第3章　Win32 API

本章内容：

- 对象 —— 以前和现在
- 多任务和多线程
- Win32内存管理
- Win32的错误处理

本章介绍Win32 API和Win32系统。还要讨论Win32系统的功能以及它与16位系统在功能上的几个主要区别。本章内容不是关于Win32系统的详细文档，而只是让对Win32系统有一个基本的了解。当已经基本了解Win32操作后，就可以在任何需要的时候使用Win32系统提供的高级功能了。

3.1　对象：以前和现在

对象这个术语可以用于不同的场合。我们谈论 Win32的体系结构时，实际上并不是指面向对象编程或组件对象模型(COM)中的对象。在不同的场合对象具有完全不同的含义，甚至 16位Windows中的对象与Win32中的对象也有细小的差别，这使对象的概念很容易混淆。我们在这里重点介绍 Win32中的对象。

Win32环境中有两种基本的对象类型：内核对象和GDI/用户对象。

3.1.1　内核对象

内核对象是Win32系统原有的，包括事件、文件映射、文件、邮件槽、互斥、管道、进程、信号灯和线程。Win32 API包含有针对不同内核对象的函数。在讨论内核对象之前，我们先要讨论进程的概念，因为这是理解Win32环境如何管理对象的基础。

1. 进程和线程

可以认为一个进程就是一个正在运行的应用程序或一个应用程序的实例。因此，在 Win32环境中可以同时激活几个进程。每个进程可以为它的代码和数据获得 4GB的地址空间。在这4GB地址空间中，存在着已分配的内存、线程、文件映射等。另外，由进程调用的动态链接库也在进程的地址空间中。这章后面的3.3节“ Win32内存管理 ”将详细介绍Win32的内存管理。

进程是惰性的。换句话说，进程本身并不执行任何代码。然而，每个进程拥有一个主线程，由主线程在进程的环境中执行代码。一个进程可以拥有几个线程；但是，一个进程只能有一个主线程。

> **注意**　线程是一种操作系统对象，代表着一个进程中要被执行的代码的路径。每一个Win32应用程序至少有一个线程 —— 通常称为主线程或默认线程 —— 但应用程序可以自由地创建其他线程来执行其他任务。第11章“ 编写多线程应用程序 ”将进一步介绍线程的用法。

当一个进程被创建后，系统就会为它创建一个主线程。如果需要的话，该主线程还可以创建其他线程。Win32系统把CPU时间片分配给这些线程。

表3-1显示了Win32 API中有关进程的函数。

表3-1 进程函数

函 数	用 途
CreateProcess()	创建进程及其主线程。该函数代替Windows 3.11中使用的WinExec()函数
ExitProcess()	退出当前进程，并终止与此进程有关的所有线程
GetCurrentProcess()	返回当前进程的句柄。但该句柄只能认为是当前进程句柄的副本。真正的进程句柄是通过调用DuplicateHandle()函数获得的
DuplicateHandle()	复制一个内核对象的句柄
GetCurrentProcessID()	返回当前进程的ID，它在整个系统中唯一地标识一个进程，直到该进程终止
GetExitCodeProcess()	返回指定进程的退出状态
GetPriorityClass()	返回指定进程的优先级类别。它和每个线程的优先级类别共同决定了该线程的基本优先级
GetStartupInfo()	返回进程创建时被初始化的TStartupInfo结构的内容
OpenProcess()	返回用进程ID指定的进程的句柄
SetPriorityClass()	设置进程的优先级类别
TerminateProcess()	终止一个进程及相关的所有线程
WaitForInputIdle()	等待，直到进程正在等待用户输入

有的Win32 API函数需要传递应用程序实例的句柄，有的函数需要传递模块的句柄。在 16位Windows中，这两者之间是有区别的。而在Win32环境中则没有区别。每一个进程可以获得它自己的实例句柄。在Delphi 5应用程序中可以通过访问一个叫 HInstance的全局变量来获取这个句柄。因为HInstance和应用程序的模块句柄是同一个概念，可以把 HInstance传递给那些需要模块句柄的 Win32 API函数，例如GetModuleFileName()函数，它返回一个指定模块的文件名。注意下面的提示，有时HInstance并不代表当前应用程序的模块句柄。

警告 如果一个程序编译成包，那么 HInstance并不代表应用程序的模块句柄。此时要用MainInstance来访问宿主程序的模块句柄，而HInstance代表代码所在模块的句柄。

