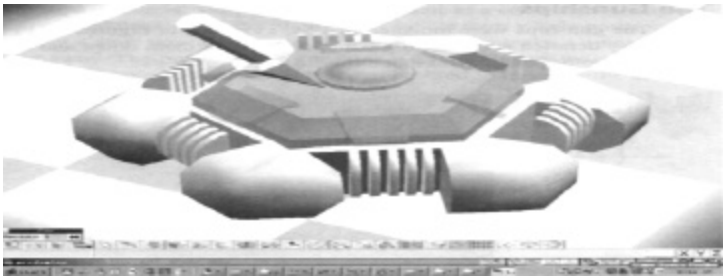


2

Windows 编程模型

Windows 编程就像去见牙科医生：明知道对你有好处，但就是没有人乐意去。是不是这样？在本章中，我将要使用‘禅’的方法——或者换句话说，就是深入浅出地向你介绍 Windows 编程基础。我可不能保证在阅读了本章后你就会“去见牙科医生”，但是我敢保证你会比以往更喜欢 Windows 编程。下面是本章的内容：

- Windows 的历史
- Windows 的基本风格
- Windows 的类
- 创建 Windows
- Windows 事件句柄
- 事件驱动编程和事件循环
- 打开多个窗口

Windows 的历史

读者可能会因为我要解放你的思想而感到十分恐惧（特别是钟情于 DOS 的顽固分子）。让我们迅速浏览一下 Windows 的发展历程以及与游戏发展的关系，好吗？

早期的 Windows 版本

Windows 的发展始于 Windows 1.0 版本。这是 Microsoft 公司在商业视窗操作系统的第

一次尝试，当然是一个非常失败的产品。Windows 1.0 完全建立在 DOS 基础上（这就是一个错误），不能执行多任务，运行速度很慢，看上去也差劲。它的外观可能是其失败的最重要原因。除了讽刺以外，问题还在于 Windows 1.0 与那个时代的 80286 计算机（或更差的 8086）所能提供的相比需要更高的硬件、图像和声音性能。

然而，Microsoft 稳步前进，很快就推出了 Windows 2.0。我记得获得 Windows 2.0 的测试版时我正在软件出版公司工作。在会议室中，挤满了公司的行政官员和董事长（像往常一样，他正拿着一杯鸡尾酒）。我们运行 Windows 2.0 测试演示版，装载了多个应用程序，看上去似乎还在工作。但是，那时 IBM 推出了 PM。PM 看上去要好得多，它是建立在比 Windows 2.0 先进得多的操作系统 OS/2 的基础上的，而 Windows 2.0 依然是建立在 DOS 基础上的视窗管理器。那天会议室中的结论是“不错，但还不是一个可行的操作系统，如果我们仍然留恋在 DOS 上，那我还能有鸡尾酒喝吗？”

Windows 3.x

1990 年，终于发生了翻天覆地的变化，因为 Windows 3.0 出世了，而且其表现的确非常出色！尽管它仍然赶不上 Mac OS 的标准，但是谁还在意呢？（真正的程序员都憎恨 Mac）。软件开发人员终于可以在 PC 机上创建迷人的应用程序了，而商用应用程序也开始脱离 DOS。这成了 PC 机的转折点，终于将 Mac 完全排除在商用应用程序之外了，而后也将其挤出台式机出版业。（那时，Apple 公司每 5 分钟就推出一种新硬件）。

尽管 Windows 3.0 工作良好，却还是存在许多的问题、软件漏洞，但从技术上说它已是 Windows 2.0 之后的巨大突破，有问题也是在所难免。为了解决这些问题，Microsoft 推出了 Windows 3.1，开始公关部和市场部打算称之为 Windows 4.0，但是，Microsoft 决定只简单地称之为 Windows 3.1，因为它还不足以称之为升级的换代版本。它还没有做到市场部广告宣传的那样棒。

Windows 3.1 非常可靠。它带有多媒体扩展以提供音频和视频支持，而且它还是一个出色的、全面的操作系统，用户能够以统一的方式来工作。另外，还存在一些其他的版本，如可以支持网络的 Windows 3.11（适用于工作组的 Windows）。惟一的问题是 Windows 3.1 仍然是一个 DOS 应用程序，运行于 DOS 扩展器上。

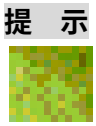
Windows 95

另一方面，游戏编程行业还在唱“DOS 永存！”的赞歌，而我则已经开始热衷于使用 Windows 3.1。但是，1995 年世界开始冷却——Windows 95 终于推出。它是一个真正 32 位的、多任务、多线程的操作系统。诚然，其中还残存一些 16 位代码，但在极大程度上，Windows 95 是 PC 机的终极开发和发布平台。

（当然，Windows NT 3.0 也同时推出，但是 NT 对于大多数用户来讲还是不可用的，

因此这里也就不再赘述)。

当 Windows 95 推出后,我才真正开始喜欢 Windows 编程。我一直痛恨使用 Windows 1.0、2.0、3.0 和 3.1 来编程,尽管随着每一种版本的推出,这种憎恨都越来越少。当 Windows 95 出现时,它彻底改变了我的思想,如同其他被征服的人的感觉一样——它看上去非常酷!那正是我所需要的。



游戏编程行业中最重要的事情是游戏表现如何,游戏的画面如何。同时还要尽可能减轻审阅人的工作。

因此几乎是一夜间,Windows 95 就改变了整个计算机行业。的确,目前还有一些公司仍然在使用 Windows 3.1(你能相信吗?),但是 Windows 95 使得基于 Intel 的 PC 成为除游戏之外的所有应用程序的选择。不错,尽管游戏程序员知道 DOS 退出游戏编程行业只是一个时间的问题了,但是 DOS 还是它们的核心。

1996 年,Microsoft 公司发布了 Game SDK(游戏软件开发工具包),这基本上就是 DirectX 的第一个版本。这种技术仅能在 Windows 95 环境下工作,但是它的确太慢了,甚至竞争不过 DOS 游戏(如 DOOM 和 Duke Nukem 等)。游戏开发人员继续使用 DOS32 来开发游戏,但是他们知道 DirectX 具有足够快的速度,从而能使游戏能够流畅地运行在 PC 机上已为时不远。

到了 3.0 版,DirectX 的速度在同样计算机上已经和 DOS32 一样快了。到了 5.0 版,DirectX 已经相当完善,实现了该技术最初的承诺。对此我们将在第五章“DirectX 基础和令人生畏的 COM”涉及 DirectX 时再作详细介绍。现在要意识到:Win32/DirectX 是 PC 机上开发游戏的惟一方式。这是历史的选择。

Windows 98

1998 年中期,Windows 98 推出了。这至多是一个改进的版本,而不像 Windows 95 那样是一个换代的产品,但毫无疑问它也是占有很重要的地位。Windows 98 像一辆旧车改装的高速汽车——实际上它是一头皮毛圆润光滑、脚力持久、飞奔跳动的驴。它是全 32 位的,能够支持你想做的所有事情,并具有无限扩充的能力。它很好地集成了 DirectX、3D 图形、网络以及 Internet。

Windows 98 和 Windows 95 相比也非常稳定。当然 Windows98 仍然经常死机,但可以相信的是,这比 Windows 95 少了许多。对即插即用支持得很好,并且能够很好地运行——这只是个时间问题。

Windows NT

现在我们来讨论一下 Windows NT。在本书编写期间，Windows NT 正在推出 5.0 版本。我所能说的是，它最终将取代 Windows 9X 成为每个人的操作系统选择。NT 要比 Windows 9X 严谨得多；而且绝大多数游戏程序员都将在 NT 上开发游戏，这将使 Windows 9X 退出历史舞台。Windows NT 5.0 最酷的是它完全支持即插即用和 Win32/DirectX，因此使用 DirectX 为 Windows 9X 编写的应用程序可以在 Windows NT 5.0 或更高版本上运行。这可是个好消息，因为从历史上看，编写 PC 游戏的开发人员现在具有最大的市场潜力。

那么最低标准是什么呢？如果你使用 DirectX（或其他工具）编写了一个 Win32 应用程序，它完全可以在 Windows 95、98 和 NT 5.0 或更高版本上运行。这可是件好事情。因此你在本书中所学到的任何东西至少可以应用到三种操作系统上，也可以运行于安装 NT 和 DirectX 的其他计算机上，如 DEC 的 Alphas。还有 Windows CE——DirectX 和 Win32 衍生的运行于其他系统上的操作系统。

Windows 的基本风格：Win9X/NT

和 DOS 不同，Windows 是一个多任务的操作系统，允许许多应用程序和更小的程序同时运行，可以最大限度的发挥硬件的性能。这表明 Windows 是一个共享的环境——一个应用程序不可能独占整个系统。尽管 Windows 95、98 和 NT 很相似，但仍然存在许多技术上的差别。但是就我们所涉及的，不可能去详细归纳。这里所参照的 Windows 机器一般是指 Win9X/NT 或 Windows 环境。让我们开始吧！

多任务和多线程

如我所说，Windows 允许不同的应用程序以轮流的方式同时执行，每一个应用程序都占用一段很短的时间段来运行，下一个应用程序轮换运行。如图 2.1 所示，CPU 由几个不同的应用程序以轮流的方式共享。判断出下一个运行的应用程序、分配给每个应用程序的时间量是调度程序的工作。

调度程序可以非常简单——每个应用程序分配固定的运行时间，也可以非常复杂——将应用程序设定为不同的优先级和抢先性或低优先级的事件。就 Win9X/NT 而言，调度程序采用基于优先级的抢先占用方式。这就意味着一些应用程序要比其他的应用程序占用处理器更多的时间，但是如果一个应用程序需要 CPU 处理的话，在另一任务运行的同时，当前的任务可以被锁定或抢先占用。

但是不要对此有太多担心，除非你正在编写 OS（操作系统）或实时代码——其细节事

关重大。大多数情况下，Windows 将执行和调度你的应用程序，无需你参与。

深入接触 Windows，我们可以看到，它不仅是多任务的，而且还是多线程的。这意味着程序由许多更简单的多个执行线程构成。这些线程（像更重要的进程）如程序一样被调度。实际上，在你的计算机上可同时运行 30~50 个线程，执行不同的任务。所以事实上你可能只运行一个程序，但这个程序由一个或多个执行线程构成。

Windows 实际的多线程示意图如图 2.2 所示，从图中可以看到，每一个程序实际上都是由一个主线程和几个工作线程构成。

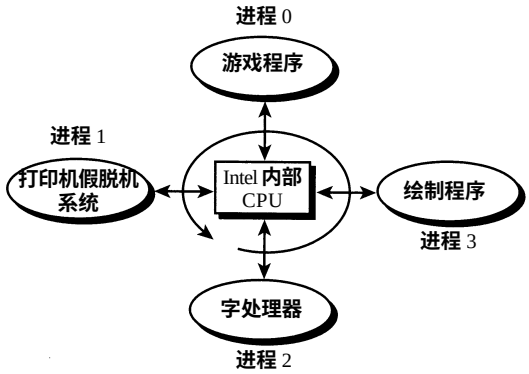


图 2.1　一个处理器执行多个程序的操作

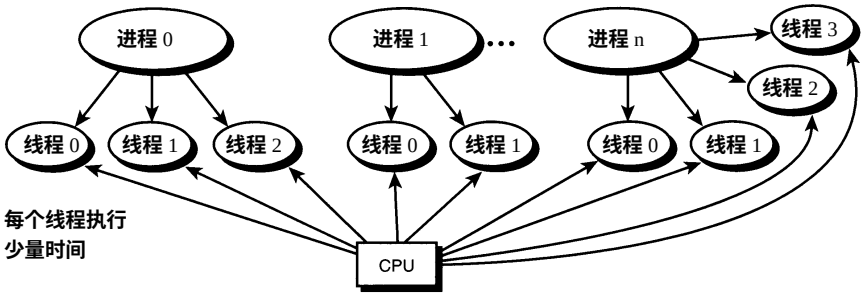


图 2.2　Windows 的更实际的多流程示意图

获取线程的信息

下面让我们来看一下你的计算机现在正在运行多少个线程。在 Windows 机器上，同时按 Ctrl+Alt+Delete 键，弹出显示正在运行的任务（过程）的当前程序任务管理器。这和我们所希望的不同，但也很接近。我们希望的是一个显示正在执行的实际线程数的工具或程序。

