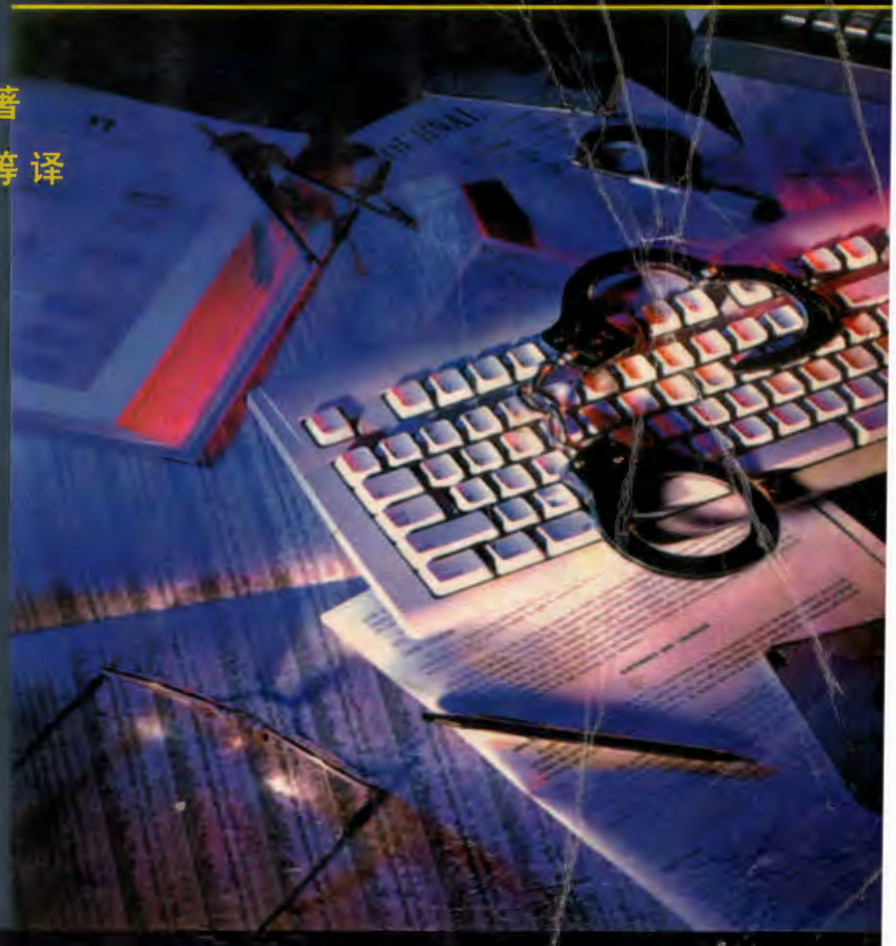
The Microsoft Windows 95 Developer's Guide

The Microsoft Windows 95

开发人员指南

(美) Stefano Maruzzi 著
周靖 王勇 宋玺如 等译



计算机软件开发
与程序设计
系列丛书

机械工业出版社 西蒙与舒斯特国际出版公司

The Microsoft Windows 95 开发人员指南

《计算机软件开发与程序设计系列丛书》

- Windows 95 开发人员指南
- Visual Basic 4开发人员指南
- Visual Foxpro 3.0 开发指南
- Windows 95 API 程序设计
- Visual Basic 4 API 程序设计
- 按实例学 Delphi 2 程序设计
- 应用 Visual Basic 开发客户机/服务器
- Java编程指南
- I/O接口程序设计入门与应用
- 精通VISUAL BASIC 4.0 多媒体程序设计

本书特色

- 介绍API Win 32中的软件开发工具以及应用程序的开发，包括：消息的重画模式及资源文件，高级用户接口，诸如弹出式菜单、对话框、表格、向导和菜单条，窗口框架的拖拽高级内存管理与动态链接库，图形设备接口，非标准输入与输出，多线程IPC和I/Windows高级技术及Windows 95外壳开发
- 适用于具有一些C,C++或VisualBasic程序设计经验而又亟待提高的读者
- 书中有大量例程可直接引用到您的应用程序中

ISBN 7-111-05459-8

9 787111 054597 >

北京华章图文信息有限公司

北京市百万庄南街14号
电话：68326444
邮编：100037

ISBN-7-111-05459-8/TP · 435
定价：86.00元

计算机软件开发与程序设计系列丛书

The Microsoft Windows 95

开发人员指南

(美)Stefano Maruzzi 著

周靖 王勇 宋玺如 等译

机械工业出版社
西蒙与舒斯特国际出版公司

本书分为两部分。前一部分讲述了 API Win32 中的软件开发、开发工具以及应程序的开发。其中着重介绍了消息的重画模式、资源文件、菜单的运用、建立窗口的艺术，对话框的管理以及预定义窗口类在 API Win32 应用程序内的开发。后一部分讲述了 Win95 的通用控件、图形设备接口、非标准输入与输出、内存管理与 DLL、多线程 IPC 和 I/Windows 高级技术、Win95 外壳开发等新技术在 Win95 中的应用。

本书可供计算机软件开发人员及大专院校电子与计算机专业的师生使用与参考。

本书版权登记号：图字：01-96-1222

图书在版编目(CIP)数据

The Microsoft Windows95 开发人员指南/(美)马入芝(Maruzzi, S.)著；周靖等译. -北京：机械工业出版社，1997.1
(计算机软件开发与程序设计系列丛书)
ISBN 7-111-05459-8

Ⅰ.T… Ⅱ.①马… ②周… Ⅲ.窗口软件，Windows95-软件开发-指南 Ⅳ.TP316-62

中国版本图书馆 CIP 数据核字(96)第 23780 号

出 版 人：马九荣(北京市百万庄南街 1 号 邮政编码 100037)
责任编辑：何伟新 东凌
北京牛山世兴印刷厂印刷 · 新华书店北京发行所发行
1997 年 1 月第 1 版第 1 次印刷
787mm×1092mm 1/16 · 49.5 印张 1236 千字
0001-5000 册
定价：86.00 元

译者的话

自从美国微软公司（Microsoft）于1995年8月发布Microsoft Windows 95之后，各种个人计算机上的应用程序都开始从DOS平台向Windows95平台转移。

对于专业程序员，或者是将要成为专业程序员的人来说，为Windows95编写应用程序看来是必由之路。国际知名的软件开发和培训专家Stefano Maruzzi在《The Microsoft Windows95开发人员指南》一书内为使用C，C++以及Visual Basic的程序员展示了怎样建立全功能的32位Windows95应用程序。在这本书中，针对Win32API开发过程的第一步骤，作者都用多个示范文件及对应的.EXE文件（可以在附带的CD盘中找到）进行了补充说明。

为了帮助广大计算机用户学习怎样从头编写32位应用程序，如何运用一些基本的和高级的技巧，从而编写出出色的Windows95应用程序，我们翻译了这本《The Microsoft Windows95开发人员指南》。参加本书翻译工作的有周靖、吴卫华、王勇、宋玺如、冀惠刚、王丽梅、张丽霞、贾银萧、李晋宏、李竹华、潘旭燕、黄为、徐茜、尤晓东、徐晓梅、黄培林、王丽梅、陈琦、陈红梅、刘涛、李明辉、刘丽、杨淑英、朱锦鸾等同志。由于时间仓促，翻译过程中难免出现错误，欢迎广大读者指正。

译者

1996.8于北京

GetOpenFileName

目　录

第 1 章　Win32 中的软件开发

Microsoft Windows 是微软公司针对配备 Intel x86 微处理芯片的个人计算机推出的第一种多任务解决方案。Windows 的第一个版本是 1.01，于 1985 年发布。在那个时候，正在使用的几乎所有个人计算机配备的都是 Intel 的 8088/86 微处理芯片。当时，第一套 80286 系统（IBM PC AT）也不过刚刚才发布（1984 年 8 月）。

不过，Windows 并没有很快地兴旺起来，这主要是由于硬件和 DOS 操作系统的内在限制决定的。另外，图形分辨率和处理器的速度也是一个很大的瓶颈。除此以外，当时的 CGA 标准只能显示 320×200 个象素，并且只有四种颜色，而且价格昂贵，这样就阻碍了它的广泛普及。另外，当时的软件开发手段落后，只有一些“高手”才会对编程感兴趣。这和近 80 年来工业化发展所取得高度生产力是很不相称的。Windows 的后续版本取得了稳定的进展，每个版本的功能都比以前更加强大，并且越来能充分发掘出硬件的潜力。

现在的情况已经完全不同了。越来越多的个人电脑用 80486 和 Pentium 芯片武装起来，80386 系统很快就成了昨日黄花。桌面计算机很容易就可以用至少 16 色达到 800×600 的分辨率。事实上，许多计算机系统都可以处理更高的视频分辨率和支持 256 色同屏显示，甚至能支持上百万种颜色同屏显示。对于新出厂的每台个人电脑来说，鼠标已经成为标准的和不可缺少的配置，几百兆的硬盘也随处可见。伴随着硬件性能的持续提高，价格的下降也是同步进行的。这样造成的结果就是：个人电脑大量充斥市场。

1992 年 4 月，微软推出了 Microsoft Windows 3.1。不久，1993 年 8 月，Microsoft Windows NT 3.1 也上市发行了。1993 年 12 月，微软公司正式宣布他们准备开发 Windows 环境的一种新的 32 位版本。Windows 环境下的第一个 32 位 API 是在 3.1 版本的 Windows NT 里出现的。要使 Win32 得到公众的接受，只有等到 Microsoft Windows 95 广泛铺开的时候才会实现。最初，这个 32 位版本的代码名叫作 Chicago（芝加哥），后来改名为 Microsoft Windows 95。在本书内，我们统一称呼为 Win95。

1.1　Microsoft Windows 的演变

最开始的时候，开发一个 Windows 应用程序意味着必须使用一种高级的编程语言，这几乎总是 C，并且还要和运行环境的软件开发包（SDK）交互作用。这些函数总合起来就是众所周知的“应用程序编程接口”（API）。

随着 NT 的降临，当 Windows API 升至 32 位的时候，它就演变成了著名的 Win32，而较老的 API 版本则叫作 Win16。通过表 1-1，我们可以大致看出 Microsoft Windows 的演变过程。

表格的“编码”栏是指构成 Microsoft Windows 环境的编码本质。API 栏是指软件开发人员在对应平台上开发应用程序时使用的应用程序编程接口。“应用程序”栏是指各 Microsoft Windows 版本所支持的程序的本质。

表 1-1 Microsoft Windows 的演变

产　品	发行日期	编码	API	应用程序
MS Windows 1.x	1985 年 11 月	16 位	Win16	16 位
MS Windows/386	1987 年 9 月	16 位/32 位	Win16	16 位
MS Windows 2.x	1987 年 12 月	16 位	Win16	16 位
MS Windows 3.0	1990 年 5 月	16 位/32 位	Win16	16 位
MS Windows 3.1	1992 年 4 月	16 位/32 位	Win16	16 位
MS Windows for Workgroups 3.1	1992 年 11 月	16 位/32 位	Win16	16 位
MS Windows for Workgroups 3.11	1993 年 11 月	16 位/32 位	Win16	16 位
MS Windows 3.11	1993 年 12 月	16 位/32 位	Win16	16 位
MS Windows NT 3.1	1993 年 8 月	32 位	Win32	32 位/16 位
MS Windows NT AS 3.1	1993 年 8 月	32 位	Win32	32 位/16 位
MS Windows NT Workstation 3.5	1994 年 10 月	32 位	Win32	32 位/16 位
MS Windows NT Server 3.5	1994 年 10 月	32 位	Win32	32 位/16 位
MS Windows NT Workstation 3.51	1995 年 7 月	32 位	Win32	32 位/16 位
MS Windows NT Server 3.51	1995 年 7 月	32 位	Win32	32 位/16 位
MS Windows 95	1995 年 8 月	32 位/16 位	Win32	32 位/16 位

Win32 代表了对 API 总体质量最显著的一次提升。它正逐渐成为应用广泛的一种目标平台，特别是发行了 Microsoft Windows 95 以后，人们对它更是刮目相看。微软的目标是发行一套统一的 Win32 开发包，使其能在 NT 和 Windows 95 支持的所有硬件平台上寻址。

微软现在这两套 32 位 API 子集会在以后几年内得以完全的合并。当前，针对 NT 和 Win95 进行 Win32 开发还存在一些差异，并且对这些差异必须引起足够的重视。NT 支持 Win32 的全集，这个全集最初是于 1992 年 7 月出版的。1993 年 8 月，随同 NT 的发行，又对它进行了一番修订。从另外一方面来说，由于 Windows 95 自身的限制，它只具有 Win32 一个子集的地位。这个子集就是我们常说的 Win32c。然而，微软的工程师已被授命在 Win95 里增加新的 Win32 API 函数，而这些函数在 NT 内却是没有的。之所以会取得这项进展，主要是由于在 Microsoft Windows 95 里引入了一种面向对象的用户界面。将来，甚至在 NT 里都会包含 Microsoft Windows 95 对 Win32 所作的全部扩展。这将在 NT 的下一个版本（当前的代码名为 Microsoft Cairo）发布时实现。表 1-2 为大家比较了当前两个 Win32 版本间的不同。

表 1-2 比较 Win32 API 在 Microsoft Windows 95 和 Microsoft WindowsNT 里的不同特性

Win32 特性	MS Windows 95	MS Windows NT
32 位内存管理	√	√
文件映射	√	√
联网	√	√
OLE 2.x	√	√
邮件槽和命名管道	√（只用于客户端）	√
Win32 线程管理	√	√
高级 GDI	√（大部分）	√
远程进程调用	√	√
MS Windows 95 UI	√	只有通用控件
GDI 转换		√
事件登录		√
安全保证		√
Unicode（统一编码）		√

以前，API 是和 Microsoft Windows 特定的版本紧密集成在一起的。但是如今，Win32 却通过两个操作系统间的微妙差别而实现了一体化。这是 Windows 编程史上最值得书写的一笔。

1.2 我们在哪里

到目前为止，我们还没有提到关于 Win32s 的任何事情。这是 Win32 的一个附加的子集，它的目标是开发出甚至有能力在 Microsoft Windows 3.x 系统上运行的 32 位编码（当然要有恰当的支持组件）。但是，这种设计方案引发了下面这个问题：应该选择什么平台来实施 Win32 开发呢？为了回答这个问题，我们可以先设计一个名为 Platform（平台）的小型的、简单的应用程序。这个应用程序的目的是告诉用户关于基础操作系统的信息。

有两个函数可以识别当前运行的是 32 位 Windows 的何种版本，它们分别是 GetVersion () 和 GetVersionEx ()。我们建议大家只使用第二个函数，它比第一个更清楚，功能也更强。下面是这两个函数的语法：

```
DWORD GetVersion (void);
DWORD GetVersionEx (LPOSVERSIONINFO osvi);
```

尽管 GetVersionEx () 要以数据结构 OSVERSIONINFO 为基础，但这两个函数都会返回一个 DWORD（双字）。GetVersion () 的返回值包含了一些信息，通过这些信息就可以知道程序是在何种 32 位平台上运行的。但是只有进行了一番处理，提取出二进制位的含义后，才有可能真正实现。请参考下面这个代码段。

```
...
DWORD dwVer;
WORD wHi;
int i;

dwVer = GetVsrsion ();
wHi = HIWORD (dwVer);
i = wHi >> 14;
switch (i)
{
        case 0:
                strcpy (szString, "MS Windows NT");
                break;

        case 1:
                strcpy (szString, "MS Windows 3.x + Win32s");
                break;

        case 3:
```

```
                strcpy (szString, "MS Windows 95");
                break;
    }
    ...
```

识别的方法是检查返回高字节的最后两位。如果为二进制的 00，就表明是 Windows NT；如果为 01，就表明是 Win32s；如果为 11，就表明是 Win95。从另外一方面来说，GetVersionEx () 则返回了 dwPlatformId 项中需要的信息，如下所示：

```
OSVERSIONINFO osvi;
DWORD version;
osvi.dwOSVersionInfoSize = sizeof (osvi);
GetVersionEx (&osvi);

switch (osvi.dwPlatformId)
{
        case VER_PLATFORM_WIN32s:
        {
                strcpy (szString, "MS Win32s on MS Windows 3.1")
        }
                break;

        case VER_PLATFORM_WIN32_WINDOWS:
        {
                strcpy (szString, "MS Windows 95");
        }
                break;

        case VER_PLATFORM_WIN32_NT:
        {
                strcpy (szString, "MS Windows NT");
        }
                break;
}
...
```

运行 Platform 以后（图 1-1），大家可能会立即对 Windows 95 的新奇特性产生兴趣。你肯定会注意到一个彩色的光标、菜单和其他窗口组件的新外观，另外还有覆盖于客户区的背景位图。这些特性和许多其他的特性都会在后续的章节中进行详细介绍。

在本书附带的 CD 里，Listing 1.1 包含了 Platform 的源代码。它强调了 Win32——32 位程序的运行环境的运用。

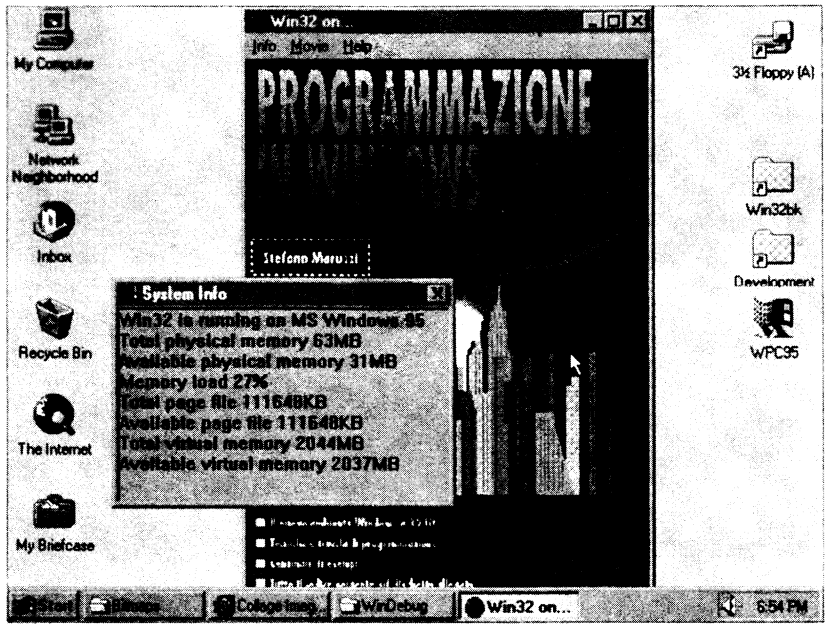


图 1-1　Platform 在 MS Windows 95 中的运行情况

1.3　32 位编程的引入

尽管出现了 Win32，同时它还在不断地进化，但是微软支持的主要 32 位平台还是 MS Windows 95 以及居于从属地位的 Windows NT。Microsoft Win32 是建立具有如下特性的程序的起点：

- ▶应用程序的执行独立于所有硬件设备以外。
- ▶图形用户界面。
- ▶Microsoft Windows 95 和 Microsoft Windows NT 之间透明的移植能力，可以移植至支持 Microsoft Windows NT 的 RISC 硬件平台。
- ▶面向对象的用户界面，减轻了用户学习的负担。
- ▶高性能的抢先式多任务和多线程。
- ▶高级的多媒体支持（声音、图形、影象等等）。
- ▶通过 OLE 技术实现的多个应用程序的对象定位。

通过这段对 Win32 的简要总结，32 位程序开发的前途应该是很明朗的了。Win32 可以应用于特定的操作系统，这种系统应能直接控制和处理 PC 硬件资源，而不必依赖于 MS-DOS 系统服务。通过图 1-2，大家可以看到系统引导以后出现的 Windows 95 面向对象的用户界面。

在 PC 的世界里，活跃着 Windows 16 位程序的许多编程高手。对于这些开发者来说，把自己的程序转换成 32 位是一件让人激动的事情。这就有机会重温一下自己的编程经历；在其中注入一些新鲜血液以后，甚至还能提高它们的性能。然而，许多新的开发者则是由于受到

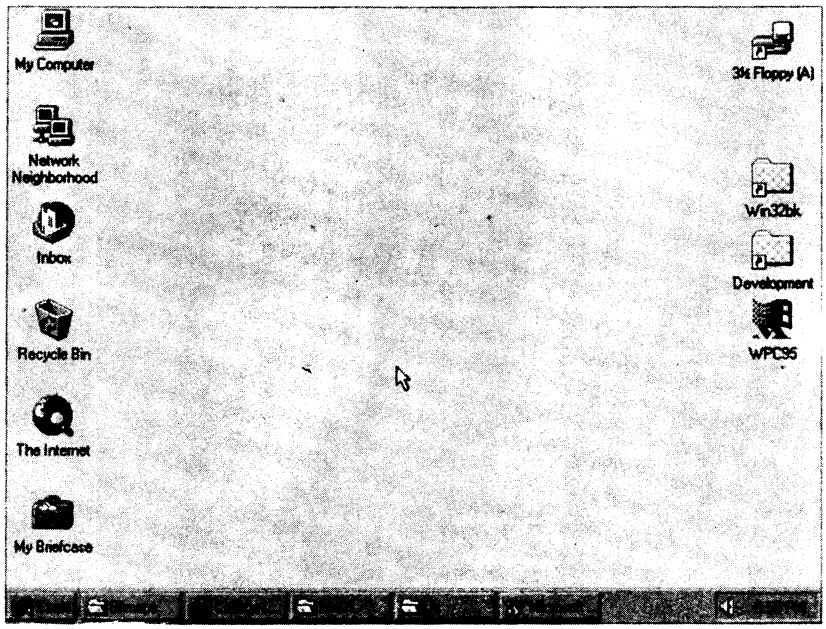


图 1-2　和以前的 Microsoft Windows 3.x 和 Microsoft Windows NT 比较起来，用 Microsoft Windows 95 装备的系统在外观上看起来颇不相同

Windows 的 32 位版本以及新 Win32 API 的强大功能的吸引，所以才投身到 Windows 编程行列中来的。无论你的背景如何，过渡至 Win32 都需要掌握一些学习技巧。

最后，Win32 是真正意义的抢先式多任务、多线程以及线性寻址内存管理（平坦内存——flat memory）。Windows 95 的新界面使你不得不重新斟酌应用程序“外观和感觉”，这通常暗示必须修正以前对用户交互进行支配的逻辑。

1.4　Windows 的硬件需求

抢先式多任务是 Win32 最具有吸引力的一个方面。发行 Windows 第一个版本的时候，几乎所有微处理芯片都是建立在 8088/86 的基础上的。Microsoft Windows 的 1.x 和 2.x 版本紧密依赖于实模式体系，几乎没有办法能够跨越这种体系内部固有的局限。随着 3.0 版本的发布，Windows 逐渐转向 Intel 的保护模式 16 位和 32 位微处理芯片。更近的 3.1 和 3.11 版本不再提供对老式的实模式的支持，那种模式已经太陈旧了。随着 Windows 的发展，以前的 8088/86 处理器已被完全淘汰。Windows for Workgroups 3.11 推出以后，80286 微处理器也面临着相同的命运。无论 Microsoft Windows 95 还是 Microsoft Windows NT，它们都是专门为 80386 处理器以及更高档的处理器设计的；换句话说，它们针对的都是 32 位保护模式。

1.4.1　Intel x86 微处理器家族

为了真正理解 16 位和 32 位 Windows 版本是如何实现多任务机制的，让我们先来看一看 Intel 微处理器的内部结构。iAPX86 家族的所有产品——8088，8086，80286，80386，80486 和 Pentium——最初都要用实模式进行自举。对于 8088 和 8086 处理器来说，这是唯一的选

择。为什么更高级的微处理器也要这样做呢？这完全是出于对整个产品系列兼容性的考虑。表1-3描述了iAPX86产品家族的不同特征。

表1-3 在个人计算机中使用的Intel iAPX86家族的处理器

iAPX86	自然模式	仿 真	RAM	虚拟内存	段
8088	实模式	无	1MB	不支持	64K
8086	实模式	无	1MB	不支持	64K
80286	保护模式16位	实模式	16MB	1GB	64K
80386	保护模式32位	实模式，保护模式16位	4GB	64TB	4GB
80486	保护模式32位	实模式，保护模式16位	4GB	64TB	4GB
Pentium	保护模式32位	实模式，保护模式16位	4GB	64TB	4GB

在这儿，我们有必要解释一下表1-3中“虚拟内存”栏的含义。80286系统的值是1GB，这意味着286处理器可以在这样大的一个内存区域内寻址。1GB这个值很容易就可以推算出来：我们只需要注意到LDT和GDT的每个描述符最多只能访问长度为64K的段。这样一来，64K×8192个描述符×2个表，结果就是1GB。

正如大家在表1-3中看到的那样，iAPX86家族的所有成员都是先从实模式开始运行的，以后再根据安装的操作系统改变运行模式。MS-DOS是完全以实模式为基础的。所以，MS-DOS不能利用高级处理器提供的任何一种增强特性，比如虚拟内存和大内存寻址空间等等。

1．16位实模式

对实模式进行管理的规则是比较简单的（特别是和保护模式比较起来），80286，80386，80486和Pentium处理器都可以使用实模式。在实模式里，寻址空间被限制在1MB以内，这是由于地址总线的宽度为20位（2^{20}=1MB）。实际寻址是通过“段地址：偏移地址”对来实现的。通过成对使用不同的值，就可以访问1MB内存内的任何位置。例如，通过下面这张表可以看出：不同的“段地址：偏移地址”对可能指向内存的同一个物理位置。

“段地址：偏移地址”对	物理位置
1000：40	10040
1001：30	10040
1002：20	10040

对于这种寻址方案来说，一个重要的注意事项在于，它没有提供任何一种内存保护机制。如果想实施一个有效的多任务运行环境，内存保护是不可缺少的。上面这三个“段地址：偏移地址”对明确揭示了这个概念。

通过图1-3，我们可以看到Intel处理器是如何利用20位总线来执行地址引用的。

2．16位保护模式

80286处理器设计工作于16位保护模式下。实模式和保护模式的根本区别在于用于访问内存位置的逻辑不同。即使在这种情况下，我们看到的也是一对16位值——“段地址：偏移地址”。

就像刚才看到的那样，在实模式下，我们获得了一个由20位组成的基准地址。整个操作简单、快速，这就是MS-DOS系统性能比较高的原因。相反，在16位保护模式下，一个地址

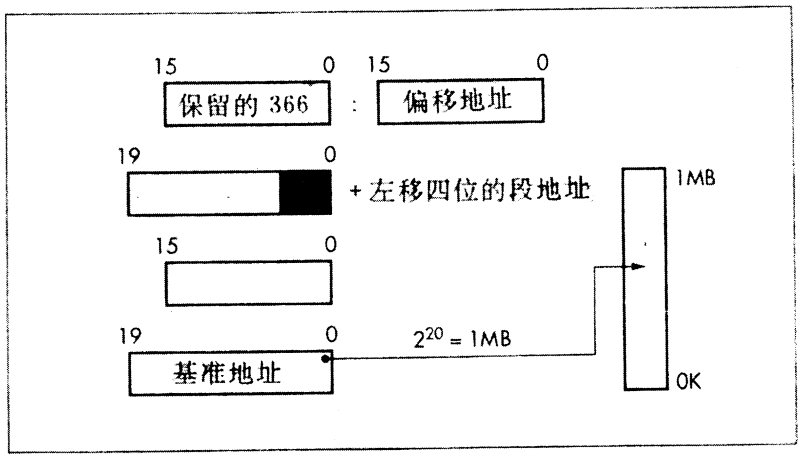


图 1-3　运行于实模式的 Intel iAPX86 处理器体系的寻址方案

的“段地址”部分被系统解释成三部分不同的信息：依次占用 2，1 和 13 个二进制位。

开头两位称为 RPL（请求符优先级，Requestor Privilege Level），它显然有四种不同的二进制组合：00，01，10 和 11。RPL 指示处理器和操作系统对正在执行的处理进行检查，看看它们的优先级之间是不是有不一致的地方。这是对系统执行的处理进行保护的一种简单方式。

段地址的第二个部分只有一个二进制位，我们把它叫作“表指示符”（Table Indicator，或者 TI）——可以是 0 或 1。表指示符的工作原理就像铁路上用的一个扳道叉。当它的值为 1 的时候，就表明更进一步的信息可以在 LDT（局部描述表，Loacal Description Table）里找到。否则，假如为 0，就会到 GDT（公共描述表，Global Description Table）里寻找信息。

段地址中的剩余 13 个二进制位叫作“选择符”（Selector）。选择符起着对表指示符生成的表进行索引的作用。描述表是一个对象，它不能超出 64K 的长度限制，因为它是由 8192 个描述符组成的，每个描述符有 8 个字节长。选择符也是一个对象，它有 13 个二进制位的长度。所以，一共有 8192 种可能的组合（$2^{13}=8192$）。一旦表指示符已经选中了一个表，就可以通过读取选择符访问对应的描述符。无论 LDT 还是 GDT 都是系统内存的一部分，和普通的内存段一样，它们的最大长度也不能超过 64K。有两个寄存器（LDTR 和 GDTR）总是随同当前使用的 GDT 和 LDT 一起载入的。

这些寄存器的长度可以达到 40 位（24 位用于地址，16 位用于描述表的尺寸）。LDT 和 GDT 的结构是类似的，但是操作系统对它们的用法却是各不相同的。OS/2 1.x 是真正利用了 16 位保护模式的唯一一种操作系统。在 OS/2 1.x 里，GDT 的任务是对和标准操作系统组件（系统内核和设备驱动程序）有关信息和引用进行跟踪，并且为每次活动的操作保留一个描述符。为了进行比较，大家可以把一个 LDT（或者 GDT）想象成一幢 8192 层的摩天大楼。编入一个段地址的选择符可以想象成电梯间的按钮，为了到达希望的楼层——或者说描述符，必须按下一个对应的按钮。如图 1-4 所示。

每个描述符的头两个字节未能在 80286 处理器中使用。接着的一个单字节叫作“存取字节”（access byte），它描述了正在引用的内存区域的一些特征。这两个八位二进制包含了我们

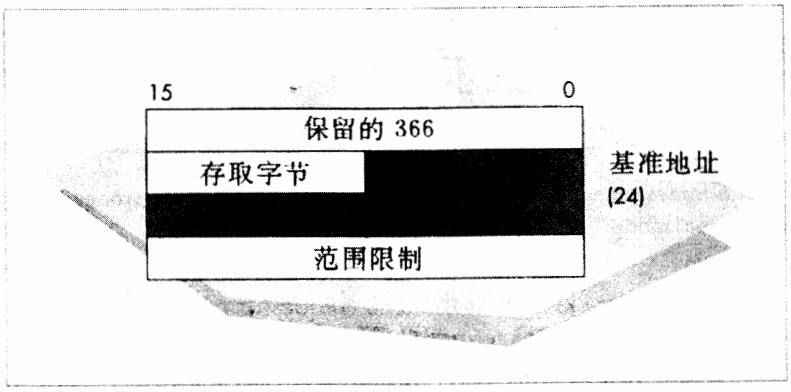


图 1-4　16 位保护模式下面的描述符是一个八字节长的对象，
它被分割成四个功能不同的区域，这样就允许你
对系统的任何一个内存区域寻址

称之为 DPL（描述符优先级，Descriptor Privilege Level）的东西。DPL 是一种特定的数据，处理器把它与当前段地址内的 RPL 进行比较，从而比较出指令执行的优先级别。下面这张表阐述了存取字节里这 8 个二进制位在 16 位保护模式处理器内的运用情况。

存取字节	二进制位的数目	分类
P	1	存在位
DPL	2	描述符优先级
S	1	段描述符
Type	3	段类型和存取类型
A	1	访问

描述符内的三个字节是系统的基准地址。由于有 24 位可用，所以物理寻址空间就变成了 16MB（参考表 1-3），因为 $2^{24}=16$MB。描述符的最后两个二进制位叫作“范围限制”(limit)。系统利用这种信息来定义内存段（正在通过段地址访问）的总长度。

保护模式和实模式之间存在大量区别，这些区别是很明显的。在实模式下，内存寻址速度很快，并且是直接和线性的。在 16 位保护模式下，内存寻址必须先经过一张描述表（LDT 或者 GDT）。只有先完成这一步骤以后才能访问物理内存。

在实模式下，段地址代表的值已经是内存段的起始基准地址了；实际的地址是由 CPU 计算出来的——只需向左面简单地移动 4 位即可。换句话说，这种运算等效于把段地址乘以 16。

在 16 位保护模式下，段地址代表的值用于访问一个正确的描述符，这个描述符依次包含了一些内存对象的物理地址。除此以外，在保护模式下，我们获得的是由有限值代表的内存段的总长度。这种信息的宽度是 16 位，所以它可以表达一个 64K 大小的区域。所以假如某个基准地址代表的是一个内存段的起点，那么“基准地址＋范围限制”就代表了那个段的终点。

迄今为止，我们还没有讨论过关于“偏移地址”的任何话题。这个值必须增加到基准地址中去，否则无法计算出一个内存位置的物理地址。很显然，“基准地址值＋偏移地址”的值

必须小于“基准地址值＋范围限制”的值。否则，处理器就会产生一个“常规保护错误”(General Protection Fault)，或者称为 GP 错误。这是一种致命的错误，Windows 3.0 把它解释成一个“不可恢复的应用程序错误”(Unrecoverable Application Error)，即 UAE。Windows 3.1 则用一个多少友好些的对话框向用户报告自己碰到了一个 GP 错误。这个对话框试图对碰到的问题给出一个合理的解释，但是我们经常看到的都是“系统崩溃”一种结局。事实上，这就是“保护模式”这个术语的全部含义。处理器本身会禁止一个应用程序访问它自己以外的内存区域。这样一来，处理器就保护了正在执行的其他程序。系统作为一个整体也得到了保护。

80286 是 iAPX86 家族内具备环形结构的第一种处理器。按照这种方案，操作系统（内核）的不同任务与应用程序很容易就可以分隔开。环形 0 分配给操作系统，环形 3 则分配给应用程序。系统可以对应用程序施加影响，但是颠转过来就不行了。系统设计人员可以使用环形 2 和 3 在几个层中对操作系统进行分配。然而，无论 NT 还是 Win95 都证明 2 个环已经足够了。

让人非常不解的是，微软已经在 Microsoft Windows 的末端用户间散布了虚拟内存(虚拟寻址）的概念。事实上，在这个术语的后面，Microsoft Windows 3.x 只是定义了一个硬盘区域，简单地把它用于内存交换目的。尽管这是一种有趣和实用的技术，但它与真正的虚拟内存却沾不上一点边（硬盘根本就不是“虚拟”的）。

Microsoft Windows 3.x 在进行系统实施的时候，它只利用了基础硬件体系的一部分特性。所有 Win16 应用程序都在一个单一的物理地址空间内工作，这个空间的大小最多不能超过 16MB。虚拟寻址方案的不同部分之间并没有优先级的差别。这种方案可以获得较高的性能，并且能够实现简单的进程间通信 (InterProcess Communications，IPC)，但是它的健壮程度和可靠性还是远远不够的。

3. 32 位保护模式

32 位保护模式的工作原理类似于 16 位保护模式。起点还是一个“段地址：偏移地址对”，但是有一个显著的区别在于，第二个部分（偏移地址）变成了 32 位的宽度。就像以前叙述的那样，段地址仍然被分割成了三个组件。此外，两个 GDT 和 LDT 表无论在长度上还是在结构上都与 16 位保护模式类似。

32 位保护模式与 16 位保护模式的第一个有趣的区别在于它们的描述符。在 32 位保护模式下，8 个字节（64 个二进制位）全都派上了用场，其中定义了下述条目：

条目	长度
基准地址	32 位
范围限制	20 位
存取权限	12 位

我们注意到，上述的存取权限比 80286 的处理器结构增加了半个字节（四个二进制位）。这些字节提供的信息具有特殊的作用。这半个字节内的其中一位是用字母 G 来标识的，它代表“间隔尺寸”(Granularity)。假如这个二进制位的值为 0，范围限制的量度单位就等于 1 个字节，总共能指定 1MB 的长度范围。在另外一方面，假如 G 等于 1，量度单位就是 4K，总共能指定 4GB 的范围限制值。

表 1-4　描述了 32 位保护模式下存取字节的内部结构。

表 1-4　32 位保护模式下存取字节的内部结构

15	8	0	15	8	0
基准地址 31：24 段地址基准 15：00	存取字节	范围限制 19：16	字节 段地址限制 15：00		基准地址 23：16

基准地址＝(00:15)＋(16:23)＋(24:31)＝32 位

范围限制＝(00:15)＋(16:19)＝20 位

存取权限＝(24:31)＋(20:23)＝12 位

存取字节

1	1	1	1	1	2	1	4
G	D	0	AVL	P	DPL	S	TYPE

存取字节中的位	说明
G(间隔尺寸位:)	定义 80386 描述符中的量度单位
AVL	为系统设计保留
D	缺省的运行尺寸(0＝16 位段,1＝32 位段)
P	段存在位
DPL	描述符优先级
S	系统段(0＝系统段,1＝CS/DS)
TYPE	段类型

在 16 位保护模式下，范围限制的值与一个段的最大尺寸是对应的。在 32 位保护模式下，Intel 的工程师不得不作出不同的决定。存取字节内包含了一个特殊的“间隔尺寸”位，这个位为范围限制值提供了一个乘法因子。假如间隔尺寸位为 0，量度单位就是一个单字节。所以，一个段不可能超出 1MB (1 个字节×2^{20}＝1MB)。假如把间隔尺寸设成 1，量度单位就变成了一个完整的页，内存大小量度单位就有 4096 个字节长。在后面这种情况下，内存段的最大尺寸可以达到 4GB (4096 个字节×2^{20}＝4GB)。这种方案把描述符的长度调节在 8 个字节以内，所以它和以前的 16 位模式是兼容的。如果不这样做，甚至范围限制值的宽度都必须达到 32 位，这样很快就会造成描述符可用空间的枯竭。

假如你准备计算 32 位保护模式下的可用虚拟地址空间，最后得到可能是一个天文数字。倘若“间隔尺寸”位等于 1，那么每个描述符都可以访问一个 4GB 大小的内存段。这样一来，计算出来的虚拟地址空间就是 4GB×8196 个描述符×2 个表＝64TB！和它的前辈比较起来，80386 处理器的计算潜力是无限的。

然而，我们还要注意一个潜在的问题。内存分段结构是 Intel 处理器的典型特征，但是其他处理器却不具备这项特征，比如 RISC（精简指令集计算机）。如果希望在两个不同的平台间对应用程序进行移植，那么分段结构就是一种不可逾越的障碍。出于对这个原因的考虑，微软公司决定把 Win32 设计成能完全移植的 API，使其能在所有 32 位的硬件框架中运行。这种设计意图通过 Microsoft Windows NT 可以很清楚地看出来，NT 现在可以同时运行于 Intel，PowerPC，MIPS 以及 Alpha 处理器。所有这些处理器都是 32 位的，尽管它们在硬件结构上存在很大的差异。

1.4.2　消除分段限制

为了避开由于内存分段带来的问题，专家们在硬件兼容性与移植需求间进行了一番妥协。Microsoft Windows NT 在 Intel 处理器中运行的时候，和 Microsoft Windows 95 一样，如果

要计算内存地址，段地址内包含的值就不再计入其中。段地址的目的是标识指向内存区（或者内存段）的一个描述符，这个内存区正是我们希望访问的位置。拥有一定数目的描述符就意味着有一定数目的内存段。尽管这种分段技术为 Intel 处理器体系带来了很强的灵活性，但它也是实现移植的一个巨大障碍。

除此以外，我们完全可以把一个单独的 4GB 内存段当作整个操作系统和所有正在执行的应用程序的唯一运行空间，对于这一点是没有什么限制的。总而言之，32 位保护模式下一个单独的 4GB 段已经足够大了，它已经远远突破了 16 位保护模式的限制，后者最多仅能使用 16MB RAM 和 64KB 的内存段。

在实践运用中，Microsoft Windows NT 和 Microsoft Windows 95 的寻址方案完全以偏移值为基础。由于采用了把一个单独的段用于整个操作系统的技巧，所以我们就获得了一个令人满意的内存寻址方案——线性并且可以控制，同时完全以 32 位地址单位为基础。事实上，偏移地址的值足以覆盖一个 4GB 的内存区域。

然而，刚才讨论的策略并没有在 Microsoft Windows NT 或者 Microsoft Windows 95 里得以实施。内存的量度单位是“页”。很显然，在 32 位计算环境里，内存分段已经消失了，这是由于页已经把它们排挤掉了。这就意味着，例如，无论在 NT 还是在 Windows 95 里，从本质上说，内存段不会产生互相冲突的问题，因为所有东西使用的都是同一个段。相反，这时会碰到另外一个问题，那就是内存页的冲突。假如某个进程试图访问一个页内包含的内存位置，同时该进程本身不能对这个页进行有效的控制，这时就会产生问题。出于这方面的考虑，Microsoft Windows 95 或者 NT 系统使用了 4GB 的虚拟地址空间。这个值与计算机能够安装的最大 RAM 数量是对应的。操作系统能够直接控制的最多也只能有这么大的内存。4GB 内存总共被分割成一百万个页，每个页 4096 个字节。因为我们用一个单独的段来容纳操作系统和应用程序，这种限制的直接后果就是最大地址空间等于最大的安装 RAM 数量。

1.4.3 页的结构

首先，让我们把注意力放在偏移地址上面。大家也许会相信 32 位计算已经完全解决了内存管理的问题，理由在于它可以容纳任何想象得到的操作系统（$2^{32}=4GB$）。然而，事实却并非如此。假如把偏移地址当作对内存位置的直接标识来使用，那么每套 32 位系统都需要安装 4GB 物理内存才行。这样会使所有的 SIMM 生产厂家欣喜若狂，也会使 PC 的价格居高不下。如果真正按照这种逻辑运行，虚拟地址就总能与对应的物理地址相同。但是实际的情况是，谁都不会安装这么多内存，真正的 RAM 数量会对虚拟地址空间进行限制，所以这是一种毫无意义的方案。

通过内存分页技术，我们可以仿真 4GB 范围内的一个物理地址，而不管其真正安装的物理内存容量是多少（前提是使用一部分硬盘空间）。通过偏移地址，我们可以把一个线性地址转换成等同的物理地址。如果一条指令准备寻找一个页，而这个页当前并未在内存里，该指令就会标识自己碰到了异常事件。这时，目标页会从硬盘中读取出来（也许要先把其他一些内存页写入磁盘，以便释放出足够的 RAM）。随后，碰到异常事件的指令将重新执行一遍，这一次在指定的内存地址处就肯定能够找到需要的页了。

在 Windows NT 和 Windows 95 里，32 位偏移地址被真正分隔成了三个部分，各自占用 10，10 和 12 个二进制位，具体的分配方案如下所示：

▶页目录条目，从第 22 位到第 31 位。

▶页表条目，从第 12 位到第 21 位。
▶偏移地址，从第 0 位到第 11 位。

31	22	21	12	11	0
页目录条目		页表条目		偏移地址	

从左面开始，头 10 位部分是页目录条目（Page Directory Entry，PDE）。PDE 跟随在页表条目（Page Table Entry，PTE）后面，PTE 再跟随于真正的偏移地址后面。操作系统建立一个新的 Win32 进程时，会先分配一块特殊内存区域，其中填写了和该进程有关的信息。这也就是我们通常所说的“进程现场”（process context）。作为所有信息的一部分，进程现场包含了进程处于活动状态时由控制寄存器＃3（CR3）装载的页目录地址。一个页目录被分割成 1024 个条目，每个条目的长度为 4 个字节（1024×4 个字节＝4096 个字节）。10 位 PDE 指出了每种特定条件下应该选择的正确条目（2^{10}＝1024）。如表 1-5 所示。

	3	2	1	1	1	1	1	1	1
页帧地址 31：12	AVAIL	00	D	A	PCD	PWT	U/S	R/W	P

左面的 20 位指定了页帧的地址、访问页表的途径以及谁是一个固定的系统表。请记住，整个 4GB 的地址空间是由一百万个页组成的（1 百万×4096 个字节＝4GB）。一旦选中了页表，系统就会对最初 32 位偏移地址的第二部分——页表条目进行控制，使其定位于正确的条目。

表 1-5　页目录或者页表条目

标志	说明	标志	说明
P	存在位	PCD	页缓冲屏蔽
R/W	读/写	A	已访问
U/S	用户/主管	D	冗余
PWT	页写透明	AVAIL	可用

就结构来说，页表与页目录是完全相同的。通过读取 PTE 的前 20 位，分页算法最终能够指向包含了实际数据或者代码的页，即页帧。采用这种方式，内存寻址方案几乎达到了顶峰。通过一个 32 位的值，系统就知道应该选择一百万个页的哪一个，并且计算出它的基准地址。32 位地址的最后一部分是 12 位的偏移地址本身，它可以让系统访问一个页内的任何一个字节（2^{12}＝4096）。整体的寻址方案至少要用到三种页（页目录、页表以及页帧），如图 1-5 所示。

每次计算一个地址的时候，并不能保证对应的页当时肯定在物理内存内。事实上，在运行 Windows 95 的系统里，如果安装的 RAM 较少（比如 4MB），许多页都会驻留于页交换文件里。这就意味着即使一个简单的操作，三个页都有可能碰到异常情况；假如存在位的值为 1，就表明这个页正在物理内存里。

尽管需要一定的 CPU 时间，但是处理页异常事件的速度仍然是相当快的。

1.4.4　后备缓冲区的转换

为了加快对内存页的定位，32 位处理器把最近用过的条目存储到一个特殊的缓存区域

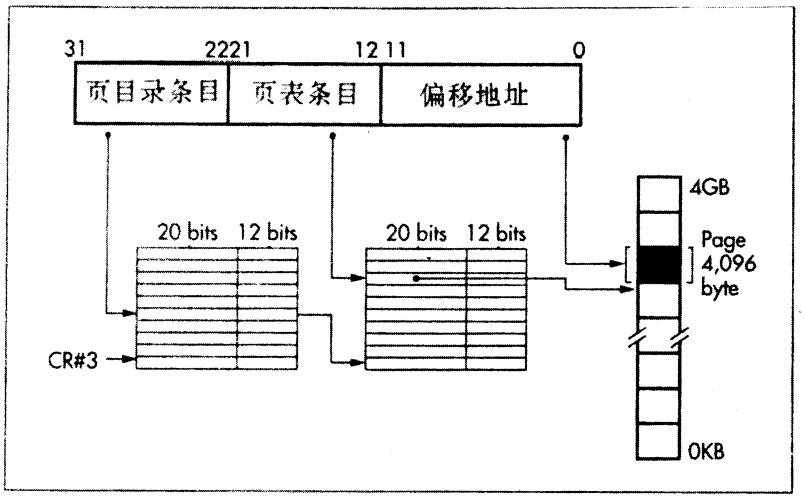


图 1-5 Windows 95 和 Windows NT 的寻址方案

里，我们把这个区域叫作“转换后备缓冲区”（TLB）。对于普通的应用来说（PL〉0），系统是不可能访问 TLB 的；只有一个操作系统任务（PL＝0）才能对它进行更新和管理。把一个页从 RAM 拷贝到分页文件内的时候，就需要用到这个缓冲区。

1.4.5 虚拟 8086 模式

Intel 的 32 位处理器提供了另外一项有趣的特性，那就是可以在不考虑 32 位保护模式的前提下执行 8086 指令。这就是我们通常所说的虚拟 8086 模式。一些软件组件可以对硬件进行特殊的控制，通过这些软件组件之间的交互作用，便可实现在一个虚拟机内对 8086 处理器进行仿真：

- ▶处理器提供了一套虚拟寄存器、一个虚拟内存地址空间以及相应的中断，它们可以支持和执行所有相关的指令。
- ▶根据当前的执行环境，并且依从特定的规则，软件对外部接口（I/O、中断和异常事件处理）进行控制，使其成为一个 VM，即虚拟机。

我们把对 VM 进行控制的软件组件叫作虚拟 8086 监视程序。这是一种 32 位的软件模块，它可以对实施虚拟机所需的基本功能进行控制。由于采用了这种方案，即使在装载了许多虚拟程序以后，Windows 95 的一个 DOS 区仍然能腾出 600KB 的常规内存容量。尽管与 Win32 开发毫无关系可言，与 DOS 的兼容性仍然是 Win95 的一个关键特征。

1.5 系统信息的管理

到此为止，我们就 Windows 95 与硬件的关系进行了一系列简短的介绍。在这一小节里，我们准备运行一个小型的 Win32 程序，以便实际运用刚才提到一些信息。设计这个程序的时候，我们很明显地利用了一些有用的 API（如图 1-6 所示）。

这个例子的名字叫作 Sysinfo，在本书附带的 CD 上，读者可以在 Listing 1.2 中找到它的源代码。该程序调用函数 GetSystemInfo（）来收集关于当前硬件和软件的信息。得到的结果再传递给一个 SYSTEM_INFO 数据结构：

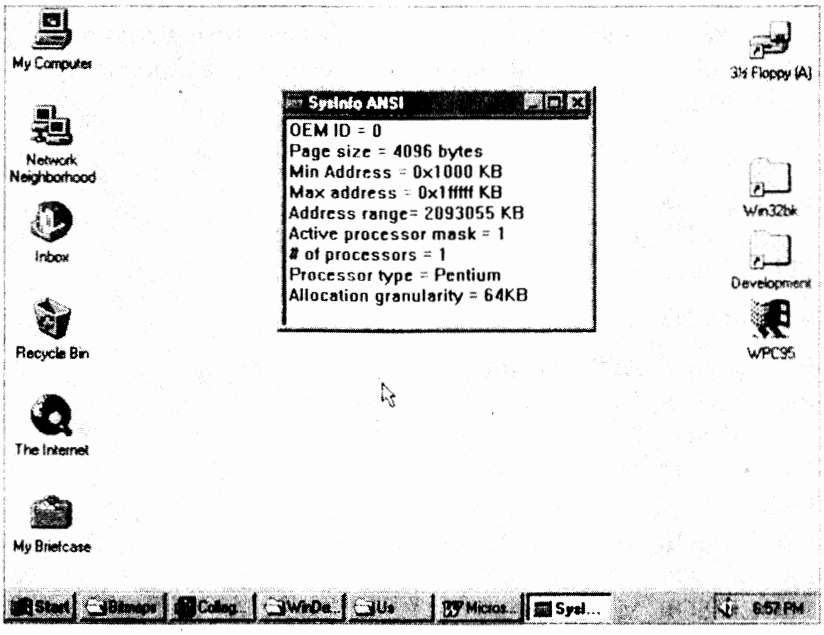


图 1-6　Sysinfo 在 Windows 95 中的运行情况

```
#include〈winbase.h〉
VOID GetSystemInfo（LPSYSTEM_INFO lpSystemInfo）;

typedef struct _SYSTEM_INFO
{ // sinf
    DWORD dwOemId;
    DWORD dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAddress;
    DWORD dwActiveProcessorMask;
    DWORD dwNumberOfProcessors;
    DWORD dwProcessorType;
    DWORD dwAllocationGranularity;
    DWORD dwReserved;
} SYSTEM_INFO;
```

稍微注意一下 SYSTEM_INFO 的所有项，我们立即会对其中的 dwPageSize 产生兴趣，这主要应归功于我们以前对这方面问题进行的大量评述和考虑。在 Windows 95 系统里，dwPageSize 的值总是设置成 4096，这是 Intel 处理器体系的标准页尺寸。处理器应该是下面这些类型的其中之一：

Windows 95

PROCESSOR_INTEL_386
PROCESSOR_INTEL_486
PROCESSOR_INTEL_PENTIUM

Windows NT

PROCESSOR_ALPHA_21046
PROCESSOR_INTEL_386
PROCESSOR_INTEL_486
PROCESSOR_INTEL_PENTIUM
PROCESSOR_INTEL_860
PROCESSOR_MIPS_R2000
PROCESSOR_MIPS_R3000
PROCESSOR_MIPS_R4000
PROCESSOR_PPC_601
PROCESSOR_PPC_603
PROCESSOR_PPC_604
PROCESSOR_PPC_620

dwAllocation“间隔尺寸”项让我们产生了一些混淆和疑问，因为在16位的世界里，64K这个内存段尺寸是众所周知的。但是，这儿的64K值与那个“臭名昭著”的64K是一点关系都沾不上的。早在几年前，NT开发组就提出了64K这个数字，因为他们不得不决定一个页尺寸，以便满足当前和未来的需要。

正如我们已经看到的那样，在Intel平台上面，一个页就是一个拥有4096字节的对象。在DEC的Alpha机器里，一个页被转换成了8K的结构。所以，我们完全可以作出这样的预测，那就是未来的硬件体系会使用越来越大的页尺寸。64K这个分配“间隔尺寸”正好说明了这一点，它只适用于NT下面的Win32开发环境。考虑到Win95并不是一个可移植的操作系统，所以在专门针对Win95开发一个新的应用程序时，dwAllocation“间隔尺寸”项造成的影响和重要性都会大打折扣。

SYSTEM_INFO也揭示了处理器的数目（dwNumberOfProcessors）和它们的类型(dwProcessorType)，并且提供了应用程序的内存地址范围。大家可以仔细研究一下CD盘上存储的Listing 1.2——Sysinfo，该程序展示了如何通过调用GetSystemInfo()来获取一些特定的系统信息。

如果把Platform和Sysinfo这两个程序的源代码合并到一起，就可以生成一个Win32程序，它能够立即反馈出某些关键的系统参数。具体的编码工作请读者自己完成！

1.6 抢先式多任务对开发的影响

到目前为止，我们已经能够知道下层运行的是何种Win32操作系统，以及一些基本的特征，比如页尺寸和系统安装的处理器数目等等。然而，在加深我们对Win32内存管理的理解

之前，还不得不先探索一下抢先式多任务的概念，从而了解它对我们开发工作会带来什么影响。

Windows 一个显著的特性在于它有能力处理多任务应用程序。在计算机里同时进行多个任务和我们现实生活中的情况相差并不大。人的大脑也许是当前最好的一种多任务设备，至少就它的灵活性（而非速度）而言是这样的，其他任何设备都无法望其项背。几乎所有人都会对 Win16 里的非抢先式多任务记忆犹新。假如从当前正在运行的进程切换至另外一个，那么其间的响应时间令人简直无法忍受。Win16 实现的只是多任务的一种有限形式，对于当今庞大和复杂的应用程序来说，它们产生的系统负荷是这种初级多任务无法承担的。

Intel 处理器的功能非常强大，它们可以在一秒钟内执行数百万条指令。为了构建一个真正的多任务环境，32 位处理器配合 32 位操作系统才是这种环境的基本原料。尽管 Windows 95 内部仍然保留了一些 16 位编码，但它仍然是一种抢先式多任务操作系统。为了达到这个目的，系统必须在当前运行的所有进程之间对 CPU 时间进行共享，这样才能确保每个进程都能频繁地访问处理器，并且实现指令的连续执行。这样一来，每个 Win32 进程都需要分配一个优先级，系统调度程序利用这种优先级来决定哪一个时刻该运行哪一个进程。

具有最高优先级的进程（或者更准确地称为“线程”）就是当前正在运行的那一个。驻留于 CPU 内的这个线程会在什么时候中止呢？一种情况是属于它的时间片到期，另一种情况是加入了另外一个具有更高优先级的线程。通过不断从一个线程切换至另一线程，就给人留下了所有活动线程同时运行的印象。线程并非随时都需要运行。经常发生的一种情况是某个线程需要等待一些用户输入，或者等待一个即将发生的 I/O 请求，否则便无法继续执行下去；有些线程则本来就已处于挂起状态。不管当前的状态如何，在一个多任务环境里，进程和线程间的同步都是一个关键性的概念。除此以外，所有进程都必须对某些物理设备进行共享，比如键盘、鼠标和屏幕等等。叠置窗口、多个输入请求、某些特定的 I/O 规则以及优先级的提高都是 Win32 用于实现抢先式多任务的方式。

1.7 Windows 3.x 使用的老式多任务

Windows 3.x 所实现的多任务处理是一种相当有限的形式，这通常反映在三个方面：协作、软件和非抢先式。由于不存在特定的硬件层，无法帮助系统工程师开发出真正的多任务环境，所以就只能建立多任务的一种不完整的形式。

正如我们早先提到的那样，对进程间的分隔地址空间进行定义的时候，或者根据不同的应用程序对操作系统进行区分的时候，实模式体系无法从硬件上提供任何帮助。由于所有这些限制和暗示，为了开发一个具有多任务形式的软件，我们只能采取一种方案。这种方案中的关键组件就是 GetMessage（）函数，该函数位于消息循环以及一个典型 Win16 应用程序的总体结构以内。

协作、软件和非抢先式多任务

每个 Win16 应用程序都应该满足两种状态的其中之一。这个应用程序既可能正在一个窗口进程内处理消息，也有可能正位于 WinMain（）消息循环的 GetMessage（）函数执行部分里，等待一条消息的到达。为了顺利实现协作式多任务，必须假设不同的应用程序通过一些“友邻规则”对 CPU 进行“共享”。如图 1-7 所示。

现在假设你准备在桌面水平移动鼠标。鼠标驱动程序可以对这种物理性的移动进行处理，

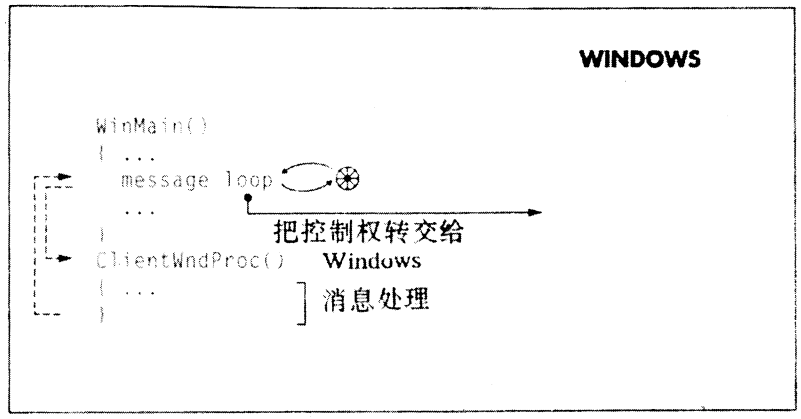


图 1-7 Win16 的协作式多任务必须依赖一个应用程序处理完相应的消息后立即释放 CPU 资源

并在 Win16 内部生成一个特定的事件。这个事件将临时性存放于系统的消息队列内——所有事件最初都要放到这儿来。每个应用程序都有它们自己的消息队列，其中只包含了与特定窗口对应的消息，这些窗口是在任务执行期间建立的。

在一个 Win16 系统里，执行消息循环内的 GetMessage () 函数时，所有任务都会暂时挂起。除非队列中进入了一条消息，否则 GetMessage () 不会返回任何值。在这段等待时间内，GetMessage () 会把控制权转交给 Windows，Windows 再把 CPU "转交" 给位于 GetMessage () 调用内的另外一个 Win16 任务。当一条消息抵达以后，GetMessage () 就会取回这条消息，然后在 TranslateMessage () 和 DispatchMessage () 的帮助下，把它传递给适当的窗口进程，在完成相应处理。下面这个代码片段阐示了消息循环的常规结构：

```
...
while (GetMessage (&msg, NULL, 0, 0))
{
    TranslateMessage (&msg);
    DispatchMessage (&msg);
}
...
```

DispatchMessage () 的任务是把执行权转交给对应的窗口进程。在消息的执行期间，CPU 的全部精力都放在这个操作上面，整个系统里不会执行其他任何操作。一旦处理完毕，执行命令流程就会离开当前的窗口进程，返回消息循环，然后执行另外一个 GetMessage ()，检查当前是不是有新的消息存在。

由于 Win16 这种协作式多任务的本质如此，所以程度开发人员的目标就是让每条消息的执行达到非常快的速度。这样才能尽可能快地返回消息循环，从而让其他应用程序有机会得以运行。对消息进行处理的时候，对于用户用键盘或者鼠标输入的任何命令，Win16 都不会

理睬，置若罔闻。由于缺乏对硬件的支持，所以就限制了系统用一个调度程序来设置优先级和定义时间片的大小。

1.7.1 Win32 的多任务

随着 Win32 的降临，应用程序开发的背景发生了戏剧性的变化，这主要反映在下面几个方面：

- ▶两个约束（缺少硬件支持和一个单一的输入队列）已经不复存在了。
- ▶无论 Windows 95 还是 NT 都需要使用 32 位保护模式的处理器。
- ▶Intel 处理器的环形结构允许把操作系统代码与应用程序分隔开来，同时引入一个附加的组件来提高系统整体的健壮程度。

Windows 95 体系把环形 0 用于操作系统本身，而环形 3 则专门用于进程。如图 1-8 所示。

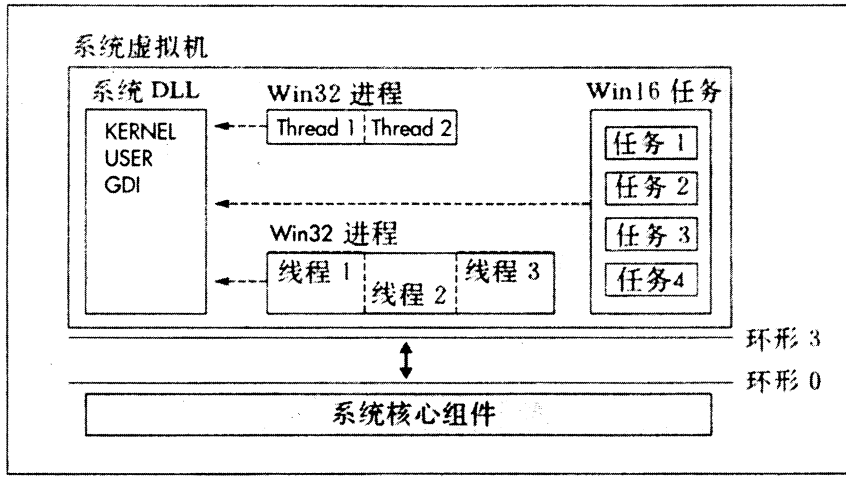


图 1-8　在 Windows 95 里，进程在系统 VM（虚拟机）的环形 3 里运行，而 DOS 应用程序则驻留于单独的 VM 内

这种方案允许开发者利用 RPL（请求符优先级），DPL（描述符优先级）以及由此而产生的进程和系统之间的地址空间分隔。图 1-8 展示了在环形 3 上运行的一些系统组件，亦即包含了 API 的一些 DLL。这种方案是由微软的系统工程师设计的，目的是为了获得优化的性能。对于 Windows 应用程序来说，它一般都需要频繁地调用窗口、GDI（图形用户界面）以及内核 API。和环内调用比较起来，需要使用环形开关的调用（环间调用），在速度上要慢许多倍。由于存在如此悬殊的性能差异，所以把 API 移到环形 3 内进行处理才是唯一可行的方案。

图 1-8 显示的 Win32 进程是由几个线程组成的。在 Win32 里，执行命令的单位和单独的进程并不一致，它采用的是比线程间隔单位更低的一种级别。相反，所有 Win16 在这方面都是完全统一的，并且采用独立的执行单位。除此以外，假如在 Win95 内运行，所有 Win16 任务都会被系统解释成一个唯一的执行线程。

接下来的这个小节向大家解释了线程的问题，对于大多数 Windows 用户来说，“线程”也许都还是一个新概念。

线程的引入

线程是系统能够调度的最少的代码数量。这难道不是一个绝妙的定义吗？现在让我们深入探索这个概念背后的含义。

本书的所有例子都是用C语言编写的。所以，我们应该假设每个程序都使用了几个通过执行的逻辑流程来调用的函数。Platform和Sysinfo是两个相当简单的例子。尽管它们都是纯粹的Win32例程，但与老式的Win16任务相比仍然存在很多共同之处。这些程序在执行的时候，都会当作一个完整的对象传送给CPU。正如我们前面提到的那样，系统调度程序将根据优先级决定运行哪个例程。

严格地说，调度程序并不会去查看进程，它选择的是线程。进程是“执行”(execution)的一个单独的线程。这样一来，系统调度程序就会把Platform与Sysinfo这两个程序当作两个独立的线程对待。一般而言，这是Win32进程最简单的形式。通过独占式的线程管理，根据各自不同的优先级别，调度程序将把Platform和Sysinfo都当作CPU的潜在宿主进行对待。

在任何给定的时刻，每个线程都要满足下列状态的其中之一：

▶正在执行

▶挂起

▶准备运行

在单处理器环境（比如Windows 95）下，同一时刻只能运行一个线程。挂起的线程将被进行分块处理，直到一个IPC对象(比如信号机)、一个事件或者一个MUTEX发出信号为止。一个准备运行的线程与正在执行的线程是类似的，只是优先级稍低一些而已。

1.7.2 多线程开发

Platform和Sysinfo并不是很复杂的程序。它们是根据单线程模式开发的两个简单的应用程序。常规情况下，假如用多线程技术开发进程，进程就会从这种技术中获得好处。起码能在性能上获得提升(能够更好地利用系统组件)，并且提供与用户之间更高的交互作用级别。这时，线程就成了关键。基本的想法是为应用程序代码赋予并行执行特性，这种特性与系统级的特性是一致的。下面这个例子将阐明这个概念。

现在让我们假设应用程序正在一个冗长的打印作业中忙碌。这时，用户和程序间的任何交互作用都是禁止的。你看来只好眼巴巴地坐在屏幕前面，等待打印完成。这就是在Win16里发生的情况。但是在Win32里，假如进程只有一个单独的线程，这种局面还是不会有太大的改变。唯一的区别在于用户现在可以切换到其他进程，这主要应归功于多重输入队列（将在下面讨论)。为了看到真正的区别，我们只有在现成代码的基础上开发它的一个多线程版本，这样才能从应用程序内部实现多任务扩展。

我们仍然不很清楚的一个问题是如何创建线程。从开发的角度来看，线程是一种特殊的函数，它的返回值是一个lresult，而且只有一个类型为lparam的参数。为了把这样一个函数转换成线程，我们可以使用CreateThread（）函数，这是一种API，我们将在第十四章“多线程、IPC和I/O”里对其进行详细的讲解。缺省情况下，一个普通的Win32进程需要一个单独的线程，即线程1，它与整个代码（WinMain（）加上窗口进程）是匹配的。

现在回过头来看看关于打印的那个例子，在与Print菜单项选择对应的消息块的主窗口进程里，我们可以调用CreateThread（）函数。这样一来，我们就启动了一个新线程，并且激活了一些InterProcess（进程间）通信途径，从而使整个进程与新建线程中的活动同步。现在，系统调度程序可以在更大的范围内选择一个线程。根据它们不同的优先级，调度程序可

以选择应用程序的主线程、打印线程或者系统中已准备好运行的其他任何一个线程。如图 1-9 所示。

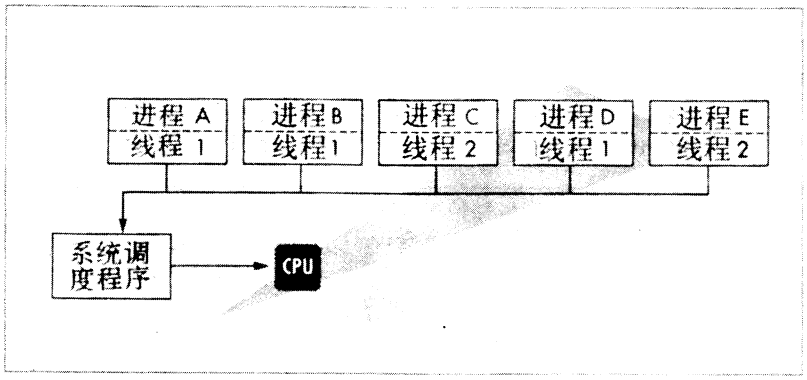


图 1-9　在一个多线程的进程里，所有线程都在竞争 CPU 时间，并且根据自己的优先级争取获得调度程序的注意

由于出现了这种全新的情况，拥有一个独立打印线程的进程允许用户启动打印输出，除非打印中断，否则不会出现系统“凝固”的情况。用户可以一边打印，一边选择一个不同的菜单项，甚至可以设置打印选项，为第二批或者第三批打印作准备。所有这一切都要归功于能够向单独的函数发出多次请求，从而得到不同的线程。

多线程技术的采用并不暗示着你的代码会运行得更快。更为准确地说，它仅仅简单意味着你可以更好地利用系统资源（调度机制、内存和物理设备），建立一个非常灵活的应用程序，使其随时都能接收用户输入，并且得到令人满意的结果。

总而言之，在 Win32 里，为了建立快速、可靠和健壮的代码，多线程开发是必须掌握的技术。第十四章“多线程、IPC 和 I/O”更详细地阐述了多线程开发，并且指出了达到特定目标的最佳技术。

1.8　异步输入模式

我们前面概略讨论了关于 Win32 抢先式多任务的问题，其中提到了 Windows 95 和 NT 里的多重输入队列。这也就是我们通常所说的“异步输入模式”。

Win32 中的总体消息流与 Win16 是完全一致的。系统掌握着一个巨大的消息队列，用它来收集后来转换成消息的所有系统事件，并且把它们派遣到应用程序消息队列中去。Win16 和 Win32 的根本区别在于把消息传递给目标队列的方式不一样。在 Win16 里，这种处理完全是在系统级连续完成的。对于普通窗口进程内的一条消息来说，对它进行处理的时候，系统根本没有机会把其他消息投递给任何一个队列。只有当执行流程返回到消息循环内部，并且在 GetMessage () 函数把控制权转交给 Windows 以后，系统才能对应用程序队列内的一条新消息进行传递。

在 Win32 里，运作方式发生了变化。一个叫作“原始输入线程”（raw input thread）的系统组件把消息连续传送给目标队列。不管系统内正在进行什么处理，也不管整体的消息流如何，这种操作都与它们完全无关。更重要的一点在于，“原始输入线程”的活动永远不会挂

起，并且它是和其他操作同时发生的。如图 1-10 所示。

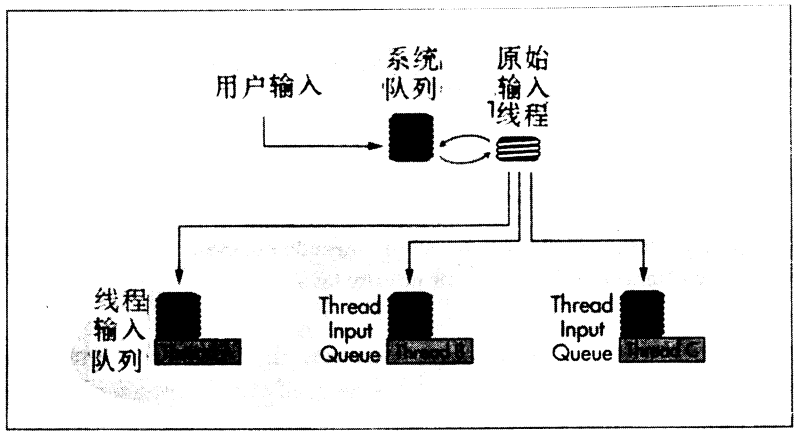


图 1-10　原始输入模式

由于采用了这种新的消息分配技术，Win32 进程变得比过去更加健壮、更具有独立性。除此以外，由于为每个进程分配了一个独立的消息队列，并且采用了多线程开发方案，所以减轻了程序设计人员在 Win16 下面固有的一些约束（不能进行快速搜索，无法对抵达一个窗口进程的每条消息进行快速执行和处理）。在 Win32 里，对于已被明确激活的一个独立的线程来说，它能进行一种特别复杂的处理，从而快速释放窗口进程，并且立即返回消息循环。

真正说来，利用 Win32 进程完全可以设计出相当灵活、相当完整的代码断。消息队列是一种特定的属性，它与一个单独的线程有关，而不是与一个进程有关。设计多线程应用程序的时候，我们甚至可以考虑使用多重输入队列——每个线程分配一个。我并不是说这是最优化的一种方案，也没有说它是每个项目的最终目标——尽管它是可行的。在某些情况下，很方便就可以命令独立的输入队列把消息流转移到两个不同的位置。在使用这种技术的时候要小心。事实上，如果两个线程都有各自独立的队列，我们建议大家尽量避免在这两个线程之间交换消息。否则，就有可能陷入非常长的 CPU 循环。第十四章“多线程、IPC 和 I/O”就这方面的问题向大家进行了详细的解释。

总而言之，有三个因素可以帮助我们开发出健壮和有效的 Win32 代码，它们分别是：多任务、多线程和多重输入队列。

1.9　内存管理的运用

对于各种系统和平台来说，内存管理都是一个关键性的方面，设计和开发任何应用程序的时候，设计者对此应该细加斟酌。Windows 95 和 NT 里的 Win32 API 已经进行了特殊的设计，它们可以充分利用 32 位平台的高级特性。和我们在 16 位环境下使用分段式内存管理技术比较起来，32 位内存管理具有很大的区别。

在本章的早些时候，我曾经提到过独立的 Win32 进程可以访问一段 4GB 的内存地址空间。这意味着开发人员可以通过后续的内存请求和分配对这一段内存区域进行寻址。这种虚拟寻址方式和物理 RAM 没有一点关系。不管对虚拟地址空间的表示在理论上如何，我们都不得不面对真正的现实。为了防止在一个进程和系统内存区之间出现危险的重叠和干扰，Win3_

对 4GB 的地址空间进行了分割，这样就为操作系统随时保留了一块固定的内存区域。

对于 4GB 内存的映射方式来说，Win95 和 NT 是各不相同的，这也反映了两个产品处理 Win16 代码时内部的区别。WinNT 采纳了 Windows on Win32（WOW）技术，而 Win95 则在它的基础 DLL 内包含了真正的 Win16 代码。WinNT 可以在它的 Win16 层上仿真 16 位代码的运行，而 Win95 则依赖于环形 3 上运行的真正的 Win16 代码。在这两种环境下，每种 Win32 都要访问低端的 2GB 内存区。如图 1-11 和 1-12 所示。

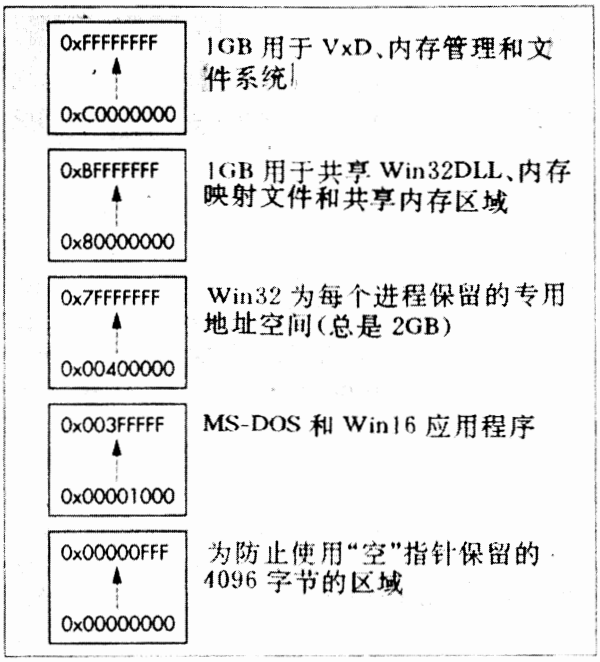


图 1-11　Windows 95 的内存映射

在 Win95 里，为 Win32 例程分配的 0－2GB 地址空间表明应用程序代码内的所有内存请求都会映射到这个范围以内。这是我准备强调的一个关键点。编写 Win32 代码的时候，我提到的都是虚拟内存的位置。操作系统的责任是把虚拟地址映射到真正的物理位置处（RAM 或者页文件内）。现在让我们一起来看看下面这个例子。

在一个财政年度的最初那段时间里，经理要在商务活动中需要对所有开支作出预算：成本、投资额以及增值费用。预算表中的那些数字不过是一些虚拟的钱。如果需要对年内收到的发票付帐，公司会计和财政官员就会把这些“钱”变成真正的钞票。例如，只有在每个月的月底才需要用到现金钞票。Win32 内存管理就是通过类似的形式来实现的。

如图 1-12 所示，在 2－3GB 的范围里，Win95 容纳了所有的共享内存块。全部 IPC 机制和系统 DLL（KERNEL32. DLL，USER32. DLL 和 GDI32. DLL）也都占据着这个内存区域。所以，所有这些 DLL 都装载于各个进程地址空间的同一个地址部分内，这就是 Win95 的特殊之处。

最后，操作系统保留了从 3GB 到 4GB 的区域。

1.9.1　分页文件的检查

对于新的内存管理规则来说，它的一个页有 4096 个字节长，所以 4096 字节就是基本的

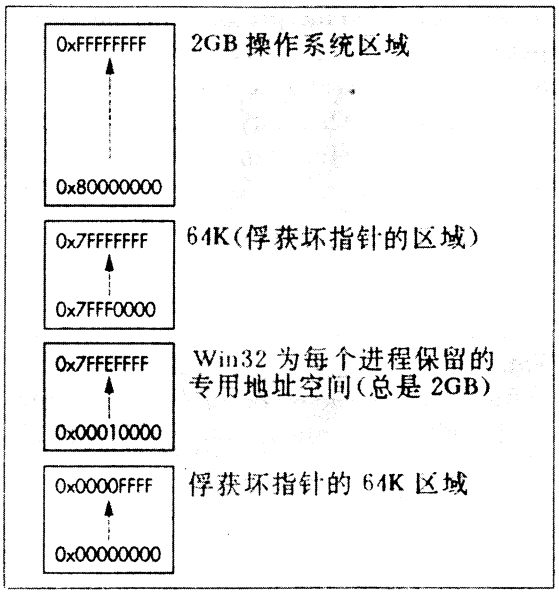


图 1-12　Windows NT 的内存映射

内存量度单位。这种规则使分页文件的使用成为可能。使用分页文件的另外一个理由是很传统的，那就是 RAM 容量和新应用程序软件的需求比较起来仍有不足（WIN386.SWP 位于\WINDOWS 目录下）。

就概念来说，分页文件与传统的 Win16 交换文件（微软仍然固执地把它叫作“虚拟内存”）是非常相似的。空闲内存所剩无几的时候，内存管理机制就忙碌起来了，它要把一部分内存拷贝到分页文件里，释放出一些系统内存，从而满足新的应用程序的需要。从性能的角度看，访问分页文件会对系统的整体性能产生影响。但是，由于这种方案同时提供了许多优点，所以不便之处很快就得到了平衡。

在 Win95 里，分页文件能够反映出内存的结构是每个页 4K。请注意，分页文件固定以 4K 为单位，而不像 Win16 那样采取长度变化的分段机制，这样就大大简化了读写操作。这种分页机制的最终结果是降低了整体的存取时间。

分页文件另外也是最有价值的一个优点在于它影响了内存管理机制。在 Win32 里，如果某个进程准备分配内存页，它只需通过标准 API（亦即我们不久要讨论的 VirtualAlloc ()）简单地发出请求即可。操作系统的任务是决定这些页的位置在系统 RAM 里还是在分页文件里。无论在哪儿，这个位置对于应用程序来说都是完全透明的。分配内存页的时候，内存管理器会在进程页表内添加一个新条目。假如新页存储于 RAM（存在位设成 1）内，剩余的 31 个二进制位就包含了那个页的全部描述信息。相反，假如这个页正好位于分页文件内，存在位就会设成 0，剩余的 31 个二进制位将全部为空。

把某个内存页重新载入新的空闲内存位置后，如果马上就访问一个空条目，这样就会导致异常事件。操作系统会拷贝页表条目内的新页帧地址，从而修改对应的存在位。后备缓冲区则会反映出这种变动，并把新的条目地址存储到自己的高速缓存里；这是操作系统肩头上的另外一个重担。到最后，导致异常事件的指令将被重新执行一遍。

在如此疯狂、忙碌的一个 Win32 环境下，一个“传统”的指针是如何构成的呢？大家也许会对这一点感到疑惑。每样东西看起来都好象处于错误的位置，或者至少不在它们应该在的地方。在 Win32 里，一个指针就是一个 4 字节的数据片，它的值并不与 4GB 地址空间内的某个物理位置对应。更为准确地说，它是通过分页机制来指定虚拟地址空间内的一个位置。

1.9.2 对地址空间的理解

也许读者曾经考虑过下面这几个问题：从 Win95 的多媒体本质来看，假如同时执行多个线程，这时会出现什么情况呢？由于每个进程都可以访问 4GB 的地址空间，系统是如何避免内存重叠的呢？应用程序会改写其他内存区域吗？万变不离其宗，只要我们理解了虚拟地址空间的机制，所有问题都能迎刃而解。

每个正在运行的进程都有可能耗尽属于它的 2GB 虚拟地址空间，同时不会影响到其他应用程序。虚拟地址空间将以小部分（页）方式映射到 RAM 里，同时把没有使用的内存信息拷贝到分页文件里。所以，即使一个进程分配了许多页，我们也不能断下结论，说系统已经耗尽了所有的内存空间。当然，此时尽管没有在进程和操作系统之间发生冲突，但是系统的整体性能已经降低了。

新的内存管理机制需要我们澄清两个基本概念。首先，内存单位固定为 4K 的页。假设你只需要 10 个字节来存储一些信息，就需要利用 VirtualAlloc（）函数向内存管理程序发出一个请求，返回的则是带有 4096 字节的一个完整的页。尽管你用不着这么多，但是通过 VirtualAlloc（）返回的只能是 4K 的一个对象，这样就浪费了一些空间。这就是内存管理的运作机制，假如你想用 VirtualAlloc（）API 函数访问和管理自己的虚拟空间，那么这种机制就是不可更改的。

需要澄清的第二点是关于页位置的问题。由于使用了分页机制，所以不可能预测到一个页位于内存的什么地方。同一对象使用的多个页可能占据系统 RAM 的非相邻位置，同时不会影响到代码的运行质量。正如我们在本章早些时候展示的那样，这种信息只与操作系统相关，而非与应用程序相关。页目录和页表的不同组合提供了成百万种地址变化。

1.9.3 预约和委托

“页”现在成了 Win32 的分配单位，但是在 32 位的世界里，这并不是内存分配的唯一特征。内存管理的优化策略涉及到一种新的词性变化，那就是动词“reserve”（预约）和“commit”（委托）的不定式“to reserve”和“to commit”。对内存块的分配是一种两阶段的处理。首先，一系列页保留于虚拟地址空间内（to reserve），然后转换成指针可以访问的物理页（to commit）。这两个步骤既有可能合并到一次单独的操作里，也有可能是分开完成的。

这种新的运作方案的优点是一目了然的。毋需任何特殊的规则和限制，一个 Win32 进程就可以向虚拟内存池发出对页的请求。在后续的执行过程中，通过一种分布式处理，这个请求会在系统内存里反映出来。另外，这种处理对整体资源的影响是微乎其微的。为了满足最苛刻的条件，需要为应用程序保留一系列线性虚拟地址，“预约”是需要采取的第一个步骤。

假设你的应用程序要求用一个缓冲区来接收特殊信息，由于这些信息只有在运行期间才能获得，所以信息的长度是无法预测的。启动应用程序后，当进程分配虚拟地址空间时，如果事先预约大量内存页，便可满足即使是条件最苛刻的情况。对物理页进行委托的时候，我们需要采用一种更为保守的途径，从而把自己预约的内存页限制在最能满足实际需求的数量。这才是节省系统 RAM 的正确方式，它能把对内存资源的影响减少到尽可能小的程度。

如果处理的页数有限，那么预约和委托步骤就有可能合并到一块儿。在这种情况下，假如还是通过个不同的调用向内存管理程序发出请求，就没有什么意义了。

VirtualAlloc（）函数调用的语法

假设你现在准备编写一段代码，用它访问至少有100个页的地址空间（100×4096字节）。内存的分配请求是通过调用VirtualAlloc（）函数发出的：

```
#include〈winbase.h〉
LPVOID WINAPI VirtualAlloc（LPVOID lpAddress，
                            DWORD dwSize，
                            DWORD flAllocationType，
                            DWORD flProtect）；
```

参数	说明
LPVOID lpAddreaa	如果分配一个新的内存块，值为NULL；如果改变页的属性，则为以前返回的内存指针
DWORD dwSize	块尺寸，通常是页尺寸的倍数
DWORD flAllocationType	分配类型
DWORD flProtect	保护类型
返回值	在正文里讨论
LPVOID	指向分配块开头的指针

第一个参数的值将随着VirtualAlloc（）当时的用法和第二个参数fdwAllocationType的不同而变化。它的值既可以设置成NULL，也可以选择上次调用VirtualAlloc（）时返回的一个指针。假如调用VirtualAlloc（）的目的是保留一个新的内存块，那么就应该把lpAddress设置成NULL；假如调用它的目的是对上次分配的页集合的属性进行修改，就应该把lpAddress设置成一个指针。

在Win32里，fdwAllocationType将采用表1-6内列出的某个值。这些标志存储于WINNT.H内，不在WINBASE.H里，这样显得更加恰当一些。

表1-6　VirtualAlloc（）分配标志

标　志	值	说　明
MEM_COMMIT	0x1000	VirtualAlloc（）分配了一个内存块，应用程序使用返回指针便可立即访问
MEM_RESERVE	0x2000	VirtualAlloc（）在虚拟分配空间里简单地预约了一系列页
MEM_TOP_DOWN	0x100000	内存分配到尽可能高的地址处，与2GB的边界接近（Win95未能实现）

前面两个标志并不是新东西。MEM_RESERVE限制VirtualAlloc（）在虚拟地址空间里预约一系列页，而MEM_COMMIT则强迫那些页具有物理性访问特性。如果使用了MEM_COMMIT，同时用VirtualAlloc（）分配一个新的内存块，那么VirtualAlloc（）的第一个参数就应该设置成NULL。如果想修改页的属性，那么肯定需要再次调用VirtualAlloc（）。

MEM_标志只能在这个函数中应用。

现在假设你需要 10 个页，其中只有五个页才需要马上委托。这时，可以连续两次调用 VirtualAlloc ()，从而预约好 10 个页；然后对前面五个页进行委托。

```
...
char *p;
...
p = (char *) VirtualAlloc (NULL, 4096 * 10, MEM_RESERVE,
PAGE_NOACCESS);
VirtualAlloc (p, 4096 * 5, MEM_COMMIT, PAGE_READWRITE);
...
```

第一次调用 VirtualAlloc () 的时候，lpAddress 参数设置成 NULL，调用这个函数的作用是返回 10 个页的内存块的起始地址。在接下来的一次调用里，我们把 p 指针明确传递给了 VirtualAlloc ()，这个指针指向了准备委托的页的起始地址。

VirtualAlloc () 的第二个参数是 cbSize，它指定预约、预约/委托或者简单的委托的字节数。最后一个参数 fdwProtect 指定分配页的保护标志。它可以是 WINNT.H 里定义的一个或者多个以 PAGE_为前缀的标志。

正如预期的那样，返回值是新分配内存块的基准地址。这个指针要求进行特殊的处理。假如第三个参数使用了 MEM_COMMIT 标志，就可以把它安全地用于访问页范围内的任何一个字节。相反，假如 VirtualAlloc () 调用里只使用了 MEM_RESERVE，那么返回指针就肯定是一个非零值，尽管我们不能用它去访问内存。这时，系统会碰到无法处理的异常事件，后果就是整个应用程序的崩溃。所以，仅仅通过检查值的大小，我们不可能知道返回指针是否有权访问已分配的内存块。

请看看下面这段代码：

```
...
char *pmem;

pmem = (char *) VirtualAlloc (NULL, 4096 * 10, MEM_RESERVE,
PAGE_NOACCESS);

*pmem = 'A';
...
```

就像我们刚才提到的那样，因为试图把'A'常数存储到第一个字节里，所以导致了一次异常事件。假如没有专门针对这段代码编写“异常处理”程序，异常事件就会导致一个错误，同时令进程中止。为了解决这个问题，我们可以有两种方案：在页的分配过程中对页进行分配，或者编写一个异常处理例程。

1.9.4 异常事件的深入理解

异常事件是一种错误，它要么在执行一条源代码语句的时候发生（软件异常事件），要么在执行一个硬件组件的时候发生（硬件异常事件）。Win32 提供了一系列功能非常强大的工具，从而尽可能地控制和解决异常事件的发生。

一个无效的参数或者“被零除”都会导致软件异常事件，而访问一个尚未委托的页则肯定要得到硬件错误。无论发生了什么异常事件，都说明源代码在运行期间碰到了错误。为了避免不正常和过早的程序中断，设计者应该在自己的代码内至始至终地采取异常处理措施，从而保证自己程序的健壮性。

发生异常事件的时候，执行流程就中止了，同时控制权被转交给系统，系统会用一种 CONTEXT（上下文相关）结构把当前的进程状态存储起来。一旦结束了最初的局面后，操作系统就会搜寻一个能处理异常事件的组件，搜寻的顺序如下所示：

- ▶首先，它检查是否有一个调试程序与发生异常事件的进程联系在一起，推算这个调试程序是否有能力处理碰到的问题。
- ▶假如不存在调试程序，或者调试程序不能处理这个异常事件，操作系统就会在发生异常事件的同一线程内寻找异常处理例程。
- ▶假如第二种尝试都失败了，操作系统就会进行第三次尝试，寻找与进程关联在一起的调试程序。
- ▶假如激活了缺省的异常处理例程，搜寻工作就结束了，这个例程一定会中断当前的进程。假如没有发现专门的异常处理例程，系统就开始执行自己的异常处理例程，同时中断进程。

作为开发者，我们应该在源代码内设置几个异常处理程序，从而避免出现第四种情况。这样能提高代码整体的健壮程度和可靠性。

1.10 异常处理程序的使用

我们前面被没有进行委托的页绊住了手脚。异常处理程序是解决这个问题的最佳方案。假如我们对此不进行干涉，程序运行的后果就是明摆着的：一次不体面的中断，紧接着是程序或者系统的完全崩溃。下面是编写异常处理程序的基本方法：

- ▶把具有潜在危险的代码部分封装起来。
- ▶编写一个异常事件过滤表达式。
- ▶根据过滤表达式的结果值，编写出异常处理程序代码。

一旦知道自己该怎样做以后，编写异常处理程序就是一件简单的工作了。它不过是一些 C/C++关键字的组合，其中集成了几个 Win32 API 而已。_try，_except 和_finally 关键字属于第一组，这已是当今异常处理领域的一种标准了。Win32 提供了下面这几个函数：GetExceptionCode（），GetExceptionInfomation（），RaiseException（），AbnormalTermination（），SetUnhandledExceptionFilter（）和 UnhandledExceptionFilter（）。通过把 C 语言的关键字与 Win32 API 组合到一起，就可以开发出健壮和可靠的 32 位代码。

1.10.1 具有潜在危险的封装代码

BASIC 已经影响了整整一代的编程人员，它教会我们开发出可靠的出错处理例程，这种例程在整个源代码里都是有效和可见的（ON ERROR GOTO）。Win32 则要求采用一种稍有

不同的方式。它不允许程序员为整个应用程序设计一个单一的异常处理例程。你必须把具有潜在危险的所有代码段封装起来，封装时采用一个_try 块结构，后面立即跟一个_except 块；也可以在后面跟一个_finally 块。下面这些例子阐明了整体的语法结构：

```
_try
{
    // try 块
}

_except (过滤器表达式)
{
    // 异常处理例程块
}
```

或者：

```
_try
{
    // try 块
}

_finally
{
    // 异常处理例程块
}
```

把_try 和_except 组合起来就构成了一个异常处理例程，而_try 和_finally 结构定义的则是中断处理例程。_try 块表达式可以认为是具有潜在危险的代码。这意味着在执行一条单独的指令或者一套有限的指令时，很容易出现异常事件。在前面的那个例子里，我们希望把'A'常数存储到一个刚刚分配的内存块里，这个内存块是用一个字符指针 * p 指定的，这个操作的危险性在于指针被重复引用了。经过对那部份代码的处理，我们现在得到了一段尽管复杂、但却有效和可靠的代码：

```
...
#define PAGESIZE 4096
#define PAGENUM 10

char * p;
```

```
pmem = (char *) VirtualAlloc (NULL, PAGENUM * PAGESIZE,
MEM_RESERVE, PAGE_NOACCESS);

_try
{
    *p = 'A';
}
_except (异常事件过滤器) {
    // 异常处理例程
}
...
```

错误仍然存在（我们试图重复引用一个指针，用它指向还没有访问权限的一个未经委托的页），这样在接下来的_except 块里就会执行异常处理例程。如果一个_try 块后面没有跟随对应的_except 或者_finally 块（这是一件让人相当头痛的事情），它就会导致出现编译错误。在上面的例子内，我们依靠一个异常处理例程来解决由于重复引用和分配操作引起的访问犯规事件。_except 关键字语法要求在异常处理块后面立即使用一个过滤器表达式。过滤器表达式在整个异常处理例程里扮演了一个非常关键的角色。很难简单地说清楚过滤器表达式里应该放置什么内容。有些时候它会调用一个用户自定义函数，或者引用一个 Win32 API，有时甚至调用一个显式定义。不管过滤器的结构如何，最终的值都必须是下面这三种定义的其中之一：

异常事件过滤器表达式	值	说明
EXCEPTION_CONTINUE_EXECUTION	-1	强迫从异常事件发生的地方继续执行
EXCEPTION_CONTINUE_SEARCH	0	指示系统搜寻异常处理例程
EXCEPTION_EXECUTE_HANDLER	1	通知系统：应用程序正在异常处理例程里对异常事件进行处理

过滤器表达式的值将指示编译器在发生了一个特定的异常事件后执行什么语句。假如异常事件是在_try 块里发生的，常规的执行流程就会绕过异常处理块。否则，由于可能有不同的选择以及过滤器表达式不同值，情况就会变得非常复杂。

在这三个值中，第一个是最有趣的。EXCEPTION_CONTINUE_EXECUTION 意味着过滤器表达式内的代码成功地进行了处理，并且解决了异常事件问题。程序将在导致异常事件的语句处继续正确地执行。这样一来，除了保护了应用程序的逻辑结构以后，还增强了程序的健壮性和可靠性。

如果过滤器表达式返回的是一个 EXCEPTION_CONTINUE_EXECUTION，就没有办法执行异常处理块内的代码。执行流程将跳回有问题的语句。假如过滤器表达式正确地修复了这条语句，流程就会正常地绕过异常处理程序。

利用 EXCEPTION_CONTINUE_SEARCH，过滤器表达式就会确认自己不能对异常事件进行处理，它将指示系统在源代码内搜寻一个新的异常处理例程（也许是当前代码块以外的一个_try—_except 块），或者在系统调试程序里搜寻。

第三个表达式是 EXCEPTION_EXECUTE_HANDLER，它的作用是强制系统执行_except 块内的语句，如图 1-13 所示。

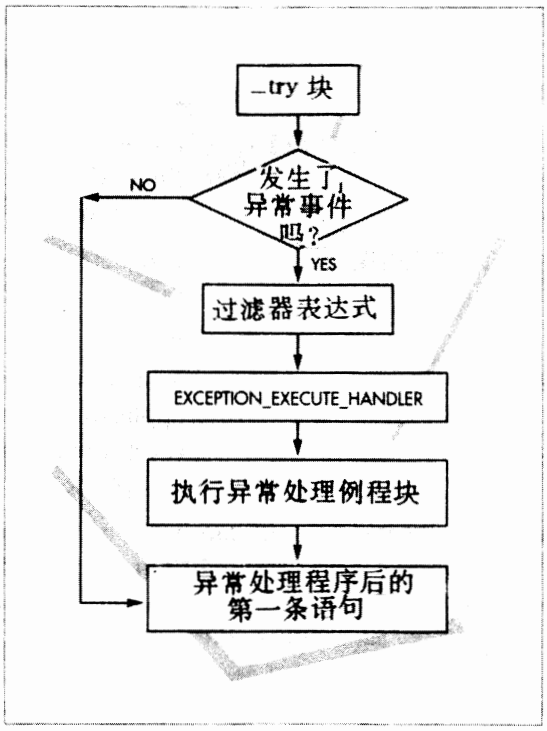


图 1-13　使用 EXCEPTION_EXECUTE_HANDLER 的一个异常事件过滤器

过滤器表达式内的代码只有在发生异常事件的时候才会执行。如果执行过程中没有碰到异常事件，所有这些代码就会被完全忽略。我在这儿要强调的一点是：三个 EXCEPTION_定义都不能放置到异常处理代码块内。对于一个 EXCEPTION_定义来说，能够容纳它的唯一两个地方就是过滤器表达式和来自过滤器表达式的一个函数调用。因此，在刚开始的时候，必须把注意力放在过滤器表达式上面，这样才能对_try 块内的每个异常事件提供正确的解决措施。出于两方面的考虑，我们认为这并不是一种没有尽头的努力。首先，它能对一个_try 块内的语句数量起到很好的限制作用。由于不断使用一条单独的指令，所以限制了异常事件的来源和发生的频率。这样就导致了一种后果——这也是第二方面的考虑——异常处理程序只能专门用于少部分异常事件。

最好的办法编写一个新的例程，在过滤器表达式内对它进行调用，这种表达式用于专门处理一些数量有限的异常事件。为了深入了解发生的是何种异步事件，我们要用到 GetExceptionCode（）这个 API 函数。

```
#include〈except.h〉
```

DWORD GetExceptioncode（void）

GetExceptionCode（）只能在过滤器表达式或者异常处理代码块内出现，不能在由过滤器表达式调用的例程里使用（如果在有效位置以外发现了这个函数，编译器就会报告出错）。通过该函数的返回值，我们就能知道发生的是什么异常事件。表 1-7 列出了各种异常事件的返回值。

表 1-7　由 WINBASE. H 定义的 Win32 异常事件清单

异常事件	值
EXCEPTION_ACCESS_VIOLATION	STATUS_ACCESS_VIOLATION
EXCEPTION_DATATYPE_MISALIGNMENT	STATUS_DATATYPE_MISALIGNMENT
EXCEPTION_BREAKPOINT	STATUS_BREAKPOINT
EXCEPTION_SINGLE_STEP	STATUS_SINGLE_STEP
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	STATUS_ARRAY_BOUNDS_EXCEEDED
EXCEPTION_FLT_DENORMAL_OPERAND	STATUS_FLOAT_DENORMAL_OPERAND
EXCEPTION_FLT_DIVIDE_BY_ZERO	STATUS_FLOAT_DIVIDE_BY_ZERO
EXCEPTION_FLT_INEXACT_RESULT	STATUS_FLOAT_INEXACT_RESULT
EXCEPTION_FLT_INVALID_OPERATION	STATUS_FLOAT_INVALID_OPERATION
EXCEPTION_FLT_OVERFLOW	STATUS_FLOAT_OVERFLOW
EXCEPTION_FLT_STACK_CHECK	STATUS_FLOAT_STACK_CHECK
EXCEPTION_FLT_UNDERFLOW	STATUS_FLOAT_UNDERFLOW
EXCEPTION_INT_DIVIDE_BY_ZERO	STATUS_INTEGER_DIVIDE_BY_ZERO
EXCEPTION_INT_OVERFLOW	STATUS_INTEGER_OVERFLOW
EXCEPTION_PRIV_INSTRUCTION	STATUS_PRIVILEGED_INSTRUCTION
EXCEPTION_IN_PAGE_ERROR	STATUS_IN_PAGE_ERROR
EXCEPTION_ILLEGAL_INSTRUCTION	STATUS_ILLEGAL_INSTRUCTION
EXCEPTION_NONCONTINUABLE_EXCEPTION	STATUS_NONCONTINUABLE_ECEPTION
EXCEPTION_STACK_OVERFLOW	STATUS_STACK_OVERFLOW
EXCEPTION_INVALID_DISPOSITION	STATUS_INVALID_DISPOSITION
EXCEPTION_GUARD_PAGE	STATUS_GUARD_PAGE_VIOLATON

现在用_except 块来完成前面的代码段：

```
...
char *p;

...
p = (char *) VirtualAlloc (NULL, 10 * 4096, MEM_RESERVE,
PAGE_NOACCESS);

_try
{
    *p = 'A';
```

```
}
_except ( (GetExceptionCode () ==
EXCEPTION_ACCESS_VIOLATION) ? xxx : yyy)
{
    // 异常处理例程块
}
...
```

异常事件过滤器内的运算符是不完整的。除此以外，并不代表只能用这一种方式对异常处理例程进行编码。这个例子简单地测试了异常事件是不是与一个访问犯规事件对应，并在重复引用 p 指针指向一个尚未委托的页时，应该采取什么样的对策。如果想解决异常事件带来的问题，那么只需调用 VirtualAlloc () 函数对那个尚未委托的页进行委托便已足够了。这条简单的指令可以直接放置于条件运算符的过滤器表达式里，尽管这并非一种实际的方案。更方便的一种方式是建立一个独立的函数：

```
...
char *p;

p =(char *) VirtualAlloc (NULL, 4096 * 10, MEM_RESERVE, PAGE_
    NOACCESS);

_try
{
    *p = 'A';
}
_except (PageExceptionFilter (GetExceptionCode () . p))
{
    MessageBox (hwnd, " Exception not handled", NULL, MB_OK);
    ExitProcess (0);
}
...

int PageExceptionFilter (int iExceptionCode, char *p)
{
    if (iExceptionCode == EXCEPTION_ACCESS_VIOLATION)
    {
        VirtualAlloc (p, 4096 * 10, MEM_COMMIT, PAGE_READWRITE);
        return EXCEPTION_CONTINUE_EXECUTION;
    }
```

```
        return EXCEPTION_EXECUTE_HANDLER;
    }
    ...
```

其中，PageExceptionFilter () 函数的作用是接收特殊的数字代码，用这个代码来识别异常事件；同时接收 p 指针，正是这个指针导致了异常事件的发生。如果由于访问一个尚未委托的页，从而发生了违规访问这种异常事件，就可以调用带有 MEM_COMMIT 标志的 VirtualAlloc () 函数，并且把保护标志设置成 PAGE_READWRITE。返回 (return) 语句的作用是指示异常处理程序从那条导致异常事件的语句处恢复执行。假如异常事件与 EXCEPTION_ACCESS_VIOLATION 不符，控制权就会转交给异步处理代码。这时会在屏幕上显示一个消息框，同时中断进程。

通过对 PageExceptionFilter () 函数的详细探讨，我们有机会对 VirtualAlloc () 函数的语法进行更加深入的理解。在上面那种情况下，第一条参数是指向未委托页的 p 指针。在调用 VirtualAlloc () 之前和之后，p 指针发生了什么样的变化呢？这个指针基本没有发生任何变化，至少从数值的角度来看是这样的。该指针的值根本没有发生任何变动。这就证明了无论目标是一个已委托的还是尚未委托的内存页，指向这个位置的指针都拥有固定的值。

为了扩展读者在异常处理机制方面的知识，并且同时为了加深自己对内存页的熟悉程度，我们现在对 PageExceptionFilter () 进行一番小小的变化。在调用 VirtualAlloc () 之前，最好先调用 VirtualQuery () 函数，用它检查导致异常事件的那个页的基本特征。VirtualQuery () 是一个相当简单的 API，然而它的功能却是相当强大的：

```
DWORD WINAPI VirtualQuery (LPCVOID lpvAddress,
                           PMEMORY_BASIC_INFORMATION pmbi,
                           DWORD cbLength);
```

参数	**说明**
LPCVOID lpvAddress	准备检查的那个页的起始地址
PMEMORY_BASIC_INFORMATION pmbi	MEMORY_BASIC_INFORMATION 结构的一种地址，它可以接收关于页的所有信息
DWORD cbLength	MEMORY_BASIC_INFORMATION 数据结构的长度
返回值	在正文里讨论
DWORD	实际读取的字节数

VirtualQuery () 函数项是采用 MEMORY_BASIC_INFORMATION 结构填写的，其中包含了与一个或者多个相邻页有关的信息，这些页都具有相同的特征。所以，第一个参数指向的是开始进行检查的地址。随后跟随的是一个采用 MEMORY_BASIC_INFORMATION 数据结构的地址，以及它的长度。MEMORY_BASIC_INFORMATION 内包含了一系列非常

有用的信息，利用这些信息就可以监测内存页的状态：

```
typedef struct _MEMORY_BASIC_INFORMATION
{
    PVOID BaseAddress;
    PVOID AllocationBase;
    DWORD AllocationProtect;
    DWORD RegionSize;
    DWORD State;
    DWORD Protect;
    DWORD Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION ;
```

通过 VirtualAlloc ()，开发者可以在返回指针的帮助下访问一系列内存页。这种信息与 MEMORY_BASIC_INFORMATION 结构内的 BaseAddress 项是对应的。除此以外，每个页都有它们自己的起始地址，这是存储在 AllocationBase 项里的一个数值(只有对内存块内的第一个页来说，它在 BaseAddress 内的地址才与 AllocationBase 一致)。因为用 PAGE_前缀可以得到一个或者更多的标志，所以 VirtualAlloc () 可以采用不同的语法结构。然而，尽管表 1-8 列出了一大串清单，但是到目前为止，我们经常用到的还只是 PAGE_NOACCESS 和 PAGE_READWRITE。

表 1-8　Win32 的页保护标志；在 Win95 里许多都是限制使用的

保护标志	值	说　明	Win95
PAGE_NOACCESS	0x01	屏蔽对页的访问	Yes
PAGE_READONLY	0x02	只读页	Yes
PAGE_READWRITE	0x04	可读、可写页	Yes
PAGE_WRITECOPY	0x08	一旦指令强制读某页，就拷贝那个页	No
PAGE_EXECUTE	0x10		No
PAGE_EXECUTE_READ	0x20		No
PAGE_EXECUTE_READWRITE	0x40		No
PAGE_EXECUTE_WRITECOPY	0x80		No
PAGE_GUARD	0x100	保护页 (Win95 未使用)	No
PAGE_NOCACHE	0x200	屏蔽页缓存	No

在 Windows 95 里，只有少数标志可以正确地实现，这也是 Win95 和 NT 间的另外一个区别。最初对内存页块进行预约的时候，就会设置使用了 PAGE_标志的 AllocationProtect，并且只要那个内存块一直存在，该设置就永远不会发生变动。Protect 项是非常相似的，但是它们又存在着一个重要的区别。在这种情况下，它随时都包含了针对当前页的保护标志的动态混合物。为了动态地修改一个页保护标志，我们要用到 VirtualProtect () 函数。

```
#include <winbase.h>
BOOL VirtualProtect (LPVOID lpvAddress,
                     DWORD cbSize,
                     DWORD fdwNewProtect,
                     PDWORD pfdwOldProtect);
```

参数	说明
LPVOID lpvAddress	内存块的起始地址，与保护标志的变动相符
DWORD cbSize	内存块的尺寸
DWORD fdwNewProtect	新的保护标志
PDWORD pfdwOldProtect	一个双字的地址，其中包含了老的保护方案
返回值	在正文里讨论
BOOL	布尔值，TRUE 代表函数调用成功，FALSE 代表失败

请注意，如果想修改页保护标志，就必须委托一个内存页。假如没有使用 MEM_COMIT 属性，对 VirtualProtoct () 的任何调用都会失败。VirtualAlloc () 允许程序设计者为一个预约或者委托的页指定最初的保护标志，而 VirtualProtoct () 的任务则是在建立好页后对其进行修改。根据作为第一个参数传递的地址以及块的尺寸（第二个参数），这个函数可以有选择地对一个页或者多个页进行处理。可选的 PAGE_标志占用了第三个参数，该参数紧接在双字地址标识符（Identifier）的后面，那个标识符的作用是接收上一个保护标志。

在 VirtualProtoct () 上面进行了一番短暂的停留以后，现在让我们返回 MEMORY_BASIC_INFORMATION 数据结构。RegionSize 的作用是让许多连续的页具有相同的属性（预约、委托或者空闲等），这个值是用字节表示的。RegionSize 内的值很容易就会超出一个内存块的原始尺寸。这并非一个设计上的失误，而是在特殊情况下的一种正确的值。假设你现在准备预约 10 个页，紧跟在最后一个页后面的页具有相同的状态。对 VirtualQuery () 的一次调用会把最初的指针传送给内存块。这样一来，RegionSize 返回的数字就会大于最初的块尺寸，这是最后一页后面跟随了额外的页的缘故。

页的状态是在 State 项里报告的。这种标志可选的值包括 MEM_RESERVE，MEM_COMMIT 和 MEM_FREE 等。前面两个标志是我们已很熟悉的，但是第三个在 Win32 API 里却从来没有出现过。它的存在只是表明在随后的内存分配过程中可以使用一个空闲的页。在一个标准的应用程序里，并非经常需要取回 MEM_FREE 标志。它的存在意味着我们可以指向一个特殊的内存区域，这个区域完全是由传递给 VirtualQuery () 的指针控制的。在 Except 这个例子里（可在本书附带 CD 的 Listing 1.3 内找到），我们有时会获得一个 MEM_FREE 值，因为设计这个应用程序的目的是为深入内存的奥秘，试验各种可能的情况。假如 MEMORY_ BASIC_ INFORMATION 的 State 项使用了 MEM_ FREE 标志，那么 AllocationBase，AllocationProtect，Protect 和 Type 项就根本没有什么意义了。表 1-9 为大家总结所有的 MEM_标志和相关的 Win32 API。

通常，Type 项使用的都是 MEM_PRIVATE 值，它的作用是指出那个内存区域是某个进程专用的。

表 1-9 MEM_标志和相关的 Win32 API

标 志	值	API
MEM_COMMIT	0x1000	VirtualAlloc ()
MEM_RESERVE	0x2000	VirtualAlloc ()
MEM_DECOMMIT	0x4000	VirtualFree ()
MEM_RELEASE	0x8000	VirtualFree ()
MEM_FREE	0x10000	MEMORY_BASIC_INFORMATION
MEM_PRIVATE	0x20000	MEMORY_BASIC_INFORMATION Type 项
MEM_MAPPED	0x40000	MEMORY_BASIC_INFORMATION
MEM_TOP_DOWN	0x100000	VirtualAlloc ()（MS Windows 95 未采用）
MEM_IMAGE	SEC_IMAGE	MEMORY_BASIC_INFORMATION

1.10.2 关于 PAGE_GUARD

一个有经验的开发都会避免使用错误的指针去访问由其他进程预约的内存区域。考虑到分段机制在 Win32 里已经不复存在，所以没有自然的界限来划定一个进程地址空间的边界。以前那种讨厌的分段违规消息不再会出现了，但是仍然有必要对进程地址空间的边界问题引起足够的重视。处理这类问题的一个准则是把具有潜在危险的所有指令封装到一个_try 块内，同时在后面使用一段强有力的异常处理例程。

访问犯规是一种非常常见的异常事件。为了深入理解如何避免访问犯规事件，我们可以用下面这个例子来说明。假设你在预约了 10 个内存页后，只对其中的四个进行了委托。假如你把指针移至第五个页的第一个字节上面，这时就会发生访问犯规异常事件。通过异常处理程序赋予代码的健壮性，这个问题很快就能得到解决。这是一个很不错的措施，但是假如指针正好指向第 10 个页最后一个字节的后面，这时会出现什么情况呢？答案是显然的：这时会发生访问犯规异常事件。这样就出现了一个问题，对于合法和可恢复的访问犯规（引用了已预约范围内的一个页），以及真正的编程错误来说，怎样对这两者作出区分呢？为了回答这个问题，我们首先需要掌握关于页保护技术的知识。

正如表 1-8 列出的那样，PAGE_GUARD 是一个页保护标志，它的作用当指针接触到带有这种属性集的一个页时，便产生一个 EXCEPTION_GUARD_PAGE 异常事件。只有第一次访问某个保护页的时候，系统才会触发这个异常事件，这和公告消息的运作形式非常相似。所以，开发者最好把 PAGE_GUARD 保护标志增添到最后一个委托页上（这个属性只能运用于委托页），而有选择地把它增添到下一个委托页上。

不幸的是，Windows 95 没有提供对保护页机制的支持，所以我们不得不采取其他策略。最简单的方法是从内存块的起始处不断检查实际的指针位置。举个例子来说，假设在已预约的 10 个页里，只有前面 8 个是已委托的。这时如果访问第 9 个页，系统将产生一个异常事件。这时就会调用 VirtualQuery () 函数，用它传递整个内存块的地址。RegionSize 项的值等于 4096×8（最前面 8 个已委托的页），同时指针指向这个值与内存块总长度之间的某个位置。这样一来，就找到了预约内存页内的一个尚未委托的页。随后便可对其进行委托，然后继续代码的执行。这种方案并不十分简洁，但是能工作得很好。

作为另一种选择，可以用 PAGE_NOACCESS 标志去指定最后一个被委托的页。对这个页的任何访问都会产生一个访问犯规异常事件。我们必须用一个 VirtualQuery () 函数对这

种事件进行测试，检查其中是否含有 PAGE_NOACCESS 标志。Except 例子可以让大家把这个保护标志分配给一个已委托的页。接下来，当用户试图通过单击鼠标右键访问这个页的时候，屏幕中就会显示出一条警告消息，说明这个页是不能被访问的。

注意：诸如_try—_finally 的中断处理程序都是用来对付违规事件的。它们的目的都是确保一段特定的代码能够正常地执行下去，而不管在_try 块内发生了什么。_finally 块内包含的代码用于释放系统资源、关闭文件句柄、释放共享内存区以及其他操作，这些操作都是在一种算法成功完成或者失败以后必须进行的。

1.10.3 内存的释放

在进程中断前的清除阶段或者不再需要一个内存块的时候，这个内存块就会被释放出来。内存块的释放通常是通过调用 VirtualFree（）这个 API 函数来实现的。

```
#include〈winbase.h〉
BOOL WINAPI VirtualFree (LPVOID lpvAddress,
                         DWORD dwSize,
                         DWORD fdwFreeType);
```

参数	说明
LPVOID lpAddress	准备释放的内存块的起始地址
DWORD dwSize	块的范围
DWORD dwFreeType	要求采取的行动
返回值	在正文里讨论
BOOL	布尔值，TRUE 代表函数调用成功，FALSE 代表失败

第一个参数定义了内存块的起始地址，紧接着是它的长度（一个或者多个页）。假如在第三个参数里出现了 MEM_RELEASE，长度值应该设置成零。VirtualFree（）可以采取两种行动。它既可以撤消对一个内存块的委托，也可以把内存释放出来。

标志	值	说明
MEM_DECOMMIT	0x4000	撤消一系列页
MEM_RELEASE	0x8000	释放一系列页，把内存释放出来

撤消对一个或者多个内存页的委托以后，这些页就会恢复成最初预约它们时的状态（MEM_RESERVE）。内存页仍然存在于虚拟地址空间内，以后仍然可以通过对 VirtualAlloc（）的一次新的调用进行委托。设置了 MEM_RELEASE 以后，物理页就被保留下来了，同时虚拟地址空间的可用内存也增多了。从这时开始，无论在虚拟还是物理地址空间内，指针都不再有效。

总而言之，VirtualFree（）是实现下述操作的理想方式：

▶撤消以前对一个或者多个页进行的委托（MEM_DECOMMIT），把它们恢复成 MEM_RELEASE 状态。

▶释放以前预约的一个或者多个页（MEM_RELEASE），把它们恢复成 MEM_FREE 状态。

对于以前没有委托（也就是说处于预约状态）的一个页来说，如果解除对它的委托，那么什么事情都不会发生，那个页仍然保持自己的 MEM_RESERVE 属性。如果想释放内存，就必须用 MEM_RELEASE 属性来调用 VirtualFree（）函数，指定一个指向一系列页的地址，所有这些页都必须处于相同的状态。用 MEM_RELEASE 属性对 VirtualFree（）进行一次单独的调用是远远不够的。这是由于所有页都必须处于同样的状态（全被委托或者全被预约），否则调用便会无效。但是，假如内存页不处于这两种状态中的任何一种，就应该调用 VirtualFree（）两次。第一次调用是用 MEM_DECOMMIT 来实现的，它把所有页都带回 MEM_RESERVE 状态。第二次调用就要用 MEM_RELEASE 来释放内存。下面是具体的代码示范：

```
...
// all the pages to MEM_RESERVE no matter their actual attributes
VirtualFree (p, PAGESIZE * PAGENUM, MEM_DECOMMIT);
// freeing up memory
VirtualFree (p, 0, MEM_RELEASE)
...
```

对 VirtualFree（）进行第二次调用时，dwSize 参数等于零。只有在撤消对页的委托时才有必要对实际的块尺寸进行设置。到释放所有内存页的时机成熟的时候，就应该在代码内提供基准分配地址、一个零和 MEM_RELEASE。VirtualFree（）不允许物理性删除最初分配的内存页的一个子集，否则就会返回一个 FALSE，报告出错。在另外一方面，撤消对内存页的委托时，如果一个指针所指的位置与一个内存块的起始处地址不同，那么用这个指针来调用 VirtualFree（）将是十分安全的。

VirtualFree（）函数的运作方式不应该让你感到吃惊。它的用途是双重的：释放物理内存（MEM_DECOMMIT），以及释放虚拟内存（MEM_RELEASE）。在第一种情况下，内存的单位是页——更准确地说，是位于前一个分配块内的任何一页。在第二种情况下，块本身就成了单位，它是是一系列虚拟连续页的集合。

通过用 VirtualFree（）撤消对页的委托，开发人员可以减少 RAM 或者分页文件内锁定内存的数量，缩减整体的内存占用比例。一个正在工作的进程应该在执行期间设定自己需要的内存容量。

1.11 灵活运用内存

现在到了对本章学到的所有新内存管理函数进行检阅的时候了。在本书附带 CD 盘的 Listing 1.3 里，读者可以找到一个名为 Except 的例子。该程序在启动的时候预约了 10 个页。在应用程序的客户区里，我们可以看见 10 个红色的矩形。这些矩形就是对这些内存页的图形化表示。如图 1-14 所示。

“已预约”状态使用的颜色是红色——表示不能接近和危险的常用色。一旦被委托，页就

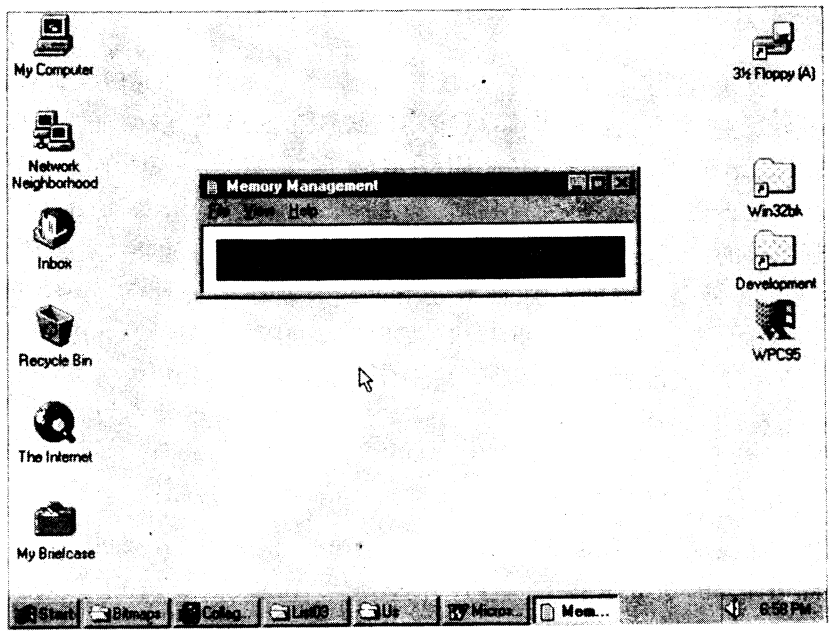


图 1-14　Except 程序启动后的样子。客户区对 10 个预约的页进行了形象的表达

变成了绿色。Except 这个例子里可以采取两种方式对内存页进行委托。如果在一个页上面单击鼠标左键，就可以强制委托这个页，并在其中填充 4096 个字母字符（从 A 开始）。事实上，在与 WM_LB UTTONDOWN 消息对应的消息块内，你应该尝试为选择的页填充对应的字母，并且只有当那个页被预约的时候才发生一次异常事件。通过对页进行委托，然后在其中填充字母，一个恰当的异常处理程序就能解决这次异常事件。

第二种方式要求用户单击鼠标右键。这时会在鼠标的光标下方显示出一个弹出式菜单，其中列出了几个菜单项。从上往下数第二个菜单项是 Status，它指出了当前页的状态（最初是被预约的），并在第二个弹出式菜单内列出了所有可行的操作（如图 1-15 所示）。

正如我们早先解释过的那样，只有第一个页才会提供 Free 选项。选择了这个菜单项以后，对所有页的委托都会被解除，然后释放出这些页。这种新的状态是用 10 个蓝色的矩形来表示的。总而言之，红色表示已预约的页，绿色表示被委托的页，而蓝色则表示是一个空闲的内存块。

Except 这个例子最显著的一个特征是它的光标。这儿使用的不是传统的白色箭头，而是一个红色的三维箭头。我知道这并不是一项重大的变动，但是我宁愿用一个彩色的三维箭头，也不愿用传统的白色箭头。Win32 允许开发者使用彩色光标，我想在本书的第 1 章就向大家展示这种令人耳目一新的新特性，我不想错过这种机会！在第三章“Win32 应用程序的开发”里，我将详细介绍如何利用崭新的、功能强大的 LoadImage () API 函数来载入一个彩色光标。我们现在忽略这个细节，把注意力集中到更有内含的一些东西上面。

用鼠标右键单击白色客户区内的任何一个地方，这时有什么情况出现呢？一个小的弹出式菜单会在鼠标热点的下方显示出来。选择其中的 View All 菜单项，主窗口就会在垂直方向放大，显示出一个列表视窗（listview）——一种新的 Win95 通用控件。View All 菜单项也可

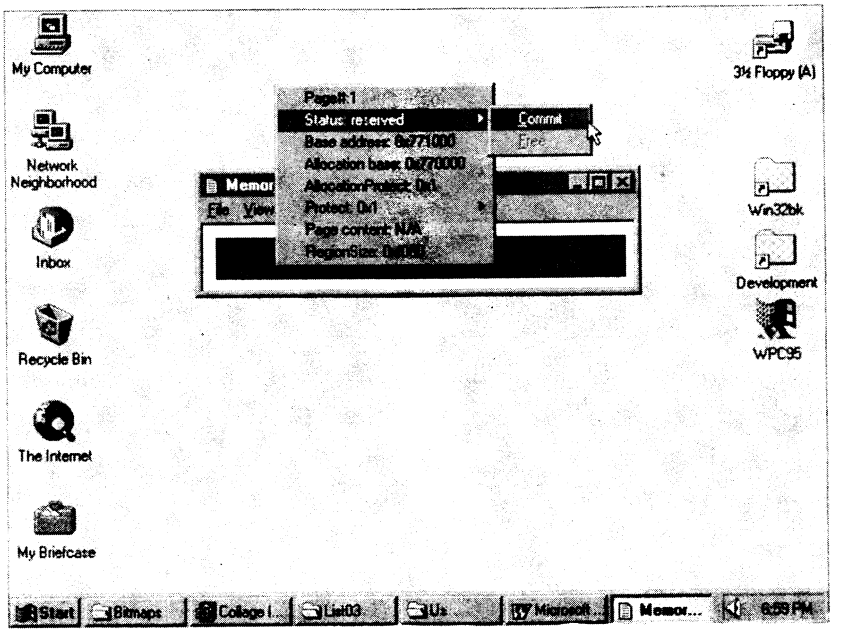


图 1-15　在第二个弹出式菜单内选择恰当的菜单项，便可对一个页进行委托

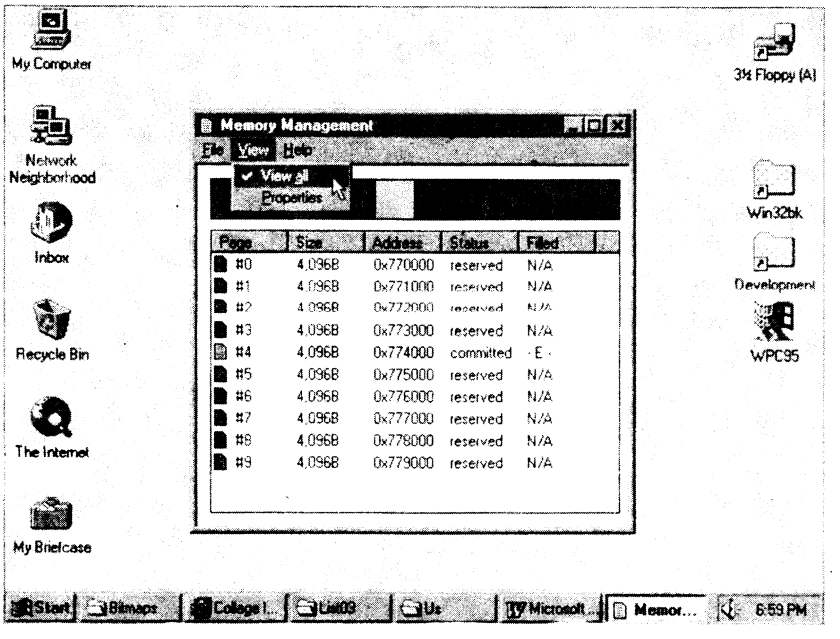


图 1-16　Except 内的 View All 菜单项显示出一个列表视窗，其中列出了所有这 10 个页，每个页都用左侧的一个小图标表示，右侧则是关于这些页的特定信息

以在 View 菜单内选择，如图 1-16 所示。

正如我们早先看到的那样，在一个页（甚至是列表视窗内的一个页）的上方单击鼠标右

键，就会出现一个不同的弹出式菜单。这并非一个传统的、用于向用户提供某些选项的菜单。取而代之的是，它提供了关于页号、基准地址、分配基准、初始和当前保护标志等等方面的信息（如图 1-17 所示）。

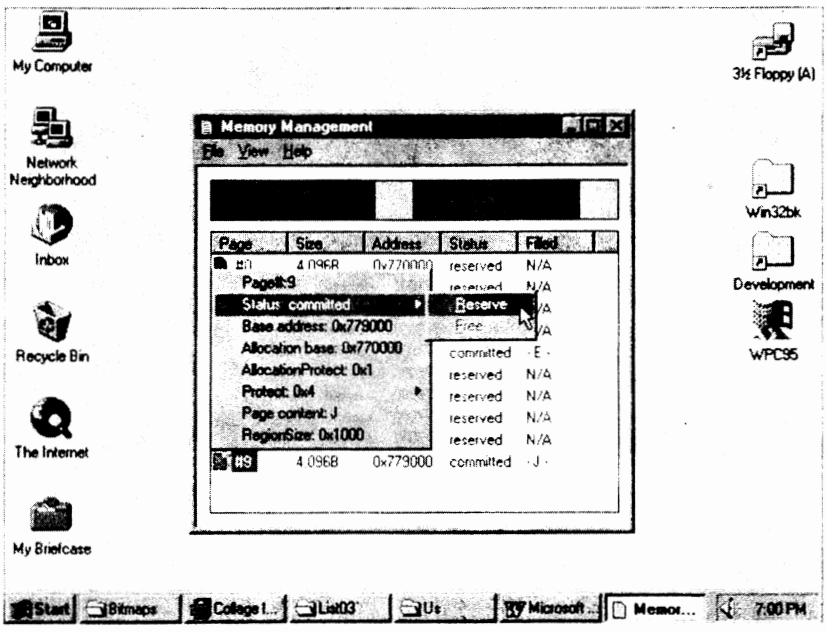


图 1-17　与列表视窗内某个页有关的弹出式菜单

弹出式菜单内的信息几乎涵括了 MEMORY_BASIC_INFORMATION 结构内的所有项，同时集成了另外两种信息：每个页的标识编号（从零到九）以及用于填充页（在该页被委托以后）的一个示范字符。弹出式菜单的内容将进行连续性的更新，从而反映出被选页的变动。

这 10 个页最初都是用 PAGE_NOACCESS 属性进行预约的，对页进行委托以后，对应的属性就变成了 PAGE_READONLY 或者 PAGE_READWRITE。由于允许用户对保护属性进行任意修改，所以异常处理算法就变得稍微有些复杂。

例如，假设一个页已被委托了，同时它的保护标志已被设置成 PAGE_READONLY。这时如果在这个页上面单击鼠标左键，试图在其中填充 4096 个字符，那么就会产生一个 EXCEPT_ACCESS_VIOLATEIN（违规访问）异常事件。然而，这种异常事件是以一种新的保护方案为基础的，而不是以被委托的页为基础。我们可以选择两种方法来解决这个问题。第一种是消除 PAGE_READONLY 属性，用一个 PAGE_READWRITE 属性来替换，然后继续程序的执行。根据这种方法的原理，我们将不得不修改用户做出的一次明确的选择。另一种方法是继续执行应用程序，向用户报告碰到了一个异常事件。在异常处理程序里，你必须检测是否由于保护违规而导致了异常事件，然后继续返回一个 EXCEPTION_EXECUTE_HANDLER 属性。

```
...
_try {
        FillMemory (pmem + iOffset, PAGESIZE, (BYTE) ('A' + iPage));
}
_except (MyExceptionHandler (pmem + iOffset, GetExceptionCode ())) {
         MessageBox (hwnd, " PAGE_READONLY is on. \nOperation not allowed at
         this time.",
" Exception", MB_OK | MB_ICONERROR);
        return FALSE;
}
...
```

图 1-18 展示了发生保护违规异常事件时显示出来的消息框，其中带有对应的公告消息。

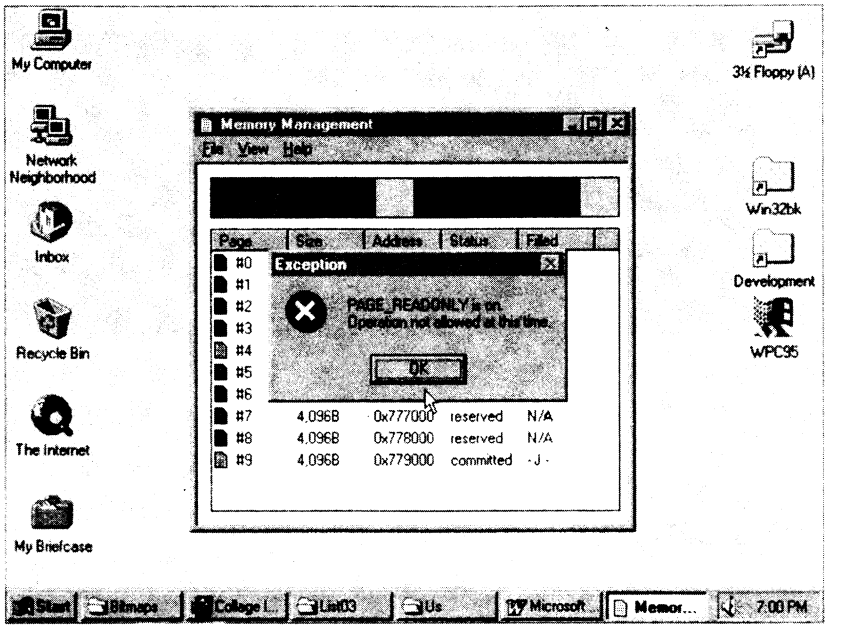


图 1-18　假如设置了 PAGE_READONLY 标志，我们就不能再向那个页写入信息了，否则就会导致一个异常事件

第 2 章　Win32 开发工具

在第 1 章“Win32 中的软件开发”里，大家已有机会接触了 Win32 API 一些具有创新性的概念和开发模式。这一章将向大家介绍用于编写本书示范程序的开发工具，另外还要介绍设置一个强有力的开发环境所需的硬件和软件组件。现在让我们首先讨论关于硬件配置的问题。

2.1　硬件组件

显然，为了获得一个理想的 Windows 95 开发工作站，必须在硬件设备上投入一定的资金。我在这儿将向诸位介绍自己的硬件配置、我遇到的问题以及我是如何开发本书这 100 多个示范程序的。

我目前使用的开发系统包括下面这些组件：

▶AST Bravo MS P90 个人计算机，配 32MB RAM。

▶700 多兆硬盘空间。

▶17 英寸显示器，运行于 Windows 95 提供的 1024×768 分辨率下。

▶Sound Blaster 16 位声卡。

▶二倍速 CD-ROM 驱动器。

▶调制解调器。

▶3Com Etherlink III 网卡。

为何需要 32MB 的 RAM？答案很简单：因为我无法安装更多的内存了！在自己的开发工作站里，你一定要安装大量的内存才行，这至少有两方面的理由。首先，它可以把分页文件的活动减低到最低程度，从而节省大量宝贵的时间。其次，Win95 里的 Win32 内存管理机制与 NT 稍有不同。由于使用了大量内存，我可以及时监视系统活动，并能更好地理解程序内部发生的一切。

硬盘至少要有 500MB，除了我的 700 多兆硬盘以外，我还必须在自己的服务器上保留一部分空间，用它来存储所有的图像、屏幕位图以及本书 CD-ROM 附带的其他内容。预测 Win95 所需的磁盘空间是一件费力的工作，因为它的安装是模块化的。平均地说，假如安装了许多可选的特性和组件，就会占用大约 70 多兆的硬盘空间，同时还不要忘了页文件。尽管我有 32MB 内存，页文件在某些特定的情况下还是会变得相当大，有时甚至会超过 20MB。它的增大或者缩小是动态进行的，但是我们至少都应该考虑到 10－20MB 的空间。

然后就是开发环境的问题，开发环境涉及到软件开发包（或者 SDK，它们现在是两个独立的对象）、实用程序、工具软件以及帮助文件。这些东西要占据大约 200MB 的硬盘空间。所以，操作系统加上开发工具最后将占用超过 300MB 的宝贵硬盘空间。我对 500MB 硬盘空间的估计已经是相当保守了。

声音卡几乎是必备的硬件。在第 12 章“非标准的输入和输出”里，我提到了在自己的系统上已经测试过的许多多媒体示例。除此以外，如果没有声音卡，你甚至连系统引导成功后

发出的那一段悦耳的声音都听不到!

为了与世界上的其他地方联系、为了进入 Microsoft Network 以及为了把我写的章节传递给我的出版商,我必须依靠自己的 28.8 Kbps 调制解调器。Modem 是一种关键的开发工具,因为通过它能访问许多电子论坛,那儿有成千上万的人可以向你提供技术信息、示例以及编程技巧。

2.2 软件组件

我是在 1993 的年末开始编写 Win95 代码的,当时我才参加了“微软专业开发者联合会议”不久,这次会议是在加拿大的 Anaheim 召开的。随后,我采纳 Visual C++作为自己的开发平台,当时用的版本是 1.0。后来,我把它升级到了 2.0 版,最近才开始使用 Visual C++ 2.1 版本。Visual C++附带提供了一系列库、工具、头文件以及开发示例,这些东西针对的主要是 WinNT,而不是 Win95。微软公司的目标是把 WinNT 和 Win95 使用的 SDK 合并到一个单独的产品里,从而简化通用 32 位应用程序产品的开发,使其能同时在这两个平台上运行。

用于 Win95 和 NT 的 SDK 会建立一个名为\MSTOOLS 的目录,其中存放了所有执行程序、库、DLL 以及其他开发组件。图 2-1 展示了最新的 SDK 版本在一个 Win95 系统下的安装情况。

图 2-1 Tools 文件夹内包含了建立 Win32 应用程序所需的软件组件。
这些应用程序是用一种高级语言编译器建立的,比如 Visual C++(显示于后台)

\MSTOOLS 内包含了 100 多个文件,总共占据了 100 多兆的磁盘空间。在安装过程要结束的时候,SDK Setup 模块在 Start 弹出式菜单内增加了一些条目,即 Win32 SDK Online Reference(联机参考)和 Win32 SDK Tools(如图 2-2 和 2-3 所示)。

SDK 提供的最好和最有用的实用程序也许要算 PVIEW95 了，这个程序能列出当前正在运行的所有进程，并在客户区内列出了相关的线程，如图 2-4 所示。

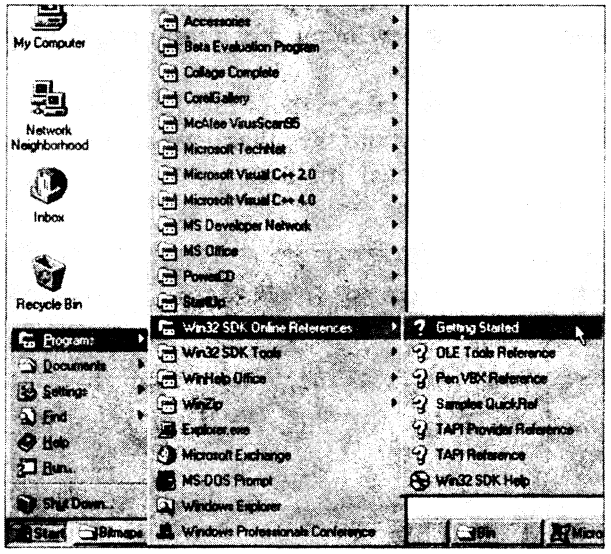


图 2-2　Win32 SDK Online Reference 菜单列出了由 SDK 提供的所有文档

图 2-3　Win32 SDK Tools 菜单列出了系统内安装的所有开发工具和实用程序，它们有助于帮助开发者建立和调试应用程序

PVIEW95 的源代码可以在\WIN32SDK\MSTOOLS\SAMPLES\FRMWORK\PVIEW95 目录下面找到。Win95 内的 Win32 进程看起来与它们在 WinNT 内的样子是不同的。这是由于考虑到由 Win16 代码带来的向后兼容问题，况且 Win95 本身就自带了一部分相当不错的 16 位代码。这就至少从一个方面解释了为什么 WinNT 的 PVIEWER 程序不能在

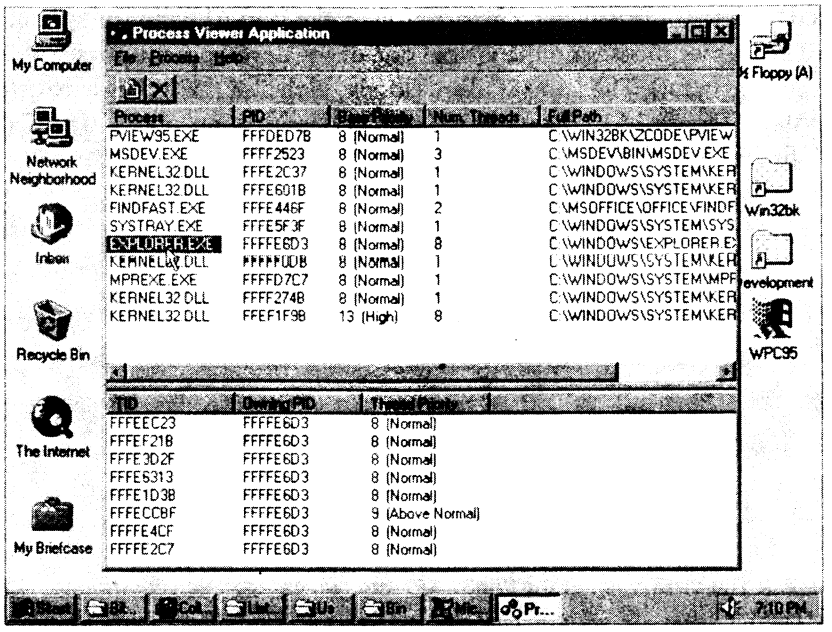


图 2-4　PVIEW95 是 NT 的 PVIEWER. EXE 的一个变体

Win95 下面正确运行。

在开始深入分析开发模式之前，应该先花些时间查看一下帮助文件。不管帮助文件在结构和内容方面作出了多么大的改进，获得自己希望的信息仍然不是那么方便的，至少在当前这个阶段如此。这主要是由于两方面的原因：可用的信息在许多领域仍然是比较初级的；另外，对于 Win32 提供的功能和在 Win95 里已真正实现的功能来说，这两者间的差距仍然是相当暧昧的。

2.3　开发模式和 API

这儿要注意的一个关键之处在于，我们应该在自己的开发方向上多下一番功夫。通过探讨 Win32 的核心 API（换句话说就是建立和管理屏幕窗口以及进行内存管理和实现其他相关功能的基本函数），我们可以归纳出两个不同的组别：Win API 和系统 API。前者强调的是开发阶段的视觉效果（建立窗口是最基本的任务）；后者则与系统的内部特性具有更紧密的联系，如图 2-5 所示。这并不是一种很精确的分类，读者在其他地方也许根本找不出这样的分类。

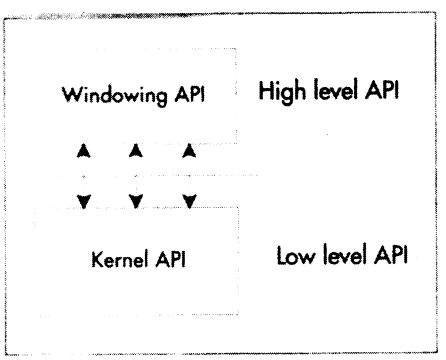


图 2-5　Win32 API 有两种不同的类别：窗口处理 API 和系统 API

高级的窗口处理 API 通常能以最简便的方式实现某种特定的功能，而只有使用系统 API 才能最终实现预期的目标。举个例子来说，假设你现在必须在两个系统之间实现一些进程间通信。这儿有许多种方案可供选择——剪贴板、动态数据交换（DDE）、对象嵌入与链接（OLE）、内存共享块、管道、命名管道、远程进程调用（RPC）以及 Win Sockets 等等。事实上，我们并不能从这一大堆名字上得到什么有价值的信息。要在其中选出符合自己开发需求的一种最佳方案是件很困难的事情。在很多情况下，我们选择的第一种通信方式都是划归窗口处理这一类的。

SendMessage () 这个 API 函数允许把一条消息发送给另外一个窗口，不管那个窗口是在哪个线程内建立的。这个简单的操作将对多个内部系统组件发生影响。假如目标窗口从属于一个不同的进程，消息就必须穿越进程地址空间，进入一个新的进程地址空间，然后修改对相关数据的所有引用。

SendMessage () 是实现进程间通信的一种最有效的方式吗？显然不是。另外有些专门的 API 正是针对这个问题设计的，它们提供了更高的性能和更大的自由程度。这些 API 的一部分是由 SendMessage () 调用的，这样就省去了开发者更多的麻烦。当然，开发者不得不以速度和控制权作为代价。根据经验来看，如果要设计一个关键性的 Win32 应用程序组件，开发者应该尽可能地使用低级的系统 API。这样虽然牺牲了易用性，然而却换来了对执行流程更加牢靠的控制权。作为一名优秀的开发者，这应该是我们的基本准则。

2.4 Win32 应用程序的建立

一切都准备好了吗？已经安装好了 Visual C++，SDK 和 Windows 95（这是最重要的）吗？准备好所有这些原料后，我们接下来就可以共同烹调出这十年来最有意义的一道菜——一个 Win32 应用程序。只需双击 Visual C++图标，数秒钟后，一个集成化的开发环境就会在你的面前展现出来。剩下来的工作就是编码，编码，再编码；不断地构建程序和进行测试。

开发 Win32 应用程序的时候，我们要遇到五种基本的文件类型：一个源代码文件（.C 或者.CPP）、一个模块定义文件（.DEF）、一个或者多个头文件（.h）、一个资源文件（.RC）以及一个项目文件（.MAK），请大家参阅表 2-1。

表 2-1 建立一个 Win32 应用程序所需的五种基本文件类型

扩展名	说明	扩展名	说明
.C 或 .CPP	C 源代码文件	.RC	资源文件
.DEF	模块定义文件	.MAK	项目文件
.h	头文件		

在早期 Win16 编程的光荣岁月里，开发者必须人工编写所有这些文件，只能从开发环境处获得非常有限的帮助。第一个编程工具是于 80 年代末期在市场上出现的。现在，Visual C++编译器以市场领导者的姿态出现了。它由一个编辑器、一个项目文件制作程序、一个编译程序以及一个调试程序组成。这样一来，以前让人感到沉闷的工作（比如编写项目文件的代码等等）就成了历史的记忆。

.MAK 文件的建立完全是自动完成的。开始一个新的项目之前，建立 .MAK 文件通常是

要采取的第一个操作。对于标准的 Win32 程序来说，建立一个 .DEF 文件已不再是强制性的了。除此以外，由 .DEF 文件支持的信息量也已经显著地减少，从而简化了这种任务。

注意：在 Win32 应用程序里增加资源也比过去简单多了，这是由于在开发环境里集成了一些特定的资源编辑器。在这方面作出的其他许多改进也是倍受人们欢迎的。以前，编程人员往往由于不正确的资源定义导致编译和运行期错误，这种情况现在已经显著地减少了。

除了这些基本的文件类型以外，其他常见的还有 .BMP，.ICO，.RES，和 .DLG 等扩展名（参见表 2-2）。这些文件都是以包含了二进制资源（通常是位图）或者资源模板（对话框模板）的资源文件（.RC）为中心展开的。

表 2-2　一些附加的文件，它们使 Win32 应用程序开发的标准模块得以完整

扩展名	说明	扩展名	说明
.RES	编译好的资源	.DLG	对话框模板
.ICO	图标文件	.BMP	位图文件

注意：对资源进行处理的时候，Visual C++会自动建立一个名为 RESOURCE.H 的新文件。所以，在本书几乎所有的例子里，大家都能找到一个特殊的文件，这个文件内包含了在 Visual C++环境内生成的资源的 ID。

接下来，假设你已经成功地通过了编译和链接步骤。在这种情况下，Visual C++将生成一个可执行模块，它带有标准的 .EXE 扩展名。就 Win32 执行程序的内部本质来说，它们与以前的 Win16 格式和老式的 MS-DOS 格式都是有区别的。表 2-3 列出了由不同 Microsoft 操作系统实现的所有可执行格式。

表 2-3　Microsoft 操作系统家族的不同执行格式

EXE	类型	说明
老式 .EXE	OE	MS-DOS 执行程序
新的 .EXE	NE	Win16 执行程序
可移植的 .EXE	PE	NT 和 MS Windows 95 执行程序

PE 格式最引人注目的一个特征是它缺少 MS-DOS 的“存根文件”。当 Windows 还不是一个完整的操作系统、仍然需要 MS-DOS 才能正确运行的时候，存根文件几乎是必须的。PE 执行程序只能在 Windows 95 和 NT 内才能运行，这是两种完整的操作系统，不再需要 MS-DOS 的介入。

NE 执行程序内包含了一个 MS-DOS 模块，假如在命令行启动一个 Win16 应用程序，MS-DOS 系统装载器就会自动激活这个模块。对于 Win16 SDK 提供的 MS-DOS 存根程序 WINSTUB.EXE 来说，下面这条出错消息是它唯一的输出内容：“This program requires Microsoft Windows”（这个程序需要 Microsoft Windows）。如果在 DOS 提示符下执行一个 Win16 程序，这条消息将马上在命令行内显示出来。

注意：PE 对于 Windows NT 和 Windows 95 来说是通用的，这样一来，Windows NT 在 Intel 或者兼容处理器上运行的时候，两种平台才能在二进制级别保证完全的兼容性。

2.4.1 Windows 95 链接程序

成功地编译了源文件以后，Visual C++编译程序就会自动执行 LINK.EXE。如果是在命令行激活的 LINK.EXE，这个程序就会产生下面这些输出：

```
Microsoft (R) 32-Bit Incremental Linker Version 2.55
Copyright (C) Microsoft Corp 1992-94. All rights reserved.

usage: LINK [options] [files] [@commandfile]
```

其中的“选项”(options) 列表相当长，如表 2-4 所示。一个开发者很少需要对它们的任何一项进行手工设置。一般情况下，我们都愿意在如图 2-6 所示的 Project Settings 属性表内选择适当的选项，从而实现对链接过程的定制。

表 2-4 Windows 95 的链接选项

```
/ALIGN: #
/BASE: {地址@文件名，关键字}
/COMMENT: 备注
/DEBUG
/DEBUGTYPE: {CV | COFF | BOTH}
/DEF: 文件名
/DEFAULTLIB: 库 [, 库]
/DLL
/ENTRY: 符号
/EXETYPE: {DEV386 | DYNAMIC}
/EXPORT: 符号
/FIXED
/FORCE [: {MULTIPLE | UNRESOLVED}]
/HEAP: 预约 [, 委托]
/IMPLIB: 文件名
/INCLUDE: 符号
/INCREMENTAL: {YES | NO}
/MAC: {BUNDLE | NOBUNDLE | TYPE = xxxx | CREATOR = xxxx}
/MACDATA: 文件名
/MACHINE: {IX86 | M68K}
/MACRES: 文件名
/MAP: 文件名
/NODEFAULTLIB {: 库 [, 库…]}
/NOENTRY
/NOLOGO
/OPT: {REF | NOREF}
/ORDER: @文件名
/OUT: 文件名
/PDB: {文件名 | NONE}
/PROFILE
/RELEASE
/SECTION: 名称，[E] [R] [W] [S] [D] [K] [L] [P] [X]
/STACK: 预约 [, 委托]
/STUB: 文件名
/SUBSYSTEM: {NATIVE | WINDOWS | CONSOLE | POSIX} [, # [.##]]
/VERBOSE
/VERSION: # [.#]
/VXD
/WARN [: 警告级别]
```

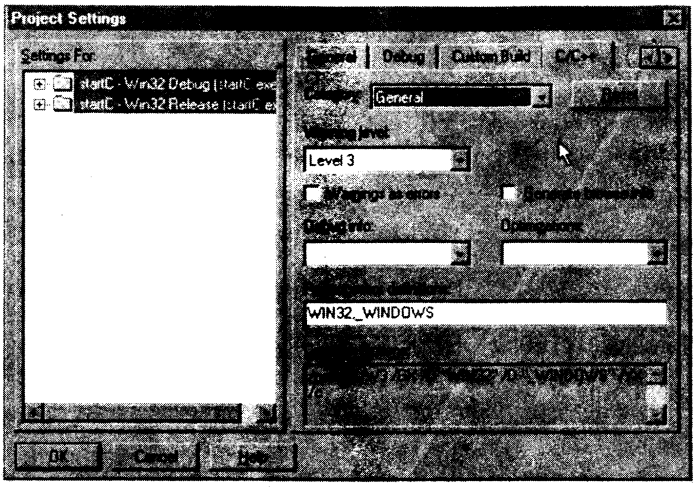


图 2-6　通过几个特定的页，利用 Project Settings 属性表便可简便地对一个项目文件进行定制

1. DUMPBIN 实用程序

链接程序建立好执行程序以后，通过使用由开发环境提供的 DUMPBIN 实用程序，就可以方便地对执行程序进行检查了。这个程序将显示出与 .EXE 文件整体特征有关的全部信息。DUMPBIN 也能显示出文件的组件和位置，比如输入和输出函数以及段名称等等。如果想调用 DUMPBIN，只需在 MS-DOS 区的命令行内键入它的名字，然后按回车键就可以了。该程序的输出情况如下所示，所有相关的标志请参见表 2-5。

```
Microsoft (R) COFF Binary File Dumper Version 2.55
Copyright (C) Microsoft Corp 1992-94. All rights reserved.

usage: DUMPBIN [options] [files]
```

表 2-5　DUMPBIN 标志

/ALL
/ARCHIVEMEMBERS
/DISASM
/EXPORTS
/FPO
/HEADERS
/IMPORTS
/LINENUMBERS
/LINKERMEMBER [:{1 | 2}]
/OUT:文件名
/PDATA
/RAWDATE [:{NONE | BYTES | SHORTS | LONGS} [,#]]
/RELOOCATIONS
/SECTION:名称
/SUMMARY
/SYMBOLS

DUMPBIN 提供了与一个 Win32 执行程序内部特征有关的许多有用的信息。如果用/IMPORTS 标志来调用 DUMPBIN，就可以列出源代码内调用的所有系统 API，这样更能深入地了解某种特性是如何实现的。

Win32 进程不再是由分段组成的，和 Win16 比较起来，这种变化在 .DEF 文件的语法以及 DUMPBIN 的/SECTION 标志上面都得到了反映。

2. 对 PE 执行程序的剖析

下面是对一个 PE 执行程序进行段结构分析的示例：

```
Microsoft (R) COFF Binary File Dumper Version 2.55
Copyright (C) Microsoft Corp 1992-94. All rights reserved.

DUMPBIN : warning LNK4044: unrecognized option " SECTION";
ignored

Dump of file \WIN32BK\WIN32_01\LIST01\WINDEGUB\PLATFORM.EXE

File Type: EXECUTABLE IMAGE

  Summary

     2000 .bss
     1000 .data
     1000 .edata
     1000 .idata
     1000 .rdata
     1000 .reloc
    2D000 .rsrc
     3000 .text
```

表 2-6 列出了对 PLATFORM.EXE 进行检查后，DUMPBIN 的输出情况。

表 2-6　对一个标准 PE 执行文件的段进行分析

段　名	说　明	段　名	说　明
.text	应用程序源代码	.rdata	调试目录和信息
.idata	输入函数	.bss	未初始化的静态和源文件标识符
.edata	输出函数	.data	已初始化的静态和源文件标识符
.reloc	重分配信息	.rsrc	应用程序资源

由 DUMPBIN 产生的输出显示出了 8 个不同的段。.text 段是指由编译程序和链接程序生成的全部应用程序代码。链接程序提取出了单独源文件内的所有代码段，然后把它们合并成一个巨大的代码段。事实上，.text 段内包含的代码要比编译程序和 .OBJ 文件（链接过程中要用到）生成的代码多一些。这些多余的代码与系统调用一个独立模块内驻留的函数的机制有关。典型地，假如在源代码内调用了一个 Win32 API，就需要用到这些多余的代码。

与输入函数有关的所有信息都存储于 .idata 段内，这些信息包括系统 API 入口点的真实地址。对同一个 API 的所有调用都会跳转至 .text 段内一个独立的位置处。同时，对 .idata 地址的所有引用都存储于这个 .text 段内。这种技术的优点在于：如果系统装载程序必须修复与输入函数有关的所有伪地址，那么它的工作强度就会大大降低。针对每个输入函数，这种工作只限于在执行程序的一个单一地点处进行。所以，.text 段列出了对 .idata 段的几次引用，每调用一个输入函数，就要进行一次引用。

源代码内声明的所有标识符都能在 .bss 和 .data 段内找到；.bss 用于尚未初始化的数据，而.data则用于已初始化的数据。系统资源存储于 .rsrc 段内，而 .edata 和 .idata 段则各自指定输出和输入名称表。.reloc 段内包含的信息是为系统装载程序准备的，该程序利用这些信息重新分配 .EXE 文件内从其他模块输入的所有组件。

2.4.2 模块定义文件

在链接程序的最后构建过程中，假如所有信息都已合并到了一起，从而构成了一个执行程序，那么 Win32 应用程序就不需要用 .DEF 文件对链接程序发出指令。构建 DLL 的时候，仍然需要用到 .DEF 文件，这是由于它包含了生成正确库文件（.LIB）必需的一些关键信息。尽管 .DEF 文件的地位已显得次要，并且对链接程序的影响力也减弱了，但是我仍然决定让本书的每个项目都包含一个 .DEF 文件。32 位 Windows 开发模式极大简化了它对链接程序行为的影响。表 2-7 列出了 Win32 开发环境支持的所有指令。

```
NAME              INCLUDE
DESCRIPTION       'Include example — The Windows
95 Developers Guide 1995'

EXPORTS
ClientWndProc
```

表 2-7 Win32 开发环境支持的 .DEF 文件指令

NAME 模块名	为执行程序分配一个名字。假如已在链接程序命令行内键入了一个名字，这儿的名字将被忽略
LIBRARY 模块名	指示链接程序生成一个 DLL，而不是一个标准的执行程序。LIBRARY 和 NAME 是相互排斥的，不能同时使用
DESCRIPTION '说明'	一个正文串，用于说明 .DEF 文件和它指定的执行程序/DLL
CODE PRELOAD LOADONCALL MOVEABLE DISCARDABLE FIXED	为构成应用程序的所有代码段定义一般属性。两个载入选项（PRELOAD 和 LOADONCALL）和三个内存选项（MOVABLE，DISCARDABLE 和 FIXED）很少用到
DATA PRELOAD LOADONCALL MOVEABLE DISCARDABLE FIXED SINGLE MULTIPLE	

（续）

SECTIONS 段列表	为构成应用程序的所有代码段定义一般属性。两个载入选项（PRELOAD 和 LOADONCALL）和三个内存选项（MOVABLE，DISCARDABLE 和 FIXED）很少用到。让开发者为代码和数据段分配不同的属性，把它们与由 CODE 和 DATA 命令分配的属性区分开来
HEAPSIZE	应用程序堆的初始尺寸
STACKSIZE	应用程序的堆栈尺寸
IMPORTS 函数列表	列出应用程序内调用的所有函数，尽管这些函数并非物理性地存在于源代码内。这个命令几乎永远不会出现，因为它已被输入库文件取代了
EXPORTS 函数列表	列出应用程序源代码内编写的所有函数，这些函数将通过其他模块调用

下面是本书使用的一个标准 .DEF 文件的布局：

```
NAME     Tweny
DESCRIPTION          'Win32 project version 1.0'

EXPORTS
ClientWndProc
```

在用于 Win32 进程的这种典型 .DEF 文件里，信息量被缩减到了最低程度。

在 .DEF 文件支持的所有命令里，只有 STACKSIZE 和 HEAPSIZE 需要引起我们特别的注意。对于 Win16 来说，这两种类型的信息在许多应用程序里都是很关键的。它们能让我们更好地协调应用程序的运行，并且优化程序对整体系统资源的影响。在缺省情况下，每个 Win32 进程都为应用程序堆栈预约了 0x00100000 字节或者说 256 个页的线性地址空间，并且仅对第一个页进行了委托。如果有必要，也可以在执行过程中对其他页进行委托。相同的值和委托方式也应用于应用程序堆。所以，除了在特定的情况下，在一个 .DEF 文件里对这两个命令进行说明几乎是毫无用处的。它们的值总是指定了最大数量的预约页，而不是委托页。由于系统内存的分页特性，指定一个小于 4096 的值是没有意义的，因为 4096 是最小的分配单位。

2.4.3 资源文件

资源是 Win32 进程的基本组件。这儿的“资源”与经常限制系统可用性的那种臭名昭著的系统资源是沾不上一点边的。开发过程中用到的资源是指一系列不同类的对象，有些是 ASCII 格式，有些则是二进制格式。例如，显示于程序标题栏下方的菜单栏几乎一定是某种资源在执行过程中载入应用程序后的结果。对话框也是一种资源。所以，图标、位图以及显示于消息框和客户区内部的正文串都是资源。正如我们刚才强调的那样，根据不同的格式，资源被分为两类：正文和二进制资源。正文资源包括菜单模板、字符串、加速键、对话框模板以及版本信息。二进制资源则由图标、位图、字体、元文件以及光标构成。除此以外，开发者完全可以针对应用程序内的一种特定用途（原始数据）设计出某种新的资源，对此是没有什么限制的。

从物理意义上说，所有资源都包含于资源文件（.RC）内；或者指定了对某个外部文件的引用，在那个文件内则包含了实际的信息。第二种情况主要应用于二进制资源，这些二进制资源是由某些工具生成的，比如位图编辑器和图标编辑器等等。资源编译程序对 .RC 文件进

行编译后，将生成带有 .RES 扩展名的一个二进制版本。链接程序建立好执行模块以后，编译资源就会插入 .EXE 内，并且成为 .EXE 模块一个不可缺少的部分。

若想建立一个 Win32 应用程序，开发者至少都要用到一个源文件模块（.C），其中包含一个或者多个头文件（.H）。对这两种文件进行编译后，将生成一个 .OBJ 文件。随后，系统链接程序会把这个 .OBJ 文件成功转换成对应的 .EXE 程序。在这种处理过程中，.DEF 文件会告诉链接程序如何对不同组件内的信息进行汇编，同时由几个 .LIB 文件去补足丢失的引用，并且提供附加的组件。同时，资源编译程序会对 .RC 文件内的资源进行编译，最后把它们插入执行程序内。

2.4.4 头文件

我在这儿问大家一个问题：在哪儿对所有 API 以及它们相关的数据类型和标志进行声明？在过去，这个问题的答案是相当简单和直接的：在 WINDOWS.H 里。现在，可怜的 Win32 开发者的生活正在变得越来越困难，这种趋势也反映在头文件上面。

随着 Win32 的问世，WINDOWS.H 不再成为包含系统描述的唯一文件。更准确地说，它现在只扮演了采用分级结构的一种入口点的角色，其中列出了数十个不同的文件。所以，WINDOWS.H 文件现在只有 4K 的长度，而以前在 Win16 里却有 150K 之长。这并不是采取了某种可怕的压缩技术的后果，而是由于以前包含在这个文件内的大量信息已经迁移到其他文件内的缘故。这种方案带来了一些优点，但也具有一些缺点。主要的缺点在于获得所需信息之前，必须先对许多文件进行检查。此外，这种操作看起来就像在没有金属探测器的情况下，在漫长的海滩上找一根针。开发环境没有提供能帮助开发者的任何工具，也没有提供任何基本的指导信息。从另外一方面来看，与特定编程主题有关的 API 和数据类型现在有集成到一个单独文件内的趋势，这样就限制了以前在 Win16 里对信息的传播级别。

从开发的角度来看，我们可以在 Win32 源代码文件的最开头设置一个 WINDOWS.H，从而保证可以在编译阶段使用基本的 API 引用。什么是基本的 API 呢？我的意思是指数百个函数，而不是指这些函数的有限集合。面对某些特定的编程问题时，我们会增加其他一些头文件。这些特定的编程问题包括多媒体、笔式计算以及远程进程调用（RPC）。

下面是从本书提供的一个 Win32 示例中提取出来的一个代码片段：

```
// INCLUDE.C
// Stefano Maruzzi 1995

// LIST02-02
// Listing all the include files used by VC++

// include files
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include <win32bk.h>
#include <winnetwk.h>
```

```
#include " resource.h"
#include " include.h"
...
```

除了 WINDOWS.H 以外，上面这段程序还包含了 WINDOWSX.H，COMMCTRL.H，WINNETWK.H 和 WIN32BK.H，这些头文件可以满足应用程序的所有需求。这个程序有两个附加的专用头文件：RESOURCE.H 和 INCLUDE.H，其中包含了应用程序特定的信息。

怎样才能知道何时把一个特定的头文件增添到包容模块列表内呢？好了，假如没有在其中包含恰当的头文件，那么编译程序一旦碰到了没有原型化的 API，就会返回一个编译错误，这样就会禁止 .OBJ 文件的生成。通过查看联机文档，开发者可以知道自己是否定义了正确的头文件。除此之外，在一个单独的 API 和对其进行原型化的包容文件之间就没有其他任何联系了。

随同 SDK 提供了数量众多的头文件。表 2-8 只列出了实际存在于 WINDOWS.H 内的头文件，也就是说，这些头文件将位于分级结构的起始处。

表 2-8 WINDOWS.H 里包含的头文件

头 文 件	头 文 件	头 文 件	头 文 件
cderr.h	stdarg.h	mcx.h	winperf.h
commdlg.h	winbase.h	mmsystem.h	winreg.h
dde.h	wincon.h	nb30.h	winsock.h
ddeml.h	windef.h	ole.h	winspool.h
dlgs.h	wingdi.h	ole2.h	winsvc.h
drivinit.h	winmm.h	rpc.h	winuser.h
excpt.h	winnetwk.h	shellapi.h	winver.h
lzexpand.h	winnls.h		

WINDOWS.H 文件限制了自己包含许多其他的头文件，这些头文件用树形来代表分级结构。就它们的名字来说，我们不可能猜测出对应的内容，尽管其中一些应该引起我们特别的注意。现在让我们先从 WINDEF.H 开始。

1. WINDEF.H 头文件

WINDEF.H 包含了对许多简单和集合数据的定义，这些数据是在 Win32 编程中经常都要用到的。这个文件的一小段用于定义 Win32 的基本数据类型，这些数据类型是一系列“typedef”（类型定义）关键字的集合，这个关键字的作用是建立新的数据类型，如下所示。

```
typedef unsigned long ULONG;
typedef ULONG *PULONG;
typedef unsigned short USHORT;
typedef USHORT *PUSHORT;
typedef unsigned char UCHAR;
typedef UCHAR *PUCHAR;
typedef char *PSZ;
```

```
typedef unsigned long DWORD;
typedef int BOOL;
typedef unsigned char BYTE;
typedef unsigned short WORD;
typedef float FLOAT;
typedef FLOAT *PFLOAT;
typedef BOOL near *PBOOL;
typedef BOOL far *LPBOOL;
typedef BYTE near *PBYTE;
typedef BYTE far *LPBYTE;
typedef int near *PINT;
typedef int far *LPINT;
typedef WORD near *PWORD;
typedef WORD far *LPWORD;
typedef long far *LPLONG;
typedef DWORD near *PDWORD;
typedef DWORD for *LPDWORD;
typedef void far *LPVOID;
typedef CONST void far *LPCVOID;
typedef int INT;
typedef unsigned int UINT;
typedef unsigned int *PUINT;
typedef WORD ATOM;
```

不要在其中的 near 和 far 关键字上面纠缠不清。正如大家不久要看到的那样，在 Win32 编程环境下，这些关键字根本就没有什么意义。之所以在 WINDEF.H 里还保留着它们，要么是由于设计者粗心，要么就是由于一个错误造成的。不管是什么原因，假如你在其他包容文件或者联机文档里发现了它，那么完全可以把它忽略掉。

新建立的一种数据类型是 USHORT——Win16 整数国王的王子，在 Win32 里却失宠了。在 Win32 里，声明一个两字节的整数既没有什么好处，也不会有什么负作用。这是由于系统是以四字节为界限排列对象的。所以，假如你需要一个循环计数器，可以依靠 int（整数）或者 UINT（无符号整数）来实现。

在这个 32 位的世界里，整数数据占据的范围如表 2-9 所示。

表 2-9　Win16 和 Win32 的整数数据类型

数据	Win16	Win32
short	2	2
int	2	4
long	4	4

假如需要用一个整数来简化从 Win16 到 Win32 的移植，那么最好使用 int 类型。正如我们不久要讲到的那样，到了 Win32，Win16 里非常流行的一些两字节长编程对象都已经变成了四字节。

WINDEF.H 提供的信息对于 Win32 采纳的函数调用规范来说是非常有用的。以前的 Win16 程序员都还记得 Win16 API 的虚拟 Pascal Calling Convention（Pascal 调用规范）。在 Win32 里，所有函数都具有 WINAPI 类型，这是 Win16 已经存在的一种定义，只是在 32 位世界里已经具有了不同的含义。微软的系统设计者定义了一种新的调用规范，对于 Windows 95 来说，我们可以把它叫作_stcall。它并不是一个新生事物，因为早在几年前它就出现了，那时是作为标准 C 关键字的一个扩展而推出的。然而，和以前的_cdecl（标准的 C 调用规范）比较起来，现在实施的这种_stdcall 又提供了几个新增的优点：

▶更快的执行速度

▶更紧凑的代码

▶强制使用函数原型

参数是按照值或者引用（前提是已明确地指定）的顺序从右到左传递的。堆栈的清除通常与被调用的函数有关。与函数和调用规范有关的所有 Win16 定义都已进行了修改，从而与新的规范保持一致，如下所示：

```
...
#define pascal          _stdcall
...
#define CALLBACK    _stdcall
#define WINAPI      _stdcall
#define WINAPIV     _cdecl
#define APIENTRY    WINAPI
#define APIPRIVATE _stdcall
#define PASCAL      _stdcall
...
```

正如大家下一章刚开头就要学习的那样，Win32 进程入口点是类型 WINAPI 的一个功能，而不像以前在 Win16 里，入口点是 PASCAL 的一个功能。除此以外，那些难以应付的 far，huge 和 near 指针已经不复存在了；对于内存模式来说也是如此。这并不是一个奇迹，而是放弃 16 位世界的必然结果。就这一问题来说，WINDEF.H 对于这些信息的回顾也是有用的。

```
...
#undef far
#undef near
#undef pascal

#define far
```

```
#define near
...
#define FAR     far
#define NEAR    near
...
```

正如读者看到的那样，无论 far 还是 near 都已从一个纯粹的 Win32 开发者那里完全消失了。该是结束恶梦的时候了！在完成对 WINDEF.H 的分析之前，我们还不得不先转移到 WINNT.H 上面，因为这个文件是包含于 WINDEF.H 里的。

2. WINNT.H 头文件

乍一看到这个名字，你首先想到几乎肯定是 Windows NT，你也许会认为这是一个 for NT 的专用文件。但是事实上，这个头文件里并没有专门为 NT 的 Win32 编程提供的任何数据和信息。通常情况下，它只是一个 Win32 包容文件，对于 Win95 来说也是相当有用的。从长度来看，这个文件超过 130K，其中包含了大量 Win32 API 定义。通过深入探索 WINNT.H，大家会在里找到句柄定义，这是 Win32 编程的一个关键性组件。在 Win32 里，句柄就是指向一个“空白”(void *) 的指针，就像下面这样：

```
...
#ifdef STRICT
typedef void * HANDLE;
#define DECLARE_HANDLE (name) struct name##_ { int unused;
}; typedef struct name##_ * name
#else
typedef PVOID HANDLE;
#define DECLARE_HANDLE (name) typedef HANDLE name
#endif
typedef HANDLE * PHANDLE;
...
```

这儿已不再是 Win16 里使用的 unsigned short。从编程的级别来说，这种变化是有些含义的。句柄现在占据了四个字节，而 Win16 里只用了两个字节。读者应该记住这个细节，这样很容易便可抓住一段现成 Win16 代码的逻辑顺序，然后把它移植到 32 位环境里。另外要注意的一点是许多函数和消息都受到了这种变动的影响，因为在某些情况下，把一个句柄和另外一个数据压缩到四字节的区域里已经不再是那么方便了。比如 WM_COMMAND 消息。

WINDEF.H 定义了许多基于 WINNT.H 通用句柄定义的句柄类型。

```
...
DECLARE_HANDLE (HWND);
DECLARE_HANDLE (HHOOK);
```

```
...
typedef HANDLE NEAR *SPHANDLE;
typedef HANDLE FAR *LPHANDLE;
typedef HANDLE HGLOBAL;
typedef HANDLE HLOCAL;
typedef HANDLE GLOBALHANDLE;
typedef HANDLE LOCALHANDLE;
...
#ifdef STRICT
typedef void NEAR * HGDIOBJ;
#else
DECLARE_HANDLE (HGDIOBJ);
#endif
...
DECLARE_HANDLE (HACCEL);
DECLARE_HANDLE (HBITMAP);
DECLARE_HANDLE (HBRUSH);
DECLARE_HANDLE (HDC);
DECLARE_HANDLE (HDESK);
DECLARE_HANDLE (HENHMETAFILE);
DECLARE_HANDLE (HFONT);
DECLARE_HANDLE (HICON);
DECLARE_HANDLE (HMENU);
DECLARE_HANDLE (HMETAFILE);
DECLARE_HANDLE (HINSTANCE);
typedef HINSTANCE HMODULE;
DECLARE_HANDLE (HPALETTE);
DECLARE_HANDLE (HPEN);
DECLARE_HANDLE (HCOLORSPACE);
DECLARE_HANDLE (HRGN);
DECLARE_HANDLE (HRSRC);
DECLARE_HANDLE (HSTR);
DECLARE_HANDLE (HWINSTA);
DECLARE_HANDLE (HKL);
typedef int HFILE;
typedef HICON HCURSOR;
```

在下一章里，我们将对这些句柄进行详细的讨论。由于系统的改进，Win16 里的几个约束在 Win32 里已经不复存在了。现在，可用句柄的数量有某些情况下是足够多的，我们完全

可以认为句柄的数量是无限的。请参考表 2-10。

表 2-10　Win32 的句柄限制

每一进程的句柄限制
每进程 16384 个 GDI 句柄
每进程 1073741824 个 KERNEL 句柄
系统范围内的 65536 个 USER 句柄

接下来，让我们考虑定义指向窗口进程的一个指针。除了老式的定义（带有的属性与分段的 16 位编程环境有关）以外，在 Win32 里，有一种非常简单和线性的定义可以很好地描述 Win32 的编程模式。如下所示：

```
typedef int (FAR WINAPI *FARPROC) ();
typedef int (NEAR WINAPI *NEARPROC) ();
typedef int (WINAPI *PROC) ();
```

其中，PROC 是指向 WINAPI 函数的一个指针，该函数返回的是一个 int 数值。Win32 编程环境内的所有窗口和对话进程都是以这种定义为基础的。

现在让我们转到 WINDOWSX. H，这是 Windows 编程员都很熟悉的一位老朋友了，因为它最初的引入可以追溯到 Windows 3. x 的时代。

3. WINDOWSX. H 头文件

这个头文件名的最后一个 X 是“扩展”的意思，它明确表明了自己的作用是对基本的 Win32 定义进行扩展。这个文件内包含了三种类型的信息：

▶宏 API

▶消息分解器

▶控件 API

WINDOWSX. H 是用一些正文和定义打包起来的。然而，这并不表示它难于辨认和不易理解。宏 API 是一系列宏定义，这些宏可以对标准的 API 函数进行处理，使其更紧凑、更易于辨认。这些宏可以让我们更容易辨读源文件；也使文件的布局变得更加清爽。下面的这个例子可以证明这一点。

假设你现在准备取回与一个特定窗口联系起来的“风格”二进制位。为了达到这个目的，可以按照下述的方式调用 GetWindowLong () 函数：

```
lStyle = GetWindowLong (hwnd, GWL_STYLE);
```

这个函数的语法要求用到一个窗口句柄 (hwnd) 和一个标志 (GWL_STYLE)，这些句柄和标志是在包容文件分级结构内的其他某个地方定义的。WINDOWSX. H 提供了一个宏函数：GetWindowStyle (hwnd)，通过可以用更出色的方式实现上面那个调用：

```
#define GetWindowStyle (hwnd) ( (DWORD) GetWindowLong (hwnd, GWL_
STYLE))
```

这样一来，调用就变成了下面这个样子：

```
lStyle = GetWindowStyle (hwnd);
```

看看 WINDOWSX. H 宏，我们就会注意到源代码和自然语言已经相当接近了，不再象编

程语言那样晦涩难懂。尽管如此，这些宏函数仍然需要开发者具有 Win32 API 深层次的知识积累，并且要有探究这些纷乱、复杂的包容文件的决心和毅力。事实上，联机文档或者书面出版物都没有对 WINDOWSX.H 里的宏函数进行详细的探讨。只能通过自己去浏览 WINDOWSX.H，才能知道哪一个适合自己的需要，并且知道它们的格式和功能。在许多情况下，这种工作都显得很复杂、很令人生厌。在这本书里，我在少数几个场合下面采用了宏 API——只是那些在联机文档里建议使用的。

消息分解器（Message cracker）可以对源代码的外观进行较大的改动。它们是一系列宏的集合，这些宏的宗旨就是取代一个窗口进程内的小部分代码——正如大家在下一章看到的那样，Windows 程序里将大量用到函数。

摘自 WINDOWSX.H 里的下面这个代码段引用了与 WM_CREATE 消息联系在一起的消息分解器：

```
/* BOOL Cls_ OnCreate (HWND hwnd, CREATESTRUCT FAR * lpCreateStruct) */
#define HANDLE_WM_CREATE (fn, hwnd, wParam, lParam) \
    ((fn)((hwnd), (CREATESTRUCT FAR *)(lParam)) ? 0L : (LRESULT)-1L)
#define FORWARD_WM_CREATE(fn, hwnd, lpCreateStruct) \
    (BOOL)(DWORD)(fn)((hwnd), WM_ CREATE, 0, (LPARAM)(CREATESTRUCTFAR *)(lpCreateStruct))
```

我必须承认自己并不喜欢用消息分解器。它们的意图是把代码块独立出去，这些代码块与窗口进程内拦截下来的一条消息有关。为了实现这种意图，必须建立一个独立的函数，用它来专门对那条消息进行处理。这种方案并没有简化源代码的读写处理。这就意味着窗口进程（代码的中心部分）将被缩减到几行代码，同时在源代码内散布无数个小函数。我们可以检查由 SDK 提供的几个例子，评估一下假如涉及到了消息分解器，那么对这种文件进行辨读将会变得多么复杂和混乱。

控件 API 也是一系列宏的集合，这些宏的作用是简化与预定义窗口类（第 9 章）和通用控件（第 10 章）的交互作用。预定义的窗口类和通用控件将分别在本书的第 9 章“预定义的窗口类”和第 10 章“Windows 95 通用控件”内讲述。应用程序对控件进行处理的时候，先要通过 SendMessage() API 函数发送出一系列预先定义好的消息。这是一种具有常规用途的函数，正如读者将在第四章“消息和重画模式”里看到的那样，该函数可以在上百种不同的情况下应用。控件 API 则会能以一种更易于理解和辨读的格式重新加工对 SendMessage() 函数的这些调用。这样一来，通过内部的封装集成，每次调用所需的参数数量就大大减少了。

假想我们的应用程序提供了一个 EDIT 窗口，在里包含了一些正文。我们现在对这些正文到底有多少行发生了兴趣。这时需要知道的全部信息就是 EDIT 的窗口句柄。EDIT 窗口句柄是作为控件 API 函数 Edit_GetLineCount () 唯一的一个参数传递的，注意这个 API 函数是在 WINDOWSX.H 头文件里定义的。

```
#define Edit_GetLineCount(hwndCtl)((int)(DWORD)SendMessage((hwndCtl),
```

EM_GETLINECOUNT, 0, 0L))

在应用程序源代码里,它就变成了:

```
...
nLineCount = Edit_GetLineCount(hwnd);
...
```

其中,hwnd 表示 EDIT 窗口的句柄。假如没有使用控件 API 宏,它看起来就会象下面这个样子:

```
...
nLineCount = (int)SendMessage((hwnd), EM_GETLINECOUNT, 0, 0);
...
```

对于一个经验丰富的 Windows 程序员来说,这两种形式之间并没有什么太大的区别。控件 API 的语法看起来更简洁、紧凑,而且最重要的一点在于,它是针对当前的情况专门设计的。我之所以建议大家提供对控件 API 的支持就是出于对这个原因的考虑。同时要注意的一点是,联机文档坚持引用了这种宏,它也建议开发者在自己的程序中采纳这种宏函数。

2.4.5 WIN32BK.H 头文件

尽管随同 SDK 提供了许多包容文件,但我还是感觉到有必要增加一个新的头文件,在这里包含某些定制的数据类型、宏以及定义。我把它叫作 WIN32BK.H,也就是这本 Win32 书籍(Book)的意思。Windows 95 支持长文件名吗?是的,的确如此。Windows NT 支持长文件名吗?是的。Windows 的其他任何一种 Win32 开发版本也支持长文件名吗?不是。Win95 和 Windows NT 都能把长文件名自动转换成传统的 8.3 格式吗?是的,你猜对了。好了,既然前面所有的假设都是正确的,那么为什么我还是只采纳了八字符文件名呢?理由很简单,因为我对八字符文件名已经非常熟悉了,在把自己的思想转换到一个新的频道之前,我想也许还不得不花上一段过渡时间。

本书附带 CD-ROM 的设置模块把 WIN32BK.H 放置到了存放包容文件的第一个目录内。这是通过 Visual C++的注册表(Registry)来实现的。所以,本书的几乎每个例子都采用了下面这种语法格式:

```
...
#include <windows.h>
#include <win32bk.h>
...
```

CD 的 Listing 2.1 内存放了 WIN32BK.H。我定义了一些有用的宏、一些新的数据类型以及一些窗口风格标志。

假如你想增加某些定义或者对开发环境提供的标准组件进行某些变动,也可以像我这样

建立一个新的包容文件，从而避免改动 WINDOWS.H 和其他文件。

2.5 INCLUDE 示例

在本书附带 CD 的 Listing 2.2 里，大家可以找到一个名为 INCLUDE 的例子，这是一个相当复杂的 Win32 应用程序。对话框、通用控件以及菜单只是 INCLUDE 里提供的一些开发组件，本书没有对它们进行分析。现在不管这些，让我们一起先来看看这个 INCLUDE 程序的工作原理。

INCLUDE 在屏幕上是以一个空窗口的样子显示出来的。在标题栏的下方有一个菜单栏，其中两个顶级菜单分别是 File 和 Edit。与 File 菜单联系在一起的弹出式菜单向我们展示了三个菜单项：Browse，Statistics 和 Exit，如图 2-7 所示。

图 2-7　与 File 顶级菜单联系在一起的弹出式菜单

其中，Browse 是我们能够选择的唯一菜单项，这是由于 Statistics（统计）项以灰色显示，而 Exit 显然是用于中断应用程序的执行。正如大家在第 6 章“菜单的运用”里将看到的那样，对于所有菜单项来说，只要在其名称后有一个省略号，就表明选中这个菜单项后，屏幕中将出现一个对话框。INCLUDE 也遵从了这个约定的规则，并且显示出了如图 2-8 所示的 Browser 对话框。

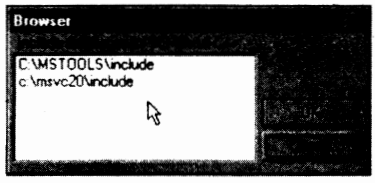


图 2-8　Browse 对话框列出了包含头文件并且被 Visual C++显示的所有系统目录

图 2-9 内的三个条目分别是与开发环境的最新版本（\MSTOOLS\H）和 Visual C++编译程序联系在一起的头文件（\MSVC\INCLUDE）以及 SDK 的一个预发行版本（\SDK\INC32）有关的。INCLUDE 怎么会显示出这三个条目呢？它们是对 HKEY_CURRENT_USER 键下面的注册表数据库进行浏览的的结果。更准确地说，Visual C++在 Software\\Microsoft\\Developer 键里存储了许多有用的信息，如图 2-9 所示。

Browser 列表框内的项目数量并不是固定的，根据系统的不同，显示出来的项目也会有所不同。RegOpenKeyEx（）这个注册表 API 函数可以帮助我们完成所有的工作。这个函数在应用程序里将被调用两次——第一次是检查是否存在编译程序，第二次则是取回需要的信息。

```
...
// is the Developer environment installed?
```

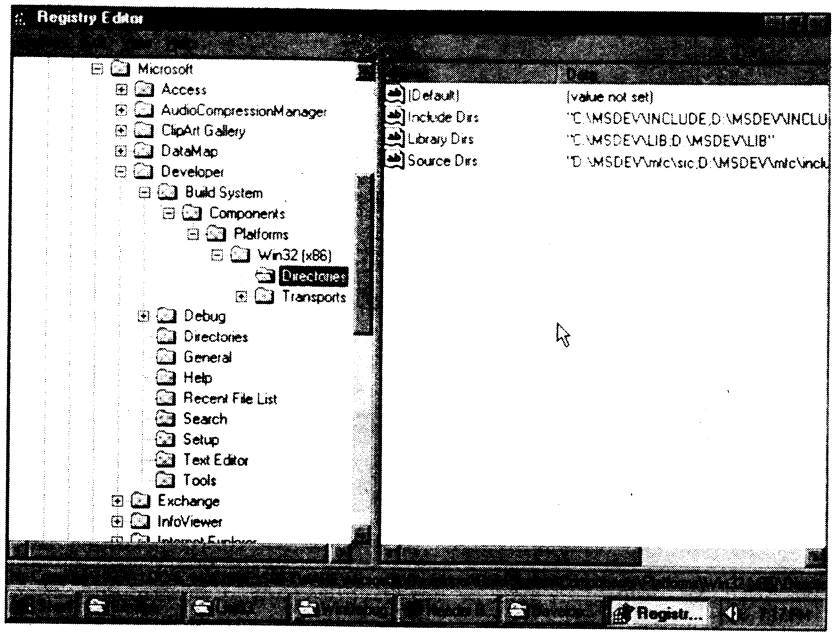


图 2-9　与 Visual C++相关的注册表树

```
if (RegOpenKeyEx (HKEY_CURRENT_USER,
                  " Software\\Microsoft\\Developer",
                  0,
                  KEY_ALL_ACCESS,
                  &VCHKey) ! = ERROR_SUCCESS)
{
        // couldn't locate VC++
        MessageBox (hwnd, " Could not locate MS Developer Environment in the
                   Registry"," Include dirs not found", MB_OK);
        // dismiss the dialog
        EndDialog (hwnd, FALSE);
        break;
}

// access the registry
if (RegOpenKeyEx (KEY_CURRENT_USER,
" Software\\Microsoft\\Developer\\Buildsystem\\Components\\Platforms\\Win32
                  (x86) \\Directories",
                  0,
                  KEY_ALL_ACCESS,
```

```
                    &VCHKey) ! = ERROR_SUCCESS)
{
        MessageBeep (0);
}
...
```

就我个人的角度来说，我并不同意微软在这种特定的环境下采纳这种策略。我指的是在主控键名内使用了版本号。正如读者在图 2-9 里看到的那样，所有信息看起来与版本 2.0 有关；但在我的系统上运行的却是版本 2.1。所以我强烈建议大家不要在注册表数据库的主控键名条目里放置版本号。

假如微软发布了开发环境的另外一个重要的版本，那时会发生什么情况呢？由于加入了一个新的条目和它的分支，所以注册表里就会变得混乱不堪，这样会给我们的视觉带来不良影响。除此以外，诸如 INCLUDE 的应用程序就会花更多的时间在注册表里寻找正确的位置，因为它与版本号是严格关联起来的。

假如对 RegOpenKeyEx () 的第一次调用就返回了一个 ERROR_SUCCESS，就会转向第二次调用，对 Software\Microsoft\Developer\Buildsystem\Components\Platforms\Win32 (x86) \Directories 键进行搜索。这是 Visual C++存储特定信息的地方，这些信息如图 2-10 的 Options 对话框所示。

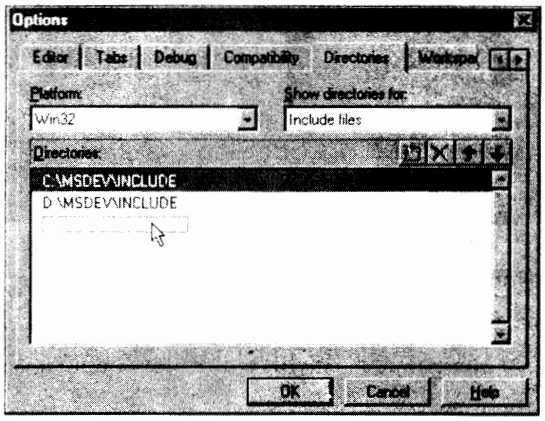


图 2-10　Visual C++的 Options 属性表，其中已选中了 Directories 卡片

一旦打开了选中的键，就可以在列表框内列出其中的内容，把对应的信息以恰当的方式显示出来。RegEnumValue () 函数可以帮助我们完成这一工作，它会把缓冲区内的第一个字符串存储到 BROWSEDATA 数据结构内——这是在 INCLUDE 例子里声明的一种新的数据类型。

```
...
// retrieving information
if (RegEnumValue ( VCHKey,
```

```
                    i,                    // include dirs
                    szValue,
                    &dwSize,
                    NULL,
                    &dwType,
                    pbrd -> szInclude,
                    &dwSizeData) ! = ERROR_SUCCESS)
{
    MessageBeep (0);
}
...
```

如果想对这种处理进行限制，我们可以分列出单个的正文串，然后把信息显示在列表框内，就像图 2-12 那样。选择第一个条目：C：\MSTOOLS\H。瞬间对话框消失了，在我的 Pentium 90 机器上，一秒钟的时间都没有用到，应用程序的客户区内就显示出了 WINDOWS.H 文件名，如图 2-11 所示。

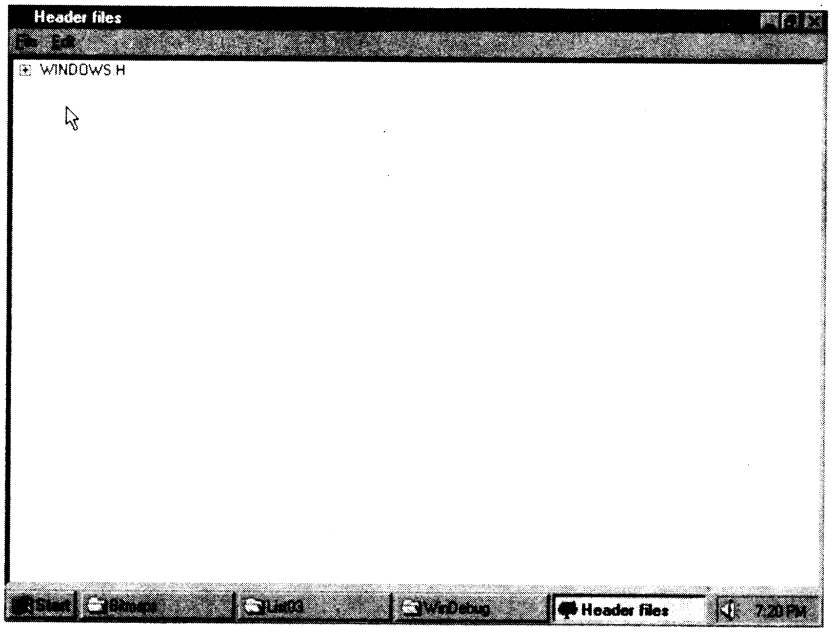


图 2-11　选择了由 Visual C++列出的其中一个目录后，INCLUDE 生成了一个树形视窗，其中列出了所有包容文件

左侧的那个小位图(里面有个加号)在 Windows 95 里是很常见的。我用一个 TREEVIEW 视窗（即树形视窗）来覆盖了应用程序客户区域。带有加号的位图表明还有很多其他的信息与 WINDOWS.H 关联在一起，只是在分级结构里，这些信息的级别要低一些。换句话说，WINDOWS.H 包含了其他许多头文件。想看到它们吗？只需单击左侧的小图标，整个树形结

构就展现在眼前了，如图 2-12 所示。

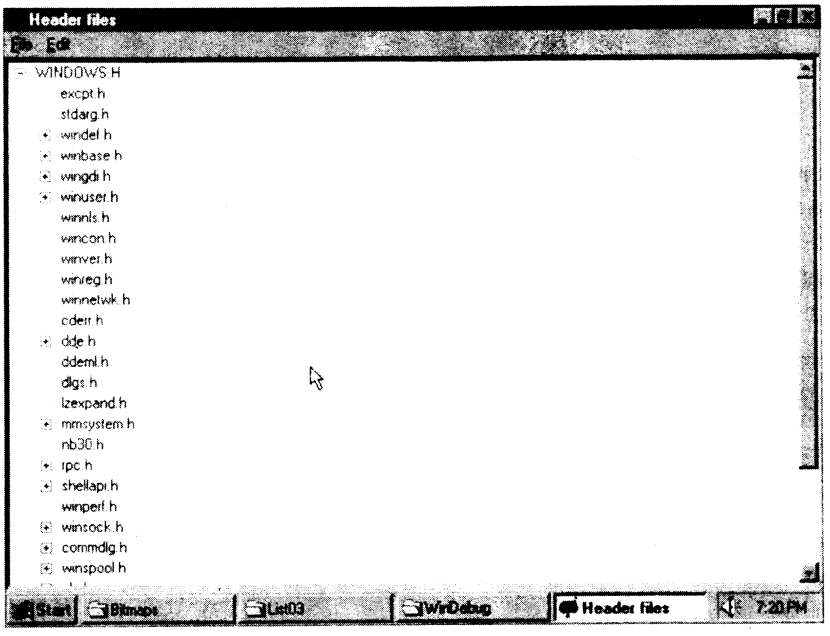


图 2-12　INCLUDE 显示出了 WINDOWS.H 里包含的所有头文件，注意 WINDOWS.H 位于这个分级结构的顶部

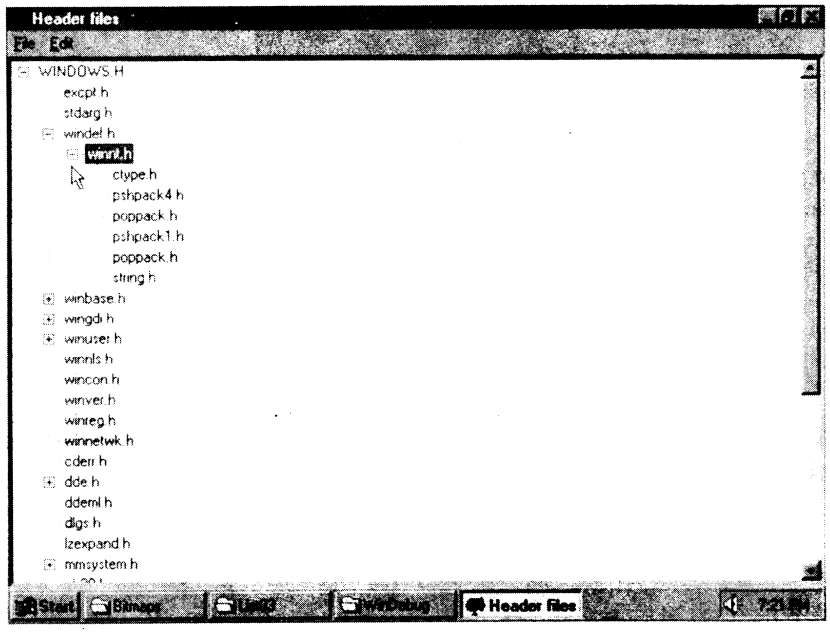


图 2-13　WINDEF.H 包含了一个头文件：WINNT.H

我们有必要在 WINDEF.H 节点图标的上方重复上一次操作，看看这个头文件里包含了

什么信息。

假如双击一个头文件，INCLUDE 就会启动系统编辑器 WORDPAD，然后把头文件载入。如图2-14所示。

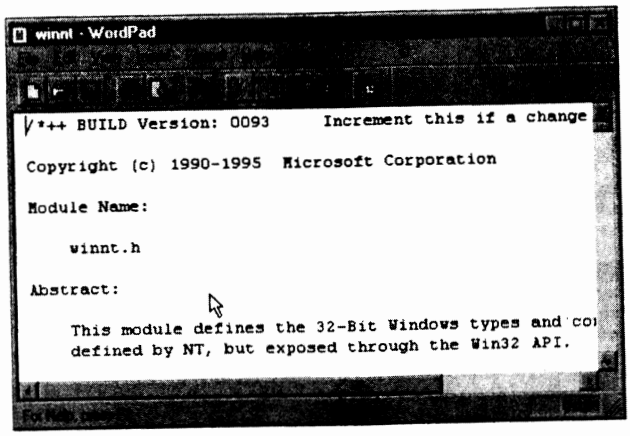


图 2-14 如果双击树形结构内的同一个节点，就会打开 WINNT.H

列出的一些头文件并没有真正存储于通过 Browser 对话框选择的目录里。由于这个原因，INCLUDE 有时会显示一条警告消息，提醒用户注意这个问题。如图 2-15 所示。

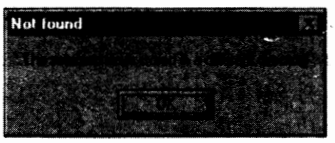


图 2-15 如果某个文件在文件系统里的存储位置和在 Browser 对话框里选择的位置不一致，此时打开该文件就会看到一个出错消息框

假如用鼠标右键单击一个 TREEVIEW 项目，就会发生两件事情，注意这两件事情是一个接一个发生的。首先，单击的项目将被选中（对于 TREEVIEW 来说，缺省状态下是没有项目被选中的）。随后，单击的屏幕位置处将显示出一个小的弹出式菜单，如图 2-16 所示。返回的信息并不是那么详尽的，尽管它对文件的位置和长度进行了说明。

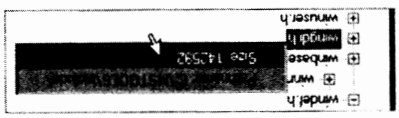


图 2-16 每个项目都有一个对应的弹出式菜单，其中列出了文件的路径名和长度

现在让我们来看看 RPC.H 节点。看到了什么呢？RPC.H 里包含了 WINDOWS.H。有什么不对吗？其实这是完全正常的。这个细节使整个应用程序的设计变得复杂了。通过不断展开 RPC.H，我们会发现 WINDOWS.H 进入了一种递归循环。这种状况什么时候会终止呢？直到消耗完了为所有文件名和一些附加信息提供的内存区域为止。如图 2-17 所示。

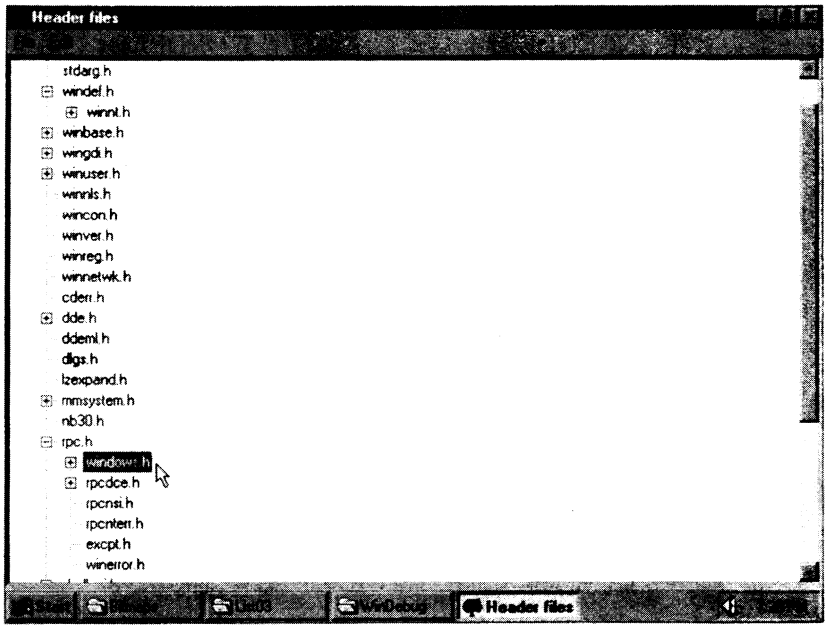


图 2-17　WINDOWS. H 包含在 RPC. H 里，RPC. H 又包含在 WINDOWS. H 里——一个典型的死循环错误

应用程序启动以后，线性地址空间里就会先预约 20 个内存页，同时只对其中的第一个进行委托。事实上，假如应用程序试图把信息存储到一个没有委托的页内，就会调用一个异常处理例程来解决这个问题。反复选择 RPC. H 和 WINDOWS. H 要求对另外一个页进行委托。INCLUDE 会控制这种负荷，一次只委托一个页，而且只有在特别有必要的情况下才会这样做。我没有在程序中实施页保护策略，尽管这种保护算法是以同样的技术为基础的。

现在让我们来看看 INCLUDE 里实现的另外一个特性。顶级菜单 Edit 里提供了一个 Find 菜单项。假如选中这个项，Find 就会显示出一个对话框，如图 2-18 所示。

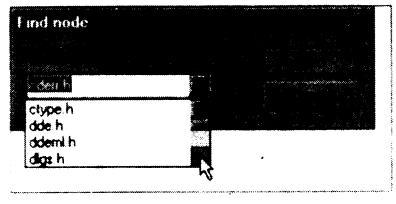


图 2-18　Find.node 对话框允许用户搜索树形结构内的一个 INCLUDE 文件

Available node（可用节点）组合框按字母顺序列出了当前在 INCLUDE 的 TREEVIEW 部分里的所有节点。选定其中一个节点，然后按下 Find 按钮，该对话框就会消失，然后在 TREEVIEW 窗口里选定对应的节点。

现在是结束这段 INCLUDE 使用指导的时候了，我们可以在 File 弹出式菜单内选择 Statistics 菜单项，如图 2-19 所示。

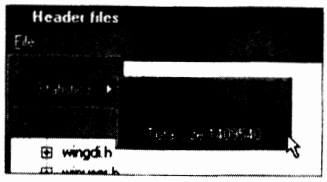


图 2-19　Statistics 菜单项提供了与系统头文件有关的一些信息

列出的统计信息涉及到节点的数目、项目总数以及 TREEVIEW 窗口里所有 INCLUDE 文件的总长度。

2.6　一段简单的 C 教程

高度浓缩的 C 语言提供了符合 ANSI 标准的 32 个关键字，它在运算符和关键字的组合上面具有相当高的灵活性。由于用 C 编写的程序难于辨读，所以经常招致批评家的责难。为了限制这种负面影响，所有 Windows 源代码都应该按照一种名为 Hungarian Notation（匈牙利记号法）的标准进行编写。这种规范是根据发明者居住的国家而命名的。它为 C 标识符的命名定义了一种非常标准化的方式，这种命名方式是以两条简单的规则为基础的。首先，标识符的名字要以一个或者多个小写字母开头，用这些字母来指定数据类型。表 2-11 列出了用于对整数、字符串和布尔值等进行定义的一些标准前缀。

表 2-11　在 Windows 里定义数据类型的一些标准前缀

前　缀	数据类型
c	字符（char）
s	短数（short）
cb	用于定义对象（典型情况下是一个结构）尺寸的整数
n	整数（integer）
sz	ASCIIZ 串
b	字节（BYTE，无符号字符）
i	int（整数）
x	短数（坐标 x）
y	短数（坐标 y）
f	BOOL
h	句柄（handle）
w	字（WORD，无符号短数）
dw	双字（DWORD，无符号长数）
l	长数（long）
h	HANDLE（无符号 int）
fn	函数（function）
p	指针（pointer）

在一个标识符内，前缀以后就是一个或者多个大写的单词，这些单词清楚地指出了源代码内那个对象的本质和用途。举个例子来说，假设你现在需要把一个整数标识符用作循环计数器。根据 HungarianNotation 规范，一个正确的名字可能是 iCounter 或者 iCntr。在这两个名字里，i 前缀都表明这是一个整数（int）类型的标识符。Counter 则表明这个标识符很有可能是当作计数器来使用的。

第二种形式是 iCntr，它剔除了所有元音字母，从而使整个名字的长度缩短了。另外再举一个例子，假设你需要一个窗口句柄来指定某个父窗口，这时可能采用的一个标识符就是 hwndPapa。它表明该标识符是一个窗口（wnd）使用的句柄（h），同时指定的是一个父窗口（Papa）。这种语法结构已由微软公司采纳，这在它的 API 名字里得到了很突出的表现，例如：

```
CreateWindow ()
ShowWindow ()
MoveWindow ()
PostQuitMessage ()
DestroyWindow ()
GetWindowLong ()
CallWindowProc ()
GetDlgItemText ()
```

尽管这些函数的名字都是用不同的大写单词组合起来的，而且这些单词也明确地表明了该函数的作用，但是它们都没有使用前缀。这种对 Windows 程序进行编码的方法已经广为流行开了，甚至许多标识符都有一个标准的名字，而不管谁是一条代码的发明者。例如，WinMain() 函数——Win32 应用程序的入口点的四个参数总是叫作 hInstance，hPrevInstance，szCmdLine 和 nCmdShow。

本书的所有例子都是遵从 Hungarian Notation 规范的，唯一的区别在于我没有打算对标识符的名字进行大写处理（一种轻微的个人化处理）。

#define 命令

Win32 API 设置了数目众多的风格和定义，这些设置都要归功于预处理器 #define。一个定义的语法结构如下所示：

```
#define synonym definition
```

其中，definition（定义）可以是一个数字、一个表达式、一段代码或者另外一个 synonym（同义字）。“同义字”是一个用大写字母表示的正文串，只要在这儿进行了定义，该同义字以后就会与对应的“定义”等效，除非用 #undef 预处理器命令对其进行了改变。请大家看看下面这个例子：

```
#define SECONDS_PER_DAY (60 * 60 * 24)
```

在上面这个例子中，SECONDS_PER_DAY 串完全等效于表达式 (60 * 60 * 24)。在源代码的任何地方，使用 SECONDS_PER_DAY 串显然能让人更容易理解它的含义。在编译过程中，预处理器会在生成 .OBJ 文件之前用数值表达式取代这个正文串。

使用 #define 命令时，一个很常见的错误是在定义宏的时候忘记用括号把占位符括起来。下面是一个典型的例子：

```
#define SQUARE (x) x * x
```

其中，SQUARE 宏的作用是让 x 占位符乘以它本身。像 SQUARE 这样的宏一般都是按照下面这种语法运用的：

```
...
```

```
nVal = SQUARE (3);
...
```

它返回的是一个正确的值：9。但是，并没有什么限制不允许我们传递一个更详尽的占位符，比如下面这个数值表达式：

```
...
nVal = SQUARE (nBase + 1);
...
```

在编译过程中，编译程序会用数值来替换SQUARE，最后的结果看起来就像下面这个样子：

```
...
nVal = nBase + 1 * nBase +1;
...
```

现在假设 nBase 的值为 2。这就意味着输出将是 5 (2+1 * 2+1)，并不是预期的 9。为了避免出现这种错误，应该随时记住用括号把所有的占位符都括起来。所以，SQUARE 的正确定义应该是下面这样的：

```
#define SQUARE (x) ( (x) * (x))
```

WINDEF.H 里包含了数十种非常有用的宏，它们能从 32 位值里提取信息，或者通过两条或者多条信息建立一个 32 位值。表 2-12 列出了其中一些常用的宏：

表 2-12 Win32 头文件里定义的一些宏

宏	定 义
#define MAKEWORD (a, b)	((WORD) (((BYTE) (a)) | ((WORD) ((BYTE) (b))) ‹ ‹8))
#define MAKELONG (a, b)	((LONG) (((WORD) (a)) | ((DWORD) ((WORD) (b))) ‹ ‹16))
#define LOWORD (I)	((WORD) (I))
#define HIWORD (I)	((WORD) (((DWORD) (I)›› 16) &0xFFFF))
#define LOBYTE (w)	((BYTE) (w))
#define HIBYTE (w)	((BYTE) (((WORD) (w)›› 8) &0xFF))

除此以外，Windows 编程还涉及到头文件内声明的新数据类型的使用。要把所有这些数据类型都列出来几乎是不可能的。在表 2-13 里，我收集了最常用的一些数据类型。

表 2-13 WINDEF.H 和 WINNT.H 里定义的经常用到的简单数据类型

typedef unsigned long ULONG;	typedef char CHAR;
typedef ULONG *PULONG;	typedef short SHORT;
typedef unsigned short USHORT;	typedef long LONG;
typedef USHORT *PUSHORT;	typedef wchar_t WCHAR;
typedef unsigned char UCHAR;	typedef WCHAR *PWCHAR;
typedef UCHAR *PUCHAR;	typedef WCHAR *LPWCH, *PWCH;
typedef char *PSZ;	typedef CONST WCHAR *LPCWCH, *PCWCH;
typedef UINT WPARAM;	typedef WCHAR *NWPSTR;
typedef LONG LPARAM;	typedef WCHAR *LPWSTR, *PWSTR

（续）

typedef unsigned long ULONG;	typedef char CHAR;
typedef LONG LRESULT;	typedef CONST WCHAR * LPCWSTR, * PCWSTR;
typedef CHAR * PCHAR;	typedef LPWSTR PTSTR, LPTSTR
typedef CHAR * LPCH, * PCH;	typedef LPCWSTR LPCTSTR;
typedef CONST CHAR * LPCCH, * PCCH;	typedef LPWSTR LP;
typedef CHAR * NPSTR;	typedef char TCHAR, * PTCHAR;
typedef CHAR * LPSTR, * PSTR;	typedef unsigned char TBYTE, * PTBYTE;
typedef CONST CHAR * LPCSTR, * PCSTR;	typedef LPSTR LPTCH, PTCH;
typedef WCHAR TCHAR, * PTCHAR;	typedef LPSTR PTSTR, LPTSTR;
typedef WCHAR TBYTE, * PTBYTE;	typedef LPCSTR LPCTSTR;
typedef LPWSTR LPTCH, PTCH;	

下面是一些集合数据（aggregate data）类型列表。

```
typedef struct tagRECT
{
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
} RECT, * PRECT, NEAR * NPRECT, FAR * LPRECT;
typedef const RECT FAR * LPCRECT;
typedef struct tagPOINT
{
    LONG x;
    LONG y;
} POINT, * PPOINT, NEAR * NPPOINT, FAR * LPPOINT;
typedef struct tagSIZE
{
    LONG      cx;
    LONG      cy;
} SIZE, * PSIZE, * LPSIZE;
typedef struct tagPOINTS
 {
    SHORT x;
    SHORT y;
} POINTS, * PPOINTS, * LPPOINTS;
```

2.7 关于句柄

句柄是 Windows 编程的一个关键性的概念。在 Win32 里，句柄就是指向一个“空白”

(void *) 的指针，换句话说就是一个 **4** 字节长的数据。无论它的本质是什么，句柄都不是一个真正意义上的指针。从构造来看，句柄是一个指针，尽管它没有指向用于存储某个对象的内存位置。事实上，句柄指向一个包含了对那个对象进行的引用的位置。句柄数据类型是在 WINNT.H 里声明的，如下所示：

```
typedef void *HANDLE
```

句柄几乎肯定是由一些 API 调用返回的。CreateWindowEx () 这个基本 API 的用途是建立一个标准的窗口，然后返回窗口句柄。它到底是什么呢？从 C 语言的角度来看，它是指向某个特定内存位置的指针。系统并没有在那个内存地址处存储实际的对象数据。相反，一个简单的数字（对真实对象的引用）存储于 RAM 的另外某个地方。采用窗口句柄的方式，应用程序可以把屏幕上的一个窗口与内存中的窗口合并起来。**Win32** 也利用句柄访问其他系统组件，比如位图、字体、元文件、图标、光标和刷子（brush）等等。

对句柄进行运用的时候，允许开发者进行的唯一操作就是检查是否存在一个有效的句柄。假如不是一个零值，就表明这个句柄是有效的；如果等于零，这个句柄就是无效的。考虑到句柄是由某些 API 调用返回的，所以假如函数调用成功地完成了，就可以通过这种迂回方式证明该句柄是有效的。

基于前面的假设，在一个分配操作里，假如 API 调用是以 rvalue 的形式实现的，句柄的形式就应该是 lvalues：

```
...
hwnd = CreateWindow (...);
...
if (! hwnd)
{
    MessageBeep (0);
    return FALSE;
}
...
```

假如返回值无效 (0x00000000)，这个测试就会失败。在上面这个代码段里，应用程序会发出一声响铃信号，然后返回调用例程。为了明确使一个句柄变得无效（这是一种很不常用的操作），把它设置成 NULL 就足够了：

```
hwnd = NULL;
```

把一个句柄变得无效以后，我们只是移去了应用程序对对象的引用，并没有真正地破坏它。在窗口示例中，我们可以通过调用 DestroyWindow () 这个 API 函数来破坏（删除）一个窗口。在这个函数破坏了与那个窗口对应的内存块后，相应的句柄就无效了。

表 2-14 为大家总结了 Win32 支持的所有句柄数据类型。

表 2-14　Win32 支持的句柄类型

HCONVLIST	HPCM	HDC
HCONV	HPENDATA	HGLRC
HSZ	HREC	HDESK
HDDEDATA	HRC	HENHMETAFILE
HDRVR	HRCRESULT	HFONT
HWAVE	HWL	HICON
HWAVEIN	HRECHOOK	HMENU
HWAVEOUT	HINKSET	HMETAFILE
HMIDI	HRASCONN	HINSTANCE
HMIDIIN	HDROP	HPALETTE
HMIDIOUT	HVOLUMEID	HPEN
HMIDISTRM	HIC	HRGN
HMIXEROBJ	HVIDEO	HRSRC
HMIXER	HWND	HSTR
HMMIO	HHOOK	HTASK
HACMDRIVERID	HGDIOBJ	HWINSTA
HACMDRIVER	HACCEL	HKL
HACMSTREAM	HBITMAP	HKEY
HACMOBJ	HBRUSH	
HTRG	HCOLORSPACE	

第 3 章　Win32 应用程序的开发

我们现在可以着手分析一个典型 Win32 应用程序的结构了。在这一章中，读者将和我一起开发 Welcome 示例，这是一个相当简单的 Win32 程序，它只支持一个主窗口。下面，我们最好先从学习应用程序入口点的整体布局开始。

3.1　检查 Win32 的一个入口

每个 C 应用程序都带有一个 main () 函数，它就是代码的入口点。在大多数情况下，main () 没有附加任何参数。所以，根据 ANSI 规范，它看起来就像下面这个样子：

```
int main (void)
{
    ...
}
```

这个函数不接收任何变量，也不会返回一个整数值。为了访问我们在命令行键入的信息，必须对 main () 进行修改，方法是在其中插入至少两个参数，一个是整数 (int)，另一个是指向某个字符 (char) 的指针。整数值用于计算命令行内键入了多少个参数，后面则跟随一系列正文串的集合。每个正文串都位于 char * * 对象内一个不同的索引里。

当 Windows 于 1985 年首次发布的时候，它用 WinMain () 取代了标准的 C 入口点。WinMain () 函数的特点是它总包含了四个参数：

```
...
int WINAPI WinMain (HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nShowCmd)
{
    ...
}
...
```

WinMain () 是一个 WINAPI 函数 (即_stdcall)，该函数返回的是一个整数，并且可以接收具有三种不同类型的四个参数：HINSTANCE，HINSTANCE，LPSTR 和 int。对于最后一种类型来说，我们不必再多作说明了，因为它是一种标准的 C 关键字。头两个参数是在 WINDEF.H 里定义的。HINSTANCE 数据类型定义了一个特殊的句柄，这个句柄里包含了虚拟地址空间内的进程载入地址。LPSTR 参数是指向一个字符 (char *) 的指针。在第 2 章的表 2-14 里，我们对 Win32 采用的所有句柄数据类型进行了总结。

尽管 WinMain () 的语法在 Win16 和 Win32 里是完全一致的，但是两者之间内在的区别

却是引人注目的。在 Win16 里，头两个参数（hInstance 和 hPrevInstance）分别定义了应用程序自动数据段（也被称为 DGROUP）的句柄以及前一个执行的实例（Instance）的数据段的句柄。考虑到几乎所有 Win16 应用程序都有它们自己的数据段（用 DEF 文件内的 MULTIPLE 命令定义），所以 hInstance 扮演了识别元素的角色，它的作用是把每个任务与系统执行程序内的其他任务区别开来。这和 Win32 里的情况是不同的，在 Win32 里，进程不会用 LDT 描述表对内存进行寻址和访问，所以不是共享的单一的地址空间。更准确地说，在 Win32 里，唯一的偏移地址提供了访问内存位置所需的一切信息，这种寻址方案是通过绕开内存分段的分页方案来实现的。Win95 系统装载程序会把 hInstance 放置于进程载入地址，这个地址和它的专用虚拟地址空间是相关的（句柄的本质毕竟还是指针）。所以，典型的 Win16 数据段引用在 Win32 里已经消失了，因为每个进程都在一个独立的地址空间内运行，与其他地址空间之间不存在自然的通信链接。

在 MS Windows 95 里，每个进程的 hInstance 都设置成 0x00400000，这样就失去了它在 Win16 里那种典型的“唯一”特征。这样造成的结果就是，通过使用 Win32 里的 hInstance，我们不能区分出相同进程内的两个或者多个实例。这样一来，根据前面的假设，Win32 里的 hPrevInstance 就具有了完全不同的含义。我们能对同一个程序执行多重拷贝，尽管每个拷贝都是以独立姿态出现的。正如大家在本章末尾要学到的那样，每个进程都提供了一个不同的识别编号，我们把这种编号叫作 Process ID（进程识别号），或者 PID。Win32 是通过 PID 对进程进行区分的，而不是通过 hInstance。

hInstance 是 WinMain () 里唯一有用的参数。lpCmdLine 很少指向带有应用程序关键信息的一个串。hPrevInstance 的值总为 NULL，而 nShowCmd 则基本上是没有什么意义的。所有这些参数都将在下文进行详细的介绍。

3.1.1 Win32 的 hPrevInstance 参数

Win32 总是把 hPrevInstance 设置成 NULL，不管当前是否正在运行同一程序的其他拷贝。只需想到每个进程都有它自己独立的虚拟地址空间，就不应该再对这种现象感到惊奇了。即使是一个 Win32 进程的两个实例，这两个实例都必须依赖进程间通信（IPC）工具，否则便无法实现信息的交换和某些内存区域的共享。系统会把数据段和代码段都映射到物理地址空间的任意位置处。这个规则对于每种 Win32 进程都是有效的。微软的系统工程师决定在 0x00400000 这个内存位置处（4MB）载入一个进程，从而避免由于空指针（理论上有效的）而导致系统错误，以及解决由老式 16 位代码带来的兼容性问题。这种处理方式在 API 级别上产生了某些影响。GetModuleFileName () 现在只有在获取当前进程名的时候才能正常工作，不能用它返回不同进程的名字，因为 hInstance 对于每个进程来说都具有相同的值。除此以外，在 Win32 里，hInstance 和 hModule 之间已经没有什么区别了。后者是在新窗口类注册过程中由 Windows 生成的一种信息。在必要的时候，hInstance 和 hModule 是可以替换使用的，因为它们都指向相同的数据。

Win16 的 hPrevInstance 参数

在 Win16 里，hPrevInstance 既可用于指定应用程序最后一个实例的数据段，假如当前进程是正在运行的唯一实例，也可以把它设置成 NULL。然而，hPrevInstance 并不是计算正在运行的实例总数的正确途径。为了达到对运行实例总数进行计算的目的，必须用到一些列举 API 函数（在 Win32 里，一个信号机就是我们正确的选择）。对 hPrevInstance 进行测试的时

候，假如前一个实例现在正在运行，系统就会通知我们这种情况。除此以外，在 Windows 的第一个版本里，利用 hPrevInstance 的帮助，Windows 应用程序可以通过 GetInstanceData () 读取来自前一个实例数据段的数据。大家应该注意到这种技术的一个限制，那就是它只能在源代码的范围内进行定义。就目前来说，这种技术几乎已经变成了史前动物，很少有人用它，而且 GetInstanceData () 函数也已经取消了。

3.1.2 lpCmdLind 参数

尽管这看起来有些奇妙，但是 Win32 进程确实可以通过命令行语法与用户进行交流。在一段简单的 C 代码程序内，假如 main () 函数采用下面这种格式，便可实现对命令行的拦截：

```
int main (int argc, char * *argv, char * *envp)
{
    ...
}
```

在 Win32 里，系统命令行内出现的信息是用 WinMain () 的第三个参数传递的。除了用鼠标双击方式激活一个新的进程以外，Win95 还提供了一些有价值的选择方案，比如 Start 弹出式菜单内的 Run 菜单项，以及 MS-DOS 区内的 Start 命令。举个例子来说，如果准备在一个 MS-DOS 区内启动 WordPad 程序，只需在命令行内键入：c:\windows\wordpad 即可。如果想对这条命令的功能进行延展，我们可以在启动的时候同时载入一个文档，比如：c:\windows\wordpad steve.doc。其中，正文串 steve.doc 会在 WinMain () 的 lpcmdLine 参数内显示出来。

GetCommandLine () 是读取整个命令行的另外一种方法：

```
#include〈winbase.h〉
LPTSTR GetCommandLine (VOID);
```

如果想启动本章介绍的 Welcome 例子，可以在 Run 编辑控件的程序名后键入：Stefano Maruzzi。在 WinMain () 里，使用 lpCmdLine 参数，并且用一个指针指向包含了 GetCommandLine () 返回地址的一个字符。

```
...
char *p;

p = GetCommandLine ();
...
```

在 lpszCmdLine 参数里包含了 “Stefano Maruzzi”，而 p 指针则指向包含了路径名 C:\Win32bk\win32_01\LIST01\WinDebug\platform.exe 的整个命令行。我猜想 C 程序员们更愿意采用第二种方法。

3.2 nShowCmd 参数

WinMain () 的第四个参数是整数 nShowCmd。我想强调的是这个参数在 Win32 里几乎是没有什么用处的。该参数只有在 MS Windows 1.x 里才能找到用武之地。在那个版本的

Windows 里，系统本身将对窗口进行自动平铺处理（叠置窗口在 1987 年末发行的 2.x 版本里才正式出现）。即使是现在，如果把 nShowCmd 当作 ShowWindow () 的第二个参数进行传递，还是可以显示出主窗口来的。但是，要注意的一点是 nShowCmd 的使用只有一次成功的机会。我个人的建议是尽量使用 ShowWindow () 的某个 SW_标志，坚决避免使用 nShowCmd。

WINNT.H 内包含了 UNREFERENCED_PARAMETER () 宏，用它可以对没有使用的参数进行封装，从而避免在编译阶段出现警告信息。根据这种思路，WinMain () 看起来就应像下面这样：

```
int WINAPI WinMain (HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nShowCmd)
{
    ...
    UNREFERENCED_PARAMETER (hPrevInstance);
    UNREFERENCED_PARAMETER (lpCmdLine);
    UNREFERENCED_PARAMETER (nShowCmd);
    ...
}
```

不要忘记在每个 UNREFERENCED_PARAMETER () 宏的末尾习惯性地放置一个分号。

3.3 窗口类的注册

通常，Win32 应用程序在屏幕中都是以窗口的形态出现的。当然，这并不是一种强制性的要求。本章这个 Welcome 例子的目的是要用一个主窗口建立一个基本的 Windows 应用程序。

形形色色的 Win32 API 让我们可以选择多种方案去建立窗口。不管最终的选择是什么，建立窗口之前，都有必要先注册一个窗口类。注册的过程是相当简单和直接的。你需要做的全部事情就是声明一个 WNDCLASS 或者 WNDCLASSEX（后者特别适用于 Win95）。一旦用恰当的值填充了结构项以后，再调用一次 RegisterClass () 或者 RegisterClassEx ()，这样就完成了注册过程。

和 Win16 类似，Win32 也提供了一套预先定义好的窗口类。如表 3-1 所示，一共有七个窗口类。在 Win95 的启动阶段，对这些“预定义窗口类”的注册就已经完成了。所以，它们是以微软公司提供的一些软件组件为基础的。

对预注册类的详细分析将在第八章“Win32 的对话框管理”里进行。在表 3-1 列出的窗口类中，前面六个通常可以叫作“控件”(control)，而 MDICLIENT 类是在 1992 年随着 Windows 3.0 引入的。它简化了对 MDI 应用程序的开发（MDI 是“多文档界面”——Multiple Document Interface 的简称）。使用这七个现成类的其中之一，也许就能绕过一个新窗口类的注册，建立属于其中一种现成类的窗口。这些类非常专门化，它们的主要目的是解决常规 Windows 程序

内的特定 UI（用户界面）问题。从技术角度看，我们完全可以在这个 Welcome 项目里显示一个滚动条，把它当作主窗口来使用，但我不相信这样做有什么实际的意义！

从根本上说，每个专业的 Windows 应用程序都在启动阶段注册了某些窗口类，从而满足自己的 UI 和设计需求。我们一般不可能规定出准确的窗口类数目，因为它会根据进程的不同发生变化。通常，每个独立的程序使用四个或者五个类是比较正常的——同时使用几十个类的情况很少见。

我们对一个窗口类进行注册的时候，一些数据会传送给 Windows，Windows 随后会把这些数据存储到它自己的 USER 堆里。这个数据块的内容和尺寸是没有规定的，它们只能由调用 RegisterClassEx（）的 Win32 进程进行访问，如图 3-1 所示。假如 RegisterClassEx（）调用成功，新增的窗口类就会使最初的集合容量变大。

对于预先定义好的窗口类和通过应用程序代码注册的一个特定窗口类来说，两者之间是存在根本区别的。第一种是“永恒”的类。当然，“永恒”这个词在这里并不是百分之百的准确，因为永恒意味着既没有开始，也没有结束。但它对这种类的本质确实进行了很形象的概括——只要 Windows 保持运行状态，随时都可以使用它们。以预注册窗口类为基础，每个 Win32 进程都能够在任何时候建立任何窗口，同时还不用为它的注册操心。相反，对于通过调用 RegisterClassEx（）注册的类来说，它们只有在注册进程运行期间才能使用。一旦进程中止了，USER 堆内的那些内存区域就会删除和释放。假如某进程想建立一个特定类的窗口，它必须首先对那种类进行注册。

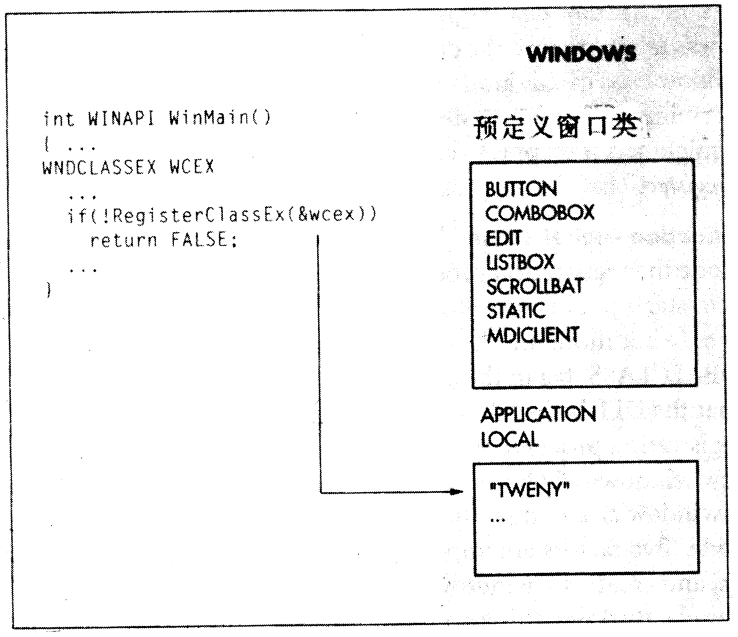


图 3-1　在 Win32 里，一个新窗口类的注册过程

Win32 提供了对三种窗口类的支持：

▶系统全局

▶应用程序专用

▶应用程序全局

下面将对这三种类进行简要的介绍。

1. 系统全局类（System Global Class）

这种类是由七种预先注册好的窗口类组成的。在开发者编写的代码里毋需进行任何注册。注意一般情况下既不能破坏它们，也不能删除它们。和 Win16 不一样，系统对这种信息的控制是以一种单进程概念为基础的。预先定义的所有窗口类都划归这一类。

2. 应用程序专用类（Application Local Class）

在 Win32 里，假如应用程序注册一个新的窗口类，就会在当前进程内生成一个应用程序专用类。这是在 Windows 编程中经常碰到的一种情况（注意这与 Win16 的使用方式不同——在 Win16 里，实例至少可以共享相同的窗口类注册代码）。进程中断以后，Windows 就会把数据类结构从它的内存区域里移去。这种窗口类只有从它注册后到进程中断期间才是活动的。UnregisterClass（）这个 API 函数允许开发者在认为有必要的时候取消对一个类的注册。不管它潜在的和明显的好处，事实上，UnregisterClass（）在应用程序的专用类中是很少调用的。

3. 应用程序全局类（Application Global Class）

如果想生成一个应用程序全局类，必须先对某个 DLL（动态链接库）内的注册进程进行编码，并在系统启动的时候把它载入。采用这种方式，注册的类对于系统的每个进程来说都是透明的。除此以外，这种类必须用 WNDCLASSEX 的 Style 项内的 CS_GLOBALCLASS 标志进行注册（NT 也要求在注册表数据库的某个特定的键内把 DLL 列出来）。

根据以后建立的是哪个新窗口，注册进程会把有关的信息传送给 Windows。很常见的一种作法是在 WinMain（）里直接注册需要用到的所有窗口类，并且把这种注册设计成源代码内的第一项操作。我们在这儿需要注意到两个重要的因素：注册一个窗口类以及建立一个窗口。为什么要注册一个新的窗口类呢？答案是很简单的。因为这样做才能使应用程序定义它自己独特的行为、外观和特征。启动一个新进程以后，它就会注册自己需要的窗口类，然后建立一个窗口。用户则会把这个窗口理解为应用程序本身。根据经验来看，每个程序至少需要一个类来决定应用程序主窗口的特性。

一旦注册好某个类后，接下来就可以在那个类的基础上建立起数百个不同的窗口，只需在特定的时刻轻微改变一下窗口的外观就可以了。这里需要注意一个重要概念：一个独立的窗口类既可由屏幕中的几十个窗口代表，亦可只由一个窗口代表。在下一章里，我们会总结出特定的方法，从而拟定新窗口类注册的准则。现在，我们只需要掌握注册进程中涉及的代码就可以了，如下文所示。

4. WNDCLASSEX 结构

WinMain（）内的代码被限制成了几条指令。大家使用这些指令的时候，注意不要滥用，只选用能注册一个或者多个窗口类、建立主窗口以及实现消息循环的几个指令就可以了。和 Win16 比较起来，指令的整体布局要简单得多，这是由于 hPrevInstance 的值总是 NULL。

```
int WINAPI WinMain (HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nShowCmd)
```

```
{
    ...
    WNDCLASSEX wcex;
    ...
    // WNDCLASSEX
    wcex.cbSize = Sizeof (wcex);
    wcex.lpszClassName = szClassName;
    wcex.hInstance = hInstance;
    wcex.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wcex.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
    wcex.hCursor = LoadCursor (NULL, IDC_ARROW);
    wcex.hbrBackground = GetStockObject (WHITE_BRUSH);
    wcex.lpszMenuName = NULL;
    wcex.cbClsExtra = 0;
    wcex.cbWndExtra = 0;
    wcex.style = CS_HREDRAW | CS_VREDRAM;
    wcex.lpfnWndProc = ClientWndProc;

    // class registration
    if (! RegisterClassEx (&wcex))
        return FALSE;
    ...
}
```

