继往开来创新高，推陈出新品佳酿

1993 年，本书第 1 版 *Advanced NT* 出版的时候，我和三个朋友一起成立了一个“四喜工作室”。由于四个人只有一台计算机，所以我们几个每天一睁眼，第一件事情便是抢占计算机，这台 386 配置简单，根本无法与现在的计算机相提并论，而且当时也没有网络，所以计算机的用途非常有限，主要也就是文字处理，玩游戏，编简单程序等，但它带给我们的乐趣至今难以忘怀。受限于当时的环境，数据和游戏的交换也基本上在圈内好友之间进行，就像搞地下活动一样约好时间地点碰头。幸运的是，由此结交了一大批计算机爱好者，后来他们大多成为 IT 届的领军人物。

其时，从大环境看，我国网络也开始悄然起步。1993 年年初，中国科学院高能物理研究所接入斯坦福大学线性加速器中心的 64K 专线开通，国内科学家开始在国内使用电子邮件。随后几个月的时间，金桥工程和域名体系的确立和部署，三大院校网的连接，最终将我国带入信息高速公路，推动我国 IT 业的迅猛发展。

由此而来的便是计算机类图书和报纸期刊的炙手可热，《电脑报》等 IT 媒体相继崛起，计算机图书更是出现供不应求的现象，在当时，即便是国外引进翻译出版的图书，也能轻松突破几万册的销量，计算机图书的发展达到全盛时期。

在这个时期，国内开发人员先后成为 Jeffrey 和 McConell 等大师的拥趸。因为在 IT 界，虽然资深程序员不胜枚举，但同时又是深受程序员喜爱的技术图书作家的乏善可陈。而像他们那样，曾经写过多部书，部部都引人入胜，令人醍醐灌顶，就更是凤毛麟角。他们是 Windows 编程世界中的中流砥柱，也是 Windows 技术当之无愧的布道者。曾有不少读者放言，只要是 Jeffrey 的书，他们必定会花时间研读，并加以收藏。这一点都不夸张，我们同时代的很多人都是在这批书的滋润下成长起来的。他们熟读了 *Advanced NT* 之后，又如痴如狂地捧起了 *Advanced Windows* 和 *Programming Application for Microsoft Windows* 等续作。他们是 Jeffrey 的粉丝，同时也是微软开发阵营的主力军。

随着微软宣布放弃对 Windows XP 以及以前版本的支持，Windows Vista 的普及势在必行，迟早会安装到普通用户的计算机上。Windows Vista 有很多吸引人的新特性，相信大家不用不知道，一用忘不了。（在翻译 Microsoft Press 的 *Windows Vista Inside Out* 一书的过程中，我已经深切体会到她的妙处）。作为一名程序员，有必要在第一时间适应在新的操作系统下的编程。历经 15 年，本书也随着 Windows 操作系统的“改朝换代”，升级到第 5 版，即 *Windows via C/C++*。如果您要用 C/C++开发 Windows 应用程序，那就不要走弯路，直接让 Jeffrey 告诉您如何利用 Windows 的新特性和新函数来编写出高效、优美的 Windows 应用程序。

对于本书的学习，谨以《史记 孔子世家》中孔子学琴一文与大家共勉（请原谅，这里引用了我另一本书的译序，其寓意深刻，忍不住又拿出来与大家分享☺）：

孔子学鼓琴师襄子，十日不进。师襄子曰：“可以益矣。”孔子曰：“丘已习其曲矣，未得其数也。”有间，曰：“已习其数，可以益矣。”孔子曰：“丘未得其志也。”有间，曰：“已习其志，可以益矣。”孔子曰：“丘未得其为人也。”有间，有所穆然深思焉，有所怡然高望而远志焉。曰：“丘得其为人，黯然而黑，几然而长，眼如望羊，如王四国，非文王其谁能为此也！”师襄子辟席再拜，曰：“师盖云文王操也。”

期望读者朋友也能达到学习的三大境界：学习掌握演奏（编程）的技巧；领会其中的志趣；熟悉乐曲（程序）的作者。

翻译过程中，感谢我的家人和朋友的诸多帮助和理解，尤其要感谢我的乖女儿。这个暑假，她的成长令人激赏！

最后，欢迎读者指出本书的疏漏和不足之处 ，如果你对我翻译的部分（1~6 章）有什么意见和建议，请访问我的博客（transbot.blog.163.com）指出，那里为我翻译的一些图书开辟了专栏，专门用于和读者们分享勘误和其他有用的信息。

周　　靖

Beijing 2008 前夕

第Ⅰ部分　程序员必读

本部分内容包括：
第 1 章　错误处理
第 2 章　字符和字符串处理
第 3 章　内核对象

第 1 章　错误处理

本章内容包括：
1.1　定义自己的错误代码
1.2　ErrorShow 示例程序

在深入讨论 Microsoft Windows 提供的诸多特性之前，应该先理解各个 Windows 函数是如何进行错误处理的。

调用 Windows 函数时，它会先验证你传给它的参数，然后再开始执行任务。如果传入的参数无效，或者由于其他原因导致操作无法执行，则函数的返回值将指出函数在某个方面失败了。表 1-1 展示了大多数 Windows 函数使用的返回值的数据类型。

表 1-1　常见的 Windows 函数返回值数据类型

数据类型	指出函数调用失败的值
VOID	这个函数不可能失败。只有极少数 Windows 函数的返回值类型为 VOID。
BOOL	如果函数失败，返回值为 0；否则，返回值是一个非零值。应避免测试返回值是否为 TRUE；最稳妥的做法是检查它是否不为 FALSE。
HANDLE	如果函数失败，则返回值通常为 NULL；否则，HANDLE 将标识一个你可以操纵的对象。请注意这种返回值，因为某些函数会返回为 INVALID_HANDLE_VALUE 的一个句柄值，它被定义为–1。函数的 Platform SDK 文档清楚说明了函数是返回 NULL 还是 INVALID_HANDLE_VALUE 来标识失败。
PVOID	如果函数调用失败，返回值为 NULL；否则，PVOID 将标识一个数据块的内存地址。
LONG/DWORD	这种类型比较棘手。返回计数的函数通常会返回一个 LONG 或 DWORD。如果函数出于某种原因不能对你想要计数的东西进行计数，它通常会返回 0 或–1（具体取决于函数）。如果要调用一个返回 LONG/DWORD 的函数，务必仔细阅读 Platform SDK 文档，确保你将正确地检查可能出现的错误。

如果一个 Windows 函数能返回错误代码，通常有助于我们理解函数调用为什么会失败。Microsoft 编辑了一个列表，其中列出了所有可能的错误代码，并为每个错误代码都分配了

一个 32 位的编号。

在内部，当一个 Windows 函数检测到错误时，它会使用一个名为“线程本地存储区”（thread-local storage）的机制将恰当的错误代码与“主调线程”（或者说发出调用的线程，即 calling thread）关联到一起（线程本地存储区的详情将在第 21 章讨论）。这个机制使不同的线程能独立运行，不会出现相互干扰对方的错误代码的情况。函数返回时，其返回值会指出已发生一个错误。要查看具体是什么错误，请调用 GetLastError 函数：

```
DWORD GetLastError();
```

它的作用很简单，就是返回由上一个函数调用设置的线程的 32 位错误代码。

有了 32 位错误代码之后，接着需要把它转换为更有用的信息。WinError.h 头文件包含了 Microsoft 定义的错误代码列表。为便于你体验，下面摘录了其中的一部分：

```
// MessageId: ERROR_SUCCESS
//
// MessageText:
//
// The operation completed successfully.
//
#define ERROR_SUCCESS                          0L
#define NO_ERROR 0L                                                  // dderror
#define SEC_E_OK                               ((HRESULT)0x00000000L)
//
// MessageId: ERROR_INVALID_FUNCTION
//
// MessageText:
//
// Incorrect function.
//
#define ERROR_INVALID_FUNCTION                     1L             // dderror
//
// MessageId: ERROR_FILE_NOT_FOUND
//
// MessageText:
//
// The system cannot find the file specified.
//
#define ERROR_FILE_NOT_FOUND                       2L
//
// MessageId: ERROR_PATH_NOT_FOUND
//
// MessageText:
//
// The system cannot find the path specified.
//
```

```
#define ERROR_PATH_NOT_FOUND                     3L
//
// MessageId: ERROR_TOO_MANY_OPEN_FILES
//
// MessageText:
//
// The system cannot open the file.
//
#define ERROR_TOO_MANY_OPEN_FILES                4L
//
// MessageId: ERROR_ACCESS_DENIED
//
// MessageText:
//
// Access is denied.
//
#define ERROR_ACCESS_DENIED                      5L
```

可以看出，每个错误都有三种表示：一个消息 ID（一个可在源代码中使用的宏，用于与 GetLastError 的返回值进行比较)、消息文本（描述错误的英文文本）和一个编号（应该避免使用此编号，尽量使用消息 ID）。注意，我只摘录了 WinError.h 头文件的极小一部分，整个文件的长度超过 39 000 行！

一个 Windows 函数失败之后，应该马上调用 GetLastError，因为假如又调用了另一个 Windows 函数，则此值很可能被改写。注意，成功调用的 Windows 函数可能用 ERROR_SUCCESS 改写此值。

一些 Windows 函数调用成功可能是缘于不同的原因。例如，创建一个命名的事件内核对象时，以下两种情况均会成功：对象实际地完成创建，或者存在一个同名的事件内核对象。应用程序也许需要知道成功的原因。为返回这种信息，Microsoft 选择采用“上一个错误代码”（last error code）机制。所以，当特定函数成功时，你可以调用 GetLastError 来确定额外的信息。对于具有这种行为的函数，Platform SDK 文档会清楚指明能以这种方式使用 GetLastError。文档中提供了 CreateEvent 函数的一个例子；如果存在命名的事件，它会返回 ERROR_ALREADY_EXISTS。

调试程序时，我发现对线程的“上一个错误代码”进行监视是相当有用的。在 Microsoft Visual Studio 中，Microsoft 的调试器支持一个很有用的功能——你可以配置 Watch 窗口，让它始终显示线程的上一个错误代码和错误的文本描述。具体的做法是：在 Watch 窗口中选择一行，然后输入$err,hr。来看看图 1-1 的例子。在这个例子中，我已经调用了 CreateFile 函数。该函数返回值为 INVALID_HANDLE_VALUE (–1)的一个 HANDLE，指出它无法打开指定文件。但是 Watch 窗口指出，上一个错误代码（也就是调用 GetLastError 函数返回的错误代码）是 0x00000002。多亏有了,hr 限定符，Watch 窗口进一步指出错误代码 2 是“The system cannot find the file specified”（系统找不到指定文件）。这就是在 WinError.h 头文件中为错误代码 2 列出的消息文本。

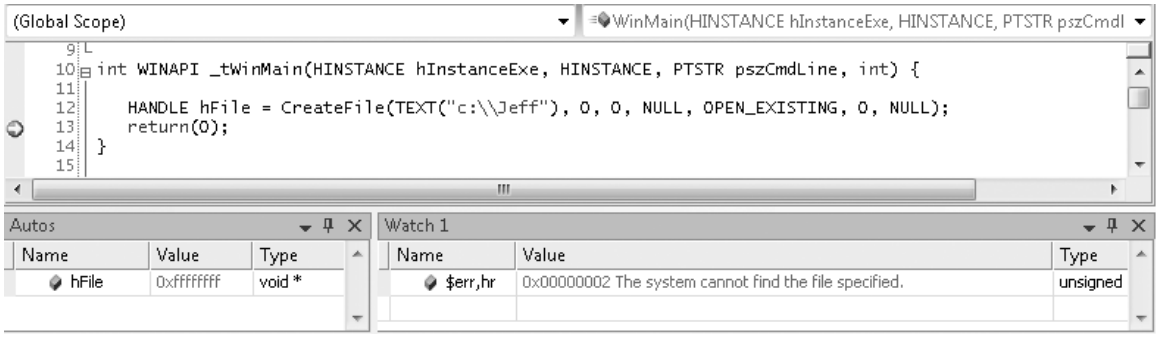
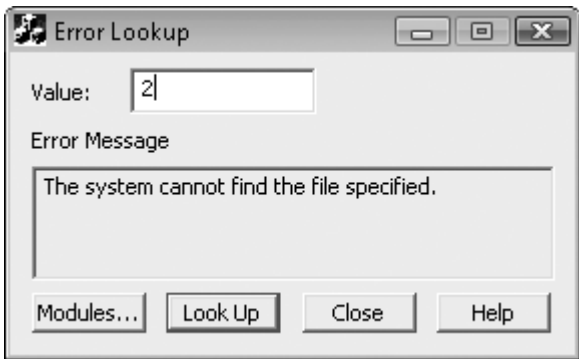


图 1-1　在 Visual Studio 的 Watch 窗口中使用$err,hr 来查看当前线程的“上一个错误代码”

Visual Studio 还搭载了一个很小的实用程序，名为 Error Lookup。利用它，可以将错误代码编号转换为相应的文本描述。如下图所示：



如果我在自己写的程序中检测到一个错误，我可能希望向用户显示文本描述，而不是显示一个干巴巴的错误编号。Windows 提供了一个函数，可以将错误代码转换为相应的文本描述。此函数名为 FormatMessage，如下所示：

```
DWORD FormatMessage(
  DWORD dwFlags,
  LPCVOID pSource,
  DWORD dwMessageId,
  DWORD dwLanguageId,
  PTSTR pszBuffer,
  DWORD nSize,
  va_list *Arguments);
```

FormatMessage 的功能实际相当丰富，为了构造要向用户显示的字符串，它是首选的一种方式。之所以说它好用，一个原因是它能轻松地支持多种语言（译注：这里的语言是自然语言，比如汉语、英语等等，而不是计算机编程语言）。它能获取一个语言标识符作为参数，并返回那种语言的文本。当然，你首先必须翻译好字符串，并将翻译好的消息表（message table）资源嵌入自己的.exe 或 DLL 模块中。但在此之后，这个函数就能自动选择正确的字符串。ErrorShow 示例程序（参见后文）演示了如何调用这个函数将 Microsoft 定义的错误代码编号转换为相应的文本描述。

经常有人问我，Microsoft 是否维护着一个主控列表，其中完整列出了每个 Windows 函数可

能返回的所有错误代码。很遗憾，答案是否定的。而且，Microsoft 决不可能提供这样的列表，因为随着新的操作系统版本的产生，很难构建和维护这样的列表。

这种列表的问题在于，你可以调用一个 Windows 函数，但在内部，这个函数可能调用另一个函数，后者又可能调用其他函数……以此类推。出于众多原因，任何一个函数都可能失败。有时，当一个函数失败时，较高级别的函数也许能够恢复，并继续执行你希望的操作。要创建这种主控列表，Microsoft 必须跟踪每个函数的路径，生成所有可能的错误代码的列表。这是非常难的。而且，随着新版本的操作系统的发布，这些函数的执行路径也可能发生改变。

1.1 定义自己的错误代码

前面讲述了 Windows 函数如何向其调用者指出错误。除此之外，Microsoft 还允许将这种机制用于你自己的函数中。假定你要写一个供其他人调用的函数。这个函数可能会因为这样或那样的原因而失败，所以需要向调用者指出错误。

为了指出错误，只需设置线程的上一个错误代码，然后令自己的函数返回 FALSE，INVALID_HANDLE_VALUE、NULL 或者其他合适的值。为了设置线程的上一个错误代码，只需调用以下函数，并传递你认为合适的任何 32 位值：

```
VOID SetLastError(DWORD dwErrCode);
```

我会尽量使用 WinError.h 中现有的代码——只要代码能很好地反映我想报告的错误。如果 WinError.h 中的任何一个代码都不能准确反映一个错误，就可以创建自己的代码。错误代码是一个 32 位数，由表 1-2 描述的几个不同的字段组成。

表 1-2 错误代码的不同字段

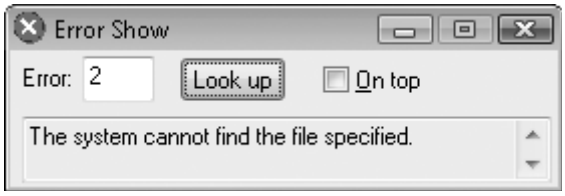
位：	31–30	29	28	27–16	15–0
内容	严重性	Microsoft/ 客户	保留	Facility 代码	异常代码
含义	0 = 成功 1 = 信息(提示) 2 = 警告 3 = 错误	0 = Microsoft 定义的代码 1 = 客户定义的代码	必须为 0	前 256 个值由 Microsoft 保留	Microsoft/ 客户定义的代码

这些字段将在第 24 章详细讨论。就目前来说，惟一需要注意的重要字段在位 29 中。Microsoft 承诺，在它所生成的所有错误代码中，此位将始终为 0。但是，如果要创建你自己的错误代码，就必须在此位放入一个 1。通过这种方式，可以保证你的错误代码绝不会与 Microsoft 现在和将来定义的错误代码冲突。注意，Facility 字段非常大，足以容纳 4096 个可能的值。其中，前 256 个值是为 Microsoft 保留的，其余的值可由你自己的应用程序来定义。

1.2 ErrorShow 示例程序

ErrorShow 应用程序（01-ErrorShow.exe），演示了如何得到一个错误代码的文本描述。此应用程序的源代码和资源文件可以在本书示例代码压缩包的 01-ErrorShow 目录中找到，请访问 *http://wintellect.com/Books.aspx* 来下载。

简单地说，这个应用程序展示了调试器的 Watch 窗口和 Error Lookup 程序是如何工作的（参见前面的两个屏幕截图）。启动程序时，将出现以下窗口。



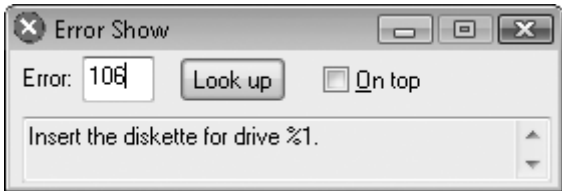
可以在编辑控件中输入任何错误编号。单击 Look Up 按钮后，错误的文本描述将在对话框底部的可滚动窗口中显示。对于这个应用程序，我们惟一感兴趣的是如何调用 FormatMessage。下面展示了我如何使用这个函数：

```
// Get the error code
DWORD dwError = GetDlgItemInt(hwnd, IDC_ERRORCODE, NULL, FALSE);

HLOCAL hlocal = NULL; // Buffer that gets the error message string

// Use the default system locale since we look for Windows messages
// Note: this MAKELANGID combination has a value of 0
DWORD systemLocale = MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL);

// Get the error code's textual description
BOOL fOk = FormatMessage(
  FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS |
  FORMAT_MESSAGE_ALLOCATE_BUFFER,
  NULL, dwError, systemLocale,
  (PTSTR) &hlocal, 0, NULL);

if (!fOk) {
  // Is it a network-related error?
  HMODULE hDll = LoadLibraryEx(TEXT("netmsg.dll"), NULL,
    DONT_RESOLVE_DLL_REFERENCES);
  if (hDll != NULL) {
    fOk = FormatMessage(
      FORMAT_MESSAGE_FROM_HMODULE | FORMAT_MESSAGE_IGNORE_INSERTS |
      FORMAT_MESSAGE_ALLOCATE_BUFFER,
      hDll, dwError, systemLocale,
```

```
        (PTSTR) &hlocal, 0, NULL);
      FreeLibrary(hDll);
   }
}

if (fOk && (hlocal != NULL)) {
   SetDlgItemText(hwnd, IDC_ERRORTEXT, (PCTSTR) LocalLock(hlocal));
   LocalFree(hlocal);
} else {
   SetDlgItemText(hwnd, IDC_ERRORTEXT,
     TEXT("No text found for this error number."));
```

第一行从编辑控件获取错误代码。然后，指向一个内存块的句柄被实例化并初始化为 NULL。FormatMessage 函数在内部分配内存块，并返回指向这个内存块的句柄。

调用 FormatMessage 时，我们向它传入了 FORMAT_MESSAGE_FROM_SYSTEM 标志。该标志告诉 FormatMessage：我们希望获得与一个系统定义的错误代码对应的字符串。另外，还传入了 FORMAT_MESSAGE_ALLOCATE_BUFFER 标志，要求该函数分配一个足以容纳错误文本描述的内存块。此内存块的句柄将在 hlocal 变量中返回。FORMAT_MESSAGE_IGNORE_INSERTS 标志则允许你获得含有%占位符的消息。Windows 利用它来提供上下文相关的信息，如下例所示：



如果不传递这个标志，就必须为 Arguments 参数中的这些占位符提供具体的值。但这对于 Error Show 程序来说是不可能的，因为消息的内容事先是未知的。

第三个参数指出想要查找的错误编号。第四个参数指出要用什么语言来显示文本描述。由于我们对 Windows 自己提供的消息感兴趣，所以语言标识符将根据两个特定的常量（即 LANG_NEUTRAL 和 SUBLANG_NEUTRAL）来生成，这两个常量联合到一起将生成一个 0 值——意味着操作系统的默认语言。这种情况下，我们不能硬编码一种特定的语言，因为事先并不知道操作系统的安装语言是什么。

如果 FormatMessage 成功，文本描述就在内存块中，我把它拷贝到对话框底部的可滚动窗口中。如果 FormatMessage 失败，我会尝试在 NetMsg.dll 模块中查找消息代码，看错误是否与网络有关（有关如何在磁盘上搜索 DLL 的详情，请参见第 20 章）。利用 NetMsg.dll 模块的句柄，我再一次调用 FormatMessage。我们知道，每个 DLL（或.exe）都可以有自己的一套错误代码。可以使用 Message Compiler（MC.exe）把这些代码添加到 DLL（或.exe）模块中，并在模块中添加一个资源。Visual Studio 的 Error Lookup 工具允许你使用 Modules 对话框来完成这个操作。

第 2 章　字符和字符串处理

本章内容

- 字符编码
- ANSI和Unicode字符和字符串数据类型
- Windows中的Unicode和ANSI函数
- C运行库中的Unicode和ANSI函数
- C运行库中的安全字符串函数
- 为何要用Unicode
- 推荐的字符和字符串处理方式
- Unicode和ANSI字符串转换

随着Microsoft Windows在世界各地日渐流行，作为软件开发人员，将眼光投向全球市场显得越发重要。美国版本的软件在发布时间上比国际版本早6个月，这样的事情一度屡见不鲜。但是，随着操作系统的国际化支持日益增强，为国际市场发布软件产品变得越来越容易，美国版本和国际化版本的软件在发布时间上的间隔变得越来越短。

Windows一如继往地为开发人员提供支持，帮助他们本地化自己的应用程序。应用程序可以通过多个函数来获得一个国家特有的信息，并能检查控制面板的设置来判断用户当前的首选项。Windows甚至能为应用程序支持不同的字体。最后一点也是非常重要的一点，Windows Vista开始提供对Unicode 5.0的支持（详情参见“Extend The Global Reach Of Your Applications With Unicode 5.0”一文，网址 为 *http://msdn.microsoft.com/msdnmag/issues/07/01/Unicode/default.aspx*）。

缓冲区溢出错误（这是处理字符串时的典型错误）已成为针对应用程序乃至操作系统的各个组件发起安全攻击的媒介。这几年，Microsoft从内部和外部两个方面主动出击，倾尽全力提升Windows世界的安全水平。本章的第二部分将介绍Microsoft在C运行库中新增的函数。你应该使用这些新函数来防止应用程序在处理字符串时发生缓冲区溢出。

本章的位置之所以如此靠前，是由于我极力主张你的应用程序始终使用Unicode字符串，而且始终应该通过新的安全字符串函数来处理这些字符串。如你所见，与如何安全使用Unicode字符串相关的问题在本书的每一章和每一个示例程序中都有涉及。如果有一个非Unicode的代码库，最好能将此代码库转移至Unicode，这会增强应用程序的执行性能，并为本地化工作奠定基础。另外，它还有利于同COM和.NET Framework的互操作。

2.1　字符编码

本地化的问题就是处理不同字符集的问题。多年来，我们一直在将文本字符串编码成一组以0结尾的单字节字符。许多人对此已经习已为常。调用strlen，它会返回“以0结尾的一个ANSI单字节字符数组”中的字符数。

问题是，某些语言和书写系统（例如日本汉字）的字符集有非常多的符号。一个字节最多只能表示256个符号，因此是远远不够的。为了支持这些语言和书写系统，双字节字符集（double-byte character set，DBCS）应运而生。在双字节字符集中，一个字符串中的每个字符都由1个或2个字节组成。以日本汉字为例，如果第一个字符在0x81到0x9F之间，或者在0xE0到0xFC之间，就必须检查下一个字节，才能判断出一个完整的汉字。对程序员而言，和双字节字符集打交道如同一场噩梦，因为某些字符是1个字节宽，而有的字符却是2个字节宽。幸运的是，我们可以把DBCS放到一边，专心利用Windows函数和C运行库对Unicode字符串的支持。

Unicode是1988年由Apple和Xerox共同建立的一项标准。1991年，成立了专门的协会来开发和推动Unicode。该协会由Apple、Compaq、Hewlett-Packard、IBM、Microsoft、Oracle、Silicon Graphics、Sybase、Unisys和Xerox等多家公司组成（协会成员的最新列表可从*http://www.Unicode.org*获得）。该组织负责维护Unicode标准。Unicode的完整描述可以参考Addison-Wesley出版的*The Unicode Standard*一书，该书可通过*http://www.Unicode.org*获得。

在Windows Vista中，每个Unicode字符都使用UTF-16编码，UTF的全称是Unicode Transformation Format（ Unicode转换格式）。UTF-16将每个字符编码为2个字节（或者说16位）。在本书中，我们在谈到Unicode时，除非专门声明，否则一般都是指UTF-16编码。Windows之所以使用UTF-16，是因为全球各地使用的大部分语言中，每个字符很容易用一个16位值来表示。这样一来，应用程序很容易遍历字符串并计算出它的长度。但是，16位不足以表示某些语言的所有字符。对于这些语言，UTF-16支持使用代理（surrogates），后者是用32位（或者说4个字节）来表示一个字符的一种方式。由于只有少数应用程序需要表示这些语言中的字符，所以UTF-16在节省空间和简化编码这两个目标之间，提供了一个很好的折衷。注意，.NET Framework始终使用UTF-16来编码所有字符和字符串，所以在你自己的Windows应用程序中，如果需要在原生代码（native code）和托管代码（managed code）之间传递字符或字符串，使用UTF-16能改进性能和减少内存消耗。

另外还有其他用于表示字符的UTF标准，具体如下。

- **UTF-8** UTF-8将一些字符编码为1个字节，一些字符编码为2个字节，一些字符编码为3个字节，一些字符编码为4个字节。值在0x0080以下的字符压缩为1个字节，这对美国使用的字符非常适合。0x0080和0x07FF之间的字符转换为2个字节，这对欧洲和中东地区的语言非常适用。0x0800以上的字符都转换为3个字节，适合东亚地区的语言。最后，代理对（surrogate pairs）被写为4个字节。UTF-8是一种相当流行的编码格式。但在对值为0x0800及以上的大量字符进行编码的时候，不如UTF-16高效。
- **UTF-32** UTF-32将每个字符都编码为4个字节。如果打算写一个简单的算法来遍历字符（任何语言中使用的字符），但又不想处理字节数不定的字符，这种编码方式就非常有用。例如，如果采用UTF-32编码方式，就不需要关心代理（surrogate）的问题，因为每个字符都是4个字符。显然，从内存使用这个角度来看，UTF-32 并不是一种高效的编码格式。因此，很少用它将字符串保存到文件或传送到网络。这种编码格式一般在应用程序内部使用。

目前，Unicode为阿拉伯语、汉语拼音、西里尔文（俄语）、希腊语、希伯来语、日语假名、朝鲜语和拉丁语（英语）字符等——这些字符称为书写符号（scripts）——定义了码位（code

point，即一个符号在字符集中的位置）。每个版本的Unicode都在现有的书写符号的基础上引入了新的字符，甚至会引入新的书写符号，比如腓尼基文（一种古地中海文字）。字符集中还包含大量标点符号、数学符号、技术符号、箭头、装饰标志、读音符号以及其他字符。

这65 536个字符被划分为若干个区域，表2-1展示了部分区域以及分配到这些区域的字符。

表2-1　Unicode字符集和字母表（译者注：部分译法借鉴了chukl000的“統一碼 5.0.0 版區塊名稱表”）

6 位代码	字符	16 位代码	字母/书写符号
0000–007F	ASCII	0300–036F	常见的变音符号
0080–00FF	拉丁字母补充-1	0400–04FF	西里尔字母
0100–017F	欧洲拉丁字母	0530–058F	亚美尼亚文
0180–01FF	拉丁字母扩充	0590–05FF	希伯来文
0250–02AF	国际音标扩充	0600–06FF	阿拉伯文
02B0–02FF	进格修饰字母	0900–097F	梵文字母

2.2　ANSI 字符和 Unicode 字符与字符串数据类型

你知道，C语言用**char**数据类型来表示一个8位ANSI字符。默认情况下，在源代码中声明一个字符串时，C编译器会把字符串中的字符转换成由8位**char**数据类型构成的一个数组：

```
// 一个 8 位字符
char c = 'A';
// 一个数组，包含 99 个 8 位字符以及一个 8 位的终止 0
char szBuffer[100] = "A String";
```

Microsoft的C/C++编译器定义了一个内建的数据类型**wchar_t**，它表示一个16位的Unicode（UTF-16） 字符。因为早期版本的Microsoft编译器没有提供这个内建的数据类型，所以编译器只有在指定了**/Zc:wchar_t**编译器开关时，才会定义这个数据类型。默认情况下，在Microsoft Visual Studio 中新建一个C++项目时，这个编译器开关是指定的。建议始终指定这个编译器开关，这样才能借助于编译器天生就能理解的内建基元类型来更好地操纵Unicode字符。

注意　在编译器内建对wchar_t的支持之前，有一个C头文件定义了一个wchar_t数据类型：

```
typedef unsigned short wchar_t;
```

声明Unicode字符和字符串的方法如下所示：

```
// 一个 16 位字符
```

```
wchar_t c = L'A';

// 一个数组，包含最多 99 个 16 位字符以及一个 16 位的终止 0
wchar_t szBuffer[100] = L"A String";
```

字符串之前的大写字母**L**通知编译器该字符串应当编译为一个Unicode字符串。当编译器将此字符串放入程序的数据段时，会使用UTF16来编码每个字符。在这个简单的例子中，在每个ASCII字符之间都用一个0来间隔。

为了与C语言稍微有一些隔离，Microsoft Windows团队希望定义自己的数据类型。于是，在Windows头文件WinNT.h中，定义了以下数据类型：

```
typedef char CHAR; // An 8-bit character
typedef wchar_t WCHAR; // A 16-bit character
```

除此之外，WinNT.h头文件还定义了一系列能为你提供大量便利的数据类型，可以用它处理字符和字符串指针：

```
// Pointer to 8-bit character(s)
typedef CHAR *PCHAR;
typedef CHAR *PSTR;
typedef CONST CHAR *PCSTR

// Pointer to 16-bit character(s)
typedef WCHAR *PWCHAR;
typedef WCHAR *PWSTR;
typedef CONST WCHAR *PCWSTR;
```

> 注意
> 仔细查看WinNT.h，会看到如下定义:
>
> typedef __nullterminated WCHAR *NWPSTR, *LPWSTR, *PWSTR;
>
> 前缀__nullterminated是一个头部注解（header annotation），它描述一个类型如何作为函数的参数和返回值使用。在Visual Studio企业版中，可以在项目属性中设置代码分析(Code Analysis)选项。这样会把/analyze开关添加到编译器的命令行中。这样一来，假如你的代码调用函数的方式违反了头部注解所定义的语义，编译器就会检测到这类问题。注意，只有编译器的企业版才支持这个/analyze开关。为保证本书所提供的代码的可读性，所有header annotation都已被删除。要想更多地了解header annotation语言，请参考MSDN文档“Header Annotations”，网址是*http://msdn2.microsoft.com/En-US/library/aa383701.aspx*。

在源代码中，具体使用哪种数据类型并不重要，但建议你尽量保持一致，以增强代码的可维护性。就我个人而言，作为Windows程序员，我会坚持使用Windows数据类型，因为这些数

据类型与MSDN文档相符，有利于增强代码的可读性。

另外，你在写代码的时候，可以让它使用ANSI或Unicode字符/字符串都能通过编译。WinNT.h定义了以下类型和宏：

```
#ifdef UNICODE

typedef WCHAR TCHAR, *PTCHAR, PTSTR;
typedef CONST WCHAR *PCTSTR;
#define __TEXT(quote) quote // r_winnt

#define __TEXT(quote) L##quote

#else

typedef CHAR TCHAR, *PTCHAR, PTSTR;
typedef CONST CHAR *PCTSTR;
#define __TEXT(quote) quote

#endif

#define TEXT(quote) __TEXT(quote)
```

利用这些类型和宏（少数不太常用的没有在这里列出）来写代码，无论使用ANSI还是Unicode字符，它都能通过编译。例如：

```
// 如果定义了UNICODE，就是一个16位字符；否则就是一个8位字符
TCHAR c = TEXT('A');

// 如果定义了UNICODE，就是由16位字符构成的一个数组；否则就是8位字符的一个数组
TCHAR szBuffer[100] = TEXT("A String");
```

2.3 Windows 中的 Unicode 和 ANSI 函数

自Windows NT起，Windows的所有版本都完全用Unicode来构建。也就是说，所有核心函数（创建窗口、显示文本、进行字符串处理等等）都需要Unicode字符串。调用一个Windows函数时，如果向它传入一个ANSI字符串（由单字节字符组成的一个字符串），函数首先会把字符串转换为Unicode，再把结果传给操作系统。如果希望函数返回ANSI字符串，那么操作系统会先把Unicode字符串转换为ANSI字符串，再把结果返回给你的应用程序。所有这些转换都是悄悄地进行的。当然，为了执行这些字符串转换，系统会产生时间和内存上的开销。

如果一个Windows函数需要获取一个字符串作为参数，则该函数通常有两个版本。例如，一个CreateWindowEx接受Unicode字符串，另一个CreateWindowEx则接受ANSI字符串。这没错，但两个函数的原型实际是这样的：

```
HWND WINAPI CreateWindowExW(
  DWORD dwExStyle,
  PCWSTR pClassName,  // A Unicode string
  PCWSTR pWindowName, // A Unicode string
  DWORD dwStyle,
  int X,
  int Y,
  int nWidth,
  int nHeight,
  HWND hWndParent,
  HMENU hMenu,
  HINSTANCE hInstance,
  PVOID pParam);

HWND WINAPI CreateWindowExA(
  DWORD dwExStyle,
  PCSTR pClassName,  // An ANSI string
  PCSTR pWindowName, // An ANSI string
  DWORD dwStyle,
  int X,
  int Y,
  int nWidth,
  int nHeight,
  HWND hWndParent,
  HMENU hMenu,
  HINSTANCE hInstance,
  PVOID pParam);
```

CreateWindowExW这个版本接受Unicode字符串。函数名末尾的大写字母*W*代表wide。Unicode字符都是16位宽，所以它们常常被称作宽（wide）字符。**CreateWindowExA** 末尾的大写字母*A*表明该函数接受ANSI字符串。

但在平时，我们只是在自己的代码中调用**CreateWindowEx**，不会直接调用**CreateWindowExW**或**CreateWindowExA**。在WinUser.h中，**CreateWindowEx**实际是一个宏，它的定义如下：

```
#ifdef UNICODE
#define CreateWindowEx CreateWindowExW
#else
#define CreateWindowEx CreateWindowExA
#endif
```

编译源代码模块时，是否定义**UNICODE**决定了要调用哪一个版本的**CreateWindowEx**。用Visual Studio创建一个新项目的时候，它默认会定义**UNICODE**。所以，在默认情况下，对

CreateWindowEx的任何调用都会扩展宏来调用**CreateWindowExW**——即Unicode版本的**CreateWindowEx**。

在Windows Vista中，**CreateWindowExA**的源代码只是一个转换层（translation layer），它负责分配内存，以便将ANSI字符串转换为Unicode字符串；然后，代码会调用**CreateWindowExW**，并向它传递转换后的字符串。**CreateWindowExW**返回时，**CreateWindowExA**会释放它的内存缓冲区，并将窗口句柄返回给你。所以，对于要在缓冲区中填充字符串的任何函数，在你的应用程序能够处理字符串之前，系统必须先将Unicode转换为非Unicode的等价物。由于系统必须执行所有这些转换，所以应用程序需要更多内存，而且运行速度较慢。为了提高使应用程序的执行更高效，你应该一开始就用Unicode来开发程序。另外，目前已知Windows的这些转换函数中存在一些缺陷，所以避免使用它们，还有助于消除一些潜在的bug。

如果是在创建供其他软件开发人员使用的动态链接库（dynamic-link library，DLL），可考虑使用这种技术：在DLL中提供导出的两个函数，一个ANSI版本的，一个Unicode版本的。在ANSI版本中，只是分配内存，执行必要的字符串转换，然后调用该函数的Unicode版本。我将在本章的2.8.1节“导出ANSI和Unicode DLL函数”演示这个过程。

Windows API中的一些函数（比如**WinExec**和**OpenFile**）存在的惟一目的就是提供与只支持ANSI字符串的16位Windows程序的向后兼容性。在新程序中，应避免使用这些方法。在使用**WinExec**和**OpenFile**调用的地方，应该用**CreateProcess**和**CreateFile**函数调用来代替。在内部，老函数总是会调用新函数。但老函数的最大问题在于，它们不接受Unicode字符串，而且支持的功能一般都要少一些。调用这些函数的时候，必须向其传递ANSI字符串。在Windows Vista中，大部分尚未废弃的函数都有Unicode和ANSI两个版本。然而，Microsoft逐渐开始倾向于某些函数只提供Unicode版本，比如**ReadDirectoryChangesW**和**CreateProcessWithLogonW**。

Microsoft将COM从16位Windows移植到Win32时，做出了一个重要决策：所有需要字符串作为参数的COM接口方法都只接受Unicode字符串。这是一个伟大的决策，因为COM一般用于让不同的组件彼此间进行“对话”，而Unicode是传递字符串最理想的选择。在你的应用程序中全面使用Unicode，可以使它与COM的交互变得更容易。

最后，当资源编译器编译完所有资源后，输出文件就是资源的一个二进制形式。资源中的字符串值（字符串表、对话框模板、菜单等等）始终都写成Unicode字符串。在Windows Vista中，如果你的应用程序没有定义**UNICODE**宏，操作系统将执行内部转换。例如，在编译源模块时，如果没有定义**UNICODE**，那么对**LoadString**的调用实际会调用**LoadStringA**函数。然后，**LoadStringA**读取资源中的Unicode字符串，并把它转换成ANSI形式。最后，转换为ANSI形式的字符串将从函数返回到应用程序。

2.4 C 运行库中的 Unicode 函数和 ANSI 函数

和Windows函数一样，C运行库提供了一系列函数来处理ANSI字符和字符串，并提供了另一系列函数来处理Unicode字符与字符串。然而，与Windows不同的是，ANSI版本的函数是“自力更生”的：它们不会把字符串转换为Unicode形式，再从内部调用函数的Unicode版本。当

然，Unicode版本的函数也是“自力更生”的，它们不会在内部调用ANSI版本。

在C运行库中，能返回ANSI字符串长度的一个函数的例子是**strlen**。与之对应的是**wcslen**，这个C运行库函数能返回Unicode字符串的长度。

这两个函数的原型都在String.h中。为了使你的源代码针对ANSI或Unicode都能编译，那么还必须包含TChar.h，该文件定义了以下宏：

```
#ifdef _UNICODE
#define _tcslen wcslen
#else
#define _tcslen strlen
#endif
```

现在，在你的代码中应该调用**_tcslen**。如果已经定义了**_UNICODE**，它会扩展为**wcslen**；否则，它会扩展为**strlen**。默认情况下，在Visual Studio中新建一个C++项目时，已经定义了**_UNICODE**（就像已经定义了**UNICODE**一样）。针对不属于C++标准一部分的标识符，C运行库始终为其附加下划线前缀。但是，Windows团队没有这样做。所以，在你的应用程序中，应确保要么同时定义了**UNICODE**和**_UNICODE**，要么一个都不要定义。附录A将详细描述CmnHdr.h；本书所有示例代码都将用这个头文件来避免这种问题。

2.5 C 运行库中的安全字符串函数

任何修改字符串的函数都存在一个安全隐患：如果目标字符串缓冲区不够大，无法包含所生成的字符串，就会破坏内存中的数据（或者说发生“内存恶化”，即memory corruption）。下面是一个例子：

```
// The following puts 4 characters in a
// 3-character buffer, resulting in memory corruption
WCHAR szBuffer[3] = L"";
wcscpy(szBuffer, L"abc"); // The terminating 0 is a character too!
```

strcpy和**wcscpy**函数（以及其他大多数字符串处理函数）的问题在于，它们不能接受指定了缓冲区最大长度的一个参数。所以，函数不知道自己会破坏内存。因为不知道会破坏内存，所以不会向你的代码报告错误。因为不知道出错，所以你不知道内存已经被破坏。另外，如果函数只是简单地失败，而不是破坏任何内存，那么是最理想不过的了。

过去，这种行为被恶意软件肆意滥用。现在，Microsoft提供了一系列新的函数来取代C运行库的不安全的字符串处理函数（比如**wcscat**）。虽然多年以来，这些函数已经成为许多开发人员的老朋友，但为了写安全的代码，你应该放弃这些熟悉的、能修改字符串的C运行库函数（不过，**strlen**、**wcslen**和**_tcslen**等函数是没有问题的，因为它们不会修改传入的字符——即使它们假设字符串是以0来终止的，而这个假设有时并不一定成立）。相反，应该使用在Microsoft的StrSafe.h文件中定义新的安全字符串函数。

注意
Microsoft已在内部更新了ATL和MFC类库，以使用新的安全字符串函数。如果你的程序使用了这些库，只需rebuild一下，就能让你的程序变得更安全。

因为本书不是专门讨论C/C++编程的，所以要想深入了解这个库的用法，推荐你参考以下信息来源：

- MSDN Magazine的一篇文章，题为“Repel Attacks on Your Code with the Visual Studio 2005 Safe C and C++ Libraries”，作者是Martyn Lovell，网址是：*http://msdn.microsoft.com/msdnmag/issues/05/05/SafeCandC/default.aspx*
- Channel9的Martyn Lovell视频演示，网址是：*http://channel9.msdn.com/Showpost.aspx?postid=186406*
- MSDN Online的有关安全字符串的主题，网址是：*http://msdn2.microsoft.com/en-us/library/ms647466.aspx*
- MSDN Online上的所有C运行时安全替代函数的列表，网址是：*http://msdn2.microsoft.com/en-us/library/wd3wzwts(VS.80).aspx*

不过，有一些细节值得在本章进行探讨。首先要讨论新函数采用的模式。然后要谈谈从遗留函数迁移到对应的安全版本时（比如使用**_tcscpy_s**来代替**_tcscpy**），你可能会遇到的一些问题。最后谈谈在哪种情况下应该调用新的**StringC***函数。

2.4.1 初识新的安全字符串函数

在应用程序中包含StrSafe.h时，String.h也会包含进来。C运行库中现有的字符串处理函数，比如**_tcscpy**宏背后的那些函数，已标记为废弃不用。如果使用了这些函数，编译时就会发出警告。注意，必须在包含了其他所有文件之后，才包含StrSafe.h。我建议你利用编译警告来显式地用安全版本来替换废弃不用的所有函数——每一次替换的时候，都考虑一下是否可能发生缓冲区溢出；另外，如果不可能从错误中恢复，至少应该考虑如何优雅地终止应用程序。

现有的每一个函数，比如**_tcscpy**或**_tcscat**，都有一个对应的新版本的函数。前面的名称相同，但最后添加了一个**_s**（代表secure）后缀 。所有这些新函数都有一个共同的特征，这有待我们进一步解释。首先，让我们根据以下代码片段来研究一下它们的原型。在以下代码片段中，展示了两个普通的字符串函数的定义：

```
PTSTR _tcscpy (PTSTR strDestination, PCTSTR strSource);
errno_t _tcscpy_s(PTSTR strDestination, size_t numberOfCharacters,
   PCTSTR strSource);

PTSTR _tcscat (PTSTR strDestination, PCTSTR strSource);
errno_t _tcscat_s(PTSTR strDestination, size_t numberOfcharacters,
   PCTSTR strSource);
```

一个可写的缓冲区作为参数传递时，必须同时提供它的大小。这个值应该是一个字符数。为

你的缓冲区使用**_countof** 宏（在stdlib.h中定义），很容易计算出这个值。

所有安全（后缀为**_s**）函数的首要任务是验证传给它们的参数值。要检查的项目包括指针不为NULL，整数在有效范围内，枚举值是有效的，而且缓冲区足以容纳结果数据。这些检查中的任何一项失败，函数都会设置局部于线程的C运行时变量**errno**。然后，函数会返回一个**errno_t**值来指出成功或失败。然而，这些函数并不实际地返回。相反，如果是一次debug buld，它们会显示如图2-1所示的一个对用户不太友好的Debug Assertion Failed对话框。然后，应用程序会终止。在release build中，则会直接自动终止。

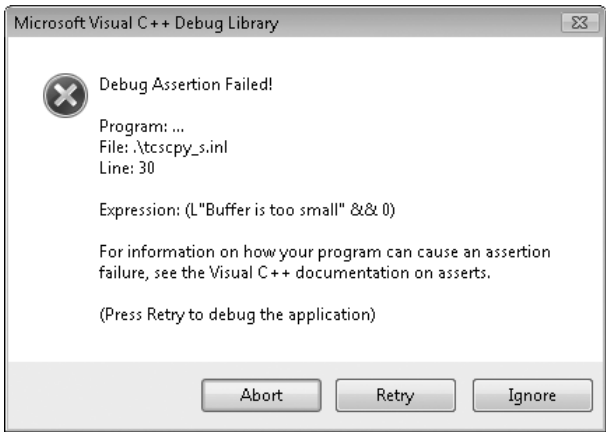


图2-1　遇到错误时显示的Debug Assertion Failed对话框

C运行时实际上允许你提供自己的函数，在它检测到一个无效参数时，将调用此函数。然后，在这个函数中，你可以记录失败，连接一个调试器，或者做其他你想做的其他事情。为了启用这个功能，必须先定义好一个函数，其原型如下：

```
void InvalidParameterHandler(PCTSTR expression, PCTSTR function,
     PCTSTR file, unsigned int line, uintptr_t /*pReserved*/);
```

其中，参数**expression**描述了C运行时实现代码中可能出现的函数调用失败，比如**(L"Buffer is too small" && 0)**。 可以看出，这种方式也不是用户友好的，不应该向最终用户显示。后面三个参数同样如此，因为**function**，**file**和**line**分别描述了出现了错误的函数名称、源代码文件和源代码行号。

> 注意
> 如果没有定义DEBUG，所有这些参数的值都将为NULL。因此，只有在测试debug build时，才适合用这个handler来记录错误。在release build中，应该用一条对用户更友好的消息来替换Debug Assertion Failed对话框，指出由于发生了非预期的错误，所以应用程序将被迫关闭——也许还可以将错误记入日志，或者重新启动应用程序。如果应用程序的内存状态被破坏，就应该停止执行应用程序。不过，最好还是等检查了errno_t之后，再判断是否能从错误中恢复。

下一步是调用**_set_invalid_parameter_handler**来注册这个handler。然而，仅仅这一步是不够的，因为Debug Assertion Failed对话框仍会出现。你要在应用程序开头的地方调用

_CrtSetReportMode(_CRT_ASSERT, 0);，禁止可能由C运行时触发的所有Debug Assertion Failed对话框。

现在，在调用String.h中定义的一个遗留替代函数时，就可以检查返回的**errno_t**值，了解发生了什么事情。只有返回**S_OK**值，才表明函数调用是成功的。其他可能的返回值在errno.h中有定义；例如，**EINVAL**指出你传递了无效的参数值（比如NULL指针）。

下面以一个字符串为例，我们要把它拷贝到一个缓冲区中，但这个缓冲区的长度刚好就小了一个字符：

```
TCHAR szBefore[5] = {
TEXT('B'), TEXT('B'), TEXT('B'), TEXT('B'), '\0'
};
TCHAR szBuffer[10] = {
TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'),
TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'), '\0'
};
TCHAR szAfter[5] = {
TEXT('A'), TEXT('A'), TEXT('A'), TEXT('A'), '\0'
};
errno_t result = _tcscpy_s(szBuffer, _countof(szBuffer),
TEXT("0123456789"));
```

调用**_tcscpy_s**前，每个变量都有如图2-2所示的内容。

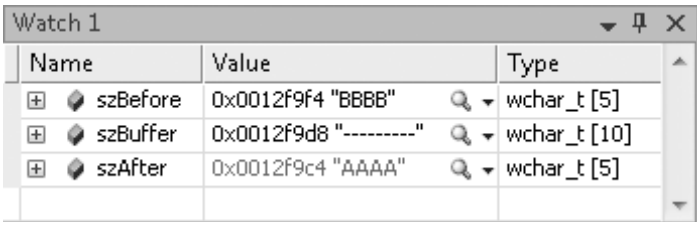


图2-2　调用_tcscpy_s之前变量的状态

将字符串"1234567890"拷贝到szBuffer，后者正好10个字符长，所以没有足够的空间来复制最后的终止字符**'\0'**。你也许认为**result**的值现在为**STRUNCATE**，最后一个字符**'9'**不被拷贝，但实情并非如此。返回的结果是**ERANGE**，每个变量的状态如图2-3所示。

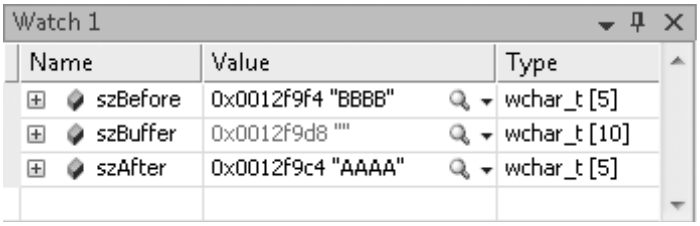


图2-3　调用_tcscpy_s之后变量的状态

这里有一个副作用，不查看如图2-4所示的**szBuffer**后的内存，我们是看不出来的。

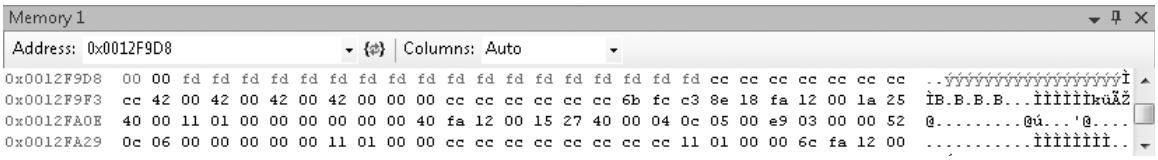


图2-4　函数调用失败之后szBuffer内存中的内容

szBuffer的第一个字符被设为'**\0**'，其他所有字节现在都包含值**0xfd**。因此，最终的字符串被截断为一个空字符串，缓冲区剩余的字节被设为一个填充符（**0xfd**）。

> 注意
> 要知道在图2-4中，为什么在所有已定义的变量后面，内存会用0xcc这个值来填充吗？答案是，这是编译器执行运行时检查（/RTCs，/RTCu或/RTC1）的结果，它们会在运行时自动检测缓冲区溢出。如果不用这些/RTC*x*标志来编译代码，在内存视图中，就会一个接一个地显示所有sz*变量。但要记住，在你进行build时，应该始终指示编译器执行运行时检查，这样才能在开发周期内，尽早检测到任何存在的缓冲区溢出。

2.4.2　字符串处理时如何获得更多控制

除了新的安全字符串函数，C运行库还新增了一些函数，用于在执行字符串处理时提供更多控制。例如，你可以控制填充符，或者指定如何进行截断。自然，C运行库同时为这些函数提供了ANSI（A）版本和Unicode（W）版本。其中部分函数的原型如下（还存在更多类似的函数，在此没有列出）：

```
HRESULT StringCchCat(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc);
HRESULT StringCchCatEx(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc,
   PTSTR *ppszDestEnd, size_t *pcchRemaining, DWORD dwFlags);

HRESULT StringCchCopy(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc);
HRESULT StringCchCopyEx(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc,
   PTSTR *ppszDestEnd, size_t *pcchRemaining, DWORD dwFlags);

HRESULT StringCchPrintf(PTSTR pszDest, size_t cchDest,
   PCTSTR pszFormat, ...);
HRESULT StringCchPrintfEx(PTSTR pszDest, size_t cchDest,
   PTSTR *ppszDestEnd, size_t *pcchRemaining, DWORD dwFlags,
   PCTSTR pszFormat,...);
```

