PATH];
    PWORD pwName = NULL;

    if (pwPath != NULL)
        {
        dName = w2kPathName (pwPath, &dExtension);

        lstrcpyn (pwName = awName, pwPath + dName,
                  min (MAX_PATH, dExtension - dName + 1));
        }
    return w2kServiceUnload (pwName, hManager);
    }

// ------------------------------------------------------------------
```

列表 3-8. 服务和驱动管理库函数

表 3-4 列出了定义于**列表 3-8** 中的函数，同时还给出了简短的介绍。其中的一些函数的名字，如 w2kServiceStart()和 w2kServiceControl()和 SC 管理器的原生 API 函数---StartService()和 ControlService()比较类似。这没有什么不一致，在这些外包函数的核心位置都能找到对这些原生函数的调用。外包函数和原生函数的主要区别在于：StartService()和 ControlService()的操作对象是服务句柄，而 w2kServiceOpen()和 w2kServiceClose()则是服务的名称。这些名字会在内部调用 w2kServiceOpen()和 w2kServiceClose()转化为对应的句柄，w2kServiceOpen()和 w2kServiceClose()会依次调用 OpenService()和 CloseServiceHandle()。

名　称	描　　述
w2kServiceAdd	向系统中增加一个服务/驱动程序
w2kServiceClose	关闭一个服务句柄
w2kServiceConnect	连接到服务控制管理器
w2kServiceContinue	继续执行暂停的服务/驱动程序

w2kServiceControl	停止、暂停、继续、查询或通知一个已加载的服务/驱动程序
w2kServiceDisconnect	断开和服务控制管理器的连接
w2kServiceLoad	加载和启动（可选的）一个服务/驱动程序
w2kServiceLoadEx	加载和启动（可选的）一个服务/驱动程序（自动生成名称）
w2kServiceManager	打开/关闭一个临时的服务控制管理器句柄
w2kServiceOpen	获取一个已加载的服务/驱动程序的句柄
w2kServicePause	暂停一个正在运行的服务/驱动程序
w2kServiceRemove	从系统中移除一个服务/驱动程序
w2kServiceStart	启动一个已加载的服务/驱动程序
w2kServiceStop	停止一个正在运行的服务/驱动程序
w2kServiceUnload	停止和卸载一个服务/驱动程序
w2kServiceUnloadEx	停止和卸载一个服务/驱动程序（自动生成名称）

表 3-4.　w2k_lib.dll 提供的 SC 管理器的外包函数

表 3-4 中函数的典型用法都需遵循如下的指导方针：

- 使用 w2kServiceLoad()或 w2kServiceLoadEx()来加载一个服务。后一个函数会根据可执行文件的路径和版本信息自动生成服务的显示名称。逻辑变量 fStart 用来确定是否在成功加载服务后自动执行该服务。在成功的情况下，该函数会为后续的调用返回一个管理器句柄。如果服务已经加载或服务已经开始运行而 fStart 为 TRUE，调用该函数不会返回任何错误。但如果发生错误，如有必要，发生错误的服务会被自动卸载。
- 使用 w2kServiceUnload()和 w2kServiceUnloadEx()来卸载一个服务，这需要用到 w2kServiceLoad()或 w2kServiceLoadEx()返回的管理器句柄。w2kServiceUnloadEx()会根据可执行文件的路径自动生成服务名称。如果你已经关闭了管理器句柄，可使用 w2kServiceConnect()来或取一个新的管理器句柄或者简单的传递一个 NULL（这表示使用临时的管理器句柄）。管理器句柄会由 w2kServiceUnload()自动关闭。如果服务已经有删除标志，则不会返回任何错误，但并不会立即删除服务，这是因为打开的设备句柄还存在着。
- 使用 w2kServiceStart()、w2kServiceStop()、w2kServicePause()或 w2kServiceContinue()来控制一个服务。这些函数也需要使用 w2kServiceLoad()

或 w2kServiceLoadEx()返回的管理器句柄。如果你提供一个值为 NULL 的管理器句柄，则使用临时管理器句柄。如果指定的服务已处于所要求的状态，则不会返回任何错误。

- 调用 w2kServiceDisconnect()来关闭一个管理器句柄。你可以在任何时候调用 w2kServiceConnect()来获取一个管理器句柄。

w2kServiceLoadEx()是一个十分强大的函数。它会构建自动加载一个服务时所需的全部参数，但你要提供可执行文件的路径。SC 管理器的 CreateService()函数所需要的服务名称将从可执行文件名（会去掉文件的扩展名）中派生出来。为了给新创建的服务构建一个适当的用于显示名称，w2kServiceLoadEx()会尝试从文件的版本信息中读取 FileDescription 字符串。如果可执行文件中不包含版本信息，或者 FileDescription 字符串不可用，则将使用缺省的服务名称。

和 w2kServiceLoad()不同，w2kServiceLoadEx()支持路径中的环境变量。换句话说，如果路径字符串中包含如*%SystemRoot%*或*%TEMP%*这样的子串，它们会被相应系统变量的当前值替换掉。w2kServiceUnloadEx()是 w2kServiceLoadEx()的很好的搭档，它会从提供的路径中提取服务的名称，与前面提及的展开过程类似，并将提取出来的服务名称传递给 w2kServiceUnload()。这两个函数是需要加载/卸载第三方设备驱动的应用程序的理想搭档，只需提供这些驱动的全路径即可。本书的光盘中包含一个这样的示例程序。

控制台模式的工具-----w2k_load.exe 是一个通用的内核驱动程序加载/卸载器，它为 w2kServiceLoadEx()和 w2kServiceUnloadEx()提供了简单的命令行接口。其源代码可以在随书 CD 的\src\w2k_load 目录下找到。**列表 3-9** 给出了相关的代码，该工具仅是一种示意性的实现。因为大量的工作都是由 w2k_lib.dll 中的 w2kServiceLoadEx()和 w2kServiceUnloadEx()完成的。

```
// ============================================================
// GLOBAL STRINGS
// ============================================================

WORD awUsage  [] =
    L"\r\n"
    L"Usage: " SW(MAIN_MODULE) L" <driver path>\r\n"
```

```
    L"       " SW(MAIN_MODULE) L" <driver path> %s\r\n"
    L"       " SW(MAIN_MODULE) L" <driver name> %s\r\n";


WORD awUnload [] = L"/unload";


WORD awOk     [] = L"OK\r\n";
WORD awError  [] = L"ERROR\r\n";


// ================================================================
// COMMAND HANDLERS
// ================================================================


BOOL WINAPI DriverLoad (PWORD pwPath)
    {
    SC_HANDLE hManager;
    BOOL      fOk = FALSE;


    _printf (L"\r\nLoading \"%s\" ... ", pwPath);


    if ((hManager = w2kServiceLoadEx (pwPath, TRUE)) != NULL)
        {
        w2kServiceDisconnect (hManager);
        fOk = TRUE;
        }
    _printf (fOk ? awOk : awError);
    return fOk;
    }


// ----------------------------------------------------------------
```

```
BOOL WINAPI DriverUnload (PWORD pwPath)
    {
    BOOL fOk = FALSE;

    _printf (L"\r\nUnloading \"%s\" ... ", pwPath);

    fOk = w2kServiceUnloadEx (pwPath, NULL);

    _printf (fOk ? awOk : awError);
    return fOk;
    }

// ===================================================================
// MAIN PROGRAM
// ===================================================================

DWORD Main (DWORD argc, PTBYTE *argv, PTBYTE *argp)
    {
    _printf (atAbout);

    if (argc == 2)
        {
        DriverLoad (argv [1]);
        }
    else
        {
        if ((argc == 3) && (!lstrcmpi (argv [2], awUnload)))
            {
```

```
            DriverUnload (argv [1]);
            }
        else
            {
            _printf (awUsage, awUnload, awUnload);
            }
        }
    return 0;
    }

// ========================================================================
// END OF PROGRAM
// ========================================================================
```

列表 3-9.　加载/卸载设备驱动

表 3-4 中剩余的库函数在更低一级的层面上工作，它们都在 w2k_lib.dll 内部使用。当然，如果你喜欢的话，你也可以从你的程序里调用它们。从**列表 3-8** 给出的它们的源代码中，可以很容易得出它们的使用方式。

3.2.3、枚举服务和驱动

有时很有必要知道系统当前加载了那个服务或驱动，以及它们现在处于什么状态。为了实现这一目的，SC 管理器提供了另一个名为 EnumServiceStatus()的强大函数。该函数需要一个管理器句柄和一个类型为 ENUM_SERVICE_STATUS 的数组，该数组中将包含有关当前已加载的服务或驱动的信息。这个列表可以根据服务/驱动的类型和状态来过滤。如果调用者提供的缓冲区不能一次性的容纳所有项目，可反复调用该函数直到获取所有的项目。

不过很难预先计算出所需的缓冲区大小，这是因为缓冲区必须为那些大小未知的字符串提供额外的空间，这些字符串由 ENUM_SERVICE_STATUS 的成员引用。幸运的是，EnumServiceStatus()会返回剩余的项目所需的字节数，因此可以通过反复尝试得出确定的缓冲区大小。**列表 3-10** 给出了 SERVICE_STATUS 和 ENUM_SERVICE_STATUS 结构的定义。这些声明位于 Win32 头文件 WinSvc.h 中。

```
typedef struct  _SERVICE_STATUS
{
    DWORD  dwServiceType;
    DWORD  dwCurrentState;
    DWORD  dwControlAccepted;
    DWORD  dwWin32ExitCode;
    DWORD  dwServiceSpecificExitCode;
    DWORD  dwCheckPoint;
    DWORD  dwWaitHint;
} SERVICE_STATUS, *LPSERVICE_STATUS;

typedef struct  _ENUM_SERVICE_STATUS
{
    LPTSTR  lpServiceName;
    LPTSTR  lpDisplayName;
    SERVICE_STATUS  ServiceStatus;
} ENUM_SERVICE_STATUS;
```

列表 3-10　SERVICE_STATUS 和 ENUM_SERVICE_STATUS 结构的定义

列表 3-11 给出的 w2kServiceList()函数是来自 w2k_lib.dll 工具库的另一个好东东。它省略了前面提到的动作，并返回一个随时可用的结构，该结构中包含所有请求的数据以及一对扩展结构。该函数将返回一个指向 W2K_SERVICES 结构的指针，该结构定义于 w2k_lib.h，在**列表 3-11** 的顶部给出了其定义。随 ENUM_SERVICE_STATUS 结构数组 aess[]，W2K_SERVICES 结构体还包含四个附加成员。dEntries 表示向状态数组中复制了多少项目，dBytes 表示返回的 W2K_SERVICES 结构的大小。dDisplayName 和 dServiceName 被分别设置为 aess[]中的 lpDisplayName 和 lpServiceName 字符串的最大长度。这些值将提供很大的方便，尤其是当你编写一个控制台模式的程序，在屏幕上输出服务/驱动列表，并要求名称列采用合适的对齐方式。

为了提供精确的系统快照，w2kServiceList()试图通过一次调用 EnumServiceStatus()来获取所有的项目。为此目的，该函数首先提供一个长度为 0 的缓冲区，这通常会导致返回

ERROR_MORE_DATA 错误代码。在此种情况下，EnumServiceStatus()将返回需要的缓冲区大小，然后按照此大小分配适当的缓冲区，然后再次调用 EnumServiceStatus()。此时，EnumServiceStatus()应该返回成功。不过，这存在一个很小的概率事件---在两次调用 EnumServiceStatus()之间另一个项目可能会被增加到列表中。因此，将会在一个循环中重复这一过程直到所有的一切都正确或者一个非 ERROR_MORE_DATA 的错误返回。

```
// ------------------------------------------------------------
typedef struct _W2K_SERVICES
    {
    DWORD               dEntries;     // number of entries in aess[]
    DWORD               dBytes;       // overall number of bytes
    DWORD               dDisplayName; // maximum display name length
    DWORD               dServiceName; // maximum service name length
    ENUM_SERVICE_STATUS aess [];      // service/driver status array
    }
    W2K_SERVICES, *PW2K_SERVICES, **PPW2K_SERVICES;


#define W2K_SERVICES_ sizeof (W2K_SERVICES)


#define W2K_SERVICES__(_n) \
        (W2K_SERVICES_ + ((_n) * ENUM_SERVICE_STATUS_))


// ------------------------------------------------------------
PW2K_SERVICES WINAPI w2kServiceList (BOOL fDriver,
                                     BOOL fWin32,
                                     BOOL fActive,
                                     BOOL fInactive)
    {
    SC_HANDLE     hManager;
    DWORD         dType, dState, dBytes, dResume, dName, i;
```

```
PW2K_SERVICES pws = NULL;


if ((pws = w2kMemoryCreate (W2K_SERVICES_)) != NULL)
    {
    pws->dEntries     = 0;
    pws->dBytes       = 0;
    pws->dDisplayName = 0;
    pws->dServiceName = 0;


    if ((fDriver || fWin32) && (fActive || fInactive))
        {
        if ((hManager = w2kServiceConnect ()) != NULL)
            {
            dType   = (fDriver ? SERVICE_DRIVER : 0) |
                      (fWin32  ? SERVICE_WIN32  : 0);


            dState  = (fActive && fInactive
                       ? SERVICE_STATE_ALL
                       : (fActive
                          ? SERVICE_ACTIVE
                          : SERVICE_INACTIVE));


            dBytes  = pws->dBytes;


            while (pws != NULL)
                {
                pws->dEntries     = 0;
                pws->dBytes       = dBytes;
                pws->dDisplayName = 0;
```

```
                pws->dServiceName = 0;


                dResume = 0;


                if (EnumServicesStatus (hManager, dType, dState,
                                        pws->aess, pws->dBytes,
                                        &dBytes, &pws->dEntries,
                                        &dResume))
                    break;


                dBytes += pws->dBytes;
                pws     = w2kMemoryDestroy (pws);


                if (GetLastError () != ERROR_MORE_DATA) break;


                pws = w2kMemoryCreate (W2K_SERVICES_ + dBytes);
                }
            w2kServiceDisconnect (hManager);
            }
        else
            {
            pws = w2kMemoryDestroy (pws);
            }
        }
    if (pws != NULL)
        {
        for (i = 0; i < pws->dEntries; i++)
            {
            dName = lstrlen (pws->aess [i].lpDisplayName);
```

```
                pws->dDisplayName = max (pws->dDisplayName, dName);


                dName = lstrlen (pws->aess [i].lpServiceName);
                pws->dServiceName = max (pws->dServiceName, dName);
                }
            }
        }
    return pws;
    }
```

列表 3-11.　枚举服务/驱动程序

w2kServiceList()需要四个逻辑类型的参数，以确定要返回的列表的内容。通过 fDriver 和 fWin32 参数，你可以分别选择是否包含驱动程序或服务。如果这两个参数都为 TRUE，那么返回的列表将同时包含驱动和服务。fActive 和 fInactive 标志用于控制加于列表上的状态过滤器。。fInactive 参数选择剩余的模块，也就是说，这些模块已经加载但已经停止运行。如果所有的四个参数都为 FALSE，函数返回的 W2K_SERVICES 结构将包含一个空的状态数组。光盘中的示例代码包含一个简单的服务/驱动浏览器，它被设计为 Win32 控制台模式，并依赖于 w2k_lib.dll 中的 w2kServiceList()。它使用 W2K_SERVICES 结构（参见**列表 3-11**）中的 dDisplayName 和 dServiceName 成员来为所有的名称选择合适的水平对齐方式。你可以在光盘的\src\w2k_svc 目录下找到此工具的源代码。其可执行文件对应光盘中的\bin\w2k_svc.exe。**示列 3-4** 列出了在我的机器上运行该工具，列出的所有活动的内核驱动程序（使用命令选项 /drivers　/active）。

```
D:\>w2k_svc /drivers /active

// w2k_svc.exe
// SBS Windows 2000 Service List V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Found 29 active drivers:

   1. Alerter . . . . . Alerter
   2. Computer Browser . . . . . Browser
   3. Creative Service for CDROM Access . . . . . Creative Service
   4. DHCP Client . . . . . Dhcp
   5. Logical Disk Manager . . . . . dmserver
   6. DNS Client . . . . . Dnscache
   7. Event Log . . . . . eventlog
   8. COM+ Event System . . . . . EventSystem
   9  Server . . . . . lanmanserver
  10  Workstation . . . . . lanmanworkstation
  11. TCP/IP NetBIOS Helper Service . . . . . LmHosts
  12. Messenger . . . . . Messenger
  13. Network Connections . . . . . Netman
  14. Removable Storage . . . . . NtmsSvc
  15. Plug and Play . . . . . PlugPlay
  16. IPSEC Policy Agent . . . . . PolicyAgent
  17. Protected Storage . . . . . ProtectedStorage
  18. Remote Access Connection Manager . . . . . RasMan
  19. Remote Registry Service . . . . . RemoteRegistry
  20. Remote Procedure Call (RPC) . . . . . RpcSs
  21. Security Accounts Manager . . . . . SamSs
  22. Task Scheduler . . . . . Schedule
  23. RunAs Service . . . . . seclogon
  24. System Event Notification . . . . . SENS
  25. Print Spooler . . . . . Spooler
  26. Telephony . . . . . TapiSrv
  27. Distributed Link Tracking Client . . . . . TrkWks
  28. Windows Management Instrumentation . . . . . WinMgmt
  29. Windows Management Instrumentation Driver Extensions . Wmi
```

EXAMPLE 3-4. *Running the Service List Utility* w2k_svc.exe

在下一章中，我们将开始开发一个可实际工作的内核驱动程序，它会侦测内核使用的内存，并且会 Crack 基本的内存管理数据结构。这个工程将伴随你阅读第 4、5 和 6 章，在每一章中，该驱动程序都会被加强。最后将得到一个通用的 Windows 2000 Kernel Spy。

四、探索 Windows 2000 的内存管理机制

内存管理对于操作系统来说是非常重要的。本章将全面的纵览 Windows 2000 的内存管理机制以及 4GB 线性地址空间的结构。针对此部分内容，将解释 Intel i386 CPU 家族的虚拟内存寻址及其分页能力，重点将在于 Windows 2000 的内核是如何使用它们的。为了帮助我们对内存的探索，本章提供了一对程序：一个内核模式的驱动程序，该驱动用来收集系统相关的信息，另一个是用户模式的应用程序，该程序将通过设备 I/O 控制来查询来自驱动程序的数据，并在控制台窗口中进行显示。在剩余的章节中将重复使用“Spy Driver”模块来完成其他几个非常有趣的任务（这些任务都需要在内核模式下执行代码）。请坚持阅读完本章的第一部分，因为它将直接面对 CPU 硬件。不过，我仍然希望你不要跳过它，因为虚拟内存管理是一个非常令人兴奋的话题，理解它是如何工作的，将帮助你洞察复杂操作系统（如 Windows 2000）采用的机制。

4.1、Intel i386 内存管理机制

Windows 2000 内核大量使用 Intel i386 CPU 系列提供的保护模式下的虚拟内存管理机制。为了更好的理解 Windows 2000 如何管理它的主内存，最低限度的熟悉 i386 CPU 的架构某些特点就显得尤为重要。Windows 2000 是针对 Pentium 以上 CPU 设计的。不过，这些新的处理器采用的内存管理模型仍源自针对 80386 CPU 的设计，不过当然会加入了一些重要的增强。因此，微软通常标注 Windows NT 和 Windows 2000 的版本为 Intel 处理器“i386”或者“x86”。不要对这些感到困惑，不管你在本书的什么地方遇到 86 或 386，请记住，这只是表示特定的 CPU 架构，而不是特定的处理器版本。

4.1.1、基本的内存布局

Windows 2000 为应用程序和系统代码提供了非常简单的内存布局。由 32 位的 Intel CPU 提供的 4GB 虚拟内存空间被分割为相等的两部分。低于 0x80000000 的内存地址由用户模式下的模块使用，这包括 Win32 子系统，剩余的 2GB 保留给了系统内核。Windows 2000 Advanced Server 还支持通常称为 4GT RAM Tuning 的另一种内存模型，该模型随 Windows NT 4.0 Server

的企业版引入。该模型可提供 3GB 的用户地址空间，另 1GB 保留给内核，通过在 boot.ini 中添加/3GB 选项来启用该模型。

Windows 2000 Advanced Server 和 DataCenter 支持称为：物理地址扩展（Physical Address Extension, PAE）的内存选项，通过在 boot.ini 中加入/PAE 就可允许这种内存方式。该选项采用了某些 Intel CPU 的特性（如，Pentium Pro 处理器）以允许大于 4GB 的物理内存映射到 32 位的地址空间上。在本章中，我将忽略这种特殊的设置。你可阅读微软的基本知识文章 Q171793（微软 2000c）、Intel 的 Pentium 手册（Intel 1999a,1999b,1999c）以及 Windows 2000 DDK 文档（微软 2000f）来获取更多此方面的信息。

4.1.2、内存分段和请求式分页

在深入 i386 架构的技术细节之前，想让我们回到 1978 年，那一年 Intel 发布了 PC 处理器之母：8086。我想将讨论限制到这个有重大意义的里程碑上。如果你打算知道更多，阅读 Robert L.的 80486 程序员参考（Hummel 1992）将是一个很棒的开始。现在看来这有些过时了，因为它没有涵盖 Pentium 处理器家族的新特性；不过，该参考手册中仍保留了大量 i386 架构的基本信息。尽管 8086 能够访问 1MB RAM 的地址空间，但应用程序还是无法“看到”整个的物理地址空间，这是因为 CPU 寄存器的地址仅有 16 位。这就意味着应用程序可访问的连续线性地址空间仅有 64KB，但是通过 16 位段寄存器的帮助，这个 64KB 大小的内存窗口就可以在整个物理空间中上下移动，64KB 逻辑空间中的线性地址作为偏移量和基地址（由 16 位的段寄存器给处）相加，从而构成有效的 20 位地址。这种古老的内存模型仍然被最新的 Pentium CPU 支持，它被称为：实地址模式，通常叫做：实模式。

80286 CPU 引入了另一种模式，称为：受保护的虚拟地址模式，或者简单的称之为：保护模式。该模式提供的内存模型中使用的物理地址不再是简单的将线性地址和段基址相加。为了保持与 8086 和 80186 的向后兼容，80286 仍然使用段寄存器，但是在切换到保护模式后，它们将不再包含物理段的地址。替代的是，它们提供了一个选择器（selector），该选择器由一个描述符表的索引构成。描述符表中的每一项都定义了一个 24 位的物理基址，允许访问 16MB RAM，在当时这是一个很不可思议的数量。不过，80286 仍然是 16 位 CPU，因此线性地址空间仍然被限制在 64KB。

1985 年的 80386 CPU 突破了这一限制。该芯片最终砍断了 16 位寻址的锁链，将线性地址空间推到了 4GB，并在引入 32 位线性地址的同时保留了基本的选择器/描述符架构。幸运

的是，80286 的描述符结构中还有一些剩余的位可以拿来使用。从 16 位迁移到 32 位地址后，CPU 的数据寄存器的大小也相应的增加了两倍，并同时增加了一个新的强大的寻址模型。真正的 32 位的数据和地址为程序员带了实际的便利。事实上，在微软的 Windows 平台真正完全支持 32 位模型是在好几年之后。Windows NT 的第一个版本在 1993 年 7 月 26 日发布，实现了真正意义上的 Win32 API。但是 Windows 3.x 程序员仍然要处理由独立的代码和数据段构成的 64KB 内存片，Windows NT 提供了平坦的 4GB 地址空间，在那儿可以使用简单的 32 位指针来寻址所有的代码和数据，而不需要分段。在内部，当然，分段仍然在起作用，就像我在前面提及的那样。不过管理段的所有责任都被移给了操作系统。

80386 的另一个新特性是在硬件上支持分页，确切的来说是：请求式分页的虚拟内存。这种技术允许一个不同于 RAM 的存储介质----硬盘来为内存提供支持，例如，在允许分页时，CPU 通过将最近最少访问的内存数据置换到备份存储器中，从而为新的数据腾出空间，这样就能访问比可用物理内存更大的内存空间。理论上来说，可以使用此种方式访问 4GB 的连续线性地址空间，提供的备份介质必须足够的大---即使只安装了非常少的物理内存。当然，分页并不是访问内存的最快方式，最好还是能提供尽可能多的物理内存。但是，这是处理大量数据的最好办法，即使这些数据超过了可用物理内存。例如，图形和数据库程序都需要一大块工作内存，如果没有分页机制的话，其中的某些程序就无法在低档的 PC 系统中运行。

80386 分页的模式是将内存划分为 4KB 或 4MB 大小的页。操作系统的设计者可以在二者之间自由的选择，也可混合使用这两个大小的页面。稍后，我会介绍 Windows 2000 采用的混合大小方案：由操作系统使用 4MB 的页面，而 4KB 页面由剩余的代码和数据使用。这些页面由分层结构的页表树管理，该页表树记录当前位于物理内存中的页，同时还记录了每个页是否实际的位于物理内存中。如果指定页已被置换到了硬盘上，而某些模块触及了位于这些页中的地址，CPU 就会产生一个缺页中断（这与外围硬件产生的中断类似）。接下来，位于操作系统内核中的缺页中断处理例程会试图将该页再次调入物理内存，这可能需要将另一块内存中的数据写入硬盘以腾出空间。通常，系统采用最近最少（LRU）算法来确定哪个页可以被置换出去。现在可以很清楚地看到为什么有时将这个过程称为----请求式分页（demand paging）：即，由软件提出请求，然后根据操作系统和应用程序使用的内存的统计数据，将物理内存中的数据移动到后备存储设备中。

由页表提供的间接寻址方式蕴含着很有趣的两件事。第一，程序所使用的地址和 CPU 使用的物理地址总线上的地址之间并没有预设的关系。如果你知道你的程序所使用的数据结构位于某一地址，如，0x00140000，你可能仍然不想知道任何有关这些数据的物理地址的信

息，除非你要检查页表树（page-table tree）。这需要操作系统来决定这些地址之间的映射关系。甚至当前有效的地址转换都是无法预测的，部分的来看，这是分页机制所固有的随机性导致的。幸运的是，在大多数应用程序中，并不需要有关物理地址的知识。不过，对于开发硬件驱动程序的人员来说还是需要某些这方面的知识。分页的另一个隐晦之处是：地址空间并不必须是连续的。实际上，根据页表的内容，4GB 的空间可以包含大量的“空洞”，这些“空洞”既没有映射到物理内存也没有映射到后备存储器中。如果一个应用程序试图读取或写入这样的一个地址，它将立即被系统中止掉。稍后，我会详细的说明 Windows 2000 是如何将可用内存扩展到 4GB 地址空间的。

80486 和 Pentium CPU 使用的分段和分页机制与 80386 很相似，但一些特殊的寻址特性除外，如 Pentium Pro 采用的物理地址扩展（Physical Address Extension, PAE）机制。随同更高的时钟频率一起，Pentium CPU 的另一特性就是其采用的双重指令流水线，这一特性允许它在同一时刻执行两个操作（只要这两个指令不互相依赖）。例如，如果指令 A 修改一个寄存器的值，而与其相邻的指令 B 需要这个修改后的值来进行计算，在 A 完成之前，B 将无法执行。但是如果指令 B 使用另一个寄存器，CPU 就可同时执行这两个指令。Pentium 系列 CPU 采用的多种优化方式为编译器的优化提供了广阔的空间。如果你对这方面的话题很感兴趣，请参考 Rick 的《Inner Loops》（Booth 1997）。

在 i386 的内存管理中，有三类地址非常有名，它们的术语---逻辑、线性和物理地址出现在 Intel 的系统编程手册（Intel 1999c）。

- **逻辑地址**：这是内存地址的精确描述，通常表示为 16 进制：xxxx:YYYYYYYY，这里 xxxx 为 selector，而 YYYYYYYY 是针对 selector 所选择的段地址的线性偏移量。除了指定 xxxx 的具体数值外，还可使用具体的段寄存器的名字来替代之，如 CS（代码段），DS（数据段），ES（扩展段），FS（附加数据段#1），GS（附加数据段#2）和 SS（堆栈段）。这些符号都来自旧的“段:偏移量”风格，在 8086 实模式下使用此种方式来指定“far pointers”（远指针）。
- **线性地址**：大多数应用程序和内核驱动程序都忽略虚拟地址。它们只对虚拟地址的偏移量部分感兴趣，而这一部分通常称为线性地址。此种类型的地址假定了一种默认的分段模型，这种模型由 CPU 的当前段寄存器确定。Windows 2000 使用 flat segmentation（平滑段），此时 CS、DS、ES 和 SS 寄存器都指向相同的线性地址空间；因此，程序可以认为所有的代码、数据和堆栈指针都可安全的相互转化。例如，

在任何时候，堆栈中的一个地址都可以转化为一个数据指针，而不需要关心相应段寄存器的值。

- **物理地址**：仅当 CPU 工作于分页模式时，此种类型的地址才会变得非常“有趣”。本质上，一个物理地址是 CPU 插脚上可测量的电压。操作系统通过设立页表将线性地址映射为物理地址。Windows 2000 所用页表的布局的某些属性，对于调试软件开发人员非常有用，本章稍后将讨论之。

虚拟地址和线性地址的差别多少有些人为的痕迹，在一些文档中会交替的使用这两个词。我会尽力保证使用这一术语的一致性。特别需要注意的是，Windows 2000 假定物理地址有 64 位宽。而 Intel i386 系统通常只有一个 32 位的地址总线。不过，某些 Pentium 系统支持大于 4GB 的物理内存。例如，使用 PAE 模式的 Pentium Pro CPU，这种 CPU 可以将物理地址扩展到 36 位，这样就可访问多大 64GB 的物理内存（Intel 1999c）。因此，Windows 2000 的 API 函数通常使用数据类型 PHYSICAL_ADDRESS 来表示物理地址，PHYSICAL_ADDRESS 实际是 LARGE_INTEGER 结构的别名，如**列表 4-1** 所示。这两种类型都定义在 DDK 头文件 ntdef.h 中。LARGE_INTEGER 实际上是 64 位有符号整数的结构化表示，它可以被解释为一对 32 位数（LowPart 和 HighPart）或一个完整的 64 位数（QuadPart）。LONGLONG 类型等价于 Visual C/C++的原生类型__int64，该类型的无符号表示叫做 ULONGLONG 或 DWORDLONG，它们都依赖基本的无符号类型__int64。

图 4-1 给出了 i386 内存的分段模型，同时说明了逻辑地址和线性地址的关系。为了更清晰些，我将描述符表（descriptor table）和段（segment）画的比较小。实际上，32 位的操作系统通常采用**图 4-2** 所示的分段方案，这就是所谓的平滑内存模型（flat memory model），它采用一个 4GB 大小的段。这种方案的不足是，描述符表变成了段的一部分，从而可以被有足够权限的代码访问到。

```
typedef LARGE_INTEGER  PHYSICAL_ADDRESS, *PPHYSICAL_ADDRESS;


typedef  union _LARGE_INTEGER
{
    struct
    {
        ULONG  LowPart;
```

```
        LONG    HighPart;
    };
    LONGLONG  QuadPart;
} LARGE_INTEGER, *PLARGE_INTEGER;
```

列表 4-1. PHYSICAL_ADDRESS 和 LARGE_INTEGER 结构的定义

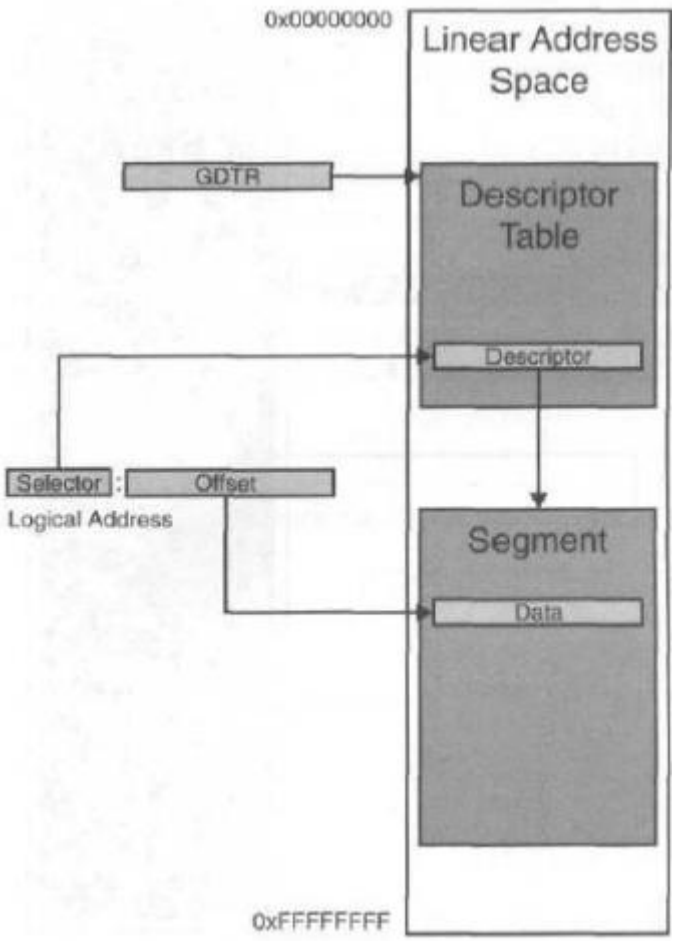


图 4-1. i386 的内存分段

图 4-2 给出的内存模型被 Windows 2000 作为标准的代码、数据和堆栈段，这意味着，所有的逻辑地址将包括 CS、DS、ES 和 SS 段寄存器。FS 和 GS 的处理方式有所不同。Windows 2000 并不使用 GS 寄存器，而 FS 寄存器被专门用来保存位于线性地址空间中的系统数据区域的基地址。因此，FS 的基地址远大于 0，其大小不会超过 4GB。有趣的是，Windows 2000 为用户模式和内核模式分别维护两个不同的 FS 段。稍后我们将详细讨论这一问题。

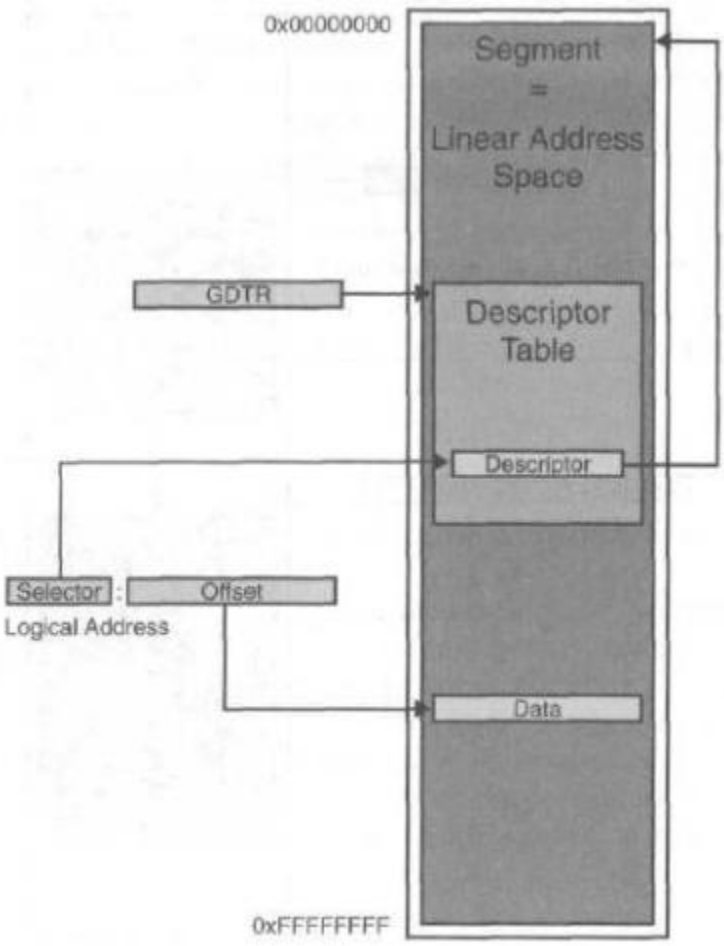
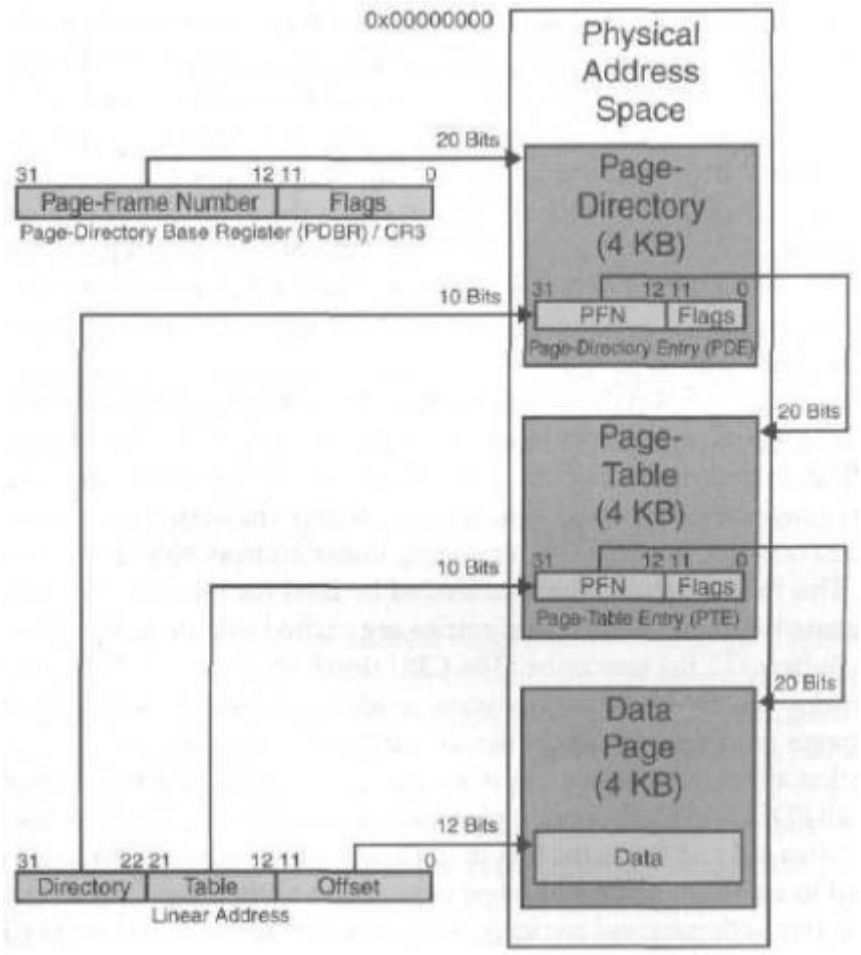


图 4-2.　平滑的 4GB 内存段

在**图 4-1** 和**图 4-2** 中，逻辑地址的 selector 指向描述符表，该描述符表由名为 GDTR 的寄存器指定。这是 CPU 的全局描述符表寄存器，该寄存器可由操作系统设置为任何适当的线性地址。GDT(全局描述符表)的第一项是保留的，该项对应的 selector 叫做“null segment selector”。Windows 2000 将其 GDT 保存在 0x80036000。GDT 可容纳多达 8,19264 位的条目，即其最大值为 64KB。Windows 2000 仅使用开始的 128 个项，并将 GDT 的大小限制为 1,024 字节。随 GDT 一起，i386 CPU 还提供了一个本地描述符表（Local Descriptor Table，LDT）和一个中断描述符表（Interrupt Descriptor Table，IDT），这两个表的起始地址分别保存在 LDTR 和 IDTR 这两个寄存器中。GDTR 和 IDTR 的值是唯一的，CPU 执行的每个任务都采用相同的值，而 LDTR 的值则是任务相关的，LDTR 可容纳一个 16 位的 selector。

图 4-3 示范了复杂的线性地址与物理地址的转换机制，如果在 4KB 分页模式下，并允许请求式分页，i386 的内存管理单元就会采用此种转换机制。图中左上角的页目录基址寄存器（Page-Directory Base Register，PDBR）包含页目录的物理地址。PDBR 由 i386 的 CR3 寄存器保存。仅用该寄存器的高 20 位来寻址。因此，页目录也是以页为边界的。PDBR 的剩

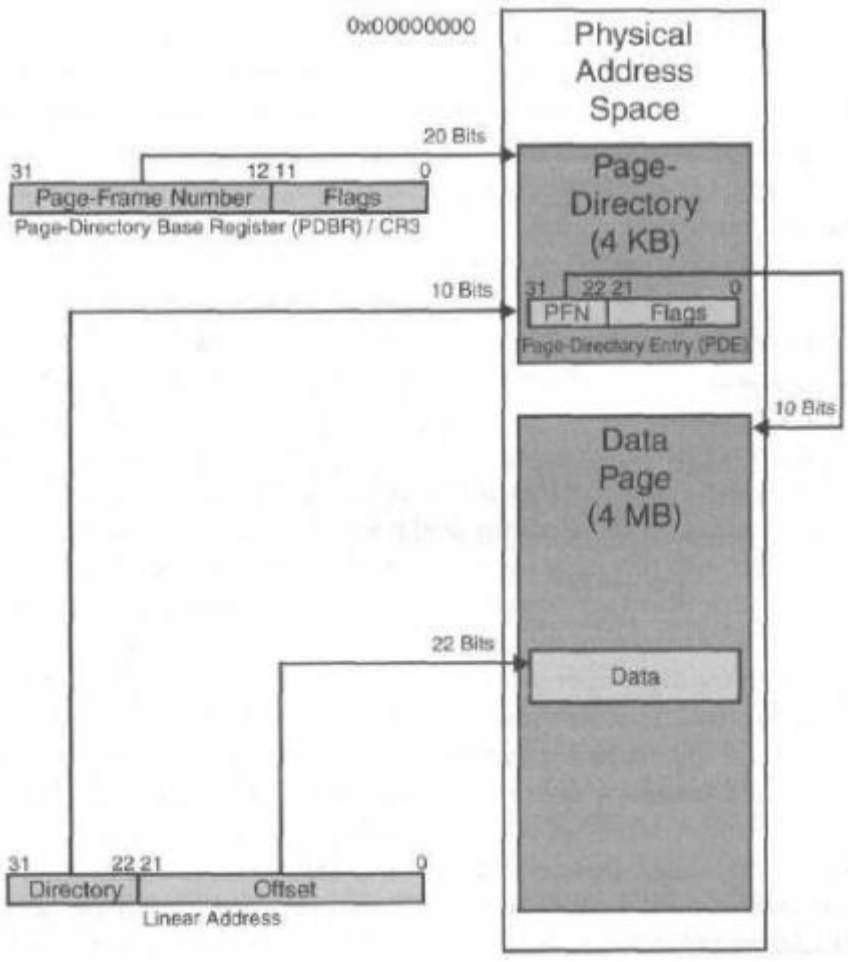
余位作为标志位或保留以便将来扩展使用。页目录占用一个完整的 4KB 页，由包含 1024 个页目录项（Page-Directory Entry）的数组构成，每个页目录项均为 32 位。和 PDBR 类似，每个 PDE 被划分为一个 20 位的页帧计数器（Page-Frame Number，PFN）和一个标志数组。PFN 用来寻址页表。每个页表都是按页对齐的，包含 1024 个页表项（Page-Table Entry，PTE）。每个 PTE 的高 20 位作为一个指针指向一个 4KB 的数据页。通过将线性地址分为三段来实现地址转换：高 10 位用来选择一个 PDE（属于页目录），接下来的 10 个位选择前面所选的 PDE 中的某个 PTE，最后剩下的 12 个位用来指定在数据页中的偏移量，该数据页由前面所选的 PTE 确定。



在 4MB 分页模式下，事情就变得很简单了，这是因为消除了一个间接层，如**图 4-4** 所示。此时，PDBR 仍然指向页目录，但仅使用了每个 PDE 的高 10 位，这是因为目标地址采用 4MB 对齐。因为没有使用页表，这个地址同样也是 4MB 数据页的基地址。所以，此时的线性地址只包含两个部分：10 个位用来选择 PDE，其余的 22 位作为偏移量。4MB 内存方案的开销没有 4KB 那么大，这是因为仅页目录需要附加的内存。这 1024 个 PDE 中的每个都可寻址一个

4MB 页。这足够覆盖整个 4GB 地址空间了。所以，4MB 分页的优势就是可以降低内存管理的开销，但结果就是寻址粒度较大。

4KB 和 4MB 分页模型各有优缺点。幸运的是，操作系统的设计人员不必非要在二者之中选择一个，可以混合使用这两种模型。例如，Windows 2000 在内存范围 0x80000000 --- 0x9FFFFFFF 使用 4MB 大小的页，内核模块 hal.dll 和 ntoskrnl.exe 均被加载到该地址范围内。剩余的线性地址采用 4KB 页来管理。Intel 大力推荐采用这种混合设计，以改进系统性能，这也因为 4KB 和 4MB 的页项（Page Entry）都会被高速缓存到不同的转换后备缓冲区（Translation Lookaside Buffers，TLBs）中，该 TLB 位于 i386 CPU 内部（Intel 1999c，pp.3-22f）。操作系统的内核通常比较大，而且需要常驻内存，因此，如果将它们保存在多个 4KB 页中将会永久性的耗尽宝贵的 TLB 空间。



注意，地址转换的所有步骤都在物理内存中进行。PDBR 和所有的 PDE、PTE 包含的都是物理地址指针。在**图 4-3** 和**图 4-4** 中可找到的线性地址位于左下角，该线性地址将转化为物理页中的偏移量。另一方面，应用程序却必须使用线性地址，它们对物理地址一无所知。不过，通过将页目录和其下属的所有页表映射到线性地址空间可以填补这一不足。在 Windows

2000 和 Windows NT 4.0 中，在线性地址范围 0xC0000000----0xC03FFFFF 可访问所有的 PDE 和 PTE，这是一个采用 4MB 页的线性内存区域。可以简单的通过线性地址的高 20 位来查找与其相关联的 PTE，这个高 20 位作为 32 位 PTE 数组的索引，PTE 数组起始于 0xC0000000。例如，地址 0x00000000 表示的 PTE 位于 0xC0000000。假定有一线性地址 0x80000000，通过将该地址右移 12 位，可得到 0x80000（即该地址的高 20 位），因为每个 PTE 占用 4 个字节，所以目标 PTE 的地址为：0xC0000000+（4*0x80000）=0xC0200000。这样的结果看起来很有趣，线性地址将 4GB 地址空间划分为相等的两部分，又映射为一个 PTE 的地址，从而将 PTE 数组也划分为了相等的两部分。

现在，让我们更进一步，通过 PTE 自身来计算数据项在 PTE 数组中的地址。常规的映射公式为：（（LinearAddress >> 12）*4）+0xC0000000。LinearAddress 取值范围为：0xC0000000----0xC0300000。位于线性地址 0xC0300000 的数据项指向 PTE 数组在物理内存中的起始位置。现在回去看一下**图 4-3**，开始于地址 0xC0300000 的 1024 个数据项肯定是页目录！这种特殊的 PDE、PTE 排列方式被多个内存管理函数使用，这些函数由 ntoskrnl.exe 导出。例如，有文档记载的 API 函数 MmIsAddressValid()和 MmGetPhysicalAddress()使用 32 位的线性地址来查找其 PDE，如可用，还会查找其 PTE，并会检查它们的内容。MmIsAddressValid()简单的检验目标页是否位于物理内存中。如果测试失败，就意味着线性地址或者无效或者该地址引用的页已经被置换到了后备存储器(由系统页面文件集表示)中。MmGetPhysicalAddress()首先从线性地址中提取相应的页帧计数器（PFN），该 PFN 就是与其相关的物理内存页（该页将按照页大小进行划分）的基地址。接下来，它通过线性地址中剩余的 12 个位，来计算在物理页中的偏移量，最后将 PFN 指出的物理页基地址和前面算出的偏移量相加即可得到该线性地址对应的物理地址。

更彻底的检查 MmGetPhysicalAddress()的实现方式，会发现 Windows 2000 内存布局的另一个有趣的特性。MmGetPhysicalAddress()函数在开始之前，首先测试线性地址是否位于 0x80000000-----0x9FFFFFFF。就像前面提到的，这里存放着 hal.dll 和 ntoskrnl.exe，而且这也是 Windows 2000 使用 4MB 页的地址块。这个有趣的特性是，如果给定的线性地址位于这一范围，MmGetPhysicalAddress()将不会关心所有的 PDE 或 PTE。替代的是，该函数简单的将线性地址的高 3 位设为零，然后加上字节偏移量，最后将得到地址作为物理地址返回。这意味着，物理地址范围：0x00000000----0x1FFFFFFF 将按照 1：1 的比例映射到线性地址 0x80000000----0x9FFFFFFF！要知道 ntoskrnl.exe 总是被加载到线性地址 0x80400000，这意味着 Windows 2000 的内核总位于物理地址 0x00400000，这种情况发生在第二个 4MB 页的

基地址位于物理内存中。事实上，通过检查这些内存区域可以证明上面的假定是正确的。本章提供的 Memory SPY 将使你有机会看到这一点。

补充：

这部分内容选择自《Windows 环境下 32 位汇编语言程序设计》

x86 的内存分页机制

当 x86 CPU 工作在保护模式和虚拟 8086 模式时，可以使用全部 32 根地址线访问 4GB 的内存。因为 80386 的所有通用寄存器都是 32 位的，所以用任何一个通用寄存器来间接寻址，不必分段就可以访问到 4GB 的内存地址。

但这并不意味着，此时段寄存器就不再有用了。实际上，段寄存器更加有用了，虽然在寻址上没有分段的限制了，但在保护模式下，一个地址空间是否可以被写入，可以被多少优先级的代码写入，是不是允许执行等等涉及保护的问题就出来了。要解决这些问题，必须对一个地址空间定义一些安全上的属性。段寄存器这时就派上了用场。但是设计属性和保护模式下段的其他参数，要表示的信息太多了，要用 64 位长的数据才能表示。我们把这 64 位的属性数据叫做段描述符（Segment Descriptor）。

80386 的段寄存器是 16 位的，无法放下保护模式下 64 位的段描述符。如何解决这个问题呢？方法是把所有段的段描述符顺序存放在内存中的指定位置，组成一个段描述符表（Descriptor Table）；而段寄存器中的 16 位用来做索引信息，指定这个段的属性用段描述符表中的第几个描述符来表示。这时，段寄存器中的信息不再是段地址了，而是段选择器（Segment Selector）。可以通过它在段描述符表中“选择”一个项目已得到段的全部信息。

那么段描述符表存放在哪里呢？80386 引入了两个新的寄存器来管理段描述符表。一个是 48 位的全局描述符表寄存器 GDTR，一个是 16 位的局部描述符表寄存器 LDTR。那么，为什么有两个描述符表寄存器呢？

GDTR 指向的描述符表为全局描述符表 GDT（Global Descriptor Table）。它包含系统中所有任务都可用的段描述符，通常包含描述操作系统所使用的代码段、数据段和堆栈段的描述符及各任务的 LDT 段等。全局描述符表只有一个。

LDTR 指向局部描述符表 LDT（Local Descriptor Table）。80386 处理器设计成每个任务都有一个独立的 LDT。它包含每个任务私有的代码段、数据段和堆栈段的描述符，也包含该任务所使用的一些门描述符，如任务门和调用门描述符等。

不同任务的局部描述符分别组成不同的内存段，描述这些内存段的描述符当作系统描述符放在全局描述符表中。和 GDTR 直接指向内存地址不同，LDTR 和 CS、DS 等段选择器一样只存放索引值，指向局部描述符内存段对应的描述符在全局描述符表中的位置。随着任务的切换，只要改变 LDTR 的值，系统当前的局部描述符表 LDT 也随之切换，这样便于个任务之间数据的隔离。但 GDT 并不随着任务的切换而切换。

16 位的段选择器如何使用全局描述符表和局部描述符表这两个表呢？实际上，段选择器中只有高 13 位表示索引值。剩下的 3 个数据位中，第 0，1 位表示程序的当前优先级 RPL；第 2 位 TI 位用来表示在段描述符的位置；TI=0 表示在 GDT 中，TI=1 表示在 LDT 中。

80386 处理器把 4KB 大小的一块内存当作一“页”内存，每页物理内存可以根据“页目录”和“页表”，随意映射到不同的线性地址上。这样，就可以将物理地址不连续的内存的映射连到一起，在线性地址上视为连续。在 80386 处理器中，除了与 CR3（保存当前页目录的地址）相关的指令使用的是物理地址外，其他所有指令都是使用线性地址寻址的。

是否启用内存分页机制是由 80386 处理器新增的 CR0 寄存器中的位 31（PG 位）决定的。如果 PG=0，则分页机制不启用，这时所有指令寻址的地址（线性地址）就是系统中实际的物理地址；当 PG=1 的时候，80386 处理器进入内存分页管理模式，所有的线性地址要经过页表的映射才得到最后的物理地址。

3.1.3、数据结构

本章随后的示例代码的某些部分将涉及底层的内存管理机制，在前面我们已快速浏览了该机制内部的大致轮廓。为了方便，我用 C 语言定义了几个数据结构。这是因为 i386 CPU 内部的很多数据项需要使用一个二进制位或一组二进制位，而 C 的位域（bit-fields）唾手可得。位域可以很有效的访问一个大的数据中的一个位或从中提取一组连续的位。微软的 Visual C/C++可以产生非常棒的代码来完成位域的操作。**列表 4-2** 是一系列 CPU 数据类型定义的一部分，该列表包含如下的内容：

- **X86_REGISTER** 这是一个基本的无符号 32 位整数类型，该类型可描述多个 CPU 寄存器。这包括：通用的、索引、指针、控制、调试和测试寄存器。
- **X86_SELECTOR** 代表一个 16 位的段选择器，如 CS、DS、ES、FS、GS 和 SS。在**图 4-1** 和**图 4-2** 中，选择器可描述 48 位逻辑地址的高 8 位，或作为描述符表的索引。为了计算的方便，16 位选择器的值被扩展到 32 位，不过高 16 位被标识为“保留”。注意，X86_SELECTOR 结构实际是两个结构的联合（union）。第一个指定了选择器的值，该值占用一个 16 位的 WORD，其名字为 wValue，第二个采用了位域。RPL 域指定了请求的特权级，在 Windows 2000 上其值或者为 0（内核模式）或者为 3（用户模式）。TI 位用来选择 GDT 或 LDT。
- **X86_DESCRIPTOR** 定义了由选择器指向的页表项的格式。这是一个 64 位的数值，由于历史演化，该结构比较让人费解。线性基地址定义了与其相关的段的起始位置，它们分散在三个位域中：Base1、Base2 和 Base3，Base1 是作用最小的部分。段的界限指定了段的大小，The segment limit specifying the segment size minus one is divided into the pair Limitl and Limit2, with the former representing the least significant half.剩余的位域存放不同的段属性（cf. Intel 1999c, pp.3-11）。例如，G 位域定义了段的粒度。如果为零，段的限制按字节指定；否则，限制值为 4KB 的倍数。像 X86_SELECTOR 一样，X86_DESCRIPTOR 结构由一个 union 组成，以允许按不同的方式解释它的值。如果你必须复制描述符（在忽略其内部情况下）那么 dValueLow 和 dValueHigh 成员将会很有帮助。
- **X86_GATE** 该结构看起来有些像 X86_DESCRIPTOR。事实上，这两个结构是相关的：X86_DESCRIPTRO 是一个 GDT 项，并描述了一个段的内存属性，X86_GATE 代表中断描述符表（IDT）中的一项，并描述了中断例程的内存属性。IDT 可以包含任务、

中断和陷阱门（不！ Bill Gates 并没有存储在 IDT 中！ 哈哈）。X86_GATE 结构可匹配上述三种类型，并通过 Type 位域来进行区分。Type 5 表示这是一个任务门；Type 6 和 14 为中断门；Type 7 和 15 为陷阱门。Type 中最重要的位是用来描述门的位数的位：该位若为 0 则表示是 16 位门；其余情况表示 32 位门。

- **X86_TABLE** 是一个巧妙的结构，该结构用来读取 GDTR 或 IDTR 的当前值，分别通过汇编指令 SGDT（存储 GDT 寄存器）和 SIDT（存储 IDT 寄存器）来实现（cf. Intel 1999b, pp.3-636）。这两个指令需要一个 48 位的内存操作数，在该操作数中存放限制值和基地址值。通过在结构体中增加一个 DWORD 来对齐 32 位的基地址，X86_TABLE 以一个 16 位的哑元成员 wReserved 开始。根据是否使用了 SGDT 或 SIDT 指令，其基地址将被解释为一个描述符指针或一个门指针，就像 PX86_DESCRIPTOR 和 PX86_GATE 中的 union 所暗示的那样。最后的 wLimit 成员在这两种类型的表中的意义均相同。

译注：

列表 4-2 中的这些结构定义可以在随书光盘的\src\common\include\w2k_spy.h 中找到。

```
typedef DWORD X86_REGISTER, *PX86_REGISTER, **PPX86_REGISTER;

// -------------------------------------------------------------

typedef struct _X86_SELECTOR
    {
    union
        {
        struct
            {
            WORD wValue;            // packed value
            WORD wReserved;
```

```
            };
        struct
            {
            unsigned RPL      :  2; // requested privilege level
            unsigned TI       :  1; // table indicator: 0=gdt, 1=ldt
            unsigned Index    : 13; // index into descriptor table
            unsigned Reserved : 16;
            };
        };
    }
    X86_SELECTOR, *PX86_SELECTOR, **PPX86_SELECTOR;

#define X86_SELECTOR_ sizeof (X86_SELECTOR)

// -------------------------------------------------------------

typedef struct _X86_DESCRIPTOR
    {
    union
        {
        struct
            {
            DWORD dValueLow;        // packed value
            DWORD dValueHigh;
            };
        struct
            {
            unsigned Limit1   : 16; // bits 15..00
            unsigned Base1    : 16; // bits 15..00
```

```
            unsigned Base2    :  8; // bits 23..16
            unsigned Type     :  4; // segment type
            unsigned S        :  1; // type (0=system, 1=code/data)
            unsigned DPL      :  2; // descriptor privilege level
            unsigned P        :  1; // segment present
            unsigned Limit2   :  4; // bits 19..16
            unsigned AVL      :  1; // available to programmer
            unsigned Reserved :  1;
            unsigned DB       :  1; // 0=16-bit, 1=32-bit
            unsigned G        :  1; // granularity (1=4KB)
            unsigned Base3    :  8; // bits 31..24
            };
        };
    }
    X86_DESCRIPTOR, *PX86_DESCRIPTOR, **PPX86_DESCRIPTOR;

#define X86_DESCRIPTOR_ sizeof (X86_DESCRIPTOR)

// ------------------------------------------------------------

typedef struct _X86_GATE
    {
    union
        {
        struct
            {
            DWORD dValueLow;          // packed value
            DWORD dValueHigh;
            };
```

```
        struct
            {
            unsigned Offset1    : 16; // bits 15..00
            unsigned Selector   : 16; // segment selector
            unsigned Parameters :  5; // parameters
            unsigned Reserved   :  3;
            unsigned Type       :  4; // gate type and size
            unsigned S          :  1; // always 0
            unsigned DPL        :  2; // descriptor privilege level
            unsigned P          :  1; // segment present
            unsigned Offset2    : 16; // bits 31..16
            };
        };
    }
    X86_GATE, *PX86_GATE, **PPX86_GATE;

#define X86_GATE_ sizeof (X86_GATE)

// ------------------------------------------------------------

typedef struct _X86_TABLE
    {
    WORD wReserved;                   // force 32-bit alignment
    WORD wLimit;                      // table limit
    union
        {
        PX86_DESCRIPTOR pDescriptors; // used by sgdt instruction
        PX86_GATE       pGates;       // used by sidt instruction
        };
```

```
    }
    X86_TABLE, *PX86_TABLE, **PPX86_TABLE;

#define X86_TABLE_ sizeof (X86_TABLE)
```

列表 4-2. i386 的寄存器、选择器、描述符、门和表

接下来的一组与 i386 内存管理相关的结构，它们收录在**列表 4-3** 中，这些结构包括：与请求式分页相关的结构和**图 4-3** 和**图 4-4** 给出的几个成员。

- **X86_PDBR** 该结构对应 CPU 的 CR3 寄存器，即众所周知的页目录基地址寄存器（PDBR）。其高 20 位为 PFN，即 4KB 物理页数组的索引。PFN=0 对应物理地址 0x00000000，PFN=1 为 0x00001000，依此类推。20 个位足够转换整个 4GB 地址空间。PDBR 中的 PFN 是物理页的索引，用来控制整个页目录。PFN 中剩余的位大多数都被保留，但 3 号位例外，它用来控制页一级的 write-through（page-level write-through, PWT），4 号位如果为 1，则禁止页一级的高速缓冲。
- **X86_PDE_4M** 和 **X86_PDE_4K** 是页目录项（PDE）的两个可选方案，用来选择 4MB 页或者 4KB 的页。一个页目录中最多包含 1024 个 PDE。PFN 是页帧号，它指向下一级的页。对于一个 4MB 的 PDE，其 PFN 位域仅有 10 个位的宽度，可寻址一个 4MB 的数据页。4KB 的 PDE 拥有 20 位的 PFN，可指向一个页表，由页表最终选择一个数据页。剩余的位用来定义多种属性。这些属性中最有趣的是“页大小”位 PS，用于控制页的大小（0=4KB，1=4MB）和“存在”位 P，标识下属的数据页（4MB 模式）或页表（4KB 模式）是否存在于物理内存中。
- **X86_PTE_4K** 定义了页表项（属于一个页表）的内部结构。和页目录类似，一个页表可拥有 1024 个项。X86_PTE_4K 和 X86_PDE_4K 的不同之处为：前者没有 PS 位，这根本不需要，因为页的大小肯定是 4KB。需要注意的是，没有所谓的 4MB 的 PTE，因为采用 4MB 页的内存模式不需要页表这一中间层。
- **X86_PNPE** 代表一个“不存在的页”项（page-not-present entry, PNPE），也就是说，一个 PDE 或 PTE 中的 P 位为 0。如 Intel 的手册所说的，保留的第 31 位是“对操作系统或执行体（executive）均可用”（Intel 1999c,pp. 3-28）。如果一个线性地址映射到了一个 PNPE，这意味着这个地址或者还未使用或者它所指向的页已经被置换到了页面文件中。Windows 2000 使用 PNPE 保留的第 31 位来存储页的信息。有

关页信息的结构没有文档记载，不过它类似于名为 PageFile 的第 10 位，如列表 4-3 所示，如果设置了该位，则表示页已被置换出物理内存。在这种情况下，Reserved1 和 Reserved2 位域将包含系统在页面文件中定位该页的信息，因此，当需要访问该页时，可很快的将其换回物理内存。

- X86_PE　该结构是为了方便使用而加入的。它仅包含一个 union，该 union 包括页项所有可能的状态，此处的页项是指：PDBR 的内容、所有 4MB 和 4KB 的 PDE、PTE，以及所有的 PNPE。

```
typedef struct _X86_PDBR // page-directory base register (cr3)
    {
    union
        {
        struct
            {
            DWORD dValue;            // packed value
            };
        struct
            {
            unsigned Reserved1 :  3;
            unsigned PWT       :  1; // page-level write-through
            unsigned PCD       :  1; // page-level cache disabled
            unsigned Reserved2 :  7;
            unsigned PFN       : 20; // page-frame number
            };
        };
    }
    X86_PDBR, *PX86_PDBR, **PPX86_PDBR;

#define X86_PDBR_ sizeof (X86_PDBR)
```

```
// ------------------------------------------------------------


typedef struct _X86_PDE_4M // page-directory entry (4-MB page)
    {
    union
        {
        struct
            {
            DWORD dValue;            // packed value
            };
        struct
            {
            unsigned P         :  1; // present (1 = present)
            unsigned RW        :  1; // read/write
            unsigned US        :  1; // user/supervisor
            unsigned PWT       :  1; // page-level write-through
            unsigned PCD       :  1; // page-level cache disabled
            unsigned A         :  1; // accessed
            unsigned D         :  1; // dirty
            unsigned PS        :  1; // page size (1 = 4-MB page)
            unsigned G         :  1; // global page
            unsigned Available :  3; // available to programmer
            unsigned Reserved  : 10;
            unsigned PFN       : 10; // page-frame number
            };
        };
    }
    X86_PDE_4M, *PX86_PDE_4M, **PPX86_PDE_4M;
```

```
#define X86_PDE_4M_ sizeof (X86_PDE_4M)


// ------------------------------------------------------------


typedef struct _X86_PDE_4K // page-directory entry (4-KB page)
    {
    union
        {
        struct
            {
            DWORD dValue;            // packed value
            };
        struct
            {
            unsigned P         :  1; // present (1 = present)
            unsigned RW        :  1; // read/write
            unsigned US        :  1; // user/supervisor
            unsigned PWT       :  1; // page-level write-through
            unsigned PCD       :  1; // page-level cache disabled
            unsigned A         :  1; // accessed
            unsigned Reserved  :  1; // dirty
            unsigned PS        :  1; // page size (0 = 4-KB page)
            unsigned G         :  1; // global page
            unsigned Available :  3; // available to programmer
            unsigned PFN       : 20; // page-frame number
            };
        };
    }
    X86_PDE_4K, *PX86_PDE_4K, **PPX86_PDE_4K;
```

```
#define X86_PDE_4K_ sizeof (X86_PDE_4K)

// ------------------------------------------------------------------

typedef struct _X86_PTE_4K // page-table entry (4-KB page)
    {
    union
        {
        struct
            {
            DWORD dValue;            // packed value
            };
        struct
            {
            unsigned P         :  1; // present (1 = present)
            unsigned RW        :  1; // read/write
            unsigned US        :  1; // user/supervisor
            unsigned PWT       :  1; // page-level write-through
            unsigned PCD       :  1; // page-level cache disabled
            unsigned A         :  1; // accessed
            unsigned D         :  1; // dirty
            unsigned Reserved  :  1;
            unsigned G         :  1; // global page
            unsigned Available :  3; // available to programmer
            unsigned PFN       : 20; // page-frame number
            };
        };
    }
```

```
    X86_PTE_4K, *PX86_PTE_4K, **PPX86_PTE_4K;

#define X86_PTE_4K_ sizeof (X86_PTE_4K)

// ------------------------------------------------------------

typedef struct _X86_PNPE // page not present entry
    {
    union
        {
        struct
            {
            DWORD dValue;            // packed value
            };
        struct
            {
            unsigned P         :  1; // present (0 = not present)
            unsigned Reserved1 :  9;
            unsigned PageFile  :  1; // page swapped to pagefile
            unsigned Reserved2 : 21;
            };
        };
    }
    X86_PNPE, *PX86_PNPE, **PPX86_PNPE;

#define X86_PNPE_ sizeof (X86_PNPE)

// ------------------------------------------------------------
```

```
typedef struct _X86_PE // general page entry
    {
    union
        {
        DWORD      dValue; // packed value
        X86_PDBR   pdbr;   // page-directory Base Register
        X86_PDE_4M pde4M;  // page-directory entry (4-MB page)
        X86_PDE_4K pde4K;  // page-directory entry (4-KB page)
        X86_PTE_4K pte4K;  // page-table entry (4-KB page)
        X86_PNPE   pnpe;   // page not present entry
        };
    }
    X86_PE, *PX86_PE, **PPX86_PE;


#define X86_PE_ sizeof (X86_PE)
```

列表 4-3.　i386 的 PDBR、PDE、PTE 和 PNPE

在**列表 4-4** 中，我增加了线性地址的结构化表示。这些结构是**图 4-3** 和 **4-4** 中的“线性地址”的正式形式。

- **X86_LINEAR_4M**　该结构是指向 4MB 数据页的线性地址的正式形式，如**图 4-4** 所示。页目录索引（PDI）是一个页目录的索引，页目录地址由 PDBR 给出，使用 PDI 可选择页目录中的一个 PDE。22 位的 Offset 成员指向一个目标地址，此目标地址对应 4MB 的物理页。
- **X86_LINEAR_4K**　是一个 4KB 线性地址类型的变量，如**图 4-3** 所示。该结构由三个位域组成：和 4MB 地址类似，高 10 位为 PDI，用来选择一个 PDE；页表索引 PTI 的任务与 PDI 相似，指向由 PDE（该 PDE 由前面的 PDI 指定）确定的页表中的一个 PTE；剩余的 12 个位是在 4KB 物理页中的偏移量。
- **X86_LINEAR**　是另一个为使用方便而加入的结构。该结构只是简单的将 X86_LINEAR_4K 和 X86_LINEAR_4M 联合为一个数据类型。详见**列表 4-4**。

```
typedef struct _X86_LINEAR_4M // linear address (4-MB page)
```

```
    {
    union
        {
        struct
            {
            PVOID pAddress;       // packed address
            };
        struct
            {
            unsigned Offset : 22; // offset into page
            unsigned PDI    : 10; // page-directory index
            };
        };
    }
    X86_LINEAR_4M, *PX86_LINEAR_4M, **PPX86_LINEAR_4M;


#define X86_LINEAR_4M_ sizeof (X86_LINEAR_4M)


// ------------------------------------------------------------


typedef struct _X86_LINEAR_4K // linear address (4-KB page)
    {
    union
        {
        struct
            {
            PVOID pAddress;       // packed address
            };
        struct
```

```
            {
            unsigned Offset : 12; // offset into page
            unsigned PTI    : 10; // page-table index
            unsigned PDI    : 10; // page-directory index
            };
        };
    }
    X86_LINEAR_4K, *PX86_LINEAR_4K, **PPX86_LINEAR_4K;

#define X86_LINEAR_4K_ sizeof (X86_LINEAR_4K)

// -------------------------------------------------------------

typedef struct _X86_LINEAR // general linear address
    {
    union
        {
        PVOID         pAddress; // packed address
        X86_LINEAR_4M linear4M; // linear address (4-MB page)
        X86_LINEAR_4K linear4K; // linear address (4-KB page)
        };
    }
    X86_LINEAR, *PX86_LINEAR, **PPX86_LINEAR;

#define X86_LINEAR_ sizeof (X86_LINEAR)
```

列表 4-4. i386 的线性地址

4.1.4、宏和常量

列表 4-5 给出的定义是对**列表 4-2** 到**列表 4-4** 所示结构的补充，让我们可以更容易的和 i386 内存管理一起工作。**列表 4-5** 的定义可以分为三大组。第一组用于控制线性地址：

1. **X86_PAGE_MASK**、**X86_PDI_MASK** 和 **X86_PTI_MASK** 都是位掩码（bit mask），用来选择线性地址中的某一部分。它们都基于常量：PAGE_SHIFT (12)、PDI-SHIFT (22)和 PTI-SHIFT (12)，这些常量定义于 Windows 2000 DDK 的头文件 ntddk.h 中。X86_PAGE_MASK 等价于 0xFFFFF000，可有效的屏蔽 4KB 线性地址（X86_LINEAR_4K）中的偏移量部分。X86_PDI_MASK 等价于 0xFFC00000，显然这可从线性地址中提取高 10 位的 PDI。X86_PTI_MASK 等价于 0x003FF0000，用于屏蔽线性地址中除 PTI 外的所有位。
2. **X86_PAGE()**、**X86_PDI()**和 **X86_PTI()** 使用上面的常量来计算给定线性地址的页索引、PDI 和 PTI。X86_PAGE()一般用来从 Windows 2000 的 PTE 数组（该数组首地址为：0xC0000000）中读取一个 PTE。X86_PDI()和 X86_PTI()只是针对给定的指针，简单的使用 X86_PDI_MASK 或 X86_PTI_MASK，并将得到的索引移动到最右边。
3. **X86_OFFSET_4M()**和 **X86_OFFSET_4K()** 分别从 4MB 或 4KB 线性地址中提取偏移量部分。
4. **X86_PAGE_4M** 和 **X86_PAGE_4K** 根据 DDK 中的常量 PDI_SHIFT 和 PTI_SHIFT 来计算 4MB 和 4KB 页的大小。X86_PAGE_4M=4,194,304，X86_PAGE_4K=4,096。注意，X86_PAGE_4K 等价于 DDK 常量 PAGE_SIZE，该常量也定义于 ntddk.h 中。
5. **X86_PAGES_4M** 和 **X86_PAGES_4K** 分别表示 4GB 地址空间中可容纳的 4MB 或 4KB 页的总数。X86_PAGES_4M 等价于 1,024，X86_PAGES_4K 等价于 1,048,576。

```
#define X86_PAGE_MASK (0 - (1 << PAGE_SHIFT))
#define X86_PAGE(_p)  (((DWORD) (_p) & X86_PAGE_MASK) >> PAGE_SHIFT)

#define X86_PDI_MASK  (0 - (1 << PDI_SHIFT))
#define X86_PDI(_p)   (((DWORD) (_p) & X86_PDI_MASK) >> PDI_SHIFT)

#define X86_PTI_MASK  ((0 - (1 << PTI_SHIFT)) & ~X86_PDI_MASK)
#define X86_PTI(_p)   (((DWORD) (_p) & X86_PTI_MASK) >> PTI_SHIFT)
```

```
#define X86_OFFSET(_p,_m) ((DWORD_PTR) (_p) & ~(_m))
#define X86_OFFSET_4M(_p) X86_OFFSET (_p, X86_PDI_MASK)
#define X86_OFFSET_4K(_p) X86_OFFSET (_p, X86_PDI_MASK|X86_PTI_MASK)

#define X86_PAGE_4M   (1 << PDI_SHIFT)
#define X86_PAGE_4K   (1 << PTI_SHIFT)

#define X86_PAGES_4M  (1 << (32 - PDI_SHIFT))
#define X86_PAGES_4K  (1 << (32 - PTI_SHIFT))

// ------------------------------------------------------------------

#define X86_PAGES         0xC0000000
#define X86_PTE_ARRAY     ((PX86_PE) X86_PAGES)
#define X86_PDE_ARRAY     (X86_PTE_ARRAY + (X86_PAGES >> PTI_SHIFT))

// ------------------------------------------------------------------

#define X86_SEGMENT_OTHER           0
#define X86_SEGMENT_CS              1
#define X86_SEGMENT_DS              2
#define X86_SEGMENT_ES              3
#define X86_SEGMENT_FS              4
#define X86_SEGMENT_GS              5
#define X86_SEGMENT_SS              6
#define X86_SEGMENT_TSS             7

// ------------------------------------------------------------------
```

```
#define X86_SELECTOR_RPL            0x0003
#define X86_SELECTOR_TI             0x0004
#define X86_SELECTOR_INDEX          0xFFF8
#define X86_SELECTOR_SHIFT          3

#define X86_SELECTOR_LIMIT          (X86_SELECTOR_INDEX >> \
                                     X86_SELECTOR_SHIFT)

// ------------------------------------------------------------

#define X86_DESCRIPTOR_SYS_TSS16A       0x1
#define X86_DESCRIPTOR_SYS_LDT          0x2
#define X86_DESCRIPTOR_SYS_TSS16B       0x3
#define X86_DESCRIPTOR_SYS_CALL16       0x4
#define X86_DESCRIPTOR_SYS_TASK         0x5
#define X86_DESCRIPTOR_SYS_INT16        0x6
#define X86_DESCRIPTOR_SYS_TRAP16       0x7
#define X86_DESCRIPTOR_SYS_TSS32A       0x9
#define X86_DESCRIPTOR_SYS_TSS32B       0xB
#define X86_DESCRIPTOR_SYS_CALL32       0xC
#define X86_DESCRIPTOR_SYS_INT32        0xE
#define X86_DESCRIPTOR_SYS_TRAP32       0xF

// ------------------------------------------------------------

#define X86_DESCRIPTOR_APP_ACCESSED     0x1
#define X86_DESCRIPTOR_APP_READ_WRITE   0x2
#define X86_DESCRIPTOR_APP_EXECUTE_READ 0x2
#define X86_DESCRIPTOR_APP_EXPAND_DOWN  0x4
```