首先，大家必须说明一种类型为 WNDCLASSEX 的标识符，这是在 WINUSER.H 里定义的一种数据类型：

```
typedef struct tagWNDCLASSEXA
{
    UINT cbSize;
    UNIT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HINSTANCE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCSTR lpszMenuName;
```

```
    LPCSTR lpszClassName;
    HICON hIconSm;
} WNDCLASSEXA, *PWNDCLASSEXA, NEAR *NPWNDCLASSEXA,
    FAR *LPWNDCLASSEXA;
```

这种数据类型的最后一个 A 表明它只接受 ANSI 标准格式的字符串（每个字符占用一个字节），而不是 Unicode 格式（每个字符占用两个字节）。Windows 95 只支持 ANSI 格式，而 NT 则同时支持 ANSI 和 Unicode。所以，每个 Win32 API 都是以两种形式实现的，一个以 A 结束，对于 Unicode 版本来说，则以一个 W（取自于 Wide）结束。根据应用程序采用的字符格式，这两种形式的字符串都会映射成标准的名字。假如定义的是 Unicode，WNDCLASSEX 就会映射成这种数据类型的 UNIX 版本。否则，WINDCLASSEX 就支持 ANSI 格式的字符串。这个规则适用于 Win32 里的每种数据类型和 API。

```
#ifdef UNICODE
typedef WNDCLASSEXW WNDCLASSEX;
...
#else
typedef WNDCLASSEXA WNDCLASSEX;
...
#endif // UNICODE
```

WNDCLASSEX 的十二个项允许开发者指定常规的类行为和外观。其中，提供的某些信息是非常陈旧的，有些甚至是无用的。接下来，我们需要对这些项进行仔细的检查，具体的讨论如下所示。

5. UNIT cbSize 项

Win32 要求传递给 RegisterClassEx () 的所有 WNDCLASSEX 定义都能正常地工作。这个信息是指该扩展数据结构的尺寸。

6. LPCSTR lpszClassName 项

LPCSTR lpszClassName 定义了窗口类的名字。通常，这个指针需要用到一个特定的串，就像下面这样：

```
...
wcex.lpszClassName = " TWENY";
...
```

我强烈建议诸位不要在代码里直接使用字符串常数。不要声明字符数组，也不要声明指向一个字符的指针，应该试着用类名对其进行初始化，以后再把这种信息传递给需要类名的任何一个地方。

```
...
char szClassName [20] = " TWENY";
...
```

```
wcex.lpszClassName = szClassName;
...
```

以后，读者就可以修改这段代码，首先从与应用程序有关的资源文件里载入类名。

7. HINSTANCE hInstance 项

HINSTANCE hInstance 标识了对应的实例，这一类的窗口进程就包含于这个实例内。WinMain ()四个参数的第一个是分配给这个项的正确的值，尽管它已经不具备自己的一部分含义——因为 hInstance 在每个 Win32 进程里几乎都是一个常数值。

8. HICON hIcon 项

标准的窗口在标题栏左上方显示了一个图标，假如把它缩小成 Win95 的一个任务栏按钮，同样的图标也会在任务栏上显示出来。由 WNDCLASSEX 的 hIcon 项定义的图标是由属于这一类的每个窗口采纳的。对于这一项来说，它可以选用三种方案。假如你不想让一个图标在自己的窗口内显示出来，只需为 hIcon 赋一个 NULL 值就可以了。当然，这并不是一种很常见的作法，事实上我们也不建议这样做，尽管从技术角度来看是行得通的。

第二和第三种方案是以 LoadIcon () 这个 API 函数为基础的。

```
#include <winuser.h>
HICON WINAPI LoadIcon (HINSTANCE hInstance, LPCSTR lpIconName);
```

参数	说明
HINSTANCE hInstance	无论 hInstance 还是 NULL 都接收一个预先定义好的图标
LPCSTR lpIconName	图标定义标签
返回值	假如该函数工作正常，它就会返回图标句柄，否则返回 NULL
HICON	图标句柄，假如函数调用失败，则为 NULL

在历史上，图标是在资源文件里用一个正文串定义的，很少用一个数字来定义。后面那种情况需要通过 MAKEINTRESOURCE () 宏把数字转换成一个串。

```
...
HICON hicon;
...
hicon = LoadIcon (hInstance, " appicon");
...
hicon = LoadIcon (hInstance, MAKEINTRESOURCE (ID_ICON);
...
```

假如不想使用一个具有个人特征的图标，也可以在 Windows 存储好的一系列预定义图象里选择一种。WINUSER.H 里包含了这些预定义图标的定义 (表 3-2)，不管是否存在与当前进程有关的资源文件，这些图标都是随时可用的。

表 3-2 预定义图标

定 义	值	说 明
IDI_APPLICATION	32512	MS Windows 徽标
IDI_HAND	32513	“无访问”符号
IDI_QUESTION	32514	蓝色圆上的一个白色问号
IDI_EXCLAMATION	32515	黄色圆上的黑色感叹号
IDI_ASTERISK	32516	信息点
IDI_WINLOGO	32517	MS Windows 徽标

```
...
hicon = LoadIcon (NULL, IDI_APPLICATION);
...
```

在这种情况下，实例句柄就变成了 NULL，第二个参数则变成了一种预先定义好的图标。

9. HICON hIconSm 项

除了标准的 32×32 象素格式以外，Windows 95 还提供了对不同形状的图标的支持。hIconSm 项指定了在系统文件夹里用详细视窗显示于屏幕上的一个 16×16 图标。

尽管 LoadIcon () 在 Win95 里仍然能够工作良好，但是开发者现在可以使用一个功能更强、更灵活的 LoadImage () 函数。

```
#include 〈winuser.h〉
HANDLE LoadImag (HINSTANCE hInstance,
                 LPCTSTR lpszName,
                 UINT uType,
                 int cxDesired,
                 int cyDesired,
                 UINT fuLoad);
```

参数	说明
HINSTANCE hInstance	包含了图象或者 0 的模块的句柄，该句柄用于载入一幅预先定义好的图标
LPCTSTR lpszName	资源标签，包含了图标或者预定义图标 ID 的物理文件的名字
UINT uType	可以是IMAGE_BITMAP，IMAGE_CURSOR 或者 IMAGE_ICON
int cxDesired	希望的图象宽度，或者设置为 0——采用原始的图象尺寸
int cyDesired	希望的图象高度，或者设置为 0——采用原始的图象尺寸
UINT fuLoad	LR_标志的组合，用于设置图象的尺寸、颜色和属性（参考表 3-3）
返回值	在正文里讨论
HANDLE	图象句柄，假如函数调用失败则返回一个 NULL 值

LoadImage（）的使用是相当灵活的。利用它可以从资源文件、预定义的 Windows 资源或者直接从一个文件里里载入任何类型的位图（DIB，高分辨率、光标和图标等等）。

表 3-3 LoadImage（）的载入选项

标 志	值	说 明
LR_DEFAULTCOLOR	0x0000	原始的颜色运用
LR_MONOCHROME	0x0001	转换成单色模式的图象
LR_COLOR	0x0002	
LR_COPYRETURNORG	0x0004	
LR_COPYDELETEORG	0x0008	
LR_LOADFROMFILE	0x0010	从一个文件里载入的图象
LR_LOADTRANSPARENT	0x0020	用 COLOR_WINDOW 取代的第一个图象象素色
LR_DEFAULTSIZE	0x0040	图象保持它的原始尺寸
LR_LOADREALSIZE	0x0080	
LR_LOADMAP3DCOLORS	0x1000	
LR_CREATEDIBSECTION	0x2000	

LoadImage（）函数解决了用不止 16 色管理图象资源时遇到的某些瓶颈问题。你现在可以在资源文件里存储一幅 256 色的位图，然后在应用程序里载入它，这样不会丢失任何分辨率属性。除此以外，LoadImage（）是载入彩色光标（Win95 支持的一种特性）的唯一途径。

10. HCURSOR hCursor 项

在屏幕上，鼠标光标通常是以传统的白色箭头的样子出现的，它斜斜地指向西北方（指向西雅图?）。有时，它会以某种不同的形状出现——就像沙漏一样。光标的形状并不是一种系统范围内的属性，更准确地说，它是不同的窗口类为基础的。假如鼠标指针覆盖的窗口正好属于某种特定的类，WNDCLASSEX 数据结构里的 hCursor 项就可以指示 Windows 在屏幕上显示何种光标图象。Windows 提供了一系列预先定义好的光标位图，如表 3-4 所示。

表 3-4 Win32 里的预定义光标位图

光标 ID	值	说 明
IDC_ARROW	MAKEINTRESOURCE（32512）	传统的箭头指针
IDC_IBEAM	MAKEINTRESOURCE（32513）	I 型光标
IDC_WAIT	MAKEINTRESOURCE（32514）	沙漏
IDC_CROSS	MAKEINTRESOURCE（32515）	十字光标
IDC_UPARROW	MAKEINTRESOURCE（32516）	垂直箭头
IDC_SIZE	MAKEINTRESOURCE（32640）	四方向箭头
IDC_ICON	MAKEINTRESOURCE（32641）	带有白色边框的小黑方块
IDC_SIZENWSE	MAKEINTRESOURCE（32642）	指向西北和东南的双向箭头
IDC_SIZENESW	MAKEINTRESOURCE（32643）	指向东北和西南的双向箭头
IDC_SIZEWE	MAKEINTRESOURCE（32644）	指向西方和东方的双向箭头
IDC_SIZENS	MAKEINTRESOURCE（32645）	指向北方和南方的双向箭头
IDC_SIZEALL	MAKEINTRESOURCE（32646）	四方向箭头
IDC_NO	MAKEINTRESOURCE（32648）	斜线圆
IDC_APPSTARTING	MAKEINTRESOURCE（32650）	标准箭头和沙漏

LoadCursor（）的语法与 LoadIcon（）的语法是完全一致的，它们都需要用到 hInstance 句柄，或者用 NULL 访问表 3-4 和资源标签内列出的某个预定义对象：

```
#include 〈winuser.h〉
HCURSOR WINAPI LoadCursor (HINSTANCE hInstance,
                           LPCSTR lpCursorName);
```

参数	说明
HINSTANCE hInstance	包含了光标或者 NULL 的模块句柄，用于载入一个预先定义好的光标
LPCSTR lpCursorName	光标标识标签
返回值	在正文里讨论
HCURSOR	光标句柄，假如函数调用失败，则返回一个 NULL 值

11. HBRUSH hbrBackground 项

WNDCLASSEX 结构的这个项可以标识刷子句柄，Windows 利用这个句柄描绘属于这一类的所有窗口的背景。WINUSER.H 提供了几个预先定义好的刷子，如表 3-5 所示。

表 3-5 Windows 95 里预先定义好的背景刷

刷 子	值	说 明
WHITE_BRUSH	0	白色背景
LTGRAY_BRUSH	1	浅灰色背景
GRAY_BRUSH	2	灰色背景
DKGRAY_BRUSH	3	深灰色背景
BLACK_BRUSH	4	黑色背景
NULL_BRUSH	5	没有背景
HOLLOW_BRUSH	NULL_BRUSH	透明背景

无论它们的定义是怎样的，表 3-5 列出的所有刷子都是严格的单色。不同的灰度取决于白色背景上黑点的不同混合情况。如果不想使用单色背景，就可以选用两种方法为窗口背景分配一种特定的彩色。WindowsAPI 里有一个名为 CreateSolidBrush（）的函数，用它可以生成一个新的背景刷。这个背景刷是以三种原色的混合为基础的：红色、绿色和蓝色。例如，假设你对黄色的窗口有意思，那么 CreateSolidBrush（）便是最佳的选择：

```
#include 〈wingdi.h〉
HBRUSH WINAPI CreateSolidBrush (COLORREF);
```

参数	说明
COLORRREF 颜色	一个 32 位数字，对应于红色、绿色和蓝色的组合
返回值	在正文里讨论
HBRUSH	新的刷子句柄，假如函数调用失败，则返回一个 NULL 值

生成一个 COLORREF 值最简单的途径是通过 WINGDI. H 里的宏 RGB：

```
#define RGB (r, g, b)
((COLORREF)(((BYTE)(r) | ((WORD)((BYTE)(g))<<8)) | (((DWORD)
(BYTE)(b))<<16)))
```

在这三种基本色（红、绿和蓝）里，每种色的浓度都是通过 0 到 255 间的一个数值来度量的。举个例子来说，带有红色客户区的一个窗口类是像下面这样设置的：

```
wcex.hbrBackground = CreateSolidBrush (RGB (255, 0, 0));
```

与 RGB 有关的其他三个宏分别用于取得红色、绿色和蓝色的浓度：

```
GetValue (rgb)      ( (BYTE) (rgb))
GetValue (rgb)      ( (BYTE) ( ( (WORD) (rgb)) >> 8))
GetValue (rgb)      ( (BYTE) ( (rgb)>>> 16))
```

事实上，我们很少对窗口客户区域的“系统颜色”分配产生过兴趣。在这儿，系统颜色是当前选择用于描绘特定 UI 部分的颜色。这些 UI 部分包括标题栏、菜单栏、滚动条、按钮以及其他组件。用户可以 Display 属性表的 Appearance 卡片里设置和改变系统的颜色方案。它们对应的数值定义列于表 3-6 里。WINGDI. H 里包含了对系统组件一系列引用，它们对于把各自的颜色分配给一个新的窗口类是很有用的，如表 3-6 所示。

如果想选择一种系统颜色，把它作为窗口类的背景色使用，便可按照下述的方式修改注册进程：

```
wcex.hbrBackground = COLOR_MENU + 1
```

按照这种方式，这一类的所有窗口都会把菜单栏颜色接纳为自己的客户区颜色。

表 3-6　WINDOWS. H 里的完整颜色定义集

颜　色	值	说　明
COLOR_SCROLLBAR	0	定义为滚动条对象分配的颜色
COLOR_DESKTOP	1	定义为桌面分配的颜色
COLOR_ACTIVECAPTION	2	定义为活动标题栏分配的颜色
COLOR_INACTIVECAPTION	3	定义为非活动标题栏分配的颜色
COLOR_MENU	4	定义为菜单栏分配的颜色
COLOR_WINDOW	5	定义为客户区分配的颜色
COLOR_WINDOWFRAME	6	定义为窗口框分配的颜色
COLOR_MENUTEXT	7	定义为菜单正文分配的颜色
COLOR_WINDOWTEXT	8	定义为窗口正文分配的颜色
COLOR_CAPTIONTEXT	9	定义为标题正文分配的颜色
COLOR_ACTIVEBORDER	10	定义为活动边框分配的颜色
119	119	定义为非活动边框分配的颜色
COLOR_APPWORKSPACE	12	定义为窗口背景分配的颜色
COLOR_HIGHLIGHT	13	定义为高亮度菜单项分配的颜色
COLOR_HIGHLIGHTTEXT	14	定义为高亮度菜单项内正文分配的颜色
COLOR_3DFACE	15	定义为一个三维对象分配的颜色
COLOR_3DSHADOW	16	定义为三维对象背景分配的颜色
COLOR_GRAYTEXT	17	定义为菜单内屏蔽正文分配的颜色
COLOR_BTNTEXT	18	定义为按钮正文分配的颜色
COLOR_INACTIVECAPTIONTEXT	19	定义为一个非活动标题栏内正文分配的颜色
COLOR_3DHILIGHT	20	定义为高亮度三维对象分配的颜色

12. LPSTR lpszMenuName 项

对于标准的应用程序来说，尽管并非一个强制性的标准，它通常都还有带有一个菜单栏。为了把菜单栏与一个窗口联系到一起，最简单的方式就是在 WNDCLASSEX 的 lpszMenuName 项里指定一个菜单资源文件。在不同例子里，我采用了不同的方式为应用程序分配菜单栏。因为我考虑到每个窗口也许都需要它自己独具特色的菜单栏。所以，在 WNDCLASSSEX 结构里，lpszMenuName 项的值总是设置成 NULL。

```
wcex.lpszMenuName = NULL;
```

13. int cbClsExtra 和 int cbWndExtra 项

Windows 执行 RegisterClassEx () 和 CreateWindowEx () 的时候，它会存储一大块内存，其中包含了与新类或者 USER.EXE 内部窗口有关的所有信息。这些数据中的很大一部分都没有进行文档化处理，并且是无法访问的（在第 7 章“建立窗口的艺术”里，我们详细讨论了关于类信息和数据的问题）。开发者可以为 WNDCLASSEX 里的 cbClsExtra 和 cbWndExtra 项分配一个非零值，从而增大这些标准内存区的容量，如图 3-2 所示。

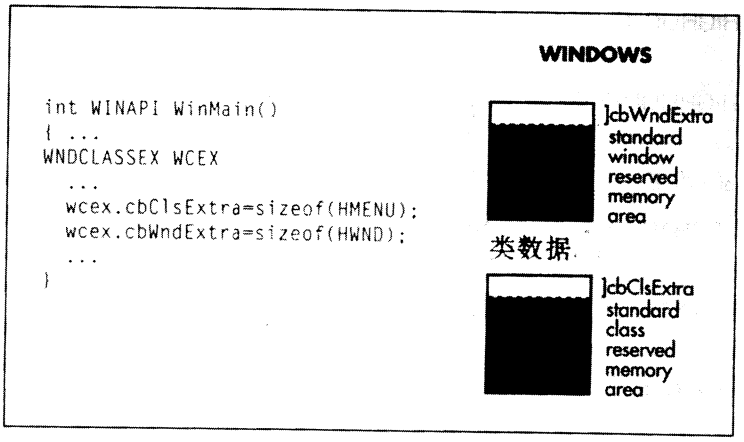


图 3-2 扩展预设置类和窗口保留的内存区域

“额外字节”（Extra bytes）这个术语是指在类和窗口级别上面增加的内存空间。这些空间是非常有用的，利用它们可以把属于某个窗口句柄的附加信息简便地存储下来。16 位环境下的一些限制（臭名昭著的系统资源）在 Windows 95 里仍然是存在的，这样就抵消了由额外字节带来的高效率（第 7 章“建立窗口的艺术”会对这一问题进行深入的探讨）。

14. UNIT Style 项

风格（Style）项的用途是把新类的行为特征通知给 Windows。与大家想象的也许有所出入，由 CS_前缀定义的风格标志对窗口的外观几乎是没有什么影响的。更准确地说，它们造成的影响主要体现在对窗口行为的控制上面，比如描绘和设备背景问题。具体的信息请大家参考表 3-7。

表 3-7 WINUSER.H 里的 CS_风格（类风格），用于 WNDCLASSEX 的 Style 项

风 格	值	说 明
CS_VREDRAW	0x0001	假如窗口的高度发生了变化，整个客户区都会无效
CS_HREDRAW	0x0002	假如窗口的宽度发生了变化，整体客户区都会无效
CS_KEYCVTWINDOW	0x0004	
CS_DBLCLKS	0x0008	Windows 侦测到了鼠标双击事件
CS_OWNDC	0x0020	
CS_CLASSDC	0x0040	
CS_PARENTDC	0x0080	
CS_NOKEYCVT	0x0100	
CS_NOCLOSE	0x0200	系统菜单内没有 Close 菜单项
CS_SAVEBITS	0x0800	
CS_BYTEALIGNCLIENT	0x1000	
CS_BYTEALIGNWINDOW	0x2000	
CS_GLOBALCLASS	0x4000	这种风格要求注册 DLL 内的一个全局类

我们平时可以试着用二进制位运算符 OR（即符号|，ASCII 代码为 124）增加两个或者更多的 CS_风格。从个人来说，我宁愿采用下面这种组合形式，从而确保每次变动窗口尺寸的时候接收到一条 WM_PAINT 消息：

```
...
wc.style = CS_HREDRAW | CS_VREDRAW;
...
```

WIN32BK.H 里有一个新的定义，叫作 CS_SIZEREDRAW，该定义总和了上面的风格：

```
...
#define CS_SIZEREDRAW (CS_HREDRAW | CS_VREDRAW)
...
```

我们注意到很有趣的一点在于，需要用 CS_DBLCLKS 标志去支持鼠标双击。

15. WNDPROC lpfnWndProc 项

从语法的角度来看，这个项要求用到一个指针，并且这个指针要指向类型为窗口进程的一个函数。我说清楚了吗？这并不是我的失误。首先，让我们先对 WNDPROC 数据进行一番简短的回顾。

WINUSER.H 里包含了下面这个定义：

```
typedef LRESULT (CALLBACK * WNDPROC) (HWND, UINT, WPARAM, LPARAM);
```

WNDPROC 是指向类型为 CALLBACK（假如你愿意，也可以称为 WINAPI）的一个函数的指针，该函数接收的是 HWND，UINT，WPARAM 和 LPARAM，返回的则是一个 LRESULT。这种类型的函数通常被称为“窗口进程”（window procedure）。假如你想注册一个新的窗口类，就需要提供类型为 WNDPROC 的一个函数的地址。假如某条消息被定址到属于那种特定窗口类的一个窗口，Windows 就会调用那个函数。在 C 语言里，函数名就是一种特殊的指针，它指向代码起始处的代码段。正如大家不久便要学到的那样，窗口进程就是带有四个参数的一种 WINAPI 函数，如下所示：

```
LRESULT WINAPI ClientWndProc (HWND hwnd,
                              UINT msg,
                              WPARAM wParam,
                              LPARAM lParam)
{
    ...
}
```

在 Win32 里，并非强制性的规定要把这种类型的函数输出到应用程序 DEF 文件的 EXPORTS 段里。然而，这却是最初在 Win16 里采用的方法。Win32 要求 DEF 文件只能应用于 DLL。在 DEF 文件里，只需把这个函数的名字放置到 EXPORTS 段里，便可把该函数简单地列出来：

```
...
EXPORTS
    ClientWndProc
...
```

和其他标准的 C 函数一样，既可以在头文件里，也可以在 C 文件的初始部分里编写这种函数。

3.3.1 类的注册

到现在为止，WNDCLASSEX 结构里的所有内容均已填写完毕，可以安全地调用 RegisterClassEx ()函数对类进行注册了，然后把信息存储到 Windows 里(更准确地说，是存储到 USER.EXE 的数据段里)。假如调用成功，RegisterClassEx () 将返回一个 TRUE 值；假如由于某些原因调用失败了，则返回一个 FALSE 值。所以，我们要尽量养成随时测试返回值的好习惯，假如调用 RegisterClassEx () 失败，就需要中断程序的执行。

```
...
if (! RegisterClassEx (&wcex))
    return FALSE;
...
```

在不同的应用程序里，窗口类的数量是不同的。这主要取决于每个程序的特定需求、它的设计以及结构。无论自己希望注册多少个类，WinMain () 都是容纳所有注册的正确场所。注意，类的定义是一个接一个按顺序排列的。

```
...
WNDCLASSEX wcex;
...
// 为第一个窗口类填写 WNDCLASSEX
...
if (! RegisterClassEx (&wcex))
```

```
return FALSE;
...
// 为第二个窗口类填写 WNDCLASSEX
...
if (! RegisterClassEx (&wcex))
return FALSE;
...
```

一个 WNDCLASSEX 数据便足以对所有类进行注册，我们只需对其中的几个项(类名、窗口进程地址、额外字节尺寸以及图标) 进行变动就可以了。请记住，对于注册的每个类来说，都有与之对应的一个窗口进程，这个进程通常是由 DEF 文件输出的。大家也许提出这样的问题：在两个类之间共享一个窗口进程是不是可行的呢？从技术角度来看，这是可以实现的，然而由此带来的好处或者说优点却是很难说得清楚的。在通常情况下，作出这种选择往往都会带来负面影响。一旦认识到两个窗口具有不同的行为，正确的方案就是把它们放置到各自独立的两个窗口类里。

例如，假设我们想设计一个特殊的窗口：用户单击鼠标右键，客户区就变成红色；再次单击，窗口就中断显示——用鼠标来控制这一切。注意，相同的操作导致了不同的行为。如果采用两个不同的窗口类，消息流自然就会抵达具有明显区别的两个函数。这样就简化了窗口进程的编写（每个窗口都要专门设计）。通过注册多个窗口类，应用程序就可以对多个窗口进程进行访问。与不同窗口有关的消息流最终会抵达独立的窗口进程，应用程序代码就要分布到几个函数里才能完成。根据这种方案，每个窗口类对于某些任务来说都是特殊的，它可以和用户、应用程序输出和数据输出进行交互作用。

hPrevInstance 和 WinMain ()

和 Win16 比较起来，hPrevInstance 含义的改变对 WinMain () 里的代码布局也产生了轻微的影响。在 Win16 里，应用程序的两个或者多个实例都共享同一个代码段，从而减轻内存负载。除此以外，窗口类的注册进程对于两个或者多个实例来说是唯一的。在 Win16 里，应用程序内的所有实例都可以访问同一个类。这样一来，注册代码就封装到了以 hPrevInstance 值为基础的一条 if 语句里。假如 hPrevInstance 不等于零，那么对类进行注册就变得毫无意义了，因为上一个实例已经达到了相同的目的。

在 Win32 里，情况变得稍有不同了。每个进程都有它自己的地址空间，这个空间对于其他应用程序来说是不可访问的。这种情况也适用于同时运行某程序的两份或者多份拷贝。窗口类只有针对相应的进程才是可见的，即使它同时运行了多份拷贝也是如此。

实际上，Win32 里发生的变化是有点奇妙的。系统内注册的每个类都可以通过一个“共用原子”(global atom) 进行访问。“共用原子”这个术语是指由系统通过 GlobalAddAtom () 生成的一个唯一的 ID 编号。该函数只有一个参数——一个字符串，它的返回值就是“共用原子”——一个整数。图 3-3里的 Spy＋＋显示了 Welcome 应用程序可用的所有信息。类的共用原子是 2CE9。

这时，假设我们又启动了 Welcome 的第二个拷贝，然后取回关于它的信息，这时就能发现自己取回的信息是相同的。这意味着尽管这两个实例为了让每个进程都能建立自己的主窗

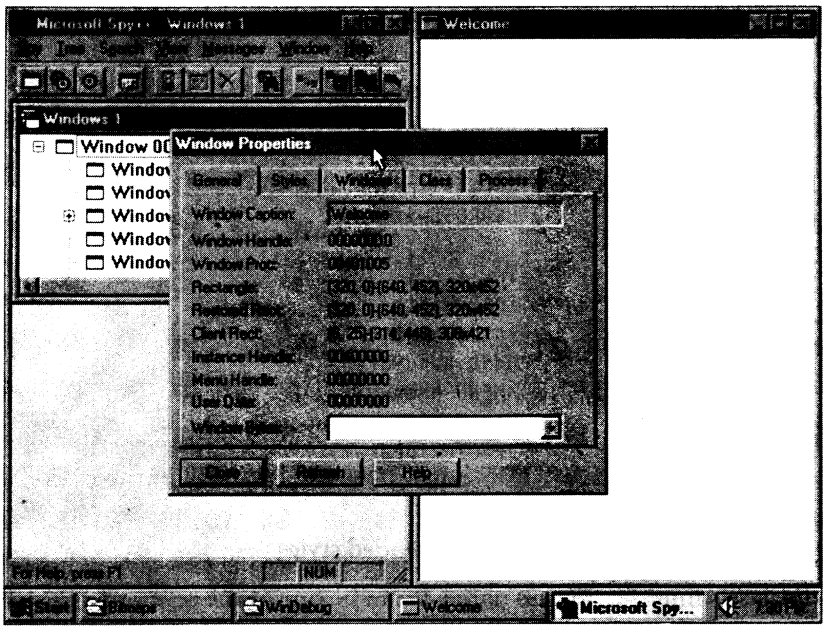


图 3-3　与 Welcome 例子相关的 Spy++返回的信息

口，所以都需要用到注册进程，但是类信息仍然存储于系统内存一个公共场所（类的共用原子编号通常位于这两个拷贝之间）。联机文档在这个问题上极易让人感到混淆不清。在 Win16 里，通过计算 hInstance 句柄可以把不同的任务区分开来。但这种方法在 Win32 里已经行不通了，因为 hInstance 对于每个进程来说都是完全一致的。对于这个问题，我们从第一章“Win32 中的软件开发”开始已经连续提到。在 Win32 里，进程只能通过进程 ID 才能识别，这个 ID 是在进程启动时分配的一个整数值。所以，把 hInstance 显示出来几乎是没有什么用处的，因为它对于所有进程来说都完全相同。

3.4　窗口的建立

现在，我们已经注册了一个新的窗口类，接下来可以开始建立应用程序的主窗口了。针对这一目的，围绕基本的 API 函数：CreateWindowEx（），Windows 还另外提供了几种选择。该函数已经取代了以前的 CreateWindow（），为开发者提供了足够的灵活性和功能。注意，CreateWindow（）函数仍然得到了保留，并且可以在 Win32 中正常地运行。

```
#include〈winuser.h〉
HWND WINAPI CreatWindowEx (DOWRD dwExStyle,
                           LPCSTR lpClassName,
                           LPCSTR lpWindowName,
                           DWORD dwStyle,
                           int X,
                           int Y,
```

```
int nWidth,
int nHeight,
HWND hWndParent,
HMENU hMenu,
HINSTANCE hInstance,
LPVOID lpParam);
```

参数	说明
dwExStyle	Windows 的扩展风格
lpClassName	类名。可以是七个预定义窗口类的其中之一、通用控件或者由应用程序预先注册好的一个类
lpWindowName	窗口标题。这个正文串会在应用程序的标题栏内显示出来。根据窗口类风格和类的整体结构不同，这个参数可能有不同的含义
dwStyle	窗口的基本风格。其中包括一些 WS_风格，以及用于各种预定义窗口类的特殊风格
X，Y	窗口左上角的坐标，该坐标是用父窗口的座表示的。CW_USEDEFAULT 标志强迫 Windows 任意决定窗口在屏幕上的显示位置
nWidth，nHeight	新窗口的宽度和高度，CW_USEDEFAULT 标志强迫 Windows 任意决定窗口的整体尺寸
hWndParent	父窗口的句柄。只有在窗口设置了 dwStyle 参数的 WS_POPUP 或者 WS_CHILD 标志的前提下，才能设置这个参数。它的作用是分别对父窗口和物主窗口进行标识
hMenu	与窗口链接在一起的菜单栏的句柄。只有叠置窗口（WS_OVERLAPPED）和弹出式窗口（WS_POPUP）才能使用菜单栏。子窗口（WS_CHILD）是禁止显示菜单栏的。对于这些窗口来说，该参数的作用是容纳窗口 ID
hInstance	实例句柄
lpParam	指向应用程序数据区的指针。通常设置成 NULL，它对窗口的建立进程是没有影响的

CreateWindowEx（）这个函数看起来相当复杂，但这只是由建立窗口所需大量参数带来的一种错误印象。事实上，关键的参数只包括类名（lpClassName）、窗口风格（dwStyle）以及实例句柄（hInstance）。

为了建立一个普通的窗口，开发者只需要声明类型为 HWND 的一个标识符，用它来容纳返回值便可以了。然后仔细阅读联机帮助，对其中的每个参数做到真正的理解。尽管可以通过风格参数内不同标志的组合，从而使窗口具有不同的特征，但窗口在大多数情况下都是标准的矩形。

系统把窗口分为三种基本的类型——叠置式、弹出式以及子窗口——每种都用一个 WS_

风格来表示。就目前来说，我只准备向大家讲述叠置窗口（WS_OVERLAPPED），这是建立应用程序主窗口的一种最简便的方式。

```
...
hwnd = CreateWindowEx (WS_EX_OVERLAPPEDWINDOW,
                       szClassName,
                       szWindowTitle,
                       WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                       CW_USEDEFAULT, 0,
                       CW_USEDEFAULT, 0,
                       NULL,
                       (HEMNU) NULL,
                       hInstance,
                       NULL);
...
```

新窗口属于以前用存储在 szClassName 标识符内的名字注册的窗口类。现在，我们把注意力放在包含了窗口风格的第四个参数上面，如表 3-8 所示。窗口可以立即显现出来（WS_VISIBLE），同时它的叠置窗口可以包含一系列增加的风格（WS_OVERLAPPEDWINDOW）。

表 3-8　各种窗口使用的基本窗口风格

风　格	值	说　明
WS_OVERLAPPED	0x00000000L	叠置式窗口
WS_POPUP	0x80000000L	弹出式
WS_CHILD	0x40000000L	子窗口
WS_MINIMIZE	0x20000000L	窗口处于最小化显示状态
WS_VISIBLE	0x10000000L	窗口可见
WS_DISABLED	0x08000000L	窗口处于屏蔽状态（没有交互作用）
WS_CLIPSIBLINGS	0x04000000L	如果窗口在同一个分级级别上重叠，Windows 将分别对这些窗口进行重画
WS_CLIPCHILDREN	0x02000000L	父窗口利用这个标志避免重画由子窗口占据的客户区
WS_MAXIMIZE	0x01000000L	窗口处于最大化显示状态
WS_CAPTION	0x00C00000L	在窗口矩形的顶部显示了一个标题栏（AKA 标题栏）
WS_BORDER	0x00800000L	一条细边框围绕着整个窗口
WS_DLGFRAME	0x00400000L	一条典型的对话框边框围绕着整个窗口
WS_VSCROLL	0x00200000L	在窗口内增加了一个垂直滚动条
WS_HSCROLL	0x00100000L	在窗口内增加了一个水平滚动条
WS_SYSMENU	0x00080000L	窗口左上角显示了一个系统菜单（要求先设置 WS_CAPTION）
WS_GHICKFRAME	0x00040000L	一条粗边框围绕着整个窗口，用户对这个边框进行操纵便可重新窗口的大小
WS_GROUP	0x00020000L	在对话框内定义一个逻辑控件组
WS_TABSTOP	0x00010000L	在对话框内定义一个制表符停止位符号
WS_MINIMIZEBOX	0x00020000L	在右上角显示一个最小化图标

（续）

风格	值	说明
WS_MAXIMIZEBOX	0x00010000L	在右上角显示一个最大化图标
WS_TILED	WS_OVERLAPPED	考虑到与 Windows 1.x 的兼容性原因而保留下来的
WS_ICONIC	WS_MINIMIZE	最小化显示窗口
WS_SIZEBOX	WS_THICKFRAME	用于调整窗口大小的边框
WS_TILEDWINDOW	WS_OVERLAPPEDWINDOW	考虑到与 Windows 1.x 的兼容性原因而保留下来的
WS_OVERLAPPEDWINDOW	(WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME | WS_MINIMIZEBOX | WS_MAXIMIZEBOX)	用几种风格属性定义了一个叠置式窗口
WS_POPUPOWINDOW	(WS_POPUP | WS_BORDER | WS_SYSMENU)	用几种风格属性定义了一个弹出式窗口
WS_CHILDWINDOW	(WS_CHILD)	证明人的想法是非常有限的

接下来，让我们继续学习第一个参数：dwExStyle。这个参数是最近才增加在 Win32 里的，用于对 Win32 支持的窗口风格进行扩展。在程序示例里，我们采用了 WS_EX_OVERLPAAEDWINDOW 风格。从表面来看，这是让人感到十分迷惑的名字。正如表 3-9 显示的那样，它是两种新扩展风格的简单组合，这两种新风格在客户区（WS_EX_CLIENTEDGE）和窗口边框（WS_EX_WINDOWEDGE）都实现了三维边框。在代码段里，我们让 Windows 来决定窗口的位置和尺寸。

表 3-9 CreateWindowEx（）使用的扩展风格

扩展风格	值	说　明
WS_EX_DLGMODALFRAME	0x00000001L	增加一个双线边框
WS_EX_NOPARENTNOTIFY	0x00000004L	假如具有这种风格的子窗口是通过 WM_PARENTNOTIFY 来建立或者破坏的，它就不会向自己的父窗口作报告
WS_EX_TOPMOST	0x00000008L	窗口总是位于屏幕的顶部
WS_EX_ACCEPTFILES	0x00000010L	该窗口接受拖放文件
WS_EX_TRANSPARENT	0x00000020L	在标题栏右上角放置一个问号图标。
WS_EX_MDICHILD	0x00000040L	
WS_EX_TOOLWINDOW	0x00000080L	建立一个工具窗口；这个窗口具有浮动工具栏的用途。工具窗口有一个标题栏，它比普通的标题栏要短一些。同时，窗口标题是用小一号的字体书写而成的。工具窗口不会在任务栏或者通过按 ALT+TAB 键显示出来的窗口中出现
WS_EX_WINDOWEDGE	0x00000100L	窗口有一个三维边框
WS_EX_CLIENTEDGE	0x00000200L	客户区有一个具有三维雕刻效果的边框
WS_EX_CONTEXTHELP	0x00000400L	建立右上角带有一个问号的图标
WS_EX_LEFTSCROLLBAR	0x00004000L	沿右边框放置的一个垂直滚动条，或者沿上边框放置的一个水平滚动条
WS_EX_RIGHTSCROLLBAR	0x00000000L	沿左边框放置的一个垂直滚动条，或者沿下边框放置的一个水平滚动条
WS_EX_CONTROLPARENT	0x00010000L	允许用户通过按 TAB 键在子窗口间切换
WS_EX_STATICEDGE	0x00020000L	建立带有三维边框风格的一个窗口，该窗口用于显示不接受用户输入的项目
WS_EX_OVERLAPPEDWINDOW	(WS_EX_WINDOWEDGE | WS_EX_CLIENTEDGE)	两个三维边框的组合
WS_EX_PALETTEWINDOW	(WS_EX_WINDOWEDGE | WS_EX_TOOLWINDOW | WS_EX_TOPMOST)	带有三维边框的最顶层窗口以及工具窗口的外观

3.4.1 注意一些常见的失误

与 LoadIcon（），LoadCursor（）以及 LoadImage（）一样，CreateWindowEx（）也会返回一个窗口句柄。正如我们在第 2 章“Win32 开发工具”中指出的那样，为了验证 CreateWindowEx（）函数的调用是否成功，必须对窗口句柄进行检查。任何一个非零值都是有效的句柄。请记住，假如没有明确添加 WS_VISIBLE 标志，窗口在屏幕上便是看不见的。

第二个潜在的错误区域与窗口的类名有关。在类参数里分配的名字与用于窗口类注册的名字有可能不符，这种情况是很常见的。为了解决名字的这种不匹配，可以把类名存储到一个标识符里，然后通过代码来使用该标识符，这样就能简单地避开这一问题。

随后，大家必须对 WS_标志引起足够的重视。假如使用了 WS_CHILD，就有必要在 hwndParent 参数里提供一个窗口句柄，这样才能建立真正的“父/子”关系。

3.4.2 窗口的显示

显示一个窗口最简单的方式就是在 CreateWindowEx（）调用里放置 WS_VISIBLE 标志。否则，也可以考虑采用 ShowWindow（）这个 API 函数：

```
BOOL WINAPI ShowWindow（HWND hwnd，int nCmdShow）；
```

参数	说明
HWND hwnd	对准备显示的窗口进行识别的句柄
int nShowCmd	可视化标志
返回值	说明
BOOL	布尔值。假如窗口以前就是可见的，值为 TRUE；假如是隐藏的，则为 FALSE

ShowWindow（）里的标志参数可以采用表 3-10 里的任何一个值。

表 3-10 用于显示或者隐藏窗口的 ShowWindow（）标志

标 志	值	说 明
SW_HIDE	0	隐藏窗口
SW_SHOWNORMAL	1	显示和激活窗口
SW_NORMAL	1	显示和激活窗口
SW_SHOWMINIMIZED	2	显示和最小化窗口
SW_MAXIMIZE	3	显示和最大化窗口
SW_SHOWMAXIMIZED	3	显示和最大化窗口
SW_SHOWNOACTIVATE	4	显示窗口，但不激活它
SW_SHOW	5	显示窗口
SW_MINIMIZE	6	显示和最小化窗口
SW_SHOWMINNOACTIVATE	7	显示和最小化窗口，但不激活它
SW_SHOWNA	8	显示窗口，但不激活它
SW_RESTORE	9	把窗口恢复成以前的尺寸和位置

这里需要考虑到两个问题：正如我们在本章开头就提到的那样，WinMain（）里的第四个参数最初是当作 ShowWindow（）调用里的标志值使用的。所以应该尽量避免使用它，平常一般使用 SW_标志。除此以外，在某些示范代码里，ShowWindow（）调用的后面总是跟随

着一个 UpdateWindow () 调用。这种处理是完全没有什么意义的，因为它对代码唯一的影响就是白白损失了大量 CPU 时间。

关于 ShowWindow () 最后一个需要注意的问题在于：有些时候，开发者需要先在屏幕中显示一个窗口，然后再把它隐藏起来。ShowWindow () 是满足这种需求的最佳选择。不要老想着要动态地建立一个窗口，然后再把这个窗口破坏（删除）掉。后面这种方法有画蛇添足的感觉，且速度奇慢。

3.4.3 消息循环的实现

一旦注册好一个类，而且屏幕中也显示出了适当的窗口，WinMain () 里就只剩下一个尚未完成的步骤了——实现消息循环。WinMain () 的这个部分对于用户和程序代码间的交互作用是有影响的。应用程序将在消息循环中等待进入消息的抵达。

```
...
// 消息循环
while (GetMessage (&msg, NULL, 0, 0))
{
    TranslageMessage (&msg);
    DispatchMessage (&msg);
}
...
```

这个代码段里包含了三个基本的 API 函数，它们在一起便构成了一个标准的消息循环。这三个函数分别为：GetMessage ()，TranslateMessage () 和 DispatchMessage ()。使用加速键和非模态对话框以后，它们对消息循环的结构都分别进行了改动。GetMessage () 是 Windows 多任务的心脏。除非应用程序消息队列内出现了一条消息，否则这个函数是不会返回任何值的。由于 GetMessage () 要在当前进程里等待消息，这就为正在运行的其他应用程序提供了一个机会，使它们能够检查各自的消息队列。投递一条消息以后，GetMessage () 最终会取回这条消息，然后把关于它的信息存储到一个 MSG 数据结构里。

```
typedef struct tagMSG
{
    HWND hwnd;
    UINT message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
} MSG, *PMSG, NEAR *NPMSG, FAR *LPMSG;
```

项	说明
HWND hwnd	目标窗口的句柄

UINT message	当前消息的数值
WPARAM wParam	一种 32 位数据，其中包含的信息与当前的消息有关
LPARAM lParam	一种 32 位数据，其中包含的信息与当前的消息有关
DWORD time	消息在应用程序消息队列里显示时的时间
POINT pt	传递消息时，鼠标光标在屏幕坐标里的位置

对于强迫循环退出和进程中断的每一条消息来说（除 WM_QUIT 以外），GetMessage () 都会返回一个 TRUE 值。通常，消息循环的后面都跟随一条独立的返回语句，用于强迫 WinMain () 返回系统。很少出现的一种情况是：返回语句之前还有一些清除指令，用它来释放执行过程中分配的资源（把这种代码放置到应用程序主窗口进程的 WM_CLOSE 消息块里是相当方便的）。

在 GetMessage () 后面紧跟着的是 TranslateMessage () 函数，它的作用是获得对 MSG 数据结构内容进行运用和修改的控制权。就真正的意义上看，TranslateMessage () 限制了它对 WM_KEYDOWN/WM_KEYUP 和 WM_SYSKEYDOWN/WM_SYSKEYUP 的影响，分别把它们转换成一个 WM_CHAR 或者 WM_DEADCHAR，以及 WM_SYSCHAR 或者 WM_SYSDEADCHAR。

我们现在再来看一看 DispatchMessage ()，它的作用是找出准备调用的窗口进程。这种寻找并不是随机的，更准确地说，它是以 MSG 的 hwnd 项所标识的目标窗口值为基础来作出决定的。应用程序队列中的每条消息最终都会传递到一个窗口进程里。从技术角度来看，它的最终目的地就是一个窗口。假如在 hwnd 里没有一个有效的值，DispatchMessage () 就不能决定正确的窗口进程。

对消息的处理是在窗口进程里进行的。我们不能预知处理一条消息需要多长的时间。一旦窗口进程中止了，控制权就会返回消息循环，而 GetMessage () 则会重新执行一遍。如图 3-4 所示。

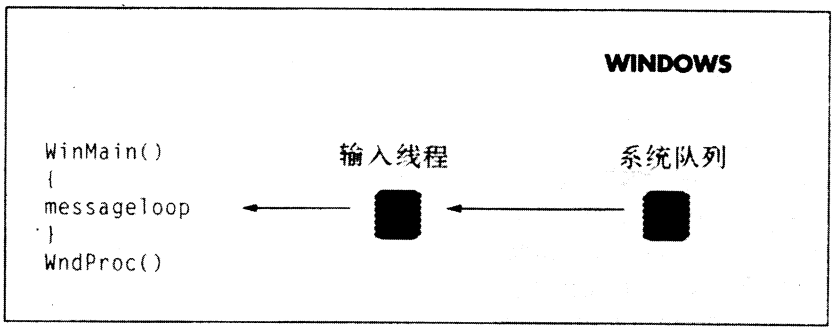


图 3-4　消息流：从 WinMain () 到一个普通的窗口进程，然后再返回消息循环

应用程序可以通过调用 PostQuitMessage()这个 API 函数来强制发生 WM_QUIT 事件。

```
#include〈winuser.h〉
VOID WINAPI PostQuitMessage (int nExitCode);
```

其中的参数并没有特定的含义。就我本人来说，我总是把它设置成零。它的值会在 MSG 数据

结构的 msg. wParam 项里出现。

3.5 理解窗口进程

窗口进程是一种函数，它是由应用程序提供的。每次发生了与特定类窗口有关的一个事件时，Windows 就会调用这个函数。窗口类的注册要求对用作类窗口进程的一个函数的名字进行声明。

到这时，大家已经知道一个应用程序需要窗口与用户进行交互作用（比如键盘输入和鼠标移动等等）。对于用户来说，所有这些行动都是看得到的，但对于应用程序来说却是不可理解的。为了解决这个问题，Windows 对每种系统事件都进行了标识和编号。这种信息（即我们常说的“消息”）随后就会传递给属于特定类的窗口的窗口进程。现在假设我们在属于 Tweny 类的某个窗口上面移动鼠标。鼠标的每次运动都会由系统进行解释，把信息转换成一条消息（WM_MOUSEMOVE），然后把该消息传递给对应的窗口进程。对窗口进程内通过的不同消息进行拦截，这样就可以让应用程序“理解”和监视正在发生的事件，并且“知道”用户作出的每一个选择。

现在，让我们再对一个普通 Windows 应用程序的整体结构进行一番审查。WinMain () 是应用程序的入口点。开发者在 WinMain () 里要根据设计需求注册一个或者多个窗口类、建立主窗口以及在消息循环里等待一条消息的到来。当一条消息到达应用程序的消息队列以后，它立即就会被派遣到适当的窗口进程里。对于普通的 Windows 应用程序来说，即使规模最小的结构至少都需要两个函数：WinMain () 和一个窗口进程。

从 C 语言的角度来看，窗口进程是一个能返回 LRESULT 值的函数，它能接受四个参数。在窗口进程的代码里，很大一部分地方都是由切换块代码占据的，只有这样才能完成应用程序的任务和实现进程的行为和功能。

下面列出来的是 ClientWndProc () ——WELCOME. C 里的窗口进程：

```
LRESULT WINAPI ClientWndProc (HWND hwnd,
                              UINT msg,
                              WPARAM wParam,
                              LPARAM lParam)
{
    switch (msg)
    {
        case WM_CLOSE:
        {
            PostQuitMessage (0);
        }
            break;

        default:
            break;
```

```
    }
    return DefWindowProc (hwnd, msg, wParam, lParam);
}
```

不管我们希望如何，ClientWndProc () 看起来都是相当短小的。唯一的 WM_CLOSE 消息被拦截了。假如用户选择系统菜单内的 Close 菜单项，这个消息就会传递至一个窗口进程。删除窗口的时候，在抵达窗口进程的所有消息里，该消息是第一条抵达的。在 ClientWndProc () 里，我们要对 PostQuitMessage () 的调用进行限制。

进入 ClientWndProc () 的每条消息最终都会抵达 DefWindowProc () ——由 Win32API 提供的一个函数。DefWindowProc () 为每条独立的消息都提供了缺省的处理，有时它什么都不做，有时则会自动生成新消息。ClientWndProc () 采用的方案对于所有消息的处理都是有利的，不管它是不是以前在切换代码块里被拦截下来的。

3.5.1 拦截和处理

忽略其长度的短小和有限的功能，Welcome 仍然不失为一个真正的 Windows 程序。正如我们以前指出的那样，抵达 ClientWndProc () 的每个 WM_（窗口消息——Window Message）最终都会传递到 DefWindowProc () 里。这才是解决问题的正确途径。在其他许多书籍和出版物里（甚至在微软公司随同 SDK 提供的一些例子里），DefWindowProc () 在切换代码块内都是以缺省状态出现的。所以，只有在窗口进程里没有明确进行拦截的消息才能传递给 DefWindowProc ()。下面这个代码段显示了窗口消息处理的不同策略：

```
...
case WM_CREATE:
    ...
    return 0;
...
default:
    return DefWindowProc (hwnd, msg, wParam, lParam);
...
```

消息块末尾的返回（return）语句可以防止消息进入 DefWindowProc ()。所以，消息块内的代码就代表了它正在进行的处理。这种方式不会产生任何运行期错误，尽管从概念来说它是不正确的。正如以后的章节里要解释的那样，最常见的一种情况是：对消息进行拦截的唯一用途是容纳适于在那种特定时刻执行的一个代码段。下面这个例子将阐明这个概念。

对于 CreateWindowEx () 来说，它的作用是生成一系列消息，这些消息都被定址给类窗口进程。在这些消息里，有一条特殊的消息叫作 WM_CREATE。这条消息最有可能在某个窗口进程中间激发，因为它提供了把代码放置到那个消息块内的机会，这样才能完成窗口的建立进程。当 WM_CREATE 到达进程以后，正在建立的窗口就不再是可见的了，尽管此时它的所有功能仍然存在。

假设你现在想用下面的进程从资源文件里载入一个图标，把它与最新建立的窗口联系起

来：

```
...
case WM_CREATE:
{
    hicon = LoadIcon (hInstance, " TWENY");
    return FALSE;
}
...
```

载入一个图标与 WM_CREATE 消息是一点边都沾不上的。无论 wParam 还是 lParam 都不会在这个进程中涉及到。最后的“返回”(return) 关键字可以防止 WM_CREATE 到达 DefWindowProc ()。接下来会发生什么事情呢？什么事都没有发生。代码工作正常，没有产生任何明显的影响。尽管我们不知道 DefWindowProc () 接收到一条 WM_CREATE 消息后会产生什么样的反应，但我们可以有把握地说：这个块内的代码并不代表它要进行的处理。这个例子允许我们利用 WM_CREATE 消息去定位与窗口建立进程具有逻辑关系的代码段。

很少有开发者明确表示自己会拒绝实施由 DefWindowProc () 提供的标准功能（这种情况的一个例子将在本章稍后的部分讲述）。假如出现了这种情况，应用程序就会用消息块内编写的指令来取代缺省的处理方式。第二种方式一次只能申请一次。

3.5.2 建立开发规则

在前面的这些开发阶段里，我们有必要遵循一定的开发规则，从而降低学习的强度，并且加快开发步骤。总而言之，编写一个 Windows 应用程序的时候，下面这些规则我们应当遵守的：

- ▶在源代码开头包括 WINDOWS. H。
- ▶对代码内的所有函数进行原型化处理。
- ▶尽量避免使用全局变量。
- ▶编写 WinMain ()，为它的四个参数分配通用的名称：hInstance，hPrevInstance，lpCmdLine 和 nCmdShow。
- ▶注册一个或者多个窗口类。
- ▶建立和显示应用程序主窗口。
- ▶检查消息循环。
- ▶编写窗口进程，实现一个切换代码块，这个切换块至少要在用 WM_CLOSE 消息调用 PostQuitMessage () 的时候激发。
- ▶把每条消息都传递给 DefWindowProc ()。
- ▶编写一个 DEF 文件。

这些是我向大家建议的基本规则，它有助于诸位顺利编写出自己的第一段 Windows 代码。一旦已经掌握了整体结构，通过增加一些新的功能和特性，很容易就可以使自己的应用程序升级。

bcc32 -c -v start.c
brcc32 start.rc
tlink32 ...
cw32.lib import.lib, start.res
c0w32.obj start.obj, start.exe

3.6 欢迎进入 Win32 的世界

本书附带 CD 的 Listing 3.1 里向大家介绍了 Welcome 程序，如图 3-5 所示。

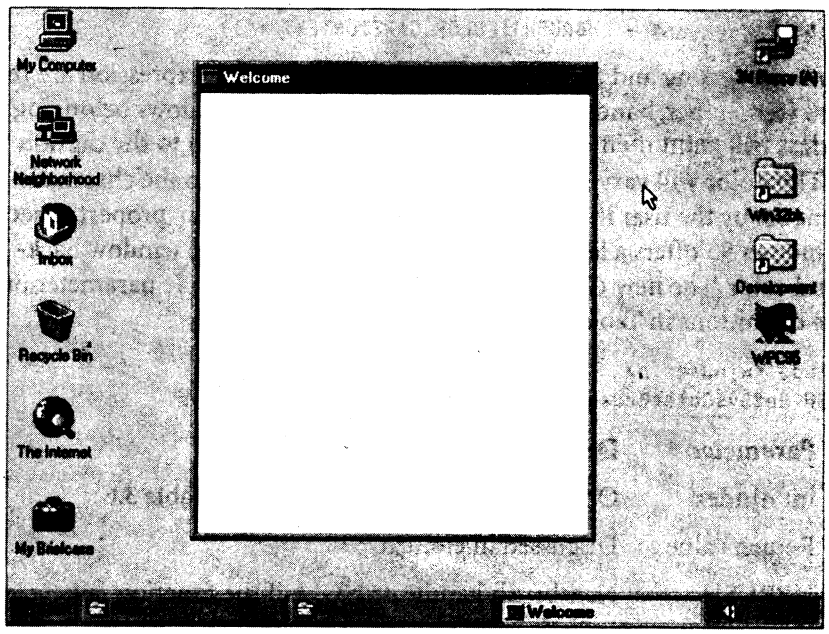


图 3-5 Welcome 是一个标准的 Windows 应用程序，它只用了一个叠置窗口作为自己的主窗口

现在，Welcome 已经启动并且运行起来了，我们可以对其进行修改，让它更具有吸引力、功能更强。第一项改变涉及到客户区的背景颜色。

改变客户区的颜色

利用 GetStockObject ()，WNDCLASSEX 数据结构允许我们在预先定义好的各种背景刷中选择一种。为了建立一种颜色刷，必须用到我们以前讲述过的 CreateSolidBrush () 函数。下面这个代码代码展示了如何获得一个红色的背景：

```
wcex.hbrBackground = CreateSolidBrush (RGB (255, 0, 0));
```

为了把一种颜色分配给窗口背景，我们还可以选用另外一个办法。不再是建立一个新的刷子，我们可以用一个类来取得已经分配给系统组件的某种系统颜色。这些组件包括滚动条、标题栏、桌面、按钮以及用户界面的其他元素。举个例子来说，如果想把标题正文的颜色分配给客户区，就可以使 COLOR_CAPTIONTEXT 定义的值增 1。然后把结果值存储到一个 WNDCLASSEX 结构的 hbrBackground 项里，如下所示：

```
wcex.hbrBackground = (HBRUSH) (COLOR_CAPTIONTEXT + 1);
```

表达式周围的括号是必需的（记住，在 Win32 里，句柄的本质就是指针）。属于这一类的所有窗口都会用标题正文的颜色对自己的客户区重画。由于用户在 Display 属性表的 Appearance 卡片里选择了不同的彩色方案，所以这种颜色也互不相同的。

Windows 95 提供了一种不实际的方法来改变窗口的背景颜色。新的 GetSysColorBursh

() 假设自己唯一的那个参数是表 3-6 里的一种定义。

```
#include 〈winuser. h〉
HBRUSH GetSysColorBrush (int nIndex);
```

参数	说明
int nIndex	表 3-6 里列出的一种 COLOR_定义
返回值	在正文里讨论
HBRUSH	一个新的刷子句柄，如果函数调用失败，则返回一个 NULL 值

代码就变成了：

```
wcex. hbrBackground = GetSystemColorBrush (COLOR_CAPTIONTEXT) ;
```

这条语句把相同的颜色分配给窗口背景。

3.7 “欢迎”的其他注意事项

我现在邀请大家执行 Welcome 程序，把窗口移至屏幕上一个特定的位置（比如与任务栏毗邻），然后关闭窗口，接着重新启动该程序。这时会出现什么情况呢？Welcome 记住了它上次关闭时的位置，并且正确恢复到了它以前的大小。这看起来是不是有些不可思议呢？事实上，这并不是一个奇迹，也不是依赖 Windows 95 的系统特性实现的。这种行为是真正编入应用程序代码里的。假如读者仔细检查 Listing 3.1 里的源代码，就会注意到两个函数：GetWindowPos () 和 SaveWindowPos ()。应用程序启动的时候和中断前将分别调用这两个函数。

下面看一看 SaveWindowPos () 的源代码：

```
BOOL SaveWindowPos(LPRECT prc, HINSTANCE hInstance)
{
    char szInfo[32];
    char szProfile[256];

    // get the executable name
    if(GetModuleFileName(hInstance, szProfile, sizeof(szProfile)))
    {
        // substitute the . EXE extension with . INI
        * strrchr(szProfile, '.') = '\0';
        strcat(szProfile, ".ini");

        // save the window position
        WritePrivateProfileString("Window Position", "X",
                                  itoa(prc -〉left, szInfo, 10),
                                  szProfile);
```

```
            WritePrivateProfileString("Window Position", "Y",
                                      itoa(prc -> top, szInfo, 10),
                                      szProfile);
            // save the window size
            WritePrivateProfileString("Window Position", "CX",
                                      itoa(prc -> right - prc -> left, szInfo, 10 ),
                                      szProfile);
            WritePrivateProfileString("Window Position", "CY",
                                      itoa(prc -> bottom - prc -> top, szInfo, 10),
                                      szProfile);
            return TRUE;
        }
        return FALSE;
    }
```

SaveWindowPos()接收了一个指针,该指针指向一个 RECT 结构和应用程序实例句柄。GetModuleFileName 则返回一个完整的进程文件名(包括扩展名)。用更合适的 .INI 来取代 .EXE 扩展名是有必要的,这样就生成了一个名为 WELCOME.INI 的文件。从那以后,对 WritePrivateProfileString()的四次连续调用就把窗口的位置和尺寸存储到初始化文件(.INI)里了,存储的结构如下所示:

```
[Window Position]
X=380
Y=71
CX=370
CY=96
```

WritePrivateProfileString () 是一个“有魔力”的 API,它帮助开发者完成了所有这些工作。在另外一方面,GetPrivateProfileInt () 会读取这些信息,然后把它存储到一个 RECT 结构里。这两个 API 在 Win32 里都已被注册表数据库 API 取代,只是由于兼容性方面的原因才把它们保留下来了。

```
BOOL GetWindowPos(LPRECT lprc, HINSTANCE hInstance)
{
    char szProfile[256];

    // get module name
    if(GetModuleFileName(hInstance, szProfile, sizeof(szProfile)))
    {
        // substitute the .EXE extension with .INI
        *strrchr(szProfile, '.') = '\0';
```

```
            strcat(szProfile, ".INI");

            // get the window's position (upper left corner coordinates)
            prc -> left = GetPrivateProfileInt("Window Position",
                                               "X", 0, szProfile);
            prc -> top = GetPrivateProfileInt("Window Position",
                                               "Y", 0, szProfile);
            // get the window dimensions (width & height)
            prc -> right = GetPrivateProfileInt("Window Position",
                                               "CX", 0, szProfile);
            prc -> bottom = GetPrivateProfileInt("Window Position",
                                               "CY", 0, szProfile);
            return TRUE;
        }
        return FALSE;
    }
```

GetWindowPos () 的作用是通过调用 GetPrivateProfileInt () 获取来自 WELCOME. INI 的信息，然后把这些信息直接放置到由一个 LPRECT 标识符指定的内存区域里。

在进行更深入的学习之前，让我们解决某些疑点，并且回答读者心中或许已经产生的问题。首先，让我们把“应用程序启动的时候”以及“中断前”这两种表达转换成对应的代码部分。在这个时候，我们在 Windows 编程上的经验就显得相当不足了，但是某些基本的概念还是存在的。举个例子来说，窗口的位置和尺寸是两种不同的数据，调用 CreateWindowEx () 之前就需要用到这两种数据。大家可以在 RegisterClassEx () 之后和 CreateWindowEx () 之前对 GetWindowPos () 进行一次调用。这时必须先声明一个 RECT 标识符，用它存储从 GetWindowPos () 取回的信息。

```
...
RECT rc;
...
if (! RegisterClassEx (&wcex))
    return FALSE;
GetWindowPos (&rc, hInstance);
...
```

一旦知道了窗口的尺寸和位置，就可以把这些信息传递给 CreateWindowEx () 了，如下所示：

```
...
```

```
hwnd = CreateWindowEx (WS_EX_CLIENTEDGE | WS_EX_WINDOWEDGE,
                       szClassName,
                       szWindowTitle,
                       WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                       rc.left, rc.top,
                       rc.right, rc.bottom,
                       NULL,
                       (HMENU) NULL,
                       hInstance,
                       NULL);
...
```

GetWindowPos () 把窗口的总宽度存储到 RECT 结构的 right（右）项里；而窗口的高度则存储到 bottom（下）项里（通常，无论宽度还是高度都是通过用 rc. right 减去 rc. left，用 rc. bottom 减去 rc. top 得出的）。SaveWindowPos () 的作用是把窗口的位置和尺寸存储到应用程序相同目录下的一个 . INI 文件里，这种存储在应用程序中断前的瞬间即可完成。第一次执行 Welcome 的时候，目录里不会找到这个 . INI 文件，由 RECT 结构内的 GetWindowPos () 返回的值是 NULL。调用 CreateWindowEx () 之前，将检查 rc 标识符内的值，假如 right 和 left 项包含了相同的数字，就会生成一些临时值。第一次执行 Welcome 的时候，程序就会在坐标（10，10）处显示它的主窗口，缺省的窗口尺寸则是每一轴跨越 200 个象素点。

```
...
GetWindowPos (&rc, hInstance);
if (rc.left == rc.right)
{
    rc.left = rc.top = 10;
    rc.right = 200;
    rc.bottom = 200;
}
hwnd = CreateWindowEx (…);
...
```

现在很容易就可以在应用程序代码内找到一个合适的地方来设置对 SaveWindowPos () 的调用了。紧随在消息循环后面的地方就是一个理想位置。

```
...
    while (GetMessage (&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg);
```

```
        DispatchMessage (&msg);
    }
    ...
    SaveWindowPos (&rc, hInstance);

    return msg.wParam;
}     //end of WinMain ()
```

经过这样处理后，最初的窗口位置和尺寸就存储到 .INI 文件里去了。在执行过程中，用户可以改变了窗口的大小，并且在屏幕中到处移动。SaveWindowPos () 应该先于调用 GetWindowRect () 之前执行，这样才能获得正确的窗口位置和尺寸。这个 API 函数将把窗口左上角和右下角的位置存储到一个 RECT 结构里。下面是具体的代码示范：

```
...
    while (GetMessage (&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    GetWindowRect (hwnd, &rc);
    SaveWindowPos (&rc, hInstance);

    return msg.wParam;
}     //end of WinMain ()
```

下次执行 Welcome 的时候，GetWindowPos () 就会从应用程序的 .INI 文件里获取正确的窗口尺寸和位置信息。你喜欢为 Welcome 增加的这种扩展功能吗？对了，这非常不错，尽管现在它还不能正常运作。在源代码里还存在着一个潜在的问题，这使得程序的运行不能正常进行。GetWindowRect () 将返回一个 FALSE 值，由此证明它的调用是不成功的。这是怎么回事呢？答案很简单。退出消息循环的时候，应用程序主窗口还没有退出，所以 GetWindowRect () 不能决定 SaveWindowPos () 所需的新值到底是什么。为了解决这个问题，你必须在源代码的范围里声明 RECT 标识符，然后把 GetWindowRect () 移至 WM_CLOSE 消息块里。经过这样处理以后，对窗口状态的检查就会在关闭（删除）窗口之前发生，然后才有正确的信息传递给 SaveWindowPos ()。考虑到我们的编程规则里说过要避免使用全局变量，所以我建议大家把 GetWindowRect () 和 SaveWindowPos () 都移到 WM_CLOSE 消息块里，就像下面这样：

```
...
case WM_CLOSE;
```

```
{
    RECT rc;

    GetWindowRect (hwnd, &rc);
    SaveWindowPos (&rc, hInstance);
}
    break;
...
```

这种方案导致了与 hInstance 有关的另外一个问题产生。这种数据在 WinMain () 里是可用的，然而在 ClientWndProc () ——主要的类窗口进程里却无法对该数据进行访问。为了解决这个问题，我们可以使用 GetWindowLong ()。

```
...
case WM_CLOSE;
{
    RECT rc;

    GetWindowRect (hwnd, &rc);
    SaveWindowPos (&rc, (HINSTANCE) GetWindowLong (hwnd, GWL_
    HINSTANCE));
}
    break;
...
```

Welcome 的最终版本列于 Listing 3.1 内。我对当前的这个 Welcome 版本仍然不是十分满意。前面讨论过的对代码的扩展和变动引入了 Windows 编程的一个关键概念：代码的布局。最初的方式是在 WinMain () 里同时调用 GetWindowPos () 和 SaveWindowPos ()。在 Win32 里，这种编程思路是错误的。WinMain () 的地位相当于应用程序的入口点，它的功能在这儿得到了很好的定义。任何附加的代码都必须增添到一个窗口进程里。为了让 Welcome 按照自己的意愿运行起来，必须把一部分代码移至 WM_CLOSE 消息里，这并不是一种巧合，而是编程的需要。标识了恰当的消息实例以后，对 GetWindowPos () 最初的调用也可以放置到窗口进程内的某个地方。

在本书附带 CD 的 Listing 3.2 里，Welcome 已经稍微进行了一些修改。它现在可以拦截 WM_CREATE 消息，用它来容纳对 GetWindowPos () 的初始调用。经过这样处理后，与 .INI 文件读写有关的所有逻辑都放置到了正确的地方——窗口进程里。

3.8 注册表数据库的运用

Windows 3.x 最大的一个难题在于它的系统配置。在 Windows 3.x 里，系统配置信息是

用多个文件来分配的，这些文件包括 CONFIG.SYS，AUTOEXEC.BAT，WIN.INI，CONTROL.INI 及其他。为了改善这种混乱的状况，微软设计了注册表数据库（Registry database），这是一系列二进制数据文件的集合。用户是看不到这些数据文件的，只能通过一种特定的 API 对其进行调用。Windows NT 也采用了“注册表”的概念，Windows 95 支持的则是经过轻微改动的一个注册表的版本，这个版本增加了一些特性，同时了剔除了某些不适用的东西。Win95 注册表内包含了重要的“即插即用”信息，这种信息在 NT 里是完全找不到的。从另外一方面来看，NT 注册表更安全。Win95 提供的 REGEDIT 实用程序（如图 3-6 所示）可以让我们对注册表数据库进行预览。

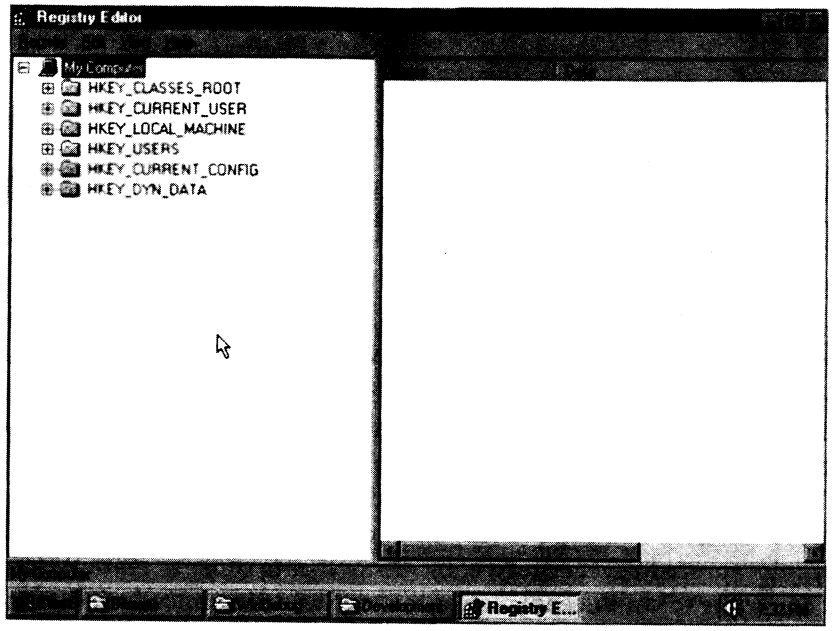


图 3-6　注册表数据库显示了所有这六种基本键

注册表数据库最引人入胜的地方在于数据和对应的物理存储方式是完全独立的。列出的所有信息都提供了几个物理性的文件，主要是存储在\WINDOWS 目录下的 USER.DAT 和 SYSTEM.DAT。USER.DAT 里包含了与不同用户有关的特定信息，这些信息在用户元文件（profile）里得到了反映。SYSTEM.DAT 则存储了与特定硬件和计算机设置有关的信息，这些信息则反映在计算机元文件里。表 3-11 列出了 WINREG.H 里现成的七个类。REGEDIT 里缺少那个是 HKEY_PERFORMANCE_DATA，它包含了与系统性能有关的信息。

HKEY_LOCAL_MACHINE 和 HKEY_USERS 是注册表数据库里两个最基本的键。对于其他键来说，比如 HKEY_CURRENT_USER 和 HKEY_CLASSES_ROOT，它们只是简单地指向前两个键的某些子分支。例如，HKEY_CURRENT_USER 映射的就是 HKEY_USERS 的一个更加深入的分支。HKEY_CURRENT_CONFIG 则与 HKEY_LOACL_MACHINE 的 CONFIG 子键对应。

从根本意义上说，每个 Windows 95 组件都要用到注册表。用于启动计算机的进程、与网络的连接以及应用程序的执行都涉及到多个配置文件，它们之间有时还要进行一些交互作用。

所有这些信息都要以注册表为中心，并且只能通过 WINREG.H 里定义的一个特殊的 API 原型集进行访问。

表 3-11　Windows 95 注册表数据库里的基本键

键	值	说　明
HKEY_CLASSES_ROOT	((HKEY)0x80000000)	包含了把文件扩展名与应用程序联系起来的所有信息，以及用于实现和支持 OLE 功能
HKEY_CURRENT_USER	((HKEY)0x80000001)	当前登录到系统的用户参数和设置：环境变量、本地和远程磁盘、应用程序设计及其他信息
HKEY_LOCAL_MACHINE	((HKEY)0x80000002)	对系统组件和软件安装的说明，其中的系统组件包括网卡、显示器分辨率和类型、支持的文件系统以及键盘布局等等
HKEY_USERS	((HKEY)0x80000003)	用于设置新用户帐号的基本信息，以及存储以前登录到这个系统的所有用户的一个数据库
HKEY_PERFORMANCE_DATA	((HKEY)0x80000004)	列出的信息是通过“性能监视器”获得的
HKEY_CURRENT_CONFIG	((HKEY)0x80000005)	显示与硬件和软件配置有关的分支 HKEY_LOCAL_MACHINE\CONFIG
HKEY_DYN_DATA	((HKEY)0x80000006)	列出与所有即插即用设备有关的动态信息。这些信息将保留在系统 RAM 里，系统增减设备的时候就会刷新这些信息

对于开发者来说，他(她)应该多花些时间掌握注册表内键和值的用法，因为这对 Win95 应用程序的编制是颇有影响的。为了能在自己的软件包装上面光明正大地印上“与 Windows 95 兼容”徽标，程序的设置是需要首先掌握的一项关键技能。在过去，设置模块的编写是最能反映开发者想象力的一种活动，我们可以用各自不同的艺术思维来实现应用程序的安装。但是现在，微软已经提出了一些固定的规范，应用程序的安装和反安装都必须按照一定的准则来进行。

在 Win16 里，几乎每个 16 位应用程序都要在 WIN.INI 里增加一些数据，或者建立自己的 .INI 文件——Welcome 正是这样做的。在 Win95 里，开发者应该把以前存储在专用 .INI 文件里的所有信息移至注册表内的某些条目内。这样一来，用一个地方便可存储下关于每个应用程序和操作系统的全部信息。HKEY_LOCAL_MACHINE 分支也许是提供这种便利最恰当的一个区域。打开 Software 键和其他许多子键，我们最终能进入 Setup 键，在这儿列出了与 Win95 当前安装情况有关的许多有价值的信息，如图 3-7 所示。

Setup 子键里列出了超过 25 个不同的键；源路径报告了系统安装的原始位置。

```
HKEY_LOCAL_MACHINE
    SOFTWARE
        MICROSOFT
            WINDOWS
                CurrentVersion
                    Setup
```

我们下面列举了另外一个例子，这个例子能引起大家对注册表数据库及其中信息更浓厚的兴趣。Lan Man 列出了共享目录的所有名字。

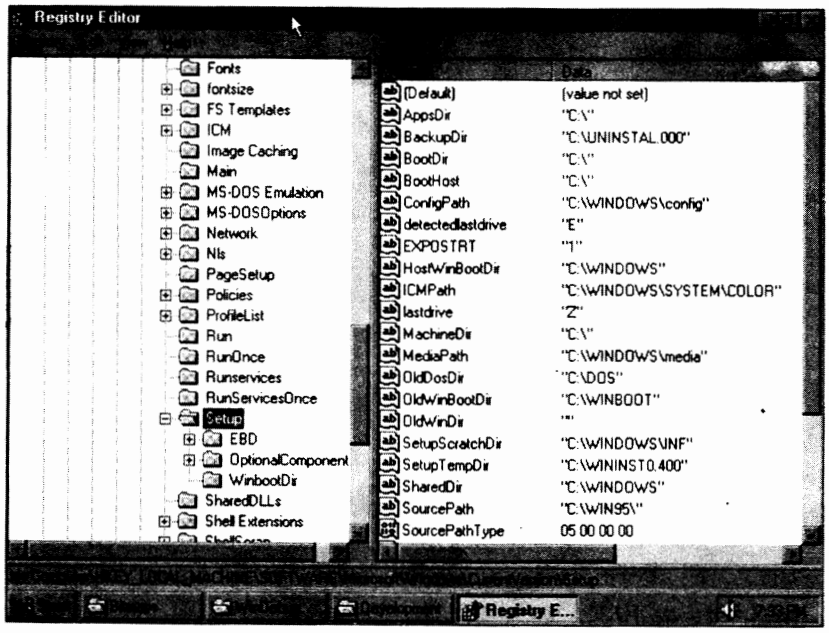


图 3-7　Setup 子键列出了与 Win95 当前安装情况有关的所有信息

HKEY_LOCAL_MACHINE
　SOFTWARE
　　MICROSOFT
　　WINDOWS
　　CurrentVersion
　　Network
　　Lan Man

现在让我们掉过头来继续探讨 Welcome 和它的 .INI 文件。假如把某些信息存储到注册表数据库里，这岂不是一桩妙事？心动不如行动，让我们马上开始！NT 提供了一种非常有价值的特性，它能把老式的 .INI 文件映射到注册表内带有 IniFileMapping 标记的一个特殊的段内。只需在注册表内增加一个简单的条目，以前的 WritePrivateProfileString()就可以映射到注册表里(对 GetPrivateProfileInt()以及这一类的其他所有函数均可如法炮制)。在 Windows 95 里，我们则不得不重新明确地编写代码，才能实现对注册表的访问。

首先，我们应该选择一个最佳的场所来放置新增的键。在 HKEY_CURRENT_USER\SOFTWARE 下，可以建立一个 Win32Book 键。这是一个新的分支，其中容纳了与本书例子有关的所有信息，如图 3-8 所示。

表 3-12 列出了全部这 26 个注册函数，其中一些在 Windows 95 里还没有实现。另外一些则仅限于 Windows NT 使用，在 Windows 95 中没有实现。

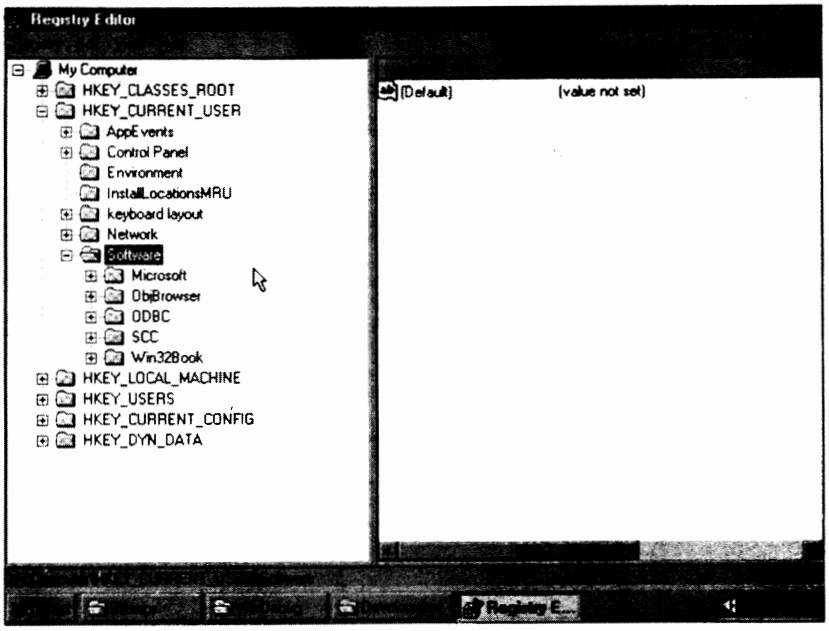


图 3-8　HKEY_CURRENT_USER 下面的 Software 子键

表 3-12　用于对注册表数据库进行访问和维护的注册表函数

函　数	说　明
RegCloseKey	关闭以前打开的一个键(对预定义键无效)
RegConnectRegistry	与不同系统内注册表的一个键连接
RegCreateKey	建立一个新键,如果该键已存在,则打开它(已经废弃的不用)
RegCreateKeyEx	建立一个新键,如果该键已存在,则打开它
RegDeleteKey	从注册表内删除一个键
RegDeleteValue	从注册表内删除一个值
RegEnumKey	列出某给定键的全部子键(已经废弃的不用)
RegEnumKeyEx	列出某给定键的全部子键
RegEnumValue	列出与一个键有联系的所有值
RegFlushKey	注册表数据文件内的拷贝,数据文件的所有值均与打开的键联系在一起
RegGetKeySecurity	返回安全描述符的一个拷贝,该描述符与一个键有关(Windows 95 尚未实现)
RegLoadKey	从一个文件载入信息,然后把信息存储到基本键 HKEY_USER 或者 HKEY_LOCAL_MACHINE 下面的一个新的子键内
RegNotifyChangeKeyValue	假如某个注册表键发生了变动,则发出通知
RegOpenKey	打开一个键(已经废弃不用)
RegOpenKeyEx	打开一个键
RegQueryInfoKey	返回与某个键有关的信息
RegQueryMultipleValues	返回来自一系列值内的信息,这些值在某个打开的键里
RegQueryValue	取回一个数据值(已经废弃不用)
RegQueryValueEx	取回一个数据值
RegReplaceKey	用一个新文件取代包含某个键和它的子键的文件
RegRestoreKey	从一个文件里读取键数据,然后把它们存储到注册表内
RegSaveKey	把键数据以及它的子键存储到一个文件里
RegSetKeySecurity	设置某个注册表键内的安全信息
RegSetValue	把键值存储为一个串(已经废弃不用)
RegSetValueEx	把键值存储为一个串
RegUnloadKey	从注册表内删除一个键以及它的子键,但是不从存储它们的物理文件内删除它们

大家现在可以对 SaveWindowPos()和 GetWindowPos()的新版本进行一番探究了。整体的逻辑并未发生任何变化，只是老式的 .INI 函数已经被注册表 API 取代了。

```
BOOL SaveWindowPos(HWND hwnd, LPRECT prc, HINSTANCE hInstance)
{
    char szKeyValue[] = "Windows size and position";
    char szKey[] = "Software\\Win32Book\\CHAP03\\LIST03";
    HKEY hkey;
    DWORD dwAction;

    // open/create the Software\\Win32Book\\CHAP03\\LIST03 key
    if(RegCreateKeyEx(HKEY_CURRENT_USER,
                      szKey,
                      0L,
                      NULL ,
                      REG_OPTION_NON_VOLATILE,
                      KEY_ALL_ACCESS,
                      NULL,
                      &hkey,
                      &dwAction) ! = ERROR_SUCCESS)
    {
        MessageBox(hwnd, "Error creating/opening the key", NULL, MB_
        OK);return FALSE;
    }

    // lets store the data
    if(RegSetValueEx(hkey,
                      szKeyValue,
                      0L,
                      REG_BINARY,
                      (BYTE * )prc,
                      sizeof(RECT)) ! = ERROR_SUCCESS)
    {
        MessageBox(hwnd, "Error saving data", NULL, MB_OK);
        return FALSE;
    }

                      // close the key
```

```
    RegCloseKey(hkey);
    return TRUE;
}
```

为了使 Welcome 能够访问注册表，必须在 HKEY_CURRENT_USER\SOFTWARE 下面建立 Win32Book 子键。SaveWindowPos()函数调用 RegCreateKeyEx()的目的就是在第一次运行 Welcome 的时候建立这个子键。此外，以后亦可利用它来打开该子键。RegCreateKeyEx()看起来相当复杂，但它在 Win95 里的实现却是很直接的。

```
#include <winreg.h>
LONG RegCreateKeyEx(HKEY hkey,
                    LPCTSTR lpszSubKey,
                    DWORD dwReserved,
                    LPTSTR lpszClass,
                    DWORD fdwOptions,
                    REGSAM samDesired,
                    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                    PHKEY phkResult,
                    LPDWORD lpdwDisposition);
```

参数	说明
HKEY hkey	某个已打开的注册表键的句柄
LPCTSTR lpszSubKey	准备打开的子键的名字
DWORD dwReserved	保留并且设置成零
LPTSTR lpszClass	包含了类名的缓冲区。假如该键已经存在，则忽略这个信息
DWORD fdwOptions	信息类型，可以是 REG_OPTION_VOLATILE，也可以是 REG_OPTION_NON_VOLATILE
REGSAM samDesired	安全访问标志(参见表 3-13)
LPSECURITY_ATTRIBUTES lpSecurityAttributes	指向某种安全属性的数据结构的指针，在 Win95 里为 NULL
PHKEY phkResult	一个 HKEY 标识符的地址，其中包含了打开的键的句柄
LPDWORD lpdwDisposition	一个双字的地址，作用是接收 REG_CREATED_NEW_KEY 或者 REG_OPENED_EXISTING_KEY
返回值	在正文里讨论
LONG	假如函数调用成功，返回 ERROR_SUCCESS；如果失败，则返回出错编号

第一个参数几乎总是用于标识一个打开的注册表键。七种预定义的键扮演了指向注册表的入口点的角色。这些键总是处于打开状态，并能通过其他函数访问。不管它的树结构如何，都不允许绕开前面所有的分级结构，从而直接访问一个已知的子键。这并不是一种严格的限制，但是开发者必须注意到访问注册表的基本操作是打开一个键。所以，利用七个预先定义好的键，我们就可以获取注册表数据库内的所有信息，这七键的地位是非常关键的。在 Welcome 里，新键放置于 HKEY_CURRENT_USER 分支下面，并且被标志为 Software\Win32Book\CHAP03\LIST03。这个串将用第二个参数传递给 RegCreateKeyEx()。假如函数调用成功，它就会把新键的句柄存储到 phkResult 参数里，这样就打开了新键，并且可以由其他函数进行访问。

表 3-13　访问许可属性

安全访问标志	值	说　明
KEY_QUERY_VALUE	(0x0001)	允许查询子键值
KEY_SET_VALUE	(0x0002)	允许设置子键数据
KEY_CREATE_SUB_KEY	(0x0004)	允许建立一个子键
KEY_ENUMERATE_SUB_KEYS	(0x0008)	允许枚举子键
KEY_NOTIFY	(0x0010)	允许改变通知消息
KEY_CREATE_LINK	(0x0020)	允许建立一个符号链接
KEY_READ	((STANDARD_RIGHTS_READ | KEY_QUERY_VALUE | KEY_ENNUMERATE_SUB_KEYS | KEY_NOTIFY)	允许读取一个键
KEY_WRITE	((STANDARD_RIGHTS_WRITE | KEY_SET_VALUE | KEY_CREATE_SUB_KEY)	允许写一个键
KEY_EXECUTE	((KEY_READ)	允许进行读访问
KEY_ALL_ACCESS	((STANDARD_RIGHTS_ALL | KEY_QUERY_VALUE | KEY_SET_VALUE | KEY_CREATE_SUB_KEY | KEY_ENUMERATE_SUB_KEYS | KEY_NOTIFY | KEY_CREATE_LINK)	所有操作均允许

如果想把窗口位置和尺寸存储到注册表内一个新的位置，和以前比较起来，现在需要采取的操作已经简单得多了。RegSetValueEx()允许开发者拷贝由 prc 指向的 RECT 结构内的全部 16 个字节，这仅需一个步骤即可完成。

```
#include <winreg.h>
LONG RegSetValueEx(HKEY hkey,
                   LPCTSTR lpValueName,
                   DWORD Reserved,
                   DWORD dwType,
                   CONST BYTE *lpData,
                   DWORD cbData);
```

参数	说明
HKEY hkey	一个打开的注册表键的句柄

LPCTSTR lpValueName	用于对值进行标识的标签
DWORD Reserved	被保留，必须设置成零
DWORD dwType	数据类型(列于表 3-14 内)
CONST BYTE * lpData	指向数据缓冲区的指针
DWORD cbData	数据缓冲区的大小
返回值	在正文里讨论
LONG	假如函数调用成功，返回 ERROR_SUCCESS；如果失败，则返回出错编号

在 SaveWindowPos () 里，第四个参数被设置成 REG_BINARY，用它以二进制格式存储窗口数据．紧跟在该参数后面的是指向 RECT 内存区域的 prc 指针．假如返回 ERROR_SUCCESS，就表明注册表函数调用成功．最后一个步骤是通过 RegCloseKey () 关闭新键．

```
#include
LONG RegCloseKey (HKEY hkey) ;
```

现在让我们来探讨一下 GetWinodwPos () 新的运用形式．这里需要涉及到两个注册表函数，它们分别是 RegOpenKeyEx () 和 RegQueryValueEx () .

```
BOOL GetWindowPos (HWND hwnd, LPRECT prc, HINSTANCE hInstance)
{
    char szKeyValue[] = "Windows size and position";
    char szKey[] = "Software\\Win32Book\\CHAP03\\LIST03";
    HKEY hkey;
    DWORD dwType, dwSize = sizeof (RECT) ;
    long lVal;

    // open the Software\\Win32Book\\CHAP03\\LIST03 key
    lVal = RegOpenKeyEx (HKEY_CURRENT_USER,
                         szKey,
                         0,
                         KEY_ALL_ACCESS,
                         &hkey) ;

    if (lVal ! = ERROR_SUCCESS)
    {
        MessageBeep (0) ;
        return FALSE;
    }

    // read the information
```

```
    if (RegQueryValueEx (hkey,
                         szKeyValue,
                         NULL,
                         &dwType,
                         (BYTE *) prc,
                         &dwSize) ! = ERROR_SUCCESS)
    {
        MessageBox (hwnd, "Error reading the values", NULL, MB_OK) ;
        return FALSE;
    }
    // close the key
    RegCloseKey (hkey) ;

    return TRUE;
}
```

在 GetWindowPos () 里，我们需要先打开应用程序键，然后再读取其中的内容．其中，两种基本的信息就是预定义的键句柄之一，以及子键的正确名字．正如我们以前提到的那样，子键名字看起来就像一个传统的路径名称，只是这儿不用在其中包含根键名罢了．子键存储在 RegOpenKeyEx () 的最后一个参数里．

```
#include <winreg. h>
LONG RegOpenKeyEx (HKEY hkey,
                   LPCTSTR lpszSubKey,
                   DWORD dwReserved,
                   REGSAM samDesired,
                   PHKEY phkResult) ;
```

参数	说明
HKEY hkey	一个打开的注册表键的句柄
LPCTSTR lpValueName	子键名
DWORD dwReserved	被保留，必须设置成零
REGSAM samDesired	希望设置的安全访问级别 (列于表 3-13 内)
PHKEY phkResult	一个 HKEY 标识符的地址，其中包含了子键句柄
返回值	在正文里讨论
LONG	假如函数调用成功，返回 ERROR_SUCCESS；如果失败，则返回出错编号

RegQueryValueEx () 与 RegSetValueEx () 是相当类似的．子键值是从注册表内读取出来

的，并且存储到一个 RECT 结构里.

```
#include <winreg.h>
LONG RegQueryValueEx (HKEY hkey,
                      LPTSTR lpszValueName,
                      LPDWORD lpdwReserved,
                      LPDWORD lpdwType,
                      LPBYTE lpbData,
                      LPDWORD lpcbData);
```

参数	说明
HKEY hkey	一个打开的注册表键的句柄
LPTSTR lpszValueName	对值进行标识的标签
LPDWORD lpdwReserved	被保留，设置成 NULL
LPDWORD lpdwType	表 3-14 内列出的其中一种数据类型
LPBYTE lpbData	数据缓冲区
LPDWORD lpcbData	一个标识符的地址，该标识符用于指定数据缓冲区的大小
返回值	在正文里讨论
LONG	假如函数调用成功，返回 ERROR_SUCCESS；如果失败，则返回出错编号

后五个参数是一系列指针的集合，这些指针都是以一个子键正文串标签起头的. 与 RegSetValueEx () 不同，数据缓冲区的大小必须用一个 DWORD (双字) 标识符进行传递，不能把这个标识符设置成 NULL.

表 3-14 注册表数据类型格式

数据类型	值	说 明
REG_NONE	(0)	没有值类型
REG_SZ	(1)	空的中断正文串
REG_EXPAND_SZ	(2)	空的中断正文串，带有对环境变量的非扩充引用
REG_BINARY	(3)	二进制格式
REG_DWORD	(4)	32 位数字
REG_DWORD_LITTLE_ENDIAN	(4)	32 位数字 (一个字最重要的字节就是其高位字)
REG_DWORD_BIG_ENDIAN	(5)	32 位数字 (一个字最重要的字节就是其低位字)
REG_LINK	(6)	符号链接 (Unicode)
REG_MULTI_SZ	(7)	由多个零中断正文串组成的数组，以一个附加的零字符结束
REG_RESOURCE_LIST	(8)	设备驱动程序资源列表
REG_FULL_RESOURCE_DESCRIPTOR	(9)	关于硬件描述的资源列表
REG_RESOURCE_REQUIREMENTS_LIST	(10)	

在本书附带 CD 的 Listing 3.3 里包含了能对注册表数据库进行访问的 Welcome 修订版本.

3.9 关于进程、窗口和实例

在 Win16 里，运行着的应用程序一般都称为“任务”. EnumTaskWindows () 和 GetWindowTask () 就是 16 位 API 的两个典型例子，它们与窗口和任务都是严格相关的. 这些函数在 Win32 里已经不再得到支持，“任务”的概念已经被“进程”取代了. 每个 Win32 进程都有一个唯一的 ID，这个 ID 是在进程启动时由系统自动生成的. 为了获得这种信息，我们应该使用 GetCurrentProcessId () 函数：

```
#include ‹winbase.h›
DWORD WINAPI GetCurrentProcessId (void) ;
```

进程 ID 是一种四字节长的数字，它唯一性地标识了某个进程（只要这个进程仍然存在）. 除此以外，任何进程都有一个唯一的句柄——通过调用 CreateProcess () 返回的值. 假如双击桌面上的一个图标，或者选择某个菜单项，从而启动了一个进程，我们就可以利用 GetCurrentProcess () 这个 API 函数来获取它的句柄.

```
#include ‹winbase.h›
HANDLE GetCurrentProcess (void) ;
```

Win32 还提供了两个类似的函数，它们能在线程级上调用，从而取回一个线程 ID 和线程句柄，这两个函数分别为：GetCurrentThreadId () 和 GetCurrentThread ().

```
#include ‹winbase.h›
DWORD WINAPI GetCurrentThreadId (void) ;

#include ‹winbase.h›
HANDLE GetCurrentThread (void) ;
```

线程 ID 也是一种具有唯一性的值，而线程句柄则只有在它的进程里才是唯一的. 线程 ID 继续维持 Win16 里某些 API 函数的关键，这些函数是以任务句柄——在 Win32 里几乎绝迹的一种概念为基础的. 举个例子来说，假如希望把一条消息传递给应用程序，而不是传递给窗口，就需要用到 PostThreadMessage () 函数，该函数的第一个参数就要求使用线程 ID（关于线程的详细讨论将在第 14 章“多线程、IPC 和 I/O”里进行）.

由 GetCurrentProcess () 返回的句柄其实是一个伪句柄. 什么是伪句柄呢？它表明自己只有对当前的进程才具有“可见度”，其他进程是无从知道它的. 有些时候，要求在原始进程以外都能知道一个进程句柄，当然，这种情况并不是经常出现的. 假如出现这种情况，开发者就可以通过调用 DuplicateHandle () 来“输出”一个句柄：

```
#include ‹winbase.h›
BOOL DuplicateHandle (HANDLE hSourceProcess,
                      HANDLE hSource,
                      HANDLE hTargetProcess,
                      LPHANDLE lphTarget,
                      DWORD fdwAccess,
                      BOOL fInherit,
```

DWORD fdwOptions);

参数	说明
HANDLE hSourceProces	原始进程的句柄
HANDLE hSource	准备复制的句柄
HANDLE hTargetProcess	目标进程的句柄
LPHANDLE lphTarget	用于接收输出句柄的句柄标识符的地址
DWORD fdwAccess	以句柄特征为基础的访问标志
BOOL fInherit	继承标志:假如目标进程能把复制句柄传递给一个新的子进程,则为 TRUE
DWORD fdwOptions	为零值,或者为 DUPLICATE_CLOSE_SOURCE 或 DUPLICATE_SAME_ACCESS
返回值	在正文里讨论
BOOL	布尔值,TRUE 代表函数调用成功,FALSE 代表失败

DuplicateHandle()可以对大量对象句柄进行处理,这些句柄包括 Mutex、相互排斥、管道、线程、进程、信号机、注册表键、文件映射以及事件等等.第14章"多线程、IPC 和 I/O"将对 Win32提供的所有 IPC 机制进行更深入的探讨.

刚开始进行32位编程实践的时候,我常常要面对的一个问题就是对进程进行跟踪,以及寻找那些相当害人的"阴魂不散"的应用程序——这种应用程序仍然保持运行状态,但它已经没有任何一个看得见的窗口.这样一来,我们就没有办法中断它,因为它在用户外壳上没有留下任何外观上的痕迹.Win95的 Process Viewer(进程查看器)程序——即 PVIEW95——是 Windows NT 里同一个程序的 Windows 95版本.这个程序自从1993年以来就一直在 NT 里提到了保留.该程序的 NT 版本可以通过一个特殊的键对注册表数据库进行查询,从而对运行着的应用程序进行跟踪,但是这个特殊的键在 Win95里却尚属于未采用之流.PVIEW95将利用 ToolHelp32这个 API 对正在运行的所有进程和线程进行跟踪.一旦具备了对某个进程句柄的访问权,中断它就是一种相当简单的事情,再也不用为"阴魂不散"的"僵尸"程序发愁了.PVIEW95的运行情况如图3-9所示.

一个非常有用的函数是 GetWindowThreadProcessId().它的作用是把开发模式的窗口处理部分与 Win32编程的系统部分链接起来.通过提供一个窗口句柄,这个函数可以同时返回线程和生成它的进程 ID.

```
#include <winuser.h>
DWORD GetWindowThreadProcessId(HWND hwnd, LPDWORD lpdwProcessId);
```

参数	说明
HWND hwnd	窗口句柄
LPDWORD lpdwProcessId	一个双字的地址,其中包含了进程 ID
返回值	在下面正文中讨论
DWORD	建立窗口的线程 ID

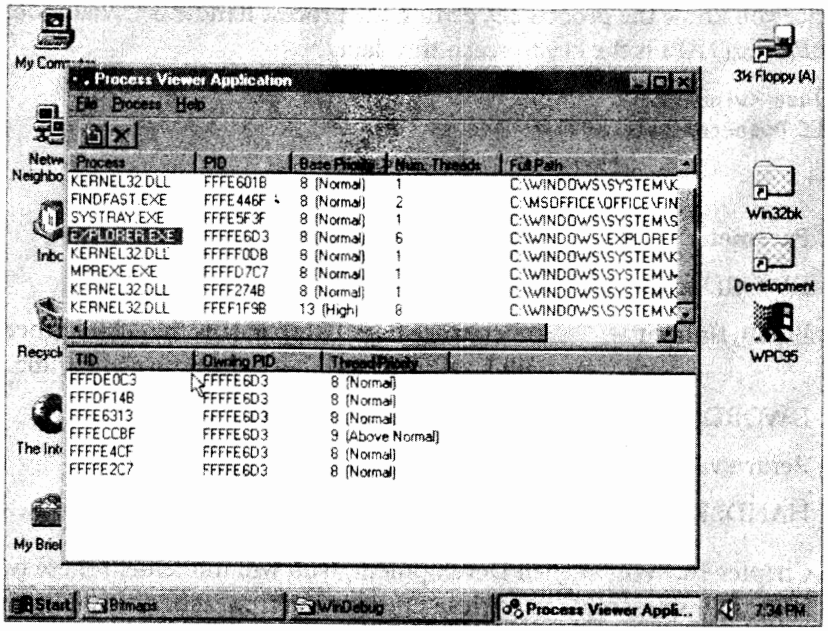


图 3-9　PVIEW95列出了正在运行的所有进程和与它们有关的线程

一旦知道了进程 ID，获取进程句柄就是举手之劳了。OpenProcess () 这个 API 是获得这种数据的关键。

```
#include ‹winbase.h›
HANDLE OpenProcess (DWORD fdwAccess,
                    BOOL fInherit,
                    DWORD IDProcess) ;
```

参数	说明
DWORD fdwAccess	访问类型
BOOL fInherit	继承标志：TRUE 意味着允许子进程继承返回句柄
DWORD IDProcess	准备打开的进程 ID
返回值	在下文中讨论
HANDLE	进程句柄

在第16章“Win95外壳的开发”里，我们将在 Winspy 例子里调用这两个 API，用它们从正在运行的其他应用程序里取回信息。一旦拥有了自己的一个进程句柄，就有义务结束它、提高它的优先权、把它挂起以及进行诸如此类的其他处理。TerminateProcess () 允许我们指定一个退出代码，从而中断一个进程。

```
#include ‹winbase.h›
BOOL TerminateProcess (HANDLE hProcess,
```

UINT uExitCode);

参数	说明
HANDLE hProcess	进程句柄
UINT uExitCode	退出代码
返回值	在下文中讨论
BOOL	布尔值，TRUE 代表函数调用成功，FALSE 代表失败

为了建立一个功能卓著的"杀手"应用程序——从根本上消灭"阴魂不散"的应用程序，GetWindowThreadProcessId()，OpenProcess() 和 TerminateProcess() 是我们必须采用的三种武器．一旦知道了进程的窗口句柄，很容易就可以它的进程 ID 和线程 ID．进程 ID 是进一步获得进程句柄的关键，它提供了对这个系统对象的完整控制．第16章"Win95外壳的开发"里的 Winspy 将利用这些 API 有选择地中断进程的运行．作为另外一种选择，通过在进程队列里传递一条 WM_QUIT 消息，PostThreadMessage() 就可以成功地取代TerminateProcess()．

如果想中断当前的进程，ExitProcess() 是我们的最佳选择——这是一种更简单、效率更高的函数．另一项优点与 ExitProcess() 如何同当前进程相连的 DLL 进行交互作用有关．利用 ExitProcess() 函数，当前的进程将执行每个 DLLMain() 函数里的分离例程．这种 DLLMain() 函数是与 TerminateProcess() 链接起来的，注意并不是在其中实现的．

```
#include ‹winbase.h›
VOID ExitProcess(UINT uExitCode);
```

参数	说明
UINT uExitCode	退出代码
返回值	在正文里讨论
BOOL	布尔值，TRUE 代表函数调用成功，FALSE 代表失败

开发者可以任意设置一个 uExitCode 值．这个值很少会被其他进程注意到，除非一个进程必须等待一个子进程才能中断．在 Win32里，衍生出一个新进程的情况并不多见，对此我们能找到很多个理由来解释．在这些理由中间，有一个是很重要的，那就是这种操作太费时了．

启动一个 Win32进程的时候，Win95会建立一个 Task（任务）数据库．任务数据库是一种典型的 Win16非文档化数据结构，它可用于管理任务、共享代码段以及在实例中对通用的数据域进行管理．软件开发包（SDK）里提供的 Heap Walker 程序可以列出正在运行的所有任务，其中包括 PVIEW95，如图3-10所示．

由 HeapWalker 显示的专用数据域内提到了应用程序菜单栏和与之相关的弹出式菜单．事实上，一些系统组件仍然是16位的，否则便无法提供与老式 Windows 代码的完全兼容能力．菜单堆就是这样的一种系统组件．

3.10 总结

到现在为止，大家应该对 Win32应用程序的整体布局有了一个清晰的印象，不应该再对其

有神秘和高深莫测的感觉. Welcome 例子为大家提供了一个很好的机会，利用这个机会可以分辨出 Win16和 Win32在 WinMain () 上的区别. 最重要的一点在于，我们知道了应该把应用程序特定的信息存储在注册表内，这才是解决问题的正确途径(关于这个主题，我们会在第10章"Windows 95通用控件"里进行更深入的学习，到时会有一个例子对注册表内的全部七个键都进行了探究).

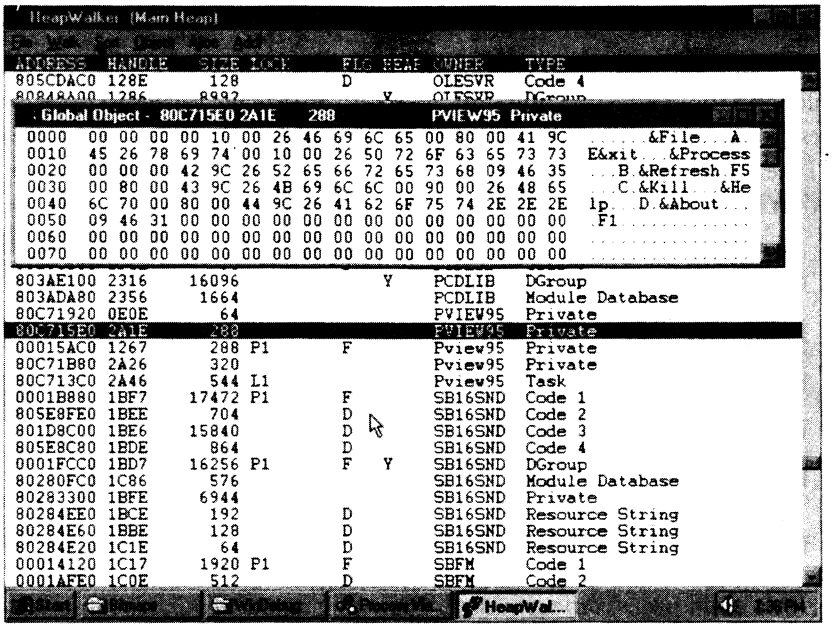


图 3-10 HeapWalker 列出了 Win32进程 PVIEW95. EXE:
选中的专用内存区域里包含了与菜单窗口有关的信息

第 4 章　消息和重画模式

消息在 Windows 环境和应用程序里扮演的角色是相当重要的。通过我们上一章进行的首次开发尝试，消息的作用应该不言而喻。更准确地说，Windows 里时刻传送的数百条 WM_消息流最终都会到达窗口进程，在那儿完成它们最终的处理。通过这些消息，应用程序可以“理解”系统里正在发生的事情，并且侦测到用户作出的选择。一旦知道了消息流，就可以指示 Windows 按照自己的需要行事，由此才能编写出紧凑和有效的代码。这一章的宗旨是加深读者对消息的理解，并学习 Win32 编程针对消息处理拟定的一些规则。

在第 3 章“Win32 应用程序的开发”里，我们知道一些消息是在应用程序专用的窗口进程里拦截下来的。迄今为止，我们对消息的产生机制都还没有一个准确的概念。我们只知道它们最终会抵达目标窗口的窗口进程。这在 99%的情况下都是正确的，所以我们可以把它当作一条普遍的规则使用。请注意，消息同样也可以定址给线程队列。

窗口进程是一段特殊的代码，它通常都是由应用程序或者系统本身在需要发送和处理一条消息的时候调用的。WinMain () 只是简单地具备了源代码入口点的地位，其中可以对一些初步操作进行设置。然而，为了对按下鼠标按钮、击键以及其他许多种事件进行响应，开发者还是需要在各种不同的窗口进程里编写对应的代码。

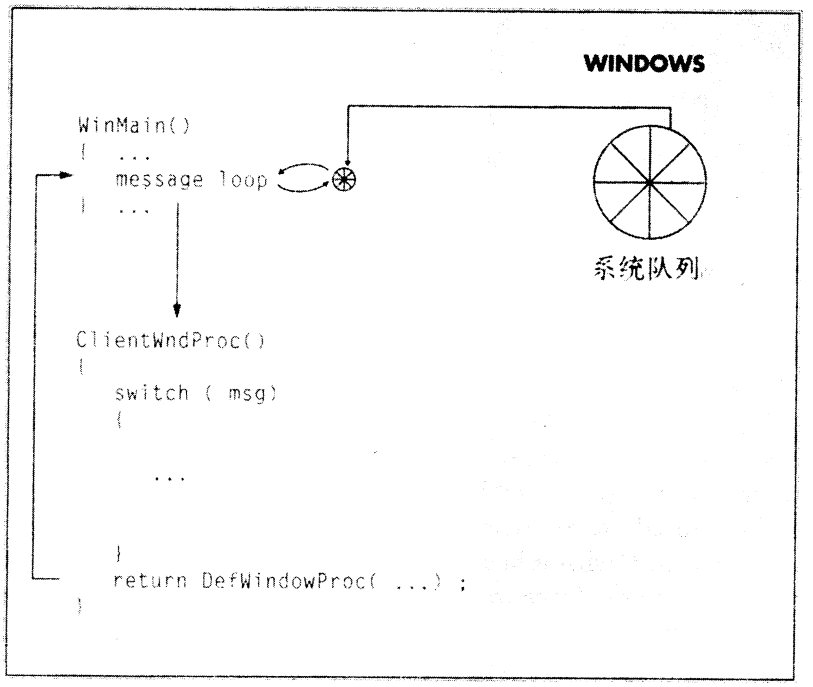


图 4-1　消息先在应用程序队列内张贴，然后传递给合适的窗口进程——消息的运动流程

抵达某个窗口进程的消息可能来源于几个不同的起点。正如我们通过第 3 章“Win32 应用程序的开发”就已知道的那样，假如用户按下 A 键，Windows 本身也会在应用程序队列里张贴一条消息。一个 Win32 进程本身也有可能生成一条消息，使这条消息定址于它的某个窗口。图 4-1 为大家总结了消息传递的标准路径，这是以我们的现有知识为基础描绘的。

每个应用程序都必须使用 GetMessage（）不停地等待消息到来。以 MSG 数据结构内 hwnd 项的目标窗口句柄为基础，DispatchMessage（）函数标识了需要跳转的正确窗口进程。

DispatchMessage（）现在必须以进入消息的相关信息为基础，否则便无法选择正确的窗口进程。它的语法只要求指向某个 MSG 结构的地址，那个 MSG 数据结构内的信息是在前面由 GetMessage（）填写的。

```
LONG WINAPI DispatchMessage (CONST MSG *lpMsg);
```

参数	说明
LPMSG lpMsg	指向一个 MSG 数据结构的地址
返回值	在下文中讨论
LONG	由 DispatchMessage（）调用的窗口进程的返回值

更准确地说，DispatchMessage（）应该返回一个类型为 LRESULT 的值，这是典型的窗口进程应该返回的一种类型。考虑到 LRESULT 除了一个简单的长数以外就别无是处，所以它影响是很有限的。

DispatchMessage（）怎样选择适合的窗口进程呢？注意，它要选择的窗口进程必须与特定的窗口类联系在一起，当前消息从属的目标窗口就是属于这个窗口类的。到目前为止，我们还没有掌握所有相关的信息，所以暂时无法从技术角度明确提供一个天衣无缝的答案。但我们至少可以作出一个合理的猜想。我们已经知道每条消息都要定址给某个窗口，因为这种信息肯定存在于 msg. hwnd 项的 MSG 数据结构里。理解了这一点，我们就向正确答案迈出了第一步。

除此以外，任何窗口都是从属于某一窗口类的。当我们注册某个类的时候，由于 WNDCLASS 或者 WNDCLASSEX 的本质如此，所以有义务把函数的名字指定为类窗口进程。与一个类（这个类某个窗口有关联）有关的所有信息总是存储在 USER32. EXE（即 Windows 的替身）的数据部分里的。DispatchMessage（）把让人伤脑筋的所有信息都放到了一块儿。在已经知道目标窗口句柄的前提下，DispatchMessage（）会“有礼貌地”询问窗口是属于哪一类的。通过类信息，DispatchMessage（）可以取回窗口进程地址，然后跳转至那个地址处。在第 7 章“建立窗口的艺术”里，我们就能在 DispatchMessage（）函数的基础上定义出客观和正确的调用机制了。

执行流只有在一种条件下才会传送给某个窗口进程：存在一条消息，这条消息被定址给从属于那个特定类的某个窗口。窗口进程里的四个参数与一个 MSG 结构的前四个项是分别对应的。这就意味着访问一个窗口进程的时候可以使用下面这些信息：

- ▶目标窗口的句柄
- ▶消息本身
- ▶wParam 和 lParam 里的附加信息

我们从中可以获得一些有趣的暗示。Welcome 里那个唯一的窗口是在 WinMain () 里建立的。hwnd 句柄只有在这个函数里才是“可见”的。然而，应用程序逻辑是以窗口进程为中心的。我们如何才能从一个窗口进程里访问窗口句柄呢？许多开发者都是把主窗口句柄声明为一个标识符，并且使该标识符的源文件范围为“全局”，从而解决了这个问题。

这种方式把句柄扩展成在整个代码范围内都具有“可见性”。说句也许有点言重的话，出于多方面原因的考虑，这也许算得上编写 Windows 应用程序代码最糟糕的一种方式了。现在让我们详细解释这一点。对于用户在 Welcome 主窗口里的每一项选择来说，Windows 都会进行处理，并且生成一条或者多条消息，这些消息最终都会传送到属于那一类的窗口进程里。执行流程抵达一个窗口进程的时候，第一个参数肯定指出了目标窗口的句柄，这就是 Welcome 那个唯一窗口的 hwnd。

这样一来，把窗口句柄声明成源文件范围内的标识符就显得毫无用处了，因为我们在一个窗口里随时都可以访问这种信息。假如在同一个应用程序里建立了多个窗口，而且这些窗口属于一个或者多个窗口类（如图 4-2 所示），此时这条规是同样适用的。一个窗口进程一次只能处理一条单独的消息，同时让属于那一类的某个窗口作为自己的最终目的地。在抢先式多任务 Win32 环境下，这种情况并没有发生变化，其中缘由在下面的段落里就要讲述到。

大家应该记住，所有窗口都会保留一种亲密的“父子”关系。一旦我们知道了一个窗口句柄，很容易便能在分级树里追根溯源，把最远的祖先“挖”出来。

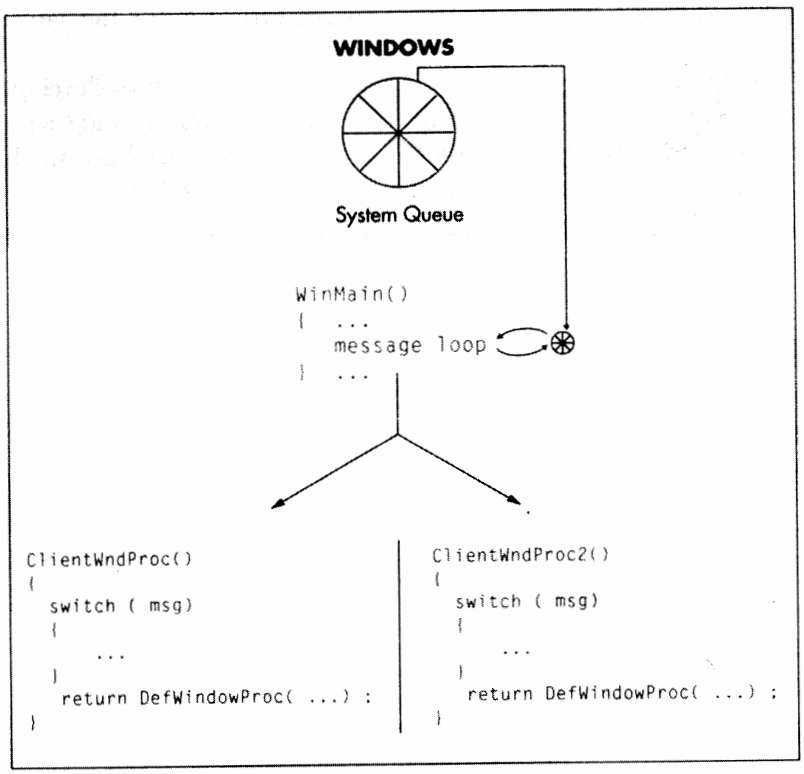


图 4-2　系统一次可以执行一个独立的窗口进程，它能接收目标窗口的句柄

判定旁系窗口和它们各自的子窗口也是一种非常直接的操作，这时只需要调用几个

Win32 API 函数就可以了。所以，从一个普通的窗口句柄开始，我们可以找出当前正在运行的所有窗口的句柄，而不管它们属于谁的进程。这正是我们在诸如 WINSPY 的侦查程序（参见第十六章“Win95 外壳的开发”）以及在 SDK（软件开发包）提供的 Spy 和 Spy++程序里采用的编程基础。

通常，对于一个标准的 Windows 应用程序来说，它很少有机会去判定与其他进程里建立的窗口有关的窗口句柄。但是，这种异想天开的想法却让我们有机会接触 Win32 编程一个有趣的问题。我们知道，每个 Win32 进程都有它自己专用的地址空间。和 Win16 不一样，通信活动在 Win32 里的介入显得很不自然。假如我们从 Windows 应用程序的内部特征着手来观察它——这些特征包括内存寻址、进程间通信（IPC）、多任务、进程 ID（PID）和线程 ID（TID）等等，那么前述的观点就不言而明了。但从另外一方面来说，如果把 Win32 进程简单地当作窗口来看待，情况就变得迥然相异了。这时，横亘在不同进程间的障碍会变得模糊不清。通过不同的 API，比如 FindWindow（），我们可以取得与另外一个进程有关的信息，这样就绕开了进程间那种标准的障碍。所以，我们设计出这样的一个应用程序是可行的——要求它取回与另一进程有关的窗口句柄，然后改变它的某些属性。想想看，假设我们用一个很小的 Win32 进程去取回 Microsoft Excel 的主窗口句柄，然后改变它的属性，使别人再也无法退出 Excel，读者认为这个主意怎么样呢？当然，尽管这完全可行，但却并非一个非常明智的例子。

4.1 关于消息

到这时为止，我们已经熟悉了与窗口处理有关的 WM_消息，这种消息会张贴到应用程序消息队列内、稍后由 GetMessage（）取回，然后利用 DispatchMessage（）传递给对应的窗口进程。WM_消息抵达一个窗口进程之前也可以绕过应用程序消息队列。从窗口进程的角度来看，两种选择方案并没有什么区别：最后的处理都只能依赖于消息的本质。

从另外一方面讲，对于 Windows 开发者来说，真正理解两种方案的区别和实现方式是非常重要的，只有这样才能编写出高效率、易于辨读的代码。图 4-3 阐明了消息是如何达到一个窗口进程的。

4.1.1 消息的张贴

当消息放置在一个应用程序的消息队列里时，我们就可以说它是一条“被张贴的”、“正在排队的”异步消息。事实上，消息张贴到队列里以及它后来被 GetMessage（）取回，这两个行动之间总是存在时间延迟的。

和 Win16 不一样，Win32 进程消息队列的规模是根据张贴消息的数量动态增长的（所以，永远都不可能发生消息丢失的情况）。挑选出某条消息是一种速度非常快的操作。但从另外一方面来讲，谁都无法准确预测出对应用程序窗口进程中的消息进行处理要花多长时间。对于一名优秀的 Win32 程序员来说，他（她）的目标是不管当时存在多少个线程，消息处理的时间都要尽可能地缩短——这样才能让应用程序对用户的选择及时作出反应。

应用程序消息队列内张贴的消息既有可能来自 Windows，也有可能来自其中一个正在运行的进程。为了使问题简化，让我们把张贴消息的来源限制在同一正在运行的应用程序里。

正如大家现在已经知道的那样，用户按下一个键，Windows 就会张贴出一条消息。能得到同样结果的其他操作包括鼠标运动（WM_MOUSEMOVE）、按下鼠标键（WM_

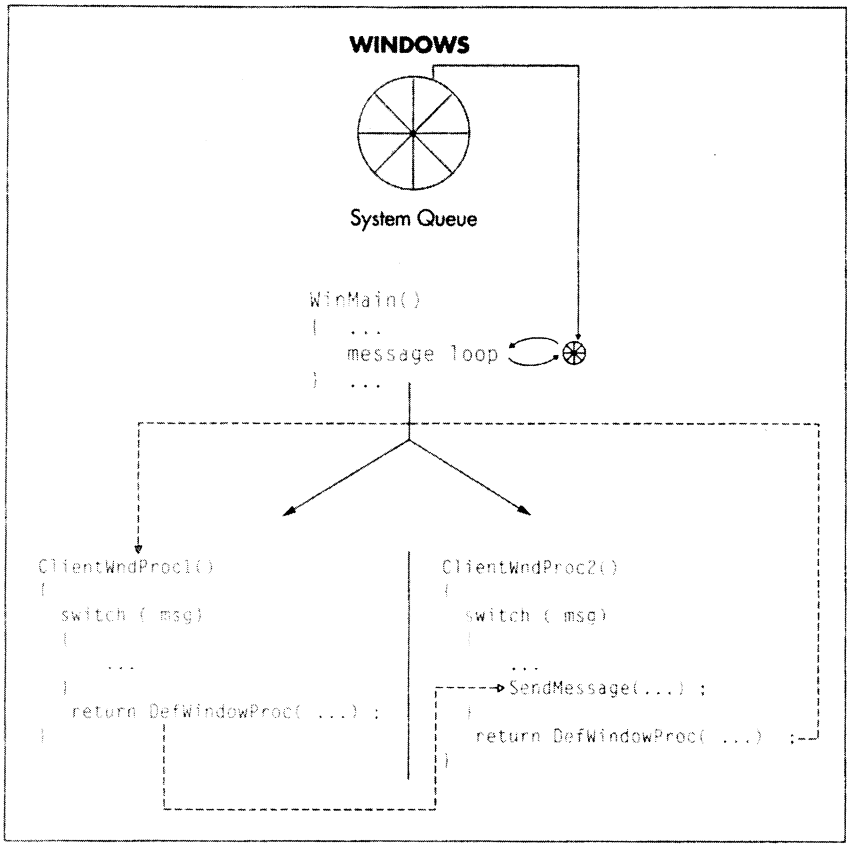


图 4-3 一条 WM_消息可以经由应用程序消息队列到达窗口进程，也可以直接跳至处理函数

xBUTTONDOWN)、客户窗口刷新（WM_PAINT）或者进程中断（WM_QUIT）等等。在到达某个应用程序消息队列之前，这些事件将临时性存储在系统消息队列里。根据原始的输入线程，它们最终都会发送到恰当的位置。除 WM_PAINT 以外，消息在队列中的张贴都是按一种精确的 FIFO（先进先出——First-in，first out）逻辑进行的。

尽管很容易便可定义生成一条张贴消息的原因，但是应用程序为什么和什么时候张贴消息却并非那么明显的的。简而言之，假如当前操作并不是按照固定顺序完成的，而且应用程序要求在当前操作之后再执行某种操作，这时就需要把一条消息张贴到进程消息队列里。下面让我们详细解释刚才的话。毫无疑问，张贴消息之前需要先使用一个重要 API 函数：PostMessage ()。

```
#include 〈winuser. h〉
BOOL WINAPI PostMessage (HWND hwnd,
                         UINT msg,
                         WPARAM wParam,
                         LPARAM lParam);
```

参数	说明
HWND hwnd	目标窗口的句柄
UINT msg	准备张贴的消息
WPARAM wParam	关于消息的附加信息
LPARAM lParam	关于消息的附加信息
返回值	在正文里讨论
BOOL	布尔值，TRUE 代表函数调用成功，FALSE 代表失败

PostMessage（）的四个参数与普通窗口进程的参数是完全一致的。我们对于这一点不应再感到奇怪，因为张贴的消息迟早都会取回，并且传送给合适的窗口进程内。

PostMessage（）可以在应用程序源代码的任何地方调用，唯一的限制就是需要一个可用和有效的目标窗口句柄。这个条件对于 WinMain（）和普通的窗口进程来说都是适用的。

对于 Windows 应用程序来说，考虑到 WinMain（）主要扮演的是一个初始化模块的角色，所以一般都不把 PostMessage（）放置到 WinMain（）里。请大家注意窗口进程在建立进程逻辑过程中的核心地位，在某个窗口进程里调用 PostMessage（）函数还是不错的。对这两个概念进行综合考虑，我们最后决定在一条 WM_消息块代码里调用 PostMessage（）函数。

这就是我们编程思想。对一条特定的 WM_消息进行处理时，要求在应用程序队列里放置一条 WM_消息。到底放置哪条消息呢?哪条都行!它可以是正在处理的同一条消息，也可以是 SDK 提供的其他任何一条消息。这里根本就没有任何限制，只是注意不要把正在处理的相同消息张贴到相同的窗口内，否则就会得到“堆栈溢出”的出错信息。当然，对正在处理的同一条消息进行张贴也是允许的，只是它的目标窗口应为一个不同的窗口。

一个张贴的例子

大家也许还能回忆起来，Welcome 里包含了张贴消息的第一个例子。当时，WM_CLOSE 消息调用了 PostQuitMessage（），这是在后台隐匿了 PostMessage（）的一个宏函数。还记得吗，源代码里含有下面这条语句：

```
PostQuitMessage (0);
```

系统把它解释为：

```
PostMessage (hwnd, WM_QUIT, 0, 0);
```

PostQuitMessage（）是到到目前为止 PostMessage（）最常见的一种用法，无论从表面还是从含蓄的角度来看都是如此。通常，消息的张贴要符合开发者在特定环境下（比如消息 DDE）的需求。整体逻辑可以确保应用程序正常处理以后经过目标窗口的一条特定消息。更为常见的一种情况是：由于发送了某条消息，所以触发了代码段内一个特定的部分，应用程序的执行流程将立即转向这个部分，这就是在图4-3里演示的第二种情况。

4.1.2 消息的发送

消息的最终接收者几乎一定是某个窗口，尽管对消息的处理是在类窗口进程里完成的。在窗口进程不同的切换代码块里，不同的条件一次只能选择一种。假如 WM_CREATE 抵达了一个窗口进程，那么只有代码的那个部分才能执行，相同的规则也适用于 WM_SIZE 和剩余的其他消息。只有与那条消息有关的代码块才会执行，执行流程将从入口点开始，直到遇到一条中断或者返回语句才会结束。

为了解释这个概念，我们依据一种虚构的时间量度对此进行一番说明。假如在时间0时接收了一条 WM_CREATE。只有在执行完 WM_CREATE 消息后，我们才能执行一条 WM_SIZE，这时正好是时间1。类似地，可以想象在时间2时接收到了一条 WM_PAINT 消息。这个例子假设0,1和2之间的时间间隔等于完成每条消息处理所需的时间。尽管从物理意义来看，这些代码段是包含在同一个窗口进程里的，但是我们可以把它们考虑成独立的代码部分，每个部分最终都要实现一种特定的任务。处理另一条 WM_消息的时候，我们可以调用与另一条消息有关的代码段，对于这一点是没有什么限制的。下面让我们用一个实际的例子来验证这种编程技术。图4-4里包含了一个典型的 Win32对话框，它专门用于对文件系统进行访问。

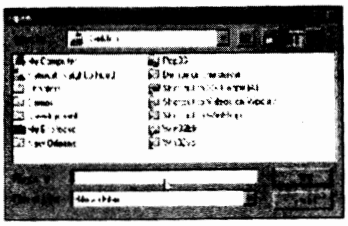


图 4-4　一个典型的 Win32通用对话框——用于访问文件系统，并能打开一个或多个文件

为了把文件载入应用程序内，我们可以选用两种不同的策略：双击文件名，或者先选定（单击）一个文件名，然后按下 Open 按钮。这两种操作将各自生成不同的消息流，但是最终的目的都是一样的：载入选中的文件。在第8章“Win32的对话框管理”里，我们将深入探讨通用对话框在 Win32应用程序里的运用。

为了获得最佳的解决方案，我们需要把获得选中文件名以及载入这个文件所需的代码集中到一个统一的地方，这个地方应该与两个事件的其中之一对应（按下 Open 按钮）。假如程序侦测到了对话框列表视窗部分内发生的鼠标双击事件，那么把与“按下 Open 按钮”对应的消息传送给窗口本身就已足够了。通过针对这两种情况编写一段独立的代码，对界面处理两种不同的办法我们都能感到满意了。这样就减少了冗余代码，并且限制了错误和执行异常的机会。如果想发送一条消息，需要用到 SendMessage（）函数：

```
#include〈winuser.h〉
LRESULT WINAPI SendMessage（HWND hwnd，
                            UINT msg，
                            WPARAM wParam，
                            LPARAM lParam）；
```

参数	说明
HWND hwnd	目标窗口的句柄
UINT msg	准备发送的消息
WPARAM wParam	关于消息的附加信息
LPARAM lParam	关于消息的附加信息

返回值	在正文里讨论
LRESULT	发送消息抵达的窗口进程的返回值

其中的参数与 PostMessage（）是完全是一致的，只是返回值是 LRESULT，这通常是由一个窗口进程返回的。

发送的消息是同步处理的，换句话说，发送的消息立即就会得到处理。和张贴消息比较起来，其间的差别是相当吻合的。在这种新的情况下，一条发送出去的消息会临时性中断对某条特定消息的处理，转而执行与发送消息有关的代码段。只有从目标窗口返回以后，程序的执行流程才会恢复到紧跟在 SendMessage（）后面的语句。和往常一样，发送消息的接收者也是一个窗口。这个窗口可能正忙于处理应用程序其他任何一个窗口内的另外一条消息（注意，不要把某窗口进程内处理的同一条消息发送给相同的窗口，否则就会得到堆栈溢出错误）。正如我们已经知道的那样，假如某窗口属于一个单独的进程，那么把消息发送给这个窗口是完全允许的。这样一来，利用消息的发送，我们就可以实现进程间通信（IPC），一种比较低级的形式。关于这方面的主题，我们以后会作更详细的论述。

消息的发送是非常频繁的，因为 Win32程序的开发就是以这种途径为基础的。在目前，为了使问题简化，我们假设消息只发送给属于相同线程的窗口。

消息的自动发送

在一个窗口进程里，不同的情况（case）块看起来就像一段段独立的代码，因为它们的注意力完全放在完成各种特定的任务上面。例如，WM_SIZE 块内的代码通常用于对窗口的位置和尺寸进行定义。

正在处理一条不同消息的时候，假如应用程序逻辑要求对窗口尺寸重新进行定义，这时会发生什么情况呢?应该对 WM_SIZE 消息内的代码进行复制，这样才能实现要求的目标吗?显然不是!最好的方法是在提出重新定义尺寸的要求时发送出一条 WM_SIZE 消息，等待操作中断，然后恢复执行临时中断的消息块代码。在这种情况下，目标窗口最有可能是正在处理另一条消息的同一个窗口。所以，消息流就会强制系统退出当前的窗口进程，用一条不同的消息（本例中是 WM_SIZE）重新进入那个窗口，执行与之相关的代码，离开窗口进程，然后从紧跟于 SendMessage（）后面的那条语句处恢复执行。如图4-5所示。

4.1.3 把消息发送给同一类的窗口

设计用户界面的时候，普遍采用的一种叫作 MDI（多文档界面）的模式。MDI 应用程序一般都是在 Windows 3.x 下运行的，它们不大适合在 Windows 95下面运行。这种应用程序有一个主叠置窗口，另外还有包含于主窗口客户区内的许多次级窗口。

根据这种整体结构来推算，文档窗口肯定是一个接一个关联在一起的。在这种情况下，应用程序本身通常都把一条消息发送给所有文档窗口，使它们都具备与当前情况相关的恰当行为。举个例子来说，假设用户中断了应用程序，所有文档窗口就都会接到关于这一情况的通知。

4.1.4 把消息发送给其他类的窗口

在现实的编程环境中，消息也经常需要发送给其他类的窗口。在这种情况下，由于目标窗口从属于一个不同的窗口类，所以需要由 SendMessage（）执行第二个窗口进程。

调用 SendMessage（）与调用一个函数是类似的。唯一的区别在于 SendMessage（）和 DispatchMessage（）一样，它也需要通过目标窗口句柄来标识适当的窗口进程，从而实现非

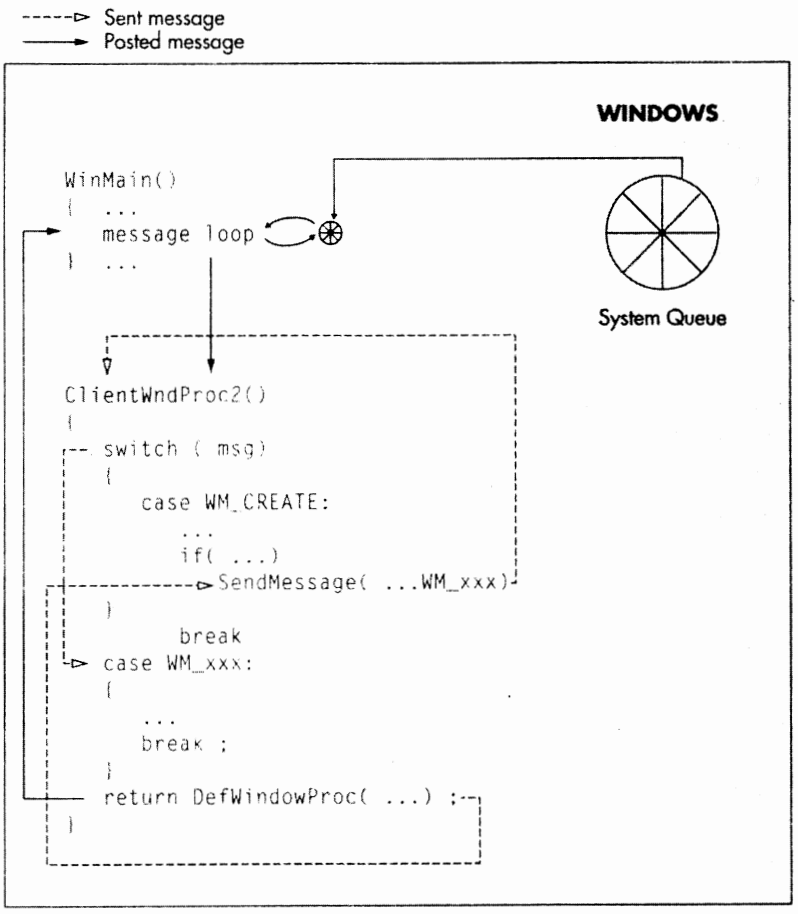


图 4-5　把一条消息发送给正在处理另外一条消息的相同窗口：Windows 编程中可行和常见的一种编码方式

直接性的选择。

4.1.5　消息发送的实践

只要熟悉了 Windows 95的用户界面 (UI)，就知道关闭一个窗口有多么容易:这通过几种 UI 组件都可以实现，比如系统菜单内的 Close 菜单、标题栏最右方的按钮以及 File 弹出式菜单内的 Exit 菜单项等。无论具体怎样操作，它们最终都要以相同的代码为基础。我们无法百分之百保证所有这三个消息流都会进入一个单一的代码区，但出现这种情况的机率是相当大的。不管微软的工程师如何编写 Windows 的这段代码，我们知道任何一种“关闭窗口”的行动都会引发一条 WM_SYSCOMMAND 消息（注意：选择菜单项通常引发的是一条 WM_COMMAND 消息，而不是 WM_SYSCOMMAND)。

假设我们在进程中断执行前必须先进行一番清除处理。考虑到有多种方式均可退出应用程序，所以把对应的清除代码段集中到一个独立的地方是比较明智的。最佳的办法是把所需的全部指令放置在 WM_SYSCOMMAND 消息的 SC_CLOSE 选项内。这条 WM_SYSCOMMAND 消息是用户与系统菜单交互作用时产生的。因为 Exit 菜单项会生成一条与

众不同的 WM_COMMAND 消息，所以最后编写出来的代码就是下面这个样子：

```
...
case WM_COMMAND:
    swithc (LOWORD (wParam))
    {
        case MN_EXIT:
            SendMessage (hwnd, WM_SYSCOMMAND,
             (WPARAM) SC_CLOSE), 0);
            break;
        ...
    }
    break;
...
case WM_SYSCOMMAND:
    switch (wParam)
    {
        case SC_CLOSE:
        {
            ...
        }
            break;
        ...
    }
    break;
...
```

假如用户选择 Close 菜单项、双击应用程序图标或者按下标题栏最右方的那个按钮，从而中断应用程序，程序这时就会接收到一条 WM_SYSCOMMAND 消息。如果选择的是 Exit 菜单项，就会有一条 WM_COMMAND 消息抵达窗口进程。在哪儿，立即会生成一条 WM_SYSCOMMAND 消息，用它维持程序的整体逻辑不致混乱。

总而言之，消息的发送是非常频繁和有用的，对消息张贴的需求则比较少。通过把代码分布到不同的情况（case）块内，就可以使应用程序的整体结构得以优化，因为一般都能保证恰当的指令放置到恰当的地方。SendMessage () 的作用则相当于各种独立部分间的一种胶粘剂，它能根据实际需要把这些 case 块有机地结合起来了。

4.2 窗口和消息

本章的第一个例子是明显针对消息处理而设计的，它的名字叫作 WINMSG（参考本书附带 CD 的 Listing 4.1）。如图4-6所示，这个程序注册了两个窗口类，并且建立了三个互相联系

的窗口。主窗口是在 WinMain () 里建立的唯一一个窗口，该窗口属于 WINMSG 类，它占据了上方的位置。另外两个窗口则都从属于第二个窗口类：TEST，所以它们共享同一个窗口进程。

主窗口里提供了一个菜单栏，里带有两个顶级菜单：Send（发送）和 Post（张贴）（关于菜单栏的详细情况，我们将在第6章“菜单的运用”里介绍）。Send 里只有一个菜单项：Change colors（改变颜色）。该菜单项的用途是强制 Test1和 Test2的客户窗口背景分别从白色变成红色和蓝色。应用程序还注册了一条伪消息：WM_CHANGECOLOR，并且把它发送给两个测试窗口：

```
...
// change color to red in Test 1
SendMessage (hwndTest1, WM_CHANGECOLOR, (WPARAM) 5, 0);
// change color to blue in Test 1
SendMessage (hwndTest1, WM_CHANGECOLOR, (WPARAM) 2, 0);
```

该消息的 wParam 部分包含了一个特殊的数字，它代表了基本颜色的浓度，具体的方案如下：

wParam	颜色
1	红色
2	绿色
4	蓝色

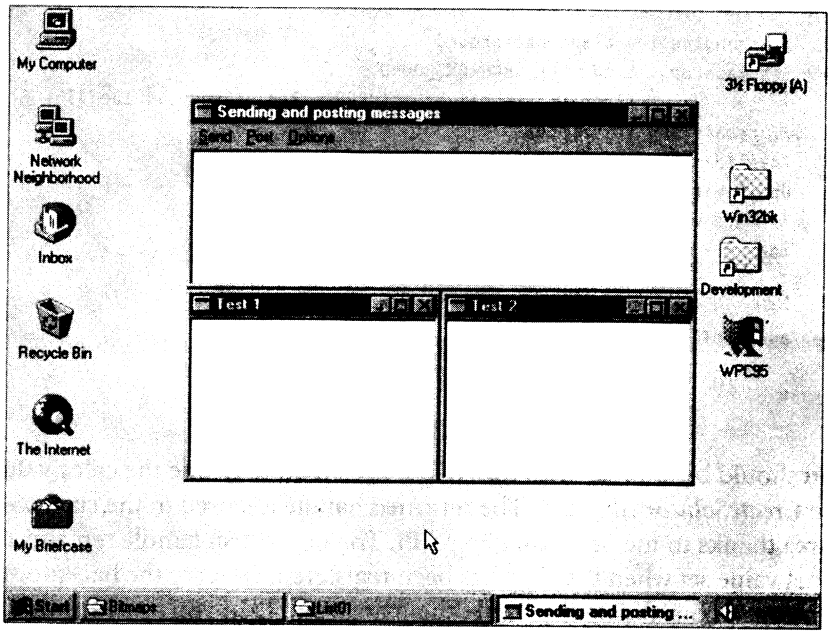


图 4-6　WINMSG 是三个不同窗口的组合，这些窗口分别属于两个不同的窗口类

所以，第一次调用 SendMessage () 的时候选择了红色和蓝色，而第二次调用则选择了纯绿色。之所以把 WM_CHANGECOLOR 称为一条伪消息，是因为它只是一个简单的数值定义，而且是以 WM_USER 这个 Win32 API 函数里最后一个使用的值为基础的：

```
#define WM_CHANGECOLOR WM_USER + 1000
```

在 TestWndProc () 里，这条消息将被拦截下来，然后对窗口背景的颜色进行变动：

```
...
case WM_CHANGECOLOR:
{
    int i, j;
    int iClrSel = (int) wParam;
    int iRed, iGreen, iBlue;

    for (i = 0; i < 256; i++)
    {
        iRed = i * (iClrSel & 1);
        iGreen = i * (iClrSel & 2) / 2;
        iBlue = i * (iClrSel & 4) / 4;

        //changing the class background
        SetClassLong (hwnd, GCL_HBRBACKGROUND,
                      (long)CreateSolidBrush(RGB(iRed, iGreen, iBlue)));
        //repainting the window
        InvalidateRect (hwnd, NULL, TRUE);
        UpdateWindow (hwnd);
        //lets waste some time
        for (j = 0; j < 100000; j++)
            ;
    }
    MessageBeep (0);
}
    break;
...
```

在 CreateSolidBrush () 里，大家应该已经熟悉了用于定义颜色值的 RGB 宏。返回句柄是在 SetClassLong () 这个 API 函数的帮助下存储到类内存区域里的。假如类已经注册好了，新的刷子句柄就会取代原始的颜色值。为了强制进行背景重新重画，必须先使整个客户区域无效，然后调用 UpdateWindow ()，从而使背景重画同步进行。所有这些操作都会在本章的后面部分进行深入的探讨。程序的运行结果如图4-7所示。

为了放慢颜色的计算速度，程序里设置了一个 for 循环，从而起到了时间延迟的作用。在整个处理过程中，系统对主窗口内发生的任何交互作用都会不加理睬。这就再一次证明了当发送者和接收者都属于同一线程时，发送消息的“同步”本质（关于线程间消息处理的问题，我们将稍后在本章详述）。

Post 这个顶级菜单允许用户激活 Test1或者 Test2窗口，方法是在应用程序消息队列里张贴一条 WM_ACTIVATE 消息，然后选择恰当的目标窗口句柄。

```
...
PostMessage (hwndTest2, WM_ACTIVATE, MAKEWPARAM (WA_ACTIVE,
    0), (LPARAM) hwnd);
...
```

能用 SendMessage () 去激活一个窗口吗?可以，而且工作得很有效。但从我个人的角度来看，我认为这是分配给 PostMessage () 的一种典型任务。

WINMSG 还提供了另外一项非常有趣的功能。观察一下移动主窗口时发生的情况。首先，拖动单独主窗口的过程中，大家也许会注意到主窗口的框架正好是三个窗口加在一起的尺寸，如图4-8所示。也就是说，和通常想象的比较起来，那个边框的尺寸要大一些。其次，结束拖动操作后，三个窗口全都在目标位置处显示出来了，并且保留了它们以前的位置对应关系。从本质上看，移动主窗口的时候，另外两个窗口好像是和主窗口粘在一起似的。

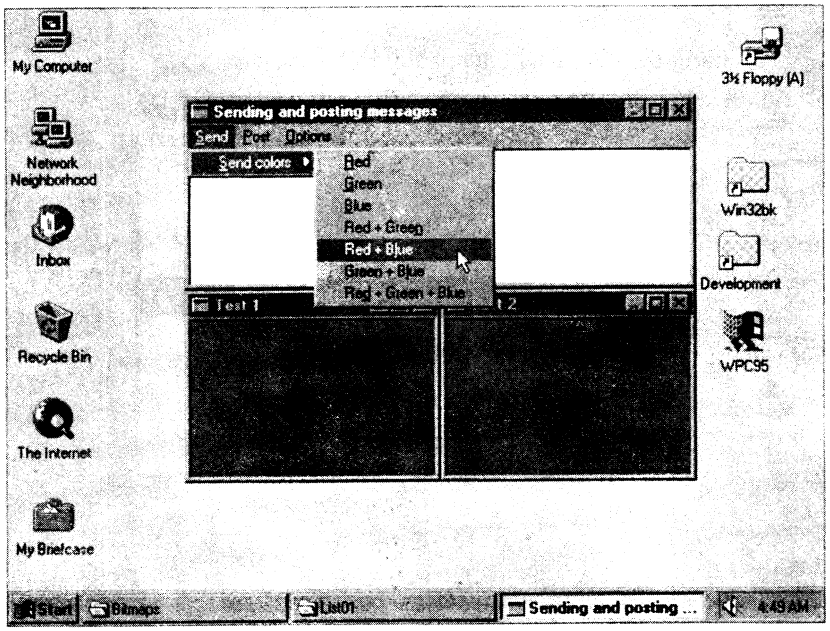


图 4-7　选择主窗口 Send 菜单内的 Change color 命令后，测试窗口的客户区现在分别变成了红色和蓝色

为了达到这个目的，我们必须对两个消息进行拦截——WM_MOVING 和 WM_WINDOWPOSCHANGED。WM_MOVING 是 Win95里才有的一条新消息。无论什么时候，只要有窗口在屏幕中移动，系统就会生成这条消息。把这条消息拦截下来以后，就可以对移动过

程中由系统在屏幕上显示出来的矩形框进行修改。在 WINMSG 里，我只是在 Y 轴上简单地扩大了矩形框，这样就增加了测试窗口的高度。

```
...
case WM_MOVING
{
    LPRECT prc = (LPRECT) lParam;

    //increase the overall rectangle
    if (!fMove)
    {
        prc -> bottom +=MAINWNDHEIGHT;
        fMove = TRUE;
    }
    //return TRUE;
}
    break;
...
```

其中，

WM_MOVING	0x0216
wParam	准备移动的窗口边线
lParam	一个 RECT 数据结构的地址，其中包含了准备重画的框架的尺寸

窗口的移动过程中，针对每一次鼠标移动，WM_MOVING 消息都会送出，但是我们只对主窗口的窗口进程里接收到的第一条感兴趣。为了抛弃后续所有的 WM_MOVING 消息，程序内声明了 fMove 这个布尔值标识符，使其在整个窗口进程里都是可见的，并且赋予其静态存储类（注意：假如必须在多条消息里使用一个标识符，为了保持它的当前值，除了把它声明为静态以外，别无他法）。

窗口的移动结束以后，应用程序就会接收到一条 WM_WINDOWPOSCHANGED 消息。这种说法并不是百分之百正确的。WM_WINDOWPOSCHANGED 会到达与上次移动毫不相干的一个窗口里。通过调用 SetWindowPos () 函数，在没有任何用户介入的前提下，窗口可以放置在不同的屏幕位置处。这条消息可以触发两个操作：把三个窗口定位于新的屏幕位置，以及把布尔值标志 fMove 重新设置成 FALSE，为下一次移动处理作好准备。

WM_WINDOWPOSCHANGED	0x0047
wParam	无
lParam	指向一个 WINDOWPOS 数据结构，该结构内包含了新的窗口位置和尺寸

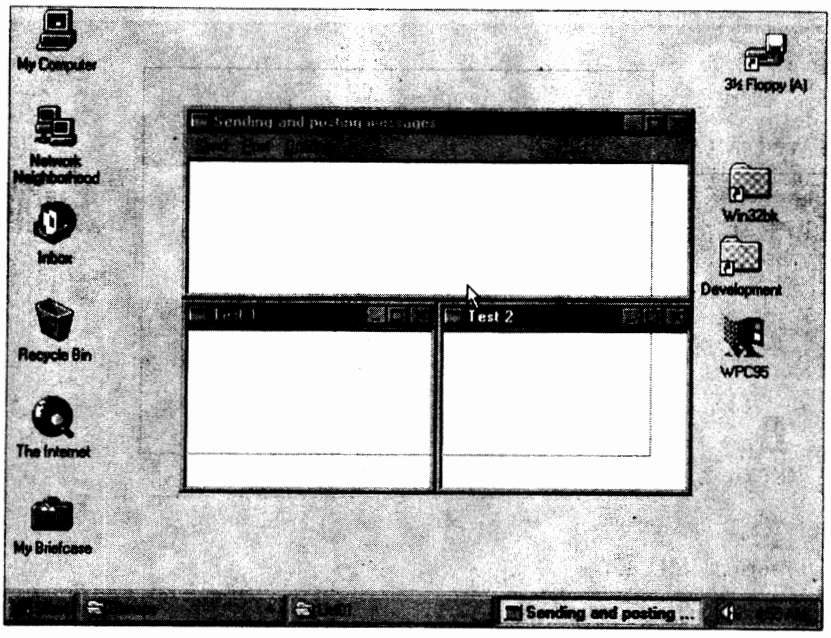


图 4-8　移动主窗口时，系统会显示出一个更大的边框，
它正好包围了全部这三个窗口

下面这个代码段是从主窗口的窗口进程——ClientWndProc（）里提取出来的。WNDHEIGHT 定义对于主窗口和两个测试窗口的高度都是对应的，而 WNDWIDTH 则代表了主窗口的宽度（次级窗口正好是主窗口的一半大小）。

```
...
case WM_WINDOWPOSCHANGED:
{
    LPWINDOWPOS pwp = (LPWINDOWPOS) lParam;

    //prevent the following code to be executed when not necessary
    if (!fMove && !wParam)
        break;

    // reposition the main window
    SetWindowPos (hwnd, HWND_TOP, pwp -> x, pwp -> y, pwp -> cx,
pwp -> cy, pwp -> flags);
    // reposition the first window
     SetWindowPos (hwndTest1, HWND_ TOP, pwp -> x, pwp -> y +
WNDHEIGHT, pwp -> cx / 2, pwp -> cy, pwp -> flags | SWP_NOACTIVATE);
```

```
            // reposition the second window
            SetWindowPos (hwndTest2, HWND_TOP, pwp -> x + WNDWIDTH / 2,
pwp -> y + WNDHEIGHT, pwp -> cx / 2, pwp -> cy, pwp -> flags | SWP_
NOACTIVATE);
            // set fMove to FALSE
                fMove = FALSE;
        }
        break;
        ...
```

窗口的移动结束以后，主窗口的重定义就要完全依赖通过 lParam 传递的信息才能实现。紧随其后，应用程序会对另外两个窗口进行修改，使它们恢复起始时的相对位置和窗口宽度。除此以外，我们还想保留主窗口的活动属性（蓝色的标题栏是标准的颜色方案），而不是让次级窗口保持活动状态。这就是我们为什么要增加 SWP_NOACTIVATE 标志的原因，因为用它可以防止次级窗口从主窗口那里夺去屏幕焦点的地位。离开这部分代码之前，我们还要把 fMove 静态标识符重新设置成 FALSE，从而为后续发生的任何窗口重定位恢复相同的应用程序逻辑。

这时我们注意到一个有趣的问题：假如用户拖动一个次级窗口，那么会得到什么样的后果呢？在这种情况下，WM_MOVING 和 WM_WINDOWPOSCHANGED 消息会进入 TestWndProc ()，而不是像以前那样进入 ClientWndProc ()。这样就使情况变得更复杂了，因为窗口进程是由这两个次级窗口同时使用的。在这个窗口进程里，我们也要声明一个 fMove 标识符，使其具有和以前一样的特性（静态）：

```
static BOOL fMove = FALSE;
```

无论选择两个次级窗口中的哪个进行移动，程序都会接收到一条 WM_MOVING 消息。这时我们需要访问由 lParam 指定的 RECT 信息，然后使顶部边框向上移动 WNDHEIGHT 个象素的距离，这样才能把主窗口也包含在矩形边框里。这是一项很容易的工作。现在，我们必须知道移动的是两个垂直边框中的哪一个。假如用户选择的是左边那个次级窗口，就必须使右边框增加 WNDWITH/2个象素的长度。相反，如果选择的是第二个次级窗口，就需要使其左边框向左延展。这儿的问题在于我们怎样才能区分出这两个不同的次级窗口。WM_MOVING 消息对于两个窗口来说显然是完全一致的，而 hwnd 参数则指定了其中一个窗口。把 hwnd 参数与次级窗口的句柄进行比较并不是一件简单的工作，因为我们并未对句柄进行处理（两个窗口都是在 ClientWndProc () 的 WM_CREATE 消息里建立的）。除此以外，没有办法把具有分级联系的一些叠置窗口链接到一起。经过这几方面的考虑以后，我决定扩展次级窗口保留的内存区域，在里为每个窗口都设置一个 ID（请参考第7章“建立窗口的艺术”，其中讲述了关于额外字节内存区更为详细的信息）。下面是这个代码部分的样子：

```
...
case WM_MOVING:
{
```

```
        LPRECT prc = (LPRECT) lParam;

        // extend the overall rectangle
        if (!fMove)
        {
            prc -> top -= WNDHEIGHT;
            //this is the first and only WM_MOVING we process
            fMove = TRUE;
            // check on which Test window we are acting
            switch (GetWindowLong (hwnd, 0))
            {
                case WN_TEST1:
                {
                    prc -> right += WNDWIDTH / 2;
                }
                    break;

                case WN_TEST2:
                {
                    prc -> left -= WNDWIDTH / 2;
                }
                    break;
            }
        }
    }
        break;
    ...
```

需要解决的第二个问题与三个窗口的实际重定位有关。窗口的移动结束以后，WM_WINDOWPOSCHANGED 消息就会进入 TestWndProc ()。由 lParam 指向的信息针对的是包含了主窗口和二个次级窗口的整个矩形。在这条消息里，我们只需重新定义主窗口的位置就可以了，甚至更简单，只需把消息传送给主窗口就可以了。无论采纳的是哪种办法，都必须以消息为中心。调用 SetWindowPos () 的时候会出现什么情况呢？这个函数会把一条 WM_WINDOWPOSCHANGED 消息发送给移动过的窗口。换句话说，假如在 TestWndProc () 的 WM_WINDOWPOSCHANGED 消息里重新定位了主窗口，那么 ClientWndProc () 里就会接收到一条 WM_WINDOWPOSCHANGED。这种情况是最适宜的，因为我们在 ClientWndProc () 里也拦截到了这条消息，这样就能对所有这三个窗口进行恰当的处理。因此，最佳的办法是把 WM_WINDOWPOSCHANGED 消息传送给主窗口，就像下面这段代码显示的那样：

```
...
case WM_WINDOWPOSCHANGED:
 {
    LPWINDOWPOS pwp = (LPWINDOWPOS) lParam;
    HWND hwndMain = (HWND) GetWindowLong (hwnd, sizeof (int));

    // skip if fMove is FALSE
    if (!fMove)
        break;

    // prevent the processing of other messages generated by the following APIs
    fMove = FALSE;

    // reposition the main window
    SendMessage (hwndMain, msg, (WPARAM) 1, lParam);
}
    break;
...
```

该程序以 WM_WINDOWPOSCHANGED 消息起始处的 fMove 为基础进行测试，从而避免了应用程序进入一种永无尽头的和危险的循环，这种死循环最终会引起整个系统的崩溃。请记住，我们现在有了两个不同的 fMove 布尔值标识符，一个在 ClientWndProc () 中使用，另一个则在 TestWndPro () 中使用。每个都可以分别对不同窗口进程里的窗口移动操作进行控制。除此以外，SetWindowPos () 还在每次调用的时候生成一条 WM_WINDOWPOSCHANGED 消息，不管它当时驻留于应用程序代码内何处。

在我们的方案里，与三个窗口重定位有关的所有逻辑操作都集中于 ClientWndProc () 里。在这个进程里，我们将三次调用 SetWindowPos ()，从而与应用程序的要求相符。代码的这一部分是由一条测试语句保护起来的，它可以检查当时是否能对三个窗口进行重定位处理。这儿用到的 if 语句是以“著名”的 fMove 和 wParam 为基础实现的。wParam？你没有说错吧？是的，的确是它。这个信息在 WM_WINDOWPOSCHANGED 消息里是没有用到的，并且它总是设置成零。单独一个 fMove 是不够用的，因为 WM_WINDOWPOSCHANGED 消息也许也是来自于 TestWndProc ()，其中是无法访问这种信息的（那儿有另外一个专用 fMove）。考虑到不能识别消息的来源，所以我决定在 wParam 里放置一些个人信息，这样就能让 ClientWndProc () 理解消息的来源。

通常，窗口在屏幕中的位置发生变化后，窗口进程里消息的接收顺序就变得相当复杂了。Spy 实用程序可以让我们监视和列出窗口在屏幕内移动后生成的所有消息：

```
WM_GETMINMAXINFO
WM_ENTERSIZEMOVE
```

WM_MOVING
WM_WINDOWPOSCHANGING
WM_WINDOWPOSCHANGED
WM_MOVE
WM_EXITSIZEMOVE

WM_MOVING 消息针对每一次微小的鼠标运动都会进行连续性的重复。假如窗口大小发生了变化，这个列表就变得更复杂了。

WM_GETMINMAXINFO
WM_ENTERSIZEMOVE
WM_SIZING
WM_CAPTURECHANGE
WM_WINDOWPOSCHANGING
WM_GETMINMAXINFO
WM_NCCALCSIZE
WM_NCPAINT
WM_GETTEXT
WM_ERASEBKGND
WM_WINDOWPOSCHANGED
WM_SIZE
WM_EXITSIZEMOVE1

在这种情况下，只要对窗口进行伸缩处理，WM_SIZING 消息就会不断重复产生。在 WINMSG 这个例子里，我们只是简单地拦截了一小部分这样的消息，另外还有一大部分没有涉及。

4.3 限制窗口的运动

在 Track 这个例子里（请参考图4-9和本书附带 CD 的 Listing 4.2），我们将对主窗口的运动进行一些限制，这是通过拦截 WM_GETMINMAXINFO 消息来实现的。这些消息是在窗口建立好以后和返回它的句柄之前由 CreateWindowEx () 产生的，或者是每次重新定义窗口尺寸或者像以前讲述的那样在屏幕中移动时由 Windows 本身产生的。

WM_GETMINMAXINFO	0x0024
wParam	未使用
lParam	一个 MINMAXINFO 数据结构的地址

lParam 指向的内存区域内包含了一个 MINMAXINFO 结构里存储的信息。四个可访问的项允许我们定义窗口在最大化显示后的位置（ptMaxPosition）、最大的尺寸（ptMaxSize）以

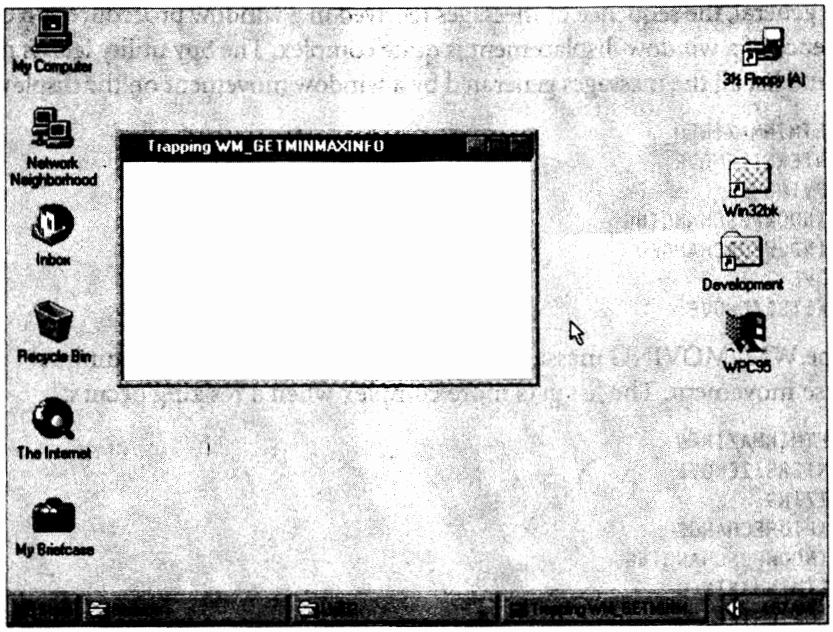


图 4-9　Track 提供了一个可以重新定义尺寸的边框，尽管拦截了 WM_GETMINMAXINFO 消息后，它最大不能超过规定的尺寸

及水平或者垂直伸缩窗口时最小和和最大尺寸（ptMaxTrackSize 和 ptMinTrackSize）。

```
typedef struct tagMINMAXINFO
{     // mmi
    POINT ptReserved;
    POINT ptMaxSize;
    POINT ptMaxPosition;
    POINT ptMinTrackSize;
    POINT ptMaxTrackSize;
} MINMAXINFO;
```

现在让我们来观察一下摘抄自 Track 例子的这个代码段：

```
...
case WM_GETMINMAXINFO:
{
    LPMINMAXINFO lpmmi;

    lpmmi = (LPMINMAXINFO) lParam;
```

```
        // maximizing dimensions
        lpmmi -> ptMaxSize.x = 330;
        lpmmi -> ptMaxSize.y = 200;

        // maximizing position
        lpmmi -> ptMaxPosition.x = 0;
        lpmmi -> ptMaxPosition.y = 0;

        // minimal size
        lpmmi -> ptMinTrackSize.x = 300;
        lpmmi -> ptMinTrackSize.y = 100;

        // maximum extensibility
        lpmmi -> ptMaxTrackSize.x = 330;
        lpmmi -> ptMaxTrackSize.y = 200;
    }
        break;
...
```

最大的窗口位置是显示器的左上角，尽管窗口的宽度和高度分别被限制在330和200个象素点以内。通过对垂直的窗口边框进行操纵，最小可以收缩成300个象素（lpmmi -> ptMinTrackSize.x = 300），最大则可扩展至330个象素（lpmmi -> ptMaxTrackSize.x = 330）。窗口的高度在100和200之间变化（lpmmi -> ptMinTrackSize.y = 100和 lpmmi -> ptMaxTrackSize.y = 200）。

限制一个 Win32进程的运行拷贝

Justone 例子向大家阐明了怎样限制 Win32进程运行拷贝的数目。这个目标可以通过本小节讲述的几种技术来实现。Justone 的运行要依赖消息在不同拷贝间的通信。注册好类以及建立好主窗口以后，Justone 就会散播在应用程序源代码内定义的一条消息：

```
...
hwnd = CreateWindowEx (WS_EX_CLIENTEDGE | WS_EX_WINDOWEDGE,
                       szClassName,
                       szWindowTitle,
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT, 0,
                       CW_USEDEFAULT, 0,
                       NULL,
                       NULL,
                       hInstance,
```

```
NULL);

    // broadcasting
    //SendMessage ( HWND_ BROADCAST, WM_ SECONDINSTANCE,
(WPARAM) hwnd, 0);
    ...
```

传递了最新建立窗口的句柄以后，我们再按照下面这种形式定义 WM_SECONDINSTANCE：

```
#define WM_SECONDINSTANCE WM_USER + 1000
```

在应用程序唯一的一个窗口进程里，我们把 WM_SECONDINSTANCE 消息块放了进去：

```
...
case WM_SECONDINSTANCE:
{
    //skip if the sending window is the recipent as well
    if (hwnd == (HWND) wParam)
        break;

    // answer back
    SendMessage ( (HWND) wParam, WM_PLEASEDIE, 0, 0);
}
    break;
...
```

第一项处理就是检查发送和接收窗口区域是否含有相同的对象。如果得到了肯定的答案，我们就可以跳过消息，继续标准的执行。当 WM_SECONDINSTANCE 抵达了 Justone 运行拷贝的窗口进程以后，if 语句就会明显地返回一个 FALSE 值，这就表明测试没有成功。在这个时候，第一种情况会发送 WM_PLEASEDIE 消息——在应用程序里定义的另外一种消息，从而把回答反馈回来。

```
#define WM_PLEASEDIEWM_USER + 1001
```

紧跟在进程中断的后面，这条消息将生成一个消息框。

```
...
case WM_PLEASEDIE:
{
    // inform the user that this second copy is terminating
      MessageBoxEx(NULL, "Found a previous running copy", "Bye - bye",
```

```
        MB_OK,MAKELANGID(LANG_ENGLISH,SUBLANG_ENGLISH_US));
        // terminate process
        ExitProcess(1);
    }
        break;
...
```

图4-10为大家展示了 Justone 执行过程中产生的消息流程。假如执行了第二个拷贝，它就会自动中止，同时显示出一个对话框。

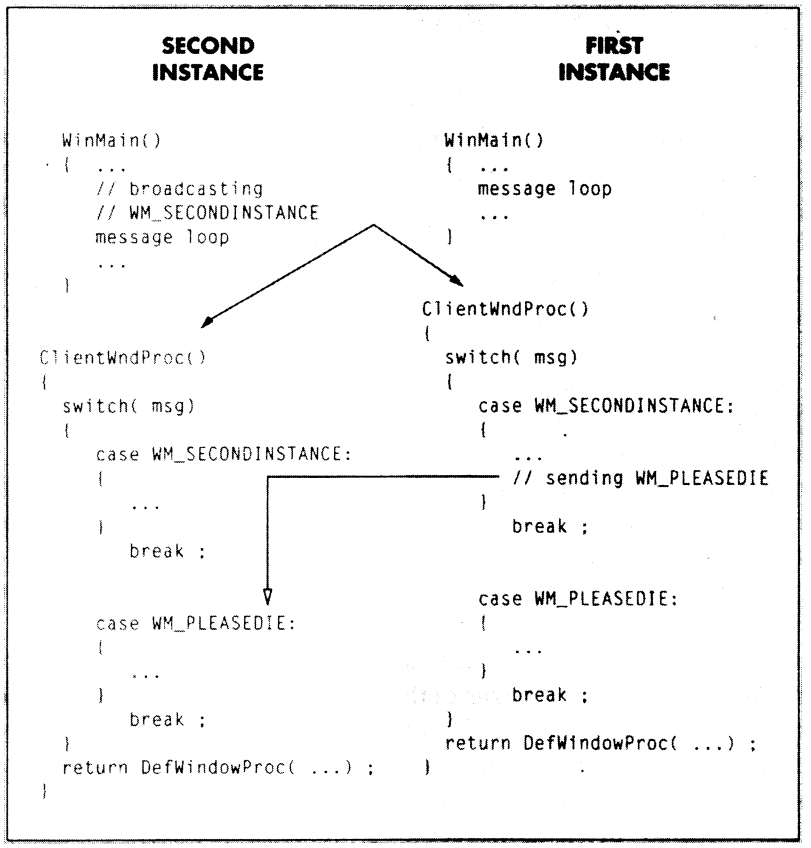


图 4-10　Justone 在执行后会立即传播一条消息到任何一个顶级窗口内。这条消息只有相同程序的另一个运行拷贝才能识别

通过线程发送消息在 Win32里并不是一种最好的办法，尽管那样也行得通。读者能猜出另外一种可选的方案吗?好了，为了对一个 Win32执行程序的拷贝进行跟踪，我至少能想出半打可选的方案。Mutex（相互排斥机制）、信号机、注册表数据库内的一个键、命名管道、共享内存块以及 API 函数 FindWindow()，这些工具都能成功地用于避免第二个实例的运行。其中一些技术会在本书后面的章节里进行介绍。

4.4 消息和抢先式多任务

抢先式多任务对消息的传送和处理会产生什么样的影响呢?和传统的协作式多任务应用程序比较起来，抢先式多任务毫无疑问代表了一种真正的进步，所以提出这个问题是非常正常的。就抢先式多任务的系统本质来说，它不会对传统的消息产生任何形式的影响。新环境和Win16之间的唯一区别关系到 CPU 内线程的永恒性。在 Win16里，只有在执行 GetMessage () 的时候，任务才会释放处理器。无论处理一条消息要花多长的时间，CPU 在这段时间内都要为此而服务。这种情况到了 Win32已经得到了改变。返回 GetMessage () 之前，一个线程会得到多次优先权，特别是在涉及到一个耗时很多的操作时。CPU 潜在的进出并不会影响到最终的结果。图4-11向大家说明了这种情况。

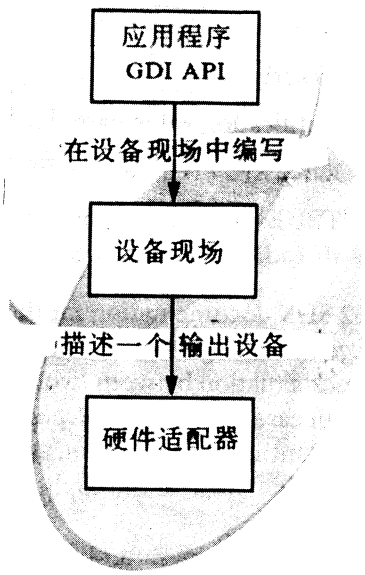


图 4-11　一个 Win32操作正在处理一条消息以及它与系统调度程序和多任务管理模块的关系

在另外一方面，假如从一个线程向另一线程发出消息，那么多线程和多任务确实也会对线程的管理发生影响。这样就使情况变得更加复杂了。下面这个例子会帮助我们理解执行一个表面似乎无害的操作时，为什么实际上却面临着潜在的危险。

假如进程 A 是一个单线程应用程序（线程 A)，它在自己的队列里有一条未决的消息。该程序会取回这条消息，然后把它派遣到合适的窗口进程内。消息处理的时间相当长，在这个操作完成之前，分配给线程的时间片就已用完了。此时，系统调度程序会选择运行于更高优先级的另外一个线程：进程 B 的线程 B。线程 B 也要忙于进行一个颇为耗时的操作（处理消息 Z)，它的时间片用完的时候，仍然在进行同一条消息的处理。现在是线程 A 恢复执行的时候了。它

仍然要处理最初的那条消息（消息 X），然后把另一条消息（消息 Y）发送给线程 B。SendMessage（）函数的调用通常都涉及到一个线程开关，然而在当前情况下这种操作是不可能进行的。事实上，当线程 B 恢复的时候，它仍然忙于处理消息 Z，而消息 Y 不能传递给目标窗口进程。这样产生的结果就是，线程 A 将处于挂起状态，因为除非消息 Z 和消息 Y 的处理都结束了，否则执行就不能恢复。这要花多长的时间呢？我们不知道。我们知道的唯一事实就是现在进入了一种死锁状态。

Win32 API 针对这个问题提供了一个局部的解决方案。InSendMessage（）这个 API 函数可以成功地用于测试是否到达了一条进入消息，这条消息的来源既可以是当前进程，也可以是一个不同的进程。

```
#include〈winuser.h〉
BOOL WINAPI InSendMessage（VOID）;
```

显然，InSendMessage（）不会极大地减轻上面描述那种情况带来的不良影响。除非执行流程再一次进入了一个窗口进程，否则该函数是无法执行的（大家还能回忆得起在前面那种情况里，当线程 A 向线程 B 发送消息 Y 的时候，线程 B 正忙于处理消息 Z）。假如 InSendMessage（）侦测到当前消息是从一个不同线程里发出来的，它就会返回一个 TRUE 值。假如碰到了这种情况，我们就可以再用 ReplyMessage（）对呼叫线程提供一次立即的答复，这样就避免它在消息处理完成之前一直处于闲等状态。

```
#include〈winuser.h〉
BOOL WINAPI ReplyMessage（LRESULT lResult）
```

参数	说明
LRESULT lResult	传递给发送线程的通知值
返回值	在正文里讨论
BOOL	假如正在处理的是其他线程发出的消息，就返回一个 TRUE 值；否则返回 FALSE

第一次阅读 Win32 SDK 的文档资料时，我就感觉到 ReplyMessage（）是程序开发人员的救星，因为它显然可以避免两个独立进程间发生任何形式的死锁现象。但是，使用这个函数的时候也要小心谨慎。通过调用 ReplyMessage（），发送方的那个线程就有机会继续它的执行，从而避免了可能发生的死锁现象。不管在哪种情况下，应用程序的整体逻辑都会进行变换，我们下面会对此进行具体的解释。

具备16位 Windows 编程经验的任何人都知道 SendMessage（）把当前的操作挂起以后，只有从目标窗口进程里返回以后才会恢复执行。ReplyMessage（）改变了这种非常死板的方式，它会向发出 SendMessage（）的线程立即返回一个值。这样一来，开发者就必须针对 ReplyMessage（）设计一种新的消息响应控制逻辑。ReplyMessage（）返回它的值时，消息处理还没有启动，它的返回值只不过是一种通知信息罢了。我们需要解决的问题是保证两个线程的同步——这个要求可以通过其他的 IPC（进程间通信）机制来实现，比如 ReplyMessage（）等，从而返回真实的消息处理结果。

为了通知发送进程消息已被处理，最简单的办法就是向起源窗口张贴一个不同的地址消

息。当张贴消息只是简单地作为一条通知使用，而不是用于传递某些数据的时候，这种方案是特别有效的。在更为复杂的情况下，发送线程内可能会再建立一个线程；另外，为了使同一应用程序的两部分同步，需要设置一个事件的时候（这是一种 IPC 机制，关于这方面的详情，我们会在第14章“多任务、IPC 和 I/O”里叙述），也有可能会建立另一个线程。

需要注意的是，尽管一条线程间消息可以在它抵达目标窗口进程后立即得到处理，我们仍然不能保证接收方线程会不会在处理过程中抢先。它返回 CPU 既有可能在瞬间发生，也有可能要等一段时间后再发生，这样就阻碍了最初发出消息的那个线程的执行。

Win32 API 提供了一些有趣的选择方案，它们可以对线程间消息处理时可能发生的死锁现象进行控制。举个例子来说，针对这一目的，SendNotifyMessage（）这个 API 就提供了一种非常有趣的特性。假如目标窗口从属于和发送方相同的线程，这个函数的表现就和一个标准的 SendMessage（）无异。所以，当前的消息执行就会暂时挂起，直到 SendMessage（）返回为止。在另外一方面，假如目标窗口是在不同线程内建立起来的，SendNotifyMessage（）就会立即返回一个布尔（Boolean）值，以此证明消息发送的结果。因此，这种操作看起来就不像传统的消息发送。以传统的眼光来看，除非接收方已经处理完毕接收到的消息，否则调用函数就会一直处于挂起状态。

```
#include <winuser.h>
BOOL WINAPI SendNotifyMessage (HWND hwnd,
                               UINT msg,
                               WPARAM wParam,
                               LPARAM lParam);
```

参数	说明
HWND hwnd	目标窗口的句柄
UINT msg	准备发送的消息
WPARAM wParam	关于消息的附加信息
LPARAM lParam	关于消息的附加信息
返回值	在正文里讨论
BOOL	假如消息发送成功，就返回一个 TRUE 值；如果失败，则返回 FALSE

SendNotifyMessage（）是一个非常有用的函数，因为即使接收方线程当时正忙于处理另一条消息，它也肯定能把消息发送出去。它的表现与我们以前讨论过的 SendMessage（）存有着很大的区别。

SendMessageTimeout（）可以说是 SendNotifyMessage（）与 SendMessage（）这两者之间的一种妥协。和往常一样，我们仍然需要根据目标窗口的本质进行分别对待。消息抵达同一线程内的某窗口时，SendMessageTimeout（）就扮演了一个普通的 SendMessage（）的角色，其中的“超时”（Timeout）设置将被忽略。否则，SendMessageTimeout（）就只有在目标窗口进程处理完了消息，或者“超时”设置到期后才会返回。即使在那个时候，我们都无法百分之百地保证接收方窗口正在与起源窗口共享同一消息队列。

SendMessageTimeout（）不仅是避免发生不愉快死锁现象的一种很好的工具，它还提供了一个附加的参数，利用它可以设置调用线程的行为。具体的语法结构如下所示：

```
#include〈winuser.h〉
LRESULT WINAPI SendMessageTimeout（HWND hwnd，
                                    UINT msg，
                                    WPARAM wParam，
                                    LPARAM lParam，
                                    UINT fuFlags，
                                    UINT uTimeout，
                                    LPDWORD lpdwResult）；
```

参数	说明
HWND hwnd	目标窗口的句柄
UINT msg	准备发送的消息
WPARAM wParam	关于消息的附加信息
LPARAM lParam	关于消息的附加信息
UINT fuFlags	定义消息发送的方式
UINT uTimeout	用时间片指定调用应该等待的时间
LPDWORD lpdwResult	消息处理的结果
返回值	在正文里讨论
LRESULT	假如消息发送成功，就返回一个 TRUE 值；如果失败，则返回 FALSE

在需要对线程间消息进行处理的时候，SendMessageCallback（）也许是我们能够利用的最佳工具了。这个函数会立即返回，同时传递回调函数的地址。以后当消息处理中止的时候，就可以执行那个回调函数。

```
#include〈winuser.h〉
LRESULT WINAPI SendMessageCallback（HWND hwnd，
                                    UINT msg，
                                    WPARAM wParam，
                                    LPARAM lParam，
                                                SENDASYNCPROC
lpResultCallBack，
                                    DWORD dwData）；
```

参数	说明
HWND hwnd	目标窗口的句柄

UINT msg	准备发送的消息
WPARAM wParam	关于消息的附加信息
LPARAM lParam	关于消息的附加信息
SENDASYNCPROC lpResultCallBack	消息处理结束时调用的应用程序回调函数
DWORD dwData	消息处理完成后传回的应用程序数据
返回值	在正文里讨论
LRESULT	假如消息发送成功，就返回一个 TRUE 值；如果失败，则返回 FALSE

这个函数是 Win32编程的一个要点，关于它的详情，我们会在第13章“内存管理和DLL”里进行介绍。

至此为止，我们已学习了进程间消息发送的许多知识。现在可以考虑这样一个问题：怎样才能避免这种情况的发生呢?非常容易。只需简单地防止出现进程间消息，或者对两个进程进行有效的编码，这样就能限制异常事件发生的机率。

4.5 API 和消息

现在，我们已经知道 WM_消息既可以发送给一个窗口，也可以张贴到一个应用程序队列里。经常碰到的一种情况是：WM_消息到达一个窗口进程的时候，它既没有调用 PostMessage（），也没有调用 SendMessage（）。这是怎么回事呢?这就是我们通常称谓的“API 的副作用”。只需简单地调用一个 API，窗口进程就会接收到一系列由 API 本身生成的消息，这种情况并不少见。从技术的角度来看，这是很容易便可实现的。为了实现它，微软公司的工程师们在一个函数的代码结尾处设置了对 SendMessage（）的一些调用。从我们的眼光来看，这些消息简直是无处不在。CreateWindowEx（）也许是最能说明问题的一个例子。该函数的目标是建立一个新窗口。一旦包含了新窗口信息的内存块存储到了 USER.EXE 里，并且剩下的其他预备任务均已完成，CreateWindowEx（）就会向新建窗口的窗口进程发送一系列固定的消息，从而中断自己的执行。发送出去的这些消息有两个目的：完成建立进程，并且通知适当的窗口进程现在已存在一个新窗口。

通过阅读联机帮助文档，我们可以知道 WM_CREATE，WM_GETMINMAXINFO 和 WM_ NCCREATE 这一系列消息都是由 CreateWindowEx（）生成的。但是观察由 CreateWindowEx（）实际生成的消息流，我们会发现顺序和上面相比稍微有些不同，消息的数量也不止三条。在第15章“Windows 高级技术”里，我们提供了一个名为 MSGFLOW 的例子，利用这个工具便可对我们上面讲述到的现象进行完整的观察。

不管 CreateWindowEx（）首先发送出来的是哪条消息，这种消息的产生都引发一个足以引起我们兴趣的问题。大家知道，CreateWindowEx（）要么返回新建窗口的句柄，要么返回一个 NULL 值（条件是函数的调用由于某种原因而失败）。然而，在这同时，只有当有一条消息需要发送的时候，执行流程才会转向一个窗口进程。这一行动首先要求目标窗口的句柄是可以使用的。不管是 CreateWindowEx（），还是其他某种 API，只要它生成了消息，就简单地意味着新建窗口的句柄会通知给 WinMain（）内的前一类窗口进程。

由 API 带来的这种“副作用”是我们在 Windows 程序开发过程中需要引起足够重视的。

对于一名出色的 Windows 程序员来说，他设计程序代码的时候应该不会花很多心思，因为这些代码能够利用窗口进程里的各种自然信息流，而且程序指令也能与特定的窗口消息对应。

不幸的是，由某些 API 类引入的消息流并未由微软公司进行具体的文档化。该公司声称在后续的操作系统版本里也许会对这一问题进行改正。不管微软口头宣称的是什么，对于我们来说，现在了解和利用这些现成的消息是非常重要的。我的经验证明，微软在最近的这八年中只是在最初的序列基础上增加了一些消息，很容易就能回想起他们上次作出改进是在什么时候。

4.6 Spy 和消息

Windows SDK 附带提供了一个名为 SPY.EXE 的工具软件，它的用途是对其他窗口进行侦测。新版本的 Spy 与最初的16位版本已经大相径庭了，这也反映出由于 Win32引入多重输入队列而带来的某些限制。在过去，Spy 可以让用户人工选择一个窗口，方法是把鼠标光标拖动到该窗口的上方，然后按下鼠标左键。这种操作在 Win32里已经不灵了。如图4-12所示，Spy 主窗口在左侧提供了一个列表框，其中列出了系统内现成的所有窗口。事实上，这份列表只限于列出所有叠置式和弹出式窗口(即具有 WS_OVERLAPPED 和 WS_POPUP 风格的窗口)，对于子窗口就只有爱莫能助了。

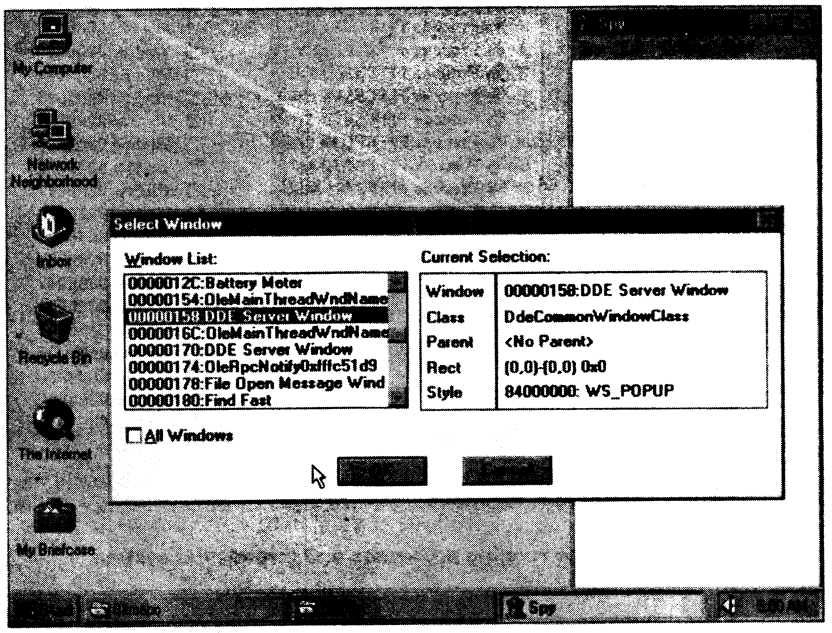


图 4-12 SPY.EXE 程序在“窗口”列表框内列出了所有顶级窗口

一旦选中了某个窗口，与之相关的消息就会在 Spy 的客户区内显示出来，显示的顺序则完全依照它们在目标窗口里接收到的顺序。为了实现这种功能，Spy 程序还安装了一个“系统挂接”(system hook) 程序，它能捕获系统队列里存在的所有事件。随后，应用程序只选择那些定址于侦测窗口的事件，然后在自己的客户区内把它们显示出来。尽管使用系统挂接看起来

似乎是解决许多开发问题的正确方案（我猜测许多读者对这种技术都还不是十分了解），但我还是奉劝大家尽量不要使用这种技术，因为它会降低系统的整体性能。

除此以外，Spy 不能拦截由 CreateWindowEx（）生成的消息，因为这些消息是在窗口的建立过程中产生的；而 Spy 不能对这种窗口建立事件进行跟踪。Visual C++编译器提供了 Spy 的一个扩展版本，名为 Spy++。这个增强程序能提供基本相同的功能，而且并不限于只能处理系统内当前存在的窗口。利用 Spy++（如图），我们能获得关于系统内正在运行的所有进程和线程的信息。

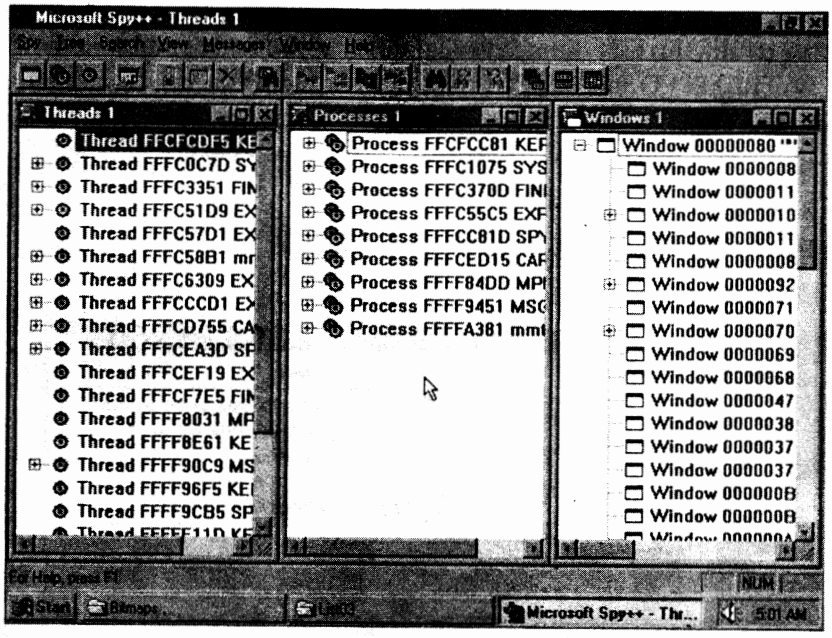


图 4-13　Spy++列出了系统内正在运行的所有进程和线程

无论 Spy 还是 Spy++，它们返回的关于进程或者窗口的信息都是有限和不完整的。如果把这些信息与 PVIEW95（参考第3章“Win32应用程序的开发”）提供的信息合并到一起，就能对系统内正在发生的事件有一个更好的把握。我鼓励大家快速跳转至第16章“Win95外壳开发”，看一看 WINSPY 程序——我自己开发的一个 Spy 版本。

4.7　重画技术

现在让我们来探讨重画技术——对应用程序在客户区内的输出进行约束的一系列规则。自从 Windows 的2.x 版本引入后，窗口就可以一个接一个相互叠置起来，从而共享一个通用的屏幕设备。通常，最后一个建立的窗口或者利用鼠标选择的一个现成窗口会在屏幕前台显示出来。这种行为是与一种名为“Z 序列”（z-order）的排序机制对应的，因为它能模拟窗口在 Z 轴上的性质。假如把两个或者多个窗口重叠到一起，客户区的部分区域就会暂时隐藏不见，因为它们已被其他窗口覆盖了。这是一种非常普遍的现象。假如当前位于后台的一个窗口进入了前台，一个明显的需要就是重画前面被隐藏起来的客户区矩形，使其清楚地显现出来。Windows 不能帮助我们重画那些部分，因为客户区是应用程序用于显示输出结果的的窗口部

分。所以，唯一能够恢复输出显示的就只有应用程序内部的代码。深入探索这方面的问题之前，让我们首先观察一下操作系统和下层硬件的交互作用。

4.7.1 硬件处理

最初，Windows 环境可以用三种基本的组件来标识：内核、GDI 和用户 DLL。这些核心组件可以通过系统控制的特定设备驱动程序与硬件设备进行交互作用。通过安装恰当的虚拟设备驱动程序（带有.VXD 扩展名的模块），Windows 本身可以实现对键盘、鼠标以及显示器的管理。

驱动程序的任务是挖掘出一个物理设备（比如显示卡）的所有潜力，以及把取回的信息传递给操作环境，如图4-14所示。

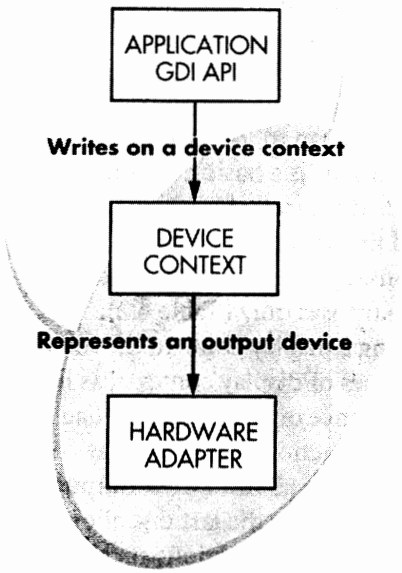


图 4-14　Windows 95的输出模式把应用程序与基层的硬件组件分隔开了

Windows 应用程序不会和任何硬件设备直接打交道。相反，程序和硬件是各自独立的，中间通过特定的驱动程序来沟通信息。正是由于这种独特的整体结构，Windows 通常被定义成一种“与设备无关”的操作环境。

举个例子来说，显示卡驱动程序会在 Windows 的内部建立一块内存区域。我们把这种区域称为“设备现场”(device context)，里存储了与特定硬件（这儿是显示卡）有关的所有信息。对于显示卡这种最常用的输出设备来说，存储的信息涉及到屏幕分辨率、每个象素的颜色数以及当前选定的刷子句柄等等。表4-1列出了一个视频设备现场（亦可称为“显示现场”）的所有属性。

屏幕是最常用到的一种输出设备现场。许多以窗口形式出现的应用程序同时驻留于窄小的屏幕内。对于一个单独的应用程序来说，它必须把设备现场信息限制成只与自己的窗口有

关，而不是与整体屏幕表面都有关系。第二个严格的限制是输出部分必须进入单独的客户区。因此，为了生成一些输出信息，必须先获得与特定客户区有关的一个设备现场。然后，我们才能获得一些信息、正文和图形等等。无论 Windows 的哪个 API，只要它能产生某种形式的输出，就属于 GDI（图形设备接口）组的一个成员。除此以外，GDI 组内的任何 API 通常都需要用到指向某设备现场的一个句柄(HDC)，把它当作自己的第一个参数来使用。没有 HDC 的帮助，Windows 应用程序就不可能产生任何形式的输出。

4.7.2 设备现场

设备现场是一个通用的术语，它用于标识由一种特定驱动程序提供的任何一种硬件设备句柄。这是一种基本的定义，因为在现实世界里，对一种设备现场进行编程典型地需要涉及到两种物理设备：显示器和打印机。除此以外，可能需要用一定的内存拷贝来自某个设备现场的信息，或者用于纯粹出于信息目的去查询一个设备现场，如表4-2所示。

让我们先来看看显示现场的情况，这是一种最常见的形式。Windows 95提供了四种类型的显示现场，如表4-3所示。其中一些要依赖类注册中设置的 CS_标志。

显示现场之间具有明显区别的一种元素就是输出区域。对于表4-3内的前三种显示现场来说，它们的输出只限制在窗口客户区域，这是最标准的一种情况。最后一种则允许开发者在一个窗口的任何地方进行输出，其中包括非客户区域，比如标题栏、可重新定义窗口尺寸的边框以及由窗口风格标志引入的其他窗口组件。

表 4-1 显示现场的标准组件

属 性	值
背景色	White（白色）
背景类型	OPAQUE（不透明）
位图	缺省状态下为 None（无）
刷子	WHITE_BRUSH
刷子起点	(0，0)
剪裁区域	整个客户区
调色板	DEFAULT_PALETTE
笔的当前位置	(0，0)
设备起点	客户区的左上角
绘图模式	R2_COPYPEN
字体	SYSTEM_FONT（对于那些为 Windows 3.0以前的版本编写的应用程序，则使用 SYSTEM_FIXED_FONT）
字符间距	0
映射模式	MM_TEXT
笔	BLACK_PEN
多边形填充模式	ALTERNATE（备选）
相对—绝对标志	
拉伸模式	BLACKONWHITE
正文颜色	Black（黑色）
视口范围	(1，1)
视口起点	(0，0)
窗口范围	(1，1)
窗口起点	(0，0)

表 4-2 Windows 95提供的四种类型设备现场

设备现场类型	说　明
显示	支持视频显示器上的绘图操作
打印机	支持打印机或者绘图仪上的绘图操作
内存	支持对位图的绘图操作
信息	用于获取关于某种设备现场的信息

表 4-3 Windows 95实现的四种类型显示现场

显示现场	类标志（CS_）	显示现场	类标志（CS_）
Common（普通）	None（无）	Private（专用）	CS_OWNDC
Class（类）	CS_CLASSDC	Window（窗口）	None（无）

1. 普通显示现场

迄今为止建立的所有例子都提供了一个普通显示现场，然而我们并没有明显地利用过它们。如表4-3所示，WNDCLASS 结构里的风格标志没有一个特定的 CS_标志，这就意味着属于那一类的所有窗口都采用了普通显示现场。执行有限输出操作的 Win32进程就是属于这一类的。

2. 类显示现场

如果想为一个窗口分配一个类显示现场，必须在注册类的时候设置 CS_CLASSDC 标志。之所以在 Win32里实现这种形式的显示现场（Display Context，DC），是出于对向后兼容问题的考虑。我们一般应尽量采用专用显示现场。

3. 专用显示现场

如果想生成专用显示现场，必须用 CS_OWNDC 标志注册一个类。和类显示现场相比，最显著的区别在于：这种形式的 DC 即使在释放了一个设备现场句柄以后，它也会保留自己的缺省信息。如果程序生成的显示输出信息很多，那么最好还是使用这种形式的 DC。

4. 窗口显示现场

假如应用程序希望在它的客户区以外进行描绘，就需要用到窗口显示现场，这是一种特殊形式的 DC。输出内容是由客户区和非客户区信息组成的。这种类型的设备现场很少有机会用到。

4.7.3 访问显示现场

设备和显示现场现在都已有了一个常规的定义。接下来，我们将深入考察如何通过一些 API 调用来访问这样的一个对象。有许多种工具都可以胜任这一任务，其中有些工具是专门设计用于完成某些特定操作的。表4-4为大家总结了与设备现场有关的 API 函数。

表 4-4 DC 管理中涉及到的 Win32函数

函数	说明
CreateCompatibleDC（）	以信息和另一 DC 的结构为基础建立一个 DC
CreateDC（）	建立一个崭新的 DC
CreateIC（）	建立一个信息现场

（续）

函　数	说　明
DeleteDC（）	删除 DC
GetDC（）	返回 DC 的句柄
GetDCEx（）	返回 DC 的句柄
GetDCOrgEx（）	获得针对特定 DC 的最后一个转换起点，亦即获得 DC 起点到屏幕起点的距离
ResetDC（）	重置一个打印机或者绘图仪 DC
RestoreDC（）	把 DC 恢复成它起先的状态
ReleaseDC（）	释放一个普通或者窗口 DC
SaveDC（）	存储 DC 当前的状态

在表4-4列出的所有函数中，最常用的还是 GetDC（）。这个函数只需要一个参数，那就是某个窗口的句柄，它能返回用于适当显示现场的句柄。

```
#include〈winuser.h〉
HDC WINAPI GetDC（HWND hwnd）;
```

参数	说明
HWND hwnd	窗口句柄
返回值	在正文里讨论
HDC	显示现场的句柄，假如函数调用失败，则返回一个 NULL 值

GetDC（）返回的显示现场句柄是与某个特定的窗口有联系的，这个句柄随后可用于访问那个特定的 DC。对于开发者来说，最好的做法是在所有相关操作都结束以后，再把这个句柄返回给系统。GetDC（）通常都是和 ReleaseDC（）成对使用的，后者的作用是中断应用程序和输出表层之间的链接。

```
#include〈winuser.h〉
int WINAPI ReleaseDC（HWND hwnd，HDC hdc）;
```

参数	说明
HWND hwnd	窗口句柄
HDC hdc	用于窗口显示现场的句柄
返回值	在正文里讨论
int	假如显示现场已经释放了，返回一个 TRUE 值；否则返回 FALSE 值

GetDC（）和 ReleaseDC（）这个函数对必须排列于同一个消息块内使用，就像下面这段代码那样：

```
...
case WM_xxx
{
```

```
        HDC hdc;

        hdc = GetDc (hwnd);
        ...
        // operation that involve hdc
        ...
        ReleaseDC (hwnd, hdc);
    }
        break;
...
```

对于要求使用某个显示现场句柄的任何一个 API 调用来说，它们都应该封装到 GetDC () —ReleaseDC () 函数块内。由 GetDC () 返回的 HDC（用于显示现场的句柄）很少用于执行实际的输出操作。更准确地说，它通常在除了 WM_PAINT 以外的其他 WM_消息里使用，这些消息里驻留了应用程序的所有输出逻辑。这里需要注意一条简单的规则，那就是要避免它对应用程序的执行带来不愿看到的副作用。除此以外，对于要求避免执行错误的代码块来说，我建议大家把 HDC 标识符声明为只能局部用于这个块。另外，开发者也应记住调用 GetDC () 从 Windows 获取一个真实的值。

Win32 API 提供了另外一种方法来访问客户区显示现场，那就是调用 GetDCEx () 函数：

```
#include <winuser.h>
HDC WINAPI GetDCEx (HWND hwnd,
                    HRGN hrgnClip,
                    DWORD fdwOptions);
```

参数	说明
HWND hwnd	窗口句柄
HRGN hrgnClip	标识窗口的剪切区域
DWORD fdwOptions	表4-5里列出的一个或者多个 DCX_标志
返回值	在正文里讨论
HDC	显示现场的句柄，假如函数调用失败，则返回一个 NULL 值

GetDCEx () 允许我们获得一个 HDC，用它来指定一个剪切区域和其他高级特性。

表 4-5 GetDCEx () 函数使用的 DCX_标志

定 义	值	说 明
DCX_WINDOW	0x00000001L	返回与整个窗口对应的一个 DC，不包括客户区
DCX_CACHE	0x00000002L	如果使用了这个标志，它就会优先于 CS_OWNDC 和 CS_CLASSDC 类风格从缓冲区里返回 HDC
DCX_NORESETATTRS	0x00000004L	假如释放了设备现场，就不把这个设备现场的属性重新设置成缺省属性

（续）

定 义	值	说 明
DCX_CLIPCHILDREN	0x00000008L	排除由 hwnd 标识的窗口下的所有子窗口的可见区域
DCX_CLIPSIBLINGS	0x00000010L	排除由 hwnd 标识的窗口上的所有父窗口的可见区
DCX_PARENTCLIP	0x00000020L	使用父窗口的可见区域。父窗口的 WS_CLIPCHILDREN 和 CS_PARENTDC 风格位将被忽略。设备现场的起点被设置成由 hwnd 标识的窗口的左上角
DCX_EXCLUDERGN	0x00000040L	由 hrgnClip 标识的剪切区域将从返回设备现场的可见区域内排除。
DCX_INTERSECTRGN	0x00000080L	由 hrgnClip 标识的剪切区域将由返回设备现成的可见区域进行分割
DCX_LOCKWINDOWUPDATE	0x00000400L	尽管存在一个有效的 LockWindowUpdate 调用，仍然允许对窗口进行绘图——否则就要把这个窗口排除掉；用于在跟踪过程中进行窗口绘图。
DCX_VALIDATE	0x00200000L	如果是附带指定了 DCX_INTERSECTUPDATE，设备现场就会完全有效；如果把它与 DCX_INTERSECTUPDATE 和 DCX_VALIDATE 联合使用，就完全等效于使用 BeginPaint 函数

4.8 什么时候用 GetDC ()

无论 GetDC () 的返回值是什么，在应用程序里产生某些输出是我们的一项关键任务。然而，GetDC() 很少针对这一用途提供服务，唯一的例外是某个窗口属于带有 CS_PRIVATEDC 标志的一个类的时候。返回的 HDC 句柄对于查询显示现场或者获取某些信息是很有用的，但它对真正产生某些输出却是没有什么帮助的。

假如我们现在想测量一个正文块内连续两行间的距离。换句话说，我们对系统字体的高度发生了兴趣，这样才能保证一个窗口在 Y 轴上的长度是这种距离的整数倍。GetDC () 正是满足这种需求的绝佳工具。下面这个代码段向大家阐示了 GetDC() 在 WM_CREATE 消息里的用法。在代码中，后来对 GetTextMetrics () 的一次调用返回了与字体有关的所有信息：

```
...
case WM_CREATE
{
    TEXTMETRICS tm;
    HDC hdc;

    hdc = GetDC ();
    GetTextMetrics (hdc, &tm);
    ...
    ReleaseDC (hwnd, hdc);
}
    break;
...
```

现在，我们很容易就可以保证一个客户区的高度正好是系统字体高度的整数倍。显然，这种信息只有在执行过程中才能获得，这是由于不同 PC 选择的显示分辨率是不尽相同的。在这

种特定的情况下，我们可以建立一个没有尺寸的窗口（cx 和 cy 都设置成0），然后根据以前由 GetTextMetrics () 获得的信息，调用 WM_SIZE 消息来强制这个窗口使用新设定的宽度和高度。

```
...
case WM_CREATE
{
    TEXTMETRICS tm;
    HDC hdc;

    hdc = GetDC ();
    GetTextMetrics (hdc, &tm);
    ...
    ReleaseDC (hwnd, hdc);
}
    break;
...
case WM_SIZE:
{
    RECT rc;

    // cy contains the system font height
    rc.bottom = cy * 10;
    ...

    // calculate the overall window size, including the no client area
    AdjustWindowRectEx (&rc,
                        WS_OVERLAPPEDWINDOW,
                        TRUE,
                        WS_EX_CLIENTEDGE | WS_EX_WINDOWEDGE);
    // set the new window dimension
    SetWindowPos (hwnd, HWND_TOP, rc.left, rc.top,
                  rc.right - rc.left,
                  0);
}
    break;
...
```

AdjustWindowRectEx () 函数会根据客户区矩形计算出窗口的整体尺寸。注意，这种尺

寸信息是作为指向某个 RECT 结构的一个地址，在函数的第一个参数里传递的。

```
#include〈winuser.h〉
BOOL AdjustWindowRectEx (LPRECT lpRect,
                         DWORD dwStyle,
                         BOOL bMenu,
                         DWORD dwExStyle);
```

参数	说明
LPRECT lpRect	一个 RECT 结构的地址，其中包含了客户区的尺寸定义
DWORD dwStyle	列出与窗口有关的基本窗口风格
BOOL bMenu	假如为 TRUE，表明窗口内有一个菜单栏
DWORD dwExStyle	列出与窗口有关的扩展风格
返回值	在正文里讨论
BOOL	布尔值，假如调用成功，返回一个 TRUE 值；失败则返回一个 FALSE 值

第一个参数里包含了窗口的整体尺寸，其中包括该窗口的基本和扩展风格。假如这里还存在一个菜单栏，那么这一信息就由第三个参数指定。只要根据目标客户区域就可知道整个窗口的尺寸，接下来就可以调用 SendWindowPos () 对窗口的尺寸和位置进行重新定义。

4.8.1 输出模式

GetDC () 和 GetDCEx () 是我们访问设备现场的第一种方案，然而它们并不是产生应用程序输出的最优方案。请大家观察图4-15，图内有一个窗口已被部分隐藏起来了。

后台窗口被隐藏不见的矩形属于更新区域或者无效区域。无论更新还是无效，这两种定义都涉及到一个矩形区域从窗口的客户区剪切下来。对于用户来说，这个区域暂时是看不见的，因为前台存在另一个窗口，那个窗口把这块区域遮掩住了。假如重新要求显示这个被遮掩住的区域，应用程序就会对这块区域进行重画。在任何给定的时刻，任何窗口内都有可能含有一块用户看不见的区域——亦即一块更新区域。从现在开始，假如应用程序准备在客户区内输出某些信息，大家就必须对更新区域进行相应的处理。

Windows 通过在自己的队列里放置一条 WM_PAINT 消息，它能通知应用程序对无效的区域进行重画（正如我们以后要讲到的那样，这种说法并不是完全准确的）。我们需要注意的一个问题是：每当有一个更新区域的时候，应用程序稍后必须对一条 WM_PAINT 消息进行处理。这时只有很渺茫的机会能够避开后续的 WM_PAINT 消息——这种情况太少见了！

基于这些前提，现在让我们试着归纳出支配 Windows 应用程序重画模式的总体规则。这其实相当简单：所有输出指令必须放置于应用程序代码内的一个单一位置处。换句话说，也就是必须放置于 WM_PAINT 消息内。这是在两种对立的需求间选择出的一种最优化妥协方案：窗口的叠置和输出的连续性。把应用程序的所有输出代码统一到一个地方（WM_PAINT）后，几乎肯定能保证即使整个客户区域都被另外一个窗口覆盖了，也能在窗口进入前台的时候恢复其正确的输出。图4-16应该能帮助大家理解这一概念。

现在让我们来观察图4-16。后台窗口的右下角被完全覆盖了，这是一种更新区域。假若有

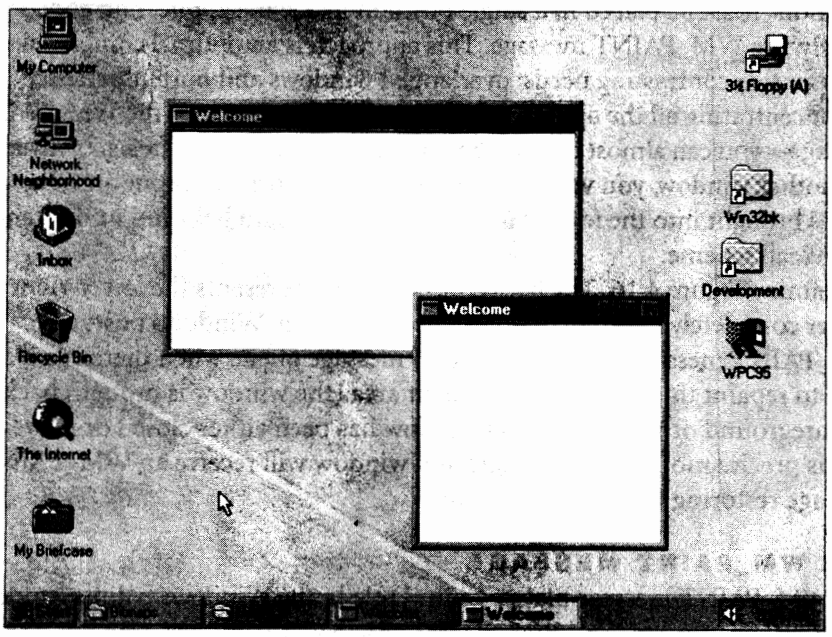


图 4-15　前台窗口把后台窗口的右下角隐藏起来了

必要对客户区的这一部分进行重画（窗口再次带入前台，或者前台窗口已被关闭或者移开），Windows 就会在应用程序队列里张贴一条 WM_PAINT 消息。随后，后台窗口就会接收到这条 WM_PAINT 消息，从而恢复自己的原始输出。

4.8.2　WM_PAINT 消息

WM_PAINT 消息在 Windows 重画模式里扮演了一个中心角色。一些作者趋向于把 WM_PAINT 定义成一种低优先级的消息。我不同意这种观点。确实，Windows 对这条消息的处理是比较独特的，例如，它不允许应用程序把该消息张贴或者发送给任何窗口。无论 PostMessage()还是 SendMessage()都不能和 WM_PAINT 联合使用。这儿问题出在 wParam 和 lParam 上面。这两个参数是由 Windows 直接保留和控制的，这样一来，消息发送或者张贴过程中分配的任何值都不会满足这条消息的真正需求。

WM_PAINT 另一个独特的地方在于它运作的方式。为了理解这一点，我们可以重新观察由图4-16展示的方案。后台窗口有一个更新区域。更准确地说，它是一个“潜在”的更新区域，只有在非常有必要的前提下（窗口必须进入前台），它最终才会进入一条 WM_PAINT 消息。在这种等待过程中，WM_PAINT 会一直驻留在消息队列以外，直到这个队列完全为空为止。

这种现象正是某种特定的策略需要的。通常，用 CPU 时间和视频刷新来衡量，各种输出操作都是相当费力的。除此以外，考虑到当前环境的多窗口现象，在未定消息的处理过程中，很容易就会对客户区的输出产生影响。假如 WM_PAINT 立即占据了消息队列中的一个位置，那么在后续的处理中，假如以前的消息进行了处理，就有可能在客户区内产生一些毫无效果的变更。但在消息队列外面等待就不会出现这种问题，因为这样就有机会按照消息处理的最新进展来依次更新被隐藏的客户区，不用担心出现更新无效的后果。这样一来，至少在下一次修改发生之前，更新处理就可以修改客户区的内容。

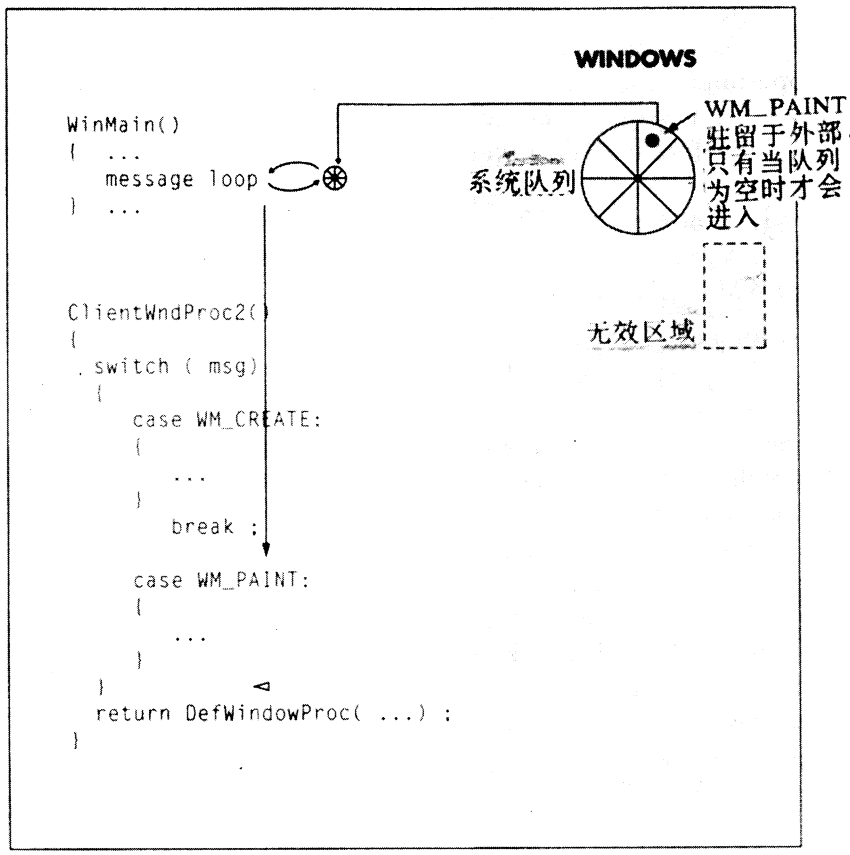


图 4-16　Windows 输出模式和 WM_PAINT 消息

更新区内的变化可能在两个方面发生。通常，更新区域都有扩大的趋势，当然也有可能会收缩得更小。对于已经放置到消息队列里的消息来说，由它引入的任何改变都会由新进的 WM_PAINT 消息得以反映。

拦截 WM_消息

到目前为止，我们都还没有拦截 WM_PAINT 消息的必要。尽管如此，我们的应用程序依旧能良好地运行。这就意味着就 DefWindowProc () 的内在功能来说，它已经知道如何如何与 WM_PAINT 消息打交道，知道如何对其进行处理。假如应用程序必须在客户区内部显示一些信息，它就会拦截 WM_PAINT，然后拟定一种策略，从而把有正确的信息按照需要放置到客户区内。用于显示由 BeginPaint () 获取的现场 (hdc) 的一个句柄在这儿是达到该目的的关键。比较由 GetDC () 返回的一个 HDC，它们之间是存在本质区别的。首先，对于 BeginPaint () 返回的句柄来说，它知道一个无效区域的存在，并且最重要的一点，这个句柄知道怎样对其进行处理。正是由于该句柄具有这种能力，所以 BeginPaint () 是 WM_PAINT 消息里使用的一种最适当的 API，它在其他任何一种消息里使用都是不合适的。和 GetDC () 一样，BeginPaint () 也提供了一种函数，用它可以释放出以前取得的 HDC，这个函数就是 EndPaint

()。

一条标准的 WM_PAINT 消息代码块看起来应该像下面这个样子：

```
...
case WM_PAINT:
{
    HDC hdc;
    PAINTSTRUCT ps;

    hdc = BeginPaint (hwnd, &ps);
    ...
    EndPaint (hwnd, &ps);
}
    break;
...
```

BeginPaint () 的语法要求使用一个窗口句柄和一个 PAINTSTRUCT 数据结构的地址：

```
HDC WINAPI BeginPaint (HWND hwnd, PAINTSTRUCT &ps);
```

在这儿，尽管很少使用，我们仍然不能忽略了 PAINTSTRUCT。它的各个项里有一个就是 HDC，其中包含了用于显示现场的句柄。

```
typedef struct tagPAINTSTRUCT
{
    HDC hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved [16];
} PAINTSTRUCT;
```

假如其中的布尔值项 fErase 等于 TRUE，就意味着窗口背景必须在更新区域重画之前清除。除此以外，假如 fErase 为 TRUE，它还会发出一条 WM_ERASEBKGND 消息，用于真正清除窗口背景。rcPaint 矩形与 hwnd 参数标识的窗口更新区域是对应的。最后三个项是保留项，只能由 Windows 本身使用。

4.9 背景的清除

与 WM_ERASEBKGND 有关的信息以及存储在 wParam 里的信息是与窗口 HDC 对应的，这个窗口的背景必须进行重画。

WM_ERASEBKGND	0x0014
wParam	窗口 hdc
lParam	未使用

DefWindowProc () 会用存储在窗口类内存区域内的颜色来删除窗口的背景。假如对应类的句柄可用，那么通过调用带有 GCL_HBRBACKGROUND 标志的 SetClassLong () 函数，很容易就可以对这些值进行修改。

4.9.1 屏蔽一个矩形

在前面讲述的与 Windows 重画模式有关的所有例子里，我总是断言存在一个无效（屏蔽）的区域，声明这种区域是一个或者多个窗口相互重叠造成的结果。后续的 WM_PAINT 消息将由 Windows 本身张贴到应用程序队列里。但是，这并不是能够生成 WM_PAINT 消息的唯一情况。应用程序本身可以专门屏蔽一个矩形区或者一个屏幕范围，从而导致一条 WM_PAINT 消息的产生。

这其实就是我们调用 InvalidateRect () 函数的结果：

```
#define 〈winuser.h〉
void WINAPI InvalidateRect (HWND hwnd,
                            CONST * LPRECT lpRect,
                            BOOL fErase);
```

参数	说明
HWND hwnd	客户区将被变成无效的窗口
CONST * LPRECT lpRect	无效矩形的地址
BOOL fErase	假如需要清除无效区域的背景，就设为 TRUE
返回值	在正文里讨论
void	没有返回值

第一个参数是窗口句柄，该窗口的客户区正是要由应用程序进行部分无效处理的那个。矩形更新区域的尺寸是用一种 RECT 结构定义的，该结构的地址是当作第二个参数传递的。左上角和右下角是用客户区坐标表达的。假如这个参数设置成 NULL，那么整个客户区就会变得无效。假如在实际的重画发生前必须先清除背景，布尔值就应为 TRUE；假如可以保留它当前的状态，则为 FALSE。

InvalidateRect () 生成了一个无效区域，并且在应用程序队列里张贴了一条 WM_PAINT 消息。调用 InvalidateRect () 最后得到的结果与张贴这样一条消息是类似的——不允许明目张胆地调用 PostMessage ()，因为无论 wParam 和 lParam 都会假定一些保留值。

迄今为止，一个窗口在任何方向进行了尺寸变化以后，我们建立的所有例子都能自动屏蔽整个客户区。这就是类注册时设置了 CS_VREDRAW 和 CS_HREDRAW 的结果。

4.9.2 显示一些正文

现在是在窗口客户区内显示一些正文的时候了。这儿提供了一个名为 Milan（米兰）的例

子——由我最喜爱的足球队得名，它为我们提供了一个绝好的机会。利用这个机会，我们可以探索与重画模式、更新区域有关的一些附加特性。Milan 这个例子可以在本书附带 CD 的 Listing 4.4里找到，如图4-17所示，它在客户区里显示了三个正文串。

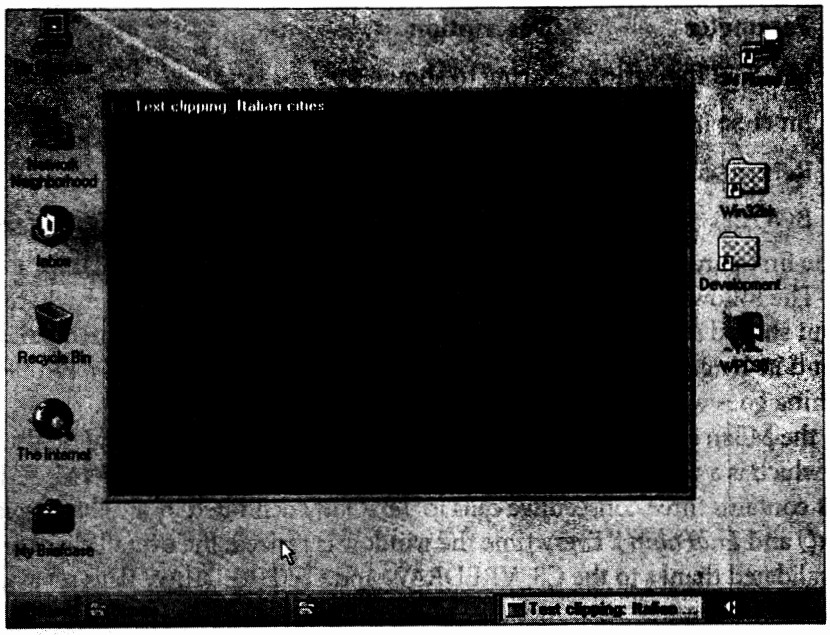


图 4-17　Milan 内的三个正文串沿着 X 轴排列，同时在 Y 轴上有一个固定的偏移

三个串存储于指向字符标识符的指针数组里，并且具有静态存储类。为了显示这些正文，我们需要用到 TextOut（）函数，这是由 Windows API 提供的、进行字串输出的最佳途径：

```
#include〈wingdi.h〉
BOOL WINAPI TextOut（HDC hdc，
                     int nXStart，
                     int nYStart，
                     LPCSTR lpszString，
                     int cbString）；
```

参数	说明
HDC hdc	一个有效的显示现场句柄
int nXStart	开始输出的 X 轴左上角位置
int nYStart	开始输出的 Y 轴左上角位置
LPCSTR lpszString	准备显示的串
int cbString	串的长度
返回值	在正文里讨论

BOOL	布尔值，假如函数调用成功，就返回一个 TRUE 值；假如失败则返回 FALSE

第一个参数是显示现场的句柄，这对于所有 GDI 函数来说都是一致的。接下来的一对参数指定了输出起始处的左上角位置。准备显示的正文串是用第四个参数表示的，它的长度则存储在第五个参数里。假如 TextOut () 调用成功，它会返回一个 TRUE 值；如果由于其他什么原因失败了，则返回 FALSE。

在 Milan 这个例子里，三个正文串在 Y 轴上面存在着标准的间隔距离，这是一个足够大的值，可以防止任何可能的重叠。WM_PAINT 块内包含了对 TextOut () 三次连续的调用，它们是由对 BeginPaint () 和 EndPaint () 的调用分隔开的。每次对窗口的尺寸进行变化的时候，由于我们在类注册的时候设置了 CS_VREDRAW 和 CS_HREDRAW 标志，所以整体的客户区域都变得无效了。

到目前为止，在 Milian 这个例子里还没有什么引起我们特别注意的。对 ShowWindow () 的一次调用或者 WS_VISIBLE 窗口风格的存在都会生成一条 WM_PAINT 消息，这就是这种 API 或者 CreateWindowEx () 的副作用。一旦窗口在屏幕中显示出来，就证明了三个正文串在应用程序客户区域内的存储。

Milan 程序会拦截窗口进程里产生的鼠标右键单击事件，然后生成一条对应的 WM_RBUTTONDOWN 消息。这个简单的操作与窗口重画本身是没有什么关系的，但是在代码里，第二和第三个串是按照下面这种形式显示的：

Milan	Unchanged
Florence	Rome
Venice	Naples

对串进行替换并不能反映出显示上的变化。如果想对输出进行刷新，必须先让整体客户区域无效（屏蔽），并且最终把一条 WM_PAINT 消息张贴到应用程序消息队列里。在这个时候，三条 TextOut () 语句就会再次执行，从而显示出最后两个新的正文串。尽管很容易就可以整体客户区域进行屏蔽，但这并非一种最优的方案，因为只有有限的区域需要重画。所以，我们采纳了一种更简单的方案，如下所示：

```
InvalidateRect (hwnd, NULL, TRUE);
```

这种方案显得更有针对性，它只屏蔽了围绕于中心正文串（第二个）的矩形区域。在进行这种处理之前，必须先知道第二个正文串在屏幕中的尺寸。起始点（左上角）是已知的，我们只需计算出右下角的位置就可以了。矩形的两个始角如图4-18所示。

正如前面指出的那样，我们知道 Florence 正文串的左上角位置，而右下角位置却是完全不知道的。起始点存储于一个 RECT 结构里，但我们不得不对它的整体范围进行度量。为了达到这一要求，GetTextExtentPoint32 () 是我们的正确选择：

```
#include 〈wingdi.h〉
BOOL GetTextExtentPoint32 (HDC hdc,
                           LPCTSTR lpString,
                           int cbString,
                           LPSIZE lpSize);
```

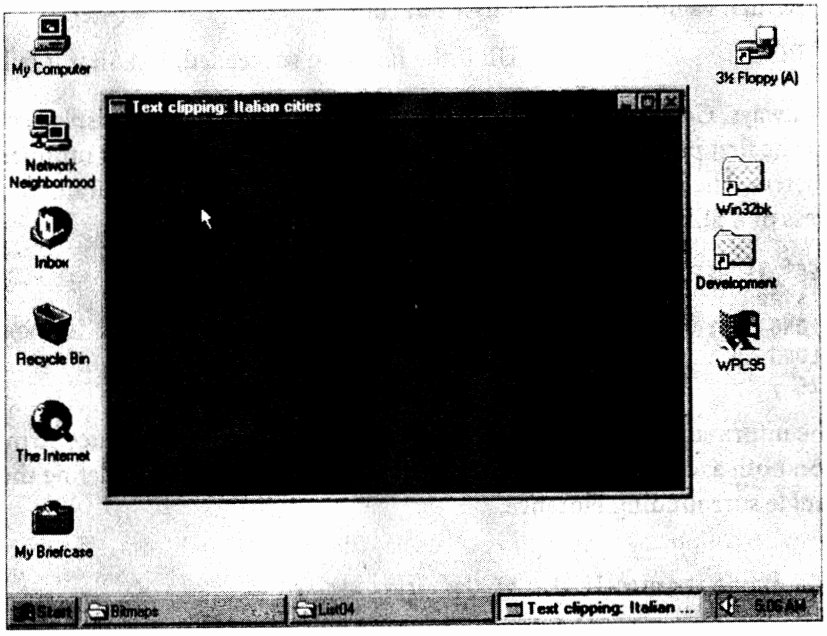


图 4-18　计算第二个正文串（Florence）在显示器上的尺寸

参数	说明
HDC hdc	一个有效的显示现场句柄
LPCTSTR lpString	正文串
int cbString	用字符数表示的串长度
LPSIZE lpSize	一个 SIZE 数据结构的地址
返回值	在正文里讨论
BOOL	布尔值，假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回 FALSE

和往常一样，GetTextExtentPoint32 () 要求用到针对一个有效显示现场的句柄，并且把它当作自己的第一个参数使用，接下来是准备度量的正文串的字符数。正文串的宽度和高度是在第四个参数里传递的——即一个 SIZE 数据结构的地址：

```
typedef struct tagSIZE
{ //size
    LONG cx;
    LONG cy;
} SIZE;
```

SIZE 结构里的信息与两个轴上的 Florence 串尺寸是对应的。这些值放置于某个 RECT 结构里，从而定义了围绕于 Florence 的矩形尺寸：

```
...
// determining the overall size of the second string
hdc = GetDC (hwnd);
GetTextExtentPoint32 (hdc, szString [1], lstrlen (szString [1]), &size);
ReleaseDC (hwnd, hdc);

...

// calculating the rectangle to be invalidated
rc.left = rc.right = X;
rc.top = rc.bottom = Y1;
rc.right += size.cx;
rc.bottom += size.cy;
...
```

对矩形进行屏蔽之前，还要对最后两个串进行替换：

```
...
// changing the second and the third strings
lstrcpy (szString [1], " Rome");
lstrcpy (szString [2], " Naples");
...
```

现在该对矩形进行屏蔽了，InvalidateRect () 是我们能够利用的最恰当的工具，紧接着应该再使用一个 UpdateWindow ()：

```
...
InvalidateRect (hwnd, &rc, TRUE);
UpdateWindow (hwnd);
...
```

UpdateWindow () 的作用是强制 WM_PAINT 消息传递到目标窗口，而且不必像往常那样在队列外面等待。从本质上说，InvalidateRect () 和 UpdateWindow () 的组合会使重画同步进行，其间没有任何等待。生成的结果如图4-9所示。

好了，我在这里好象并没有出错。图4-19里的三个正文串是 Milan（米兰），Rome（罗马）和 Venice（威尼斯）。但是 Naples（那不勒斯）到哪儿去了呢？难道有什么不对吗？假如在 WM_PAINT 消息块里放置一个断点，我们就可以推断出所有正文串都已成功地改变了（参见图4-20），所以肯定是程序的其他部分有错。

根据函数 GDI 子集内输出剪切的内部机制，图4-20出现的情况是很正常的。如果屏蔽了

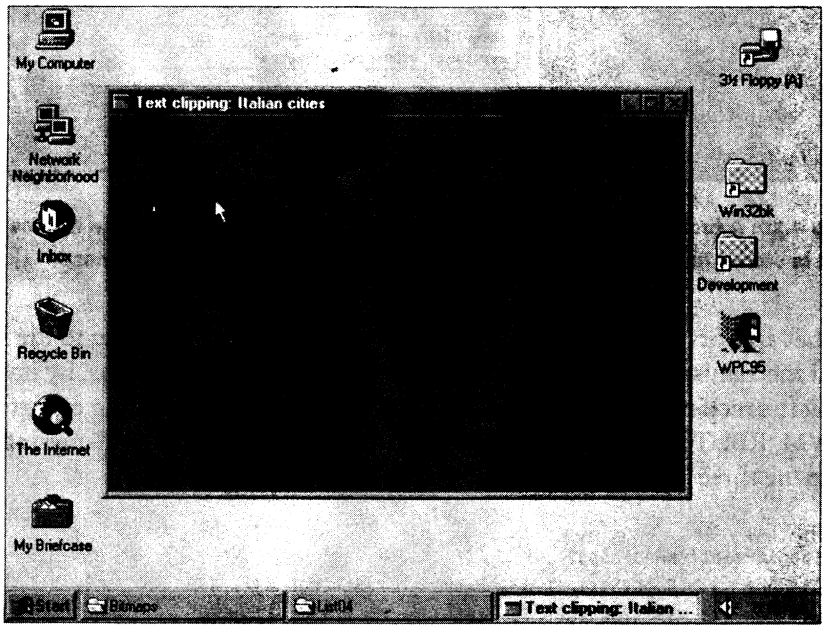


图 4-19　单击鼠标右键后，Milan 的输出情况

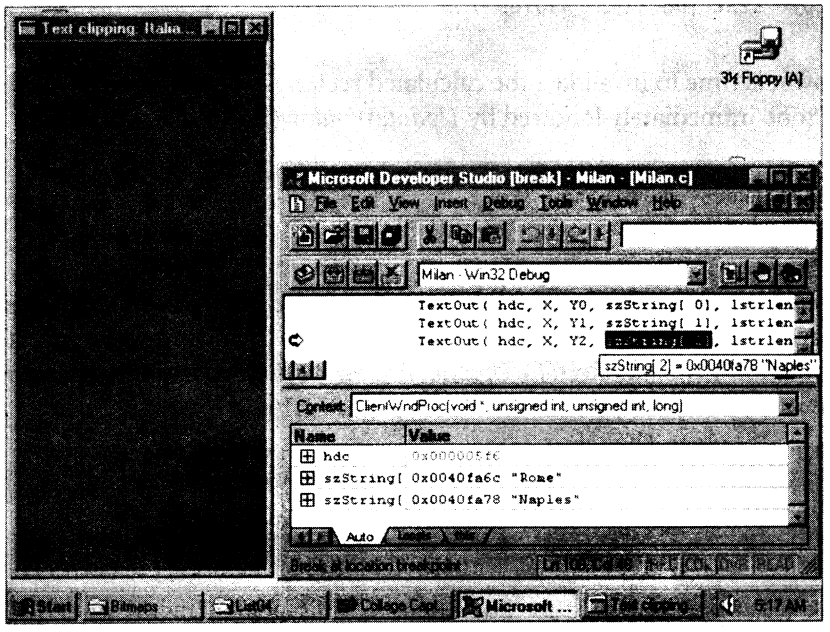


图 4-20　WM_PAINT 消息块里的一个断点显示 Naples 串存储于 szString 字符指针数组里的第三个位置处

与 WM_RBUTTONDOWN 消息里第二个正文串尺寸对应的矩形，就会生成一条 WM_PAINT 消息，三个 TextOut（）函数就驻留于这条消息内：

```
...
hdc = BeginPaint (hwnd, &ps);
TextOut (hdc, X, Y0, szString [0], lstrlen (szString [0]));
TextOut (hdc, X, Y1, szString [1], lstrlen (szString [1]));
TextOut (hdc, X, Y2, szString [2], lstrlen (szString [2]));
EndPaint (hwnd, &ps);
...
```

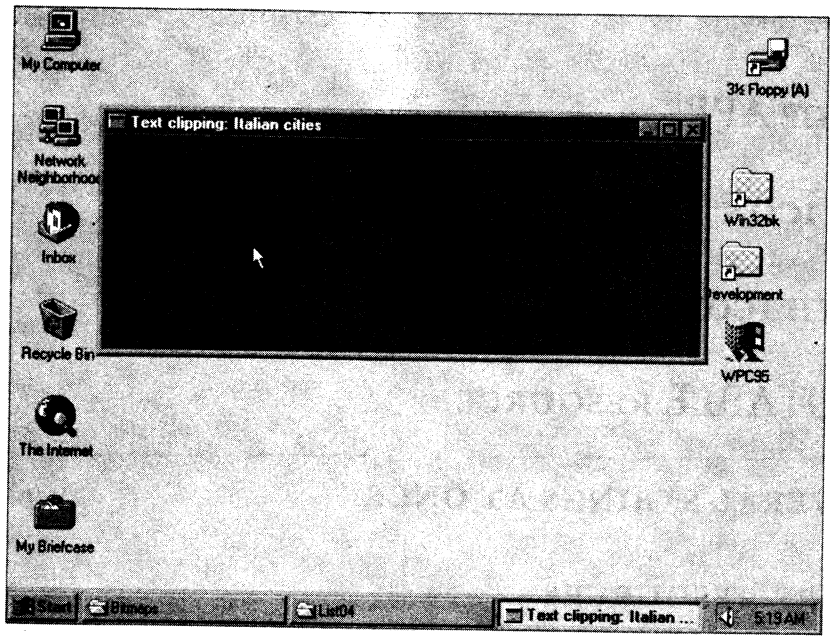


图 4-21　按下鼠标右键并且改变了整个窗口的尺寸后，Milan 的显示情况

对 WM_PAINT 进行处理的过程中，三个 TextOut () 函数是按照一种精密的顺序一个接一个执行的。由 BeginPaint () 返回的 HDC 句柄与 GetDC () 返回句柄不同，因为后者不可能对更新区域进行处理。对于指定 Milan 的第一个 TextOut () 来说，它的输出与当前的更新区域是沾不上边的。所以，这个函数不会进行任何实际的输出操作，而且立即就能返回。事实上，根据更新区域的当前状态，屏幕上的显示在这个函数执行以后将不会发生任何变化。由第二个 TextOut () 生成的输出却不能按照这种方式来实现，因为它与更新域（无效区域）是完全对应的。这就是为什么 Rome 在屏幕上显示出来的原因。第三个 TextOut () 函数的表现与第一个差不多，它也会立即返回，因为它的输出与当前的更新域没有什么关系。然而，在这种情况下，还是存在一点细微的差别，因为实际的正文串已经发生了变化。无论怎样，我们都无法在屏幕中看到 Naples，除非这时改变了整个窗口的尺寸。改变了窗口大小后，整体客户区域都会进行重画，而且 Milan，Rome 和 Naples 会同时在屏幕上显示出来。