可以看出，在所有方法的名称中，都含有一个“Cch”。这表示Count of characters，即字符数；通常使用**_countof**宏来获取此值。另外还有一系列名称中含有“Cb”的函数，比如**StringCbCat(Ex)**，**StringCbCopy(Ex)**和**StringCbPrintf(Ex)**。这些函数要求用字节数来指定大小，而不是用字符数；通常使用**sizeof**操作符来获取此值。

所有这些函数返回一个**HRESULT**，具体的值如表2-2所示。

表**2-2　安全字符串函数的HRESULT值**

HRESULT 值	**描述**
S_OK	成功。目标缓冲区中包含源字符串，并以**'\0'**终止
STRSAFE_E_INVALID_PARAMETER	失败。将 NULL 值传给了一个参数
STRSAFE_E_INSUFFICIENT_BUFFER	失败。指定目标缓冲区过小，放不下整个源字符串

不同于安全（后缀为**_s**）的函数)，当缓冲区过小的时候，这些函数会执行截断。为了判断是否发生这种情况，你可以检测是否返回了**STRSAFE_E_INSUFFICIENT_BUFFER**。从StrSafe.h中可以看出，此代码的值是**0x8007007a**，被**SUCCEEDED/FAILED**宏定义成一个失败。然而，在这种情况下，源缓冲区中可以装入目标可写缓冲区中的那一部分会被拷贝，而且最后一个可用的字符会被设为**'\0'**。所以，在前面的例子中，如果用**StringCchCopy**来替代**_tcscpy_s**，那么**szBuffer**将包含字符串"012345678"。注意，“截断”这个功能可能是、也可能不是你希望的，具体取决于你想达到的目标。这是为什么它会被视为失败的原因（默认情况下）。例如，如果想连接两个字符串来生成一个路径，那么一个截断的结果是没有用处的。相反，如果是在生成一条用于用户反馈的消息，这或许就是能够接受的。如何处理一个截断的结果，完全由你自己来决定。

最后但同时也是很重要的一点，前面出现的许多函数都存在一个扩展（Ex）版本。这些扩展版本有三个额外的参数，详见表2-3。

表2-3　扩展版本的参数

参数与值	**描述**
size_t* pcchRemaining	指向一个变量的指针，该变量表示目标缓冲区中还有多少字符尚未使用。拷贝的终止字符**'\0'**不计算在内。例如，如果一个字符被拷贝到一个 10 字符长的缓冲区中，那么返回的结果会是 9——虽然在不截断的情况下，你最多只能使用 8 个字符。如果 **pcchRemaining** 为 **NULL**，就不返回计数
LPTSTR* ppszDestEnd	如果 ppszDestEnd 不为 NULL，它将指向终止字符**'\0'**，该字符位于目标缓冲区所包含的那个字符串的末尾
DWORD dwFlags	一个或多个由“|”分隔的以下值：
STRSAFE_FILL_BEHIND_NULL	如果函数成功，dwFlags 的低字节用于填充目标缓冲区的剩余部分（也就是终止字符'\0'之后的部分）。欲知详情，请参见此表之后对 **STRSAFE_FILL_BYTE** 的讨论。
STRSAFE_IGNORE_NULLS	把 NULL 字符串指针视为空字符串**(TEXT(""))**
STRSAFE_FILL_ON_FAILURE	如果函数失败，**dwFlags** 的低字节将用于填充整个目标缓冲区，但第一个字符被设为**'\0'**，从而确保结果是一个空字

参数与值	描述
	符串。欲知详情，请参见此表之后对**STRSAFE_FILL_BYTE** 的讨论。如果是一次**STRSAFE_E_INSUFFICIENT_BUFFER** 失败，在返回的字符串中，所有字符都会被替换成填充符。
STRSAFE_NULL_ON_FAILURE	如果函数失败，目标缓冲区的第一个字符会设为'\0'，从而定义一个空字符串**(TEXT(""))**。如果是一次**STRSAFE_E_INSUFFICIENT_BUFFER** 失败，所有截断的字符串都会被覆盖
STRSAFE_NO_TRUNCATION	和 **STRSAFE_NULL_ON_FAILURE** 的情况一样，如果函数失败，目标缓冲区会被设为空字符串**(TEXT(""))**。在**STRSAFE_E_INSUFFICIENT_BUFFER** 失败，所有截断的字符串都会被覆盖

注意
即使指定了**STRSAFE_NO_TRUNCATION**标志，源字符串的字符仍然会被拷贝——直至目标缓冲区的最后一个可用字符。然后，目标缓冲区的第一个和最后一个字符都被设为**'\0'**。除非是出于对安全性的考虑（你不想保留垃圾数据），否则这个问题并不重要。

最后要提到的一个细节与前面说过的“填充符”（filler）有关。在图2-4中，0xfd这个填充符替换了'\0'之后的所有字符——直至目标缓冲区的末尾。使用这些函数的Ex版本，你可以决定是否执行这种代价不菲的填充操作（尤其是在目标缓冲区很大的时候），以及用什么字节值来作为填充符使用。如果将STRSAFE_FILL_BEHIND_NULL加到dwFlag上，剩余的字符会被设为'\0'。如果用STRSAFE_FILL_BYTE宏来替代STRSAFE_FILL_BEHIND_NULL，就会用指定的字节来填充目标缓冲区中剩余的部分。

2.4.3 Windows 字符串函数

Windows也提供了各种字符串处理函数。其中许多函数（比如**lstrcat**和**lstrcpy**）已经不赞成使用了，因为它们无法检测缓冲区溢出问题。与此同时，ShlwApi.h定义了大量方便好用的字符串函数，可以用来对操作系统有关的数值进行格式化操作，比如**StrFormatKBSize**和**StrFormatByteSize**。有关shell字符串处理函数的描述，可访问以下网址：*http://msdn2.microsoft.com/en-us/library/ms538658.aspx*

我们经常都要比较字符串，以便进行相等性测试或者进行排序。为此，最理想的函数是**CompareString(Ex)**和**CompareStringOrdinal**。对于需要以符合用户语言习惯的方式向用户显示的字符串，请用**CompareString(Ex)**进行比较。**CompareString**函数的原型如下：

```
int CompareString(
  LCID locale,
  DWORD dwCmdFlags,
  PCTSTR pString1,
  int cch1,
```

```
  PCTSTR pString2,
  int cch2);
```

这个函数对两个字符串进行比较。**CompareString**的第一个参数指定一个区域设置ID（locale ID， LCID），这是一个32位值，用来标识一种语言。**CompareString**使用这个LCID来比较两个字符串，具体做法是检查字符在一种语言中的含义。以符合当地语言习惯的方式来比较，得到的结果对最终用户来说更有意义。不过，这种比较要比基于序数的比较（ordinal comparison）慢。可以调用Windows函数**GetThreadLocale**来得到主调线程的LCID：

```
LCID GetThreadLocale();
```

CompareString的第二个参数是一组标志，这些标志用于修改函数在比较字符串时采用的方法。表2-4展示了可能的标志。

表**2-4　CompareString函数所用的标志**

标志	含义
NORM_IGNORECASE LINGUISTIC_IGNORECASE	忽略大小写
NORM_IGNOREKANATYPE	不区分平假名和片假名字符
NORM_IGNORENONSPACE LINGUISTIC_IGNOREDIACRITIC	忽略 non-spacing 字符（译者注：non-spacing 字符通常是一些读音符号）
NORM_IGNORESYMBOLS	忽略符号
NORM_IGNOREWIDTH	不区分同一个字符的单字节和双字节形式
SORT_STRINGSORT	标点符号当成符号来处理

CompareString的其余4个参数指定了两个字符串及其各自的字符长度（字符数，而不是字节数据）。如果为**cch1**参数传入负值，函数会假设**pString1**字符串是以0来终止的，并计算字符串的长度；同样的道理也适用于**cch2**参数和**pString2**字符串。如果需要更高级的语言选项，那么应该考虑**CompareStringEx**函数。

为了比较编程类的字符串（如路径名、注册表项/值、XML元素/属性等等），应该使用**CompareStringOrdinal**，如下所示：

```
int CompareStringOrdinal(
  PCWSTR pString1,
  int cchCount1,
  PCWSTR pString2,
  int cchCount2,
  BOOL bIgnoreCase);
```

由于这个函数执行的是码位（code-point）比较，不考虑区域设置，所以速度很快。另外，由于编程类的字符串一般不会向最终用户显示，所以在这种情况下，最适合使用这个函数。注意，此函数只支持Unicode字符串。

CompareString和**CompareStringOrdinal**函数的返回值有别于C运行库的***cmp**字符串比较函数的返回值。**CompareString(Ordinal)**返回**0**表明函数调用失败，返回**CSTR_LESS_THAN**（定义为**1**）表明**pString1**小于**pString2**，返回**CSTR_EQUAL**（定义为**2**）表明**pString1**等于**pString2**，返回**CSTR_GREATER_THAN**（定义为**3**）表明**pString1**大于**pString2**。为方便起见，如果函数成功，你可以从返回值中减去2，使结果值与C运行库函数的结果值（**-1**，**0**和**+1**）保持一致。

2.6 为何要用 Unicode

开发应用程序的时候，强烈建议你使用Unicode字符和字符串。下面是一些理由。

- Unicode使程序的本地化变得更容易。
- 使用Unicode，只需发布一个二进制（.exe或DLL）文件，即可支持所有语言。
- Unicode提升了应用程序的效率，因为代码执行速度更快，占用内存更少。Windows内部的一切工作都是使用Unicode字符和字符串来进行的。所以，假如你非要传入ANSI字符或字符串，Windows就会被迫分配内存，并将ANSI字符或字符串转换为等价的Unicode形式。
- 使用Unicode，你的应用程序能轻松调用所有不反对使用（nondeprecated）的Windows函数，因为一些Windows函数提供了只能处理Unicode字符和字符串的版本。
- 使用Unicode，你的代码很容易与COM集成（后者要求使用Unicode字符和字符串）。
- 使用Unicode，你的代码很容易与.NET Framework集成（后者要要求使用Unicode字符和字符串）。
- 使用Unicode，能保证你的代码能够轻松操纵你自己的资源（其中的字符串总是Unicode的）

2.7 推荐的字符和字符串处理方式

基于本章到目前为止的内容，本节首先要总结开发代码时始终要牢记的几点。接下来要提供一些提示与技巧，帮你更好地处理Unicode和ANSI字符串。你最好现在就将应用程序转换为支持Unicode的形式，即使并不计划立即开始使用Unicode字符。下面是一些应该遵循的基本准则：

- 开始将文本字符串想象为字符的数组，而不是**char**或字节的数组。
- 为文本字符和字符串使用泛型（比如**TCHAR/PTSTR**）。
- 为字节、字节指针和数据缓冲区使用显式数据类型（**BYTE**和**PBYTE**） 。
- 为literal字符和字符串使用**TEXT**或**_T**宏，但为了保持一致性和更好的可读性，请避免两者混用。
- 执行全局替换。（例如，用**PTSTR**替换**PSTR**）。
- 修改字符串算术问题。例如，函数经常希望你传给它缓冲区的字符数，而不是字节数。这意味着你应该传入**_countof(szBuffer)**，而不是**sizeof(szBuffer)**。而且，如果需要为一个字符串分配一个内存块，而且知道字符串中的字符数，那么记住内存是以字节来分配的。这意味着你必须调用**malloc(nCharacters * sizeof(TCHAR))**，而不是调用**malloc(nCharacters)**。在前面列出的所有基本准则中，这是最难记住的一条，而且如果出错，编译器不会提供任何警告或错误信息。所以，最好定义一个宏来避免犯错：

```
#define chmalloc(nCharacters) (TCHAR*)malloc(nCharacters * sizeof(TCHAR)).
```

- 避免使用**printf**系的函数，尤其是不要用**%s**和**%S**字段类型来进行ANSI与字符串的相互转换。正确的做法是使用 **MultiByteToWideChar**和**WideCharToMultiByte**函数，详情参见后面的“Unicode与ANSI字符串转换”一节。
- **UNICODE**和**_UNICODE**符号要么都指定，要么一个都不指定。

对于字符串处理函数，应该遵循以下基本准则：

- 始终使用安全的字符串处理函数，比如那些后缀为**_s**的，或者前缀为**StringCch**的。后者主要在你想明确控制截断的时候使用；如果不想明确控制截断，则首选前者。
- 不要使用不安全的C运行库字符串处理函数（参见前面的建议）。一般情况下，你使用或实现的任何缓冲区处理例程都必须获取目标缓冲区的长度作为一个参数。C运行库提供了一系列缓冲区处理替代函数，比如**memcpy_s**，**memmove_s**，**wmemcpy_s**或**wmemmove_s**。只要定义了**__STDC_WANT_SECURE_LIB__**符号，所有这些方法都是可用的；CrtDefs.h默认定义了此符号。所以，不要对**_STDC_WANT_SECURE_LIB__**进行undef。
- 利用**/GS** (*http://msdn2.microsoft.com/en-us/library/aa290051(VS.71).aspx*)和**/RTCs**编译器标志来自动检测缓冲区溢出。
- 不要用Kernel32方法来进行字符串处理，比如**lstrcat**和**lstrcpy**。
- 在我们的代码中，需要要比较两种字符串。其中，编程类的字符串包括文件名、路径、XML元素/属性以及注册表项/值等等。对于这些字符串，应使用**CompareStringOrdinal**来进行比较。因为它非常快，而且不会考虑用户的区域设置。这是完全合理的，因为不管程序在世界上的什么地方运行，这种字符串都是不变的。用户字符串则一般要在用户界面上显示。对于这些字符串，应使用**CompareString(Ex)**来比较，因为在比较字符串的时候，它会考虑用户的区域设置。

你别无选择：作为专业开发人员，基于不安全的缓冲区处理函数来写代码是不允许的。正是这个原因，本书所有代码都是使用C运行库中的这些更安全的函数来写的。

2.8 Unicode 与 ANSI 字符串转换

我们使用Windows函数**MultiByteToWideChar**将多字节字符串转换为宽字符串。如下所示：

```
int MultiByteToWideChar(
   UINT uCodePage,
   DWORD dwFlags,
   PCSTR pMultiByteStr,
   int cbMultiByte,
   PWSTR pWideCharStr,
   int cchWideChar);
```

uCodePage参数标识了与多字节字符串关联的一个代码页值。**dwFlags**参数允许你进行额外的控制，它会影响使用了读音符号（比如重音）的字符。但是，一般情况下都不使用这些标志，所以为**dwFlags**参数传入的是0值（要想更多地了解这个标志的值，请阅读MSDN联机帮助，网址是*http://msdn2.microsoft.com/en-us/library/ms776413.aspx*）。**pMultiByteStr**

参数指定要转换的字符串，**cbMultiByte**参数指定字符串的长度（字节数）。如果为**cbMultiByte**参数传入-1的话，函数会自动判断源字符串的长度。

转换所得的Unicode版本的字符串被写入**pWideCharStr**参数所指定地址的内存缓冲区中。必须在**cchWideChar**参数中指定这个缓冲区的最大长度（字符数）。如果调用**MultiByteToWideChar**，并为**cchWideChar**参数传入0，函数就不会执行转换，而是返回为了成功转换，缓冲区必须提供的宽字符数（包括终止字符'\0'）。一般按照以下步骤将一个由多字节字符串转换为Unicode形式：

1. 调用MultiByteToWideChar，为pWideCharStr参数传入NULL，为cchWideChar参数传入0，为cbMultiByte参数传入-1。
2. 分配足以容纳转换后的Unicode字符串的一个内存块。它的大小是上一个MultiByteToWideChar调用的返回值乘以sizeof(wchar_t)。
3. 再次调用MultiByteToWideChar，这一次将缓冲区地址作为pWideCharStr参数的值传入，将第一次MultiByteToWideChar调用的返回值乘以sizeof(wchar_t)后得到的大小作为cchWideChar参数的值传入。
4. 使用转换后的字符串。
5. 释放Unicode字符串占用的内存块。

对应地，WideCharToMultiByte函数将宽字符字符串转换为多字节字符串，如下所示：

```
int WideCharToMultiByte(
   UINT uCodePage,
   DWORD dwFlags,
   PCWSTR pWideCharStr,
   int cchWideChar,
   PSTR pMultiByteStr,
   int cbMultiByte,
   PCSTR pDefaultChar,
   PBOOL pfUsedDefaultChar);
```

这个函数类似于**MultiByteToWideChar**函数。同样地，**uCodePage**标识了要与新转换的字符串关联的代码页。**dwFlags**参数允许你指定额外的转换控制。这些标志会影响有有读音符号的字符，以及系统不能转换的字符。但一般都不需要进行这种程度的转换控制，因而为**dwFlags**参数传入0。

pWideCharStr参数指定要转换的字符串的内存地址，**cchWideChar**参数指出该字符串的长度（字符数）。如果为**cchWideChar**参数传入-1，则由函数来判断源字符串的长度。

转换所得的多字节版本的字符串被写入**pMultiByteStr**参数所指定的缓冲区。必须在**cbMultiByte**参数中指定此缓冲区的最大大小（字节数）。调用**WideCharToMultiByte**函数时，如果将0作为**cbMultiByte**参数的值传入，会导致该函数返回目标缓冲区需要的大小。将宽字符串转换为多字节字符串时，采取的步骤和前面将多字节字符串转换为宽字符串的步骤相似；唯一不同的是，返回值直接就是确保转换成功所需的字节数，所以无需执行乘法运算。

注意，与**MultiByteToWideChar**函数相比，**WideCharToMultiByte**函数接受的参数要多两个，分别是**pDefaultChar**和**pfUsedDefaultChar**。只有一个字符在**uCodePage**指定的代码页中无表示时，**WideCharToMultiByte**函数才会使用这两个参数。遇到一个不能转换的宽字符，函数便会使用**pDefaultChar**参数指向的字符。如果这个参数为NULL（这是很常见的一个情况），函数就会使用一个系统默认的字符。这个默认字符通常是一个问号。这对文件名来说非常危险，因为问号是一个通配符。

pfUsedDefaultChar参数指向一个布尔变量；在宽字符字符串中，如果至少有一个字符不能转换为其多字节形式，函数就会把这个变量设为**TRUE**。如果所有字符都能成功转换，就会把这个变量设为**FALSE**。可以在函数返回后测试该变量，验证宽字符字符串是否已成功转换。同样地，通常为此参数传入**NULL**值。

有关如何使用这些函数的更完整的描述，请参阅Platform SDK文档。

2.8.1 导出 ANSI 和 Unicode DLL 函数

使用上一节描述的这两个函数，可以轻松创建一个函数的Unicode版本和ANSI版本。例如，假定你有一个动态链接库，其中的一个函数能反转字符串中的所有字符。可以像下面这样写这个函数的Unicode版本：

```
BOOL StringReverseW(PWSTR pWideCharStr, DWORD cchLength) {

// Get a pointer to the last character in the string.
PWSTR pEndOfStr = pWideCharStr + wcsnlen_s(pWideCharStr , cchLength) - 1;
wchar_t cCharT;
// Repeat until we reach the center character in the string.
while (pWideCharStr < pEndOfStr) {
  // Save a character in a temporary variable.
  cCharT = *pWideCharStr;
  // Put the last character in the first character.
  *pWideCharStr = *pEndOfStr;
  // Put the temporary character in the last character.
  *pEndOfStr = cCharT;
  // Move in one character from the left.
  pWideCharStr++;
  // Move in one character from the right.
  pEndOfStr--;
  }
  // The string is reversed; return success.
  return(TRUE);
}
```

另外，在写函数的ANSI版本时，可以让它根本不执行反转字符串的实际工作。相反，在ANSI版本中，你只需让它将ANSI字符串转换成Unicode字符串，然后将Unicode字符串

传给刚才的**StringReverseW**函数，最后将反转后的字符串转换回ANSI字符串。如下所示：

```
BOOL StringReverseA(PSTR pMultiByteStr, DWORD cchLength) {
  PWSTR pWideCharStr;
  int nLenOfWideCharStr;
  BOOL fOk = FALSE;
  // Calculate the number of characters needed to hold
  // the wide-character version of the string.
  nLenOfWideCharStr = MultiByteToWideChar(CP_ACP, 0,
    pMultiByteStr, cchLength, NULL, 0);
  // Allocate memory from the process' default heap to
  // accommodate the size of the wide-character string.
  // Don't forget that MultiByteToWideChar returns the
  // number of characters, not the number of bytes, so
  // you must multiply by the size of a wide character.
  pWideCharStr = (PWSTR)HeapAlloc(GetProcessHeap(), 0,
    nLenOfWideCharStr * sizeof(wchar_t));
  if (pWideCharStr == NULL)
    return(fOk);
  // Convert the multibyte string to a wide-character string.
  MultiByteToWideChar(CP_ACP, 0, pMultiByteStr, cchLength,
    pWideCharStr, nLenOfWideCharStr);
  // Call the wide-character version of this
  // function to do the actual work.
  fOk = StringReverseW(pWideCharStr, cchLength);
  if (fOk) {
  // Convert the wide-character string back
  // to a multibyte string.
  WideCharToMultiByte(CP_ACP, 0, pWideCharStr, cchLength,
    pMultiByteStr, (int)strlen(pMultiByteStr), NULL, NULL);
  }
  // Free the memory containing the wide-character string.
  HeapFree(GetProcessHeap(), 0, pWideCharStr);
  return(fOk);
}
```

最后，在随同动态链接库发布的头文件中，像下面这样提供两个函数的原型：

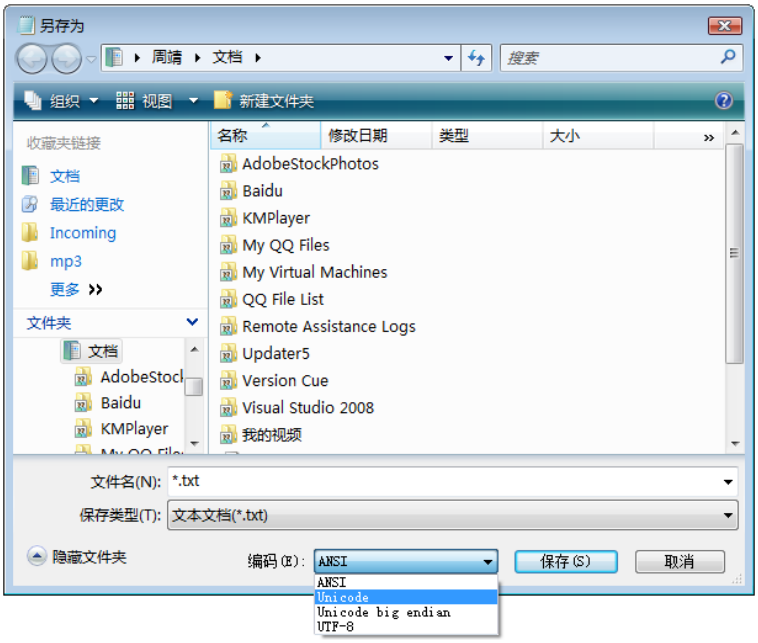
```
BOOL StringReverseW(PWSTR pWideCharStr, DWORD cchLength);
BOOL StringReverseA(PSTR pMultiByteStr, DWORD cchLength);

#ifdef UNICODE
#define StringReverse StringReverseW
#else
#define StringReverse StringReverseA
```

```
#endif // !UNICODE
```

2.8.2 判断文本是 ANSI 还是 Unicode

Windows的“记事本”应用程序不仅能打开Unicode文件和ANSI文件，还能创建这两种文件。来看看图2-5展示的记事本的“另存为”对话框，请注意保存一个文本文件的不同方式。

图2-5 Windows Vista记事本应用程序的“另存为”对话框



对于需要打开文本文件并进行处理的大多数应用程序（比如编译器）而言，如果应用程序能够在打开一个文件之后，分辨此文件包含的是ANSI字符，还是Unicode字符，将是多么惬意而方便啊！由AdvApi32.dll导出、在WinBase.h中声明的**IsTextUnicode**函数有助于进行这种分辨：

```
BOOL IsTextUnicode(CONST PVOID pvBuffer, int cb, PINT pResult);
```

文本文件的问题在于，它们的内容没有任何硬性的、可供快速判断的规则。所以，要判断文件中包含的是ANSI字符还是Unicode字符，就显得相当困难。**IsTextUnicode**函数使用一系列统计和决策方法来猜测缓冲区中的内容。由于这并非一种精确的科学，**IsTextUnicode**函数可能会返回错误的结果。

它的第一个参数是**pvBuffer**，标识了要测试的缓冲区的地址。此数据是一个void指针，因为还不知道即将面对的是一组ANSI字符还是Unicode字符。

第二个参数是**cb**，它指定**pvBuffer**指向的缓冲区的字节数。同样地，由于不知道缓冲区中是什么，所以**cb**是一个字节数而不是字符数。注意，你不必指定整个缓冲区的长度。当然，**IsTextUnicode**函数测试的字节数越多，结果越精确。

第三个参数是**pResult**，这是一个整数的地址，在调用**IsTextUnicode**函数之前，必须初始化这个整数。在这个整数的初始值中，应指出希望**IsTextUnicode**执行哪些测试。也可以为此参数转入**NULL**，在这种情况下，**IsTextUnicode**函数将执行它能执行的每一项测试（详情参阅Platform SDK文档）。

如果**IsTextUnicode**函数认为缓冲区包含的是Unicode文本，就会返回**TRUE**；反之返回**FALSE**。在**pResult**参数指向的整数中，如果指定了具体的测试项目，那么函数在返回之前，还会设置此整数中的位，以反映每个测试项目的结果。

第17章的FileRev示例程序将演示**IsTextUnicode**函数的具体用法。

第 3 章　内核对象

本章内容包括：

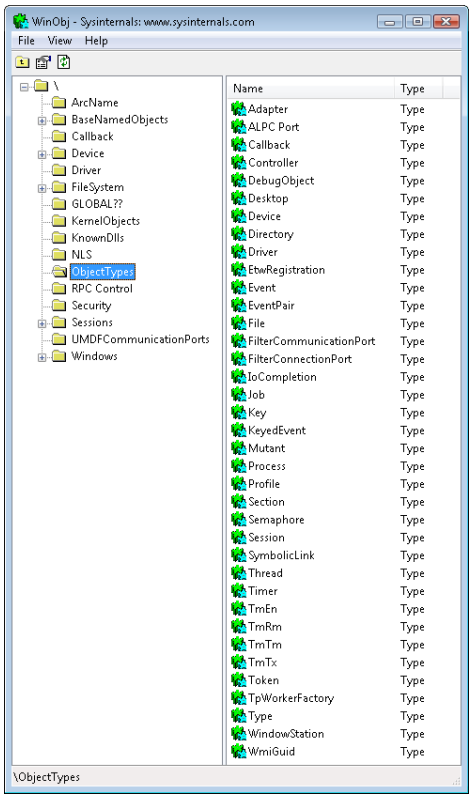
- 何为内核对象
- 进程内核对象句柄表
- 跨进程边界共享内核对象

为了帮助你理解 Windows 应用程序编程接口（application programming interface，AP），首先让我们探讨一下内核对象（kernel objects）及其句柄（handles）。本章讨论的是一些相对抽象的概念——我们不打算讨论任何具体的内核对象的细节。相反，讨论的是所有内核对象共通的一些特性。

我本来更愿意从一个更具体的主题开始的，但要想成为一名专业 Windows 软件开发人员，透彻理解内核对象至关重要。在系统和我们写的应用程序中，内核对象用于管理进程、线程和文件等诸多种类的大量资源。本章要陈述的概念将频繁出现在本书其他各章中。不过，我也的确意识到，除非你开始使用实际的函数来操纵内核对象，否则本章讨论的一些主题是不太容易理解的。所以，当你阅读本书其他各章的时候，可能需要回过头来参考本章。

3.1　何为内核对象

作为 Windows 软件开发人员，你经常都要创建、打开和处理内核对象。系统会创建和处理几种类型的内核对象，比如访问令牌（access token）对象、事件对象、文件对象、文件映射对象、I/O 完成端口对象、作业对象、mailslot 对象、mutex 对象、pipe 对象、进程对象、semaphore 对象、线程对象、waitable timer 对象以及 thread pool worker factory 对象等等。利用 Sysinternals 的免费工具 WinObj（*http://www.microsoft.com/technet/sysinternals/utilities/winobj.mspx*），可以查看所有内核对象类型的一个列表。为了看到这个列表，必须在 Windows 资源管理器中，以管理员身份运行此工具。



这些对象是通过形形色色名称的函数来创建的，函数的名称并非肯定与内核级别上使用的对象类型对应。例如，调用 **CreateFileMapping** 函数，系统将创建和一个 **Section** 对象对应的文件映射（如你在 WinObj 中所见）。每个内核对象都只是一个内存块，它由内核分配，并只能由内核访问。这个内存块是一个数据结构，其成员维护着与对象相关的信息。少数成员（安全描述符、使用计数等等）是所有对象都有的，但其他大多数成员都是不同的对象类型特有的。例如，进程对象有一个进程 ID，一个基本的优先级和一个退出代码；而文件对象有一个字节偏移量（byte offset）、一个共享模式和一个打开模式。

由于内核对象的数据结构只能由内核访问，所以应用程序不能在内存中定位这些数据结构并直接更改其内容。Microsoft 有意强化了这个限制，确保内核对象结构保持一致性状态。正是因为有这个限制，所以 Microsoft 能自由地添加、删除或修改这些结构中的成员，同时不会干扰任何应用程序的正常运行。

既然不能直接更改这些结构，应用程序如何操纵这些内核对象呢？答案是利用 Windows 提供的一组函数，以经过良好定义的方式来操纵这些结构。使用这些函数，始终可以访问这些内核对象。调用一个会创建内核对象的函数后，函数会返回一个句柄（handle），它标识了创建的对象。可以将这个句柄想象为一个不透明（opaque）的值，它可由进程中的任何线程使用。在 32 位 Windows 进程中，句柄是一个 32 位值；在 64 位 Windows 进程中，则是一个 64 位值。可将这个句柄传给各种 Windows 函数，告诉系统你想操纵哪一个内核对象。本章后面会更多地谈及这些句柄的详情。

为了增强操作系统的可靠性，这些句柄值是与进程相关的。所以，如果将句柄值传给另一个进程中的线程（通过某种进程间通信方式），那么另一个进程用你的进程的句柄值来发出调用时，就可能失败；甚至更糟，它们会在你的进程句柄表的同一个索引位置处，创建到一个

完全不同的内核对象的引用。3.3 节“跨进程边界共享内核对象”将介绍 3 种机制，可利用它们实现多个进程成功共享同一个内核对象。

3.1.1 使用计数

内核对象的所有者是内核，而非进程。换言之，如果你的进程调用一个函数来创建了一个内核对象，然后进程终止运行，则内核对象并不一定会销毁。大多数情况下，这个内核对象是会销毁的，但假如另一个进程正在使用你的进程创建的内核对象，那么在其他进程停止使用它之前，它是不会销毁的。总之，内核对象的生命期可能长于创建它的那个进程。

内核知道当前有多少个进程正在使用一个特定的内核对象，因为每个对象都包含一个使用计数（usage count）。使用计数是所有内核对象类型都有的一个数据成员。初次创建一个对象的时候，其使用计数被设为 1。另一个进程获得对现有内核对象的访问后，使用计数就会递增。进程终止运行后，内核将自动递减此进程仍然打开的所有内核对象的使用计数。一个对象的使用计数变成 0，内核就会销毁该对象。这样一来，可以保证系统中不存在没有被任何进程引用的内核对象。

3.1.2 内核对象的安全性

内核可以用一个安全描述符（SD）来保护。安全描述符描述了谁（通常是它的创建者）拥有对象；哪些用户和用户允许访问或使用此对象；以及哪些组和用户拒绝访问此对象。安全描述符通常在编写服务器应用程序的时候使用。但是，在 Microsoft Windows Vista 中，对于具有专用（private）命名空间的客户端应用程序，这个特性变得更加明显，详见本章以及 4.4 节“当管理员以标准用户的身份运行时”。

用于创建内核对象的所有函数几乎都有指向一个 **SECURITY_ATTRIBUTES** 结构的指针作为参数，如下面的 **CreateFileMapping** 函数所示：

```
HANDLE CreateFileMapping(
  HANDLE hFile,
  PSECURITY_ATTRIBUTES psa,
  DWORD flProtect,
  DWORD dwMaximumSizeHigh,
  DWORD dwMaximumSizeLow,
  PCTSTR pszName);
```

大多数应用程序只是为这个参数传入 NULL，这样创建的内核对象具有默认的安全性——具体包括哪些默认的安全性，要取决于当前进程的安全令牌（security token）。但是，你也可以分配一个 **SECURITY_ATTRIBUTES** 结构，初始化它，再将它的地址传给这个参数。**SECURITY_ATTRIBUTES** 结构如下所示：

```
typedef struct _SECURITY_ATTRIBUTES {
  DWORD nLength;
  LPVOID lpSecurityDescriptor;
```

```
  BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

虽然这个结构称为 **SECURITY_ATTRIBUTES**，但它实际只包含一个和安全性有关的成员，即 **lpSecurityDescriptor**。如果想限制对你创建的一个内核对象的访问，就必须创建一个安全描述符，然后像下面这样初始化 **SECURITY_ATTRIBUTES** 结构：

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa); // Used for versioning
sa.lpSecurityDescriptor = pSD; // Address of an initialized SD
sa.bInheritHandle = FALSE; // Discussed later
HANDLE hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, &sa,
   PAGE_READWRITE, 0, 1024, TEXT("MyFileMapping"));
```

由于这个成员与安全性没有任何关系，所以我将把 **bInheritHandle** 成员推迟到 3.3.1 节“使用对象句柄继承”小节讨论。

如果想访问现有的内核对象（而不是新建一个），必须指定打算对此对象执行哪些操作。例如，如果想访问一个现有的文件映射内核对象，以便从中读取数据，那么可以像下面这样调用 **OpenFileMapping**：

```
HANDLE hFileMapping = OpenFileMapping(FILE_MAP_READ, FALSE,
  TEXT("MyFileMapping"));
```

将 **FILE_MAP_READ** 作为第一个参数传给 **OpenFileMapping**，我指出要在获得对这个文件映射对象的访问权之后，从中读取数据。**OpenFileMapping** 函数在返回一个有效的句柄值之前，会先执行一次安全检查。如果我（登录的用户）被许可访问现有的文件映射内核对象，**OpenFileMapping** 会返回一个有效的句柄值。但是，如果被拒绝访问，**OpenFileMapping** 就会返回 **NULL**；如果调用 **GetLastError**，将返回值 5（**ERROR_ACCESS_DENIED**）。记住，如果利用返回的句柄来调用一个 API，但这个 API 需要有别于 **FILE_MAP_READ** 的权限，那么同样会发生“拒绝访问”错误。由于大多数应用程序都不使用安全性，所以我不打算进一步讨论这个主题了。

虽然许多应用程序都不需要关心安全性，但许多 Windows 函数都要求你传入必要的安全访问信息。为以前版本的 Windows 设计的一些应用程序之所以在 Windows Vista 上不能正常工作，就是因为在实现这些程序时，没有充分考虑安全性。

例如，假定一个应用程序在启动时要从一个注册表子项中读取一些数据。正确的做法是调用 **RegOpenKeyEx**，向其传入 **KEY_QUERY_VALUE**，从而指定查询子项数据的权限。

然而，许多应用程序都是为 Windows 2000 之前的操作系统开发的，对安全性没有任何考虑。有的软件开发人员还是按照老习惯，在调用 **RegOpenKeyEx** 函数的时候，传入 **KEY_ALL_ACCESS** 作为期望的访问权限。之所以喜欢这样做，是由于它更简单，不需要动脑筋想需要什么权限。但是，这样做的问题在于，对于一个不是管理员的标准用户，注册表项（比如 HKLM）也许是只读的。所以，当这样的应用程序在 Windows Vista 上面运行时，调用 **RegOpenKeyEx** 函数并传递 **KEY_ALL_ACCESS** 就会失败。另外，如果没有正确的

错误检查，运行这样的应用程序会得到完全不可预期的结果。

其实，开发人员只需稍微注意一下安全性，将 **KEY_ALL_ACCESS** 改为 **KEY_QUERY_VALUE**（在本例中只需如此），应用程序就能在所有操作系统平台上正常运行了。

忽略正确的安全访问标志是很多开发人员最大的失误之一。只要使用了正确的安全访问标志，你的程序在不同版本的 Windows 之间移植时，绝对会变得更容易。不过，你还需注意到，每个新版本的 Windows 会带来老版本中没有的一套新的限制。例如在 Windows Vista 中，你需要关注“用户帐户控制”（User Account Control，UAC）特性。默认情况下，为安全起见，UAC 会强制应用程序在一个受限的上下文中运行，即使当前用户是 Administrators 组的成员。我们将在第 4 章“进程”详细讨论 UAC。

除了使用内核对象，应用程序可能还要使用其他类型的对象，比如菜单、窗口、鼠标光标、画刷和字体。这些属于 User 对象或 GDI（Graphical Device Interface）对象，而非内核对象。首次进行 Windows 编程时，往往很难区分 User/GDI 对象和内核对象。例如，图标是 User 对象还是内核对象？要想判断一个对象是不是内核对象，最简单的方式是查看创建这个对象的函数。几乎所有创建内核对象的函数都有一个允许你指定安全属性信息的参数，就像前面展示的 **CreateFileMapping** 函数一样。

相反，用于创建 User 或 GDI 对象的函数都没有 **PSECURITY_ATTRIBUTES** 参数。例如下面的 **CreateIcon** 函数：

```
HICON CreateIcon(
  HINSTANCE hinst,
  int nWidth,
  int nHeight,
  BYTE cPlanes,
  BYTE cBitsPixel,
  CONST BYTE *pbANDbits,
  CONST BYTE *pbXORbits);
```

MSDN 上有一篇文章（网址为 *http://msdn.microsoft.com/msdnmag/issues/03/01/GDILeaks*）详细讨论了 GDI 和 User 对象，以及如何跟踪这些对象。

3.2 进程内核对象句柄表

一个进程在初始化时，系统将为它分配一个句柄表（handle table）。这个句柄表仅供内核对象使用，不适用于 User 或 GDI 对象。句柄表的结构如何？如何管理句柄表？这些细节尚无文档可以参考。一般情况下，我会避免讨论操作系统中没有编档的主题。但这里要破例一下，因为我认为作为一名优秀的 Windows 程序员，必须理解如何管理进程的句柄表。由于这些信息还没有正式编入文档，所以我不敢保证这里所讨论的细节都是准确无误的，而且不同 Windows 版本的内部实现肯定有所区别。所以，下面的讨论只是帮助你增强理解，而不是让你正式地学习系统的运行机制。

表 3-1 显示了一个进程的句柄表。可以看出，它只是一个由数据结构组成的数组。每个结构

都包含指向一个内核对象的指针、一个访问掩码（access mask）和一些标志。

表 3-1 进程的句柄表的结构

索引	指向内核对象内存块的指针	访问掩码(包含标志位的一个 DWORD)	标志
1	0 x ? ? ? ? ? ? ? ?	0 x ? ? ? ? ? ? ? ?	0 x ? ? ? ? ? ? ? ?
2	0 x ? ? ? ? ? ? ? ?	0 x ? ? ? ? ? ? ? ?	0 x ? ? ? ? ? ? ? ?
...	...	...	...

3.2.1 创建一个内核对象

一个进程首次初始化的时候，其句柄表为空。当进程内的一个线程调用一个会创建内核对象的函数时（比如 **CreateFileMapping**），内核将为这个对象分配并初始化一个内存块。然后，内核扫描进程的句柄表，查找一个空白的记录项（empty entry）。由于表 3-1 展示的是一个空白句柄表，所以内核在索引 1 位置找到空白的记录项，并对其进行初始化。具体地说，指针成员会被设置成内核对象的数据结构的内部内存地址，访问掩码将被设置成拥有完全访问权限，标志也会设置（我将在 3.3.1 节“使用对象句柄继承”讨论标志的问题）。

下面列出了一些用创建内核对象的函数（当然并不完整）：

```
HANDLE CreateThread(
  PSECURITY_ATTRIBUTES psa,
  size_t dwStackSize,
  LPTHREAD_START_ROUTINE pfnStartAddress,
  PVOID pvParam,
  DWORD dwCreationFlags,
  PDWORD pdwThreadId);

HANDLE CreateFile(
  PCTSTR pszFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  PSECURITY_ATTRIBUTES psa,
  DWORD dwCreationDisposition,
  DWORD dwFlagsAndAttributes,
  HANDLE hTemplateFile);

HANDLE CreateFileMapping(
  HANDLE hFile,
  PSECURITY_ATTRIBUTES psa,
  DWORD flProtect,
  DWORD dwMaximumSizeHigh,
  DWORD dwMaximumSizeLow,
  PCTSTR pszName);
```

```
HANDLE CreateSemaphore(
  PSECURITY_ATTRIBUTES psa,
  LONG lInitialCount,
  LONG lMaximumCount,
  PCTSTR pszName);
```

用于创建内核对象的任何函数都会返回一个相对于进程的句柄，这个句柄可由同一个进程中运行的所有线程使用。句柄值实际应该除以 4（或右移两位，以忽略 Windows 操作系统内部使用的最后两位），从而得到在进程句柄表中的真正索引（内核对象的信息将保存在这个位置处）。所以，在调试应用程序，并查看一个内核对象句柄的实际值时，会看到 4、8 之类的小值。记住，句柄的含义尚未文档化，将来可能发生变化。

调用一个函数时，如果它接受一个内核对象句柄作为参数，就必须把 Create*函数返回的值传给它。在内部，这个函数会查找进程的句柄表，获得目标内核对象的地址，然后以一种良好定义的方式来操纵对象的数据结构。

如果传入一个无效的句柄，函数就会失败，**GetLastError** 会返回 6（**ERROR_INVALID_HANDLE**）。由于句柄值实际是作为进程句柄表的索引来使用的，所以这些句柄是相对于当前这个进程的，无法供其他进程使用。如果你真的在其他进程中使用它，那么实际引用的是那个进程的句柄表的同一个索引位置处的内核对象——只是索引值相同而已，你根本不知道它会指向什么对象。

调用函数来创建一个内核对象时，如果调用失败，那么返回的句柄值通常为 0（**NULL**），这就是为什么第一个有效的句柄值为 4 的原因。之所以失败，可能是由于系统内存不足，或者遇到了一个安全问题。遗憾的是，有几个函数在调用失败时会返回句柄值–1（也就是在 WinBase.h 中定义的 **INVALID_HANDLE_VALUE**）。例如，如果 **CreateFile** 无法打开指定文件，它会返回 **INVALID_HANDLE_VALUE**，而不是 **NULL**。凡是用于创建内核对象的函数，在你检查它们的返回的值时，务必相当仔细。具体说，以当前这个例子为例，只有在调用 CreateFile 时，才准许将它的返回值与 **INVALID_HANDLE_VALUE** 比较。以下代码是不正确的：

```
HANDLE hMutex = CreateMutex(…);
if (hMutex == INVALID_HANDLE_VALUE) {
  // 这里的代码永远不会执行，
  // 因为 CreateMutex 在失败时总是返回 NULL。
}
```

类似地，以下代码也是不正确的：

```
HANDLE hFile = CreateFile(…);
if (hFile == NULL) {
  // 这里的代码永远不会执行，因为 CreateFile
  // 在失败的时候会返回 INVALID_HANDLE_VALUE(-1)。
}
```

3.2.2 关闭内核对象

无论以什么方式创建内核对象，都要调用 **CloseHandle** 向系统指出你已经结束使用对象，如下所示：

```
BOOL CloseHandle(HANDLE hobject);
```

在内部，该函数首先检查主调进程的句柄表，验证“传给函数的句柄值”标识的是“进程确实有权访问的一个对象”。如果句柄是有效的，系统就将获得内核对象的数据结构的地址，并在结构中递减“使用计数”成员。如果使用计数变成 0，内核对象将被销毁，并从内存中删除。

如果传给 **CloseHandle** 函数的是一个无效的句柄，那么可能发生以下两种情况之一：如果进程是正常运行的，**CloseHandle** 将返回 **FALSE**，而 **GetLastError** 返回 **ERROR_INVALID_HANDLE**。如果进程正在被调试，那么系统将抛出 0xC0000008 异常(“指定了无效的句柄”)，便于你调试这个错误。

就在 **CloseHandle** 函数返回之前，它会清除进程句柄表中的记录项——这个句柄现在对你的进程来说是无效的，不要再试图用它。无论内核对象当前是否销毁，这个清除过程都会发生！一旦调用 **CloseHandle**，你的进程就不能访问那个内核对象；但是，如果对象的使用计数没有递减至 0，它就不会被销毁。这是完全正常的；它表明另外还有一个或多个进程在使用该对象。当其他进程全部停止使用这个对象后（通过调用 **CloseHandle**），对象就会被销毁。

> 注意
> 通常，在创建一个内核对象时，我们会将它的句柄保存到一个变量中。将此变量作为参数调用了 CloseHandle 函数后，还应同时将这个变量设为 NULL。如果不小心用这个变量来调用了一个 Win32 函数，可能会发生两种意外情况。第一种可能的情况是，由于此变量所引用的句柄表记录项已被清除，所以 Windows 会接收到一个无效的参数，所以会报告一个错误。另外，还可能发生另一种更难调试的情况。创建一个新的内核对象时，Windows 会在句柄表中查找空白记录项。所以，如果新的内核对象已在你的应用程序工作流中构造好，那个变量所引用的这个句柄表中，肯定包含某个新建的内核对象。因此，一旦错误地用这个尚未设为 NULL 的变量来调用一个 Win32 函数，就可能会定位到一个错误类型的内核对象（这种情况会报错）——甚至更糟，可能会定位到一个类型（和已经关闭的内核对象）相同的内核对象（这种情况不会报错）。在第二种情况下，你的应用程序的状态将损坏，没有任何办法可以恢复。

假定你忘记调用 **CloseHandle**，会发生对象泄漏的情况吗？嗯，不一定。在进程运行期间，进程可能发生资源（比如内核对象）泄漏的情况。但是，当进程终止运行，操作系统会确保此进程所使用的所有资源都被释放——这是可以担保的！对于内核对象，操作系统执行的是以下操作：进程终止时，系统自动扫描该进程的句柄表。如果这个表中有任何有效的记录项（即进程终止前没有关闭的对象），操作系统会为你关闭这些对象句柄。任何这些对象的使用计数递减至 0，内核就会销毁对象。

所以，当你的应用程序运行时，它可能会泄漏内核对象；但当进程终止运行，系统能保证一切都被正确清除。顺便说一下，这适用于所有内核对象、资源（包括 GDI 对象在内）以及内存块
。进程终止运行时，系统会确保你的进程不会留下任何东西。要在应用程序运行期间检测内核对象泄漏，一个简单的办法是使用 Windows 任务管理器。首先，如图 3-1 所示，你需要选择 View（查看）| Select Columns（选择列）菜单，然后在 Select Process Page Columns（选择进程页列）对话框中，指定在 Processes（进程）卡片中显示 Handles（句柄数）列。

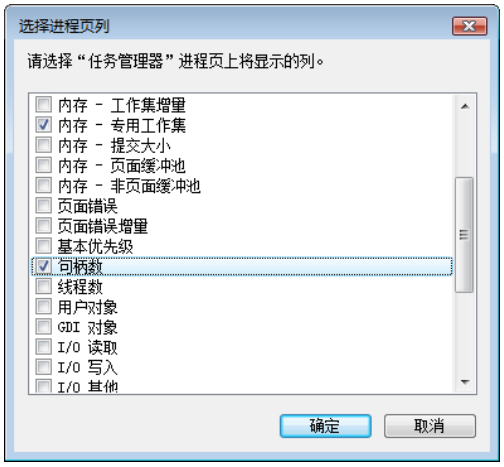


图 3-1　在“选择进程页列”对话框中选择“句柄数”列

然后就可以监视任何一个应用程序使用的内核对象数了，如图 3-2 所示。

图 3-2　在 Windows 任务管理器中统计句柄数

如果“句柄数”列显示的数字持续增长，下一步就是判断哪些内核对象尚未关闭。为此，可以使用 Sysinternals 提供的一款免费 Process Explorer 工具（网址是 *http://www.microsoft.com/technet/sysinternals/ProcessesAndThreads/ProcessExplorer.mspx*）。首先，右击下方 Handlers 窗格的标题行（如果这个窗格没有出现，请从 View 菜单中选择 Show Lower Pane），并从弹出菜单中选择 Select Columns。然后，在图 3-3 所示的“Select Columns”对话框中，勾选择所有列标题。

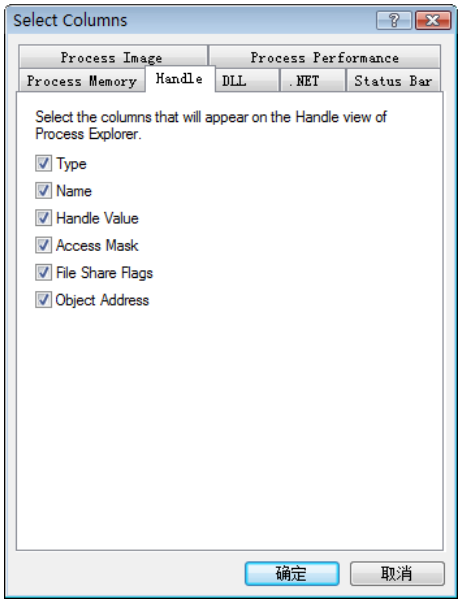


图 3-3　选择要在 Process Explorer 的 Handle 视图中显示的详细信息

完成此项操作之后，在 View 菜单中将 Update Speed 改为 Paused。在顶部的窗格中选择想检查的进程，按 F5 键来获得一份最新的内核对象列表。然后，开始使用你的应用程序，执行一个你想要验证的工作流。完成之后，再次在 Process Explorer 中按 F5 键。在此期间生成的每个新的内核对象都显示为绿色，如图 3-4 下方较深的区域所示。

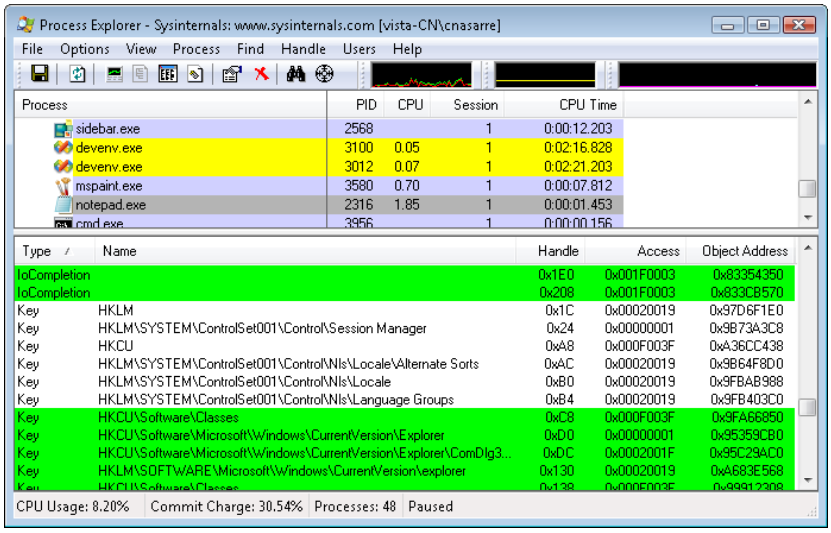


图 3-4　在 Process Explorer 中检测新的内核对象

注意，第一列显示了没有关闭的内核对象的类型。为了使你有更大的机会确定泄漏位置，第二列提供了内核对象的名称。如下一节所述，利用作为内核对象名称的字符串，你可以在不同的进程之间共享这个对象。显然，根据第一列的类型和第二列的名称，你可以更容易地判断出哪个对象没有关闭。如果泄漏了大量对象，它们并不一定会被命名，因为只能创建一个命名对象的一个实例——其他尝试会单纯地打开那个实例。