许多共享软件和商用软件工具都能做到这一点，但是 Windows 内嵌了这几个工具。

在安装 Windows 的目录（一般是 Windows\）下，可以发现一个名字为 SYSMON.EXE（Windows 95/98）或 PREFMON.EXE（Windows NT）的可执行程序。图 2.3 描述了在我的 Windows 98 机器上运行的 SYSMON.EXE 程序。图中除了正在运行的线程外还有大量的信息，如：内存使用和处理器装载等。实际上在进行程序开发时，我喜欢使 SYSMON.EXE 运行，由此可以了解正在进行什么以及系统如何加载程序。

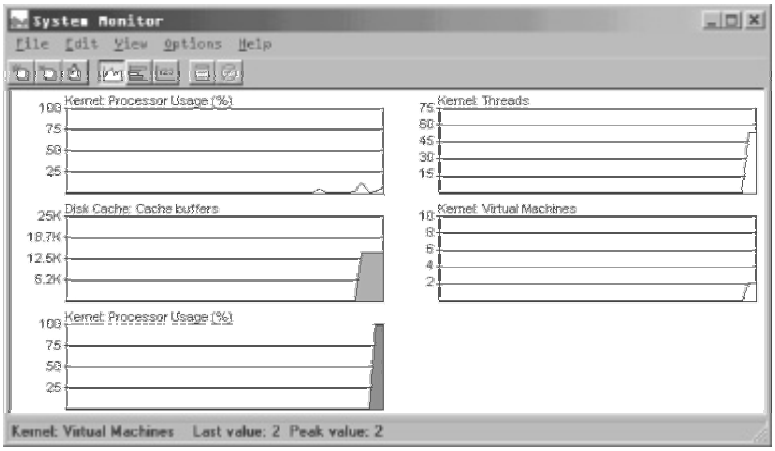


图 2.3 运行的 SYSMON.EXE

你可能想知道能否对线程的创建进行控制，答案是能够！！！实际上这是 Windows 游戏编程最令人激动的事情之一—就像我们所希望的那样除了游戏主进程外，还能够执行其他的任务，我们也能够创建像其他任务一样多的线程。

注 意

在 Windows 98/NT 环境下，实际上还有一种叫 fiber 的新型执行对象，它比线程还简单（明白吗？线程是由 fiber 构成的）。

这和 DOS 游戏程序的编写有很大不同。DOS 是单线程操作系统，也就是说一旦你的程序开始运行，就只能运行该程序（不时出现的中断管理除外）。因此，如果想使用任何一种多任务或多线程，就必须自己来模拟（参阅《Sams Teach Yourself Game Programming in 21 Days》中关于一个完整的基于 DOS 的多任务核心部分）。这也正是游戏程序员在这么多年中所作的事。的确，模拟多任务和多线程远远不能和拥有一个完整的支持多任务和多线程的操作系统相提并论，但是对于单个游戏来讲，它足可以良好地工作。

在我们接触到真正的 Windows 编程和那些工作代码之前，我想提及一个细节。你可能在想，Windows 真是一个神奇的操作系统，因为它允许多个任务和程序立即执行。请记住，实际上并不是这样的。如果只有一个处理器的话，那么一次也只能执行一个执行流、线程、程序或你所调用的任何对象。Windows 相互之间的切换太快了，以至于看上去就像几个程序

在同时运行一样。另一方面，如果有几个处理器的话，可以同时运行多个程序。例如，我有一个双 CPU 的 Pentium II 计算机，有两个 400MHz 的 Pentium II 处理器在运行 Windows NT 5.0。使用这种配置，可以同时执行两个指令流。

我希望在不远的将来，个人计算机的新型微处理器结构能够允许多个线程或 fiber 同时执行，将这样一个目标作为处理器设计的一部分。例如，Pentium 具有两个执行单元——U 管和 V 管。因此它能够立即执行两个指令。但是，这两个指令都是来自同一个线程。类似的是 Pentium II 能够立即执行 5 个简单的指令，但也是来自同一个线程。

事件模型

Windows 是个多任务/多线程 的操作系统，并且还是一个事件驱动的操作系统。和 DOS 程序不同的是，Windows 程序都是等着用户去使用，由此而触发一个事件，然后 Windows 对该事件发生响应，进行动作。请看图 2.4 所示的示意图，图中描述了大量的应用程序窗口，每个程序都向 Windows 发送待处理的事件和消息。Windows 对其中的一些进行处理，大部分的消息和事件被传递给应用程序来处理。

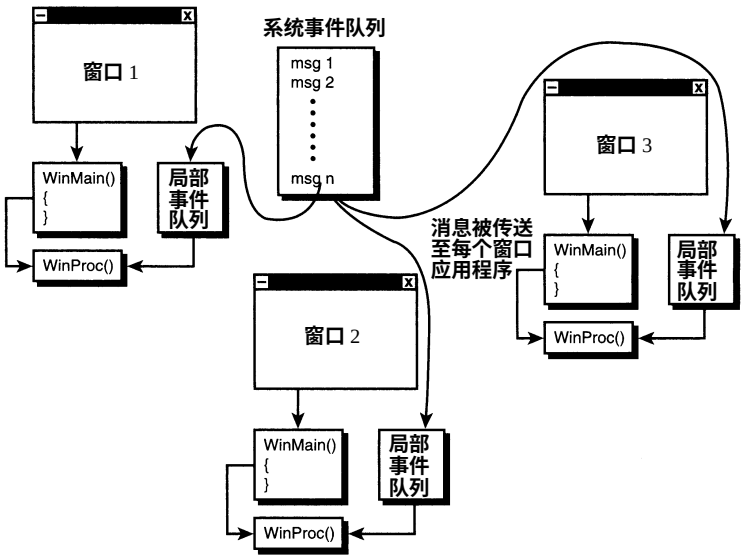


图 2.4　Windows 事件管理程序

这样做的好处是你不必去关心其他的正在运行的应用程序，Windows 会为你处理它们。你所要关心的就是你自己的应用程序和窗口中信息的处理。这在 Windows 3.0/3.1 中是根本不可能的。Windows 的那些版本并不是真正的多任务操作系统，每一个应用程序都要产生下一个程序。也就是说，在这些版本的 Windows 下运行的应用程序感觉相当粗糙、缓慢。如果有其他应用程序干扰系统的话，这个正在“温顺地”运行的程序将停止工作。但这种情况

在 Windows 9X/NT 下就不会出现。操作系统将会在适当的时间终止你的应用程序——当然，运行速度非常快，你根本就不会注意到。

到现在为止，读者已了解了所有有关操作系统的概念。幸运的是，有了 Windows 这个目前最好的编写游戏的操作系统，读者根本就不必担心程序调度——你所要考虑的就是游戏代码和如何最大限度地发挥计算机的性能。

在本章后面内容中，我们要接触一些实际的编程工作，便于读者了解 Windows 编程有多容易。但是（永远都有但是）在进行实际编程之前，我们应当了解一些 Microsoft 程序员喜欢使用的约定。这样你就不会被那些古怪的函数和变量名弄得不知所措。

按照 Microsoft 方式编程：匈牙利符号表示法

如果你正在运作一个像 Microsoft 一样的公司，有几千个程序员在干不同的项目，在某一点上就应当提出一个编写代码的标准方式。否则，结果将是一片混乱。因此一个名字叫 Charles Simonyi 的人被委托创立了一套编写 Microsoft 代码的规范。这个规范已经用作编写代码的基本指导说明书。所有 Microsoft 的 API、界面、技术文件等等都采用这些规范。

这个规范通常被称为匈牙利符号表示法，可能是因为创立这个规范工作花了很长时间，弄得他饥肠辘辘的原因吧（英文中饥饿和匈牙利谐音），或者可能他是匈牙利人。对此我们根本就不知道，关键是你必须了解这个规范，以便于你能够阅读 Microsoft 代码。

匈牙利符号表示法包括许多与下列命名有关的约定：

- 变量
- 函数
- 类型和常量
- 类
- 参数

表 2.1 给出了匈牙利符号表示法使用的前缀代码。这些代码在大多数情况下一半用于前缀变量名，其他约定根据名称确定。其他解释可以参考本表。

表 2.1 匈牙利符号表示法的前缀代码指导说明书

前 缀	数据类型（基本类型）
c	字符
by	字节（无符号字符）
n	短整数和整数（表示一个数）
i	整数
x, y	短整数（通常用于 x 坐标和 y 坐标）

续表

前　缀	数据类型（基本类型）
cx, cy	短整数（通常用于表示 x 和 y 的长度；c 表示计数）
b	布尔型（整数）
w	UINT（无符号整数）和 WORD（无符号字）
l	LONG（长整数）
dw	DWORD（无符号长整数）
fn	函数指针
s	串
sz, str	以 0 字节终止的字符串
lp	32 位的长整数指针
h	编号（常用于表示 Windows 对象）
msg	消息

变量的命名

应用匈牙利符号表示法，变量可用表 2.1 中的前缀代码来表示。另外，当一个变量是由一个或几个子名构成时，每一个子名都要以大写字母开头。下面是几个例子：

```
char *szFileName;     // a nulla terminated string
int *lpiDate;         // a 32-bit pointer to an int
BOOL bSemaphore;      // a boolean value
WORD dwMaxCount;      // a 32-bit unsigned WORD
```

尽管我了解一个函数的局部变量没有说明，但是也有个别表示全局变量：

```
int g_iXPos;         // a global x-position
int g_iTimer;        // a global y-position
char *g_szString;    // a global NULL terminated string
```

总的来说，变量以 g_开头，或者有时就只用 g。

函数的命名

函数和变量命名方式相同，但是没有前缀。换句话说，子名的第一个字母要大写。下面是几个例子：

```
int PlotPixel(int ix, int iy, int ic);
void *MemScan(char *szString);
```

而且，下划线是非法的。例如，下面的函数名表示是无效的匈牙利符号表示法：

```
int Get_Pixel(int ix, int iy);
```

类型和常量的命名

所有的类型和常量都是大写字母，但名字中可以允许使用下划线。例如：

```
const LONG NUM_SECTORS = 100;    // a C++ style constant
#define MAX_CELLS 64;            // a C style constant
#define POWERUNIT 100;           // a C style constant
typedef unsigned char UCHAR;     // a user defined type
```

这儿并没有什么不同的地方——非常标准的定义。尽管大多数 Microsoft 程序员不使用下划线，但我还是喜欢用，因为这样能使名字更具有可读性。

C++

在 C++中，关键字 const 不止一个意思。在前面的代码行中，它用来创建一个常数变量。这和#define 相似，但是它增加了类型信息这个特性。Const 不仅仅像#define 一样是一个简单的预处理文本替换，而且更像是一个变量。它允许编译器进行类型检查和替换。

