第3章 内核对象

在介绍Windows API的时候，首先要讲述内核对象以及它们的句柄。本章将要介绍一些比较抽象的概念，在此并不讨论某个特定内核对象的特性，相反只是介绍适用于所有内核对象的特性。

首先介绍一个比较具体的问题，准确地理解内核对象对于想要成为一名 Windows软件开发能手的人来说是至关重要的。内核对象可以供系统和应用程序使用来管理各种各样的资源，比如进程、线程和文件等。本章讲述的概念也会出现在本书的其他各章之中。但是，在你开始使用实际的函数来操作内核对象之前，是无法深刻理解本章讲述的部分内容的。因此当阅读本书的其他章节时，可能需要经常回过来参考本章的内容。

3.1 什么是内核对象

作为一个Windows软件开发人员，你经常需要创建、打开和操作各种内核对象。系统要创建和操作若干类型的内核对象，比如存取符号对象、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、信箱对象、互斥对象、管道对象、进程对象、信标对象、线程对象和等待计时器对象等。这些对象都是通过调用函数来创建的。例如， CreateFileMapping函数可使系统能够创建一个文件映射对象。每个内核对象只是内核分配的一个内存块，并且只能由该内核访问。该内存块是一种数据结构，它的成员负责维护该对象的各种信息。有些数据成员（如安全性描述符、使用计数等）在所有对象类型中是相同的，但大多数数据成员属于特定的对象类型。例如，进程对象有一个进程 ID、一个基本优先级和一个退出代码，而文件对象则拥有一个字节位移、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被内核访问，因此应用程序无法在内存中找到这些数据结构并直接改变它们的内容。 Microsoft规定了这个限制条件，目的是为了确保内核对象结构保持状态的一致。这个限制也使Microsoft能够在不破坏任何应用程序的情况下在这些结构中添加、删除和修改数据成员。

如果我们不能直接改变这些数据结构，那么我们的应用程序如何才能操作这些内核对象呢？解决办法是，Windows提供了一组函数，以便用定义得很好的方法来对这些结构进行操作。这些内核对象始终都可以通过这些函数进行访问。当调用一个用于创建内核对象的函数时，该函数就返回一个用于标识该对象的句柄。该句柄可以被视为一个不透明值，你的进程中的任何线程都可以使用这个值。将这个句柄传递给 Windows的各个函数，这样，系统就能知道你想操作哪个内核对象。本章后面还要详细讲述这些句柄的特性。

为了使操作系统变得更加健壮，这些句柄值是与进程密切相关的。因此，如果将该句柄值传递给另一个进程中的一个线程（使用某种形式的进程间的通信）那么这另一个进程使用你的进程的句柄值所作的调用就会失败。在 3.3节“ 跨越进程边界共享内核对象 ”中，将要介绍 3种机制，使多个进程能够成功地共享单个内核对象。

3.1.1 内核对象的使用计数

内核对象由内核所拥有，而不是由进程所拥有。换句话说，如果你的进程调用了一个创建

内核对象的函数，然后你的进程终止运行，那么内核对象不一定被撤消。在大多数情况下，对象将被撤消，但是如果另一个进程正在使用你的进程创建的内核对象，那么该内核知道，在另一个进程停止使用该对象前不要撤消该对象，必须记住的是，内核对象的存在时间可以比创建该对象的进程长。

内核知道有多少进程正在使用某个内核对象，因为每个对象包含一个使用计数。使用计数是所有内核对象类型常用的数据成员之一。当一个对象刚刚创建时，它的使用计数被置为 1。然后，当另一个进程访问一个现有的内核对象时，使用计数就递增 1。当进程终止运行时，内核就自动确定该进程仍然打开的所有内核对象的使用计数。如果内核对象的使用计数降为 0，内核就撤消该对象。这样可以确保在没有进程引用该对象时系统中不保留任何内核对象。

3.1.2 安全性

内核对象能够得到安全描述符的保护。安全描述符用于描述谁创建了该对象，谁能够访问或使用该对象，谁无权访问该对象。安全描述符通常在编写服务器应用程序时使用，如果你编写客户机端的应用程序，那么可以忽略内核对象的这个特性。

> Windows 98 根据原来的设计，Windows 98并不用作服务器端的操作系统。为此，Microsoft公司没有在Windows 98中配备安全特性。不过，如果你现在为 Windows 98 设计软件，在实现你的应用程序时仍然应该了解有关的安全问题，并且使用相应的访问信息，以确保它能在Windows 2000上正确地运行

用于创建内核对象的函数几乎都有一个指向 SECURITY_ATTRIBUTES结构的指针作为其参数，下面显示了CreateFileMapping函数的指针：

```
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该参数传递 NULL，这样就可以创建带有默认安全性的内核对象。默认安全性意味着对象的管理小组的任何成员和对象的创建者都拥有对该对象的全部访问权，而其他所有人均无权访问该对象。但是，可以指定一个 SECURITY_ATTRIBUTES结构，对它进行初始化，并为该参数传递该结构的地址。SECURITY_ATTRIBUTES结构类似下面的样子：