3.3 跨进程边界共享内核对象

在很多时候，不同进程中运行的线程需要共享内核对象。下面罗列了一些理由。

- 利用文件映射对象，可以在同一台机器上运行的两个不同进程之间共享数据块。
- 借助 mailslots 和 named pipes，在网络中的不同计算机上运行的进程可以相互发送数据块。
- mutexes、semaphores 和事件允许不同进程中的线程同步执行。例如，一个应用程序可能需要在完成某个任务之后，向另一个应用程序发出通知。

由于内核对象的句柄是相对于每一个进程的，所以执行这些任务并不轻松。不过，Microsoft 也有充分的理由需要将句柄设计成“相对于进程”（process-relative）的。其中最重要的原因是健壮性（可靠性）。如果把内核对象句柄设计成相对于整个系统，或者说把它们设计成“系统级”的句柄，一个进程就可以很容易获得到“另一个进程正在使用的一个对象”的句柄，从而对该进程造成严重破坏。之所以将句柄设计成“相对于进程”，或者说把它们设计成“进程级”的句柄，还有一个原因是安全性。内核对象是用安全性保护起来的，一个进程在试图操纵一个对象之前，必须先申请操纵它的权限。对象的创建者为了阻止一个未经许可的用户“碰”自己的对象，只需拒绝该用户访问它。

在下一节，我们要讨论如何利用三种不同的机制来允许进程共享内核对象：使用对象句柄继承；为对象命名；以及复制对象句柄。

3.3.1 使用对象句柄继承

只有在进程之间有一个父－子关系的时候，才可以使用对象句柄继承。在这种情况下，父进程有一个或多个内核对象句柄可以使用，而且父进程决定生成一个子进程，并允许子进程访问父进程的内核对象。为了使这种继承生效，父进程必须执行几个步骤。

首先，当父进程创建一个内核对象时，父进程必须向系统指出它希望这个对象的句柄是可以继承的。我有时听到别人说起“对象继承”这个词。但是，世界上根本没有“对象继承”这样的事情。Windows 支持的是“对象句柄的继承”；换言之，只有句柄才是可以继承的，对象本身是不能继承的。

为了创建一个可继承的句柄，父进程必须分配并初始化一个 **SECURITY_ATTRIBUTES** 结构，并将这个结构的地址传给具体的 **Create** 函数。以下代码创建了一个 mutex 对象，返回它的一个可继承的句柄：

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE; // 使返回的句柄成为可继承的句柄

HANDLE hMutex = CreateMutex(&sa, FALSE, NULL);
```

以上代码初始化了一个 **SECURITY_ATTRIBUTES** 结构，表明对象要用默认安全性来创建，而且返回的句柄应该是可继承的。

接下来谈谈在进程的句柄表记录项中保存的标志。句柄表中的每个记录项都有一个指明句柄是否可以继承的标志位。如果在创建内核对象的时候将 **NULL** 作为 **PSECURITY_ATTRIBUTES** 参数传入，则返回的句柄是不可继承的，这个标志位为 0。将 **bInheritHandle** 成员设为 **TRUE**，则导致这个标志位被设为 1。

以表 3-2 的进程句柄表为例。在这个例子中，进程有权访问两个内核对象（句柄 1 和 3）。句柄 1 是不可继承的，但句柄 3 是可以继承的。

表 3-2　包含两个有效记录项的进程句柄表

索引	指向内核对象内存块的指针	访问掩码（包含标志位的一个 DWORD）	标志
1	0xF0000000	0x????????	0x00000000
2	0x00000000	(不可用)	(不可用)
3	0xF000010	0x????????	0x00000001

为了使用对象句柄继承，下一步是由父进程生成子进程。这是通过 **CreateProcess** 函数来完成的，如下所示：

```
BOOL CreateProcess(
  PCTSTR pszApplicationName,
  PTSTR pszCommandLine,
  PSECURITY_ATTRIBUTES psaProcess,
  PSECURITY_ATTRIBUTES psaThread,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  PVOID pvEnvironment,
  PCTSTR pszCurrentDirectory,
  LPSTARTUPINFO pStartupInfo,
  PPROCESS_INFORMATION pProcessInformation);
```

我们将在第 4 章详细讨论这个函数，现在请注意 **bInheritHandles** 参数。通常，在生成一个进程时，要向该参数传递 **FALSE**。这个值向系统表明：你不希望子进程继承父进程句柄表中的“可继承的句柄”。

相反，如果向这个参数传递 **TRUE**，子进程就会继承父进程的“可继承的句柄”的值。传递 **TRUE** 时，操作系统会创建新的子进程，但不允许子进程立即执行它的代码。当然，系统会为子进程创建一个新的、空白的进程句柄表——就像它为任何一个新进程所做的那样。但是，由于你向 **CreateProcess** 函数的 **bInheritHandles** 参数传递了 **TRUE**，所以系统还会多做一件事情：它会遍历父进程的句柄表，对它的每一个记录项进行检查。凡是包含一个有效的“可继承的句柄”的项，都会被完整地拷贝到子进程的句柄表。在子进程的句柄表中，拷贝项的位置与它在父进程句柄表中的位置是完全一样的。这是非常重要的一个设计，因为它意味着：在父进程和子进程中，对一个内核对象进行标识的句柄值是完全一样的。

除了拷贝句柄表的记录项，系统还会递增内核对象的使用计数，因为两个进程现在都在使用这个对象。一个内核对象要想被销毁，父进程和子进程要么都对这个对象调用 **CloseHandle**，要么都终止运行。子进程不一定先终止——但父进程也不一定。事实上，父进程可以在 **CreateProcess** 函数返回之后立即关闭它的内核对象句柄，子进程照样可以操纵这个对象。

表 3-3 显示了子进程在被允许开始执行之前的句柄表。可以看出，第一项和第二项没有初始化，所以对子进程来说无效的句柄，不可以使用。但是，索引 3 标识了一个内核对象。事实上，它标识的是地址 0xF0000010 处的内核对象，与父进程句柄表中的对象一样。

表 3-3 继承了父进程的“可继承的句柄”之后，子进程的句柄表的样子

索引	指向内核对象内存块的指针	访问掩码（包含标志位的一个 DWORD）	标志
1	0x00000000	(不可用)	(不可用)
2	0x00000000	(不可用)	(不可用)
3	0xF0000010	0x????????	0x00000001

第 13 章要讲到，内核对象的内容被保存在内核地址空间中——系统上运行的所有进程都共享这个空间。对于 32 位系统，这是 0x80000000 到 0xFFFFFFFF 之间的内存空间。对于 64 位系统，则是 0x00000400’00000000 到 0xFFFFFFF’FFFFFFFF 之间的内存空间。访问掩码与父进程中的一样，标志也是一样的。这意味着假如子进程用 CreateProcess 来生成它自己的子进程（其父的孙进程），那么在将 **bInheritHandles** 参数设为 TRUE 的前提下，孙进程也会继承这个内核对象句柄。在孙进程的句柄表中，继承的对象句柄将具有相同的句柄值，相同的访问掩码，以及相同的标志。内核对象的使用计数将再次递增。

记住，对象句柄的继承只会在生成子进程的时候发生。假如父进程后来又创建了新的内核对象，并同样将它们的句柄设为可继承的句柄。那么正在运行的子进程是不会继承这些新句柄的。

对象句柄继承还有一个非常奇怪的特征：子进程并不知道自己继承了任何句柄。在子进程的文档中，应指出当它从另一个进程生成时，希望获得对一个内核对象的访问权——只有在这种情况下，内核对象的句柄继承才是有用的。通常，父和子应用程序是由同一家公司编写的；但是，假如在公司的文档中，已经指出子应用程序有这方面的期待，另一家公司就可以据此来编写一个子应用程序。

到目前为止，子进程为了判断自己期望的一个内核对象的句柄值，最常见的方式是将句柄值作为命令行参数传给子进程。子进程的初始化代码将解析命令行（通常是调用**_stscanf_s**），并提取句柄值。子进程获得句柄值之后，就会拥有和父进程一样的内核对象访问权限。注意，句柄继承之所以能够实现，惟一的原因就是“共享的内核对象”的句柄值在父进程和子进程中是完全一样的。这正是父进程能将句柄值作为命令行参数来传递的原因。

当然，也可以使用其他进程间通信技术将继承的内核对象句柄值从父进程传入子进程。一个技术是让父进程等待子进程完成初始化（利用第 9 章讨论的 **WaitForInputIdle** 函数）；然后，父进程可以将一条消息 send 或 post 到由子进程中的一个线程创建的一个窗口。

另一种方式是让父进程向其环境块添加一个环境变量。变量的名称应该是子进程知道去查找的一个名称，而变量的值应该是准备被子进程继承的那个内核对象的句柄值。然后，当父进程生成子进程的时候，这个子进程会继承父进程的环境变量，所以能轻松调用 **GetEnvironmentVariable** 来获得这个继承到的内核对象的句柄值。如果子进程还要生成另一个子进程，这种方式就非常不错，因为环境变量是可以反复继承的。Microsoft 知识库的一篇文章（网址为 *http://support.microsoft.com/kb/190351*）描述了子进程继承父控制台的特例。（译者友情提示：不要看 Microsoft 自动“机器翻译”的版本。）

3.3.2 改变句柄的标志

有时可能遇到这样一种情况：父进程创建了一个内核对象，得到了一个可继承的句柄，然后生成了两个子进程。但是，父进程只希望其中的一个子进程继承内核对象句柄。换言之，你有时可能想控制哪些子进程能继承内核对象句柄。可以调用 **SetHandleInformation** 函数来改变内核对象句柄的继承标志。如下所示：

```
BOOL SetHandleInformation(
  HANDLE hObject,
  DWORD dwMask,
  DWORD dwFlags);
```

可以看出，这个函数有 3 个参数。第一个参数 **hObject** 标识了一个有效的句柄。第二个参数 **dwMask** 告诉函数你想更改哪个或者哪些标志。目前，每个句柄都关联了两个标志：

```
#define HANDLE_FLAG_INHERIT             0x00000001
#define HANDLE_FLAG_PROTECT_FROM_CLOSE  0x00000002
```

如果每个对象的标志都想一次性更改完毕，可以对这两个标志执行一次按位 OR 运算。**SetHandleInformation** 函数的第三个参数 **dwFlags** 指出希望把标志设为什么。例如，要打开一个内核对象句柄的继承标志，可以像下面这样写：

```
SetHandleInformation(hObj, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT);
```

要关闭这个标志，可以像下面这样写：

```
SetHandleInformation(hObj, HANDLE_FLAG_INHERIT, 0);
```

HANDLE_FLAG_PROTECT_FROM_CLOSE 标志告诉系统不允许关闭句柄：

```
SetHandleInformation(hObj, HANDLE_FLAG_PROTECT_FROM_CLOSE,
    HANDLE_FLAG_PROTECT_FROM_CLOSE);
CloseHandle(hObj); // 会引发异常
```

如果在调试器下运行，一旦线程试图关闭一个受保护的句柄，**CloseHandle** 就会引发一个异常。如果在调试器的控制之外，**CloseHandle** 只是返回 **FALSE**。很少有必要阻止句柄被关闭。但是，如果你的一个进程会生成一个子进程，后者再生成一个孙进程，那么这个标志还是有用的。父进程可能希望孙进程继承自己拿给子进程的对象句柄。但是，子进程可能在生成孙进程之前就关闭了那个句柄。如果发生这种情况，父进程就不能和孙进程通信了，因为可怜的孙进程根本没有继承到内核对象（句柄）。相反，如果将句柄标记为“PROTECT FROM CLOSE”（禁止关闭），孙进程就有更大的机会继承到指向一个有效的、活动的内核对象的句柄。

不过，正如我上一句话说的，孙进程现在只是“机会”更大一些。这种方式的不足之处在于，处于中间位置的那个子进程可以调用以下代码来关闭 **HANDLE_FLAG_PROTECT_FROM_CLOSE** 标志，然后关闭句柄：

```
SetHandleInformation(hobj, HANDLE_FLAG_PROTECT_FROM_CLOSE, 0);
CloseHandle(hObj);
```

也就是说，父进程其实是在**赌**自己的子进程不会执行上述代码。但即使没有这个问题，子进程真的就会生成孙进程吗？这同样是在**赌**。反正都是赌，第一个赌看起来也就没有那么危险了。

考虑到内容的完整性，下面再来讨论一下 **GetHandleInformation** 函数：

```
BOOL GetHandleInformation(
  HANDLE hObject,
  PDWORD pdwFlags);
```

执行这个函数，将在由 pdwFlags 指向的一个 DWORD 中返回指定句柄的当前标志设置。要检查一个句柄是不是可以继承的，请执行以下代码：

```
DWORD dwFlags;
GetHandleInformation(hObj, &dwFlags);
BOOL fHandleIsInheritable = (0 != (dwFlags & HANDLE_FLAG_INHERIT));
```

3.3.3 为对象命名

跨进程边界共享内核对象的第二个办法是为对象命名。许多（但不是全部）内核对象都可以进行命名。例如，以下所有函数都可以创建命名的内核对象：

```
HANDLE CreateMutex(
  PSECURITY_ATTRIBUTES psa,
  BOOL bInitialOwner,
  PCTSTR pszName);

HANDLE CreateEvent(
  PSECURITY_ATTRIBUTES psa,
  BOOL bManualReset,
  BOOL bInitialState,
  PCTSTR pszName);

HANDLE CreateSemaphore(
  PSECURITY_ATTRIBUTES psa,
  LONG lInitialCount,
  LONG lMaximumCount,
  PCTSTR pszName);

HANDLE CreateWaitableTimer(
  PSECURITY_ATTRIBUTES psa,
  BOOL bManualReset,
  PCTSTR pszName);

HANDLE CreateFileMapping(
  HANDLE hFile,
```

```
  PSECURITY_ATTRIBUTES psa,
  DWORD flProtect,
  DWORD dwMaximumSizeHigh,
  DWORD dwMaximumSizeLow,
  PCTSTR pszName);

HANDLE CreateJobObject(
  PSECURITY_ATTRIBUTES psa,
  PCTSTR pszName);
```

所有这些函数的最后一个参数都是 **pszName**。向此参数传入 **NULL**，相当于向系统表明你要创建一个未命名的（即匿名）内核对象。如果创建的是一个无名对象，可以利用上一节讨论过的继承技术，或者利用下一节即将讨论的 **DuplicateHandle** 函数来实现进程间的对象共享。如果要根据对象名称来共享一个对象，你必须为此对象指定一个名称。

如果不为 **pszName** 参数传递 **NULL**，则应该传入一个“以 0 来终止的名称字符串”的地址。这个名称可以长达 **MAX_PATH** 个字符（定义为 260）。遗憾的是，Microsoft 没有提供任何专门的机制来保证为内核对象指定的名称是惟一的，。例如，假如你试图创建一个名为"JeffObj"的对象，那么没有任何一种机制来保证当前不存在一个名为"JeffObj"的对象。更糟的是，所有这些对象都共享同一个命名空间，即使它们的类型并不相同。例如，以下 **CreateSemaphore** 函数调用肯定会返回 **NULL**，因为已经有一个同名的 mutex 对象了：

```
HANDLE hMutex = CreateMutex(NULL, FALSE, TEXT("JeffObj"));
HANDLE hSem = CreateSemaphore(NULL, 1, 1, TEXT("JeffObj"));
DWORD dwErrorCode = GetLastError();
```

执行上述代码之后，如果检查 **dwErrorCode** 的值，会发现返回的代码为 6（**ERROR_INVALID_HANDLE**）。这个错误代码当然说明不了什么问题，不过我们目前对此无能为力。

知道如何命名对象之后，接着来看看如何以这种方式共享对象。假设 Process A 启动并调用以下函数：

```
HANDLE hMutexProcessA = CreateMutex(NULL, FALSE, TEXT("JeffMutex"));
```

这个函数调用创建一个新的 mutex 内核对象，并将其命名为"JeffMutex"。注意，在 Process A 的句柄（表）中，**hMutexProcessA** 并不是一个可继承的句柄——但是，通过为对象命名来实现共享时，是否可以继承并非一个必要条件。

在以后某个时间，假定某个进程生成了 Process B。Process B 不一定是 Process A 的子进程；它可能是从 Windows 资源管理器或者其他某个应用程序生成的。利用对象的名称（而非利用继承）来共享内核对象时，最大的一个优势就是“Process B 不一定是 Process A 的子进程”。Process B 开始执行时，它执行以下代码：

```
HANDLE hMutexProcessB = CreateMutex(NULL, FALSE, TEXT("JeffMutex"));
```

当 Process B 发出对 **CreateMutex** 的调用时，系统首先会查看是否存在一个名为“JeffMutex”的内核对象。由于确实存在这样的一个对象，所以内核接着检查对象的类型。由于试图创建一个 mutex，而名为“JeffMutex”的对象也是一个 mutex，所以系统接着执行一次安全检查，验证调用者是否拥有对象的完全访问权限。如果答案是肯定的，系统就会在 Process B 的句柄表中查找一个空白记录项，并将其初始化为指向现有的内核对象。如果对象的类型不匹配，或调用者被拒绝访问，**CreateMutex** 就会失败（返回 **NULL**）。

注意
用于创建内核对象的函数（比如 CreateSemaphore）总是返回具有完全访问权限的句柄。如果想限制一个句柄的访问权限，可以使用这些函数的扩展版本（有一个 Ex 后缀），它们接受一个额外的 DWORD dwDesiredAccess 参数。例如，可以在调用 CreateSemaphoreEx 时使用或不使用 SEMAPHORE_MODIFY_STATE，从而允许或禁止对一个 semaphore 句柄调用 ReleaseSemaphore。请阅读 Windows SDK 文档，了解与每种内核对象的权限细节，网址是 *http://msdn2.microsoft.com/en-us/library/ms686670.aspx*。

Process B 调用 **CreateMutex** 成功之后，不会实际地创建一个 mutex。相反，会为 Process B 分配一个新的句柄值（当然，和所有句柄值一样，这是一个相对于该进程的句柄值），它标识了内核中的一个现有的 mutex 对象。当然，由于在 Process B 的句柄表中，用一个新的记录项来引用了这个对象，所以这个 mutex 对象的使用计数会被递增。在 Process A 和 Process B 都关闭这个对象的句柄之前，该对象是不会销毁的。注意，两个进程中的句柄值极有可能是不同的值。这没有什么关系。Process A 用它自己的句柄值来引用那个 mutex 对象，Process B 也用它自己的句柄值来引用同一个 mutex 对象。

注意
让内核对象通过名称来实现共享的时候，务必关注一点：Process B 调用 CreateMutex 时，它会向函数传递安全属性信息和第二个参数。如果已经存在一个指定名称的对象，这些参数就会被忽略。（译者注：换言之，函数不知道自己刚才是新建了一个对象，还是打开了一个现有的对象。）
事实上，完全可以在调用了 Create*之后，马上调用一个 GetLastError，判断自己刚才是真的创建了一个新的内核对象，还是仅仅是打开了一个现有的：

```
HANDLE hMutex = CreateMutex(&sa, FALSE, TEXT("JeffObj"));
if (GetLastError() == ERROR_ALREADY_EXISTS) {
    // Opened a handle to an existing object.
    // sa.lpSecurityDescriptor and the second parameter
    // (FALSE) are ignored.
} else {
    // Created a brand new object.
    // sa.lpSecurityDescriptor and the second parameter
    // (FALSE) are used to construct the object.
}
```

为对象分配了名称之后，为了实现这些对象的共享，还有另一个办法可供考虑。你可以不调用 **Create***函数；相反，可以调用如下所示的一个 **Open***函数：

```
HANDLE OpenMutex(
```

```
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenEvent(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenSemaphore(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenWaitableTimer(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenFileMapping(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenJobObject(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);
```

注意，所有这些函数的原型都是一样的。最后一个参数 **pszName** 指出内核对象的名称。不能为这个参数传入 **NULL**，必须传入一个以 0 终止的字符串的地址。这些函数将在同一个内核对象命名空间搜索，以查找一个匹配的对象。如果没有找到这个名称的一个内核对象，函数将返回 **NULL**，**GetLastError** 将返回 2（**ERROR_FILE_NOT_FOUND**）。如果找到了这个名称的一个内核对象，但类型不对，函数将返回 **NULL**，**GetLastError** 将返回 6（**ERROR_INVALID_HANDLE**）。如果名称对，类型也对，系统会检查请求的访问（通过 **dwDesiredAccess** 来指定）是否允许。如果允许，主调进程的句柄表就会更新，对象的使用计数会递增。如果为 **bInheritHandle** 参数传入了 TRUE，那么返回的句柄就是“可继承”的。

调用 **Create***函数和调用 **Open***函数的主要区别在于，如果对象不存在，**Create***函数会创建它；**Open***函数则不同，如果对象不存在，它只是简单地以调用失败而告终。

如前所述，Microsoft 没有提供任何专门的机制来保证我们创建独一无二的对象名。换言之，如果用户试图运行来自不同公司的两个程序，而且每个程序都试图创建一个名为“MyObject”的对象，那么就会出问题。为了确保名称的惟一性，我的建议是创建一个 GUID，

并将这个 GUID 的字符串形式作为自己的对象名称使用。在稍后的 3.3.4 节“private 命名空间”中，介绍了另一种保证名称惟一性的方式。

我们经常利用命名的对象来防止运行一个应用程序的多个实例。为此，只需在你的**_tmain** 或**_tWinMain** 函数中调用一个 **Create***函数来创建一个命名的对象（具体创建什么类型无关紧要）。**Create*** 函数返回后，再调用一下 **GetLastError**。如果 **GetLastError** 返回 **ERROR_ALREADY_EXISTS**，表明你的应用程序的另一个实例正在运行，新的实例就可以退出了。以下代码对此进行了说明：

```
int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR pszCmdLine,
  int nCmdShow) {
  HANDLE h = CreateMutex(NULL, FALSE,
    TEXT("{FA531CC1-0497-11d3-A180-00105A276C3E}"));
  if (GetLastError() == ERROR_ALREADY_EXISTS) {
    // There is already an instance of this application running.
    // Close the object and immediately return.
    CloseHandle(h);
    return(0);
  }

  // This is the first instance of this application running.
  ...
  // Before exiting, close the object.
  CloseHandle(h);
  return(0);
}
```

3.3.4 Terminal Services 命名空间

注意，Terminal Services（终端服务）的情况和前面描述的稍微有所区别。在正在运行 Terminal Services 的计算机中，有多个用于内核对象的命名空间。其中一个是全局命名空间，所有客户端都能访问的内核对象要放在这个命名空间中。这个命名空间主要由服务使用。此外，每个客户端会话（client session）都有一个自己的命名空间。采取这个安排之后，可以避免正在运行同一个应用程序的两个或多个会话彼此干扰——一个会话不能访问另一个会话的对象，即使对象的名称相同。

并非只有服务器才会遇到这种情况，因为 Remote Desktop（远程桌面）和 Fast User Switching（快速用户切换）特性也是利用 Terminal Services 会话来实现的。

注意

在任何用户都没有登录的时候，服务会在第一个会话（称为 Session 0）中启动，这个会话是非交互式的。和以前版本的 Windows 不同，在 Windows Vista 中，只要用户登录，应用程序就会在一个新的会话（和服务专用的 Session 0 不同的一个会话）中启动。采用这个设计之后，系统核心组件（通常具有较高的权限）就可以更好地与用户不慎启动的恶意软件

（malware）隔离。对于服务开发人员，由于必须在和客户端应用程序不同的一个会话中运行，所以会影响到共享内核对象的命名约定。任何对象要想和用户应用程序共享，就必须在全局命名空间中创建它。“快速用户切换”也会带来类似的问题。我们知道，利用“快速用户切换”功能，不同的用户可以登录不同的会话，并分别启动自己的用户应用程序。如果你写的一个服务要同这些应用程序通信，就不能假定它和用户应用程序在同一个会话中运行。要想更多地了解 Session 0 隔离问题，及其对服务开发人员的影响，请阅读“Impact of Session 0 Isolation on Services and Drivers in Windows Vista”一文，网址是 *http://www.microsoft.com/whdc/system/vista/services.mspx*。

如果必须知道你的进程在哪个 Terminal Services 会话中运行，可以借助于 ProcessIdToSessionId 函数（由 kernel32.dll 导入，在 WinBase.h 中声明），如下例所示：

```
DWORD processID = GetCurrentProcessId();
DWORD sessionID;
if (ProcessIdToSessionId(processID, &sessionID)) {
   tprintf(
     TEXT("Process '%u' runs in Terminal Services session '%u'"),
     processID, sessionID);
} else {
   // ProcessIdToSessionId might fail if you don't have enough rights
   // to access the process for which you pass the ID as parameter.
   // Notice that it is not the case here because we're using our own process ID.
   tprintf(
     TEXT("Unable to get Terminal Services session ID for process '%u'"),
     processID);
}
```

一个服务的命名内核对象始终是在全局命名空间内的。默认情况下，在 Terminal Services 中，应用程序自己的命名内核对象在会话的命名空间内。不过，也可以强迫一个命名的对象进入全局命名空间，具体做法是在其名称前加上“Global\”前缀，如下面的例子所示：

```
HANDLE h = CreateEvent(NULL, FALSE, FALSE, TEXT("Global\\MyName"));
```

也可以显式指出你希望一个内核对象进入当前会话的命名空间，具体做法式在名称前加上“Local\”前缀，如下面的例子所示：

```
HANDLE h = CreateEvent(NULL, FALSE, FALSE, TEXT("Local\\MyName"));
```

Microsoft 认为 Global 和 Local 是保留关键字，所以除非为了强制一个特定的命名空间，否则不应在对象名称中使用它们。Microsoft 还认为 Session 是保留关键字。所以（举一个例子），你可以使用 Session\<*当前会话 ID*>\。不过，你不能使用另一个会话中的名称和 Session 前缀来新建一个对象，这样会导致函数调用失败，**GetLastError** 会返回 **ERROR_ACCESS_DENIED**。

注意
所有保留关键字都是区分大小写的。

3.3.4 private 命名空间

创建内核对象时，可以传递指向一个 **SECURITY_ATTRIBUTES** 结构的指针，从而保护对该对象的访问。不过，在 Windows Vista 发布之前，你不可能防范一个共享对象的名称被“劫持”。任何进程——即使是最低权限的进程——都能用任何指定的名称来创建一个对象。以前面的例子为例（应用程序用一个命名的 mutex 对象来检测该程序的一个实例是否正在运行），很容易另外写一个应用程序来创建一个同名的内核对象。如果它先于单实例应用程序启动，“单实例”的应用程序就变成了一个“无实例”的应用程序——始终都是一启动就退出，错误地认为它自己的另一个实例已在运行。这是大家熟悉的几种拒绝服务（DoS）攻击的基本机制。注意，未命名的内核对象不会遭受 DoS 攻击。另外，应用程序使用未命名对象是相当普遍的，即使这些对象不能在进程之间共享。

如果想确保你的应用程序创建的内核对象名称永远不会和其他应用程序的名称冲突，或者想确保它们不会成为劫持攻击的目标，那么可以定义一个自定义的前缀，并把它作为自己的 private 命名空间使用，这和使用 Global 和 Local 前缀是相似的。负责创建内核对象的服务器进程将定义一个边界描述符（boundary descriptor），它对命名空间的名称自身进行保护。

Singleton 应用程序（即 03-Singleton.exe，对应的 Singleton.cpp 源代码将在稍后列出）展示了如何利用 private 命名空间，以一种更安全的方式来实现前面描述的单实例应用程序。（Singleton 是单身的意思，即同时只允许该程序的一个实例）。启动程序后，会出现如图 3-5 所示的窗口。

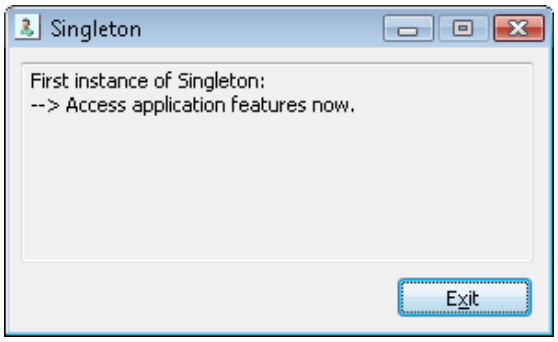


图 3-5　Singleton 的第一个实例正在运行

保持程序的运行状态，然后启动同一个程序，就会出现如图 3-6 所示的窗口，它指出已经检测到了程序另一个实例。

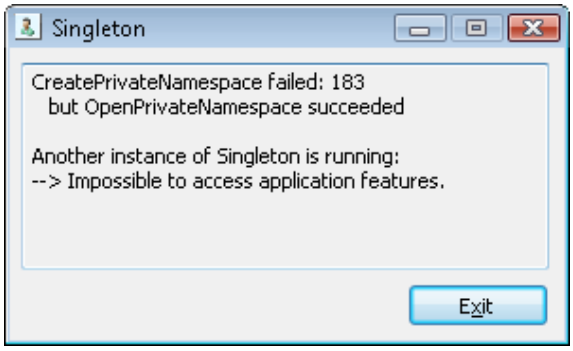


图 3-6　第一个实例正在运行，又启动了 Singleton 的第二个实例

研究一下 Singleton.cpp 源代码中的 CheckInstances 函数，你可以理解三个“如何”：(1)如何创建一个边界（bondary）；(2)如何将对应于 Local Administrators 组的一个安全描述符（security identifier，SID）和它关联起来；(3)如何创建或打开其名称要作为 mutex 内核对象前缀使用的一个 private 命名空间。边界描述符将获得一个名称，但更重要的是，它会获得与它关联的一个特权用户组的 SID。这样一来，对于在任何用户的上下文中运行的应用程序，只有在该用户是这个特权组的一个成员的前提下，应用程序才能在相同的边界中创建相同的命名空间，并因而能访问这个边界中创建的、以 private 命名空间的名称作为前缀的内核对象。

如果由于名称和 SID 泄密，导致一个低特权的恶意程序创建了相同的边界描述符，那么（举个例子来说）当它试图创建或打开使用一个高特权帐户保护的 private 命名空间时，调用就会失败，**GetLastError** 会返回 **ERROR_ACCESS_DENIED**。如果恶意程序有足够的权限来创建或打开命名空间，再担心这个就多余了——因为恶意程序有足够的能力来造成更大的破坏，而非仅仅是劫持一个内核对象名称那么简单。

Singleton.cpp

```
/******************************************************************************
Module:  Singleton.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "resource.h"

#include "..\CommonFiles\CmnHdr.h"    /* See Appendix A. */
#include <windowsx.h>
#include <Sddl.h>        // for SID management
#include <tchar.h>
#include <strsafe.h>



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


// Main dialog
HWND    g_hDlg;

// Mutex, boundary and namespace used to detect previous running instance
HANDLE   g_hSingleton = NULL;
HANDLE   g_hBoundary = NULL;
HANDLE   g_hNamespace = NULL;
```

```
// Keep track whether or not the namespace was created or open for clean-up
BOOL    g_bNamespaceOpened = FALSE;

// Names of boundary and private namespace
PCTSTR  g_szBoundary = TEXT("3-Boundary");
PCTSTR  g_szNamespace = TEXT("3-Namespace");


#define DETAILS_CTRL GetDlgItem(g_hDlg, IDC_EDIT_DETAILS)


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


// Adds a string to the "Details" edit control
void AddText(PCTSTR pszFormat, ...) {

   va_list argList;
   va_start(argList, pszFormat);

   TCHAR sz[20 * 1024];

   Edit_GetText(DETAILS_CTRL, sz, _countof(sz));
   _vstprintf_s(
      _tcschr(sz, TEXT('\0')), _countof(sz) - _tcslen(sz),
      pszFormat, argList);
   Edit_SetText(DETAILS_CTRL, sz);
   va_end(argList);
}


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


void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {
      case IDOK:
      case IDCANCEL:
         // User has clicked on the Exit button
         // or dismissed the dialog with ESCAPE
         EndDialog(hwnd, id);
         break;
   }
```

```
}

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

void CheckInstances() {

  // Create the boundary descriptor
  g_hBoundary = CreateBoundaryDescriptor(g_szBoundary, 0);

  // Create a SID corresponding to the Local Administrator group
  BYTE localAdminSID[SECURITY_MAX_SID_SIZE];
  PSID pLocalAdminSID = &localAdminSID;
  DWORD cbSID = sizeof(localAdminSID);
  if (!CreateWellKnownSid(
    WinBuiltinAdministratorsSid, NULL, pLocalAdminSID, &cbSID)
    ) {
    AddText(TEXT("AddSIDToBoundaryDescriptor failed: %u\r\n"),
      GetLastError());
    return;
  }

  // Associate the Local Admin SID to the boundary descriptor
  // --> only applications running under an administrator user
  //    will be able to access the kernel objects in the same namespace
  if (!AddSIDToBoundaryDescriptor(&g_hBoundary, pLocalAdminSID)) {
    AddText(TEXT("AddSIDToBoundaryDescriptor failed: %u\r\n"),
      GetLastError());
    return;
  }

  // Create the namespace for Local Administrators only
  SECURITY_ATTRIBUTES sa;
  sa.nLength = sizeof(sa);
  sa.bInheritHandle = FALSE;
  if (!ConvertStringSecurityDescriptorToSecurityDescriptor(
    TEXT("D:(A;;GA;;;BA)"),
    SDDL_REVISION_1, &sa.lpSecurityDescriptor, NULL)) {
    AddText(TEXT("Security Descriptor creation failed: %u\r\n"), GetLastError());
    return;
  }

  g_hNamespace =
```

```
   CreatePrivateNamespace(&sa, g_hBoundary, g_szNamespace);

// Don't forget to release memory for the security descriptor
LocalFree(sa.lpSecurityDescriptor);



// Check the private namespace creation result
DWORD dwLastError = GetLastError();
if (g_hNamespace == NULL) {
   // Nothing to do if access is denied
   // --> this code must run under a Local Administrator account
   if (dwLastError == ERROR_ACCESS_DENIED) {
      AddText(TEXT("Access denied when creating the namespace.\r\n"));
      AddText(TEXT("   You must be running as Administrator.\r\n\r\n"));
      return;
   } else {
      if (dwLastError == ERROR_ALREADY_EXISTS) {
      // If another instance has already created the namespace,
      // we need to open it instead.
         AddText(TEXT("CreatePrivateNamespace failed: %u\r\n"), dwLastError);
         g_hNamespace = OpenPrivateNamespace(g_hBoundary, g_szNamespace);
         if (g_hNamespace == NULL) {
            AddText(TEXT("   and OpenPrivateNamespace failed: %u\r\n"),
            dwLastError);
            return;
         } else {
            g_bNamespaceOpened = TRUE;
            AddText(TEXT("   but OpenPrivateNamespace succeeded\r\n\r\n"));
         }
      } else {
         AddText(TEXT("Unexpected error occured: %u\r\n\r\n"),
            dwLastError);
         return;
      }
   }
}

// Try to create the mutex object with a name
// based on the private namespace
TCHAR szMutexName[64];
StringCchPrintf(szMutexName, _countof(szMutexName), TEXT("%s\\%s"),
   g_szNamespace, TEXT("Singleton"));

g_hSingleton = CreateMutex(NULL, FALSE, szMutexName);
```

```
   if (GetLastError() == ERROR_ALREADY_EXISTS) {
      // There is already an instance of this Singleton object
      AddText(TEXT("Another instance of Singleton is running:\r\n"));
      AddText(TEXT("--> Impossible to access application features.\r\n"));
   } else  {
      // First time the Singleton object is created
      AddText(TEXT("First instance of Singleton:\r\n"));
      AddText(TEXT("--> Access application features now.\r\n"));
   }
}

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

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_SINGLETON);

   // Keep track of the main dialog window handle
   g_hDlg = hwnd;

   // Check whether another instance is already running
   CheckInstances();

   return(TRUE);
}

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

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
   }

   return(FALSE);
}

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

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
                  HINSTANCE hPrevInstance,
                  LPTSTR    lpCmdLine,
                  int      nCmdShow)
{
   UNREFERENCED_PARAMETER(hPrevInstance);
   UNREFERENCED_PARAMETER(lpCmdLine);

   // Show main window
   DialogBox(hInstance, MAKEINTRESOURCE(IDD_SINGLETON), NULL, Dlg_Proc);

   // Don't forget to clean up and release kernel resources
   if (g_hSingleton != NULL) {
      CloseHandle(g_hSingleton);
   }

   if (g_hNamespace != NULL) {
      if (g_bNamespaceOpened) {  // Open namespace
         ClosePrivateNamespace(g_hNamespace, 0);
      } else { // Created namespace
         ClosePrivateNamespace(g_hNamespace, PRIVATE_NAMESPACE_FLAG_DESTROY);
      }
   }

   if (g_hBoundary != NULL) {
      DeleteBoundaryDescriptor(g_hBoundary);
   }

   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

下面来分析一下 **CheckInstances** 函数的几个不同的步骤。第一步是创建边界描述符。为此，你需要用一个字符串标识符来命名一个范围，private 命名空间将在这个范围中定义。这个名称作为以下函数的第一个参数进行传递：

```
HANDLE CreateBoundaryDescriptor(
   PCTSTR pszName,
   DWORD dwFlags);
```

在当前版本的 Windows 中，第二个参数还没有什么用，因此应该为它传入 0。注意，函数的签名暗示返回值是一个内核对象句柄，但实情并非如此。返回的是一个指针，它指向一个用户模式的结构，结构中包含了边界的定义。由于这个原因，永远都不要把返回的句柄值传给 **CloseHandle**；相反，应该把它传给 **DeleteBoundaryDescriptor**。

第二步，通过调用以下函数，将一个特权用户组的 SID（客户端应用程序将在这些用户的上下文中运行）与边界描述符关联起来：

```
BOOL AddSIDToBoundaryDescriptor(
  HANDLE* phBoundaryDescriptor,
  PSID pRequiredSid);
```

在 Singleton 示例程序中，为了创建 Local Administrator 组的 SID，我们的办法是调用 **AllocateAndInitializeSid** 函数，并将用于描述这个组的 **SECURITY_BUILTIN_DOMAIN_RID** 和 **DOMAIN_ALIAS_RID_ADMINS** 作为参数来传递。在 WinNT.h 头文件中，定义了所有已知的组的一个列表。

调用以下函数来创建 private 命名空间时，边界描述符（伪）句柄作为第二个参数传给该函数：

```
HANDLE CreatePrivateNamespace(
  PSECURITY_ATTRIBUTES psa,
  PVOID pvBoundaryDescriptor,
  PCTSTR pszAliasPrefix);
```

作为第一个参数传给该函数的 **SECURITY_ATTRIBUTES** 是供 Windows 使用的，用于允许或禁止一个应用程序通过调用 **OpenPrivateNamespace** 来访问命名空间并在其中打开/创建对象。具体可以使用的选项和“在一个文件系统的目录中可以使用的选项”是完全一样的。这是你为“打开命名空间”设置的一个筛选层。为边界描述符添加的 SID 决定了谁能进入边界并创建命名空间。在 Singleton 例子中，**SECURITY_ATTRIBUTE** 是通过调用 **ConvertStringSecurityDescriptorToSecurityDescriptor** 函数来构造的。该函数获取一个具有复杂语法结构的字符串作为第一个参数。要具体了解安全描述符字符串的语法，请参考 *http://msdn2.microsoft.com/en-us/library/aa374928.aspx* 和 *http://msdn2.microsoft.com/en-us/library/aa379602.aspx*。
pvBoundaryDescriptor 的类型是 **VOID**，虽然 **CreateBoundaryDescriptor** 返回的是一个 **HANDLE**（虽然在 Microsoft 那里，这不是一个内核对象句柄，而是一个伪句柄）。用于创建内核对象的字符串前缀被指定为第三个参数。如果试图创建一个已经存在的 private 命名空间，**CreatePrivateNamespace** 将返回 **NULL**，**GetLastError** 将返回 **ERROR_ALREADY_EXISTS**。在这种情况下，需要使用以下函数来打开现有的 private 命名空间：

```
HANDLE OpenPrivateNamespace(
  PVOID pvBoundaryDescriptor,
  PCTSTR pszAliasPrefix);
```

注意，**CreatePrivateNamespace** 和 **OpenPrivateNamespace** 返回的 **HANDLE** 并不是内核对象句柄；可以调用 **ClosePrivateNamespace** 来关闭它们返回的这种伪句柄：

```
BOOLEAN ClosePrivateNamespace(
  HANDLE hNamespace,
  DWORD dwFlags);
```

如果你已经创建了命名空间，而且不希望它在关闭后仍然可见，应该将 **PRIVATE_NAMESPACE_FLAG_DESTROY** 作为第二个参数传给上述函数，反之则传入 0。边界将在以下两种情况下关闭：进程终止运行；或者调用 **DeleteBoundaryDescriptor**，并将边界伪句柄作为惟一的参数传给它。如果还有内核对象正在使用，命名空间一定不能关闭。如果在内部还有内核对象时关闭了一个命名空间，就可以在同一个边界中，在重新创建的一个相同的命名空间中，创建一个同名的内核对象，使 DoS 攻击再次成为可能。

总之，private 命名空间相当于可供你在其中创建内核对象的一个目录。和其他目录一样，private 命名空间也有一个和它关联的安全描述符，这个描述符是在调用 **CreatePrivateNamespace** 的时候设置的。但是，和文件系统的目录不同的是，这个命名空间是没有父目录的，也没有名称——我们将“边界描述符”作为对这个边界进行引用的一个名称来使用。正是由于这个原因，所以在 Sysinternals 的 Process Explorer 中，对于“前缀以一个 private 命名空间为基础”的内核对象，它们显示的是“…\”前缀，而不是你预期的“*namespace name*\”前缀。“…\”前缀隐藏了需要保密的信息，能更好地防范潜在的黑客。你为 private 命名空间指定的名称是一个别名，只在进程内可见。其他进程（甚至是同一个进程）可以同一个 private 命名空间，并为它指定一个不同的别名。

创建普通目录时，需要对父目录执执行一次访问检查，确定是否能在其中创建一个子目录。类似地，为了创建命名空间，要执行一次边界测试——在当前线程的令牌（token）中，必须包含作为边界一部分的所有 SID。

3.3.5 复制对象句柄

为了跨越进程边界来共享内核对象，最后一个技术是使用 **DuplicateHandle** 函数：

```
BOOL DuplicateHandle(
  HANDLE hSourceProcessHandle,
  HANDLE hSourceHandle,
  HANDLE hTargetProcessHandle,
  PHANDLE phTargetHandle,
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  DWORD dwOptions);
```

简单地说，这个函数获得一个进程的句柄表中的一个记录项，然后在另一个进程的句柄表中创建这个记录项的一个拷贝。**DuplicateHandle** 需要获取几个参数，但它的工作过程实际是

非常简单的。正如本节稍后的例子演示的那样，该函数最常见的一种用法可能涉及到系统中同时运行的三个不同的进程。

调用 **DuplicateHandle** 时，它的第一个参数和第三个参数——**hSourceProcessHandle** 和 **hTargetProcessHandle**——是内核对象句柄。这两个句柄本身必须相对于调用 **DuplicateHandle** 函数的那个进程。此外，这两个参数标识的必须是**进程内核对象**；如果你传递的句柄指向的是其他类型的内核对象，函数调用就会失败。我们将在第 4 章详细讨论进程内核对象。就目前来说，你只需知道一旦在系统中调用了一个新的进程，就会创建一个进程内核对象。

第二个参数 **hSourceHandle** 是指向任何类型的内核对象的一个句柄。但是，它的句柄值一定不能相对于调用 **DuplicateHandle** 函数的那个进程。相反，该句柄必须相对于 **hSourceProcessHandle** 句柄所标识的那个进程。第四个参数是 **phTargetHandle**，它是一个 **HANDLE** 变量的地址；在 hTargetProcessHandle（第四个参数）标识的目标进程的句柄表中，会拷贝源进程（第一个参数）中的一个源内核对象句柄（第二个参数）的句柄信息。这些句柄信息会拷贝到句柄表的一个记录项中。而在第四个参数 phTargetHandle 指向的那个变量中，将接收与这个记录项对应的句柄值。

DuplicateHandle 的最后三个参数用于指定访问掩码和继承标志；在目标进程的句柄表中，将在与这个内核对象句柄对应的记录项中使用它们。**dwOptions** 参数可以为 0（零）或者以下两个标志的任意组合：**DUPLICATE_SAME_ACCESS** 和 **DUPLICATE_CLOSE_SOURCE**。

如果指定 **DUPLICATE_SAME_ACCESS** 标志，将向 **DuplicateHandle** 函数表明我们希望目标句柄拥有与源进程的句柄一样的访问掩码。使用这个标志后，**DuplicateHandle** 函数会忽略它的 **dwDesiredAccess** 参数。

如果指定 **DUPLICATE_CLOSE_SOURCE** 标志，会关闭源进程中的句柄。利用这个标志，一个进程可以轻松地将一个内核对象传给另一个进程。如果使用了这个标志，内核对象的使用计数不会受到影响。

我将用一个例子来演示 **DuplicateHandle** 函数的工作方式。在这个例子中，Process S 是源进程，它拥有对一个内核对象的访问权；Process T 是目标进程，它将获得对这个内核对象访问权。Process C 则是一个催化剂（catalyst）进程，它执行对 **DuplicateHandle** 的调用。在本例中，我是出于演示函数工作方式的目的才使用了硬编码（嵌入源代码）的句柄值。在实际应用中，应该将句柄值放到变量中，再将变量作为参数传给函数。

Process C 的句柄表（参见表 3-4）包含两个句柄值，即 1 和 2。句柄值 1 标识了 Process S 的**进程内核对象**，而句柄值 2 标识 Process T 的**进程内核对象**。

表 3-4　Process C 的句柄表

索引	指向内核对象内存块的指针	访问掩码（包含标志位的一个 DWORD）	标志
1	0xF0000000（与 Process S 对应的的内核对象）	0x????????	0x00000000

索引	指向内核对象内存块的指针	访问掩码(包含标志位的一个DWORD)	标志
2	0xF0000010（与 Process T 对应的内核对象）	0x????????	0x00000000

表 3-5 是 Process S 的句柄表，其中只包含一项，该项的句柄值为 2。这个句柄可以标识任何类型的内核对象——不一定是**进程内核对象**。

表 3-5　Process S 的句柄表

索引	指向内核对象内存块的指针	访问掩码(包含标志位的一个 DWORD)	标志
1	0x00000000	(不可用)	(不可用)
2	0xF0000020（任意内核对象）	0x????????	0x00000000

表 3-6 显示了在 Process C 调用 **DuplicateHandle** 函数之前，Process T 的句柄表所包含的内容。可以看出，Process T 的句柄表只包含一项，该项的句柄值为 2；句柄项 1 目前没有使用。

表 3-6　调用 DuplicateHandle 函数之前，Process T 的句柄表

索引	指向内核对象内存块的指针	访问掩码(包含标志位的一个 DWORD)	标志
1	0x00000000	(不可用)	(不可用)
2	0xF0000030（任意内核对象）	0x????????	0x00000000

如果 Process C 现在用以下代码来调用 **DuplicateHandle**，那么只有 Process T 的句柄表会发生变化，如表 3-7 所示：

```
DuplicateHandle(1, 2, 2, &hObj, 0, TRUE, DUPLICATE_SAME_ACCESS);
```

表 3-7　调用 DuplicateHandle 函数之后，Process T 的句柄表

索引	指向内核对象内存块的指针	访问掩码(包含标志位的一个 DWORD)	标志
1	0xF0000020	0x????????	0x00000001
2	0xF0000030（任意内核对象）	0x????????	0x00000000

在这个过程中，Process S 句柄表的第 2 项被复制到 Process T 的句柄表的第 1 项。**DuplicateHandle** 还用值 1 来填充了 Process C 的 **hObj** 变量；这个 1 是 Process T 的句柄表中的索引，新的记录项被复制到这个位置。

由于向 DuplicateHandle 函数传递了 DUPLICATE_SAME_ACCESS 标志，所以在 Process T 的句柄表中，这个句柄的访问掩码与 Process S 句柄表的那一个项的访问掩码是一样的。同时，**DUPLICATE_SAME_ACCESS** 标志会导致 **DuplicateHandle** 忽略其 **dwDesiredAccess** 参数。最后要注意，继承标志位已经被打开，因为已经为 **DuplicateHandle** 函数的 **bInheritHandle** 参数传入 **TRUE**。

使用 **DuplicateHandle** 函数（来复制内核对象句柄）会遇到和继承（内核对象句柄）时同样的一个问题：目标进程不知道它现在能访问一个新的内核对象。所以，Process C 必须以某种方式来通知 Process T，告诉它现在可以访问一个内核对象了，而且必须使用某种形式的进程间通信机制，将 hObj 中的句柄值传给 Process T。显然，使用命令行参数或者更改 Process T 的环境变量是行不通的，因为进程已经启动并开始运行了。你必须使用窗口消息或者其他

进程间通信（IPC）机制。

刚才介绍的是 **DuplicateHandle** 最常见的用法。可以看出，它是一个非常灵活的函数。不过，在涉及三个不同的进程时，它其实是很少使用的（可能是由于 Process C 不知道 Process S 使用的一个对象的句柄值）。通常，在只涉及到两个进程的时候，才会调用 **DuplicateHandle** 函数。来设想下面这个情景：一个进程能访问另一个进程也想访问的一个对象；或者说，一个进程想把一个内核对象的访问权授予另一个进程。例如，假定 Process S 能访问一个内核对象，并希望 Process T 也能访问这个对象，那么可以像下面这样调用 **DuplicateHandle** 函数：

```
// All of the following code is executed by Process S.

// Create a mutex object accessible by Process S.
HANDLE hObjInProcessS = CreateMutex(NULL, FALSE, NULL);

// Get a handle to Process T's kernel object.
HANDLE hProcessT = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
   dwProcessIdT);

HANDLE hObjInProcessT; // An uninitialized handle relative to Process T.

// Give Process T access to our mutex object.
DuplicateHandle(GetCurrentProcess(), hObjInProcessS, hProcessT,
   &hObjInProcessT, 0, FALSE, DUPLICATE_SAME_ACCESS);

// Use some IPC mechanism to get the handle value of hObjInProcessS into Process T.
...
// We no longer need to communicate with Process T.
CloseHandle(hProcessT);
...
// When Process S no longer needs to use the mutex, it should close it.
CloseHandle(hObjInProcessS);
```

调用 **GetCurrentProcess** 会返回一个伪句柄，该句柄始终标识主调进程——本例是 Process S。一旦 **DuplicateHandle** 函数返回，**hObjInProcessT** 就是一个相对于 Process T 的句柄，它标识的对象就由“Process S 中的代码引用的 **hObjInProcessS**”所标识的对象。在 Process S 中，永远不要执行以下代码：

```
// Process S should never attempt to close the duplicated handle.
CloseHandle(hObjInProcessT);
```

如果 Process S 执行了上述代码，CloseHandle 函数调用可能会、也可能不会失败。但 CloseHandle 成功与否并不重要。重要的是：假如 Process S 碰巧能访问句柄值和 hObjInProcessT 相同的一个内核对象，上述函数调用就会成功。换言之，这个调用可能错误

地关闭一个内核对象；下次 Process S 试图访问这个内核对象时，肯定会造成应用程序出现你不希望的行为（这还只是好听的一种说法）。

还可以通过另一种方式来使用 **DuplicateHandle**：假设一个进程拥有对一个文件映射对象的读写权限。在程序中的某个位置，我们调用了一个函数，并希望它对文件映射对象进行只读访问。为了使你的应用程序变得更健壮，可以使用 **DuplicateHandle** 为现有的对象创建一个新句柄，并确保这个新句柄只设置了只读权限。然后，把这个只读句柄传给函数。采取这种方式，函数中的代码绝对不会对文件映射对象执行意外的写入操作。以下代码对此进行了演示：

```
int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE,
   LPTSTR szCmdLine, int nCmdShow) {

   // Create a file-mapping object; the handle has read/write access.
   HANDLE hFileMapRW = CreateFileMapping(INVALID_HANDLE_VALUE,
      NULL, PAGE_READWRITE, 0, 10240, NULL);

   // Create another handle to the file-mapping object;
   // the handle has read-only access.
   HANDLE hFileMapRO;
   DuplicateHandle(GetCurrentProcess(), hFileMapRW, GetCurrentProcess(),
      &hFileMapRO, FILE_MAP_READ, FALSE, 0);

   // Call the function that should only read from the file mapping.
   ReadFromTheFileMapping(hFileMapRO);

   // Close the read-only file-mapping object.
   CloseHandle(hFileMapRO);

   // We can still read/write the file-mapping object using hFileMapRW.
   ...
   // When the main code doesn't access the file mapping anymore,
   // close it.
   CloseHandle(hFileMapRW);
```