```
#define X86_DESCRIPTOR_APP_CONFORMING   0x4
#define X86_DESCRIPTOR_APP_CODE         0x8
```

列表 4-5.　附加的 i386 内存管理相关定义

第二组宏和常量与 Windows 2000 的 PDE、PTE 数组有关。和其他几个系统地址不同，这些数组的基地址并没有在系统启动时作为一个全局变量出现，而是被定义成了一个常量。可以通过反编译内存管理 API 函数：MmGetPhysicalAddress()和 MmIsAddressValid()来证明，在这些函数里，这些地址都以“魔术数字”的形式出现。这些常量并没有包括在 DDK 头文件中，不过**列表 4-5** 展示了如何定义它们。

- **X86_PAGES**　是一个硬编码的地址和指针（指向 0xC0000000），0xC0000000 是 Windows 2000 的 PTE 数组开始的地方。
- **X86_PTE_ARRAY**　等价于 X86_PAGES，但是被转型为 PX86_PE，也就是说，指向一个 X86_PE 类型的数组，X86_PE 定义于列表 4-2。
- **X86_PDE_ARRAY**　是一个巧妙的定义，它通过 PTE 数组的位置来计算 PDE 数组的基地址，这需要用到 PTI_SHIFT 常量。将线性地址映射为 PTE 地址的通用格式为：（（LinearAdress >> 12）*4）+0xC0000000，线性地址 0xC0000000 转换后的地址为页目录的基地址。

列表 4-5 的最后两部分包括选择器和特殊类型的描述符，以及对**列表 4-2** 的补充。

- X86_SELECTOR_RPL、X86_SELECTOR_TI 和 X86_SELECTOR_INDEX 都是位掩码，分别对应 X86_SELECTOR 结构中的 RPL、TI 和 Index 成员。
- X86_SELECTOR_SHIFT　是一个右移因子，用来使选择器的 Index 的数值向右对齐。
- X86_SELECTOR_LIMIT　定义了选择器可使用的最大索引值，该限制为 8,191。这个值确定了描述符表的最大尺寸。每个选择器索引均指向一个描述符，每个描述符包含 64 个位（即 8 个字节）。所以，描述符表的最大尺寸为：8,192*8=64KB。
- X86_DESCRIPTOR_SYS_*　是一组常量，用于定义系统描述符类型。如果描述符的 S 位被设为 0，那么描述的 Type 成员将采用这一组类型中的某一个。请参考**列表 4-2** 中的 X86_DESCRIPTOR 的定义。系统描述符类型在 Intel 手册中有详细介绍（Intel 1999c, pp. 3-15f），**表 4-1** 给出了所有可用的系统描述符类型。

列表 4-5 中的 X86_DESCRIPTOR_APP_*常量也可用于定义描述符的 Type 成员，前提是描述符的 S 位不为 0。此时，该应用程序描述符可能需要引用一个代码或数据段。因为应用程序描述符类型的属性受 Type 域的第四个位影响，所以 X86_DESCRIPTOR_APP_*常量被定义为单位掩码（single-bit mask），这样一些位就可针对数据和代码段有不同的解释。

- X86_DESCRIPTOR_APP_ACCESSED　如果一个段可以被访问，则采用
- X86_DESCRIPTOR_APP_READ_WRITE　决定一个数据段是否允许只读或读/写访问。
- X86_DESCRIPTOR_APP_CONFORMATING　说明一个代码段是否相匹配。也就是说，它是否可以被以被弱特权代码（less privileged code）调用（参考 Intel 1999c, pp. 4-13ff）。
- X86_DESCRIPTOR_APP_CODE　用来区别代码段和数据段。注意，堆栈属于数据段的范畴，而且必须总是可写的。

稍后，当下一章中的 Memory Spy 程序开始运行时，我们将重温系统描述符。**表 4-1** 算是 i386 内存管理的一个简短总结。有关本话题的更多内容，请参考 Intel Pentium 手册（Intel 1999a，1999b，1999c）。

表 4-1.　　系统描述符类型

名　　称	值	描　　述
X86_DESCRIPTOR_SYS_TSS16A	0x1	16 位任务状态段（可用）
X86_DESCRIPTOR_SYS_LDT	0x2	本地描述符表（LDT）
X86_DESCRIPTOR_SYS_TSS16B	0x3	16 位任务状态段（繁忙）
X86_DESCRIPTOR_SYS_CALL16	0x4	16 位调用门
X86_DESCRIPTOR_SYS_TASK	0x5	任务门
X86_DESCRIPTOR_SYS_INT16	0x6	16 位中断门
X86_DESCRIPTOR_SYS_TRAP16	0x7	16 位陷阱门
X86_DESCRIPTOR_SYS_TSS32A	0x9	32 位任务状态段（可用）
X86_DESCRIPTOR_SYS_TSS32B	0xB	32 位任务状态段（繁忙）
X86_DESCRIPTOR_SYS_CALL32	0xC	32 位调用门
X86_DESCRIPTOR_SYS_INT32	0xE	32 位中断门
X86_DESCRIPTOR_SYS_TRAP32	0XF	32 位陷阱门

4.2、Memory Spy Device 示例

微软对 Windows NT 和 2000 说的最多的就是它们是*安全的操作系统*。它们不但在网络环境中加入了用户验证系统，同时还加强了系统的稳健性（robustness），以进一步降低错误应用程序危及系统完整性的概率，这些错误的程序可能使用了非法的指针或者在其内存数据结构以外的地方进行了写入操作。这些在 Windows 3.x 上都是十分让人头疼得问题，因为 Windows 3.x 系统和所有的应用程序共享单一的内存空间。Windows NT 为系统和应用程序内存以及并发的进程提供了完全独立的内存空间。每个进程都有其独立的 4GB 地址空间，如**图 4-2** 所示。无论何时发生任务切换，当前的地址空间都会被换出（switch out），同时另一个被映射进来，它们各自使用不同的段寄存器、页表和其他内存管理结构。这种设计避免了应用程序无意中修改另一个程序所使用的内存。由于每个进程必然会要求访问系统资源，所以在 4GB 空间中总是包含一些系统数据和代码，并采用了一个不同的技巧来保护这些内存区域不被恶意程序代码所覆写（overwritten）。

4.2.1、Windows 2000 的内存分段

Windows 2000 继承了 Windows NT 4.0 的基本内存分段模型，默认情况下，该模型将 4GB 地址空间划分为相等的两块。低一半的地址范围是：0x00000000 ---- 0x7FFFFFFF，其中包含运行于用户模式（用 Intel 的术语来说是，是特权级 3 或 Ring 3）的应用程序的数据和代码。高一半的地址范围是：0x80000000 --- 0xFFFFFFFF，默认全部保留给系统使用，位于这一范围的代码运行于内核模式（即特权级为 0 或 Ring 0）。特权级决定了代码可以执行什么操作以及可以访问那一个块内存。这意味着对于低特权级的代码来说，会被禁止执行某些 CPU 指令或访问某些内存区域。例如，如果一个用户模式下的程序触及了任何 0x80000000（即 4GB 地址空间中的高一半）以上的地址，系统会抛出一个异常并同时终止该程序的运行，不会给其任何机会。

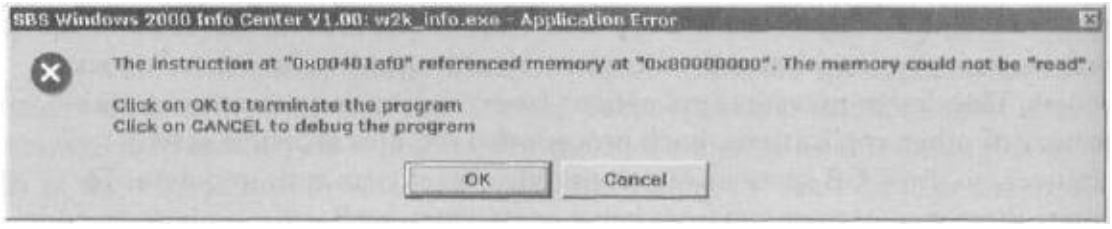


图 4-5.　用户模式下不能访问 0x80000000 以上的地址

图 4-5 展示了程序试图读取 0x80000000 地址时的情况。这种严格的访问限制对于系统的完整性来说是好事，但对于调试工具就不是什么好消息了，因为调试工具需要访问所有可用内存。幸运的是，存在着一个简单的方法：采用内核驱动程序，和系统本身类似，它也运行于高特权级（即 Ring 3），因此它们可以执行所有的 CPU 指令，可访问所有的内存区域。这其中的诀窍就是将一个 Spy 驱动程序注入系统，用它来访问需要的内存，并将读到的内容发送到它的搭档程序，该搭档程序会在用户模式下等待。当然，内核驱动程序不能读取虚拟内存地址，而且得不到分页机制的支持。因此，这样的驱动程序必须在访问一个地址之前小心的检查它，以避免出现蓝屏死机（Blue Screen Of Death，BSOD）。相对于应用程序引发的异常（仅会终止出现问题的程序），驱动程序引发的异常会停止整个系统，并强迫进行重启。

4.2.2、设备 I/O 控制 Dispatcher（Device I/O Control Dispatcher）

本书光盘上有一个通用Spy Device的源代码，该Spy Device作为内核驱动程序实现。可以在\src\w2k_spy目录下找到它的源代码。这个设备基于第三章的驱动向导所产生的驱动程序骨架。其用户模式下的接口为w2k_spy.sys，w2k_spy.sys采用Win32 的设备I/O控制（IOCTL），在第三章中曾简要的谈过IOCTL。Spy Device定义了一个名为\Device\w2k_spy的设备和一个符号链接\DosDevices\w2k_spy，定义符号链接是为了能在用户模式下访问该设备。非常可笑的是符号链接的名字空间居然是\DosDevice，而在这儿，我们使用的可不是一个DOS设备驱动。这就像历史上有名的root，原本是叫做石头的☺。安装好符号链接后，驱动程序就可以被用户模式下的任何模块打开了，方法是：使用Win32 API函数CreateFile()，路径为\\.\w2k_spy。字符串\\.\是通用转义符，表示本地设备。例如，\\.\C:指向本地硬盘上的C：分区。从SDK的文档中可了解CreateFile()的更多细节。

该驱动程序的头文件有一部分已经由列表 4-2 到列表 4-5 给出。这个文件有些像 DLL 的头文件：它包含在编译过程中，模块所需的定义，而且还为客户端程序提供了足够的接口信息。DLL 和驱动程序以及客户端程序都包含相同的头文件，但每个模块会取出各自所需的定义以完成正确的操作。不过，头文件的这种两面性给内核驱动程序带来的麻烦要远多于给 DLL 带来的，这都是因为微软给驱动程序提供的特殊开发环境所致。不幸的是，DDK 中的头文件并不能和 SDK 中的 Win32 文件兼容。至少在 C 工程，二者的头文件是不能混合使用的。

这样的结果就是陷入了僵局，此种情况下，内核驱动可以访问的常量、宏和数据类型对于客户端程序来说是却是无法使用的。因此，w2k_spy.c 定义了一个名为_W2K_SPY_SYS_的标志常量，w2k_spy.h 通过#ifdef…..#else…..#endif 来检查该常量是否出现，以决定需要补充哪些缺少的定义。这意味着，所有出现在#ifdef _W2K_SPY_SYS_ 之后的定义仅可被驱动代码看到，位于#else 之后的则专用于客户端程序。w2k_spy.h 中条件语句之外的所有部分被这两个模块同时使用。

在第三章中，在讨论我的驱动向导时，我给出了向导生成的驱动程序骨架，如**列表 3-3** 所示。由该驱动向导生成的新的驱动工程均开始于 DeviceDispatcher()函数。该函数接受一个设备上下文指针，以及一个指向 IRP（I/O 请求包）的指针，该 IRP 随后将会被分派。向导的样板代码已经处理了基本的 I/O 请求：IRP_MJ_CREATE、IRP_MJ_CLEANUP 和 IRP_MJ_CLSE,当客户要关闭一个设备时,会给该设备发送这些 I/O 请求。DeviceDispatcher()针对这些请求只是简单的返回 STATUS_SUCCESS，因此设备可以被正确的打开和关闭。对于某些设备，这种动作已经足够，但有些设备还需要初始化和清理代码，这些代码多少都有些复杂。对于其他的请求，第三章中的驱动程序骨架总是返回 STATUS_NOT_IMPLEMENTED。扩展该骨架代码的第一步是修改默认的动作，以便处理更多的 I/O 请求。就像 w2k_spy.sys 的主要任务之一：通过 IOCTL 调用将在用户模式下无法访问的数据发送给 Win32 应用程序，因此首先需要在 DeviceDispatcher()中添加处理 IRP_MJ_DEVICE_CONTROL 的函数。**列表 4-6** 给出了更新后的代码。

```
NTSTATUS DeviceDispatcher (PDEVICE_CONTEXT pDeviceContext,
                           PIRP            pIrp)
    {
    PIO_STACK_LOCATION pisl;
    DWORD              dInfo = 0;
    NTSTATUS           ns    = STATUS_NOT_IMPLEMENTED;

    pisl = IoGetCurrentIrpStackLocation (pIrp);

    switch (pisl->MajorFunction)
        {
```

```
        case IRP_MJ_CREATE:
        case IRP_MJ_CLEANUP:
        case IRP_MJ_CLOSE:
            {
            ns = STATUS_SUCCESS;
            break;
            }
        case IRP_MJ_DEVICE_CONTROL:
            {
            ns = SpyDispatcher (pDeviceContext,
                                pisl->Parameters.DeviceIoControl.IoControlCode,
                                pIrp->AssociatedIrp.SystemBuffer,
pisl->Parameters.DeviceIoControl.InputBufferLength,
                                pIrp->AssociatedIrp.SystemBuffer,
pisl->Parameters.DeviceIoControl.OutputBufferLength,
                                &dInfo);
            break;
            }
        }
    pIrp->IoStatus.Status      = ns;
    pIrp->IoStatus.Information = dInfo;

    IoCompleteRequest (pIrp, IO_NO_INCREMENT);
    return ns;
    }
```

列表 4-6. 为 Dispatcher 增加处理的 IRP_MJ_DEVICE_CONTROL 函数

列表 4-6 中的 IOCTL 处理代码非常简单，它仅调用了 SpyDispatcher()，并将一个扩展后的 IRP 结构和当前 I/O 堆栈位置作为参数传递给 SpyDispatcher()。SpyDispatcher()在**列表 4-7** 中给出，该函数需要如下的参数：

- pDeviceContext 一个驱动程序的设备上下文指针。驱动程序向导提供了的基本 Device_Context 结构，该结构中包含驱动程序和设备对象指针（参见**列表 3-4**）。不过，Spy 驱动程序在该结构中增加了一对私有成员。
- dCode 指定了 IOCTL 编码，以确定 Spy 设备需要执行的命令。一个 IOCTL 编码是一个 32 位整数，它包含 4 个位域，如**图 4-6** 所示。
- pInput 指向一个输入缓冲区，用于给 IOCTL 提供输入数据。
- dInput 输入缓冲区的大小。
- pOutput 指向用来接收 IOCTL 输出数据的输出缓冲区。
- dOutput 输出缓冲区的大小
- pdInfo 指向一个 DWORD 变量，该变量保存写入输出缓冲区中的字节数。

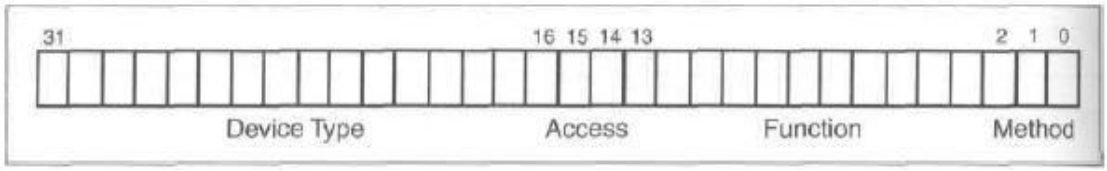


图 4-6. 设备 I/O 控制编码的结构

根据所用的 IOCTL 使用的传输模式，输入/输出缓冲区会以不同的方式从系统传递给驱动程序。Spy 设备使用已缓存的 I/O（buffered I/O），系统将输入数据复制到一个安全的缓冲区（此缓冲区由系统自动分配）中，在返回时，将指定数目的数据从同样的系统缓冲区中复制到调用者提供的输出缓冲区中。一定要牢记：在这种情况下，输入和输出缓冲区是重叠的，因此 IOCTL 的处理代码必须在向输出缓冲区中写入任何数据之前，保存所有它稍后可能需要使用的输入数据。系统 I/O 缓冲区的指针保存在 IRP 结构中的 SystemBuffer 成员中（参见 ntddk.h）。输入/输出缓冲区的大小保存在一个不同的地方，它们是 IRP 的参数成员 DeviceIoControl 的一部分，分别为 InputBufferLength 和 OutputBufferLength。DeviceIoControl 子结构还通过其 IoControlCode 成员提供了 IOCTL 编码。有关 Windows NT/2000 的 IOCTL 的传输模式的信息以及它们如何传入/传出数据，请参考我在 Windows Developer’s Journal(Schreiber 1997)发表的文章“A Spy Filter Driver for Windows NT”。

```
NTSTATUS SpyDispatcher (PDEVICE_CONTEXT pDeviceContext,
                        DWORD           dCode,
                        PVOID           pInput,
                        DWORD           dInput,
                        PVOID           pOutput,
                        DWORD           dOutput,
                        PDWORD          pdInfo)
    {
    SPY_MEMORY_BLOCK smb;
    SPY_PAGE_ENTRY   spe;
    SPY_CALL_INPUT   sci;
    PHYSICAL_ADDRESS pa;
    DWORD            dValue, dCount;
    BOOL             fReset, fPause, fFilter, fLine;
    PVOID            pAddress;
    PBYTE            pbName;
    HANDLE           hObject;
    NTSTATUS         ns = STATUS_INVALID_PARAMETER;

    MUTEX_WAIT (pDeviceContext->kmDispatch);

    *pdInfo = 0;

    switch (dCode)
        {
        case SPY_IO_VERSION_INFO:
            {
            ns = SpyOutputVersionInfo (pOutput, dOutput, pdInfo);
            break;
```

```
        }
    case SPY_IO_OS_INFO:
        {
        ns = SpyOutputOsInfo (pOutput, dOutput, pdInfo);
        break;
        }
    case SPY_IO_SEGMENT:
        {
        if ((ns = SpyInputDword (&dValue,
                                 pInput, dInput))
            == STATUS_SUCCESS)
            {
            ns = SpyOutputSegment (dValue,
                                   pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_INTERRUPT:
        {
        if ((ns = SpyInputDword (&dValue,
                                 pInput, dInput))
            == STATUS_SUCCESS)
            {
            ns = SpyOutputInterrupt (dValue,
                                     pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_PHYSICAL:
```

```
            {
            if ((ns = SpyInputPointer (&pAddress,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                pa = MmGetPhysicalAddress (pAddress);

                ns = SpyOutputBinary (&pa, PHYSICAL_ADDRESS_,
                                      pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_CPU_INFO:
            {
            ns = SpyOutputCpuInfo (pOutput, dOutput, pdInfo);
            break;
            }
        case SPY_IO_PDE_ARRAY:
            {
            ns = SpyOutputBinary (X86_PDE_ARRAY, SPY_PDE_ARRAY_,
                                  pOutput, dOutput, pdInfo);
            break;
            }
        case SPY_IO_PAGE_ENTRY:
            {
            if ((ns = SpyInputPointer (&pAddress,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
```

```
            SpyMemoryPageEntry (pAddress, &spe);


            ns = SpyOutputBinary (&spe, SPY_PAGE_ENTRY_,
                                  pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_MEMORY_DATA:
        {
        if ((ns = SpyInputMemory (&smb,
                                  pInput, dInput))
            == STATUS_SUCCESS)
            {
            ns = SpyOutputMemory (&smb,
                                  pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_MEMORY_BLOCK:
        {
        if ((ns = SpyInputMemory (&smb,
                                  pInput, dInput))
            == STATUS_SUCCESS)
            {
            ns = SpyOutputBlock (&smb,
                                 pOutput, dOutput, pdInfo);
            }
        break;
        }
```

```
        case SPY_IO_HANDLE_INFO:
            {
            if ((ns = SpyInputHandle (&hObject,
                                      pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputHandleInfo (hObject,
                                          pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_HOOK_INFO:
            {
            ns = SpyOutputHookInfo (pOutput, dOutput, pdInfo);
            break;
            }
        case SPY_IO_HOOK_INSTALL:
            {
            if (((ns = SpyInputBool (&fReset,
                                     pInput, dInput))
                 == STATUS_SUCCESS)
                &&
                ((ns = SpyHookInstall (fReset, &dCount))
                 == STATUS_SUCCESS))
                {
                ns = SpyOutputDword (dCount,
                                     pOutput, dOutput, pdInfo);
                }
            break;
```

```
        }
    case SPY_IO_HOOK_REMOVE:
        {
        if (((ns = SpyInputBool (&fReset,
                                 pInput, dInput))
             == STATUS_SUCCESS)
            &&
            ((ns = SpyHookRemove (fReset, &dCount))
             == STATUS_SUCCESS))
            {
            ns = SpyOutputDword (dCount,
                                 pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_HOOK_PAUSE:
        {
        if ((ns = SpyInputBool (&fPause,
                                pInput, dInput))
            == STATUS_SUCCESS)
            {
            fPause = SpyHookPause (fPause);

            ns = SpyOutputBool (fPause,
                                pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_HOOK_FILTER:
```

```
            {
            if ((ns = SpyInputBool (&fFilter,
                                    pInput, dInput))
                == STATUS_SUCCESS)
                {
                fFilter = SpyHookFilter (fFilter);

                ns = SpyOutputBool (fFilter,
                                    pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_HOOK_RESET:
            {
            SpyHookReset ();
            ns = STATUS_SUCCESS;
            break;
            }
        case SPY_IO_HOOK_READ:
            {
            if ((ns = SpyInputBool (&fLine,
                                    pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputHookRead (fLine,
                                        pOutput, dOutput, pdInfo);
                }
            break;
            }
```

```
        case SPY_IO_HOOK_WRITE:
            {
            SpyHookWrite (pInput, dInput);
            ns = STATUS_SUCCESS;
            break;
            }
        case SPY_IO_MODULE_INFO:
            {
            if ((ns = SpyInputPointer (&pbName,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputModuleInfo (pbName,
                                          pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_PE_HEADER:
            {
            if ((ns = SpyInputPointer (&pAddress,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputPeHeader (pAddress,
                                        pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_PE_EXPORT:
```

```
            {
            if ((ns = SpyInputPointer (&pAddress,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputPeExport (pAddress,
                                        pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_PE_SYMBOL:
            {
            if ((ns = SpyInputPointer (&pbName,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputPeSymbol (pbName,
                                        pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_CALL:
            {
            if ((ns = SpyInputBinary (&sci, SPY_CALL_INPUT_,
                                      pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputCall (&sci,
                                    pOutput, dOutput, pdInfo);
```

```
                }
            break;
            }
        }
    MUTEX_RELEASE (pDeviceContext->kmDispatch);
    return ns;
    }
```

列表 4-7. Spy 驱动程序的内部命令 Dispatcher

```
#define  CTL_CODE  (DeviceType, Function, Method, Access) \
        (( (DeviceType) << 16 ) | ( Access << 14 ) | ( (Function) << 2 ) (Method) )
```

列表 4-8. 用来构建 I/O 控制编码的 CTL_CODE()宏

DDK 的主要头文件 ntddk.h 和 SDK 中的 Win32 文件 winioctl.h 均定义了一个简单但非常有用的宏---- CTL_CLOSE()，如**列表 4-8** 所示。该宏可方便的建立**图 4-6** 所示的 IOCTL 编码。该编码中的四个部分分别服务于以下四个目的：

1. DeviceType 这是一个 16 位的设备类型 ID。ntddk.h 列出了一对预定义的类型，由符号常量 FILE_DEVICE_*表示。0x0000 到 0x7FFF 保留给微软内部使用，开发人员可使用 0x8000 到 0xFFFF。Spy 驱动程序定义了它自己的设备 ID:FILE_DEVICE_SPY，其值为 0x8000。
2. 2 位的访问检查值用来确定 IOCTL 操作所需的访问权限。可能的值有：FILE_ANY_ACCESS (0)，FILE_READ_ACCESS (1)，FILE_WRITE_ACCESS (2)和最后两个的组合：FILE_READ_ACCESS | FILE_WRITE_ACCESS (3)。详见 ntddk.h。
3. 12 个位的 ID 表示所选择的操作函数，所选操作将由设备来执行。0x0000 到 0x7FFF 保留给微软内部使用，开发人员可使用 0x8000 到 0xFFFF。Spy 设备采用的 IOCTL 函数 ID 位于 0x8000 到 0xFFFF。
4. 传输模式占用 2 个位，可在四个可用 I/O 传输模式中选择一个，这四个模式为：METHOD_BUFFERED (0)，METHOD_IN_DIRECT (1)，METHOD_OUT_DIRECT (2)和 METHOD_NETTHER (3)，可在 ntddk.h 中找到这些定义。Spy 设备针对所有请求使用 METHOD_BUFFERED，这是一个非常安全但有些慢的模式，因为数据需要在客户端和

系统缓冲区之间进行复制。因为 Memory Spy 对 I/O 的处理速度并不敏感，所以选择安全是一个不错的注意。如果你希望知道其他模式的细节，请参考我在 Windows Developer’s Journal(Schreiber 1997)发表的文章“A Spy Filter Driver for Windows NT”

表 4-2 列出了 w2k_spy.sys 支持的所有 IOCTL 函数。0 到 10 的函数 ID 为最基本的内存探测函数，绝大部分的任务都会用到它们；本章稍候将讨论它们。剩余的函数 ID 从 11 到 23 分属于不同的 IOCTL 组，在下一章我们将讨论它们，在下一章，我们将讨论 Native API hook 和在用户模式下调用内核。注意某些 IOCTL 编码需要写入权限，由第 15 号位表示（参见**图 4-6**）。确切的说，所有形如 0x80006nnn 的 IOCTL 命令只需读权限，而形如 0x8000Ennn 的命令需要读/写权限。典型的要求读权限的例子是 CreateFile()，它通过指定 dwDesiredAccess 参数为 GENERIC_READ 和 GENERIC_WRITE 的组合来打开设备。

表 4-2 最左面的函数名称同样出现在 SpyDispatcher()（见**列表 4-7**）中那个庞大的 switch/case 语句中。这些函数首先获取设备的 dispatcher mutex，这样就能保证，如果一个以上的客户端或一个多线程的程序和设备通讯时，在同一时间只有一个请求被执行。MUTEX_WAIT()是 KeWaitForMutexObject()的外包宏（wrapper marco），KeWaitForMutexObject()至少需要 5 个参数。KeWaitForMutexObject()本身也是一个宏，它将传入的参数向前传递给 KeWaitForSingleObject()。**列表 4-9** 给出了 MUTEX_WAIT()以及它的伙伴 MUTEX_RELEASE()和 MUTEX_INITIALIZE()。在 mutex 对象变为有信号（signaled）状态后，根据接收到的 IOCTL 编码，SpyDispatcher()会转向不同的分支，每个分支都包含多种简单的代码序列。

表 4-2. w2k_spy.sys 支持的 IOCTL 函数

函数名称	ID	IOCTL 编码	描　　述
SPY_IO_VERSION_INFO	0	0x80006000	返回 Spy 的版本信息
SPY_IO_OS_INFO	1	0x80006004	返回操作系统的版本信息
SPY_IO_SEGMENT	2	0x80006008	返回一个段的属性
SPY_IO_INTERRUPT	3	0x8000600C	返回一个中断门的属性
SPY_IO_PHYSICAL	4	0x80006010	线性地址转换为物理地址
SPY_IO_CPU_INFO	5	0x80006014	返回特殊 CPU 寄存器的值
SPY_IO_PDE_ARRAY	6	0x80006018	返回位于 0xC0300000 的 PDE 数组

SPY_IO_PAGE_ENTRY	7	0x8000601C	Return the PDE or PTE of a linear address
SPY_IO_MEMORY_DATA	8	0x80006020	返回内存块中的内容
SPY_IO_MEMORY_BLOCK	9	0x80006024	返回内存块中的内容
SPY_IO_HANDLE_INFO	10	0x80006028	从句柄中查找对象属性
SPY_IO_HOOK_INFO	11	0x8000602C	返回有关 Native API Hook 的信息
SPY_IO_HOOK_INSTALL	12	0x8000E030	安装 Native API Hook
SPY_IO_HOOK_REMOVE	13	0x8000E034	移除一个 Native API Hook
SPY_IO_HOOK_PAUSE	14	0x8000E038	暂停/恢复 Hook 协议
SPY_IO_HOOK_FILTER	15	0x8000E03C	允许/禁止 Hook 协议过滤器
SPY_IO_HOOK_RESET	16	0x8000E040	清除 Hook 协议
SPY_IO_HOOK_READ	17	0x8000E044	从 Hook 协议中读取数据
SPY_IO_HOOK_WRITE	18	0x8000E048	向 Hook 协议中写入输入
SPY_IO_MODULE_INFO	19	0x8000E04C	返回已加载模块的信息
SPY_IO_PE_HEADER	20	0x8000E050	返回 IMAGE_NT_HEADERS 数据
SPY_IO_PE_EXPORT	21	0x8000E054	返回 IMAGE_EXPORT_DIRECTORY 数据
SPY_IO_PE_SYMBOL	22	0x8000E058	返回导出的系统符号的地址
SPY_IO_CALL	23	0x8000E05C	调用已加载模块中的一个函数

```
#define MUTEX_INITIALIZE(_mutex) \
        KeInitializeMutex        \
            (&(_mutex), 0)


#define MUTEX_WAIT(_mutex)       \
        KeWaitForMutexObject     \
            (&(_mutex), Executive, KernelMode, FALSE, NULL)


#define MUTEX_RELEASE(_mutex)    \
        KeReleaseMutex           \
```

```
            (&(_mutex), FALSE)
```

列表 4-9. 管理 Kernel-Mutex 的宏

SpyDispatcher()使用一对帮助函数来读取输入参数，以获取被请求的数据，并将产生的数据写入调用者提供的输出缓冲区中。就像前面提到的，内核模式的驱动程序总是过分挑剔的对待它接受到的来自用户模式的参数。以驱动程序的观点来看，所有用户模式下的代码都是有害的，它们除了让系统崩溃就什么都不知道了。这种多少有些多疑症的观点并不是荒谬的----仅有很小的比率会导致整个系统立即终止，同时出现蓝屏。因此，如果一个客户端程序说：“这是我的缓冲区-----它最多可容纳 4,096 个字节”，驱动程序不会接受这个缓冲区----即使该缓冲区指向有效的内存，并且其大小也是正确的。在 IOCTL 的可缓冲的 I/O 模式（Buffered I/O）下（如果 IOCTL 编码的模式部分为 METHOD_BUFFERED），系统会很小心的检查并分配一个足够容纳所有输入/输出数据的缓冲区。然而，其他的 I/O 传输模式，尤其是 METHOD_NETTHER，驱动程序会接受原始的用户模式的缓冲区指针。

尽管 Spy 设备使用可缓冲的 I/O，但它还是会检查输入/输出缓冲区的有效性。因为客户端程序传入的数据可能比所需的少或者提供的缓冲区不够容纳输出数据。系统不能捕获这些语意错误，因为它不知道在一次 IOCTL 传输中所传输的数据的类型。因此，SpyDispatcher()调用帮助函数 SpyInput*()和 SpyOutput*()来从 I/O 缓冲区中复制或写入数据。这些函数仅在缓冲区大小与操作的需求相匹配时才执行。**列表 4-10** 给出了基本的输入函数，**列表 4-11** 给出了基本的输出函数。SpyInputBinary()和 SpyOutputBinary()被广泛的使用，它们测试缓冲区的大小，如果 OK，则使用 Windows 2000 运行时库函数 RtlCopyMemory()复制被请求的数据。剩余的函数只是上述两个基本函数的简单外包，用来操作常见的数据类型 DWORD，BOOL，PVOID 和 HANDLE 等。SpyOutputBlock()复制由调用者在 SPY_MEMORY_BLOCK 结构中指定的数据块，当然这需要首先验证请求范围内的字节都是可读的。如果传入的输入缓冲区的大小不正确，SpyInput*()函数将返回 STATUS_INVALID_BUFFER_SIZE，如果输出缓冲区比需要的小，SpyOutput*()函数将返回 STATUS_BUFFER_TOO_SMALL。

```
NTSTATUS SpyInputBinary (PVOID  pData,
                         DWORD  dData,
                         PVOID  pInput,
                         DWORD  dInput)
    {
```

```
    NTSTATUS ns = STATUS_INVALID_BUFFER_SIZE;


    if (dData <= dInput)
        {
        RtlCopyMemory (pData, pInput, dData);
        ns = STATUS_SUCCESS;
        }
    return ns;
    }


// ------------------------------------------------------------------


NTSTATUS SpyInputDword (PDWORD pdValue,
                        PVOID  pInput,
                        DWORD  dInput)
    {
    return SpyInputBinary (pdValue, DWORD_, pInput, dInput);
    }


// ------------------------------------------------------------------


NTSTATUS SpyInputBool (PBOOL  pfValue,
                       PVOID  pInput,
                       DWORD  dInput)
    {
    return SpyInputBinary (pfValue, BOOL_, pInput, dInput);
    }


// ------------------------------------------------------------------
```

```
NTSTATUS SpyInputPointer (PPVOID ppAddress,
                          PVOID  pInput,
                          DWORD  dInput)
    {
    return SpyInputBinary (ppAddress, PVOID_, pInput, dInput);
    }

// ------------------------------------------------------------

NTSTATUS SpyInputHandle (PHANDLE phObject,
                         PVOID   pInput,
                         DWORD   dInput)
    {
    return SpyInputBinary (phObject, HANDLE_, pInput, dInput);
    }
```

列表 4-10. 从 IOCTL 缓冲区中读取输入数据

```
NTSTATUS SpyOutputBinary (PVOID  pData,
                          DWORD  dData,
                          PVOID  pOutput,
                          DWORD  dOutput,
                          PDWORD pdInfo)
    {
    NTSTATUS ns = STATUS_BUFFER_TOO_SMALL;

    *pdInfo = 0;

    if (dData <= dOutput)
```

```
        {
        RtlCopyMemory (pOutput, pData, *pdInfo = dData);
        ns = STATUS_SUCCESS;
        }
    return ns;
    }

// ------------------------------------------------------------

NTSTATUS SpyOutputBlock (PSPY_MEMORY_BLOCK psmb,
                         PVOID             pOutput,
                         DWORD             dOutput,
                         PDWORD            pdInfo)
    {
    NTSTATUS ns = STATUS_INVALID_PARAMETER;

    if (SpyMemoryTestBlock (psmb->pAddress, psmb->dBytes))
        {
        ns = SpyOutputBinary (psmb->pAddress, psmb->dBytes,
                              pOutput, dOutput, pdInfo);
        }
    return ns;
    }

// ------------------------------------------------------------

NTSTATUS SpyOutputDword (DWORD  dValue,
                         PVOID  pOutput,
                         DWORD  dOutput,
```

```
                        PDWORD pdInfo)
    {
    return SpyOutputBinary (&dValue, DWORD_,
                            pOutput, dOutput, pdInfo);
    }

// -------------------------------------------------------------

NTSTATUS SpyOutputBool (BOOL   fValue,
                        PVOID  pOutput,
                        DWORD  dOutput,
                        PDWORD pdInfo)
    {
    return SpyOutputBinary (&fValue, BOOL_,
                            pOutput, dOutput, pdInfo);
    }

// -------------------------------------------------------------

NTSTATUS SpyOutputPointer (PVOID  pValue,
                           PVOID  pOutput,
                           DWORD  dOutput,
                           PDWORD pdInfo)
    {
    return SpyOutputBinary (&pValue, PVOID_,
                            pOutput, dOutput, pdInfo);
    }
```

列表 4-11. 向 IOCTL 的缓冲区中写入数据

你可能注意到**列表 4-7** 中的 SpyDispatcher()还引用了其他的 SpyInput*()和 SpyOutput*()函数。尽管这些函数最终还是调用 SpyInputBinary()和 SpyOutputBinary()，但它们还是比**列表 4-10** 和 **4-11** 中的基本函数要复杂些，因此，稍后我们在讨论它们。现在，让我们从 SpyDispatcher()开始，一步步的分析它的 switch/case 语句。

4.2.3、IOCTL 函数 SPY_IO_VERSION_INFO

IOCTL 的 SPY_IO_VERSION_INFO 函数用有关 Spy 驱动自身的数据填充调用者提供的 SPY_VERSION_INFO 结构。该功能不需要输入参数，需要使用 SpyOutputVersionInfo()帮助函数。**列表 4-12** 给出了该函数和 SPY_VERSION_INFO 结构，该函数很简单，它将 dVersion 成员设置为 SPY_VERSION 常量（当前是 100，表示 V1.00），该常量定义于 w2k_spy.h 中。然后复制驱动程序的符号化名称，即字符串常量 DRV_NAME(“SBS Windows 2000 Spy Device”)到 awName 成员。通过整除 dVersion 可获取主版本号，剩下的是次版本号。

```
typedef struct _SPY_VERSION_INFO
    {
    DWORD dVersion;
    WORD  awName [SPY_NAME];
    }
    SPY_VERSION_INFO, *PSPY_VERSION_INFO, **PPSPY_VERSION_INFO;


#define SPY_VERSION_INFO_ sizeof (SPY_VERSION_INFO)


NTSTATUS SpyOutputVersionInfo (PVOID  pOutput,
                               DWORD  dOutput,
                               PDWORD pdInfo)
    {
    SPY_VERSION_INFO svi;


    svi.dVersion = SPY_VERSION;
```

```
    wcscpyn (svi.awName, USTRING (CSTRING (DRV_NAME)), SPY_NAME);


    return SpyOutputBinary (&svi, SPY_VERSION_INFO_,
                            pOutput, dOutput, pdInfo);
    }
```

列表 4-12. 获取 Spy 驱动程序的版本信息

4.2.4、IOCTL 函数 SPY_IO_OS_INFO

该函数比上一个有趣的多。它是另一个只有输出的函数，不需要输入参数，使用几个操作系统的内部参数来填充调用者提供的 SPY_OS_INFO 结构。**列表 4-13** 列出了该结构的定义，和 Dispatcher 调用的 SpyOutputOsInfo()帮助函数。有些结构体成员只是被简单的设为定义于 DDK 头文件和 w2k_spy.h 中的常量;其他的将被设为从几个内部的内核变量和结构体中读取的当前值。在第二章中，你已经了解了变量 NtBuildNumber 和 NtGlobalFlag（由 ntoskrnl.exe 导出，参见**附录 B** 中的**表 B-1**）。和其他的 Nt*符号不同，这两个符号不指向 API 函数，而是指向位于内核的.data section 中的变量。在 Win32 世界里，导出变量是十分罕见的。不过， Windows 2000 的几个内核模块都使用了这一技术。Ntoskrnl.exe 导出了至少 55 个变量，ntdll.dll 提供了 4 个，hal.dll 提供了 1 个。SpyOutputOsInfo()将从 ntoskrnl.exe 导出的变量中复制 MmHighestUserAddress、MmUserProbeAddress、MmSystemRangeStart、NtGlobalFlag、KeI386MachineType、KeNumberProcessors 和 NtBuildNumber 到输出缓冲区中。

当一个模块从另一个模块中导入数据时，它需要使用 extern 关键字来通知编译器和链接器。这会使链接器生成一个进入模块导出节的入口，并会解析符号名以确定其地址。有些 extern 声明已经包含在 ntddk.h。**列表 4-13** 给出了缺失的那些 extern 声明。

```
extern PWORD                    NlsAnsiCodePage;
extern PWORD                    NlsOemCodePage;
extern PWORD                    NtBuildNumber;
extern PDWORD                   NtGlobalFlag;
```

```
extern PDWORD                    KeI386MachineType;


typedef struct _SPY_OS_INFO
    {
    DWORD   dPageSize;
    DWORD   dPageShift;
    DWORD   dPtiShift;
    DWORD   dPdiShift;
    DWORD   dPageMask;
    DWORD   dPtiMask;
    DWORD   dPdiMask;
    PX86_PE PteArray;
    PX86_PE PdeArray;
    PVOID   pLowestUserAddress;
    PVOID   pThreadEnvironmentBlock;
    PVOID   pHighestUserAddress;
    PVOID   pUserProbeAddress;
    PVOID   pSystemRangeStart;
    PVOID   pLowestSystemAddress;
    PVOID   pSharedUserData;
    PVOID   pProcessorControlRegion;
    PVOID   pProcessorControlBlock;
    DWORD   dGlobalFlag;
    DWORD   dI386MachineType;
    DWORD   dNumberProcessors;
    DWORD   dProductType;
    DWORD   dBuildNumber;
    DWORD   dNtMajorVersion;
    DWORD   dNtMinorVersion;
```

```
    WORD    awNtSystemRoot [MAX_PATH];
    }
    SPY_OS_INFO, *PSPY_OS_INFO, **PPSPY_OS_INFO;

#define SPY_OS_INFO_ sizeof (SPY_OS_INFO)

NTSTATUS SpyOutputOsInfo (PVOID  pOutput,
                          DWORD  dOutput,
                          PDWORD pdInfo)
    {
    SPY_SEGMENT     ss;
    SPY_OS_INFO     soi;
    NT_PRODUCT_TYPE NtProductType;
    PKPCR           pkpcr;

    NtProductType = (SharedUserData->ProductTypeIsValid
                     ? SharedUserData->NtProductType
                     : 0);

    SpySegment (X86_SEGMENT_FS, 0, &ss);
    pkpcr = ss.pBase;

    soi.dPageSize               =  PAGE_SIZE;
    soi.dPageShift              =  PAGE_SHIFT;
    soi.dPtiShift               =  PTI_SHIFT;
    soi.dPdiShift               =  PDI_SHIFT;
    soi.dPageMask               =  X86_PAGE_MASK;
    soi.dPtiMask                =  X86_PTI_MASK;
    soi.dPdiMask                =  X86_PDI_MASK;
```

```
    soi.PteArray                =  X86_PTE_ARRAY;
    soi.PdeArray                =  X86_PDE_ARRAY;
    soi.pLowestUserAddress      =  MM_LOWEST_USER_ADDRESS;
    soi.pThreadEnvironmentBlock =  pkpcr->NtTib.Self;
    soi.pHighestUserAddress     = *MmHighestUserAddress;
    soi.pUserProbeAddress       =  (PVOID) *MmUserProbeAddress;
    soi.pSystemRangeStart       = *MmSystemRangeStart;
    soi.pLowestSystemAddress    =  MM_LOWEST_SYSTEM_ADDRESS;
    soi.pSharedUserData         =  SharedUserData;
    soi.pProcessorControlRegion =  pkpcr;
    soi.pProcessorControlBlock  =  pkpcr->Prcb;
    soi.dGlobalFlag             = *NtGlobalFlag;
    soi.dI386MachineType        = *KeI386MachineType;
    soi.dNumberProcessors       = *KeNumberProcessors;
    soi.dProductType            =  NtProductType;
    soi.dBuildNumber            = *NtBuildNumber;
    soi.dNtMajorVersion         =  SharedUserData->NtMajorVersion;
    soi.dNtMinorVersion         =  SharedUserData->NtMinorVersion;

    wcscpyn (soi.awNtSystemRoot, SharedUserData->NtSystemRoot,
             MAX_PATH);

    return SpyOutputBinary (&soi, SPY_OS_INFO_,
                            pOutput, dOutput, pdInfo);
    }
```

列表 4-13. 获取有关操作系统的信息

SPY_OS_INFO 结构的剩余成员会由位于内存中的系统数据结构填充。例如，SpyOutputOsInfo()将内核的进程控制区域（Kernel’s Processor Control Region, KPCR）的基地址赋值给 pProcessorControlRegion 成员。KPCR 是一个非常重要的数据结构，该结

构包含很多线程相关的数据项，因此，它位于自己的内存段中，该内存段的地址由 CPU 的 FS 寄存器给出。Windows NT 4.0 和 Windows 2000 都将 FS 指向处于内核模式的线性地址 0xFFDFF000。SpyOutputOsInfo()调用 SpySegment()函数（稍后讨论它）来查询 FS 段在线性地址空间中的基地址。这个段中还包含内核的进程控制块（Kernel’s Processor Control Block, KPRCB),KPCR 结构的 Prcb 成员指向 KPRCB 结构的首地址,紧随其后的是一个 CONTEXT 结构，该结构包含当前线程的底层 CPU 信息。KPCR、KPRCB 和 CONTEXT 结构定义在 ntddk.h 头文件中。

列表 4-13 中引用的另一个内部数据结构是 SharedUserData。该结构实际上是一个由一个“众所周知的地址”通过类型转化（TypeCast）得来的结构体指针。**列表 4-14** 给出了它在 ntddk.h 中的定义。那个“众所周知的地址”位于线性地址空间中，它会在编译时被设置，因此不需要花费额外的时间或进行配置。显然，SharedUserData 是一个指向 KUSER_SHARED_DATA 结构的指针，该结构的基地址在 0xFFDF0000（这是一个线性地址）。这个内存区域由系统和用户模式的应用程序共享，它包含像操作系统版本号这样的数据，SpyOutputOsInfo()将该版本数据复制到 SPY_OS_INFO 结构（由调用者提供）的 dNtMajorVersion 和 dNtMinorVersion 成员。就像我稍后要展示的那样，KUSER_SHARED_DATA 结构将被映射到 0x7FFE0000，这样用户模式的代码就可以访问它了。

在对 Spy 设备的 IOCTL 函数的讲解之后还将提供了一个示例程序，该示例程序会把返回的数据显示在屏幕上。

```
#define  KI_USER_SHARED_DATA  0xFFDF0000
#define  SharedUserData ((KUSER_SHARED_DATA *const)KI_USER_SHARED_DATA)
```

列表 4-14. SharedUserData 结构定义

4.2.5、IOCTL 函数 SPY_IO_SEGMENT

到现在讨论以变得更加有趣了。SPY_IO_SEGMENT 函数通过一些更底层的操作来查询指定段的属性，调用者需要首先给出一个选择器（selector）。SpyDispatcher()首先调用 SpyInputDword()来获取由调用程序传入的选择器的值。你可能还记得选择器（selector）是一个 16 位的数。不过，只要可能，我就会尝试避免使用 16 位的数据类型，这是因为原生的 WORD 在 i386 CPU 的 32 位模式下是 32 位的 DWORD 类型。因此，我将选择器参数扩展为 DWORD，不过其高 16 位总是 0。如果 SpyInputDword()报告操作成功，接下来就会调用

SpyOutputSegemnt()函数（**列表 4-15** 给出了此函数）。不管 SpySegment()帮助函数如何，SpyOutputSegemnt()总是返回到调用者。基本上来说，SpySegment()将填充 SPY_SEGMENT 结构，该结构定义于**列表 4-15** 的顶部。它以 X86_SELECTOR 结构（参见**列表 4-2**）的形式给出选择器的值，紧随其后的是 64 位的 X86_DESCRIPTOR，以及相应的段基址，段的大小限制以及一个名为 fOk 的标志，该标志用来指出 SPY_SEGMENT 结构是否有效。在稍后的一些函数中需要一次返回多个段的属性，利用 fOk 成员，调用者就可以将无效的段信息从输出数据中筛选出来。

```
typedef struct _SPY_SEGMENT
    {
    X86_SELECTOR   Selector;
    X86_DESCRIPTOR Descriptor;
    PVOID          pBase;
    DWORD          dLimit;
    BOOL           fOk;
    }
    SPY_SEGMENT, *PSPY_SEGMENT, **PPSPY_SEGMENT;

#define SPY_SEGMENT_ sizeof (SPY_SEGMENT)

NTSTATUS SpyOutputSegment (DWORD  dSelector,
                           PVOID  pOutput,
                           DWORD  dOutput,
                           PDWORD pdInfo)
    {
    SPY_SEGMENT ss;

    SpySegment (X86_SEGMENT_OTHER, dSelector, &ss);

    return SpyOutputBinary (&ss, SPY_SEGMENT_,
```

```
                            pOutput, dOutput, pdInfo);
    }



BOOL SpySegment (DWORD        dSegment,
                 DWORD        dSelector,
                 PSPY_SEGMENT pSegment)
    {
    BOOL fOk = FALSE;


    if (pSegment != NULL)
        {
        fOk = TRUE;


        if (!SpySelector   (dSegment, dSelector,
                            &pSegment->Selector))
            {
            fOk = FALSE;
            }
        if (!SpyDescriptor (&pSegment->Selector,
                            &pSegment->Descriptor))
            {
            fOk = FALSE;
            }
        pSegment->pBase  =
            SpyDescriptorBase  (&pSegment->Descriptor);


        pSegment->dLimit =
            SpyDescriptorLimit (&pSegment->Descriptor);
```

```
        pSegment->f0k = f0k;
        }
    return f0k;
    }
```

列表 4-15. 查询段的属性

SpySegment()函数依赖其他几个帮助函数，以构建 SPY_SEGMENT 结构的某些部分。首先，SpySelector()复制一个选择器的值到传入的 X86_SELECTOR 结构中。如果 SpySelector()函数的第一个参数 dSegment 被设置为 X86_SEGMENT_OTHER（即 0），dSelector 参数将假定已经指定了一个有效的选择器值，因此该值将被简单的附给输出结构 X86_SELECTOR 的 wValue 成员。否则，dSelector 将被忽略，dSegment 会被用于一个 switch/case 结构中以便选择一个段寄存器或任务寄存器 TR。注意，这种请求需要少量的嵌入式汇编，C 语言没有提供标准的方法访问处理器相关的特性，如段寄存器。

```
#define X86_SEGMENT_OTHER          0
#define X86_SEGMENT_CS              1
#define X86_SEGMENT_DS              2
#define X86_SEGMENT_ES              3
#define X86_SEGMENT_FS              4
#define X86_SEGMENT_GS              5
#define X86_SEGMENT_SS              6
#define X86_SEGMENT_TSS             7

//-------------------------------------------------------------

BOOL SpySelector (DWORD          dSegment,
                  DWORD          dSelector,
                  PX86_SELECTOR pSelector)
    {
    X86_SELECTOR Selector = {0, 0};
```

```
BOOL         fOk      = FALSE;


if (pSelector != NULL)
    {
    fOk = TRUE;


    switch (dSegment)
        {
        case X86_SEGMENT_OTHER:
            {
            if (fOk = ((dSelector >> X86_SELECTOR_SHIFT)
                       <= X86_SELECTOR_LIMIT))
                {
                Selector.wValue = (WORD) dSelector;
                }
            break;
            }
        case X86_SEGMENT_CS:
            {
            __asm mov Selector.wValue, cs
            break;
            }
        case X86_SEGMENT_DS:
            {
            __asm mov Selector.wValue, ds
            break;
            }
        case X86_SEGMENT_ES:
            {
```

```
            __asm mov Selector.wValue, es
            break;
            }
        case X86_SEGMENT_FS:
            {
            __asm mov Selector.wValue, fs
            break;
            }
        case X86_SEGMENT_GS:
            {
            __asm mov Selector.wValue, gs
            break;
            }
        case X86_SEGMENT_SS:
            {
            __asm mov Selector.wValue, ss
            break;
            }
        case X86_SEGMENT_TSS:
            {
            __asm str Selector.wValue
            break;
            }
        default:
            {
            fOk = FALSE;
            break;
            }
        }
```

```
        RtlCopyMemory (pSelector, &Selector, X86_SELECTOR_);
        }
    return fOk;
    }
```

列表 4-16. 获取选择器（selector）的值

SpyDispatcher()将从一个 64 位的描述符中读取数据，段选择器指向该描述符（见**列表 4-17**）。像你记得的那样，所有的选择器都包含一个表指示符（Table Indicator, TI）位，以确定选择器引用的描述符是位于 GDT（TI=0）中还是 LDT（TI=1）中。**列表 4-17** 的上半部分处理了是 LDT 的情况。首先，使用汇编指令 SLDT 和 SGDT 分别读取 LDT 选择器的值以及段的大小限制和 GDT 的基地址。还记得 GDT 的线性基地址是显示指定的，而 LDT 是由 GDT 中的选择器间接引用的吗？所以，SpyDispatcher()会首先验证 LDT 选择器的值。如果段选择器不为空并且没有超过 GDT 的限制，就会调用 SpyDescriptorType()、SpyDescriptorLimit()和 SpyDescriptorBase()(**列表 4-17** 给出了这些函数)来获取 LDT 的基本属性：

- SpyDescriptorType()返回描述符的类型数据及其 S 位域（参见**列表 4-2**）。LDT 选择器必须指向一个类型为 X86_DESCRIPTOR_SYS_LDT 的系统描述符。
- SpyDescriptorLimit()从描述符的 Limit1、Limit2 这两个位域中汇总段的大小限制。根据描述符的 G 标志指定的内存分配粒度的不同，其处理方式也会不同。
- SpyDescriptorBase()只是简单的通过适当的组织描述符的 Base1、Base2 和 Base3 位域以获取一个 32 位的线性地址。

```
BOOL SpyDescriptor (PX86_SELECTOR   pSelector,
                    PX86_DESCRIPTOR pDescriptor)
    {
    X86_SELECTOR    ldt;
    X86_TABLE       gdt;
    DWORD           dType, dLimit;
    BOOL            fSystem;
    PX86_DESCRIPTOR pDescriptors = NULL;
    BOOL            fOk          = FALSE;
```

```
if (pDescriptor != NULL)
    {
    if (pSelector != NULL)
        {
        if (pSelector->TI) // ldt descriptor
            {
            __asm
                {
                sldt ldt.wValue
                sgdt gdt.wLimit
                }
            if ((!ldt.TI) && ldt.Index &&
                ((ldt.wValue & X86_SELECTOR_INDEX)
                 <= gdt.wLimit))
                {
                dType  = SpyDescriptorType  (gdt.pDescriptors +
                                             ldt.Index,
                                             &fSystem);

                dLimit = SpyDescriptorLimit (gdt.pDescriptors +
                                             ldt.Index);

                if (fSystem && (dType == X86_DESCRIPTOR_SYS_LDT)
                    &&
                    ((DWORD) (pSelector->wValue
                              & X86_SELECTOR_INDEX)
                     <= dLimit))
                    {
                    pDescriptors =
```

```
                        SpyDescriptorBase (gdt.pDescriptors +
                                                ldt.Index);
                    }
                }
            }
        else // gdt descriptor
            {
            if (pSelector->Index)
                {
                __asm
                    {
                    sgdt gdt.wLimit
                    }
                if ((pSelector->wValue & X86_SELECTOR_INDEX)
                    <= gdt.wLimit)
                    {
                    pDescriptors = gdt.pDescriptors;
                    }
                }
            }
        }
    if (pDescriptors != NULL)
        {
        RtlCopyMemory (pDescriptor,
                       pDescriptors + pSelector->Index,
                       X86_DESCRIPTOR_);
        fOk = TRUE;
        }
    else
```

```
            {
            RtlZeroMemory (pDescriptor,
                           X86_DESCRIPTOR_);
            }
        }
    return fOk;
    }

// ------------------------------------------------------------------

PVOID SpyDescriptorBase (PX86_DESCRIPTOR pDescriptor)
    {
    return (PVOID) ((pDescriptor->Base1      ) |
                    (pDescriptor->Base2 << 16) |
                    (pDescriptor->Base3 << 24));
    }

// ------------------------------------------------------------------

DWORD SpyDescriptorLimit (PX86_DESCRIPTOR pDescriptor)
    {
    return (pDescriptor->G ? (pDescriptor->Limit1 << 12) |
                             (pDescriptor->Limit2 << 28) | 0xFFF
                           : (pDescriptor->Limit1      ) |
                             (pDescriptor->Limit2 << 16));
    }

// ------------------------------------------------------------------
```

```
DWORD SpyDescriptorType (PX86_DESCRIPTOR pDescriptor,
                         PBOOL           pfSystem)
    {
    if (pfSystem != NULL) *pfSystem = !pDescriptor->S;
    return pDescriptor->Type;
    }
```

列表 4-17. 获取描述符的值

如果选择器的 TI 位指定了一个 GDT 描述符，事情就简单了。再次使用 SGDT 指令来取出 GDT 在线性内存中的位置和大小，如果选择器指定的描述符索引位于适当的范围，pDescriptors 变量将被设置为指向 GDT 的基地址。对于 LDT 和 GDT 来说，pDescriptors 变量都不会为空。如果调用者传入的选择器是有效的，64 位的描述符值将被复制到调用者提供的 X86_DESCRIPTOR 结构中。否则，该结构的所有成员都会被 RtlZeroMemory()设为 0。

我们仍然在讨论**列表 4-15** 中的 SpySegment()函数。SpySelector()和 SpyDescriptor()调用已经解释了。只剩下最后的 SpyDescriptorBase()和 SpyDescriptorLimit()调用，不过你应该已经知道这些函数作了些什么(见**列表 4-17**)。如果 SpySelector()和 SpyDescriptor()成功,返回的 SPY_SEGMENT 结构将是有效的。SpyDescriptorBase()和 SpyDescriptorLimit()不会返回出错标志。因为它们不可能失败，如果提供的描述符无效，只是会让它们返回错误的数据而已。

4.2.6、IOCTL 函数 SPY_IO_INTERRUPT

SPY_IO_INTERRUP 类似于 SPY_IO_SEGEMT，不过该函数仅影响存储在系统中断描述符表（IDT）的中断描述符，不会涉及 LDT 或 GDT 描述符。IDT 最多可容纳 256 个描述符，这些描述符可用来描述任务门、中断门或陷阱门（参见 Intel 1999c, pp. 5-11ff）。顺便说一下，中断和陷阱在本质上十分相似，二者只存在微小的差异：在进入一个中断处理例程后，总是会屏蔽其他中断；而进入陷阱处理例程却不会修改中断标志。SPY_IO_INTERRUPT 的调用者提供一个 0 到 255 之间的中断号,该中断号将位于输入缓冲区中,而一个 SPY_INTERRUPT 结构将作为输出数据被存放到输出缓冲区中，如果成功返回，该结构中将包含对应的中断处理例程的属性。由 Dispatcher 调用的帮助函数 SpyOutputInterrupt()只是一个简单的外包函数，它实际上调用 SpyInterrupt()函数并且将需要返回的数据复制到输出缓冲区中。**列**

表4-18给出了这两个函数，以及它们操作的SPY_INTERRUPT结构。稍后一些，SpyInterrupt()函数将填充如下项目：

- Selector　用来指定一个任务状态段（Task-State Segment, TSS）或代码段（Code Segment）的选择器。代码段选择器用来确定中断或陷阱处理例程所在的段。
- Gate　用来表示一个64位的任务门、中断门或陷阱门描述符，由Selector确定其地址。
- Segment　包含段的属性，该段的地址由前面的Gate给出。
- pOffset　指定中断或陷阱处理例程的入口地址相对基地址的偏移量。这里的基地址是指中断或陷阱处理例程所在代码段的起始地址。因为任务门不包含偏移量，所以，如果输入的选择器指向一个TSS，则忽略该成员。
- fOk　一个标志变量，用来指示SPY_INTERRUPT结构中的数据是否有效。

通常情况下，TSS被用来保证一个错误情况可以被一个有效的任务处理。这是一个特殊的系统段类型（system segment type），它可以保存104个字节的进程状态信息，该信息在任务切换时，用来进行任务的恢复，如表4-3所示。当与任务相关的中断发生时，CPU总是强制切换该任务，并将所有的CPU寄存器保存到TSS中。Windows 2000在中断位置0x02（非屏蔽中断[NMI]，0x08[Double Fault]和0x12[堆栈段故障]）处保存任务门。剩余的位置指向中断处理例程。不使用的中断由一个哑元例程---KiUnexpectedInterruptNNN()处理，这里的NNN为一个十进制数。这些哑元例程最后都汇集到内部函数KiEndUnexpectedRange()，在这里，这些例程将依次进入KiUnexpectedInterruptTail()。

```
typedef struct _SPY_INTERRUPT
    {
    X86_SELECTOR Selector;
    X86_GATE     Gate;
    SPY_SEGMENT  Segment;
    PVOID        pOffset;
    BOOL         fOk;
    }
    SPY_INTERRUPT, *PSPY_INTERRUPT, **PPSPY_INTERRUPT;
```

```
#define SPY_INTERRUPT_ sizeof (SPY_INTERRUPT)


// -------------------------------------------------------------
NTSTATUS SpyOutputInterrupt (DWORD  dInterrupt,
                             PVOID  pOutput,
                             DWORD  dOutput,
                             PDWORD pdInfo)
    {
    SPY_INTERRUPT si;


    SpyInterrupt (dInterrupt, &si);


    return SpyOutputBinary (&si, SPY_INTERRUPT_,
                            pOutput, dOutput, pdInfo);
    }


// -------------------------------------------------------------
BOOL SpyInterrupt (DWORD          dInterrupt,
                   PSPY_INTERRUPT pInterrupt)
    {
    BOOL fOk = FALSE;


    if (pInterrupt != NULL)
        {
        if (dInterrupt <= X86_SELECTOR_LIMIT)
            {
            fOk = TRUE;


            if (!SpySelector (X86_SEGMENT_OTHER,
```

```
                            dInterrupt << X86_SELECTOR_SHIFT,
                            &pInterrupt->Selector))
                {
                fOk = FALSE;
                }
            if (!SpyIdtGate  (&pInterrupt->Selector,
                            &pInterrupt->Gate))
                {
                fOk = FALSE;
                }
            if (!SpySegment  (X86_SEGMENT_OTHER,
                            pInterrupt->Gate.Selector,
                            &pInterrupt->Segment))
                {
                fOk = FALSE;
                }
            pInterrupt->pOffset = SpyGateOffset (&pInterrupt->Gate);
            }
        else
            {
            RtlZeroMemory (pInterrupt, SPY_INTERRUPT_);
            }
        pInterrupt->fOk = fOk;
        }
    return fOk;
    }


// -------------------------------------------------------------
PVOID SpyGateOffset (PX86_GATE pGate)
```

```
{
return (PVOID) (pGate->Offset1 | (pGate->Offset2 << 16));
}
```

列表 4-18. 查询中断属性

表 4-3. 任务状态段（TSS）中的 CPU 状态域

偏移量	位数	ID	描　　述
0x00	16		前一个任务的链接
0x04	32	ESP0	Ring0 级的堆栈指针寄存器
0x08	16	SS0	Ring0 级的堆栈段寄存器
0x0C	32	ESP1	Ring1 级的堆栈指针寄存器
0x10	16	SS1	Ring1 级的堆栈段寄存器
0x14	32	ESP2	Ring2 级的堆栈指针寄存器
0x18	16	SS2	Ring2 级的堆栈段寄存器
0x1C	32	CR3	页目录基址寄存器（PDBR）
0x20	32	EIP	指令指针寄存器
0x24	32	EFLAGS	处理器标志寄存器
0x28	32	EAX	通用寄存器
0x2C	32	ECX	通用寄存器
0x30	32	EDX	通用寄存器
0x34	32	EBX	通用寄存器
0x38	32	ESP	堆栈指针寄存器
0x3C	32	EBP	基地址指针寄存器
0x40	32	ESI	源索引寄存器
0x44	32	EDI	目标索引寄存器
0x48	16	ES	扩展段寄存器
0x4C	16	CS	代码段寄存器
0x50	16	SS	堆栈段寄存器
0x54	16	DS	数据段寄存器
0x58	16	FS	附加的数据段寄存器#1

0x5C	16	GS	附加的数据段寄存器#2
0x60	16	LDT	本地描述符标的段选择器
0x64	1	1	调试陷阱标志
0x66	16		I/O Map 的基地址
0x68	-		CPU 状态信息结束

SpyInterrupt()调用的 SpySegment()、SpySelector()函数已经在**列表 4-5** 和**列表 4-16** 中给出。SpyGateOffset()位于**列表 4-18** 的末尾，它的工作和 SpyDescriptorBase()、SpyDescriptorLimit()类似，从 X86_GATE 结构中取出 Offset1 和 Offset2 位域，并适当的组织它们以构成一个 32 位地址。SpyIdtGaet()定义于**列表 4-19**。它与 SpyDescriptor()十分类似。汇编指令 SIDT 存储一个 48 位的值，该值就是 CPU 的 IDT 寄存器的内容，它由一个 16 位的表大小限制值和 IDT 的 32 位线性基地址构成。**列表 4-19** 中的剩余代码将选择器的描述符索引和 IDT 的大小限制值进行比较，如果 OK，则对应的中断描述符将被复制到调用者提供的 X86_GATE 结构中。否则，门结构的所有成员都将被设置为 0。

```
BOOL SpyIdtGate (PX86_SELECTOR pSelector,
                 PX86_GATE     pGate)
    {
    X86_TABLE idt;
    PX86_GATE pGates = NULL;
    BOOL      fOk    = FALSE;

    if (pGate != NULL)
        {
        if (pSelector != NULL)
            {
            __asm
                {
                sidt idt.wLimit
                }
            if ((pSelector->wValue & X86_SELECTOR_INDEX)
```

```
                <= idt.wLimit)
                {
                pGates = idt.pGates;
                }
            }
        if (pGates != NULL)
            {
            RtlCopyMemory (pGate,
                           pGates + pSelector->Index,
                           X86_GATE_);
            fOk = TRUE;
            }
        else
            {
            RtlZeroMemory (pGate, X86_GATE_);
            }
        }
    return fOk;
    }
```

列表 4-19. 获取 IDT 门的值

4.2.7、IOCTL 函数 SPY_IO_PHYSICAL

SPY_IO_PHYSICAL 函数很简单，它完全依赖于 ntoskrnl.exe 导出的 MmGetPhysicalAddress()函数。该 IOCTL 函数通过简单的调用 SpyInputPointer()（参见**列表 4-10**）来获取需要转换的线性地址，然后让 MmGetPhysicalAddress()查找对应的物理地址，最后将结果作为 PHYSICAL_ADDRESS 结构返回给调用者。注意，PHYSICAL_ADDRESS 是一个 64 位的 LARGE_INTEGER。在大多数 i386 系统上，其高 32 位总是为 0。不过，若系统启用了物理地址扩展（Physical Address Extension, PAE），并且安装的内存大于 4GB，这些位可能就是非 0 值了。

MmGetPhysicalAddress()使用起始于线性地址 0xC0000000 的 PTE 数组，来进行物理地址的查找。其基本的工作机制如下：

- 如果线性地址位于：0x80000000----0x9FFFFFFF，则其高 3 位将被设为零，最后产生的物理地址位于：0x00000000-----0x1FFFFFFF。
- 否则，线性地址的高 20 位将作为 PTE 数组（起始于 0xC0000000）的索引。
- 如果目标 PTE 的 P 位已被设置，这表示其对应得数据页存在于物理内存中。除了 20 位的 PFN 外，所有的 PTE 位都可以被剥离出来，线性地址最低的 12 位将作为在数据页中的偏移量被加到最后的 32 位物理地址上去。
- 如果数据页没有存在于物理内存中，MmGetPhysicalAddress()返回 0。

MmGetPhysicalAddress()假设内核内存范围：0x80000000----0x9FFFFFF 之外的所有线性地址都使用 4KB 的页。而其他函数，如 MmIsAddressValid()，会首先加载线性地址的 PDE，并且检查该 PDE 的 PS 位，以检查页大小是 4KB 还是 4MB。这是一个非常通用的方法，可以处理任意的内存配置。不过上述两个函数都会返回正确的结果，这是因为 Windows 2000 仅针对内存范围：0x80000000-----0x9FFFFFFF，使用 4MB 页。不过某些内核 API 函数，显然设计的比其它的灵活许多。

4.2.8、IOCTL 函数 SPY_IO_CPU_INFO

个别的 CPU 指令仅对运行于 Ring 0 级的代码有效，Ring 0 是五个特权级（Intel 系列的 CPU 只支持两个特权级：Ring0 和 Ring3）中级别最高的一个。用 Windows 术语来说，Ring 0 意味着内核模式（Kernel-mode）。这些被禁止的指令有：读取控制寄存器 CR0、CR2 和 CR3 的内容。因为这些寄存器中保存着非常有趣的信息，应用程序可能想要找到一个办法来访问它们，解决方案就是 SPY_IO_CPU_INFO 函数。如**列表 4-20** 所示，IOCTL 处理例程调用的 SpyOutputCpuInfo()函数使用了一些嵌入式汇编来读取控制寄存器，以及其他一些有价值的信息，比如 IDT 的内容，GDT 和 LDT 寄存器以及存储在寄存器 CS、DS、ES、FS、GS、SS 和 TR 中的段选择器。任务寄存器（Task Register, TR）还包含一个涉及当前任务的 TSS 的选择器。

```
typedef struct _SPY_CPU_INFO
    {
    X86_REGISTER cr0;
```

```
    X86_REGISTER cr2;
    X86_REGISTER cr3;
    SPY_SEGMENT  cs;
    SPY_SEGMENT  ds;
    SPY_SEGMENT  es;
    SPY_SEGMENT  fs;
    SPY_SEGMENT  gs;
    SPY_SEGMENT  ss;
    SPY_SEGMENT  tss;
    X86_TABLE    idt;
    X86_TABLE    gdt;
    X86_SELECTOR ldt;
    }
    SPY_CPU_INFO, *PSPY_CPU_INFO, **PPSPY_CPU_INFO;

#define SPY_CPU_INFO_ sizeof (SPY_CPU_INFO)

// -------------------------------------------------------------

NTSTATUS SpyOutputCpuInfo (PVOID  pOutput,
                           DWORD  dOutput,
                           PDWORD pdInfo)
    {
    SPY_CPU_INFO  sci;
    PSPY_CPU_INFO psci = &sci;

    __asm
        {
        push    eax
```

```
        push    ebx
        mov     ebx, psci

        mov     eax, cr0
        mov     [ebx.cr0], eax

        mov     eax, cr2
        mov     [ebx.cr2], eax

        mov     eax, cr3
        mov     [ebx.cr3], eax

        sidt    [ebx.idt.wLimit]
        mov     [ebx.idt.wReserved], 0

        sgdt    [ebx.gdt.wLimit]
        mov     [ebx.gdt.wReserved], 0

        sldt    [ebx.ldt.wValue]
        mov     [ebx.ldt.wReserved], 0

        pop     ebx
        pop     eax
        }
    SpySegment (X86_SEGMENT_CS,  0, &sci.cs);
    SpySegment (X86_SEGMENT_DS,  0, &sci.ds);
    SpySegment (X86_SEGMENT_ES,  0, &sci.es);
    SpySegment (X86_SEGMENT_FS,  0, &sci.fs);
    SpySegment (X86_SEGMENT_GS,  0, &sci.gs);
```

```
    SpySegment (X86_SEGMENT_SS,  0, &sci.ss);
    SpySegment (X86_SEGMENT_TSS, 0, &sci.tss);


    return SpyOutputBinary (&sci, SPY_CPU_INFO_,
                            pOutput, dOutput, pdInfo);
    }
```

列表 4-20. 查询 CPU 状态信息

可使用帮助函数 SpySegement()获取段选择器，在前面，我们已讨论过该函数。参见**列表 4-15**。

4.2.9、IOCTL 函数 SPY_IO_PDE_ARRAY

SPY_IO_PDE_ARRAY 是另一个普通的函数，它只是简单的把整个页目录（开始于地址 0xC0300000）复制到调用者提供的输出缓冲区中。该缓冲区采用**列表 4-21** 所示的 SPY_PDE_ARRAY 结构。你可能已猜到，该结构的大小正好是 4KB，它由 1,024 个 32 位的 PDE 组成。X86_PE 结构将在这里使用，X86_PE 结构代表一个一般化的页项（page entry），可在**列表 4-3** 中找到该结构的定义，常量 X86_PAGES_4M 定义在**列表 4-5**。SPY_PDE_ARRAY 的结构体成员总是页目录项(PDE)，X86_PE 结构可以是 X86_PDE_4M 类型，也可以是 X86_PDE_4KB 类型，这取决于 PDE 的 PS 位的取值。

在无法保证源数据页存在于物理内存时，就开始复制内存中的数据通常并不是一个好主意。不过，页目录是少数列外中的一个。在当前任务处于运行状态时，它的页目录总是存在于物理内存中。它不会被置换到页面文件中，除非另一个任务被置换进来。这就是为什么 CPU 的页目录基地址寄存器（PDBR）没有 P（present）位的原因，PDE 和 PTE 也类似。请参考**列表 4-3** 中的 X86_PDBR 结构的定义，以验证这一点。

```
typedef struct _SPY_PDE_ARRAY
    {
    X86_PE apde [X86_PAGES_4M];
    }
    SPY_PDE_ARRAY, *PSPY_PDE_ARRAY, **PPSPY_PDE_ARRAY;
```

```
#define SPY_PDE_ARRAY_ sizeof (SPY_PDE_ARRAY)
```

列表 4-21. SPY_PDE_ARRY 结构的定义

4.2.10、IOCTL 函数 SPY_IO_PAGE_ENTRY

如果你对给定线性地址的 page entry 感兴趣的话，这个函数就是一个很好的选择。**列表 4-22**给出了 SpyMemoryPageEntry()的内部细节,该函数就是用来处理 SPY_IO_PAGE_ENTRY 请求的。该函数返回的 SPY_PAGE_ENTRY 结构本质上是一个 X86_PE page entry（定义于**列表 4-3**），不过这里增加了两个新成员（为了使用方便）：dSize 和 fPresent。其中 dSize 成员用于说明页的大小（以字节为单位），其值不是 X86_PAGE_4KB（4,096 字节）就是 X86_PAGE_4MB（4,194,304 字节）；fPresent 成员用来说明页是否存在于物理内存中。这个标志必须和 SpyMemoryPageEntry()自身的返回值进行对比，即使 fPresent 为 FALSE，函数自身的返回值也可为 TRUE。此时，提供的线性地址时有效的，但它指向的数据页已被置换到了页面文件中。这种情况可通过设置 page entry 的第 10 位（即**列表 4-22** 中出现的 PageFile）来表示。当 P 位（该位属于 X86_PNPE 结构）被置 0 时，PageFile 就会被设置。请参考本章稍早讨论过的 X86_PNPE 结构的细节。X86_PNPE 结构代表一个 page-not-persent entry，该结构定义于**列表 4-3**。

SpyMemoryPageEntry()首先假定目标页是 4MB 页，然后，从系统的 PDE 数组（此数组起始于 0xC0300000）中复制指定线性地址的 PDE 到 SPY_PAGE_ENTRY 结构体的 pe 成员。如果 P 位不为 0，则肯定存在下一级的页或页表，所以接下来检查 PS 位以确定页面大小。如果 PS 位不为 0，则表示此 PDE 指向一个 4MB 数据页，工作到此就可结束了------SpyMemoryPageEntry()返回 TRUE，并且 SPY_PAGE_ENTRY 结构体的 fPresent 成员也同时被设为 TRUE。如果 PS 位为 0，则 PDE 指向的是一个 PTE，所以代码从起始于 0xC0000000 的数组中提取该 PTE，并检查它的 P 位。如果不为 0，则包含指定线性地址的 4KB 页存在于物理内存中，此时，SpyMemoryPageEntry()和 fPresent 都会报告 TRUE。否则，找到的必定是一个 page-not-present entry,因此 SpyMemoryPageEntry()返回 TRUE,不过仅当 PageFile 位不为 0 时，fPresent 成员才会被设为 FALSE。

```
typedef struct _SPY_PAGE_ENTRY
    {
    X86_PE pe;
```

```
    DWORD  dSize;
    BOOL   fPresent;
    }
    SPY_PAGE_ENTRY, *PSPY_PAGE_ENTRY, **PPSPY_PAGE_ENTRY;

#define SPY_PAGE_ENTRY_ sizeof (SPY_PAGE_ENTRY)

// -------------------------------------------------------------

BOOL SpyMemoryPageEntry (PVOID           pVirtual,
                         PSPY_PAGE_ENTRY pspe)
    {
    SPY_PAGE_ENTRY spe;
    BOOL           fOk = FALSE;

    spe.pe       = X86_PDE_ARRAY [X86_PDI (pVirtual)];
    spe.dSize    = X86_PAGE_4M;
    spe.fPresent = FALSE;

    if (spe.pe.pde4M.P)
        {
        if (spe.pe.pde4M.PS)
            {
            fOk = spe.fPresent = TRUE;
            }
        else
            {
            spe.pe    = X86_PTE_ARRAY [X86_PAGE (pVirtual)];
```

```
            spe.dSize = X86_PAGE_4K;


            if (spe.pe.pte4K.P)
                {
                fOk = spe.fPresent = TRUE;
                }
            else
                {
                fOk = (spe.pe.pnpe.PageFile != 0);
                }
            }
        }
    if (pspe != NULL) *pspe = spe;
    return fOk;
    }
```

列表 4-22. 查询 PDE 和 PTE

需要注意的是，SpyMemoryPageEntry()不能识别被置换出物理内存的 4MB 页。如果 PDE 指向的 4MB 页并不存在，将无法判断给定的线性地址是否有效的，以及该页是否还保存在当前页面文件中。4MB 页仅用于内核内存范围：0x80000000----0x9FFFFFFF。不过我从来没见过这样的一个页被置换出去，即使物理内存极端少的时候也没有过，因此我不需要检查任何与此相关的 page-not-present entries。

4.2.11、IOCTL 函数 SPY_IO_MEMORY_DATA

SPY_IO_MEMORY_DATA 函数是重量级函数中的一个，因为它可以复制任意数量的内存数据到调用者提供的缓冲区中。正如你可能还记得的那样，用户模式下的应用程序很容易传入一个无效的地址。因此，该函数在触及源地址之前，会非常谨慎的检验这些地址的有效性。记住，蓝屏可以潜伏在内核模式的任何地方。

调用程序通过传入一个 SPY_MEMORY_BLOCK 结构来请求一个内存块中的数据，在**列表 4-23** 的顶部给出了该结构体的定义，该结构体会指定内存块的地址和大小。为了方便，此

处的地址被定义为一个 union，以允许将其解释为一个字节类型的数组（PBYTE pbAddress）或解释为一个无类型的指针（PVOID pAddress）。**列表 4-23** 中的 SpyInputMemory()函数将从 IOCTL 的输入缓冲区中复制该结构。其搭档函数 SpyOutputMemory()（在**列表 4-23** 的末尾处)只是SpyMemoryReadBlock()的一个外包而已，**列表4-24**给出了SpyMemoryReadBlock()函数。SpyOutputMemory()的主要职责是在 SpyMemoryReadBlock()读取数据后，返回适当的 NTSTATUS 值。

SpyMemoryReadBlock()通过一个 SPY_MEMORY_DATA 结构返回它读到的内存数据。该结构定义于**列表 4-25**。我选择了一中不同的定义方式，因为 SPY_MEMORY_DATA 是一个针对变量大小的数据类型。基本上，它包含一个名为 smb 的 SPY_MEMORY_BLOCK 结构，随后是一个 WORD 类型的数组，名为 awData[]。该数组的长度由 smb 的 dBytes 成员给出。为了允许方便的按预定大小定义 SPY_MEMORY_DATA 的全局或局部实体，该结构的定义采用了一个宏----SPY_MEMORY_DATA_N()。该宏的唯一参数用于指定 awData[]数组的大小。实际的结构体定义在宏定义之后，它提供的结构体中包含一个长度为 0 的 awData[]数组。SPY_MEMORY_DATA__()宏首先计算 SPY_MEMORY_DATA 结构的全部大小，然后按这一大小分配结构中的数组，剩下的定义允许将 WORD 型的数据加入数组或从数组中取出。显然，每个 WORD 的低半位包含内存数据的字节数，高半位作为标志位。现在，仅有第 8 位有意义，用于表示位于 0—7 位的内存字节数是否有效。

```
typedef struct _SPY_MEMORY_BLOCK
    {
    union
        {
        PBYTE pbAddress;
        PVOID pAddress;
        };
    DWORD dBytes;
    }
    SPY_MEMORY_BLOCK, *PSPY_MEMORY_BLOCK, **PPSPY_MEMORY_BLOCK;


#define SPY_MEMORY_BLOCK_ sizeof (SPY_MEMORY_BLOCK)
```

```
// ------------------------------------------------------------------
NTSTATUS SpyInputMemory (PSPY_MEMORY_BLOCK psmb,
                         PVOID             pInput,
                         DWORD             dInput)
    {
    return SpyInputBinary (psmb, SPY_MEMORY_BLOCK_, pInput, dInput);
    }

// ------------------------------------------------------------------
NTSTATUS SpyOutputMemory (PSPY_MEMORY_BLOCK psmb,
                          PVOID             pOutput,
                          DWORD             dOutput,
                          PDWORD            pdInfo)
    {
    NTSTATUS ns = STATUS_BUFFER_TOO_SMALL;

    if (*pdInfo = SpyMemoryReadBlock (psmb, pOutput, dOutput))
        {
        ns = STATUS_SUCCESS;
        }
    return ns;
    }
```

列表 4-23. 处理内存块

```
DWORD SpyMemoryReadBlock (PSPY_MEMORY_BLOCK psmb,
                          PSPY_MEMORY_DATA  psmd,
                          DWORD             dSize)
    {
    DWORD i;
```

```
    DWORD n = SPY_MEMORY_DATA__ (psmb->dBytes);


    if (dSize >= n)
        {
        psmd->smb = *psmb;


        for (i = 0; i < psmb->dBytes; i++)
            {
            psmd->awData [i] =
                (SpyMemoryTestAddress (psmb->pbAddress + i)
                 ? SPY_MEMORY_DATA_VALUE (psmb->pbAddress [i], TRUE)
                 : SPY_MEMORY_DATA_VALUE (0, FALSE));
            }
        }
    else
        {
        if (dSize >= SPY_MEMORY_DATA_)
            {
            psmd->smb.pbAddress = NULL;
            psmd->smb.dBytes    = 0;
            }
        n = 0;
        }
    return n;
    }


// ------------------------------------------------------------


BOOL SpyMemoryTestAddress (PVOID pVirtual)
```

```
    {
    return SpyMemoryPageEntry (pVirtual, NULL);
    }

// ------------------------------------------------------------

BOOL SpyMemoryTestBlock (PVOID pVirtual,
                         DWORD dBytes)
    {
    PBYTE pbData;
    DWORD dData;
    BOOL  fOk = TRUE;

    if (dBytes)
        {
        pbData = (PBYTE) ((DWORD_PTR) pVirtual & X86_PAGE_MASK);
        dData  = (((dBytes + X86_OFFSET_4K (pVirtual) - 1)
                   / PAGE_SIZE) + 1) * PAGE_SIZE;
        do  {
            fOk = SpyMemoryTestAddress (pbData);

            pbData += PAGE_SIZE;
            dData  -= PAGE_SIZE;
            }
        while (fOk && dData);
        }
    return fOk;
    }
```

列表 4-24. 复制内存块中的数据

SpyMemoryTestAddress()用于测试数据的有效性，SpyMemoryReadBlock()针对要读取的每个字节都会调用 SpyMemoryTestAddress()。SpyMemoryTestAddress()在**列表 4-24** 的下半部分给出，该函数只是简单的调用 SpyMemoryPageEntry()，不过传入的第二个参数为 NULL。SpyMemoryPageEntry()在讨论 SPY_IO_PAGE_ENTRY 时已经介绍过（**列表 4-22**）。将其 PSPY_PAGE_ENTRY 指针参数设为 NULL，意味着调用者不关心指定线性地址对应的 page entry，因此，如果线性地址有效，函数将返回 TRUE。在 SpyMemoryPageEntry()的上下文中，仅当一个线性地址对应的数据页存在于物理内存中，或者位于页面文件中，该地址才是有效的。注意，这种行为与 ntoskrnl.exe 中的 API 函数 MmIsAddressValid()并不一致，当指定的页不存在于物理内存中时，MmIsAddressValid()总是返回 FALSE，即使这个有效的数据据页位于页面文件中也会如此。**列表 4-24** 中的另一个函数 SpyMemoryTestBlock()是 SpyMemoryTestAddress()的增强版。它可测试一个内存区域的有效性，它每次可测试指定块中的 4,096 个字节，直到测试完区域中的所有页为止。

```
#define SPY_MEMORY_DATA_N(_n) \
        struct _SPY_MEMORY_DATA_##_n \
            { \
            SPY_MEMORY_BLOCK smb; \
            WORD             awData [_n]; \
            }

typedef SPY_MEMORY_DATA_N (0)
        SPY_MEMORY_DATA, *PSPY_MEMORY_DATA, **PPSPY_MEMORY_DATA;

#define SPY_MEMORY_DATA_ sizeof (SPY_MEMORY_DATA)
#define SPY_MEMORY_DATA__(_n) (SPY_MEMORY_DATA_ + ((_n) * WORD_))

#define SPY_MEMORY_DATA_BYTE  0x00FF
#define SPY_MEMORY_DATA_VALID 0x0100

#define SPY_MEMORY_DATA_VALUE(_b,_v) \
```

```
        ((WORD) (((_b) & SPY_MEMORY_DATA_BYTE     ) | \
                 ((_v) ? SPY_MEMORY_DATA_VALID : 0)))
```

列表 4-25. SPY_MEMORY_DATA 的定义

将置换出去的页作为有效的地址范围有一个很重要的好处：当 SpyMemoryReadBlock()试图读取这些页中的第一个字节时，这些页就会被很快的再次调入内存中。稍后给出的内存 Dump 工具如果依赖 MmIsAddressValid()，有时就会拒绝显示指定地址范围中的数据（即使 5 分钟之前，它还可以显示这些数据），而这仅仅是因为这些页可能已被传送到了页面文件中。

4.2.12、IOCTL 函数 SPY_IO_MEMORY_BLOCK

SPY_IO_MEMORY_BLOCK 依赖于 SPY_IO_MEMORY_DATA,因为它也是从任意地址复制内存块到调用者的缓冲区中。主要的区别是：SPY_IO_MEMORY_DATA 试图复制所有可读取的字节，而对于 SPY_IO_MEMORY_BLOCK 来说，只要请求的范围中包含无效地址它就会失败，一个字节也不会复制。第 6 章中需要这个函数来将位于内核空间中的数据结构传递给用户模式下的程序。这一要求显然会大大限制这个函数，所以若一个结构体中包含无法读取的字节，就跳过它们，仅复制可读取的字节。

和 SPY_IO_MEMORY_DATA 类似，SPY_IO_MEMORY_BLOCK 期望输入一个 SPY_MEMORY_BLOCK 结构来指定要复制的内存块的基地址和大小。返回的数据，将是原始数据的 1:1 复制品。输出缓冲区必须足够容纳要复制的全部内容。否则，将会报告一个错误，并且不会返回任何数据。

4.2.13、IOCTL 函数 SPY_IO_HANDLE_INFO

和前面介绍的 SPY_IO_PHSICAL 类似，这个函数允许用户模式下的程序调用其他途经无法调用的内核模式 API。内核驱动程序可通过 ntoskrnl.exe 导出的 obReferenceObjectByHandle()来获取由句柄描述的对象的指针。而在 Win32 下没有对等的函数。不过，应用程序可以命令 Spy 设备执行这一函数，并返回对象的指针。**列表 4-26** 展示了由 SpyDispatcher()调用的 SpyOutputHandleInfo()函数。可通过 SpyInputHandle()获（定义于**列表 4-10**）取输入的句柄。

列表 4-26 顶部的 SPY_HANDLE_INFO 结构包含与句柄相关的对象体的指针，以及该句柄的属性，这两个都会由 ObReferenceObjectByHandle()返回。特别重要的一点是：如果 ObReferenceObjectByHandle()调用成功，就必须调用 ObDereferenceObject()来将对象的引用计数器恢复到先前的值。如果没有这样做，将会导致“对象引用漏洞”。

```
typedef struct _SPY_HANDLE_INFO
    {
    PVOID pObjectBody;
    DWORD dHandleAttributes;
    }
    SPY_HANDLE_INFO, *PSPY_HANDLE_INFO, **PPSPY_HANDLE_INFO;

#define SPY_HANDLE_INFO_ sizeof (SPY_HANDLE_INFO)

// -------------------------------------------------------------

NTSTATUS SpyOutputHandleInfo (HANDLE hObject,
                              PVOID  pOutput,
                              DWORD  dOutput,
                              PDWORD pdInfo)
    {
    SPY_HANDLE_INFO           shi;
    OBJECT_HANDLE_INFORMATION ohi;
    NTSTATUS                  ns = STATUS_INVALID_PARAMETER;

    if (hObject != NULL)
        {
        ns = ObReferenceObjectByHandle (hObject,
                                        STANDARD_RIGHTS_READ,
                                        NULL, KernelMode,
```

```
                                        &shi.pObjectBody, &ohi);
        }
    if (ns == STATUS_SUCCESS)
        {
        shi.dHandleAttributes = ohi.HandleAttributes;

        ns = SpyOutputBinary (&shi, SPY_HANDLE_INFO_,
                              pOutput, dOutput, pdInfo);

        ObDereferenceObject (shi.pObjectBody);
        }
    return ns;
    }
```

列表 4-26. 通过句柄引用一个对象

4.3、内存 Dump 工具----本书示例程序

现在你已经学完了复杂和让人困惑的内存 Spy 设备的 IOCTL 函数的代码，你可能想看这些函数运行起来是什么样子。因此，我创建了一个控制台模式的工具，名为：“SBS Windows 2000 Memory Spy”，该工具会加载 Spy 驱动程序，根据命令行出入的参数，它会调用多个 IOCTL 函数。该程序的可执行文件为：w2k_mem.exe，其源代码位于本书光盘的\src\w2k_mem 目录下。

4.3.1、命令行格式

你可以从光盘中运行内存 Spy 工具：d:\bin\w2k_mem.exe，这里 d:应该由你的 CD-ROM 盘符代替。如果无参数启动 w2k_mem.exe，将会列出冗长的命令信息，如**示列 4-1** 所示。W2k_mem.exe 基本的命令体系是：一个命令包含一个或多个数据请求，每个命令都至少提供一个线性基址，内存 Dump 将从该地址开始。如果你愿意还可指定内存块的大小，不过这是可选的，内存块的默认大小是 256。命令中的内存大小必须以“#”开始。可通过增加多个

选项来改变命令的默认行为。一个选项包括一个单字符的选项 ID 和一个“+”或“-”前缀。“+”或“-”表示允许或禁止该选项。默认情况下，所有选项都是允许的。

```
// w2k_mem.exe
// SBS Windows 2000 Memory Spy V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Usage: w2k_mem { { [+option|-option] [/<path>] } [#[[0]x]<size>] [[0]x]<base> }

<path> specifies a module to be loaded into memory.
Use the +x/-x switch to enable/disable its startup code.
If <size> is missing, the default size is 256 bytes.