Win32和16位Windows的另一个区别是全局变量HPrevInst。在16位Windows中，这个变量代表先前运行的应用程序实例句柄。可以利用这个变量来避免同时运行应用程序的多个实例。 Win32中已经没有这个变量了。每个进程都运行在自己的4GB的地址空间中，相互之间不能看到。这样， HPrevInst变量总是被赋值为0。必须使用其他方法来避免同时运行应用程序的多个实例，请参见第 13章“核心技术”。

2. 内核对象的类型

有若干种内核对象。当一个内核对象被创建后，它存在于进程的地址空间，进程可以获得这个对象的句柄。这个句柄不能被传递给其他进程或被下一个需要访问同一内核对象的进程重用。然而，下一个进程可以通过调用合适的Win32 API函数获得这个内核对象的句柄。例如， CreateMutex()函数能够创建一个有名或无名的互斥对象，并且返回它的句柄。 OpenMutex()函数可以返回一个有名称的互斥对象的句柄。OpenMutex()函数需传递要返回句柄的互斥对象名称。

注意 当调用CreateXXXX()函数创建内核对象时，可以用一个以null结束的字符串给内核对象命名。内核对象的名称注册在Win32系统中。其他进程可以调用OpenXXXX()函数并传递对象名称来打开该内核对象。第13章“核心技术”的技术应用范例介绍了如何避免同时运行应用程序的多个实例。

如果想在进程间共享互斥对象，首先在一个进程中调用 CreateMutex()函数创建它。这个进程必须为新创建的互斥对象传递名称。其他进程要使用 OpenMutex()函数，并且传递第一个进程使用的对象

名称。OpenMutex()函数将返回一个指定名称的互斥对象的句柄。其他进程访问存在的内核对象时将被加以不同的安全约束。这些安全约束是在互斥对象初始创建时指定的。可以在在线帮助中查找有关内核对象安全约束的主题。

由于多个进程可以共享内核对象，因此系统用一个计数器来维护它。当第二个应用程序访问内核对象时，这个计数器就加1。当这个应用程序结束对内核对象的使用时，它应该调用 CloseHandle()函数来使计数器减1。

3.1.2 GDI和用户对象

16位Windows中的对象是指能够被句柄引用的实体。这不包括内核对象，因为 16位Windows中还没有内核对象的概念。

在16位Windows中，有两种类型的对象：一种是存储在 GDI和用户局部堆中的对象，例如画刷、画笔、字体、调色板、位图和区域；另一种是分配于全局堆中的对象，例如窗口、窗口类、原子和菜单。

对象与它的句柄之间存在着直接的关系。一个对象的句柄实际上是一个指针，这个指针指向构成对象的数据。依赖于不同的对象类型，对象数据存储于 GDI或用户数据段中。另外，对于分配于全局堆的对象，它们的句柄也是指针，指向全局内存段。

这样特殊设计的原因是 16位Windows中的对象是共享的。全局可访问的 LDT(局部描述符表)中存储了所有对象的句柄。在 16位Windows中，GDI和用户数据段对于所有的应用程序或 DLL也是全局可访问的。因此，任何应用程序或 DLL都可以得到其他应用程序的对象。需要注意的是， LDT只在16位Windows中是共享的。许多应用程序出于不同的目的使用这种安排，其中一个目的是使应用程序能够共享内存。

Win32处理GDI和用户对象就有些不同。原来在 16位Windows中使用的技术在Win32环境下可能无法使用。

首先， Win32引入了内核对象的概念，在前面我们已经介绍过。而且， GDI和用户对象的实现也与16位Windows有些不同。

在Win32下，GDI对象不像在16位环境中那样可以共享。GDI对象存储在进程的地址空间而不是全局可访问的内存块中(每个进程有4GB的地址空间)。另外，每个进程有它自己的句柄表来存储该进程内的GDI对象的句柄。这是非常重要的一点，因为并不想把GDI对象的句柄传递给其他进程。

前面我们提到， LDT是所有应用程序都可以访问的。而在 Win32中，每个进程的地址空间是由它自己的LDT定义的。因此， Win32中的LDT只能在本进程中使用。

注意 尽管一个进程可以用SelectObject()函数调用另一个进程的句柄，而且能成功使用这个句柄，然而这完全是一种巧合。GDI对象在不同的进程中有不同的含义，所以最好不要使用这种方法。

Win32 GDI子系统对GDI句柄的管理包含两个方面，一个是对GDI对象的校验，另一个是句柄的重复使用。

用户对象和GDI对象有些类似，它是由Win32用户子系统管理的。然而，用户对象的句柄不像 GDI对象那样存储于进程的地址空间，而是有一个专门的用户句柄表。因此，像窗口、窗口类、原子等对象可以在不同的进程之间共享。