第II部分　完成编程任务

本部分内容包括
- 第4章　进程
- 第5章　作业
- 第6章　线程基础
- 第7章　线程进度安排、优先级和线程亲和性
- 第8章　用户模式下的线程同步
- 第9章　线程与内核对象的同步
- 第10章　同步和异步设备I/O
- 第11章　Windows线程池
- 第12章　纤程

第 4 章　进程

本章内容包括：
- 编写第一个Windows应用程序
- CreateProcess函数
- 终止进程
- 子进程
- 当管理员以标准用户的身份运行时

本章将讨论系统如何管理正在运行的所有应用程序。首先要解释什么是进程，以及系统如何创建一个进程内核对象来管理每个进程。然后，我们要解释如何利用与一个进程关联的内核对象来操纵该进程。接下来，我们要讨论进程的各种不同的属性（或特性），以及用于查询和更改这些属性的几个函数。另外，还要讨论如何利用一些函数在系统中创建或生成额外的进程。当然，最后还要讨论如何终止线程，这是讨论进程时必不可少的一个主题。

一般将进程定义成一个正在运行的程序的一个实例，它由以下两个组件构成：
- 一个内核对象，操作系统用它来管理进程。内核对象也是系统保存进程统计信息的地方。
- 一个地址空间，其中包含所有执行体（executable）或DLL模块的代码和数据。此外，它还包含动态内存分配，比如线程堆栈和堆的分配。

进程是有“惰性”的。进程要做任何事情，都必须让一个线程在它的上下文中运行。该线程要执行进程地址空间包含的代码。事实上，一个进程可以有多个线程，所有线程都在进程的地址空间中“同时”执行代码。为此，每个线程都有它自己的一组CPU寄存器和它自己的堆栈。每个进程至少要有一个线程执行进程地址空间包含的代码。一个进程创建的时候，系统会自动

创建它的第一个线程，这称为主线程（primary thread）。然后，这个线程再创建更多的线程，后者再创建更多的线程……。如果没有线程要执行进程地址空间包含的代码，进程就失去了继续存在的理由。所以，系统会自动销毁进程及其地址空间。

对于所有要运行的线程，操作系统会轮流为每个线程调度一些CPU时间。它会采取round-robin（轮询或轮流）方式，为每个线程都分配时间片（称为“量”或者“量程”，即quantum），从而营造出所有线程都在“并发”运行的假象。图4-1展示了一台单CPU的机器的工作方式。

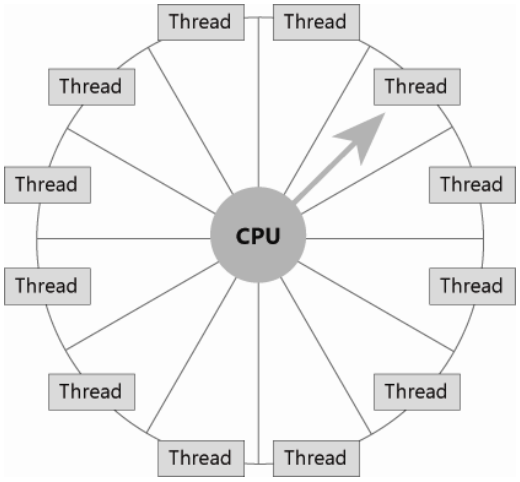


图4-1　在单CPU的计算机上，操作系统以round-robin方式**为每个单独的线程分配时间量**

如果计算机配备了多个CPU，操作系统会采用更复杂的算法为线程分配CPU时间。Microsoft Windows可以同时让不同的CPU执行不同的线程，使多个线程能真正并发运行。在这种类型的计算机系统中，Windows内核将负责线程的所有管理和调度任务。你不必在自己的代码中做任何特别的事情，即可享受到多处理器系统带来的好处。不过，为了更好地利用这些CPU，你需要在应用程序的算法中多做一些文章。

4.1　编写第一个 Windows 应用程序

Windows支持两种类型的应用程序：GUI程序和CUI程序。前者是“图形用户界面”（Graphical user interface）的简称，后者是“控制台用户界面”（Console user interface）的简称。GUI程序一个图形化的前端。它可以创建窗口，可以拥有菜单，能通过对话框与用户交互，还能使用所有标准的“视窗化”的东西。Windows的几乎所有附件应用程序（比如记事本、计算器和写字板等等）都是GUI程序。控制台程序则是基于文本的。它们一般不会创建窗口或进程消息，而且不需要GUI。虽然CUI程序是在屏幕上的一个窗口中运行的，但这个窗口中只有文本。“命令提示符”（CMD.EXE）是CUI程序的一个典型的例子。

其实，这两种应用程序的界线是非常模糊的。你完全可以创建出能显示对话框的CUI应用程序。例如，在执行CMD.EXE并打开“命令提示符”后，你可以执行一个特殊的命令来显示一个图形化对话框。并在其中选择想要执行的命令，而不必强行记忆各种控制台命令。另外，还可以创建一个要向控制台窗口输出文本字符串的GUI应用程序。例如，我自己写的GUI程序经常都要创建一个控制台窗口，便于我查看应用程序执行期间的调试信息。不过，我当然要鼓励你尽可能在程序中使用一个GUI，而不要使用老式的字符界面，后者对用户来说不太友好！

用Microsoft Visual Studio来创建一个应用程序项目时，集成开发环境会设置各种链接器开关，

使链接器将子系统的正确类型嵌入最终生成的**执行体**（executable，或者称为“可执行文件”）中。对于CUI程序，这个链接器开关是**/SUBSYSTEM:CONSOLE**，对于GUI程序，则是**/SUBSYSTEM:WINDOWS**。用户运行应用程序时，操作系统的加载器（loader）会检查执行体映像的header，并获取这个子系统值。如果此值表明是一个CUI程序，加载程序会自动确定有一个可用的文本控制台窗口（比如从命令提示符启动这个程序的时候）。另外，如有必要，会创建一个新窗口（比如从Windows资源管理器启动这个CUI程序的时候）。如果此值表明是一个GUI程序，加载器就不会创建控制台窗口；相反，它只是加载这个程序。一旦应用程序开始运行，操作系统就不再关心应用程序的界面是什么类型的。

Windows应用程序必须有一个入口函数，应用程序开始运行时，这个函数会被调用。C/C++ 开发人员可以使用以下两种入口函数：

```
int WINAPI _tWinMain(
   HINSTANCE hInstanceExe,
   HINSTANCE,
   PTSTR pszCmdLine,
   int nCmdShow);

int _tmain(
   int argc,
   TCHAR *argv[],
   TCHAR *envp[]);
```

注意，具体的符号要取决于你是否要使用Unicode字符串。操作系统实际并不调用你所写的入口函数。相反，它会调用由C/C++运行库实现并在链接时使用-entry:命令行选项来设置的一个C/C++运行时启动函数。该函数将初始化C/C++运行库，使你能调用**malloc**和**free**之类的函数。它还确保了在你的代码开始执行之前，你声明的任何全局和静态C++对象都被正确地构造。表4-1总结了你的源代码要实现什么入口函数，以及每个入口函数应该在什么时候使用。

表4-1　应用程序类型和相应的入口函数

应用程序类型	入口函数（入口点）	嵌入执行体的启动函数
处理ANSI字符和字符串的GUI应用程序	_tWinMain (WinMain)	WinMainCRTStartup
处理Unicode字符和字符串的GUI应用程序	_tWinMain (wWinMain)	wWinMainCRTStartup
处理ANSI字符和字符串的CUI应用程序	_tmain (Main)	mainCRTStartup
处理Unicode字符和字符串的CUI应用程序	_tmain (Wmain)	wmainCRTStartup

链接你的执行体时，链接器将选择正确的C/C++运行时启动函数。如果指定了**/SUBSYSTEM:WINDOWS**链接器开关，链接器就会寻找**WinMain**或**wWinMain**函数。如果没有找到这两个函数，链接器将返回一个“unresolved external symbol（未解析的外部符号）”错误；否则，它将根据具体情况分别选择**WinMainCRTStartup**或**wWinMainCRTStartup**函数。

类似地，如果指定了**/SUBSYSTEM:CONSOLE**链接器开关，链接器就会寻找**main**或**wmain**函数，并根据情况分别选择**mainCRTStartup**或**wmainCRTStartup**函数。同样地，如果**main**和**wmain**函数都没有找到，链接器会返回一个“unresolved external symbol（未解析的外部符号）”错误。

不过，一个鲜为人知的事实是，完全可以从自己的项目中移除**/SUBSYSTEM**链接器开关。一旦这样做，链接器就会自动判断应该将应用程序设为哪一个子系统。链接时，链接器会检查代码中包括4个函数中的哪一个（**WinMain**，**wWinMain**，**main**或**wmain**），并据此推算你的执行体应该是哪个子系统，以及应该在执行体中嵌入哪个C/C++启动函数。

Windows/Visual C++新手开发人员常犯的一个错误是在创建一个新项目时错误选择了项目类型。例如，开发人员可能选择创建一个新的Win32应用程序项目，但创建的入口函数是**main**。生成应用程序时，会报告一个链接器错误，因为Win32应用程序项目会设置**/SUBSYSTEM:WINDOWS**链接器开关，但**WinMain**或**wWinMain**函数并不存在。此时，开发人员有以下4个选择。

- 把**main**函数改为**WinMain**。这通常不是最佳方案，因为开发人员真正希望的可能是创建一个控制台应用程序。
- 在Visual C++中创建一个新的Win32控制台应用程序项目，然后在新项目中添加现有的源代码模块。这个办法过于繁琐。它相当于一切都从头开始，而且必须删除原来的项目文件。
- 在项目属性对话框中，定位到Configuration Properties（配置属性）/Linker（链接器）/System（系统）/SubSystem（子系统）选项，把**/SUBSYSTEM:WINDOWS**开关改为**/SUBSYSTEM:CONSOLE**，如图4-2所示。这是最简单的解决方案，很少有人知道这个窍门。
- 在项目属性对话框中，删除**/SUBSYSTEM:WINDOWS**开关。这是我个人最偏爱的选项，因为它能提供最大的灵活性。现在，链接器将根据源代码中实现的函数来执行正确的操作。用Visual Studio创建一个新的Win32应用程序或Win32控制台应用程序项目时，这才应该是真正的默认设定啊！

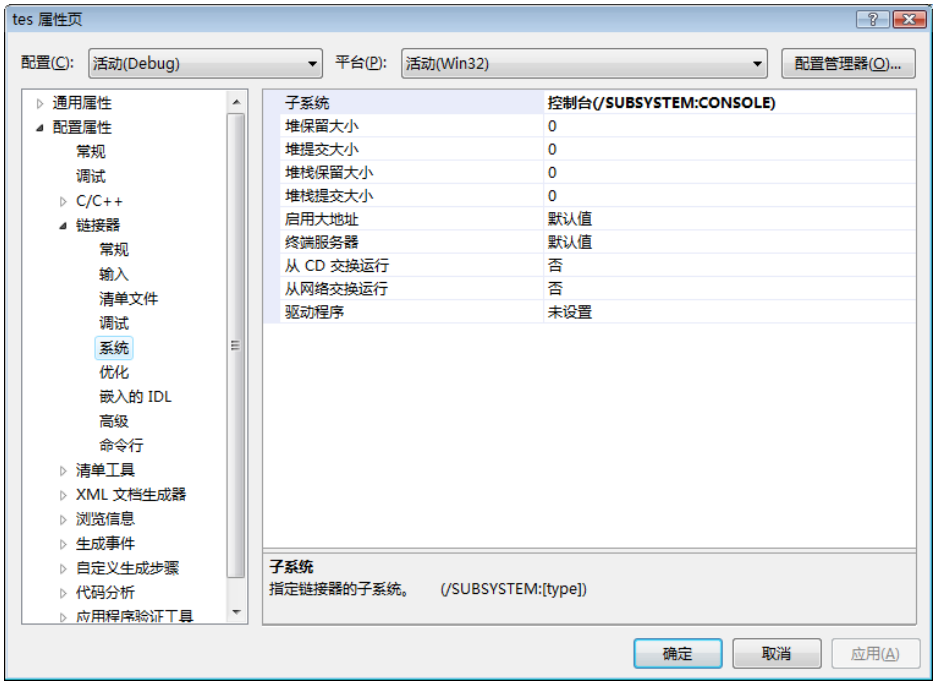


图4-2　在项目的属性对话框中，为一个项目选择一个CUI子系统

所有C/C++运行时启动函数所做的事情基本都是一样的，区别在于它们要处理的是ANSI字符串，还是Unicode字符串；以及在初始化C运行库之后，它们调用的是哪一个入口函数。Visual C++自带C运行库的源代码。可以在crtexe.c文件中找到4个启动函数的源代码。这些启动函数的用途简单总结如下。

- 获取指向新进程的完整命令行的一个指针。
- 获取指向新进程的环境变量的一个指针。
- 初始化C/C++运行库的全局变量。如果include了StdLib.h，你的代码就可以访问这些变

量。表4-2总结了这些变量。

- 初始化C运行库内存分配函数（**malloc**和**calloc**）和其他低级I/O例程使用的堆（heap）。
- 调用所有全局和静态C++类对象的构造函数。

表4-2　程序可以访问的C/C++运行时全局变量

变量名称	类型	描述和推荐使用的Windows函数
_osver	unsigned int	操作系统的build版本号。例如，Windows Vista RTM为build 6000。所以，_osver的值就是6000。请换用GetVersionEx。
_winmajor	unsigned int	以十六进制表示的Windows系统的major版本号。对于Windows Vista，该值为6。请换用GetVersionEx。
_winminor	unsigned int	以十六进制表示的Windows系统的minor版本号。对于Windows Vista，该值为0。请换用GetVersionEx。
_winver	unsigned int	(_winmajor << 8) + _winminor。请换用GetVersionEx。
__argc	unsigned int	命令行上传递的参数的个数。请换用GetCommandLine。
__argv __wargv	char wchar_t	长度为__argc的一个数组，其中含有指向ANSI/Unicode字符串的指针。数组中的每一项都指向一个命令行参数。注意，如果定义了_UNICODE，__argv就为NULL；如果没有定义，则__wargv为NULL。请换用GetCommandLine。
_environ _wenviron	char wchar_t	一个指针数组，这些指针指向ANSI/Unicode字符串。数组中的每一项都指向一个环境字符串。注意，如果没有定义_UNICODE，_wenviron就为 NULL；如果已经定义了_UNICODE ， _environ 就 为 NULL 。 请 换 用 GetEnvironmentStrings 或 GetEnvironmentVariable。
_pgmptr _wpgmptr	char wchar_t	正在运行的程序的名称及其ANSI/Unicode完整路径。注意，如果已经定义了_UNICODE，_pgmptr就为NULL。如果没有定义_UNICODE，_wpgmptr就为 NULL。请换用GetModuleFileName，将NULL作为第一个参数传给该函数

完成所有这些初始化工作之后，C/C++启动函数就会调用应用程序的入口函数。如果你写了一个**_tWinMain**函数，而且定义了**_UNICODE**，其调用过程将如下所示：

```
GetStartupInfo(&StartupInfo);
int nMainRetVal = wWinMain((HINSTANCE)&__ImageBase, NULL, pszCommandLineUnicode,
  (StartupInfo.dwFlags & STARTF_USESHOWWINDOW)
     ? StartupInfo.wShowWindow : SW_SHOWDEFAULT);
```

如果没有定义**_UNICODE**，其调用过程将如下所示：

```
GetStartupInfo(&StartupInfo);
int nMainRetVal = WinMain((HINSTANCE)&__ImageBase, NULL, pszCommandLineAnsi,
  (StartupInfo.dwFlags & STARTF_USESHOWWINDOW)
     ? StartupInfo.wShowWindow : SW_SHOWDEFAULT);
```

注意，**_ImageBase**是一个链接器定义的伪变量，表明执行体文件映射到应用程序内存中的什么位置。后面的“进程实例句柄”一节将进一步讨论这个问题。

如果你写了一个**_tmain**函数，而且定义了**_UNICODE**，那么其调用过程如下：

```
int nMainRetVal = wmain(argc, argv, envp);
```

如果没有定义**_UNICODE**，调用过程如下：

```
int nMainRetVal = main(argc, argv, envp);
```

注意，用Visual Studio向导生成应用程序时，CUI应用程序的入口中没有定义第三个参数（环境变量块），如下所示：

```
int _tmain(int argc, TCHAR* argv[]);
```

如果需要访问进程的环境变量，只需将上述调用替换成下面这一行：

```
int _tmain(int argc, TCHAR* argv[], TCHAR* env[])
```

这个**env**参数指向一个数组，数组中包含所有环境变量及其值，两者用等号（=）分隔。对环境变量的详细讨论将在后面的“进程的环境变量”小节进行。
入口函数返回后，启动函数将调用C运行库函数**exit**，向其传递你的返回值（**nMainRetVal**）。**exit**函数执行以下任务。

- 调用**_onexit**函数调用所注册的任何一个函数。
- 调用所有全局和静态C++类对象的析构函数。
- 在**DEBUG**生成中，如果设置了_CRTDBG_LEAK_CHECK_DF标志，就通过调用_CrtDumpMemoryLeaks函数来生成内存泄漏报告。
- 调用操作系统的**ExitProcess**函数，向其传入**nMainRetVal**。这会导致操作系统杀死你的进程，并设置它的退出代码。

注意，为安全起见，所有这些变量都是不赞成使用的，因为使用了这些变量的代码可能会在C运行库初始化这些变量之前开始执行。有鉴于此，你应该直接调用对应的Windows API函数。

4.1.1 进程实例句柄

加载到进程地址空间的每一个执行体或者DLL文件都被赋予了一个独一无二的实例句柄。执行体文件的实例被当作**(w)WinMain**函数的第一个参数**hInstanceExe**传入。在需要加载资源的函数调用中，一般都要提供此句柄的值。例如，为了从执行体文件的映像中加载一个图标资源，就需要调用下面这个函数：

```
HICON LoadIcon(
  HINSTANCE hInstance,
  PCTSTR pszIcon);
```

LoadIcon函数的第一个参数指出哪个文件（执行体或DLL文件）包含了想要加载的资源。许多应用程序都会将**(w)WinMain**的**hInstanceExe**参数保存在一个全局变量中，使其很容易由执行体文件的所有代码访问。

Platform SDK文档指出，有的函数需要一个**HMODULE**类型的参数。下面的**GetModuleFileName**函数便是一个例子：

```
DWORD GetModuleFileName(
  HMODULE hInstModule,
  PTSTR pszPath,
  DWORD cchPath);
```

> 注意
> 可以看出，HMODULE和HINSTANCE完全是一回事。如果某个函数的文档指出需要一个HMODULE参数，你可以传入一个HINSTANCE，反之亦然。之所以有两种数据类型，是由于在16位Windows中，HMODULE和HINSTANCE是两回事。

(w)WinMain的**hInstanceExe**参数的实际值是一个基内存地址；在这个位置，系统将执行体文件的映像加载到进程的地址空间中。例如，假如系统打开执行体文件，并将它的内容加载到地址0x00400000，则**(w)WinMain**的**hInstanceExe**参数值为0x00400000。

执行体文件的映像具体加载到哪一个基地址，这是由链接器决定的。不同的链接器使用不同的默认基地址。由于历史原因，Visual Studio链接器使用的默认基地址是0x00400000，这是在运行Windows 98时，执行体文件的映像能加载到的最低的一个地址。使用Microsoft链接器的**/BASE:*address***链接器开关，可以更改你的应用程序加载到的基地址。

为了知道一个执行体或DLL文件被加载到进程地址空间的什么位置，可以使用如下所示的GetModuleHandle函数来返回一个句柄/基地址：

```
HMODULE GetModuleHandle(PCTSTR pszModule);
```

调用这个函数时，要传递一个以0终止的字符串，它指定了已在主调进程的地址空间中加载的一个执行体或DLL文件的名称。如果系统找到了指定的执行体或DLL文件名称，**GetModuleHandle**就会返回执行体/DLL文件映像加载到的基地址。如果没有找到文件，系统将返回**NULL**。**GetModuleHandle**的另一个用法是为**pszModule**参数传入NULL，这样可以返回主调进程的执行体文件的基地址。如果你的代码在一个DLL中，那么可利用两个办法来了解代码正在什么模块中运行。第一个办法是利用链接器提供的伪变量**__ImageBase**，它指向当前正在运行的模块的基地址。如前所述，这是C运行时启动代码在调用你的**(w)WinMain**函数时所做的事情。

第二个办法是调用**GetModuleHandleEx**，将**GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS**作为它的第一个参数，将当前方法的地址作为第二个参数。最后一个参数是指向一个**HMODULE**的指针，该指针将由**GetModuleHandleEx**来填充，它就是包含了传入函数的那个DLL的基地址。以下代码对这两个办法都进行了演示：

```
extern "C" const IMAGE_DOS_HEADER __ImageBase;

void DumpModule() {
   // Get the base address of the running application.
   // Can be different from the running module if this code is in a DLL.
   HMODULE hModule = GetModuleHandle(NULL);
   _tprintf(TEXT("with GetModuleHandle(NULL) = 0x%x\r\n"), hModule);

   // Use the pseudo-variable __ImageBase to get
   // the address of the current module hModule/hInstance.
   _tprintf(TEXT("with __ImageBase = 0x%x\r\n"), (HINSTANCE)&__ImageBase);

   // Pass the address of the current method DumpModule
   // as parameter to GetModuleHandleEx to get the address
   // of the current module hModule/hInstance.
   hModule = NULL;
   GetModuleHandleEx(
   GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS,
   (PCTSTR)DumpModule,
   &hModule);
   _tprintf(TEXT("with GetModuleHandleEx = 0x%x\r\n"), hModule);
}

int _tmain(int argc, TCHAR* argv[]) {
   DumpModule();
   return(0);
}
```

记住**GetModuleHandle**函数的两大重要特征。首先，它只检查主调进程的地址空间。如果主调进程没有使用任何通用对话框函数，那么一旦调用**GetModuleHandle**，并向其传递ComDlg32，就会导致返回**NULL**——即使ComDlg32.dll也许已经加载到其他进程的地址空间。其次，调用**GetModuleHandle**并向其传递NULL值，会返回进程的地址空间中的执行体文件的基地址。所以，即使从包含在DLL中的代码调用**GetModuleHandle(NULL)**，返回值仍是执行体文件的基地址，而非DLL文件的基地址。

4.1.2 进程的前一个实例句柄

如前所述，C/C++运行时启动代码总是向**(w)WinMain**的**hPrevInstance**参数传递**NULL**。该参数用于16位Windows系统，并依然是 **(w)WinMain**的一个参数，目的只是方便你移植16位Windows应用程序。绝对不要在自己的代码中引用这个参数。因此，我始终会像下面这样写自己的**(w)WinMain**函数：

```
int WINAPI _tWinMain(
  HINSTANCE hInstanceExe,
  HINSTANCE,
  PSTR pszCmdLine,
  int nCmdShow);
```

由于没有为第二个参数指定参数名，所以编译器不会报告一个“parameter not referenced（参数未引用）”警告。Visual Studio 选择了一个不同的解决方案：在向导生成的C++ GNU项目中，利用了**UNREFERENCED_PARAMETER**宏来移除这种警告。下面这个代码段对此进行了演示：

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR lpCmdLine,
                       int nCmdShow) {
  UNREFERENCED_PARAMETER(hPrevInstance);
  UNREFERENCED_PARAMETER(lpCmdLine);
  ...
}
```

4.1.4 进程的命令行

一个新进程在创建时，一个命令行会传给它。这个命令行几乎总是非空的；至少，用于创建新进程的执行体文件的名称是命令行上的第一个标记（token）。不过，在后面讨论CreateProcess函数的时候，你会知道进程能接收只由一个字符构成的命令行，这个字符就是用于终止字符串的0。C运行库的启动代码开始执行一个GUI应用程序时，会调用Windows函数**GetCommandLine**来获取进程的完整命令行，忽略执行体文件的名称，然后将指向命令行剩余部分的一个指针传给**WinMain**的**pszCmdLine**参数。

应用程序可以通过自己选择的任何一种方式来分析和解释命令行字符串。可以实际地写数据到**pszCmdLine**参数所指向的内存缓冲区，但在任何情况下，都不要超过缓冲区的末尾。就我个人而言，我始终把它当作一个只读的缓冲区来对待。如果想对命令行进行改动，我首先会将命令行缓冲区复制到我的应用程序的一个本地缓冲区，再对自己的本地缓冲区进行修改。

在C运行库的例子之后，你还可以通过调用**GetCommandLine**函数来获得一个指向进程完整命令行的指针，如下所示：

PTSTR GetCommandLine();

该函数返回一个缓冲区指针，缓冲区中包含了完整的命令行（包括已执行的文件的完整路径名）。注意，**GetCommandLine**返回的总是同一个缓冲区的地址。这是不应该向**pszCmdLine**写入数据的另一个理由：它指向同一个缓冲区，修改它之后，就没办法知道原来的命令行是什么。

许多应用程序都需要将命令行解析成一组单独的标记（tokens）。应用程序可以使用全局变量**__argc**和**__argv**（或 **__wargv**）来访问对命令行的各个单独的组件，即使这些变量现在已经不提倡使用了。利用在ShellAPI.h文件中声明并由Shell32.dll导出的函数**CommandLineToArgvW**，

可以将任何Unicode字符串分解成单独的标记：

```
PWSTR* CommandLineToArgvW(
   PWSTR pszCmdLine,
   int* pNumArgs);
```

正如函数名最后的W所暗示的一样，这个函数只有Unicode版本（W代表wide）。 第一个参数**pszCmdLine**指向一个命令行字符串。这通常是前面的**GetCommandLineW**函数调用的返回值。**pNumArgs**参数是一个整数的地址，该整数被设为命令行中的实参的数目。**CommandLineToArgvW**返回的是一个Unicode字符串指针数组的地址。

CommandLineToArgvW在内部分配内存。许多应用程序不会释放这个内存——它们指望操作系统在进程终止时释放内存。这是完全可以接受的。不过，如果你想自己释放内存，正确的做法就是像下面这样调用**HeapFree**：

```
int nNumArgs;
PWSTR *ppArgv = CommandLineToArgvW(GetCommandLineW(), &nNumArgs);

// Use the arguments…
if (*ppArgv[1] == L'x') {
   ...
}
// Free the memory block
HeapFree(GetProcessHeap(), 0, ppArgv);
```

4.1.5 进程的环境变量

每个进程都有一个与它关联的环境块（environment block），这是在进程地址空间内分配的一个内存块，其中包含和下面相似的一组字符串：

```
=::=::\ ...
VarName1=VarValue1\0
VarName2=VarValue2\0
VarName3=VarValue3\0 ...
VarNameX=VarValueX\0
\0
```

每个字符串的第一部分是一个环境变量的名称，后跟一个等号，等号之后是希望赋给此变量的值。注意，除了第一个**=::=::**字符串，块中可能还有其他字符串是以等号（=）开头的。这种字符串不作为环境变量使用，详情参见后面的“进程的当前目录”小节。

前面已介绍了访问环境块的两种方式，它们各自使用了一种不同的解析方式，会提供一个不同的输出。第一种方式是调用**GetEnvironmentStrings**函数来获取完整的环境块。格式与前一段描述的完全一致。下面的代码展示了如何在这种情况下提取环境变量及其内容：

```
void DumpEnvStrings() {
   PTSTR pEnvBlock = GetEnvironmentStrings();
   // Parse the block with the following format:
   // =::=::\
   // =...
   // var=value\0
   // ...
   // var=value\0\0
   // Note that some other strings might begin with '='.
   // Here is an example when the application is started from a network share.
   // [0] =::=::\
   // [1] =C:=C:\Windows\System32
   // [2] =ExitCode=00000000
   //
   TCHAR szName[MAX_PATH];
```

```
   TCHAR szValue[MAX_PATH];
   PTSTR pszCurrent = pEnvBlock;
   HRESULT hr = S_OK;
   PCTSTR pszPos = NULL;
   int current = 0;

   while (pszCurrent != NULL) {
      // Skip the meaningless strings like:
      // "=::=::\"
      if (*pszCurrent != TEXT('=')) {
         // Look for '=' separator.
         pszPos = _tcschr(pszCurrent, TEXT('='));

         // Point now to the first character of the value.
         pszPos++;

         // Copy the variable name.
         size_t cbNameLength = // Without the' ='
            (size_t)pszPos - (size_t)pszCurrent - sizeof(TCHAR);
         hr = StringCbCopyN(szName, MAX_PATH, pszCurrent, cbNameLength);
         if (FAILED(hr)) {
            break;
         }

         // Copy the variable value with the last NULL character
         // and allow truncation because this is for UI only.
         hr = StringCchCopyN(szValue, MAX_PATH, pszPos, _tcslen(pszPos)+1);
         if (SUCCEEDED(hr)) {
            _tprintf(TEXT("[%u] %s=%s\r\n"), current, szName, szValue);
         } else // something wrong happened; check for truncation.
         if (hr == STRSAFE_E_INSUFFICIENT_BUFFER) {
            _tprintf(TEXT("[%u] %s=%s...\r\n"), current, szName, szValue);
         } else { // This should never occur.
            _tprintf(
               TEXT("[%u] %s=???\r\n"), current, szName
               );
            break;
         }
      } else {
         _tprintf(TEXT("[%u] %s\r\n"), current, pszCurrent);
   }

   // Next variable please.
   current++;

   // Move to the end of the string.
   while (*pszCurrent != TEXT('\0'))
      pszCurrent++;
   pszCurrent++;

   // Check if it was not the last string.
   if (*pszCurrent == TEXT('\0'))
      break;

   };

   // Don't forget to free the memory.
   FreeEnvironmentStrings(pEnvBlock);
}
```

以=开头的无效字符串会被跳过。其他每个有效的字符串会被逐一解析。=字符被用作名称与值之间的分隔符。如果不再需要**GetEnvironmentStrings**函数返回的内存块，应调用**FreeEnvironmentStrings**函数来释放它：

```
BOOL FreeEnvironmentStrings(PTSTR pszEnvironmentBlock);
```

注意，在上述代码段中，使用了C运行库的安全字符串函数。目的是利用StringCbCopyN算出的大小（以字节为单位），如果值太长以至于拷贝缓冲区装不下，就用**StringCchCopyN**函数来截断。

访问环境变量的第二种方式是CUI程序专用的，它通过应用程序main入口函数所接收的**TCHAR* env[]**参数来实现。不同于**GetEnvironmentStrings**返回的值，**env**是一个字符串指针数组，每个指针都指向一个不同的环境变量（定义采用常规的“名称=值”的格式）。**NULL**指针出现在指向最后一个变量字符串的指针之后，如下所示：

```
void DumpEnvVariables(PTSTR pEnvBlock[]) {
   int current = 0;
   PTSTR* pElement = (PTSTR*)pEnvBlock;
   PTSTR pCurrent = NULL;
   while (pElement != NULL) {
      pCurrent = (PTSTR)(*pElement);
      if (pCurrent == NULL) {
         // 没有更多的环境变量了
         pElement = NULL;
      } else {
         _tprintf(TEXT("[%u] %s\r\n"), current, pCurrent);
         current++;
         pElement++;
      }
   }
}
```

注意，以等号开头的那些奇怪的字符串在你接收到env之前就已经被移除了，所以不必专门处理它们。

由于等号用于分隔名称和值，所以它并不是名称的一部分。另外，空格是有意义的。例如，如果声明了以下两个变量，然后再比较XYZ和ABC的值，系统就会报告两个变量不相同，因为等号之前和之后的任何空格都会被考虑在内：

```
XYZ= Windows （注意等号后的空格。）
ABC=Windows
```

例如，如果你想添加以下两个字符串到环境块，那么其后带有空格的环境变量*XYZ*就会包含*Home*，而不带空格的环境变量*XYZ*就会包含*Work*，如下所示：

```
XYZ =Home （注意等号前的空格。）
XYZ=Work
```

用户登录Windows时，系统会创建shell进程，并将一组环境字符串与其关联。系统通过检查注册表中的两个项来获得初始的环境字符串。

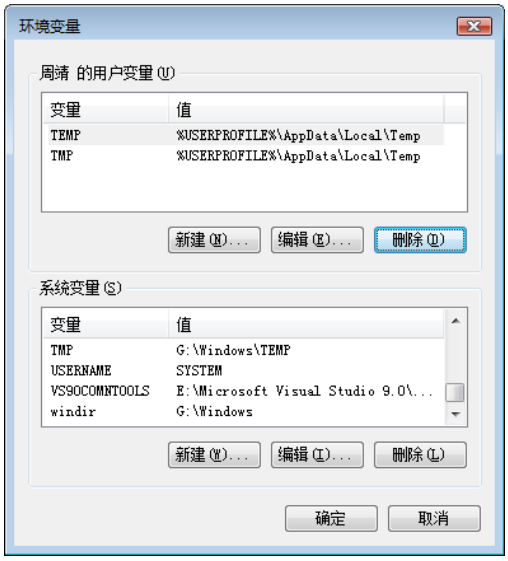
第一个项包含应用于系统的所有环境变量的列表：

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Environment
```

第二个项包含应用于当前登录用户的所有环境变量的列表：

```
HKEY_CURRENT_USER\Environment
```

用户可以添加、删除或更改所有这些变量，具体做法是打开“控制面板”，选择“系统”，然后单击“高级”按钮，在“高级”选项卡中单击“环境变量”按钮，随后将弹出以下对话框。



要有管理员权限才能更改“系统变量”列表中包含的变量。

应用程序还可以使用各种注册表函数来修改这些注册表项。不过，为了使你的改动对所有应用程序生效，用户必须注销并重新登录。有的应用程序（比如资源管理器、任务管理器和控制面板）可以在其主窗口接收到**WM_SETTINGCHANGE**消息时，用新的注册表项来更新它们的环境块。例如，假如更新了注册表项，并希望应用程序立即更新它们的环境块，可以进行如下调用：

```
SendMessage(HWND_BROADCAST, WM_SETTINGCHANGE, 0, (LPARAM) TEXT("Environment"));
```

通常，子进程会继承一组环境变量，这些环境变量和父进程的环境变量相同。不过，父进程可以控制哪些环境变量允许子进程继承，详情参见后文对**CreateProcess**函数的讨论。这里所说的“继承”是指子进程获得父进程的环境块的一个拷贝，这个拷贝是子进程专用的。换言之，子进程和父进程并不共享同一个环境块。这意味着子进程可以在自己的环境块中添加、删除或修改变量，但这些改动不会影响到父进程的环境块。

应用程序经常利用环境变量让用户精细地调整其行为。用户创建一个环境变量并进行初始化。然后，当用户调用应用程序时，应用程序在环境块中查找变量。如果找到了变量，就会解析变量的值，并调整其自己的行为。

环境块的问题是，用户不容易设置或理解它们。用户需要正确拼写变量名称，而且还必须知道变量值的确切格式。另一方面，绝大多数图形应用程序都允许用户使用对话框来调整应用程序的行为。这个办法对用户来说要友好得多。

如果仍然想使用环境变量，有几个函数可供你的应用程序调用。可以使用**GetEnvironmentVariable**函数来判断一个环境变量是否存在；如果存在，它的值是什么。如下所示：

```
DWORD GetEnvironmentVariable(
    PCTSTR pszName,
    PTSTR pszValue,
    DWORD cchValue);
```

调用**GetEnvironmentVariable**时，**pszName**指向预期的变量名称，**pszValue**指向保存变量值的

缓冲区，而 **cchValue**指出缓冲区大小（用字符数来表示）。如果在环境中找到变量名，**GetEnvironmentVariable**函数将返回复制到缓冲区的字符数；如果在环境中没有找到变量名，就返回0。然而，由于我们不知道为了保存一个环境变量的值需要多少个字符，所以当0值被当作**cchValue**参数的值传入时，**GetEnvironmentVariable**会返回包括末尾的NULL字符在内的字符的数量。以下代码演示了如何安全地使用这个函数：

```
void PrintEnvironmentVariable(PCTSTR pszVariableName) {
   PTSTR pszValue = NULL;
   // Get the size of the buffer that is required to store the value
   DWORD dwResult = GetEnvironmentVariable(pszVariableName, pszValue, 0);
   if (dwResult != 0) {
      // Allocate the buffer to store the environment variable value
      DWORD size = dwResult * sizeof(TCHAR);
      pszValue = (PTSTR)malloc(size);
      GetEnvironmentVariable(pszVariableName, pszValue, size);
      _tprintf(TEXT("%s=%s\n"), pszVariableName, pszValue);
      free(pszValue);
   } else {
      _tprintf(TEXT("'%s'=<unknown value>\n"), pszVariableName);
   }
}
```

在许多字符串的内部，都包含了“可替换字符串”。例如，我在注册表的某个地方发现了下面这个字符串：

```
%USERPROFILE%\Documents
```

两个百分号(%)之间的这部分内容就是一个“可替换字符串”。在这种情况下，环境变量**USERPROFILE**的值应该放在这里。在我的机器上，**USERPROFILE**环境变量的值是：

```
C:\Users\jrichter
```

所以，执行字符串替换之后，生成的扩展字符串是：

```
C:\Users\jrichter\Documents
```

由于这种形式的字符串替换非常常见，所以Windows专门提供了**ExpandEnvironmentStrings**函数，如下所示：

```
DWORD ExpandEnvironmentStrings(
   PTCSTR pszSrc,
   PTSTR pszDst,
   DWORD chSize);
```

调用这个函数时，**pszSrc**参数是包含“可替换环境变量字符串”的一个字符串的地址。**pszDst**参数是用于接收扩展字符串的一个缓冲区的地址，而**chSize**参数是这个缓冲区的最大大小（用字符数来表示）。返回值是保存扩展字符串所需的缓冲区的大小（用字符数来表示）。如果**chSize**参数小于此值，**%%**变量就不会扩展，而是被空字符串替换。所以，通常要调用两次**ExpandEnvironmentStrings**函数，如下所示：

```
DWORD chValue =
   ExpandEnvironmentStrings(TEXT("PATH='%PATH%'"), NULL, 0);
PTSTR pszBuffer = new TCHAR[chValue];
chValue = ExpandEnvironmentStrings(TEXT("PATH='%PATH%'"), pszBuffer, chValue);
_tprintf(TEXT("%s\r\n"), pszBuffer);
delete[] pszBuffer;
```

最后，可以使用**SetEnvironmentVariable**函数添加一个变量，删除一个变量，或者修改一个变量的值：

```
BOOL SetEnvironmentVariable(
    PCTSTR pszName,
    PCTSTR pszValue);
```

此函数将**pszName**所标识的一个变量设为**pszValue**参数所标识的值。如果已经存在具有指定名称的一个变量，**SetEnvironmentVariable**函数会修改它的值。如果指定的变量不存在，就添加这个变量。如果**pszValue**为**NULL**，变量会从环境块中删除。

应该始终使用这些函数来操纵进程的环境块。

4.1.6 进程的亲和性

通常，进程中的线程可以在主机的任何CPU上执行。然而，也可以强迫线程在可用CPU的一个子集上运行，这称为“处理器亲和性”（processor affinity），详情将在第7章讨论。子进程继承了其父进程的亲和性。

4.1.7 进程的错误模式

与每个进程都关联了一组标志，这些标志的作用是让系统知道进程如何响应严重错误，包括磁盘介质错误、未处理的异常、文件查找错误以及数据对齐错误等。进程可以调用**SetErrorMode**函数来告诉系统如何处理这些错误：

```
UINT SetErrorMode(UINT fuErrorMode);
```

fuErrorMode参数是表4-3列出的标志按位OR组合结果。

表4-3 SetErrorMode的标志

标志	描述
SEM_FAILCRITICALERRORS	系统不显示严重错误处理程序（critical-error-handler）消息框，并将错误返回主调进程。
SEM_NOGPFAULTERRORBOX	系统不显示常规保护错误（general-protection-fault）消息框。此标志只应该由调试程序设置；该调试程序用一个异常处理程序来自行处理常规保护（general protection，GP）错误。
SEM_NOOPENFILEERRORBOX	系统查找文件失败时，不显示消息框。
SEM_NOALIGNMENTFAULTEXCEPT	系统自动修复内存对齐错误，并使应用程序看不到这些错误。此标志对x86/x64处理器无效

默认情况下，子进程会继承父进程的错误模式标志。换言之，如果一个进程已经打开了**SEM_NOGPFAULTERRORBOX**标志，并生成了一个子进程，则子进程也会打开这个标志。不过，子进程自己并不知道这一点，而且在编写它时，或许根本没有考虑到要处理GP错误。如果一个GP错误发生在子进程的一个线程上，则子进程可能在不通知用户的情况下终止。父进程可以阻止子进程继承其错误模式，方法是在调用**CreateProcess**时指定**CREATE_DEFAULT_ERROR_MODE**标志（我们将在本章稍后讨论**CreateProcess**）。

4.1.8 进程当前所在的驱动器和目录

如果不提供完整的路径名，各种Windows函数会在当前驱动器的当前目录查找文件和目录。例如，如果进程中的一个线程调用**CreateFile**来打开一个文件(未指定完整路径名)，系统将在当前驱动器和目录查找该文件。

系统在内部跟踪记录着一个进程的当前驱动器和目录。由于这种信息是以进程为单位来维护的，所以假如进程中的一个线程更改了当前驱动器或目录，该进程中的所有线程都会更改此信息。

一个线程可以调用以下两个函数来获取和设置其进程的当前驱动器和目录：

```
DWORD GetCurrentDirectory(
    DWORD cchCurDir,
    PTSTR pszCurDir);
BOOL SetCurrentDirectory(PCTSTR pszCurDir);
```

如果提供的缓冲区不够大，**GetCurrentDirectory**将返回保存此文件夹所需要的字符数（包括末尾的'\0'字符），而且不会向你提供的缓冲区复制任何内容。在此情况下，可以将缓冲区设为**NULL**。如果调用成功，就会返回字符串的长度（字符数）。在这个长度中，末尾的'\0'不计算在内。

> 注意
>
> WinDef.h文件中被定义为260的常量MAX_PATH是目录名称或文件名称的最大字符数。所以在调用GetCurrentDirectory的时候 ,向该函数传递由MAX_PATH个TCHAR类型的元素构成的一个缓冲区是非常安全的。

4.1.9 进程的当前目录

系统跟踪记录着进程的当前驱动器和目录，但它没有记录每个驱动器的当前目录。不过，利用操作系统提供的支持，可以处理多个驱动器的当前目录。这个支持是通过进程的环境字符串来提供的。例如，一个进程可以有如下所示的两个环境变量：

```
=C:=C:\Utility\Bin
=D:=D:\Program Files
```

上述变量指出进程在C驱动器的当前目录为\Utility\Bin，在D驱动器的当前目录为\Program Files。如果调用一个函数，向其传递一个限定了驱动器的名称，而且指定的驱动器不是当前驱动器，系统会在进程的环境块中查找与指定盘符（中文版Windows中翻译成“驱动器号”）关联的变量。如果找到与指定盘符关联的变量，系统就将变量的值作为当前目录使用。如果变量没有找到，系统就假定指定驱动器的当前目录是它的根目录。

例如，假定进程的当前目录为C:\Utility\Bin，而且你调用**CreateFile**来打开D:ReadMe.Txt，那么系统就会查找环境变量**=D:**。由于**=D:**变量是存在的，所以系统将尝试从D:\Program Files目录打开ReadMe.Txt文件。如果**=D:**变量不存在，系统就会试着从D盘的根目录打开ReadMe.Txt文件。Windows的文件函数从来不会添加或更改盘符环境变量——它们只是读取这种变量。

> 注意
> 可以使用C运行库函数_chdir而不是Windows SetCurrentDirectory函数来更改当前目录。_chdir函数在内部调用SetCurrentDirectory，但_chdir还可以调用SetEnvironmentVariable来添加或修改环境变量，从而使不同驱动器的当前目录得以保留。

如果一个父进程创建了一个希望传给子进程的环境块，子进程的环境块就不会自动继承父进程的当前目录。相反，子进程的当前目录默认为每个驱动器的根目录。如果希望子进程继承父进程的当前目录，父进程就必须在生成子进程之前，创建这些盘符环境变量，并把它们添加到环境块中。父进程可以通过调用**GetFullPathName**来获得它的当前目录：

```
DWORD GetFullPathName(
    PCTSTR pszFile,
    DWORD cchPath,
    PTSTR pszPath,
    PTSTR *ppszFilePart);
```

例如，要想获得C驱动器的当前目录，可以像下面这样调用**GetFullPathName**：

```
TCHAR szCurDir[MAX_PATH];
DWORD cchLength = GetFullPathName(TEXT("C:"), MAX_PATH, szCurDir, NULL);
```

其结果就是，盘符环境变量通常必须放在环境块的开始处。

4.1.10 系统版本

很多时候，应用程序需要判断用户所运行的Windows系统的版本。例如，应用程序也许会调用**CreateFileTransacted**之类的函数，以利用Windows的事务处理式（transacted）文件系统功能。但是，这些函数目前仅在Windows Vista上完整实现。

在很长的时间里，Windows应用程序编程接口（Application Programming Interface，API）一直在提供一个**GetVersion**函数：

```
DWORD GetVersion();
```

这个函数具有悠久的历史。它最初是为16位Windows系统设计的。其思路非常简单，在高字（high word）中返回MS-DOS版本号，在低字（low word）中返回Windows版本号。在每个word中，高字节（high byte）代表major版本号，低字节（low byte）代表minor版本号。

遗憾的是，写代码的程序员犯了一个小错误，造成Windows版本号的顺序颠倒了，即major版本号跑到了低字节，minor版本号跑到了高字节。由于许多程序员已经开始使用这个函数，所以Microsoft被迫保留这个函数的错误形式，并修改了它的文档，以指明这个错误。

鉴于围绕着**GetVersion**而产生的一些困惑，Microsoft添加了一个新的函数**GetVersionEx**，如下所示：

```
BOOL GetVersionEx(POSVERSIONINFOEX pVersionInformation);
```

这个函数要求你在自己的应用程序中分配一个**OSVERSIONINFOEX**结构，并把此结构的地址传给**GetVersionEx**。**OSVERSIONINFOEX**结构如下所示：

```
typedef struct {
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
    DWORD dwBuildNumber;
    DWORD dwPlatformId;
    TCHAR szCSDVersion[128];
    WORD wServicePackMajor;
    WORD wServicePackMinor;
    WORD wSuiteMask;
    BYTE wProductType;
    BYTE wReserved;
} OSVERSIONINFOEX, *POSVERSIONINFOEX;
```

OSVERSIONINFOEX结构从Windows 2000开始就一直存在。Windows系统的其他版本使用的是较老的**OSVERSIONINFO**结构，后者没有service pack，suite mask，product type和reserved members等信息。

注意，此结构为系统版本号的每一个组成部分都提供了不同的成员。这样做是避免程序员过于麻烦地去提取low words，high words，low bytes和high bytes，使应用程序更容易将希望的版本号与主机系统的版本号进行对比。表4-4描述了**OSVERSIONINFOEX**结构的成员。

表4-4 OSVERSIONINFOEX结构的成员

成员	描述
dwOSVersionInfoSize	调用GetVersionEx前，必须设为sizeof(OSVERSIONINFO)或sizeof(OSVERSIONINFOEX)
dwMajorVersion	主机系统的major版本号
dwMinorVersion	主机系统的minor版本号
dwBuildNumber	当前系统的build号
dwPlatformId	标识当前系统支持的平台。值可以是VER_PLATFORM_WIN32s（Win32s），VER_PLATFORM_WIN32_WINDOWS（Windows 95/Windows 98）或VER_PLATFORM_WIN32_NT（Windows NT/Windows 2000，Windows XP，Windows Server 2003以及Windows Vista）
szCSDVersion	此字段包含额外的文本，提供了与已安装的操作系统有关的更多的信息
wServicePackMajor	最新安装的Service Pack的major版本号
wServicePackMinor	最新安装的Service Pack的minor版本号
wSuiteMask	标识当前系统上可用的suite(s)，包括VER_SUITE_SMALLBUSINESS，VER_SUITE_ENTERPRISE，VER_SUITE_BACKOFFICE，VER_SUITE_COMMUNICATIONS，VER_SUITE_TERMINAL，VER_SUITE_SMALLBUSINESS_RESTRICTED，VER_SUITE_EMBEDDEDNT，VER_SUITE_DATACENTER，VER_SUITE_SINGLEUSERTS（每个用户一个终端服务会话），VER_SUITE_PERSONAL（区别Vista的Home和Professional版），VER_SUITE_BLADE，VER_SUITE_EMBEDDED_RESTRICTED，VER_SUITE_SECURITY_APPLIANCE，VER_SUITE_STORAGE_SERVER和VER_SUITE_COMPUTE_SERVER
wProductType	指出安装的是以下操作系统产品中的哪一个：VER_NT_WORKSTATION，VER_NT_SERVER或VER_NT_DOMAIN_CONTROLLER
wReserved	保留，供将来使用

MSDN网站的“Getting the System Version”网页（*http://msdn2.microsoft.com/en-gb/library/ms724429.aspx*）提供了基于**OSVERSIONINFOEX**结构的一个详细的代码示例，展示了如何解读这个结构的每一个字段。

为了进一步简化编程，Windows Vista还提供了**VerifyVersionInfo**函数，它能比较主机系统的版本和应用程序要求的版本，如下所示：

```
BOOL VerifyVersionInfo(
    POSVERSIONINFOEX pVersionInformation,
    DWORD dwTypeMask,
    DWORDLONG dwlConditionMask);
```

要使用此函数，必须分配一个**OSVERSIONINFOEX**结构，将它的**dwOSVersionInfoSize**成员初始化为结构的大小，然后初始化结构中对你的应用程序而言很重要的其他任何成员。调用**VerifyVersionInfo**时，**dwTypeMask**参数指出你初始化了此结构中的哪些成员。**dwTypeMask**参数是以下任何标志通过按位OR运算组合起来的结果：**VER_MINORVERSION**，**VER_MAJORVERSION**，**VER_BUILDNUMBER**，**VER_PLATFORMID**，**VER_SERVICEPACKMINOR**，**VER_SERVICEPACKMAJOR**，**VER_SUITENAME**和**VER_PRODUCT_TYPE**。最后一个参数是**dwlConditionMask**，它是一个64位值，决定了函数如何将系统的版本信息与你希望的版本信息进行比较。

dwlConditionMask使用一套复杂的位组合对比较方式进行了描述。为了创建恰当的位组合，可以使用**VER_SET_CONDITION**宏：

```
VER_SET_CONDITION(
    DWORDLONG dwlConditionMask,
    ULONG dwTypeBitMask,
    ULONG dwConditionMask)
```

第一个参数**dwlConditionMask**表示你正在对它的位进行操纵的变量。注意，不要传入这个变量的地址，因为**VER_SET_CONDITION**是一个宏，而不是一个函数。**dwTypeBitMask**参数指出**OSVERSIONINFOEX**结构中想比较的一个成员。为了比较多个成员，必须多次调用**VER_SET_CONDITION**，为每个成员都调用一次。向**VerifyVersionInfo**的参数(**VER_MINORVERSION**、**VER_BUILDNUMBER**等等)传递的标志与你为**VER_SET_CONDITION**的**dwTypeBitMask**参数使用的标志是一样的。

VER_SET_CONDITION的最后一个参数是**dwConditionMask**，它指出你想如何进行比较。它可以是下面这些值之一：**VER_EQUAL**，**VER_GREATER**，**VER_GREATER_EQUAL**，**VER_LESS**或**VER_LESS_EQUAL**。注意，在比较**VER_PRODUCT_TYPE**信息的时候，你可以使用这些值。例如，**VER_NT_WORKSTATION**小于**VER_NT_SERVER**。不过，对于**VER_SUITENAME**信息，就不能执行这种测试。相反，必须使用**VER_AND**（所有suite产品都必须安装）或**VER_OR**（至少安装了其中的一个suite产品）。