Display address options (mutually exclusive):

   +z -z   zero-based display         on / OFF
   +r -r   physical RAM addresses     on / OFF

Display mode options (mutually exclusive):

   +w -w   WORD  data formatting      on / OFF
   +d -d   DWORD data formatting      on / OFF
   +q -q   QWORD data formatting      on / OFF

Addressing options (mutually exclusive):

   +t -t   TEB-relative addressing    on / OFF
   +f -f   FS-relative  addressing    on / OFF
   +u -u   user-mode   FS:[<base>]    on / OFF
```

```
   +k -k   kernel-mode FS:[<base>]    on / OFF
   +h -h   handle/object resolution   on / OFF
   +a -a   add bias  to  last base    on / OFF
   +s -s   sub bias from last base    on / OFF
   +p -p   pointer  from last block   on / OFF

System status options (cumulative):

   +o -o   display OS  information    on / OFF
   +c -c   display CPU information    on / OFF
   +g -g   display GDT information    on / OFF
   +i -i   display IDT information    on / OFF
   +b -b   display contiguous blocks  on / OFF

Other options (cumulative):

   +x -x   execute DLL startup code   on / OFF

Example: The following command displays the first 64
bytes of the current Process Environment Block (PEB)
in zero-based DWORD format, assuming that a pointer to
the PEB is located at offset 0x30 inside the current
Thread Environment Block (TEB):

   w2k_mem +t #0 0 +pzd #64 0x30

Note: Specifying #0 after +t causes the TEB to be
addressed without displaying its contents.
```

示列 4-1. 内存 Spy 工具的帮助信息

每个命令行所执行的数据请求不等同于选项，数据大小的说明，路径或任何其他的命令修饰成分。命令中的每个无格式的数字都被假定是一个线性地址，并且将从该地址开始，按16进制显示其内容。数字默认按10进制格式解释，如果有前缀“0x”或“x.”则按照16进制格式解释。

如果提供一些简单的示例，很容易掌握w2k_mem.exe采用的复杂命令行选项，下面就给出一些：

- w2k_mem 0x80400000 显示从线性地址0x80400000开始的256个字节，产生的内容可能会类似于示列4-2。顺便说一下，这是ntoskrnl.exe的DOS stub（注意开始的“MZ”ID）。
- w2k_mem #0x40 0x80400000 显示从线性地址0x80400000开始的64个字节，#0x40表示要显示的块大小为64
- w2k_mem +d #0x40 0x80400000 在前一命令的基础上，按照32位的DWORD Chunk来显示，这就是+d选项的作用。在同一个命令中，首先出现的+选项将会一直有效，除非使用相应的-选项或使用其互斥选项。如+d的互斥选项为：+w、+q。
- w2k_mem +wz #0x40 0x10000 +d -z 0x200000 包含两个数据请求。首先，线性地址范围：0x10000----0x1003F中的内容将按照16位WORD格式来显示，随后的0x20000---0x2003F按照32位DWORD格式显示（见**示列4-3**）。第一个请求中还包含一个+z选项，该选项将使“Address”列的数字从0开。在第二个请求中，通过-z选项，禁用了从0开始的显示模型。
- w2k_mem +rd #4096 0xC0300000 以DWORD格式显示起始于0xC0300000的系统页目录。+r选项表示在“Address”列中以物理内存地址代替线性地址。

现在，你应该基本上明白命令行格式是如何工作的了。在下一小节中，将详细讨论一些比较特别的选项和特性。它们中的大多数会改变对出现在它们之前的地址的解释方式。在默认情况下，指定的地址是一个线性基址，内存Dump将从那里开始。选项：+t、+f、+u、+k、+h、+a、+s和+p将以多种方式改变这种默认解释方式。

```
E:\>w2k_mem 0x80400000

// w2k_mem.exe
// SBS Windows 2000 Memory Spy V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Loading "SBS Windows 2000 Spy Device" (w2k_spy) ...
Driver: "E:\bin\w2k_spy.sys"
Opening "\\.\w2k_spy" ...

SBS Windows 2000 Spy Device V1.00 ready

80400000..804000FF: 256 valid bytes

Address  | 00 01 02 03-04 05 06 07 : 08 09 0A 0B-0C 0D 0E 0F | 0123456789ABCDEF
---------|-------------------------:-------------------------|----------------
80400000 | 4D 5A 90 00-03 00 00 00 : 04 00 00 00-FF FF 00 00 | MZ?.........??..
80400010 | B8 00 00 00-00 00 00 00 : 40 00 00 00-00 00 00 00 | ?.......@.......
80400020 | 00 00 00 00-00 00 00 00 : 00 00 00 00-00 00 00 00 | ................
80400030 | 00 00 00 00-00 00 00 00 : 00 00 00 00-C8 00 00 00 | ............?...
80400040 | 0E 1F BA 0E-00 B4 09 CD : 21 B8 01 4C-CD 21 54 68 | ..?.???L?Th
80400050 | 69 73 20 70-72 6F 67 72 : 61 6D 20 63-61 6E 6E 6F | is program canno
80400060 | 74 20 62 65-20 72 75 6E : 20 69 6E 20-44 4F 53 20 | t be run in DOS
80400070 | 6D 6F 64 65-2E 0D 0D 0A : 24 00 00 00-00 00 00 00 | mode....$.......
80400080 | 50 7A C4 CE-14 1B AA 9D : 14 1B AA 9D-14 1B AA 9D | Pz??..a?..a?..a?
80400090 | 14 1B AB 9D-53 1B AA 9D : 18 3B A4 9D-5B 1B AA 9D | ..琥S.猿.; [.猿
804000A0 | 42 13 AC 9D-15 1B AA 9D : 14 1B AA 9D-1A 19 AA 9D | B.??..a?..a?..a?
804000B0 | 4D 38 B9 9D-12 1B AA 9D : 52 69 63 68-14 1B AA 9D | M81?..a?Rich..a?
804000C0 | 00 00 00 00-00 00 00 00 : 50 45 00 00-4C 01 13 00 | ........PE..L...
804000D0 | 17 9B 4D 38-00 00 00 00 : 00 00 00 00-E0 00 0E 03 | .泼8........?..
804000E0 | 0B 01 05 0C-C0 2D 14 00 : 80 D6 04 00-00 00 00 00 | ....?..■?.....
804000F0 | 20 D1 00 00-C0 04 00 00 : 80 73 06 00-00 00 40 00 |  ?.?..■s....@.

       256 bytes requested
       256 bytes received

Closing the spy device ...
```

示列 4-2.　数据请求示列

```
E:\>w2k_mem +wz #0x40  0x10000 +d -z 0x20000

// w2k_mem.exe
// SBS Windows 2000 Memory Spy V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Loading "SBS Windows 2000 Spy Device" (w2k_spy) ...
Driver: "E:\bin\w2k_spy.sys"
Opening "\\.\w2k_spy" ...

SBS Windows 2000 Spy Device V1.00 ready

00010000..0001003F: 64 valid bytes

Address  | 0000 0002-0004 0006 : 0008 000A-000C 000E | 00 02 04 06 08 0A 0C 0E
---------|---------------------:---------------------|-----------------------
00000000 | 003D 003A-003A 003D : 003A 003A-005C 0000 | .= .: .: .= .: .: .\ ..
00000010 | 003D 0043-003A 003D : 0043 003A-005C 0062 | .= .C .: .= .C .: .\ .b
00000020 | 0069 006E-0000 003D : 0045 0078-0069 0074 | .i .n .. .= .E .x .i .t
00000030 | 0043 006F-0064 0065 : 003D 0030-0030 0030 | .C .o .d .e .= .0 .0 .0

00020000..0002003F: 64 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|-------------------
00020000 | 00001000 - 000006B4 : 00000001 - 00000000 | .... ...?.... ....
00020010 | 002F0001 - 00000000 : 00000003 - 00000018 | ./.. .... .... ....
00020020 | 0000000B - 0208000E : 00020290 - 00000054 | .... .... ...? ...T
00020030 | 01280126 - 00020498 : 00260024 - 000205C0 | .(.& ...?.&.$ ...?■
       128 bytes requested
       128 bytes received

Closing the spy device ...
```

示列 4-3.　以指定格式显示数据

4.3.2、与 TEB 相关的地址

进程中的每个线程都有其自己的线程环境块（Thread Environment Block，TEB），系统在此 TEB 中保存频繁使用的线程相关的数据。在用户模式下，当前线程的 TEB 位于独立的 4KB 段，可通过 CPU 的 FS 寄存器来访问该段。而在内核模式下，FS 却指向不同的段，下面将解释之。一个进程的所有 TEB 都以堆栈的方式，存放在从 0x7FFDE000 开始的线性内存中，每 4KB 为一个完整的 TEB，不过该内存区域是向下扩展的。这意味着，第二个线程的 TEB 的地址将是 0x7FFDC000，这和堆栈类似。在第七章，我们会详细讨论 TEB 的内容和进程环境块（Process Environment Block，PEB）的地址 0x7FFDF000（参见**列表 7-18** 和 **7-19**）。这里知道 TEB 的存在，而且知道其地址由 FS 寄存器给出就足够了。

如果在一个地址之前出现了+t 选项，w2k_mem.exe 将自动把 FS 段的基地址加到该地址上，**示列 4-4** 展示了 `w2k_mem +dt #0x38 0` 命令执行后的输出。这一次我省略了 w2k_mem.exe 输出的标题和状态信息。

```
E:\>w2k_mem +dt #0x38 0
[...]
7FFDE000..7FFDE037: 56 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
7FFDE000 | 0012F458 - 00130000 : 0012D000 - 00000000 | ..?X .... ..D. ....
7FFDE010 | 00001E00 - 00000000 : 7FFDE000 - 00000000 | .... .... .  . ....
7FFDE020 | 000007E4 - 00000758 : 00000000 - 00000000 | ...? ...X .... ....
7FFDE030 | 7FFDF000 - 00000000 :          -          | .ye. ....
[...]
```

示列 4-4.　显示第一个线程环境块（TEB）

4.3.3、与 FS 相关的地址

我前面已经提到过，在用户和内核模式下，FS 将指向不同的段。+t 选项将选择用户模式下 FS 所指向的地址，+f 选项则使用在内核模式下 FS 指向的地址。当然，Win32 应用程序没有办法获取该地址，因此，需要再次请求 Spy 设备。w2k_mem.xe 调用 IOCTL 函数 SPY_IO_CPU_INFO，来读去 CPU 的状态信息，这包括所有段寄存器在内核模式下的值。从此开始，所有的事情和+t 选项相同。

内核模式的 FS 指向另一个线程相关的结构，Windows 2000 内核回经常使用该结构，其名称为：内核的处理器控制区域（Kernel’s Processor Control Region，KPCR）。该结构在讨论 IOCTL 函数 SPY_IO_OS_INFO 时已经提及过，在第七章我们还会再次提到它（见**列表 7-16**）。再次强调，现在你只需要知道该结构存在于线性地址 0xFFDFF000 处即可，使用+f 选项就可访问它。在**示列 4-5** 中，我使用命令：`w2k_mem +df #0x54 0` 来演示，在实际情况下，使用+f 选项的结果。

```
E:\>w2k_mem +df #0x54 0
[...]
FFDFF000..FFDFF053: 84 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
FFDFF000 | F5E6FCF0 - F5E6FDF0 : F5E6C000 - 00000000 | 蹑   蹑   蹑? ....
FFDFF010 | 00000000 - 00000000 : 7FFDE000 - FFDFF000 | .... .... .  . ÿ咆.
FFDFF020 | FFDFF120 - 00000000 : 00000000 - 00000000 | ???  .... .... ....
FFDFF030 | FFFFA0E0 - 00000000 : 8003F400 - 8003F000 | ÿÿ犁 .... ■.? ■.?
FFDFF040 | 80042000 - 00010001 : 00000001 - 00000064 | ?. . .... .... ...d
FFDFF050 | 00000000 -          :          -          | ....
[...]
```

示列 4-5.　显示内核的处理器控制区域（KPCR）

4.3.4、FS:[Base]寻址方式

在察看 Windows 2000 内核代码时，你会经常遇到像 MOV　EAX,　FS:[18h]这样的指令。这些指令用于取出属于 TEB 或 KPCR 的成员的值，或者是属于其他包含在 FS 段中的结构体的成员的值。它们中的大多数都指向其他的内部结构。命令行选项+u 和+k 允许你；+u 表示使用用户模式下的 FS 段；+k 表示使用内核模式下的 FS 段。例如，命令：w2k_mem +du #0x1E8 0x30（见**示列 4-6**）将在用户模式下，从位于 FS:[30h]处的内存块中转储（dump）488 个字节。而命令：w2k_mem +dk #0x1C 0x20（见**示列 4-7**）将显示由内核模式下的 FS:[20h]指向的内存块的前 28 个字节，这实际上是指向 KPRCB 的一个指针。如果你不知道 PEB 或 KPRCB 是什么，不要着急，读完本书你就会一目了然了。

```
E:\>w2k_mem +du #0x1E8 0x30

// w2k_mem.exe
// SBS Windows 2000 Memory Spy V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Loading "SBS Windows 2000 Spy Device" (w2k_spy) ...
Driver: "E:\bin\w2k_spy.sys"
Opening "\\.\w2k_spy" ...

SBS Windows 2000 Spy Device V1.00 ready

7FFDF000..7FFDF1E7: 488 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
7FFDF000 | 00000000 - FFFFFFFF : 00400000 - 00131E90 | .... ???? .@.. ...?
7FFDF010 | 00020000 - 00000000 : 00130000 - 77FD0170 | .... .... .... wy.p
7FFDF020 | 77F82060 - 77F82091 : 00000001 - 77DF3900 | w? ` w? ? .... w?9.
7FFDF030 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF040 | 77FD01A8 - 00000001 : 00000000 - 7F6F0000 | w??.... .... .o..
7FFDF050 | 7F6F0000 - 7F6F0688 : 7FFA0000 - 7FFA0000 | .o.. .o.?.?. .?.
7FFDF060 | 7FFD1000 - 00000001 : 00000000 - 00000000 | .y.. .... .... ....
7FFDF070 | 079B8000 - FFFFE86D : 00100000 - 00002000 | .怦. ÿÿ鑝 .... .. .
7FFDF080 | 00010000 - 00001000 : 00000003 - 00000010 | .... .... .... ....
7FFDF090 | 77FD11E0 - 004E0000 : 00000000 - 00000014 | w??.N.. .... ....
7FFDF0A0 | 77FD0348 - 00000005 : 00000000 - 04000893 | wy.H .... .... ...?
7FFDF0B0 | 00000002 - 00000003 : 00000004 - 00000000 | .... .... .... ....
7FFDF0C0 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF0D0 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF0E0 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF0F0 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF100 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF110 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF120 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF130 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF140 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF150 | 77FD1270 - 00000000 : 00000000 - 00000000 | wy.p .... .... ....
7FFDF160 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF170 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF180 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF190 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF1A0 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF1B0 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF1C0 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF1D0 | 00000000 - 00000000 : 00000000 - 001E001C | .... .... .... ....
7FFDF1E0 | 7F6F06C2 - 00000000 :          -          | .o.? ....

       488 bytes requested
       488 bytes received

Closing the spy device ...
```

示列 4-6.　显示进程环境块（PEB）

```
E:\>w2k_mem +dk #0x1C 0x20
[...]
FFDFF120..FFDFF13B: 28 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
FFDFF120 | 00010001 - 81AABDA0 : 00000000 - 805448C0 | .... 佰絳 .... ■TH?
FFDFF130 | 00020000 - 00000001 : 08030106 -          | .... .... ....
[...]
```

示列 4-7. 显示内核的处理器控制区域（KPRCB）

4.3.5、句柄/对象 解析

假设你有一个对象句柄，而且你想要看看该句柄对应的对象在内存中是什么样子。如果你使用+h 选项，你就会发现完成这一任务太简单了，该选项将调用 Spy 设备的 SPY_IO_HANDLE_INFO 函数（见**列表 4-26**）来查找给定句柄的对象体（Object Body）。Windows 2000 对象世界是一个令人惊讶的主题，我将在第七章深入剖析它。所以，现在先把它丢掉一边去。

4.3.6、相对寻址

有时使用这种寻址方式可以很容易显示一系列内存块，这些内存块间隔相同大小的字节。这很有可能，比如，一个数组结构，像朵线程程序中的 TEB 堆栈。+a 和+s 选项通过将给定的地址解释为一个偏移量，来进行对寻址。这两个选项的区别是：+a（add bias）将产生一个正的偏移量，+s（subtract bias）则产生一个负的偏移量。**示列 4-8** 展示了命令：`w2k_mem +d #32 0xC0000000 +a 4096 4096` 的输出结果。它将取出三个连续 4KB 页中的前 32 个字节，起始地址为：0xC0000000，系统的页表就位于此处。注意，+a 选项接近命令的结尾处。它将使随后的“4096”将被解释为偏移量，该偏移量将被加到前面的基地址上。

```
E:\>w2k_mem +d #32 0xC0000000 +a 4096 4096
[...]
C0000000..C000001F: 32 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
C0000000 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
C0000010 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....

C0001000..C000101F: 32 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
C0001000 | 0D6BE025 - 1C910025 : 1F6D1025 - 0D467025 | .k? .?% .m.% .Fp%
C0001010 | 1AC92067 - 02D53225 : 00394067 - 09D83025 | .?g .?% .9@g .?%

C0002000..C000201F: 32 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
C0002000 |          -          :          -          |
C0002010 |          -          :          -          |

        96 bytes requested
        96 bytes received
[...]
```

示列 4-8.　页表样本

示列 4-8 还展示了如果传入一个无效的线性地址会发生什么。显然，第一对页表涉及的 4MB 地址范围：0x00000000----0x003F0000 和 0x00400000-----0x007F0000 是有效的。而第三对页表则是无效的。w2k_mem.exe 会通过显示一个空表来反映这一现实。程序知道那个地址范围是有效的，因为 Spy 设备的 SPY_IO_MEMORY_DATA 函数将此信息放入作为结果的 SPY_MEMORY_DATA 结构中（参见列表 4-25）。

4.3.7、间接寻址

我所钟爱的命令选项之一就是：+p，因为在我准备这本书的时候，它为我节省了很多打字的时间。该选项和+u 和+k 的工作方式类似，但不使用 FS 段，而是使用先前显示过的数据块。这是一个很棒的特性，如果你想向下寻找链表上的对象，例如，读取下一个成员的地址，随该命令一起，键入一个新的命令等等，通过简单在命令中加入+p 选项和一系列偏移量，就可以指定下一个对象的链接在前一个 16 进制 Dump 表中的位置。

在**示列 4-9** 中，我使用该选项来向下遍历当前活动进程的链表。首先，我告诉通过内核调试器获取系统内部变量 PsActiveProcessHead 的地址，该地址是一个 LIST_ENTRY 结构，用于标识进程链表的开始。LIST_ENTRY 结构中包含一个 Flink（向前指针）成员和一个 Blink（向后）成员。Flink 成员位于偏移量 0 处，Blink 成员位于偏移量 4 处（参见**列表 2-7**）。命令：`w2k_mem #8 +d 0x8046A180 +p 0 0 0 0` 首先转储 PsActiveProcessHead（这是一个 LIST_ENTRY 结构），然后从+p 选项出开始转为间接寻址。选项后的四个 0 是用来告诉 w2k_mem.exe 提取前一个数据块中偏移量为 0 的值，这正是 Flink 所在的位置。注意，**示列 4-9** 中的 Blink 成员在偏移量为 4 的位置上，它指向前一个 LSIT_ENTRY 之后，就像我们期望的那样。

译注：

对于 w2k_mem #8 +d 0x8046A180 +p 0 0 0 0 命令

0x8046A180 需要由你自己系统中的 PsActiveProcessHead 的地址来替代。

可通过内核调试器来查找 PsActiveProcessHead 的地址，我在这里使用的是 livekd，命令为：ln PsActiveProcessHead

如果命令中加入了足够的值为 0 的参数，16 进制转储最终会回到 PsActiveProcessHead，它用来标识进程链表的开始和结束。就像第二章里解释的那样，Windows 2000 维护的双向链表实际上是一个环；也就是说，链表中最后一个成员的 Flink 将指向链表中的第一个成员，而链表中第一个成员的 Blink 指向最后一个成员。

```
E:\>w2k_mem #8 +d 0x8046A180 +p 0 0 0 0
[---]
8046A180..8046A187: 8 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
8046A180 | 8149D900 - 840D2BE0 :           -         | ?Y. 仔.?

8149D900..8149D907: 8 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
8149D900 | 8131A4A0 - 8046A180 :           -         | 佮.  ■U炷

8131A4A0..8131A4A7: 8 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
8131A4A0 | 812FFDE0 - 8149D900 :           -         | 佁? ?Y.

812FFDE0..812FFDB7: 8 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
812FFDE0 | 812FA460 - 8131A4A0 :           -         | 佑&■ 佮.

812FA460..812FA467: 8 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
812FA460 | 812E30C0 - 812FFDE0 :           -         | 佢.?佁?
[---]
```

示列 4-9.　向下遍历活动进程链表

4.3.8、加载模块

有时你可能会想 dump 一个模块在内存中的映像,但是该模块还没有映射到 w2k_mem.exe 进程的线性地址空间。通过使用/<path>和+x 选项来显示的加载一个指定模块就可解决这一问题。每个前缀为斜线（“/”）的命令项将被解释为模块的全路径名，w2k_mem.exe 将尝试使用 Win32 API 函数 LoadLibraryEx()从该路径出加载模块。默认情况下，将使用加载选项 DON’T_RESOLVE_DLL_REFERENCES，这会使模块被加载到内存中，但不会被初始化。对于一个 DLL，这意味着它的 DllMian()入口点将不会被调用。同样，在该 DLL 的导入节中指定的依赖模块也都不会被加载。然而，如果你在路径参数之前，指定了+x 选项，那么模块将在加载后进行完整的初始化。注意，有些模块可能会拒绝在 w2k_mem.exe 进程的上下文环境中被初始化。例如，内核模式的设备驱动程序就不能在使用+x 选项的情况下，被加载到内存中。

加载和显示一个模块一般需要经过两个操作步骤，如**示列 4-10** 所示。首先，你应该加载模块，而不显示任何数据，以找出系统分配给该模块的基地址。幸运的是，只要在此期间，没有其他的模块加入到进程中，模块的加载地址就将是唯一的，因此，接下来尝试通过相同的基地址来加载模块。在**示列 4-10** 中，我加载了内核模式的设备驱动程序 nwrdr.sys，它是微软的 NetWare 重定向器。在我的系统里没有使用 IPX/SPX，因此，默认没有加载该驱动程序。

```
E:\>w2k_mem /e:\winnt\system32\drivers\nwrdr.sys
[...]
You didn't request any data!

LoadLibrary (e:\winnt\system32\drivers\nwrdr.sys) = 0x007A0000
[...]
E:\>w2k_mem 0x007A0000 /e:\winnt\system32\drivers\nwrdr.sys 0x007A0000
[...]
007A0000..007A00FF: 0 valid bytes

Address  | 00 01 02 03-04 05 06 07 : 08 09 0A 0B-0C 0D 0E 0F | 0123456789ABCDEF
---------|-------------------------:-------------------------|-----------------
007A0000 |            -            :            -            |
007A0010 |            -            :            -            |
007A0020 |            -            :            -            |
007A0030 |            -            :            -            |
007A0040 |            -            :            -            |
007A0050 |            -            :            -            |
007A0060 |            -            :            -            |
007A0070 |            -            :            -            |
007A0080 |            -            :            -            |
007A0090 |            -            :            -            |
007A00A0 |            -            :            -            |
007A00B0 |            -            :            -            |
007A00C0 |            -            :            -            |
007A00D0 |            -            :            -            |
007A00E0 |            -            :            -            |
007A00F0 |            -            :            -            |

LoadLibrary (e:\winnt\system32\drivers\nwrdr.sys) = 0x007A0000

007A0000..007A00FF: 256 valid bytes

Address  | 00 01 02 03-04 05 06 07 : 08 09 0A 0B-0C 0D 0E 0F | 0123456789ABCDEF
---------|-------------------------:-------------------------|-----------------
007A0000 | 4D 5A 90 00-03 00 00 00 : 04 00 00 00-FF FF 00 00 | MZ?.........??..
007A0010 | B8 00 00 00-00 00 00 00 : 40 00 00 00-00 00 00 00 | ?.......@.......
007A0020 | 00 00 00 00-00 00 00 00 : 00 00 00 00-00 00 00 00 | ................
007A0030 | 00 00 00 00-00 00 00 00 : 00 00 00 00-D8 00 00 00 | ............?...
007A0040 | 0E 1F BA 0E-00 B4 09 CD : 21 B8 01 4C-CD 21 54 68 | ..?.???L?Th
007A0050 | 69 73 20 70-72 6F 67 72 : 61 6D 20 63-61 6E 6E 6F | is program canno
007A0060 | 74 20 62 65-20 72 75 6E : 20 69 6E 20-44 4F 53 20 | t be run in DOS
007A0070 | 6D 6F 64 65-2E 0D 0D 0A : 24 00 00 00-00 00 00 00 | mode....$.......
007A0080 | 39 F0 4B C1-7D 91 25 92 : 7D 91 25 92-7D 91 25 92 | 9餃羹?抁?抁?
007A0090 | 71 B1 2B 92-7F 91 25 92 : 24 B2 36 92-7A 91 25 92 | q?????抇?
007A00A0 | 7D 91 24 92-E0 91 25 92 : 2B 99 23 92-7C 91 25 92 | }?揣???抾?
007A00B0 | 7D 91 25 92-4C 91 25 92 : 52 69 63 68-7D 91 25 92 | }?%?L?%?Rich}?%?
007A00C0 | 00 00 00 00-00 00 00 00 : 00 00 00 00-00 00 00 00 | ................
007A00D0 | 00 00 00 00-00 00 00 00 : 50 45 00 00-4C 01 09 00 | ........PE..L...
007A00E0 | 07 5D 56 3E-00 00 00 00 : 00 00 00 00-E0 00 0E 03 | .]V>........?..
007A00F0 | 0B 01 05 0C-E0 35 02 00 : 00 3B 00 00-00 00 00 00 | ....?...;......
[...]
```

列表 4-10.　加载和显示一个模块映像（Module Image）

4.3.9、请求式分页动作

在讨论 Spy 设备的 SPY_IO_MEMORY_DATA 函数时，我提到过该函数可以读取已被置换到页面文件中的内存页。要证明这一点，首先，必须让系统处于低内存状态，以强迫它将不马上使用的数据置换到页面文件中。我喜欢采用的方法如下：

1. 使用 PrintKey，将 Windows 2000 的桌面复制到剪切板中。
2. 将该图片粘贴到一个图形处理程序中。
3. 将该图片的尺寸放到最大。

现在，执行命令：`w2k_mem +d #16 0xC02800000 0xA0000000 0xA0001000 0xA0002000 0xC0280000`，察看它在屏幕上的输出。你可能会惊讶。在触及某些 PTE 所引用的页之前，它会获取这些 PTE 的快照。在地址 0xC0280000 处发现的四个 PTE 与地址范围：0xA0000000---0xA0003FFF 相关，这是内核模块 win32k.sys 的一部分。如**示列 4-11** 所示，该地址范围已经被置换出去了，因为在地址 0xC0280000 的四个 DWORD 都是偶数，这意味着它们的最低位（即 PTE 的 P 位）为零，这表示没有存在于物理内存中的页。接下来的三块 16 进制 Dump 信息属于 0xA0000000、0xA0001000、0xA0002000，w2k_mem 可以毫无问题的访问这些页（系统会根据请求将它们再次换入内存）。

```
E:\>w2k_mem +d #16 0xC02800000 0xA0000000 0xA0001000 0xA0002000 0xC0280000
[...]
C0280000..C028000F: 16 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
C0280000 | 056A14E0 - 056A14E2 : 056A14E4 - 056A1AE6 | .??# .?A! .-Q! .-a!

A0000000..A000000F: 16 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
A0000000 | 00905A4D - 00000003 : 00000004 - 0000FFFF | .?ZM .... .... ..??

A0001000..A000100F: 16 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
A0001000 | 8D0476FF - E850FC45 : 000062C7 - 8BFC458B | ?vij 鑀斢 ..b?嫩E?■
A0002000..A000200F: 16 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
A0002000 | A1645CEB - 00000124 : 8B44408B - 00021488 | ?d\? ...$ ?D@? ...?

C0280000..C028000F: 16 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
C0280000 | 0556B123 - 028C2121 : 05AD1121 - 056A14E6 | .?vij .D@? .@jjj .b?
[...]
```

示列 4-11　观察 PTE 的状态变化

在开始下一节之前，请再次研究一下**示列 4-11** 中的第一栏。位于地址 0xC0280000 的四个 PTE 看上去都很像。但事实上，它们仅有最低的三个位不同。如果你检查所有位于页面文件中的 PNPE，你会发现它们的第 10 位都为 1。这就是为什么我在**列表 4-3** 中，将该位的名字取为 PageFile。如果该位为 1，除 P 位外的所有位都将用来表示该页在页面文件中的位置。

4.3.10、更多的命令选项

示列 4-1 给出的某些命令选项还没有解释过。例如，系统状态选项：+o、+c、+g、+i 和+b，我会在本章的最后一节介绍它们，在那儿我们将发现几个 Windows 2000 内存系统的秘密。

4.4、Spy 设备的接口

现在你已经知道如何使用 w2k_mem 了，该是介绍它是如何工作的了。现在来看看这个程序是如何与 w2k_spy.sys 中的 Spy 设备通讯的。

4.4.1、回顾-----设备 I/O 控制（Device I/O Control）

IOCTL 通讯的内核模式端已经由**列表 4-6** 和**列表 4-7** 给出了。Spy 设备只是简单的等待 IRP 并处理其中的某些 IRP，尤其是标识为 IPR_MJ_DEVICE_CONTROL，其中的一些请求在用户模式下是被禁止的。调用 Win32 API 函数 DeviceIoControl()，**列表 4-27** 给出了该函数的原型。可能你已经熟悉了 dwIocontrolCode、lpInBuffer、nInBufferSize、lpOutBuffer、nOutBufferSize 和 lpBytesReturned 参数。事实上，它们一一对应于：SpyDispatcher()的 dCode、pInput、dInput、pOutput、dOutput 和 pdInfo 参数，SpyDispatcher 定义于**列表 4-7**。剩下的参数很快就会解释。hDevice 是 Spy 设备的句柄，lpOverlapped（可选的）指向一个 OVERLAPPED 结构，异步 IOCTL 需要该结构。我们不需要发送异步请求，所以该参数总是 NULL。

列表 4-28 列出了所有执行基本 IOCTL 操作的外包函数。最基本的一个是：IoControl()，该函数调用 DeviceControl()并测试返回的输出数据的大小。因为 w2k_mem.exe 精确的提供了输出缓冲区的大小，所以，输出的字节数应该总是等于缓冲区的大小。ReadBinary()是 IoControl()的简单版本，它不需要输入数据。ReadCpuInfo()、ReadSegment()和

ReadPhysical()专用于 Spy 函数 SPY_IO_CPU_INFO、SPY_IO_SEGEMNT 和 SPY_IO_PHYSICAL，因为它们会经常被用到。将它们封装为 C 函数，可读性会更好些。

```
BOOL WINAPI DeviceIoControl( HANDLE        hDevice,
                             DWORD         dwIoControlCode,
                             PVOID         lpInBuffer,
                             DWORD         nInBufferSize,
                             PVOID         lpOutBuffer,
                             DWORD         nOutBufferSize,
                             PDWORD        lpBytesReturned,
                             POVERLAPPED   lpOverlapped);
```

列表 4-27. DeviceIoControl 函数的原型

```
BOOL WINAPI IoControl (HANDLE hDevice,
                       DWORD  dCode,
                       PVOID  pInput,
                       DWORD  dInput,
                       PVOID  pOutput,
                       DWORD  dOutput)
    {
    DWORD dData = 0;

    return DeviceIoControl (hDevice, dCode,
                            pInput,  dInput,
                            pOutput, dOutput,
                            &dData,  NULL)
           &&
           (dData == dOutput);
    }
// ------------------------------------------------------------
```

```
BOOL WINAPI ReadBinary (HANDLE hDevice,
                        DWORD  dCode,
                        PVOID  pOutput,
                        DWORD  dOutput)
    {
    return IoControl (hDevice, dCode, NULL, 0, pOutput, dOutput);
    }

// -------------------------------------------------------------

BOOL WINAPI ReadCpuInfo (HANDLE        hDevice,
                         PSPY_CPU_INFO psci)
    {
    return IoControl (hDevice, SPY_IO_CPU_INFO,
                      NULL,    0,
                      psci,    SPY_CPU_INFO_);
    }

// -------------------------------------------------------------

BOOL WINAPI ReadSegment (HANDLE       hDevice,
                         DWORD        dSelector,
                         PSPY_SEGMENT pss)
    {
    return IoControl (hDevice,    SPY_IO_SEGMENT,
                      &dSelector, DWORD_,
                      pss,        SPY_SEGMENT_);
    }
```

```
// -------------------------------------------------------------


BOOL WINAPI ReadPhysical (HANDLE            hDevice,
                          PVOID             pLinear,
                          PPHYSICAL_ADDRESS ppa)
    {
    return IoControl (hDevice,  SPY_IO_PHYSICAL,
                      &pLinear, PVOID_,
                      ppa,      PHYSICAL_ADDRESS_)
           &&
           (ppa->LowPart || ppa->HighPart);
    }
```

列表 4-28　几个 IOCTL 的外包函数

到目前为止，本节列出的所有函数都需要 Spy 设备的一个句柄。现在，我将介绍如何获取该句柄。这实际上是一个非常简单的 Win32 操作，和打开文件类似。**列表 4-29** 展示了 w2k_mem.exe 的命令处理例程的实现细节。该代码使用 API 函数 w2kFilePath()、w2kServiceLoad()和 w2kServiceUnload()，这几个函数由 w2k_lib.dll 导出。如果你已经读过第三章中关于 Windows 2000 服务控制管理器的介绍，你应该通过**列表 3-8** 已了解了 w2kServiceLoad()和 w2kServiceUnload()。这些强大的函数可随时加载或卸载内核模式的设备驱动，并且能处理一些良性的错误，如，妥善的处理加载一个已经载入内存的驱动程序。w2kFilePath()是一个帮助函数。w2k_mem.exe 调用它来获取 Spy 驱动程序的完整路径。

```
WORD awSpyFile      [] = SW(DRV_FILENAME);
WORD awSpyDevice    [] = SW(DRV_MODULE);
WORD awSpyDisplay   [] = SW(DRV_NAME);
WORD awSpyPath      [] = SW(DRV_PATH);


// -------------------------------------------------------------


void WINAPI Execute (PPWORD ppwArguments,
```

```
                DWORD  dArguments)
{
SPY_VERSION_INFO svi;
DWORD            dOptions, dRequest, dReceive;
WORD             awPath [MAX_PATH] = L"?";
SC_HANDLE        hControl          = NULL;
HANDLE           hDevice           = INVALID_HANDLE_VALUE;

_printf (L"\r\nLoading \"%s\" (%s) ...\r\n",
         awSpyDisplay, awSpyDevice);

if (w2kFilePath (NULL, awSpyFile, awPath, MAX_PATH))
    {
    _printf (L"Driver: \"%s\"\r\n",
             awPath);

    hControl = w2kServiceLoad (awSpyDevice, awSpyDisplay,
                               awPath, TRUE);
    }
if (hControl != NULL)
    {
    _printf (L"Opening \"%s\" ...\r\n",
             awSpyPath);

    hDevice = CreateFile (awSpyPath, GENERIC_READ,
                          FILE_SHARE_READ | FILE_SHARE_WRITE,
                          NULL, OPEN_EXISTING,
                          FILE_ATTRIBUTE_NORMAL, NULL);
    }
```

```
else
    {
    _printf (L"Unable to load the spy device driver.\r\n");
    }
if (hDevice != INVALID_HANDLE_VALUE)
    {
    if (ReadBinary (hDevice, SPY_IO_VERSION_INFO,
                    &svi, SPY_VERSION_INFO_))
        {
        _printf (L"\r\n%s V%lu.%02lu ready\r\n",
                 svi.awName,
                 svi.dVersion / 100, svi.dVersion % 100);
        }
    dOptions = COMMAND_OPTION_NONE;
    dRequest = CommandParse (hDevice, ppwArguments, dArguments,
                             TRUE, &dOptions);

    dOptions = COMMAND_OPTION_NONE;
    dReceive = CommandParse (hDevice, ppwArguments, dArguments,
                             FALSE, &dOptions);
    if (dRequest)
        {
        _printf (awSummary,
                 dRequest, (dRequest == 1 ? awByte : awBytes),
                 dReceive, (dReceive == 1 ? awByte : awBytes));
        }
    _printf (L"\r\nClosing the spy device ...\r\n");
    CloseHandle (hDevice);
    }
```

```
    else
        {
        _printf (L"Unable to open the spy device.\r\n");
        }
    if ((hControl != NULL) && gfSpyUnload)
        {
        _printf (L"Unloading the spy device ...\r\n");
        w2kServiceUnload (awSpyDevice, hControl);
        }
    return;
    }
```

列表 4-29.　控制 Spy 设备

请注意**列表 4-29** 顶部给出的四个全局字符串的定义。常量 DRV_FILENAME、DRV_MODULE、DRV_NAME 和 DRV_PATH 来自 Spy 驱动的头文件 w2k_spy.h。**表 4-4** 列出了它们的当前值。你不会在 w2k_mem.exe 的源代码中发现设备相关的定义，w2k_spy.h 提供了客户端程序所需的一切。这非常重要：如果以后改变了任何设备相关的定义，就不需要更新任何程序文件了。只需要以新的头文件编译、链接程序即可。

列表 4-29 顶部调用的w2kFilePath()可以保证由全局变量awSpyFile（见**表 4-4**）指定的w2k_spy.sys总是从w2k_mem.exe所在目录中加载。接下来，**列表 4-29** 中的代码将全局字符串awSpyDevice和awSpyDisplay（）传递给w2kServiceLoad()，以加载Spy设备的驱动。如果驱动没有被加载，这些字符串将被保存在驱动的属性列表中，可以由其他程序取出；否则，将保留当前的属性设置。尽管**列表 4-29** 中的w2kServiceLoad()调用可返回一个句柄，但这并不是一个可用于任何IOCTL函数的句柄。要获取Spy设备的句柄，必须使用Win32 的多用途函数CreateFile()。该函数可打开或创建Windows 2000 中几乎所有可被打开和创建的东西。如果提供了内核设备的符号链接名，形如\\.\<SymbolicLink>给CreateFile()的lpFileName参数，那么该函数就可打开这个内核设备。Spy设备的符号链接名是：w2k_spy，因此，CreateFile()的第一个参数必须是\\.\w2k_spy，这正是**表 4-4** 中的awSpyPath的值。

表 4-4.　设备相关的字符串定义

w2k_spy 常量	w2k_mem 变量	值

DRV_FILENAME	awSpyFile	w2k_spy.sys
DRV_MODULE	awSpyDevice	w2k_spy
DRV_NAME	awSpyDisplay	SBS Windows 2000 Spy Device
DRV_PATH	awSpyPath	\\.\ w2k_spy

如果 CreateFile()成功，它将返回一个设备的句柄，该句柄可传递给 DeviceIoControl()。**列表 4-29** 中的 Execute()函数使用该句柄来查询 Spy 设备的版本信息，如果 IOCTL 调用成功，该信息将会在屏幕上显示出来。接下来，CommandParser()函数将被调用两次,第一次调用只是简单的检查命令行中是否有无效的参数,并显示任何可能的错误。第二次调用则执行所有的命令。我不想讨论该函数的细节。**列表 4-29** 中的剩余代码是为了进行清理工作，如关闭句柄和卸载 Spy 驱动（该功能是可选的）。w2k_mem.exe 的源代码中还有一些有趣的代码片断，但我不在这里讨论它们了。请参考本书光盘的\src\w2k_mem 目录下的 w2k_mem.c 和 w2k_mem.h。

现在唯一需要注意的就是 gfSpyUnload 标志，该标志决定是否卸载 Spy 驱动。我已经将这个全局标志设为了 FALSE,因此不会自动卸载该驱动。这提高 w2k_mem.exe 或 w2k_spy.sys 的任何客户端的性能，因为加载一个驱动需要花费一定的时间。只有第一个客户端会产生加载开销。这种设置还可避免多个客户端间的竞争，如，一个客户试图卸载该驱动而此时另一个还在使用这个驱动。当然，Windows 2000 不会卸载一个驱动，除非该驱动的所有句柄都被关闭了，但系统会将驱动置于 STOP_PENDING 状态，这样新的客户端将无法访问此设备。不过，如果你不在一个多客户端的环境下运行 w2k_spy.sys，而且你需要经常更新设备的驱动程序，你就应该将 gfSpyUnload 标志设为 TRUE。

4.5、深入 Windows 2000 内存

引入用户模式和内核模式的独立 4GB 地址空间被再次划分为多个更小的块。正如你可能猜到的，它们中的大多数都包含未文档化的结构，而且服务于未文档化的地目的。其中某些东西对于任何开发系统诊断或调试软件的人来说都是真正的金矿。

4.5.1、基本的操作系统信息

如果你注意过**示列 4-1** 下半部分的帮助信息，你会发该节的标题是：“系统状态选项”。现在试试名为“显示操作系统信息”的选项：+o。**示列 4-12** 给出了在我的机器上使用该选

项的输出结果。这里显示的信息都是 SPY_OS_INFO 结构的内容，该结构定义与**列表 4-13**，由 Spy 设备函数 SpyOutputOsInfo()实际创建该结构，此函数也包含在**列表 4-13** 中。在**示列 4-12** 中，你可以看到位于 4GB 地址空间中的进程的一些典型地址。例如，有效的用户地址范围是：0x00010000 ---- 0x7FFFFFFF。你可能阅读过其他有关 Windows NT 或 2000 的程序设计书籍，用户模式的第一个和最后一个 64KB 线性内存区域是“不能访问区域”，访问这一区域将引发一个错误（参见第五章，Solomon 1998），W2k_mem.exe 输出证明了这一点。

```
E:\>w2k_mem +o
[...]
OS information:
---------------

Memory page size         : 4096 bytes
Memory page shift        : 12 bits
Memory PTI  shift        : 12 bits
Memory PDI  shift        : 22 bits
Memory page mask         : 0xFFFFF000
Memory PTI  mask         : 0x003FF000
Memory PDI  mask         : 0xFFC00000
Memory PTE  array        : 0xC0000000
Memory PDE  array        : 0xC0300000

Lowest user address      : 0x00010000
Thread environment block : 0x7FFDE000
Highest user address     : 0x7FFEFFFF
User probe address       : 0x7FFF0000
System range start       : 0x80000000
Lowest system address    : 0xC0800000
Shared user data         : 0xFFDF0000
Processor control region : 0xFFDFF000
Processor control block  : 0xFFDFF120

Global flag              : 0x00000000
i386 machine type        : 0
Number of processors     : 1
Product type             : Windows NT Workstation (1)
Version & Build number   : 5.00.2195
System root              : "E:\WINNT"
[...]
```

示列 4-12. 显示操作系统信息

示列 4-12 中的最后三行包含的信息非常有趣，它们都是有关系统的。这些信息大部分都取自位于地址 0xFFDF0000 处的 SharedUserData 区域中。系统在该处维护一个名为 KUSER_SHARED_DATA 的结构，该结构定义于 DDK 头文件 ntddk.h。

4.5.2、Windows 2000 的分段和描述符

w2k_mem.exe 的另一个很棒的选项是+e，该选项将显示和说明处理器的段寄存器和描述表的内容。**示列 4-13** 给出了其典型输出。CS、DS 和 ES 段寄存器的内容非常清晰的证明了 Windows 2000 为每个进程提供了平坦的 4GB 地址空间：起始于 0x00000000，终止于 0xFFFFFFFF。**示列 4-13** 中最右边的标志符用来表示段的类型，该段的类型由它的描述符的 Type 成员给出。代码和数据段的 Type 属性可分别符号化为“cra”和“ewa”。省略号“-”意味着相应的属性没有设置。一个任务状态段（Task State Segment，TSS）仅能有“a”（可用）和“b”（忙）两种属性。**表 4-5** 给出了所有可用的属性。**示列 4-13** 展示了 Windows 2000 的 CS 段的不一致性，CS 段允许执行和读取，而 DS、ES、FS 和 SS 段的属性则是可扩展和读/写访问。另一个不明显但十分重要的细节是 CS、FS 和 SS 段的 DPL 在用户模式和内核模式并不相同。DPL 是描述符特权级别（Descriptor Privilege Level）。对于代码段（CS），仅当调用者位于其 DPL 指定的特权级时才能调用该段中的代码（参考 Intel 1999c, pp. 4-8f）。在用户模式，CS 段的 DPL 为 3；在内核模式，其 DPL 为 0。对于数据段（DS），其 DPL 是最低的特权级，在用户模式下，所有特权级都可访问它，而在内核模式下，仅允许特权 0 访问。

```
E:\>w2k_mem +c
[...]
CPU information:
----------------

User mode segments:

CS  : Selector = 001B, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = CODE -ra
DS  : Selector = 0023, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = DATA -wa
ES  : Selector = 0023, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = DATA -wa
FS  : Selector = 0038, Base = 7FFDE000, Limit = 00000FFF, DPL3, Type = DATA -wa
SS  : Selector = 0023, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = DATA -wa
TSS : Selector = 0028, Base = 80042000, Limit = 000020AB, DPL0, Type = TSS32 b

Kernel mode segments:

CS  : Selector = 0008, Base = 00000000, Limit = FFFFFFFF, DPL0, Type = CODE -ra
DS  : Selector = 0023, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = DATA -wa
ES  : Selector = 0023, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = DATA -wa
FS  : Selector = 0030, Base = FFDFF000, Limit = 00001FFF, DPL0, Type = DATA -wa
SS  : Selector = 0010, Base = 00000000, Limit = FFFFFFFF, DPL0, Type = DATA -wa
TSS : Selector = 0028, Base = 80042000, Limit = 000020AB, DPL0, Type = TSS32 b

IDT : Limit    = 07FF, Base = 8003F400
GDT : Limit    = 03FF, Base = 8003F000
LDT : Selector = 0000

CR0 : Contents = 8001003B
CR2 : Contents = 7FFD2012
CR3 : Contents = 1816B000
[...]
```

示列 4-13. 显示 CPU 信息

DT 和 GDT 寄存器的内容显示了 GDT 的范围是：0x8003F000 --- 0x8003F3FF，紧随其后的就是 IDT，其地址范围是：0x8003F400 --- 0x8003FBFF。由于每个描述符占用 64 位，故 GDT 和 IDT 分别包含 128 和 256 个项。注意，GDT 可容纳 8,192 个项，但 Windows 2000 仅使用了其中的一小部分。

表 4-5 代码和数据段的 Type 属性

段	属 性	描 述
CODE	c	使段一致（低特权的代码可能进入）
CODE	r	允许读访问（和仅执行访问相斥）
CODE	a	段可以访问
DATA	e	向下扩展段（堆栈段的典型属性）
DATA	w	允许写访问（和仅读取访问相斥）
DATA	a	段可以访问
TSS32	a	任务状态段可用
TSS32	b	任务状态段繁忙

W2k_mem.exe 还提供了两个很有特色的选项----+g 和+i，这两个选项可显示 GDT 和 IDT 的更多细节。**示列 4-14** 示范了+g 选项的输出。它很类似于**示列 4-13** 中的“kernel-model segment:”一节，但列出了在内核模式下所有可用的段选择子（selector），而不仅仅是存储在段寄存器中的那些。W2k_mem.exe 通过遍历整个 GDT 来获取所有的段选择子，可通过 IOCTL 函数 SPY_IO_SEGMENT 来指示 Spy 设备查询段信息。仅显示有效的选择子。比较**示列 4-13** 和 **4-14** 中的 GDT 选择子将十分有趣，GDT 的选择子定义于 ntddk.h 中，汇总在**表 4-6**。显然，它们与 w2k_mem.exe 的输出是一致的。

```
E:\>w2k_mem +g
[---]
GDT information:
----------------

001 : Selector = 0008, Base = 00000000, Limit = FFFFFFFF, DPL0, Type = CODE -ra
002 : Selector = 0010, Base = 00000000, Limit = FFFFFFFF, DPL0, Type = DATA -wa
003 : Selector = 0018, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = CODE -ra
004 : Selector = 0020, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = DATA -wa
005 : Selector = 0028, Base = 80042000, Limit = 000020AB, DPL0, Type = TSS32 b
006 : Selector = 0030, Base = FFDFF000, Limit = 00001FFF, DPL0, Type = DATA -wa
007 : Selector = 0038, Base = 7FFDE000, Limit = 00000FFF, DPL3, Type = DATA -wa
008 : Selector = 0040, Base = 00000400, Limit = 0000FFFF, DPL3, Type = DATA -w-
00A : Selector = 0050, Base = 80548B10, Limit = 00000068, DPL0, Type = TSS32 a
00B : Selector = 0058, Base = 80548B78, Limit = 00000068, DPL0, Type = TSS32 a
00C : Selector = 0060, Base = 80022F30, Limit = 0000FFFF, DPL0, Type = DATA -wa
00D : Selector = 0068, Base = 000B8000, Limit = 00003FFF, DPL0, Type = DATA -w-
00E : Selector = 0070, Base = FFFF7000, Limit = 000003FF, DPL0, Type = DATA -w-
00F : Selector = 0078, Base = 80400000, Limit = 0000FFFF, DPL0, Type = CODE -r-
010 : Selector = 0080, Base = 80400000, Limit = 0000FFFF, DPL0, Type = DATA -w-
011 : Selector = 0088, Base = 00000000, Limit = 00000000, DPL0, Type = DATA -w-
014 : Selector = 00A0, Base = 820612E8, Limit = 00000068, DPL0, Type = TSS32 a
01C : Selector = 00E0, Base = EB050000, Limit = 0000FFFF, DPL0, Type = CODE cra
01D : Selector = 00E8, Base = 00000000, Limit = 0000FFFF, DPL0, Type = DATA -w-
01E : Selector = 00F0, Base = 80503A1C, Limit = 000003B7, DPL0, Type = CODE ---
01F : Selector = 00F8, Base = 00000000, Limit = 0000FFFF, DPL0, Type = DATA -w-
020 : Selector = 0100, Base = EB060000, Limit = 0000FFFF, DPL0, Type = DATA -wa
021 : Selector = 0108, Base = EB060000, Limit = 0000FFFF, DPL0, Type = DATA -wa
022 : Selector = 0110, Base = EB060000, Limit = 0000FFFF, DPL0, Type = DATA -wa
[---]
```

示列 4-14. 显示 GDT 描述符

表 4-6. 定义于 ntddk.h 中的 GDT 选择子（selector）

符　　号	值	注　　释
KGDT_NULL	0x0000	空的段选择子（无效）
KGDT_R0_CODE	0x0008	内核模式的 CS 寄存器
KGDT_R0_DATA	0x0010	内核模式的 SS 寄存器
KGDT_R3_CODE	0x0018	用户模式的 CS 寄存器
KGDT_R3_DATA	0x0020	用户模式的 DS、ES 和 SS 寄存器，内核模式的 DS 和 ES 寄存器
KGDT_TSS	0x0028	位于用户和内核的任务状态段
KGDT_R0_PCR	0x0030	内核模式的 FS 寄存器（处理器控制区域）
KGDT_R3_TEB	0x0038	用户模式的 FS 寄存器（线程环境块）
KGDT_VDM_TILE	0x0040	基地址 0x00000400，限制 0x0000FFFF（DOS 虚拟机）
KGDT_LDT	0x0048	本地描述符表
KGDT_DF_TSS	0x0050	Ntoskrnl.exe 变量 KiDoubleFaultTSS
KGDT_NMI_TSS	0x0058	Ntoskrnl.exe 变量 KiNMITSS

示列 4-14 中的选择子（selector）没有在**表 4-6** 中列出，其中的某些选择子可以通过查找熟悉的基地址或其内存内容来确认它们。使用内核调试器可查找其中某些选择子的基地址对应的符号。**表 4-7** 给出了我已经确认的选择子。

W2k_mem.exe 的+i 选项可转储 IDT 中的门描述符（Gate Descriptor）。**示列 4-15** 给出了 IDT 的门描述符的部分内容，Intel 仅定义了 IDT 中的前 20 个门描述符（Intel 1999c, pp. 5-6）。IDT 中的中断 0x14 到 0x1F 由 Intel 保留；剩余的 0x20 到 0xFF 由操作系统使用。

在**表 4-8** 中，我给出了所有可确认的特殊的中断、陷阱和任务门。大多数用户自定义的中断都指向哑元例程---KiUnexpectedinterruptnNNN()，在前面我们已经解释过它。对于某些中断处理例程的地址，内核调试器也无法解析其地址对应的符号。

表 4-7. 更多的 GDT 选择子（selector）

值	基地址	描　　述
0x0078	0x80400000	Ntoskrnl.exe 的代码段
0x0080	0x80400000	Ntoskrnl.exe 的数据段
0x00A0	0x814985A8	TSS（EIP 成员指向 HalpMcaExceptionHandlerWrapper）

0x00E0	0xF0430000	ROM BIOS 代码段
0x00F0	0x8042DCE8	Ntoskrnl.exe 函数 KiI386CallAbios
0x0100	0xF0440000	ROM BIOS 数据段
0x0108	0xF0440000	ROM BIOS 数据段
0x0110	0xF0440000	ROM BIOS 数据段

```
E:\>w2k_mem +i
[...]
IDT information:
----------------

00 : Pointer = 0008:8053CB78, Base = 00000000, Limit = FFFFFFFF, Type = INT32
01 : Pointer = 0008:8053CCC4, Base = 00000000, Limit = FFFFFFFF, Type = INT32
02 : TSS     = 0058,          Base = 80548B78, Limit = 00000068, Type = TASK
03 : Pointer = 0008:8053CFB4, Base = 00000000, Limit = FFFFFFFF, Type = INT32
04 : Pointer = 0008:8053D114, Base = 00000000, Limit = FFFFFFFF, Type = INT32
05 : Pointer = 0008:8053D254, Base = 00000000, Limit = FFFFFFFF, Type = INT32
06 : Pointer = 0008:8053D3B0, Base = 00000000, Limit = FFFFFFFF, Type = INT32
07 : Pointer = 0008:8053DA04, Base = 00000000, Limit = FFFFFFFF, Type = INT32
08 : TSS     = 0050,          Base = 80548B10, Limit = 00000068, Type = TASK
09 : Pointer = 0008:8053DDF4, Base = 00000000, Limit = FFFFFFFF, Type = INT32
0A : Pointer = 0008:8053DEF8, Base = 00000000, Limit = FFFFFFFF, Type = INT32
0B : Pointer = 0008:8053E020, Base = 00000000, Limit = FFFFFFFF, Type = INT32
0C : Pointer = 0008:8053E260, Base = 00000000, Limit = FFFFFFFF, Type = INT32
0D : Pointer = 0008:8053E46C, Base = 00000000, Limit = FFFFFFFF, Type = INT32
0E : Pointer = 0008:8053EB3C, Base = 00000000, Limit = FFFFFFFF, Type = INT32
0F : Pointer = 0008:8053EE40, Base = 00000000, Limit = FFFFFFFF, Type = INT32
10 : Pointer = 0008:8053EF44, Base = 00000000, Limit = FFFFFFFF, Type = INT32
11 : Pointer = 0008:8053F060, Base = 00000000, Limit = FFFFFFFF, Type = INT32
12 : TSS     = 00A0,          Base = 820612E8, Limit = 00000068, Type = TASK
13 : Pointer = 0008:8053F1AC, Base = 00000000, Limit = FFFFFFFF, Type = INT32
[...]
```

示列 4-15. 显示 IDT 门描述符

表 4-8. Windows 2000 中断、陷阱和任务门

INT	Intel 定义的描述符	拥有者	处理例程/TSS
0x00	整除错误（DE）	ntoskrnl.exe	KiTrap00
0x01	调试（DB）	ntoskrnl.exe	KiTrap01
0x02	NMI 中断	ntoskrnl.exe	KiNMITSS
0x03	断点（BP）	ntoskrnl.exe	KiTrap03
0x04	溢出（OF）	ntoskrnl.exe	KiTrap04
0x05	越界（BR）	ntoskrnl.exe	KiTrap05
0x06	未定义的操作码（UD）	ntoskrnl.exe	KiTrap06
0x07	没有数学协处理器（NM）	ntoskrnl.exe	KiTrap07
0x08	Double Fault（DF）	ntoskrnl.exe	KiDouble

0x09	协处理器段溢出	ntoskrnl.exe	KiTrap09
0x0A	无效的 TSS（TS）	ntoskrnl.exe	KiTrap0A
0x0B	段不存在（NP）	ntoskrnl.exe	KiTrap0B
0x0C	堆栈段故障（SS）	ntoskrnl.exe	KiTrap0C
0x0D	常规保护（GP）	ntoskrnl.exe	KiTrap0D
0x0E	页故障（PF）	ntoskrnl.exe	KiTrap0E
0x0F	Intel 保留	ntoskrnl.exe	KiTrap0F
0x10	Math Fault（MF）	ntoskrnl.exe	KiTrap10
0x11	对齐检查（AC）	ntoskrnl.exe	KiTrap11
0x12	Machine Check（MC）	？	？
0x13	流 SIMD 扩展	ntoskrnl.exe	KiTrap0F
0x14-0x1F	Intel 保留	ntoskrnl.exe	KiTrap0F
0x2A	用户自定义	ntoskrnl.exe	KiGetTickCount
0x2B	用户自定义	ntoskrnl.exe	KiCallbackReturn
0x2C	用户自定义	ntoskrnl.exe	KiSetLowWaitHighThread
0x2D	用户自定义	ntoskrnl.exe	KiDebugSerice
0x2E	用户自定义	ntoskrnl.exe	KiSystemService
0x2F	用户自定义	ntoskrnl.exe	KiTrap0F
0x30	用户自定义	hal.dll	HalpClockInterrupt
0x38	用户自定义	hal.dll	HalpProfileInterrupt

4.5.3、Windows 2000 的内存区域

W2k_mem.exe 的最后一个还未讨论的选项是：+b 选项。该选项会产生 4GB 地址空间中相邻内存区域的列表，这个列表非常大。W2k_mem.exe 使用 Spy 设备的 IOCTL 函数 SPY_IO_PAGE_ENTRY 遍历整个 PTE 数组（位于地址 0xC0000000）来生成这个列表。在作为结果的每个 SPY_PAGE_ENTRY 结构中，通过将它们的 dSize 成员与其对应的 PTE 线性地址相加即可得到下一个 PTE 的地址。**列表 4-30** 给出了该选项的实现方式。

```
DWORD WINAPI DisplayMemoryBlocks (HANDLE hDevice)
    {
```

```
SPY_PAGE_ENTRY spe;
PBYTE          pbPage, pbBase;
DWORD          dBlock, dPresent, dTotal;
DWORD          n = 0;

pbPage   = 0;
pbBase   = INVALID_ADDRESS;
dBlock   = 0;
dPresent = 0;
dTotal   = 0;

n += _printf (L"\r\nContiguous memory blocks:"
              L"\r\n------------------------\r\n\r\n");

do  {
    if (!IoControl (hDevice, SPY_IO_PAGE_ENTRY,
                    &pbPage, PVOID_,
                    &spe,    SPY_PAGE_ENTRY_))
        {
        n += _printf (L" !!! Device I/O error !!!\r\n");
        break;
        }
    if (spe.fPresent)
        {
        dPresent += spe.dSize;
        }
    if (spe.pe.dValue)
        {
        dTotal += spe.dSize;
```

```
        if (pbBase == INVALID_ADDRESS)
            {
            n += _printf (L"%5lu : 0x%08lX ->",
                          ++dBlock, pbPage);

            pbBase = pbPage;
            }
        }
    else
        {
        if (pbBase != INVALID_ADDRESS)
            {
            n += _printf (L" 0x%08lX (0x%08lX bytes)\r\n",
                          pbPage-1, pbPage-pbBase);

            pbBase = INVALID_ADDRESS;
            }
        }
    }
while (pbPage += spe.dSize);

if (pbBase != INVALID_ADDRESS)
    {
    n += _printf (L"0x%08lX\r\n", pbPage-1);
    }
n += _printf (L"\r\n"
              L" Present bytes: 0x%08lX\r\n"
              L" Total   bytes: 0x%08lX\r\n",
```

```
                dPresent, dTotal);
    return n;
    }
```

列表 4-30. 查找相邻的线性内存块

示列 4-16 摘录了在我的机器上使用+b 选项的输出列表，可以看出其中的几个区域非常有趣。一些非常明显的地址是：0x00400000，这是 w2k_mem.exe 内存映像的起始地址（第 13 号块），还有一个是 0x10000000，此处是 w2k_lib.dll 的基地址（第 23 号块）。TEB 和 PEB 页也很容易认出（第 104 号块），hal.dll（第 105 号块），ntoskrnl.exe（第 105 号块），win32k.sys（第 106 号块）。第 340---350 号块是系统 PTE 数组的一小段，第 347 号块是页目录的一部分。第 2122 号块包含 SharedUserData 区域，第 2123 号块由 KPCR、KPRCB 和包含线程和进程状态信息的 CONTEXT 结构组成。

```
E:\>w2k_mem +b
[...]
Contiguous memory blocks:

    1   0x00010000 -> 0x00010FFF (0x00001000 bytes)
    2   0x00020000 -> 0x00020FFF (0x00001000 bytes)
    3   0X0012D000 -> 0X00138FFF (0x0000C000 bytes)
    4   0x00230000 -> 0X00230FFF (0x00001000 bytes)
    5   0x00240000 -> 0X00241FFF (0x00002000 bytes)
    6   0x00247000 -> 0x00247FFF (0x00001000 bytes)
    7   0X0024F000 -> 0x00250FFF (0x00002000 bytes)
    8   0x00260000 -> 0x00260FFF (0x00001000 bytes)
    9   0x00290000 -> 0x00290FFF (0x00001000 bytes)
   10   0x002E0000 -> 0x002E0FFF (0x00001000 bytes)
   11   0x002E2000 -> 0x002E3FFF (0x00002000 bytes)
   12   0x003B0000 -> 0x003B1FFF (0x00002000 bytes)
   13   0x00400000 -> 0x00404FFF (0x00005000 bytes)
   14   0x00406000 -> 0x00406FFF (0x00001000 bytes)
   15   0x00410000 -> 0x00410FFF (0x00001000 bytes)
   16   0x00419000 -> 0x00419FFF (0x00001000 bytes)
   17   0x0041B000 -> 0x0041BFFF (0x00001000 bytes)
   18   0x00450000 -> 0x00450FFF (0x00001000 bytes)
   19   0x00760000 -> 0X00760FFF (0x00001000 bytes)
   20 : 0x00770000 -> 0x00770FFF (0x00001000 bytes)
   21 : 0x00780000 -> 0x00783FFF (0x00004000 bytes)
   22 : 0x00790000 -> 0x00791FFF (0x00002000 bytes)
   23 : 0x10000000 -> 0x10003FFF (0x00004000 bytes)
   24 : 0x10005000 -> 0x10005FFF (0x00001000 bytes)
   25 : 0x1000E000 -> 0x10016FFF (0x00009000 bytes)
   26 : 0x759B0000 -> 0X759B1FFF (0x00002000 bytes)
[...]
  103 : 0x7FFD2000 -> 0x7FFD3FFF (0x00002000 bytes)
  104 : 0x7FFDE000 -> 0x7FFE0FFF (0x00003000 bytes)
  105 : 0x80000000 -> 0xA01A5FFF (0x201A6000 bytes)
  106 : 0xA01B0000 -> 0XA01F2FFF (0x00043000 bytes)
  107 : 0xA0200000 -> 0XA02C7FFF (0x000C8000 bytes)
  108 : 0xA02F0000 -> 0xA03FFFFF (0x00110000 bytes)
  109 : 0xA4000000 -> 0XA4001FFF (0x00002000 bytes)
  110 : 0xBE63B000 -> 0XBE63CFFF (0x00002000 bytes)
[...]
  340 : 0xC0000000 -> 0xC0001FFF (0x00002000 bytes)
  341 : 0xC0040000 -> 0xC0040FFF (0x00001000 bytes)
  342 : 0xC01D6000 -> 0XC01D6FFF (0x00001000 bytes)
  343 : 0xC01DA000 -> 0xC01DAFFF (0x00001000 bytes)
  344 : 0xC01DD000 -> 0xC01E0FFF (0x00004000 bytes)
  345 : 0xC01FD000 -> 0xC01FDFFF (0x00001000 bytes)
  346 : 0xC01FF000 -> 0xC0280FFF (0x00082000 bytes)
  347 • 0xC0290000 -> 0XC0301FFF (0x00072000 bytes)
  348   0xC0303000 -> 0xC0386FFF (0x00084000 bytes)
  349 : 0xC0389000 -> 0xC038CFFF (0x00004000 bytes)
  350 • 0xC039E000 -> 0XC03FFFFF (0x00062000 bytes)
[...]
 2121   0xFFC00000 -> 0xFFD0FFFF (0x00110000 bytes)
 2122   0xFFDF0000 -> 0xFFDF0FFF (0x00001000 bytes)
 2123   0xFFDFF000 -> 0xFFDFFFFF (0x00001000 bytes)
[...]
 Present bytes: 0x22AA9000
 Total   bytes: 0x2B8BA000
 [...]
```

示列 4-16. 相邻内存块列表示列

还需要补充一下，W2k_mem.exe 的+b 选项会报告有大量的内存被使用，这可能超出了一个合理的值（比如，你机器上的物理内存数）。请注意**示列 4-16** 底部给出的汇总信息。我现在真的使用了 700MB 的内存吗？Windows 2000 的任务管理器显示是 150MB，那么这儿的又是什么呢？这种奇特的效果都是由第 105 号内存块产生的，该内存块表示的范围：0x80000000----0xA01A5FFF 占用了 0x201A6000 字节，也就是说占用了 538,599,424 字节。这显然是不可能的。问题是整个线性地址空间：0x80000000 ---- 0x9FFFFFFF 都被映射到了物理内存：0x00000000 ---- 0x1FFFFFFF，在前面我已经提及过这一点。该区域中的所有 4MB 页都对应地址 0xC0300000 处的页目录中的一个有效的 PDE，我们可以使用 w2k_mem +d #0x200 0xC0300800 命令来证明这一点（**示列 4-17**）。因为结果列表中的所有 PDE 都是奇数（译注：如果 PDE 为奇数，证明其 P 位肯定为 1），所以它们对应的页都必须存在；不过，它们并不需真正占用物理内存。事实上，这一内存区域的大部分都是“空洞（hole）”，如果将其复制到缓冲区中，可发现它们都被 0xFF 填充。因此，对于 w2k_mem.exe 输出的内存使用情况，你不需要过于认真。

```
E:\>w2k_mem +d #0x200 0xC0300800
[...]
C0300800..C03009FF: 512 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|-------------------
C0300800 | 000001E3 - 004001E3 : 008001E3 - 00C001E3 | ...?.@.?.▒.?.??▒
C0300810 | 010001E3 - 014001E3 : 018001E3 - 01C001E3 | ...?.@.?.▒.?.??▒
C0300820 | 020001E3 - 024001E3 : 028001E3 - 02C001E3 | ...?.@.?.▒.?.??▒
C0300830 | 030001E3 - 034001E3 : 038001E3 - 03C001E3 | ...?.@.?.▒.?.??▒
C0300840 | 040001E3 - 044001E3 : 048001E3 - 04C001E3 | ...?.@.?.▒.?.??▒
C0300850 | 050001E3 - 054001E3 : 058001E3 - 05C001E3 | ...?.@.?.▒.?.??▒
C0300860 | 060001E3 - 064001E3 : 068001E3 - 06C001E3 | ...?.@.?.▒.?.??▒
C0300870 | 070001E3 - 074001E3 : 078001E3 - 07C001E3 | ...?.@.?.▒.?.??▒
C0300880 | 080001E3 - 084001E3 : 088001E3 - 08C001E3 | ...?.@.?.▒.?.??▒
C0300890 | 090001E3 - 094001E3 : 098001E3 - 09C001E3 | ...?.@.?.▒.?.??▒
C03008A0 | 0A0001E3 - 0A4001E3 : 0A8001E3 - 0AC001E3 | ...?.@.?.▒.?.??▒
C03008B0 | 0B0001E3 - 0B4001E3 : 0B8001E3 - 0BC001E3 | ...?.@.?.▒.?.??▒
C03008C0 | 0C0001E3 - 0C4001E3 : 0C8001E3 - 0CC001E3 | ...?.@.?.▒.?.??▒
C03008D0 | 0D0001E3 - 0D4001E3 : 0D8001E3 - 0DC001E3 | ...?.@.?.▒.?.??▒
C03008E0 | 0E0001E3 - 0E4001E3 : 0E8001E3 - 0EC001E3 | ...?.@.?.▒.?.??▒
C03008F0 | 0F0001E3 - 0F4001E3 : 0F8001E3 - 0FC001E3 | ...?.@.?.▒.?.??▒
C0300900 | 100001E3 - 104001E3 : 108001E3 - 10C001E3 | ...?.@.?.▒.?.??▒
C0300910 | 110001E3 - 114001E3 : 118001E3 - 11C001E3 | ...?.@.?.▒.?.??▒
C0300920 | 120001E3 - 124001E3 : 128001E3 - 12C001E3 | ...?.@.?.▒.?.??▒
C0300930 | 130001E3 - 134001E3 : 138001E3 - 13C001E3 | ...?.@.?.▒.?.??▒
C0300940 | 140001E3 - 144001E3 : 148001E3 - 14C001E3 | ...?.@.?.▒.?.??▒
C0300950 | 150001E3 - 154001E3 : 158001E3 - 15C001E3 | ...?.@.?.▒.?.??▒
C0300960 | 160001E3 - 164001E3 : 168001E3 - 16C001E3 | ...?.@.?.▒.?.??▒
C0300970 | 170001E3 - 174001E3 : 178001E3 - 17C001E3 | ...?.@.?.▒.?.??▒
C0300980 | 180001E3 - 184001E3 : 188001E3 - 18C001E3 | ...?.@.?.▒.?.??▒
C0300990 | 190001E3 - 194001E3 : 198001E3 - 19C001E3 | ...?.@.?.▒.?.??▒
C03009A0 | 1A0001E3 - 1A4001E3 : 1A8001E3 - 1AC001E3 | ...?.@.?.▒.?.??▒
C03009B0 | 1B0001E3 - 1B4001E3 : 1B8001E3 - 1BC001E3 | ...?.@.?.▒.?.??▒
C03009C0 | 1C0001E3 - 1C4001E3 : 1C8001E3 - 1CC001E3 | ...?.@.?.▒.?.??▒
C03009D0 | 1D0001E3 - 1D4001E3 : 1D8001E3 - 1DC001E3 | ...?.@.?.▒.?.??▒
C03009E0 | 1E0001E3 - 1E4001E3 : 1E8001E3 - 1EC001E3 | ...?.@.?.▒.?.??▒
C03009F0 | 1F0001E3 - 1F4001E3 : 1F8001E3 - 1FC001E3 | ...?.@.?.▒.?.??▒
[...]
```

示列 4-17.　地址范围是：0x80000000　--- 0x9FFFFFFF 的 PDE

4.5.4、Windows 2000 的内存布局

本章的最后一部分将给出在一个Windows 2000 进程“看”来，4GB线性地址空间的总体布局是什么样子。**表 4-9** 给出了多个基本数据结构的内存范围。它们之间的“大洞（big hole）”有不同的用途，如，用于进程模块和设备驱动程序的加载区域，内存池，工作集链表等等。注意，有些内存地址和内存块的大小在不同的系统之间有很大的差异，这取决于物理内存和硬件的配置情况、进程的属性以及其他一些系统变量。因此，这里给出的仅仅是一个草图而已，并不是精确的布局图。

有些物理内存块在线性地址空间中出现的两次或更多次。例如，SharedUserData 区域位于线性地址 0xFFDF0000，并且并镜像到 0x7FFE0000。这两个地址都指向物理内存中的同一个页，这意味着，如果向 0xFFDF0000+n 处写入一个字节，那么 0x7FFE0000+n 处的值也会

随之改变。这是一个虚拟内存的世界----一个物理地址可以被映射到线性地址空间中的任何地方，即使一个物理地址在同一时间映射到多个线性地址也是可以的。回忆一下**图 4-3** 和**图 4-4**，它们清楚地展示了线性地址的这种“虚假行为”。它们的目录和表位域正确的指向用来确定数据实际位置的结构体。如果两个 PTE 的 PFN 恰好是相同的，那么它们对应的线性地址将指向物理内存相同位置。

表 4-9. 进程地址空间中的可确认的内存区域

起始地址	结束地址	十六进制大小	类型/描述
0x00000000	0x0000FFFF	10000	底部的受保护块（Lower guard block）
0x00010000	0x0001FFFF	10000	WCHAR[]/环境字符串，在一个 4KB 页中分配
0x00020000	0x0002FFFF	10000	PROCESS_PARAMETERS/在一个 4KB 页中分配
0x00030000	0x0012FFFF	1000000	DWORD[4000]/进程堆栈（默认；1MB）
0x7FFDD000	0x7FFDDFFF	1000	TEB/1#线程的线程环境块
0x7FFDE000	0x7FFDEFFF	1000	TEB/2#线程的线程环境块
0x7FFDF000	0x7FFDFFFF	1000	PEB/进程环境块
0x7FFE0000	0x7FFE02D7	2D8	KUSER_SHARED_DATA/用户模式下的 SharedUserData
0x7FFF0000	0x7FFFFFFF	10000	顶部的受保护块（Upper guard block）
0x80000000	0x800003FF	400	IVT/中断向量表
0x80036000	0x800363FF	400	KGDTENTRY[80]/全局描述符表
0x80036400	0x80036BFF	800	KIDTENTRY[100]/中断描述符表
0x800C0000	0x800FFFFF	40000	VGA/ROM BIOS
0x80244000	0x802460AA	20AB	KTSS/内核任务状态段（繁忙）
0x8046AB80	0x8046ABBF	40	KeServiceDescriptorTable
0x8046AB	0x8046ABFF	40	KeServiceDescriptorTableShadow
0x80470040	0x804700A7	68	KTSS/KiDoubleFaultTSS
0x804700A8	0x8047010F	68	KTSS/KiNMITSS
0x804704D8	0x804708B7	3E0	PROC[F8]/KiServiceTable

0x804708B8	0x804708BB	4	DWORD/KiServiceLimit
0x804708BC	0x804709B3	F8	BYTE[F8]/KiArgumentTable
0x814C6000	0x82CC5FFF	1800000	PFN[100000]/MmPfnDatabase（最大为4GB）
0xA01859F0	0xA01863EB	9FC	PROC[27F]/W32pServiceTable
0xA0186670	0x A01863EE	27F	BYTE[27F]W32pArgumentTable
0xC0000000	0xC03FFFFF	400000	X86_PE[100000]/页目录和页表
0xC1000000	0xE0FFFFFF	20000000	系统缓存（MmSystemCacheStart, MmSystemCacheEnd）
0xE1000000	0xE77FFFFF	6800000	页池（Paged Pool）（MmPagedPoolStart, MmPagedPoolEnd）
0xF0430000	0xF043FFFF	10000	ROM BIOS 代码段
0xF0440000	0xF044FFFF	10000	ROM BIOS 数据段
0xFFDF0000	0xFFDF02D7	2D8	KUSER_SHARED_DATA/内核模式下的 SharedUserData
0xFFDFF000	0xFFDFF053	54	KPCR/处理器控制区（内核模式 FS 段）
0xFFDFF120	0xFFDFF13B	1C	KPRCB/处理器控制块
0xFFDFF13C	0xFFDFF407	2CC	CONTEXT/线程 CONTEXT（CPU 状态）
0xFFDFF620	0xFFDFF71F	100	后备链表目录（Lookaside list directories）

五、监控 Native API 调用

拦截系统调用在任何时候都是程序员们的最爱。这种大众化爱好的动机也是多种多样的：代码性能测试（Code Profiling）和优化，逆向工程，用户活动记录等等。所有这些都有一个共同的目的：将控制传递给一块特殊的代码，这样无论一个应用程序何时调用系统服务，都可以发现哪个服务被调用了，接收了什么参数，返回的结果是什么以及执行它花费了多少时间。根据最初由 Mark Russinovich 和 Bryce Cogswell 提出的技巧，本章将介绍一个可以 hook 任意 Native API 函数的通用框架。这里使用的方法完全是数据驱动（data-driven）的，因此，它可以很容易被扩展，并能适应其他 Windows NT/2000 版本。所有进程的 API 调用产生的数据都被写入一个环状缓冲区中，客户端程序可以通过设备 I/O 控制来读取该缓冲区。采用的数据协议（protocol data）的格式是以行为导向的 ANSI 文本流，它符合严格的格式化规则，应用程序可以很容易的再次处理它们（postprocessing）。为了示范此种客户端程序的基本框架，本章还提供了一个示例性的数据协议察看器，该程序运行于控制台窗口中。

译注：

profiling 一般是指对程序做性能方面的测试，主要是速度上的。 在翻译时，可译为：剖析，最好是根据上文环境进行翻译。

5.1、修改服务描述符表

对比“原始”的操作系统，如 DOS 或 Windows 3.x，它们对程序员在 API 中加入 hook 的限制很少，而 Win32 系统，如 Windows 2000/NT 和 Windows 9.x 则很难驾驭，因为它们使用了巧妙的保护机制把不相关的代码分离出来。在 Win32 API 上设置一个系统范围的 hook 绝不是一个小任务。幸运的是，我们有像 Matt Pietrek 和 Jeffery Richter 这样的 Win32 向导，他们做了大量的工作来向我们展示如何完成这一任务，尽管事实上，并没有简单和优雅的解决方案。在 1997 年，Russinovich 和 Cogswell 介绍了一种可在 Windows NT 上实现系统范围 hook 的完全不同的方法，可在更低一层上拦截系统调用（Russinovich 和 Cogswell 1997）。他们提议向 Native API Dispatcher 中注入日志机制，这仅比用户模式和内核模式

之间的边界低一些，在这个位置上 Windows NT 暴露出一个“瓶颈”：所有用户模式的线程必须通过此处，才能使用操作系统内核提供的服务。

5.1.1、服务和参数表

就像在第二章讨论过的，发生在用户模式下的 Native API 调用必须通过 INT 2eh 接口，该接口提供一个 i386 的中断门来改变特权级别。你可能还记得所有 INT 2eh 调用都是由内部函数 KiSystemService()在内核模式下处理的，该函数使用系统服务描述符表（SDT）来查找 Native API 处理例程的入口地址。**图 5-1** 给出了这种分派机制的基本组件之间的相互关系。**列表 5-1** 再次给出了 SERVICE_DESCRIPTOR_TABLE 结构及其子结构的正式定义，在第二章中，已经给出过它们的定义（**列表 2-1**）。

调用 KiSystemService()时，需提供两个参数，由 INT 2eh 的调用者通过 EAX 和 EDX 寄存器传入。EAX 包含从 0 开始的索引，该索引可用于一个 API 函数指针的数组，EDX 指向调用者的参数堆栈。KiSystemService()通过读取 ServiceTable 的一个成员的值（该成员是 ntoskrnl.exe 的一个公开数据结构：KeServiceDescriptorTable，**图 5-1** 的左面列出了该结构）来获取函数数组的基地址。实际上，KeServiceDescriptorTable 指向一个包含四个服务表的结构，但默认情况下，仅有第一个服务表是有效的。KiSystemService()通过 EAX 中的索引值进入内部结构 KiServiceTable，以查找可处理此 API 调用的函数的入口地址。在调用目标函数之前，KiSystemServie()以相同的方式查询 KiArgumentTable 结构，以找出调用者在参数堆栈中传入了多少字节，然后使用这个值将参数复制到当前内核堆栈中。此后，就只需要一个简单的汇编指令：CALL 来执行 API 处理例程了。这里涉及的所有函数都符合 __stdcall 标准。

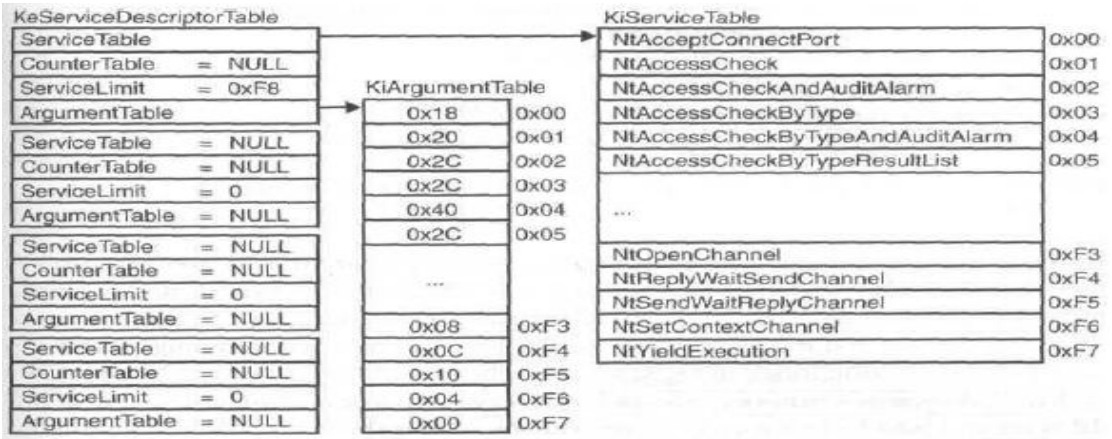


图 5-1. KeServiceDescriptorTable 的结构图