```
typedef struct _SECURITY_ATTRIBUTES {
   DWORD nLength;
   LPVOID lpSecurityDescriptor;
   BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

尽管该结构称为SECURITY_ATTRIBUTES，但是它包含的与安全性有关的成员实际上只有一个，即lpSecurityDescriptor。如果你想要限制人们对你创建的内核对象的访问，必须创建一个安全性描述符，然后像下面这样对SECURITY_ATTRIBUTES结构进行初始化：

```
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, "MyFileMapping");
:
:
```

由于bInheritHandle这个成员与安全性毫无关系，因此准备推迟到本章后面部分继承性一节中再介绍bInheritHandle这个成员。

当你想要获得对相应的一个内核对象的访问权（而不是创建一个新对象）时，必须设定要对该对象执行什么操作。例如，如果想要访问一个现有的文件映射内核对象，以便读取它的数据，那么应该调用下面这个OpenfileMapping函数：

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

通过将FILE_MAP_READ作为第一个参数传递给OpenFileMapping，指明打算在获得对该文件映象的访问权后读取该文件，OpenFileMapping函数在返回一个有效的句柄值之前，首先执行一次安全检查。如果（已登录用户）被允许访问现有的文件映射内核对象，OpenFileMapping就返回一个有效的句柄。但是，如果被拒绝访问该对象，OpenFileMapping将返回NULL，而调用GetLastError函数则返回5（ERROR_ACCESS_DENIED），同样，大多数应用程序并不使用该安全性，因此将不进一步讨论这个问题。

Windows 98 虽然许多应用程序不需要考虑安全性问题，但是Windows的许多函数要求传递必要的安全访问信息。为Windows 98设计的若干应用程序在Windows 2000上无法正确地运行，因为在实现这些应用程序时没有对安全问题给于足够的考虑。

例如，假设一个应用程序在开始运行时要从注册表的子关键字中读取一些数据。为了正确地进行这项操作，你的代码应该调用 RegOpenKeyEx，传递KEY_QUERY_VALUE，以便获得必要的访问权。

但是，许多应用程序原先是为Windows 98开发的，当时没有考虑到运行Windows 2000的需要。由于Windows 98没有解决注册表的安全问题，因此软件开发人员常常要调用RegOpenKeyEx函数，传递KEY_All_ACCESS，作为必要的访问权。开发人员这样做的原因是，它是一种比较简单的解决方案，意味着开发人员不必考虑究竟需要什么访问权。问题是注册表的子关键字可以被用户读取，但是不能写入。

因此，当该应用程序现在放在Windows 2000上运行时，用KEY_ALL_ACCESS调用RegOpenKeyEx就会失败，而且，没有相应的错误检查方法，应用程序的运行就会产生不可预料的结果。

如果开发人员想到安全问题，把KEY_ALL_ACCESS改为KEY_QUERY_VALUE，则该产品可适用于两种操作系统平台。

开发人员的最大错误之一就是忽略安全访问标志。使用正确的标志会使最初为Windows 98 设计的应用程序更易于向Windows 2000 转换。

除了内核对象外，你的应用程序也可以使用其他类型的对象，如菜单、窗口、鼠标光标、刷子和字体等。这些对象属于用户对象或图形设备接口（GDI）对象，而不是内核对象。当初次着手为Windows编程时，如果想要将用户对象或GDI对象与内核对象区分开来，你一定会感到不知所措。比如，图标究竟是用户对象还是内核对象呢？若要确定一个对象是否属于内核对象，最容易的方法是观察创建该对象所用的函数。创建内核对象的所有函数几乎都有一个参数，你可以用来设定安全属性的信息，这与前面讲到的CreateFileMapping函数是相同的。

用于创建用户对象或GDI对象的函数都没有PSECURITY_ATTRIBUTES参数。例如，让我们来看一看下面这个CreateIcon函数：

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

3.2 进程的内核对象句柄表

当一个进程被初始化时,系统要为它分配一个句柄表。该句柄表只用于内核对象不用于用户对象或GDI对象。句柄表的详细结构和管理方法并没有具体的资料说明。通常我并不介绍操作系统中没有文档资料的那些部分。不过，在这种情况下，我会进行例外处理，因为，作为一个称职的Windows程序员，必须懂得如何管理进程的句柄表。由于这些信息没有文档资料，因此不能保证所有的详细信息都正确无误，同时，在Windows 2000、Windows 98和Windows CE中，它们的实现方法是不同的。为此，请认真阅读下面介绍的内容以加深理解，在此不学习系统是如何进行操作的。

表3-1显示了进程的句柄表的样子。可以看到，它只是个数据结构的数组。每个结构都包含一个指向内核对象的指针、一个访问屏蔽和一些标志。

表3-1 进程的句柄结构

索 引	内核对象内存块的指针	访问屏蔽（标志位的DWORD）	标 志（标志位的DWORD）
1	0x????????	0x????????	0x????????
2	0x????????	0x????????	0x????????
…	…	…	…