建立了一组条件之后，就可以调用**VerifyVersionInfo**。如果成功（主机系统满足你的应用程序的所有要求），它将返回一个非零的值。如果**VerifyVersionInfo**返回0，就表明主机系统不符合要求，或者表明调用函数的方式不正确。可以通过调用**GetLastError**来判断函数为什么返回0。如果**GetLastError**返回**ERROR_OLD_WIN_VERSION**，表明函数调用是正确的，但系统不符合应用程序的要求。

下面的例子展示了如何测试主机系统是不是Windows Vista：

```
// Prepare the OSVERSIONINFOEX structure to indicate Windows Vista.
OSVERSIONINFOEX osver = { 0 };
osver.dwOSVersionInfoSize = sizeof(osver);
osver.dwMajorVersion = 6;
osver.dwMinorVersion = 0;
osver.dwPlatformId = VER_PLATFORM_WIN32_NT;

// Prepare the condition mask.
DWORDLONG dwlConditionMask = 0;// You MUST initialize this to 0.
VER_SET_CONDITION(dwlConditionMask, VER_MAJORVERSION, VER_EQUAL);
VER_SET_CONDITION(dwlConditionMask, VER_MINORVERSION, VER_EQUAL);
VER_SET_CONDITION(dwlConditionMask, VER_PLATFORMID, VER_EQUAL);

// Perform the version test.
if (VerifyVersionInfo(&osver, VER_MAJORVERSION | VER_MINORVERSION | VER_PLATFORMID,
   dwlConditionMask)) {
   // The host system is Windows Vista exactly.
} else {
   // The host system is NOT Windows Vista.
}
```

4.2 CreateProcess 函数

我们用**CreateProcess**函数来创建一个进程，如下所示：

```
BOOL CreateProcess(
  PCTSTR pszApplicationName,
  PTSTR pszCommandLine,
  PSECURITY_ATTRIBUTES psaProcess,
  PSECURITY_ATTRIBUTES psaThread,
  BOOL bInheritHandles,
  DWORD fdwCreate,
  PVOID pvEnvironment,
  PCTSTR pszCurDir,
  PSTARTUPINFO psiStartInfo,
  PPROCESS_INFORMATION ppiProcInfo);
```

一个线程调用**CreateProcess**时，系统将创建一个进程内核对象，其初始使用计数为1。进程内核对象不是进程本身，而是操作系统用来管理这个进程的一个小型数据结构——可以把进程内核对象想象成由进程统计信息构成的一个小型数据结构。然后，系统为新进程创建一个虚拟地址空间，并将执行体文件（和所有必要的DLL）的代码及数据加载到进程的地址空间。

然后，系统为新进程的主线程创建一个线程内核对象（使用计数为1）。和进程内核对象一样，线程内核对象也是一个小型数据结构，操作系统用它来管理这个线程。这个主线程一开始就会执行由链接器设为应用程序入口的C/C++运行时启动例程，并最终调用你的**WinMain**，**wWinMain**，**main**或**wmain**函数。如果系统成功创建了新进程和主线程，**CreateProcess**将返回**TRUE**。

> **注意**
> **CreateProcess**在进程完全初始化好之前就返回**TRUE**。这意味着操作系统加载器（loader）尚未尝试定位所有必要的DLL。如果一个DLL找不到或者不能正确初始化，进程就会终止。因为**CreateProcess**返回**TRUE**，所以父进程不会注意到任何初始化问题。

OK，前面只是泛泛而谈，下面将分小节逐一讨论**CreateProcess**的参数。

4.2.1 pszApplicationName 和 pszCommandLine 参数

pszApplicationName和**pszCommandLine**参数分别指定新进程要使用的执行体文件的名称，以及要传给新进程的命令行字符串。先来谈谈**pszCommandLine**参数。

注意，**pszCommandLine**参数被原型化为一个**PTSTR**。这意味着**CreateProces**期望你传入的是一个非“常量字符串”的地址。在内部，**CreateProcess**实际上会修改你传给它的命令行字符串。但在**CreateProcess**返回之前，它会将这个字符串还原为原来的形式。

这是很重要的，因为如果命令行字符串包含在你的文件映像的只读部分，就会引起访问冲突（违例）。例如，以下代码就会导致冲突，因为Microsoft的C/C++编译器把"NOTEPAD"字符串放在只读内存中：

```
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
CreateProcess(NULL, TEXT("NOTEPAD"), NULL, NULL,
   FALSE, 0, NULL, NULL, &si, &pi);
```

CreateProcess试图修改字符串时，会引起一个访问冲突（Microsoft C/C++编译器的早期版本把字符串放在可读/写内存中。所以对**CreateProcess**函数的调用不会引起访问冲突）。
解决这个问题的最佳方式是：在调用**CreateProcess**之前，把常量字符串复制到一个临时缓冲区，如下所示：

```
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
TCHAR szCommandLine[] = TEXT("NOTEPAD");
CreateProcess(NULL, szCommandLine, NULL, NULL,
   FALSE, 0, NULL, NULL, &si, &pi);
```

还可以使用Microsoft C++的**/Gf**和**/GF**编译器开关，它们可以消除重复的字符串，以及判断那些字符串是否放在一个只读区域。(还要注意**/ZI**开关，它允许使用Visual Studio的Edit & Continue调试功能，它包含了**/GF**开关的功能。)最佳做法是使用**/GF**编译器开关和一个临时缓冲区。目前，Microsoft最应该做的一件事情就是修复**CreateProcess**，使它自己就能创建字符串的一个临时拷贝，从而使我们得到解放。Windows未来的版本或许会进行这个修复。

顺带提一下，如果在Windows Vista上调用**CreateProcess**函数的ANSI版本，就不会发生访问冲突，因为会创建命令行字符串的一个临时拷贝（详情参见第2章）。

可以使用**pszCommandLine**参数来指定一个完整的命令行，供**CreateProcess**用于创建新进程。当**CreateProcess**解析**pszCommandLine**字符串时，它会检查字符串中的第一个标记（token），并假定此标记是你想运行的执行体文件的名称。如果执行体文件的名称没有扩展名，就会默认是.exe扩展名。**CreateProcess**还会按照以下顺序搜索执行体。

1. 主调进程.EXE文件所在的目录
2. 主调进程的当前目录
3. Windows系统目录，即GetSystemDirectory返回的System32子文件夹
4. Windows目录
5. PATH环境变量中列出的目录

当然，假如文件名包含一个完整路径，系统就会利用这个完整路径来查找执行体，而不会搜索目录。如果系统找到了执行体文件，就创建一个新进程，并将执行体的代码和数据映射到新进程的地址空间。然后，系统调用由链接器设为应用程序入口的C/C++运行时启动例程。如前所

述，C/C++运行时启动例程会检查进程的命令行，将执行体文件名之后的第一个实参的地址传给 **(w)WinMain**的**pszCmdLine**参数。

只要**pszApplicationName**参数为NULL（99%以上的情况都是如此），就会发生上述情况。但是，也可以不在**pszApplicationName**中传递NULL，而是传递一个字符串地址，并在字符串中包含想要运行的执行体文件的名称。但在这种情况下，你就必须指定文件扩展名，系统不会自动假定文件名有一个.exe扩展名。**CreateProcess**假定文件位于当前目录，除非文件名前有一个路径。如果没有在当前目录中找到文件，**CreateProcess**不会在其他任何目录查找文件——调用会以失败而告终。

然而，即使在**pszApplicationName**参数中指定了文件名，**CreateProcess**也会将**pszCommandLine**参数中的内容作为新进程的命令行传给它。例如，假设像下面这样调用**CreateProcess**：

```
// Make sure that the path is in a read/write section of memory.
TCHAR szPath[] = TEXT("WORDPAD README.TXT");

// Spawn the new process.
CreateProcess(TEXT("C:\\WINDOWS\\SYSTEM32\\NOTEPAD.EXE"),szPath,...);
```

系统会调用记事本应用程序，但记事本程序的命令行是WORDPAD README.TXT。虽然这看起来有点儿怪异，但**CreateProcess**的工作机制就是这样的。之所以为**CreateProcess**添加**pszApplicationName**参数的这个能力，实际是为了支持Windows的POSIX子系统。

4.2.2 psaProcess，psaThread 和 bInheritHandles 参数

为了创建一个新的进程，系统必须创建一个进程内核对象和一个线程内核对象（用于进程的主线程）。由于这些都是内核对象，所以父进程有机会将安全属性关联到这两个对象上。可以分别使用**psaProcess**和**psaThread**参数为进程对象和线程对象指定你希望的安全性。可以为这两个参数传递**NULL**；在这种情况下，系统将为这两个内核对象指定默认的安全描述符。也可以分配并初始化两个**SECURITY_ATTRIBUTES**结构，以便创建你自己的安全权限，并将它们分配给进程对象和线程对象。

为**psaProcess**和**psaThread**参数使用**SECURITY_ATTRIBUTES**结构的另一个原因是：这两个对象句柄可由父进程将来生成的任何子进程继承（第3章讨论了内核对象句柄的继承机制）。

下面展示的Inherit.cpp是一个简单的程序，它演示了内核对象句柄的继承。假设现在由Process A来创建Process B，它调用CreateProcess，并为psaProcess参数传入一个SECURITY_ATTRIBUTES结构的地址（在这个结构中，bInheritHandle成员被设为TRUE）。在同一个调用中，psaThread 参数指向另一个SECURITY_ATTRIBUTES结构，该结构的bInheritHandle成员被设为FALSE。

系统创建Process B时，会同时分配一个进程内核对象和一个线程内核对象，并在ppiProcInfo参数指向的一个结构中，将句柄返回给Process A。ppiProcInfo参数的详情将在稍后讨论。现在，利用返回的这些句柄，Process A就可以操纵新建的进程对象和线程对象。

现在，假设Process A再次调用**CreateProcess**来创建Process C。Process A可以决定是否允许Process C操纵Process A能访问的一些内核对象。**bInheritHandles**参数便是针对这个用途而提供的。如果**bInheritHandles**设为**TRUE**，Process C将继承Process A中的所有可继承的句柄。在本例中，Process B的进程对象句柄是可继承的。Process B的主线程对象的句柄则是不可继承的，不管传给**CreateProcess**的**bInheritHandles**参数值是多少。另外，如果 Process A调用

CreateProcess，并为**bInheritHandles**参数传入**FALSE**，则Process C不会继承Process A当前所用的任何一个句柄。

Inherit.cpp

```
/******************************************************************
Module name: Inherit.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************/

#include <Windows.h>

int WINAPI _tWinMain (HINSTANCE hInstanceExe, HINSTANCE,
   PTSTR pszCmdLine, int nCmdShow) {

   // Prepare a STARTUPINFO structure for spawning processes.
   STARTUPINFO si = { sizeof(si) };
   SECURITY_ATTRIBUTES saProcess, saThread;
   PROCESS_INFORMATION piProcessB, piProcessC;
   TCHAR szPath[MAX_PATH];

   // Prepare to spawn Process B from Process A.
   // The handle identifying the new process
   // object should be inheritable.
   saProcess.nLength = sizeof(saProcess);
   saProcess.lpSecurityDescriptor = NULL;
   saProcess.bInheritHandle = TRUE;

   // The handle identifying the new thread
   // object should NOT be inheritable.
   saThread.nLength = sizeof(saThread);
   saThread.lpSecurityDescriptor = NULL;
   saThread.bInheritHandle = FALSE;

   // Spawn Process B.
   _tcscpy_s(szPath, _countof(szPath), TEXT("ProcessB"));
   CreateProcess(NULL, szPath, &saProcess, &saThread,
      FALSE, 0, NULL, NULL, &si, &piProcessB);

   // The pi structure contains two handles
   // relative to Process A:
   // hProcess, which identifies Process B's process
   // object and is inheritable; and hThread, which identifies
   // Process B's primary thread object and is NOT inheritable.
   // Prepare to spawn Process C from Process A.
   // Since NULL is passed for the psaProcess and psaThread
   // parameters, the handles to Process C's process and
   // primary thread objects default to "noninheritable."
   // If Process A were to spawn another process, this new
   // process would NOT inherit handles to Process C's process
   // and thread objects.
   // Because TRUE is passed for the bInheritHandles parameter,
   // Process C will inherit the handle that identifies Process
   // B's process object but will not inherit a handle to
   // Process B's primary thread object.
   _tcscpy_s(szPath, _countof(szPath), TEXT("ProcessC"));
   CreateProcess(NULL, szPath, NULL, NULL,
      TRUE, 0, NULL, NULL, &si, &piProcessC);

   return(0);
}
```

4.1.3 fdwCreate 参数

fdwCreate参数标识了影响新进程创建方式的标志（flag）。多个标志可以使用按位OR运算符来组合。可用的标志如下。

- **DEBUG_PROCESS**标志向系统表明父进程希望调试子进程以及子进程将来生成的所有进程。该标志向系统表明，在任何一个子进程（现在的身份是被调试对象，或者说debugee）

中发生特定的事件时，要通知父进程（现在的身份是调试器，或者说debugger）。

- **DEBUG_ONLY_THIS_PROCESS**标志类似于**DEBUG_PROCESS**，但是，只有在关系最近的子进程中发生特定事件时，父进程才会得到通知。如果子进程又生成了新的进程，那么在这些新进程中发生特定事件时，调试器是不会得到通知的。要进一步了解如何利用这两个标志来写一个调试器，并获取被调试应用程序中的DLL和线程的信息，请阅读MSDN的一篇文章：“Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities, Part 2”，网址是*http://msdn.microsoft.com/msdnmag/issues/02/08/EscapefromDLLHell/*。
- **CREATE_SUSPENDED**标志造成系统创建新进程，但会挂起其主线程。这样一来，父进程就可以修改子进程地址空间中的内存，更改子进程的主线程的优先级，或者在进程执行任何代码之前，将此进程添加到一个作业（job）中。父进程修改好子进程之后，可以调用**ResumeThread**函数来允许子进程执行代码。欲知这个函数的详情，请参见第7章。
- **DETACHED_PROCESS**标志阻止一个基于CUI（控制台用户界面）的进程访问其父进程的控制台窗口，并告诉系统将它的输出发送到一个新的控制台窗口，如果一个基于CUI的进程是由另一个基于CUI的进程创建的，那么在默认情况下，新进程将使用父进程的控制台窗口。（在命令提示符中运行C++编译器的时候，不会新建一个控制台窗口；输出将附加到现有控制台窗口的底部。）通过指定这个标志，新进程如果需要将输出发送到一个新的控制台窗口，就必须调用**AllocConsole**函数来创建它自己的控制台。
- **CREATE_NEW_CONSOLE**标志指示系统为新进程创建一个新的控制台窗口。如果同时指定 **CREATE_NEW_CONSOLE**和**DETACHED_PROCESS**标志，会导致一个错误。
- **CREATE_NO_WINDOW**标志指示系统不要为应用程序创建任何控制台窗口。可以使用这个标志来执行没有用户界面的控制台应用程序。
- **CREATE_NEW_PROCESS_GROUP**标志修改用户按Ctrl+C或Ctrl+Break时获得通知的进程列表。按下这些组合键时，假如有多个CUI进程正在运行，系统将通知一个进程组中的所有进程，告诉它们用户打算中断当前操作。创建一个新的CUI进程时，假如指定了这个标志，就会创建一个新的进程组。组中的一个进程处于活动状态时，一旦用户按下Ctrl+C或Ctrl+Break，系统就只是向这个组的进程发出通知。
- **CREATE_DEFAULT_ERROR_MODE**标志向系统表明新进程不会继承父进程所用的错误模式。（本章前面已经讨论了**SetErrorMode**函数。）
- **CREATE_SEPARATE_WOW_VDM**标志只有在你运行16位Windows应用程序时才有用。它指示系统创建一个单独的虚拟DOS机（Virtual DOS Machine，VDM），并在这个VDM上运行16位Windows应用程序。默认情况下，所有16位Windows应用程序都在一个共享的VDM中执行。在独立的VDM中运行的好处是，假如应用程序崩溃，它只需要杀死这个VDM，在其他VDM中运行的其他程序仍然能正常工作。另外，在独立的VDM中运行的16位Windows应用程序有独立的输入队列。这意味着假如一个应用程序暂时挂起，独立VDM中运行的应用程序仍然能接收输入。运行多个VDM的缺点在于，每个VDM都要消耗较多的物理内存。Windows 98在单独一个虚拟机中运行所有16位Windows应用程序——你不能覆盖这个行为。
- **CREATE_SHARED_WOW_VDM**标志只有在你运行16位Windows应用程序时才有用。默认情况下，所有16位Windows应用程序都在单独一个VDM中运行的，除非指定了**CREATE_SEPARATE_WOW_VDM**标志。不过，你也可以覆盖这个默认行为。办法是在注册表中将HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\WOW下**DefaultSeparateVDM**的值设为yes。在此之后，如果设置**CREATE_SHARED_WOW_VDM**标志，16位Windows应用程序就会在系统的共享VDM中运行。（修改了这个注册表设置后，必须重启电脑。）注意，为了检测在64位操作系统下运行的32位进程，你可以调用**IsWow64Process**函数，它的第一个参数是你要检测的进程的句柄，第二个参数则是指向一个Boolean值的指针；如果进程是在WOW64下运行，这个值就会设为TRUE；否则会设为

FALSE。

- **CREATE_UNICODE_ENVIRONMENT**标志告诉系统子进程的环境块应包含Unicode字符。进程的环境块默认包含的是ANSI字符串。
- **CREATE_FORCEDOS**标志强制系统运行嵌入一个16位OS/2应用程序中的MS-DOS应用程序。
- **CREATE_BREAKAWAY_FROM_JOB**标志允许一个作业中的进程生成一个和作业无关的进程。(详情参见第5章。)
- **EXTENDED_STARTUPINFO_PRESENT**标志向操作系统表明一个**STARTUPINFOEX**结构被传给了**psiStartInfo**参数。

fdwCreate参数还允许你指定一个优先级类（priority class）。不过，这样做没有多大必要，而且对于大多数应用程序，都不应该这样做——系统会为新进程分配一个默认的优先级类。表4-5展示了可能的优先级类。

表4-5 fdwCreate参数设置的优先级类

优先级类	标志
Idle	IDLE_PRIORITY_CLASS
Below normal	BELOW_NORMAL_PRIORITY_CLASS
Normal	NORMAL_PRIORITY_CLASS
Above normal	ABOVE_NORMAL_PRIORITY_CLASS
High	HIGH_PRIORITY_CLASS
Realtime	REALTIME_PRIORITY_CLASS

这些优先级类决定了相对于其他进程的线程，这个进程中的线程的调度方式，详情参见第 7 章的“优先级的抽象视图”一节。

4.2.3 pvEnvironment 参数

pvEnvironment参数指向一个内存块，其中包含了新进程要使用的环境字符串。大多数时候，为这个参数传入的值都是**NULL**，这将导致子进程继承其父进程使用的一组环境字符串。另外，还可以使用**GetEnvironmentStrings**函数：

```
PVOID GetEnvironmentStrings();
```

此函数获取主调进程正在使用的环境字符串数据块的地址。可以将这个函数返回的地址用作**CreateProcess**函数的**pvEnvironment**参数的值。如果你为**pvEnvironment**参数传入NULL值，**CreateProcess**函数就会这样做。不再需要这个内存块的时候，应该调用**FreeEnvironmentStrings**函数来释放它：

```
BOOL FreeEnvironmentStrings(PTSTR pszEnvironmentBlock);
```

4.2.4 pszCurDir 参数

pszCurDir参数允许父进程设置子进程的当前驱动器和目录。如果这个参数为NULL，则新进程的工作目录与生成新进程的应用程序一样。如果这个参数不为NULL，则**pszCurDir**必须指向一

个用0来终止的字符串，其中包含了你希望的工作驱动器和目录。注意，必须在路径中指定一个盘符（驱动器号）。

4.2.5 psiStartInfo 参数

psiStartInfo参数指向一个**STARTUPINFO**结构或**STARTUPINFOEX**结构：

```
typedef struct _STARTUPINFO {
DWORD cb;
PSTR lpReserved;
PSTR lpDesktop;
PSTR lpTitle;
DWORD dwX;
DWORD dwY;
DWORD dwXSize;
DWORD dwYSize;
DWORD dwXCountChars;
DWORD dwYCountChars;
DWORD dwFillAttribute;
DWORD dwFlags;
WORD wShowWindow;
WORD cbReserved2;
PBYTE lpReserved2;
HANDLE hStdInput;
HANDLE hStdOutput;
HANDLE hStdError;
} STARTUPINFO, *LPSTARTUPINFO;
typedef struct _STARTUPINFOEX {
STARTUPINFO StartupInfo;
struct _PROC_THREAD_ATTRIBUTE_LIST *lpAttributeList;
} STARTUPINFOEX, *LPSTARTUPINFOEX;
```

Windows在创建新进程的时候使用这个结构的成员。大多数应用程序都希望生成的应用程序只是使用默认值。最起码要将此结构中的所有成员初始化为0，并将*cb*成员设为此结构的大小，如下所示：

```
STARTUPINFO si = { sizeof(si) };
CreateProcess(..., &si, ...);
```

如果没有把结构的内容清零，则成员将包含主调线程的堆栈上的垃圾数据。把这种垃圾传给**CreateProcess**，会造成新进程有时能创建，有时则不能，具体取决于垃圾的内容。因此，必须将这个结构中的未使用的成员清零，确保**CreateProcess**始终都能正常地工作。这是很容易犯的一个错误，许多开发人员都忘记了做这个工作。

现在，如果想初始化此结构中的某些成员，只需在调用**CreateProcess**之前完成这些初始化即可。我们将依次讨论每一个成员。一些成员仅在子应用程序创建了一个重叠的窗口时才有意义；另一些成员仅在子应用程序执行CUI输入和输出时才有意义。表4-6描述了每个成员的用途。

表4-6 STARTUPINFO和STARTUPINFOEX结构的成员

成员	窗口，控制台，或两者	用途
cb	两者	包含STARTUPINFO结构中的字节数。充当版本控制，以备Microsoft未来扩展这个结构之用（就像STARTUPINFOEX那样）。应用程序必须将cb初始化为sizeof(STARTUPINFO)或sizeof(STARTUPINFOEX)。
lpReserved	两者	保留。必须初始化为NULL。
lpDesktop	两者	标识要在上面启动应用程序的桌面的名称。如果桌面已经存在，则新进程会与指

		定的桌面关联。如果桌面不存在，则用指定的名称和默认的属性为新进程创建一个桌面。如果lpDesktop为NULL（这是最为常见的），进程就会与当前桌面关联。
lpTitle	控制台	指定控制台窗口的窗口标题。如果lpTitle被设置为NULL，就将执行体文件的名称作为窗口标题。
dwX dwY	两者	指定应用程序窗口在屏幕上的位置（即x和y坐标，以像素为单位）。只有在子过程用CW_USEDEFAULT作为CreateWindow函数的x参数来创建其第一个重叠窗口的时候，才会用到这些坐标。对于创建控制台窗口的应用程序，这些成员指定的是控制台窗口的左上角位置。
dwXSize dwYSize	两者	指定应用程序窗口的宽度和高度（以像素为单位）。只有在子进程将CW_USEDEFAULT作为CreateWindow函数的nWidth参数来创建其第一个重叠窗口的时候，才会用到这些值。对于创建控制台窗口的应用程序，这些成员指定的是控制台窗口的宽度和高度。
dwXCountChars dwYCountChars	控制台	指定子进程的控制台窗口的宽度和高度（用字符数来表示）。
dwFillAttribute	控制台	指定子进程的控制台窗口所用的文本和背景色。
dwFlags	两者	参见下一小节和表4-7
wShowWindow	窗口	指定应用程序的主窗口如何显示。在第一个ShowWindow调用中，将使用wShowWindow的值，并忽略ShowWindow的nCmdShow参数。在后续的ShowWindow调用中，只有在将 SW_SHOWDEFAULT传给ShowWindow函数的前提下，才会使用wShowWindow的值。注意，除非dwFlags指定了STARTF_USESHOWWINDOW标志，否则wShowWindow会被忽略。
cbReserved2	两者	保留。必须被初始化为0
lpReserved2	两者	保留。必须被初始化为NULL。cbReserved2和lpReserved2供C运行时使用，在_dospawn被用于启动一个应用程序的时候，C运行时将用它们来传递信息。要了解实现细节，请参见Visual Studio目录下的VC\crt\src\子目录中的dospawn.c和ioinit.c文件。
hStdInput hStdOutput hStdError	控制台	指定到控制台输入和输出缓冲区的句柄。默认情况下，hStdInput标识一个键盘缓冲区，hStdOutput和hStdError标识一个控制台窗口的缓冲区。这些字段用于重定向子进程的输入/输出，详情参见MSDN文章“How to spawn console processes with redirected standard handles”，网址是*http://support.microsoft.com/kb/190351*。

为了履行前面的承诺，现在让我们讨论一下**dwFlags**成员。这个成员包含一组标志，用于修改子进程的创建方式。大多数标志都只是告诉**CreateProcess**函数：P**STARTUPINFO**结构中的其他成员是否包含有用的信息，或者是否应该忽略一些成员。表4-7展示了可能的标志及其含义。

表4-7　dwFlags的标志

标志	含义
STARTF_USESIZE	使用dwXSize和dwYSize成员
STARTF_USESHOWWINDOW	使用wShowWindow成员
STARTF_USEPOSITION	使用dwX and dwY成员
STARTF_USECOUNTCHARS	使用dwXCountChars和dwYCountChars成员
STARTF_USEFILLATTRIBUTE	使用dwFillAttribute成员
STARTF_USESTDHANDLES	使用hStdInput，hStdOutput和hStdError成员
STARTF_RUNFULLSCREEN	使x86计算机上运行的一个控制台应用程序以全屏模式启动

另外还有两个标志，即**STARTF_FORCEONFEEDBACK**和**STARTF_FORCEOFFFEEDBACK**，它们可以在你调用一个新进程时控制鼠标指针。由于Windows支持真正的抢占多任务处理，所以你可以调用（或“唤出”，即invoke）一个应用程序，并在进程初始化期间使用另一个程序。为了向用户提供视觉反馈，**CreateProcess**临时将系统的鼠标指针（光标）改为一个特殊的形状：

这个光标指出你既可以等待某件事情的发生，也可以继续使用系统。**CreateProcess**函数允许你在调用另一个进程时对光标进行更多的控制。如果指定**STARTF_FORCEOFFFEEDBACK**标志， **CreateProcess**就不会把指针改为上述特殊的形状。

STARTF_FORCEONFEEDBACK会令**CreateProcess**监视新进程的初始化过程，并根据结果更改光标形状。如果调用**CreateProcess**时设置了这个标志，指针就会更改为上述特殊的形状。在两秒之后，如果新进程并没有执行任何GUI调用，**CreateProcess**就会将光标重置为普通的箭头形状。

如果进程在两秒内执行了一个GUI调用，则**CreateProcess**函数会等待应用程序显示一个窗口。这必须在进程执行了GUI调用之后的5秒钟之内发生。如果没有显示窗口，**CreateProcess**会重置光标。如果显示了窗口，**CreateProcess**会继续保持特殊形状的光标5秒钟。在任何时候，一旦应用程序调用了**GetMessage**函数（表明初始化已完毕），**CreateProcess**就会立即重置光标，并停止监视新进程。

可以把**wShowWindow**成员初始化为传给**(w)WinMain**函数最后一个参数**nCmdShow**的值。这个成员指出你想传给新进程的**(w)WinMain**函数的最后一个参数**nCmdShow**的值。它是可以传给**ShowWindow**函数的标识符之一。通常，**nCmdShow**的值要么是**SW_SHOWNORMAL**，要么是**SW_SHOWMINNOACTIVE**。不过，它有时也可能是**SW_SHOWDEFAULT**。

从Windows资源管理器调用一个应用程序的时候，此应用程序的**(w)WinMain**函数会被调用，**SW_SHOWNORMAL**会作为**nCmdShow**参数的值传入。如果为应用程序创建一个快捷方式，可以在快捷方式的属性页中，告诉系统应用程序的窗口最初应该如何显示。图4-3显示了一个快捷方式的属性页，此快捷方式用于启动“记事本”程序。注意，可以通过“运行”（Run）选项的组合框来指定如何显示记事本程序的窗口。

在Windows资源管理器中启动这个快捷方式时，Windows资源管理器将正确准备**STARTUPINFO**结构，并调用**CreateProcess**。“记事本”将执行，其**(w)WinMain**函数的**nCmdShow**参数会被传入**SW_SHOWMINNOACTIVE**。

通过这种方式，用户就可以轻松启动一个应用程序，指定以常规状态、最小化状态或者最大化状态显示其主窗口。

在结束本小节的讨论之前，我想强调一下**STARTUPINFOEX**结构的角色。自Win32问世以来，**CreateProcess**的签名一直没有变过。Microsoft的策略是在保持函数签名不变的同时提供更多的扩展，并避免专门创建一个**CreateProcessEx**，再专门创建一个**CreateProcess2**……以此类推。所以，除了预期的**StartupInfo**字段之外，**STARTUPINFOEX**结构另外只有一个字段，即**lpAttributeList**，后者用于传递额外的称为“属性”（attribute）的参数：

```
typedef struct _STARTUPINFOEXA {
   STARTUPINFOA StartupInfo;
   struct _PROC_THREAD_ATTRIBUTE_LIST *lpAttributeList;
} STARTUPINFOEXA, *LPSTARTUPINFOEXA;
typedef struct _STARTUPINFOEXW {
   STARTUPINFOW StartupInfo;
   struct _PROC_THREAD_ATTRIBUTE_LIST *lpAttributeList;
} STARTUPINFOEXW, *LPSTARTUPINFOEXW;
```

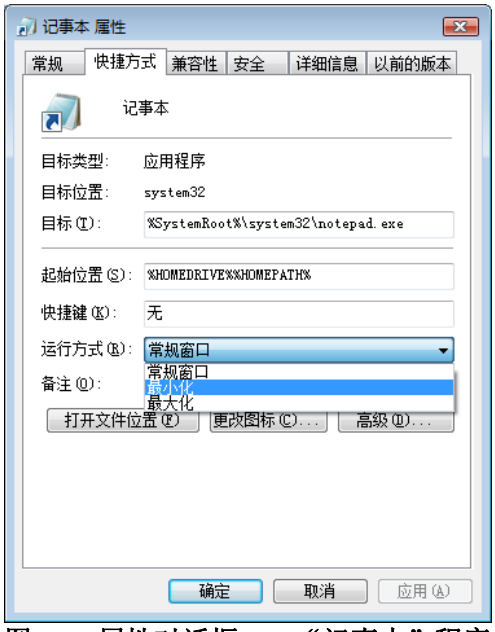


图4-3　属性对话框——“记事本”程序的快捷方式

在attribute列表中，包含了一系列key/value对，每个attribute都有一对key/value。目前，仅有两个attribute key被写入文档：

- **PROC_THREAD_ATTRIBUTE_HANDLE_LIST**　这个attribute key告诉**CreateProcess**子进程究竟应该继承哪一些内核对象句柄。这些对象句柄在创建时必须指定成“可继承”（在**SECURITY_ATTRIBUTES**结构中包含一个设为TRUE的**bInheritHandle**字段，详情参见前面的“psaProcess，psaThread和bInheritHandles”小节）。不过，并非一定要为**CreateProcess**函数的**bInheritHandles**参数传入**TRUE**。使用这个attribute，子进程只能继承一组选定的句柄，而不是继承所有可继承的句柄。假如一个进程要在不同的安全上下文中创建多个子进程，这一点就尤其重要。在这种情况下，不能让每个子进程都继承父进程的全部句柄，否则会导致安全问题。
- **PROC_THREAD_ATTRIBUTE_PARENT_PROCESS**　这个attribute key预期的值是一个进程句柄。指定的进程（而不是当前调用CreateProcess的进程）将成为正在创建的这个进程的父进程。从指定进程继承的属性包括可继承的句柄、处理器亲和性、优先级类、配额（quota）、用户令牌（user token）以及关联的作业。注意，以这种改变父进程后，不会改变调试器进程（debugger）和被调试进程（debuggee）的关系。换言之，仍然会创建一个被调试进程，而你指定的父进程仍然负责接收调试通知，并管理管理被调试进程的生存期。在本章后面的“枚举系统中正在运行的进程”小节，我们给出了一个ToolHelp API，它展示了如何将这个attribute指定的进程作为要创建的一个进程的父进程来使用。

由于attribute列表是不透明的，所以要调用以下函数两次，才能创建一个空白的attribute列表：

```
BOOL InitializeProcThreadAttributeList(
  PPROC_THREAD_ATTRIBUTE_LIST pAttributeList,
  DWORD dwAttributeCount,
  DWORD dwFlags,
  PSIZE_T pSize);
```

注意，**dwFlags**参数是保留的，而且你始终都要为这个参数传入0。第一个调用的目的是知道Windows用来保存attributes的内存块的大小：

```
SIZE_T cbAttributeListSize = 0;
BOOL bReturn = InitializeProcThreadAttributeList(
   NULL, 1, 0, &cbAttributeListSize);
// bReturn is FALSE but GetLastError() returns ERROR_INSUFFICIENT_BUFFER
```

pSize指向的**SIZE_T**变量将接收内存块的大小值，这个内存块是根据**dwAttributeCount**所指定的attribute的数目来分配的：

```
pAttributeList = (PPROC_THREAD_ATTRIBUTE_LIST)
   HeapAlloc(GetProcessHeap(), 0, cbAttributeListSize);
```

为attribute列表分配了内存之后，要再次调用**InitializeProcThreadAttributeList**来初始化它的内容（这些内容是“不透明”的）：

```
bReturn = InitializeProcThreadAttributeList(
   pAttributeList, 1, 0, &cbAttributeListSize);
```

分配并初始化好attribute列表之后，就可以根据自己的需要，用下面的函数来添加key/value对：

```
BOOL UpdateProcThreadAttribute(
   PPROC_THREAD_ATTRIBUTE_LIST pAttributeList,
   DWORD dwFlags,
   DWORD_PTR Attribute,
   PVOID pValue,
   SIZE_T cbSize,
   PVOID pPreviousValue,
   PSIZE_T pReturnSize);
```

pAttributeList参数是之前分配并初始化的attribute列表，函数将在其中添加一个新的key/value对。**Attribute**参数是其中的key部分，它要么接收**PROC_THREAD_ATTRIBUTE_PARENT_PROCESS**，要么接收**PROC_THREAD_ATTRIBUTE_HANDLE_LIST**值。如果是前者，**pValue**参数必须指向一个变量，该变量包含了新的父进程的句柄，而**cbSize**应该用**sizeof(HANDLE)**来作为它的值；如果是后者，**pValue**参数必须指向一个数组的起始位置，该数组包含了允许子进程访问的、可继承的内核对象句柄，而**cbSize**应该等于**sizeof(HANDLE)**乘以句柄数的结果。**dwFlags**，**pPreviousValue**和**pReturnSize**参数是保留参数，必须分别设为**0**，**NULL**和**NULL**。

> 警告
> 如果你同时传入两个attributes，一定要保证在与PROC_THREAD_ATTRIBUTE_PARENT_PROCESS关联的新的父进程中，与PROC_THREAD_ATTRIBUTE_HANDLE_LIST关联的句柄必须是有效的，因为这些句柄将从指定的新的父进程继承，而不是从调用CreateProcess函数的当前进程继承。

调用CreateProcess函数时，如果要在dwCreationFlags中指定EXTENDED_STARTUPINFO_PRESENT，那么在调用CreateProcess之前，就要先定义好一个STARTUPINFOEX变量（pAttributeList字段设为你刚才初始化好的attribute列表），此变量将作为**pStartupInfo**参数来使用：

```
STARTUPINFOEX esi = { sizeof(STARTUPINFOEX) };
esi.lpAttributeList = pAttributeList;
bReturn = CreateProcess(
   ..., EXTENDED_STARTUPINFO_PRESENT, ...
   &esi.StartupInfo, ...);
```

不再需要参数的时候，需要在释放已分配的内存之前，先用以下方法来清除不透明的attribute列表：

```
VOID DeleteProcThreadAttributeList(
   PPROC_THREAD_ATTRIBUTE_LIST pAttributeList);
```

最后，应用程序可以调用下面的函数来获得**STARTUPINFO**结构的一个拷贝，此结构是由父进程初始化的。子进程可以检查这个结构，并根据结构成员的值来更改其行为，如下所示：

```
VOID GetStartupInfo(LPSTARTUPINFO pStartupInfo);
```

这个函数填充的总是一个**STARTUPINFO**结构——即使在调用CreateProcess来创建当前子进程时，传递的是一个**STARTUPINFOEX**结构——attributes在父进程地址空间才有意义，只有在那个地方，才为attribute列表分配了内存。所以，正如以前解释过的那样，你需要通过另一种方式来传递已继承的句柄的值，比如通过命令行。

4.2.6 ppiProcInfo 参数

ppiProcInfo参数指向你必须分配的一个**PROCESS_INFORMATION**结构，**CreateProcess**函数在返回之前，会初始化这个结构的成员。该结构如下所示：

```
typedef struct _PROCESS_INFORMATION {
   HANDLE hProcess;
   HANDLE hThread;
   DWORD dwProcessId;
   DWORD dwThreadId;
} PROCESS_INFORMATION;
```

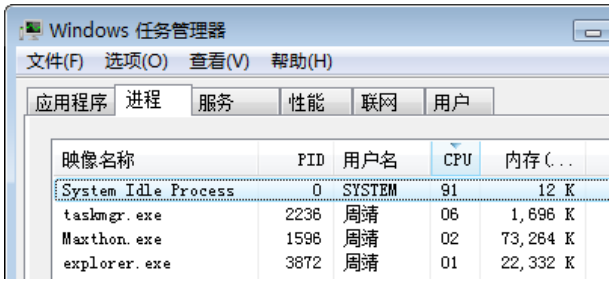
如前所述，创建一个新的进程，会导致系统创建一个**进程内核对象**和一个**线程内核对象**。在创建时，系统会为每个对象指定一个初始的使用计数1。然后，就在**CreateProcess**返回之前，它会使用完全访问权限来打开进程对象和线程对象，并将各自的与进程相关的（相对于进程的）句柄放入**PROCESS_INFORMATION**结构的**hProcess**和**hThread**成员中。当**CreateProcess**在内部打开这些对象时，每个对象的使用计数就变为2。

这意味着系统要想释放进程对象，进程必须终止（使用计数递减1），而且父进程必须调用**CloseHandle**（使用计数再次递减1，变成0）。类似地，要想释放线程对象，线程必须终止，而且父进程必须关闭到线程对象的句柄（要想进一步了解如何释放线程对象，请参见4.3节“子进程”）。

> 注意
> 应用程序运行期间，必须关闭到子进程及其主线程的句柄，以避免资源泄漏。当然，系统会在你的进程终止后自动清理这种泄漏。但是，如果是一个编写精妙的软件，肯定会在进程不再需要访问一个子进程及其主线程的时候，显式地调用CloseHandle来关闭这些句柄。忘记关闭这些句柄是开发人员最容易犯的错误之一。
>
> 不知道为什么，许多开发人员都有这样的一个误解：关闭到一个进程或线程的句柄，会强迫系统杀死此进程或线程。但这是大谬不然的。关闭句柄只是告诉系统你对进程或线程的统计数据不再感兴趣了。进程或线程会继续执行，直至自行终止。

创建一个进程内核对象时，系统会为此对象分配一个独一无二的标识符，系统中没有别的进程内核对象会有相同的ID编号。这同样适用于线程内核对象。创建一个线程内核对象时，此对象会被分配一个独一无二的、系统级别的ID编号。进程ID和线程ID分享同一个号码池。这意味着线程和进程不可能有相同的ID。此外，一个对象决不会被分配到0作为其ID。注意，Windows任务管理器将进程ID 0与“System Idle Process”（系统空闲进程）关联，如下所示。但是，实

际上并没有System Idle Process这样的东西。任务管理器将这个虚构的进程作为Idle线程的占位符来创建；在没有别的正在运行时，就会运行这个空闲进程。System Idle Process中的线程数量始终等于计算机的CPU的数量。所以，它始终代表未被真实进程使用的CPU usage的百分比。



CreateProcess返回之前，它会将这些ID填充到**PROCESS_INFORMATION**结构的**dwProcessId**和**dwThreadId**成员中。ID使我们很容易识别系统中的进程和线程。ID主要由实用程序（比如任务管理器）使用，很少由生产应用程序使用。因此，大多数应用程序都会忽略ID。

如果你的应用程序要使用ID来跟踪进程和线程，那么必须注意这一点：进程和线程ID会被系统立即重用。例如，假定在创建一个进程之后，系统初始化了一个进程对象，并将ID值124分配给它。如果再创建一个新的进程对象，系统不会将同一个ID编号分配给它。但是，如果第一个进程对象已经释放，系统就可以将124分配给下一个创建的进程对象。请务必牢记这个特点，避免自己的代码引用不正确的进程或线程对象。进程ID很容易获得，也很容易保存。但就像前面所说的那样，你刚刚保存好一个ID，与它对应的进程就可能被释放了。所以，系统在创建下一个新进程的时候，会将这个ID分配给它。一旦你使用保存的进程ID，操纵的就是新进程，而不是原先那个进程。

可以使用**GetCurrentProcessId**来得到当前进程的ID，使用**GetCurrentThreadId**来获得当前正在运行的线程的ID。另外，还可以使用**GetProcessId**来获得与指定句柄对应的一个进程的ID，使用**GetThreadId**来获得与指定句柄对应的一个线程的ID。最后，根据一个线程句柄，你可以调用**GetProcessIdOfThread**来获得其归属进程的ID。

个别情况下，你的应用程序可能想确定它的父进程。但是，你首先应该知道的是，只有在一个子进程生成的那一瞬间，才存在一个父-子关系。到子进程开始执行代码之前的那一刻，Windows就已经不认为存在任何父-子关系了。ToolHelp函数允许进程通过**PROCESSENTRY32**结构查询其父进程。在此结构内部有一个**th32ParentProcessID**成员；根据文档，它能返回父进程的ID。

系统确实会记住每个进程的父进程的ID，但由于ID会被立即重用，所以等你获得父进程的ID的时候，那个ID标识的可能已经是系统中运行的一个完全不同的进程。你的父进程也许已经终止了。如果你的应用程序需要与它的“创建者”通信，最好不要使用ID。应该定义一个更持久的通信机制，比如内核对象、窗口句柄等等。

要保证一个进程或线程ID不被重用，惟一的办法就是保证进程或线程对象不被销毁。为此，在创建了一个新进程或线程之后，不关闭到这些对象的句柄即可。等到应用程序不再使用ID的时候，再调用**CloseHandle**来释放内核对象。但是，一旦调用了CloseHandle，再使用或依赖进程ID就不安全了。这一点务必牢记。对于子进程，不用做任何事情，就能保证父进程或线程ID的有效性，除非父进程为它自己的进程或线程对象复制了句柄，并允许子进程继承这些句柄（参见3.3.5节）。

4.2 终止进程

进程可以通过以下4种方式终止：
- 主线程的入口函数返回（强烈推荐的方式）。
- 进程中的一个线程调用**ExitProcess**函数（要避免这个方式）
- 另一个进程中的线程调用**TerminateProcess**函数（要避免这个方式）
- 进程中的所有线程都“自然死亡”（这是很难发生的）

本节将讨论所有这4种方法，并描述进程终止时实际发生的情况。

4.2.1 主线程的入口函数返回

设计一个应用程序时，应该保证只有在主线程的入口函数返回之后，这个应用程序的进程才终止。只有这样，才能保证主线程的所有资源都被正确清理。

让主线程的入口函数返回，可以保证发生以下几件事情：
- 该线程创建的任何C++对象都将由这些对象的析构函数正确销毁。
- 操作系统将正确释放线程堆栈使用的内存。
- 系统将进程的退出代码（在进程内核对象中维护）设为你的入口函数的返回值。
- 系统递减进程内核对象的使用计数。

4.2.2 ExitProcess 函数

进程会在该进程中的一个线程调用**ExitProcess**函数时终止：

VOID ExitProcess(UINT fuExitCode);

该函数将终止进程，并将进程的退出代码设为**fuExitCode**。**ExitProcess**不会返回值，因为进程已经终止了。如果**ExitProcess**之后还有别的代码，那些代码永远不会执行。

当主线程的入口函数（**WinMain**，**wWinMain**，**main**或**wmain**）返回时，会返回到C/C++运行时启动代码，后者将正确清理进程使用的全部C运行时资源。释放了C运行时资源之后，C运行时启动代码将显式调用**ExitProcess**，向它传递从入口函数返回的值。这便解释了为什么只需从主线程的入口函数返回，就会终止整个进程。注意，进程中运行的其他任何线程都会随进程一起终止。

Windows Platform SDK文档指出，一个进程在其所有线程都终止之后才会终止。从操作系统的角度出发，这种说法是正确的。不过，C/C++运行时为应用程序采取了一个不同的策略。C/C++运行时启动代码确保了下面这一点：一旦你的应用程序的主线程从它的入口函数返回，那么不管当前在进程中是否正在运行其他线程，都会调用ExitProcess来终止进程。不过，如果在入口函数中调用**ExitThread**，而不是调用**ExitProcess**或者简单地返回，应用程序的主线程将停止执行，但只要进程中还有其他线程正在运行，进程就不会终止。

注意，调用**ExitProcess**或**ExitThread**会导致进程或线程直接“死亡”——即使当前正在一个函

数中。就操作系统而言，这样做是没有什么问题的，进程或线程的所有操作系统资源都会被正确清理。不过，C/C++应用程序应避免调用这些函数，因为C/C++运行时也许不能正确清理。来看看以下代码：

```
#include <windows.h>
#include <stdio.h>

class CSomeObj {
public:
   CSomeObj() { printf("Constructor \r\n"); }
   ~CSomeObj() { printf("Destructor \r\n"); }
};

CSomeObj g_GlobalObj;

void main () {
   CSomeObj LocalObj;
   ExitProcess(0); // 不应该放在这里

   // 在这个函数的末尾，编译器本来会自动添加
   // 必要的代码来调用LocalObj的析构函数。
   // 但ExitProcess阻止了这个析构函数的执行。
}
```

执行上述代码，会显示：

```
Constructor
Constructor
```

它构造了两个对象，一个是全局对象，另一个是本地对象。“Destructor”字样永远都不会显示。C++对象没有被正确析构，因为**ExitProcess**造成进程“当场死亡”：C/C++运行时没有机会执行清理工作。

就像我说的那样，任何时候都不要显式地调用**ExitProcess**。在前面的代码中删除对**ExitProcess**函数的调用，再运行这个程序就会得到以下结果：

```
Constructor
Constructor
Destructor
Destructor
```

只需从主线程的入口函数返回，C/C++运行时就能执行其清理工作，并正确析构所有C++对象。顺便提一下，这里的讨论并不只适用于C++对象。C/C++运行时代表你的进程做了许多事情；最好允许运行时正确地完成清理工作。

注意
许多应用程序无法正确清理它自己，都是因为显式调用了ExitProcess和ExitThread的原因。在ExitThread的情况下，进程会继续运行，但可能泄漏内存或其他资源。

4.2.3 TerminateProcess 函数

调用**TerminateProcess**也可以终止一个进程，如下所示：

```
BOOL TerminateProcess(
    HANDLE hProcess,
```

UINT fuExitCode);

此函数与**ExitProcess**函数有一个明显的区别：任何线程都可以调用**TerminateProcess**来终止另一个进程或者它自己的进程。**hProcess**参数指定了要终止的进程的句柄。进程终止时，其退出代码将成为作为**fuExitCode**参数来传递的值。

只有在无法通过其他方法来强制进程退出时，才应使用**TerminateProcess**。被终止的进程得不到自己要被终止的通知——应用程序不能正确清理，也不能阻止它自己被“杀死”（除非通过正常的安全机制）。例如，在这种情况下，进程无法将它在内存中的任何信息转储到磁盘上。

虽然进程没有机会执行自己的清理工作，但操作系统会在进程终止之后彻底进行清理，确保不会泄漏任何操作系统资源。这意味着进程使用的所有内存都会被施放，所有打开的文件都会关闭，所有内核对象的使用计数都将递减，所有的User和GDI对象都会被销毁。

系统保证当一个进程终止之后（不管是如何终止的），它的任何部分都不会遗留下来。绝对没有任何办法知道那个进程是否运行过。**进程在终止后绝对不会泄漏任何东西**。希望大家都已经明确这一点了。

注意
TerminateProcess函数是异步的——换言之，它告诉系统你希望进程终止，但到函数返回的时候，并不能保证进程已经被“杀死”了。所以，为了确定进程是否已经终止，应该调用WaitForSingleObject（详见第9章）或者一个类似的函数，并将进程的句柄传给它。

4.2.4 当进程中的所有线程死亡时

如果一个进程中的所有线程都死亡了（要么是因为它们都调用了**ExitThread**，要么是因为它们都用**TerminateThread**来终止了），操作系统就认为没有任何理由再保持进程的地址空间。这是非常合理的，因为没有线程在执行地址空间中的任何代码。一旦系统检测到一个进程中没有任何线程在运行，就会终止这个进程。进程的退出代码会设为最后一个“死亡”的线程的退出代码。

4.2.5 当进程终止运行时

一个进程终止时，会依次采取以下操作：

1. 进程中遗留的任何线程都被终止。
2. 进程分配的所有User和GDI对象都被释放，所有内核对象都被关闭（如果没有其他进程打开了到它们的句柄，这些内核对象也会被销毁。不过，如果其他进程打开了到它们的句柄，这些内核对象就不会被销毁）。
3. 进程的退出代码从STILL_ACTIVE变为传给ExitProcess或TerminateProcess函数的代码。
4. 进程内核对象的状态变成signaled（关于信号机制的详情，请参见第9章）。这就是为什么系统中的其他线程可以挂起它们自己，直至进程终止运行。
5. 进程内核对象的使用计数递减1。

注意，进程内核对象的生命期至少能像进程本身一样长。但是，进程内核对象也许比它的进程“活”得更久。一个进程终止时，系统会自动递减其内核对象的使用计数。如果计数减至0，表明没有其他进程打开了这个对象的句柄，所以在进程被销毁时，对象也会被销毁。

但是，如果系统中还有另一个进程打开了正在“死亡”的这个进程的内核对象的句柄，进程内核对象的使用计数就不会变成0。当父进程忘记关闭到它的一个子进程的句柄时，往往会发生这种情况。这是一个特性（或功能），而不是bug。记住，进程内核对象会维护与进程有关的统计信息。即使是在进程终止之后，这些信息也可能有用。例如，你可能想知道一个进程需要多少CPU时间。或者，一个更有可能的原因是，你想通过调用**GetExitCodeProcess**来获得已经死亡的一个进程的退出代码：

```
BOOL GetExitCodeProcess(
    HANDLE hProcess,
    PDWORD pdwExitCode);
```

该函数会查找进程内核对象（由**hProcess**参数标识），并从内核对象的数据结构中提取标识了进程退出代码的成员。退出代码的值在**pdwExitCode**参数指向的一个**DWORD**中返回。

任何时候都可以调用这个函数。如果在调用**GetExitCodeProcess**的时候进程还没有终止，函数将用**STILL_ACTIVE**标识符（定义为0x103）来填充**DWORD**。如果进程已经终止，就返回实际的退出代码值。

有人可能会想，我是不是可以编写代码，定期调用**GetExitCodeProcess**并检查退出代码，从而判断一个进程是否终止？这在很多情况下都是行得通的，但效率也是令人不敢恭维的。下一节将介绍如何通过正确的方式来判断进程是在什么时候终止的。

再次重申，应该调用**CloseHandle**来告诉操作系统你已经对进程中的统计数据不感兴趣了。如果进程已经终止，**CloseHandle**函数将递减内核对象的使用计数，并释放它。

4.3 子进程

设计一个应用程序时，可能会遇到想用另一个代码块来执行工作的情况。为此，我们总是调用函数或子程序来分配这样的工作。如果是调用一个函数，那么除非函数返回，否则你的代码不能继续处理。在很多情况下，都需要这种单任务同步机制。为了让另一个代码块来执行工作，另一个方法是在进程内新建一个线程，让它帮助你进行处理。这样一来，当另一个线程执行你指定的工作时，你的代码可以继续处理。这是很有用的一种技术，但在你的线程需要查看新线程的结果时，就会遇到同步问题。