类的命名

类命名的约定可能要麻烦一点。但我也看到有很多人在使用这个约定，并独立地进行补充。不管怎样说，所有 C++的类必须以大写 C 为前缀，类名字的每一个子名的第一个字母都必须大写。下面是几个例子：

```
class CVector
{
public

CVector (); {ix=iy=iz=imagnitude = 0;}
CVector(int x, int y, int z) {ix=x; iy=y; iz=z;}
.
.
private:
int ix, iy, iz;   // the position of the vector
int imagnitude;  // the magnitude of the vector

};
```

参数的命名

函数的参数命名和标准变量命名的约定相同。但也不总是如此。例如下面例子给出了一个函数定义：

```
UCHAR GetPixel(int x, int y);
```

这种情况下，更准确的匈牙利函数原型是：

```
UCHAR GetPixel(int ix, int iy);
```

但我认为这并没有什么两样。

最后，你甚至可能都看不到这些变量名，而仅仅看到类型，如下所示：

```
UCHAR GetPixel(int, int);
```

当然，这仅仅是原型使用的，真正的函数声明必须带有可赋值的变量名，这一点你已经掌握了。

注　意　仅仅会读匈牙利符号表示法并不代表你能使用它。实际上，我进行编程工作已经有 20 多年了，我也不准备为谁改变我的编程风格。因此，本书中的代码使用类匈牙利符号表示法的编码风格，这是 Win32 API 造成的，在其他位置将使用我自己的风格。必须注意的是，我使用的变量名的第一个字母没有大写，并且我还使用下划线。

世界上最简单的 Windows 程序

现在读者已经对 Windows 操作系统及其性能和基本设计问题有了一般的了解，那就让我们从第一个 Windows 程序开始真正的 Windows 编程吧。

以每一种新语言或所学的操作系统来编写一个“世界你好”的程序是一个惯例，让我们也来试试。清单 2.1 是标准的基于 DOS 的“世界你好”程序。

程序清单 2.1　基于 DOS 的“世界你好”程序

```
// DEMO2_1.CPP - standard version
#include <stdio.h>

// main entry point for all standard DOS/console programs
void main(void)
{
printf("\nTHERE CAN BE ONLY ONE!!!\n");
} // end main
```

现在让我们看一看用 Windows 如何编写它。

技 巧 顺便说一句，如果读者想编译 DEMO2_1.CPP 的话，就应当用 VC++或 Borland 编译器实际创建一个调用内容的控制应用程序。这是一个类 DOS 的应用程序，只是它是 32 位的，它仅以文本模式运行，但对于检验一个想法和算法是很有用的。

总是从 WinMain()开始

如前面所述，所有的 Windows 程序都以 WinMain()开始，这和简单直观的 DOS 程序都以 Main()开始一样。WinMain()中的内容取决于你。如果你愿意的话，可以创建一个窗口、开始处理事件并在屏幕上画一些东西。另一方面，你可以调用几百个（或者是几千个）Win32 API 函数中的一个。这正是我们将要做的。

我只想在屏幕上的一个信息框中打印一点东西。这正是一个 Win32 API 函数 MessageBox()的功能。清单 2.2 是一个完整的、可编译的 Windows 程序，该程序创建和显示了一个能够到处移动和关闭的信息框。

程序清单 2.2 第一个 Windows 程序

```
// DEMO2_2.CPP - a simple message box
#define WIN32_LEAN_AND_MEAN

#include <windows.h>       // the main windows headers
#include <windowsx.h>      // a lot of cool macros

// main entry point for all windows programs
int WINAPI WinMain(HINSTANCE hinstance,
                   HINSTANCE hprevinstance,
                   LPSTR lpcmdline,
                   int ncmdshow)
{
// call message box api with NULL for parent window handle
MessageBox(NULL, “THERE CAN BE ONLY ONE!!!”,
                 “MY FIRST WINDOWS PROGRAM”,
                  MB_OK | MB_ICONEXCLAMATION);

// exit program
return(0);

} // end WinMain
```

要编译该程序，按照下面步骤：

1．创建新的 Win32.EXE 项目并包含 CD-ROM 上 T3DCHAP02\下的 DEMO2_2.CPP。

2．编译和联接程序。

3．运行！（或在 CD-ROM 上直接运行预编译版本 DEMO2_2.EXE）。

你可能会以为一个基本的 Windows 程序有几百行代码。当你编译和运行程序时，可能会看到如图 2.5 所示的内容。

图 2.5　运行 DEMO2_2.EXE

程序剖析

现在已经有了一个完整的 Windows 程序，让我们一行一行地来分析程序的内容。首先第一行程序是

```
#define Win32_LEAN_AND_MEAN
```

这个应稍微解释一下。创建 Windows 程序有两种方式——使用 Microsoft 基础类（Microsoft Foundation Classes,MFC），或者使用软件开发工具包（Software Development Kit,SDK）。MFC 完全基于 C++和类，要比以前的游戏编程所需的工具复杂得多，功能和难度也要强大和复杂 10 倍。而 SDK 是一个可管理程序包，可以在一到两周内学会（至少初步学会），并且使用了简单明了的 C 语言。因此，我在本书中所使用的工具是 SDK。

Win32_LEAN_AND_MEAN 指示编译器（实际上是逻辑头文件）不包含无关的 MFC 操作。现在我们又离题了，回来继续看程序。

之后，下面的头文件是：

```
#include “windows.h”
#include “windowsx.h”
```

第一个引用“windows.h”实际上包括所有的 Windows 头文件。Windows 有许多这样的头文件，这就有点像包含宏，可以节省许多手工包含显式头文件的时间。

第二个引用“windowsx.h”是一个含有许多重要的宏和常量的头文件，该文件可以简化 Windows 编程。

下面就到了最重要的部分——所有 Windows 应用程序的主要入口位置 WinMain()：

```
int WINAPI WinMain(HINSTANCE hinstance,
                   HINSTANCE hprevinstance,
              LPSTR lpcmdline,
             int ncmdshow);
```

首先，应当注意到奇怪的 WINAPI 声明符。这相当于 PASCAL 函数声明符，它强制参数从左边向右边传递，而不是像默认的 CDECL 声明符那样参数从右到左转移。但是，PASCAL 调用约定声明已经过时了，WINAPI 代替了该函数。必须使用 WinMain()的 WINAPI 声明符；否则，将向函数返回一个不正确的参数并终止开始程序。

测试参数

下面让我们详细看一下每个参数：

- hinstance——该参数是一个 Windows 为你的应用程序生成的实例句柄。实例是一个用来跟踪资源的指针或数。本例中，hinstance 就像一个名字或地址一样，用来跟踪你的应用程序。
- hprevinstance——该参数已经不再使用了，但是在 Windows 的旧版本中，它跟踪应用程序以前的实例（换句话说，就是产生当前实例的应用程序实例）。难怪 Microsoft 要去除它，它就像一次长途跋涉——让我们为之头疼。
- lpcmdline——这是一个空终止字符串，和标准 C/C++ main(int argc, char **argv)函数中的命令行参数相似。不同的是，它不是一个单独的像 argc 那样指出命令行的参数。例如，如果你创建一个名字为 TEST.EXE 的 Windows 应用程序，并且使用下面的参数运行：

```
TEST.EXE one
```

lpcmdline 将含有下面数据：

```
lpcmdline = "one two three"
```

注意，.EXE 的名字本身并不是命令行的一部分。

- ncmdshow——最后一整型参数在运行过程中被传递给应用程序，带有如何打开主应用程序窗口的信息。这样，用户便会拥有一点控制应用程序如何启动的能力。当然，作为一个程序员，如果想忽略它也可以，而想使用它也行。（你将参数传递给 ShowWindow()，我们又超前了！）表 2.2 列出了 ncmdshow 最常用的参数值。

表 2.2 ncmdshow 的 Windows 代码

值	功 能
SW_SHOWNORMAL	激活并显示一个窗口。如果该窗口最小化或最大化的话，Windows 将它恢复到原始尺寸和位置。当第一次显示该窗口时，应用程序将指定该标志
SW_SHOW	激活一个窗口，并按当前尺寸和位置显示
SW_HIDE	隐藏一个窗口，并激活另外一个窗口

续表

值	功　能
SW_MAXIMIZE	将指定的窗口最大化
SW_MINIMIZE	将指定的窗口最小化
SW_RESTORE	激活并显示一个窗口。如果该窗口最小化或最大化的话，Windows将它恢复到原始尺寸和位置。当恢复为最小化窗口时，应用程序必须指定该标志
SW_SHOWMAXIMIZED	激活一个窗口，并以最大化窗口显示
SW_SHOWMINIMIZED	激活一个窗口，并以最小化窗口显示
SW_SHOWMINNOACTIVE	以最小化窗口方式显示一个窗口，激活的窗口依然保持激活的状态
SW_SHOWNA	以当前状态显示一个窗口，激活的窗口依然保持激活的状态
SW_SHOWNOACTIVATE	以上一次的窗口尺寸和位置来显示窗口，激活的窗口依然保持激活的状态

如表 2.2 所示，ncmdshow 有许多设置（目前许多值都没有意义）。实际上，这些设置大部分都不在 ncmdshow 中传递。可以应用另一个函数 ShowWindow() 来使用它们，该函数在一个窗口创建时就开始显示。对此我们在本章后面将进行详细讨论。

我想说的一点是，Windows 带有大量的你从未使用过的选项和标志等等，就像 VCR 编程选项一样——越多越好，任你使用。Windows 就是按照这种方式设计的。这将使每个人都感到满意，这也意味着它包含了许多选项。实际上，我们在 99%时间内将会使用 SW_SHOW、SW_SHOWNORMAL 和 SW_HIDE，但是你还要了解在 1%的时间内会用到的其他选项。

选择一个信息框

最后让我们讨论一下 WinMain()中调用 MessageBox()的实际机制。MessageBox()是一个 Win32 API 函数，它替我们做某些事，使我们不需自己去做。该函数经常以不同的图标和一个或两个按钮来显示信息。你看，简单的信息显示在 Windows 应用程序中非常普通，有了这样一个函数就节省了程序员半个多小时的时间，而不必每次使用都要编写它。

MessageBox()并没有什么多少功能，但是能够在屏幕上显示一个窗口，提出一个问题，并且等候用户的输入。下面是 MessageBox()的原型：

```
int MessageBox( HWND   hwnd,        // handle of owner window
                LPCTSTR  lptext,    // address of text in message box
                LPCTSTR  lpcaption, // address of title of message box
                UINT     utype);    // style of message box
```

参数定义如下：

hwnd——这是信息框连接窗口的句柄。目前我们还不能谈及窗口句柄，因此只能认为它是信息框的父窗口。在 DEMO2_2.CPP，我们将它设置为空值 NULL，因此使用 Windows