Windows 会对自己 GDI 函数的输出进行剪切，然后把它们放置到当前的更新区域内，这样就减少了花在输出处理上的时间，同时也优化了 CPU 的性能。

第5章　资源文件

对于图 2-7 描述的那种 Win32 开发模式来说，几乎一定需要在标准的程序里用到某些应用程序资源。现在让我们观察图 5-1 展示的流程，特别要注意其中的资源问题。

.RC 文件在资源定义的建立中扮演了一种中心角色。这是一种 ASCII 文件，它的内容随着项目的不同会不断地发生变化。正如我们在上一章强调过的那样，资源文件里包含了两种形式的资源：正文和二进制。正文资源通常存放于 .RC 文件里，然而它们也可以在独立的文件内存储。在后面那种情况下，.RC 文件内包含了对那些文件的引用。

二进制资源不能用 .RC 文件来包含了描述。这种资源的本质要求使用独立的文件，而且必须用二进制格式存储信息。所以，.RC 文件对这些资源的引用是按照特定的语法实现的。表 5-1 列出了 Windows 95 里实现的所有正文资源，表 5-2 则对二进制资源进行了总结。

表 5-1　正文资源

菜单模板
版本
对话框模板
串表
加速器表

表 5-2　二进制资源

字体
位图
元文件
光标
图标

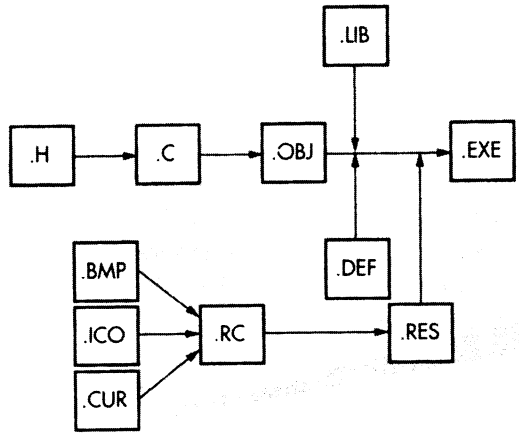


图 5-1　一个 Win32 应用程序的开发流程图内涉及到 .RC 文件的应用

不管资源的本质如何，资源编译器（RC.EXE）都能读取 .RC 文件，然后把它转换成一个 .RES 文件——资源的编译版本。随后，.RES 文件将插入特定位置处的执行流程内，插入的依据是 .PE（可移植 .EXE）格式。这样一来，.EXE 文件的缔造者除了应用程序源代码和数据以外，另外还要加上 .RC文件里定义的资源。

现在，由于诸如 Visual C++的开发环境都提供了对应的工具，资源文件的生成已经变得

相当简单了。开发者很少需要对资源文件原本进行检查，因为集成化的工具能够对资源的建立、更新、删除和修改进行全方面的控制，如图 5-2 所示。

资源组是在 WINUSER.H 里通过 RT_ 前缀定义的，如表 5-3 所示。MAKEINTRESOURCE 宏内的值与特定的编号是对应的，它可以从一个执行文件或者标签未知的一个 DLL 内载入特定种类的资源。

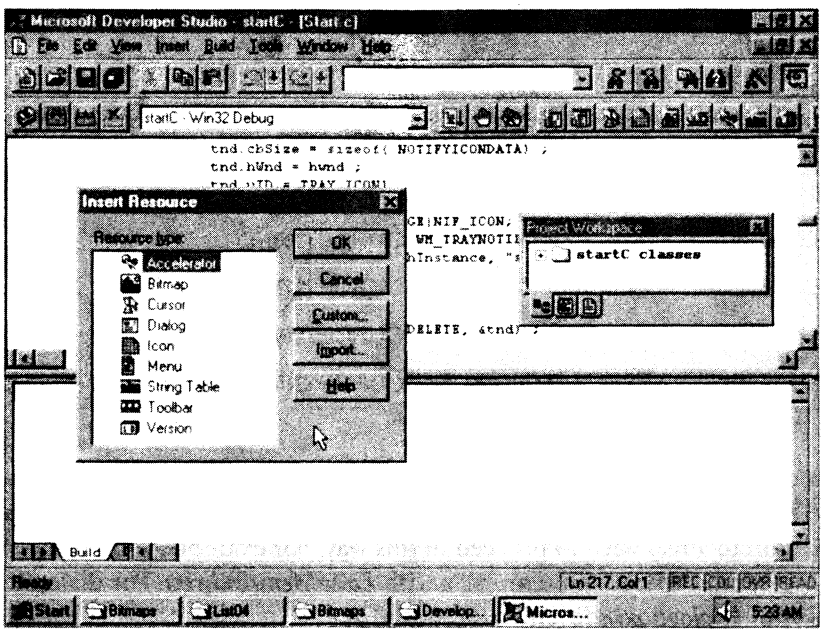


图 5-2　Visual C++通过 Resource 顶级菜单对资源进行控制

表 5-3　WINUSER.H 里的 RT_定义——用于标识 Win32 的资源组

定　义	值
RT_CURSOR	MAKEINTRESOURCE (1)
RT_BITMAP	MAKEINTRESOURCE (2)
RT_ICON	MAKEINTRESOURCE (3)
RT_MEUN	MAKEINTRESOURCE (4)
RT_DIALOG	MAKEINTRESOURCE (5)
RT_STRING	MAKEINTRESOURCE (6)
RT_FONTDIR	MAKEINTRESOURCE (7)
RT_FONT	MAKEINTRESOURCE (8)
RT_ACCELERATOR	MAKEINTRESOURCE (9)
RT_RCDATA	MAKEINTRESOURCE (10)
RT_GROUP_CURSOR	MAKEINTRESOURCE (12)
RT_GROUP_ICON	MAKEINTRESOURCE (14)
RT_VERSION	MAKEINTRESOURCE (16)
RT_DLGINCLUDE	MAKEINTRESOURCE (17)
RT_PLUGPLAY	MAKEINTRESOURCE (19)
RT_VXD	MAKEINTRESOURCE (20)

除了现成的资源文件以外，资源也可以在程序执行过程中现场生成。这种方式一般都很少采用，况且除了真正需要这样做以外，还需要很好的编程技巧才行。例如，我们可以用LoadMenuIndirect（）动态建立运行期菜单模板，或者用DialogBoxIndirect（）建立对话框模板。

5-1 资源 API

Win32 提供一种 HRSRC 类型的句柄，设计它是为了专门对一般资源进行控制。FindResource（）这个 API 可以返回一个 HRSRC 句柄，然后可以再调用 LoadResource（）函数，从而把资源数据载入应用程序地址空间内。

```
#include〈winbase.h〉
HRSRC WINAPI FindResource（HINSTANCE hinstance，
                           LPCSTR lpszName，
                           LPCSTR lpszType）；
```

参数	说明
HINSTANCE hinstance	应用程序实例句柄
LPCSTR lpszName	资源名称
LPCSTR lpszType	资源类型
返回值	在正文里讨论
HRSRC	资源句柄；假如函数调用失败，则返回一个 NULL 值

第一个参数是应用程序的实例句柄——这对于我们来说已经是一种古老的知识了，它的作用是引用包含了我们准备载入的资源的一个模块。在此我们重复声明；考虑到 Win32 的每个进程都具有相同的 hInstance 值，所以我们只能从当前的模块内对某种资源进行访问。第二个参数是 lpszName，它指定了资源的名称。这个参数可以是一个正文串，也可以一个数字。假如是数字，那么对应的信息就存储于低字内，同时把高字设置成零。lpszType 具有类似的特性，这个参数的低字内包含了由表 5-3 列出的一种_RT 定义。

一旦资源成功进入了资源文件内的某个地方，接下来就可以调用 LoadResource（）了：

```
#include〈winbase.h〉
HGLOBAL WINAPI LoadResource HINSTANCE hinstance，HRSRC hrsrc）；
```

参数	说明
HINSTANCE hinstance	应用程序实例句柄
HRSRC hrsrc	资源句柄
返回值	在正文里讨论
HGLOBAL	资源句柄；假如函数调用失败，则返回一个 NULL 值

函数内的两个参数指定的是应用程序实例句柄和资源句柄，这两个句柄是前面通过调用 FindResource（）获得的。返回值句柄是指一个内存区域，其中包含了实际的资源数据。为了

把它转换成一个指针，我们需要用到 LockResource（）：

```
#include〈winbase.h〉
LPVOID LockResource（HGLOBAL hglb）;
```

返回的指针允许我们对那个内存区域内的信息进行访问和读取。在 Win32 里，我们没有必要对资源使用的内存区域进行解锁或者释放。所以，以前的 UnlockResource（）和 FreeResource（）在 Win32 里已经不再获得支持，而且也没有使用了。

对于资源的访问和控制来说，刚才叙述的实际只是一些低级手段。实际上，Win32 API 针对资源问题专门提供了一系列完整的函数，它们都是经过专门设计的，用于解决特定的问题。利用这些函数，我们往往只需一个步骤就可以实现资源的定位、访问以及装载。举个例子来说，假如你想载入一幅位图，就可以考虑使用 LoadBitmap（）或者功能更强的 LoadImage（）。典型的菜单通常都是通过 LoadMenu（）访问的，而 LoadAccelerators（）则专门设计用于装载一张加速键表。在本章和下一章，我们将对其中一些函数进行详细的介绍。就目前来说，我们需要先在应用程序内增加一些资源。下面让我们先在资源文件里放置一个图标。

5.2 图标的载入

图标是一幅尺寸较小的位图，它的大小是相当标准的固定的。Win32 API 对普通位图和图标进行了明显的区分，因为它针对图标处理专门提供了一系列特定的 API。除此以外，图标在 Win32 程序的开发过程中扮演着某种特殊的角色，对此我将在下面进行详细解释。

在过去，图标只是用于代表屏幕上某个窗口在最小化时的状态。在 Windows 95 里，这种限制已经取消了，因为最小化处理在面向对象的外壳里具有一种完全不同的含义。在窗口的左上角仍然有一个小图标，通过它可以引入应用程序的系统菜单。另外，启动一个新进程后，就会建立一个任务栏按钮，它会从任务栏的左侧开始排列。Win16 和 Win32 环境内图标运用的另外一种明显的区别是和它们的尺寸有关的。在过去，图标一般都是 32×32 象素的位图，而 Windows 95 则支持三种不同的图标格式：16×16（小图标），32×32（普通图标）以及 48×48（大图标）。正如大家在第 3 章 Win32 应用程序的开发”里学到的那样，即使对于注册类 WNDCLASSEX 来说，它现在都提供了对一种新项的支持——该项引用了存储于应用程序类数据内的一个小图标。由此看来，难道图标在 Windows 95 里已经没有过去重要了吗？并非完全如此；它们现在在总体结构里扮演的是一种完全不同的角色。

我们在 .RC 文件里无法物理性地声明一个图标。事实上，我们只能在其中插入对某个特定文件的引用。从物理意义上说，关于图标的具体信息就是存储于那个文件内的。具体的语法结构如下所示：

```
label ICON file_name
```

其中，label（标签）可以是自己希望的任何名字——甚至可以是一个数字。事实上，第二种方案（选用数字）现在已经变得越来越流行，MS Visual C++生成一种新资源时常常需要采用那种方案。文件名是指文件系统内的一个物理文件，由于 Win95 的新特性，我们知道这个文件并不只是局限于传统的 8.3 字符命名规范。

下面是一个 .RC 文件对图标进行声明的例子：

```
...
TWENY icon TWENY.ICO
```

```
...
```

其中，TWENY 是一个标签，而 TWENY.ICO 则是包含了原始数据的那个文件。

现在让我们假设已经成功编译和链接了这个应用程序。怎样才能在执行过程中从 .EXE 文件里载入某个图标呢？好了，这时只需记住使用 LoadIcon () 就可以了：

```
#include <winuser.h>
HICON WINAPI LoadIcon (HINSTANCE hInstance, LPCSTR lpszIcon);
```

参数	说明
HINSTANCE hinstance	应用程序实例句柄
LPCSTR Icon	图标标签
返回值	在正文里讨论
HICON	图标句柄；假如函数调用失败，则返回一个 NULL 值

LoadIcon () 是我们讲述的所有 API 中对资源进行处理的第一个例子。这个函数能够返回应用程序实例句柄，以及图标的标签。仍旧拿前面那个例子来开刀，它现成变成了：

```
...
hicon = LoadIcon (hInstance, " TWENY");
...
```

其中，hicon 是类型为 HICON 的一个句柄。MAKEINTRESOURCE 宏可以把一个整数值转换成串值。对于 LoadIcon () 的第二个参数来说，串值才是它真正需要的东西。所以，我们可以假设一个图标资源在 .RC 文件里是这样声明的：

```
...
12 icon TWENY.ICO
...
```

要载入这个图标，可以按照下述的语法调用 LoadIcon () 函数：

```
...
hicon = LoadIcon (hInstancee, MAKEINSTRESOURCE (12));
...
```

请记住，WNDCLASSEX 数据结构提供了两个需要使用图标句柄的项：hIcon 和 hIconSm。现在大家已经知道了 LoadIcon () 的工作原理，接下来可以轻松地修改自己的代码了：

```
...
wcex.hIcon = LoadIcoon (hInstance, " TWENY");
...
```

在本书附带 CD 的 Listing 5.1 内，大家可以找到一个名为 ICONLDR 的程序示例，如图 5-3 所示。这个应用程序的系统菜单是由一个定制图标代表的，在客户区内则显示了四个不同的图标和一幅位图。

应用程序图标已经注册成了两种格式：普通图标 (32×32) 和小图标 (16×16)。我画了两个非常简单的图标(请大家原谅我的那几个有限的艺术细胞)，它们的样子显示于图 5-4 内。

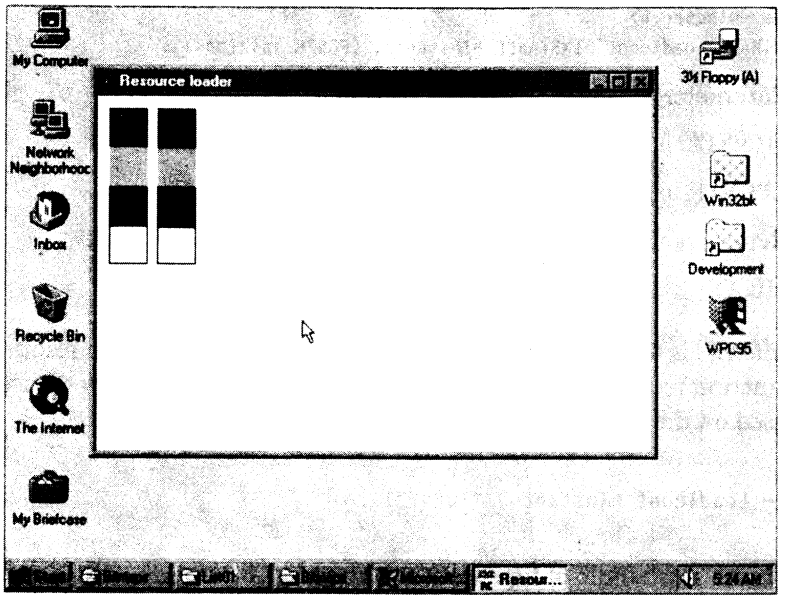


图 5-3　对主窗口类进行注册的时候，ICONLDR 载入了一个标准的小图标，并在客户区内显示了四个图标和一幅位图

普通图标会在应用程序的标题栏和任务栏按钮上显示出来。由此看来，小图标的用场何在呢？答案请在图 5-5 里找！这些小图标可以在以细节视窗显示的文件夹窗口内派上用场。在图 5-5 里，普通图标已被一些更小的图标取代了。

把某个图标与应用程序的系统菜单联系起来是非常容易的，只需要在类注册的时候进行一些简单的处理即可：

```
...
wcex.hIcon = LoadIcon (hInstance, " iconldr");
wcex.hIconSm = LoadIcon (hInstance, " iconldr");
...
```

Visual C++把普通图标和小图标存储于一幅独立的位图内，并用一个唯一（不能重复）的标识标签对该位图进行引用。那么谁来把它们分隔成两个独立的部分，分别由应用程序和系统本身对其进行访问呢？对于这一点，开发者是不用操心的，系统自己可以完成这一工作。

让一个图标在客户区内显示出来并不一件容易的事情。首先，开发者仍然必须从资源文件里把它载入，然后才能在希望的位置显示出来。实际上，ICONLDR 程序总共载入了四个不同的图标，然后一个接一个显示出来。这些工作的一部分是在 WM_CREATE 里完成的：

```
...
// icons loading
hicon [0] = LoadIcon (hInstance, " red");
hicon [1] = LoadIcon (hInstance, " green");
```

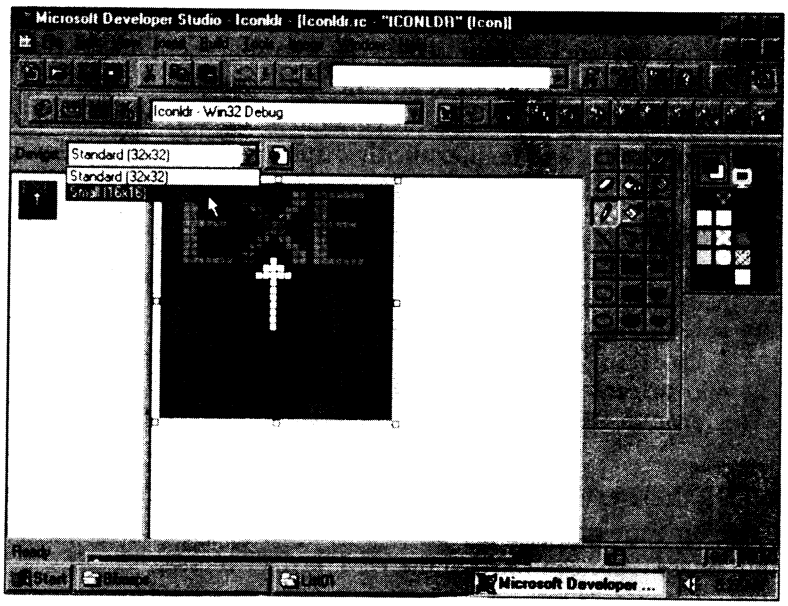


图 5-4　ICONLDR 里普通图标和小图标

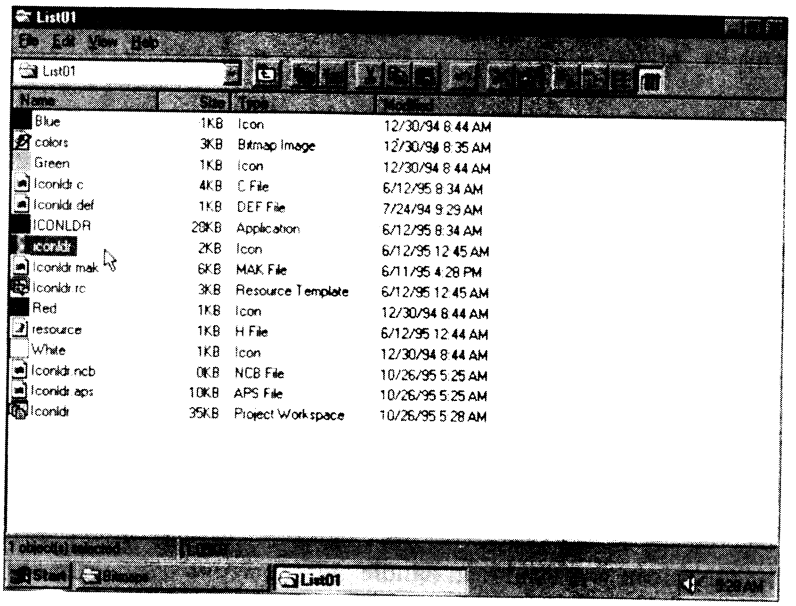


图 5-5　以细节视窗显示的一个文件夹，其中列出了缩小的普通图标，或者采用了自带的小图标（如果有的话）

```
hicon [2] = LoadIcon (hInstance, " blue");
hicon [3] = LoadIcon (hInstance, " white");
```

```
...
```

ICONLDR 在窗口进程内声明了由四个 HICON 句柄组成的一个数组，并为其分配了静态存储类。接着再按照标准的进程，把输出阶段集中在 WM_PAINT 消息里：

```
...
hdc = BeginPaint (hwnd, &ps);
for (i = 0; i < NUMICON; i++)
    DrawIcon (hdc, 10, 10 + SYS (SM_CXICON) * i, hicon [i]);
...
EndPaint (hwnd, &ps);
...
```

DrawIcon () 是一种 Win32 API，它的主要任务就是对图标的输出进行管理。该函数的语法是非常直观的：

```
#include <winuser.h>
BOOL DrawIcon (HDC hDC,
               int X,
               int Y,
               HICON hIcon);
```

参数	说明
HDC hDC	一个无效设备现场的句柄
int X	X 轴方向左上角的位置
int Y	Y 轴方向左上角的位置
HICON hIcon	一个有效的图标句柄
返回值	在正文里讨论
BOOL	布尔值，假如函数调用成功，就返回 TRUE；如果失败，则返回 FALSE

我们在 WIN32BK.H 里定义了 SYS 宏。这个宏里封装了 GetSystemMetrics () 函数，该函数的作用是取得与用户界面内几种组件有关的一些有用信息。在当前这种特定的情况下，GetSystemMetrics ()可以测量图标宽度——由 SM_CXICON 定义完成。现在，我们应该清楚了为什么要把图标句柄数组定义成静态。图标最初装入的地方是 WM_CREATE 消息，而对它们的描绘则是在 WM_PAINT 消息里完成的。所以，为了使两种不同消息间传递的消息不致于走样，最简单的办法就是把它声明成整个窗口进程内通用，并且具有静态的存储类。

在四个垂直排列图标的右面，我还放置了一幅位图。位图的总高与四个图标加起来的高度是完全一致的。要达到这样的效果很简单，只需用 32×4 象素计算位图的高度即可。然而，这幅位图是通过什么载入的呢？LoadBitmap () 是我们最恰当的选择：

```
#include <winuser.h>
```

```
HBITMAP LoadBitmap (HINSTANCE hinst,
                    LPCTSTR lpszBitmap);
```

参数	说明
HINSTANCE hinst	应用程序的实例句柄
LPCTSTR lpszBitmap	位图标签
返回值	在正文里讨论
HBITMAP	位图句柄；假如函数调用失败，则返回一个 NULL 值

LoadBitmap () 的语法与刚才介绍的 LoadIcon () 是完全一致的：实例句柄后面跟上一个资源标签。ICONLDR 程序将在拦截 WM_CREATE 消息的时候载入这幅位图：

```
...
// bitmap loading
hbmp = LoadBitmap (hInstannce, " colors");
...
```

再声明一句，hbmp 已被定义成静态存储类，它在整个窗口进程内都是通用（可见）的。位图输出是一种更为复杂的操作，因为 Win32 API 没有提供一种直接了当的 DrawBitmap () 函数。这就意味着开发者必须避开这种不足，在 WM_PAINT 消息里提供自己的一套方案才行。

首先，我们需要建立一个兼容的显示现场。这种现场只是计算机 RAM 内简单的一部分，它可以仿真标准的显示 DC——包括它的基本特征和特性（分辨率和每象素的颜色数等等）。从本质上说，它只不过是标准输出 DC 的一个“复印件”罢了。

做好上面这种准备工作以后，接下来需要在内存 DC 里为自己选择一幅位图。假如大家对第四章“消息和重画模式”还有印象，就应该记起我们曾经提到过位图是 DC 的基本元素之一。在内存 DC 里选择了位图以后，位图就拷贝到了 DC 里，尽管此时并未在屏幕上发生任何变化。这正是我们预期的结果，因为内存 DC 是系统内存一个单一的区域，它与应用程序的输出活动是没有什么关系的。

在进行实际的操作前，先要用 GetObject () 函数对位图信息进行检查，把位图的尺寸存储到一个 BITMAP 结构里——这个结构是由 bm 标识符指定的。

现在，我们要着手开始显示位图了，也就是说最终要把它从内存 DC 拷贝到屏幕上。BitBlt () 是胜任这一工作的最佳工具。

```
...
// create a compatible DC
hdcmem = CreateCompatibleDC (hdc);
// select the bitmap in the compatible DC
hbmpold = SelectObject (hdcmem, hbmp);

// get the bitmapsize
```

```
GetObject (hbmp, sizeoof (bm), (LPSTR) &bm);
// show the bitmap
BitBlt (hdc,
        50,
        10,
        bm. bmWidth,
        bm. bmHeight,
        hdcmem,
        0, 0,
        SRCCOPY);
...
```

BitBlt () 函数的语法结构如下所示：

```
#include <wingdi. h>
BOOL BitBlt (HDC hdcDest,
             int nXDest,
             int nYDest,
             int nWidth,
             int nHeight,
             HDC hdcSrc,
             int nXSrc,
             int nYSrc,
             DWORD dwRop);
```

参数	**说明**
HDC hdcDest	目标设备现场
int nXDest	在 X 轴上的目标位置左上角
int nYDest	在 Y 轴上的目标位置左上角
int nWidth	目标对象的宽度
int nHeight	目标对象的高度
HDC hdcSrc	源设备现场
int nXSrc	在 X 轴上的起点位置左上角
int nYSrc	在 Y 轴上的起点位置左上角
DWORD dwRop	光栅操作代码
返回值	在正文里讨论
BOOL	布尔值，假如函数调用成功，就返回 TRUE；如果失败，则返回一个 FALSE 值

从表面看起来，BitBlt（）好象是一种相当复杂的函数。然而，它的逻辑是相当直接的，它的任务就是简单地把图象从一个 DC 拷贝到另外一个 DC。第一个参数是目标 DC，紧接着是位图在目的地显示时左上角的坐标。点（50，10）位于四个图标内第一个图标的同一条水平线上，只是稍稍靠这些图标的右侧。接下来的两个参数指定了位图的尺寸（cx 和 cy）。这就是我们为什么要用GetObject（）对位图进行测量的原因。其次，BitBlt（）还需要用到源 DC 和位图拷贝最初的起始位置。最后一个参数是光栅操作代码，这些代码可在表 5-4 内选用。

表 5-4　光栅操作代码

代　码	说　明
BLACKNESS	对目标矩形进行填色处理，采用的颜色是与物理调色板内的索引 0 联系在一起的
DSTINVERT	反转目标矩形
MERGECOPY	在布尔运算符 AND 的基础上，把源矩形的颜色与指定的图案合并到一起
MERGEPAIINT	布尔运算符 OR 的基础上，把反转源矩形的颜色与目标矩形的颜色合并到一起
NOTSRCCOPY	把反转的源矩形拷贝到目标位置处
NOTSRCERASE	通过布尔运算符 OR，把源矩形和目标矩形的颜色组合到一起，然后对生成的颜色进行反转
PATCOPY	把指定的图案拷贝到目标位图内
PATINVERT	把指定图案的颜色与目标矩形的颜色组合到一起，基于 XOR 布尔运算符
PATPAINT	在布尔运算符 OR 的基础上，把图案的颜色与反转源矩形的颜色组合到一起。这个操作得到的结果颜色再通过布尔运算符 OR 与目标矩形的颜色组合到一起
SRCAND	使用布尔运算符 AND，把源矩形的颜色与目标矩形的颜色组合起来
SRCCOPY	把源矩形直接拷贝到目标矩形内
SRCERASE	通过布尔运算符 AND，把目标矩形的反转颜色与源矩形的颜色组合到一起
SRCINVERT	在布尔运算符 XOR 的基础上，把源矩形和目标矩形的颜色组合起来
SRCPAINT	通过使用布尔运算符 OR，把源矩形的颜色与目标矩形的颜色合并到一起
WHITENESS	对目标矩形进行填色处理，采用的颜色是与物理调色板内的索引 1 联系在一起的

5.3　图标的运用

在本书附带 CD 的 Listing 5.2 里，大家可以找到一个名为 Icon 的例子，这是一个 Win32 应用程序，它最多可在应用程序客户区内显示 25 个图标。Icon 一个特别的地方在于每个图标对象都是以圆的形状显示的，不是标准的方形图标。实际上，应用程序画出来的仍然是一种方形图标，但是用户只能对其中的某个圆形部分进行访问，由此就生成了这种效果。图 5-6 向大家展示了刚开始执行 Icon 时的显示情况。

ICON 并不是一个高度复杂的应用程序。假如用鼠标左键单击某个图标，程序就会自动生成一个新图标。每个图标都代表了一个不断增大的标识编号，客户区内图标的总数被限制在 25 个以内。每次建立一个图标的时候，整个客户区都会暂时屏蔽（无效），强迫应用程序随机替换现成的对象，如图 5-7 所示。

每个图标都有一个标识编号，这种编号是用黑色在透明的背景上书写的。这种效果是通过调用 TextOut（）获得的，该函数要先于 SetBkMode（）调用，从而把正文的背景设置成透明：

```
#include〈wingdi.h〉
int WINAPI SetBkMode（HDC hdc，int fnBkMode）；
```

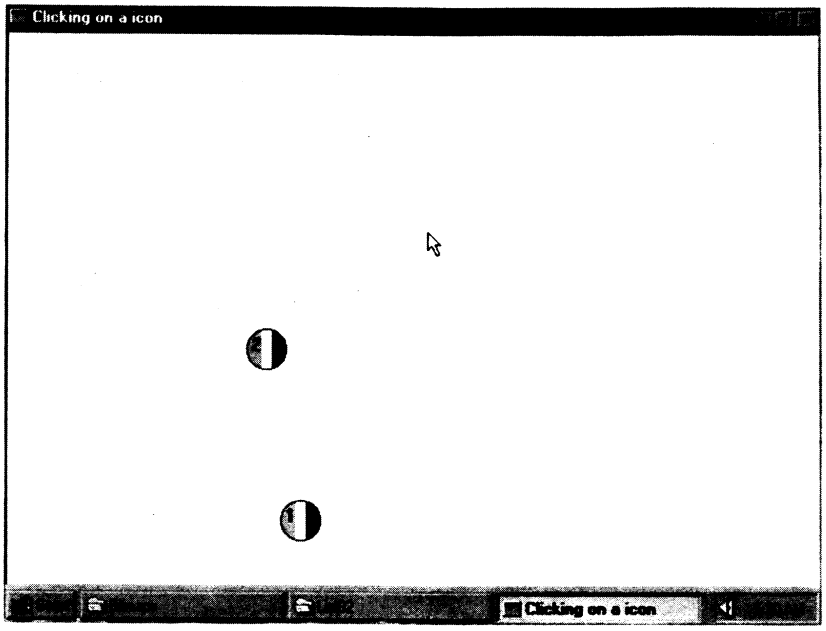


图 5-6　ICON 最初只显示了一个独立的图标。此时假如用鼠标右键单击它，整体客户区就会重画，然后显示出另一个图标

参数	**说明**
HDC hdc	设备现场的句柄
int fnBkMode	背景模式：OPAQUE（不透明）或者 TRANSPARENT（透明）
返回值	在正文里讨论
int	上一种背景模式；假如函数调用失败，则返回一个 NULL 值

假如选择 OPAQUE，我们就可以把当前的背景颜色或者以前由 SetBkColor () 分配的一种颜色当作设备现成的输出颜色使用。TRANSPARENT 则会保持背景颜色的未修改状态。这样一来，我们就可以在圆形绘图上书写图标 ID，同时保留背景颜色。

图标的显示位置是根据整体客户区的大小经过随机运算得出的，它们的位置会通过 WM_SIZE 消息的拦截不断地进行测量和更新，拦截下来的信息会从 lParam 里取出，然后存储到两个静态整数里。

WM_SIZE	0x0005
wParam	指出尺寸改变的类型
lParam	客户区的宽度（低字）和高度（高字）

这种信息是由图标尺寸集成的，它为我们提供了通过调用 API 函数 CreateEllipticRgn () 而生成一个椭圆区域的机会：

```
#include 〈wingdi. h〉
HRGN CreateEllipticRgn (int nLeftRect,
```

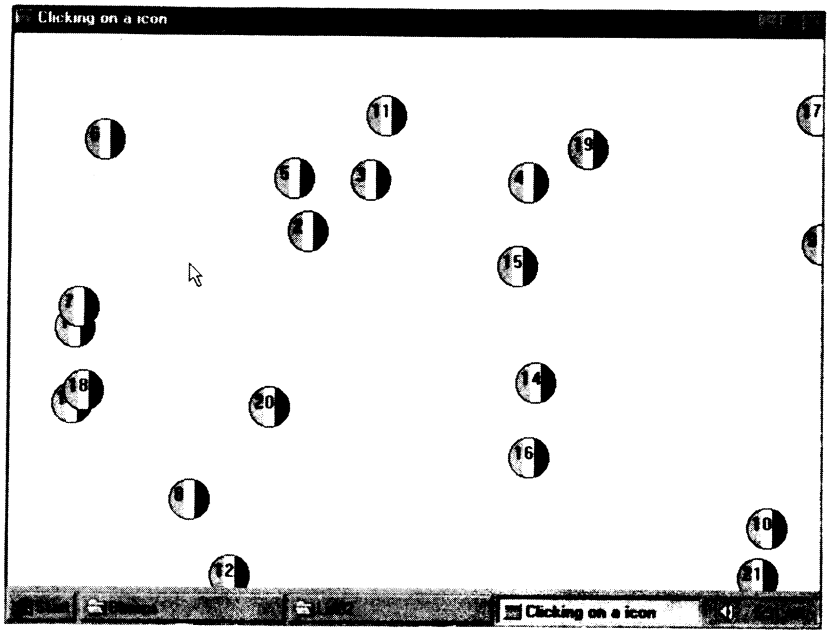


图 5-7　ICON 程序在屏幕上显示了多个图标

```
int nTopRect,
int nRightRect,
int nBottomRect);
```

参数	说明
int nLeftRect	约束矩形左上角的 X 坐标
int nTopRect	约束矩形左上角的 Y 坐标
int nRightRect	约束矩形右下角的 X 坐标
int nBottomRect	约束矩形右下角的 Y 坐标
返回值	在正文里讨论
HRGN	区域句柄；假如函数调用失败，则返回一个 NULL 值

CreateEllipticRgn ( ) 函数可以接收四个整数，这些整数与某个矩形区域的左上角和右下角位置是对应的。以这些信息为基础，它可以计算出一个圆形区域的位置和大小。在 Icon 这个例子里，该函数定义了围绕于每个图标的一个椭圆形区域，并且把计算出来的圆形区域存储到了一个区域句柄数组内。使用 DrawIcon ( ) 和 TextOut ( ) 这两个函数可以最终完成 WM_PAINT 的处理，把图标显示出来，并且公开它的数值标识编号。

在 Icon 这个例子里，每次单击一个现成的图标，应用程序的客户区域就会重画。正是由于存在对每个图标的位置进行定义的圆形区域，所以我们才有机会对屏幕显示进行限制，使其只显示出代表每个图标的圆形区域。

```
...
case WM_LBUTTONDOWN:
{
        POINT pt;
        short i;

        pt.x = MAKEPOINTS (lParam) .x;
        pt.y = MAKEPOINTS (lParam) .y;

        if (nCnt == MAXICONS)
                break;

        // did we click on an icon?
        for (i = 0; i <= nCnt; i++)
                // create the new icon
                if (PtInRegion (hrgn [i], pt.x, pt.y))
                {
                        ++nCnt;
                        InvalidateRect (hwnd, NULL, TRUE);
                        UpdateWindow (hwnd);
                        return 0L;
                }
}
        break;
...
```

我们还需要对图标图象上方发生的 WM_LBUTTONDOWN(单击鼠标左键)事件进行检查。为了实现这一要求，可以调用 PtInRgn () 函数。该函数能检查一个给定点（鼠标光标当前在屏幕内的位置）在一个给定区域（在 WM_PAINT 处理过程中计算出来的）的内部还是外部。假如检查出来表明鼠标光标正好位于圆形区域上方，整体客户区域就会屏蔽起来，并在屏幕上新增一个图标。

5.4 STRINGTABLE 资源

对于最终需要在应用程序执行过程中显示出来的所有正文串来说，资源文件是存放它们的最佳场所。当然，这并不是一条很严格的规则，但的确是我们强烈建议的。因为这便于设计软件的国际化版本，使应用程序从一种语言转换成另一种语言变得相对容易。STRINGTABLE 资源里最多可以包含 65536 个正文串，每个串都用一个数值 ID 进行标识，这个 ID 是不能重复的。标识 ID 可以由程序员设置，也可以由开发环境帮助开发者自动设置。由

于一个正文串不能超过 255 个字符的限制，所以它没有资格对联机帮助信息进行设置。在另外一方面，主窗口的类名、主窗口标题以及不同消息框内显示出来的消息都是正文串应用的典型例子，这些串一般都存储于 .RC 文件的 STRINGTABLE 资源里。

下面是 STRINGTABLE 的语法结构：

```
STRINGTABLE
 {
    stringID, " string"
}
```

实际的正文必须用双引号封闭起来，并在前面加上它的 ID。STRINGTABLE 信息块里包含实际数字的情况是很少见的。更准确地说，这些 ID 通常都是在一个特定头文件里定义的，.RC 文件只是利用了这些定义，这样就增强了整体的可读性。Visual C++会自动生成 RESOURCE.H 头文件，其中包含了与 .RC 文件定义的资源有关的所有 ID 信息。开发者可以自己动手把这种包容文件插入应用程序的源代码内，从而提供一种方式对这些资源进行访问。

现在假设我们已经定义了一个正文串，这个串与主窗口的类名有关。在 RESOURCH.H 里，对应的定义就应该像下面这个样子：

```
// RESOURCE.H
#define ST_CLASSNNAME 1000
...

// TWENY.RC
...
#include " resource.h"
...
STRINGTABLE
 {
    ST_CLASSNAME, " Main class"
    ...
}
...
```

包含了正文串 ID 的头文件在资源文件和源文件里都会出现。事实上，为了把一个串资源载入应用程序代码内，我们需要用到 LoadString ()。该函数需要把正文串 ID 当作对每个串的识别手段：

```
#include <winuser.h>
int WINAPI LoadString (HINSTANCE hinst,
                       UIINT idResource,
                       LPSTR lpszBuffer,
                       int cbBuffer);
```

参数	说明
HINSTANCE hinst	应用程序的实例句柄
UIINT idResource	资源 ID
LPSTR lpszBuffer	用于接收串的缓冲区
int cbBuffer	缓冲区尺寸
返回值	在正文里讨论
int	实际读取的字符数目（不包括最后一个\0）；假如调用失败，则返回一个 0

应用程序实例句柄指定了资源 ID 以后，它就会在 STRINGTALBE 资源里唯一性地标识出来。LoadString () 必须使用一个缓冲区地址和这个缓冲区的尺寸，否则便无法成功地调用。假如函数成功地调用，实际读取的字符数（最后的那个\0 除外）就会返回。在许多应用场合下，大家都需要从资源文件里载入主窗口的类名，就像下面这样：

```
...
char scClassName [20];
...
LoadString (hInsance, ST_CLASSNAME, szClassName, sizeof (szClassName));
...
```

在上面这个例子里，ST_CLASSNAME 代表资源 ID，szClassName 则代表接收字符串的缓冲区。根据我自己的经验，我一般都设置了一个 ST_ 前缀，用它指明某个 ID 是与 STRINGTABLE 资源有关的。当然，菜单项的 ID 都是以 MN_ 前缀起头的，对此我们在下一章还要详细地说明。

5.5 一次性载入多个串

有些情况下，我们有必要同时载入多个正文串，用它们来填充一个列表框或者列表视窗。即使在这种情况下，STRINGTABLE 资源仍然是存储所有正文串的最佳场所，它为正文串的载入提供了一种严格的顺序。这儿要注意的问题是为正文串分配连续性的数值 ID，并在最末尾的地方放置一个空串。此外，在某些情况下，最后那个串在 STRINGTABLE 资源里是没有用处的。假如我们对此有把握，也可以省略最后设置的那个空串。

```
...
#include " RESOURCE. H"
...
STRINGTABLE
{
    ST_STATE + 0, " Alabama"
    ST_STATE + 1, " Washington"
    ST_STATE + 2, " Florida"
```

```
        ST_STATE + 3, ""
        ...
    }
    ...
```

其中的 ST_STATE 是在 RESOURCE. H 里定义的：

```
    ...
    // RESOURCE. H
    ...
    #define ST_STATE 1000
    ...
```

Alabama 这个正文串内包含了值 1000，Washington 1001 和 Florida 1002。其中，ID 1003 指定的是一个空串。另外，我们也可以选择不为 STRINGTABLE 资源里的其他任何串分配 ID。在应用程序代码里，我们在一个 while 循环里设置了对 LoadString () 的一次调用，该函数的有效性是由返回值决定的：

```
    ...
    i = ST_STATE;
    ...
    while (LoadString (hInstance, i, szBuffer, sizeof (szBuffer))
    {
        // string is inserted in the listbox
        ...;

        // moving to the following ID
        i++;
    }
    ...
```

整个循环会不停地运转下去，直到碰到一个空的字符串为止（或者没有一个特定的 ID 没有正文串与其关联起来），这时就会返回一个零值，从而使函数的调用无效。

5.6 其他二进制资源

位图和图标具有差不多的属性，唯一的区别在于我们需要用 LoadBitmap () 去调用一幅位图。这个函数是针对从资源文件里调用位图而专门设计的。在 Win32 里，16 色的限制已经不复存在，这样就提高了资源在应用程序开发中的地位，资源的包容力也更强了。

```
#include〈wingdi.h〉
HBITMAP WINAPI LoadBitmap (HINSTANCE hInstance, LPCSTR lpszIcon);
```

参数	说明
HINSTANCE hInst	应用程序的实例句柄
LPCSTR lpszIcon	位图标签
返回值	在正文里讨论
HBITMAP	位图句柄；假如函数调用失败，则返回一个 NULL 值

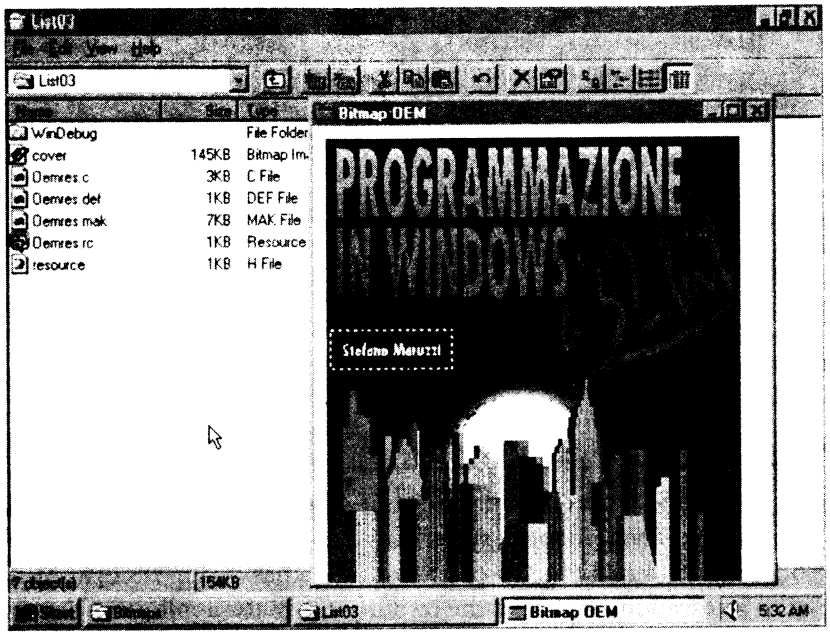


图 5-8　BMPDRAW 从资源文件里装载了一幅位图，并且把它画到了应用程序客户区内

假如函数调用成功，那么返回值就是一幅位图的句柄。然而，和图标不一样，Win32 API 没有提供 DrawBitmap () 函数，从而无法轻而易举地在一种设备现场内显示位图。正如我们在前面的章节里强调过的那样，描绘一幅位图之前需要先做一番准备工作，从而在当前的设备现场内选择位图，然后使用 BitBlt () 或者另外一种位块传输函数。为了详细阐述这种技术，我设计了一个名为 BMPDRAW 的例子，读者可以在本书附带 CD 的 Listing 5.3 里找到它。BMPDRAW 的运行情况如图 5-8 所示。

我们在 .RC 文件里可以看到，为 BITMAP 资源分配的封面标签是存储在 COVER.BMP 文件里的。

```
cover BITMAP DIISCARDABLE " cover.bmp"
```

这个对象是在 hbitmap 句柄的 WM_CREATE 里载入的，并且被声明成了静态类。这样

一来，整体窗口进程里都能对其进行引用。

```
...
hbitmap = LoadBitmap (hInstance, " cover");
...
```

在 WM_PAINT 代码块里，首先需要通过 BeginPaint () 取回一个设备现场句柄。以后，专门针对这一任务设计的 DrawBitmap () 函数就可以对这个句柄进行调用：

```
BOOL DrawBitmap (HDC hdc,
                 HBITMAP hbm,
                 int x,
                 int y)
{
    HBITMAP hbmpOld;
    BITMAP bm;
    HDC hdcMem;

    // compatibile DC
    hdcMem = CreateCompatibleDC (hdc);
    // get the bitmap dimensions
    GetObject (hbm, sizeof (BITMAP), (LPSTR) &bm);

    // save the old bitmap
    hbmpOld = SelectObject (hdcMem, hbm);

    // display the bitmap
    BitBlt(hdc, x, y, bm.bmWidth, bm.bmHeight, hdcMem, 0, 0, SRCCOPY);
    // restore the old bitmap in the DC
    SelectObject(hdc, hbmpOld);
    // delete the memory DC
    return DeleteDC(hdcMem);
}
```

对于位图可视化后面隐藏的逻辑来说，它要求开发者先把位图选择到一个内存 DC（设备现场）里，随后从那个 DC 里拷贝到真正的显示 DC 内。针对第一步操作，我们需要用到 CreateCompatibleDC ()，该函数的作用是对显示 DC 进行处理：

```
#include <wingdi.h>
HDC WINAPI CreateCompatibleDC (HDC hdc);
```

参数	说明
HDC hdc	设备现场的句柄
返回值	在正文里讨论
HDC	兼容的设备现场句柄

随后，我们还要利用 SelectObject （） 对内存 DC 相关设备现场里的位图进行选择：

```
...
// compatible DC
hdcMem = CreateCompatibleDC (hdc);
hbmpOld = SelectObject (hdcMem, hbm);
...
```

在进行实际的拷贝操作之前，我们还需要知道位图的整体尺寸。GetObject （） 是帮我们实现这一目标的理想工具。尺寸信息是存储在一个 BITMAP 数据结构里的：

```
...
GetObject (hbm, sizeof (BITMAP), (LPSTR) &bm);
...
```

BitBlt () 是我们需要利用的最后一种工具，它的作用是把位图拷贝到应用程序客户区内：

```
...
BitBlt (hdc, 10, 10, bm.bmWidth, bm.bmHeight, hdcMem, 0, 0, SRCCOPY);
```

最后需要进行的工作是把原始位图恢复到显示现场里，同时清除现在已经没有用处的内存 DC。

5.7 用户自定义资源

本章最后一个例子展示了如何通过一种新资源的定义对资源文件进行更深层次的利用——这种新资源是专门针对应用程序的特定需求而设计的。RAWDATA（这是示范程序的名字）利用了 RCDATA 命令来声明一种用户自定义的资源类型。RCDATA 的语法与二进制资源是别无二致的，只是用户自定义资源相当多的场合都是用于收集正文数据。

```
label RCDATA
 {
    data
}
```

其中，label 可以是任何一种字符序列或者一个数值。用户自定义数据是跟随于资源定义后面的。在这个例子里，我们利用了由 INCLUDE 程序示例（在本书附带 CD 的 Listing 2.2 里）返回的某些信息，从而建立一个 TREE 数据块。这种数据部分重复了系统头文件的树形结构。

除了存储实际的文件名以外，我们还可以根据每个包容文件在分级结构里的位置，从而对其进行定位。这样一来，根文件 WINDOWS.H 的前面就加上了一个 1；而 EXCEPT.H 则在前面加上 2，以此类推。

原始的信息块看起来就像下面这个样子：

```
TREE RCDATA
 {
    1 WINDOWS.H
    2 EXCEPT.H
    2 STDARG.H
    2 WINDEF.H
    3 WINNT.H
    ...
}
```

所有这些信息都必须自己动手插入 .RC 文件里，因为现在的这个 MS Visual C++版本还没有提供对这种特性的支持。经过资源编译器的一番处理后，原始的数据格式就变成了下面这个样子：

```
TREE RCDATA
BEGIN
    0x0001, 0x0000, 0x4957, 0x444e, 0x574f, 0x2e53, 0x0048, 0x0002, 0x0000,
    0x5845, 0x5043, 0x2e54, 0x0048, 0x0002, 0x0000, 0x5453, 0x4144, 0x4752,
    ...
    0x6376, 0x682e, 0x0200, 0x0000, 0x6d00, 0x7863, 0x682e, " \000
END
```

根据 Intel 的标准规范，资源编译器已经把具有分级格式的所有信息转换成了对应的数据。为了把这些十六进制的数字转换成对应的 ASCII 字母，我们必须先取出每个块的最后两位（例如在 0x4957 里，应该先取出 57，亦即字母 W），紧接着取出前两位（49，对应于字母 I）。

原始数据块将通过 WM_CREATE 消息载入应用程序内存内，如下所示：

```
...
case WM_CREATE
{
    HRSRC hrsrc;
    HANDLE hres;
    char *pmem;

    // accessing the resource data
    hrsrc = FindResource (hInstance, " TREE", RT_RCDATA);
```

```
    // lock the data area
    hres = LoadResource (hInstance, hrsrc);
    // transform the handle into a pointer
    pmem = LockResource (hres);
...
```

在这以后，我们就可以对内存块进行处理了。方法是取出其中的 pmem 指针，然后把每个串都放置于一个 TREEVIEW（树形视窗）窗口内，这个窗口覆盖了整个客户区域。最后的结果如图 5-9 所示。

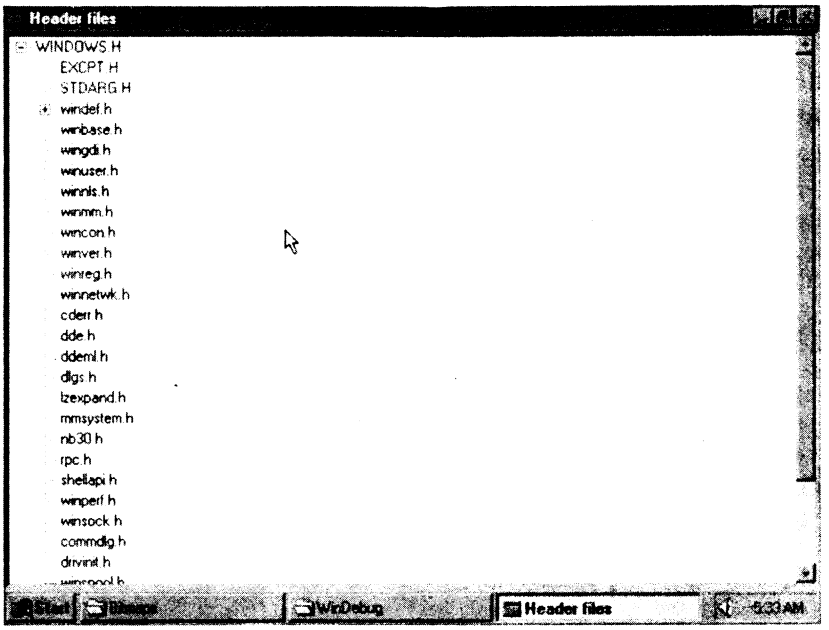


图 5-9　Rawdata 和以前看到过的 Include 例子有些形似，
其中列出了 WINDOWS.H 内包含的部分头文件

注意：在 Win32 里，没有必要对通过 .RC 文件载入的这样一种资源进行解锁和释放处理。

你的计算机系统安装了一块声卡了吗？你在启动 Rawdata 程序后听到了一段优美的旋律吗？对了，这段音乐与 Windows 95 启动成功后发生的声音是完全一样的。由于我们把对应的 .WAV 文件存储到了资源文件里，所以当应用程序主窗口在屏幕上显示出来的时候，就能载入和倾听到这段声音。最让人兴奋的是，在一个 Win32 程序里，要实现这种声音功能并不需要进行费力的设置工作。.WAV（波形文件）可以当作一种资源进行处理，并且可以通过非常简单和直接的方式封装到 .EXE 程序内。首先，我们先要在 .RC文件里声明一个波形声音资源：

```
sound WAVE sound.wav
```

在本书付印的时候，Visual C++还没有提供对长文件名资源的支持，所以记住按照传统的 8.3 格式定义所有文件的文件名。为了播放这个文件，我们还需要调用 sndPlaySound ()，这是一种功能很强的函数。

```
#inclue <mmsystem.h>
BOOL sndPlaySound (LPCTSTR lpszSoundName, UINT fuOptions);
```

参数	说明
LPCTSTR lpszSoundName	指向 .WAV 文件数据的指针，这些数据已载入内存块了
UINT fuOptions	播放选项
返回值	在正文里讨论
BOOL	布尔值，假如函数调用成功，就返回 TRUE；假如失败，则返回 FALSE

在播放一个波形声音资源之前，必须用 FindResource ()，LoadResource () 以及 LockResource () 先把这个资源载入内存。被指定为第二个参数的 SND_标志定义了如何播放 .WAV 文件。在 Rawdata 这个例子里，我们选用 SND_MEMORY 标志来说明 .WAV 文件已被载入由第一个参数指定的那个内存位置处。除此以外，SND_ASYNC 还要指示 sndPlaySound () 这个 API 异步播放出实际的曲调；假如由于某种原因导致函数播放目标文件失败，还要用 SND_NODEFAULT 标志来防止程序播放一种缺省的声音。下面列出的 PlayResource () 函数把涉及声音播放的所有语句都封装到了一处：

```
BOOL PlayResource (HINSTANCE hInstance, char *pszSound)
{
    LPSTR pRes;
    HANDLE hRResInfo, hRes;

    //find the resource in the resource file
    hResInfo = FindResource (hInstance, pszSound, " WAVE");
    if (hResInfo == NULL)
        return FALSE;

    //loading the wave resource
    hRes = LoadResource (hInstance, hResInfo);
    if (hRes ==NULL)
        return FALSE;

    //accessing the resource in memory
    pRes = LockResource (hRes));
    if (pRes ! =NULL)
```

```
        {
                return sndPlaySound (pRes, SND_ MEMORY | SND_ ASYNC | SND_
NODEFAULT);
        }
    }
```

PlaySound () 这个 Win32 API 是播放 .WAV 文件的一种替代方法，它可以直接从资源文件或者注册表数据库内载入准备播放的 .WAV 声音。

第 6 章　菜单的运用

我并不认为一个正常的 Win32 应用程序会不使用菜单。尽管已经从面向应用程序的用户界面过渡成了一个面向对象的用户界面，微软公司仍然决定保留菜单在外壳中的关键地位，这个决定是正确的。从历史上看，菜单代表了用户和应用程序之间的首选通信方式。人们可以在一个下拉式菜单内选择某个菜单项，从而执行相应的任务。在另一方面，Win95 外壳在这种传统模式的基础上还进行了一番扩展。在 Win95 里，在屏幕上任何地方单击鼠标右键都会弹出一个关联菜单，微软公司对这方面的进步进行了显著的强调。在这儿，“菜单”这个术语并不只是指一种特定类型的对象，更准确地说，它代表了一系列不同选项的组合。

现在让我们观察一下第一章“Win32 中的软件开发”的 Except 例子。Win32 程序支持的菜单包括下面这四种：一个“系统菜单”（以前叫作“控制菜单”），它位于窗口的左上角；正好位于应用程序标题栏下方有一个菜单栏，其中包含了数个顶级菜单；几个下拉菜单，这些下拉菜单与菜单栏是连接到一起的；以及一个或者多个关联菜单。如图 6-1 所示。

正如大家通过阅读第三章“Win32 应用程序的开发”知道的那样，之所以存在系统菜单，是由于建立一个窗口的时候设置了 WS_SYSMENU 标志。这种菜单的外观和内容对于每个窗口来说都是差不多的，其中一般都包含了六个菜单项：Restore（恢复）、Move（移动）、Size（尺寸）、Minimize（最小化）、Maximize（最大化）和 Close（关闭）。这些菜单项的一部分可能永远都是以灰色显示的（已被屏蔽）——表明缺少一种特定的窗口风格或者窗口当时正处于某种状态下（假如当时正以最大化显示，那么对应的 Maximize 菜单项就会变成灰色）。

通过包括附加的项目，我们也可以对系统菜单的内容进行扩展，然而这种处理是我们特别不赞成的。之所以不赞成，并不是说这与某些技术限制或者 API 的约束有关，更准确地说，这只是一种风格准则。正如我们在这一章里看到的那样，菜单 API 的数量是相当多的，并且已经经历了各种改变和进化。对于开发者来说，应该时刻参照菜单风格方面的准则和建议，使菜单具有通用的格式和外观，从而让用户易于接受、易于上手。

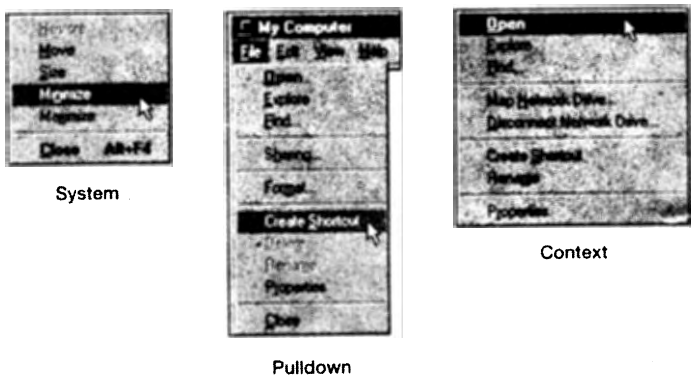


图 6-1　一个系统菜单、正在显示下拉式菜单的一个标题栏菜单以及一个关联菜单

标题栏内包含了几个简短的单词，它们在技术上的定义就是“顶级菜单”。选择其中一个顶级菜单，就会显示出一个下拉式菜单，这几乎已经成了一种真理。从技术的角度来看，尽管没有什么成文的规定要求我们必须为每个顶级菜单都设计一个下拉式菜单，但在设计的风格准则里，这已经是一条不成文的规定，任何开发者都应该遵守。

关联菜单在应用程序客户区的任何地方都有可能显示，只需单击一下鼠标右键就可以了。这是一种普通的下拉式菜单，只是上面没有任何顶级菜单在支持着它。关联菜单通常是和客户区内一种特定的对象关联在一起的。在 Except 这个例子里（参考第一章），针对用户在一个空白处的鼠标右键单击操作，我们为整个应用程序定义了一个关联菜单（亦被称为“弹出式菜单”）；第二个弹出式菜单则特别设计用于提供关于选定页的信息。

对于一个标准菜单（系统菜单、标题栏菜单和对应的下拉式菜单）来说，用户只需要在其上面简单地单击鼠标左键就可以了，Win32 环境随后就可以自动完成相应的处理。但是，对于弹出式菜单（关联菜单）来说，情况则有所变化——必须在应用程序代码内部对其进行控制。在典型情况下，一条 WM_CONTEXTMENU 到达窗口进程的时候，必须计算出鼠标指针当时的坐标是否与客户区内一个可变的位置对应。然后再从资源文件里载入一个弹出式菜单，最后才能把它显示于屏幕上。

6.1 菜单项的选用

菜单栏的主要用途是提供一种简便的方式，从而把各自独立的下拉式菜单内的不同命令组合到一起。一般来说，假如在菜单栏上单击一个与下拉式菜单没有任何联系的地方，应用程序是没有什么反应的。要想显示出下拉式菜单，必须在一个顶级菜单项上单击鼠标左键，或者通过键盘进行人工选择。通常，假如按下 Alt 键，第一个顶级菜单（最左边的那个）就会以反转色显示（假如采用的是 Windows 缺省颜色方案，就应该是白色正文、蓝色背景）。光标键允许用户移至另外一个顶级菜单，并且可以在不同的菜单之间循环移动。这种循环也会把系统菜单包括进去。假如某个应用程序是按照 MDI 规范开发出来的，那么活动的文档控制菜单也会包括进去。如果想显示对应的下拉式菜单，只需简单地按“下”方向箭头即可；如果想让它消失，按 Esc 键即可。

除此以外，每个菜单都提供了带有下划线字母的选项。从技术角度来看，这就是我们常说的“记忆键”。这些字母在同一级别的所有菜单项中都是唯一的，不能发生重复现象。例如，对于 File 和 Edit 这两个最常见的顶级菜单来说，它们的记忆键分别就是单词的第一个字母：F 和 E。此时假如同时按下 Alt 和 E 键，就会立即显示出 Edit 下拉式菜单，如图 6-2 所示。

通过图 6-2，我们可以看到每个下拉式菜单项都有一个相互之间不能重复的记忆键。在这种情况下，只需按下对应的字母键（无论大写还是小写），就可以选中那个菜单项。

关联菜单是无法通过键盘操作激活的，它只能通过鼠标或者一种加速键组合来实现。在这儿，加速键（后面要进行详细介绍）是两个或者多个按键的一种键盘组合，它们可以模拟由鼠标选择生成的某种事件。如果想显示出外壳桌面上某个常规对象的关联菜单，可以把鼠标指针定位到该对象上方，然后单击鼠标右键即可。对于加速键来说，则可以按下 Shift+F10，从而得到相同的结果。

假如不再作进一步的选择，下拉式菜单会一直保持打开状态，并在其中列出了许多菜单项——其中，当前选定的那一项是用反转色（高亮度）显示的。假如在下拉式菜单中上下移

动鼠标指针，选定的菜单项就会相应地变化；这和在 Windows 3.x 里看到的情况是不一样的。为了执行一个命令——也就是说，选中一个菜单项，然后起动和它相关的操作，只需按下鼠标左键即可。从这以后，下拉式菜单就消失了，应用程序接着执行与那个菜单项有关的代码段。这种行为无论对标准的下拉式菜单还是弹出式菜单都是适用的。

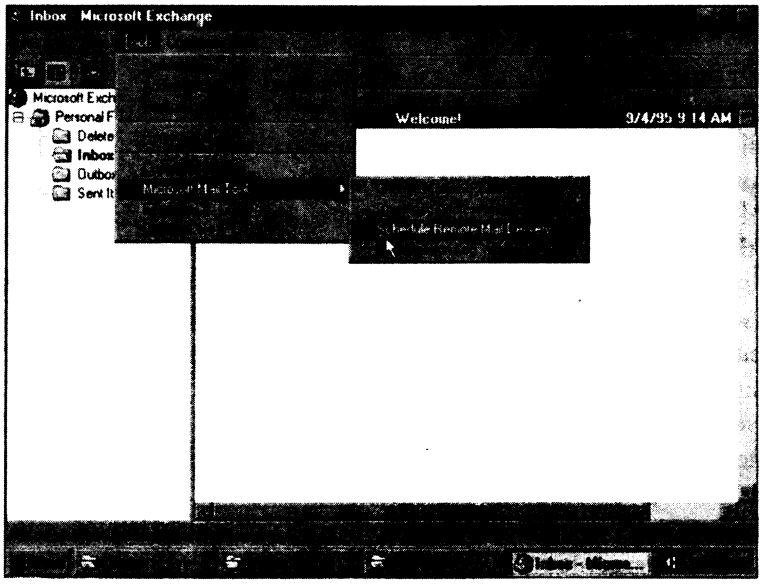


图 6-2　每个菜单项都提供了一个下划线字母，即我们常说的记忆键，利用它可以加快键盘选择

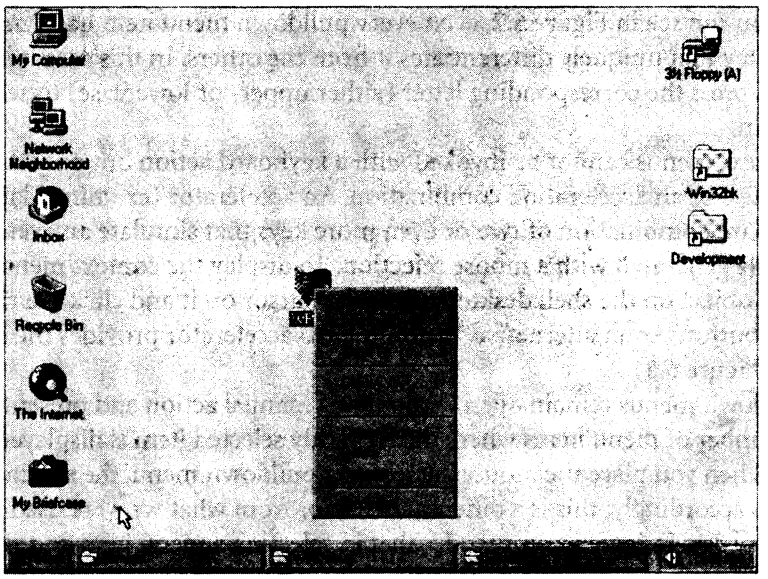


图 6-3　系统桌面内这个对象的关联菜单可以通过按下 Shift+F10 加速键组合来激活

经过前面这段简短的介绍以后，现在让我们把注意力集中在菜单的开发上面。这些菜单通常要被定义成.RC 文件里的某种资源——关于.RC 文件的详情，大家可以参阅第 5 章“资源文件”的内容。当然，资源文件并不是生成菜单的唯一途径，但这确实是最常用，也是最方便的一种办法。所以，我们接下来要观察菜单模板的整体布局，并且学习如何定义不同的组件。

6.2 观察菜单模板

菜单模板的编写是一种相当简单和直观的操作，它涉及到由资源编译器支持的两条命令：POPUP 和 MENUITEM。其中，POPUP 命令用于指定一个顶级菜单或者下拉式菜单内的一个菜单项，这个菜单项可以再引出第二个（或者说更低级的）下拉式菜单。下拉式菜单内的每个菜单项都必须用 MENUITEM 命令进行声明。这几乎就是我们的全部工作。所有这些信息都必须封装到一个范围更大的代码块内，并且要用 MENU 命令来定义这个块。在这儿，MENU 使我们有机会把一个 ID 分配给对应的菜单模板。这样一来，.RC 文件内一次就可以容纳多个菜单模板，每个模板都用自己的 ID 进行标识，这种 ID 是不能重复的。

```
labelID MENU, [optional statements]
BEGIN
    // leftmost top level menu
    POPUP " top-level string", options
    BEGIN
        MENUITEM " menu item text", menuitemID, options
        MENUITEM " menu item text", menuitemID, options
        MENUITEM " menu item text", menuitemID, options

        // the following directive introduces a
        // secondary level pulldown menu
        POPUP " item text", options
        BEGIN
            MENUITEM " menu item text", menuitemID, options
            MENUITEM " menu item text", menuitemID, options
            MENUITEM " menu item text", menuitemID, options
             ⋮
        END

        // more item in the first pulldown menu
         ⋮
    END

    // second top-level menu
```

```
        POPUP " top-level string", options
        BEGIN
            MENUITEM " menu item text", menuitemID, options
            MENUITEM " menu item text", menuitemID, options
            MENUITEM " menu item text", menuitemID, options
            ⋮
        END

        // third top level menu technically possible, though
        // strongly discouraged from a stylistic perspective
        MENUITEM " menu item text", menuitemID, options
    END
```

每个 MENU 资源都要求使用一个标识标签。这与我们在第五章“资源文件”里对其他资源的处理方式并没有什么两样。虽然 Visual C++开发环境鼓励开发者为 .RC 文件内的每个资源都分配一个对应的数字，但就目前来说这几乎是不可能的，因为 Windows API 最初是设计用于对正文串进行处理，而不是数字。正如我们在第五章里看到的那样，诸如 LoadIcon () 和 LoadBitmap () 的函数都要求把指向字符的一个指针当作第二个参数使用。假如资源已经用数字标识了，就应该用 MAKEINTRESOURCE 对其进行转换。

前面展示的普通菜单模板提供了三个顶级菜单，其中两个都带有对应的下拉菜单。第三个菜单（模板最底部的那个）从技术上证明了可以把一个菜单项直接放置到菜单栏内——换言之，菜单不用下拉也可以。假如用户选中了这样的一个顶级菜单，它就会向应用程序自动发送一条命令。然而，菜单模板的这种设计方式是我们极力反对的，因为它不符合设计风格要求。

另外要注意的一个问题是：与第一个顶级菜单对应的下拉式菜单在几个 MENUITEM 之间提供了一个 POPUP 命令。在一个 POPUP 里存在另一个 POPUP 是可行的，系统支持这种设计——它的结果就是生成了一个次级别的下拉式菜单。从理论上说，并没有什么条文限制开发者不能把一个下拉式菜单放置到另外一个不同的下拉式菜单里。然而，尽管这样可以获得一种新颖的层叠式效果，但是也必须留意由这种设计带来的副作用。次级下拉式菜单会在屏幕上占据大量空间。更重要的一点在于，假如用户是用指点设备进行选择，必须十分小心地操作这种菜单（请注意：丢失目标的情况并不少见，特别是屏幕处于高分辨率状态时，所有的对象都要比平常小）。总而言之，我们应该为自己拟定这样两条规则：千万不要在菜单栏内放置一个 MENUITEM（菜单项）；同时把层叠式菜单限制在两层以内。

大家或许都见到过这样的景观：Start 菜单有时会显示出连续四个级别的菜单，就像图 6-4 显示的那样。我估计每选择一个下拉式菜单都要花至少一秒钟的时间。因此，对于一名有经验的用户来说，选中图 6-4 那个 System Monitor 工具程序至少要花去他（她）四秒或者五秒的时间——假设这名用户在整个选择过程中没有按过鼠标左键。

BEGIN-END 命令可以对构成一个 MENU 或者一个下拉式菜单的元素集进行限制。通常，在标准的 C 代码块内，这些定义都是用花括号封闭起来的。

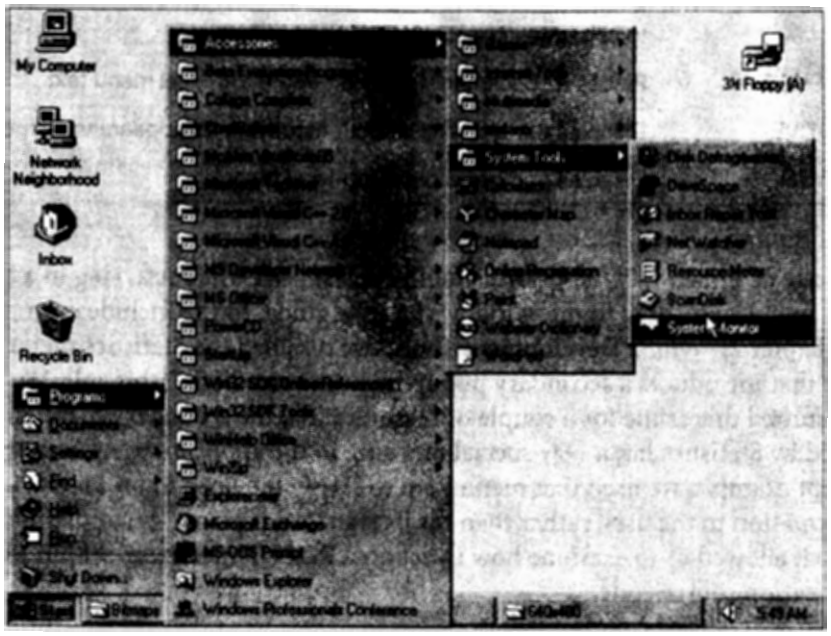


图 6-4　通过 Start 菜单选择 System Monitor 实用程序并非一件简单的事情

POPUP 命令可以生成一个顶级菜单或者一个菜单项——假如选中它们，就会显示出一个次级菜单。这些对象没有自己的 ID——这就限制了开发出高质量的程序代码。微软已经注意到了这个问题，并且定义了一种名为 MENUEX 的新菜单资源，我们稍后再对此进行详述。

从设计风格的角度来看，POPUP 命令支持开发者通过某些选项来定义菜单显示时正文串的外观。表 6-1 为大家列出了由 POPUP 支持的所有选项。

表 6-1　POPUP 的选项

选　项	说　明
MENUBREAK	顶级菜单放置于一个不同的栏内，从而获得了一种多栏效果
MENUBARBREAK	与 MENUBREAK 的表现类似，只是用一个垂直线条来分隔连续的顶级菜单
CHECKED	在顶级菜单正文的左边放置一个核选符号
INACTIVE	在不影响显示外观的前提下，对顶级菜单进行屏蔽
GRAYED	屏蔽并且用灰色显示顶级菜单

根据 Windows 95 的设计风格准则，开发者设计一个顶级菜单的时候，不应该使用任何一种 POPUP 选项。在第 2 章“Windows 开发工具”提供的 Include 示范程序里，我们屏蔽了一个 Statistics 菜单项，这个菜单项可以生成一个次级弹出式菜单，这种做法是与设计准则相抵触的。我们之所以会犯这样的“错误”是出于两方面原因的考虑。首先，由 Statistics 生成的次级菜单在应用程序里具有非常特殊的含义。和 Except 类似，我们用那个菜单项来满足向用户提供某些信息的需要，而不是让其具有自己的标准用途。其次，这种方法允许我们观察如何通过应用程序源代码内部对 POPUP 项进行处理。

MENUITEM 的语法要求用到一个正文串，把它当作菜单项的名称。紧接在这个串后面的是一个标识项。正文串可以是字符的任意组合，并且通常都包含了 & 符号，用这个符号对那个菜单项进行快速记忆（记忆键）。在下面这个例子里，字母 O 就起着记忆键的作用：

MENUITEM " &Open…", …

ID 是一个数字，它标识了位于 MENU 资源内的那个菜单项，这个 ID 是不能重复的。正如大家稍后就要看到的那样，应用程序通过接收 WM_COMMAND 消息，然后提取出 wParam 低字内存放的菜单项 ID，从而知道用户选中的是哪个菜单项：

MENUITEM " &Open…", MN_OPEN

在 RESOURCH.H 里定义这个 ID 是我们一般的做法。同时，就像我们刚才做的那样，把定义放置于 MENUITEM 语句的右侧。就我个人来说，一般都要用到 MN_前缀，用它标识与某个菜单项有关的定义。

6.2.1 菜单项定义

这儿需要再次强调的一点是，只有 MENUITEM 命令才允许我们为这些对象分配一个 ID。POPUP 项里恰好缺少这种信息，这个问题在 Windows 的早期历史中对编程并没有太大的影响。这样一来，RESOURCH.H 里的一部分就需要专门用于对 MENUITEM 命令相关的所有 ID 进行定义，就像下面这样：

```
…
// RESOURCE.H
#define MN_NEW          100
#define MN_OPEN         101
#define MN_SAVE         102
#define MN_SAVEEAS      103
…
```

其中那些数字是可以任意分配的，它们唯一的用途就是避免资源编译器在编译的时候碰到重复现象。这种方式并不特别符合我们的逻辑思路。事实上，从应用程序的角度来看，它最关心的事情就是自己接收到的 ID。用户选择一个菜单项，这个菜单项对应的 ID 就应该让应用程序知道。假如两个菜单项带有相同的数字，应用程序就会针对两条独立的命令实施相同的行为——我个人认为这种事情有点离奇。Visual C++对这些 ID 重新进行了处理，使它们都从 40000 开始，每增加一个菜单项，数字就增 1。

6.2.2 MENUITEM 选项

MENUITEM 的语法结构是由从表 6-2 里选出的一个或者多个选项完成的。

在这五个选项中，只有 CHECKED 和 GRAYED 才符合 Windows 95 的风格设计规范，但是它们两个不应该在一块儿使用。举个例子来说，当菜单载入应用程序的时候，假如必须对某个菜单项进行屏蔽，实现这一目标的最简便方式就是在 MENUITEM 语句的左边放置一个 GRAYED 选项，就像下面这样：

MENUITEM " &Save", MN_SAVE, GRAYED

应用程序启动以后，Save 选项很有可能是处于屏蔽状态的，因为这时一般没有什么东西

可以存储。

表 6-2 MENUITEM 命令使用的选项

选 项	说 明
MENUBREAK	菜单项被移动到一个新栏，这样就生成了一种多栏布局
CHECKED	在菜单正文左边显示一个核选符号
INACTIVE	菜单项处于屏蔽状态，但它的显示外观并未改变
GRAYED	屏蔽一个菜单项，并用灰色把它显示出来
HELP	在 Windows 95 里无效

6.2.3 一个典型的 MENU 资源

现在让我们来进行测试。下面这个菜单模板代码段里只包含了前两级菜单：File 和 Edit，每个都带有自己对应的几个菜单项。请注意 MENUITEM SEPARATOR 语句的使用，它的作用是在两个连续的菜单之间描绘一条水平分隔线。事实上，分隔线是一个不能选择的项目，它的运用是相当风格化的。对于开发者来说，最好把某些在逻辑上相关的菜单项集合到一个下拉式菜单块内，并在它和接在后面的其他菜单项之间描绘一条分隔线。New 和 Open 菜单项能够让用户建立一个新文档，或者打开一个现成文档。选择这两个选项后产生的效果是有区别的，尽管它们相当形似。相同的考虑亦可应用于 Save 和 Save As——File 下拉式菜单内的另一个逻辑组。

```
mainmenu MENU
{
    POPUP " &File"
    {
        MENUITEM " &New",                    MN_NEW
        MENUITEM " &Open…\tCtrl+F12",        MN_OPEN
        MENUITEM SEPARATOR
        MENUITEM " &Save\tShift+F12",        MN_SAVE
        MENUITEM " Save &As…\tF12",          MN_SAVEAS
        MENUITEM SEPARATOR
        MENUITEM " Page Se&tup…",            MN_PAGESETUP
        MENUITEM " &Print…\tCtrl++F12",      MN_PRINT
        MENUITEM " P&rint Setup…",           MN_PRINTSETUP
        MENUITEM SEPARATOR
        MENUITEM " E&xit\tAlt+F4",           MN_EXIT
    }

    POPUP " Edit"
    {
```

```
        MENUITEM " &Undo\tCtrl+Z",          MN_UNDO
        MENUITEM " Cu&t\tCtrl+X",           MN_CUT
        MENUITEM " &Copy\tCtrl+C",          MN_COPY
        MENUITEM " &Paste\tCtrl+V",         MN_PASTE, GRAYED
        MENUITEM SEPARATOR
        MENUITEM " Select &All\tCtrl+A",    MN_SELECTALL
        }
        ...
}
```

这个模板是用手工编写出来的，但它与 Visual C++编译器产生的结果从本质上来看却没有什么不同。某些菜单项字符串内的\t 字符可以强迫正文沿着普通下拉式菜单内第三栏的右侧起始处开始排列，如图 6-5 所示。

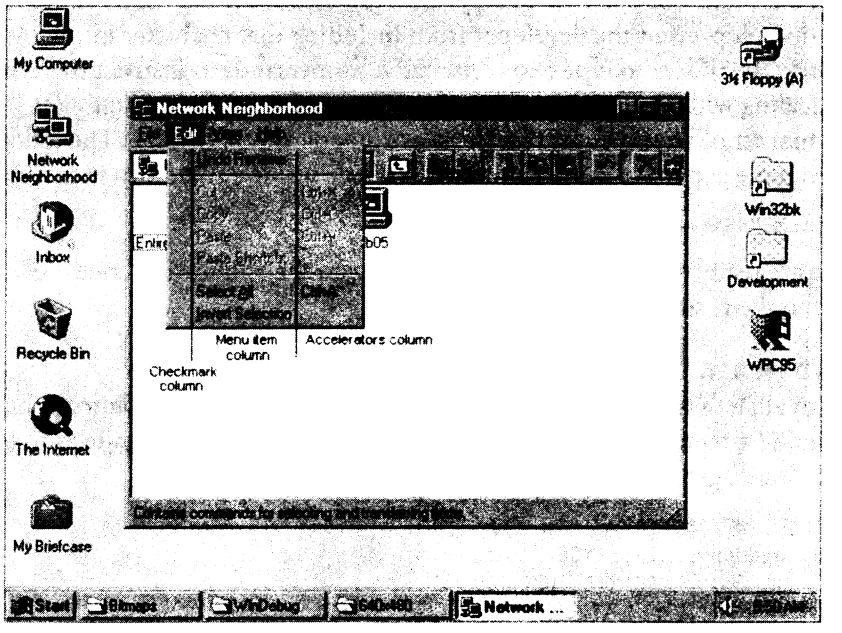


图 6-5　典型的下拉式菜单结构提供了范围可变的三个选项区域

菜单基本上也可以称为窗口，但是 Win32 API 并不允许我们把这些对象当作真正的窗口来控制。下拉式菜单的宽度是由菜单项正文的容量来决定的，通常以其中最长的一个正文串为基础进行衡量。正文串放置于最中间的那一栏内，它的左侧留出了一个窄长的栏，用于容纳可能有的核选符号。最开始这一个窄长的栏肯定是存在的——无论其中是否实际显示出了核选符号。第三栏——也是最后一栏用于容纳加速键序列，这些加速键与不同的菜单项是各自对应起来的。

正如我们已经知道的那样，加速键是两个或者三个（这种情况很少见）按键的组合。按下加速键组合就相当于选中了对应的菜单项，这样可以加快熟练用户的操作。在我们前面列

出的菜单模板代码块内，大家可以看到 File 和 Edit 下拉式菜单内都包含了一些加速键序列。由于在其中使用了\t 字符，所以加速键的显示是与实际的菜单项正文分开的。这样一来，考虑到下拉式菜单的宽度是以模板内指定正文的长度为基础决定的，所以我们强烈建议大家选择一些简短并且含义准确的单词，从而限制菜单在屏幕上占据的空间。

除此以外，对于设置记忆键的 & 符号来说，并没有什么规定要求开发者不能在菜单项正文里使用这种符号。我们可以键入两个连续的 & 符号，从而指示 Win32 API 把菜单模板的载入和转换成实际的窗口集，从而把它们当作标准的 ASCII 值进行处理。下面这个例子将在一个下拉式菜单里生成正文：Bitmap & Image：

```
MENUITEM " &Bitmap && Image"
```

到此为止，我们已经作好了把这种新资源与应用程序联系起来的准备工作，所以接下来让我们转向 C 源文件的讨论。

6.2.4 载入菜单模板

首先，让我们先来研究一下 LoadMenu () 这个 API 函数的语法结构。该函数的用途是对菜单模板的载入进行控制，它能把菜单原本转换成一系列对象的集合，这些对象可以通过返回值（一个句柄）进行管理：

```
#include <winuser.h>
HMENU LoadMenu (HINSTANCE hinst,
                LPCTSTR lpMenuName);
```

参数	说明
HINSTANCE hinst	应用程序的实例句柄
LPCTSTR lpMenuName	菜单模板的标识标签
返回值	在正文里讨论
HMENU	菜单句柄；假如函数调用失败，则返回一个 NULL 值

LoadMenu () 的语法要求获得应用程序的实例句柄，通过它可以知道用哪个模块来获取菜单模板。紧跟在这个参数后面的是资源标签，这种标签可以是一个正文串，也可以是转换成串的一个数字——这种转换工作是由 MAKEINTRESOURCE 完成的。LoadMenu 要求使用一个模块句柄，后面跟上一个资源名称。第一个参数用于指定资源的来源，这种来源通常是应用程序的实例句柄。资源标签既也可以是一个正文串，也可以是通过 MAKEINTRESOURCE 宏转换成串的一个数字。在本书提供的例子里，MAINMENU 标签是分配给应用程序主菜单的：

```
...
HMENU hmenu;
...
HMENU = LoadMenu (hInstance, " mainmenu");
...
```

其中，通过调用 LoadMenu () 返回的 hmenu 句柄是指一个菜单栏，这个菜单栏里包含了所有顶级菜单以及各自对应的下拉式菜单。这种对象只能与某个窗口联系到一起，否则便无法

成为 Win32 进程里可见并且有用的一部分。为了达到这一目标，API 至少为我们提供了两种方案：窗口类注册和窗口建立。WNDCLASSEX 数据结构的某个项是经过特殊设计的，它专门用于接收一个菜单模板标签，并且强迫属于那一类的所有窗口都自动支持那个菜单：

```
...
wcex.lpszMenuName = " mainmenu";
...
```

这种方案看起来限制了开发者能够做出的选择，因为它强迫属于那一类的所有窗口都采纳完全一致的菜单布局。在另外一方面，在我们注册某窗口类的时候，很多时候已经事先知道应用程序只会生成属于那一类的一个窗口。所以，刚才讲述的方案就不存在太严重的强制性了。然而，在通常情况下，正如我们在第 3 章“Win32 应用程序的开发”里讲述的那样，可以在一个 WNDCLASSEX 结构的 lpszMenuName 项里设置一个 NULL，这样就可以避开这种问题。如下所示：

```
...
wcex.lpszMenuName = NULL;
...
```

通常地，当我们用 CreateWindowEx () 函数建立一个窗口的时候，通常把一个菜单和窗口关联起来。CreateWindowEx () 的第 10 个参数就是用于指定菜单句柄的，这个句柄正好就是由 LoadMenu () 返回的那个。下面是对 CreateWindowEx () 的一次调用示例，其中就关联一个菜单模板：

```
...
hwnd = CreateWindowEx (WS_EX_CLIENTEDGE | WS_EX_WINDOWEDGE,
                       szClassName,
                       szWindowTitle,
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT, 0,
                       CW_USEDEFAULT, 0,
                       NULL,
                       LoadMenu (hInstance, " mainmenu"),
                       hInstance,
                       NULL);
...
ShowWindow (hwnd, SW_SHOWNORMAL);
...
```

窗口在屏幕中显示的时候，它会在自己的标题栏下方显示出一个菜单栏。假如我们想获得与某个特定窗口联系在一起的菜单的句柄，那么 GetMenu () 就是理所当然的选择。这个函数的参数只有一个，那就是目标窗口的句柄。

```
#include <winuser.h>
```

HMENU WINAPI GetMenu（HWND hwnd）；

参数	说明
HWND hwnd	一个窗口的句柄，这个窗口既可以是叠置式窗口，也可以是弹出式窗口
返回值	在正文里讨论
HMENU	与那个窗口联系在一起的菜单的句柄；假如函数调用失败，则返回一个 NULL 值

类似地，建立一个窗口时，并没有成文的规定要求我们一定要为它联系一个菜单。通过调用 SetMenu（）这个 API 函数，上面提到的这两种操作均可用单独的步骤来完成：

#include〈winuser. h〉

HMENU WINAPI SetMenu（HWND hwnd，HMENU hmenu）；

参数	说明
HWND hwnd	一个窗口的句柄，这个窗口既可以是叠置式窗口，也可以是弹出式窗口
HMENU hmenu	一个菜单的句柄
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败则返回 FALSE

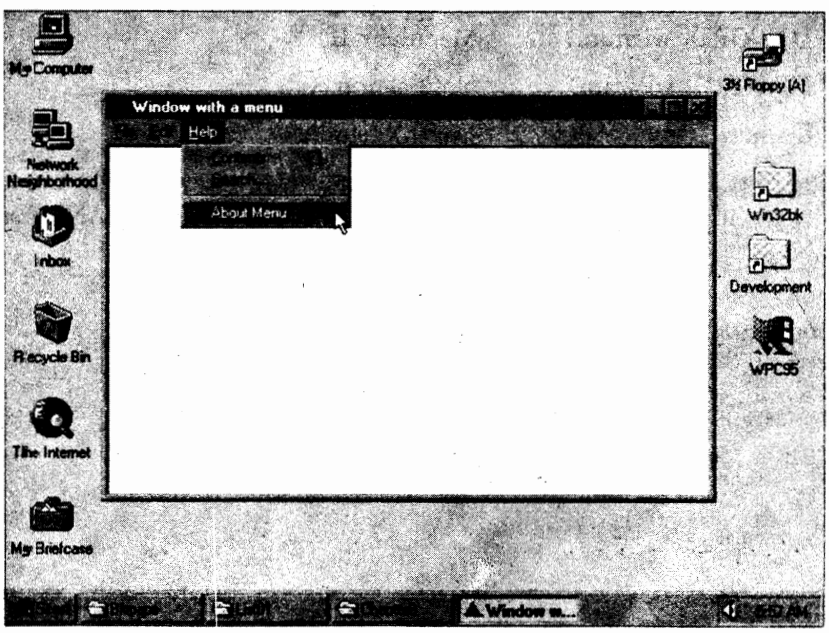


图 6-6　Menu 例子中显示的菜单栏是通过拦截 WM_CREATE 消息与主窗口联系起来的

图 6-6 显示的是 Menu 程序示例运行的情况，这是一个非常简单的 Win32 应用程序，其中带有一个菜单栏（Menu 例子可以在本书附带 CD 的 Listing 6.1 里找到）。在 Menu 里，拦截 WM_CREATE 消息以后，我们把由 MAINMENU 标签引用的菜单模板与应用程序窗口进程联系了起来。

```
...
WM_CREATE
{
    HMENU hmenu;

    hInstance = ( (LPCREATESTRUCT) lParam) -> hInstance;

    hmenu = LoadMenu (hInstance, " mainmenu");
    SetMenu (hwnd, hmenu);
}
break;
...
```

无论菜单模板是如何关联的，这种操作都不会涉及到太多的工作量。然而，我们仍然有一个大问题还没有解决。应用程序怎样“理解”用户已经选择了 Exit 或者 Open 菜单项呢？答案就在 WM_COMMAND 消息里。对于附带自己专用 ID 的每个菜单项来说，它都能生成一条 WM_COMMAND 消息，该消息定址于关联在一起的那个窗口的类窗口进程。

WM_COMMAND	0x0111
LOWORD (wParam)	菜单项 ID
HIWORD (wParam))	通知代码
lParam	控制句柄

把一个菜单模板与窗口联系起来后，我们还需要强制性地修改应用程序的窗口进程，使其能对 WM_COMMAND 进行处理，就像下面这样：

```
...
case WM_COMMAND:
    switch (LOWORD (wParam))
    {
        case MN_EXIT:
            PostQuitMessage (0);
            break;

        case MN_NEW:
        {
```

```
            MessageBox (hwnd, " You selected the New menu item"," Mnu",
            MB_OK);
        }
            break;

        case MN_SAVE:
        {
            MessageBox (hwnd, " You selected the Save menu item"," Mnu",
            MB_OK);
        }
            break;
        ...
    }
    break;
...
```

我们在菜单模板内分配的菜单项 ID 会在 wParam 的低字内显示出来，wParam 正是我们在切换（switch）语句里测试的内容。由此看来，WM_COMMAND 消息代码块就相当于一个大型的切换块，其中为菜单模板内定义的每个菜单项都包含了对应的 case（情况）块。假如 WM_COMMAND 引用的是用户在菜单内作出的选择，那么无论 wParam 的高字还是整体 lParam 都会设置成 0。这条消息亦可用于负载从一个控件到它的父/物主窗口的通知代码。在这种情况下，lParam 和 wParam 高字内的信息都会是一个非零值。对于 WM_COMMAND 消息，我们还会在第八章“Win32 的对话框管理”和第九章“预定义的窗口类”里进行更深入的探讨。

在前面列出的代码段里，假如用户选择了那个菜单项，屏幕上就会生成一个消息框，这种功能是十分有限的。在现实的编程环境中，每种菜单项情况都应该容纳一些重要的代码段，用它们来辅助定义应用程序整体行为。举个例子来说，在 MN_EXIT 的情况下（与 Exit 菜单项有关），程序会调用 PostQuitMessage（）这个 API，从而对应用程序进行中断。现在，这种功能是通过应用程序系统菜单和这个菜单项来实现的。

6.3 与菜单的交互作用

到目前为止，我们已经看到了某些基本的菜单特性。这一小节将加深大家对菜单的认识，揭示用户选择一个菜单以后在系统幕后发生的情况。

需要澄清的第一个事实是菜单栏侵占了客户区的空间，从而降低了客户区的高度；而下拉式菜单却不会与客户窗口发生冲突，因为它们占用的是从屏幕取得的一个矩形区域。为了证明这一事实，我们可以把 Menu 程序的主窗口拖到显示器底部边框附近，然后单击 File 顶级菜单。如图 6-7 所示，下拉式菜单移到了上方，这就证明是由屏幕本身提供了用于描绘菜单内容的象素。

这种事实产生的效果是相当有趣的。这些下拉式菜单对象从构造上来说与窗口内的其他

组件都不相关。唯一明显的是它们在概念上是与菜单栏连接起来的：假如用户在菜单栏上选择一个菜单，或者在另外一个下拉式菜单内选择一个弹出式菜单项，它们就会显示出来。假如这时单击了屏幕上其他某个地方，或者按下 Esc 键。在下拉式菜单内选择一个菜单项就会生成一条 WM_COMMAND 消息，这条消息会发送给物主窗口的窗口进程，如图 6-8 所示。

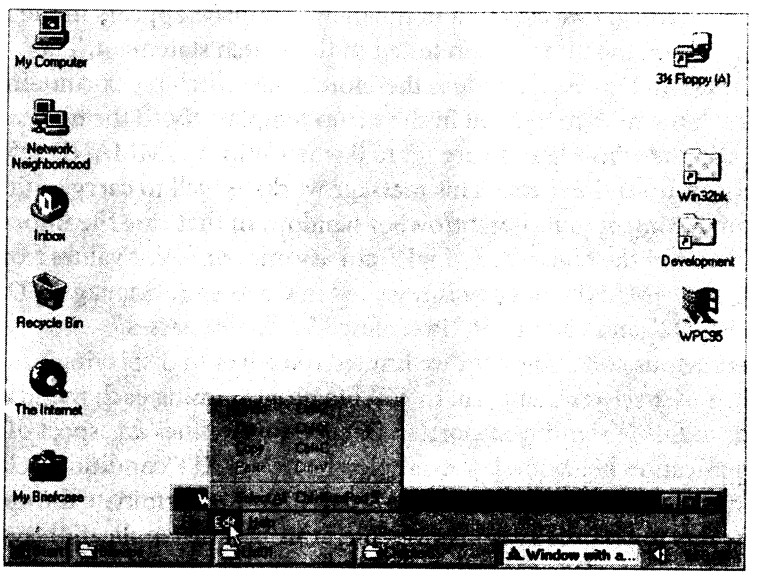


图 6-7　下拉式菜单从屏幕内获得了显示信息需要的矩形区域，而不是从客户区获得的

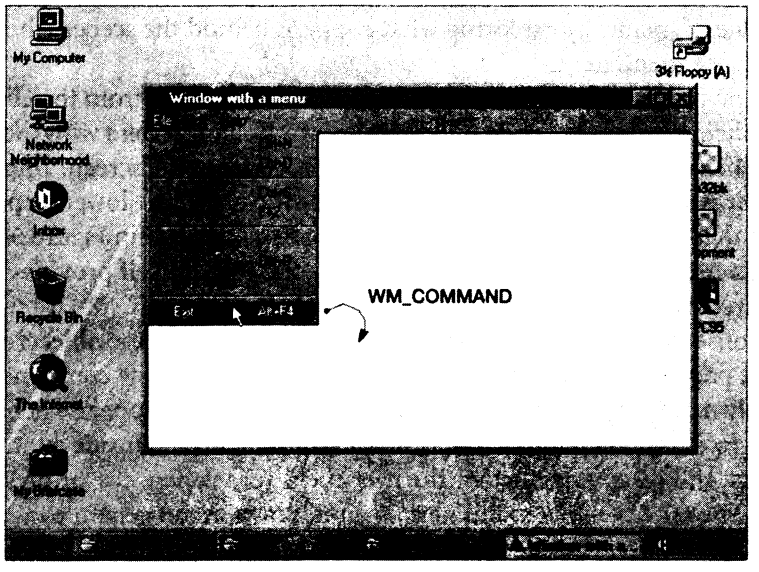


图 6-8　下拉式菜单、标题栏以及客户区之间的关系

上面揭示的只是关于菜单的很少一部分问题。假如用户按下 Alt 键或者选择一个顶级菜单，应用程序内部就会发生某些变化；视觉焦点会从窗口移至菜单栏（在这儿，“视觉焦点”的英语原文是 focus，意指一个 Win32 组件从键盘接收输入信息的能力。在同一时刻，具有视觉焦点的对象只能有一个）。视觉焦点转向菜单栏以后，应用程序会接收到一条 WM_INITMENU 消息。

WM_INITMENU	0x0016
wParam	具有视觉焦点的菜单的句柄
lParam	未使用

WM_INITMENU 是对菜单对象相关的一种初始化消息。通过对这条消息进行拦截，应用程序就可以自动捕获菜单栏句柄，同时不必调用任何一种诸如 GetMenu（）的函数。勿庸置疑，这是对窗口进程内的菜单信息进行访问的最佳手段。WM_INITMENU 消息的发送只有一次，要么在用户按下 Alt 键的时候发送，要么在用户选择一个顶级菜单的时候发送，从而使顶级菜单以反转颜色显示。应用程序则会立即接收到一条 WM_MENUSELECT 消息，并在后面紧接着一条 WM_INITMENUPOPUP 消息。这两条消息并不是很出名，但是在创建带有复杂用户界面的高质量 Win32 应用程序时却显得相当有用。实事求是地说，我们对 WM_MENUSELECT 消息的解释是相当繁杂的，这是由于在 MENU 布局的基础上实施一个菜单模板显得比较笨拙。系统会针对 POPUP 和 MENUITEM 同时生成这条消息，在消息里根据来源封装了许多不同的信息。假如 WM_MENUSELECT 指定的是一个 POPUP 菜单项，那么其中就应该包含下面这些信息：

用于 POPUP 的 WM_MENUSELECT	0x011F
LOWORD（wParam）	与 POPUP 相关的 ID
HIWORD（wParam）	表 6-3 内列出的一个或者多个标志
lParam	包含了相关下拉式菜单的句柄

用于 MENUITEM 的 WM_MENUSELECT	0x011F
LOWORD（wParam）	菜单模板内设置的菜单项的 ID
HIWORD（wParam）	表 6-3 内列出的一个或者多个标志
lParam	包含了相关下拉式菜单的句柄——前提是已设置了用于指定系统菜单的 MF_SYSMENU

表 6-3　与某些菜单项属性对应的 MF_标志

标　志	值	说　明
MF_ENABLED	0x00000000L	菜单项处于活动状态
MF_GRAYED	0x00000001L	菜单项以灰色显示，处于非活动状态
MF_DISABLED	0x00000002L	菜单项处于非活动状态（被屏蔽）
MF_BITMAP	0x00000004L	菜单项是一幅位图
MF_CHECKED	0x00000008L	设置了核选符号属性
MF_POPUP	0x00000010L	菜单项已经用 POPUP 命令进行了声明
MF_OWNERDRAW	0x00000100L	菜单项是某种所有者绘图
MF_SEPARATOR	0x00000800L	分隔符
MF_DEFAULT	0x00001000L	菜单项有缺省的属性
MF_SYSMENU	0x00002000L	菜单项指定的是系统菜单
MF_MOUSESELECT	0x00008000L	菜单项已由鼠标选定了

假如用户把视觉焦点从一个菜单转移到了窗口的另一部分、单击一次鼠标按钮或者按下 Esc 键，这些时候也会生成 WM_MENUSELECT 消息。在所有这些情况下，wParam 的低字内包含了 0xFFFF，高字则设置成 0；而 lParam 则等于 NULL。

在与菜单栏有关的消息序列最末尾处，我们可以找到 WM_INITMENUPOPUP。每次选中一个顶级菜单的时候（或者用 POPUP 命令声明一个菜单栏的时候），应用程序就会接收到这条消息，其中带有下面这些信息：

WM_INITMENUPOPUP	0x0117
wParam	相关下拉式菜单的句柄
LOWORD（lParam）	在菜单模板里用于生成下拉式菜单的菜单项的位置
HIWORD（lParam）	假如指定的是系统菜单，就为 TRUE；假如指定其他任何一个弹出式菜单，则为 FALSE

应用程序接收到与某个下拉式菜单相关的 WM_INITMENUPOPUP 消息以后，那个菜单在屏幕上还没有显示出来。因此，这条消息为我们提供了一个极佳的机会，使我们可以利用它对下拉式菜单作出某些修改。lParam 低字的内容是相当不可思议的——这个数值并不与菜单模板定义期间分配的任何一种信息对应。这是由于用 POPUP 命令声明的菜单项并不支持任何 ID。不管怎样，Windows 都会在内部为每一种菜单项（无论是 MENUITEM 还是 POPUP）保留一个编号。最左侧的那个顶级菜单编号为零，从左向右，每增加一个菜单项，对应的编号就增 1。类似地，下拉式菜单内的菜单项也带有这种根据位置定义的 ID——第一个为零，后续的菜单项则连续增 1。图 6-9 向大家展示了 Windows 用于计算菜单项的这种内部机制。

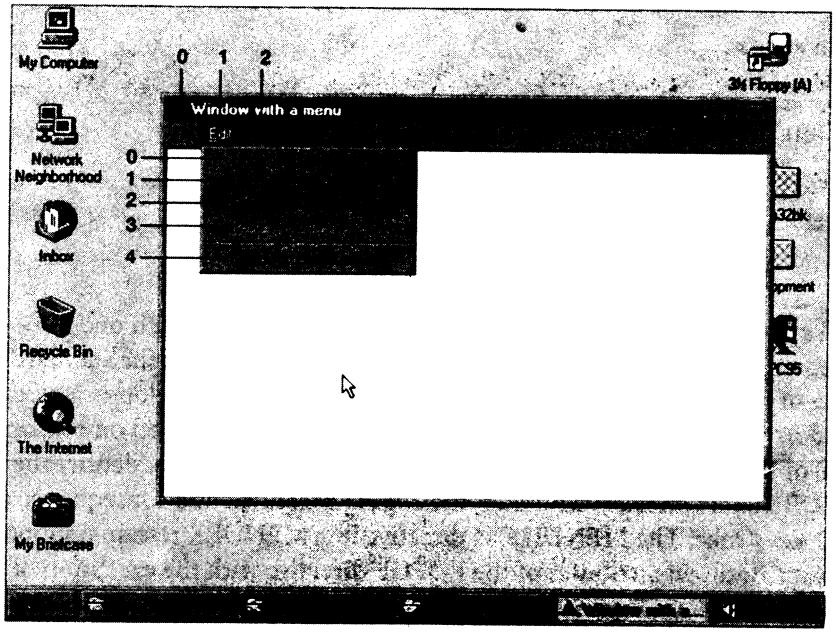


图 6-9　顶级菜单在内部是由系统自动生成的一个位置 ID 标识的，ID 的取值不能小于零。同样的规则亦适用于下拉式菜单

某些菜单 API 允许我们指定一个菜单项的 ID 或者它的位置编号，从而对这个菜单项进行访问。其中，第二种位置编号采用的就是图 6-9 展示的那种编号机制。当用户单击下拉式菜单内的某个菜单项时，它就会连续生成一条 WM_MENUSELECT 和一条 WM_COMMAND 消息。

6.4 扩展菜单

对 POPUP 命令声明的所有菜单项进行标识的时候，假如菜单项缺少对应的 ID，就会为开发者带来很多问题，从而对设计出具有良好外观的应用程序留下障碍。为了解决这个问题，微软公司定义了一种新的菜单模板布局，也就是我们在这一节要讨论的扩展菜单。资源编译器现在支持 MENUEX 命令，该命令允许我们根据一种更可靠、更有效的模式对菜单模板进行设计。这种方案对 POPUP 菜单项进行的最大改动就是提供了一种更加复杂、更加灵活的语法结构，它能同时支持 ID 和一个辅助性的标识符。不幸的是，Visual C++当前还没有实现对这种资源的支持，所以我们不得不对其进行手工编写。

MENUEX 命令的语法与 MENU 是非常类似的，它们之间只存在一些细微的差别：

```
menuID MENUEXE
BEGIN
    POPUP itemText [, [id] [, [type] [, [state] [, [helpID]]
    BEGIN
        MENUITEM itemText [, [id] [, [type] [, state]]
        ...
    END
    ...
END
```

其中，最关键的组件仍然是 POPUP 和 MENUITEM，每一个都用一个非常明显的附加参数集进行了合成。除了标准的正文串以外，POPUP 菜单项现在可以支持由开发者提供的一个 ID、通过 WINUSER.H 的 MFT_定义而选择的一种类型、以 WINUSER.H 内的一种或者多种 MFS_风格为基础的显示状态以及用于标识 POPUP 菜单的一个 ID——这个 POPUP 菜单是在用户按下 F1 键，从而生成一条 WM_HELP 消息的时候显示出来的。MENUEX 资源内的 MENUITEM 命令支持与 POPUP 命令类似的可选属性，只是帮助 ID 除外。表 6-4 为大家总结了所有的 MFT_定义——即菜单类型。

对于一个 MENUEX 资源内的 POPUP 和 MENUITEM 命令来说，它们支持的所有菜单风格都列于表 6-5 内。

考虑到表 6-4 和 6-5 里列出的所有标志都是在 WINUSER.H 头文件里定义的，所以有必要在资源文件里包括这个头文件，这样才能避免编译器在处理一个 MENUEX 对象的时候生成出错消息。下面这段标签名为 MAINMENUEX 的资源是从 Menuex 这个例子的资源文件里摘抄下来的。Menuex 这个例子可以在本书附带 CD 的 Listing 6.2 里找到。

表 6-4　用于定义菜单项对象本质的 MFT_标志

标　志	值	说　明
MFT_STRING	MF_STRING	菜单项是一个正文串
MFT_BITMAP	MF_BITMAP	菜单项是一幅位图
MFT_MENUBARBREAK	MF_MENUBARBREAK	行为与 MENUBREAK 类似，只是用垂直线分隔了连续的顶级菜单
MFT_MENUBREAK	MF_MENUBREAK	菜单项移动到一个新栏内，从而产生了一种多栏效果
MFT_OWNERDRAW	MF_OWNERDRAW	菜单项是所有人绘图
MFT_RADIOCHECK	0x00000200L	生成一个单选按钮菜单项，正文左边带有一个小圆点
MFT_SEPARATOR	MF_SEPARATOR	菜单项是一条水平的细线
MFT_RIGHTORDER	0x000020000L	
MFT_RIGHTJUSTIFY	MF_RIGHTJUSTIFY	

表 6-5　MENUEX 资源内任何一种菜单项均可运用的菜单风格

标　志	值	说　明
MFS_GRAYED	0x00000003L	菜单项被屏蔽（非活动状态），并以灰色显示
MFS_DISABLED	MF_DISABLED	菜单项处于屏蔽状态
MFS_CHECKED	MF_CHECKED	菜单项被核选
MFS_HILITE	MF_HILITE	菜单项以高亮度显示
MFS_ENABLED	MF_ENABLED	菜单项处于活动状态
MFS_UNCHECKED	MF_UNCHECKED	菜单项处于非核选状态
MFS_UNHILITE	MF_UNHILITE	菜单项未以高亮度显示
MFS_DEFAULT	MF_DEFAULT	缺省的菜单项

```
...
mainmenuex MENUEX
BEGIN
    POPUP " &File", MN_FILE, MFT_STRING, MFS_ENABLED, 777
    BEGIN
        MENUITEM" &New", MN_NEW, MFT_STRING, MFS_GRAYED
        MENUITEM " &Save", MN_SAVE, MFT_STRING
        MENUITEM "",, MFT_SEPARATOR
        MENUITEM" &Open…", MN_OPEN, MFT_STRING, MFS_
            DEFAULT
        MENUITEM "",, MFT_SEPARATOR
        MENUITEM " E&xit", MN_EXIT, MFT_STRING
    END

    POPUP " &Edit", MN_EDIT, MFT_STRING, MFS_ENABLED, 767
    BEGIN
        MENUITEM " &Undo\tCtrl+Z", MN_UNDO,
        MFT_STRING
```

```
        MENUITEM " Cu&t\tCtrl+X", MN_CUT, MFT_STRING
        MENUITEM " &Copy\tCtrl+C", MN_COPY, MFT_STRING
        MENUITEM " &Paste\tCtrl+V", MN_PASTE, MFT_STRING,
          MFS_GRAYED
        MENUITEM "",, MFT_SEPARATOR
        MENUITEM " Select &All\tCtrl+NumPad 5", MN_SELECTALL,
          MFT_STRING
    END

        POPUP " &Help", MN_HELP, MFT_STRING, MFS_ENABLED, 757
    BEGIN
        MENUITEM " &Contents\tF1",      ID_ HELP_ CONTENTS, MFT_
        STRING
        MENUITEM " &Search…", ID_HELP_SEARCH, MFT_STRING
        MENUITEM "",, MFT_SEPARATOR
        MENUITEM " About Menu…",    ID_ HELP_ ABOUTMENU, MFT_
        STRING
    END
END
…
```

请大家观察图 6-9，从 Menuex 示例程序的显示外观来看，我们无法区分出由 MENU 资源生成的一个标准菜单栏。LoadMenu () 的作用通常是把一种 MENUEX 资源载入应用程序的内存地址空间内，然后返回一个普通的菜单句柄。

我们怎样利用由一个 MENUEX 资源提供的扩展菜单信息呢？在回答这个问题之前，我们需要先强调一个重要的准则：在 Win32 里，菜单只应该按照 MENUEX 语法进行设计，不要沿用老式的实现方法——尽管出于兼容性的考虑，这种方法在 Win95 里仍然得到了实施和支持。

扩展菜单模板是通过调用 LoadMenu () 载入内存的，然后用 SetMenu () 函数把它与主窗口关联到一起，关于这方面的问题我们在前面已经讨论过了。对于 POPUP 菜单项来说，它们现在已有了自己对应的 ID，这些菜单项会生成 WM_COMMAND 消息吗？不会！尽管 ID 的存在可以唯一性地标识 POPUP 菜单项在 MENUEX 资源里的存在，但是这些菜单项根本不会生成 WM_COMMAND 消息。然而，在对 POPUP ITEMS 进行侦测，并为用户选中的每个菜单项提供特定行动的时候，POPUP ID 就显得相当有用了。

对于一些非常流行的应用程序来说，比如 Microsoft Excel 和 MS Word，它们已经教会了大多数程序员怎样通过状态栏提供合成的、有效的反馈信息。在这儿，状态栏是客户区底部的一个小窗口。把鼠标指针定位于某个菜单项上方时，状态栏就会同时显示出一段简短的正文，它解释了假如选中并执行这个菜单项，会得到什么效果，会采取何种对应的行动。对于带有状态栏的应用程序来说，它把所有这些正文串都存储于 STRINGTABLE 资源内，并且根

据菜单项的 ID 载入对应的正文。为了实现状态栏正文的显示，首先要根据当前选中的菜单项，从而对 WM_MENUSELECT 消息进行拦截。假如这条消息引用了由 MENUITEM 命令定义的某个菜单项，那么以后的操作就会很顺利地完成。程序会接收到菜单项的 ID，并且根据这个 ID 提取出事先存储好的正文串，然后在状态栏内把它们显示出来。

假如 WM_MENUSELECT 消息引用的是一个 POPUP 菜单项，开发者就需要针对这种情况设计相应的代码。对于我们提到的 Menuex 示例程序来说，它调用了 GetMenuItemInfo() 函数，从而取回了 POPUP 菜单项 ID。这个新的 API 函数具有强大的功能，它是针对与扩展菜单有关的所有问题专门设计的。

```
#include〈winuser.h〉
BOOL WINAPI GetMenuItemInfo (HMENU hMenu,
                             UINT uItem,
                             BOOL fByPosition,
                             LPMENUITEMINFO lpmii);
```

参数	说明
HMENU hMenu	菜单句柄
UINT uItem	菜单项 ID 或者它在菜单模板内的位置
BOOL fByPosition	如为 TRUE 则说明 uItem 参数是一个菜单项的位置而非一个 ID
LPMENUITEMINFO lpmii	一个 MENUITEMINFO 数据结构的地址
返回值	在正文里讨论
BOOL	布尔值，假如函数调用成功，就返回 TRUE；假如函数调用失败，则返回一个 FALSE 值

这个新 API 的关键元素就是 MENUITEMINFO 数据结构：

```
typedef struct tabMENUITEMINFO
 {
    UINT cbSize;
    UINT fMask;
    UINT fType;
    UINT fState;
    UINT wID;
    HMENU hSubMenu;
    HBITMAP hbmpChecked;
    HBITMAP hbmpUnchecked;
    DWORD dwItemData;
    LPSTR dwTypeData;
    UINT cch;
```

} MENUITEMINFO, * LPMENUITEMINFO;

其中的第一个参数是 cbSize，该参数内包含了整体结构的范围，这在许多新的 Win32 数据结构里已经是一种非常常见的特性了。在 MENUITEMINFO 结构里，真正起决定作用的参数还是 fMask。该参数的值是由表 6-6 内列出的一个或者几个标志组成的。

表 6-6 用于 MENUITEMINFO 结构 fMask 项的标志

fMask 标志	值	说明
MIIM_CHECKMARKS	0x00000008	激活 hbmpChecked 和 hbmpUnchecked 项
MIIM_DATA	0x00000020	指定 dwItemData 项
MIIM_ID	0x00000002	指定 wID 项
MIIIM_STATE	0x00000001	指定 fState 项
MIIM_SUBMENU	0x00000004	激活 hSubMenu 项
MIIM_TYPE	0x00000010	指定 fType 和 dwTypeData 项

对 GetMenuItemInfo () 的调用和以前进行的各种调用都是有很大区别的，这是由于由多个 MIIM_标志产生了各种各样不同的组合。这些标志的基本作用是指示 GetMenuItemInfo () 函数对目标菜单项采取什么样的操作。事实上，我们利用这个函数可以很方便地获得一个菜单项的 ID、把菜单项的状态从活动状态改为灰色显示、取回与某个 POPUP 项关联起来的一个下拉式菜单的句柄、修改菜单项内显示的信息以及获得相关的核选符号。除此以外，它还具有一种最受人欢迎的功能，那就是每个菜单项都有一个专用的内存区域，每个区域的长度是四个字节。在这种内存区域内，应用程序开发者可以存储任何种类的信息和数据。GetMenuItemInfo () 只会对那些已被适当 MIIM_标志激活的那些项进行填写。如果想改变某个菜单项的属性、状态或者其他任何一种信息，我们可以接着调用 SetMenuItemInfo () 这个 API，它的语法结构与 GetMenuItemInfo () 是完全一致的：

```
#include <winuser.h>
BOOL WINAPI SetMenuItemInfo (HMENU hMenu,
                             UINT uItem,
                             BOOL fByPosition,
                             LPMENUITEMINFO lpmii);
```

在 MENUEX 里，我们拦截了 WM_MENUSELECT 消息，并且立即把 wParam 里的信息存储到另外一个 WPARAM 标识符——wp 里，如下所示：

```
WPARAM wp = wParam;
```

随后，我们测试了 ME_POPUP 标志是否存在于 wParam 的高字里。如果在高字里，就表明消息的起源是一个 POPUP 项。我们接下来的行动是调用 GetMenuItemInfo () 函数，并且把其中的 fMask 项设置成 MIIM_ID：

```
mii.cbSize = sizeof (mii);
mii.fMask = MIIM_ID;
mii.wId = 0;
```

为了取得一个 POPUP 项的 ID，我们现在要调用 GetMenuItemInfo ()，并且通过调用

GetMenu () 传递菜单句柄，同时传递位于 wParam 低字内的菜单项位置。

```
if (! GetMenuItemInfo (GetMenu (hwnd), LOWRD (wParam), TRUE, &mii))
    MessageBeep (0);
```

在 mii. wID 项内包含了 POPUP 项的 ID，这正是我们千方百计要寻找的。从另外一方面来看，假如 WM_MOUSESELECT 引用了一个 MENUITEM 项，那么当我们改变应用程序窗口的标题栏时，就会返回一些相关的信息，这些信息涉及到鼠标在下拉式菜单上的位置、菜单项的总数、鼠标坐标以及菜单句柄等等：

```
...
else
{
    // hmenu of the pulldown
    hmenu = (HMENU) lParam;
    // get the current cursor position
    GetCursorPos (&pt);
    // get the ID
    iPos = LOWORD (wParam);
    // menuitem position
    i = MenuItemFromPoint (hwnd, hmenu, pt);
    nItems = GetMenuItemCount (hmenu);
    wsprintf(szText, "Pos %d - hmenu %ld - X:%ld Y:%ld - #items: %
    d", i, (HMENU)lParam, pt.x, pt.y, nItems);
    SetWindowText(hwnd, szText);
}
...
```

图 6-10 向大家展示了 MENUEX 这个例子，这是一个具有标准外观的应用程序。其中的菜单栏则来自 MENUEX 资源。

6.4.1 从头建立一个菜单

现在让我们自己动手，从头开始建立一个菜单，不利用资源文件里任何现成的模板。首先，我们要调用 CreateMenu () 这个 API，它能返回指定空对象的一个菜单句柄。这样一来，函数成功地执行以后，它就会返回对应的菜单句柄，同时不会生成与某个窗口联系起来的任何一个可见的菜单。

```
#include <winuser.h>
HEMNU WINAPI CreateMenu (void);
```

为了插入 POPUP 和 MENUITEM 菜单项，我们需要使用一系列菜单 API，比如传统的 ChangeMenu () 或者更新、更可靠的 AppendMenu ()，InsertMenu ()，ModifyMenu ()、DeleteMenu () 和 RemoveMenu () 等等。Win95 的 API 还提供了与 MENUITEMINFO 数据结构联系起来的第三种函数：InsertMenuItem ()。解决我们碰到的这种问题时，该函数就

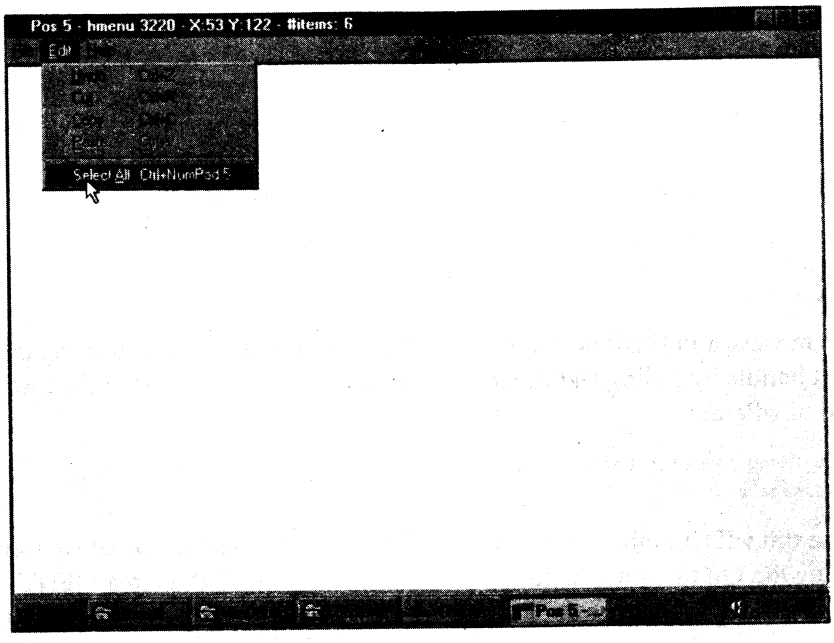


图 6-10　MENUEX 支持一种扩展的菜单模板——Win32 下的新型菜单格式

显得相当有用了，它的用法相当灵活，功能也比较强。

```
#include〈winuser.h〉
BOOL WINAPI InsertMenuItem (HMENU hMenu,
                            UINT uItem,
                            BOOL fByPosition,
                            LPMENUITEMINFO lpmii);
```

参数	说明
HMENU hMenu	菜单句柄
UINT uItem	菜单项 ID 或者它在菜单模板内的位置
BOOL fByPosition	假如为 TRUE，就说明 uItem 参数是一个菜单项的位置，而不是一个 ID
LPMENUITEMINFO lpmii	一个 MENUITEMINFO 数据结构的地址
返回值	在正文里讨论
BOOL	布尔值，假如函数调用成功，就返回 TRUE 值；假如失败，则返回一个 FALSE

InsertMenuItem () 的运作要以前面对 CreateMenu () 进行调用后返回的菜单句柄为基础，并会用菜单栏或者下拉式菜单内的新项目不断地对其进行填充。这种方式看起来显得有些累赘，但它的确是解决这种特定问题的最佳方案。通常情况下，我们宁愿在资源文件里生

成两个甚至更多的菜单模板，这些模板是针对应用程序执行过程中的特定条件而设计的。开发者一般都不愿意动态地改变、增加或者删除某些菜单项。在本书附带 CD 的 Listing 6.5 内，我提供了一个名为 CR-MENU 的例子，它解释了如何实现这种技术。

6.4.2 在运行期间修改菜单

根据应用程序的逻辑，菜单项有时会在程序执行过程中改变自己的状态。某些情况下，程序要求一个菜单项变成以灰色显示、放置一个核选符号、激活某个菜单项或者在现成菜单内增加新菜单项。这些变动都是在运行期间发生的，所以在设计菜单模板的时候预先把它们都计划进去。正如我们以前看到的那样，SetMenuItemInfo () 是一种功能很强并且具有常规用途的 API 函数，它有能力满足我们刚才提出的这些需要。作为另外一种选择，Win32 仍然支持开发者用传统的方式对菜单项的属性进行动态修改。CheckMenuItem () 允许我们在任何一个给定的菜单项内分配或者消隐核选符号，它的语法结构如下所示：

```
#include <winuser.h>
BOOL WINAPI CheckMenuItem (HMENU hmenu, UINT idCheckItem, UNIT
uCheck);
```

标志	值	说明
MF_BYCOMMAND	0x00000000L	指出第二个参数引用的是一个菜单项 ID
MF_BYPOSITION	0x00000400L	指出第二个参数引用的是一个菜单项位置
MF_CHEDKED	0x00000008L	设置核选符号
MF_UNCHECKED	0x00000000L	删除核选符号

返回值既可能是 MF_CHECKED，也有可能是 MF_UNCHECKED，它们表明了修改过的菜单项以前的状态。如果想激活或者屏蔽一个菜单项，我们可以使用 EnableMenuItem () 函数。该函数具有完全一致的语法结构，只是增加了三个新的 MF_标志，如下所示：

```
#include <winuser.h>
BOOL WINAPI EnableMenuItem (HMENU hmenu, UNIT idEnableItem, UNIT
uEnable)
```

标志	值	说明
MF_BYCOMMAND	0x00000000L	指出第二个参数引用的是一个菜单项 ID
MF_BYPOSITION	0x00000400L	指出第二个参数引用的是一个菜单项位置
MF_DISABLED	0x00000002L	屏蔽菜单项
MF_ENABLED	0x00000000L	激活菜单项
MF_GRAYED	0x00000001L	屏蔽菜单，并使其以灰色显示

对于一个被屏蔽或者灰色显示的菜单项来说，用户选择它的时候，即使它能以高亮度显示，也依然无法生成一条对应的 WM_COMMAND 消息。在 Windows 95 里，假如鼠标指针覆盖于一个菜单项上方，该菜单项就会自动以高亮度显示。假如想强制使用或者消隐这种视觉效果，就应该考虑调用 HiliteMenuItem () 函数：

```
#include <winuser.h>
BOOL WINAPI HiliteMenuItem (HWND hwnd,
                            HMENU hmenu,
                            UINT idHiliteItem,
                            UINT fuHilite);
```

和前面两个函数不一样，HiliteMenuItem()要求同时使用一个窗口句柄和一个菜单句柄，用它们分别指定包含了菜单的窗口以及菜单本身。除了菜单项 ID（或者它的位置）以外，还要用到下面这些标志中的一个或者多个，否则是无法完成这一函数调用的：

标志	值	说明
MF_BYCOMMAND	0x00000000L	指出第二个参数引用的是一个菜单项 ID
MF_BYPOSITION	0x00000400L	指出第二个参数引用的是一个菜单项的位置
MF_HILITE	0x00000080L	对菜单项进行高亮度处理
MF_UNHILITE	0x00000000L	从菜单项内消去高亮度效果

在本书附带 CD 的 Listing 6.3 里，大家可以找到一个名为 Checkmrk 的例子，该例子充分展示了菜单函数的应用。其中，Icon 菜单项的左边有一个核选符号，以此证明了应用程序客户区内一个相同图标的存在。如图 6-11 所示。

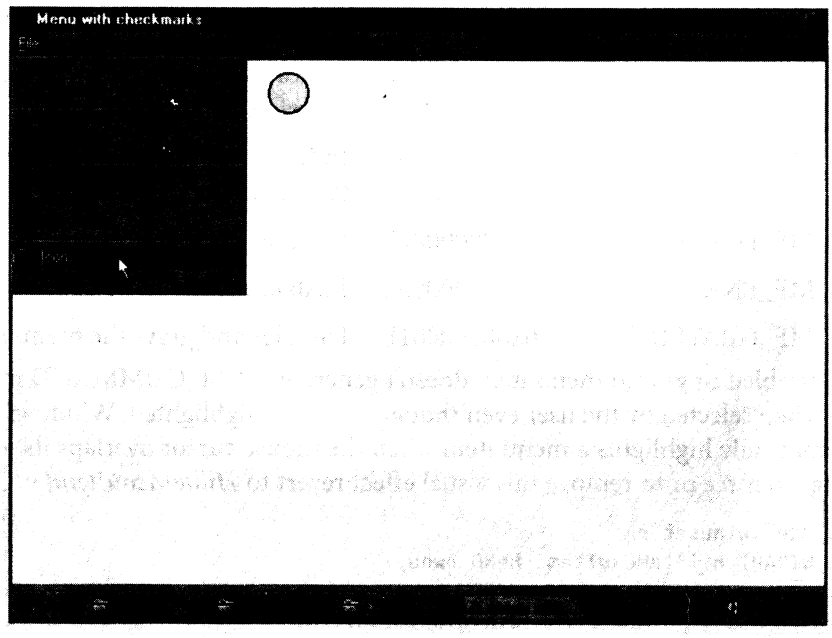


图 6-11　CHECKMRK 能够动态显示或者消隐客户区内的一个图标，
并且对 Icon 菜单项作出相应的修改

通过选择 Icon 菜单项，应用程序内部会发生两件事情：核选符号改变自己的形状以及图

标消失（如图 6-12 所示）。

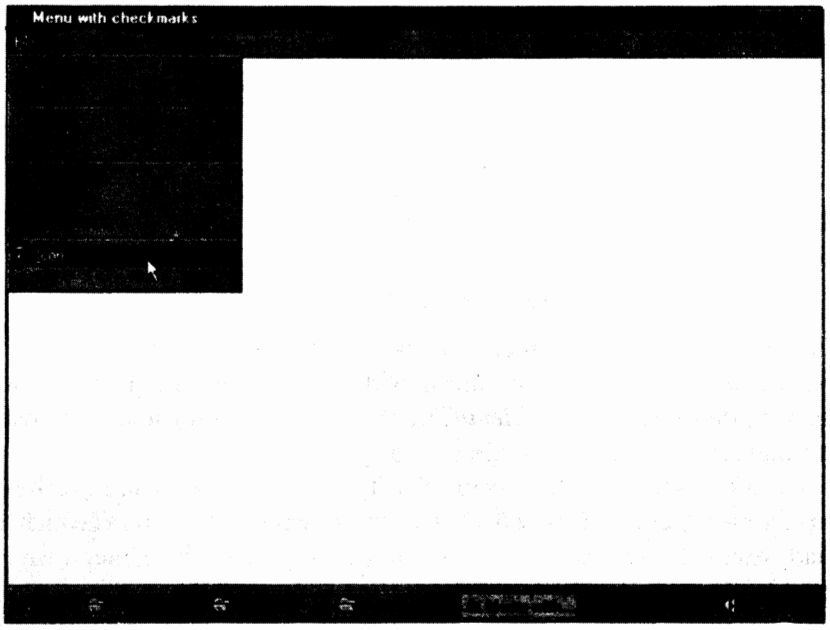


图 6-12　选择 Icon 菜单项后，客户区内的图标消失了，
这时的核选符号看起来和以前也颇不相同

我们在这个例子里实现的东西对于 Win32 应用程序来说并不是一种传统的行为。通常，在菜单项正文左边显示出的符号是一种标准的核选符，并非图 6-11 和 6.12 那样的一个圆或者带×号的圆。Win32 为我们提供了对核选符号进行定制的工具，开发者可以选择用户自定义的一幅位图。在 SetMenuItemBitmap（）和 GetMenuCheckMarkDimensions（）这两个函数的辅助下，要实现核选符的变动是件轻而易举的事情：

```
#include〈winuser. h〉
BOOL WINAPI SetMenuItemBitmap (HMENU hmenu,
                               UINT idItem,
                               UINT fuFlags,
                               HBITMAP hbmUnchedked,
                               HBITMAP hbmChecked);
```

参数	说明
HMENU hmenu	菜单句柄
UINT idItem	菜单项 ID 或者菜单项在菜单模板内的位置
UINT fuFlags	指定 idItem 应该当作 ID 对待，还是应该当作一种位置标识对待（MF_BYCOMMAND 或者 MF_BYPOSITION）
HBITMAP hbmUnchedked	指定菜单项未被核选的时候需要显示的位图

HBITMAP hbmChecked	指定当作核选符号显示的位图
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回 TRUE，否则返回 FALSE

我们在这儿需要做的一切就是把两幅位图与特定的项联系在一起。当应用程序调用 HiliteMenuItem ()（或者 SetMenuItemInfo ()）选择视觉效果或者消隐核选符号的时候，hbmCheckedt 和 hbmUnchecked 位图就会在下拉式菜单里显示出来。应用程序必须装载这两幅位图，并且在中断前的清除操作中对它们的删除进行管理。正是由于使用了这个函数，我们才能把任何一幅位图设置成核选符号，并且对下拉式菜单第一栏的标准尺寸进行修改。GetMenuCheckMarkDimensions () 返回的信息存储在一个长字标识符内。其中，位图宽度存放于低字，位图高度存放于高字：

```
#include〈winuser.h〉
long WINAPI GetMenuCheckMarkDimensions (void);
```

假如我们的目标是联系一幅新位图，用它来替换掉标准的核选符号，但是是不改变栏宽和连续两个菜单项之间的距离，那么就可以调用 GetMenuCheckMarkDimensions ()；用它载入和建立新的位图，如有必要，还可以对原始图标进行伸缩处理。

Checkmrk 程序还提供了另一种特性。为了证明它，我们可以看看其中的 Options 下拉式菜单，如图 6-13 所示。这个下拉式菜单列出了四个菜单项，它们是互相排斥的。这其实就是单选菜单的一种典型形式。对于这种菜单来说，同时只能选择其中的一项，不能多选。

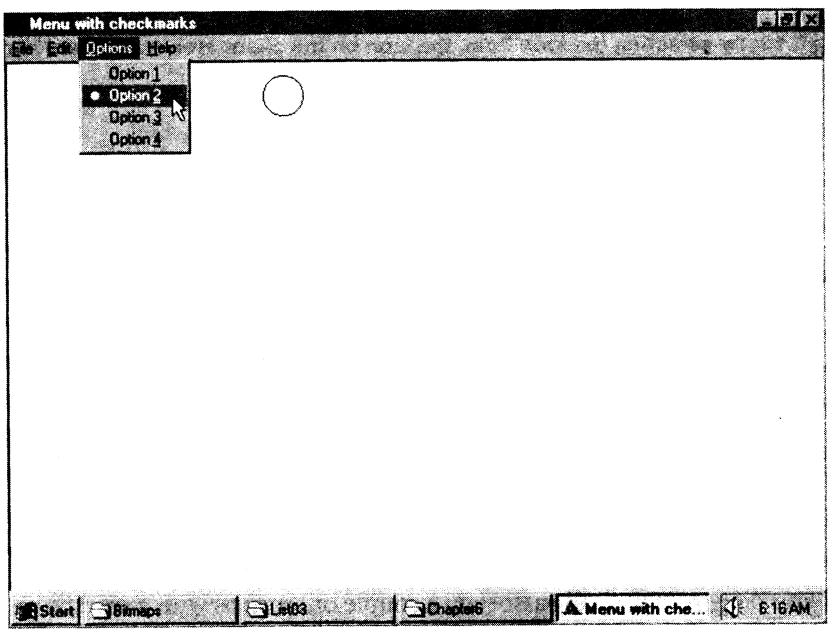


图 6-13　Options 下拉式菜单是一个典型 Windows 95 单选菜单。只有一项在自己的左侧显示了一个圆点，这是用于表明当前谁被选中的一种图形化属性

在编程中，为了达到这种效果，我们可以在 MENUEX 资源里为每个菜单项分配 MFS_RADIOCHECK 风格；或者在执行过程中调用 CheckMenuRadioItem () 函数。下面这个代码段向大家展示了图 6-13 那四个选项是如何转换成单选菜单的：

```
...
// treat the menuitems in the Options pulldown as a radio menu
CheckMenuRadioItem (GetMenu (hwnd), MN_OPTION1, MN_OPTION4, MN_
OPTION1, MF_BYCOMMAND);
...
```

其中有两个定义：MN_OPTION1 MN_OPTION4，它们指定了一系列连续菜单项的两个极端，第四个参数则用于指示菜单在哪儿放置那个圆点。当用户选中一个单选项的时候，程序就会再次调用 CheckMenuRadioItem () 函数，从而消隐选定圆点，并且把它放置到对应菜单项的左侧：

```
...
case MN_OPTION1;
case MN_OPTION2;
case MN_OPTION3;
case MN_OPTION4;
{
    ...
    // set the currently selected radio menu
    CheckMenuRadioItem (GetMenu (hwnd),
                        MN_OPTION1, MN_OPTION4,
                        LOWORD (wParam), MF_BYCOMMAND);
}
    break;
...
```

6.4.3 一次性载入多个菜单

对于按照 MDI 规范开发的 Windows 应用程序来说，其中同时用到几种菜单模板是很常见的事情。在这种情况下，主窗口的地位只是简单地相当于几个子窗口（即常说的文档窗口）的一个容器，所有实际的工作都是在子窗口里完成的。现在让我们同时启动 Microsoft Excel 的两个拷贝——在一个拷贝里选择一个文档，在另一拷贝里则选择一个宏表（macro sheet），然后自己来模板这种行为。在本书附带 CD 的 Listing 6.4 里，我们提供了一个名为 Mmenu 的例子。这个例子运行起来就像一个匿名窗口，其中不具备任何引人注目的特征。这时，假如选择 File 菜单内的 New 菜单项（如图 6-14 所示），整体菜单栏都会被另一个菜单栏取代，如图 6-15 所示。

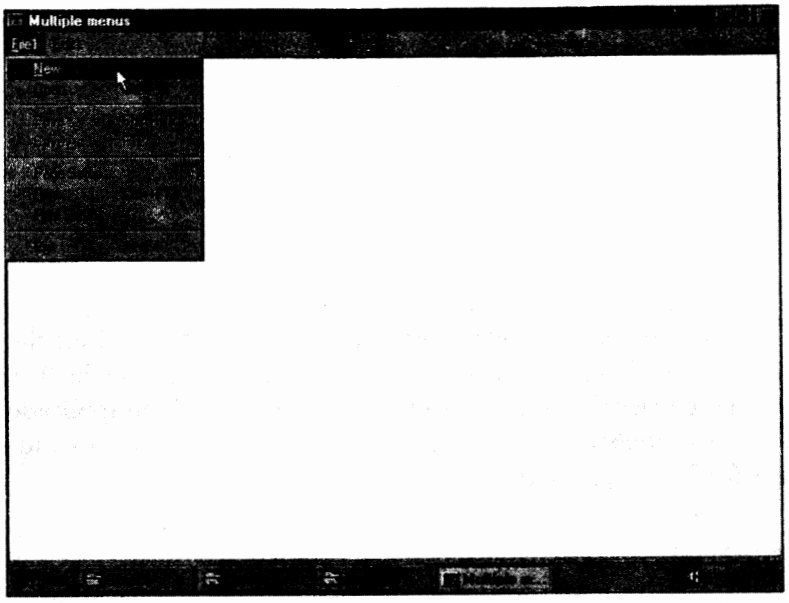


图 6-14　通过选择 New 和 Back 菜单项，MMENU 可以在两种菜单模板之间来回切换

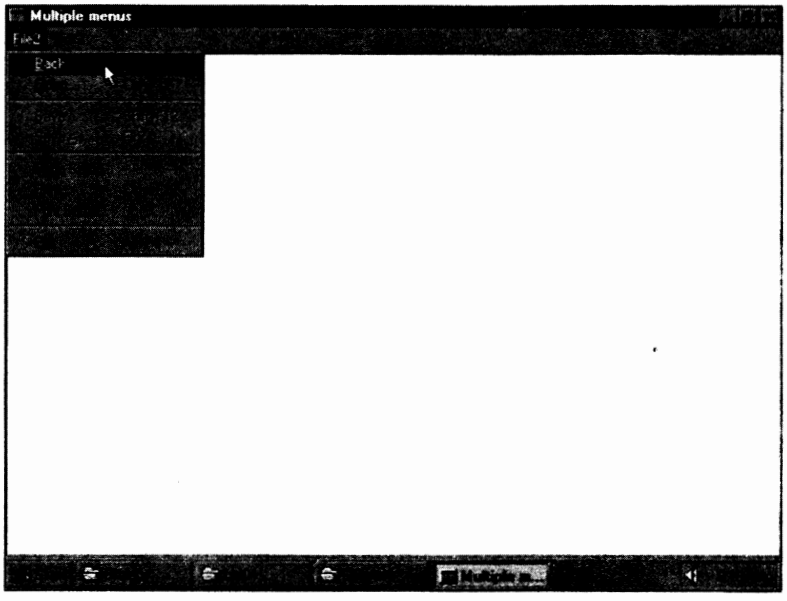


图 6-15　选择 Back 菜单项后，MMENU 又恢复成了原始的菜单

MMENU 拦截 WM_CREATE 消息的时候一次性载入了两个菜单模板。返回的句柄存储于两个 HMENU 静态标识符内，从而让它们在整个窗口进程里都是“可见”的。随后，通过 SetMenu（）的调用，第一个句柄就与主窗口联系起来了：

```
SetMenu (hwnd, hmenu1);
```

当用户选择了 New 菜单项后，程序就会接收到一条 WM_COMMAND 消息，其中 wParam 低字内就包含了 MN_NEW：

```
...
case WM_COMMAND:
    switch (LOWORD (wParam))
     {
        case MN_NEW:
            //associating the second menu template
            SetMenu (hwnd, hmenu2);
            break;
        ...
```

如果想用新菜单替换当前使用的菜单，只需再次调用 SetMenu ()，并在其中附带另外一个适当的菜单句柄即可。这个菜单模板与第一个是基本一致的，只是它多了这样一个选项：Back。从这个名字我们就可以推测出它的作用——一旦选中 Back，就会通过对 SetMenu () 一次类似的调用，从而恢复原始的菜单：

```
...
case MN_BACK:
    // restoring the first menu template
    SetMenu (hwnd, hmenu);
    break;
...
```

两个菜单句柄应该存储于应用程序地址空间的其他某个地方。正如我们以前提到的那样，有几种途径都可供选择，比如使用属性列表、窗口字以及动态分配内存区域等等。

6.5 菜单的修改

除了装载和删除菜单以外，在程序的执行过程中，我们还需要根据应用程序的特定需求对现成的菜单项属性进行修改。Checkmrk 示例程序向我们展示了如何分配和删除一个定制的核选符号。现在，让我们再试着动态地激活和屏蔽一个菜单项，或者对缺省属性进行设置。在进行这种工作之前，先让我们在设计理论上下一点功夫。

下拉式菜单内列出的菜单项应该归属于下面这三种类别的其中之一：命令、扩展命令和状态设置器。对于命令菜单项来说，假如选中它，就会立即生成一条 WM_COMMAND 消息。Exit 是属于这种类别的一个典型的菜单项。应用程序把用户对这种菜单项的选择理解为直接的命令，从而转向执行消息块内放置的相应的代码。假如用户想打开一个文件，那么命令菜单项就不管用了。这种操作需要用到一个扩展命令。以程序开发的眼光来看，命令菜单项和

扩展命令菜单项之间并没有什么区别。这两种命令都会生成一条定址于应用程序窗口进程的WM_COMMAND 消息。这两者间的唯一差异表现于视觉效果和概念上。对于一条扩展命令来说，从视觉效果的角度来看，它的正文串后面带有一个省略号（…）。这个省略号向用户表明：假如选中这个菜单项，那么不会立即生成一条命令，而是强制性地生成一个对话窗口。所以，对我们来说，扩展命令是帮助自己引入一个对话框的合适工具。Open 正是这样的一种菜单项。

与命令和扩展命令菜单项比较起来，我们用到状态设置器菜单项的机会要少得多。从用户操作的角度来看，这种菜单项与命令菜单项是类似的，因为它们都能在应用程序内立即执行一条命令。但是与命令菜单项不同，状态设置器提供了核选符号，从而指出那种行动当前正在进行。对于两种互相排斥的操作来说，比如显示和隐藏一个窗口，状态设置器就显得相当有用了。假如，假如某个辅助性窗口是可见的，对应状态设置器菜单项就会显示出一个核选符号；假如把该窗口隐藏起来了，这个核选符号就会消失。一些例子可以让我们更直观地掌握这些概念。如图 6-16 所示，其中显示的是 MS Word 的 Edit 菜单。这个菜单里的 Find 和 Go To，另外还有 File 菜单里的 Open 都属于典型的扩展命令，它们后面都带有一个省略号。假如选择这些命令，就会在屏幕中显示出相应的对话框，如图 6-17 所示。

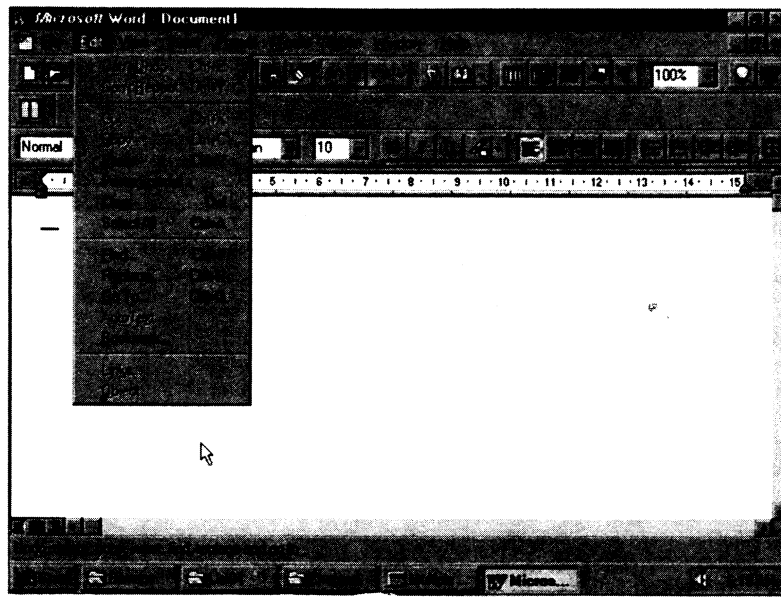


图 6-16　Microsoft Word 的 Edit 菜单里包含了几个命令和扩展命令项

Close，Save 和 Print Preview 是命令菜单项的三个例子。选择它们中的任何一个，都会立即执行相应的命令，并且采取相应的行动。对这三个命令来说，它们分别会关闭和存储活动的文档，以及提供当前文档的打印预览效果。如果想看一看状态设置器的样子，我们可以转向其中的 View 顶级菜单，如图 6-18 所示。

正如图 6-18 显示的那样，标尺（位于菜单栏下方的第三个水平对象）是在 MSWord 当前

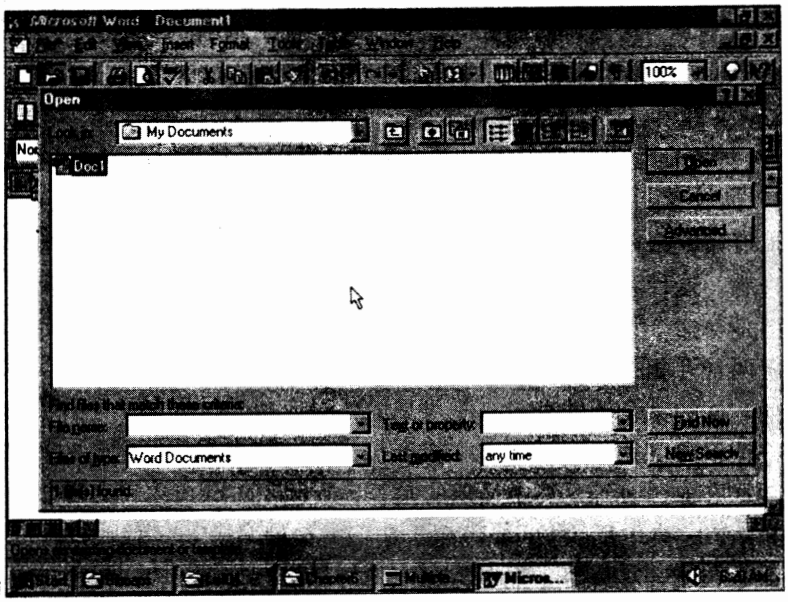


图 6-17　Microsoft Word 里与 Open 菜单项对应的新的通用对话框

的状态下是显示出来的。它的激活与否是通过 View 菜单的 Ruler 菜单项控制的。如果想从 MS Word 里把标尺消隐掉，只需再次选择 Ruler 菜单项即可；在这以后，无论标尺还是核选符都从 Word 里消失了，如图 6-19 所示。

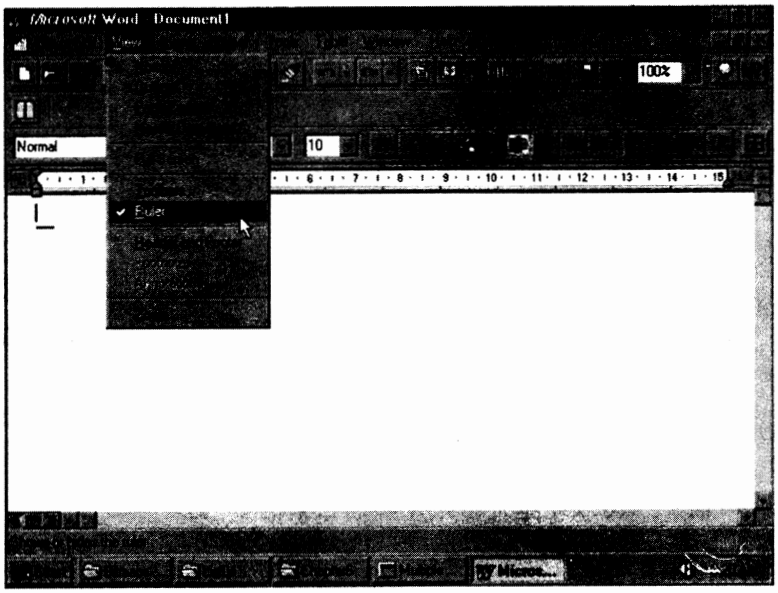


图 6-18　View 菜单内的 Ruler 项是一种典型的状态设置器

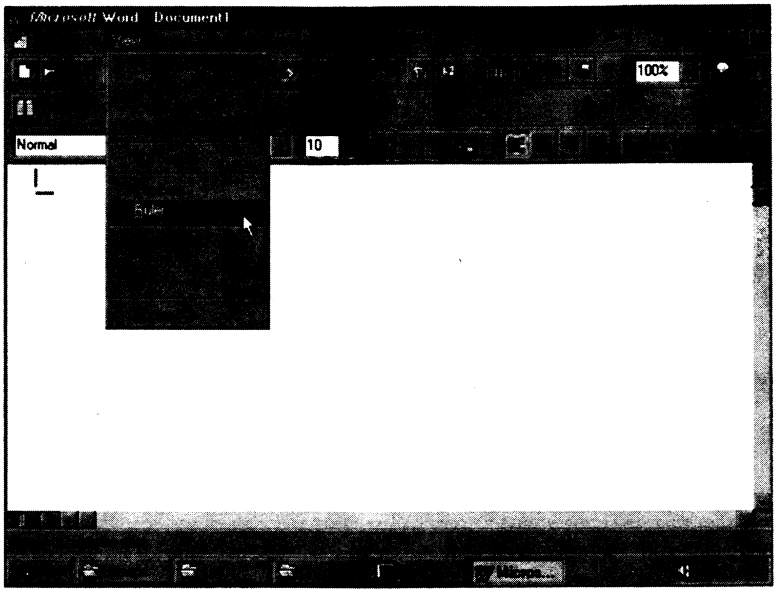


图 6-19　标尺消失了，同时 Ruler 菜单项旁边也不见了核选符号

6.5.1　缺省菜单项

Win95 对缺省菜单项的概念进行延展。自从 Windows 1.x 版本以来，缺省菜单项特性就一直是“保留节目”。假如对系统菜单图标进行双击，就相当于选中了其中的 Close 选项。微软保留了这种动作（鼠标双击），并在下拉式菜单里以黑体字显示某个选项，这就是所谓的缺省菜单项。对每个下拉式菜单来说，我们都可以利用 SetMenuDefaultItem（）这个 API 函数为其中的某个菜单项分配缺省属性：

```
#include〈winuser.h〉
BOOL SetMenuDefaultItem（HMENU hmenu，UINT idDefault，UINT fByPos）；
```

参数	说明
HMENU hmenu	菜单句柄
UINT idDefault	菜单项 ID 或者菜单项的位置
UINT fByPos	假如为 TRUE，就表明 idDefault 是一个菜单项位置；假如为 FALSE，则表明它是一个菜单项 ID
返回值	在正文里讨论
BOOL	布尔值，假如函数调用成功，就返回一个 TRUE；假如失败则返回 FALSE

如果想剔除缺省属性，同时不为其他任何一个菜单项分配该属性，我们可以把 idDefault 设置成-1。GetMenuDefaultItem（）函数的作用是取回当前设置了 MFS_DEFAULT 属性的菜单项的 ID，该函数的语法结构如下所示：

```
#include <winuser.h>
UINT WINAPI GetMenuDefaultItem (HMENU hMenu,
                                UINT fByPos,
                                UINT gmdiFlags);
```

参数	说明
HMENU hMenu	菜单句柄
UINT fByPos	假如为 FALSE，表示返回菜单项标识符；为 TRUE，则返回菜单项的位置
UINT gmdiFlags	为零、GMDI_GOINTOPOPUPS 或者 GMDI_USEDISABLED
返回值	在正文里讨论
UINT	假如函数调用成功，就返回一个 TRUE；假如失败，则返回一个 FALSE 值

这个函数能够返回带有已设置了 MFS_DEFAULT 标志的一个菜单项 ID；假如没有菜单被设置成缺省状态，则返回一个-1 值。GMDI_ USEDISABLED 的作用是指示 GetMenuDefaultItem () 返回一个菜单项——即使该菜单项已处于非活动状态，也照样返回无误。而 GMDI_GOINTOPOPUPS 的作用则是强迫这个函数对联系在一起的弹出式菜单进行搜索。

6.5.2 在运行期间建立一个菜单

正如我们预期的那样，Win32 提供了两种可选的方法在运行期间建立一个菜单项，同时不必使用资源文件里的菜单模板。第一种方法相当复杂，其中涉及到直接在内存里定义一个菜单模板资源，然后通过调用 LoadMenuIndirect () 把它转换成实际的资源。这种技术的细节将在第十五章“Windows 高级技术”里进行讨论。第二种方法——也是最简单的一种方法需要先调用 CreateMenu ()，然后通过 InsertMenuItemInfo () 来实施这种资源，细节性的介绍请大家继续阅读下面的段落。

CreateMenu () 将返回与一个空对象有关的菜单句柄：

```
hmenu = CreateMenu ();
```

调用了这个函数以后，接着插入第一个顶级菜单，然后定义与之相关的下拉式菜单，以后对剩余的顶级菜单重复相同的处理。我们另外还可以采用一种更简单、更直接的方式对这种算法进行编码。内存资源内的信息很容易就可以通过 SetMenuItemInfo () 进行更改。这样一来，我们就可以先在空菜单里填入一系列正文串，稍后再为它们分配 MFS_POPUP 属性。如图 6-20 所示，CRMENU 证明了这种技术最终的结果。该程序的源代码可以在本书附带 CD 的 Listing 6.5 里找到。

6.6 弹出式菜单

在 Windows 95 里，在某个感兴趣的对象上方单击鼠标右键是一种很常见的行动。这种行动的后果往往是正好在鼠标单击的那个地方显示了一个弹出式菜单（关联菜单）。弹出式菜单的最后一项与某个系统外壳对象（MyComputer，Network Neighborhood，Recycle Bin 及其

他对象）有关，这一项通常都标记为 Properties（属性），用户选择它就可以看到对象的属性表（如图 6-21 所示）。

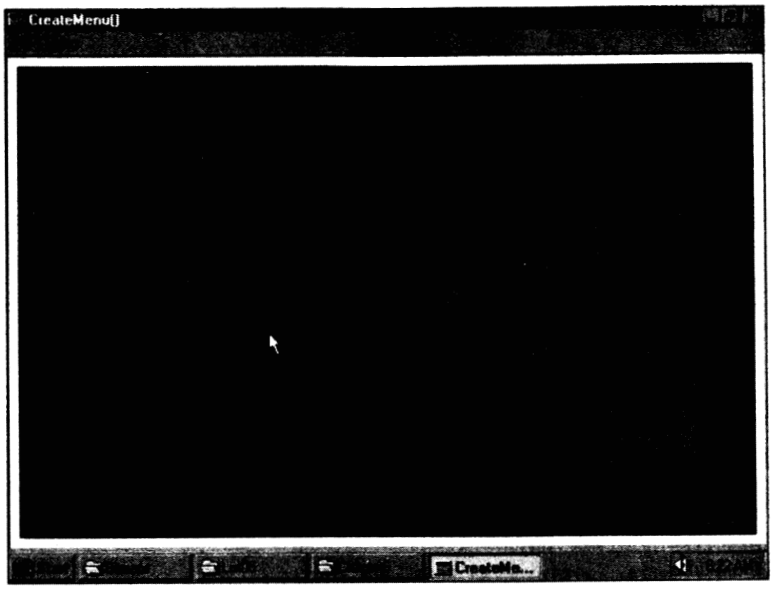


图 6-20　CRMENU 提供了两个顶级菜单和四个下拉式菜单——都是在运行期间建立的

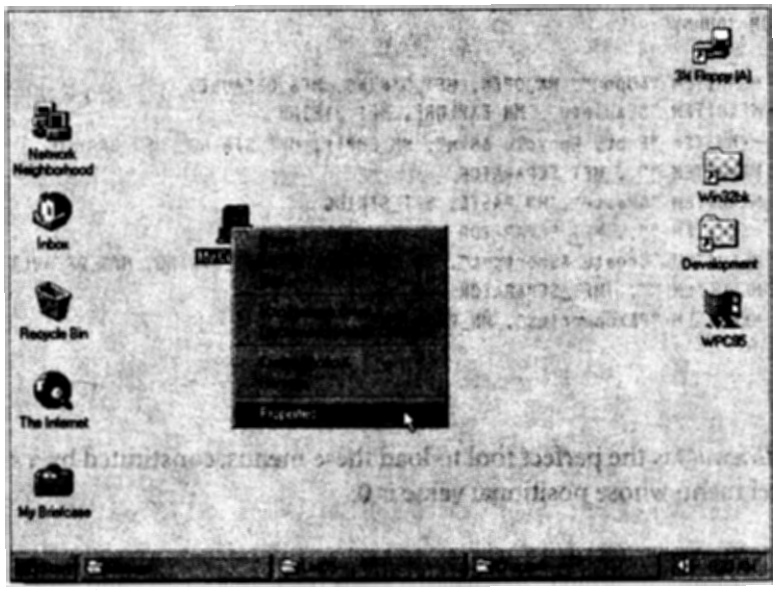


图 6-21　在 My Computer 的弹出式菜单里，Properties 选项已经作好了让用户选择的准备

这个菜单只不过是一种标准的下拉式菜单而已，只是缺省一个顶级菜单——在 Win32 里通过 GetSubMenu（）这个 API 很容易就可以实现的一种进程。对于关联菜单来说，最佳的方式是在 .RC 文件里建立它们——当作独立的 MENU 或者 MENUEX 资源进行处理。不幸的是，MENUEX 语法要求开发者必须用相关的顶级菜单来生成一个弹出式菜单。当然，载入了菜单资源后，可以在运行期间动态地删除这些顶级菜单。

在下面这个代码段里，大家可以看出与 My Computer 和 Recycle Bin 相关的模板在外壳资源文件里是如何定义的：

```
...
MyComputer MENUEX
{
    POPUP "DUMMY"
    {
        MENUITEM "&Open", MN_OPEN, MFT_STRING, MFS_DEFAULT
        MENUITEM "&Explore", MN_EXPLORE, MFT_STRING
        MENUITEM "&Find...", MN_FIND, MFT_STRING
        MENUITEM "",,MFT_SEPARATOR
        MENUITEM "Map &Network Drive...", MN_MAPNTW,
        MFT_STRING
        MENUITEM "&Disconnect Network Drive...", MN_DISCNTW,
        MFT_STRING
        MENUITEM "",,MFT_SEPARATOR
        MENUITEM "Create &Shortcut", MN_CREATESHRT,
        MFT_STRING,MFS_DEFAULT
        MENUITEM "Rena&me", MN_REANME, MFT_STRING
        MENUITEM "",,MFT_SEPARATOR
        MENUITEM "P&roperties", MN_PROP, MFT_STRING
    }
}
...
Recycle MENUEX
{
    POPUP "DUMMY"
    {
        MENUITEM "Open", MN_OPEN, MFT_STRING,
        MFS_DEFAULT
        MENUITEM "&Explore", MN_EXPLORE, MFT_STRING
        MENUITEM "Empty Recycle &Bin", MN_EMPTY, MFT_STRING,
        MFS_GRAYED
```

```
        MENUITEM "",,MFT_SEPARATOR
        MENUITEM "&Paste", MN_PASTE, MFT_STRING
        MENUITEM "",,MFT_SEPARATOR
        MENUITEM "Create &Shortcut", MN_CREATESHRT, MFT_STRING
        MFS_DEFAULT
        MENUITEM "",,MFT_SEPARATOR
        MENUITEM "P&roperties", MN_PROP, MFT_STRING
    }
}
...
```

LoadMenu()是装载这些菜单的最佳手段,这些菜单是由一个顶级菜单构成的,它的位置值是 0:

```
...
hmenu = LoadMenu(hInstance, "MYCOMPUTER");
hmenu = GetSubMenu(hmenu 0);
...
```

假如在 LoadMenu()后面紧接着使用一个 GetSubMenu()函数,这样就可以在 hmenu 标识符内存储与下拉式菜单相关的句柄——准备把这个菜单当作关联菜单显 示出来。假如关联菜单没有对自己的内容进行变动,就可以把前面提到的两个语句放置到 WM_CREATE 消息代码块内。这种思路的前提是应用程序在运行期间要多次调用这个消息块。作为另外一种选择,我们也可以在窗口进程接收到一条 WM_CONTEXTMENU 消息的时候载入对应的关联菜单。对于 WM_CONTEXTMENU 来说,它是严格按照某种顺序生成的,必须位于 WM_RBUTTONDOWN 和 WM_RBUTTONUP 消息的后面。

```
...
case WM_CONTEXTMENU:
{
    HMENU hmenu;

    hmenu = LoadMenu(hInstance, "MYCOMPUTER");
    hmenu = GetSubMenu(hmenu, 0);;
    ...
}
    break;
...
```

我们需要解决的第二个问题是如何计算出关联菜单的正确显示位置。通常情况下,这个菜单应该在显示于光标指针当时在屏幕上的位置处。为了计算这个位置,首先需要对一条 WM_

CONTEXTMENU 消息里的 lParam 进行分析，如下所示：

WM_CONTEXTMENU	0x007B
wParam	用户按下鼠标右键的那个窗口的句柄
LOWORD(lParam)	鼠标位置在屏幕内的 X 坐标
HIWORD(lParam)	鼠标位置在屏幕内的 Y 坐标

光标位置是用屏幕坐标系统表示的(也就是说，与屏幕左上角位置有关)。光标位置正是我们希望显示菜单的位置，请记住，这种类型的对象是从桌面获得它的像素值的。

现在只剩下一个步骤了——把菜单显示出来，这可以通过调用 TrackPopupMenu()来实现：

```
#include 〈winuser.h〉
BOOL WINAPI TrackPopupMenu(HMENU hmenu,
                           UINT fuFlags,
                           int x,
                           int y,
                           int nReserved,
                           HWND hwnd,
                           LPRECT lprc);
```

参数	说明
HMENU hmenu	指定准备显示的菜单
UINT fuFlags	一个或者多个标志，用于指定菜单在屏幕中的位置以及对鼠标事件进行跟踪捕获
int x	屏幕坐标内的水平位置
int y	屏幕坐标内的垂直位置
int nReserved	一般设置为 0
HWND hwnd	指定用于接收菜单生成的所有消息的那个窗口
LPRECT lprc	一个 RECT 数据结构的地址，其中包含了一个特定的屏幕区域，用户可以在其中选择一个菜单项
返回值	在正文里讨论
BOOL	布尔值，假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

x 和 y 参数是指菜单在屏幕上的位置。这个点真正的含义要受到 fuFlags 参数内设置的 TPM_标志的影响。对这些 TPM_标志的总结请读者参考表 6-7 和表 6-8。

任何外壳显示的关联菜单都是用 TPM_TOPALIGN 和 TPM_RIGHTBUTTON 标志建立的。假如设置了后面那个标志，用户既可以使用鼠标右键，也可以使用左键来选择某个菜单项。

用户选中了一个菜单项后，有几条消息都会到达应用程序的窗口进程。在这些消息中，有一条名为 WM_CAPTURECHANGED 的消息引起了我们的注意。这是一种新的 Win95 消

息，它的作用是通知目标窗口对鼠标事件的捕获已经释放了。这就证明了屏幕显示出一个弹出式菜单之前，TrackPopupMenu()函数已经捕获了鼠标事件，并且就在消隐菜单之前释放了这种事件。

表 6-7　影响菜单在屏幕内显示状况的 TrackPopupMenu()标志

标　志	值	说　明
TPM_LEFTALIGN	0x0000L	左边框根据 x 参数里指定的值进行排列
TPM_TOPALIGN	0x0000L	关联菜单正好从鼠标指针所在位置开始向下显示
TPM_HORIZONTAL	0x0000L	水平排列
TPM_CENTERALIGN	0x0004L	使弹出式菜单沿着水平方向居中排列，具体的位置与 x 和 y 参数指定的指针有关
TPM_RIGHTALIGN	0x0008L	右边框根据 x 参数里指定的值进行排列
TPM_VCENTERALIGN	0x0010L	使弹出式菜单沿着垂直方向居中排列，具体的位置与 x 和 y 参数指定的指针有关
TPM_BOTTOMALIGN	0x0020L	关联菜单正好从鼠标指针所在位置开始向上显示
TPM_VERTICAL	0x0004L	垂直排列
TPM_NONOTIFY	0x0080L	不发送任何通知消息

表 6-8　影响菜单在屏幕内显示状况的 TrackPopupMenu()标志

标　志	值	说　明
TPM_LEFTBOTTON	0x0000	用鼠标左键选择一个菜单项
TPM_RIGHTBUTTON	0x0002	用鼠标右键选择一个菜单项

如图 6-22 所示，这幅图向大家展示了 Trckmenu 示例程序，这个程序的源代码可以在本书附带 CD 的 Listing 6.6 里找到。假如我们在程序启动以后立即单击鼠标右键，那么在程序里不会发生任何事情，也没有任何反应。

在这个程序里，Colors 顶级菜单允许用户为应用程序客户区分配一种预先定义好的颜色。选择这个菜单后，假如又在应用程序内单击鼠标右键，便可直接利用弹出式菜单对窗口的背景颜色进行变动，如图 6-23 所示。

弹出式菜单正好显示于鼠标指针热点的下方，并且引用了最近一次显示的下拉式菜单(该菜单与 Colors 或者 File 是关联在一起的)。为了捕获最近的一个下拉式菜单句柄，TRCKMENU 程序拦截了 WM_INITMENUPOPUP 消息，这对于我们来说应该不再显得陌生了。每次选择了资源文件里用 POPUP 命令定义的一个菜单项后，就会生成这条消息。正如我们以前提到过的那样，wParam 内包含了弹出式菜单的句柄，而 lParam 则封装了一个 32 位值的菜单项位置以及一个设置为 TRUE(前提是引用了系统菜单)的标志。TRCKMENU 把菜单句柄存储在一个静态标识符内，使其在后续的消息里亦可使用：

```
...
case WM_INITMENUPOPUP:
{
    hmenu = wParam;
}
```

```
        break;
...
```

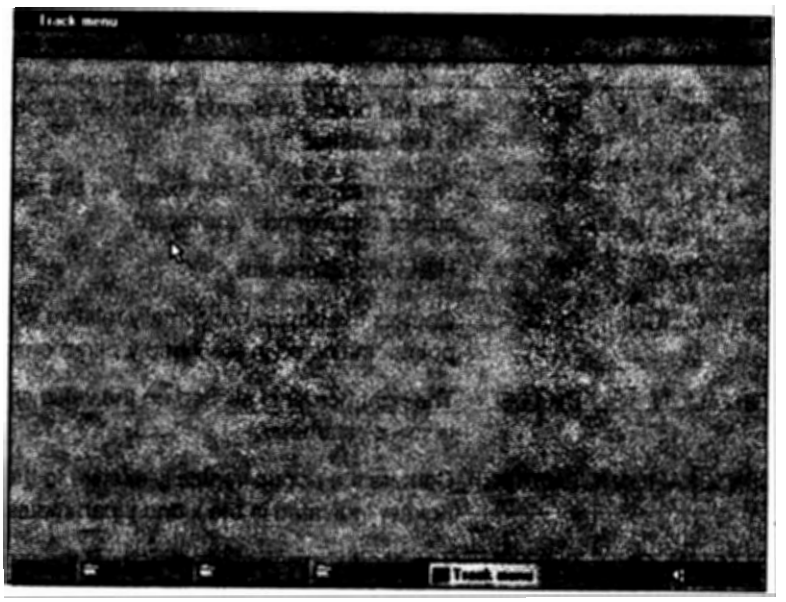


图 6-22　TRCKMENU 在客户区重画完成之前，不会对鼠标右键的单击事件作出响应

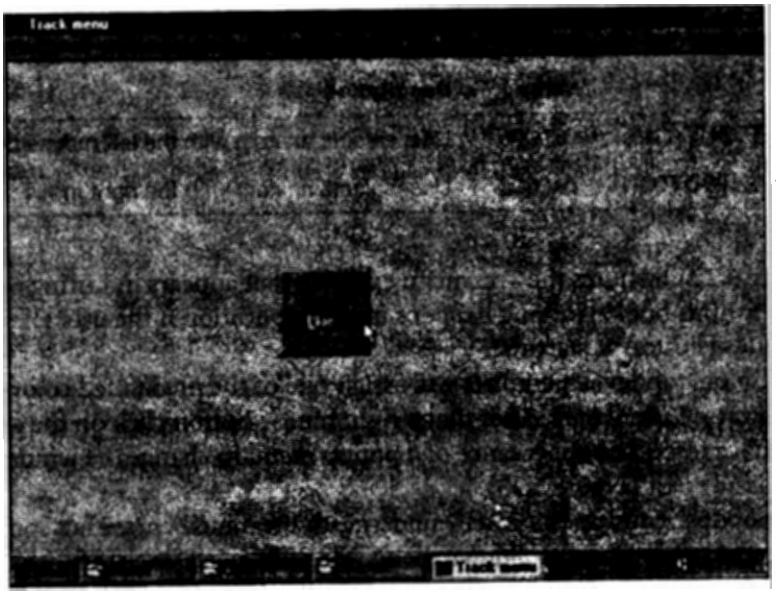


图 6-23　TRCKMENU 显示了与应用程序客户区联系起来的
一个弹出式菜单——用于改变背景颜色

应用程序启动以后，标识符会自动设置成 NULL（编译器分配给所有非初始化静态标识

符的一种标准值),并且证实是否有一个弹出式菜单存在。当一条 WM_CONTEXTMENU 消息到达窗口进程以后,程序就会从 lParam 里取回鼠标指针在屏幕上的当前位置,这是通过用适当信息来调用 TrackPopupMenu()函数实现的:

```
...
case WM_CONTEXTMENU:
{
    POINT pt;

    pt.x = MAKEPOINT(lParam).x;
    pt.y = MAKEPOINT(lParam).y;

    // display popup menu
    TrackPopupMenu(hmenu,
                   TPM_CENTERALIGN | TPM_RIGHTBUTTON,
                   pt.x,
                   pt.y,
                   0,
                   hwnd,
                   NULL);
}
...
```

在这个代码段里,我们增加了 TPM_RIGHTBUTTON 标志,从而使鼠标右键也有能力在系统外壳里选择一个菜单项。

6.7 把位图用作菜单项

对单调和阴暗的正文串不感兴趣了吗?好了,为什么不用位图来代替它们呢?对于那些艺术家来说,这也许才是一种明智的决定。不幸的是,对于我们这些程序员来说,作出这种决定却并非很恰当。我们在这儿一定要理解自己准备做的事情是在一个菜单项里放置一幅位图,而不是在西斯廷教堂里描绘一幅艺术作品。所以,我把自己在这方面的精力限制在只描绘红色、绿色和蓝色的位图上面。

Menubmp 这个示例程序的目标是在一个下拉式菜单里放置一些位图,而不是传统的正文串(该例子可在本书附带 CD 的 Listing 6.7 里找到)。正如大家事先或许已经预料到的那样,我画的这个下拉式菜单看起来并不是那么美观,如图 6-24 所示。然而它的确证明了可以把位图当作菜单项来使用。

该应用程序的 .RC 文件里包含了下面这些资源说明:

```
...
Red BITMAP "red.bmp"
```

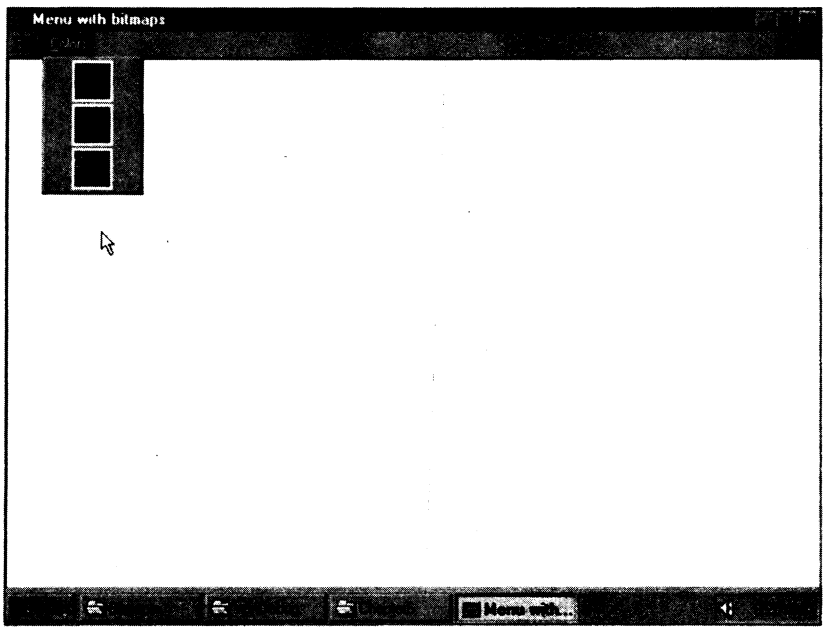


图 6-24　MENUBMP 程序的下拉式菜单里提供了位图，而不是传统的正文

```
Green BITMAP "green.bmp"
Blue BITMAP "blue.bmp"
...
```

当该应用程序通过调用 LoadBitmap()拦截了 WM_CREATE 消息以后，就会载入三幅位图：

```
...
case WM_CREATE:
{
    int i;
    HBITMAP hbmp;
    HMENU hmenu = ((LPCREATESTRUCT)lParam) -> hMenu;

    hInstance = ((LPCREATESTRUCT)lParam) -> hInstance;

    // loading the bitmaps
    for(i = 0; i < MAXCOLORS; ++i)
    {
        hbmp = LoadBitmap(hInstance, items[i].pszRes);
        ...
    }
```

```
}
    break;
...
```

资源文件里包含了一个 MENU 资源，它用于对所有的下拉式菜单进行描述，其中包括我们准备在其中显示位图的那个菜单。正如大家在下面这个代码段里看到的那样，Colors 下拉式菜单是用三个正文串来定义的。这是 MENU 资源支持的唯一语法规则，相同的规则亦可应用于 MENUEX 资源。

```
MAINMENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New", MN_NEW
        MENUITEM SEPARATOR
        MENUITEM "E&xit", MN_EXIT
    END
    POPUP "&Colors"
    BEGIN
        MENUITEM "&Red", MN_RED
        MENUITEM "&Green", MN_GREEN
        MENUITEM "&Blue", MN_BLUE
    END
END
```

因此，一旦启动应用程序后，我们的策略就是从资源文件里载入位图，然后用它们取代现成的字串。由 RESOURCE.H 定义的 ITEM 结构里容纳了菜单项的 ID，以及指向每个菜单项内所显示正文串的一个指针。

```
...
static struct_ITEMS
{
    WORD wMenuID;
    PSTR pszRes;
} items[] = { MN_RED, "Red",
              MN_GREEN, "Green",
              MN_BLUE, "Blue" };
...
```

对字串进行替换的任务可以由 SetMenuItemInfo()或者 ModifyMenu()来完成。下面这个代码段引用了 ModifyMenu(),其中的最后一个参数与某幅位图的句柄对应:

```
...
ModifyMenu(hmenu,
           items[i].wMenuID,
           MF_BITMAP,
           items[i].wMenuID,
           (LPSTR)(LONG)hbmp);
...
```

用户选择一个位图化的菜单项时,应用程序里没有发生任何变化。我们仍然和往常一样接收到了一条 WM_COMMAND 消息,其中 wParam 的低字内包含了菜单项的 ID。

6.8 物主绘图菜单

物主绘图菜单提供了设计定制菜单项的一种新方式。在这儿,"物主"这个术语明确地表示了这种菜单的作用。

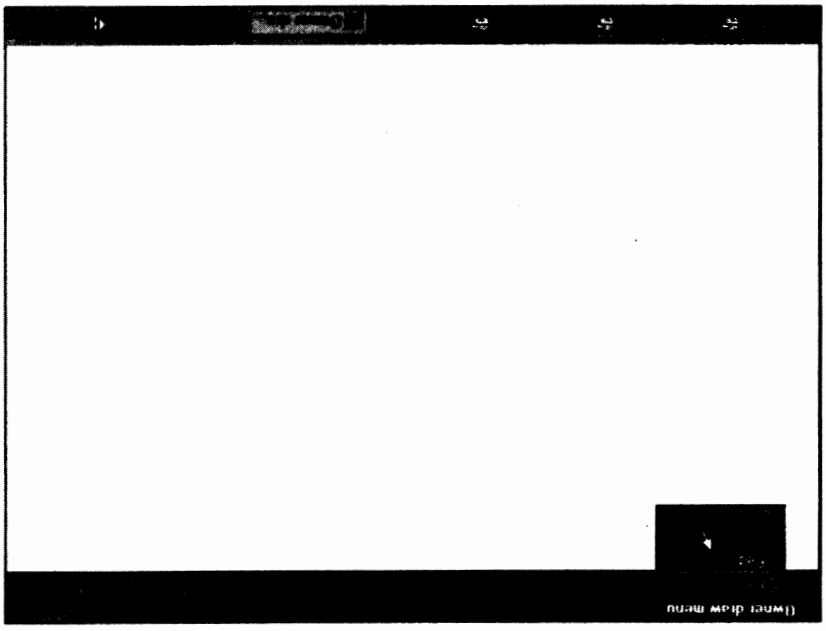


图 6-25 Colors 下拉式菜单内包含了三个物主绘图菜单项,
每一项都是由正文和位图组合起来的

菜单是一些独立的和非文档化的窗口,它的几乎每一项功能都是在 Win32 内部进行控制的。不管微软的工程师们是如何对这些功能进行编码的,他们都要遵循一些环境规范,这些规范涉及到诸如重画和输出生成之类的活动。对于一个物主菜单项来说,它的作用很简单,仅仅

是把所有输出信息都传递给它的物主——即应用程序窗口进程。这种信息的传递是通过WM_MEASUREITEM 和 WM_DRAWITEM 消息来实现的。假如到了对菜单项输出信息进行刷新的时候，菜单窗口就会调用菜单物主，同时不用介入任何一个输出进程。举个例子来说，设计了一个物主绘图菜单，我们就可以把正文和图象同时合并起来，或者改变标准菜单项的外观。图 6-25 向大家展示了一个名为 Menudraw 的示范程序，这个程序可以在本书附带 CD 的 Listing 6.8 内找到。

我们可以用 MFT_OWNERDRAW 标志在资源模板里把一个扩展菜单直接设置成物主绘图属性。传统的菜单模板没有提供这种特性，所以我们需要对其进行物理处理。在 Menudraw 这个例子里，我们拦截了 WM_CREATE 消息，从而为三个菜单项内的每一个都增加了 MFS_OWNERDRAW 标志：

```
...
for(i = 0; i < MAXCOLORS; ++i)
{
    ModifyMenu(hmenu,
               items[i].wMenuID,
               MFS_OWNERDRAW,
               items[i].wMenuID,
               NULL);
}
...
```

MFT_OWNERDRAW 标志(或者 MF_OWNERDRAW)的作用是指示一个菜单项把所有输出信息都传递给对应的物主窗口，这种传递是分两步完成的。建立一个控件的时候，就会生成一条 WM_MEASUREITEM 消息。在这儿，“控件”这个术语用得并非百分之百准确，因为它也能引用其他类型的预定义窗口类，比如 BUTTON(按钮)，COMBOBOX(组合框)以及 LISTBOX(列表框)等等。注意，关于列表框的问题，我们会在第九章“预定义的窗口类”里进行讨论。对于一个物主绘图菜单来说，它意味着当下拉式菜单首次在屏幕上显示出来的时候，针对带有 MSF_OWNERDRAW 属性每个菜单项，物主窗口都会接收到一条对应的 WM_MEASUREITEM 消息。WM_MEASUREITEM 消息的行为就好象一条初始化消息，它可以把某些有价值的信息传递给物主窗口。

WM_MEEASUREITEM	0x002C
wParam	假如消息是从一个菜单项内发出的，那么肯定设置为 0
lParam	一个 MEASUREITEMSTRUCT 数据结构的地址

这种数据结构内包含了用于某个菜单项有限值的有限信息。从根本上说，其中唯一有效的数据是菜单项 ID，通过它可以知道这条消息当前引用的是哪个菜单项。对 WM_MEASUREITEM 消息进行处理的时候，应用程序把新菜单项的尺寸存储到 itemWidth 和

itemHeight 项内。这个操作是根据程序准备显示的内容来完成的。这就意味着物主绘图菜单项在垂直轴上并不局限于标准的尺寸大小。

```
typedef struct tagMEASUREITEMSTRUCT
{     //mis
    UINT CtlType;
    UINT CtlID;
    UINT itemID;
    UINT itemWidth;
    UINT itemHeight;
    DWORD itemData;
} MEASUREITEMSTRUCT;
```

Menudraw 程序把菜单项 ID 存储在 ITEMS 数据结构内，这个结构是在 RESOURCH.H 里定义的。随后，它会调用 GetTextExtentPoint32()函数，从而计算出正文的宽度。同时，假如菜单项的高度小于位图的高度，它也会对前者进行调整。在离开这条消息之前，程序会按照计算出来的尺寸分别对 itemWidth 和 itemHeight 项进行设置：

```
...
case WM_MEASUREITEM:
{
    LPMEASUREITEMSTRUCT lpmi=(LPMEASUREITEMSTRUCT)lParam;
    HDC hdc;
    DWORD dwCheckExt;
    struct _ITEMS *lpit;
    SIZE siz;

    // incrementing the pointer to the info
    lpit = lpitems + lpmi -> itemID - START;

    // retrieving the text dimensions
    hdc = GetDC(hwnd);
    GetTextExtentPoint32(hdc,
                         lpit -> pszRes,
                         lstrlen(lpit -> pszRes),
                         &siz);
    ReleaseDC(hwnd, hdc);

    dwCheckExt = GetMenuCheckMarkDimensions();
```

```
        if(siz.cy < HIWORD(dwCheckExt))
            siz.cy = HIWORD(dwCheckExt);

        siz.cx += LOWORD(dwCheckExt) + siz.cy * 2;

        lpmi -> itemWidth = siz.cx;
        lpmi -> itemHeight = siz.cy;
    }
        break;
...
```

每次在屏幕上显示下拉式菜单的时候，应用程序就会接收到一条 WM_DRAWITEM 消息。菜单管理模块把所有信息都封装到一个 DRAWITEMSTRUCT 结构里，并且把 WM_DRAWITEM 消息传送给物主窗口，同时等待窗口重画完成。和往常一样，WM_DRAWITEM 引用一个菜单项的时候，wParam 里不会包含任何有价值的信息。

WM_DRAWITEM	0x002B
wParam	对于菜单来说肯定为 00
lParam	一个 DRAWITEMSTRUCT 数据结构的地址

其中，DRAWITEMSTRUCT 结构里包含了许多有用的信息，这些信息是与当时一个单独菜单项的输出活动有关的。

```
typedef struct tagDRAWITEMSTRUCT
 {     //dis
    UINT CtlType;
    UINT CtlID;
    UINT itemID;
    UINT itemAction;
    UINT itemState;
    HWND hwndItem;
    HDC hDC;
    RECT rcItem;
    DWORD itemData;
} DRAWITEMSTRUCT;
```

Win32 利用这种消息和这种相同的数据结构为系统内支持的每个物主绘图对象提供输出信息。在传递的所有这些信息中，省略了一种非常明显的东西：菜单项的正文。MENUDRAW

把这种省略的信息存储在整个窗口进程内均可使用的一个静态数组内。

```
...
static struct _ITEMS
 {
    WORD wMenuID;
    PSTR pszRes;
    long iCr;
} items [] = { MN_RED, " Red", CL_RED,
                MN_GREEN, " Green", CL_GREEN,
                MN_BLUE, " Blue", CL_BLUE}, FAR *lpitems = items;
...
```

为了解决这个问题，我们有一种方案可供选择。这时，我们可以使用 ModifyMenu () 函数，用它的最后一个参数来接收普通的 32 位值。在这个参数里，我们可以放置指向一个正文串的指针或者一个内存位置的地址。这个内存位置位于应用程序的整个地址空间以内，其中包含了所有需要的数据。通过使用 WM_MEASUREITEM 或者 WM_DRAWITEM 消息，我们很容易便可对这个专用的内存区域进行访问。无论 MEASUREITEM 还是 WM_DRAWITEM 消息，它们都提供了名为 itemData 的一个项，这个项的内存是由开发者自己定义的。itemData 项可以指向以前随同 ModifyMenu () 传递的值。

用户选择三种基本颜色的其中之一时，整个客户区域都会进行相应的重画。在这个时候，选中的菜单项的输出与其他两个仍然是不同的。一个标准的核选符会在颜色正文旁边显示出来，如图 6-26所示。

图 6-26　选定的颜色菜单项带有一个核选符号

6.9 加速键的实施

现在让我们来观察一下 Menu 程序示例，这个程序可在本书附带 CD 的 Listing6.1 内找到。几乎每个菜单项都提供了对应的加速键组合，尽管它们之中的任何一个都不会起作用。加速键是 80 年代中期由 Windows 的第一个版本引入的，它的作用是帮助熟练用户加快菜单选择。加速键的本质是用两个按键的组合来模拟对下拉式菜单内一个菜单项的选择；一个按键或三个按键组合构成加速键的情况则很少见。

为了让一张加速键表格产生作用，我们需要采取三个步骤：

▶在 .RC 文件里建立一个 ACCELERATORS 资源

▶把加速键表格载入应用程序内存

▶修改应用程序消息循环

除此以外，我们还需要直接在菜单项正文串里列出对应的加速键，关于这一点，大家可以从 Menu 例子内看出（请参考本书附带 CD 的 Listing 6.1）。一个 ACCELERATORS 资源定义了某些键盘按键与一个菜单项之间的链接关系——按键与菜单是两个完全不同的概念。下面是一张 ACCELERATORS 表格的样子：

```
accelID ACCELERATORS
{
    "character", ID [.NOINVERT][,CONTROL][,SHIFT][.ALT]
    "^ character", ID [,NOINVERT][,SHIFT][,ALT]
    ASCII code, ID, ASCII [,NOINVERT][,CONTROL][,SHIFT][,ALT]
    Virtual Key, ID, VIRKEY [,NOINVERT][,CONTROL][,SHIFT][,ALT]
}
```

这种语法结构尽管看起来相当复杂，然而这只是一种初步的印象而已。对于上面那四种加速键的定义方式来说，每一种的关键组件都是分配给 MENUEX 资源内菜单项的一个 ID。通常情况下，加速键组合是由一个控制键（Alt，Ctrl 或者 Shift）以及另外一个按键（ASCII 集、虚拟键或者控制键）组成的。在这儿，我们亦可把控制键叫作“死键”，单独按下这样的一个键是没有用处的。Alt+F4 也许是我们最熟悉的一个加速键了，Ctrl+S 也广泛运用于存储当前的活动文档。

假设我们现在希望定义这样的一些加速键：Alt+F1，Shift+Alt+Ctrl+A 以及 Ctrl+C，分别把它们分配给 Open，Save 和 Close 命令。这时的 ACCELERATORS 表格看起来就应像下面这个样子：

```
myacceltable ACCELERATORS
 {
    VK_F1, MN_OPEN, VIIRTKEY, ALT
    " A", MN_SAVE, VIRTKEY, CONTROL, ALT
    " ^C", MN_CLOSE
```

}

其中，VIRTKEY（虚拟键）类型是唯一能和 Alt，Shift 以及 Ctrl 键组合起来使用的。ASCII 类型的影响有限，它们只能用于定义一个单一的字符加速键，所以它的应用场合并不多。

现在让我们先来看一看第一个加速键 Alt+F1：

VK_F1，MN_OPEN，VIIRTKEY，ALT

其中，F1 键应该定义成虚拟键。由于 F1 不属于 ASCII 集的一部分，所以我们必须设置 VIRTKEY 类型。Alt 选项则是这种语法里的最后一个参数。

对于非常复杂的 Shift+Alt+Ctrl+A 来说，它的语法结构是下面这种样子：

" A"，MN_SAVE，VIRTKEY，CONTROL，ALT

其中，封闭在引号内的字母必须采用大写形式，这就表示无论用户通过键盘输入的是大写字母还是小写字母，都可以实现事先定义好的加速键功能。其中的 VIRTKEY 用于支持 Ctrl 和 Alt 选择键。第三种加速键组合是按照一种很老的规范实现的，这种规范以 WordStar 字处理软件为基础。在这种规范里，由 Ctrl 键代表的命令是用屏幕上一个尖号（^）表示的。NOINVERT 选项已经非常陈旧了。假如设置这个选项，就能防止顶级菜单用反转色显示一小段时间——以此表明已经捕获到了加速键。

解决了与 ACCELERATORS 资源有关的所有问题以后，我们现在可以集中精力学习如何把它载入应用程序。我们将通过 LoadAccelerators() 函数从源代码内对这种信息进行访问，对于这种方式我们不应感奇怪：

```
#include〈winuser.h〉
HACCEL WINAPI LoadAccelerators (HINSTANCE hInstance,
                                LPCSTR lpszTableName);
```

参数	说明
HINSTANCE hInstance	应用程序实例句柄
LPCSTR lpszTableName	加速键资源标签
返回值	在正文里讨论
HACCEL	加速键表；假如函数调用失败，返回的则是一个 NULL 值

这个函数可以返回一个资源句柄——在 WinMain() 里定义的一种为 HACCEL 类型的标识符：

```
HACCEL haccel;
```

考虑到 ACCELERATORS 表格的存在，所以我们需要对应用程序的消息循环进行变动。这种操作应该发生于 LoadAccelerators () 以后，它是 WinMain () 里需要采取的首批操作的其中之一。在消息循环里，我们应该在调用 TranslateMessage () 和 DispatchMessage () 之前先对 TranslateAccelerator () 进行调用。TranslateAccelerator () 会检查 GetMessage () 获取的消息是否与定义好的加速键组合的其中之一匹配。这样就能在消息循环的传统处理进行之前验证加速键是否按下。

```
#include〈winuser.h〉
```

```
int TranslateAccelerator (HWND hwnd,
                          HACCEL haccel,
                          LPMSG lpmsg);
```

参数	说明
HWND hwnd	窗口句柄
HACCEL haccel	加速键表的句柄
LPMSG lpmsg	一个 MSG 数据结构的地址
返回值	在正文里讨论
int	假如接收到的消息引用的是加速键，就返回一个 TRUE 值；否则返回 FALSE

对 TranslateAccelerator () 会向由第一个参数指定的窗口进程发送一条 WM_COMMAND 消息，从而绕过消息循环剩余的部分。对于加速键来说，有个问题在于这些加速键是与特定的窗口对应的，由 TranslateAccelerator () 的语法我们就可以看出这一点。假如一个应用程序希望安装与不同窗口有关的多个 ACCELERATORS 资源，那么消息循环就会变得发生信息拥挤的情况。所以我们这时要采取另外一种解决方案，这种方案要求使用 WinMain () 里的几个窗口句柄，藉此强制性地把它们声明为源文件标识符。

ACCEL 示范程序（参考本书附带 CD 的 Listing 6.9）实现在一个标准 Win32应用程序里对几个加速键的支持。客户区内显示的图象可以通过按下 Ctrl+I 组合键使其消隐掉。

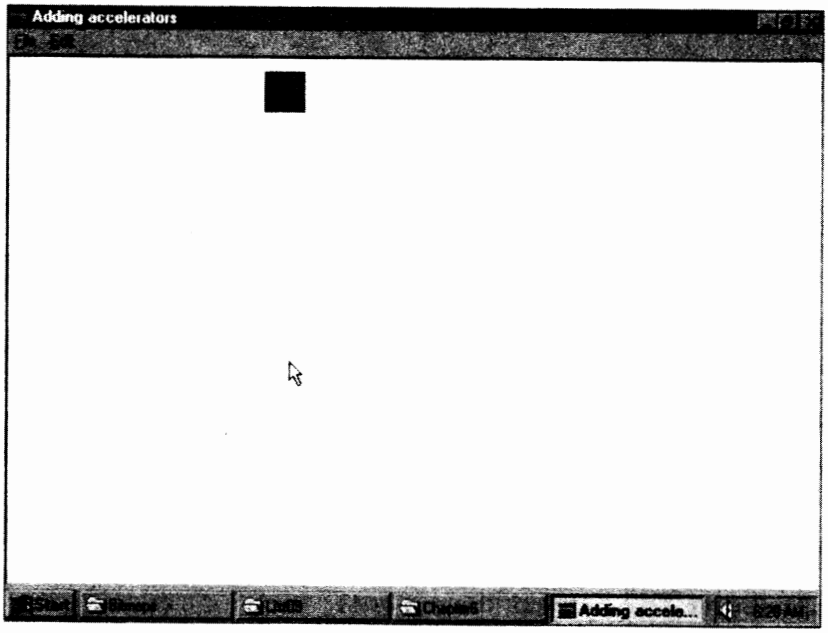


图 6-27　ACCEL 在应用程序客户区内显示了一幅位图

不管我们早些时候说过什么，事实上，加速键与菜单项之间并非具有很严格的联系。尽管

如此，加速键与菜单仍然具有密切的关系，尽管这并不是必须的。假如在应用程序消息循环里捕获了一个加速键，它就会转换到一条 WM_COMMAND 消息内，在其中 wParam 的低字内包含了相关菜单项的 ID。没有什么成文的规定限制我们在没有对应菜单项的前提下，不能在 RESOURCH. H 里定义一个常规的 ID。一旦我们有了一个 ID，就可以定义一个加速键。同时，应用程序最终也会接收到包含了那个 ID 的 WM_COMMAND 消息。

除此以外，Win32现在还提供了一个名为 CreateAcceleratorTable () 的 API 函数，它可以在程序运行期间在内存里建立一个加速键资源。

6.10 热键特性

在本章快结束的时候，我们准备介绍由 Win32实现的一种新特性——热键。热键的行为类似于加速键，但它们更好、也更容易实现，因为这种热键不需要任何资源文件的介入，它们完全是驻留于应用程序源代码内的。

到现在为止，大家应该知道如何利用预先分配好的加速键，比如用 Alt+F4来关闭一个窗口，或者自己针对几个菜单项建立一些新的加速键。传统加速键的问题在于对它们的设置太麻烦了。首先，我们必须设置 ACCELERATORS 资源模板，随后还要把它载入内存。还有最糟糕的事情，那就是加速键要求我们对消息循环进行修改，并且在大多数情况下都要与菜单联系起来。

现在假设我们想通过按下一种按键组合，从而激活一个代码段，同时这种按键组合与菜单项是完全无关的。怎样达到我们的目的呢?对了，热键就是最佳的一种方案。热键通常是两个按键的组合，开发者对这种热键使用可以有更广的自由度。我们可以在源代码内设置对 RegisterHotKey () 的一个调用，该函数的语法结构如下所示：

```
#include 〈winuser. h〉
BOOL WINAPI RegisterHotKey (HWND hwnd,
                            int idHotKey,
                            UINT fuModifiers,
                            UINT uVirtKey);
```

参数	说明
HWND hwnd	标识用于接收 WM_HOTKEY 消息的窗口。假如为 NULL，就表明消息被张贴到了应用程序消息队列内
int idHotKey	线程级上唯一的一个热键 ID，范围在0x0000到0xBFFF 之间
UINT fuModifiers	指出三个可应用的标识符中哪个用于和 uVirKey 指定的热键组合起来使用
UINT uVirtKey	虚拟键的代码
返回值	在正文里讨论
BOOL	布尔值，假如函数调用成功，就返回 TRUE；假如失败，则返回一个 FALSE 值

其中，第一个参数是指某个特定的窗口。每次按下热键后都需要用这个窗口的窗口进程来

接收 WM_HOTKEY 消息。假如我们定义了多个热键，并且这些热键均与同一个窗口进程有关，为了对这些热键进行区分，我们可以为它们分配各自不同的一个 ID。这种 ID 的取值范围在0x0000到0xBFFF 之间。ID 信息是通过 WM_HOTKEY 消息的 wParam 进行传递的。最后两个参数定义了实际的热键组合。RegisterHotKey () 这个 API 函数支持表6-9内列出的三种控制键，它们分别与 Alt，Ctrl 以及 Shift 键对应。

表 6-9 用于注册热键的修改键定义

标 志	值	说 明
MOD_ALT	0x0001	Alt 键必须未按下
MOD_CONTROL	0x0002	Ctrl 键必须未按下
MOD_SHIFT	0x0004	Shift 键必须未按下
MOD_WIN	0x0008	Microsoft 自然键盘上的 WIN 键

虚拟键代码是指由 Win32分配给 ASCII 字符集内没有提供支持的几乎所有按键的一个假想数字，具体的定义请参考表6-10。

由此可以得出结论，热键通常——但并不总是一个或者多个控制键与一个虚拟键的组合。显然，我们应该避免定义系统和（或者）应用程序已在使用的按键组合。例如，把 Alt+F4定义成热键看起来就并不是一种明智之举。其实，我们可以选择的按键组合是相当多的。假如三个控制键中没有一个需要用到，那么为第三个参数分配一个0值即可。

表 6-10 WINUSER. H 里对虚拟键的定义

虚拟键	值	虚拟键	值
VK_LBUTTON	0x01	VK_PRINT	0x2A
VK_RBUTTON	0x02	VK_EXECUTE	0x2B
VK_CANCEL	0x03	VK_SNAPSHOT	0x2C
VK_MBUTTON	0x04	VK_INSERT	0x2D
VK_BACK	0x08	VK_DELETE	0x2E
VK_TAB	0x09	VK_HELP	0x2F
VK_CLEAR	0x0C	VK_LWIN	0x5B
VK_RETURN	0x0D	VK_PWIN	0x5C
VK_SHIFT	0x10	VK_APPS	0x5D
VK_CONTROL	0x11	VK_NUMPAD0	0x60
VK_MENU	0x12	VK_NUMPAD1	0x61
VK_PAUSE	0x13	VK_NUMPAD2	0x62
VK_CAPITAL	0x14	VK_NUMPAD3	0x63
VK_ESCAPE	0x1B	VK_NUMPAD4	0x64
VK_SPACE	0x20	VK_NUMPAD5	0x65
VK_PRIOR	0x21	VK_NUMPAD6	0x66
VK_NEXT	0x22	VK_NUMPAD7	0x67
VK_END	0x23	VK_NUMPAD8	0x68
VK_HOME	0x24	VK_NUMPAD9	0x69
VK_LEFT	0x25	VK_MULTIPLY	0x6A
VK_UP	0x26	VK_ADD	0x6B
VK_RIGHT	0x27	VK_SEPARATOR	0x6C
VK_DOWN	0x28	VK_SUBTRACT	0x6D

（续）

虚拟键	值	虚拟键	值
VK_SELECT	0x29	VK_DECIMAL	0x6E
VK_DIVIDE	0x6F	VK_F22	0x85
VK_F1	0x70	VK_F23	0x86
VK_F2	0x71	VK_F24	0x87
VK_F3	0x72	VK_NUMLOCK	0x90
VK_F4	0x73	VK_SCROLL	0x91
VK_F5	0x74	VK_LSHIFT	0xA0
VK_F6	0x75	VK_RSHIFT	0xA1
VK_F7	0x76	VK_LCONTROL	0xA2
VK_F8	0x77	VK_RCONTROL	0xA3
VK_F9	0x78	VK_LMENU	0xA4
VK_F10	0x79	VK_RMENU	0xA5
VK_F11	0x7A	VK_PROCESSKEY	0xE5
VK_F12	0x7B	VK_ATTN	0xF6
VK_F13	0x7C	VK_CRSEL	0xF7
VK_F14	0x7D	VK_EXSEL	0xF8
VK_F15	0x7E	VK_EREOF	0xF9
VK_F16	0x7F	VK_PLAY	0xFA
VK_F17	0x80	VK_ZOOM	0xFB
VK_F18	0x81	VK_NONAME	0xFC
VK_F19	0x82	VK_PA1	0xFD
VK_F20	0x83	VK_OEM_CLEAR	0xFE
VK_F21	0x84		

现在假设我们需要对 Shift＋Backspace 这个热键组合进行注册。这时的代码看起来就应该像下面这个样子：

```
...
RegisterHotKey (hwnd, 0x0010, MOD_SHIFT, VK_BACK);
...
```

按下这个热键以后，应用程序就会在窗口进程里接收到一条 WM_HOTKEY 消息。这个窗口进程与用于分配热键的那个窗口进程是对应的。

```
...
case WM_HOTKEY:
{
    switch (wParam):
    {
        case 0x0010:
          {
```

```
                // process the hot key
            }
                break;
            ...
        }
    }
        break;
    ...
```

RegisterHotKey（）会返回一个布尔值——假如函数调用成功，就返回 TRUE；假如出于某种原因导致了调用的失败，则返回一个 FALSE 值。在中断应用程序之前，应该先通过调用 UnregisterHotKey（）函数删除注册的热键：

```
#include 〈winuser.h〉
BOOL UnregisterHotKey (HWND hwnd, int idHotKey);
```

参数	说明
HWND hwnd	窗口句柄
int idHotKey	热键的 ID
返回值	在正文里讨论
BOOL	假如函数调用成功，返回 TRUE；假如失败，则返回一个 FALSE

6.11 系统菜单

在许多例子里，假如用户双击系统菜单图标，应用程序就会在屏幕中央显示出一个消息框，询问是不是真的想退出。我们怎样对这样的一种功能进行代码编写呢?这种机制要以 WM_SYSCOMMAND 消息为基础才能实现。每次用户选中一个系统菜单的时候，系统就会自动生成一条 WM_SYSCOMMAND 消息。该消息与 WM_COMMAND 类似，然而它所针对的完全是系统菜单，正如通过它的名字就可以推断出的那样。

每个标准的系统菜单都有一个唯一的 ID，这个 ID 是在 WINUSER.H 里定义的。ID 信息将封装到 WM_SYSCOMMAND 消息的 wParam 里，然后直接传送给应用程序的窗口进程。由于 WM_SYSCOMMAND 消息里包含了缺省的选项，亦即对应的 SC_CLOSE 定义（参考表 6-11），所以假如用户双击系统菜单图标，就会转向进行缺省处理（在这儿是退出程序）。

假如用户选中的是标准窗口右上方三个按钮的其中之一，系统就会生成一条 WM_SYSCOMMAND 消息，其中包含了 SC_MINIMIZE，SC_MAXIMIZE 或者 SC_CLOSE ID。下面这个代码段向大家展示了如何对来自系统菜单的信息进行捕获，并且采取相应的行动：

```
...
case WM_SYSCOMMAND:
    switch(wParam)
    {
```

```
            case SC_CLOSE:
            {
                char szText[] = "Do you really want to terminate?";
                char szCaption[] = "Accel";

                if(MessageBox(hwnd, szText, szCaption, MB_YESNO) == IDNO)
                    return FALSE;
            }
                break;

            default:
                break;
        }
        break;
...
```

假如用户在这时按下了消息框内的 No 按钮，API 函数 MessageBox () I 就会返回相应的 IDNO。这个值与 if 语句的测试条件是相符的，所以就会继续执行 if 下面的命令。应用程序使用了 SC_CLOSE 来防止 wParam 里的 WM_SYSCOMMAND 消息到达 DefWindowProc ()，并且防止程序的继续执行。这样一来，应用程序就能防止 WM_SYSCOMMAND 消息到达 DefWindowProc ()，防止了程序的继续执行，也避免了应用程序依然保持活动状态。

假如在 File 菜单里提供了 Exit 菜单项，开发者应该要求用户对自己的“退出”选择进行确认。所以，当拦截了 Exit 菜单项的时候，我们可以模拟一条 WM_SYSCOMMAND 消息的存在，从而执行前面展示过的那部分代码。

```
...
case MN_EXIT :
{
    SendMessage(hwnd, WM_SYSCOMMAND, SC_CLOSE, 0L);
    PostQuitMessage(0 );
}
    break;
...
```

在一个调试程序或者一种开发环境中运行的时候，我们仍然调用 PostQuitMessage () 来确保进行干净和正确的应用程序中断。由于我们采用了这种小范围的变动，所以屏幕中央总会弹出一个消息框，要求用户对退出操作进行确认，如图6-28所示。这和用户在何处中断程序是无关的。

表 6-11　用于标准菜单项的系统菜单 ID

系统菜单标志	值	说　明
SC_SIZE	0xF000	Size 菜单项
SC_MOVE	0xF010	Move 菜单项
SC_MINIMIZE	0xF020	Minimize 菜单项
SC_MAXIMIZE	0xF030	Maximize 菜单项
SC_NEXTWINDOW	0xF040	下一个窗口（在 MDI 文档窗口里）
SC_PREVWINDOW	0xF050	前一个窗口（在 MDI 文档窗口里）
SC_CLOSE	0xF060	Close 菜单项
SC_VSCROLL	0xF070	垂直滚动
SC_HSCROLL	0xF080	水平滚动
SC_MOUSEMENU	0xF090	System 菜单的显示是一次鼠标单击的结果
SC_KEYMENU	0xF100	System 菜单的显示是一次按键操作的结果
SC_RESTORE	0xF120	把窗口的尺寸和位置恢复成上一次的状态
SC_TASKLIST	0xF130	执行或者激活 Windows Task Manager
SC_SCREENSAVE	0xF140	执行屏幕保护程序
SC_HOTKEY	0xF150	激活与特定热键联系在一起的那个窗口。lParam 的低字指定了准备激活的窗口
SC_DEFAULT	0xF160	选择缺省的菜单项——通常是 Close
SC_CONTEXTHELP	0xF180	
SC_SEPARATOR	0xF00F	
SC_ICON	SC_MINIMIZE	
SC_ZOOM	SC_MAXIMIZE	

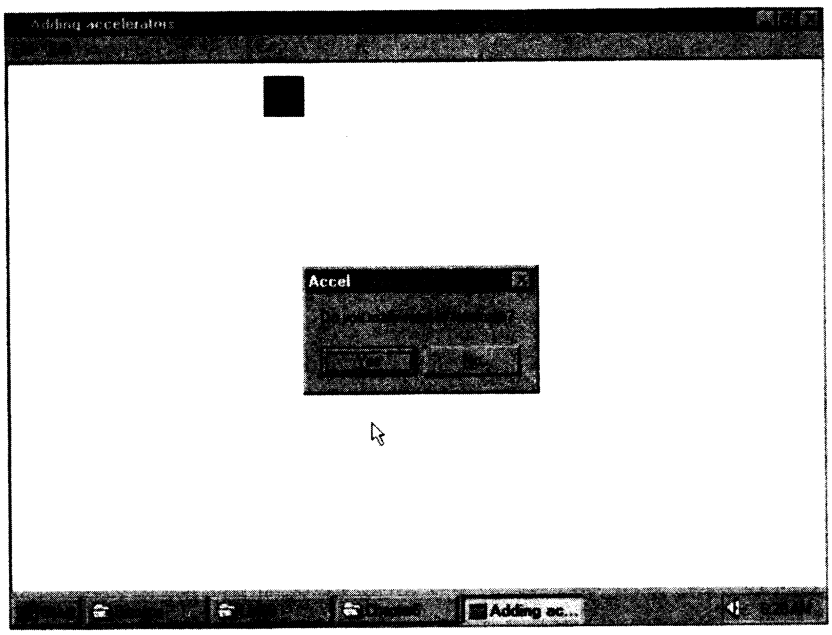


图 6-28　程序中断之前，应用程序会询问用户是不是真的想退出

第7章　建立窗口的艺术

在前面的章节里，我们开发了几个小项目，它们都采用了基本的窗口类型，但对建立窗口的艺术却并不是很在意。在这一章里，我们将深入探讨由 Win32 API 提供的多种选择方案，从而对这一主题进行全方面的论述。

正如我们在第 3 章“Win32 应用程序的开发”里指出的那样，CreateWindowEx（）是用于生成一个 Win32 窗口的基本工具。该函数的语法相当丰富，但是大家仍然需要小心地对其进行调用，特别要把注意力放在它的第一个和第四个参数上面——dwExStyle 和 dwStyle。

```
#include〈winuser.h〉
HWND WINAPI CreatWindowEx（DOWRD dwExStyle，
                          LPCSTR lpClassName，
                          LPCSTR lpWindowName，
                          DWORD dwStyle，
                          int X，
                          int Y，
                          int nWidth，
                          int nHeight，
                          HWND hWndParent，
                          HMENU hMenu，
                          HINSTANCE hInstance，
                          LPVOID lpParam）；
```

其中新增的扩展风格可以让开发者对窗口的外观和行为进行更加广泛的控制（请参见表 3-9，其中列出的所有 WS_EX_风格）。无论风格的本质是否单一，它在窗口的建立进程中确实占据了相当重要的地位。某种窗口风格应该属于下面这三种类别中的其中一类：

▶叠置式

▶弹出式

▶子窗口

如果想生成一个叠置式窗口，我们就要用到 WS_OVERLAPPED 标志。在几乎所有的例子里，由于 WS_OVERLAPPEDWINDOW 包含了某些附加的风格，这些风格在功能上对原始的 WS_OVERLAPPED 进行了扩展，所以我们使用的一般都是前者。作为另一种选择，一个窗口也可以使用 WS_POPUP 标志，从而使自己变成一个弹出式窗口。在 Windows 的 1.x 版本里，弹出式窗口是非常流行的。然而在后续的 Windows 版本里，弹出式窗口露面的场合就很少了。对话框和属性表就归属于弹出式窗口一类。因此，在 Windows 95 里，我们碰到这种类型窗口的机会不会很少。正如我们不久将要讨论的那样，现在的对话框和属性表是通过

一种不同的策略建立起来的。这种策略不需要用到 CreateWindowEx（）或者其他任何一种类似的 API。

子窗口是第三种，也是最后一种可选的窗口风格。一个典型的 Win32 应用程序提供了对几十种子窗口的支持，这些子窗口是通过 WS_CHILD 标示来进行区分的。总而言之，系统内的每个窗口都有可能是叠置式、弹出式或者子窗口的其中一种。具体属于哪一种要取决于显式或者隐式调用 CreateWindowEx（）函数时传递的 WS_风格是哪一个。

我在这儿需要事先强调一个问题：无论窗口的类型如何，一个单独的窗口类都能够代表几个窗口，这几个窗口采用叠置式、弹出式或者子窗口均可。当然，用同一类代表几个混杂窗口的情况并不多见，但这在技术上是完全可能的。表 7-1 为大家总结了指定一个窗口类型的基本风格。

表 7-1　Win32 的三种窗口类

风　格	值	说　明
WS_OVERLAPPED	0x00000000L	生成叠置式窗口
WS_POPUP	0x80000000L	生成弹出式窗口
WS_CHILD	0x40000000L	生成一个子窗口；每个窗口都可以有数个子窗口

在这三种原始定义的基础上，WINUSER.H 里还包含了其他一些衍生出来的窗口风格，如表 7-2所示。我们最好使用 WS_OVERLAPPEDWINDOW 标志，因为它能非常方便地包含一系列有用的值。

表 7-2　把标志集合到一起，从而简化窗口风格的设置

风　格	值	说　明
WS_OVERLAPPEDWINDOW	(WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME | WS_MINIMIZEBOX|WS_MAXIMIZEBOX)	用几种风格属性定义一个叠置式窗口
WS_POPUPWINDOW	(WS_POPUP | WS_BORDER | WS_SYSMENU)	用几种风格属性定义一个弹出式窗口
WS_CHILDWINDOW	(WS_CHILD)	这个定义在基本风格基础上没有增加任何新东西

其中，WS_POPUPWINDOW 标志增加了两种有用的标志。但这些标志仍然需要和其他 WS_风格集成到一起，否则便无法生成一个典型的窗口——带有标题栏，并在左上角带有图标。相反，WS_CHILDWINDOW 没有什么变化，它在这儿是没有什么用处的。在三种基本的窗口类型中，WS_OVERLAPPED 明显是最常用的一种，从它的值（**0x00000000L**）就可以看出这一点。这些类型之间的区别关系到叠置式窗口在标准 **Win32** 应用程序中的地位，弹出式窗口和子窗口扮演的角色显然是比较次要的，所以让我们先从叠置式窗口开始进行探讨。

7.1　叠置式窗口类型

对于一个 Win32 进程来说，它在屏幕上几乎百分之百是以一个叠置式窗口的面目出现的。在过去，一个标准的 Windows 应用程序往往只提供了一个叠置式窗口，这个窗口就代表

了程序的整体布局——在这个窗口的下面也许还有一些子窗口。对于按照 MDI（多文档界面）规范设计的应用程序来说，这种布局是相当标准的。这些应用程序的例子包括 Microsoft Excel——正是由这个软件产品引入了 MDI 规范，并且为它注入了活力。Windows 95 采用了面向对象的用户界面，并且吸纳 OLE 作为新开发软件的主心骨，它并不鼓励开发者把 MDI 模式作为开发 Win32 应用程序的首选模式。更准确地说，Windows 95 鼓励程序开发者为每个程序设计多个窗口，每个窗口都用于完成某种特定的任务。

为了满足我们刚才提到的要求，叠置式窗口势必在 Windows 95 里变得越来越流行，这是由于每个应用程序都有采用多个叠置式窗口的趋势。这样造成的结果就是，以单独一个叠置式窗口为基础的应用程序将逐渐减少，直到最后趋向于消失。据我所知，现在仍有许多忠心耿耿的 MDI 追随者。尽管如此，这些人还是要作好准备，迎接 Windows 95 应用程序外观全面改变的时代到来。Win95 外壳是迄今为止设计得最好的一个 Win32 应用程序原型，它是完全按照新的用户界面规范设计的。文件夹的行为就相当于单独的窗口（既有弹出式的，也有叠置式的），对象总是提供了一个关联菜单，而且提供了对拖放功能的全面支持。正因为如此，我们可以先来看看一个标准的文件夹——叠置式窗口的一个典型代表物。我们现在利用 WINSPY 这个示范程序（请参考第 16 章）对 My Computer 对象的打开文件夹进行检查，它可以侦测到下面这些基本的风格：

WS_OVERLAPPED	0x0000
WS_MAXIMIZEBOX	0x0001
WS_MINIMIZEBOX	0x0002
WS_THICKFRAME	0x0004
WS_SYSMENU	0x0008
WS_CAPTION	0x00C0
WS_CLIPCHILDREN	0x0200
WS_CLIPSIBLINGS	0x0400

微软的工程师们采用的是 WS_OVERLAPPED 标志，因为它提供了一系列特性集，特别适合文件夹使用。事实上，一个叠置式窗口可以占据屏幕上的任何位置，并且很容易就可以改变它的尺寸，使其占据整个显示器。

WS_MAXIMIZEBOX 和 WS_MINIMIZEBOX 标志的作用是在窗口右上角显示出两个按钮，从而让用户方便地选择窗口的最小或者最大化显示。除此以外，由于采用了 WS_THICKFRAME，所以用户很容易就可以对文件夹的边框进行操作，从而直接实现窗口的尺寸变化。标题栏的生成则是 WS_CAPTION 标志的功劳；没有这个标志，就不可能显示出两个最大化和最小化按钮，以及位于左上角的系统菜单（WS_SYSMENU）。

WS_CLIPCHILDREN 标志的作用是指示叠置式窗口在客户区域切开一个矩形区域，用以容纳子窗口（子窗口会接管它所占据的客户区的所有重画操作）。对于叠置式窗口来说，WS_CLIPSIBLINGS 标志基本上是没有什么用处的，但该标志对于子窗口来说却几乎是必需的。假如位于同一级别的两个兄弟窗口发生了冲突，一个叠置于另一个的上方，那么使用 WS_CLIPSIBLINGS 就可以解决这一问题。WS_CLIPSIBLINGS 在这儿的存在不会造成任何害

处，它也不会放慢程序运行的速度。

除此以外，诸如 My Computer 这样的标准文件夹还提供了两种扩展风格，从而对窗口的整体外观进行了改进。这两种风格分别是 WS_EX_CLIENTEDGE 和 WS_EX_WINDOWEDGE，它们的作用是强迫客户区域的显示具有三维效果，如图 7-1 所示。

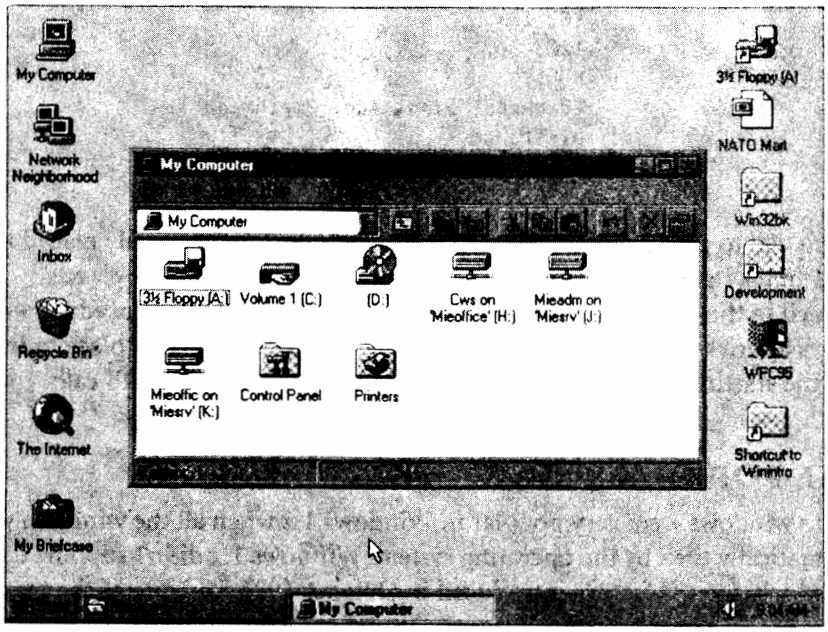


图 7-1　My Computer 文件夹是一个典型的叠置式窗口，
它是根据 Windows 95 引入的风格规范设计的

关于叠置窗口的主题是相当繁杂的，更详细的信息我们还会在第十六章“Win95 外壳的开发”里进行介绍。

叠置式窗口的建立

WS_OVERLAPPEDWINDOW 是由一系列经常用到的标志组成的，其中包括用于重新定义窗口尺寸的边框以及一个标题栏，这两者都是标准窗口的基本特征。除此以外，WS_EX_OVERLAPPEDWINDOW 风格还为每一种 Windows 95 对象增加了一种独特的三维外观。对于 CreateWindowEx () 的语法来说，它还需要用到一个菜单句柄，这是在窗口建立时从资源文件里载入的一种资源。请大家参考下面这的个代码段：

```
...
hwnd = CreateWindowEx (WS_EX_CLIENTEDGE | WS_EX_WINDOWEDGE,
                       szClassName,
                       pszWindowTitle,
                       WS_OVERLAPPEDWINDOW,
                       10, 10
                       120, 120
```

```
                    NULL,
                    LoadMenu (hInstance, " mainmenu"),
                    hInstance,
                    NULL);
...
```

用于生成主窗口的这种调用通常放置于 WinMain () 里，而且要在程序内计划的所有窗口类的注册完成后马上进行。通过这种调用，类窗口进程就会接收到一系列消息，其中包括 WM_CREATE。在 WM_CREATE 里，我们可以通过对其他 API 的调用，从而中断应用程序的起动阶段。

7.2 弹出式窗口类型

弹出式窗口在 Windows 1.x 里是非常流行的。在那个时候，所有窗口都会由操作系统自动进行平铺处理。Windows 1.x 还没有提供对叠置式窗口的支持——这种特性是于 1987 年十月首次引入的，它也是那场著名的“苹果——微软”诉讼案所争论的一个焦点。在那时，弹出式窗口已经可以避开那种强制性的窗口平铺规则，可以散布于屏幕上的各个地方。由于这个原因，由系统发出的每一条通知消息都会放置到弹出式窗口里，从而立即引起用户的注意。随着叠置式窗口的引入，弹出式窗口丧失了它的霸方地位，它们现在只是系统的一种普通特性。正如我以前提到的那样，每个对话框都是一个弹出式窗口，相同的规律亦适用于外壳属性表，就像图 7-2 显示的那样。

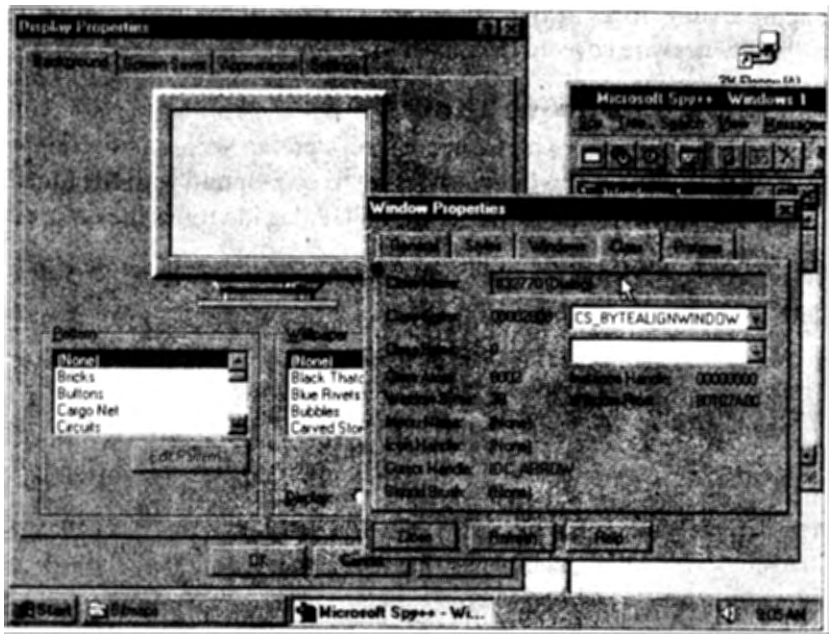


图 7-2 桌面属性表是一种弹出式窗口，它归属于未文档化的 #32770 窗口类，同时也当作一个独立的进程运行

从视觉效果的角度来看，要区分出叠置式窗口和弹出式窗口几乎是一件不可能的事情。一名开发者可以针对这两种窗口应用相同的基本和扩展风格集，从而使它们的外观没有什么分别。所以我们不要根据对图 7-1 和 7-2 的的比较，就草率地作出某些结论——因为这些窗口是完全不同的。

与叠置式窗口类似，弹出式窗口也可以放置于屏幕上的任何地方，也能够显示出一个菜单栏，从而与用户进行交互作用。这样一来，这两种类型的窗口之间到底有什么区别呢？在某些环境下，只有根据两个特征也许才能把它们区分开来：

▶弹出式窗口可被分配一个物主

▶叠置式窗口肯定提供了一个标题栏

在通常情况下，开发者都愿意使用叠置式窗口。因此，弹出式窗口主要应用于对话框和属性表。在第 8 章“Win32 的对话框管理”和第十章“Windows 通用控件”里，我们将用两种不同的办法来建立这两种窗口。至于“物主”，我们会在本章的后面部分进行介绍。

7.2.1 弹出式窗口的建立

从根本上说，弹出式窗口和叠置式窗口建立的唯一区别在于它们各自对应的窗口风格集不同，这对应于 CreateWindowEx () 的第四个参数。请注意，WS_POPUP 标志明确地说明要建立的是一个弹出式窗口。

```
...
hwnd = CreateWindowEx (WS_EX_CLIENTEDGE | WS_EX_WINDOWEDGE,
                       szClassName,
                       pszWindowTitle,
                       WS_POPUP | WS_SYSMENU | WS_MINIMIZEBOX|
                           WS_MAXIMIZEBOX | WS_THICKFRAME,
                       10, 10,
                       120, 80,
                       NULL,
                       LoadMenu (hInstance, " mainmenu"),
                       hInstance,
                       NULL);
...
```

CW_USEDEFAULT 定义要求用到由 Windows 决定的一个位置和一个窗口尺寸，这种定义在叠置式窗口里是可行的，但在弹出式窗口里就行不通了。因此，开发者必须为弹出式窗口指定一些数值，从而标识出它的窗口位置和大小。

和叠置式窗口的另一处差异在于 hwndParent——从上往下数第九行参数。弹出式窗口需要直接从屏幕获得显示象素，所以它的“父窗口”就是桌面，换言之即 HWND_DESKTOP。通过把第九行参数指定为 NULL，就可以避免错误的发生，因为这正好就是在 WINUSER.H 里定义的 HWND_DESKTOP 定义的值：

```
#define HWND_DESKTOP ( (HWND) 0)
```

弹出式窗口肯定是没有真正的父窗口的，然而它们却可以通过指定一个物主窗口来建立。物主窗口在 Windows 编程中是一种新引入的概念。在 Win32 API 里，对窗口物主身份的定义是相当暧昧的，因为并没有实际的参数可用于明显地指定一个窗口物主。建立弹出式窗口的时候，我们可以 hwndParent 参数里指定一个窗口句柄，从而避开这种限制。这样一来，建立的窗口就会成为弹出式物主窗口，同时不会影响到它的“父窗口”——系统桌面。因此，Win32 支持两种类型的弹出式窗口：一个带有物主，一个则没有物主。本书附带 CD 的 Listing 7.4 向大家展示了这两种窗口在行为上的差异。

当我们对两个窗口间的关系进行阐述的时候，物主身份也许就是其中的一种关系。叠置式窗口的行为就好象一个物主，而它所拥有的窗口正好就是弹出式窗口。勿庸我们多作说明，对于带有物主的一个弹出式窗口来说，假若它的物主窗口以最小化显示，那么它也会从屏幕上消失——尽管这两种窗口并未在物理意义上连接到一起。关于物主关系另外一个需要注意的问题是在物主窗口以最大化显示的时候产生的，在这时，物主窗口会占据整个显示空间。物主窗口最大化显示以后，它所拥有的那个弹出式窗口并不会消失在叠置式窗口的下方——相反，它会停留于前台，并且一直保持可见状态。OWNER 示范程序可以让大家观察这种行为，并且可以比较带有物主和未带物主的弹出式窗口在行为上的差异。

7.3 子窗口类型

子窗口的命运显然没有弹出式窗口那么好。在 Windows 1.x 的风格设计准则里，子窗口很不出名，也没有引起大家对它重视。在 Windows 2.x 里，随着 Microsoft Excel 和 MDI 模式的出台，它才作为一种优选的窗口工具渐渐地有了一些名气。在 Windows 95 里，子窗口失去了它的大多数重要性，所以叠置和弹出式窗口的地位又变得重要起来。根据这种新的用户界面设计准则，以及作为面向对象的设计模型的结果，MDI 模式已经变得过时，或者说它与数据和对象的直接操作不能很好地匹配。在以后几年内，我们可以预见到应用程序将逐渐从 MDI 模式过渡到单一的文档界面（SDI）。到那个时候，子窗口的应用场合将变得越来越少。

除了这些方面的考虑以外，子窗口的行为也是很奇怪的，对它的设计也不是很方便。通常，活动窗口的标题栏都是蓝色的，或者采用系统用于标识活动状态的那种颜色。但这对于子窗口来说却是无法自动生成的。此外，我们不能为子窗口分配菜单栏，原因很简单，那就是在技术上不可能实现。子窗口总是需要一个标识编号才能与它的父窗口共同工作，这种特性无论弹出式窗口还是叠置式窗口都是不存在的。

根据定义，由于子窗口必须驻留于父窗口的客户区域内，所以它的显示尺寸就受到了限制。假如一个子窗口带有一个标题栏，那么它就可以在父窗口的客户区域内自由移动。任何把它拖出父窗口边框的企图都是无谓的——超出的显示区会自动地剪切掉，从而限制了它的可见范围。图 7-3 显示了右侧被剪切掉的 Excel 文档。

在 Windows 95 里，我们更常见的一种情况是子窗口没有提供标题栏，并把自己的显示保持于父窗口客户区内一个固定的位置。这种情况对于对话框或者属性表内的所有控件来说都是适合的，如图 7-4 所示。

下面简单总结一下我们以前对子窗口的那些简要介绍。子窗口作为对话框或者属性表内的控件元素使用时，它的地位就显得很突出，但在其他场合却不大适用。子窗口作为文档窗口在 MDI 应用程序里应用的机会在以后几年内也会越来越少。同时，在本书剩余的部分里，我