另一个办法是生成一个新的进程——称为子进程（child process）——来帮助你处理工作。例如，假定你现在要做的工作非常复杂。为了完成工作，你在同一个进程中创建了一个新的线程。你写了一些代码，进行了测试，但得到了不正确的结果。究其原因，也许是算法有误，也许是不正确地提领了某个数据，不慎改写了地址空间中的重要数据。为了在处理工作期间保护地址空间，一个办法是让一个新进程来执行工作。然后，既可以在新进程终止之后，才继续你自己的工作，也可以在新进程工作期间继续自己的工作。

遗憾的是，新进程可能需要操作你的地址空间中的数据。在这种情况下，最好让进程在它自己的地址空间中运行，并只允许它访问父进程地址空间中与它的工作有关的数据，从而保护与正

在进行的处理无关的其他数据。Windows提供了几种方式在不同进程之间传递数据，其中包括动态数据交换（Dynamic Data Exchange，DDE）， OLE，pipes，mailslots等等。共享数据最方便的方式之一就是使用内存映射文件（有关这方面的详细讨论，请参见第17章）。

如果你希望创建一个新线程，让它执行一些工作，然后等候结果，可以像下面这样编码：

```
PROCESS_INFORMATION pi;
DWORD dwExitCode;

// Spawn the child process.
BOOL fSuccess = CreateProcess(..., &pi);
if (fSuccess) {

    // Close the thread handle as soon as it is no longer needed!
    CloseHandle(pi.hThread);

    // Suspend our execution until the child has terminated.
    WaitForSingleObject(pi.hProcess, INFINITE);

    // The child process terminated; get its exit code.
    GetExitCodeProcess(pi.hProcess, &dwExitCode);

    // Close the process handle as soon as it is no longer needed.
    CloseHandle(pi.hProcess);
}
```

在上述代码段中，你创建了新进程；如果创建成功，就调用**WaitForSingleObject**函数：

```
DWORD WaitForSingleObject(HANDLE hObject, DWORD dwTimeout);
```

我们将在第9章全面讨论**WaitForSingleObject**函数。现在，你只需要知道此函数会一直等待，直至**hObject**参数所标识的对象变为signaled。进程对象在终止的时候，就会变为signaled。所以，对**WaitForSingleObject**函数的调用将暂停执行父进程的线程，直至子进程终止。**WaitForSingleObject**返回后，可以调用**GetExitCodeProcess**来获得子进程的退出代码。

在前面的代码段中，对**CloseHandle**函数的调用导致系统将线程和进程对象的使用计数递减至0，使对象占用的内存可以被释放。

注意，在上述代码中，我们在**CreateProcess**返回之后，立即关闭了到子进程的主线程内核对象的句柄。这不会导致子进程的主线程终止，它只是递减了子进程的主线程内核对象的使用计数。下面解释了为什么说这是一个良好的编程习惯：假定子进程的主线程生成另一个线程，然后主线程终止。此时，系统就可以从内存中释放子进程的主线程对象——前提是父进程没有打开到这个线程对象的句柄。但是，假如父进程打开了到子进程的主线程对象的一个句柄，系统就不会释放这个对象，除非父进程关闭这个句柄。

运行分离的子进程

大多数时候，应用程序以一个“分离的进程”（detached process）的形式来启动另一个进程。这意味着一旦进程创建并开始执行，父进程就不再与新进程通信，或者不用等它完成工作之后才继续自己的工作。Windows资源管理器就是这样工作的。当Windows资源管理器为用户创建了一个新的进程之后，就不再关心这个进程是否继续存在，也不关心用户是否要终止它。

为了放弃与子进程的所有联系，Windows资源管理器必须调用**CloseHandle**，关闭到新的进程及其主线程的句柄。以下代码示例展示了如何新建一个进程，然后让它分离出去运行：

```
PROCESS_INFORMATION pi;

// Spawn the child process.
BOOL fSuccess = CreateProcess(..., &pi);
if (fSuccess) {
    // Allow the system to destroy the process & thread kernel
    // objects as soon as the child process terminates.
    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);
}
```

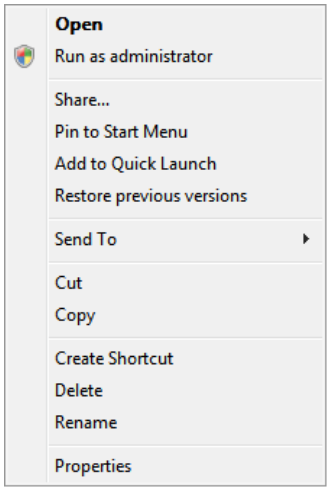
4.4 当管理员以标准用户的身份运行时

感谢一系列新技术，Windows Vista为最终用户提高了安全水准。对于应用程序开发人员，影响最大的技术当属“用户帐户控制”（User Account Control，UAC）。

Microsoft注意到这样一个事实：大多数用户都用一个Administrator（管理员）帐户来登录Windows。利用这个帐户，用户能几乎没有任何限制地访问重要的系统资源，因为该帐户被授予了很高的权限。一旦用户用这样的一个特权帐户来登录Vista之前的某个Windows操作系统，就会创建一个安全令牌（security token）。每当有代码试图一个受保护的安全资源时时，操作系统就会使用（出示）这个安全令牌。这个令牌会与新建的所有进程关联。第一个进程就是Windows资源管理器，后者随即将令牌拿给它的所有子进程，并以此类推。在这样的配置中，如果从Internet下载的一个恶意程序开始运行，或者电子邮件中的一个恶意脚本开始运行，就会继承Administrator帐户的高特权（因为它们的宿主应用程序正在这个帐户下运行）——因而更改机器上的任何内容，甚至可以启动另一个进程，并让启动的进程继承相同的高特权。

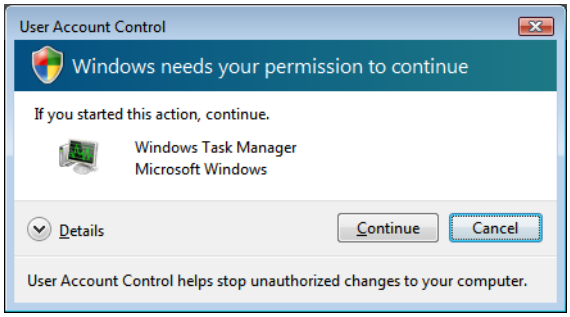
相反，在Windows Vista中，如果用户使用Administrator这样的一个被授予高特权的帐户登录，那么除了与这个帐户对应的安全令牌之外，还会创建一个经过筛选的令牌（filtered token），后者将只被授予Standard User（标准用户）的权限。以后，系统代表最终用户启动的所有新进程都会关联这个筛选令牌。第一个进程仍然是Wndows资源管理器。你可能马上会对此提出疑问：既然所有应用程序都只有标准用户的权限集，那么如何访问受限制的资源呢？比较短的一个回答是：权限受限的进程无法访问需要更高权限才能访问的安全资源。接着，让我们来给出一个较长的回答。另外，在本节剩余的部分，将集中讨论开发者如何利用UAC。

首先，你可以要求操作系统提升权限，但只能在进程边界上提升。这是什么意思呢？默认情况下，一个进程启动时，它会与当前登录用户的筛选令牌关联起来。要为进程授予更多的权限，你（开发人员）需要指示Windows做这样一件事情：在进程启动**之前**，友好地征求最终用户（对于提升权限）的同意。作为最终用户，可以使用快捷菜单中的“以管理员身份运行”命令（在Windows资源管理器中右击一个应用程序，即可打开这个菜单）。

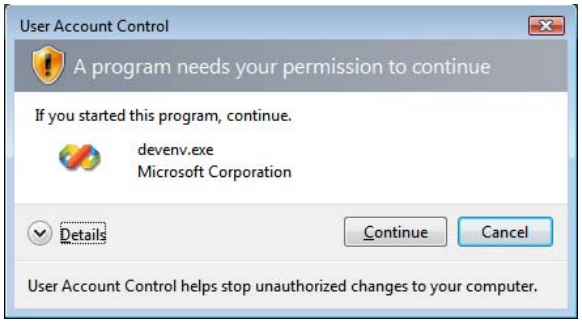


如果本身就是以管理员身份登录的，那么一旦选择“以管理员身份运行”，就会在一个“安全Windows桌面”中显示一个确认对话框，要求你批准将权限提升到未筛选的安全令牌的级别。选择“以管理员身份运行”之后，最终用户可能看到三种类型的对话框。下面分别进行了解释。

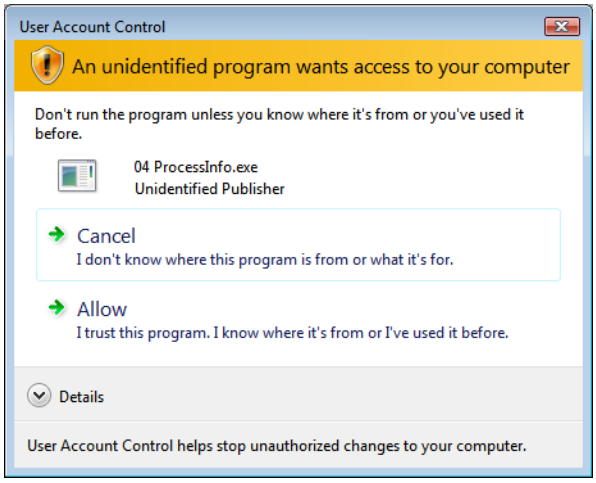
如果应用程序是系统的一部分，就会显示以下安全确认对话框，上面是一个蓝色的横幅：



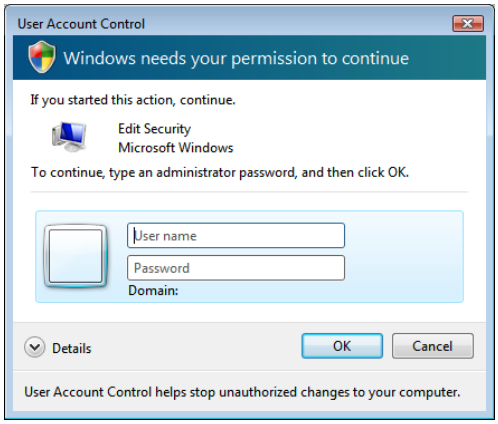
如果应用程序进行了签名，对话框中将显示一个较没有自信的灰色横幅：



最后，如果应用程序没有签名，会在对话框中显示一个橙色的横幅：



注意，如果当前以一个标准用户的身份登录，会弹出另一个对话框，要求你提供一个提升了权限的帐户的登录凭据。采用这个设计，管理员就可以帮助登录到这台计算机的标准用户执行需要更高权限才能执行的任务。这称为over-the-shoulder登录（参见右图）。

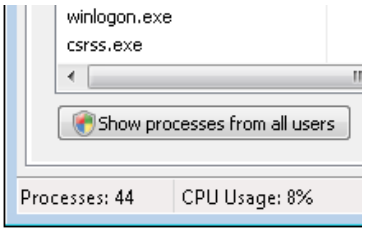


注意

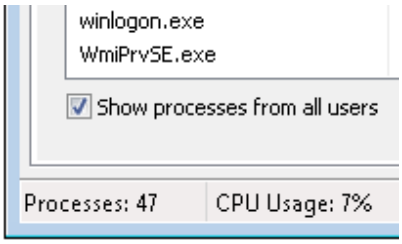
很多人会问，为什么Windows Vista不是只问一次，然后把安全数据保存到系统中，并让用户随意以管理员的身份来运行应用程序呢？这样一来，就不必每次都弹出一个UAC对话框来打扰用户了。Windows Vista不提供这个功能，因为否则的话，它就必须将数据保存到某个地方（比如保存到注册表中或者其他某个文件中）。一旦这个存储区域被成功入侵，恶意程序就可以修改这个存储，使它总是以提升的权限来运行，而不会向用户进行任何提示。

除了Windows资源管理器快捷菜单中的“以管理员身份运行”命令，你肯定还注意到了一个新的盾牌图标。这个图标会出现在负责执行Windows Vista管理任务的一些链接旁边或者按钮上面。这个新的界面元素向用户清楚地表明：一旦点击这个链接或按钮，就会弹出一个权限提升确认对话框。本章最后提供的Process Information示例代码展示了如何采取一些简单操作，就可以在应用程序的一个按钮上显示这个盾牌图标。

来看一个简单的例子。右击任务栏，从弹出菜单中选择“任务管理器”。在“任务管理器”的“进程”卡片底部，可以清楚地看到“显示所有用户的进程”按钮上的盾牌图标。



单击这个按钮之前，先看看任务管理器的进程ID（也就是taskmgr.exe的PID）。在确认了权限提升之后，任务管理器将短暂地消失，但随后它会重新出现，不过一个复选框代替了盾牌按钮。



再检查一下任务管理器的PID，你会注意到它与提升权限前的PID不一样。这是否意味着任务管理器必须生成它自己的另一个实例，才能获得权限提升呢？是的，答案是肯定的。如前所述，Windows只允许在进程边界上进行权限提升。一旦进程启动，再要求更多的权限就已经迟了。不过，一个未提升权限的进程可以生成另一个提升了权限的进程，后者将包含一个COM服务器。这个新进程将保持活动状态。这样一来，老进程就可以向已经提升了权限的新进程发出IPC调用，而不必启动一个新实例再终止它自身。

注意
在一段视频和一份Microsoft Office PowerPoint演示文稿中（网址为*http://www.microsoft.com/emea/spotlight/sessionh.aspx?videoid=360*），Mark Russinovich详细介绍了UAC的内部机制，比如Windows的系统资源虚拟化技术，它使那些在设计时没有考虑到新的Administator权限限制的应用程序能够更好地兼容新的操作系统。

4.4.1 自动提升进程的权限

如果你的应用程序一直都需要管理员权限，比如在它是一个安装程序的前提下，那么每次调用（invoke）应用程序时，操作系统就可以自动提示用户提升权限。每当一个新进程生成时，Windows的UAC组件如何判断应该采取什么操作呢？

假如应用程序的执行体已经中嵌入了一种特定的资源（**RT_MANIFEST**），系统就会检查**<trustInfo>**区域，并解析其内容。下面摘录了一个示例manifest（清单）文件中的<trustInfo>区域：

```
...
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
        <requestedPrivileges>
            <requestedExecutionLevel
                level="requireAdministrator"
```

```
            />
        </requestedPrivileges>
    </security>
</trustInfo>
...
```

level属性可能有3个不同的值，如表4-8所示。

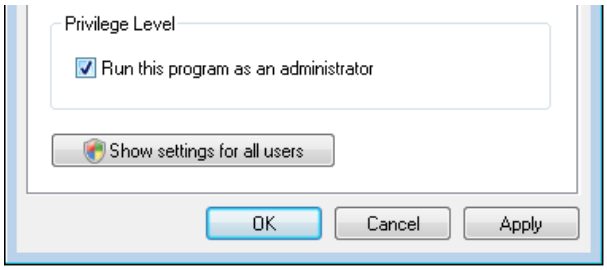
表4-8　level属性的值

值	描述
requireAdministrator	应用程序必须以管理员权限启动；否则不会运行
highestAvailable	应用程序以当前可用的最高权限运行。 如果用户使用一个管理员帐户登录，会出现要求批准提升权限的一个对话框。 如果用户使用一个普通用户帐户登录，应用程序就用这些标准权限来启动（不会提示用户提升权限）
asInvoker	应用程序使用与主调应用程序一样的权限来启动

也可以选择不将manifest嵌入执行体的资源中。相反，可以将manifest保存到与执行体文件相同的目录中，名称和执行体文件相同，但扩展名使用.manifest。

但是，外部manifest文件也许不会立即被操作系统发现，尤其假如你在manifest文件就位之前就已经启动了执行体。在这种情况下，你需要先注销再重新登录，才能使Windows注意到外部的manifest文件。在任何情况下，只要执行体文件中嵌入了一个manifest，外部的manifest文件就会被忽略。

除了通过XML manifest来显式地设置，操作系统还会根据一系列特定的兼容性规则来“智能”地判断一个程序是不是安装程序。如果是安装程序，就会自动显示一个提升权限对话框。对于其他应用程序，假如没有发现任何manifest，也没有发现它具有安装程序的行为，那么最终用户可以自行决定是否以管理员身份启动进程。具体的办法是在执行体文件的属性对话框的“兼容性”卡片中勾选对应的复选框，如下图所示。



注意
与应用程序兼容性相关的主题不在本书讨论范围内，但可以访问以下网址来了解详情，这是有关如何在Windows Vista中开发相容于UAC的应用程序的一份白皮书：
http://www.microsoft.com/downloads/details.aspx?FamilyID=ba73b169-a648-49af-bc5e-a2eebb74c16b&DisplayLang=en

4.4.2 手动提升进程的权限

如果你仔细阅读过本章前面对**CreateProcess**函数的描述，肯定已经注意到它没有专门提供什么标记或参数来指定对这种权限提升的需求。相反，你需要调用的是**ShellExecuteEx**函数：

```
BOOL ShellExecuteEx(LPSHELLEXECUTEINFO pExecInfo);
typedef struct _SHELLEXECUTEINFO {
  DWORD cbSize;
  ULONG fMask;
  HWND hwnd;
  PCTSTR lpVerb;
  PCTSTR lpFile;
  PCTSTR lpParameters;
  PCTSTR lpDirectory;
  int nShow;
  HINSTANCE hInstApp;
  PVOID lpIDList;
  PCTSTR lpClass;
  HKEY hkeyClass;
  DWORD dwHotKey;
  union {
    HANDLE hIcon;
    HANDLE hMonitor;
  } DUMMYUNIONNAME;
  HANDLE hProcess;
} SHELLEXECUTEINFO, *LPSHELLEXECUTEINFO;
```

SHELLEXECUTEINFO结构中惟一有趣的字段是**lpVerb**和**lpFile**。前者必须被设为“*runas*”，后者必须包含使用提升后的权限来启动的一个执行体文件的路径，如以下代码段所示：

```
// Initialize the structure.
SHELLEXECUTEINFO sei = { sizeof(SHELLEXECUTEINFO) };
// Ask for privileges elevation.
sei.lpVerb = TEXT("runas");
// Create a Command Prompt from which you will be able to start
// other elevated applications.
sei.lpFile = TEXT("cmd.exe");
// Don't forget this parameter; otherwise, the window will be hidden.
sei.nShow = SW_SHOWNORMAL;
if (!ShellExecuteEx(&sei)) {
  DWORD dwStatus = GetLastError();
  if (dwStatus == ERROR_CANCELLED) {
  // The user refused to allow privileges elevation.
  }
  else
  if (dwStatus == ERROR_FILE_NOT_FOUND) {
  // The file defined by lpFile was not found and
  // an error message popped up.
  }
}
```

如果用户拒绝提升权限，**ShellExecuteEx**将返回**FALSE**，**GetLastError**通过使用一个**ERROR_CANCELLED**值来指出这个情况。

注意，当一个进程使用提升后的权限启动时，它每次用**CreateProcess**来生成另一个进程时，子进程都会获得和它的父进程一样的提升后的权限，在这种情况下，不需要调用**ShellExecuteEx**。假如一个应用程序是用一个筛选后的令牌来运行的，那么一旦试图调用**CreateProcess**来生成一个要求提升权限的执行体，这个调用就会失败，**GetLastError**会返回**ERROR_ELEVATION_REQUIRED**。

总之，要想成为Windows Vista中的“好公民”，你的应用程序大多数时候都应该以“标准用户”的身份运行。另外，在它要求更多权限的时候，用户界面应该在与这个管理任务对应的用户界

面元素（按钮、链接或菜单项）旁边明确显示一个盾牌图标（本章稍后的Process Information例子给出了一个例子）。由于管理任务必须由另一个进程或者另一个进程中的COM服务器来执行，所以你应该在另一个应用程序中收集好需要管理员权限的所有任务，并通过调用**ShellExecuteEx**（为lpVerb 传递“runas”）来提升它的权限。然后，具体要执行的特权操作应该作为新进程的命令行上的一个参数来传递。这个参数是通过SHELLEXECUTEINFO的lpParameters字段来传递的。

提示
对权限提升/筛选的进程进行调试可能比较麻烦。但你可以遵循一条非常简单的黄金法则：希望被调试的进程继承什么权限，就以那种权限来启动Visual Studio。

如果需要调试的是一个以标准用户身份运行的已筛选的进程，就必须以标准用户的身份来启动Visual Studio。每次单击它的默认快捷方式（或者通过「开始」菜单来启动）时，都是以标准用户的身份来启动它的。否则，被调试的进程会从以管理员身份启动的一个Visual Studio实例中继承提升后的权限，这并不是你所期望的。

如果需要调试的是一个以管理员身份运行的进程（例如，根据那个进程的manifest中的描述，它可能必须以管理员身份运行），那么Visual Studio必须同样以管理员身份启动。否则会显示一条错误消息，指出“the requested operation requires elevation”（请求的操作需要提升权限），而且被调试的进程根本不会启动。

4.4.3 何为当前权限上下文

前面描述过任务管理器的例子，它在“进程”卡片的底部要么显示一个盾牌图标，要么显示一个复选框，具体取决于它是如何生成的。稍加思索，你就应该想到两个问题：如何判断应用程序是否是以管理员身份运行；更重要的是，如何判断它是以提升的权限来启动的，还是正在使用筛选的令牌运行。

下面这个名为**GetProcessElevation**的helper函数能返回提升类型和一个指出你是否正在以管理员身份运行的布尔值。

```
BOOL GetProcessElevation(TOKEN_ELEVATION_TYPE* pElevationType, BOOL* pIsAdmin) {
  HANDLE hToken = NULL;
  DWORD dwSize;

  // Get current process token
  if (!OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken))
    return(FALSE);

  BOOL bResult = FALSE;

  // Retrieve elevation type information
  if (GetTokenInformation(hToken, TokenElevationType,
    pElevationType, sizeof(TOKEN_ELEVATION_TYPE), &dwSize)) {
    // Create the SID corresponding to the Administrators group
    BYTE adminSID[SECURITY_MAX_SID_SIZE];
    dwSize = sizeof(adminSID);
    CreateWellKnownSid(WinBuiltinAdministratorsSid, NULL, &adminSID,
         &dwSize);

    if (*pElevationType == TokenElevationTypeLimited) {
      // Get handle to linked token (will have one if we are lua)
      HANDLE hUnfilteredToken = NULL;
      GetTokenInformation(hToken, TokenLinkedToken, (VOID*)
           &hUnfilteredToken, sizeof(HANDLE), &dwSize);
```

```
      // Check if this original token contains admin SID
      if (CheckTokenMembership(hUnfilteredToken, &adminSID, pIsAdmin)) {
           bResult = TRUE;
      }

      // Don't forget to close the unfiltered token
      CloseHandle(hUnfilteredToken);
      } else {
         *pIsAdmin = IsUserAnAdmin();
         bResult = TRUE;
      }
    }
    // Don't forget to close the process token

    CloseHandle(hToken);
    return(bResult);
  }
```

注意，GetTokenInformation使用与进程关联的安全令牌和TokenElevationType参数来获得提升类型，提升类型的值由TOKEN_ELEVATION_TYPE枚举类型来定义，如表4-9所示。

表4-9　TOKEN_ELEVATION_TYPE的值

值	描述
TokenElevationTypeDefault	进程以默认用户运行，或者UAC被禁用
TokenElevationTypeFull	进程的权限被成功提升，而且令牌没有被筛选过
TokenElevationTypeLimited	进程使用和一个筛选过的令牌对应的受限的权限运行

根据这些值，可以知道你是否正在使用一个筛选的令牌运行。下一步是判断用户是不是管理员。如果令牌没有被筛选过，那么为了知道你是否正在以管理员的身份运行，**IsUserAnAdmin**就是最理想的函数。在令牌已被筛选的情况下，你需要获取未筛选的令牌（把**TokenLinkedToken**传给**GetTokenInformation**），然后判断其中包含一个管理员SID（借助于**CreateWellKnownSid**和**CheckTokenMembership**）。

举个例子来说，下一小节将详细介绍的Process Information示例程序就在**WM_INITDIALOG**消息处理代码中使用了这个helper函数，目的是为标题附加提升细节前缀，并显示或隐藏一个盾牌图标。

> 提示
> 注意，Button_SetElevationRequiredState宏（在CommCtrl.h中定义）用于显示或隐藏按钮中的盾牌图标。也可以调用SHGetStockIconInfo，并将SIID_SHIELD 作为参数，从而直接将盾牌作为图标来获取；两者都在shellapi.h中定义。要了解其他还有哪些类型的控件支持盾牌隐喻，请查阅MSDN联机帮助，网址为*http://msdn2.microsoft.com/en-us/library/aa480150.aspx*。

4.4.4　枚举系统中正在运行的进程

许多软件开发人员在为Windows编写工具或实用程序时，都曾遇到过需要枚举正在运行的进程的情况。Windows API最初并没有函数能枚举正在运行的进程。不过，Windows NT有一个不断更新的数据库，称为“性能数据”（Performance Data）数据库。该数据库包含海量信息，并可以通过注册表函数来访问，例如RegQueryValueEx（把根项设为KEY_PERFORMANCE_DATA）。由于以下原因，几乎很少有Windows程序员知道这个性能数据库：

- 它没有自己专门的函数，使用的是现成的注册表函数
- 在Windows 95和Windows 98上不可用
- 数据库中的信息布局非常复杂，许多开发人员都怕用它，这妨碍了它的流行

为了更方便地使用这个数据库，Microsoft创建了一套Performance Data Helper函数（包含在PDH.dll中）。要想进一步了解这个库，请在Platform SDK文档中搜索Performance Data Helper。

正如我提到的那样，Windows 95和98没有提供这个性能数据库。它们有自己的一套函数可用来枚举进程及相关信息。这些函数在ToolHelp API中。欲知详情，请在Platform SDK文档中搜索**Process32First**和**Process32Next**函数。

有趣的是，Windows NT开发团队不喜欢ToolHelp函数，所以没有把它们加入Windows NT。相反，他们自行开发了Process Status函数（包含在PSAPI.dll中）来枚举进程。欲知详情，请在Platform SDK文档中搜索**EnumProcesses**函数。

看起来，Microsoft也许是想为工具和实用程序的开发人员增加难度。不过，从Windows 2000开始，他们终于在Windows操作系统中添加了ToolHelp函数。现在，开发人员终于可以写出适用性更强的工具和实用程序了，让它们在Windows 95、Windows 98……直至Windows Vista中都能使用一套通用的源代码。

4.4.5 Process Information 示例程序

ProcessInfo应用程序（04-ProcessInfo.exe）展示了如何利用各种ToolHelp函数来写一个非常有用的实用程序。这个应用程序的源代码和资源文件可以在04-ProcessInfo目录中找到。启动该程序后，将显示图4-4所示的窗口。

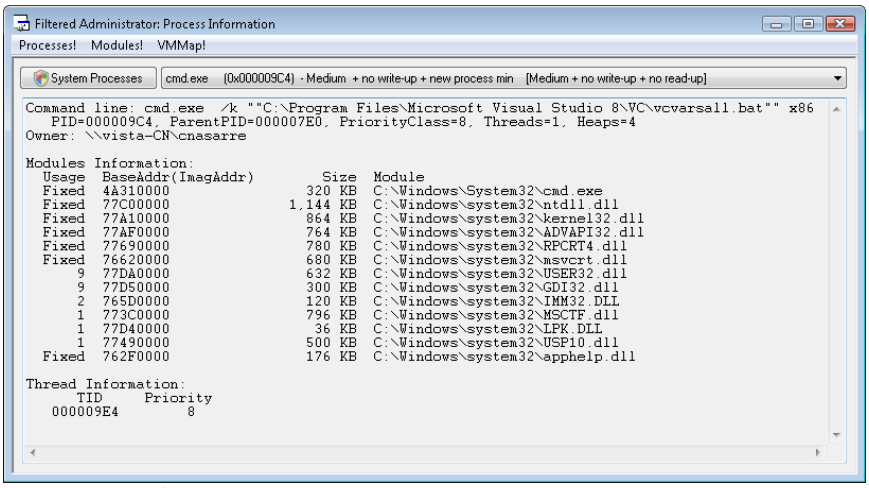


图4-4　运行中的 ProcessInfo

ProcessInfo首先枚举目前正在运行的所有进程，将每个进程的名称和ID放在顶部的组合框内。然后，第一个进程被选中，在下方的只读编辑框中显示与这个进程有关的信息。如你所见，程序显示了进程ID、命令行、所有者、父进程ID、进程的优先级类（PriorityClass）以及在此进程的上下文中运行的线程数。大多数信息都超出了本章的范围，但会在后续的各章中陆续讨论。

查看进程列表时，可以使用VMMap菜单（查看模块信息时，这个菜单是禁用的）。选择VMMap菜单，会运行VMMap示例程序（将在第14章讨论）。这个程序将遍历所选进程的地址空间。

在模块信息（Module Information）区域，列出了映射到进程地址空间的模块（执行体和DLL）。

所谓Fixed（固定）模块，是指进程初始化时隐式载入的模块。对于显式载入的DLL，会显示其使用计数。第二列显示的是模块映射到的内存地址。如果模块没有映射到它首选的基地址，则首选基地址会在圆括号中显示。第三列显示模块大小（以KB为单位）。最后一列显示的是模块的完整路径名。在线程信息（Thread Information）区域，显示了当前在此进程上下文中运行的所有线程，每个线程的ID和优先级都在此显示。

除了线程信息，还可以选择Modules!菜单项。这将使ProcessInfo枚举系统中目前已载入的所有模块，并将每个模块的名称放入顶部的组合框中。然后，ProcessInfo选中第一个模块并显示其相关信息，如图4-5所示。

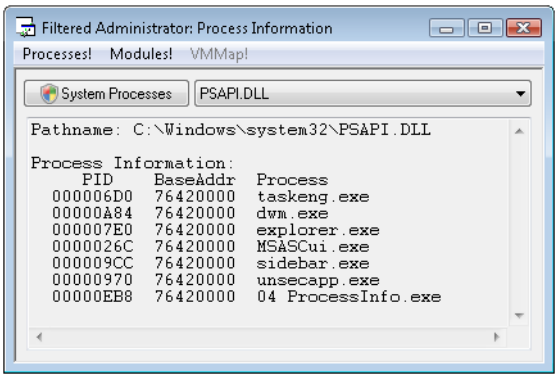


图4-5 ProcessInfo显示了在进程的地址空间内加载了Psapi.dll的所有进程

以这种方式使用ProcessInfo实用程序时，可以很容易地判断出哪些进程正在使用一个特定的模块。模块的完整路径显示在顶部。在下方的Process Information（进程信息）区域，显示了包含此模块的所有进程的一个列表。除了每个进程的ID和名称，还显示了模块加载到每个进程的什么地址。

简单地说，ProcessInfo应用程序显示的所有信息都是通过调用各种ToolHelp函数来生成的。为了使ToolHelp函数更易于使用，我创建了一个CToolhelp C++类（包含在Toolhelp.h文件中）。这个C++类封装了一个ToolHelp snapshot，简化了对其他ToolHelp函数的调用。

ProcessInfo.cpp 中的**GetModulePreferredBaseAddr**函数特别有意思：

```
PVOID GetModulePreferredBaseAddr(
    DWORD dwProcessId,
    PVOID pvModuleRemote);
```

此函数接受一个进程ID以及这个进程内的一个模块的地址作为参数。然后，它会查看那个进程的地址空间，定位那个模块，然后读取模块的header信息以确定这个模块的首选基地址。模块应该始终加载到它的首选基地址；否则，使用此模块的应用程序在初始化的时候就需要更多的内存，并会使其性能受到影响。因为这是一个比较让人讨厌的情形，所以我特地添加了这个函数，并在界面中指出了一个模块未在其首选基地址加载的情况。20.7节将进一步介绍首选基地址以及它对时间/内存性能的影响。

进程的命令行不能直接获得。正如*MSDN Magazine*题为“Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities, Part 2”的一篇文章（网址为*http://msdn.microsoft.com/msdnmag/issues/02/08/EscapefromDLLHell/*）所解释的那样，你需要挖掘一下远程进程的“进程环境块”（Process Environment Block，PEB）来找到命令行。不过，

自Windows XP起发生了两个重要的变化，我们需要对此专门解释一下。

首先，在WinDbg（可从*http://www.microsoft.com/whdc/devtools/debugging/default.mspx*下载）中，为了获得一个内核风格的PEB结构的详情，你要使用的命令已经发生了变化。现在不再使用kdex2x86扩展所实现的“**strct**”。相反，你只需调用“**dt**”命令。例如，执行“**dt nt! PEB**”，将列出以下PEB定义：

```
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged : UChar
+0x003 BitField : UChar
+0x003 ImageUsesLargePages : Pos 0, 1 Bit
+0x003 IsProtectedProcess : Pos 1, 1 Bit
+0x003 IsLegacyProcess : Pos 2, 1 Bit
+0x003 IsImageDynamicallyRelocated : Pos 3, 1 Bit
+0x003 SpareBits : Pos 4, 4 Bits
+0x004 Mutant : Ptr32 Void
+0x008 ImageBaseAddress : Ptr32 Void
+0x00c Ldr : Ptr32 _PEB_LDR_DATA
+0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
+0x014 SubSystemData : Ptr32 Void
+0x018 ProcessHeap : Ptr32 Void
...
```

RTL_USER_PROCESS_PARAMETERS结构的定义如下，这是由WinDbg中的“**dt nt!_RTL_USER_PROCESS_PARAMETERS**”命令所列出的：

```
+0x000 MaximumLength : Uint4B
+0x004 Length : Uint4B
+0x008 Flags : Uint4B
+0x00c DebugFlags : Uint4B
+0x010 ConsoleHandle : Ptr32 Void
+0x014 ConsoleFlags : Uint4B
+0x018 StandardInput : Ptr32 Void
+0x01c StandardOutput : Ptr32 Void
+0x020 StandardError : Ptr32 Void
+0x024 CurrentDirectory : _CURDIR
+0x030 DllPath : _UNICODE_STRING
+0x038 ImagePathName : _UNICODE_STRING
+0x040 CommandLine : _UNICODE_STRING
+0x048 Environment : Ptr32 Void
...
```

这样一来，就可以对以下内部结构进行计算，从而帮助我们“挖掘”出命令行：

```
typedef struct
{
   DWORD Filler[4];
   DWORD InfoBlockAddress;
} __PEB;
typedef struct
{
   DWORD Filler[17];
   DWORD wszCmdLineAddress;
} __INFOBLOCK;
```

其次，如同第14章要讲到的那样，在Windows Vista中，系统DLLs是在进程地址空间的随机地址加载的。所以，不要像在Windows XP中那样将PEB的地址硬编码为0x7ffdf000。相反，你需要调用**NtQueryInformationProcess**，并传递**ProcessBasicInformation**作为参数。别忘了，在一个版本的Windows中发现的未文档化的细节可能在下一个版本中发生改变。

最后但同样很重要的一点，在使用Process Information应用程序时，你会注意到某些进程已在组合框中列出，但没有已加载的DLL之类的详细信息可以显示。例如，audiodg.exe（Windows Audio Device Graph Isolation）是一个**受保护的进程**。这种新的进程类型是从Windows Vista开始引入的。

例如，可以用它为DRM（数字权限保护）应用程序提供更大程度的隔离。另外，理所当然地，远程进程访问受保护进程的虚拟内存的权限也被取消了。由于这个权限是列出已加载的DLL所必须的，所以ToolHelp API自然无法返回这些细节了。可从以下网址下载受保护进程的白皮书：*http://www.microsoft.com/whdc/system/vista/process_Vista.mspx*。

Process Information应用程序之所以不能获取一个正在运行的进程的所有详情，还可能是由于另一个原因。如果这个工具是在未提升权限的情况下启动的，就可能不能访问（自然也不能修改）以提升的权限来启动的进程。事实上，其限制远远不止在程序的用户界面上显示的那么单纯。Windows Vista还实现了另一个安全机制，即“Windows完整性机制”（Windows Integrity Mechanism），以前称为“强制完整性控制”（Mandatory Integrity Control）。

除了众所周知的安全描述符（SID）和访问控制列表（ACL），现在还为一个受保护的资源分配了一个所谓的**完整性级别**（integrity level），这是通过**系统访问控制列表**（SACL）中的一个新增的ACE（访问控制项）来实现的。凡是没有这个ACE的安全对象，将被操作系统默认为拥有“中”（Medium）完整性级别。另外，每个进程都有一个基于其安全令牌的完整性级别，它与系统授予的一个信任级别是对应的，如表4-10所示。

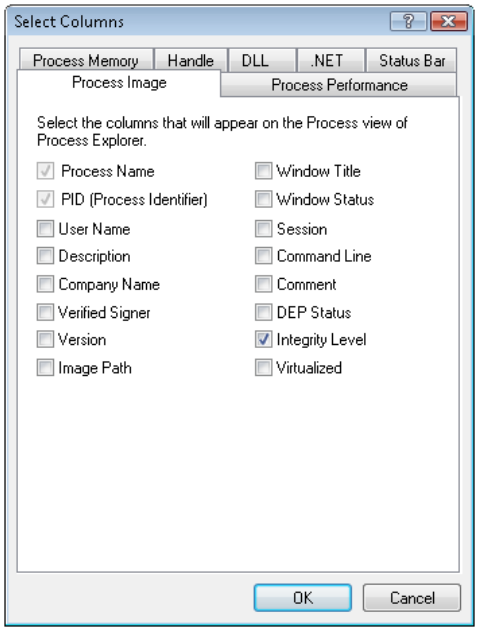
表4-10　信任级别

级别	应用程序示例
低	保护模式中的Internet Explorer是以“低”信任级别来运行的，它将拒绝从网上下载的代码修改其他应用程序和Windows环境。
中	默认情况下，所有应用程序都以“中”信任级别来启动，并使用一个筛选过的令牌来运行。
高	如果以提升后的权限来启动，会以“高”信任级别来运行。
系统	只有以Local System或Local Service的身份运行的进程，才能获得这个信任级别。

代码试图访问一个内核对象时，系统会将主调进程的完整性级别与内核对象的完整性级别进行比较。如果后者高于前者，就拒绝修改和删除操作。注意，这个比较是在检查ACL之前就完成的。所以，即便进程拥有访问资源的权限，但由于它运行时使用的完整性级别低于资源所要求的完整性级别，所以仍会被拒绝访问。假如一个应用程序要运行从网上下载的代码或脚本，这个设计就尤其重要。在Windows Vista上运行的Internet Explorer 7正是利用了这个机制，以“低”完整性级别来运行。这样一来，下载的代码就不能更改其他任何应用程序的状态，因为那些应用程序的进程默认是以“中”完整性级别来运行的。

提示

利用Sysinternals免费提供的Process Explorer工具（网址为*http://www.microsoft.com/technet/sysinternals/ProcessesAndThreads/ProcessExplorer.mspx*），可以查看进程的完整性级别。为此，请在Select Columns对话框的“Process Image”卡片中勾选“Integrity Level”。



源代码中的**GetProcessIntegrityLevel**函数演示了如何以编程方式来获取同样的细节及其他更多的内容。利用Sysinternals免费提供的一个控制台模式的AccessChk工具（网址为*http://www.microsoft.com/technet/sysinternals/utilities/accesschk.mspx*），可以列出访问文件、文件夹和注册表项等资源时所需的完整性级别（使用-i或-e命令行开关）。最后但同时也是最重要的一点，Windows Vista的控制台实用程序icacls.exe提供了一个**/setintegritylevel**命令行开关，它可以设置一个文件系统资源的完整性级别。

一旦知道了一个进程的令牌的完整性级别，以及它要访问的一个内核对象的完整性级别，系统就可以根据令牌和资源中都存储着的一个代码*策略*来核实具体能采取哪种操作。首先调用**GetTokenInformation**，向它传入**TokenMandatoryPolicy**和进程的安全令牌句柄。**GetTokenInformation**返回的是一个DWORD值，其中包含了一个按位掩码（bitwise mask），后者详细描述了适用的策略。表4-11列出了可能的策略。

表4-11 代码策略

WinNT.h 中的 TOKEN_MANDATORY_*常量	**描述**
POLICY_NO_WRITE_UP	在这个安全令牌下运行的代码不能向具有更高完整性级别的资源写入。
POLICY_NEW_PROCESS_MIN	在这个安全令牌下运行的代码启动一个新的进程时，子进程将检查父进程和manifest中描述的优先级，并从中选择最低的一个优先级。如果没有manifest的话，就假定manifest中的优先级为“中”。

利用已经定义好的另外两个常量，可以轻松地判断要么没有策略（**TOKEN_MANDATORY_POLICY_OFF**定义为**0**），要么有一个按位掩码（**TOKEN_MANDATORY_POLICY_VALID_MASK**），以便你对一个策略值进行验证（参见ProcessInfo.cpp中的源代码）。

其次，将根据与内核对象关联的Label ACE的按位掩码来设置资源策略（参见ProcessInfo.cpp中的**GetProcessIntegrityLevel**函数来查看实现细节）。为了决定允许或拒绝在资源上进行哪种访

问，可以使用两个**资源策略**。默认的资源策略是**SYSTEM_MANDATORY_LABEL_NO_WRITE_UP**，它指出一个较低完整性级别的进程可以读取但不能写入或删除一个较高完整性的资源。**SYSTEM_MANDATORY_LABEL_NO_READ_UP**资源策略的限制性更强，因为它不允许较低完整性级别的进程读取较高完整性的资源。

注意
对于一个高完整性级别的进程内核对象，即使已经设置了“No-Read-Up”，另一个完整性级别较低的进程也能在较高完整性地址空间中读取——只要向这个进程授予了Debug权限。这解释了为什么以管理员身份运行Process Information工具时（为了授予自己Debug权限，就必须以管理员身份运行），它能读取具有System完整性级别的那些进程（比如一个服务进程）的命令行。

除了在进程之间提供内核对象的访问保护，窗口系统还利用完整性级别来拒绝低完整性级别的进程访问/更新高完整性级别的进程的用户界面。这个机制称为**用户界面权限隔离**（User Interface Privilege Isolation，UIPI）。操作系统将封锁从完整性级别较低的进程post的（通过**PostMessage**）、发送的（通过**SendMessage**）或者拦截的（通过Windows hook）Windows消息，阻止完整性级别较高的进程拥有的一个窗口获取信息或者被注入虚假的信息。为了更好地体会这一点，你可以使用WindowDump这个实用程序来做实验（*http://download.microsoft.com/download/8/3/f/83f69587-47f1-48e2-86a6-aab14f01f1fe/EscapeFromDLLHell.exe*）。为了获得一个列表框的每一项所显示的文本，WindowDump调用**SendMessage**函数，并将**LB_GETCOUNT**作为参数来传递，这样可以获得要读取文本内容的元素的数量。随后，我们调用**SendMessage**，并将**LB_GETTEXT**作为参数来获取文本内容。然而，假如WindowDump进程运行时使用的完整性级别低于拥有列表框的那个进程，那么第一个**SendMessage**调用虽然能成功，但会返回0作为元素数量。如果以“中”完整性级别来启动Spy++，那么用它获取一个窗口的消息时，如果那个窗口是由一个较高完整性级别的进程创建的，就会观察到相同的行为（对结果进行筛选）。

第 5 章　作业

本章内容包括：
- 对作业中的进程施加限制
- 将进程放入作业中
- 终止作业中的所有进程
- 作业通知
- Job Lab示例程序

我们经常都需要将一组进程当作单个实体来处理。例如，在指示Visual Studio构造一个C++项目的时候，它会生成Cl.exe，后者可能必须生成更多的进程（比如编译器的每一个pass；在每个pass中，都对源程序或源程序的中间结果从头到尾扫描一次，并进行相关的加工处理，从而生成新的中间结果或目标程序——译者注)。但是，如果用户希望提前停止构造过程，Visual Studio必须能够以某种方式终止Cl.exe及其所有子进程。虽然这是一个简单而常见的问题，但在Microsoft Windows中解决起来非常难，这是由于Windows没有维护进程之间的父/子关系。具体地说，即

使父进程已经终止运行，子进程仍在继续运行。

设计一个服务器时，也必须把一组进程当作一个单独的组来处理。例如，一个客户端也许会请求服务器执行一个应用程序（后者也许生成自己的子进程）并向客户端返回结果。由于许多客户端都可能连接到此服务器，所以服务器应该以某种方式限制客户端能请求的东西，避免任何一个客户端独占其所有资源。这些限制包括可以分配给客户端请求的最大CPU时间；最小和最大工作区（working set）大小；禁止客户端应用程序关闭计算机；以及安全限制。

Windows提供了一个作业（job）内核对象，它允许你将进程组合在一起并创建一个“沙箱”来限制进程能够做什么。最好将作业对象想象成一个进程容器。但是，即使作业中只包含一个进程，也是非常有用的，因为这样可以对进程施加平时不能施加的限制。

我的**StartRestrictedProcess**函数将一个进程放入一个作业中，以限制此进程具体能够做哪些事情，如下所示：

```
void StartRestrictedProcess() {
  // Check if we are not already associated with a job.
  // If this is the case, there is no way to switch to
  // another job.
  BOOL bInJob = FALSE;
  IsProcessInJob(GetCurrentProcess(), NULL, &bInJob);
  if (bInJob) {
    MessageBox(NULL, TEXT("Process already in a job"),
         TEXT(""), MB_ICONINFORMATION | MB_OK);
    return;
  }
  // Create a job kernel object.
  HANDLE hjob = CreateJobObject(NULL,
    TEXT("Wintellect_RestrictedProcessJob"));
  // Place some restrictions on processes in the job.
  // First, set some basic restrictions.
  JOBOBJECT_BASIC_LIMIT_INFORMATION jobli = { 0 };
  // The process always runs in the idle priority class.
  jobli.PriorityClass = IDLE_PRIORITY_CLASS;
  // The job cannot use more than 1 second of CPU time.
  jobli.PerJobUserTimeLimit.QuadPart = 10000; // 1 sec in 100-ns intervals
  // These are the only 2 restrictions I want placed on the job (process).
  jobli.LimitFlags = JOB_OBJECT_LIMIT_PRIORITY_CLASS
    | JOB_OBJECT_LIMIT_JOB_TIME;
  SetInformationJobObject(hjob, JobObjectBasicLimitInformation, &jobli,
    sizeof(jobli));
  // Second, set some UI restrictions.
  JOBOBJECT_BASIC_UI_RESTRICTIONS jobuir;
  jobuir.UIRestrictionsClass = JOB_OBJECT_UILIMIT_NONE; // A fancy zero
  // The process can't log off the system.
  jobuir.UIRestrictionsClass |= JOB_OBJECT_UILIMIT_EXITWINDOWS;
  // The process can't access USER objects (such as other windows)
  // in the system.
  jobuir.UIRestrictionsClass |= JOB_OBJECT_UILIMIT_HANDLES;
  SetInformationJobObject(hjob, JobObjectBasicUIRestrictions, &jobuir,
  sizeof(jobuir));
  // Spawn the process that is to be in the job.
  // Note: You must first spawn the process and then place the process in
  // the job. This means that the process' thread must be initially
  // suspended so that it can't execute any code outside of the job's
  // restrictions.
  STARTUPINFO si = { sizeof(si) };
  PROCESS_INFORMATION pi;
  TCHAR szCmdLine[8];
  _tcscpy_s(szCmdLine, _countof(szCmdLine), TEXT("CMD"));
  BOOL bResult =
  CreateProcess(
  NULL, szCmdLine, NULL, NULL, FALSE,
  CREATE_SUSPENDED | CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
  // Place the process in the job.
  // Note: If this process spawns any children, the children are
  // automatically part of the same job.
  AssignProcessToJobObject(hjob, pi.hProcess);
  // Now we can allow the child process' thread to execute code.
```

```
  ResumeThread(pi.hThread);
  CloseHandle(pi.hThread);
  // Wait for the process to terminate or
  // for all the job's allotted CPU time to be used.
  HANDLE h[2];
  h[0] = pi.hProcess;
  h[1] = hjob;
  DWORD dw = WaitForMultipleObjects(2, h, FALSE, INFINITE);
  switch (dw – WAIT_OBJECT_0) {
  case 0:
  // The process has terminated...
  break;
  case 1:
  // All of the job's allotted CPU time was used...
  break;
  }
  FILETIME CreationTime;
  FILETIME ExitTime;
  FILETIME KernelTime;
  FILETIME UserTime;
  TCHAR szInfo[MAX_PATH];
  GetProcessTimes(pi.hProcess, &CreationTime, &ExitTime,
  &KernelTime, &UserTime);
  StringCchPrintf(szInfo, _countof(szInfo), TEXT("Kernel = %u | User = %u\n"),
  KernelTime.dwLowDateTime / 10000, UserTime.dwLowDateTime / 10000);
  MessageBox(GetActiveWindow(), szInfo, TEXT("Restricted Process times"),
  MB_ICONINFORMATION | MB_OK);
  // Clean up properly.
  CloseHandle(pi.hProcess);
  CloseHandle(hjob);
  }
```

现在来解释**StartRestrictedProcess**的工作方式。首先，我将**NULL**作为第2个参数传给以下函数，验证当前进程是否在一个现有的作业控制之下运行：

```
BOOL IsProcessInJob(
   HANDLE hProcess,
   HANDLE hJob,
  PBOOL pbInJob);
```

如果进程已与一个作业关联，就无法将当前进程或者它的任何子进程从作业中去除。这个安全特性可以确保你无法逃避对你施加的限制。

警告

默认情况下，在Windows Vista中通过Windows资源管理器来启动一个应用程序时，进程会自动同一个专用的作业关联，此作业的名称使用了"PCA"字符串前缀。正如本章稍后的“作业通知”一节要讲述的那样，作业中的一个进程退出时，我们是可以接收到一个通知的。所以，一旦通过Windows资源管理器启动的一个历史遗留的程序出现问题，就会触发Program Compatibility Assistant（程序兼容性助手）。

如果你的应用程序需要创建像本章最后展示的Job Lab程序那样的一个作业，只能说你的运气不太好。这个创建必定会失败，因为已经和你的进程关联了带"PCA"前缀的作业对象。

Windows Vista提供这个功能的目的是检测兼容性问题。所以，如果你已经像第4章描述的那样为应用程序定义了一个清单（manifest），Windows资源管理器就不会将你的进程同"PCA"前缀的作业关联，它会假定你已经解决了任何可能的兼容性问题。

但是，在需要调试应用程序的时候，如果调试器是从Windows资源管理器启动的，即使有一个清单（mainifest），应用程序也会从调试器继承带有"PCA"前缀的作业。一个简单的解决方案是从命令行而不是Windows资源管理器中启动调试器。在这种情况下，不会发生与作业的关联。

然后，我通过以下调用来创建一个新的作业内核对象：

```
HANDLE CreateJobObject(
    PSECURITY_ATTRIBUTES psa,
    PCTSTR pszName);
```

和所有内核对象一样，第一个参数将安全信息与新的作业对象关联，然后告诉系统，是否想要使返回的句柄成为可继承的句柄。最后一个参数对此作业对象进行命名，使其能够由另一个进程通过**OpenJobObject**函数（详见本章后文）进行访问，如下所示：

```
HANDLE OpenJobObject(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);
```

和往常一样，如果确定在自己的代码中不再访问作业对象，就必须调用CloseHandle来关闭它的句柄。这一点在前面的**StartRestrictedProcess**函数的末尾有所体现。务必记住，关闭一个作业对象，不会迫使作业中的所有进程都终止运行。作业对象实际只是加了一个删除标记，只有在作业中的所有进程都已终止运行之后，才会自动销毁。

注意，关闭作业的句柄会导致所有进程都不可访问此作业，即使这个作业仍然存在。如以下代码所示：

```
// Create a named job object.
HANDLE hJob = CreateJobObject(NULL, TEXT("Jeff"));
// Put our own process in the job.
AssignProcessToJobObject(hJob, GetCurrentProcess());
// Closing the job does not kill our process or the job.
// But the name ("Jeff") is immediately disassociated with the job.
CloseHandle(hJob);
// Try to open the existing job.
hJob = OpenJobObject(JOB_OBJECT_ALL_ACCESS, FALSE, TEXT("Jeff"));
// OpenJobObject fails and returns NULL here because the name ("Jeff")
// was disassociated from the job when CloseHandle was called.
// There is no way to get a handle to this job now.
```