桌面作为父窗口。

lptext——这是一个包含显示文本的空值终止字符串。

lpcaption——这是一个包含显示文本框标题的空值终止字符串。

utype——这大概是该簇参数中惟一令人激动的参数了，控制信息显示框的种类。

表 2.3 列出了几种 MessageBox()选项（有些删减）。

表 2.3 MessageBox()选项

标　志	描　述
下列设置控制信息框的一般类型	
MB_OK	信息框含有一个按钮：OK，这是默认值
MB_OKCANCEL	信息框含有两个按钮：OK 和 Cancel
MB_RETRYCANCEL	信息框含有两个按钮：Retry 和 Cancel
MB_YESNO	信息框含有两个按钮：Yes 和 No
MB_YESNOCANCEL	信息框含有三个按钮：Yes、No 和 Cancel
MB_ABORTRETRYIGNORE	信息框含有三个按钮：Yes、No 和 Cancel
这一组控制在图标上添加一点“穷人的多媒体”	
MB_ICONEXCLAMATION	信息框显示一个惊叹号图标
MB_ICONINFORMATION	信息框显示一个由圆圈中的小写字母 I 构成的图标
MB_ICONQUESTION	信息框显示一个问号图标
MB_ICONSTOP	信息框显示一个终止符图标
该标志组控制默认时高亮的按钮	
MB_DEFBUTTONn	其中 n 是一个指示默认按钮的数字（1~4），从左到右计数

注意：还有其他的高级 OS 级标志，我们没有讨论。如果希望了解更多细节的话，可以通过编译器 Win32 SDK 的在线帮助来查阅。

可以同时使用表 2.3 中的值进行逻辑或运算，来创建一个信息框。一般情况下，只能从每一组中仅使用一个标志来进行或运算。

当然，和所有 Win32 API 函数一样，MessageBox()函数返回一个值来通知编程者所发生的事件。但在这个例子中谁关心这个呢？通常情况下，如果信息框是 yes/no 提问之类的情况的话，就希望知道这个返回值。表 2.4 列出了可能的返回值。

表 2.4 MessageBox()的返回值

值	按钮选择
IDABORT	Abort
IDCANCEL	Cancel
IDIGNORE	Ignore
IDNO	No
IDOK	OK
IDRETRY	Retry
IDYES	Yes

最后，这个表已经毫无遗漏地列出了所有的返回值。现在已经完成了对我们第一个 Windows 程序——单击的逐行分析。

技　巧

现在希望你能轻松地对这个程序进行修改，并以不同的方式进行编译。使用不同的编译器选项，例如优化。然后尝试通过调试程序来运行该程序，看看你是否已经领会。做完后，请回到此处。

如果希望听到声音的话，一个简单的技巧就是使用 MessageBeep()函数，可以在 Win32 SDK 中查阅，它和 MessageBox()函数一样简单好用。下面就是该函数原型：

```
BOOL MessageBeep(UINT utype);      // 运行声音
```

可以从表 2.5 所示常数中得到不同的声音。

表 2.5　MessageBeep()函数的声音标识符

值	声　音
MB_ICONASTERISK	系统星号
MB_ICONEXCLAMATION	系统惊叹号
MB_ICONHAND	系统指针
MB_ICONQUESTION	系统问号
MB_OK	系统默认值
0xFFFFFFFF	使用计算机扬声器的标准嘟嘟声，令人讨厌！

注意：如果已经安装了 MS-Plus 主题曲的话，你应能得到有意思的结果。

看 Win32 API 多酷啊！可以有上百个函数使用。它们虽然不是世界上最快的函数，但是对于一般的内部管理 I/O 和 GUI 来讲，它们已经很棒了。

让我们稍微花点时间总结一下我们目前所知的有关 Windows 编程方面的知识。首先，Windows 支持多任务/多线程，因此可以同时运行多个应用程序。我们不必费心就可以做到这一点。我们最关心的是 Windows 支持事件触发。这就意味着我们必须处理事件（在这一点上目前我们还不知如何做）并且做出反应。好，听上去不错。最后所有 Windows 程序都以函数 WinMain()开始，WinMain()函数中的参数要比标准 DOS Main()多得多，但这些参数都属于逻辑和推理的领域。

掌握了上述的内容，就到了编写一个真正的 Windows 应用程序的时候了。

真实的 Windows 应用程序

尽管本书的目标是编写在 Windows 环境下运行的 3D 游戏，但是你并不需要了解更多的 Windows 编程。实际上，你所需要的就是一个基本的 Windows 程序，可以打开一个窗口、处理信息、调用主游戏循环等等。了解了这些，本章中的目标是首先向你展示如何创建简单的 Windows 应用程序，同时为编写类似 32 位 DOS 环境的游戏外壳程序奠定基础。

一个 Windows 程序的关键就是打开一个窗口。一个窗口就是一个显示文本和图形信息的工作区。要创建一个完全实用的 Windows 程序，只要进行下列工作：

1. 创建一个 Windows 类。
2. 创建一个事件句柄或 WinProc。
3. 用 Windows 注册 Windows 类。
4. 用前面创建的 Windows 类创建一个窗口。
5. 创建一个能够从事件句柄获得或向事件句柄传递 Windows 信息的主事件循环。

让我们详细了解一下每一步的工作。

Windows 类

Windows 实际上是一个面向对象的操作系统，因此 Windows 中大量的概念和程序都出自 C++。其中一个概念就是 Windows 类。Windows 中的每一个窗口、控件、列表框、对话框和小部件等等实际上都是一个窗口。区别它们的就是定义它们的类。一个 Windows 类就是 Windows 能够操作的一个窗口类型的描述。

有许多预定义的 Windows 类，如按钮、列表框、文件选择器等等。你也可以自己任意创建你的 Windows 类。实际上，你可以为自己编写的每一个应用程序创建至少一个 Windows 类。否则你的程序将非常麻烦。因此你应当在画一个窗口时，考虑一个 Windows 类来作为 Windows 的一个模板，以便于在其中处理信息。

控制 Windows 类信息的数据结构有两个：WNDCLASS 和 WNDCLASSEX。WNDCLASS 是比较古老的一个，可能不久将废弃，因此我们应当使用新的扩展版 WNDCLASSEX。二者结构非常相似，如果有兴趣的话，可以在 Win32 帮助中查阅 WNDCLASS。让我们看一下在 Windows 头文件中定义的 WNDCLASSEX。

```
typedef struct _WNDCLASSEX
{
UINT     cbSize;         // size of this structure
UINT     style;          // style flags
```

```
WNDPROC lpfnWndProc;    // function pointer to handler
int       cbClsExtra;   // extra class info
int       cbWndExtra;   // extra instance info
HANDLE  hInstance;      // the instance of the application
HICON    hIcon;         // the main icon
HCURSOR hCursor;        // the cursor for the window
HBRUSH hbrBackground;   // the background brush to paint the window
LPCTSTR lpszMenuName;   // the name of the menu to attach
LPCTSTR lpszClassName; // the name of the class itself
HICON   hIconSm;        // the handle of the small icon
} WNDCLASSEX
```

因此你所要做的就是创建一个这样的结构，然后填写所有的字段：

```
WNDCLASSEX winclass;   // a blank windows class
```

第一个字段 cbSize **非常重要（但** Petzold **在《**Programing Windows 95**》中忘记了这个内容），它是** WNDCLASSEX **结构本身的大小。你可能要问，为什么应当知道该结构的大小？这个问题问得好，原因是如果这个结构作为一个指针被传递的话，接收器首先检查第一个字段，以确定该数据块最低限度有多大。这有点像提示和帮助信息，以便于其他函数在运行时不必计算该类的大小。因此，我们应当这样做：**

```
winclass.cbSize = sizeof(WNDCLASSEX);
```

第二个字段包含描述该窗口一般属性的结构信息标志。有许多这样的标志，因此我们不能全部列出它们。只要能够使用它们创建任何类型的窗口就行了。表 2.6 **列出了常用的标志。读者可以任意对这些值进行逻辑“或”运算，来派生所希望的窗口类型。**

表 2.6　Windows **类的类型标志**

标　志	说　明
CS_HREDRAW	若移动或改变了窗口宽度，则刷新整个窗口
CS_VREDRAW	若移动或改变了窗口高度，则刷新整个窗口
CS_OWNDC	为该类中每个窗口分配一个单值的设备描述表（在本章后面详细描述）
CS_DBLCLKS	当用户双击鼠标时向窗口程序发送一个双击的信息，同时，光标位于属于该类的窗口中
CS_PARENTDC	在母窗口中设定一个子窗口的剪切区，以便于子窗口能够画在母窗口中
CS_SAVEBITS	在一个窗口中保存用户图像，以便于在该窗口被遮住、移动时不必每次刷新屏幕。但是，这样会占用更多的内存，并且比人工同样操作要慢得多
CS_NOCLOSE	禁止系统菜单上的关闭命令

注意：用黑体显示的部分为最常用的标志。

表 2.6 **包含了大量的标志，即使读者对此尚有疑问，也无关紧要。现在，设定类型标识**

符，描述如果窗口移动或改变尺寸就进行屏幕刷新，并可以获得一个静态的设备描述表以及处理双击事件的能力。

我们将在第三章“高级 Windows 编程”中详细讨论设备描述表，但基本说来，它被用作窗口中图像着色的数据结构。因此，如果你要处理一个图像，就应为感兴趣的特定窗口申请一个设备描述表。如果设定了一个 Windows 类，它就通过 CS_OMNDC 得到了一个设备描述表，如果你不想每次处理图像时都申请设备描述表，可以将它保存一段时间。上面说的对你有帮助还是使你更糊涂？Windows 就是这样——你知道得越多，问题就越多。好了！下面说一下如何设定类型字段：

```
winclass.style = CS_VERDRAW | CS_HREDRAW | CS_OWNDC | CS_DBLCLICKS;
```

WNDCLASSEX 结构的下一个字段 lpfnWndProc 是一个指向事件句柄的函数指针。基本上这里所设定的都是该类的回调函数。回调函数在 Windows 编程中经常使用，工作原理如下：当有事件发生时，Windows 通过调用一个你已经提供的回调函数来通知你，这省去你盲目查询的麻烦。随后在回调函数中，再进行所需的操作。

这个过程就是基本的 Windows 事件循环和事件句柄的操作过程。向 Windows 类申请一个回调函数（当然需要使用特定的原型）。当一个事件发生时，Windows 按如图 2.6 所示的的那样替你调用它。关于该项内容我们将在下面部分进行更详细的介绍。但是现在，读者只要将其设定到你将编写的事件函数中去：

```
winclass.lpfnWndProc = WinProc;  // 这是我们的函数
```

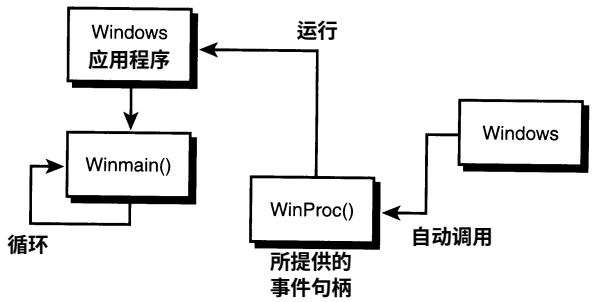


图 2.6 Windows 事件句柄回调函数的操作流程

C++

函数指针有点像 C++中的虚函数。如果你对它们不熟悉的话，在这里讲一下。假设有两个函数用以操作两个数：

```
int Add(int op1,int op2) {return(op1+op2);}
int Sub(int op1,int op2) {return(op1-op2);}
```

C++

要想用同一调用来调用两个函数，可能用一个函数指针来实现。如下：

```
// define a function pointer that takes two int and returns
an int
int (Math*)(int, int);
```

然后可以如下配置函数指针：

```
Math = Add;
int result = Math(1,2);// this reslly calls Add(1,2)
// result will be 3

 Math = Sub;
 Int result = Math(1,2);// this really calls Sub(1,2)
// result will be –1
```