3.2.1 创建内核对象

当进程初次被初始化时，它的句柄表是空的。然后，当进程中的线程调用创建内核对象的函数时，比如CreateFileMapping，内核就为该对象分配一个内存块，并对它初始化。这时，内核对进程的句柄表进行扫描，找出一个空项。由于表3-1中的句柄表是空的，内核便找到索引1位置上的结构并对它进行初始化。该指针成员将被设置为内核对象的数据结构的内存地址，访问屏蔽设置为全部访问权，同时，各个标志也作了设置（关于标志，将在本章后面部分的继承性一节中介绍）。

下面列出了用于创建内核对象的一些函数（但这决不是个完整的列表）：

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

HANDLE CreateFile(
   PCTSTR pszFileName,
   DWORD dwDesiredAccess,
```

```
    DWORD dwShareMode,
    PSECURITY_ATTRIBUTES psa,
    DWORD dwCreationDistribution,
    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所有函数均返回与进程相关的句柄，这些句柄可以被在相同进程中运行的任何或所有线程成功地加以使用。该句柄值实际上是放入进程的句柄表中的索引，它用于标识内核对象的信息存放的位置。因此当调试一个应用程序并且观察内核对象句柄的实际值时，会看到一些较小的值，如1，2等。请记住，句柄的含义并没有记入文档资料，并且可能随时变更。实际上在Windows 2000中，返回的值用于标识放入进程的句柄表的该对象的字节数，而不是索引号本身。

每当调用一个将内核对象句柄接受为参数的函数时，就要传递由一个Create*&函数返回的值。从内部来说，该函数要查看进程的句柄表，以获取要生成的内核对象的地址，然后按定义得很好的方式来生成该对象的数据结构。

如果传递了一个无效索引（句柄），该函数便返回失败，而GetLastError则返回6（ERROR_INVALID_HANDLE）。由于句柄值实际上是放入进程句柄表的索引，因此这些句柄是与进程相关的，并且不能由其他进程成功地使用。

如果调用一个函数以便创建内核对象，但是调用失败了，那么返回的句柄值通常是0（NULL）。发生这种情况是因为系统的内存非常短缺，或者遇到了安全方面的问题。不过有少数函数在运行失败时返回的句柄值是-1（INVALID_HANDLE_VALUE）。例如，如果CreateFile未能打开指定的文件，那么它将返回INVALID_HANDLE_VALUE，而不是返回NULL。当查看创建内核对象的函数返回值时，必须格外小心。特别要注意的是，只有当调用CreateFile函数时，才能将该值与INVALID_HANDLE_VALUE进行比较。下面的代码是不正确的：

```
HANDLE hMutex = CreateMutex(…);
if (hMutex == INVALID_HANDLE_VALUE) {
   // We will never execute this code because
   // CreateMutex returns NULL if it fails.
}
```

同样，下面的代码也不正确：

```
HANDLE hFile = CreateFile(…);
if (hFile == NULL) {
   // We will never execute this code because CreateFile
   // returns INVALID_HANDLE_VALUE(-1) if it fails.
}
```

3.2.2 关闭内核对象

无论怎样创建内核对象，都要向系统指明将通过调用CloseHandle来结束对该对象的操作：

```
BOOL CloseHandle(HANDLE hobj);
```

该函数首先检查调用进程的句柄表，以确保传递给它的索引（句柄）用于标识一个进程实际上无权访问的对象。如果该索引是有效的，那么系统就可以获得内核对象的数据结构的地址，并可确定该结构中的使用计数的数据成员。如果使用计数是0，该内核便从内存中撤消该内核对象。

如果将一个无效句柄传递给 CloseHandle，将会出现两种情况之一。如果进程运行正常，CloseHandle返回FALSE，而GetLastError则返回ERROR_INVALID_HANDLE。如果进程正在排除错误，系统将通知调试程序，以便能排除它的错误。

在CloseHandle返回之前，它会清除进程的句柄表中的项目，该句柄现在对你的进程已经无效，不应该试图使用它。无论内核对象是否已经撤消，都会发生清除操作。当调用CloseHandle函数之后，将不再拥有对内核对象的访问权，不过，如果该对象的使用计数没有递减为0，那么该对象尚未被撤消。这没有问题，它只是意味着一个或多个其他进程正在使用该对象。当其他进程停止使用该对象时（通过调用 CloseHandle），该对象将被撤消。

假如忘记调用CloseHandle函数，那么会不会出现内存泄漏呢？答案是可能的，但是也不一定。在进程运行时，进程有可能泄漏资源（如内核对象）。但是，当进程终止运行时，操作系统能够确保该进程使用的任何资源或全部资源均被释放，这是有保证的。对于内核对象来说，系统将执行下列操作：当进程终止运行时，系统会自动扫描进程的句柄表。如果该表拥有任何无效项目（即在终止进程运行前没有关闭的对象），系统将关闭这些对象句柄。如果这些对象中的任何对象的使用计数降为0，那么内核便撤消该对象。