3.2 多任务和多线程

多任务是指操作系统可以同时运行多个应用程序。操作系统把 CPU的时间分成片分配给每个应用

程序。在这种情况下，多任务其实并不是真正的多任务，只能说是任务切换。或者说，操作系统并没有真正同时运行多个应用程序。相反，它先运行一个应用程序一定的时间，再切换到另一个应用程序运行一定的时间。它对每个应用程序都这样处理。因为时间被划分得很短，对于用户来说，就好像多个应用程序在同时运行一样。

多任务的机制并不是Windows的新功能，它的早期版本就已经有了。 Win32与它的早期版本在实现多任务方面的关键区别是， Win32使用有优先级的多任务，而早期版本则只实现了无优先级的多任务(这意味着Windows并不是按照系统时钟来给应用程序安排时间)。在应用程序结束并通知Windows之前，操作系统无法把时间分配给其他应用程序。这样存在一个问题，如果一个应用程序占用的时间过长，就可能使操作系统长时间被挂起。因此，除非程序员确保他的应用程序能够把时间让给其他应用程序，否则用户就会遇到麻烦。

在Win32下，操作系统把CPU时间划分为片分配给每个线程。Win32系统基于每个线程的优先级来管理时间的分配。第11章“编写多线程应用程序”将详细介绍多线程的概念。

注意 Windows NT/2000允许使用多处理器，从而实现了真正的多任务。在这种情况下，每个应用程序由各自的处理器分配时间。实际上，每个单独的线程能够从任何可用的处理器上获取CPU时间。

多线程是指一个应用程序内部的多任务。这意味着应用程序可以同时进行不同类型的处理。一个进程可以有几个线程，每个线程都有各自不同的执行代码。一个线程可能要依赖于另一个线程，这样必须要同步。例如，不能假设一个线程在另一个线程要使用它的结果时已完成了处理。线程同步技术用于使多个线程能够同步执行。

第11章“编写多线程应用程序”将进一步讨论线程。

3.3 Win32内存管理

Win32环境引入了32位线性内存模式。结果， Pascal程序员可以声明一个很大的数组而不会导致编译错误：

```
BigArray = array[1..100000] of integer;
```

下面介绍Win32内存模式以及怎样在Win32系统下管理内存。

3.3.1 什么是线性内存模式

16位环境使用分段的内存模式。在这种模式下，地址用 segment:offset来表示。Segment代表基地址，offset代表从基地址开始的偏移量。这种模式带来的问题是，初级的程序员很容易混淆，尤其是涉及大内存需求的时候。这种模式还有一个限制，当数据结构超过64KB时，程序员将非常难以管理。

在线性内存模式下，这些限制都不复存在。每个进程都有自己的 4GB的地址空间，可以安排很大的数据结构。另外，每个地址总是代表一个唯一的内存位置。

3.3.2 Win32系统是怎样管理内存的

你的计算机不太可能安装4GB的物理内存。那么， Win32系统是怎样获得比实际安装的物理内存大得多的地址空间的？32位的地址并不真正代表物理内存的一个位置，其实Win32使用的是虚拟地址。

通过虚拟地址，每一个进程可以获得4GB的虚拟地址空间。上端的2MB空间属于Windows，下端的2MB空间是放置应用程序及可以分配内存的地方。这种模式的优势在于一个进程中的线程不能访问其他进程的内存。同样的地址$54545454在不同的进程中指向不同的位置。

一个进程并不是真的有4GB的内存而只是具有访问4GB内存的能力，注意到这一点是很重要的。

一个进程真正能访问的内存大小取决于计算机安装了多少物理内存以及磁盘上有多少空间可被页交换文件使用。对于一个进程来说，物理内存和页交换文件是按页来划分使用的。页的大小取决于 Win32 安装在什么类型的系统上。在Intel的平台上，每页的长度是4KB；在Alpha平台上，每页的长度是8KB。对于PowerPC和MIPS平台来说，每页的长度也是4KB。系统会把页从页交换文件移到物理内存中，需要的时候再移回来。系统会维护进程当中虚拟地址与物理地址之间的映射关系。我们不想涉及更多的细节，只要了解这些就够了。

在Win32环境下，程序员用3种方式有效地使用内存：虚拟内存、文件映射对象和堆。

1. 虚拟内存

Win32为使用虚拟内存提供了一些底层的函数。内存一般处于下列3种状态之一：

- Free 内存是自由的，可以被保留或分配。
- Reserved 一定范围的内存被保留，以便将来使用。在这段范围之内的内存不能被其他应用程序分配。但是，在被分配之前这段内存也不能被进程访问，因为没有物理内存与它关联。函数VirtualAlloc()可以用于保留内存。
- Committed 内存已经被分配并与物理内存关联。被分配的内存可以被进程访问。函数 VirtualAlloc()可以用于分配虚拟内存。