看，不错吧。

下面两个字段，cbClsExtra 和 cbWndExtra 原是为指示 Windows 将附加的运行时间信息保存到 Windows 类某些单元中而设计的。但是绝大多数人使用这些字段并简单地将其值设为 0，如下所示：

```
winclass.cbClsExtra = 0;   // extra class info space
winclass.cbWndExtra = 0;   // extra window info space
```

下一个是 hInstance 字段。这是一个简单的、在启动时传递给 WinMain()函数的句柄实例，因此只需简单地从 WinMain()中复制即可：

```
winclass.hInstance = hinstance;  // assign the application instance
```

剩下的字段和 Windows 类的图像方面有关，在讨论它们之前，先花一点时间回顾一下句柄。

在 Windows 程序和类型中将一再看到句柄：位图句柄、光标句柄、任意事情的句柄。请记住，句柄只是一个基于内部 Windows 类型的标识符。其实它们都是整数。但是 Microsoft 可能改变这一点，因此安全使用 Microsoft 类型是个好主意。总之，你将会看到越来越多的‘[…]句柄”，请记住，有前缀 h 的任何类型通常都是一个句柄。好，回到原来的地方继续吧。

下一个字段是设定表示应用程序的图标的类型。你完全可以装载一个你自己定制的图标，但现在你使用系统图标，需要为它设置一个句柄。要为一个常用的系统图标检索一个句柄，可以使用 LoadIcon()函数：

```
winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

这行代码装载一个标准的应用程序图标——虽然烦人，但是简单。如果对 loadIcon()函数有兴趣的话，请看下面的它的原型，表 2.7 给出了几个图标选项：

```
HICON LoadIcon(HINSTANCE hInstance,  // handle of application instance
  LPCTSTR lpIconName);   // icon-name string or icon resource identifier
```

hInstance 是一个从应用程序装载图标资源的实例（后面将详细讨论）。现在将它设置为 NULL 来装载一个标准的图标。LpIconName 是包含被装载图标资源名称的 NULL 终止字符串。当 hInstance 为 NULL 时，lpIconName 的值如表 2.7 所示。

表 2.7 LoadIcon()的图标标识符

值	说 明
IDI_APPLICATION	默认应用程序图标
IDI_ASTERISK	星号
IDI_EXCLAMATION	惊叹号
IDI_HAND	手形图标
IDI_QUESTION	问号
IDI_WINLOGO	Windows 徽标

好，现在我们已介绍了一半的字段了。做个深呼吸休息一会，让我们进行下一个字段 hCursor 的介绍。和 hIcon 相似，它也是一个图像对象句柄。不同的是，hCursor 是一个指针进入到窗口的用户区才显示的光标句柄。使用 LoadCursor()函数可以得到资源或预定义的系统光标。我们将在后面讨论资源，简单而言资源就是像位图、光标、图标、声音等一样的数据段，它被编译到应用程序中并可以在运行时进行访问。Windows 类的光标设定如下所示：

```
winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
```

下面是 LoadCursor()函数的原型（表 2.8 列出了不同的系统光标标识符）：

```
HCURSOR lpCursor( HINSTANCE hInstance,  // handle of application instance
LPCTSTR lpCursorName);  // icon_name string or icon resource identifier
```

hInstance 是你的.EXE 的应用程序实例。该.EXE 应用程序包含订制光标名称来源的资源。但现在读者还不能使用该功能，仅将默认的系统光标值设定为 NULL。

lpCursorName 标识了资源名字符串或资源句柄（我们一般不使用），或者是一个常数，以标识如表 2.8 中所示的系统默认值。

表 2.8 LoadCursor()的值

值	说 明
IDC_ARROW	标准箭头
IDC_APPSTARTING	标准箭头标和小沙漏标
IDC_CROSS	横标线
IDC_IBEAM	文本 I 型标
IDC_NO	带正斜线的圆圈

续表

值	说　明
IDC_SIZEALL	四向箭头
IDC_SIZENESW	指向东北—西南方向的双向箭头
IDC_SIZENS	指向南北方向的双向箭头
IDC_SIZENWSE	指向东南—西北方向的双向箭头
IDC_SIZEWE	指向东西方向的双向箭头
IDC_UPARROW	垂直方向的箭头
IDC_WAIT	沙漏

现在我们就要解放了，因为我们几乎已经全部介绍完了——剩下的字段更有意义。让我们看一看 hbrBackground。

无论在什么时候绘制或刷新一个窗口，Windows 都至少将以用户预定义的颜色或 Windows 内部设置的画笔颜色填充该窗口的背景。因此，hbrbackground 是一个用于窗口刷新的画笔句柄。画笔、笔、色彩和图形都是 GDI（图形设备接口）的一部分，我们将在下一章中详细讨论。现在，介绍一下如何申请一个基本的系统画笔来填充窗口。该项功能由 GetStockObject()来实现，如下面程序所示：

```
winclass.hbrBackground = GetStockObject(WHITE_BRUSH)；
```

GetStockObject()是一个通用函数，用于获得 Windows 系统画笔、笔、调色板或字体的一个句柄。GetStockObject()只有一个参数，用来指示装载哪一项资源。表 2.9 仅列出了画笔和笔的可能库存对象。

表 2.9　GetStockObject()的库存对象标识符

值	说　明
BLACK_BRUSH	黑色画笔
WHITE_BRUSH	白色画笔
GRAY_BRUSH	灰色画笔
LTGRAY_BRUSH	淡灰色画笔
DKGRAY_BRUSH	深灰色画笔
HOLLOW_BRUSH	空心画笔
NULL_BRUSH	无效（NULL）画笔
BLACK_PEN	黑色笔
WHITE_PEN	白色笔
NULL_PEN	无效（NULL）笔

WNDCLASS 结构中的下一个字段是 lpszMenuName。它是菜单资源名称的空终止 ASCII

字符串，用于加载和选用窗口。其工作原理将在第三章“高级 Windows 编程”中讨论。现在我们只需将值设为 NULL：

```
Winclass.lpszmenuName=NULL；//the name of the menu to attach
```

如我刚提及的那样，每个 Windows 类代表你的应用程序所创建的不同窗口类型。在某种程度上，类与模板相似，Windows 需要一些途径来跟踪和识别它们。因此，下一个字段 lpszClassName，就用于该目的。该字段被赋以包含相关类的文本标示符的空终止字符串。我个人喜欢用诸如“WINCLASS1”、“WINCLASS2”等标示符。读者以自己喜好而定，以简单明了为原则，如下所示：

```
Winclass.lpszClassName = “WINCLASS1”;  // the name of the class itself
```

这样赋值以后，你可以使用它的名字来引用这个新的 Windows 类——很酷，是吗？

最后就是小应用程序图表。这是 Windows 类 WNDCLASSEX 中新增加的功能，在老版本 WNDCLASS 中没有。首先，它是指向你的窗口标题栏和 Windows 桌面任务栏的句柄。你经常需要装载一个自定义资源，但是现在只要通过 LoadIcon()使用一个标准的 Windows 图标即可实现：

```
winclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION)；  // 小图标句柄
```

下面让我们迅速回顾一下整个类的定义：

```
WNDCLASSEX winclass; // this will hole the class we create

// first fill in the window class structure
winclass.cbSize = sizeof(WNDCLASSEX);
winclass.style = CS_DBLCLKS | CS_OWNDC | CS_HREDRAW |CS_VREDRAW;
winclass.lpfnWndProc = WindowProc;
winclass.cbClsExtra = 0;
winclass.cbWndExtra = 0;
winclass.hInstance = hinstance;
winclass.hIcon = LoadIcon(NULL,IDI_APPLICATION);
winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBacground = GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName = NULL;
winclass.lpszClassName = “WINCLASS”;
winclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
```

当然，如果想节省一些打字时间的话，可以像下面这样简单地初始化该结构：

```
WNDCLASSEX winclass = {
                winclass.cbSize = sizeof(WNDCLASSEX),
               CS_DBLCLKS | CS_OWNDC | CS_HREDRAW | CS_VREDRAW,
               WindowProc,
               0,
```

```
0,
hinstance,
LoadIcon(NULL,IDI_APPLICATION),
LoadCursor(NULL, IDC_ARROW),
GetStockObject(BLACK_BRUSH),
NULL,
“WINCLASS1”,
LoadIcon(NULL, IDI_APPLICATION)};
```

这样就省去了许多输入！

注册 Windows 类

现在 Windows **类已经定义并且存放在** winclass **中，必须将新的类通知** Windows**。该功能通过** RegisterClassEx()**函数，使用一个指向新类定义的指针来完成，如下所示：**

```
RegisterClassEx( &winclass );
```

警 告

注意我并没有使用我们例子中的“ WINCLASS1**”的类名，对于** RegisterClassEx()**来讲，必须使用保存该类的实际结构，因为在该类调用** RegisterClassEx()**函数之前，**Windows **并不知道该类的存在。明白了吧？**

此外还有一个旧版本的 RegisterClass()**函数，用于注册基于旧结构** WNDCLASS **基础上的类。**

该类一旦注册，我们就可以任意创建它的窗口。请看下面如何进行这个工作，然后再详细看一下事件句柄和主事件循环，了解使一个 Windows **应用程序运行还要做哪些工作。**

创建窗口

要创建一个窗口（或者一个类窗口的对象），使用 CreateWindow()**或** CreateWindowEx() **函数。后者是更新一点的版本，支持附加类型参数，我们就使用它。该函数是创建** Windows **类的函数，我们要多花一点时间来逐行分析。在创建一个窗口时，必须为这个** Windows **类提供一个正文名——我们现在就使用“** WINCLASS1 **”命名。这是识别该** Windows **类并区别于其他类以及内嵌的诸如按钮、文本框等类型的标示。**

下面是 CreateWindowEx()**函数的原型：**

```
HWND CreateWindowEx(
DWORD dwExStyle,        // extended window style
LPCTSTR lpClassName,    // pointer to registered class name
```