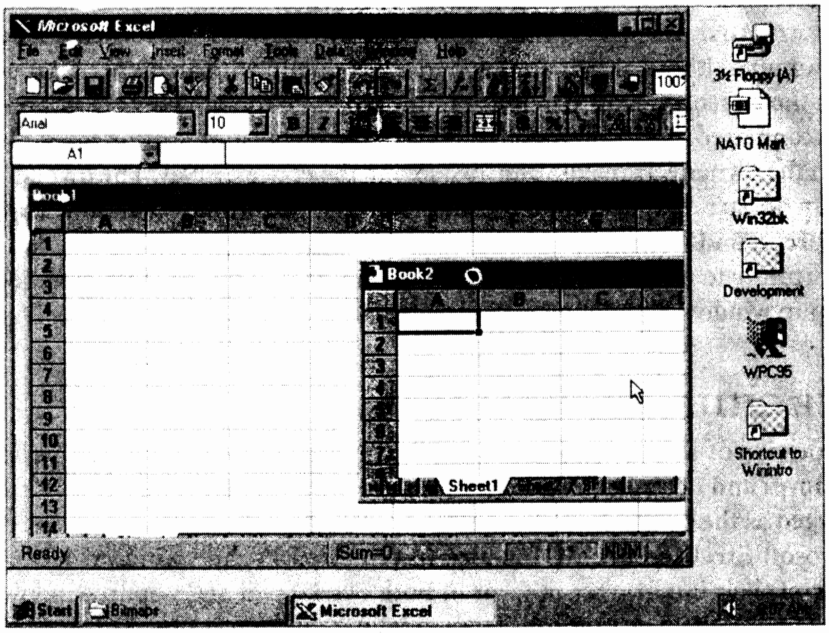


图 7-3　每个 Excel 文档都驻留于主窗口的客户区内，
假如向边框拖动它们，一些部分就会被剪切掉

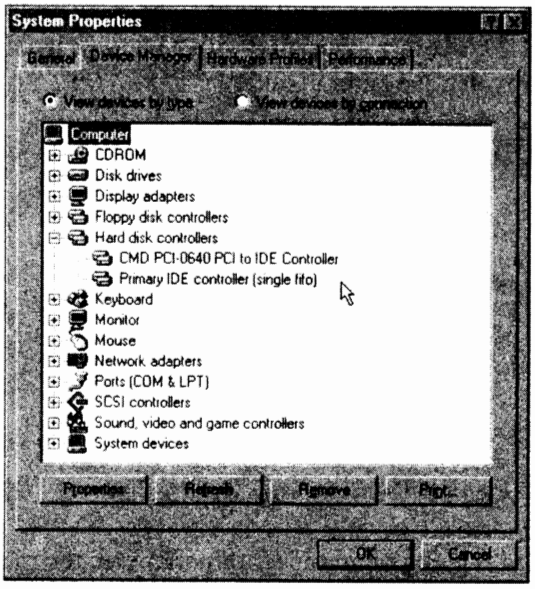


图 7-4　系统属性表内 Device Manager 页提供了一个 TREEVIEW 窗口，
并在下面排列了几个按钮。所有这些控件都是子窗口

们也不会再对子窗口作更多的说明。表 7-3 对子窗口和父窗口间的关系进行了总结，其中涉及到窗口内发生的一些基本行动。

表 7-3 子窗口和父窗口之间的关系

行 动	子窗口	父窗口
删除	在父窗口之前	在子窗口之后
隐藏	在父窗口之前	在子窗口之后
移动	同时	同时
显示	在父窗口之后	在子窗口之前

某个子窗口的父窗口到底是哪一个，父窗口和子窗口之间的关系如何呢？我们将在下一小节对这个问题作出解答。

7.3.1 子窗口的建立

处理子窗口的时候，CreateWindowEx()语法对标准的规则进行了一些变动。WS_CHILD 标志简单地通知 Win32 开发者准备建立不准备从桌面取得显示空间的一个窗口。因此，WS_CHILD 通常都要与指定象素实际提供者——父窗口的窗口句柄联系在一起。

```
...
hwnd = CreateWindowEx (WS_EX_WINDOWEDGE | MS_ED_CLIENTEDGE,
                       szChildClass,
                       szWindowTitle,
                       WS_CHILD | WS_SYSMENU |
WS_MINIMIZEBOX | WS_MAXIMIZEBOX |
                           WS_THICKFRAME,
                      10, 10,
                      120, 80,
                      hwndParent,
                       (HMENU) ID_CHILD,
                      hInstance,
                      NULL);
...
```

其中，第十行参数（菜单句柄）有些奇怪。迄今为止，我们知道这个参数只有两种变化，它要么是一个菜单句柄，要么是一个 NULL 值（假如没有菜单与窗口关联）。子窗口从结构上说是不可能分配一个菜单的，但是它们却要求使用一个窗口标识编号。考虑到 CreateWindowEx() 函数的参数数目是固定的，所以微软的工程师们采用一种折衷的办法，那就是用菜单句柄参数来接收子窗口的 ID。这种 ID 可以是我们想象得出的任何一个数字，通常按照下面这条语句展示的那样进行定义：

```
#define ID_CHILD      1000
```

显然，并没有什么规定要求我们必须为子窗口分配一个 ID。然而，根据开发准则的要求，我们强烈建议开发者为它们分配对应的 ID，因为这在子窗口运用的时候是相当有用的。在常规的 Win32 应用程序里，进程会在所有活动的窗口间进行频繁的交互作用——这种交互作用

是建立在由源代码生成的消息流的基础上的。这时如果知道了一个子窗口的 ID，我们就有机会知道它的句柄，这个句柄对任何窗口操作来说都是很一种很基本的信息。

7.3.2 从属：父子关系

在第 2 章“Win32 开发工具”里，我们对窗口 API 和系统 API 进行了区分。掌握这两者的区别对于 Win32 程序的开发是非常重要的。无论是哪个进程生成的窗口，它的信息都会保存于 Windows 的内部。因此，窗口管理模块将按照一种分级结构对所有的窗口进行控制。这种分级结构将建立在调用 CreateWindowEx（）时指定的值的基础上。

桌面可以想象成所有现成窗口的一个“祖先”，叠置式窗口是它的第一代“继承人”。桌面相当于 .SDK 头文件内的 HWND_DESKTOP 定义。我们可以把这个“桌面”想象成自己的笔记本计算机或者桌面 PC 机的物理显示屏。

从程序开发的眼光来看，我们必须把桌面考虑成 Windows 的一种内部组件，它位于窗口分级结构的最顶部，并为第一代窗口（弹出式和叠置式）提供了显示象素。外壳本身的输出区域是从桌面取得的，并且占据了全部桌面。通过调查，我们发现外壳的主窗口归属于 PROGMAN 类，并且带有属于 SHELLDLL_DefView 类的一个子窗口。这两个窗口都会根据用户的当前设置对屏幕的尺寸进行假设。属于 SHELLDLL_DefView 类的窗口还提供了一个子窗口——一个标准的列表视窗——亦即属于 SysListView32 类的一个窗口，它的 ID 设置成 1。这种窗口的堆叠最后终结于属于 SysHeader32 类的一个窗口，该窗口根本就没有设置自己的尺寸。图 7-5 向大家展示了外壳窗口的设置。

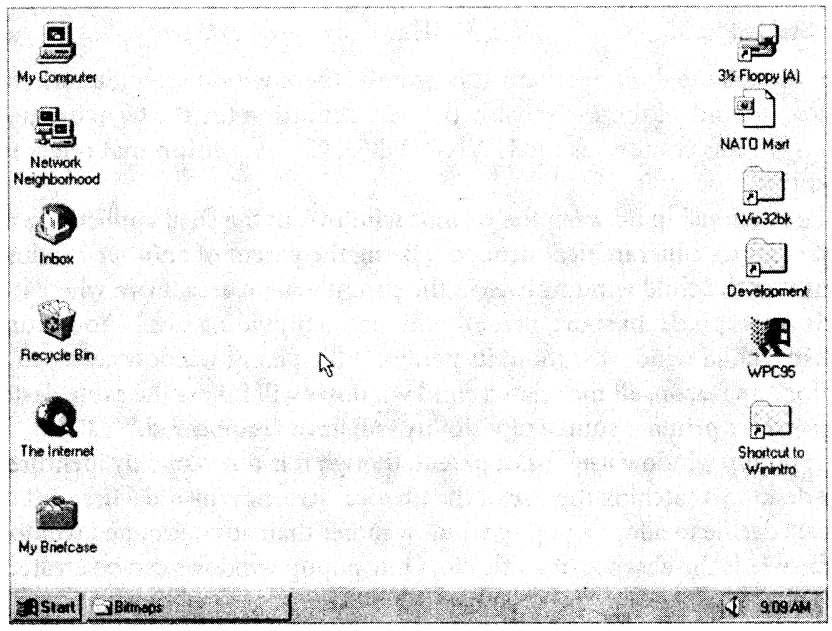


图 7-5 Windows 95 外壳是由几个叠置式窗口组成的一个标准 Win32 进程。其中最主要的是 WS_POPUP 窗口

“桌面”这个术语在这儿让人感到十分迷惑，而且极易让人误解，因为周围存在着如此多的桌面！根本地说，外壳被分割成两个组件：任务栏和桌面。其中桌面是对象驻留的显示部分。Windows 95 外壳是一种标准的 Win32 进程。引导阶段结束以后，系统就会立即载入并启动这个进程。实际上，外壳是标记为 Program Manager（程序管理器）的一个窗口。然而用户是无法看到这种标记正文的，因为它没有提供对应的标题栏。外壳覆盖于整个屏幕表面，上面叠置了一系列窗口。假如对此进行详细的研究，我们就会发现外壳主窗口归属于 PROGMAN 类，并且带有属于 SHELLDLL_DefView 类的一个子窗口。

用于定义系统外壳桌面部分的四种基本窗口分别属于下面这些窗口类：

类	风格	父
PROGMAN	WS_POPUP	HWND_DESKTOP
SHELLDELL_DefView	WS_CHILD	PROGRAM
SysListView32	WS_CHILD	SHELLDLL_DefView
SysHeader32	WS_CHILD	SysListView32

由此看来，桌面上到底是什么呢？并非上面这些窗口的任何一个，而是隐藏在它们后面的另外一个窗口！对于“桌面”这个术语来说，也许最好的定义就是 HWND_DESKTOP（至少在这种情况下如此），该定义引用了这个实体。

对于外壳应用程序内不同窗口的关系来说，它明显定义了一个分级结构。假如某个窗口是另外一个窗口的父窗口，那么就意味着子窗口位于父窗口的客户区内。只有在这个客户区内，子窗口才能获得它显示所需的象素。简而言之，父子关系意味着父窗口要为子窗口提供象素。假如没有父窗口，那么一个子窗口是无法单独成活的。假如父窗口出于某种原因而中断退出了，那么相关的所有子窗口也逃不脱相同的命运，因为它们显示的基本来源消失了。

弹出式窗口本身就有一个父窗口，尽管我们并未对此作明显的定义。这个父窗口就是“桌面”，它的大小相当于整个屏幕的显示尺寸。为什么微软的工程师在这儿决定采用弹出式窗口的概念，而不采用叠置式窗口的概念呢？答案很简单，那就是前者没有标题栏。只有弹出式窗口才能在不具备标题栏的前提下创建（尽管子窗口也具备类似的条件，但是它在这种情况下并不是恰当的工具，因为这里需要的是顶级窗口）。

叠置式窗口的象素来源肯定是桌面，弹出式窗口亦是如此。而子窗口则要依赖于一个父窗口的支持。这个父窗口可以是叠置式窗口、弹出式窗口或者另外一个子窗口。因此，通过对 GetParent（）的一系列调用，我们很容易就可以知道窗口的分级结构是怎样的。该函数能返回父窗口的句柄：

```
#include〈winuser.h〉
HWND GetParent（HWND hwndChild）;
```

假如返回值等于 HWND_DESKTOP 定义，这就意味着已经到达了这个树形分级的最高处——系统桌面。在本章的稍后部分，我们将介绍一种沿着窗口分级结构向下摸索的方法，它需要我们先具备一些附加的知识。

表 7-4 为大家总结了 Win32 三种基本窗口所支持的基本特征。

表 7-4 Win32 的三类窗口

类 型	菜 单	ID	标题栏	物 主	父	主窗口	每个应用程序内的数目
WS_OVERLAPPED	支持	不支持	肯定有	不支持	不支持	支持	通常一个
WS_POPUP	支持	不支持	支持	支持	不支持	不一定	有一些
WS_CHILD	不支持	支持	支持	不支持	肯定有	不支持	有许多

7.4 标题栏按钮

放置于标题栏最右侧的图标相当于系统菜单内特定窗口命令的一些快捷。这时，我们毋需选择系统菜单内的某个菜单项，只需用鼠标左键简单地单击一个标题栏按钮就可以了。这时，应用程序就会接收到一条 WM_SYSCOMMAND 消息，其中的 wParam 内包含了对应的 SC_定义。假如在标题栏的任何空白位置处单击鼠标右键，就会显示出系统菜单。

典型情况下，下面这些按钮会在一个主窗口内显示出来（前提是窗口支持这些基本功能）。

命令按钮　　作用

关闭窗口

最小化显示窗口

最大化显示窗口

恢复窗口

对窗口和 CreateWindowEx () 进行运用的时候，有一些相关的事项是需要引起我们注意的。CreateWindowEx () 的第三个参数是指窗口标题栏内显示的正文。这种定义尽管很平常，但它也并非能应用于每一种环境。建立一个窗口的时候，假如这个窗口属于诸如 LISTBOX 这样的预定义窗口类，标题栏的概念就会产生误导作用，因为 LISTBOX（列表框）里并未提供这种对象。BUTTONS（按钮）也没有什么标题栏，通过 CreateWindowEx () 第三个参数传递的正文最终变成了按钮上的正文串。除此以外，第十二个参数也是一种比较暧昧的指针，它指向的位置并没有进行很完善的定义。

正如大家以后在本章要看到的 PARTY3 示例程序那样（参考本书附带 CD 的 Listing 7.3），这个参数可以当作存放有价值信息的一个区域来使用，开发者到那时就可以在其中放置任何种类的值。

7.5 三种窗口尝试

为了牢固地掌握窗口的建立、各窗口的关系以及它们的工作原理，我设计了三个连贯的程序示例，通过它们可以了解如何在 Win32 应用程序里运用不同的窗口类型。这儿需要注意几个问题：多个窗口类的注册、几个窗口进程的编写、在窗口间传递消息、采用适当的窗口风格及其他。

7.5.1 一起来聚会！版本 1

本书附带 CD 的 Listing 7.1 内包含了 PARTY1 例子的源代码。这是一个 Win32 程序，它在屏幕上模拟显示了三个窗口。这三个窗口分别归属于三种不同的窗口类，如图 7-6 所示。这三个窗口的建立是在 WinMain () 里直接完成的——就在注册完窗口类以后。当然，这种策略并不是最优的，但它很容易实现。在以后的例子里，我们会对此作出改进。

PARTY1 唯一特殊的地方在于它允许用户通过主窗口菜单去关闭弹出式窗口或者子窗口。子窗口应该带有自己的 ID，这是一种不能重复的标识编号，分配它的目的一般都是为了获得子窗口的句柄。Win32 的 API 函数 GetDlgItem () 可以根据 ID 和父窗口句柄返回一个子窗口的句柄，该函数的语法结构如下：

```
#include〈winuser.h〉
HWND GetDlgItem (HWND hDlg, int nIDDlgItem);
```

参数	说明
HWND hDlg	父窗口的句柄
int nIDDlgItem	子窗口 ID
返回值	说明
HWND	子窗口句柄；假如函数调用失败，则返回一个 NULL 值

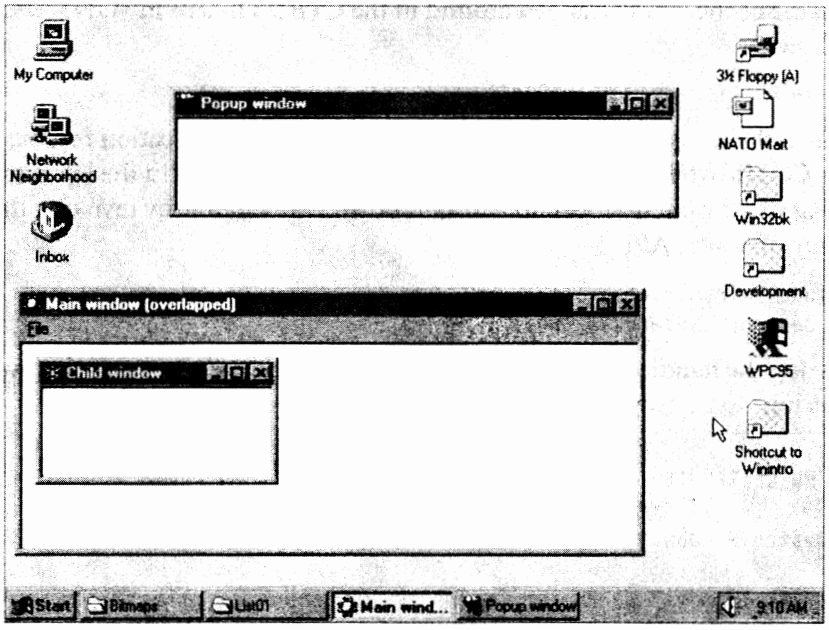


图 7-6 PARTY1 程序启动以后，屏幕上显示了一个叠置式窗口、一个弹出式窗口以及一个子窗口

尽管它的名字有些不可议，但是 GetDlgItem () 函数对于任何父-子窗口都能工作得很好。

在最初的时期，子窗口只能在对话框里使用——就在 MDI 模式诞生之前。这样就解释了该 API 函数名称的来历（在英语里，对话框的原文是 DialogBox，这就是函数名中 Dlg 的起源）。针对这个函数，我在 WIN32BK.H 里为它换为了另一个名字 CTRL（），如下所示：

```
#define CTRL (x, y) GetDlgItem (x, y)
```

假如用户选择了 Close child 菜单项，程序就会接收到一条 WM_COMMAND 消息，其中的 wParam 低字内包含了 MN_CLOSECHILD。在这以后，我们再通过调用 DestroyWindow（）这个 API 就可以很容易地关闭那个窗口。

```
#include 〈winuser.h〉
BOOL DestroyWindow (HWND hwnd);
```

该函数会返回准备中断的那个函数的句柄。下面这个代码段展示了如何消除子窗口。

```
...
case MN_CLOSECHILD:
{
    DestroyWindow (CTRL (hwnd, CT_CHILD));
}
    break;
...
```

对于弹出式窗口来说，同样的操作就变得稍微有些复杂了，因为没有 ID 来标识这样一个窗口（HMENU 项既可以包含 NULL，亦可包括一个有效的菜单句柄）。非常不幸，Win32 API 和窗口处理模式没有提供任何一种途径能够方便地取回一个弹出式窗口的句柄——无论该窗口是否一个物主。正是由于 ID 的缺乏，所以叠置式窗口和弹出式窗口之间不存在结构化的链接关系。因此，我们必须采取不同的策略来达到目的。在 PARTY1 这个程序里，我们调用 FindWindow（）来获取弹出式窗口的句柄，并且把这个句柄直接传送给 DestroyWindow（）：

```
...
case MN_CLOSEPOPUP
{
    char szPopupClass[20];

    LoadString(hInstance, ST_POPUPCLASS, szPopupClass, sizeof(szPopupClass));
    DestroyWindow(FindWindow(szPopupClass, szPopupClass));
}
    break;
...
```

FindWindow（）这个 API 函数为我们提供了一种简便的方式，利用它就可以解决当前环境下碰到的问题。该函数会返回一个窗口句柄，这个句柄是通过对整个窗口管理模块进行搜

索而得到的——这种搜索要以两类信息为基础：窗口类的名称以及窗口标题：

```
#include〈winuser.h〉
HWND FindWindow（LPCTSTR lpClassName，LPCTSTR lpWindowName）；
```

参数	说明
LPCTSTR lpClassName	窗口类的名称
LPCTSTR lpWindowName	标题栏内出现的正文串；假如不考虑这种信息，把搜索范围扩展到所有窗口，亦可设置成 NULL
返回值	在正文里讨论
HWND	子窗口句柄；假如函数调用失败，则返回一个 NULL 值

在 PARTY1 这个例子里，我们从 .RC 文件的 STRINGTABLE 资源里获得了类名以及窗口标题，然后把它传递给 FindWindow ()。在这种特定的情况下，两个正文串是完全一致的。比较常见的一种情况是，我们通常把 FindWindow () 的第二个参数设置成 NULL。这样一来，该函数就只会对类名称进行搜索。我们编写的这段代码运行得很好，大家可以自己运行一下 PARTY1 程序。假如存在属于同一类的多个窗口，这时会有什么情况发生呢？FindWindow () 在这时返回的是什么呢？好了，我们不可能预知搜索的结果。除此以外，我们也不可能猜测出由第一个参数指定那一类窗口中的哪一个会被取回。因此，PARTY1 实现的算法存在一些潜在的问题。对属于同一类的多个窗口进行处理时，我们需要对其进行改进。

PARTY1 给我们提出了两个应该引起注意的问题。假如用户用鼠标左键选择叠置式窗口，这时会发生什么情况？窗口被激活了，这种新的状态从视觉效果来说是用一个蓝色的标题栏表示的，如图 7-7 所示。同时，这种属性从前一个活动的弹出式窗口内消失了，那个窗口现在处于非活动状态。

现在选择子窗口，这时又会发生什么情况呢？什么也不会发生。这是由于 Windows 的内部结构化本质造成的。子窗口从来都不会显示一个蓝色的活动标题栏——这对于开发者来说是相当讨厌的。为了解决这个问题，我们需要增加另外一些代码。

首先，我们需要从整体策略上进行综合考虑。激活一个子窗口的时候，我们希望它明确地显示出一个蓝色的标题栏，用户选择了其他窗口以后，这种属性又需要消失掉。但是“其他窗口”又是什么呢？其中包括它的父窗口吗？显然不包括！只有系统内其他任何一个窗口激活以后，子窗口才会失去它的活动状态，“其他窗口”内并不包括它的父窗口。

下面介绍如何实现蓝色标题栏的激活。在子窗口进程里，我们可以捕获 WM_MOUSEACTIVATE 消息。假如用户通过鼠标单击选中了一个窗口，就会生成这样的消息，这条消息的结构如下所示：

WM_MOUSEACTIVATE	0x0021
wParam	顶级父窗口的句柄
LOWORD (lParam)	由 DefWindowProc () 返回的“命中”测试值
HIWORD (lParam)	用户按下鼠标按钮时产生的鼠标消息标识符

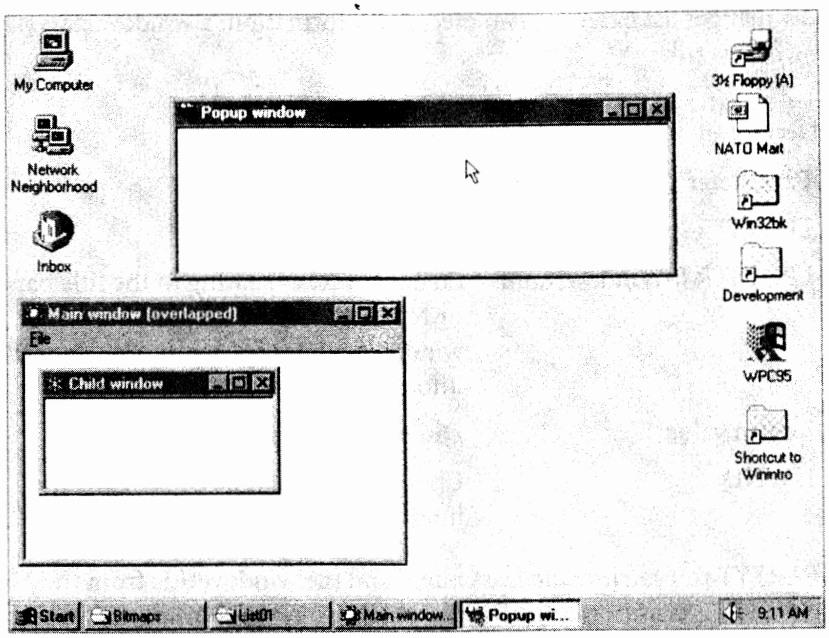


图 7-7　通过选择弹出式和叠置式窗口，用户可以在
这两种窗口之间转换视觉焦点

为了指示 Windows 为子窗口分配活动颜色，我们应该按照下面这种形式发送 WM_NCACTIVATE 消息：

```
...
case WM_MOUSEACTIVATE:
{
    //activate the child window displaying a blue title bar
    SendMessage (hwnd, WM_NCACTIVATE, (WPARAM) TRUE, 0);
}
    break;
...
```

WM_NCACTIVATE 消息的语法相当简单，只需在 wParam 里设置一个 TRUE，便可强迫标题栏采用活动颜色。假如设置成 FALSE，标题栏则会恢复非活动状态。

WM_NCACTIVATE	0x0086
wParam	假如为 TRUE，就显示出活动标题栏；假如为 FALSE，则屏蔽它
lParam	未使用

在这条消息里，NC 来源于英语“nonclient”，即“非客户区”——表明这条消息是针对窗口的非客户区应用的。如图 7-8 所示，它显示的子窗口带有活动的标题栏——这是在它上面单击鼠标按钮的结果。

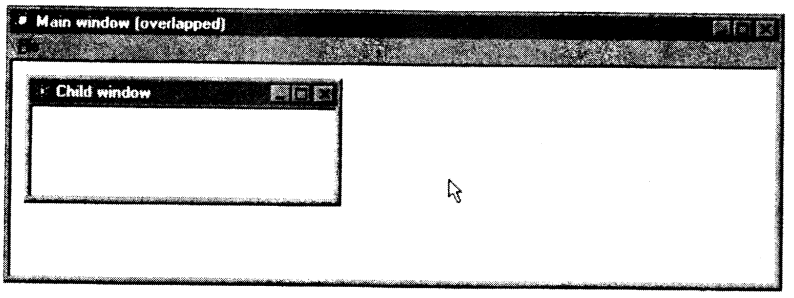


图 7-8　子窗口现在支持活动的标题栏

假如想取消活动状态，我们可以依靠 WM_ACTIVATE 消息来实现。在叠置式窗口进程里捕获了这条消息以后，就可以在父窗口丧失视觉焦点后屏蔽子窗口。

WM_ACTIVATE	0x0006
LOWORD（wParam）	活动标志。可以是下面这些标志中的一个：WA_ACTIVE，WA_INACTIVE，WA_CLICKACTIVE
HIWORD（wParam）	假如为一个非零值，就表明该窗口正以最小化显示
lParam	激活或者屏蔽窗口的句柄

叠置式窗口处于非活动（屏蔽）状态以后，我们可以把 WM_NCACTIVATE 的 wParam 设置成 FALSE，这样很容易就可以实现对子窗口标题栏的屏蔽。

```
...
case WM_ACTIVATE:

    //are we deactivating the overlapped window
    if(LOWORD(wParam) == WA_INACTIVE)
    {
        //remove the blue title bar form the child window
        SendMessage(CTRL(hwnd, CT_CHILD), WM_NCACTIVATE,
          (WPARAM)FALSE, 0);
    }

    break;
...
```

父窗口被选中以后，强制性地自动选择子窗口是一种相当直接的操作。首先，从子窗口进程里删除 WM_MOUSEACTIVATE 情况。然后通过对 WM_ACTIVE 条件进行检测，从而扩展 WM_ACTIVATE 情况。

在 PARTY1 里，我们还有最后一个问题需要注意。请大家注意图 7-9 内的任务栏，你发现了什么？叠置式窗口有自己的一个任务栏按钮，另外一个按钮则是用于弹出式窗口的，但是子窗口却没有使用任何一个任务栏按钮。

7.5.2 一起来聚会！版本 2

接下来这个 PARTY 的修订版本只是对程序的整体结构进行了几处改动。

正如我们前面曾经提及的那样，用连续三个 CreateWindowEx () 调用分别建立叠置式窗口、弹出式窗口以及子窗口是相当不明智的。更简便的一种做法是，我们在 WinMain () 里只建立应用程序的一个主窗口，然后把另外两个窗口移至叠置式窗口进程里。这就是 PARTY1 的设计思想。

建立子窗口和弹出式窗口最恰当的地方在 WM_CREATE 消息里。这条消息是我们在 WinMain () 里调用 CreateWindowEx () 时生成的一种副产物。并非 WinMain () 内所有可用的信息都能在叠置式窗口进程里找到。从根本上说，我们需要的是应用程序实例句柄：hInstance，它是程序启动后从 Windows 内部直接发出的。

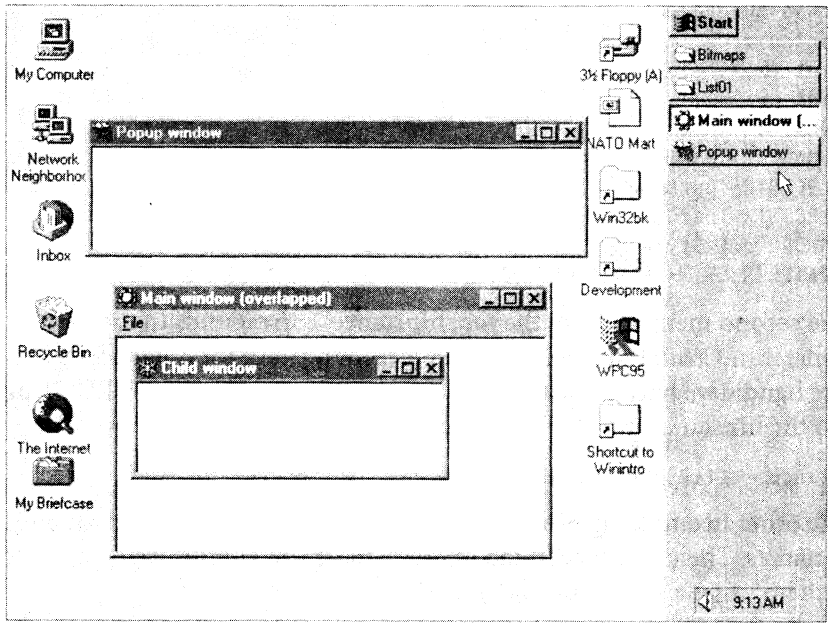


图 7-9 在系统任务栏内，只有弹出式和叠置式窗口才有对应的任务栏按钮；子窗口没有这种资格

至少出于对两方面原因的考虑，我们才认为 WM_CREATE 是建立另外两个窗口的最佳场所。首先，这条消息是在建立应用程序主窗口时自动生成的。其次，叠置式窗口在屏幕上显示出来之前，这条消息就会抵达窗口进程，从而使我们有机会定制应用程序的行为。WM_

CREATE 消息的语法结构如下所示：

WM_CREATE	0x0001
wParam	未使用
lParam	指向 LPCREATESTRUCT 数据结构一个指针

lParam 是当作指向 LPCREATESTRUCT 数据结构的一个指针使用的。这种数据结构内包含了 API 函数 CreateWindowEx（）的所有参数：

```
typedef struct tagCREATESTRUCT
 { // cs
    LPVOID lpCreateParams;
    HINSTANCE hInstance;
    HMNEU hMenu;
    HWND hwndParent;
    int cy;
    int cx;
    int y;
    int x;
    LONG style;
    LPCTSTR lpszName;
    LPCTSTR lpszClass;
    DWORD dwExStyle;
} CREATESTRUCT, *LPCREATESTRUCT;
```

从上往下数第二个项是 hInstance，它与 CreateWindowEx（）内倒数第二个参数是对应的。所以，如果想取得应用程序的实例句柄，我们必须让 lParam 成为 CREATESTRUCT 的一个指针，使其指向 hInstance 项，如下所示：

```
hInstance = ( (LPCREATESTRUCT) lParam) -> hInstance;
```

另外两种丢失的信息是弹出式窗口和子窗口的名称。由于它们存在于资源文件里，所以我们可以通过对 LoadString（）的一次标准调用，从而把它们载入：

```
...
case WM_CREATE:
{
    char szChildClass[30];
    char szPopupClass[30];
    HWND hwndChild, hwndPopup;
```

```
    hInstance = ((LPCREATESTRUCT)lParam) -> hInstance;

    LoadString(hInstance, ST_CHILDCLASS, szChildClass,
      sizeof(szChildClass));
    LoadString(hInstance, ST_POPUPCLASS, szPopupClass,
      sizeof(szPopupClass));

    hwndChild = CreateWindow(WS_EX_CLIENTEDGE | WS_EX_
      WINDOWEDGE,
                             szChildClass,
                             szChildClass,
                             WS_CHILD | WS_THICKFRAME |
                               WS_CAPTION | WS_SYSMENU |
                               WS_MINMAX,
                             10, 10,
                             170, 80,
                             hwnd,
                             (HMENU)CT_CHILD,
                             hInstance,
                             NULL);

    ShowWindow(hwndChild, SW_SHOWNORMAL);

    hwndPopup = CreateWindow(szPopupClass,
                             szPopupClass,
                               WS_POPUPWINDOW | WS_THICKFRAME |
                               WS_CAPTION |
                               WS_SYSMENU |WS_MINMAX,
                             210, 10,
                             170, 80,
                             NULL,
                             NULL,
                             hInstance,
                             NULL);

    ShowWindow(hwndPopup, SW_SHOWNORMAL);
}
    break;
...
```

拦截了 WM_CREATE 消息后，我们就可以很方便地从一个 CREATESTRUCT 对象里挑选自己需要的某些信息。在这儿，一个很容易犯的错误是认为 WM_CREATE 具备的行为能够扩展应用于其他任何一条消息——这是一种错误的推断。CREATESTRUCT 信息块只能在 WM_CREATE 消息通过窗口进程传递的时候才能使用，在这以后，系统就会自动把它删除掉。假如出于某种原因，我们需要保存由 CREATESTRUCT 对象提供的信息，使这种信息在整个窗口进程里均可使用，就必须把它拷贝到以前分配的内存块里。在现实的编程环境下，我们没有很大的必要采取这样的行动，大家通过本书提供的例子就可以看出这一点。

PARTY2 看起来与 PARTY1 是完全一致的，然而它们的内部结构却存在很大的区别。大家可以在本书附带 CD 的 Listing 7.2 里找到 PARTY2 的源代码。

我们现在该揭开 Windows 编程最神秘的面纱了——即 CreateWindowEx（）的最后一个参数。PARTY3 将帮助我们解开心中这个最大的疑团。

7.5.3 一起来聚会！版本 3

这是 PARTY 程序第三个，也是最后一个修订版本。

CreateWindowEx（）最后那个参数是指向某个空对象的一个指针，如下所示：

```
hwnd = CreateWindowEx (… , LPVOID lpParam);
```

在联机帮助文档里，这个参数被解释为“……窗口建立数据的地址”。正如我们在前面的例子里已经证明过的那样，这个参数在窗口建立过程中是没有任何用处的。更准确地说，微软的工程师们为开发者提供了一种特殊的途径，利用这种途径就可以把应用程序的某些相关信息自动传送给准备构建的那个窗口的窗口进程。

正如大家在 PARTY2 示范程序里看到的那样，一个 CREATSTRUCT 里包含了和 CreateWindowEx（）函数相同数目的参数。每一项都与 CreateWindowEx（）的某个参数对应。因为这一规则对于 hInstance 是有效的，所以它也适用于 CreateWindowEx（）内的最后一个参数。

CREATESTRUCT 的 lpCreateParams 项等效于 CreateWindowEx（）的 lpParam 参数。因此，假如我们把 lpParam 设置成 NULL，就相当于把 lpCreateParams 也设置成了 NULL。这样造成的结果就是，分配给 lpParam 的任何值都能在窗口类的窗口进程里访问。我们只需简单捕获 WM_CREATE 消息，然后指向 CREATESTRUCT 对象内的 lpCreateParams 项即可。我把这种技术称为“数据走私”。

在 PARTY3.H 里，我定义了一种新的数据结构，把它命名为 DATA。DATA 里包含了主窗口进程用于建立子窗口和弹出式窗口所需的全部信息：

```
typedef struct _DATA
{
    char szChildClass[20];
    char szPopupClass[20];
    char szChildText[20];
    char szPopupText[20];
} DATA, * PDATA;
```

在 WinMain () 里，PARTY3 用一个 Data 标识符的四个项存储了两个类的名称以及窗口标题：

```
...
DATA Data ;
...
LoadString(hInstance, ST_STRING + 0, (LPSTR)&Data.szChildClass, sizeof(
Data.szChildClass));
LoadString(hInstance, ST_STRING + 1, (LPSTR)&Data.szPopupClass, sizeof(
Data.szPopupClass));
LoadString(hInstance, ST_STRING + 2, (LPSTR)&Data.szChildText, sizeof(
Data.szChildText));
LoadString(hInstance, ST_STRING + 3, (LPSTR)&Data.szPopupText, sizeof(
Data.szPopupText));
...
```

在主窗口的建立过程中，PARTY3 把 Data 标识符的地址当作 CreateWindowEx () 的最后那个参数进行传递：

```
...
hwnd = CreateWindowEx(WS_EX_CLIENTEDGE | WS_EX_WINDOWEDGE,
                      szClassName,
                      szWindowTitle,
                      WS_OVERLAPPEDWINDOW|WS_CLIPCHILDREN,
                      10, 300,
                      CW_USEDEFAULT, 0,
                      NULL,
                      LoadMenu(hInstance, "mainmenu"),
                      hInstance,
                      (LPSTR) &Data);
...
```

在窗口进程里拦截 WM_CREATE 消息的时候，PARTY3 把值存储到 lpCreateParams 里。这个值具体存储在属于类型 PDATA 的一个 pData 标识符里，这个标识符是在子窗口和弹出式窗口建立好后(下面那个代码段未对此进行具体说明)，使用存储于 DATA 结构内的信息生成的：

```
...
hwndChild = CreateWindow(pData -> szChildClass,
```

```
                                pData -> szChildText,
                                WS_CHILD | WS_THICKFRAME |WS_CAPTION |
                                    WS_SYSMENU | WS_MINMAX,
                                10, 10,
                                170, 80,
                                hwnd,
                                (HMENU)CT_CHILD,
                                hInstance,
                                NULL);
...
```

在上面的例子中，CreateWindow () 的最后一个参数设置成 NULL，但是大家也可以用它传递另外一个内存块的地址或者 pData 本身，前提是有必要对 pData 在子窗口进程内指定的信息进行访问。这种把数据块从一个函数传递给另外一个函数的技术是非常有用的，我们将在本书剩余的部分里不断采用这种技术，甚至处理对话框的时候亦是如此。作为另一种选择，也可以声明源文件的范围标识符。然而，正如大家已经理解的那样，源文件范围标识符的采用会降低文件列表的整体可靠性，从而使其难于调试。

大家可在本书附带 CD 的 Listing 7.3 内找到 PARTY3 的源代码。

7.6 OWNER 弹出式窗口示例

OWNER 示范程序对两个弹出式菜单的行为进行了探索，一个有物主，另一个则没有。图 7-10 显示了 OWNER 程序执行后的三个窗口。叠置式窗口就是某个弹出式窗口的物主窗口。大家能猜出是哪个弹出式窗口吗？

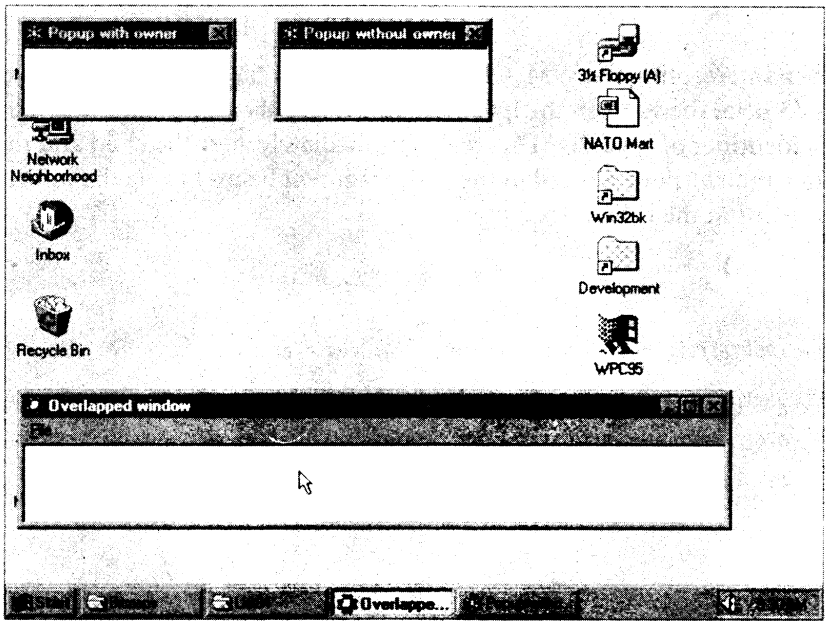


图 7-10 OWNER 程序阐述了带有物主和不带物主的两个弹出式窗口在行为上的差异

以最大化显示叠置式窗口的时候，不带物主的弹出式窗口就会从屏幕上消失掉，而另外一个则会在物主窗口的顶部继续保持其可见性，如图 7-11 所示。物主窗口最小化显示后（在 Windows 95 里，这意味着从屏幕上消失掉），弹出式窗口也会同时消失掉，如图 7-12 所示。

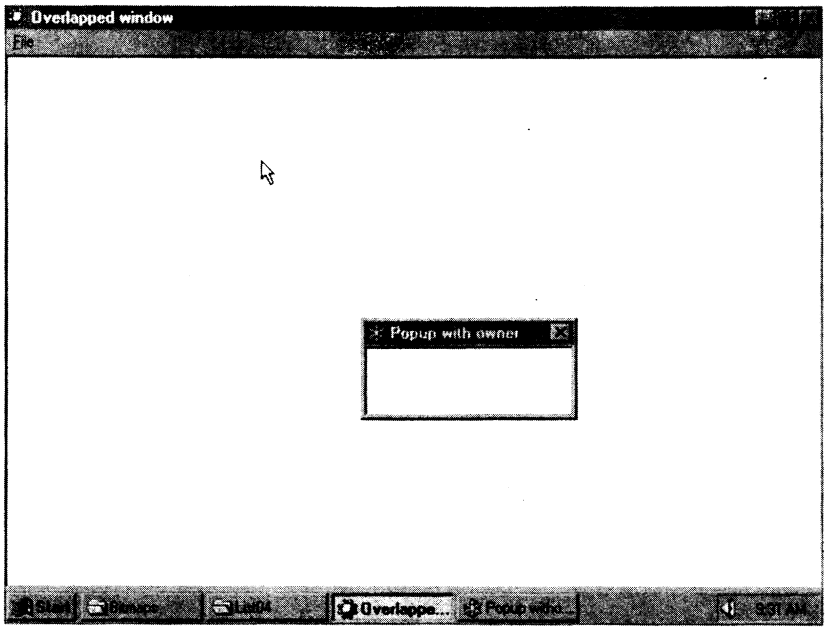


图 7-11　带有物主的弹出式窗口不会从屏幕内消失

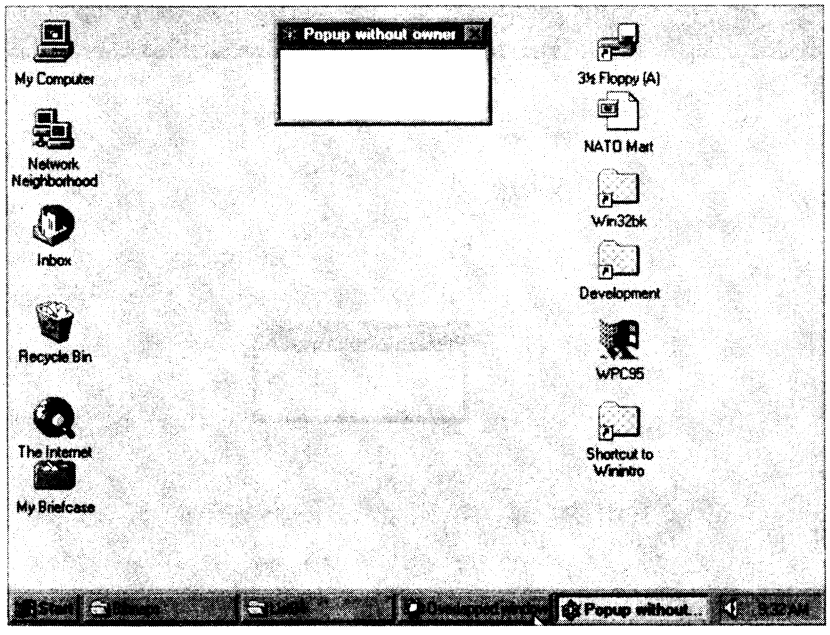


图 7-12　物主窗口最小化显示的时候，弹出式窗口也逃不脱相同的命运，尽管在这两个窗口之间并不存在物理性的链接关系

通过检查OWNER示范程序的源代码，我们发现当物主窗口建立两个弹出式窗口的其中之一时，传递的唯一信息就是叠置式窗口的句柄。

现在出现了一个明显的问题。带有物主的那个窗口如何才能取得自己物主窗口的句柄呢？调用GetParent()即可！我并非开玩笑——GetParent()的确能够为带有物主的一个弹出式窗口返回物主窗口的句柄。假如弹出式窗口没有物主，返回值就会是一个NULL，或者HWND_DESKTOP。为了避免GetParent()这个名字可能带来的混乱，我们还可以选择GetWindow()，其中的hwnd标识符指定了物主窗口：

```
hwndOwner = GetWindow (hwnd, GW_OWNER)
```

正如我们预期的那样，这种操作从另外一个角度来说是相当复杂的，这是由于我们没有办法决定带有物主的那个弹出式窗口的句柄。这时，GetLastActivePopup()看起来好象是我们唯一的救命稻草，然而它实际上会向我们反馈回错误的信息：

```
#include <winuser.h>
HWND GetLastActivePopup (HWND hwnd);
```

这个函数唯一的参数就是物主窗口的句柄。假如这个窗口没有包含弹出式窗口，假如它就是最近处于活动状态的窗口，假如hwnd并非一个顶级窗口或者该窗口本身就从属于另一个物主窗口，在所有这些情况下，GetLastActivePopup()都会返回hwnd参数。因此，该函数相当于从GetLastActiveWindow()里返回相同的句柄，并非返回一个真正的弹出式窗口句柄。

这个函数的行为是非常危险的，造成的结果也是不可预知的。所以我们应该果断地舍弃这根“救命稻草”！我们唯一可行的方案是设计一系列物主窗口的一张列表，把与这些窗口相关的句柄存储于物主窗口一个易于访问的位置处。随着SDI开发模式的引入，这种方案最终会显示出它的重要性。因为在SDI模式下，独立的窗口（无论弹出式还是叠置式）将逐渐取代子窗口的地位。

大家可在本书附带CD的Listing 7.4内找到OWNER示范程序的源代码。

7.7 窗口坐标

每个窗口的显示起点都是其左上角，并且肯定是与它的父窗口客户区有关的。对于叠置式窗口来说，它的位置与屏幕的左上角位置有关，因为这种窗口的显示象素是直接从桌面取得的。相同的规则亦适用于弹出式窗口——无论它是不是带有一个物主。子窗口的显示行为最独特。对于子窗口来说，它的显示位置与父窗口的座标空间有关。换言之，也就是与父窗口客户区的左上角有关。子窗口的显示起点肯定与父窗口在屏幕上的位置有关，如图7-13所示。

X轴的正值是从左到右延展的，而Y轴则是从顶部向底部延展。这就解释了为什么RECT结构定义一个矩形的时候要提供矩形的左上角坐标和右下角坐标。有些API可以让开发者改变坐标的起点和它在两个轴上的延展（SetWindowOrgEx()和SetWindowExtEx()）。对于多个窗口来说，由于存在多个起点，所以我们需要把屏幕上的某个点从一种坐标系统转换到另一种坐标系统内。为了实现这一目标，我们可以调用ClientToScreen()和ScreenToClient()。第一个函数能够把显示点从客户区的坐标空间转换到系统空间，第二个

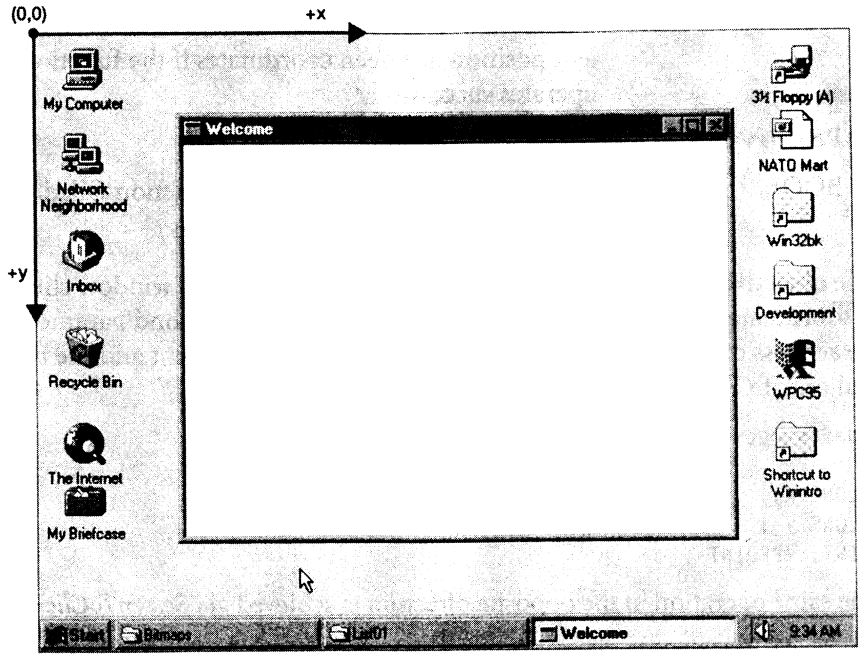


图 7-13　窗口的坐标空间向右和向下增长

则正好相反。

```
#include〈winuser.h〉
BOOL WINAPI ClientToScreen (HWND hwnd, LPPOINT lpPoint);
```

参数	**说明**
HWND hwnd	准备对自己的客户区坐标进行转换的那个窗口的句柄
LPPOINT lpPoint	某个 POINT 数据结构的地址，其中包含了用客户区坐标表示的一个点；假如函数调用成功，其中还会包含屏幕坐标内等效的位置
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

这个函数的使用是把某个点从窗口客户区坐标空间转换成系统坐标空间。第二个参数代表了一个 POINT 数据结构的地址，该结构内包含了原始点的坐标；假如调用 ClientToScreen () 函数成功，那么还会包含转换过的坐标信息。

```
typedef struct tagPOINT
 {
    LONG x;
    LONG y;
```

```
} POINT, *PPOINT;
```

利用 ScreenToClient () 则可以实现逆转换：

```
#include <winuser.h>
BOOL WINAPI ScreenToClient (HWND hwnd, LPPOINT lpPoint);
```

参数	说明
HWND hwnd	一个窗口的句柄，该窗口的客户区正是准备转换的目标坐标空间
LPPOINT lpPoint	某个 POINT 数据结构的地址，其中包含了用屏幕坐标表示的一个点；假如函数调用成功，其中还会包含客户区坐标内等效的位置
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

Win32 提供一种更为简便的方式，利用它可以把一个或多个点从一种坐标系统方便地转换成另外一种坐标系统，用不着通过屏幕进行中转。MapWindowPoints () 函数能帮助我们有效地完成这一工作：

```
#include <winuser.h>
int WINAPI MapWindowPoint (HWND hWndFrom,
                           HWND hWndTo,
                           LPPOINT lpPoints,
                           UINT cPoints);
```

参数	说明
HWND hWndFrom	包含准备转换的那个点的窗口的句柄
HWND hWndTo	目标窗口的句柄
LPPOINT lpPoints	一个 POINT 数据结构的地址，或者包含了待转换点的一个 POINT数组的名称
UINT cPoints	准备转换的点的数目
返回值	在正文里讨论
int	转换过程中在 X 轴（低字）和 Y（高字）上增加/减少的偏移距离

MapWindowRect () 宏是 MapWindowPoints () 这个基本 API 函数一种非常有用的扩展。它能把 RECT 数据结构从一个坐标空间转换到另一个坐标空间：

```
#include <windowsx.h>
#define MapWindowRect (hwndFrom, hwndTo, lprc) \
        MapWindowPoints ( (hwndFrom), (hwndTo), (POINT *) (lprc), 2)
```

客户区的大小

在窗口的建立过程中，API 函数 CreateWindowEx () 要求用到两种数值信息，它们分别

与窗口的显示位置以及在两个轴上的延长距离有关。

假设我们希望窗口在两个轴上都占据 200 个象素。在这儿，200 个象素是整体窗口的宽度和高度，其中包括由各种普通和扩展风格定义的所有非客户区组件。实际的客户区域肯定要小于 200 个象素，因为它的地盘已被一些非客户区元素侵占了。我们有几个办法可以计算出实际的客户区大小。最好的一个办法是在类窗口进程内拦截 WM_SIZE 消息。这条消息的 lParam 内封装了客户区的宽度和高度信息，并且在 wParam 里还存储了用于重定义尺寸的某些标志：

WM_SIZE	0x0005
wParam	用于重定义尺寸的标志
LOWORD (lParam)	客户区的宽度
HIWORD (lParam)	客户区的高度

其中，WINDEF.H 里定义的 HIWORD 和 LOWORD 是把 16 位值从 lParam 里提取出来的最佳工具：

```
...
case WM_SIZE:
 {
    int cx, cy;

    cx = (int) LOWORD (lParam);
    cy = (int) HIWORD (lParam);
    ...
}
...
```

在缺省情况下，客户区的左上角要窗口的起点是对应 (0, 0)，以后的值将向右和向下不断地增大，如图 7-13 所示。

每次对窗口尺寸进行变动的时候，就会通过 CreateWindowEx () 生成的初始消息流传送一条 WM_SIZE 消息。假如采用了 CS_HREDRAW 和 CS_VREDRAW 类风格，那么在水平和垂直方面上对窗口进行伸缩的时候，就会自动生成一条 WM_SIZE 消息。

计算客户区大小的另外一个办法是调用 API 函数 GetClientRect ()，这个函数的语法结果如下所示：

```
#include <winuser.h>
BOOL GetClientRect (HWND hwnd, LPRECT lpRect);
```

参数	**说明**
HWND hwnd	准备计算客户区尺寸的那个窗口的句柄
LPRECT lpRect	一个 RECT 数据结构的地址，其中包含了客户区左上角和右上角

	的位置信息
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

左上角的位置通常都是（0，0），除非开发者事先变动了窗口的起点位置。这儿的所有值都是用客户区坐标系统表达的。

7.8 窗口定位

假如在窗口标题栏的上方按下鼠标左键，并且按住不放，然后在屏幕上拖动鼠标，这样就可以实现窗口的移动。在这儿，标题栏是窗口建立的时候通过使用窗口风格 WS_CAPTION 来生成的。假如没有标题栏，用户就没有办法在屏幕上“抓住”和拖动这个窗口。这种限制并没有通过应用程序代码内部实施。任何窗口都可以强制移动到与自己父窗口相对的一个不同位置处，这是通过调用 MoveWindow（）函数来实现的——这并不是一个新函数，但是 Win32 里仍然得到了支持：

```
#include〈winuser.h〉
BOOL WINAPI MoveWindow（HWND hwnd，
                        int X，
                        int Y，
                        int nWidth，
                        int nHeight，
                        BOOL bRepaint）；
```

参数	说明
HWND hwnd	准备在屏幕内移动的那个窗口的句柄
int X	用父窗口坐标系统表示的左上角 X 轴坐标
int Y	用父窗口坐标系统表示的左上角 Y 轴坐标
int nWidth	新的窗口宽度
int nHeight	新的窗口高度
BOOL bRepaint	假如为 TRUE，那么窗口移动以后必须重画
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

MoveWindow（）不仅可用于定义窗口在屏幕上的新显示位置（第二和第三个参数），它也可以定义窗口的宽度和高度（第四和第五个参数）。因此，为这个函数分配的名称并没有表达出它所有的功能（仅有“移动窗口”的含义）。需要进行一个单纯的移动操作时，还需要一些附加的工作来决定当前的窗口尺寸。

GetWindowRect（）这个 API 可以帮助我们取得窗口的整体尺寸，然后把有关的信息存

储到一个 RECT 数据结构里。

```
#include〈winuser.h〉
BOOL GetWindowRect (HWND hwnd, LPRECT lpRect);
```

参数	说明
HWND hwnd	准备进行测量的那个窗口的句柄
LPRECT lpRect	一个 RECT 数据结构的地址，其中包含了窗口左上角和右下角的位置信息
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

由 GetWindowRect () 函数填写的 RECT 结构是指整个窗口表面的尺寸，其中包括所有非客户区组件。作为另一种选择方案，我们也可以拦截 WM_NCCLACSIZE 消息。每次对窗口进行伸缩处理的时候，就会生成这条消息。lParam 指向包含了整体窗口尺寸的一个 RECT 数据结构。

WM_NCCALCSIZE	0x0083
wParam	假如为 FALSE，表明 lParam 指定的是一个 RECT 数据结构；假如为 TRUE，则表明 lParam 是指向 NCCALCSIZE_PARAMS 结构的一个指针
lParam	既可以是一个 RECT 的地址，也可以是一个 NCCALCSIZE_PARAMS 结构

wParam 值为 FALSE 的前提下，由 lParam 指定的就是一个 RECT 数据结构，其中包含了窗口左上角和右下角坐标位置。更为有趣的一种情况在于，假如把 wParam 设置成 TRUE，那么 lParam 就会指向一个 NCCALCSIZE_PARAMS 结构，它的特征是包括了三个 RECT 结构和一个 WINDOWPOS 结构。

```
typedef struct _NCCALCSIZE_PARAMS
{ // nccp
    RECT rgrc [3];
    PWINDOWPOS lppos;
} NCCALCSIZE_PARAMS;
```

其中，第一个 RECT 结构指定了窗口的新尺寸；第二个指定了窗口移动或者重新定位前的尺寸；而第三个则指定了窗口客户区域的尺寸。因此，通过对这三个 RECT 进行检查，我们就能知道当前和上一个窗口的位置以及最后一次改变之前客户区的尺寸。除此以外，WINDOWPOS 结构内还包含了与尺寸和位置值有关的其他信息，这些尺寸和位置值是在窗

口移动或者重新定位过程中指定的。

```
typedef struct _WINDOWPOS
{    // wp
    HWND hwnd;
    HWND hwndInsertAfter;
    int x;
    int y;
    int cx;
    int cy;
    UINT flags;
} WINDOWS;
```

通过对 WM_NCCALCSIZE 消息进行拦截，我们就可以取得关于一个窗口的某些特定信息，就像下面这个代码段显示的那样。其中，WM_xxx 是一条普通的消息，它的作用是对一个窗口的位置和尺寸进行重新定义：

```
...
static int cx, cy;
...
switch (msg)
{
    ...
    case WM_NCCALCSIZE:
    {
        if (! wParam)
        {
            cx = LOWORD (lParam);
            cy = HIWORD (lParam);
        }
    }
        break;

    case WM_xxx:
     {
        POINT pt;
        ...
        MoveWindow (hwnd,
                    pt.x,
```

```
                    pt.y,
                    cx,
                    cy,
                    TRUE);
            ...
        }
        ...
    }
```

指向 RECT 结构的第二个参数

C 语言的初学者要注意：GetWindowRect () 和 GetClientRect () 这两个函数都需要指向一个 RECT 结构的指针，将其作为自己的第二个参数使用。这两个函数的语法很容易令人产生混淆，特别是那些在 C 语言编程上涉足不深的人士。根据我授课的经验，我注意到许多人都爱犯一个错误，他们传递的是指向 RECT 结构的一个真实的指针，而不是 RECT 类型的一个标识符的地址。假如声明类型为 LPRECT 的一个指针，就不会针对窗口尺寸的存储分配任何内存区域，而只是用于指针本身的四个字节：

```
...
LPRECT prect;

GetWindowRect (hwnd, prect);
...
```

这一部分代码生成了一个异常事件，因为 prect 指针指向的是进程地址空间内的一个随机位置。那并不是我们真正需要的。为了避免这类错误的产生，我们应该随时记住 API 需要的是指向对象的一个指针，这实际意味着开发者必须传递属于那种类型的一个标识符的地址，就像下面这样：

```
...
RECT rc;

GetWindowRect (hwnd, &rc);
...
```

移动，移动，再移动

WINPOS 示范程序向大家阐明了前面介绍的技术。这个 Win32 程序将显示出一个叠置式窗口和一个子窗口，这两个窗口分别从属于不同的窗口类。假如在主窗口客户区内任何一个地方按下鼠标左键，整个窗口都会移至那个屏幕位置。同时，窗口的左上角正好就是鼠标光标指向的位置。假如用户在子窗口的客户区域内单击鼠标左键，那么相同的规律亦会应用于子窗口。如图 7-14 所示。

这个非常简单的例子让我们有机会实践某些 API 函数的用法。新的窗口位置肯定是用与父窗口相对的窗口坐标表示的。在这儿，叠置式窗口的父窗口就是桌面；而子窗口的父窗口

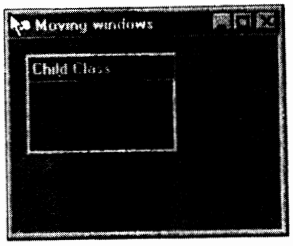


图 7-14　在子窗口或者叠置式窗口的客户区按下鼠标左键，WINPOS 就会把指定的窗口重新定位到那个屏幕位置处

则是主窗口)。这样一来，目标位置就要转换到父窗口的坐标空间里，这种转换是在 FollowTheMouse () 函数里通过对 MapWindowPoints () 的调用来实现的。按下鼠标左键的时候，假如 hwnd 标识了主窗口或者子窗口，那么无论哪个窗口进程都会调用 MapWindowPoints () 函数：

```
MapWindowPoints (hwnd, GetParent (hwnd), &pt, 1);
```

在本书附带 CD 的 Listing 7.5 内，大家可以找到 WINPOS 示范程序的源代码。该程序能把叠置窗口和它的子窗口移动到用户单击鼠标左键时的位置。

7.9　窗口的重定位

除了 MoveWindow () 以外，Win32 还另外提供了几种可选的方案，用于重新定义窗口在屏幕上的位置。其中最常用的也许要算 SetWindowPos ()。这个函数的应用非常灵活，它同时提供了数种不同的功能。通过名字推断出来的只是该函数的一部分功能而已。

```
#include <winuser.h>
BOOL WINAPI SetWindowPos (HWND hwnd,
                          HWND hWndInsertAfter,
                          int X,
                          int Y,
                          int cx,
                          int cy,
                          UINT uFlags);
```

参数	说明
HWND hwnd	准备在屏幕上改变位置的那个窗口的句柄
HWND hWndInsertAfter	用 Z 序列标识准备重新定位的那个窗口之前一个窗口。通常把这个参数设置成 HWND_TOP
int X	与左上角相关的 X 坐标
int Y	与左上角相关的 Y 坐标
int cx	窗口宽度
int cy	窗口高度

UINT uFlags	SWP_标志
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

第一个参数代表了准备在屏幕上移动或者重新定位的那个窗口的句柄。Windows 95 是根据一种三维基础对窗口进行管理的，它把最后一个建立或者选择的窗口放置于这种假想的窗口堆的最上层。SetWindowPos（）的第二个参数定义了和屏幕上另一个窗口的相对位置。这种信息可以是实际的窗口句柄，也可以是表 7-5 列出的某种定义。

表 7-5　SetWindowsPos（）第二个参数使用的定义

定　义	值	说　明
HWND_TOP	((HWND)0)	窗口在前台显示
HWND_BOTTOM	((HWND)1)	把窗口发送到后台
HWND_TOPMOST	((HWND)-1)	窗口在 Z 序列的最顶部，即使处于非活动状态时也保持这种属性
HWND_NOTOPMOST	((HWND)-2)	窗口放置于其他所有窗口的顶部，只是位于最顶部窗口的下面

一个最顶部（TOPMOST）的窗口总能维持它在 Z 序列里的位置，甚至在它处于非活动状态时也是如此，这与 Windows 95 的基本行为是无关的。这种属性可以在建立一个新窗口的时候通过 WS_EX_TOPMOST 扩展风格进行分配。接下来的四个参数定义了新窗口的位置和尺寸。第七个参数特别引人注目。这是一个或者多个 SWP_标志的组合，它们指出了 SetWindowPos（）将要进行的操作，如表 7-6 所示。

表 7-6　用于 SetWindowPos（）的 SWP_标志

标　志	值	说　明
SWP_NOSIZE	0x0001	不改变窗口尺寸，忽略 cx 和 cy 参数
SWP_NOMOVE	0x0002	不改变窗口位置，忽略 x 和 y 参数
SWP_NOZORDER	0x0004	不考虑 hwndAfter 参数
SWP_NOREDRAW	0x0008	客户区不进行重画
SWP_NOACTIVATE	0x0010	窗口未被激活
SWP_FRAMECHANGED	0x0020	向窗口发送一条 WM_NCCALCSIZE 消息，即使窗口尺寸已被改变
SWP_SHOWWINDOW	0x0040	显示窗口
SWP_HIDEWINDOW	0x0080	隐藏窗口
SWP_NOCOPYBITS	0x0100	抛弃客户区的所有内容，窗口重定位结束以后立即进行连续重画

SetWindowPos（）的运作是按照一种奇妙的逻辑进行的：假如不想改变窗口的大小，就必须使用 SWP_NOSIZE 标志。相同的规则亦适用于窗口重定位和 SWP_NOMOVE。除此以外，SetWindowPos（）还包含了由 ShowWindow（）提供的功能，这是由其中的 SWP_SHOWWINDOW 和 SWP_HIDEWINDOW 标志实现的。

改变窗口的位置、尺寸或者 Z 序列内的位置之前，SetWindowPos（）会向目标窗口发送一条 WM_WINDOWPOSCHANGING 消息。

WM_WINDOWPOSCHANGING	0x0046
wParam	未使用
lParam	一个 WINDOWPOS 结构的地址

通过对这条消息进行拦截，我们仍有机会对调用 SetWindowPos () 时设置的值进行更改。通常情况下，在真正的重定位操作发生之前，我们都没有必要重新设置一个新的窗口位置或者改变它的大小。更准确地说，我们在重定位操作完成以后，假如再对一个窗口接收到的 WM_WINDOWPOSCHANGED 消息进行捕获，那么这种方式才是最简便的。对 WINDOWPOS 结构内的各个项进行检查，这样我们才有机会决定窗口的新特征。

7.10 一次性定位多个窗口

根据 MDI 开发规范，它需要对几个子窗口进行频繁的重新定位。举个例子来说，MS Excel 允许用户在 Window 菜单内选择 Arrange All 选项，从而对所有处于活动状态的文档进行重定位——所有这些只需要一次单独的操作即可完成。在 Windows 95 里，外壳本身就具有类似的功能。假如在任务栏上面单击鼠标右键，就会显示出一个小的弹出式菜单，如图 7-15 所示。

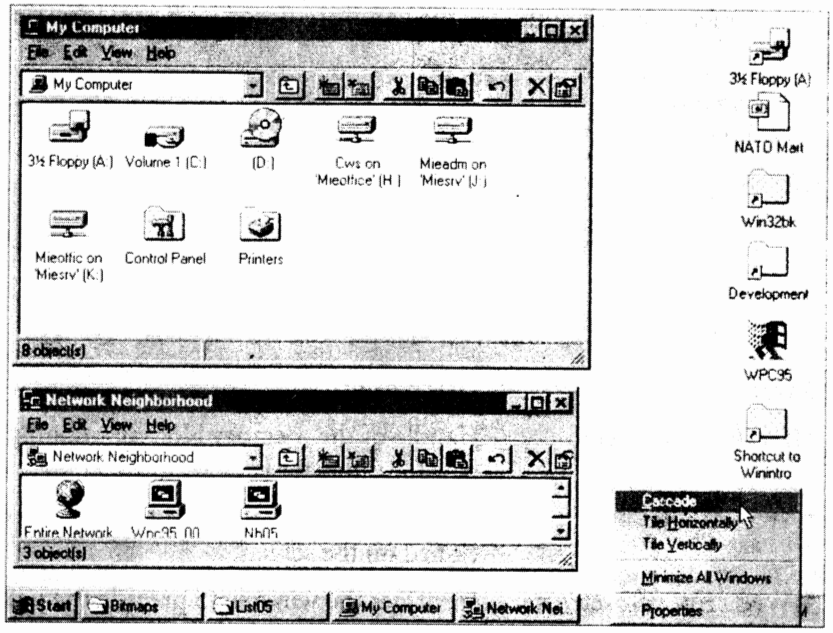


图 7-15 任务栏弹出式菜单提供了三个选项，利用它们可以对桌面上打开的所有窗口进行快速重定位

通过对 SetWindowPos () 的几次调用，弹出式菜单内的这些选项都是可以实现的，其中一项就是对窗口进行重定位。假如窗口在屏幕上是一个一个叠置起来的（叠置选项），那么这种方案就会导致性能上的缺陷。在这种情况下，重新定位一个窗口意味着同时要改变它的尺寸——换言之，就是要对整个客户区进行更新重画。假如窗口内显示了容量很大的信息，那

么这种操作就会“吃”掉大量 CPU 时间。除此以处，根据叠置算法，下一个窗口的重新定位会把前一个窗口的大部分都隐藏起来，从而使上一次重画处理变成无劳之举。

假如几个窗口要求在各自的客户区域上进行叠置，那么我们最好还是更换一种方法，按照一定的顺序依次调用三个函数。这三个函数的效果与 SetWindowPos () 是基本一致的，然而却采用了一种更有效、更能节省 CPU 时间的方案。从理论上说，它们的考虑是先建立一个内存块，在其中包括与每个窗口的新尺寸和新位置有关的所有信息。这些准备工作做好以后，在屏幕上同时显示出所有窗口即可。按照这种思路，每个窗口都只会重画它实际显示出来的那一部分客户区域，从而节省了大量时间。

第一个步骤是调用 BeginDeferWindowPos ()：

```
#include〈winuser.h〉
HDWP WINAPI BeginDeferWindowPos (int nNumWindows);
```

在这个函数里，唯一的那个参数与屏幕上准备重新定位的窗口数对应。返回值是一个指定了某个内存块的句柄，这个内存块里存储了所有需要保留的信息。在这以后，我们就可以按照预先设定好的方案为每个窗口分配它在屏幕上的新位置和尺寸。为了实现这些处理，我们必须依靠 DefWindowPos () 函数，这个函数与 SetWindowPos () 的语法基本上是完全一致的。它的第一个参数对应于由 BeginDeferWindowPos () 返回的句柄：

```
#include〈winuser.h〉
HDWP WINAPI DeferWindowPos (HDWP hWinPosInfo,
                            HWND hWnd,
                            HWND hWndInsertAfter,
                            int x,
                            int y,
                            int cx,
                            int cy,
                            UINT uFlags);
```

参数	说明
HDWP hWinPosInfo	一个内存块的句柄，这个内存块内包含了每个窗口使用的新位置
HWND hWnd	准备在屏幕上重新定位的那个窗口的句柄
HWND hWndInsertAfter	用 Z 序列标识准备重新定位的那个窗口之前一个窗口。通常把这个参数设置成 HWND_TOP
int X	与左上角相关的 X 坐标
int Y	与左上角相关的 Y 坐标
int cx	窗口宽度
int cy	窗口高度
UINT uFlags	SWP_标志
返回值	在正文里讨论
HDWP	更新内存块的句柄，这个内存块包含了每个窗口使用的新位置

所有窗口在内存里的重新定位准备就绪以后，接下来就应该让用户看到这种重定位的效果。我们在这儿要用到 EndDefWindowPos ()：

```
#include <winuser.h>
BOOL WINAPI EndDeferWindowPos (HDWP hWinPosInfo);
```

其中的参数是指由 BeginDeferWindowPos () 分配好的内存块，其中的信息已经通过对 DeferWindowPos () 的几次调用成功地填写好了。下面这个代码段向大家阐示了如何运用这三个函数对五个窗口进行重新定位。注意，应用程序此时已经知道了这些窗口的句柄：

```
...
HDWP hdwp;
int i, nNumWidnows = 5;
...
// let's start repositioning the windows
hdwp = BeginDeferWindowPos (nNumWindows);

for (i = 0, i < nNumWindows; i++)
{
    // get the ith window handle
    ...

    // measure x, y, cx; and cy
    ...

    // repositiong the window in memory
    hwdp = DeferWindowPos (hdwp,
                           hwndFound,
                           HWND_TOP,
                           x, y,
                           cx, cy,
                           SWP_NOACTIVATE);
    ...
}

// show all the windows at once
EndDeferWindowPos (hdwp);
...
```

这儿仍然遗漏了一个细节：开发一个 SDI 应用程序的时候，我们怎样才能获得所有的窗

口句柄呢？在一个 MDI 应用程序里，所有的文档窗口都是子窗口，它们只能依附于父窗口的客户区才能显示出来。假设开发应用程序的方式正确，那么通过传递每个控件的对应 ID，几次 GetDlgItem（）函数调用即可很容易获得全部窗口句柄。在 SDI 环境下，由于叠置式和弹出式窗口取代了所有的子窗口，所以为了获取同样的信息，我们需要采取某些附加的操作。请大家参考本章早些时候讨论弹出式窗口的内容。

7.11 消息框的建立

你想要一个速度很快、很灵活、很可靠并且易于定制的窗口吗？在满足这些条件的同时，你还不想依次完成类注册、编写窗口进程以及最后建立属于那一类的某个窗口的所有这些繁琐的步骤，怎样才能满足你这些苛刻的要求呢？假如你真想这样做，那么消息框就是你正在寻找的东西。

消息框是一种标准的窗口，它是专门针对显示两个正文串、几个有特色的按钮以及一些装饰性的图标而专门设计的。MessageBoxEx（）函数的语法结构如下所示：

```
#include 〈winuser.h〉
int WINAPI MessageBoxEx (HWND hwnd,
                         LPCSTR lpText,
                         LPCSTR lpCaption,
                         UINT uType,
                         WORD wLanguageId);
```

参数	说明
HWND hwnd	物主窗口句柄
LPCSTR lpText	消息框客户区内显示的正文
LPCSTR lpCaption	标题栏正文
UINT uType	MB_标志，用于定义消息框内包含的按钮和图标的编号和种类
WORD wLanguageId	与按钮内显示的正文串有关的语言定义
返回值	在正文里讨论
int	假如函数调用不成功，就返回一个零值；或者是表 7-7 列出的某项定义

根据第一个参数推断，它好象应该是消息框的父窗口。然而，这是不正确的，因为消息框是一个弹出式窗口，它的象素来源肯定是屏幕。事实上，第一个参数代表的是消息框的物主窗口——消息框在屏幕中显示出来后，那个物主窗口就会变成非活动（屏蔽）状态。两个正文串代表了窗口的输出以及通过 wsprintf（）存储在一个缓冲区内的任何有效字符的组合。wsprintf（）是运行期函数 sprintf（）的一个“类 Windows”版本，如下所示：

```
#include 〈winuser.h〉
int WINAPIV wsprintf (LPSTR, LPCSTR, …);
```

表 7-7　用于定制消息框的 MB_标志

标　志	值	说　明
MB_OK	0x00000000L	建立一个 OK 按钮
MB_OKCANCEL	0x00000001L	建立 OK 和 Cancel 按钮
MB_ABORTRETRYIGNORE	0x00000002L	建立 Abort，Retry 和 Ignore 按钮
MB_YESNOCANCEL	0x00000003L	建立 Yes，No 和 Cancel 按钮
MB_YESNO	0x00000004L	建立 Yes 和 No 按钮
MB_RETRYCANCEL	0x00000005L	建立 Retry 和 Cancel 按钮
MB_ICONHAND	0x00000010L	显示一个红色的圆，其中带有一个红色的叉号
MB_ICONQUESTION	0x00000020L	显示带有一个问号的图标
MB_ICONEXCLAMATION	0x00000030L	显示一个黄色三角形，其中带有一个黑色的感叹号
MB_ICONASTERISK	0x00000040L	显示一个带有蓝色“i”字样的图标
MB_ICONINFORMATION	MB_ICONASTERISK	显示一个带有蓝色“i”字样的图标
MB_ICONSTOP	MB_ICONHAND	显示一个红色的圆，其中带有一个红色的叉号
MB_DEFBUTTON1	0x00000000L	第一个按钮是缺省按钮
MB_DEFBUTTON2	0x00000100L	第二个按钮是缺省按钮
MB_DEFBUTTON3	0x00000200L	第三个按钮是缺省按钮
MB_DEFBUTTON4	0x00000300L	第四个按钮是缺省按钮
MB_APPLMODAL	0x00000000L	建立一个应用程序模态消息框
MB_SYSTEMMODAL	0x00001000L	建立一个系统模态消息框
MB_TASKMODAL	0x00002000L	建立一个任务模态消息框
MB_HELP	0x00004000L	在消息框内增加帮助按钮
MB_SETFOREGROUND	0x00010000L	消息框成为前台窗口
MB_DEFAULT_DESKTOP_ONLY	0x00020000L	只适用于 NT。指定登录以后的用户桌面
MB_USERICON	0x00000080L	只能与 MessageBoxIndirect（）联合使用
MB_TOPMOST	0x00040000L	消息框作为最顶部的窗口
MB_NOFOCUS	0x00008000L	消息框不获得视觉焦点

7.11.1　定制消息框

通过表 7-7 列出的那些标志，我们可以对 MessageBoxEx（）的输出进行全方位的控制。

7.11.2　语言和子语言定义

MessageBoxEx（）的最后一个参数是两个定义的组合，这两个定义是在 WINNT.H 里定义的，具体的定义请参见表 7-8 和 7-9。包含基本语言定义的 32 字存放于低字内，而子语言定义则存放于高字内。这些定义是按照 MAKELANGID 宏的结构组合起来的：

```
#define MAKELANGID (p, s)       ((WORD) (s)) << (10) | (WORD) (p))
```

除了基础语言的指定以外，我们还有必要指定一种“变种语言”，它的含义是一种特定的方言或者语言修正。举个例子来说，在瑞士南部有一个讲意大利语的小社区——Ticino 人。因此，我们需要使用两个子语言定义 SUBLANG_ITALIAN（在意大利本土）和 SUBLANG_ITALIAN_SWISS（在瑞士），从而指出它们之间细微的差异。

表 7-8　基本的语言定义

语　言	值	语　言	值
LANG_NEUTRAL	0x00	LANG_ICELANDIC	0x0f
LANG_ALBANIAN	0x1c	LANG_ITALIAN	0x10
LANG_ARABIC	0x01	LANG_JAPANESE	0x11
LANG_BAHASA	0x21	LANG_KOREAN	0x12
LANG_BULGARIAN	0x02	LANG_NORWEGIAN	0x14
LANG_CATALAN	0x03	LANG_POLISH	0x15
LANG_CHINESE	0x04	LANG_PORTUGUESE	0x16
LANG_CZECH	0x05	LANG_RHAETO_ROMAN	0x17
LANG_DANISH	0x06	LANG_ROMANIAN	0x18
LANG_DUTCH	0x13	LANG_RUSSIAN	0x19
LANG_ENGLISH	0x09	LANG_SERBO_CROATIAN	0x1a
LANG_FINNISH	0x0b	LANG_SLOVAK	0x1b
LANG_FRENCH	0x0c	LANG_SPANISH	0x0a
LANG_GERMAN	0x07	LANG_SWEDISH	0x1d
LANG_GREEK	0x08	LANG_THAI	0x1e
LANG_HEBREW	0x0d	LANG_TURKISH	0x1f
LANG_HUNGARIAN	0x0e	LANG_URDU	0x20

表 7-8　子语言定义

子语言	值	说　明
SUBLANG_NEUTRAL	0x00	中立语言
SUBLANG_DEFAULT	0x01	缺省语言
SUBLANG_SYS_DEFAULT	0x02	系统缺省语言
SUBLANG_CHINESE_SIMPLIFIED	0x02	汉语（简体）
SUBLANG_CHINESE_TRADITIONAL	0x01	汉语（繁体）
SUBLANG_DUTCH	0x01	荷兰语
SUBLANG_DUTCH_BELGIAN	0x02	荷兰语（比利时）
SUBLANG_ENGLISH_US	0x01	英语（美国）
SUBLANG_ENGLISH_UK	0x02	英语（英国）
SUBLANG_ENGLISH_AUS	0x03	英语（澳大利亚）
SUBLANG_ENGLISH_CAN	0x04	英语（加拿大）
SUBLANG_ENGLISH_NZ	0x05	英语（新西兰）
SUBLANG_ENGLISH_EIRE	0x06	英语（爱尔兰）
SUBLANG_FRENCH	0x01	法语
SUBLANG_FRENCH_BELGIAN	0x01	法语（比利时）
SUBLANG_FRENCH_CANADIAN	0x03	法语（加拿大）
SUBLANG_FRENCH_SWISS	0x04	法语（瑞士）
SUBLANG_GERMAN	0x01	德语
SUBLANG_GERMAN_SWISS	0x02	德语（瑞士）
SUBLANG_GERMAN_AUSTRIAN	0x03	德语（奥地利）
SUBLANG_ITALIAN	0x01	意大利语
SUBLANG_ITALIAN_SWISS	0x02	意大利语（瑞士）
SUBLANG_NORWEGIAN_BOKMAL	0x01	挪威语（Bikma）
SUBLANG_NORWEGIAN_NYNORSK	0x02	挪威语（Nynorsk）
SUBLANG_PORTUGUESE	0x02	葡萄牙语
SUBLANG_PORTUGUESE_BRAZILIAN	0x01	葡萄牙语（巴西）
SUBLANG_SERBO_CROATIAN_CYRILLIC	0x02	塞尔维亚－克罗地亚语（西里尔体系）
SUBLANG_SERBO_CROATIAN_LATIN	0x01	克罗地亚－塞尔维亚语（拉丁语）
SUBLANG_SPANISH	0x01	西班牙语（古代语）
SUBLANG_SPANISH_MEXICAN	0x02	西班牙语（墨西哥）
SUBLANG_SPANISH_MODERN	0x03	西班牙语（现代语）

如果想提取出基本语言和子语言，我们可以使用通常在 WINNT.H 里定义的 PRIMARYLANGID 和 SUBLANGID 宏：

```
#define PRIMARYLANGID (lgid)    ((WORD)(lgid) & 0x3ff)
#define SUBLANGID (lgid)        ((WORD)(lgid) >> 10)
```

7.11.3 用按钮建立消息框

用 MessageBoxEx () 建立了几个按钮以后，应用程序就会捕获到该函数返回的值。用户的选择是用表 7-10 里列出的某个定义来指定的。

表 7-10 消息框内的按钮类型

定 义	值	定 义	值
IDOK	1	IDYES	6
IDCANCEL	2	IDNO	7
IDABORT	3	IDCLOSE	8
IDRETRY	4	IDHELP	9
IDIGNORE	5		

7.11.4 有趣的消息框

无论我们用什么标志对消息框进行定制，现在仍然不能在消息框里放置一个用户自定义的图标，或者对消息框的基本行为进行修改，从而设计出适用自己需要的一个消息框。通过调用 MessageBoxIndirect () 这个 API 函数，我们就能很容易地避开所有这些限制。这个函数的功能十分强大，然而使用却显得相对的简单，它可以根据 MSGBOXPARAMS 数据结构里传递的信息建立一个消息框。如果想在消息框内显示自己提供的一个图标，就可以把 dwStyle 项设置成 MB_USERICON 标志，同时使用相应的 MB_标志，如下例所示：

```
msgbox.dwStyle = MB_YESNO | MB_USERICON
```

接下来，我们为 lpszIcon 项分配一个图标资源的名字，这种资源是从资源文件里获得的。

```
msgbox.lpszIcon = " findwnd";
```

另外一种定制方法是采用一个回调函数，它能在用户按下 Help 按钮后进行调用。一个普通的消息框回调函数只需要用到一个参数：一个 HELPINFO 数据结构的地址，如下所示：

```
VOID CALLBACK MsgBoxCallback (LPHELPINFO lpHelpInfo);
```

一旦用恰当的信息把 MSGBOXPARAMS 结构设置好以后，接着就可以调用 MessageBoxIndirect () 来动态地生成消息框：

```
#include <winuser.h>
BOOL WINAPI MessageBoxIndirect (LPMSGBOXPARAMS lpMsgBoxParams);
```

参数	说明
LPMSGBOXPARAMS lpMsgBoxParams	一个 MSGBOXPARAMS 数据结构的地址
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

下面这个代码段向大家展示捕获了 MessageBoxIndirect () 函数的具体运用。用户选中了系统菜单内 Close 选项以后，就会生成相应的消息。捕获了这条消息以后，就可以利用这个函数对消息框进行定制。

```
...
case SC_CLOSE:
{
        char szText[] = "Do you really want to terminate?";
        char szCaption[] = "Terminate?";
        MSGBOXPARAMS msgbox;

        msgbox.cbSize = sizeof(msgbox);
        msgbox.hwndOwner = hwnd;
        msgbox.hInstance = hInstance;
        msgbox.lpszText = szText;
        msgbox.lpszCaption = szCaption;
        msgbox.dwStyle = MB_YESNO | MB_USERICON;
        msgbox.lpszIcon = "findwnd";
        msgbox.dwContextHelpId = 1;
        msgbox.lpfnMsgBoxCallback = NULL;
        msgbox.dwLanguageId = MAKELANGID(LANG_NEUTRAL,
            SUBLANG_NEUTRAL);

        if(MessageBoxIndirect(&msgbox) == IDNO)
            return FALSE;
}
        break;
...
```

它的输出结果如图 7-16 所示。

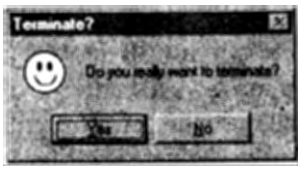


图 7-16　通过 MessageBoxIndirect () 生成的消息框内带有一个定制图标

7.12 一次运行一个程序拷贝

为了保证开发高质量的 Win32 应用程序，我们强烈建立大家只在执行过程中运行同一程序的一个拷贝。这一准则并不与系统的多任务本质相对立，相反，这样能提高用户与系统间进行交互作用的级别。

为什么要拟定这一条准则呢？假如在前台不断放置应用程序的运行拷贝，那么整个屏幕就会充斥大量程序，从而导致混乱。在本书附带 CD 的 Listing 7.6 里，大家能够找到一个名为 FINDWND 示范程序，该程序很容易就可以找出正在运行着同一个程序，这主要是通过调用 API 函数 FindWindow ()（在本章前面已经讨论过了，请参考图 7-17）来实现的。

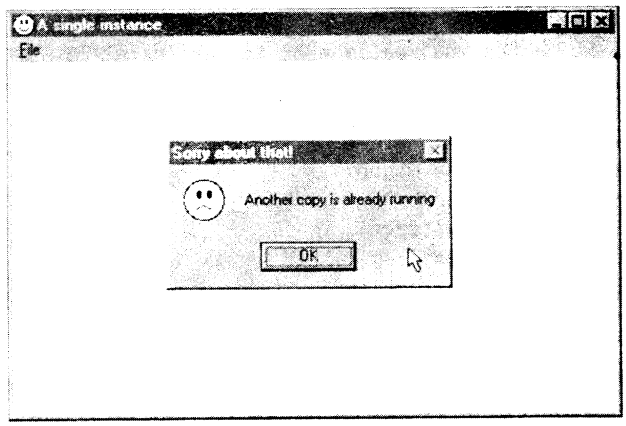


图 7-17 假如 FINDWND 侦测到了自己的另外一个拷贝，就会中断自己，并且把前一个拷贝带入前台

在注册窗口类之前，FINDWND 程序会寻找自己的另外一个拷贝。为了达到这个目的，它需要把类名和主窗口标题传递给 FindWindow ()：

```
hwndPrev = FindWindow (szClassName, szWindowTitle);
```

假如 hwndPrev 是一个非零的值，就意味着另外一个拷贝已在系统内处于活动状态。假如出现了这种情况，当前的拷贝自己就会中断，并且显示出一个消息框，指出自己将要中断。但是，在进行这些操作之前，它还会把另外一个拷贝带入前台，这种做法是 Windows 95 风格设计准则建议的。为了完成这一行动，前一个窗口将设置成前台窗口，然后进入 Z 序列的顶部，最后激活。在这以后，中断消息才会在屏幕上显示出来，提醒用户当前的状况，如图 7-18 所示。

消息框消失以后，前一个拷贝将恢复活动状态，第二个则中断了。

```
...
if(hwndPrev)
{
    MSGBOXPARAMS msgbox;
    char szText[] = "Another copy is already running";
```

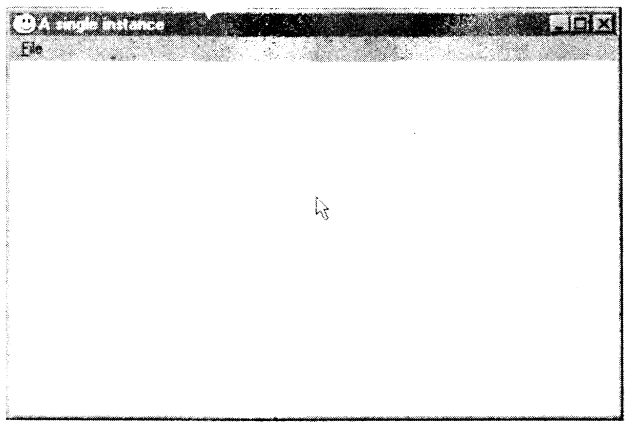


图 7-18　FINDWND 程序的第二个拷贝会立即中断，并且把
第一个也是系统唯一的一个拷贝带入前台

```
char szCaption[] = "Sorry about that!";

SetForegroundWindow(hwndPrev);
SetWindowPos(hwndPrev, HWND_TOP, 0, 0, 0, 0, SWP_NOMOVE |
    SWP_NOSIZE);
SetActiveWindow(hwndPrev);

msgbox.cbSize = sizeof(msgbox);
msgbox.hwndOwner = HWND_DESKTOP;
msgbox.hInstance = hInstance;
msgbox.lpszText = szText;
msgbox.lpszCaption = szCaption;
msgbox.dwStyle = MB_OK | MB_USERICON;
msgbox.lpszIcon = "sorry";
msgbox.dwContextHelpId = 1;
msgbox.lpfnMsgBoxCallback = NULL;
msgbox.dwLanguageId = MAKELANGID(LANG_NEUTRAL,
    SUBLANG_NEUTRAL);

if(MessageBoxIndirect(&msgbox))
    return FALSE;
...
```

对前一个运行拷贝的存在进行侦测的另外一个办法是使用“原子”。在这儿，“原子”是一个短的、2 字节的整数，它唯一性标识了整个系统内可用的一个正文串。Win32 内部保留了

可见原子的一份列表，每个应用程序均可对这份列表进行访问。第一个程序拷贝运行以后，它就会根据类名定义一个新的原子，它不会随着拷贝的增多而变化。第二个拷贝会检查与类名有关的原子是否已经存在。假如存在，就意味着自己是应用程序的第二份拷贝，所以就会立即中断。这种方案没有提供决定第一个拷贝窗口句柄的机会，从而避免了它在前台显示出来。

在这种特定的情况下，程序是把原子当作信号机的形式来使用的。接下来的一个例子将展示如何把一个信号机当作一种计数设备进行运用，从而限制运行拷贝的数目。

7.12.1 用信号机限制拷贝

大家现在至少已经知道了两种方法来防止相同程序的第二份拷贝的运行。但是怎样任意限制运行拷贝的数目呢？举个例子来说，我们应该采取什么办法把运行拷贝的数目限制在三个以内，但是又不超过三个呢？有一种简便的办法能帮助我们实现这一要求吗？我们可以把这种要求当作一种单纯的编程练习来进行。

为了实现这一要求，我们需要用到一种全局计数器，它对于任何运行拷贝来说都是“可见”的，并且可以简便地进行更新。Win32 提供了“进程间通信”（IPC）这种技术，通过它就可以计算出对某种共享资源的访问次数，并对未经许可的访问进行限制。我们利用这种系统对象来计算出运行的程序拷贝数目，从而对用户的操作进行跟踪。我们把这种对象称为“信号机”（Semaphore）。

从根本上说，信号机是一种计数设备。在 MAXINST 示范程序里（可在本书附带 CD 的 Listing 7.7内找到），我们建立了一个信号机，并把它最大计数值设定为 3，它的初始值则设定为 0。这就意味着它不允许 MAXINST 应用程序的运行拷贝数目超过三个。事实上，信号机与新进程的建立并没有直接的联系。在 MAXINST 程序里，信号计数器通过检查信号机的值来决定是否允许一个新的程序拷贝运行。

下面是建立信号机时使用的语句：

```
hsem = CreateSemphore (NULL, 0, MAXINSTANCES, " MAXINST");
```

其中，MAXINSTANCES 已经预先定义好了：

```
#define MAXINSTANCE3
```

第二个参数是计数器的初始值，而 MAXINST 则是分配给该信号机的名字。假如函数调用成功，就会返回信号机的句柄；假如失败，则返回一个 NULL 值（第 14 章“多线程、IPC 和 I/O”更加深入地探讨了关于信号机的问题）。在 MAXINST 这个例子里，信号机最初设置成 0，假如这个值达到了我们设定的上限（3），计数器就会返回一条出错信息。建立好信号机以后，信号计数器的值就会通过调用 ReleaseSemaphore () 函数立即增一，如下所示：

```
if (! ReleaseSemaphore (hsem, 1, &lCnt))
{
    MSGBOXPARAMS msgbox;
    char szText [] = " Max limit already passed";
    char szCaption [] = " Sorry about that!";

    msgbox.cbSize = sizeof (msgbox);
    msgbox.hwndOwner = HWND_DESKTOP;
```

```
        msgbox.hInstance = hInstance;
        msgbox.lpszText = szText;
        msgbox.lpszCaption = szCaption;
        msgbox.dwStyle = MB_OK | MB_USERICON;
        msgbox.lpszIcon = " sorry";
        msgbox.dwContextHelpId = 1;
        msgbox.lpfnMsgBoxCallback = NULL;
        msgbox.dwLanguageId = MAKELANGID (LANG_NEUTRAL,
            SUBLANG_NEUTRAL);

        if (MessageBoxIndirect (&msgbox))
            return FALSE;

        // closing the semphore
        CloseHandle (hsem);
        return FALSE;
    }
```

假如信号计数器已经达到了它的上限，这个函数就会返回一个 FALSE 值。第二个参数指出应在信号计数器当前值的基础上增加的值。考虑到它刚刚建立，并且设定为零，所以它的新值就应该为 1。假如这时执行了 MAXINST 的第二个拷贝，会有什么情况发生呢？显然，这时必须经历我们刚才介绍的那段相同的代码。这样就会建立 MAXINST 信号机，增大它的计数值，以此类推。需要再次建立信号机吗？好了，这种说法并不完全正确。CreateSemaphore () 是一个智能 API 函数，它只会在第一次运行的时候建立一个信号机。从那时以后，假如再次执行到它，同时要建立的信号机已在系统内存在了，那么它就只是简单地“打开”它。这正是我们执行 MAXINST 程序的第二个拷贝时发生的情况。信号机已经存在于系统内，同时应用程序会获得上次建立的那个资源的句柄。由于信号计数器已经设置成 1，所以它会在调用了 ReleaseSemaphore () 函数以后增至 2 第三次执行 MAXINST 则会继续增至 3。在这以后，假如试图启动 MAXINST 的第四个拷贝，如图 7-19 所示的那个消息框就会在屏幕上显示出来。

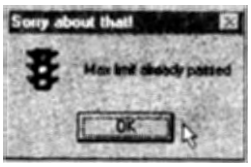


图 7-19　MAXINST 的运行拷贝不能超过三个

假如这时关闭了一个拷贝，就可以再建立一个新拷贝。运行拷贝的关闭将导致信号计数器自动减一，从而为附加的应用程序腾出空间。在 MAXINST 示范程序里，你也许会得到带

有相同标题的两个拷贝，因为标题栏内的编号是根据信号计数器得出的——范围在 1 到 3 之间。这儿真正重要的是信号计数器不允许同时启动的拷贝数目多于三个。

把拷贝限制在一个以内

现在让我们继续深入讨论信号机的用法，我们在这儿准备把运行拷贝的总数限制在一个以内。在这种情况下，我们可以建立一个名为 JUSTONE 的信号机，然后把它的当前值和最大值都设置成一。假如执行了同一程序的第二份拷贝，CreateSemaphore（）就会返回一个错误条件，这是通过 GetLastError（）与 ERROR_ALRADY_EXISTS 定义比较得出的结果：

```
...
// creating the semaphore
hsem = CreateSemaphore (NULL, 1, 1, " JUSTONE");
if (GetLastError () == ERROR_ALREADY_EXISTS)
{
    ...
    // display an error message
    ...
    // closing the semaphore
    CloseHandle (hsem);
    return FALSE;
}
...
```

假如校验得到了一个肯定的结果，就表明用户试图执行相同应用程序的第二个拷贝，这是程序不允许的。图 7-20 在屏幕上显示了一个消息框，它提醒用户这个应用程序运行的拷贝数不能多于一个。

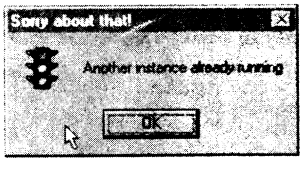


图 7-20　用一个信号机限制每个进程的拷贝数

7.12.2　建立一个简单的字处理程序

让我们继续对窗口世界的探索。接下来，我们准备在 EDIT 预定义窗口类提供的功能的基础上，建立一个简单的字处理程序。这个例子使得我们有机会对迄今为止学到的所有进程进行练习。图 7-21 显示了 WORDPRO 示范程序的显示情况。该程序与莲花（Lotus）公司开发的那个功能强大的字处理程序没有任何关系。WORDPRO 程序的源代码可在本书附带 CD 的 Listing 7.8 内找到。

EDIT 窗口覆盖了百分之百的客户区表面，并且会自动适应主窗口的尺寸变化。EDIT 窗

口是在 WM_CREATE 消息里建立的，它的初始位置与客户区的起点匹配。由于主窗口内还存在着一些非客户区组件，所以我们暂时不能决定实际的客户区域大小。因此，我们需要暂缓执行用于定义尺寸的代码段，把它放到应用程序窗口进程的另外一个部分执行。

首先把 EDIT 窗口在两个轴上的尺寸设置成零。真正改变 EDIT 窗口尺寸的操作要在 WM_SIZE 消息到达主窗口进程以后才能进行。我们需要调用 SetWindowPos () 这个 API 函数，从而实现对 EDIT 窗口尺寸的延展，使其覆盖整个客户区域：

```
...
case WM_SIZE:
{
    SetWindowPos (hwndEdit, HWND_TOP,
                  -1, -1,
                  LOWORD (lParam) + 1,
                  HIWORD (lParam) + 1,
                  0);
}
    break;
...
```

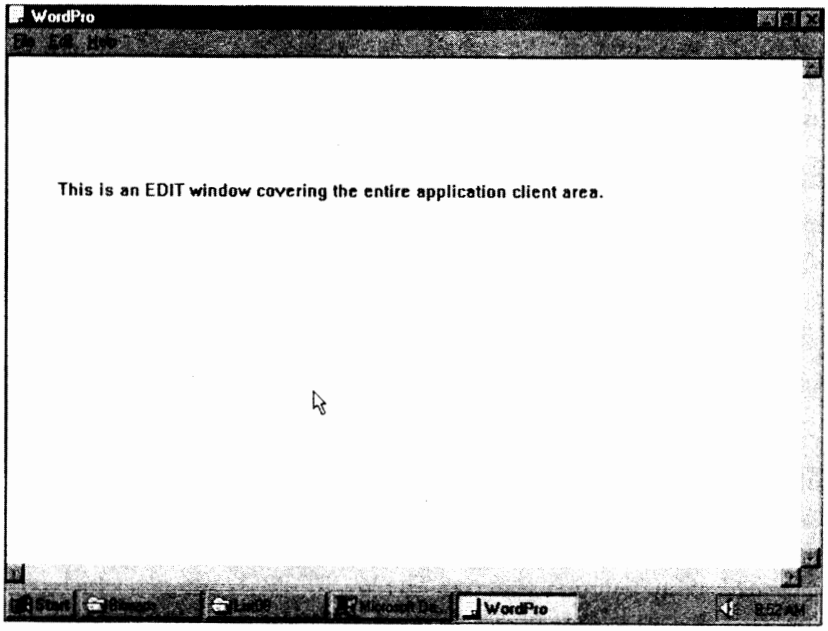


图 7-21　WORDPRO 是一个叠置式窗口，其中包含覆盖整个客户区的一个 EDIT 窗口

窗口起点的负值将由围绕于 EDIT 窗口的细线边框进行调整。程序把 EDIT 窗口四个边

框都拉长一个象素的距离，从而确保它的边框对于用户来说是可见的，这样就改进了应用程序的整体外观。

7.13 标准内存区的延展

接下来，我们准备探讨 Windows 编程一个比较容易引起混淆的领域，即我们常说的附加字节。为了真正理解附加字节，首先牢固掌握 WNDCLASSEX 的语法是很有帮助的。在它的语法里有两个值得注意的语法项：cbWndExtra 和 cbClsExtra。这两项分别是指可以增加到属于那个类的每个窗口的一些内存空间，以及增加到那个类本身的一些内存空间。在以前的例子里，我们总是把这两个项设置成零，这表示没有必要对标准的窗口内存区域进行扩展。

应用程序注册一个窗口类或者建立一个窗口的时候，它就会把一些相关信息存储到系统内存数据里，而不是存储到地址空间里。这两个区域的大小是未知的，而且它们的布局也是事先无法具体化的。

一个 Win32 窗口类建立以后，它会自动包含了一个 4 字节长的区域，该区域用于存放一些应用程序数据。我们可以在 SetWindowLong（）和 GetWindowLong（）这两个 API 函数里设置 GWL_USERDATA 定义，从而对这个区域进行访问。根据应用程序的特定需求，开发者可以在 GWL_USERDAT 内存区里存储任何种类的数值（两个短整数、一个长整数或者指向内存位置的一个指针）。为了把应用程序数据与窗口句柄联系起来，采用这种办法就是最简便的。这样一来，一旦知道了窗口句柄，我们就可以相当容易地访问对应的应用程序数据。假如标准的 4 字节内存区域不足以满足应用程序的需要，就必须增加一些附加的内存，从而对标准内存区域进行扩展。

开发者是附加内存的唯一所有者和管理者，他们能在其中存储任何有用的信息，从而简化应用程序的设计和逻辑。然而，Windows 95 在这方面设置了某些限制。任何扩展内存区域的大小都不能超过 40 字节。假如分配的值超出了 40 字节的上限，RegisterClassEx（）就会返回一条出错信息，并且中断类注册进程。为什么要设置成 40 字节，多设置一些不行吗？这个问题的答案归根溯源还要归咎于那臭名昭著的系统资源，用于容纳所有类和窗口信息的系统数据区域在空间上是有限制的。实际应用中，为附加内存分配的更为合理的尺寸在 4 到 12 字节之间——正好存放一个或者三个 4 字节长的数值。这些数值可以是整数、一个指针和四个短整数，或者其他任何形式的组合，只要不超过 12 字节的限制就行。显然，开发者要保留足够的空间，用它来存放与属于某一类的所有窗口关联起来的附加信息，或者与那一类本身关联起来的附加信息。第一种方案的应用场合是最多的。对于属于某一类的所有窗口来说，只有需要对与之相关的许多信息进行存放时，对那个类预约的内存区域进行扩展才是有用的。这在同时提供几种文档类型的一个 MDI 应用程序里是很常见的。在第六章“菜单的运用”里，我们探讨了如何根据用户的选择来显示出几个菜单模板。在 MDI 应用程序里，每个文档窗口都需要与它对应的菜单栏和相应的下拉式菜单，否则便无法实现自己支持的功能（请记住，文档窗口就是子窗口，它本身是不允许显示菜单的）。用户把视觉焦点从一个文档窗口转移到另外一个的时候，主窗口的菜单也需要进行对应的变动。为了实现这些要求，我们可以为每个文档类分配一些附加内存。在 PARTY4 示范程序（可在本书附带 CD 的 Listing 7.9 里找到）里，主窗口类有 4 个附加字节，这些附加字节的用途是存放活动子窗口的句柄，两个文档类中的每一个都有自己的 4 字节内存区域，它的作用是容纳与那种文档有关的菜单句柄。

PARTY4 看起来就像一个初级的 MDI 应用程序。启动这个程序以后，其中没有建立任何文档，同时，主窗口只提供了一个菜单，其中包含了很少的几个选项，利用这些选项可以建立属于两种支持类型之一的文档。

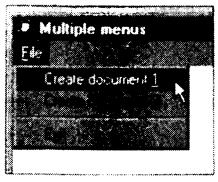


图 7-22　与 PARTY4 关联起来的初始菜单提供了两个菜单项，
用于建立相应的文档窗口

第一个文档建立起来以后，菜单栏也会发生相应的变化，从而反映出活动文档的存在。为了让大家易于理解哪个菜单是与主窗口链接起来的，我建立了第二个顶级菜单项，它唯一性地标识了那个菜单。如图 7-23 所示。

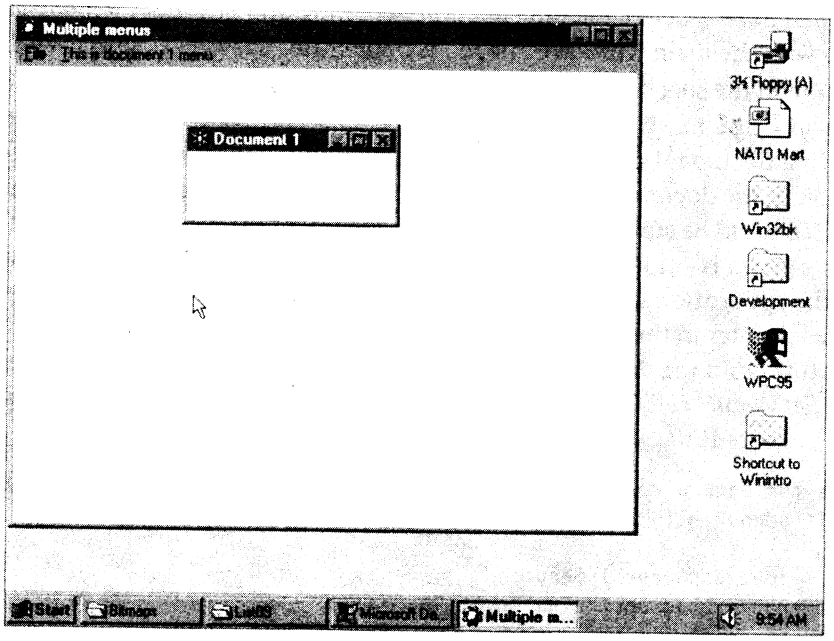


图 7-23　与主窗口关联起来的菜单将根据活动文档进行变化

每个文档窗口的句柄都适于存储到类的附加字节里。当用户通过鼠标选中了一个特定的文档以后，应用程序就会接收到类窗口进程里的 WM_MOUSEACTIVATE 消息。在这个消息块里会发生三件事情。首先，应用程序会检查选中的文档是不是当前处于活动状态的那个。这个简单的测试结果是通过当前文档句柄（hwnd）与存储于主窗口附加内存里的文档句柄比较得出的。主窗口类已经为每个窗口分配了 4 字节的内存区域（cbWndExtra 项），用它来存

储当前活动文档的句柄。通过 WM_NCACTIVATE 消息对活动属性进行动态分配和删除的时候，存储下来的这种信息就显得相当有用了。

GetActiveDoc () 实际是一种定义，而不是真正的 API 函数。在这个伪函数的背后，存在着对 GetWindowLong () 函数的一次调用。后面这种函数是由 Win32 API 提供的，它的作用是访问一个窗口预约内存区域内的特定位置（在后面讨论）。假如用户单击的是一个本已处于活动状态的文档窗口，就没有必要改变这个窗口的外观，也没有必要把相同的句柄值存放到主窗口附加字节里。另外一方面，假如用户选择的是一个不同的文档窗口，那么首先就需要把当前活动文档的“激活”标题栏属性消除，然后把这种属性分配给接收到 WM_MOUSEACTIVATE 消息的那个文档。

最后一个操作涉及到为用户选中那个文档窗口句柄分配的附加内存（这种分配在主窗口内实现的）。下面是在文档窗口进程内完成这些操作的代码段：

```
case WM_MOUSEACTIVATE:
{
    // is the selected doc already active?
    if(hwnd == GetActiveDoc((HWND)wParam))
        break;

    // remove the blue title bar from the active doc
    SendMessage(GetActiveDoc((HWND)wParam), WM_NCACTIVATE,
        (WPARAM)FALSE, 0);

    // activate the child window displaying a blue title bar
    SendMessage(hwnd, WM_NCACTIVATE, (WPARAM)TRUE, 0);
    // store the new active document
    SetActiveDoc((HWND)wParam, hwnd);
    // set the menu
    SetMenu((HWND)wParam, (HMENU)GetClassLong(hwnd, 0));
}
    break;
```

PARTY4 在什么时候载入文档菜单，并且怎样把它和主窗口关联起来呢？在这些工作是在 WM_COMMAND 消息里进行的。建立了一个单一的文档以后，就需要在 WM_COMMAND 消息里进行更进一步的处理。显示一个最新建立的文档窗口之前，应用程序会检查这是不是属于那一类的第一个窗口。为了进行这种检查，首先需要取回文档类附加内存里的信息——必须把类菜单句柄存放到类附加内存里。假如要建立的文档窗口确实属于那一类的第一个窗口，附加内存区域里就是空的，因此会返回一个 NULL 句柄。在这种情况下，对 if 语句的测试就是有效的——应用程序会从资源文件里载入类菜单，然后把它立即存放到类附加内存里。

在这个时候，文档菜单会通过一次 SetMenu () 调用取代当前的主窗口菜单。每次建立属于那一类的新文档窗口时，这一步骤都会重复执行。

```
// is the menu already stored?
if (! (hmenu2 = (HMENU) GetClassLong (hwndChild, 0)))
{
    // load document 1 menu
    hmenu2 = LoadMenu (hInstance, " mainmenu2");
    // store it
    SetClassLong (hwndChild, 0, (long) hmenu2);
}
// set the menu
SetMenu (hwnd, hmenu2);
```

所有活动的文档都已被选择性地删除以后，就会恢复最初的那个菜单。

访问预约内存区

早些时候，我们曾经提到过：在 Win32 里，与类或者窗口有关的信息是存储在一个内存区域里的——不管是哪一个进程注册了类或者建立了窗口。这种说法是正确的。另外我还要增加一点，这两种操作都会生成一个内存块，其中包含了未被文档化的信息。这种说法也是正确的。然而，我们访问窗口或者一个类预约区域的时候，也需要小心谨慎。

正如我们在 PARTY4 这个示范程序里证明的那样，Win32 提供了四个函数对类和窗口预约内存区域里的特定位置进行读写，这四个函数分别是：GetWindowWordk ()，SetWindowWord ()，GetWindowLong () 以及 SetWindowLong ()。如图 7-21 所示，对标准内存进行扩展的时候，附加内存的分配最终是从正偏移位置处起始的。四个 API 函数都需要用到一个有效的窗口句柄，把它当作自己的第一个参数使用；并且需要用到一个定义，用它指定预约内存区域里哪个位置是需要指向的。

下面让我们研究一下 API 函数 GetWindowLong () 的语法结构：

```
#include <winuser.h>
LONG GetWindowLong (HWND hwnd, int nIndex);
```

参数	说明
HWND hwnd	窗口句柄
int nIndex	一个 GWL_定义
返回值	在正文里讨论
LONG	返回存储在某个内存位置的信息，这个内存位置是由 nIndex 参数指定的

这个 API 函数的关键元素就是 nIndex 参数，它可以在表 7-11 内列出的所有定义中选择一种。通过对预约内存区域进行查询，应用程序就可以获得一些相当有用的信息，比如窗口

风格的集合（包括扩展风格）、父窗口的句柄以及应用程序实例句柄等等。

表 7-11　指向窗口预约内存区域内 4 字节位置的各种定义

索　引	值	说　明
GWL_WNDPROC	-4	与准备检查的窗口关联在一起的窗口进程地址
GWL_HINSTANCE	-6	应用程序实例句柄
GWL_HWNDPARENT	-8	父窗口句柄
GWL_STYLE	-16	窗口风格集（WS_）
GWL_EXSTYLE	-20	扩展的窗口风格集（WS_EX_）
GWL_USERDATA	-21	适用于任何一种应用程序需求的 4 字节区域
GWL_ID	-12	用于子窗口的窗口 ID 或者菜单句柄（弹出式和叠置式）

Win32 能自动预约一个 4 字节的区域，用于存放用户数据。应用程序在这儿可以存放任何信息——从指针到整数均可。这个区域肯定是存在的，无论开发者是否准备通过附加内存分配附加的空间。

由于存在这种能自由使用的数据暂存区域，所以附加字节的使用就显得没有那么重要了。这是由于 4 个字节足以存放指向某个内存位置的指针，或者存放一个表格的索引。这个区域让我们有机会对信息进行简便的封装，然后把它与窗口句柄结合在一起——这是 Win32 编程的一项重要目标。

除此以外，在 Win32 里，预约内存区域内包含了与窗口有关的窗口进程的地址。这就解释了为什么 DispatchMessage () 函数需要发送一条消息的时候，它可以自动跳转至正确的窗口进程。大家现在可以想象一下微软的工程师们是如何编写 DispatchMessage () 函数的：

```
LRESULT WINAPI DispatchMessage(LPMSG lpmsg)
{
    WNDPROC pfn;

    // get the window procedure address from the window reserved memory area
    pfn = (WNDPROC)GetWindowLong(lpmsg -> hwnd, GWL_WNDPROC);
    // call the window procedure
    return (*pfn)(lpmsg -> hwnd, lpmsg ->message, lpmsg ->wParam,
        lpmsg ->lParam);
}
```

对 SendMessage()也感兴趣吗？下面是它的样子：

```
LRESULT WINAPI SendMessage(HWND hwnd, UINT msg, WPARAM
    wParam,LPARAM lParam)
{
    WNDPROC pfn;
```

```
    // get the window procedure address from the window reserved memory area
    pfn = (WNDPROC)GetWindowLong(hwnd, GWL_WNDPROC);
    // call the window procedure
    return (*pfn)(hwnd, msg, wParam, lParam);
}
```

这两个函数看起来有些相似，是吗？假如在预约内存区域里存放了窗口进程的地址，就会产生一些额外的影响，关于这方面的问题，我们将在第15章“Windows高级技术”里进行详细的介绍。

如果想把信息存放到预约内存区域的任何位置，我们可以调用SetWindowLong()函数，它的语法结构与GetWindowLong()几乎是完全一致的：

```
#include <winuser.h>
LONG SetWindowLong (HWND hwnd, int nIndex, LONG lNewLong);
```

参数	说明
HWND hwnd	窗口句柄
int nIndex	一个GWL_定义
LONG lNewLong	准备存放到窗口内存区域内的值
返回值	在正文里讨论
LONG	返回存储在某个内存位置的信息，这个内存位置是由nIndex参数指定的

现在假设我们准备把p指针存放到GWL_USERDATA内。下面这个代码段解释了该如何实现这一目标：

```
...
char *p;
...
SetWindowLong (hwnd, GWL_USERDATA, (long) p);
...
```

假如准备存储的信息不属于或者不适合标准的预约内存位置，就必须使用附加内存。从程序开发的角度来看，这时并没有发生什么变动，只是采用一个正偏移：

```
...
SetWindowLong (hwnd, 0, (long) p);
...
```

这就是PARTY4程序把活动文档的句柄存放到主窗口附加字节内产生的结果（为了节省系统资源，我们也可以使用GWL_USERDATA定义）。GetWindowWord()和SetWindowWord()这两个API函数在Win32里几乎已经废弃不用了。

为了访问由一个类预约的内存位置，Win32 API提供了几乎完全一致的函数集，它们包括：GetClassLong()，SetClassLong()，GetClassWord()以及SetClassWord()。然而，

在使用这些 API 之前，我们有必要按照下述的语法为每一个类建立一个窗口：

```
#include〈winuser.h〉
DWORD SetClassLong (HWND hwnd, int nIndex, LONG lNewVal);
```

表 7-12 内列出了这一系列函数可以选用的所有索引值。系统注册一个类的时候，就会把该类的名字自动转换成一个“原子”——一个短整数，它在整个系统的范围以内唯一性地标识了这个类。

表 7-12　与类预约内存区域使用有关的函数索引

索　引	值	说　明
GCL_MENUNAME	(-8)	与类关联在一起的菜单资源的标签
GCL_HBRBACKGROUND	(-10)	用于重画窗口背景的刷子句柄
GCL_HCURSOR	(-12)	光标句柄
GCL_HICON	(-14)	图标句柄
GCL_HMODULE	(-16)	模块句柄，与应用程序实例句柄完全一致
GCL_CBWNDEXTRA	(-18)	窗口附加字节尺寸
GCL_CBCLSEXTRA	(-20)	类附加字节尺寸
GCL_WNDPROC	(-24)	类窗口进程的地址
GCL_STYLE	(-26)	类风格
GCL_ATOM	(-32)	类原子。这个数字在整个系统范围内唯一性标识了类
GCL_HICONSM	(-34)	类小图标的句柄

第 8 章　Win32 的对话框管理

在一个标准的 Win32 应用程序里，我们不可能计算出准确的窗口数目。我们知道，对于普通的 Win32 程序来说，它一般都要提供某些特定的对话框，这些对话框是对基本窗口概念的一种延伸。我们甚至可以这样说，外壳内所有属性表的本质也都是对话框（实际是几个对话框的组合）。但是对话框的准确定义到底是什么呢？

简而言之，对话框是一系列窗口的集合，其中包括一个主窗口以及当作控件使用的几个子窗口。对话框在 Win32 应用程序里扮演了一种特殊的角色。为了彻底理解为微软公司的工程师们为什么决定把对话框与标准的窗口区分开来，我们可以先来看看早期 Windows 的情况。在 80 年代初期，微软开发了一些非常初级的工具，用它在屏幕上设计和描绘窗口，同时把所有信息都存储到一个 ASCII 文件里，这个文件就是众所周知的“模板”。假如需要同时画出几个窗口，这些工具就显得相当有用了，对话框的显示特别需要用到它们。从那以后，对话框一直都是通过特定的工具在屏幕上画出的，然后存储到一个对话框模板里。这个模板是许多 ASCII 信息的集合，并且需要把它存储于资源文件里。

应用程序需要在屏幕上显示一个模板的时候，必须先从资源文件里读取对话框模板，然后再把模板显示出来。最后那个步骤完全等效于建立一个单独的窗口，只是这个窗口是由对话框组成的。利用资源文件里的模板建立这样的一个窗口——这是只有对话框才具备的一种特征。除此以外，对话框肯定也是弹出式窗口。也就是说，它们提供了 WS_POPUP 消息。图 8-1 向大家展示了从 Microsoft Word 7.0 for Windows 95 里截取下来的一个对话框。

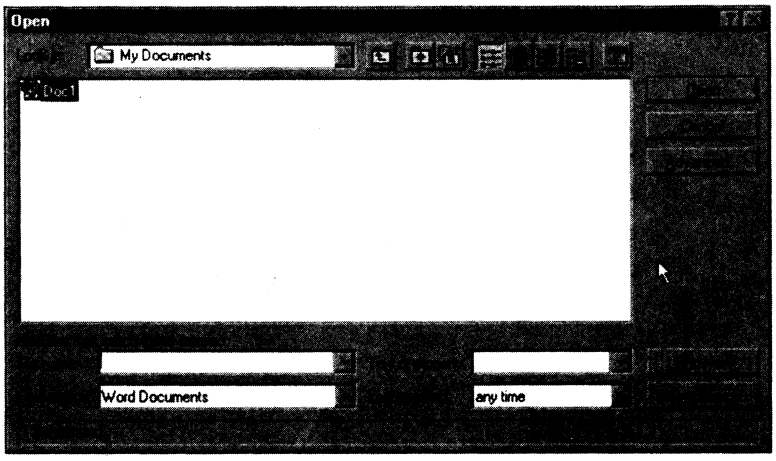


图 8-1　选择 Open 菜单项以后，Microsoft Word 就会显示出这个用于打开文档的对话框

从外观上看，这个窗口是非常特殊的。它没有提供任何一个菜单栏，同时客户区是由几个子窗口占据的。这个 Open 对话框内包含了一些按钮，底部有几个组合框，中央占据主要部

分的则是一个大型列表框——只用到了几个控件。这些子窗口通常属于预定义窗口类对象，或者属于通用控件对象。图 8-2 向大家展示了从 FILEOPEN.DLG 文件里提取出来的一个对话框模板，这个文件可在开发包（SDK）里找到。

图 8-2 中程序如下：

```
 FILEOPENORD DIALOG LOADONCALL MOVEABLE DISCARDABLE
#ifdef _WIN95
36, 24,  268, 134
CAPTION "Open"
STYLE WS_CAPTION | WS_SYSMENU | WS_POPUP | DS_MODALFRAME | DS_3DLOOK |
DS_CONTEXTHELP
#else // _WINNT
36, 24,  264, 134
CAPTION "Open"
STYLE WS_CAPTION | WS_SYSMENU | WS_POPUP | DS_MODALFRAME | DS_3DLOOK |
DS_CONTEXTHELP
#endif
FONT 8, "Helv"
BEGIN
```

```
    LTEXT "File &name:", stc3,   6,   6,  76,  9
    CONTROL "", edt1, "edit", ES_LEFT | ES_AUTOHSCROLL | WS_BORDER |
            WS_TABSTOP | WS_CHILD | ES_OEMCONVERT,
            6,    16,  90,  12
    CONTROL "", lst1, "listbox",
            LBS_SORT | LBS_HASSTRINGS | LBS_NOTIFY | LBS_DISABLENOSCROLL
            | WS_VSCROLL | WS_CHILD | WS_BORDER | WS_TABSTOP
            | LBS_OWNERDRAWFIXED,
            6,    32,  90,  68

    LTEXT "&Folders:", -1, 110,   6,  96,   9
    LTEXT "",             stc1, 110,  18,  96,   9, SS_NOPREFIX
    CONTROL "", lst2, "listbox",
            LBS_SORT | LBS_HASSTRINGS | LBS_NOTIFY | LBS_DISABLENOSCROLL
            | WS_VSCROLL | WS_CHILD | WS_BORDER | WS_TABSTOP
            | LBS_OWNERDRAWFIXED,
#ifdef _WIN95
            110,  32,  96,  68
#else
            110,  32,  92,  68
#endif

    LTEXT "List files of &type:", stc2, 6,  104,  90,   9
    CONTROL "", cmb1, "combobox", CBS_DROPDOWNLIST | CBS_AUTOHSCROLL |
            WS_BORDER | WS_VSCROLL | WS_TABSTOP | WS_CHILD,
#ifdef _WIN95
            6,  114,  90,  96
#else
            6,  114,  90,  36
#endif
```

```
#ifdef _WIN95
    LTEXT "Dri&ves:", stc4,      110, 104,  96,  9
#else
    LTEXT "Dri&ves:", stc4,      110, 104,  92,  9
#endif
    CONTROL "", cmb2, "combobox",
            CBS_SORT | CBS_HASSTRINGS | CBS_OWNERDRAWFIXED | CBS_DROPDOWNLIST
            | WS_CHILD | CBS_AUTOHSCROLL | WS_BORDER | WS_VSCROLL
            | WS_TABSTOP,
#ifdef _WIN95
            110, 114,  96, 68
#else
            110, 114,  92, 68
#endif

#ifdef _WIN95
```

```
    DEFPUSHBUTTON "OK", IDOK,       212,  6, 50, 14, WS_GROUP
    PUSHBUTTON "Cancel", IDCANCEL,  212, 24, 50, 14, WS_GROUP

    PUSHBUTTON "&Help", pshHelp,    212, 46, 50, 14, WS_GROUP
    CHECKBOX "&Read only", chx1,    212, 68, 50, 12,
                        BS_AUTOCHECKBOX | WS_TABSTOP | WS_GROUP
#else
    DEFPUSHBUTTON "OK", IDOK,       208,  6, 50, 14, WS_GROUP
    PUSHBUTTON "Cancel", IDCANCEL,  208, 24, 50, 14, WS_GROUP

    PUSHBUTTON "&Help", pshHelp,    208, 46, 50, 14, WS_GROUP
    CHECKBOX "&Read only", chx1,    208, 68, 50, 12,
                        WS_TABSTOP | WS_GROUP

    PUSHBUTTON "Net&work...", psh14,   208,  114, 50,  14, WS_GROUP

#endif

END
```

图 8-2　推出 Windows 95 之前，一个标准 Windows Open 对话框的模板

对话框窗口是用对话框模板内的 DIALOG 命令描述的。然而，每个控件最终都需要用一条单独的语句来引入。当应用程序通过 DialogBoxParam（）或者其他 Win32API 函数调用对话框模板的时候，就需要用到那条语句内包含的所有信息，这些信息对于特定控件的生成是必须的。

总而言之，对话框是一种弹出式窗口，它的主要功用是在自己的客户区内容纳几个控件，每个控件都具有 WS_CHILD（子窗口）风格。

8.1　模态和非模态对话框

到这时为止，我们已经知道了对话框很容易建立并且容易在屏幕上显示出来，这是由于系统把每个组件窗口都当作一种单一对象进行处理。这是用对话框取代标准窗口的原因之一。至于另外一个原因，我们会在这一小节内进行详细的解释。

Windows 支持两种特色的对话框：模态和非模态对话框。模态对话框是各种应用程序里

都是最常用的。从另外一方面来看，自从微软公司推出了 Windows 2.x 以后，由于子窗口的大规模应用，非模态对话框就逐渐丧失了它的主导地位。然而，假如读者仍然对非模态对话框有兴趣，也可以参考我们在本章后面提供的一些信息。

非模态对话框非常类似于一个标准窗口，只是它们建立的途径不一样。应用程序是从 CreateDialogParam () 函数里获得非模态对话框句柄的。它同时要对定址于那个窗口句柄的消息进行收发，从而与非模态对话框进行交互作用。

模态对话框不会返回对应的窗口句柄，并且更重要的一点在于，它能吸引应用程序的注意——只要该对话框在屏幕上保持显示，应用程序的主窗口就会一直处于屏蔽状态。只有当模态对话框消失以后，应用程序主窗口才会重新得以恢复。为了证明这一事实，我们可以在 Microsoft Word 7.0 或者其他 Win32 应用程序里选择 Open 菜单项，然后看看会发生什么情况。打开了 Open 对话框以后，除非按下 Open 或者 Cancel 按钮，否则就没有办法对应用程序的其他任何一种组件进行操作。这并非一种限制——更准确地说，它与这一类对话框的行为需求是对应的。假如大家对第 6 章“菜单的运用”还有印象，那么应该记得起我们曾讲述过一种扩展命令菜单项。这种菜单项的特征是在其末尾有一个省略号，假如选中该菜单项，就会显示出一个对话框。在 99%的情况下，这时显示出来的都应该是一个模态对话框。

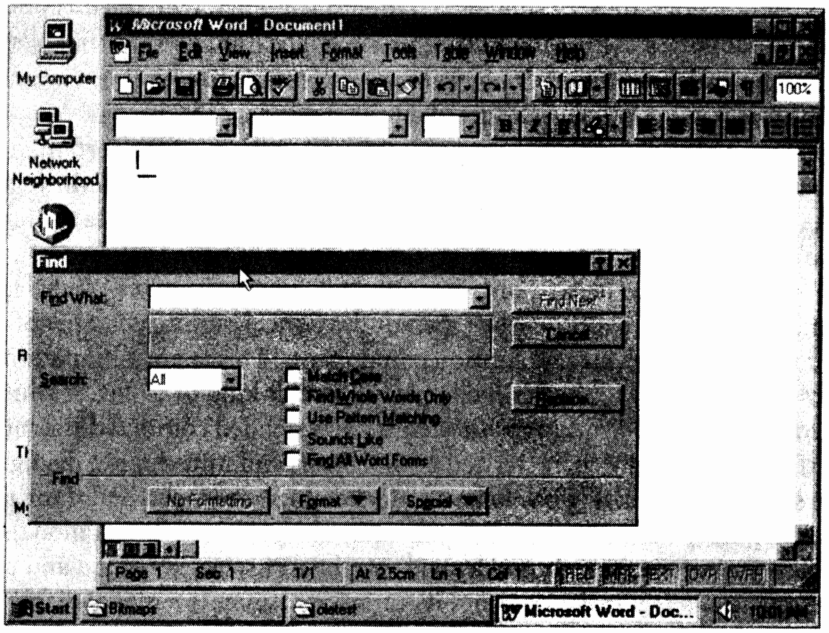


图 8-3　非模态对话框的显示象素肯定是从桌面取得的

对话框模板（稍后我们再详细讲述）既可以生成一个模态对话框，也可以生成一个非模态对话框。具体生成哪一个取决于开发者选用什么 API 来装载和运用模板。DialogBoxParam () 和它的一些变种函数可以生成模态对话框，而 CreateDialogParam () 则能生成平时很少用到的非模态对话框。

非模态对话框与应用程序是共存的，这种行为与子窗口是完全一致的。和子窗口比较起来，非模态对话框的唯一优点在于它仅需通过一个对话框模板就能建立一系列窗口，不像子

窗口那样非要多次调用 CreateWindowEx () 才行。非模态对话框的应用场合现在已经变得越来越少，仅局限于一些特定的场合才需要用到它。举个例子来说，在 MS Word 里，假如在文档里搜索一个正文串，那时显示出来的就是一个非模态对话框。用户可以在对话框内插入正文，同时与文档进行交互作用。

为了找出 Word 7.0 里的另外一个非模态对话框，我们可以选择 Edit 顶级菜单内的 Find 菜单项。随后显示出来的就是一个非模态对话框，尽管它处于非活动状态时仍然能够在屏幕上看到它。事实上，对话框在屏幕上显示的时候，光标仍然能在正文上闪烁。

非模态对话框与子窗口的另外一处不同在于它们与父窗口的关系。非模态对话框所用的显示象素肯定是从桌面获得的，而子窗口则只能从父窗口的客户区域内取得象素。在图 8-3 里，Find 对话框（另外一个非模态对话框）已经部分超出了 Word 7.0 的主窗口，这就证明它的显示象素是从桌面获得的。

接下来，我们准备就对话框窗口建立过程中涉及到的所有步骤进行详尽的分析。

8.2 对话框的建立

对话框的本质就是窗口。它属于未文档化的 #32770 类，这一点通过运行 Spy++ 程序可以看出。建立对话框之前，我们没有必要先去注册一个窗口类。因此，大家也许会顺理成章地认为它也不需要窗口进程。但这种推测并不是完全正确的。

正如我们以前提到的那样，对话框的主要用途是在其中包含几个子窗口，这些窗口归属于预定义窗口类或者新的 Win95 通用控件。对话框实际上是一种对象，它是由一系列窗口组成的，这些窗口中没有任何一个属于应用程序内注册的窗口类。这意味着什么呢？它意味着应用程序无法知道用户选择了什么，或者在一个对话框组件里插入了什么？无论用户进行了什么操作，这种操作都会传递给对话框本身或者一个控件——它们的窗口进程就驻留于 Windows 95 的内部。这样一来，应用程序怎样理解用户的操作呢？举个例子来说，它怎样才能知道用户在列表框里选择了第二个选项，从而需要打开某个文件呢？假如它判断错误，或者根本不知道怎样判断，就会对用户的操作产生错误的理解。为了解决这个问题，我们需要自己动手编写对话框进程，这也是一种窗口进程。

8.2.1 对话进程

无论用户在一个对话框里进行何种操作，由此产生的消息流将完全与应用程序断绝联系。对话框里发生的这些操作是相当固定的（对话框不过是一个简单的容器，各种控件需要的象素都是从中取得的）。对每一个控件的操作在窗口进程里进行的时候，它都会通知自己的父/物主窗口。因此，属于 #32770 类的窗口进程就会接收到来自对话框内不同控件的通知消息，从而知道用户当前采取了什么操作。

不幸的是，对话框窗口归属于一种未文档化的预注册类，并不属于应用程序代码。#32770 窗口进程会把从控件处接收到的所有数据都传递给自己的物主窗口——也就是说，传递给我们在应用程序代码内部编写的对话框进程。这样一来，应用程序就能理解对话框窗口内部发生的事件，这涉及到不在它控制范围以内的一小部分代码。接下来讲述的是它的详细工作原理。

在一个对话框控件上方单击鼠标左键的时候，该控件就会生成一条 WM_LBUTTONDOWN 消息，这条消息定址于对应的控件。控件随后会利用由 WM_COMMAND

发送的一条通知消息，从而通知它的父窗口（即对话框）当前发生了什么事件。所有这些操作都是在应用程序的控制范围以外发生的，因为这些操作只涉及到预定义窗口类。对话框类#32770会把它接收到的每一条消息都发送给应用程序内注册的对话框进程，由这个进程最终决定用户选择了什么，或者在对话框内做了什么事情。一个控件窗口会把一条消息传送给它的父/物主窗口（对话框），再由对话框把消息依次转交给自己的物主。对话框窗口（主窗口）是在应用程序里建立的。

在应用程序里增添一个对话框需要涉及到对话框进程的编写。某个现成的控件发出了自己的消息以后，对话框就会自动调用对话框进程。然而，理解下面这一点对于我们来说是至关重要的：开发者在自己的代码内编写的对话框进程并非对话框的窗口进程！

#32770类的窗口进程驻留于Windows内部，它是一种稳定和不可更改的代码块。我们在程序源代码内编写的对话框进程在结构上与类窗口进程是完全一致的，尽管前者抄了消息流的近道。下面是具体的解释。消息首先经过#32770窗口进程传递，然后暂时性跳转至我们编写的对话框进程，最终返回它的初始位置。经过#32770类窗口进程的每条消息都会到达对话框里程。但是，正如大家看到的那样，我们只对这些消息中的少数几条有兴趣。

8.2.2 从资源文件装载模板

除了对话框进程以外，我们在C源代码和资源文件里建立一个对话框的时候，还需要涉及到另外几个步骤。首先，我们必须从资源文件里载入模板，然后把它转换成真正的对话框。这是生成对话框最常用的一种方式，然而并非唯一途径。对话框模板也可以在内存里直接装配，从而实现对话框在程序运行期间的动态建立。这是一种相当简单的技术，只是很少使用。

正如我们以前提到的那样，DialogBoxParam（）是我们建立对话框的最佳API函数：

```
#include〈winusr.h〉
int DialogBoxParam（HINSTANCE hInstance，
                    LPCTSTR lpTemplateName，
                    HWND hwndOwner，
                    DLGPROC lpDialogFunc，
                    LPARAM dwInitParam）；
```

参数	说明
HINSTANCE hInstance	应用程序实例句柄
LPCTSTR lpTemplateName	在资源文件里的对话框模板名称
HWND hwndOwner	对话框物主窗口的句柄
DLGPROC lpDialogFunc	对话框进程函数的名称
LPARAM dwInitParam	由应用程序定义的值
返回值	在正文里讨论
int	假如函数调用失败，就为-1；或者转向API函数EndDialog（）

在应用程序实例句柄以后，我们必须提供存放于资源文件里的对话框模板的名称，这是利用MAKEINTRESOURCE宏转换后生成的一种正文串或者数字。第三个参数是一个句柄，

它指定了对话框的物主窗口。根据这个参数的正式文档资料，它应该指定对话框的父窗口。

考虑到对话框其实是支持 WS_POPUP 风格的窗口，所以它们的显示象素是从桌面取得的。作为第三个参数传递的窗口句柄指定了这样一个特殊的窗口：当模态对话框在屏幕上显示出来后，该窗口就会暂时转入非活动（屏蔽）状态，从而防止了用户对这个窗口进行交互操作。DialogBoxParam（）函数的语法结构还要求使用对话框进程的名称，每次需要对一条消息进行处理的时候，#32770类就会调用这个进程。

最后那个参数是一个普通的 LPARAM 数据，这是一个完全开放的数据区域，任何应用程序均可在此存放数据。其中可以包含一个简单的整数值，或者指向某个内存位置的指针。很常见的一种情况是，开发者可以把应用程序实例句柄传送到这个区域，使其在对话框建立以后马上就可以由应用程序利用。从本质上看，最后这个参数等价于 CreateWindowEx（）的最后一个参数。请大家参考第七章“建立窗口的技术”。

DialogBoxParam（）函数的语法要求用到对话框进程的名称，用它来接收与那个特定对话框相关的所有消息。正如我们以前提到的那样，对话框进程是一种特殊的函数。无论从它的参数来看，还是从调用规范来看，对话框进程都完全等价于一个普通的窗口进程。唯一的区别在于它们的返回值不同：窗口进程返回的是 LRESULT，而对话框进程返回的却是 BOOL。在许多情况下，返回值通常都只有四个字节的长度。

下面这个代码段向大家展示了典型对话框进程的结构：

```
BOOL CALLBACK AboutDlgProc (HWND hwnd,
                            UINT msg,
                            WPARAM wParam,
                            LPARAM lParam)
{
    switch (msg)
    {
        case WM_INITDIALOG:
        {

        }
            break;

        case WM_COMMAND:
        {

        }
            break;
    }
    return FALSE;
}
```

正如我们以前提到的那样，在一个典型的对话框进程里，我们只能对 WM_COMMAND 和 WM_INITDIALOG 消息进行拦截处理。对于前一条消息来说，我们可以把它想象成一辆不知疲倦的卡车，它能把所有通知代码从控件运载到父/物主窗口。关于这方面的问题，我们会在第九章“预定义的窗口类”里进行详细的解释。

WM_INITDIALOG 消息在对话框的建立过程中扮演了一个非常有趣的角色。为了理解它的功用，我们需要注意到这样一个事实：对话框本身以及它的控件都归属于几种预定义窗口类，它们中的任何一个都无法经由应用程序进行管理。这种情况就使得开发者编写的代码不能接收到所有那些有用的 WM_CREATE 消息，因为这些消息都定址于 Windows 内部的某些窗口进程。就 WM_INITDIALOG 的本质来看，它其实就是 WM_CREATE 消息的一个变种。对话框在屏幕上显示出来之前，这条消息对它的初始化和定制都是很有用的。一个对话框进程接收到 WM_INITDIALOG 消息以后，就意味着无论对话框还是其中的全部控件都已经成功地建立起来了，尽管这些东西在屏幕上都还不是可见的。因此，对于那些用不同信息初始化以及填写各个控件的所有指令来说，最好把它们都放置于 WM_INITDIALOG 代码块（WM_INITDIALOG case 代码块）内部。就 WM_INITDIALOG 消息来说，它本身并未提供任何值得注意的信息，如下所示：

WM_INITDIALOG	0x0110
wParam	用于接收输出焦点的控件的句柄
lParam	DialogBoxParam（）或者其他类似函数最后一个参数的值

我强烈建议大家利用 DialogBoxParam（）函数去调用一个对话框模板，因为这样一来，我们就可以在以后拦截由 lParam 指定的值（在 WM_INITDIALOG 里指定的），从而把一些附加的信息传递给对话框进程。

假如对话框进程已经处理完了一条消息，它应该返回一个非零值。对 WM_INITDIALOG 进行处理的时候，返回值就会发生一些变化。一个非零值可以把焦点设置在由 wParam 标识的控件上面；假如已经通过调用 SetFocus（）把焦点设置到了一个控件上面，那么对话框就应该返回一个零值。就作者本人来说，对话框进程里捕获到的任何消息通常都让它返回一个 FALSE 值，这样程序能够运行得很好。

除非用户按下了 OK 或者 Cancel 按钮，否则模态对话框就会移交自己手中的控制权；有些时候，其他按钮也会具有类似的效果，比如 Open 或者 Save 按钮。这就意味着一个模态对话框必须事先编写好退出代码，因为没有任何一种办法可以从它的代码外部令其消失。针对这一条规则，处理与一条中断按钮有关的 WM_COMMAND 消息时，我们需要在对话框进程里调用 EndDialog（）函数。该函数的语法结构如下所示：

```
#include 〈winuser.h〉
BOOL EndDialog (HWND hDlg, int nResult);
```

参数	**说明**
HWND hDlg	对话框句柄
int nResult	准备向建立对话框的那个应用程序返回的值

返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

DialogBoxParam（）的返回值是 EndDialog（）的第二个参数，假如用户按下的是 OK 按钮，那么通常把它设置成 TRUE；否则设置成 FALSE。

总而言之，为了建立一个对话框，我们需要采取下面这些步骤（并非一定要按顺序进行）：

▶生成一个对话框模板。

▶在源文件里引用适当的头文件，这个头文件里包含了对话框控件使用的 ID。

▶编写一个对话框进程。

▶调用对话框模板，建立和显示对话框。

这些简单的步骤需要开发者与资源编辑器以及自己的几个项目文件进行交互作用。

8.3 窗口还是对话框

在本书附带 CD 的 Listing 8.1内，大家可以找到 WINDLG 示范程序的源代码。这个例子的宗旨是向大家证明对话框与传统的窗口并无多少分别。唯一的区别在于这些窗口在某个 Win32项目内部的建立和管理方式不同。

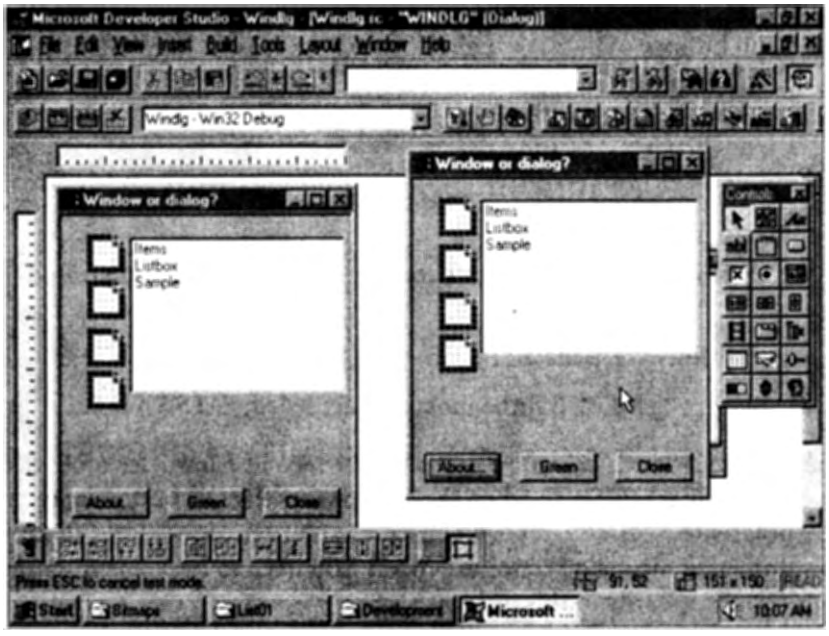


图 8-4　对话框的原型位于屏幕左侧，右侧是它真正运行时的状态

现在让我们利用 Visual C＋＋提供的资源功能描绘一个新的对话框。图8-4显示了由 WINDLG 例子生成的对话框。这是一个相当简单的对话框，在靠近底部边框的地方摆放了三

个按钮，左侧放置了一个列表框，四个图标则位于左侧。Visual C++会把这一系列窗口转换成如图8-5所示的对话框模板。

```
WINDLG DIALOGEX DISCARDABLE  91, 52, 151, 150
STYLE WS_MINIMIZEBOX | WS_MAXIMIZEBOX | WS_VISIBLE | WS_CLIPCHILDREN |
    WS_CAPTION | WS_SYSMENU | WS_THICKFRAME
EXSTYLE WS_EX_WINDOWEDGE | WS_EX_CLIENTEDGE
CAPTION "Window or dialog?"
CLASS "WINDLG"
FONT 8, "Helv"
BEGIN
    PUSHBUTTON      "About...",101,6,131,40,14
    PUSHBUTTON      "Green",102,56,131,40,14
    PUSHBUTTON      "Close",103,106,131,40,14
    LISTBOX         106,35,10,114,87,LBS_SORT | WS_VSCROLL | WS_TABSTOP
    ICON            "flake",107,13,8,18,20
    ICON            "sun",108,13,30,18,20
    ICON            "rain",110,13,53,18,20
    ICON            "clouds",111,13,73,18,20
END
```

图 8-5　与图8-4那个对话框对应的对话框模板

图中程序：

```
WINDLG DIALOGEX DISCARDABLE 91, 52, 151 150
STYLE WS_MINIMIZEBOX | WS_MAXIMIZEBOX | WS_VISIBLE |
    WS_CLIPCHILDREN |WS_CAPTION | WS_SYSMENU |
    WS_THICKFRAME
EXSTYLE WS_EX_WINDOWEDGE | WS_EX_CLIENTEDGE
CAPTION " Window or dialog?"
CLASS " WINDLG"
FONT 8, " Helv"
BEGIN
    PUSHBUTTON    " About…", 101, 6, 131, 40, 14
    PUSHBUTTON    " Green", 102, 56, 131, 40, 14
    PUSHBUTTON    " Close", 103, 106, 131, 40, 14
    LISTBOX       106, 35, 10, 114, 87, LBS_SORT | WS_VSCROLL |
                  WS_TABSTOP
    ICON          " flake", 107, 13, 8, 18, 20
    ICON          " sun", 108, 13, 30, 18, 20
    ICON          " rain", 110, 13, 53, 18, 20
    ICON          " clouds", 111, 13, 73, 18, 20
END
```

对于这个模板，我们以后会再做详细的分析。现在请大家把自己的注意力放在两个问题在上面。首先，模板被分割成两个基本的部分：一个标题和一个主体。前面六条语句指定了对话框窗口，而主体部分则对每个控件分别进行了控制（读者一眼就可以看出其中有8个控件：四个图标定义、一个列表框以及三个按钮，从模板的底部向上观看即可）。

随时要注意这样一个问题：对话框是属于＃32770类的一个窗口，每个控件都是属于预定义窗口类，或者属于通用控件的一个窗口。在对话框模板里，大家应该能够找出传递给CreateWindowEx（）的典型值，这些值都是针对不同的组件生成的（主窗口和它的控件）。

现在让我们逐个参数讨论一下CreateWindowEx（）的语法。第一个参数是一系列扩展风格的集合。WINDLG对话框模板明显提供了传统的边框扩展风格：

```
EXSTYLE WS_EX_WINDOWEDGE | WS_EX_CLIENTEDGE
```

CreateWindowEx（）的第二个参数是类名称；CLASS命令就是针对这一用途设置的：

```
CLASS " WINDLG"
```

接在这个类后面的是窗口标题：

```
CAPTION " Window or dialog?"
```

随后，我们通常还需要设置一些窗口风格，这些传统的WS_风格太长了，以致于我们一行都放不下：

```
STYLE WS_MINIMIZEBOX | WS_MAXIMIZEBOX | WS_VISIBLE
| WS_CLIPCHILDREN | WS_CAPTION | WS_SYSMENU
| WS_THICKFRAME
```

窗口位置和它的大小通常是由四个连续的参数定义的：

```
WINDLG DIALOGEX DISCARDABLE 91, 52, 151 150
```

为了最终完成CreateWindowEx（）函数的语法，我们还需要使用父窗口的句柄，但这对于对话框来说却是没有必要的，因为对话框的本质就是WS_POPUP（弹出式窗口）。用于窗口菜单的句柄与CreateWindowEx（）是一样的。对话框可以支持菜单栏的显示（作为弹出式窗口，它应该可以支持菜单的使用），然而这并不是一种很好的设计风格，所以我们要避免在对话框这种特殊的窗口里使用菜单。我们还需要应用程序实例句柄，以及与应用程序有关的特定数据。实例句柄在对话框的设计过程中没有什么用处，因为这种信息只有在执行了一个进程以后才会有用。显然，设置对话框模板的时候，我们不可能提供与应用程序有关的任何特定数据。请大家回忆一下我们以前讲述该如何处理资源文件的风格信息。

用于DIALOGEX的模板利用了FONT命令为对话框和它的控件自动分配一种字体。命令如下：

```
FONT 8, " Helv"
```

这样一来，我们就可以强制对话框用8点的Helvetica字体显示出所有正文。这种方法很简单，也很容易实现。

类似的规范亦可应用于BEGIN-END代码块内的每条语句。每条语句都标识了一个控件，并能在模板执行的时候传递用于生成该控件的所有信息。请注意紧跟在PUSHBUTTON和ICON命令串后面的数字，这些数字是对窗口ID对应的。这些数字唯一性地标识了一个子窗口。Visual C++把分配给控件的所有ID都存放在名为RESOURCH.H的一个文件里，这个文件最终会由我们包含到应用程序源代码里。

接下来，我们应该准备编写对话框进程了。在这儿，我们把 WINDLG 模板当作应用程序的主窗口来使用，而不是当作对话框主窗口使用。就这种选择来说，它具体反映在对话框模板内存储的 CLASS 命令，这种命令在标准对话框里一般都是存在的。在 WINDLG 示范程序里，我们从资源文件里装载了一个模板，强迫主窗口利用以前在应用程序源代码内编写的一个真正的窗口进程。所以不要试图在对话框进程里找出这样的一个模板，因为它根本就不在那里！WINDLG 启动以后，它就会注册 WINDLG 类。这个操作需要用到 DLGWINDOWEXTRA 定义，从而针对属于这一类的所有窗口对窗口内存区域进行扩展：

```
wc.cbWndExtra = DLGWINDOWEXTRA;
```

一旦注册好类以后，应用程序接下来就会调用 CreateDialog () 函数，从而载入对话框模板：

```
#include 〈winuser.h〉
HWND CreateDialog (HINSTANCE hInstance,
                   LPCTSTR lpTemplate,
                   HWND hwndOwner,
                   DLGPROC lpDialogFunc);
```

参数	说明
HINSTANCE hInstance	应用程序实例句柄
LPCTSTR lpTemplate	资源文件里的对话框模板名
HWND hwndOwner	对话框物主窗口的句柄
DLGPROC lpDialogFunc	对话框进程函数的名字
返回值	在正文里讨论
HWND	对话框句柄；假如函数调用失败，则返回一个 NULL 值

CreateDialog () 和 CreateDialogParam () 函数都能读取一个对话框模板，并且生成一个非模态对话框，同时返回自己的窗口句柄。CreateDialog () 的最后一个参数就代表了这个对话框的对话框进程。更准确地说，在这种情况下，窗口的窗口进程是从资源文件里载入的。

```
...
// creating the application main window
hwnd = CreateDialog (hInstance," windlg", GetDesktopWindow (),
    ClientWndProc);
...
```

GetWndProc () 是一种标准的窗口进程，只是在实现的机制上有自己的独到之处。我们可以把它想象成标准窗口进程与对话框进程混合以后生成的一种产物。正是由于它具有这种“混血儿”本性，所以我们既可以对 WM_CREATE 消息进行拦截，也可以对 WM_INITDIALOG 进行拦截。WM_CREATE 是在建立对话框窗口的时候生成的，而 WM_INITDIALO 则是在所有控件都已建好，并且准备在屏幕上显示出来的时候传递给窗口进程的。正是在这个时候，应用程序才会用一系列正文串去填写列表框的具体内容。

```
...
case WM_INITDIALOG:
{
    char szString [30];
    int i;
    HWND hwndList = CTRL (hwnd, CT_LIST);

    for (i = 0; i < 25; i++)
     {
        wsprintf (szString, " String #%02d", i);
        ListBox_AddString (hwndList, szString);
    }
}
    return TRUE;
...
```

所有这些消息最终都会传递给 DefWindowProc () 函数，在其中进行缺省的处理。这就再一次证明了这个例子的窗口本质。WINDLG 除了是一个混血窗口以外，其他就没有什么特殊的地方了。

8.4 对话框模板

DIALOGEX 命令引入了与某个对话框窗口以及相应控件有关的信息块。它完整的语法结构应该象下面这种样子：

```
nemaID DIALOG [load-option] [mem-option] X, Y, width, height [helpID]
[options]
```

其中，载入选项 (load-option) 和内存选项 (mem-option) 在32位编程环境下面几乎是毫无用处，因此可以考虑把它们省略。位置坐标 (X, Y) 和以前一样也是指左上角的坐标，而对话框的大小则是用一个奇怪的单位来度量的，即标准 Windows 字体分别在 X 轴 Y 和轴上尺寸的四分之一和八分之一。

如果想知道一个对话框的基准单位，必须调用 GetDialogBaseUnits () 这个 API 函数：

```
#include <winuser.h>
DWORD WINAPI GetDialogUnits (void);
```

返回值的高字包含了对话框在 X 轴上的基准单位，而低字则包含了在 Y 轴上的基准单位。为了知道对话框的象素宽度，必须用对话框的宽度乘以 X 轴基准单位，然后用4除得到的结果，如下所示：

```
pixelX = (dialogunitX * baseunitX) /4
```

在 Y 轴上则变成了下面这个样子：

```
pixelY = (dialogunitY * baseunitY) /8
```

DIALOGEX 选项

DIALOGEX 命令内包含的选项可以在下面这些值内进行挑选，它们改变了对话框的外观和行为：

CAPTION
CHARACTERISTICS
CLASS
EXSTYLE
FONT
LANGUAGE
MENU
STYLE
VERSION

通过把选项的一部分与传统的窗口风格混合起来，对话框很容易就可以变得跟普通的窗口没有什么两样：带有一个菜单栏、系统菜单以及标题栏右上角的三个通用图标按钮。

不幸的是，Visual C++开发环境的当前版本只提供了针对对话框构建的有限支持，从而限制了它能采用的特性。因此，很常见的做法就是自己动手，修改对话框模板，从而增加 API 级别才提供支持的一些功能。

除了普通的窗口风格以外，对话框还提供了几种 DS_风格，它们明显是针对这种特殊的窗口专门设计的。表8-1为大家列出了所有可用的风格。

表8-2列出了用于在对话框模板内标识预定义窗口的所有命令。

表 8-1　用于外观定制的对话框风格

对话框风格	值	说　明
DS_ABSALIGN	0x0001L	对话框的左上角坐标与屏幕起点相关，而不是与父窗口的左上角位置有关
DS_SYSMODAL	0x0002L	把对话框转换成系统模态窗口
DS_3DLOOK	0x0004L	采用非黑体字，并且使用三维边框
DS_FIXDSYS	0x0008L	使用 SYSTEM_FIXED_FONT 字体
DS_NOFALICREATE	0x0010L	忽略产生的错误，继续建立对话框
DS_LOCALEDIT	0x0020L	过时，现已不用
DS_SETFONT	0x0040L	把模板载入内存的时候，强制 Windows 发送一条 WM_SETFONT 消息，从而为对话框的每一种正文申请对应的字体
DS_MODALFRAME	0x0080L	用一条粗边框把对话框围绕起来
DS_NOIDLEMSG	0x0100L	禁止模态窗口向物主窗口发送一条 WM_ENTERIDLE 消息
DS_SETFOREGROUND	0x0200L	把对话框带入前台
DS_CONTROL	0x0400L	可用于建立特殊的子对话框，它位于另一个定制的普通对话框内部
DS_CENTER	0x0800L	使对话框在屏幕内居中显示
DS_CENTERMOUSE	0x1000L	根据鼠标在屏幕上的位置，使对话框居中显示
DS_CONTEXTHELP	0x2000L	在标题栏的左侧放置一个问号图标

表 8-2　用于标识对话框窗口里预定义控件的命令

命　令	说　明
CHECKBOX	核选框，一种正文形按钮，右侧显示了一个正文串
COMBOBOX	组合框，一个 EDIT 控件与一个 LISTBOX 的组合
CONTROL	控件，对子窗口控件的一种简称

（续）

命　令	说　明
CTEXT	居中正文，STATIC（静态）窗口，内部带有居中显示的正文
DEFPUSHBUTTON	标准的下推式按钮，它带有一个较粗的边框，这表明假如用户按下回车键，就相当于缺省地选中这个按钮
EDITTEXT	属于 EDIT 类的窗口
GROUPBOX	分组框，一个矩形区域，左上角显示有一个正文串
ICON	STATIC（静态）窗口，其中包含了一个图标或者一幅位图
LISTBOX	列表框窗口
LTEXT	左对齐正文，STATIC（静态）窗口，其中包含了左对齐的正文串
PUSHBUTTON	下推按钮，标准的下推按钮
RADIOBUTTON	单选按钮，圆形按钮
RTEXT	右对齐正文，STATIC（静态）窗口，其中包含了右对齐的正文串
SCROLLBAR	滚动栏窗口

8.5 About 框

在本书附带 CD 的 Listing 8.2内，大家能够找到 ABOUT 这个示范程序，它的显示结果如图8-6所示。

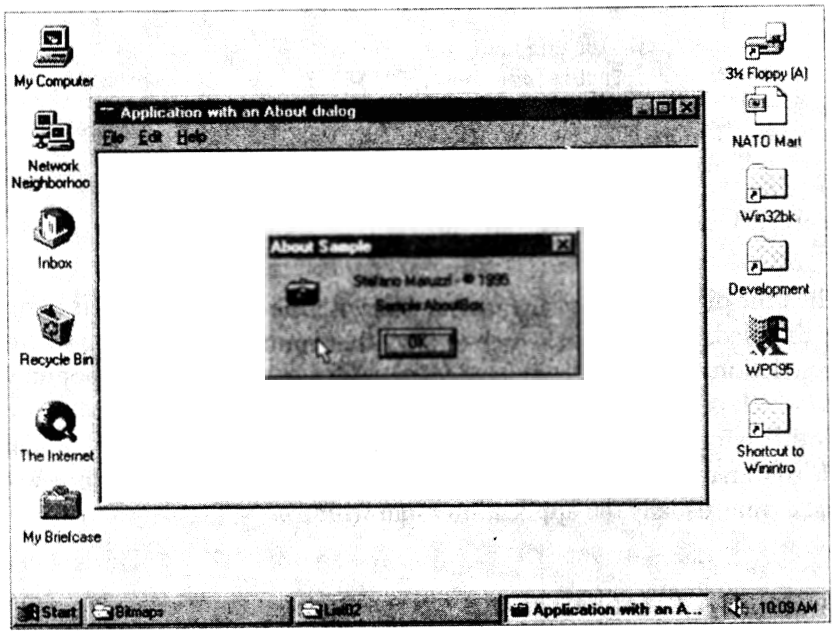


图 8-6　ABOUT 示范程序在屏幕中央显示了一个 About 框

这个程序的对话框进程调用了 WM_COMMAND 消息，从而对按下 OK 按钮的事件进行跟踪。

```
...
BOOL WINAPI AboutDlgProc (HWND hwnd,
```

```
                                UINT msg,
                                WPARAM wParam,
                                LPARAM lParam)
{
    switch (msg)
    {
        case WM_INITDIALOG:
        {
        }
            return TRUE;

        case WM_COMMAND:
        switch (LOWORD (wParam))
        {
            case IDOK:
                EndDialog (hwnd, TRUE);
                return TRUE;

            case IDCANCEL:
                EndDialog (hwnd, FALSE);
                return TRUE;
        }
        break;
    }
    return FALSE;
}
```

对话框处于活动状态的时候，对主窗口进行的任何交互操作都将临时性地屏蔽起来。同时，用户可以切换到正在运行的其他任何一个应用程序，这一点是没有什么限制的。假如应用程序提供了多个叠置式或者弹出式窗口（不是 ABOUT 例子那种情况），用户仍然可以在模态对话框处于活动状态的前提下自由切换到同一应用程序的另外一个窗口。事实上，用于生成对话框的任何 API 在同一时刻都只能屏蔽一个窗口，这个窗口通常都是应用程序主窗口。

8.6 通知代码

和以前的例子比较起来，USCITIES 示范程序（可以在本书附带 CD 的 Listing 8.3里找到）提供的对话框进程要复杂得多。如图8-7所示，这个对话框显示了一个编辑控件、两个按钮以及一个列表框。这些都是属于预窗口类的组件，至于预窗口类，我们会在下一章进行详细的探讨。

假如在 USCITIES 对话框的编辑窗口里键入一个字母，列表框就会自动选择和滚动到以

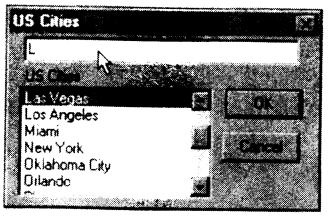


图 8-7 US Cities（美国城市）对话框向我们展示了控件如何利用通知代码以及消息相互间进行通信

那个字母起头的第一个项目处。随后，它就会尝试把选中的项目显示到可视区域的顶部。只有当选中的项目后面跟随的正文串数量足够多时，这种尝试才有可能成功。自动选定一个列表框项目的逻辑并不局限于编辑控件内键入的第一个字符，而需要延展到整个单词以内。举个例子来说，假如用户连续键入了 L 和 O 两个字母，列表框就会选定 Los Angeles（洛杉矶）。除此以外，还要注意到除非进行了一次有效的选择，否则 OK 按钮就一直是屏蔽起来的。换句话说，只有在 EDIT 控件里输入了首字母，并且自己选择或者让系统自动选择了一个城市名以后，才能正常地按下 OK 按钮。

正如我们预先提到的那样，控件是通过一条 WM_COMMAND 消息把自己的通知代码传送给父/物主窗口的，这此代码最终会转移到应用程序的对话框进程内部。下面这个代码段简要阐述了对话框进程处理所有通知代码的方式：

```
...
case WM_COMMAND:
    switch (LOWORD (wParam))
    {
        case IDOK:
            EndDialog (hwnd, TRUE);
            return TRUE;

        case IDCANCEL:
            EndDialog (hwnd, FALSE);
            return FALSE;

        case DL_LIST:
            switch (HIWORD (wParam))
            {
                case LBN_DBLCLK:
                    SendMessage (hwnd, WM_COMMAND, IDOK, 0L);
                    break;
```

```
                case LBN_SELCHANGE:
                {
                    ...
                }
                    break;
            }
            break;

        case DL_EDIT:
            switch (HIWORD (wParam))
            {
                case EN_CHANGE:
                 {
                    ...
                 }
                    break;
            }
            break;
    }
    break;
...
```

列表框的 ID 是 DL_LIST，而编辑控件则是用 DL_EDIT 这个 ID 来标识的。

在这种特定的情况下，控件是通过一条 WM_PARENTNOTIFY 与自己的父窗口通信的：

WM_PARENTNOTIFY	0x0210
LOWORD (wParam)	对事件进行标识。可以在下列值中选择一个：WM_CREATE，WM_DESTROY 或者 WM_xBUTTONDOWN
HIWORD (wParam)	用于 WM_CREATE 或者 WM_DESTROY 事件的控件 ID
lParam	控件句柄或者光标的坐标

8.7 非模态对话框的运用

大家对非模态对话框真有兴趣吗?如果是这样，请参考本书附带 CD 的 Listing 8.4，其中包含了一个名为 MODELESS 示范程序，它向大家展示了非模态对话框建立和使用方面的一些情况。

图8-8可以证明 Color modeless 对话框是一种非模态对话框，因为它具有两种特殊的行

为。首先，它的显示空间是从桌面获得的。其次，尽管当时主窗口已处于活动状态，它仍然能在前台保留下来，这种行为是任何传统的窗口都具备的。

在 WINDLG 这个示范程序里，我们调用 CreateDialog () 这个 API 函数从资源文件里读取一个模板，然后把它转换成非模态对话框。我强烈建议大家使用 CreateDialogParam () 对前一种函数进行扩展，因为它增加了一个特别有用的参数。开发者可以把任何有价值的信息存储到这个参数里，以便对话框进程直接对其进行访问。

正如我们以前指出的那样，非模态对话框与一个标准的窗口是基本一致的，它们都能通过用户的选择接收信息，或者从消息循环里接收信息。假如我们设计的一个应用程序提供了对非模态对话框的支持，就必须稍微修改一下消息循环的结构。具体的做法是增加对 IsDialogMessage () 的一次调用，这个 API 函数可以检查从消息队列里取得的每条消息的目标句柄。

```
#include <winuser.h>
BOOL IsDialogMessage (HWND hDlg, LPMSG lpMsg);
```

参数	说明
HWND hDlg	非模态对话框的句柄
LPMSG lpMsg	一个 MSG 数据结构的地址，其中包含了准备处理的消息
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

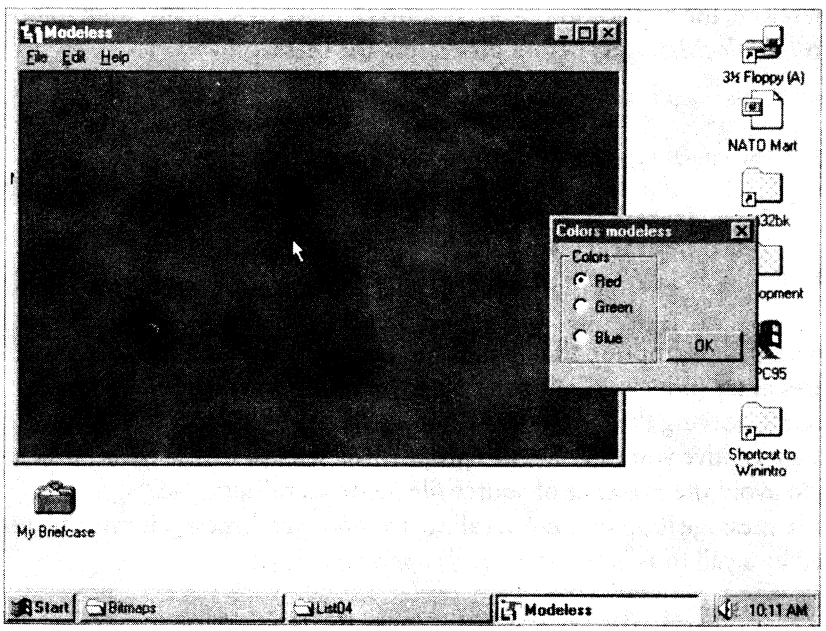


图 8-8 MODELESS 提供了一个非模态对话框，用于改变应用程序的背景颜色

因此，应用程序消息循环会改变自己的标准形式，从而在处理消息之前，先对

IsDialogMessage () 进行一次调用：

```
while (GetMessage (&msg, NULL, 0, 0))
{
    if (!hwndMod || !IsDialogMessage (hwndMod, &msg))
    {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
}
```

由上看出，这种调用的基础是假设 hwndMod 就是非模态句柄。这种方案要求每个非模态对话框的句柄在整个源文件内都是“可见”的。因此，开发者必须在任何代码块的外部声明这些标识符。作为另外一种选择，我们也可以利用属性列表、附加内存或者其他方案来避免使用在整个源文件范围内都有效的标识符。

假如消息循环提供了对 TranslateAcclerator () 的一次调用，那么必须在这次调用之前首先完成 IsDialogMessage () 函数的调用，就像下面这样：

```
while (GetMessage (&msg, NULL, 0, 0))
{
    if (!hwndMod || !IsDialogMessage (hwndMod, &msg))
    {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
}
```

如果想从应用程序代码内部消隐一个非模态对话框，必须调用 DestroyWindow () 函数，同时传递这个对话框的句柄。这一操作可在对话框进程内部或者其他代码段内部进行。

8.8 对话框的收缩

通过 Win32设计的对话框提供了一个标题栏。利用标题栏，用户可以在屏幕上任意移动这些对话框。SHRINKDL 示范程序（可在本书附带 CD 的 Listing 8.5内找到）向大家说明了如果在应用程序进程内部拦截除了 WM_COMMAND 和 WM_INITDIALOG 以外的其他消息，从而对对话框的基本行为进行扩展。在这个程序里，用户可以同时按下鼠标右键和 Shift 键，从而使对话框在垂直方向上缩小。假如重复相同的操作，对话框的垂直尺寸又会恢复到以前的状态，如图8-9和8-10所示。

在对话框进程里，我们捕获了 WM_NCRBUTTONDOWN 消息。第一项处理是用 GetKeyState () 来侦测 Shift 键是否已经按下。假如这个键已被按下，这个函数的高位就会设

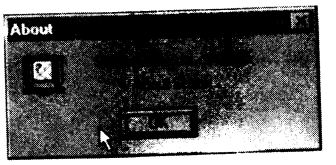


图 8-9　SHRINKDL 对话框最初的尺寸

图 8-10　同时按下鼠标右键和 Shift 键后，SHRINKDL 对话框的样子。它现在只剩下一个标题栏看得见了

置成1；假如测试结构正好相反，应用程序就会离开对话框进程。在 Windows 95里，假如用户在窗口标题栏上单击鼠标右键，就会生成一条 WM_NCRBUTTONDOWN 消息，表明鼠标右键已经按下。这样一来，系统就会发出一条 WM_CONTEXTMENU（关联菜单）消息，从而最终在系统菜单上显示出一个关联菜单。如图8-11所示。

```
case WM_NCRBUTTONDOWN:
{
    static RECT rc;

    // is the SHIFT key down?
    if(!(GetKeyState(VK_SHIFT) & 1<<15))
        return TRUE;

    // if the dialog is already shrunk lets restore it
    if(fShrink == TB_REDUCED)
    {
        SetWindowPos(hwnd,
                     HWND_TOP,
                     0,
                     0,
                     rc.right - rc.left,
                     rc.bottom - rc.top,
                     SWP_NOMOVE);

        fShrink = TB_NORMAL;
        return FALSE;
    }
```

```
        // the dialog is shrinking
        fShrink = TB_REDUCED;

        // dialog dimensions
        GetWindowRect(hwnd, &rc);

        SetWindowPos(hwnd, HWND_TOP,
                     0, 0,
                     rc.right - rc.left,
                     SYS(SM_CYCAPTION)+SYS(SM_CYDLGFRAME) * 2 +2,
                   SWP_NOMOVE);
    }
    return FALSE;
```

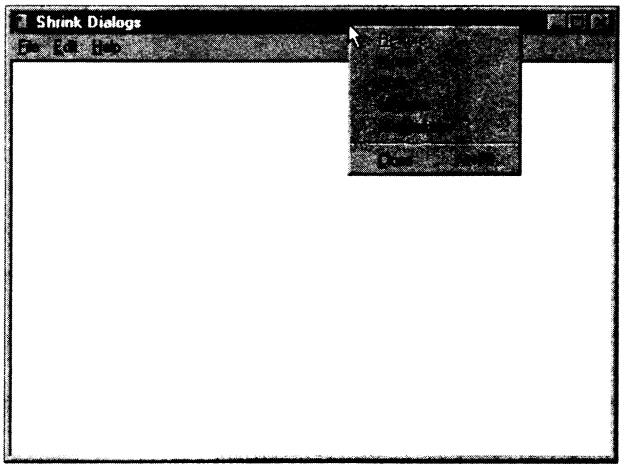


图 8-11　假如在窗口标题栏上按下鼠标右键，系统菜单就会正好在指针位置处显示出来

假如测试成功地通过，程序就会检查 fShrink 这个布尔值是否等于 TB_REDUCED，这个定义表明要对窗口进行缩小处理。新的对话框尺寸会被计算出来，从而使对话框在垂直方向的尺寸等于标题栏的高度，另外加上边框以及一些附加的象素。SetWindowPos () 可以相应地重新定位对话框，并且把它原始的尺寸和位置保存在一个 RECT 结构里。这种信息以后可用于把对话框恢复成它的原始大小。

8.9　通用对话框

Win32应用程序的开发过程中，需要掌握的一个关键点就是通用对话框，这是一系列预先定义好的对话框模板，它们能完成一些普通和基本的任务，比如打开一个文件以及替换正文串等等。Windows 95支持一系列预先定义好的通用对话框，如表8-3所示。

表 8-3 Windows 95里预先定义好的通用对话框

通用对话框	说 明	通用对话框	说 明
Color（颜色）	选择颜色	Page Setup（页面设置）	显示页面设置对话框
Font（字体）	选择字体	Print Setup（打印设置）	访问打印机设置对话框
Open（打开）	访问文件系统	Find（寻找）	查找一个正文串
Save As（另存为）	存储文档	Replace（替换）	查找和替换一个正文串
Print（打印）	显示打印屏幕		

实际上，由于 Windows 95支持两套预定义对话框，所以情况显得稍微有些复杂。如图8-2所示，其中显示的窗口是以 Open 对话框模板作为基础，而这个模板是根据传统的 Windows 3.x 风格编制的。在另外一方面，请大家观察图8-12。其中的 Open 对话框是通过在 MS Paint 里选择 Open 菜单项显示出来的，它看起来和以前的样子颇有不同。在本书附带 CD 的 Listing 8.6里，大家可以找到一个名为 OPEN 的例子。由于这个程序提供了几种不同的通用对话框，所以大家也许觉得它有点“名不符实”（从它的名字推断好象只有一个 Open 对话框）。

开发环境里包括了某些通用对话框的模板，这些模板对应的文件名如下所示。在 COLORDLG.H和 COMMDLG.H 文件里包含了所有 API、定义以及通用对话框的数据类型：

COLOR.DLG
FILEOPEN.DLG
FINDTEXT.DLG
FONT.DLG
PRNSETUP.DLG
MSACMDLG.DLG

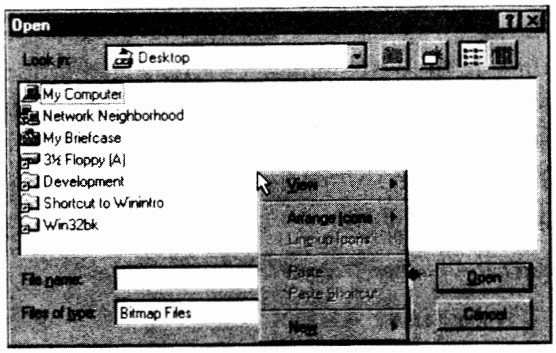


图 8-12 标准的 Windows 95 Open 对话框与系统文件夹有些相似，
它支持外壳提供的所有拖放功能

很有趣的一点是，尽管在开发 Win32应用程序的时候，通用对话框是一项很重要的元素，

但是微软恰恰是第一家没有提供通用对话框支持的公司。如图8-1所示，所有 Office 95应用程序都提供了一种不同形式的对话框，这是介于 Open 和 Find 这两种对话框之间的一种混合物。因此，我猜想我们完全可以开发出与通用对话框不一致的对话框。甚至还可以自由一些，开发者能够选择最适合自己风格、最适合应用程序需求的一种对话框。

建立一个通用对话框

建立一个通用对话框之前，我们需要先进行一些准备工作。首先，必须在项目设置里包含 COMMDLG. H 头文件，并且增加 COMMDLG. LIG 输入库，从而避免发生某些编译和链接错误。如果想建立通用对话框，必须声明属于某一类的一个或者多个据结构，然后对 COMMDLG. H 里原型化的宏函数进行调用，这样才把信息传到实际的对话框里。表8-4为大家列出了与通用对话框有关的所有数据结构，表8-5则总结了通用对话框使用的 API 函数。

表 8-4　用于建立通用对话框的数据结构

数据结构	说　明	数据结构	说　明
OPENFILENAME	打开或者存储文档	PRINTDLG	建立一个打印对话框
CHOOSECOLOR	选择一种颜色	PAGESETUPDLG	建议一个页面设置对话框
CHOOSEFONT	选择一种字体	FINDREPLACE	查找和替换一个正文串

表 8-5　与通用对话框有关的函数集

CommDlgExtendedError ()	返回与通用对话框操作有关的一个出错 ID
ChooseColor ()	显示一个对话框，用于选择一种屏幕颜色
ChooseFont ()	显示一个对话框，用于选择一种字节
FindText ()	显示一个对话框，用于查找一个正文串
GetFileTitle ()	把选定文件的标题返回到通用对话框内
GetOpenFileName ()	建立一个 Open 对话框
GetSaveFileName ()	建立一个 Save As 对话框
PageSetupDlg ()	建立一个页面设置对话框
PrintDlg ()	建立一个对话框，用于启动打印操作
ReplaceText ()	建立一个对话框，用于查找和替换正文串

我们首先准备实现一个 Open 对话框，用它开始我们对通用对话框这个领域的探索。在现实的编程环境中，这种 Open 对话框通常也最先需要设计的几种元素之一。OPENFILENAME 数据结构是相当复杂的，大家看看下面这些语句行就能知道：

```
#include <commdlg.h>
typedef struct tagOPENFILENAME
{     // ofn
    DWORD lStructSize;
    HWND hwndOwner;
    HINSTANCE hInstance;
    LPCSTR lpstrFilter;
    LPSTR lpstrCustomFilter;
```

```
    DWORD nMaxCustFilter;
    DWORD nFilterIndex;
    LPSTR lpstrFile;
    DWORD nMaxFile;
    LPSTR lpstrFileTitle;
    DWORD nMaxFileTitle;
    LPCSTR lpstrInitialDir;
    LPCSTR lpstrTitle;
    DWORD Flags;
    UINT nFileOffset;
    UINT nFileExtension;
    LPCSTR lpstrDefExt;
    LPARAM lCustData;
    UINT (CALLBACK * lpfnHook) (HWND, UINT, WPARAM, LPARAM);
    LPCSTR lpTemplateName;
} OPENFILENAME;
```

尽管上面这一长串定义显得相当恐怖，但是假如仔细观看，大家会发现OPENFILENAME数据结构的定义是相当直观的。这个结构里并非所有项都需要填写恰当的值。一些项是可以省略的，同时不会影响到后面GetFileOpenName ()函数调用的结果。在这儿，我们需要把握一个基本设置是物主窗口的句柄以及对显示文档的类型进行描述的正文串。这个API的最后是一些项是用于接收选中文件名的缓冲区，以及对通用对话框的外观和行为进行定制的一些标志。

表8-6向大家说明了OPENFILENAME数据结构各个项的功用。

下面这个代码段展示了如何利用OPENFILENAME结构只列出那些带有.C和.H扩展名的文件：

```
...
OPENFILENAME ofn;
char szFilter [] = " C source code (*.c) \0*.c\0H headers (*.h) \0*.h\0";
char szCustFilter [60] = "  *.C\0*.h\0", szFile [200] = "", szFileTitle
 [100] = "";

ofn.lStructSize = sizeof (OPENFILENAME);
ofn.hwndOwner = hwnd;
ofn.lpstrFilter = szFilter;
ofn.lpstrCustomFilter = NULL;
ofn.nMaxCustFilter = 0;
ofn.nFilterIndex = 1;
```

由于新的结构出现，该句话的写法会引发兼容性问题，详见头文件定义。VC6可以通过，BC55出错。

```
ofn.lpstrFile= szFile;
ofn.nMaxFile = sizeof (szFile);
ofn.lpstrFileTitle = szFileTitle;
ofn.nMaxFileTitle = sizeof (szFileTitle);
ofn.lpstrInitialDir = NULL;
ofn.lpstrTitle = NULL;
ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST |
OFN_ENABLEHOOK |OFN_EXPLORER;
ofn.nFileOffset = 0;
ofn.nFileExtension = 0;
ofn.lpfnHook = DlgHookProc;
ofn.lCustData = 0;
ofn.lpstrDefExt = (LPSTR) NULL;
...
```

表 8-6 OPENFILENAME 数据结构组成部分

结构项	说 明
lStructSize	数据结构的尺寸
hwndOwner	物主窗口的句柄
hinstance	应用程序实例句柄
lpstrFilter	包含了文档说明和扩展名信息的正文串，这些正文串准备在对话框内显示出来。一个有效的正文串看起来应该类似下面这个样子：" C source code (*.c) \0*.c\0H headers (*.h)\0*.h\0"。这个串必须以一个附加的\0结束。C 源代码和 H 头文件正文串将显示于 Files of Type 组合框内，如图8-16所示。Open 通用对话框的中央部分是一个列表视窗窗口，其中只显示了与每个独立子串的第二部分指定的扩展名(在上面那种情况下面是 *.c 和 *.h) 相符的文档
lpstrCustomFilter	从概念上与上一项完全相同的正文串，它的作用是指定一个定制过滤器
nMaxCustFilter	nMaxCustFilter 缓冲区的大小，通常要大于40字节
nFilterIndex	过滤器正文串的索引，对话框在屏幕中出现时，这个正文串也会显示出来。第一个正文串的索引值是1
lpstrFile	一个缓冲区，用于容纳最初在 File Name 编辑控件内显示的正文串。对话框消失以后，选中文件的完整路径就会返回这个缓冲区内
nMaxFile	lpstrFile 缓冲区的大小
lpstrFileTitle	一个缓冲区，只能用于接收对话框消失后被选中的那个文件名
nMaxFileTitle	lpstrFileTitle 缓冲区的尺寸
lpstrInitialDir	包含某个文件夹名称的缓冲区，这个文件夹会在通用对话框内显示出来。假如为 NULL，就表示准备显示的是当前目录。
lpstrTitle	通用对话框的标题。若为 NULL，就表明使用未被修改的标准标题
Flags	一个或者多个 OFN_标志，这些标志可在表8-7中选用
nFileOffset	lpstrFile 缓冲区内所含文件名第一个字符的偏移量
nFileExtension	lpstrFile 缓冲区内所含文件扩展名第一个字符的偏移量
lpstrDefExt	包含了缺省扩展名的缓冲区，这个扩展名会自动添加到选中文件或者在 File name 编辑控件里输入的文件名的后面
lCustData	指定由应用程序定义的某些数据，假如设置了 OFN_ENABLEHOOK 标志，这个扩展名会自动传递给挂接 (hook) 例程
lpfnHook	挂接例程的名字，该例程类似于一个窗口进程函数，可以接收与通用对话框有关的所有消息，其中包括 WM_INITDIALOG
lpTemplateName	用于建立对话框的模板资源的标签。这一项只有在设置了 OFN_ENABLETEMPLATE 标志的前提下才会生效

所有这些数据随后会传递给 GetOpenFileName（）这个 API 函数，从而在屏幕上实际地显示出通用对话框。假如用户已经选中了一个文件，该函数就会返回一个 TRUE 值。假如 FALSE，则并不一定表明函数调用失败。按下 Cancel 按钮或者选择 Close 菜单项等类似事件都有可能导致通用对话框返回一个 FALSE 值。正如我们以前提到的那样，lpstrFile 项里包含了选中文件的完整路径名，如图8-13所示。

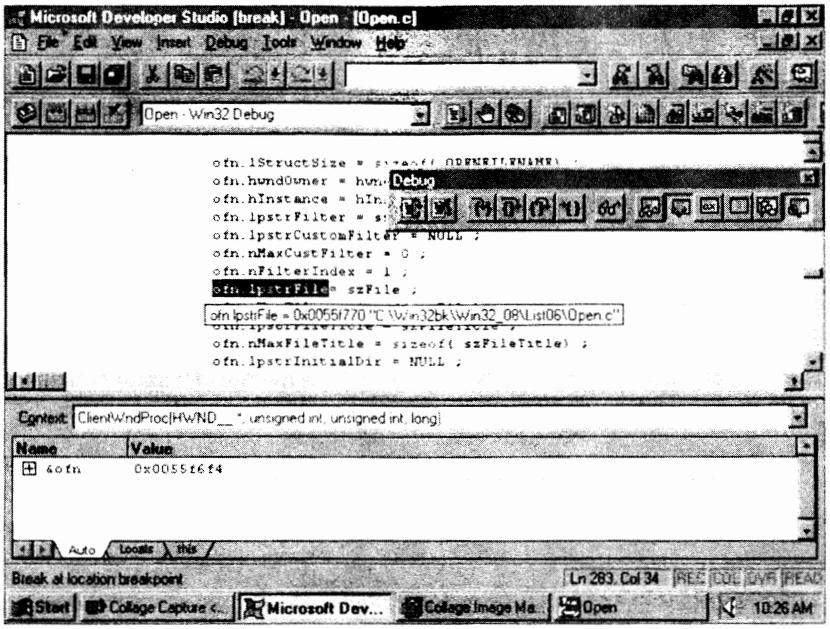


图 8-13　用户选中一个文件并准备打开它时的 OPENFILENAME 数据结构状态

针对通用对话框的外观和行为定义，OFN_标志扮演了一个非常关键的角色。如表8-7所示，其中列出了所有可用的值，以及它们各自的含义。新的 Windows 95外观是由 OFN_EXPLORER 标志的存在来决定的。

表 8-7　OPENFLENAME 数据结构的标志

标　志	值	说　明
OFN_READONLY	0x00000001	Read-Only（只读）核选框处于核选状态
OFN_OVERWRITEPROMPT	0x00000002	假如选中的文件已经存在，就强制 Save As 对话框生成一个消息框
OFN_HIDEREADONLY	0x00000004	隐藏 Read-Only 核选框
OFN_NOCHANGEDIR	0x00000008	把原始文件夹强制用作显示文件的起始位置
OFN_SHOWHELP	0x00000010	显示帮助按钮
OFN_ENABLEHOOK	0x00000020	激活一个挂接进程，从而对通用对话框进行定制
OFN_ENABLETEMPLATE	0x00000040	激活使用由应用程序提供的对话框模板
OFN_ENABLETEMPLATEHANDLE	0x00000080	hInstance 项指定包含了一个通用对话框模板的内存区域
OFN_NOVALIDATE	0x00000100	表明返回的名字也许包含了当前文件不支持的某些字符
OFN_ALLOWMLTISELECT	0x00000200	允许同时选定多个文档
OFN_EXTENSIONDIFFERENT	0x00000400	表明选定文件的扩展名与 lpstrDefExt 里指定的不一致
OFN_PATHMUSTEXIST	0x00000800	强迫用户输入一个有效的路径名
OFN_FILEMUSTEXIST	0x00001000	指出用户只能输入一个现成文件的名字

（续）

标　志	值	说　明
OFN_CREATEPROMPT	0x00002000	假如选中的文件不存在，一个对话框就会提醒用户是否真的想建立它
OFN_NOREADONLYRETURN	0x00008000	指定返回的文件没有选定 Read-Only 核选框，同时并非位于一个写保护的文件夹内
OFN_NOTESTFILECREATE	0x00010000	对话框关闭之前不建立文件（只用于文件保存操作）
OFN_NONETWORKBUTTON	0x00020000	隐藏网络按钮
OFN_NOLONGNAMES	0x00040000	长文件名不在对话框内显示出来
OFN_EXPLORER	0x00080000	应用程序 Win95风格和外观
OFN_NODEREFERENCELINKS	0x00100000	不进行复引用（De-reference）链接
OFN_LONGNAMES	0x00200000	支持长文件名

OPENFILENAME 数据结构的部分项以及一些 OFN_标志不能应用于标准的 Open 对话框。对于这些设置项来说，只能应用于 Save As 对话框的建立。事实上，相同的 OPENFILENAME 数据结构在生成 Save As 对话框的时候也能工作得很好，就像 Open 示范程序那样（请参考本书附带 CD 的 Listing 8.6）。

假如选中 Choose color 菜单项，屏幕就会显示出如图8-14所示的一个 Color 通用对话框。

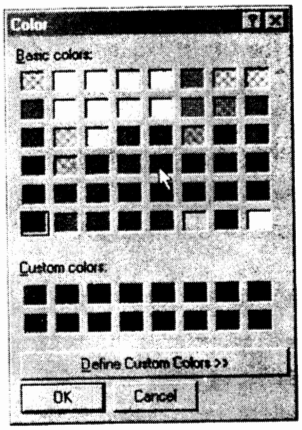


图 8-14　Color 通用对话框允许用户选中一种颜色，把它分配给窗口背景

这个窗口分割成两个部分。在上半部分里，所有预先定义好的颜色都用一个小方格显示出来。在下半部分里，应用程序显示了一些常用的颜色，用户应该首选这些颜色。在 OPEN 这个程序里，Color 对话框的下半部分显示出了以前选中的颜色，这是在 Windows 应用程序里一种很常见的风格。

图8-15显示了 Font 对话框，这是另外一种通用对话框。假设想在应用程序里简便地选取某种字体，这种对话框就显得相当有用了。图8-16则是一个标准的 Open 通用对话框，经过专门的设计后，它可以显示.H 和.C 文件。

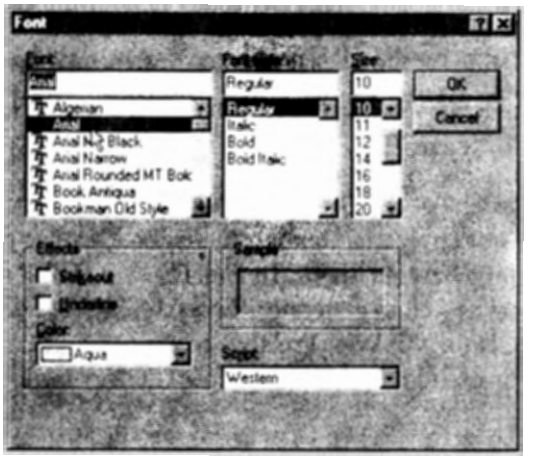


图 8-15　Font 对话框

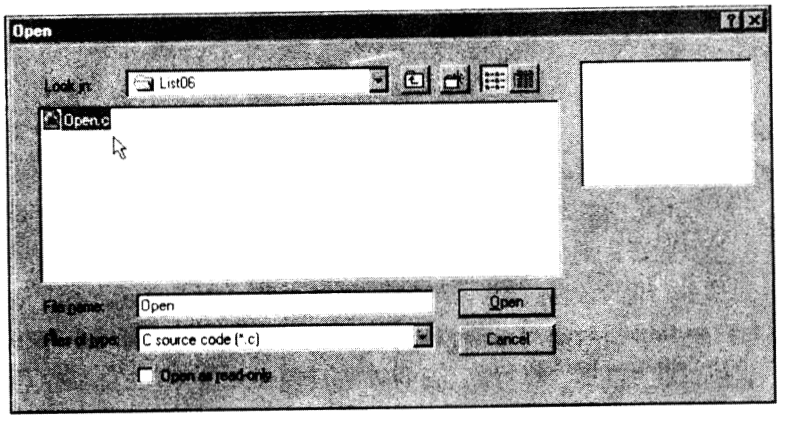


图 8-16　标准的 Open 通用对话框

8.10　对话框的居中显示

Windows 95可以让开发者通过简单的途径让一个对话框在屏幕上居中显示。这种效果是

通过在不同的对话框标志内放置 DS_CENTER 风格获得的。这样一来，对话框就可以在屏幕上自动地居中显示。作为另一种选择，也可以使用 DS_CENTERMOUSE 标志，从而使对话框相对于当前的鼠标位置居中显示。尽管存在这样的一个标志，但是要把它应用于 Open 通用对话框则几乎是不可能的，特别是在这个对话框支持 OFN_EXPLORER 风格的时候。设计对话框模板的时候，由于Visual C++2.x 本身造成的限制，所以设计工作变得更加复杂了。Visual C++ 2.x 的资源编辑器只能支持 Win32 SDK 实现的所有 DS_风格的一部分。DS_CENTER 恰巧不在这一部分里。因此，我们需要自己动手在其中增加 DS_CENTER 风格。然而这也有不便的地方，那就是下一次通过 VisualC++2.x 访问对话框模板的时候，这个风格又不复存在了。Microsoft 开发环境的最新版本 Version 4.0在这个问题上要稍微好一些。它的资源编辑器(Resource Editor)现在能够支持 DS_CENTER 和其他一些新风格，这样就简化了通用对话框的创建。然而，对通用对话框的设计仍然存在着一些问题。

不幸的是，通用对话框在缺省状态下不提供对任何 DS_标志的支持。假如没有 DS_CENTER 标志，通用对话框就会在物主窗口左上角显示出来，那儿正好是应用程序客户区的起点。有两个办法可以临时性地帮我们解决这个问题。在这两个办法中，我们需要用到一个挂接例程。挂接例程可以把消息流暂时转移到由应用程序控制的一个代码段内。OFN_ENABLEHOOK 标志允许开发者在一个 OPENFILENAME 结构里指定一个挂接例程。通过对 WM_INITDIALOG 消息的拦截，我们可以在对话框的预约内存区域内分配一个 DS_CENTER 标志。尽管这种操作很容易实现，但它事实上并不能产生任何效果。只要我们对其进行仔细的研究，就会发现原因其实很简单。DS_CENTER 标志应该分配给最初的那一个父对话框窗口，它不是在挂接例程里传递的窗口句柄。

通用对话框其实就是一种相当复杂的代码段，其中包含了几个窗口以及两个不同的对话框。SPY++（由 Visual C++开发环境提供的一个实用工具）能够让我们更加深刻地理解 Win95建立这种特殊窗口时在幕后发生的情况。我们可以检查一下由 WordPad 显示的 Open 通用对话框，然后就能理解设置了 OFN_EXPLORER 标志以后，通用对话框的实现有多么复杂！

对于一个打开的通用对话框来说，它的本质不外乎是在对话框里显示了几个控件而已。其中有个控件就是另外一个对话框，这个对话框对于用户来说是看不见的。

挂接例程里传递的窗口句柄并不是主对话框的句柄，更准确地说，它是第二个窗口的句柄，这个窗口属于未文档化的＃32770类。因此，挂接例程里首先需要采取的步骤就是获得父对话框的句柄——这才是真正的对话框窗口。一旦获得了这个句柄，接下来就可以强制 DS_CENTER 标志进入该句柄的预约内存区，然后令其在屏幕的中央显示。

另外还可以选择的办法是对通用对话框进行人工定位，使其位于屏幕的中央。这需要根据屏幕的分辨率计算出正确的位置。具体涉及到的算法只需要几条简单的指令即可：

```
...
case WM_INITDIALOG:
{
    RECT rcDlg;
    int cxDlg, cyDlg;
```

```
        GetWindowRect (hwnd, &rcDlg);

        cxDlg = rcDlg.right - rcDlg.left;
        cyDlg = rcDlg.bottom - rcDlg.top;

        SetWindowPos (hwnd,
                      HWND_TOP,
                      SYS (SM_CXSCREEN) / 2 - cxDlg / 2,
                      SYS (SM_CYSCREEN) / 2 - cyDlg / 2,
                      0, 0,
                      SWP_NOSIZE);
    }
        return FALSE;
...
```

仅需对上面这种算法进行少许修改，大家就可以根据不同的情况，让自己程序中的对话框在屏幕上居中显示了。

在通用对话框里还存在着第二个对话框并不是它唯一的特征。正如我们在第十五章“Windows 高级技术”里将要谈及的那样，我们可以在标准的组件里包含一个子对话框，从而对通用对话框进行定制，并且扩展其功能。API 本身是通过 GetOpenFileNamePreview () 函数实现这种技术的。通过这个函数可以生成一个标准的 Open 通用对话框，这个对话框有能力在一个附加的控件里预览一个 AVI 影象剪辑，这样就实现了这种类型窗口的标准外观的扩展。

第 9 章　预定义的窗口类

Win32 提供了七种预定义窗口类。这个数字在许多软件产品中的数目都是固定的，甚至在 MS Windows 95 里引入了新通用控件以后仍然如此（请参考第 10 章“Windows 95 通用控件”）。预定义的窗口类与我们在前一章介绍的注册窗口类非常相似。类数据是为 User DLL 数据区编写的，这些数据与每个单独的窗口有关，而这些窗口则属于这些类的其中一种。

当然，每个类都有自己对应的一个窗口进程，这个进程位于 USER 内。对于由应用程序注册的窗口来说，类窗口进程和前者唯一的区别在于这些类是 Windows 载入的那一刻就存在了，并且会随着操作环境的存在而一直延续下去。在某些示范程序里（比如 WORDPRO），我曾经使用了一个属于预定义 EDIT 类的窗口，我们那时对它的了解还是很一般的。

在以前的例子里，为了建立一个 EDIT 类窗口，只需采用 CreateWindowEx（）的传统语法进行设置即可。表 9-1 列出了 Win32 里的所有这七个类，它们的名字必须当作 CreateWindow（）的第一个参数进行传递，或者作为 CreateWindowEx（）的第二个参数进行传递。因此，假如我们需要建立一个属于 EDIT 或者 BUTTON 类的窗口，就没有必要进行任何注册。其中的原因很简单，因为这种注册操作是由 Windows 95 在系统装载的时候自动完成的。在本书附带 CD 的 Listing 9.1 内，大家可以找到一个名为 CONTROLS 的示范程序（执行结果如图 9-1 所示）。建立这个程序的意图是向大家展示属于不同预定义窗口类的所有窗口的外观。

表 9-1　Win32 的七种预定义窗口类

预定义类	含　义	预定义类	含　义
BUTTON	按钮	MDICLIENT	多文档界面客户区
COMBOBOX	组合框	SCROLLBAR	滚动条
EDIT	编辑框	STATIC	静态窗口
LISTBOX	列表框		

属于预定义类的窗口几乎肯定是当作对话框内的一个子窗口使用的。然而，最近以来，它们的应用领域已经得到了扩展，就像图 9-1 显示的那个叠置式窗口一样。在最初的时候，预定义窗口类的宗旨是为用户提供一种良好的操作方式，并且使应用程序的设计具有良好的风格（比如在载入一个文件的时候）。由于这个原因，预定义窗口类也被简单地称为“控件”。然而，“控件”这种称呼并不包含 MDICLIENT 这个预定义类。MDICLIENT 是由 Windows 3.0 引入的，它的作用是简化 MDI（多文档界面）应用程序的开发（请参阅第 15 章“Windows 高级技术”）。

如图所示，屏幕左侧的最后那个对象是一个按钮，它用一个图标取代了传统的正文串（也可以显示位图）。右侧显示的是本书的封面，它在一个静态窗口内显示了一幅传统的位图。这是一种新的 Win32 特性，功能很强大，亦易于使用。我们只需简单地用 SS_BITMAP 风格建立一个 STATIC 窗口，然后用一个位图资源标签取代对象标题。这样就可以生成图 9-1 那

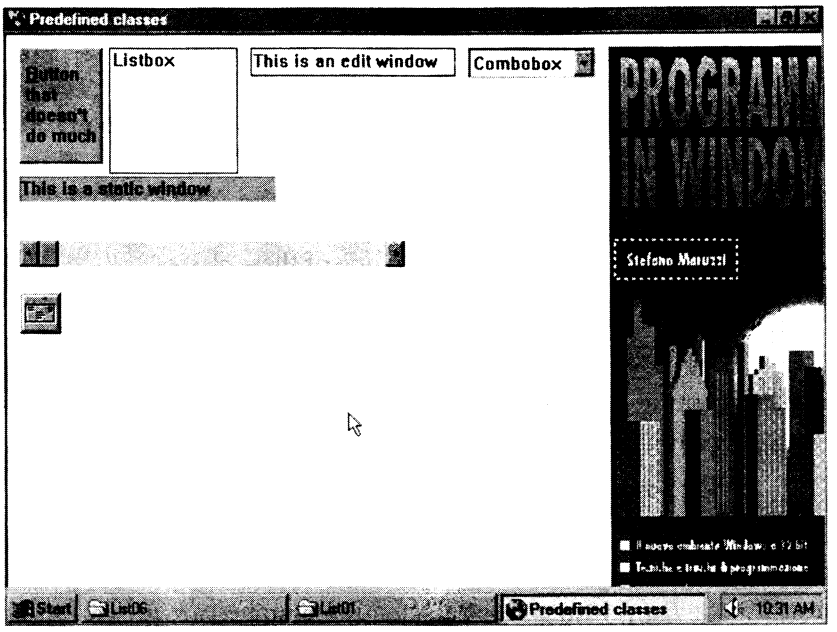


图 9-1　在 CONTROLS 程序里，大家可以看到六种预定义窗口类的示范；这些窗口可以当作控件使用

样的结果。

9.1　控件的建立

在这儿，“控件”只是一个普通的窗口而已，它属于随同 Windows 95 引入的某个预定义窗口类。然而，与应用程序注册的那些类不一样，控件的运用更灵活，功能也更强，因为它们可以满足许多场合下的编程需要。和窗口一样，即使控件也可以延用我们在前一章介绍的那些规则。然而，控件也有它们自己的一些特征，如下所示：

- ▶风格
- ▶消息
- ▶通知代码

这些特征能够对控件的建立、管理以及与构成整个应用程序的其他窗口的交互作用产生一定的影响。风格和数据结构在控件建立的时候就扮演了一种非常重要的角色，而消息和通知代码则要等到用户使用一个控件的时候才会发挥其功效。

9.1.1　关于风格

现在让我们来看看 CreateWindowEx () 用于建立一个控件的语法。指定了一种或者多种扩展风格以后，我们还需要指定类名称，并且用双引号把它封闭起来，然后再附上窗口的标题。其中，第三个参数（窗口标题）有一些特殊的地方。对于传统的窗口来说，第三个参数与标题栏内显示的正文对应，所以我们需要用到 WS_CAPTION 风格。然而，对于某些预定义窗口类来说，第三个参数却具有不同的含义。

需要考虑的第一件事情是 WS_CAPTION 标志的使用与否。对于属于 SCROLLBAR，

LISTBOX，COMBOBOX，EDIT，STATIC 或者 BUTTON 类的窗口来说，标题栏的使用与否是没有什么意义的。因为标题的作用就是允许用户在屏幕上到处移动窗口而已。当然，对于预定义窗口来说，这种“全屏幕移动”功能是我们绝对不允许的。因此，根据 Windows 95 的风格规范，控件的建立语法要求禁止 WS_CAPTION 的使用。说到这里，大家也许会认为 CreateWindowEx （）的第三个参数肯定应该设置成 NULL。然而，这种猜测对于 EDIT，STATIC 和 BUTTON 类来说又是不准确的。

在 EDIT 窗口、STATIC 窗口或者按钮里，假如存在“标题栏”字符串，那么对应就要求用户在其中插入一些正文信息。通过图 9-1，大家可以看出这一点。

```
...
CreateWindowEx (0L,
                " BUTTON"," Close",
                WS_CHILD | BS_PUSHBUTTON | WS_VISIBLE,
                70, 10, 100, 100
                hwnd,
                 (HMENU) CT_PUSH,
                hInstance,
                NULL);
...
```

对于风格来说，CreateWindowEx （）的第四个参数是一个 32 位的 DWORD（双字）。在这 32 个二进制位里，16 个是为 WS_风格保留的，而属于低位的 16 个二进制位则特定于每一种预定义类。每一类控件都有自己的一系列风格定义，这些风格会对所建对象的“外观和感觉”发生影响。软件设计者必须知道每个标志对所建窗口产生的效果，因为不同的控件之间存在着很大的区别。例如，对于 BUTTON 类来说，其中的 BS_PUSHBUTTON 风格会生成一个普通的按钮，而 BS_RADIOBUTTON 则会生成一个单选按钮，两者的外观是完全不同的。

在各种 WS_风格中间，大家几乎肯定会找到 WS_CHILD。在程序里，需要把一个控件当作主窗口（叠置式窗口）或者附属窗口（弹出式窗口）使用的情况很少见。除此以外，我们通常可以使用一个 WS_VISIBLE 风格。这样一来，窗口的建立和显示仅需一个简单的步骤便可完成。

窗口的位置和尺寸是用父窗口的单位来度量的。在子窗口里使用 CW_USEDEFAULT 标志的情况只能限于几个特殊的类。我们通常都用父窗口的尺寸来表达一个控件的位置和尺寸。下一个参数是父窗口的句柄：必须指定这个句柄，用它来提醒系统为新建立的窗口提供显示象素。

正如我们以前提到的那样，子窗口不能使用菜单栏。为一个子窗口分配菜单句柄的尝试是徒劳的，因为这种操作不会产生任何效果。取而代之，我们可以在指定菜单句柄的地方分配一个 ID，这个 ID 可以是任何一个整数。这样一来，父窗口就可以用一种更简便的方法对自己的控件进行控制。对控件的访问既可以通过窗口句柄实现，也可以通过该控件的 ID 实现。

正如大家以后会看到的那样，Win32 API 提供了许多有用的工具，这些工具都能通过设计者分配的 ID 与控件进行交互作用。假如忘记分配 ID，或者不愿意为控件分配 ID，那么对于前一种做法，我们应该力图避免；后一种做法则是相当不明智的。

9.1.2 消息和控件

调用 CreateWindowEx () 函数来建立控件的时候，产生所有消息都会发送给 USER 里对应类的窗口进程。在表 9-2 里，大家能看到一些函数的地址，它们在六类控件使用的窗口进程里都扮演了关键性的角色（这些值会根据应用程序的不同而发生变化）。

表 9-2 用于所有窗口控件的类和窗口进程地址

类	窗口进程地址	类	窗口进程地址
BUTTON	0x8013dc56	COMBOBOX	0x8013dc98
LISTBOX	0x8013dc6c	STATIC	0x8013dcae
EDIT	0x8013dc82	SCROLLBAR	0x8013dcc4

很显然，假如用户与任何一个属于预定义类的窗口进行交互作用，那么这时生成的消息完全不会对应用程序产生任何影响。举个例子来说，假如在一个列表框上面按下鼠标左键，就会生成一条 WM_LBUTTONDOWN 消息。这条消息会发送给属于对应类的窗口进程。随后，这个进程会采取相应的措施，从而对鼠标的单击事件进行响应。就列表框内发生的 WM_LBUTTONDOWN 事件来说，进程响应的结果就是鼠标光标下方的项目被选中。通过通知代码，预定义类窗口才能与自己的父/物主窗口链接到一起。

预定义类另外一个特殊的地方在于它可以使用一系列消息，这些消息允许一个应用程序与属于那一类的某个窗口进行交互作用。这其实就是我们刚才讨论的那种运作机制的一种直接性后果，那种机制涉及到程序代码与预定义类代码之间的本质性区别。只有把对应的消息发送给一个类，应用程序才能判断用户进行了什么操作（选择了列表框内的一个项目吗？如果是这样，那么选择的是哪个项目呢?），然后才能据此调整自己的行为。

9.1.3 通知代码

对六种控件类进行研究的时候，大家或许已经看出来了，控件和应用程序之间的所有交互作用都是在通知代码和消息的基础上完成的。由于发送给应用程序主窗口的消息流和发送给控件的消息流是完全分隔开的，所以需要在应用程序的各个实体之间实施某种交互作用机制。通知代码就是这样的一种方案。通知代码的本质是一种信息，它从一个控件的窗口进程发送给自己的物主/父窗口，通知它在控件内已经发生了某种事件。下面这个例子会帮助我们理解这一概念。

现在假如用户在一个列表框上方按下了鼠标左键，这种操作表明该用户希望选择列表框内的一个项目。对于 LISTBOX 窗口类来说，这种操作将导致一条 WM_LBUTTONDOWN（左键按下）消息的产生。我们不知道这条消息的处理细节，但是最终的结果却是明摆着的。鼠标指针热点下方的正文串将以反转色显示，从而表明自己已被选中。一旦这种响应结束以后，窗口进程就会把通知代码发送给自己的物主/父窗口。

通过对接收到的通知代码进行检查，父/物主窗口就能“推断”出某个特定控件内发生了什么事件。几乎所有通知代码都是由 WM_COMMAND 消息传输的，只有一个例外，那就是

SCROLLBAR 类。应用程序捕获了 WM_COMMAND 消息以后，它首先对通知代码的相应含义进行理解，然后用恰当的类消息对发送端窗口进行查询。受到一条通知代码的激发，应用程序可以向那种控件类发送一条或者多条特定的消息，从而获取更详细的信息，利用这些信息就可以继续程序的执行。

总而言之，假如使用了一个属于预定义类的窗口，就有必要仔细考虑下面这三个方面的问题。在这一章的后面部分，我们将针对每个类依次对它们进行讨论：

▶风格

▶通知代码

▶消息

在本书附带 CD 的 Listing 9.2 里，大家可以找到一个名为 CLASSES 的程序示例，它能提取出与某个 Win32 预定义窗口类有关的信息。为了获得对窗口类进行描述的信息，我们有两个办法可以选择。一个是调用 GetClassInfo（）或者 GetClassInfoEx（）函数，另一个则是先建立一个窗口，然后调用 GetClassLong（）函数。

利用 GetClassInfo()，我们可以在一个 WNDCLASS/WNDCLASSEX 数据结构里填写与这种数据项有关的所有信息，只是要把与菜单资源有关的名称除外。之所以要把菜单资源排除在外，是因为这六个控件类中没有一个与菜单在类级别上有关。下面是该函数的语法结构：

```
#include〈winuser.h〉
BOOL WINAPI GetClassInfoEx（HINSTANCE hInstance，
                            LPCSTR lpszClass，
                            LPWNDCLASSEX lpwcx）；
```

参数	说明
HINSTANCE hInstance	用于类注册的应用程序实例句柄。对于预定义类来说，必须将其指定为 NULL
LPCSTR lpszClass	准备检查的那个类的名称
LPWNDCLASSEX lpwcx	一个 WNDCLASSEX 结构的地址，其中需要填写相应的类信息
返回值	在正文里讨论
BOOL	应用程序的输出结果

第一个参数与类注册时使用的实例句柄对应，并且在当前这种应用场合下必须设置成 NULL。如图 9-2 所示，CLASSES 应用程序的显示结果看起来和前一个例子 CONTROLS 是类似的。但是事实上，前者提供了一些附加的特性。例如，它在列表框窗口里插入了 10000 个正文串。

通过 CLASSES 程序的运行，大家也可以看出它产生的一些副作用。应用程序启动的时候，屏幕上会显示出一个列表框。在其他控件完全显示出来之前，我们必须等上几秒钟的时间（等待的时间取决于 CPU 的性能）。在这段等待时间内，CLASSES 要忙于在列表框内填写 10000 个正文串；即使在 Pentium 系统里，这都是一种相当耗时的操作。显然，这段代码根本没有进行任何优化处理。在第十四章“多线程、IPC 和 I/O”里引入了次级线程后，我们会再

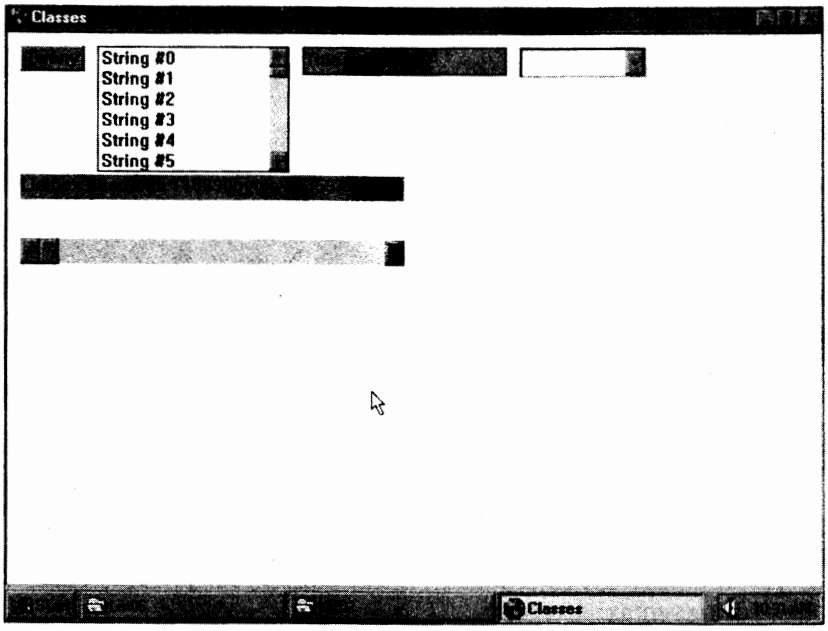


图 9-2　CLASSES 程序在应用程序客户区内显示了一些控件。
列表框内填写了 10000 个正文串

对这段代码进行适当的优化。

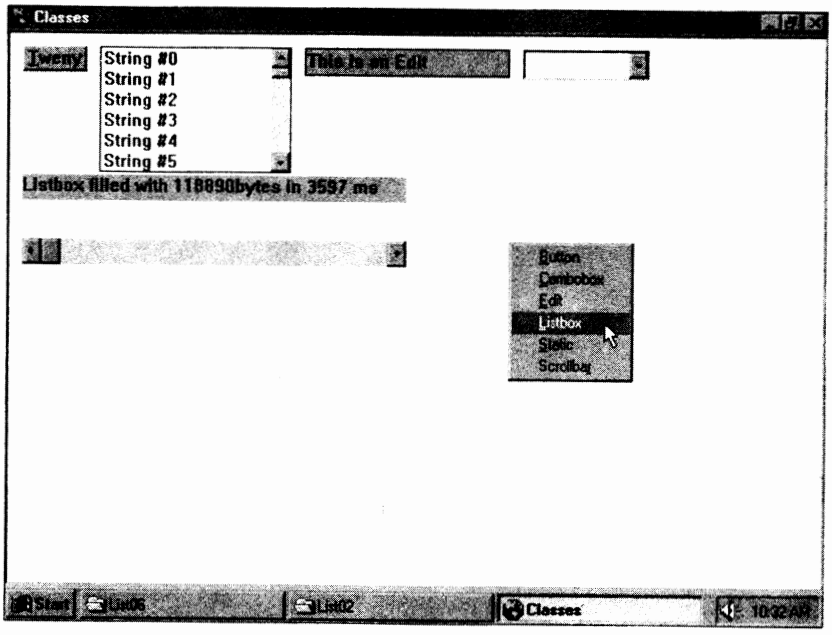


图 9-3　CLASSES 显示了一个弹出式菜单，这是单击应用程序客户区的结果

除此以外，CLASSES 程序还向我们展示出现在的一个 Win32 LISTBOX 对象可以接收多于 64KB 的信息。事实上，这 10000 个正文串的总容量已经超过了 118KB，这在 STATIC 窗

口里已经显示出来了（同时还显示了列出这些串耗费的时间），这是一种非常合理的容量。除此以外，假如在应用程序客户区的某个空白位置单击鼠标右键，就会在那个位置显示出相应的弹出式菜单，如图 9-3 所示。

在这个弹出式菜单内，列出来的六个项目对应于六种预定义窗口类，这些窗口类均可当作控件进行处理（已经排除了 MDICLIENT 类）。假如选择这些选项中的一个，就会显示出一个弹出式窗口，其中包含了与那一类有关的所有信息，如图 9-4 所示。

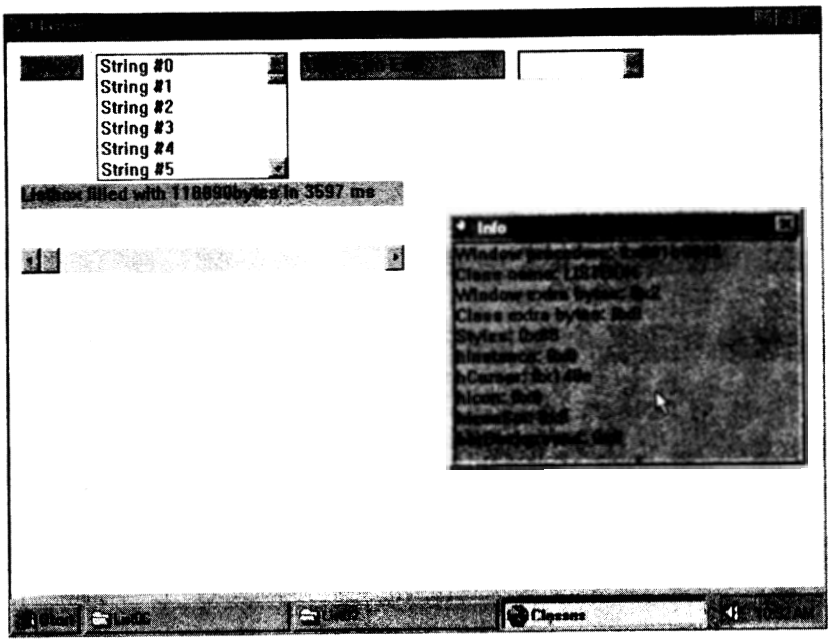


图 9-4　Info 窗口内显示了与预定义窗口类有关的所有信息

Info 窗口向我们报告了类窗口进程、类名以及与 WNDCLASSEX 数据结构有关的其他所有信息。

9.2　列出 Win32 进程

尽管 Win32 应用程序的开发过程充满了挑战和趣味，然而有些时候也会碰到一些预想不到的障碍。Zombies（僵尸）也许是我们经历的最糟糕的一种情况。什么是僵尸呢？从根本上说，它们是一些仍然在系统内运行的 Win32 进程，但是不能看到任何窗口。由于缺少窗口可供利用，所以我们无法使这种进程中断（破坏一个窗口是标准中断进程的一部分，尽管这两件事情并不是等效的）。在本书附带 CD 的 Listing 9.3 里，大家可以找到一个名为 TOOL 的应用程序，它为我们提供了一个机会对 Windows 95 引入的新的 TOOLHELP32 API 函数进行探究。该函数专门设计用于列出系统内正在运行的所有进程。图 9-5 展示了 TOOL 程序执行后的显示情况。

客户区覆盖了一个 LISTBOX 窗口，其中包含系统内当前正在运行的所有进程的全名。正如大家看到的那样，TOOL 也是其中的一个。用 TOOL 能做什么呢？假如双击一个进程的名字，就能强迫它中断，如图 9-6 所示。

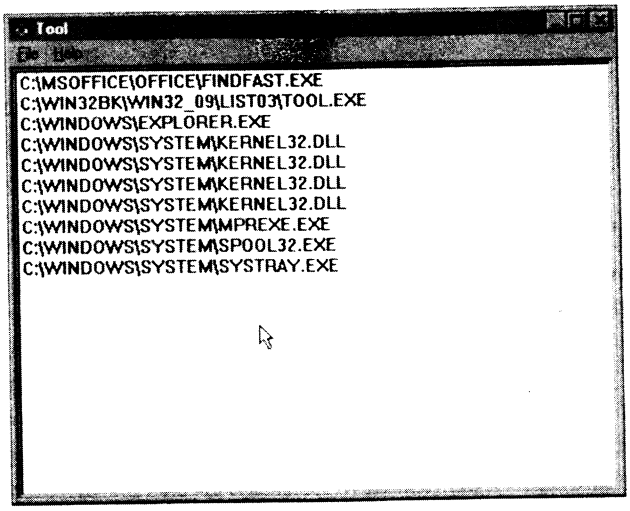


图 9-5　TOOL 窗口列出了正在运行的所有进程；按下 F5 键可对列表进行更新

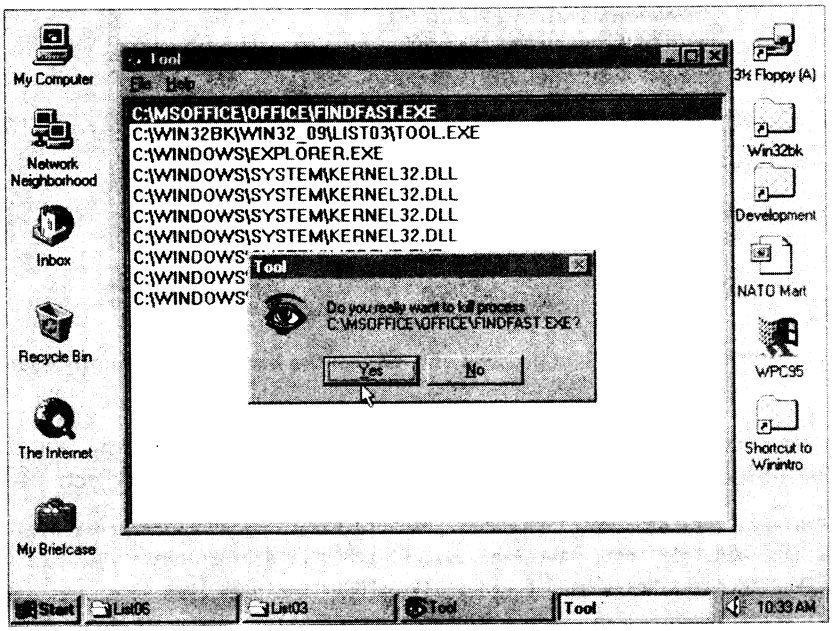


图 9-6　中断一个正在运行的进程之前，TOOL 会要求确认

相同的结果也可以换种方法得到，我们可以在列表框内选择一个进程，然后选择应用程序主菜单内的 Kill 菜单项即可（当然，用一个弹出式菜单来完成这种操作也许要更好此，但是在这样做之前，大家必须先学会如何建立子类）。假如按下 F5 加速键，或者选择 Refresh list 菜单项，TOOL 程序就会重新填写系统进程列表，从而对自己的输出进行更新，如图 9-7 所示。除此以外，再次激活 TOOL 的时候，它也会对列表进行更新。

TOOL 程序的结构是相当简单的，主要依赖于 TLHELP32.H 包容文件里的列出的 API

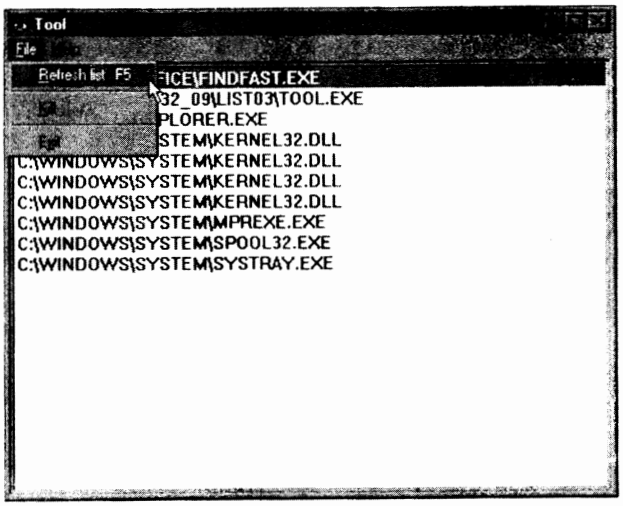


图 9-7　TOOL 允许用户中断一个进程，并且对活动进程列表进行刷新

函数。这个 API 函数在 Windows 95 的 SDK 里仍然没有进行具体的文档化处理；另外，它在 Windows NT 里也是不能使用的。LEHELP32. H 里包含了几种函数原型，然而我们只对这儿涉及到的一种原型进行探讨。

```
HANDLE WINAPI CreateToolhelp32Snapshot (DWORD dwFlags, DWORD
    th32ProcessID);
BOOL WINAPI Module32First (HANDLE hSnapshot, LPMOUDULEENTRY32
    lmpe);
BOOL WINAPI Module32Next (HANDLE hSnapshot, LPMOUDULEENTRY32
    lmpe);
BOOL WINAPI Process32First (HANDLE hSnapshot, LPPROCESSENTRY32
    lppe);
BOOL WINAPI Process32Next (HANDLE hSnapshot, LPPROCESSENTRY32
    lppe);
```

第一个函数原型是 CreateToolhelp32Snapshot ()（注意，Snapshot 是“瞬态图”的意思），它从内部对系统的不同部分进行了分析，比如正在运行的进程和线程，或者现成的模块等等。它的表现就像一个初始化函数——可以返回一个句柄，这个句柄对于以后对其余每个 TLHELP32 函数的调用都是非常有用的。拦截到 WM_CREATE 消息以后，TOOL 程序就会调用 CreateToolhelp32Snapshot () 函数。接下来再调用一个 ProcessList () 函数，这个函数在 TOOL 程序里用于完成由 CreateToolhelp32Snapshot()“复印”下来的进程列表。Tool Help API 包含于 KERNEL32. DLL 模块里，然而它在任何一种系统输出库里都没有输出。因此，我们必须从外部输入这些函数，方法是在应用程序的 DEF 文件里使用下面这个代码段：

```
...
IMPORTS
```

```
KERNEL32. CreateToolhelp32Snapshot
KERNEL32. Module32First
KERNEL32. Module32Next
KERNEL32. Process32First
KERNEL32. Process32Next
...
```

通过明确地包含这些函数，现在就可以从应用程序的源代码内调用它们了。然而，在TOOL 示范程序里，我按照函数地址的使用形式采取了另外一种不同的办法。应用程序将在WM_CREATE 消息里载入 KERNEL32. DLL 库，然后通过 GetProcAddress () 函数有选择地取回进程地址，如下所示：

```
...
// get the KERNEL32. DLL handle
hKernel = GetModuleHandle (" KERNEL32. DLL");
...
// get the pointers to these APIs
pCreateToolhelp32Snapshot = (CREATESNAPSHOT) GetProcAddress (hKernel,
" CreateToolhelp32Snapshot");
pMod32F = (MODULEWALK) GetProcAddress (hKernel," Module32First");
...
```

不同 GetProcAddress () 调用的返回值存储在一些静态标识符里，这些标识符在整个源代码的范围内都是“可见”的。通过 typedef 关键字，我们可以事先把这些标识符定义成指向对应函数的指针：

```
...
typedef BOOL (WINAPI * MODULEWALK) (HANDLE hSnapshot
    LPMODULEENTRY32 lpme);
...
```

经过了这些初始化设置工作以后，TOOL 就会放置一个 LISTBOX 窗口，用它覆盖整个客户区域，然后把系统内存里取得的所有活动模块的名字填入其中。这种操作是通过调用 Process32First () 函数和 Process32Next () 这两个 Tool Help API 函数来完成的。这两个函数在一个 PROCESSENTRY32 数据结构里填写了许多有用的信息，其中包括进程 ID 以及对应的优先级别，如下所示：

```
typedef struct tagPROCESSENTRY32
{
    DWORD dwSize;
    DWORD dntUsage;
```

```
    DWORD th32ProcessID;
    DWORD th32DefaultHeapID;
    DWORD th32ModuleID;
    DWORD cntThreads;
    DWORD th32ParentProcessID;
    LONG pcPriClassBase;
    DWORD dwFlags;
} PROCESSENTRY32;
```

每个进程的模块名称都是通过 Process32First () 和 Process32Next () 函数侦测出来的。随后，这些名称将插入列表框内，并且在与每个项目相关的一个专用内存区域里把对应的 PID (进程 ID) 存储下来。模块名称是 Module32First () 和 Module32Next () 函数对系统内存进行另外一次检查得到的结果，这两个函数的作用是对 MODULEENTRY32 数据结构进行填写：

```
typedef struct tabMODULEENTRY32
{
    DWORD dwSize;
    DWORD th32ModuleID;
    DWORD th32ProcessID;
    DWORD GlblcntUsage;
    DWORD ProccntUsage;
    BYTE * modBaseAddr;
    DWORD modBaseSize;
    HMODULE hModule;
    char szModule [MAX_MODULE_NAME32 + 1];
    char szExePath [MAX_PATH];
} MODULEENTRY32;
```

中断这一部分程序之前，需要先关闭由 CreateToolhelp32Snapshot () 函数返回的句柄，这一工作通过 CloseHandle () 函数完成的。因为放置到列表框内的信息应该反应了连续变化的系统环境，所以 TOOL 程序还提供了 Refresh list (更新列表) 菜单项，从而在用户需要的任何时刻执行进程搜索，这个菜单项亦可通过按下 F5 键激活。

9.3 六种预定义的类

通过本章前面举的这三个例子，大家对 Win32 控件应该多少有些熟悉了。接下来，我们应该理解如何利用它们对 Windows 进程的功能进行扩展。掌握了本章后面部分讲述的这些知识以后，大家就可以胸有成竹地对这六种预定义类进行实际运用了。

3.1 BUTTON 类

Windows 内出现的所有按钮都归属于 BUTTON 类，尽管它们在屏幕上也许会以形形色色不同的外观出现。事实上，这种预定义类牵涉到核选框、单选按钮以及分组框等等。不同形状按钮的生成是由一些特殊的标志决定的，这些标志可以在建立窗口的时候设置。表 9-3 为大家总结了所有的 BS_风格，这些风格只能由 BUTTON 类使用，它们也是对不同类型按钮进行区分的方式。

表 9-3 BUTTON 类使用的风格

风 格	值	说 明
BS_PUSHBUTTON	0x00000000L	建立一个传统的下推式按钮
BS_DEFPUSHBUTTON	0x00000001L	让一个传统的下推式按钮成为缺省按钮（假如按下 Enter 键，就相当于缺省地按下这个按钮）
BS_CHECKBOX	0x00000002L	建立一个核选框
BS_AUTOCHECKBOX	0x00000003L	建立一个核选框，它的选择是自动的
BS_PUSHBUTTON	0x00000004L	建立一个单选按钮
BS_3STATE	0x00000005L	建立具有三种状态的一个核选框：被选择、没有选择以及不明确选择
BS_AUTO3STATE	0x00000006L	建立具有三种状态的一个核选框（被选择、没有选择以及不明确选择），并且自动选择
BS_GROUPBOX	0x00000007L	建立一个矩形框，并在左上角显示一个标签
BS_USERBUTTON	0x00000008L	用户自定义按钮
BS_AUTORADIOBUTTON	0x00000009L	自动选择的单选按钮
BS_OWNERDRAW	0x0000000BL	物主绘图按钮
BS_LEFTTEXT	0x00000020L	左对齐正文
BS_TEXT	0x00000000L	指出按钮内是否显示正文
BS_RIGHTBUTTON	0x00000020L	强迫正文串显示于核选框或者单选按钮的右侧
BS_ICON	0x00000040L	按钮内显示一个图标
BS_BITMAP	0x00000080L	按钮内显示一幅位图
BS_LEFT	0x00000100L	按钮内的正文左对齐
BS_RIGHT	0x00000200L	按钮内的正文右对齐
BS_CENTER	0x00000300L	按钮内的正文居中显示
BS_TOP	0x00000400L	把按钮正文放置于顶部
BS_BOTTOM	0x00000800L	把按钮正文放置于底部
BS_VCENTER	0x00000C00L	在垂直方向中居中显示按钮正文
BS_PUSHLIKE	0x00001000L	应用于下推按钮、核选框、单选按钮或者三状态核选框的风格，但是不能应用于其他按钮
BS_MULTILINE	0x00002000L	假如按钮正文超出了按钮的宽度，就用多行显示这些正文
BS_NOTIFY	0x00004000L	把物主/父窗口发送给通知消息

预定义窗口类的特征表现在三个方面：一系列保留风格（参考表 9-3）、一些专用消息（参考表 9-4）以及一系列通知代码（参考表 9-5）。和其他所有控件一样，BUTTON 类的这些消息是用 WM_USER 起头开始定义的。因此，我们可以更准确地把它们考虑成一些简单的数值重定义，它们在窗口的常规管理中是没有使用的。

BM_消息的作用是很明显的。用户对一个按钮执行的特定操作从直观上说是可见的。然而，对于应用程序来说，它却完全不知道用户的这种操作，因为这种操作不会生成能直接到达应用程序窗口进程的消息流。假如程序某部分需要关于 BS_3STATE 核选框的状态信息，就必须依赖 BM_GETSTATE 消息，并且把它定址于某个 BUTTON 类窗口。在 BUTTON 类

窗口进程里，存在一个 0x00F2 条件，它是由某条代码生成的，这条代码的作用就是把核心的当前状态反馈回呼叫者。

表 9-4 BUTTON 类使用的消息

消息	值	说明
BM_GETCHECK	0x00F0	返回一个单选按钮或者核选框的选定状态
BM_SETCHECK	0x00F1	设置一个单选按钮或者核选框的选定状态
BM_GETSTATE	0x00F2	返回一个按钮和一个核选框的状态
BM_SETSTATE	0x00F3	设置一个按钮和一个核选框的状态
BM_SETSTYLE	0x00F4	修改按钮的状态
BM_CLICK	0x00F5	模拟一次鼠标单击
BM_GETIMAGE	0x00F6	返回与按钮关联起来的位图或者图标的句柄
BM_GETIMAGE	0x00F7	为按钮分配一幅位图或者一个图标