前面提到，Win32提供了不同的VirtualXXXX()函数来使用内存，如表3-2所列。这些函数在在线帮助中有详细的介绍。

表3-2 虚拟内存函数

函 数	用 途
VirtualAlloc()	在进程的虚拟地址空间中保留和/或分配页
VirtualFree()	释放进程虚拟地址空间中的页
VirtualLock()	锁定进程虚拟地址空间的一段区域，使其不能被交换到磁盘的分页文件中
VirtualUnLock()	解除对指定区域的内存的锁定，使其可以被交换到分页文件中
VirtualQuery()	返回有关进程虚拟地址空间的信息
VirtualQueryEx()	作用与VirtualQuery()类似，但它可以指定进程
VirtualProtect()	修改进程虚拟地址空间中已分配页的保护模式
VirtualProtectEx()	作用与VirtualProtect()类似，但它可以指定进程

注意 表3-2中的XXXEx()函数只适用于那些具有调试其他进程权限的进程。这些函数比较复杂且很少用，一般只有调试器才会调用。

2. 内存映射文件

内存映射文件(文件映射对象)允许像访问动态分配的内存那样访问磁盘文件。这是通过映射文件的全部或部分到进程的地址空间来实现的。这样，只要使用一个指针，就可以访问文件中的数据。

第12章“文件处理”将进一步讨论内存映射文件。

3. 堆

堆是可以被分配的小块连续内存。使用堆能够有效地分配和使用动态内存。 Win32 API函数HeapXXXX()用于使用堆。这些函数列在表3-3中，在线帮助中也有这些函数的详细介绍。

表3-3 堆函数

函 数	用 途
HeapCreate()	在虚拟地址空间中保留一块连续的地址，并且在物理内存中为堆首部分配空间
HeapAlloc()	在一个堆中分配一块不可移动的内存
HeapReAlloc()	在一个堆中重新分配一块内存，可以改变堆的大小和属性

(续)

函 数	用 途
HeapFree()	从堆中释放用HeapAlloc()分配的内存
HeapDestroy()	删除用HeapCreate()创建的堆

注意 应当注意Windows NT/2000和Windows 95/98在Win32的实现上有一些区别。这些区别通常影响到安全性和速度。例如，Windows 95/98的内存管理功能不如Windows NT那么强大(NT可以提供更多的堆信息)。而且NT的虚拟内存管理也要比Windows 95/98快得多。

当调用相关Windows对象的函数时，需要知道这些区别。在线帮助中将专门指出每个函数在不同平台上的用法。无论何时使用这些函数都要参考帮助。

3.4 Win32的错误处理

大多数Win32 API函数都返回True或False，以表明函数调用成功或失败。如果函数调用不成功 (返回False)，必须使用GetLastError()函数来获得出错线程的错误代码。

注意 并非所有的Win32 API函数都可以用GetLastError()获取错误代码。例如，许多GDI例程就没有错误代码。

错误代码是与线程有关的，因此GetLastError()函数必须在出错线程的环境中调用。下面的代码演示了该函数的用法：

```
if not CreateProcess(CommandLine, nil, nil, nil, False,
  NORMAL_PRIORITY_CLASS, nil, nil, StartupInfo, ProcessInfo) then
    raise Exception.Create('Error creating process: '+
    IntToStr(GetLastError));
```

提示 Delphi 5的SysUtils.pas单元提供了一个标准的异常类和两个工具函数，可以将系统错误转化为异常。这两个函数是Win32Check()和RaiseLastWin32Error()，能够触发EWin32Error异常。可以利用这些例子程序进行自己的错误处理。

上述代码试图以空结尾的字符串CommandLine创建一个进程。我们将会在后面的章节里进一步讨论CreateProcess()的用法。如果函数CreateProcess()失败，就触发一个异常，并调用GetLastError()函数显示错误代码。也可以在自己的程序中使用类似的方法。

提示 在线帮助文档中函数说明的下面列出了该函数失败时由GetLastError()返回的错误代码。因此，CreateMutex()函数的错误代码就在Win32在线帮助中CreateMutex()函数说明的下面。

3.5 总结

本章介绍了Win32 API。你现在应该对内核对象和Win32的内存管理有了一定的了解，并且熟悉了不同的内存管理方法。作为一个Delphi开发者，不必知道Win32系统的细节。但是，应该大致了解Win32系统、它的函数以及怎样使用这些函数增强开发效果。本章仅是一个起点。