```
typedef NTSTATUS (NTAPI*NTPROC)();
typedef NTPROC* PNTPROC;
#define NTPROC_ sizeof(NTPROC)

typedef struct _SYSTEM_SERVICE_TABLE
{
    PNTPROC  ServiceTable;   // array of entry points
    PDOWRD  CounterTable;   // array of usage counters
    DWORD   ServiceLimit;    // number of table entries
    PBYTE    ArgumentTable;  // array of byte counts
}
SYSTEM_SERVICE_TABLE,
*PSYSTEM_SERVICE_TABLE,
**PPSYSTEM_SERVICE_TABLE;

//-------------------------------------------------------------------------
-----------------------------

typedef struct _SERVICE_DESCRIPTOR_TABLE
{
    SYSTEM_SERVICE_TABLE ntoskrnl;  // ntoskrnl.exe ( native api )
    SYSTEM_SERVICE_TABLE win32k;    // win32k.sys (gdi/user support)
    SYSTEM_SERVICE_TABLE Table3;    // not used
    SYSTEM_SERVICE_TABLE Table4;    // not used
}
SYSTEM_DESCRIPTOR_TABLE,
*PSYSTEM_DESCRIPTOR_TABLE,
**PPSYSTEM_DESCRIPTOR_TABLE;
```

列表 5-1. SERVICE_DESCRIPTOR_TABLE 结构的定义

Windows 2000 还提供了另一个服务描述符表参数块----KeServiceDescriptorTableShadow。不过，KeServiceDescriptorTable 已经由 ntoskrnl.exe 公开的导出了，因此，内核模式的驱动程序可以很容易的访问它，而 KeServiceDescriptorTableShadow 则不行。在 Windows 2000 下，KeServiceDescriptorTableShadow 紧随 KeServiceDescriptorTable 之后，但是你不能在 Windows NT 中以这样的方法找到它，因为，这一规则并不使用于 Windows NT。可能在 Windows 2000 的后续版本的中，这种方法也不行。这两个参数块的不同之处在于：KeServiceDescriptorTableShadow 中的第二个项被系统使用了，用来指向内部的 W32pServiceTable 和 w32pArgumentTable 结构，Win32 的内核模式组件 Win32K.sys 使用这两个结构来分派自己的 API 调用，如**图 5-2** 所示。KiSystemService()通过检查 EAX 中索引值的第 12 和 13 位来确认是不是应该由 Win32K.sys 处理 API 调用。如果这两个位都是 0，则是由 ntoskrnl.exe 处理的 Native API 调用，因此 KiSystemService()使用第一个 SDT。如果第 12 位为 1 并且第 13 位为 0，KiSystemService()使用第二个 SDT，这个 SDT 并没有被当前系统使用。这意味着 Native API 调用的索引值的潜在范围是：0x0000 --- 0x0FFFF，Win32K.sys 调用使用的索引范围是：0x1000 --- 0x1FFF。因此，0x2000 --- 0x2FFF 和 0x3000 --- 0x3FFF 保留给剩下的两个 SDT。在 Windows 2000 中，Native API 服务表包含 248 个项，Win32K.sys 表包含 639 个项。

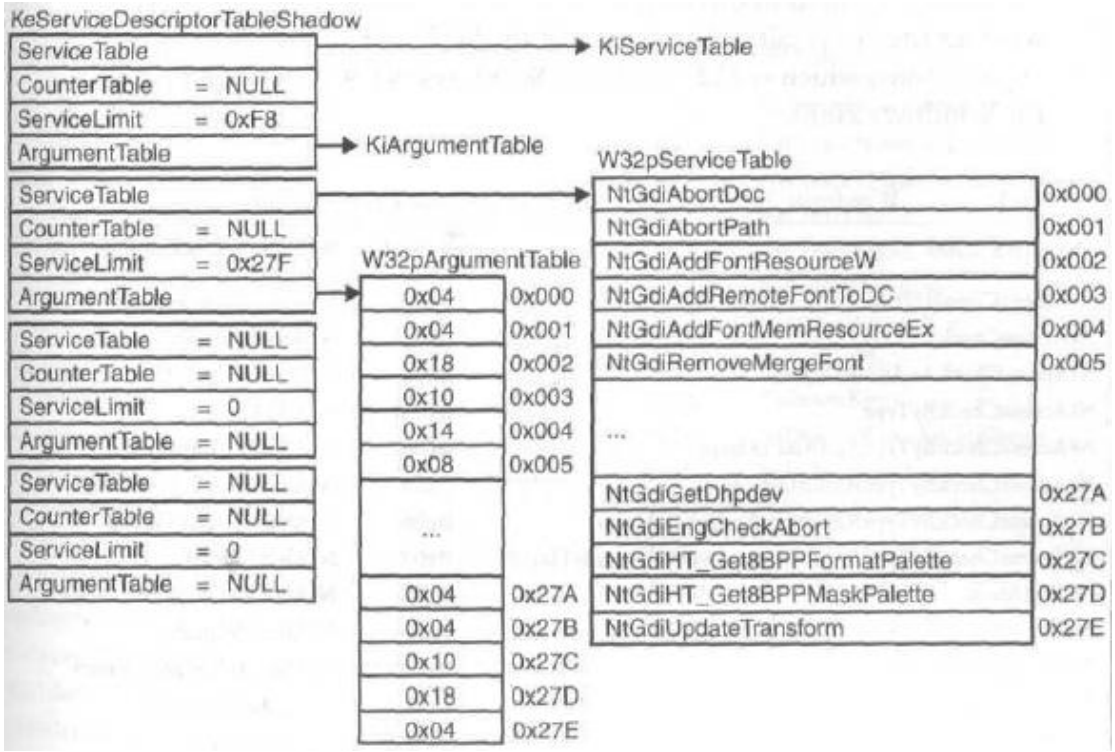


图 5-2. KeServiceDescriptorTableShadow 的结构图

Russinovich 和 Cogswell 的独具特色的方法是：通过简单的向 KiServiceTable 数组中放入一个不同的处理例程来 hook 所有 API 调用。这个处理例程最终会调用位于 ntoskrnl.exe 中的原始处理例程，但它有机会察看一下被调用函数的输入/输出参数。这个方法非常强大却又如此简单。因为所有用户模式的线程必须经过这个“针眼”才能获得 Native API 的服务，安装一个全局 hook 来简单的替换一个函数指针的方法，在启动一个新的进程和线程的情况下，也能很稳定的工作。这并不需要一种通讯机制来通知新加入或将要移除的进程/线程。

不幸的是，系统服务表在不同 Windows NT 版本上不相同。**表 5-1** 比较了 Windows NT/2000 的 KiServiceTable。很显然，不仅是处理例程的号码从 211 增加到了 248，而且新的处理例程并不是直接添加到列表的末尾，而是被插入到了各个地方！因此，一个服务函数索引，如 0x20 在 Windows 2000 中指向 NtCreateFile()，而在 Windows NT 中却指向 NtCreateProfile()。所以，通过操作服务函数表进行 hook 的 API 调用监控器必须小心的检查它所在的 Windows NT 的版本。这可以通过如下几个方式来完成：

- 一种可能性是，检查由 ntoskrnl.exe 导出的公开变量：NtBuildNumber，就像 Russinovich 和 Cogswell 在他们的原文中所作的那样。Windows NT 4.0 为所有 Service Pack 提供的 Build Number 是：1381。Windows 2000 的当前 Build Number 是：2195。看来有希望，这个版本号在 Windows NT 的早期版本中很稳定。
- 另一个可能性是，检查 SharedUserData 结构中的 NtMajorVersion 和 NtMinorVersion 成员，该结构定义与 Windows 2000 头文件 ntddk.h 中。Windows NT 4.0 的所有 Service Pack 都将 SharedUserData->NtMajorVersion 设为 4，将 SharedUserData->NtMinorVersion 设为 0。Windows 2000 的当前版本为 Windows NT Version 5.0。
- 本章给出的代码采用了另一中替代方法：它测试 SDT 的 ServiceLimit 成员是否和它的预期值相匹配，该预期值是 211（0xD3）针对 Windows NT 4.0 和 248（0xF8）针对 Windows 2000。

表 5-1. Windows 2000/NT 服务表对比

Windows 2000	索引	Windows NT 4.0
NtAcceptConnectPort	0x00	NtAcceptConnectPort
NtAccessCheck	0x01	NtAccessCheck

NtAccessCheckAndAuditAlarm	0x02	NtAccessCheckAndAuditAlarm
NtAccessCheckByType	0x03	NtAddAtom
NtAccessCheckByTypeAndAuditAlarm	0x04	NtAdjustGroupsToken
NtAccessCheckByTypeResultList	0x05	NtAdjustPrivilegesToken
NtAccessCheckByTypeResultListAndAuditAlarm	0x06	NtAlertResumeThread
NtAccessCheckByTypeResultListAndAuditAlarmByHandle	0x07	NtAlertThread
NtAddAtom	0x08	NtAllocateLocallyUniqueId
NtAdjustGroupsToken	0x09	NtAllocateUuids
NtAdjustPrivilegesToken	0x0A	NtAllocateVirtualMemory
NtAlertResumeThread	0x0B	NtCallbackReturn
NtAlertThread	0x0C	NtCancelIoFile
NtAllocateLocallyUniqueId	0x0D	NtCancelTimer
NtAllocateUserPhysicalPages	0x0E	NtClearEvent
NtAllocateUuids	0x0F	NtClose
NtAllocateVirtualMemory	0x10	NtCloseObjectAuditAlarm
NtAreMappedFilesTheSame	0x11	NtCompleteConnectPort
NtAssignProcessToJobObject	0x12	NtConnectPort
NtCallbackReturn	0x13	NtContinue
NtCancelIoFile	0x14	NtCreateDirectoryObject
NtCancelTi mer	0x15	NtCreateEvent
NtCancelDeviceWakeupRequest	0x16	NtCreateEventPair
NtClearEvent	0x17	NtCreateFile
NtClose	0x18	NtCreateIoCompletion
NtCloseObjectAuditAlarm	0x19	NtCreateKey
NtCompleteConnectPort	0x1A	NtCreateMailslotFile
NtConnectPort	0x1B	NtCreateMutant
NtContinue	0x1C	NtCreateNamedPipeFile

NtCreateDirectoryObject	0x1D	NtCreatePagingFile
NtCreateEvent	0x1E	NtCreatePort
NtCreateEventPair	0x1F	NtCreateProcess
NtCreateFile	0x20	NtCreateProfile
NtCreateloCompletion	0x21	NtCreateSection
NtCreateJobObject	0x22	NtCreateSemaphore
NtCreateKey	0x23	NtCreateSymbolicLinkObject
NtCreateMailslotFile	0x24	NtCreateThread
NtCreateMutant	0x25	NtCreateTimer
NtCreateNamedPipeFile	0x26	NtCreateToken
NtCreatePagingFile	0x27	NtDelayExecution
NtCreatePort	0x28	NtDeleteAtom
NtCreateProcess	0x29	NtDeleteFile
NtCreateProfile	0x2A	NtDeleteKey
NtCreateSection	0x2B	NtDeleteObjectAuditAlarm
NtCreateSemaphore	0x2C	NtDeleteValueKey
NtCreateSymbolicLinkObject	0x2D	NtDeviceloControlFile
NtCreateThread	0x2E	NtDisplayString
NtCreateTimer	0x2F	NtDuplicateObject
NtCreateToken	0x30	NtDuplicateToken
NtCreateWaitablePort	0x31	NtEnumerateKey
NtDelayExecution	0x32	NtEnumerateValueKey
NtDeleteAtom	0x33	NtExtendSection
NtDeleteFile	0x34	NtFindAtom
NtDeleteKey	0x35	NtFlushBuffersFile
NtDeleteObj ectAuditAlarm	0x36	NtFlushlnstructionCache
NtDeleteValueKey	0x37	NtFlushKey
NtDeviceloControlFile	0x38	NtFlushVirtualMemory
NtDisplayString	0x39	NtFlushWriteBuffer

NtDuplicateObject	0x3A	NtFreeVirtualMemory
NtDuplicateToken	0x3B	NtFsControlFile
NtEnumerateKey	0x3C	NtGetContextThread
NtEnumerateValueKey	0x3D	NtGetPlugPlayEvent
NtExtendSection	0x3E	NtGetTickCount
NtFilterToken	0x3F	NtlmpersonateClientOfPort
NtFindAtom	0x40	NtlmpersonateThread
NtFlushBuffersFile	0x41	NtlnitializeRegistry
NtFlushlnstructionCache	0x42	NtListenPort
NtFlushKey	0x43	NtLoadDriver
NtFlushVirtualMemory	0x44	NtLoadKey
NtFlushWriteBuffer	0x45	NtLoadKey2
NtFreeUserPhysicalPages	0x46	NtLockFile
NtFreeVirtualMemory	0x47	NtLockVirtualMemory
NtFsControlFile	0x48	NtMakeTemporaryObject
NtGetContextThread	0x49	NtMapViewOfSection
NtGetDevicePowerState	0x4A	NtNotifyChangeDirectoryFil e
NtGetPlugPlayEvent	0x4B	NtNotifyChangeKey
NtGetTickCount	0x4C	NtOpenDirectoryObject
NtGetWriteWatch	0x4D	NtOpenEvent
NtlmpersonateAnonymousToken	0x4E	NtOpenEventPair
NtlmpersonateClientOfPort	0x4F	NtOpenFile
NtlmpersonateThread	0x50	NtOpenloCompletion
NtlnitializeRegistry	0x51	NtOpenKey
NtlnitiatePowerAction	0x52	NtOpenMutant
NtlsSystemResumeAutomatic	0x53	NtOpenObjectAuditAlarm
NtListenPort	0x54	NtOpenProcess
NtLoadDriver	0x55	NtOpenProcessToken

NtLoadKey	0x56	NtOpenSection
NtLoadKey2	0x57	NtOpenSemaphore
NtLockFile	0x58	NtOpenSymbolicLinkObject
NtLockVirtualMemory	0x59	NtOpenThread
NtMakeTemporaryObject	0x5A	NtOpenThreadToken
NtMapUserPhysicalPages	0x5B	NtOpenTimer
NtMapUserPhysicalPagesScatter	0x5C	NtPlugPlayControl
NtMapViewOfSection	0x5D	NtPrivilegeCheck
NtNotifyChangeDirectoryFile	0x5E	NtPrivilegedServiceAuditAlarm
NtNotifyChangeKey	0x5F	NtPrivilegeObjectAuditAlarm
NtNotifyChangeMultipleKeys	0x60	NtProtectVirtualMemory
NtOpenDirectoryObject	0x61	NtPulseEvent
NtOpenEvent	0x62	NtQuerylnformationAtom
NtOpenEventPair	0x63	NtQueryAttributesFile
NtOpenFile	0x64	NtQueryDefaultLocale
NtOpenloCompletion	0x65	NtQueryDirectoryFile
NtOpenJobObject	0x66	NtQueryDirectoryObject
NtOpenKey	0x67	NtQueryEaFile
NtOpenMutant	0x68	NtQueryEvent
NtOpenObjectAuditAlarm	0x69	NtQueryFullAttributesFile
NtOpenProcess	0x6A	NtQuerylnformationFile
NtOpenProcessToken	0x6B	NtQueryloCompletion
NtOpenSection	0x6C	NtQuerylnformationPort
NtOpenSemaphore	0x6D	NtQuerylnformationProcess
NtOpenSymbolicLinkObject	0x6E	NtQuerylnformationThread
NtOpenThread	0x6F	NtQuerylnformationToken
NtOpenThreadToken	0x70	NtQuerylntervalProfile

NtOpenTimer	0x71	NtQueryKey
NtPlugPlayControl	0x72	NtQueryMultipleValueKey
NtPowerlnformation	0x73	NtQueryMutant
NtPrivilegeCheck	0x74	NtQueryObject
NtPrivilegedServiceAuditAlarm	0x75	NtQueryOleDirectoryFile
NtPrivilegeObjectAuditAlarm	0x76	NtQueryPerformanceCounter
NtProtectVirtualMemory	0x77	NtQuerySection
NtPulseEvent	0x78	NtQuerySecurityObject
NtQuerylnformationAtom	0x79	NtQuery Semaphore
NtQueryAttributesFile	0x7A	NtQuerySymbolicLinkObject
NtQueryDefaultLocale	0x7B	NtQuerySystemEnvironmentValue
NtQueryDefaultUILanguage	0x7C	NtQuerySystemlnformation
NtQueryDirectoryFile	0x7D	NtQuerySystemTime
NtQueryDirectoryObject	0x7E	NtQuery Timer
NtQueryEaFile	0x7F	NtQueryTimerResolution
NtQueryEvent	0x80	NtQuery ValueKey
NtQueryFullAttributesFile	0x81	NtQuery VirtualMemory
NtQuerylnformationFile	0x82	NtQuery VolumelnformationFile
NtQuerylnformationJobObject	0x83	NtQueueApcThread
NtQueryloCompletion	0x84	NtRaiseException
NtQuerylnformationPort	0x85	NtRaiseHardError
NtQuerylnformationProcess	0x86	NtReadFile
NtQuerylnformationThread	0x87	NtReadFileScatter
NtQuerylnformationToken	0x88	NtReadRequestData
NtQuerylnstallUILanguage	0x89	NtReadVirtualMemory
NtQuerylntervalProfile	0x8A	NtRegisterThreadTerminatePort

NtQueryKey	0x8B	NtReleaseMutant
NtQueryMultiple ValueKey	0x8C	NtReleaseSemaphore
NtQueryMutant	0x8D	NtRemoveloCompletion
NtQueryObject	0x8E	NtReplaceKey
NtQueryOpenSubKeys	0x8F	NtReplyPort
NtQueryPerformanceCounter	0x90	NtReplyWaitReceivePort
NtQueryQuotalnformationFile	0x91	NtReplyWaitReplyPort
NtQuerySection	0x92	NtRequestPort
NtQuerySecurityObject	0x93	NtRequestWaitReplyPort
NtQuerySemaphore	0x94	NtResetEvent
NtQuerySymbolicLinkObject	0x95	NtRestoreKey
NtQuerySystemEnvironmentValue	0x96	NtResumeThread
NtQuerySystemlnformation	0x97	NtSaveKey
NtQuerySystemTime	0x98	NtSetloCompletion
NtQueryTimer	0x99	NtSetContextThread
NtQueryTimerResolution	0x9A	NtSetDefaultHardErrorPort
NtQueryValueKey	0x9B	NtSetDefaultLocale
NtQueryVirtualMemory	0x9C	NtSetEaFile
NtQueryVolumelnformationFile	0x9D	NtSetEvent
NtQueueApcThread	0x9E	NtSetHighEventPair
NtRaiseException	0x9F	NtSetHighWaitLowEventPair
NtRaiseHardError	0xA0	NtSetHighWaitLowThread
NtReadFile	0xA1	NtSetlnformationFile
NtReadFileScatter	0xA2	NtSetlnformationKey
NtReadRequestData	0xA3	NtSetlnformationObject
NtReadVirtualMemory	0xA4	NtSetlnformationProcess
NtRegisterThreadTerminatePort	0xA5	NtSetlnformationThread
NtReleaseMutant	0xA6	NtSetlnformationToken
NtReleaseSemaphore	0xA7	NtSetlntervalProfile

NtRemoveloCompletion	0xA8	NtSetLdtEntries
NtReplaceKey	0xA9	NtSetLowEventPair
NtReplyPort	0xAA	NtSetLowWaitHighEventPair
NtReplyWaitReceivePort	0xAB	NtSetLowWaitHighThread
NtReplyWaitReceivePortEx	0xAC	NtSetSecurity Object
NtReplyWaitRepiyPort	0xAD	NtSetSystemEnvironmentValue
NtRequestDeviceWakeup	0xAE	NtSetSystemlnformation
NtRequestPort	0xAF	NtSetSystemPowerState
NtRequestWaitReplyPort	0xB0	NtSetSystemTime
NtRequestWakeupLatency	0xB1	NtSetTimer
NtResetEvent	0xB2	NtSetTimerResolution
NtResetWriteWatch	0xB3	NtSetValueKey
NtRestoreKey	0xB4	NtSetVolumelnformationFile
NtResumeThread	0xB5	NtShutdownSystem
NtSaveKey	0xB6	NtSignalAndWaitForSingleObject
NtSaveMergedKeys	0xB7	NtStartProfile
NtSecureConnectPort	0xB8	NtStopProfile
NtSetloCompletion	0xB9	NtSuspendThread
NtSetContextThread	0xBA	NtSystemDebugControl
NtSetDefaultHardErrorPort	0xBB	NtTerminateProcess
NtSetDefaultLocale	0xBC	NtTerminateThread
NtSetDefaultUILanguage	0xBD	NtTestAlert
NtSetEaFile	0xBE	NtUnloadDriver
NtSetEvent	0xBF	NtUnloadKey
NtSetHighEventPair	0xC0	NtUnlockFile
NtSetHighWaitLowEventPair	0xC1	NtUnlockVirtualMemory
NtSetlnformationFile	0xC2	NtUnmapViewOfSection

NtSetlnformationJobObject	0xC3	NtVdmControl
NtSetlnformationKey	0xC4	NtWaitForMultipleObjects
NtSetlnformationObject	0xC5	NtWaitForSingleObject
NtSetlnformationProcess	0xC6	NtWaitHighEventPair
NtSetlnformationThread	0xC7	NtWaitLowEventPair
NtSetlnformationToken	0xC8	NtWriteFile
NtSetlntervalProfile	0xC9	NtWriteFileGather
NtSetLdtEntries	0xCA	NtWriteRequestData
NtSetLowEventPair	0xCB	NtWriteVirtualMemory
NtSetLowWaitHighEventPair	0xCC	NtCreateChannel
NtSetQuotalnformationFile	0xCD	NtListenChannel
NtSetSecurity 0 b j ect	0xCE	NtOpenChannel
NtSetSystemEnvironment Value	0xCF	NtReplyWaitSendChannel
NtSetSystemlnformation	0xD0	NtSendWaitReplyChannel
NtSetSystemPowerSrate	0xD1	NtSetContextChannel
NtSetSystemTime	0xD2	NtYieldExecution
NtSetThreadExecutionState	0xD3	N/A
NtSetTimer	0xD4	N/A
NtSetTimerResolution	0xD5	N/A
NtSetUuidSeed	0xD6	N/A
NtSetValueKey	0xD7	N/A
NtSetVolumelnformationFile	0xD8	N/A
NtShutdownSystem	0xD9	N/A
NtSignalAndWaitForSingleObject	0xDA	N/A
NtStartProfile	0xDB	N/A
NtStopProfile	0xDC	N/A
NtSuspendThread	0xDD	N/A
NtSystemDebugControl	0xDE	N/A
NtTerminateJobObject	0xDF	N/A

NtTerminateProcess	0xE0	N/A
NtTerminateThread	0xE1	N/A
NtTestAlert	0xE2	N/A
NtUnloadDriver	0xE3	N/A
NtUnloadKey	0xE4	N/A
NtUnlockFile	0xE5	N/A
NtUnlockVirtualMemory	0xE6	N/A
NtUnmapViewOfSection	0xE7	N/A
NtVdmControl	0xE8	N/A
NtWaitForMultipleObjects	0xE9	N/A
NtWaitForSingleObject	0xEA	N/A
NtWaitHighEventPair	0xEB	N/A
NtWaitLowEventPair	0xEC	N/A
NtWriteFile	0xED	N/A
NtWriteFileGather	0xEE	N/A
NtWriteRequestData	0xEF	N/A
NtWriteVirtualMemory	0xF0	N/A
NtCreateChannel	0xF1	N/A
NtListenChannel	0xF2	N/A
NtOpenChannel	0xF3	N/A
NtReplyWaitSendChannel	0xF4	N/A
NtSendWaitReplyChannel	0xF5	N/A
NtSetContextChannel	0xF6	N/A
NtYieldExecution	0xF7	N/A

Russinoich 和 Cogewell 采用的最重要的一步是：编写一个内核模式的设备驱动程序来安装和维护 Native API Hook。因为，用户模式下的模块没有修改系统服务描述符表的权限。就像第四章中的 Spy 驱动程序，这是一种多少有些特殊的驱动程序，因为它不处理通常的 I/O 请求。它只是导出一个简单的设备 I/O 控制（IOCTL）接口，以让用户模式下的代码访问它收集到的数据。该驱动程序的主要任务是修改 KiServiceTable、拦截并记录所选的

Windows 2000 Native API 调用。尽管这种方法很简单而且优雅，它还是有些让人担心。它的简单使我想起了在 DOS 时代，hook 一个系统服务只需要简单的修改处理器的中断向量表中的指针。任何知道如何编写基本的 Windows 2000 内核驱动程序的人都可以 hook 任意的 NT 系统服务而不需要而外的努力。

Russinovich 和 Cogswell 使用他们的技术开发了一个非常有用的 Windows NT 注册表监视器。当使用他们的技术来完成其他“间谍”任务时，我很快就变得烦躁起来，这是因为我需要为我要监控的每个 API 函数都编写一个独立的 hook API 函数。为了避免编写大量的代码，我打算找出一种方法来强迫所有我感兴趣的 API 函数进入同一个 hook 函数中。这个任务花费了我大量的时间，并给我展示了多种多样的蓝屏。不过，最终的结果是我得到了一个通用的解决方案，只需花费很少的努力，我就能 hook 不同的 API 函数集合。

5.1.2、汇编语言的救援行动

通用解决方案的主要障碍是 C 语言的典型参数传递机制。就像你知道的，C 通常在调用函数的入口点之前会将函数参数传递到 CPU 堆栈中。根据函数需要的参数数量，参数堆栈的大小将有很大的差别。Windows 2000 的 248 个 Native API 函数需要的参数堆栈的大小位于 0 到 68 字节。这使得编写一个唯一的 hook 函数变得非常困难。微软的 Visual C/C++提供了一个完整的汇编（ASM）编译器，该编译器可处理复杂度适中的代码。具有讽刺意味的是，在我的解决方案中所使用的汇编语言的优点正是通常被认为是其最大缺点的特性：汇编语言不提供严格的类型检查机制。只要字节数正确就一切 OK 了，你可以在任何寄存器中存储几乎所有的东西，而且你可以调用任何地址，而不需要关心当前堆栈的内容是什么。尽管这在应用程序开发中是一种很危险的特性，但这确实最容易获取的：在汇编语言中，很容易以不同的参数堆栈调用同一个普通的入口点，稍后将介绍的 API hook Dispatcher 将采用这一特性。

通过将汇编代码放入以关键字__asm 标记的分隔块中就可调用 Microsoft Visual C/C++嵌入式汇编程序。嵌入式汇编缺少宏定义以及 Microsoft’s big Macro Assembler（MASM）的评估能力，但这些并没有严重的限制它的可用性。嵌入式汇编的最佳特性是：它可以访问所有的 C 变量和类型定义，因此很容易混合 C 和 ASM 代码。不过，当在 C 函数中包含有 ASM 代码时，就必须遵守 C 编译器的某些重要的基本约定，以避免和 C 代码的冲突：

- C 函数调用者假定 CPU 寄存器 EBP、EBX、ESI 和 EDI 已经被保存了。

- 如果在单一函数中，将 ASM 代码和 C 代码混合在一起，则需要小心的保存 C 代码可能保存在寄存器中的中间值。总是保存和恢复在__asm 语句中使用的所有寄存器。
- 8 位的函数结果（CHAR，BYTE 等）由寄存器 AL 返回。
- 16 位的函数结果（SHORT，WORD 等）由寄存器 AX 返回。
- 32 位的函数结果（INT，LONG，DWORD 等）由寄存器 EAX 返回。
- 64 位的函数结果（__int64，LONGLONG，DWORDLONG 等）由寄存器对 EDX：EAX 返回。寄存器 EAX 包含 0 到 31 位，EDX 保存 32 到 63 位。
- 有确定参数的函数通常按照__stdcall 约定进行参数的传递。从调用者的角度来看，这意味着在函数调用之前参数必须以相反的顺序压入堆栈中，被调用的函数负责在返回前从堆栈中移除它们。从被调用的函数的角度来看，这意味着堆栈指针 ESP 指向调用者的返回地址，该地址紧随最后一个参数（按照原始顺序）。（**译注：**这意味着，最先被压入堆栈的是函数的返回地址）参数的原始顺序被保留下来，因为堆栈是向下增长的，从高位线性地址到低位线性地址。因此，调用者压入堆栈的最后一个参数（即，参数#1）将是由 ESP 指向的数组中的第一个参数。
- 某些有确定参数的 API 函数，如著名的 C 运行时库函数（由 ntdll.dll 和 ntoskrnl.exe 导出），通常使用__cdecl 调用约定，该约定采用与__stdcall 相同的参数顺序，但强制调用者清理参数堆栈。
- 由__fastcall 修饰的函数声明，则希望前两个参数位于 CPU 寄存器 ECX 和 EDX 中。如果还需要更多的参数，它们将按照相反的顺序传入堆栈，最后由被调用者清理堆栈，这和__stdcall 相同。

```
; this is the function's prologue
push   ebp                    ; save current value ebp
mov    ebp, esp               ; set stack frame base address
sub    esp, SizeOfLocalStorage   ; create local storage area

; this is the function's epilogue
mov    esp, ebp               ; destroy local storage area
pop    ebp                    ; restore value of ebp
ret
```

列表 5-2. 堆栈帧，序言和尾声

- 很多 C 编译器在进入函数后，会立即针对函数参数构建一个堆栈帧，这需要使用 CPU 的基地址指针寄存器 EBP。**列表 5-2** 给出了此代码，这通常被称为函数的“序言”和“尾声”。有些编译器采用更简洁的 i386 的 ENTER 和 LEAVE 操作符，在“序言被执行后，堆栈将如**图 5-3** 所示。EBP 寄存器作为一分割点将函数的参数堆栈划分为两部分：（1）局部存储区域，该区域中包含所有定义于函数范围内的局部变量(2)调用者堆栈，其中保存有 EBP 的备份和返回地址。注意，微软的 Visual C/C++ 的最新版中默认不使用堆栈帧。替代的是，代码通过 ESP 寄存器访问堆栈中的值，不过这需要指定变量相对于当前栈顶的偏移量。这种类型的代码非常难以阅读，因为每个 PUSH 和 POP 指令都会影响 ESP 的值和所有参数的偏移量。在此种情况下不再需要 EBP，它将作为一个附加的通用寄存器。
- 在访问 C 变量时必须非常小心。经常出现在嵌入式 ASM 中的 bug 是：你将一个变量的地址而不是它的值加载到了寄存器中。使用 ptr 和 offset 地址操作符存在潜在的二义性。例如，指令：mov eax，dword ptr SomeVariable 将加载 DWORD 类型的 SomeVariable 变量的值到 EAX 寄存器，但是，mov eax，offset SomeVariable 将加载它的线性地址到 EAX 中。

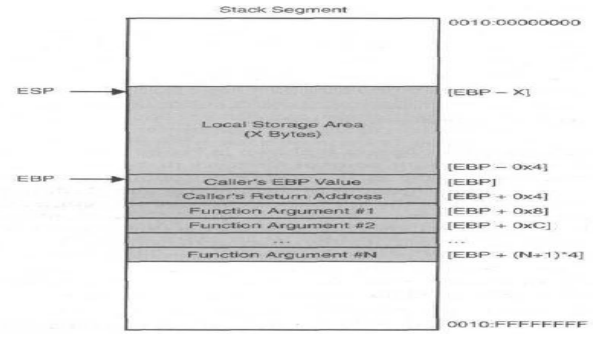


图 5-3. 堆栈帧的典型布局

5.1.3、Hook 分派程序(Hook Dispatcher)

这部分的代码将较难理解。编写它们花费了我很多时间，而且在这一过程中我还欣赏了无数的蓝屏。我最初的方法是提供一个完全用汇编语言编写的模块。不过，这个方法在链接阶时带来了很大的麻烦，因此，我改为在 C 模块中使用嵌入式汇编。为了避免创建另一个内核模式的驱动程序，我决定将 hook 代码整合到 Spy 设备驱动程序中。还记得在**表 4-2** 底部

列出的形如 SPY_IO_HOOK_*的 IOCTL 函数吗？现在我们将和它们来一次亲密接触。后面的示列代码来自 w2k_spy.c 和 w2k_spy.h，可以在随书 CD 的\src\w2k_spy 中找到它们。

列表 5-3 的核心部分是 Native API Hook 机制的实现代码。该列表开始处是一对常量和结构体定义，后面的 aSpyHooks[]需要它们。紧随这个数组的是一个宏，该宏实际上是三行嵌入式汇编语句，这三行汇编语句非常重要，稍后我将介绍它们。**列表 5-3** 的最后一部分用来建立 SpyHookInitializeEx()函数。猛地一看，这个函数的功能似乎很难理解。该函数组合了一下两个功能：

1. SpyHookInitializeEx()的表面部分包括一段用来设置 aSpyHooks[]数组的 C 代码，这部分代码用 Spy 设备的 Hook 函数指针以及与之相关联的字符串格式协议来初始化 aSpyHooks[]数组。SpyHookInitializeEx()函数可被分割为两部分：第一部分到第一个__asm 语句后的 jmp SpyHook9 指令。第二部分显然是从 ASM 标签----SpyHook9 开始，该部分位于第二个__asm 语句块的最后。
2. SpyHookInitializeEx()的内部部分包括位于两块 C 代码段之间的所有代码。这部分在一开始大量使用了 SpyHook 宏，紧随其后的是一大块复杂的汇编代码。可能你已经猜到了，这些汇编代码就是前面提到的通用 Hook 例程。

```
#define SPY_CALLS             0x00000100 // max api call nesting level
#define SDT_SYMBOLS_NT4     0xD3
#define SDT_SYMBOLS_NT5     0xF8
#define SDT_SYMBOLS_MAX    SDT_SYMBOLS_NT5

// -------------------------------------------------------------

typedef struct _SPY_HOOK_ENTRY
    {
    NTPROC Handler;
    PBYTE  pbFormat;
    }
    SPY_HOOK_ENTRY, *PSPY_HOOK_ENTRY, **PPSPY_HOOK_ENTRY;
```

```
#define SPY_HOOK_ENTRY_ sizeof (SPY_HOOK_ENTRY)

// ------------------------------------------------------------

typedef struct _SPY_CALL
    {
    BOOL            fInUse;               // set if used entry
    HANDLE          hThread;              // id of calling thread
    PSPY_HOOK_ENTRY pshe;                 // associated hook entry
    PVOID           pCaller;              // caller's return address
    DWORD           dParameters;          // number of parameters
    DWORD           adParameters [1+256]; // result and parameters
    }
    SPY_CALL, *PSPY_CALL, **PPSPY_CALL;

#define SPY_CALL_ sizeof (SPY_CALL)

// ------------------------------------------------------------

SPY_HOOK_ENTRY aSpyHooks [SDT_SYMBOLS_MAX];

// ------------------------------------------------------------
// The SpyHook macro defines a hook entry point in inline assembly
// language. The common entry point SpyHook2 is entered by a call
// instruction, allowing the hook to be identified by its return
// address on the stack. The call is executed through a register to
// remove any degrees of freedom from the encoding of the call.

#define SpyHook                               \
```

```
        __asm   push    eax                 \
        __asm   mov     eax, offset SpyHook2 \
        __asm   call    eax


// ------------------------------------------------------------
// The SpyHookInitializeEx() function initializes the aSpyHooks[]
// array with the hook entry points and format strings. It also
// hosts the hook entry points and the hook dispatcher.




// ------------------------------------------------------------
// The SpyHookInitializeEx() function initializes the aSpyHooks[]
// array with the hook entry points and format strings. It also
// hosts the hook entry points and the hook dispatcher.

void SpyHookInitializeEx (PPBYTE ppbSymbols,
                          PPBYTE ppbFormats)
    {
    DWORD dHooks1, dHooks2, i, j, n;


    __asm
        {
        jmp     SpyHook9
        ALIGN   8
SpyHook1:       ; start of hook entry point section
        }


// the number of entry points defined in this section
```

```
// must be equal to SDT_SYMBOLS_MAX (i.e. 0xF8)


SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //08
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //10
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //18
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //20
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //28
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //30
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //38
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //40
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //48
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //50
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //58
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //60
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //68
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //70
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //78
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //80
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //88
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //90
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //98
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //A0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //A8
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //B0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //B8
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //C0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //C8
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //D0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //D8
```

```
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //E0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //E8
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //F0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //F8


    __asm
        {
SpyHook2:       ; end of hook entry point section
        pop     eax                     ; get stub return address
        pushfd
        push    ebx
        push    ecx
        push    edx
        push    ebp
        push    esi
        push    edi
        sub     eax, offset SpyHook1    ; compute entry point index
        mov     ecx, SDT_SYMBOLS_MAX
        mul     ecx
        mov     ecx, offset SpyHook2
        sub     ecx, offset SpyHook1
        div     ecx
        dec     eax
        mov     ecx, gfSpyHookPause     ; test pause flag
        add     ecx, -1
        sbb     ecx, ecx
        not     ecx
        lea     edx, [aSpyHooks + eax * SIZE SPY_HOOK_ENTRY]
        test    ecx, [edx.pbFormat]     ; format string == NULL?
```

```
        jz      SpyHook5
        push    eax
        push    edx
        call    PsGetCurrentThreadId    ; get thread id
        mov     ebx, eax
        pop     edx
        pop     eax
        cmp     ebx, ghSpyHookThread    ; ignore hook installer
        jz      SpyHook5
        mov     edi, gpDeviceContext
        lea     edi, [edi.SpyCalls]     ; get call context array
        mov     esi, SPY_CALLS          ; get number of entries
SpyHook3:
        mov     ecx, 1                  ; set in-use flag
        xchg    ecx, [edi.fInUse]
        jecxz   SpyHook4                ; unused entry found
        add     edi, SIZE SPY_CALL      ; try next entry
        dec     esi
        jnz     SpyHook3
        mov     edi, gpDeviceContext
        inc     [edi.dMisses]           ; count misses
        jmp     SpyHook5                ; array overflow
SpyHook4:
        mov     esi, gpDeviceContext
        inc     [esi.dLevel]            ; set nesting level
        mov     [edi.hThread], ebx      ; save thread id
        mov     [edi.pshe], edx         ; save PSPY_HOOK_ENTRY
        mov     ecx, offset SpyHook6    ; set new return address
        xchg    ecx, [esp+20h]
```

```
        mov     [edi.pCaller], ecx      ; save old return address
        mov     ecx, KeServiceDescriptorTable
        mov     ecx, [ecx].ntoskrnl.ArgumentTable
        movzx   ecx, byte ptr [ecx+eax] ; get argument stack size
        shr     ecx, 2
        inc     ecx                     ; add 1 for result slot
        mov     [edi.dParameters], ecx  ; save number of parameters
        lea     edi, [edi.adParameters]
        xor     eax, eax                ; initialize result slot
        stosd
        dec     ecx
        jz      SpyHook5                ; no arguments
        lea     esi, [esp+24h]          ; save argument stack
        rep     movsd
SpyHook5:
        mov     eax, [edx.Handler]      ; get original handler
        pop     edi
        pop     esi
        pop     ebp
        pop     edx
        pop     ecx
        pop     ebx
        popfd
        xchg    eax, [esp]              ; restore eax and...
        ret                             ; ...jump to handler
SpyHook6:
        push    eax
        pushfd
        push    ebx
```

```
        push    ecx
        push    edx
        push    ebp
        push    esi
        push    edi
        push    eax
        call    PsGetCurrentThreadId    ; get thread id
        mov     ebx, eax
        pop     eax
        mov     edi, gpDeviceContext
        lea     edi, [edi.SpyCalls]     ; get call context array
        mov     esi, SPY_CALLS          ; get number of entries
SpyHook7:
        cmp     ebx, [edi.hThread]      ; find matching thread id
        jz      SpyHook8
        add     edi, SIZE SPY_CALL      ; try next entry
        dec     esi
        jnz     SpyHook7
        push    ebx                     ; entry not found ?!?
        call    KeBugCheck
SpyHook8:
        push    edi                     ; save SPY_CALL pointer
        mov     [edi.adParameters], eax ; store NTSTATUS
        push    edi
        call    SpyHookProtocol
        pop     edi                     ; restore SPY_CALL pointer
        mov     eax, [edi.pCaller]
        mov     [edi.hThread], 0        ; clear thread id
        mov     esi, gpDeviceContext
```

```
        dec     [esi.dLevel]            ; reset nesting level
        dec     [edi.fInUse]            ; clear in-use flag
        pop     edi
        pop     esi
        pop     ebp
        pop     edx
        pop     ecx
        pop     ebx
        popfd
        xchg    eax, [esp]              ; restore eax and...
        ret                             ; ...return to caller
SpyHook9:
        mov     dHooks1, offset SpyHook1
        mov     dHooks2, offset SpyHook2
        }
    n = (dHooks2 - dHooks1) / SDT_SYMBOLS_MAX;

    for (i = j = 0; i < SDT_SYMBOLS_MAX; i++, dHooks1 += n)
        {
        if ((ppbSymbols != NULL) && (ppbFormats != NULL) &&
            (ppbSymbols [j] != NULL))
            {
            aSpyHooks [i].Handler  = (NTPROC) dHooks1;
            aSpyHooks [i].pbFormat =
                SpySearchFormat (ppbSymbols [j++], ppbFormats);
            }
        else
            {
            aSpyHooks [i].Handler  = NULL;
```

```
            aSpyHooks [i].pbFormat = NULL;
            }
        }
    return;
    }
```

列表 5-3. Hook Dispatcher 的实现方式

SpyHook 宏实际是什么呢？在 SpyHookInitializeEx()函数中,这个宏被重复了多大 248（0xF8）次，这正好是 Windows 2000 Native API 函数的数目。在**列表 5-3** 的顶部，这个数目被定义为 SDT_SYMBOLS_MAX 常量，该宏可以使 SDT_SYMBOLS_NT4 或 SDT_SYMBOLS_NT5。因为我打算支持 Windows NT 4.0。回到 SpyHook 宏上来：该宏调用的汇编语句在**列表 5-4** 中给出了。每个 SpyHook 都产生同样的三行代码：

1. 第一行，将当前 EAX 寄存器的内容保存到堆栈中。
2. 第二行，将 SpyHook2 的线性地址保存到 EAX 中。
3. 第三行，调用 EAX 中的地址（即：call eax）。

你可能会惊讶：当这个 CALL 返回时会发生什么。接下来的一组 SpyHook 代码会被调用吗？不----这个 CALL 并不支持返回，因为在到达 SpyHook2 之后，这个 CALL 的返回地址就会被立即从堆栈中移出，**列表 5-4** 最后的 POP EAX 指令可以证明这一点。这种看上去毫无疑义的代码在古老的汇编程序设计时代曾被广泛的讨论的一种技巧,就像今天我们讨论面向对象的程序设计一样。当 ASM 老大级人物需要构建一个数组，而此数组的每一项都有类似的进入点，但却需要被分派到独立的函数时，就会采用这种技巧。对所有进入点使用几乎相同的代码可以保证它们之间有相等的间隔，因此客户端就可以很容易的通过 CALL 指令的返回地址计算出进入点的在数组中的索引值，数组的基地址和大小以及数组中共有多少项

```
SpyHook1:
       push    eax
       mov    eax,  offset  SpyHook2
       call     eax
       push    eax
       mov    eax,  offset  SpyHook2
       call     eax
```

```
;244 boring repetitions cimitted
       push    eax
       mov    eax,  offset  SpyHook2
       call     eax
       push    eax
       mov    eax,  offset  SpyHook2
       call     eax
SpyHook2:
       pop    eax
```

列表 5-4. 扩充 SpyHook 宏调用

例如，**列表 5-4** 中第一个 CALL EAX 指令的返回地址是其下一个语句的地址。通常，第 N 个 CALL EAX 指令的返回地址是第 N+1 个语句的地址，但最后一个除外，最后这个将返回 SpyHook2。因此，从 0 开始的所有进入点的索引可以由**图 5-4** 中的通用公式计算出来。这三条规则中的潜在规则是：SDT_SYMBOLS_MAX 进入点符合内存块 SpyHook2---SpyHook1。那么有多少个进入点符合 ReturnAddress---SpyHook1 呢？因为计算结果是位于 0 到 SDT_SYMBOLS_MAX 中的某一个数值，所以，肯定要使用该数值来获取一个从 0 开始的索引。

$$\text{Index} = \frac{(\text{ReturnAddress} - \text{SpyHook1}) * \text{SDT\_SYMBOLS\_MAX}}{\text{SpyHook2} - \text{SpyHook1}} - 1$$

图 5-4. 通过 Hook 进入点的返回地址确定一个 Hook 进入点

图 5-4 所示公式的实现方式可以在**列表 5-3** 中找到，在汇编标签 SpyHook2 的右边。在**图 5-5** 的左下角也给出了该公式的实现代码，它展示了 Hook Dispatcher 机制的基本原理。注意，i386 的 mul 指令会在 EDX:EAX 寄存器中产生一个 64 位的结果值，这正是其后的 div 指令所期望的，因此，这里没有整数溢出的危险。在**图 5-5** 的左上角，是对 KiServiceTable 的描述，该表将被 SpyHook 宏生成的进入点地址修改。在图的中部展示了展开后的宏代码（来自**列表 5-4** 中）。进入点的线性地址位于图的右手边。为了完全一致，每个进入点的大小都是 8 字节，因此，通过将 KiServiceTable 中每个函数的索引值乘以 8，然后再将乘积加上 SpyHook1 的地址就可得出进入点的地址。

事实上，每个进入点并不都是纯粹的 8 字节长。我花费了大量的时间来寻找最佳的 hook 函数的实现方式。尽管按照 32 位边界对齐代码并不是必须的，但这从来都不是个坏主意，

因为这会提高性能。当然，能提升的性能十分有限。你或许会奇怪：为什么我要通过 EAX 寄存器间接的调用 SpyHook2，而不是直接使用 CALL SpyHook2 指令，这不是更高效吗？是的！不过，问题是 i386 的 CALL（还有 jmp）指令可以有多种实现方式，而且都具有相同的效果，但是产生的指令大小却不相同。请参考：Intel’s Instruction Set Reference of the Pentium CPU family（Intel 199c）。因为最终的实现方式要由编译器/汇编器来确定，这不能保证所有的进入点都会有相同的编码。换句话说，MOV EAX 和一个 32 位常量操作数总是以相同的方式编码，同样的，这也适用于 CALL EAX 指令。

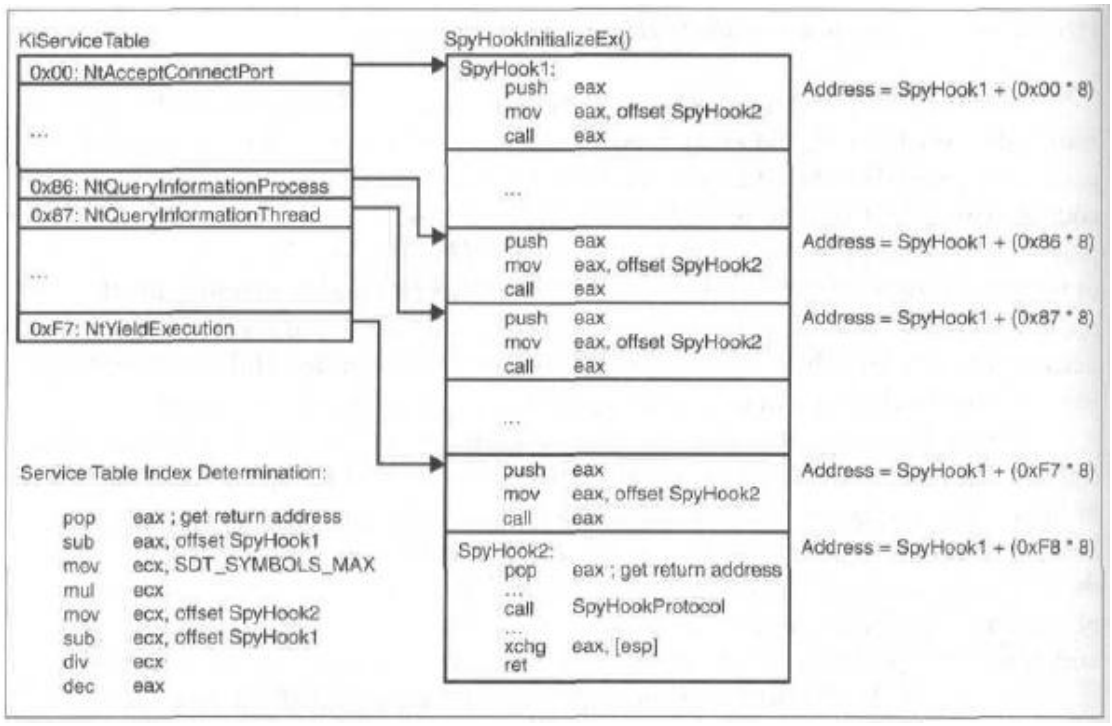


图 5-5. Hook Dispatcher 的功能原理

列表 5-3 中还有一点需要澄清。让我们从 SpyHook9 标签后的最后一快 C 代码段开始。紧随 SpyHook9 之后的汇编代码将 SpyHook1 和 SpyHook2 的线性地址保存在 dHook1 和 dHook2 变量中。接下来，变量 n 被设为每个进入点的大小（由进入点数组的大小除以进入点的个数而得出）。当然，这个值将是 8。**列表 5-3** 的剩余部分是一个循环语句，用来初始化全局数组 aSpyHooks[]中的所有项。这个数组所包含的 SPY_HOOK_ENTRY 结构定义于列**表 5-3** 的顶部，该数组中的每一项都对应一个 Native API 函数。要理解该结构中的 Handler 和 pbFormat 成员是如何被设置的，就必须进一步了解传递给 SpyHookInitializeEx()的 ppbSymbols 和 ppbFormats 参数，**列表 5-5** 给出了外包函数 SpyHookInitialize()，该函数会选择适合当前 OS 版本的参数来调用 SpyHookInitializeEx()。前面已经提示过，我使用的代码不直接测试 OS 版本或 Build Number，而是用常量 SPY_SYMBOLS_NT4、SPY_SYMBOLS_NT5 和 SDT 中与

ntoskrnl.exe 相关的 ServiceLimit 成员的值进行比较。如果没有一个匹配，Spy 设备将把 aSpyHooks[]数组内容全部初始化为 NULL，从而有效的禁止 Native API Hook 机制。

```
BOOL SpyHookInitialize (void)
    {
    BOOL fOk = TRUE;

    switch (KeServiceDescriptorTable->ntoskrnl.ServiceLimit)
        {
        case SDT_SYMBOLS_NT4:
            {
            SpyHookInitializeEx (apbSdtSymbolsNT4, apbSdtFormats);
            break;
            }
        case SDT_SYMBOLS_NT5:
            {
            SpyHookInitializeEx (apbSdtSymbolsNT5, apbSdtFormats);
            break;
            }
        default:
            {
            SpyHookInitializeEx (NULL, NULL);
            fOk = FALSE;
            break;
            }
        }
    return fOk;
    }
```

列表 5-5. SpyHookInitialize()选择匹配当前 OS 版本的符号表

将全局数组：apbSdtSymbolsNT4[]和 apbSdtSymbolsNT5[]传递给 SpyHookInitializeEx()函数作为其第一个参数 ppbSymbols，这两个数组只是简单的字符串数组，包含 Windows NT 4.0 和 Windows 2000 的所有 Native API 函数的名称，按照它们在 KiServiceTable 中的索引顺序来存储,最后以 NULL 结束。**列表 5-6** 给出了 apbStdFormats[]字符串数组。这个格式字符串列表也是 hook 机制中很重要的一部分，因为它确定了记录了那个 Native API 调用，以及每个记录项的格式。显然，这些字符串的结构借鉴了 C 运行时库中的 printf()函数，但针对 Native API 经常使用的数据类型进行了修改。**表 5-2** 列出了所有可被 API Logger 识别的格式化 ID。

```
PBYTE apbSdtFormats [] =
    {
    "%s=NtCancelIoFile(%!,%i)",
    "%s=NtClose(%-)",
    "%s=NtCreateFile(%+,%n,%o,%i,%l,%n,%n,%n,%n,%p,%n)",
    "%s=NtCreateKey(%+,%n,%o,%n,%u,%n,%d)",
    "%s=NtDeleteFile(%o)",
    "%s=NtDeleteKey(%-)",
    "%s=NtDeleteValueKey(%!,%u)",
    "%s=NtDeviceIoControlFile(%!,%p,%p,%p,%i,%n,%p,%n,%p,%n)",
    "%s=NtEnumerateKey(%!,%n,%n,%p,%n,%d)",
    "%s=NtEnumerateValueKey(%!,%n,%n,%p,%n,%d)",
    "%s=NtFlushBuffersFile(%!,%i)",
    "%s=NtFlushKey(%!)",
    "%s=NtFsControlFile(%!,%p,%p,%p,%i,%n,%p,%n,%p,%n)",
    "%s=NtLoadKey(%o,%o)",
    "%s=NtLoadKey2(%o,%o,%n)",
    "%s=NtNotifyChangeKey(%!,%p,%p,%p,%i,%n,%b,%p,%n,%b)",
    "%s=NtNotifyChangeMultipleKeys(%!,%n,%o,%p,%p,%p,%i,%n,%b,%p,%n,%b)",
    "%s=NtOpenFile(%+,%n,%o,%i,%n,%n)",
    "%s=NtOpenKey(%+,%n,%o)",
```

```
    "%s=NtOpenProcess(%+,%n,%o,%c)",
    "%s=NtOpenThread(%+,%n,%o,%c)",
    "%s=NtQueryDirectoryFile(%!,%p,%p,%p,%i,%p,%n,%n,%b,%u,%b)",
    "%s=NtQueryInformationFile(%!,%i,%p,%n,%n)",
    "%s=NtQueryInformationProcess(%!,%n,%p,%n,%d)",
    "%s=NtQueryInformationThread(%!,%n,%p,%n,%d)",
    "%s=NtQueryKey(%!,%n,%p,%n,%d)",
    "%s=NtQueryMultipleValueKey(%!,%p,%n,%p,%d,%d)",
    "%s=NtQueryOpenSubKeys(%o,%d)",
    "%s=NtQuerySystemInformation(%n,%p,%n,%d)",
    "%s=NtQuerySystemTime(%l)",
    "%s=NtQueryValueKey(%!,%u,%n,%p,%n,%d)",
    "%s=NtQueryVolumeInformationFile(%!,%i,%p,%n,%n)",
    "%s=NtReadFile(%!,%p,%p,%p,%i,%p,%n,%l,%d)",
    "%s=NtReplaceKey(%o,%!,%o)",
    "%s=NtSetInformationKey(%!,%n,%p,%n)",
    "%s=NtSetInformationFile(%!,%i,%p,%n,%n)",
    "%s=NtSetInformationProcess(%!,%n,%p,%n)",
    "%s=NtSetInformationThread(%!,%n,%p,%n)",
    "%s=NtSetSystemInformation(%n,%p,%n)",
    "%s=NtSetSystemTime(%l,%l)",
    "%s=NtSetValueKey(%!,%u,%n,%n,%p,%n)",
    "%s=NtSetVolumeInformationFile(%!,%i,%p,%n,%n)",
    "%s=NtUnloadKey(%o)",
    "%s=NtWriteFile(%!,%p,%p,%p,%i,%p,%n,%l,%d)",
    NULL
    };
```

列表 5-6. Native API Logger 使用的格式化字符串

这里要特别提出的是：每个格式字符串要求必须提供函数名的正确拼写。SpyHookInitializeEx()遍历它接受到的 Native API 符号列表（通过 ppbSymbols 参数），并试图从 ppbFormats 列表中找出与函数名匹配的格式字符串。由帮助函数 SpySearchFormat()来进行比较工作，**列表 5-3** 底部的 if 语句中调用了该函数。因为要执行大量的字符串查找操作，我使用了一个高度优化的查找引擎，该引擎基于“Shift/And”搜索算法。如果你想更多的学习它的实现方式，请察看随书 CD 的\src\w2k_spy\w2k_spy.c 源文件中的 SpySearch*()函数。当 SpyHookInitializeEx()推出循环后，aSpyHooks[]中的所有 Handler 成员都将指向适当的 Hook 进入点，pbFormat 成员提供与之匹配的格式字符串。对于 Windows NT 4.0，所有索引值在 0xD3---0xF8 的数组成员都将被设为 NULL，因为在 NT4 中，它们并没有被定义。

表 5-2. 可识别的格式控制 ID

ID	名　称	描　　述
%+	句柄（登记）	将句柄和对象名写入日志，并将其加入句柄表。
%!	句柄（检索）	将句柄写入日志，并从句柄表中检索其对应的对象名。
%-	句柄（撤销登记）	将句柄和对象名写入日志，并将其从句柄表移除
%a	ANSI 字符串	将一个由 8 位 ANSI 字符构成的字符串写入日志
%b	BOOLEAN	将一个 8 位的逻辑值写入日志
%c	CLIENT_ID*	将 CLIENT_ID 结构的成员写入日志
%d	DWORD *	将该 DWORD 所指变量的值写入日志
%i	IO_STATUS_BLOCK *	将 IO_STATUS_BLOCK 结构的成员写入日志
%l	LARGE_INTEGER *	将一个 LARGE_INTEGER 的值写入日志
%n	数值(DWORD)	将一个 32 位无符号数写入日志
%o	OBJECT_ATTRIBUTES *	将对象的 ObjectName 写入日志
%p	指针	将指针的目标地址写入日志
%s	状态(NTSTATUS)	将 NT 状态代码写入日志
%u	UNICODE_STRING *	将 UNICOD_STRING 结构的 Buffer 成员写入日志
%w	宽字符串	将一个由 16 位字符构成的字符串写入日志
%%	百分号转义符	将一个“%”号写入日志

本书设计的 hook 机制的最大特色就是它是完全数据驱动的（data-driven）。只需简单的增加一个新的 API 符号表，该 hook dispatcher 就可适应 Windows 2000 的新版本。而且，通过向 apdSdtFormats[]数组中加入新的 API 函数的格式化字符串就可在任何时候记录对这些附加的 API 函数的调用。这并不需要编写任何附加的代码---API Spy 的动作可完全由一组字符串来确定！不过，在定义新的格式化字符串是必须要小心，因为 w2k_spy.sys 是运行于内核模式的驱动程序。因为在这一系统层次上，系统不能温和的处理发生错误。给 Win32 API 函数提供了一个无效的参数并不是问题-----你会收到一个错误提示窗口，同时程序会被系统自动终止。在内核模式下，一个微小的访问违规都会引发系统蓝屏。因此，一定要小心。在需要的地方如果没有出现一个正确的格式化控制 ID 或缺失了这一 ID 都会使你的系统彻底崩溃。即使一个简单的字符串有时都是致命的！

现在仅剩 SpyHookInitializeEx()中的那一大块 ASM 代码还未讨论，这段代码由 SpyHook2 和 SpyHook9 标识。这段代码的一个有趣的特性是：在 SpyHookInitializeEx()被调用的时候，它们从来都不会被执行。在进入 SpyHookInitializeEx()后，函数代码将跳过这一整段代码，然后在 SpyHook9 标签处开始恢复执行，此处包含 aSpyHooks[]数组的初始化代码。这一大块 ASM 代码只有通过 aSpyHooks[]数组中的 Handler 成员才能进入。稍候，我将展示这些进入点是如何连接到 SDT 的。

在设计这段 ASM 代码时，我的重要目标之一就是使其是完全非侵入式的。截获操作系统调用非常危险，因为你从来不会知道被调用的代码是否会依赖调用上下文（calling context）的某些未知特性。理论上来说，这些 ASM 代码完全符合__stdcall 约定，但仍存在出错的可能性。我不得不选择将原始的 Native API 处理例程放入几乎完全相同的环境中，这意味着这些原始函数将使用最初的参数堆栈并且可以访问所有的 CPU 寄存器，就像它们被正常调用一样。当然，必须接受由于插入 hook 所带来的最低限度的危险，否则，监控将不可能实现。在这里，有意义的改动就是维护堆栈中的返回地址。如果你翻回到**图 5-3**，你会发现在进入函数时，调用者的返回地址并不位于堆栈的顶部。SpyHookInitializeEx()中的 hook dispatcher 占用了此地址，将它自己的 SpyHook6 标签的地址写在了这里。因此，原始 Native API 处理例程将被打断，然后进入 SpyHook6 中，这样 hook dispatcher 才能检查原始 Native API 处理例程的参数和它要返回的数据。

在调用原始处理例程之前，dispatcher 将建立一个 SPY_CALL（参见**列表 5-3**）控制块，该控制块中包含它稍候将会用到的参数。其中的一些参数在正确记录 API 调用时会用到，另外一些则提供了有关调用者的信息，因此 dispatcher 可以在写完 log 后，把控制返回给调用者，就像什么都没有发生一样。Spy 设备在它的全局数据块 DEVICE_CONTEXT 中维护着一个 SPY_CALL 结构的数组，可通过全局变量 gpDeviceContext 来访问。Hook Dispatcher 通过检查 SPY_CALL 结构中的 InUse 成员来在数组中找到一个空的 SPY_CALL。Hook Dispatcher 使用 CPU 的 XCHG 指令来加载和设置该成员的值（译注：XCHG 指令可以保证此操作为原子操作）。这一点非常重要，因为当代码运行于多线程环境中时，读写全局数据时必须采取保护措施以避免条件竞争。如果在数组中找到了一个空的 SPY_CALL，dispatcher 就会将调用者的线程 ID（通过 PsGetCurrentThreadId()获取）、与当前 API 函数相关的 SPY_HOOK_ENTRY 结构的地址以及整个参数堆栈保存到该 SPY_CALL 结构中。需要复制的参数的字节数取自 KiArqumentTable 数组，该数组保存在系统的 SDT 中。如果所有的 SPY_CALL 都被使用了，原始的 API 函数处理例程将被调用而不会产生任何日志记录。

必须采用 SPY_CALL 数组是因为 Windows 2000 的多线程本性。当 Native API 函数被暂停（suspended）时，这种情况就会经常出现----此时，另一个线程将获得控制权，然后在它自己的时间片（time slice）内调用另一个 Native API 函数。这意味着 Spy 设备的 Hook Dispatcher 必须允许在任何时间和任何执行点上的重进入（reenter）。如果 Hook Dispatcher 有单一的全局 SPY_CALL 存储区域，它就可能在处于等待状态的线程使用完之前被当前运行的线程覆写（overwritten）。而这种情况正是蓝屏的最佳候选人。为了进一步了解 Native API 的嵌套，我在 Spy 的 DEVICE_CONTEXT 结构中增加了 dLevel 和 dMisses 成员。无论何时只要重进入 hook dispatcher（如，向 SPY_CALL 数组中增加一个新的 SPY_CALL）dLevel 都不会累加一个 1。如果超过了最大嵌套层数（如，SPY_CALL 数组已满），dMisses 就会累加一个 1，来标识丢失了一个日志记录。根据我的观察，在实际环境下，可以很容易的发现嵌套层达到 4。这表示即时在高负载（heavy-load）的情况下，Native API 也会被重进入，因此，我将嵌套层数的上限设为 256。

在调用原始的 API 处理例程之前，Hook Dispatcher 会保存所有的 CPU 寄存器（包括 EFLAGS），随后执行路径将导向函数的进入点。这会在**列表 5-3** 中的 SpyHook5 标签之前立即完成。此时，SpyHook6 将位于栈顶，仅随其后的是调用者的参数。一旦 API 处理例程

推出了，控制将被传回到 hook dispatcher 的 SpyHook6 标签。从此处开始执行的代码也被设计为非入侵的。此时，主要目标是允许调用者可以看到调用上下文，这和原始 API 函数建立的上下文几乎完全一致。Dispatcher 的主要问题是要能立即找到保存有当前 API 调用信息的 SPY_CALL 结构。唯一可以依赖的就是调用者的线程 ID，该 ID 保存在 SPY_CALL 结构的 hThread 成员中。因此，Dispatcher 循环遍历整个 SPY_CALL 数组以寻找匹配的线程 ID。注意，代码不会关心 fmuse 标志的值；这并不是必须的，因为数组中所有未使用的 SPY_CALL 结构的 hThread 都被设为了 0，这是系统空闲线程的 ID。循环会在到达数组结尾时终止。否则的话（译注：即没有找到匹配的线程 ID），Dispatcher 不会将控制返回给调用者，因为这样做将是致命的。在这种情况下，代码的选择余地很小，因此，它会进入 KeBugCheck()，这样做的结果当然是使系统以受控的方式终止。不过这种情况应该从来不会发生，但如果它发生了，那表示系统必然出现了很严重的错误，因此，使系统终止是最佳解决方案。

如果发现了匹配的 SPY_CALL，Hook Dispatcher 将结束它的工作。最后的动作是调用日志记录函数 SpyHookProtocol()，需要给该函数传入一个指向 SPY_CALL 结构的指针。日志记录所需的信息都保存在该结构中。当 SpyHookProtocol()返回后，Dispatcher 就释放它刚才使用的 SPY_CALL，恢复所有的 CPU 寄存器，然后返回到调用者。

5.1.4、API HOOK 协议

一个好的 API Spy 应该可以在原始函数被调用后还能察看它使用的参数，因为函数可能会通过传入的缓冲区返回附加的数据。因此，日志函数 SpyHookProtocol()在 hook 例程结束时将被调用，而此时 API 函数还未返回到调用者。在讨论它的实现秘诀之前，请先看看下面给出的两个示例性的协议（Protocol），它们会为你提供一个大概的方向。**图 5-6** 是在命令行下执行 dir c:\时产生的日志文件的快照。

请对比**图 5-6** 中列出的日志项和**列表 5-6** 给出的协议格式化字符串。在**示列 5-1** 中，NtOpenFile()和 NtClose()的格式化字符串分别对应**图 5-6** 中的第一行和第四行。它们有着惊人的相似处；每一个格式化控制 ID 都紧随在一个%号后（参考**表 5-2**），与其相关的参数项将包含在协议中。不过，协议还包含一些附加的信息，这些信息明显不属于格式字符串。稍后我将解释这种差异的原因。

示例 5-2 给出了一个协议项的一般格式。每一项包含相同个数的域，这些域采用分隔符隔开。这样分隔可以使程序很容易的解析它。这些域按照如下的一组简单的基本规则来构建：

- 所有的数字都已十六进制表示，没有 0 前缀或常见的前缀“0x”
- 函数的多个参数由逗号隔开
- 字符串参数将位于一对双引号中
- 结构体成员的值由“.”符号隔开

图 5-6. 命令 dir c:\的示列协议

```
"%s=NtOpenFile(%+,%n,%o,%i,%n, %n) "
18:sO=NtOpenFile(+46C.18,nl00001,o"\??\C:\",i0.1,n3,n4021)lBFEE5AE05B6710,278,2
"%s=NtClose(%-l)"
lB:sO=NtClose(-46C.18="\??\C:\")lBFEE5AE05B6710,278,l
```

示列 5-1. 比较格式化字符串和协议项

```
<#> : <status>=<function> (<arguments>) <time> , <thread>, <handles>
```

示列 5-2. 协议项的一般格式

- 与句柄相关的对象名称和句柄的值采用“=”进行分割。
- 日期/时间的 stamp 为 1601-01-01 至今逝去的毫秒数，其格式依赖 Windows 2000 的基本时间格式，精度可达到 1/10 毫秒。

- 线程 ID 是调用 API 函数的线程的唯一数字标识。
- 句柄计数的状态表示当前注册到 Spy 设备句柄列表中的句柄的数量。协议函数使用该列表查找与对象名称相关的句柄。

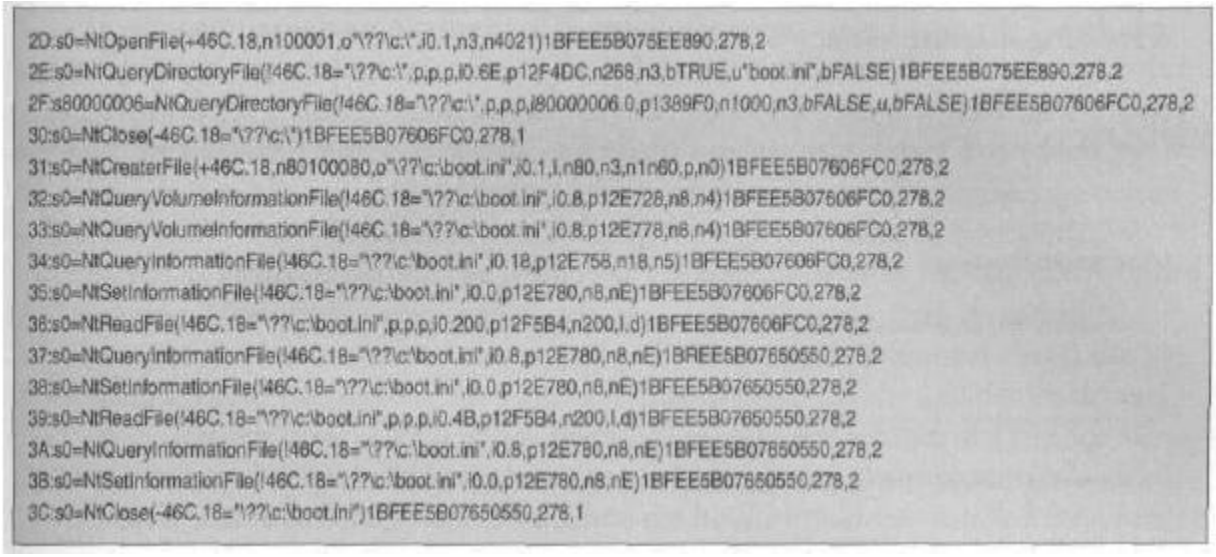


图 5-7. 命令 type c:\boot.ini 的示列协议

图 5-7 是在控制台中执行：type c:\boot.ini 命令产生的 API Spy 协议结果。下面给出日志项中的某些列的含义：

- 在 0x31 行，调用了 NtCreateFile()来打开\??\c:\boot.ini 文件。（o"\??\c:\boot.ini"）该函数返回的 NTSTATUS 的值为 0（s0），即 STATUS_SUCCESS，并分配了一个新的文件句柄，其值为 0 小 8，该句柄属于进程 0x46c（+46C.18）。因此，句柄计数从 1 增加到 2。
- 在 0x36 行，type 命令将文件\??\c:\boot.ini 的前 512 个字节（n200）读入位于线性地址 0x0012F5B4 处的缓冲区中，并把从 NtCreateFile()获取的句柄解析给 NtReadFile()函数。系统成功的返回 512 字节（io.200）。
- 在 0x39 行，将处理另一块 512 个字节的文件块。这一次，将到达文件的末尾，因此 NtReadFile()仅返回了 75 个字节（io.4B）。显然，我的 boot.ini 文件的大小为：512+75=587 字节。
- 在 0x3C 行，NtClose()成功的释放了指向\??\c:\boot.ini 的文件句柄（-46.18="\??\c:\boot.ini"），因此，句柄计数将从 2 减少为 1。

现在，你应该已经明白 Spy 协议的 API 是如何构建的了，这会帮助你掌握协议生成机制的细节，接下来我们将讨论这一机制。在前面我曾提及过，用于日志记录的主要 API 函数是 SpyHookProtocol()。**列表 5-7** 给出了该函数，它将使用 SPY_CALL 结构中的数据来为每个 API 函数生成一个协议记录并将其写入一个环形缓冲区中，这里的 SPY_CALL 结构由 Hook Dispatcher 传入。一个 Spy 设备的客户端可以通过 IOCTL 调用来读去这一协议。每个记录项都是一行文本，每行都由单个行结束符（即 C 语言中的"\n"）表示行的结束。通过使用内核的 Mutext KMUTEX kmProtcol 来实现串行读去协议缓冲区，kmProtocol 位于 Spy 设备的全局结构 DEVICE_CONTEXT 中。**列表 5-7** 中的 SpyHookWait()和 SpyHookRelease()函数用于请求和释放此 Mutext 对象。所有对协议缓冲区的访问都必须由 SpyHookWait()预处理并在结束时由 SpyHookRelease()处理，SpyHookProtocol()函数展示了这种行为。

```
NTSTATUS SpyHookWait (void)
    {
    return MUTEX_WAIT (gpDeviceContext->kmProtocol);
    }
// ------------------------------------------------------------
LONG SpyHookRelease (void)
    {
    return MUTEX_RELEASE (gpDeviceContext->kmProtocol);
    }
// ------------------------------------------------------------
// <#>:<status>=<function>(<arguments>)<time>,<thread>,<handles>
void SpyHookProtocol (PSPY_CALL psc)
    {
    LARGE_INTEGER liTime;
    PSPY_PROTOCOL psp = &gpDeviceContext->SpyProtocol;
    KeQuerySystemTime (&liTime);
    SpyHookWait ();
```