表 9-5 BUTTON 类使用的通知代码

标识代码	值	说 明
BN_CLICKED	0	用户按下一个按钮以后发出
BN_PAINT	1	正好在描绘按钮前发出
BN_HILITE	2	选定按钮后发出
BN_UNHILITE	3	撤销对一个按钮的选定后发出
BN_DISABLE	4	BN_NOTIFY 按钮准备屏蔽的时候发出
BN_DBLCLK	5	用户双击带有 BS_OWNERDRAW 或者 BS_RADIOBUTTON 的一个按钮时发出
BN_SETFOCUS	6	按钮获得视觉焦点后发出
BN_KILLFOCUS	7	按钮失去视觉焦点后发出

为了把一幅图象与一个按钮关联起来，我们只需在 lParam 里存放对应的图象句柄，然后把 BM_SETIMAGE 消息发送出去就可以了，如下所示：

```
...
// assigning an icon to the button
SendMessage (hwndt, BM_SETIMAGE, (WPARAM) IMAGE_ICON,
   (LPARAM) (HBITMAP) hicon);
...
```

我们几乎可以肯定地说，所有类消息都需要在类进程向自己的父/物主窗口发送了相应的通知代码以后才能使用。BUTTON 类的通知代码是用一个 BN_前缀进行标识的，利用它们可以对用户在某些类进程里采取的操作进行判断。由于针对按钮能够进行的操作是有限的（只有按下或者释放），所以通知代码和消息也反应了用户对 BUTTON 类对象能够进行的那些有限的交互操作。

就按钮的应用来说，我们要注意到一个有趣的趋势，那就是它在对话框环境以外的应用场合变得越来越多。在 Win32 里，我们只需要进行简单的处理，就可以用一幅彩色图象取代以前用于标识按钮的正文串，这样就使按钮变得更富有表达力。关于这方面的问题，我们将在下面这一部分进行详细的介绍。

1. 图形化按钮

9.3 在第六章“菜单的运用”里，大家知道了如何在物主窗口里描绘一个菜单。要达到这种的，我们需要在设置了 MF_OWNERDRAW 风格以后对 WM_DRAWITEM 消息进行跟踪。相同的操作亦可适用于按钮，唯一的区别在于它们分配物主绘图行为的方式不尽相同。

由于我们准备处理的是一个窗口（而非菜单），所以在建立好窗口的过程中分配 BS_OWNERDRAW 风格显得更简单一些（而不是在这以后分配）。设置了这个风格标志以后，就可以直接在应用程序里描绘按钮；随后，我们需要发出 WM_MEASUREITEM 消息，接着发出一条 WM_DRAWITEM 消息。

利用 WM_MEASUREITEM 消息，应用程序可以通知控件关于它的尺寸的信息；而 WM_DRAWITEM 消息的作用则是实际执行所有的输出操作。假如窗口属于 BUTTON 类，并且设置了 BS_OWNERDRAW 标志，那么就只会接收 WM_DRAWITEM 消息，这是由于按钮的尺寸在建立的时候就已经定义好了。因此，接收到 WM_DRAWITEM 消息以后，程序需要实现的逻辑与我们在第六章“菜单的运用”里看到的针对菜单的逻辑是完全一致的。

在本书附带 CD 的 Listing 9.4 里，大家可以找到一个名为 ODBUTTON 的示范程序，该程序的显示结果如图 9-8 所示。在这个简单的窗口里，只显示了两个按钮，每个按钮都包含了一个图标。这个程序最有趣的地方在于按钮按下时出现的三维效果，这是由于物主窗口的机会造成的。

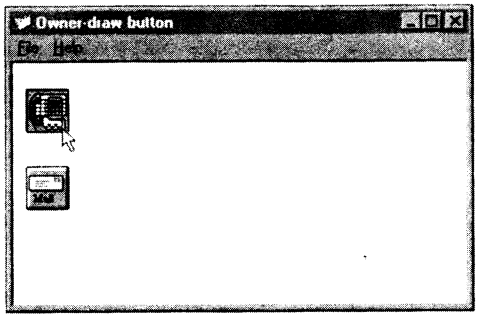


图 9-8　在 Win32 里建立三维图形按钮的一个例子

大家在图 9-8 里看到的两个按钮分别是用 CT_PHONE 和 CT_EMAIL 这两个 ID 定义的。它们的行为几乎完全一致，只是用于装饰它们的图象有所区别。在 WM_DRAWITEM 代码块里，我们首先取得了与 DRAWITEMSTRUCT 数据结构有关的信息，这是通过对 lParam 进行计算得到的。WM_DRAWITEM 内的程序逻辑是以对 wParam 的计算为基础的。例如，我们可以得到在那时用于指定待画对象的一个标识符。正如大家在针对菜单的学习过程中已经知道的那样，WM_DRAWITEM 消息可以派遣到窗口内存在的任何一个物主绘图对象里。针对 ID 的切换（switch）代码块允许我们根据需要处理的对象，从而令其采取不同的行为方式。

ODBUTTON 程序内的两个按钮在行为上是一致的，所以我们只用一段代码就实现了对它们两者的控制：

```
...
```

```
switch (wParam)
{
    case CT_PHONE:
    case CT_EMAIL:
        …;

    case CT_xxx:
        …;
}
…
```

三维效果是通过两个几乎完全一致的图标实现的——这两个图标使用的图象在垂直轴和水平轴上稍微有些移位。随后，绘图过程最后还要对灰色和白色加以适当的运用。灰色用于造成阴影效果，而白色则用于造成光线的区别。ODBUTTON 使用的四个图标列表图 9-9 内。第一幅图象用于未被按下的按钮，第二幅图象则用于按下以后的按钮。

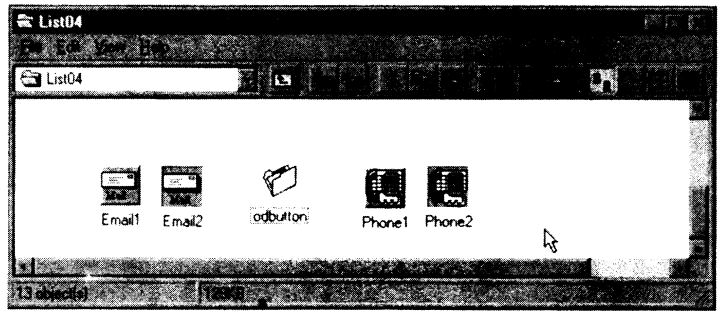


图 9-9　图中显示的前两个和后两个图标在 ODBUTTON 程序里用于建立三维按钮。中间那个按钮代表应用程序

三维效果可以采用不同的方式实现。代表电话机的那个图标为我们展示了微软公司的风格：底部和右侧边框最初是用深灰色描绘的（对应的 RGB 组合为：192，192，192），而其他边框则是白色的。假如按下显示电话机的那个图标，其中的图象会向下和向右移动了两个象素的距离（原来的灰色轮廓消失了，但又在左侧边框和顶部边框显示出来了）。

电子邮件图标则代表了 Borland 公司建议的图形化按钮风格，该公司建议在自己的所有软件产品中都使用这种风格。无论按钮按下与否，图形化对象（在这里是一个信封）都是在一个固定位置处显示出来的。但是按下按钮后，正文 Mail 会向下和向右进行移位，同时阴影边框也会发生变化。无论采用哪个公司建议的风格，按下按钮前后的视觉效果都是开发者对绘图和颜色进行运用的结果。

对应的 BUTTON 类建立好以后，程序需要立即调用 LoadIcon（）函数，从而载入这四幅图象。一旦载入到内存以后，应用程序就会把它们存储到数据段内，并且分别定义名为 icon1 和 icon2 的两种属性。

```
...
// load the icon
hicon1 = LoadIcon (hInstance, " phone1");
hicon2 = LoadIcon (hInstance, " phone2");

// create properties
SetProp (hwndPh, " icon1", hicon1);
SetProp (hwndPh, " icon2", hicon2);
...
```

这种语句序列随后将重复执行一遍——针对与“电子邮件”(ID 为 CT_EMAIL) 按钮有关的那两个图标。需要对一个按钮的内容进行描绘的时候，应用程序就会接收 WM_DRAWITEM 消息，该消息指定了 ODA_DRAWENTIRE (全部重画) 行动。在这种情况下，释放的按钮图标就会取回 (定义成属性 icon 1 的那个)，然后通过调用 DrawIcon () 函数在屏幕上显示出来。在实现最后一项操作之前，要先描绘一个带有圆角的矩形框，用它围绕图标的显示区域。

```
case CT_PHONE:
case CT_EMAIL:
{
    if ( (lpdis -> itemAction & ODA_DRAWENTIRE )
        || (lpdis -> itemAction & ODA_SELECT )
        && (! (lpdis -> itemState & ODS_SELECTED)))
     {
        HICON hicon;

        // retrieving the icon handle
        hicon = GetProp (lpdis -> hwndItem, " icon1");
        RoundRect (lpdis -> hDC,
                   lpdis -> rcItem.left,
                   lpdis -> rcItem.top,
                   lpdis -> rcItem.right,
                   lpdis -> rcItem.bottom,
                   2, 2);

        // draw the icon
        DrawIcon (lpdis -> hDC,
                  lpdis -> rcItem.left + OFFSET / 2,
                  lpdis -> rcItem.top + OFFSET / 2,
```

```
                    hicon);
        }
    ...
    }
    break;
```

对于代表未按下状态的图标来说，对它的描绘要在该按钮被选中之前完成，接下来的重画步骤是由选定状态决定的。相反，对于显示按钮按下状态的第二个图标来说，它要在用户按下对应按钮或者选择下一次重画步骤的时候显示出来。

```
case CT_PHONE:
case CT_EMAIL:
{
    ...
    if ( (lpdis -> itemAction & ODA_SELECT ) &&
            (lpdis -> itemState & ODS_SELECTED) )
    {
        HICON hicon;

        // retriving the icon
        hicon = GetProp (lpdis -> hwndItem, " icon1");

        RoundRect (lpdis -> hDC,
                   lpdis -> rcItem.left,
                   lpdis -> rcItem.top,
                   lpdis -> rcItem.right,
                   lpdis -> rcItem.bottom,
                   2, 2);

        // draw the icon
        DrawIcon (lpdis -> hDC,
                  lpdis -> rcItem.left + OFFSET / 2,
                  lpdis -> rcItem.top + OFFSET / 2,
                  hicon);
    }
}
break;
```

另外一种方案是把图标当作属性存储起来，这时需要用到每个窗口内可用的附加字节，可

资源文件里多次载入图象。假如采用后面那种办法（从资源文件载入），虽然可以省下一些宝贵的系统内存，但这是以降低应用程序性能作为代价的。

显然，使用两个不同的图标来代表按钮的两种状态是一种可行的办法。然而，在某些情况下，比如对于 Visual C++的工具栏来说，除了传统的按下和松开状态以外，按钮还可以有第三种状态——屏蔽按钮。在这种情况下，不会产生直接与物主绘图机制有关的任何问题。其中的原因很简单，这时没有任何物理性操作能够实际地发生在这种按钮上面。是否激活一个按钮只能通过应用程序自己的逻辑来决定，这种逻辑是通过调用 EnableWindow () 函数实现的。屏蔽一个窗口并不会影响到相应的按钮，显示新状态的任务由应用程序来完成。为了强制性对按钮内容进行重画，我们必须用 InvalidateRect () 对显示表面屏蔽，同时，假如希望马上更新按钮的外观，还需要调用 UpdateWindow ()。对一个按钮进行屏蔽以后，会生成一条 WM_DRAWITEM 消息，这条消息会直接进入父/物主窗口的窗口进程。

在第一种条件下，显示出来将是原始的图标。我们这时不应再显示标准的图标，而应显示它的一个新“版本”，用于强调该图标已被屏蔽的事实。为了达到这个目的，我们可以调用 IsWindowEnabled () 函数，用它对按钮状态进行检查。假如该函数返回一个 TRUE 值，就必须显示原始的图标；假如返回的 FALSE，就必须显示出屏蔽图标。为了获得这些功能，用几行代码即可实现，大家可以自己动手试一试。

正如大家在第 10 章“Windows 95 通用控件”里要学到的那样，在 Windows 95 里，Win32 提供了多种用于按钮建立的工具。这些工具函数能在一个工具栏内建立按钮，或者在屏幕上其他任何一个位置建立按钮。

2. 其他类型的按钮

在 Win32 里，存在着传统按钮的多个变种。根据核选框以及单选按钮的不同配置，我们的选择可以是多种多样的。很少出现的一种情况是把单选按钮和核选框放置于叠置式或者弹出式窗口的客户区内。它们的传统容器通常都是一个对话框。

通过前一章的学习，大家已经看到了装备这些控件的对话框。对于核选框和单选按钮来说，它们的旁边通常都显示了一个对应的正文标签，这个标签指出了自己的含义。通常，无论单选按钮还是核选框都要通过一个框架分组到对应的区域内，这些区域在框架里需要进行良好的分隔。同时，这种框架也需要用一个正文标签进行说明。我们把这种类型的框架称为“分组框”，这是 BUTTON 窗口类的另外一个变种。在分组框里，我们不可能与窗口进行任何形式的交互作用，因为它只是扮演了一种装饰性的角色，不具备任何功能性因素。

作为一条通用的规则，对于按钮来说，无论它们的形状如何，都具有两种可能的状态：要么是按下的，要么是释放的。核选框可以设计成具备第三种状态，我们将其称为“不确定”状态。在这种情况下，核选框会用灰色显示，而不是显示传统的选定“x”符号，或者什么都不显示（置空）。例如，三状态核选框的一个典型应用可以在 MS Excel 里找到。在用户定义希望分配给多个单元格的属性时，假如选择组内只有少数几个单元格具有由那个核选框控制的属性，就会建立一种“不确定”状态。

3. BUTTON 类使用的宏

如表 9-6 所示，其中列出了 BUTTON 类使用的宏，这些宏可以在 WINDOWSX.H 头文件里找到。假如以黑体字显示，就表明对 SendMessage () 函数返回值进行的计算操作是构成这些宏中大多数的基础。

表 9-6 BUTTON 类的宏函数

宏	宏
Button_Enable (hwndCtl, fEnable)	Button_SetCheck (hwndCtl, check)
Button_GetText (hwndCtrl, lpch, cchMax)	Button_GetState (hwndCtl)
Button_GetTextLength (hwndCtl)	Button_SetState (hwndCtl, state)
Button_SetText (hwndCtl, lpsz)	Button_SetStyle (hwndCtl, style, fRedraw)
Button_GetCheck (hwndCtl)	

9.3.2 LISTBOX 类

假如开发者需要处理属于同一种类的项目，比如文件名称、希尔顿饭店在全世界的分布位置以及股票报价等等，对这些项目进行表示的最佳途径就是使用一个列表框窗口。LISTBOX 类窗口在屏幕上只占据了很少的一部分空间，其中只列出了所有正文串的一小部分。正因为如此，列表框几乎肯定是和一个垂直滚动条关联在一起的。假如列表的项目数目超出了能在窗口中显示的项目数，就需要用到这种垂直滚动条。LISTBOX 类的使用相当灵活，功能也很强，这主要是由于它提供了大量风格和消息的缘故。

对于这一类的窗口来说，它的一个优点是可以存储大量信息。Win16 那碍手碍脚的 64K 限制在 Win32 里已经完全消失了。我们现在可以在列表框内存储比往常多得多的信息，而且还能获得优化的性能。

现在让我们讨论一下 LISTBOX 类使用的风格。之所以设计列表框，一般都是为了方便用户进行某种简单的选择；假如设置了特殊的 LBS_标志，还能设计出扩展选择和多重选择列表框。假如开发者没有设置 LBS_MULTIPLESEL 或者 LBS_EXTENDEDSEL 风格标志，列表框就会成为一个最简单的选择列表框。对窗口行为产生的这种影响也会反应在用户的操作上面。假如选定了一个项目，鼠标指针热点下方的正文串就会以反转色显示：黑色背景上的白色正文。

选定一个项目需要用户在该项目上方按下鼠标左键（更准确地说，是按下缺省的鼠标按钮），用鼠标右键进行选择是不允许的。除此以外，以后在已选中的项目上面按下鼠标左键不会产生任何附加的效果。为了撤消对一个项目的选定，必须在列表框内选择另外一个项目。

在多重选择列表框（LBS_MULTIPLESEL）里，刚才讨论的行为就不成立了。在这种特殊的情况下，用户一次可以选择多个项目。假如在一个已选中的项目上方单击鼠标左键，就会自动撤消对它的选定。这种特性也对父/物主窗口和列表框之间的交互作用产生了影响。在标准列表框的情况下，返回的值只有一个。而对于一个 LBS_MULTIPLESEL 列表框来说，却有必要对大量选中的项目进行控制，因此一般都需要开发者设计一个 while 循环，用它对每个项目进行重复。

多重选择列表框与扩展选择列表框（LBS_EXTENDEDSEL）之间的唯一区别在于它们对鼠标的控制不一样。LBS_MULTIPLESEL 列表框以前介绍的所有规则对于扩展列表框都是成立的，唯一的区别在于现在能够在第一个项目上方按下鼠标左键，然后向下垂直拖动鼠标，直到选定了最后一个项目为止。这样一来，我们就选定了第一个到最后一个项目之间的所有项目。

表 9-7　用于 LISTBOX 类的 LBS_风格

风格	值	说明
LBS_NOTIFY	0x0001L	通知父/物主窗口在列表框内发生的事件
LBS_SORT	0x0002L	正文串开始插入列表框内的时候，按照 ANSI 代码顺序对其进行排序
LBS_NOREDRAW	0x0004L	屏蔽列表框内的所有重画活动；这样可以有效地防止发生字符闪烁现象
LBS_MULTIPLESEL	0x0008L	建立带有多重选择的列表框
LBS_OWNERDRAWFIXED	0x0010L	建立带有固定高亮度显示项目的列表框，这些项目是由应用程序选择的
LBS_OWNERDRAWVARIABLE	0x0020L	建立带有固定高亮度显示项目的列表框，这些项目是由应用程序选择的
LBS_HASSTRINGS	0x0040L	这个标志只能针对需要显示正文串的物主绘图列表框设置
LBS_USETABSTOPS	0x0080L	这个标志允许列表框按照特殊形式的制表位
LBS_NOINTEGRALHEIGHT	0x0100L	允许建立和控制在垂直轴上能使用任何尺寸一个的列表框，不局限于必须使用系统字体尺寸的整数倍
LBS_MULTICOLUMN	0x0200L	建立跨越多列的一个列表框
LBS_WANTKEYBOARDINPUT	0x0400L	允许列表框的父/物主窗口接收消息 WM_和 VKEYTOITEM 和 WM_CHARTOITEM
LBS_EXTENDEDSEL	0x0800L	建立和扩展列表框的选定
LBS_DISABLENOSCROLL	0x1000L	总是显示一个垂直滚动条
LBS_NODATA	0x2000L	建立一个列表框，同时不为它的数据分配显示空间
LBS_STANDARD	(LBS_NOTIFY | LBS_SORT | WS_VSCROLL | WS_BORDER)	用标准属性建立一个列表框

列表框还有另外一个特殊的地方。它们的高度一般都是字体高度的整数倍，这种字体正用于显示列表框内容的那一种。这个规则有时也会导致出现不舒服的视觉效果，特别是在表框正被其他窗口围绕，并且这个窗口有一个重定义尺寸边框的时候。此时，列表框不能续在 Y 轴上的任何度量，这样就在高度上产生了不连续的跳跃——为了保证正好是系统字体高度的整体倍。为了避免这个问题，我们可以采用 LBS_NOINTEGRALHEIGHT。设置了这个标志以后，列表框就可以采用任何一个垂直高度。有些时候，这种处理会导致字符串只有部分显示出来。

建立一个列表框最好和最简单的方法就是使用 LBS_STANDARD 风格，这种风格集成了下面这些 LBS_和 WS_风格：

LBS_STANDARD（LBS_NOTIFY | LBS_SORT | WS_VSCROLL | WS_BORDER）

尽管用户只能对一个列表框采取有限的几种操作（单一选择或者多重选择），开发者仍然能够选用这一类专用的许多消息，所有这些消息都总结于表 9-8 内。

表 9-8　用于 LISTBOX 类的 LB_消息

LB_ADDSTRING	0x0180	在列表框内插入一个正文串，把它追加到列表后面，或者按字母顺序插入其他项中（前提是列表框同时使用了 LBS_SORT 风格）
LB_INSERTSTRING	0x0181	在列表内的指定位置插入一个正文串
LB_DELETESTRING	0x0182	删除一个正文串
LB_SELITEMRANGEEX	0x0183	在多重选择列表框内选择多个连贯的项目
LB_RESETCONTENT	0x0184	腾空列表框
LB_SETSEL	0x0185	把列表框的选择方式设置成多重选择
LB_SETCURSEL	0x0186	把列表框的选择方式设置成单一选择（一次只能选一个）

（续）

LB_GETSEL	0x0187	返回一个多重选择列表框内的当前选择
LB_GETCURSEL	0x0188	返回一个单一选择列表框内的当前选择
LB_GETTEXT	0x0189	返回当前选择的正文
LB_GETTEXTLEN	0x018A	返回当前选择的正文的长度
LB_GETCOUNT	0x018B	返回当前在列表框内的项目个数
LB_SELECTSTRING	0x018C	用一个正文串与消息内传送的正文串以及列表框正文串内的匹配字符数进行比较，从而选择某个项目。如有必要，会提供垂直滚动条，从而显示出查找到的项目
LB_DIR	0x018D	用一个目录内的文件名填写一个列表框
LB_GETTOPINDEX	0x018E	返回列表框内最顶部项目的索引
LB_FINDSTRING	0x018F	查找一个正文串
LB_GETSELCOUNT	0x0190	返回多重选择或者扩展选择列表框内的已选中项目的个数
LB_GETSELITEMS	0x0191	在一个多重选择或者扩展选择列表框内，定义可选项目的最大数目
LB_SETTABSTOPS	0x0192	定义由列表框识别的制表位的数目和位置
LB_GETHORIZONTALEXTENT	0x0193	返回用 WS_HSCROLL 建立的一个列表框的水平滚动步长（用象素表示）
LB_SETHORIZONTALEXTENT	0x0194	设置用 WS_HSCROLL 建立的一个列表框的水平滚动步长（用象素表示）
LB_SETCOLUMNWIDTH	0x0195	设置一个多栏列表框（LBS_MULTICOLUMN）的栏宽
LB_ADDFILE	0x0196	为包含目录列表的一个列表框增加一个文件名
LB_SETTOPINDEX	0x0197	设定准备在列表框顶部显示的项目
LB_GETITEMRECT	0x0198	返回围绕于列表框项目的矩形
LB_GETITEMDATA	0x0199	返回列表框内的项目数据
LB_SETITEMDATA	0x019A	设置列表框项目的数据
LB_SELITEMRANGE	0x019B	允许选择或者撤销选择多重选择列表框内的项目范围
LB_SETANCHORINDEX	0x019C	把一个项目标识成多重选择中的第一个项目
LB_GETANCHORINDEX	0x019D	返回多重选择中的第一个项目
LB_SETCARETINDEX	0x019E	围绕列表框内某个给定项目描绘选定符号
LB_GETCARETINDEX	0x019F	返回由选定符号围绕着的那个项目
LB_SETITEMHEIGHT	0x01A0	设置一个项目以高亮度显示
LB_GETITEMHEIGHT	0x01A1	返回亮度显示的项目
LB_FINDSTRINGEXACT	0x01A2	根据正确的匹配方案在列表框内查找一个串
LB_SETLOCALE	0x01A5	设置列表框的显示地点
LB_GETLOCALE	0x01A6	返回列表框的显示地点
LB_SETCOUNT	0x01A7	设置用 LBS_NODATA 创建的列表框内项目数
LB_INITSTORAGE	0x01A8	分配用于存放项目的内存
LB_ITEMFROMPOINT	0x01A9	返回与给定点最接近的项目

之所以有如此众多的消息，主要是由于标准的列表框的不同行为造成的。这个预定义窗口类既可以是一个单一选择列表框，也可以是一个多重选择列表框，甚至还可以是一个扩展选择列表框。不同的外观也影响到了通知代码的不同形式，如表 9-9 所示。

表 9-9　LISTBOX 类使用的 LBN_通知代码

LBN_ERRSPACE	(-2)	列表框的内存管理出现了问题
LBN_SELCHANGE	1	列表框内的选定已经改变了
LBN_DBLCLK	2	用户双击了鼠标左键
LBN_SELCANCEL	3	列表框内某一项的选定已经撤销了（列表框必须包含风格标志 LBS_NOTIFY）
LBN_SETFOCUS	4	列表框接收到了视觉焦点
LBNA_KILLFOCUS	5	列表框失去了视觉焦点

假如用户在列表框内选定了一个项目，这时会发生什么情况呢？此时，在属于 LISTBOX 类的窗口进程里，会接收到一条 WM_LBUTTONDOWN 消息，表示鼠标左键已被按下。同时，鼠标指针热点下方的正文串会用自己的反转色显示出来。随后，窗口进程会利用 WM_COMMAND 消息向自己父/物主窗口通报这一情况。这条 WM_COMMAND 消息内只包含了 LBN_SELCHANGE 通知代码。因此，LISTBOX 只向上通知了选定发生改变的情况，没有说明选定项目的索引编号。并且更重要的一点是，它没有通知以反转色显示的正文串。假如物主窗口需要获得这方面的信息（几乎肯定是需要的），它就必须发送出一对消息。首先，物主窗口必须使用 LB_GETCURSEL 消息，从而获得选中项目的索引，此时应该按照下面这种语法使用：

```
...
wPos = (WORD) SendMessage (hwndList, LB_GETCURSEL, 0, 0L)
...
```

很显然，上面列举的是简单选择列表框的情况。在 WINDOWSX.H 里，甚至还包含了一个等价的宏函数，如下所示：

```
ListBox_GetCurSel (hwndCtl);
```

返回值与项目的索引编号对应。这个编号通常都是从零开始的，如图 9-10 所示。

显然，只有滚动条向上滚动到自己最顶部的位置时，索引 0 才与列表框内的第一个项目对应。这正是在窗口内插入了某些字符串以后出现的情形（前提是根据标准进程，把新项目增添到列表的末尾）。

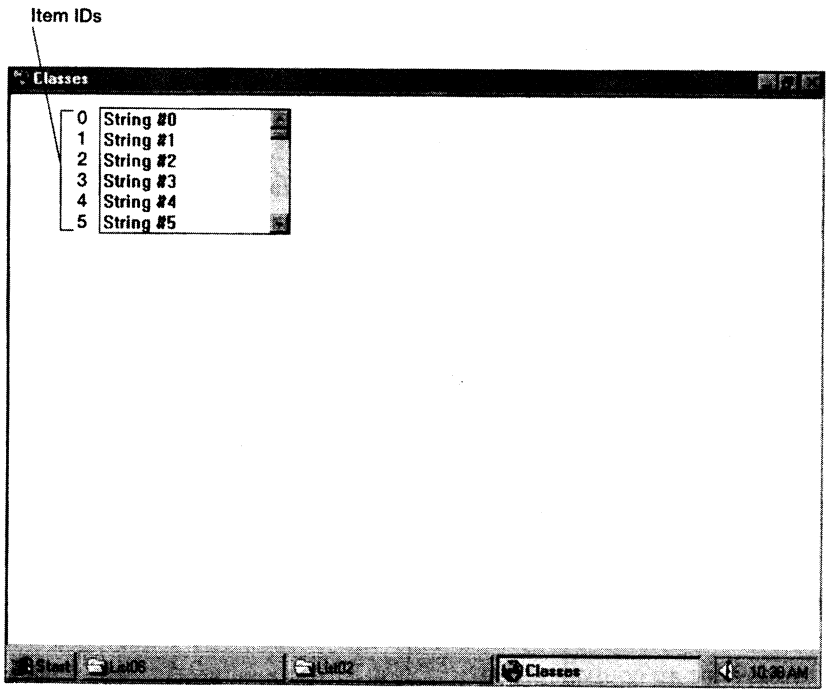


图 9-10　分配给列表框项目的索引编号

为了返回选中项目内包含的正文，父/物主窗口必须随之发送一条 LB_GETTEXT 消息，并在其中传递刚才由 LB_GETCURSEL 取回的值。

```
...
SendMessage (hwndList, LB_GETTEXT, wPos, (LPARAM) (LPSTR) szText);
...
```

或者依靠下面这个宏：

```
ListBox_GetText (hwndCtl, index, lpszBuffer);
```

对于其他常见的操作来说，比如插入正文串或者接收选中项目数目（甚至删除它们）等等，大家可以参考一下本书附带 CD 的 Listing 9.6。

1. 水平滚动

滚动对于用户来说有时是比较讨厌的，因为这意味着他们也许不能同时看到自己感兴趣的所有信息。水平滚动的使用应该尽可能地避免，因为它显著地降低了用户把自己的注意力集中在真正感兴趣的项目上的能力。然而，列表框是这方面应用的一个典型例子，某些情况下甚至没有其他选择，只能选择水平滚动条。

为了建立水平滚动条，第一个步骤是在建立窗口的时候包含 WS_HSCROLL 标志。然而，尽管这样做是正确无误的，但是滚动条也不会显示出来。实际上，为了真正得到水平滚动条，为了真正向用户显示出这种工具，首先需要的就是设置滚动的范围。这儿准备使用的消息是 LB_SETHORIZONTALEXTENT：

LB_SETHORIZONTALEXTENT	0x0194
wParam	滚动宽度
lParam	未使用

2. 改变列表框的结构

在我们讨论列表框的时候，我建立了一个简单的示范程序，如图 9-11 所示。该程序名为 LISTBOX，大家可以在本书附带 CD 的 Listing 9.11 内找到它。LISTBOX 示范程序主要由一个简单的测试窗口以及一些按钮组成。假如按下某个按钮，就可以对列表框执行某种操作；这个操作是通过类消息或者一个等价宏的途径来实现的。

这个程序运行以后，必须采取的第一项操作是建立列表框。为了实现这一目标，我们可以按下列这三个按钮中的一个：

▶Single（单一选择）

▶Multiple（多重选择）

▶Extended（扩展选择）

结果是一个简单的列表框、一个多重选择列表框或者一个扩展选择列表框。在后面那两种情况下，我们需要分别对 LBS_MULTIPLESEL 或者 LBS_EXTENEDSEL 标志进行设置。

对于可以根据用户的选择改变自己行为的一个列表框来说，第一种考虑涉及到 LBS_MULTIPLESEL 或者 LBS_EXTENEDSEL 标志的动态设置或者清除，这种设置或者清除是以用户的操作为基础的。对于所有 WS_风格来说，这种机制的运行都很良好。请大家参考第 16 章“Win95 外壳的开发”，其中讲述了关于用户界面设计的问题。在列表框的情况下，我们不可能把一个多重选择列表框的行为改变成一个简单的单一选择列表框。其中的原因在于微软公司实现这类窗口进程的途径比较特殊。

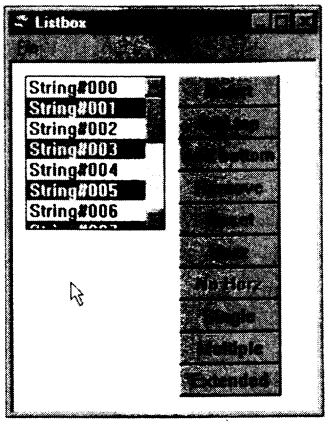


图 9-11　LISTBOX 示范程序显示了属于 LISTBOX 类的窗口的某些结构化特性

LBS_MULTIPLESEL 或者 LBS_EXTENEDSEL 标志只有在建立好列表框以后才会有效。或许，在列表框类窗口进程里，执行选择操作的时候（典型地，在接收到一条 WM_LBUTTONDOWN 消息的时候），某些值为针对窗口的附加字节而核选。试图通过 SetWindowLong（）函数对风格进行分配并不是改变附加字节信息的正确方案。因此，尽管接收到了风格标志，窗口也不会按照我们的希望进行响应。

我在这儿不准备对 LISTBOX 窗口的附加字节再做更深层次的讨论了，因为这样会导致麻烦。由于未作具体的文档化，微软在任何时候都有可能改变这个区域的含义，因此依赖这一区域存放的信息并不是明智之举。另外，用户每次按下这三个按钮中的一个时，都需要建立一个新窗口。与这个操作有关的是把前一个列表框的内容转换成新的内容。这样也许会在视觉上造成一些闪烁。但是，要解决这个问题也很简单，只需要按照下面这种显示逻辑设计即可：先删除用于新列表框的 WS_VISIBLE 标志，然后再调用 ShowWindow（）。

为了对窗口按钮的按下进行处理，我们应该尽可能地使用 WINDOWSX.H 里 LISTBOX 类的宏函数。和使用 SendMessage（）得到的结果比较起来，这样得到的结果要精简得多，有时甚至还具有更强的可读性。

3. 列表框内滚动条的管理

对列表框进行处理的时候，必须考虑到关于垂直滚动条（有时甚至是水平滚动条）一个相当不容易想到的问题。和大家猜测的或许不一样，我们决不可能找到一个属于 SCROLL 类的窗口。取而代之的是，我们倒常常能看到一系列图形化对象的集合（如果大家还有怀疑，可以试着用 SPY 或者 WINSPY 程序对此进行研究）。这就意味着列表框实际上是一个单一的对象，这和单单通过外观推测出来的结论是不同的。奇怪的地方在于这儿用于建立两类滚动条的方法：开发者必须使用 WS_xSCROLL，就好象正在处理的是一个真正的窗口一样。在 LISTBOX 类的窗口进程里接收到一条 WM_CREATE 消息以后，应用程序就会检查这两个标志，并且显示出两个图形化对象。

4. 扩展选择列表框的建立

除了在调用 CreateWindowEx（）函数时设置一个 LBS_EXTENEDSEL 标志以外，建立一个扩展选择列表框并不意味着还有其他什么特别的地方。然而，与这个种类的列表框有关，

倒确实有一种有趣的行为是需要强调的。这种行为涉及到通过鼠标进行选择，选择的范围并不局限于窗口的特殊显示区域——也就是说，选择范围可以延展到窗口的底部或者顶部，具体向上还是向下延展要取决于鼠标的移动方向。很显然，只要用户按下了鼠标左键，并且一直按住不放，列表框就会不断地捕获鼠标的这种行动，这样完全有可能实现我们刚才谈及的选择行为。从窗口物理边界退出被理解成一系列的 WM_MOUSEMOVE 消息，而且用负值表示这种鼠标的运动已经超出了窗口的限制。无论在哪种情况下，对 WM_MOUSEMOVE 消息进行控制的代码都会模拟在垂直滚动条上的单击操作，而且会按照恰当的方向进行，直至最后选中了窗口内可见的每一个项目为止。

5. 物主绘图列表框的建立

设计列表框的唯一目的就是用它容纳正文串。LB_ ADDSTRING 和 LB_INSERTSTRING 消息是开发者用于插入新项目的唯一工具。然而，假如指定了 LBS_OWNERDRAWFIXED 或者 LBS_OWNERDRAWVARIABLE 标志，列表框就可以把所有绘图操作派遣给父/物主窗口来完成。在这个时候，只能处理正文的限制就不复存在了，因为任何类型的对象都可以用一个项目的形式出现。

这两种风格产生的唯一区别与项目的垂直尺寸有关。假如使用的是 LBS_OWNERDRAWFIXED，列表框内的所有项目都会用相同的高度显示，具体的高度是通过 WM_ MEASUREITEM 消息来设置的。在另外一方面，假如设置的是 LBS_OWNERDRAWVARIABLE 标志，就会向列表框内的每个项目发送 WM_MEASUREITEM 消息，因为它们的垂直高度都是可变的。

在一个物主绘图列表框内进行描绘的时候，由 WM_DRAWITEM 的 lParam 指定的 DRAWITEMSTRUCT 数据结构是一种关键性元素。正如我们在讨论菜单和按钮时已经知道的那样，DRAWITEMSTRUCT 内包含了用于对无效矩形、它的项目索引、窗口句柄以及设备现场的句柄进行说明的所有信息。其中，设备现场的句柄是最重要的信息，所有输出操作都需要在它里执行。

EXTSEL 程序（在本书附带 CD 的 Listing 9.6 里可以找到）为我们提供了对扩展选择和物主绘图列表框进行研究的机会。如图 9-12 所示，列表框项目是由正文串以及前面的一个图标构成的。其中的每个图标都有两种状态，一种是平常的状态，一种是找到一本书时的样子（大家可以发现本人在绘图上的水平并不高明）。对于后面那种状态，会在图标内显示一支手的形状。

在 EXTSEL 程序里，有一个有趣的现象引起了我们的注意。假如用鼠标选择一个项目，就会根据鼠标指针的位置而产生不同的效果。假如只是简单地在项目的正文部分单击鼠标左键，程序就会象往常一样对待，强迫所有正文以反转色显示，如图 9-13 所示。假如在每个项目的左边那一部分（精确地说，是左边 40 个象素点以内）单击鼠标左键，正文同样会以反转色显示，同时图标的第二种状态会显示出来，如图 9-14 所示。

在这儿，需要强调的一个重点在于，用户在列表框内同一个项目的两个连续部分上面采取的相同操作（用鼠标单击）会生成两种不同的行为，其中任何一种行为都不是标准的。因此，应用程序必须对鼠标单击事件进行区分，判断是在正文部分上单击，还是在图标部分上单击。从而图形化地显示出整个项目。因此，在这个例子里，大家并不只是能体会用于建立物主绘图列表框的技术，而且也能知道如何根据鼠标热点的相关位置，从而采取不同的处理

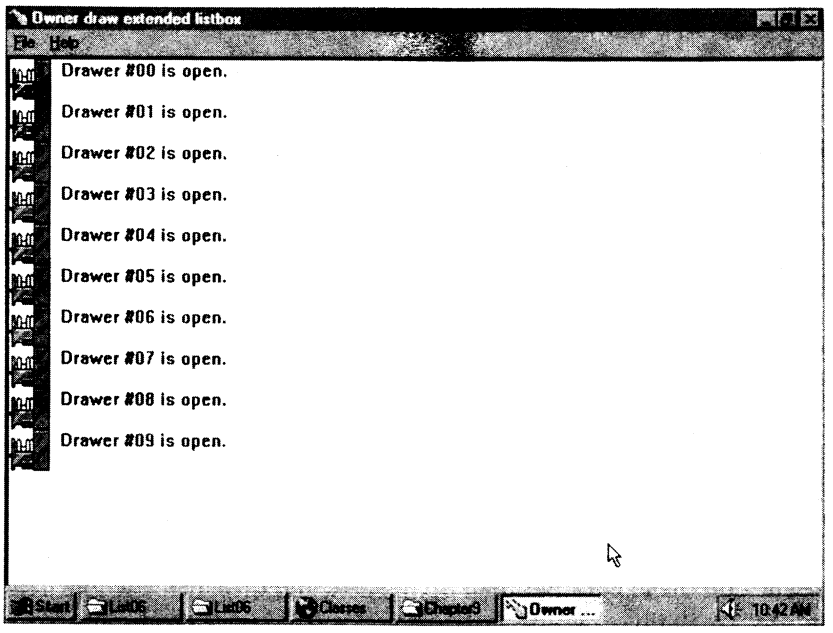


图 9-12　EXTSEL 应用程序所含列表框的输出是由程序控制的，其中的图标已与一个正文串组合起来了

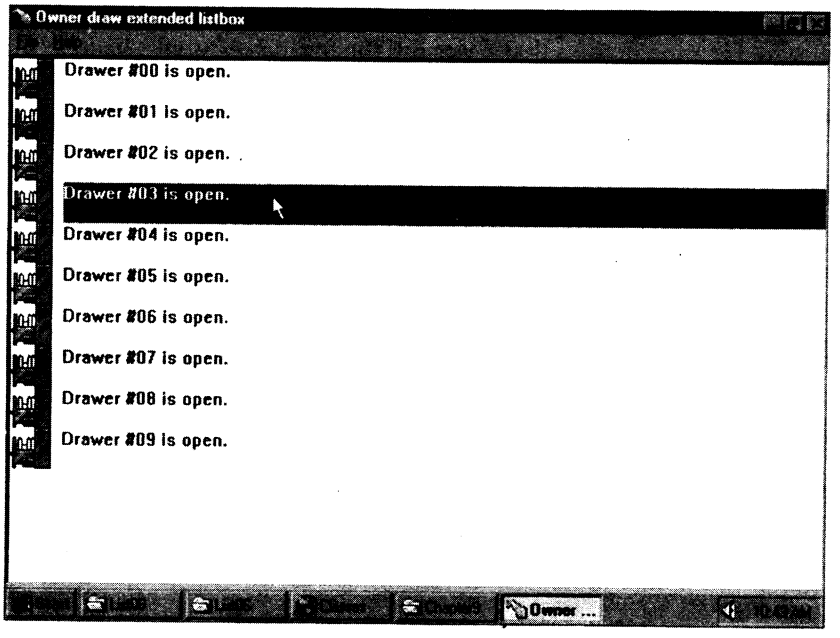


图 9-13　正文的选择是通过按鼠标左键实现的，选定的正文以反转色显示

方式。这样就对列表框的功能进行了延展。

现在让我们首先探讨一下这个例子的图形化功能。列表框建立起来以后，程序紧接着就会使用一个 for 循环，从而用一些正文串对这个列表框的内容进行填充。

```
// inserting MAXSTRING strings in the listbox
for (i = 0; i < MAXSTRING; i++)
 {
    wsprintf (szString, " Drawer #%02d is open.", i);
    sPos = ListBox_AddString (hwndList, szString);
    ListBox_SetItemData (hwndList, sPos, (LPARAM) 0);
}
```

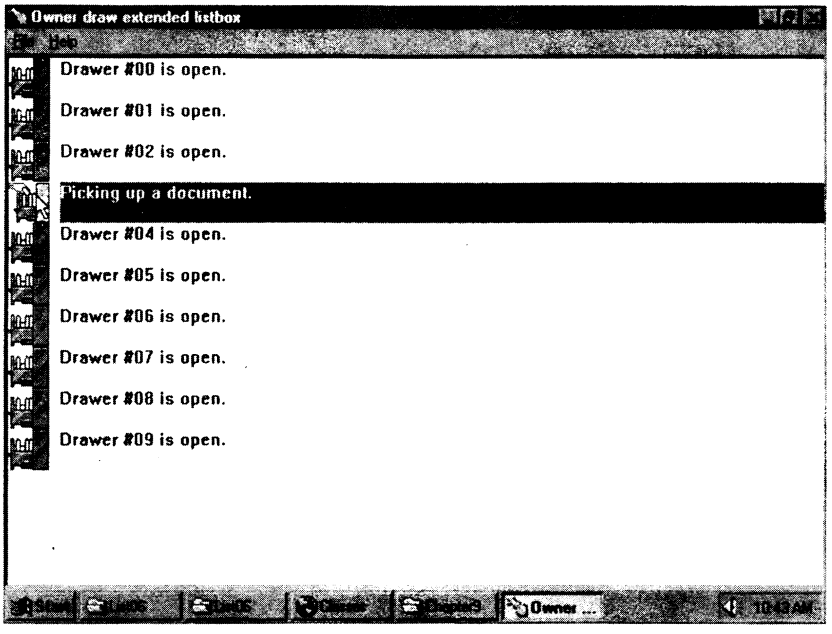


图 9-14　对图标的选择会改变图标的样子，并且以反转色显示相应的正文

即使在这种情况下，我们仍然使用了 LISTBOX 类的宏函数（大家可以参考本书附带 CD 的 Listing 9.5）。利用 ListBox_AddString ()，我们可以把一些字串插入列表框内。一旦正文的插入工作完成以后，就会把两个图标的句柄存储到每个项目的支持字节以内。什么是支持字节呢？从版本3.0开始，列表框的每个项目都有一个四字节宽的预约内存区域，应用程序可在其中存储自己觉得有必要的任何信息，只要它放得下。从概念上说，这些字节和我们在窗口中找到的附加字节非常类似。然而，它们之间存在的一些重要区别是值得我们注意的。

附加内存与窗口有关，而支持字节则是列表框内每个项目的一个组成部分。除此以外，列表框也为开发者提供了 GWL_USERDATA 区域。另外，附加字节是可选的，而支持字节则肯定是存在的，不能缺少（除非在建立列表框的时候指定了 LBS_NODATA 标志，这种情况很少见。对这个区域进行访问要经过 LB_GETITEMDATA 和 LB_SETITEMDATA 消息的许可。对于第二条消息来说，我们也可以使用对应的宏函数来达到相同的目的。大家可以在图 9-15 里看到支持字节的一种图形化解释。

我初次开发这个例子的时候是在 1989 年进行的，那时 32 位计算环境还似乎是遥不可及

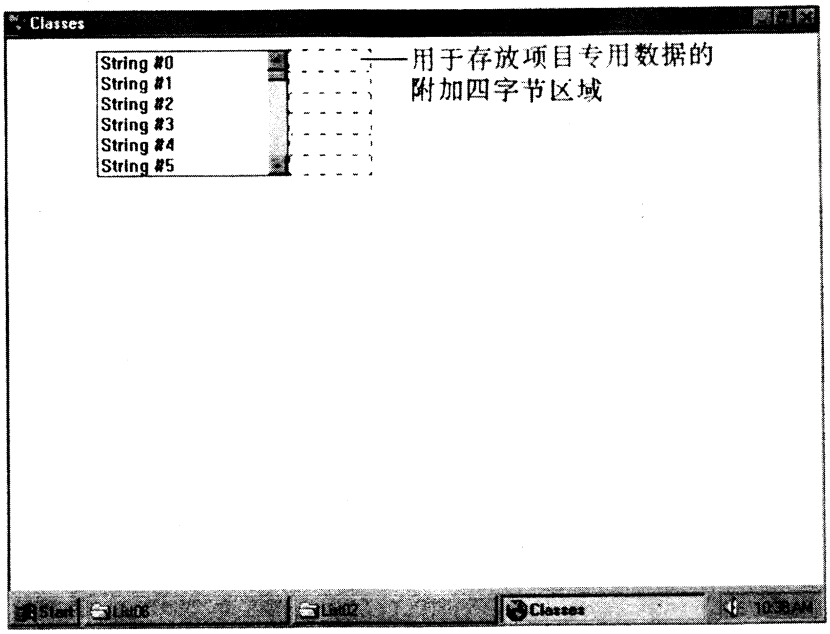


图 9-15　所有列表框肯定都有自己的支持字节

的。在那个时候，我决定在每个四字节项目区域里存放显示于项目正文左侧的两个图标的句柄。这种方案显得相当明智，这是由于那时的句柄有两个字节宽。随着 Win32 的降临，句柄的长度扩展到了四个字节，因此我似乎必须对应用程序的这一部分重新进行设计。但是让人惊奇的事情发生了，代码未经任何修改在 Win32 里仍然运行良好。出了什么奇迹吗？本来我是把两个四字节长的对象合并到一个总长才四字节的区域里的，但是真正的奇迹是程序居然工作正常！其中的原因说穿了其实很简单，因为一个图标句柄全部有价值的信息都存储于它的低字内。两个图标句柄被塞入一个四字节长的区域时，它们就丢失了自己的高字，但这并未影响到其中存放的真正有价值的信息。

为了采用一种更准确的方法，我们需要采取不同的算法，需要把两个图标句柄存储到另外一个内存区域或者其他类似的地方。EXTSEL 程序在主窗口预约的附加内存区域内分配了一些空间，用它们来存放两个图标句柄。这两个图标各自放置于偏移位置 0 和 sizeof (HICON)，也就是偏移位置 4 处。用正文串填写列表框的时候，EXTSEL 就会存储一个 0 值，用它指定两个图标的第一个应该在屏幕上显示出来。用户单击了这个图标以后，应用程序就会接收存储于项目存储区域内的值，对应地改变这个值，然后把修改后的值存储起来。这个值用于判断存储于主窗口附加内存区域内两个图标中的哪一个应该作为项目图标显示出来。

需要显示图标/正文串组合的时候，应用程序就会接收到一条名为 WM_DRAWITEM 的消息。对每个单独项目的输出进行设计的任务是由 DrawEntire () 函数完成的，这个函数是专门针对 EXTSEL 应用程序里的这种用途设计的。这个函数要执行的第一步操作是得到两个图标的句柄，然后显示出需要的那一个。随后，输出的效果会根据项目自己的状态得到改进。

```
...
// retrieving the inserted text
ListBox_GetText(lpdis -> hwndItem, lpdis -> itemID, szString);
// getting the icon value
iIcon = ListBox_GetItemData(lpdis -> hwndItem, lpdis -> itemID);
// showing the appropriate icon
DrawIcon(lpdis -> hDC, rc.left, rc.top,
        (HICON)GetWindowLong(PAPA(lpdis -> hwndItem), iIcon * sizeof
        (HICON)));
// showing the text
TextOut(lpdis -> hDC, rc.left + OFFSET, rc.top, szString, lstrlen(szString));
...
```

用户改变了选择时，我们可以留意一下此时在程序代码内部发生的一些有趣的变化。假如用户选择的是一个已被选中的项目，一条 WM_DRAWITEM 消息就会再次进入列表框的父/物主窗口的窗口进程。第一次到达的时候，这条消息内包含了用于把以前选择的项目重新设置成自己初始状态时的信息。第二次到达的时候，消息就会指定最新选定的项目。所以，这一部分代码将首先判断鼠标在屏幕上的位置。由于这种信息是用屏幕坐标表达的，所以，指定的屏幕点必须用列表框的度量单位转换成相应的坐标，如下所示：

```
...
// Get the mouse's position
GetCursorPos(&pt);
// Convert to listbox coordinates
ScreenToClient(lpdis -> hwndItem, &pt);
...
```

经过这种操作以后，鼠标的位置就会与列表框的水平轴进行比较，这是为了判断它是否位于包含了图标的那个区域上方。这种测试也要与重画操作结束时对项目状态进行的检查关联起来。假如 DRAWITEMSTRUCT 的 itemState 项内包含了 ODS_SELECTED 标志，WM_DRAWITEM 消息就会引用准备选择的那个项目。假如两种情况下都得到了 TRUE 的结果，就表明用户单击了图标竞争（项目左侧的那一部分），所以程序就会切换到相应的图标处理部分：

```
...
// is the mouse over the icon ?
if(pt.x < OFFSET && (lpdis -> itemState & ODS_SELECTED))
{
    int iIcon;
```

```
        char *szText[] = { "Drawer is open.", "Picking up a document."};
        RECT rc = lpdis -> rcItem;

        MessageBeep(0);
        // get the current icon reference
        iIcon = ListBox_GetItemData(lpdis -> hwndItem, lpdis -> itemID);

        // get the other icon
        iIcon = ! iIcon;
        // save the new icon reference
        ListBox_SetItemData(lpdis -> hwndItem, lpdis -> itemID, (LPARAM)
            iIcon);

        // display the icon on the screen
        DrawIconEx(lpdis -> hDC,
                   lpdis -> rcItem.left,
                   lpdis -> rcItem.top,
                   (HICON)GetWindowLong(hwnd, iIcon * sizeof(HICON)),
                   32, 32, 0, GetStockObject(WHITE_BRUSH),
                   DI_DEFAULTSIZE);

        // clean the background
        rc.left += OFFSET;
        FillRect(lpdis -> hDC, &rc, CreateSolidBrush(RGB(255, 255, 255)));
        // display the text
        DrawText(lpdis -> hDC, szText[iIcon], strlen(szText[iIcon]), &rc,
        DT_LEFT);
    }
...
```

这种行为很容易就可以扩展到更多的图形化对象上面，这些图形化对象可能会在一个项目内同时显示出来，也许还有可能在正文的不同部分出现。单一的 if 语句由此会变得更加重要，但是在实际的编程环节上，采用的逻辑并没有什么分别。

我相信 EXTSEL 示范程序采用的这种方案能够在一个“对用户友好”的应用程序里发挥它更大的作用。举个例子来说，假设列表框内的一份列表列出了近四年以来由 Ziff-Davis 出版社出版的所有图书的标题。为了实现这种设计，最简单的方案就是建立一个小型的字符串数据库，每个项目都对应一本图书的标题（也许是资源文件里的一个 STRINGTALBE——串表），然后依次提取出这些标题，从而实现列表框的显示。一种更有效的方法是实现一个物主绘图列表框，其中除了包含恒定不变的正文以外，还包含了一些很有说服力的图标。其中一

个图标可以是一张笑脸，它表示自己可以提供关于本书作者更详尽的一些信息。另外一个图标则可以提供与书籍有关的详细技术数据，其中包括排字、布局、艺术作品以及印刷等等方面的信息。

这种方法还能给我们更多的启迪，比如可以通过它在单独一个列表框内综合与同一主题有关的一系列信息。这样就给用户带来了方便，他们可以用简单和直观的方式（只需要鼠标单击）获取自己需要的数据。除此以外，假如列表框允许用户进行多重选择或者扩展选择，就能够同时对一组项目进行操作，这样就减少了用户们重复单一操作的次数。这儿提供的所有方便都是开发者乃至用户十分欢迎的。

6. LISTBOX 类使用的宏

在表 9-10 里，大家可以看到与 LISTBOX 类窗口进行交互作用的所有宏函数的名称。这个列表框仅限于一些常用宏函数，事实上，只要有必要，开发者可以在这儿使用任何类型的宏定义。

表 9-10 LISTBOX 类的宏函数

宏	宏
ListBox_Enable (hwndCtrl, fEnable)	ListBox_SelectString (hwndCtl, indexStart, lpszFind)
ListBox_GetCount (hwndCtl)	ListBox_SelectItemData (hwndCtl, indexStart, data)
ListBox_Resetcontent (hwndCtl)	ListBox_GetSel (hwndCtl, index)
ListBox_AddString (hwndCtl, lpsz)	ListBox_GetSelCount (hwndCtl)
ListBox_InsertString (hwndCtl, index, lpsz)	ListBox_GetTopIndex (hwndCtl)
ListBox_AddItemData (hwndCtl, data)	ListBox_GetSelItems (hwndCtl, cItems, lpItems)
ListBox_InsertItemData (hwndCtl, index, data)	ListBox_SetTopIndex (hwndCtl, indexTop)
ListBox_DeleteString (hwndCtl, index)	ListBox_SetColumnWidth (hwndCtl, cxColumn)
ListBox_GetTextLen (hwndCtl, index)	ListBox_GetHorizontalExtent (hwndCtl)
ListBox_GetText (hwndCtrl, index, lpszBuffer)	ListBox_SetHorizontalExtent (hwndCtl, cxExtent)
ListBox_GetItemData (hwndCtl, index)	ListBox_SetTabStops (hwndCtl, cTabs, lpTabs)
ListBox_SetItemData (hwndCtrl, index, data)	ListBox_GetItemRect (hwndCtl, index, lprc)
ListBox_FindString (hwndCtl, indexStrart, lpszFind)	ListBox_SetCaretIndex (hwndCtl, index)
ListBox_FindItemData (hwndCtl, indexStart, data)	ListBox_GetCaretIndex (hwndCtl)
ListBox_SetSel (hwndCtl, fSelect, index)	ListBox_FindStringExact (hwndCtl, indexStart, lpszFind)
ListBox_SelItemRange (hwndCtl, fSelect, first, last)	ListBox_SetItemHeight (hwndCtl, index, cy)
ListBox_GetCurSel (hwndCtl)	ListBox_GetItemHeight (hwndCtl, index)
ListBox_SetCurSel (hwndCtl, index)	ListBox_Dir (hwndCtl, attrs, lpszFileSpec)

9.3.3 EDIT 类

当应用程序需要某些键盘输入的时候，最实际的办法就是使用 EDIT（编辑）类窗口。采用这种办法可以为我们带来很多好处。

首先，EDIT 类提供了多种方法对键盘键入进行控制，并且能够提供对鼠标的充分支持。

这对于大多数情况下的设计需求都已经足够了。除此以外，利用 EDIT 类提供的服务，还能显著地减少针对特殊窗口编写代码、检查通知代码以及发送消息（对输入进行捕获）等等方面的工作量。这些特性的重要性是不容低估的。

为了真正理解这种类窗口的灵活性，我们只需要观察一下它使用的风格就足够了。一个 EDIT 控件可以占据几个正文行，或者局限于一个单一的输入区（这对数据表单或者其他类似界面的建立是很用的）。由于存在多种风格组合，所以能在外观和感觉上生成具有明显区别的 EDIT 控件。许多开发者甚至希望在这一类窗口的管理上面得到更大的灵活性。如表 9-11 所示，其中为大家总结了这一类可以采用的所有 ES_风格。

关于风格，我想强调的一点是 ES_AUTOHSCROLL 和/或 ES_AUTOVSCROLL 并不表明能够自动显示一个垂直和/或滚动条。要得到这种结果，必须依赖于对 WS_VSCROLL 和 WS_HSCROLL 风格的设置。EDIT 类可以使用形形色色的消息，这些消息的作用是向自己的父/物主窗口说明当前的状态以及属于这一类的任何窗口的内容。如表 9-12 所示。

EDIT 类的通知代码列于表 9-13 内。

尽管在 Win32 里存在着平滑的线性地址空间，EDIT 类窗口仍然不能处理超过 32K 的信息。这个限制还没有从 Win16 里取消，尽管其中的部分原因是由于新的 RichEdit 通用控件的引入。

表 9-11　EDIT 类使用的风格

风　格	值	说　明
ES_LEFT	0x00000000L	正文左对齐排列
ES_CENTER	0x00000001L	正文居中显示
ES_RIGHT	0x00000002L	正文右对齐排列
ES_MULTILINE	0x00000004L	建立一个多行编辑区域
ES_UPPERCASE	0x00000008L	把键入的所有字母转换成大写形式
ES_LOWERCASE	0x00000010L	把键入的所有字母转换成小写形式
ES_PASSWORD	0x00000020L	在键入的字符位置处显示一系列星号；键入的任何字符都会由星号取代
ES_AUTOVSCROLL	0x00000040L	输入的内容在当前窗口区域放不下以后，就沿着垂直轴滚动。这个标志只能与 ES_MULTILINE 组合使用
ES_AUTOHSCROLL	0x00000080L	输入的内容在当前窗口区域放不下以后，就沿着水平轴滚动字符
ES_NOHIDESEL	0x00000100L	禁止使用这一类窗口的缺省行为，从而使选中的正文不会像传统的“黑底白字”那样显示出来
ES_OEMCONVERT	0x00000400L	把输入的字符转换成 OEM 字符
ES_READONLY	0x00000800L	建立一个只读 EDIT 窗口
ES_WANTRETURN	0x00001000L	带有 ES_MULTILINE 风格的窗口允许插入回车字符，而不是传统的那样按下 Enter 键

表 9-12 EDIT 类使用的消息

消息	值	说明
EM_GETSEL	(WM_USER+0)	返回当前的选择情况，指出选中的第一个字符以及最后一个未选字符
EM_SETSEL	(WM_USER+1)	定义准备选择的一系列字符
EM_GETRECT	(WM_USER+2)	返回对键入输入进行控制的那个矩形的尺寸；这个值与整个窗口矩形的尺寸也许不一致
EM_SETRECT	(WM_USER+3)	定义输入矩形的尺寸
EM_SETRECTNP	(WM_USER+4)	只能应用于由 ES_MULTILINE 标志建立的窗口，作用是定义输入矩形的尺寸(不实际地重画窗口)。输入矩形不能小于或者大于 EDIT 窗口的总体尺寸
EM_LINESCROLL	(WM_USER+6)	用于定义垂直滚动的距离（行数）和水平滚动的距离（列数）
EM_GETMODIFY	(WM_USER+8)	了解一个窗口的内存是否已被更改
EM_SETMODIFY	(WM_USER+9)	设置一个 EDIT 窗口的修改标志
EM_GETLINECOUNT	(WM_USER+10)	返回多栏编辑控件内的正文行数
EM_LINEINDEX	(WM_USER+11)	在 EDIT 窗口内，返回从一行起始处到某个给定字符处的字符总数
EM_SETHANDLE	(WM_USER+12)	设置某个内存区域的内存句柄，用户键入的字符将存储于这个内存区域里
EM_GETHANDLE	(WM_USER+13)	返回某个内存区域的内存句柄，用户键入的字符将存储于这个内存区域里
EM_LINELENGTH	(WM_USER+17)	返回正文行的长度
EM_REPLACESEL	(WM_USER+18)	替换选定的正文
EM_SETFONT	(WM_USER+19)	通过指定一个对应的句柄，从而设置 EDIT 窗口的字体
EM_GETLINE	(WM_USER+20)	从 EDIT 窗口内取回一个正文行
EM_LIMITTEXT	(WM_USER+21)	限制 EDIT 窗口内可以键入的字符总数
EM_CANUNDO	(WM_USER+22)	检查是否有可能撤消（Undo）上一次操作
EM_UNDO	(WM_USER+23)	执行撤消
EM_FMTLINES	(WM_USER+24)	在因为字卷绕需要而换行的正文行内，允许或者不允许使用软换行符
EM_LINEFROMCHAR	(WM_USER+25)	从正文起始处计数，某个特定编号的字符所在的正文行数
EM_SETWORDBREAK	(WM_USER+26)	激活/屏蔽单词间断
EM_SETTABSTOPS	(WM_USER+27)	设置一个多行编辑控件内的制表位，用对话框的度量单位进行表达
EM_SETPASSWORDCHAR	(WM_USER+28)	在编辑控件使用了 ES_PASSWORD 标志的前提下，指定用键盘输入的字符应用什么字符替换
EM_EMPTYUNDOBUFFER	(WM_USER+29)	腾空“撤消”（Undo）缓冲区
EM_GETFIRSTVISIBLELINE	(WM_USER+30)	返回第一个可见正文行的行号
EM_SETREADONLY	(WM_USER+31)	屏蔽对 EDIT 控件进行的任何输入操作
EM_SETWORDBREAKPROC	(WM_USER+32)	指出应该使用哪个窗口对“单词间断”进行控制
EM_GETWORDBREAKPROC	(WM_USER+33)	返回对“单词间断”进行控制的那个窗口进程的句柄
EM_GETPASSWORDCHAR	(WM_USER+34)	在编辑控件使用了 ES_PASSWORD 标志的前提下，返回用于替换键盘输入字符的那个字符

表 9-13 EDIT 类使用的通知代码

通知代码	值	说明
EN_SETFOCUS	0x0100	窗口接收到了视觉焦点
EN_KILLFOCUS	0x0200	窗口失去了视觉焦点
EN_CHANGE	0x0030	窗口的内容已经改变了
EN_UPDATE	0x0400	在窗口内容显示出来之前的那一刻接收到
EN_ERRSPACE	0x0500	与内存存储有关的错误
EN_MAXTEXT	0x0501	允许键入的最大字符数已经达到了
EN_HSCROLL	0x0601	水平滚动
EN_VSCROLL	0x0602	垂直滚动

9.3.4 EDIT 类的宏

EDIT 类的所有宏函数如下所示。宏函数的高位数字对消息的高位数字对应。

```
Edit_Enable (hwndCtl, fEnable)
Edit_GetText (hwndCtl, lpch, cchMax)
Edit_GetTextLength (hwndCtl)
Edit_SetText (hwndCtl, lpsz)
 (void) Edit_LimitText (hwndCtl, cchMax)
 (int) (DWORD) Edit_GetLineCount (hwndCtl)
 (int) (DWORD) Edit_GetLine (hwndCtl , line, lpch, cchMax)
 (void) Edit_GetRect (hwndCtl, lprc)
 (void) Edit_SetRect (hwndCtl, lprc)
 (void) Edit_SetRectNoPaint (hwndCtl, lprc)
 (DWORD) Edit_GetSel (hwndCtl)
 (void) Edit_SetSel (hwndCtl)
 (void) Edit_ReplaceSel (hwndCtl, lpszReplace)
 (BOOL) (DWORD) Edit_getModify (hwndCtl)
 (void) Edit_SetModify (hwndCtl, fModified)
 (int) (DWORD) Edit_LineFromChar (hwndCtl, ich)
 (int) (DWORD) Edit_LineIndex (hwndCtl, line)
 (int) (DWORD) Edit_LineLength (hwndCtl, line)
 (void) Edit_Scroll (hwndCtl, dv, dh)
 (BOOL) (DWORD) Edit_CanUndo (hwndCtl)
 (BOOL) (DWORD) Edit_Undo (hwndCtl)
 (void) Edit_EmptyUndoBuffer (hwndCtl)
 (void) Edit_SetPasswordChar (hwndCtl, fAddEOL)
 (void) Edit_SetTabStops (hwndCtl, cTabs, lpTabs)
```

(BOOL) (DWORD) Edit_FmtLines (hwndCtl, fAddEOL)
(HLOCAL) (UINT) (DWORD) Edit_GetHandle (hwndCtl)
(void) Edit_SetHandle (hwndCtl, h)
(int) (DWORD) Edit_GetFirstVisibleLine (hwndCtl)
(BOOL) (DWORD) Edit_SetReadOnly (hwndCtl, fReadOnly)
(char) (DWORD) Edit_GetPasswordChar (hwndCtl)
(void) Edit_SetWordBreakProc (hwndCtl, lpfnWordBreak)
(EDITWORDBREAKPROC) Edit_GetWordBreakProc (hwndCtl)

9.3.5 COMBOBOX 类

仅从名字来看，COMOBOX（组合框）就足以揭示这一类窗口的本质了。组合框窗口实际是属于预定义类的其他两个窗口的组合。通常是一个列表框与一个编辑或者静态类窗口关联在一起。具体怎样选择要取决于窗口建立时设置的风格是什么。因此，组合框并不是一种单一的对象，而是两个不同实体的组合。尽管如此，我们在调用 CreateWindow () 函数建立一个组合框的时候，该函数返回的句柄仍然是唯一的。假如这个句柄引用的是其中的列表框部分或者其他组件，那么它是根本不可能建立起来的。

和其他所有窗口类一样，COMBOBOX 也有自己的一系列风格、消息以及通知代码。其中相当一部分与我们已经研究过的列表框、编辑框或者静态类窗口相似。因此，我们接下来只探讨这一类的某些特殊元素；大家可以参考关于列表框的对应小节，从中了解更详细的信息。

在表 9-14 里，大家可以看到组合框窗口类的一系列风格总结。表 9-15 显示了组合框使用的消息，而表 9-16 则列出了它使用的通知代码。

表 9-14 COMBOBOX 类使用的风格

风 格	值	说 明
CBS_SIMPLE	0x0001L	列表框部分是可见的
CBS_DROPDOWN	0x0002L	用一个列表框和一个编辑框建立组合框，其中的列表框部分在最初是隐藏起来的
CBS_DROPDOWNLIST	0x0003L	用一个列表框和一个静态窗口建立组合框，其中的列表框部分在最初是隐藏起来的
CBS_OWNERDRAWFIXED	0x0010L	用应用程序描绘出来的固定高度项目建立一个组合框
CBS_OWNERDRAWVARIABLE	0x0020L	用应用程序描绘出来的可变高度项目建立一个组合框
CBS_AUTOHSCROLL	0x0040L	EDIT 窗口部分可以进行水平滚动
CBS_OEMCONVERT	0x0080L	把正文从 ANSI 转换成 OEM 格式
CBS_SORT	0x0100L	插入正文串的时候，按照它们的 ANSI 值进行排序
CBS_HASSTRINGS	0x0200L	这个标志只能针对物主绘图组合框设置，其中只包含了正文串
CBS_NOINTEGRALHEIGHT	0x0400L	允许建立和控制一个特殊的组合框，它在垂直轴上能够使用任何尺寸，高度甚至可以不必局限于系统字体的正数倍
CBS_DISABLENOSCROLL	0x0800L	屏蔽水平滚动

表 9-15 COMBOBOX 类使用的消息

消 息	值	说 明
CB_GETEDITSEL	(WM_USER+0)	取回编辑框部分的选定
CB_LIMITTEXT	(WM_USER+1)	限制编辑框部分的字符数
CB_SETEDITSEL	(WM_USER+2)	设置编辑框部分的选定
CB_ADDSTRING	(WM_USER+3)	插入一个正文串
CB_DELETESRING	(WM_USER+4)	删除一个正文串
CB_DIR	(WM_USER+5)	用一个目录的内容填写列表框部分
CB_GETCOUNT	(WM_USER+6)	返回列表框部分的项目总数
CB_GETCURSEL	(WM_USER+7)	返回列表框部分当前的选定
CB_GETLBTEXT	(WM_USER+8)	从列表框部分取回正文
CB_GETLBTEXTLEN	(WM_USER+9)	返回列表框部分的正文长度
CB_INSERTSTRING	(WM_USER+10)	在列表框部分插入一个正文串
CB_RESETCONTENT	(WM_USER+11)	腾空列表框部分
CB_FINDSTRING	(WM_USER+12)	在列表框部分查找一个正文串
CB_SELECTSTRING	(WM_USER+13)	选择一个正文串
CB_SETCURSEL	(WM_USER+14)	定义列表框部分的当前选定
CB_SHOWDROPDOWN	(WM_USER+15)	显示列表框部分
CB_GETITEMDATA	(WM_USER+16)	返回列表框部分选定项目支持字节内的数据
CB_SETITEMDATA	(WM_USER+17)	设置列表框部分选定项目支持字节内的数据
CB_GETDROPPEDCONTROLRECT	(WM_USER+18)	用屏幕坐标系统返回一个组合框的列表框部分尺寸
CB_SETITEMHEIGHT	(WM_USER+19)	设置列表框部分的项目高度
CB_GETITEMHEIGHT	(WM_USER+20)	返回列表框部分的项目高度
CB_SETEXTENDEDUI	(WM_USER+21)	激活/屏蔽组合框的扩展人工交互规则
CB_GETEXTENDEDUI	(WM_USER+22)	报告组合框扩展人工交互标志的状态
CB_GETDROPPEDSTATE	(WM_USER+23)	报告组合框的列表框部分的可见性
CB_FINDSTRINGEXACT	(WM_USER+24)	在组合框的列表框部分内查找一个完全匹配的正文串

表 9-16 COMBOBOX 类使用的通知代码

通知代码	值	说 明
CBN_ERRSPACE	(-1)	与组合框有关的内存错误
CBN_SELCHANGE	1	在组合框的列表框部分内，选定发生了变化
CBN_DBLCLK	2	在组合框的列表框部分内，用户双击一个项目
CBN_SETFOCUS	3	组合框得到了视觉焦点
CBN_KILLFOCUS	4	组合框失去了视觉焦点
CBN_EDITCHANGE	5	组合框的 EDIT 部分发生了变化
CBN_EDITUPDATE	6	这条通知代码正好在编辑框部分实际输出到屏幕之前发出
CBN_DROPDOWN	7	组合框的列表框部分准备在屏幕上显示出来
CBN_CLOSEUP	8	组合框的列表框部分准备在屏幕上消失
CBN_SELENDOK	9	在组合框的列表框部分内，用户在某个项目上方连续按了两次鼠标左键（或者只按了一次鼠标左键，然后按下 Enter 键）
CBN_SELENDCANCEL	10	用户在列表框的某个项目上方按下了鼠标左键，然后在的其他某个窗口内按下了鼠标左键，从而使列表框在屏幕上消失

我们在前面已经介绍了建立一个物主绘图列表框时需要用到的规则，这些规则不作任何修改即可应用于组合框。另外，我们还要用到 COMBOBOX 类的一些宏函数，如下所示：

ComboBox_Enable (hwndCtl, fEnable)
ComboBox_GetText (hwndCtl, lpch, cchMax)
ComboBox_GetTextLength (hwndCtl)
ComboBox_SetText (hwndCtl, lpsz)
(int) (DWORD) ComboBox_LimitText (hwndCtl, cchLimit)
(DWORD) ComboBox_GetEditSel (hwndCtl)
(int) (DWORD) ComboBox_SetEditSel (hwndCtl, ichStart, ichEnd)
(int) (DWORD) ComboBox_GetCount (hwndCtl)
(int) (DWORD) ComboBox_ResetContent (hwndCtl)
(int) (DWORD) ComboBox_AddString (hwndCtl, lpsz)
(int) (DWORD) ComboBox_InsertString (hwndCtl, index, lpsz)
(int) (DWORD) ComboBox_AddItemData (hwndCtl, data)
(int) (DWORD) ComboBox_InsertItemData (hwndCtl, index, data)
(int) (DWORD) ComboBox_DeleteString (hwndCtl, index)
(int) (DWORD) ComboBox_GetLBTextLen (hwndCtl, index)
(int) (DWORD) ComboBox_GetLBText (hwndCtl, index, lpszBuffer)
(LRESULT) (DWORD) ComboBox_GetItemData (hwndCtl, index)
(int) (DWORD) ComboBox_SetItemData (hwndCtl, index, data)
(int) (DWORD) ComboBox_FindString (hwndCtl, indexStart, lpszFind)
(int) (DWORD) ComboBox_FindItemData (hwndCtl, indexStart, data)
(int) (DWORD) ComboBox_GetCurSel (hwndCtl)
(int) (DWORD) ComboBox_SetCurSel (hwndCtl, index)
(int) (DWORD) ComboBox_SelectString (hwndCtl, indexStart, lpszSelect)
(int) (DWORD) ComboBox_SelectItemData (hwndCtl, indexStart, data)
(int) (DWORD) ComboBox_Dir (hwndCtl, attrs, lpszFileSpec)
(BOOL) (DWORD) ComboBox_ShowDropdown (hwndCtl, fShow)
(int) (DWORD) ComboBox_FindStringExact (hwndCtl, indexStart, lpszFind)
(BOOL) (DWORD) ComboBox_GetDroppedState (hwndCtl)
(void) ComboBox_GetDroppedControlRect (hwndCtl, lprc)
(int) (DWORD) ComboBox_GetItemHeight (hwndCtl)
(int) (DWORD) ComboBox_SetItemHeight (hwndCtl, cyItem)
(UINT) (DWORD) ComboBox_GetExtendedUI (hwndCtl)
(int) (DWORD) ComboBox_SetExtendedUI (hwndCtl, flags)

9.3.6 STATIC 类

这一类窗口扮演的角色是非常特殊的，因为它为设计者提供了一系列容器，用它们来存放正文或者图形。用户与 STATIC（静态）类窗口之间的交互作用实际是不可能发生的，这是因为用户不能通过键盘或者鼠标在其中插入任何正文或者改变其中业已存在的内容。正如

家也许已经猜到的那样，这一种窗口类没有任何消息或者通知代码可用。然而，我们仍然以针对它应用几种风格，这些风格在表 9-17 内进行了总结。

表 9-17 STATIC 类使用的风格

风 格	值	说 明
SS_LEFT	0x00000000L	正文左对齐
SS_CENTER	0x00000001L	正文居中排列
SS_RIGHT	0x00000002L	正文右对齐
SS_ICON	0x00000003L	把控件的表面定义成包含了图标的一个区域
SS_BLACKRECT	0x00000004L	建立一个黑色矩形
SS_GRAYRECT	0x00000005L	建立一个灰色矩形
SS_WHITERECT	0x00000006L	建立一个白色矩形
SS_BLACKFRAME	0x00000007L	用黑色边框建立一个空矩形
SS_GRAYFRAME	0x00000008L	用灰色边框建立一个空矩形
SS_WHITEFRAME	0x00000009L	用白色边框建立一个空矩形
SS_USERITEM	0x0000000AL	用户自定义项目
SS_SIMPLE	0x0000000BL	用左对齐正文串定义窗口
SS_LEFTNOWORDWRAP	0x0000000CL	用左对齐正文串定义窗口。任何制表符都会扩展，而且正文也会自动剪切，从而与窗口的尺寸匹配
SS_BITMAP	0x0000000EL	把 STATIC 窗口转换成用于显示位图图象的一个区域
SS_NOPREFIX	0x00000080L	任何 & 字符都不解释成用于指定控件加速键的前缀
SS_OWNERDRAW	0x0000000DL	建立一个物主绘图 STATIC 窗口
SS_ENHMETAFILE	0x0000000FL	显示一个增强型元文件
SS_ETCHEDHORZ	0x00000010L	三维水平边框
SS_ETCHEDVERT	0x00000011L	三维垂直边框
SS_ETCHEDFRAME	0x00000012L	围绕于控件的三维边框
SS_NOTIFY	0x00000100L	用户在一个静态窗口上方按下鼠标按钮时，强迫控件发送一个 STN_NOTIFY 通知代码
SS_CENTERIMAGE	0x00000200L	使 STATIC 控件内的图象居中显示
SS_RIGHTJUST	0x00000400L	使 STATIC 窗口内的正文右对齐显示
SS_REALSIZEIMAGE	0x00000800L	按照所含图象的真实尺寸重新定义控件的大小
SS_SUNKEN	0x00001000L	使编辑窗口具有一种凹陷外观

在图 9-6 里，大家可以看到一个对话框。这个对话框包含了属于 Windows 95 预定义 STATIC 类的几个窗口。

为了理解怎样运用 STATIC 类，并且显示一幅图象，大家可以参考一下本书附带 CD 的 Listing 9.1，其中介绍了一个名为 CONTROLS 的例子。该程序在自己的客户区内重新生成了本书封面使用的图象。然而，这种说法也不完全正确，因为本书的封面图象要依赖于用 SS_BITMAP 标志建立的一个 STATIC 窗口，指示该窗口显示一幅位图图象。在这种情况下，我们放置一个位图资源标签就足够了，这个标签与窗口标题有关的参数是对应的，就像下面这个代码段展示的那样：

```
...
// static
hwndt = CreateWindow (" static", szTitle,
```

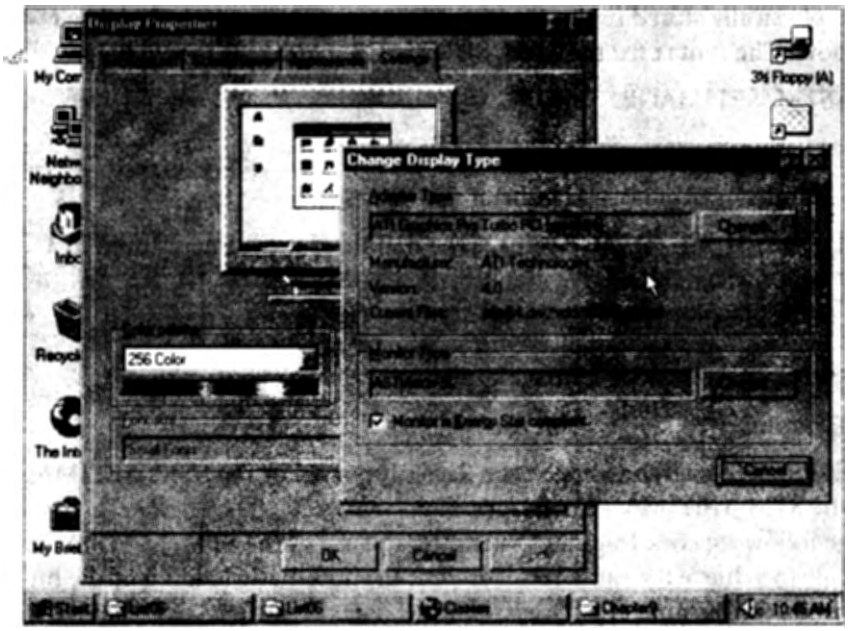


图 9-16　来自桌面属性表的 Change Display Type（更改显示器类型）对话框包含了属于 STATIC 类的几个窗口

```
                    WS_CHILD | WS_VISIBLE | SS_SUNKEN,
                    10, 110, 200, 20,
                    hwnd,
                    (HMENU) CT_STATIC,
                    hInstance, NULL);
...
```

在这儿，我们并不需要根据位图尺寸来设置窗口的大小，因为静态窗口可以根据实际情况自动地调整自己的尺寸。假如位图并没有物理性地存储于资源文件里，那么前面介绍的那种方法就不再适用了。STM_SETIMAGE 消息的语法如下所示：

STM_SETIMAGE	0x0172
wParam	图象类型标志
lParam	图象的句柄

类似地，STM_GETIMAGE 的作用是返回存储于静态窗口内的那幅位图的句柄：

STM_SETIMAGE	0x0173
wParam	图象类型标志
lParam	未使用

在表 9-18 里，我们列出了由 STM_SETIMAGE 和 STM_GETIMAGE 消息支持的所有图象类型标志。

下面这个代码段向我们阐明了如何利用 STM_SETIMAGE 消息来显示位图，从而获得与从资源文件里自动装载位图一样的结果：

```
...
HBITMAP hbmp;

hbmp = LoadBitmap (hInstance, " cover");
SendMessage (hwndt, STM_SETIMAGE, IMAGE_BITMAP,
     (LPARAM) hbmp);
...
```

表 9-18　把一幅位图与 STATIC 窗口关联起来的位图类型标志

位图类型	说明	位图类型	说明
IMAGE_BITMAP	位图图象	IMAGE_ENHMETAFILE	增强型元文件
IMAGE_CURSOR	光标图象	IMAGE_ICON	图标图象

下面列出来的是两个宏函数，利用它们可以简化图标在一个静态窗口内的插入，以及把图标从相同的窗口里取出来：

```
Static_SetIcon (hwndCtl, hIcon) ( (HICON) (UINT) (DWORD) SendMessage
                ( (hwndCtl), \STM_SETICON, (WPARAM) (HICON)
                (hIcon), 0L))
Static_GetIcon (hwndCtl, hIcon) ( (HICON) (UINT) (DWORD) SendMessage
                ( (hwndCtl), \STM_GETICON, 0L, 0L))
```

STATIC 控件的另外一项新特性是它采用了 SS_NOTIFY 风格。STATIC 窗口肯定是静态的，无法与用户进行任何形式的交互作用。然而，这种情况在 Win32 里也有可能发生例外。因为 SS_NOTIFY 风格的存在，所以父窗口也可以从静态控件里接收到一条通知代码。到现在为止，我们只能使用四条静态通知代码。通过它们可以捕获一个 STATIC 窗口内发生的鼠标事件，或者判断控件是否处于活动状态。在设置了 SS_NOTIFY 的前提下，假如用户在一个 STATIC 控件内单击鼠标按钮，父窗口就会接收到一条 WM_COMMAND 消息。这条消息的 lParam 内包括了常规的句柄信息，控件 ID 包含于 wParam 的低字内，STN_CLICKED 通知代码则包含于 wParam 的高字内。用于这一类窗口的另外三条通知代码包括 STN_DBLCLK，STN_ENABLE 以及 STN_DISABLE 等。

用 STATIC 控件显示一幅图象的时候对 SS_NOTIFY 风格的支持变显得相当重要了。通过对 STN_CLICKED 通知代码进行捕获，程序很容易就可以判断用户是否单击了位图的一

个特定部分，并且采取对应的操作。

9.3.7 SCROLLBAR 类

注意下面这一个事实是至关重要的：单独的滚动条无法完成任何实际的滚动工作！滚动条是一种简单的图形化对象，利用它可以到达一幅位图或者正文总体尺寸内的某个特定位置。对于与滚动条关联在一起的窗口来说，它是通过调用 ScrollWindow () 或者 ScrollWindowEx () 函数来执行滚动操作的。

SCROLLBAR 类另外一项不寻常的概念与它的风格、消息以及通知代码有关。如表 9-19 所示，它没有什么特殊的风格，只有以 WS_引入的两种风格。

表 9-19 SCROLLBAR 类使用的消息

风 格	值	说 明
WS_HSCROLL	0x00100000L	建立一个水平滚动栏
WS_VSCROLL	0x00200000L	建立一个垂直滚动栏

假如位置指示器（亦常常被称为“滑块”）发生了运动，就会生成对应的 WM_HSCROLL（水平滚动）或者 WM_VSCROLL（垂直滚动）消息。这与其他所有的窗口都是不一样的，其他窗口通过 WM_COMMAND 消息指出自己内部的变化。这两条消息的语法结构如下所示：

WM_HSCROLL－WM_VSCROLL	0x00100000L－0x00200000L
LOWORD（wParam）	滚动栏标识符
HIWORD（wParam）	滑块的当前位置
lParam	滚动栏句柄（hwnd）

wParam 的值是至关重要的，因为只有通过它，父/物主窗口才能判断用户在滚动条上进行了什么操作。在表 9-20 里，大家可以看到一系列可能的 WM_HSCROLL 和 WM_VSCROLL 值。

表 9-20 由滚动条通过 WM_VSCROLL 和 WM_HSCROLL 消息的 wParam 返回的值

通知值	值	说 明
SB_BOTTOM	7	移动到滚动栏的底部或者最左侧位置
SB_RIGHT	7	移动到右侧
SB_ENDSCROLL	8	移动到最后的滚动位置处
SB_LINEDOWN	1	向下滚动一行
SB_LINERIGHT	1	向右滚动一行
SB_LINEUP	0	向上滚动一行
SB_LINELEFT	0	向左滚动一行
SB_PAGEDOWN	3	向下滚动一页
SB_PAGELEFT	3	向左滚动一页
SB_PAGEUP	2	向上滚动一页
SB_PAGELEFT	2	向左滚动一页
SB_THUMBPOSITION	4	滑块的位置被用户改变了
SB_THUMBTRACK	5	用户正在拖动滑块
SB_TOP	6	滚动到滚动条的顶部
SB_LEFT	6	向左滚动

应用程序可以利用 SetScrollRange () 函数指定一个最大值和一个最小值，从而建立实际的滚动范围。

```
#include 〈winuser.h〉
void WINAPI SetScrollRange (HWND hwnd,
                            int fnBar,
                            int nMin,
                            int nMax,
                            BOOL fRedraw);
```

参数	说明
HWND hwnd	窗口句柄
int fnBar	用 SB_CTL 指定一个控件，用 SB_HORZ 指定一个水平滚动栏，或者用 SB_VERT 指定一个垂直滚动栏
int nMin	最小的滚动位置
int nMax	最大的滚动位置
BOOL fRedraw	指出滚动栏准备重画 (TRUE) 或者不准备重画 (FALSE)
返回值	在正文里讨论
void	没有返回值

第一个句柄直接对应于 SCROLLBAR 类的窗口或者其他窗口(前提是这个窗口内包含了属于这一类的控件)。假如 SetScrollRange () 行动的接收者是一个滚动栏，那么第二个参数必须设置成 SB_CTL 定义。假如是父窗口里带有属于 SCROLLBAR 类的一个控件，第二个参数的值就必须设置成 SB_HORZ，SB_VERT 或者 SB_BOTH。这些定义分别对应于一个水平滚动栏、一个垂直滚动栏或者两者均有。下面是这些定义的值：

```
SB_HORZ    0
SB_VERT    1
SB_CTL     2
SB_BOTH    3
```

第三个参数指定了滑块的起始位置，后面跟上滚动可以接受的最大距离。这些标识符可以采用任何值，因为滚动通常都是应用程序的任务。这个函数最后指定了一个布尔值，它表明滚动栏窗口是否需要重画，从而对滚动状态进行强调。滚动范围的下限和上限分别对应于0和100。

为了定义滑块的位置，我们必须调用 SetScrollPos () 函数：

```
#include 〈winuser.h〉
```

```
int WINAPI SetScrollPos (HWND hwnd,
                         int fnBar,
                         int nPos,
                         BOOL fRepaint);
```

参数	说明
HWND hwnd	窗口句柄
int fnBar	用 SB_CTL 指定一个控件，用 SB_HORZ 指定一个水平滚动栏．或者用 SB_VERT 指定一个垂直滚动栏
int nPos	用滚动范围定义的新位置
BOOL fRepaint	指出对滚动栏进行重画（TRUE）还是不重画（FALSE）
返回值	在正文里讨论
int	滚动栏的前一次位置

前面两个参数与 SetScrollRange () 里的对应参数是完全一致的。滑块的位置是用第三个参数指定的，这个值显然必须在 SetScrollRange () 函数设定的范围以内。最后一个布尔值参数指定了是否需要进行重画操作。

用于与滚动栏进行交互作用的函数还有 GetScrollRange () 和 GetScrollPos ()。通过对 WM_VSCROLL 和 WM_HSCROLL 消息的拦截，应用程序可以对不同的 SB_值进行区分，并且把标识符给定的新值分配给一个静态标识符。例如，滚动的距离可以是1，10或者5；用于度量的单位并不重要，因为这要取决于应用程序。

程序接收到 SB_THUMBPOSITION 或者 SB_THUMBTRACK 代码以后，并不需要为标识符分配任何值，从而对位置进行跟踪，因为我们需要的全部信息都包含在 wParam 的低字里。无论用什么规范对滑块的位置进行比较，最终都需要调用 SetScrollPos ()，利用它对滑块在滚动条窗口内的位置进行更新。

下面陈列的是与滚动栏进行交互作用的一种基本框架：

```
...
static WORD wPos;
...
case WM_HSCROLL:
    switch (LOWORD (wParam))
    {
        case SB_LINEDOWN:
            wPos++;
            break;

        case SB_PAGEDOWN:
            wPos += 10;
```

```
            break;

        case SB_THUMBPOSITION:
            wPos = LOWORD (lParam);
            break;

        case SB_xxx:
            ...;
            break;
    }
    // out of range?
    if (wPos < wMin)
        wPos = wMin);
    if (wPos > wMax)
        wPos = wMax);

SetScrollPos (hwndScrollBar, SB_CTL, wPos, TRUE);
}
break;
...
```

我们也可以把滚动栏当作一种输入工具使用，方法是把它与一个 STATIC 或者 EDIT 窗口关联起来，让用户在一个预先定义好的范围以内作出选择（文档的页号、病毒清单等等）。在这种情况下，滚动的概念就显得有些模糊了。

9.4 资源列举

请大家参考本书附带 CD 的 Listing 9.7，其中介绍了一个名为 ENUMRES 的示范程序。这个程序与 TOOL 例子非常相似，都采用了叠置于应用程序客户区上面的一个列表框窗口。这个 Win32程序的作用是列举出可执行模块内存在的所有资源，为每一种资源显示一个正文串，说明该资源的名称和本质。除此以外，假如双击一个资源名称，还可以显示出一个弹出式窗口，显示出实际的资源情况。

ENUMRES 程序的另外一项功能是它可以对整个文件系统进行浏览，从而对任何一个可执行程序和 DLL（动态链接库）进行检查。图9-17显示的就是对 PLATFORM 应用程序进行检查后的输出结果。PLATFORM 程序可以在本书附带 CD 的 Listing 1.1内找到，大家亦可参考本书第一章“Win32中的软件开发”对该程序的介绍。通过图9-17，我们注意到一个非常有趣的现象，那就是在 Win32里，一个传统的图标现在被当作一组图标进行处理，这是由于小图标和普通图标的存在。

尽管所有的输出操作现在都非常明显，但我们探究一下对 PE 执行模块进行实际读取的代码段还是很有帮助作用的。通过这个代码段，应用程序能对现成的资源进行判断和侦测。为

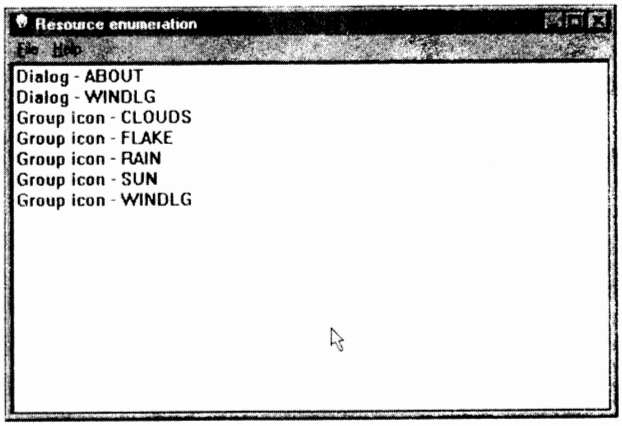


图 9-17　ENUMRES 列出了执行模块内的所有资源类型

了达到这个目的，用户在一个通用对话框里选择了正确的 EXE 或者 DLL 文件以后，ENUMRES 就会调用 LoadLibraryEx（）这个 API 函数，如图9-18所示。

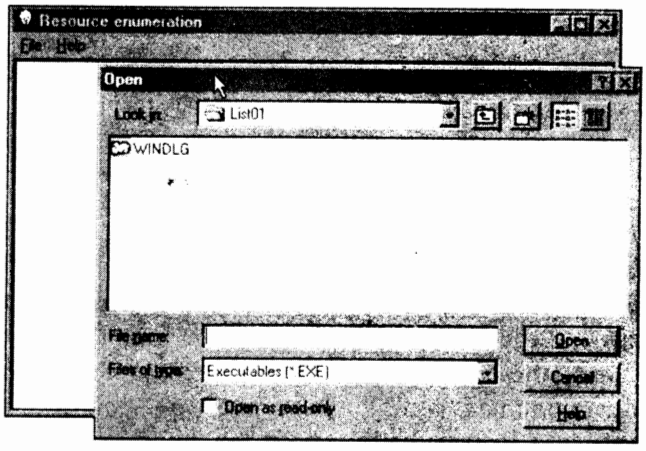


图 9-18　选择一个文件用 ENUMRES 进行检查

LoadLibraryEx（）是一种功能很强的函数，然而它在 Windows 95里并未完全得以实现。该函数可以把任何种类的 EXE 模块（包括 DLL 模块）载入一个进程地址空间内，同时返回它的模板句柄。

```
// get the module of the selected executable or DLL file
hModule = LoadLibrary (qfn.lpstrFile, NULL, 0);
```

在上面这个代码段里，LoadLibraryEx（）的表现就像一个普通的 LoadLibrary（），并没有什么特别的功能增强之处。从现在开始，ENUMRES 程序会用这个模板句柄去指定准备检查的对象文件。这个简单的技巧可以帮助我们解决本书一开头就提到的那个臭名昭著的 hInstance 问题。

在这个时候，ENUMRES 已经做好准备，可以把待检查文件内的所有资源列举出来了。假如选中 Enumerate 菜单项，屏幕上就会列出 PE 执行模板内的所有资源信息。所有这些逻辑都

是以 EnumResourceTypes () 和 EnumResourceNames () 这两个 API 函数为基础的：

```
#include <winbase.h>
BOOL EnumResourceTypes (HMODULE hModule,
                        ENUMRESTRYPEPROC lpEnumFunc,
                        LONG lparam);
```

参数	说明
HMODULE hModule	应用程序模块句柄
ENUMRESTRYPEPROC lpEnumFunc	用于列出所有资源类型的列举进程的名字
LONG lparam	由应用程序定义的参数，可以为任何值
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

这个函数要求把一个模块句柄用作自己的第一个参数，用于指定准备进行检查的那个进程；对于 ENUMRES 程序来说，它是通过浏览文件系统这种准备工作获得这种信息的。第二个参数是指一个用户自定义函数，它将由 EnumResourceTypes () 函数进行周期性的调用，每调用一次就取回一种资源类型。PE 对象是根据表5-3所列的某个 RT_定义对资源进行分类的。ENUMRES 程序内的资源类型列举函数是相当简单的，它的作用是启动另外一次查找，从而判断出属于那个种类的所有资源。

```
BOOL CALLBACK EnumResTypeProc (HANDLE hModule,
                               LPTSTR pszType,
                               LONG lParam)
{
    PDATA pdata = (PDATA) lParam;

    // increase the resource counter
    pdata -> iCnt++;
    EnumResourceNames ( (HMODULE) pdata -> hModule, pszType,
                        EnumResNameProc, (long) lParam);

return TRUE;
```

EnumResourceTypes () 的第三个和最后一个参数是一个四字节长的值，应用程序开发者可以通过任何形式对其进行利用。正如大家在下面这个 EnumResTypeProc () 函数里看到的那样，ENUMRES 程序把一个指针传递给了由应用程序定义的数据结构，这个数据结构属于 DATA 类型，其中的信息包括一个资源类型的计数器以及一个模块句柄：

```
typedef struct _DATA
{
    HWND hwnd;
    HWND hwndList;
    HMODULE hModule;
    int iCnt;
} DATA, *PDATA;
```

总而言之，EnumResTypeProc () 会根据需要调用多次，每一次都会返回待查文件里找到的一种资源类型。在这个函数里，ENUMRES 程序调用了 EnumResourceNames () 函数，从而实际地列出所有资源名以及它们各自的类型。这种信息最终会在重叠于客户区上方的列表框内显示出来。

```
#include <winbase.h>
BOOL EnumResourceNames (HINSTANE hModule,
                        LPCTSTR lpszType,
                        ENUMRESNAMEPROC lpEnumFunc,
                        LONG lParam);
```

参数	说明
HMODULE hModule	应用程序模块句柄
LPCTSTR lpszType	资源类型，可以在表5-3列出的 RT_定义中选用一个
ENUMRESNAMEPROC lpEnumFunc	回调函数的名字，针对属于那一类的每种列举资源名称进行调用
LONG lParam	由应用程序定义的参数；可以为任何值
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

EnumResourceNames () 函数与 EnumResourceTypes () 函数具有非常类似的行为。EnumResourceNames () 函数的作用是接收关于资源类型的信息，具体则反应在其中的 lpszType 参数上面。随后，它会调用由第三个参数指定的函数。如前面那个代码段所示，这时调用的是 EnumResNameProc ()，用它来列出属于那种类型的所有资源。

```
BOOL CALLBACK EnumResNameProc (HANDLE hModule,
                               LPCTSTR lpszType,
                               LPTSTR lpszName,
                               LONG lParam)
```

```
{
    PDATA pdata = (PDATA) lParam;
    char szText [100];
    int iPos;

    if ( * lpszName)
    {
        switch ( (int) lpszType)
        {
            case RT_BITMAP:
                ...
                break;

            case RT_FONT:
                ...
                break;

            case RT_FONTDIR:
                ...
                break;

            case RT_ACCELERATOR:
                ...
                break;
            ...

        }
        // insert the text in the listbox
        iPos = ListBox_AddString (pdata -> hwndList, szText);
        // store the resource type in the item extra bytes
        ListBox_SetItemData (pdata-> hwndList, iPos, (int) lpszType);
    }
    return TRUE;
}
```

在这个函数里，“切换”(switch)代码块的作用是让我们对最终设置到列表框窗口内的输出字符串进行定制。

ENUMRES 程序还能帮助我们做更多的事情：它能用一个单独的弹出式窗口显示出所列资源的实际情况。图9-19展示了与位图资源（标签名为 WIN32BK）关联在一起的位图图象的

情况。

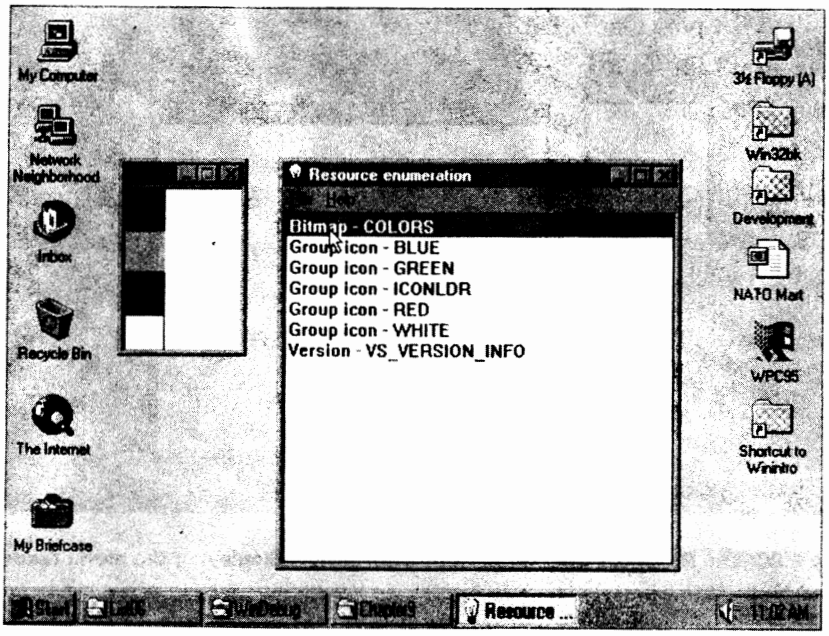


图 9-19　双击列表框窗口内某种资源的名字，Resource enumeration 窗口就能显示出选中资源的具体情况

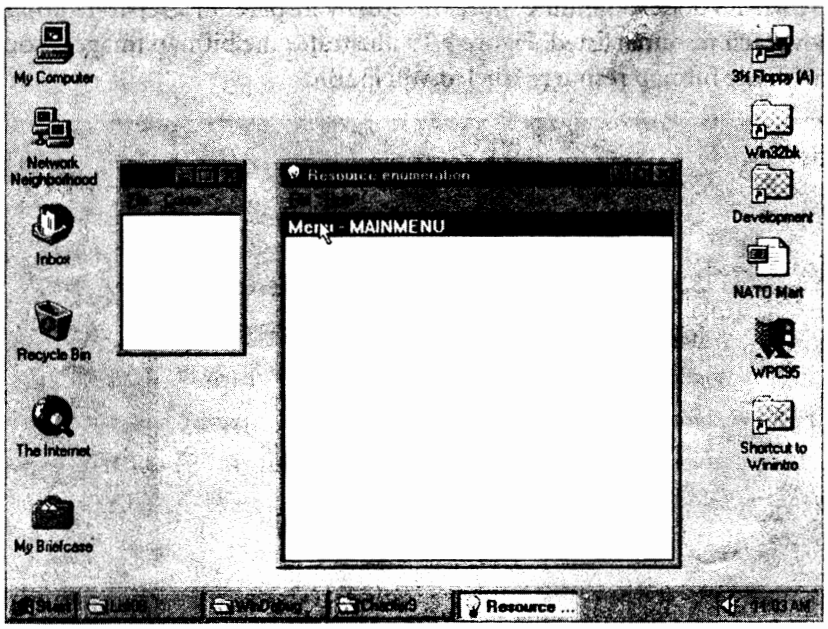


图 9-20　资源列举窗口现在显示的是标签名为 MAINMENU 的菜单资源

假如列表框窗口内的选定情况发生了变化，弹出式窗口就会相应地发生改变，如图9-20所

示——其中的位图已经消失了，取而代之的是一个菜单资源。

在开始讨论本章的最后一个主题之前，我要先向开发者们提一条建议：请确保希望用ENUMRES 程序检查的那个可执行文件是用版本2.55或者更高版本的链接程序 生成的。以前的链接程序版本是按照一种不同的方案对资源信息进行链接的(除此以外，在Visual C++2.1里，把编译的资源插入一个可执行模块时，会发生一些奇怪的现象)。因此，有时并不能用这个示范程序侦查到实际存在于一个 EXE 程序内的所有资源。假如希望摆脱这些烦恼，只需在 Windows NT 3.51系统内测试 ENUMRES 程序，到时所有问题都会迎刃而解！

9.5 图标的提取

本章的最后一个例子可以帮助我们做一些有趣的事情：它能从用户选择的任何 EXE 或者 DLL 文件里提取出图标。这个 EXTRACT 示范程序允许用户选择一个图标，然后把它拷贝到剪贴板内，以便其他应用程序对其进行利用。

EXTRACT 启动的时候，它不会做任何事情。Open 是其中唯一可用的一个菜单项，利用 Open 可以访问一个通用对话框，这个对话框与我们在 ENUMRES 程序里看到的是完全一致的。一旦选中了某个有效的文件，EXTRACT 就会在自己的客户区显示出与那个文件关联在一起的所有图标；同时在应用程序的标题内增加它提取出来的图标数目，如图9-21所示。

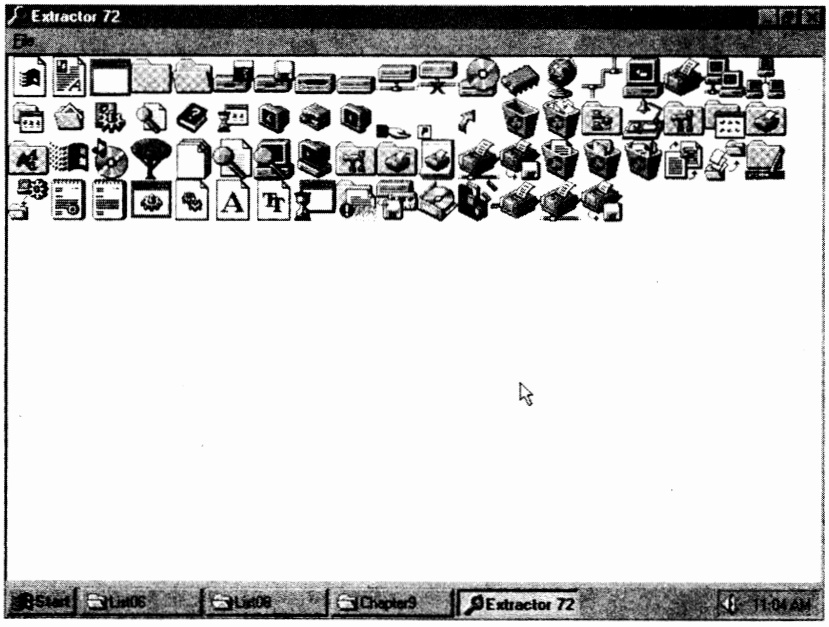


图 9-21 EXTRACT 显示出了 SHELL32.DLL 里所有的72个图标

图标是根据一种看不见的网格在客户区内排列的，这种网格与窗口的尺寸是对应的。假如窗口缩小了，图标也会相应地重新排列。如图9-22所示。

在某个图标上方单击鼠标右键，就会在屏幕上显示出一个小的弹出式关联菜单，它正好位于鼠标单击位置的下方，如图9-23所示。图9-24则在剪贴板应用程序内显示了一个相同的图标。这个图标随后可以粘贴到 MS Paint 里，这是随同 Windows 95提供的一个附件程序。

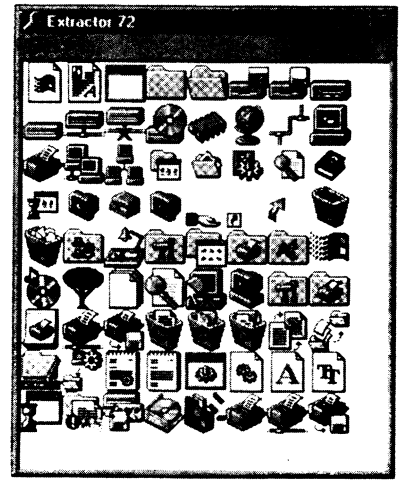


图 9-22　窗口缩小后，72个图标的排布情况有所不同

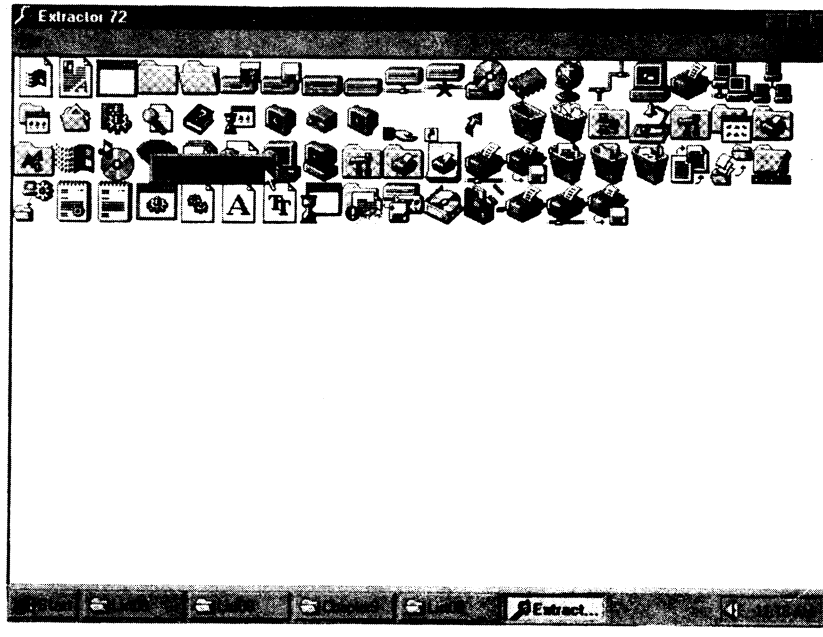


图 9-23　如果想把一个图标拷贝到剪贴板内，只需在它上方按下鼠标右键即可

现在让我们来探讨一下 EXTRACT 程序的工作原理。用户通过通用对话框选中了一个有效的文件以后，应用程序就会解除对10个内存页的分配。这个操作等效于把以前存储在这些页内的所有信息都删除掉。这个内存块最初是在拦截 WM_CREATE 消息的时候预约下来的，同时没有对其中的任何一个页进行委托。就在这以后，EXTRACT 程序需要通过调用 API 函数 ExtractIcon（），从而计算出目标文件内的图标数量。该函数的语法结构如下所示：

```
#include <shellapi.h>
HICON ExtractIcon (HINSTANCE hInst,
```

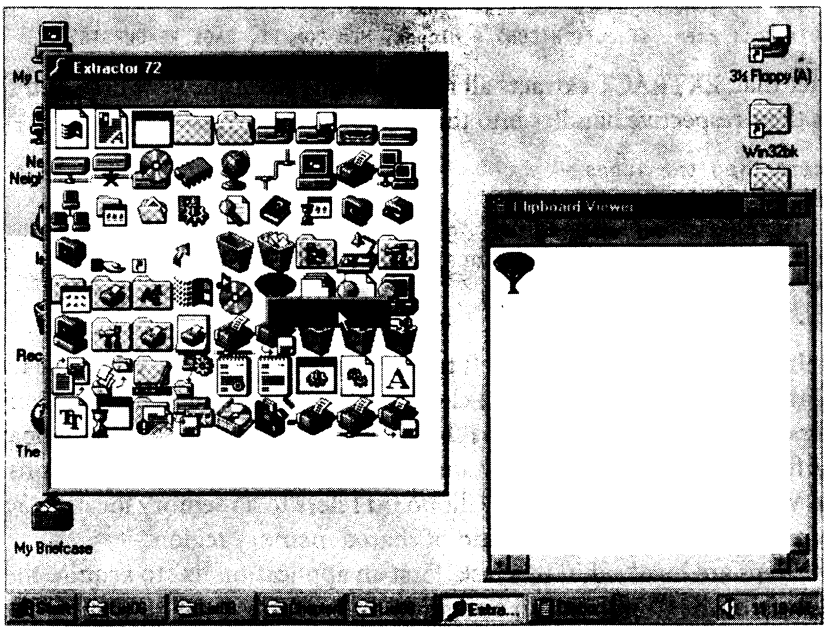


图 9-24　选中的图标已经拷贝到剪贴板的系统内存区域内了

```
                    LPCTSTR lpszExeFileName,
                    UINT nIconIndex);
```

参数	说明
HINSTANCE hInst	应用程序实例句柄
LPCTSTR lpszExeFileName	准备检查的 EXE 或者 DLL 文件的名字
UINT nIconIndex	准备提取的图标的编号，亦可设置成-1，表示返回 lpszExeFileName 里存在的图标总数
返回值	在正文里讨论
HICON	图标句柄，或者图标的总数

第一个参数是应用程序实例句柄，紧接着是准备检查的那个可执行模块的名字。第三个参数（也是最后一个）是一个数字，用于指定提取出那个图标；假如为-1，则表明我们只对那个文件内的图标总数感兴趣。在最初的时候，EXTRACT 是根据第二种方法来决定自己需要分配多少个页，从而存储所有的图标句柄：

```
// determing the number of icons stored in that file
nIcons = (UINT)ExtractIcon(hInstance, ofn.lpstrFileTitle, (UINT)-1);
// allocating some pages to store all the icons
VirtualAlloc(pmem,sizeof(HICON) * nIcons,MEM_COMMIT,PAGE_
    READWRITE);
```

在这以后，EXTRACT 会从第一个图标开始提取出所有图标，然后把各自对应的句柄放置到内存区域里：

```
// extracting the icons
while (i < nIcons)
{
        hicon = ExtractIcon (hInstance, ofn.lpstrFileTitle, i);
        * (pmem + i++) = hicon;
}
```

到这时为止，剩下的唯一操作就是在屏幕上把这些图标显示出来。假如用户用鼠标右键单击一个图标占据的客户区域位置，就会显示出一个小的弹出式菜单，这时就启动了拷贝操作，并把剪贴板（Clipboard）当作自己的最终目的地。作为 Windows 应用程序进程间通信的首要途径，剪贴板的本质是多个进程均可访问的一个内存区域——一种共享内存区。

为了完成这一任务，需要涉及到几个步骤。首先，应用程序必须取得剪贴板的所有权，这需要调用 OpenClipboard () 函数来实现。接下来还要调用 EmptyClipboard () 函数，用它把当前在剪贴板里的内容删除掉：

```
if (!OpenClipboard (hwnd))
    break;

EmptyClipboard ();
...
```

在这个时候，我们需要再次调用 ExtractIcon ()，从而把选定的图标从资源文件里提取出来。iIcon 整数内包含了图标的序号，这种信息是用户用鼠标右键单击客户区显示图标的时候由 EXTRACT 程序计算出来的。

```
...
// load the selected icon
hicon = ExtractIcon (hInstance, szFileName, iIcon);
// get icon info
GetIconInfo (hicon, &ii);
...
```

我们现在得到了一个图标，但是仍然存在问题！剪贴板只接收一些预先定义好的格式，这些格式中没有一个是专门针对图标的处理而设计的。我们可以通过一种简便的方法把图标转换成真正的位图对象吗？答案是……大概可以吧！我在这儿采用的方法是相当直接的，而且更为重要的一点是，它很有效！

这儿要回答的第一个问题与图标的内部本质有关。图标的本质是什么呢？它是一种具有特殊属性的尺寸固定的位图吗？这正是我对图标的理解。为了对这种概念有一个更清晰的理解，

我们可以调用新的 Win32 API 函数 GetIconInfo ()，该函数会在一个 ICONINFO 数据结构里填充某些有用的信息：

```
#include 〈winuser.h〉
BOOL GetIconInfo (HICON hIcon, PICONINFO piconinfo);
```

参数	说明
HICON hIcon	一个有效的图标句柄
PICONINFO piconinfo	一个 ICONINFO 数据结构的地址
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

其中，第二个参数是一个 ICONINFO 数据结构的地址，这种数据结构的定义如下所示：

```
typedef struct _ICONINFO
 { // ii
    BOOL fIcon;
    DWORD xHotspot;
    DWORD yHotspot;
    HBITMAP hbmMask;
    HBITMAP hbmColor;
} ICONINFO;
```

这个 API 以及相关的数据结构可以对图标和光标进行很好的处理。假如引用的信息是与图标有关的，那么 ICONINFO 的第一个参数就应设置成 TRUE；假如引用的是光标，就应设置成 FALSE。热点位置是由第二和第三个参数指定的，对于这种信息，我们在这儿是暂时用不上的。真正引起我们注意的是指定一个图标的两幅位图。图标实际就是两幅位图的结果，彩色和屏蔽（mask）位图，它们通过一个光栅操作代码（ROP）标志合并到了一块儿。

为了把图标转换成真正的位图，在这儿必须进行某些技术处理。EXTRACT 程序建立了两幅设备兼容位图，一幅选择成屏蔽图标位图，另一幅则选择成彩色图标位图，如下所示：

```
...
hdc = GetDC (hwnd);
hdcdst = CreateCompatibleDC (hdc);
hdcsrc = CreateCompatibleDC (hdc);
hbmpold = SelectObject (hdcdst, ii.hbmColor);
hbmpold = SelectObject (hdcsrc, ii.hbmMask);
...
```

对这两个 DC（设备现场）进行操作的 BitBlt（）为我们提供了访问结果位图的机会，如下所示：

```
...
BitBlt (hdcdst, 0, 0, SYS (SM_CXICON), SYS (SM_CYICON), hdcsrc, 0, 0,
    SRCPAINT);
// retrieve the new bitmap
hbmpnew = SelectObject (hdcdst, hbmpold);
...
```

hbmpnew 位图是指构成一个图标的两幅位图生成的合并位图。这种信息是通过 SetClipboardData（）函数传递给剪贴板的：

```
...
// pass the bitmap to the Clipboard
SetClipboardData (CF_BITMAP, hbmpnew);
// close the Clipboard
CloseClipboard ();

ReleaseDC (hwnd, hdc);
DeleteDC (hdcsrc);
DeleteDC (hdcdst);
...
```

这一部分代码结束的时候，需要清除以前建立的所有 DC，并且释放用于显示设备现场的缓冲区。

9.6 MDICLIENT 类

这个类是从3.0版本开始引入的，它的作用是简化多文档界面应用程序的编写。尽管这也属于七种预定义类的一种，但它并不遵循“控件”的概念，这种概念在前面介绍的其他六个类里是很典型的。由于这个原因，我们将在第十四章“多线程、IPC 和 I/O”里对它进行单独介绍。

第 10 章　Windows 95 通用控件

Windows 95 最有趣的特性就是它提供了一系列新的控件，我们把这些控件称为通用控件。除了传统的七种预定义窗口类以外（这七种窗口类自 1990 年 3 月推出 Windows 3. x 以来就一直存在至今），现在又新增了这十四种通用控件。它们使用户界面得到了极大的改进。系统本身也在自己的外壳里应用了这些新控件，从而具有了比以往更漂亮的外观。例如，我们可以用文件夹为例来说明这一点。文件夹的本质是一种列表框视窗通用控件，它的底部带有一个状态栏，菜单栏下方则带有一个工具栏。树形视窗用于直观地表达文件名的分级结构。在 Browse 按钮里则是 EXPLORER. EXE 的 Find 引擎。软盘对象内的 Copy Disk 菜单项利用了一个进展指示条来可视化地表示正在进行的拷盘操作。通用控件也在 MS Windows NT 3. 51 里得到了支持和实现。

表 10-1 为大家列出了所有这 14 种通用控件，并且提供了相应的类名以及简短的说明信息。

除了这些类以外，通用控件这个术语还应包含另外两个对象：属性表和图象列表。它们中的任何一个都不是实际的窗口类。更准确地说，它们各自都是一系列对话框、一些通用控件以及存储于一幅位图内的一系列图象或者图标的组合。属性表在系统外壳里是经常用到的，假如用户选择显示某个对象的属性，这个属性表就会显现出来。图象列表不能通过一个看得到的外壳组件进行展示，因为它们的功能是在系统幕后完成的。通常情况下，每个列表视窗控件都有自己对应的图象列表，有时甚至有多个这样的列表。除此以外，拖动列表框是对通用列表框类的一种扩展，它本身并不代表一个新类。

现在让我们首先定义这一组窗口类的整体特征，以此开始我们对通用控件的探讨。

表 10-1　Windows 95 的新通用控件

通用控件	类　名	说　明
动画	ANIMATE_CLASS	用于显示无声 AVI 文件的 MCI 控件
拖动列表框	LISTBOX	支持内部项目拖放功能的列表框
标题	SysHeader32	列表视窗在细节模式时使用的控件
热键	HOTKEY_CLASS	允许用户输入一系列组合键，将其用作热键的窗口类
列表视窗	SysListview32…	多功能控件，可以用多种方式显示信息（小图标和普通图标、详细列表模式和列表模式）
进展条	PROGRESS_CLASS	进展指示条
多功能编辑框	RichEdit	高级 EDIT 窗口控件
状态窗口	STATUSCLASSNAME	状态栏控件
制表符	WS_TABCONTROL	用于标识属性表内一个页（卡片）的控件
工具栏	TOOLBARCLASSNAME	工具栏控件
工具提示	TOOLTIPS_CLASS	小的正文窗口，假如鼠标指针在某个 UI（用户界面）组件上方停留半秒钟，就会在窗口内显示对这种 UI 组件的说明信息
滑杆	TRACKBAR_CLASS	滑动窗口，带有可选的刻度线
树形视窗	WC_TREEVIEW	用树形结构显示相应的信息
上下控件	UPDOWN_CLASS	旋转按钮控件

10.1 建立通用控件

与标准的控件比较，尽管通用控件为开发者提供了范围更广的机会，利用它们可以开发出更好、功能更强的应用程序，但是通用控件往往显得更复杂、更难以实现。在第九章“预定义的窗口类”里，我们对其中介绍的控件进行处理的时候，曾经提到过每个控件都有自己专用的一系列窗口风格、消息以及通知代码。这个规则同样也适用于通用控件，尽管它们之间还是存在一些区别。

首先，在没有调用 InitCommonControls () 这个 API 函数的前提下，我们不能建立任何一个通用控件。该函数的作用是载入和初始化 COMMCTL32.DLL 这个动态链接库。对 InitCommonControls () 的调用既可以在 WinMain () 里进行，也可以在主窗口进程内处理 WM_CREATE 消息时进行。只需要调用这个 API 函数一次，以后的整个进程都会持续有效，即使在使用了多线程的情况下也是如此。为了把通用控件包含在自己的源代码内，对这个函数的调用并不是需要做的唯一准备工作。除此以外，我们还需要把 COMCTL32.LIB 库文件包含于项目设置属性表内，以便链接程序对包括它在内的一系列库文件进行访问。第三个也是最后一个步骤是把 COMMCTRL.H 头文件包含在应用程序代码内，从而提供相应的定义、数据类型、风格、消息、通知代码以及宏等等。

总而言之，在编写一个通用控件的代码之前，必须记住要先完成下面这些工作：

▶增加 COMCTL32.LIB 库文件，从而对项目属性进行修改。

▶调用 InitCommonControls () 这个 API 函数，从而载入和初始化通用控件的动态链接库。

▶在源代码内包含 COMMCTRL.H，从而对用于 API 的所有通用控件提供支持。

所有这些准备工作都完成以后，接下来就可以自由地建立自己的通用控件了。对 CreateWindowEx () 的一次调用也许是生成一个新窗口最常用的方式。之所以要调用这个函数，是由于通用控件的本质就是普通的窗口。当然，尽管这种说法在常规的情况下是正确的，但是某些通用控件还提供了一些特定的函数，利用它们可以简化和加快窗口的建立进程，并且剔除了 CreateWindowEx () 里使用的某些不必要的参数。表 10-2 列出了可用于建立通用控件的所有函数。

除了这些通用控件以外，图象列表和属性表也提供了它们自己的 API 函数，利用它们可以简便地建立这些类型的对象。ImageList_Create () 函数建立了一个新的图象列表，而 PropertySheet () 则可以对一个属性表进行装配。

表 10-2 用于建立通用控件的对应 API 函数

通用控件	API 函数	通用控件	API 函数
动画	Animate_Create ()	状态窗口	CreateStatusWindow ()
拖动列表框	CreateWindowEx ()	制表符	CreateWindowEx ()
标题	CreateWindowEx ()	工具栏	CreateWindowEx ()
热键	CreateWindowEx ()	工具提示	CreateWindowEx ()
列表视窗	CreateWindowEx ()	跟踪条	CreateWindowEx ()
进展条	CreateWindowEx ()	树形视窗	CreateWindowEx ()
多功能编辑框	CreateWindoeEx ()	上下控件	CreateUpDownControl ()

继续与标准控件进行比较，我们发现它们之间还存在一些区别。通用控件在列表内增加了编程项目（风格、消息和通知代码）、数据结构、一条新的通知消息、一些通用的窗口风格以及通知代码。因此，建立一个通用控件的时候，必须全面考虑下面所有这些元素：

▶常规通用控件的窗口风格
▶特定通用控件的窗口风格
▶通用控件消息
▶WM_NOTIFY 消息
▶常规通知代码
▶特定的通知代码
▶数据结构

可以预见，通用控件具有很高的复杂性，这主要是由于开发者必须对大量信息进行管理，否则便无法成功建立这种类型的窗口。

10.2 通用风格

对于 Win32 里的通用控件来说，有几种通用组件对于所有通用控件来说都是适用的。除了用于生成通用控件的 API 函数以外，还需要传递一个或者多个风格（WS_）。这些 WS_风格可以与不同类使用的特殊风格集成到一起使用，另外还可以和某些常规的通用控件风格（CCS_）联合使用。如表 10-3 所示，其中为大家列出了 COMMCTRL.H 里支持的所有 CCS_风格。

表 10-3 可应用于几乎所有控件种类的通用控件风格

风 格	值	说 明
CCS_TOP	0x00000001L	强迫控件把自己定位于父窗口客户区的顶部，与父窗口的宽度匹配
CCS_NOMOVEY	0x00000002L	禁止控件改变自己的尺寸以及垂直移动，与 WM_SIZE 消息对应
CCS_BOTTOM	0x00000003L	强迫控件把自己定位于父窗口客户区的底部，与父窗口的宽度匹配
CCS_NORESIZE	0x00000004L	控件在建立的时候就决定好了自己的位置和尺寸
CCS_NOPARENTALIGN	0x00000008L	控件保持自己的位置不变，不对父窗口尺寸的变化作出响应
CCS_ADJUSTABLE	0x00000020L	用于工具栏。假如双击一个工具栏控件，或者把按钮从工具栏上拖动下来，这种风格定义就会激活 Customize Toolbar（定制工具栏）对话框，令其显示出来
CCS_NODIVIDER	0x00000040L	禁止在控件顶部描绘一个两象素高的高亮度显示

考虑到这些风格带来的显示效果，它们主要用于标题控件、工具栏控件以及状态栏窗口。事实上，这三种通用控件既可以在父窗口客户区的顶部显示，也可以在底部显示，它们必须对父窗口内发生的任何变动作出迅速的响应。建立这样的一个通用控件时，大家也许会觉得有必要在 CreateWindowEx () 函数的第四个参数内增加一种或者多种这样的风格。就像下面这个摘录下来的代码段那样，它表明 WC_HEADER 是定义 SysHeader32 控件的另外一种途径：

```
...
hwndHeader = CreateWindowEx (0, WC_HEADER, NULL, CCS_TOP | WS_
```

```
VISIBLE | … , …);
        …
```

10.3 通知代码

对于所有通用控件来说，它们都是通过 WM_NOTIFY 消息与各自对应的父窗口进行通信的。WM_NOTIFY 是一种新设计的条目，它的作用是把信息流与 WM_COMMAND 消息已经提取出来的消息分隔开来。

WM_NOTIFY	0x
wParam	发送消息的那个控件的 ID
lParam	NMHDR 数据结构的地址

接收到一条 WM_NOTIFY 消息以后，父窗口马上就会知道这是由控件发出来的。随后，父窗口会对 NMHDR 数据结构进行检查，从中提取出某些有价值的信息，比如控件句柄以及通知代码等等。因此，开发者最好立即计算 lParam，从而得到一个 LPNMHDR 指针，如下所示：

```
…
case WM_NOTIFIY
{
    LPNMHDR pnmhdr = (LPNMHDR) lParam;

    switch (pnmhdr -> code)
    {
        …
    }
}
…
```

NMHDR 数据结构内包含了控件窗口句柄（hwndFrom）、通知代码（code）以及发送消息的那个控件的标识符（idFrom）。后面这种信息实际上是重复了由 WM_NOTIFY 内的 wParam 提供的信息。

```
typedef struct tagNMHDR
{
    HWND hwndFrom;
    UINT idFrom;
    UINT code;
} NMHDR;
```

```
typedef NMHDR * LPNMHDR;
```

考虑到通用控件的99.99%都是当作子窗口使用的，所以父窗口进程也提供了WM_COMMAND消息，用它来接收来自于标准控件的通知代码，WM_NOTIFY则用于新的通用控件。

NMHDR的code项既可以包含一个常规的通知代码，也可以包含特定于控件的通知代码。就目前来说，我们应该把注意力放在第一类通知代码身上，即表10-4所列的带有前缀NM_的通知代码。

表10-4 通用控件使用的常规通知代码

通知代码	值	说 明
NM_OUTOFMEMORY	(NM_FIRST-1)	因为没有足够的内存可用，所以操作无法完成
NM_CLICK	(NM_FIRST-2)	用户在某个控件上方单击了鼠标左键后产生
NM_DBLCLK	(NM_FIRST-3)	用户在某个控件上方双击了鼠标左键后产生
NM_RETURN	(NM_FIRST-4)	输入焦点位于某个控件上时，假如用户按下回车键，就会产生这条通知代码
NM_RCLICK	(NM_FIRST-5)	用户在某个控件上方单击了鼠标右键后产生
NM_RDBLCLK	(NM_FIRST-6)	用户在某个控件上方双击了鼠标右键后产生
NM_SETFOCUS	(NM_FIRST-7)	输入焦点转移到某个控件上时，就会接收到这条通知代码
NM_KILLFOCUS	(NM_FIRST-8)	某个控件失去了输入焦点时，就会接收到这条通知代码

由于采用了与普通操作（比如鼠标单击）相关的一系列常规通知代码，微软的工程师们极大地简化了这些窗口类的使用，从而减少了讨厌的重复劳动。我们可以注意到这样一个有趣的现实，那就是所有这些通知代码都是以WM_FIRST定义的值为基础的，这种定义是在WINUSER.H里设置的（0U－0U）。因此，NM_CLICK的值为“4294967296-2”，以此类推。

前面对WM_NOTIFY消息的实施只有在某些情况下才能正常实现。考虑到这些消息携带的是几个控件的通知代码，而这些控件又是位于同一个父窗口里的，所以我们直接以通知代码段内的转换代码块为基础，这样产生的危险最小。但是，要达到相同的目标，更通用的一种办法却需要首先判断由通知代码指定的控件，然后以实际接收到的通知代码为基础，再进行转换操作。因此，WM_NOTIFY消息块看起来应该像下面这个样子：

```
...
case WM_NOTIFIY
{
    LPNMHDR pnmhdr = (LPNMHDR) lParam;

    switch (pnmhdr -> idFrom)
    {
        case DL_TREE VIEW:
            switch (pnmhdr -> code)
```

```
                {
                    case NM_CLICK:
                    {
                        ...
                    }
                        break;
                    ...
                }
                break;
            ...
    }
    ...
```

在前一段摘录下来的代码里，应用程序捕获了来自一个树形视窗控件内的 WM_NOTIFY 消息，这个控件的 ID 是 DL_TREEVIEW，并对实际的通知代码（在例子里是 NM_CLICK）进行了检查。

正如我们以前提到的那样，通用控件并不一定非要当作子窗口使用。在第十四章“多线程、IPC 和 I/O”里，我们会提到一个名为 SYNCHRO 的示范程序，该程序为我们展示了如何建立一个弹出式列表视窗，这就是以这种预定义类为基础的一个典型的顶级窗口。并不是所有控件都能转换成弹出式窗口。比如，要建立属于弹出式窗口的一个进展指示器就毫无意义了。

10.4 通用控件探秘

新的通用控件是一系列预定义窗口类，它们对标准的控件进行了延展。这些通用控件特别设计用于支持和实现外壳的新外壳和新行为。在下面这些小节里，大家会学习如何建立它们，以及如何利用它们的高级功能。掌握这些知识以后，我们就可以毫不费力地建立起类似于系统外壳的 Win32 应用程序，并使它的行为也具有外壳的风格。

拖动列表框

拖动列表框是一种标准的 LISTBOX 窗口控件，只是其功能在 Win32 里得到了扩展。利用这个控件，用户可以选择一个列表框项目，然后在列表框内对其进行拖动。首先，我们必须建立一个标准的单一选择列表框，令其采用标准的 LBS_风格。要把普通的控件转换成一个拖动列表框，我们需要调用 MakeDragList（）函数，如下所示：

```
#include <commctrl.h>
BOOL MakeDragList (HWND hLB);
```

这个函数里唯一的参数就是一个单一选择列表框控件的句柄，这个列表框是以前通过 CreateWindowEx（）函数建立的。假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值。MakeDragList（）函数注册了一条新消息，用它来控制控件内进行的所有拖放操作。为了知道这条消息的值，应用程序必须调用一个 RegisterMessage（）函数，同时传递 DRAGLISTMSGSTRING 定义。这种行为要求开发者必须对父窗口进程的“切换”代

码块进行少许修改，以便能对拖动列表框消息进行跟踪。举个例子来说，假如我们需要把一个拖动列表框直接定位于应用程序主窗口客户区域内。调用 MakeDragList () 函数的一种简便途径是在 WM_CREATE 消息里进行。在离开这一部分代码之前，我们需要调用 RegisterMessage ()，用它取得与拖放操作有关的标识符的值，如下所示：

```
...
case WM_CREATE:
{
    ...
    uiDragListBox = RegisterWindowMessage (DRAGLISTMSGSTRING);
}
break;
...
```

uiDragListBox 标识符在整个窗口进程里都必须是“可见”的。所以开发者需要在切换代码块的外部对其进行声明，并为它分配静态存在类。

需要对应用程序窗口进程的常规结构进行的第二项修改是与切换块的缺省条件有关的。在这个时候，我们必须检查接收到的消息是否与修改过的列表框内的一次拖放操作有关。针对这种处理，我们只需要一个简单的 if 测试就足够了。通过测试，我们就能知道存储于 msg 标识符内的消息是否与以前由 RegisterMessage () 返回的值相等，如下所示：

```
...
default:
    // is it a drag and drop operation on the listbox ?
    if (msg ==uiDragListBox)
    {
        ...
```

假如拦截下来的消息确实与一次拖放操作相关，那么 wParam 内就包含了进行这种拖放操作的那个列表框的 ID。这样一来，我们就有机会在相同的代码段内包含几个拖动列表框。以 wParam 为基础的一个切换代码块正是我们需要的：

```
...
switch (wParam)
{
    case CT_DRAGLBOX:
    {
        LPDRAGLISTINFO pDragListInfo = (LPDRAGLISTINFO) lParam;
        static int iItem;
```

```
static int iDraggingSpot;
...
```

在 lParam 里包含了一个 DRAGLISTINFO 数据结构的地址，其中填写了执行拖放操作需要的所有信息。uNotification 项内包含了表 10-5 列出来的其中一种 DL_定义，它的作用是指示应用程序拖放操作当前的状态。

```
typedef struct
{
    UINT uNotification;
    HWND hWnd;
    POINT ptCursor;
} DRAGLISTINFO, *LPDRAGLISTINFO;
```

DRAGLISTINFO 数据结构里的另外两项是指列表框的窗口句柄，以及用屏幕座标系统表示的鼠标位置。通知代码的值可以是表 10-5 内列出的任何一个定义，这种值反映了四种状态：开始拖动、移动鼠标、松开鼠标左键以及退出拖动操作。

表 10-5　用于拖动列表框的拖放通知代码

拖放通知代码	值	说　明
DL_BEGINDRAG	(WM_USER+133)	用户在一个列表框项目上方单击鼠标左键，从而选定了这个项目以后发出
DL_DRAGGING	(WM_USER+134)	拖动操作开始以后发出；用于判断鼠标指针当前在屏幕上的位置
DL_DROPPED	(WM_USER+135)	通知拖动操作已经结束
DL_CANCELDRAG	(WM_USER+136)	用户按下 Esc 键或者单击鼠标右键，从而中断当前拖动操作的时候发出

拖动操作开始以后，应用程序就会检查选中的是哪个项目。在这个时候，DL_BEGINDRAG 通知值为我们提供了机会，利用它可以对列表框进行查询。API 函数 LBItemFromPt () 会把 DRAGLISTINFO 内 ptCursor 项的屏幕坐标点转换成对应的项目编号。

```
#include <commctrl.h>
int LBItemFromPt (HWND hLB, POINT pt, BOOL bAutoScroll);
```

参数	说明
HWND hLB	列表框控件
POINT pt	准备检查的屏幕坐标点
BOOL bAutoScroll	假如为 FALSE，就在屏幕点位于列表框的下方或者上方时禁止列表框的自动滚动。这种情况下的返回值为-1

返回值	在正文里讨论
int	列表框项目标识符；假如当前的位置位于列表框以外，则为-1

用户并不一定要在拖动列表框内对所有的项目进行拖动。应用程序可以对不同的项目区别对待，只允许其中的一些进行拖动。要达到这样的目标，只需在处理 DL_BEGINDRAG 通知代码的时候简单地返回 TRUE 或者 FALSE 就可以了。拦截 DL_DRAGGING 通知消息的时候，就会重复进行光标在列表框内的当前位置到对应项目的转变。这样就可以准确地知道光标指针正指在列表框内哪一个项目的上方。除此以外，我们还要调用 DrawInsert () 函数，从而对鼠标指针进行修改，从而提供可视的拖动反馈信息。

```
#include <commctrl.h>
void DrawInsert (HWND hwndParent, HWND hLB, int nItem);
```

参数	**说明**
HWND hwndParent	列表框父窗口的句柄
HWND hLB	列表框句柄
int nItem	准备描绘的图标项的标识符
返回值	在正文里讨论
void	没有返回值

第一个参数是列表框父窗口的句柄，后面跟随的是列表框句柄。这个函数的语法是通过表 10-6内列出的某一种定义来完成的。

表 10-6　与列表框内拖放操作有关的一些图标形状和对应的定义

图标形状	值	说　明
DL_CURSORSET	0	拖动一个项目的时候形状不变化
DL_STOPCURSOR	1	停止光标图标
DL_COPYCURSOR	2	拷贝光标图标
DL_MOVECURSOR	3	移动光标图标

下面这个代码段向我们展示了对 DL_DRAGGING 通知代码进行拦截的标准方式：

```
...
case DL_DRAGGING:
{
    iDraggingSpot = LBItemFromPt (hwndDLB, pDragListInfo -> ptCursor,
        TRUE);
    DrawInsert (hwnd, hwndDLB, DL_MOVECURSOR);
}
    break;
...
```

第三个也是最后一个通知代码是DL_DROPPED，它与具体的拖放操作有关。假如用户释放了鼠标按钮，列表框父窗口进程就会接收到这个代码。这就意味着即使对列表框的一个简单的选定都会由系统解释成DL_BEGINDRAG和DL_DROPPED的一种组合。我们必须对这种行为引起足够的重视，因为对DL_DROPPED一次不正确的处理也许会与列表框对一个选定项目的标准视觉效果处理发生冲突。在本书附带CD的Listing 10.1内，大家可以找到一个名为DRAGLIST的示范程序，这个应用程序可以跟踪与选定项目有关的位置。对DL_DROPPED进行处理的时候，这个值会与拖动项目落下的位置进行比较。再一次地，当前的鼠标指针位置会转换成对应的列表框项目标识符，从而获得这种信息。假如两个值完全一致，对DL_DROPPED的处理就会中断，不会针对它执行任何附加的语句。

为了使程序有效，拖动操作必须在列表框内部发生，同时限制拖动操作在垂直方向的移动，将其限制在正文项目实际占据的显示空间以内。然而，我们没有办法防止用户移动鼠标指针，甚至把指针移到列表框的上下边界以外我们也管不着。在这种情况下，LBItemFromPt()会返回一个－1值，用它指出当前位置不能与项目的标识符匹配。这个值让人相当迷惑，因为-1意味着对列表框进行处理的时候碰到了某种不同的情况。假如希望当前插入项目的最底部插入一个项目，我们只需发送一条LB_INSERTSTRING消息即可，同时传递-1值，把它当作列表框内的插入点使用。因此，我们必须先来做一番准备工作，否则无法真正理解光标的位置到底在列表框内当前插入项目的上方还是下方。

为了达到我们刚才指出的目标，首先需要调用ListBox_GetCount()宏函数，用它可以方便地判断列表框内的项目总数。这个宏会返回项目总数信息，假如用户把鼠标指针移动到最后一个项目的下方，这种信息就显得相当有用了。除此以外，假如起始项目存储于目标位置以上的某个位置处，那么目标位置就必须减一，从而把插入事件发生时移走的那个项目考虑进去。拖动项目是由iItem标识符标识的，而落下位置则是由iDropPlace标识符指定的。

真正执行拖动操作之前，我们需要考虑的最后一种情况是把某个项目插入列表框的顶部。考虑到每次当光标位置位于项目列表的外部时（无论在上面还是下面），LBItmFromPt()就会返回值－1，所以必须实现一种算法，用以判断用户是不是想把拖动的项目放置到列表的顶部。我们在这儿的考虑是计算鼠标指针在列表坐标系统里的垂直位置。假如为负值，就表明由LBItemFromPt()返回的－1值引用的是列表框的顶部。在DL_DROPPED代码块内，随后就需要用GetCursorPos()判断鼠标指针当前的位置。然后马上把它转换到列表框坐标系统里，这一工作是通过ScreenToClient()函数来实现的。假如iDropPlace等于－1，而且鼠标指针在Y轴上的当前位置为负数，那么只需要简单地把iDropPlace设置成0，这就是第一个列表框项目的标识符。

```
...
case DL_DROPPED:
{
    int iDropPlace, iTotItems;
    char szText[50];
    POINT pt;
```

```
        // determine the drop location
        iDropPlace = LBItemFromPt(hwndDLB, pDragListInfo -> ptCursor,
            FALSE);
        // skip if th user simply selected an item
        if(iItem == iDropPlace)
            break;
        // retrieve the text of the dragged item
        ListBox_GetText(hwndDLB, iItem, szText);
        // determine the relative position
        if(iItem < iDropPlace)
            iDropPlace--;

        // where is the mouse at the end of the dragging?
        GetCursorPos(&pt);
        // convert the point in client coordinates
        ScreenToClient(hwndDLB, &pt);
        // are we dragging above the listbox?
        if(iDropPlace == -1 && pt.y < 0)
            iDropPlace = 0;

        // get the total item count
        iTotItems = ListBox_GetCount(hwndDLB);
        // remove the string from its original position
        ListBox_DeleteString(hwndDLB, iItem);
        // if the new position is at the bottom of the listbox change it
        if(iDropPlace ==-1)
            iDropPlace = iTotItems - 1;
        // insert the string in the new location
        ListBox_InsertString(hwndDLB, iDropPlace, szText);
        // select the moved item
        ListBox_SetCurSel(hwndDLB, iDropPlace);
    }
        return TRUE;
    ...
```

需要采取的最后一个行动是实际移动项目，并且把它插入新位置处。这种操作通过调用 ListBox_DeleteString () 和 ListBox_InsertString () 这两个宏函数很容易就可以完成。注意分别为这两个函数的 iItem 和 iDropPlace 提供恰当的值。图 10-1 向大家展示了 DRAGLIST 程序的运行情况，它的客户区完全被一个列表框占据了。列出来的项目可以用鼠标左键进行

拖动处理。

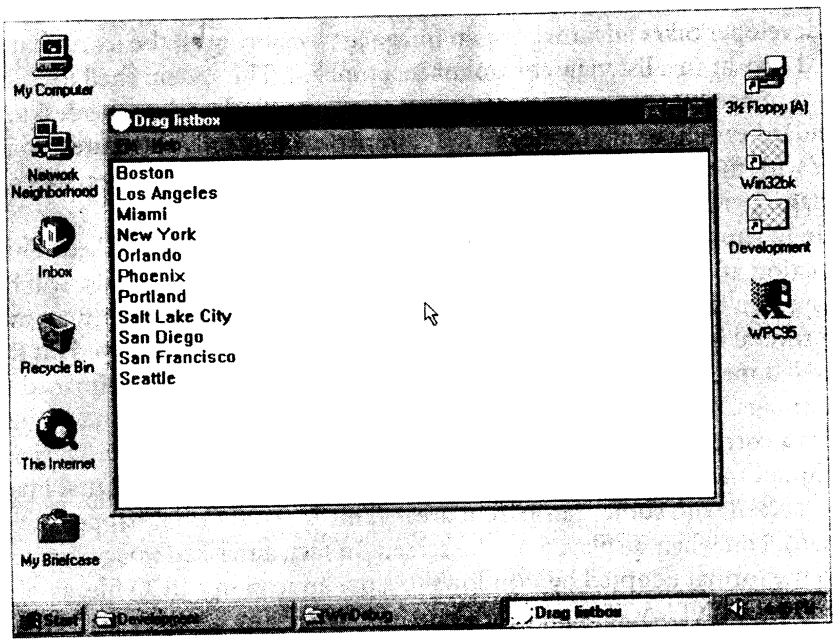


图 10-1　DRAGLIST 程序显示了如何实现一个拖动列表框

从技术上说，没有什么能够限制我们实现一个物主绘图列表框，只需要把 DRAGLIST 程序里提供的代码与第九章“预定义的窗口类”里的那个 EXTSEL 例子合并起来就可以了。

10.5　图象列表

图象列表是一种 32 位对象，设计它的宗旨是简化图标和位图在 Win32 应用程序里的运用。这是一种单独的位图屏幕设备格式，由几个尺寸相等的图标或者位图组成。我们可以把图象列表想象成一个长标题，它被分割成了尺寸相同的一些更小的部分，每个部分都能通过一个位置 ID 进行访问。位图的最大尺寸可以有 32K×32K 象素那么大。一个图象列表里的标准图标的最大数量可以用单独一幅图象的尺寸乘以图象数目，从而很容易地计算出来。

在当前的 Windows 版本里，GDI 对象会消耗大量的内存和系统资源，这样就对 Windows 95 系统的可靠性和有效性带来了潜在的影响。考虑到诸如 Windows 95 的图形环境将出现越来越多的图象，所以微软公司决定引入图象列表的概念，把它当作一种新的 Win32 工具，以此提供更好和更简便的方式对图象进行控制。

一名开发者可以把那些应该在列表视窗控件或者工具栏里出现的所有图标存储到一个图象列表内，从而发掘出这种通用控件的潜力。系统外壳本身有自己的图象列表，其中包容了在标准系统配置中显示于桌面的几个图标。因此，图象列表可以通过一系列特定的消息、风格以及标志实现自己与其他通用控件的集成以及交互作用。

Win32 支持两种形式的图象列表：屏蔽和非屏蔽图象列表。这种区别会影响到包含于图象列表内的图象是否在屏幕上显示出来。非屏蔽图象列表是一种单一的彩色位图，它由几个

图标或者位图组成。这些图象中的一个在屏幕上显示出来的时候，它就会按照自己当初描绘出来的样子显示。屏蔽图象列表是由两幅位图组成的。第一幅位图包含了实际的彩色图象。另外一幅则是相应的单色图象。彩色和单色图象的组合在屏幕上生成了实际显示出来的图象，这种图象带有一些透明区域。这其实就是一个标准图标在屏幕上显示时发生的情况。事实上，屏蔽图象列表类似于 Windows 采纳的图标存储格式，这种图标是存储在一个 .ICO 文件里的，正如我们在第 9 章“预定义的窗口类”里学习到的那个 EXTRACT 示范程序那样。

无论希望建立什么类型的图象列表，我们都需要用到 ImageList_Create () 这个函数来达到自己的目的。

```
#include <commctrl, h>
WINCOMMCTRLAPI HIMAGELIST WINAPI ImageList_Create (int cx,
                                                   int cy,
                                                   UINT flags,
                                                   int cInitial,
                                                   int cGrow);
```

参数	说明
int cx	以象素为单位指定每幅图象的宽度
int cy	以象素为单位指定每幅图象的高度
UINT flags	表 10-7 所列的一个 ILC_标志
int cInitial	最初包含于图象列表内的图象的数量
int cGrow	允许增加到图象列表内的附加图象的数量
返回值	在正文里讨论
HIMAGELIST	图象列表的句柄，假如建立失败，则返回一个 NULL 值

建立一个图象列表的时候，我们必须指定其中包含的对象的尺寸。这两个值分别对应于 ImageList_Create () 函数的第一和第二个参数，即图象的宽度和高度。第三个参数是一个 ILC_标志，它的作用是定义新图象列表的类型。如表 10-7 所示，我们可以在屏蔽图象列表到 DIB 位图集合这个范围之间作出选择。

表 10-7 在 ImageList_Create () 函数里用于定义图象列表类型的标志

标 志	值	说 明
ILC_MASK	0x0001	建立一个屏蔽图象列表
ILC_COLOR	0x0000	定义一个彩色图象列表
ILC_COLORDDB	0x00FE	使用与设备无关位图
ILC_COLOR4	0x0004	建立一个图象列表，其中只包含 4 字节与设备无关位图 (DIB) 部分
ILC_COLOR8	0x0008	建立一个图象列表，其中只包含 8 字节与设备无关位图 (DIB) 部分
ILC_COLOR16	0x0010	建立一个图象列表，其中只包含 16 字节与设备无关位图 (DIB) 部分
ILC_COLOR24	0x0018	建立一个图象列表，其中只包含 24 字节与设备无关位图 (DIB) 部分
ILC_COLOR32	0x0020	建立一个图象列表，其中只包含 32 字节与设备无关位图 (DIB) 部分
ILC_PALETTE	0x0800	为图象列表使用彩色模板

cInitial 项指出最初在图象列表内包含的图象数目。这个数这可以在执行过程中加以扩展，扩展的依据是第五个（最后那个）参数 cGrow 指定的附加图象数量。从本质上说，我们既可以在刚开始的时候建立一个空的图象列表，然后在以后用可用的图象填充；也可以一开始就让它包含一定数量的图象。除此以外，根据应用程序的需要，包含于图象列表内的初始图象数目可以在以后动态地发生改变。最后这种方案是最灵活的；当然，这要求一开始就为 cGrow 参数分配一个足够大的数字。

假如图象列表建立成功，ImageList_Create（）函数就会返回针对那个图象列表的句柄，这是一种属于 HIMAGELIST 类型的对象。通过对这个句柄的访问，我们可以方便地增删图象或者把整个图象列表与一个通用控件关联起来。假如不再需要图象列表，可以通过指定相应的句柄来删除它。为了实现这一要求，我们可以调用 ImageList_Destroy（）函数，如下所示：

```
#include 〈commctrl. h〉
WINCOMMCTRLAPI BOOL WINAPI ImageList_Destroy (HIMAGELIST himl);
```

如表 10-8 所示，其中列出了针对图象列表的建立、管理、访问以及删除而专门设计的所有函数。

表 10-8　图象列表的完整清单

图象列表函数	说　明
ImageList_Add	在图象列表内增加一幅位图（假如是一个屏蔽图象列表，则增加一幅屏蔽位图）
ImageList_AddMasked	在图象列表内增加一幅或者多幅位图，并且根据独立参数内指定的颜色，从而自动建立相应的屏蔽图象
ImageList_BeginDrag	启动涉及到图象拖动的操作
ImageList_Create	建立一个新的图象列表
ImageList_Destroy	消除一个图象列表
ImageList_DragEnter	在一个图象列表内描绘已消除的图象
ImageList_DragLeave	在拖动一幅图象的时候中断绘图操作
ImageList_DragMove	在图象列表内描绘已消除的图象，从而与 WM_MOUSEMOVE 消息响应
ImageList_DragShowNolock	显示或者隐藏正在拖动的图象
ImageList_Draw	在目标设备现场内描绘一幅重叠的屏蔽图象
ImageList_DrawEx	在目标设备现场内描绘一幅重叠的屏蔽图象
ImageList_EndDrag	中止一次拖动操作
ImageList_GetBkColor	取得图象列表的背景颜色
ImageList_GetDragImage	返回由系统建立的临时图象列表的句柄，用于控制与图象列表内一幅拖动图象有关的所有内容
ImageList_GetIcon	以图象列表内的一幅图象和屏蔽图象为基础，建立一个图标或者光标
ImageList_GetIconSize	返回图象列表内一幅图象的尺寸
ImageList_GetImageCount	返回图象列表内的图象数目
ImageList_GetImageInfo	在一个 IMAGEINFO 数据结构里填写关于图象列表内所存图象的信息
ImageList_LoadImage	根据指定的位图、光标或者图标资源，从而建立一个崭新的图象列表
ImageList_Merge	建立一个新的图象列表，其中包含的图象是由两幅现成图象合并形成的
ImageList_Read	从一个数据流里读取一个图象列表
ImageList_Remove	从图象列表内删除一幅图象
ImageList_Replace	用图象列表内的一幅新图象取代一幅现成图象
ImageList_ReplaceIcon	用一个图标取代一幅图象
ImageList_SetBkColor	设置图象列表的背景颜色
ImageList_SetDragCursorImage	把拖动图象与一个光标图标组合到一起，从而建立一幅新图象
ImageList_SetIconSize	腾空一个图象列表，然后为以后将插入其中的图象设置新的尺寸
ImageList_SetOverlayImage	把一幅图象的索引增加到重叠屏蔽图象列表内
ImageList_Write	把一个图象列表编写成数据流

除了这些函数以外，COMCTRL.H 内还包含了四个定义，这些定义对函数集进行了更为深入的拓展，如表 10-9 所示。

表 10-9 COMMCTRL.H 里定义的宏函数简化了对图象列表的访问

图象列表宏	图象列表函数
ImageList_AddIcon (himl, hicon)	ImageList_ReplaceIcon (himl, -1, hicon)
ImageList_RemoveAll (himl)	ImageList_Remove (himl, -1)
ImageList_ExtractIcon (hi, himl, i)	ImageList_GetIcon (himl, i, 0)
ImageList_LoadBitmap (hi, lpbmp, cx, cGrow, crMask)	ImageList_LoadImage (hi, lpbmp, cx, cGrow, crMask, IMAGE_BITMAP, 0)

通过观察表 10-8 和表 10-9，大家对图象列表的功用以及如何进行管理应该有一个全面的印象了。建立好一个图象列表以后，我们通常都要在其中填充图象。例如，假设我们现在希望在一个图象列表里存储一系列图标。为了完成这一任务，我们可以调用 ImageList_AddIcon () 宏函数，该函数要求用到图象列表的句柄以及相应的图标。返回值是进入图象列表内的图标的索引编号，假如返回值为-1，则表明操作失败：

```
...
int iPos, i;
HICON hicon;
...
himg = ImageList_Create (32, 32, ILC_COLOR, MAXICONS, 0);
...
for (i = 0; i < MAXICONS; i++)
{
    // load the icon
    hicon = LoadIcon (hInstance, MAKEINTRESOURCE (i));
    // add the icon
    iPos = ImageList_AddIcon (himg, hicon);
    // delete the icon handle
    DeleteObject (hicon);
    // check if the isertion succeeded
    if (iPos == -1)
    {
        ...
        return FALSE;
    }
}
...
```

在前面这个代码段里，我们建立了一个图象列表，用它存储最多 MAXICONS 个对象。图象列表里包含的是 32×32 象素的彩色图标，最后生成的将是一个 32 象素高，32×20 象素宽的一个“大标题”。iPos 标识符把新的图标索引临时性地存储在图象列表里。索引编号是从零开始的，假如为-1，则表明在插入过程中发生了错误。对出错条件进行测试之前，我们最好先用 DeleteObject () 消除图标句柄。以前载入的图标与图象列表没有一点关系，这个图象列表的每幅图象都是存储于一幅单一的位置里的。

对于屏蔽图象列表来说，我们就必须提供两幅图象，一幅用于主图象显示，另一幅则用于屏蔽图象显示。ImageList_Add () 函数的作用是把图象插入两个对应的图象列表内：

```
#include <commctrl.h>
int WINAPI ImageList_Add (HIMAGELIST himl,
                          HBITMAP hbmImage,
                          HBITMAP hbmMask);
```

参数	说明
HIMAGELIST himl	图象列表句柄
HBITMAP hbmImage	位图句柄
HBITMAP hbmMask	屏蔽图象的句柄
返回值	在正文里讨论
int	新图象的句柄；假如为-1，则表明函数调用失败了

作为另外一种选择，我们也可以要求图象列表生成屏蔽图象，这需要在指定用于对位图图象进行屏蔽的颜色：

```
#include <commctrl.h>
int WINAPI ImageList_AdMasked (HIMAGELIST himl,
                               HBITMAP hbmImage,
                               COLORREF crMask);
```

参数	说明
HIMAGELIST himl	图象列表句柄
HBITMAP hbmImage	位图句柄
COLORREF crMask	把位图内指定颜色的象素改变成黑色，把屏蔽位图内对应的二进制设置成 1
返回值	在正文里讨论
int	新图象的句柄，假如函数调用失败，则返回一个-1 值

第三种也是最快的一种方案是通过 ImageList_LoadImage () 函数建立一个图象列表。这个 API 并不只是载入一幅图象（一个图标、光标或者位图），而且还会根据那幅图象的信息，从而自动建立一个崭新的图象列表。

```
#include <commctrl.h>
HIMAGELIST WINAPI ImageList_LoadImage (HINSTANCE hInstance,
                                       LPCSTR lpbmp,
                                       int cx,
                                       int cGrow,
                                       COLORREF crMask,
                                       UINT uType,
                                       UINT uFlags);
```

参数	说明
HINSTANCE hInstance	应用程序实例句柄
LPCSTR lpbmp	一个正文中，其中包含了准备载入的那幅图象的名字
int cx	图象列表内每幅图象的宽度
int cGrow	可以增添到图象列表内的附加图象的数目
COLORREF crMask	用于生成屏蔽图象的颜色；也可以为 CLR_NONE，表示建立一个非屏蔽图象列表。载入图象内带有这种颜色的每个象素都会变成黑色，而屏蔽图象内对应的二进制位则会设置成-1
UINT uType	图象的类型。可以在下面这三个值中选用一个：IMAGE_BITMAP，IMAGE_CURSOR 或者 IMAGE_ICON
UINT uFlags	表 10-10 内列出的一种 LR_定义
返回值	在正文里讨论
HIMAGELIST	图象列表的句柄；假如函数调用失败，则返回一个 NULL 值

表 10-10 列出了用于这个函数的所有标志，这些标志在 LoadImage () 提供的功能基本上都增加了由 ImageList_Create () 以及 ImageList_Add () 提供的功能。

就我个人来说，我比较喜欢调用 ImageList_Create () 来建立一个图象列表，以后再根据程序的需要增加一些图象，这样显得对整个程序的控制很有条理。另外，这样做还可以很容易地把两种行动分隔开，然后把它们放置到明显不同的两个代码段里。

表 10-10 与 ImageList_LoadImage () 联合使用的 LR_定义

LR_定义	说 明
LR_DEFAULTCOLOR	使用显示器的颜色格式
LR_LOADDEFAULTSIZE	忽略 cx 参数里设置的值，从而判断出图象的宽度
LR_LOADFROMFILE	lpbmp 串内包含的是一个文件名，而不是资源标签
LR_LOADMAP3DCOLORS	在图象使用的颜色表内查找下面这些灰度值，并用对应的三维颜色替换：DK Gray，RGB（128，128，128）COLOR_3DSHADOW Gray，RGB（192，192，192）COLOR_3DFACE Lt Gray，RGB（223，223，223）COLOR_3DLIGHT
LR_LOADTRANSPARENT	用图象颜色表内的 COLOR_WINDOW 颜色替换图象内第一个象素的颜色
LR_MONOCHROME	用黑白两色转换图象
LR_SHARED	假如图象多次载入，则对图象句柄进行共享

10.5.1 图象列表的管理

有几个函数可以帮助我们从一个图象列表内提取出有用的信息。ImageList_GetImageCount () 可以返回已经插入的图象数目，而 ImageList_GetIconSize () 则提供了作为图象列表基本组件的那些图象的信息。尽管该函数的名字中有一个 Icon（图标）字样，但事实上它能对任何种类的图象列表进行处理。最有趣的也许要算 ImageList_GetImageInfo () 函数，它的作用是在一个 IMAGEINFO 数据结构里填写图象句柄、它的屏蔽图象（假如有的话）以及图象的圆角矩形。

```
typedef struct _IMAGEINFO
{
    HBITMAP hbmImage;
    HBITMAP hbmMask;
    int Unused1;
    int Unused2;
    RECT rcImage;
} IMAGEINFO;
```

把一幅图象转换成相应的图标是一种相当直接的操作，我们在这时需要用到 ImageList_ExtractIcon () 函数，这是以更复杂的 ImageList_GetIcon () 为基础的一个宏函数：

```
#include <commctrl.h>
HICON WINAPI ImageList_GetIcon (HIMAGELIST himg,
                                int i,
                                UINT flags);
```

参数	说明
HIMAGELIST himg	图象列表句柄
int i	图象列表索引
UINT flags	表 10-11 列出的某个标志
返回值	在正文里讨论
HICON	图标句柄，假如函数调用失败，则返回一个 FALSE 值

使用表 10-11 内列出的某个恰当的标志，就可以生成一个新图标，并使其与初始图象具有不同的外观——这是包含了混合效果所致。ImageList_ExtractIcon () 内有助于简化我们进行的提取操作，只需用到应用程序的实例句柄、图象列表句柄以及图象索引即可。

10.5.2 图象列表和拖放

在表 10-8 里，我们列出了几个看起来和拖动操作有关的函数。对于一个图象列表来说，它并不仅仅提供了对图象集合进行管理的工具。它还提供了一种简便和有效的方式来实现图象（光标、图标或者位图等）在屏幕上的拖动，同时不会造成任何传统的负作用，比如闪烁等。在本书附带 CD 的 Listing 10.2 内，大家可以找到一个名为 DRAGIL 的示范程序。这个程序

向大家阐示了如何利用图象列表提供的服务来达到这种效果。

表 10-11 用于 ImageList_Draw ()，ImageList_DrawEx () 以及 ImageList_GetIcon 函数的定义

标 志	值	说 明
ILD_NORMAL	0x0000	用图象列表的背景颜色描绘图象。假如背景颜色设置成 CLR_NONE 值，图象就会用屏蔽色画成透明的
ILD_TRANSPARENT	0x0001	针对一个屏蔽图象列表，忽略其背景色，用屏蔽色描绘透明图象
ILD_MASK	0x0010	描绘屏蔽图象
ILD_IMAGE	0x0020	描绘图象
ILD_BLEND25	0x0002	针对一个屏蔽图象列表描绘图象，同时混合 25%的屏幕高亮颜色
ILD_BLEND50	0x0004	针对一个屏蔽图象列表描绘图象，同时混合 50%的屏幕高亮颜色
ILD_OVERLAYMASK	0x0F00	用于重叠屏蔽图象的标志值
ILD_SELECTED	ILD_BLEND50	等效于 ILD_BLEND50
ILD_FOCUS	ILD_BLEND25	等效于 ILD_BLEND25
ILD_BLEND	ILD_BLEND50	等效于 ILD_BLEND50

DRAGIL 是一个简单的 Win32 应用程序，它在自己客户区的左上角显示了一个标枪形状的图标，如图 10-2 所示。那个图标是在拦截 WM_CREATE 消息的时候载入的，并且和以往一样需要通过 WM_PAINT 显示出来。

下面这个代码段向大家展示了建立一个屏蔽图象列表所需的各个步骤。其间需要涉及到把它的句柄存储到主窗口预约内存区域里、载入图标以及最后把它添加到图象列表里。最后一项操作则是通过 DeleteObject () 函数删除那个图标。

```
...
// load the common control DLL
InitCommonControls();

// create an image list
himg = ImageList_Create(SYS(SM_CXICON), SYS(SM_CYICON),
    ILC_MASK, 1, 0);
// store the imagelist handle in the GWL_USERDATA area
SetWindowLong(hwnd, GWL_USERDATA, (long)himg);

// load one icon & insert it
hicon = LoadIcon(hInstance, "dragil");
if(ImageList_AddIcon(himg, hicon) == -1)
        MessageBeep(0);

// delete the icon handle
DeleteObject(hicon);
...
```

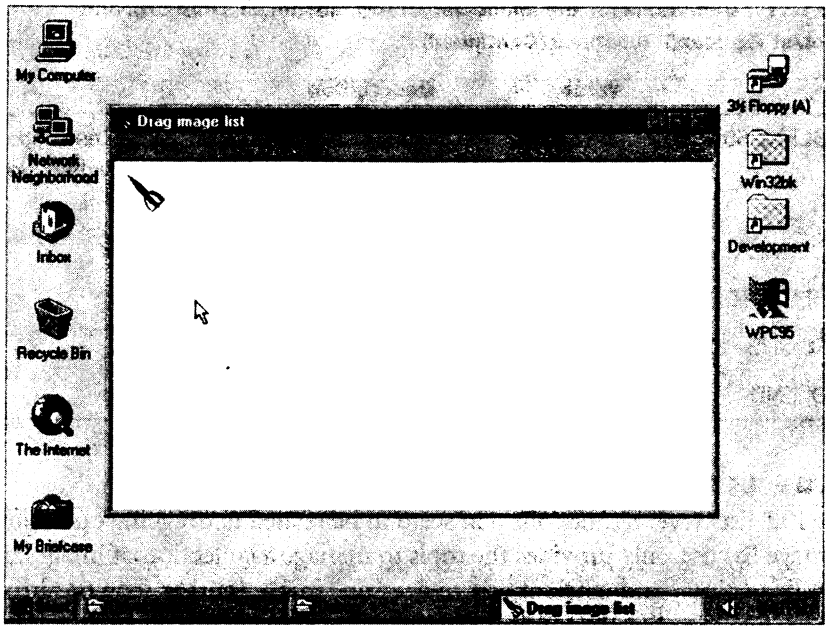


图 10-2　在拖动客户区内显示的那个图标之前，DRAGIL 程序的显示情况

尽管已经建立了一个屏蔽图象列表，最后仍然需要调用 ImageList_AddIcon () 函数，用它插入应该在其中包含的图象，因为这种类型的对象在缺省情况下是屏蔽的。应用程序声明了一个 POINT 结构，并赋予其静态存储类。这个数据结构的作用是永久性地记住图标在客户区内的当前位置。拦截 WM_PAINT 消息的时候，需要用这种位置信息在屏幕上显示图标。在那个时候，由 LoadIcon () 返回的图标句柄已经不再可用了。因此，必须调用 ImageList_Draw () 函数把它显示出来。

```
...
hdc = BeginPaint(hwnd, &ps);
// draw the image list
ImageList_Draw(himg, 0, hdc, ptIcon.x, ptIcon.y, ILD_TRANSPARENT);
EndPaint(hwnd, &ps);
...
```

ILD_TRANSPARENT 标志可以确保图标在客户区内透明地画出。迄今为止，在 DRAGIL 这个示范程序里涉及到的所有图象列表 API 完成的都是一些基本的操作，比如建立一个属于这个种类的新对象、增加一幅图象以及显示从列表内取得的一幅图象等等。当用户按下鼠标左键以后，就会生成一条相应的 WM_LBUTTONDOWN 消息，这条消息会直接进入应用程序的窗口进程。在这个时候，我们需要对当前的鼠标指针位置进行检查，核对它是否与图象矩形内的某个点对应，这一工作是通过 PtInRect () 来完成的。在这以后，应用程序就会测量鼠标指针当前位置到图标左上角的距离，这种距离要分别用 X 轴和 Y 轴上的象

素距离来度量。拖动操作开始以后，这两个值就显得很有用了。在这个时候，我们应该调用两个图象列表函数，它们分别是 ImageList_BeginDrag () 和 ImageList_DragEnter ()。这两个函数的宗旨都是为不久将至的拖动操作作好准备。ImageList_BeginDrag () 要求使用的参数包括图象列表句柄、图象列表内的图标位置索引以及图标的热点，这个点与鼠标指针的位置是对应的。最后的信息是从图标左上角到光标位置的偏移距离。

```
#include <commctrl.h>
BOOL WINAPI ImageList_BeginDrag (HIMAGELIST himlTrack,
                                 int iTrack,
                                 int dxHotspot,
                                 int dyHotspot);
```

参数	说明
HIMAGELIST himlTrack	图象列表句柄
int iTrack	准备拖动的图象的索引
int dxHotspot	从鼠标指针到图标左上角在 X 轴上的距离
int dyHotspot	从鼠标指针到图标左上角在 Y 轴上的距离
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

鼠标指针到图标左上角的距离通过 ImageList_BeginDrag () 函数很容易就可以计算出来，并且可以通过这个函数进行传递。ImageList_DragEnter () 函数通过向拖动过程中不会在视觉效果上进行更新的窗口发出句柄请求，以及向图象的显示位置发出请求，从而完成拖动的准备工作。

```
#include <commctrl.h>
BOOL WINAPI ImageList_DragEnter (HWND hwndLock,
                                 int x,
                                 int y);
```

参数	说明
HWND hwndLock	在拖动过程中不会重画的窗口句柄
int x	显示图象时在 X 轴上的位置。这个值是用窗口坐标系统表达的，而不是用客户区坐标表达的
int y	显示图象时在 Y 轴上的位置。这个值是用窗口坐标系统表达的，而不是用客户区坐标表达的
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

第一个参数是窗口的句柄，通常是指物理性容纳拖动图象的那个窗口。假如把ImageList_DragEnter () 函数内的 hwndLock 参数设置成 NULL，就表明自己希望把整个屏幕区域定义成拖动表面。这种特性在某些情况下显得相当有用的。例如，假如希望一次拖动操作能够在同一应用程序的几个窗口之间进行，这个选项就特别有用了。另外，假如拖动的目标窗口是系统外壳，就必须使用 OLE API。

最初的图象位置是由第二和第三个参数定义的。这儿需要注意的一个不寻常的情况是坐标系统。起点并不是客户区的左上角，而是窗口的左上角。这就意味着我们必须人工计算出非客户区组件的尺寸，这些组件包括标题栏、菜单栏以及一个可能存在的工具栏等等。以后在处理 WM_LBUTTONDOWN 消息的时候，要在 lParam 内存放的当前光标位置里减去以前人工计算出来的那种非客户区组件尺寸。相同的规范也适用于其他所有图象拖动函数（假设这些函数都利用了坐标作为自己的参数）。假如把用于容纳拖动图象的窗口的句柄设置成 NULL，就必须用屏幕坐标系统表达每个位置，从而就避开了对窗口坐标空间的计算。

下面这个代码段向大家展示了处理 WM_LBUTTONDOWN 消息时需要采取的操作：

```
...
// determine the current mouse position
pt.x = MAKEPOINTS(lParam).x;
pt.y = MAKEPOINTS(lParam).y;
// determine the icon rectangle
rc.left = rc.right = ptIcon.x;
rc.top = rc.bottom = ptIcon.y;
rc.right += SYS(SM_CXICON);
rc.bottom += SYS(SM_CYICON);
// evaluate if we clicked on the icon
if(! PtInRect(&rc, pt))
        break;

// calculate the offset from the upper left corner
cx = pt.x - ptIcon.x;
cy = pt.y - ptIcon.y;

// get the imagelist handle
himg = (HIMAGELIST)GetWindowLong(hwnd, GWL_USERDATA);

// convert the cursor position in screen coordinates
ClientToScreen(hwnd, &pt);
// define the icon attributes
ImageList_BeginDrag(himg, 0, cx, cy);
// enter dragging mode
```

```
ImageList_DragEnter(NULL, pt.x, pt.y);
// set the drag flag to TRUE
fDrag = TRUE;

// capture the mouse to drag the icon everywhere on the screen
SetCapture(hwnd);
...
```

假如把 fDrag 这个布尔值标识符设置成 TRUE，就表明已经启动了一次拖动操作。除此以外，应用程序还会对鼠标事件进行捕获，从而即使当鼠标指针位于客户区以外时，也能接收到 WM_MOUSEMOVE 消息。假如从属窗口就是客户区，那么就没有必要使用 SetCapture ()。图 10-3 向大家展示了刚刚发生拖动操作时，DRAGIL 程序的显示情况。

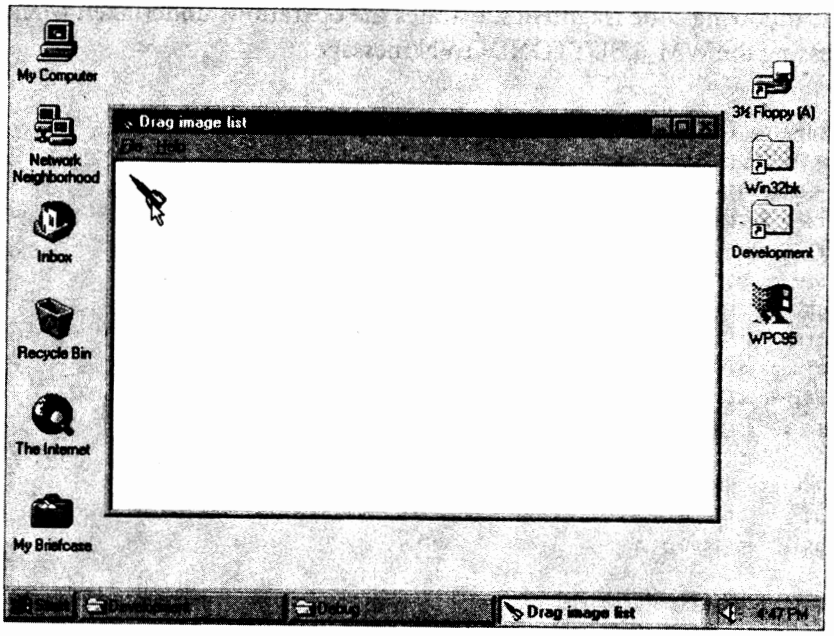


图 10-3　鼠标光标和拖动图象仍然在最初的位置，现在已经做好了开始拖动操作的一切准备

用户在桌面移动鼠标指针的时候，会生成一系列的 WM_MOUSEMOVE 调用，这些调用都定址于主窗口进程。对于接收到的每一条 WM_MOUSEMOVE 消息来说，应用程序都会及时地更新图象位置。这种任务是通过调用 ImageList_DragMove () 函数来完成的，该函数的语法结构如下所示：

```
#include <commctrl.h>
BOOL WINAPI ImageList_DragMove (int x, int y);
```

两个坐标指定了图象在屏幕上的新位置，我们需要把 lParam 内的值转换到屏幕坐标系统

里，从而计算出这个新位置。

```
...
pt.x = MAKEPOINTS (lParam) .x;
pt.y = MAKEPOINTS (lParam) .y;
// convert the cursor position in screen coordinates
ClientToScreen (hwnd, &pt);

// drag the icon
ImageList_DragMove (pt.x, pt.y);
...
```

假如用户把图象拖动到了客户区以外，那么程序依然为正常地运行下去，这是由于前面已经把桌面选择成了物主窗口。图 10-4 展示了图象拖动过程中的显示情况。

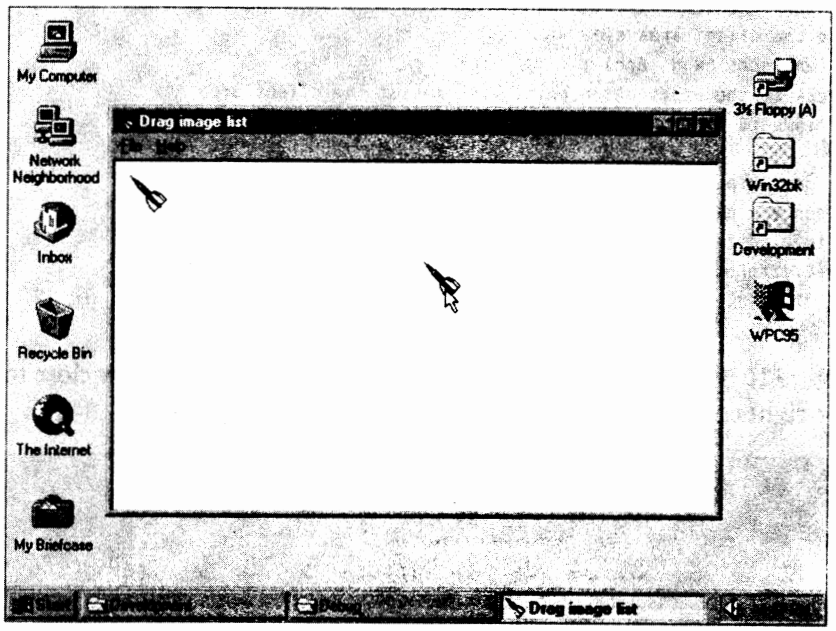


图 10-4　在拖动过程中，光标图标是与拖动图象关联起来的

拖动操作的中断是通过一条 WM_LBUTTONUP 消息的到达而知道的。松开了鼠标按钮以后，应用程序就会调用 ImageList_DragLeave () 和 ImageList_EndDrag ()，从而通知图象列表拖动已经结束。这时剩下的最后一项任务是检查目标位置是否在应用程序客户区的可见区域以内。如果这种测试通过，整个客户区都会暂时无效，并且在应用程序消息队列里发出一条 WM_PAINT 消息，从而对客户区进行重画：

```
...
// release the mouse
ReleaseCapture();

// unlock the window
ImageList_DragLeave(NULL);
ImageList_EndDrag();

// define the new icon position
pt.x = LOWORD(lParam);
pt.y = HIWORD(lParam);
// get the client area size
GetClientRect(hwnd, &rc);
// check if the destination point lays inside the client area
if(PtInRect(&rc, pt))
{
    ptIcon.x = pt.x - cx;
    ptIcon.y = pt.y - cy;
    // invalidate the entire client area
    InvalidateRect(hwnd, NULL, TRUE);
}
...
```

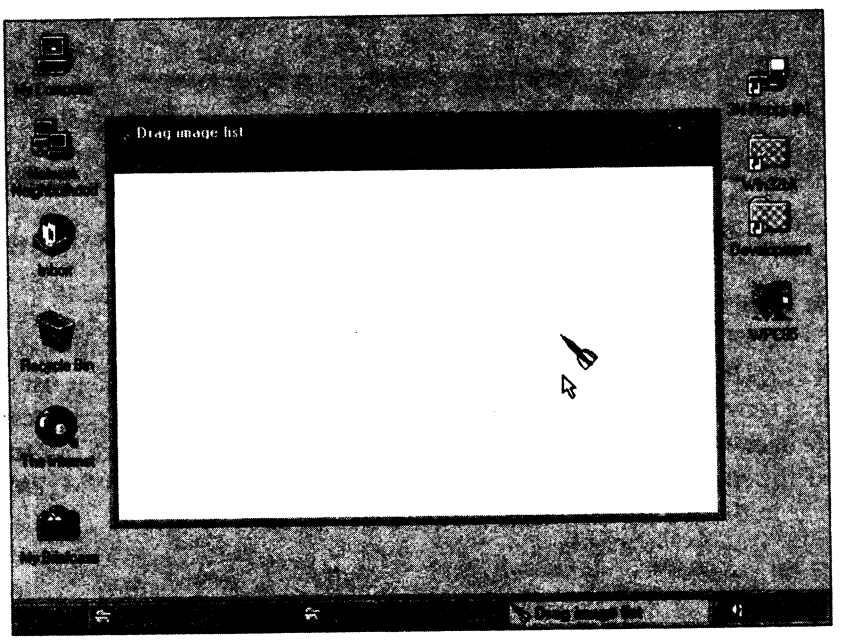


图 10-5　松开鼠标按钮，从而中断拖动操作以后，DRAGIL 程序的显示情况

图 10-5 向大家展示了 DRAGIL 程序在拖动操作结束时的显示情况。左上角的图标现在已经消失了。

相同的程序逻辑同样亦可应用于位图。当然，对 WM_LBUTTONDOWN，WM_MOUSEMOVE 以及 WM_LBUTTONUP 消息进行拦截的时候还需要多编写几行代码。总而言之，图象列表对象提供的工具给我们带来的好处是不言而喻的。

我建议大家修改一下第 9 章“预定义的窗口类”里的那个 EXTRACT 示范程序（可以在本书附带 CD 的 Listing 9.8 里找到那个程序），在其中利用图象列表功能。假如大家还有印象，就应该知道 EXTRACT 在自己的客户区内显示了从一个执行文件或者 DLL 模块内提取出来的所有图标。所有图标句柄都是存储于应用程序内部的，而图象列表看起来正是对这种对象进行管理的最恰当、最有效的一种工具。

最后，大家还要注意系统外壳本身就拥有一个图象列表，其中包含了我们常见的几幅图象。在某些情况下，我们在运行期间可以动态地访问这种图象数据库，而不对某些图标进行重画，这样能使自己的程序更精简。在本章的后面部分，我们会向大家详细介绍如何实现这种想法。

10.6 树形视窗控件

树形视窗控件是 MS Windows 95 系统外壳里一种非常普通的对象。假如打开了 EXPLORER.EXE 模块，如图 10-6 所示，就会在左边框的方框内发现一个树形视窗控件。系统外壳内的其他窗口也会利用树形视窗来显示具有分级结构的数据。在本书第 2 章“Win32 开发工具”里，我们曾经提到一个名为 INCLUDE 的例子，它使用一个树形视窗来显示出 Microsoft Development Environment for MS Windows 95 里用到的所有头文件。

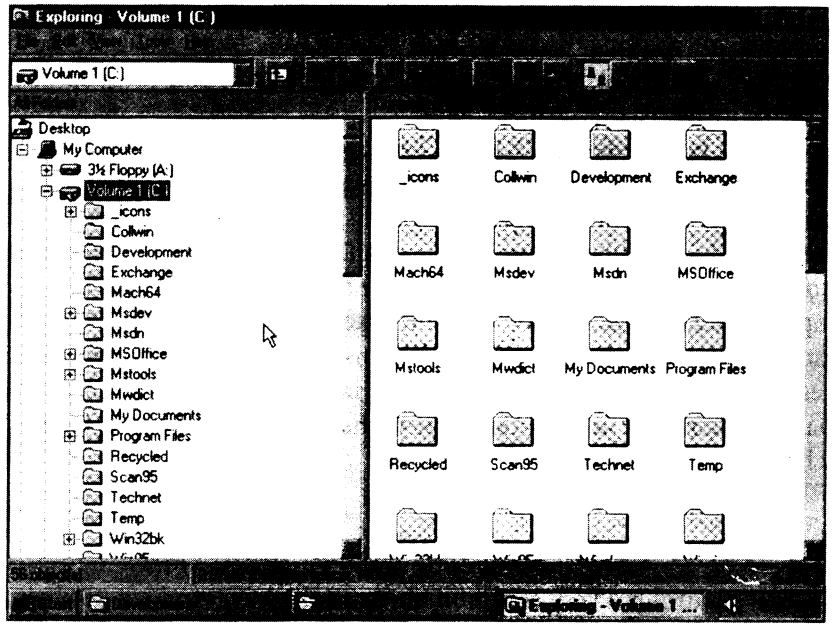


图 10-6 EXPLORER.EXE 利用一个树形视窗控件显示出具有分级结构的文件名信息

树形视窗是一种特殊的窗口，设计它的目的是用一种分级结构显示多个项目的一份列表。通常，在树形视窗里，每个节点都用一根细线互相连接到一起。这是一种相当复杂的控件类型，它为每个项目都提供了一个正文串标签，以及一幅可选的位图。在列表框里，我们利用索引对每个项目进行唯一性的标识。但在树形视窗里，它却为每个项目分配了一个特定的句柄。这种句柄信息是通过插入一个项目返回的，在树形视窗内插入新的项目时，也需要用到这种信息。在分级结构的顶部，我们看到的是一个“根项目”。在根项目下面还有一系列属于从属地位的项目，我们把这些项目叫作“子项目”。每个项目也都可以再包含更下级子项目，从而把自己变成一个“父项目”（亦可把这种关系称为“分支和节点”）。

在应用程序里，通过调用 CreateWindowEx () 这个 API 函数，同时在函数里指定 WC_TREEVIEW 定义或者 SysTreeview32 串，就可以建立起一个树形视窗控件。返回的句柄标识了整个窗口，以后会在这个窗口内填充用于定义单一项目的信息。创建进程会受到一个或者多个树形视窗风格的影响。这些风格定义都带有一个 TVS_前缀，大家可以参考表 10-12。

表 10-12　所有树形视窗风格的列表

树形视窗风格	值	说　明
TVS_HASBUTTONS	0x0001	在父项目的左侧放置一个加号（+）和减号（-）按钮
TVS_HASLINES	0x0002	用细线把树形视窗的节点连接起来
TVS_LINESATROOT	0x0004	连接位于树形视窗控件根部的项目。同时必须设置 TVS_HASLINES
TVS_EDITLABELS	0x0008	允许用户对树形项目的标签进行编辑
TVS_DISABLEDRAGDROP	0x0010	禁止树形视窗控件发送 TVN_BEGINDRAG 通知代码
TVS_SHOWSELALWAYS	0x0020	用系统高亮颜色描画一个项目，从而选定它

下面这个代码段向大家展示了如何建立一个树形视窗控件：

```
...
hwndTree = CreateWindowEx(WS_EX_CLIENTEDGE,
                          WC_TREE VIEW,
                          NULL,
                          WS_CHILD | WS_VISIBLE | TVS_HASLINES |
                            TVS_LINESATROOT|TVS_HASBUTTONS,
                          0, 0, 0, 0,
                          hwnd,
                          (HMENU)CT_TREE VIEW,
                          hInstance,
                          NULL);
...
```

一个典型的树形视窗控件提供了 TVS_HASBUTTON 风格，它的作用是强制性在每个节点的左侧显示一个加号按钮（矩形框中带有一个＋号）。当然，前提要求该节点带有一个或者多个子项目。假如分支已经展开，这时的加号按钮就会自动转换成减号按钮，从而向用户

指出现在可以对它进行折叠。

建立树形视窗的时候，还有另外两种风格也是很常用的，它们就是 TVS_HASLINES 和 TVS_LINESATROOT。这两种风格可以强制性地用细线把各个节点连接起来（其中包括根项目），从而从视觉上增强对它们之间链接关系的表达。增加了 TVS_EDITLABELS 这个定义以后，用户就可以当场对一个项目的标签进行编辑。这种特性在几种系统外壳对象中是很常见，比如文件夹、通用对话框以及 Find 选项等等。

作为一个子窗口，我们一般都把它的尺寸设置为零，把它的定位与尺寸定义工作推迟到拦截到 WM_SIZE 消息以后再进行。

10.6.1 插入一个新条目

在树形视窗中进行插入要以 TVM_INSERTITEM 消息以及 TV_INSERTITEM 数据结构为基础。这个数据结构是相当复杂的，其中不还包含了一个 TV_ITEM 结构。

```
typedef struct _TV_INSERTSTRUCT
{
    HTREEITEM hParent;
    HTREEITEM hInsertAfter;
    TV_ITEM item;
} TV_INSERTSTRUCT, *LPTV_INSERTSTRUCT;
```

其中的两个 HTREEITEM 项分别用于指定项目的父项目以及句柄（新的项目插入以后）。很常见的一种情况是，第二个句柄用下面这些定义的一种进行替换：TVI_FIRST，TVI_LAST 和 TVI_SORT，如表 10-13 所示。其中，TVI_FIRST 的作用是强迫树形视窗控件把新项目当作自己的第一个子项目进行放置。TVI_LAST 则沿用相同的运作机制，只是它把新项目放置到子列表的最底部。利用 TV_SORT，树形视窗控件可以按照字母顺序对新项目进行排序，然后把它插入子项目列表的恰当位置处。

表 10-13 用于定义树形视窗控件内项目插入点的标志

插入标志	值	说 明
TVI_ROOT	(HTREEITEM) 0xFFFF0000)	定义树形视窗控件的根项目
TVI_FIRST	(HTREEITEM) 0xFFFF0001)	把项目插入列表的的最起始处
TVI_LAST	(HTREEITEM) 0xFFFF0002)	把项目插入列表的的最末尾处
TVI_SORT	(HTREEITEM) 0xFFFF0003)	按照字母表顺序插入项目

所有实际的项目信息都是用 TV_ITEM 结构定义的，这是一个扩展性的数据块。除了 TVM_INSERTITEM 以外，它还用到了另外几条树形视窗消息。

正如我们以前提到的那样，树形视窗是由一系列相当复杂的信息组成的。某些情况下，它甚至还要比一个列表框复杂数倍。因此，插入一个新项目并不意味着仅仅是正文串的简单插入。对于插入的每个项目来说，都必须为其准备一个数据结构，用它来容纳某些有用的信息。然后，再把这些信息传递给树形视窗控件。在某些情况下，我们甚至可以省略一些实际正文的传递。取而代之的是，我们可以把所有正文信息都存储于应用程序的某些数据结构里，而

不是存放于控件里。这种策略要求在显示一个项目的时候，树形视窗控件与应用程序之间必须进行频繁的交互作用。在那个时候，树形视窗会呼叫应用程序，从而获得正文串的输出许可。

现在让我们依次进行讨论。TV_ITEM 数据结构的具体定义如下所示，其中包含了 10 个项。由这么多的设置项可以看出一个树形视窗项目具有的复杂程度，同时也反映出我们需要用到大量的数据。

```
typedef struct _TV_ITEM
{
    UINT mask;
    HTREEITEM hItem;
    UINT state;
    UINT stateMask;
    LPSTR pszText;
    int cchTextMax;
    int iImage;
    int iSelectedImage;
    int cChildren;
    LPARAM lParam;
} TV_ITEM, *LPTV_ITEM;
```

其中，mask 项包含了表 10-14 列出的一个或者多个定义。这个项在 TV_ITEM 的整体结构中扮演了一种相当关键的角色。那些定义可以指示一个树形视窗控件在插入一个新项目时必须考虑到那些 TV_ITEM 结构项。正如我们以后会看到的那样，TV_ITEM 结构的作用在于把其他几条消息联接起来，从而实现对某些项目属性的读取或者更改。

表 10-14　TV_ITEM 数据结构 mask 项使用的标志定义

用于 TV_ITEM 屏蔽项的标志	值	说　明
TVIF_TEXT	0x0001	pszText 和 hTextMax 项是有效的
TVIF_IMAGE	0x0002	iImage 项是有效的
TVIF_PARAM	0x0004	lParam 项是有效的
TVIF_STATE	0x0008	state 和 StateMask 项是有效的
TVIF_HANDLE	0x0010	hItem 项是有效的
TVIF_SELECTEDIMAGE	0x0020	iSelectedImage 项是有效的
TVIF_CILDREN	0x0040	cChildren 项是有效的

插入一个新项目的时候，TVIF_HANDLE 标志几乎肯定是用不上的。它的作用是强制树形视窗窗口对 TV_ITEM 结构内的 hItem 项进行检查。在树形视窗控件内增添一个新项目的时候，这种信息将缺少任何真正的值。但是假如发送了其他信息时，比如 TVM_GETITEM 或者 TVM_SETITEM，这个项的存在就有意义了。在那些情况下，hItem 内包含了树形视窗

的句柄，这种句柄唯一性地标识了一个项目——这是针对这一目的可以利用的唯一信息。

TVIF_TEXT 激活了 pszText 和 cchTextMax 项，并且需要用某个正文串来指定一个项目。在这个时候，我们可以有两种选择：既可以把一个正文串的正文和相应的串长度传递给树形视窗控件，也可以在某些数据结构里保存所有的项目正文串，这些数据结构是专门在应用程序源代码内分配的。在后面那种情况下，我们将为 pszText 项分配一个 LPSTR_TEXTCALBACK 定义。这样一来，该显示某个项目的时候，就可以指示树形视窗向应用程序发出对对应项目标签的请求。在那个时候，控件的父窗口就会接收到一条 TVN_GETDISPINFO 通知代码，这条通知代码封装于一条 WM_NOTIFY 消息内，同时还会接收到一个 TV_DISPINFO 数据结构的地址，在这个数据结构里准备填写相应的输出信息，如下所示：

```
typedef struct _TV_DISPINFO
{
    NMHDR hdr;
    TV_ITEM item;
} TV_DISPINFO;
```

在 TV_DISPINFO 里，第一个参数是一个 NMHDR 结构，其中包含了通过一条 WM_NOTIFY 消息传送的标准信息，接下来则是一个 TV_ITEM 数据结构。

与把一个项目标签简单地插入树形视窗的内存区域相比，采用LPSTR_TEXTCALLBACK 标志也许会显得比较累赘，尽管这种方法还是提供了某些附加的优点。通过这种方式，正文串可以用一种更加灵活的方式进行维护，查找工作也能更有效地实现，并且可以根据不同的标准进行查找。

TVIF_CHILDREN 标志指定的是 cChildren 项。假如节点提供了一个或者多个子项目，就必须把 cChildren 设置成 1，否则应该设置成 0。

正如我们以前提到的那样，树形视窗的项目支持与每个项目关联起来的一幅可选的图象。用于显示的图象通常是从一个图象列表里取得的，这个图象列表则是与树形视窗控件关联在一起的。TVIF_IMAGE 标志激活了 iSelectdImage 和 iImage 项。这两个项的作用是分别指定显示于已选中和未选中项目标签左侧的那幅图象。通过为每个项目动态地指定两幅图象，开发者很容易就可以把一幅图象与未选择状态关联起来，而把另一幅与选择状态关联起来。就 EXPLORER.EXE 来说，假如展开了一个节点，我们就会发现这种情况。这时显示出来的是一个打开的文件夹，未选中时则是一个关闭的文件夹。

对于 TVIF_STATE 里的两个项 state 和 stateMask 来说，它们中的每一个都分配了表 10-15 内列出的某一种风格标志。

最后一个项是 lParam，它指定了一个四字节的内存区域，每个项目都关联了这样的一个区域。它的作用是为与那个特定项目有关的任何类型的数据提供一定的存储空间。这种为每个项目分配附加字节的方案与列表框预定义窗口类的情况是完全一致的。我们可以在其中存储一个整数值、指向其他内存位置的一个指针、一个句柄或者对项目有用的其他任何形式的数据，只要这个区域放得下。

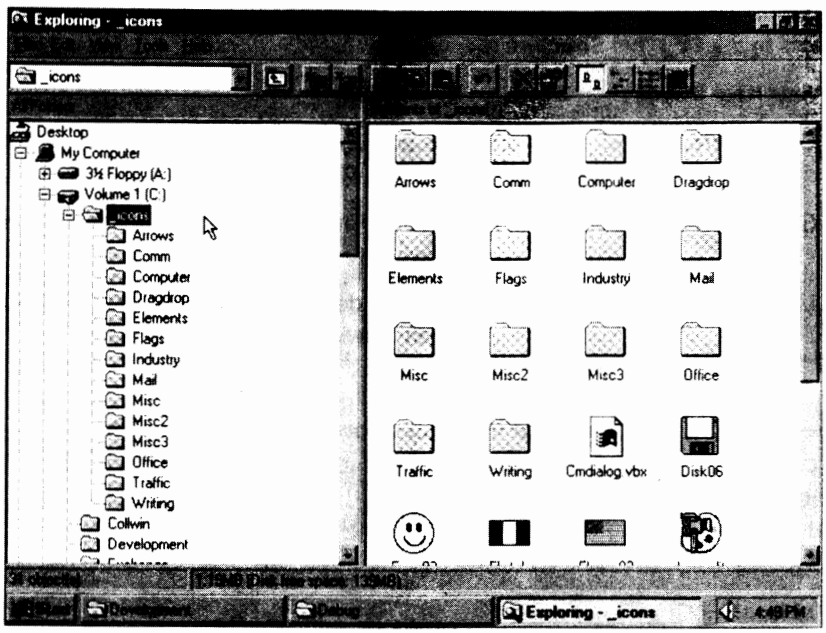


图 10-7　在 EXPLORER 窗口的左侧方框内，一个已伸张开并被选中的节点用一个不同的图标来指出用户的选择

表 10-15　分配给 TV_ITEM 数据结构 State 和 StateMask 项的树形视窗状态标志

树形视窗项目状态	值	说　明
TVIS_FOCUSED	0x0001	项目已有视觉焦点
TVIS_SELECTED	0x0002	项目已被选中
TVIS_CUT	0x0004	项目作为一次剪切操作的部分被选中
TVIS_DROPHILITED	0x0008	项目作为拖放操作的目标被选中
TVIS_BOLD	0x0010	项目以黑体字显示
TVIS_EXPANDED	0x0020	指定项目的所有子项目都已展开，并且假定为可见的
TVIS_EXPANDEDONCE	0x0040	表明指定的父项目至少已经展开过一次
TVIS_OVERLAYMASK	0x0F00	描绘一个项目的时候，用于那个项目的重叠图象也包含在内
TVIS_STATEIMAGEMASK	0xF000	描绘一个项目的时候，用于那个项目的状态图象也包含在内
TVIS_USERMASK	0xF000	描绘一个项目的时候，用于那个项目的用户自定义图象也包含在内

下面这个代码段向大家展示了怎样在树形控件里插入一个新项目。我们通常都从根项目开始，把 TVI_ROOT 定义分配给 TV_INSERTITEM 数据结构内的 hParent 项，然后任意选择 TVI_LAST 或者 TVI_FIRST 作为自己的插入方案。

```
...
char szText[30];
TV_INSERTSTRUCT tvis;
HTREEITEM hTreeNew;
```

```
...
// filling TREEVIEW
lstrcat(szText, "windows.h");
// inserting the include files in TREEVIEW
tvis.hParent = TVI_ROOT;
tvis.hInsertAfter = TVI_LAST;
    tvis.item.mask = TVIF_TEXT | TVIF_PARAM | TVIF_IMAGE |
    TVIF_SELECTEDIMAGE;
tvis.item.lParam = TRUE;
tvis.item.pszText = "WINDOWS.H";
tvis.item.cchTextMax = strlen("WINDOWS.H");
tvis.item.iImage = 0;
tvis.item.iSelectedImage = 1;
hTreeNew = TreeView_InsertItem(hwndTree, &tvis);
```

这段代码是从我们在第 2 章“Win32 开发工具”里讨论的那个 INCLUDE 示范程序里摘录下来的，其中每个项目的所有信息都局限于一个单独的正文串（TVIF_TEXT）。除此以外，应用程序还利用了附加的四字节内存（TVIF_PARAM）。方法是设置一个布尔值，用它指定一系列子项目的存在。

大家也许会想到，由 TV_ITEM 的 lParam 项指定单独的项目数据区域是一种相当浪费内存空间的做法，这是由于 cChildren 项基本上具有相同的作用。然而，考虑到尽管是个整数，cChildren 的行为却更像一个布尔值——为 1，表示存在一个或者多个子项目；为 0，表示没有子项目。除此以外，也可以强迫一个负值进入项目的内存区域，即使并没有实际的子项目存在也能如此，这样就欺骗了树形窗口控件。

在 INCLUDE 示例里，假如试图从根项目开始立即填写所有项目，就会遇到不可接受的性能退化。为了避免这种限制，我们最好一次只插入同一分级上的所有项目，这样就为用户造成了整个树结构在控件内已经存放好了的假象。这种策略要求在父项目的左侧显示一个加号（+），即使它的子项目信息还没有插入。通过把 cChildren 设置成 1，视觉效果就会自动激活，尽管此时应用程序内部知道分支结构已经展开了。展开逻辑是在应用程序接收到一条 TVN_ITEMEXPANDING 消息以后发生的，其中带有用于树形视窗类的一条通知代码，所有的通知代码稍后会列于表 10-22 内。在那个时候，需要把 FALSE 变换成 TRUE，从而指出那个分支内已经填写好了适当的信息。

插入进程将由一个新项目句柄的返回而中断。对于一个应用程序来说，它应该首先计划好一种策略，从而保存这种信息，以便以后利用。举个例子来说，进入了根项目以后，根项目的句柄在所有子项目的插入过程中就显得相当重要了，所有子项目都会引用它。在这种情况下，树形视窗窗口类没有提供对这种工作进行简化的任何便利，然而，我们可以采取下面这种策略。树形视窗也是一种典型的窗口，它能用一种递归算法进行访问和管理，在内部把所有信息都存放于一个单一的函数内。除此以外，这一类使用的许多通知代码都提供了某些简便的方法，利用它们可以判断和获取选中节点的句柄。

10.6.2 项目标签的编辑

一个项目是用自己的标签进行标识的，并且可以选择在旁边显示一幅图象。项目标签是在树形视窗控件内插入一个新的项目时定义的，或者到以后需要显示它的时候再提供相应的标签（LPSTR_TEXTCALLBACK）。

树形视窗支持的另外一种有趣的特性是它可以实现项目标签的现场编辑，这种编辑很容易实现，而且采用的手段也是相当有效的。从根本上说，我们需要做的一切就是在建立控件的时候添加一个 TVS_EDITLABELS 风格，然后通过 WM_NOTIFY 消息对相应的通知代码进行跟踪。为了激活现场编辑，只需要在选定的项目标签上方单击一次鼠标左键。等待一小会儿，一个 EDIT 窗口就会覆盖用于显示正文的整个标签区域。在这个时候，父窗口在自己的窗口进程内就会接收到一条 WM_ NOTIFY 消息，其中带有相应的 TVN_BEGINLABELEDIT 通知代码。由 lParam 引用的信息指向一个 TV_DISPINFO 数据结构的地址，这个数据结构的 item. pszText 里包含了当前的项目标签正文。

```
TVN_BEGINLABELEDIT
ptvdi = (TV_DISPINFO *) lParam
```

图 10-8 向大家展示了在 INCLUDE 示范程序的修订版本里，对根项目标签进行编辑的情况。

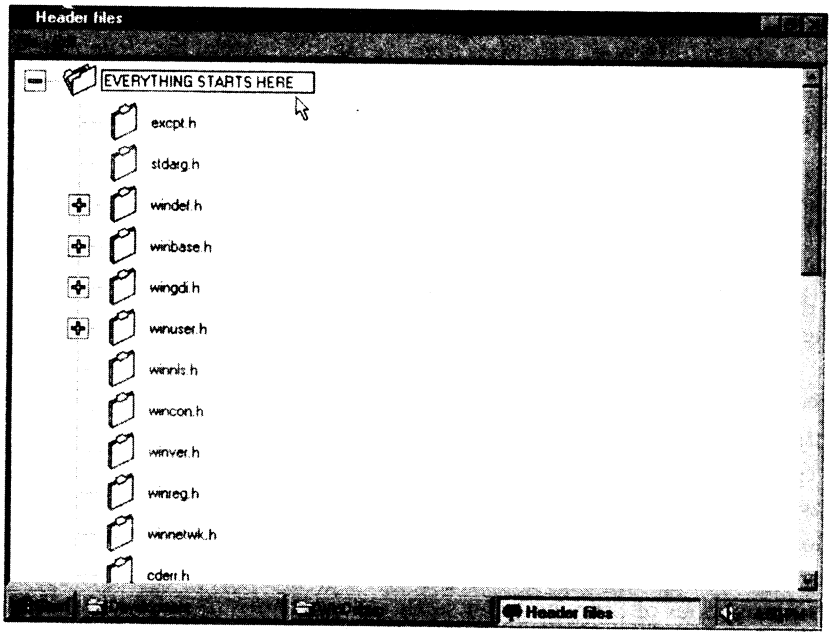


图 10-8 用正文串 EVERYTHING STARTS HERE 取代了原始的 WINDOWS. H 标签

通过对这条通知代码进行拦截和处理，应用程序就可以知道那个特定的正文串是否能够进行编辑，这样可以避免用户通过错误的方式改变树形结构内某个关键性的组件。除此以外，TVM_GETEDITCONTROL 消息会返回编辑控件窗口的句柄。这个窗口的句柄信息是相当关键的，利用它可以和编辑窗口直接进行交互作用，通过一条 EM_SETLIMITTEXT 消息限制其中的正文数量，激活某种特定的行为（比如 ES_PASSWORD），甚至还可以对其进行子

类处理。从这个时候开始，输入焦点就会转移到编辑控件上面，直到按下了 Esc 键或者 Return 键为止。这时，树形视窗的父窗口会再次在自己的窗口进程里接收到一条 WM_NOTIFY 消息，这一次它携带的是 TVN_ENDLABELEDIT 通知代码：

```
TVN_ENDLABELEDIT
ptvdi = (TV_DISPINFO *) lParam
```

假如按下的是 Esc 键，pszText 项内就会包含一个空的字符串，这是每次接收到一条 TVN_ENDLABELEDIT 通知代码时都应该首先进行检查的一种情况。接下来，我们必须对这条消息进行处理，从而永久性地反映出编辑控件对项目标题作出的变动。

TV_DISPINFO 数据结构里的 TV_ITEM 项包含了完成这一任务所需的所有信息。假设 ptvdi 是指向一个 TV_DISPINFO 数据结构的指针，那么 ptvdi —〉item. pszText 内就包含了编辑过后的新正文。我们必须把 TV_DISPINFO 数据结构里的 TV_ITEM 项回传给树形视窗控件，这样才能让新标题在项目标签内显示出来。在这儿，TVM_SETITEM 消息是我们能够利用的最佳工具。然而，在实现标签的改变之前，我们需要对 TV_DISPINFO 数据结构的 TV_ITEM 的两个项进行某些变动。首先要进行的一个最重要处理是：让 mask 项提供 TVIF_TEXT 标志，否则便无法激活 pszText 和 cchTextMax 项。其次，我们还需要用实际的正文长度对 cchTextMax 进行更新。这两种操作需要的代码量都不大。在这以后，我们就做好了把信息回传给树形视窗窗口的所有准备：

```
...
case TVN_ENDLABELEDIT:
{
    TV_DISPINFO *ptvdi = (TV_DISPINFO *)lParam;

    // check if the string is empty
    if(! ptvdi —〉item. pszText)
        break;
    // prepare the information to be passed to the selected item
    ptvdi —〉item. mask = TVIF_TEXT;
    ptvdi —〉item. cchTextMax = sizeof(ptvdi —〉item. pszText);
    TreeView_SetItem(hwndTree, &(ptvdi —〉item));
}
    break;
...
```

最后生成的结果请大家参照图 10-9。现在的根项目是用正文串 EVERYTHING STARTS HERE 标识的。

TVM_SETITEM 消息向树形视窗控件传递了一个 TV_ITEM 数据结构，用于更新它当

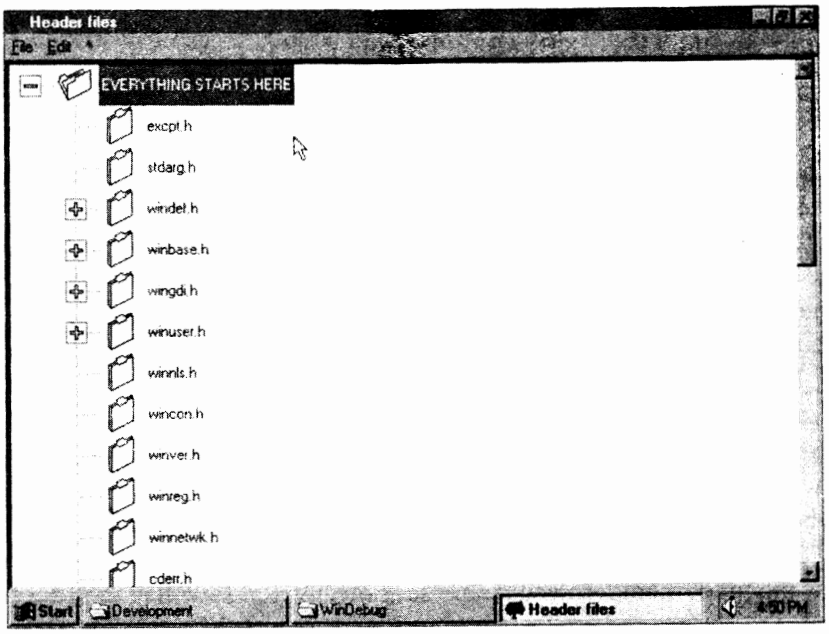


图 10-9　根项目标签已用一个正文串成功地替换掉了，
这个正文串是直接在项目里编辑的

前的状态、标签或者其他属性。类似地，TVM_GETITEM 的作用是用当前与一个特定树形视窗项目关联起来的所有信息填充一个 TV_ITEM 数据结构。表 10-6 为大家总结了所有可用的树形视窗消息。

表 10-16　树形视窗消息

树形视窗消息	值	说　明
TVM_INSERTITEM	(TV_FIRST + 0)	在树形视窗控件里插入一个新项目
TVM_DELETEITEM	(TV_FIRST + 1)	从树形视窗控件里删除一个现成项目
TVM_EXPAND	(TV_FIRST + 2)	对树形视窗控件的一个父节点进行展开
TVM_GETITEMRECT	(TV_FIRST + 4)	为某个树形视窗项目返回一个限制矩形，指定其是否可见
TVM_GETCOUNT	(TV_FIRST + 5)	返回一个树形视窗控件内所有项目的总数
TVM_GETINDENT	(TV_FIRST + 6)	取得子项目缩进的距离大小(用象素数表示)，这种缩进与自己的父项目是相对的
TVM_SETINDENT	(TV_FIRST + 7)	设置子项目缩进的距离大小(用象素数表示)，这种缩进与自己的父项目是相对的
TVM_GETIMAGELIST	(TV_FIRST + 8)	返回与树形视窗关联在一起的图象列表的句柄
TVM_SETIMAGELIST	(TV_FIRST + 9)	把一个图象列表与一个树形视窗关联在一起
TVM_GETNEXTITEM	(TV_FIRST + 10)	返回一个项目的信息，这个项目与某个指定项目是相对的。表 10-17 列出了所有可能的值
TVM_SELECTITEM	(TV_FIRST + 11)	选择指定的树形视窗项目
TVM_GETITEM	(TV_FIRST + 12)	取得与一个树形视窗项目有关的某些信息和属性
TVM_SETITEM	(TV_FIRST + 13)	设置与一个树形视窗项目有关的某些信息和属性

（续）

树形视窗消息	值	说　明
TVM_EDITLABEL	(TV_FIRST + 14)	激活现场编辑操作
TVM_GETEDITCONTROL	(TV_FIRST + 15)	返回一个编辑控件的句柄，这个编辑控件正用于对树形视窗的正文进行编辑
TVM_GETVISIBLECOUNT	(TV_FIRST + 16)	返回树形视窗项目的数目，要求这些视窗项目在树形视窗控件窗口内是完全可见的
TVM_HITTEST	(TV_FIRST + 17)	判断当前的鼠标指针位置，这个位置与树形视窗控件窗口是相对的
TVM_CREATEDRAGIMAGE	(TV_FIRST + 18)	针对树形视窗内指定的项目建立一个拖动位图
TVM_SORTCHILDREN	(TV_FIRST + 19)	对树形视窗内指定父项目的所有子项目进行排序处理
TVM_ENSUREVISIBLE	(TV_FIRST + 20)	确保一个特定的项目是可见的，如有必要，就把父项目展开
TVM_SORTCHILDRENCB	(TV_FIRST + 21)	用应用程序定义的一个回调函数对树形视窗项目进行排序处理，这个回调函数的作用是定义相应的排序标准
TVM_ENDEDITLABELNOW	(TV_FIRST + 22)	中断树形视窗项目标签的编辑工作
TVM_GETISEARCHSTRING	(TV_FIRST + 23)	取得用于树形视窗控件的增量检索字符串

10.6.3　分支排序

假如使用了 TVI_SORT 标志，那么就可以在用户插入项目的同时，在一个节点分支里对插入的项目进行自动排序。即使已经通过发送 TVM_SORTCHILDREN 消息，从而插入了所有的子项目，相同的结果仍然可以实现。

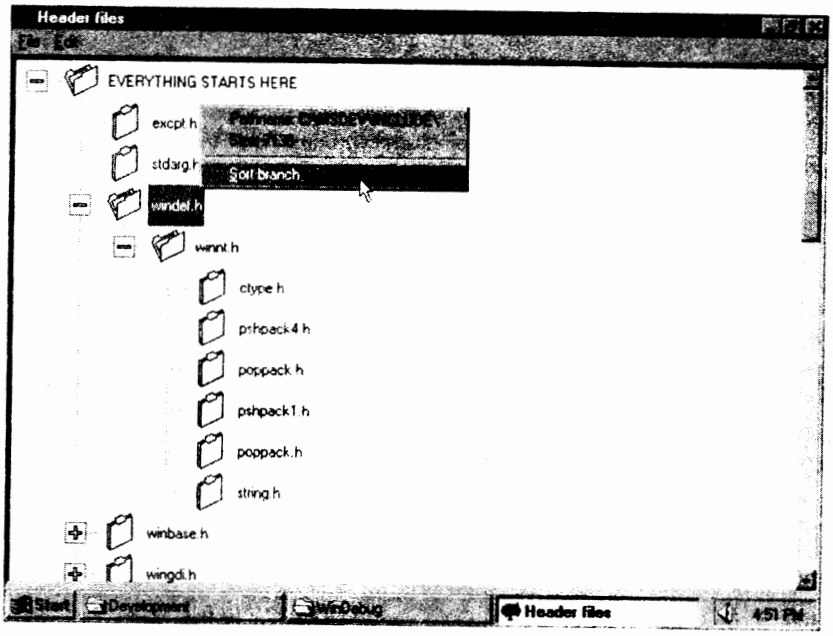


图 10-10　与每个项目相关的弹出式菜单可能提供了 Sort branch（分支排序）菜单项，前提要求这是一个父项目

下面这个代码段是从 INCLUDE 的更新版本里提取出来的，该示范程序可以在本书附带 CD 的 Listing 10.3 里找到，它向我们展示了怎样对一个子项目进行排序，从而对菜单项的选定作出响应。在这个例子里，假如在某个项目上方单击鼠标右键，就会显示出一个相应的弹出式菜单，如图 10-10 所示。

选中一个项目以后，应用程序就会接收到一条 WM_COMMAND 消息，其中的 wParam 低字内还包含了 MN_SORTBRANCH，这和传统的菜单项选定是一致的。在这个消息块内，我们必须取得当前选中的那个树形视窗项目的句柄。为了实现这一要求，可以发送一条 TVM_GETNEXTITEM 消息，并且指定 TVGN_CARET 标志（所有这些均可通过调用函数 TreeView_GetSelection () 方便地完成）。所有 TVGN_定义的一份完整列表可以在表 10-17 里找到，这些 TVGN_定义是与 TVM_GETNEXTITEM 消息关联起来的。只要知道了选定项目的句柄，紧接着再发送一条 TVM_SORTCHILDREN 消息就足够了。这条消息利用树形视窗控件完成了最终的排序工作。

```
...
case MN_SORTBRANCH:
{
    HTREEITEM hTreeSel;

    // get the handle of the selected item
    hTreeSel = TreeView_GetSelection(hwndTree);
    // sort that branch
    TreeView_SortChildren(hwndTree, hTreeSel, 0);
}
    break;
...
```

表 10-17　与 TVM_GETNEXTITEM 消息联合使用的各种定义

用于 TVM_GETNEXTITEM 的标志	值	说　明
TVGN_ROOT	0x0000	返回一个树形视窗控件内根项目的句柄
TVGN_NEXT	0x0001	返回下一个同级（兄弟）树形视窗项目的句柄
TVGN_PREVIOUS	0x0002	返回上一个同级（兄弟）树形视窗项目的句柄
TVGN_PARENT	0x0003	返回树形窗口父项目的句柄
TVGN_CHILD	0x0004	返回树形窗口第一个子项目的句柄
TVGN_FIRSTVISIBLE	0x0005	返回树形窗口第一个可见子项目的句柄
TVGN_NEXTVISIBLE	0x0006	返回树形窗口下一个可见项目的句柄
TVGN_PREVIOUSVISIBLE	0x0007	返回树形窗口上一个可见项目的句柄
TVGN_DROPHILITE	0x0008	返回一次拖放操作中目标树形视窗项目的句柄
TVGN_CARET	0x0009	返回当前选中的树形视窗项目的句柄

图 10-11 为我们显示了从根项目开始排序以后的头文件树形结构。

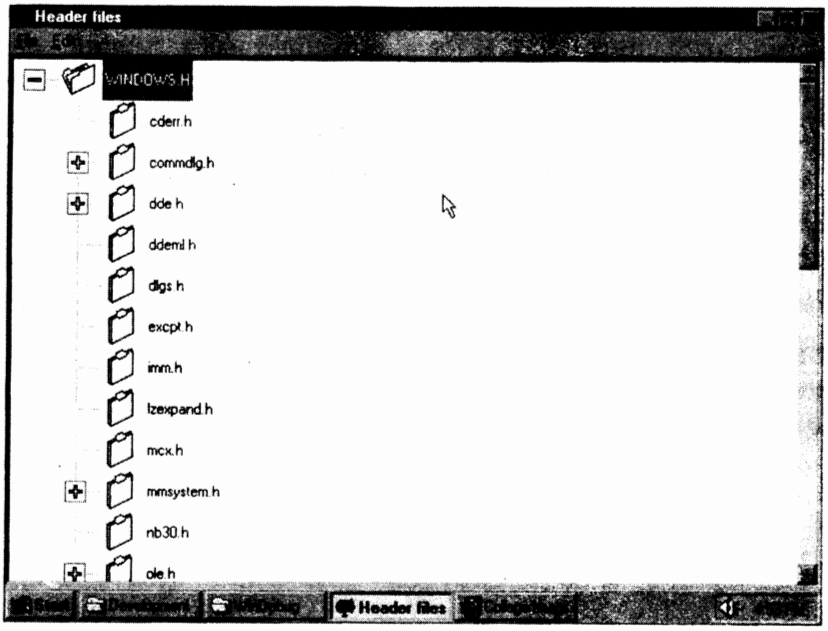


图 10-11　从 WINDOWS.H 这个根项目开始，按照字母顺序对 Win32 包容文件进行排序以后得到的结果

作为另外一种选择，我们也可以使用 TVM_SORTCHILDRENCB 消息。这条消息提供了一种更加灵活、定制度更大的方式，利用它可以简便地对树形视窗控件内的项目标签进行排序处理。这条消息会向一个树形视窗控件传递数据结构的地址，这个数据结构的名字叫作 TV_SORTCB。

```
typedef struct _TV_SORTCB
{
    HTREEITEM hParent;
    PFNTVCOMPARE lpfnCompare;
    LPARAM lParam;
} TV_SORTCB, *LPTV_SORTCB;
```

其中，hParent 项的作用是标识一个起始节点，排序算法就是从这个节点处开始实施的。第二个参数代表了某个回调函数的地址，只要有必要对两个列表项目的相对顺序进行比较，就需要调用到这个回调函数。这个函数的结构如下所示：

```
int CALLBACK CompareFunc (LPARAM lParam1, LPARAMlParam2, LPARAM
    lParamSort);
```

前面两个参数与 TV_ITEM 数据结构里的 lParam 项对应，它们分别与准备比较的两个项目对应。而 lParamSort 则是由应用程序定义的一个值，它的值与 TV_SORTCB 的 lParam 项对应。我们在这儿利用了 lParam，这意味着为了实现与简单的串比较方式不同的一种排序

标准，必须事先考虑好。事实上，这种方案强迫我们利用 lParam 的四字节内存位置，并且在这个内存区域内包含一些有用的信息，这些信息对于排序标准的制订是很有价值的。通常，lParam 是指向一个应用程序定义数据结构的指针，其中包含了每个节点内显示出来的正文。

因此，就我们以前在本书附带 CD-ROM 的 Listing 2.2 内实现的那个 INCLUDE 示范程序来说，其中的 lParam 表明当子项目插入父节点内时，已经实现了一种定制的排序机制。因此，假如我们计划在某个树形视窗控件里实现某种定制排序机制，那么在设计阶段就必须先考虑好。事先应该计划出对 TV_ITEM 数据结构内 lParam 项的一种正确的利用方案。相同的逻辑亦可应用于一个列表视窗控件，对此我们会在本章的稍后部分进行详细的介绍。

假如第一个项目应该位于第二个之前，回调函数就会返回一个负值；相反，假如第一个项目应该位于第二个之后，返回的就是一个正值。假如两个项目是等价的，就返回一个零值。

10.6.4 消息和宏函数

如表 10-16 所示，其中列出的所有树形视窗消息都容易封装到表 10-18 列出来的那些树形视窗宏函数里。请注意，TreeView_GetCount () 返回的是树形视窗控件的项目总数，其中包括所有的子分支。

表 10-18 可以封装 TVM_消息的所有树形视窗宏函数

树形视窗宏	说 明
TreeView_InsertItem (hwnd, lpis)	在树形视窗控件内插入一个项目
TreeView_DeleteItem (hwnd, hitem)	从树形视窗控件内删除一个项目
TreeView_DeleteAllItems (hwnd)	从树形视窗控件内删除所有项目
TreeView_Expand (hwnd, hitem, code)	展开一个父节点
TreeView_GetItemRect (hwnd, hitem, prc, code)	返回围绕于一个项目的矩形区域
TreeView_GetCount (hwnd)	返回一个树形视窗控件内的项目总数
TreeView_GetIndent (hwnd)	返回一个子项目缩进的距离（用象素点表示）
TreeView_SetIndent (hwnd, indent)	设置一个子项目缩进的距离（用象素表示）
TreeView_GetImageList (hwnd, iImage)	返回与那个树形视窗控件关联起来的图象列表的句柄
TreeView_SetImageList (hwnd, himl, iImage)	把一个图象列表与一个树形视窗控件关联起来
TreeView_GetNextItem (hwnd, hitem, code)	取得分配给一个树形视窗项目几种属性（参考表 10-17）
TreeView_GetChild (hwnd, hitem)	返回指定父项目的第一个子项目的句柄
TreeView_GetNextSibling (hwnd, hitem)	返回第一个同级（兄弟）项目的句柄
TreeView_GetPrevSibling (hwnd, hitem)	返回前一个同级（兄弟）项目的句柄
TreeView_GetParent (hwnd, hitem)	返回树形视窗父项目的句柄
TreeView_GetFirstVisible (hwnd)	返回第一个可见项目的句柄
TreeView_GetNextVisible (hwnd, hitem)	返回下一个可见项目的句柄
TreeView_GetSelection (hwnd)	返回当前选中的那个项目的句柄
TreeView_GetDropHilight (hwnd)	返回一次拖放操作的目标项目
TreeView_GetRoot (hwnd)	返回根项目的句柄
TreeView_Select (hwnd, hitem, code)	根据表 10-17 中的某种 TVGN_定义选择一个项目
TreeView_SelectItem (hwnd, hitem)	选择一个项目
TreeView_SelectDropTarget (hwnd, hitem)	选择拖放操作中的目标项目
TreeView_SelectSetFirstVisible (hwnd, hitem)	选择第一个可见的项目
TreeView_GetItem (hwnd, pitem)	获取与某个树形视窗项目有关的某些信息和属性
TreeView_SetItem (hwnd, pitem)	设置与某个树形视窗项目有关的某些信息和属性
TreeView_EditLabel (hwnd, hitem)	激活现场编辑
TreeView_GetEditCount (hwnd)	返回正在用于编辑树形视窗项目正文的一个编辑控件的句柄

（续）

树形视窗宏	说　明
TreeView_GetVisibleCount (hwnd)	返回树形视窗控件窗口内完全可见的项目总数
TreeView_HitTest (hwnd, lpht)	判断与树形视窗控件窗口相对的当前光标位置
TreeView_CreateDragImage (hwnd, hitem)	为树形视窗内指定的一个项目建立拖动位图
TreeView_SortChildren (hwnd, hitem, recurse)	对树形视窗控件内指定父项目的所有子项目进行排序处理
TreeView_EnsureVisible (hwnd, hitem)	确保指定的项目是可见的，如有必要，就把父节点展开
TreeView_SortChildrenCB (hwnd, psort, recurse)	用应用程序定义的一个回调函数对树形视窗的项目进行排序处理，这个回调函数的作用是定义排序的标准
TreeView_EndEditLabelNow (hwnd, fCancel)	中断对树形视窗项目标签的编辑工作
TreeView_GetISearchString (hwndTV, lpsz)	取得树形视窗控件使用的增量检索串

通过发出一条 TVM_EXPAND 消息或者等价的 TreeView_Expand ()，应用程序就可以强制性地对树形项目进行展开处理。这条消息的行为是在 TVE_定义的基础上实现的。所有 TVE_定义均可在表 10-19 里找到。父项目很容易就可以展开、折叠或者从当前状态转为另外一种状态。

表 10-19　用于展开或者折叠一个树形视窗节点的标志

用于 TVM_EXPAND 的节点标志	值	说　明
TVE_COLLAPSE	0x0001	强迫列表折叠
TVE_EXPAND	0x0002	强迫列表展开
TVE_TOGGLE	0x0003	假如列表当前处于展开（折叠）状态，就转为相反的折叠（展开）状态
TVE_COLLAPSERESET	0x8000	TVE_COLLAPSE 标志的扩展。除了把一个分支折叠起来以外，它还要删除子项目

为了判断鼠标指针当前在树形视窗控件里的位置，我们可以调用 TreeView_HitTest()函数，该函数提供了这方面一些很有用的信息。相关的 TV_HITTESTINFO 结构内包含了用客户区坐标系统表达的鼠标光标位置信息，这是表 10-20 里列出某种 TVHT_定义。另外还包含了占据控件内那个特定位置的项目的句柄（鼠标指针正好在该项目的上方）。

```
typedef struct _TV_HITTESTINFO
{
    POINT pt;
    UINT flags;
    HTREEITEM hItem;
} TV_HITTESTINFO, *LPTV_HITTESTINFO;
```

假如用户按下了鼠标右键，就需要把这条消息发送给树形视窗控件。用户的这种操作生成了一条 MN_RCLICK 通知代码，其中并未携带与树形视窗有关的任何特定信息。因此，应用程序自己必须判断用户是否在一个敏感位置（比如项目标签）单击了鼠标右键。假如测试

成功地通过，hItem 项就会包含与研究过的标准匹配的项目句柄。

表 10-20　与 TVM_HITTEST 消息关联在一起的“点击测试”定义

点击测试标志	值	说明
TVHT_NOWHERE	0x0001	在客户区内，然而在最后一个项目的下方
TVHT_ONITEMICON	0x0002	在一个项目图标上方
TVHT_ONITEMLABEL	0x0004	在一个项目标签上方
TVHT_ONITEM		(TVHT_ ONITEMICON | TVHT_ OITEMLABEL | TVHT_ ONITEMSTATEICON) 在一个项目上方
TVHT_ONITEMINDENT	0x0008	在一个项目缩进区域的上方
TVHT_ONITEMBUTTON	0x0010	在一个项目按钮上方
TVHT_ONITEMRIGHT	0x0020	在项目右侧的区域内
TVHT_ONITEMSTATEICON	0x0040	在用于树形视窗项目的状态图标上方，该项目处于用户自定义状态下
TVHT_ABOVE	0x0100	在客户区的上面
TVHT_BELOW	0x0200	在客户区的下面
TVHT_TORIGHT	0x0400	在客户区的右侧
TVHT_TOLEFT	0x0800	在客户区的左侧

下面这段示范代码向我们展示了如何判断用户是否在一个项目标签上方单击了鼠标右键：

```
...
case NM_RCLICK:
{
    POINT pt;
    TV_HITTESTINFO tvht;

    // saving the mouse coordinate
    GetCursorPos (&pt);
    tvht.pt = pt;
    // converting the coordinate in the TreeView window
    ScreenToClient (hwndTree, &tvht.pt);
    // search text only
    tvht.flags = TVHT_ONITEMLABEL;
    // did we select an item?
    TreeView_HitTest (hwndTree, &tvht);

    // yes, we selected an item
    if (tvht.hItem)
...
```

通过为一个 TV_HITTESTINFO 数据结构内的标志项分配其他 TVHT_定义，应用程序就可以判断光标位置是否在一个树形视窗控件的上方，接着即可根据特定的需求定制相应的行为，采取相应的行动。

10.6.5 图象列表和树形视窗

假如有一个图象列表与树形视窗关联在一起，那么树形视窗就能利用图象列表改进自己每个项目显示时的外观。INCLUDE 示范程序在图象列表里插入了两个图标，分别代表一个关闭的文件夹和一个打开的文件夹。第一幅图象是每个项目都要使用的，而第二个则只在父项目展开以后才显示出来。

应用程序可以发送一条 TVM_SETIMAGELIST 消息，同时传递树形视窗句柄、图象列表句柄以及表 10-21 里列出的某个定义，从而把某个图象列表与树形视窗控件关联起来。

```
// associate the image list to the tree-view control
TreeView_SetImageList (hwndTree, himg, TVSIL_NORMAL);
```

表 10-21 用于展开或者折叠树形视窗节点的标志

图象列表类型	值	说 明
TVSIL_NORMAL	0	由一系列图象组成的普通图象列表
TVSIL_STATE	2	取得状态图象列表，其中包含了处于用户自定义状态时的树形视窗项目图象

通常情况下，与树形视窗关联起来的图象列表只包含了两幅图象，用于反映项目的当前状态：选定或者未选定。在缺省时，图象列表内的第一个图标代表未选定状态，而第二个则为很少出现的选定状态保留。插入一个新项目时，应用程序就会定义图标以哪种状态显示，方法是利用对应的标志（TVIF_IMAGE 或者 TVIF_SELECTEDIMAGE），从而分别激活 iSelectedImage 或者 iImage 项。然后对希望的图象索引进行分配，如下所示：

```
...
// inserting the include files in TREEVIEW
tvis.hParent = TVI_ROOT;
tvis.hInsertAfter = TVI_LAST;
    tvis.item.mask = TVIF_TEXT | TVIF_PARAM | TVIF_IMAGE |
        TVIF_SELECTEDIMAGE;
tvis.item.lParam = TRUE;
tvis.item.pszText = "WINDOWS.H";
tvis.item.cchTextMax = strlen("WINDOWS.H");
tvis.item.iImage = 0;
tvis.item.iSelectedImage = 1;
hTreeNew = TreeView_InsertItem(hwndTree, &tvis);
...
```

在 INCLUDE 这个例子里，一个分支首次展开的时候，树形视窗控件内的相应子项目就

会插入其中。通过对一条 TVM_SETITEM 消息的发送，把新项目插入分支以后，新项目与图象之间的关联处理才会进行：

```
...
// now we expanded the branch
tvi.mask = TVIF_PARAM | TVIF_IMAGE | TVIF_SELECTEDIMAGE |
    TVIF_HANDLE;
tvi.lParam = TRUE;
tvi.hItem = pnmtv -> itemNew.hItem;
tvi.iImage = 1;
tvi.iSelectedImage = 1;
TreeView_SetItem(hwndTree, &tvi);
...
```

一个分支实际展开以后（TVE_EXPAND），上面这个代码段就会从 TVN_EXPANDED 通知消息里插入。在那个时候，无论未选定图象还是选定图象都会设置成等效于一个打开的文件夹，即树形视窗图象列表内的第二幅图象。拦截了 TVE_COLLAPSE 以后（表明分支已经折叠起来了），两幅图象就会设置成零。这种方案也为我们带来了一种具有积极效果的副作用。为未选定和选定状态分配了相同的图象以后，我们就忽略了屏幕上保持的缺省树形视窗行为，从而为每种情况下不同的项目分配对应的图标。图 10-12 显示了 INCLUDE 程序在展