因此，应用程序在运行时有可能泄漏内核对象，但是当进程终止运行时，系统将能确保所有内容均被正确地清除。另外，这个情况适用于所有对象、资源和内存块，也就是说，当进程终止运行时，系统将保证进程不会留下任何对象。

3.3 跨越进程边界共享内核对象

许多情况下，在不同进程中运行的线程需要共享内核对象。下面是为何需要共享的原因：

- 文件映射对象使你能够在同一台机器上运行的两个进程之间共享数据块。
- 邮箱和指定的管道使得应用程序能够在连网的不同机器上运行的进程之间发送数据块。
- 互斥对象、信标和事件使得不同进程中的线程能够同步它们的连续运行，这与一个应用程序在完成某项任务时需要将情况通知另一个应用程序的情况相同。

由于内核对象句柄与进程相关，因此这些任务的执行情况是不同的。不过， Microsoft公司有若干很好的理由将句柄设计成与进程相关的句柄。最重要的理由是要实现它的健壮性。如果内核对象句柄是系统范围的值，那么一个进程就能很容易获得另一个进程使用的对象的句柄，从而对该进程造成很大的破坏。另一个理由是安全性。内核对象是受安全性保护的，进程在试图操作一个对象之前，首先必须申请获得操作该对象的许可权。对象的创建人只需要拒绝向用户赋予许可权，就能防止未经授权的用户接触该对象。

在下面的各节中，将要介绍允许进程共享内核对象的3个不同的机制。

3.3.1 对象句柄的继承性

只有当进程具有父子关系时，才能使用对象句柄的继承性。在这种情况下，父进程可以使

用一个或多个内核对象句柄，并且该父进程可以决定生成一个子进程，为子进程赋予对父进程的内核对象的访问权。若要使这种类型的继承性能够实现，父进程必须执行若干个操作步骤。

首先，当父进程创建内核对象时，必须向系统指明，它希望对象的句柄是个可继承的句柄。请记住，虽然内核对象句柄具有继承性，但是内核对象本身不具备继承性。

若要创建能继承的句柄，父进程必须指定一个 SECURITY_ATTRIBUTES结构并对它进行初始化，然后将该结构的地址传递给特定的 Create函数。下面的代码用于创建一个互斥对象，并将一个可继承的句柄返回给它：

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;   // Make the returned handle inheritable.

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

该代码对一个SECURITY_ATTRIBUTES结构进行初始化，指明该对象应该使用默认安全性（在Windows 98中该安全性被忽略）来创建，并且返回的句柄应该是可继承的。

Windows 98 尽管Windows 98不拥有完整的对安全性的支持，但是它却支持继承性，因此，Windows 98能够正确地使用bInheritHandle成员的值。

现在介绍存放在进程句柄表项目中的标志。每个句柄表项目都有一个标志位，用来指明该句柄是否具有继承性。当创建一个内核对象时，如果传递NULL作为PSECURITY_ATTRIBUTES的参数，那么返回的句柄是不能继承的，并且该标志位是 0。如果将bInheritHandle成员置为TRUE，那么该标志位将被置为1。

表3-2显示了一个进程的句柄表。

表3-2 包含两个有效项目的进程句柄表

索 引	内核对象内存块的指针	访问屏蔽（标志位的DWORD）	标 志（标志位的DWORD）
1	0xF0000000	0x????????	0x00000000
2	0x00000000	(无)	(无)
3	0xF0000010	0x????????	0x00000001

表3-2表示该进程拥有对两个内核对象（句柄1和3）的访问权。句柄1是不可继承的，而句柄3是可继承的。

使用对象句柄继承性时要执行的下一个步骤是让父进程生成子进程。这要使用 CreateProcess函数来完成：

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

下一章将详细介绍这个函数的用法，不过现在我想要让你注意bInheritHandle这个参数。一般来说，当生成一个进程时，将为该参数传递FALSE。该值告诉系统，不希望子进程继承父进程的句柄表中的可继承句柄。

但是，如果为该参数传递TRUE，那么子进程就可以继承父进程的可继承句柄值。当传递TRUE时，操作系统就创建该新子进程，但是不允许子进程立即开始执行它的代码。当然，系统为子进程创建一个新的和空的句柄表，就像它为任何新进程创建句柄表那样。不过，由于将TRUE传递给了CreateProcess的bInheritHandles参数，因此系统要进行另一项操作，即它要遍历父进程的句柄表，对于它找到的包含有效的可继承句柄的每个项目，系统会将该项目准确地拷贝到子进程的句柄表中。该项目拷贝到子进程的句柄表中的位置将与父进程的句柄表中的位置完全相同。这个情况非常重要，因为它意味着在父进程与子进程中，标识内核对象所用的句柄值是相同的。

除了拷贝句柄表项目外，系统还要递增内核对象的使用计数，因为现在两个进程都使用该对象。如果要撤消内核对象，那么父进程和子进程必须调用该对象上的CloseHandle函数，也可以终止进程的运行。子进程不必首先终止运行，但是父进程也不必首先终止运行。实际上，CreateProcess函数返回后，父进程可以立即关闭对象的句柄，而不影响子进程对该对象进行操作的能力。