```
LPCTSTR lpWindowName,   // pointer to window name
DWORD dwStyle,          // window style
int x,                  // horizontal position of window
int y,                  // vertical position of  window
int nWidth,             // window width
int nHeight,            // window height
HWND hWndParent,        // handle to parent or owner window
HMENU hMenu,            // handle to menu,or child window identifier
HINSTANCE hInstance,    // handle to application instance
LPVOID lpParam);        // pointer to window-creation data
```

如果该函数执行正确的话，将返回一个指向新建窗口的句柄；否则就返回空值 NULL。**上述大多数参数是不需要加以说明的，现在让我们浏览一下：**

- dwExStyle——**该扩展类型标志具有高级特征，大多数情况下，可以设为** NULL。**如果读者对其取值感兴趣的话，可以查阅** Win32 SDK **帮助，上面有详细的有关该标识符取值的说明。**WS_EX_TOPMOST **是我惟一使用过的一个值，该功能使窗口一直保持在上部。**
- lpClassName——**这是你所创建的窗口的基础类名——例如“** WINCLASS1 **”。**
- lpWindowName——**这是包含窗口标题的空终止文本字符串——例如“我的第一个窗口”。**
- dwStyle——**这是一个说明窗口外观和行为的通用窗口标志——非常重要！表** 2.10 **列出了一些最常用的值。当然，可以任意组合使用这些值来得到希望的各种特征。**
- x，y——**这是该窗口左上角位置的像素坐标。如果你无所谓，可使用** CW_USEDEFAULT，**这将由** Windows **来决定。**
- nWidth，nHeight——**这是以像素表示的窗口宽度和高度。如果你无所谓，可使用** CW_USEDEFAULT，**这将由** Windows **来决定。**
- hWndParent——**假如父窗口存在，这是指向父窗口的句柄。如果没有父窗口，桌面就是父窗口。**
- hMenu——**这是指向附属于该窗口菜单的句柄。下一章中将详细介绍，现在将其赋值** NULL。
- hInstance——**这是应用程序实例。这里从** WinMain()**中使用实例。**
- lpParam——**高级特征，设置为** NULL。

表 2.10 **列出了各种窗口标志设置。**

表 2.10　dwStyle **的通用类型值**

类　　型	所创建的内容
WS_POPUP	**弹出式窗口**
WS_OVERLAPPED	**带有标题栏和边界的重叠式窗口，类似** WS_TILED **类型**

续表

类　　型	所创建的内容
WS_OVERLAPPEDWINDOW	具有 WS_OVERLAPPED、WS_CAPTION、WS_SYSMENU、WS_THICKFRAME 、 WS_MAXIMIZEBOX 和 WS_MINIMIZEBOX 样式的重叠式窗口
WS_VISIBLE	开始就可见的窗口
WS_SYSMENU	标题栏上有窗口菜单的窗口
WS_BORDER	有细线边界的窗口
WS_CAPTION	有标题栏的窗口（包括 WS_BORDER 样式）
WS_ICONIC	开始就最小化的窗口，类似 WS_MINIMIZE 样式
WS_MAXIMIZE	开始就最大化的窗口
WS_MAXIMIZEBOX	具有最大化按钮的窗口。不能和 WS_EX_ CONGTEXTHELP 样式合并。WS_SYSMENU 也必须指定
WS_MINIMIZE	开始就最小化的窗口，类似 WS_ICONIC 样式
WS_MINIMIZEBO X	具有最小化按钮的窗口。不能和 WS_EX_ CONGTEXTHELP 样式合并。WS_SYSMENU 也必须指定
WS_POPUPWINDOW	带有 WS_BORDER、WS_POPUP 和 WS_SYSMENU 类型的弹出式窗口
WS_SIZEBOX	一个窗口边界可以变化，和 WS_THICKFRAME 类型相同
WS_HSCROLL	带有水平滚动条的窗口
WS_VSCROLL	带有垂直滚动条的窗口

注意：突出显示的是经常使用的值。

下面是使用标准控件在（0,0）位置创建一个大小为 400X400 像素的、简单的重叠式窗口。

```
HWND hwnd;   // window handle

// create the window,bail if problem
if (!(hwnd = CreateWindowEx(NULL,   // extended style
                  " WINCLASS";           // class
                  " Your Basic Window",  // title
                  WS_OVERLAPPEDWINDOW | WS_VISIBLE;
                  0,0         // initial x,y
                  400,400      // initial width,height
                  NULL,        // handle to parent
                  NULL,        // handle to menu
                  hinstance,   // instance of this application
                  NULL)))     // extra creation parms
return( 0 );
```

一旦创建了该窗口，它可能是可见或不可见的。但是，在这个例子中，我们增加了自动

显示的类型标识符 WS_VISIBLE。如果没有添加该标识符，则调用下面的函数来人工显示该窗口：

```
// this shows the window
ShowWindow(hwnd, ncmdshow);
```

记住 WinMain()中的 ncmdshow 参数了吗？这就是使用它的方便之处。尽管我们使用 WS_VISIBLE 覆盖了 ncmdshow 参数，但还是应将其作为一个参数传递给 ShowWindow()。下面让 Windows 更新窗口的内容，并且产生一个 WM_PAINT 信息，这通过调用函数 UpdateWindow()来完成：

```
// this sends a WM_PAINT message to window and makes
// sure the contents are refreshed
UpdateWindow();
```

事件处理程序

我并不了解你的情况，但注意我现在正使你掌握 Windows 的核心。它有如一本神秘小说。请记住，我所说的事件处理程序就是当事件发生时 Windows 从主事件循环调用的回调函数。回顾一下图 2.6，刷新一下你对通用数据流的印象。

事件处理器由读者自己编写，它能够处理你所关心的所有事件。其余的工作就交给 Windows 处理。当然，请记住，你的应用程序所能处理的事件和消息越多，它的功能越强。

在编写程序之前，让我们讨论一下事件处理器的一些细节，即事件处理器能做什么，工作机理如何。首先，对于创建的任何一个 Windows 类，都有一个独立的事件处理器，我指的是 Windows' Procedure，从现在开始简称 WinProc。当收到用户或 Windows 发送的消息并放在主事件序列中时，WinProc 就接收到主事件循环发送的消息。这简直是一个智力绕口令，我换个方式来说明……

当用户和 Windows 运行任务时，你的窗口和/或其他应用程序窗口产生事件和消息。所有消息都进入一个队列，而你的窗口的消息发送到你的窗口专用队列中。然后主事件循环检索这些消息，并且将它们发送到你的窗口的 WinProc 中来处理。

这几乎有上百个可能的消息和变量，因此，我们就不全部分析了。值得庆幸的是，你只需处理很少的消息和变量，就可以启动并运行 Windows 应用程序。

简单地说，主事件循环将消息和事件反馈到 WinProc，WinProc 对它们进行处理。因此不仅你要关注 WinProc，主事件循环同样也要关心 WinProc。现在我们简要地了解一下 WinProc，现假定 WinProc 只接收消息。

现在来看一下 WinProc 的工作机制，让我们看一下它的原型：

```
LRESULT CALLBACK WindowProc(
                HWND hwnd,        // window handle of sender
                UINT msg,         // the message id
                WPARAM wparam,    // further defines message
                LPARAM lparam);   // further defines message
```

当然，这仅仅是回调函数的原型。只要将函数地址作为一个函数指针传递给 winclass.lpfnWndProc**，就可以调用该函数的任何信息，如下所示：**

```
winclass.lpfnWndProc = WindowProc;
```

参数的含义是不言自明的：

- hwnd**——这是一个** Windows **句柄，只有当你使用同一个** Windows **类打开多个窗口时它才用到。这种情况下，**hwnd **是表明消息来自哪个窗口的惟一途径。图** 2.7 **表示了这种情况。**

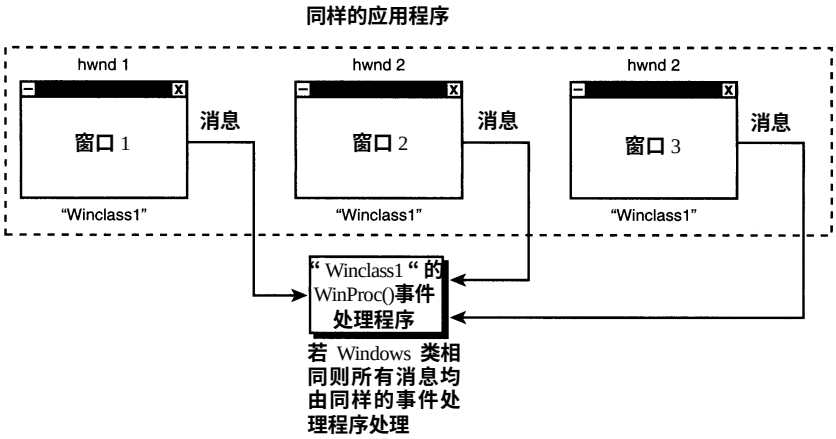


图 2.7 **使用同一个类打开的多个窗口**

- msg**——这是一个实际的** WinProc **处理的消息标识符。这个标识符可以是众多主要消息中的一个。**
- Wparam **和** lparam**——进一步限定或细分发送到** msg **参数中的信息。**

最后，我们感兴趣的是返回类型 LRESULT **和声明说明符** CALLBAK**。这些关键字都是必需的，不能忽略它们。**

因此大多数人所要做的就是使用 switch()**来处理** msg **所表示的消息，然后为每一种情况编写代码。在** msg **的基础上，你可以知道是否需要进一步求** wparam **和/或** lparam **的值。很酷吗？因此让我们看一下由** WinProc **传递过来的所有可能的消息，然后再看一下** WinProc **的工作机理。表** 2.11 **简要列出了一些基本的消息说明符。**

表 2.11　**消息说明符的简表**

值	说　明
WM_ACTIVATE	当窗口被激活或者成为一个焦点时传递
WM_CLOSE	当窗口关闭时传递
WM_CREATE	当窗口第一次创建时传递
WM_DESTROY	当窗口可能要被破坏时传递
WM_MOVE	当窗口移动时传递
WM_MOUSEMOVE	当移动鼠标时传递
WM_KEYUP	当松开一个键时传递
WM_KEYDOWN	当按下一个键时传递
WM_TIMER	当发生定时程序事件时传递
WM_USER	允许传递消息
WM_PAINT	当一个窗口需重画时传递
WM_QUIT	当 Windows 应用程序最后结束时传递
WM_SIZE	当一个窗口改变大小时传递

要认真看表 2.11，了解所有消息的功能。在应用程序运行时将有一个或多个上述消息传递到 WinProc。消息说明符本身在 msg 中，而其他信息都存储在 wparam 和 lparam 中。因此参考在线 Win32 SDK 帮助来了解某个消息的参数所代表的意思是个不错的方法。

幸好我们现在只对下面三个消息感兴趣：

- ✧ WM_CREATE——当窗口第一次创建时传递该消息，以便你进行启动、初始化或资源配置工作。
- ✧ WM_PAINT——当一个窗口内容需重画时传递该消息。这可能有许多原因：用户移动窗口或改变其尺寸、弹出其他应用程序而遮挡了你的窗口不清楚等。
- ✧ WM_DESTROY——当窗口可能要被破坏时该消息传递到你的窗口。通常这是由于用户单击该窗口的关闭按钮，或者是从该窗口的系统菜单中关闭该窗口造成的。无论上述哪一种方式，都应当释放所有的资源，并且通过发送一个 WM_QUIT 消息来通知 Windows 完全终止应用程序。后面还将详细介绍。

OK！让我们看一看 WinProc 处理这些消息的整个过程。

```
LRESULT CALLBACK WindowProc(HWND hwnd,
                            UINT msg,
                            WPARAM wparam,
                            LPARAM lparam);
{
// this is the main message handler of the system
PAINTSTRUCT  ps;  // Used in WM_PAINT
HDC   hdc;    // handle to a device context
```

```
// what is the message
switch(msg)
   {
case WM_CREATE;
   {
// do initialization stuff here

   // return success
return ( 0 )
} break;

case WM_PAINT;
{
// simply validate the window
hdc = BeginPaint(hwnd, &ps);
// you woudle do all your painting here
   EndPaint(hwnd, &ps);

   // return success
return ( 0 )
} break;

case WM_DESTROY;
{
// Kill the application,this sends a WM_QUIT message
PostQuitMessage ( 0 );

   // return success
return ( 0 )
} break;

default:break;

} // end switch

// process any message that we didn't take care of
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc
```

由上面可以看到，函数的大部分是由空白区构成——这真是件好事情。让我们就以 WM_CREATE **处理程序开始吧。该函数所作的一切就是** return(0)**。这就是通知** Windows **由编程人员自己处理该函数，因此无需更多的操作。当然，也可以在** WM_CREATE **消息中进行全部的初始化工作，但那是你的事了。**

下一个消息 WM_PAINT **非常重要。该消息在窗口需要重画时被发送。一般来说这表示你应当进行重画工作。对于** DirectX **游戏来说，这并不是件什么大事，因为你将以** 30 **到** 60 **帧/秒的频率来重画屏幕。但是对于标准** Windows **应用程序来说，它就是件大事了。我将在**

后面章节中更详细地介绍 WM_PAINT，目前的功能就是通知 Windows，你要重画该窗口了，因此就停止发送 WM_PAINT 消息。

要完成该功能，你必须激活该窗口的客户区。有许多方法可以做到，但调用函数 BeginPaint()和 EndPaint()最简单。这一对调用将激活窗口，并使用原先存储在 Windows 类中的变量 hbrBackground 的背景刷来填充背景。下面是程序代码：