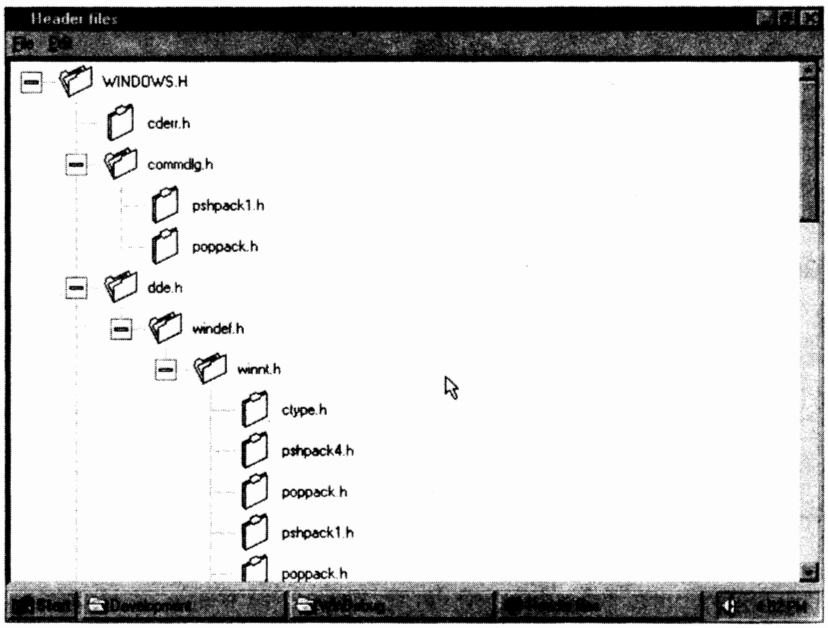


图 10-12　展开了一个父项目以后，它的图标就会发生变化，从而反映出子项目列表的存在。打开文件夹的图标也表明一个选定的展开父项目

开了几个父项目以后的情况。

10.6.6 通知代码

正如我们以前提到的那样，树形视窗控件通过发出一条 WM_NOTIFY 消息，从而把自己的通知代码传递给父窗口。对于这一类的窗口来说，lParam 是指向某个内存块的指针，这个内存块内包含了一个 NM_TREEVIEW 结构，其中填写了一些有用的信息：

```
typedef struct _NM_TREEVIEW
{
    NMHDR hdr;
    UINT action;
    TV_ITEM itemOld;
    TV_ITEM itemNew;
    POINT ptDrag;
} NM_TREEVIEW, *LPNM_TREEVIEW;
```

这个结构中的第一个项是另一个数据结构，常见的 NMHDR 信息块通常与一条 WM_NOTIFY 消息关联在一起。action 项内包含了特定于通知代码的行动标志，这种信息会根据通知代码的不同而发生变化。itemOld 项是一种 TV_ITEM 数据结构，它可以负载上一次选定项目的相关信息，而 itemNew 则指定了当前选定的项目。鼠标指针在客户区内的位置是用最后一个项 ptDrag 表达的。表 10-22 内列出了用于这个通用控件的所有通知代码。

表 10-22 树形视窗通用控件使用的通知代码

通知代码	值	说 明
TVN_SELCHANGING	(TVN_FIRST-1)	在当前的项目选定改变之前发出
TVN_SELCHANGED	(TVN_FIRST-2)	指出选定项目已发生了变化
TVN_GETDISPINFO	(TVN_FIRST-3)	用于通知应用程序，令其针对某个特定的项目提供相应的信息，从而在树形视窗控件内把这个项目显示出来
TVN_SETDISPINFO	(TVN_FIRST-4)	用于通知应用程序，用户已经完成了一个项目标签的编辑工作，将其设置成了 LPSTR_TEXTCALLBACK
TVN_ITEMEXPANDING	(TVN_FIRST-5)	在展开或者折叠一个项目节点之前发出
TVN_ITEMEXPANDED	(TVN_FIRST-6)	在展开或者折叠一个项目节点之后发出
TVN_BEGINDRAG	(TVN_FIRST-7)	指出一次标准的拖放操作已经开始了
TVN_BEGINRDRAG	(TVN_FIRST-8)	指出一次非标准的拖放操作已经开始了
TVN_DELETEITEM	(TVN_FIRST-9)	删除了一个项目以后，这条通知代码就会发送给树形视窗父窗口
TVN_BEGINLABELEDIT	(TVN_FIRST-10)	在项目标签的编辑工作开始之前发出
TVN_ENDLABELEDIT	(TVN_FIRST-11)	在项目标签的编辑工作结束以后发出
TVN_KEYDOWN	(TVN_FIRST-12)	通知父窗口用户已经按下了某个键

在表 10-22 列出的各种不同的通知代码中，TVN_SELCHANGING 和 TVN_SELCHANGED 扮演了最关键的角色。树形视窗内的一个选定改变之前以及在改变完成以后，这两条通知代码就会分别抵达某个窗口进程。通过对这两条通知代码的拦截，应用程序既可以防止出现未经许可的选定，也可以在某个特定项目被选定以后执行相应的代码段。对

于这两条通知代码来说，NM_TREEVIEW 数据结构里的 action 项会采用表 10-23 内列出的某个值。

表 10-23 NM_TREEVIEW 结构的 action 项使用的定义，用于指出导致选定发生改变的行动

用于 TVN_SELCHANGED 的标志	值	说 明
TVC_UNKNOWN	0x0000	导致树形视窗的选定发生改变的行动是未知的
TVC_BYMOUSE	0x0001	选定的改变是由按下一个鼠标按钮造成的
TVC_BYKEYBOARD	0x0002	选定的改变是通过一次键盘击键造成的

10.6.7 树形视窗项目的拖动

为了结束我们对树形视窗控件的讨论，接下来，我们准备对 INCLUDE 示范程序进行改造，为它增加一些拖动功能。通过这种改造，我们就有机会重新回顾支持拖放功能的一个树形视窗窗口在建立过程中需要涉及的所有步骤。

建立属于这种类型的一个窗口时，必须忍受住增加 TVS_DISABLEDRAGDROP 窗口风格的诱惑。假如增加了这个风格标志，树形视窗内对拖放操作的支持就会自动屏蔽起来。用户通过鼠标左键或者右键都可以选定一个项目，从而完成一次拖动操作的初始化处理。在这以后，只需在屏幕上稍微一下鼠标，树形视窗的父窗口进程就会接收到一条 TVN_BEGINDRAG 通知代码（假如是按下鼠标右键拖动的，则是 TVN_BEGINRDRAG 通知代码）。在这个时候，我们应该根据与图象列表那种基本规则一样的规则开始进行拖动处理，这在本章的前面部分已经讲述过了。

引用了拖动项目的树形视窗句柄会在 NM_TREEVIEW 结构里自动传递，这个句柄具体是存放在 itemNew. hItem 项里的。我们应该把这种信息保留在一个安全的地方，因为需要在新位置把它描绘出来的时候，就需要用到这种信息了。不幸的是，拖动操作结束以后，树形视窗的 API 没有提供任何可以利用工具查询出对象的源项目。因此，我决定声明一个 hdragged 句柄，使其在整体窗口进程里都是“可见”。显然，这需要将其分配成一个静态存储类。

随后，为了对拖动的项目进行表示，还需要两种补充信息：首先是与树形视窗关联在一起的图象列表的句柄，其次是目标图象的索引。通过对树形视窗控件进行简单的查询，TreeView_GetImageList () 函数就可以返回图象列表的句柄。INCLUDE 里的拖动图象与当前显示于选定项目左侧的那幅图象是对应的。通过调用 TreeView_GetItem () 函数，并将其中的 mask（屏蔽）项设置成 TVIF_IMAGE，TVIF_SELECTEDIMAGE 和 TVIF_HANDLE，我们就可以接收到当前显示的图标。ImageList_BeginDrag () 需要利用这两种数据对拖动模式进行准备。这个函数并不是什么新东西，我们在以前就曾经学过。

正如以前在 DRAGIL 例子里曾经做过的那样，我们仍然把整个屏幕当作拖动操作的物主窗口。这种选择强迫应用程序在把信息传递给 ImageList_DragEnter () 之前，先将任何鼠标位置转换成屏幕坐标系统。

```
...
case TVN_BEGINDRAG:
{
    HIMAGELIST himg;
    TV_ITEM tvi;

    // get the dragged item
    hdragged = pnmtv -> itemNew.hItem;
    // collapse it
    TreeView_Expand(hwndTree, hdragged, TVE_COLLAPSE);

    // create an image list
    himg = TreeView_GetImageList(hwndTree, TVSIL_NORMAL);

    tvi.mask = TVIF_IMAGE | TVIF_SELECTEDIMAGE | TVIF_HANDLE;
    tvi.hItem = hdragged;
    tvi.iImage = tvi.iSelectedImage = 0;
    TreeView_GetItem(hwndTree, &tvi);

    // Start the drag operation.
    ImageList_BeginDrag(himg, tvi.iImage, 0, 0);
    ClientToScreen(hwndTree, &pnmtv -> ptDrag);
    ImageList_DragEnter(NULL, pnmtv -> ptDrag.x, pnmtv -> ptDrag.y);
    // direct mouse input to the parent window
    SetCapture(GetParent(hwndTree));
    fDrag = TRUE;
}
    break;
```

从理论上说，用户拖动的既可以是一个展开节点，也可以是一个折叠节点，至少从视觉的角度来看是这样的。然而，我们对这种操作需要进行更加深入的考虑。举个例子来说，假如拖动操作结束了，现在应该把当前的分支从它的原始位置移动到目标位置。假如拖动的是一个折叠起来的节点，那么只需要移动一个单独的项目就可以了。相反，一个展开节点则意味着应用程序必须把所有项目从源位置移动到目标位置。

对于第二种情况，我们必须事先进行周密的计划。正如大家也许已经注意到的那样，这个类提供的所有消息中唯独没有 TVM_ITEMMOVE 消息。简而言之，这意味着拖动操作要求应用程序把与每个项目相关的所有信息都保存下来，以后再把它们插入树形分级结构的一个不同的位置。

除此以外，我们没有办法动态地改变一个项目的父项目，这样也许能节省下大量的时间。

假如可以对存储父项目句柄的内存位置进行访问，那么整个分支的移动也许就会变得相当容易。能够显示 hParent 信息的唯一场所是在 TV_INSERTSTRUCT 数据结构里。

表 10-18 为大家列出了以 TVM_GETNEXTITEM 消息为基础的 TreeView_GetParent () 宏函数。不幸的是，没有类似的函数可用于对父项目的句柄进行设置。所以把一个父项目从它的原始位置拖动离开以后，INCLUDE 程序里会发生什么情况呢？该程序怎样把项目放置到一个新地方呢？为简化这种算法，INCLUDE 程序强迫一个父项目在拖动操作刚开始进行的时候就把自己折叠起来。这种方法具有某些副作用。拖动的项目插入树形结构的新位置时，它就会丢失自己的所有子项目。cChildren 项总是设置成 1，而 lParam 则变成 FALSE。没有什么情况能强迫应用程序对头文件进行浏览，从而判断 .H 文件里不同的包容情况。那个节点一次成功的展开会被当作第一次展开尝试进行处理。这种方案的优点是不言而喻的，拖动操作的速度非常快。这种新特性与应用程序的整体方案非常匹配。事实上，EXPLORER.EXE 程序采用的也是和刚才讨论的一样的算法。假如拖动的是一个展开的文件夹，它在目标位置就是折叠起来的。

假如应用程序需要同时对所有子项目进行移动，就必须实现一种更为复杂的递归算法。不走运的是，树形视窗窗口类没有提供对子列表内所有项目进行计数的任何方式，这样就限制了开发者对其进行的附加处理。下面是一些基本的设计准则，它们对这种算法的实现也许会有所帮助。

10.6.8 算法的考虑

我们在这儿的想法是动态地分配一个内存块，使其有能力容纳几个 TV_ITEM 数据结构，每个子项目都分配一个结构。在递归例程里，我们应该通过一条 TVM_GETITEM 消息的发送，并且把返回的信息块存储到内存区域里，从而有选择地对每个项目进行查询。一旦整个分支都进行了成功的检查和存储，就可以开始把父项目插入一个新位置里。所有的子项目都应该重复相同的处理。开始这种代码段的时候，必须注意到所有项目都能通过一个树形视窗句柄相互之间进行引用。也要记住的一点是，假如把一个项目从树形分级结构里移走，并且插入一个新位置，那么信息就会失去其所有的含义。因此，动态分配的内存块应该包含 TV_ITEM 数据结构，这个数据结构是由一些信息集成到一起的，用于对不同项目之间的关系进行跟踪。

10.6.9 树形视窗控件的最后几点注意事项

应该随时记住的一点是，在实际编写任何代码之前，都应该事先拟定详细的计划，对希望在自己的应用程序里实现的所有特性做到心中有数。然后还要考虑性能问题。一次只插入一小块数据肯定能满足对用户操作的及时响应要求。最后，还要考虑到尽管树形视窗在数据结构、标志、消息以及通知代码方面具有高度的复杂性，需要开发者对大量数据进行设置，但是仍然要把一些经常用到的树形视窗行为与一些相应的代码集成到一起，从而令其具有大多数通用的特性，能够实现常见的功能。

10.7 列表视窗控件

列表视窗是一种功能强大的通用控件，它只能在系统外壳里使用。桌面本身就是一个列表视窗窗口，每个文件夹也是这样的一个窗口。这种类型的窗口能以四种方式显示项目，从技术角度讲，也就是能以四种视窗显示项目。项目的显示方式包括下面这些：

▶Large icons（大图标）

▶Small icons（小图标）

▶Lists（列表）

▶Details（详细列表）

系统桌面就是一个列表视窗窗口，它只能以大图标模式运行。文件夹是列表视窗类的一种更加灵活的实现方式，这是由于它能以多种方式实现信息的显示。如图 10-13 所示，其中显示了桌面和以详细列表模式显示的一个文件夹。

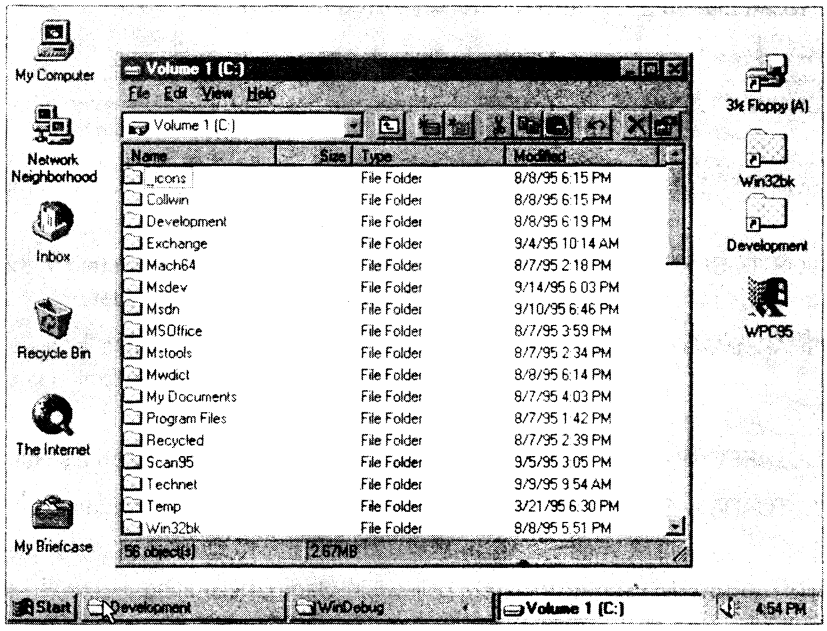


图 10-13　系统桌面是一个以大图标模式显示的列表视窗窗口，
而显示 C 驱动器情况的文件夹则处于详细列表视窗下

创建列表视窗所用的方案与树形视窗窗口的创建差不多。和往常一样，单单建立这样的一个窗口并不能实现对这种对象的完整管理。附加的工作需要我们对这种控件进行准备，使其正确地容纳信息和在屏幕上显示出来。表 10-24 为大家总结了这种通用控件支持的所有风格类型。

表 10-24　列表视窗通用控件使用的风格

列表视窗风格	值	说　明
LVS_ICON	0x0000	图标视窗
LVS_REPORT	0x0001	报表视窗
LVS_SMALLICON	0x0002	小图标视窗
LVS_LIST	0x0003	列表视窗
LVS_TYPEMASK	0x0003	用于标识一个列表视窗的当前视窗的标志值，这是通过对自己的专用内存区域进行检查得出的
LVS_SINGLESEL	0x0004	单一选择列表视窗

（续）

列表视窗风格	值	说明
LVS_SHOWSELALWAYS	0x0008	尽管控件没有处于活动状态，也显示选定高亮度
LVS_SORTASCENDING	0x0010	以第一栏内项目的正文为基础，按升序对项目排序
LVS_SORTDESCENDING	0x0020	以第一栏内项目的正文为基础，按降序对项目排序
LVS_SHAREIMAGELISTS	0x0040	控件消除以后，与列表视窗关联在一起的图象列表不会同时解散
LVS_NOLABELWRAP	0x0080	在图标视窗里，正文在一个单行内显示
LVS_AUTOARRANGE	0x0100	在图标和小图标视窗内，控件会对图标进行自动排列
LVS_EDITLABELS	0x0200	允许进行标签的现场编辑
LVS_NOSCROLL	0x2000	屏蔽列表视窗的滚动
LVS_ALIGNTOP	0x0000	项目沿着列表视窗控件的顶部排列，在图标和小图标视窗中适用
LVS_ALIGNLEFT	0x0800	在图标和小图标视窗中，项目左对齐
LVS_ALIGNMASK	0x0c00	用于对指定排列方式的窗口风格进行隔离
LVS_OWNERDRAWFIXED	0x0400	定义一个物主绘图列表视窗
LVS_NOCOLUMNHEADER	0x4000	防止栏标题在报表视窗中显示出来
LVS_NOSORTHEADER	0x8000	指出栏标题不能像按钮那样工作

四种视窗模式分别与最开始那四种风格对应，缺省情况下使用的是LVS_ICON（大图标模式）。通过对表10-24进行检查，大家会发现有些菜单项提供了一个文件夹弹出式菜单。比如，Arrange Icons/Auto Arrange选项就是在列表视窗内存区域里设置了一个LVS_AUTOARRANGE风格的结果。如图10-14所示。

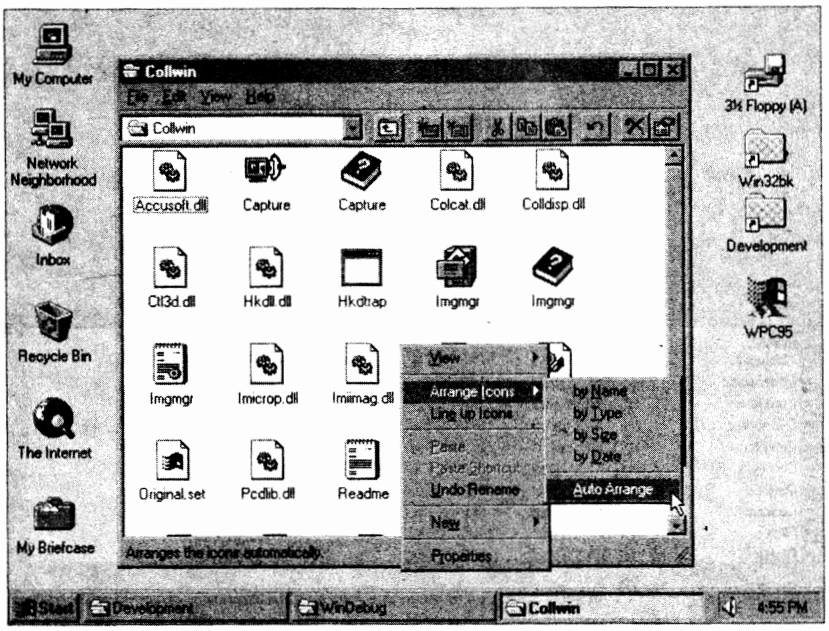


图10-14　假如选中了对应的菜单项，文件夹的项目就会随时自动排列。这表明在窗口预约内存区域里使用了LVS_AUTOARRANGE标志

一些风格看起来很面熟，因为它们反映了由树形视窗窗口类提供支持的某些风格。除此以外，一些风格只能应用于详细列表模式下的列表视窗控件。事实上，只有当一个列表视窗

提供了与命名对象有关的附加细节时，栏标题才会在窗口的顶部显示出来。Find 外壳组件显示了在一个详细列表视窗控件里找到的对象，如图 10-15 所示。

假如我们不想让处于详细列表模式下面的某个列表视窗控件显示出栏标题，可以在建立这个控件的时候增加 LVS_NOCLUMNHEADER 风格。这种风格对于不同模式下的列表视窗是没有效果的。

栏标题在列表视窗控件里提供了一个附加的特性。假如按下这种标题按钮，列表视窗就会根据存储于那一栏内的值对所有项目进行排序。这种行为要求应用程序提供一个比较例程，以便列表视窗控件调用，从而一次对两个项目进行比较。某些情况下，我们也许会希望栏标题显示出来，但是不是实现相应的排序功能。为了避免用户单击一个栏标题以后，控件内却什么情况也没有发生，最好增加一个 LVS_NOSORTHEADER 风格，从而禁止这种功能（事实上，它只是简单地不生成 LVN_COLUNMCLICK 通知代码）。

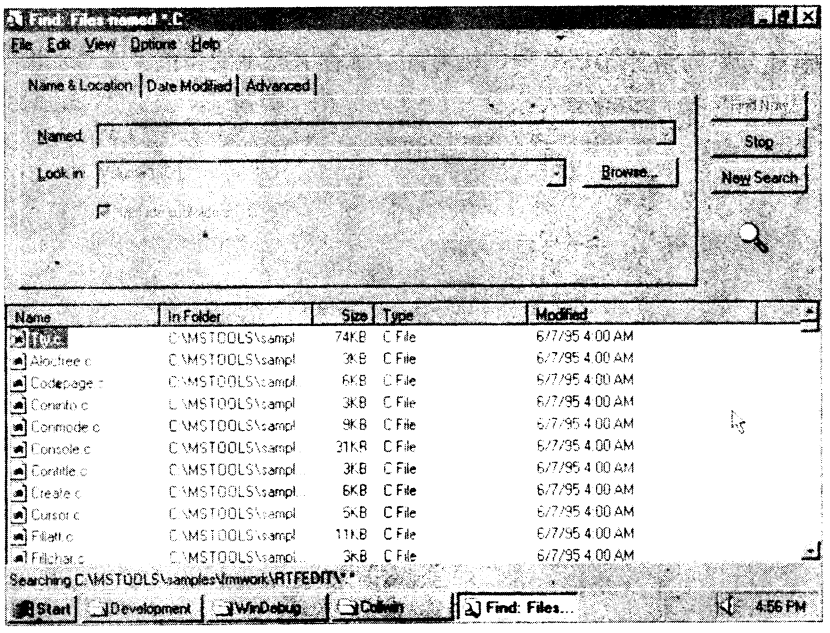


图 10-15　Find 外壳组件显示了一个列表视窗，其中显示了与查找标准匹配的所有名字

10.7.1　列表视窗控件的建立

就我个人来说，我认为把原始的列表视窗控件以及它的初始化定制信息放置在一个单独的函数里比较合适，这个函数就是 CreateListView ()。为了向大家展示列表视窗控件具备的所有基本功能，我建立了 EUROPE 这个示范程序。这是一个简单的应用程序，它提供关于欧盟 (UI) 的信息。欧盟是由一些欧洲国家联合组建起来的，以前叫作“欧洲经济委员会”(EEC)。EUROPE 应用程序列出了欧盟的 15 个成员国，提供了与其中每个国家有关的一些信息。图 10-16 向大家展示了六个发起国的名单。

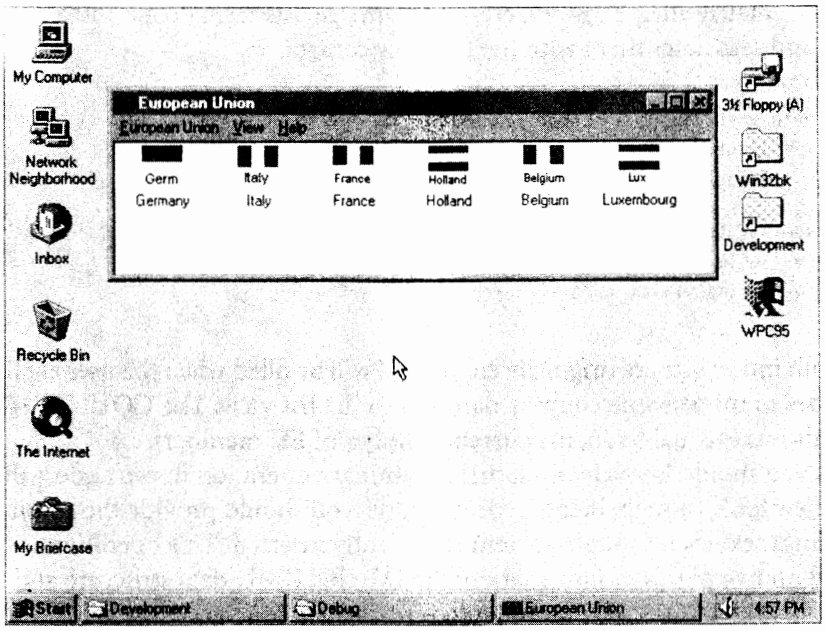


图 10-16　EUROPE 示范程序显示了最初提议组成欧盟的六个国家

EUROPE 提供了一个普通的叠置式窗口，它的客户区已被一个列表视窗控件覆盖起来了。窗口的建立过程相当简单，如下所示：

```
...
// create the list view window
hwndLV = CreateWindowEx(WS_EX_CLIENTEDGE,
                        WC_LISTVIEW, NULL,
                        WS_CHILD | WS_VISIBLE | LVS_REPORT,
                        0, 0, 0, 0,
                        hwndPapa,
                        (HMENU)CT_LISTVIEW,
                        hInstance,
                        NULL);
...
```

CreateListView () 例程是在主窗口进程拦截了 WM_CREATE 消息以后调用的。最初的窗口没有分配具体的显示尺寸，它的宽度和高度都设置成 0。和往常一样，我们以后会在 WM_SIZE 消息里对此进行设置。

hwndPapa 句柄引用的是叠置式窗口。建立一个列表视窗的时候，必须指定四个视窗标志中的一个。在这种特定的情况下，列表视窗支持的是 LVS_REPORT 风格，也就是详细列表模式。

就在这以后，EUROPE 紧接着建立了两个图象列表，一个规模比较小，一个则属于中等水平，然后把它们与列表视窗控件关联起来。

```
...
// create a small and a normal image list
hSmall = ImageList_Create(16, 16, ILC_MASK, 0, COUNTRIES);
hLarge = ImageList_Create(32, 32, ILC_MASK, 0, COUNTRIES);

// Associate the image list with the list view
ListView_SetImageList(hwndLV, hSmall, LVSIL_SMALL);
ListView_SetImageList(hwndLV, hLarge, LVSIL_NORMAL);
...
```

这两个图象列表最初都是置空的，用户决定把某些国家名字插入列表视窗以后，我们才对其进行填充。COUNTRIES 定义设置成 15，这是当前欧盟的成员国数量。

大家现在也许已经明白了这一点，这种准备工作并不会对列表视窗的外观产生影响。然而，在详细列表模式下，我们应该先提供栏标题正文，尽管此时还没有插入的项目。这种操作是以两种信息为基础的：LV_COLUMN 数据结构以及 LVM_INSERTCOLUMN 消息。

```
typedef struct _LV_COLUMN
{
    UINT mask;
    int fmt;
    int cx;
    LPSTR pszText;
    int cchTextMax;
    int iSubItem;
} LV_COLUMN;
```

其中，mask 项作为相当于 LV_COLUMN 数据结构整体功能方案中的一个中枢。表 10-25 里列出的各种标志定义了向一个列表视窗控件传递 LV_COLUMN 内存块时应该考虑进去的设置项。

表 10-25　用于 LV_COLUMN 数据结构 mask 项的标志定义

列表视窗的 LV_COLUMN 标志	值	说　明
LVCF_FMT	0x0001	使 fmt 项有效
LVCF_WIDTH	0x0002	使 cx 项有效
LVCF_TEXT	0x0004	使 pszText 项有效
LVCF_SUBITEM	0x0008	使 iSubItem 项有效

在EUROPE程序里，所有这四个标志都是处于活动状态的，正如下面这个代码段向我们展示的那样：

```
...
// prepare the information for the column headings
lvc.mask = LVCF_FMT | LVCF_WIDTH | LVCF_TEXT | LVCF_SUBITEM;
// left align the column
lvc.fmt = LVCFMT_LEFT;
// width of the column, in pixels
lvc.cx = 80;
lvc.pszText = szText;
...
```

其中，正文是以左对齐方式排列的，每一栏的宽度为75个象素。表10-26为我们总结了针对LV_COLUMN数据结构的fmt项可以选择的所有值。

表10-26 LV_COLUMN的fmt项使用的所有标志

用于fmt项的列表视窗标志	值	说 明
LVCFMT_LEFT	0x0000	栏内正文左对齐排列（第一栏必须如此）
LVCFMT_RIGHT	0x0001	栏内正文右对齐排列
LVCFMT_CENTER	0x0002	栏内正文居中排列
LVCFMT_JUSTIFYMASK	0x0003	用于定义当前排列方式的标志

在这种情况下，我决定为所有栏都提供一个通用的宽度，但这并不是必须的，其他程序应该根据具体情况具体考虑。栏标题正文串存放于EUROPE.RC文件的STRINGTABLE资源里。应用程序提供了六个栏（NUM_COLUMNS），分别是由后续的ID定义的。在这儿，循环结构是用于载入它们的最常见手段，最后用LV_COLUMN项补全了所有不足的信息。每一栏都标识了特定的信息，这些信息与插入列表视窗控件内的项目是有关的。

列表视窗类为插入的项目分配了连续的ID编号，这些编号都是从零开始的。就这一点来说，我们能在它上面找到一点标准列表框的影子。它们之间的区别在于当列表视窗处于详细列表模式时，其中还会存在一些子项目。现在让我们澄清这个问题。列表视窗内包含了一系列项目，所有项目都是用一个连续的ID标识的。在详细列表模式下面时，每个项目都提供了一个或者多个子项目，它们的ID等价于栏的位置。在EUROPE这个例子里，假如处于详细列表模式时，第一栏会标记为Nation，后台的栏目分别是Captial，Sport等等。Nation栏下面显示的国家名称对应于子项目0；Captial下面的一个对象对应于子项目1；Sport下面的则对应于子项目2；以此类推。LV_COLUMN结构内的iSubItem就是根据这种方案定义的，其中包含了对应的位置ID。

```
...
// add the columns
```

```
for(i = 0; i <= NUM_COLUMNS; i++)
{
    lvc.iSubItem = i;
    // column names
    lvc.cchTextMax = LoadString(hInstance, ST_NATION + i, szText,
        sizeof(szText));
    if(ListView_InsertColumn(hwndLV, i, &lvc) == -1)
        return NULL;
}
...
```

一旦插入了所有栏标题以后，CreateListView () 例程就会返回列表视窗的句柄。另外一个例程 FillTheListView () 则会对列表视窗的对象填写进行管理，这是用户在主菜单内进行了选择以后发生的。如图 10-7 所示。

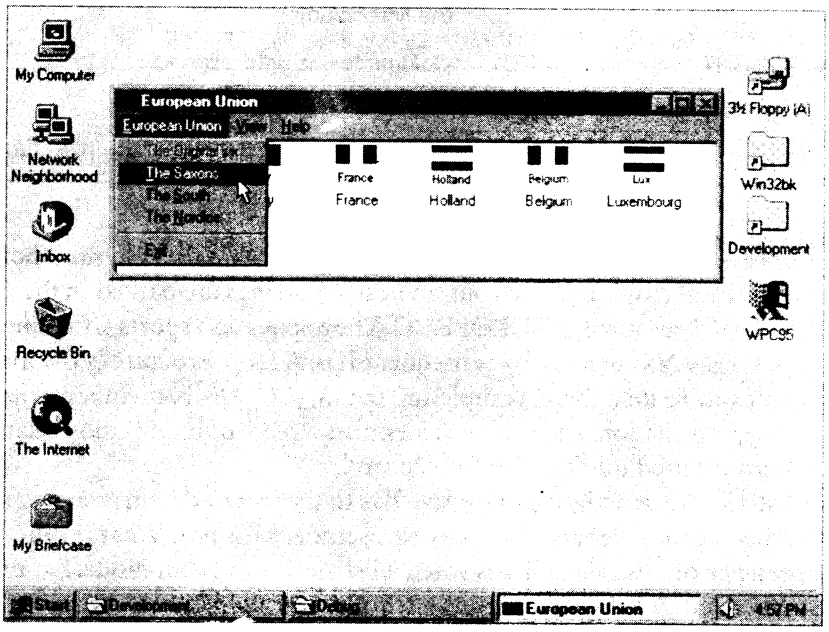


图 10-17　欧盟成员国名称的插入需要由 European Union
顶级菜单内的菜单项进行调整

接下来，假如用户选择的是 Original six（最初的六个国家）菜单项。此时，应用程序会调用 FillTheListView () 例程，同时传递菜单项 ID（wParam 的低字）、准备插入的项目数目（ORIGINALSIX）、用于存储国名信息的 STRINGTABLE 资源起始位置开始计算的偏移（0）、列表视窗的句柄（hwndLV）以及类型为 EUROPE 的一个内存块的地址（eu）。在 EUROPE 类型的内存块里，我们存放了准备在控件内插入的信息。

```
...
case MN_THEORIGINALSIX:
{
    FillTheListView(LOWORD(wParam), ORIGINALSIX, 0, hwndLV, eu);
}
    break;
...
```

EUROPE 数据结构是由六个设置项组成的——对应于列表视窗窗口内的每一栏。

```
typedef struct tagEUROPE
{
    char szCountry[20];
    char szCapital[20];
    char szCurrency[10];
    char szSport[20];
    char szLanguage[20];
    char szPop[10];
} ERUOPE, *PEUROPE;
```

这个数据结构主要设计用于支持栏项目排序，但是它也可以用于实现我们从技术上所说的“回调项目”。这个术语是指避免在列表视窗控件内插入一个项目的正文的能力，而用 LPSTR_TEXTCALLBACK 定义来替换它。假如需要显示这样的一个项目，列表视窗控件就会向应用程序发出对它的请求。

如果想插入一个新项目，我们需要用到 LV_ITEM 数据结构以及 LVM_INSERTITEM 消息。这个结构看起来与 TV_ITEM 颇有类似之处。它们的逻辑都是一致的，都要以分配给 mask 项的标志为基础。这些标志的完整清单请大家参考表 10-27。

```
typedef struct _LV_ITEM
{
    UINT mask;
    int iItem;
    int iSubItem;
    UINT state;
    UINT stateMask;
    LPSTR pszText;
    int cchTextMax;
    int iImage;
    LPARAM lParam;
```

} LV_ITEM;

表 10-27 LV_ITEM 数据结构的 mask 项使用的标志

用于 mask 项的列表视窗标志	值	说 明
LVIF_TEXT	0x0001	激活 pszText 和 cchTextMax 项
LVIF_IMAGE	0x0002	激活 iImage 项
LVIF_PARAM	0x0004	激活 lParam 项
LVIF_STATE	0x0008	激活 state 和 stateMask 项

表 10-28 为大家总结了 LV_ITEM 数据结构里 state 和 stateMask 项使用的所有标志，它们反映了可能存在的项目状态。

表 10-28 LV_ITEM 数据结构 state 和 stateMask 项使用的标志

列表视窗项目状态标志	值	说 明
LVIS_FOCUSED	0x0001	项目拥有了视觉焦点
LVIS_SELECTED	0x0002	项目已被选中
LVIS_CUT	0x0004	项目成为一个剪切操作的目标
LVIS_DROPHILITED	0x0008	项目作为一个拖放目标高亮显示
LVIS_OVERLAYMASK	0x0F00	隔离了状态位，这个状态位包含了重叠图象的以 1 为基础的索引
LVIS_STATEIMAGEMASK	0xF000	隔离了状态位，这个状态位包含了状态图象的以 1 为基础的索引

在 FillTheListView () 例程里，应用程序声明了一个属于类型 LV_ITEM 的 lvi 标识符，并且对其进行了相应的初始化。首先，应用程序需要取回插入控件内的项目总数，这一工作是通过调用 ListView_GetItemCount () 宏函数为完成的。这个宏函数的基础是 LVM_GETITEMCOUNT 消息。随后，我们再利用返回值判断新项目的 ID，如下所示：

```
...
// get the current # of items in the listview
iCnt = ListView_GetItemCount (hwndLV);
...
```

接下来，我们再从资源文件里载入用于代表国家标志的图标，然后把它们插入与列表视窗控件关联在一起的图象列表里。

pszText，cchTextMax，iImage，lParam，state 和 statMask 项里都包含了一些有用的信息，利用这些信息可以把一个新项目顺利地插入列表视窗内。非常基本的一个操作是：我们必须为 iItem 项分配恰当的 ID，同时强迫 iSubItem 项等于 0，这个值的用途是对一个栏目进行标识。事实上，假如列表视窗控件当时正处于详细列表模式下，一个项目的插入进程会分隔成两个单独的步骤。首先，我们需要插入项目标签以及与这个标签关联在一起的那个图标，这种操作是通过调用 ListView_InsertItem () 函数实现的。接下来，以上一次对新项目当前位置进行指定的操作为基础，利用那次操作的返回值，我们还要增加与子项目（从第一号开始）对应的正文。刚才讲述的第二个插入进程需要涉及到 ListView_SetItemText () 宏函数。

在 lParam 项里，我们在其中存储了一个特殊的指针，令其指向 EUROPE 程序某个内存

块的实际位置。在这个内存块内，我们可以根据当时的情况存放一些适当的信息。这两个插入进程都是在一个循环结构里完成，需要循环的次数正好是 iItems 次。iItems 这个标识符内存储了插入列表视窗内的国家的数目。

```
...
lvi.mask = LVIF_TEXT | LVIF_IMAGE | LVIF_PARAM | LVIF_STATE;
lvi.state = 0;
lvi.stateMask = 0;

for(i = 0; i < iItems; i++)
{
    ...
    // filling the LV_ITEM structure
    lvi.iItem = i + iCnt;
    lvi.iSubItem = 0;
    // loading the country name
    lvi.cchTextMax = LoadString(hInstance,
                                ST_NATIONS + iPrevItems + i,
                                szString,
                                sizeof(szString));
    lvi.pszText = szString;

    lvi.iImage = iImage;
    lvi.lParam = (LPARAM)(peu + i + iCnt);

    // save the country name
    lstrcpy(peu[iCnt + i].szCountry, szString);

    if((iPos = ListView_InsertItem(hwndLV, &lvi)) == -1)
            return FALSE;
...
```

对于进入的每个项目来说，都需要五个子项目来完成整个操作。在这儿，我们可以使用一个新的 for 循环，让这个循环以整数标识符 iSubItem 为基础，从而完成我们的目标。从资源文件里的某个对应位置载入了子项目的正文串以后，应用程序会调用 ListView_SetItemText () 函数，从而强迫每个单独的标签进入列表视窗控件内。

```
...
for(iSubItem = 1; iSubItem <= NUM_COLUMNS; iSubItem++)
```

```
{
    // load some country specific information
    LoadString(hInstance,
                    (ST_NATIONS + iPrevItems+i) * 10+iSubItem-1,
                    szString,
                    sizeof(szString));
    ListView_SetItemText(hwndLV,iPos,iSubItem,szString);

    switch(iSubItem)
    {
        case 1:     // Capital
            lstrcpy(peu[iCnt + i].szCapital, szString);
            break;

        case 2:     // Currency
            lstrcpy(peu[iCnt + i].szCurrency, szString);
            break;

        case 3:     // Sport
            lstrcpy(peu[iCnt + i].szSport, szString);
            break;

        case 4:     // Language
            lstrcpy(peu[iCnt + i].szLanguage, szString);
            break;

        case 5:     // Population
            lstrcpy(peu[iCnt + i].szPop, szString);
            break;
    }
}
...
```

这个内部 for 循环的最后一部分与列表视窗控件的结构化组件并没有什么关系，增加它的目的是为对栏目排序提供支持。应用程序为插入列表视窗内的每个项目都保存了一个 EUROPE 数据结构，主要用它拷贝控件内显示的信息。

一个菜单项选中以后，它就会自动屏蔽起来，从而使相同的信息不会同时插入列表视窗窗口两次。如图 10-18 所示。

10.7.2 视窗的改变

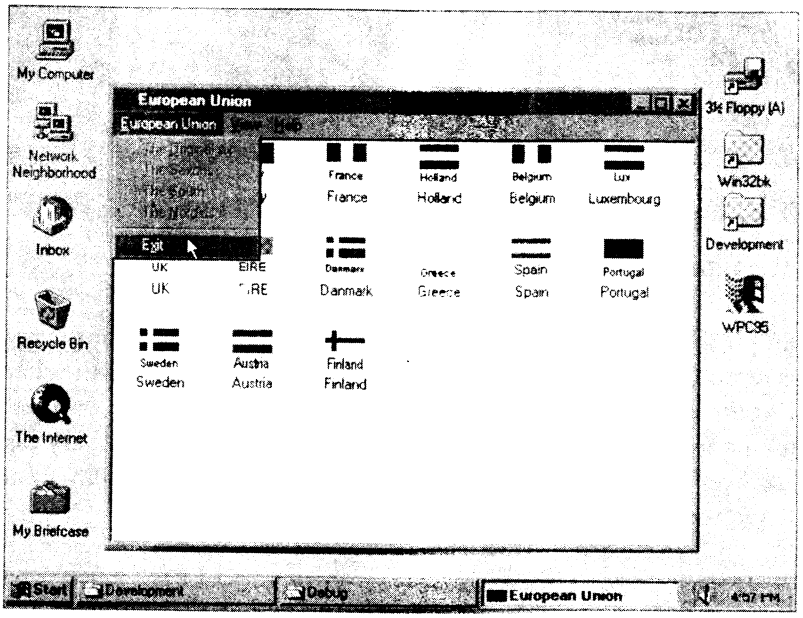


图 10-18　15 个国家全部插入列表视窗控件以后，European Union 菜单内的所有菜单项都屏蔽起来了

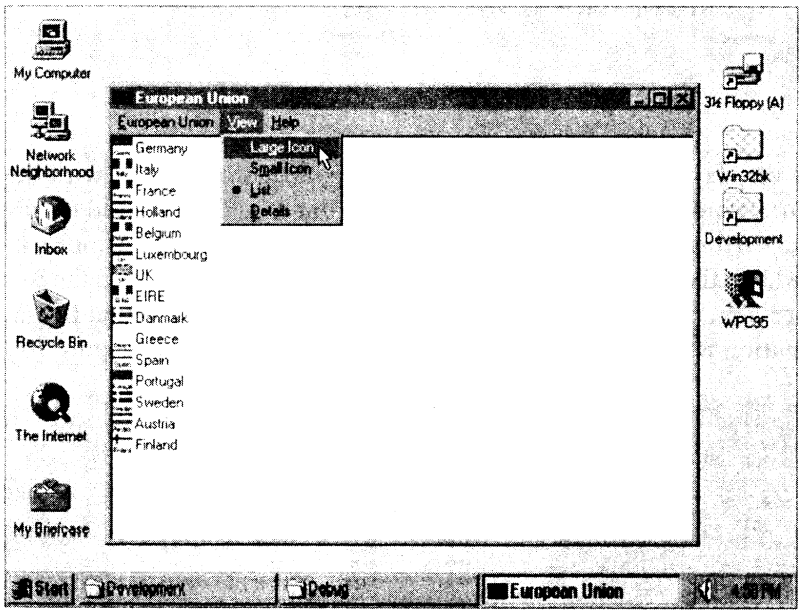


图 10-19　EUROPE 应用程序里的 View 下拉式菜单

EUROPE 这个示范程序在自己的列表视窗窗口里填充了与欧盟成员有关的所有信息，另外还加上对这些成员国进行唯一性标识的一些子项目。选择不同的视窗模式时，这些子项目

数据在列表视窗控件里是看不到的。View 顶级菜单允许用户在应用程序支持的四种视窗模式中进行选择，如图 10-19 所示。

只需简单地选择 Large Icon（大图标）菜单项，列表视窗变会相应地变成图 10-20 的样子。注意，由于这是一种单选菜单项，所以选中菜单项的左侧增加了一个圆点。

为了实现从一种视窗到另外一种视窗的切换，我们需要采取的方法是很简单的。只需要在列表视窗内存区域里设置相应的 LVS_风格就可以了，如下面这个代码段所示：

```
...
case MN_LARGEICON:
{
    DWORD dwStyle;

    dwStyle = GetWindowLong(hwndLV, GWL_STYLE);

    if((dwStyle & LVS_TYPEMASK) ! = LVS_ICON)
    {
        SetWindowLong(hwndLV, GWL_STYLE, (dwStyle & ~LVS_
                    TYPEMASK) | LVS_ICON);
        // activate the right radio menu item
        CheckMenuRadioItem(hmenu, MN_LARGEICON, MN_DETAILS,
                        LOWORD(wParam), MF_BYCOMMAND);
    }
}
    break;
...
```

用户选中了 Large Icon 菜单项以后，应用程序就会取回窗口预约内存区域内存放的风格位，检查 LVS_ICON 风格标志的存在。假如这个标志不在那儿，它就会增添到现成的窗口风格池里，然后为选中的项目指定圆点核选符号，从而完成对下拉式菜单外观的修改。这就是为了从一个视窗切换到另外一个视窗，我们需要完成的所有工作。

10.7.3 列表视窗的消息

列表视窗类提供了数目众多的消息，这些消息反映出开发者在实现这一类型窗口时是相当灵活的，可以有很多种选择。除了用于插入一个项目、提取相应正文、计算现成项目数量以及获取/设置颜色信息的的基本消息以外，另外还有一些消息可用于拖动操作、栏控制以及项目属性的动态修改等等。表 10-29 为大家总结了所有这些 LVM_消息。

LVM_GETNEXTITEM 消息显得特别灵活，这是由于表 10-30 内列出的那些 LVNI_标志的缘故。通过多次发出 LVM_GETNEXTITEM 消息，我们就可以对列表视窗的内容进行彻底的探索。这种“探索”可以向任何方向“前进”，对应的标志则是 LVNI_ABOVE，LVNI_BELOW，LVNI_TOLEFT 和 LVNI_TORIGHT 等。甚至还可以直接跳转至某个特定的项目

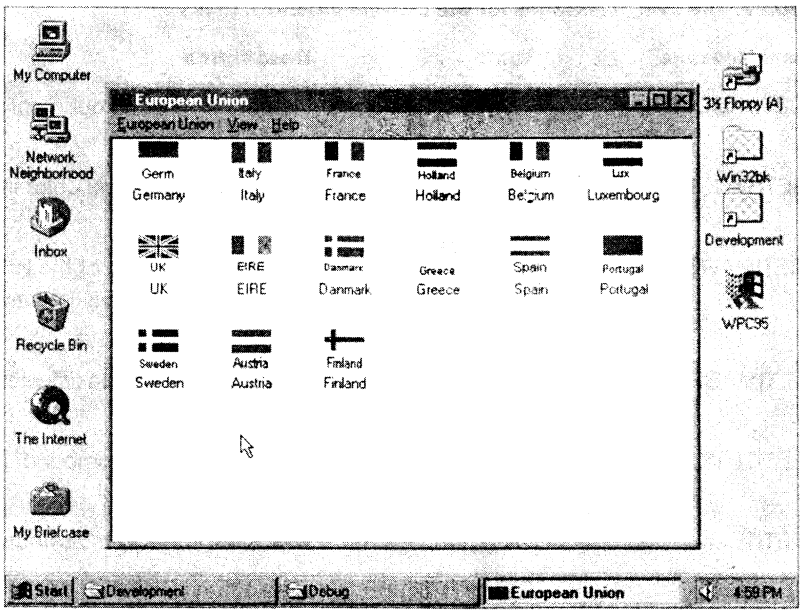


图 10-20　列表视窗控件现在已经处于大图标模式下了

(LVNI_SELECTED 和 LVNI_FOCUSED)。

表 10-29　列表视窗控件类使用的 LVM 消息

列表视窗消息	值	说　明
LVM_GETBKCOLOR	(LVM_FIRST + 0)	返回列表视窗控件的背景颜色
LVM_SETBKCOLOR	(LVM_FIRST + 1)	设置列表视窗控件的背景颜色
LVM_GETIMAGELIST	(LVM_FIRST + 2)	返回与一个列表视窗控件关联在一起的图象列表的句柄
LVM_SETIMAGELIST	(LVM_FIRST + 3)	把一个图象列表与列表视窗关联到一起
LVM_GETITEMCOUNT	(LVM_FIRST + 4)	返回列表视窗控件内的项目总数
LVM_GETITEM	(LVM_FIRST + 5)	取回关于项目的一些属性
LVM_SETITEM	(LVM_FIRST + 6)	设置项目的一些属性
LVM_INSERTITEM	(LVM_FIRST + 7)	在列表视窗控件内插入一个新项目
LVM_DELETEITEM	(LVM_FIRST + 8)	在列表视窗控件内删除一个现成项目
LVM_DELETEALLITEMS	(LVM_FIRST + 9)	删除列表视窗控件内的所有项目
LVM_GETCALLBACKMASK	(LVM_FIRST + 10)	指出什么输出信息是由应用程序内部控制的
LVM_SETCALLBACKMASK	(LVM_FIRST + 11)	定义什么输出信息将由应用程序提供，而不是由控件本身提供
LVM_GETNEXTITEM	(LVM_FIRST + 12)	返回一个列表视窗项目的索引编号，这个项目具有指定的属性，并且和一个指定项目具有特定的关系，请参考表 10-30
LVM_FINDITEM	(LVM_FIRST + 13)	对列表视窗的内容进行检查，查看是否有一个项目符合指定的特征
LVM_GETITEMRECT	(LVM_FIRST + 14)	取回当前视窗内一个项目全部或者部分使用的限制矩形，参考表 10-30

（续）

列表视窗消息	值	说明
LVM_SETITEMPOSITION	(LVM_FIRST + 15)	把一个项目移至列表视窗控件内指定的位置，只有在图标和小图标视窗中才有效
LVM_GETITEMPOSITION	(LVM_FIRST + 16)	取得当前项目在控件窗口内的当前位置
LVM_GETSTRINGWIDTH	(LVM_FIRST + 17)	返回列表视窗控件内一个正文串的宽度
LVM_HITTEST	(LVM_FIRST + 18)	判断鼠标指针是否正好位于列表视窗控件的一个特定位置处
LVM_ENSUREVISIBLE	(LVM_FIRST + 19)	强迫一个特定的项目在列表视窗控件内部分或者完全可见
LVM_SCROLL	(LVM_FIRST + 20)	对列表视窗控件的内容进行滚动
LVM_REDRAWITEMS	(LVM_FIRST + 21)	对一定范围内的项目进行强制重画
LVM_ARRANGE	(LVM_FIRST + 22)	根据表 10-32 内定义的某个标志，从而按照不同的规范对列表视窗控件内的项目进行排列
LVM_EDITLABEL	(LVM_FIRST + 23)	进入一个列表视窗项目的现场编辑状态
LVM_GETEDITCONTROL	(LVM_FIRST + 24)	返回现场编辑中涉及到的编辑窗口的句柄
LVM_GETCOLUMN	(LVM_FIRST + 25)	用指定栏内的所有信息填写一个 LV_COLUMN 数据结构
LVM_SETCOLUMN	(LVM_FIRST + 26)	设置一个列表视窗栏目的属性
LVM_INSERTCOLUMN	(LVM_FIRST + 27)	插入一个新栏
LVM_DELETECOLUMN	(LVM_FIRST + 28)	删除一个旧栏
LVM_GETCOLUMNWIDTH	(LVM_FIRST + 29)	返回当前栏的宽度
LVM_SETCOLUMNWIDTH	(LVM_FIRST + 30)	设置栏宽
LVM_CREATEDRAGIMAGE	(LVM_FIRST + 33)	建立一幅图象，用于显示拖动过程中鼠标指针在屏幕上的形状
LVM_GETVIEWRECT	(LVM_FIRST + 34)	在图标或者小图标模式下，取回列表视窗控件内所有项目的限制矩形
LVM_GETTEXTCOLOR	(LVM_FIRST + 35)	返回正文颜色
LVM_SETTEXTCOLOR	(LVM_FIRST + 36)	设置正文颜色
LVM_GETTEXTBKCOLOR	(LVM_FIRST + 37)	返回背景正文颜色
LVM_SETTEXTBKCOLOR	(LVM_FIRST + 38)	设置背景正文颜色
LVM_GETTOPINDEX	(LVM_FIRST + 39)	在列表或者详细列表模式下面时，返回最顶部可见项目的索引编号
LVM_GETCOUNTPERPAGE	(LVM_FIRST + 40)	在列表或者详细列表模式下面时，返回列表视窗控件在垂直方向上的项目总数
LVM_GETORIGIN	(LVM_FIRST + 41)	返回一个列表视窗控件的当前视窗起点
LVM_UPDATE	(LVM_FIRST + 42)	可视化地更新一个列表视窗项目
LVM_SETITEMSTATE	(LVM_FIRST + 43)	改变一个项目的状态
LVM_GETITEMSTATE	(LVM_FIRST + 44)	返回一个项目的状态
LVM_GETITEMTEXT	(LVM_FIRST + 45)	返回一个项目的正文
LVM_SETITEMTEXT	(LVM_FIRST + 46)	设置一个项目的正文
LVM_SETITEMCOUNT	(LVM_FIRST + 47)	通知一个列表视窗控件它不久会接收到大量项目
LVM_SORTITEMS	(LVM_FIRST + 48)	以应用程序定义的比较例程为基础，对列表视窗控件内的项目进行排序
LVM_SETITEMPOSITION32	(LVM_FIRST + 49)	用 32 位坐标设置一个项目在列表视窗控件内的位置
LVM_GETSELECTEDCOUNT	(LVM_FIRST + 50)	返回列表视窗控件内选定项目的总数
LVM_GETITEMSPACING	(LVM_FIRST + 51)	返回列表视窗控件内各项目之间的间隔距离
LVM_GETISEARCHSTRING	(LVM_FIRST + 52)	返回列表视窗控件的增量检索串

表 10.30 LVM_GETNEXTITEM 消息的标志总结

列表视窗的 LVM_GETNEXTITEM 标志	值	说 明
LVNI_ALL	0x0000	缺省值
LVNI_FOCUSED	0x0001	项目使用了 LVIS_FOCUSED 状态标志
LVNI_SELECTED	0x0002	项目已经设置了 LVIS_SELECTED 状态标志
LVNI_CUT	0x0004	项目已经设置了 LVS_CUT 状态标志
LVNI_DROPHILITED	0x0008	项目已经设置了 LVIS_DROPHILITED 状态标志
LVNI_ABOVE	0x0100	查找位于当前项目以上的一个项目
LVNI_BELOW	0x0200	查找位于当前项目以下的一个项目
LVNI_TOLFT	0x0400	查找位于当前项目左侧的一个项目
LVNI_TORIGHT	0x0800	查找位于当前项目右侧的一个项目

对于更加复杂的应用程序来说，也许需要判断由某些项目组件占据的表面区域，这些组件包括图标或者正文标签等等。LVM_GETITEMRECT 可以帮助我们返回这种信息，返回的信息则存储于一个 RECT 数据结构里。表 10-31 为大家列出了用于这条消息的所有标志。

表 10-31 用于 LVM_GETITEMRECT 消息的标志

列表视窗的 LVM_GETITEMRECT 消息	值	说 明
LVIR_BOUNDS	0	用包含了图标和标签的限制矩形填写 RECT 结构
LVIR_ICON	1	只返回图标的限制矩形
LVIR_LABEL	2	只返回标签的限制矩形
LVIR_SELECTBOUNDS	3	返回整个区域，用户在这个区域内单击便可选中一个项目

通过发送一条 LVM_ARRANGE 消息，应用程序可以强迫图标重新排列，对应的定义如表 10-32 所示。

表 10-32 LVM_ARRANGE 消息使用的排列标准

用于 LVM_ARRANGE 列表视窗标志	值	说 明
LVA_DEFAULT	0x0000	按照当前的排列风格对项目进行排列
LVA_ALIGNLEFT	0x0001	沿着左边界排列项目
LVA_ALIGNTOP	0x0002	沿着顶部边界排列项目
LVA_SNAPTOGRID	0x0005	把项目拉到最近的网格位置

10.7.4 项目的比较

在 EUROPE 程序里，假如用鼠标单击一个栏标题，项目就会进行相应的排序。这种行动生成了一条 WM_NOTIFY 消息，这条消息会发送给列表视窗控件的父窗口。在 WM_NOTIFY 消息里同时还会携带一条 LVN_COLUMNCLICK 通知代码（如果想了解这种通知代码的一份完整清单，可以参考表 10-36）。列表视窗的 LVM_SORTITEMS 消息可以对排序工作进行管理。实际上，这条消息只是简单地定义了用于实现排序算法的整体规范，其中也涉及到由应用程序定义的回调函数。这种回调函数可以与列表视窗控制进行交互作用，从而完成最终的排序工作。在 EUROPE 程序里，ListViewCompareProc () 就是由应用程序自己定义的一个回调例程，调用它可以实现项目的排序。在 LVM_SORTITEMS 这条消息里，我

们要设置回调例程的地址，另外还有应用程序定义的值。每次调用这个回调函数的时候，就会同时传递这些值。正如大家在下面这个代码段里看到的那样，EUROPE 程序传递了用于指定与选中栏目对应的那个子项目的 ID。这个 ID 的传送是很关键的，因为整个算法都要以那种排序标准作为基础。

```
...
// sort the item based on the column selection
ListView_SortItems (hwndLV, ListViewCompareProc, pnmlv -> iSubItem);
...
```

正如我们以前讨论树形视窗类时已经知道的那样，比较例程要利用应用程序定义的值(在这种情况是栏目 ID)，从而把两个项目集成到一起。这样一来，比较例程就可以接收到两个项目集成到一起所生成的那个 lParam 内存区域里存储的信息。在 lParam 里，EUROPE 程序存储了一个名为 EUROPE 的数据结构，其中存放了所有子项目数据。通过对这个区域的访问，假如第一个项目应该在第二个项目之后，排序例程就可以返回一个正值；假如第一个项目应该在第二个之前，就会返回一个负值；假如两个项目属于同一级别，就返回一个零值。

```
...
int CALLBACK ListViewCompareProc (LPARAM lParam1, LPARAM lParam2,
LPARAM lParamSort)
{
    EUROPE *peu1 = (EUROPE *) lParam1;
    EUROPE *peu2 = (EUROPE *) lParam2;
    LPSTR pStr1, pStr2;
    int iResult;

    if (peu1 && peu2)
    {
        switch (lParamSort)
        {
            case 0:        // sort by Country
                pStr1 = peu1 -> szCountry;
                pStr2 = peu2 -> szCountry;
                iResult = lstrcmpi (pStr1, pStr2);
                break;

            case 1:        // sort by Capital
                pStr1 = peu1 -> szCapital;
                pStr2 = peu2 -> szCapital;
                iResult = lstrcmpi (pStr1, pStr2);
```

```
                    break;

                ...
            }
        }
        return iResult;
    }
    ...
```

图 10-21 向大家展示了单击 Country 栏目标题以后按照国名排序的情况。

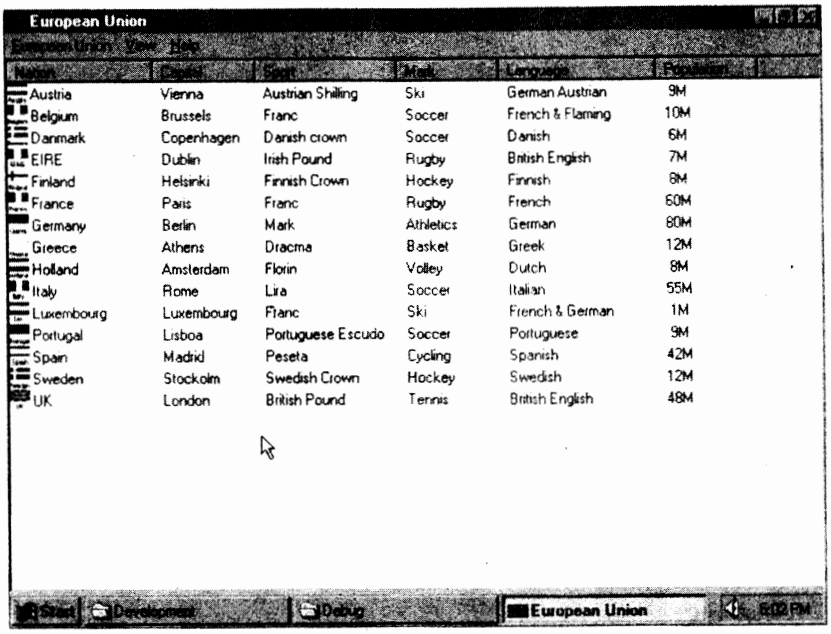


图 10-21 按照国名排序的欧盟成员国

10.7.5 列表视窗的宏函数

列表消息已经封装到一系列宏函数里，从而简化我们对代码的编写。表 10-33 内包含了用于这种通用控件的所有函数的一份完整清单。

表 10.33 列表视窗控件使用的宏函数

列表视窗宏函数	说 明
ListView_GetBkColor(hwnd)	返回列表视窗控件的背景颜色
ListView_SetBkColor(wnd,clrBk)	设置列表视窗控件的背景颜色
ListView_GetImageList(hwnd,iImageList)	返回与一个列表视窗控件关联在一起的图象列表的句柄
ListView_SetImageList(hwnd,himl,iImageList)	把一个图象列表与列表视窗关联到一起
ListView_GetItemCount(hwnd)	返回列表视窗控件内的项目总数
ListView_GetItem(hwnd,pitem)	取回关于项目的一些属性

（续）

列表视窗宏函数	说　明
ListView_SetItem(hwnd,pitem)	设置项目的一些属性
ListView_InsertItem(hwnd,pitem)	在列表视窗控件内插入一个新项目
ListView_DeleteItem(hwnd,i)	在列表视窗控件内删除一个项目
ListView_DeleteAllItems(hwnd)	删除列表视窗控件内的所有项目
ListView_GetCallbackMask(hwnd)	指出什么输出信息是由应用程序内部控制的
ListView_SetCallbackMask(hwnd,mask)	定义什么输出信息将由应用程序提供，而不是由控件本身提供
ListView_GetNextItem(hwnd,i,flags)	返回一个列表视窗项目的索引编号，这个项目具有指定的属性，并且和一个指定项目具有特定的关系，请参考表 10-30
ListView_FindItem(hwnd,iStart,plvfi)	对列表视窗的内容进行检查，查看是否有一个项目符合指定的特征
ListView_GetItemRect(hwnd,i,prc,code)	取回当前视窗内一个项目全部或者部分使用的限制矩形，参考表 10-30
ListView_SetItemPosition(hwndLV,i,x,y)	把一个项目移至列表视窗控件内指定的位置，只有在图标和小图标视窗中才有效
ListView_GetItemPosition(hwndLV,i,ppt)	取得当前项目在控件窗口内的当前位置
ListView_GetStringWidth(hwndLV,psz)	返回列表视窗控件内一个正文串的宽度
ListView_HitTest(hwndLV,pinfo)	判断鼠标指针是否正好位于列表视窗控件的一个特定位置处
ListView_EnsureVisible(hwndLV,i,fpartialOK)	强迫一个特定的项目在列表视窗控件内部分或者完全可见
ListView_Scroll(hwndLV,dx,dy)	对列表视窗控件的内容进行滚动
ListView_RedrawItems(hwndLV,iFirst,iLast)	对一定范围内的项目进行强制重画
ListView_Arrange(hwndLV,code)	根据表 10-32 内定义的某个标志，从而按照不同的规范对列表视窗控件内的项目进行排列
ListView_EditLabel(hwndLV,i)	进入一个列表视窗项目的现场编辑状态
ListView_GetEditControl(hwndLV)	返回现场编辑中涉及到的编辑窗口的句柄
ListView_GetColumn(hwnd,iCol,pcol)	用指定栏内的所有信息填写一个 LV_COLUMN 数据结构
ListView_SetColumn(hwnd,iCol,pcol)	设置一个列表视窗栏目的属性
ListView_InsertColumn(hwnd,iCol,pcol)	插入一个新栏
ListView_DeleteColumn(hwnd,iCol)	删除一个旧栏
ListView_GetColumnWidth(hwnd,iCol)	返回当前栏的宽度
ListView_SetColumnWidth(hwnd,iCol,cx)	设置栏宽
ListView_CreateDragImage(hwnd,i,lppUpLeft)	建立一幅图象，用于显示拖动过程中鼠标指针在屏幕上的形状
ListView_GetViewRect(hwnd,prc)	在图标或者小图标模式下，取回列表视窗控件内所有项目的限制矩形
ListView_GetTextColor(hwnd)	返回正文颜色
ListView_SetTextColor(hwnd,clrText)	设置正文颜色
ListView_GetTextBkColor(hwnd)	返回背景正文颜色
ListView_SetTextBkColor(hwnd,clrTextBk)	设置背景正文颜色
ListView_GetTopIndex(hwndLV)	在列表或者详细列表模式下面时，返回最顶部可见项目的索引编号
ListView_GetCountPerPage(hwndLV)	在列表或者详细列表模式下面时，返回列表视窗控件在垂直方向上的项目总数
ListView_GetOrigin(hwndLV,ppt)	返回一个列表视窗控件的当前视窗起点
ListView_Update(hwndLV,i)	可视化地更新一个列表视窗项目
ListView_SetItemState(hwndLV,i,data,mask)	改变一个项目的状态
ListView_GetItemState(hwndLV,i,mask)	返回一个项目的状态

（续）

列表视窗宏函数	说　明
ListView_ GetItemText（hwndLV，i，iSubItem，pszText，cchTextMax）	返回一个项目的正文
ListView_ SetItemText（hwndLV，i，iSubItem，pszText，cchTextMax）	设置一个项目的正文
ListView_SetItemCount(hwndLV,cItems)	通知一个列表视窗控件它不久会接收到大量项目
ListView_SortItems(hwndLV,pfnCompare,lPrm)	以应用程序定义的比较例程为基础，对列表视窗控件内的项目进行排序
ListView_SetItemPosition32(hwndLV,i,x,y)	用 32 位坐标设置一个项目在列表视窗控件内的位置
ListView_GetSelectedCount(hwndLV)	返回列表视窗控件内选定项目的总数
ListView_GetItemSpacing(hwndLV,fSmall)	返回列表视窗控件内各项目之间的间隔距离
ListView_GetSearchString(hwndLV,lpsz)	返回列表视窗控件的增量检索串

列表视窗类支持一种复杂的检索机制，应用程序利用这种机制可以在项目内查找特定的信息。LVM_FINDITEM 消息需要用到一个 LV_FINDINFO 数据结构的地址，利用该消息即可实现对特定项目的查找。其中，标志项可以选用表 10-34 内列出的某个值。和其他的控件不一样，在列表视窗里，我们可以根据存储于 lParam 区域内的信息、它们的标签正文串或者一个指定的内存位置，从而对项目进行检索。

```
typedef struct _LV_FINDINFO
{
    UINT flags;
    LPCSTR psz;
    LPARAM lParam;
    POINT pt;
    UINT vkDriection;
} LV_FINDINFO;
```

表 10-34　用于 LV_FINDINFO 数据结构的查找标准

列表视窗的 LVM_FINDITEM 标志	值	说　明
LVFI_PARAM	0x0001	以 lParam 区域内的内容为基础进行查找
LVFI_STRING	0x0002	以项目的标签正文为基础进行查找
LVFI_PARTIAL	0x0008	查找部分正文串，这个正文串起始于由 LV_FINDINFO 数据结构的 psz 项指定的串
LVFI_WRAP	0x0020	假如查找到头，就从起始处继续查找
LVFI_NEARESTXY	0x0040	寻找最接近指定坐标点的项目

只有在设置了 LVFI_NEARESTXY 标志的前提下，才能考虑类型为 POINT 的项。在这种情况下，vkDirection 项内包含了与查找方向对应的光标虚拟键。

与这个主题有关的操作是由 LVM_HITTEST 消息执行的。LVM_HITTESTINFO 标志项内包含了表 10-35 内列出的某个标志、光标位置以及由 iItem 项返回的选中项目的 ID。

```
typedef struct _LV_HITTESTINFO
{
    POINT pt;
    UINT flags;
    int iItem;
} LV_HITTESTINFO;
```

表 10-35 里列出了所有“点击测试”定义，它们与树形视窗使用的定义几乎完全一致。

表 10-35 LV_HITTEST 数据结构使用的标志

用于列表视窗的点击测试标志	值	说 明
LVHT_NOWHERE	0x0001	在客户区内，然而在最后一个项目的下方
LVHT_ONITEMICON	0x0002	在一个项目图标上方
LVHT_ONITEMLABEL	0x0004	在一个项目标签上方
LVHT_ONITEMSTATEICON	0x0008	在一个列表视窗项目的状态图象上方
LVHT_ONITEM		(LVHT_ONITEMICON | LVHT_OITEMLABEL | LVHT_ONITEMSTATEICON) 在一个项目上方
LVHT_ABOVE	0x0008	在客户区的上面
LVHT_BELOW	0x0010	在客户区的下面
LVHT_TORIGHT	0x0020	在客户区的右侧
LVHT_TOLEFT	0x0040	在客户区的左侧

10.7.6 通知代码

列表视窗是通过 WM_NOTIFY 消息把自己的通知代码传递给父窗口的。在 WM_NOTIFY 消息里，需要把 lParam 设置成指向一个 NM_LISTVIEW 数据结构的指针。

```
typedef struct _NM_LISTVIEW
{
    NMHDR hdr;
    int iItem;
    int iSubItem;
    UINT uNewState;
    UINT uOldState;
    UINT uChanged;
    POINT ptAction;
    LPARAM lParam;
} NM_LISTVIEW, * LPNM_LISTVIEW;
```

hdr.code 项是实际接收到的通知代码：

```
...
case WM_NOTIFY:
{
    NM_LISTVIEW *pnmlv = (NM_LISTVIEW *)lParam;

    switch(pnmlv -> hdr.code)
    {
        case LVN_COLUMNCLICK:
        {
            ...
        }
            break;
    }
...
```

表 10-36 为大家列出了用于列表视窗类的所有通知代码。请注意，列表视窗也可以提供对拖放操作的支持。为了实现这种特性，我们必须参考针对树形视窗控件讨论的那种方案，对 LVN_BEGINDRAG 通知代码进行跟踪（假如是非缺省的拖动操作，则需要跟踪 LVN_BEGINRDRAG）。

表 10-36 列表视窗使用的通知代码

列表视窗通知代码	值	说 明
LVN_ITEMCHANGING	(LVN_FIRST-0)	一个列表视窗项目正在改变
LVN_ITEMCHANGED	(LVN_FIRST-1)	一个列表视窗项目已经发生了改变
LVN_INSERTITEM	(LVN_FIRST-2)	插入了一个列表视窗项目
LVN_DELETEITEM	(LVN_FIRST-3)	删除了一个列表视窗项目
LVN_DELETEALLITEMS	(LVN_FIRST-4)	所有项目都被删除了
LVN_BEGINLABELEDIT	(LVN_FIRST-5)	现场标签编辑开始了
LVN_ENDLABELEDIT	(LVN_FIRST-6)	现场标签编辑结束了
LVN_COLUMNCLICK	(LVN_FIRST-7)	用户单击了一个栏目标题
LVN_BEGINDRAG	(LVN_FIRST-8)	拖动操作开始了
LVN_BEGINRDRAG	(LVN_FIRST-9)	拖动操作结束了
LVN_GETDISPINFO	(LVN_FIRST-11)	向显示出来代表某个项目的信息发出请求
LVN_SETDISPINFO	(LVN_FIRST-50)	表明一个项目必须更新
LVN_KEYDOWN	(LVN_FIRST-55)	键盘上一个按键已被按下

请注意，在这一章中，我们探讨了由 MS Windows 95 引入的两种关键性通用控件——树形视窗和列表视窗，另外还加上图象列表对象。其他通用控件怎么办呢？假如读者现在很急于了解它们的情况，请直接跳至第 15 章“Windows 高级技术”，其中提供的例子对属性表、动画控件、工具栏以及其他控件进行了全景式的描述。

第 11 章　图形设备接口示例

在这一章里，大家将就迄今为止学习到的一些理论进行实际运用。在前面的章节里，我们曾介绍过关于 GDI 对象的情况。例如，本书附带 CD 的 Listing 9.8 里提供了一个名为 EXTRACT 的示范程序，Listing 5.2 里则提供了一个名为 ICON 的示范程序，这两个程序对 GDI 对象进行了应用。除此以外，在第 4 章“消息和重画模式”里，我们对 WM_PAINT 消息的语法以及它在设备现场里的实现进行了详细的介绍。

11.1　MESSY 示例

现在让我们从一个简单的例子开始。运行 MESSY 程序，整个屏幕的范围内就会显示出许多个小的彩色圆。这些彩色圆会在用户回答了一个消息框以后消失。

它在许多方面都是比较独特的，这主要是由于我们在屏幕上显示了一个窗口。应用程序代码只限于一个单独的 WinMain () 函数内，这是 Windows 程序一种很普遍的处理方式。由于只存在一个窗口进程，就意味着缺少相应的窗口类注册进程，这种进程在 MESSY 里是用不着的。应用程序启动以后，它就会通过 GetDC () 函数简单地访问整个屏幕设备，同时向函数传递一个 NULL 值，如下所示：

```
...
hdc = GetDC (NULL);
...
```

现在让我们重新强调这样一个问题，用 NULL 值取代窗口句柄不失为指定系统桌面的一种最简单的方法。一种更好的方案则是使用 HWND_DESKTOP 定义：

```
#define HWND_DESKTOP ( (HWND) 0)
```

一旦获得了对屏幕桌面设备现场句柄的访问权，接下为就可以在上面“为所欲为”了，只要自己喜欢。

MESSY 程序能够完成的任务非常有限，这种任务是利用下面列出的一个 while 循环代码段来实现的：

```
...
// creating some thousand dots
while(nCnt++ <= MAXDOTS)
{
    hbrush = CreateSolidBrush(RGB(rand() % 255, rand() % 255,
                                  rand() % 255));
    oldhbrush = SelectObject(hdc, hbrush);
```

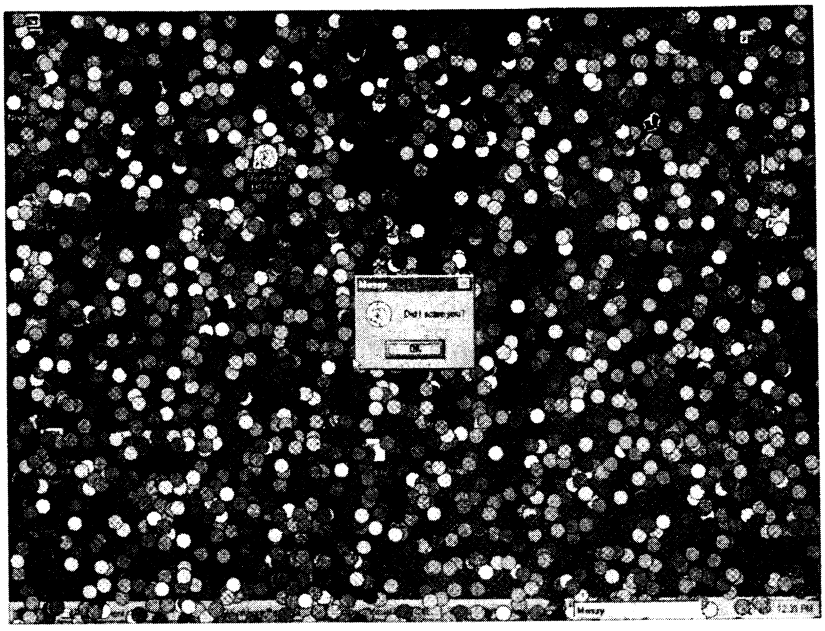


图 11-i　MESSY 在屏幕上生成了上千个小的彩色圆，
这是通过访问显示设备现场达到的效果

```
    x = rand() % SYS(SM_CXSCREEN);
    y = rand() % SYS(SM_CYSCREEN);

    Ellipse(hdc, x, y, x + 20, y + 20);
    SelectObject(hdc, oldhbrush);
    DeleteObject(hbrush);
}
...
```

利用 rand () 运行期库，MESSY 可以计算出在屏幕宽度和高度范围以内的一个点。屏幕的宽度和高度信息可以调用 GetSysMetrics()函数获得，这个函数已经转换成了 WIN32BK.H 里定义的 SYS () 宏。

在一个 Win95 系统里，我们必须与显示器有关的两个矩形区域进行区分。整个屏幕的尺寸是由通过调用 GetSystemMetrics () 函数来取得的，同时要传递 SM_CXSCREEN 和 SM_CYSCREEN 值，MESSY 程序采用的就是这种办法。屏幕上的小点则是调用 Ellipse () 函数的结果，它使用了前面建立的画笔对这些点进行描绘。

循环结束以后，MESSY 就会释放屏幕 DC (设备现场)，然后在程序中断之前显示一条消息：

```
...
```

```
release the screen DC
ReleaseDC (NULL, hdc);
...
```

为了清除屏幕并且删除由 MESSY 生成的所有小圆，必须先调用一个 InvalidateRect()函数，从而使整个屏幕无效。接着再调用 UpdateWindow () 对屏幕进行更新：

```
...
// cleaning the screen
InvalidateRect (NULL, NULL, TRUE);
UpdateWindow (NULL);
...
```

正如大家看到的那样，MESSY 的程序的源代码非常简单。设计它的宗旨是向大家揭示从一个 Win32 进程里访问整个屏幕设备现场有多么容易！事实上，对客户区边界以外的地方进行描绘是很少遇到的一种情况。这种做法实际上已经违背了常规的编程规则。除此以外，从设计风格的角度来看，在一个标准窗口的非客户区组件上进行描图处理是我们强烈反对的。

11.2 对象的描绘和移动

现在让我们转向一个更严肃、更专业的程序：OBJECTS 示范程序。这个程序的源代码可以在本书附带 CD 的 Listing 11.2 里找到。启动 OBJECTS 程序以后，用户就可以在客户区描绘矩形。进行具体选择的菜单项列于 Objects 菜单内，如图 11-2所示。其中，缺省的绘图彩色被设置成红色，如图 11-3 所示。然而，我们可以通过一个标准的 Choose color（选择颜色）对话框对这种颜色进行定制，如图 11-4 所示。

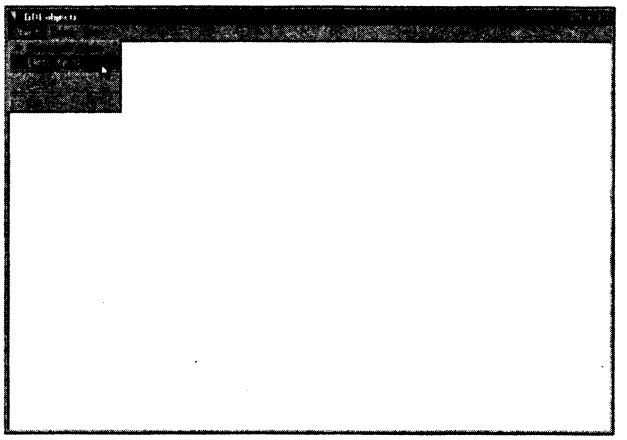


图 11-2 OBJECTS 程序的 Objects 菜单可用于选择应用程序客户区内下次准备描绘的几何形状

图 11-3　Colors 下拉式菜单提供了三种基本颜色，另外还有个菜单项可用于访问 Choose color 通用对话框

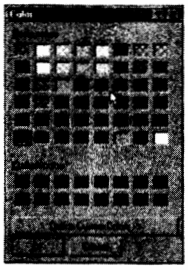


图 11-4　OBJECTS 程序通过 Color 通用对话框提供了更广范围内的颜色选择

现在，我们已经选好了自己希望使用的几何形状，以及自己喜欢的一种颜色，接下来就可以在应用程序客户区内开始绘图操作了。这时只需按下鼠标左键不放，然后拖动它，直到抵达一个希望的位置为止。如图 11-5 所示，这种绘图的准备工作在屏幕上是以一个细线矩形框的形式表达出来的，它会不断跟踪鼠标的当前位置。图 11-6 则展示了松开鼠标左键以后得到的绘图结果。

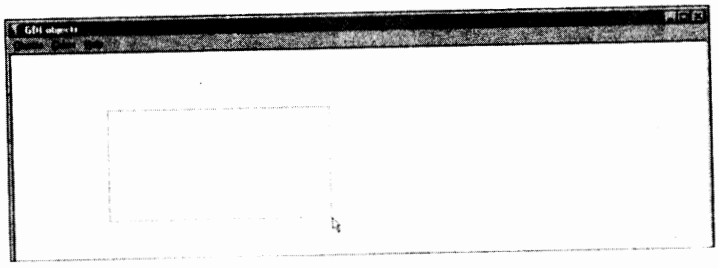


图 11-5　描绘一个几何形状的时候，OBJECTS 提供了一些视觉上的反馈信息

现在让我们重复刚才讲述的同样的操作，只是这次画的是一个椭圆，并且选用不同的颜色对其进行填充。正如大家在图 11-7 里看到的那样，绘图是向一个不同的方向进行的，这时的起始点位于当前点的右侧。而在上一个例子里，起始点则位于当前点的左侧（参考图 11-5）。松开鼠标左键以后，一个浅黄色的椭圆出现了（在黑白两色的书页上面印刷出来则是浅灰色），它覆盖了前面画的那个矩形的左下角。如图 11-8 所示。

现在，我们有客户区内已经有两个几何图形了，可以选择其中的一个，然后把拖至某个新的位置处。把鼠标指针放置于希望拖动的那个对象上方，然后按下鼠标左键不放，把它拖

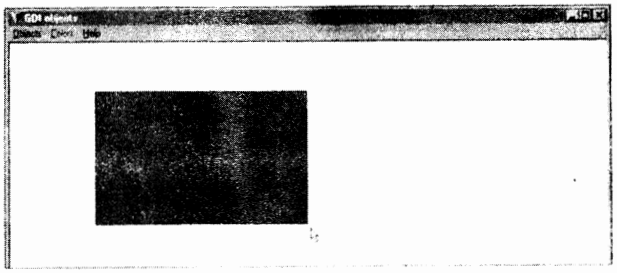


图 11-6　第一个矩形已经在应用程序客户区描绘好了

动到一个新位置即可。在拖动过程中，图形会继续保持它在老地方的显示，只有围绕它一个细线框会随着鼠标的移动而移动，如图 11-9 所示。一旦松开鼠标左键，这个图形对象就会立即移动到新位置处，如图 11-10所示。

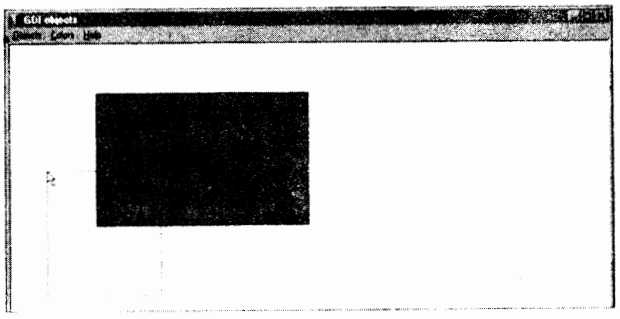


图 11-7　向左上角拖动鼠标，从而生成一个椭圆对象

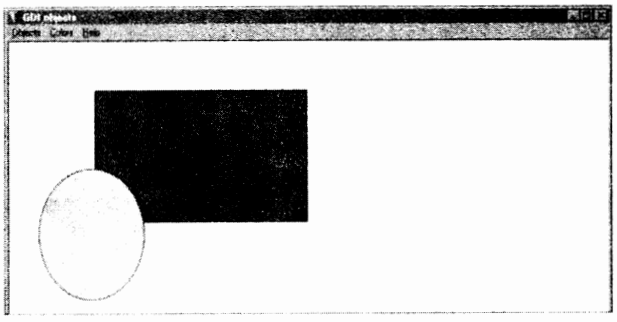


图 11-8　椭圆部分覆盖于矩形的上方

假如用鼠标右键在某个对象上方单击，这时就会显示出一个弹出式菜单，其中提供了与那个几何形状有关的特定信息（如图 11-11 所示）。正如大家通过图 11-11 看到的那样，这个弹出式菜单的作用是提供一些视觉反馈信息，而不是能够从中选择菜单项的一个标准菜单。相同的逻辑亦可应用于一个椭圆对象，如图 11-12 所示。在这种情况下，程序只允许用户在一个椭圆形状以内单击鼠标右键。

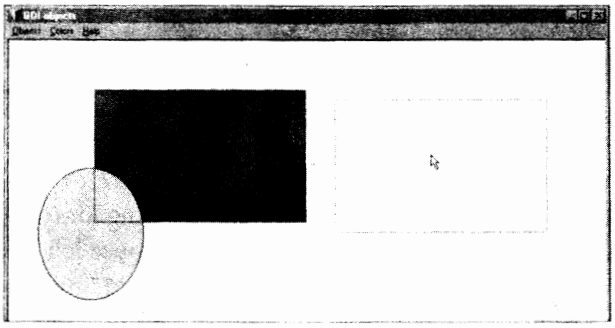


图 11-9　向窗口的右侧拖动矩形对象

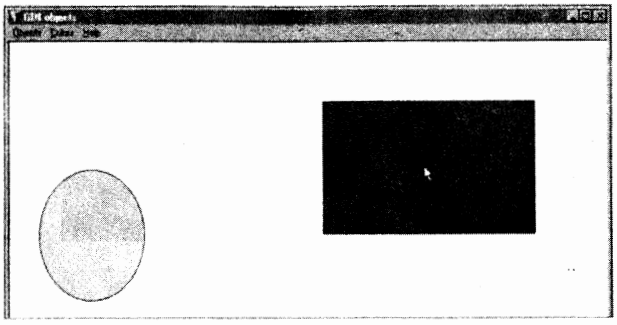


图 11-10　深色的矩形已经移到了客户区的右侧

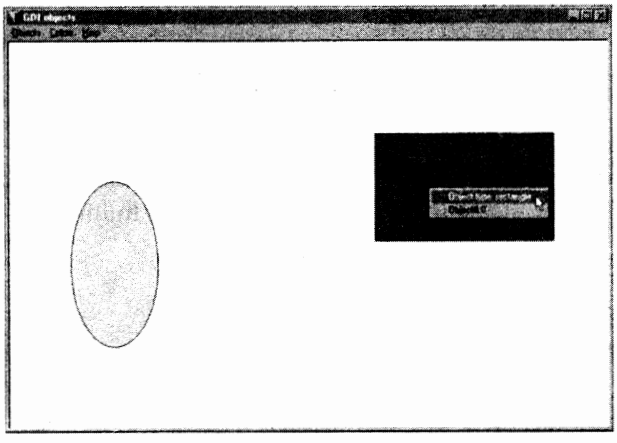


图 11-11　与一个矩形对象有关的弹出式菜单

刚才，我们对OBJECTS 程序里实现的所有特性都进行了简略的介绍。对于这个应用程序的功能，我认为还能进行更加深入的扩展。例如，我建议大家试着实现这样的一些功能：拖动对象的一个边框，从而使几何形状伸缩，把一个对象带入前台或者把它送入后台，改变对象的颜色以及增加更多的几何形状等等。

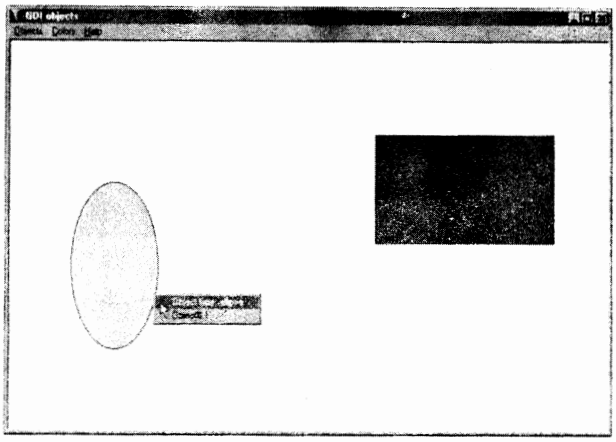


图 11-12　显示出一个椭圆对象的弹出式菜单

接下来，我们准备对 OBJECTS 的代码进行探讨。

11.2.1　数据结构

与每个绘图对象有关的信息都是存储在一个 SHAPE 数据结构里的，这个结构是在应用程序源代码内定义的，如下所示：

```
typedef struct _SHAPE
{
        RECT rc;
        int iShape;
        COLORREF clr;
} SHAPE, *PSHAPE;
```

应用程序把每个图形对象的尺寸、形状 (OB_RECTANGLE 或者 OB_ELLIPSE) 以及颜色引用编号都存储于这个数据结构内：其中，颜色引用信息是用于建立一个画笔的基本信息。

在窗口里程里，我们建立了用于容纳 MAXSHAPES 对象的一个 SHAPE 数组，并且把它设定成静态标识符，使其在整个进程内都是“可见”的。

```
static SHAPE Shapes [MAXSHAPES];
static int iPos;
```

除此以外，iPos 静态整数可以对客户区内画出的图形对象总数进行跟踪。Shaps 结构的最后两个项初始化设置成描绘的是一个红色的矩形。

```
// define the next color
Shapes [  iPos] .clr = RGB (  255, 0, 0);
// define the next shape
Shapes [  iPos] .iShape = OB_RECTANGLE;
```

这两个值将自动分配给下一个对象，除非用户自己选择了不同的颜色或者形状。

11.2.2 几何形状的描绘

所有的绘图和移动行动都是围绕三条消息展开的，这三条消息分别是：WM_LBUTTONDOWN，WM_MOUSEMOVE 以及 WM_LBUTTONUP。用户按下鼠标左键以后，应用程序就会把鼠标指针的当前位置存储到两个静态整数变量里。这两个变量分别为 xStart 和 yStart，它们在整个窗口进程的范围里都是“可见”的。静态布尔值标志 fDrawing 则设置成 TRUE，从而指出准备进行一次新的绘图操作。

只要鼠标一发生动作，应用程序就会接收到一条 WM_MOUSEMOVE 消息。在这个时候，OBJECTS 程序会检查 fDrawing 标识符，从而验证鼠标左键是否仍然处于按下状态。假如左键仍然处于按下状态，就表明准备实际描绘一个新的风何形状。假如用户松开了鼠标左键，就会接收到一条 WM_LBUTTONUP 消息，这时会把布尔值再次设置成 FALSE。

从起始位置到当前位置描绘一个细线方框是 WM_MOUSEMOVE 消息代码块内需要首先完成的任务。API 函数 DrawFocusRect（）可以帮助我们围绕一个列表框项目或者在一个按钮内描绘一个点线框架，从而从视觉上标识出对话框内当前已选中的项目。OBJECTS 函数利用了这个函数对准备构建的几何形状进行定界处理。如图 11-5 所示。

```
#include <winuser.h>
BOOL DrawFocusRect (HDC hDC, CONST RECT *prc);
```

参数	说明
HDC hdc	一个有效的设备现场句柄
RECT *prc	某个 RECT 数据结构的地址
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

GetDC（）提供了客户区的设备现场，而矩形区域仍然需要进行计算，从而判断出起始位置和当前位置。在显示实际的对象形状之前，OBJECTS 程序需要先从客户区删除上一个 WM_MOUSEMOVE 消息块里画出的围绕细线框。只有这样做，OBJECTS 才能保证屏幕的“干净”，从而剔除与前一个临时对象有关的所有框架。为了达到这一要求，我们需要实际地重画以前的临时细线框架，只是采取了一种不同的 ROP（光栅操作）代码，就像下面这个代码段显示的那样：

```
...
if (fDrawing)
{
    HDC hdc;

    hdc = GetDC (hwnd);
    SetROP2 (hdc, R2_NOTXORPEN);
```

```
// erasing the previous rectangle
DrawFocusRect (hdc, &Shapes [iPos] .rc);
...
```

可以看出，其中扮演关键角色的就是 SetROP2 () 函数。

```
#include <wingdi.h>
int WINAPI SetROP2 (HDC hdc, int fnDrawMode);
```

第一个参数是一个有效的设备现场句柄，后面紧接着一个或者多个 ROP 代码，这此代码可以在表 11-1 里选择。

表 11-1 用于 SetROP2 () API 的光栅操作代码

标 志	值	说 明
R2_BLACK	1	总是为黑色
R2_NOTMERGEPEN	2	R2_MERGEPEN 颜色的反转色
R2_MASKNOTPEN	3	通用于屏幕颜色和画笔反转色的颜色组合
R2_NOTCOPYPEN	4	画笔颜色的反转色
R2_MASKPENNOT	5	通用于画笔颜色和屏幕反转色的颜色组合
R2_NOT	6	屏幕颜色的反转色
R2_XORPEN	7	画笔内颜色与屏幕内颜色的组合
R2_NOTMASKPEN	8	R2_MASKPEN 颜色的反转色
R2_MASKPEN	9	通用于画笔和屏幕的颜色的组合
R2_NOTXORPEN	10	R2_XORPEN 颜色的反转色
R2_NOP	11	未改变
R2_MERGENOTPEN	12	屏幕颜色和画笔颜色反转色的组合
R2_COPYPEN	13	画笔颜色
R2_MERGEPENNOT	14	画笔颜色和屏幕颜色反转色的组合
R2_MERGEPEN	15	画笔颜色和屏幕颜色的组合
R2_WHITE	16	总是为白色

我们在这儿的目标是要保存现成的基础对象，而不是破坏它们。同时，还要为用户提供当前所画对象的可视化反馈信息。R2_NOTXORPEN 标志正是达成我们这一目标的最恰当的工具。

在我们前面从 OBJECTS.C 里摘录下来的代码段里，DrawFocusRect () 函数在更新当前的矩形之前先描绘了一个矩形框，这样就消除了处理上一条 WM_MOUSEMOVE 消息时描绘出来的点线框。

在这以后，通过测量鼠标指针的起始点和当前位置，我们就可以计算出准备构建的那个几何形状所占据的矩形区域大小了。不走运的是，这儿没有一个现成的 API 函数可以帮助我们把两个点转换成一个矩形，尽管确实存在一个 SetRect () API 函数。这个函数只有在起始

点位于矩形区域的左上角时才能使用，在这种情况下，鼠标是向屏幕的右下角拖动的。从本质上说，这个函数并不显得具有足够的智能程度，它不能动态地理解（X，Y）的值到底是标识矩形区域的左上角，还是它的右下角。因此我们不得不自己先进行一番处理，如下所示：

```
...
// going left?
if(xStart > x)
{
    Shapes[iPos].rc.left = x;
    Shapes[iPos].rc.right = xStart;
}
else    // going right
{
    Shapes[iPos].rc.left = xStart;
    Shapes[iPos].rc.right = x;
}

// going down?
if(yStart < y)
{
    Shapes[iPos].rc.bottom = y;
    Shapes[iPos].rc.top = yStart;
}
else    // going right
{
    Shapes[iPos].rc.bottom = yStart;
    Shapes[iPos].rc.top = y;
}

// show the new position
DrawFocusRect(hdc, &Shapes[iPos].rc);
ReleaseDC(hwnd, hdc);
...
```

在 WM_MOUSEMOVE 代码里，我们最后还要描绘新的对象边框，接着再利用 ReleaseDC () 把设备现场传递回 Windows。

最后的绘图行动是在 WM_LBUTTONUP 消息块里完成的。其中涉及的代码与用于 WM_MOUSEMOVE 消息的代码基本上完全一致。正如以前提到的那样，fDrawing 布尔值会设置成 FALSE，客户区的输出也会由于实际几何形状的描绘而得到更新。所有这些都是在

WM_PAINT 消息里进行的，正如我们预期的那样。

```
...
hdc = BeginPaint(hwnd, &ps);
// black pen
hpen = CreatePen(PS_SOLID, 1, RGB(0, 0, 0));
holdpen = SelectObject(hdc, hpen);

for(i = 0; i < iPos; i++)
{
    // set the brush color
    SelectObject(hdc, CreateSolidBrush(Shapes[i].clr));

    switch(Shapes[i].iShape)
    {
        case OB_RECTANGLE:
        {
            Rectangle(hdc,
                      Shapes[i].rc.left,
                      Shapes[i].rc.top,
                      Shapes[i].rc.right,
                      Shapes[i].rc.bottom);
        }
            break;

        case OB_ELLIPSE:
        {
            Ellipse(hdc,
                    Shapes[i].rc.left,
                    Shapes[i].rc.top,
                    Shapes[i].rc.right,
                    Shapes[i].rc.bottom);
        }
            break;
    }
}
// select the old pen
SelectObject(hdc, holdpen);
// destroy the new pen
```

```
DeleteObject(hpen);
EndPaint(hwnd, &ps);
...
```

11.2.3 现成对象的移动

在客户区拖动某个现成对象是一种相当直接的任务。在用于生成新对象的那个代码段里，许多代码都可以重新利用，用于完成这种操作。用户按下鼠标左键的时候，应用程序就会检查是否正好有一个对象位于鼠标指针的下方。PtInRect () 和 PtInRgn () 函数允许我们判断一个点是否位于一个矩形或者另外一个区域内。事实上，我们在第 5 章“资源文件”的 ICON 例子里已经对这个函数进行了讨论。

在拖动操作里，除了 Shapes 数组以外，不会涉及到其他任何一个数据或者标识符。对象的当前位置将不断地更新，并在屏幕上显示，同时保持鼠标指针与对象边界之间固定的偏移距离。

```
...
if(fDrag)
{
    HDC hdc;

    hdc = GetDC(hwnd);
    SetROP2(hdc, R2_NOTXORPEN);
    // erasing the previous rectangle
    DrawFocusRect(hdc, &Shapes[iDrag].rc);

    //OffsetRect(&Shapes[iDrag].rc, x - xStart, y - yStart);
    Shapes[iDrag].rc.left += x - xStart;
    Shapes[iDrag].rc.right += x - xStart;
    Shapes[iDrag].rc.top += y - yStart;
    Shapes[iDrag].rc.bottom += y - yStart;
    // remember the previous location
    xStart = x;
    yStart = y;

    // draw the rectangle in its current location
    DrawFocusRect(hdc, &Shapes[iDrag].rc);

    ReleaseDC(hwnd, hdc);
    break;
}
...
```

我们在这儿能够改变的是在拖动一个椭圆的时候提供给用户的视觉效果。在这儿，我们可以不再使用对椭圆区域进行限制的一个矩形，最好是拖动一个点线椭圆或者实际的对象；假如安装了 Microsoft Plus!，我们就可以看到第二种情况的出现（拖动实际的对象）。这了实现这一目标，只需要用 API 函数 Rectangle () 或者 Ellipse () 替换掉对 DrawFocusRect () 的调用即可。在这个时候，我们可以把 WM_PAINT 消息块内的许多适用的代码剪切和粘贴到 WM_MOUSEMOVE 消息里，这样就可以完成该特性的增添。

11.2.4 位图的载入

本章第三个也是最后一个例子是 LOADBMP，它向我们展示了如何载入一幅 256 色的位图文件（.BMP）。能载入一幅 256 色的位图并不是什么新鲜事情，Win32 提供的 API 函数 LoadImage () 就是专门针对这一目的而设计的，该函数的运用非常灵活。

LOADBMP 程序（如图 11-13 所示）可以针对选中的位图建立一个内存映射文件，判断它的尺寸和分辨率，然后建立和激活相应的调色板。对于由多种颜色建立的复杂位图来说，最后那种处理是相当有用的。一旦图象成功地载入，LOADBMP 就会对主窗口的尺寸进行相应的调整，然后在自己的标题栏内显示出位图文件的完整路径名。如图 11-14 所示。

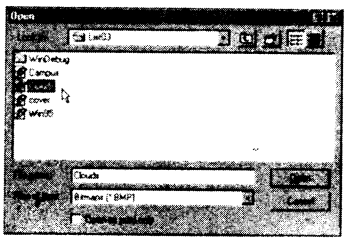


图 11-13 LOADBMP 程序通过一个通用对话框载入位图

图 11-14 CLOUDS 墙纸可以在你的 Windows 95 系统里使用

这并不是在为了 LOADBMP 里放置一幅位图而支持的唯一方式。另外一种可选的办法是把一幅或者多幅位图直接拖动到应用程序客户区内，然后松开鼠标左键，如图 11-15 所示。现

在系统内同时运行了 LOADBMP 的两份拷贝。假如把单独一幅位图拖动到 LOADBMP 里，应用程序要么把它直接显示出来，要么用新位图替换掉原来就有的一幅位图。对于拖动到 LOADBMP 里的一系列.BMP文件来说，其中的第一幅位图会直接装载到客户区内。剩余的其他位图则会存放于 LOADBMP 程序的其他运行拷贝（程序实例）里。这些程序实例是通过调用 CreateProcess（）函数来激活的。

现在让我们一起来讨论 LOADBMP 程序使用的源代码。

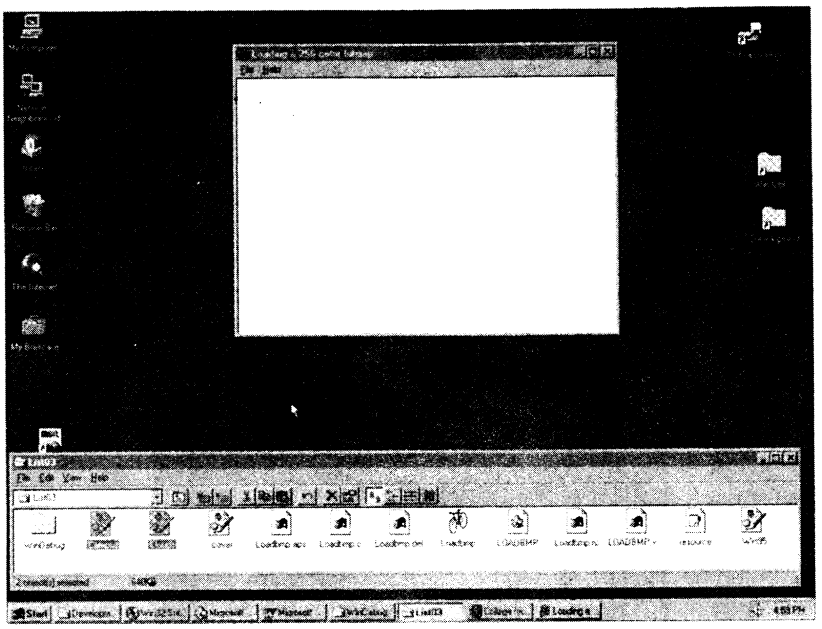


图 11-15　把两幅位图拖动到 LOADBMP 里是打开和显示这些位图的最简便的方式

11.2.5　源代码剖析

从程序开发的角度来看，位图就是几个数据结构的集合，每个数据结构都提供了对图象进行描述的特定信息。位图对象内包含了 BITMAPINFOHEADER 数据结构，后面跟随了一系列 RGBQUAD 结构，这些结构指定了位图图象内使用的颜色。RGBQUAD 数据结构一种通常的称呼是“调色板”。当一幅位图存放于文件系统内时，它的内部结构就会发生改变，这是由于一个附加头结构的存在造成的。这个头结构属于 BITMAPFILEHEADER 类型，它优先于其他所有结构。表 11-2 为大家总结了磁盘上存储的一幅位图的所有组件。

从文件里载入一幅位图时会发生什么情况呢？LOADBMP 会调用内部 DIBRead（）函数，从而针对指定的位图建立一个位图文件，这样就节省了系统 RAM 的大量处理时间和空间。

表 11-2　与 .BMP 文件内 Win32 数据结构关联起来的数据字节

数据结构	字　节	数据结构	字　节
BITMAPFILEHEADER	0x00-0x0D	RGBQUAD 数组	0x32-0x75
BITMAPINFOHEADER	0x0E-0x31	颜色索引数组	0x76-0x275

```
...
// opening the file in pszInclFile
hFile = CreateFile(pszBitmap,
                   GENERIC_READ,
                   0,
                   NULL,
                   OPEN_EXISTING,
                   FILE_ATTRIBUTE_NORMAL
                       | FILE_FLAG_SEQUENTIAL_SCAN,
                   NULL);

if(hFile == INVALID_HANDLE_VALUE)
{
    return FALSE;
}
// get the file size
dwSize = GetFileSize(hFile, NULL);

// create file mapping
hFileMap = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, dwSize,
  NULL);

if(! hFileMap)
{
    // closing the include file
    if(!CloseHandle(hFile))
        MessageBox(NULL, "Error closing the file", "Include", MB_OK);
    MessageBox(NULL, "Error creating the file mapping", "hFileMap invalid",
    MB_OK);
    return FALSE;
}

// access the file
pFile = MapViewOfFile(hFileMap, FILE_MAP_READ, 0, 0, 0);
if(! pFile)
{
    // removing the file mapping
    if(! CloseHandle(hFileMap))
        MessageBox(NULL, "Error closing the map file", "Include", MB_OK);
```

```
    // closing the include file
    if(! CloseHandle(hFile))
        MessageBox(NULL, "Error closing the file", "Include", MB_OK);
    return FALSE;
}
...
```

ReadDIB () 例程的作用是对映射文件进行分析，在里查找不同的位图段，并且取得关于图象的信息。这些信息包括分辨率、每比特的颜色数以及 RGBQUAD 数据结构的地址等等。

根据位图的实际大小而改变了应用程序主窗口的尺寸以后，就会生成一条 WM_PAINT 消息，并且最终在窗口进程里进行处理。只有在那个时候，应用程序才会根据位图头结构段内的 RGBQUAD 数据结构信息，从而建立一个相应的调色板。

11.2.6 拖放位图的接收

LOADBMP 不是一个兼容于 OLE 2.x 的应用程序。它接收拖动文件的能力是建立在 WM_DROPFILES 消息以及相应 API 函数的基础上的。

就这方面来说，Windows 95 外壳的行为就好象一个老式的文件管理器。它会在一个数据结构里填写关于拖动文件有关的信息。通过对 WM_DROPFILES 消息进行拦截，应用程序就可以对这个信息块进行分析，从而判断其中是否包含了有用的东西。LOADBMP 程序只提供了对 .BMP 文件的支持，只要某文档没有带 .BMP 扩展名，就会毫不犹豫地忽略这个文档。不幸的是，这种简单的拖放协议并没有提供数量充足的工具，从而建立起具有专业水准的应用程序。因此，我们转向采纳 OLE 看来是唯一的出路。关于拖放协议更详细的信息，我们会在第十五章“Windows 高级技术”里进行详细的探讨。在第 16 章“Win95 外壳开发”里，SHELLDRG 示范程序会向大家阐述如何通过 OLE 与外壳进行交互作用。

第12章　非标准的输入和输出

在这一章中，我们将讨论几种输入形式。除了传统的键盘和鼠标输入以外，其他几种工具也可以从广义上当作输入设备来进行考虑。Windows 95 出现的文件夹就属于这样的一个例子。相当大一部分 Windows 95 用户情愿在菜单栏下方显示一个工具栏。和菜单栏或者键盘比较起来，工具栏提供了一种更简便、更有效的方式对命令进行选择。

本章最后那个例子是一个多媒体 Windows 应用程序。这是一个音乐 CD 播放器，它为我们提供了一些有趣的人体工程学特性、大量 MCI（媒体控制接口）命令以及与 MS Access 7.0 数据库的一个 ODBC 连接。首先，让我们先从一种最传统的输入设备——键盘——开始讨论。

12.1　键盘

正如我们在第一章“Win32 中的软件开发”里就已经提到的那样，Windows 95 采用的是与 Windows 3.x 不同的一种输入模式。通过对输入机制的特殊处理，异步输入模式（在 Windows NT 也得到了实现）为末端用户提供了许多优点。

由于系统线程为每个线程都配备了一个独立的消息队列，所以输入信息流是分隔开的。系统线程通常也可以称为 RIT，即“原始输入线程”（Raw Input Thread）。这样一来，即使一个应用程序完成的任务需要漫长的时间，它也不会对整个系统产生影响。所以，用户现在可以更加有效地同时访问和使用另外一个应用程序。在一个更低的间隔级别下，这也意味着一个单独的进程可以构建成多个线程，每个线程都带有单独的输入队列。这种结构反映了单一应用程序内相同的系统范围行为。最终的结果就是用户获得了一个更好的多任务环境，而且应用程序总能对用户发出的请求提供及时的响应。

显然，RIT 和多重输入队列的存在并不能解决单独由于只有一个键盘而造成的可用性问题。这种设备在同一时刻只能为一个窗口提供服务，扮演一种信息源的角色。更准确地说，键盘只能为当前的活动窗口或者具备键盘输入焦点的窗口提供服务。以前的 Win16 开发者们对于这些概念是非常熟悉的。在 Windows 95 里，由于 RIT 模式的引入，所以不得不考虑另外一种不同的解决方案。在 Windows 95 里，活动窗口是与某个单一消息队列关联在一起的定义，这个消息队列不再像过去那样反映的是整个窗口池。活动窗口肯定是当前位于前台的某个顶级窗口。来自键盘的所有输入都会直接进入建立那个窗口的线程消息队列。用户可以通过鼠标或者激活一个新进程，从而指定一个活动窗口。

键盘输入焦点是活动窗口的一种附加属性，或者是它的某个子窗口的属性。为了澄清这个概念，我们可以拿一个对话框来作比方。为了展示活动窗口与带有键盘输入焦点的窗口之间的区别，对话框是最好的一种例子。在对话框里，活动窗口就是对话框本身，而键盘输入焦点则是其中某个控件的属性。

通过调用 API 函数 SetActiveWindow（），应用程序就能够准确无误地激活由当前进程建立的一个顶级窗口。类似地，它可以通过调用 GetActiveWindow （）函数来取得活动顶级窗口的句柄。

```
#include〈winuser.h〉
HWND GetActiveWindow (VOID);
```

返回句柄指定了与调用该函数的那个线程关联在一起的活动窗口。某个顶级窗口变成活动状态以后，它就会接收到一条 WM_ACTIVATE 消息，其中的 wParam 低字（LOWORD）内带有相应的 WA_ACTIVE 值。顶级窗口失去活动性之前，同一条消息也会抵达这个窗口，这一次则在 wParam 的低字内带有相应的 WA_INACTIVE 值。

GetFocus () 和 SetFocus () 函数允许我们取得具备键盘输入焦点的一个窗口的句柄。通过传递对应的通知代码，一个通用控件就可以通知自己的父窗口关于键盘输入焦点的取得或者丢失情况。

键盘输入的控制

当作一次单一行动进行对待，这主要是由于键盘是用于在一个编辑窗口内输入正文的基本设备。尽管这种说法是正确的，但是系统仍然要把物理性的按下某个键与后续的松开这个键区分开，从而生成两条独立的消息。另外，Win32 还要根据击键序列的本质对这些事件进行区分。假如 Alt 键已经按下，Win32 就会生成一条系统键盘消息（先是 WM_SYSKEYDOWN，然后是 WM_SYSKEYUP）。在其他所有情况下，应用程序都会先接收到一条 WM_KEYDOWN 消息，稍后则会再接收到一条 WM_KEYUP。所有这些消息都会张贴到线程消息队列里，从而在消息循环里进行传输。

对于应用程序来说，需要它亲自对系统消息进行拦截和处理是很不容易碰到的一种情况。这些工作一般都是由系统自动完成的，不需要在应用程序源代码内对其进行任何干涉。因此，一个标准的加速键组合（比如用于关闭窗口并最终中断一个进程的“Alt+F4”）是在系统内部进行管理的。相同的规则亦可延展到单独一个 Alt 键的按下。假如用户单独按下了一个 Alt 键，控制权就会立刻转向应用程序菜单栏内的最左侧那个顶级菜单。

和系统键盘消息比较起来，WM_KEYDOWN 和 WM_KEYUP 消息看起来要稍微常用一些，因为它们可以用于其他任何一种击键操作。这些消息的语法结构与包含了虚拟键的 wParam 以及 lParam 是完全一致的，对此大家可以参考表 12-1。lParam 的 32 个二进制位被分隔成七个从属部分，用于提供某些附加的信息。

WM_KEYDOWN	0x0100
wParam	虚拟键代码
lParam	按键数据

WINUSER.H 头文件定义了 Windows 使用的全部非系统按键的所有虚拟键代码。这种虚拟代码是一种数值定义，对它们的总结请参考表 6-10。

尽管这两条消息可以携带大量信息，但是它们在一个窗口进程里却是很少出现的。事实上，消息循环里的 TranslateMessage () 已经把 WM_KEYDOWN 和 WM_KEYUP 合并到了一起，并且提供了一条 WM_CHAR 消息。这条消息可以张贴到应用程序的消息队列里，并能在下次执行 GetMessage () 的时候再接收到。这种行为极大地简化了窗口进程的代码编写工作。事实上，直接处理用户输入的某个应用程序主要是对 ASCII 字符感兴趣，它需要在屏幕上响应出这些字符。WM_CHAR 正是针对这一目的设计的，利用它就不用对键按下和松开事

件进行分别考虑，这些考虑已经浓缩在一条单独的消息内了：

WM_CHAR	0x0080
wParam	字符代码
lParam	按键数据

表 12-1 在 WM_KEYDOWN 和 WM_KEYUP 消息的 lParam 内使用的 32 个二进制位

lParam	说 明
0—15	指定重复次数；换言之，即松开一个键之前需要按下它的次数（对于 WM_KEYUP 来说肯定是 1）
16—23	根据原始设备制造商（OEM）键盘布局的最初配备指定键盘扫描码
24	指出某个键是否为扩展键，比如增强型 101 或者 102 键键盘上的右手 Alt 和 Ctrl 键等等。假如是个扩展键，这个二进制就设置成 1；否则就设置成 0
25—28	保留
29	指定现场代码。对于 WM_KEYDOWN 或者 WM_KEYUP 消息来说，这个值肯定应该设置成 0
30	指出上一次按键状态。假如消息发送之前该键已被按下，就设置成 1；假如没有按下则设置成 0（对于 WM_KEYUP 来说肯定设置成 1）
31	指定传输状态。对于 WM_KEYDOWN 消息来说，这个值肯定为 0；对于 WM_KEYUP 来说，则肯定为 1

lParam 内的信息反映了表 12-1 内列出的内容，而 wParam 则提供了字符代码，这是一种数字信息。对一个预定义窗口进行子类定义时（参考第十五章“Windows 高级技术”），或者开发类似于字处理器的一个应用程序时，对 WM_CHAR 消息进行拦截和处理是很有好处的。在这些环境下，应用程序必须知道用户键入了什么，并且进行相应的行动（在屏幕上显示出相应的字符、移动光标以及删除上一个字符等等）。在其他环境下，对这种消息进行跟踪就没有必要了。因为在不需要任何应用程序代码的干涉下，Windows 就能提供对多种键盘交互作用的支持。

在一个对话框里，我们可以利用相应的记忆键来选择一个菜单项或者一个控件，这种操作完全是由 Windows 本身控制的。就目前来说，一个更紧迫的问题是了解自己 Win32 开发环境支持什么样的字符集，并且对这些字符集进行区分。从一开始，Windows 就配备了 ANSI 字符集。ANSI 是美国国家标准协会的简称，即英语中的 American National Standard Institute。这个字符集的前 128 个字符与 ASCII 字符集内的字符是完全等价的。ASCII 是美国信息交换标准码（American Set of Characters for Infomation Interchange）的简称，由 IBM 公司在世界上第一台 PC 中采用（1981 年）。在 Windows 术语中，ASCII 就相当于 OEM（原始设备制造商）。这就解释了为什么某些 API 里包含了 OEM 字符的缘故。NT 支持一种新的字符格式，即 Unicode（统一编码），这种编码在 Windows 95 里还没有得以完全实施。

12.2 ANSI 或 ASCII

在缺省的情况下，Windows 95 支持的是 ANSI 字符集，而我们却不得不把一个 ANSI 字符串转换成 ASCII 格式，这样才能访问 128 个字符组以上的其他不同符号。在本书附带 CD 的 Listing 12.1 里，大家可以找到一个名为 ANSI 的示范程序。这是一个简单的应用程序，它能

在应用程序客户区内显示出这两套字符集内的所有符号。缺省时，它显示的是ANSI字符（如图12-1所示）。这时假如选择相应的菜单项，就可以立刻切换到ASCII字符（如图12-2所示）。如果在其中某个字符上单击鼠标左键，就会在屏幕中央显示出一个消息框，并在一个正文串里列出了点中的字符。

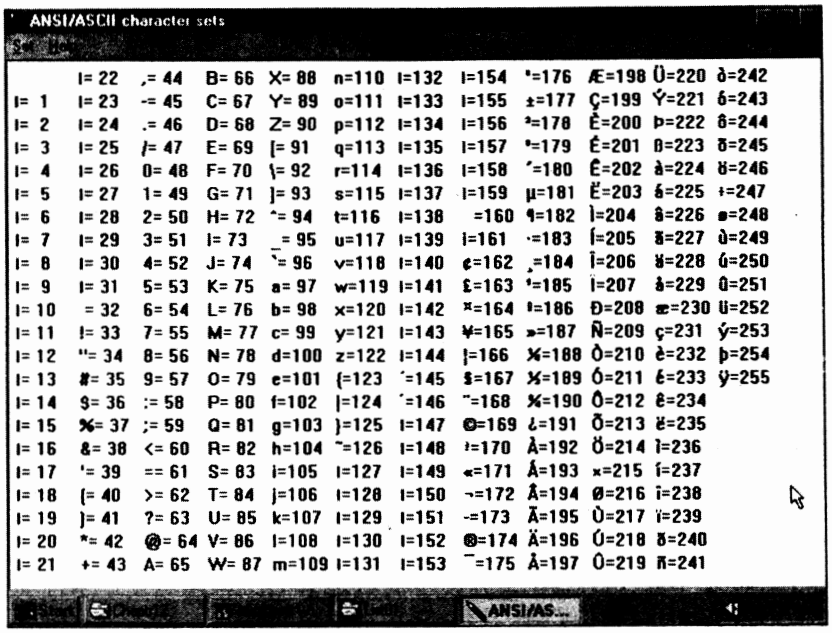


图 12-1　ANSI字符集内的符号

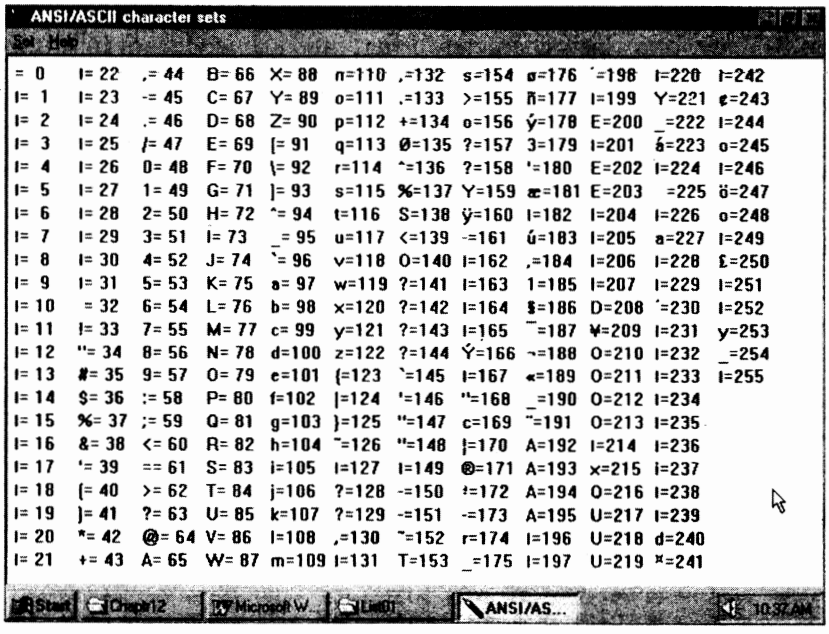


图 12-2　ASCII字符集内的符号

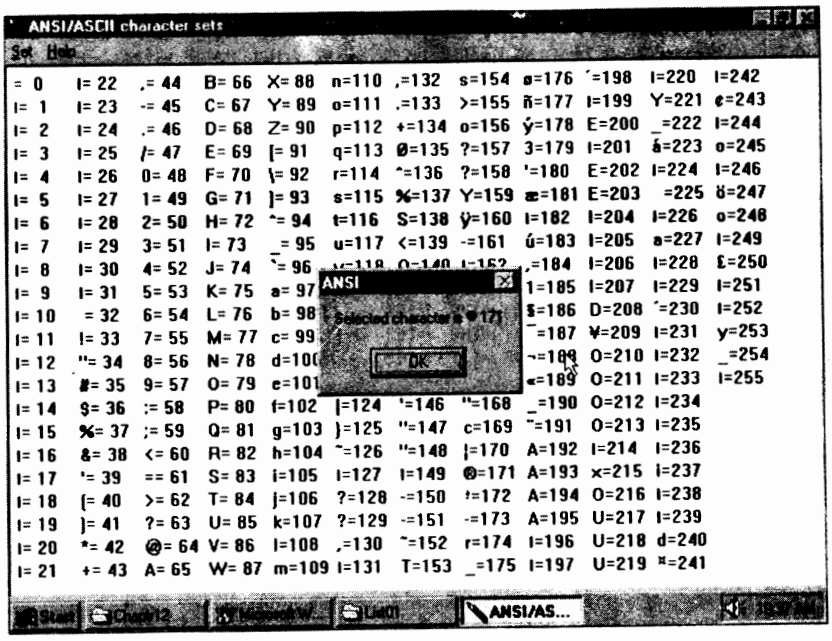


图 12-3　单击一个字符后会显示一个消息框，其中带有击中的字符

CharToOem () 和 OemToChar () 这两个 API 函数可以把用 ANSI 字符集表示的正文串转换成 ASCII 正文串，或者逆向转换。

```
#include〈winuser.h〉
BOOL CharToOem (LPCTSTR lpszSrc, LPSTR lpszDst);
```

参数	说明
LPCTSTR lpszSrc	准备转换的正文串
LPSTR lpszDst	转换后的正文串
返回值	在正文里讨论
BOOL	总是设定为 TRUE

这就是为了访问 ASCII 字符集在自己的源代码内需要完成的一切工作。请记住，255 个字符后半部分内的符号可能会根据系统的不同而发生变化，这取决于当时实际安装了什么样的代码页。

12.3　Unicode 和 Windows 95

Windows NT 在自己的系统级层次上采用了这种世界范围内通用的字符编码标准，通过它对字符和正文串进行处理。使用 Unicode 的主要区别以及优点在于它支持 16 位宽的字符，而不是传统的八位字符宽度。由于为每个符号都提供了加倍的空间，所以任何软件在任何语言里都能更好、更容易地实现本土化处理——即使在那些与西欧语言和字母毫不相干的语言

里也能很好地运行。

由于为每个字符保存了两个字节（16 位）的存储区域，所以我们也可以把 Unicode 字符串叫作“宽字符串”。就目前来说，Unicode 还只限于在 Windows NT 内使用，这就意味着一个纯正的 Unicode Win32 应用程序不能在 Windows 95 下正常地运行。尽管如此，这种情况并不意味着我们必须避免访问依赖于 Unicode 字符串的系统 API 函数。举个例子来说，在资源文件里用于定义扩展菜单模板的 MENUEX 命令就要求使用 Unicode 字符串。正如我们在 MENUEX 示范程序（参考本书附带 CD 的 Listing 6.2）里已经证明过的那样，这种只需要 Unicode 字符串的情况并没有把 MENUEX 资源限制在 NT 里使用。在那个例子里，我们用宽字符串定义了所有菜单项的正文，但是并没有碰到任何阻碍，这是由于资源编译器已经帮助我们完成了相应的转换。

假如我们准备在 MENUEX 命令的基础上建立一个 RAM 菜单模板，那么刚才讨论的情况就有所变化了。在这种情况下，我们必须用宽字符格式提供菜单项正文，尽管 Windows 95 还没有提供对这种格式的支持。我们能在 Windows95 里那样干吗？API 函数 MultiByteToWideChar()正是针对这目的而设计的，它能把一个字符串映射成宽字符 Unicode 串。

```
#include 〈winnls.h〉
int MultiByteToWideChar (UINT CodePage
                         DWORD dwFlags,
                         LPCSTR lpMultiByteStr,
                         int cchMultiByte,
                         LPWSTR lpWideCharStr,
                         int cchWideChar);
```

参数	说明
UINT CodePage	表 12-2 里列出的某个定义
DWORD dwFlags	表 12-3 里列出的一个或者多个标志
LPCSTR lpMultiByteStr	准备转换的字符串
int cchMultiByte	待转换字符串的长度
LPWSTR lpWideCharStr	用于接收转换后字符串的缓冲区
int cchWideChar	用于接收转换后字符串的缓冲区的长度
返回值	在正文里讨论
int	假如函数调用失败，就为一个零值；如果成功，就是转换后字符串内的字符数目，或者接收缓冲区 id cchWideChar 的长度为 0

宽字符串存储于类型为 WCHAR 的一个标识符内，这个 WCHAR 标识符是在 WINNT.H 里定义的：

```
typedef wchar_t WCHAR;
```

其中，wchar_t 是 BASETYPS.H 头文件内的一个无符号值。

```
typedef unsigned short wchar_t
```

类型为 LPWSTR 的标识符只是指向 WCHAR 的一个指针而已，它也是在 WINNT.H 里定义的，如下所示：

```
typedef WCHAR *LPWSTR, *PWSTR;
```

表 12-2 代码页定义

定 义	值	说 明
CP_ACP	0	ANSI 代码页
CP_OEMCP	1	OEM 代码页
CP_MACCP	2	MAC 代码页

表 12-3 用于 MultiByteToWideChar () 函数的标志

标 志	值	说 明
MB_PRECOMPOSED	0x00000001	肯定使用预先组合好的字符，其中的一个基础字符以及一个无间距字符有一个单一的字符值
MB_COMPOSITE	0x00000002	肯定使用合成字符，其中的一个基础字符以及一个无间距字符有不同的字符值
MB_USEGLYPHCHARS	0x00000004	使用局面字符取代控制字符
MB_ERR_INVALID_CHARS	0x00000008	假如碰到了一个无效的输入字符，就强迫函数把出错值设置成 ERROR_NO_UNICODE_TRANSLATION

正是由于每个字符使用了 16 个二进制位，所以我们才把一个 Unicode 字符串称为“宽字符串”，其中每个字符都有可能是 65536 个不同值中的任何一个。

MENUEX 资源接收了宽字符串以后，它可以对一个 RAM 菜单模板内的每个菜单项进行定义。为了达到一箭双雕的效果，现在让我们开发一个简单的应用程序来试试看。这个程序有助于我们复习关于菜单的知识，也能针对 Unicode 字符串进行一番实际的探讨。在本书附带 CD 的 Listing 12.2 内，大家能够找到一个名为 RAMMENU 的示范程序。这个程序向我们展示了如何在 Windows 95 里运用宽字符串，以及如何通过 RAM 菜单模板建立一个菜单。

根据 Win32 提供的文档资料，我们知道 RAM 菜单模板主要是两种对象类型的组合。这两种对象类型分别是 MENUEX_TEMPLATE_HEADER 和 MENUEX_TEMPLATE_ITEM。后面那种对象类型会重复几次，每个菜单项都需要使用一个。这些结构的总和定义了一个长度可变的内存块，它的起始处带有一个 MENUEX_TEMPLATE_HEADER 结构，该结构提供了一些常规的信息：

```
typedef struct
{
    WORD wVersion;
    WORD wOffset;
    DWORD dwHelpId;
```

} MENUEX_TEMPLATE_HEADER;

其中，wVersion 项的值在 Windows 95 里肯定应该设置成 1，而 wOffset 则指定了一定数目的字节，从而把这一项与第一个 MENUEX_TEMPLATE_ITEM 数据结构的起始处分隔开来。在 RAMMENU 示范程序里，用于 RAM 菜单模板的所有数据结构都是封装起来的，因此我们需要把 wOffset 项的值设置成 4，以此作为到下一个内存块的距离。最后，dwHelpId 是用于整个菜单的帮助标识符。当用户按下 F1 键，同时键盘输入焦点位于菜单资源提供的任何一个菜单项上方时，标识值就会在 dwHelpId 里出现。假如与某个顶级菜单关联在一起的菜单资源属于 MENUEX 类型，那么这个值就可以通过调用 GetMenuContextHelpId () 函数取得，也可以通过调用函数 SetMenuContextHelpId () 进行设置。

MENUEX_TEMPLATE_HEADER 这个初始化数据结构后面跟随的是一系列 MENUEX_TEMPLATE_ITEM 结构，这种结构的定义如下所示：

```
typedef struct
{
    DWORD dwType;
    DWORD dwState;
    UINT uId;
    WORD bResInfo;
    WCHAR szText [1];
    //DWORD dwHelpId;
} MENUEX_TEMPLATE_ITEM
```

MENUEX_TEMPLATE_ITEM 结构提供了用于定义一个弹出式菜单项的所有信息。相同的结构亦可应用于用 MENUITEM 命令定义的菜单项，唯一的区别在于其中的 dwHelpId 项上面，这个项在 MENUITEM 里是不支持的。因此，我们应该定义第三个数据结构，从而反映出这种细微的区别。RAMMENU 示范程序向我们展示了一种另外一种可选方案，它既满足 MENUITEM 的需求，又能满足 POPUP 菜单项的需求，同时不用定义两个单独的数据结构。

这个结构中的某些项反映了崭新的 Win32MENUITEMINFO 结构的信息。MENUITEMINFO 数据结构是针对菜单对象的有效管理而专门设计的。例如，dwType 可以采用表 6-4 内列出的任何一种定义。类似地，dwState 也可以是任何 MFS_风格的有效组合（对这些 MFS_风格的总结请参考表 6-5）。扩展菜单不仅要为通过 MENUITEM 命令定义的菜单项分配 ID，也要为 POPUP 菜单项分配 ID。uId 项在 ID 的分配过程中直到了关键性的作用。

bResInfo 项的地位是相当重要的。这个单一的字节域可以指示 API 函数 LoadMenuIndirect () 根据菜单项（既可以是弹出式菜单项，也可以是普通菜单项）的本质进行处理。假如某个菜单项是当前现场内的最后一个信息块，那么也可以通过 bResInfo 指出这一点。把信息存储到一个内存块内，从而建立一个 RAM 菜单模板的时候，我们首先要指定最左边的一个顶级菜单，这通常是 File。这是一个弹出式菜单，要求把 0x01 值分配给 bResInfo 项。下一个内存块（类型为 MENUEX_TEMPLATE_ITEM 的另外一个数据结构）需要指定

与那个弹出式菜单关联在一起的下拉式菜单内的第一个菜单项。例如，我们可以假设这个菜单内只包含了三个菜单项。这三个结构相互之间可以紧挨在一起，只需在前面加上另外一个MENUEX_TEMPLATE_ITEM 结构即可。这个结构里包含了用于 File 顶级菜单的信息。这个序列的最后那个内存块（用于定义第三个菜单项）在 bResInfo 项内提供了一个 0x80 值，从而表明自己是下拉式信息块内的最后一项。为了增强源代码的可读性，在 RAMMENU 这个示范程序里，我们已经进行了如下所示的定义：

```
#define MFR_END 0x80
#define MFR_POPUP 0x01
```

对于指定最右侧那个顶级菜单（通常是 Help）的内存块来说，它在 bResInfo 项里同时提供了 MFR_END 和 MFR_POPUP。设置一个次级弹出式菜单的时候，相同的规则也是适用的。引入次级下拉式菜单的那个菜单项应该分配一个 MFR_POPUP，那个下拉式菜单内的最后一个菜单项则应分配一个 MFR_END。

szText 项是一个长度可变的数组，类型为 WCHAR。在这个数组里，我们需要存储菜单项的名字。单独一个字符显然就足以容纳整个正文串了。这样一来，开发者就必须分配一个更大的内存块，以便与这些长度可变的正文串相适应。最后，我们专门保存了 dwHelpId，它只能用于通过 POPUP 命令声明的项目。假如用户按下 F1 键向附加的支持信息发出请求，同时当时正好有一个弹出式菜单处于高亮度显示状态，这时就需要用到 dwHelpId 里的信息，利用它可以和帮助引擎进行交互作用。

图 12-4 向大家展示了 RAMMENU 程序的显示情况，它的菜单是在程序执行过程中通过调用 LoadMenuIndirect（）函数动态地建立起来的。从表面上看起来，以 RAM 菜单模板为基础的一个菜单与从资源文件里取得的菜单是完全一致的。只有分析其源代码，我们才能看出它们建立菜单对象的不同途径。MENUEX_TEMPLATE_HEADER 和 MENUEX_TEMPLATE_ITEM 数据结构不是在开发环境的任何一个头文件里声明的，因此它们会排布于 RAMMENU 源代码的起始处。除此以外，RAMMENU 示范程序还定义了第三种数据结构：MENUEX，该结构明显是为了简化新菜单项在 RAM 菜单模板内的插入而专门设计的。

```
typedef struct _MENUEX
{
    PMENUEX_TEMPLATE_ITEM pMenuItem;
    DWORD uHelpID;
    LPSTR pszText;
    int cchText;
} MENUEX, *PMENUEX;
```

RAM 菜单模板的创建需要分配一个内存块，用它来容纳所有数据结构。进程堆是用于保存一系列内存页的最佳场所：

```
LPBYTE pTemp = (LPVOID) HeapAlloc (GetProcessHeap (), HEAP_ZERO_
```

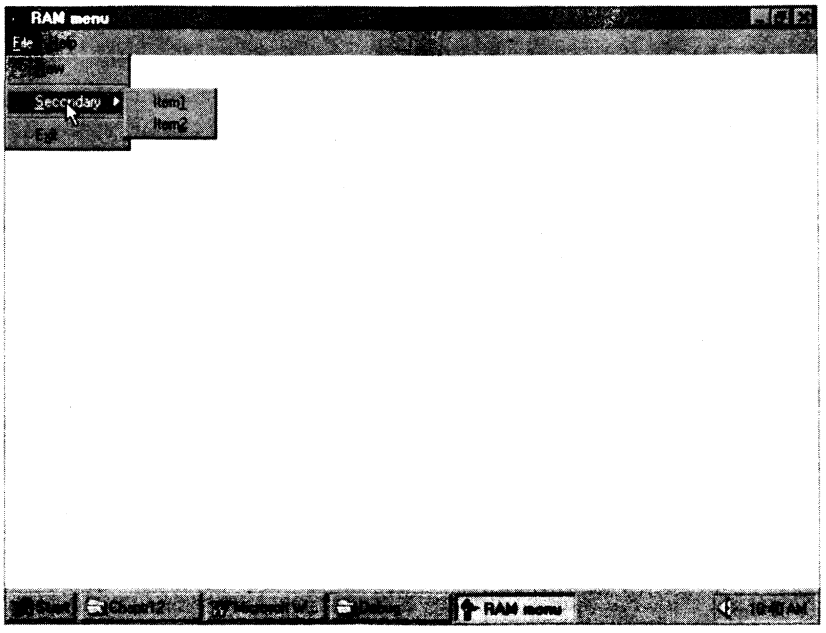


图 12-4　RAMMENU 应用程序的菜单是利用一个 RAM 菜单
模板在运行期间建立的

MEMORY，10000)；

这个操作是在由应用程序定义的例程 MakeTheMenu () 里实现的。我们为整个菜单分配了一个 ID：100，而 wOffset 项则设置成与 dwHelpId 项的尺寸相等。

```
...
PMENUEX_TEMPLATE_HEADER pmh = (PMENUEX_TEMPLATE_
    HEADER)pTemp;
...
// fill in the header information
pmh -> wVersion = 1;
pmh -> wOffset = sizeof(pmh -> dwHelpId);
pmh -> dwHelpId = 100;

pmi = (PMENUEX_TEMPLATE_ITEM)((LPBYTE)pmh + sizeof
    (MENUEX_TEMPLATE_HEADER));
...
```

完成了这种初始化工作以后，应用程序接起来就可以计算出第一个 MENUEX_TEMPLATE_ ITEM 数据结构开始的内存位置，接着再把 MENUEX_ TEMPLATE_HEADER 数据结构的起始位置考虑进去。

```
...
PMENUEX_TEMPLATE_ITEM pmi;
...
pmi = (PMENUEX_TEMPLATE_ITEM)((LPBYTE)(pmh) + sizeof
    (MENUEX_TEMPLATE_HEADER));
...
```

对于每个菜单项的插入来说，无论是弹出式菜单项还是一般（下拉）菜单项，都要由另外一个名为 AddMenuItemEx () 的例程进行控制，该例程的作用是接收指向某个 MENUEX 数据结构的指针。在调用这个函数之前，应用程序会先在 MENUEX 数据结构里填写相应的信息。一些菜单项可以立即到达它们在 PMENUEX_TEMPLATE_ITEM 数据区域内的目标位置，另外一些则需要临时性封装到 MENUEX 的其他一些项里。AddMenuItemEx () 例程可以返回一个新内存位置的地址，其中应该放置下一个 PMENUEX_TEMPLATE_ITEM 数据结构。因此，项目在 RAM 菜单模板内的插入就变成了一种非常简单的操作，正如下面这个代码段向我们展示的那样。这个代码段与 File 顶级菜单以及 New 菜单项的插入有关。

```
...
// add the File popup
mex.pMenuItem = pmi;
pmi -> dwType = MFT_STRING;
pmi -> dwState = MFS_ENABLED;
pmi -> uID = 777;
pmi -> bResInfo = MFR_POPUP;
mex.uHelpID = 100;
mex.pszText = "&File";
mex.cchText = lstrlen("&File");
pmi = AddMenuItemEx(&mex);

// add the New item to this popup
mex.pMenuItem = pmi;
pmi -> dwType = MFT_STRING;
pmi -> dwState = MFS_ENABLED;
pmi -> uID = MN_NEW;
pmi -> bResInfo = 0;
mex.uHelpID = 0;
mex.pszText = "&New";
mex.cchText = lstrlen("&New");
pmi = AddMenuItemEx(&mex);
...
```

AddMenuItemEx () 例程可以把正文转换成一个相应的宽字符串，计算出所占内存区域的结束位置，然后检查每个菜单项里是否设置了 MFR_POPUP 位。假如设置了这个位，它就会向那个弹出式菜单增加帮助 ID 值：

```
...
if(IsBadReadPtr(pmex -> pszText, 1))
    nChars = 0;
else
{
    nChars = MultiByteToWideChar(CP_ACP, 0, pmex -> pszText, pmex ->
                        cchText,pmex -> pMenuItem -> szText, 64) + 1;
}

// add the help id
lpReturn = (LPBYTE)(pmex -> pMenuItem) +(sizeof(MENUEX_
            TEMPLATE_ITEM) - sizeof(pmex ->pMenuItem ->
 szText)) +(nChars * sizeof(WCHAR));
if(pmex -> pMenuItem -> bResInfo & MFR_POPUP)
{
    * (DWORD * )lpReturn = pmex -> uHelpID;
    lpReturn += sizeof(DWORD);
}
...
```

在这种信息填写处理将要结束的时候，剩下的最后一个操作是把用于描述一个菜单模板的整个信息块转换成相应的菜单栏。这个操作是通过调用 LoadMenuIndirect () 函数来完成的。在这以后，菜单控件就会返回，同时把它与主窗口关联到一起：

```
...
// create the menu from the RAM template
hmenu = LoadMenuIndirect(pmh);

// free the memory block
HeapFree(GetProcessHeap(), 0, pmh);

return hmenu;
...
```

RAMMENU 示范程序向我们证明了建立一个 RAM 菜单模板以及在 Windows 95 里使用宽字符串有多么容易！这个例子第三个值得注意的特性是它与 WM_HELP 消息的拦截有关。MENUEX 资源是建立一个尽管复杂然而能够“对用户友好”的 Win32 应用程序的最恰当工具，因为这种资源提供了与系统帮助引擎更好的集成度。在选中一个菜单项的前提下，假如按下 F1 键，应用程序就会在窗口进程里接收到一条 WM_HELP 消息。用于这条消息的 lParam 正是指向 HELPINFO 数据结构的一个指针：

```
typedef struct tagHELPINFO
{
    UINT cbSize;
    int iContextType;
    int iCtrlId;
    HANDLE hItemHandle;
    DWORD dwContextID;
    POINT MousePos;
} HELPINFO, *LPHELPINFO
```

这条消息有可能是从不同的出发点到达同一个窗口进程的。它可能是在某个菜单项高亮度显示时按下 F1 键的结果，也有可能是在对话框的某个控件上方单击鼠标右键的结果。对于前一种情况来说，iContextType 项的值为 HELPINFO_MENUITEM (0x0002)；对于第二种情况，假如引用的是一个控件，这个值就是 HELPINFO_WINDOW (0x0001)。iCtrlId 项内包含了高亮度项目的 ID 或者一个子窗口的 ID。假如引用的是一个菜单，同时这个菜单是按照 MENUEX 规范设计的，那么它既可能是一个弹出式菜单的 ID，也有可能是一个菜单项的 ID。对控件窗口进行处理的时候，句柄项 hItemHandle 就显得相当有用了。它的值指定了窗口句柄。需要提供的最后一种信息是以屏幕坐标系统表达的鼠标位置。

拦截了一条与菜单项有关的 WM_HELP 消息以后，通过对 dwContextId 的分析，我们就可以判断出需要对哪个菜单项进行处理。请注意，对于和应用程序关联在一起的所有菜单来说，分别都需要用一个唯一的 dwContextId 进行标识。除此以外，iCtrlId 还标识了当前以高亮度显示的菜单项，从而指定了准备调用的帮助主题。WinHelp () 函数的作用就是载入 Windows 95 帮助引擎，依次载入相关的 .HLP，并且激活相应信息的显示。

12.4 鼠标

作为指点设备的鼠标是 Windows 里迄今为止最好的输入工具。和键盘不一样，鼠标的行动显得相当独立。这是由于在缺省情况下，它并不局限于只能在某一个窗口内使用。鼠标的基本工作原理是相当简单的。对于用户在桌面的每一次运动，或者按钮选定来说，鼠标都只会把相关事件的信号发送给位于指针热点下方的那个窗口。因此，连接了鼠标以后，与活动窗口或者键盘输入焦点联系在一起的概念就显得没有什么意义了。图 12-5 向我们证明了这一点。正如我们看到的那样，MS Word 7.0 现在是活动窗口（标题栏颜色是最深的），而鼠标指针正位于属于另一窗口的某个区域上方，这个窗口的名字叫作 MMOVE——在本书附带 CD

的 Listing 12.3 里可以找到的一个示范程序。这个应用程序的作用是在自己的标题栏内显示出鼠标指针的坐标位置。

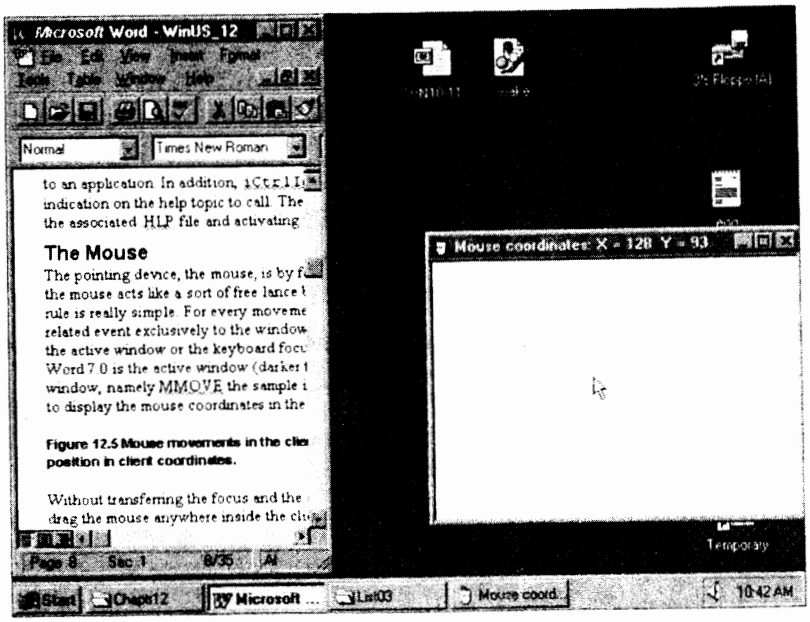


图 12-5　鼠标在客户区内的运动能在标题栏内反映出来，
从而指出鼠标指针当前在客户区坐标系统内的位置

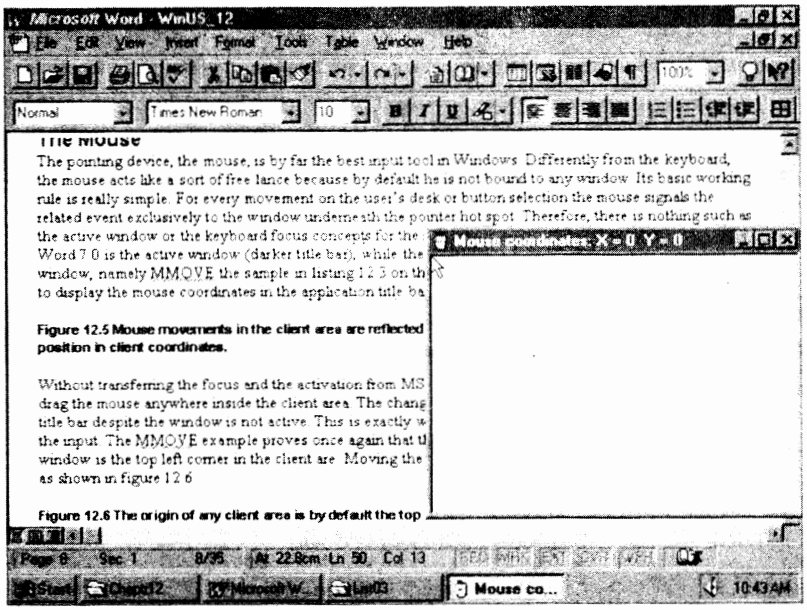


图 12-6　在缺省情况下，任何客户区的起点都是左上角

不用把输入焦点和活动性从 MS Word 转移给标记为“Mouse coordinates”的窗口，只需在客户区内到处拖动鼠标就可以了。尽管窗口并未处于活动状态，但是指针位置的改变一样会在那个应用程序的标题栏内反映出来。这是对鼠标相对独立性的一个很好的证明。MMOVE 示范程序再次证明了标准窗口的缺省坐标起点在客户区的左上角位置。假如把鼠标指针移动到那个地方，就会显示出（0，0）坐标，如图 12-6 所示。

鼠标可以通过一系列特定的消息来指出自己的位置和按钮状态，对这些消息的总结请大家参考表 12-4。这十条消息反映了所有可能的行动。除了 WM_MOUSEMOVE 以外（它的作用是指出指点设备在用户桌面的移动），假如按下一个鼠标按钮，就会生成一条 WM_xBUTTONDOWN 消息；假如接着松开了按钮，就会立即生成一条相应的 WM_xBUTTONUP。一次双击事件会生成一条 WM_xBUTTONDBLCLK 消息。在刚才提到的消息里，小写的 x 应该用 R，L 或者 M 替换，这三个字母各自表示按下的是鼠标左键、右键以及中键。

对于所有这些消息来说，最通用的一个特性就是在其 lParam 内存放了鼠标指针在屏幕内的当前位置，这个位置是用客户区坐标表达的。这个位置不能用其他方式表达，因为一个窗口必须“知道”与自己坐标空间有关的鼠标位置。利用 ClientToScreen（），ScreenToClient（）以及 MapWindowPoints（）函数，很容易就可以把这种值从一个坐标空间切换到另外一个坐标空间。wParam 的内容对于所有鼠标消息来说也是通用的。其中存放了一个或者多个 MK_定义，这些定义反映了 Ctrl 和 Shift 控制键的当前状态，如表 12-5 所示。

表 12-4　基本的鼠标消息

消　息	值	说　明
WM_MOUSEMOVE	0x0200	只要鼠标在桌面发生了移动，就送出这条消息
WM_LBUTTONDOWN	0x0201	左鼠标按钮按下时发出
WM_LBUTTONUP	0x0202	左鼠标按钮松开时发出
WM_LBUTTONDLBCLK	0x0203	左鼠标按钮连续点击两次时发出
WM_RBUTTONDOWN	0x0204	右鼠标按钮按下时发出
WM_RBUTTONUP	0x0205	右鼠标按钮松开时发出
WM_RBUTTONDLBCLK	0x0206	右鼠标按钮连续点击两次时发出
WM_MBUTTONDOWN	0x0207	中鼠标按钮按下时发出
WM_MBUTTONUP	0x0208	中鼠标按钮松开时发出
WM_MBUTTONDLBCLK	0x0209	中鼠标按钮连续点击两次时发出

表 12-5　存储于 wParam 内的鼠标消息标志

鼠标消息标志	值	说　明
MK_LBUTTON	0x0001	鼠标左键按下
MK_RBUTTON	0x0002	鼠标右键按下
MK_SHIFT	0x0004	Shift 键按下
MK_CONTROL	0x0008	Ctrl 键按下
MK_MBUTTON	0x0010	鼠标中键按下

WM_MOUSEMOVE	0x00A0
wParam	控制按键标志
LOWORD (lParam)	鼠标指针的水平位置
HIWORD (lParam)	鼠标指针的垂直位置

我们可以通过 MAKEPOINTS 宏提取出相应的 X 和 Y 坐标，如下所示：

```
...
case WM_MOUSEMOVE:
{
    POINT pt;

    pt.x = MAKEPOINTS(lParam).x;
    pt.y = MAKEPOINTS(lParam).y;
    ...
}
...
```

通过对 wParam 进行检查，一个应用程序就可以实现需要同时用到鼠标和键盘的某些操作。举个例子来说，在几乎每个 Microsoft 应用程序里，假如在按住 Ctrl 键不放的同时用鼠标进行选择，就能把一个新对象增添到当前的选定项目池里。

MMOVE 向我们提出了与 Win32 窗口整体结构有关的另外一个关键问题。假如把鼠标热点放置于属于标题栏（或者重定义尺寸边框）的象素上方，鼠标的坐标就不再更新了。通过这个简单的操作，我们可以得到一个有趣的结论。标题栏以及其他所有非客户区组件都是与客户区互相独立的对象。尽管对它们的处理完全不同于客户区，但是我们仍然不可能把它们归为不同的窗口类。因此，我们几乎完全可以断言：注册一个新的窗口类几乎完全等价于定义一种新的客户区类型。

另外一项简单的测试可以对我们刚才描述的概念进行补充。假如双击标题栏，会发生什么事情呢？根据它当前的状态，窗口会以最大化显示，或者恢复成它的原始大小。现在双击客户区域，然后对 WM_LBUTTONDBLCLK 进行跟踪。在这种情况下，除非用 CS_DBLCLKS 标志注册了这个窗口类，否则什么事情也不会发生。用这个标志注册以后，就可以实现对客户区内双击事件的支持。关于这方面的主题，我们会在本章的后续部分进行详细的介绍。就双击标题栏这样的操作来说，它会生成一条 WM_NCLBUTTONDBLCLK 消息。这是一条非客户区消息，它会传递给应用程序窗口进程，从而通知它标题栏内发生了双击事件。

鼠标单击

用鼠标进行单击肯定是 PC 用户最自然的一种行动。Windows 95 极大拓展了鼠标右键的功能，利用它可以完成外壳内一些关键任务。每个对象都应该有它自己的关联菜单，其中列出了由那个对象支持的操作。单击鼠标右键（右击）的时候，应用程序就会依次接收到 WM_RBUTTONDOWN，WM_RBUTTONUP 以及 WM_CONTEXTMENU 这三条消息。正如通

过名字可以看出的那样，我们最好把与关联菜单有关的所有代码都放置于一条 WM_CONTEXTMENU 消息里，在另外两条消息里则完成一些更为传统的任务。

在本书附带 CD 的 Listing 12.4 里，大家可以找到一个名为 MCHILD 的程序示例。该程序向大家展示了如何利用指点设备来实现几种特殊的行为。如图 12-7 所示，应用程序没有提供菜单栏，而是在自己的客户区内容纳了许多子窗口。这些子窗口是通过在客户区任何一个地方单击鼠标左键而生成的。

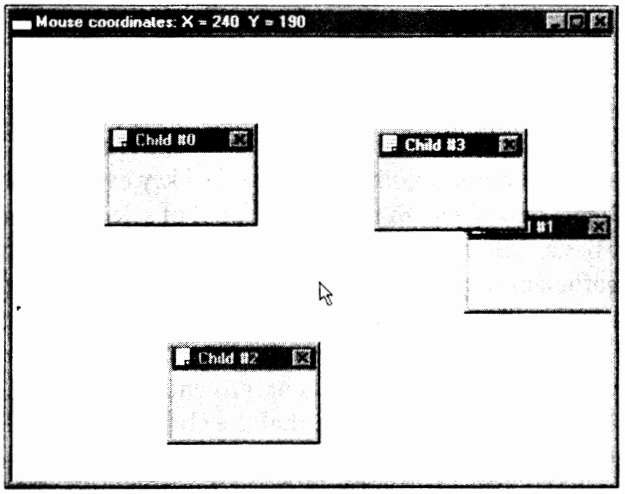


图 12-7　MCHILDE 程序展示了如何对鼠标事件进行捕获

每个文档（子）窗口都带有一个关联菜单，其中只列出了两个菜单项。假如用鼠标选中一个现成的文档，就会生成一条 WM_MOUSEACTIVATE 消息，这条消息会直接发送给文档窗口进程。在这个消息块里，应用程序会取回当前活动文档的句柄，把它与自己的句柄比较，并且最终为新选定的子窗口分配一个活动标题栏颜色。所有这些操作都必须在源程序里进行编码，因为每当牵涉到子窗口的时候，窗口管理程序都不会自动实现这些处理。

```
...
case WM_MOUSEACTIVATE:
{
    HWND hwndpapa = (HWND)wParam;
    HWND hwndActive;

    // is there an active doc?
    hwndActive = (HWND)GetWindowLong(hwndpapa, GWL_USERDATA);
    // check if it is the same document
    if(hwndActive == hwnd)
        break;
```

```
        // remove the activation bar
        SendMessage(hwndActive, WM_NCACTIVATE, FALSE, 0);

        // set the activation bar
        SendMessage(hwnd, WM_NCACTIVATE, TRUE, 0);
        // store the handle
        SetWindowLong(hwndpapa, GWL_USERDATA, (long)hwnd);
        // bring it to the top
        SetWindowPos(hwnd, HWND_TOP, 0, 0, 0, 0, SWP_NOMOVE |
            SWP_NOSIZE);
    }
        break;
...
```

请注意，标题栏内的鼠标坐标只有在指针位于应用程序客户区内的一个空白位置时才会发生更新。把鼠标移动到一个文档上方以后，就中断了向主窗口进程发送的信息流。信息流这时的目的地变成了相应的文档窗口进程。

12.5 鼠标捕获

新文档的创建是在用户松开了鼠标按钮以后进行的。我们很难得有机会对按钮松开事件进行处理，但在与鼠标捕获有关的 MCHILD 程序里却是个例外。假如在按住 Shift 键不放的同时按下了鼠标右键，我们就对鼠标进行了捕获。这种说法基本上是不难理解的。在缺省情况下，鼠标扮演着一种独立输入设备的角色，它只会向位于当前指针热点下方的窗口发出通知消息。通过对鼠标进行捕获，我们就可以改变这种行为，它能强制性地把鼠标消息发送一个单一的窗口，无论这个窗口当前是否位于鼠标热点的下方。

显然，这种捕获需要用到一个特定的线程队列。和 Windows 3.x 不一样，以前的普遍规则已经不适用了，所有窗口并非都限制在同一个输入机制里。这样一来，由于考虑到在 Windows 95 里，一旦退出窗口边界，另外一个线程队列就会控制这条鼠标消息，所以对鼠标进行捕获还有什么好处吗？这个问题的答案显得有些模棱两可：有好处，也有可能没有好处！对于某些应用程序来说，即使指点设备正好位于窗口矩形以外，程序也需要用到鼠标消息。Spy++工具程序就是这样的一个典型例子。在 Windows 95 里，因为存在几个独立的线程队列，所以就带来了这样的一个限制：假如鼠标位于准备捕获鼠标的那个窗口以外，而此时仍然希望在这个窗口里希望接收到鼠标消息，就必须保持鼠标的按下状态。从本质上讲，在位于客户区以内时，只要鼠标按钮处于按下状态，那么尽管 WM_MOUSEMOVE 消息以及与按下某个按钮有关的某条消息到达了用于捕获的那个窗口，我们仍然可以对鼠标进行捕获。

我刚才描述的其实正是 Spy++在 Find 窗口菜单项引入的对话框里的工作原理。MCHILD 程序支持的鼠标捕获是通过按下 Shift 键不放，然后单击鼠标右键来实现的。在这以后，假如按下 Ctrl 键，然后单击鼠标右键，就可以实现鼠标的释放。

```
...
case WM_RBUTTONDOWN:
{
    if(wParam & MK_SHIFT)
    {
        // capture the mouse
        SetCapture(hwnd);
        // play the message
        PlayResource(hInstance, "captured");
    }

    if(wParam & MK_CONTROL)
    {
        // release the mouse
        ReleaseCapture();
        // play the message
        PlayResource(hInstance, "released");
    }
}
    break;
...
```

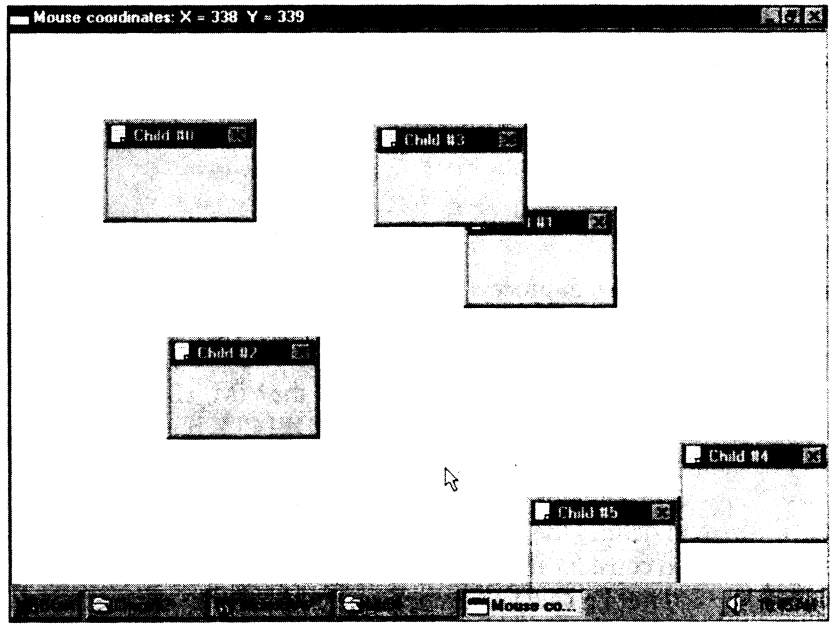


图 12-8　在捕获了鼠标的前提下，文档窗口可以在最初的可见客户区以外建立

除此以外，一条声音消息也会播放出来，从而通知用户已经进行了什么操作。假设现在已经捕获了鼠标，然后按下鼠标左键，这时就可以在屏幕上到处拖动它。标题栏会不断地更新，从而反映出新的鼠标位置。假如把鼠标移到左上角以上的地方，就会发现标题栏内出现了负值。这并不是编程上的错误，而是一种正确的结果。事实上，大家必须记住这样一点，X轴上的正值是从左到右增大的，Y轴上的正值则是从上到下增大的。位于客户区左侧和顶部起点以外的一个点不可避免地会成为负值坐标。

鼠标指针位于客户区矩形以外的时候，假如这时松开了鼠标左键，一条WM_LBUTTONUP消息就会发送给应用程序的窗口进程。这个事件将触发一个新文档的建立，建立的那个文档正好位于鼠标指针的当前位置处。对于客户区处的一个点来说，在那儿建立的文档窗口我们是看不见的。在这个时候，我们可以释放鼠标捕获，然后使主窗口放大显示。刚才建立的那个文档窗口就会显示出来，如图12-8所示。

正如我们在本章稍后讲述的那个WINSPY程序一样，对鼠标进行捕获是一种相当有用的技术，利用它可以发现和取回属于其他进程的某个窗口的信息。

12.6 鼠标双击

如果想打开一个文件夹或者启动一个新的应用程序，就必须在一个对象上方双击。对于PC机的新用户来说，双击并不是一种很自然的行动，刚开始的时候多少会碰到一些困难，会产生不习惯的感觉。Windows支持双击事件，然而这种事件并不是以一个整体的面貌出现的。事实上，双击只不过是两次连续的单击事件，中间没有移动鼠标指针的时间。用户双击的时候，他（她）首先会按下鼠标左键（WM_LBUTTONDOWN），然后松开它（WM_LBUTTONUP），紧接着再按下（第二条WM_LBUTTONDOWN）。只有到这个时候（刚刚第二次按下），系统才会检查刚才描述的两次“按下－松开－按下”操作是否符合双击条件。假如内部的这种测试通过，系统就会用一条WM_LBUTTONDBLCLK取代刚才的第二条WM_LBUTTONDOWN消息。等到出现最后一条WM_LBUTTONUP消息以后，生成的所有消息都会保留下来，就像下面这个样子：

```
WM_LBUTTONDOWN
WM_LBUTTONUP
WM_LBUTTONDBLCLK
WM_LBUTTONUP
```

这就意味着假如希望对同一窗口进程内的WM_xBUTTONDBLCLK和WM_xBUTTONDOWN消息进行处理，就必须采用一种具有相当智能的方案，从而把它们区分开来。否则，每次用户双击的时候，应用程序首先就会执行WM_xBUTTONDOWN消息内的代码，然后再执行WM_xBUTTONDBLCLK内的代码，这是我们不愿看到的。在本书附带CD的Listing 12.4里，大家可以找到一个名为DBLCLK的示范程序，它向我们展示了怎样解决这个有趣的问题。

首先，系统要定义两次连续按键之间的最大间隔时间，在这段时间内连续单击就会合并成一次双击事件考虑。这种时间信息可以通过调用GetDoubleClickTime（）这个API函数获得。类似地，利用SetDoubleClickTime（）则可对时间进行设置。

```
#include〈winuser.h〉
```

```
void WINAPI SetDoubleClickTime (UINT wCount);
```

参数	说明
UINT wCount	用毫秒表示的间隔时间
返回值	在正文里讨论
void	没有返回值

在缺省情况下，Windows 95 把双击间隔时间设置成 500 毫秒，这是一个中等程度的值。末端用户可以利用鼠标属性表对这个值进行间接性的修改。如图 12-9 所示。

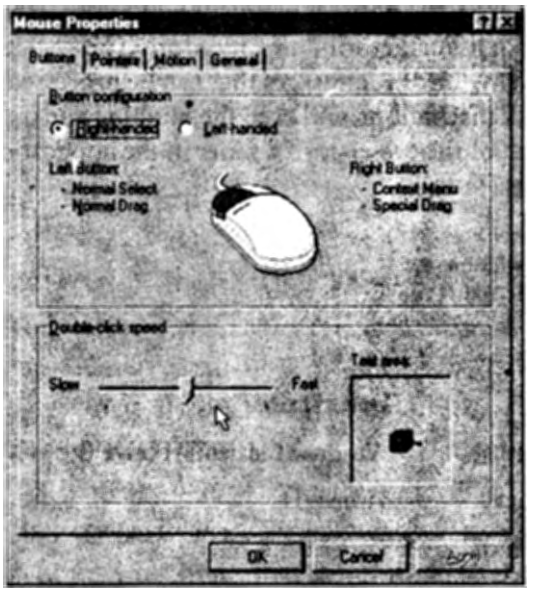


图 12-9　在鼠标属性表内修改双击速度

最后，假设我们希望建立一个特殊的应用程序。假如用户用鼠标左键单击，单击区域就会显示出一个按钮。除此以外，假如用户用鼠标左键双击，就开始在客户区内放映一段影象剪辑。这其实是某些软件作品的典型做法。为了达到我们的目标，首先必须掌握关于计时器的知识。

对计时器的理解

计时器可以想象成一种输入设备，尽管它实际上与任何物理设备都是不相干的。准确地说，计时器是一种 Windows 内部组件。为计时器设定的时间溢出时，它就生成一条 WM_TIMER 消息。

假如应用程序每隔一段时间就要去执行一项特定的任务，那么计时器就有用武之地了。在本书附带 CD 的 Listing 9.3 里，我们曾经提到一个示范程序，它的工作是在客户区内列出系统内当前正在运行的所有进程。这个程序其实是应该使用计时器的一个最佳候选者。在这个程序里，如果使用了计时器，就可以强迫它每隔一定时间就对运行进程列表进行更新。例如，我们可以让它每隔五秒钟的时间就重新执行一遍对运行进程的内部系统列表进行检查的代

码。需要注意的是，我们不建议过于频繁地接收 WM_TIMER 消息。WM_TIMER 消息与 Windows 的标准多任务规则是相悖的。假如应用程序正在处理一条特殊的消息，这条消息使 CPU 在很长一段时间内都处于“忙”状态，那么 WM_TIMER 消息就不能到达目标窗口进程。这样一来，系统就会丢弃这条消息，只发送那些可以及时处理的 WM_TIMER 消息。

鼠标双击机制，甚至于字处理程序内光标的闪烁，它们都是在应用程序内使用一个计时器的结果。我们可以调用 SetTimer () 函数来设置一个计时器：

```
#include 〈winuser.h〉
UINT SetTimer (HWND hwnd,
               UINT idTimer,
               UINT uTimeout,
               TIMERPROC tmprc);
```

参数	说明
HWND hwnd	用于接收 WM_TIMER 消息的窗口的句柄
UINT idTimer	计时器 ID
UINT uTimeout	计时器溢出时间
TIMERPROC tmprc	计时器进程
返回值	在正文里讨论
UINT	计时器 ID

窗口句柄指定了某个特定的窗口进程，这个进程准备接收与那个计时器相关的 WM_TIMER 消息。第二个参数是 idTimer，这是分配给新计时器的一个临时性 ID。这个函数返回的才是实际的计时器 ID——这个数字也许会与 idTimer 中的值不符。计时器溢出时间是用毫秒数表示的，并且代表了接收 WM_TIMER 消息时希望的重复频率。最后，计时器可以拥有一个关联在一起的计时器进程。每次当计时器时间溢出以后，就会调用这个进程内的代码。在这儿，我们一般都愿意接收来自计时器的以消息形式表达的通知信息，而不是专门去设计一个独立的进程。因此，最后一个参数可以设置成 NULL。一条 WM_TIMER 消息到达窗口进程以后，它在其中的 wParam 里包含了自己的 ID。对于同一个窗口来说，我们允许为它关联多个计时器。

为了消除一个计时器，我们可以调用 KillTimer ()：

```
#include 〈winuser.h〉
BOOL KillTimer (HWND hwnd, UINT uIDEvent);
```

参数	说明
HWND hwnd	用于接收 WM_TIMER 消息的窗口
UINT uIDEvent	计时器 ID
返回值	在正文里讨论
BOOL	假如计时器成功地消除了，就返回一个 TRUE 值

在DBLCLK示范程序里，假如用户在客户区内单击鼠标左键，就会启动一个新的计时器。它的溢出时间与前面通过GetDoubleClickTime ()取得的双击间隔时间是一致的。缺省情况下，针对用户在客户内的一次鼠标左键单击事件采取相应的行动前，应用程序会等待半秒钟的时间。接收到WM_TIMER消息以后，就表示用户确实希望在客户区进行单击，而不是双击。在这个时候，应用程序就会执行与这个事件关联在一起的行动——消除计时器的存在。为了限制用户的等待时间，建议把双击间隔时间设置在300毫秒到500毫秒之间。设置更高的值会导致不能忍受的延时，从而使用户怀疑应用程序是否已经挂起，或者是否不能执行自己希望的命令。在这些情况下，我们尽量让用户单击数次以后就能满意地获得他们希望达到的结果。

```
...
case WM_LBUTTONDOWN:
{
    // start the timer
    SetTimer(hwnd, ID_TIMER, wTime, NULL);

    // is it a double click?
    if(fTime)
    {
        // post a fake double click message
        PostMessage(hwnd, WM_MYDBLCLK, 0, 0);
        // reset the switch
        fTime = FALSE;
        // kill the timer
        KillTimer(hwnd, ID_TIMER);
        break;
    }
    // this is the first click
    fTime = TRUE;
}
    break;
...
```

应用程序接收到一条WM_TIMER消息以后，它就会把计时器删除掉，把布尔值标志设置成FALSE，然后发送一条“伪造”的单击消息，如下所示：

```
...
case WM_TIMER:
    switch(wParam)
```

```
    {
        case ID_TIMER:
        {
            POINT pt;

            // kill the timer
            KillTimer(hwnd, wParam);
            // reset the boolean flag
            fTime = FALSE;

            // get the current location
            GetCursorPos(&pt);
            // convert it
            ScreenToClient(hwnd, &pt);
            // post a fake single click message
            PostMessage(hwnd, WM_MYSINGLECLK, 0, MAKELPARAM
                (pt.x, pt.y));
        }
            break;
    }
    break;
...
```

图 12-10 向大家展示了 DBLCLK 程序刚刚执行后的显示情况。本书的封面经过一番处理以后，占据了客户区的背景空间。假如在背景位置上单击鼠标左键，窗口的尺寸就会自动调整，从而与客户区内的 AVI 影像剪辑匹配，然后把它播放出来。假如用鼠标左键双击，一段不同的 AVI 影像就会播放出来，如图 12-11 所示。

主窗口客户区临时性地由一个 MCI 窗口覆盖，这个窗口在最初的时候是看不见的。同时，整个窗口会根据影像剪辑的格式调整自己的大小。MCIWndOpen () 宏函数可用于载入对应的 AVI 影像，而 MCIWndPlay () 则可以在窗口可见以后把它播放出来。下面这个代码段为我们阐述了这一阶段涉及到的几个步骤：

```
...
case WM_MYDBLCLK:
{
    RECT rc;
    HWND hwndMovie = CTRL(hwnd, CT_MOVIE);

    // load the AVI
```

图 12-10　DBLCLK 程序能够区别对待单击事件的相关行动
与双击事件的相关行动

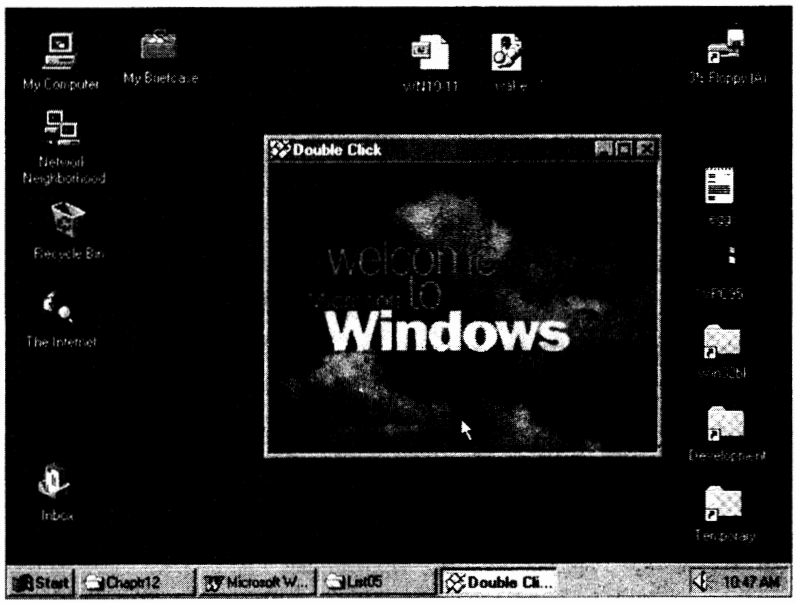


图 12-11　用户在客户区内任何地方双击时，影像剪辑就会播放出来

```
if(MCIWndOpen(hwndMovie, "WELCOME.AVI", WS_CHILD |
```

```
        MCIWNDF_NOTIFYMODE |MCIWNDF_NOPLAYBAR))
            MessageBeep(0);

        // get the movie window size
        GetWindowRect(hwndMovie, &rc);
        AdjustWindowRect(&rc, WS_CAPTION, FALSE);
        SetWindowPos(hwnd, HWND_TOP, 0, 0, rc.right - rc.left, rc.bottom -
        rc.top,SWP_NOMOVE);
        // show the window
        ShowWindow(hwndMovie, SW_SHOWNORMAL);
        // play the clip
        if(MCIWndPlay(hwndMovie))
            MessageBeep(0);
    }
        break;
...
```

请记住，由 MCIWndCreate()建立的窗口有能力播放几乎任何类型的 MCI 设备——AVI 只是这种设备的一个例子，但是假如希望把这个窗口建立成子窗口，该函数的语法则不允许为那个窗口分配一个 ID。因此，我们必须调用 SetWindowLong ()，利用 GWL_ID 为它分配一个唯一的 ID。这对于其他场合下访问那个窗口是很有利的，正如我们在上一个代码段里看到的那样。

12.7 左键和右键的同时单击

在一些 Windows 应用程序里，用户可以同时按下鼠标左键和右键，从而执行相应的命令。显然，应用程序不可能同时接收两条消息，即使在 Windows 95 这样的多任务环境下也是不可能的。经过仔细的观察，证明了“同时”按下两个鼠标按钮会生成下面这样的消息流：

```
...
WM_RBUTTONDOWN
WM_NCHITTEST
WM_SETCURSOR
WM_MOUSEMOVE
WM_NCHITTEST
WM_SETCURSOR
WM_LBUTTONDOWN
WM_NCHITTEST
WM_SETCURSOR
WM_RBUTTONUP
```

```
WM_CONTEXTMENU
WM_NCHITTEST
WM_SETCURSOR
WM_LBUTTONUP
...
```

可以看出，同时按下两个鼠标键与双击的情况是不一样的。假如同时按下两个鼠标键，两次事件之间没有预先定义好的时间延迟。因此，我们不能在一个事件发生以后再监视第二个事件的发生，从而判断出是否双键同时按下。其中的原因在于，从而 Windows 编程的眼光来看，两种行动完全没有什么关系。

表 12-5 为大家列出了接收到一条鼠标消息时在 wParam 内的所有可能值。通过 WM_xBUTTON 定义，我们就可以知道接收到一条鼠标消息的时候，一个鼠标按钮是否还处于按下状态。举个例子来说，假如一条 WM_LBUTTONDOWN 消息到达了窗口进程，其中的 wParam 内包含了 MK_RBUTTON。这就意味着接收到这条消息的时候鼠标右键正处于按下状态，据此就可以推断出其他按钮也处于相同的状态。下面这个代码段向我们揭示了这一点：

```
...
case WM_LBUTTONDOWN:
{
    // check  if both buttons are pressed in the same time
    if (wParam & MK_RBUTTOn)
    {
        PlayResource (hInstance, " both");
        break;
    }
...
```

在上面那个代码里，假如两个按钮同时处于按下状态，就会播放一个 .WAV（波形）文件。

光标的剪裁

在这个小节里，我们将结束本章对鼠标的讨论。我们准备建立一个应用程序，用它来限制鼠标指针可以拖动的屏幕表面。在本书附带 CD 的 Listing 12.6 里，大家可以找到这个 CLIPCUR 程序的源代码，它的最大特点就是调用了 API 函数 ClipCursor ()。这个函数接受的唯一参数就是一个 RECT 数据结构的地址，这个结构与准备限制光标拖动的那个屏幕矩形是对应的。

```
#include <winuser.h>
BOOL ClipCursor (CONST RECT * lpRect);
```

假如调用成功，这个函数就会返回一个 TRUE 值，否则就返回一个 FALSE 值。RECT 结

构内包含了用屏幕坐标系统表达的信息。图 12-12 显示了 CLIPCUR 程序的运行结果。

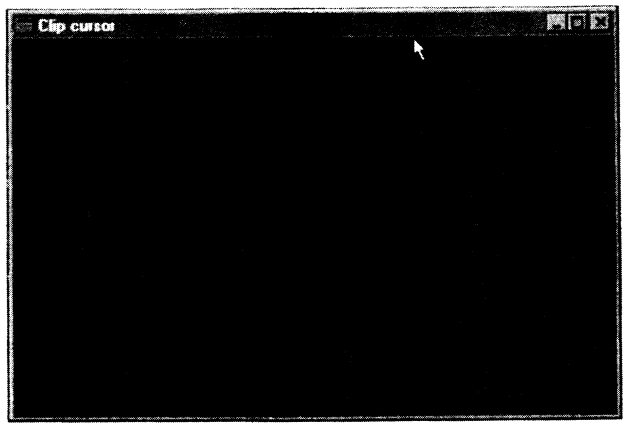


图 12-12　按下鼠标左键以后，CLIPCUR 就把光标的移动范围限制在应用程序的客户区以内

12.8　工具栏

在一个外壳文件夹里，最有用的一个选项也许要算用于显示工具栏的那个了（View Toolbar）。工具栏是 Windows 95 的一种通用控件。这种控件窗口通常放置于菜单栏的下方，其中包含了用于简化 Win32 应用程序使用的一系列工具。某些程序还提供了浮动工具栏，这是一种独立的窗口，它们不与菜单栏依附在一起，可以在主窗口区域内到处移动。

我们可以调用 CreateToolbarEx () 来建立一个工具栏窗口。这是一种宏函数，其中集成了 CreateWindowEx()函数的特性，另外还增加了能满足这种通用控件特定需要的一些特性。从本质上说，工具栏属于子窗口的范畴，它最初的缺省尺寸是 24×22 象素。一个工具栏里看起来似乎是包含了几个按钮控件。但是事实上，工具栏内的按钮只不过是一些小图象而已（缺省情况下是 16×15 象素），这些图象是从一幅单独的位图里提取出来的，这幅位图里则包含了需要用到的所有按钮图象。

每个工具栏按钮通常都是用一幅位图定制的。另外，我们还可以选择在位图下方显示一个正文串。除此以外，我们最后还能为它使用相应的工具提示（Tool Tip）。这种工具提示为每个按钮提供了可视的反馈信息，并为初学者提供了方便。在一个工具栏内，按钮的用途是提供一种可选的并且更直接的方案，以便向应用程序发送一条相应的命令。最常见的一种情况是，工具栏按钮是一些常用菜单项的替身。这些菜单项包括 New，Open 以及 Save 等等。在本书附带 CD 的 Listing 12.7 里，大家可以找到一个名为 TOOLBAR 的示范程序，该程序向我们展示了如何建立和管理一个工具栏、一个状态栏以及一个动画控件。该程序的运行情况请参考图 12-13。

工具栏控件需要用到的某些设置和定制信息可以在建立期间传递给工具栏。其他信息则必须在窗口已经成功地建立好以后，再通过一系列类消息以及数据结构进行指定。

```
#include 〈commctrl.h〉
```

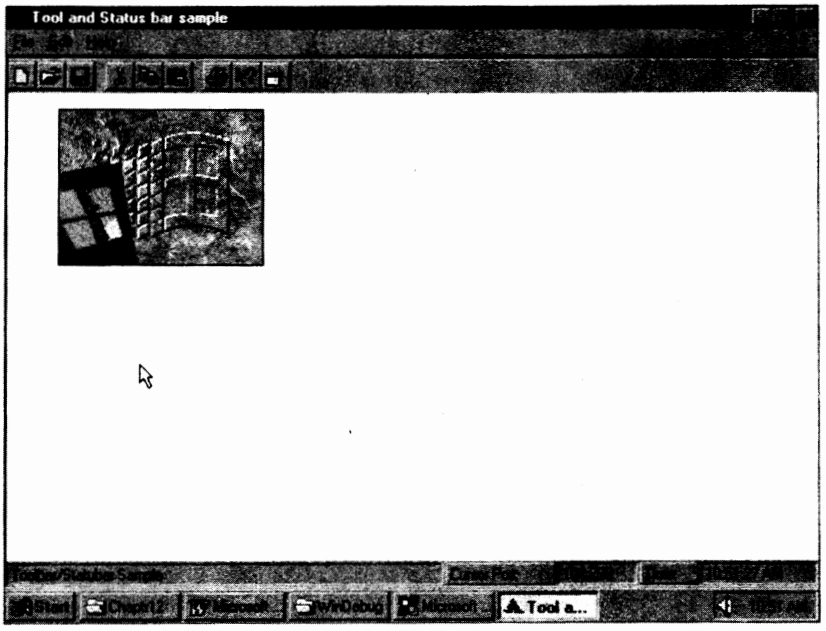


图 12-13　TOOLBAR 示范程序演示了 Win32 应用程序里对工具栏的使用

```
HWND CreateToolbarEx(HWND hwnd,
                     DWORD ws,
                     UINT wID,
                     int nBitmaps,
                     HINSTANCE hBMInst,
                     UINT wBMID,
                     LPCTBBUTTON lpButtons,
                     int iNumButtons,
                     int dxButton,
                     int dybutton,
                     int dxBitmap,
                     int dyBitmap,
                     UINT uStructSize );
```

参数	说明
HWND hwnd	父窗口的句柄，通常是主窗口的句柄
DWORD ws	工具栏风格的组合（参考表 12-7）
UINT wID	工具栏 ID
int nBitmaps	由 hBMInst 资源模块句柄以及 wBMID 资源 ID 指定的位图内的按钮图象数目
HINSTANCE hBMInst	包含了工具栏位图的模块的句柄
UINT wBMID	位图 ID

LPCTBBUTTON lpButtons	TBBUTTON 按钮数组的名称
int iNumButtons	准备添加到工具栏内的按钮数目
int dxButton	按钮水平尺寸（象素）
int dybutton	按钮垂直尺寸（象素）
int dxBitmap	位图水平尺寸（象素）
int dyBitmap	位图垂直尺寸（象素）
UINT uStructSize	一个 TBBUTTON 数据结构的尺寸
返回值	在正文里讨论
HWND	工具栏按钮；假如函数调用失败，则返回一个 NULL 值

所有风格都可以在通过 CreateToolbarEx（）函数建立工具栏的时候事先分配好（TBSTYLE_BUTTON 是缺省情况下的风格）。当控件已经建立好，并且在主窗口内显示出来的时候，假如需要增加一个新的工具栏按钮，那么相同的风格也会向我们提供帮助。在缺省情况下，假如用户在一个工具栏按钮上方松开了鼠标按钮，这个按钮就会立刻返回它最初的状态，这种行为与 Windows 程序的标准按钮是别无二致的。TBSTYLE_CHECK 风格则可以对这种行为进行变动，它能使按钮象现实世界里用到的那种按钮一样。按下以后，就一直保持这种按下状态，直到再次按下它为止。假如我们需要设计一系列相互排斥的按钮，那么就必须为每个这样的按钮分配相应的 TBSTYLE_GROUP 风格，就像下面这样：

```
...
TBBUTTON tbButton[] = {
                    (STD_ FILENEW, MN_ NEW, TBSTATE_ ENABLED,
                      TBSTYLE_BUTTON | TBSTYLE_GROUP,0, 0),
                    (STD_ FILEOPEN, MN_ OPEN, TBSTATE_ ENABLED,
                      TBSTYLE_BUTTON |TBSTYLE_GROUP, 0, 0),
                    (STD_ FILESAVE, MN_ SAVE, TBSTATE_ ENABLED,
                      TBSTYLE_BUTTON |TBSTYLE_GROUP, 0, 0),
            ...
                          };
...
```

利用 TBSTYLE_TOOLTIP，我们为工具栏按钮自动分配一个工具提示控件。工具提示是一种通用的控件，它的任务是在一个矩形小窗口内显示一个简短的正文串。工具提示通常是与其他控件关联在一起，很少单独使用。它可以关联的控件包括标准按钮以及工具栏按钮等等，以便为这些对象提供附加的解释信息。任务栏最右侧的通知区域支持用于每个内部组件的工具提示控件。只需要简单地增加 TBSTYLE_TOOLTIP 风格，我们就可以为每个工具栏按钮提供一条工具提示信息，同时不需要进行任何附加的编码工作（请大家参考本书附带 CD 的 Listing 12.9，其中的 PLAYCD 示范程序为我们提供了关于这方面更详细的信息）。

CreateToolbarEx（）里最关键的参数是 TBBUTTON 结构数组。这个信息块内包含了定

义一个工具栏按钮所需的所有细节信息：

```
typedef struct _TBBUTTON
{
    int iBitmap;
    int idCommand;
    BYTE fsState;
    BYTE fsStyle;
    BYTE bReserved [2 ];
    DWORD dwData;
    int iString;
} TBBUTTON, *PTBBUTTON, * LPTBBUTTON;
typedef const TBBUTTON * LPCTBBUTTON;
```

表 12-6　用于工具栏通用控件内一个按钮的状态定义

工具栏状态	值	说　明
TBSTATE_CHECKED	0x01	按钮处于核选状态
TBSTATE_PRESSED	0x02	按钮被按下
TBSTATE_ENABLED	0x04	按钮处于活动状态
TBSTATE_HIDDEN	0x08	按钮临时性隐藏起来
TBSTATE_INDETERMINATE	0x10	按钮状态不确定（灰色显示）
TBSTATE_WRAP	0x20	按钮后面跟随一个换行符

表 12-7　工具栏通用控件风格

工具栏风格	值	说　明
TBSTYLE_BUTTON	0x00	建立带有标准下推按钮的一个工具
TBSTYLE_SEP	0x01	在两个相邻按钮之间放置一个分隔符
TBSTYLE_CHECK	0x02	建立一个可在按下和未按下两种状态之间切换的按钮
TBSTYLE_GROUP	0x04	建立一个按钮，除非组内另一按钮被按下，否则就一直保持按下状态
TBSTYLE_CHECKGROUP		(TBSTYLE_GROUP | TBSTYLE_CHECK)
		建立一个核选按钮，除非按下组内另一按钮，否则就一直保持按下状态
TBSTYLE_TOOLTIPS	0x0100	为工具栏窗口关联一个工具提示控件
TBSTYLE_WRAPABLE	0x0200	工具栏控件可以显示多行按钮
TBSTYLE_ALTDRAG	0x0400	允许用户在按下 Alt 键的前提下在控件内拖动一个工具栏按钮

工具栏可以根据通过 CreateToolbarEx () 函数传递的位图来建立一个内部图象列表，同时为每幅图象分配一个连续的 ID (从零开始)。TBUTTON 的 Bitmap 项与工具栏内部图象列表里的图象索引是对应的。相同的逻辑亦可应用于 String 项。一个工具栏控件在内部存放了一系列正文串，这些正文串与每个按钮上显示的正文是对应的。与一个按钮关联起来的串值将分配给 iString，这些值是以索引过的正文串数组（从 0 开始索引）为基础的。正如我们以前提到的那样，一个工具栏内的按钮代表了菜单项选择的另外一种可选方案。因此，每个按

钮都有个对应的 ID，这个 ID 与等效的菜单命令是对应的。idCommand 项内就存储了这种类型的数据。

fsState 内的按钮状态是由一个或者多个 TBSTATE_值表示的，对这些 TBSTATE_状态值的总结请大家参考表 12-16。fsStyle 内的按钮风格则可以是表 12-7 那些定义的任何一种组合。

dwData 项是一个四字节的区域，其中可以存储由应用程序定义的任何数据。CreateToolbarEx () 函数里的其他参数就毋需我们进行更进一步的解释了，从它们的名字就可以推断出相应的功能。这些参数的作用是指定准备建立的按钮数、按钮的大小以及包含的图象等等。在 TOOLBAR 示范程序里，我们定义的一个数组里包含了 18 个按钮定义，其中代表的菜单项包括 File 和 Edit 等等，从各自的 ID 便可分别看出对应的菜单项。所有这些按钮都处于活动状态，我们用分隔符（TBSTYLE_SEP）把这些按钮分隔成了三个组别。

```
...
static TBBUTTON tbButton[40] = {
  STD_FILENEW,      MN_NEW,       TBSTATE_ENABLED, TBSTYLE_BUTTON,
                                  0, 0},
  {STD_FILEOPEN,    MN_OPEN,      TBSTATE_ENABLED, TBSTYLE_BUTTON,
                                  0, 0},
  {STD_FILESAVE,    MN_SAVE,      TBSTATE_ENABLED, TBSTYLE_BUTTON,
                                  0, 0},
  {0,               0,            TBSTATE_ENABLED, TBSTYLE_SEP,
                                  0, 0},
  {STD_CUT,         MN_CUT,       TBSTATE_ENABLED, TBSTYLE_BUTTON,
                                  0, 0},
  {STD_COPY,        MN_COPY,      TBSTATE_ENABLED, TBSTYLE_BUTTON,
                                  0, 0},
  {STD_PASTE,       MN_PASTE,     TBSTATE_ENABLED, TBSTYLE_BUTTON,
                                  0, 0},
  {0,               0,            TBSTATE_ENABLED, TBSTYLE_SEP,
                                  0, 0},
  {STD_PRINT,       MN_PRINT,     TBSTATE_ENABLED, TBSTYLE_BUTTON,
                                  0, 0},
  {STD_HELP,        MN_ABOUT,     TBSTATE_ENABLED, TBSTYLE_BUTTON,
                                  0, 0},
  {VIEW_
  NEWFOLDER,        MN_
                    NEWFOLDER,    TBSTATE_ENABLED, TBSTYLE_BUTTON,
                                  0, 0},
  { STD_DELETE,     MN_DELETE,    TBSTATE_ENABLED, TBSTYLE_BUTTON,
                                  0, 0},
  { STD_PRINTPRE,   MN_           TBSTATE_ENABLED, TBSTYLE_BUTTON,
                    PRINTPREVIEW, 0, 0},
```

```
    { STD_FIND,         MN_FIND,        TBSTATE_ENABLED, TBSTYLE_BUTTON,
                                        0, 0},
    { STD_REPLACE,      MN_REPLACE,     TBSTATE_ENABLED, TBSTYLE_BUTTON,
                                        0, 0},
    { STD_PROPERTIES,   MN_PROPERTIES,  TBSTATE_ENABLED, TBSTYLE_BUTTON,
                                        0, 0},
    { STD_UNDO,         MN_UNDO,        TBSTATE_ENABLED, TBSTYLE_BUTTON,
                                        0, 0},
    { STD_REDOW,        MN_REDO,        TBSTATE_ENABLED, TBSTYLE_BUTTON,
                                        0, 0},
};
...
```

这个 TBBUTTON 数据结构代表了所有按钮的集合，这些按钮都有可能在应用程序的工具栏内显示出来。建立好工具栏以后，这个数组就会作为第七个参数传递过去，但是最初只插入前面 11 个按钮，剩下的 7 个按钮则用于以后用户自己进行定制。

```
...
// create the toolbar
hwndToolbar = CreateToolbarEx(hwnd,
                              WS_CHILD | WS_VISIBLE | CCS_
ADJUSTABLE |TBSTYLE_TOOLTIPS,
                              CT_TOOLBAR,
                              15,
                              HINST_COMMCTRL,
                              IDB_STD_SMALL_COLOR,
                              tbButton,
                              iInsertBtn,
                              0, 0, 0, 0,
                              sizeof(TBBUTTON));
...
```

iInsertBtn 标识符在整个窗口进程里都是“可见”的，而且它会根据工具栏内显示的按钮总数不断地进行更新：

```
...
static int iInsertBtn = 11;
...
```

另外一个整数标识符是 iTotBtns，它在工具栏的整体结构中扮演着重要的角色。最初的时候，这个标识符指定了 TBBUTTON 数据结构里的按钮定义总数，如下所示：

```
...
```

```
static int iTotBtns = 18;
...
```

TBBUTTON 数组可以容纳 40 多个按钮定义，这样就保留了一些空间，以便在应用程序运行期间随时增加新的按钮定义。假如某个程序需要对新工具栏对象的定义进行增添，那么这种设计思路就显得相当有用了。

最后一个参数前面的四个零值用于指示工具栏窗口为按钮和它们的图象分配缺省值。上面两个代码段并不代表建立一个工具栏时最常见的方式。事实上，用于这个控件的图象是直接向系统本身取得的，而不是从应用程序里取得的。IDB_STD_SMALL_COLOR 定义可以指示工具栏类在预定义图象的基础上建立一个内部图象列表。这些预定义图象是包含于 COMMCTRL.DLL 文件内的。通过用于替换模块句柄的 HINST_COMMCTRL 定义，我们可以澄清这种概念，如下所示：

```
#define HINST_COMMCTRL ( (HINSTANCE) -1)
```

COMMCTRL.H 头文件内包含了下面四个定义，它们指定了表 12-8 内列出来的预定义图标。

```
#define IDB_STD_SMALL_COLOR     0
#define IDB_STD_SMALL_COLOR     1
#define IDB_STD_SMALL_COLOR     4
#define IDB_STD_SMALL_COLOR     5
```

考虑到存在着大量的预定义图标，所以利用这些标准控件显然才是最明智的。这种选择也反映在 TBBUTTON 数据结构数组上面。每个图标定义都与表 12-8 内列出某个值对应。

不走运的是，CreateToolbarEx () 的语法允许我们使用一个单一的图象资源，无论这个资源是由系统定义的，还是定制生成的。这就意味着假如我们想显示出属于不同位图资源的工具栏图标，就必须把前面那段代码集成到一起。现在，我们可以来看看系统文件夹的情况。如图 12-14 所示，工具栏提供了从 IDB_STD_SAMLL_COLOR 里提取出来的对象以及相应的位图引用。

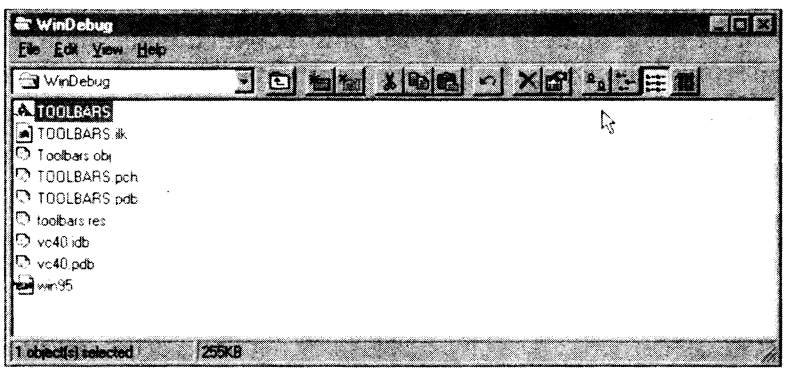


图 12-14　系统文件夹内的工具栏提供了来自 IDB_STD_SMALL_COLOR 和 IDB_VIEW_SMALL_COLOR 预定义资源的一些图象

表 12-8 工具栏控件内的预定义图标

图标定义	值	说 明
STD_CUT	0	Cut（剪切）按钮
STD_COPY	1	Copy（拷贝）按钮
STD_PASTE	2	Paste（粘贴）按钮
STD_UNDO	3	Undo（撤消）按钮
STD_REDOW	4	Redo（重做）按钮
STD_DELETE	5	Delete（删除）按钮
STD_FILENEW	6	Create new file（建立新文件）按钮
STD_FILEOPEN	7	Open an existing file（打开一个现成文件）按钮
STD_FILESAVE	8	Save the current file（存储当前文件）按钮
STD_PRINTPRE	9	Print preview（打印预览）按钮
STD_PROPERTIES	10	Properties（属性）按钮
STD_HELP	11	Help（帮助）按钮
STD_FIND	12	Find（查找）按钮
STD_REPLACE	13	Replace（替换）按钮
STD_PRINT	14	Print（打印）按钮
VIEW_LARGEICONS	0	Large icons（大图标）视窗
VIEW_SMALLICONS	1	Small icons（小图标）视窗
VIEW_LIST	2	List（列表）视窗
VIEW_DETAILS	3	Details（详细列表）视窗
VIEW_SORTNAME	4	Arrange by name（按名称排列）图标
VIEW_SORTSIZE	5	Arrange by size（按大小排列）图标
VIEW_SORTDATE	6	Arrange by data（按数据排列）图标
VIEW_SORTTYPE	7	Arrange by type（按类型排列）图标
VIEW_PARENTFOLODER	8	Go to parent folder（返回上一级文件夹）图标
VIEW_NETCONNECT	9	Network connection（网络连接）图标
VIEW_NETDISCONNECT	10	Network disconnect（撤消网络连接）图标
VIEW_NEWFOLDER	11	Create new folder（建立新文件夹）图标

工具栏窗口将按照连贯的顺序对图象进行存储，同时在自己的图象列表里为它们分配连续的 ID。因此，在前一个代码段里，以前缀 STD_起头的十五幅图象占据了 0—14 号位置。假如我们希望对这些图象的数目进行扩展，就必须用到 TB_ADDBITMAP 消息。我们在这里的想法是用 VIEW_前缀把另外 12 幅图象存放到工具栏图象列表里，同时为它们分配 15 到 26 的索引编号。在这儿，TB_ADDBITMAP 消息的作用就是对这种把新图象集合增加到已经与一个工具栏窗口关联起来的某个现成图象集合内进行管理，其中的 wParam 用以传递位图集合内的图象数目，而 lParam 则用以传递一个 TBADDBITMAP 数据结构，如下所示：

```
...
TBADDBITMAP tab;
...
// add more bitmaps
tab.hInst = HINST_COMMCTRL;
tab.nID = IDB_VIEW_SMALL_COLOR;
```

```
// add the bitmaps
iBmp = SendMessage(hwndToolbar, TB_ADDBITMAP, 12, (LPARAM)&tab);
if(! SendMessage(hwndToolbar, TB_CHANGEBITMAP, MN_NEWFOLDER,
MAKELPARAM(iBmp +11, 0)))
    MessageBeep(0);
...
```

另外一条工具栏消息(请参考表 12-9 对这些消息的完整总结)是 TB_CHANGEBITMAP，利用它可以对一个工具栏按钮内的位图引用进行修改。wParam 内存放的是菜单项 ID，而 lParam 的低字内则包含了工具栏位图的内位图索引编号。利用这种操作，应用程序就可以用工具栏图象列表内的最后一幅位图(#26 图象)替换掉最右边那个按钮的当前位图索引。

表 12-9 工具栏消息

工具栏消息	值	说 明
TB_ENABLEBUTTON	(WM_USER + 1)	激活或者屏蔽一个工具栏按钮
TB_CHECKBUTTON	(WM_USER + 2)	核选或者非核选一个工具栏按钮
TB_PRESSBUTTON	(WM_USER + 3)	按下或者松开一个工具栏按钮
TB_HIDEBUTTON	(WM_USER + 4)	隐藏或者显示一个按钮
TB_INDETERMINATE	(WM_USER + 5)	设置或者清除一个工具栏按钮的未确定状态
TB_ISBUTTONENABLED	(WM_USER + 9)	检查一个按钮是否处于活动状态
TB_ISBUTTONCHECKED	(WM_USER + 10)	检查一个按钮是否处于核选状态
TB_ISBUTTONPRESSED	(WM_USER + 11)	检查一个按钮是否处于按下状态
TB_ISBUTTONHIDDEN	(WM_USER + 12)	检查一个按钮是否处于隐藏状态
TB_ISBUTTONINDETERMIANTE	(WM_USER + 13)	检查一个按钮是否处于未确定状态
TB_SETSTATE	(WM_USER + 17)	为按钮分配一个或者多个 TBSTATE_标志
TB_GETSTATE	(WM_USER + 18)	取得一个按钮的状态
TB_ADDBITMAP	(WM_USER + 19)	在工具栏控件内增加一幅位图
TB_ADDBUTTONS	(WM_USER + 20)	在工具栏控件内增加按钮
TB_INSERTBUTTON	(WM_USER + 21)	在工具栏控件内插入一个按钮
TB_DELETEBUTTON	(WM_USER + 22)	从工具栏控件内删除一个按钮
TB_GETBUTTON	(WM_USER + 23)	取得关于一个工具栏按钮的信息
TB_BUTTONCOUNT	(WM_USER + 24)	返回工具栏内按钮的数目
TB_COMMANDTOINDEX	(WM_USER + 25)	从一个命令 ID 开始返回按钮索引
TB_SAVERESTOREA	(WM_USER + 26)	存储或者恢复一个工具栏的状态
TB_SAVERESTOREW	(WM_USER + 76)	存储或者恢复一个工具栏的状态(UNICODE)
TB_CUSTOMIZE	(WM_USER + 27)	显示 Customize Toolbar(定义工具栏)对话框
TB_ADDSTRINGA	(WM_USER + 28)	把一个正文串与某个按钮关联起来
TB_ADDSGRINGW	(WM_USER + 77)	把一个正文串与某个按钮关联起来(UNICODE)
TB_GETITEMRECT	(WM_USER + 29)	返回工具栏内某个按钮的限制矩形
TB_BUTTONSTRUCTSIZE	(WM_USER + 30)	用 CreateWindowEx()函数建立一个工具栏时指定 TBBUTTON 数据结构的长度
TB_SETBUTTONSIZE	(WM_USER + 31)	设置一个工具栏按钮的大小
TB_SETBITMAPSIZE	(WM_USER + 32)	设置一个工具栏按钮内的位图的大小
TB_AUTOSIZE	(WM_USER + 33)	指示工具栏重新定义自己的尺寸
TB_GETTOOLTIPS	(WM_USER + 35)	返回工具提示控件的句柄

（续）

工具栏消息	值	说　明
TB_SETTOOLTIPS	(WM_USER + 36)	把一个工具提示控件与一个工具栏控件关联到一起
TB_SETPARENT	(WM_USER + 37)	设置工具栏的父窗口
TB_SETROWS	(WM_USER + 39)	定义工具栏内的按钮行数
TB_GETROWS	(WM_USER + 40)	返回工具栏内的按钮行数
TB_SETCMDID	(WM_USER + 42)	为一个按钮分配一个命令 ID
TB_CHANGEBITMAP	(WM_USER + 43)	改变一个工具栏按钮内的位图引用
TB_GETBITMAP	(WM_USER + 44)	返回与一个工具栏按钮关联在一起的图象索引
TB_GETBUTTONTEXTA	(WM_USER + 45)	返回按钮正文
TB_GETBUTTONTEXTW	(WM_USER + 75)	返回按钮正文(UNICODE)
TB_REPLACEBITMAP	(WM_USER + 46)	替换与一个工具栏窗口关联在一起的位图

12.8.1　工具栏的定制

CCS_ADJUSTABLE 定制控件风格可以让我们定义一个可定制的工具栏。什么是“可定制的工具栏”呢？它在这儿的含义是假如用鼠标左键双击工具栏本身（而不是其中的一个按钮），屏幕上就会自动显示出一个定制工具栏对话框。图 12-15 向我们展示了 TOOLBAR 示范程序里的 Customize Toolbar（定制工具栏）对话框的样子。

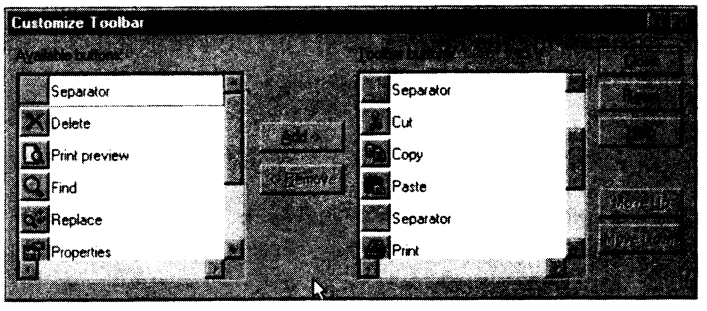


图 12-15

我们可以在应用程序源代码内强制性地显示出这个对话框，方法是向工具栏控件直接发送一条 TB_CUSTOMIZE 消息。无论我们怎样让 Customize Toolbar 对话框显示出来，它的左侧列表框内都包含了可以增加到工具栏内的按钮，右侧的另外一个列表框则反映了当前的工具栏按钮布局。显然，这两个列表框的内容必须根据工具栏内当前的情况进行定制。在显示出这个对话框之前，工具栏控件会向自己的父窗口发送一条通知代码 TBN_GETBUTTONINFO，这个代码是通过 WM_NOTIFY 消息传递的。WM_NOTIFY 消息的 lParam 里包含了一个 TBNOTIFY 数据结构的地址：

```
typedef struct tagTBNOTIFY
{
    NMHDR hdr;
    int iItem;
```

```
    TBBUTTON tbButton;
    int cchText;
    LPWSTR pszText;
} TBNOTIFY, *LPTBNOTIFY;
```

其中，第一个参数是一个传递的 NMHDR，它在所有通用控件里都是通用的。iItem 项则指定了第一个工具栏按钮的索引编号。在这个问题上面，开发环境提供的联机文档资料极易产生误导作用（它事实上没有提供任何帮助）。TBNOTIFY 数据结构里的 TBBUTTON 项是完全置空的，其中没有包含有价值的信息。因此，我们不能通过工具栏控件把数据传回应用程序，而必须由应用程序提供相应的数据，把这些数据填写到 Customize Toolbar 对话框的两个列表框里。在 TBBUTTON 数据结构里，我们已经存放了 11 个定义——这 11 个定义与工具栏内九个实际的按钮对应（加上两个分隔符），另外还存放了 7 个附加的定义。这就意味着索引 0 到 11 是由已经使用的按钮占据的，而其他按钮则占据了从 12 到 17 的索引编号。

除非应用程序返回一个 TRUE 值，否则工具栏控件就会不断地发送出 TBN_GETBUTTONINFO 通知代码。存储于 iItem 项内的值是一个不断增大的计数器，它的值会从 0 开始不停地更新。因此，我们需要采取的第一个步骤是判断 pTBNotify -> iItem 是否大于 iTotBtns，后者代表了工具栏按钮定义的总数。假如这个条件满足，应用程序就会返回，同时中断 TBN_GETBUTTONINFO 通知代码的发出。

```
...
case TBN_GETBUTTONINFO:
{
    LPTBNOTIFY pTBNotify = (LPTBNOTIFY)lParam;
    char szItemText[20];
    static int iCnt;

    // skip when reached the end
    if(pTBNotify -> iItem >= iTotBtns)
    {
        // reset the counter
        iCnt = 0;
        break;
    }
...
```

不幸的是，我们在这儿的工作并没有完。在与 TBN_GETBUTTONINFO 通知代码相关的代码部分里，我们必须用恰当的工具栏按钮信息填写 TBNOTIFY 里的 tbButton 项。这样就导致一个问题：到底填写什么信息，并且以什么顺序填写呢？经过一番测试以后，我认识到自己首先必须提供与还没有插入工具栏内的按钮有关的 TBBUTTON 信息，然后提供与现

成按钮有关的信息。

为了实现对可用按钮定义的控制，我们需要把它们从 tbButton 数组拷贝到 TBNOTIFY 里等效的项内。拷贝的起始位置是从 iInsertBtn 开始，每拷贝一次位置就增加 pTBNotyfi —〉 iItem。相同的方案也可以应用于现成的工具栏按钮，唯一的区别是由于分隔符的存在而造成的。这些工具栏按钮在 Customize Toolbar 对话框里是根据一种特殊的方式来对待的。Available buttons 以及 Toolbar buttons 列表框里显示的工具栏按钮正文是由对话框自己自动生成的。因此，我们没有必要把这种信息再传递回工具栏控件。每当遇到一个分隔符，一个静态计数器 iCnt 就会自动增值。这个值随后再从当前的按钮值内扣掉，这样就能从资源文件里取得正确的字符串说明：

```
...
// retrieve information for the additional buttons
if(pTBNotify -〉 iItem 〈 (iTotBtns - iInsertBtn))
{
    pTBNotify -〉 tbButton = tbButton[iInsertBtn + pTBNotify -〉
        iItem];
}
else
{
    pTBNotify -〉 tbButton = tbButton[pTBNotify -〉 iItem - (iTotBtns -
        iInsertBtn)];

    if(pTBNotify -〉 tbButton.idCommand == 0)
    {
        iCnt++;
        return TRUE;
    }
}
// load the button strings
pTBNotify -〉 cchText = LoadString(hInstance,
                                  MN_DESCRIPTION + pTBNotify -〉 iItem -
                                  iCnt,szItemText, sizeof(szItemText));
lstrcpy(pTBNotify -〉 pszText, szItemText);
return TRUE;
}
break;
...
```

我们在 TOOLBAR 示范程序里采用的方案需要进行一些扩展，这样才有可能成为其他应

用程序能够重新利用的“再生资源”。在这种控件类使用的所有消息、数据结构以及通知代码里，我们缺少的是一种标准的粘着机制。利用这种机制可以把工具栏按钮信息与必须提供的附加数据粘合起来。例如，我情愿使用工具提示正文，或者应该在 Customize Toolbar 对话框里显示出来的正文。只有在不允许自己的用户对工具栏进行修改的前提下，我们才能把这种信息存放在资源文件的 STRINGTABLE 资源里。只要增加了新按钮，甚至只是改变了一下现成按钮的排列顺序，与应用程序内静态存储的信息建立的链接就会遭到破坏。与每个工具栏按钮关联起来的四字节存储区域也许可以用作对资源文件里正文串的一种引用。但是，假如允许用户在程序运行期间动态地建立新按钮（也许是在应用程序里建立了一个内部宏以后进行），那么这种方案仍然存在弊端。最终，真正能解决这种问题的方案是先建立一个定制内存块，并且在其中填写用于对每个按钮进行标识的信息，然后在 TBBUTTON 数据结构的 dwData 项里存储对这个内存块的一个引用。表 12-10 为大家列出了工具栏通用控件使用的所有通知代码。

表 12-10　工具栏控件用于和父窗口通信的通知代码

工具栏通知代码	值	说　明
TBN_GETBUTTONINFOA	(TBN_FIRST-0)	用与某个工具栏按钮相关的信息填写一个 TBNOTIFY 数据结构
TBN_GETBUTTONINFOW	(TBN_FIRST-20)	用与某个工具栏按钮相关的信息填写一个 TBNOTIFY 数据结构（UNICODE）
TBN_BEGINDRAG	(TBN_FIRST-1)	用于通知用户已经开始了一个按钮在工具栏内的拖动
TBN_ENDDRAG	(TBN_FIRST-2)	用于通知按钮在工具栏内的拖动已经结束了
TBN_BEGINADJUST	(TBN_FIRST-3)	用于通知父窗口用户已经启动了工具栏控件的定制操作
TBN_ENDADJUST	(TBN_FIRST-4)	通知父窗口用户已经结束了工具栏控件的定制操作
TBN_RESET	(TBN_FIRST-5)	指出用户已经重新设置了 Customize Toolbar（定制工具栏）对话框的内容
TBN_QUERYINSERT	(TBN_FIRST-6)	指出在定制一个工具栏的时候，一个按钮是否能够插入另一指定按钮的左侧
TBN_QUERYDELETE	(TBN_FIRST-7)	指出在定制一个工具栏的时候，是否可以删除一个按钮
TBN_TOOLBARCHANGE	(TBN_FIRST-8)	通知父窗口用户已经完成了对工具栏的定制
TBN_CUSTHELP	(TBN_FIRST-9)	用户已经选中了 Customize Toolbar 对话框中的 Help 按钮

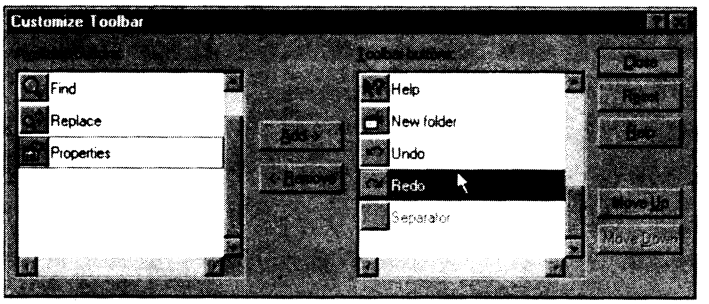


图 12-16　Redo 和 Undo 按钮已经从 Available buttons（可用按钮）对话框移到了另外一个对话框

尽管存在这些方面的考虑，TOOLBAR 示范程序的工具栏仍然为我们提供了完整的功能。我们可以利用 Customize Toolbar 对话框在工具栏进而增加新按钮（如图 12-16 所示），作出的改动能够在工具栏内马上反映出来（如图 12-17 所示）。

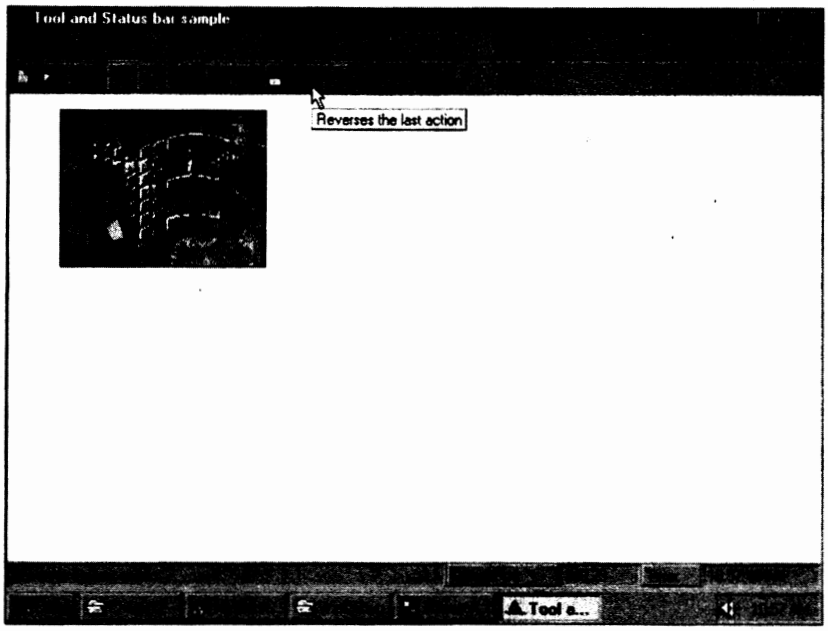


图 12-17　新的 Redo 和 Undo 按钮在工具栏内显示出来了

TOOLBAR 的最后一项特性是它可以在程序运行期间建立一个新按钮，这种特性是严格限制于工具栏通用控件使用的。假如按下最左侧的那个按钮，一个新按钮就会在最右侧显示出来，其中带有一个红色的 A，如图 12-18 所示。

为了实现这种功能，我们编写的程序代码与以前建立工具栏的时候用到的代码是比较相似的。新的按钮分配了名为 MN_EXIT 的一个 ID，从而强迫应用程序中断。在这种情况情况下，代码的区别在于我们使用了一幅资源位图，它的 ID 是 TEST，而不再是以前的那种系统资源。由此以来，工具栏位图总共需要对三种不同的资源进行访问：两幅系统位图以及一个定制位置。

```
...
case MN_NEW:
{
    TBBUTTON tb;
    TBADDBITMAP tbab;
    int iNewBmp;
    short iPos = 0;

    tbab.hInst = hInstance;
```

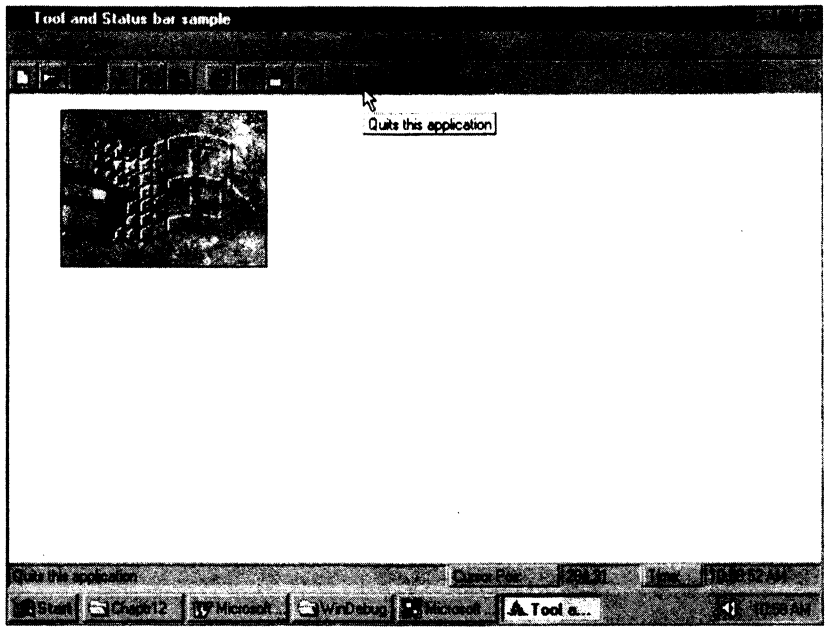


图 12-18　一个新的定制按钮已经添加到工具栏内了

```
        tbab.nID = TEST;
        // add the bitmap to the Toolbar
        iNewBmp = SendMessage(hwndToolbar, TB_ADDBITMAP, (WPARAM)
1, (LPARAM)&tbab);

        tb.iBitmap = iNewBmp;
        tb.idCommand = MN_EXIT;
        tb.fsState = TBSTATE_ENABLED;
        tb.fsStyle = TBSTYLE_BUTTON;
        tb.dwData = 0;
        tb.iString = 0;

        // add the button
        SendMessage(hwndToolbar, TB_ADDBUTTONS, (WPARAM)1,
            (LPARAM)&tb);
    }
        break;
...
```

增加的项目会在 Customize Toolbar 对话框里自动显示出来，如图 12-19 所示。

12.8.2　工具提示

假如让鼠标指针在某个工具栏按钮上方停留片刻，就会显示出一条工具提示信息，这就

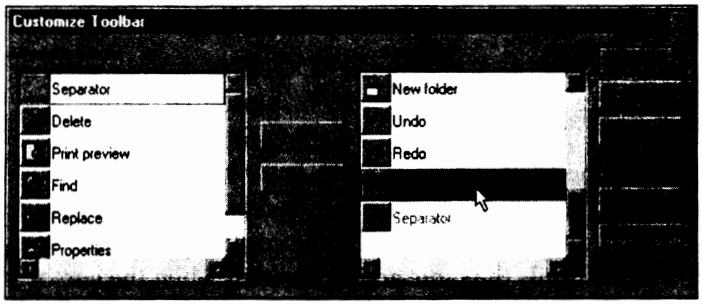


图 12-19 Customize Toolbar 对话框现在包含了与运行期间增加的定制按钮相关的图标

是 TBSTYLE_TOOLTIPS 风格标志的功劳。工具提示控件可以把每条工具提示正文串存放于自己的缓冲区内，或者从上级应用程序发出对正文串的请求。对于和一个工具栏关联在一起的工具提示来说，其中实现的正是第二种行为。因此，假如显示条件符合，父窗口就会接收到一条 WM_NOTIFY 消息，其中带有相应的 TTN_NEEDTEXT 通知代码。再一次提醒大家，工具提示正文是从 STRINGTABLE（串表）资源内取得的，然后会传递给相应的控件，从而把自己显示出来，如下所示：

```
...
// display the tooltip
if(pToolTipText -> hdr.code == TTN_NEEDTEXT)
{
    // load the tooltip text
    LoadString(hInstance, pToolTipText -> hdr.idFrom,
               szToolTipText, sizeof(szToolTipText));

    // update the status bar
    UpdateStatusBar(hwndStatusbar, pToolTipText -> lpszText, 0, 0);

    // save the information
    pToolTipText -> hinst = hInstance;
    pToolTipText -> lpszText = MAKEINTRESOURCE(pToolTipText ->
        hdr.idFrom);
}
...
```

12.8.3 状态栏

我们在 TOOLBAR 程序里还能找到另外一种通用控件，那就是不很引人注目的一个状态

栏。实际上，我在这本书的第一个示范程序里就已实现了状态栏，尽管当时并没有对它进行深入的探讨。大家可以参考本书附带 CD 的 Listing 1.1。状态是沿着顶级窗口（通常是一个叠置式窗口）底部边界摆放的一个窗口。它的主要用途是显示一个简短的正文串，这个正文串概略解释了当前高亮度显示菜单项能够完成的任务是什么。在 TOOLBAR 示范程序里，假如把鼠标指针放置于一个工具栏按钮上方，同样也可以看到状态栏的显示情况。构成这种机制的基本原理是完全一样的，因为这些工具栏按钮只不过是某些常用菜单项的图形化表达而已。我们可以调用 API 函数 CreateStatusWindow()，同时用它传递一系列标志、标题父窗口的句柄以及一个 ID，这样就可以完成状态栏的创建，如下所示：

```
...
// create the status bar
hwndStatusbar = CreateStatusWindow(WS_CHILD | WS_VISIBLE |
    WS_BORDER,
                        "Toolbar/Statusbar Sample",
                        hwnd,
                        CT_STATUSBAR);
...
```

无论工具栏还是状态栏，一项有趣的特性是我们不必自己计算它们的大小和位置，这些工作是由它们自动完成的。我们只需要把定址于父窗口的 WM_SIZE 消息传递给每个控件就足够了，就像下面这样：

```
...
case WM_SIZE:
{
    // resize the status bar
    SendMessage(hwndStatusbar, msg, wParam, lParam);
    // resize the toolbar
    SendMessage(hwndToolbar, msg, wParam, lParam);

    // re-position the panes in the status bar
    InitializeStatusBar(hwndStatusbar, hwnd);
}
...
```

稍微复杂一些的处理在于对状态栏的定制过程中。我们通常把控件分割成几个从属部分，以便对不同的信息进行处理。这种处理是通过一系列特定的消息来操纵的，其中用到的消息都是属于状态栏这一类专用的，如表 12-11 所示。SB_SETPARTS 消息正是我们在这儿需要用到的。在这条消息的 wParam 里，我们需要指定从属部分的总数，而 lParam 则指定了一个

整数数组的地址，其中包含了每一部分的边界信息：

```
...
SendMessage(hwndStatusbar, SB_SETPARTS, 5, (LPARAM)ptArray);
...
```

表 12-11　状态栏通用控件使用的消息

状态栏消息	值	说　明
SB_SETTEXT	(WM_USER+1)	设置一个状态栏部分的正文
SB_GETTEXT	(WM_USER+2)	从一个状态栏部分里取得正文
SB_GETTEXTLENGTH	(WM_USER+3)	返回一个状态栏部分的正文长度
SB_SETPARTS	(WM_USER+4)	设置一个状态栏的部分
SB_GETPARTS	(WM_USER+6)	返回一个状态栏内所有从属部分的数量，以及它们右侧边界的位置
SB_GETBORDERS	(WM_USER+7)	返回一个状态栏窗口水平和垂直边界的当前宽度
SB_SETMINHEIGHT	(WM_USER+8)	设置状态栏绘图区域的最小高度
SB_SIMPLE	(WM_USER+9)	切换至简单模式，以便实现多重输出(一个单一的部分或者几个部分)
SB_GETRECT	(WM_USER+10)	计算状态栏窗口内某个部分的限制矩形

状态栏支持的唯一风格就是 SBARS_SIZEGRIP，它可以在控件的右边角处建立一个网格区域，这样就对显示表面进行了延展，用户可以利用鼠标对整个窗口进行拉伸。这种通用控件支持物主绘图特性，然而这种支持是通过一种不寻常的途径实现的。通常情况下，当我们建立一个物主绘图控件的时候，整个控件都需要依赖于父窗口，否则无法完成自己的输出任务。然而相反，在状态栏里，我们可以通过 SBT_OWNERDRAW 标志分配这种物主绘图属性，这个标志专门用于整个窗口内的一个单独部分。通过 SB_SETTEXT 消息设置一个状态栏部分使用的正文时，我们可以像下面这样增加相应的 SB_OWNERDRAW 标志：

```
...
SendMessage(hwndStatusbar, SB_SETTEXT, 2 | SBT_OWNERDRAW,
    &pData);
...
```

wParam 内同时包含了部分编号（2）以及一个 SBT_标志（请参考表 12-12），而一旦设置了 SBT_OWNERDRAW 以后，lParam 就变成了一个常规的 32 位数值。父窗口会通过一条 WM_DRAWITEM 消息接收到这个定制数据块，它包含于一个 DRAWITEMSTRUCT 数据结构的 itemData 项内。

表 12-12　SB_SETTEXT 状态栏消息使用的标志

SB_SETTEXT 标志	值	说　明
SBT_OWNERDRAW	0x1000	定义一个物主绘图部分
SBT_NOBORDERS	0x0100	正文没有边框
SBT_POPOUT	0x0200	正文显示于高度比其他部分大的某个部分里
SBT_RTLREADING	0x0400	用从右到左的读取顺序显示正文（希伯来或者阿拉伯语言系统支持这种特性）

12.9 动画控件

在 TOOLBAR 程序里，我们还能找到另外一种有趣的控件，那就是一个动画控件。这种类型的窗口可以显示某些无声的 AVI 文件，这是一种很短的影象剪辑，只是没有附带音轨。Windows 95 在许多场合下都用到了这种通用控件。拷贝一个大型文件，或者把许多对象从一个地方移至另一地方的时候，或者甚至在删除一些文件的时候，屏幕上显示的动画就是一段无声 AVI 影象，它们就是一个动画控件里播放的。Animate_Create () 宏函数可以返回这种新窗口的句柄，这个新窗口最初是看不见的，并且没有与特定的 AVI 文件关联起来。表 12-13 为大家列出了这一类控件支持的所有风格。

表 12-13 动画控件风格

动画风格	值	说 明
ACS_CENTER	0x0001	使影象剪辑在动画控件中央播放
ACS_TRANSPARENT	0x0002	用一个透明背景描绘动画，不用 AVI 剪辑内指定好的背景颜色
ACS_AUTOPLAY	0x0004	只要打开了 AVI 剪辑，就开始自动播放其中的动画

Animate_Open () 函数的作用是载入一段无声 AVI 影象，然后把它分配给动画控件。在这以后，我们随时可以通过调用 Animate_Play () 宏函数把它播放出来：

```
#include <commctrl.h>
BOOL Animate_Play(HWND hwndAnim, short wFrom, short wTo,
    UINT cRepeat);
```

在这个宏函数下面隐藏了 ACM_PLAY 消息，这条消息的 wParam 内传递了 AVI 文件的重复播放次数（假如为-1，则表示播放次数不定），而 lParam 内则传递了准备播放的第一帧（wFrom）和最后一帧（wTo）动画。

```
HWND CreateAnimationCtrl(HWND hwnd, HINSTANCE hInstance)
{
    HWND hwndAnim;

    // create the animation control
    hwndAnim = Animate_Create(hwnd, DL_ANIMATE, WS_BORDER |
        WS_CHILD, hInstance);

    // position the animation control
    SetWindowPos(hwndAnim, HWND_TOP, 40, 40, 550, 750,
        SWP_NOZORDER |SWP_DRAWFRAME);
    ShowWindow(hwndAnim, SW_SHOWNORMAL);
```

```
    // open the AVI clip, and show the animation control
    if(! Animate_Open(hwndAnim, "MyComputer2.AVI"))
            MessageBeep(0);
        Animate_Play(hwndAnim, 0, -1, -1);

    return hwndAnim;
}
```

表 12-14 为大家总结了用于动画通用控件的消息。

表 12-14 动画控件消息

动画控件消息	值	说 明
ACM_OPENA	(WM_USER+100)	打开和载入一个无声 AVI 文件
ACM_OPENW	(WM_USER+103)	打开和载入一个无声 AVI 文件 (UNICODE)
ACM_PLAY	(WM_USER+101)	播放一个 AVI 文件
ACM_STOP	(WM_USER+102)	停止播放 AVI 文件

12.10 侦查其他窗口

假如你是一名 Windows 开发人员，那么时常对其他窗口进行侦查就是一种有趣和有用的习惯。无论 SPY 还是 SPY++，这两个工具软件都能提供关于某个系统窗口的一些有用的信息，比如外壳桌面或者任务栏的消息等等。在常规情况下，我们可以对现成的任何一个窗口进行侦查，从中获取一些有用的信息，甚至能够把这些信息集成到自己的源代码内加以利用。在本书附带 CD 的 Listing 12.8 里，大家能够找到一个名为 WINSPY 的示范程序。这是除 SPY 和 SPY+以外，我们能够选择的另外一种侦查工具，这个工具程序浓缩了我们在以前章节里学习到的知识。如图 12-20 所示，这个应用程序的所有功能都要建立在鼠标活动的基础上，鼠标在这儿扮演了桌面上到处都可以出现的一个灵活的“潜望镜”的角色。

刚开始运行的时候，WINSPY 看起来完全空空如也。几乎整个客户区都由一个空的列表视窗通用控件占据着。在这个列表视窗的左侧是一个 Start 按钮以及一个鼠标形状的图标。假如希望对其他窗口进行侦查，首先需要按下 WINSPY 的 Start 按钮。单击了这个按钮以后，我们能够注意到下方的鼠标图标发生了相应的变化。一张网格覆盖了整个图标区域，从而提醒用户现在的鼠标已经被成功地捕获了（如图 12-21 所示）。