表3-3显示了子进程被允许运行前该进程的句柄表。可以看到，项目1和项目2尚未初始化，因此是个无效句柄，子进程是无法使用的。但是，项目3确实标识了一个内核对象。实际上，它标识的内核对象的地址是0xF0000010，这与父进程的句柄表中的对象地址相同。访问屏蔽与父进程中的屏蔽相同，两者的标志也相同。这意味着如果该子进程要生成它自己的子进程（即父进程的孙进程），该孙进程也将继承与该内核对象句柄相同的句柄值、同样的访问权和相同的标志，同时，对象的使用计数再次被递增。

表3-3 继承父进程的可继承句柄后的子进程句柄表

索 引	内核对象内存块的指针	访问屏蔽（标志位的DWORD）	标 志（标志位的DWORD）
1	0x00000000	（无）	（无）
2	0x00000000	（无）	（无）
3	0xF0000010	0x????????	0x00000001

应该知道，对象句柄的继承性只有在生成子进程的时候才能使用。如果父进程准备创建带有可继承句柄的新内核对象，那么已经在运行的子进程将无法继承这些新句柄。

对象句柄的继承性有一个非常奇怪的特征，那就是当使用它时，子进程不知道它已经继承了任何句柄。只有在另一个进程生成子进程时记录了这样一个情况，即它希望被赋予对内核对象的访问权时，才能使用内核对象句柄的继承权。通常，父应用程序和子应用程序都是由同一个公司编写的，但是，如果另一个公司记录了子应用程序期望的对象，那么该公司也能够编写子应用程序。

子进程为了确定它期望的内核对象的句柄值，最常用的方法是将句柄值作为一个命令行参数传递给子进程，该子进程的初始化代码对命令行进行分析（通常通过调用sscanf函数来进行分析），并取出句柄值。一旦子进程拥有该句柄值，它就具有对该对象的无限访问权。请注意，句柄继承权起作用的唯一原因是，父进程和子进程中的共享内核对象的句柄值是相同的，这就是为什么父进程能够将句柄值作为命令行参数来传递的原因。

当然，可以使用其他形式的进程间通信，将已继承的内核对象句柄值从父进程传送给子进程。方法之一是让父进程等待子进程完成初始化（使用第9章介绍的WaitForInpuIdle函数），然后，父进程可以将一条消息发送或展示在子进程中的一个线程创建的窗口中。

另一个方法是让父进程将一个环境变量添加给它的环境程序块。该变量的名字是子进程知道要查找的某种信息，而变量的值则是内核对象要继承的值。这样，当父进程生成子进程时，子进程就继承父进程的环境变量，并且能够非常容易地调用 GetEnvironmentVariable函数，以获取被继承对象的句柄值。如果子进程要生成另一个子进程，那么使用这种方法是极好的，因为环境变量可以被再次继承。

3.3.2 改变句柄的标志

有时会遇到这样一种情况，父进程创建一个内核对象，以便检索可继承的句柄，然后生成两个子进程。父进程只想要一个子进程来继承内核对象的句柄。换句话说，有时可能想要控制哪个子进程来继承内核对象的句柄。若要改变内核对象句柄的继承标志，可以调用SetHandleInformation函数：

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

可以看到，该函数拥有3个参数。第一个参数hObject用于标识一个有效的句柄。第二个参数dwMask告诉该函数想要改变哪个或那几个标志。目前有两个标志与每个句柄相关联：

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

如果想同时改变该对象的两个标志，可以逐位用 OR将这些标志连接起来。 SetHandle Information函数的第三个参数是dwFlags，用于指明想将该标志设置成什么值。例如，若要打开一个内核对象句柄的继承标志，请创建下面的代码：

```
SetHandleInformation(hobj, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT);
```

若要关闭该标志，请创建下面的代码：

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

HANDLE_FLAG_PROTECT_FROM_CLOSE标志用于告诉系统，该句柄不应该被关闭：

```
SetHandleInformation(hobj, HANDLE_FLAG_PROTECT_FROM_CLOSE,
   HANDLE_FLAG_PROTECT_FROM_CLOSE);
CloseHandle(hobj);   // Exception is raised
```

如果一个线程试图关闭一个受保护的句柄， CloseHandle就会产生一个异常条件。很少想要将句柄保护起来，使他人无法将它关闭。但是如果一个进程生成了子进程，而子进程又生成了孙进程，那么该标志可能有用。父进程可能希望孙进程继承赋予子进程的对象句柄。不过，子进程有可能在生成孙进程之前关闭该句柄。如果出现这种情况，父进程就无法与孙进程进行通信，因为孙进程没有继承该内核对象。通过将句柄标明为“受保护不能关闭”，那么孙进程就能继承该对象。