```
    if (SpyWriteFilter (psp, psc->pshe->pbFormat,
                                    psc->adParameters,
                                    psc->dParameters))
        {
        SpyWriteNumber (psp, 0, ++(psp->sh.dCalls));      // <#>:
        SpyWriteChar     (psp, 0, ':');
                                                          // <status>=
        SpyWriteFormat (psp, psc->pshe->pbFormat, //    <function>
                                    psc->adParameters);    //      (<arguments>)
        SpyWriteLarge    (psp, 0, &liTime);                    // <time>,
        SpyWriteChar     (psp, 0, ',');
        SpyWriteNumber (psp, 0, (DWORD) psc->hThread); // <thread>,
        SpyWriteChar     (psp, 0, ',');
        SpyWriteNumber (psp, 0, psp->sh.dHandles);         // <handles>
        SpyWriteChar     (psp, 0, '\n');
        }
    SpyHookRelease ();
    return;
    }
```

列表 5-7. 主要的 Hook 协议函数 SpyHookProtocol()

如果你比较一下**列表 5-7** 给出的 SpyHookProtocol()函数的主要部分和**示列 5-2** 给出的协议项的一般格式，将很容易找出那个语句生成了协议项中的哪一个域（field）。这样一来一切就很清楚了为什么**列表 5-6** 中的协议字符串没有说明整个数据项---有些独立于功能的数据将由 SpyHookProtocol()添加，而这将不需要格式字符串的帮助。SpyHookProtocl()的核心调用是 SpyWriteFormat()，该函数生成<status>=<function>[<arguments>]部分，这依赖于与要记录的当前 API 函数相关的格式字符串。请参考位于随书光盘的\src\w2k_spy 目录下的源文件 w2k_spy.c 和 w2k_spy.h，以获取 Spy 设备驱动程序中使用的 SpyWrite*()函数的更多实现信息。

请注意，这些代码稍微有些危险。这些代码编写与 1997 年是针对 Windows NT 4.0 的。在移植到 Windows 2000 之后，当 hook 工作一段较长时间后会偶尔引发蓝屏。更糟糕的是，有些特殊的操作将立即引发蓝屏，例如，在 My Favoriter 文本编辑器的 File\Open 对话框中打开我的电脑时。在分析过多过 crash dump 后，我发现是由于将 NULL 指针传递给了某些函数从而导致了系统崩溃。一但 Spy 设备试图使用这些指针中的某个来记录该指针引用的数据时，系统就会崩溃。典型的就是，指向 IO_STATUS_BLOCK 结构的指针，在 UNICODE_STRING 和 OBJECT_ATTRIBUTES 结构中存在无效的字符串指针。我还发现某些有 Buffer 成员的 UNICODE_STRING 结构没有\0 结束符。因此，我再次强调你不应该假定所有的 UNICODE_STRING 结构都以\0 结束。在不能确定时，请使用 Length 成员，它总能正确地告诉你在 Buffer 中存放的有效的字节数。

为了修正这一问题，我为所有使用客户指针的日志函数增加了指针有效性检查。在结束时，我使用第四章讨论过的 SpyMemoryTestAddress()函数来检验一个线性地址指针是否指向一个有效的页表项（PTE）。更详细的信息请参考**列表 4-22** 和**列表 4-24**。另一种可能的替代方案是使用结构化异常（__try/__except）。

5.1.5、管理句柄

要特别注意的是：SpyHookProtocol()函数仅在它内部包含 SpyWriteFilter()函数的条件语句返回 TRUE 时，才会记录 API 函数调用。这种行为是为了减少 Hook 协议中的无用信息。例如，移动鼠标将触发一系列 NtReadFile()函数调用。其他产生无用信息的源头和物理实验中的测量有些相似：比如，你在实验环境下测量某种物理效果时，测量方法本身也会对要测量的物理效果产生影响，这将导致测量结果的“失真”。这同样发生在记录 API 时。注意：NtDeviceIoControlFile()函数也包含在**列表 5-6** 中的格式化字符串数组中。不过，Spy 设备的客户端采用 Device I/O Control 调用来读取 API Hook 协议。这意味着客户端将在协议数据中发现它自己的 NtDeviceIoControlFile()调用。根据 IOCTL 事务的频率，你所期望的数据可能会淹没在它自己的函数调用记录中（**译注：**此处是指，如果客户端频繁的使用 IOCTL，那么记录下的 IOCTL 函数调用中，仅有少量是我们所期望关注的，而大部分都是客户端自己产生的），Spy 设备通过记录安装 API Hook 的那个线程的 ID，并忽略来自该线程的 API 调用来解决这一问题。

对于所有产生句柄，并且产生的句柄未被记录的 API 调用，SpyWriteFilter()都将忽略，以消除无用信息。如果 Spy 设备发现一个句柄已经关闭或已返回给系统，那么任何使用该句柄的函数序列都将被忽略。更有效的是，如果 API 调用包含由系统或其他在启动 API Hook 协议之前就存在的进程创建的句柄（这种句柄的生命期都很长），那么该 API 调用将会被禁止。当然，客户端可以通过 IOCTL 禁止或允许这种过滤。你可以使用本章稍后将介绍的一个简单的客户端程序来来测试这种过滤机制。你将会很惊讶这个简单的“噪音过滤”功能的强大。

在**列表 5-6** 中，产生句柄的函数有：NtCreateFile()、NtCreateKey()、NtOpenFile()、NtOpenKey()、NtOpenProcess()和 NtOpenThread()。所有这些函数的格式字符串中都包含一个%+控制符，该控制符用于表示**表 5-2** 中的“句柄（注册）”。对于关闭或释放句柄的函数：NtClose()和 NtDeleteKey()，在二者的格式字符串中均包含一个%-控制符，对应**表 5-2** 中的“句柄（撤销注册）”。其他函数只是简单的使用句柄（这些函数并不会创建获释放这些句柄），在它们的格式字符串中包含%！。基本上，句柄是在进程的上下文中，用于唯一标识一个对象的数字。当一个 API 函数产生一个新的句柄时，客户端通常必须传入一个 OBJECT_ATTRIBUTES 结构来存放该句柄以及其他的一些东西，对于对象的名字来说，它应该是可以访问的。稍后，将不再需要该名称，因为系统可以使用对象句柄在句柄表中查寻其对应的对象的属性。这对于 API Spy 的用户来说却不是好消息，因为它必需在大量的协议项中用毫无疑义的数字代替符号名称。因此，我的 Spy 设备将注册所有的对象名称及其对应的句柄和该句柄所有者进程的 ID，每当一个新的句柄出现，就会更新这一列表。当一个注册的句柄/进程对在稍后出现时，API 记录者将会从列表中检索出其对应的原始符号名称，并将其加入的协议中。

已注册的句柄在被某个 API 函数显示关闭前，都是有效的。如果在某个可产生新的句柄的 API 调用中，再次出现已注册句柄，那么该句柄将不再处于已注册状态。对于 Windows 2000，我频繁的观测系统有时会返回相同的句柄值，尽管在此之前，协议里没有包含任何关闭该句柄的调用。我不记得在 Windows NT 4.0 中是否也存在类似的情况。一个已注册的句柄，若再次代表不同的对象属性出现时，会被显示的由某种方式关闭掉，因此必须要撤销此种句柄的注册。否则，Spy 设备的句柄目录将最终进入溢出状态。

SpyHookPrtocol()调用的 SpyWriteFilter()函数（在**列表 5-7** 中给出）是句柄跟踪机制的一个基本组成部分。任何被 hook 的 API 函数在被调用时，都会经过该函数。**列表 5-8** 给出了该函数的实现方式。

```
BOOL SpyWriteFilter (PSPY_PROTOCOL psp,
                     PBYTE          pbFormat,
                     PVOID          pParameters,
                     DWORD          dParameters)
    {
    PHANDLE                 phObject = NULL;
    HANDLE                  hObject  = NULL;
    POBJECT_ATTRIBUTES poa          = NULL;
    PDWORD                  pdNext;
    DWORD                   i, j;
    pdNext = pParameters;
    i = j  = 0;
    while (pbFormat [i])
        {
        while (pbFormat [i] && (pbFormat [i] != '%')) i++;
        if (pbFormat [i] && pbFormat [++i])
            {
            j++;
            switch (pbFormat [i++])
                {
                case 'b':
                case 'a':
                case 'w':
                case 'u':
```

```
                case 'n':
                case 'l':
                case 's':
                case 'i':
                case 'c':
                case 'd':
                case 'p':
                        {
                        break;
                        }
                case 'o':
                        {
                        if (poa == NULL)
                                {
                                poa = (POBJECT_ATTRIBUTES) *pdNext;
                                }
                        break;
                        }
                case '+':
                        {
                        if (phObject == NULL)
                                {
                                phObject = (PHANDLE) *pdNext;
                                }
                        break;
                        }
                case '!':
                case '-':
                        {
```

```
                        if (hObject == NULL)
                            {
                            hObject = (HANDLE) *pdNext;
                            }
                        break;
                        }
                    default:
                        {
                        j--;
                        break;
                        }
                    }
                pdNext++;
                }
            }
    return // number of arguments ok
                (j == dParameters)
                &&
                 // no handles involved
                (((phObject == NULL) && (hObject == NULL))
                 ||
                 // new handle, successfully registered
                 ((phObject != NULL) &&
                   SpyHandleRegister (psp, PsGetCurrentProcessId (),
                                      *phObject, OBJECT_NAME (poa)))
                 ||
                 // registered handle
                 SpyHandleSlot (psp, PsGetCurrentProcessId (), hObject)
                 ||
```

```
            // filter disabled
            (!gfSpyHookFilter));
    }
```

列表 5-8. SpyWriteFilter()将从协议中去除未注册的 API 调用

SpyWriteFilter()主要通过扫描协议格式字符串来查找%o（对象属性）、%+（新的句柄）、%!（打开句柄）和%-（关闭句柄），并对其中某几个控制 ID 的组合采取特殊的动作，如下所示：

- 如果没有涉及句柄，所有的 API 调用都将被记录。这涉及所有格式字符串中不包含格式控制 ID%+、%!和%-的 API 函数。
- 如果格式字符串中包含%+，这表示该 API 函数将分配一个新的句柄，将使用帮助函数 SpyHandleRegister()来注册该句柄，并且将该句柄与格式字符串中第一个%o 项的名字关联起来。如果这样的名字不存在，则该句柄将以一个空字符串注册。如果注册成功，此 API 调用将被记录。
- 如果格式字符串中包含%!或%-，则被调用的函数将使用或关闭一个句柄。这种情况下，SpyWriteFilter()将使用 SpyHandleSlot()函数来查询该句柄的索引，以测试该句柄是否已注册。如果测试成功，该 API 调用将被记录。
- 对于所有其他的情况，仅当过滤机制关闭时，函数调用才被记录。过滤机制是否开启由全局逻辑变量 gfSpyHookFilter 表示。

句柄目录是 SPY_PROTOCOL 结构的一部分，包含在 Spy 设备（w2k_spy.sys）的全局 DEVICE_CONTEXT 结构中，**列表 5-9** 给出了它的定义以及 SPY_HEADER 子结构的定义。在这些定义之后的是四个句柄管理函数（SpyHandleSlot()、SpyHandleName()、SpyHandleUnregister()和 SpyHandleRegister()）的源代码。通过将句柄的值加入到 ahObjets[]数组的末尾来注册该句柄。同时，拥有该句柄的进程 ID 将被保存到 ahprocesses[]数组中，对象名将被复制到 awNames[]缓冲区中，名字的起始偏移地址将被保存到 adNames[]数组中。当注销一个句柄时，过程与上述类似，这可能需要移动数组中的其它元素以保证数组中不存在“空洞”。**列表 5-9** 顶部定义的常量定义了句柄目录的大小：它最多可存放 4,096 个句柄，名称的大小被限制为 1,048,576 个 Unicode 字符（2MB），协议缓冲区的大小为 1MB。

```
#define SPY_HANDLES               0x00001000 // max number of handles
#define SPY_NAME_BUFFER           0x00100000 // object name buffer size
#define SPY_DATA_BUFFER           0x00100000 // protocol data buffer size
// -------------------------------------------------------------
typedef struct _SPY_HEADER
      {
      LARGE_INTEGER liStart;    // start time
      DWORD             dRead;       // read data index
      DWORD             dWrite;      // write data index
      DWORD             dCalls;      // api usage count
      DWORD             dHandles; // handle count
      DWORD             dName;       // object name index
      }
      SPY_HEADER, *PSPY_HEADER, **PPSPY_HEADER;
#define SPY_HEADER_ sizeof (SPY_HEADER)
// -------------------------------------------------------------
typedef struct _SPY_PROTOCOL
      {
      SPY_HEADER       sh;                                    // protocol header
      HANDLE           ahProcesses [SPY_HANDLES];         // process id array
      HANDLE           ahObjects      [SPY_HANDLES];         // handle array
      DWORD            adNames          [SPY_HANDLES];         // name offsets
      WORD             awNames          [SPY_NAME_BUFFER]; // name strings
      BYTE             abData           [SPY_DATA_BUFFER]; // protocol data
      }
      SPY_PROTOCOL, *PSPY_PROTOCOL, **PPSPY_PROTOCOL;
#define SPY_PROTOCOL_ sizeof (SPY_PROTOCOL)
// =============================================================
```

```
// HANDLE MANAGEMENT
// ================================================================
DWORD SpyHandleSlot (PSPY_PROTOCOL psp,
                     HANDLE          hProcess,
                     HANDLE          hObject)
    {
    DWORD dSlot = 0;
    if (hObject != NULL)
        {
        while ((dSlot < psp->sh.dHandles)
               &&
               ((psp->ahProcesses [dSlot] != hProcess) ||
                (psp->ahObjects   [dSlot] != hObject ))) dSlot++;
        dSlot = (dSlot < psp->sh.dHandles ? dSlot+1 : 0);
        }
    return dSlot;
    }
// ----------------------------------------------------------------
DWORD SpyHandleName (PSPY_PROTOCOL psp,
                     HANDLE          hProcess,
                     HANDLE          hObject,
                     PWORD           pwName,
                     DWORD           dName)
    {
    WORD    w;
    DWORD i;
    DWORD dSlot = SpyHandleSlot (psp, hProcess, hObject);
    if ((pwName != NULL) && dName)
        {
```

```
        i = 0;
        if (dSlot)
            {
            while ((i+1 < dName) &&
                    (w = psp->awNames [psp->adNames [dSlot-1] + i]))
                {
                pwName [i++] = w;
                }
            }
        pwName [i] = 0;
        }
    return dSlot;
    }
// ------------------------------------------------------------
DWORD SpyHandleUnregister (PSPY_PROTOCOL psp,
                           HANDLE       hProcess,
                           HANDLE       hObject,
                           PWORD        pwName,
                           DWORD        dName)
    {
    DWORD i, j;
    DWORD dSlot = SpyHandleName (psp, hProcess, hObject,
                                 pwName, dName);
    if (dSlot)
        {
        if (dSlot == psp->sh.dHandles)
            {
            // remove last name entry
            psp->sh.dName = psp->adNames [dSlot-1];
```

```
                }
            else
                {
                i = psp->adNames [dSlot-1];
                j = psp->adNames [dSlot    ];
                // shift left all remaining name entries
                while (j < psp->sh.dName)
                    {
                    psp->awNames [i++] = psp->awNames [j++];
                    }
                j -= (psp->sh.dName = i);
                // shift left all remaining handles and name offsets
                for (i = dSlot; i < psp->sh.dHandles; i++)
                    {
                    psp->ahProcesses [i-1] = psp->ahProcesses [i];
                    psp->ahObjects     [i-1] = psp->ahObjects     [i];
                    psp->adNames       [i-1] = psp->adNames       [i] - j;
                    }
                }
            psp->sh.dHandles--;
            }
        return dSlot;
        }
// -----------------------------------------------------------------
DWORD SpyHandleRegister (PSPY_PROTOCOL     psp,
                         HANDLE            hProcess,
                         HANDLE            hObject,
                         PUNICODE_STRING puName)
    {
```

```
PWORD pwName;
DWORD dName;
DWORD i;
DWORD dSlot = 0;
if (hObject != NULL)
    {
    // unregister old handle with same value
    SpyHandleUnregister (psp, hProcess, hObject, NULL, 0);
    if (psp->sh.dHandles == SPY_HANDLES)
        {
        // unregister oldest handle if overflow
        SpyHandleUnregister (psp, psp->ahProcesses [0],
                                    psp->ahObjects [0], NULL, 0);
        }
    pwName = ((puName != NULL) && SpyMemoryTestAddress (puName)
            ? puName->Buffer
            : NULL);
    dName  = ((pwName != NULL) && SpyMemoryTestAddress (pwName)
            ? puName->Length / WORD_
            : 0);
    if (dName + 1 <= SPY_NAME_BUFFER - psp->sh.dName)
        {
        // append object to end of list
        psp->ahProcesses [psp->sh.dHandles] = hProcess;
        psp->ahObjects   [psp->sh.dHandles] = hObject;
        psp->adNames     [psp->sh.dHandles] = psp->sh.dName;
        for (i = 0; i < dName; i++)
            {
            psp->awNames [psp->sh.dName++] = pwName [i];
```

```
                    }
                psp->awNames [psp->sh.dName++] = 0;
                psp->sh.dHandles++;
                dSlot = psp->sh.dHandles;
                }
            }
    return dSlot;
    }
```

列表 5-9. 句柄管理相关的结构和函数

5.2、在用户模式下控制 API Hooks

运行在用户模式的 Spy 客户端可通过一组 IOCTL 函数来控制 API Hook 机制及其生成的协议。这一组函数的名字都以 SPY_IO_HOOK_开始，在第四章已介绍过它们，在第四章还讨论了 w2k_spy.sys 的内存 Spy 函数（参见**列表 4-7** 和**表 4-2**）。

下面的**表 5-3** 再次给出了与**表 4-2** 相关的部分。**列表 5-10** 是**列表 4-7** 的一个摘录，示范了 Hook 管理函数是如何被分派（dispatch）的。这些管理函数在后面将会被反复提及。

表 5-3. Spy Device 支持的 IOCTL Hook 管理函数

函数名称	ID	IOCTL 代码	描　述
SPY_IO_HOOK_INFO	11	0x8000602C	返回 Native API Hook 的相关信息
SPY_IO_HOOK_INSTALL	12	0x8000E030	安装 Native API Hook
SPY_IO_HOOK_REMOVE	13	0x8000 E034	移除 Native API Hook
SPY_IO_HOOK_PAUSE	14	0x8000 E038	暂停/恢复 Hook 协议
SPY_IO_HOOK_FILTER	15	0x8000 E03C	允许/禁止 Hook 协议过滤器
SPY_IO_HOOK_RESET	16	0x8000 E040	清除 Hook 协议
SPY_IO_HOOK_READ	17	0x80006044	从 Hook 协议中读取数据
SPY_IO_HOOK_WRITE	18	0x8000E048	向 Hook 协议中写入数据

```
NTSTATUS SpyDispatcher (PDEVICE_CONTEXT pDeviceContext,
                        DWORD           dCode,
                        PVOID           pInput,
                        DWORD           dInput,
                        PVOID           pOutput,
                        DWORD           dOutput,
                        PDWORD          pdInfo)
    {
    SPY_MEMORY_BLOCK smb;
    SPY_PAGE_ENTRY   spe;
    SPY_CALL_INPUT   sci;
    PHYSICAL_ADDRESS pa;
    DWORD            dValue, dCount;
    BOOL             fReset, fPause, fFilter, fLine;
    PVOID            pAddress;
    PBYTE            pbName;
    HANDLE           hObject;
    NTSTATUS         ns = STATUS_INVALID_PARAMETER;
    MUTEX_WAIT (pDeviceContext->kmDispatch);
    *pdInfo = 0;
    switch (dCode)
        {
        case SPY_IO_VERSION_INFO:
            {
            ns = SpyOutputVersionInfo (pOutput, dOutput, pdInfo);
            break;
            }
        case SPY_IO_OS_INFO:
```

```
        {
        ns = SpyOutputOsInfo (pOutput, dOutput, pdInfo);
        break;
        }
    case SPY_IO_SEGMENT:
        {
        if ((ns = SpyInputDword (&dValue,
                                        pInput, dInput))
            == STATUS_SUCCESS)
            {
            ns = SpyOutputSegment (dValue,
                                            pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_INTERRUPT:
        {
        if ((ns = SpyInputDword (&dValue,
                                        pInput, dInput))
            == STATUS_SUCCESS)
            {
            ns = SpyOutputInterrupt (dValue,
                                                pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_PHYSICAL:
        {
        if ((ns = SpyInputPointer (&pAddress,
```

```
                                        pInput, dInput))
            == STATUS_SUCCESS)
            {
            pa = MmGetPhysicalAddress (pAddress);
            ns = SpyOutputBinary (&pa, PHYSICAL_ADDRESS_,
                                        pOutput, dOutput, pdInfo);
            }
        break;
        }
```

```
    case SPY_IO_CPU_INFO:
        {
        ns = SpyOutputCpuInfo (pOutput, dOutput, pdInfo);
        break;
        }
    case SPY_IO_PDE_ARRAY:
        {
        ns = SpyOutputBinary (X86_PDE_ARRAY, SPY_PDE_ARRAY_,
                                    pOutput, dOutput, pdInfo);
        break;
        }
    case SPY_IO_PAGE_ENTRY:
        {
        if ((ns = SpyInputPointer (&pAddress,
                                        pInput, dInput))
            == STATUS_SUCCESS)
            {
            SpyMemoryPageEntry (pAddress, &spe);
            ns = SpyOutputBinary (&spe, SPY_PAGE_ENTRY_,
                                        pOutput, dOutput, pdInfo);
```

```
            }
        break;
        }
    case SPY_IO_MEMORY_DATA:
        {
        if ((ns = SpyInputMemory (&smb,
                                  pInput, dInput))
            == STATUS_SUCCESS)
            {
            ns = SpyOutputMemory (&smb,
                                  pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_MEMORY_BLOCK:
        {
        if ((ns = SpyInputMemory (&smb,
                                  pInput, dInput))
            == STATUS_SUCCESS)
            {
            ns = SpyOutputBlock (&smb,
                                 pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_HANDLE_INFO:
        {
        if ((ns = SpyInputHandle (&hObject,
                                  pInput, dInput))
```

```
            == STATUS_SUCCESS)
            {
            ns = SpyOutputHandleInfo (hObject,
                                        pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_HOOK_INFO:
        {
        ns = SpyOutputHookInfo (pOutput, dOutput, pdInfo);
        break;
        }
    case SPY_IO_HOOK_INSTALL:
        {
        if (((ns = SpyInputBool (&fReset,
                                    pInput, dInput))
              == STATUS_SUCCESS)
            &&
            ((ns = SpyHookInstall (fReset, &dCount))
              == STATUS_SUCCESS))
            {
            ns = SpyOutputDword (dCount,
                                    pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_HOOK_REMOVE:
        {
        if (((ns = SpyInputBool (&fReset,
```

```
                                        pInput, dInput))
            == STATUS_SUCCESS)
          &&
          ((ns = SpyHookRemove (fReset, &dCount))
            == STATUS_SUCCESS))
          {
          ns = SpyOutputDword (dCount,
                                        pOutput, dOutput, pdInfo);
          }
        break;
        }
      case SPY_IO_HOOK_PAUSE:
        {
        if ((ns = SpyInputBool (&fPause,
                                        pInput, dInput))
          == STATUS_SUCCESS)
          {
          fPause = SpyHookPause (fPause);
          ns = SpyOutputBool (fPause,
                                        pOutput, dOutput, pdInfo);
          }
        break;
        }
      case SPY_IO_HOOK_FILTER:
        {
        if ((ns = SpyInputBool (&fFilter,
                                        pInput, dInput))
          == STATUS_SUCCESS)
          {
```

```
                fFilter = SpyHookFilter (fFilter);
                ns = SpyOutputBool (fFilter,
                                        pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_HOOK_RESET:
            {
            SpyHookReset ();
            ns = STATUS_SUCCESS;
            break;
            }
        case SPY_IO_HOOK_READ:
```

```
            {
            if ((ns = SpyInputBool (&fLine,
                                        pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputHookRead (fLine,
                                            pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_HOOK_WRITE:
            {
            SpyHookWrite (pInput, dInput);
            ns = STATUS_SUCCESS;
            break;
            }
```

```
        case SPY_IO_MODULE_INFO:
            {
            if ((ns = SpyInputPointer (&pbName,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputModuleInfo (pbName,
                                          pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_PE_HEADER:
            {
            if ((ns = SpyInputPointer (&pAddress,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputPeHeader (pAddress,
                                        pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_PE_EXPORT:
            {
            if ((ns = SpyInputPointer (&pAddress,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputPeExport (pAddress,
```

```
                                                pOutput, dOutput, pdInfo);
                    }
                break;
                }
        case SPY_IO_PE_SYMBOL:
                {
                if ((ns = SpyInputPointer (&pbName,
                                                pInput, dInput))
                    == STATUS_SUCCESS)
                    {
                    ns = SpyOutputPeSymbol (pbName,
                                                pOutput, dOutput, pdInfo);
                    }
                break;
                }
```

```
        case SPY_IO_CALL:
                {
                if ((ns = SpyInputBinary (&sci, SPY_CALL_INPUT_,
                                                pInput, dInput))
                    == STATUS_SUCCESS)
                    {
                    ns = SpyOutputCall (&sci,
                                                pOutput, dOutput, pdInfo);
                    }
                break;
                }
        }
    MUTEX_RELEASE (pDeviceContext->kmDispatch);
    return ns;
```

```
    }
```

列表 5-10. Spy Driver 的 Hook 命令分派器

5.2.1、IOCTL 函数 SPY_IO_HOOK_INFO

IOCTL 函数 SPY_IO_HOOK_INFO 采用 Hook 机制的当前信息来填充一个 SPY_HOOK_INFO 结构，这和系统的 SDT 类似。SPY_HOOK_INFO 结构（**列表 5-11**）中引用了多个前面介绍过的结构：

- SERVICE_DESCRIPTOR_TABLE 结构，定义于**列表 5-1**
- SPY_CALL 和 SPY_HOOK_ENTRY 结构，定义于**列表 5-2**
- SPY_HEADER 和 SPY_PROTOCOL 结构，定义于**列表 5-9**

```
typedef struct _SPY_HOOK_INFO
    {
        SPY_HEADER                          sh;
        PSPY_CALL                           psc;
        PSPY_PROTOCOL                       psp;
        PSERVICE_DESCRIPTOR_TABLE           psdt;
        SERVICE_DESCRIPTOR_TABLE            sdt;
        DWORD                               ServiceLimit;
        NTPROC                              ServiceTable    [SDT_SYMBOLS_MAX];
        BYTE                                ArgumentTable [SDT_SYMBOLS_MAX];
        SPY_HOOK_ENTRY                      SpyHooks        [SDT_SYMBOLS_MAX];
    }
    SPY_HOOK_INFO, *PSPY_HOOK_INFO, **PPSPY_HOOK_INFO;
#define SPY_HOOK_INFO_ sizeof (SPY_HOOK_INFO)
```

列表 5-11. SPY_HOOK_INFO 结构定义

在计算该结构体成员的值时一定要小心。某些成员中的指针指向内核模式下的内存块，这些内存地址在用户模式下是无法访问的。不过，你可以使用 Spy 设备的 SPY_IO_MEMORY_DATA 函数来检查这些内存块中的数据。

5.2.2、IOCTL 函数　SPY_IO_HOOK_INSTALL

SPY_IO_HOOK_INSTALL 函数使用存储在全局 aSpyHooks[]数组中的 Hook 进入点（Hook Entry Point）来修改（patch）ntoskrnl.exe 在系统 SDT 中的服务表（Service Table）。该全局数组在驱动程序初始化时由 SpyHookInitialize()(**列表 5-5**)和 SpyHookInitializeEx()(**列表 5-3**)函数构建。aSpyHooks[]数组中的每个有效的元素均包含一个 hook 进入点以及相应的格式字符串地址。SpyDispatcher()调用辅助函数 SpyHookInstall()（见**列表 5-12**）来安装 Hook。SpyHookInstall()使用的 SpyHookExchange()函数也包含在**列表 5-12** 中。

```
DWORD SpyHookExchange (void)
    {
    PNTPROC ServiceTable;
    BOOL       fPause;
    DWORD     i;
    DWORD     n = 0;
    fPause        = SpyHookPause (TRUE);
    ServiceTable = KeServiceDescriptorTable->ntoskrnl.ServiceTable;
    for (i = 0; i < SDT_SYMBOLS_MAX; i++)
        {
        if (aSpyHooks [i].pbFormat != NULL)
            {
            aSpyHooks [i].Handler = (NTPROC)
                InterlockedExchange ((PLONG) ServiceTable+i,
                                                ( LONG) aSpyHooks [i].Handler);
            n++;
```

```
                }
            }
        gfSpyHookState = !gfSpyHookState;
        SpyHookPause (fPause);
        return n;
        }
// -------------------------------------------------------------
NTSTATUS SpyHookInstall (BOOL      fReset,
                                    PDWORD pdCount)
        {
        DWORD       n    = 0;
        NTSTATUS ns = STATUS_INVALID_DEVICE_STATE;
        if (!gfSpyHookState)
            {
            ghSpyHookThread = PsGetCurrentThreadId ();
            n = SpyHookExchange ();
            if (fReset) SpyHookReset ();
            ns = STATUS_SUCCESS;
            }
        *pdCount = n;
        return ns;
        }
```

列表 5-12. Patch the System's API Service Table

在安装和移除 Hooks 时都需要用到 SpyHookExchange()函数，该函数只是简单的交换系统的 API 服务表和 aSpyHooks[]数组中的内容。因此，若调用两次 SpyHookExchange()，则可恢复服务表和数组的初始状态。SpyHookExchange()遍历 aSpyHooks[]数组来寻找包含格式字符串指针的数组元素。这里的格式字符串是为了指定要监控的 API 函数。在这种情况下，将使用 ntoskrnl.exe 中的 InterlockedExchange()函数来交换服务表中的 API 函数指针和 aSpyHooks[]元素的 Handler 成员，使用 InterlockedExchange()函数是为了保证没有其他线程

干扰这一操作。协议机制在这一过程中将被临时停止，直到对服务表的修改结束。SpyHookInstall()仅是 SpyHookExchange()的一个外包函数，这样做是为了执行一些附加的操作：

- 如果全局标志 gfSpyHookState 表示 Hooks 仍在安装，则禁止访问服务表（Service Table）。
- 调用者的线程 ID 将被写入全局变量 ghSpyHookThread。SpyHookInitializeEx()函数中的 Hook 分派器（Dispatcher）将屏蔽该变量所指线程发出的 API 调用。以避免 Hook 协议受到干扰。
- 根据客户端的请求，协议会被重置。这意味着所有的缓冲区都将被清空，并且句柄目录也将被重新初始化。

SPY_IO_HOOK_INSTALL 接收调用者传入的一个逻辑型参数。如果为 TRUE，协议将在 Hooks 安装完成后被重置。这是常用的一个选项。如果为 FALSE，协议将从前一次 Hook 会话结束的位置继续。该函数的返回值表示服务表中被修改的项数。如果是 Windows 2000，SPY_IO_HOOK_INSTALL 返回的是 44，这一数值正是**列表 5-6** 给出的格式字符串数组 apdSdtFormats[]的大小。在 Windows NT 4.0 下，则仅有 42 个 Hook 被安装，因为 API 函数 NtNotifyChangeMultipleKeys()和 NtQueryOpenSubKeys()并不被 NT 4.0 支持。

IOCTL 函数　SPY_IO_HOOK_REMOVE

IOCTL 函数 SPY_IO_HOOK_REMOVE 函数和 SPY_IO_HOOK_INSTALL 很相似，只不过它执行与 SPY_IO_HOOK_INSTALL 相反的操作。这两个函数的输入、输出参数都是相同的。下面是**列表 5-12** 和**列表 5-13** 的对比：

- 如果全局标志 gfSpyHookState 表示当前并未安装 Hooks，那么调用将被忽略。
- 在将 Service Table 恢复到修改前的状态后，在全局变量 ghSpyHookThread 中保存的安装 Hook 的线程 ID 将被置零。
- 最重要的扩展特性是位于**列表 5-13** 中部的 do/while 循环，SpyHookRemove()通过测试全局结构 DEVICE_CONTEXT 中的所有 SPY_CALL 子结构的 fInUse 成员来判断 Hook Dispatcher 是否还在为其他线程提供服务。这种测试是必须的，因为客户可能会在卸载 Hook 后就立即卸载 Spy 驱动程序。如果在 Hook Dispatcher 中还存在其他程序的 API 调用的情况下，就卸载 Spy 驱动程序，系统将抛出一个异常，

随后将引发蓝屏。对 fInUse 的测试每隔 100msec 就进行一次，以使其他线程有机会推出 Spy。

```
NTSTATUS SpyHookRemove (BOOL      fReset,
                        PDWORD pdCount)
    {
    LARGE_INTEGER liDelay;
    BOOL              fInUse;
    DWORD             i;
    DWORD             n    = 0;
    NTSTATUS          ns = STATUS_INVALID_DEVICE_STATE;
    if (gfSpyHookState)
        {
        n = SpyHookExchange ();
        if (fReset) SpyHookReset ();
        do   {
            for (i = 0; i < SPY_CALLS; i++)
                {
                if (fInUse = gpDeviceContext->SpyCalls [i].fInUse)
                    break;
                }
            liDelay.QuadPart = -1000000;
            KeDelayExecutionThread (KernelMode, FALSE, &liDelay);
            }
        while (fInUse);
        ghSpyHookThread = 0;
        ns = STATUS_SUCCESS;
        }
```

```
    *pdCount = n;
    return ns;
    }
```

列表 5-13. 恢复系统的 API Service Table

注意，即使所有的 fInUse 标志都被清除，最后的 100msec 延迟也是必须的。这种预防措施是由于在 Hook Dispatcher 中存在一个很小的安全漏洞，这一漏洞存在于重置当前 SPY_CALL 中的 fInUse 标志的指令和 Dispatcher 将控制返回给调用着的 RET 指令（参考**列表 5-2** 中，SpyHook8 和 SpyHook9 之间的 ASM 代码）之间。如果所有的 fInUse 标志都是 FALSE，则存在一个很小的可能性，使得某些线程在要执行 RET 指令之前被暂停（Suspend）。继续延迟 100msec 后再移除 Hook，则可以使得这些线程有机会离开这一临界区。

5.2.3、IOCTL 函数 SPY_IO_HOOK_PAUSE

列表 5-14 给出了 SPY_IO_HOOK_PAUSE 函数，该函数允许一个客户临时禁止（或再次允许）Hook 协议函数。实质上，该函数让客户设置全局逻辑变量 gfSpyHookPause 并将其原始值返回给客户来实现这一功能的，这里用到了 ntoskrnl.exe 中的 InterlockExchange()函数。默认情况下，协议是被允许的；这就意味着，gfSpyHookPause 的初值为 FALSE。

```
BOOL SpyHookPause (BOOL fPause)
    {
    BOOL fPause1 = (BOOL)
                    InterlockedExchange ((PLONG) &gfSpyHookPause,
                                         ( LONG) fPause);
    if (!fPause) SpyHookReset ();
    return fPause1;
    }
```

列表 5-14. 打开/关闭协议

这里要特别注意的是，SPY_IO_HOOK_PAUSE 的工作完全依赖于 SPY_IO_HOOK_INSTALL 和 SPY_IO_HOOK_REMOVE。如果在安装 Hook 时，禁止了协议，Hook 仍然会起作用，但是此时 Hook Dispatcher 将对所有捕获到的 API 调用不进行任何

干扰。如果你不希望 SPY_IO_HOOK_INSTALL 修改 Service Table 后，协议就自动开始的话，你可以在安装 Hook 之前禁止 Hook 协议。注意，当协议被再次恢复时，协议将被自动重置。

5.2.4、IOCTL 函数　SPY_IO_HOOK_FILTER

IOCTL 函数 SPY_IO_HOOK_FILTER 维护一个全局标志，如**列表 5-15** 所示。这里，全局标志 gfSpyHookFilter 被设置为一个由客户提供的值，其先前的值将被返回给客户。该标志的默认值为 FALSE；这意味着，默认情况下，过滤器是被禁止的。

```
BOOL SpyHookFilter (BOOL fFilter)
    {
    return (BOOL) InterlockedExchange ((PLONG) &gfSpyHookFilter,
                                       ( LONG) fFilter);
    }
```

列表 5-15.　打开/关闭协议过滤器

在讨论**列表 5-8** 中的 SpyWriteFilter()函数时，就已经涉及到 gfSpyHookFilter，你应该对该变量不再陌生。如果 gfSpyHookFilter 为 TRUE，辅助函数 SpyHookProtocol()（参见**列表 5-7**）将忽略所有的 API 调用，这包括含有先前并未在 Spy device 中注册的句柄的 API 调用。

5.2.5、IOCTL 函数　SPY_IO_HOOK_RESET

IOCTL 函数 SPY_IO_HOOK_RESET 用于将协议机制恢复到原始状态，这包括清除数据缓冲区和“丢掉”所有已注册的句柄。由 SpyDispatcher()调用的 SpyHookReset()函数只是 SpyWriteReset()的一个外包函数而已。这两个函数都包含在**列表 5-16** 中。SpyHookReset()的附加特性只是调用了使用 mutex 的 SpyHookWait()和 SpyHookRelease()函数（参见**列表 5-7**）来进行同步。

```
void SpyWriteReset (PSPY_PROTOCOL psp)
{
    KeQuerySystemTime (&psp->sh.liStart);
```

```
        psp->sh.dRead       = 0;
        psp->sh.dWrite      = 0;
        psp->sh.dCalls      = 0;
        psp->sh.dHandles = 0;
        psp->sh.dName       = 0;
}
// -------------------------------------------------------------
void SpyHookReset (void)
{
        SpyHookWait       ();
        SpyWriteReset    (&gpDeviceContext->SpyProtocol);
        SpyHookRelease ();
        return;
}
// -------------------------------------------------------------
NTSTATUS SpyHookWait (void)
{
        return MUTEX_WAIT (gpDeviceContext->kmProtocol);
}
// -------------------------------------------------------------
LONG SpyHookRelease (void)
{
        return MUTEX_RELEASE (gpDeviceContext->kmProtocol);
}
```

列表 5-16. 重置协议

5.2.6、IOCTL 函数 SPY_IO_HOOK_READ

API Hook 记录器（logger）将协议数据写入 abData[]缓冲区中，该缓冲区位于全局结构 SPY_PROTOCOL 中，该结构体已在**列表 5-9** 中给出。这种类型的缓冲区被设计为一个环形

缓冲区。这意味着，其特点是使用一对指针来分别进行读/写访问。只要其中的一个指针移动到了缓冲区的尾端，它就会被重置并指向缓冲区的头部。读指针一直试图追上写指针，如果这两个指针指向相同地址，那意味着缓冲区为空。

SPY_IO_HOOK_READ 是到目前为止，Spy Device 提供的最重要的管理 Hook 的函数之一。它可以从协议数据缓冲区中读取任意大小的数据，并适当的调整读指针。在协议被允许时，该函数应该被频繁的调用，以避免缓冲区的溢出。**列表 5-17** 给出了处理此种 IOCTL 请求的一组函数。其中最基本的是 SpyReadData()和 SpyReadLine()。二者的区别在于如果可能，前者将返回所请求的数据，而后者仅返回一整行数据。当客户端要对读取到的数据进行过滤时，行模式就显得很方便了。SPY_IO_HOOK_READ 的调用者通过传入一个逻辑型变量来确定是读取方式是行模式还是块模式。

```
DWORD SpyReadData (PSPY_PROTOCOL psp,
                   PBYTE        pbData,
                   DWORD          dData)
{
    DWORD i = psp->sh.dRead;
    DWORD n = 0;
    while ((n < dData) && (i != psp->sh.dWrite))
        {
        pbData [n++] = psp->abData [i++];
        if (i == SPY_DATA_BUFFER) i = 0;
        }
    psp->sh.dRead = i;
    return n;
}
// -------------------------------------------------------------
DWORD SpyReadLine (PSPY_PROTOCOL psp,
                   PBYTE        pbData,
                   DWORD          dData)
{
    BYTE   b = 0;
```

```
	DWORD i = psp->sh.dRead;
	DWORD n = 0;
	while ((b != '\n') && (i != psp->sh.dWrite))
	{
		b = psp->abData [i++];
		if (i == SPY_DATA_BUFFER) i = 0;
		if (n < dData) pbData [n++] = b;
	}
	if (b == '\n')
	{
		// remove current line from buffer
		psp->sh.dRead = i;
	}
	else
	{
		// don't return any data until full line available
		n = 0;
	}
	if (n)
	{
		pbData [n-1] = 0;
	}
	else
	{
		if (dData) pbData [0] = 0;
	}
	return n;
}
// -------------------------------------------------------------
```

```
DWORD SpyHookRead (PBYTE pbData,
                   DWORD dData,
                   BOOL    fLine)
{
    DWORD n = 0;
    SpyHookWait ();
    n = (fLine ? SpyReadLine : SpyReadData)
            (&gpDeviceContext->SpyProtocol, pbData, dData);
    SpyHookRelease ();
    return n;
}
// -------------------------------------------------------------
NTSTATUS SpyOutputHookRead (BOOL      fLine,
                            PVOID    pOutput,
                            DWORD    dOutput,
                            PDWORD pdInfo)
{
    *pdInfo = SpyHookRead (pOutput, dOutput, fLine);
    return STATUS_SUCCESS;
}
```

列表 5-17. 从协议缓冲区中读取数据

SpyOutputHookRead()和 SpyHookRead()函数没有太大价值。SpyHookRead()使用了 Mutex 来进行同步，而且可以选择 SpyReadLine()或 SpyReadData()，SpyOutputHookRead()则是根据 IOCTL 框架的要求来提交它的结果。

5.2.7、IOCTL 函数 SPY_IO_HOOK_WRITE

该函数允许客户向协议缓冲区中写入数据。应用程序可使用这一特性来向协议中增加独立的或附加的状态信息。**列表 5-18** 给出了该函数的实现部分。SpyHookWrite()是另一个外

包函数，它使用了 Mutex 进行同步。它调用的 SpyWriteData()函数是 Spy Device 中最基本的协议生成器。所有的 SpyWrite*()辅助函数（如，SpyWriteFormat()、SpyWriteNumber()、SpyWriteChar()和 SpyWriteLarge()函数由 SpyHookProtocol()使用，见**列表 5-7**）都是基于 SpyWriteData()的。

```
DWORD SpyWriteData (PSPY_PROTOCOL psp,
                    PBYTE         pbData,
                    DWORD         dData)
{
    BYTE    b;
    DWORD i = psp->sh.dRead;
    DWORD j = psp->sh.dWrite;
    DWORD n = 0;
    while (n < dData)
        {
        psp->abData [j++] = pbData [n++];
        if (j == SPY_DATA_BUFFER) j = 0;
        if (j == i)
            {
            // remove first line from buffer
            do   {
                b = psp->abData [i++];
                if (i == SPY_DATA_BUFFER) i = 0;
                }
            while ((b != '\n') && (i != j));
            // remove half line only if single line
            if ((i == j) &&
                ((i += (SPY_DATA_BUFFER / 2)) >= SPY_DATA_BUFFER))
```

```
                        {
                        i -= SPY_DATA_BUFFER;
                        }
                    }
                }
        psp->sh.dRead    = i;
        psp->sh.dWrite = j;
        return n;
}
// -------------------------------------------------------------
DWORD SpyHookWrite (PBYTE pbData,
                            DWORD dData)
{
        DWORD n = 0;
        SpyHookWait ();
        n = SpyWriteData
                (&gpDeviceContext->SpyProtocol, pbData, dData);
        SpyHookRelease ();
        return n;
}
```

列表 5-18. 向协议缓冲区中写入数据

这里要注意 SpyWriteData()是如何管理缓冲区溢出情况的。如果读指针前进的太慢，写指针将会覆盖它。此时，两个指针都是有效的：

1. 写访问将被禁止，直到读指针领先于写指针。
2. 已缓冲的数据将被丢弃，以腾出空间。

Spy Device 采用上述第二种方式。如果溢出发生了，从当前读指针的位置开始的一行协议数据将被丢弃，并将读指针移动到下一行开始处。如果缓冲区中仅有一行数据（这种情况极少出现），则仅丢弃该行的前一半数据。**列表 5-18** 中处理这些情况的代码都有相应的注释。

5.3、一个简单的 Hook 协议读取程序

为了帮助你编写自己的 API Hook Client 程序，我给出了一个简单的示例行的程序，该程序可以读取 Hook 协议缓冲区中的数据并在控制台窗口中显示。通过按下 P、F 和 R 键，可以实现暂停、过滤和重置功能，输出可以按照用户自定义的函数名模板进行过滤。这个示例程序叫做“SBS Windows 2000 API Hook Viewer”，其源代码位于本书光盘的\src\w2k_hook 目录下。

5.3.1、控制 Spy Device

为了方便，w2k_hook.exe 程序使用了一组针对 SPY_IO_HOOK_*函数的外包函数，**列表 5-19** 给出了这些函数。这些工具函数使得代码的可读性更好，并且在开发 Spy Device 的客户端程序时，大大降低了参数出错的可能性。

```
BOOL WINAPI SpyIoControl (HANDLE hDevice,
                          DWORD    dCode,
                          PVOID    pInput,
                          DWORD    dInput,
                          PVOID    pOutput,
                          DWORD    dOutput)
    {
    DWORD dInfo = 0;
    return DeviceIoControl (hDevice, dCode,
                            pInput,    dInput,
                            pOutput, dOutput,
                            &dInfo,    NULL)
           &&
           (dInfo == dOutput);
    }
// -------------------------------------------------------------------
BOOL WINAPI SpyVersionInfo (HANDLE                  hDevice,
```

```
                                    PSPY_VERSION_INFO psvi)
    {
    return SpyIoControl (hDevice, SPY_IO_VERSION_INFO,
                                    NULL, 0,
                                    psvi, SPY_VERSION_INFO_);
    }
// -------------------------------------------------------------
BOOL WINAPI SpyHookInfo (HANDLE             hDevice,
                                    PSPY_HOOK_INFO pshi)
    {
    return SpyIoControl (hDevice, SPY_IO_HOOK_INFO,
                                    NULL, 0,
                                    pshi, SPY_HOOK_INFO_);
    }
// -------------------------------------------------------------
BOOL WINAPI SpyHookInstall (HANDLE hDevice,
                                      BOOL      fReset,
                                      PDWORD pdCount)
    {
    return SpyIoControl (hDevice, SPY_IO_HOOK_INSTALL,
                                    &fReset, BOOL_,
                                    pdCount, DWORD_);
    }
// -------------------------------------------------------------
BOOL WINAPI SpyHookRemove (HANDLE hDevice,
                                     BOOL      fReset,
                                     PDWORD pdCount)
    {
    return SpyIoControl (hDevice, SPY_IO_HOOK_REMOVE,
```

```
                                    &fReset, BOOL_,
                                    pdCount, DWORD_);
        }
// -------------------------------------------------------------
BOOL WINAPI SpyHookPause (HANDLE hDevice,
                                     BOOL      fPause,
                                     PBOOL     pfPause)
        {
        return SpyIoControl (hDevice, SPY_IO_HOOK_PAUSE,
                                    &fPause, BOOL_,
                                    pfPause, BOOL_);
        }
// -------------------------------------------------------------
BOOL WINAPI SpyHookFilter (HANDLE hDevice,
                                     BOOL      fFilter,
                                     PBOOL     pfFilter)
        {
        return SpyIoControl (hDevice, SPY_IO_HOOK_FILTER,
                                    &fFilter, BOOL_,
                                    pfFilter, BOOL_);
        }
// -------------------------------------------------------------
BOOL WINAPI SpyHookReset (HANDLE hDevice)
        {
        return SpyIoControl (hDevice, SPY_IO_HOOK_RESET,
                                    NULL, 0,
                                    NULL, 0);
        }
// -------------------------------------------------------------
```

```
DWORD WINAPI SpyHookRead (HANDLE hDevice,
                          BOOL     fLine,
                          PBYTE    pbData,
                          DWORD    dData)
    {
    DWORD dInfo;
    if (!DeviceIoControl (hDevice, SPY_IO_HOOK_READ,
                          &fLine, BOOL_,
                          pbData, dData,
                          &dInfo, NULL))
        {
        dInfo = 0;
        }
    return dInfo;
    }
// -------------------------------------------------------------
BOOL WINAPI SpyHookWrite (HANDLE hDevice,
                          PBYTE    pbData)
    {
    return SpyIoControl (hDevice, SPY_IO_HOOK_WRITE,
                         pbData, lstrlenA (pbData),
                         NULL,     0);
    }
```

列表 5-19. Device I/O Control 工具函数

在使用**列表 5-19** 中的函数之前，Spy 设备驱动程序必须首先被加载并启动。这一操作和在第四章讨论的内存 Spy 程序 w2k_mem.exe 的要求大致相同。**列表 5-20** 给出了该程序的主函数：Execute()，该函数可加载/卸载 Spy 设备驱动程序、打开/关闭一个设备句柄并可通过 IOCTL 和设备进行交互。如果你对比一下**列表 5-20** 和**列表 4-29**，它们在开始和结尾处显然都是相似的。只是在中间部分，有所不同，因为这部分的代码依赖于具体的程序。

```
void WINAPI Execute (PPWORD ppwFilters,
                     DWORD    dFilters)
    {
    SPY_VERSION_INFO svi;
    SPY_HOOK_INFO       shi;
    DWORD                dCount, i, j, k, n;
    BOOL                 fPause, fFilter, fRepeat;
    BYTE                 abData [HOOK_MAX_DATA];
    WORD                 awData [HOOK_MAX_DATA];
    WORD                 awPath [MAX_PATH] = L"?";
    SC_HANDLE            hControl            = NULL;
    HANDLE               hDevice             = INVALID_HANDLE_VALUE;
    _printf (L"\r\nLoading \"%s\" (%s) ...\r\n",
             awSpyDisplay, awSpyDevice);
    if (w2kFilePath (NULL, awSpyFile, awPath, MAX_PATH))
        {
        _printf (L"Driver: \"%s\"\r\n",
                 awPath);
        hControl = w2kServiceLoad (awSpyDevice, awSpyDisplay,
                                   awPath, TRUE);
        }
    if (hControl != NULL)
        {
        _printf (L"Opening \"%s\" ...\r\n",
                 awSpyPath);
        hDevice = CreateFile (awSpyPath,
                              GENERIC_READ       | GENERIC_WRITE,
                              FILE_SHARE_READ | FILE_SHARE_WRITE,
                              NULL, OPEN_EXISTING,
                              FILE_ATTRIBUTE_NORMAL, NULL);
```

```
        }
    else
        {
        _printf (L"Unable to load the spy device driver.\r\n");
        }
    if (hDevice != INVALID_HANDLE_VALUE)
        {
        if (SpyVersionInfo (hDevice, &svi))
            {
            _printf (L"\r\n"
                    L"%s V%lu.%02lu ready\r\n",
                    svi.awName,
                    svi.dVersion / 100, svi.dVersion % 100);
            }
        if (SpyHookInfo (hDevice, &shi))
            {
            _printf (L"\r\n"
                    L"API hook parameters:          0x%08lX\r\n"
                    L"SPY_PROTOCOL structure:       0x%08lX\r\n"
                    L"SPY_PROTOCOL data buffer:     0x%08lX\r\n"
                    L"KeServiceDescriptorTable:     0x%08lX\r\n"
                    L"KiServiceTable:               0x%08lX\r\n"
                    L"KiArgumentTable:              0x%08lX\r\n"
                    L"Service table size:           0x%lX (%lu)\r\n",
                    shi.psc,
                    shi.psp,
                    shi.psp->abData,
                    shi.psdt,
                    shi.sdt.ntoskrnl.ServiceTable,
```

```
                    shi.sdt.ntoskrnl.ArgumentTable,
                    shi.ServiceLimit, shi.ServiceLimit);
        }
    SpyHookPause    (hDevice, TRUE, &fPause ); fPause    = FALSE;
    SpyHookFilter (hDevice, TRUE, &fFilter); fFilter = FALSE;
    if (SpyHookInstall (hDevice, TRUE, &dCount))
        {
        _printf (L"\r\n"
                    L"Installed %lu API hooks\r\n",
                    dCount);
        }
    _printf (L"\r\n"
                L"Protocol control keys:\r\n"
                L"\r\n"
                L"P       -   pause    ON/off\r\n"
                L"F       -   filter ON/off\r\n"
                L"R       -   reset protocol\r\n"
                L"ESC    -    exit\r\n"
                L"\r\n");
    for (fRepeat = TRUE; fRepeat;)
        {
        if (n = SpyHookRead (hDevice, TRUE,
                                    abData, HOOK_MAX_DATA))
            {
            if (abData [0] == '-')
                {
                n = 0;
                }
            else
```

```
                    {
                    i = 0;
                    while (abData [i] && (abData [i++] != '='));
                    j = i;
                    while (abData [j] && (abData [j] != ')) j++;
                    k = 0;
                    while (i < j) awData [k++] = abData [i++];
                    awData [k] = 0;
                    for (i = 0; i < dFilters; i++)
                        {
                        if (PatternMatcher (ppwFilters [i], awData))
                            {
                            n = 0;
                            break;
                            }
                        }
                    }
                if (!n) _printf (L"%hs\r\n", abData);
                Sleep (0);
                }
            else
                {
                Sleep (HOOK_IOCTL_DELAY);
                }
            switch (KeyboardData ())
                {
                case 'P':
                    {
                    SpyHookPause (hDevice, fPause, &fPause);
```

```
                    SpyHookWrite (hDevice, (fPause ? abPauseOff
                                                    : abPauseOn));
                    break;
                    }
                case 'F':
                    {
                    SpyHookFilter (hDevice, fFilter, &fFilter);
                    SpyHookWrite    (hDevice, (fFilter ? abFilterOff
                                                    : abFilterOn));
                    break;
                    }
                case 'R':
                    {
                    SpyHookReset (hDevice);
                    SpyHookWrite (hDevice, abReset);
                    break;
                    }
                case VK_ESCAPE:
                    {
                    _printf (L"%hs\r\n", abExit);
                    fRepeat = FALSE;
                    break;
                    }
                }
            }
        if (SpyHookRemove (hDevice, FALSE, &dCount))
            {
            _printf (L"\r\n"
                     L"Removed %lu API hooks\r\n",
```

```
                        dCount);
                }
        _printf (L"\r\nClosing the spy device ...\r\n");
        CloseHandle (hDevice);
        }
    else
        {
        _printf (L"Unable to open the spy device.\r\n");
        }
    if ((hControl != NULL) && gfSpyUnload)
        {
        _printf (L"Unloading the spy device ...\r\n");
        w2kServiceUnload (awSpyDevice, hControl);
        }
    return;
    }
```

列表 5-20. 程序的主框架

需要注意的是，**列表 5-20** 中的 Execute()函数在调用 CreateFile()时需要设置 GENERIC_READ 和 GENERIC_WRITE 访问标志，而**列表 4-29** 中的函数仅使用了 GENERIC_READ。产生这种不同的原因是，这些应用程序使用的 IOCTL 编码不同。第四章的内存 Spy 程序仅使用了只读的 IOCTL 函数，而这里讨论的 API Hook Viewer 调用的函数可以修改系统数据，因此它需要有写权限的设备句柄。如果你检查过**表 5-3** 的第三列的 IOCTL 编码，你会发现它们十六进制编码，大多数在右起第四个位置上都是 E，而 SPY_IO_HOOK_INFO 和 SPY_IO_HOOK_READ 却是数字 6。根据第四章的**图 4-6**，这意味着有后面的 Hook 管理函数需要一个读权限的设备句柄，而其余的需要读/写两种权限。设备驱动程序的设计者必须决定设备可处理的 I/O 请求所需的是读权限还是写权限，或者是二者的组合。修改系统的 API Service Table 肯定要进行写操作，所以此时客户端要求获取一个有写权限的句柄则是理所当然的了。

列表 5-20 中，剩余的大多数代码都很容易读懂。不过下面的特性值得重点说一下：

- SPY_IO_HOOK_READ 函数以行模式进行操作，这是由**列表 5-20** 中的循环开始时调用的 SpyHookRead()函数的第二个参数决定的。
- 应用程序的用户可以通过命令行提供一系列的模式字符串（通过嵌入的通配符*和?）。辅助函数 PatternMatcher()顺序的使用这些模式串和每一行协议数据中的函数名进行比较，**列表 5-21** 给出了该函数。如果没有可以匹配模式的函数名，那么该行协议数据将被忽略。要查看未过滤得协议数据，请使用命令行：w2k_hook *
- 在处理完一行协议数据后，应用程序将调用 sleep(0)进入休眠状态，并将它剩余的时间片归还给系统，这些时间片可供其他进程使用。
- 如果没有有效的协议数据，则应用程序会在休眠（suspend）10msec（HOOK_IOCTL_DELAY）后，再次向 Spy 设备进行查询。这样可及时地降低 CPU 的负载，并降低对 Native API 的使用率。
- 在**列表 5-20** 的主循环中，也会查询键盘输入。除 P、E、R 和 ESC 键外，其他的键都将被忽略。P 键用于打开/关闭暂停模式（默认为：打开），F 键允许/禁止过滤（默认：允许），R 键将重置协议，ESC 键则用来终止程序的运行。
- 如果 P、F、R 或 Esc 中的某一个键被按下，则一个分隔行将被写入协议缓冲区中（通过 SPY_IO_HOOK_WRITE 函数）。这个分隔行是用来表示根据输入的命令而进行的状态改变。向缓冲区中写入一个分隔行要好于直接在控制台中进行显示，因为要在屏幕上体现出状态的改变会有一定的延时。例如，如果 P 键被按下，此时应该停止显示了，但应用程序还是会继续生成输出数据直到从协议缓冲区中读取了所有的数据。但如果 P 命令在缓冲区的末尾增加一个分隔行，就可以正确的显示数据了。
- 和第四章的 w2k_mem.exe 程序类似，w2k_hook.exe 仅在全局标志 gfSpyUnload 被设置的情况下，才会卸载 Spy 驱动程序。默认情况下，该全局标志并未被设置，至于为什么，在第四章已经解释过了。

```
BOOL WINAPI PatternMatcher (PWORD pwFilter,
                            PWORD pwData)
    {
    DWORD i, j;
    i = j = 0;
    while (pwFilter [i] && pwData [j])
```

```
{
if (pwFilter [i] != '?')
    {
    if (pwFilter [i] == '*')
        {
        i++;
        if ((pwFilter [i] != '*') && (pwFilter [i] != '?'))
            {
            if (pwFilter [i])
                {
                while (pwData [j] &&
                    (!PatternMatcher (pwFilter + i,
                                      pwData      + j)))
                    {
                    j++;
                    }
                }
            return (pwData [j]);
            }
        }
    if ((WORD) CharUpperW ((PWORD) (pwFilter [i])) !=
        (WORD) CharUpperW ((PWORD) (pwData      [j])))
        {
        return FALSE;
        }
    }
i++;
j++;
}
```

```
if (pwFilter [i] == '*') i++;
return !(pwFilter [i] || pwData [j]);
}
```

列表 5-21. 一个简单的字符串模式匹配器

图 5-6 和**图 5-7** 中的例子，是 w2k_hook.exe 使用名字模式*file 和 ntclose 时生成的。这将筛选出所有的对文件管理函数的调用，同时还包括 NtClose()。要特别注意的是，名字模式串不影响协议数据的生成，它只在显示时对已生成的协议数据进行过滤，而 Spy 设备的“垃圾”过滤器是根据已注册的句柄来决定生成什么样的协议数据的。如果你指定 w2k_hook.exe 的名字模式串来排除某些协议项，那么是不会影响协议数据的生成的。唯一的影响就是，如果从协议缓冲区中取出的数据不符合你的要求，那么就丢掉而已。

5.3.2、总结和不足

Russinovich 和 Cogswell（Russinovich and Cogswell 1997）提出的 API Hooking 机制在此也是使用的，而且非常漂亮和具有独创性。下面就是此种机制中值得关注的特点：

- 在系统的 API Service Tabel 中安装和卸载一个 Hook，只是简单的指针交换操作。
- 安装完 Hook 后，它将可接收到系统中所有进程发出的 Native API 调用，即使是在 Hook 安装后才启动的进程。
- 因为 Hook 设备运行于内核模式，所以它有最大的权限来访问系统资源。甚至可以执行特权级的 CPU 指令。

下面是我在开发自己的 Spy device 时，遇到的问题：

- 必须十分小心的设计和编写 Hook device。因为在 Native API 一级产生的流量会经过多个程序的上下文空间，它必须向操作系统内核一样稳定才行。一个很小的疏忽也会使系统立即玩完。
- 仅有一小部分内核的 API 流量被记录下来。例如，由其他内核模式的模块发出的 API 调用不会经过系统的 INT 2Eh 门，因此，也不可能经过我们的 Hook。还有，ntdll.dll 和 ntoskrnl.exe 导出的很多重要函数并不属于 Native API，因此，对于它们，通过 Service Table 是无法进行 Hook 的。

不完善的 API 覆盖率带来更多的是限制而不是对稳定性的需求。总之，通过跟踪 Native API 调用能收集到如此多的有关程序内部信息的有用数据，还是很令人惊讶。例如，我可以

简单的通过观察由微软提供的 NetWare Redirector（nwrdr.sys）对 NtFsControlFile()的使用（traffic）来深入了解 NetWare Core Protocol（NCP）的运作。因此，这种监控 API 的方法的确是一种非常专业的备选方案，通过它我们可以从 Windows 2000 中获取我们感兴趣的数据。

六、在用户模式下调用内核 API 函数

在第二章，我解释了 Windows 2000 是如何通过中断门机制，来允许用户模式下的程序调用其内核 API 函数的一个子集----Native API 的。第四、五章中提到的程序依赖于设备 I/O 控制（IOCTL）来执行无法在用户模式下进行的附加任务。Native API 和 IOCTL 都很强大，但是如果可以像调用普通用户模式 DLL 中的函数一样来调用任何内核模式下的函数，我们将得到更强大的力量。一般看来这似乎不可能。不过，在本章中，我将演示使用一些疯狂的编程技巧，这将变得可能。IOCTL 将再次登场来解决我们首先碰到的看似无法解决的问题。本章将是革命性的，因为它在内核模式和用户模式间架设了一座桥梁，这样 Win32 应用程序就可以像调用 Win32 API 一样调用内核 API 函数了。更进一步，在随同 Windows 2000 调试工具一起提供符号文件的帮助下，应用程序可以调用对于内核驱动程序来说都无法使用的内核函数。这个“内核调用接口”可以完美的在后台运行，而几乎不被调用程序所察觉。

6.1、一个通用的内核调用接口

在第四章，我们使用内核模式的驱动程序来调用用户模式的程序所选择的内核 API 函数。例如，由 Spy 驱动程序 w2k_spy.sys 提供的 SPY_IO_PHYSICAL 函数只是内存管理函数---MmGetPhysicalAddress()的外包函数。另一个例子是 SPY_IO_HANDLE_INFO，它基于对象管理函数----ObReferenceObjectByHandl()和 ObDereferenceObject()。尽管这种技术可以很好的工作，但是为每个内核 API 设计这样一个自定义的 IOCTL 函数是枯燥和低效的。因此，我为 Spy 设备增加了一个通用的 IOCTL 函数，只需提供一个函数名或函数的入口点以及相应的参数列表，就可以调用任意的内核函数。这听起来似乎需要很多工作，但你一定会对实际所需的代码的简洁程度而感到惊讶。唯一的难点是，我们需要再次处理嵌入式汇编代码（Inline ASM）。

6.1.1、设计通向内核的门

如果运行于用户模式的程序想调用内核模式的函数，它就必须解决两个问题。首先，它必须以某种方式跳过用户态和内核态之间的障碍。其次，它必须将数据传入和传出。对于由 Native API 组成的子集，ntdll.dll 组件替我们完成了这一任务，它使用一个中断门来实现模

式的切换，并使用 CPU 寄存器来传入指向调用者参数堆栈的指针，函数的运行结果也是通过 CPU 寄存器返回给调用者的。对于非 Native API 的内核函数，操作系统没有提供这种门机制。因此，我们必须自己来实现一个这样的门。这一问题的一部分很容易解决：第四章介绍的 w2k_spy.sys 驱动程序在第五章经过扩展之后，在 IOCTL 事务中，可多次来回穿越内核模式和用户模式的边界。因为 IOCTL 允许双向的传输任意的数据块，数据传输问题就可以得到解决了。最后，整个问题可浓缩为如下几个简单的顺序步骤：

1. 用户模式的应用程序提交一个 IOCTL 请求，并传入调用该函数所需的信息，如，指向函数参数堆栈的指针。
2. 内核模式的驱动程序处理这一请求，并将参数复制到自己的堆栈中，然后调用制定函数，最后通过 IOCTL 的输出缓冲区将函数的执行结果返回给调用者。
3. 调用者取出 IOCTL 操作的结果，并像对普通的 DLL 函数调用一样进行处理。

这一模式的主要问题是：内核模式的模块必须处理多种数据格式和调用约定（calling convention）。下面是驱动程序必须准备的：

- 参数堆栈的大小依赖于目标函数。因为给驱动程序提供所有可能被调用的函数的细节信息是不切实际的，函数调用者必须提供参数堆栈的大小。
- Windows 2000 内核 API 函数使用三种调用约定：__stdcall、__cdecl 和__fastcall，它们处理参数的方式有很大的不同。__stdcall 和__cdecl 要求所有的参数都应传入到堆栈中，但是，__fastcall 为了使搜索参数堆栈的开销最小化，会将前两个参数传入 CPU 寄存器 ECX 和 EDX。从另一方面来看，__stdcall 和__fastcall 将参数从堆栈中移除的方式是一致的，这两种调用约定均强制被调用代码来完成这一工作。不过，对于__cdecl 来说，这一工作将由调用者完成。尽管通过保存堆栈指针（在调用和将堆栈指针恢复到其原始位置之前，来完成堆栈指针的保存）可以很容易的解决清除堆栈的问题，但不论采用什么样的调用约定，驱动程序对于__fastcall 约定都显得无能为力。因此，调用者必须针对每个调用来明确地指出其是否采用了__fastcall 调用约定，这样驱动程序才能确定是否需要 ECX 和 EDX 寄存器（如果必须的话）。
- Windows 2000 内核函数的返回值可以大小不同，其范围从 0---64bit 不等。返回值由 EDX:EAX 构成的 64 位寄存器对，来返回给调用者。Data is filled in from the least-significant end toward the most-significant end.例如，如果函数返回了一个 SHORT 类型的 16 位数据，则此时仅有 AX 寄存器（由 AL 和 AH 构成）是有意义

的。EAX 的高 16 位和整个 EDX 中的内容都是未定义的。因为驱动程序将忽略被调用函数的 I/O 数据，驱动程序必须假定最坏的情况，即数据大小为 64 位。否则，结果将可能被截断。

- 应用程序应能处理参数无效的情况。在用户模式下，这样做通常会使程序显得比较“友善”。在最坏的情况下，应用程序将被终止，并且弹出一个错误对话框。有时，这种错误需要通过重起计算机才能恢复。在内核模式下，在编程中最常见的错误是：“坏指针”，这样的错误将会立即导致系统蓝屏死机，这种情况下用户数据可能会丢失。通过使用操作系统提供的结构化异常处理机制（SEH）可以在很大的范围内找到这种错误。

这样看来，我们需要检查 Spy 驱动程序是如何处理函数的属性、参数和返回值的。**列表 6-1** 给出了 IOCTL 中涉及到的输入/输出结构----SPY_CALL_INPUT 和 SPY_CALL_OUTPUT。SPY_CALL_OUTPUT 结构非常简单，它包含一个 ULARGE_INTEGER 结构，Windows 2000 使用 ULARGE_INTEGER 结构来表示一个 64 位的值（即可以是一个单一的 64 位整型数据，也可以使一对 32 位的值）。请参考第二章中的**列表 2-3**，来了解这一结构的布局。

```
typedef struct _SPY_CALL_INPUT
    {
    BOOL    fFastCall;
    DWORD dArgumentBytes;
    PVOID pArguments;
    PBYTE pbSymbol;
    PVOID pEntryPoint;
    }
    SPY_CALL_INPUT, *PSPY_CALL_INPUT, **PPSPY_CALL_INPUT;
#define SPY_CALL_INPUT_ sizeof (SPY_CALL_INPUT)
// -------------------------------------------------------------
typedef struct _SPY_CALL_OUTPUT
    {
    ULARGE_INTEGER uliResult;
```

```
    }
    SPY_CALL_OUTPUT, *PSPY_CALL_OUTPUT, **PPSPY_CALL_OUTPUT;
#define SPY_CALL_OUTPUT_ sizeof (SPY_CALL_OUTPUT)
```

列表 6-1. SPY_CALL_INPUT 和 SPY_CALL_OUTPUT 结构的定义

SPY_CALL_INPUT 需要稍微说明一下。其 fFastCall 成员的含义是显而易见的。它通知 Spy Driver 调用函数时将遵守__fastcall 调用约定。因此，调用函数时所需的前两个参数（如果有的话）不能通过堆栈来传入，而是应该通过 CPU 寄存器来进行传递。dArgumnetBytes 记录了压入参数堆栈中的字节数，pArguments 指向参数堆栈的顶部。剩下的两个成员---pbSymbol 和 pEntryPoint 是互斥的，它们用于告诉驱动程序应该执行那个函数。你可以指定一个函数名或者一个无格式的入口地址。其余的成员应该总是被设置为 NULL。如果 pbSymbol 和 pEntryPoint 都不为 NULL，那么 pbSymbol 将优先于 pEntryPoint 被采用。通过函数名进行调用要比通过入口地址多出一步，多出的这一步用于确定函数名的入口地址。如果该地址可以获取，则将通过此地址调用该函数。直接传入一个函数入口点则将绕过名称解析步骤。

找到内核模块所导出的符号（symbol）的入口地址其实只是听其来比较容易而已。Win32 函数 GetModuleHandle()和 GetProcAddress()可以很好的与 Win32 子系统中的所有组件一起工作，但它们不能识别内核模式下的系统模块和驱动程序。实现这一部分的示例代码将很困难，其实现细节将涉及到本章的下一节。现在，让我们假定可以得到一个有效的入口点指针，先不要去关心它是如何得到的。**列表 6-2** 给出了 SpyCall()函数是我的内核调用接口中的核心部分。正如你所见，它几乎 100%都是汇编语言。在 C 程序中借助于汇编总不是一件很容易的事，但是有些任务如果用纯粹的 C 语言将很难简单的完成。在这里，我们的问题是 SpyCall()需要完全控制堆栈和 CPU 寄存器，因此它必须绕过 C 编译器和优化器，因为它们会按照自己的方式来使用堆栈和寄存器。

在深入研究 SpyCall()函数的细节之前，让我先描述一下 SpyCall()函数的另一个特性，这一特性使得代码看起来更加晦涩。就像在第二章解释的那样，Windows 2000 系统模块按名称导出了其内部的一些变量。典型的例子是 NtBuildNumber 和 KeServiceDescriptorTable。Windows 2000/NT/9x 使用的 PE 文件提供了一种通用的机制来将符号名和地址关联起来，而不管地址指针是指向代码还是数据。因此，一个 Windows 2000 模块自由的将它的导出符号和它的任意一个全局变量关联起来。客户端模块可以和它们进行动态链接，就像链接到函数

的符号名上一样，然后客户端就可以使用这些变量，就好像这些变量位于自己的全局数据段中一样。当然，我的内核调用接口也可以很好的处理此种类型的符号，因此，我决定如果 SPY_CALL_INPUT 结构中的 dArgumentBytes 成员为负值，则表示从入口地址复制数据而不是调用该入口地址。有效值的范围从-1 到-9，这里，-1 意味着将入口地址自身复制到 SPY_CALL_OUTPUT 缓冲区中。对于剩下的值，它们的补码将用于表示应该从入口地址复制的字节数，这意味着，-2 表示复制一个字节；-3 表示一个 16 位 WORD 或 SHORT；-5 表示一个 32 位 DWORD 或 LONG；-9 表示一个 64 位 DWORDLONG 或 LONGLONG。你可能会很困惑：为什么必须复制入口地址自身？因为，有些内核符号，如 KeServiceDescriptorTable 指向的结构体大于 64 位，而返回值的大小是不能超过 64 位的，所以，最佳的办法是返回一个无格式的指针而不是将返回值截断为 64 位。

```
void SpyCall (PSPY_CALL_INPUT    psci,
                    PSPY_CALL_OUTPUT psco)
    {
    PVOID pStack;
    __asm
        {
        pushfd
        pushad
        xor         eax, eax
        mov         ebx, psco                       ; get output parameter block
        lea         edi, [ebx.uliResult]        ; get result buffer
        mov         [edi    ], eax                  ; clear result buffer (lo)
        mov         [edi+4], eax                    ; clear result buffer (hi)
        mov         ebx, psci                       ; get input parameter block
        mov         ecx, [ebx.dArgumentBytes]
        cmp         ecx, -9                         ; call or store/copy?
        jb          SpyCall2
        mov         esi, [ebx.pEntryPoint]    ; get entry point
        not         ecx                             ; get number of bytes
        jecxz       SpyCall1                        ; 0 -> store entry point
```

```
        rep     movsb                   ; copy data from entry point
        jmp     SpyCall5
SpyCall1:
        mov     [edi], esi              ; store entry point
        jmp     SpyCall5
SpyCall2:
        mov     esi, [ebx.pArguments]
        cmp     [ebx.fFastCall], eax    ; __fastcall convention?
        jz      SpyCall3
        cmp     ecx, 4                  ; 1st argument available?
        jb      SpyCall3
        mov     eax, [esi]              ; eax = 1st argument
        add     esi, 4                  ; remove argument from list
        sub     ecx, 4
        cmp     ecx, 4                  ; 2nd argument available?
        jb      SpyCall3
        mov     edx, [esi]              ; edx = 2nd argument
        add     esi, 4                  ; remove argument from list
        sub     ecx, 4
SpyCall3:
        mov     pStack, esp             ; save stack pointer
        jecxz   SpyCall4                ; no (more) arguments
        sub     esp, ecx                ; copy argument stack
        mov     edi, esp
        shr     ecx, 2
        rep     movsd
SpyCall4:
        mov     ecx, eax                ; load 1st __fastcall arg
        call    [ebx.pEntryPoint]       ; call entry point
```

```
        mov     esp, pStack                     ; restore stack pointer
        mov     ebx, psco                       ; get output parameter block
        mov     [ebx.uliResult.LowPart ], eax   ; store result (lo)
        mov     [ebx.uliResult.HighPart], edx   ; store result (hi)
SpyCall5:
        popad
        popfd
        }
    return;
    }
```

列表 6-2. 内核调用接口的核心函数

对于访问导出变量的这一特殊情况必须要牢记。这样列表 6-2 中的代码将不再是特别难以理解的了。首先，64 位的结果缓冲区将被清空，以保证未使用的位总是零。接下来，输入数据的 dArgumentBytes 成员将和-9 比较，以确定客户端是请求一个函数调用还是一个数据复制操作。函数调用处理代码从 SpyCall2 标签处开始。通过计算 pArgumnets 成员的值，将 ESI 寄存器指向参数堆栈的顶部，接下来，检查调用约定。如果需要__fastcall 并且在堆栈中至少有一个 32 位值，则 SpyCall()将该值从堆栈中移除，并将其临时保存在 EAX 寄存器中。如果堆栈中还有另一个 32 位值，它也将被移除并保存到 EDX 寄存器中。剩余的参数则全部保留在堆栈中。此时，将到达 SpyCall3 标签处。现在堆栈的当前栈顶地址将被保存到局部变量 pStack 中，然后使用 i386 的 REP MOVSD 指令将参数堆栈（不包括在__fastcall 情况下已经移除的参数）复制到 Spy driver 自己的堆栈中。注意，方向标志用于确定 MOVSD 在内存中是向上移动还是向下移动，可以假定 Spy driver 堆栈默认已被清空。这意味着，ESI 和 EDI 寄存器在复制的每个阶段之后都会被累加。现在，在执行 CALL 指令之前，唯一剩下需要做的就是将__fastcall 的第一个参数从它的临时位置 EAX 复制到其最终位置---ECX。SpyCall()将直接复制 EAX 到 EXC，因为即使调用约定是__stdcall 或__cdecl，直接复制也不会产生严重错误。MOV ECX, EAX 执行速度很快，直接执行这一指令将比先测试 fFastCall 成员然后再进行处理会更快一些。

在对函数入口点的调用返回之后，SpyCall()将堆栈指针恢复到 pStack 所指的位置。需要注意的是，__stdcall 和__fastcall 采用与__cdecl 不同的堆栈清理策略。一个__cdecl 调用在

返回时，会将 ESP 寄存器指向参数堆栈的栈顶，而__stdcall 和__fastcall 则将 ESP 恢复到其调用前的位置。强制将 ESP 恢复到其先前位置，总是可以很优雅的清除参数堆栈，而且这样做也不需要关心采用的是那种调用约定。SpyCall()中最后几行汇编代码用于将函数执行结果保存到 EDX:EAX，以返回给调用者的 SPY_CALL_OUTPUT 结构。这里不需要知道结果的确切大小。这并不是必须的，因为调用者明确的知道它所期望的有效位的个数。复制过多的位不会产生 Bug，多出的位将会被调用者忽略。

对于**列表 6-2** 中代码的，有件事也应该注意，即这些代码不会阻止无效的参数。它甚至不检查堆栈指针本身是否有效。在内核模式下，这等同于玩火。不过，Spy driver 该如何验证所有参数的有效性呢？堆栈中的一个 32 位值也许是一个计数器、一个位域数组或者可能是一个指针。只有调用者和被调用的目标函数知道参数的确切含义。SpyCall()函数只是一个简单的通过层，它并不知道它传递的数据的类型。向函数中增加对上下文敏感的参数的检查等同于重写了操作系统的一大部分。幸运的是，Windows 2000 提供一中简单的方法来完成这一任务：结构化异常处理（Structured Exception Handling, SEH）。

SHE 是一个非常易于使用的框架，它允许程序捕获可能会引起系统崩溃的异常。一个异常是指一个非正常的状态，它会强制 CPU 停止工作，而不管 CPU 正在做什么。产生异常的典型操作是：从一个无效线性地址读取或写入数据（这里的无效线性地址指的是没有映射到物理内存或页面文件的线性地址）、向代码段中写入数据、试图在数据段中执行指令或者除数为零。有些个别异常是良性的，例如，如果要访问的内存已经被置换到页面文件中，则也会产生一个异常，操作系统通过将目标页再次调入内存来解决这一异常。不过，大多数异常都是致命的，因为操作系统不知道如何从异常中恢复过来，因此系统就简单的将自己 shutdown 来表示自己的不满。这种反应似乎有些过于激进，不过，在事情变得更严重之前，将系统挂起还是一个比较好的选择。通过使用 SEH，产生异常的程序将获得一个机会来处理此异常。使用微软专用的__try/__except，可以监控一段任意代码中可能产生的异常，如果一个异常将系统引入了临界状态，那么一个自定义的处理例程（位于用户自己的程序中）将被调用，这样就允许程序员提供一个比蓝屏更好的处理方法。

显然，SHE 也可以完成我的 Spy device 所需的参数有效性验证问题。**列表 6-3** 给出了一个将 SpyCall()放入 SHE 帧中的外包函数。被保护的代码位于__try 语句之后。当然，保护的不仅是 SpyCall()；所有在调用的上下文环境中被执行的代码都会得到保护。如果抛出了一个异常，则将执行__except 语句后面的代码,当然这一异常要满足过滤表达式 EXCEPTION_EXECUTE_HANDLER。**列表 6-3** 中的异常处理例程没有太大价值。它仅会使

SpyCallEx()返回状态代码：STATUS_ACCESS_VIOLATION，而不是正常的STATUS_SUCCESS，这一状态代码仅会使在用户模式下对 DeviceIoControl()的调用失败，而不是出现蓝屏。在异常发生之后，唯一存在的问题是：被调用函数的返回值并未定义。不过这是调用者应该处理得事情。

```
NTSTATUS SpyCallEx (PSPY_CALL_INPUT    psci,
                            PSPY_CALL_OUTPUT psco)
    {
    NTSTATUS ns = STATUS_SUCCESS;
    __try
          {
          SpyCall (psci, psco);
          }
    __except (EXCEPTION_EXECUTE_HANDLER)
          {
          ns = STATUS_ACCESS_VIOLATION;
          }
    return ns;
    }
```

列表 6-3. 在内核调用接口中使用结构化异常

尽管 SEH 可以捕获大多数常见的参数错误，但是，你不能指望它可以阻止客户端程序可能传递给内核 API 函数的所有“垃圾”。某些糟糕的函数参数虽然并不会使系统崩溃，但却仍在悄无声息的破坏着系统。例如，一个函数在复制一个字符串时，如果指定了错误的目标地址指针，则可以很容易的覆盖掉系统内存的关键部分。可能很长时间都无发发现这种 Bug，直到执行到被覆盖的内存区域后，系统突然意外的玩完了，我们才可能发觉。在测试 Spy driver 的过程中，有时，我使测试程序发出的 IOCTL 调用（针对 spy device）处于挂起状态。测试程序没有任何反应，甚至拒绝从内存中移除。更糟的是，系统会变得无法关闭。这和蓝屏一样让人厌烦。

6.2、在运行时链接到系统模块

在实现了基本的内核调用接口之后，接下来的问题是将符号化的函数名解析为线性地址，这一地址时 CALL 指令所需要的（见**列表 6-2**）。这一步很重要，因为你不能确定内核 API 函数的入口地址在很长一段时间内都不发生变化。只要可能，就应该按照名称调用函数。按照地址调用系统函数只是一种例外，典型的受限函数是指那些目标模块没有导出的函数（但它们确实存在）。在大多数情况下，使用符号化的名字更为清晰，这些符号化的名字位于模块的导出节（export section）中。

6.2.1、在 PE 映像中查找导出的符号名

对一个 Win32 程序员来说，在运行时链接到 DLL 的导出函数几乎每天都会碰到。例如，如果你想编写一个使用了 Windows 2000 某些增强特性（译注：这些特性在旧版本中也存在，只不过 Windows 2000 对其进行了某种程度的增强而已。）的 DLL，而且还要求在一些老的操作系统（如 Windows 95 或 98）中也能运行，此时，你应该在运行时链接到 DLL 的导出函数（指仅在 Windows 2000 中有效的函数），并且，如果这些函数无法使用（指，此时使用的系统可能是 Windows 9x），则应该“悄无声息”的采用默认的行为（意即：此时，不要做弹出一个出错对话框等类似的操作，这里的默认行为指的是使用旧版本中的功能）。在这种情况下，你应该调用 GetModuleHandle()函数（如果 DLL 已经位于内存中，并且可以保证在足够长的时间内该 DLL 都是有效的），或者调用 LoadLibrary()函数（如果 DLL 还没有加载到内存中，或者为了防止过早的卸载该 DLL）。返回的模块句柄可以在多次 GetProcAddress()调用中被重复使用，这样就可以找出应用程序所需的 DLL 的导出函数的所有入口地址。理论上来讲，可以使用相同的方法来调用 ntoskrnl.exe、hal.dll 或者其他系统模块导出的内核函数。不过，事实上这些系统模块导出的内核函数没有一个能在这种情况下工作！GetModuleHandle()将报告：“此模块还未加载”，并且，如果你强行向 GetProcAddress()传入一个硬编码（hard-coded）的系统模块句柄，则 GetProcAddress()将返回 NULL，例如，ntoskrnl.exe 的句柄 HMODULE 为 0x80400000。进一步来来看的话，这也是合理的。这些模块被设计为运行于用户模式的 Win32 组件，因此可以将它们加载到 4GB 线性地址空间中的低 2GB 地址空间中。

如果 Win32 子系统不知道模块是否位于内核空间，那么理论上，我们就可以使内核驱动程序按照我们的想法工作了---这就是贯穿本书的思路。未文档化的 MmGetSystemRoutineAddress()函数（由 ntoskrnl.exe 导出），显然就可以完成这一工作，不过，很不走运，该函数不支持 Windows NT 4.0。因为本书所给出示例代码的一大前提就是在兼容 Windows 2000 先前版本的基础上，提供最大的扩展性，所以，我放弃使用这一函数，转而寻找一种不需要系统帮助就能确定函数入口地址的方法。Windows 2000 运行时库仅为映像文件的解析提供了有限的支持，如未文档化的 RtlImageNtHeader()函数，**列表 6-4** 给出了该函数的原型。该函数可将模块映像的基地址映射到线性地址空间中（即，一个指向 IMAGE_DOS_HEADER 结构的指针，该结构体定义于 Win32 SDK 头文件 winnt.h 中）并返回一个指向 PE 表头（PE Header）的指针，DOS 表头的 e_Ifanew 成员指出了 PE 表头的相对偏移量（译注，这里的 DOS 表头指的是 IMAGE_DOS_HEADER 结构，该结构体定义于 winnt.h 中，具体细节参见后面的译注）。使用 RtlImageNtHeader()函数必须小心，因为它并对传入的指针的有效性进行全面的检查。它仅测试指针是否为 NULL 或 0xFFFFFFFF，并验证指针指向的内存块的起始处包含 MZ 标志。这意味着，如果你传入一个伪造的地址（该地址不为 NULL 或 0xFFFFFFFF），那么，当 RtlImageNtHeader()读取 DOS header 标志时将会立即触发蓝屏。奇怪的是，Windows NT 4.0 将这部分代码包含在一个 SEH 帧中，而 Windows 2000 却没有这样做。