从现在开始，我们就可以在按住左鼠标键不放的前提下，在屏幕上到处拖动鼠标了。对于这一点，我们在以前曾经介绍过。现在应用程序可以接收到由于鼠标在屏幕上的每次运动而生成的所有 WM_MOUSEMOVE 消息。一旦我们离开了在屏幕上 WINSPY 程序的矩形区域，第一个窗口就由鼠标侦查到了，并且立即插入列表视窗控件内。鼠标指针在屏幕上的漫游过程中，只要热点下方侦查到了一个新窗口，就会针对这个窗口进行重复的处理，如图 12-22 所示。

假如想中断对新窗口的侦查，可以按下位于同一位置处的那个按钮。这时我们注意到，以前的 Start 标签已经变成了 Stop。假如侦查到的窗口没有实际的图象与它关联在一起，列表视

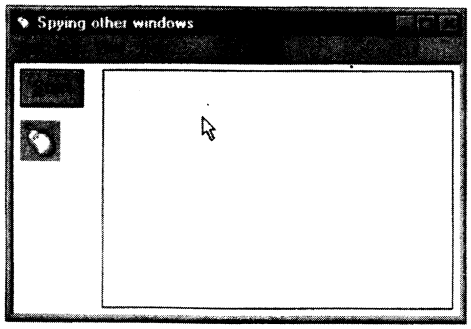


图 12-20　WINSPY 示范程序向我们揭示了如何从属于其他进程的窗口里收集信息

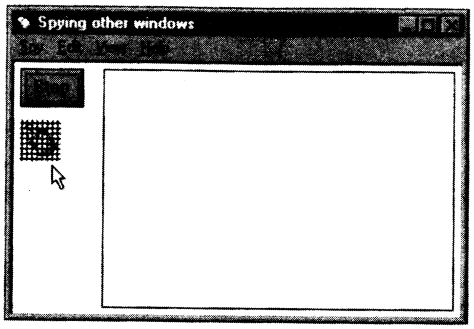


图 12-21　鼠标被捕获以后，图标后面的网格图形化地表达了这一情况

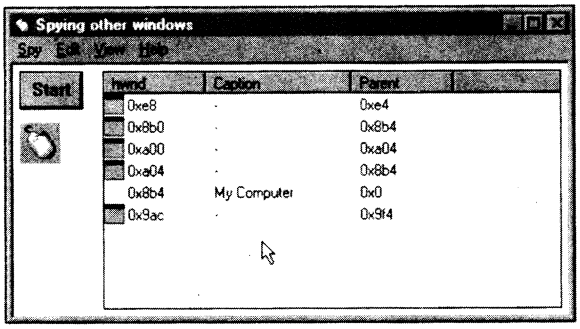


图 12-22　由 WINSPY 应用程序侦查到的几个窗口

窗控件就用一个普通的图标表示这个窗口。否则，列表视窗控件就会显示真实的图标，从而提供更佳的视觉效果。列表视窗项目都是用它们各自的句柄为标识的，这个句柄在系统级别对它们进行了唯一性标识。假如切换到详细列表模式，就能再提供一些附加的信息，比如窗口的标题（如果有的话），以及父窗口的句柄等等，如图 12-23 所示。

为了对侦查过程中针对某个特定窗口取得的信息进行全方位的了解，我们可以双击相应

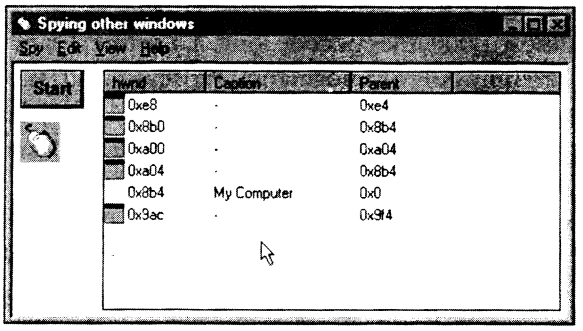


图 12-23　在详细列表模式下，列表视窗控件提供了一些附加的信息

的列表视窗项目。这时会在屏幕上显示出一个新的、独立的弹出式窗口，被侦查窗口的句柄会作为这个新窗口的标题显示出来，如图 12-24 所示。

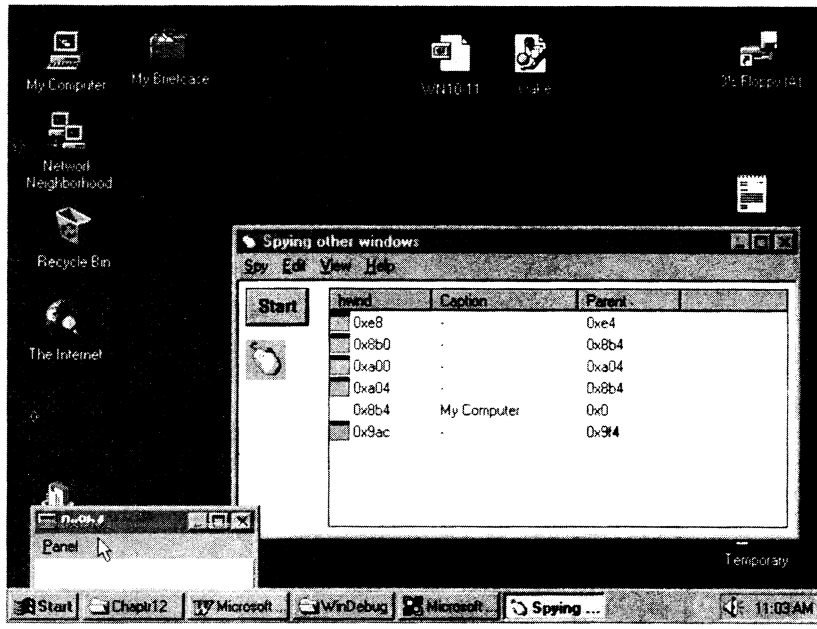


图 12-24　为了形象地显示出与被侦查窗口有关的附加信息，WINSPY 针对每个被侦查的窗口都建立了一个单独的弹出式窗口

WINSPY 本身组合了一个主窗口以及一些控件。所有这些结构化控件都会在鼠标捕获以后自动地排除掉，从而避免用户不小心地插入了属于应用程序本身的对象。

Spy 菜单内包含了两个选项，这两个选项最初都是屏蔽起来的。为了腾空列表视窗控件，以及消除以前建立的所有弹出式窗口，我们就需要用到这两个选项。除此以外，列表视窗内的每个窗口都有它们自己的关联菜单。在这种关联菜单里，前面两个选项分别指出了它在控件内的位置以及被侦查窗口的总数。第三个也是最后一个菜单项是 Terminate（中断），这个选项的作用是很特殊的。假如选中这个选项，就会实际中断被侦查窗口从属的那个进程。我

知道这并不是一种很理想的方案，然而在不使用 PVIEW95 程序的前提下，这样做确实能够发现并中止那些隐匿于后台的“僵尸”程序。

到此为止，WINSPY 程序实现的特性基本上就介绍完了。然而，只需要增添少许几行代码，我们就能再为它增加更多、更实用的功能。在第 16 章“Win95 外壳的开发”里，我们会利用一个程序示例来讲述如何把一些信息从 Win32 进程里拖至系统外壳。这种特性显著地改善了每个单一窗口的灵活性。举个例子来说，我们可以把标题栏图标拖动到桌面，从而把属于一个窗口的信息永久性地保存下来，或者把它打印下来，留待日后进行检查。

WINSPY 的工作原理

这个程序大概有 1000 行的代码长度。因此，我建议大家把它打印下来，以便对其进行仔细的研究。从被侦查窗口里取回的所有信息都动态地存储于内存块内。这些内存块是在程序刚开始执行的时候就已分配好的。另外，如有必要，这些内存块的数目还能根据需要进行相应的扩充。在其中一个内存块里，应用程序把自己使用的所有窗口句柄都存储于其中。捕获了鼠标以后，这种窗口句柄数据库就会不断地进行扫描，从而核对鼠标下方的窗口是否已经被捕获了。

```
...
case WM_MOUSEMOVE:
{
    HWND hwndSpy, hwndLV, *phwnd;
    POINT pt;
    int i, iWnd, iTot;

    // skip if the mouse is not captured
    if(! fCapture)
            break;

    // hwndLV
    hwndLV = CTRL(hwnd, CT_LISTVIEW);

    // mouse position
    pt.x = MAKEPOINTS(lParam).x;
    pt.y = MAKEPOINTS(lParam).y;
    // convert it in screen coordinated
    ClientToScreen(hwnd, &pt);

    // pointer to the handle database
    phwnd = (HWND *)GetWindowLong(hwnd, 0);
    // how many standard windows?
    iWnd = (int)GetWindowLong(hwnd, 4);
```

```
        // skip if the page is already full
        if(iWnd == (4096 / sizeof(int)))
                break;

        // is there a window underneat the mouse hot-spot?
        hwndSpy = WindowFromPoint(pt);
        // exclude the windows already in the database
        for(i = 0; i < iWnd; i++)
        {
                if(hwndSpy == *(phwnd + i))
                        return FALSE;
        }

        // store the new window handle
        *(phwnd + iWnd) = hwndSpy;
        // increment the window counter
        iWnd++;

        // save the updated counter
        SetWindowLong(hwnd, 4, (long)iWnd);
        // how many items in the listview?
        iTot = ListView_GetItemCount(hwndLV);

        // insert the new window in the listview
        FillListView(hwndLV, hwndSpy, pspywnd + iTot);

        // abilitazione menuitem Delete All Items
        EnableMenuItem(GetMenu(hwnd), MN_DELETEALLITEMS,
            MF_BYCOMMAND | MF_ENABLED);
    }
        break;
...
```

当前的鼠标位置会转换到屏幕坐标系统里，然后传递给 WindowFromPoint () 这个 API 函数。这个函数的作用是返回正好位于屏幕上那个位置处的窗口的句柄。它肯定不会返回一个 NULL 值，因为屏幕上肯定有一个窗口存在，要么是桌面，要么就是其他任何一个窗口。这也解释了为什么鼠标指针刚一离开 WINSPY 的客户区，就能返回一个窗口的句柄。在这以后，新侦查到的窗口就会与存储于特定内存块内的现成窗口句柄进行比较，从而确保它不是以前

侦查过一个老窗口。一旦这种测试通过，一个新项目就会插入列表视窗控件内。

在 WINSPY 里，另外一段值得注意的代码是与进程的中断有关的。假如用户在某个列表视窗对象的关联菜单内选中了 Terminate 菜单项，与这个对象关联在一起的进程就会中止。

```
...
case MN_TERMINATE;
{
    DWORD dwPID, dwTID;
    HANDLE hProcess;
    HWND hwndLV = CTRL(hwnd, CT_LISTVIEW);
    LV_ITEM lvi;
    int iItem;
    HWND hwndSpy;
    PSPYWND pspywnd;

    // which item is currently selected?
    iItem = ListView_GetNextItem(hwndLV, -1, LVNI_SELECTED);
    // access the memory block
    pspywnd = (PSPYWND)GetWindowLong(hwnd, 8);

    // get the item information
    lvi.mask = LVIF_IMAGE | LVIF_PARAM;
    lvi.iItem = iItem;
    lvi.iSubItem = 0;
    ListView_GetItem(hwndLV, &lvi);

    // selected window handle in the listview
    hwndSpy = (HWND)((pspywnd + lvi.lParam) -> hwndSpy);

    // get the process PID and TIP of the selected window
    dwTID = GetWindowThreadProcessId(hwndSpy, &dwPID);
    // get the process handle
    hProcess = OpenProcess(PROCESS_TERMINATE, FALSE, dwPID);
    // terminate the process
    TerminateProcess(hProcess, 0);
}
...
```

GetWindowThreadProcessId () 能帮助我们完成所有这些工作。这个函数可以返回线程

ID，并且只需要通过对窗口句柄简单的检查，就能提供出相应的进程 ID。一旦得到了相应的 PID（进程 ID），我们就能通过调用 OpenProcess（），从而对那个进程进行访问，这个函数能返回进程的句柄。相反，调用 TerminateProcess（）则能轻而易举地“杀”掉一个进程！正如我们在第 2 章“Win32 开发工具”里曾经看到的那样，在那儿我们探讨了“两层理论”。在 WINSPY 程序里，从一个窗口句柄开始，应用程序可以取回一些系统级别上的信息（PID 和 TID 与其他进程有关，并非与当前进程有关）。

到现在为止，大家已经知道了如何取得和处理进程的 PID（进程 ID）和 TID（线程 ID）。接下来，大家可以试着对 WINSPY 程序的功能进行一些有用的扩展。举个例子来说，假如我们能够对属于某一指定线程的所有窗口进行计算，那么从中得到的好处显然是不言而喻的。另外，我们还能用比 SPY＋＋更有效的一种方式来表达这种信息。就这种想法来说，EnumThreadWindows（）函数可以帮助我们列出某个指定线程内创建的所有窗口，前提是给出这个线程的 ID。

12.11 一个多媒体 CD 播放器

本书的最后一个例子是 PLAYCD，这个程序的源程序可以在本书附带 CD 的 Listing 12.9 里找到。从名字就可以推断出，这是一个 CD 播放器。Windows 95 本身已经装备了一个非常不错的 CDPlayer 应用程序，另外在 BBS、联机服务以及其他资源里都能找到类似的程序。为什么我们要重新费一番力气再设计一个播放器呢？好了，这儿至少有三个原因促使我们这样做。首先，我想对 Win32 提供的多媒体 API 进行测试，并对开发这种应用程序的难度有个感性认识。其次，我想试试与 ODBC 的连通性如何，从而试验从一个 MS Access 数据库里取得相应的信息。第三，尽管存在数量众多的 CD 播放器，但是真正能够满足我的要求的却一个也没有。我并不是一个高水平的音乐鉴赏家，然而我个人认为那些 CD 应用程序在设计上都不是很周到。例如，它们都要求人工按下前进或者倒退键才能到达一个特定的音轨。这种操作是相当麻烦的，而且也浪费了宝贵的时间，使我们无法立即选择自己喜爱的音乐。所有这些问题都促使我不得不考虑编写这个 PLAYCD，它的显示情况如图 12-25 所示。

图 12-25 PLAYCD 是一个标准的 CD 播放应用程序，
在功能上已经有了很大的改进

我并没有花上几天的时间来设计用户界面，然而这个程序确实是按照 Windows 95 外壳的某些设计准则实施的。正如大家通过图 12-25 可以看出的那样，这个程序没有菜单栏。我在这儿的想法是：所有命令都应该通过按下特定的按钮，或者通过一个关联菜单来执行。在

PLAYCD 的任何地方单击鼠标右键，都会显示出一个弹出式菜单，其中列出了一些选项，如图 12-26 所示。其中一些只是用于程序调试的目的，假如读者准备把它变成一个商业程序，就应该删除那些选项。其中一个选项是 Select CD title（选择 CD 标题）。假如选中这条命令，整体窗口的尺寸都会扩大，从而在右侧显示出一个列表视窗控件，并且在列表视窗下方显示出一个小按钮，如图 12-27 所示。列表视窗里填写了来自 MS Access 7.0 的某个 .MDB 文件内的所有 CD 标题。假如选中一个标题，PLAYCD 的窗口就会立即恢复到原始的大小，并且在命令按钮下方的三个静态窗口内填写相应的歌唱者名字、CD 标题以及当前音轨上的乐曲名称。

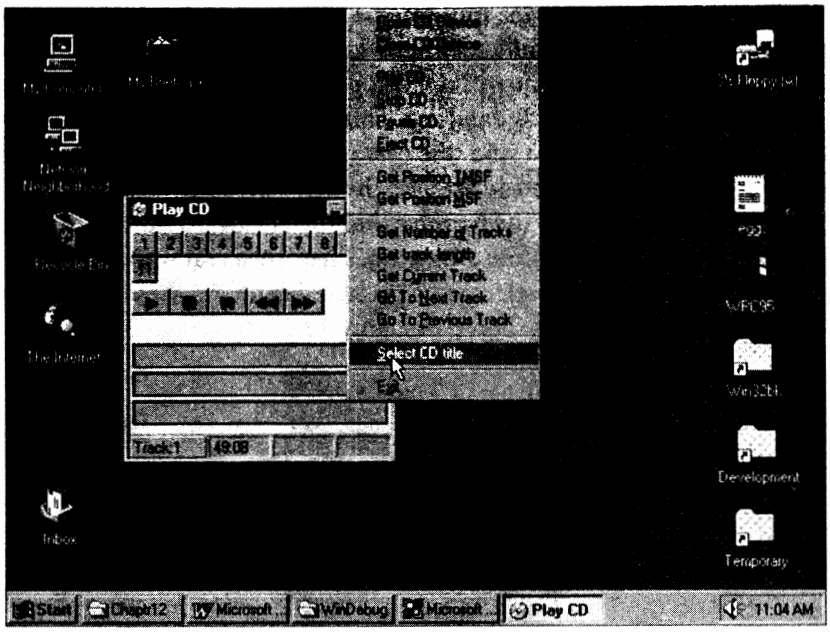


图 12-26　PLAYCD 的关联菜单内包含了用于对应用程序进行查询的几个菜单项

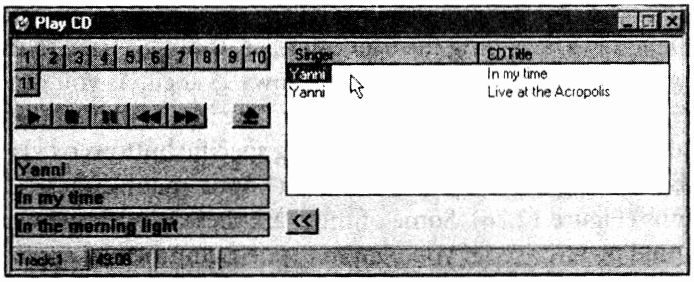


图 12-27　通过 ODBC 从一个相应的 MS Access 数据库内选择一个 CD 标题

对于显示音轨信息的那个静态窗口来说，每当播放一首新乐曲的时候，其中的内容就会自动更新。假如数据库内没有包含当前播放 CD 的信息，我们就可以按下带有两个尖角括号的

小按钮，从而把窗口缩小到原始尺寸。

PLAYCD 的另外两个特性才是这个应用程序最精彩的地方。首先，窗口内的所有组件都会自动适应系统的颜色方案，无论用户是否对这种方案进行了更改。大家可以亲自动手试验一下这个特性，看看它以另外一种颜色方案显示出来的样子。其次也是最显著的一个特性，那就是在 PLAYCD 里选择一个特定的音轨只需要按下对应的按钮就可以了。花不到一秒钟，我们就可以从一个音轨切换到另外一个音轨，这样就不用对整张 CD 进行浏览了。PLAYCD 可以在两行的范围内对音轨按钮进行调整，每一行最多可容纳 10 个按钮，所以一共可以显示 20 个音轨按钮。当然，只要你愿意，这种布局很容易就可以改变。比如，我们可以改动源程序，使其能用第三行显示按钮。

窗口中部的那些按钮控制着一些的基本的、传统的功能，比如播放 CD、暂停播放或者步进到下一个音轨等等。对于每个按钮来说，我们都可以看到相应的工具提示信息，它们对不同按钮的功能进行了简短的说明。

弹出式按钮可以对 CD 驱动器进行控制。按下这个按钮，音乐播放就会中断，然后把 CD 唱片弹出来。假如再次按下这个按钮，CD 门就会关上（前提是 CD 驱动器要支持这项特性）。不幸的是，在 MCI API 函数里，没有提供一种标准的方式可以对 CD 驱动器门状态进行监视。

在窗口的最底部是一个小的状态栏，其中显示了当前的音轨编号、当前 CD 的总长度、当前音轨已经播放的时间以及播放完这条音轨所需的时间总长。当前已播放时间是用数字形式显示出来的，这个数字显示于状态栏的一个物主绘图部分里。这就意味着在该应用程序的资源文件里，我们能找到每个阿拉伯数字对应的一幅图象，而源代码则向我们阐述了如何建立一个图象列表，这个列表可以在显示的时候对其中的图象进行伸缩处理。

12.11.1 PLAYCD 的工作原理

PLAYCD 对 MCI 消息以及命令进行了多方面的运用，这些消息和命令可以对不同个人计算机内的 CD 单元进行查询和驱动。我不准备对每个命令都进行详细的解释，要不然这一章就要变成一本 MCI 手册了。出于对方便性和可读性的考虑，我把实现特定功能的所有 MCI 命令都封装到了几个单独的例程内。这样就可以简化我们对这份总长为 1600 行的源代码清单的分析。

下面列举的 GetNumberOfTracks () 函数可以帮助我们理解如何把一个 MCI 命令发送给 Media Control Interface（媒体控制接口，即 MCI）。MCI 命令的基础是一个 MCI_STATUS_PARMS 数据结构，这是在其中的 dwItem 项里定制的。API 函数 mciSendCommand () 的语法随后用一个设备标识符（wDevID）、一条消息（MCI_STATUS）以及一系列标志（dwFlags）来完成。返回值指出了函数的执行是否成功。GetNumberOfTracks () 例程可以对媒体接口进行查询，从而判断出一张音乐 CD 里的音轨总数，如下所示：

```
DWORD GetNumberOfTracks(HWND hwnd, WORD wDevID)
{
    MCI_STATUS_PARMS mcistatus;
    DWORD dwFlags;
    DWORD dwRes;
```

```
    // retrieving the total number of tracks
    dwFlags = MCI_STATUS_ITEM;
    mcistatus.dwItem = MCI_STATUS_NUMBER_OF_TRACKS;
    dwRes = mciSendCommand(wDevID, MCI_STATUS, dwFlags, (DWORD)
        (LPSTR)&mcistatus);
    if(dwRes)
    {
            ErrorProc(dwRes);
            return FALSE;
    }
    return mcistatus.dwReturn;
}
```

相同的逻辑亦可应用于 PLAYCD 里以各种菜单项或者按钮形式实现的其他几乎所有查询机制。对这些数据进行访问所需的一种关键数据是设备的 ID，这个 ID 是在启动 PLAYCD 以后立即发出一条 MCI_OPEN 命令而返回的。

12.11.2 MS Access 7.0 数据库

MUSIC.MDB 数据库里提供的所有 CD 标题和信息是用两张表格合成起来的，这两张表格分别名为 Singers（歌唱者）和 Tracks（音轨）。也许会有人事先希望将出现一个规范化的超级关系型数据库，现在这些人要失望了。Singers 表格的结构如下所示：

歌唱者表格	字段类型
CDID（唱片 ID）	AutoNumber
Singer（歌唱者）	正文（50 个字符）
CD title（CD 标题）	正文（40 个字符）

其中，CDID 扮演的角色最为关键。Tracks 表格内则包含了每张 CD 所有节目的标题（两张表格之间是“一对多”的关系，即 1:: M）。

音轨表格	字段类型
TrackID（音轨 ID）	AutoNumber
CDID（唱片 ID）	数字
TrackName（音轨名）	正文（30 个字符）

其中，TrackID 的作用最为关键，CDID 的作用则是定义两张表格之间的链接关系。正如预期的那样，这个数据库非常简单。它在这儿的目的是向大家揭示如何通过 ODBC 以及一个 32 位驱动程序访问其他任何一个 .MDB 文件。首先，我们必须通过控制面板内的 ODBC32 对象设置与数据库间一种正确的和功能性的连接，如图 12-28 所示。

如图 12-29 所示，其中的 Setup 窗口列出了与 MUSIC.MDB 数据库的所有连接细节，这

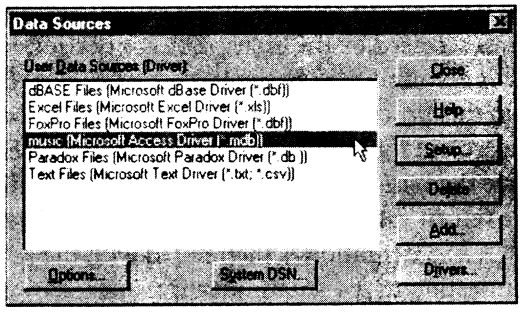


图 12-28　控制面板内的 ODBC32 提供了音乐数据源

个数据库存储于与 PLAYCD 程序相同的文件夹内。假如读者在运行的时候碰到了任何问题，请首先检查这个窗口，从中排除所有的干扰。

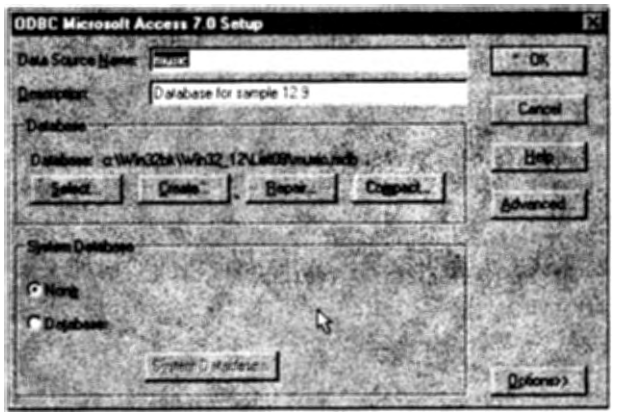


图 12-29　Setup 对话框向我们展示了与 MUSIC. MDB 这个 MS Access 7.0 数据库连接的所有细节

利用控制面板设置好与一个数据源的链接以后，为了测试它的工作是否正常，我们可以利用 MS Excel 或者由 ODBC SDK 提供的测试程序对 MUSIC. MDB 内的信息进行访问。在 PLAYCD. C 里，我们必须包含下面这些头文件，否则应用程序便无法正常编译：

```
#include <sql.h>
#include <sqlext.h>
#include <odbcinst.h>
```

除此以外，ODBC32. LIB 输入库应该增添到由链接程序访问的标准库文件序列里。与 MUSIC. MDB 数据库的连接是在 WM_CREATE 消息里完成的。这种连接需要涉及到几个步骤。首先，我们需要用到两个句柄，一个与环境有关，另一个则与实际的连接有关。SQLAllocEnv（）和 SQLAllocConnect（）可以分别提供这两个句柄。对 ODBC API 函数进行运用的时候，我们还必须测试返回值，从而检查函数的执行是否成功。SQL_SUCCESS 定义随后将与这些 API 的返回值一起进行测试。与数据源的连接是通过 SQLConnect（）函数进行管理的，数据源的名字将在该函数的 szDSN 参数里进行传递：

```
...
char szDSN[] = "music";
...
// get the environment handle
rc = SQLAllocEnv(&gMapInfo.henv);
if(rc ! = SQL_SUCCESS)
    return FALSE;

// Attempt a connection
if(SQL_SUCCESS ! = SQLAllocConnect(gMapInfo.henv, &gMapInfo.hdbc))
    return FALSE;

// connect to the data source
rc = SQLConnect(gMapInfo.hdbc, szDSN, SQL_NTS, NULL, 0, NULL, 0);
if(rc == SQL_ERROR)
{
    DWORD dwErr;
    char szErrorMsg[200], szSqlState[200];
    short cbErrorMsg;

    // get the connection error
    SQLError(gMapInfo.henv, gMapInfo.hdbc, SQL_NULL_HSTMT,
      szSql State,&dwErr, szErrorMsg, sizeof(szErrorMsg), &cbErrorMsg);
    SQLDisconnect(gMapInfo.hdbc);
    SQLFreeConnect(gMapInfo.hdbc);
    break;
}
// storing the database name
strcpy(gMapInfo.szDSN, szDSN);

// allocate a statement handle
rc = SQLAllocStmt(gMapInfo.hdbc, &gMapInfo.hstmt);
if(rc == SQL_ERROR)
{
    SQLDisconnect(gMapInfo.hdbc);
    SQLFreeConnect(gMapInfo.hdbc);
}
...
```

假如在这个阶段产生了错误，我们就可以通过 SQLError () 取得相应的扩展出错信息，然后利用 SQLDisconnect () 以及 SQLConnect () 这两个函数屏蔽两个句柄。从编程的角度来讲，我们最好时刻取得扩展出错信息，因为这在许多场合下都是相当有用的。在这一部分结束的时候，我们通过 SQLAllocStmt () 函数分配了一个语句句柄。这是通过 ODBC 向数据库发出 SQL 语句的最基本的一种工具。与 ODBC 的每一次交互作用都必须在最后用 SQLCancel () 函数进行清除。

假如连接成功地建立起来了，应用程序就会在主窗口的一个不可见区域创建列表视窗控件，同时从数据库表格栏内直接取得栏目标题。SQLColumns () 函数指定了我们感兴趣的表格栏。随后，我们再利用 SQLBindCol () 完成表格栏与应用程序代码内某个缓冲区的实际连结工作。在下面这个代码段里，我们利用了缓冲区来定义栏标题，这个栏标题位于列表视窗的 LV_COLUMN 数据结构里，该数据结构扮演了在列表视窗控件与 Singers 表格之间进行牵线搭桥的角色：

```
...
// initializing the columns
lvc.mask = LVCF_FMT | LVCF_WIDTH | LVCF_TEXT | LVCF_SUBITEM;
lvc.fmt = LVCFMT_LEFT;
lvc.cx = EXPANDWINDOW / 2;
lvc.pszText = szString;

// calculating the columns in the Singers Table
ret = SQLColumns(pMapInfo -> hstmt, NULL, 0, NULL, 0, szTable,
    sizeof(szTable),NULL, 0);
// 4 identifies the column names
ret = SQLBindCol(pMapInfo -> hstmt, 4, SQL_C_CHAR, szString,
    sizeof(szString),NULL);

// skip the CDID column
ret = SQLFetch(pMapInfo -> hstmt);
i = 0;

// inserting the columns
while((ret = SQLFetch(pMapInfo -> hstmt)) == SQL_SUCCESS)
{
    lvc.iSubItem = i;

    // insert the column title
    if(ListView_InsertColumn(hwndLV, i, &lvc) == -1)
```

```
            return NULL;
        // increment the counter
        i++;
    }
    ...
```

假如用户在一位歌唱者的名字上面双击，窗口就会恢复成它最先的大小，而列表视窗则会通过一条 WM_NOTIFY 消息与主窗口进行通信（无论是 NM_RETURN 还是 NM_DBLCLK）。这时，应用程序关联菜单内的 SelectCD title 菜单项会通过 SetMenuItemInfo () 函数暂时屏蔽起来，同时决定使用选中的列表视窗项目。列表视窗的内容（歌唱者的名字以及 CD 标题）随后会拷贝到静态窗口里。为了取得音轨标题，应用程序必须发出一条 SQL 语句，从而取得来自 Tracks 表格内与选中 CD 有关的所有标题。SQLExecDirect () 这个函数可以帮助我们完成向 ODBC 驱动程序发出一条 SQL 指令的任务（从音轨中选择 *，其中的 tracks.cdid 等于%d）。ODBC 驱动程序最终会把这条指令转换成 MS Access 引擎能够理解的一条命令。

```
    ...
    case NM_RETURN:
    case NM_DBLCLK:
    {
        char szText[200];
        int iPos = 0, i, iData;
        LV_ITEM lvi;
        char szSQL[300] = "select * from tracks where tracks.cdid = %d";
        RETCODE rc;
        HWND hwndLV = pnmv -> hdr.hwndFrom;
        short int sCols;
        DWORD dwValue;
        MENUITEMINFO mii;

        // enable the Select CD command
        mii.cbSize = sizeof(mii);
        mii.fMask = MIIM_STATE;
        mii.fState = MFS_ENABLED;
        SetMenuItemInfo(hmenu, MN_SELECTCD, FALSE, &mii);

        // determine the selected CD
        iPos = ListView_GetNextItem(hwndLV, -1, LVNI_SELECTED);
        // skip if iPage is -1
```

```
if(iPos == -1)
        break;

// get the item text
ListView_GetItemText(hwndLV, iPos, 0, szText, sizeof(szText));
// set the item in the CT_SINGER window
SetWindowText(CTRL(hwnd, CT_SINGER), szText);

// get the CD TITLE
ListView_GetItemText(hwndLV, iPos, 1, szText, sizeof(szText));
// set the item in the CT_SINGER window
SetWindowText(CTRL(hwnd, CT_CDTITLE), szText);

// get item position in LISTVIEW coordinates
//ListView_GetItemPosition(hwndLV, iPos, &pt);

// get the CD TITLE ID
lvi.iItem = iPos;
lvi.iSubItem = 0;
lvi.mask = LVIF_PARAM;
ListView_GetItem(hwndLV, &lvi);
iData = lvi.lParam;

// prepare the SQL statement
wsprintf(szSQL, szSQL, lvi.lParam);

// lets query the table
rc = SQLExecDirect(gMapInfo.hstmt, szSQL, sizeof(szSQL));
if(rc)
    MessageBeep(0);

// count the table columns
SQLNumResultCols(gMapInfo.hstmt, &sCols);

// zero the counter
i = 0;
while((rc = SQLFetch(gMapInfo.hstmt)) == SQL_SUCCESS)
{
    // get the data for Track title column (#3)
```

```
            SQLGetData(gMapInfo.hstmt, 3, SQL_C_CHAR, szTitle[i],
            sizeof(szTitle[i]), &dwValue);
            i++;
        }
        SQLCancel(gMapInfo.hstmt);

        // show the track title
        SetWindowText(CTRL(hwnd, CT_TRACKNUM), szTitle[wCurrTrack
            - 1]);

        // restore the window to its original size
        SetWindowPos(hwnd, HWND_TOP,
                     0, 0,
                     TRACKBTNSZ * MAXBTNS + 4 * SYS(SM_CXEDGE) +
                     6, CDWINDOWHEIGHT,
                     SWP_NOMOVE);
    }
        return FALSE;
    ...
```

符合 SQL 语句条件的结果栏目会通过 SQLNumResultCols () 函数返回。根据数据库的布局，在 Tracks 表格的第三栏里包含了音轨标题。SQLSetch () 和 SQLGetData () 函数的组合分别从查询结果里取走了一行数据，并且针对当前行内的一个单一未限制栏目返回了数据。音轨标题的整个集合随后就会在内存里保存下来，并且对 PLAYCD 窗口下侧那个静态控件的显示内容进行周期性的动态更新。

12.12 建立一条工具提示

对 CD 播放器基本操作进行控制的按钮 (Play，Stop 以及 Pause 等等) 都拥有一个关联在一起的工具提示。假如让鼠标指针在这些按钮上方静止不动一小会儿，就会显示出一个正文串，这个正文串对不同按钮的功能进行了简短的介绍。这种行为是由一个工具提示通用控件管理的，如下所示：

```
    ...
    // creating a tooltip
    hwndToolTip = CreateWindowEx(WS_EX_TOPMOST,
                        TOOLTIPS_CLASS, NULL,
                        TTS_ALWAYSTIP,
                        CW_USEDEFAULT,CW_USEDEFAULT,CW_
                        USEDEFAULT,CW_USEDEFAULT,
```

```
                    NULL,
                    NULL,
                    hInstance,
                    NULL);
...
```

这个窗口是看不见的，它的主要任务就是对已经“注册”了一个工具的所有控件进行监视，这种监视是通过发送一条 TTM_ADDTOOL 消息来实现的。这条消息是专门针对工具提示通用控件类而设计的，它可以通过一个 TOOLINFO 数据结构向工具提示控件传送相应的信息。

```
...
// filling the TOOLINFO structure
tlinf.cbSize = sizeof(TOOLINFO);
tlinf.uFlags = TTF_IDISHWND;
tlinf.hwnd = hwnd;
tlinf.uId = (UINT)hwndBtn;
tlinf.hinst = hInstance;
tlinf.rect.left = tlinf.rect.right = tlinf.rect.top = tlinf.rect.bottom = 0;
tlinf.lpszText = "Play";

// adding a tool
if(! SendMessage(hwndToolTip, TTM_ADDTOOL, 0L, (LPARAM)
    (LPTOOLINFO)&tlinf))
    MessageBeep(0);
...
```

在 TOOLINFO 结构的不同项中，工具提示应该监视的是控件的句柄，以及将显示出来的正文串。一个单独的工具提示可以同时对几个控件进行控制，甚至可以在鼠标指针位于一个特定的矩形窗口区域内时显示出工具提示信息。

第13章　内存管理和DLL

尽管我们在第一章“Win32中的软件开发”里就已经讲述了关于内存管理规则的一些问题，但是围绕这个主题仍然有大量的东西需要我们去进行探讨。在这一章里，我们准备讨论用于创建快速执行程序的一些方法和设计准则。本章的内容也涉及到堆管理技术以及独立进程之间的内存共享问题。除此以外，我们还要专门去揭示动态链接库（DLL）背后隐藏的“秘密”。事实上，尽管名字听起来很吓人，但是DLL实际上是一个相当简单的主题。

13.1　关于内存页的更多问题

从理论上说，就目前在Intel 32位处理芯片上实现的Windows 95和Windows NT来说，每个Win32进程都可以访问4GB的专用地址空间。这个庞大的“平滑”地址空间主要分割成两个具有明显区别的区域，每个区域占据2GB的空间。假如大家对第一章还有印象，就应该回忆起我们曾经提到过NT和Win95对这个问题的处理是各不相同的。之所以会出现这种差异，主要是由于Win95的“混血”本质。出于兼容性方面的考虑，Win95仍然可以与16位计算环境带来的某些产物共存。无论在哪一种情况下，对于同时运行于这两种操作系统下的一个Win32进程来说，我们都可以假设它至少能对2GB的虚拟地址空间进行寻址，这已经是一个相当大的内存容量了。

尽管这种内存地址空间是“平滑”的，但是我们仍然不能把它当作一个唯一的和线性的表面来进行访问。这些空间在内部都被分割成了4K大小的内存单元，我们把这种单元叫作“页”，这是最小的内存分配单位。第一章介绍的EXCEPT示范程序有助于我们熟悉这种分页技术，并且理解怎样在一个Win32应用程序里对其加以利用。

Win32进程访问某个物理内存位置的时候，首先要经过一个Page Directory（页目录）以及一个Page Table（页表），最终才能抵达包含了实际数据的Page Frame（页帧）。这种操作既有可能相当复杂，也有可能相当简单。它的优点主要在于实现了内存管理从系统内安装的真实物理内存到程序设计上的抽象化，允许为每个进程分配比物理内存更多的地址空间。然而这种技术的缺点却在于它影响了系统的性能。通过对这种技术更为细致的研究，我们会发现有两个负面因素影响了系统的性能：内存开销以及页错误。

每个Win32进程都有它们自己的Page Directory，这是一个长度为4K的页，它的基准位置是由系统在Control Register #3内载入的。考虑到一个Win32进程最高可以对4GB的空间寻址，所以就意味着需要其他一些页来完成这种寻址操作。换句话说，一个Win32进程需要一个Page Directory加上1024个Page Table，才能对整个地址空间进行访问（1024个PDE×1024个PTE×4KB=4GB，其中PDE和PTE是实际的条目）。因此，每个进程都需要占据1025个页（1+1024），这总共已经超过了4MB的内存（1025×4K=4198400）。

启动的进程越多，必须使用的页数就越多。幸运的是，启动一个新进程的时候，所有1024个页都还没有建立起来，它们严格限制在只有与应用程序的需求匹配的时候才会建立起来。尽管如此，从理论的角度来看，对于系统内建立的每个进程来说，它们都有一个4MB多的内存

开销。

准确地说，这些页在什么地方呢？它们都驻留于物理内存里吗？我们几乎不可能准确地回答出这个问题，因为这是一种混合起来的情况。一些页可能驻留在内存里，另外一些则有可能移到了系统硬盘驱动器的分页文件里。Win32 内存管理程序掌握了每个页的细节信息，尽管这些信息会连续不断地发生改变。最糟糕的情况发生于所有页都需要访问驻留于分页文件里一个特定内存位置的时候；对于物理内存有限的计算机系统来说，这种情况并不少见。在这种情况下，从 32 位虚拟地址到内存里一个单独的字节的过渡会显得相当困难，这是由于缺少一个中间步骤造成的——这个中间步骤也就是实际的内存页。系统试图访问一个内存位置的时候，假如这个位置已经跟随一个内存页在以前交换出去了，那么就会产生 Page Fault（页错误），从而放慢了整体的处理步调。

现在假设一个应用程序需要面对我们想象得出的最坏的一种情况：无论页目录、页表还是页帧都不在物理内存里。这时的程序代码就会导致三个页错误异常事件，这样就损失了一定数量的时间从分页文件里对每个页进行恢复，从而降低了系统的整体性能。一个页被载入对应的内存位置以后，系统就会再次执行导致异常事件的指令，从而继续完成自己的工作。

作为应用程序的开发者，我们无法对所有这些内部的内存管理特性进行控制，并且除了尽量保证这种异常事件不经常发生以外，我们几乎不能再针对它做其他任何事情。总而言之，我们不可能防止这类事件的发生。我唯一的忠告就是无论作为开发者还是用户，都应该在自己的计算机内安装大量物理内存，这样可以显著减少这类事件发生的频率。这也是我现在为什么要使用一台带有 64 MB 的 AST Pentium 90 计算机，放弃使用以前的 32MB 系统的原因！

13.1.1 转换后备缓冲区

正如我们在第一章“Win32 中的软件开发”里就已经指出的那样，Intel 32 位处理器体系有一个内部的内存区域，叫作“转换后备缓冲区”（Translation Lookaside Buffer），即我们常说的 TLB。TLB 的行为就好象一个片上高速缓冲，其中存储了 32 个最近用到的虚拟地址以及它们对应的物理地址（页帧位置）。因此，TLB 是一个 64KB 的区域，其中容纳了以前从虚拟地址到物理地址的一些最终的转换结果，这样就避开了麻烦的分页算法，不用一个一个计算页目录、页表以及最终的页帧。

每个 TLB 条目都包含了虚拟地址值（初始信息）以及 PTE 的 20 位宽度地址，这些地址标识了对应的页帧。通过在物理页地址上面增加虚拟地址的偏移部分（最后 12 位），系统很容易就可以计算出那个页内的实际位置。这样一来，对驻留于相同物理页内不同内存位置的连续引用就能生成一条单一的 TLB 条目，其中保存了对其他页进行引用的空间。

当所有 TLB 条目都被占据以后，TLB 管理器就会丢弃最早的条目，从而为新来者腾出空间。由于转换后备缓冲区的存在，所以系统性能得到了显著的改善，同时限制了某些情况的负面影响。这种情况我们在后面就要讲到一种，其中所有内存页都很不凑巧地交换到分页文件里去了。

1. TLB、线程以及进程

正如我们以前强调的那样，现场切换、TLB 的作用以及线程的选择都是在幕后发生的，我们在应用程序源代码内无法对其进行任何干涉。显然，TLB 里的 32 个条目足以满足所有运行进程的需求。我们几乎不可能定义一个系统里正在执行的所有进程的真正数目，因为这个参数可以有不同的变化。为了对这一点有个感性认识，大家可以参考图 13-1，其中列出了我编

写到这一章时正在运行的所有进程。

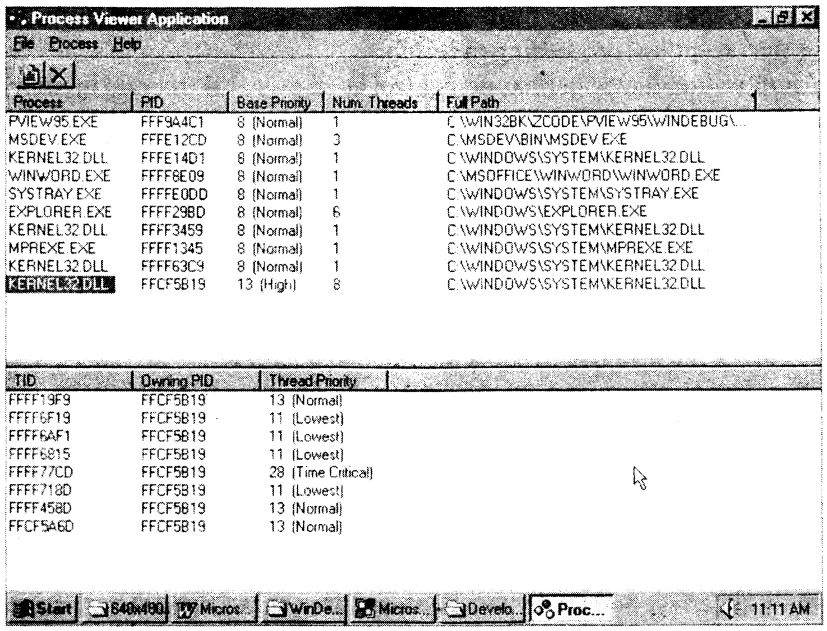


图 13-1　PVIEW95 列出了正在运行的所有进程以及它们对应的线程编号

显示出这个屏幕的时候，我的 Windows 95 系统正在运行 11 个进程，总共有 21 个线程。然而很难断定这是否与一般的情况符合，我并不认为这与人们通常运行的程序有多大的区别，但是不同的用户的确可能运行不同的应用程序组合。在这儿，我们最好能知道正在运行的每个进程所占据的真实内存空间。对于这种内存空间来说，，从技术角度来看有一个很好的术语能统称它们，那就是“工作集”（working set）。实际上，Win32 提供了 GetProcessWorking SetSize () 和 SetProcessWorkingSetSize () 这两个 API 函数，用于各自判断和定义一个 Win32 进程的工作集大小。这两个函数只能在一个 Windows NT 系统里调用，在 Win95 里则没有实现。

```
#include 〈winbase.h〉
BOOL GetProcessWorkingSetSize(HANDLE hProcess,
                              LPDWORD lpMinimumWorkingSetSize,
                              LPDWORD lpMaximumWorkingSetSize);
```

参数	说明
HANDLE hProcess	带有 PROCESS_QUERY_INFORMATION 访问权限的进程句柄
LPDWORD lpMinimumWorkingSetSize	用于接收最小工作集尺寸的一个双字地址
LPDWORD lpMaximumWorkingSetSize	用于接收最大工作集尺寸的一个双字地址
返回值	在正文里讨论

BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

尽管具备多任务能力，但事实上 Windows 95 系统在同一时刻仍然只能运行一个单独的线程。同时执行多个任务其实只是一种表面现象，这是在各个进程之间进行快速切换得到的一种结果。正如大家在第 14 章“多线程、IPC 和 I/O”里将要学到的那样，单独一个线程在 CPU 里存在的时间只有 17 毫秒左右。在这个相当有限的时间段里，一块功能强劲的 Pentium 90 处理芯片可以执行几十万条指令——尽管这种处理器的技术指标可以达到 150 MIPS (MIPS 的含义是每秒钟执行多少百万条指令)。

系统调度程序执行一次现场切换的时候，它会自动对 TLB 的内容进行更新，这样就丢失了以前存储下来的所有信息。重新填写这个数据区域是一种非常简单的工作，只需要几毫秒的时间即可完成，前提是某处的一个线程要花自己 30%到 50%的时间来计算新地址。从程序开发的角度来看，这种因素为线程带来的好处要大于为进程带来的好处。和创建一个新线程比较起来，新进程的生成是一种相当耗时的操作。除此以外，从相同进程的一个线程移至另一线程并不涉及到现场切换。因此，TLB 能把自己的内容保存下来，这样就为整体性能的提高带来了机会，这主要是由于 TLB 缓冲区内保存的一个缓存条目造成的。

2. 页表条目

图 1.12 向大家展示了一个典型的 Page Table Entry(页表条目)布局，这种布局是以 Intel 的规范和文档说明为基础的。实际实现它的时候却会随着操作系统的不同而发生变化。图 13-2 向大家展示了 Windows 95 里实现的一个页表条目的样子。

5 Bits	20 Bits	4 Bits	3 Bits
保护标志	页帧地址	分页文件	状态

图 13-2　MS Windows 95 里的一个页表条目

20 位的页帧地址相当于进入合适页文件内的一个偏移地址，用于对目标内存页进行定位。在这种情况下，存在状态 (State) 位都应该设置成零，从而表明它在分页文件内的存在。NT 和 Win95 都支持多达 16 个分页文件——每个驱动器分配一个。四个 Page file(页文件)位指出准备引用的正确页文件，以便对目标页进行访问。

存储于一个 PTE 里的信息足以描述一个内存页的完整状态。Win95 内存管理器保存了一个内部，名为 Page Frame Database (页帧数据库)，用于对每个 PTE 进行反向引用，并且按照不同内存页各自的状态，从而对它们进行分类。虚拟内存管理器需要对一个页状态位进行更新的时候，这种分类信息就显得相当重要了。

13.1.2　页边界

假设我们现在准备保留一定数量的内存页，以后再用这些页存储一个用户自定义数据结构。这个数据结构有多大呢？对于这个问题，我们很难得出周全的答案。所以有必要改变一下问法：数据结构的大小肯定是 4096 的整数倍吗？

为了回答这个经过重新考虑后提出的问题，下面让我们考虑下述的情况。我们需要把相

同数据结构的连续拷贝存储到多个内存页里。结构的数量会随着程序的进展而发生改变，这需要我们先预约一系列内存页，然后只对最初需要用到的那些页进行委托。随着程序的运行，我们也许需要存储更多的信息，所以就要对紧跟在已委托的最后一个页后面的那个页进行委托。

整个算法要用一个异常处理例程保护起来，这个例程能捕获对未委托页的一次访问，并能在产生访问违规异常事件以后对未委托页的状态进行相应的改变。只要应用程序数据结构的多份拷贝能在一个页内很好地适应，我们就可以保证自己的程序能够平稳地运行。相反，假如一系列连续的数据结构超出了某个页的边界，就会发生一系列潜在的死锁现象，因为这种情况会无休止地生成访问违规异常事件。

下面这个代码段指定了对一个新的集合数据类型的定义，它的整体尺寸并不是4096的一个整除因子：

```
typedef struct _DATA
{
    int i;
    HWND hwnd;
    char c;
} DATA, *PDATA
```

假如Structure Member Alignment（结构项排列）编译程序标志设置成四个字节，那么通过sizeof关键字对数据结构的尺寸发出请求以后，我们就可以得到12这个值。很常见的一种情况是，Win16应用程序把这个编译标志设置成1，以便在存储结构的多份拷贝必须连续存放的时候对数据进行打包信息。在这种情况下，sizeof关键字就会返回一个5，这是对每个组件范围的正确总和。无论12还是5，它们都不是4096的整除因子。在341个连续的数据结构以后，我们在第一个页里就没有足够的地方再容纳一个数据结构。第342个数据结构将同时占据第一个和第二页的空间。

很正常，刚才描述的操作会生成一次访问违规异常事件。即使当时那个指针正指向整体结构的起始处（这个位置仍然驻留于第一个页内），仍然不可避免地会产生这种异常事件。这是一种典型的Catch 22情况。异常处理例程会试图委托由这个指针指向的页；但事实上，这个页已经委托过了。因此，异常处理例程就无法解决这个问题，因为它将连续委托错误的页。

此时，我们应该修改标准的异常处理例程，使其能够理解当前面对的现状。实际上，这并不是利用具有常规用途的一个异常处理例程就可以轻松解决的问题。此外，它要求开发者进行一些特殊的编码工作，以便能反映出偶然会碰到的一些情况。然而，假如我们改变了数据结构的尺寸，就不得不同时对异常处理例程进行修改。为了避免这种潜在的运行期错误，我强烈建议大家把数据结构的尺寸限制在四字节以内，并由此设计自己的信息，使其正好是4096的一个整除除数。这样也许会浪费一些时间，但是经过这种处理后的代码显得相当稳定，并且也便于在以后的版本中对其进行维护和升级。

在本书第2章“Win32开发工具”里，我们曾经提到过一个名为INCLUDE的示范程序，这个程序并未受到页边界的限制。我们将在探讨了内存映射文件以后对这个程序的代码进行

更加深入的分析。

13.2 Malloc（）和C运行期库

需要对一些内存块进行预约、委托以及管理的时候，我强烈建议大家使用VirtualAlloc（）以及类似的API函数。尽管内存API的结构相对已经比较简单了，但是大家也许仍然愿意用更简单的函数来实现内存的访问和管理。

把现成的16位代码转换到Win32里的时候，C语言里的运行期函数malloc（）仍然被沿用下来了。malloc（）和其他所有类似的函数在Win32里都得到了支持吗？显然如此！我们可以继续使用malloc()，它们不会在Win32里产生任何问题。除此以外，它们的工作和在MS-DOS下面使用时别无它样。

当我们需要一个内存块的时候，只需调用malloc（）即可。假如执行成功，这个函数会返回一个相应的指针。调用malloc（）的时候发生了什么事情呢？首先，大家应该知道malloc（）的行为在每次调用它的时候并不完全一致。第一次调用malloc（）的时候，它会预约一系列线性内存页，这种处理在后续的操作中并不会重复。在函数返回之前，它只会分配那些严格满足应用程序需求的内存。在同一进程内，对malloc（）的后续调用肯定只限于引用最初预约的那些线性地址集，只是根据需要不断委托程序需要的内存页。

VirtualAlloc（）扮演了幕后指挥官的角色，它提供了涉及到的所有基本功能。这样一来，和Win32 API比较起来，某些运行期函数显得更加灵活、更易于使用，并且具有更强的可移植性——尽管它们要以我们在第一章介绍过的那些函数集合为基础。从性能的角度来讲，我们完全可以认为VirtualAlloc（）是一种最好的内存管理工具，尽管该函数使用麻烦，并且功能也比较低级。malloc（）部分也要依赖于堆管理API函数，以便在用户需要小于一个页的内存区域时，对页进行次级分配。

13.3 堆管理

大家还记得.DEF文件里那条不很出名的HEAPSIZE命令吗？在Windows的16位版本里，这条语句可以指示系统链接器在程序的自动数据段（Automatic Data Segment，ADS）里建立一个堆区域，以便满足应用程序的特定需要。举个例子来说，所有EDIT控件都需要把自己的缓冲区分配到应用程序的堆区域里。从第二章“Win32开发工具”开始，我们知道了在缺省情况下，一个Win32进程会在地址空间里预约256个页，从而满足堆区域的需要。然而，我们怎样访问这个内存区域，并且为什么说它在Win32里也是有用的呢？对于第一个问题，答案在于GetProcessHeap（）函数：

```
#include〈winbase.h〉
HANDLE GetProcessHeap（VOID）
```

假如调用成功，这个函数可以返回进程的堆句柄；假如失败，则返回一个NULL值。假如返回的是一个NULL值，就表明系统整体出现了严重的问题，因为我们并不需要向GetProcessHeap（）传递任何参数。一旦得到了相应的句柄，我们就可以通过调用HeapAlloc（）或者HeapReAlloc（）函数来分配一个内存块，这种操作在概念上与一次普通的malloc（）调用是没有区别的。这种方式的优点在于它比malloc（）的速度快，因为运行期库函数也

会在内部调用所有堆管理 API 函数，同时不会提供一个附加的功能。

下面考虑这样一个问题。假设我们需要存储 20 个字节的信息。这时该怎么办？用 VirtualAlloc ()分配一个新页？这样做是可行的，然而它远远谈不上最佳的一种方案。根据我们的经验来看，假如处理的对象比较小，一个页就足以容下它，那么就可以把它存储到应用程序的堆区域内。这才是最简便的一种方案，原因至少有两条。首先，对于那个对象在某个页内占据的空间来说，它也许已经在进程的虚拟地址空间里预约好了（系统载入模块在进程执行的时候就已经完成了这一工作）。因此，其他内存就不会浪费掉。其次，对应用程序堆区域内的对象进行访问可以通过标准的指针来进行，这种访问是相当直接了当的，正如下面这个代码段所示：

```
...
#define MAXLEN      200
...
HEAP heap;
char * p;
...
heap = GetProcessHeap();
...
p = (char *)HeapAlloc(heap, HEAP_ZERO_MEMORY, MAXLEN);
...
LoadString(hInstance, ST_CLASSNAME, p, MAXLEN);
...
```

进程堆里可以存储任何类型的信息，存储下来的信息在整个源代码的范围内都是“可见”的。我们可以把它看作附加内存区域（无论类附加内存还是窗口附加内存）、属性列表以及源文件范围内的标识符的一种很有价值的替换物。在同一时刻，只能有一个进程线程能够访问进程堆，这样就自动避免了线程之间的冲突。

我们可以通过调用 HeapCreate （） 函数来建立附加的堆，如下所示：

```
#include 〈winbase.h〉
HANDLE HeapCreate (DWORD flOptions, DWORD dwInitialSize, DWORD dwMaximumSize);
```

参数	说明
DWORD flOptions	可以是 HEAP_GENERATE_EXCEPTIONS 或者 HEAP_NO_SERIALIZE
DWORD dwInitialSize	用字节表示的初始堆尺寸，能自动调整到下一个页的边界处。假如为零值，则表明最初建立堆的时候没有为它分配任何字节
DWORD dwMaximumSize	假如为零值，就表明这是一个尺寸可以增大的堆
返回值	在正文里讨论

HANDLE	堆句柄；假如函数调用失败，则返回一个 NULL 值

这样的堆其实就是一个特殊的内存块，它只允许建立它的那个进程对其进行访问。我们没有办法可以把一个堆当作在应用程序之间传递信息的一个共享内存块来使用。一旦获得了相应的堆句柄以后，当前进程内的所有线程都可以访问这个堆，这个句柄是由 HeapCreate () 函数返回的。堆内存块是由一系列从虚拟内存里获得的预约页组成的，并且以 dwMaximnuSize 参数的值为基础，从而把自己的尺寸调整到上端的页边界处。假如 dwInitialSize 和 dwMaximumSize 参数的值都被设置成零，那么在缺省情况下系统就会预约 256 个页，然而不对其中的任何一个页进行委托。应用程序第一次从堆区域里申请某些空间的时候，一个或者多个页就会根据实际情况自动进行委托。假如把 dwMaximumSize 设置成零，就表示定义了一个尺寸可以增大的堆区域，尺寸的这种变化是以应用程序的需要为基础的。

我们最好按照下述的方式建立一个新堆：

```
heap = heapCreate (0, 0, 0)
```

返回的句柄指定了一个新建的堆区域，它最初的大小被设置成零个页，并且没有预先定义好的上限。缺省情况下，这样的一个堆具有串行访问本质。换句话说，就是只能一个线程接一个线程地对其进行访问，这样就避免了同一进程的两个线程同时访问这个堆区域。利用 HEAP_NO_SERIALIZE 标志，这种行为很容易就可以得到改变。当然，这种缺省的行为改变以后，既会带来一些优点，也会带来一些不良影响。

改变这种行为以后，优点在于对这种类型的堆进行访问不需要进行预防检查或者其他任何一种预处理。有冲突的线程同时对同一个堆区域进行访问是我们应该尽力避免的。假如我们的代码是严格按照单线程编写的，或者它提供了某些 IPC (进程间通信) 形式对线程的活动进行协调，那么增加 HEAP_NO_SERIALIZE 标志就不会带来任何副作用。显然，假如两个线程对相同的内存对象进行访问，同时没有使用任何一种同步协调模式，那么很有可能就会发生冲突。

Win32 提供了一种新的函数：GetProcessHeaps ()，利用它只需要一个步骤便可取得与一个进程有关的所有堆句柄：

```
#include 〈winbase. h〉
DWORD GetProcessHeaps (DWORD NumberOfHeaps, PHANDLE
ProcessHeaps);
```

参数	说明
DWORD NumberOfHeaps	指出可以在 ProcessHeaps 参数里存储的堆的数量
PHANDLE ProcessHeaps	一个 HANDLE 内存位置的地址，其中填写了几个堆句柄
返回值	在正文里讨论
DWORD	指出与当前进程关联在一起的堆的数量

返回的第一个句柄是进程堆，后面跟随了通过 HeapCreate () 建立的附加堆。

对堆的管理

无论进程堆，还是在应用程序执行期间建立的堆，只要我们拥有了一个堆的句柄，就可以调用 HeapAlloc（）函数为其分配一个内存块：

```
#include〈winbase.h〉
LPVOID HeapAlloc（HANDLE hHeap，
                  DWORD dwFlags，
                  DWORD dwBytes）；
```

参数	说明
HANDLE hHeap	堆句柄
DWORD dwFlags	堆分配标志：HEAP_NO_SERIALIZE，HEAP_ZERO_MEMORY，HEAP_GENERATE_EXCEPTIONS
DWORD dwBytes	准备分配的字节数
返回值	在正文里讨论
LPVOID	准备存放对象的内存位置的地址

假如函数调用失败，并且以前没有设置 HEAP_GENERATE_EXCEPTIONS 标志，就返回一个 NULL 值。假如已经设置了 HEAP_GENERATE_EXCEPTIONS 标志，那么调用失败的返回值就会变成 STATUS_NO_MEMORY，从而指出内存分配失败这个事实。之所以会调用失败，主要是由于缺少可用内存，或者由于某些内部原因导致了 STATUS_ACCESS_VIOLOATION（访问违规）事件。

通过 HeapAlloc（）分配的每个对象以后都可以通过调用 HeapFree（）来进行释放：

```
#include〈winbase.h〉
BOOL HeapFree（HANDLE hHeap，
               DWORD dwFlags，
               LPVOID lpMem）；
```

参数	说明
HANDLE hHeap	堆句柄
DWORD dwFlags	HEAP_NO_SERIALIZE
LPVOID lpMem	一个指针，指向准备从堆区域内删除的对象
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

除了释放内存块以外，应用程序还可以利用 HeapDestroy（）函数来消除一个新建的堆。这个函数对于以前通过 HeapCreate（）建立的所有堆来说都能成功地执行，但是千万不要把它用于对进程堆的处理。

```
#include〈winbase.h〉
BOOL HeapDestroy (HANDLE hHeap);
```

假如返回的是TRUE值，就表明指定的堆已被成功地删除了。在与堆管理有关的这个函数集里，最后几个函数包括HeapSize（），它可以返回某个堆里分配的一个内存块的尺寸；HeapWalk（）的作用则是列举出某个堆内存储的所有内存对象；而HeapCompact（）的作用是尝试对堆区域进行缩小化处理，主要用于撤消对空闲内存块的委托。

13.4 共享内存

使用Win32最大的优点是可以为每个进程分配一个专用地址空间。与此相比，Windows 3.x里的所有任务都共享一个地址空间，其中包括Windows本身。这种变化对新的Win32进程的设计和实施具有深刻的印象，现在我们不得不考虑如何与其他应用程序共享信息。

使用多个专用地址空间的主要优点到现在为止应该很清楚了——内存保护以及更宽的地址空间是其中最主要的。然而，在正在运行的所有应用程序之间共享单独一个地址空间也并不是没有优点的，尽管这种优点只有一个：我们不必实现任何形式的进程之间通信，因为所有信息都可以立即得到。这就是在16位计算环境里，诸如GetInstanceData（）以及hPrevInstance这样的函数和标识符如此流行的原因。除了允许信息块在相关任务间简便地进行传输以外，16位环境的共享内存还意味着简单地使用GMEM_DDESHARE标志就可以分配一个内存块。返回的句柄可能已经从一个任务传递到另外一个里去了，这里根本就没有任何限制。

由于只存在一个地址空间，我们还能得到另外一种有趣的推论。一个任务里计算出来的内存位置可以由另外一个应用程序访问，这时只需要指定相同的地址单元就可以了。因此，我们利用指针可以把句柄从一个任务传递给另外一个，只需要对它们的引用进行正确的更新，从而保证它们的有效性就可以了

对单独一个地址空间进行共享在Win32里已经不再能够接受了。在一个地址空间里分配的内存对象对于其他进程来说是完全“看不见”的，除非这是一个共享内存块。相同的规则亦可应用于指针。这就意味着在Win32里，我们不再允许对另外一个地址空间建立的窗口进行子类处理。对于这种技术，我们将在第15章“Windows高级技术”里进行技术性的探讨。

13.5 数据拷贝

把一个内存块传递给另外一个不同进程的最简单途径就是利用WM_COPYDATA消息。这是一条崭新的Win32消息，利用它可以绕过独立地址空间引入的某些限制和障碍。

WM_COPYDATA	0x004A
wParam	发送窗口使用的句柄
lParam	一个COPYDATASTRUCT数据结构的地址

在wParam里，我们可以存储发送窗口的句柄，而lParam则指向一个COPYDATASTRUCT数据结构的地址，这个结构的样子如下所示：

```
typedef struct tagCOPYDATASTRUCT
 { // cds
    DWORD dwData;
    DWORD cbData;
    PVOID lpData;
} COPYDATASTRUCT;
```

其中，dwData 项内包含了由应用程序定义的一个 32 位数值，这个数值会传递给接收窗口。实际的数据块是用 lpData 指定的，数据块的尺寸信息则存储于 cbData 内。以我们前面的假设为基础，lpData 内存储的数据千万不能包含指向其他内存位置的指针，因为这样的一个值在接收方的地址空间里没有任何含义。

下面这个代码段向大家阐述了 WM_COPYDATA 的用法，以及为什么一条消息只能通过 SendMessage () 发送出去：

```
...
case MN_SENDDATA:
{
    COPYDATASTRUCT cds;
    RECT rc;

    // get the window position
    GetWindowRect(hwnd, &rc);

    cds.dwData = 10;
    cds.cbData = sizeof(RECT);
    cds.lpData = (PVOID)&rc;

    // send the data
    SendMessage(hwndFind, WM_COPYDATA, (WPARAM)hwnd,
        (LPARAM)&cds);

}
    break;
...
```

上面这个代码段摘录自 SENDER 示范程序（该程序可以在本书附带 CD 的 Listing 13.1 里找到）。其中，发送方应用程序会取得自己在屏幕上的实际位置，定义一个偏移地址，然后通过一条 WM_COPYDATA 消息把这些数据传送给目标窗口。其中，hwndFind 窗口句柄指定的是目标窗口，那个窗口已经通过以前的一次 FindWindow () 查找标识出来了。在

TARGET 示范程序里（参考本书附带 CD 的 Listing 13.2），这种信息将在应用程序主窗口进程的 WM_COPYDATA 条件下进行拦截。随后，程序马上利用这种信息对源窗口下面的窗口进行重新定位，只是在 X 和 Y 轴上有一些微小的偏移而已。

```
...
case WM_COPYDATA:
{
    LPRECT prc;
    COPYDATASTRUCT *pcds = (COPYDATASTRUCT *)lParam;
    HWND hwndSender;

    // sender
    hwndSender = (HWND)wParam;
    prc = pcds -> lpData;
    SetWindowPos(hwnd, HWND_BOTTOM,
                 prc->left+pcds->dwData,prc->top+pcds->dwData
                 prc->right-prc->left,prc->bottom-prc ->top,0);
}
    break;
...
```

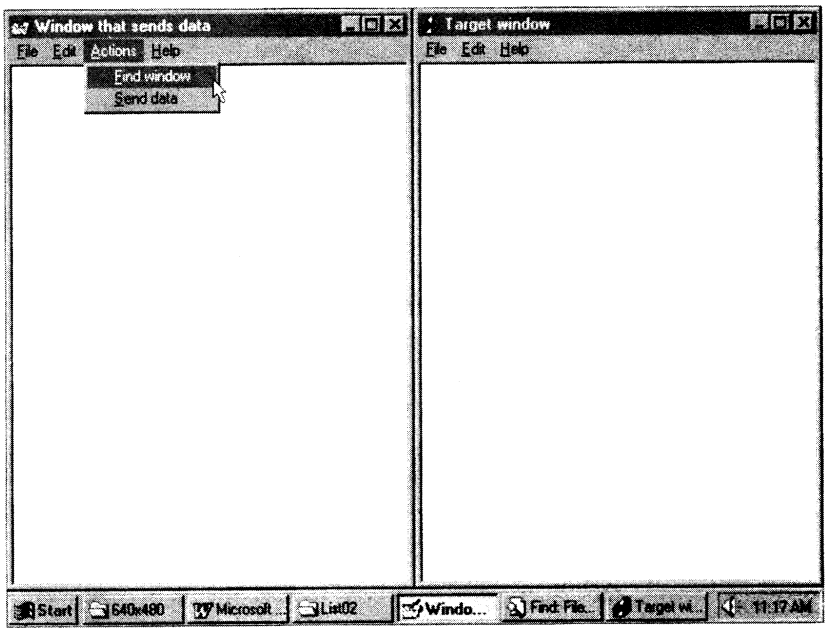


图 13-3　左侧的发送方窗口已经通过选择 Find Window 菜单项定位了目标窗口

在图 13-3 里，大家可以看到在屏幕上不同位置显示出来的源窗口和目标窗口。

假如选择 Send data 菜单项，发送方窗口在内部就会调用 SendMessage（）函数传递一条 WM_COPYDATA 消息，其中存储了关于窗口大小和位置的信息。目标窗口使用这种信息重新定义自己的位置以及大小，如图 13-4 所示。

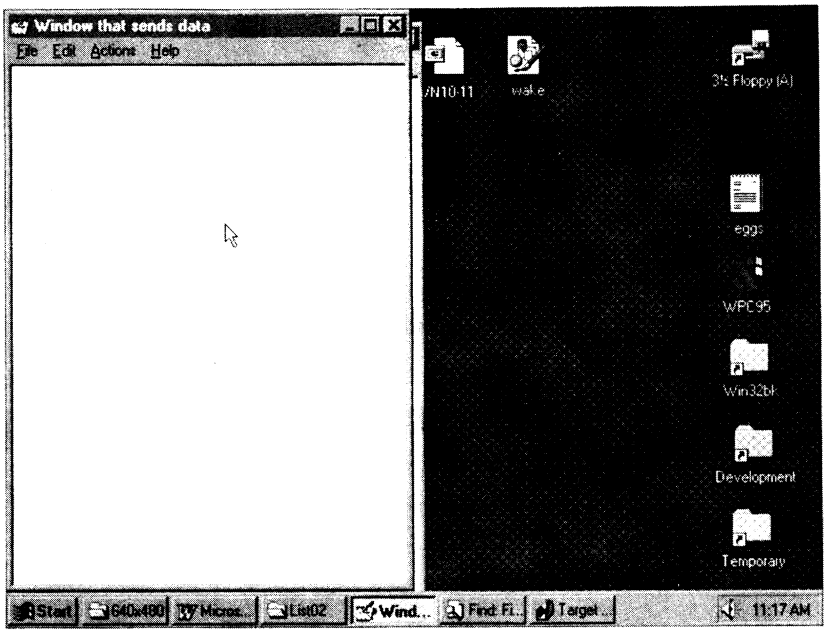


图 13-4　根据 WM_COPYDATA 消息传递的信息改变了自己的位置和大小以后，目标窗口现在几乎完全隐藏到源窗口的下面去了

发送 WM_COPYDATA 消息的时候，系统会在两个进程之间分配一个共享内存块，把有关的数据拷贝到那个区域，然后再发出这条消息。接收方对 WM_COPYDATA 消息进行了拦截以后，就会读取相应的数据。在这个时候，我们应该把数据拷贝到一个本地缓冲区内，因为 DefWindowProc（）函数对这条消息的处理完毕以后，就会同时消除这种缓冲区。

13.6　内存映射文件

在 Win32 里对内存进行共享要按照一种特殊的方式进行。例如，如果我们专门去寻找能对内存共享进行处理的一个特定的 API 函数，那么一个也不会找到。在这儿，我们必须掌握关于“内存映射文件”的概念。内存映射文件是指虚拟内存里一系列预约好的页，它们可以在需要的时候进行委托。这些页物理性地存储了由驻留于文件系统内的某个文件占据的页。因此，文件映射的作用就是把一个物理文件与进程虚拟地址空间内的某些页关联起来。考虑到我们不能对虚拟地址空间进行更多的处理，所以整个文件映象或者一个更小的部分可以映射到 RAM 里，并且可以通过一个标准的指针进行访问。从本质上看，对内存进行共享的时候，文件映射可以指示系统参考指定的文件或者分页文件，从而复原某个进程一定区域的虚拟地址空间。

Win32 可以使用内存映射文件来完成三种任务：

▶载入和执行一个应用程序

▶简便地读取一个文件，或者向文件内写入，同时不进行任何实际的 I/O 操作

▶在进程间对内存进行共享

现在让我们把注意力放在最后一点上面——跨越进程的边界限制，对内存进行共享。

13.6.1 在进程边界之间共享内存

这是一种多步骤操作，其中至少需要涉及到四个 API 函数：CreateFileMapping（），MapViewOfFile（），UnmapViewOfFile（）以及 CloseHandle（）。CreateFileMapping（）可以在虚拟地址空间里创建文件映射对象，并且根据自己参数的设定，从而预约一定数量的连续内存页：

```
#include〈winbase.h〉
HANDLE CreateFileMapping(HANDLE hFile,
                         LPSECURITY_ATTRIBUTES
                         lpFileMappingAttributes,
                         DWORD flProtect,
                         DWORD dwMaximumSizeHigh,
                         DWORD dwMaximumSizeLow,
                         LPCTSTR lpName);
```

参数	说明
HANDLE hFile	假如调用函数的目的是建立一个共享内存块，则设定为 0xFFFFFFFF
LPSECURITY_ATTRIBUTES	lpFileMappingAttributes 在 Windows 95 里设定为 NULL
DWORD flProtect	一个页面保护标志（在 Win95 里设置成 PAGE_READWRITE 或者 PAGE_READONLY）以及一个段标志(在 Win95 里设置成 SEC_COMMIT 或者 SEC_RESERVE)
DWORD dwMaximumSizeHigh	共享内存区域的高 32 位，必须设置成 0
DWORD dwMaximumSizeLow	一个 32 位数值，用于指定共享内存的大小
LPCTSTR lpName	可选的名称，用于在不同的进程之间标识共享内存块
返回值	在正文里讨论
HANDLE	文件映射对象的句柄，假如函数调用失败，则返回一个 NULL 值

Win32 载入一个进程或者一个 DLL，从而准备执行它们的时候，总共需要涉及到三个步骤。CreateFileMapping（）这个 API 函数实际只是三个步骤的中间那个。在这个过程中，CreateFile（）是第一个需要调用的 API 函数，然后是 CreateFileMapping（），最后用

MapViewOfFile（）函数结束整个操作。事实上，第一个参数肯定应该设置成一个文件句柄，这是从函数的名字就推断出来的。特别地，文件映射需要把文件的内容与某个进程虚拟地址空间的一个部分关联起来。对共享内存进行处理的时候，CreateFileMapping（）这个 API 函数会限制自己预约（或者委托）进程地址空间里的一系列页，同时要求分页文件为它返回某些物理内存页。在这种情况下，文件句柄就用一个 0xFFFFFFFF 常数取代了，它表明我们定义了一个共享内存区域。利用 CreateFileMapping（）载入一个执行模块或者一个数据文件，从而进行相应的执行或者访问操作时，第一个参数就是一个真正的文件句柄。

这个函数的语法具有很明显的 NT 趋向性。除了第一个参数以外，第二个参数是一个 SECURITY_ATTRIBUTES 数据结构的地址。这种数据结构在 Windows 95 里还没有实现，因此一般都把它设置成 NULL。保护标志定义了虚拟内存管理模块对分配页进行控制的方式。通常情况下，对于一个共享内存块来说，正确的组合是 PAGE_READWRITE，从而同时取得读取和写入权限。

共享内存块的大小是用两个参数定义的，每个都是一个 32 位宽的数值。一般对它的第一印象就这个数值太大了！然而这是必须的，因为三十二个二进制位足以定义一个 4GB 的内存对象——Win32 理论上能够控制的最大值。在现实的应用环境中，共享内存块必须小于这个上限值，因为操作系统本身需要在 RAM 里占据一定的空间。在 CreateFileMapping（）里使用 64 位值并不是针对共享内存块而设计的，而是专门针对 NT 文件系统设计的。NTFS 支持长达 18EB 的文件，这可是一个 100 亿亿大的天文数字！纯粹从理论上说，一个 NT 应用程序可以执行或者在系统内存里载入一个大于 4GB 的文件，这需要一个更大的位数才能对其进行描述。NTFS 在 Windows 95 里没有得到支持，而且在最近的一段时期以内微软也不会在 Win95 里支持它。

最后那个参数是分配给共享内存区域的一个可选名字。我们并不一定要为共享内存区域分配一个唯一的名字，但是假如另外一个进程希望访问相同的内存区域，这种信息就显得相当有用了，因此随时设置它不失为一种明智之举。这样一来，为了生成一个 4096 字节大小的共享内存区域，我们可以像下面这个代码段那样调用 CreateFileMapping（）函数来实现：

```
...
HANDLE hFileMap;
...
hFileMap = CreateFileMapping(0xFFFFFFFF, NULL, PAGE_READWRITE,
   0, 4096,
 "TWENY");
...
```

假如 hFileMap 不是 NULL，就意味着函数调用成功了，我们现在可以对一个名为 TWENY 的共享内存区域进行访问。请注意，对于 TWENY 共享内存的建立，以及对于访问以前生成的一个名为 TWENY 的共享内存区来说，相同的语法都是适用的。

作为 Windows 编程的一种习惯，我们应该把句柄转换成一个更友好的指针。这种工作可以通过 MapViewOfFile（）函数来完成：

```
#include 〈winbase.h〉
LPVOID MapViewOfFile(HANDLE hFileMappingObject,
                     DWORD dwDesiredAccess,
                     DWORD dwFileOffsetHigh,
                     DWORD dwFileOffsetLow,
                     DWORD dwNumberOfBytesToMap);
```

参数	说明
HANDLE hFileMappingObject	一个有效的文件映射对象的句柄
DWORD dwDesiredAccess	表 13-1 内列出的其中一个访问标志
DWORD dwFileOffsetHigh	视图范围
DWORD dwFileOffsetLow	视图范围
DWORD dwNumberOfBytesToMap	准备映射的文件的长度。假如为零值，就表明对整个文件进行映射
返回值	在正文里讨论
LPVOID	映射文件的起始地址，假如函数调用失败，则返回一个 NULL 值

MapViewOfFile()可以让一个应用程序对某个文件的某一部分进行映射，这只需要指定文件映射的起始位置和结束位置就可以了。这个 API 函数也提供了在 Win32 里的双重身份。对标准内存对象进行处理的时候，它的基本用途就是把文件映射句柄转换成一个指针，同时对这个分页文件涉及到的所有内存页进行委托。在另外一种身份下，它用一个视图(view)来指定物理性存在于文件系统内的某个文件，而返回的指针则允许应用程序对那文件进行读写操作，就好象该文件已经载入内存里一样，这样时间和空间都得到了节省。

访问标志(参考表 13-1)必须与上一次调用 CreateFileMapping()时分配的保护标志和段标志匹配。视图范围几乎肯定是指共享内存块的总体尺寸。这个 API 函数提供了机会来定义不同于第一个字节的一个起点，这个起点是把两个 32 位数值组合到一起得到的。与 NTFS 文件系统相关的相同考虑在这儿亦可适用。把这两个参数设置成 0 以后，就相当于指出自己准备对整个共享内存区域进行访问。

表 13-1　对一个文件映射对象进行映射的时候用到的访问标志

访问标志	说　明
FILE_MAP_WRITE	允许对映射文件的视图进行读写访问。前提是必须在建立映射文件的时候使用了 PAGE_READWRITE 保护标志
FILE_MAP_READ	允许对映射文件的视图进行只读访问。前提是必须在建立映射文件的时候使用了 PAGE_READWRITE 或者 PAGE_READ 保护标志
FILE_MAP_ALL_ACCESS	与 FILE_MAP_WRITE 效果相同

从本质上看，共享内存块的建立需要涉及到两个步骤。首先，我们必须利用CreateFileMapping()函数建立一个映射文件，用它指定内存块的大小。这种操作在虚拟地址空间里预约了一系列连续的内存页。其次，我们还要调用 MapViewOfFile()来访问那些内存页。内存管理模块会用内存里的一些物理页对虚拟页进行备份。

1. 建立一个通用内存区域

共享内存块也许是两个独立进程之间进行进程间通信(IPC)的一种最简单的形式了。我们根本没有必要建立一个内存映射文件，从而对同一进程不同线程之间的信息进行共享，因为线程可以自动访问进程地址空间里声明的每一个全局标识符。因此，为了理解如何建立一个通用内存区域，我们既可以同时需要两个进程，也可以让同一个程序执行两遍。在本书附带 CD 的 Listing 13.3 里，大家可以找到一个名为 MAZE 的示范程序，这个程序向我们演示了同一程序执行两遍的情况。

MAZE 是由 Kevin P. Welch 编写的一个示范程序，这个程序是六年或者七年前在《微软系统杂志》上出现的。该程序的宗旨是展示如何通过“动态数据交换”(DDE)协议实现进程间通信。在那个年代里，DDE 是当时最高级的一种技术，利用它可以实现 Windows 应用程序之间的通信。

MAZE 后面的基本原理是相当简单的。我们必须启动程序的多份拷贝，使其按照一定的顺序共享屏幕空间，如图 13-5 所示，这样就构成了应用程序的多个实例。每个 MAZE 实例都会尝试与其他实例建立 DDE 会话，从而构成了把所有程序都连接起来的一张网。在最开始的时候，只有第一个实例才会显示出在客户区内乱撞的一个小球。这个球在屏幕上显示了大量尾迹，它们几乎把整个客户区都隐藏到背景里去了。

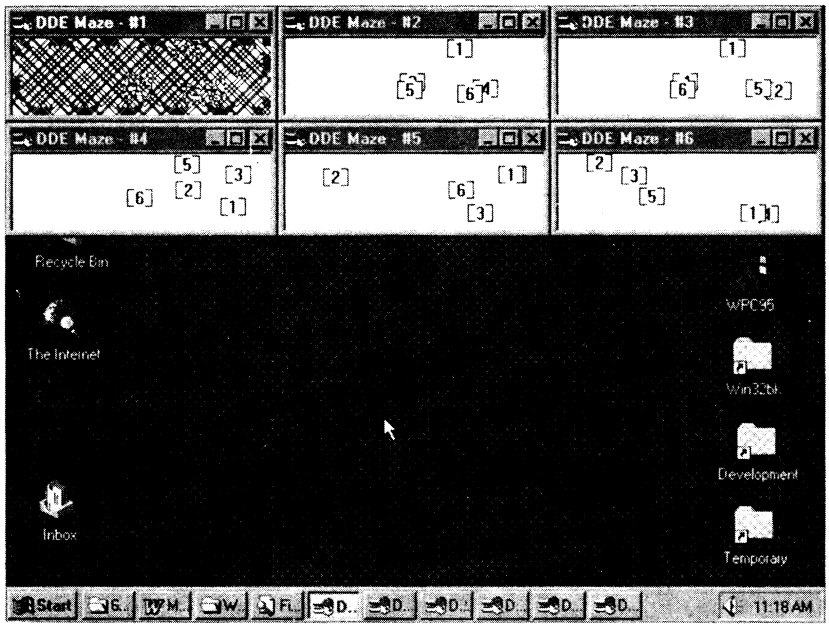


图 13-5 MAZE 的六份拷贝同时运行，所有拷贝都通过它们之间的 DDE 会话连接起来

通过对其他五个运行实例的客户区进行检查，我们会注意下面这种现象。每个实例都提供

了五个一组的数字，这些数字分别封闭在一对方括号里，它们代表了一系列球洞。编号为#2的实例里包含了五个球洞，编号从1到6，只是没有第2号球洞。编号为#3的实例里也包含了五个球洞，编号从1到6，只是没有第3号。以此类推。从中我们可以总结出这样一条规律：每个实例都提供了与其他实例有关的所有球洞，只是没有自己的。这些球洞在MAZE程序里派了什么用场呢？答案在它们的系统菜单上。假如在特定的实例里选择Grab the ball菜单项，就意味着在所有运行的实例里“打开了那个球洞”，这种事件是在一个黑色的“洞”里用一个白色的数字显示出来的。我们可以在实例#2上面做这种实验，就像图13-6显示的那样。

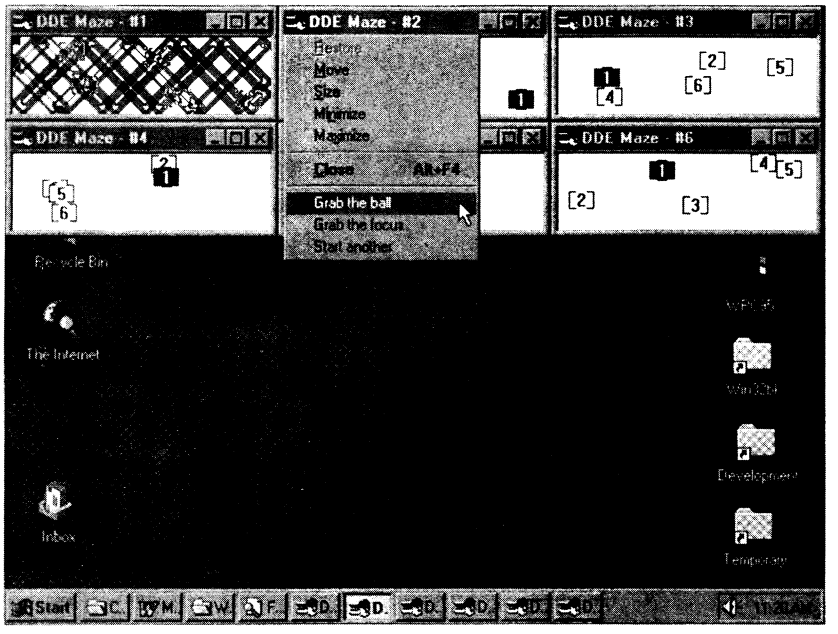


图13-6　对MAZE第二个实例里的球进行捕获

不一会儿，撞动的球跳进了实例#1的#2球洞，从而落入了一个DDE虚拟通道，然后在实例#2的客户区里浮现出来了，如图13-7所示。所有这些操作都是通过DDE链接完成的，这种链接把正在运行的所有实例都连接到了一起。假如在其他实例上面重复相同的操作，我们就能把相应的球洞变黑，最后造成的交通拥挤现象可以与每天在纽约市街道上面发生的情况媲美！球也会不断地撞击，并且落入不同的球洞里，然后又在其他某个地方浮现出来。

正如我们事先提到的那样，MAZE的最初目标是展示如何利用DDE的强大功能与其他应用程序进行通信。在这些例子里，MAZE的所有实例都通过DDE对信息进行共享，这些信息包括球的位置、应该画多少个洞以及哪个洞是用黑色显示的等等。但是其中的一些数据也可以输出到第三方应用程序里。图13-8向大家展示了MAZE如何与MS Excel 7.0 for Windows 95进行交互影响。这是通过传递两种数据来实现的：球在客户区撞击的时间以及客户区的大小。

图13-8显示的三维饼图可以动态地进行更新，从而反映出电子表格第一栏内发生的变化，这一栏内存储了相应的时间信息。MAZE最初的16位版本要明显依赖于Win16的内部特性才能运行——所有任务都共享相同的地址空间。MAZE的第二个实例之所以能知道存在以

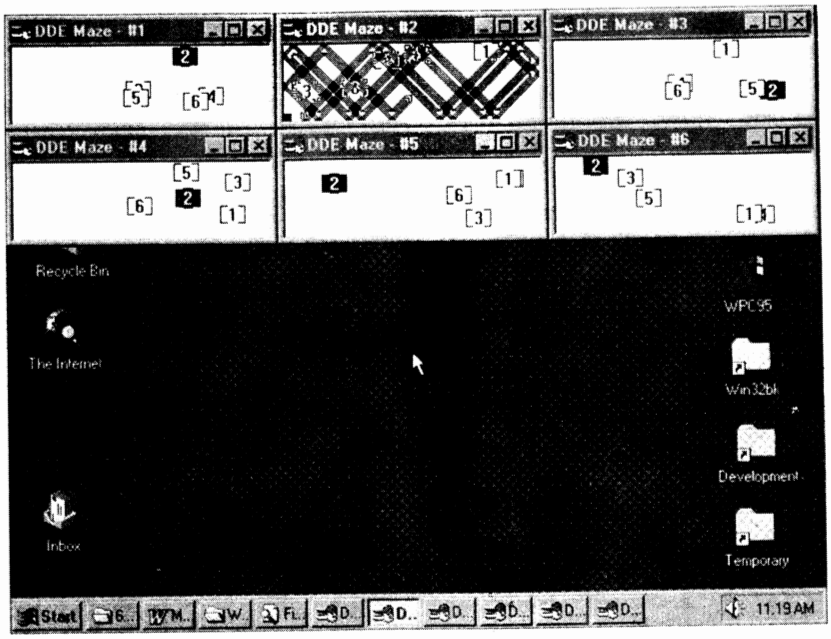


图 13-7　撞击的球现在转移到了 MAZE #2 上

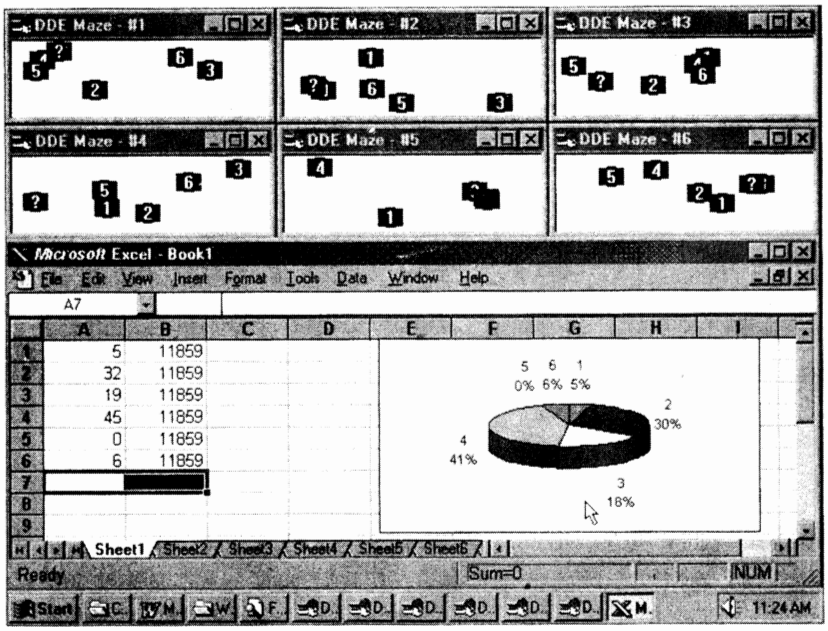


图 13-8　Excel 7.0 从 MAZE 里收集了一些信息，并且用
一个三维饼图对其进行表示

前运行的一个实例，是由于使用了 hPrevInstance 参数，并且通过 GetIntanceData()函数从它的数据段里取得了一些相关的信息。但是由于采用了新的内存管理规则，所以这些策略在

Win32 里就根本行不通了。

对于一个新进程来说，访问前一个实例数据段是非常方便的，因为它可以立即取回应用程序内定义的两个数据结构里的一些有用信息。构成这种数据结构的信息包括实例引用计数、实际建立的链接的数目、窗口的位置和大小以及其他信息。整个 MAZE 程序的运行策略都是以这种信息为基础的；假如要把代码转移到 Win32 环境下，就必须考虑一些选择方案，使其能够满足以前存在的应用程序逻辑，同时又不对信息流进行太多的改变。

对代码进行检查的时候，我们必须把注意力放在两个方面：

▶定义一种机制，从而简化对 MAZE 运行实例的列举

▶把所有信息都存储于一个公用的位置，使所有实例都能对其进行访问

为了解决第一个问题，我们需要用到信号机。信号机是一种进程间通信对象，它的行为好比一个计数器(进程间通信是下一章的主题)。信号机的信息以及与实例间行动协调有关的其他信息都存储于一个新的数据结构内，如下所示：

```
typedef struct _IPC
{
    HANDLE hsem;
    HANDLE hmem;
    void * pmem;
    int iSemCnt;
} IPC, * PIPC;
```

注册了窗口类以后，MAZE 紧接着调用 API 函数 CreateSemaphore () 建立和打开了信号机，如下所示：

```
// creating/accessing the semaphore
ipc.hsem = CreateSemaphore (NULL, 0, MAZE_ROWS * MAZE_COLS, "
MAZESEM");
```

正如我们以前指出的那样，信号机是一种可以在几个进程中进行引用的对象，它的计数值是通过 API 函数 ReleaseSemaphore () 来增大的，如下所示：

```
// counter increased
if (! ReleaseSemaphore (ipc.hsem, 1, &ipc.iSemCnt))
    MessagesBeep (0);
```

其中，ipc.iSemCnt 就扮演了 MAZE 实例计数器的角色。第一个实例执行以后，ReleaseSemaphore () 就把这个项的值设置成 0，以后每增加一个实例，这个值就相应地增 1。接下来，我们该准备建立共享内存块，用它来存储与两个应用程序数据结构——MAZE 和 LINKS 相关的数据，如下所示：

```
typedef struct
{
    int wNum;                    /* maze window number */
```

```
    int wLinks;              /* number of maze links */
    int wWidth;              /* width of maze client area */
    int wHeight;             /* height of maze client area */
    int bInitiate;           /* in initiate flag */
    BOOL bGrabBall;          /* grab the ball flag */
    BOOL bGrabFocus;         /* grab the focus flag */
    int wGoingAway;          /* maze going away counter */
} MAZE, *PMAZE;
```

MAZE 结构内包含的信息涉及到 MAZE 实例编号、当前活动链接的数目、客户区大小以及一些布尔值，这些布尔值指定了与撞球活动有关的当前状态。

```
typedef struct
{
    HWND hwnd;               /* client window handle */
    int wNum;                /* client window number */
    RECT rHole;              /* client hole position */
    BOOL bAdviseBall;        /* advise client of ball flag */
    BOOL bAdviseStat;        /* advise client of stats flag */
} LINK, *PLINK;
```

对于 MAZE 实例之间的每一个链接来说，都使用了一个 LINK 数据结构存储与球洞位置以及窗口句柄等等有关的数据。共享内存块的尺寸应该足够大，使其能容下一个 MAZE 结构，另外还要加上最多 15 个 LINK 结构，等于 MAZE_ROWS 乘以 MAZE_COLS，如下所示：

```
// creating/accessing the shared memory block
ipc.hmem = CreateFileMapping ( (HANDLE) 0xFFFFFFFF,
                               NULL,
                               PAGE_READWRITE,
                               0,
                               sizeof (MAZE) + MAZE_ROWS *
                               MAZE_COLS * sizeof (LINK),
                               " MAZEMEM");
```

共享内存区域的标签名为 MAZEMEM，返回的句柄存储到 ipc.hmem 项内。在第一次执行进程的时候，或者在以后的实例里对内存区域进行访问的时候，CreateFileMapping () 这个 API 函数能够很好地建立起内存区域。PAGE_READWRITE 保护标志使程序获得了对这个区域进行读写的权限，对于这样的一种 IPC 机制来说，一般都应该使用这个标志。

这两种准备工作（访问应用程序信号机和共享内存区域）不会与 MAZE 的标准逻辑产生

冲突。对于它的标准逻辑来说，首先是要建立一个应用程序窗口，然后在最后一个参数里传递 IPC 数据结构的地址。在 WM_CREATE 消息块里，应用程序会把 IPC 数据结构的地址存储在一个 PIPC 静态指针里，然后通过 MapViewOfFile（）函数对共享区域进行映射：

```
// accessing the IPC data structure
pipc = (PIPC) ( (LPCREATESTRUCT) lParam) -> lpCreateParams;

// accessing the shared memory block
pmem = MapViewOfFile (pipc -> hmem,
                      FILE_MAP_ALL_ACCESS,
                      0, 0, 0);

// storing the shared memory block pointer in IPC
pipc -> pmem = pmem;
```

三个连续的零值表明整个内存块都准备映射，并且取得对那个区域的任何访问权限。WM_CREATE 消息的处理会继续计算出在屏幕上的正确实例位置，同时对系统菜单进行更新（如图 13-9所示），然后用 MAZE 的所有现成实例对一个 DDE 会话进行初始化处理。

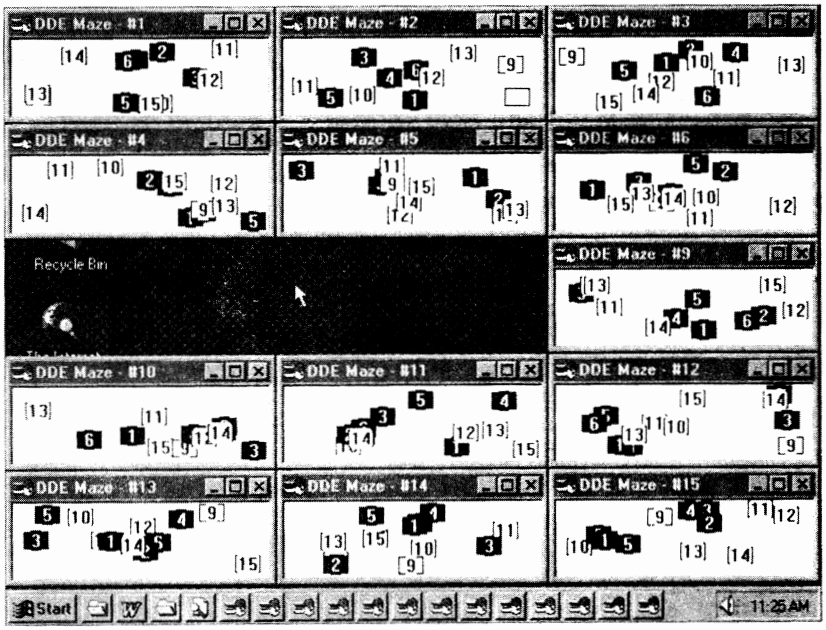


图 13-9　MAZE 在系统菜单里增加了三个菜单项。F2 加速键是作为一个热键实现的，不是作为一个标准的 ACCELERATORS（加速键）资源实现的

一旦完成了这些准备工作以后，MAZE 程序就会用与 MAZE 和 LINK 结构相关的更新信息填写共享内存区域，这种操作是通过对 CopyMemory（）的调用来完成的，如下所示：

```
// copying data to the shared block
CopyMemory(pMaze, &Maze, sizeof(Maze));
// copying data to the shared block
CopyMemory(pLink, Link, sizeof(LINK) * Maze.wLinks);
```

经过所有这些准备步骤以后，MAZE 就会利用数量众多的传递消息对球的位置以及统计信息进行更新，从而实现几个 DDE 会话链接。图 13-10 向大家展示了同时运行几个 MAZE 实例时的 DDE 传输情况。

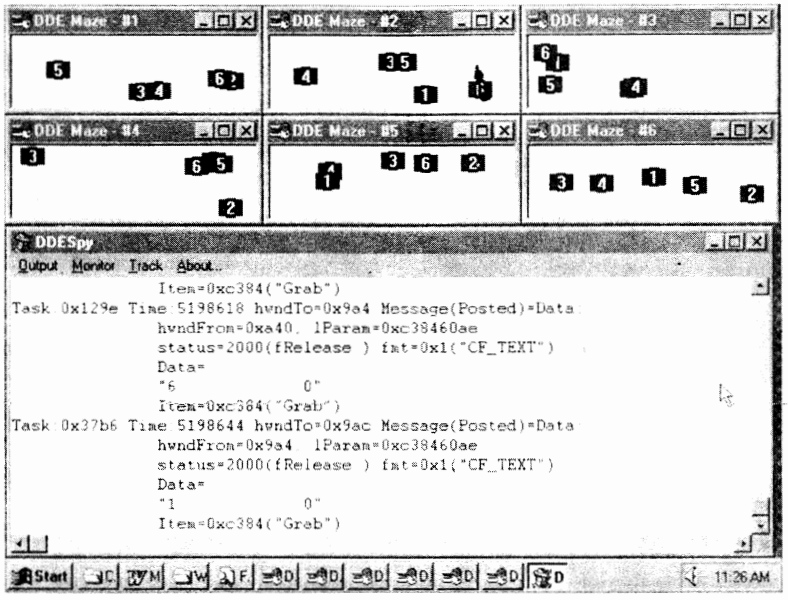


图 13-10　DDESPY 可以对 MAZE 实例进行的所有 DDE 发送和传递行动进行跟踪

2. 与共享内存有关的其他问题

Windows 95 把所有共享内存块都分配在 1GB 的内存区域以内，这个区域的地址起始于 0x8000000，终止于 0xBFFFFFFF。系统内运行的所有 Win32 进程都可以对这个区域进行利用。换言之，这个区域对于所有进程来说都是“可见”的。除此以外，这个 1GB 区域存放的所有对象在每个应用程序里都提供了相同的物理地址。如图 13-11 所示，其中的两份 MS Visual C++环境拷贝正在执行 MAZE 的两个实例。

对一个共享内存区内的信息进行存储和访问是迄今为止在两个进程间通信的最有效的方式，然而这种方式也存在一些缺点。首先，无论怎样对共享内存区内的信息进行构造和组织，它总是保持一种非公共的通信形式。什么是“非公共的通信形式”呢？它在这儿的含义是指在共享内存区域里，只有同一个人或者同一个开发小组编写出来的程序才是可读、可写的。假如程序开发的一个目标是与第三方的软件产品进行通信，共享内存就失去了它所有的价值和

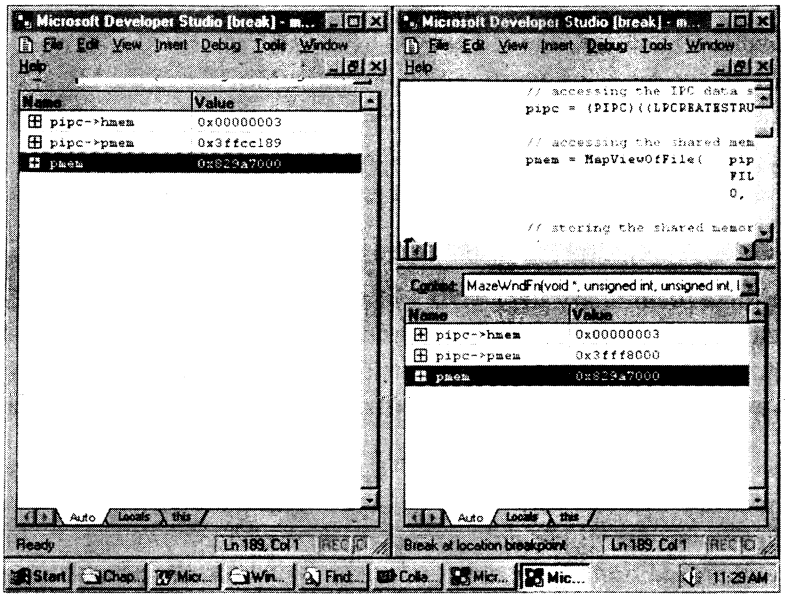


图 13-11　MAZE 的两个实例在调试模式下面停止了，在与共享内存块相关的 pmem 指针里显示出了完全一致的地址值

优点。在这种情况下，我们不得不改为使用标准的 IPC（进程间通信）机制，比如 DDE 或者“远程进程调用”（RPC）、OLE（对象嵌入与链接）、WinSockets 以及其他机制。

除此以外，共享内存的关键问题在于需要判断共享对象的名字，这种判断需要依靠一个不同的进程来实现，而不是依靠建立这个对象的那个进程。这个问题在 MAZE 程序里不会碰到，因为它只不过是相同的代码多次执行而已。通过某种方式，每个 MAZE 实例都“与自己交谈”，因为这类似一种“单向”会话。更为常见的情况则是：一个共享内存区域要由两个不同的进程访问，一个是建立它的进程，一个则是以后打开它的进程。

这儿的问题在于第二个进程怎样才能知道共享内存区域的名字?只有知道了这个名字,它才能对其进行访问。根据我自己的经验来看，有两种办法可以解决这个问题。最简单的方案就是确保两个进程在应用程序源代码里都进行了相应的编码，从而事先列出了共享内存区域的名字。这就意味着第二个进程可以顺利地访问通用的内存区域，但这同时也引入了另外一个有趣的问题：两个应用程序到底扮演了什么角色呢？在这样的一种情况下，我们可以认为某个进程扮演了“服务器”的角色，而另外一个进程则相当于一台“客户机”。这种类比是很准确的，并且是严格按照通信原本来进行表述的。在这种情况下，“客户机”那端将通过调用 OpenFileMapping（）来访问共享内存，而不是 CreateFileMapping（），如下所示：

```
#include〈winbase.h〉
HANDLE OpenFileMapping (DWORD dwDesiredAccess,
                        BOOL bInheritHandle,
                        LPCTSTR lpName);
```

参数	说明
DWORD dwDesiredAccess	一个 FILE_MAP_xxx 标志
BOOL bInheritHandle	假如为 TRUE，就表明返回句柄可以在建立一个新进程的时候进行继承
LPCTSTR lpName	共享内存区域的名字
返回值	在正文里讨论
HANDLE	内存映射文件的句柄；假如函数调用失败，则返回一个 NULL 值

除了上面这种方法以外，我们另外还可以为两个进程间通信的动态实现提供方便之门，这需要依靠一种附加的 IPC 机制，从而把共享内存块传递给其他进程。这种方案在实践中不易管理，因为它根本没有提供任何真正的好处。

13.6.2 数据文件的访问

在继续学习下一个主题之前，让我们先来复习一下本书附带 CD Listing 1.2 里提供 INCLUDE 程序。图 13-12 为我们展示了 INCLUDE 的执行情况，它列出了包含于 WINDOWS.H 内的所有头文件——WINDOWS.H 是 Win32 编程环境中所有 .H 文件的“祖先”。

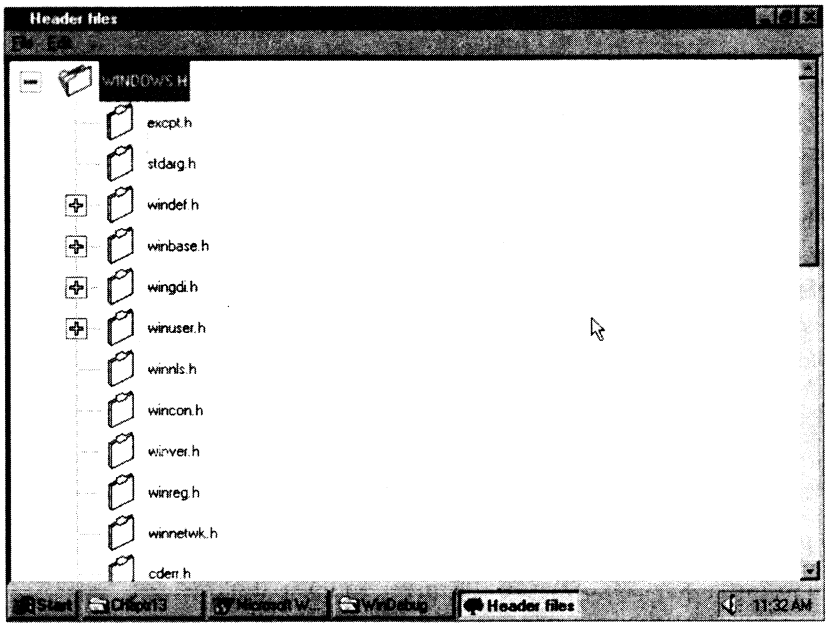


图 13-12 INCLUDE 用一个树形视窗显示出了包含的所有头文件

正如我们在第 2 章“Win32 开发工具”里指出的那样，应用程序预约了进程地址空间里的 20 个线性内存页，同时没有对其中的任何一个进行委托，如下所示：

```
# reserving 20 pages
pnode = (PNODE)VirtualAlloc(NULL, 20 * 4096, MEM_RESERVE, PAGE_
```

NOACCESS)；

用户选中一个目录准备进行浏览的时候，程序就会在第一个内存页内填写首先通过检查WINDOWS. H 而取得的信息，然后对其他头文件进行访问。这种尝试导致了经过明显设计的一次内存访问违规事件，这个事件会被 PageExceptionHandler（）例程捕获到。

FillTree（）例程可以对一个头文件进行浏览，从而搜寻其中的“#include”字符串，这种字符串指定了一个新的头文件的生成。INCLUDE 怎样读取 WINDOWS. H 以及其他包容文件的内容呢？这是通过内存映射文件来实现的。所以 INCLUDE 示范程序为我们提供了机会，使我们可以检查从一个现成的物理文件到建立一个内存映射文件之间经历的所有步骤进行检查。我们现在知道了目录（用户已经选中它了），也知道了准备扫描的第一个文件(WINDOWS. H)，所以可以开始继续下面的操作了。

FileTree（）例程可以接收到下面这五个参数：

▶pszInclFile，它指定准备打开的文件

▶hwndTree，树形视窗窗口的句柄

▶hTree，树形视窗当前的节点

▶pnode，指向一个应用程序定义的数据结构的指针

▶pszInclude，目录路径名

```
PNODE FillTree (char * pszInclFile
                HWND hwndTree,
                HTREEITEM hTree,
                PNODE pnode,
                char * pszInclude);
```

建立一个内存映射文件的时候，第一个步骤是通过 CreateFile（）打开文件。我知道这个函数的名字看起来不大合适，但是它的确是建立或者打开这样一个文件的最佳工具：

```
#include 〈winbase. h〉
HANDLE CreateFile (LPCTSTR lpFileName,
                   DWORD dwDesiredAccess,
                   DWORD dwShareMode,
                   LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                   DWORD dwCreationDistribution,
                   DWORD dwFlagAndAttributes,
                   HANDLE hTemplateFile);
```

参数	**说明**
LPCTSTR lpFileName	准备建立或者打开的那个文件的名字
DWORD dwDesiredAccess	零、GENERIC_READ 或者 GENERIC_WRITE
DWORD dwShareMode	零、FILE_SHARE_READ 或者 FILE_SHARE_

	WRITE
LPSECURITY_ATTRIBUTES	lpSecurityAttributes
	在 Windows 95 里设置成 NULL
DWORD dwCreationDistribution	表 13-2 里列出的某个标志
DWORD dwFlagAndAttributes	一个或者多个文件标志
HANDLE hTemplateFile	提供扩展属性的一个模板文件的句柄
返回值	在正文里讨论
HANDLE	文件句柄，假如函数调用失败，则返回 INVALID_HANDLE_VALUE

我们不准备对这个函数的语法进行细致的讨论，因为这儿的目标只是讨论与内存映射文件建立进程有关的信息。

表 13-2 CreateFile () 里的 dwCreationDistribution 参数使用的标志

标 志	说 明
CREATE_NEW	建立一个新文件。假如指定的文件已经存在，函数调用就会失败
CREATE_ALWAYS	建立一个新文件。假如指定的文件已经存在，就覆盖这个文件
OPEN_EXISTING	打开文件。假如指定的文件不存在，函数调用就会失败
OPEN_ALWAYS	打开文件。假如指定的文件不存在，函数就建立这个文件。这相当于把 dwCreationDistribution 设置成 CREATE_NEW
TRUNCATE_EXISTING	打开文件。一旦成功地打开，就对这个文件进行裁剪，使它的长度变成零字节。呼叫进程至少都要用 GENERIC_WRITE 访问权限来打开文件。假如文件不存在，该函数的调用就会失败

在 INCLUDE 这个示范程序里，我们知道被访问的所有文件都是存在的，并且准备用读写权限来打开它们。这儿的 GENERIC_WRITE 属性可以省略，因为我们只对快速浏览每个头文件感兴趣，不准备对其进行具体的修改。

在 Win32 里，有一个特性是它强制性地采用 GENERIC_WRITE 标志，我个人认为这是一种错误。我们还可以选择另外一种算法来判断当前打开的头文件里是否存储一个包容文件，这样就可以避开必须使用那个标志的问题：

```
// opening the file in pszInclFile
hFile = CreateFile(pszInclFile,
                   GENERIC_READ |GENERIC_WRITE,
                   0,
                   NULL,
                   OPEN_EXISTING,
                       FILE_ATTRIBUTE_NORMAL | FILE_FLAG_
                           SEQUENTIAL_SCAN,
                   NULL);
```

```
if(hFile == INVALID_HANDLE_VALUE)
{
    return FALSE;
}
```

完成了这种准备工作以后，我们可以通过调用 GetFileSize () 函数对整体的文件尺寸进行查询，从而立刻在内存里建立一个内存映射文件，如下所示：

```
// get the file size
dwSize = GetFileSize(hFile, NULL);
...
// create file mapping
hFileMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE,
                             0, dwSize, NULL);
```

返回值既可能是一个有效的文件映射句柄，也可能是一个 NULL 值（函数调用失败的时候）。假如我们通过了这种测试，剩下的唯一步骤就是把文件映射到进程地址空间里，并且把句柄转换成一个指针。MapViewOfFile () 可以帮助我们完成这种操作：

```
// access the file
pFile = MapViewOfFile(hFileMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);
```

这个函数在内部调用了 VirtualAlloc ()，从而对需要的内存页进行了委托，然后返回一个有效的指针。在 INCLUDE 程序里，从第一个字节到最后一个字节的整个文件都会进行映射。pFile 是用于对“#include”串进行扫描的一个指针。怎样对此进行编码呢？我的第一次尝试是使用一个普通的 strstr () 函数，从而节省时间和工作量。最后得到的结果却是一系列访问违规事件，一个接一个，颇为壮观。这儿的问题与 MapViewOfFile () 的实施有关。我对页进行了集中，并且没有在映射文件的最后一个字节后面放置任何记号。因此，strstr () 就无法判断终点在哪儿，导致最后访问的是非法位置。

接下来，我采取的方案是在内存映射文件的末尾放置一个“\O”字符，在建立它的时候强制使用 GENERIC_WRITE 标志。

```
* (pFile + dwSize) = '\O';
```

另外一种办法则需要建立一个函数，用它不断检查由内存映射文件指针指定的当前位置是否已经超出了文件长度的限制。不幸的是空字节没有用 NULL 来填写。这种搜索是以 pFile 指针为基础的，不需要任何真正的 I/O 操作。这就是利用内存映射文件来访问一个现成文件的好处。

数据的修改

需要澄清的最后一点与我们对内存映射文件进行的改变有关，以及这种改变会以什么形式反映在驻留于系统内的文件上面。读写操作是由操作系统控制的。尽管读操作并不会给我们带来任何新的问题，但是写操作却是根据一种缓冲 I/O 算法来实现的，这是由于考虑到需要提高系统整体的性能。通常情况下，对物理文件的改变是在通过 UnmapViewOfFile () 撤

消了映射文件的映射以后才注册的。作为另外一种选择，我们也可以通过调用FlushViewOfFile（）函数舍弃这种标准的行为，从而强迫字节流写入物理文件：

```
#include〈winbase.h〉
BOOL FlushViewOfFile（LPCVOID lpBaseAddress，DWORD
                    dwNumberOfBytesToFlush）；
```

参数	说明
LPCVOID lpBaseAddress	准备拷贝到磁盘上的一系列字节的起始地址
DWORD dwNumberOfBytesToFlush	准备送出的字节数
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个TRUE值；假如失败，则返回一个FALSE值

13.6.3 内存映射文件的释放

我们最好清除访问一个内存映射文件的时候建立的所有句柄和指针。UnMapViewOfFile（）能够接受的唯一参数就是以前映射的基准地址，并且返回一个布尔值，从而指出这种操作的输出情况，如下所示：

```
#include〈winbase.h〉
BOOL UnMapViewOfFile（LPVOID lpBaseAddress）；
```

随后，我们必须连续两次调用CloseHandle（），从而分别关闭文件映射句柄以及物理文件。请大家参考从INCLUDE.C里摘录下来的这段程序代码：

```
// unmapping
UnmapViewOfFile(pFile)；

// removing the mapping file
CloseHandle(hFileMap)；

// closing the include file
CloseHandle(hFile)；
```

清除一个共享内存块的时候，我们只需要调用UnmapViewOfFile（）以及一个单一的CloseHandle（）即可，从而对当前的环境进行正确的控制。

13.6.4 关于页边界更多的问题

INCLUDE程序总共预约了20个页，但只有在十分必要的情况下才一次委托一个页。假如出现了访问违规事件，程序就会通过PageExceptionHandler（）例程来解决这个问题：

```
LONG PageExceptionHandler(PNODE pmem，DWORD dwExCode，int iArea)
{
    MEMORY_BASIC_INFORMATION mbi；
```

```
    static PNODE p;

    // save the block initial location
    if(! p)
            p = pmem;

    // get page info
    VirtualQuery(pmem, &mbi, sizeof(mbi));
    if(mbi.Protect == PAGE_READONLY)
            return EXCEPTION_EXECUTE_HANDLER;

    if(dwExCode ! = EXCEPTION_ACCESS_VIOLATION)
            return EXCEPTION_CONTINUE_SEARCH;

    // should we commit the page?
    if(mbi.State == MEM_RESERVE)
    {
            // commit
            VirtualAlloc(pmem, PAGESIZE, MEM_COMMIT,
                    PAGE_READWRITE);
    }
    else
    {
            int i;

            // determine the page to commit
            i = ((char * )pmem - (char * )p) / 4096 + 1;
            // commit the page immediately following
            VirtualAlloc((char * )p + 4096 * i, PAGESIZE, MEM_
                    COMMIT, PAGE_READWRITE);
    }
    // get page info
    VirtualQuery(pmem, &mbi, sizeof(mbi));
    if(mbi.State ! = MEM_COMMIT)
            MessageBeep(0);

    return EXCEPTION_CONTINUE_EXECUTION;
}
```

INCLUDE 在第一个委托页内存储了从 WINDOWS. H 开始的的浏览进程中侦查到的所有头文件的名称。这些名称是一些长度可变的正文串，每个串最多可以有 12 个字符（传统的 8+3 格式，另外加一个句点分隔符）。很有可能出现的情况是一个正文串超出了页边界，从而把其中的一些字节插入了尚未委托的内存页内。

这种情况会导致一系列无休止的访问违规事件，从而使应用程序陷入进退两难的困境。事实上，传递给 PageExceptionHandler () 例程的指针指向的是一个已委托位置，然而准备拷贝的正文串却大于从那个位置到页终点位置之间的距离，由此就产生了异常事件。发生这样的情况时，PageExceptionHandler () 就会侦测到这种错误，然后立即对紧靠在当前指针所指向的那个页后面的一个页进行委托。正如我们以前提到的那样，我们必须在自己的源代码里针对这种事件进行编码，从而捕获和修复这种问题。图 13-13 向我们展示了 INCLUDE 和一名特别聪明的用户之间的交互作用，那名用户同时展开了 WINDOWS. H 和 RPC. H 分支。

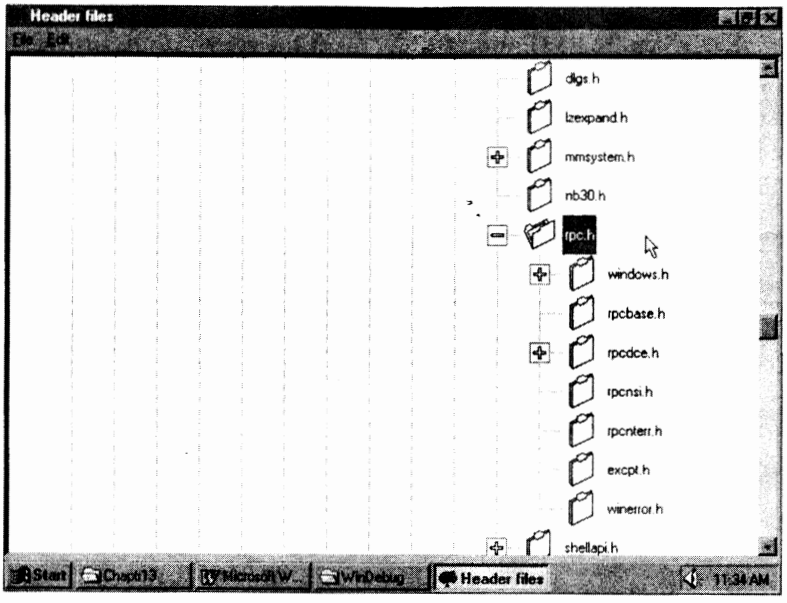


图 13-13　根据需要，INCLUDE 可以随时对附加的内存页进行委托。如图所示，其中的包容树已经被动态地展开了

13.7　虚拟内存、物理内存和页文件

为了加深我们对内存管理的讨论，我专门编制了 MEMORY 示范程序，这个程序可以在本书附带 CD 的 Listing 13.4 里找到。在图 13-14 里，大家可以看到 MEMORY 程序刚刚执行后的显示情况。

Actions 菜单内的 Starting System Monitor 菜单项（如图 13-15 所示）可以激活 SYSMON. EXE 应用程序，这是随同 Windows 95 提供的一种系统工具。在 Windows 95 的标准安装过程中，SYSMON 程序不会一起拷贝到硬盘上来，但是我希望应用程序的开发者们应该在自己的专用工具清单里列出这个程序。假如系统内还没有安装 System Monitor 程序，MEMORY 就会提醒使用者安装它，相应的显示情况如图 13-16 所示。

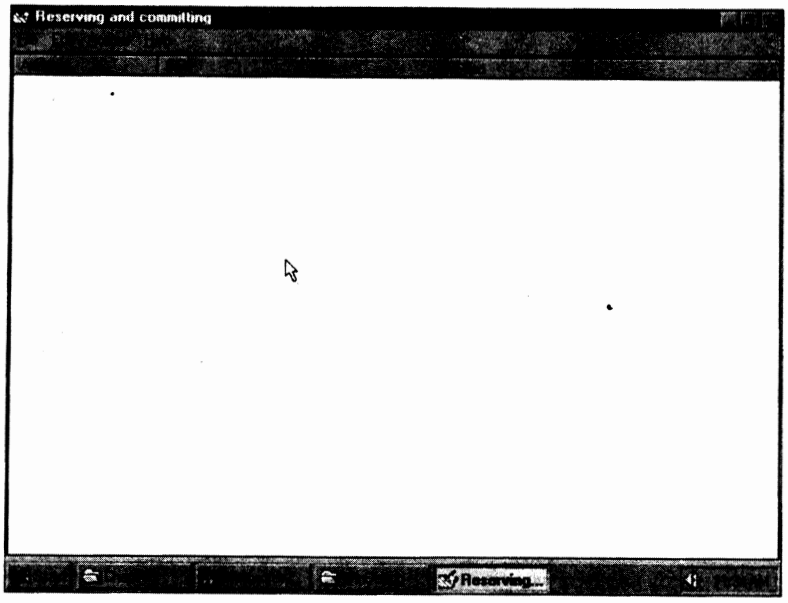


图 13-14　MEMORY 的客户区包含了一个列表视窗窗口，
其中将不断地填写关于预约和委托内存块的信息

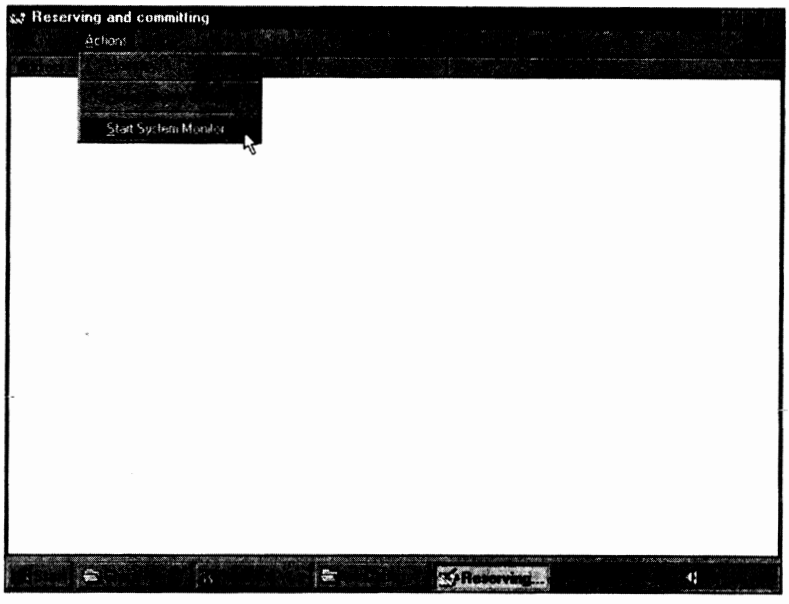


图 13-15　Starting Ssytem Monitor 菜单项可以对
某些系统参数进行监视

这时，我们可以选择 Windows Setup 卡片，然后依次选择 Accessories 项目和 Details 按钮。这时就能看到 System Monitor 列于其中，选择它，然后继续安装即可。经过这样的准备

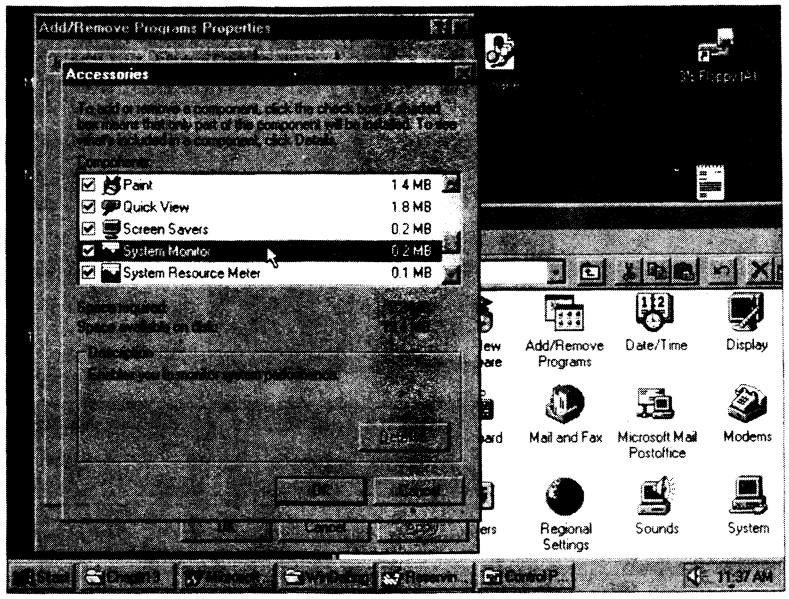


图 13-16　Add/Remove Programs 控制面板程序允许
你立刻安装 System Monitor

工作以后，以后就可以选择 Starting Ssytem Monitor 菜单项来执行相应的功能了。这时，MEMORY 和 SYSMON 这两个程序会同时占据显示屏幕，如图 13-17 所示。

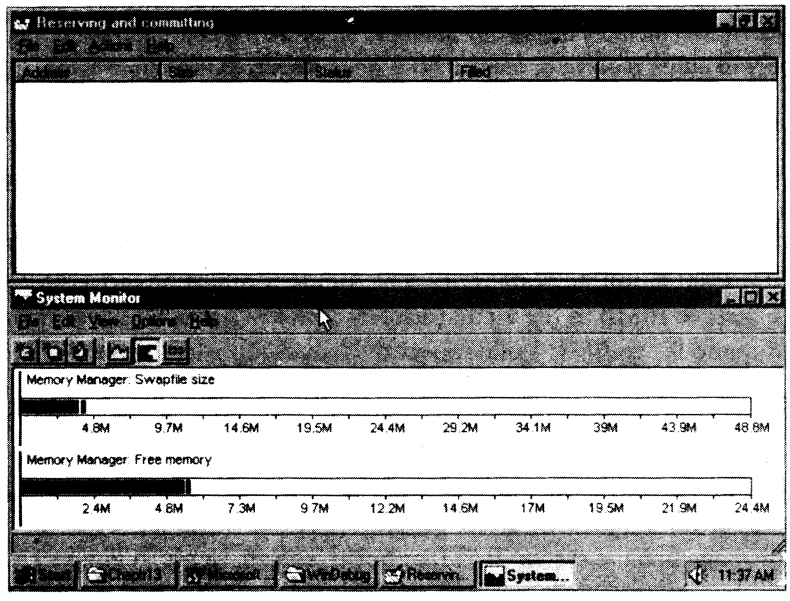


图 13-17　MEMORY 和 System Monitor 同时显示出来了，它们准备协同工作，
向我们展示内存管理模块的工作情况

正如我们通过图 13-17 看到的那样，System Monitor 正在显示的信息与三个参数有关，并且是用水平条显示出来的：

▶已分配的内存

▶空闲内存

▶交换文件尺寸

利用 System Monitor 里的 Edit 菜单，我们可以选择这些项目，也可以删除已经存在的项目。在这个时候，我们已经作好准备利用 MEMORY 开展工作了。利用这个程序能做什么事情呢？主要能做两件事情：在 Actions 顶级菜单里选择 Memory 菜单，从而对内存进行预约和分配。图 13-18 向我们展示了由这个扩展命令而引入的一个对话框。

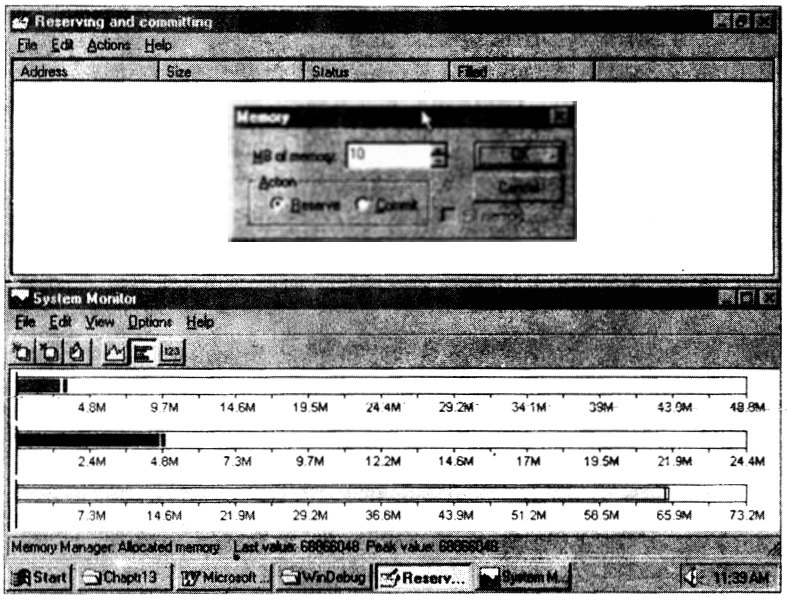


图 13-18　Memory 对话框为我们提供了一个“上下”控件，用于对准备预约或者委托的内存数量进行设置。显示值的单位是“兆字节”

System Monitor 向我们表明已有 65MB 已分配内存、13 兆空闲内存以及一个 4MB 左右的交换文件（由于我使用了屏幕抓图软件，所以大家看到的这幅图与实际的系统情况有些差别）。正如大家通过任务栏按钮看到的那样，我正在运行的几个应用程序都占据了数量极大的内存空间。MEMORY 里的 Memory 对话框准备在虚拟地址空间里预约 10MB 的内存。执行这个命令会得到什么后果呢？它会影响到由 System Monitor 报告的值吗？图 13-19 提供了这个令人迷惑的问题的答案。

正如我们预期的那样，在线性地址空间里预约内存不会影响 System Monitor 里正在监视的任何一个参数值。

现在让我们在不接触那些页的前提下委托 10MB 的内存空间。这儿的问题是要理解哪个物理内存页供应源将对这种请求进行响应：分页文件？系统 RAM？还是两者都要响应？答案可以在图 13-20 里找到。

分页文件的长度现在增大为 15MB 左右，而已分配内存则跳至接近 78MB 的地方。Win32

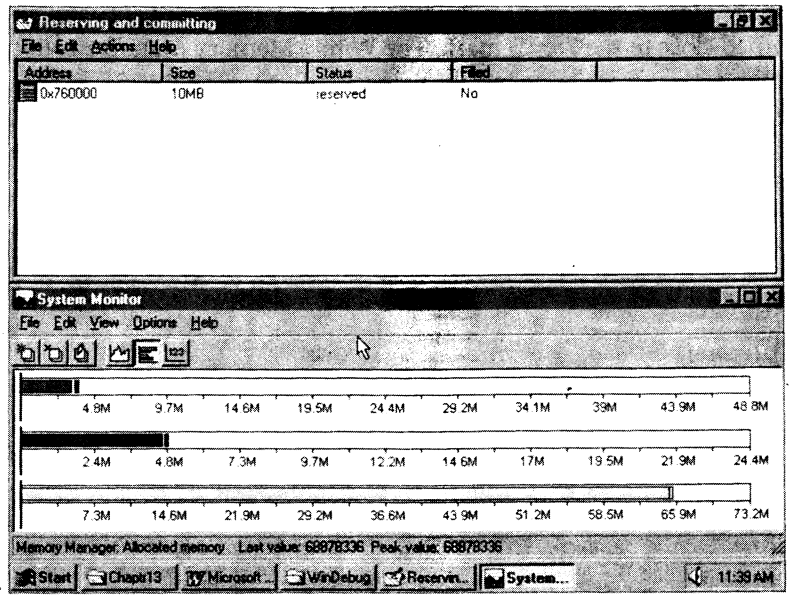


图 13-19　MEMROY 列出了预约的第一个 10MB 的内存块。System Monitor 里没有发生任何变化

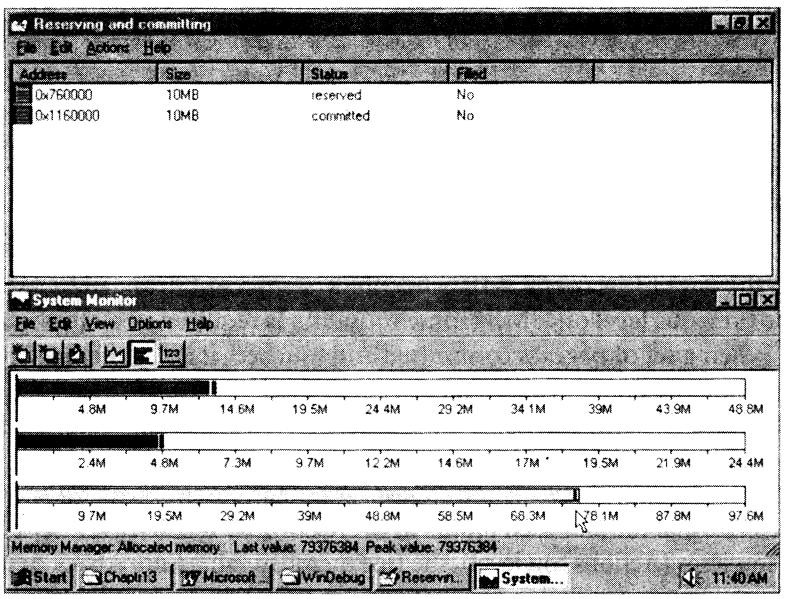


图 13-20　委托了 10MB 的内存块以后，System Monitor 里的水平栏发生了变化

内存管理模块从分页文件里得到了自己需要的空间，同时没有对 RAM 产生影响。假如不仅要对一些页进行委托，而且要在其中填写相应的数据，这种情况就会发生戏剧性的变化。这正

是我们在分配一个新的 10MB 内存块时看到的情况。如图 13-21 所示。

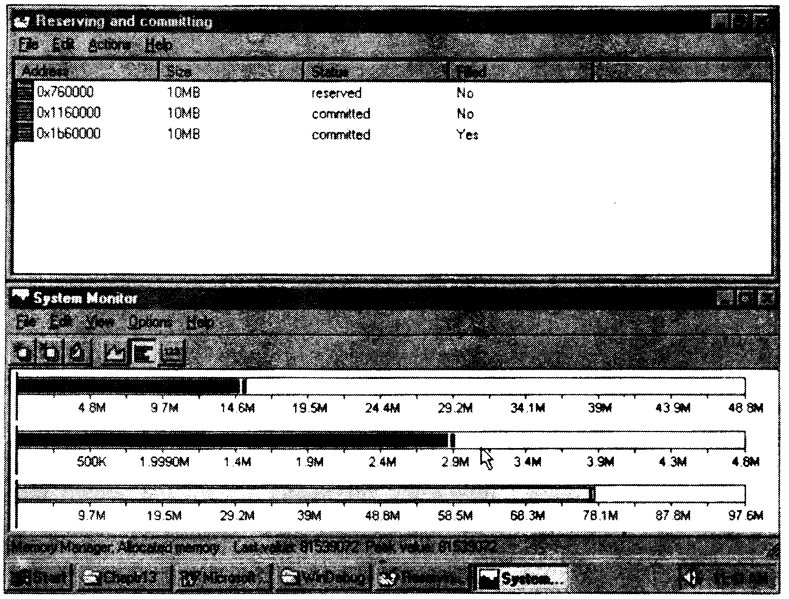


图 13-21　委托和填写了一个新的 10MB 内存块以后，System Monitor 报告空闲内存的显著减少（使用新的刻度），以及分页文件的一致性增长

现在，分页文件的长度超过了 23MB，而系统 RAM 则从以前 13MB 的级别减少至 3MB（所有这些计算都需要一些四舍五入）。委托并且填写了一系列页以后，是内存池对这种请求提供了响应，而不是分页文件。这并不意味着那些页必须永久性地驻留于内存里。请注意，与空闲内存相关的刻度在图 13-20 里已经发生了变化，它为我们留下了还有更多空闲内存的一个最初的假象。

在 MEMORY 程序里，假如使用了 Reset listview content 菜单项（在 Actions 顶级菜单内），空闲的内存值就会跳回到它的前一个级别。然而，分页文件的尺寸看起来好象在 22MB 左右阻塞住了。这一现象应该使我们回忆起这样一个事实：在 Windows 95 里，假如分页文件里包含的信息已经过时，并且不再发生作用，系统就会自动缩小分页文件的尺寸。Windows 95 可以在内部对这种行动进行管理，同时，它会在信息变成无用的那个时候起经历一段可观时间延迟。在图 13-22 里，大家可以看到已分配内存和空闲内存值都回到了它们以前的数字状态。

对页进行预约只会影响到虚拟地址空间，它对分页文件以及物理内存都不会产生影响。对某些页进行委托则一定需要某些物理性的响应，这种响应通常是由分页文件提供的。假如一个应用程序写入或者接触一个已委托页的时候，这个页就会立即进入系统 RAM，这在 MEMORY 示范程序里已经得到的证明。因此，只有在真正需要的时候才进行内存页的委托，并且在满足应用程序需求的前提下，把这种行动限制在数量尽量少的内存页上面，从而减小它对系统 RAM 的影响。

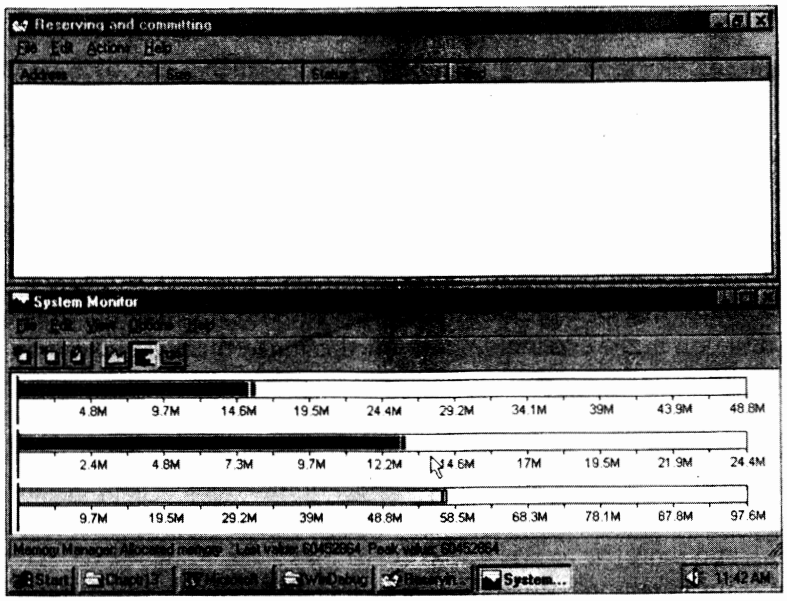


图 13-22　交换文件尺寸栏仍然显示了很高的数字，因为 Win32 内存管理模块在丢弃无用页之前要等待一段时间

13.8　动态链接库

在专门讨论 DLL 内存管理之前，让我们先来回顾处理 DLL 时碰到的一些重要概念。DLL（动态链接库）事实是我们编写的应用程序代码中的一部分，这些代码放置于一个不同的容器里，这个容器是一种独立于主 .EXE 程序的可执行模块。更准确地说，DLL 是一系列服务或者说函数的集合，其他应用程序的执行过程中可以对其进行调用。

Win32 API 实现了一系列 DLL，比如 KERNEL32，GDI32 以及 USER32，这些 DLL 都是相当常用的。根据这种模式，DLL 成了存放一些多功能函数的好地方。这些函数在同一时刻可以为多个应用程序提供服务，系统 API 就属于这种情况。

开发一个复杂的应用程序时，假如这个应用程序分割成了几个模块，我们就可以把一些常用的服务放置于一个或者多个 DLL 里，这些 DLL 有能力为每个模块提供相应的服务。从构造的角度来看，DLL 是一种 C 源文件，其中包含了几个函数，这些函数可能或者不可能相互之间进行交互作用。载入一个进程的地址空间以后，对 DLL 内包含的所有函数都很容易进行访问，它们与执行代码内编写的本地函数并没有什么两样。除此以外，我们能够从一个 DLL 内调用的 API 函数并没有什么限制，或者说能够实现的操作没有限制。例如，一个 DLL 里可以容纳诸如图标、对话框模板以及自己认为有必要的其他任何一种资源。可以利用它来注册一个新窗口类、建立窗口或者简单的提供一系列函数（算术例程、复杂的三维绘图算法以及其他），使这些东西与主程序成为一个整体，从而完成最终的应用程序代码编写。

尽管一些常规的功能相似，纯粹的可执行模块与 DLL 之间还是存在一些区别的，如下所示：

▶DLL 需要一个 DEF 文件

▶DLL 提供了一个 DllMain（）入口点，而不是传统的 WinMain（）函数

▶DLL 只能载入，不能执行

关于这些区别的详细分析，请大家继续下面的小节。

13.8.1 DLL 的 DEF 文件

正如我们曾经提到的那样，DLL 的本质是一个容纳函数的容器。这儿最重要的事实是每个这样的函数都必须从一个独立的模块内访问。这个模块要么是一个执行模块，要么就是另外一个 DLL。为了达到这样的目标，我们需要在 DLL 的 DEF 文件的 EXPORTS 段内填写所有输出函数的名字。如下所示，其中的 Services 就要用一个真正函数名取代的占位符：

```
...
EXPORTS
    Service1
    Service2
    Service3
    ...
```

举个例子来说，假如我们有机会对 USER32.DEF 进行检查，就可以看到下面这样的代码：

```
...
EXPORTS
    SetWindowPos
    CreateWindow
    MoveWindow
    ...
```

除此之外，传统的 NAME 目录在 DLL 的 DEF 文件里消失了，相同的位置由 LIBRARY 取代：

```
...
LIBRARY MYDLL
DESCRIPTION 'Sample DLL'
...
```

对于 DLL 的 DEF 文件来说，它的语法最终需要一条 SECTIONS 命令。实际上，这条命令可以在任何 DEF 文件里使用，并没有什么特别的限制。假如大家对第 2 章“Win32 开发工具”还有印象，那么也许还记得我们在那儿曾经提到过：SECTIONS 命令的作用是把一些特殊的数据和代码区域集合到一块儿，开发者准备按照与标准代码和数据对象不同的方式来对

待它们。在 SECTIONS 命令的下面，我们需要列出应用程序和 DLL 里使用的所有段名，每个段名后面还跟随了更多的属性信息，用于对它们的功能进行限制。除了常见的 READ，WRITE 和 EXECUTE 属性以外，最有趣的也许要算 SHARED 了。这种属性可以指示链接程序把当前段当作一个共享内存区域来对待，与 DLL 链接起来的所有进程都可以对其进行访问：

```
...
SECTIONS
    COMMONDATA SHARED
...
```

经过这样处理以后，COMMONDATA 区域就转换成了一个共享内存块。尽管这在 Win16 里是一种标准的属性，但在 Win32 里却不得不进行明确的定义。

13.8.2 DLL 入口点

DLL 入口点通常叫作 DllMain（），这个函数可以接受三个参数，如下所示：

```
BOOL WINAPI DllMain (HINSTANCE hinstDLL,
                     DWORD fdwReason,
                     LPVOID lpvReserved);
```

参数	说明
HINSTANCE hinstDLL	DLL 实例句柄，通常也被称为模块句柄
DWORD fdwReason	用于指定怎样对 DLL 进行调用的标志
LPVOID lpvReserved	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

与 Win16 不同，只有在 DLL 存在于进程地址空间的情况下，才有几种情况需要对这个入口点进行调用。事实上，在 Win32 里，DllMain（）同时肩负着入口和出口的重任。其中的 fdwReason 参数指定了当前调用的形式。表 13-3 为大家总结了 fdwReason 四个可能的值。

表 13-3 DLL 入口点四个可能的标志

标 志	值	说 明
DLL_PROCESS_ATTACH	1	一个 DLL 首次载入进程地址空间的时候传递
DLL_THREAD_ATTACH	2	无论什么时候，只要在与 DLL 相关的进程内建立了一个新线程，就传递这个标志
DLL_THREAD_DETACH	3	无论什么时候，只要明确中断了 DLL 相关进程里的一个线程，就传递这个标志
DLL_PROCESS_DETACH	0	假如 DLL 从一个进程地址空间分离开，就传递这个标志。这种分离是由于应用程序的中断或者明确调用 FreeLibrary（）造成的

因此，一个典型的 DllMain（）函数看起来应该象下面这个样子：

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL,
                    DWORD fdwReason,
                    LPVOID lpvReserved)
{
    switch(fdwReason)
    {
        case DLL_PROCESS_ATTACH:
        {
            ...
        }
            break;

        case DLL_PROCESS_DETACH:
        {
            ...
        }
            break;

        case DLL_THREAD_ATTACH:
        {
            ...
        }
            break;

        case DLL_THREAD_DETACH:
        {
            ...
        }
            break;
    }
    return TRUE;
}
```

假如初始化工作成功完成，DllMain () 就会返回一个 TRUE 值；假如由于其他什么原因导致了失败，则返回一个 FALSE。只有在设定了 DLL_PROCESS_ATTACH 标志的前提下调用这个入口点函数，返回的值具有真正的含义。其他情况下的返回值均可忽略。

假如进程与一个 DLL 依附在一起，就需要传递 DLLDLL_PROCESS_ATTACH 标志。只有当进程至少提供了对 DLL 所提供函数的一次调用，启动这个进程的时候才会与某个 DLL

依附起来。另外，也可以在一个进程线程里明确地调用 LoadLibrary () 函数，从而实际与 DLL 的连接。显然，我们并没有必要在 DLL_PROCESS_ATTACH 条件下放置任何代码，但这儿的确是对 DLL 进行初始化，以便日后使用的一个好地方。

注册一个新窗口类、分配一个内存块或者在注册表内存储某些信息，所有这些操作都是在接收到 DLL_PROCESS_ATTACH 标志时可以进行的有效行动。在另外一方面，假如进程中断或者某个进程线程调用 FreeLibrary () 函数以后，就会发生 DLL_PROCESS_DETACH 事件。在这儿，线程调用 FreeLibrary () 的目的是解除与 DLL 的依附关系，同时释放它在进程地址里占据的空间。这种情况特别适合于我们执行清除进程的时候。

线程标志提供了一个附加的工具，用它可以把一段代码与某个 DLL 关联在一起。某个进程建立了一个新线程的时候，就会发出 DLL_THREAD_ATTACH，它使我们有机会执行专门针对新线程和 DLL 而设计的一段代码。一个 DLL 首次载入进程地址空间的时候，这个标志不会传递给应用程序。类似地，DLL_THREAD_DETACH 会在某个进程线程中断的时候传递，从而使我们有机会执行一些清除操作。

除了 DllMain () 以外，我们还可以在 DLL 源代码内使用其他任何一种自己认为有用的函数。同时只输出那些希望在一个单独模块内能够访问的函数。

13.8.3 DLL 的装载

假如执行源代码提供了对 DLL 内包含的某个函数的至少一次调用，到那个时候，相应的 DLL 就会自动载入进程地址空间内。但这样的一个模块是不完整的，因为链接程序在希望对 DLL 函数进行调用的地方提供了一些伪地址（在链接期间，那些地址是无法使用的）。因此，一个 EXE 程序必须在内部对此进行修补，否则便无法正常执行，这种修补处理是由系统载入模块自动完成的。

用户执行某个程序的时候，系统载入模块会把自己的注意力放在 EXE 文件内的重定位记录上面。这些记录是由链接程序建立的，其中包含了一些准确的信息。利用这些信息，就可以唯一性地标识那些提供了服务的 DLL，这些 DLL 是在应用程序源代码内调用的。系统载入模块会在系统内存里寻找指定的 DLL，从而对行踪不明的地址进行分析。假如 DLL 不存在，就会对与 DLL 函数有关的所有地址进行分析，并由此中断 DLL 的激活阶段。

另外一种方法需要调用 LoadLibrary () 或者 LoadLibraryEx ()，从而在应用程序启动以后明确地载入一个 DLL 模块。其中，lpLibFileName 参数指定了准备载入的 DLL 的名字。

```
#include <winbase.h>
HINSTANCE LoadLibrary (LPCTSTR lpLibFileName);
```

假如调用成功，这个函数就会返回相应 DLL 的句柄。

在相同的程序里，通过对 LoadLibrary () 的多次调用，这种操作可以重复多次。这时会发生什么情况呢？显然，系统在进程地址空间里只保存了一段单一的 DLL 代码和数据。然而，针对每一次调用，系统都增大了 DLL 用度计数器的计数值。这种信息对于标识与 DLL 链接起来的进程总数是很有帮助的。

每调用 FreeLibrary () 一次，DLL 用度计数器的值就会减一，最终会变成零值。只有在用度计数器的计数值变成零的前提下，系统才会授权释放由 DLL 占据的内存空间。这种技术可以保证即使只有一个进程仍在使用 DLL，系统也会把相关的 DLL 代码和数据保存下来。

定义一种策略

对于载入 DLL 的两种方法来说，哪一种最好呢？这个问题很难回答，然而间接载入机制是迄今为止最流行的，也是应用得最广泛的。对于第二种技术——直接装载来说，它肯定要显得灵活一些，然而除此以外并没有其他什么优点。在这两种情况下，我们的源程序都应该知道调用的是哪种服务，以及需要传递的是哪个参数。唯一的区别在于根据运行期间的具体需要而载入 DLL 的时候（第二种方法），我们必须按照地址来调用每种服务，而不是按照名字来调用。

GetProcAddress () 函数可以返回一个 DLL 服务的地址，需要在其中传递的参数则包括 DLL 实例句柄以及函数名：

```
#include 〈winbase.h〉
FARPROC GetProcAddress (HMODULE hModule, LPCSTR lpProcname);
```

参数	说明
HMODULE hModule	DLL 实例句柄
LPCSTR lpProcname	函数名
返回值	在正文里讨论
FARPROC	函数地址；假如调用失败，则返回一个 NULL 值

一旦知道了函数的地址，并且知道了相应的语法以及参数的完整集合，那么调用相应的服务就很容易了。下面是按照地址调用 DefWindowProc () 函数的形式示范，注意它不是按照名字来调用的：

```
...
FARPROC pfn;
...
pfn = GetProcAddress (hModule, " USER");
 (*pfn) (hwnd, msg, wParam, lParam);
...
```

总而言之，预先载入 DLL 与根据当时的需要动态载入 DLL 的这两种方式之间的区别是很小的，并且主要与进程启动以后对 RAM 的影响有关。

现在有这样一个问题引起了我们的兴趣：DllMain () 函数是在 WinMain () 之前还是之后调用的呢？假如 DLL 是预先载入的，那么 DllMain () 就会首先执行，而 WinMain () 则尾随其后。除此以外，对 DllMain () 的访问是一个接一个连续进行的，这意味着它不能由两个线程同时执行。第一个线程对 DllMain () 进行访问的时候，第二个线程会一直保持挂起状态，直到第一个线程的初始工作进行完毕。因此，我们强烈建议大家不要在 DllMain () 里提供对 CreateThread () 的调用，从而避免谁都不希望看到的死锁现象。

13.8.4 DLL 内存管理

在本章列举的几种情况下，我们都曾经提到过一个 DLL 是在调用它的线程里执行的。这意味着 DLL 要依靠调用线程堆栈以及调用进程的地址空间。除此以外，由 DLL 代码分配的

每个内存对象都存储于调用进程的地址空间内。这种情形和 Win16 相比已经发生了戏剧性的变化。在 Win16 里，DLL 可以成功地用于进程间的信息共享。然而现在，每个 DLL 缺省时都在链接起来的每个进程地址空间内建立了一个数据块，我们通常把这种数据块称为“实例数据”(Instance data)。

假设进程 A 和 B 都要对 DLL X 进行访问，同时 DLL X 声明了一个源文件标识符，并将其初始化成 10。就像下面这个代码段那样：

```
int i = 10;
...
BOOL DllMain (...)
{
    ...
}
```

在这儿，i 对于进程 A 和 B 都是“可见”的吗？你说对了！它是同一个对象吗？非也！事实上，即使进程 A 把 i 设置成 12，它对于进程 B 来说仍然会保持最初的 10。在这个时候，我希望在座各位都清楚这是 Win32 里唯一可行的方案，其中每个进程都有它自己的进程地址空间。对内存块的运行期动态分配也是按照相同的规则实现的，从而保持了它们在不同进程间的完整性。

DLL 里的共享内存

为了实现进程间的内存共享，使其能对相同的 DLL 进行访问，需要用到我们已经熟知的内存映射文件技术。除此以外，我们还可以通过 #pragma_seg () 来建立一个通用的段，如下所示：

```
#pragma_seg (" Tweny")
int i = 10;
HWND hwnd = NULL;
#pragma_seg ()
```

新的 TWENY 段内包含了两个标识符，它们对于访问这个 DLL 的所有进程来说都是“可见”的。到此为止，我们的工作还只完成了一半。正如早先提到的那样，链接程序必须通过 DEF 文件里的 SECTIONS 命令来检查这种新段的存在：

```
...
SECTIONS
    TWENY SHARED
...
```

最后，我们还要注意：只有初始化的标识符才能放置到一个共享内存块内。

第 14 章　多线程、IPC 和 I/O

Windows 95 是一种多任务、多线程的操作系统，我们到现在为止应该对这个概念有一个很清楚的认识。在第一章“Win32 中的软件开发”里，大家首次接触了与这些主题有关的一些概念。在本章内，我们将把注意力集中在如何设计和实现多线程代码上面。此外，还要讲述一个相关的问题：进程间通信（IPC）。

正如我们在本书一开头就讨论过的那样，开发多线程应用程序的主要宗旨是用尽量少的时间对用户的请求作出响应。在过去，即使一次简单的操作，比如对打印作业进行初始化，都意味着要在计算机屏幕前面等待一段漫长的时间，而且这段时间内根本没有机会做任何事情。Windows 95 的多任务特性只解决了这个问题的一部分，因为它只是简单地允许用户在运行的不同任务间切换，不用局限于在同一时刻只能做一件事情。

尽管操作系统具备这种多任务特性，但是系统调度模块仍然把应用程序当作一个统一的对象来对待。因此，调度一个对象进行处理的时候，那段时间内就只能完成一个任务。为了避免这种操作麻烦且功能有限的特性，我们应该在自己的代码内实现多线程机制。

从根本上说，线程是可由系统进行调度的一个最简单的代码单元。从程序开发的角度来看，线程则是一个普通的 C 函数，它与针对单线程环境编写的一个传统例程之间并没有什么分别。根据 Win32 的设计准则，线程是属于 WINAPI 类型的一个函数，它可以返回一个 LRESULT 值，并能接收一个长参数。下面这个代码段为我们展示了一个函数的开头部分，这个函数最终可以在执行过程中当作一个线程进行处理：

```
LRESULT WINAPI ThreadFunction (long lParam)
{
    ...
}
```

作为一个标准的 C 函数，我们可以为线程分配任何一个名字。参数是一个常规的 32 位数值，通常要把它当作指向某个内存位置的指针使用。这个内存位置里包含了准备传递给新线程的一系列数据。除了在应用程序内部声明的所有局部变量以外，一个线程可以对应用程序里的任何全局标识符进行访问。正是考虑到这个原因，所以强烈建议大家尽量不要使用全局数据，以便提高代码的整体可读性。与一个新进程进行通信的唯一途径就是长参数，这个参数是我们根据实际情况定制的。假如相同的函数要作为一个线程多次执行，这种方式就显得特别有用了。

举个例子来说，假设我们把一个 Win32 进程的打印功能封装到一个独立的进程里实现。通常情况下，尽管具有多线程本质，许多应用程序在同一时刻都只允许执行一个打印功能。通过运用更复杂的方式来编写代码，我们可以重新利用相同的线程，从而激活多个打印作业，使这些打印作业一个接一个地排列，排列多少并没有限制。因此，每次启动一个新的打印作业

后，就必须向线程传递不同的信息。这种信息的传递可以通过线程参数来完成。

编写源代码的时候，两个花括号之间并不限于只能放置一个线程代码块。声明代码范围标识符、调用其他函数或者建立新的进程等等都是允许的操作。作为一个普通的代码块，当碰到了一个“返回”(return) 关键字或者关闭花括号 (}) 的时候，进程就会自动中断。另外，调用另外一个 Win32 API 函数的时候，假如那个函数要求必须先中断调用进程，那么这个进程也会中断。因此，我们不能认为倘若某个线程从属的那个进程一直处于活动状态，那么相应的线程也会一直运行下去。

刚才描述的方案特别适用于单线程应用程序。在那种情况下，存在的唯一线程会一起保持运行状态。这个线程中断以后，整个进程也就失却了意义，所以也要停止运行。这样就引入了线程管理的一个关键特性：线程只能在另外一个运行着的线程里建立。源代码只能把一个多线程执行模块当作自己设计的一部分来实现，尽管这只有在程序运行的时候才能进行测试和验证。

通常情况下，多线程应用程序是由一个主线程和几个附加的线程构成的。在主线程里包含了所有的基本代码（比如 WinMain () 函数和主窗口类进程），附加线程则是在程序的执行过程中建立的。第二种线程通常专门用于完成某些特定的任务，比如打印文档、对一系列名字进行排序或者对数据库进行索引等等。这些任务完成以后，相应的线程要么中断，要么进入一种“昏睡”状态——这种状态是与完全挂起的状态比较而得到的——就像一个陷入沉睡的人，尽管活着，然而所有的生理功能都进入了最低程度。尽管如此，线程的建立和消除都是一种相当简便和快速的操作，往往仅需几个毫秒的时间就能完成。很少见的一种情况是：某个应用程序从头至尾都保持了几十个线程的同时运行状态。相反，线程的激活和屏蔽是动态进行的，只有在真正的需要的时候才对它们采取行动。不久以后，我会向大家展示当一个应用程序里同时运行了大量线程的时候，会对系统的整体性能产生什么样的影响。

14.1 线程的建立

从 C 语言的眼光来看，线程就是一个函数。因此，我们怎样才能把一个静态的代码段转换成能与 CPU 周期进行抗争的东西呢？答案就是 CreateThread ()！这个函数可以在极短的时间内建立并且有选择地激活一个线程。

```
#include 〈winbase.h〉
HANDLE CreateThread (LPSECURITY_ATTRIBUTES lpThreadAttributes,
                     DWORD dwStackSize,
                     LPTHREAD_START_ROUTINE lpStartAddress,
                     LPVOID lpParameter,
                     DWORD dwCreationFlags,
                     LPDWORD lpThreadId);
```

参数	说明
LPSECURITY_ATTRIBUTES lpThreadAttributes	指向一个安全属性描述符的指针。在 MS Windows 95 里要设置成

	NULL
DWORD dwStackSize	堆栈尺寸。假如把这个参数设置成0，就表明线程堆栈的尺寸等于主线程堆栈尺寸，这是位于进程地址空间里的1MB预约内存页。
LPTHREAD_START_ROUTINE lpStartAddress	准备当作一个线程实施的函数的名字
LPVOID lpParameter	传递给所建线程的信息
DWORD dwCreationFlags	既可以为0，也可以为CREATE_SUSPENDED，后者用于在新进程建立好后立即挂起
LPDWORD lpThreadId	一个DWORD标识符的地址，用于接收线程ID
返回值	在正文里讨论
HANDLE	线程句柄；假如函数调用失败，则返回一个NULL值

在MS Windows 95里，第一个参数肯定应该设置成NULL，这是由于Win95还没有实现安全信息。在WindowsNT里，这个参数可以设置成一个SECURITY_ATTRIBUTES数据结构的地址。指定一个堆栈尺寸基本上没有什么用处，因为这种信息是由系统自动管理的；如有必要，最终能达到1MB这个上限尺寸。因此，只有一在一些非常特殊的场合下才需要把这个参数设置成非零值(为零表明缺省的最大尺寸为1MB)，从而满足需要使用特别大堆栈的那些线程的需求。

第三个参数是准备转换成一个独立线程的函数的名字。正如前面看到的那样，这个线程函数会返回一个LRESULT，并且只接受一个长类型的参数值。建立过程与新进程之间的通信是通过第四个参数来完成的，这通常是指向某个内存位置的指针。从理论上说，它可以是应用程序内任何一个标识符的地址，然而其间涉及到了一些逻辑上的限制。

源文件范围标识符对于一个进程内的所有线程都是“可见”的，所以没有必要传递它们的地址。块范围标识符可以传递给新线程，然而这种操作需要我们进行一些附加的准备工作。

一个线程启动以后，它将完全独立于建立它的那个线程运行。对于这两种线程来说，我们不可能预测哪个会首先中断，因为它们的执行是根据各自的优先次序来进行的。因此，在这儿传递一个本地变量的地址是相当危险的，因为标识符也许永久都位于范围以外，这样就导致了在第二个线程内发生访问违规事件。从本质上说，唯一安全的方法是把标识符的地址声明为静态存储类，这样除非用于创建的那个进程中断，否则在其中就可以不断地使用下去。

总而言之，由于存在我们想象得出的多种组合，所以针对一个线程传递给另外一个线程的信息，我们几乎不可能定义出相当准确和普遍适用的一种情况。我们在这儿必须掌握一个要点：开发者并不是需要对自己的程序负责的唯一人士。以外，也不是只有开发者本人才能预测到自己的代码内真正会发生什么情况。所以，假如我们决定从一个次级线程向另外一个线程传递某些信息，就应该考虑实现两者之间的一些同步操作，从而防止出现可能的程序死

锁或者谁都不愿意看到的数据崩溃现象。

14.1.1 同步的实现

建立线程的第一种方法以及线程间的交互作用引入了针对多线程开发环境的一个关键性问题：线程间的同步。为了百分之百保证两个或者多个线程间进行正确的交互作用，唯一的办法就是实现某种形式的同步。根据以前的例子，第一个线程必须发出对WaitForSingleObject（）函数的一次调用，从而将自己中断，直到第二个线程中断为止。这样可以保证传递给第二个线程的信息在整个执行期间都能使用。一个线程中断的时候，它的句柄会发射出来，同时所有正在等待的函数（比如 WaitForSingleObject（）都不会返回，直到目标对象发出信号为止。

CreateThread（）的第五个参数与初始化标志对应。这个函数现在只提供了对建立标志CREATE_SUSPENDED 的支持，这个标志的作用是指示新线程保持自己的挂起状态，直到用恰当的线程句柄调用了 ResumeThread（）函数为止。如果不使用 CREATE_SUSPENDED 标志，一个新线程就会在执行了自己代码内的所有语句以后立即启动。

最后那个参数是一个双字类型的数据，用它可以接收线程 ID，这是在整个系统内适用的具有唯一性的标识编号。一个线程可以用这个 ID 以及 CreateThread（）的返回值（进程句柄）来进行标识。Win32 没有提供任何工具可以从一个线程 ID 切换到的句柄，或者逆向切换。这个现实是非常不幸的，因为它限制了与当前进程内线程的交互作用，也限制了与其他进程内线程的交互作用。

一旦建立好以后，线程就会完全独立于其他线程以外运行，除非我们用某些 IPC 工具使其同步。下面这个代码段向大家展示了线程函数的用法，以及如何建立和执行一个线程：

```
...
// creating second thread
if (! (hThread =
        CreateThread (NULL,
            0, (LPTHREAD_START_ROUTINE_SecondThread,
            (LPVOID) &trdata,
            0,
            &dwThreadID)))
{
    ...
}
...
LRESULT WINAPI SecondThread (LONG lParam)
{
    ...
}
...
```

14.1.2 建立一些准则

在深入讨论优先级以及同步技术之前，我们应该首先掌握与多线程开发模式有关的一些常规设计准则。心中应该有针对性地考虑到一些实质性的问题，比如："每个进程使用多少个线程比较合适?"、"在一个线程里应该放置什么代码?"以及"怎样针对多线程应用程序进行编码"等等。

正如我们在第一章"Win32 中的软件开发"里讨论的那样，线程只有在预先规划好的前提下才能有效地使用。首先，我们几乎不可能准确地指出一个常规 Win32 进程里将要用到的线程总数。单线程应用程序在 MS Windows 95 里也能良好地运行，而且几个补充线程的引入也要事先作好周密的计划。

在许多情况下，多线程应用程序的运行速度要比单线程的版本慢。为了对这种现象有个感性认识，我们可以对所有 MS Office 95 应用程序进行检查，从而了解微软的工程师们针对它们实现的线程数量。通过图 14-1 我们可以看出，MS Word 95 是一个单线程程序，或者至少表面上看起来如此。

WinWord 真的是一个单线程应用程序吗？对此我们不能完全通过 PVIEW95 程序来进行调查，因为它提供的只是 Word 执行过程中某一特定时刻的一幅静态图像而已。线程可以动态地建立，所以假如以后激活了某种特定的任务（比如打印），附加的线程就有可能出现。MS PowerPoint 采用了一种与众不同的方式，从而证明了尽管这些程序都来自同一家公司，但却采用了不同的设计思路。程序刚刚启动，PowerPoint 就提供了三个线程，其中有一个运行于非常低的优先级上面，如图 14-2 所示。

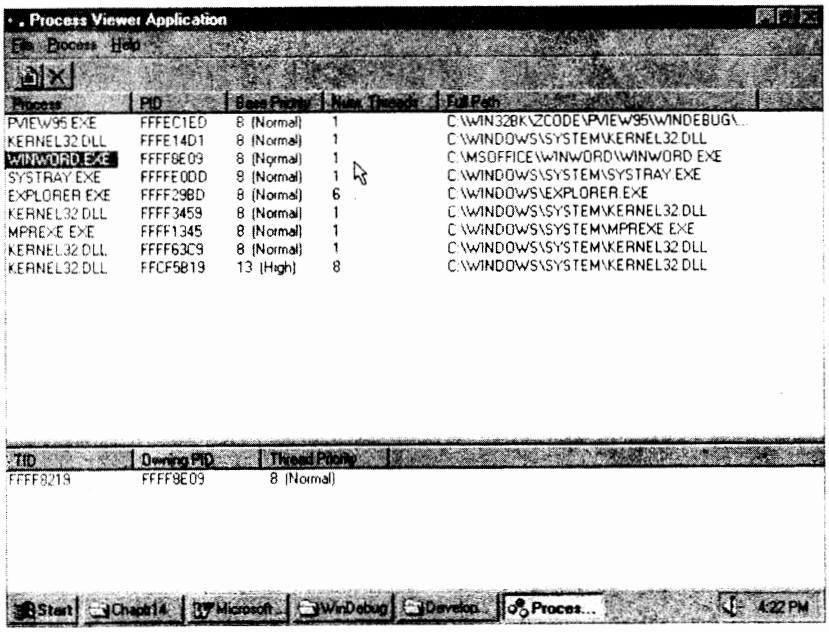


图 14-1　PVIEW95 只列出了 WINWORD.EXE 内运行的一个线程

MS Excel 又是怎样的呢？这个程序在启动以后也只建立了一个线程。相反，MS Access 会针对查询和打印生成各自独立的线程。从这四个例子里我们可以得出一条普遍适用的什么

规则呢？也许，针对不同的 Win32 应用程序我们无法得到一条标准的规则。每个应用程序几乎都代表了一个独立和唯一的领域，它们都有自己的规则和需求。

根据经验来看，一个 Win32 程序应该支持包括 WinMain () 和应用程序主窗口类进程在内的主线程。除此以外，所有看得见的窗口——无论叠置式窗口还是弹出式窗口——都应该属于相同的线程。这种方案极大简化了几种任务的实施，比如消息的发送、窗口同步以及性能调整等等。相反，假如强迫特定应用程序内的所有窗口都从属于单一的线程，那么就会对我们的程序设计思路造成限制。单独进程内建立的窗口允许在相同的源代码内存在多个输入队列。这就意味着尽管其他行动都是在某个独立线程内进行的，应用程序里仍然有可能至少提供了一个能随时对用户的请求进行响应的窗口。然而，正如我们在本章后面部分会讲述到的那样，相同的结果也可以通过一种不同的途径来得到。

除了主线程以外（这个主线程应该包含与应用程序 UI 部分有关的所有代码），其他线程也可以根据到达主窗口进程的特定消息来进行创建。举个例子来说，假设我们接收到了一条消息，它指示应用程序从磁盘上的某个文件里载入一幅位图，以便以后把它显示出来。与内存映射文件的建立有关的所有指令都可以方便地封装到单独的进程里，该进程可以通过一些 IPC 机制对读取操作的结果进行报告。这里可以有两种选择。结束既可以一开始就建立好，然后在应用程序启动以后把自己挂起，也可以在以后特别必要的时候再动态地建立。

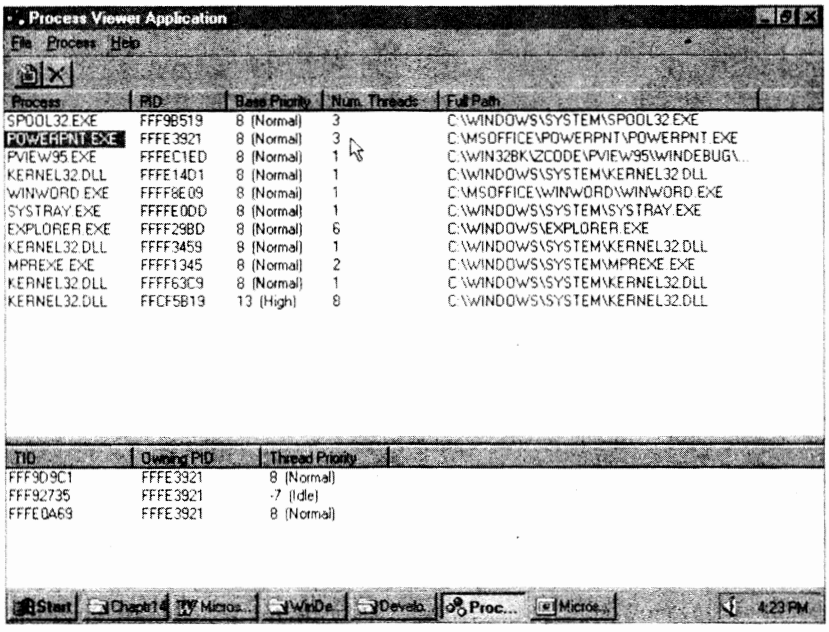


图 14-2　PVIEW95 列出了 MS PowerPoint 在启动以后所有活动的线程

假如采用第一种方案，那么建立的线程在整个进程的运行期间都将存在，只是在需要它的时候再挂起或者恢复。对于第二种方案来说，只有在绝对必要的时候才建立一个进程，然后要么立刻消除，要么挂起留待以后利用。这儿的想法是保证主线程总是处于存在以及活动状态，以便对所有的 UI 消息进行处理。这种方法也允许我们建立附加的线程，或者根据需求

恢复其活动状态。实际上，线程的基本工作就是执行后台任务或者操作，同时尽量避免对应用程序 UI 造成影响。

14.1.3 决定线程的数量

现在让我们尝试一下回答关于每个进程多少个线程的问题。这并没有一个准确的数字，并且每个程序在这方面都具有一定的特殊性，使用的线程数目也是不尽相同的。例如，载入一个简报的时候，PowerPoint 就会激活一个高优先级的线程，使其在第 10 级运行，如图 14-3 所示。

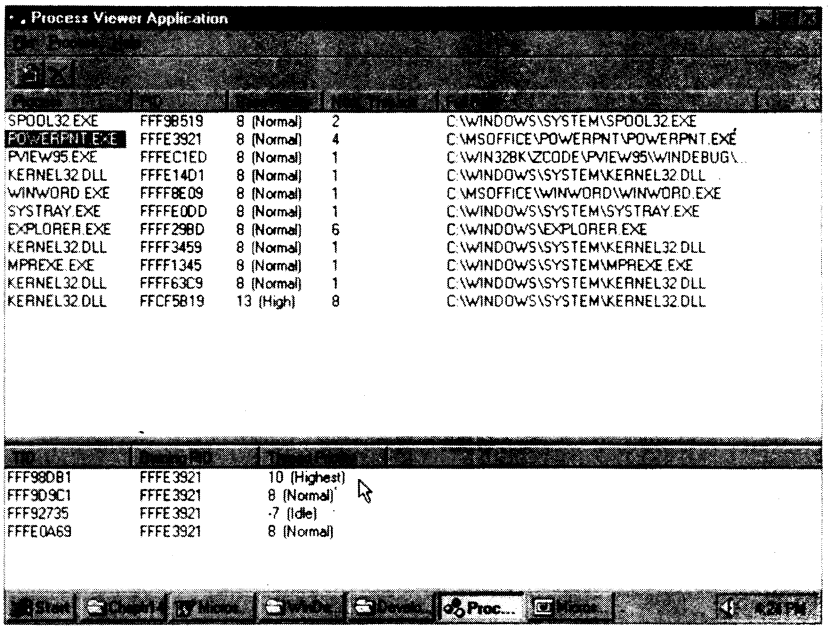


图 14-3　打开一个现成简报的时候，PowerPoint 建立了第四个线程

假如切换至幻灯片排序器（Slide Sorter）模式，PowerPoint 就会建立第五个线程，它的优先级要比前一种情况低一级（如图 14-4 所示）。所有幻灯片都显示出来后，这个线程就会中断并且消失掉。

根据经验来看，同时保持的线程数量越多，编写的代码就会越复杂。除此以外，更多的线程并不一定意味着可以获得更好的性能以及响应敏感度更好的一个用户界面。

通常情况下，对于有几百上千行源代码的一个应用程序来说，六个线程是一个比较合适的数字。真正应该考虑的是这些线程怎样利用 CPU 以及对程序运行进行优化，这是影响程序整体质量的两个关键因素。我们更强调的是线程的质量，而不是它的数量。某些情况下，和一个更加复杂的多线程应用程序比较起来，一个只有两个线程的程序也许会被评定成更好的 Win32 进程，因为它可以提供更高的功能性。请大家参考图 14-5 的例子，其中的 MS Word 在一个打印作业进行进行的前提下，不允许另外一个打印作业的启动。

请大家参考本章后面的“用多少线程?”部分，其中能找到关于这个问题更为详尽的信息。

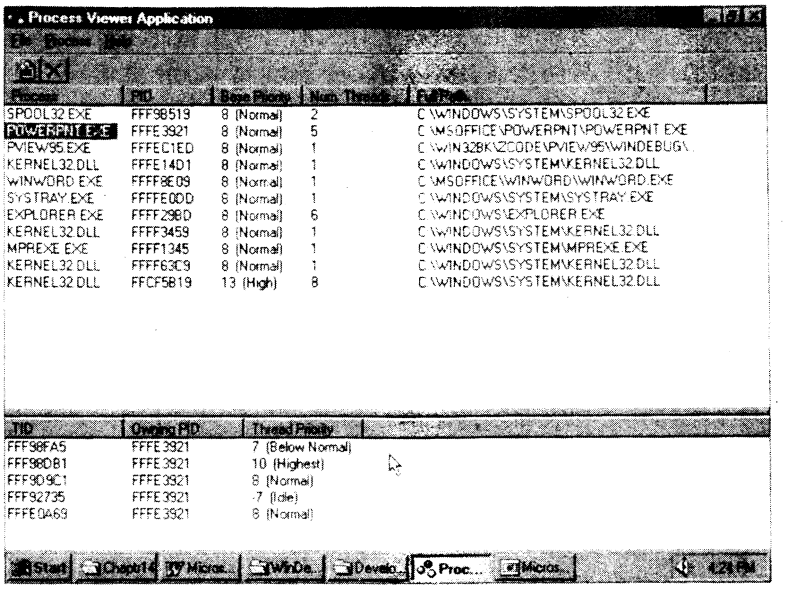


图 14-4　PowerPoint 使用运行于中等优先级的第五个线程对幻灯片进行排序

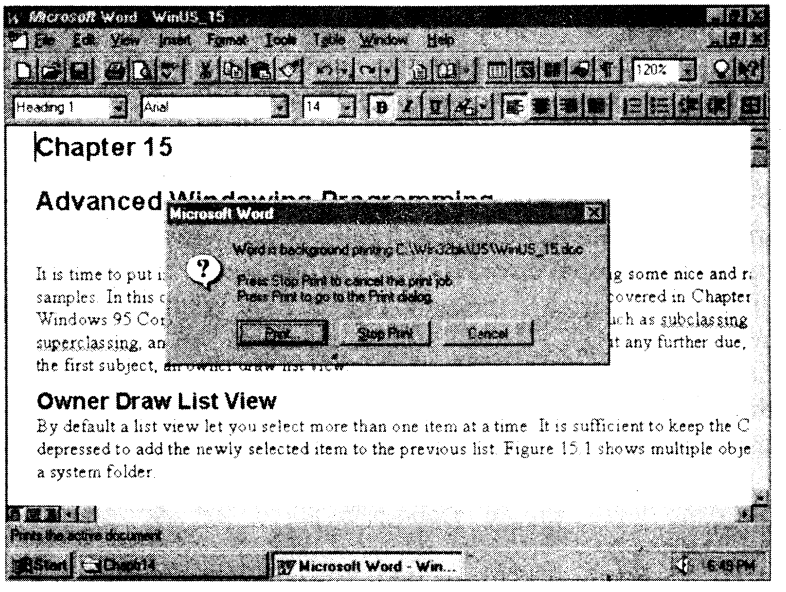


图 14-5　MS WinWord for Office 95 不支持多重打印

编写可重复利用的线程

并没有什么限制开发者编写一个可以重复利用的线程。通过对 CreateThread（）的多次调用，一个函数可以被激活多次使用，这样就构成了可以重复利用的一个线程。举个例子来

说，假设我们已经开发了一个进程，它可以考虑到与文件打印有关的所有问题。假如与打印有关的那个代码部分完全可以重新利用，就没有什么能够防止我们重复数次去执行它。这也意味着用户可以同时激活多个打印作业，一个接一个排列起来依次完成。每个打印作业都作为一个独立的进程运行，每个进程与其他进程在结构上都是完全一致的，然而处理的却是不同的数据。从末端用户的眼光来看，这就表示他（她）不局限于一次只能对一个打印作业进行控制。

为了在一个 Win32 进程里实现这种行为，编写一段可以完全重复利用（可重入）的代码是是关键。这种可以重复利用的代码其实就是一个特殊的函数，它可以在执行的时候动态地分配数据，同时不用调用其他任何形式的静态信息。经过这样的处理以后，线程就能确保每次激活它的时候都为它分配专门针对某个实例使用的一个新的数据拷贝，不会发生潜在的危险，以至于把与两个不同任务相关的数据混淆起来。

通常情况下，一个可以重复利用的线程将通过 VirtualAlloc () 或者堆函数来动态地分配数据缓冲区，同时把返回的指针存储在一个单一的进程里，使其唯一能够由某个单一的线程进行访问。Win32 API 提供了一系列函数，用它们来自动化完成这些任务。这种自动化处理是通过一种叫作“线程本地化存储”（Thread Local Storage）的机制来实现的。

14.2 线程本地化存储

在线程本地化存储（TLS）机制里，某个给定进程里的每个线程都分配了一个内存位置，其中存储了与线程有关的特定数据。这种机制是通过是一个四字节长的值来实现的，其中通常存放了指向某些内存区域的一个指针。在 TLS 机制里涉及到了四个 API 函数：TlsAlloc ()，TlsSetValue ()，TlsGetValue 以及 TlsFree ()。

TLS 背后的理论是相当简单的。TlsAlloc () 可以返回一个索引值，它唯一性地标识了属于那个进程的所有线程。假如由于某种原因导致了函数调用的失败，它就会返回一个特殊的值：0xFFFFFFFF。

```
#include〈winbase.h〉
DWORD TlsAlloc (void);
```

返回的索引值可在以后通过进程内业已存在的任何一个线程加以利用，这些线程可以通过 TlsSetValue () 函数存储一个四字节的值。

```
#include〈winbase.h〉
BOOL TlsSetValue (DWORD dwTlsIndex, LPVOID lpvTlsValue);
```

参数	说明
DWORD dwTlsIndex	由以前对 TlsAlloc () 的调用返回的一个 TLS
LPVOID lpvTlsValue	准备存储的一个四字节长的值，由当前进程调用
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值

在通常情况下，我们在一个进程里首次调用了 TlsAlloc () 以后，以后就没有必要调用第二次。这是由于每个线程都可以利用由 TlsAlloc () 返回的索引值，从而存储自己专用的一个

32 位指针。对于最初通过调用 TlsAlloc () 而返回的 TLS 索引值来说，系统会把与这个索引有关的内部存储值与每个线程关联起来。尽管这也许是对 TLS 进行利用的一种最恰当的方式，但是一个单一的进程或者 DLL 也有权利通过 TlsAlloc () 返回的索引值获得 TLS_MINIMUM_AVAIABLE 索引值。

如果想通过 TLS 机制获得存储下来的一个值，线程可以调用 TlsGetValue () 函数，同时在其中传递进程的 TLS 索引，如下所示：

```
#include〈winbase.h〉
LPVOID TlsGetValue (DWORD dwTlsIndex);
```

参数	说明
DWORD dwTlsIndex	由以前对 TlsAlloc () 进行调用而返回的 TLS 索引值
返回值	在正文里讨论
LPVOID	在调用线程的 TLS 位置存放的一个值，这个位置与指定的索引值是对应的

总而言之，TLS 就好象一个简单的 RAM 数据库，它与某个索引关键字关联在一起，这个索引关键字标识了一个进程或者 DLL 里的所有线程。利用 TLS 索引，我们可以对一个四字节的内存区域进行访问，这个四字节区域是为每个线程保留的。显然，TlsSetValue () 函数必须从一段进程代码内进行调用，这样才能在那个进程与自己的 TLS 位置之间建立内部的关联。对于每个 TLS 位置内信息的存取来说，其中涉及到的所有工作都是由系统从内部进行管理的，对此不必考虑额外的开销。所有的线程都处理完毕以后，进程就会发出对 TlsFree () 的一次调用，从而释放 TLS 索引，并使其准备好由另外一个进程在将来进行调用。

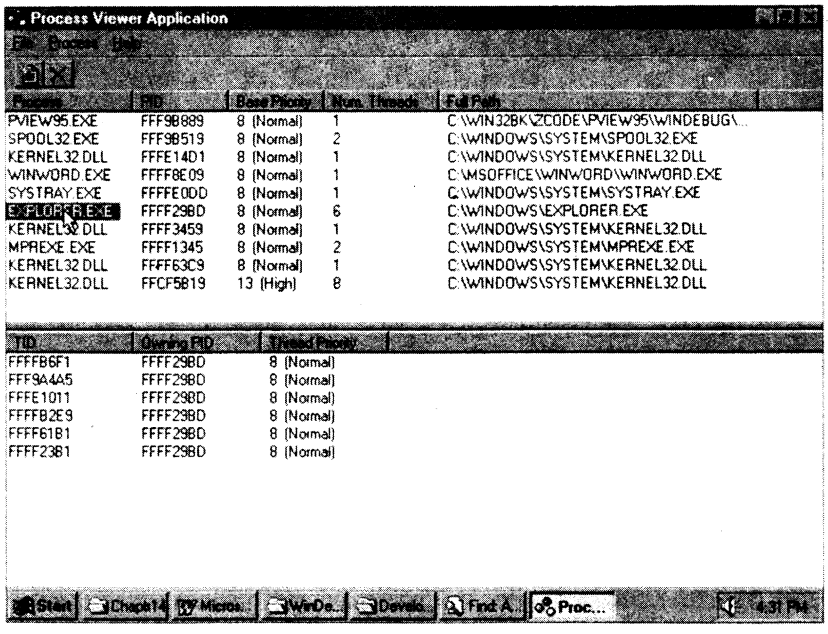


图 14-6　执行 Find 后激活了 EXPLORER.EXE 进程里的一个新线程

TLS 机制代表了与动态内存分配技术的一种很有价值的集成，这样可以实现一个多线程环境下对可重复利用代码的编制。可重复利用代码的一个很好的例子可以参考 Start 菜单内的 Find 选项。正如我们将在第十六章“Win95 外壳的开发”里提到的那样，Win95 任务栏只不过是 EXPLORER. EXE 模块的一个组件而已。执行 Find 以后，EXPLORER. EXE 里就会建立一个新的进程。PVIEW95 现在列出了一个 EXPLORER 进程里的六个线程，其中两个与正在执行的两份 Find 的拷贝有关，如图 14-6 所示。

显然，编写这种线程的时候必须考虑到再利用的问题。否则，就不可能同时启动两种不同的搜索——由 Find 支持的一种特性。我们不知道这在 EXPLORER. EXE 里是如何实现的，微软公司的工程师们也许采纳了一种很特殊的方法，让相关的数据完全独立于线程代码以外。

14.3 线程、窗口和消息

现在让我们对关于线程的一些知识进行实践。本章的第一个例子展示了线程与窗口之间的某些关联，以及针对信息跨越进程边界进行传递的问题应该事先采取什么样的一些预防措施。

WTHREADS 程序的显示情况如图 14-7 所示。两个窗口属于相同的进程，然而它们是在不同的进程里建立的。屏幕上端的 PVIEW95 证明了 WTHREADS 是一个多线程程序。

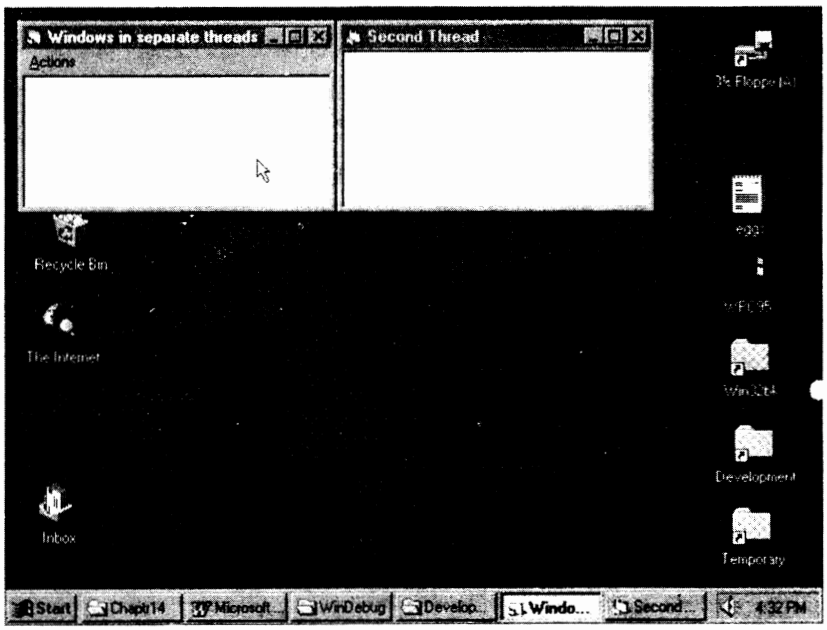


图 14-7　WTHREADS 在两个单独的线程里建立了两个窗口

从视觉效果的角度来看，我们完全不可能断言那两个窗口是在同一个线程里建立的，还是在两个不同的线程里分别建立的。进行判断的唯一办法就是对背后的代码进行检查。处理 WM_CREATE 消息的时候，应用程序生成了另外一个线程，它激活了 SecondThread () 函数：

```
...
case WM_CREATE:
{
    hInstance = ( (LPCREATESTRUCT) lParam) -> hInstance;

    // preparing information to start the second thread
    trdata.hwndFirst = hwnd;
    trdata.hInstance = hInstance;

    // creating second thread
    if (! (hThread = CreateThread (NULL,
                                   0L,
                                   (LPTHREAD_START_ROUTINE)
                                       SecondThread,
                                   (LPVOID) &trdata,
                                   0,
                                   &dwThreadID)))
        MessageBeep (0);
}
...
```

在执行 CreateThread () 之前，要先在 THREADDATA 数据结构里填写主窗口句柄以及应用程序的实例句柄。这个 THREADDATA 结构是在源代码的起始处建立的。当第二个进程必须把数据传递给一个窗口的时候，存储在这个结构里的主窗口句柄以及应用程序实例句柄信息就会派上用场了。

SecondThread () 与标准的 WinMain () 入口点是相当类似的，就像下面这个代码段显示的那样：

```
...
LRESULT WINAPI SecondThread (LONG lParam)
{
    HWND hwnd;
    MSG msg;
    char szClassName [] = " SECONDTHREAD";
    char szWindowTitle [] = " Second Thread";
    WNDCLASSEX wc;
    PTHREADDATA ptrdata = (PTHREADDATA) lParam;
    HINSTANCE hInstance = ptrdata -> hInstance;
```

```
wc.cbSize = sizeof (wc);
wc.style = CS_VREDRAW | CS_HREDRAW;
wc.lpfnWndProc = ThreadWndProc;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance = hInstance;
wc.hIcon = LoadIcon (hInstance, " second");
wc.hIconSm = LoadIcon (hInstance, " second");
wc.hCursor = LoadCursor (NULL, IDC_ARROW);
wc.hbrBackground = GetStockObject (WHITE_BRUSH);
wc.lpszMenuName = NULL;
wc.lpszClassName = szClassName;

if (! RegisterClassEx (&wc))
{
    MessageBeep (0);
    return FALSE;
}

hwnd =CreateWindowEx (WS_EX_CLIENTEDGE | WS_EX_
        WINDOWEDGE,
                    szClassName,
                    szWindowTitle,
                    WS_OVERLAPPEDWINDOW,
                    10 + WINDOWWIDTH, 10,
                    WINDOWWIDTH, WINDOWHEIGHT,
                    NULL,
                    NULL,
                    hInstance,
                    (LPVOID) ptrdata);

SetWindowPos (hwnd, HWND_TOP, 0, 0, 0, 0,
            SWP_NOMOVE | SWP_NOSIZE | SWP_SHOWWINDOW);

// message loop
while (GetMessage (&msg, NULL, 0, 0))
{
    TranslateMessage (&msg);
```

```
            DispatchMessage (&msg);
        }

        return FALSE;
    }
    ...
```

SecondThread () 的作用是注册另外一个窗口类，然后建立属于那个类的一个窗口（即图 14-7 里显示于右侧的那个窗口）。建立进程时传递的 THREADDATA 数据结构的地址会在 SecondThread () 的 lParam 参数里接收到。

第二个类窗口进程——ThreadWndProc () ——是用 C 语言编写的一个标准窗口进程。仅仅观察这个函数的名字，也许会让我们感觉有些迷惑，因为它实际上与进程没有一点关系。这个函数将在线程的环境下执行，但是和用一个线程开发一个标准的 Win32 应用程序的情况比较起来，这种执行并没有什么区别。即使在这种情况下，窗口进程函数也会在只有一个线程的环境中执行。在 WTHREADS 程序里，ThreadWndProc () 函数是在第二个线程的环境中执行的。在一条 WM_QUIT 消息张贴到自己的队列中之前，SecondThread () 是不会中断执行的。在这种情况下，我们没有必要实现与其他线程的任何形式的同步处理，因为以此为基础的消息机制提供了一个稳固的运行基础，在这个基础上的线程几乎可以永远运行下去。

WTHREADS 程序也需要我们进行一些特殊的考虑。这个程序有两个消息循环和两个队列，每个都需要同时针对两个窗口接收和提供消息。经过这样的处理以后，即使其中某个窗口需要处理一项特别耗时的任务，用户仍然能够与应用程序进行交互作用，这是一种具有积极意义的后果。对于两个不同的队列来说，数据的周转是独立进行的。每当有消息可以使用的时候，这些数据就会由原始输入线程产生。这种方法提高了代码的灵活性，并且可以对用户的请求提供及时的响应。Win32 API 提供了一个名为 AttachThreadInput () 的函数，用于把一个线程输入列表与另外一个线程联系起来，这样就消除了由相同应用程序内独立消息队列带来的所有优点。

在一个多线程应用程序里，只要当我们需要在属于不同线程的窗口间交换数据的时候，对多重输入队列的控制才有可能变得相当复杂。正如我们在第四章“消息和重画模式”里提到的那样，穿越进程边界的一条发送消息可能对程序产生不可预知的后果。在 WTHREADS 程序里，第一窗口提供了一些特殊的菜单，它们可以专门用于测试独立线程间窗口的交互作用。

首先，我们可以试验挂起一个线程的真正含义是什么。在 Actions 菜单内的 Supend second thread（挂起第二个线程）菜单可以冻结第二个线程以及它的窗口。正如图 14-8 显示的那样，除非恢复了进程以前的状态，否则用户对它的交互作用是没有效果的。假如在右边那个窗口的标题栏上双击鼠标左键，我们不能得到任何响应。但是当线程 1 恢复了线程 2 以后，毋需再次进行双击操作，第二个窗口就可以立即以最大化显示（这是双击标题栏后的正常反应）。这就证明了尽管线程已经临时性地处于挂起状态，但是消息队列仍然存储了用户对它采取的操作。

Resume second thread（恢复第二个线程）菜单项可以恢复第二个线程的起始状态。使用 SupendThread () 以及 ResumeThread () 是迄今为止最糟糕的方案，因为它们将用于冻结一

个进程，然后唐突地释放它，其间没有经过任何同步处理。

接下来的两个菜单项——Send big job（发送大作业）和 Post big job（传递大作业）——可以在第二个线程里启动一个相当耗时的计算操作，这是通过张贴一条对应的消息得到的结果。这两个菜单项的第一个会发送一条 SendMessage（）消息，该消息将抵达第二个线程内的窗口进程，并且只有在计算作业完成以后才返回。在这段时期内，两个窗口都似乎已经挂起，不能对用户的任何请求提供响应。与窗口的任何交互作用都注定要失败。对于不明真象的末端用户来说，这种现象显然是令人相当失望的，所以我们应该尽量避免发生这种不愉快的事件。然而，假如我们用 PostMessage（）取代 SendMessage（），情况就能发生一些好转。目标窗口仍然处于挂起状态，然而第一个窗口却可以对用户的请求提供令人满意的响应。

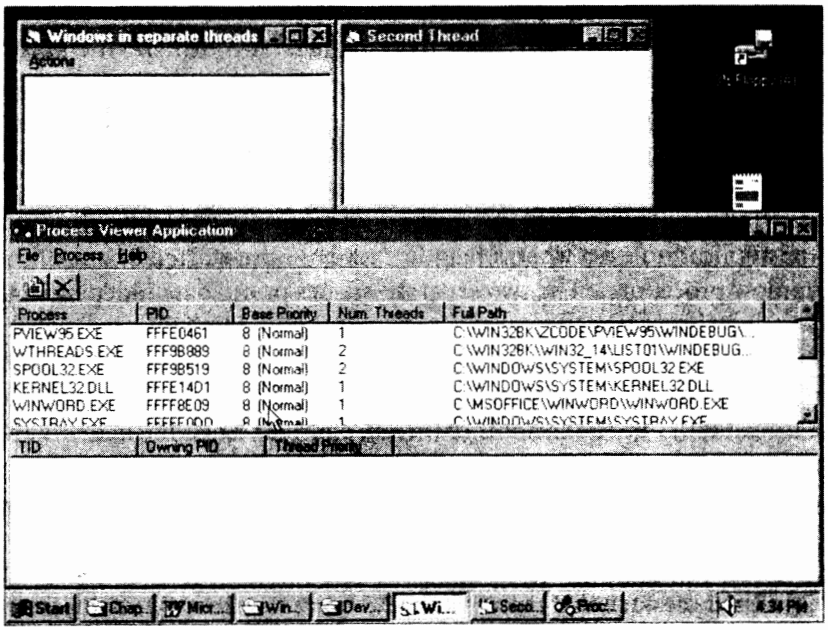


图 14-8　属于第二个线程的窗口不能对用户的交互作用提供响应，
因为线程已经挂起了。窗口输出只有在线程恢复以后才能更新

至此为止，我们应该明白对消息进行张贴并不是一种很明智的办法，特别是需要对用户请求立即作出响应，或者需要进行其他处理的时候更是如此。在第四章“消息和重画模式”和第七章“建立窗口的技术”里，我们首次接触了一些新的 API 函数。这些函数可以在线程间发送消息的时候避免可能出现的死锁现象。对这些 API 函数的一次简要回顾也许可以帮助我们在对待进程间消息处理的时候决定出最佳的方案。

ReplyMess age（）可以放置于某个接收方线程的窗口进程里，以便对调用方线程作出即时的响应以及释放调用方线程的执行模块。尽管此时真正的处理还没有开始，从调用方线程发出的消息流仍然可以通过 ReplyMessage（）获得一个及时的答复。然而，尽管 ReplyMessage（）能够释放调用方线程，它仍然没有提供针对进程间消息处理的一套完整的解决方案，因为两个线程的处理是不同步的。因此，我们可以设置对 InSendMessage（）的调用，利用它判断进入消息的起点，假如判断通过，就用 ReplyMessag（）立即提供响应。然而，开发一个多线

程应用程序的时候，假如从一个线程到另一个线程将发出几个消息流，这种方案就显得没有什么意义了。

SendNotifyMessage（）的行为类似于一个 PostMessage（）函数，它也可以把一条消息传递给目标窗口进程，并且在这以后返回。SendNotifyMessage（）具备的唯一优点在于当目标窗口和源窗口从属于同一个进程的时候，它的行为就类似于一个标准的 SendMessage（）函数。

SendMessageTimeout（）提供了一些更有趣的特性。“超时”（timeout）参数允许我们指定一个时间范围。假如接收方窗口正处于“忙”状态，那么一旦设定的时间已到，这个函数就会立即返回，这样就为发送方线程提供了继续执行的机会。SendMessageTimeout（）函数在两种情况下都可以返回：要么执行正常地结束，要么设定的时间已到。

在 WTHREADS 程序里，假如我们选择 SendMessageTimeout 菜单项，它就会向第二个窗口发出一条消息。这条消息发出后一小会儿时间，SendMessageTimeout（）就会返回，因为设定的时间已经超出了（等待时间已经预先设定为五秒，然而对于由 SendMessageTimeout（）发出的消息来说，它所触发的操作则需要耗费更多的时间）。

从一个进程向另一个进程发送消息的最后一个办法是调用 API 函数 SendMessage Callback（）。这个函数可以发出一条消息，然后立即返回，继续执行自己常规的处理。

```
#include〈winbase.h〉
BOOL SendMessageCallback (HWND hwnd,
                          UINT uMsg,
                          WPARAM wParam,
                          LPARAM lParam,
                          SENDASYNCPROC lpResultCallBack,
                          DWORD dwData);
```

参数	说明
HWND hwnd	目标窗口的句柄
UINT uMsg	发送给目标窗口的消息
WPARAM wParam	附加信息
LPARAM lParam	附加信息
SENDASYNCPROC lpResultCallBack	一个函数的地址，消息处理结束后将自动调用这个函数
DWORD dwData	由应用程序定义的数值，准备把它传递给异步进程
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值

除了在 SendMessage（）里使用的一些标准参数以外，我们还要提供一个特殊函数的址。消息处理结束以后，系统就会自动调用这个函数。这个异步函数会返回一个空值，并且需要

接收一个窗口句柄、消息标识符、由用户定义的某些数据以及一个消息从属值，这个值定义了消息处理的结果。在 WTHREADS 程序里，大家可以发现像下面这种样子的 SendAsyncProc () 函数：

```
VOID CALLBACK SendAsyncProc (HWND hwnd,
                             UINT msg,
                             DWORD dwData,
                             LRESULT lResult)
{
    switch (msg)
    {
        case WM_BIGJOB:
        {
            MessageBeep (0);
        }
            break;

    }
    return;
}
```

在 WTHREADS 里，对消息 WM_BIGJOB 的处理结束以后，第一个线程将会简单地发出一声铃响。在这儿，有趣的一个地方在于第一个线程的调用是自由的，不需要在其中实现任何 IPC 机制。我建议大家对 WTHREAD 程序进行修改，根据第二个线程的状态发出不同情况下的一条新消息，看看会发生什么事情。

总而言之，我们必须注意到 MS Windows 95 是一个多任务、多线程的环境，在 Win16 世界里存在的所有同步的可能都已经不复存在。

14.4 线程性能的衡量

建立和消除一个线程花去了多少时间？或者一个线程在 CPU 里运行了多长时间？尽管这是两个非常合理的问题，但是在 Windows 95 里确实不好作出准确地回答。Win32 API 提供了 GetThreadTime () 函数，它专门用于提供这种类型的信息，这对我们测试和调整一种特定的算法或者一段特定的代码是大有裨益的。不幸的是，GetThreadTime () 在 MS Windows 95 里没有得到支持，只能在 NT 里使用。因此，我们必须降低自己的期望值，并把注意力放在 MS Windows 95 里真正可行的办法上面。在本书附带 CD 的 Listing 14.2 里，大家可以找到一个名为 TIME 的示范程序，它可以对线程创建过程中涉及到的时间开销进行测量。图 14-9 向大家展示了在建立了 18 个线程的时候，由 TIME 程序显示出来的耗费时间。

正如我们预期的那样，由 TIME 返回的值只是指出了线程的建立和消除两个连续操作两者之间经历的时间。对于列表框内指定时间的那两栏来说，前面那栏表示了刚刚建立线程之

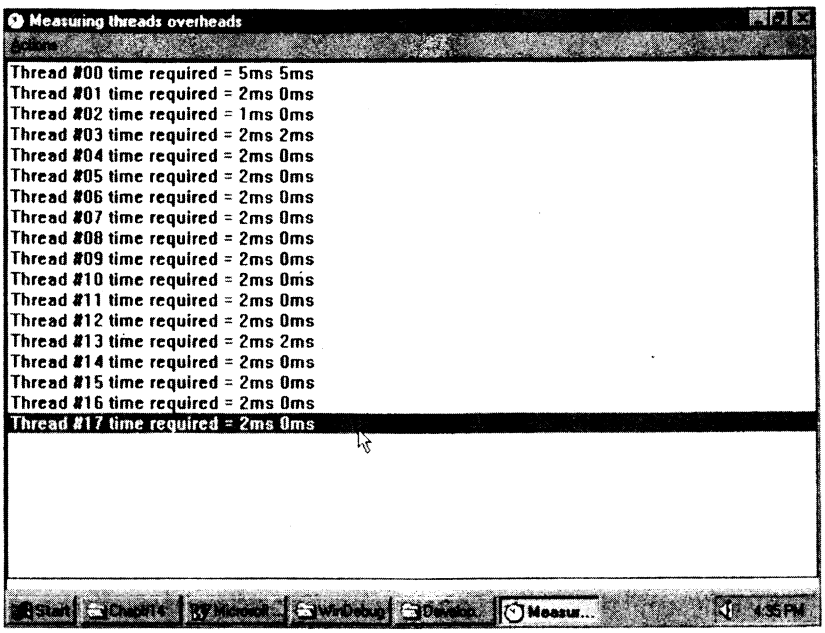


图 14-9　对建立一个线程所需的时间进行测量

前到该线程的句柄返回（意味着线程已经中断）这期间所经历的时间。第二栏则测量了执行一个 CreateThread（）函数所需的时间。根据线程的不同，这些时间数据也会发生变化。然而，除了第一对数据以外，其他数据的变化都是非常有限的。尽管进程的建立需要涉及到系统内的几个行动，但是这种测量的确可以让我们理解一个进程的建立有多快。下面摘录的这个代码段向我们展示了在应用程序里如何对这两个值进行计算：

```
...
// let's create MAXTHREADS threads originally suspended
for (i = 0; i < MAXTHREADS; i++)
{
    dw1 = GetTickCount ();
    hThread [i] = CreateThread ( (LPSECURITY_ATTRIBUTES) NULL,
                                 4096 * 12,
                                 (LPTHREAD_START_ROUTINE)
                                     SecondThread,
                                 (LPVOID) &data,
                                 0,
                                 &dwtid);

    dw2 = GetTickCount ();

    // wait until the thread dies
```

```
        WaitForSingleObject (hThread [i], INFINITE);
        // get the thread return value
        GetExitCodeThread (hThread [i], &dw3);

        // fill the listbox
        wsprintf (szText, " Thread # %02d time required = %dms %dms", i,
            dw3 - dw1, dw2 - dw1);
        ListBox_AddString (hwndListBox, szText);
    }
    ...
```

GetTickCount () 函数可以返回自从引导以后经历的时间，这个值是用毫秒数来表示的。对 WaitForSingleObject () 函数的调用可以保证主线程把自己挂起，直到新建立的线程到达了它的中断状态为止。GetExitCodeThread () 函数可以在自己的第二个参数里存储线程的返回值。

```
#include <winbase.h>
BOOL GetExitCodeThread (HANDLE hThread, LPDWORD lpExitCode);
```

参数	说明
HANDLE hThread	线程句柄
LPDWORD lpExitCode	包含了线程的中断状态
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值

其中，lpExitCode 参数可以接收用其中一种线程状态定义指定的值。它可以通过一个 return（返回）关键字明确地返回，这个关键字既可以在 ExitThread () 里设置，也可以在 TerminateThread () 里设置。另外，它可以通过一个中断值返回。通过调用 WaitForSingleObject () 函数，用它传递进程句柄，然后再调用 GetExitCodeThread () 函数，应用程序的基本进程就可以确定第二个线程已经中断，然后取得它的中断时间。所有这些时间值都是用毫秒来表示的，但是如果使用一个更小的度量单位也许显得更恰当一些。

14.5 用多少线程

对于线程来说，一个普遍关心的问题是系统能够控制的最大线程数量。这个数量是有限的吗？也许如此，但是任何技术文档都没有对此给出一个准确的数字。我以前就曾经提到过，一个能良好运行的 Win32 程序与代码内的线程数量并没有直接的关系。并不是说线程越多，这个程序就越好——程序运行好坏只是与线程的编写质量有关。在本书附带 CD 的 Listing 14.3 里，我提供了一个名为 THREADS 的示范程序，这个程序可以建立数目众多的线程。应用程序启动以后，它把准备创建的线程的缺省数目设置成 400，我们可以通过一个滑杆（Trackbar）通用控件对这个数字进行设置。大家可以注意到，滑杆上的刻度到 1 到 2000 之间。

2000 是一个相当大的数字，在现实的编程环境中几乎不可能遇到这种情况。Create 和 Destroy 按钮则对应于完成线程的建立和消除处理。如图14.10所示。

线程既可以建立成活动线程，也可以在建立以后立刻挂起。Createactive thread（建立活动线程）菜单项可以对线程的最初状态进行控制。除此以外，Actions 顶级菜单提供了两个菜单项，分别用于启动 System Monitor 工具软件以及 PVIEW95，如图 14-11 所示。在图 14-11

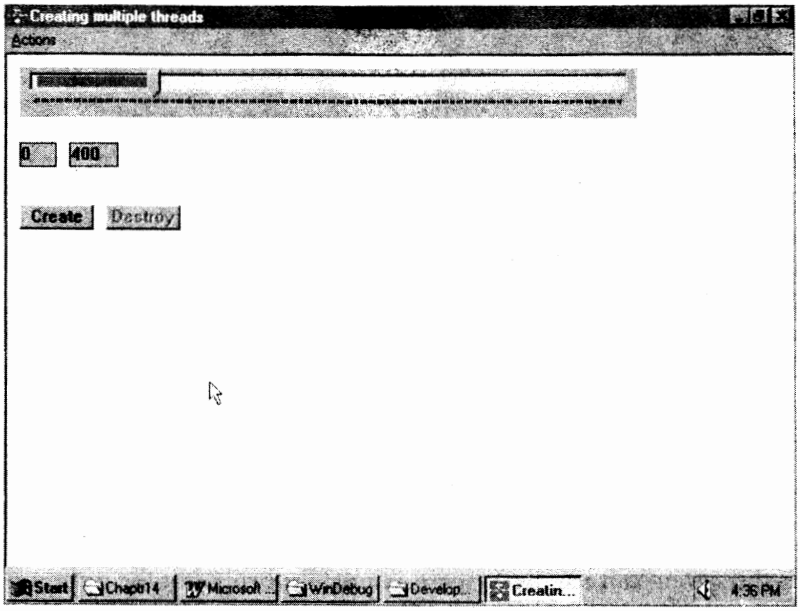


图 14-10　THREADS 是一个测试应用程序，它最多可以建立 2000 个线程

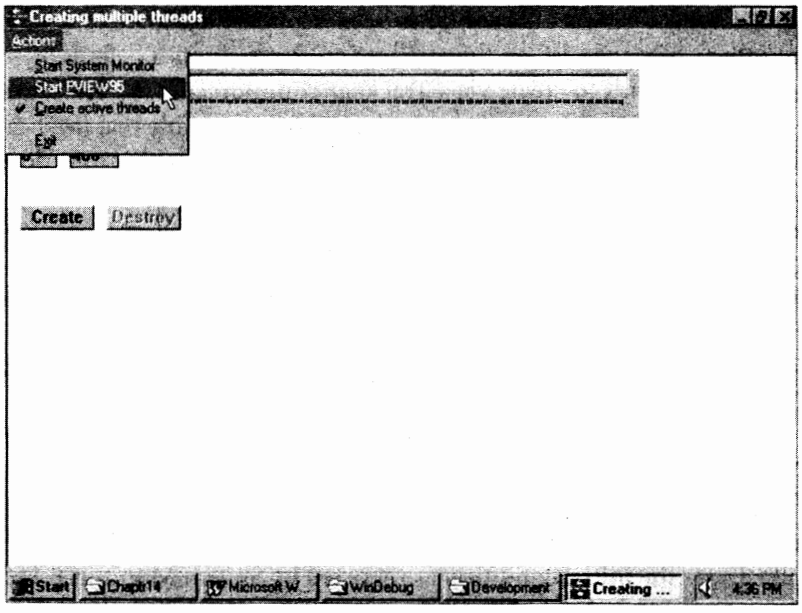


图 14-11　Actions 下拉式菜单定义了 THREADS 应用程序里的一些选项

里，系统内活动线程的总数为 30。THREADS 程序的 Create 按钮可以根据我们的希望把这个值动态地增加到 430，如图14.12所示。PVIEW95 在图 14-13 的显示情况证明了 THREADS 进程现在由 401 个线程组成，新建的 400 个加上最初的那个线程。

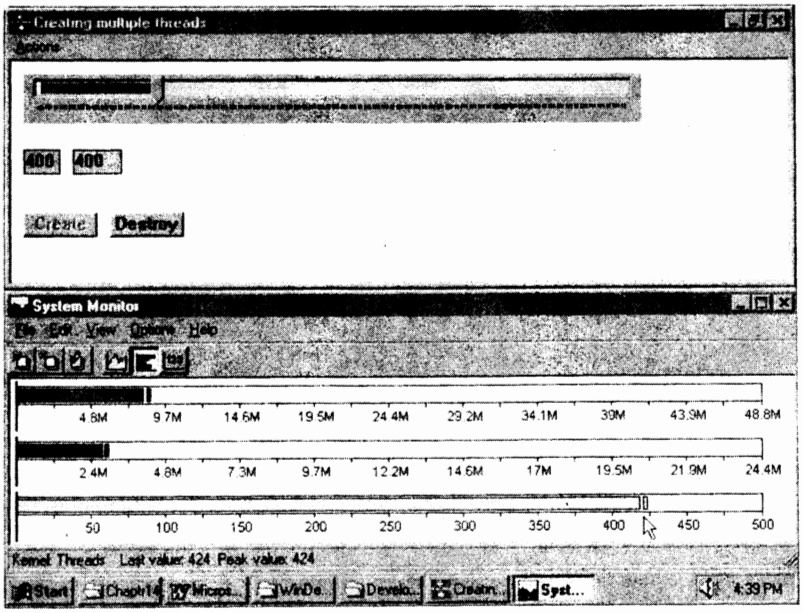


图 14-12　系统内的线程总数现在超过了 400

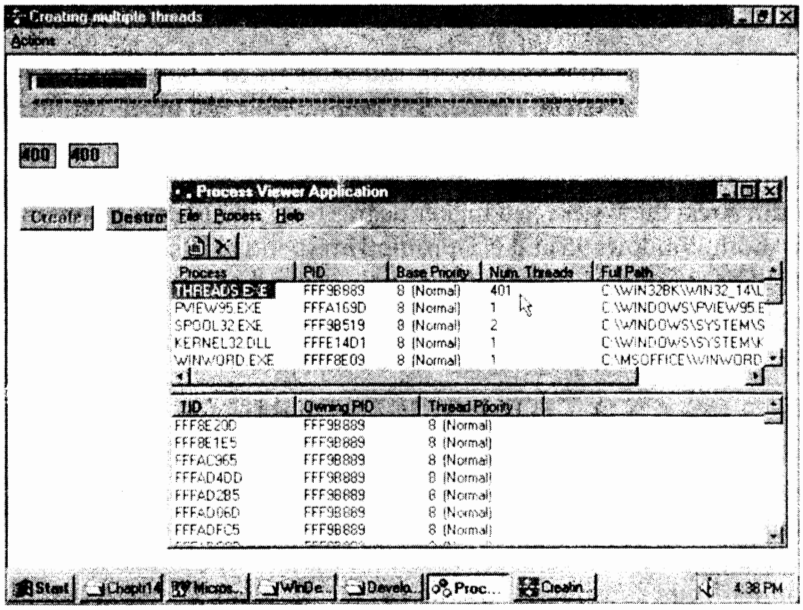


图 14-13　PVIEW95 列出了在 THREAS 进程里建立的 401 个线程

MS Windows 95 能够支持 1000 个线程吗？现在让我们看一看假如用 THREADS 示范程序建立了另外 1000 个线程以后会发生什么情况。图 14-14 展示了由 System Monitor 报告的最初状态：8.1M的线程正在运行，同时交换文件的长度上升到 5.3 兆字节。

执行 THREADS 的时候，假如把线程值设置成 1000，系统内就会发两件愉快的事情。首先，MS Windows 95 仍然保持活动以及运行状态。其次，交换文件不断地增大，从而容纳下了所有线程堆栈，如图 14-15 所示。

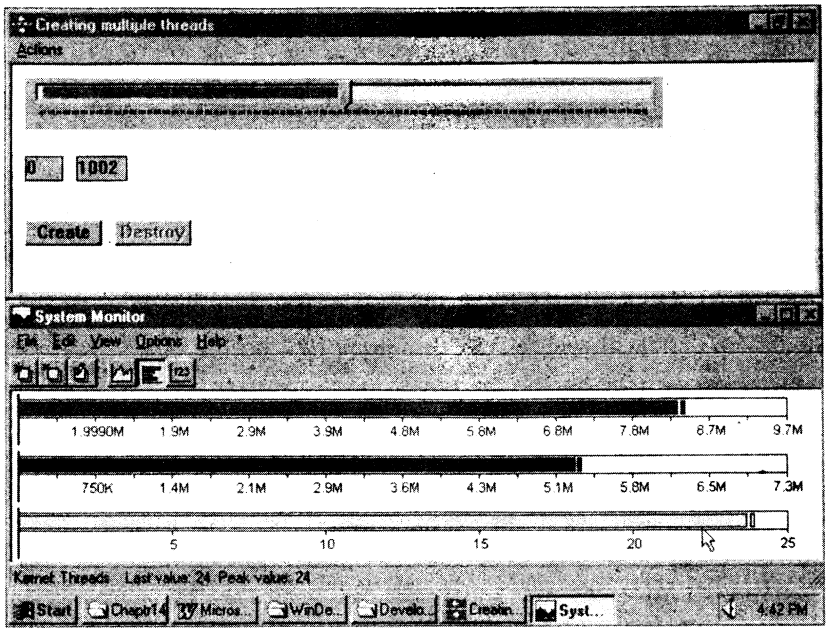


图 14-14　建立附加的 1000 个线程之前，系统交换文件的大小以及线程数

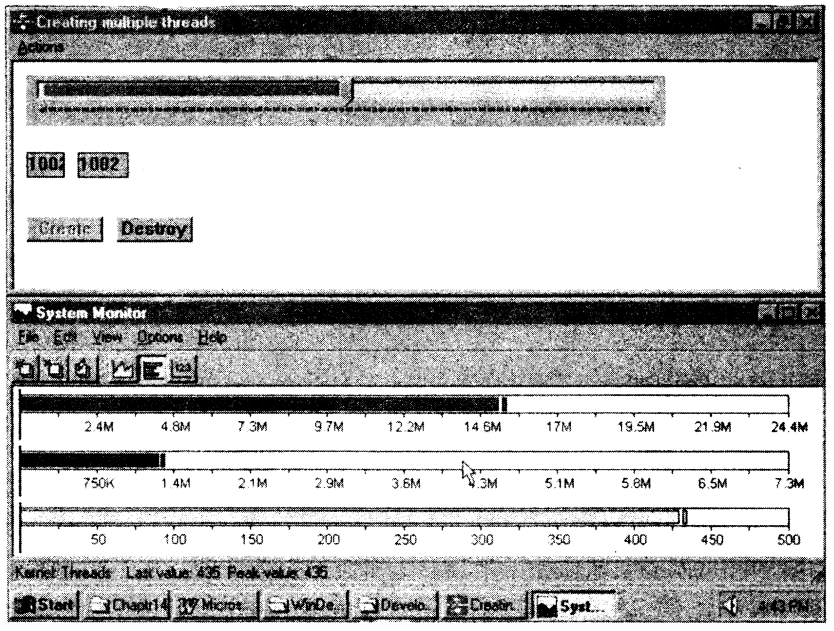


图 14-15　执行了 1000 个线程以后，系统的配置情况

我们可以尝试建立更多的线程，从而为系统施加更大的压力。无论 Windows 95 还是 NT 都能对这种请求提供迅捷的响应。但是，大家只应该把这当作一种试验，在现实的编程环境中一般不会碰到这种问题。

14.6 线程和用户界面

通过前面两个例子，我们可以总结出两条规律。首先，线程的建立时间是相当短的；其次，系统同时可以支持的线程数远远超出现实的需求。接下来，我们的目标是利用线程的存在来提高 UI 的响应敏感程度。

如果大家还有印象，应该记得本书 CD 的 Listing 9.2 里曾经列举过一个简单的应用程序，这个程序叫作 CLASSES，它能显示出全部六种 Windows 预定义控件类。

对于这些预定义控件来说，Windows 95 只对它们进行了少许改变。例如，按钮现在可以支持图象（图标和位图），编辑控件可以带有一个弹出式菜单，这个菜单通过按下鼠标右键便可激活，以及其他一些变化。列表框现在不再局限于臭名昭著的缓冲区尺寸限制，并且可以包含上兆的数据。在 CLASSES 示范程序里，我们在一个列表框里填写了 10000 个项目，长度则达到了 118890 字节。在我用的这台 AST Premmia P90 系统上面，整个填写操作耗费了大约三秒钟的时间，如图 14-16 所示。

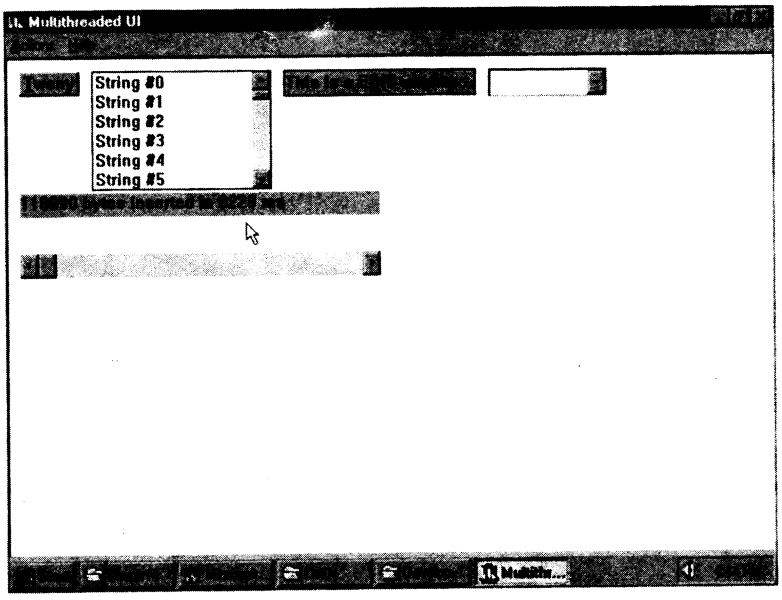


图 14-16 CLASSES 示范程序向我们展示了 MS Windows 95 的所有预定义类

填写过程中，整个应用程序都会完全挂起，直到列表框接收到了第 10000 个正文串为止。这种情况是无法避免的，因为它是一个单线程应用程序，因此所有的信息都会通过源代码进行流通。这时，假如增加一个线程应该为 CLASSES 程序带来好处，因为这样几乎能把用户等待程序响应的时间降低到零，从而显著地提高了系统的整体性能。在本书附带 CD 的 Listing 14.4 里，大家可以找到 CLASSES 程序的一个修订版本。这个修订版本已经转换成了一个多线程应用程序。程序启动以后，没有任何一个控件是自动可见的。这个版本的 CLASSES 提供

了一个菜单栏，其中的选项可以控制预定义控件的建立以及列表框的填写进程。如图 14-17 所示。

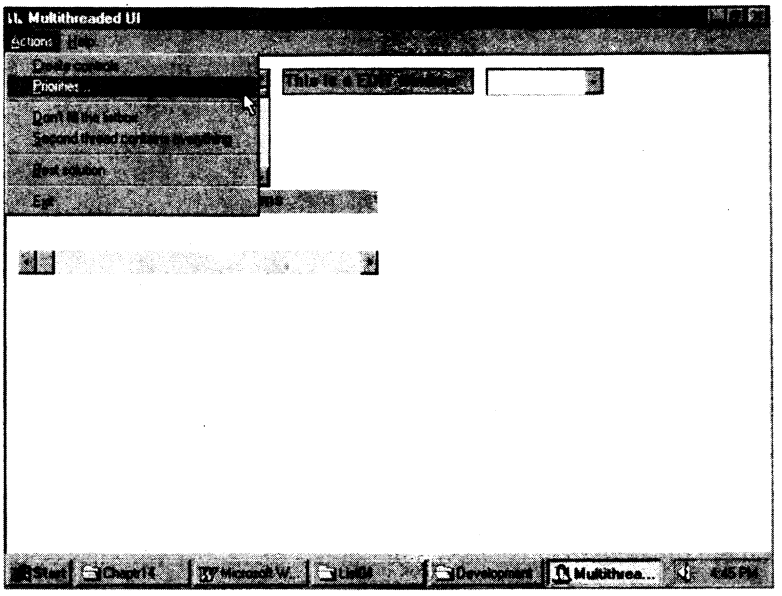


图 14-17　CLASSES 的多线程版本用一些菜单项定义了应用程序的行为

建立准备插入列表框的大约 120KB 数据是一种非常快的操作，具体的速度受 CPU 芯片的影响。在我的 Windows 95 系统里，只需花 80 毫秒的时间便可准备好 10000 个正文串，这个时间是相当短的，如图 14-18 所示。这个数据应该与 CLASSES 单线程版本的性能进行比

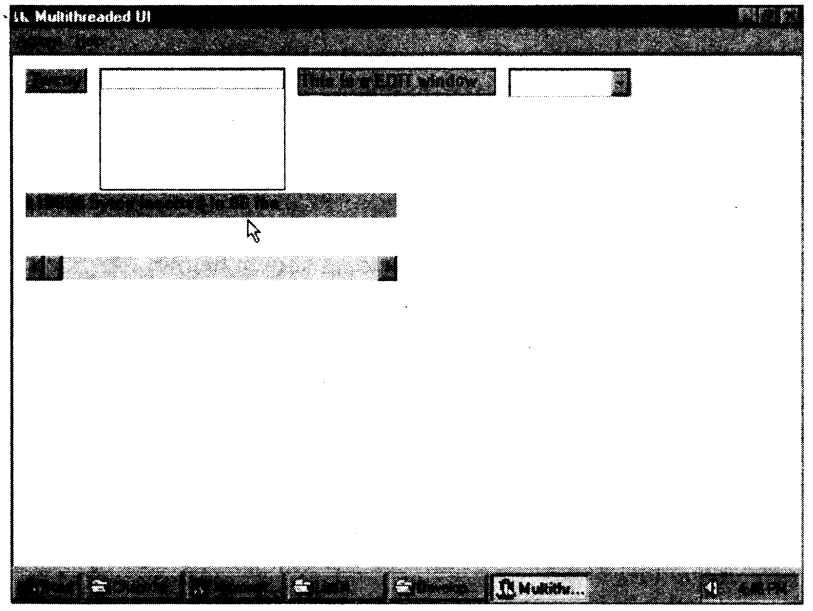


图 14-18　测量准备传递给一个列表框控件的 10000 个正文串的准备时间

较，如图 14-16 所示。两者之间巨大的差异清楚地揭示了通过 16 位 USER 和 GDI API 进行传递的系统开销。为了让一个正文串在列表框控件内显示出来，这种传递增加了额外的工作量，显示出来就意味着浪费大量时间。假如让 CLASSES 示范程序在 NT 环境下执行，并且只对这种算法需要的时间进行测量，那么也能返回一个相同的值——约 80 毫秒。因此，就纯粹的计算处理进行衡量，这 Win95 和 NT 这两种系统之间并没有根本的区别。

在编写实际的代码之前，我们应该先花些时间试着设计 CLASSES 的多线程版本，以便总结出全部有价值的选择方案。第二个线程应该足以提高这个示范程序的整体性能。然而，我们现在对这第二个线程应该完成什么工作还不是很清楚。它的目标是把用户从等待数据插入列表框的这段漫长的初始化时间里解放出来。第二个线程应该考虑到这个任务，从而对主线程进行释放，以便能立即满足用户发出的请求，比如选择一个菜单项等。总而言之，我们需要采取下面两种行动，这两种行动针对的都是第二个线程：

▶准备正文串以及填写列表框

▶建立列表框并在其中填写信息

14.6.1 情况 A：填写列表框的第二个线程

最初的时候，这看起来好象是最有趣和最具有逻辑性的一种方案。当主线程能够对用户的请求提供及时响应的时候，第二个线程则在准备和填写列表框。采取了这种方案以后，由此带来的线程间消息流通会造成可能的不利情况。

第二个线程将发出对 SendMessage () 的 10000 次调用，它们将到达主线程里建立的列表框窗口。针对线程间的消息流通，假如应用程序的主线程需要完成一个很大的作业，它就会使窗口进程在相当长的时间内都处于“忙”状态。这样一来，由这种方案得到的结果就会让人相当失望。在这样一种情况下，第二个线程会处于挂起状态，直到线程 1 空闲下来为止，这种结果正好与我们的期望相悖。为了对这种方案进行评估，我们可以把用于正文串准备和发送的语句封装到第二个线程里，就像下面这样：

```
...
LRESULT WINAPI SecondThread (LONG lParam)
{
    char szText [30];
    HWND hwndList, hwnd;
    PDATA pdata = (PDATA) lParam;
    HWND hwndStatic;
    register int i, iTot = 0;
    int iPrty;
    DWORD dw1, dw2;
    MSG msg;

    // main window hwnd
    hwnd = pdata -> hwnd;
    hwndStatic = CTRL (hwnd, CT_STATIC);
```

```
    hwndList = CTRL (hwnd, CT_LISTBOX);

    // get the time
    dw1 = GetTickCount ();

    // filling the listbox
    for (i = 0; i < LIMIT; i++)
    {
        iTot += wsprintf (szText, " String #%d", i);
        if (pdata -> fFill)
            ListBox_AddString (hwndList, szText);
    }
    dw2 = GetTickCount ();