但是这种处理方法有一个问题。子进程可以调用下面的代码来关闭 HANDLE_FLAG_PROTECT_FROM_CLOSE标志，然后关闭句柄。

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

父进程可能打赌说，子进程将不执行该代码。当然，父进程也可能打赌说，子进程将生成孙进程。因此这种打赌没有太大的风险。

为了完整地说明问题，也要讲一下GetHandleInformation函数的情况：

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

该函数返回pdwFlags指向的DWORD中特定句柄的当前标志的设置值。若要了解句柄是否是可继承的，请使用下面的代码：

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

3.3.3 命名对象

共享跨越进程边界的内核对象的第二种方法是给对象命名。许多（虽然不是全部）内核对象都是可以命名的。例如，下面的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);
```

所有这些函数都有一个共同的最后参数pszName。当为该参数传递NULL时，就向系统指明了想创建一个未命名的（匿名）内核对象。当创建一个未命名的对象时，可以通过使用继承

性（如上一节介绍的那样）或DuplicateHandle(下一节将要介绍)共享跨越进程的对象。若要按名字共享对象，必须为对象赋予一个名字。

如果没有为pszName参数传递MULL，应该传递一个以0结尾的字符串名字的地址。该名字的长度最多可以达到MAX_PATH（定义为260）个字符。但是，Microsoft没有提供为内核对象赋予名字的指导原则。例如，如果试图创建一个称为“ JeffObj ”的对象，那么不能保证系统中不存在一个名字为“ JeffObj ”的对象。更为糟糕的是，所有这些对象都共享单个名空间。由于这个原因，对下面这个CreateSemaphore函数的调用将总是返回NULL：

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

如果在执行上面的代码后观察 dwErrorcode的值，会看到返回的代码是 6（ERROR_INVALID_HANDLE)。这个错误代码没有很强的表意性，但是你又能够做什么呢？

既然已知道如何给对象命名，那么让我们来看一看如何用这种方法来共享对象。比如说，Process A启动运行，并调用下面的函数：

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

调用该函数能创建一个新的互斥内核对象，为它赋予名字“ JeffMutex ”。请注意，Process A的句柄hMutexProcessA不是一个可继承的句柄，并且当你只是命名对象时，它不必是个可继承的句柄。

过些时候，某个进程会生成Process B。Process B不一定是Process A的子进程。它可能是Explorer或其他任何应用程序生成的进程。 Process B不必是Process A的子进程这一事实正是使用命名对象而不是继承性的优越性。当Process B启动运行时，它执行下面的代码：

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

当Process B调用CreateMutex时，系统首先要查看是否已经存在一个名字为“ JeffMutex ”的内核对象。由于确实存在一个带有该名字的对象，因此内核要检查对象的类型。由于试图创建一个互斥对象，而名字为“ JeffMutex ”的对象也是个互斥对象，因此系统会执行一次安全检查，以确定调用者是否拥有对该对象的完整的访问权。如果拥有这种访问权，系统就在Process B的句柄表中找出一个空项目，并对该项目进行初始化，使该项目指向现有的内核对象。如果该对象类型不匹配，或者调用者被拒绝访问，那么CreateMutex将运行失败（返回NULL)。

当Process B对CreateMutex的调用取得成功时，它并不实际创建一个互斥对象。相反，Process B只是被赋予一个与进程相关的句柄值，用于标识内核中现有的互斥对象。当然，由于Process B的句柄表中的一个新项目要引用该对象，互斥对象的使用计数就会递增。在Process A和Process B同时关闭它们的对象句柄之前，该对象是不会被撤消的。请注意，这两个进程中的句柄值很可能是不同的值。这是可以的。 Process A将使用它的句柄值，而Process B则使用它自己的句柄值来操作一个互斥内核对象。

注意 当你的多个内核对象拥有相同的名字时，有一个非常重要的细节必须知道。当Process B调用CreateMutex时，它将安全属性信息和第二个参数传递给该函数。如果已经存在带有指定名字的对象，那么这些参数将被忽略。应用程序能够确定它是否确实创建了一个新内核对象，而不是打开了一个现有的对象。方法是在调用 Create*函数后立即调用GetLastError：

```
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.
   }
```

按名字共享对象的另一种方法是，进程不调用Create*函数，而是调用下面显示的Open*函数中的某一个：

```
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，所有这些函数都拥有相同的原型。最后一个参数pszName用于指明内核对象的名字。不能为该参数传递NULL，必须传递以0结尾的地址。这些函数要搜索内核对象的单个名空间，以便找出匹配的空间。如果不存在带有指定名字的内核对象，该函数返回NULL，GetLastError返回2（ERROR_FILE_NOT_FOUND）。但是，如果存在带有指定名字的内核对象，并且它是相同类型的对象，那么系统就要查看是否允许执行所需的访问（通过dwDesiredAccess参数进行访问）。如果拥有该访问权，调用进程的句柄表就被更新，对象的使用计数被递增。如果为bInheritHandle参数传递TRUE，那么返回的句柄将是可继承的。