5.1 对作业中的进程施加限制

创建好一个作业之后，接着一般需要限制作业中的进程能做的事情；换言之，现在要设置一个“沙箱”。可以向作业应用以下几种类型的限制：

- 基本限制和扩展基本限制，防止作业中的进程独占系统资源。
- 基本的UI限制，防止作业内的进程更改用户界面。
- 安全限制，防止作业内的进程访问安全资源（文件、注册表子项等）。

可以通过调用以下代码向作业应用限制：

```
BOOL SetInformationJobObject(
  HANDLE hJob,
  JOBOBJECTINFOCLASS JobObjectInformationClass,
  PVOID pJobObjectInformation,
  DWORD cbJobObjectInformationSize);
```

第一个参数指定要限制的作业。第二个参数是一个枚举类型，指定了要应用的限制的类型。第三个参数是一个数据结构的地址，数据结构中包含了具体的限制设置。第四个参数指出此数据结构的大小（用于版本控制）。表5-1总结了如何设置这些限制。

表5-1 限制类型

限制	第二个参数的值	第三个参数的结构

类型		
基本限制	JobObjectBasicLimitInformation	JOBOBJECT_BASIC_LIMIT_INFORMATION
扩展后的基本限制	JobObjectExtendedLimitInformation	JOBOBJECT_EXTENDED_LIMIT_INFORMATION
基本的UI限制	JobObjectBasicUIRestrictions	JOBOBJECT_BASIC_UI_RESTRICTIONS
安全限制	JobObjectSecurityLimitInformation	JOBOBJECT_SECURITY_LIMIT_INFORMATION

在我的**StartRestrictedProcess**函数中，我只对作业设置了一些基本的限制。我分配了一个**JOBOBJECT_BASIC_LIMIT_INFORMATION**结构，初始化它，然后调用了**SetInformationJobObject**。**JOBOBJECT_BASIC_LIMIT_INFORMATION**结构如下所示：

```
typedef struct _JOBOBJECT_BASIC_LIMIT_INFORMATION {
    LARGE_INTEGER PerProcessUserTimeLimit;
    LARGE_INTEGER PerJobUserTimeLimit;
    DWORD LimitFlags;
    DWORD MinimumWorkingSetSize;
    DWORD MaximumWorkingSetSize;
    DWORD ActiveProcessLimit;
    DWORD_PTR Affinity;
    DWORD PriorityClass;
    DWORD SchedulingClass;
} JOBOBJECT_BASIC_LIMIT_INFORMATION,
*PJOBOBJECT_BASIC_LIMIT_INFORMATION;
```

表5-2简要描述了这些成员。

表5-2　JOBOBJECT_BASIC_LIMIT_INFORMATION成员

成员	描述	备注
PerProcessUserTimeLimit	指定分配给每个进程的最大用户模式时间（时间间隔为100纳秒）	对于占用时间超过其分配时间的任何进程，系统将自动终止它的运行。要设置这个限制，请在LimitFlags成员中指定JOB_OBJECT_LIMIT_PROCESS_TIME标志
PerJobUserTimeLimit	限制分配给作业对象的最大用户模式时间（时间间隔为100纳秒)	默认情况下，在达到该时间限制时，系统将自动终止所有进程的运行。可以在作业运行时定期改变这个值。要设置这个限制，请在LimitFlags成员中指定JOB_OBJECT_LIMIT_JOB_TIME标志
LimitFlags	指定将哪些限制应用于作业	详细说明参见后文
MinimumWorkingSetSize MaximumWorkingSetSize	指定每个进程（并不是作业中的所有进程）的最小和最大工作区大小	正常情况下，进程的工作区能扩展至最大值以上；设置了MaximumWorkingSetSize后，就可以对其施加硬性限制。一旦进程的工作区抵达这个限制，进程就会对自己进行分页。除非进程只是尝试清空它的工作区，否则一个单独的进程对SetProcessWorkingSetSize的调用会被忽略。要设置该限制，请在LimitFlags成员中指定JOB_OBJECT_LIMIT_WORKINGSET标志
ActiveProcessLimit	指定作业中能并发运行	超过此限制的任何企图都会导致新进程终止，并报告一

	的进程的最大数量	个“配额不足”错误。要设置这个限制，请在LimitFlags成员中指定JOB_OBJECT_LIMIT_ACTIVE_PROCESS标志
Affinity	指定能够运行进程的CPU子集	单独的进程可以进一步对此进行限制。要设置这个限制，请在LimitFlags成员中指定JOB_OBJECT_LIMIT_AFFINITY标志
PriorityClass	指定关联的所有进程的优先级类(priority class)	如果一个进程调用SetPriorityClass函数，即使该函数调用失败，也会成功返回。如果进程调用GetPriorityClass函数，该函数将返回进程已经设置的优先级类，尽管这可能并不是进程的实际优先级类。此外，SetThreadPriority无法将线程的优先级提高到Normal以上，但是可以用它降低线程的优先级。要设置这个限制，请在LimitFlags成员中指定JOB_OBJECT_LIMIT_PRIORITY_CLASS标志
SchedulingClass	为作业中的线程指定一个相对时间量差（relative time quantum difference）	值可以在0～9之间（包括0和9），默认是5。详细说明参见后文。要设置这个限制，请在LimitFlags成员中指定JOB_OBJECT_LIMIT_SCHEDULING_CLASS标志

关于这个结构，我认为Platform SDK文档中解释得不够清楚，所以这里要稍微多说几句。在**LimitFlags**成员中，可以设置位标志来指定希望应用于此作业的限制条件。例如，在我的**StartRestrictedProcess**函数中，我设置了**JOB_OBJECT_LIMIT_PRIORITY_CLASS**和**JOB_OBJECT_LIMIT_JOB_TIME**这两个位标志。这意味着我只对作业施加了两个限制。至于CPU Affinity（CPU亲和性，即限制进程能用什么CPU来运行）、工作区大小、每进程的CPU时间等等，我没有进行任何限制。

作业运行过程中，它会维护一些统计信息，比如作业中的进程使用了多少CPU时间。每次使用**JOB_OBJECT_LIMIT_JOB_TIME** 标志来设置基本限制的时候，作业都会扣除已终止运行的进程的CPU时间统计信息，从而显示当前活动的进程使用了多少CPU时间。但是，假如你想改变作业的CPU亲和性，同时不想重置CPU时间统计信息，又该怎么办呢？为此，必须使用**JOB_OBJECT_LIMIT_AFFINITY**标志设置一个新的基本限制，同时必须取消设置**JOB_OBJECT_LIMIT_JOB_TIME**标志。但是，假如这样做，相当于告诉作业你不再希望施加CPU时间限制。而这并不是你希望的。

现在，你希望既更改Affinity限制，又保留现有的CPU时间限制；只是不希望扣除已终止进程的CPU时间统计信息。为了解决这个问题，请使用一个特殊的标志，即**JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME**。这个标志和**JOB_OBJECT_LIMIT_JOB_TIME**标志是互斥的。**JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME**标志指出你希望在改变限制条件的同时，不扣除已终止运行的那些进程的CPU时间统计信息

接着研究一下**JOBOBJECT_BASIC_LIMIT_INFORMATION**结构的**SchedulingClass**成员。设想现在有两个作业正在运行，而且你将这两个作业的优先级类（priority class）都设为**NORMAL_PRIORITY_CLASS**。但是，你还希望一个作业中的进程能够比另一个作业中的进程获得更多的CPU时间。为此，可以使用**SchedulingClass**成员来更改具有相同“优先级类”的两个作业的相对作业调度（relative scheduling of jobs）。可以设置0~9（含0和9）的任何一个值，默认值是5。在Windows Vista中，这个值越大，表明系统要为特定作业中的进程中的线程分配一个更长的时间量 (time quantum)；值越小，表明线程的时间量越小。

例如，假定现在有两个Normal优先级类的作业。每个作业都有一个进程，而且每个进程都只有一个（Normal优先级）线程。正常情况下，这两个线程以round-robin（轮询）方式调度，各自

获得相同的时间量。但是，如果将第一个作业的**SchedulingClass**成员设置为3，那么为这个作业中的线程调度CPU时间时，它们的时间量将少于第二个作业中的线程。

使用**SchedulingClass**成员时，应避免使用很大的数字（进而造成很大的时间量）。这是由于时间量变得越大，系统中的其他作业、进程以及线程的反应会变得越来越迟钝。

值得注意的最后一个限制是**JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION**限制标志。这个限制会导致系统关闭与作业关联的每一个进程的“未处理的异常”对话框。为此，系统会为作业中的每个进程调用SetErrorMode函数，并向它传递SEM_NOGPFAULTERRORBOX标志。作业中的一个进程在引发一个未处理的异常后，会立即终止运行，不会显示任何用户界面。对于服务和其他面向批处理任务的作业，这是相当有价值的一个限制标志。如果不设置这个标志，作业中的进程就能抛出异常，而且永远不会终止运行，从而造成系统资源的浪费。

除了基本限制外，还可以使用**JOBOBJECT_EXTENDED_LIMIT_INFORMATION**结构对作业施加扩展限制：

```
typedef struct _JOBOBJECT_EXTENDED_LIMIT_INFORMATION {
    JOBOBJECT_BASIC_LIMIT_INFORMATION BasicLimitInformation;
    IO_COUNTERS IoInfo;
    SIZE_T ProcessMemoryLimit;
    SIZE_T JobMemoryLimit;
    SIZE_T PeakProcessMemoryUsed;
    SIZE_T PeakJobMemoryUsed;
} JOBOBJECT_EXTENDED_LIMIT_INFORMATION,
*PJOBOBJECT_EXTENDED_LIMIT_INFORMATION;
```

可以看出，该结构包含一个**JOBOBJECT_BASIC_LIMIT_INFORMATION**结构，这就使它成为基本限制的一个超集。该结构有点诡异，因为它包含了与限制作业无关的成员。首先，**IoInfo**是保留成员，不应以任何方式访问它。我将在本章稍后讨论如何查询I/O计数器信息。此外，**PeakProcessMemoryUsed**和**PeakJobMemoryUsed**成员是只读的，分别告诉你作业中的任何一个进程和全部进程所需的最大已提交存储空间量。

其余两个成员**ProcessMemoryLimit**和**JobMemoryLimit**分别限制着作业中的任何一个进程或全部进程所使用的已提交存储空间。为了设置这样的限制，需要在**LimitFlags**成员中分别指定**JOB_OBJECT_LIMIT_JOB_MEMORY**和**JOB_OBJECT_LIMIT_PROCESS_MEMORY**标志。

再来看看能对作业施加的其他限制。JOBOBJECT_BASIC_UI_RESTRICTIONS结构如下所示：

```
typedef struct _JOBOBJECT_BASIC_UI_RESTRICTIONS {
    DWORD UIRestrictionsClass;
} JOBOBJECT_BASIC_UI_RESTRICTIONS, *PJOBOBJECT_BASIC_UI_RESTRICTIONS;
```

该结构只有一个数据成员，即**UIRestrictionsClass**，它容纳着如表5-3所示的位标志集合。

表5-3　针对作业对象基本用户界面限制的位标志

标志	描述
JOB_OBJECT_UILIMIT_EXITWINDOWS	阻止进程通过ExitWindowsEx函数注销、关机、重启或断开系统电源。
JOB_OBJECT_UILIMIT_READCLIPBOARD	阻止进程读取剪贴板中的内容

JOB_OBJECT_UILIMIT_WRITECLIPBOARD	阻止进程清除剪贴板中的内容
JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS	阻止进程通过SystemParametersInfo 函数更改系统参数
JOB_OBJECT_UILIMIT_DISPLAYSETTINGS	阻止进程通过ChangeDisplaySettings函数更改显示设置
JOB_OBJECT_UILIMIT_GLOBALATOMS	为作业指定其专有的全局atom表，并规定作业中的进程只能访问作业的表
JOB_OBJECT_UILIMIT_DESKTOP	阻止进程使用CreateDesktop或SwitchDesktop函数来创建或切换桌面
JOB_OBJECT_UILIMIT_HANDLES	阻止作业中的进程使用同一个作业外部的进程所创建的USER对象(比如HWND)

最后一个标志**JOB_OBJECT_UILIMIT_HANDLES**特别有意思。该限制意味着作业中的任何一个进程都不能访问作业外部的进程所创建的**USER**对象。所以，如果试图在一个作业内运行Microsoft Spy++，就只能看到Spy++自己创建的窗口，看不到其他任何窗口。图5-1展示了已打开两个MDI子窗口的Spy++程序。

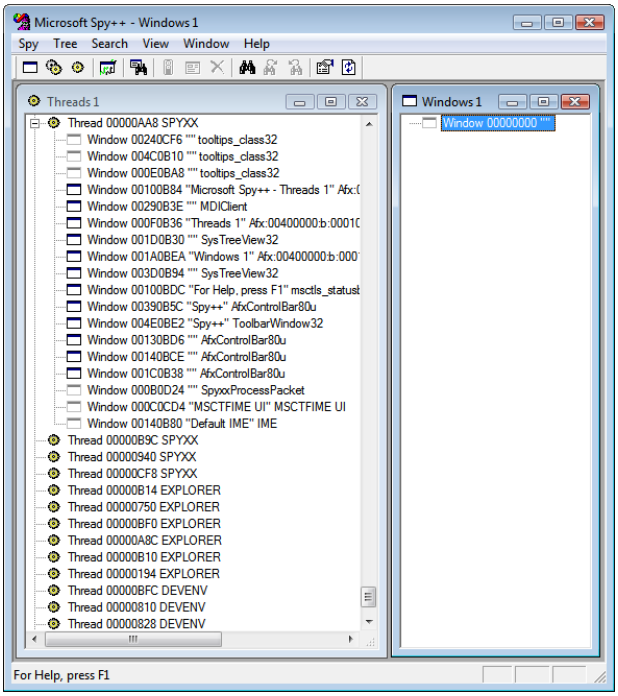


图5-1　在限制了对UI句柄的访问的一个作业中运行的Microsoft Spy++

注意，Threads 1窗口包含系统中的线程的一个列表。在这些线程中，只有一个线程00000AA8 SPYXX似乎已经创建了窗口。这是因为我让Spy++在它自己的作业中运行，并限制了它对UI句柄的使用。在同一个窗口中，可以看到**EXPLORER**和**DEVENV**线程，但它看上去并没有创建任何窗口。但我可以肯定这些线程已创建了窗口，只是Spy++不能访问它们。在右侧，可以看到Windows 1窗口。在这个窗口中，Spy++显示了桌面上现有的所有窗口的一个层次结构。注意其中只有一项，即00000000。Spy++只是把它作为占位符使用。

注意，这个UI限制只是单向的。也就是说，作业外部的进程可以看到作业内部的进程所创建的USER对象。例如，假如我在一个作业内部运行记事本程序，在一个作业外部运行Spy++程序，那么Spy++是可以看到记事本程序的窗口的，即使记事本程序所在的那个作业指定了**JOB_OBJECT_UILIMIT_HANDLES**标志。另外，如果Spy++在它自己的作业中，那么也可以看到记事本程序的窗口，除非作业已经指定了**JOB_OBJECT_UILIMIT_HANDLES**标志。

要为作业中的进程创建一个真正安全的沙箱，对UI句柄进行限制是十分强大的一个能力。不过，有时仍然需要让作业内部的一个进程同作业外部的一个进程通信。

为了做到这一点，一个简单的办法是使用窗口消息。但是，如果作业中的进程不能访问UI句柄，那么作业内部的进程就不能向作业外部的进程创建的一个窗口发送或张贴窗口消息。幸运的是，可以用另一个函数来解决这个问题，如下所示：

```
BOOL UserHandleGrantAccess(
    HANDLE hUserObj,
    HANDLE hJob,
    BOOL bGrant);
```

hUserObj参数指定一个USER对象，我们想允许或拒绝作业内部的进程访问此对象。这几乎总是一个窗口句柄，但它也可能是其他USER对象，比如桌面、挂钩（hook）、图标或菜单。最后两个参数**hJob**和**bGrant**指出要对哪个作业进行准许或拒绝访问授权。注意，如果从**hJob**所标识的作业内的一个进程内调用这个函数，函数调用会失败——这样可以防止作业内部的一个进程自己向自己授予一个对象的访问权。

我们可以为作业施加的最后一种类型的限制与安全性有关。注意，一旦应用，安全限制就不能撤消。**JOBOBJECT_SECURITY_LIMIT_INFORMATION**结构如下所示：

```
typedef struct _JOBOBJECT_SECURITY_LIMIT_INFORMATION {
DWORD SecurityLimitFlags;
HANDLE JobToken;
PTOKEN_GROUPS SidsToDisable;
PTOKEN_PRIVILEGES PrivilegesToDelete;
PTOKEN_GROUPS RestrictedSids;
} JOBOBJECT_SECURITY_LIMIT_INFORMATION,
*PJOBOBJECT_SECURITY_LIMIT_INFORMATION;
```

表5-4简要描述了其成员。

表5-4　JOBOBJECT_SECURITY_LIMIT_INFORMATION结构的成员

成员	描述
SecurityLimitFlags	指明是否不允许管理员访问，不允许无限制的令牌访问，强制使用特定的访问令牌，或者禁止特定的安全标识符（SID）和特权
JobToken	由作业中所有进程使用的访问令牌
SidsToDisable	指定要禁止对哪些SID进行访问访问检查
PrivilegesToDelete	指定要从访问令牌中删除哪些特权
RestrictedSids	指定将一组只能拒绝的SID添加到访问令牌中

很自然，既然能对作业施加限制，就必须能查询这些限制。通过调用以下函数，很容易实现这一点:

```
BOOL QueryInformationJobObject(
    HANDLE hJob,
    JOBOBJECTINFOCLASS JobObjectInformationClass,
    PVOID pvJobObjectInformation,
    DWORD cbJobObjectInformationSize,
    PDWORD pdwReturnSize);
```

向此函数传递作业的句柄（这类似于**SetInformationJobObject**函数）——这是一个枚举类型，它指出了你希望哪些限制信息，要由函数初始化的数据结构的地址，以及包含该数据结构的数据块的大小。最后一个参数是**pdwReturnSize**，它指向由此函数来填充的一个**DWORD**，指出缓冲区中已填充了多少个字节。如果对这个信息不在意，可以（通常也会）为此参数传递一个NULL值。

注意
作业中的进程可以调用QueryInformationJobObject获得其所属作业的相关信息（为作业句柄参数传递NULL值）。这是很有用的一个技术，因为它使进程能看到自己被施加了哪些限制。不过，如果为作业句柄参数传递NULL值，SetInformationJobObject函数调用会失败——目的是防止进程删除施加于自己身上的限制。

5.2 将进程放入作业中

好了，关于设置和查询限制的讨论到此为止。现在回过头来讨论我的**StartRestrictedProcess**函数。在对作业施加了某些限制之后，我通过调用**CreateProcess**来生成打算放到作业中的进程。但请注意，调用**CreateProcess**时，我使用的是**CREATE_SUSPENDED**标志。这样虽然会创建新进程，但是不允许它执行任何代码。由于**StartRestrictedProcess**函数是从不属于作业一部分的进程中执行的，所以子进程也不是作业的一部分。如果我允许子进程立即开始执行代码，它会“逃离”我的沙箱，成功地做一些我想禁止它做的事情。所以，在我创建此子进程之后且在允许它运行之前，必须调用以下函数，将进程显式地放入我新建的作业中：

```
BOOL AssignProcessToJobObject(
    HANDLE hJob,
    HANDLE hProcess);
```

这个函数向系统表明将此进程(由**hProcess**标识)当作现有作业(由**hJob**标识)的一部分。注意，这个函数只允许将尚未分配给任何作业的一个进程分配给一个作业，你可以使用**IsProcessInJob**函数对此进行检查。一旦进程已经属于作业的一部分，它就不能再移动到另一个作业中，也不能成为所谓“无作业”的。还要注意，当作业中的一个进程生成了另一个进程的时候，新进程将自动成为父进程所属于的作业的一部分。但可以通过以下方式改变这种行为。

- 打开JOBOBJECT_BASIC_LIMIT_INFORMATION的LimitFlags成员的JOB_OBJECT_LIMIT_BREAKAWAY_OK标志，告诉系统新生成的进程可以在作业外部执行。为此，必须在调用CreateProcess函数时指定新的CREATE_BREAKAWAY_FROM_JOB标志。如果这样做了，但作业并没有打开JOB_OBJECT_LIMIT_BREAKAWAY_OK限制标志，CreateProcess调用就会失败。如果希望由新生成的进程来控制作业，这就是非常有用的一个机制。
- 打开JOBOBJECT_BASIC_LIMIT_INFORMATION的LimitFlags成员的JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK标志。此标志也告诉系统新生成的子进程不是作业的一部分。但是，现在就没有必要向CreateProcess函数传递任何额外的标志。事实上，此标志会强制新进程脱离当前作业。如果进程在设计之初对作业对象一无所知，这个标志就相当有用。

至于我的**StartRestrictedProcess**函数，在调用了**AssignProcessToJobObject**之后，新进程就成为我的受限制的作业的一部分。然后，我调用**ResumeThread**，使进程的线程可以在作业的限制下执行代码。与此同时，我还关闭了到线程的句柄，因为我不再需要它了。

5.3 终止作业中的所有线程

对于作业，我们经常想做的一件事情就是杀死作业中的所有进程。本章伊始，我提到Visual Studio没有一个简单的办法来停止正在进行的一次生成，因为它必须知道哪些进程是从它生成的第一个进程生成的。(这非常难。我在Microsoft Systems Journal 1998年6月期的Win 32 Q&A专栏讨论

过Developer Studio是如何做到这一点的，可以通过以下网址找到这篇文章：*http://www.microsoft.com/msj/0698/win320698.aspx*。)也许Visual Studio未来的版本会转而使用作业，因为这样一来，代码的编写会变得更容易，而且可以用作业来做更多的事情。

要杀死作业内部的所有进程，只需调用以下代码：

```
BOOL TerminateJobObject(
    HANDLE hJob,
UINT uExitCode);
```

这类似于为作业内的每一个进程调用**TerminateProcess**，将所有退出代码设为**uExitCode**。

5.3.1 查询作业统计信息

前面讨论了如何使用**QueryInformationJobObject**函数来查询作业当前的限制。此外，我们还可以用它来获得作业的统计信息。例如，要获得基本的统计信息，可以调用函数**QueryInformationJobObject**，向第二个参数传递**JobObjectBasicAccountingInformation**和一个**JOBOBJECT_BASIC_ACCOUNTING_INFORMATION**结构的地址：

```
typedef struct _JOBOBJECT_BASIC_ACCOUNTING_INFORMATION {
LARGE_INTEGER TotalUserTime;
LARGE_INTEGER TotalKernelTime;
LARGE_INTEGER ThisPeriodTotalUserTime;
LARGE_INTEGER ThisPeriodTotalKernelTime;
DWORD TotalPageFaultCount;
DWORD TotalProcesses;
DWORD ActiveProcesses;
DWORD TotalTerminatedProcesses;
} JOBOBJECT_BASIC_ACCOUNTING_INFORMATION,
*PJOBOBJECT_BASIC_ACCOUNTING_INFORMATION;
```

表5-5简要描述了该结构的成员。

表5-5 JOBOBJECT_BASIC_ACCOUNTING_INFORMATION结构的成员

成员	描述
TotalUserTime	指出作业中的进程已使用了多少用户模式的CPU时间
TotalKernelTime	指出作业中的进程已使用了多少内核模式的CPU时间
ThisPeriodTotalUserTime	与TotalUserTime一样，不同的是，如果调用SetInformationJobObject来更改基本限制信息，同时没有使用JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME限制标志，这个值被重置为0
ThisPeriodTotalKernelTime	与ThisPeriodTotalUserTime一样，不同的是，这个值显示的是内核模式CPU时间
TotalPageFaultCount	指出作业中的进程产生的页错误总数
TotalProcesses	指出曾经成为作业一部分的所有进程的总数
ActiveProcesses	指定作业的当前进程的总数
TotalTerminatedProcesses	指出因为已超过预定CPU时间限制而被杀死的进程数

从**StartRestrictedProcess**函数实现的末尾可以看出，我们可以获得任何一个进程的CPU占用时间信息，即使此进程不属于任何一个作业，这是通过调用**GetProcessTimes**函数来实现的，详情请参见第7章。

除了查询基本的统计信息，还可以执行一个调用来同时查询基本统计信息和I/O（输入/输出）统计信息。为此，要向第二个参数传递**JobObjectBasicAndIoAccountingInformation**和一个**JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION**结构的地址：

```
typedef struct JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION {
    JOBOBJECT_BASIC_ACCOUNTING_INFORMATION BasicInfo;
    IO_COUNTERS IoInfo;
} JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION,
    *PJOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION;
```

可以看出，这个结构返回了**JOBOBJECT_BASIC_ACCOUNTING_INFORMATION**和一个**IO_COUNTERS**结构：

```
typedef struct _IO_COUNTERS {
    ULONGLONG ReadOperationCount;
    ULONGLONG WriteOperationCount;
    ULONGLONG OtherOperationCount;
    ULONGLONG ReadTransferCount;
    ULONGLONG WriteTransferCount;
    ULONGLONG OtherTransferCount;
} IO_COUNTERS, *PIO_COUNTERS;
```

这个结构指出已由作业中的进程执行过的读取、写入以及未读/未写操作的次数（以及这些操作期间传输的字节总数)。顺便说一句，可以使用**GetProcessIoCounters**函数来获得没有放入作业的那些进程的信息，如下所示：

```
BOOL GetProcessIoCounters(
    HANDLE hProcess,
    PIO_COUNTERS pIoCounters);
```

任何时候都可以调用**QueryInformationJobObject**，获得作业中当前正在运行的所有进程的进程ID集。为此，必须首先估算一下作业中有多少个进程，然后，分配一个足够大的内存块来容纳由这些进程ID构成的一个数组，另加一个**JOBOBJECT_BASIC_PROCESS_ID_LIST**结构的大小：

```
typedef struct _JOBOBJECT_BASIC_PROCESS_ID_LIST {
    DWORD NumberOfAssignedProcesses;
    DWORD NumberOfProcessIdsInList;
    DWORD ProcessIdList[1];
} JOBOBJECT_BASIC_PROCESS_ID_LIST, *PJOBOBJECT_BASIC_PROCESS_ID_LIST;
```

所以，为了获得作业中当前的进程ID集，必须执行以下类似的代码：

```
void EnumProcessIdsInJob(HANDLE hjob) {
// I assume that there will never be more
// than 10 processes in this job.
#define MAX_PROCESS_IDS 10
// Calculate the number of bytes needed for structure & process IDs.
DWORD cb = sizeof(JOBOBJECT_BASIC_PROCESS_ID_LIST) +
(MAX_PROCESS_IDS – 1) * sizeof(DWORD);
// Allocate the block of memory.
PJOBOBJECT_BASIC_PROCESS_ID_LIST pjobpil =
```

```
(PJOBOBJECT_BASIC_PROCESS_ID_LIST)_alloca(cb);
// Tell the function the maximum number of processes
// that we allocated space for.
pjobpil->NumberOfAssignedProcesses = MAX_PROCESS_IDS;
// Request the current set of process IDs.
QueryInformationJobObject(hjob, JobObjectBasicProcessIdList,
pjobpil, cb, &cb);
// Enumerate the process IDs.
for (DWORD x = 0; x < pjobpil->NumberOfProcessIdsInList; x++) {
// Use pjobpil->ProcessIdList[x]...
}
// Since _alloca was used to allocate the memory,
// we don't need to free it here.
}
```

这就是使用这些函数所能获得的所有信息。但是，操作系统实际保存着与作业相关的更多的信息。这是通过性能计数器来实现的，可以使用Performance Data Helper函数库（PDH.dll）中的函数来获取这些信息。还可以使用Reliability and Performance Monitor（“可靠性和性能监视器”，在“管理工具”中打开）来查看作业信息。但是，这样只能看到全局命名的作业对象。不过，利用Sysinternals的Process Explorer（*http://www.microsoft.com/technet/sysinternals/ProcessesAndThreads/ProcessExplorer.mspx*），可以很好地观察作业。默认情况下，作业限制下的所有进程都用棕色来突出显示。

这个软件一个更出色的设计是，对于这样的一个进程，其属性对话框的Job选项卡会列出作业名称及其限制（如果有的话），如图5-2所示。

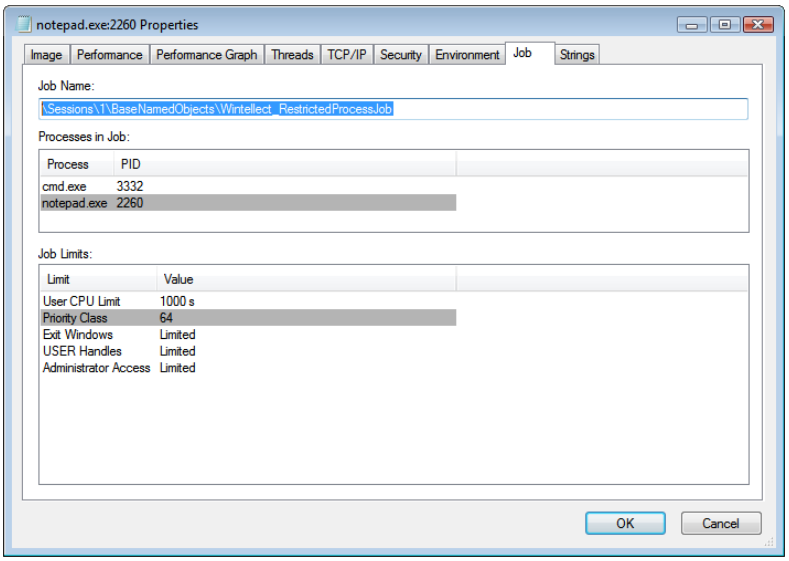


图5-2　Process Explorer的Job选项卡报告了详细的限制条件

警告
“User CPU Limit”显示有误，它的单位应该是毫秒，而不是秒。但这很快会在下一次更新中得以纠正。

5.4 作业通知

现在，你已经掌握了作业对象的基本知识，作业通知是我们最后要讨论的主题。例如，你是不是希望知道作业中的所有进程在何时终止执行，或者是否所有已分配的CPU时间已经到期？你是否想知道作业内部何时生成了一个新的进程，或者作业中的进程何时终止执行？如果不关心这些通知（大多数应用程序其实都是不关心的），那么就像前面描述的那样使用作业吧。如果关心这些事件通知，就还需要多学一些东西。

如果只是关心所有已分配的的CPU时间是否已经到期，那么可以非常简单地获得这个通知。作业中的进程如果尚未用完已分配的CPfU时间，作业对象就是nonsignaled（无信号）的。一旦用完所有已分配的CPU时间，Windows就会强行杀死作业中的所有进程，作业对象的状态会变成signaled（有信号）。通过调用**WaitForSingleObject**（或者一个类似的函数），可以轻松捕捉到这个事件。顺便提一句，可以调用**SetInformationJobObject**并授予作业更多的CPU时间，将作业对象重置为原来的nonsignaled状态。

我首次接触“作业”的时候，曾以为当作业对象中没有任何进程运行的时候，作业的状态就应该是signaled。毕竟，进程和线程对象在停止运行时都是singaled的；所以我认为，作业也应该在它停止运行时signaled。这样一来，就可以轻松判断作业什么时候结束运行。但是，Microsoft选择在已分配的CPU时间到期时，才将作业的状态变成signaled，因为那意味着一个错误条件（error condition）。在许多作业中，都会有一个父进程一直在运行，直至其所有子进程全部结束。所以，我们可以只等待父进程的句柄，借此得知整个作业何时结束。在我的**StartRestrictedProcess**函数中，展示了如何判断作业的已分配CPU时间已经到期，或者作业中的父进程何时终止。

前面描述了如何获得一些简单的通知，但没有解释如何获得一些更“高级”的通知（比如进程创建/终止运行）。要获得这些额外的通知，必须在自己的应用程序中建立更多的基础结构。具体来讲，你必须创建一个I/O完成端口（completion port）内核对象，并将自己的作业对象与完成端口关联。然后，必须有一个或者多个线程在完成端口上等待并处理作业通知。

一旦创建了I/O完成端口，就可以调用**SetInformationJobObject**将它与一个作业关联起来，如下所示：

```
JOBOBJECT_ASSOCIATE_COMPLETION_PORT joacp;
joacp.CompletionKey = 1; // Any value to uniquely identify this job
joacp.CompletionPort = hIOCP; // Handle of completion port that
                              // receives notifications
SetInformationJobObject(hJob, JobObjectAssociateCompletionPortInformation,
   &joacp, sizeof(jaocp));
```

执行上述代码后，系统将监视作业，只要有事件发生，就会把它们post到I/O完成端口（顺便提一句，你可以调用**QueryInformationJobObject**来获取completion key和完成端口句柄，但很少有必要这样做）。线程通过调用**GetQueuedCompletionStatus**来监视完成端口：

```
BOOL GetQueuedCompletionStatus(
    HANDLE hIOCP,
    PDWORD pNumBytesTransferred,
    PULONG_PTR pCompletionKey,
    POVERLAPPED *pOverlapped,
    DWORD dwMilliseconds);
```

当这个函数返回一个作业事件通知的时候，在**pCompletionKey**中，将包含completion key的值；这个值是在调用**SetInformationJobObject**将作业与完成端口关联时设置的。这样一来，就可以知道哪个作业有事件。**pNumBytesTransferred**的值指出具体发生了什么事件（参见表5-6）。根据事件，**pOverlapped**的值指出了一个对应的进程ID（而不是地址）。

表5-6　作业事件通知，系统可以将它们发送给与作业相关联的完成端口

事件	描述
JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO	作业中没有进程在运行时，就post通知
JOB_OBJECT_MSG_END_OF_PROCESS_TIME	进程已分配的CPU时间到期时，就post通知。进程将终止运行，并给出进程ID
JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT	试图超过作业中的活动进程数时，就post通知
JOB_OBJECT_MSG_PROCESS_MEMORY_LIMIT	进程试图提交的存储超过进程的限制时，就post通知。同时给出进程ID
JOB_OBJECT_MSG_JOB_MEMORY_LIMIT	进程提交的存储超过作业的限制时，就post通知。同时给出进程ID
JOB_OBJECT_MSG_NEW_PROCESS	一个进程添加到一个作业时，就post通知。同时给出进程ID
JOB_OBJECT_MSG_EXIT_PROCESS	一个进程终止运行时，就post通知。同时给出进程ID
JOB_OBJECT_MSG_ABNORMAL_EXIT_PROCESS	一个进程由于未处理的异常而终止运行时，就post通知。同时给出进程ID
JOB_OBJECT_MSG_END_OF_JOB_TIME	作业分配的CPU时间到期时，就post通知。但其中的进程不会自动终止。你可以允许进程继续运行，可以设置一个新的时间限制，也可以自己调用TerminateJobObject

最后要提醒你注意的是，在默认情况下，作业对象是这样配置的：当作业已分配的CPU时间到期时，它的所有进程都会自动终止，而且不会post **JOB_OBJECT_MSG_END_OF_JOB_TIME**这个通知。如果你想阻止作业对象杀死进程，只是简单地通知你CPU时间到期，就必须像下面这样执行代码：

```
// Create a JOBOBJECT_END_OF_JOB_TIME_INFORMATION structure
// and initialize its only member.
JOBOBJECT_END_OF_JOB_TIME_INFORMATION joeojti;
joeojti.EndOfJobTimeAction = JOB_OBJECT_POST_AT_END_OF_JOB;
// Tell the job object what we want it to do when the job time is
// exceeded.
SetInformationJobObject(hJob, JobObjectEndOfJobTimeInformation,
&joeojti, sizeof(joeojti));
```

针对EndOfJobTimeAction，你惟一能指定的另一个值就是**JOB_OBJECT_TERMINATE_AT_END_OF_JOB**。这是创建作业时的默认值。

5.6　Job Lab 示例程序

可以通过Job Lab应用程序（05-JobLab.exe）来轻松体验“作业”。此应用程序的源代码和资源文件都在05-JobLab目录中。启动该程序时，将出现如图5-3所示的窗口。

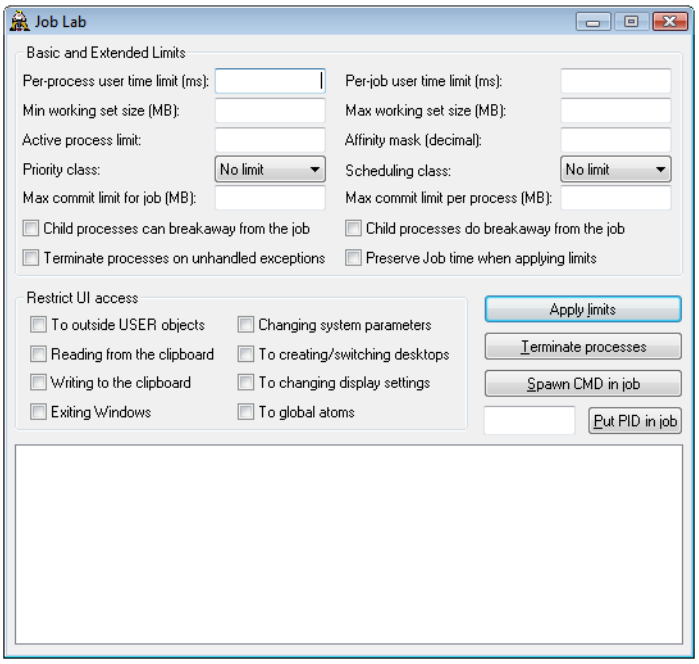


图5-3 Job Lab示例程序

进程初始化时，它会创建一个作业对象。这个作业对象命名为JobLab，便于你使用“性能监视器”MMC管理单元来查看它，并监视其性能。应用程序还创建了一个I/O完成端口，并把作业对象与它关联。这样一来，就可以监视来自作业的通知，并在窗口底部的列表框中显示。

最开始，作业中是没有进程的，也没有添加任何限制。顶部的框用于设置作业对象的基本和扩展限制。你只需用有效的值来填充这些框，并点击Apply limits按钮。如果有一个框留空不填，就不会应用那个限制。除了基本和扩展限制外，还可以打开和关闭各种UI限制。注意，Preserve Job Time When Applying Limits复选框不是用来设置限制的；它允许你在查询基本统计信息的时候更改作业的限制，同时不必重置ThisPeriodTotalUserTime和ThisPeriodTotalKernelTime成员。如果应用了一个per-job time limit，这个复选框会被禁用。利用其他按钮，你可以以其他方式操纵作业。Terminate Processes按钮用于杀死作业中的所有进程。Spawn CMD In Job按钮用于生成一个与此作业关联的CMD.exe（命令提示符）进程。在命令提示窗口中，你可以生成更多的子进程，并观察它们作为作业的一部分是如何工作的。出于实验性研究的目的，我认为这是一个非常有用的设计。最后一个按钮是Put PID In Job，将将一个现有的、没有与任何作业关联的进程与这个作业关联。

窗口底部的列表框用于显示最新的作业状态信息。每隔10秒，这个窗口就会显示基本和I/O统计信息以及进程/作业内存使用峰值。作业中的每个进程的进程ID和完整路径名也会显示出来。

警告

使用psapi.h中的函数（比如GetModuleFileNameEx和GetProcessImageFileName），可以根据进程ID来获得进程的完整路径名称。但是，当作业收到通知，知道一个新进程在它的限制下创建时，前一个函数调用会失败。因为在这个时候，地址空间尚未完全初始化：模块尚未与它建立映射。GetProcessImageFileName则很有意思，因为即使在那种极端情况下，它也能获取完整路径名。但是，路径名的格式近似于内核模式（而不是用户模式）中看到的格式。例如，是\Device\HarddiskVolume1\Windows\System32\notepad.exe，而不是C:\Windows\System32\notepad.exe。这就是你应该依赖于新的QueryFullProcessImageName函数的

原因。该函数在任何情况下都会返回你预期的完整路径名。

除了所有这些统计信息，列表框还显示了从作业到应用程序I/O完成端口的所有通知。只要有通知被发送到这个列表框，就会立即更新当时的状态信息

最后提醒一句，如果修改了源代码，并创建了一个没有命名的作业内核对象，你可以运行此应用程序的多个副本，在同一台计算机上创建两个甚至更多的作业对象，并进行更多的试验。

就源代码本身而言，这里没有任何特别的地方值得讨论，因为源代码中已经有很好的注释。不过，我的确创建了一个Job.h文件，它定义了一个CJob C++类，此类封装了操作系统的作业对象。这样一来，我的编程就变得更简单了，因为不需要到处传递作业的句柄。同时，这个类还减少了类型转换的工作量。调用**QueryInformationJobObject**和**SetInformationJobObject**函数时，这样的事情本来是不可避免的。

第 6 章　线程基础

本章内容包括
- 何时创建线程
- 何时不应该创建线程
- 编写第一个线程函数
- CreateThread 函数
- 终止运行线程
- 线程内幕
- C/C++运行库注意事项
- 了解自己的身份

理解线程是至关重要的，因为每个进程至少都有一个线程。在本章，我们将讲述线程更多的细节。具体地说，我们将解释线程和进程有何区别，它们各自有何职责。同时，还要解释系统如何使用线程内核对象来管理线程。就像进程内核对象一样，线程内核对象也拥有属性，我们将探讨用于查询和更改这些属性的函数。此外，还要介绍可在进程中创建和生成更多线程的函数。

在第 4 章，我们讨论了进程实际是由两个组件组成的：一个进程内核对象和一个地址空间。类似地，线程也由两个组件组成：
- 一个是线程的内核对象，操作系统用它管理线程。内核对象还是系统用来存放线程统计信息的地方。
- 一个线程堆栈，用于维护线程执行时所需的所有函数参数和局部变量。（在第 16 章中，我将详细讨论系统如何管理线程的堆栈。）

第 4 章讲过，进程是有惰性的。进程从来不执行任何东西，它只是一个线程的容器。线程必然是在某个进程的上下文中创建的，而且会在这个进程内部“终其一生”。这意味着线程要在其进程的地址空间内执行代码和处理数据。所以，假如一个进程上下文中有两个以上的进程运行，这些线程将共享同一个地址空间。这些线程可以执行同样的代码，可以处理相同的数据。此外，这些线程还共享内核对象句柄，因为句柄表是针对每一个进程的，而不是针对每一个线程。

可以看出，相较于线程，进程所使用的系统资源更多。其原因在于地址空间。为一个进程创建一个虚拟的地址空间需要大量系统资源。系统中会发生大量的记录活动，而这需要用到大量内存。而且，由于.exe和.dll文件要加载到一个地址空间，所以还需要用到文件资源。另一方面，线程使用的系统资源要少得多。事实上，线程只有一个内核对象和一个堆栈；几乎不涉及记录活动，所以不需要占用多少内存。

由于线程需要的开销比进程少，所以建议你尽量使用额外的线程来解决你的编程问题，避免创建新进程。但是，也不要把这个建议当作金科玉律。许多设计更适合用多个进程来实现。应该知道如何权衡利弊，让经验来指导你进行编程。

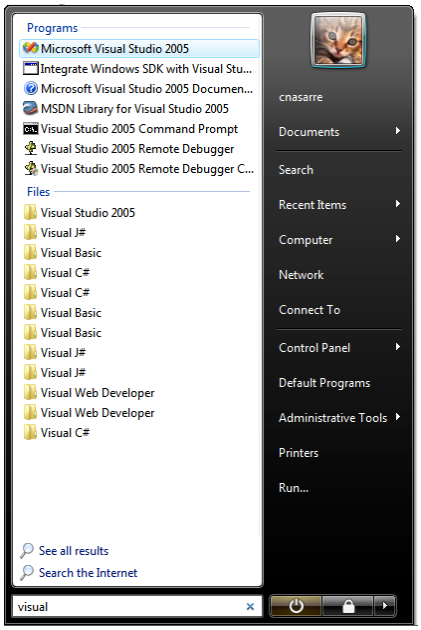
在深入讨论线程之前，稍微花一点时间来讨论如何在应用程序的构架中正确地使用线程。

6.1 何时创建线程

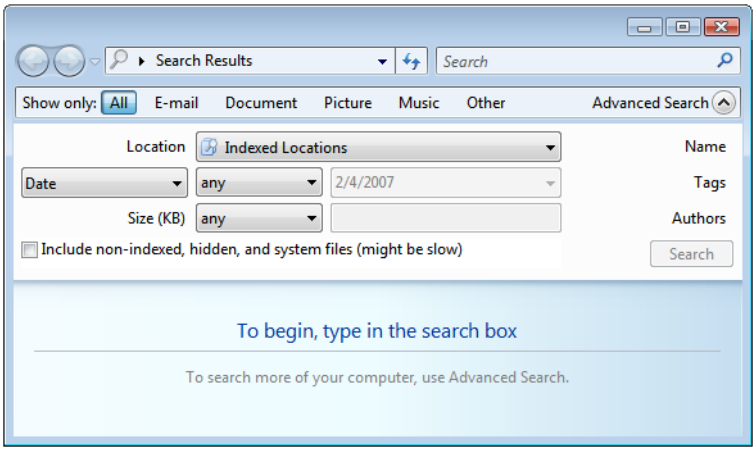
线程描述了进程内部的一个执行路径。每次初始化进程时，系统都会创建一个主线程。对于用Microsoft C/C++编译器生成的应用程序，这个线程首先会执行C/C++运行库的启动代码，后者调用入口函数(**_tmain**或**_tWinMain**)，并继续执行，直至入口函数返回C/C++运行库的启动代码，后者最终将调用**ExitProcess**。对于许多应用程序来说，这个主线程是应用程序惟一需要的线程。但是，进程也可以创建额外的线程来帮助它们完成自己的工作。

每个计算机都有一个特别强大的资源： CPU。让CPU空闲着是没有任何道理的（忽略省电和发热问题）。为了让CPU保持“忙碌”，我们为它指定了各种各样的任务让它执行。下面列举了其中的少数几个例子：

- 操作系统的Windows Indexing Services（Windows索引服务）创建了一个低优先级的线程，此线程定期醒来，并对硬盘上的特定区域的文件内容进行索引。Windows索引服务极大改进了性能，因为一旦成功建立索引，就不必在每次搜索时都打开、扫描和关闭硬盘上的每一个文件。配合这种索引服务，Microsoft Windows Vista提供了一套高级的搜索功能。可以通过两种方式寻找文件。第一，单击“开始”按钮，在底部的搜索框中输入。在左侧的列表中，将根据索引显示与输入的文本匹配的程序、文件和文件夹。如下所示。



第二，可以调用“搜索”窗口（右击“开始”按钮，从弹出菜单中选择“搜索”）。然后，请在“搜索”文本框中输入搜索条件。只有选择了“Location”组合框中的“Indexed Locations”（这是默认设置），才会在索引位置中搜索，如下所示。



- 可以使用操作系统附带的磁盘碎片整理程序。通常，这类实用程序有许多普通用户无法理解的管理选项，比如程序多久运行一次，在什么时间运行等等。如果使用低优先级的线程，可以在系统空闲的时候，在后台运行这个实用程序并进行磁盘碎片整理。
- 只要暂停输入，Microsoft Visual Studio IDE就会自动编译C#和Microsoft Visual Basic .NET源代码文件。在编辑窗口中，无效的表达式将用下划线标识，鼠标滑过这些表达式时，会显示相应的警告和错误信息。
- 电子表格软件可以在后台执行重新计算。
- 字处理软件可以在后台重新分页、拼写检查、语法检查以及打印。
- 文件可以在后台拷贝到其他存储介质。
- Web浏览器可以在后台与其服务器进行通信。在当前网站的结果显示出来之前，用户可以调整浏览器窗口的大小，或者转到其他网站。

对于这些例子，你应该注意一个重点，即多线程简化了应用程序的用户界面的设计。例如，如果你一结束输入，编译器就能开始生成你的应用程序，是不是就不必提供Build（生成）菜单项了呢？又例如，既然始终能在后台执行拼写检查和语法检查，字处理器应用程序是不是就不必提供相应的菜单项了呢？

在Web浏览器的例子中，由于为输入和输出（可能是网络、文件或其他方面的输入和输出）使用了一个单独的线程，所以应用程序的用户界面可以一直保持可响应的状态。来设想这样的一个应用程序，它用于对数据库的记录进行排序，打印文档，或者拷贝文件。通过为这种严重依赖于输入/输出的任务使用一个单独的线程，当操作进行时，用户可以随时利用应用程序的界面来取消操作。

将一个应用程序设计成多线程的，可以使该应用程序更易于扩展。如下一章所述，每个线程都被分配了一个CPU。所以，如果我计算机有两个CPU，而且应用程序有两个线程，那么两个CPU都会很忙。其结果就是，只需花一个任务的时间，两个任务都可以完成。

每个进程内部至少有一个线程。所以，假如不在应用程序中做一些特别的事情，就已经能从多线程操作系统获益。例如，你可以同时生成应用程序和使用字处理软件（我经常这样做）。如果计算机有两个CPU，那么生成过程将在一个处理器上进行，另一个处理器则负责处理文档。换言之，用户感觉不到性能有所下降，而且在他进行输入的时候，用户界面也不会出现经常失去响应的问题。此外，如果编译器的一个bug导致其线程进入无限循环，其他进程仍可运行。(16位Windows和MS-DOS应用程序就不是这样的了。)

6.2 何时不应该创建线程

到目前为止，我一直再为多线程应用程序高唱赞歌。尽管多线程应用程序好处多多，但仍有一些不足之处。有些开发人员认为，**任何**问题都可以通过把它分解成线程来解决。但是，这样想是大错特错的！

线程相当有用，而且占有重要地位，但在使用线程时，可能会在尝试解决旧问题时产生新问题。例如，假定现在要开发一个字处理程序，并且希望允许打印函数在它自己的线程中运行。这听起来不错，因为只要开始打印文档，用户就可以立即返回并开始编辑文档。但等等，这意味着在打印文档期间，文档中的数据可能已经发生了改变。那么，也许最好的办法是不让打印函数在自己的线程中运行？但这个“解决方案”似乎又有点儿极端了。假如让用户编辑另一个文档但锁定正在打印的文档，使其在打印完成之后再进行修改，又如何呢？或者说还有第三个办法：将文档复制到一个临时文件中，打印临时文件的内容，并允许用户修改原件。临时文件打印完成后，删除临时文件。

由此可见，线程能解决一些问题，但又会产生新的问题。开发应用程序的用户界面时，可能遇到对线程的另一种误用。在几乎所有应用程序中，所有用户界面组件（窗口）都应该共享同一个线程。一个窗口的所有子窗口无疑应该由一个线程来创建。有时，也许需要在不同的线程上创建不同的窗口，但这类情形相当少见。

通常，应用程序应该有一个用户界面线程，此线程负责创建所有窗口，另外还有一个**GetMessage**循环。进程中的其他所有线程都是受计算机制约或者受I/O（输出/输出）限制的工作线程，这些线程永远不会创建窗口。另外，用户界面线程的优先级通常高于工作线程。这样一来，用户界面才能灵敏地响应用户的操作。

尽管很少需要在一个进程中包含多个用户界面线程，但有时确实需要这样做。例如，Windows资源管理器就为每个文件夹的窗口创建了一个独立的线程。这样一来，就可以把文件从一个文件夹复制到另一个文件夹，同时仍然可以查看系统上的其他文件夹。另外，如果Windows资源管理器出现一个bug，正在操纵一个文件夹的线程就可能崩溃，但我们仍然可以操纵其他文件夹——至少在干了导致其他计算机也崩溃的事情之前。

总之，你应该合理使用多线程。不要因为一件事情是你能做的就非要去做。仅仅使用分配给进程的主线程，就能写出许多有用而且强大的应用程序。

6.3 编写第一个线程函数

每个线程都必须有一个入口函数，这是线程执行的起点。前面已讨论过主线程的入口函数：**_tmain**或**_tWinMain**。如果想在进程中创建辅助线程，它必须有自己的入口函数，形式如下：

```
DWORD WINAPI ThreadFunc(PVOID pvParam){
   DWORD dwResult = 0;
   ...
   return(dwResult);
}
```