```
   // begin painting
hdc = BeginPaint(hwnd, &ps);
// you would do all your painting here
   EndPaint(hwnd, &ps);
```

下面要提醒几件事情。第一，请注意，每次调用的第一个参数是窗口句柄 hwnd。这是一个非常必要的参数，因为 BeginPaint—EndPaint 函数能够在任何应用程序窗口中绘制，因此该窗口句柄指示了要重画哪个窗口。第二个参数是包含必须重画矩形区域的 PAINTSTRUCT 结构的地址。下面是 PAINTSTRUCT 结构：

```
typedef struct tagPAINTSTRUCT
        {
        HDC  hdc;
        BOOL fErase;
        RECT rcPaint;
        BOOL fRestore;
        BOOL fIncUpdate;
        BYTE rgbReserved[32];
        } PAINTSTRUCT;
```

实际上不需要关心这个函数，当我们讨论图形设备接口时会再讨论这个函数。其中最重要的字段就是 rcPaint。图 2.8 表示了这个字段的内容。注意 Windows 一直尽可能地试图作最少的工作，因此当一个窗口内容破坏之后，Windows 至少会告诉你要恢复该内容并能够重画的最小的矩形。如果你对矩形结构感兴趣的话，会发现只有矩形的四个角是最重要的，如下所示：

```
typedef struct tagRECT
        {
        LONG  left;     // left x-edge of rect
        LONG  top;      // top y-edge of rect
        LONG  right;    // right x-edge of rect
        LONG  bottom;   // bottom y-edge of rect
        } RECT;
```

调用 BeginPaint()函数应注意的最后一件事情是,它返回一个指向图形环境或 hdc 的句柄：

```
HDC hdc;      // handle to graphics context
Hdc = BeginPaint(hwnd, &ps);
```

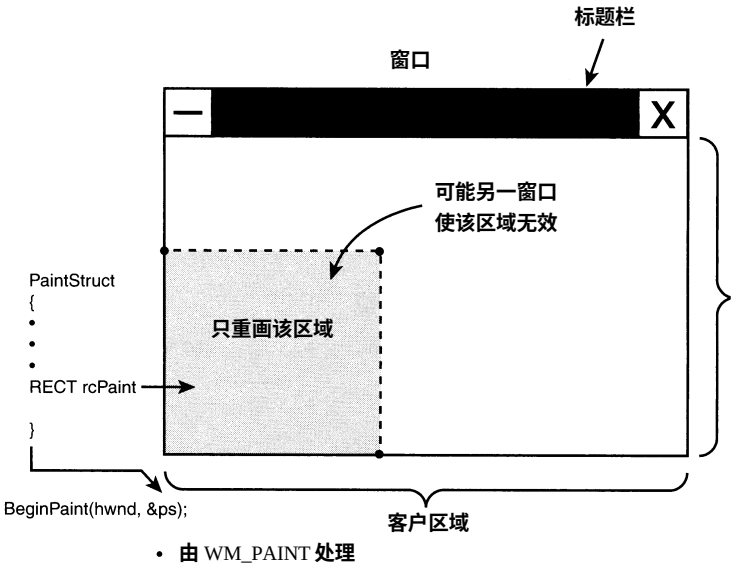


图 2.8 仅重新绘制无效区

图形环境就是描述视频系统和正在绘制表面的数据结构。奇妙的是，如果你需要绘制图形的话，只要获得一个指向图形环境的句柄即可。这便是关于 WM_PAINT 消息的内容。

WM_DESTROY 消息实际上非常有意思。WM_DESTROY 在用户关闭窗口时被发送。当然仅仅是关闭窗口，而不是关闭应用程序。应用程序继续运行，但是没有窗口。对此要进行一些处理。大多数情况下，当用户关闭主要窗口时，也就意味着要关闭该应用程序。因此，你必须通过发送一个消息来通知系统。该消息就是 WM_QUIT。因为该消息经常使用，所以有一个函数 PostQuitMessage()来替你完成发送工作。

在 Wm_DESTROY 处理程序中你所要做的就是清除一切，然后调用 PostQuitMessage(0) 通知 Windows 终止应用程序。接着将 WM_QUIT 置于消息队列，这样在某一个时候终止主事件循环。

在我们所分析的 WinProc 句柄中还有许多细节应当了解。首先，你肯定注意到了每个处理程序体后面的 return(0)。它有两个目的：退出 WinProc 以及通知 Windows 你已处理的信息。第二个重要的细节是默认消息处理程序 DefaultWindowProc()。该函数是一个传递 Windows 默认处理消息的传递函数。因此，如果不处理该消息的话，可通过如下所示的调用来结束你的所有事件处理函数：

```
// process any messages that we didn't take care of
return (DefWindowProc(hwnd, msg, wparam, lparam));
```

我认为代码本身过多并且过于麻烦。然而，一旦你有了一个基本 Windows 应用程序架构的话，你只要将它复制并在其中添加你自己的代码就行了。正如我所说的那样，我的主要目标是帮助你创建一个可以使用的类 DOS32 的游戏操作台，并且几乎忘记了任何正在运行

的 Windows 工作。让我们转到下一部分——主事件循环。

主事件循环

最难的一部分终于结束了。我正要脱口而出：主事件循环太简单了。下面讨论一下：

```
// enter main event loop
while(GetMessage(&msg, NULL, 0, 0))
      {
      // translate any accelerator keys
      TranslateMessage(&msg);

      // send the message to the window proc
      DispatchMessage(&msg);
      } // end while
```

这是什么？OK！让我们来研讨一下。只要 GetMessage()返回一个非零值，主程序 while()就开始执行。GetMessage()是主事件循环的关键代码，其惟一的用途就是从事件队列中获得消息，并进行处理。你会注意到 GetMessage()有四个参数。第一个参数对我们非常重要，而其余的参数都可以设置为 NULL 或 0。下面列出其原型，以供参考：

```
BOOL GetMessage(
     LPMSG lpMsg,          // address of structure with message
     HWND hWnd,           // handle of window
     UINT wMsgFilterMin,  // first message
     UINT wMsgFilterMax); // last message
```

msg 参数是 Windows 放置下一个消息的存储器。但是和 WinProc()的 msg 参数不同的是，该 msg 是一个复杂的数据结构，而不仅仅是一个整数。当一个消息传递到 WinProc 时，它就被处理并分解为各个组元。MSG 的结构如下所示：

```
typedef struct tagMSG
        {
        HWND hwnd;         // window where message occurred
        UINT  message;     // message id itself
        WPARAM wParam;     // sub qualifies message
        LPARAM lParam;     // sub qualifies message
        DWORD time;        // time if message event
        POINT pt;          // position of mouse
        } MSG;
```

看出点眉目来了，是吗？注意所有向 WinProc()传递的参数都包含在该结构中，还包括其他参数，如事件发生时的时间和鼠标的位置。

GetMessage()从时间序列中获得下一个消息，然后下一个被调用的函数就是

TranslateMessage()。TranslateMessage()是一个虚拟加速键转换器——换句话说就是输入工具。现在只是调用它，不必管其功能。最后一个函数 DispatchMessage()指出所有操作发生的位置。当消息被 GetMessage()获得以后，由函数 TranslateMessage()稍加处理和转换，通过函数 DispatchMessage()调用 WinProc 进行进一步的处理。

DispatchMessage()调用 WinProc，并从最初的 MSG 结构中传递适当的参数。图 2.9 表示了整个处理过程的最后部分。

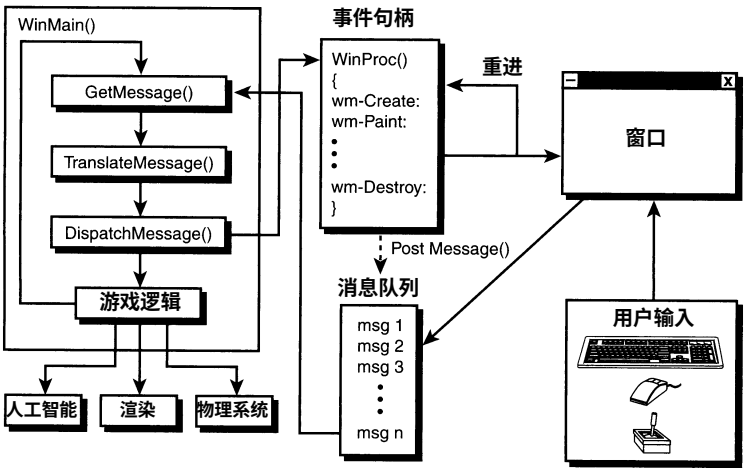


图 2.9　事件循环消息处理机制

这样，你就成了 Windows 专家了！如果你已经理解了上面刚刚讨论过的概念以及事件循环、事件处理程序等等的重要性，那你至少已经掌握了 90%的内容了。剩下的就是一些细节问题了。

程序清单 2.3 是一个完整的 Windows 程序，内容是创建一个窗口，并等候关闭。

程序清单 2.3　一个基本的 Windows 程序

```
// DEMO2_3.CPP - A complete windows program

// INCLUDES ///////////////////////////////////////////////
#define WIN32_LEAN_AND_MEAN  // just say no to MFC

#include <windows.h>   // include all the windows headers
#include <windowsx.h>  // include useful macros
#include <stdio.h>
#include <math.h>

// DEFINES ////////////////////////////////////////////////

// defines for windows
```

```
#define WINDOW_CLASS_NAME “WINCLASS1”

// GLOBALS ////////////////////////////////////////////


// FUNCTIONS //////////////////////////////////////////
LRESULT CALLBACK WindowProc(HWND hwnd,
                UINT msg,
                          WPARAM wparam,
                          LPARAM lparam)
{
// this is the main message handler of the system
PAINTSTRUCT    ps;    // used in WM_PAINT
HDC       hdc;    // handle to a device context

// what is the message
switch(msg)
   {
   case WM_CREATE:
      {
   // do initialization stuff here

      // return success
   return(0);
   } break;

   case WM_PAINT:
   {
   // simply validate the window
   hdc = BeginPaint(hwnd,&ps);
   // you would do all your painting here
      EndPaint(hwnd,&ps);

      // return success
   return(0);
   } break;

   case WM_DESTROY:
   {
   // kill the application, this sends a WM_QUIT message
   PostQuitMessage(0);

      // return success
   return(0);
   } break;

   default:break;

   } // end switch
```

```
// process any messages that we didn't take care of
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc

// WINMAIN ////////////////////////////////////////////
int WINAPI WinMain(HINSTANCE hinstance,
           HINSTANCE hprevinstance,
           LPSTR lpcmdline,
           int ncmdshow)
{

WNDCLASSEX winclass; // this will hold the class we create
HWND      hwnd;     // generic window handle
MSG      msg;         // generic message