```
PIMAGE_NT_HEADER    NTAPI    RtlImageNtHeader(PVOID    base);
```

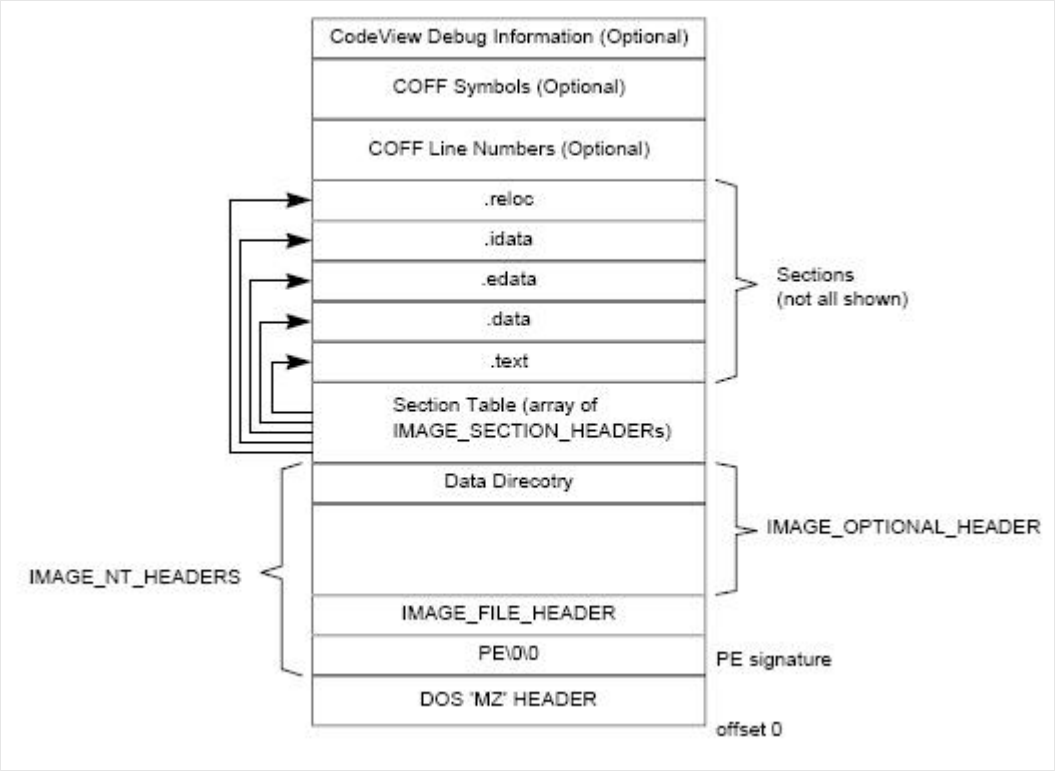
列表 6-4.　RtlImageNtHeader()的原型

译注：

PE 表头（PE Header）并非在 PE 文件的最开始位置。PE 文件最前面的数百个位是所谓的 DOS Stub（即，DOS Header）：这是一个极小的 DOS 程序，它用来输出像“This Program cannot be run in DOS mode”这样的信息。这里的 DOS Stub 实际上是一个 IMAGE_DOS_HEADER 结构，该结构定义于 winnt.h 中。DOS 表头的 e_Ifanew 成员实际是一个相对偏移值（RVA）。下面的公式可以获取实际的 PE 表头的地址：

pNTHeader = dosHeader + dosHeader->e_Ifanew;

下面是 PE 表头的大致布局：



有关 PE 文件结构的具体细节，请参考《Windows 95 System Programming SECRETS》一书的第八章 PE 与 COFF OBJ 格式。

列表 6-4 给出的 RtlImageNtHeader()函数返回一个指向 IMAGE_NT_HEADER 结构的指针。有关 PE 文件的结构体均定义于 winnt.h 中。很不幸，在 DDK 头文件没有这些结构体的定义，因此这些定义必须手工添加进来。我的 Spy driver 在它的头文件 w2k_spy.h 中包含了它用来查找符号的结构体定义（见**列表 6-5**）。IMAGE_NT_HEADERS 由 PE 标志”PE\0\0”、一个 IMAGE_FILE_HEADER 结构和一个 IMAGE_OPTIONAL_HEADER 结构顺序连接而成。IMAGE_OPTIONAL_HEADER 结构的最后一部分是一个 IMAGE_DATA_DIRECTORY 类型的数据，可使用该数组来快速的查找文件的节（section）。数组的第一项为 IMAGE_DIRECTORY_ENTRY_EXPORT（参见**列表 6-5**，该数组的第二个元素指向导入函数表---imported functions table），它指向文件的导出节（也称作导出函数表），在导出节中包含模块导出的所有函数的名字和其相对地址。我们就是在导出节中查找传递给内核调用接口的函数的名字，从而计算出它们的入口地址。

```
#define IMAGE_DIRECTORY_ENTRY_EXPORT          0
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3
```

```
#define IMAGE_DIRECTORY_ENTRY_SECURITY                4
#define IMAGE_DIRECTORY_ENTRY_BASERELOC               5
#define IMAGE_DIRECTORY_ENTRY_DEBUG                   6
#define IMAGE_DIRECTORY_ENTRY_COPYRIGHT               7
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR               8
#define IMAGE_DIRECTORY_ENTRY_TLS                     9
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG            10
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT           11
#define IMAGE_DIRECTORY_ENTRY_IAT                    12
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT           13
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR         14
 #define IMAGE_NUMBEROF_DIRECTORY_ENTRIES            16
 #define IMAGE_NUMBEROF_DIRECTORY_ENTRIES            16
 // ====================================================================
 // ====================================================================
// WINDOWS 2000 IMAGE STRUCTURES
// ====================================================================
 typedef struct _IMAGE_FILE_HEADER
 typedef struct _IMAGE_FILE_HEADER
      {
      WORD    Machine;
      WORD    NumberOfSections;
      DWORD TimeDateStamp;
      DWORD PointerToSymbolTable;
      DWORD NumberOfSymbols;
      WORD    SizeOfOptionalHeader;
      WORD    Characteristics;
      }
      IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

```
// ------------------------------------------------------------
 typedef struct _IMAGE_DATA_DIRECTORY
 typedef struct _IMAGE_DATA_DIRECTORY
      {
      DWORD VirtualAddress;
      DWORD Size;
      }
      IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
// ------------------------------------------------------------
 typedef struct _IMAGE_OPTIONAL_HEADER
 typedef struct _IMAGE_OPTIONAL_HEADER
      {
      WORD                 Magic;
      BYTE                 MajorLinkerVersion;
      BYTE                 MinorLinkerVersion;
      DWORD                SizeOfCode;
      DWORD                SizeOfInitializedData;
      DWORD                SizeOfUninitializedData;
      DWORD                AddressOfEntryPoint;
      DWORD                BaseOfCode;
      DWORD                BaseOfData;
      DWORD                ImageBase;
      DWORD                SectionAlignment;
      DWORD                FileAlignment;
      WORD                 MajorOperatingSystemVersion;
      WORD                 MinorOperatingSystemVersion;
      WORD                 MajorImageVersion;
      WORD                 MinorImageVersion;
      WORD                 MajorSubsystemVersion;
```

```
    WORD                    MinorSubsystemVersion;
    DWORD                   Win32VersionValue;
    DWORD                   SizeOfImage;
    DWORD                   SizeOfHeaders;
    DWORD                   CheckSum;
    WORD                    Subsystem;
    WORD                    DllCharacteristics;
    DWORD                   SizeOfStackReserve;
    DWORD                   SizeOfStackCommit;
    DWORD                   SizeOfHeapReserve;
    DWORD                   SizeOfHeapCommit;
    DWORD                   LoaderFlags;
    DWORD                   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory
                            [IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
    }
    IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
// -------------------------------------------------------------
 typedef struct _IMAGE_NT_HEADERS
 typedef struct _IMAGE_NT_HEADERS
    {
    DWORD                   Signature;
    IMAGE_FILE_HEADER       FileHeader;
    IMAGE_OPTIONAL_HEADER OptionalHeader;
    }
    IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
 // -------------------------------------------------------------
 // -------------------------------------------------------------
 typedef struct _IMAGE_EXPORT_DIRECTORY
```

```
typedef struct _IMAGE_EXPORT_DIRECTORY
    {
    DWORD Characteristics;
    DWORD TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD Name;
    DWORD Base;
    DWORD NumberOfFunctions;
    DWORD NumberOfNames;
    DWORD AddressOfFunctions;
    DWORD AddressOfNames;
    DWORD AddressOfNameOrdinals;
    }
    IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

列表 6-5. PE 文件基本结构的子集

PE 文件中的导出节（export section）的布局，是由 IMAGE_EXPORT_DIRECTORY 结构来管理的，可在**列表 6-5** 的底部找到 IMAGE_EXPORT_DIRECTORY 结构体的定义。一个导出节基本上包含一个由 IMAGE_EXPORT_DIRECTORY 结构体构成的表头，再加上三个大小可变的数组和一个存放以零结尾的 ANSI 字符串的数组。通常情况下，一个导出项由如下三个部分构成：

1. 一个以零结尾的名称，由 8 个 ANSI 字符组成。
2. 一个 16 位的序列号
3. 一个 32 位的相对偏移量（针对文件映像的基地址）

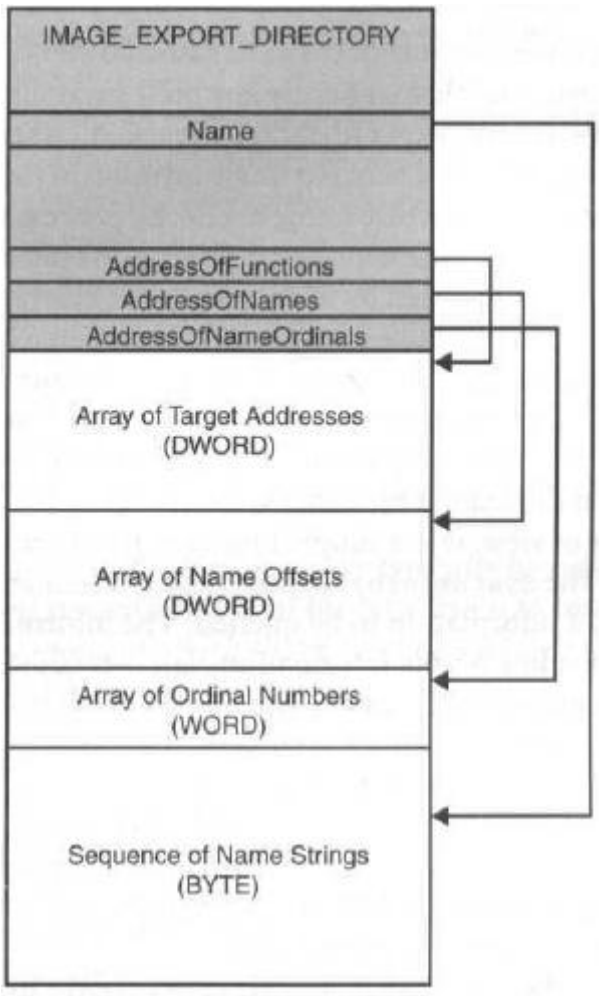


图 6-1. PE 文件导出节的典型布局

导出机制并不仅适用于函数。它只是为 PE 映像中的地址提供了符号化的名称。对于函数来说，其符号（即函数名）将与其入口地址相关联。对于公开的变量，其符号（即变量名）将指向该变量的基地址。可通过用符号的特征参数填充三个平行的数组来完成符号和地址的关联过程。在**图 6-1** 中，这三个平行的数组将作为目标地址数组、名称偏移量数组和序列号数组来引用。它们分别对应 IMAGE_EXPORT_DIRECTORY 结构中的 AddressOfFunctions、AddressOfNames 和 AddressOfNameOrdinals 成员。这三个结构体成员中保存的都是相对于映像文件基地址的偏移量（即，相对偏移量）。IMAGE_EXPORT_DIRECTORY 结构的 Name 成员包含文件自身名称字符串的相对偏移量。如果可执行文件被重新命名，可使用 Name 来找回文件的原始名称。**图 6-1** 只是导出节的一个常见布局，符号和名称数组（Name string sequence）的位置并不是固定的。PE 文件编写者可以按照他们的喜好来移动它们，只要 IMAGE_EXPORT_DIRECTORY 结构的成员可以正确的引用它们。对于 Name 成员指向的文件字符串来说也是一样的，尽管，一般情况下，文件自身的名称总是位于名称数组中的开始位置，但这并不是必须的。记住，永远不要假定导出节中变量的位置。

IMAGE_EXPORT_DIRECTORY 结构的 NumberOfFunctions 和 NumberOfNames 成员指出了 AddressOfFunctions 和 AddressOfNames 所引用的数组各包含了多少项。没有针对 AddressOfNameOrdinals 所引用的数组的计数，因为该数组所包含的项数总是和 AddressOfNames 所引用的数组相同。独立维护地址和名称数组的项数暗示着，可以建立这样一个可执行文件，该可执行文件导出了没有名称的地址。不过，到目前为止，我还没有见过这样的文件，但是在访问这些数组时，还是应该记住这种可能性的存在。再次强调，不要依赖于假设。

按名称查找导出函数或变量所对应的地址的，可按照如下步骤进行，首先，需要一个模块基地址（在 Win32 中，就是 HMODULE）：

1. 使用模块基地址调用 RtlImageNtHeader()，以获取模块的 IMAGE_NT_HEADERS。如果函数返回 NULL，则说明地址没有指向一个有效的 PE 映像。
2. 用 IMAGE_DIRECTORY_ENTRY_EXPORT 作为索引，来访问 OptionalHeader 的 DataDirectory 成员，以找出导出节（export section）的相对偏移量。
3. 通过计算 IMAGE_EXPORT_DIRECTORY 表头的 AddressOfNames 成员，来定位导出节中的名称数组（即图 6-1 中的 Sequence of Name Strings）。
4. 遍历名称数组，直到找到匹配的项，或者到达数组末尾（此时表示已到达 NumberOfNames）。
5. 如果找到了匹配项，则可到 AddressOfNameOrdinals 所引用的数组的对应位置读取该名字的序列号。该数组是从 0 开始计数的，所以，可以使用读取到的序列号作为索引来直接访问 AddressOfFunctions 所引用的数组。
6. 将模块的基地址与 AddressOfFunctions 所引用的数组中取出的偏移量相加，即可得到指定名称（函数或者变量）的线性地址。

上述步骤看似简单清晰。但，还存在一个未知量：模块的基地址。尽管上述操作基本上反映出了 GetProcAddress()函数的基本行为，但是要找到模块的地址，则意味着需要模仿 GetModuleHandle()的行为。如果你查看 ntoskrnl.exe 的导出函数，你会发现没有函数可以完成这一工作。这是因为 Windows 2000 内核提供了一个更强大的函数来完成上述工作以及其他任务，这些任务包括：访问系统内部数据。这个函数就是：NtQuerySystemInformation()。

6.2.2、将系统模块和驱动程序加载到内存中

NtQuerySystemInformation()是 Windows 2000 系统编程中主要 API 函数之一，几乎所有内建的管理工具都使用了该函数，但是你不会在 DDK（Device Driver Kit）文档的任何地方找到它。唯一提及该函数的就是 ntddk.h 中 CONFIGURATION_INFORMATIONS 结构的注释，这证实了这一函数的存在。如果存在“无正式文档系数”的话，并且按照在微软文档中出现频率来划分函数的有用程度，那么 NtQuerySystemInformation()将当仁不让的位居榜首。随同很多其他让人振奋的功能，该函数还可返回已加载的系统模块的列表，这包括所有的系统核心组件和内核模式的驱动程序。

Spy driver 的源文件中包含最少的直接代码和类型定义，以从 NtQuerySystemInformation()获取已加载模块的列表。从调用者角度来看，这是一个非常简单的函数。该函数需要四个参数，如**列表 6-6** 所示。SystemInformationClass 是一个从 0 开始的数值，它用来指定要查询的信息的类型。这里的 Information 可以是变长的，其具体大小依赖于所查询的信息的类型，查询到的信息将被复制到调用者提供的 SystemInformation 指向的缓冲区中。缓冲区的长度由 SystemInformationLength 参数指定。如果调用成功，复制到缓冲区的实际字节数将被写入 ReturnLength 指向的变量中。这个函数的问题是，在它发现缓冲区太小的情况下，它不会报告它实际想要复制多少字节。因此，调用者必须在一个循环中不断尝试，直到函数的返回代码从 STATUS_INFO_LENGTH_MISMATCH（0xC0000004）变为 STATUS_SUCCESS（0x00000000）。

```
NTSTATUS NTAPI ZwQuerySystemInformation( DWORD SystemInformationClass,
                                        PVOID    SystemInformation,
                                        DWORD SystemInformationLength,
                                        PDWORD ReturnLength);
```

列表 6-6. NtQuerySystemInformation()的原型

列表 6-6 没有给出 NtQuerySystemInformation()自身，但给出了该函数的另一个“兄弟”：ZwQuerySystemInformation()，这一函数除了函数名前缀不同之外，其运行机理与 NtQuerySystemInformation()是相同。你或许还记得第二章中的 Nt*和 Zw* Native API 函数集合。如果从用户模式进行调用的话，这两种函数的工作方式非常相似，在用户模式下，这两组函数都将通过 ntdll.dll 到达相同的 INT 2Eh Stub。不过，在内核模式下，情形就有些不同了。此时，ntoskrnl.exe 将控制对 Native API 的调用，Nt*()和 Zw*()的执行路径将不再相同。

Zw*()函数还是通过 INT 2Eh 中断门，这和 ntdll.dll 的处理方式相同。而 Nt*()却绕过了此中断门。在 DDK 文档的术语表中，微软是这样描述 Zw*()函数集的（Microsoft 2000f）：

“一组与执行体的系统服务（executive’s system services）平行的入口点。从内核模式的代码（）中调用一个 ZwXxx 入口点将获得相应的系统服务，只是在使用 Zw()函数时，不会检查调用者的访问权限和参数的有效性，而且调用不会将先前模式（previous mode）切换到用户模式”（Windows 2000 DDK\Kernel-Mode Drivers\Design Guide\Kernel-Mode Glossary \Z\Zw routines.）*

上文最后一句中提到的“先前模式（previous mode）”非常重要。Peter G. Viscarola 和 W. Anthony Mason：

“尽管任意一组函数都可以从内核模式调用，但如果用 Zw*()函数来代替 Nt*()函数，则可将先前模式（此后的模式才是请求被发出的模式）切换到内核模式。”（Viscarola 和 Mason 1999，p.18）

对先前模式（previous-mode）的处理带来的副作用是，在没有任何附加预防措施的情况下，从内核模式的驱动程序中调用 NtQuerySystemInformation()函数将返回一个出错状态代码：STATUS_ACCESS_VIOLATION（0xC0000005），而对 ZwQuerySystemInformation()的调用则可成功，或者返回 STATUS_INFO_LENGTH_MISMATCH。

在**列表 6-7** 中给出了 SystemInformationClass 所需的常量和类型的定义。已加载模块的列表将通过一个 MODULE_LIST 结构返回，每个模块均包含一个 32 位的模块计数和一个 MODULE_INFO 类型的数组。

```
#define SystemModuleInformation 11 // SYSTEMINFOCLASS
typedef struct _MODULE_INFO
    {
    DWORD dReserved1;
    DWORD dReserved2;
    PVOID pBase;
    DWORD dSize;
    DWORD dFlags;
    WORD    wIndex;
    WORD    wRank;
```

```
    WORD    wLoadCount;
    WORD    wNameOffset;
    BYTE    abPath [MAXIMUM_FILENAME_LENGTH];
    }
    MODULE_INFO, *PMODULE_INFO, **PPMODULE_INFO;
#define MODULE_INFO_ sizeof (MODULE_INFO)
// -------------------------------------------------------------
typedef struct _MODULE_LIST
    {
    DWORD           dModules;
    MODULE_INFO aModules [];
    }
    MODULE_LIST, *PMODULE_LIST, **PPMODULE_LIST;
#define MODULE_LIST_ sizeof (MODULE_LIST)
```

列表 6-7. SystemModuleInformation 定义

现在调用 ZwQuerySystemInformation()所需的一切都已准备好。**列表 6-8** 给出了 SpyModuleList()函数的实现方式，该函数使用一个 trial-and-error 循环（指，出错-尝试方式的循环，在第一章提及过），和两个简单的内存管理函数---SpyMemoryCreate()和 SpyMemoryDestroy()，这两个函数内部将调用 Windows 2000 执行体函数（Executive function）ExAllocatePoolWithTag()和 ExFreePool()。SpyModuleList()函数在开始时将使用 4,096Byte 的缓冲区，如果 ZwQuerySystemInformation()的返回值为 STATUS_INFO_LENGTH_MISMATCH，则将缓冲区扩大一倍，然后再次尝试调用 ZwQuerySystemInformation()。如果 ZwQuerySystemInformation()返回了其他的值，将终止循环。SpyModueList()的可选参数 pdData 和 pns，将关返回更详细的信息。如果 SpyModueList()返回 NULL，则表示调用失败，此时 pns 指向的缓冲区中将保存一个错误代码，*pdData 将被设为 0。如果 SpyModueList()调用成功，*pdData 将保存复制到缓冲区中的字节数，*pns 的值将为 STATUS_SUCCESS。

```
#define SPY_TAG                 '>YPS'                  // SPY>
PVOID SpyMemoryCreate (DWORD dSize)
```

```
    {
    return ExAllocatePoolWithTag (PagedPool, max (dSize, 1),
                                            SPY_TAG);
    }
// -------------------------------------------------------------
PVOID SpyMemoryDestroy (PVOID pData)
    {
    if (pData != NULL) ExFreePool (pData);
    return NULL;
    }
// -------------------------------------------------------------
PMODULE_LIST SpyModuleList (PDWORD       pdData,
                                    PNTSTATUS pns)
    {
    DWORD           dSize;
    DWORD           dData = 0;
    NTSTATUS        ns      = STATUS_INVALID_PARAMETER;
    PMODULE_LIST pml     = NULL;
    for (dSize = PAGE_SIZE; (pml == NULL) && dSize; dSize <<= 1)
        {
        if ((pml = SpyMemoryCreate (dSize)) == NULL)
            {
            ns = STATUS_NO_MEMORY;
            break;
            }
        ns = ZwQuerySystemInformation (SystemModuleInformation,
                                            pml, dSize, &dData);
        if (ns != STATUS_SUCCESS)
            {
```

```
            pml     = SpyMemoryDestroy (pml);
            dData = 0;
            if (ns != STATUS_INFO_LENGTH_MISMATCH) break;
            }
        }
    if (pdData != NULL) *pdData = dData;
    if (pns      != NULL) *pns       = ns;
    return pml;
    }
```

列表 6-8. 使用 ZwQuerySystemInformation()获取模块列表

剩下的操作将用来获取给定模块的基地址，这将非常简单。**列表 6-9** 定义了两个函数：SpyModuleFind()是 SpyModuleList()的增强版，它可以根据指定的模块文件名来扫描 ZwQuerySystemInformation()返回的模块列表，SpyModuleBase()反复调用 SpyModuleFind()，从模块的 MODULE_INFO 结构中提取出模块的基地址。SpyModuleHeader()函数调用 SpyModuleBase()并将获取的模块基地址传递给 RtlImageNtHeader()。该函数是进入已加载模块导出节（export section）的第一步。

```
PMODULE_LIST SpyModuleFind (PBYTE        pbModule,
                            PDWORD       pdIndex,
                            PNTSTATUS pns)
    {
    DWORD            i;
    DWORD            dIndex = -1;
    NTSTATUS         ns        = STATUS_INVALID_PARAMETER;
    PMODULE_LIST pml       = NULL;
    if ((pml = SpyModuleList (NULL, &ns)) != NULL)
        {
        for (i = 0; i < pml->dModules; i++)
            {
            if (!_stricmp (pml->aModules [i].abPath +
```

```
                                    pml->aModules [i].wNameOffset,
                                    pbModule))
                        {
                        dIndex = i;
                        break;
                        }
                    }
                if (dIndex == -1)
                    {
                    pml = SpyMemoryDestroy (pml);
                    ns  = STATUS_NO_SUCH_FILE;
                    }
                }
        if (pdIndex != NULL) *pdIndex = dIndex;
        if (pns     != NULL) *pns     = ns;
        return pml;
        }
// -------------------------------------------------------------
PVOID SpyModuleBase (PBYTE      pbModule,
                     PNTSTATUS pns)
        {
        PMODULE_LIST pml;
        DWORD        dIndex;
        NTSTATUS     ns     = STATUS_INVALID_PARAMETER;
        PVOID        pBase = NULL;
        if ((pml = SpyModuleFind (pbModule, &dIndex, &ns)) != NULL)
            {
            pBase = pml->aModules [dIndex].pBase;
            SpyMemoryDestroy (pml);
            }
```

```
    if (pns != NULL) *pns = ns;
    return pBase;
    }
// ------------------------------------------------------------
PIMAGE_NT_HEADERS SpyModuleHeader (PBYTE         pbModule,
                                   PPVOID        ppBase,
                                   PNTSTATUS pns)
    {
    PVOID                   pBase = NULL;
    NTSTATUS                ns       = STATUS_INVALID_PARAMETER;
    PIMAGE_NT_HEADERS pinh    = NULL;
    if (((pBase = SpyModuleBase (pbModule, &ns)) != NULL) &&
        ((pinh    = RtlImageNtHeader (pBase))          == NULL))
        {
        ns = STATUS_INVALID_IMAGE_FORMAT;
        }
    if (ppBase != NULL) *ppBase = pBase;
    if (pns       != NULL) *pns       = ns;
    return pinh;
    }
```

列表 6-9. 查找指定模块的信息

6.2.3、解析导出函数、变量的符号

前一小节解释了如何搜索一个 PE 文件映像中导出函数和变量的符号化名称，以及如何确定已加载系统模块或驱动程序的基地址。现在，是时候将这些零碎的东西整理一下了。基本上，查找一个给定模块的导出符号需要如下三个步骤：

1. 找到模块的线性基地址
2. 搜索模块导出节中的符号
3. 将找到的符号的相对偏移量和模块基地址相加

第一步已经讨论过。**列表 6-10** 提供了剩余步骤地实现细节。SpyModuleExport()需要一个文件名，如 ntoskrnl.exe、hal.dll、ntfs.sys 等等，pbModule 参数返回一个指向模块的 IMAGE_EXPORT_DIRECTORY 结构的指针。可选参数 ppBase 和 pns 返回附加的信息：*ppBase 在成功的情况下返回模块的基地址，*pns 在出错的情况下返回错误状态的诊断信息。首先，SpyModuleExport()调用 SpyModuleHeader()来定位 IMAGE_NT_HEADERS；然后，它计算 PE DataDirectory 数组中第一个元素（该元素是一个 IMAGE_DATA_DIRECTORY 结构）所保存的有关导出节的信息。如果 IMAGE_DATA_DIRECTORY 结构的 VirtualAddress 成员不是 NULL，并且 Size 成员是一个合理的值，则可以断定该 PE Image 包含至少一个导出节。此时，SpyModuleExport()使用 PTR_ADD()宏（见列表 6-10）将 VirtualAddress 加上模块基地址，从而得到 IMAGE_EXPORT_DIRECTORY 的绝对线性地址。否则，SpyModuleExport()将返回 NULL，并将状态代码设为 STATUS_DATA_ERROR（0xC000003E）。

```
#define PTR_ADD(_base,_offset) \
        ((PVOID) ((PBYTE) (_base) + (DWORD) (_offset)))
// ------------------------------------------------------------
PIMAGE_EXPORT_DIRECTORY SpyModuleExport (PBYTE      pbModule,
                                         PPVOID     ppBase,
                                         PNTSTATUS pns)
    {
    PIMAGE_NT_HEADERS          pinh;
    PIMAGE_DATA_DIRECTORY      pidd;
    PVOID                      pBase = NULL;
    NTSTATUS                   ns    = STATUS_INVALID_PARAMETER;
    PIMAGE_EXPORT_DIRECTORY pied  = NULL;
    if ((pinh = SpyModuleHeader (pbModule, &pBase, &ns)) != NULL)
        {
        pidd = pinh->OptionalHeader.DataDirectory
                + IMAGE_DIRECTORY_ENTRY_EXPORT;
        if (pidd->VirtualAddress &&
            (pidd->Size >= IMAGE_EXPORT_DIRECTORY_))
```

```
                {
                pied = PTR_ADD (pBase, pidd->VirtualAddress);
                }
            else
                {
                ns = STATUS_DATA_ERROR;
                }
            }
        if (ppBase != NULL) *ppBase = pBase;
        if (pns    != NULL) *pns    = ns;
        return pied;
        }
// ------------------------------------------------------------
PVOID SpyModuleSymbol (PBYTE      pbModule,
                       PBYTE      pbName,
                       PPVOID     ppBase,
                       PNTSTATUS pns)
        {
        PIMAGE_EXPORT_DIRECTORY pied;
        PDWORD                  pdNames, pdFunctions;
        PWORD                   pwOrdinals;
        DWORD                   i, j;
        PVOID                   pBase    = NULL;
        NTSTATUS                ns       = STATUS_INVALID_PARAMETER;
        PVOID                   pAddress = NULL;
        if ((pied = SpyModuleExport (pbModule, &pBase, &ns)) != NULL)
            {
            pdNames     = PTR_ADD (pBase, pied->AddressOfNames);
            pdFunctions = PTR_ADD (pBase, pied->AddressOfFunctions);
```

```
        pwOrdinals    = PTR_ADD (pBase, pied->AddressOfNameOrdinals);
        for (i = 0; i < pied->NumberOfNames; i++)
            {
            j = pwOrdinals [i];
            if (!strcmp (PTR_ADD (pBase, pdNames [i]), pbName))
                {
                if (j < pied->NumberOfFunctions)
                    {
                    pAddress = PTR_ADD (pBase, pdFunctions [j]);
                    }
                break;
                }
            }
        if (pAddress == NULL)
            {
            ns = STATUS_PROCEDURE_NOT_FOUND;
            }
        }
    if (ppBase != NULL) *ppBase = pBase;
    if (pns       != NULL) *pns       = ns;
    return pAddress;
    }
```

列表 6-10. 在模块的导出节中查找符号

SpyModuleSymbol()函数将完成最后的工作。在这里你会发现访问**列表 6-1** 所示结构体的代码。在从 SpyModuleExport()返回一个 IMAGE_EXPORT_DIRECTORY 指针后，地址、名称和序列号数组的线性地址就可以确定下来了，再次使用 PTR_ADD()宏。很幸运，PE 文件格式规定指向其内部数据结构的指针总是一个相对于映像文件基地址的偏移量，因此 PTR_ADD()宏提供了一个通用的转换方式，可根据偏移量计算线性地址。在查找地址时，要特别注意序列号数组。如果在相同的数组中找到了符号，变量 i 中将包含该符号在数组中

的索引，此索引从 0 开始计数。这个索引值不能直接拿来访问地址数组，它必须经过序列号数组的转换。j = pwordinals[i];就是完成这一工作的。注意，这里的序列号是一个 16 位的值，而其他两个数组包含的都是 32 位的值。如果传给 SpyModuleSymbol()的符号名（由 pbName 引用）无法解析，则 SpyModuleSymbol()将返回一个 NULL，此时的返回代码将为 STATUS_PROCEDURE_NOT_FOUND（0xC000007A）。

尽管看上去 SpyModuleSymbol()提供了我们按名字调用内核函数所需的一切，我还是引入了该函数的另一个外包函数。**列表 6-11** 给出了最终结果：SpyModuleSymbolEx()函数使用一个由模块/符号名组成的单一字符串，其格式为”module!symbol”，当然符号的解析还是要有 SpyModuleSymbol()来完成。SpyModuleSymbolEx()中的大部分代码都用来将输入的字符串解析为一个模块名和一个符号。如果没有找到”!”分隔符，SpyModuleSymbolEx()将假定 ntoskrnl.exe 为目标模块，因为该模块是最常用的一个。

```
PVOID SpyModuleSymbolEx (PBYTE         pbSymbol,
                         PPVOID        ppBase,
                         PNTSTATUS pns)
    {
    DWORD       i;
    BYTE        abModule [MAXIMUM_FILENAME_LENGTH] = "ntoskrnl.exe";
    PBYTE       pbName      = pbSymbol;
    PVOID       pBase       = NULL;
    NTSTATUS ns             = STATUS_INVALID_PARAMETER;
    PVOID       pAddress = NULL;
    for (i = 0; pbSymbol [i] && (pbSymbol [i] != '!'); i++);
    if (pbSymbol [i++])
        {
        if (i <= MAXIMUM_FILENAME_LENGTH)
            {
            strcpyn (abModule, pbSymbol, i);
            pbName = pbSymbol + i;
            }
        else
```

```
            {
            pbName = NULL;
            }
        }
    if (pbName != NULL)
        {
        pAddress = SpyModuleSymbol (abModule, pbName, &pBase, &ns);
        }
    if (ppBase != NULL) *ppBase = pBase;
    if (pns      != NULL) *pns      = ns;
    return pAddress;
    }
```

列表 6-11. 一个强大的符号查找函数

6.2.4、通往用户模式的桥梁

现在，内核调用接口的演化已经缓慢的到达了终点----至少已经涉及内核模式（kernel-mode）。让我们总结一下到目前为止，我们已经获得了什么：

- 名为 SpyCallEx()的函数（见**列表 6-3**）将收到一个 SPY_CALL_INPUT 控制块，该控制块中包含目标地址和一些函数所需的参数。SpyCallEx()调用指定的地址，并且通过一个 SPY_CALL_OUTPUT 控制块将结果返回。
- 一种按名字查找导出的系统函数和变量的方法，该方法由 SpyModuleSymbolEx()函数实现（见**列表 6-11**）。

现在，最后一个问题是：“我们如何让用户模式下的应用程序访问这些资源？”回答当然是：“通过设备 I/O 控制（Device I/O Control）”。到现在为止，Spy device 提供了一组 IOCTL 函数，**表 6-1** 列出了这些函数。该表是第四章的**表 4-2** 的摘要，**表 4-2** 包含 w2k_spy.sys 提供的所有 IOCTL 函数。**列表 6-12** 给出了与 SpyDispatcher()函数相关的部分，第四章的**列表 4-7** 给出了 SpyDispatcher()函数的具体细节。

表 6-1 最后一行中，名为 SPY_IO_CALL 的函数将作为通向用户模式的桥梁。我相信一旦 Spy device 可以访问这些极具价值的信息，它将很容易使用户模式的应用程序获取这些数据。就像在第四章和第五章中一样，下面我们将对新引入的 IOCTL 函数作一个简短的介绍。

表 6-1. 与内核调用接口相关的 IOCTL 函数

函数名	ID	IOCTL 编码	描　　述
SPY_IO_MODULE_INFO	19	0x8000604C	返回有关已加载的系统模块的信息
SPY_IO_PE_HEADER	20	0x80006050	返回 IMAGE_NT_HEADERS 数据
SPY_IO_PE_EXPORT	21	0x80006054	返回 IMAGE_EXPORT_DIRECTORY 数据
SPY_IO_PE_SYMBOL	22	0x80006058	返回一个导出符号的地址
SPY_IO_CALL	23	0x8000E05C	调用一个位于模块（已加载）内部的函数

```
NTSTATUS SpyDispatcher (PDEVICE_CONTEXT pDeviceContext,
                        DWORD           dCode,
                        PVOID           pInput,
                        DWORD           dInput,
                        PVOID           pOutput,
                        DWORD           dOutput,
                        PDWORD          pdInfo)
    {
    SPY_MEMORY_BLOCK smb;
    SPY_PAGE_ENTRY   spe;
    SPY_CALL_INPUT   sci;
    PHYSICAL_ADDRESS pa;
    DWORD            dValue, dCount;
    BOOL             fReset, fPause, fFilter, fLine;
    PVOID            pAddress;
    PBYTE            pbName;
    HANDLE           hObject;
```

```
NTSTATUS            ns = STATUS_INVALID_PARAMETER;
MUTEX_WAIT (pDeviceContext->kmDispatch);
*pdInfo = 0;
switch (dCode)
    {
    // unrelated IOCTL functions ommitted (cf. Listing 4-7)
    case SPY_IO_MODULE_INFO:
        {
        if ((ns = SpyInputPointer (&pbName,
                                   pInput, dInput))
            == STATUS_SUCCESS)
            {
            ns = SpyOutputModuleInfo (pbName,
                                      pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_PE_HEADER:
        {
        if ((ns = SpyInputPointer (&pAddress,
                                   pInput, dInput))
            == STATUS_SUCCESS)
            {
            ns = SpyOutputPeHeader (pAddress,
                                    pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_PE_EXPORT:
        {
```

```
        if ((ns = SpyInputPointer (&pAddress,
                                        pInput, dInput))
            == STATUS_SUCCESS)
            {
            ns = SpyOutputPeExport (pAddress,
                                        pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_PE_SYMBOL:
        {
        if ((ns = SpyInputPointer (&pbName,
                                        pInput, dInput))
            == STATUS_SUCCESS)
            {
            ns = SpyOutputPeSymbol (pbName,
                                        pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_CALL:
        {
        if ((ns = SpyInputBinary (&sci, SPY_CALL_INPUT_,
                                        pInput, dInput))
            == STATUS_SUCCESS)
            {
            ns = SpyOutputCall (&sci,
                                        pOutput, dOutput, pdInfo);
            }
```

```
                break;
                }
            }
    MUTEX_RELEASE (pDeviceContext->kmDispatch);
    return ns;
    }
```

列表 6-12. Spy driver 的 Hook Command Dispatcher（摘录）

6.2.5、IOCTL 函数 SPY_IO_MODULE_INFO

IOCTL 函数 SPY_IO_MODULE_INFO 接收一个模块基地址，并返回一个 SPY_MODULE_INFO 结构（如果该地址指向了一个有效的 PE Image）。**列表 6-13** 给出了该结构的定义以及与其相关的 SpyOutputModuleInfo()帮助函数（**列表 6-12** 中的 SpyDispatcher()将调用该函数）。SpyOutputModuleInfo()基于 SpyModuleFind()函数（参见**列表 6-9**），SpyModuleFind()函数返回它从 ZwQuerySystemInformation()获取的 MODULE_INFO 数据。MODULE_INFO 数据将被转换为 SPY_MODULE_INFO 格式后发送给调用者。

```
typedef struct _SPY_MODULE_INFO
    {
    PVOID pBase;
    DWORD dSize;
    DWORD dFlags;
    DWORD dIndex;
    DWORD dLoadCount;
    DWORD dNameOffset;
    BYTE    abPath [MAXIMUM_FILENAME_LENGTH];
    }
    SPY_MODULE_INFO, *PSPY_MODULE_INFO, **PPSPY_MODULE_INFO;
#define SPY_MODULE_INFO_ sizeof (SPY_MODULE_INFO)
// -------------------------------------------------------------
NTSTATUS SpyOutputModuleInfo (PBYTE    pbModule,
```

```
                                    PVOID    pOutput,
                                    DWORD    dOutput,
                                    PDWORD pdInfo)
{
SPY_MODULE_INFO smi;
PMODULE_LIST       pml;
PMODULE_INFO       pmi;
DWORD                  dIndex;
NTSTATUS              ns = STATUS_INVALID_PARAMETER;
if ((pbModule != NULL) && SpyMemoryTestAddress (pbModule) &&
      ((pml = SpyModuleFind (pbModule, &dIndex, &ns)) != NULL))
      {
      pmi = pml->aModules + dIndex;
      smi.pBase            = pmi->pBase;
      smi.dSize            = pmi->dSize;
      smi.dFlags           = pmi->dFlags;
      smi.dIndex           = pmi->wIndex;
      smi.dLoadCount    = pmi->wLoadCount;
      smi.dNameOffset = pmi->wNameOffset;
      strcpyn (smi.abPath, pmi->abPath, MAXIMUM_FILENAME_LENGTH);
      ns = SpyOutputBinary (&smi, SPY_MODULE_INFO_,
                                    pOutput, dOutput, pdInfo);
      SpyMemoryDestroy (pml);
      }
return ns;
}
```

列表 6-13. SPY_IO_MODULE_INFO 的实现方式

6.2.6、IOCTL 函数 SPY_IO_PE_HEADER

IOCTL 函数 SPY_IO_PE_HEADER 只是一个简单的 IOCTL 外包函数，其核心部分是 ntoskrnl.exe 导出的 RtlImageNtHeader()函数，如列表 6-14 所示。和 SPY_IO_MODULE_INFO 类似，SPY_IO_PE_HEADER 也需要一个模块基地址。返回的数据是模块的 IMAGE_NT_HEADER 结构。

```
NTSTATUS SpyOutputPeHeader (PVOID    pBase,
                            PVOID    pOutput,
                            DWORD    dOutput,
                            PDWORD pdInfo)
    {
    PIMAGE_NT_HEADERS pinh;
    NTSTATUS              ns = STATUS_INVALID_PARAMETER;
    if ((pBase != NULL) && SpyMemoryTestAddress (pBase) &&
        ((pinh = RtlImageNtHeader (pBase)) != NULL))
        {
        ns = SpyOutputBinary (pinh, IMAGE_NT_HEADERS_,
                              pOutput, dOutput, pdInfo);
        }
    return ns;
    }
```

列表 6-14. SPY_IO_PE_HEADER 的实现方式

6.2.7、IOCTL 函数 SPY_IO_PE_EXPORT

这个函数比上一个 IOCTL 函数有趣多了。该函数返回与调用者提供的模块基地址相关的 IMAGE_EXPORT_DIRECTORY 结构。仔细观察**列表 6-15** 给出的该函数的实现方式，你会发现它和**列表 6-10** 中的 SpyModuleExport()极其相似。不过，SpyOutputPeExport()需要完成了更多的工作。这是因为 IMAGE_EXPORT_DIRECTORY 包含相对地址的缘故，这一点前面已经解释过。在数据被复制到独立的缓冲区之后，调用者还是无法使用这些偏移量，这

是因为与这些偏移量相关的基地址已经改变了。在 PE 表头中没有其他附加的地址信息，因此不可能计算出一个新的与之匹配的基地址。为了减少调用者的工作，SpyOutputPeExport()将所有指向导出节内部的偏移量转换为相对于导出节起始位置的偏移量，这是通过减去它们在 IMAGE_DATA_DIRECTORY 结构中的 VirtualAddress 而得到的。地址数组中的数据项必须采用不同的方法进行处理，因为它们引用的是 PE Image 中的其他节区。因此，SpyOutputPeExport()将它们加上 Image 的基地址，从而将它们转换为绝对线性地址。

```
NTSTATUS SpyOutputPeExport (PVOID    pBase,
                            PVOID    pOutput,
                            DWORD    dOutput,
                            PDWORD pdInfo)
    {
    PIMAGE_NT_HEADERS          pinh;
    PIMAGE_DATA_DIRECTORY      pidd;
    PIMAGE_EXPORT_DIRECTORY pied;
    PVOID                      pData;
    DWORD                      dData, dBias, i;
    PDWORD                     pdData;
    NTSTATUS                   ns = STATUS_INVALID_PARAMETER;
```

```
    if ((pBase != NULL) && SpyMemoryTestAddress (pBase) &&
        ((pinh = RtlImageNtHeader (pBase)) != NULL))
        {
        pidd = pinh->OptionalHeader.DataDirectory
                + IMAGE_DIRECTORY_ENTRY_EXPORT;
        if (pidd->VirtualAddress &&
            (pidd->Size >= IMAGE_EXPORT_DIRECTORY_))
            {
            pData = (PBYTE) pBase + pidd->VirtualAddress;
```

```
            dData = pidd->Size;
            if ((ns = SpyOutputBinary (pData, dData,
                                        pOutput, dOutput, pdInfo))
                == STATUS_SUCCESS)
                {
                pied    = pOutput;
                dBias = pidd->VirtualAddress;
                pied->Name                          -= dBias;
                pied->AddressOfFunctions        -= dBias;
                pied->AddressOfNames                -= dBias;
                pied->AddressOfNameOrdinals -= dBias;
                pdData = PTR_ADD (pied, pied->AddressOfFunctions);
                for (i = 0; i < pied->NumberOfFunctions; i++)
                    {
                    pdData [i] += (DWORD) pBase;
                    }
                pdData = PTR_ADD (pied, pied->AddressOfNames);
                for (i = 0; i < pied->NumberOfNames; i++)
                    {
                    pdData [i] -= dBias;
                    }
                }
            }
        else
            {
            ns = STATUS_DATA_ERROR;
            }
        }
    return ns;
```

```
    }
```

列表 6-15. SPY_IO_PE_EXPORT 的实现细节

6.2.8、IOCTL 函数 SPY_IO_PE_SYMBOL

该函数使得用户模式下的应用程序可以访问内核调用接口的符号查找引擎。列表 6-16 给出了该函数的实现方式，但看起来并不是很让人兴奋，因为它只是列表 6-11 中的 SpyModuleSymbolEx()的外包函数而已。调用者必须传入一个形如”module!symbol”的字符串或者一个”symbol”（如果该 symbol 属于 ntoskrnl.exe）。如果该函数调用成功，函数将返回与符号相关的线性地址，如果调用者提供的 symbol 无效或者发生了其他错误，则函数返回 NULL。

```
NTSTATUS SpyOutputPeSymbol (PBYTE    pbSymbol,
                            PVOID    pOutput,
                            DWORD    dOutput,
                            PDWORD pdInfo)
    {
    PVOID       pAddress;
    NTSTATUS ns = STATUS_INVALID_PARAMETER;
    if ((pbSymbol != NULL) && SpyMemoryTestAddress (pbSymbol)
          &&
          ((pAddress = SpyModuleSymbolEx (pbSymbol, NULL, &ns))
            != NULL))
          {
          ns = SpyOutputPointer (pAddress,
                                 pOutput, dOutput, pdInfo);
          }
    return ns;
    }
```

列表 6-16. SPY_IO_PE_SYMBOL 的实现细节

6.2.9、IOCTL 函数 SPY_IO_CALL

最后是我们等待良久的 SPY_IO_CALL 函数了。**列表 6-17** 提供了该函数的实现细节。如果传入的字符串地址正确，此函数将调用 SpyModuleSymbolEx()，如果传入的字符串可以被解析，则继续调用 SpyCallEx()。和 SPY_IO_PE_SYMBOL 类似，此函数期望一个形如”module!symbol”或一个简单的”symbol”作为符号名，符号字符串将作为 SPY_CALL_INPUT 结构的一部分被初始化。如果成功，SPY_IO_CALL 返回一个 SPY_CALL_OUTPUT 结构，该结构中将包含函数调用的结果（如果传入的符号引用的是一个 API 函数），或者目标变量的值（如果符号引用的是一个公共变量，如 NtBuildNumber 或 KeServiceDescriptorTable）。

如果 SPY_IO_CALL 调用失败，则将不会返回任何数据。调用者必须适当的处理这种情况。忽略这一错误意味着从内核调用接口返回的是无效数据。如果将这样的数据传递给其他内核函数，将会出现错误。如果你很走运，则错误将由 SpyCallEx()内的异常处理例程捕获。如果你很不走运，则整个进程都将挂起在 Spy device 的 IOCTL 调用中。而且，这里还存在着出现蓝屏的可能性。但是不要过于担心，在下一节中，将展示如何在用户模式的应用程序中恰当的使用内核调用接口。

```
NTSTATUS SpyOutputCall (PSPY_CALL_INPUT psci,
                        PVOID           pOutput,
                        DWORD           dOutput,
                        PDWORD          pdInfo)
    {
    SPY_CALL_OUTPUT sco;
    NTSTATUS        ns = STATUS_INVALID_PARAMETER;
    if (psci->pbSymbol != NULL)
        {
        psci->pEntryPoint =
            (SpyMemoryTestAddress (psci->pbSymbol)
             ? SpyModuleSymbolEx  (psci->pbSymbol, NULL, &ns)
             : NULL);
        }
```

```
if ((psci->pEntryPoint != NULL)                    &&
      SpyMemoryTestAddress (psci->pEntryPoint) &&
      ((ns = SpyCallEx (psci, &sco)) == STATUS_SUCCESS))
      {
      ns = SpyOutputBinary (&sco, SPY_CALL_OUTPUT_,
                                    pOutput, dOutput, pdInfo);
      }
return ns;
}
```

列表 6-17. SPY_IO_CALL 的实现细节

6.3、将调用接口封装为 DLL

尽管 w2k_spy.sys 已经导出了针对内核函数的 IOCTL 调用接口，但该接口在使用时多少有些不太方便。如果你想调用一个简单的函数，如 MmGetPhysicalAddress()或 MmIsAddressValid()。首先，你必须使用要调用的函数及其所需参数来构建一个 SPY_CALL_INPUT 结构，接下来，你必须发出一个 Win32 DeviceIoControl()调用。如果该函数返回 ERROR_SUCCESS，则你必须计算返回的 SPY_CALL_OUTPUT 结构，而且必须恰当的处理发生的错误。这看起来一点也不吸引人，不是吗？解决这一问题的方案就是将 IOCTL 机制隐藏到 DLL 中。这就是包含在本书光盘中的 w2k_call.dll 项目的目的所在。本节给出了摘自 w2k_call.c 和 w2k_call.h 中的代码片断，可以在光盘的\src\w2k_call\目录下找到这两个文件。

6.3.1、处理 IOCTL 函数调用

在开始其他工作之前，必须先以一种简便的方式将 DeviceIoControl()调用封装起来，因为这是所有内核函数调用必须经过的一个瓶颈。**列表 6-18** 给出了外包函数 w2kSpyControl()，该函数的核心部分还是调用 DeviceIoControl()。总的来说，此外包函数将执行如下的任务：

- 验证输入/输出参数
- 如果还未加载 Spy device，则加载 Spy device driver，并打开 Spy device。

- 调用 DeviceIoControl()
- 测试输出数据是否为预期大小
- 设置恰当的 Win32 错误代码（Win32 last-error code）

如果成功的话，应用程序通过 GetLastError()获取到的系统最后错误代码将被设置为 ERROR_SUCCESS（其值为 0）。否则，将按照如下策略来设置错误代码：

- 如果输入或输出参数无效，错误代码将被设为 ERROR_INVALID_PARAMETER（87），表示“参数不正确”（依照 SDK 头文件 winerror.h）。
- 如果 Spy device 无法初始化，错误代码将被设为 ERROR_GEN_FAILURE（31），这表示“与系统相关的一个设备无法工作”。
- 如果 spy device 返回的数据的大小与调用者提供的缓冲区大小不一致，错误代码将被设为 ERROR_DATATYPE_MISMATCH，这表示“提供的数据的类型有错误”。
- 对于其他的情况，w2kSpyControl()保留 DeviceIoControl()设置的错误代码。这通常是 spy device 返回的 NTSTATUS，不过该返回值已映射为适当的 Win32 状态代码。

```
BOOL WINAPI w2kSpyControl (DWORD dCode,
                           PVOID pInput,
                           DWORD dInput,
                           PVOID pOutput,
                           DWORD dOutput)
    {
    DWORD dInfo = 0;
    BOOL    fOk     = FALSE;
    SetLastError (ERROR_INVALID_PARAMETER);
    if (((pInput    != NULL) || (!dInput )) &&
        ((pOutput != NULL) || (!dOutput)))
        {
        if (w2kSpyStartup (FALSE, NULL))
            {
            if (DeviceIoControl (ghDevice, dCode,
                                 pInput,     dInput,
                                 pOutput,    dOutput,
```

```
                                        &dInfo,     NULL))
                {
                if (dInfo == dOutput)
                    {
                    SetLastError (ERROR_SUCCESS);
                    fOk = TRUE;
                    }
                else
                    {
                    SetLastError (ERROR_DATATYPE_MISMATCH);
                    }
                }
            }
        else
            {
            SetLastError (ERROR_GEN_FAILURE);
            }
        }
    return fOk;
    }
```

列表 6-18. 基本的 DeviceIoContorl 外包函数

对于**列表 6-18** 中调用的 w2kSpyStartup()（在调用 DeviceIoControl 之前），我们应更多的关注一下，因为 w2k_call.dll 依赖于内核模式的驱动程序所提供的服务，该驱动程序必须在第一个 IOCTL 调用之前被加载到内存中。而且该设备的句柄也必须被打开，以表示可通过 DeviceIoControl()访问该目标设备。为了使 DLL 具有尽可能大的灵活性，我选择了混合模式，在此模式下，调用者可以完全的控制 Spy device 的加载/卸载、打开/关闭或者采用默认的机制（将设备的管理任务交给 DLL 来完成）。这种自动化机制非常简单：直道第一次请求 IOCTL 调用，才加载并打开 Spy device。一旦 DLL 被卸载了，将自动关闭设备的句柄，但仍将内核模式的驱动程序保留在内存中。接下来，我们将制订一个预防性的策略。只要调

用者不提供任何有关如何处理驱动程序的具体信息，w2k_call.dll 就将假定其他客户还在使用驱动程序，此时它将不会卸载该驱动程序。产生问题的并不是仍然持有 Spy device 句柄的进程。Windows 2000 服务控制器（Service Control Manager，SCM）仅在设备的所有句柄都关闭的情况下才会卸载驱动程序。现在的问题是，SCM 不允许打开任何新的设备句柄。

w2k_call.dll 的客户程序可以通过 w2kSpyStartup()和 w2kSpyCleanup()来控制 Spy device 的状态，**列表 6-19** 给出了这两个函数。由于这两个函数可能在多线程环境下被调用，因此它们使用了一个临界区对象来进行同步。同一时刻，只能有一个线程可以加载/打开或者关闭/卸载 Spy device。例如，两个线程几乎同时调用了 w2kStartup()，那么只有一个线程可以打开设备句柄。另一个将被暂停。

```
BOOL WINAPI w2kSpyLock (VOID)
    {
    BOOL fOk = FALSE;
    if (gpcs != NULL)
        {
        EnterCriticalSection (gpcs);
        fOk = TRUE;
        }
    return fOk;
    }
// ------------------------------------------------------------
BOOL WINAPI w2kSpyUnlock (VOID)
    {
    BOOL fOk = FALSE;
    if (gpcs != NULL)
        {
        LeaveCriticalSection (gpcs);
        fOk = TRUE;
        }
    return fOk;
    }
```

```
// -------------------------------------------------------------
BOOL WINAPI w2kSpyStartup (BOOL          fUnload,
                           HINSTANCE hInstance)
    {
    HINSTANCE hInstance1;
    SC_HANDLE hControl;
    BOOL          fOk = FALSE;
    w2kSpyLock ();
    hInstance1 = (hInstance != NULL ? hInstance : ghInstance);
    if ((ghDevice == INVALID_HANDLE_VALUE) &&
        w2kFilePath (hInstance1, awSpyFile, awDriver, MAX_PATH)
        &&
        ((hControl = w2kServiceLoad (awSpyDevice, awSpyDisplay,
                                     awDriver, TRUE))
         != NULL))
        {
        ghDevice = CreateFile (awSpyPath,
                               GENERIC_READ       | GENERIC_WRITE,
                               FILE_SHARE_READ | FILE_SHARE_WRITE,
                               NULL, OPEN_EXISTING,
                               FILE_ATTRIBUTE_NORMAL, NULL);
        if ((ghDevice == INVALID_HANDLE_VALUE) && fUnload)
            {
            w2kServiceUnload (awSpyDevice, hControl);
            }
        else
            {
            w2kServiceDisconnect (hControl);
            }
```

```
        }
    fOk = (ghDevice != INVALID_HANDLE_VALUE);
    w2kSpyUnlock ();
    return fOk;
    }
// -------------------------------------------------------------
BOOL WINAPI w2kSpyCleanup (BOOL fUnload)
    {
    BOOL fOk = FALSE;
    w2kSpyLock ();
    if (ghDevice != INVALID_HANDLE_VALUE)
        {
        CloseHandle (ghDevice);
        ghDevice = INVALID_HANDLE_VALUE;
        }
    if (fUnload)
        {
        w2kServiceUnload (awSpyDevice, NULL);
        }
    w2kSpyUnlock ();
    return fOk;
    }
```

列表 6-19. Spy device 的管理函数

6.3.2、针对特定类型（type-specific）的调用接口函数

对 DeviceIoControl()调用以及 Spy device 的自动管理机制已被整理为一组相关的函数，w2kSpyControl()是这组函数的主入口点。接下来我们将提供一个函数来执行 Spy device 的 SPY_IO_CALL 调用。**列表 6-20** 给出了内核调用接口在用户模式下的基本实现方式，这由 w2kCallExecute()、w2kCall()和 w2kCallV()组成。它们所需的输入参数，在形式上等价于用

户模式下的 SpyCallEx()（参见**列表 6-3**）。事实上，从 w2kCallExecute()的实现方式上可以看出，它首先确认在传入的控制块中是否包含一个符号名或者一个入口地址，然后通过 w2kSpyControl()调用 Spy device 的 SPY_IO_CALL 函数。从**列表 6-12** 中，我们可看出 SPY_IO_CALL 是由 SpyOutputCall()（见**列表 6-17**）实现的，SpyOutputCall()依赖于 SpyModuleSymbolEx()和 SpyCallEx()。

```
BOOL WINAPI w2kCallExecute (PSPY_CALL_INPUT    psci,
                                    PSPY_CALL_OUTPUT psco)
    {
    BOOL fOk = FALSE;
    SetLastError (ERROR_INVALID_PARAMETER);
    if (psco != NULL)
        {
        psco->uliResult.QuadPart = 0;
        if ((psci != NULL)
            &&
            ((psci->pbSymbol       != NULL) ||
             (psci->pEntryPoint != NULL)))
            {
            fOk = w2kSpyControl (SPY_IO_CALL,
                                    psci, SPY_CALL_INPUT_,
                                    psco, SPY_CALL_OUTPUT_);
            }
        }
    return fOk;
    }
// -------------------------------------------------------------
BOOL WINAPI w2kCall (PULARGE_INTEGER puliResult,
                        PBYTE                pbSymbol,
                        PVOID                pEntryPoint,
                        BOOL                 fFastCall,
```

```
                        DWORD           dArgumentBytes,
                        PVOID           pArguments)
    {
    SPY_CALL_INPUT    sci;
    SPY_CALL_OUTPUT sco;
    BOOL              fOk = FALSE;
    sci.fFastCall          = fFastCall;
    sci.dArgumentBytes = dArgumentBytes;
    sci.pArguments         = pArguments;
    sci.pbSymbol           = pbSymbol;
    sci.pEntryPoint        = pEntryPoint;
    fOk = w2kCallExecute (&sci, &sco);
    if (puliResult != NULL) *puliResult = sco.uliResult;
    return fOk;
    }
// -------------------------------------------------------------
BOOL WINAPI w2kCallV (PULARGE_INTEGER puliResult,
                        PBYTE           pbSymbol,
                        BOOL            fFastCall,
                        DWORD           dArgumentBytes,
                        ...)
    {
    return w2kCall (puliResult, pbSymbol, NULL, fFastCall,
                        dArgumentBytes, &dArgumentBytes + 1);
    }
```

列表 6-20. 基本的内核调用接口函数

列表 6-20 中的 SpyCall()和 w2kCallV()是内核调用接口的核心函数，它们位于 w2k_call.dll 中。这两个函数为几个特定的函数提供基本服务。w2kCall()的主要目标是在调用 w2kCallExecute()之前将它的参数放入 SPY_CALL_INPUT 结构中，并将 w2kCallExecute()

返回的 ULARGE_INTEGER 作为自己的返回值。就像前面解释过的那样，返回值的所有位并不必须都是有效的，这依赖于被调用内核函数的返回值类型。w2kCallV()只是 w2kCall()的一个简单的外包函数，它的参数列表是可变的（这也是函数名末尾的 V 的含义）。由于 w2kCall()的参数列表是针对通用的内核 API 调用来设计的，因此它并不适合大多数常见的函数类型。比如，最常见的函数类型是__stdcall（或者 NTAPI），这些函数将返回一个 NTSTATUS。在这种情况下，fFastCall 参数将总为 FALSE，并且返回的 64 位 ULARGE_INTEGER 中仅有低 32 位包含有效的数据。因此，**列表 6-21** 提供了 w2kCallNT()函数来更好的完成这一工作。请注意 w2kCallNT()是如何控制 w2kCall()产生的错误的。如果 w2kCall()返回 FALSE，这意味着 w2kSpyControl()调用失败，函数的返回值是无效的。此时，就没有必要取出 uliResult 结构中的低 32 位值。因此，w2kCallNT()默认将返回 STATUS_IO_DEVICE_ERROR（0xC0000185）。在这一切都结束后，调用者必须准备好处理返回值不是 STATUS_SUCESS 的情况，对于非 STATUS_SUCESS 的返回值报告一个错误代码将是一个合理的决定。对于其他不返回 NTSTATUS 的内核函数，当它们失败时，必须小心的选择默认的返回值。

```
NTSTATUS WINAPI w2kCallNT (PBYTE pbSymbol,
                           DWORD dArgumentBytes,
                           ...)
    {
    ULARGE_INTEGER uliResult;
    return (w2kCall (&uliResult, pbSymbol, NULL, FALSE,
                     dArgumentBytes, &dArgumentBytes + 1)
            ? uliResult.LowPart
            : STATUS_IO_DEVICE_ERROR);
    }
```

列表 6-21. 针对 NTAPI/NTSTATUS 函数类型的简单接口

列表 6-22 给出了针对__stdcall 类型的 API 函数的 5 个附加接口函数，这些函数将返回基本的数据类型：BTYE、WORD、DWORD、DWORDLOG 和 PVOID。函数名末尾的数字表示返回值中的有效位的个数。w2kCallP()基本上等价于 w2kCall32()，不同之处在于

w2kCallP()将返回的 32 位值转型为一个指针。没有必要针对返回有符号数据类型或任意类型的指针而提供单独的函数。

编译器可以针对这些微小的不同自动进行类型转化（typecasting）。这里需要注意的是，**列表 6-22** 中所有函数的第一个参数都将作为其默认的返回值。这样做是必须的，因为调用接口无法知道当对内核函数的调用失败后，最好返回什么样的值，所以调用者应该负起这一责任。

```
BYTE WINAPI w2kCall08 (BYTE    bDefault,
                       PBYTE pbSymbol,
                       BOOL    fFastCall,
                       DWORD dArgumentBytes,
                       ...)
    {
    ULARGE_INTEGER uliResult;
    return (w2kCall (&uliResult, pbSymbol, NULL, fFastCall,
                     dArgumentBytes, &dArgumentBytes + 1)
            ? (BYTE) uliResult.LowPart
            : bDefault);
    }
// -------------------------------------------------------------
WORD WINAPI w2kCall16 (WORD    wDefault,
                       PBYTE pbSymbol,
                       BOOL    fFastCall,
                       DWORD dArgumentBytes,
                       ...)
    {
    ULARGE_INTEGER uliResult;
    return (w2kCall (&uliResult, pbSymbol, NULL, fFastCall,
                     dArgumentBytes, &dArgumentBytes + 1)
            ? (WORD) uliResult.LowPart
            : wDefault);
```

```
    }
// ------------------------------------------------------------
DWORD WINAPI w2kCall32 (DWORD dDefault,
                        PBYTE pbSymbol,
                        BOOL    fFastCall,
                        DWORD dArgumentBytes,
                        ...)
    {
    ULARGE_INTEGER uliResult;
    return (w2kCall (&uliResult, pbSymbol, NULL, fFastCall,
                        dArgumentBytes, &dArgumentBytes + 1)
            ? uliResult.LowPart
            : dDefault);
    }
// ------------------------------------------------------------
QWORD WINAPI w2kCall64 (QWORD qDefault,
                        PBYTE pbSymbol,
                        BOOL    fFastCall,
                        DWORD dArgumentBytes,
                        ...)
    {
    ULARGE_INTEGER uliResult;
    return (w2kCall (&uliResult, pbSymbol, NULL, fFastCall,
                        dArgumentBytes, &dArgumentBytes + 1)
            ? uliResult.QuadPart
            : qDefault);
    }
// ------------------------------------------------------------
PVOID WINAPI w2kCallP (PVOID pDefault,
```

```
                        PBYTE pbSymbol,
                        BOOL    fFastCall,
                        DWORD dArgumentBytes,
                        ...)
    {
    ULARGE_INTEGER uliResult;
    return (w2kCall (&uliResult, pbSymbol, NULL, fFastCall,
                        dArgumentBytes, &dArgumentBytes + 1)
            ? (PVOID) uliResult.LowPart
            : pDefault);
    }
```

列表 6-22. 针对常见的函数类型的接口函数

6.3.3、用于数据复制的接口函数

我在前面提到过 Spy device 的内核调用接口还可以处理内核模块导出的公开变量（public variable）。在**列表 6-2** 中给出了 SpyCall()函数，其中针对参数堆栈大小的负值（通过 SPY_CALL_INPUT 结构的 dArgumentBytes 成员传入）的补码将被解释为导出变量的大小。此时，SpyCall()将不会调用指定的入口点（entry point），而是从该入口点处复制适当的字节到结果缓冲区中。如果 dArgumentBytes 被设为-1，则意味着将入口点自身的地址复制到缓冲区中（-1 的补码为 0）。

列表 6-23 给出了由 w2k_call.dll 导出的数据复制函数。这组函数与**列表 6-22** 中的函数非常相似。不过，**列表 6-23** 中的函数只需要很少的输入参数。复制导出变量的值，只需要提供变量的名字即可，不需要多余的输入参数以及调用约定。

```
BOOL WINAPI w2kCopy (PULARGE_INTEGER puliResult,
                        PBYTE               pbSymbol,
                        PVOID               pEntryPoint,
                        DWORD               dBytes)
    {
    return w2kCall (puliResult, pbSymbol, pEntryPoint, FALSE,
```

```
                                0xFFFFFFFF - dBytes, NULL);
        }
// -------------------------------------------------------------
BYTE WINAPI w2kCopy08 (BYTE    bDefault,
                                PBYTE pbSymbol)
        {
        ULARGE_INTEGER uliResult;
        return (w2kCopy (&uliResult, pbSymbol, NULL, 1)
                ? (BYTE) uliResult.LowPart
                : bDefault);
        }
// -------------------------------------------------------------
WORD WINAPI w2kCopy16 (WORD    wDefault,
                                PBYTE pbSymbol)
        {
        ULARGE_INTEGER uliResult;
        return (w2kCopy (&uliResult, pbSymbol, NULL, 2)
                ? (WORD) uliResult.LowPart
                : wDefault);
        }
// -------------------------------------------------------------
DWORD WINAPI w2kCopy32 (DWORD dDefault,
                                 PBYTE pbSymbol)
        {
        ULARGE_INTEGER uliResult;
        return (w2kCopy (&uliResult, pbSymbol, NULL, 4)
                ? uliResult.LowPart
                : dDefault);
        }
// -------------------------------------------------------------
```

```
QWORD WINAPI w2kCopy64 (QWORD qDefault,
                        PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;
    return (w2kCopy (&uliResult, pbSymbol, NULL, 8)
            ? uliResult.QuadPart
            : qDefault);
    }
// -------------------------------------------------------------
PVOID WINAPI w2kCopyP (PVOID pDefault,
                       PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;
    return (w2kCopy (&uliResult, pbSymbol, NULL, 4)
            ? (PVOID) uliResult.LowPart
            : pDefault);
    }
// -------------------------------------------------------------
PVOID WINAPI w2kCopyEP (PVOID pDefault,
                        PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;
    return (w2kCopy (&uliResult, pbSymbol, NULL, 0)
            ? (PVOID) uliResult.LowPart
            : pDefault);
    }
```

列表 6-23. 针对基本数据类型的数据复制接口函数

在**列表 6-23** 中，w2kCopy()将完成大多数的工作，这很像用来处理函数调用的 w2kCall()。再次强调，w2k_call.dll 为基本的数据类型：BTYE、WORD、DWORD、DWORDLOG 和

PVOID，提供了单独的函数。其函数名末尾的数字表示返回值中有效位的个数。w2kCpyP()返回一个指针，w2kCopyEP()用来处理查询一个入口地址。调用 w2kCopyEP()等价于调用 Spy device 的 SPY_IO_PE_SYMBOL 函数。是的，这个函数是有些多余，不过，有两条可以回家的路总比一条没有的好，不是吗？

6.3.4、实现内核 API 的 Thunks

其实，可替代简单内核 API 函数的基本框架已经存在。我称其为“Thunks”，这是 Windows 行话中常见的一个术语，它代表一小段代码，这段代码是一个前端函数，为在系统的不同空间中实现的函数提供服务。另一个常见的术语是“代理”，不过它与微软组件对象模型（COM）紧密相关，因此，如果在这里使用它可能会引起混淆。让我们先从两个非常简单的 Windows 2000 内存管理函数开始，这两个函数是我在开发 w2k_call.dll 模块时的主要实验对象，它们是：MmGetPhysicalAddress()和 MmIsAddressValid()。**列表 6-24** 展示了在 w2kCall64()和 w2kCall08()的帮助下如何实现它们的 Thunks。为了避免和原始的目标函数冲突，我在所有的 Thunks 名称前增加了一个下划线作为前缀。

```
PHYSICAL_ADDRESS WINAPI
_MmGetPhysicalAddress (PVOID BaseAddress)
    {
    PHYSICAL_ADDRESS pa;
    pa.QuadPart = w2kCall64 (0, "MmGetPhysicalAddress", FALSE,
                                    4, BaseAddress);
    return pa;
    }
// -------------------------------------------------------------
BOOLEAN WINAPI
_MmIsAddressValid (PVOID VirtualAddress)
    {
    return w2kCall08 (FALSE, "MmIsAddressValid", FALSE,
                            4, VirtualAddress);
    }
```

列表 6-24. MmGetPhysicalAddress()和 MmIsAddressValid()的 Thunks 示列

MmGetPhysicalAddress()接受一个 32 位的线性地址，返回一个 64 位的 PHYSICAL_ADDRESS 结构，该结构只是 LARGE_INTEGER 结构的一个别名。因此，该函数的 Thunks 需要调用 w2kCall64()，这意味着将会向参数堆栈中传入四个字节的参数，而 BaseAddress 则作为 Thunks 的参数出现。在发生严重的 IOCTL 错误时，默认的返回值为 0，这也是原始函数返回的出错值。由于 MmGetPhysicalAddress()使用__stdcall 约定，因此，fFastCall 必须设为 FALSE。MmIsAddressValid()的 Thunks 的实现方式与之类似，不同之处仅在于 SpyCallEx()返回的第八个最低有效位，该位对应一个 BOOLEAN 数据类型。默认的返回值为 FALSE，这是一种预防性的选择。MmIsAddressValid()的典型应用是在访问内存前调用它，以避免潜在的页错误（page fault）。所以，不能假定函数的实际返回值为 TRUE，因为一个 IOCTL 错误将增大出现蓝屏的风险。

现在，让我们看一下在这一框架下导出变量是如何被访问的。**列表 6-25** 给出了两个 Thunks：_NtBulidNumber()和_KeServiceDescriptorTable()。NtBulidNumber 是 ntoskrnl.exe 导出的一个 16 位的 WORD 类型，因此，对应的 w2k_call.dll 接口函数为 w2kCopy16()。在发生错误是，该 Thunks 返回 0（如果你知道更合适的值，请告诉我 ☺）。_KeServiceDescriptorTable()稍微有些不同，因为 ntoskrnl.exe 导出的 KeServiceDescriptorTable 所指向的结构体大于 64 位。此时，最好的选择就是返回 KeServiceDescriptorTable 自身的地址。因此，该 Thunks 将使用 w2kCopyEP()，**列表 6-23** 给出了该函数。

```
WORD WINAPI
_NtBuildNumber (VOID)
    {
    return w2kCopy16 (0, "NtBuildNumber");
    }
// -------------------------------------------------------------
PSERVICE_DESCRIPTOR_TABLE WINAPI
_KeServiceDescriptorTable (VOID)
    {
    return w2kCopyEP (NULL, "KeServiceDescriptorTable");
    }
```

列表 6-25. NtBulidNumber 和 KeServiceDescriptorTable 的 Thunks

你可以想象当我发现这些 Thunks 居然真的可用时我是多么兴奋！当时我想：我要尝试调用一些最低层的函数---这些函数都与低层硬件绑定在一起，用于进行读写 I/O 端口等操作。很幸运，我先前设计的 SpyModuleSymbolEx()函数（参见**列表 6-11**）允许解析任何系统模块的导出符号，这包括内核模式的驱动程序。我的下一个任务就是调用 Windows 2000 硬件抽象层（Hardware Abstraction Layer，HAL）的导出函数。在检查完 hal.dll 导出节（export section）中的所有符号后，我决定尝试两个较简单的函数以保证可以和底层硬件直接对话：HalMakeBeep()和 HalQueryRealTimeClock()。通过可编程的定时器以及并行 I/O（Parallel I/O，PIO，其 I/O 地址为 0x0042、0x0043 和 0x0061）可以控制 PC 喇叭的发声，因此 HalMakeBeep()是测试与硬件相关的函数的 Thunk 的理想候选人。

列表 6-26 给出了_HalMakeBeep() Thunk 的实现代码，得益于 w2kCall08()辅助函数，实现代码非常简单。HalMakeBeep()可以根据所要求的强度使 PC 喇叭发出蜂鸣声。如果强度参数为 0，喇叭将停止发声。如果传入的强度值有效，则函数返回 TRUE。确切的说，0 或者大于 18 的强度值都是有效的。这里要注意的是，在调用 w2kCall08()时指定的字符串包含目标模块的名字，这里的目标模块为 hal.dll。**列表 6-24** 和**列表 6-25** 中均未指定模块，因为它们所引用的函数均由 ntoskrnl.exe 导出，而 ntoskrnl.exe 是默认的模块。

尽管 HalMakeBeep()是一个非常简单的函数，我还是非常非常高兴得看到_HalMakeBeep()终于可以工作了。PC 喇叭可以按我的要求发出声音了！而这时在 Windows 2000 而不是 DOS 下实现的，这证明了 Win32 应用程序可以调用 HAL 函数（这些函数直接访问硬件）。我将我在 DOS 下编写的 Beep Sequencer 移植到了 Windows 2000 中，**列表 6-27** 给出了其代码。w2kBeep()可发出一个指定强度和持续时间的单音节。w2kBeepEx()使用一组强度值/持续时间并顺序的演奏它们，直到遇到 0 强度。这两个函数均有 w2k_call.dll 导出。可能你会使用它们为 Win32 程序增加 DOS 风格的背景音乐。

```
BOOLEAN WINAPI
_HalMakeBeep (DWORD Pitch)
    {
    return w2kCall08 (FALSE, "hal.dll!HalMakeBeep", FALSE,
                                4, Pitch);
    }
```

列表 6-26. HalMakeBeep()的 Thunks

```
BOOL WINAPI
w2kBeep (DWORD dDuration,
         DWORD dPitch)
    {
    BOOL fOk = TRUE;
    if (!_HalMakeBeep (dPitch)) fOk = FALSE;
    Sleep (dDuration);
    if (!_HalMakeBeep (0      )) fOk = FALSE;
    return fOk;
    }
// -------------------------------------------------------------
BOOL WINAPI
w2kBeepEx (DWORD dData,
           ...)
    {
    PDWORD pdData;
    BOOL   fOk = TRUE;
    for (pdData = &dData; pdData [0]; pdData += 2)
        {
        if (!w2kBeep (pdData [0], pdData [1])) fOk = FALSE;
        }
    return fOk;
    }
```

列表 6-27. 一个简单的 Beep 音序程序

接下来，我将尝试更有用的函数，如 HalQueryRealTimeClock()。我记得在一个 DOS 程序中访问主板上的真实时钟（real-time clock）曾经是很困难的。这需要读/写硬件上的一对 I/O 端口。**列表 6-28** 给出了 HalQueryRealTimeClock()及其兄弟 HalSetRealTimeClock()的

Thunks，同时还给出了这两个函数所处理的 TIME_FIELDS 结构。在 ntddk.h 中定义了 TIME_FIELDS 结构。

```
typedef struct _TIME_FIELDS {
        CSHORT Year;
        CSHORT Month;
        CSHORT Day;
        CSHORT Hour;
        CSHORT Minute;
        CSHORT Second;
        CSHORT Milliseconds;
        CSHORT Weekday;
} TIME_FIELDS, *PTIME_FIELDS;
// -------------------------------------------------------------
VOID WINAPI
_HalQueryRealTimeClock (PTIME_FIELDS TimeFields)
        {
        w2kCallV (NULL, "hal.dll!HalQueryRealTimeClock", FALSE,
                        4, TimeFields);
        return;
        }
// -------------------------------------------------------------
VOID WINAPI
_HalSetRealTimeClock (PTIME_FIELDS TimeFields)
        {
        w2kCallV (NULL, "hal.dll!HalSetRealTimeClock", FALSE,
                        4, TimeFields);
        return;
        }
```

列表 6-28. HalQueryRealTimeClock()和 HalSetRealTimeClock()的 Thunks

列表 6-29 提供了一个使用_HalQueryRealTimeClock()的典型程序，该程序在控制台窗口中显示当前日期和时间。

```
VOID WINAPI DisplayTime(void)
{
    TIME_FIELDS tf;
    _HalQueryRealTimeClock(&tf);
    printf(L"\r\nData/Time: %02hd-%02hd-%04hd%02hd:%02hd:%02hd\r\n",
            tf.Month,   tf.Day,        tf.Year,
            tf.Hour,    tf.Minute,   tf.Second);
    return;
}
```

列表 6-29. 显示当前日期和时间

尽管内核调用接口可以工作，但仍然有些遗憾。很多年来，我们都被告知 Windows NT/2000 是一个安全的操作系统，在那里应用程序是不能为所欲为的。大多数 Win32 开发人员都已远离硬件了。经验更丰富些的 NT 开发人员至少知道如何通过 ntdll.dll 调用 Native API 函数。现在，使用本书提供的 DLL，所有的 Win32 开发人员都可以调用任意的内核函数，就像调用 Win32 API 函数那样。这是 Windows 2000 内核的一个安全漏洞吗？不，这并不是。只有让应用程序什么都不能访问的操作系统才是 100%安全的，不过这样的操作系统也没有什么实际价值。一旦我们可通过某种途径影响系统，那么系统将变得易于受到攻击。一但操作系统供应商允许第三方开发人员向系统中添加组件，就有可能获取通向内核的桥梁，就像 w2k_spy.sys/w2k_call.dll。不存在 100%安全的操作系统，除非该系统不与它周围的环境进行交互。

6.3.5、数据访问的支持函数

我向 w2k_call.dll 中加入了几打内核 API 函数的 Thunk。例如，由 Windows 2000 运行时库导出的所有字符串管理函数都可通过 w2k_call.dll 来调用。不过，你在实验这些预定义的 Thunks 或者你自己增加的 thunks，你会发现从用户模式调用内核 API 函数与调用普通的 Win32 函数还是有些不同。这里介绍的内核调用接口的简易性掩盖了调用程序仍然运行于用户模式、仅有有限的特权这一事实。例如，程序调用的内核函数可能会一个返回指向

UNICODE_STRING 结构的指针，而这一指针很有可能指向的是内核内存空间。任何读取该字符串的尝试都将引发一个异常，并导致程序终止，这是由于指令试图读取的地址是被禁止访问的地址。为了解决这一问题，我给 w2k_call.dll 增加了支持函数，用来更容易的访问内核 API 调用中大多数常见的数据类型。

列表 6-30 给出的 w2kSpyRead()函数是一个通用函数，它可以将任意的内存数据块复制到调用者提供的缓冲区中。该函数依赖于 w2k_spy.sys 提供的 IOCTL 函数-----SPY_IO_MEMORY_BLOCK（在第四章里简要的介绍过该函数）。可以使用 w2kSpyRead()读取位于内核空间中的结构体的任意部分。这里要特别注意的是，如果提供的内存块的地址范围无效，则对 w2kSpyRead()的调用将失败。这里的无效指的是没有与指定地址相关的物理内存或页面文件。w2kSpyClone()是 w2kSpyRead()的增强版本，w2kSpyClone()可以自动分配大小适当的缓冲区，然后将内核数据复制到该缓冲区中。

```
BOOL WINAPI w2kSpyRead (PVOID pBuffer,
                        PVOID pAddress,
                        DWORD dBytes)
    {
    SPY_MEMORY_BLOCK smb;
    BOOL             fOk = FALSE;
    if ((pBuffer != NULL) && (pAddress != NULL) && dBytes)
        {
        ZeroMemory (pBuffer, dBytes);
        smb.pAddress = pAddress;
        smb.dBytes   = dBytes;
        fOk = w2kSpyControl (SPY_IO_MEMORY_BLOCK,
                             &smb,    SPY_MEMORY_BLOCK_,
                             pBuffer, dBytes);
        }
    return fOk;
    }
// -------------------------------------------------------------
PVOID WINAPI w2kSpyClone (PVOID pAddress,
```

```
                                DWORD dBytes)
    {
    PVOID pBuffer = NULL;
    if ((pAddress != NULL) && dBytes &&
        ((pBuffer = w2kMemoryCreate (dBytes)) != NULL) &&
        (!w2kSpyRead (pBuffer, pAddress, dBytes)))
        {
        pBuffer = w2kMemoryDestroy (pBuffer);
        }
    return pBuffer;
    }
```

列表 6-30. 通用数据访问函数

读取字符串需要稍微复杂些。内核组件使用的常见字符串类型是 UNICODE_STRING，该结构由一个字符串缓冲区指针、有关缓冲区大小的信息和当前字符串占用的字节数构成。读取一个 UNICODE_STRING 结构通常需要两步来完成。第一，必须复制 UNICODE_STRING 结构以确定字符串缓冲区的大小和基地址。第二，读取实际的字符串数据。为了简化这一常见任务，w2k_call.dll 提供了一组函数（见列表 6-31）来完成这一工作。

w2kStringAnsi()和 w2kStringUnicode()分别用于分配并初始化空的 ANSI_STRING 和 UNICODE_STRING 结构，在初始化后的结构中将包含指定大小的字符串缓冲区。为了简单起见，字符串的头部和缓冲区使用一个内存块。这些结构体可用于字符串复制，w2kStringClone()给出的一个范例。w2kStringClone()可在用户内存空间中创建指定 UNICODE_STRING 结构的一个精确副本。副本的 MaximumLength 通常与原始结构相同，除非原始字符串结构中的参数不一致。例如，如果 MaximumLength 小于或等于 Length 成员，则 MaximumLength 将被视为无效，实际的最大长度将为 Length+2。不过，副本的 MaximumLength 永远不会小于原始的 MaximumLength。

```
PANSI_STRING WINAPI w2kStringAnsi (DWORD dSize)
    {
    PANSI_STRING pasData = NULL;
    if ((pasData = w2kMemoryCreate (ANSI_STRING_ + dSize))
```

```
        != NULL)
        {
        pasData->Length        = 0;
        pasData->MaximumLength = (WORD) dSize;
        pasData->Buffer        = PTR_ADD (pasData, ANSI_STRING_);
        if (dSize) pasData->Buffer [0] = 0;
        }
    return pasData;
    }
// -------------------------------------------------------------
PUNICODE_STRING WINAPI w2kStringUnicode (DWORD dSize)
    {
    DWORD           dSize1  = dSize * WORD_;
    PUNICODE_STRING pusData = NULL;
    if ((pusData = w2kMemoryCreate (UNICODE_STRING_ + dSize1))
        != NULL)
        {
        pusData->Length        = 0;
        pusData->MaximumLength = (WORD) dSize1;
        pusData->Buffer        = PTR_ADD (pusData, UNICODE_STRING_);
        if (dSize) pusData->Buffer [0] = 0;
        }
    return pusData;
    }
// -------------------------------------------------------------
PUNICODE_STRING WINAPI w2kStringClone (PUNICODE_STRING pusSource)
    {
    DWORD           dSize;
    UNICODE_STRING  usCopy;
    PUNICODE_STRING pusData = NULL;
```

```
if (w2kSpyRead (&usCopy, pusSource, UNICODE_STRING_))
    {
    dSize = max (usCopy.Length + WORD_,
                usCopy.MaximumLength) / WORD_;
    if (((pusData = w2kStringUnicode (dSize)) != NULL) &&
        usCopy.Length && (usCopy.Buffer != NULL))
        {
        if (w2kSpyRead (pusData->Buffer, usCopy.Buffer,
                                        usCopy.Length))
            {
            pusData->Length = usCopy.Length;
            pusData->Buffer    [usCopy.Length / WORD_] = 0;
            }
        else
            {
            pusData = w2kMemoryDestroy (pusData);
            }
        }
    }
return pusData;
}
```

列表 6-31. 字符串管理函数

将内核字符串复制到应用程序内存空间的另一方法是使用内核运行时函数。例如，你可以使用 w2k_call.dll 提供的_RtlInitUnicodeString()和_RtlCopyUnicodeString() Thunks 来完成这一任务。不过，调用 w2kStringClone()将更容易些，因为该函数可以自动分配复制字符串所需的内存。

6.4.6、访问未导出的符号

为什么我们要允许应用程序执行本该内核驱动程序完成的操作？我们能将应用程序的能力增强到内核驱动程序也无法达到的程度吗？我们可以调用既没有文档化也没有导出的内部函数吗？这听起来有些危险，不过，就像我接下来要展示的那样，如果小心的进行控制，它将不会像看起来那样“可怕”。

6.4.7、查找内部符号

前面所讲的内核调用接口将查找导出符号的地址的工作委托给了 Spy device，该 Spy device 可全面访问位于高一半的线形地址空间（upper half of the linear address space）中的内核模块的 PE 映像。不过，如果要调用的函数或者要访问的全局变量并没有导出，那么 Spy device 将没有机会找到它们的地址。在编写本章以及检查内核调试器输出的某些反汇编代码时，我总是想：“他们不导出这个漂亮的函数真是可惜！”这是因为内核调试器向我展示了确切的函数名，这让我非常恼火，但我的应用程序将完全忽略它们。当然，我可以使用我的内核调用接口跳到这些函数的无格式二进制入口点（the plain binary entry point of the function），但这不是一种很好的编程风格。以后发布的 Service Pack 很可能将这些入口点移动到其他位置。

我相信如果调试器可以做到这一点，我的程序也应该可以做到。在第一章中给出的一个示例 DLL 将我引上了正确的道路。只要操作系统的的符号文件正确安装了，w2k_img.dll 就可以查找由 Windows 2000 内核模块定义的任意符号的地址。因此，我进一步扩展了 w2k_call.dll，增加了一个新的 API 函数，该函数可先查找一个内部符号对应的线性地址，然后调用 w2kCall()执行它。当然，针对全局变量也提供了类似的函数。

列表 6-32 给出了所有扩展后的调用接口函数。并且，为了方便，还为每种主要函数类型提供了单独的函数，这些单独的函数对应**列表 6-20** 到 **6-22** 中的函数。w2kXCall()将完成主要的工作，它调用 w2k_img.dll 中的 API 函数 imgTableResolve()以获取指定符号的线性地址，如果成功，该地址将被列入随后的 w2kCall()调用列表中。因为 w2kCall()被设计为调用一个地址而不是一个符号，NULL 指针将作为 w2kCall()的 pbSymbol 参数传入。pEntryPoint 参数将被设置为符号的地址 pie->pAddress，该地址从对应的符号文件中获取。就像在第一章解释的那样，w2k_img.dll 可以确定大多数内部函数所需的调用约定，因此，通过测试

IMG_CONVENTION_FASTCALL 结构中 pie->dconvention 的值可自动设置 fFastcall 参数。参数的字节数和指向参数的指针将向前传递，就像接收自调用者一样。可以从符号信息中获取参数的个数，不过这仅对__stdcall 和__fastcall 函数有效。__cdecl symbols don't encode the argument stack size in their decoration.