调用Create*函数与调用Open*函数之间的主要差别是，如果对象并不存在，那么Create*函数将创建该对象，而Open*函数则运行失败。

如前所述，Microsoft没有提供创建唯一对象名的指导原则。换句话说，如果用户试图运行来自不同公司的两个程序，而每个程序都试图创建一个称为“MyObject”的对象，那么这就是个问题。为了保证对象的唯一性，建议创建一个GUID，并将GUID的字符串表达式用作对象名。命名对象常常用来防止运行一个应用程序的多个实例。若要做到这一点，只需要调用main或

WinMain函数中Create*函数，以便创建一个命名对象（创建的是什么对象则是无所谓的）。当Create*函数返回时，调用GetLastError函数。如果GetLastError函数返回ERROR_ALREADY_EXISTS，那么你的应用程序的另一个实例正在运行，新实例可以退出。下面是说明这种情况的部分代码：

```
int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE, PSTR pszCmdLine,
   int nCmdShow) {
   HANDLE h = CreateMutex(NULL, FALSE,
      "{FA531CC1-0497-11d3-A180-00105A276C3E}");
   if (GetLastError() == ERROR_ALREADY_EXISTS) {
      // There is already an instance of this application running.
      return(0);
   }

   // This is the first instance of this application running.
   :

   // Before exiting, close the object.
   CloseHandle(h);
   return(0);
}
```

3.3.4 终端服务器的名字空间

注意，终端服务器能够稍稍改变上面所说的情况。终端服务器拥有内核对象的多个名字空间。如果存在一个可供内核对象使用的全局名字空间，就意味着它可以供所有的客户程序会话访问。该名字空间主要供服务程序使用。此外，每个客户程序会话都有它自己的名字空间。它能防止运行相同应用程序的两个或多个会话之间出现互相干扰的情况，也就是说一个会话无法访问另一个会话的对象，尽管该对象拥有相同的名字。在没有终端服务器的机器上，服务程序和应用程序拥有上面所说的相同的内核对象名字空间，而在拥有终端服务器的机器上，却不是这样。

服务程序的名字空间对象总是放在全局名字空间中。按照默认设置，在终端服务器中，应用程序的命名内核对象将放入会话的名字空间中。但是，如果像下面这样将“Global\”置于对象名的前面，就可以使命名对象进入全局名字空间：

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

也可以显式说明想让内核对象进入会话的名字空间，方法是将“Local\”置于对象名的前面：

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

Microsoft将Global和Local视为保留关键字，除非要强制使用特定的名字空间，否则不应该使用这两个关键字。Microsoft还将Session视为保留关键字，虽然目前它没有任何意义。请注意，所有这些保留关键字是区分大小写字母的。如果主机不运行终端服务器，这些关键字将被忽略。

3.3.5 复制对象句柄

共享跨越进程边界的内核对象的最后一个方法是使用DuplicateHandle函数：

```
BOOL DuplicateHandle(
   HANDLE hSourceProcessHandle,
```

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

简单说来，该函数取出一个进程的句柄表中的项目，并将该项目拷贝到另一个进程的句柄表中。DuplicateHandle函数配有若干个参数，但是实际上它是非常简单的。DuplicateHandle函数最普通的用法要涉及系统中运行的3个不同进程。

当调用DuplicateHandle函数时，第一和第三个参数hSourceProcessHandle和hTargetProcessHandle是内核对象句柄。这些句柄本身必须与调用DuplicateHandle函数的进程相关。此外，这两个参数必须标识进程的内核对象。如果将句柄传递给任何其他类型的内核对象，那么该函数运行就会失败。第4章将详细介绍进程的内核对象，而现在只需要知道，每当系统中启动一个新进程时都会创建一个进程内核对象。

第二个参数hSourceHandle是任何类型的内核对象的句柄。但是该句柄值与调用DuplicateHandle的进程并无关系。相反，该句柄必须与hSourceProcessHandle句柄标识的进程相关。第四个参数phTargetHandle是HANDLE变量的地址，它将接收获取源进程句柄信息拷贝的项目索引。返回的句柄值与hTargetProcessHandle标识的进程相关。

DuplicateHandle的最后3个参数用于指明该目标进程的内核对象句柄表项目中使用的访问屏蔽值和继承性标志。dwOptions参数可以是0（零），也可以是下面两个标志的任何组合：DUPLICATE_SAME_ACCESS和DUPLICATE_CLOSE_SOURCE。

如果设定了DUPLICATE_SAME_ACCESS标志，则告诉DuplicateHandle函数，你希望目标进程的句柄拥有与源进程句柄相同的访问屏蔽。使用该标志将使DuplicateHandle忽略它的dwDesiredAccess参数。

如果设定了DUPLICATE_CLOSE_SOURCE标志，则可以关闭源进程中的句柄。该标志使得一个进程能够很容易地将内核对象传递给另一个进程。当使用该标志时，内核对象的使用计数不会受到影响。

下面用一个例子来说明DuplicateHandle函数是如何运行的。在这个例子中，Process S是目前可以访问某个内核对象的源进程，Process T是将要获取对该内核对象的访问权的目标进程。Process C是执行对DuplicateHandle调用的催化进程。

Process C的句柄表（表3-4）包含两个句柄值，即1和2。句柄值1用于标识Process S的进程内核对象，句柄值2则用于标识Process T的进程内核对象。