线程函数可以执行我们希望它执行的任何任务。最终，线程函数将终止运行并返回。此时，线程将终止运行，用于线程堆栈的内存也会被释放，线程内核对象的使用计数也会递减。如果使用计数变成0，线程内核对象会被销毁。类似于进程内核对象，线程内核对象的寿命至少可以达到它们相关联的线程那样长。不过，对象的寿命可能超过线程本身的寿命。

关于线程函数，有几点要注意：

- 主线程的入口函数默认必定命名为**main**，**wmain**，**WinMain**或**wWinMain**（有一种情况除外，也就是使用**/ENTRY:**链接器选项来指定另一个函数作为入口函数）。和它不同的是，线程函数可以任意命名。事实上，如果应用程序中有多个线程函数，必须为它们指定不同的名称，否则编译器/链接器会认为你创建了一个函数的多个实现。
- 由于向主线程的入口函数传递了字符串参数，所以可以使用入口函数的ANSI/Unicode版本：**main/wmain**和**WinMain/wWinMain**。相反，向线程函数传递的是一个意义由我们（而非操作系统）来定义的参数。因此，不必担心ANSI/Unicode问题。
- 你的线程函数必须返回一个值，它会成为该线程的退出代码。这类似于C/C++运行库的策略：令主线程的退出代码成为进程的退出代码。
- 你的线程函数（实际就是你的所有函数）应该尽可能使用函数参数和局部变量。使用静态和全局变量时，多个线程可以同时访问这些变量，这样可能会破坏变量中保存的内容。然而，参数和局部变量是在线程堆栈上创建的。因此，不太可能被其他线程破坏。

在知道了如何实现线程函数之后，下面要讨论如何让操作系统实际创建一个线程来执行你的线程函数。

6.4 CreateThread 函数

我们已经讨论了在调用**CreateProcess**时，进程的主线程是如何产生的。如果想创建一个或

多个辅助线程，只需让一个正在运行的线程调用**CreateThread**：

```
HANDLE CreateThread(
   PSECURITY_ATTRIBUTES psa,
   DWORD cbStackSize,
   PTHREAD_START_ROUTINE pfnStartAddr,
   PVOID pvParam,
   DWORD dwCreateFlags,
   PDWORD pdwThreadID);
```

调用**CreateThread**时，系统会创建一个线程内核对象。这个线程内核对象不是线程本身，而是一个较小的数据结构，操作系统用这个结构来管理线程。可以把线程内核对象想象为一个由线程统计信息构成的小型数据结构。这与进程和进程内核对象之间的关系是相同的

系统将进程地址空间的内存分配给线程堆栈使用。新线程在与负责创建的那个线程相同的进程上下文中运行。因此，新线程可以访问进程内核对象的所有句柄、进程中的所有内存以及同一个进程中其他所有线程的堆栈。这样一来，同一个进程中的多个进程可以很容易地互相通信。

> 注意
> CreateThread函数是用于创建线程的Windows函数。不过，如果写的是C/C++代码，就绝对不要调用CreateThread。相反，正确的选择是使用Microsoft C++运行库函数_beginthreadex。如果使用的不是Microsoft C++编译器，你的编译器的提供商应该提供类似的函数来替代CreateThread。不管这个替代函数是什么，都必须使用它。本章稍后将解释_beginthreadex函数的用途及其重要性。

好了，大致情况就介绍到这里，下面将详细解释**CreateThread**的每一个参数。

6.4.1 psa

psa参数是指向**SECURITY_ATTRIBUTES**结构的一个指针。如果想使用线程内核对象的默认安全属性，可以向此参数传入**NULL**（一般都会这样做）。如果希望所有子进程都能继承到这个线程对象的句柄，必须指定一个**SECURITY_ATTRIBUTES**结构，该结构的**bInheritHandle**成员初始化为**TRUE**。详情参见第3章。

6.4.2 cbStackSize

cbStackSize参数指定线程可以为其线程堆栈使用多少地址空间。每个线程都拥有自己的堆栈。当**CreateProcess**函数开始一个进程的时候，它会在内部调用**CreateThread**来初始化进程的主线程。对于**cbStackSize**参数，**CreateProcess**使用了保存在可执行文件内部的一个值。可以使用链接器的**/STACK**开关来控制这个值，如下所示：

```
/STACK:[reserve] [,commit]
```

reserve参数用于设置系统将为线程堆栈预留多少地址空间，默认是1 MB（在Itanium芯片组上，默认大小为 4 MB）。**commit**参数指定最初应为堆栈的保留区域提交多少物理存储空间，默认是1页。随着线程中的代码开始执行，需要的存储空间可能不止1页。如果线程溢出它的堆栈，会产生异常。(有关线程堆栈和堆栈溢出异常的详情，请参见第16章。有关常见异常处理的详情，请参见第23章。)系统将捕获这种异常，并将另一个页（或者为commit参数指定的任何大小）提交给保留空间。这样一来，线程堆栈就可以根据需要动态地增大。

调用**CreateThread**时，如果传入非0值，函数会保留并提交（commit）线程堆栈的所有存储。由于所有存储都已经事先提交，所以可以保证线程有指定的堆栈存储可用。保留空间的大小要么

由/STACK链接器开关来指定，要么由**cbStack**的值来指定，取其中较大的那一个。提交的存储空间大小与传递的csStack参数值匹配。如果为cbStack参数传入0值，CreateThread函数就会保留一个区域，并提交由/STACK链接器开关指定的存储量（这个值由链接器嵌入.exe文件中）。

保留空间的容量设定了堆栈空间的上限，这样才能捕获代码中的无穷递归bug。例如，假设你写了一个函数以递归方式调用其自身。而且这个函数存在一个bug，会导致无穷递归。每次此函数调用自身时，都会在内存堆栈上创建一个新的堆栈帧。如果系统没有设定堆栈空间的上限，这个递归调用的函数就永远不会终止调用自身。进程的所有地址空间都会被分配出去，大量物理存储会提交给堆栈。通过设置堆栈空间的上限，可以防止应用程序耗尽物理内存区域，而且还可以尽早察觉程序中的bug（第16章的Summation示例程序展示了如何捕获和处理堆栈溢出）。

6.4.2 pfnStartAddr 和 pvParam

pfnStartAddr参数指定你希望新线程执行的线程函数的地址。线程函数的**pvParam**参数与最初传给CreateThread函数的**pvParam**参数是一样的。**CreateThread**不用这个参数做别的事情，只是在线程开始执行时将其传给线程函数。通过这个参数，可以将一个初始值传给线程函数。这个初始值可以是一个数值，也可以是指向一个数据结构（其中包含额外的信息）的指针。

创建多个线程时，可以让它们使用同一个函数地址作为起点。这样做完全合法，而且非常有用。例如，我们可以这样实现一个Web服务器，令其创建一个新的线程来处理每个客户端的请求。每个线程都知道自己正在处理哪个客户端的请求，因为在创建每个线程的时候，都向其传递了不同的**pvParam**值。

记住，Windows是一个抢占式的多线程系统（preemptive multithreading system）。这意味着新的线程和调用CreateThread函数的线程可以同时执行。因为两个线程是同时运行的，所以可能出现问题。来观察如下所示的代码：

```
DWORD WINAPI FirstThread(PVOID pvParam) {
   // Initialize a stack-based variable
   int x = 0;
   DWORD dwThreadID;

   // Create a new thread.
   HANDLE hThread = CreateThread(NULL, 0, SecondThread, (PVOID) &x,
      0, &dwThreadID);

   // We don't reference the new thread anymore,
   // so close our handle to it.
   CloseHandle(hThread);

   // Our thread is done.
   // BUG: our stack will be destroyed, but
   // SecondThread might try to access it.
   return(0);
}

DWORD WINAPI SecondThread(PVOID pvParam) {
   // Do some lengthy processing here. ...
   // Attempt to access the variable on FirstThread's stack.
   // NOTE: This may cause an access violation – it depends on timing!
   * ((int *) pvParam) = 5; ... return(0);
}
```

在上述代码中，**FirstThread**可能会在SecondThread函数将5赋给FirstThread函数的*x*之前完成任务。如果发生这种情况，**SecondThread**不知道**FirstThread**已经不存在了，所以会试图更改现已无效的一个地址的内容。这会导致**SecondThread**产生访问冲突，因为**FirstThread**的堆栈已在

FirstThread终止运行的时候被销毁。解决这个问题的一种方法是将*x*声明为一个静态变量，使编译器能在应用程序的数据section（而不是线程堆栈）中为*x*创建一个存储区域。

但是，这会使函数不可重入。换言之，不能创建两个线程来执行同一个函数，因为这两个线程将共享同一个静态变量。为了解决这个问题（以及它的其他更复杂的变化形式），另一个办法是使用正确的线程同步技术（详情参见第8章和第9章）。

6.4.3 dwCreateFlags

dwCreateFlags参数指定额外的标志来控制线程的创建。它可以是两个值之一。如果值为0，线程创建之后立即就可以进行调度。如果值为**CREATE_SUSPENDED**，系统将创建并初始化线程，但是会暂停该线程的运行，这样它就无法进行调度。

在线程有机会执行执行任何代码之前，应用程序可以利用**CREATE_SUSPENDED**标志来更改线程的一些属性。由于很少有必要这样做，所以该标志并不常用。第5章最后一节的Job Lab应用程序演示了该标志的正确用法。

.

pdwThreadID

CreateThread函数的最后一个参数是**pdwThreadID**，它必须是**DWORD**的一个有效地址。**CreateThread**函数用它来存储系统分配给新线程的ID（进程和线程ID的详情参见第4章）。你可以为这个参数传递NULL（一般都是这样做的），它告诉函数你对线程ID没有兴趣。

6.5 终止运行线程

线程可以通过以下4种方法来终止运行。

- 线程函数返回（这是强烈推荐的）。
- 线程通过调用**ExitThread**函数“杀死”自己（要避免使用这种方法）。
- 同一个进程或另一个进程中的线程调用**TerminateThread**函数（要避免使用这种方法）。
- 包含线程的进程终止运行（这种方法避免使用）。

下面将讨论终止线程运行的这4种方法，并描述线程终止运行时会有哪些情况发生。

6.5.1 线程函数返回

设计线程函数时，应该确保当你希望线程终止运行时，就让它们返回。这是保证线程的所有资源都被正确清理的惟一方式。

让线程函数返回，可以确保以下正确的应用程序清理工作都得以执行：

- 线程函数中创建的所有C++对象都通过其析构函数被正确销毁。
- 操作系统正确释放线程堆栈使用的内存。
- 操作系统把线程的退出代码（在线程的内核对象中维护）设为线程函数的返回值。
- 系统递减少线程的内核对象的使用计数。

6.5.2 ExitThread 函数

为了强迫线程终止运行，可以让它调用**ExitThread**：

```
VOID ExitThread(DWORD dwExitCode);
```

该函数将终止线程的运行，并导致操作系统清理该线程使用的所有操作系统资源。但是，你的C/C++资源（如C++类对象）不会被销毁。有鉴于此，更好的做法是直接从线程函数返回，不要自己调用**ExitThread**函数（详情参见第4章的“终止进程”一节的“ExitProcess函数”小节)。

当然，可以使用**ExitThread**的**dwExitCode**参数来告诉系统将线程的退出代码设为什么。**ExitThread**函数没有返回值，因为线程已终止，而且不能执行更多的代码。

> 注意
> 终止线程运行的推荐方法是让它的线程函数返回 (如上一节所述)。但是，如果使用本节描述的方法，务必注意ExitThread函数是用于“杀死”线程的Windows函数。如果你要写C/C++代码，就绝对不要调用ExitThread。相反，应该使用C++运行库函数_endthreadex。如果使用的不是Microsoft的C++编译器，那么你的编译器提供方应该提供它们自己的ExitThread替代函数。不管这个替代函数是什么，都必须使用它。本章稍后将具体解释_endthreadex的用途及其重要性。

6.5.3 TerminateThread 函数

调用**TerminateThread**函数也可以杀死一个线程，如下所示：

```
BOOL TerminateThread(
  HANDLE hThread,
  DWORD dwExitCode);
```

不同于**ExitThread**总是“杀死”主调线程（calling thread），**TerminateThread**能“杀死”任何线程。**hThread**参数标识了要终止的那个线程的句柄。线程终止运行时，其退出代码将变成你作为**dwExitCode**参数传递的值。同时，线程的内核对象的使用计数会递减。

> 注意
> TerminateThread函数是异步的。也就是说，它告诉系统你想终止线程，但在函数返回时，并不保证线程已经终止了。如果需要确定线程已终止运行，还需要调用WaitForSingleObject（详情参见第9章）或类似的函数，并向其传递线程的句柄。

一个设计良好的应用程序决不会使用这个函数，因为被终止运行的线程收不到它被“杀死”的通知。线程无法正确清理，而且不能阻止自己被终止运行。

> 注意
> 如果通过返回或调用ExitThread函数的方式来终止一个线程的运行，该线程的堆栈也会被销毁。但是，如果使用的是TerminateThread，那么除非拥有此线程的进程终止运行，否则系统不会销毁这个线程的堆栈。Microsoft故意以这种方式来实现TerminateThread。否则，假如其他还在运行的线程要引用被“杀死”的那个线程的堆栈上的值，就会引起访问冲突。让被“杀死”的线程的堆栈保留在内存中，其他的线程就可以继续正常运行。
>
> 此外，动态链接库（DLL）通常会在线程终止运行时收到通知。不过，如果线程是用TerminateThread强行“杀死”的，则DLL不会收到这个通知，其结果是不能执行正常的清理工作（详情请参见第20章）。

6.5.4 当进程终止运行时

第4章介绍的ExitProcess和TerminateProcess函数也可用于终止线程的运行。区别在于，这些函数会使终止运行的进程中的所有线程全部终止。同时，由于整个进程都会关闭，所以它所使用的所有资源肯定都会被清理。其中必然包括所有线程的堆栈。这两个函数会导致进程中剩余的所有线程被强行“杀死”，这就好象是我们为剩余的每个线程都调用了**TerminateThread**。显然，这意味着正确的应用程序清理工作不会执行：C++对象的析构函数不会被调用，数据不会回写到磁盘……等等。正如我在本章开始就解释的一样，当应用程序的入口函数返回时，C/C+运行库的启动代码将调用**ExitProcess**。因此，如果应用程序中并发运行着多个线程，你需要在主线程返回之前，显式地处理好每个线程的终止过程。否则，其他所有正在运行的线程都会在毫无预警的前提下突然“死亡”。

6.5.5 当线程终止运行时

线程终止运行时，会发生下面这些事情：

- 线程拥有的所有用户对象句柄会被释放。在Windows中，大多数对象都是由包含了“创建这些对象的线程”的进程拥有的。但是，一个线程有两个User对象：窗口（window）和挂钩（hook）。一个线程终止运行时，系统会自动销毁由线程创建或安装的任何窗口，并卸载由线程创建或安装的任何挂钩。其他对象只有在拥有线程的进程终止时才被销毁。
- 线程的退出代码从**STILL_ACTIVE**变成传给**ExitThread**或**TerminateThread**的代码。
- 线程内核对象的状态变为signaled。
- 如果线程是进程中的最后一个活动线程，系统认为进程也终止了。
- 线程内核对象的使用计数递减1。

线程终止运行时，其关联的线程对象不会自动释放，除非对这个对象的所有未结束的引用都被关闭了。

一旦线程不再运行，系统中就没有别的线程可以处理该线程的句柄。但是，其他线程可以调用**GetExitCodeThread**来检查**hThread**所标识的那个线程是否已终止运行；如果已终止运行，就判断其退出代码是什么：

```
BOOL GetExitCodeThread(
  HANDLE hThread,
  PDWORD pdwExitCode);
```

退出代码的值通过**pdwExitCode**指向的**DWORD**来返回。如果在调用GetExitCodeThread时，线程尚未终止，函数就用**STILL_ACTIVE**标识符（被定义为0x103）来填充**DWORD**。如果函数调用成功，就返回**TRUE**（要想进一步了解如何使用线程的句柄来判断线程是在什么时候终止运行的，请参见第9章）。

6.6 线程内幕

前面解释了如何实现一个线程函数，以及如何让系统创建一个线程来执行此线程函数。在本小节，我们要研究系统具体是如何实现这一点的。图6-1展示了系统是通过哪些必不可少的步骤来创建和初始化一个线程的。

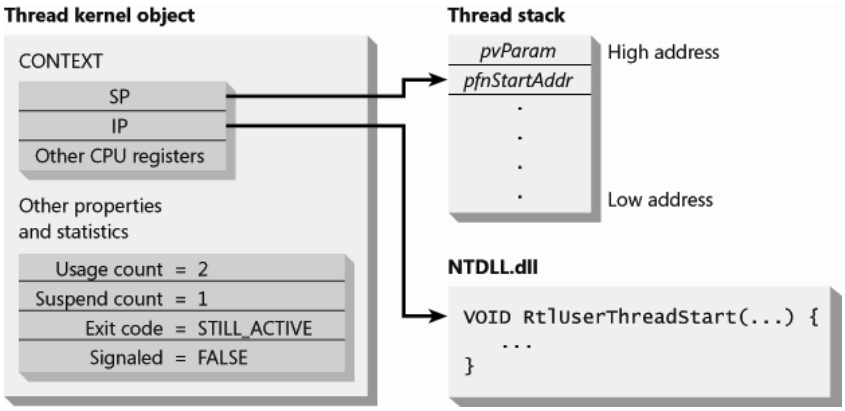


图6-1　如何创建和初始化一个线程

仔细看看这幅图，了解究竟发生了什么。对**CreateThread**函数的一个调用导致系统创建一个线程内核对象。该对象最初的使用计数为2。(除非线程终止，而且从CreateThread返回的句柄关闭，否则线程内核对象不会被销毁。)该线程内核对象的其他属性也被初始化：暂停计数被设为1，退出代码被设为**STILL_ACTIVE** (0x103)，而且对象被设为nonsignaled状态。

一旦创建了内核对象，系统就分配内存，供线程的堆栈使用。此内存是从进程的地址空间内分配的，因为线程没有自己的地址空间。然后，系统将两个值写入新线程堆栈的最上端。(线程堆栈始终是从高位内存地址向低位内存地址构建的。)写入线程堆栈的第一个值是传给CreateThread函数的**pvParam**参数的值。紧接在它下方的是传给CreateThread函数的**pfnStartAddr**值。

每个线程都有其自己的一组CPU寄存器，称为线程的上下文(context)。上下文反映了当线程上一次执行时，线程的CPU寄存器的状态。线程的CPU寄存器全部保存在一个CONTEXT结构（在WinNT.h头文件中定义）。**CONTEXT**结构本身保存在线程内核对象中。

指令指针和栈指针寄存器是线程上下文中最重要的两个寄存器。记住，线程始终在进程的上下文中运行。所以，这两个地址都标识了“拥有线程的那个进程”的地址空间中的内存。当线程的内核对象被初始化的时候，**CONTEXT**结构的堆栈指针寄存器被设为**pfnStartAddr**在线程堆栈中的地址。而指令指针寄存器被设为**RtlUserThreadStart**函数（该函数未见于正式文档）的地址，此函数是NTDLL.dll模块导出的。图6-1对此进行了演示。

RtlUserThreadStart函数的基本用法如下：

```
VOID RtlUserThreadStart(PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam) {
   __try {
      ExitThread((pfnStartAddr)(pvParam));
   }
   __except(UnhandledExceptionFilter(GetExceptionInformation())) {
      ExitProcess(GetExceptionCode());
   }
   // NOTE: We never get here.
}
```

线程完全初始化好之后，系统将检查**CREATE_SUSPENDED**标志是否传给**CreateThread**函数。如果此标记没有传递，系统将线程的暂停计数递增至0；随后，线程就可以调度给一个处理器去执行。然后，系统在实际的CPU寄存器中加载上一次在线程上下文中保存的值。现在，线程可以在其进程的地址空间中执行代码并处理数据了。

因为新线程的指令指针被设为**RtlUserThreadStart**，所以这个函数实际就是线程开始执行的地方。观察**RtlUserThreadStart**的原型，你会以为它接收了两个参数，但这就暗示着该函数是从另一个函数调用的，而实情并非如此。新线程只是在此处产生并且开始执行。之所以能能访问这两个参数，是由于操作系统将值显式地写入线程堆栈（参数通常就是这样传给函数的）。注意，有的CPU构架使用CPU寄存器而不是堆栈来传递参数。对于这种构架，系统会在允许线程执行**RtlUserThreadStart**函数之前正确初始化恰当的寄存器。

新线程执行**RtlUserThreadStart**函数的时候，将发生以下事情。

- 围绕你的线程函数，会设置一个结构化异常处理（Structured Exception Handling，SEH）帧。这样一来，线程执行期间所产生的任何异常都能得到系统的默认处理。(有关结构化异常处理的详情，请参见第23章、第24章和第25章)
- 系统调用你的线程函数，把你传给CreateThread函数的**pvParam**参数传给它。
- 线程函数返回时，**RtlUserThreadStart**调用**ExitThread**，将你的线程函数的返回值传给它。线程内核对象的使用计数递减，而后线程停止执行。
- 如果线程产生了一个未被处理的异常，**RtlUserThreadStart**函数所设置的SEH帧会处理这个异常。通常，这意味着会向用户显示一个消息框，而且当用户关闭此消息框时，**RtlUserThreadStart**会调用**ExitProcess**来终止整个进程，而不只是终止有问题的线程。

注意，在**RtlUserThreadStart**内，线程会调用**ExitThread**或者**ExitProcess**。这意味着线程永远不能退出此函数；它始终在其内部“消亡”。这就是为什么**RtlUserThreadStart**函数被原型化为返回VOID的原因，它永远都不会返回。

此外，因为有了**RtlUserThreadStart**函数，所以线程函数可以在完成它的工作之后返回。当**RtlUserThreadStart**调用你的线程函数时，它会将线程函数的返回地址压入堆栈，使线程函数知道应在何处返回。但是，**RtlUserThreadStart**函数是不允许返回的。如果它没有在强行杀死线程的前提下尝试返回，几乎肯定会引起访问冲突，因为线程堆栈上没有函数返回地址，**RtlUserThreadStart**将尝试返回某个随机的内存位置。

一个进程的主线程初始化时，其指令指针会被设为同一个未文档化的函数**RtlUserThreadStart**。

当**RtlUserThreadStart**开始执行时，它会调用C/C++运行库的启动代码，后者初始化继而调用你的**_tmain**或**_tWinMain**函数。你的入口函数返回时，C/C++运行时启动代码会调用**ExitProcess**。所以对于C/C++应用程序来说，主线程永远不会返回到**RtlUserThreadStart**函数。

6.7 C/C++运行库注意事项

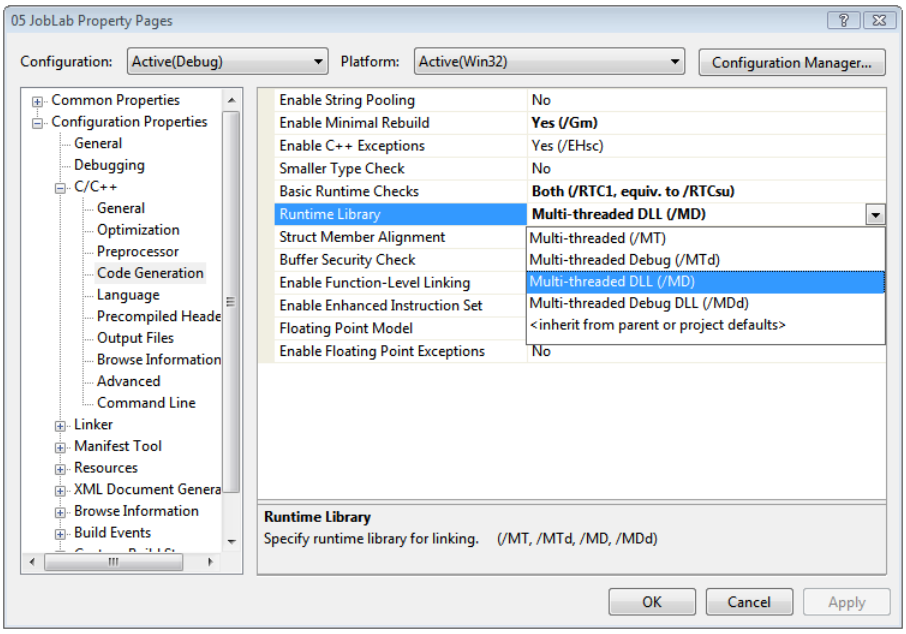
Visual Studio附带了4个原生的C/C++运行库，还有2个库面向Microsoft .NET的托管环境。注意，所有这些库都支持多线程开发：不再有单独的一个C/C++库专门针对单线程开发。表6-1对这些库进行了描述。

表6-1 Microsoft Visual Studio附带的C/C++库

库名称	描述
LibCMt.lib	库的静态链接Release版本
LibCMtD.lib	库的静态链接Debug版本
MSVCRt.lib	导入库，用于动态链接MSVCR80.dll 库的Release版本。 (这是新建项目时的默认库)
MSVCRtD.lib	导入库，用于动态链接MSVCR80D.dll库的Debug版本
MSVCMRt.lib	导入库，用于托管/原生代码混合

MSVCURt.lib	导入库，编译成百分之百纯MSIL代码

实现任何类型的项目时，都必须先知道这个项目要链接哪个库。可以通过如下所示的项目属性对话框来选择库。打开Configuration Properties，C/C++，Code Generation。在Runtime Library类别中，从Use Run-Time Library组合框中选择这4个可用的选项之一。



让我们简单回顾一下历史。很早以前，是一个库用于单线程应用程序，另一个库用于多线程应用程序。之所以采用这个设计，是由于标准C运行库是在1970年左右发明的。要在很久很久之后，才会在操作系统上出现线程的概念。标准C运行库的发明者根本没有考虑到为多线程应用程序使用C运行库的问题。让我们用一个例子来了解可能遇到的问题。

以标准C运行库的全局变量**errno**为例。有的函数会在出错时设置该变量。假定现在有这样的一个代码段：

```
BOOL fFailure = (system("NOTEPAD.EXE README.TXT") == -1);

if (fFailure) {
   switch (errno) {
   case E2BIG: // Argument list or environment too big
      break;
   case ENOENT: // Command interpreter cannot be found
      break;
   case ENOEXEC: // Command interpreter has bad format
      break;
   case ENOMEM: // Insufficient memory to run command
      break;
   }
}
```

假设在调用了system函数之后，并在执行if语句之前，执行上述代码的线程被中断了。另外还假设，这个线程被中断后，同一个进程中的另一个线程开始执行，而且这个新线程将执行另一个C运行库函数，后者设置了全局变量**errno**。当CPU后来被分配回第一个线程时，对于上述代码中的system函数调用，**errno**反映的就不再是正确的错误码。为了解决这个问题，每个线程都需要它自己的**errno**变量。此外，必须有某种机制能够让一个线程引用它自己的**errno**变量，同时不能让它去碰另一个线程的**errno**变量。

这仅仅是证明了“标准C/C++运行库最初不是为多线程应用程序而设计”的众多例子中的一个。

在多线程环境中会出问题的C/C++运行库变量和函数有**errno**，**_doserrno**，**strtok**，**_wcstok**，**strerror**，**_strerror**，**tmpnam**，**tmpfile**，**asctime**，**_wasctime**，**gmtime**，**_ecvt**和**_fcvt**等等。

为了保证C和C++多线程应用程序正常运行，必须创建一个数据结构，并使之与使用了C/C++运行库函数的每个线程关联。然后，在调用C/C++运行库函数时，那些函数必须知道去查找主调线程的数据块，从而避免影响到其他线程。

那么，系统在创建新的线程时，是如何知道要分配这个数据块的呢？答案是它并不知道。系统并不知道应用程序是用C/C++来写的，不知道你调用的函数并非天生就是线程安全的。保证线程安全是程序员的责任。创建新线程时，一定不要调用操作系统的**CreateThread**函数。相反，必须调用C/C++运行库函数**_beginthreadex**：

```
unsigned long _beginthreadex(
   void *security,
   unsigned stack_size,
   unsigned (*start_address)(void *),
   void *arglist,
   unsigned initflag,
   unsigned *thrdaddr);
```

_beginthreadex函数的参数列表与**CreateThread**函数的一样，但是参数名称和类型并不完全一样。这是因为Microsoft的C/C++运行库开发组认为，C/C++运行库函数不应该对Windows数据类型有任何依赖。**_beginthreadex**函数也会返回新建线程的句柄，就像**CreateThread**那样。所以，如果已经在自己的源代码中调用了**CreateThread**函数，可以非常方便地用**_beginthreadex**来全局替换所有**CreateThread**。但是，由于数据类型并不完相同，所以可能还必须执行一些类型转换，以便顺利地通过编译。为了简化这个工作，我创建了一个名为**chBEGINTHREADEX**的宏，并在自己的源代码中使用：

```
typedef unsigned (__stdcall *PTHREAD_START) (void *);

#define chBEGINTHREADEX(psa, cbStack, pfnStartAddr, \
   pvParam, fdwCreate, pdwThreadID)                 \
      ((HANDLE) _beginthreadex(                     \
         (void *) (psa),                            \
         (unsigned) (cbStackSize),                  \
         (PTHREAD_START) (pfnStartAddr),            \
         (void *) (pvParam),                        \
         (unsigned) (dwCreateFlags),                \
         (unsigned *) (pdwThreadID)))
```

根据Microsoft为C/C++运行库提供的源代码，很容易看出**_beginthreadex**能而**CreateThread**不能做的事情。事实上，在搜索了Visual Studio安装文件夹后，我在<Program Files>\Microsoft Visual Studio 8\VC\crt\src\Threadex.c中找到了**_beginthreadex**的源代码。为节省篇幅，这里没有全部照抄一遍。相反，我在这里提供了该函数的伪代码版本，强调了其中最有意思的地方：

```
uintptr_t __cdecl _beginthreadex (
void *psa,
unsigned cbStackSize,
unsigned (__stdcall * pfnStartAddr) (void *),
void * pvParam,
unsigned dwCreateFlags,
unsigned *pdwThreadID) {
_ptiddata ptd; // Pointer to thread's data block
uintptr_t thdl; // Thread's handle
// Allocate data block for the new thread.
if ((ptd = (_ptiddata)_calloc_crt(1, sizeof(struct _tiddata))) == NULL)
goto error_return;
// Initialize the data block.
initptd(ptd);
// Save the desired thread function and the parameter
// we want it to get in the data block.
ptd->_initaddr = (void *) pfnStartAddr;
ptd->_initarg = pvParam;
```

```
ptd->_thandle = (uintptr_t)(-1);
// Create the new thread.
thdl = (uintptr_t) CreateThread((LPSECURITY_ATTRIBUTES)psa, cbStackSize,
_threadstartex, (PVOID) ptd, dwCreateFlags, pdwThreadID);
if (thdl == 0) {
// Thread couldn't be created, cleanup and return failure.
goto error_return;
}
// Thread created OK, return the handle as unsigned long.
return(thdl);
error_return:
// Error: data block or thread couldn't be created.
// GetLastError() is mapped into errno corresponding values
// if something wrong happened in CreateThread.
_free_crt(ptd);
return((uintptr_t)0L);
}
```

对于**_beginthreadex**函数，以下几点需要重点关注。

- 每个线程都有自己的专用**_tiddata**内存块，它们是从C/C++运行库的堆（heap）上分配的。
- 传给**_beginthreadex**的线程函数的地址保存在**_tiddata**内存块中。(**_tiddata**结构在Mtdll.h文件的C++源代码中。)纯粹是为了增加趣味性，我在下面重现了这个结构。要传入**_beginthreadex**函数的参数也保存在这个数据块中。
- **_beginthreadex**确实会在内部调用**CreateThread**，因为操作系统只知道用这种方式来创建一个新线程。
- **CreateThread**函数被调用时，传给它的函数地址是**_threadstartex**（而非**pfnStartAddr**）。另外，参数地址是**_tiddata**结构的地址，而非**pvParam**。
- 如果一切顺利，会返回线程的句柄，就像**CreateThread**那样。任何操作失败，会返回0。

```
struct _tiddata {
unsigned long _tid; /* thread ID */
unsigned long _thandle; /* thread handle */
int _terrno; /* errno value */
unsigned long _tdoserrno; /* _doserrno value */
unsigned int _fpds; /* Floating Point data segment */
unsigned long _holdrand; /* rand() seed value */
char* _token; /* ptr to strtok() token */
wchar_t* _wtoken; /* ptr to wcstok() token */
unsigned char* _mtoken; /* ptr to _mbstok() token */
/* following pointers get malloc'd at runtime */
char* _errmsg; /* ptr to strerror()/_strerror() buff */
wchar_t* _werrmsg; /* ptr to _wcserror()/__wcserror() buff */
char* _namebuf0; /* ptr to tmpnam() buffer */
wchar_t* _wnamebuf0; /* ptr to _wtmpnam() buffer */
char* _namebuf1; /* ptr to tmpfile() buffer */
wchar_t* _wnamebuf1; /* ptr to _wtmpfile() buffer */
char* _asctimebuf; /* ptr to asctime() buffer */
wchar_t* _wasctimebuf; /* ptr to _wasctime() buffer */
void* _gmtimebuf; /* ptr to gmtime() structure */
char* _cvtbuf; /* ptr to ecvt()/fcvt buffer */
unsigned char _con_ch_buf[MB_LEN_MAX];
/* ptr to putch() buffer */
unsigned short _ch_buf_used; /* if the _con_ch_buf is used */
/* following fields are needed by _beginthread code */
void* _initaddr; /* initial user thread address */
void* _initarg; /* initial user thread argument */
/* following three fields are needed to support signal handling and runtime errors */
void* _pxcptacttab; /* ptr to exception-action table */
void* _tpxcptinfoptrs;/* ptr to exception info pointers */
int _tfpecode; /* float point exception code */
/* pointer to the copy of the multibyte character information used by the thread */
pthreadmbcinfo ptmbcinfo;
/* pointer to the copy of the locale information used by the thread */
pthreadlocinfo ptlocinfo;
int _ownlocale; /* if 1, this thread owns its own locale */
/* following field is needed by NLG routines */
unsigned long _NLG_dwCode;
/*
* Per-Thread data needed by C++ Exception Handling
```

```
*/
void* _terminate; /* terminate() routine */
void* _unexpected; /* unexpected() routine */
void* _translator; /* S.E. translator */
void* _purecall; /* called when pure virtual happens */
void* _curexception; /* current exception */
void* _curcontext; /* current exception context */
int _ProcessingThrow; /* for uncaught_exception */
void* _curexcspec; /* for handling exceptions thrown from std::unexpected */
#if defined (_M_IA64) || defined (_M_AMD64)
void* _pExitContext;
void* _pUnwindContext;
void* _pFrameInfoChain;
unsigned __int64 _ImageBase;
#if defined (_M_IA64)
unsigned __int64 _TargetGp;
#endif /* defined (_M_IA64) */
unsigned __int64 _ThrowImageBase;
void* _pForeignException;
#elif defined (_M_IX86)
void* _pFrameInfoChain;
#endif /* defined (_M_IX86) */
_setloc_struct _setloc_data;
void* _encode_ptr; /* EncodePointer() routine */
void* _decode_ptr; /* DecodePointer() routine */
void* _reserved1; /* nothing */
void* _reserved2; /* nothing */
void* _reserved3; /* nothing */
int _ cxxReThrow; /* Set to True if it's a rethrown C++ Exception */
unsigned long __initDomain; /* initial domain used by _beginthread[ex] for managed
function */
};
typedef struct _tiddata * _ptiddata;
```

为新线程分配并初始化**_tiddata**结构之后，接着应该知道这个结构是如何与线程关联的。来看看**_threadstartex**函数(它也在C/C++运行库的Threadex.c文件中)。下面是我为这个函数及其helper函数**__callthreadstartex**编写的伪代码版本：

```
static unsigned long WINAPI _threadstartex (void* ptd) {
// Note: ptd is the address of this thread's tiddata block.
// Associate the tiddata block with this thread so
// _getptd() will be able to find it in _callthreadstartex.
TlsSetValue(__tlsindex, ptd);
// Save this thread ID in the _tiddata block.
((_ptiddata) ptd)->_tid = GetCurrentThreadId();
// Initialize floating-point support (code not shown).
// call helper function.
_callthreadstartex();
// We never get here; the thread dies in _callthreadstartex.
return(0L);
}
static void _callthreadstartex(void) {
_ptiddata ptd; /* pointer to thread's _tiddata struct */
// get the pointer to thread data from TLS
ptd = _getptd();
// Wrap desired thread function in SEH frame to
// handle run-time errors and signal support.
__try {
// Call desired thread function, passing it the desired parameter.
// Pass thread's exit code value to _endthreadex.
_endthreadex(
( (unsigned (WINAPI *)(void *))(((_ptiddata)ptd)->_initaddr) )
( ((_ptiddata)ptd)->_initarg ) ) ;
}
__except(_XcptFilter(GetExceptionCode(), GetExceptionInformation())){
// The C run-time's exception handler deals with run-time errors
// and signal support; we should never get it here.
_exit(GetExceptionCode());
}
}
```

关于**_threadstartex**函数，要注意以下重点：

- 新的线程首先执行**RtlUserThreadStart** (在NTDLL.dll文件中)，然后再跳转到**_threadstartex**。
- **_threadstartex**惟一的参数就是新线程的**_tiddata**内存块的地址。
- **TlsSetValue**是一个操作系统函数，它将一个值与主调线程关联起来。这就是所谓的线程本地存储(Thread Local Storage，TLS)，详情参见第21章。**_threadstartex**函数将**_tiddata**内存块与新建线程关联起来。
- 在无参数的helper函数**_callthreadstartex**中，一个SEH帧将预期要执行的线程函数包围起来。这个帧处理着与运行库有关的许多事情——比如运行时错误(如抛出未被捕捉的C++异常)——和C/C++运行库的signal函数。这一点相当重要。如果用CreateThread函数新建了一个线程，然后调用C/C++运行库的signal函数，那么signal函数不能正常工作。
- 预期要执行的线程函数会被调用，并向其传递预期的参数。前面讲过，函数的地址和参数由**_beginthreadex**保存在TLS的_tiddata数据块中；并会在**_callthreadstartex**中从TLS中获取。
- 线程函数的返回值被认为是线程的退出代码。
  注意**_callthreadstartex**不是简单地返回到**_threadstartex**，继而到**RtlUserThreadStart**；如果是那样的话，线程会终止运行，其退出代码也会被正确设置，但线程的**_tiddata**内存块不会被销毁。这会导致应用程序出现内存泄漏。为防止出现这个问题，会调用**_endthreadex**（也是一个C/C++运行库函数），并向其传递退出代码。

最后一个需要关注的函数是**_endthreadex**(也在C运行库的Threadex.c文件中)。下面是我编写的该函数的伪代码版本：

```
void __cdecl _endthreadex (unsigned retcode) {
  _ptiddata ptd; // Pointer to thread's data block

  // Clean up floating-point support (code not shown).

  // Get the address of this thread's tiddata block.
  ptd = _getptd_noexit ();

  // Free the tiddata block.
  if (ptd != NULL)
    _freeptd(ptd);

  // Terminate the thread.
  ExitThread(retcode);
}
```

对于**_endthreadex**函数，要注意几下几点：

- C运行库的**_getptd_noexit**函数在内部调用操作系统的**TlsGetValue**函数，后者获取主调线程的**tiddata**内存块的地址。
- 然后，此数据块被释放，调用操作系统的**ExitThread**函数来实际地销毁线程。当然，退出代码会被传递，并被正确地设置。

在本章早些时候，我曾建议大家应该避免使用**ExitThread**函数。这是千真万确的，而且我在这里并不打算自相矛盾。前面说过，此函数会杀死主调线程，而且不允许它从当前执行的函数返回。由于函数没有返回，所以构造的任何C++对象都不会被析构。现在，我们又有了不调用**ExitThread**函数的另一个理由：它会阻止线程的**_tiddata**内存块被释放，使应用程序出现内存泄漏（直到整个进程终止）。

Microsoft的C++开发团队也意识到，总有一些开发人员喜欢调用**ExitThread**。所以，他们必须使这成为可能，同时尽可能避免应用程序出现内存泄漏的情况。如果真的想要强行杀死自己的线程，可以让它调用**_endthreadex**(而不是**ExitThread**)来释放线程的**_tiddata**块并退出。不过，我并不鼓励你调用**_endthreadex**。

现在，你应该理解了C/C++运行库函数为什么要为每一个新线程准备一个独立的数据块，而且应该理解了**_beginthreadex**如何分配和初始化此数据块，并将它与新线程关联起来。另外，你还应理解了**_endthreadex**函数在线程终止运行时是如何释放该数据块的。

一旦这个数据块被初始化并与线程关联，线程调用的任何需要“每线程实例数据”的C/C++运行库函数都可以轻易获取主调线程的数据块的地址(通过**TlsGetValue**)，并操纵线程的数据。这对函数来说是没有问题的。但是，对于errno之类的全局变量，它又是如何工作的呢？errno是在标准C headers中定义的，如下所示：

```
_CRTIMP extern int * __cdecl _errno(void);
#define errno (*_errno())

int* __cdecl _errno(void) {
   _ptiddata ptd = _getptd_noexit();
   if (!ptd) {
      return &ErrnoNoMem;
   } else {
      return (&ptd->_terrno);
   }
}
```

任何时候引用**errno**，实际都是在调用内部的C/C++运行库函数**_errno**。该函数将地址返回给“与主调线程关联的数据块”中的**errno**数据成员。注意，**errno**宏被定义为获取该地址的内容。这个定义是必要的，因为很可能写出下面这样的代码：

```
int *p = &errno;
if (*p == ENOMEM) {
   ...
}
```

如果内部函数**_errno**只是返回**errno**的值，上述代码将不能通过编译。
C/C++运行库还围绕特定的函数放置了同步原语（synchronization primitives）。例如，如果两个线程同时调用**malloc**，堆就会损坏。C/C++运行库函数阻止两个线程同时从内存堆中分配内存。具体的办法是让第2个线程等待，直至第1个线程从**malloc**函数返回。然后，才允许第2个线程进入。(线程同步将在第8章和第9章详细讨论。)显然，所有这些额外的工作影响了C/C++运行库的多线程版本的性能。

C/C++运行库函数的动态链接版本被写得更加泛化，使其可以被使用了C/C++运行库函数的所有运行的应用程序和DLL共享。因此，库只有一个多线程版本。由于C/C++运行库是在一个DLL中提供的，所以应用程序(.exe文件)和DLL不需要包含C/C++运行库函数的代码，所以可以更小一些。另外，如果Microsoft修复了C/C++运行库DLL的任何bug，应用程序将自动获得修复。

就像你期望的一样，C/C++运行库的启动代码为应用程序的主线程分配并初始化了一个数据块。这样一来，主线程就可以安全地调用任何C/C++运行库函数。当主线程从其入口函数返回的时候，C/C++运行库函数会释放关联的数据块。此外，启动代码设置了正确的结构化异常处理代码，使主线程能成功调用C/C++运行库的**signal**函数。

6.7.1 用_beginthreadex 而不要用 CreateThread 创建线程

你可能会好奇，假如调用**CreateThread**而不是C/C++运行库的**_beginthreadex**来创建新线程，会发生什么呢？当一个线程调用一个需要**_tiddata**结构的C/C++运行库函数时，会发生下面的情况。(大多数C/C++运行库函数都是线程安全的，不需要这个结构。)首先，C/C++运行库函数尝试取得线程数据块的地址(通过调用**TlsGetValue**)。如果**NULL**被作为**_tiddata**块的地址返回，表明主调线程没有与之关联的**_tiddata**块。在这个时候，C/C++运行库函数会为主调线程分配并初

始化一个**_tiddata**块。然后，这个块会与线程关联(通过**TlsSetValue**) ，而且只要线程还在运行，这个块就会一直存在并与线程关联。现在，C/C++运行库函数可以使用线程的**_tiddata**块，以后调用的任何C/C++运行库函数也都可以使用。

当然，这是相当诱人的，因为线程（几乎）可以顺畅运行。但事实上，问题还是有的。第一个问题是，假如线程使用了C/C++运行库的**signal**函数，则整个进程都会终止，因为结构化异常处理（SEH）帧没有就绪。第二个问题是，假如线程不是通过调用**_endthreadex**来终止的，数据块就不能被销毁，从而导致内存泄漏。(对于一个用**CreateThread**函数来创建的线程，谁会调用**_endthreadex**呢？)

注意：当你的模块链接到C/C++运行库的DLL版本时，这个库会在线程终止时收到一个DLL_THREAD_DETACH通知，并会释放_tiddata 块（如果已分配的话）。虽然这可以防止_tiddata块的泄漏，但仍然强烈建议使用_beginthreadex来创建线程，而不要用CreateThread。

6.7.2 绝对不要调用的 C/C++运行库函数

C/C++运行库还包括以下两个函数：

```
unsigned long _beginthread(
   void (__cdecl *start_address)(void *),
   unsigned stack_size,
   void *arglist);
```

和：

```
void _endthread(void);
```

新的**_beginthreadex**和**_endthreadex**函数已经取代了这两个传统的函数。如你所见，**_beginthread**函数的参数较少，所以和全功能的**_beginthreadex**函数相比，它的局限性较大。例如，使用**_beginthread**函数，你不能创建具有安全属性的线程，不能创建可以挂起的线程，也不能获得线程ID值。**_endthread**函数的情况与此类似：它是无参数的，这意味着线程的退出代码被硬编码为0。

_endthread函数还存在另一个鲜为人知的问题。**_endthread**函数在调用**ExitThread**前，会调用**CloseHandle**，向其传入新线程的句柄。为了理解这为什么会成为一个问题，来看看以下代码：

```
DWORD dwExitCode;
HANDLE hThread = _beginthread(...);
GetExitCodeThread(hThread, &dwExitCode);
CloseHandle(hThread);
```

在第一个线程调用**GetExitCodeThread**之前，新建的线程就可能已经执行，返回，并终止运行了。如果发生上述情况， **hThread**就是无效的，因为**_endthread**已关闭了新线程的句柄。不用说，对**CloseHandle**函数的调用也会因为相同的原因而失败。

新的**_endthreadex**函数不会关闭线程的句柄。所以，用**_beginthreadex**函数调用来替换**_beginthread**函数调用，上述代码片断就没有bug了。记住，线程函数返回时，**_beginthreadex**函数调用的是**_endthreadex**，而**_beginthread**调用的是**_endthread**函数。

6.8 了解自己的身份

线程执行时，经常希望调用能改变执行环境的Windows函数。例如，一个线程也许希望更改它或者它的进程的优先级（优先级将在第7章详细讨论）。由于线程经常要改变它（或者它的进程）的环境，所以Windows提供了一些函数来方便线程引用它的进程内核对象或者它自己的线程内核对象：

```
HANDLE GetCurrentProcess();
HANDLE GetCurrentThread();
```

这两个函数都返回到主调线程的进程或线程内核对象的一个伪句柄（pseudohandle）。它们不会在主调进程的句柄表中新建句柄。而且，调用这两个函数，不会影响进程或线程内核对象的使用计数。如果调用CloseHandle，将一个伪句柄作为参数传入，CloseHandle只是简单地忽略此调用，并返回FALSE。在这种情况下，GetLastError将返回ERROR_INVALID_HANDLE。

调用一个Windows函数时，如果此函数需要到进程或线程的一个句柄，那么可以传递一个伪句柄，这将导致函数在主调进程或线程上执行它的操作。例如，通过像下面这样调用GetProcessTimes函数，一个线程可以查询其进程的计时信息：

```
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
GetProcessTimes(GetCurrentProcess(),
&ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
```

类似地，通过调用GetThreadTimes，一个线程可以查询自己的线程时间：

```
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
GetThreadTimes(GetCurrentThread(),
&ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
```

为了标识一个特定的进程或线程，可以使用几个Windows函数，用进程或线程惟一的系统级ID来标识它。一个线程可以通过以下函数来查询它的进程的惟一ID或其它自己的惟一ID：

```
DWORD GetCurrentProcessId();
DWORD GetCurrentThreadId();
```

这两个函数通常都不如返回伪句柄的那些函数有用，但在个别情况下，它们更方便。

6.8.1 将伪句柄转换为真正的句柄

有时或许需要一个真正的线程句柄，而不是一个伪句柄。 所谓“真正的句柄”，指的是能明确、无歧义地标识一个线程的句柄。来仔细分析下面的代码：

```
DWORD WINAPI ParentThread(PVOID pvParam) {
HANDLE hThreadParent = GetCurrentThread();
CreateThread(NULL, 0, ChildThread, (PVOID) hThreadParent, 0, NULL);
// Function continues...
}
DWORD WINAPI ChildThread(PVOID pvParam) {
HANDLE hThreadParent = (HANDLE) pvParam;
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
GetThreadTimes(hThreadParent,
&ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
// Function continues...
}
```

能看出这个代码段的问题吗？其意图是让父线程向子线程传递一个可以标识父线程的句柄。但是，父线程传递的是一个伪句柄，而不是一个真正的句柄。子线程开始执行时，它把这个伪句柄传给GetThreadTimes函数，这将导致子线程得到的是它自己的CPU计时数据，而不是父线程的。之所以会发生这种情况，是因为线程的伪句柄是一个指向当前线程的句柄；换言之，指向的是发出函数调用的那个线程。

为了修正这段代码，必须将伪句柄转换为一个真正的句柄。DuplicateHandle函数（详见第3章的讨论）可以执行这个转换：

```
BOOL DuplicateHandle(
HANDLE hSourceProcess,
HANDLE hSource,
HANDLE hTargetProcess,
PHANDLE phTarget,
DWORD dwDesiredAccess,
BOOL bInheritHandle,
DWORD dwOptions);
```

正常情况下，利用这个函数，你可以根据与进程A相关的一个内核对象句柄来创建一个新句柄，并让它同进程B相关。但是，我们可以采取一种特殊的方式来使用它，以纠正前面的那个代码段的错误。纠正过后的代码如下：

```
DWORD WINAPI ParentThread(PVOID pvParam) {
HANDLE hThreadParent;
DuplicateHandle(
GetCurrentProcess(), // Handle of process that thread
// pseudohandle is relative to
GetCurrentThread(), // Parent thread's pseudohandle
GetCurrentProcess(), // Handle of process that the new, real,
// thread handle is relative to
&hThreadParent, // Will receive the new, real, handle
// identifying the parent thread
0, // Ignored due to DUPLICATE_SAME_ACCESS
FALSE, // New thread handle is not inheritable
DUPLICATE_SAME_ACCESS); // New thread handle has same
// access as pseudohandle
CreateThread(NULL, 0, ChildThread, (PVOID) hThreadParent, 0, NULL);
// Function continues...
}
DWORD WINAPI ChildThread(PVOID pvParam) {
HANDLE hThreadParent = (HANDLE) pvParam;
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
GetThreadTimes(hThreadParent,
&ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
CloseHandle(hThreadParent);
// Function continues...
}
```

现在，当父线程执行时，它会把标识父线程的有歧义的伪句柄转换为一个新的、真正的句柄，后者明确、无歧义地标识了父线程。然后，它将这个真正的句柄传给CreateThread。当子线程开始执行时，其pvParam参数就会包含这个真正的线程句柄。在调用任何函数时，只要传入这个句柄，影响的就将是父线程，而非子线程。

因为DuplicateHandle递增了指定内核对象的使用计数，所以在用完复制的对象句柄后，有必要把目标句柄传给CloseHandle，以递减对象的使用计数。前面的代码体现了这一点。调用GetThreadTimes之后，子线程紧接着调用CloseHandle来递减父线程对象的使用计数。在这段代码中，我假设子线程不会用这个句柄调用其他任何函数。如果还要在调用其他函数时传入父线程的句柄，那么只有在子线程完全不需要此句柄的时候，才能调用CloseHandle。

还要强调一点，DuplicateHandle函数可用于把进程的伪句柄转换为真正的进程句柄，如下所示：

```
HANDLE hProcess;
DuplicateHandle(
GetCurrentProcess(), // Handle of process that the process
// pseudohandle is relative to
GetCurrentProcess(), // Process' pseudohandle
GetCurrentProcess(), // Handle of process that the new, real,
```

```
// process handle is relative to
&hProcess, // Will receive the new, real
// handle identifying the process
0, // Ignored because of DUPLICATE_SAME_ACCESS
FALSE, // New process handle is not inheritable
DUPLICATE_SAME_ACCESS); // New process handle has same
// access as pseudohandle
```