// first fill in the window class structure
winclass.cbSize  = sizeof(WNDCLASSEX);
winclass.style    = CS_DBLCLKS | CS_OWNDC |
                    CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc    = WindowProc;
winclass.cbClsExtra    = 0;
winclass.cbWndExtra    = 0;
winclass.hInstance     = hinstance;
winclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor        = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground  = GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName   = NULL;
winclass.lpszClassName  = WINDOW_CLASS_NAME;
winclass.hIconSm        = LoadIcon(NULL, IDI_APPLICATION);

// register the window class
if (!RegisterClassEx(&winclass))
   return(0);

// create the window
if (!(hwnd = CreateWindowEx(NULL, // extended style
              WINDOW_CLASS_NAME,   // class
              "Your Basic Window", // title
             WS_OVERLAPPEDWINDOW | WS_VISIBLE,
             0,0,       // initial x,y
             400,400,  // initial width, height
             NULL,       // handle to parent
             NULL,       // handle to menu
             hinstance,// instance of this application
             NULL)))   // extra creation parms
return(0);
```

```
// enter main event loop
while(GetMessage(&msg,NULL,0,0))
     {
     // translate any accelerator keys
    TranslateMessage(&msg);

     // send the message to the window proc
    DispatchMessage(&msg);
    } // end while

// return to Windows like this
return(msg.wParam);

} // end WinMain

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

要编译 DEMO2_3.CPP**，只需创建一个** Win32 **环境下的**.EXE **应用程序，并且将** DEMO2_3.CPP **添加到项目中即可。假如你喜欢的话，可以直接从** CD-ROM **上运行预先编译好的程序** DEMO2_3.EXE**。图** 2.10 **显示了运行中的该程序。**

图 2.10　**运行中的** DEMO2_3.EXE

在进行下一部分内容之前，我还有事情要说。首先，如果你认真阅读了事件循环的话，会发现它看上去并不是个实时程序。也就是说，当程序在等待通过 GetMessage()**传递的消息的同时，主事件循环基本上是锁定的。这的确是真的；你必须以各种方式来避免这种现象，因为你需要连续地运行你的游戏处理过程，并且在** Windows **事件出现时处理这些事件。**

产生一个实时事件循环

这种实时的无等候的事件循环很容易实现。你所需要的就是一种测试在消息序列中是否

有消息的方法。如果有，你就处理它；否则，继续处理其他的游戏逻辑并重复进行。运行的测试函数是 PeekMessage()。其原型几乎和 GetMessage()相同，如下所示：

```
BOOL PeekMessage(
     LPMSG lpMsg,          // pointer to structure for message
     HWND hWnd,            // handle to window
     UINT wMsgFilterMin,   // first message
     UINT wMsgFilterMax,   // last message
     UINT wRemoveMsg);     // removal flags
```

如果有可用消息的话返回值非零。

区别在于最后一个参数，它控制如何从消息序列中检索消息。对于 wRemoveMsg，有效的标志有：

- ✧ PM_NOREMOVE——PeekMessage()处理之后，消息没有从序列中去除。
- ✧ PM_REMOVE——PeekMessage()处理之后，消息已经从序列中去除。

如果将这两种情况考虑进去的话，你可以做出两个选择：使用 PeekMessage()和 PM_NOREMOVE，如果有消息的话，就调用 GetMessage()；另一种选择是：使用 PM_REMOVE，如果有消息则使用 PeekMessage()函数本身来检索消息。一般使用后一种情况。下面是核心逻辑的代码，我们在主事件循环中稍作改动以体现这一新技术：

```
while(TRUE)
   {
   // test if there is a message in queue,if so get it
   if (PeekMessage(&msg, NULL,0,0,PM_REMOVE)
      {
      // test if this a quit
         if (msg.message = = WM_QUIT)
               break;
   // translate any accelerator keys
   TranslateMessage( &msg );

   // send the message to the window proc
   DispatchMessage( &msg );
   } // end if

   // main game processing goes here
   Game_Main();
} // end while
```

我已经将程序中的重要部分突出显示。粗体的第一部分内容是：

```
if (msg.message = = WM_QUIT) break;
```

下面是如何测试从无限循环体 while(true)中退出。请记住，当在 WinProc 中处理 WM_DESTROY 消息时，你的工作就是通过调用 PostQuitMessage()函数来传递 WM_QUIT 消息。WM_QUIT 就在事件序列中慢慢地移动，你可以检测到它，所以可以跳出主循环。

突出显示的程序最后一部分指出调用主游戏程序代码循环的位置。但是请不要忘记，在运行一幅动画或游戏逻辑之后，调用 Game_Main()或者调用任意程序必须返回。否则，Windows 主事件循环将不处理消息。

这种新型的实时结构的例子非常适合于游戏逻辑处理程序，请看源程序 DEMO2_4.CPP 以及 CD-ROM 上相关的 DEMO2_4.EXE。这种结构实际上是本书剩下部分的模型。

打开多个窗口

在完成本章内容之前，我想讨论一个你可能非常关心的更重要的话题——如何打开多个窗口。实际上，这是小事一桩，其实你已经知道如何打开多个窗口。你所需要做的就是多次调用函数 CreateWindowEx()来创建这些窗口，事实也的确如此。但是，对此还有一些需要注意的问题。

首先，请记住当你创建一个窗口时，是建立在 Windows 类的基础之上的。在该类中定义了 WinProc 或者整个类的事件处理程序。这是非常重要的细节，因此应当注意。你可以使用同一个类来创建多个窗口，这些窗口的所有消息都要传递到同一个 WinProc 中，正如由 WINCLASSEX 结构中定义 lpfnWndProc 字段指向的事件处理程序一样。图 2.11 表示了这种情况下的消息流。

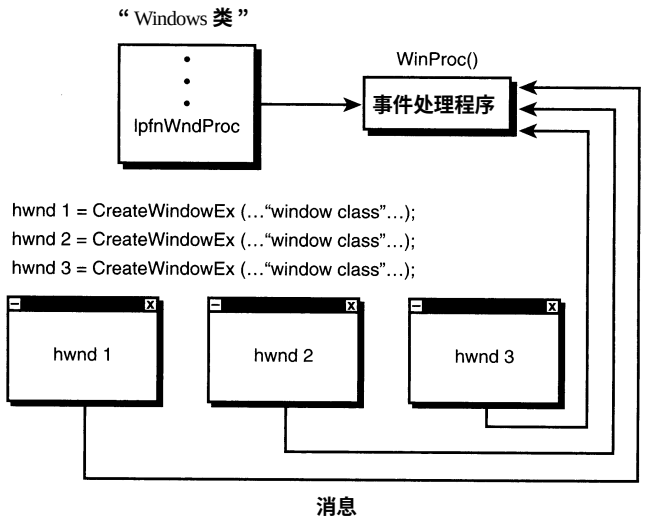


图 2.11 使用相同的 Windows 类打开多个窗口的消息流

这样可能达到你的愿望，也可能达不到。如果想应用不同的 WinProc 打开各个窗口的话，你必须创建多个 Windows 类，并且使用每一个类创建各个窗口。这样每一个类的窗口就指向各自的 WinProc 并向其传递消息。图 2.12 表示了这种情况。

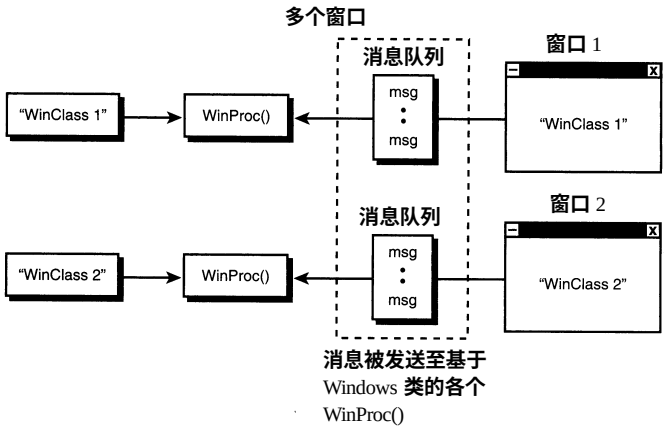


图 2.12　多个窗口的多个 Windows 类

了解了这些内容后，下面是基于同一个类来创建多个窗口的程序代码：

```
// create the first window
if (!(hwnd = CreateWindowEx(NULL,     // extended style
              WINDOW_CLASS_NAME,     // class
               "Window 1 Based on WINCLASS1",        // title
               WS_OVERLAPPEDWINDOW | WS_VISIBLE,
               0,0,      // initial x,y
               400,400  // initial width,height
               NULL,    // handle to parent
               NULL,    // handle to menu
               hinstance, // instance of this application
               NULL))) // extra creation parms
return (0);

// create the second window
if (!(hwnd = CreateWindowEx(NULL,    // extended style
               WINDOW_CLASS_NAME,      // class
               "Window 2 Also Based on WINCLASS1",  // title
               WS_OVERLAPPEDWINDOW | WS_VISIBLE,
               100,100,  // initial x,y
               400,400  // initial width,height
               NULL,   // handle to parent
               NULL,   // handle to menu
               hinstance, // instance of this application
               NULL))) // extra creation parms
return (0);
```

当然，你可能希望用不同的变量而不是同一个变量来跟踪每一个窗口，就像 hwnd 中的例子那样，其实你已掌握了其要义，如一次打开两个窗口的例子。请看 DEMO2_5.CPP 以及 CD-ROM 上相关的可执行文件 DEMO2_5.EXE。当你运行.EXE 执行文件时，应当可以看到

如图 2.13 所示的情况。注意当你关闭任何一个窗口时，两个窗口将同时关闭，并且应用程序也终止。看一看是否能够找到每次只关闭一个窗口的方法。（提示：创建两个 Windows 类，直到两个窗口都关闭之后再传递 WM_QUIT 消息。）

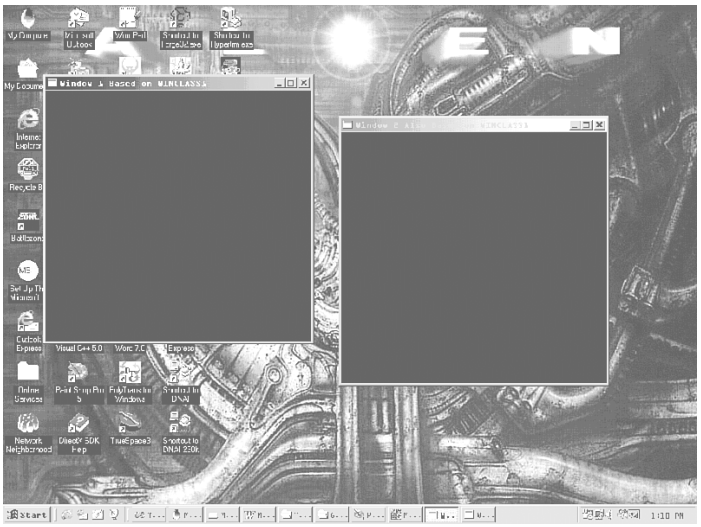


图 2.13　多窗口程序 DEMO2_5.EXE

总　　结

尽管我并不了解你，但是我还是很为你激动！到目前为止，你已具备了 Windows 编程的基本知识并需要开始掌握更复杂的 Windows 编程知识。你已了解了 Windows 和多任务的结构，并且也知道了如何创建一个 Windows 类、注册类、创建窗口、编写事件循环和句柄等等内容。因此你已经打下了坚实的 Windows 编程基础。你完成了一项最杰出的任务。

下一章中，我们将了解更多的和 Windows 相关的内容，如：使用资源、创建菜单、使用对话框以及获取信息等。