```
BOOL WINAPI w2kXCall (PULARGE_INTEGER puliResult,
                      PBYTE           pbSymbol,
                      DWORD           dArgumentBytes,
                      PVOID           pArguments)
    {
    PIMG_TABLE pit;
    PIMG_ENTRY pie;
    BOOL       fOk = FALSE;
    if (((pit = w2kSymbolsGlobal (NULL))          != NULL) &&
        ((pie = imgTableResolve (pit, pbSymbol)) != NULL) &&
        (pie->pAddress != NULL))
        {
        fOk = w2kCall (puliResult, NULL, pie->pAddress,
                       pie->dConvention == IMG_CONVENTION_FASTCALL,
                       dArgumentBytes, pArguments);
        }
    else
        {
        if (puliResult != NULL) puliResult->QuadPart = 0;
        }
    return fOk;
    }
// -------------------------------------------------------------
BOOL WINAPI w2kXCallV (PULARGE_INTEGER puliResult,
                       PBYTE           pbSymbol,
```

```
                                DWORD                   dArgumentBytes,
                                ...)
    {
    return w2kXCall (puliResult, pbSymbol,
                            dArgumentBytes, &dArgumentBytes + 1);
    }
// ---------------------------------------------------------------
NTSTATUS WINAPI w2kXCallNT (PBYTE pbSymbol,
                                    DWORD dArgumentBytes,
                                    ...)
    {
    ULARGE_INTEGER uliResult;
    return (w2kXCall (&uliResult, pbSymbol,
                                dArgumentBytes, &dArgumentBytes + 1)
            ? uliResult.LowPart
            : STATUS_IO_DEVICE_ERROR);
    }
// ---------------------------------------------------------------
BYTE WINAPI w2kXCall08 (BYTE    bDefault,
                                PBYTE pbSymbol,
                                DWORD dArgumentBytes,
                                ...)
    {
    ULARGE_INTEGER uliResult;
    return (w2kXCall (&uliResult, pbSymbol,
                                dArgumentBytes, &dArgumentBytes + 1)
            ? (BYTE) uliResult.LowPart
            : bDefault);
    }
```

```
// ------------------------------------------------------------
WORD WINAPI w2kXCall16 (WORD    wDefault,
                        PBYTE pbSymbol,
                        DWORD dArgumentBytes,
                        ...)
    {
    ULARGE_INTEGER uliResult;
    return (w2kXCall (&uliResult, pbSymbol,
                        dArgumentBytes, &dArgumentBytes + 1)
            ? (WORD) uliResult.LowPart
            : wDefault);
    }
// ------------------------------------------------------------
DWORD WINAPI w2kXCall32 (DWORD dDefault,
                         PBYTE pbSymbol,
                         DWORD dArgumentBytes,
                         ...)
    {
    ULARGE_INTEGER uliResult;
    return (w2kXCall (&uliResult, pbSymbol,
                        dArgumentBytes, &dArgumentBytes + 1)
```

```
            ? uliResult.LowPart
            : dDefault);
    }
// ------------------------------------------------------------
QWORD WINAPI w2kXCall64 (QWORD qDefault,
                         PBYTE pbSymbol,
```

```
                                    DWORD dArgumentBytes,
                                    ...)
    {
    ULARGE_INTEGER uliResult;
    return (w2kXCall (&uliResult, pbSymbol,
                                dArgumentBytes, &dArgumentBytes + 1)
            ? uliResult.QuadPart
            : qDefault);
    }
// -------------------------------------------------------------
PVOID WINAPI w2kXCallP (PVOID pDefault,
                                    PBYTE pbSymbol,
                                    DWORD dArgumentBytes,
                                    ...)
    {
    ULARGE_INTEGER uliResult;
    return (w2kXCall (&uliResult, pbSymbol,
                                dArgumentBytes, &dArgumentBytes + 1)
            ? (PVOID) uliResult.LowPart
            : pDefault);
    }
```

列表 6-32. 扩展的调用接口

注意在**列表 6-32** 中的 w2kXCall()在开始工作之前会首先调用 w2kSymbolsGlobal()。**列表 6-33** 给出了 w2kSymbolsGlobal()函数，以及其他一些帮助函数，w2kSymbolsGlobal()的目的是在第一次调用 w2kXCall()之前加载 ntoskrnl.exe 的符号文件。符号表存储在名为 gpit 的全局变量中，该变量的类型为 PIMG_TABLE，在随后调用的函数中可以使用该表。在辅助函数的帮助下，w2kSymbolsLoad()可通过*pdstatus 参数返回**列表 6-2** 中的某一个状态代码。

为了避免由于不匹配的符号信息而跳到无效的地址处，w2kSymbolsLoad()函数将使用w2kPeCheck()函数小心的将符号文件的时间戳、校验和（checksum）与目标模块的内存映像的相应域进行比较，如果它们不匹配，则丢弃相应的符号表。

```
PIMG_TABLE WINAPI w2kSymbolsLoad (PBYTE    pbModule,
                                  PDWORD pdStatus)
    {
    PVOID          pBase;
    DWORD          dStatus = W2K_SYMBOLS_UNDEFINED;
    PIMG_TABLE pit         = NULL;
    if ((pBase = imgModuleBaseA (pbModule)) == NULL)
        {
        dStatus = W2K_SYMBOLS_MODULE_ERROR;
        }
    else
        {
        if ((pit = imgTableLoadA (pbModule, pBase)) == NULL)
            {
            dStatus = W2K_SYMBOLS_LOAD_ERROR;
            }
        else
            {
            if (!w2kPeCheck (pbModule, pit->dTimeStamp,
                                       pit->dCheckSum))
                {
                dStatus = W2K_SYMBOLS_VERSION_ERROR;
                pit         = imgMemoryDestroy (pit);
                }
            else
                {
                dStatus = W2K_SYMBOLS_OK;
```

```
                        }
                    }
                }
        if (pdStatus != NULL) *pdStatus = dStatus;
        return pit;
        }
// ------------------------------------------------------------------
PIMG_TABLE WINAPI w2kSymbolsGlobal (PDWORD pdStatus)
        {
        DWORD           dStatus = W2K_SYMBOLS_UNDEFINED;
        PIMG_TABLE pit          = NULL;
        w2kSpyLock ();
        if ((gdStatus == W2K_SYMBOLS_OK) && (gpit == NULL))
                {
                gpit = w2kSymbolsLoad (NULL, &gdStatus);
                }
        dStatus = gdStatus;
        pit         = gpit;
        w2kSpyUnlock ();
        if (pdStatus != NULL) *pdStatus = dStatus;
        return pit;
        }
// ------------------------------------------------------------------
DWORD WINAPI w2kSymbolsStatus (VOID)
        {
        DWORD dStatus = W2K_SYMBOLS_UNDEFINED;
        w2kSymbolsGlobal (&dStatus);
        return dStatus;
        }
```

```
// -------------------------------------------------------------
VOID WINAPI w2kSymbolsReset (VOID)
    {
    w2kSpyLock ();
    gpit        = imgMemoryDestroy (gpit);
    gdStatus = W2K_SYMBOLS_OK;
    w2kSpyUnlock ();
    return;
    }
```

列表 6-33. 符号表管理函数

列表 6-33 底部的 w2kSymbolsStatus()和 w2kSymbolsReset()函数可按需加载/卸载符号表。如果符号表不存在，w2kSymbolsStatus()将尝试加载它并返回其状态代码。如果 w2k_call.dll 已经尝试加载符号表但没有成功的话，那么 w2kSymbolsStatus()将返回一个最后错误代码（见**表 6-2**），除非调用 w2kSymbolsReset()重置符号表。w2kSymbolsReset()将释放符号表占用的内存块（如果有的话），并在下一次请求之前强制重新加载整个符号表，这也包括 ntoskrnl.exe 的符号表。

表 6-2. w2kSymbolsLoad()状态代码

状态代码	描　　述
W2K_SYMBOLS_OK	模块的符号表已加载
W2K_SYMBOLS_MODULE_ERROR	模块没有常住内存
W2K_SYMBOLS_LOAD_ERROR	无法加载模块的符号文件
W2K_SYMBOLS_VERSION_ERROR	符号文件与内存中的模块映像不匹配
W2K_SYMBOLS_UNDEFINED	符号表的状态未定义

列表 6-34 中的 w2kXCopy*()函数集构成了扩展的复制接口，它们分别对应于**列表 6-23** 中的相应函数。w2kXCopy()简单的使用值为负数的 dArgumentBytes 参数调用 w2kXCall()，剩余的复制函数只是一组外包函数，用于提供更简洁的参数列表。

```
BOOL WINAPI w2kXCopy (PULARGE_INTEGER puliResult,
                      PBYTE           pbSymbol,
                      DWORD           dBytes)
```

```
    {
    return w2kXCall (puliResult, pbSymbol,
                            0xFFFFFFFF - dBytes, NULL);
    }
// -------------------------------------------------------------
BYTE WINAPI w2kXCopy08 (BYTE    bDefault,
                            PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;
    return (w2kXCopy (&uliResult, pbSymbol, 1)
            ? (BYTE) uliResult.LowPart
            : bDefault);
    }
// -------------------------------------------------------------
WORD WINAPI w2kXCopy16 (WORD    wDefault,
                            PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;
    return (w2kXCopy (&uliResult, pbSymbol, 2)
            ? (WORD) uliResult.LowPart
            : wDefault);
    }
// -------------------------------------------------------------
DWORD WINAPI w2kXCopy32 (DWORD dDefault,
                             PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;
    return (w2kXCopy (&uliResult, pbSymbol, 4)
            ? uliResult.LowPart
            : dDefault);
```

```
    }
// ------------------------------------------------------------
QWORD WINAPI w2kXCopy64 (QWORD qDefault,
                         PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;
    return (w2kXCopy (&uliResult, pbSymbol, 8)
            ? uliResult.QuadPart
            : qDefault);
    }
// ------------------------------------------------------------
PVOID WINAPI w2kXCopyP (PVOID pDefault,
                        PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;
    return (w2kXCopy (&uliResult, pbSymbol, 4)
            ? (PVOID) uliResult.LowPart
            : pDefault);
    }
// ------------------------------------------------------------
PVOID WINAPI w2kXCopyEP (PVOID pDefault,
                         PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;
    return (w2kXCopy (&uliResult, pbSymbol, 0)
            ? (PVOID) uliResult.LowPart
            : pDefault);
    }
```

列表 6-34. 扩展的复制函数

6.4.8、实现内核函数的 Thunk

实现内部（指未导出的）内核函数的 Thunk 的方式与实现已导出的 API 函数的 Thunk 的方式相同，目前只有 ntoskrnl.exe 中的未导出函数可以被调用。这一限制是由于当前的 w2k_call.dll 中的符号表管理器只加载了 ntoskrnl.exe 的符号表，并不是调用接口自身的限制。仅加载 ntoskrnl.exe 的符号表只是为了让事情更简单的些，因为该模块中包含了大多数我们感兴趣的符号（当然，你可以扩展 w2k_call.dll 以加载多个符号表）。**列表 6-35** 包含两个用于 Windows 2000 对象管理的内部函数的 Thunk，这两个 Thunk 可返回对象类型的相关信息。（对象类型将在第七章里详细讨论）。

列表 6-36 给出了三个非常重要的内部数据结构的 Thunk，第七章的示例代码需要这些 Thunk。注意，我对使用扩展的内核调用接口的 Thunk 的名称之前增加了两个下划线作为前缀。这表示仅在提供相应的符号文件的情况下，才能使用这些 Thunk。如果你安装了 Service Pack 而没有更新相应的符号文件，则 w2kSymbolsLoad()将拒绝加载符号文件（因为时间戳、校验和均不匹配），并且对有双划线前缀的 Thunk 的调用也将失败，这些 Thunk 将返回默认值。换句话说，有一个下划线前缀的 Thunk 仍然可以正常的工作（即使没有匹配的符号文件），因为这些 Thunk 是根据新模块的内存映像的导出表来解析符号的。不过，在更新系统后，如果被更新的模块不再导出我们所引用的 API 函数或者某些函数的参数列表发生了变化，那么它们仍可能失败，

```
NTSTATUS WINAPI
__ObQueryTypeInfo (POBJECT_TYPE          ObjectType,
                   POBJECT_TYPE_INFO TypeInfo,
    /* bytes        */ DWORD                  TypeInfoLength,
    /* init to 0 */ PDWORD                  ReturnLength)
     {
     return w2kXCallNT ("ObQueryTypeInfo",
                        16, ObjectType, TypeInfo, TypeInfoLength,
                            ReturnLength);
     }
// -------------------------------------------------------------
NTSTATUS WINAPI
```

```
__ObQueryTypeName (POBJECT                          Object,
                   POBJECT_NAME_INFORMATION NameString,
        /* bytes */ DWORD                           NameStringLength,
                   PDWORD                           ReturnLength)
    {
    return w2kXCallNT ("ObQueryTypeName",
                       16, Object, NameString, NameStringLength,
                           ReturnLength);
    }
```

列表 6-35. ObQueryTypeInfo()和 ObQueryTypeName()的 Thunk 示列

```
PERESOURCE WINAPI
__ObpRootDirectoryMutex (VOID)
    {
    return w2kXCopyP (NULL, "ObpRootDirectoryMutex");
    }
// -------------------------------------------------------------
POBJECT_DIRECTORY WINAPI
__ObpRootDirectoryObject (VOID)
    {
    return w2kXCopyP (NULL, "ObpRootDirectoryObject");
    }
// -------------------------------------------------------------
POBJECT_DIRECTORY WINAPI
__ObpTypeDirectoryObject (VOID)
    {
    return w2kXCopyP (NULL, "ObpTypeDirectoryObject");
    }
```

列表 6-36. 一些内部变量的 Thunk 示列

现在，已经足够了。不过你或许有一点失望，因为我没有给出使用 w2k_call.dll 的示例代码。不要担心，你会在下一章中得到这些示例代码

七、Windows 2000 的对象管理

在 Windows 2000 内部几乎没有什么比它的对象还有趣。如果可以像观看行星表面一样查看操作系统的内存空间,那么对象看起来就像活在行星上的生物。存在着多种类型的对象，有大有小，有复杂的也有简单的，它们通过多种方式互相影响。一个巧妙的、结构化的对象管理机制是 Windows 2000 的一大特色，不过这一机制几乎没有文档记载。本章将试图带你深入这个庞大而复杂的世界里。很不幸的是，Windows 2000 的这一部分是微软保护最好的秘密，许多问题在这里仍然没有答案。不过，我希望本章能成为一个起点，以帮助我们探索“在此之前，还没有人去过的地方”。

7.1、Windows 2000 对象的结构

本书光盘上有一个很大的头文件名为：w2k_def.h，它位于\src\common\include 目录下，它为 Windows 2000 系统编程者那颗痛苦跳动着的心带来了一丝喜悦。该文件包含多年来深入研究 Windows NT/200 所获取的常量和类型定义。w2k_def.h 可以用于 Win32 应用程序也可用于内核模式的驱动程序中，它使用条件编译选项来区分不同的构建环境。例如，Win32 应用程序无法使用 ntdef.h 和 ntddk.h，因为这俩个文件包含很多内核数据类型的定义。因此，w2k_def.h 包含了所有可以在 DDK 头文件中找到的#define 和 typedef 定义，未文档化的内容需要这些定义。为了在构建内核模式的驱动程序时，出现避免重定义错误，这些定义被放入了#ifdef _USER_MODE_子句中，因此，如果没有定义_USER_MDE_没有定义，编译器将忽略它们。这意味着你必须在包含 w2k_def.h 的源代码中放入#define _USER_MODE_，这一语句必须位于#include “w2k_def.h”之前，这样才能允许预处理器在 Win32 应用程序或 DLL 中处理 DDK 中的定义。#ifdef _USER_MODE_的#else 子句包含少量 Windows 2000 DDK 头文件中缺失的定义，如 SECURITY_DESCRIPTOR 和 SECURITY_DESCRIPTOR_CONTROL 类型。

7.1.1、对象的基本分类

尽管对象是 Windows 2000 送给我们的一个大礼，但在 DDK 中你很难发现有关它们的内部结构的有用信息。由 ntoskrnl.exe 导出的 21 个用于对象管理的 Ob*()函数，仅有 6 个出现在 DDK 文档中。这些函数接受一个指向对象的指针作为参数，该指针参数的类型通常为 PVOID。如果你在 DDK 的主要头文件 ntdef.h 和 ntddk.h 中查找与对象相关的类型定义，你会发现几乎没有什么有用的信息。有些重要的对象数据类型仅被定义为一个占位符

（placeholder）。例如，OBJECT_TYPE 结构出现方式为：typedef struct _OBJECT_TYPE *POBJECT_TYPE;这样做只是为了让编译器高兴而以，没有透露出任何有关该对象内部的信息。

无论你在何时遇到对象指针，你都会发现从其线形地址来看，它将常驻内存的结构体划分为两部分：一个对象头和一个对象体。对象指针并没有指向对象自身的基地址，而是指向了对象体，由于紧接着对象头的就是对象体，所以可以给对象指针加上一个负的偏移量来访问对象头。对象体的内部结构完全依赖于对象的类型，对于不同的类型其差别很大。最简单的对象是 Event 对象，该对象只有一个 16 字节的对象体。最复杂对象的代表就是进程和线程对象，这俩个对象的大小达到了几百字节。基本上，可以将对象体的类型划分为如下三个主要类别：

1. **Dispatcher 对象** 此类对象位于系统的最底层，在它们的对象体的开始处都有一个共享的公共数据结构----DISPATCHER_HEADER（参见**列表 7-1**）。在它们的对象头中包含一个对象类型 ID 和对象体的长度（保存在 32 位的 DWORD 变量中）。所有 Dispatcher 对象结构体的名字都以字母 K 开始，这表示它们是内核（Kernel）对象。
   DISPATCHER_HEADER 结构的存在使得对象是“可等待的（waitable）”。这意味着，此种类型的对象可以被传递给同步函数 KeWaitForSingleObject()和 KeWaitForMultipleObjects()，Win32 函数 WaitForSingleObject()和 WaitForMultipleObjects()就构建于它们之上。

```
typedef struct _DISPATCHER_HEADER

        {

/*000*/ BYTE       Type;          // DISP_TYPE_*

/*001*/ BYTE       Absolute;

/*002*/ BYTE       Size;          // number of DWORDs

/*003*/ BYTE       Inserted;

/*004*/ LONG       SignalState;

/*008*/ LIST_ENTRY WaitListHead;
```

```
/*010*/ }

        DISPATCHER_HEADER,

     * PDISPATCHER_HEADER,

    **PPDISPATCHER_HEADER;

#define DISPATCHER_HEADER_ \

        sizeof (DISPATCHER_HEADER)
```

列表 7-1. DISPATCHER_HEADER 结构的定义

译注：

可以在本书 CD 的\src\common 目录下的 w2k_def.h 中找到该结构的定义。

2. **I/O 系统数据结构（I/O 对象）** 这类对象是最高层的对象，其对象体的开始位置是一个 SHORT 类型的成员，该成员用来标识该对象的 ID。通常，此 ID 之后还有一个 SHORT 或 WORD 类型的成员用来记录对象体的大小。不过，此类对象并不都遵守这一规则。
3. **其他对象** 不属于上述两种对象的对象。

从现在起就要注意 Dispatcher 对象和 I/O 对象的类型 ID，因为这些 ID 都是单独维护的，所以有些 ID 可能会发生重复。**表 7-1** 给出了我所知道的 Dispatcher 对象的类型。出现在表 7-1 的“C 结构”一栏的某些结构体在 DDK 头文件 ntddk.h 中有相应的定义。但很不幸的是，很多非常重要的结构，如 KPROCESS 和 KTHREAD 都没有官方给出的定义（译注：可以理解，这些结构在未来的版本中可能会发生变化）。不过无需担心，稍后我们将详细讨论这些特殊对象类型的细节。在 w2k_def.h 中你可以找到所有未文档化的结构体（当然这仅限于我了解得那些）的定义，附录 C 也会列出这些结构体的定义。

表 7-1. Dispatcher 对象汇总

ID	类　型	C 结构	定　义
0	DISP_TYPE_NOTIFICATION_EVENT	KEVENT	ntddk.h
1	DISP_TYPE_SYNCHRONIZATION_EVENT	KEVENT	ntddk.h

2	DISP_TYPE_MUTANT	KMUTANT,KMUTEX	ntddk.h
3	DISP_TYPE_PROCESS	KPROCESS	w2k_def.h
4	DISP_TYPE_QUEUE	KQUEUE	w2k_def.h
5	DISP_TYPE_SEMAPHORE	KSEMAPHORE	ntddk.h
6	DISP_TYPE_THREAD	KTHREAD	w2k_def.h
8	DISP_TYPE_NOTIFICATION_TIMER	KTIMER	ntddk.h
9	DISP_TYPE_SYNCHRONIZATION_TIMER	KTIMER	ntddk.h

表 7-2 列出了到目前为止所有我知道的 I/O 对象。仅有前 13 个可以在 ntddk.h 中找到它们的定义，

表 7-2. I/O 对象汇总

ID	类　型	C 结构	定　义
1	IO_TYPE_AD_APTER	ADAPTER_OBJECT	
2	IO_TYPE_CONTROLLER	CONTROLLER_OBJECT	ntddk.h
3	IO_TYPE_DEVICE	DEVICE_OBJECT	ntddk.h
4	IO_TYPE_DRIVER	DRIVER_OBJECT	ntddk.h
5	IO_TYPE_FILE	FILE_OBJECT	ntddk.h
6	IO_TYPE_IRP	IRP	ntddk.h
7	IO_TYPE_MASTER_ADAPTER		
8	IO_TYPE_OPEN_PACKET		
9	IO_TYPE_TIMER	IO_TIMER	w2k_def.h
10	IO_TYPE_VPB	VPB	ntddk.h
11	IO_TYPE_ERROR_LOG	IO_ERROR_LOG_ENTRY	w2k_def.h
12	IO_TYPE_ERROR_MESSAGE	IO_ERROR_LOG_MESSAGE	ntddk.h
13	IO_TYPE_DEVICE_OBJECT_EXTENSION	DEVOBJ_EXTENSION	ntddk.h
18	IO_TYPE_APC	KAPC	ntddk.h
19	IO_TYPE_DPC	KDPC	ntddk.h
20	IO_TYPE_DEVICE_QUEUE	KDEVICE_QUEUE	ntddk.h

21	IO_TYPE_EVENT_PAIR	KEVENT_PAIR	w2k_def.h
22	IO_TYPE_INTERRUPT	KINTERRUPT	
23	IO_TYPE_PROFILE	KPROFILE	

7.1.2、对象的表头（header）

Windows 2000 对于对象体的大小和结构没有任何限制。与此相反的是，对象的表头则几乎没有什么自由可言。**图 7-1** 给出了一个对象实例的内存布局，该实例拥有完整的对象特性，并且其拥有最多的表头字段（header fields）。每一种对象特性都至少拥有一个基本的 OBJECT_HEADER 结构，紧随该结构之后就是对象体，在对象体之前最多可有四个可选的结构，这些结构用于提供了有关对象的附加信息。前面已经提到过，一个对象指针总是指向对象体，而不是对象的表头，因此需要在对象指针上增加一个负的偏移量才能访问表头字段。基本的表头包含有关可选表头字段的位置及其是否存在的信息，这些可选字段都位于 OBJECT_HEADER 结构之上，其顺序如**图 7-1** 所示。不过，这种顺序并不是强制的，你的应用程序不应该依赖这一顺序。OBJECT_HEADER 结构中的信息足够定位所有的表头字段（唯一不能确定的就是它们的顺序）。唯一列外的是：如果 OBJECT_CREATOR_INFO 结构存在的话，那么它将总是出现在 OBJECT_HEADER 结构之前

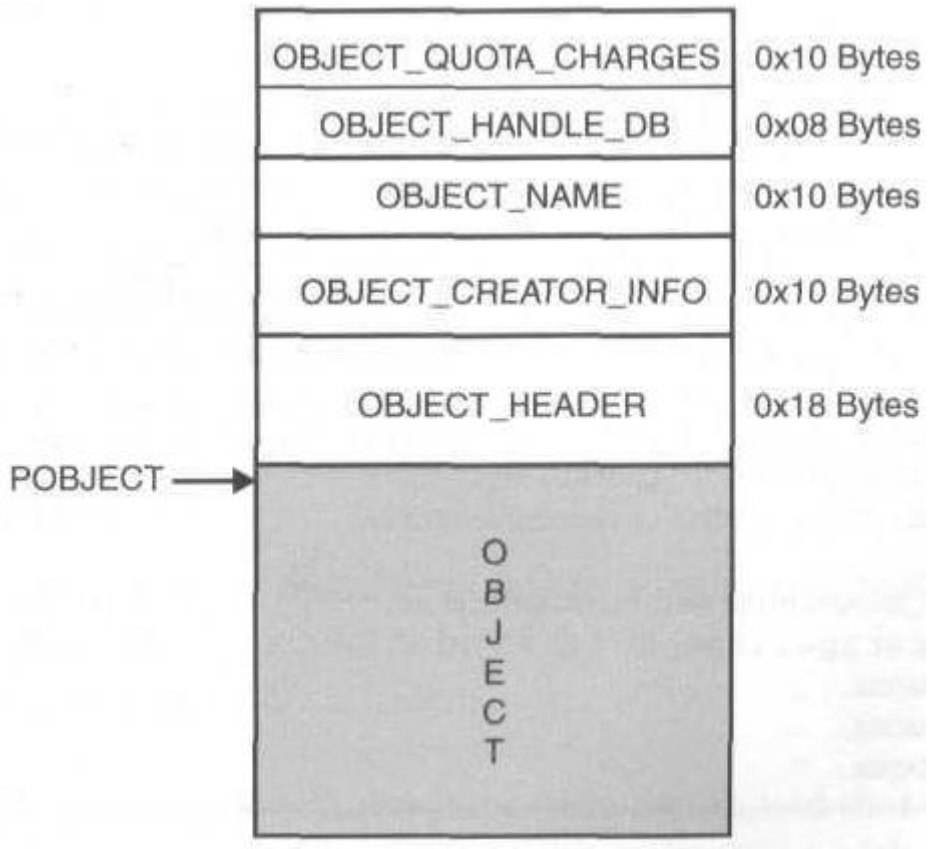


图 7-1. 对象的内存布局

```
#define OB_FLAG_CREATE_INFO      0x01 // has OBJECT_CREATE_INFO

#define OB_FLAG_KERNEL_MODE    0x02 // created by kernel

#define OB_FLAG_CREATOR_INFO    0x04 // has OBJECT_CREATOR_INFO

#define OB_FLAG_EXCLUSIVE        0x08 // OBJ_EXCLUSIVE

#define OB_FLAG_PERMANENT       0x10 // OBJ_PERMANENT

#define OB_FLAG_SECURITY          0x20 // has security descriptor

#define OB_FLAG_SINGLE_PROCESS   0x40 // no HandleDBList

typedef struct _OBJECT_HEADER

        {

/*000*/ DWORD        PointerCount;       // number of references

/*004*/ DWORD        HandleCount;        // number of open handles

/*008*/ POBJECT_TYPE  ObjectType;

/*00C*/ BYTE         NameOffset;         // -> OBJECT_NAME

/*00D*/ BYTE         HandleDBOffset;     // -> OBJECT_HANDLE_DB

/*00E*/ BYTE         QuotaChargesOffset; // -> OBJECT_QUOTA_CHARGES

/*00F*/ BYTE         ObjectFlags;        // OB_FLAG_*

/*010*/ union

            { // OB_FLAG_CREATE_INFO ? ObjectCreateInfo : QuotaBlock

/*010*/         PQUOTA_BLOCK        QuotaBlock;
```

```
/*010*/         POBJECT_CREATE_INFO  ObjectCreateInfo;

/*014*/     };

/*014*/ PSECURITY_DESCRIPTOR  SecurityDescriptor;

/*018*/ }

        OBJECT_HEADER,

     * POBJECT_HEADER,

    **PPOBJECT_HEADER;

#define OBJECT_HEADER_ \

        sizeof (OBJECT_HEADER)
```

列表 7-2. OBJECT_HEADER 结构定义

列表 7-2 给出了 OBJECT_HEADER 结构的定义，该结构中的各个成员分别用于如下目的：

- PonterCount 成员用于记录当前有多少指针指向此对象。这与 COM 中的引用计数非常类似。ntoskrnl.exe 中的 ObReferenceObject()、ObReferenceObjectByHandle()、ObReferenceObjectByName()和 ObReferenceObjectByPointer()函数用于增加 PointerCount，而 ObfDereferenceObject()和 ObDereferenceObject()则用于减少 PointerCount。
- HandleCount 成员用来记录当前有多少个打开的句柄引用了此对象。
- ObjectType 成员指向一个 OBJECT_TYPE 结构（稍后将讨论），该结构用来代表一个类型对象，在对象的创建中将使用该结构。
- NameOffset 成员给出了对象表头中的 OBJECT_NAME 结构的地址(用 OBJECT_HEADER 结构的地址减去 NameOffset 就可得到），如果为 0，则表示对象表头中不存在 OBJECT_NAME 结构。
- HandleDBOffset 成员给出了对象表头中的 OBJECT_HANDLE_DB 结构的地址（用 OBJECT_HEADER 结构的地址减去 HandleDBOffset 就可得到），如果为 0，则表示对象表头中不存在 OBJECT_HANDLE_DB 结构。

- QuotachargesOffset 成员给出了对象表头中的 OBJECT_QUOTA_CHARGES 结构的地址（用 OBJECT_HEADER 结构的地址减去 QuotachargesOffset 就可得到），如果为 0，则表示对象表头中不存在 OBJECT_QUOTA_CHARGES 结构。
- ObjectFlags 指出了对象的多个属性（每个属性占用一个二进制位），列表 7-2 的顶部列出了这些属性。如果 OB_FLAG_CREATOR_INFO 位被设置了，那么就意味着在对象表头中存在一个 OBJECT_CREATOR_INFO 结构，该结构紧挨着 OBJECT_HEADER，并位于 OBJECT_HEADER 之前。在《Windows NT/2000 Native API Reference》中，Gray Nebbett 在介绍 ZwQuerySystemInformation()函数所使用的 SystemObjectInformation 标志时提到过对象属性，不过他的描述与本书给出的略有不同。**表 7-3** 给出了它们之间的区别。
- QuotaBlock 和 ObjectCreateInfo 成员是互斥的（它们两个构成了一个 union）。如果 ObjectFlags 成员的 OB_FLAG_CREATE_INFO 标志被设置，那么该成员（此时为 ObjectCreateInfo）将包含一个指向 OBJECT_CREATE_INFO 结构（稍后介绍）的指针。否则，该成员（此时为 QuotaBlock）将指向一个 QUOTA_BLOCK 结构，该结构提供了 Paged 和 NonPaged 内存池的使用量的相关信息。很多对象都将它们的 QuotaBlock 指针指向系统内部的 PspDefaultQuotaBlock 结构。该 union 可以为 NULL。
- SecurityDescriptor 成员　如果 ObjectFlags 成员的 OB_FLAG_SECURITY 位被设置的话，那么该成员将指向一个 SECURITY_DESCRIPTOR 结构，否则该成员为 NULL。

上面的列表中提及到多个结构现在还没有介绍，接下来我们将开始介绍它们，首先要介绍的是**图 7-1** 中的四个可选表头字段。

表 7-3.　ObjectFlags 描述对比

本书的描述	值	《Windows NT/2000 Native API Reference》
OB_FLAG_CREATE_INFO	0x01	N/A
OB_FLAG_KERNEL_MODE	0x02	KERNEL_MODE
OB_FLAG_CREATOR_INFO	0x04	CREATOR_INFO
OB_FLAG_EXCLUSIVE	0x08	EXCLUSIVE
OB_FLAG_PERMANENT	0x10	PERMANENT
OB_FLAG_SECURITY	0x20	DEFAULT_SECURITY_QUOTA
OB_FLAG_SINGLE_PROCESS	0x40	SINGLE_HANDLE_ENTRY

7.1.3、对象创建者的相关信息

如果 ObjectFlags 成员的 OB_FLAG_CREATOR_INFO 位被设置，那么对象的 OBJECT_HEADER 将紧随 OBJECT_CREATOR_INFO 结构之后。**列表 7-3** 给出了该可选结构的定义。OBJECT_CREATOR_INFO 结构中的 ObjectList 成员是一个双向链表（参见第二章的列表 2-7）中的一个节点。该双向链表由类型相同的对象构成。默认情况下，仅有 Port 和 WaitablePort 对象在它们的对象表头中包含 OBJECT_CREATOR_INFO 结构。使用 SystemObjectInformation 标志的 ZwQuerySystemInformation()函数就是使用 ObjectList 来返回当前已分配对象的完整列表，这些对象将按照对象类型进行分类。在《Windows NT/2000 Native API Reference》中，Gray Nebbett 指出“[….]仅在系统启动时，使用 NtGlobalFlags()设置了 FLG_MAINTAIN_OBJECT_TYPELIST，该信息标志才有效”（Nebbett 2000, p.25）

```
typedef struct _OBJECT_CREATOR_INFO

        {

/*000*/ LIST_ENTRY ObjectList;      // OBJECT_CREATOR_INFO

/*008*/ HANDLE     UniqueProcessId;

/*00C*/ WORD       Reserved1;

/*00E*/ WORD       Reserved2;

/*010*/ }

        OBJECT_CREATOR_INFO,

     * POBJECT_CREATOR_INFO,

    **PPOBJECT_CREATOR_INFO;

#define OBJECT_CREATOR_INFO_ \

        sizeof (OBJECT_CREATOR_INFO)
```

列表 7-3. OBJECT_CREATOR_INFO 结构的定义

OBJECT_CREATOR_INFO 结构中的 UniqueProcessId 成员是一个从 0 开始的数字，它实际上是创建对象的进程的 ID。尽管该成员被定义为一个句柄，但实际上它并不是一个句柄。它将被解释为一个不透明的 32 位无符号整数。Win32 函数 GetCurrentProcessId()返回的数据类型实际为 DWORD。

7.1.4、OBJECT_NAME 结构

如果 OBJECT_HEADER 结构中的 NameOffset 成员是一个非零值，那么就意味着存在着 OBJECT_NAME 结构，NameOffset 的值就是该结构相对于 OBJECT_HEADER 的基地址的负偏移量。该负偏移量通常是 0x10 或 0x20，这依赖于是否存在 OBJECT_CREATOR_INFO 结构。**列表 7-4** 给出了 OBJECT_NAME 结构的定义。OBJECT_NAME 结构的 Name 成员是一个 UNICODE_STRING 结构，该结构的 Buffer 成员指向实际的名称字符串，该字符串占用的内存并不包含在对象中。并不是所有的命名对象都使用一个 OBJECT_NAME 结构来存储自己的名称。例如，有些对象使用 QueryNameProcedure()来获取一个名字。QueryNameProcedure()会根据与这些对象相关联的 OBJECT_TYPE 来产生这一名字。

OBJECT_NAME 结构的 Directory 成员不为 NULL，那么它将指向一个目录对象（OBJECT_DIRECTORY），该对象用于记录该对象在系统的对象层次结构中位于哪一层。类似于文件位于文件系统中，Windows 2000 的对象也保存在由目录对象和叶子对象（leaf object）构成的层次化的树型结构中。稍后我们将详细介绍 OBJECT_DIRECTORY 结构。

```
typedef struct _OBJECT_NAME

        {

/*000*/ POBJECT_DIRECTORY Directory;

/*004*/ UNICODE_STRING    Name;

/*00C*/ DWORD             Reserved;

/*010*/ }

        OBJECT_NAME,

     * POBJECT_NAME,
```

```
    **PPOBJECT_NAME;

#define OBJECT_NAME_ \

        sizeof (OBJECT_NAME)
```

列表 7-4. OBJECT_NAME 结构

7.1.5、OBJECT_HANDLE_DATABASE 结构

某些对象所维护的进程相关的句柄计数保存在一个称为“句柄数据库”的结构中。在这种情况下，OBJECT_HEADER 结构的 HandleDBOffset 成员将包含一个非零值。和前面讨论的 NameOffset 类似，这一偏移量也是相对于 OBJECT_HEADER 基地址的负偏移量。**列表 7-5** 给出了 OBJECT_HANDLE_DB 结构的定义。如果 ObjectFlags 的 OB_FLAG_SINGLE_PROCESS 标志被设置，那么 OBJECT_HANDLE_DB 结构中的 Process 成员（位于该结构的一个 union 子结构中）将指向一个进程对象。如果一个以上的进程持有该对象的句柄，那么 OB_FLAG_SINGLE_PROCESS 标志将被清除，此时 HandleDBList 成员（该成员与前面的 Process 成员一起构成了 OBJECT_HANDLE_DB 结构中的一个 union 子结构）将指向一个 OBJECT_HANDLE_DB_LIST 结构，该结构包含一个 OBJECT_HANDLE_DB 结构数组。

```
typedef struct _OBJECT_HANDLE_DB

        {

/*000*/ union

            {

/*000*/     struct _EPROCESS              *Process;

/*000*/     struct _OBJECT_HANDLE_DB_LIST *HandleDBList;

/*004*/     };

/*004*/ DWORD HandleCount;

/*008*/ }
```

```
        OBJECT_HANDLE_DB,

     * POBJECT_HANDLE_DB,

    **PPOBJECT_HANDLE_DB;

#define OBJECT_HANDLE_DB_ \

        sizeof (OBJECT_HANDLE_DB)

// ------------------------------------------------------------

typedef struct _OBJECT_HANDLE_DB_LIST

        {

/*000*/ DWORD            Count;

/*004*/ OBJECT_HANDLE_DB Entries [];

/*???*/ }

        OBJECT_HANDLE_DB_LIST,

     * POBJECT_HANDLE_DB_LIST,

    **PPOBJECT_HANDLE_DB_LIST;

#define OBJECT_HANDLE_DB_LIST_ \

        sizeof (OBJECT_HANDLE_DB_LIST)
```

列表 7-5. OBJECT_HANDLE_DB 结构

7.1.6、资源使用费用（Charges）和使用限额（Quota）

如果一个进程打开一个对象的句柄，那么该进程必须为该操作所使用的系统资源“付费”。我们把应付款称为费用，进程可以使用的资源上限称为限额。在 DDK 文档的术语表中，微软以如下的方式定义了“限额”：

限额（Quota）：

针对每个进程可使用的系统资源的限制。

针对每个进程，Windows NT/Windows 2000 对某些系统资源设置了限制，这些资源是进程的线程需要使用的，这包括针对 Paging-file、 Paged-pool 和 Nonpaged-pool 的使用限额等等。例如，内存管理器使用限额来限制线程对 page-file、paged-pool 或 nonpaged-pool 内存的使用；当线程释放占用的内存后，管理器会更新配这些值。（Windows 2000 DDK\Kernel-Mode Drivers\Design Guide\Kernel-Mode Glossary\Q\Quota）

默认情况下，对象的 OBJECT_TYPE 用来确定 paged/nonpaged poll 的使用费用以及因为安全而产生的相关费用。不过，通过向对象中增加一个 OBJECT_QUOTA_CHARGES 结构可以改变这种默认设置。该结构相对于 OBJECT_HEADER 的偏移量由 OBJECT_HEADER 的 QuotaChargesOffset 成员给出，该偏移量是一个负偏移量。**列表 7-6** 给出了 OBJECT_QUOTA_CHARGES 结构的定义。Paged 和 Nonpaged Pool 的使用量是独立统计的。如果对象要求使用安全特性，那么附加的 SecurityCharge 将被增加到 paged-pool 的使用量上。默认的安全开销是 0x800。

如果 ObjectFlags 成员（属于 OBJECT_HEADER 结构）的 OB_FLAG_CREATE_INFO 标志位被设为 0，那么 QuotaBlock 成员将指向一个 QUOTA_BLOCK 结构（见**列表 7-7**），该结构包含有关对象的资源使用量的统计信息。

```
#define OB_SECURITY_CHARGE 0x00000800

typedef struct _OBJECT_QUOTA_CHARGES

        {

/*000*/ DWORD PagedPoolCharge;

/*004*/ DWORD NonPagedPoolCharge;

/*008*/ DWORD SecurityCharge;

/*00C*/ DWORD Reserved;

/*010*/ }
```

```
        OBJECT_QUOTA_CHARGES,

     * POBJECT_QUOTA_CHARGES,

    **PPOBJECT_QUOTA_CHARGES;

#define OBJECT_QUOTA_CHARGES_ \

        sizeof (OBJECT_QUOTA_CHARGES)

typedef struct _QUOTA_BLOCK

        {

/*000*/ DWORD Flags;

/*004*/ DWORD ChargeCount;

/*008*/ DWORD PeakPoolUsage [2]; // NonPagedPool, PagedPool

/*010*/ DWORD PoolUsage     [2]; // NonPagedPool, PagedPool

/*018*/ DWORD PoolQuota     [2]; // NonPagedPool, PagedPool

/*020*/ }

        QUOTA_BLOCK,

     * PQUOTA_BLOCK,

    **PPQUOTA_BLOCK;

#define QUOTA_BLOCK_ \

        sizeof (QUOTA_BLOCK)
```

列表 7-7. QUOTA_BLOCK 结构

7.1.7、对象目录

在讨论 OBJECT_HEADER 表头时我们已经提到过对象目录对象，Windows 2000 对象管理器将独立的对象保存在一个树型的 OBJECT_DIRECTORY 结构中，该结构一般称为“目录对象”。一个 OBJECT_DIRECTORY 结构是另一个较复杂的对象类型，和 OBJECT_HEADER 对象类似，它也是一个实际的对象所必需的。Windows 2000 管理对象目录的方式非常巧妙。如**列表 7-8** 所示，OBJECT_DIRECTORY 本质上是一个 Hash 表，该表最多可容纳 37 个表项。之所以选择这 37 或许是因为这是一个质数吧 ☺。每个表项可存放一个指向 OBJECT_DIRECTORY_ENTRY 结构的指针，OBJECT_DIRECTORY_ENTRY 结构的 Object 成员是一个指向实际对象的指针。当一个对象被创建时，对象管理器根据对象的名字来计算该对象的 Hash 值（该值范围为 0---36）并为该对象创建一个 OBJECT_DIRECTORY_ENTRY。如果 Hash 表中的目标表项为空，那么前面创建的 OBJECT_DIRECTORY_ENTRY 结构的指针将被保存在该表项中。如果目标表项已经被使用了，那么新的 OBJECT_DIRECTORY_ENTRY 结构将被插入一个单向链表中，该链表的表头就位于该表项中。OBJECT_DIRECTORY_ENTRY 结构中的 NextEntry 成员就是用来构建此单向链表的。为了表现出对象之间的层次关系，对象目录可以是嵌套的：通过增加一个新的 OBJECT_DIRECTORY_ENTRY 结构，并将该结构的 Object 成员指向其下属目录对象。

为了优化对频繁使用的对象的访问，对象管理器采用了简单的最近最多使用算法（most recently used，MRU）。只要可以成功的访问某个对象，那么该对象将被移动到它所在的单向链表的前端（**译注：**这里的对象目录对象构成的结构很像数据结构中的邻接表与 Hash 表的结合，采用单向链表来组织 Hash 值相同的目录对象，是 Hash 技术中最常用的处理冲突的方法）。更新后的链表指针保存在 OBJECT_DIRECTORY 结构中的 CurrentEntry 成员中。CurrentEntryValid 标志用来表示 CurrentEntry 是否有效。通过使用一个称为 ObpRootDirectoryMutex 的资源锁可实现同步访问系统全局对象目录。这个锁既没有文档记录也没有导出。

```
typedef struct _OBJECT_DIRECTORY_ENTRY

        {

/*000*/ struct _OBJECT_DIRECTORY_ENTRY *NextEntry;

/*004*/ POBJECT                        Object;

/*008*/ }
```

```
        OBJECT_DIRECTORY_ENTRY,

     * POBJECT_DIRECTORY_ENTRY,

    **PPOBJECT_DIRECTORY_ENTRY;

#define OBJECT_DIRECTORY_ENTRY_ \

        sizeof (OBJECT_DIRECTORY_ENTRY)

// ------------------------------------------------------------

#define OBJECT_HASH_TABLE_SIZE 37

typedef struct _OBJECT_DIRECTORY

        {

/*000*/ POBJECT_DIRECTORY_ENTRY HashTable [OBJECT_HASH_TABLE_SIZE];

/*094*/ POBJECT_DIRECTORY_ENTRY CurrentEntry;

/*098*/ BOOLEAN                 CurrentEntryValid;

/*099*/ BYTE                    Reserved1;

/*09A*/ WORD                    Reserved2;

/*09C*/ DWORD                   Reserved3;

/*0A0*/ }

        OBJECT_DIRECTORY,

     * POBJECT_DIRECTORY,

    **PPOBJECT_DIRECTORY;

#define OBJECT_DIRECTORY_ \
```

```
        sizeof (OBJECT_DIRECTORY)
```

列表 7-8. OBJECT_DIRECTORY 和 OBJECT_DIRECTORY_ENTRY 结构

7.1.8、OBJECT_TYPE 结构

在前面讨论对象表头字段时，总是涉及到“类型对象”或 OBJECT_TYPE 结构，那么现在就让我们开始讨论它们。正式的来讲，一个类型对象是一种特殊类型的对象，它可以是一个事件（event）、设备（device）或者进程（process），也可是一个 OBJECT_HEADER 或其它可选的对象表头子结构。和普通对象的唯一的不同是：类型对象使用一种特殊的方式与其他对象发生关联。一个类型对象属于“主要对象（master object）”类别，此类对象的作用是：定义同类对象的公共属性，并可以将这些对象的子对象保存到一个双向链表中。因此，类型对象也常被称为“对象类型”，以强调它们与普通对象的区别。

类型对象的对象体由一个 OBJECT_TYPE 结构体和一个内嵌的 OBJECT_TYPE_INITIALIZER 结构组成，**列表 7-9** 给出了这两个结构体的定义。ObCreateObject()在创建对象的过程中，使用 OBJECT_TYPE_INITIALIZER 结构来构建适当的对象头。例如，ObpAllocateObject()（由 ntoskrnl.exe 导出）分别使用 MaintainHandleCount 和 MaintainTypeList 来决定新创建的对象表头中是否需要包含 OBJECT_HANDLE_DB 和 OBJECT_CREATOR_INFO 结构。如果设置了 MaintainTypeList 标志，那么此类型的对象将会构成一个双向链表，OBJECT_TYPE 结构中的 ObjectListHead 成员是该链表的表头和表尾（这意味着该双向链表是一个环形结构）。OBJECT_TYPE_INITIALIZER 结构中的 DefaultPagedPoolCharge 和 DefaultNonPagedPoolCharge 成员提供了默认的限额和开销大小（在前面讨论 OBJECT_QUOTA_CHARGES 结构时曾讨论过限额问题）。

```
typedef struct _OBJECT_TYPE_INITIALIZER

        {

/*000*/ WORD            Length;          //0x004C

/*002*/ BOOLEAN         UseDefaultObject;//OBJECT_TYPE.DefaultObject

/*003*/ BOOLEAN         Reserved1;

/*004*/ DWORD           InvalidAttributes;
```

```
/*008*/ GENERIC_MAPPING GenericMapping;

/*018*/ ACCESS_MASK     ValidAccessMask;

/*01C*/ BOOLEAN         SecurityRequired;

/*01D*/ BOOLEAN         MaintainHandleCount; // OBJECT_HANDLE_DB

/*01E*/ BOOLEAN         MaintainTypeList;    // OBJECT_CREATOR_INFO

/*01F*/ BYTE            Reserved2;

/*020*/ BOOL            PagedPool;

/*024*/ DWORD           DefaultPagedPoolCharge;

/*028*/ DWORD           DefaultNonPagedPoolCharge;

/*02C*/ NTPROC          DumpProcedure;

/*030*/ NTPROC          OpenProcedure;

/*034*/ NTPROC          CloseProcedure;

/*038*/ NTPROC          DeleteProcedure;

/*03C*/ NTPROC_VOID     ParseProcedure;

/*040*/ NTPROC_VOID     SecurityProcedure; // SeDefaultObjectMethod

/*044*/ NTPROC_VOID     QueryNameProcedure;

/*048*/ NTPROC_BOOLEAN  OkayToCloseProcedure;

/*04C*/ }

        OBJECT_TYPE_INITIALIZER,

     * POBJECT_TYPE_INITIALIZER,
```

```
    **PPOBJECT_TYPE_INITIALIZER;

#define OBJECT_TYPE_INITIALIZER_ \

        sizeof (OBJECT_TYPE_INITIALIZER)

// -------------------------------------------------------------

typedef struct _OBJECT_TYPE

        {

/*000*/ ERESOURCE      Lock;

/*038*/ LIST_ENTRY     ObjectListHead; // OBJECT_CREATOR_INFO

/*040*/ UNICODE_STRING ObjectTypeName; // see above

/*048*/ union

            {

/*048*/     PVOID DefaultObject; // ObpDefaultObject

/*048*/     DWORD Code;          // File: 5C, WaitablePort: A0

            };

/*04C*/ DWORD                   ObjectTypeIndex; // OB_TYPE_INDEX_*

/*050*/ DWORD                   ObjectCount;

/*054*/ DWORD                   HandleCount;

/*058*/ DWORD                   PeakObjectCount;

/*05C*/ DWORD                   PeakHandleCount;

/*060*/ OBJECT_TYPE_INITIALIZER ObjectTypeInitializer;
```

```
/*0AC*/ DWORD                    ObjectTypeTag;   // OB_TYPE_TAG_*

/*0B0*/ }

        OBJECT_TYPE,

     * POBJECT_TYPE,

    **PPOBJECT_TYPE;

#define OBJECT_TYPE_ \

        sizeof (OBJECT_TYPE)
```

列表 7-9. OBJECT_TYPE 和 OBJECT_TYPE_INITIALIZER 结构

由于类型对象（也称对象类型）是 Windows 2000 对象世界的最基本的构建单位，ntoskrnl.exe 在命名变量中保存它们，将其和对象的 ObjectType（位于对象的 OBJECT_HEADER 中）进行比较就可以很容易的验证对象的类型。类型对象是唯一的---系统只针对每类对象创建一个类型对象。**表 7-4** 给出了 Windows 2000 涉及到的所有类型对象。该表各列的含义如下：

表 7-4. 有效的对象类型

索引	标志	名称	C 结构	是否公开	符号
1	“ObjT”	“Type”	OBJECT_TYPE	No	ObpTypeObjectType
2	“Dire”	“Directory”	OBJECT_DIRECTORY	No	ObpDirectoryObjectType
3	“Symb”	“Symbolic Link”		No	ObpSymbolicLinkObjectType
4	“Toke”	“Token”	TOKEN	No	SepTokenObjectType
5	“Proc”	“Process”	EPROCESS	Yes	PsProcessType

6	“Thre”	“Thread”	ETHREAD	Yes	PsThreadType
7	“Job”	“Job”		Yes	PsJobType
8	“Even”	“Event”	KEVENT	Yes	ExEventObjectType
9	“Even”	“EventPair”	KEVENT_PAIR	No	ExEventPairObjectType
10	“Muta”	“Mutant”	KMUTANT	No	ExMutantObjectType
11	“Call”	“Callback”	CALLBACK_OBJECT	No	ExCallbackObjectType
12	“Sema”	“Semaphore”	KSEMAPHORE	Yes	ExSemaphoreObjectType
13	“Time”	“Timer”	ETIMER	No	ExTimerObjectType
14	“Prof”	“Profile”	KPROFILE	No	ExProfileObjectType
15	“Wind”	“WindowStation”		Yes	ExWindowStationObjectType
16	“Desk”	“Desktop”		Yes	ExDesktopObjectType
17	“Sect”	“Section”		Yes	MmSectionObjectType
18	“Key”	“Key”		No	CmpKeyObjectType
19	“Port”	“Port”		Yes	LpcPortObjectType
20	“Wait”	“WaitablePort”		No	LpcWaitablePortObjectType
21	“Adap”	“Adapter”	ADAPTER_OBJECT	Yes	IoAdapterObjectType
22	“Cont”	“Controller”	CONTROLLER_OBJECT	No	IoControllerObjectType
23	“Devi”	“Device”	DEVICE_OBJECT	Yes	IoDeviceObjectType
24	“Driv”	“Driver”	DRIVER_OBJECT	Yes	IoDriverObjectType

25	“IoCo”	“IoCompletion”	IO_COMMPLETION	No	IoCompletionObjectType
26	“File”	“File”	FILE_OBJECT	Yes	IoFileObjectType
27	“WmiG”	“WmiGuid”	GUID	No	WmipGuidObjectType

- “索引”列对应 OBJECT_TYPE 结构中的 ObjectTypeIndex 成员
- “标志”列是一个 32 位的标识符，它保存在 OBJECT_TYPE 结构中的 ObjectTypeTag 成员中。Windows 2000 的标识符是由四个 ANSI 字符表示的二进值。在调试时，这些字符在 16 进制 Dump 中可很容易识别出来。通过检查 ObjectTypeTag 可以很容易的确定给定的对象是否是我们所期望的类型。在分配对象所占用的内存时，Windows 2000 将 ObjectTypeTag 与 0x80000000 进行逻辑或运算后的值作为新内存块的标志。
- “名称”列给出了对象的名称，该列对应于类型对象的 OBJECT_NAME 结构。很明显类型标志是由对象名称的前四个字符构成的。如果对象名称不足四个，则用空格代替。
- “C 结构”列是与对象类型相关的对象体的名字。这些 C 结构有的可以在 DDK 中找到，其他的可在 w2k_def.h 中找到。如果没有名字，那么意味着这个结构目前还是未知的。
- “符号”列给出了指向类型对象的指针的名字。如果对应的“是否公开”一列是“Yes”，则表示该指针变量是导出的，可以被内核模式驱动程序或应用程序访问（通过第六章给出的 w2k_call.dll）。

“索引”列还需要进一步的解释。这里给出的索引值均来自对应的 OBJECT_TYPE 结构的 ObjectTypeIndex 成员。这里的索引值并不是预定义常量。这和 Dispatcher 和 I/O 对象使用的 DISP_TYPE_*和 IO_TYPE_*常量不同（参见**表 7-1** 和**表 7-2**）。这里的索引值仅仅反映了系统创建这些类型对象的顺序。因此，你绝不应该使用 ObjectTypeIndex 来标识一个对象的类型。如果需要，则应该使用 ObjectTypeTag，在以后 OS 中，ObjectTypeTag 有很的稳定性。

7.1.9、对象句柄

尽管内核模式驱动程序可通过查询指向对象体的指针来联系该对象，但用户模式的程序却无法做到这一点。对于用户模式下的程序，当它调用相应的 API 打开一个对象时，它收到

的是该对象的一个句柄，在随后的操作中只能使用该句柄来访问对象。尽管 Windows 2000 在很多地方都使用“句柄”这一术语，句柄的本质是一个进程相关的 16 位数字（它通常是四的倍数），用来作为索引访问进程的句柄表，内核为每个进程维护一个独立的句柄表。**列表 7-10** 中给出了 HANDLE_TABLE 结构。该表指向一个 HANDLE_LAYER1 结构，HANDLE_LAYER1 结构由一个指向 HANDLE_LAYER2 结构的指针构成，而 HANDLE_LAYER2 结构又包含一个指向 HANDLE_LAYER3 结构。最后，在 HANDLE_LAYER3 结构中才包含指向实际的句柄表项的指针。句柄表项实际上就是一个 HANDLE_ENTRY 结构。

```
// HANDLE BIT-FIELDS

// -----------------

//  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

//  F E D C B A 9 8 7 6 5 4 3 2 1 0 F E D C B A 9 8 7 6 5 4 3 2 1 0

// ________________________________________________________________

// |x|x|x|x|x|a|a|a|a|a|a|a|a|b|b|b|b|b|b|b|b|c|c|c|c|c|c|c|c|y|y|

// | not used  | HANDLE_LAYER1 | HANDLE_LAYER2 | HANDLE_LAYER3 |tag|

#define HANDLE_LAYER_SIZE 0x00000100

// ----------------------------------------------------------------

#define HANDLE_ATTRIBUTE_INHERIT 0x00000002

#define HANDLE_ATTRIBUTE_MASK    0x00000007

#define HANDLE_OBJECT_MASK       0xFFFFFFF8

typedef struct _HANDLE_ENTRY // cf. OBJECT_HANDLE_INFORMATION

        {

/*000*/ union

            {
```

```
/*000*/     DWORD          HandleAttributes;// HANDLE_ATTRIBUTE_MASK

/*000*/     POBJECT_HEADER ObjectHeader;    // HANDLE_OBJECT_MASK

/*004*/     };

/*004*/ union

            {

/*004*/     ACCESS_MASK    GrantedAccess;   // if used entry

/*004*/     DWORD          NextEntry;       // if free entry

/*008*/     };

/*008*/ }

        HANDLE_ENTRY,

     * PHANDLE_ENTRY,

    **PPHANDLE_ENTRY;

#define HANDLE_ENTRY_ \

        sizeof (HANDLE_ENTRY)

// ------------------------------------------------------------

typedef struct _HANDLE_LAYER3

        {

/*000*/ HANDLE_ENTRY Entries [HANDLE_LAYER_SIZE]; // bits 2 to 9

/*800*/ }

        HANDLE_LAYER3,
```

```
     * PHANDLE_LAYER3,

    **PPHANDLE_LAYER3;

#define HANDLE_LAYER3_ \

        sizeof (HANDLE_LAYER3)

// ------------------------------------------------------------

typedef struct _HANDLE_LAYER2

        {

/*000*/ PHANDLE_LAYER3 Layer3 [HANDLE_LAYER_SIZE]; // bits 10 to 17

/*400*/ }

        HANDLE_LAYER2,

     * PHANDLE_LAYER2,

    **PPHANDLE_LAYER2;

#define HANDLE_LAYER2_ \

        sizeof (HANDLE_LAYER2)

// ------------------------------------------------------------

typedef struct _HANDLE_LAYER1

        {

/*000*/ PHANDLE_LAYER2 Layer2 [HANDLE_LAYER_SIZE]; // bits 18 to 25

/*400*/ }

        HANDLE_LAYER1,
```

```
     * PHANDLE_LAYER1,

    **PPHANDLE_LAYER1;

#define HANDLE_LAYER1_ \

        sizeof (HANDLE_LAYER1)

// -------------------------------------------------------------

typedef struct _HANDLE_TABLE

        {

/*000*/ DWORD             Reserved;

/*004*/ DWORD             HandleCount;

/*008*/ PHANDLE_LAYER1    Layer1;

/*00C*/ struct _EPROCESS *Process; // passed to PsChargePoolQuota ()

/*010*/ HANDLE            UniqueProcessId;

/*014*/ DWORD             NextEntry;

/*018*/ DWORD             TotalEntries;

/*01C*/ ERESOURCE         HandleTableLock;

/*054*/ LIST_ENTRY        HandleTableList;

/*05C*/ KEVENT            Event;

/*06C*/ }

        HANDLE_TABLE,

     * PHANDLE_TABLE,
```

```
    **PPHANDLE_TABLE;

#define HANDLE_TABLE_ \

        sizeof (HANDLE_TABLE)
```

列表 7-10.　句柄表、句柄表层和表项

这种三层寻址机制是一种非常巧妙的设计，它允许动态增加或减少句柄表项所需的存储空间，并且仅需很少的操作，而且其内存使用率也非常高。因为每个句柄表层最多可容纳 256 个指针，所以一个进程理论上可打开 256*256*256（即 16,777,216）个句柄。其中每个句柄表项占用 8 个字节，因此其最多占用 128MB 存储空间。不过，由于一个进程很少需要如此多的句柄，因此进程如果一开始就分配整个句柄表，那么将造成存储空间的浪费。Windows 2000 使用的三层结构在开始时仅为每一层分配一个最小空间。如果不将 HANDLE_TABLE 自身计算在内的话，那么在开始时仅需 256*4+256*4+256+89（即 4,096）字节。可以看出最初的句柄结构正好可以放入一个物理内存页中（在 32 位平台上，一个物理页占用 4KB 字节）。

为了查找句柄的 HANDLE_ENTRY，系统将句柄的 32 位值划分为 3 个 8 位分段， 0 位、1 位和高 6 位不使用。使用这三个 8 位分段，句柄解析机制以如下方式进行：

1. 句柄的 18 到 25 位作为 HANDLE_LAYER1 结构中的 Layer2 数组的索引，HANDLE_LAYER1 结构由 HANDLE_TABLE 结构中的 Layer1 引用。
2. 句柄的 10 到 17 位作为第一步中所选择的 HANDLE_LAYER2 结构中的 Layer3 数组的索引。
3. 句柄的 2 到 9 位作为第二步中所选择的 HANDLE_LAYER3 结构中的 Entries 数组的索引。
4. 从上一步的 Entries 数组中取出的 HANDLE_ENTRY 结构包含一个指向 OBJECT_HANDER（该对象头就是与给定 Handle 相关的对象头）的指针。

这听起来很容易让人糊涂，**图 7-2** 或许可以清楚地表示出到底发生了什么，图 7-2 和第四章的**图 4-3** 非常类似，**图 4-3** 描述了 i386 CPU 的线性地址与物理地址之间的转换机制。这两个算法都是将输入值分割为 3 个分段，其中的两个用于选择层次结构中的两个间接层，最后一个分段用于从目标层中选出一个具体表项。注意：层次化的句柄表模型是 Windows 2000 引入的，Windows NT 4.0 仅提供了单层的句柄表。

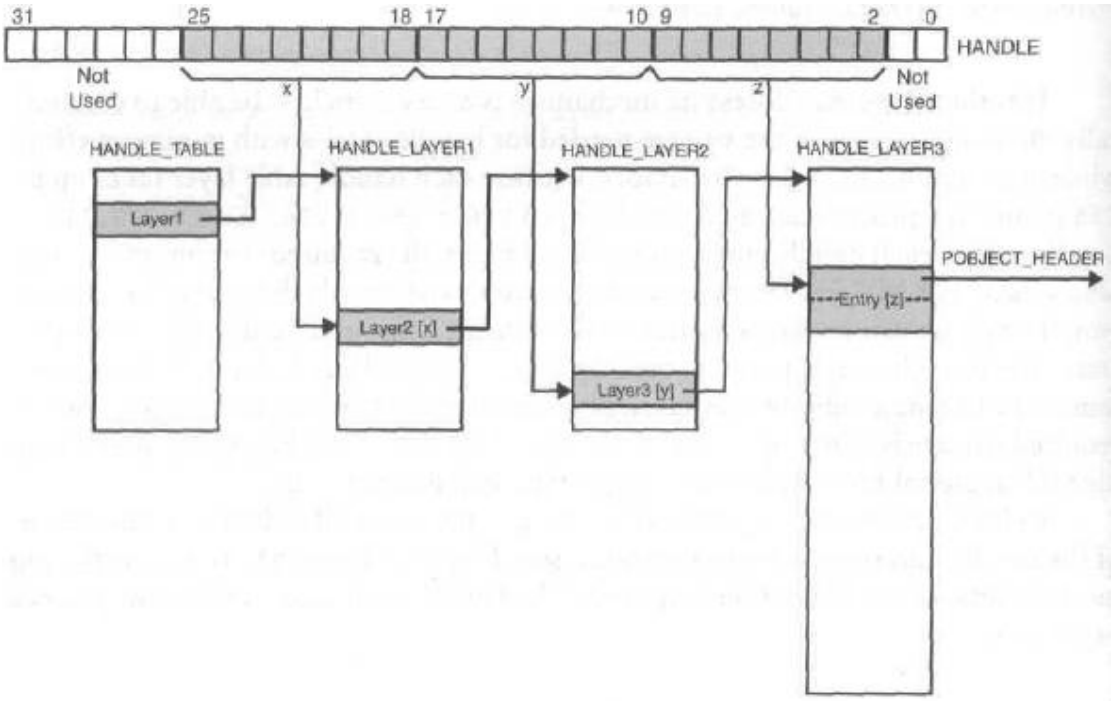


图 7-2.　从 HANDLE 到 OBJECT_HANDER 的解析过程

```
typedef struct  _LIST_ENTRY

{

    struct  _LIST_ENTRY  *Flink;

    struct  _LIST_ENTRY  *Blink;

} LIST_ENTRY, *PLIST_ENTRY;
```

补充：LIST_ENTRY 结构

由于每个进程都拥有自己的句柄表，故内核必须以某种方式跟踪当前已分配的句柄表。因此，ntoskrnl.exe 维护一个名为 HandleTableListHead 的 LIST_ENTRY 类型的变量，该变量指向由 HANDLE_TABLE 结构组成的双向链表的表头，内核使用 HANDLE_TABLE 结构中的 HandleTableList 将它们串联在一起。如果你使用 Flink 或 Blink 指针来访问 HANDLE_TABLE 时，你必须从指针地址中减去 HandleTableList 成员的偏移量 0x54 才能获取相应的 HANDLE_TABLE 的基地址。通过 UniqueProcessId 可很容易的确定拥有该句柄表的进程。通常情况下，链表中的第一个 HANDLE_TABLE 的所有者是系统进程（ID=8），其后的句柄表则

是属于系统空闲进程（ID=0）的。系统空闲进程的句柄表可通过内部变量 ObpKernelHandleTable 来访问。

在访问句柄表时，系统将使用一对同步对象来保证数据的完整性（尤其是在多线程环境下）。全局的 HandleTableListLock（由 ntoskrnl.exe 导出）将锁住整个句柄表链表，这里的 HandleTableListLock 实际上是一个 ERESOURCE 结构。这种类型的同步对象可作为一个排他锁也可用作一个共享锁。使用 ExAcquireResourceExclusiveLite()函数将得到一个排他锁，而使用 ExAcquireResourceSharedLite()函数得到的将是一个共享锁。对于这两种锁均可使用 ExReleaseResourceLite()函数来释放获得的锁。在使用排他锁锁住句柄表链表后，在你释放锁之前，系统不会更改此链表的任何内容。链表中的每个 HANDLE_TABLE 可以拥有自己的 ERESOURCE 锁，HANDLE_TABLE 结构中的 HandleTableLock（参见**列表 7-10**）提供了该锁。可使用 ntoskrnl.exe 提供的 ExLockHandleTableExclusive()和 ExLockHandleTableShared()来获取这个 ERESOURCE 锁，ExUnlockHandleTableShared()则用于释放这两种 ERESOURCE 锁。这几个函数只是 ExAcquireResourceExclusiveLite()、ExAcquireResourceSharedLite()和 ExReleaseResourceLite()函数的外包函数。

很不幸，所有这些由内核句柄管理器使用的基本函数和全局变量都没有文档化，而且无法访问，因为 ntoskrnl.exe 并没有导出它们。尽管根据对象句柄，可使用第六章提供的内核调用接口来找到它们对应的对象，但我不推荐这样做。原因之一是：这样的代码将无法在 Windows NT 4.0 下运行，因为 Windows NT 4.0 的句柄表与 Windows 2000 并不相同。另一个原因是：内核提供了一个“豪华”的函数来返回当前活动进程的句柄表的所有内容。这个函数就是 NtQuerySystemInformation()，用来获取句柄信息的类别是：SystemHandleInformation(16)。请参考 Schreiber(1999)或 Nebbett(2000)的文章来了解如何调用此函数。SystemHandleInformation 数据来自内部函数 ExpGetHandleInformation()，这一函数又依赖于 ObGetHandleInformation()函数。ObGetHandleInformation()函数将循环调用 ExSnapShotHandleTables()函数，该函数将完成实际的枚举句柄表链表的工作。ExSnapShotHandleTables()函数需要一个 Callback 函数指针，它会针对对象涉及到的每个 HANDLE_ENTRY 调用该函数。ObGetHandleInformation()函数使用内部的 Callback 函数 ObpCaptureHandleInformation()来填充调用者提供的缓冲区。这里用来填充的数据是一个结构体数组，该数组包含当前系统维护的每个句柄的相关信息。

7.1.10、进程和线程对象

或许最吸引人同时也是最复杂的 Windows 2000 对象就是进程、线程对象了。这些对象通常是软件开发人员必须处理的顶层对象。一个内核模式组件总是运行在某个线程的上下文

中，而一个线程也总是属于某个进程。因此，从本质上来看，进城和线程对象都是类型对象，在调试过程中它们被研究的频率最高。Windows 2000 内核调试器解决了这一问题，它提供了两个命令：!processfields 和!threadfields（译注，最新版的内核调试器已不在支持这两个命令，对应的新命令为：dt nt!_EPROCESS 和 dt nt!_ETHREAD），这两个命令由 kdextx86.dll 导出。这两个命令可针对 EPROCESS 和 ETHREAD 结构，分别输出由名称/偏移量构成的简单列表（参考第一章的**示例 1-1** 和**示例 1-2**）。这两个结构体都没有文档化，这两个命令是目前唯一的官方信息来源。

译注：

在 Windows XP Professional + SP2 下，HANDLE_TABLE 结构发生了很大的变化，如下所示：

```
kd> dt nt!_HANDLE_TABLE

   +0x000 TableCode        : Uint4B

   +0x004 QuotaProcess     : Ptr32 _EPROCESS

   +0x008 UniqueProcessId  : Ptr32 Void

   +0x00c HandleTableLock  : [4] _EX_PUSH_LOCK

   +0x01c HandleTableList  : _LIST_ENTRY

   +0x024 HandleContentionEvent : _EX_PUSH_LOCK

   +0x028 DebugInfo        : Ptr32 _HANDLE_TRACE_DEBUG_INFO

   +0x02c ExtraInfoPages   : Int4B

   +0x030 FirstFree        : Uint4B

   +0x034 LastFree         : Uint4B

   +0x038 NextHandleNeedingPool : Uint4B

   +0x03c HandleCount      : Int4B
```

```
   +0x040 Flags            : Uint4B

   +0x040 StrictFIFO       : Pos 0, 1 Bit
```

可以预见本章所给的 w2k_obj.exe 在 Windows XP 下将无法工作。

很不幸，!processfields 命令的输出（参见第一章中的**示例 1-1**）中最开始的是一个名为 Pcb 的成员，该成员指向一个 0x6C 字节大小的子结构（这是因为下一个成员 ExitStatus 的偏移量为 0x6c）。Pcb 是一个 KPROCESS 结构，该结构没有任何文档记载。这种排列非常有趣：显然，一个进程可由嵌入在庞大的可执行对象中的一个很小的内核对象来描述。这种嵌套式的结构也出现在线程对象中。从调试器的!threadfields 命令（参见第一章中的**示例 1-2**）的输出可以看到在 ETHREAD 结构的开始位置存在一个大小为 0x1B0 字节的 Tcb 成员。这就是 KTHREAD 结构，它代表可执行对象中的另一个内核对象。

译注：

本书中所给的 EPROCESS 和 ETHREAD 结构的定义，在 Windows XP+SP2 中已发生了很大变化。

尽管内核调试器提供的有关进程和线程对象的符号化信息非常有帮助，但没有提供足够的信息来说明这些结构体成员的类型。而且，Pcb 和 Tcb 成员的不透明性进一步加大了掌握这些对象的本质的难度。在内核调试器生成的反汇编列表中，你会发现有很多指令涉及到这些不透明的数据结构。它们使用的偏移量对于我们来说毫无价值，它无法提供任何有关这些数据的名字和类型的信息。因此，我收集了各方面的资料再加上我自己的研究结果，以推断出这些对象的具体类型。列表 7-11 和列表 7-12 给出部分研究结果，这两个列表分别给出了 KPROCESS 和 KTHREAD 结构的定义。在这两个结构体开始处的 DISPATCHER_HEADER 结构明确表示了这两个结构体都属于 Dispatcher 对象。这意味着可使用 KeWaitForSingleObject()和 KeWaitForMultipleObjects()来等待它们。当线程结束执行时，线程对象才会变为“有信号”状态，一个进程仅当其所有线程都终止时才会进入“有信号”状态。这对于 Win32 程序员来说没什么新鲜的。不过，现在你终于明白了为什么等待一个进程和线程对象是可行的了。