表3-4 Process C的句柄表

索 引	内核对象内存块的指针	访问屏蔽（标志位的DWORD）	标 志（标志位的DWORD）
1	0xF0000000 (Process S的内核对象)	0x????????	0x00000000
2	0xF0000010 (Process T的内核对象)	0x????????	0x00000000

表3-5是Process S的句柄表，它包含句柄值为2的单个项目。该句柄可以标识任何类型的内核对象，就是说它不必是进程的内核对象。

表3-5 Process S的句柄表

索 引	内核对象内存块的指针	访问屏蔽（标志位的DWORD）	标 志（标志位的DWORD）
1	0x00000000	(无)	(无)
2	0xF0000020 (任何内核对象)	0x????????	0x00000000

表3-6显示了Process C调用DuplicateHandle函数之前Process T的句柄表包含的项目。如你所见，Process T的句柄表只包含句柄值为2的单个项目，句柄项目1目前未用。

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

索 引	内核对象内存块的指针	访问屏蔽（标志位的DWORD）	标 志（标志位的DWORD）
1	0x00000000	(无)	(无)
2	0xF0000030 (任何内核对象)	0x????????	0x00000000

如果Process C现在使用下面的代码来调用DuplicateHandle，那么只有Process T的句柄表改变更，如表3-7所示。

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

索 引	内核对象内存块的指针	访问屏蔽（标志位的DWORD）	标 志（标志位的DWORD）
1	0xF00000020	0x????????	0x00000001
2	0xF00000030 (任何内核对象)	0x????????	0x00000000

Process S的句柄表中的第二项已经被拷贝到 Process T的句柄表中的第一项。 DuplicateHandle也已经将值1填入Process C的hObj变量中。值1是Process T的句柄表中的索引，新项目将被放入该索引。

由于DUPLICATE_SAME_ACCESS标志被传递给了DuplicateHandle，因此Process T的句柄表中该句柄的访问屏蔽与 Process S的句柄表项目中的访问屏蔽是相同的。另外，传递DUPLICATE_SAME_ACCESS标志将使DuplicateHandle忽略它的DesiredAccess参数。最后请注意，继承位标志已经被打开，因为给DuplicateHandle的bInheritHandle参数传递的是TRUE。

显然，你永远不会像在这个例子中所做的那样，调用传递硬编码数字值的 DuplicateHandle函数。这里使用硬编码数字，只是为了展示函数是如何运行的。在实际应用程序中，变量可能拥有各种不同的句柄值，可以传递该变量，作为函数的参数。

与继承性一样，DuplicateHandle函数存在的奇怪现象之一是，目标进程没有得到关于新内核对象现在可以访问它的通知。因此， Process C必须以某种方式来通知Process T,它现在拥有对内核对象的访问权，并且必须使用某种形式的进程间通信方式，以便将 hObj中的句柄值传递给Process T。显然，使用命令行参数或者改变Process T的环境变量是不行的，因为该进程已经启动运行。因此必须使用窗口消息或某种别的IPC机制。

上面是DuplicateHandle的最普通的用法。如你所见，它是个非常灵活的函数。不过，它很少在涉及3个不同进程的情况下被使用（因为 Process C不可能知道对象的句柄值正在被 Process

S使用)。通常，当只涉及两个进程时，才调用DuplicateHandle函数。比如一个进程拥有对另一个进程想要访问的对象的访问权，或者一个进程想要将内核对象的访问权赋予另一个进程。例如，Process S拥有对一个内核对象的访问权，并且想要让Process T能够访问该对象。若要做到这一点，可以像下面这样调用DuplicateHandle:

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

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

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

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

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

// Use some IPC mechanism to get the handle
// value in hObjProcessS 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(hObjProcessS);
```

在这个例子中，对GetCurrentProcess的调用将返回一个伪句柄，该句柄总是用来标识调用端的进程Process S。一旦DuplicateHandle返回，hObjProcessT就是与Process T相关的句柄，它所标识的对象与引用Process S中的代码时hObjProcessS的句柄标识的对象相同。Process S决不应该执行下面的代码：

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

如果Process S要执行该代码，那么对代码的调用可能失败，也可能不会失败。如果Process S恰好拥有对内核对象的访问权，其句柄值与hObjProcessT的值相同，那么调用就会成功。该代码的调用将会关闭某个对象，这样Process S就不再拥有对它的访问权，这当然会导致应用程序产生不希望有的行为特性。

下面是使用DuplicateHandle函数的另一种方法。假设一个进程拥有对一个文件映射对象的读和写访问权。在某个位置上，一个函数被调用，它通过读取文件映射对象来访问它。为了使应用程序更加健壮，可以使用DuplicateHandle为现有的对象创建一个新句柄，并确保这个新句柄拥有对该对象的只读访问权。然后将只读句柄传递给该函数，这样，该函数中的代码就永远不会偶然对该文件映射对象执行写入操作。下面这个代码说明了这个例子：

```
int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE,
   LPSTR 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);
}
```