```
typedef struct _KPROCESS

        {
```

```
/*000*/ DISPATCHER_HEADER Header; // DO_TYPE_PROCESS (0x1B)

/*010*/ LIST_ENTRY        ProfileListHead;

/*018*/ DWORD             DirectoryTableBase;

/*01C*/ DWORD             PageTableBase;

/*020*/ KGDTENTRY         LdtDescriptor;

/*028*/ KIDTENTRY         Int21Descriptor;

/*030*/ WORD              IopmOffset;

/*032*/ BYTE              Iopl;

/*033*/ BOOLEAN           VdmFlag;

/*034*/ DWORD             ActiveProcessors;

/*038*/ DWORD             KernelTime; // ticks

/*03C*/ DWORD             UserTime;   // ticks

/*040*/ LIST_ENTRY        ReadyListHead;

/*048*/ LIST_ENTRY        SwapListEntry;

/*050*/ LIST_ENTRY        ThreadListHead; // KTHREAD.ThreadListEntry

/*058*/ PVOID             ProcessLock;

/*05C*/ KAFFINITY         Affinity;

/*060*/ WORD              StackCount;

/*062*/ BYTE              BasePriority;

/*063*/ BYTE              ThreadQuantum;
```

```
/*064*/ BOOLEAN           AutoAlignment;

/*065*/ BYTE              State;

/*066*/ BYTE              ThreadSeed;

/*067*/ BOOLEAN           DisableBoost;

/*068*/ DWORD             d068;

/*06C*/ }

        KPROCESS,

     * PKPROCESS,

    **PPKPROCESS;

#define KPROCESS_ \

        sizeof (KPROCESS)
```

列表 7-11. KPROCESS 结构

```
typedef struct _KTHREAD

        {

/*000*/ DISPATCHER_HEADER         Header; // DO_TYPE_THREAD (0x6C)

/*010*/ LIST_ENTRY                MutantListHead;

/*018*/ PVOID                     InitialStack;

/*01C*/ PVOID                     StackLimit;

/*020*/ struct _TEB              *Teb;

/*024*/ PVOID                     TlsArray;
```

```
/*028*/ PVOID                   KernelStack;

/*02C*/ BOOLEAN                 DebugActive;

/*02D*/ BYTE                    State; // THREAD_STATE_*

/*02E*/ BOOLEAN                 Alerted;

/*02F*/ BYTE                    bReserved01;

/*030*/ BYTE                    Iopl;

/*031*/ BYTE                    NpxState;

/*032*/ BYTE                    Saturation;

/*033*/ BYTE                    Priority;

/*034*/ KAPC_STATE              ApcState;

/*04C*/ DWORD                   ContextSwitches;

/*050*/ DWORD                   WaitStatus;

/*054*/ BYTE                    WaitIrql;

/*055*/ BYTE                    WaitMode;

/*056*/ BYTE                    WaitNext;

/*057*/ BYTE                    WaitReason;

/*058*/ PLIST_ENTRY             WaitBlockList;

/*05C*/ LIST_ENTRY              WaitListEntry;

/*064*/ DWORD                   WaitTime;

/*068*/ BYTE                    BasePriority;
```

```
/*069*/ BYTE                        DecrementCount;

/*06A*/ BYTE                        PriorityDecrement;

/*06B*/ BYTE                        Quantum;

/*06C*/ KWAIT_BLOCK                 WaitBlock [4];

/*0CC*/ DWORD                       LegoData;

/*0D0*/ DWORD                       KernelApcDisable;

/*0D4*/ KAFFINITY                   UserAffinity;

/*0D8*/ BOOLEAN                     SystemAffinityActive;

/*0D9*/ BYTE                        Pad [3];

/*0DC*/ PSERVICE_DESCRIPTOR_TABLE pServiceDescriptorTable;

/*0E0*/ PVOID                       Queue;

/*0E4*/ PVOID                       ApcQueueLock;

/*0E8*/ KTIMER                      Timer;

/*110*/ LIST_ENTRY                  QueueListEntry;

/*118*/ KAFFINITY                   Affinity;

/*11C*/ BOOLEAN                     Preempted;

/*11D*/ BOOLEAN                     ProcessReadyQueue;

/*11E*/ BOOLEAN                     KernelStackResident;

/*11F*/ BYTE                        NextProcessor;

/*120*/ PVOID                       CallbackStack;
```

```
/*124*/ struct _WIN32_THREAD     *Win32Thread;

/*128*/ PVOID                     TrapFrame;

/*12C*/ PKAPC_STATE               ApcStatePointer;

/*130*/ PVOID                     p130;

/*134*/ BOOLEAN                   EnableStackSwap;

/*135*/ BOOLEAN                   LargeStack;

/*136*/ BYTE                      ResourceIndex;

/*137*/ KPROCESSOR_MODE           PreviousMode;

/*138*/ DWORD                     KernelTime; // ticks

/*13C*/ DWORD                     UserTime;   // ticks

/*140*/ KAPC_STATE                SavedApcState;

/*157*/ BYTE                      bReserved02;

/*158*/ BOOLEAN                   Alertable;

/*159*/ BYTE                      ApcStateIndex;

/*15A*/ BOOLEAN                   ApcQueueable;

/*15B*/ BOOLEAN                   AutoAlignment;

/*15C*/ PVOID                     StackBase;

/*160*/ KAPC                      SuspendApc;

/*190*/ KSEMAPHORE                SuspendSemaphore;

/*1A4*/ LIST_ENTRY                ThreadListEntry;  // see KPROCESS
```

```
/*1AC*/ BYTE                    FreezeCount;

/*1AD*/ BYTE                    SuspendCount;

/*1AE*/ BYTE                    IdealProcessor;

/*1AF*/ BOOLEAN                 DisableBoost;

/*1B0*/ }

        KTHREAD,

     * PKTHREAD,

    **PPKTHREAD;

#define KTHREAD_ \

        sizeof (KTHREAD)
```

列表 7-12. KTHREAD 结构

KPROCESS 结构通过自己的 ThreadListHead 成员来组织它下属的线程对象，对于由 KTHREAD 对象构成的双向链表来说，ThreadListHead 既是该链表的起点也是该链表的终点（呵呵，又是一个环形链表）。每个线程也有一个类似的结构来保存各自的对象，那就是它们的 ThreadListEntry 成员，线程自己的链表是由一个个 LIST_ENTRY 结构组成。要得到这一个个 LIST_ENTRY 所在的对象的基地址，则必须从 LIST_ENTRY 结构的地址值中减去其在所属结构中的偏移量才能得到，这是因为 LIST_ENTRY 结构的 Flink 和 Blink 成员总是指向链表中的下一个 LIST_ENTRY 结构，而不是拥有该节点结构的对象。这种特性可以很容易的将同一个对象加入到多个链表中而不会产生冲突。

在**列表 7-11**、**列表 7-12** 以及下面即将给出的列表中，你会发现一个特殊的成员，该成员的名字由一个小写字符和三个十六进制数组成。这些成员的确切含义我目前还不清楚。开始的第一个字符反映出了该成员可能的类型（比如，d 表示 DWORD，p 表示 PVOID），随后的数字表示该成员相对于结构基地址的偏移量。

列表 7-13 和**列表 7-14** 分别给出了 EPROCESS 和 ETHREAD 可执行对象。这些结构包含一些还未确定的成员，希望在本书的鼓励下，有人能把它们确定下来。不过，最重要和最常用的成员都已确定下来，至少我们知道丢失了那些信息。

```
typedef struct _EPROCESS

        {

/*000*/ KPROCESS               Pcb;

/*06C*/ NTSTATUS               ExitStatus;

/*070*/ KEVENT                 LockEvent;

/*080*/ DWORD                  LockCount;

/*084*/ DWORD                  d084;

/*088*/ LARGE_INTEGER          CreateTime;

/*090*/ LARGE_INTEGER          ExitTime;

/*098*/ PVOID                  LockOwner;

/*09C*/ DWORD                  UniqueProcessId;

/*0A0*/ LIST_ENTRY             ActiveProcessLinks;

/*0A8*/ DWORD                  QuotaPeakPoolUsage [2]; // NP, P

/*0B0*/ DWORD                  QuotaPoolUsage     [2]; // NP, P

/*0B8*/ DWORD                  PagefileUsage;

/*0BC*/ DWORD                  CommitCharge;

/*0C0*/ DWORD                  PeakPagefileUsage;

/*0C4*/ DWORD                  PeakVirtualSize;
```

```
/*0C8*/ LARGE_INTEGER          VirtualSize;

/*0D0*/ MMSUPPORT              Vm;

/*100*/ DWORD                  d100;

/*104*/ DWORD                  d104;

/*108*/ DWORD                  d108;

/*10C*/ DWORD                  d10C;

/*110*/ DWORD                  d110;

/*114*/ DWORD                  d114;

/*118*/ DWORD                  d118;

/*11C*/ DWORD                  d11C;

/*120*/ PVOID                  DebugPort;

/*124*/ PVOID                  ExceptionPort;

/*128*/ PHANDLE_TABLE          ObjectTable;

/*12C*/ PVOID                  Token;

/*130*/ FAST_MUTEX             WorkingSetLock;

/*150*/ DWORD                  WorkingSetPage;

/*154*/ BOOLEAN                ProcessOutswapEnabled;

/*155*/ BOOLEAN                ProcessOutswapped;

/*156*/ BOOLEAN                AddressSpaceInitialized;

/*157*/ BOOLEAN                AddressSpaceDeleted;
```

```
/*158*/ FAST_MUTEX             AddressCreationLock;

/*178*/ KSPIN_LOCK             HyperSpaceLock;

/*17C*/ DWORD                  ForkInProgress;

/*180*/ WORD                   VmOperation;

/*182*/ BOOLEAN                ForkWasSuccessful;

/*183*/ BYTE                   MmAgressiveWsTrimMask;

/*184*/ DWORD                  VmOperationEvent;

/*188*/ HARDWARE_PTE           PageDirectoryPte;

/*18C*/ DWORD                  LastFaultCount;

/*190*/ DWORD                  ModifiedPageCount;

/*194*/ PVOID                  VadRoot;

/*198*/ PVOID                  VadHint;

/*19C*/ PVOID                  CloneRoot;

/*1A0*/ DWORD                  NumberOfPrivatePages;

/*1A4*/ DWORD                  NumberOfLockedPages;

/*1A8*/ WORD                   NextPageColor;

/*1AA*/ BOOLEAN                ExitProcessCalled;

/*1AB*/ BOOLEAN                CreateProcessReported;

/*1AC*/ HANDLE                 SectionHandle;

/*1B0*/ struct _PEB           *Peb;
```

```
/*1B4*/ PVOID                   SectionBaseAddress;

/*1B8*/ PQUOTA_BLOCK            QuotaBlock;

/*1BC*/ NTSTATUS                LastThreadExitStatus;

/*1C0*/ DWORD                   WorkingSetWatch;

/*1C4*/ HANDLE                  Win32WindowStation;

/*1C8*/ DWORD                   InheritedFromUniqueProcessId;

/*1CC*/ ACCESS_MASK             GrantedAccess;

/*1D0*/ DWORD                   DefaultHardErrorProcessing; // HEM_*

/*1D4*/ DWORD                   LdtInformation;

/*1D8*/ PVOID                   VadFreeHint;

/*1DC*/ DWORD                   VdmObjects;

/*1E0*/ PVOID                   DeviceMap; // 0x24 bytes

/*1E4*/ DWORD                   SessionId;

/*1E8*/ DWORD                   d1E8;

/*1EC*/ DWORD                   d1EC;

/*1F0*/ DWORD                   d1F0;

/*1F4*/ DWORD                   d1F4;

/*1F8*/ DWORD                   d1F8;

/*1FC*/ BYTE                    ImageFileName [16];

/*20C*/ DWORD                   VmTrimFaultValue;
```

```
/*210*/ BYTE                    SetTimerResolution;

/*211*/ BYTE                    PriorityClass;

/*212*/ union

            {

            struct

                {

/*212*/         BYTE                SubSystemMinorVersion;

/*213*/         BYTE                SubSystemMajorVersion;

                };

            struct

                {

/*212*/         WORD                SubSystemVersion;

                };

            };

/*214*/ struct _WIN32_PROCESS *Win32Process;

/*218*/ DWORD                   d218;

/*21C*/ DWORD                   d21C;

/*220*/ DWORD                   d220;

/*224*/ DWORD                   d224;

/*228*/ DWORD                   d228;
```

```
/*22C*/ DWORD                   d22C;

/*230*/ PVOID                   Wow64;

/*234*/ DWORD                   d234;

/*238*/ IO_COUNTERS             IoCounters;

/*268*/ DWORD                   d268;

/*26C*/ DWORD                   d26C;

/*270*/ DWORD                   d270;

/*274*/ DWORD                   d274;

/*278*/ DWORD                   d278;

/*27C*/ DWORD                   d27C;

/*280*/ DWORD                   d280;

/*284*/ DWORD                   d284;

/*288*/ }

        EPROCESS,

     * PEPROCESS,

    **PPEPROCESS;

#define EPROCESS_ \

        sizeof (EPROCESS)
```

列表 7-13. EPROCESS 结构

```
typedef struct _ETHREAD

        {

/*000*/ KTHREAD       Tcb;

/*1B0*/ LARGE_INTEGER CreateTime;

/*1B8*/ union

            {

/*1B8*/     LARGE_INTEGER ExitTime;

/*1B8*/     LIST_ENTRY    LpcReplyChain;

            };

/*1C0*/ union

            {

/*1C0*/     NTSTATUS      ExitStatus;

/*1C0*/     DWORD         OfsChain;

            };

/*1C4*/ LIST_ENTRY    PostBlockList;

/*1CC*/ LIST_ENTRY    TerminationPortList;

/*1D4*/ PVOID         ActiveTimerListLock;

/*1D8*/ LIST_ENTRY    ActiveTimerListHead;

/*1E0*/ CLIENT_ID     Cid;
```

```
/*1E8*/ KSEMAPHORE    LpcReplySemaphore;

/*1FC*/ DWORD         LpcReplyMessage;

/*200*/ DWORD         LpcReplyMessageId;

/*204*/ DWORD         PerformanceCountLow;

/*208*/ DWORD         ImpersonationInfo;

/*20C*/ LIST_ENTRY    IrpList;

/*214*/ PVOID         TopLevelIrp;

/*218*/ PVOID         DeviceToVerify;

/*21C*/ DWORD         ReadClusterSize;

/*220*/ BOOLEAN       ForwardClusterOnly;

/*221*/ BOOLEAN       DisablePageFaultClustering;

/*222*/ BOOLEAN       DeadThread;

/*223*/ BOOLEAN       Reserved;

/*224*/ BOOL          HasTerminated;

/*228*/ ACCESS_MASK   GrantedAccess;

/*22C*/ PEPROCESS     ThreadsProcess;

/*230*/ PVOID         StartAddress;

/*234*/ union

            {

/*234*/     PVOID         Win32StartAddress;
```

```
/*234*/     DWORD         LpcReceivedMessageId;

            };

/*238*/ BOOLEAN       LpcExitThreadCalled;

/*239*/ BOOLEAN       HardErrorsAreDisabled;

/*23A*/ BOOLEAN       LpcReceivedMsgIdValid;

/*23B*/ BOOLEAN       ActiveImpersonationInfo;

/*23C*/ DWORD         PerformanceCountHigh;

/*240*/ DWORD         d240;

/*244*/ DWORD         d244;

/*248*/ }

        ETHREAD,

     * PETHREAD,

    **PPETHREAD;

#define ETHREAD_ \

        sizeof (ETHREAD)
```

列表 7-14. ETHREAD 结构

除!processfields 和!threadfields 命令列出的成员之外，在 EPROCESS 和 ETHREAD 结构中实际上还包含一些附加成员。有两种主要的方式可发现有关未文档化成员的详细信息。其一是：观察系统函数是如何访问这些对象成员的；其二是：检查对象是如何被创建并被初始化的。第二种方法可以获得对象的实际大小。基本的对象创建函数是 ntoskrnl.exe 导出的 ObCreateObject()函数，该函数为对象表头和对象体分配内存，并初始化常见的对象参数。不过，ObCreateObject()对它创建的对象是什么类型却丝毫不知，因此，调用者必须给定对象体所需内存的确切字节数。因此，找出对象实际大小这一问题就转化为针对此类对象

所调用的 ObCreateObject()。进程对象是由 Native API 函数 NtCreateProcess()创建的，而 NtCreateProcess()又调用 PspCreateProcess()来完成实际工作。在 PspCreateProcess()函数中，可找到调用 ObCreateObject()的地址，在这里你会发现所需的对象体的大小为 0x288 字节（即 648 字节）。这就是为什么在**列表 7-13** 中会包含几个没有确切名称的成员的原因，就是为了让对象大小达到 0x288。ETHREAD 结构也是如此：NtCreateThread()函数调用 PspCreateThread()，后者在转而调用 ObCreateObject()以获取大小为 0x248 字节的对象。

当前正在运行的进程使用 EPROCESS 结构中的 ActiveProcessLinks 成员互相链接起来构成了一个链表。该链表的表头保存在全局变量 PsActiveProcessHead 中，与该链表相关的同步对象是类型为 FAST_MUTEX 的 PspActiveProcessMutex。很不幸，ntoskrnl.exe 并未导出 PsActiveProcessHead 变量，但它导出了一个名为 PsInitialSystemProcess 的变量，该变量实际上是一个指向进程 ID 为 8 的系统进程的 EPROCESS 结构。通过该结构的 ActiveProcessLinks 成员的 Blink 指针就可找到 PsActiveProcessHead。**图 7-3** 给出了进程和线程链接构成的结构图。**图 7-3** 只是一个简化图，它给出的进程链表中仅包含两项。在实际情况下，这个链表会非常的长。为了使该图简洁明了，我仅给出了一个进程的线程链表，并假设该进程只有两个线程。

从**列表 7-12** 和**列表 7-13** 可看出，在内核和执行体之上还存在一个进程和线程对象，它们分别指向位于 EPROCESS 和 KTHREAD 中的 WIN32_PROCESS 和 WIN32_THREAD 结构。这些未文档化的结构构成了 Win32 子系统中的进程和线程对象。尽管这些结构体的某些成员的含义非常明显，但它们仍包含很多用意不明的成员。这或许是将来要探索的领域了。

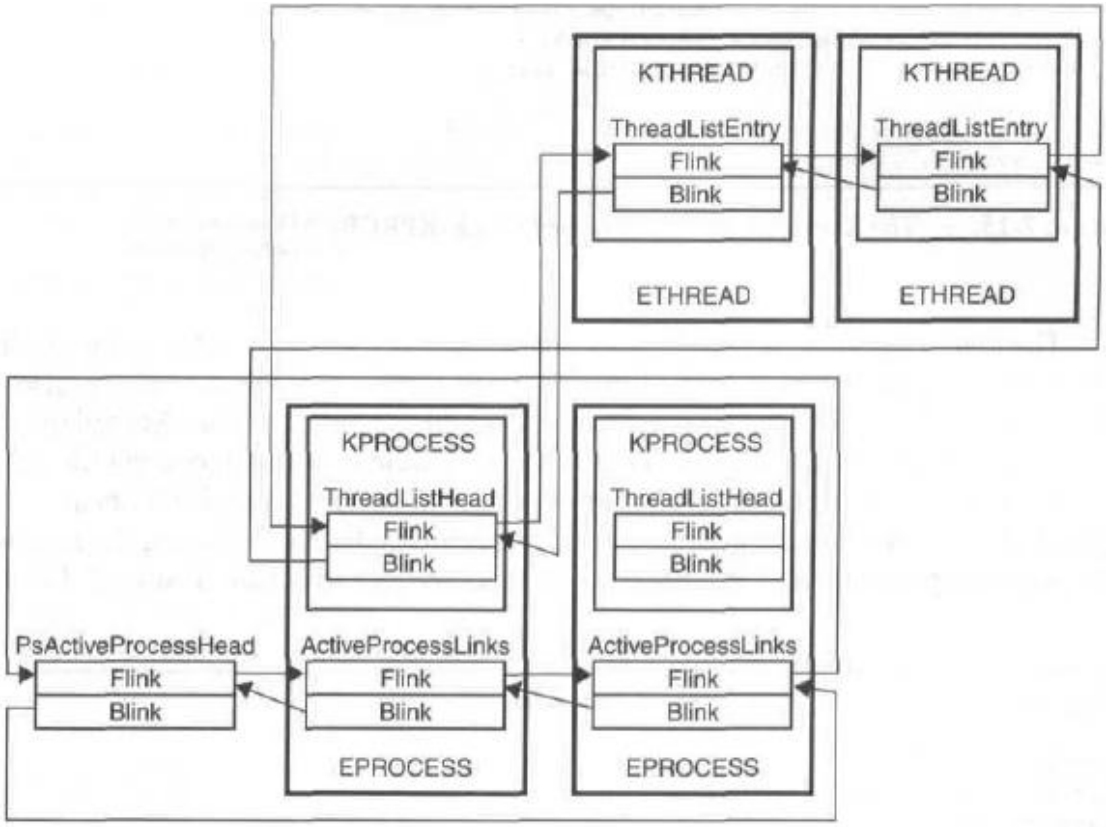


图 7-3.　进程和线程对象链表

7.1.11、线程和进程的上下文（Context）

当系统执行代码时，执行将总是在该进程的某个线程的上下文中进行。在很多情况下，系统必须从当前上下文中寻找与线程或进程相关的信息。因此，系统总是将当前线程的指针保存在一个内核处理器控制块（Kernel’s Processor Control Block，KPRCB）中。该结构定义于 ntddk.h 中，列表 7-15 给出了该结构。

```
typedef struct _KPRCB // processor control block

        {

/*000*/ WORD                    MinorVersion;

/*002*/ WORD                    MajorVersion;

/*004*/ struct _KTHREAD        *CurrentThread;

/*008*/ struct _KTHREAD        *NextThread;
```

```
/*00C*/ struct _KTHREAD       *IdleThread;

/*010*/ CHAR                   Number;

/*011*/ CHAR                   Reserved;

/*012*/ WORD                   BuildType;

/*014*/ KAFFINITY              SetMember;

/*018*/ struct _RESTART_BLOCK *RestartBlock;

/*01C*/ }

        KPRCB,

     * PKPRCB,

    **PPKPRCB;

#define KPRCB_ \

        sizeof (KPRCB)
```

列表 7-15. 内核处理器控制块（KPRCB）

在线性地址 0xFFDFF120 处可找到 KPRCB 结构，指向该结构的指针存放在 KPCR 结构（Kernel’s Processor Control Region）的 Prcb 成员中。KPCR 结构的定义可在 Ntddk.h 中找到，该结构位于线性地址 0xFFDFF000 处。就像在第四章解释的那样，内核模块通过 FS 寄存器可以很容易的访问该结构体。从地址：FS:0 处读取等价于从线性地址 DS:0xFFDFF000 处读取。系统将最基本的 CPU 信息保存在地址 0xFFDFF13C 处（紧随 KPRCB 结构之后）的 CONTEXT 结构（见**列表 7-17**）中。

```
typedef struct _KPCR // processor control region

        {

/*000*/ NT_TIB             NtTib;
```

```
/*01C*/ struct _KPCR      *SelfPcr;

/*020*/ PKPRCB             Prcb;

/*024*/ KIRQL              Irql;

/*028*/ DWORD              IRR;

/*02C*/ DWORD              IrrActive;

/*030*/ DWORD              IDR;

/*034*/ DWORD              Reserved2;

/*038*/ struct _KIDTENTRY *IDT;

/*03C*/ struct _KGDTENTRY *GDT;

/*040*/ struct _KTSS      *TSS;

/*044*/ WORD               MajorVersion;

/*046*/ WORD               MinorVersion;

/*048*/ KAFFINITY          SetMember;

/*04C*/ DWORD              StallScaleFactor;

/*050*/ BYTE               DebugActive;

/*051*/ BYTE               Number;

/*054*/ }

        KPCR,

     * PKPCR,

    **PPKPCR;
```

```
#define KPCR_ \

        sizeof (KPCR)
```

列表 7-16. 内核处理器控制区域（KPCR）

```
#define SIZE_OF_80387_REGISTERS 80

typedef struct _FLOATING_SAVE_AREA

        {

/*000*/ DWORD ControlWord;

/*004*/ DWORD StatusWord;

/*008*/ DWORD TagWord;

/*00C*/ DWORD ErrorOffset;

/*010*/ DWORD ErrorSelector;

/*014*/ DWORD DataOffset;

/*018*/ DWORD DataSelector;

/*01C*/ BYTE  RegisterArea [SIZE_OF_80387_REGISTERS];

/*06C*/ DWORD Cr0NpxState;

/*070*/ }

        FLOATING_SAVE_AREA,

     * PFLOATING_SAVE_AREA,

    **PPFLOATING_SAVE_AREA;

#define FLOATING_SAVE_AREA_ \
```

```
        sizeof (FLOATING_SAVE_AREA)

#define MAXIMUM_SUPPORTED_EXTENSION 512

typedef struct _CONTEXT

        {

/*000*/ DWORD       ContextFlags;

/*004*/ DWORD       Dr0;

/*008*/ DWORD       Dr1;

/*00C*/ DWORD       Dr2;

/*010*/ DWORD       Dr3;

/*014*/ DWORD       Dr6;

/*018*/ DWORD       Dr7;

/*01C*/ FLOATING_SAVE_AREA FloatSave;

/*08C*/ DWORD       SegGs;

/*090*/ DWORD       SegFs;

/*094*/ DWORD       SegEs;

/*098*/ DWORD       SegDs;

/*09C*/ DWORD       Edi;

/*0A0*/ DWORD       Esi;

/*0A4*/ DWORD       Ebx;

/*0A8*/ DWORD       Edx;
```

```
/*0AC*/ DWORD       Ecx;

/*0B0*/ DWORD       Eax;

/*0B4*/ DWORD       Ebp;

/*0B8*/ DWORD       Eip;

/*0BC*/ DWORD       SegCs;

/*0C0*/ DWORD       EFlags;

/*0C4*/ DWORD       Esp;

/*0C8*/ DWORD       SegSs;

/*0CC*/ BYTE        ExtendedRegisters [MAXIMUM_SUPPORTED_EXTENSION];

/*2CC*/ }

        CONTEXT,

     * PCONTEXT,

    **PPCONTEXT;

#define CONTEXT_ \

        sizeof (CONTEXT)
```

列表 7-17　CPU 的 CONTEXT 和 FLOATING_SAVE_AREA 结构

对照**列表 7-15**，可看出 KPRCB 结构包含三个指向 KTHREAD 结构的指针，其偏移量分别为：0x004、0x008 和 0x00C：

1. CurrentThread 指向当前正在执行的线程的 KTHREAD 对象。内核代码经常访问该成员。
2. NextThread 指向在下一次上下文切换后将要运行的线程的 KTHREAD 对象。

3. IdleThread 指向空闲线程的 KTHREAD 对象，当没有线程准备好去执行时，该线程将执行后台任务。系统为每个已安装 CPU 提供了一个专用的空闲线程。在单处理器系统中，唯一的空闲线程对象被称作 POBootThread，并且它是 PsIdleProcess 对象的线程链表中仅有的一个线程。

由于 ETHREAD 结构中的第一个成员是 KTHREAD，而 KTHREAD 指针又总是指向 ETHREAD。所以 KTHREAD 和 ETHREAD 之间是可以相互进行类型转化的。KPROCESS 和 EPROCESS 也是如此。

由于 Windows 2000 内核将线性地址 0xFFDFF000 映射到了 CPU 的 FS 的内核模式段的 0x00000000,所以系统总是能在地址:FS:0x0、FS:0x120 和 FS:13C 处找到当前的 KPCR、KPRCB 和 CONTEXT 结构。当你在调试器中反编译内核代码时，你会发现系统经常从 FS:0x124 处取出一个指针，很明显，该指针指向的是当前的线程对象。**示例 7-1** 给出了内核调试器命令：u PsGetCurrentProcessId 的执行结果，该命令将使内核调试器从 PsGetCurrentProcessId 所处的地址开始反编译 10 行代码。可看出，PsGetCurrentProcessId()函数只是简单的取出当前线程的 KTHREAD/ETHREAD 结构，然后返回结构中偏移量为 0x1E0 的数值，此处恰是 CLIENT_ID 类型的 cid 成员（属于 ETHREAD 结构）的 UniqueProcessID（参见列表 7-14）。PsGetCurrentThreadId()与之类似，不同之处是它在偏移量 0x1E4 处取出 UniqueThreadID。在第二章的列表 2-8 中有 CLIENT_ID 结构的定义。

```
kd> u PsGetCurrentProcessId

ntoskrnl!PsGetCurrentProcessId:

8052ba52 64a124010000     mov     eax,fs:[00000124]

8052ba58 8b80e0010000     mov     eax,[eax+0x1e0]

8052ba5e c3               ret

8052ba5f cc               int     3

ntoskrnl!PsGetCurrentThreadId:

8052ba60 64a124010000     mov     eax,fs:[00000124]

8052ba66 8b80e4010000     mov     eax,[eax+0x1e4]

8052ba6c c3               ret
```

```
8052ba6d cc              int     3
```

示例 7-1. 获取进程和线程 ID

有时，系统需要当前线程所属进程对象的指针。可通过当前 KTHREAD 结构中的 ApcState 子结构的 Process 成员来获取该指针。

7.1.12、线程和进程环境块

你可能会很困惑:KTHREAD 和 EPROCESS 结构中的 Teb 和 Peb 成员有什么实际价值？Teb，指向一个线程环境块(TEB),见**列表 7-18**。TEB 的第一部分是线程信息块(Thread Information Block，NT_TIB），该结构在 DDK 的 ntddk.h 和 SDK 的 winnt.h 中均有定义，其他部分则没有相应的文档记载。Windows 2000 为系统中的每个线程维护一个 TEB 结构。在当前进程的地址空间中，其下属线程的 TEB 被映射到线性地址 0x7FFDE000、0x7FFDD000、0x7FFDC000 以此类推，每个线程均占用一个完整的 4KB 页。就像在第四章提到的那样，在用户模式下仍可通过 FS:0x18 来访问当前线程的 TEB，FS:0x18 处存放的是内嵌的 NT_TIB 的 Self 成员。该成员总是指向其所在 TEB 的线性地址（该线性地址位于当前线程的 4GB 地址空间中）。

```
// typedef struct _NT_TIB // see winnt.h / ntddk.h

//         {

// /*000*/ struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;

// /*004*/ PVOID                                    StackBase;

// /*008*/ PVOID                                    StackLimit;

// /*00C*/ PVOID                                    SubSystemTib;

// /*010*/ union

//             {

// /*010*/     PVOID FiberData;

// /*010*/     ULONG Version;
```

```
//             };

// /*014*/ PVOID           ArbitraryUserPointer;

// /*018*/ struct _NT_TIB *Self;

// /*01C*/ }

//         NT_TIB,

//      * PNT_TIB,

//     **PPNT_TIB;

// ------------------------------------------------------------

// located at 0x7FFDE000, 0x7FFDD000, ...

typedef struct _TEB

        {

/*000*/ NT_TIB    Tib;

/*01C*/ PVOID     EnvironmentPointer;

/*020*/ CLIENT_ID Cid;

/*028*/ HANDLE    RpcHandle;

/*02C*/ PPVOID    ThreadLocalStorage;

/*030*/ PPEB      Peb;

/*034*/ DWORD     LastErrorValue;

/*038*/ }

        TEB,
```

```
     * PTEB,

    **PPTEB;

#define TEB_ \

        sizeof (TEB)
```

列表 7-18.　线程环境块（TEB）

就像每个线程都用于各自的 TEB 一样，每个进程也有与之相关的 PEB（进程环境块）。PEB 结构要比 TEB 复杂得多，如列表 7-19 所示。PEB 结构包含多个指向各种子结构的指针，这些子结构又引用了更多的子结构，而且这些子结构大多都没有相应的文档。列表 7-19 中包含其中一些子结构的原始草图，很多成员的名字都是假设的。PEB 位于线性地址：0x7FFDF000 处，这意味着，TEB 堆栈之后的第一个 4KB 页存放的就是 PEB。这样系统通过当前线程的 TEB 结构中的 Peb 成员就可很容易的访问 PEB 结构。

```
typedef struct _MODULE_HEADER

        {

/*000*/ DWORD      d000;

/*004*/ DWORD      d004;

/*008*/ LIST_ENTRY List1;

/*010*/ LIST_ENTRY List2;

/*018*/ LIST_ENTRY List3;

/*020*/ }

        MODULE_HEADER,

     * PMODULE_HEADER,

    **PPMODULE_HEADER;
```

```
#define MODULE_HEADER_ \

        sizeof (MODULE_HEADER)

// ------------------------------------------------------------------

typedef struct _PROCESS_MODULE_INFO

        {

/*000*/ DWORD         Size; // 0x24

/*004*/ MODULE_HEADER ModuleHeader;

/*024*/ }

        PROCESS_MODULE_INFO,

     * PPROCESS_MODULE_INFO,

    **PPPROCESS_MODULE_INFO;

#define PROCESS_MODULE_INFO_ \

        sizeof (PROCESS_MODULE_INFO)

// ------------------------------------------------------------------

// see RtlCreateProcessParameters()

typedef struct _PROCESS_PARAMETERS

        {

/*000*/ DWORD          Allocated;

/*004*/ DWORD          Size;

/*008*/ DWORD          Flags; // bit 0: all pointers normalized
```

```
/*00C*/ DWORD           Reserved1;

/*010*/ LONG            Console;

/*014*/ DWORD           ProcessGroup;

/*018*/ HANDLE          StdInput;

/*01C*/ HANDLE          StdOutput;

/*020*/ HANDLE          StdError;

/*024*/ UNICODE_STRING WorkingDirectoryName;

/*02C*/ HANDLE          WorkingDirectoryHandle;

/*030*/ UNICODE_STRING SearchPath;

/*038*/ UNICODE_STRING ImagePath;

/*040*/ UNICODE_STRING CommandLine;

/*048*/ PWORD           Environment;

/*04C*/ DWORD           X;

/*050*/ DWORD           Y;

/*054*/ DWORD           XSize;

/*058*/ DWORD           YSize;

/*05C*/ DWORD           XCountChars;

/*060*/ DWORD           YCountChars;

/*064*/ DWORD           FillAttribute;

/*068*/ DWORD           Flags2;
```

```
/*06C*/ WORD           ShowWindow;

/*06E*/ WORD           Reserved2;

/*070*/ UNICODE_STRING Title;

/*078*/ UNICODE_STRING Desktop;

/*080*/ UNICODE_STRING Reserved3;

/*088*/ UNICODE_STRING Reserved4;

/*090*/ }

        PROCESS_PARAMETERS,

     * PPROCESS_PARAMETERS,

    **PPPROCESS_PARAMETERS;

#define PROCESS_PARAMETERS_ \

        sizeof (PROCESS_PARAMETERS)

// ------------------------------------------------------------

typedef struct _SYSTEM_STRINGS

        {

/*000*/ UNICODE_STRING SystemRoot;       // d:\WINNT

/*008*/ UNICODE_STRING System32Root;     // d:\WINNT\System32

/*010*/ UNICODE_STRING BaseNamedObjects; // \BaseNamedObjects

/*018*/ }

        SYSTEM_STRINGS,
```

```
     * PSYSTEM_STRINGS,

    **PPSYSTEM_STRINGS;

#define SYSTEM_STRINGS_ \

        sizeof (SYSTEM_STRINGS)

// ------------------------------------------------------------------

typedef struct _TEXT_INFO

        {

/*000*/ PVOID           Reserved;

/*004*/ PSYSTEM_STRINGS SystemStrings;

/*008*/ }

        TEXT_INFO,

     * PTEXT_INFO,

    **PPTEXT_INFO;

#define TEXT_INFO_ \

        sizeof (TEXT_INFO)

// ------------------------------------------------------------------

// located at 0x7FFDF000

typedef struct _PEB

        {

/*000*/ BOOLEAN              InheritedAddressSpace;
```

```
/*001*/ BOOLEAN               ReadImageFileExecOptions;

/*002*/ BOOLEAN               BeingDebugged;

/*003*/ BYTE                  b003;

/*004*/ DWORD                 d004;

/*008*/ PVOID                 SectionBaseAddress;

/*00C*/ PPROCESS_MODULE_INFO ProcessModuleInfo;

/*010*/ PPROCESS_PARAMETERS  ProcessParameters;

/*014*/ DWORD                 SubSystemData;

/*018*/ HANDLE                ProcessHeap;

/*01C*/ PCRITICAL_SECTION    FastPebLock;

/*020*/ PVOID                 AcquireFastPebLock; // function

/*024*/ PVOID                 ReleaseFastPebLock; // function

/*028*/ DWORD                 d028;

/*02C*/ PPVOID                User32Dispatch;     // function

/*030*/ DWORD                 d030;

/*034*/ DWORD                 d034;

/*038*/ DWORD                 d038;

/*03C*/ DWORD                 TlsBitMapSize;      // number of bits

/*040*/ PRTL_BITMAP           TlsBitMap;          // ntdll!TlsBitMap

/*044*/ DWORD                 TlsBitMapData [2];  // 64 bits
```

```
/*04C*/ PVOID                 p04C;

/*050*/ PVOID                 p050;

/*054*/ PTEXT_INFO            TextInfo;

/*058*/ PVOID                 InitAnsiCodePageData;

/*05C*/ PVOID                 InitOemCodePageData;

/*060*/ PVOID                 InitUnicodeCaseTableData;

/*064*/ DWORD                 KeNumberProcessors;

/*068*/ DWORD                 NtGlobalFlag;

/*06C*/ DWORD                 d6C;

/*070*/ LARGE_INTEGER         MmCriticalSectionTimeout;

/*078*/ DWORD                 MmHeapSegmentReserve;

/*07C*/ DWORD                 MmHeapSegmentCommit;

/*080*/ DWORD                 MmHeapDeCommitTotalFreeThreshold;

/*084*/ DWORD                 MmHeapDeCommitFreeBlockThreshold;

/*088*/ DWORD                 NumberOfHeaps;

/*08C*/ DWORD                 AvailableHeaps; // 16, *2 if exhausted

/*090*/ PHANDLE               ProcessHeapsListBuffer;

/*094*/ DWORD                 d094;

/*098*/ DWORD                 d098;

/*09C*/ DWORD                 d09C;
```

```
/*0A0*/ PCRITICAL_SECTION    LoaderLock;

/*0A4*/ DWORD                NtMajorVersion;

/*0A8*/ DWORD                NtMinorVersion;

/*0AC*/ WORD                 NtBuildNumber;

/*0AE*/ WORD                 CmNtCSDVersion;

/*0B0*/ DWORD                PlatformId;

/*0B4*/ DWORD                Subsystem;

/*0B8*/ DWORD                MajorSubsystemVersion;

/*0BC*/ DWORD                MinorSubsystemVersion;

/*0C0*/ KAFFINITY            AffinityMask;

/*0C4*/ DWORD                ad0C4 [35];

/*150*/ PVOID                p150;

/*154*/ DWORD                ad154 [32];

/*1D4*/ HANDLE               Win32WindowStation;

/*1D8*/ DWORD                d1D8;

/*1DC*/ DWORD                d1DC;

/*1E0*/ PWORD                CSDVersion;

/*1E4*/ DWORD                d1E4;

/*1E8*/ }

        PEB,
```

```
     * PPEB,

    **PPPEB;

#define PEB_ \

        sizeof (PEB)
```

列表 7-18.　进程环境块（PEB）

7.2、实时访问系统中的对象（Accessing Live System Objects）

前面的章节讲解了很多理论上的知识。我们现在需要一个实际的例子来演示对象管理机制，我为此编写了一个内核对象浏览器。该程序可以展示对象在分层结构中是如何排列的，以及如何获取这些对象的某些属性。不幸的是，ntoskrnl.exe 没有导出该程序所需的几个关键结构和相应的函数。这意味着即使是内核驱动程序也无法访问它们，它们成了系统的个人私藏品了。从另一方面来看，第六章介绍了如何通过计算 Windows 2000 的符号文件来访问这些未导出的数据和代码，这里的对象浏览器正好可以测试一下这种理论是否真的可行。很不错，第六章的符号调用接口通过了测试，w2k_obj.exe 示例程序的源代码位于本书 CD 的\src\w2k_obj 目录下。不过，该程序中最有趣的部分并不在 w2k_obj.c 中。第六章介绍的 w2k_call.dll 完成了大量的工作。因此，随后给出的示例代码均来自 w2k_call.c。

7.2.1、枚举对象目录项

你或许知道 Windows 2000 DDK 提供的一个小工具软件 objdir.exe，它位于\ntddk\bin 目录中。objdir.exe 可通过为文档化的 Native API 函数 NtQueryDirectoryObject()（由 ntdll.dll 导出）获取对象目录信息。已知相反的是，我的对象浏览器 w2k_obj.exe 将直接访问对象目录及其目录树中的叶子对象。这听起来很疯狂，但它确实做到了。最好的证据是可同时运行在 Windows 2000 和 Windows NT 4.0，它不依赖任何与特定版本相关的代码。不可否认，这两个版本中的对象结构稍微有些不同，但基本的模型还是一致的。提供一个可直接读取原始对象结构的程序而不是使用更高一级的 API 函数更能验证我们在前面提到的结构定义都是正确的。

在读取系统全局数据结构之前，最先要做的事情就是锁定它们。否则，系统可能会在另一个并发线程中改写这些数据，而我们的程序将可能会读取到无效数据或者到达一个空地址处。Windows 2000 为其维护的众多内部数据项提供了大量的锁。我们面临的唯一问题是：这些锁通常都没有导出。尽管内核驱动程序可完成所有在用户模式下无法完成的工作，但它还是不能安全的访问这些未导出的数据结构。不过，第六章讨论过的扩展的内核调用接口可完成这一工作，该扩展接口实现在 w2k_call.dll 中，它通过从操作系统的符号文件中查找这些内部符号的地址来完成这一工作。该 DLL 导出了如下三个对象管理 Thunks，以允许访问内核的对象目录：

1. __ObpRootDirectoryMutex()返回 ERESOURCE 锁的地址，该锁用于同步访问对象目录。
2. __ObpRootDirectoryObject()返回指向 OBJECT_DIRECTORY 结构的指针，该结构代表对象目录的 Root 节点。
3. __ObpTypeDirectoryObject()返回指向 OBJECT_DIRECTORY 结构的指针，该结构代表对象目录中的\ObjectTypes 节点。

应用程序在使用指向内核对象的指针时，必须非常小心，尤其是在获取全局锁之后。如果全局锁不能被适当的释放，那么系统将可能进入死锁状态。

尽管 root 目录的锁叫做 ObpRootDirectoryMutex，但严格来讲它并不是一个真正的 Mutex。它实际上是一个 ERESOURCE 而不是 KMUTEX。必须使用 ExAcquireResourceExclusiveLite()和 ExAcquireResourceSharedLite()函数来获取这些锁。函数名中的”Lite”后缀非常重要----永远不要使用这两个函数的兄弟函数：ExAcquireResourceExclusive ()和 ExAcquireResourceShared ()来获取 Windows 2000 或 Windows NT 4.0 的 ERSOURCE 锁。从 Windows NT 3.x 开始 ObpRootDirectoryMutex 结构就做了一些修改，而 ExAcquireResourceExclusive ()和 ExAcquireResourceShared ()函数仅能使用旧的 ERESOURCE 类型，该类型包含在 w2k_def.h 中（也可参考附录 C）。和 ExAcquireResource*Lite()极为类似的函数是 ExReleaseResourceLite()，该函数的老版本为：ExReleaseResource()。再次强调不要使用没有 Lite 后缀的函数。

本书提供的对象浏览器的基本工作方式为：首先锁定对象目录，然后获取整个层次结构中各节点的一个快照（snapshot），最后释放目录锁，然后再显示获取的快照数据。这种处理过程，可保证对系统的干扰最少，程序可有足够的时间来显示获取到的数据，而不会长时间占用系统。必须对系统对象的机构有非常深入的理解才能获取一个可靠的目录快照，因此

对象浏览器器将使用很多测试用例来保证对象信息的可靠性。这一任务可划分为如下两个基本任务：

1. 复制对象目录树的结构。这包括：复制和连接多个 OBJECT_DIRECTORY 结构，每个 OBJECT_DIRECTORY 结构都代表一个独立的非叶子节点（nonleaf node）。
2. 复制对象目录树的内容。这包括：复制 OBJECT_HEADER 以及叶节点的与其相关的结构。

列表 7-20 给出的 w2kDirectoryOpen()函数执行上述第一项工作。该函数先锁住根目录树，然后开始所有的子目录及 OBJECT_DIRECTORY 结构。为了获取完整的目录树结构，该函数必须针对每个 OBJECT_DIRECTORY 子结构递归的调用。由于每个对象目录节点都包含一个 Hash 表，该 Hash 表最多可有 37 个表项。每个 Hash 表项通过一个链表可存放任意数目的对象。所以，枚举目录项需要两个嵌套的循环：外循环扫描 Hash 表 37 个表项中所有非空表象项，内循环遍历表项中的链表。这就是 w2kDirectoryOpen()函数所做的一切。最后获取的数据在结构上和原始模型完全一致。基本的复制动作包括：自动分配所需的内存（由 w2kSpyClone()函数完成，该函数由 w2k_Call.dll 导出，参见**列表 6-30**）。**列表 7-20** 中的 w2kDirectoryClose()函数则用于释放 w2kDirectoryOpen()所占用的内存块。

```
POBJECT_DIRECTORY WINAPI

w2kDirectoryOpen (POBJECT_DIRECTORY pDir)

    {

    DWORD                       i;

    PERESOURCE                  pLock;

    PPOBJECT_DIRECTORY_ENTRY ppEntry;

    POBJECT_DIRECTORY        pDir1 = NULL;

    if (((pLock = __ObpRootDirectoryMutex ()) != NULL) &&

        _ExAcquireResourceExclusiveLite (pLock, TRUE))

        {
```

```
        if ((pDir1 = w2kSpyClone (pDir, OBJECT_DIRECTORY_)) != NULL)
            {
            for (i = 0; i < OBJECT_HASH_TABLE_SIZE; i++)
                {
                ppEntry = pDir1->HashTable + i;
                while (*ppEntry != NULL)
                    {
                    if ((*ppEntry =
                            w2kSpyClone (*ppEntry,
                                         OBJECT_DIRECTORY_ENTRY_))
                        != NULL)
                        {
                        (*ppEntry)->Object =
                            w2kObjectOpen ((*ppEntry)->Object);
                        ppEntry = &(*ppEntry)->NextEntry;
                        }
                    }
                }
            }
        _ExReleaseResourceLite (pLock);
```

```
        }

    return pDir1;

    }

// ------------------------------------------------------------

POBJECT_DIRECTORY WINAPI

w2kDirectoryClose (POBJECT_DIRECTORY pDir)

    {

    POBJECT_DIRECTORY_ENTRY pEntry, pEntry1;

    DWORD                   i;

    if (pDir != NULL)

        {

        for (i = 0; i < OBJECT_HASH_TABLE_SIZE; i++)

            {

            for (pEntry  = pDir->HashTable [i];

                 pEntry != NULL;

                 pEntry  = pEntry1)

                {

                pEntry1 = pEntry->NextEntry;

                w2kObjectClose   (pEntry->Object);

                w2kMemoryDestroy (pEntry);
```

```
                }

            }

        w2kMemoryDestroy (pDir);

        }

    return NULL;

    }
```

列表 7-20.　w2kDirectoryOpen()和 w2kDirectorClose()函数

仔细观察**列表 7-20**你会发现 w2kDirectoryOpen()和 w2kDirectorClose()在分别在其内部调用了 w2kObjectOpen()和 w2kObjectClose()函数。w2kObjectOpen()负责目录复制中的第二步：复制叶子对象。w2kObjectOpen()不会进行复制完整的对象，因为我们只需要识别每个对象的类型，并从对象体中复制适当的字节数即可。w2kObjectOpen()将复制对象的完整表头，及其下属的大部分结构体，然后构建一个“假的”对象体，该对象体中包含指向实际对象体的指针，和多个指向对象表头分段副本的指针。列表 7-21 给出了 w2kObjectOpen()构建和初始化的数据结构。W2K_OBJECT_FRAME 是一个集成的数据块，它包含对象表头副本和前面提到的“假的”对象体。这个“假的”对象体由 W2K_OBJECT 结构表示，该结构由一组指向 W2K_OBJECT_FRAME 各成员的指针。w2kObjectOpen()为 W2K_OBJECT_FRAME 结构分配内存并用原始对象的数据对其进行初始化，最后该函数将返回一个指向 W2K_OBJECT_FRAME 结构的 Object 成员的指针。如果你还记得前面讲述的有关对象体和对象头的内容，那么很容易看出，W2K_OBJECT_FRAME 实际上是对实际对象的模仿。这意味着，该结构拥有和原始对象头完全相同的对象头，应用程序可以象系统访问自己的内核对象一样，使用偏移量和标志位来访问“仿造”的 OBJECT_HEADER。

```
typedef struct _W2K_OBJECT

    {

    POBJECT                pObject;

    POBJECT_HEADER         pHeader;
```

```
    POBJECT_CREATOR_INFO  pCreatorInfo;

    POBJECT_NAME          pName;

    POBJECT_HANDLE_DB     pHandleDB;

    POBJECT_QUOTA_CHARGES pQuotaCharges;

    POBJECT_TYPE          pType;

    PQUOTA_BLOCK          pQuotaBlock;

    POBJECT_CREATE_INFO   pCreateInfo;

    PWORD                 pwName;

    PWORD                 pwType;

    }

    W2K_OBJECT, *PW2K_OBJECT, **PPW2K_OBJECT;

#define W2K_OBJECT_ sizeof (W2K_OBJECT)

// -------------------------------------------------------------------

typedef struct _W2K_OBJECT_FRAME

    {

    OBJECT_QUOTA_CHARGES QuotaCharges;

    OBJECT_HANDLE_DB     HandleDB;

    OBJECT_NAME          Name;

    OBJECT_CREATOR_INFO  CreatorInfo;

    OBJECT_HEADER        Header;
```

```
    W2K_OBJECT          Object;

    OBJECT_TYPE         Type;

    QUOTA_BLOCK         QuotaBlock;

    OBJECT_CREATE_INFO  CreateInfo;

    WORD                Buffer [];

    }

    W2K_OBJECT_FRAME, *PW2K_OBJECT_FRAME, **PPW2K_OBJECT_FRAME;

#define W2K_OBJECT_FRAME_ sizeof (W2K_OBJECT_FRAME)

#define W2K_OBJECT_FRAME__(_n) (W2K_OBJECT_FRAME_ + ((_n) * WORD_))
```

列表 7-21. 对象的克隆结构

我不想深入讲解 w2kObjectOpen()及其子结构的细节。出于说明的目的，列表 7-22 给出的函数就已足够了。w2kObjectHeader()用于创建一个实际对象的 OBJECT_HEADER 的副本，w2kObjectCreateInfo()和 w2kObjectName()则用于复制对象表头中的 OBJECT_CREATOR_INFO 和 OBJECT_NAME 结构（如果这些结构存在的话）。再次强调，w2kSpyClone()完成了大部分的主要工作。更多的信息，请参考本书光盘中的 w2k_call.c 源代码。

```
#define BACK(_p,_d) ((PVOID) ( ( (PBYTE) (_p) ) - (_d) ) )

// -------------------------------------------------------------

POBJECT_HEADER WINAPI

w2kObjectHeader (POBJECT pObject)

    {

    DWORD          dOffset = OBJECT_HEADER_;
```

```
    POBJECT_HEADER pHeader = NULL;

    if (pObject != NULL)
        {
        pHeader = w2kSpyClone (BACK (pObject, dOffset),
                               dOffset);
        }
    return pHeader;
    }
// -------------------------------------------------------------
POBJECT_CREATOR_INFO WINAPI
w2kObjectCreatorInfo (POBJECT_HEADER pHeader,
                      POBJECT        pObject)
    {
    DWORD                dOffset;
    POBJECT_CREATOR_INFO pCreatorInfo = NULL;
    if ((pHeader != NULL) && (pObject != NULL) &&
        (pHeader->ObjectFlags & OB_FLAG_CREATOR_INFO))
        {
        dOffset = OBJECT_CREATOR_INFO_ + OBJECT_HEADER_;
        pCreatorInfo = w2kSpyClone (BACK (pObject, dOffset),
```

```
                                        OBJECT_CREATOR_INFO_);

        }

    return pCreatorInfo;

    }

// ------------------------------------------------------------------

POBJECT_NAME WINAPI

w2kObjectName (POBJECT_HEADER pHeader,

               POBJECT        pObject)

    {

    DWORD        dOffset;

    POBJECT_NAME pName = NULL;

    if ((pHeader != NULL) && (pObject != NULL) &&

        (dOffset = pHeader->NameOffset))

        {

        dOffset += OBJECT_HEADER_;

        pName = w2kSpyClone (BACK (pObject, dOffset),

                             OBJECT_NAME_);

        }

    return pName;

    }
```

列表 7-22. 用于对象克隆的辅助函数

在整个复制过程的最后，w2kDirectoryOpen()接受一个指向实时的 OBJECT_DIRECTORY 的节点，并返回一个副本，该副本中包含一个 W2K_OBJECT 指针，该指针指向的就是原始的目录对象的对象体。对象浏览器针对每一层目录都调用该函数。列表 7-23 给出了经过修改后的浏览器代码，将其核心部分直接暴露出来。w2k_obj.c 中的原始代码包含很多容易让人分散注意力的代码。最高层的函数是 DisplayObjects()。该函数需要 __ObpRootDirectoryObject()为其提供一个 root 对象的指针，随后它将显示传入对象的类型和名称，如果传入的对象是 OBJECT_DIRECTORY，它会再次调用自身（递归调用）。DisplayObject()会为每个嵌套层增加三个空格。我将**列表 7-23** 中的函数加入到了 w2k_obj.c 中，不过，尽管该函数可以工作，但最好不要调用它。

```
VOID WINAPI _DisplayObject (PW2K_OBJECT pObject,
                            DWORD       dLevel)
    {
    POBJECT_DIRECTORY       pDir;
    POBJECT_DIRECTORY_ENTRY pEntry;
    DWORD                   i;
    for (i = 0; i < dLevel; i++) printf (L"   ");
    _printf (L"%+.-16s%s\r\n", pObject->pwType, pObject->pwName);
    if ((!lstrcmp (pObject->pwType, L"Directory")) &&
        ((pDir = w2kDirectoryOpen (pObject->pObject)) != NULL))
        {
        for (i = 0; i < OBJECT_HASH_TABLE_SIZE; i++)
            {
            for (pEntry  = pDir->HashTable [i];
```

```
                pEntry != NULL;

                pEntry  = pEntry->NextEntry)

                {

                _DisplayObject (pEntry->Object, dLevel+1);

                }

            }

        w2kDirectoryClose (pDir);

        }

    return;

    }

// ---------------------------------------------------------------

VOID WINAPI _DisplayObjects (VOID)

    {

    PW2K_OBJECT pObject;

    if ((pObject = w2kObjectOpen (__ObpRootDirectoryObject ()))

        != NULL)

        {

        _DisplayObject (pObject, 0);

        w2kObjectClose (pObject);

        }
```

```
    return;

    }
```

列表 7-23.　一个非常简单的对象浏览器

示例 7-2 给出的对象目录的特征就是通过列表 7-23 中的代码获取到的。例如，\BaseNamedObjects 子结构包含一个命名对象，该对象通常被多个进程共享，并可根据对象名来打开。\ObjectTypes 子结构包含系统支持的所有 27 种 OBJECT_TYPE 类型对象（参考**列表 7-9**），**表 7-4** 也列出了这些对象。

```
Directory. . . . . \
   Directory. . . . . ArcName
      SymbolicLink. . . .multi(0)disk(0)rdisk(0)
      SymbolicLink. . . .multi(0)disk(0)rdisk(1)
      SymbolicLink. . . .multi(0)disk(0)rdisk(1)partition(1)
      SymbolicLink. . . .multi(0)disk(0)rdisk(0)partition(1)
      SymbolicLink. . . .multi(0)disk(0)fdisk(0)
      SymbolicLink. . . .multi(0)disk(0)rdisk(0)partition(2)
   Device. . . . . Ntfs
   Port. . . . . SeLsaCommandPort
   Key. . . . . REGISTRY
   Port. . . . . XactSrvLpcPort
   Port. . . . . DbgUiApiPort
   Directory. . . . . NLS
      Section. . . . . NlsSectionCP874
      Section. . . . . NlsSectionCP950
      Section. . . . . NlsSectionCP20290
      Section. . . . . NlsSectionCP1255c 1255.nls

   Directory. . . . . BaseNamedObjects
      Section. . . . . Df SharedHeapE445BB
      Section. . . . . DFMap0-14765686
      Mutant. . . . . ZonesCacheCounterMutex
      Section. . . . . DFMap0-14364447
      Event. . . . . WINMGMT CORBDLL UNLOADED
      Mutant. . . . . MCICDA DeviceCritSec 19
      Event. . . . . AgentToWkssvcEvent
      Event. . . . . userenv: Machine Group Policy has been applied
      SymbolicLink. . . .Local
      Section. . . . . DFMap0-15555297
      Section. . . . . Df SharedHeapED2256
      Section. . . . . Df SharedHeapE8F975
      Section. . . . . DFMap0-15232696
      Section. . . . . DFMap0-15170325
      Event. . . . . Shell NotificationCallbacksOutstanding
      Section. . . . . DFMap0-14364985
      Event. . . . . SETTermEvent
      Event. . . . . winlogon:  User GPO Event 112121

   Directory. . . . . ObjectTypes
      Type. . . . . Directory
      Type. . . . . Mutant
      Type. . . . . Thread
      Type. . . . . Controller
      Type. . . . . Profile
      Type. . . . . Event
      Type. . . . . Type
      Type. . . . . Section
      Type. . . . . EventPair
      Type. . . . . SymbolicLink
      Type. . . . . Desktop
      Type. . . . . Timer
      Type. . . . . File
      Type. . . . . WindowStation
      Type. . . . . Driver
      Type. . . . . WmiGuid
      Type. . . . . Device
      Type. . . . . Token
      Type. . . . . IoCompletion
      Type. . . . . Process
      Type. . . . . Adapter
      Type. . . . . Key
      Type. . . . . Job
      Type. . . . . WaitablePort
      Type. . . . . Port
      Type. . . . . Callback
      Type. . . . . Semaphore
   Directory. . . . . Security
      Event. . . . . TRKWKS EVENT
      WaitablePort. . . .TRKWKS_PORT
      Event. . . . . LSA AUTHENTICATION INITIALIZED
      Event. . . . . NetworkProviderLoad
```

示例 7-2.　对象目录的一个摘要

w2k_obj.exe 是一个拥有全部功能的对象浏览器，它不仅可以以多种格式显示对象目录树，而且还允许显示对象的一些附加特性并且支持按照对象类型进行过滤。示例 7-3 展示 w2k_obj.exe 提供的多个命令行选项。

```
// w2k_obj.exe

// SBS Windows 2000 Object Browser V1.00

// 08-27-2000 Sven B. Schreiber

// sbs@orgon.com

Usage: w2k_obj [+-atf] [<type>] [<#>|-1] [/root] [/types]

       +a -a  : show/hide object addresses  (default: -a)

       +t -t  : show/hide object type names (default: -t)

       +f -f  : show/hide object flags      (default: -f)

       <type> : show <type> objects only    (default:  *)

       <#>    : show <#> directory levels   (default: -1)

       -1     : show all directory levels

       /root  : show ObpRootDirectoryObject tree

       /types : show ObpTypeDirectoryObject tree

Example: w2k_obj +atf *port 2 /root

This command displays all Port and WaitablePort objects,

starting in the root and scanning two directory levels.

Each line includes address, type, and flag information.
```

示例 7-3. w2k_obj.exe 的命令行帮助

在**示例 7-4** 中，我演示了在命令行帮助中提到的命令：`w2k_obj  +atf *port 2 /root`。该命令将仅输出 Port 和 WaitablePort 对象，因为指定了类型过滤表达式：*port，此外还将显示对象体的地址、类型名和每项的标志。输出被限制为两个子目录层（命令行中参数 2 的作用）。

```
Root directory contents: (2 levels shown)


 8149CDD0 Directory     <32> \
> |   E26A0540 Port          <24> SeLsaCommandPort
> |   E130CC20 Port          <24> XactSrvLpcPort
> |   E13E2380 Port          <24> DbgUiApiPort
> |   E13E4BA0 Port          <26> SeRmCommandPort
> |   E26A9D20 Port          <24> LsaAuthenticationPort
> |   E13E4CA0 Port          <24> DbgSsApiPort
> |   E13E3260 Port          <24> SmApiPort
> |   E2707680 Port          <24> ErrorLogPort
  |   81499B70 Directory     <32> \ArcName
  |   812FDB60 Directory     <10> \NLS
  |   814940B0 Directory     <32> \Driver
    81490B30 Directory     <32> \WmiGuid
  |   81499A90 Directory     <32> \Device
  |     | 814AEA90 Directory     <32> \Device\DmControl
  |     |  814AE4F0 Directory     <32> \Device\HarddiskDmVolumes
  |     |  8148BE50 Directory     <32> \Device\Ide
  |     |  814AB3D0 Directory     <32> \Device\Harddisk0
  |     |  814852F0 Directory     <32> \Device\Harddisk1
  |     |  814A9F50 Directory     <22> \Device\WinDfs
  |     \ 814AB030 Directory     <32> \Device\Scsi
  |   81319030 Directory     <30> \Windows
>       E2615520 Port          <24> SbApiPort
>       E260E1A0 Port          <24> ApiPort
        812FC810 Directory     <32> \Windows\WindowStations
    81319150 Directory     <30> \RPC Control
>       E26B6A20 Port          <24> tapsrvlpc
>       E3228440 Port          <24> OLE3c
>       E269F360 Port          <24> spoolss
>       E269B6E0 Port          <24> OLE2
>       E2C96C60 Port          <24> OLE3f
>       E1306BC0 Port          <24> OLE3
>       E269BD20 Port          <24> LRPC0000021c.00000001
>       E276D520 Port          <24> OLE5
>       E2699D40 Port          <24> OLE6
>       E2697C00 Port          <24> OLE7
>       E26F0AE0 Port          <24> ntsvcs
>       E26B6B20 Port          <24> policyagent
>       E2814CA0 Port          <24> OLEa
>       E29DC3C0 Port          <24> OLEb
>       E304C8A0 Port          <24> OLE40
>       E3165660 Port          <24> OLE41
>       E26979A0 Port          <24> eqmapper
> |     E13069A0 Port          <24> senssvc
> |   \ E2C8D040 Port          <24> OLE42
  |   812FD030 Directory     <30> \BaseNamedObjects
  |   \ 812FDF50 Directory     <30> \BaseNamedObjects\Restricted
  |   8149CBD0 Directory     <32> \??
  1   814B5030 Directory     <32> \FileSystem
  |   8149CCB0 Directory     <32> \ObjectTypes
  |   81499C50 Directory     <32> \Security
> |   \_ 8121EB20 WaitablePort___<24> TRKWKS_PORT
  |   8149B2D0 Directory     <32> \Callback
  \   81446E90 Directory     <30> \KnownDlls

54 objects
```

示例 7-4. 命令：w2k_obj +atf *port 2 /root 的输出结果

注意，即使与提供的类型名称模版（type name pattern）不匹配目录对象也将包含在相同的链表中，否则，将无法在目录树中找到与匹配对象相关联的节点。在第一列出现的>符号是为了在附近加的目录对象中将匹配对象类型区分出来。

7.3、我们将走向哪里？

有关 Windows 2000 内部的东西还有很多值得一说。但一本长度适当的书是无法容纳如此多的内容的，因此，我们必须在某个地方结束它。本书的七章内容读起来一定很费劲，或许还有些恐怖。但如果你现在能以不同的眼光看待 Windows 2000 的话，那么我的目的就达到了。如果你是一个程序员或调试工具开发人员，那么本书中的编程技巧和接口技术将帮助你向你的产品中加入你的竞争对手还无法提供的功能。如果你开发针对 Windows 2000 的其它类型的程序，那么通过理解本书讲解的系统内部机理，你将编写出更高效、更能利用操作系统特性的代码。我还希望本书能激励更多开发人员的钻研精神，从而进一步揭开笼罩在 Windows 2000 内核上的神秘面纱。

我从来不认为将操作系统当作一个黑盒子是一种好的编程范例，并且我现在仍然这样想。

附录 B

内核 API 函数（Kernel API Functions）

附录 B 包含在第二章讨论的系统模块：win32k.sys、ntdll.dll 和 ntoskrnl.exe 导出的函数列表。N/A 表示不支持（Not Available）。

表 B-1. Windows 2000 Native API

	函数名称	INT 2eh	Ntdll.Nt*	Ntdll.Zw*	Ntoskrnl.Nt*	Ntoskrnl.Zw*
1	NtAcceptConnectPort	0x0000			N/A	N/A
2	NtAccessCheck	0x0001			N/A	N/A
3	NtAccessCheckAndAuditAlarm	0x0002			N/A	
4	NtAccessCheckByType	0x0003			N/A	N/A
5	NtAccessCheckByTypeAndAuditAlarm	0x0004			N/A	N/A
6	NtAccessCheckByTypeResultList	0x0005			N/A	N/A
7	NtAccessCheckByTypeResultListAndAuditAlarm	0x0006			N/A	N/A
8	NtAccessCheckByTypeResultListAndAuditAlarmByHandle	0x0007			N/A	N/A
9	NtAddAtom	0x0008				N/A
10	NtAdjustGroupsToken	0x0009			N/A	N/A
11	NtAdjustPrivilegesToken	0x000A				
12	NtAlertResumeThread	0x000B			N/A	N/A
13	NtAlertThread	0x000C			N/A	

14	NtAllocateLocallyUnique ld	0x000D				N/A
15	NtAllocateUserPhysicalP ages	0x000E			N/A	N/A
16	NtAllocateUuids	0x000F				N/A
17	NtAllocateVirtualMemory	0x0010				
18	NtAreMappedFilesTheSame	0x0011			N/A	N/A
19	NtAssignProcessToJobObj ect	0x0012			N/A	N/A
20	NtBuildNumber	N/A	N/A	N/A		N/A
21	NtCallbackReturn	0x0013			N/A	N/A
22	NtCancelDeviceWakeupReq uest	0x0016			N/A	N/A
23	NtCancelloFile	0x0014			N/A	
24	NtCancelTimer	0x0015			N/A	
25	NtClearEvent	0x0017			N/A	
26	NtClose	0x0018				
27	NtCloseObjectAuditAlarm	0x0019			N/A	
28	NtCompleteConnectPort	0x001A			N/A	N/A
29	NtConnectPort	0x001B				
30	NtContinue	0x001C			N/A	N/A
31	NtCreateChannel	0x00F1			N/A	N/A
32	NtCreateDirectoryObject	0x001D			N/A	
33	NtCreateEvent	0x001E				
34	NtCreateEventPair	0x001F			N/A	N/A
35	NtCreateFile	0x0020				
36	NtCreateloCompletion	0x0021			N/A	N/A
37	NtCreateJobObject	0x0022			N/A	N/A
38	NtCreateKey	0x0023			N/A	

39	NtCreateMailslotFile	0x0024			N/A	N/A
40	NtCreateMutant	0x0025			N/A	N/A
41	NtCreateNamedPipeFile	0x0026			N/A	N/A
42	NtCreatePagingFile	0x0027			N/A	N/A
43	NtCreatePort	0x0028			N/A	N/A
44	NtCreateProcess	0x0029			N/A	N/A
45	NtCreateProfile	0x002A			N/A	N/A
46	NtCreateSection	0x002B				
47	NtCreateSemaphore	0x002C			N/A	N/A
48	NtCreateSymbolicLinkObject	0x002D			N/A	
49	NtCreateThread	0x002E			N/A	N/A
50	NtCreateTimer	0x002F			N/A	
51	NtCreateToken	0x0030			N/A	N/A
52	NtCreateWaitablePort	0x0031			N/A	N/A
53	NtCurrentTeb	N/A		N/A	N/A	N/A
54	NtDelayExecution	0x0032			N/A	N/A
55	NtDeleteAtom	0x0033				N/A
56	NtDeleteFile	0x0034				
57	NtDeleteKey	0x0035			N/A	
58	NtDeleteObjectAuditAlarm	0x0036			N/A	N/A
59	NtDeleteValueKey	0x0037			N/A	
60	NtDeviceloControlFile	0x0038				
61	NtDisplayString	0x0039			N/A	
62	NtDuplicateObject	0x003A				
63	NtDuplicateToken	0x003B				
64	NtEnumerateKey	0x003C			N/A	
65	NtEnumerateValueKey	0x003D			N/A	

66	NtExtendSection	0x003E			N/A	N/A
67	NtFilterToken	0x003F			N/A	N/A
68	NtFindAtom	0x0040				N/A
69	NtFlushBuffersFile	0x0041			N/A	N/A
70	NtFlushInstructionCache	0x0042			N/A	
71	NtFlushKey	0x0043			N/A	
72	NtFlushVirtualMemory	0x0044			N/A	
73	NtFlushWriteBuffer	0x0045			N/A	N/A
74	NtFreeUserPhysicalPages	0x0046			N/A	N/A
75	NtFreeVirtualMemory	0x0047				
76	NtFsControlFile	0x0048				
77	NtGetContextThread	0x0049			N/A	N/A
78	NtGetDevicePowerState	0x004A			N/A	N/A
79	NtGetPlugPlayEvent	0x004B			N/A	N/A
80	NtGetTickCount	0x004C			N/A	N/A
81	NtGetWriteWatch	0x004D			N/A	N/A
82	NtGlobalFlag	N/A	N/A	N/A		N/A
83	NtImpersonateAnonymousToken	0x004E			N/A	N/A
84	NtImpersonateClientOfPort	0x004F			N/A	N/A
85	NtImpersonateThread	0x0050			N/A	N/A
86	NtInitializeRegistry	0x0051			N/A	N/A
87	NtInitiatePowerAction	0x0052			N/A	
88	NtIsSystemResumeAutomatic	0x0053			N/A	N/A
89	NtListenChannel	0x00F2			N/A	N/A
90	NtListenPort	0x0054			N/A	N/A
91	NtLoadDriver	0x0055			N/A	

92	NtLoadKey	0x0056			N/A	
93	NtLoadKey2	0x0057			N/A	N/A
94	NtLockFile	0x0058				N/A
95	NtLockVirtualMemory	0x0059			N/A	N/A
96	NtMakeTemporaryObject	0x005A			N/A	
97	NtMapUserPhysicalPages	0x005B			N/A	N/A
98	NtMapUserPhysicalPagesScatter	0x005C			N/A	N/A
99	NtMapViewOf Section	0x005D				
100	NtNotifyChangeDirectoryFile	0x005E				N/A
101	NtNotifyChangeKey	0x005F			N/A	
102	NtNotifyChangeMultipleKeys	0x0060			N/A	N/A
103	NtOpenChannel	0x00F3			N/A	N/A
104	NtOpenDirectoryObject	0x0061			N/A	
105	NtOpenEvent	0x0062			N/A	
106	NtOpenEventPair	0x0063			N/A	N/A
107	NtOpenFile	0x0064				
108	NtOpenloCompletion	0x0065			N/A	N/A
109	NtOpenJobObject	0x0066			N/A	N/A
110	NtOpenKey	0x0067			N/A	
111	NtOpenMutant	0x0068			N/A	N/A
112	NtOpenObjectAuditAlarm	0x0069			N/A	N/A
113	NtOpenProcess	0x006A				
114	NtOpenProcessToken	0x006B				
115	NtOpenSection	0x006C			N/A	
116	NtOpenSemaphore	0x006D			N/A	N/A
117	NtOpenSymbolicLinkObjec	0x006E			N/A	

	t					
118	NtOpenThread	0x006F			N/A	
119	NtOpenThreadToken	0x0070			N/A	
120	NtOpenTimer	0x0071			N/A	
121	NtPlugPlayControl	0x0072			N/A	N/A
122	NtPowerlnformation	0x0073			N/A	
123	NtPrivilegeCheck	0x0074			N/A	N/A
124	NtPrivilegedServiceAuditAlarm	0x0075			N/A	N/A
125	NtPrivilegeObjectAuditAlarm	0x0076			N/A	N/A
126	NtProtectVirtualMemory	0x0077			N/A	N/A
127	NtPulseEvent	0x0078			N/A	
128	NtQueryAttributesFile	0x007A			N/A	N/A
129	NtQueryDefaultLocale	0x007B			N/A	
130	NtQueryDefaultUILanguage	0x007C			N/A	
131	NtQueryDirectoryFile	0x007D				
132	NtQueryDirectoryObject	0x007E			N/A	
133	NtQueryEaFile	0x007F				
134	NtQueryEvent	0x0080			N/A	N/A
135	NtQueryFullAttributesFile	0x0081			N/A	N/A
136	NtQuerylnformationAtom	0x0079				N/A
137	NtQuerylnformationFile	0x0082				
138	NtQuerylnformationJobObject	0x0083			N/A	N/A
139	NtQuerylnformationPort	0x0085			N/A	N/A
140	NtQuerylnformationProce	0x0086				

	ss					
141	NtQuerylnformationThread	0x0087			N/A	N/A
142	NtQuerylnformationToken	0x0088				
143	NtQuerylnstallUILanguage	0x0089			N/A	
144	NtQuerylntervalProfile	0x008A			N/A	N/A
145	NtQueryIoCompletion	0x0084			N/A	N/A
146	NtQueryKey	0x008B			N/A	
147	NtQueryMultipleValueKey	0x008C			N/A	N/A
148	NtQueryMutant	0x008D			N/A	N/A
149	NtQueryObject	0x008E			N/A	
150	NtQueryOpenSubKeys	0x008F			N/A	N/A
151	NtQueryPerformanceCounter	0x0090			N/A	N/A
152	NtQueryQuotalnformation File	0x0091				N/A
153	NtQuerySection	0x0092			N/A	
154	NtQuerySecurityObject	0x0093				
156	NtQuerySemaphore	0x0094			N/A	N/A
157	NtQuerySymbolicLinkObject	0x0095			N/A	
158	NtQuerySystemEnvironment Value	0x0096			N/A	N/A
159	NtQuerySystemlnformation	0x0097				
160	NtQuerySystemTime	0x0098			N/A	N/A
161	NtQuery Timer	0x0099			N/A	N/A
162	NtQueryTimerResolution	0x009A			N/A	N/A

163	NtQueryValueKey	0x009B			N/A	
164	NtQuery VirtualMemory	0x009C			N/A	N/A
165	NtQuery VolumeInformationFile	0x009D				
166	NtQueueApcThread	0x009E			N/A	N/A
167	NtRaiseException	0x009F			N/A	N/A
168	NtRaiseHardError	0x00A0			N/A	N/A
169	NtReadFile	0x00A1				
170	NtReadFileScatter	0x00A2			N/A	N/A
171	NtReadRequestData	0x00A3			N/A	N/A
172	NtReadVirtualMemory	0x00A4			N/A	N/A
173	NtRegisterThreadTerminatePort	0x00A5			N/A	N/A
174	NtReleaseMutant	0x00A6			N/A	N/A
175	NtReleaseSemaphore	0x00A7			N/A	N/A
176	NtRemoveIoCompletion	0x00A8			N/A	N/A
177	NtReplaceKey	0x00A9			N/A	
178	NtReplyPort	0x00AA			N/A	N/A
179	NtReplyWaitReceivePort	0x00AB			N/A	N/A
180	NtReplyWaitReceivePortEx	0x00AC			N/A	N/A
181	NtReplyWaitReplyPort	0x00AD			N/A	N/A
182	NtReplyWaitSendChannel	0x00F4			N/A	N/A
183	NtRequestDeviceWakeup	0x00AE			N/A	N/A
184	NtRequestPort	0x00AF				N/A
185	NtRequestWaitReplyPort	0x00B0				
186	NtRequestWakeupLatency	0x00B1			N/A	N/A
187	NtResetEvent	0x00B2			N/A	
188	NtResetWriteWatch	0x00B3			N/A	N/A

189	NtRestoreKey	0x00B4			N/A	
190	NtResumeThread	0x00B5			N/A	N/A
191	NtSaveKey	0x00B6			N/A	
192	NtSaveMergedKeys	0x00B7			N/A	N/A
193	NtSecureConnectPort	0x00B8			N/A	N/A
194	NtSendWaitReplyChannel	0x00F5			N/A	N/A
195	NtSetContextChannel	0x00F6			N/A	N/A
196	NtSetContextThread	0x00BA			N/A	N/A
197	NtSetDefaultHardErrorPort	0x00BB			N/A	N/A
198	NtSetDefaultLocale	0x00BC			N/A	
199	NtSetDefaultUILanguage	0x00BD			N/A	
200	NtSetEaFile	0x00BE				
201	NtSetEvent	0x00BF				
202	NtSetHighEventPair	0x00C0			N/A	N/A
203	NtSetHighWaitLowEventPair	0x00C1			N/A	N/A
204	NtSetlnformationFile	0x00C2				
205	NtSetlnformationJobObject	0x00C3			N/A	N/A
206	NtSetlnformationKey	0x00C4			N/A	N/A
207	NtSetlnformationObject	0x00C5			N/A	
208	NtSetlnformationProcess	0x00C6				
209	NtSetlnformationThread	0x00c7				
210	NtSetlnformationToken	0x00C8			N/A	N/A
211	NtSetlntervalProfile	0x00C9			N/A	N/A
212	NtSetloCompletion	0x00B9			N/A	N/A
213	NtSetLdtEntries	0x00CA			N/A	N/A
214	NtSetLowEventPair	0x00CB			N/A	N/A

215	NtSetLowWaitHighEventPair	0x00CC			N/A	N/A
216	NtSetQuotaInformationFile	0x00CD				N/A
217	NtSetSecurityObject	0x00CE				
218	NtSetSystemEnvironmentValue	0x00CF			N/A	N/A
219	NtSetSystemInformation	0x00D0			N/A	
220	NtSetSystemPowerState	0x00D1			N/A	N/A
221	NtSetSystemTime	0x00D2			N/A	
222	NtSetThreadExecutionState	0x00D3			N/A	N/A
223	NtSetTimer	0x00D4			N/A	
224	NtSetTimerResolution	0x00D5			N/A	N/A
225	NtSetUuidSeed	0x00D6			N/A	N/A
226	NtSetValueKey	0x00D7			N/A	
227	NtSetVolumeInformationFile	0x00D8				
228	NtShutdownSystem	0x00D9			N/A	N/A
229	NtSignalAndWaitForSingleObject	0x00DA			N/A	N/A
230	NtStartProfile	0x00DB			N/A	N/A
231	NtStopProfile	0x00DC			N/A	N/A
232	NtSuspendThread	0x00DD			N/A	N/A
233	NtSystemDebugControl	0x00DE			N/A	N/A
234	NtTerminateJobObject	0x00DF			N/A	N/A
235	NtTerminateProcess	0x00E0			N/A	
236	NtTerminateThread	0x00E1			N/A	N/A
237	NtTestAlert	0x00E2			N/A	N/A

238	NtUnloadDriver	0x00E3			N/A	
239	NtUnloadKey	0x00E4			N/A	
240	NtUnlockFile	0x00E5				N/A
241	NtUnlockVirtualMemory	0x00E6			N/A	N/A
242	NtUnmapViewOfSection	0x00E7			N/A	
243	NtVdmControl	0x00E8				N/A
244	NtWaitForMultipleObjects	0x00E9			N/A	
245	NtWaitForSingleObject	0x00EA				
246	NtWaitHighEventPair	0x00EB			N/A	N/A
247	NtWaitLowEventPair	0x00EC			N/A	N/A
248	NtWriteFile	0x00ED				
249	NtWriteFileGather	0x00EE			N/A	N/A
250	NtWriteRequestData	0x00EF			N/A	N/A
251	NtWriteVirtualMemory	0x00F0			N/A	N/A
252	NtYieldExecution	0x00F7			N/A	