    // inserting in the STATIC window
    wsprintf (szText, "%d bytes inserted in %d ms", iTot, dw2 - dw1);
    SetWindowText (hwndStatic, szText);
    InvalidateRect (hwndStatic, NULL, TRUE);
    return TRUE;
}
...
```

对 LIMIT 定义对应，for 循环将执行 10000 遍。整数 iTot 里存储了第二个线程内建立的字节总数，并且正好在线程返回之前用一个静态 (STATIC) 类窗口显示出这种数据。你急切地想看到这种方案的结果吗？图 14-19 为我们显示了 CLASSES 程序执行后的显示情况。

我们针对第二个线程采取的第一种多线程实现方案看起来已经达到了目标。更准确地说，是部分达到了目标。列表框已经部分填写好了，对此可以用一个正文串的存在来证明，这个正文串报告了完成操作需要经历的时间。当我把这个屏幕通过抓图程序抓取下来的时候，第二个线程仍在运行，仍在向列表框里插入正文串。图 14-20 可以让我们更好地理解当条二个线程在幕后活动的时候，CLASSES 里发生的情况。

多线程版本要比单线程版本慢 2.26 倍。这已经不错了！CLASSES 里完成的任务需要花 80 毫秒的时间对数据进行准备。假如 SendMessage () 函数在线程间工作，我们要花 3.5 秒的时间观看数据在列表框内的显示。这个时间也许会随着 LISTBOX 的内部结构以及它对数据封装形式的变化而变化。举个例子来说，另外一个控件也许会缩短或者增大用于显示数据的时间。尽管存在与这种延迟原因有关的不确定性，我们仍然可以确切地知道：为了完成两个独立线程之间的相同操作，我们要付出更大的代价。

为了对这种方案进行更准确的评估，我们应该在一个 NT 系统上执行相同的代码；所有的 Windows API 布局在 NT 系统里都是纯粹的 32 位。CLASSES 应用程序在具有相同硬件配置的 NT 系统内运行得有多快或者多慢呢？答案是 NT 上运行单线程版本需要耗时 2.766 秒，而

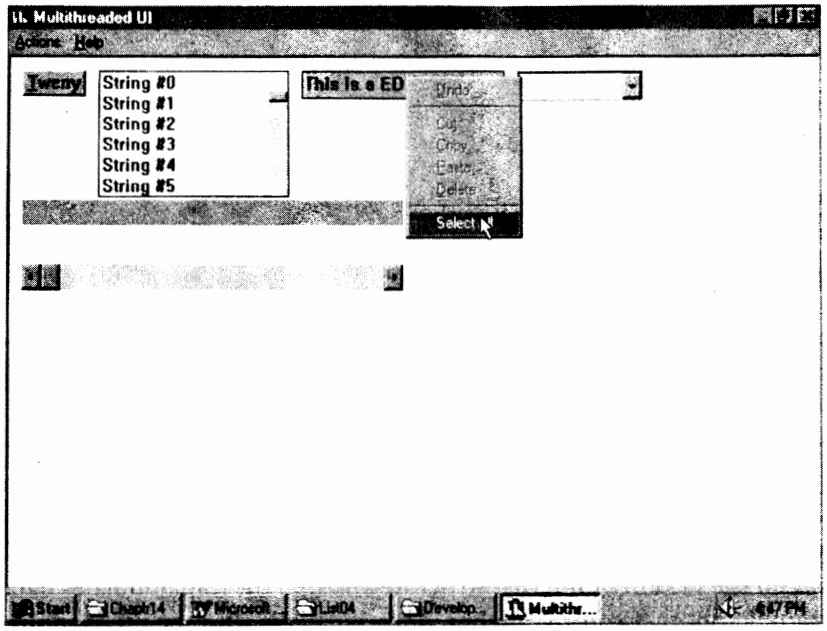


图 14-19　CLASSES 的第一种多线程方案——列表框已填写好，
其他控件也都是可见的

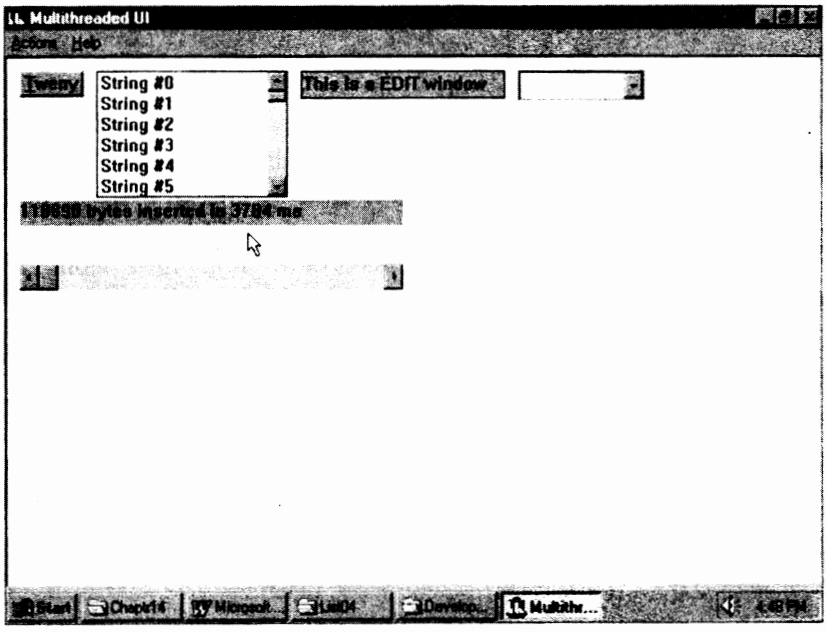


图 14-20　对第二个线程进行这样的处理以后，它的运行速度
比第九章讨论的单线程方案慢几倍

带有进程间消息传送的双进程版本则需耗时 6.766 秒。下面这张表格为大家总结了 CLASSES 的单线程和双线程版本在 NT 和 Windows 95 系统中运行时收集到的数据，注意两个操作系统下面的硬件配置都是完全一致的：

CLASSES	NT	Windows 95	两个系统间的系统开销
单线程版本	1.537	3.708	241%
双线程版本	3.845	8.162	212%
系统开销	250%	220%	

对于多线程方案来说，尽管在性能上有所降低，但是它提供了近乎完美的用户界面实施方案。所有应用程序组件在请求发出以后都能立刻显示出来，用户可以其中的任何一个组件进行交互作用，这些方面不存在任何限制。事实上，列表框并没有完全填写。假如对其进行滚动，从而查找最后的一些正文串，那么也许会导致很慢的处理步调。

关于优先级的情况如何呢？假如我们提高第二个线程的优先级，让它能以更快的步调执行，这种方法行得通吗？对于这些方面的问题，请大家继续阅读下面的小节。

提高优先级

线程是要求系统调度程序进行管理的一种基本对象。在某个特定的时刻，调度程序会计算出每个线程的优先级，然后把具有最高优先级的那个线程传递给 CPU 进行处理。调用 CreateProcess () 或者双击对应的图标，从而建立一个进程的时候，就会为它分配一个类优先级。Win32 利用两个因素来判断一个线程的优先级设置：一个类优先级值，以及在那一类中的一个线程优先级。通过对这两种因素的组合，就可以判断出整体线程优先级，这是由系统调度程序计算得到的一种数字。这个数字的范围是从 0 到 31，假如为零，就表示将为系统保留一个优先级使用。随着线程优先级的变化，类优先级将相应地增大或者减小，整体的优先级编号就是根据这种增大或者减小的类优先级计算出来的。

表 14-1 为大家列出了来自 WINBASE.H 的四种优先级类定义。通过指定这些定义值中的一个，SetClassPriority () 函数就可以实现对一个进程基准优先级的改动。这些值与我们以前提到的算术计算并没有什么关系。它们只是一个 Win32 头文件里的一些简单的定义而已。

表 14-1 Win32 提供的四种优先级类

类	值	说 明
NORMAL_PRIORITY_CLASS	0x00000020	这一类容纳了通过系统外壳内的直接选择而启动的所有进程
IDLE_PRIORITY_CLASS	0x00000040	这一类适于只有在系统处于空闲 (Idle) 状态时才需要运行的线程，比如假脱机程序以及屏幕保护程序等等
HIGH_PRIORITY_CLASS	0x00000080	这一类只能用于执行那些耗时极少的关键性任务
REALTIME_PRIORITY_CLASS	0x00000100	优先级相当高的一类，只能应用于需要在短时间内完成的相当关键的任务

```
#include <winbase.h>
BOOL SetPriorityClass (HANDLE hProcess, DWORD fdwPriority);
```

参数	说明
HANDLE hProcess	准备改变其优先级类的某个进程的句柄
DWORD fdwPriority	表 14-1 列出的某种优先级类定义
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

在 SetPriorityClass () 函数里，第一个参数是某个进程的句柄，这个句柄通过调用 GetCurrentProcess () 很容易便可得到。为了对另外一个进程进行变动，必须先取得它的句柄。只有针对那些严格相关的信息，Win32 才可以自动提供这种信息。第二个参数是表 14-1 内列出的某个定义。这些定义中的每一个都与 0 到 31 这个范围之间的某些数字对应——0 到 31 正是 Win32 能够提供的优先级。表 14-2 为我们列出了所有这些值的对应关系。

表 14-2 与优先级类定义对应的优先级

优先类定义	优先级	说 明
REALTIME_PRIORITY_CLASS	24	实时类
HIGH_PRIORITY_CLASS	13	高优先级类
NORMAL_PRIORITY_CLASS	9	进程窗口位于前台时的普通优先级类
NORMAL_PRIORITY_CLASS	7	进程窗口位于后台时的普通优先级类
IDLE_PRIORITY_CLASS	4	空闲优先级类

利用 SetPriorityClass () 函数，我们可以改变整个进程的基准优先级。因此，所有现成的以及新的线程都会自动继承这种重新分配的基准优先级设置。一旦设置好基准优先类级别以后，程序就可以通过 SetThreadPriority () 函数来改变某个线程的优先级——要么增大，要么减小。

```
#include 〈winbase.h〉
BOOL SetThreadPriority (HANDLE hThread, int nPriority);
```

参数	说明
HANDLE hThread	准备改变优先级的那个线程的句柄
int nPriority	表 14-3 内列出的其中一种定义
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

正如刚才提到的那样，SetThreadPriority () 函数的作用是改变当前线程或者位于同一进程内的另外一个线程的优先级别。要取得其他进程里的某个线程的句柄是非常困难的。第二个参数是表 14-3 内列出的某个定义。除了 THREAD_PRIORITY_TIME_CRITICAL 和 THREAD_PRIORITY_IDLE 以外，表内列出的其他值实际上需要在基准类优先级里增加或

者减掉。

假如把一个线程的优先级设置成 THREAD_PRIORITY_HIGHEST，这简单地意味着在当前优先级类的基础上再加上两上级别。对于运行于前台的、属于 NORMAL_PRIORITY_CLASS 的一个进程来说，这个设置将把自己的优先级增加到 11（9+2）。类似地，假如把同一个线程设置成 THREAD_PRIORITY_LOWEST，它的优先级就会降到 7。相反，两个走向极端的定义——15 和-15——则会分别把一个线程的优先级设定成那一类能够使用最高优先级和最低优先级。

表 14-3　用于修改线程当前优先级的线程优先级定义

线程优先级定义	值	说　明
THREAD_PRIORITY_TIME_CRITICAL	15	为空闲、普通以及高优先级指定一个 15 级的基准优先级，并且为实时类指定一个 31 级的基准优先级
THREAD_PRIORITY_HIGHEST	2	指定当前优先级在普通优先级的基础上上浮 2 级
THREAD_PRIORITY_ABOVE_NORMAL	1	指定当前优先级在普通优先级的基础上上浮 1 级
THREAD_PRIORITY_NORMAL	0	把当前优先级直接指定成普通优先级
THREAD_PRIORITY_BELOW_NORMAL	-1	指定当前优先级在普通优先级的基础上下浮 1 级
THREAD_PRIORITY_LOWEST	-2	指定当前优先级在普通优先级的基础上下浮 2 级
THREAD_PRIORITY_IDLE	-15	为空闲、普通以及高优先级指定一个 1 级的基准优先级，并且为实时类指定一个 16 级的基准优先级

表 14-3 内列出的数值可以根据 WINNT.H 里的其他一些定义来进行判断，这些定义列于表 14-4 内。出于对简化和可读性的考虑，我把表 14-3 里“值”一栏内的定义用它们各自的数字取代了。

为了对这些值进行总结，大家可以参考表 14-5。这张表对 1 到 32 这个范围内的不同优先级别进行了归纳。空闲（Idle）、普通（Normal）以及 High（高）类的范围从 1 到 15，而时间关键（Time critical）类是占据了上端剩余 16 个编号的唯一优先级类。

表 14-4　用于修改线程优先级的偏移值

优先级偏移	值	说　明
THREAD_BASE_PRIORITY_LOWRT	15	把当前的线程优先级向上提升 15 级
THREAD_BASE_PRIORITY_MAX	2	把当前的线程优先级向上提升 2 级
THREAD_BASE_PRIORITY_MIN	-2	把当前的线程优先级向下降低 2 级
THREAD_BASE_PRIORITY_IDLE	-15	把当前的线程优先级向下降低 15 级

表 14-5　Win32 的类优先级

线程优先级	空闲	后台普通	前台普通	高优先级	时间关键
时间关键	15	15	15	15	31
最高	6	9	11	15	26
在普通以上	5	9	10	14	25
普通	4	7	9	13	24
在普通以下	3	6	8	12	23
最低	2	5	7	11	22
空闲	1	1	1	1	16

为一个线程分配优先级的时候，还要考虑到一些新的问题。缺省情况下，对于属于普通类的一个 Win32 进程来说，它的优先级别被设置成 7。我敢保证许多人都觉得这种设置对代码的创造性、健壮程度以及功能性来说都是一种真正的限制。

对于那些想让计算机系统对自己的程序施展全力进行配合的开发者来说，把代码转移至“时间关键”类优先级无疑是一种强烈的诱惑。然而，假如真的这样做了，那么这也许是我们在自己的应用程序里能够实行的一种最糟糕的处理了。把整个进程都移至“时间关键”类以后，整体系统就会变得相当敏感和脆弱，很容易就会滑向不可逆转的深渊。可怜的调度程序继续排外地运行那些具有最高优先级的线程，禁止其他任何一个线程访问 CPU。系统本身就这样不可避免了挂起来了，这是导致系统响应缓慢的最主要的原因。

一个 Win32 进程和它的线程只应该放置于 1－15 的范围以内，只有当情况非常特殊的时候，同时所需时间极短的前提下才能考虑打破这个障碍。我们在这儿的想法是编写多线程代码，其中的每个线程都能完成一种特定的任务。正如我们以后会发现的那样，多线程应用程序的一个关键性问题在于线程间的同步。

对于一个线程来说，它经常都需要停下来，等待其他线程中断一个作业或者释放了系统资源（比如一个通信端口）的时候才能启动。在这种情况下，我们推荐正在给系统施加压力的那个线程尽可能快地中断。假如一个线程当前正运行于优先级 15，那么这就是一种最佳的方案。通过临时性升高这个线程的优先级，我们可以确保相关的关键性任务能够在极短的时间内完成，这样就增强了程序的整体性能。一旦任务结束以后，线程要么回到它最初的优先级别，要么挂起，要么就中断。即使在这种情况下，15 级也应该考虑成标准应用程序里所有线程不可逾越的一个关卡。

对于普通类的一个成员来说，它也提供了某些附加的好处。正如我们以前的表格里指出的那样，当作一个前台应用程序运行的时候，系统会自动提高相应线程的优先级别。基准优先级将从七级提升到九级，尽管这是一种幅度很小的提升，但这却能为用户的请求提供更有效的响应。系统实现了其他一些形式的动态提升，从而增强了系统整体的响应敏感度。

接下来，假设一个高优先级的线程想要访问某种资源，但是这种资源现在是被一个低优先级的线程占用的。从常规的眼光来看，这种情况是令人非常失望的，因为考虑到两个原因。首先，由于低优先级的线程本身的状态，所以它很少有机会得以执行。这真是一件憾事，然而最遗憾的还要算高优先级的那个线程了。尽管它具有相对高的特权，但在这个时候也只好干瞪眼，不得不进入“挂起”状态。它访问早已被别人抢占的资源的机会看来很渺茫，至少这不是一时半会儿能做到的事情。

除此以外，作为一个高优先级的线程，同时意味着需要这个线程完成的任务比较紧迫。但是由于需要的资源被一个低优先级的家伙占用了，所以这种任务短时间以内也是无法完成的。调度程序在侦测这类事件上的“智能”还是足够高的，并且能够相应作出一些比较聪明的决定。首先，低优先级线程的优先级将动态地提高，直到提高到与挂起线程匹配的一个相同的优先级上。这样就显著地增大了它得以执行的机会，同时也就缩短了那个高优先级线程的等待时间。每当线程访问一次 CPU，提升的优先级就会临时性地降低一个级别。

优先级自动提升的第二种形式是前台窗口有关。除了进入前台以后提升的两个级别以外，线程的优先级还必须等于或者高于正在后台运行的那个具有最高优先级的线程。总而言之，不要尝试永久性地把几个线程放置于高优先级的位置上面，因为无论对自己的应用程序还是对

用户来说都是没有好处的。除了“动态提升”这种完全由操作系统控件的行为以外，开发者还应该在自己编写的线程里实现临时性的优先级提升，从而更好地利用 CPU 的潜能，并且尽可能以最短的时间完成某些关键性的任务。

对一个线程或者进程的优先级进行处理的时候，我们必须分别用到 GetThreadPriority () 和 GetClassPriority () 这两个函数。

14.6.2 提高第二个线程的优先级

Priorities 菜单项引入了如图 14-21 的那个对话框，用户可以在其中选择第二个线程使用的优先级。组合框内列出了所有可能的值，这些值与表 14-5 列出的定义是对应的。正如大家盼望的那样，我选择的最高优先级对应于 1 到 31 这个范围内的 15 级。现在让我们建立控件，然后观察作出这种改动后造成的结果。哎呀！从图 14-22 的显示来看，情况可有些不妙！

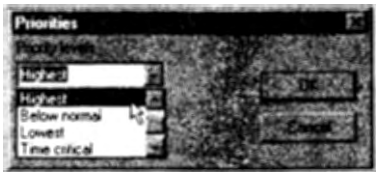


图 14-21 选择第二个线程使用的优先级

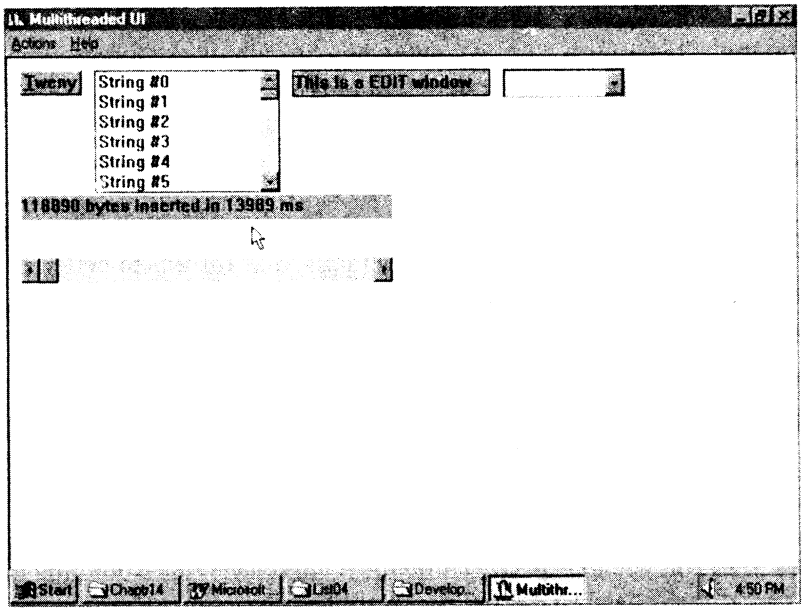


图 14-22 提高第二个线程的优先级别并没有改进程序的性能

实际上，由于进行了这种变动，我们耗费的时间在原来的基础上又增加了 1.5 秒。我们也许能够考虑的另外一种选择是同时增大两个线程的优先级别。从总体上看，这好象并不是一种没有根据的想法，因为两个线程之间必须不停地进行通信。如果只是把第二个线程增大

到更高的优先级别，它就有机会得到更好的执行。但是，这同时对主线程造成了危害，使主线程很少有访问 CPU 的机会。

Priorities 对话框内包含了 Main thread too（同时作用于主线程）核选框，它的作用是指示应用程序为主线程和第二个线程分配相同的优先级。为这两个线程都分配了最高的优先级以后（它们现在共享相同的优先级），接下来让我们看看是否降低了整体耗费的时间。如图 14-23 所示，其中显示的结果是让人振奋的，尽管这算不上一次显著的性能提升。

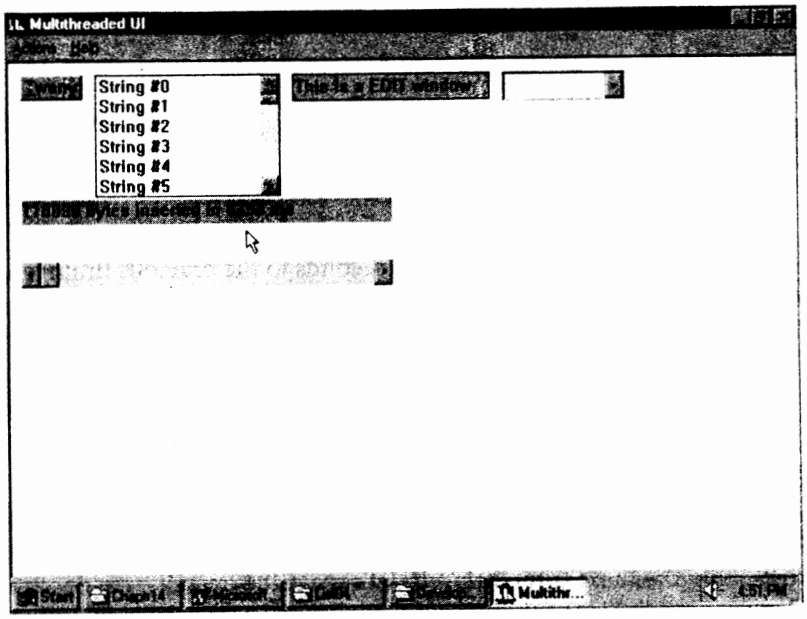


图 14-23　通过同时提高两个线程的优先级，耗费的时间可以降低至 8 秒钟以下

14.6.3　情况 B：让第二个线程包揽一切

我们在这儿的考虑是怎样避免由于消息穿越线程进行传递而带来的系统开销。针对这种想法，我们把列表框窗口的建立工作放到第二个线程里进行。从程序设计的角度来看，我们对此只需要进行少量的干涉即可。其中的关键问题是要在第二个线程里也增加一个消息循环，否则列表框窗口就没有办法接收到用户与应用程序进行交互作用后生成的消息。随后，第二个线程将建立一个列表框窗口，同时把主窗口当作自己的父窗口。这就证明了我们可以把属于不同线程的窗口混合起来，然后通过一种强有力的“父子”关系把它们连结到一起。随后，10000 个正文串将增加到列表框内，而线程的代码则是通过编写一个标准的消息循环来完成的。

```
...
LRESULT WINAPI SecondThread (LONG lParam)
{
    char szText [30];
    HWND hwndList, hwnd;
```

```
PDATA pdata = (PDATA) lParam;
HWND hwndStatic;
register int i, iTot = 0;
int iPrty;
DWORD dw1, dw2;
DWORD dwThreadID;
MSG msg;

// main window hwnd
hwnd = pdata -> hwnd;
dwThreadID = pdata -> dwThreadID;
hwndStatic = CTRL (hwnd, CT_STATIC);
hwndList = CTRL (hwnd, CT_LISTBOX);

// get the current thread priority
iPrty = GetThreadPriority (GetCurrentThread ());

// raise the thread priority
SetThreadPriority (GetCurrentThread (), pdata -> iPrtyLevel);
iPrty = GetThreadPriority (GetCurrentThread ());

// check if we have to create the listbox locally
if (pdata -> fSecond)
{
    hwndList = CreateWindow (" listbox", NULL,
                             WS_CHILD | WS_VISIBLE | LBS_
                                 STANDARD & ~LBS_SORT,
                             70, 10,
                             150, 100,
                             hwnd,
                             (HMENU) CT_LISTBOX,
                             pdata -> hInstance,
                             NULL);
    // save the listbox handle
    pdata -> hwndListBox = hwndList;
    // DON'T set focus sulla LISTBOX
}
dw1 = GetTickCount ();
```

```
        // filling the listbox
        for (i = 0; i < LIMIT; i++)
        {
            iTot += wsprintf (szText, " String #%d", i);
            if (pdata -> fFill)
                ListBox_AddString (hwndList, szText);
        }
        dw2 = GetTickCount ();

        // inserting in the STATIC window
        wsprintf (szText, "%d bytes inserted in %d ms", iTot, dw2 - dw1);
        SetWindowText (hwndStatic, szText);
        InvalidateRect (hwndStatic, NULL, TRUE);

        if (pdata -> fSecond || pdata -> fBest)
        {
            while (GetMessage (&msg, NULL, 0, 0))
            {
                TranslateMessage (&msg);
                DispatchMessage (&msg);
            }
        }
    }
...
```

CLASSES 程序允许我们对情况 A 和情况 B 下面的所有选项进行测试，只需从 Actions 顶级菜单内选择一些对应的菜单项就可以了。对上面这段程序进行检查，我们发现第二个线程已经包揽了一切，它指示应用程序不要在主线程里建立列表框控件，而是把这种工作转移到第二个线程里进行。图 14-24 向大家展示了第二种方法的结果。屏幕显示是在列表框还没有完全填写好的时候抓取下来的。就在这个时候，我们在 EDIT 控件上方单击了鼠标右键，从而显示出了它的关联菜单。

这种方法唯一的缺点是列表框在所有项目都已经插入之前不会有任何显示。这一过程在用户选中了 Create controls 菜单项以后会持续大约 3.5 秒的时间，如图 14-25 所示，这个时间稍微低于单线程方案需要的时间。然而，假如我们在第二个线程里设置了对 SetFocus () 函数的一次调用，这种调整过后的机制就会失败。对这个函数的调用将禁止用户与其他控件的交互作用，这样就抵消了由应用程序多线程本质带来的任何好处。

列表框内一个正文串都没有显示出来，这并不是设计上的一种失误，然而这确实令人相当失望。事实上，我们在第 9 章“预定义窗口类”里提到的那个 CLASSES 单线程版本里也会发生同样的情况。如果使用的是单线程，LISTBOX 类窗口在所有正文串的填写操作都结束之

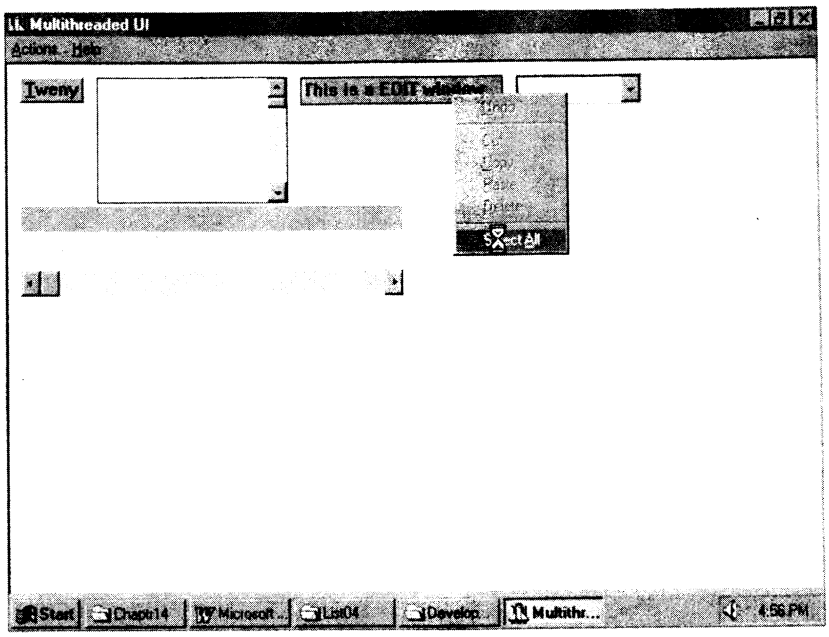


图 14-24　在第二个线程里对列表框控件的建立和填写
不会影响到用户与其他组件的交互作用

前不会产生任何输出。针对这个问题，我能想到的唯一技巧就是执行一些迭代运算，比如先显示出前面的 10 个或者 20 个正文串，然后用第二个循环完成其余 990 或者 980 个正文串的输出。对于用户来说，他（她）能够理解某个控件正在忙于处理一些数据，并且除非填写工作全部结束，否则就不能与这个控件进行交互作用。

在我们刚才描述的情况下，线程间方式提供了最好的结果。事实上，我们可以在第二个线程填写列表框的时候对其进行滚动操作。在这儿要强调的一点是这种限制只与 LISTBOX 类有关，并不是整个系统范围内的限制。例如，假设我们需要把通过某种 ODBC 机制从一个 SQL 数据库内取得的数据填写到一个列表框里。这时，我们不是显示出列表框，而是建立它并保持其不可见状态。在这同时，我们在屏幕上显示一个静态（STATIC）类窗口，用它显示当前从数据库里读取出来的记录。这就意味着用户仍然可以与应用程序的其他组件进行交互作用，同时还能获得数据库查询进程的可视反馈信息。

Create controls 菜单项可以在 THREADS 的执行过程中多次选择。与它对应的代码段会检查控件是否已经在以前的试验中建立好了，并且尝试用 DestroyWindow（）消除建立好的控件。这个函数只能作用于调用它的那个相同线程里的窗口。因此，假如列表框是在独立的线程里建立的，应用程序就会调用 TerminateThread（）来中断那个线程，这个操作同时也会导致列表框控件的消除。

最后需要注意的一个问题是：SetMenuItemInfo（）和 GetMenuItemInfo（）这两个函数在 MS Windows NT 3.51 环境下根本不起作用，即使安装了最新的 Service Pak 亦是如此。因此，为了测试出在一个纯粹 32 系统下的最佳方案，我不得不稍微修改了一下源代码。请大家参考下表：

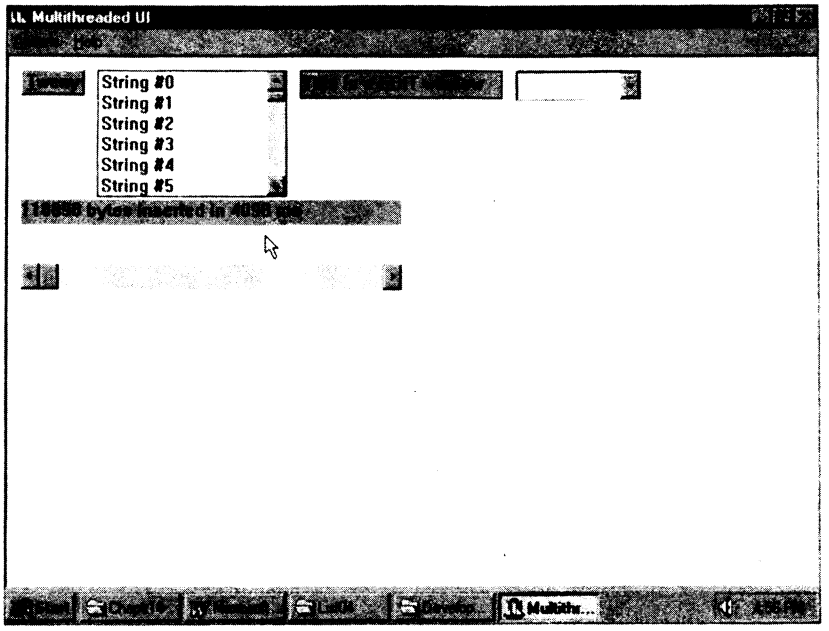


图 14-25　列表框的填写耗费了与单线程方案差不多的时间，
这个结果比以前的多线程方案要好得多

CLASSES 程序	NT	Win95	两个系统之间的开销
单线程版本	1.537	3.708	241%
双线程版本	3.845	8.162	212%
第二个线程包揽一切	1.872	3.544	189%
最佳方案	13.880	10.823	-77%

通过比较可以发现，假如运行的是一个纯粹的 32 位程序，NT 不失为最好的一种操作系统，只是最后一个数据指出 STATIC（静态）类是按照一种很拙劣的方法实现的。在第二个线程里，假如列表框的建立与填写算法同时存在，那么 NT 花费的时间几乎只有 MS Windows 95 的一半。这个时间与单线程方案都是有一比的。

从本质上说，要想成为最佳的方案，必须保证列表框窗口或者其他任何控件都存在于相同线程的内部，这个线程能对自己的所有 I/O 操作进行控制。这种方案提供了最佳的程序性能以及用户响应敏感度。

14.7　窗口和线程

为了把窗口与线程间接性地关联起来，另外一种方法是调用 API 函数 EnumThreadWindows ()。这个函数能列出属于某个特定线程的所有窗口，当前，前提是要先给出那个线程的 ID。这个函数采用的方案与 Windows 的列举函数是一致的。它要求使用一个线程 ID，并在后面接上由开发者定义的某个函数的址，以便在找到一个窗口后对这个自定义函数进行自动调用。

#include〈winbase.h〉

BOOL EnumThreadWindows (DWORD dwThreadId, WNDENUMPROC lpfn, LPARAM lParam);

参数	说明
DWORD dwThreadId	准备对其中的窗口进行列举的某个线程的 ID
WNDENUMPROC lpfn	由应用程序定义的某个函数的址，属于那个线程的某个窗口被发现以后，就调用那个函数
LPARAM lParam	由应用程序定义的值，它将传递给由开发者定义的函数
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；否则返回一个 FALSE 值

寻找到一个窗口以后，系统就会自动调用由应用程序定义的回调函数，同时传递窗口句柄以及 lParam 参数的值。我们根据自己的意愿为列举函数任意分配一个名字，比如我们在下面使用的 EnumThreadWndProc ()：

BOOL CALLBACK EnumThreadWndProc (HWND hwnd, LPARAM lParam);

假如返回 TRUE 值，迭代工作就会继续进行；假如为 FALSE，函数就会中断。为了判断属于某个线程的窗口总数，我们必须使用一个局部计数器。一旦获得了对某个窗口句柄的访问权，我们就可以取得大量有关的信息，本书的几个例子都向大家阐述了这一点。

14.8 IPC 机制

我们在本章前面几次都曾经提到过这样一个问题：线程的使用是与由 Win32 提供的 IPC 机制严格相关的。IPC 的英语“Inter-Process Communications”的简称，即“进程间通信”的意思。对于一名开发者来说，针对这个问题的基本考虑是让所有线程互相之间协调运行、保护共享资源以及对系统设备或者内存区域进行同步访问。针对不同的任务，Win32 提供了下面这些 IPC 工具：

- ▶信号机：在这儿，“信号机”(Semaphore) 可用于限制同时访问一个计数器的数目，这个计数器是一种共享资源。在第四章“消息和重画模式”里，我们已经讨论过怎样利用一个信号来限制同时执行的实例数量。因此，我们应该把信号想象成一种最高级别的全局计数器，而不是现实生活使用的只有两三种状态的那种真正的信号机。
- ▶MUTEX：MUTEX 是一种 IPC 机制，对于现成的所有线程来说，它们将相互排斥地访问一个共享资源。也就是说，同时只能有一线程能取得对资源的访问权。在一个共享内存块里进行数据写入就是 MUTEX 的一个典型应用场合。在这儿，MUTEX 这个词起源于“MUTually EXclude”，即“相互排斥”的意思。
- ▶事件：需要对整个或者多个线程的行动进行协调的时候，事件就显得相当有用了。线程可以通过 CreateThread () 函数在挂起模式下建立。然而这样一来，以后就没有其他办法可以重新挂起了，除非调用一个来势汹汹的 SupendThread () 函数。事件提供

了另外一种更有效的办法来实现一个总是处于活动状态的线程，然而实际只有在有必要的时候才执行事件。例如，应用程序启动的时候，假设我们准备建立多个线程，但是只执行其中的少数几个。这样一来，某些线程就会挂起，等待一个事件产生，因此进入了一种有效的等待状态。事件产生以后，等待的线程就可以自由地执行了。

▶临界区：临界区（Critical Section）是一种非常简单的IPC机制，它可以对多线程环境中的一系列数据进行保护。假如读者具备OS/2编程经验，那么最好忘记那种环境中使用的“临界区”。在这儿，由Win32实现的临界区已经完全不同了。

针对这些工具，我们将在下面进行更详尽的讨论。

14.8.1 对信号机的理解

和其他机制一样，IPC机制也具备两种状态：发出信号和未发信号。我们可以把两种状态想象成一名水手正在用旗语与其他船只通讯。当旗子举起，其他人可以看见的时候，我们就说这名水手发出了信号；旗子放下以后，他显然处于“未发信号”状态。所有Win32的等待函数，比如WaitForSingleObject ()和WaitForSingleObjectEx ()都会在信号机发出信号以后返回（或者预定的时间超出以后返回）。对于多对象等待函数来说，比如WaitForMultipleObjects ()和WaitForMultipleObjectsEx ()，它们的运作具有高度的灵活性。这些函数可以在指定线程中的一个或者全部发出信号以后返回。

一个信号机的发出信号和未发出信号状态取决于信号计数器：假如计数器大于零，它就发出信号；假如等于零，就不发出信号。只有建立了同步对象以后，我们才能设置相应的一个信号计数器，这方面是有严格限制的。我们可以调用CreateSemaphore ()函数来建立一个信号机，该函数可以返回一个信号机句柄。

```
#include 〈winbase.h〉
HANDLE CreatSemaphore (LPSECURITY_ATTRIBUTES lpSemaphore
    Attributes,
                       LONG lInitialCount,
                       LONG lMaximumCount,
                       LPCTSTR lpName);
```

参数	说明
LPSECURITY_ATTRIBU TES lpSemaphoreAttributes	包含了安全描述符的一个安全属性结构的地址。在MS Windows 95里没有实现。
LONG lInitialCount	最初的信号机计数
LONG lMaximumCount	信号机计数的最大值
LPCTSTR lpName	信号对象的名字（可选）
返回值	在正文里讨论
HANDLE	信号机句柄；假如函数调用失败，则返回一个NULL值

信号机的建立是相当简单的。在 MS Windows 95 里实现的 API 函数 CreateSemaphore () 要求用到两个数值，分别对应于初始化信号机计数以及它的最大值。这两个数字可以全部设置成零，但更常见的一种情况是第二个值大于或者等于第一个值。下面两个算法表达式综合了计数值。

```
0 <= lInitialCount <= lMaximumCount
lMaximumCount > 0
```

为了对信号进行标识，我们可以为它分配一个名字，任何有效的正文串都足以对它进行描述。显然，只有需要从几个进程里访问这个信号机的时候才有必要为它分配一个名字。假如信号机只是作为某个进程内部的一种 IPC 机制来运作，就没有必要用名字对它进行标识。信号机的句柄已经唯一性地对它进行了标识，进程内的每个线程都可以对其进行访问。请注意，诸如信号机的对象只有在多线程的应用程序里才有用，或者在几个进程需要用到某种形式的 IPC 的时候才能发挥作用。

根据前面的假设，下面摘录的这个代码段可以建立一个发出信号的信号机，并且只能在建立它的那个进程里对其进行访问：

```
...
hsem = CreateSemaphore (NULL, 1, 3, NULL);
...
```

相反，为了建立一个未发出信号的信号机，我们可以采取如下的形式：

```
...
hsem = CreateSemaphore (NULL, 0, 1, NULL);
...
```

尽管提供了 OpenSemaphore () 这个 API 函数，但是假如以前已经用 CreateSemaphore () 建立了一个取好名字的信号机，那么以后用 CreateSemaphore () 仍然可以很好地完成对信号机的访问工作。在这种情况下，CreateSemaphore () 的行为与 OpenSendMessage () 没有什么两样，都能返回指定信号机的句柄。对于实际建立信号机的 CreateSemaphore () 函数以及只用于打开一个现成信号机的 CreateSemaphore () 函数来说，为了对它们进行区分，我们必须调用 GetLastError () 函数。在第二种情况下（打开信号机），GetLastError () 会返回一个 ERROR_ALREADY_EXISTS，从而证明现在访问的是一个现成的信号机。这种技术在本书附带 CD 的 Listing 7.7 里已经得到了实现，它允许执行应用程序的一个单一拷贝。名为 JUSTONE 的信号机是通过把初始计数值设置成 1 来实现的，以后这个值不能再增大。执行了另外一个实例以后，CreateSemaphore () 函数就会返回一个有效的句柄。然而，为了测试同一程序另外一个实例的存在，我们检查 GetLastError () 的返回值就足够了。

```
...
// creating the semaphore
hsem = CreateSemaphore (NULL, 1, 1, " JUSTONE");
if (GetLastError () == ERROR_ALREADY_EXISTS)
{
    ...
```

如果想中断和关闭一个信号机的句柄，我们需要用到常规的 CloseHandle () 函数。

现在让我们重新对 Listing 7.7 的 MAXINST 示范程序进行检查，该程序可以把同时运行的实例数目限制在三个以内。在这种典型的环境下，一个已经命名的信号机直到了关键性的作用。在这儿，信号机计数最初设置成零，最大的数字则设置成 3。

```
...
hsem = CreateSemaphore (NULL, 0, MAXINSTANCES, " MAXINST");
    if (! ReleaseSemaphore (hsem, 1, &lCnt))
...
```

在这个例子里，信号机没有发出信号，然而应用程序并未把这种信息考虑在内。这段代码的目标是利用 API 函数 ReleaseSemaphore () 使计数值增一。这个函数可以按照指定的增幅增大一个信号计数值，同时在当前值增大之前指出这个值的大小。

```
#include 〈winbase.h〉
BOOL ReleaseSemaphore (HANDLE hSemaphore,
                       LONG cReleaseCount,
                       LPLONG lplPreviousCount);
```

参数	说明
HANDLE hSemaphore	信号机的句柄
LONG cReleaseCount	大于零的正增幅值
LPLONG lplPreviousCount	一个长整数的地址，可以用它接收以前的计数值；如果不需要这种信息，亦可设置成 NULL
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

如果增加到当前计数值上的增幅值超出了信号机设置的最大值，ReleaseSemaphore () 函数就会返回一个 FALSE。正是由于这种原因，应用程序才可以利用一个信号机来限制对一个共享资源进行访问的线程或者进程的数目。每个线程都将发出对 ReleaseSemaphore () 函数的一次调用，从而增大计数值，然后对返回值进行仔细的测试就可以了。

一个 FALSE 返回值表明最大的数字已经达到了，这样就能禁止那个线程进行相关的操作。这也是我们应该掌握的另外一个重要概念。信号机以及其他所有 IPC 对象都只是简单地当作一种共享资源的触发器或者控制器来使用，但是信号机和其他 IPC 对象之间根本没有任何关系。例如，根据应用程序的逻辑，假设最多只允许四个线程同时访问一个共享内存块。这四个线程的每一个都会发出对 ReleaseSemaphore () 的一次调用，从而使计数值增一。所有四个线程都完成了这种操作以后，应用程序里其他任何一个线程对 ReleaseSemaphore () 发出的调用都会无效。

一个线程中断了对共享内存块的写入或者读取以后，它可以使信号机的计数减一，从而为其他线程留下空间。没有任何一个专用的函数可以完成这一工作，但是我们可以用普通的API函数WaitForSingleObject（）来达到相同的目的。在“等待”一个信号的时候，这个函数简单地使计数值减1。计数值减到0以后，就恢复到未发出信号的状态。只有当函数计数值为零以后，这个函数才会中止当前进程的执行。尽管计数值每一次只能减小固定的数值，但是ReleaseSemaphore（）可以用任何幅度增大当前的计数值，只要最后得到的值不会超出预先设定好的最大值就行。

和其他同步对象相反，我们不用经常等待信号，因为它们的基本任务就是守卫一扇大门——只要有空闲的房间，任何人都可以进入。假如没有多余的房间了，这扇门就会关掉，门外的任何人都需要依次排队等候。

对信号机进行处理的第三个也是最后一个函数是OpenSemaphore（）。通常，这个函数是在访问一个已命名对象后由某个独立的进程调用的。在这儿，必须先为信号机分配一个名字，否则除了建立这个信号机的那个进程以外，其他进程都无法对其进行访问。

```
#include〈winbase.h〉
HANDLE OpenSemaphore（DWORD fdwAccess，BOOL fInherit，LPCTSTR lpszSemName）；
```

参数	说明
DWORD fdwAccess	既可以是SEMAPHORE_ALL_ACCESS，也可以SEMAPHORE_MODIFY_STATE，用于指定访问标志
BOOL fInherit	假如为TRUE，就表明子进程可以对信号机句柄进行继承
LPCTSTR lpszSemName	信号机的名字
返回值	在正文里讨论
HANDLE	信号机的句柄；假如函数调用失败，则返回一个NULL值

总而言之，只有三个函数是专门针对信号机的建立和管理而设计的，另外还有两个通用的函数可以进行一些辅助的处理：CloseHandle（）以及WaitForSingleObject（），请大家参考表14-6。

表14-6　使用信号机时涉及到的函数

函　数	说　明
CreateSemaphore	建立或者打开一个信号机
ReleaseSemaphore	增大信号机计数值，把它转换成发出信号状态
OpenSemaphore	打开一个信号机，同时返回它的句柄
CloseHandle	关闭一个信号机
WaitForSingleObject	使信号机的计数值减一

14.8.2 MUTEX 的管理

假如应用程序必须让某种特定资源同时只能由一个单一的线程访问或者拥有，那么这时能够使用的最恰当的 IPC 就是一个 MUTEX。MUTEX 是可以由一个线程拥有的唯一 IPC 对象。

请大家考虑下面这个例子：假设两个线程从属于相同或者不同的进程，它们准备对一个通信端口进行竞争，并且一次只允许一个线程在那个设备上面执行 I/O 操作。我们应该怎样保护通信端口，从而防止潜在的冲突呢？正如大家也许已经猜到的那样，使用 MUTEX 将是这种情况下的最佳选择。只有拥有某个特定 MUTEX 的线程才允许对通信端口进行访问，另外一个线程就不得不通过 WaitForSingleObject（）进行耐心的等待。一旦时机成熟，正在等待的那个线程就会变成 MUTEX 的拥有者。只有当前的拥有者通过 ReleaseMutex（）释放了手中的 MUTEX 以后，这种 MUTEX 的交接才有可能发生。因此，我们很容易就可以判断出一个 MUTEX 与它的状态有关的行为。假如没有被拥有，它就会发出信号；一旦被某个线程拥有，就停止信号的发送。假如程序指定了一个已被拥有的 MUTEX，未发送信号的状态就证明另外一个线程正在对 WaitForSingleObject（）的调用里处于挂起状态。和信号机类似，MUTEX 既可以对同一进程内多个线程的行动进行协调，也可以对不同进程间的多个线程进行协调。

如果想建立一个 MUTEX，我们可以调用 CreateMutex（）函数，如下所示：

```
#include〈winbase.h〉
    HANDLE CreateMutex (LPSECURITY_ATTRIBUTES lpMutexAttributes,
                        BOOL bInitialOwner,
                        LPCTSTR lpName);
```

参数	说明
LPSECURITY_ATTRIBUTES lpMutexAttributes	一个 SECURITY_ATTRIBUTES 结构的地址，在 MS Windows 95 里应该设置成 NULL
BOOL bInitialOwner	假如为 TRUE，表明建立它的线程立即就能拥有它
LPCTSTR lpName	MUTEX 的名字；假如只是在同一个进程内使用，就设置成 NULL
返回值	在正文里讨论
HANDLE	MUTEX 的句柄；假如函数调用失败，则返回一个 NULL 值

建立一个 MUTEX 的时候，我们可以把第二个参数设置成 TRUE，从而使当前进程立即获得所建 MUTEX 的拥有权。这是我们经常采取的一种处理；否则，为了请求获得一个 MUTEX 的拥有权，唯一的方式就是使用某个等待函数。MUTEX API 函数的使用情况与我们以前介绍的信号机 API 差不多。除了能建立一个新的 MUTEX 以外，CreateMutex（）还能用于打开一个现成 MUTEX。在这种情况下，假如 GetLastError（）返回的值是 ERROR_ALREADY_EXISTS，就表明目标 MUTEX 已经在系统内建立好了。

OpenMutex（）函数提供了另外一种可选的方案：

```
#include〈winbase.h〉
HANDLE OpenMutex（DWORD fdwAccess，
                  BOOL fInherit，
                  LPCTSTR lpszMutexName）；
```

参数	说明
DWORD fdwAccess	访问选项：MUTEX_ALL_ACCESS
BOOL fInherit	假如为 TRUE，就表明句柄可以由其他进程进行继承
LPCTSTR lpszMutexName	MUTEX 的名字
返回值	在正文里讨论
HANDLE	MUTEX 的句柄；假如函数调用失败，则返回一个 NULL 值

某个进程调用 OpenMutex（）函数来获得对以前建立的某个 MUTEX 的访问权的时候，必须在参数里提供正确的对象名称，并且指明是否允许子进程对那个句柄进行继承。在 MS Windows 95 里，访问选项只能选择为 MUTEX_ALL_ACCESS。这样一来，假如某个进程去打开当前正由另外一个线程拥有的 MUTEX，那么会出现什么情况呢？这个函数是成功地返回，还是调用失败呢？事实上，尽管目标 MUTEX 当前正在由另外一个线程拥有，只要我们设置的参数正确，OpenMutex（）函数仍能正常返回一个有效的句柄。这个函数的基本任务是为对象提供一个有效的句柄，如此而已，以外并没有什么特别的地方。

假如 MUTEX 已经由另外一个线程拥有，那么以后假如为了指定那个 MUTEX 的句柄，从而发出对 WaitForSingleObject（）的一次调用，就会根据设定的等待条件，从而进入相应的等待状态。这儿的一个关键问题是要理解第二个进程怎样知道 IPC 资源的名字，这儿的 IPC 资源就是 MUTEX，但是同样也可适用于其他类似的对象。从根本上说，有两个办法可以穿越两个或者多个进程与某个对象的名字进行通信。第一种最也是最常用的一种办法就是在两个进程的源代码内强行编入对象的名字，我们称之为“硬编码”。例如，假设已经建立了一个名为 STEFANO 的 MUTEX 对象。如果应用程序是由同一个人或者同一个开发小组开发的，STEFANO 这个正文串就会在几个模块中出现，并且作为一个常数运用。

第二种也是比较少用的一种方案是依靠另外一种 IPC 机制的存在，从而把对象名从一个进程传递给另外一个。这种方法显得有些不便，因为它使一个 Win32 应用程序的整体结构变得复杂起来，所以带来的好处很有限。假如我们希望与其他进程建立动态的联系，就应该使用一些更可靠、更有效的手段，比如 DDE 或者 OLE 等等。这样就可以独立开发出不同的组件，然后通过这些手段把它们集成到一起。

解除线程对一个 MUTEX 的拥有是通过 ReleaseMutex（）函数来实现的：

```
#include〈winbase.h〉
BOOL ReleaseMutex（HANDLE hMutex）；
```

这个函数唯一要用到的参数就是一个有效的 MUTEX 句柄。假如调用成功，

ReleaseMutex () 就会返回一个 TRUE 值；假如失败，则返回一个 FALSE。如果线程没有获得对指定 MUTEX 的拥有权，或者 MUTEX 的句柄由于某种原因失效了，函数的调用就不能成功地进行。一旦脱离了某个线程的控制，MUTEX 就会变成发出信号状态，以前调用 WaitForSingleObject () 时挂起的另外一个线程随即就可以对它进行访问了。

考虑到 MUTEX 在系统里的这种控制方式，我们可以想象到一种有趣的情况：假如拥有一个或者多个 MUTEX 的线程出于某种原因中断并消失了，这时会出现什么后果呢？在这个时候，很有可能还有其他线程正在耐心地等待自己成为 MUTEX 的拥有者。当物主线程突然中断以后，正在等待那个被物主线程拥有的 MUTEX 的所有函数都会自动返回，同时带回一个 WAIT_ABANDONED 值。这个值将指示当前的线程继续执行，但这并不是由于获得了对某个 MUTEX 的拥有权而造成的，而是由于应用程序里出现了某种没有预料到的错误而造成的。因此，我们强烈建议大家随时保持对所有等待函数返回值进行的测试，从而谨慎地判断导致函数返回的真正原因。

恢复那个已中断线程的运行看起来并不是很好的一着棋。事实上，由于对 MUTEX 进行控制的那个线程的突然“死亡”，当前线程也许会碰到一些不愿意看到的情况。因此，程序中断以后立即进行一次清除操作应该是一种比较明智的作法。

执行一个等待函数以后，它可以把 MUTEX 的状态改变成未发出信号状态。表 14-7 为我们总结了与 MUTEX 有关的所有函数。

表 14-7　与 MUTEX 的建立和控制有关的函数

函　数	说　明	函　数	说　明
CreateMutex	建立一个 MUTEX	ReleaseMutex	解除对一个 MUTEX 的拥有权
OpenMutex	访问一个 MUTEX	CloseMutex	关闭 MUTEX

14.8.3　利用事件使线程同步

为了使某个 Win32 进程以内或者几个线程之间的各个线程进行同步处理，我们需要用到“事件”(Event) 对象。事件的行为类似于一种信号，它可以通知其他线程某个特定的事件已经发生。事件主要用于动态地挂起以及恢复一个或者多个线程的执行，利用它可以构成应用程序的基本设计思路和程序逻辑。

在本章开始的时候，大家已经看到在 CreateThread () 函数的参数里增加了 CREATE_SUSPENDED 标志以后，可以令一个线程挂起。尽管这种方法相当简单和有效，然而最好还是把它的执行与“事件”对象的状态约束在一起，这样才能使其一直处于应用程序的控制下，避免出现某些不希望的意外情况。

通常，当我们建立了一个线程以后，一般都要令其在一段不定的时间内保持“昏睡”状态，只有在需要的时候才“唤醒”它。这种操作可以通过一个事件对象来简便地实现。在建立线程之前，我们可以先建立一个事件，将其设定成未发出信号状态，从而使对其进行监视的任何等待函数都处于挂起状态。新线程最开始的指令之一就是发出对 WaitForSingleObject () 函数的调用，使其把线程挂起，直到事件对象发出信号为止。

现在让我们考虑下面这个例子。假设我们需要开发一个相当复杂的多线程应用程序。根据设计规范，运行于高优先级的一个线程会准备好所有内部数据结构，访问和读取一些注册表 (Registry) 条目，然后根据取得的信息，从而启动一个文档的载入。同时，其他线程将在

启动代码块内保持挂起状态，在主线程中断I/O操作之前，这些线程会一直保持等待状态。怎样实现这样的一个应用程序呢？我们在这儿的想法是建立一个单一的事件，把它设置成未发出信号状态，然后对所有线程进行相应的设计，使它们都针对那个普通的事件在一个等待函数里处于挂起状态。主线程完成了自己的工作以后，它就会重新设置那个事件，将其变成发出信号状态，这样就解除了对所有线程的封锁。

我们可以把事件想象成一个灯塔——在漆黑的晚上能为上百艘船指引正确的航向。我并不想为大家留下一个事件必须同时为多个线程提供服务的印象。我们可以将一个事件用于管理两个线程：应用程序主线程以及一个次级线程。

如果想建立一个事件，我们需要用到CreateEvent（）函数，该函数的语法如下所示：

```
#include〈winbase.h〉
HANDLE CreateEvent（LPSECURITY_ATTRIBUTES lpEventAttributes，
                    BOOL bManualReset，
                    BOOL bInitialState，
                    LPCTSTR lpName）；
```

参数	说明
LPSECURITY_ATTRI BUTES lpEventAttributes	一个SECURITY_ATTRIBUTES数据结构的地址，在MSWindows 95里应该设置成NULL
BOOL bManualReset	假如为TRUE，表明事件是一个人工重设对象；FALSE则表明是一个自动重设事件
BOOL bInitialState	假如为TRUE，表明事件处于发出信号状态；FALSE则表明处于未发出信号状态
LPCTSTR lpName	事件名称，这个名称在所有进程间都是可见的
返回值	在正文里讨论
HANDLE	事件句柄；假如函数调用失败，则返回一个NULL值

1．人工重设和自动重设事件

第一个参数（在MS Windows 95里通常设置成NULL）以及对象的名字是所有建立函数的通用元素，对此我们在这一章中已经讨论过多次了。除此以外，我们还必须指定事件的性质以及它的初始化状态。事件既可以人工重设，也可以自动重设。这两者之间的区别与事件从发出信号状态转换成未发出信号状态以及逆向转换的方式有关。

正如通过名字就可以推断出来的那样，除非用ResetEvent（）函数明确地设置成未发出信号状态，否则人工重设事件就会一直保持在发出信号的状态。为了恢复成发出信号状态，我们只需再调用SetEvent（）函数即可。

然而，自动重设事件的情况就会变得稍微复杂一些。假设事件处于发出信号的状态，同时有一个线程正在执行某个等待函数。毋需调用ResetEvent（）函数，WaitForSingleObject（）可以把事件自动重设为未发出信号状态。

这两种行为对于我们来说具有某些有趣的暗示。一个人工重设事件发出信号以后，正在等待的所有线程都可以恢复执行。然而，这种现象对于自动重设事件来说却并不成立。只有

执行等待函数的第一个线程才允许运行，其他所有线程仍然将保持挂起状态，因为WaitForSingleObject（）函数会把事件自动重设成未发出信号状态。表14-8为大家总结了SetEvent（）以及ResetEvent（）函数对一个人工重设或者一个自动重设事件的影响。

表14-8　事件的设置和重新设置

函　数	人工重设	自动重设
SetEvent	从未发出信号状态转向发出信号状态	从未发出信号状态转向发出信号状态，同时保持这种状态，直到另外一个线程被唤醒为止
ResetEvent	从发出信号状态转向未发出信号状态	一般都不用于自动重设事件

2. CreateEvent（）和OpenEvent（）函数

假如某个独立的进程想访问一个事件，需要用到CreateEvent（）或者更为特殊的OpenEvent（）函数：

```
#include〈winbase.h〉
HANDLE OpenEvent（DWORD fdwAccess，
                  BOOL fInherit，
                  LPCTSTR lpszEventname）；
```

参数	说明
DWORD fdwAccess	访问选项：EVENT_ALL_ACCESS
BOOL fInherit	假如为TRUE，表明句柄可以由其他进程继承
LPCTSTR lpszEventname	事件名称
返回值	在正文里讨论
HANDLE	事件句柄；假如函数调用失败，则返回一个NULL值

3. SetEvent（）和ResetEvent（）函数

接下来，让我们对SetEvent（）以及ResetEvent（）的语法进行讨论。这两个函数都只接受一个单独的参数，该参数对应于一个有效的事件句柄，同时返回一个布尔值，从而指出操作的输出情况。

```
#include〈winbase.h〉
BOOL SetEvent（HANDLE hEvent）；

#include〈winbase.h〉
BOOL ResetEvent（HANDLE hEvent）；
```

这两种行动如果用PulseEvent（）函数来完成显得更精炼。这个函数可以对信号进行设置，然后把一个事件重新设置成未发出信号状态，从而解除对一个或者多个正在等待的线程的冻结。这种差别主要取决于事件的类型。对于人工重设事件来说，正在等待的所有线程都

会释放；对于自动重设事件来说，只有一个线程会重新运行起来。

```
#include 〈winbase.h〉
BOOL PulseEvent (HANDLE hEvent);
```

表 14-9 为大家总结了事件的建立和管理所涉及到的所有函数。

表 14-9 用于事件建立和管理的函数

函 数	说 明
CreateEvent	建立事件
OpenEvent	访问事件
SetEvent	设置事件的信号状态
ResetEvent	把一个事件变成未发出信号状态
PulseEvent	设置和重新设置一个事件
CloseHandle	关闭事件
WaitForSingleObject	重新设置一个自动重设事件

最后还要提一点，无论准备建立的是哪种类型的对象，并且准备让它在自己的应用程序内同步或者发出信号，这个对象都必须在新线程启动以前建立好。

14.8.4 临界区的定义

按照 Win32 的术语定义，“临界区”(Critical Section) 代表一部分特殊的代码，它可以访问某些需要进行附加保护的应用程序数据，因为其他线程可能要对它们进行某些变动。我们可以把所有这些特殊的代码部分封装到 EnterCriticalSection () 和 LeaveCriticalSection () 之间的几个线程内，从而保证同一时刻只允许一个线程对那些数据进行访问和修改。临界段只能在一个多线程进程里使用。它们是最简单的一种 IPC 形式，这是由于它们执行的是一种相当脆弱的 IPC 同步机制。

首先，我们需要声明一个 CRITICAL_SECTION 数据结构，这个对象几乎可以当作一个进程范围内通用的信号机来使用。典型情况下，这种标识符是在任何函数以外定义的，以便使自己具备源文件范围以内的“可见性”。当应用程序开始的时候，我们发出对 InitializeCriticalSection () 函数的一次调用，从而对这个临界区的结构进行初始化。

```
#include 〈winbase.h〉
VOID InitializeCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
```

其中，唯一用到的参数是一个 CRITICAL_SECTION 数据结构的地址，这个结构是预先已经声明好的。在中断以前，或者有必要的时候，应用程序会调用 DeleteCriticalSection () 函数来释放这种资源，如下所示：

```
#include 〈winbase.h〉
VOID DeleteCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
```

API 函数 EnterCriticalSection () 可以接受一个初始化的 CRITICAL_SECTION 数据结构的地址，并且可以在内部对其进行修改。对于其他任何一个线程来说，假如它试图执行另外一个 EnterCriticalSection ()，并令其指向相同的 CRITICAL_SECTION 数据结构，这个线程就会自动地挂起，直到前一个线程执行了相应的 LeaveCriticalSection () 为止。由临界区提

供的数据保护与CRITICAL_SECTION结构根本没有一点关系。更准确地说，代码段内的所有数据都会由“进入/离开”这一对API函数(即EnterCriticalSection()和LeaveCriticalSection())分隔开，从而保护它们不会受到任何外来干涉。

14.9 Wait函数详探

贯穿本章，我们曾经数次提到了一些最常用的等待函数，比如WaitForSingleObject()等等。在这一小节内，我们准备就这一类API函数进行详细的探讨。

几乎每种内核句柄都可以成为一个同步对象，其中包括线程、进程、事件、MUTEX以及信号机句柄等等。一旦我们拥有了相应的句柄，就可以停止当前线程的执行，方法是调用WaitForSingleObject()函数。这个函数并不能百分之百地保证中断线程的执行，因为这种事件要取决于两个因素：在等待函数里设定的溢出时间以及作为基础的那个对象的状态。

```
#include〈winbase.h〉
DWORD WaitForSingleObject (HANDLE Object, DWORD dwTimeout);
```

参数	说明
HANDLE Object	对象句柄
DWORD dwTimeout	时间溢出间隔，用毫秒数表示
返回值	在正文里讨论
DWORD	假如为WAIT_FAILED，表明函数调用出现了错误；或者为表14-10列出的其中某个值

第一个参数必须是一个有效的对象句柄。接下来是用毫秒数表示的溢出时间，也可以设置成INFINITE（无限）定义，它的意思是无休止地等待下去。这个函数返回的原因有两个：要么由于设定的时间到了，或者由于目标对象已经变成了发出信号状态。事实上，还有可能出现与这两种情况都有关的第三种情况。假如把溢出时间设定为零，函数就会立即返回。这样做有什么意义吗？有！这是检查某个指定对象的一种简单、有效的办法。假如返回值等于WAIT_OBJCT_0，就意味着对象已经处于发出信号状态。相反，假如返回值为WAIT_TIMEOUT，则意味着对象处于未发出信号状态。表14-10为我们列出了这个函数的所有返回值。

表14-10 函数调用成功后，WaitForSingleObject()的返回值情况

定 义	值	说 明
WAIT_FAILED	0xFFFFFFFF	函数调用失败
WAIT_OBJECT_0	((STATUS_WAIT_0) +0)	指定的对象处于发出信号状态
WAIT_ABANDONED	((STATUS_ABANDONED_WAIT_0) +0)	拥有一个MUTEX的线程已经中断了，同时没有释放这个MUTEX
WAIT_TIMEOUT	STATUS_TIMEOUT	设定的溢出时间已经超过，对象仍然处于未发出信号状态

WaitForSingleObject () 并不是一个特别具有针对性的等待函数。对信号机进行处理的时候，它会减小信号机的计数值；对自动重设事件进行处理的时候，释放了在那个事件上挂起的一个线程以后，它会把事件重新设置成未发出信号状态。

Win32 提供了另外一种工具来挂起一个线程，这种行动是根据多个内核对象的值来完成的。WaitForMultipleObjects () 是一种相当灵活的函数，它可以接收对象控件的一个数组，这些对象可以属于相同或者不同的类型。

```
#include 〈winbase.h〉
DWORD WaitForMultipleObjects (DWORD cObjects,
                              CONST HANDLE * lphObjects,
                              BOOL fWaitAll,
                              DWORD dwTimeout);
```

参数	说明
DWORD cObjects	lphObjects 参数里的对象句柄的数目
CONST HANDLE * lphObjects	对象句柄的数组
BOOL fWaitAll	假如为 TRUE，表明函数只有在所有对象都处于发出信号状态的时候才返回；假如为 FALSE，那么只要有一个对象处于发出信号状态，函数就返回
DWORD dwTimeout	用毫秒数表示的溢出时间间隔
返回值	在正文里讨论
DWORD	假如为 WAIT_FAILED，表明函数调用出现了错误；或者为表 14-11 内列出的某个值

尽管这个函数的名字看起来相当长，但它的用法却是相当简单的，同时功能也很强。在它的所有参数里，最关键的要算第三个：fWaitAll。假如把设置成 TRUE，就表明函数只有在所有对象都处于发出信号状态的时候才返回。如果设置成 FALSE 就可以极大地简化这种工作，只要有一个对象处于发出信号状态，就会强迫 WaitForMultipleObjects () 返回。从本质上说，这个函数可以容纳动态的句柄列表，句柄的总数则会小于或者等于在 WINNT.H 里设置的 MAXIMUM_WAIT_OBJECTS 值——64。这一系列对象将在有效的等待阶段里不停地进行监视。假如把 fWaitAll 设置为 TRUE，列出的所有对象都必须同时发出信号，这样才能让 WaitForMultipleObjects () 返回，这种条件对于相应的线程来说是非常严格的。

列表里也可以包含不同类型的对象，比如信号机、MUTEX、事件以及其他内核对象。同时对多个对象进行等待是一种相当复杂的情况，根据对象的类型以及它们与等待函数的交互作用，可能产生上百种不同的组合。例如，由线程拥有的一个 MUTEX 几乎会一直会被线程拥有，只有很小及有限的、未被拥有的“自由”时期。

我们知道 WaitForSingleObject () 在返回之前可以把一个 MUTEX 改变到未发出信号状态。假设应用程序正在等待的其中一个对象就是一个 MUTEX，这个 MUTEX 已经变成了发出信号状态，而其他对象仍然处于未发出信号状态。WaitForMultipleObjects () 不会修改

MUTEX 的状态，直到剩余的所有对象都变成发出信号状态为止，这样导致的等待时间就会变得相当长。系统怎样对待这个 MUTEX 呢？它可以发出信号，并且能够成为正在等待机会拥有它的一个线程的战利品吗？答案是肯定的。对于正在同时等待多个对象的线程来说，它所耗费的等待时间肯定要长一些，因为其中一个满足要求的有利条件不久又发生了变化。

现在让我们稍微乐观一些，想象所有对象都已处于发出信号状态。这时会发生什么情况呢？首先，正在等待的线程会恢复执行，然后等待函数会改变包含于列表内的信号机以及 MUTEX 的状态。相反，把 fWaitAll 设置成 FALSE，就简单地意味着只需要任何一个列出的对象变成发出信号状态，线程的等待状态就到期了。在这种情况下，一种更为常见的现状是：线程正在等待的对象都属于相同的类型。因此，WaitForMultipleObjects（）的返回值将受到等待类型的影响。表 14-11 为我们总结了根据 fWaitAll 参数的不同值产生的不同情况。

表 14-11　WaitForMultipleObjects（）的返回值

返回值	fWaitAll	说明
WAIT_OBJECT_0 到（WAIT_OBJECT_0+cObjects-1）	TRUE	所有对象都处于发出信号状态
WAIT_OBJECT_0 到（WAIT_OBJECT_0+cObjects-1）	FLASE	返回值减去与发出信号的对象的索引对应的 WAIT_OBJECT_0
WAIT_ ABANDONED_ 0 到（WAIT_ABANDONED_0+cObjects-1）	TRUE	所有对象都处于发出信号状态，但是一个 MUTEX 被放弃了
WAIT_ ABANDONED_ 0 到（WAIT_ABANDONED_0+cObjects-1）	FLASE	返回值减去与放弃的 MUTEX 对应的 WAIT_ABANDONED_0
WAIT_TIMEOUT	TRUE 或者 FALSE	溢出时间已经超过

dwTimeout 参数既可以是用毫秒数表示的一个数值，也可以设置成 INFINITE 定义或者零值。第一种情况（数值）没有什么值得解释的。实际上，设置成 INFINITE（无限）才是最常见的一种情况，因为它提供了机会让我们挂起一个线程，直到系统内发生了某种情况为止。假如为零值，就表明需要立即对几个对象进行测试，检查它们的状态。

现在让我们把学到的所有知识都运用起来，建立一个真正的多线程应用程序，在源代码内利用几个事件对程序的内部行动进行同步处理。

14.10　线程的同步

如图 14-26 所示，SYNCHRO 应用程序是由两个独立的窗口组成的。主窗口内包含了 10 个按钮，它们代表了一个红色信号机的样子。次级窗口是一个弹出式列表视窗。这个列表视窗比较特殊，它是我们把通用控件当作一个独立窗口建立的第一个例子，不再像以前那样把它当作一个子窗口来创建。

应用程序启动以后，它会建立 10 个线程，每个按钮一个。另外还有一个附加的线程，它用于对列表视窗进行处理。每个线程的行动都要以不同事件为基础，这些事件具体则反映在对应的按钮上面：红色表示未发出信号状态，绿色则表示发出信号状态。我希望这种图形化的表达方式不会使大家在信号和事件之间造成混淆，事件在这儿扮演了一个线程的触发器的角色。

经过一些准备工作以后，每个线程都进入了一种死循环。在这以后，线程立即参加了对 WaitForSingleObject（）的一次调用，同时在其中传递了对应事件对象的句柄。

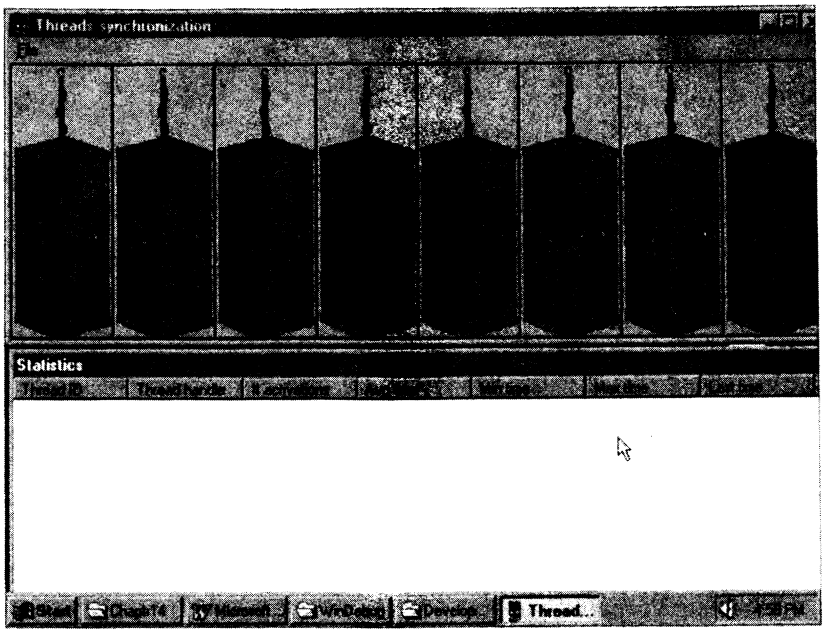


图 14-26　SYNCHRO 应用程序建立了 10 个线程，每个都由主窗口内的一个按钮控制

```
...
LRESULT WINAPI SecondThread (LONG lParam)
{
    PDATA pdata = (PDATA) lParam;
    char szText [30];
    register int i, iTot = 0, iTime;
    DWORD dwRes;
    HBITMAP hbmp;
    HINSTANCE hInstance = HINST (pdata -> hwnd);

    // semred
    hbmp = (HBITMAP) GetWindowLong (pdata -> hwnd, 0);
    // set the button image
      SendMessage ( (HWND) pdata -> hwndBtn, BM_SETIMAGE,
          (WPARAM) 0, (LPARAM) hbmp);

    // reset the semaphore
    pdata -> iState = ! pdata -> iState;
    // store current thread id
    pdata -> dwThreadID = GetCurrentThreadId ();
```

```
        // wait for the event
        while (TRUE)
        {
            // wait for the event
            dwRes = WaitForSingleObject (pdata -> hevent, INFINITE);
...
```

SecondThread () 函数唯一需要用到的参数就是指向 DATA 数据结构的一个指针，这个数据结构在源代码内已经事先定义好了。这个函数的设置项里包括主窗口句柄、主线程 ID 以及对它进行控制的事件对象的句柄。WaitForSingleObject () 等待的时间是不确定的，反正需要等到事件发出信号为止。从一名末端用户的眼光来看，这就意味着它应该在主窗口内按下对应的按钮。信号灯从红色变成绿色以后，线程就会恢复自己的执行。这种线程的的主要目的是启动一个大型作业，这种行动将耗费一定的 CPU 时间，如图 14-27 所示。

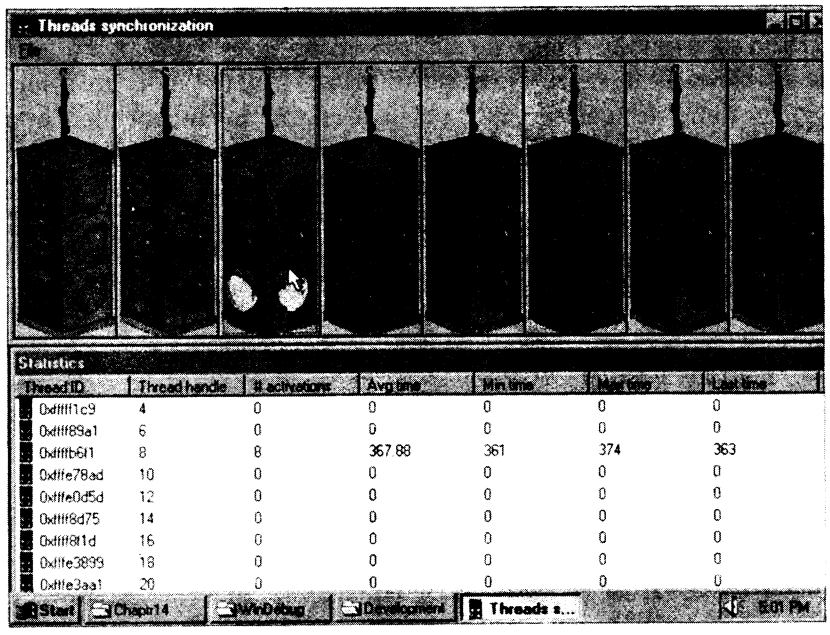


图 14-27　释放了一个单一的线程以后，SYNCHRO 程序的运行情况

一旦线程激活以后，列表视窗就会报告一些统计数据，比如用于完成任务所需的时间：平均、最短以及最长时间等。除此以外，它还会显示出线程 ID、线程句柄以及线程执行以来经历的时间。

用户按下主窗口内的一个按钮以后，从内部来说，应用程序类 SetEvent () 就会向对应的事件发出信号。作为一个人工重设事件，在 SecondThread () 内实现的对 ResetEvent () 的一次成功的调用正好在线程主体一次新的启动之前执行。

```
...
LRESULT WINAPI SecondThread (LONG lParam)
```

```
{
    ...
    // wait for the event
    while (TRUE)
    {
        // wait for the event
        dwRes = WaitForSingleObject (pdata -> hevent, INFINITE);
        ...
        // block the current thread
        ResetEvent (pdata -> hevent);
    }
    return TRUE;
}
...
```

因此，很关键的一点是要访问位于应用程序代码内一个不同位置的事件句柄，从而使线程的行动同步。

全部统计信息在一个进程里收集并且计算完毕以后，列表视窗窗口就会自动进行填写这些信息。另外一个独立的事件则控制了 SyncThread () 函数的执行。和前面一样，这个线程也需要等待一个事件发出信号才能执行。为了让这个事件发出信号，需要一个线程调用 SetEvent () 函数，同时传递恰当的事件句柄。

```
LRESULT WINAPI SyncThread (LONG lParam)
{
    PSYNC psync = (PSYNC) lParam;
    HWND hwndLV = psync -> hwndLV;

    while (1)
    {
    WaitForSingleObject (psync -> pdata -> heventSync, INFINITE);
    ListView_DeleteAllItems (hwndLV);
    FillListView (hwndLV, psync -> pdata);

    ResetEvent (psync -> pdata -> heventSync);
    }

    return TRUE;
}
```

这个线程简单地调用了 FillListView () 函数，从而把更新过的数据插入列表视窗窗口内。由于消息流穿越了一个线程的边界，所以正如我们以前就曾经指出的那样，这并不是一种优选的方案。然而，在这种情况下，准备传递的数据量限制在几百个字节以内。这种操作要不到一秒钟就能完成，所以也没有什么大的妨碍。

SYNCHRO 提供了机会让所有线程都同时启动。从外观上看，就是可以同时按下所有的信号机按钮。Start All 菜单项可以帮助我们实现这种操作。在一个非常简单和快速的循环里，全部 10 个事件都返回了发出信号的状态，图象也会相应地更新，然后启动相当耗费时间的大型作业。全部 10 个线程都共享相同的基础函数，即 SecondThread ()。因此，这些线程是完全一致的。显然，对于应用程序和系统来说，同时执行所有线程会显得相当耗时。这样造成的结果就是，用于完成每个大型作业的时间增多了，从而造成了显著的性能降低，如图 14-28 所示。

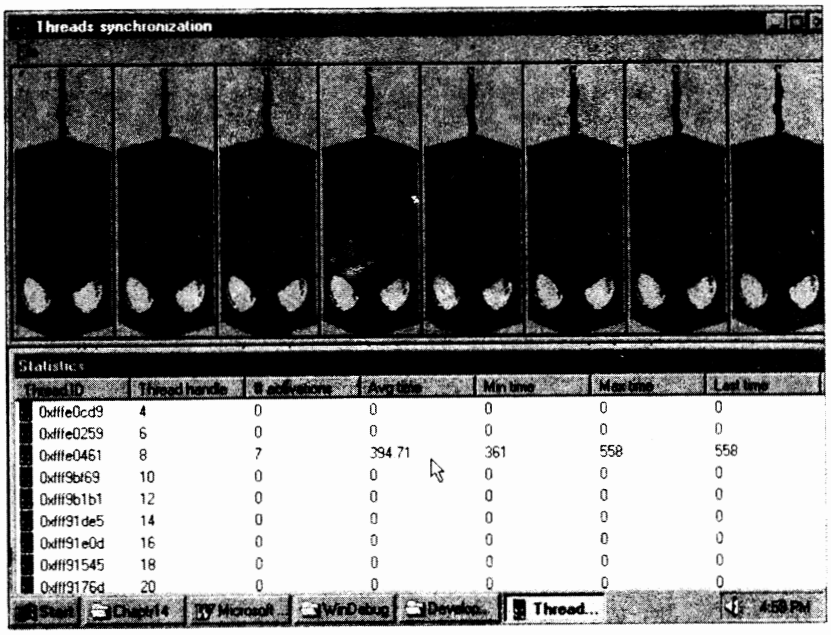


图 14-28　同时执行全部 10 个线程

14.11　总结

我在这儿希望澄清两个事实。首先，设计一个标准的 Win32 程序时，大家不能认为不从线程的角度来设计它是允许的。线程的优点在于它使整个应用程序成为模块化。显然，不使用线程将会是 Win95 程序开发者一个极大的失误。其次，随时注意 MS Windows 95 是一个“混血”操作系统，其中包含了一些 32 位代码以及大量 16 位代码。也要注意的一点是，16 位组件的存在影响了多线程的性能。对于这个问题，我们曾经用一套硬件设备完全一致的 NT 系统进行了比较。

作为我对大家最后的一点建议，不要设计过于复杂的应用程序。在其中编写几十个线程和 IPC 对象是不明智的，请限制自己程序里这两种关键成分的数量。

第 15 章　Windows 高级技术

在这一章里，大家将学习第 10 章“Windows 95 通用控件”里没有详细涉及到的几种通用控件的知识。我们同时也要讨论几种有用的 Windows 编程技巧，比如子类处理以及超类处理等等。这些知识有助于我们在应用程序里引入一些有趣的特性，这些特性往往能使人耳目一新。接下来让我们进入第一个主题：物主绘图列表视窗。

15.1　物主绘图列表视窗

在缺省情况下，列表视窗允许我们同时选择多个项目。我们只需要简单地按下 Ctrl 键不放，就可以把新项目不断增添到以前选中的列表内。图 15-1 向我们展示了在一个系统文件夹内选择多个对象的情况。

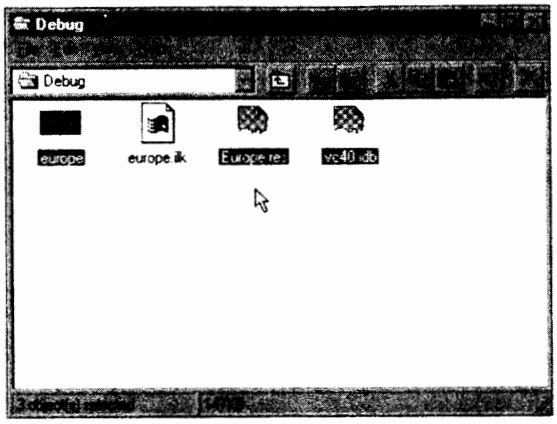


图 15-1　在 Windows 文件夹内选择多个对象

多重对象选定在每种列表视窗模式下都是支持的，这些模式包括：大图标、小图标、列表以及详细列表。在详细列表模式下，只有项目（或者称为 0 号子项目，一行信息由多个子项目构成）才能被选定。即使我们用鼠标左键在其余的子项目上面单击，也不能选中它们。事实上，在详细列表模式下面选择某个项目的唯一方式就是用鼠标左键在它上面单击，如图 15-2 所示。

根据我们在第十章“Windows 95 通用控件”里讲述的列表视窗的逻辑，这种行为完全符合要求，并且也是可以接受的。然而，某些情况下它会显得有些讨厌，特别是需要同时用到几个子项目构成的信息时。假如我们想修改这种自然的行为，把彩色的选定光条扩展到整个行（包括所有子项目），唯一的方法就是建立一个物主绘图列表视窗。在本书附带 CD 的 Listing 15.1 里，大家可以找到一个名为 EUROPE 的示范程序，它向大家演示了怎样建立一个物主绘图列表视窗。

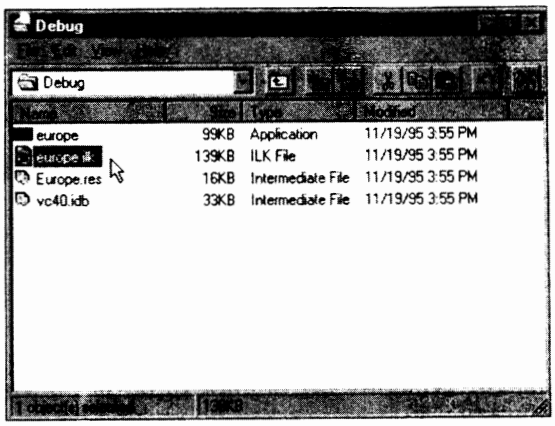


图 15-2　在详细模式下，一个项目的选定只通过单击第一个子项目才能实现

为了把这种通用控件转换成一个窗口，使它的输出由应用程序直接进行控制，我们需要采取几个步骤。首先，建立列表视窗的时候，应该预先设置好 LVS_OWNERDRAWFIXED 类风格，如下所示：

```
...
// create the list view window
hwndLV = CreateWindowEx (WS_EX_WINDOWEDGE,
                         WC_LISTVIEW, NULL,
                         WS_CHILD | WS_VISIBLE | LVS_REPORT |
                             LVS_OWNERDRAWFIXED,
                         0, 0, 0, 0,
                         hwndPapa,
                          (HMENU) CT_LISTVIEW,
                         hInstance,
                         NULL);
...
```

其中，LVS_OWNERDRAWFIXED 风格可以强迫所有输出都直接进入应用程序。控件建立好以后，需要立即发出一条 WM_MEASUREITEM 消息，紧接着再针对准备显示出来的每个项目发出一条 WM_DRAWITEM 消息。事实上，到现在为止，列表视窗的任何内容都还没有显示出来。对于物主绘图列表视窗来说，它采用的方案与列表框或者菜单相差无几。

捕获到一条 WM_MEASUREITEM 消息以后，应用程序会查询与列表视窗控件关联在一起的小图象列表，从而判断自己将包容的图象的高度。对于详细列表模式下面的每个项目来说，这种测量得到的结果就相当于项目的高度：

```
...
case WM_MEASUREITEM:
{
    MEASUREITEMSTRUCT * pmis = (MEASUREITEMSTRUCT *)lParam;
     HIMAGELIST himg;
     int cxIcon;

     // skip if it is something different from a list view
     if (pmis -> CtlType ! = ODT_LISTVIEW)
          break;

     // get the list view image list handle
     himg = ListView_GetImageList (hwndLV, LVSIL_SMALL);
     // Find out how big the image we just drew was
     ImageList_GetIconSize (himg, &cxIcon, &pmis -> itemHeight);
}
     return TRUE;
...
```

显然，整个代码最关键的部分是对 WM_DRAWITEM 消息进行处理。每次输出一个项目的时候，我们就会在列表视窗的父窗口进程里接收到这条消息。对这条消息进行处理的时候，首先应该检查它的来源。对于由消息的 lParam 指定的一个 DRAWITEMSTRUCT 数据结构来说，其中的 CtlType 项包含了表 15-1 列出的其中某个定义。假如消息的来源是一个列表视窗控件，就应该在这个项里找到 ODT_LISTVIEW 定义。

表 15-1　DRAWITEMSTRUCT 结构的 CtlType 项使用的定义

用于 CtlType 项的标志	值	说　明
ODT_HEADER	100	“标题”通用控件
ODT_TAB	101	“卡片”控件
ODT_LISTVIEW	102	“列表视窗”控件
ODT_MENU	1	菜单对象
ODT_LISTBOX	2	“列表框”控件
ODT_COMBOBOX	3	“组合框”控件
ODT_BUTTON	4	“按钮”控件
ODT_STATIC	5	“静态”控件

```
DRAWITEMSTRUCT * pdis = (DRAWITEMSTRUCT * ) lParam;

// skip if it is something different from a list view
if (pmis -> CtlType ! = ODT_LISTVIEW)
```

```
        break;
    ...
```

一旦成功地通过了测试，接下来就可以集中精力对 DRAWITEMSTRUCT 结构的 itemAction 项进行处理，这个项内包含了对当前项目行动的描述。对这个项的选择指出了针对列表视窗控件内不同项目需要采取的绘图行动。表 15-2 为我们列出了三种可能的选项。

表 15-2 DRAWITEMSTRUCT 结构的 itemAction 项使用的标志值

itemAction 项使用的标志	值	说 明
ODA_DRAWENTIRE	0x0001	整个控件都需要描绘
ODA_SECLECT	0x0002	选定状态发生了改变。需要检查 itemState 项来判断新的选定状态
ODA_FOCUS	0x0004	控件已经失去或者获得了键盘输入焦点。需要检查 itemState 项来判断控件是否获得了输入焦点

无论 itemAction 项里包含了什么东西，应用程序都必须执行一个相应的代码块，这样才能正确地显示出每个项目。这个代码块需要采取的第一步操作是测试 itemState 项的状态，从而检查其中是否存在 ODS_SELECTED 标志。假如得到了肯定的答案，前景和背景颜色就会发生改变，从而反映出新的状态。

```
...
case WM_DRAWITEM:
{
    switch (pdis -> itemAction)
    {
        case ODA_DRAWENTIRE:
        case ODA_FOCUS:
        case ODA_SELECT:
        {
            HIMAGELIST himg;
            LV_ITEM lvi;
            int cxIcon = 0, cyIcon = 0;
            int iFirstColWidth;
            RECT rcClip;
            UINT uiFlags = ILD_TRANSPARENT;

            // check to see if this item is selected
            if (pdis -> itemState & ODS_SELECTED)
            {
```

```
            // set the text background and foreground colors
            SetTextColor (pdis -> hDC, GetSysColor (COLOR_
                HIGHLIGHTTEXT));
            SetBkColor (pdis -> hDC, GetSysColor (COLOR_
                HIGHLIGHT));

            // also add the ILD_BLEND50 so the images come out selected
            uiFlags |= ILD_BLEND50;
        }
        else
        {
            // set the text background and foreground colors to the standard
                window colors
            SetTextColor (pdis -> hDC, GetSysColor (COLOR_
                WINDOWTEXT));
            SetBkColor(pdis -> hDC, GetSysColor(COLOR_WINDOW));
        }
    ...
```

在这以后，就可以调用 ImageList_Draw () 宏函数，从而沿着左窗口边框把项目图标显示出来。显示的图象是事先利用 ListView_GetItem () 宏函数从项目属性里取得的：

```
...
// get the item image to be displayed
lvi.mask = LVIF_IMAGE | LVIF_STATE;
lvi.iItem = pdis -> itemID;
lvi.iSubItem = 0;
ListView_GetItem (pdis -> hwndItem, &lvi);

// Get the image list and draw the image
himg = ListView_GetImageList (pdis -> hwndItem, LVSIL_SMALL);
if (himg)
{
    ImageList_Draw (himg,
                    lvi.iImage,
                    pdis -> hDC,
                    pdis -> rcItem.left,
                    pdis -> rcItem.top,
                    uiFlags);
```

```
}
...
```

从现在开始，我们必须着手准备将不同子项目里的正文信息显示出来，首先从与对象图标关联在一起的那个标签开始。rcItem 项内包含了与必须重画的那个项目有关的矩形区域。正如我们期望的那样，右边界指定的是整个项目最右侧的位置，并不是指每个子项目的位置。因此，应用程序必须计算出每个子矩形的位置，它的尺寸必须根据当前的栏宽进行测量。ListView_GetColumnWidth () 函数可以返回第一栏（栏目 ID 为 0）的宽度，而 ImageList_GetIconSize () 则可以返回图标在 X 和 Y 轴上的坐标距离。据此就可以计算出用于容纳子项目 0 的正文串的矩形区域。接下来，我们准备把这个子项目正文串显示出来，方法是调用 DisplayItem () 例程，如下所示：

```
...
// Find out how big the image we just drew was
ImageList_GetIconSize (himg, &cxIcon, &cyIcon);
// Calculate the width of the first column after the image width. If
// there was no image, then cxIcon will be zero.
iFirstColWidth = ListView_GetColumnWidth (hwndLV, 0);

// set up the new clipping rect for the first column text and draw it
rcClip.left = pdis -> rcItem.left + cxIcon;
rcClip.right = rcClip.left + iFirstColWidth - cxIcon;
rcClip.top = pdis -> rcItem.top;
rcClip.bottom = pdis -> rcItem.bottom;

// draw the first column
DisplayItem (pdis -> hDC, eu [pdis -> itemID] .szCountry, &rcClip);
...
```

下面三种信息将传递给 DisplayItem () 例程：

▶用于设备现场的相应句柄

▶准备显示的正文串

▶矩形区域

显示正文串并不是一种很麻烦的工作，TextOut () 函数就足以胜任，不需要对此加以特殊的考虑。然而，DisplayItem () 却显得相当复杂，因为针对它必须考虑到另外一个问题。假如用户对栏目进行拖动，从而不断缩小它的栏宽，正文标签最后就根本放不下了。这种情况下，系统外壳将对正文串进行剪切，并在末尾加上一个省略号，指出对正文串的这种处理，如图 15-3 所示。

为了实现这种特性，我们应该以象素为单位来衡量一个正文串的宽度，然后把它与栏宽

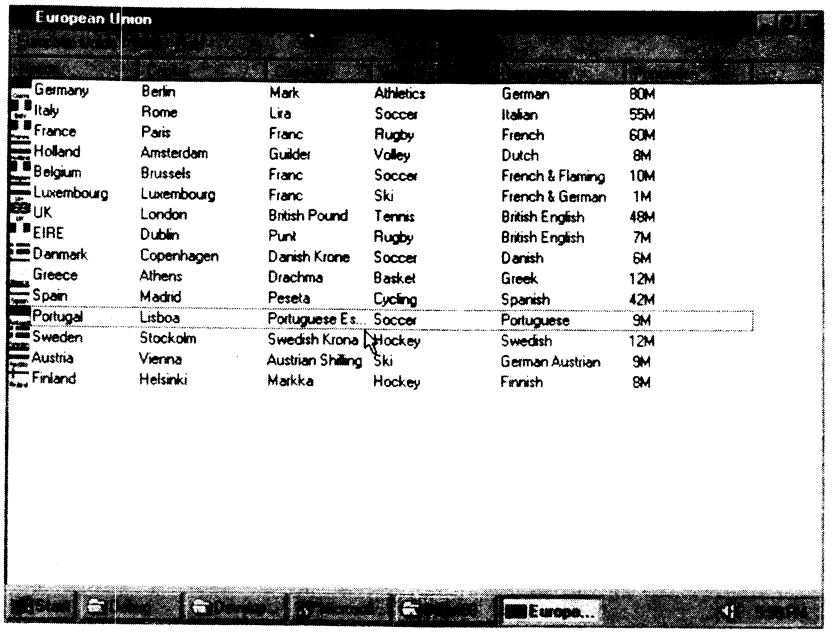


图 15-3　假如一栏的宽度不足以容下整个正文串，它就会进行剪切，并用一个省略号取代切下的部分

进行比较。假如正文串的宽度超出了栏宽，那么就放弃末尾的一个或者多个字母，然后再进行相同的比较，直到最后满足要求为止。我们对现实设备现场内对一个串进行的测量不应该感到陌生，因为这种情况在第 4 章“消息和重画模式”的 Milan 示范程序里就曾经碰到过。GetTextExtentPoint32 () 将用一个正文串宽度以及高度值来填写某个 SIZE 数据结构。假如正文串的宽度超出了栏宽，应用程序就会进入相应的代码块，从而连续性地减小它的宽度，直到最后能与栏宽正确地匹配为止。

```
...
// if the width of the string is greater than the column width shave the string
and add the ellipsis
if (sizItem.cx > iColWidth)
{
    if (! GetTextExtentPoint32 (hdc, szEllipsis, lstrlen (szEllipsis), &sizDots))
        return FALSE;

    while (cbString > 0)
    {
        szTemp [--cbString] = 0;
        if (! GetTextExtentPoint32 (hdc, szTemp, cbString, &sizItem))
            return FALSE;
```

```
            if ( (sizItem.cx + sizDots.cx) <= iColWidth)
            {
                // the string with the ellipsis finally fits
                if (MAX_PATH > = (cbString + lstrlen (szEllipsis)))
                {
                    // merge the two strings and exit
                    lstrcat (szTemp, szEllipsis);
                    lstrcpy (szString, szTemp);
                    break;
                }
            }
        }
    }
    ...
    // A call to ExtTextOut () completes the output process:
    ...
    // print the text
    ExtTextOut (hdc, prcClip -> left + 2, prcClip -> top + 1,
                ETO_CLIPPED | ETO_OPAQUE,
                prcClip, szString, lstrlen (szString), NULL);
    }
    ...
```

相同的逻辑亦可应用于其他所有子项目，这些子项目同时还省去了显示的图标的麻烦。下面这个代码段阐述了怎样根据子项目的栏宽计算出输出矩形：

```
    ...
    // CAPITAL column
    rcClip.left = rcClip.right;
    rcClip.right = rcClip.left + ListView_GetColumnWidth (hwndLV, 1);
    DrawItem (pdis -> hDC, eu [pdis -> itemID] .szCapital, &rcClip);
    ...
```

对 WM_DRAWITEM 消息的处理几乎贯穿了整个进程。在中断之前，假如设置了 ODS_SELECTED，我们就必须恢复使用标准的颜色，并且画出选定矩形，从而在包括子项目在内的整个项目周围描绘一个点状的方框。

```
    ...
    // If we changed the colors for the selected item, undo it
```

```
if (pdis -> itemState & ODS_SELECTED)
{
    // Set the text background and foreground colors
    SetTextColor (pdis -> hDC, GetSysColor (COLOR_WINDOWTEXT));
    SetBkColor (pdis -> hDC, GetSysColor (COLOR_WINDOW));
}

// If the item is focused, now draw a focus rect around the entire row
if (pdis -> itemState & ODS_FOCUS)
{
    // Adjust the left edge to exclude the image
    rcClip = pdis -> rcItem;
    rcClip.left += cxIcon;

    // Draw the focus rect
    DrawFocusRect (pdis -> hDC, &rcClip);
}
...
```

如图 15-4 所示，最后得到的结果是非常不错的。整行现在都用选定颜色显示出来，只需要鼠标单击，便可选中任何一个项目，并不像以前那样只能选择 0 号子项目。这个程序还有另外一个特色，那就是即使切换到了一个新的视窗模式，选中项目的属性也将保持不变，如图 15-5 所示。

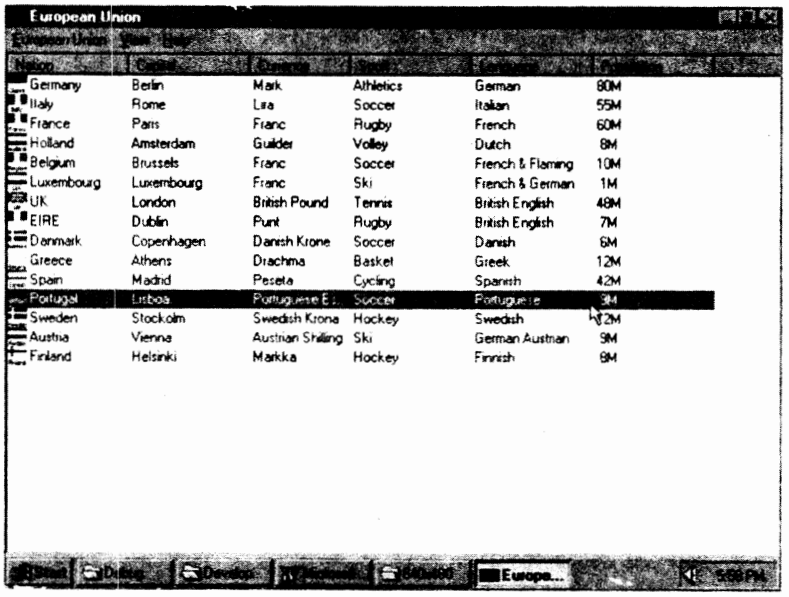


图 15-4　EUROPE 示范程序的物主绘图版本可以同时选中多个子项目

最后，假如收缩了一个栏目的宽度，以至于正文的宽度超出了栏宽，那么多余的正文就会缩短，并在末尾显示出一个省略号，如图 15-6 所示。

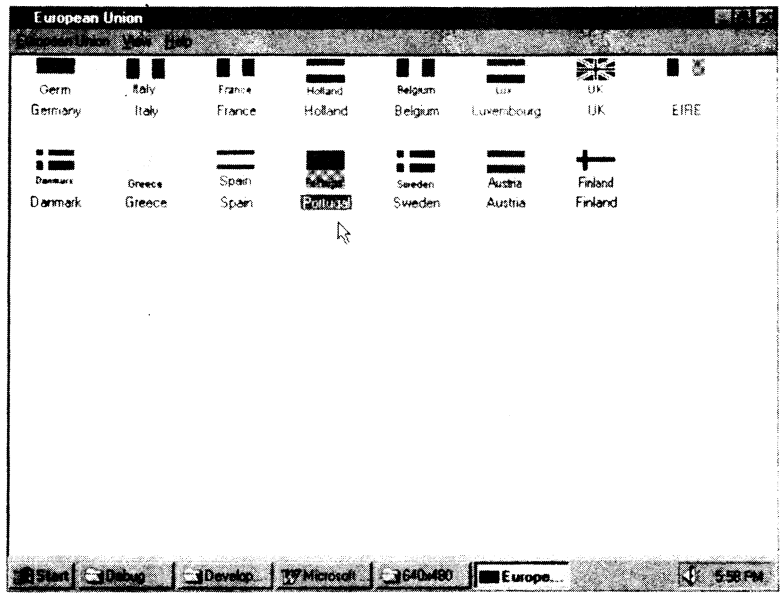


图 15-5　假如切换到与详细列表不同的一个视窗模式，列表视窗将显示出自己的标准外观，同时保持选中项目以前的状态

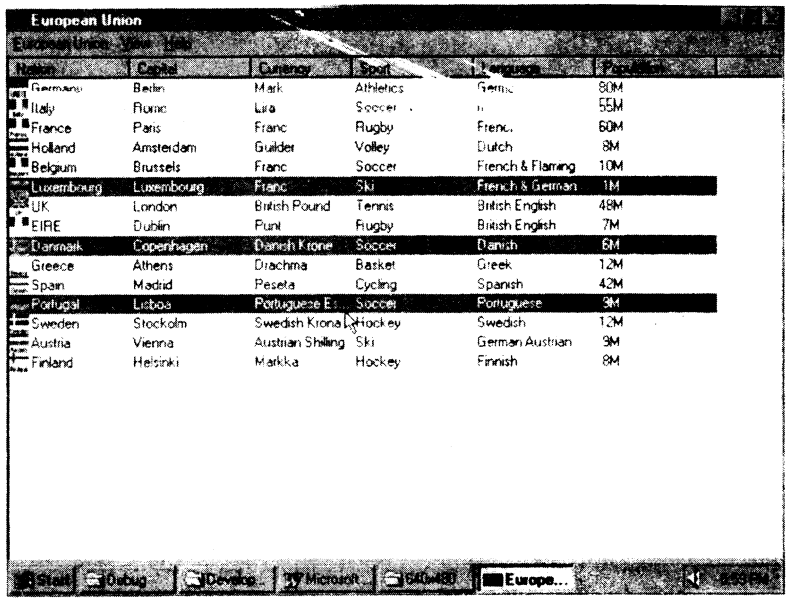


图 15-6　假如一个项目的正文超出了栏宽，多余的字符就会截去，并用一个省略号取代截去的字符

15.2 属性表

作为 Windows 95 的一名用户，我们应该很熟悉属性表的用法了，因为它们存在于我们常用的外壳界面内。通过属性表，我们可以很容易地改变屏幕颜色以及分辨率设置，并且可以设置一个对象的属性，以及其他设置工作。如图 15-7 所示。

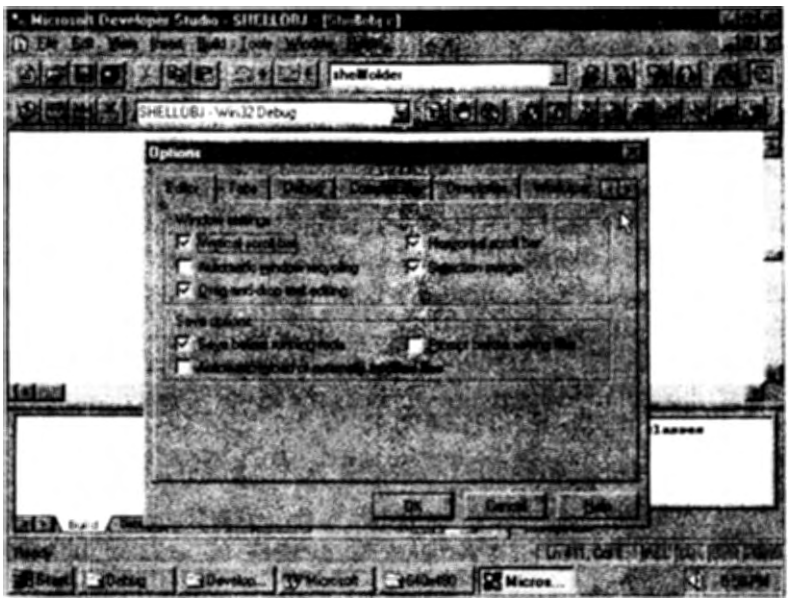


图 15-7 可以通过属性表来设置一个 Microsoft Developer Studio 项目文件的属性

为什么属性表现在变得如此流行呢？为了回答这个问题，我们首先应该知道这种对象的本质。属性表其实是一个对话框，其中包括了某些更小的对话框。属性表在每个 Windows 95 应用程序里都得到了实施，因为通过它们可以很方便地把大量信息提供给用户，从而有效地节省了屏幕空间。除此以外，对它们的构建也是很方便的，Win32 API 针对它提供了多种工具。观察某个属性表，我们能发现其中四种基本组件：

▶用户理解为属性表的总体窗口

▶窗口下端的 OK，Cancel 以及 Apply 按钮，或者加上 Help 按钮

▶几个索引标签，每个标签都对应一张不同的卡片

▶卡片

正如我们前面提到的那样，属性表是一种特殊的对话框，它属于未文档化的 #32770 类窗口。这种对话框是用一系列风格以及扩展风格设计出来的。实际上，我们并不需要建立这个对话框——API 集没有提供相应的函数。作为一个对话框，属性表意味着是一个弹出式窗口，其中带有几个子窗口。下端部分的按钮明显是属于 BUTTON 类的子窗口。

在属性表和卡片之间还存在着一种中间层，这是属于 SysTabControl32 类的一种窗口，它的作用是对涉及卡片选择的所有行动进行控制。这个窗口也是主属性表的一种子窗口。卡片本身是对话框，然而它们使用的却是 WS_CHILD 风格，这是由于属性表是它们的父窗口。因此，从窗口间的关系来看，属性表就是包含了几个子控件的一个对话框。

对于属性表来说，主要的设计工作应该集中在卡片的设计上面。这些卡片是一系列对话框，用户可在其中选择项目、下推按钮以及输入正文等等。为了建立一个属性表，我们需要用到两个数据结构：PROPSHEETPAGE 以及 PROPSHEETHEADER。通常，尽管某些外壳组件只提供了一张卡片，但是平常用到的属性表都是一个多卡片对象。针对每张卡片，我们都需要声明一个 PROPSHEETPAGE 数据结构，而 PROPSHEETHEADER 则需要一个就可以了。所有信息都准备好以后，对 PropertySheet（）函数的一次调用就足以生成一个完整的属性表，它能把所有信息都转换成一个单独的对象。

#include〈winbase.h〉

WINCOMMCTRLAPI int WINAPI PropertySheet（LPCPROPSHEETHEADER）；

其中，PROPSHEETHEADER 数据结构是我们目前为止碰到的第一种用 union 项设计的 Windows 结构。第一眼看上去，这个结构显得相当复杂，然而它的基础逻辑与列表视窗以及树形视窗控件是相差无几的。dwSize 项内包含了整体的数据结构长度，这在许多新的 Win32 结构内都是一致的。dwFlags 项（在其中情况下叫作 mask）里则使用了表 15-3 内列出的一种或者多种标志。

```
typedef struct _PROPSHEETHEADER
{
    DWORD dwSize;
    DWORD dwFlags;
    HWND hwndParent;
    HINSTANCE hInstance;
    union {
            HICON hIcon;
            LPCWSTR pszIcon;
        } DUMMYUNIONNAME;
    LPCWSTR pszCaption;
    UINT nPages;
    union {
            UINT nStartPage;
            LPCWSTR pStartPage;
        } DUMMYUNIONNAME2;
    union {
            LPCPROPSHEETPAGEW ppsp;
            HPPOPSHEETPAGE * phpage;
        } DUMMYUNIONNAME3;
    PFNPROPSHEETCALLBACK pfnCallback;
} PROPSHEETHEADER, * LPPROPSHEETHEADER;
```

其中，hwndParent 是指属性表物主，而不是实际的父窗口，因为它肯定是从桌面取得显示象

素的。对于属性表来说，我们很少看到它的标题栏内带有一个图标。尽管如此，我们仍然可以通过设置 PSH_USEICON 或者 PSH_USEICONID 标志来达到这种目的。属性表的标题是通过 pszCaption 项设置的，而构成整个对象的所有卡片的数目则是由 nPages 设置的。激活属性表以后，它甚至能够决定最初显示出哪张卡片，这是通过指定它的 ID（从 0 开始自动分配）或者指定卡片的标题来实现的。

PROPSHEETHEADER 数据结构还需要用到另外两种信息。显然，我们需要一种方式来指定构成属性表的所有卡片。这既可以通过提供一个 PROPSHEETPAGE 数组的名字来实现，也可以指定指向某个卡片句柄数组的指针来实现。最后，pfnCallback 内包含了一个回调函数的名字。属性表的建立过程中需要调用这种函数。拦截了 PSCB_INITIALIZED 消息以后，我们就有机会在属性表真正显示出来之前，对它进行定制以及部分修改它的行为，请大家参考表 15-4。

表 15-3　PROPSHEETHEADER 数据结构内 dwFlags 项使用的各种标志

标　志	值	说　明
PSH_DEFAULT	0x0000	所有结构项都用缺省值填写
PSH_USEHICON	0x0002	hIcon 项指定了属性表标题栏内使用的小图标
PSH_USEICONID	0x0004	pszIcon 项代表图标资源标签，它将在属性表的标题栏显示出来
PSH_PROPSHEETPAGE	0x0008	激活 ppsp 项，其中包含了一个 PROPSHEETPAGE 结构的地址
PSH_WIZARD	0x0020	定义一个向导属性表
PSH_USEPSTARTPAGE	0x0040	显示属性表的初始化卡片时，激活 pStartPage 项，同时忽略 nStartPage 项
PSH_NOAPPLYNOW	0x0080	把 Apply 按钮从属性表的下端部分删去
PSH_USECALLBACK	0x0100	属性表初始化的时候，强制性调用用 pfnCallback 项指定的一个函数
PSH_HASHELP	0x0200	在属性表内增加 Help 按钮
PSH_MODELESS	0x0400	建立一个非模态属性表
PSH_RTLREADING	0x0800	强迫使用从右到左的阅读顺序

表 15-4　属性表回调函数使用的消息

标　志	值	说　明
PSCB_INITIALIZED	1	属性表已经建立好以后发出
PSCB_PRECREATE	2	在建立属性表以前发出

下面这个代码段是从 Registry 示范程序里提取出来的，它的显示结果如图 15-2 所示。该程序可以在本书附带 CD 的 Listing 15.2 里找到。整个应用程序都是一个属性表，其中总共包含了七张卡片，根据 Win32 SDK 提供的信息，每张卡片都对应于注册表数据库里的一个键。物主窗口的句柄将分配给 PROPSHEETHEADER 数据结构的 hwndParent 项，从而把系统桌面指定为自己的物主窗口（HWND_DESKTOP），而不是其他某个窗口。

```
...
psh.dwSize = sizeof (PROPSHEETHEADER);
psh.dwFlags = PSH_PROPSHEETPAGE | PSH_USECALLBACK | PSH_
    NOAPPLYNOW;
```

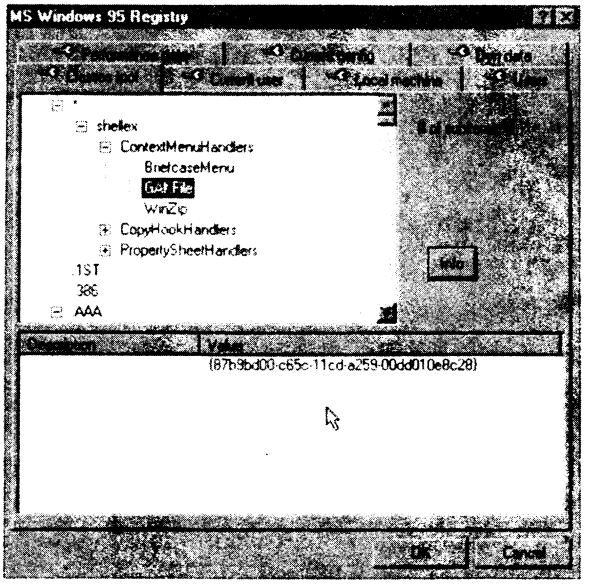


图 15-8　Registry 程序用一个属性表内的几张卡片显示了相应的注册表数据库键值

```
psh.hwndParent = hwnd;
psh.hInstance = hInstance;
psh.nStartPage = 0;
psh.pfnCallback = PropSheetDlgProc;
psh.pszCaption = (LPSTR)" MS Windows 95 Registry";
psh.nPages = sizeof (psp) /sizeof (PROPSHEETPAGE);
psh.ppsp = (LPCPROPSHEETPAGE) &psp;
psh.hIcon = LoadIcon (hInstance," registry");
...
```

PROPSHEETHEADER 数据结构里的关键元素是某个 PROPSHEETPAGE 结构数组的地址，其中包含了每张卡片的实际显示信息。

```
typedef struct _PROPSHEETPAGE
{
    DWORD dwSize;
    DWORD dwFlags;
    HINSTANCE hInstance;
    union {
            LPCWSTR pszTemplage;
            LPCDLGTEMPLAGE pResource;
        } DUMMYUNIONNAME;
```

```
    union {
            HICON hIcon;
            LPCWSTR pszIcon;
        } DUMMYUNIONNAME2;
    LPCWSTR pszTitle;
    DLGPROC pfnDlgProc;
    LPARAM lParam;
    LPFNPSPCALLBACKW pfnCallback;
    UINT * pcRefParent;
} PROPSHEETPAGEW, *LPPROPSHEETPAGE;
```

和以前一样，PROPSHEETPAGE 结构也提供了类型为 union 的两个项，这反映了数据结构设计中的最新潮流。dwFlags 项内包含了表 15-5 内列出的一个或者多个定义，它们可用于判断需要考虑进去的项。

表 15-5 PROPSHEETPAGE 数据结构使用的标志

标 志	值	说 明
PSP_DEFAULT	0x0000	为所有结构项分配缺省值
PSP_DLGINIRECT	0x0001	卡片是通过一个 RAM 对话框模板建立的
PSP_USEHICON	0x0002	每张卡片的标签都提供了通过它的句柄取得的一个图标
PSP_USEICONID	0x0004	每张卡片的的标签都提供了由它的 ID 取得的一个图标
PSP_USETITLE	0x0008	每张卡片都有自己的正文标题
PSP_RTLREADING	0x0010	正文按照从右到左的顺序显示，在希伯来语和阿拉伯语系统中适用
PSP_HASHELP	0x0020	显示这张卡片的时候，激活属性表的 Help 按钮
PSP_USEREFPARENT	0x0040	针对通过这个结构建立的属性表卡片，对 pcRefParent 指定的引用计数值进行跟踪
PSP_USECALLBACK	0x0080	一个回调函数，只要建立和破坏了一张卡片，就调用这个函数

假如组合使用 PSP_USEHICON 和 PSP_USETITLE，我们就可以定义一个卡片标签，其中提供了一个图标，后面再接上一个正文串。尽管系统外壳本身并没有采用这种风格，但是它确实能产生相当美观的视觉效果。到目前为止，PROPSHEETPAGE 数据结构里最重要的项是 pfnDlgProc，它设置了与一张卡片关联在一起的某个对话框进程的名称。通常情况下，应用程序都配备了许多对话框进程，它们的数量等于或者大于属性表内的卡片张数。如果每张卡片都提供了一些唯一的特性和信息，那么这种情况就是很常见的。然而，这种情况并不适用于 Registry 示范程序，其中全部七张卡片都共享同一个对话框进程，即 GenericDlgProc() 例程。除此以外，这个进程还是一种标准的对话框进程，它的宗旨是对所有的用户选择进行控制。这种控制是通过对 WM_COMMAND 和 WM_NOTIFY 消息发送的输入通知代码进行响应而实现的。

下面这段代码是从 Registry 源程序里提取出来的，它向我们展示了如何填写一个 PROPSHEETPAGE 结构：

```
...
for (i = 0; i < PAGENUM; i++)
{
    // loading the title
    iLen = LoadString (hInstance, ST_TITLES + i, p, sizeof (szTitle) /
        PAGENUM);

    psp [i].dwSize = sizeof (PROPSHEETPAGE);
    psp [i].dwFlags = PSP_USETITLE | PSP_USEHICON | PSP_
        USEREFPARENT;
    psp [i].hInstance = hInstance;
    psp [i].pszTemplate = szTemplate;
    psp [i].pfnDlgProc = pDlgProc;
    psp [i].pszTitle = p;
    psp [i].hIcon = LoadIcon (hInstance, " key");
    // page ID
    psp [i].pcRefParent = &i;
    psp [i].lParam = i;
    p += iLen + 1;
}
...
```

为了把恰当的标题传递给每张卡片，应用程序会从相应的资源里读取一个串，然后把它存储到一个字符数组里，并用一个 NULL 值对其进行分隔。随后，PROPSHEETPAGE 结构数组的地址将存储到 PROPSHEETHEADER 对象的 ppsp 项内。而整个数据块以后将通过 PropertySheet () 函数进行处理。

属性表卡片

尽管属性表是对话框的一种类型，我们却不需在应用程序源代码内编写相应的对话框进程。然而，这并不意味着将造成我们的不便，因为属性表的主要用途就是对卡片进行包容。作为标准的组件，OK，Cancel，Help 以及 Apply 按钮会在卡片的底部显示出来，并且它们应该把自己的通知代码传递给父窗口——即属性表。在这种情况下，消息流将经过不用的途径，从而把通知代码发送给与当前活动卡片相关的对话框进程。因此，大家需要注意这样一个问题，卡片对话框进程内接收到的某些通知代码引用的将是属性表本身。

对通知代码进行负载的消息是 WM_NOTIFY，这是一种标准的通用控件消息载体。假如 WM_NOTIFY 引用的是属性表，那么在这种条件下，wParam 里缺少窗口 ID 将显得极为反常。然而这并不是一种系统错误。更准确地说，这是属性表为我们带来的一种自然的结果，因为它自己就是一个弹出式窗口。请记住，弹出式窗口并不支持窗口 ID，因为它有能力显示一个菜单。从本质上说，当我们在一个卡片对话框进程里捕获了一条 WM_NOTIFY 消息以后，必须注意某些通知代码指定的也许是属性表本身。下面这个代码段向我们阐述了怎样对属性

表发出的 PSN_通知代码进行跟踪和捕获，wParam 里的一个零表示引用的是整个对象：

```
...
case WM_NOTIFY:
{
    LPNMHDR pnmhdr = (LPNMHDR) lParam;

    switch (wParam)
    {
        case 0:                    // property sheet
        {
            switch (pnmhdr -> code)
            {
                case PSN_KILLACTIVE:
                {
                    ShowWindow (hwndPopup, SW_HIDE);
                }
                    break;

                case PSN_SETACTIVE:
                {
                    // MessageBeep (0);
                }
                    break;
            }
        }
            break;
...
```

属性表能够使用的所有通知代码都列于表 15-6 里。正如我们从各自的说明里能推断出来的那样，通知代码指定了在属性表底部的那些按钮上采取的各种行动。

表 15-6　属性表通知代码

通知代码	值	说　明
PSN_SETACTIVE	(PSN_FIRST-0)	发送给一张准备变成活动状态的卡片
PSN_KILLACTIVE	(PSN_FIRST-1)	一张卡片准备失去活动状态的时候发出
PSN_VALIDATE	(PSN_FIRST-1)	一张卡片准备失去活动状态的时候发出
PSN_APPLY	(PSN_FIRST-2)	用户按下了 OK 或者 Apply 按钮以后发出
PSN_RESET	(PSN_FIRST-3)	用户按下 Cancel 按钮后发出

（续）

通知代码	值	说　明
PSN_CANCEL	(PSN_FIRST-3)	用户按下 Cancel 按钮后发出
PSN_HELP	(PSN_FIRST-5)	用户按下 Help 按钮后发出
PSN_WIZBACK	(PSN_FIRST-6)	用户按下一个向导属性表内的 Back 按钮后发出
PSN_WIZNEXT	(PSN_FIRST-7)	用户按下一个向导属性表内的 Next 按钮后发出
PSN_WIZFINISH	(PSN_FIRST-8)	用户按下一个向导属性表内的 Finish 按钮后发出
PSN_QUERYCANCEL	(PSN_FIRST-9)	用户按下 Cancel 按钮后发出

和以前一样，通过属性表接收到的信息通常需要应用程序发送一条或者多条消息，同时令这些消息定址于属性表。对于表 15-7 列出的所有消息来说，利用其中一些可以对用户在应用程序代码内模拟用户进行的操作。其他消息则可用于执行一些基本的任务，比如选择一张卡片、增加一张新卡片或者删除现成卡片等等。

表 15-7　属性表使用的消息

属性表消息	值	说　明
PSM_SETCURSEL	(WM_USER+101)	通过提供相应的句柄，从而激活一张卡片
PSM_REMOVEPAGE	(WM_USER+102)	把一张卡片从属性表内删去
PSM_ADDPAGE	(WM_USER+103)	在属性表内增加一张新卡片
PSM_CHANGED	(WM_USER+104)	指出卡片内的选定情况已发生了变化，强迫属性表激活 Apply 按钮
PSM_RESTARTWINDOWS	(WM_USER+105)	用户按下 OK 按钮后，强迫属性表返回 ID_PSRESTARTWINDOWS，从而指出有必要重新启动 Windows
PSM_REBOOTSYSTEM	(WM_USER+106)	用户按下 OK 按钮后，强迫属性表返回 ID_PSREBOOTSYSTEM，从而指出有必要重新启动整个系统
PSM_CNACELTOCLOSE	(WM_USER+107)	屏蔽 Cancel 按钮，并把 OK 按钮的正文变成 Close
PSM_QUERYSIBLINGS	(WM_USER+108)	把信息传递给属性表内的所有卡片
PSM_UNCHANGED	(WM_USER+109)	指出一张卡片内的信息必须转换成以前预先存储状态
PSM_APPLY	(WM_USER+110)	模拟按下 Apply 按钮
PSM_SETTITLE	(WM_USER+111)	设置属性表的标题
PSM_SETWIZBUTTONS	(WM_USER+112)	分别用下面这些定义激活 Backup，Next 以及 Finish 按钮：PSWIZB_ BACK，PSWIZB_ NEXT 和 PSWIZB_ FINISH
PSM_PRESSBUTTON	(WM_USER+113)	模拟按下一个属性表按钮（完整的清单请参考表 15-8）
PSM_SETCURSELID	(WM_USER+114)	通过资源 ID 指定一张卡片，从而激活它
PSM_SETFINISHTEXT	(WM_USER+115)	激活 Finish 按钮、设置它的正文以及屏蔽 Back 和 Next 按钮
PSM_GETTABCONTROL	(WM_USER+116)	返回属性表内的卡片标签控件句柄
PSM_ISDIALOGMESSAGE	(WM_USER+117)	判断是否为一条对话框消息
PSM_GETCURRENTPAGEHWND	(WM_USER+118)	返回当前卡片的句柄

表 15-8　与 PSM_PRESSBUTTON 组合使用的按钮定义

按钮 ID	值	说　明	按钮 ID	值	说　明
PSBTN_BACK	0	Back 按钮	PSBTN_APPLYNOW	4	Apply 按钮
PSBTN_NEXT	1	Next 按钮	PSBTN_CANCEL	5	Cancel 按钮
PSBTN_FINISH	2	Finish 按钮	PSBTN_HELP	6	Help 按钮
PSBTN_OK	3	OK 按钮			

卡片可以通过两种不同的方法进行标识，最简单的方法就是使用它的 ID。卡片 ID 是在每张卡片插入时由属性表自动分配的。一些消息则需要指定卡片的句柄，以便完成某些特殊的任务。增加一张新卡片的时候，就需要指定它的句柄。通过在 PROPSHEETPAGE 数据结构里填写某些恰当的信息，然后把这个结构的地址传递给 CreatePropertySheetPage () 函数，这样就可以实现一张新卡片的加入。该函数的返回值既可能是一张新建卡片的句柄，也有可能是 NULL（假如函数调用由于某种原因失败了），如下所示：

```
#include 〈prsht.h〉
HPROPSHEETPAGE CreatePropertySheetPage (LPCPROPSHEETPAGE lppsp);
```

消息 PSM_ADDPAGE 可以在属性表内增加一张新卡片，同时把它放置于卡片列表的最末尾。这张新卡片将自动分配一个 ID，这个 ID 等于当前卡片的总数减 1——因为卡片 ID 编号是从 0 开始的。不幸的是，PSM_消息集里没有提供任何方便可以让我们立刻得到属性表内的卡片总数，也没有办法可以对它们进行排序。

很重要的一点是要把握住在属性表内建立卡片区域的准确时机。每张卡片都有它们自己的对话框进程，这个进程建立好以后会接收到一条传统的 WM_INITDIALOG 消息（在 Registry 示范程序里，所有卡片都共享一个相同的对话框进程，但这种情况并不多见）。这种进程的建立事件只有在卡片首次进入前台显示的时候才会发生。从那个时候开始，对话框就会保留在应用程序的内存里，以便后续的使用。所以，假如我们希望一张卡片每次进入前台时都执行某段特定的代码，就必须采用一种不同的策略，因为对 WM_INITDIALOG 消息进行拦截并不是次次都管用。

为了得到一种正确的方案，我们应该拦截 PSN_SETACTIVE 通知代码。只要用户选中了一张不同的卡片，这条通知代码就会到达一个窗口进程。这儿的问题在于系统并没有提供任何方式可以把不同的卡片区分开来。从本质上说，这条通知代码没有提供卡片 ID 或者卡片句柄。我们知道的全部情况就是用户已经选择了另外一张卡片。因此，我们该怎样解决这个问题？假如卡片拥有独立的对话框进程，我们就可以采取一种自动处理方案，因为 PSN_SETACTIVE 通知代码每次都会抵达一个不同的进程。在 Registry 示范程序里，情况显得稍微复杂一些，因为所有卡片都一致共享同一个对话框进程。在一张属性表卡片里，存在着属于 #32770 类的一个子窗口，它的父窗口是另外一个对话框。作为子窗口，所有卡片都会在 GWL_ID 内存区域里保留窗口 ID。

在缺省情况下，对于作为卡片运作的一个对话框来说，属性表不会为它分配任何一个值。相反，它会留下余地，以便从应用程序源代码内部对进行控制。在每张卡片初始化的时候，

Registry 程序都为它们任意分配了一个 ID，这种事件会生成著名的 WM_INITDIALOG 消息。在那个时候，我们可以强制性地将一个 ID 引入卡片预约内存区域，如下所示：

```
...
// Set the page ID
SetWindowLong (hwnd, GWL_ID, ppsp -> lParam);
    ...
```

在这种情况下，每张卡片都分配了预先存储在 PROPSHEETPAGE 数据结构的 lParam 项内的数字，它的作用是对卡片进行说明。PSN_SETACTIVE 消息抵达一个窗口进程以后，应用程序就可以从卡片预约内存区域里取得相应的 ID，就像下面这个代码段演示的那样：

```
...
case PSN_SETACTIVE:
{
    int iID;

    // get the page ID
    iID = (int) GetWindowLong (hwnd, GWL_ID);
    break;
}
    break;
    ...
```

调用 GetWindowLong () 的时候，hwnd 参数是对话框进程里四个参数中的第一个，它指定了当前的卡片（一个对话框）。PSN_SETACTIVE 通知代码是通过 WM_NOTIFY 消息负载的，WM_NOTIFY 消息在自己的 lParam 里包含了一个 NMHDR 数据结构的地址。和标准对话框进程里发生的情况不一样，NMHDR 结构的 hwndFrom 项是指属性表对象（活动卡片的父窗口），而不是发出通知代码的那个控件。这就解释了为什么说 hwnd 是用于指定活动卡片的最恰当的句柄。

Registry 程序会把与一个注册表键关联在一起的串值显示在列表视窗控件的一张卡片的底部。假如用户展开了一个节点，然后选择一个分支，应用程序就会自动尝试把选中的项目移至树形控件的顶部，如图 15-9 所示。除此以外，选中项目使用的子键总数也会在卡片的右侧部分显示出来。假如用鼠标右键单击某个键值，鼠标指针下方的项目就会选中，并且显示出一个弹出式菜单，如图 15-10 所示。

属性表的标题栏内容纳了一个小图标，其中带有一个小问号（?），这是我们使用扩展风格标志 WS_EX_CONTEXTHELP 以后的结果。Registry 的一个特色在于这个对象是在应用程序总体方案内实现的。就系统外壳来说，它对这种对象的运用是很广泛的，几乎任何一个属性表内都可以看到它的踪影。如果选中这个图标，光标上面就会增加一个小问号，同时让

一系列帮助命令作好准备。事实上，假如通过鼠标左键单击而选中一个窗口组件，应用程序就会针对这个组件显示一条简短的帮助消息，如图 15-11 所示。这是通过提供对帮助引擎的支持来实现的。

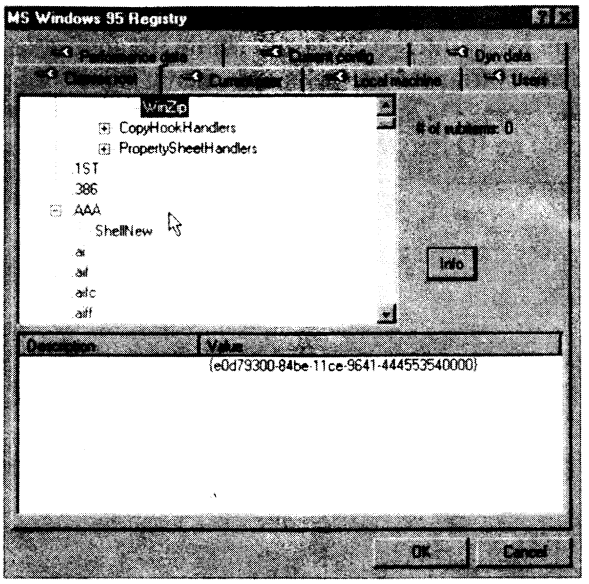


图 15-9　选中的项目肯定位于树形视窗控件的顶部

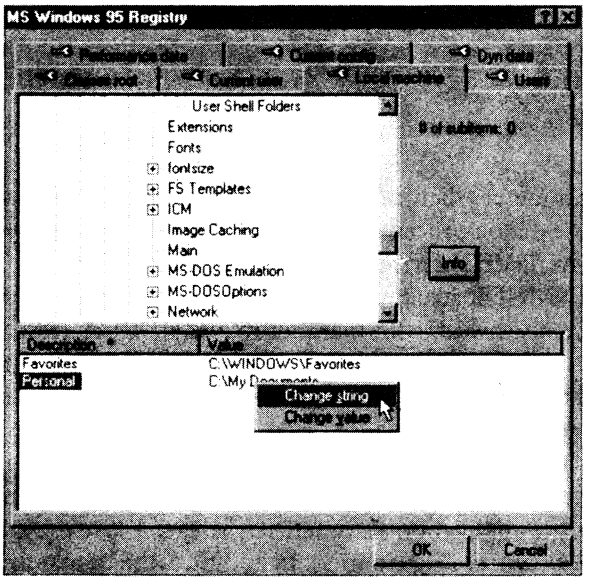


图 15-10　用户在列表视窗内某个项目上方按下鼠标右键以后，一个弹出式菜单就会显示出来

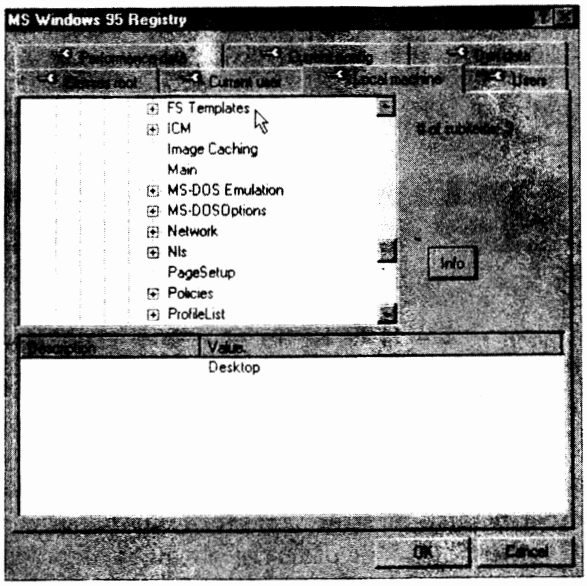


图 15-11　先选择标题内的问号图标，然后选择一个窗口组件，这样就能引入一条相应的帮助消息

用户选择了标题栏上的帮助按钮以后，Registry 程序会采取一些不同的处理措施。它仍然会提供一条帮助消息，然而却是用不同的格式提供的。我们只能用鼠标左键单击一个注册表节点——亦即树形视窗控件内的一个项目。事件已经发生以后，应用程序就会显示一个弹出式菜单，其中包含了一个树形视窗控件，它向我们展示了注册表数据库分级结构内的整个键值路径。如图 15-12 所示。

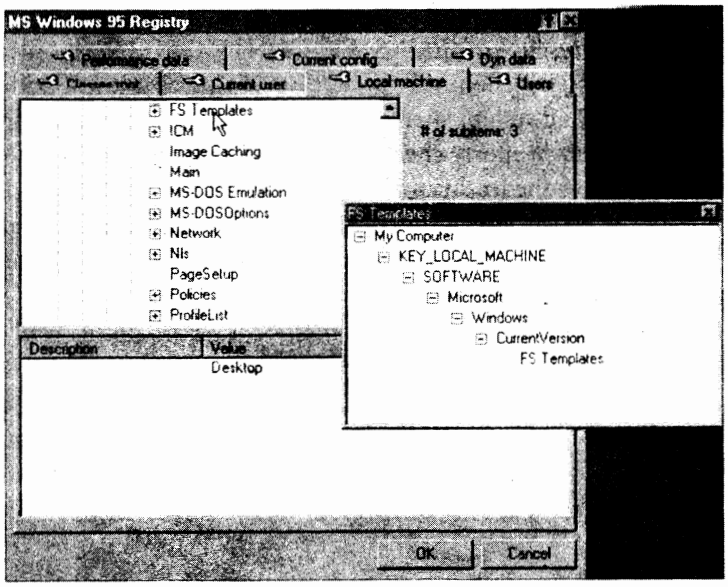


图 15-12　屏幕中的弹出式窗口立即用可视化的反馈信息表达了某个键值在注册表数据库内的位置

弹出式窗口实际是用 WS_POPUP 标志以及其他某些标志建立的一个树形视窗，其他那些标志的作用是让它看起来更像一个普通的窗口。假如用户用鼠标右键单击一个树形视窗项目，应用程序首先会判断鼠标指针下方显示的那个项目的树形视窗句柄。随后，从进程堆里将动态地分配一个内存块，以便存放它的完整的注册表路径名称。树形分级结构随后就会从选中的节点向后移动，直到最后进入整个结构的根部。在移动过程中，每次移动取得的串值都会存储到分配的内存块内。最后，取得的树形结构信息将用一个弹出式窗口的形式向用户显示出来。

```
...
case WM_HELP:
{
    LPHELPINFO phi = (LPHELPINFO) lParam;

    switch (phi -> iCtrlId)
    {
        case CT_TREEVIEW:
        {
            POINT pt;
            TV_HITTESTINFO tvht;

            TV_ITEM tvi;
            char szKeyName [40];
            HWND hwndTree = CTRL (hwnd, CT_TREEVIEW);

            // saving the mouse coodinate
            GetCursorPos (&pt);
            tvht.pt = pt;
            // converting the coordinate in the TreeView window
            ScreenToClient (hwndTree, & (tvht.pt));
            // search text only
            tvht.flags = TVHT_ONITEMLABEL;
            // did we select an item?
            TreeView_HitTest (hwndTree, &tvht);

            // yes, we selected an item
            if (tvht.hItem)
            {
                HTREEITEM hParentItem, hTreeNew;
                char *p, *p1;
```

```
int iLines = 0;
TV_INSERTSTRUCT tvis;

// allocate a chunk of memory
p = (char *) HeapAlloc (hHeap, 0, PATHSIZE);
TreeView_DeleteAllItems (hwndPopup);
// visually select that item
TreeView_Select (hwndTree, tvht.hItem, TVGN_CARET);

// retrieve information from the current item
tvi.mask = TVIF_TEXT | TVIF_PARAM | TVIF_
    CHILDREN;
tvi.hItem = tvht.hItem;
tvi.pszText = szKeyName;
tvi.lParam = 0;
tvi.cchTextMax = sizeof (szKeyName);
TreeView_GetItem (hwndTree, &tvi);

// store the node name
p1 = strcpy (p + PATHSIZE - strlen (tvi.pszText) - 2,
     tvi.pszText);

// set the popup window title
SetWindowText (hwndPopup, p1);

iLines++;

// walk the hierarchical path backwards
while (tvi.hItem ! = TreeView_GetRoot (hwndTree))
{
     hParentItem = TreeView_GetParent (hwndTree,
          tvi.hItem);
     tvi.mask = TVIF_TEXT | TVIF_PARAM | TVIF_
          CHILDREN;
     tvi.hItem = hParentItem;
     tvi.pszText = szKeyName;
     tvi.lParam = 0;
     tvi.cchTextMax = sizeof (szKeyName);
     TreeView_GetItem (hwndTree, &tvi);
```

```
                    p1 = strcpy (p1 - strlen (tvi.pszText) - 1,
                        tvi.pszText);
                    iLines ++;
                }
                // add the MY Computer heading
                p1 = strcpy (p1 - strlen (" My Computer"), " My
                  Computer");
                    .
                // display the popup window
                ShowWindow (hwndPopup, SW_SHOWNORMAL);
                hTreeNew = TVI_ROOT;
                while (*p1)
                {
                    // Inserting the root item
                    tvis.hParent = hTreeNew;
                    tvis.hInsertAfter = TVI_SORT;
                    tvis.item.mask = TVIF_TEXT | TVIF_STATE;
                    tvis.item.state = TVIS_EXPANDED;
                    tvis.item.pszText = p1;
                    tvis.item.cchTextMax = strlen (p1) + 1;
                    hTreeNew = TreeView_InsertItem (hwndPopup, &tvis);
                    p1 += strlen (p1) + 1;
                }

                // free the chunck of memory
                HeapFree (hHeap, 0, p);
            }
        }
            break;
    }
}
    break;
...
```

一旦用户在树形视窗控件内选择了一个新键，弹出式窗口就会消失。此外，假如选择HKEY_PERFORMANCE_DATA，程序还会提醒用户这个键在 Windows 95 里没有实现。事实上，这只是 Windows NT 使用的一种键值。NT 这种功能强劲的 32 位操作系统可以利用这个键对几种信息进行跟踪，比如正在运行的进程数目和种类等等。这也是为什么 PVIEWER.EXE 工具程序从一开头就与 NT 捆绑在一起的原因，那时 Windows 95 还没有出

世呢！请大家参考图 15-13。

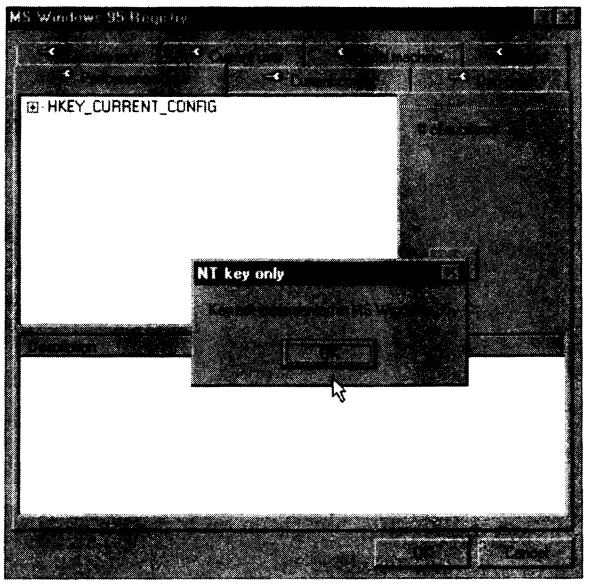


图 15-13　假如选择 HKEY_PERFORMANCE_DATA 键，REGISTRY 程序就会显示一个消息框，提醒用户这个键在 Windows 95 里还没有实现

15.3　向导的建立

在 Windows 95 系统里，当我们安装一种新硬件或者修改 Start 菜单的时候，就会出现一个向导，它会一步一步指导我们完成整个操作。在这儿，向导（Wizard）的本质就是一种属性表，其中由一系列对话框构成。向导与标准属性表之间唯一的区别在于显示这些对话框的方法不同。在向导里，用户一次只能看到并访问一张卡片。卡片的显示可以进退自如，这主要归功于向导底部显示的 Back（上一步）和 Next（下一步）按钮。除此以外，Cancel（退出）按钮也会在某些场合下出现。同时，假如用户到达了最后一张卡片，Finish（完成）按钮也会显示出来，从而指出整个操作可以结束了。

假如用户利用 Create New Shortcut（创建新快捷）在系统桌面建立一个新的快捷时，一个向导就会自动激活，然后引导用户完成整个创建过程，如图 15-14 所示。在第十六章“Windows 95 的外壳开发”里，我们将对向导活动背后的这种逻辑进行详细的解释。

为了生成一个向导，我们只需要在标准属性表的建立过程中进行少许的变动就可以了。首先，我们必须设置 PSH_WIZARD 标志，从而强制性地用更恰当的 Back 和 Next 按钮替代传递的 OK 和 Cancel 按钮。这种变动带来的另外一个影响是每张卡片的标题都会变成当前向导的标题。从一张卡片移至另一张卡片的时候，标题也会发生相应的变化。

为了与系统向导的风格吻合，我们建立的卡片应该反映出由设计准则建议的尺寸。这些尺寸是用一系列以前缀 WIZ_开头的值定义的，如表 15-9 所示。

图 15-5 向大家展示了如何把表 15-9 的定义应用于标准的向导卡片设计。

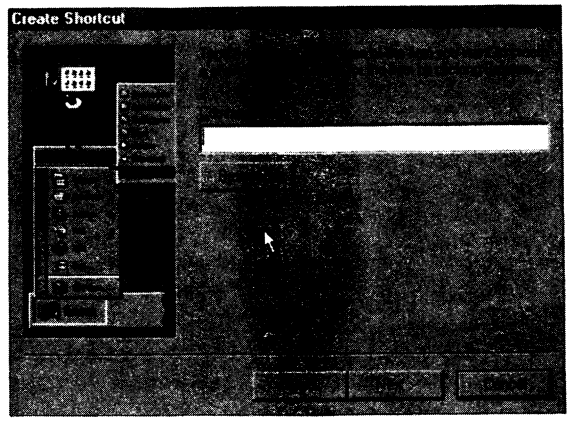


图 15-14　系统向导可以引导用户人工建立一个新快捷

表 15-9　由向导建议的尺寸

向导推荐的尺寸	值	说　明
WIZ_CXDLG	276	卡片宽度
WIZ_CYDLG	140	卡片高度
WIZ_CXBMP	80	向导卡片内位图区域的宽度
WIZ_BODYX	92	向导卡片内主体部分的水平初始位置
WIZ_BODYCX	184	与向导卡片对应的整体水平宽度

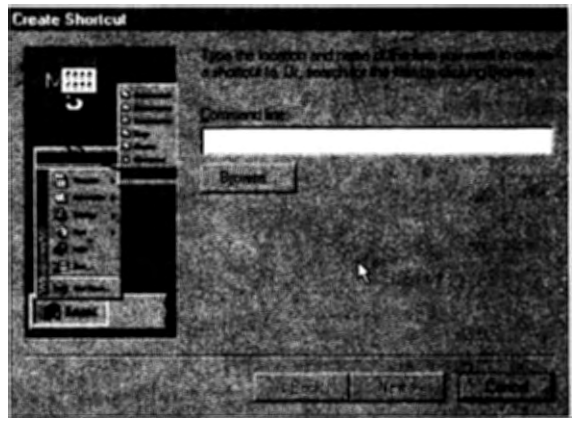


图 15-15　向导对象以及它们的建议尺寸

在表 15-6 内，我们列出了 PSN_WIZBACK，PSN_WIZNEXT 以及 PSN_WIZFINISH 这三条通知代码，它们是专门针对从向导按钮里接收反馈信息而设计的。尽管存在这些对象，并且向导卡片可以从一张自动切换到另一张，但是应用程序的开发者仍然需要仔细地引导向导控件的逻辑，使其不致于遗漏需要涉及的任何一种操作。向导卡片以及它们的内容都是由一个基础结构提供的，这个结构将专门用于向导的建立。尽管如此，向导的所有逻辑都都只能以在应用程序里编写的代码为基础。图 15-16 向大家展示了用于安装本书附带 CD-ROM 示范

程序的那个向导的第一张卡片。

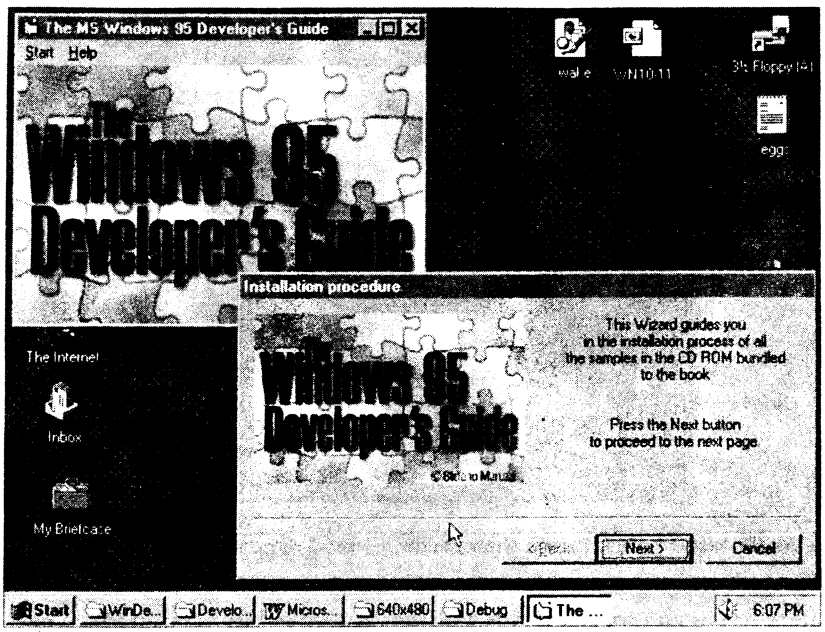


图 15-16　CD-ROM 设置模块里的安装向导

15.4　子类处理和超类处理

控件类以及新的通用控件都提供了高级的特性，但是，正如我们经常发现的那样，它们仍然没有达到能够胜任任何一种编程需求的程度。对一个预定义窗口类进行处理的时候，我们可以采取子类处理或者超类处理，从而对它的功能进行扩展。这两个术语是指两种相似的技术，它们的宗旨是在一个窗口或者整个预定义窗口类里增加一个定制的窗口进程。

我们在这儿的基本考虑是把一个类与其他某些类集成到一起，从而发掘出它的大多数特性。例如，假设某个应用程序需要用到一个窗口，这个窗口只能用于接收数字。在美国，社会保险号码是由九位数字组成的，其中不能使用字母。为了接收用户输入，一个“编辑”(Edit）窗口看起来是最佳途径。然而，如果只是单纯地使用这样一个窗口，我们就没有办法在需要输入数字的地方防止这些用户输入字母。因此，输入焦点从一个对象移至下一个对象的时候，或者用户确认整个对话框的操作以后，我们的应用程序就应该对输入字符串进行检查。假如侦测到了一个错误，通常就会显示一个消息框，提醒用户重新输入合法字符。但是，假如应用程序本身就能防止这种潜在的错误，那岂不是更好？比如，它可以禁止用户键入除数字以外的其他字符。为了让这种想法在自己的程序代码得以实施，就需要用到子类处理。

建立了一个窗口以后，无论它属于哪个类，Windows 内部都会为它分配一个内存块，在其中存放与那个新系统对象有关的所有信息。对于这种由窗口预约的内存区域来说，我们在其中包含的信息里应该能找到类窗口进程的地址。实际上，只要有消息发送给某个窗口，对应的窗口进程就会被调用。到目前为止，那个地址还肯定是与类窗口进程地址对应的，然而我们现在为它分配一个不同的值。假如使用 SetWindowLong（），并在其中设置了 GWL_

WNDPROC 标志，我们就可以把一个新窗口进程的地址分配给系统内的任何一个窗口，从而使那个窗口具有不同的行为。对实现这种效果所需的所有步骤进行描述之前，让我们首先掌握与子类处理技术有关的一些要点。

首先，我们只能对属于预定义窗口类的某个窗口进行子类处理。我们应该很清楚，对应用程序代码内注册的某个类属下的一个窗口进行子类处理是没有什么真正价值的。实际上，这种做法可以说是很愚蠢的。假如我们建立的一个窗口从属于应用程序内某个已经注册的类，后来又觉得它应该从属于一个不同的窗口进程，那么此时对它进行子类处理的后果将是非常严重的。正确的方案应该是把窗口移至另外一个类，如果有必要，甚至可以再注册一个新类。

除此以外，子类处理并不意味着我们要用一个新的类窗口进程取代原始进程。更准确地说，我们只需要简单地增加一个窗口进程就可以了，这个进程能对定址于那个窗口的所有消息进行预处理——对它们进行过滤，并且最终把它们传递给类窗口进程，在其中完成标准的处理。我们必须把子类窗口想象成一种特殊的对象，它需要在类窗口进程开始之前进行一系列的准备工作。仍以前面那个例子为例，我们应该对它进行优化，使其在一个 Edit 窗口内只接受数字，不接受其他字符。一旦进行了子类处理以后，定制窗口进程就会对 WM_CHAR 消息进行跟踪，只把那些包含了数字的 WM_CHAR 消息传递给类窗口进程，其他所有 WM_CHAR 都不放行。

15.4.1 对一个编辑窗口进行子类处理

在本书附带 CD 的 Listing 15.3 里，大家可以找到一个名为 Numonly 的示范程序，它向我们展示了怎样对一个 Edit 窗口进行子类处理，令其只接收 0 到 9 之间的一个数字，其他输入都不予接收。从表面上看起来，图 15-17 显示的 Edit 窗口与一个标准的 Edit 控件并没有什么区别。然而，假如用户试图在其中键入除了 10 个数字以外的其他字符，Edit 窗口就会发出一声警告，同时不接受这个字符。

在客户区内建立好 Edit 窗口以后，我们需要立刻对其进行子类处理，方法是改变 GWL_WNDPROC 内存位置处的值，如下所示：

```
...
hwndEdit = CreateWindow ("Edit", NULL,
                    WS_CHILD | WS_VISIBLE | WS_BORDER | ES_
                        NUMONLY,
                    10, 35, 100, 25,
                    hwnd,
                    (HMENU) ID_NEWEDIT,
                    hInstance, NULL);
// focus on the EDIT window
SetFocus (hwndEdit);

// subclassing the edit window
lpfnOld = SubclassWindow (hwndEdit, DetourWndProc);
// passing the original window procedure address
```

```
SendMessage (hwndEdit, WM_PASSPROC, 0, (LPARAM) lpfnOld);
...
```

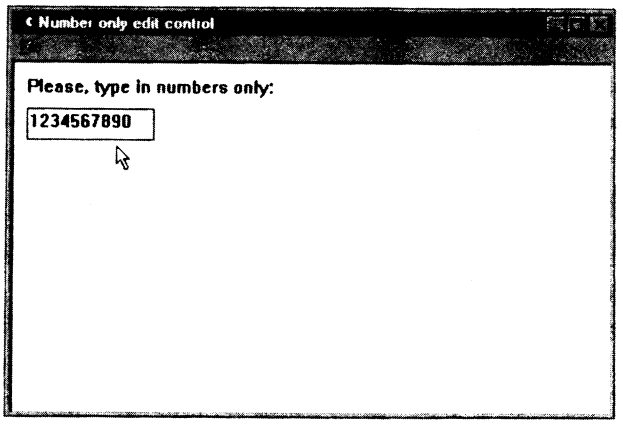


图 15-17　Numonly 程序向我们展示了如何对 Edit 窗口进行子类处理

SubclassWindow () 是一个简单的宏函数，其中封装了对 SetWindowLong () 的一次调用。其中，第一个参数是准备进行子类处理的那个窗口的句柄，接下来是新窗口进程的名称。假如我们希望通过调用 SetWindowLong () 对它进行编码，那么程序看起来就像下面这个样子：

```
...
// subclassing the edit window
lpfnOld = SubclassLong(hwndEdit, GWL_WNDPROC, (long)DetourWndProc);
...
```

SetWindowLong () 将返回预先与那个窗口关联在一起的窗口进程的地址。这种信息特别有用，因为在预先进行准备工作的那个窗口进程里，所有进入的消息经过过滤以后，将剩下一些合法消息。这些合法消息将传递给类窗口进程，在其中完成标准的处理。利用 API 函数 CallWindowProcedure ()，我们可以在某个窗口进程里调用另外一个窗口进程。它的语法结构如下所示：

```
#include <winuser.h>
LRESULT WINAPI CallWindowProc (WNDPROC lpfn,
                               HWND hwnd,
                               UINT msg,
                               WPARAM wParam,
                               LPARAM lParam);
```

参数	说明
WNDPROC lpfn	窗口进程的地址
HWND hwnd	窗口句柄
UINT msg	窗口消息
WPARAM wParam	附加的信息
LPARAM lParam	附加的信息
返回值	在正文里讨论
LRESULT	窗口进程返回值

CallWindowProcedure () 与一个标准的窗口进程函数几乎是完全一致的，只是它的第一个参数与众不同，这个参数指定了另外一个窗口进程的地址。这个 API 函数可以调用与某个窗口进程对应的一段特定的代码，然后把传统的四个参数传递出去。

这儿的关键问题是如何把原始类窗口进程的地址传递给一个特殊的函数。只要有一条消息发送给了经过子类处理的窗口，就会调用那个特殊的函数。实际上，要实现这些要求并不难，因为在这个时候，经过子类处理的窗口已经在自己的预约内存区域里存储了新窗口进程的地址。因此，我们只需要把应用程序定义的一条消息发送给经过子类处理的窗口，然后存储一个静态类的标识符，从而保证它永远都能从那段代码里进行访问。下面这个代码段向大家阐述了这种技术：

```
...
#define WM_PASSPROC      WM_USER + 100
...
// pass the original window procedure address
SendMessage (hwndEdit, WM_PASSPROC, 0, lpfnOld);
...
```

在新的窗口进程里，应用程序将跟踪并捕获 WM_PASSPROC 消息，在其中存储类型为 WNDPROC 的一个静态标识符，如下所示：

```
...
static WNDPROC pfn;
...
    case WM_PASSPROC:
    {
        pfn = (WNDPROC) lParam;
        ...
    }
    ...
    CallWindowProc (pfn, hwnd, msg, wParam, lParam);
```

```
}
...
```

显然，这并是唯一可行的办法。作为另外一种选择，我们也可以对类预约内存区域进行查询，从而获得原始的类地址（请记住，经过子类处理以后，我们只改变了一个单一窗口的窗口进程）。从这时开始，定址于这个窗口的所有消息都会首先到达新的窗口进程。然后经过一番过滤以后，也许有部分消息能够到达原始的窗口进程，在那里完成对它们的标准处理。

在 Numonly 示范程序里，我们拦截了 WM_CHAR 消息，从而判断用户键入的是哪个字符。ASCII 值低于 48 和高于 57 的都不允许到达 Edit 窗口类进程，这样就能避免它们在 Edit 窗口内显示出来。在编写对 WM_CHAR 消息进行控制的那一部分代码之前，我们先花些时间作好计划。

我们对一个预定义窗口进行子类处理的时候，应该提供相应的代码段，使其作为该窗口的预备窗口进程运行。这个代码段支持那个窗口的特定需求，同时，它也能为其他窗口提供服务。事实上，在某些情况下，我们最好为经过子类处理的几个窗口使用相同的例程，这样可以避免窗口进程的数量猛增。假如采取了这种方法，我们也许要出于两种尽管类似，然而并不完全一致的原因对属于同一类的两个窗口进行子类处理。举个例子来说，假设我们希望一个 Edit 窗口只接受数字输入，而另外一个 Edit 窗口则可以同时接受数字和字母。为它们分别设计两个不同的窗口进程是可行的，然而这两个进程之间的区别如此有限，以至于白白浪费了许多时间。

一种更好也更专业的方案是只实现一个窗口进程，用它来支持属于同一类的不同窗口的不同行为。为了对这种理论提供支持，请记住我们也许也要对两个 Edit 窗口进行子类处理——它们都只能接收数字输入。根据我们的方案，显然它暗示着要让这两个子类窗口都共享同一个窗口进程。除此以外，我们还要用到第三个窗口，用它同时接收数字和字母，并且它也有可能要共享相同的窗口进程。

为了完成所有这些要求，我们应该定义一种整体策略，从而判断针对一个指定子类窗口，需要在应用程序定义的窗口进程里对哪条消息进行跟踪并且捕获。这儿可以选用多种方案，但是其中最好的一种也许要算直接利用窗口风格。窗口风格是一种非常简单的设置，它能指示窗口采用一种特定的行为或者外观。对于预定义窗口类来说，它并没有全部用到 16 种风格，在它们中留下了一些空白。在这些空白的地方，我们可以定义一些新风格，使其只应用于属于一个特定类的某个子类窗口。

假如我们对表 9-11 里列出的所有 Edit 风格进行检查，就能发现 0x00000200 值还没有为这一类分配任何一种风格。因此，我们可以把它用作一种新类风格，用这种风格来指示窗口对 WM_CHAR 消息进行拦截，并且截留其中除了数字以外的其他所有字符。Numonly 程序沿用的正是这种思路，它定义了一种新的风格：ES_NUMONLY。当然，这种风格只能应用于这个例子里的一个子类窗口。

```
...
#define ES_NUMONLY 0x00000200
...
```

建立一个最终要进行子类处理的 Edit 窗口时，应用程序会增加一个 ES_NUMONLY 标

志，如下所示：

```
...
hwndEdit = CreateWindow (" Edit", NULL,
                         WS_CHILD | WS_VISIBLE | WS_BORDER | ES_
                             NUMONLY,
                         10, 35, 100, 25,
                         hwnd,
                         (HMENU) ID_NEWEDIT,
                         hInstance, NULL);
...
```

针对一个 Edit 窗口设置了 0x00000200 风格位以后，假如不对这个窗口进行子类处理，并且不提供一个对应的窗口进程，那么这种设置就是毫无用处的。我们在这儿的想法是在一个相对特殊的类窗口进程里设置这种风格（这种情况下是在子类窗口进程里设置的），把它当作能触发某种特定行为的一个开关来使用。相同的逻辑亦可应用于由应用程序定义的新风格。在经过子类处理的 Edit 窗口进程里拦截了 WM_CHAR 消息以后，应用程序就会寻找窗口预约内存区域内的一个 ES_NUMONLY 标志。只有在这个标志存在的前提下，应用程序才会检查 wParam 的内容，这样就避免了除数字以外的其他字符到达类窗口进程。正如大家通过下面这个代码可以看出的那样，设置了 ES_NUMONLY 标志以后，应用程序就会对用户的输入进行相应的过滤：

```
...
case WM_CHAR:
{
    if (GetWindowLong (hwnd, GWL_STYLE) & ES_NUMONLY)
        switch (wParam)
        {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
            case 8: // backspace
```

```
                    break;

                default:
                    MessageBeep (0);
                    return 0L;
            }
    }
        break;
    ...
```

正如我们看到的那样，子类窗口进程里的代码通常都比较简单，一般都限制在几行以内。这就是为什么我们在很多情况下都建立一个更为灵活的子类窗口进程的原因，它有能力实现几种很有用的附加功能，而不是简单地建立第二个窗口了事。

对于我们以前提到的那个能同时接收数字和字母的 Edit 窗口来说，一种很自然的方案是定义另外一个窗口风格，用它来标识这种能“兼收并蓄”所有字符的状态。注意这种情况仍然与用户的输入有关。拦截到 WM_CHAR 消息以后，应用程序就会检查用于接收进入消息的那个窗口的预约内存区域，从而判断自己应该实施哪种行为。

总而言之，我们在进行子类处理的时候应该记住下面两个要点：

▶规则一：针对属于由应用程序注册的某个类的一个窗口，对它进行子类处理是徒劳无益的。

▶规则二：为了对一个窗口进行子类处理，必须首先建立这个窗口。这就意味着在子类窗口进程里，我们已经不能对诸如 WM_CREATE 之类的消息进行拦截。新的地址值存储到预约内存区域以后，定址于一个子类窗口的消息紧接着就可以到达新的窗口进程。只有从这个时候开始，子类窗口才能表现出自己的新行为。

15.4.2 子类处理、回调函数和物主绘图

到现在为止，大家已经知道了怎样对一个预定义窗口进行子类处理，我敢打赌已经不只一位读者马上产生了以后要广泛运用这种技术的欲望。然而，通常情况下，我们只有在特别有必要的时候，才对一个窗口进行子类处理。进行这种处理的时候，要注意它或许也会给窗口的行为带来一些副作用，这主要是在对鼠标消息进行捕获的时候产生的。因此，我们应该把所有子类处理的影响都限制在最小范围以内。Windows 程序开发者正在不断地在扩展由预定义类提供的功能。不管怎样，Windows 95 的一个静态类都能“理解”用户在它上面的单击，并且相应地通知父窗口。

子类处理应该考虑成对窗口行为进行修改的最后一种方案来实施。API 提供了其他一些很有价值的选择方案，比如回调函数以及物主绘图等。某些新的通用控件提供了对回调函数的广泛支持，这种函数是位于应用程序代码内的一些例程。发生了某种特定的事件以后，控件就会自动调用这种函数。在以前的章节里，大家已经碰到过这种控件与应用程序进行集成的例子。物主绘图控件的行为类似于一个经过子类处理的窗口。物主绘图对象可以把所有输入信息都传递给自己的父/物主窗口，允许它对某些内部信息进行访问——除了那些确实不能

访问的以外。

根据经验来看，在实际的编程环境中，大家应该首选回调函数或者物主绘图控件，而不是首先考虑对窗口进行子类处理。

15.5 超类处理

超类处理技术为我们提供了一种功能更强的方式来建立一种崭新的窗口类，这种类仍是以预定义窗口类为基础的。显然，我们对预定义窗口类的一种简单的复制品并不感兴趣，因为它不会为应用程序带来任何新东西。超类处理背后的逻辑是在兼容预定义窗口类的基础上，对它进行一些细微的修改。和子类处理比较起来，这种方案的优点是明摆着的。注册了这样的一种新类以后，应用程序开发者就可以属于这一类的新窗口。这些窗口同样将运用由原始类窗口进程提供的逻辑；只是在这之前，它需要先在另外一个窗口进程内进行处理。

整个进程起始于获取与准备进行超类处理的那个窗口类有关的信息。这个任务既可以通过调用 GetClassInfo（）来完成，亦可通过调用 GetClassInfoEx（）来完成。这两个函数分别需要填写一个 WNDCLASS 和 WNDCLASSEX 数据结构，填写的信息涉及到类注册时存储在 Windows 里的几乎所有的相关信息。

```
#include〈windows.h〉
BOOL GetClassInfoEx（HINSTANCE hInstance，
                     LPCTSTR lpszClass，
                     LPWNDCLASSEX lpwcx）；
```

参数	说明
HINSTANCE hInstance	用于指定已经注册了目标类的那个应用程序的句柄
LPCTSTR lpszClass	类名称
LPWNDCLASSEX lpwcx	一个 WNDCLASSEX 数据结构的地址
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

如果我们希望从一个预定义窗口类里取得信息，就需要把应用程序的实例句柄设置成 NULL。第二个参数是类名称，这是一个标准的正文串，比如“edit”等等，注意它是大小写相关的。接下来是一个 WNDCLASSEX 标识符的地址。假如函数调用成功，整个数据结构就会用类信息进行填充，但这些信息里并未包括 lpszMenuName 项的设置。为了取得这种数据，我们应该调用 GetClassLong（），并在其中设置 GCL_MENUNAME 定义。由于预定义窗口类不提供对菜单栏的支持，所以通过调用 GetClassInfoEx（）函数获得的信息就足够我们对一个类进行超类处理了。

在这个时候，我们应该对存储于 WNDCLASSEX 数据结构里的某些值进行少许改动。显然，我们必须改变类名称。否则，当 RegisterClass（）试图注册一个已经存在的类时，就会返回一条错误信息。除此以外，也是最重要的一点，我们应该为 hInstance 项分配当前应用程

序的实例句柄。这种信息是相当关键的，没有它就无法对一个新类进行正确的控制。进程中断以后，一个应用程序里注册的所有类都会自动解除注册，从而释放它们在 Windows 内部数据区里占据的内存空间。所有预定义类（包括通用控件）都已经由 Windows 注册好了，其中的 hInstance 项被设置成 NULL。保持这个值可以避免进程中断以后自动放弃对这个类的注册。

第三项重要的改动是对 GetClassInfo () 取得的信息实行的。这种信息关系在类窗口进程的地址。和子类处理类似，应用程序应该提供一个新的窗口进程，以便在消息传送给类进程之前对它进行调用。这种方案在超类处理的时候基本上也是适用的，只是存在少许差异则已。在这个时候，我们需要在一个类级别上提供新窗口进程的地址，而不是在窗口内存区域里提供。这就意味着应用程序可以建立一个经过超类处理后的新窗口，它在缺省情况下会把消息传送给由一个应用程序支持的窗口进程，最终将调用原始的类窗口进程。这样一来，成功地注册一个超类并且建立好窗口以后，应用程序就可以对与那个窗口有关的所有消息进行拦截，根本没有任何限制。

下面这个代码段是 Numonly 示范程序的修订版本里摘录下来的，这个程序可以在本书附带 CD 的 Listing 15.4 里找到。Numonly 的这个修订版本将在 Edit 预定义类的基础上注册一个新的窗口类。

```
...
// retrieve the EDIT class information
GetClassInfoEx (NULL, " EDIT", &wc);

// modify it
lpfn = wc.lpfnWndProc;
wc.lpfnWndProc = NewEditWndProc;
wc.hInstance = hInstance;
wc.lpszClassName = NEWEDIT;

// registering NEWEDIT
if (! RegisterClassEx (&wc))
...
```

在 WNDCLASSEX 数据结构里，只有三个项的设置被改动了。假如 RegisterClassEx () 调用成功，一个新的窗口类就会增添到应用程序可访问的类池里。Numonly 将在自己的客户区内显示出属于新窗口类 Newedit 的五个窗口，如图 15-18 所示，然后把输入焦点集中在第一个上面。

拦截到 WM_CREATE 消息以后，应用程序将按照传统的方式建立五个控件：

```
...
for (i = 0; i < 5; i++)
```

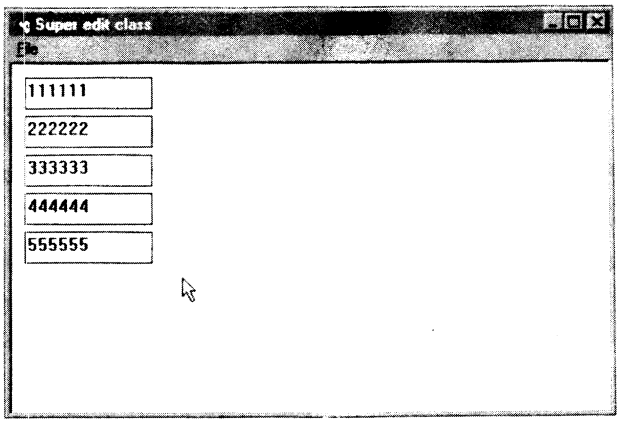


图 15-18　NUMONLY 程序的这个修订版本可以建立五个编辑控件窗口，它们都属于 NEWEDIT 类，并且只能接收纯数字输入

```
{
    CreateWindow (NEWEDIT, NULL,
                  WS_CHILD | WS_VISIBLE | WS_BORDER | ES_
                      NUMONLY,
                  10, 10 + i * 30,
                  100, 25,
                  hwnd,
                  (HMENU) (ID_NEWEDIT + i),
                  hInstance, NULL);
}
// focus on the first EDIT window
SetFocus (CTRL (hwnd, ID_NEWEDIT));
...
```

和以往一样，新建立的类支持 ES_NUMONLY 风格，这个风格可以强迫窗口进程拦截 WM_CHAR 消息，并且把用户的输入，除了数字以外的所有字符过滤掉。

应用程序中断以后，它就会撤消对超类的注册，这种撤消是通过调用 UnregisterClass () 函数来实现的。正如我们以前提到的那样，这种操作并非必需的，尽管它不会带来任何负作用。

15.6　关于超类的一些考虑

什么时候应该进行子类处理，什么时候应该进行超类处理呢？为了作出这种选择，大家在刚开始的时候也许会觉得难以下手。如果一个或者数量有限的几个窗口准备提供某些附加的功能，同时这些功能最初并未在一个预定义类里计划好，这时就应该考虑进行子类处理。子类处理很容易就可以实现，而且也非常有效，而且最重要的一点是它并非永远存在。事实上，

一个应用程序很容易就可以恢复存储于窗口预约内存区域里的原始窗口进程地址，这只需要调用 SetWindowLong（）或者 SubclassWindow（）函数即可。

超类和 DLL

假如开发者准备建立几个窗口，并且用一个甚至多个模块为它们提供附加的功能，那么这个时候超类处理被证明是最有效的。在这种情况下，倘若我们把整个进程都放在一个单一的新类注册里，那么这种处理显然能提供更多的方便。甚至还能更好，我们可以考虑把整个代码放在一个 DLL（动态链接库）里。根据这种方式，几个进程都可以利用一个新建并且经过优化的窗口类。事实上，当今针对通用控件的编程趋势正是这样的。这些类的集合都存储于一个 DLL 里，即 COMMCTRL. DLL，其中包含了整个注册进程以及相关的窗口进程。假如我们希望采取这种处理方式，那么需要注意的唯一问题是：对于包含了一个新窗口类（也许是一个超类）的 DLL 来说，它不能通过一条重定位记录与应用程序代码自动链接到一起。到目前为止，这个概念应该相当明显了。然而，还是让我们在下面对它进行一番简短的叙述。

DLL 是存放新窗口类的一个很方便的地方。注册进程可以在入口点处进行，即 DllMain（）里。然而，我们我们在这儿选择的是类窗口进程，因为一个输出函数可以从模块以外进行访问。DLL 里也许还包含了其他例程和函数，然而，在通常情况下，它最好只用于存储一个或者多个类以及它们相关的窗口进程——这正是我们在 COMMCTRL. DLL 里采取的策略。

根据这种设计方案，我们可以推断这样的一个 DLL 里不会包含能从其他进程内调用的其他任何例程。这样就导致了一个问题：在建立一种窗口之前，应用程序必须先载入一个 DLL，这个 DLL 里只包含了那种窗口从属的一个或者多个窗口类。假如大家仔细考虑这个问题，就会发现这与通用控件的情况正好是符合的。在建立一个通用控件之前，我们必须执行 InitCommonControls（）函数，这个 API 的唯一用途就是把 COMMCTRL. DLL 载入活动的进程地址空间。因此，把一个超类放置到 DLL 里是非常聪明的一着棋，尽管它要求应用程序明确调用包含了它的输出的一种服务，否则便无法载入 DLL。针对 DLL 的载入，我们有两种方案可以选择：既可以通过调用 LoadLibrary（）来装载 DLL，也可以在 DLL 里实现一种伪服务，以便让链接程序增加针对应用程序的一条重定位记录。针对这种伪服务，一个不带任何参数的空函数便是最简单的方式。

总而言之，超类处理是一种功能强大的技术，它可以让我们的应用程序充分利用预定义窗口类的现成特性，并能把这些特性与新的特性集成到一块儿，这样就简化了一个复杂应用程序的开发。假如我们需要在数据条目很集中的应用程序里建立一个复杂的对话框，超类处理被证明是最有效的一种解决方案。

下面是进行超类处理时需要牢记的两条规则：

▶规则一：假如某个窗口属于应用程序源代码内注册的一个窗口类，千万不要对这个窗口进行超类处理。

▶规则二；建立了超类以后，我们就对应用程序源代码内能够访问的类集合进行了延展，这样就自始至终对属于那一类的某个窗口进行控制。

作为最后个项考虑，在注册新类之前，我们也许希望对原始 WNDCLASSEX 数据结构内的其他项进行修改。请记住把一些重要的信息保存下来，比如类和窗口的附加字节等等。假如我们需要对类或者窗口级别的更多信息进行封装，就可以对附加字节内存区域进行扩展，这样就不用担心覆盖对于原始窗口类来说很关键的一些信息。举个例子来说，假如我们准备在

一个超类窗口的附加字节内存区域内存放一个窗口句柄，首先应该把句柄的长度增加到以前的附加字节长度上面，如下所示：

```
...
wcx.cbWndExtra += sizeof (HWND);
...
```

为了获得一种简单和有效的方式，从而对超类窗口中的附加字节进行控制，前面这种操作是不够的。事实上，尽管像前面那样增加更多的空间来容纳一个窗口句柄是技术上是无可挑剔的，但在对附加字节内存区域进行动态管理的时候，这种方式会带来一个重要的问题。这儿的问题在于应用程序需要某些空间来存储属于最初那个类的附加字节区域的长度，以便判断超类附加字节实际的起始位置。因此，一种考虑周全的方案应该是把附加字节的长度存储到一个整数标识符里，然后用合适的幅度来增大 cbWndExtra 项的设置值，如下所示：

```
...
iExtraBytesOffset = wcx.cbWndExtra;
wcx.cbWndExtra += sizeof (HWND);
...
```

其中，iExtraBytesOffset 变量内包含了超类附加字节实际的起始位置。

15.7 消息流

在本书附带 CD 的 Listing 15.5 里，大家可以找到一个名为 Msgflow 的示范程序，它的作用是列出流向类窗口进程的所有消息，这些消息都是通过对 CreateWindowEx () 函数的一次简单调用而生成的。启动 Msgflow 程序以后，屏幕只会显示一个空白的窗口，如图 15-9 所示。

假如这时在 Action 顶级菜单内选择 Start 菜单项，就会建立另外一个窗口，同时主窗口将开始列出到达类窗口进程的所有消息，这些消息都是建立 Tester 窗口的过程中产生的。如图 15-20 所示。

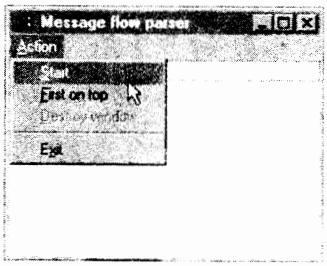


图 15-19 MSGFLOW 是一个简单的窗口，它的客户区内可以列出建立一个新窗口时到达窗口进程的所有消息的名字

Action 菜单可以对消息插入的顺序进行控制。在缺省情况下，最后一条消息将显示于 Msgflow 客户区的顶部。假如我们希望改变这种行为，只需要在 Action 菜单内选择 First on top 菜单项就可以了，它的意思是把第一条消息显示于客户区顶部，如图 15-21 所示。

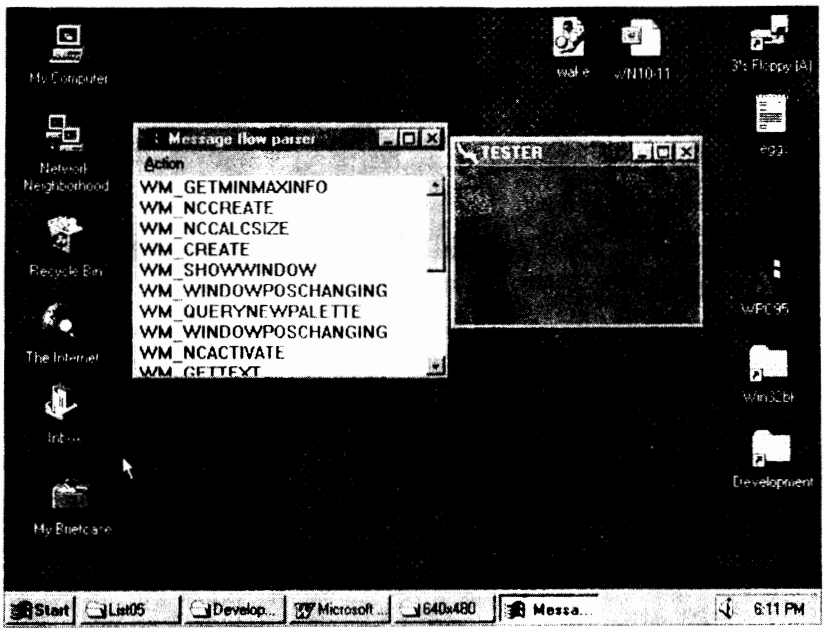


图 15-20　建立 Tester 窗口的时候，主窗口会开始填写由 CreateWindowEx（）发送给类窗口进程的所有消息

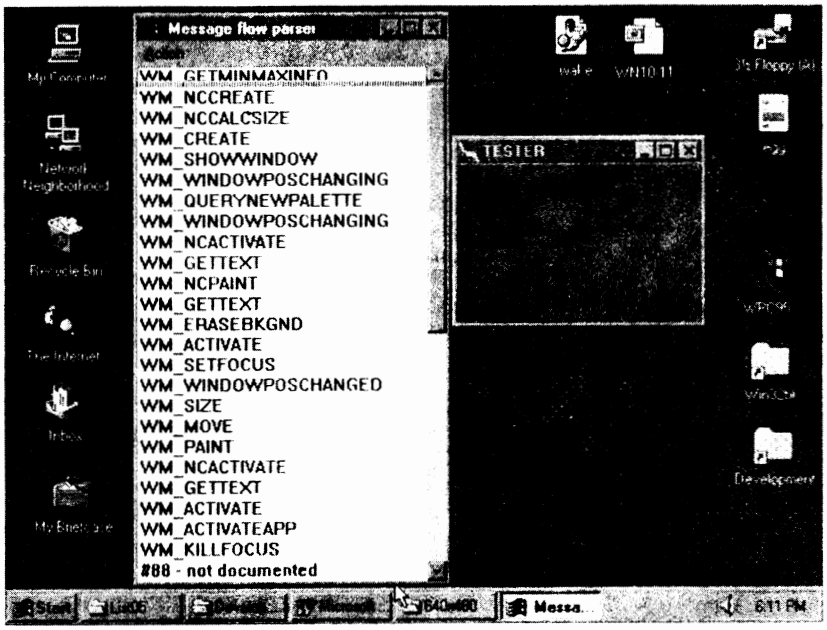


图 15-21　调用 CreateWindowEx（）时生成的所有消息将按照从第一条到最后一条的顺序排列

Tester 窗口属于与应用程序主窗口不同的一个类，因为它具备自己专用的窗口进程。在这个窗口进程里，我们不会用特定的 case（条件）代码块对任何标准的 WM_消息进行处理。更准确地说，所有 WM_消息都会到达标准 switch（切换）代码块的缺省位置。在这儿，应用程序将从资源文件里载入与正在处理的消息对应的正文串，然后把它插入主窗口客户区内，如图 15-21 所示。

```
...
LRESULT WINAPI ClientWndProc (HWND hwnd,
                              UINT msg,
                              WPARAM wParam,
                              LPARAM lParam)
{
    char szMsg [30];
    static HWND hwndListbox;
    static int iOrder;
    static HINSTANCE hInstance;

    switch (msg)
    {
        case WM_PASSPROC:
        {
            hwndListbox = (HWND) wParam;
            hInstance = (HINSTANCE) LOWORD (lParam);
        }
            return FALSE;

        case WM_ORDER:
        {
            iOrder = wParam * -1;
        }
            return FALSE;

        default:
        {
            if (! LoadString (hInstance, msg, szMsg, sizeof (szMsg)))
                wsprintf (szMsg, " #%x - not documented", msg);
        }
            break;
    }
```

```
        ListBox_InsertString (hwndListbox, iOrder, szMsg);
        return DefWindowProc (hwnd, msg, wParam, lParam);
    }
    ...
```

其中，WM_PASSPROC 和 WM_ORDER 消息是插入的伪消息，它们是在源代码内定义的。这两条消息的作用分别是把主窗口句柄传递给应用程序实例句柄，以及传递一个排序参数，从而对主窗口内的消息插入顺序进行定义。

STRINGTABLE（串表）资源内的消息正文串已经从不同的头文件里提取出来了，这些头文件包含于微软公司提供的 Win32 开发环境里。

15.8 控制面板对象

在本书附带 CD 的 Listing 15.6 里，大家可以找到一个名为 Welcome 的示范程序，它使我们有机会对某些特殊的编程问题进行探讨，这些编程问题涉及到系统外壳、通用控件以及进程等等。运行这个程序以后，它可以在一个列表视窗控件内显示出系统文件夹内提供的所有 CPL 对象，如图 15-22 所示。

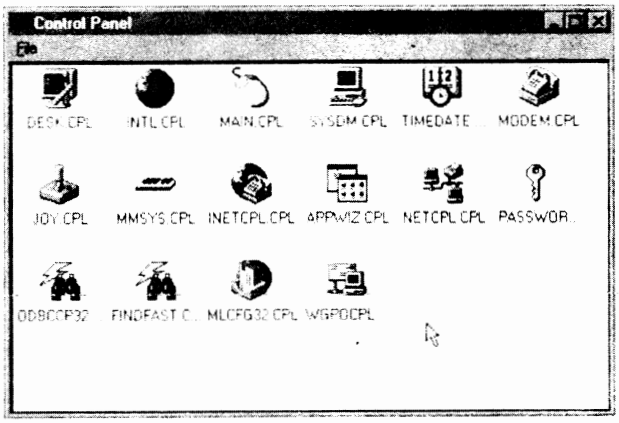


图 15-22 Welcome 显示了带有 .CPL 扩展名的所有应用程序，这些程序都驻留于控制面板文件夹内

.CPL 文件就是我们常说的控制面板应用程序，设计它们的宗旨是帮助用户对 Windows 95 系统进行配置。从技术的角度来看，.CPL 模块的本质就是一种动态链接库（DLL），它是按照某些特殊的规则开发出来的，只是最后把它命名为 .CPL 而已。在缺省情况下，.CPL 模块都包含于“系统”文件夹（\WINDOWS\SYSTEM）内，并且与系统应用程序 Control.EXE 严格相关。事实上，我们可以在 Run（运行）这个编辑控件内键入一个命令行，从而直接启动和激活一个 .CPL 模块，如图 15-23 所示。Control.EXE 是一种系统应用程序，它的任务是对所有 .CPL 模块进行管理。尽管如此，只要我们按照某些特定的规则，那么任何 Win32 进程都可以载入一个 .CPL 模块（一种 DLL 对象）。至于这些规则，我们稍后就会进行详细的讲述。

实际上，我们在这儿列出的.CPL 对象要少于系统控制面板文件夹内显示的对象总数，如图 15-24 所示。把图 15-24 与图 15-22 进行比较，我们立刻就可以看出某些.CPL 模块内包含了控制面板内实际显示出来的多个对象。

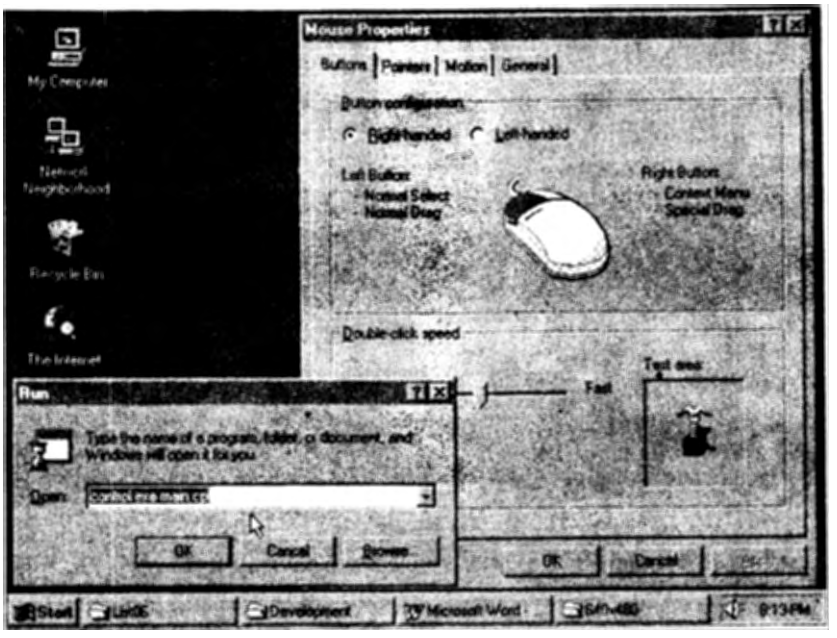


图 15-23　通过 Start 菜单的 Run 选项，我们可以直接执行 MAIN.CPL

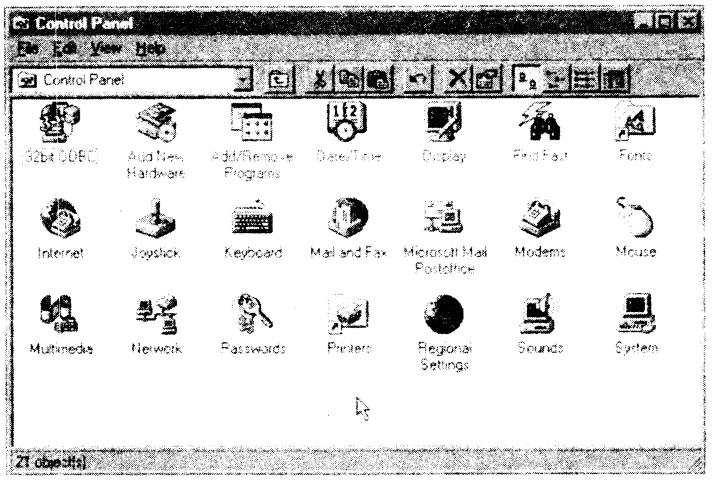


图 15-24　系统控制面板文件夹内显示了比 System 文件夹内 CPL 模块更多的对象

之所以 Welcome 应用程序与控制面板之间产生了这种显示上的区别，主要是由于某些.CPL 对象的本质造成的。正如我们在前面提到的那样，一些.CPL 对象实际是一系列属性表

的集合。例如，Main.CPL 内包含了与鼠标、键盘、字体文件夹、打印机文件夹、电源管理对象以及 PC Card 控制器有关的属性表。控制将单独显示这些对象，但是这种效果通过简单地列出系统文件夹内的 .CPL 文件是不可能获得的。图 12-25 向我们展示了显示 Power（电源管理）属性表所需的语法结构。从图中看出，我们需要按照下面这种语法来调用 Control.EXE：CONTROL.EXE MAIN.COL POWER。

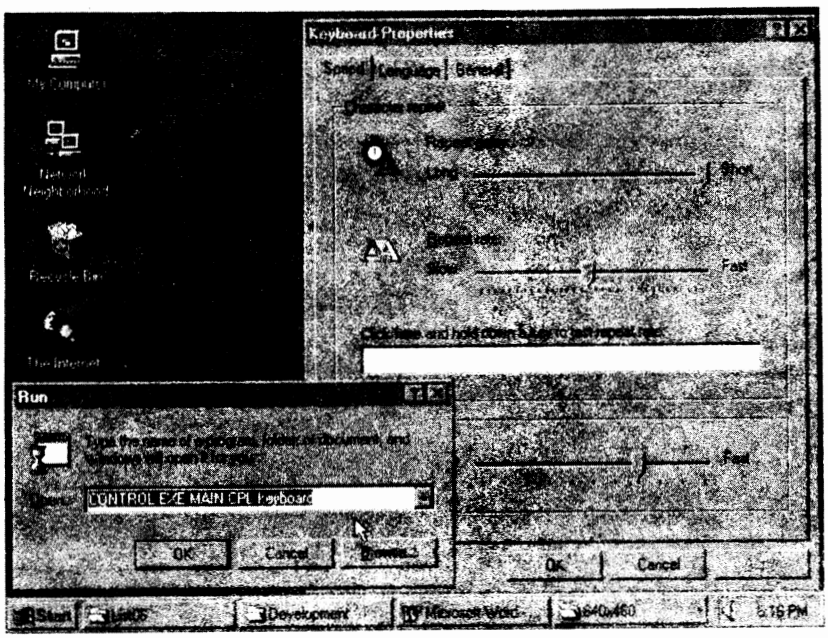


图 15-25　Power 属性表的执行

对于 Welcome 示范程序与控制面板内显示的对象来说，它们之间的另外一处差异与图标有关。在两幅图中，相同的对象是用不同的图标表示的。这种差异是由于应用程序对一个 .CPL 模块进行控制和管理时采用的标准不同造成的。在 Welcome 示范程序里，我决定显示存储于每个 DLL 资源段内的第一个图标。控制面板则不同，它有时会选用第一个图标，有时却会选用第二个图标。由于存在多个图标，所以我们可以考虑对每个 .CPL 模块进行查询，获得与它关联在一起的所有图标的完整集合，然后在一个独立的列表视窗内把这些图标显示出来。假如在图 15-23 显示的任何一个 .CPL 对象上方单击鼠标右键，就会显示出一个弹出式菜单，如图 15-26 所示。其中的两个选项允许我们既可以通过控制面板载入那个特定的 .CPL 模块，也可以显示出与它相关的所有图标。假如选择的是 Change icon 菜单项，就会引入一个弹出式列表视窗，其中列出了存放于那个控制面板应用程序内的所有图标。

Welcome 程序的客户区内覆盖了一个列表视窗窗口，它刚开始的时候是空白的。用户可以选择 Populate 菜单项在其中填充 .CPL 模块图标。这种选择强迫应用程序对 System 目录进行浏览，在其中搜索带有 .CPL 扩展名的所有文件。一旦侦测到某个 .CPL 动态链接库，它的图标就会一个接一个地提取出来，然后放置到与列表视窗控制关联在一起的图象列表内。这种搜索和图标提取工作结束以后，图象列表就包含了与不同 .CPL 对象有关的所有图标。

每当在列表视窗里插入一个新的控制面板应用程序时，应用程序就会在一个 LV_ITEM

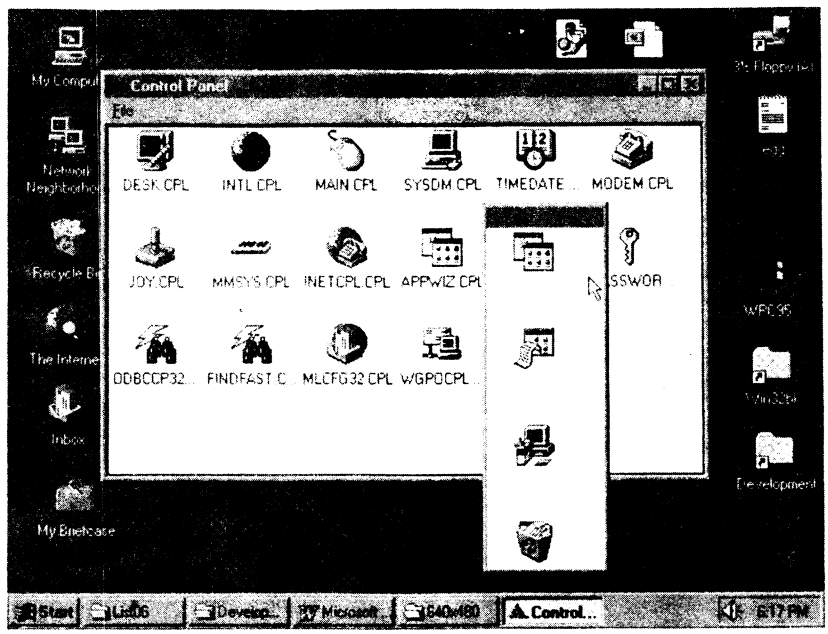


图 15-26　假如在某个图标上方单击鼠标右键，另外一个列表视窗就会显示出来，并在其中列出与那个对象有关的所有图标

结构的 lParam 项内插入两个 16 位数值，并且把它们合成为一个 32 值。其中，低字内的信息指定了与那个对象的第一个图标关联在一起的图象列表索引。高字内则包含了与那个图标有关的图标总数。利用这两个值，我们以后可以在用户选择了 Change icon 菜单项后显示出与某个对象关联在一起的图标子集，如图 15-26 所示。

```
...
do
{
    // storing the CPL app name
    strcpy (p, ffd.cFileName);
    iLen = strlen (ffd.cFileName) + 1;

    // extracting the icon
    iTotIcons = (int) ExtractIcon (hInstance, ffd.cFileName, (UINT) -1);
    // extract all the icons
    for (i = 0; i < iTotIcons; i++)
    {
        hicon = ExtractAssociatedIcon(hInstance, ffd.cFileName, &(WORD)i);
        iIcon = ImageList_AddIcon (himg, hicon);
        // destroying the icon
```

```
            DeleteObject (hicon);
        }

        // inserting items
        lvi.mask = LVIF_TEXT | LVIF_IMAGE | LVIF_STATE | LVIF_
            PARAM;
        lvi.state = 0;
        lvi.stateMask = 0;
        lvi.cchTextMax = iLen;
        lvi.iItem = iItem++;
        lvi.iSubItem = 0;
        lvi.pszText = p;
        lvi.iImage = (int) sStart;
        lvi.lParam = (LPARAM) MAKELPARAM (sStart, (short) iIcon);

        // inserting an item
        if (ListView_InsertItem (hwndLV, &lvi) == -1)
            return FALSE;

        // starting index in imagelist
        sStart = (short) iIcon + 1;

        // update p
        p += iLen + 1;
    } while (FindNextFile (hFind, &ffd));
    ...
```

每个图标下方显示的正文是用大写字母表示的 .CPL 文件名。发生了鼠标右键单击事件以后，就会强制性地显示出一个小的弹出式菜单，其中只带有两个菜单项。选择 Open 菜单项就会执行 Control.EXE，同时载入选中的 .CPL 模块。在这以后，一个新的 Control.EXE 拷贝就会在系统内运行起来，同时会载入相应的控制面板应用程序，并在屏幕上把它显示出来。关于这方面的信息，请大家参考图 15-27。

靠近选中控制面板应用程序的那个弹出式列表视窗里列出了所有相关的图标。这个列表视窗是作为第二个，也是独立的一个弹出式窗口来实现的。从程序设计的眼光来看，在顶级窗口里建立一个单一的列表视窗，既有自己的优点，也存在一些缺点。就优点来说，对 CreateWindowEx () 的一次调用就足以建立一个独立的对象，并令其运行起来。就缺点来说，这个通用控件没有办法把自己的通知代码传递给应用程序代码的任何一个部分。显然，它在这种情况下缺少一个父窗口，因此我们必须采用另一种途径对通知代码进行跟踪。

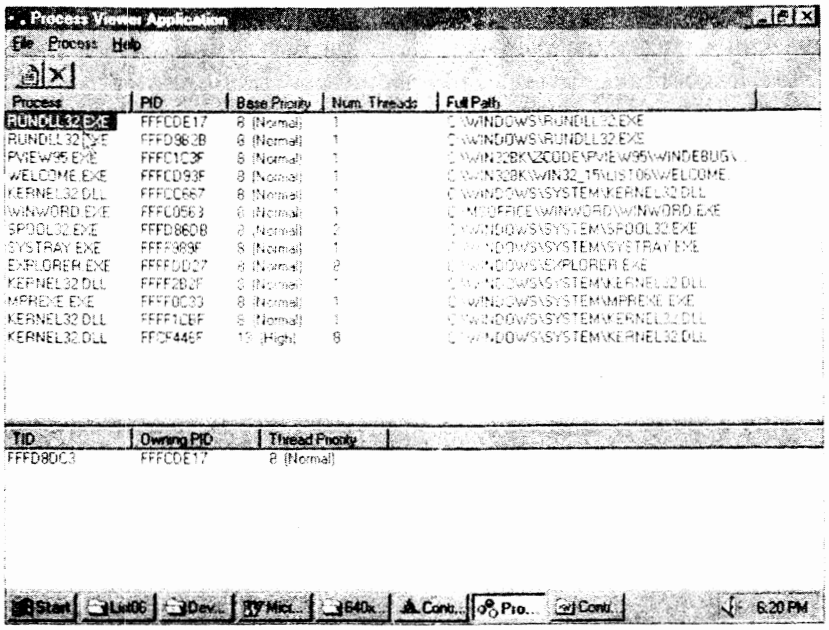


图 15-27　PVIEW95 证明每个控制面板对象的行为都类似于一个独立的模块——CONTROL.EXE

一种方案是对弹出式列表视窗进行子类处理，从而把进入的所有消息都传递给由应用程序提供的一个窗口进程。在这种特殊的情况下，对列表视窗进行子类处理需要程序开发者付出巨大的努力。在以前，我们对 WM_NOTIFY 消息以及相关的数据结构进行处理就可以得到满意的结果，但是现在为了得到相同的结果，我们必须在本地进行局部性的处理。刚开始的时候，我就是按照这种方案来操作的，随后我仔细检查了与弹出式列表视窗进行交互作用时在主窗口进程内发生的情况。假如大家仔细考虑这个问题，就能发现这个弹出式窗口也有一个物主——即主窗口。这个窗口真正缺少的是一个 ID，我们对来自一个 Win32 控件内的通知代码进行处理时，这种 ID 信息是相当关键的。

由于考虑到主窗口是弹出式列表视窗的物主，在好奇心的驱使下，我检查了由后者生成的通知代码是否也传递给了主窗口进程。在这以后，我确定了这正是在 Welcome 以及 Windows 95 里发生的情况。唯一的问题在于弹出式列表视窗根本没有自己的 ID。实际上，这种说法也不是百分之百准确的，因为在我们建立这个弹出式窗口的时候，CreateWindowEx() 的 menu 参数分配的是一个 NULL 值。因此，看起来有可能让这个窗口把 0 当作自己的 ID 使用。这正是我们在 Welcome 程序里采取的策略。弹出式窗口可以通知主窗口，同时在 wParam 里把零值设置成自己的 ID。因此，我们对这个窗口进行子类处理是根本没有用处的。在知道伪弹出式窗口 ID 的值是 0 以后，对恰当的通知代码进行跟踪就已经足够了（在源代码内，我把 CT_POPLISTVIEW 定义成零值）。

```
...
case WM_NOTIFY:
```

```
{
    switch (wParam)
    {
        case CT_POPLISTVIEW:
        {
            NM_LISTVIEW *pnmv = (NM_LISTVIEW *) lParam;

            switch (pnmv -> hdr.code)
            {
                case NM_DBLCLK:
                {
                    ...
                }
                    break;
            }
        }
            break;

        case CT_LISTVIEW:
        {
            NM_LISTVIEW *pnmv = (NM_LISTVIEW *) lParam;

            switch (pnmv -> hdr.code)
            {
                case NM_RCLICK:
                {
                ...
            }
            ...
        }
        ...
    }
        break;
...
```

为了改变一个对象的图标，我们只需要双击它的某个可选图标就足够了，应用程序会对主列表视窗内提供的.CPL 模块进行更新。两个列表视窗都共享同一个图象列表，这种策略极大地简化了对控制面板应用程序以及与它关联起来的可选图标的表示，如图 15-28 所示。为主列表视窗内的选中对象分配新图标是一种相当简单的任务，因为我们现在可以利用来自于两

个控件内的通知代码。

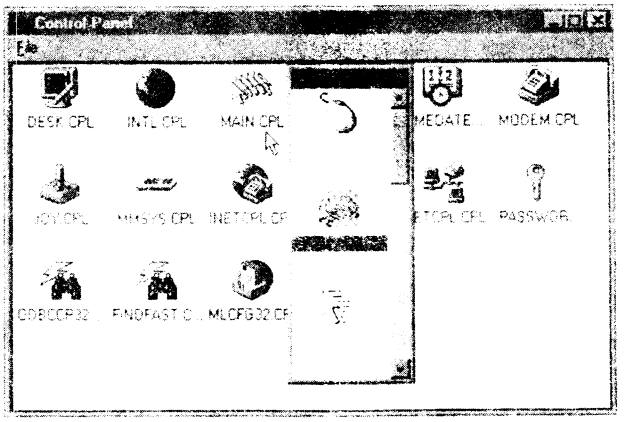


图 15-28　改变对象的图标需要在弹出式列表视窗内显示的某个可选图标上面双击

下面这个代码段是从 Welcome 示范程序里摘录下来的。首先，它将判断主列表视窗和图象列表内选中的项目是否与这个窗口链接起来了。随后，它将取得弹出式窗口内选中图标的索引编号。最后一些步骤则涉及到为选中的 .CPL 模块分配相应的图象。

```
...
case CT_POPLISTVIEW:
{
    NM_LISTVIEW *pnmv = (NM_LISTVIEW *) lParam;

    switch (pnmv -> hdr.code)
    {
        case NM_DBLCLK:
        {
            int iCtlLV, iCtl;
            LV_ITEM lvi, lvi2;
            HIMAGELIST himg;
            HWND hwndLV, hwndLV2 = pnmv -> hdr.hwndFrom;
            char szText [100];
            int i = 0;

            // main list view handle
            hwndLV = CTRL (hwnd, CT_LISTVIEW);
            // determine the selected item
            iCtlLV = ListView_GetNextItem (hwndLV, -1, LVNI_
```

```
                SELECTED);
            // retrieve the associated image list
            himg = ListView_GetImageList (hwndLV, LVSIL_NORMAL);

            // retrieve information from the selected object in the main list view
            lvi.mask = LVIF_TEXT | LVIF_PARAM | LVIF_IMAGE;
            lvi.iItem = iCtlLV;
            lvi.iSubItem = 0;
            lvi.pszText = szText + strlen (szText);
            lvi.cchTextMax = sizeof (szText) - strlen (szText);
            ListView_GetItem (hwndLV, &lvi);

            //determine the selected item int he popup list view
            iCtl = ListView_GetNextItem (hwndLV2, -1, LVNI_
                SELECTED);

            // retrieve information from the selected object in the main list view
            lvi2.mask = LVIF_TEXT | LVIF_PARAM | LVIF_IMAGE;
            lvi2.iItem = iCtl;
            lvi2.iSubItem = 0;
            lvi2.pszText = szText + strlen (szText);
            lvi2.cchTextMax = sizeof (szText) - strlen (szText);
            ListView_GetItem (hwndLV2, &lvi2);

            // change the icon in the selected item
            lvi.mask = LVIF_IMAGE;
            lvi.iItem = iCtlLV;
            lvi.iSubItem = 0;
            lvi.iImage = lvi2.iImage;
            ListView_SetItem (hwndLV, &lvi);
        }
            break;
    }
}
    break;
...
```

在接下来的小节里，我们将针对如何在一个应用程序里明确地载入.CPL 模块进行探讨。这种处理的目标是绕过 Control.EXE，并且设计另外一种方案，从而显示出单独一个 CPL 文

件里包含的所有属性表对象。在进行这些讨论的同时，我将介绍 Control.EXE 的工作原理。

15.9 建立一个应用程序来载入 .CPL 模块

控制面板应用程序是一种动态链接库 (DLL)，其中包含了建立、显示和管理屏幕内某个属性表对象所需的所有代码。除了这些显然需要完成的任务以外，.CPL 源文件还包含了 CPlApplet () 回调函数，它的结构如下所示：

```
#include <cpl.h>
LONG APIENTRY CPlApplet (HWND hwndCPl,
                         UINT uMsg,
                         LONG lParam1,
                         LONG lParam2);
```

参数	说明
HWND hwndCPl	对一个 .CPL 模块进行控制的应用程序的窗口句柄
UINT uMsg	控制面板应用程序消息 (参考表 15-10)
LONG lParam1	附加信息
LONG lParam2	附加信息
返回值	在正文里讨论
LONG	取决于发送出去的消息

回调函数在应用程序(对 .CPL 模块进行控制的那个)与每个属性表之间充当了一种链接器的角色。其中，四个参数的语法根据传递给回调函数的消息不同而发生变化。表 15-10 列出了所有 CPL_消息，这些消息是由应用程序控制一个 CPL 模块而发出的，利用它们可以取得某些信息以及激活特定的模块。

对整个过程进行详尽的解释之前，我们最好先澄清一下对某些术语的理解。对于载入 .CPL 模块的那个应用程序来说，从现在开始，我们把它定义成“控制器”(controller)。根据我们以前的知识，一个 .CPL 模块就相当于一个 DLL。从这个角度来看，这两个术语是可以互换使用的。LoadLibrary()是返回模块句柄的一种有效方案。一旦我们核实了这种信息的有效性，就可以通过查询 GetProcAddress()，从而判断出 CPlApplet()函数的地址。从这个时候开始，控制器就会通过直接调用 .CPL 模块内的 CPlApplet()例程，从而传递 CPL_消息。

表 15-10 控制面板应用程序消息

控制面板消息	值	说 明
CPL_INIT	1	对控制面板应用程序进行初始化
CPL_GETCOUNT	2	返回一个 .CPL 模块内的对象数目
CPL_INQUIRE	3	已废弃不用
CPL_SELECT	4	指出选定的图标代表某个 .CPL 模块
CPL_DBLCLK	5	用户双击与一个 .CPL 模块对应的图标时发出
CPL_STOP	6	发送给每个活动的 .CPL 模块，从而指出控制应用程序准备中断
CPL_EXIT	7	在控制应用程序释放一个 .CPL 模块之前发出
CPL_NEWINQUIRE	8	取得关于某个对话框的信息，那个对话框内包含了与一个系统组件有关的信息
CPL_STARTWPARMS	9	

由于存在专门针对与.CPL 模块的通信而设计的消息，所以大家也许会产生某些误解。事实上，我们建议大家通过一个标准的 SendMessage () 调用把这些消息发送给接收窗口。现在的问题在于，某个.CPL 模块载入一个进程地址空间以后并不存在任何窗口。在控制器与.CPL 模块之间进行通信的唯一途径就是直接调用 CPlApplet () 函数，并且每次调用时都包含一条 CPL_消息。

准备通过 CPlApplet () 函数发出的第一条消息是 CPL_INIT。它的任务是对 DLL 进行初始化。对这条消息进行处理的时候，我们几乎不可能判断一个.CPL 模块内发生了什么情况。我知道的一切就是模块准备对它的返回值进行处理。任何正值都意味着初始化工作成功地完成了。在这以后，CPL_GETCOUNT 消息将强迫.CPL 模块返回所含属性表的数量。举个例子来说，假如把这条消息发送给 Main.CPL，就会返回 6，这在以前已经看到过了。准备阶段的最后一项工作是发送 CPL_NEWINQUIRE 消息，从而对模块内存在的每个对象（属性表）进行初始化。这条消息需要用到一个 Newinquire 数据结构的地址，.CPL 模块会在这个结构内填写一些适当的信息，比如用于在屏幕上代表这个模块的那个选中的图标（到现在，我们应该彻底理解了为什么 Welcome 示范程序里显示的图标与 MS Windows 95 控制面板文件夹内显示的图标有所区别）。

```
typedef struct tagNEWCPLINFO
{ // ncpli
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwHelpContext;
    LONG lData;
    HICON hIcon;
    TCHAR szName [32];
    TCHAR szInfo [64];
    TCHAR szHelpFile [128];
} NEWCPLINFO;
```

在这种处理要结束的时候，我们已经准备好了激活和显示一个属性表对象——亦即一个控制面板应用程序。这种任务是通过向 CPlApplet () 函数发送一条 CPL_DBLCLK 消息来完成的，同时注意传递相应 NEWINQUIRE 数据结构的地址以及用于标识那个对象的 ID。下面这个代码段向我们展示了怎样载入和激活 Main.CPL 模块。

```
...
case MN_LOAD:
{
    HMODULE hmod;
    FARPROC pfn;
    int nPages, iRet, i;
```

```
NEWCPLINFO ncpli [10];

// load the main.CPL module
hmod = LoadLibrary (" main.cpl");
// get the CpiApplet address
pfn = GetProcAddress (hmod, " CPlApplet");
// initialize it
iRet = (*pfn) (hwnd, CPL_INIT, 0, 0);
// send the CPL_GETCOUNT message
nPages = (*pfn) (hwnd, CPL_GETCOUNT, 0, 0);

// initialize all the dialogs
for (i = 0; i < nPages; i++)
{
    iRet = (*pfn) (hwnd, CPL_NEWINQUIRE, i, ncpli + i);
}
// start a page
for (i = 0; i < nPages; i++)
{
    iRet = (*pfn) (hwnd, CPL_DBLCLK, i, ncpli [i] .lData);
}
```

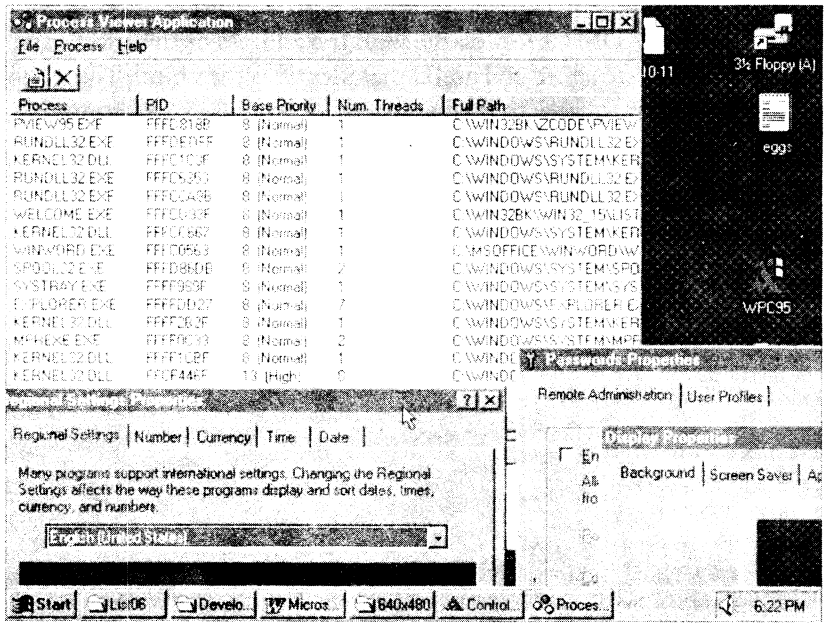


图 15-29　PVIEW95 证明尽管存在几个控制面板应用程序，然而 Control.EXE 现在并没有激活

```
            break;
    }
        break;
    ...
```

根据中断的顺序，我们需要为以前激活的每个对象发送一条CPL_STOP消息，并在控制器退出之间传递CPL_EXIT消息，从而完成整个操作。图15-29向大家展示了Welcome示范程序以及直接从应用程序里直接激活的几个属性表（控制面板应用程序）——这些属性表不是依赖Control.EXE激活的。

15.10 定制控件的建立

在这一章中，我们已经探讨了怎样利用子类处理、超类处理以及物主绘图技术对窗口的标准功能进行扩展，从而建立具有高级功能的窗口。在本书附带CD的Listing 15.7和15.8里，大家分别能找到Clrdll和Color两个示范程序。我们将同时利用这两个程序来阐述如何创建一个定制控件。

在这儿，“定制控件”(Custom Control）标识了一种新的注册窗口类，它的注册和控制是在一个DLL模块内部完成的。使用这种方案的优点在于它相当灵活，且易于使用。把所有代码都封装到DLL里以后，这种窗口类就可以由第三方开发者或者公司内部的其他组织使用。图15-30向大家展示了Color应用程序的运行情况，它带有两个子窗口，这两个子窗口都从属于Clrdll这个动态链接库里注册的定制控件类。这一类的控件对鼠标左键单击的响应就是显示一个颜色定制对话框，如图15-31所示，用户可在其中选择一种预定义颜色。

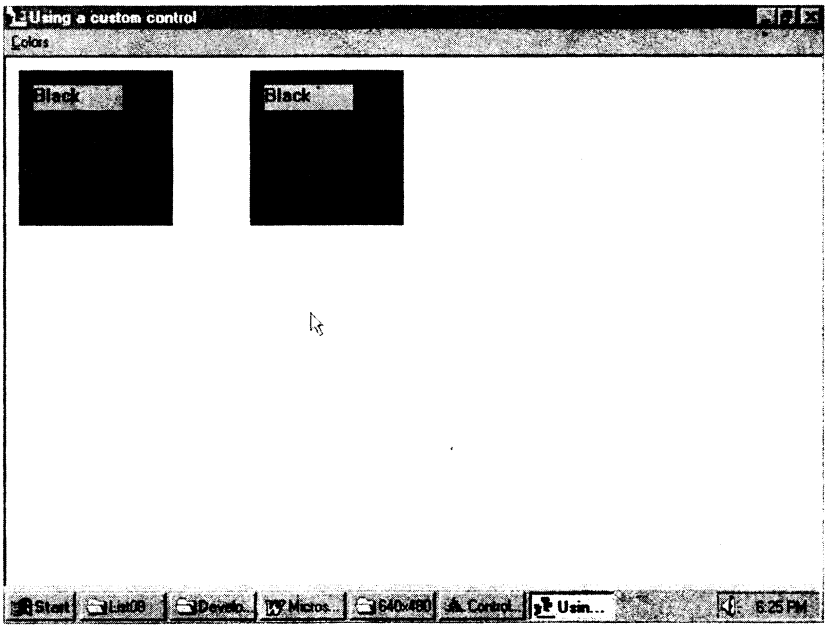


图15-30 Color示范程序提供了两个定制控件，
它们从属于Clrdll动态链接库内注册的一个窗口

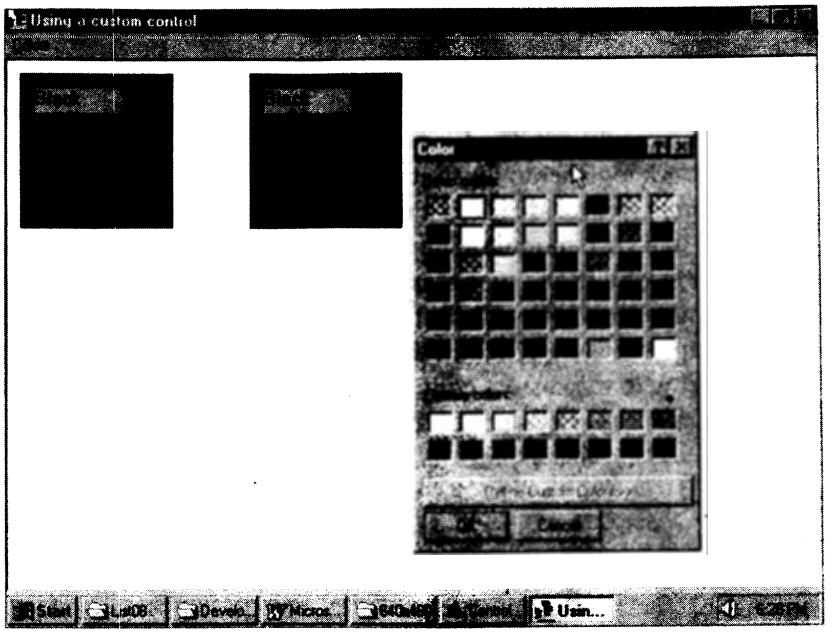


图 15-31　选择一种颜色，把它分配给 Color 示范程序里的一个子窗口

在进行了这种操作以后，另外一个对话框会显示出来，请求为选中的颜色关联一个正文标签。除了这种视觉上的效果以外，包含了定制控件的 Clrdll 还会发出一些消息，从而对一个颜色控件进行查询，以及获取一些相应的信息。设计一个新的窗口类时，假如希望把它当作定制控件来使用，请记住对支持的所有风格、标志、消息以及通知消息进行设置。除此以外，还要记住利用 CreateWindowEx () 的最后那个参数，从而在建立好一个新窗口以后立刻向它传递一个信息块。

本书附带 CD 的 Listing 15.9 内列出了一个名为 Format 的示范程序，如图 15-32 所示。该程序是向我们阐述怎样成功定制一个新窗口类的极好的范例。应用程序可以在客户区内显示出两个类似于 EDIT 控件类的窗口，它们将准备接收用户的输入。这两个控件属于在源代码内注册的一个超类。与以前的例子不同，这个超类支持格式化输入。每个窗口都会拒绝对输入的非法字符提供响应，因为它们不符合建立控件时对格式化字符串的设置。

尽管这两个窗口被指示支持不同的格式，但它们都属于同一个窗口类。位于顶部的那个窗口可以接受字符和数字的组合输入，我们把这种信息叫作：Fiscal code。Fiscal code 与美国的社会保险号码类似，只不过它在这儿需要分配给所有意大利居民，或者居留于意大利的外国人。这种 16 个字符长的正文串是根据居民的名字以及他们的生日和出生地点建立的。第二个控件窗口支持一个 ISBN（国际标准书刊编号），它唯一性地标识一本书。在我们这种情况下，它包含了几个数字，这些数字分配到不同的组别内，各个组之间则用一个破折号分隔。

这两个控件的作用是严格限制用户的输入，只接收符合当时情况的输入字符。意大利的 Fiscal code 里依顺序排列是六个字母、两个数字、一个字母、两个数字、一个字母、三个数字以及一个字母，比如 PLTSFN63T10G205Z 就是一个有效的意大利 Fiscal code。由于字符串本身比较复杂，所以很容易发生输错的情况。为了减少和限制出错的机率，第一个控件窗口首

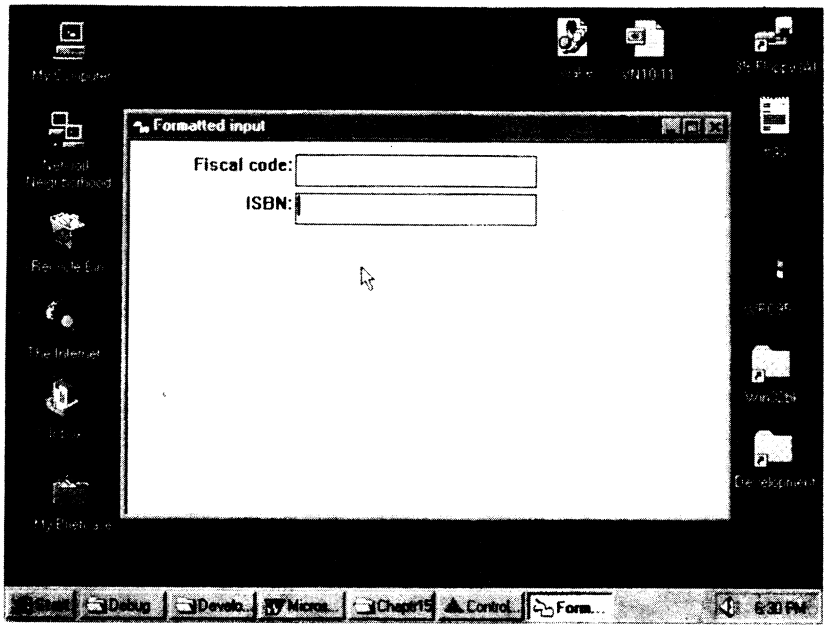


图 15-32　Format 示范程序提供了两个控件，它们只接收符合一定标准的输入正文串

先将只接收六个字母。除此以外，这些字母都必须是大写字母。除非用户键入了六个大写字母，否则就没有办法继续他（她）的输入。

第二个控件只接收一系列数字，并且把数字限制在 0 到 9 的范围以内。此外，这个控件还会自动对 ISBN 进行格式化，方法是在恰当的位置放置一个破折号符号。一个典型的 ISBN 看起来应该像这个样子：1-23456-789-0。图 15-33 向大家展示了两个完整的正文串插入输入控件后，Format 示范程序的显示情况。

为了对这种增强的功能提供支持，我们显然需要在标准编辑控件的基础上注册一个新的窗口类。超类的定义对我们来说已经不显得神秘了，GetClassInfoEx () 就是我们的正确工具。

```
...
// retrieve information fro the EDIT class
GetClassInfoEx (NULL, " EDIT", &wc);

// chage some information
pfn = wc.lpfnWndProc;
wc.lpfnWndProc = NewEditWndProc;
wc.hInstance = hInstance;
wc.lpszClassName = szNewEdit;
wc.lpszMenuName = NULL;

// increasing the class extra bytes
```

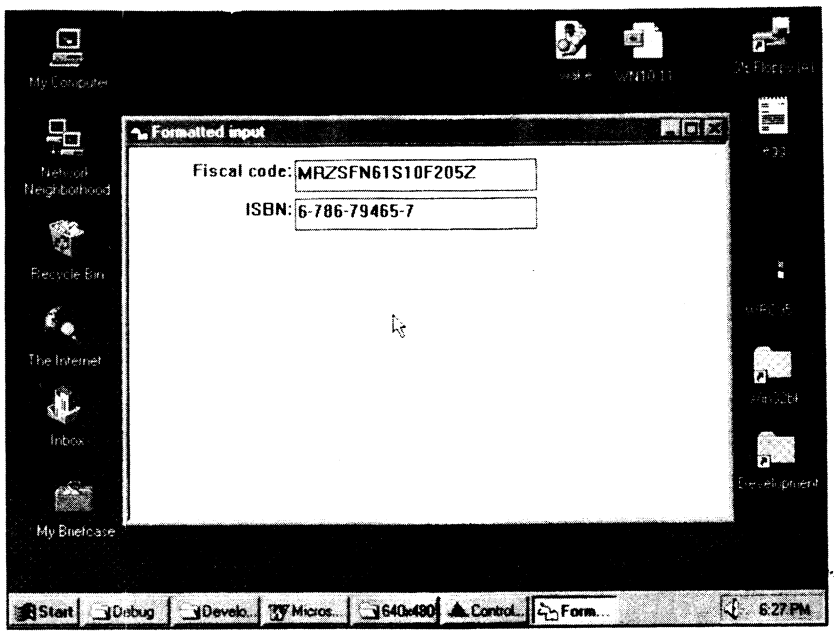


图 15-33　对应的输入正文串已经插入到两个控件里

```
wClass = wc.cbClsExtra;
wc.cbClsExtra += sizeof (LPSTR) + sizeof (int);
...
```

随后将注册新的窗口类，同时传递 Edit 窗口进程的地址，就像我们以前看到的那样。建立属于这一类的某个窗口时，Format 示范程序会利用 CreateWindowEx () 的最后一个参数，从而传递包含了格式化正文的一个正文串。应用程序已经按照下述的字符和符号对自己的规则集合进行了定义：

3 [az, AZ, 09] 1 {-1} 2 [19]

其中的逻辑是按照下面这种方案来实现的：最开头的数字指定了字符的数量，而方括号内则包含了允许用户键入的所有符号。花括号是指必须自动放置于控件内的符号，从而对正文串的格式进行自动处理。例如，1 {-1} 表示输入了三个字符以后，就应该插入一个破折号。这个格式化正文串的最后一部分指出只允许输入两个数字，选择的范围是从 1 到 9。

```
...
hwndEdit = CreateWindow (szNewEdit,
                         NULL,
                         WS_CHILD | WS_VISIBLE | WS_BORDER,
                         130, 40,
                         190, 25,
                         hwnd,
```

```
                                    (HMENU) ID_NEWEDIT,
                                    hInstance,
                                    (LPSTR)"1[09]1{-}3[09]1{-}5[09]1{-}1[09]");
    ...
```

这样就解释了格式化方案是怎样实现的。建立属于这个超类的一个新控件时，格式化字符串就会作为CreateWindowEx ()的最后一个参数进行传递。在类窗口进程里，这个串随后将拷贝到一个内存块里，这个内存块是从应用程序堆空间内自动分配的。除了把格式化字符串拷贝到这个内存区域里以外，其他杂项处理还包括把窗口里的串指针存储于GWL_USERDATA空闲内存位置处。

新的超类将利用某些消息从这个类里获取一些有用的信息，这些消息可以在类里提供某些附加的功能，如表15-11所示。

表15-11 Newedit超类使用的NM_消息

消 息	值	说 明
NM_COUNTINPUCHAR	WM_USER + 40	返回准备插入控件内的字符数，排除其中格式化符号
NM_COUNTFRMTCHAR	WM_USER + 41	返回字符串内的格式化符号数目
NM_GETFRMTSTRING	WM_USER + 42	返回格式化正文串
NM_GETSTRING	WM_USER + 43	返回插入控件窗口内的字符串
NM_GETSTRINGLENGTH	WM_USER + 44	返回字符串的长度

15.11 输入控制

Newedit超类出色的设计几乎完全体现于WM_CHAR消息块内。无论什么时候，只要这条消息到达了类窗口进程，应用程序就会取得与当前控件对应的格式化正文串。随后，这个格式化串将与用户的输入字符进行比较，判断它是合法字符还是非法字符。同时某些编辑特性仍然得到了保留，例如退格以及把光标移至输入正文开头或者末尾等。

15.12 圆形的窗口

对传统的矩形窗口感到厌倦了吗？大家有没有想过要对它的样子进行一些改变呢？一个新的Win32 API函数允许我们从代表窗口的标准矩形里切割下任何形状。

这个新的API就是SetWindowRgn ()。在本书附带CD的Listing 15.10里，大家可以找到一个名为Circle的示范程序，其中就应用了这个函数。

```
#include <winuser.h>
int SetWindowRgn (HWND hwnd, HRGN hRgn, BOOL bRedraw);
```

参数	说明
HWND hwnd	窗口句柄
HRGN hRgn	区域句柄

BOOL bRedraw	假如为 TRUE，表示在应用了区域以后对窗口进行重画
返回值	在正文里讨论
int	假如函数执行，就是一个非零值

第一个参数是窗口句柄，接下来是指定那个窗口内某个区域的一个句柄。在这儿，一个“区域”可以是以前通过调用 API 函数 CreatexxxRgn () 而建立起来的任何一种几何形状。这个程序的意图是在一个窗口内显示一幅整圆位图，同时不使用任何基本的窗口组件，比如标题栏以及系统菜单等等。对于顶级窗口来说，可以不建立标题栏的唯一类型就是弹出式窗口。建立好窗口以后，当 WM_CREATE 消息抵达窗口进程时，应用程序就会从资源文件里载入圆形位图，然后通过 GetObject () 计算出它的大小。为了限制它在屏幕上的尺寸，我们把宽度和高度都减小到初始值的四分之一，然后根据这个尺寸建立一个圆形区域。在这个时候，我们就需要用到 SetWindowRgn ()，这个函数可以把圆形区域从弹出式窗口的客户区内切割下来。

```
...
case WM_CREATE:
{
    HRGN hrgn;
    BITMAP bmp;

    // application instance handle
    hInstance = ( (LPCREATESTRUCT) lParam) -> hInstance;

    // loading the bitmap from the resource file
    hbm = LoadBitmap (hInstance, " CROW");
    // get the object size
    GetObject (hbm, sizeof (bmp), &bmp);
    // reduce it to a fourth
    cx = bmp.bmWidth / 4;
    cy = bmp.bmHeight / 4;

    // create elliptic region
    hrgn = CreateEllipticRgn (0, 0, cx, cy);
    // impose the circular window size
    SetWindowRgn (hwnd, hrgn, TRUE);
    // show the window
    SetWindowPos (hwnd, HWND_TOP, 0, 0, cx, cy, SWP_NOMOVE);

    // loading the popup menu
```

```
        hpopup = LoadMenu (hInstance, " popup");
        hpopup = GetSubMenu (hpopup, 0);
    }
        break;
    ...
```

从这个时候开始，窗口在屏幕上就以一个圆形来代表了——其中没有任何一个标准的非客户区组件。和往常一样，位图是通过对 WM_PAINT 消息的处理在应用程序客户区内显示出来的。

应用程序标题栏的缺少限制了我们在屏幕上对窗口的移动。为了避免这种限制，我们只需使用下面这个代码段：

```
    ...
    case WM_LBUTTONDOWN:
    {
        SendMessage (hwnd, WM_NCLBUTTONDOWN, (WPARAM)
            HTCAPTION, MAKELPARAM (5, 5));
    }
    break;
    ...
```

用户单击鼠标左键的时候，这种操作就会被认为是标题栏上发生的相同事件。通过这种对窗口的欺骗，用户就可以在屏幕上拖动圆形窗口——就像一个标准的窗口那样。该程序的最后一项特色是用户可以在位图上方单击鼠标右键，从而实现窗口的关闭。这时，一个小的弹出式菜单会显示出来，其中列出了 Exit 选项。

第 16 章　Win95 外壳的开发

打开我们的 PC 机，引导过程结束以后，最终显示出来的就是系统外壳。系统外壳其实是一个名为 EXPLORER. EXE 的程序，这个 Win32 进程将自动由系统激活。在 SYSTEM. INI 的 Boot（引导）部分里，我们可以找到相应的 shell 条目，它的作用就是指示引导阶段结束以后载入哪个应用程序当作系统外壳使用，如下所示：

```
[Boot]
...
shell=EXPLORER. EXE
...
```

我们可以花些时间对注册表数据库进行浏览，期望在其中能寻找一些类似的信息。但是很令人失望，注册表数据库里没有找到与系统外壳初始化有关的任何信息。我发现假如缺少 SYSTEM. INI 文件，或者 shell 条目指定的不是一个有效的 Win32 进程，Windows 95 系统根本不能启动。正如大家现在已经知道的那样，系统外壳的外壳就像图 16-1 那样。

在缺省情况下，屏幕的最底部占据了任务栏（Taskbar），这是一种矩形的定制窗口，可以沿着屏幕四个边界中的任何一个摆放。屏幕的剩余部分就是桌面，几乎所有的行动都是在这个活动区域里发生的。EXPLORER. EXE 是一个多线程进程，最初激活了三个线程，以后随着新任务的执行，会有越来越多的线程涌现。

16. 1　检查任务栏

通过仔细检查任务栏，我们发现这个对象包含了三种具有明显区别的组件。位于最右侧的是通知区域，它的主要任务是容纳几个小图标，用它们来代表系统内的活动对象。中间的区域包含了任务栏切换按钮，这个区域将动态地由几个矩形按钮占据，每个按钮都代表了一个活动进程或者顶级窗口。在最左侧是 Start 按钮，它是整体系统外壳的核心部件。事实上，根据微软公司“可用性实验室”的测试报告，在所有典型的用户操作里，几乎 95%都是由访问 Start 按钮和相关的 Start 菜单完成的，如图 16-2 所示。

从窗口的角度来看，任务栏是一系列窗口的集合。任务栏的主窗口是属于 Shell_TrayWnd 的一个弹出式窗口（WS_POPUP）。这个窗口拥有三个子窗口，其中两个又分别提供了一个子窗口。因此，总而言之，任务栏里合成了六个窗口。对这些窗口的总结请大家参考表 16-1（句柄值显然要根据系统的不同而发生变化）。

任务栏主窗口占据了任务栏的整个矩形区域，而通知区域则对应于右侧的可见矩形。数字时钟是属于 TrayClockWClass 类的一个窗口，它部分覆盖了通知区域。Start 按钮用不着我们多说。MSTaskSwWClass 和 SysTabControl32 窗口定义了任务栏的任务切换区域。这两个窗口占据了相同的屏幕位置，同时 SysTabControl32 完全覆盖了它的父窗口。

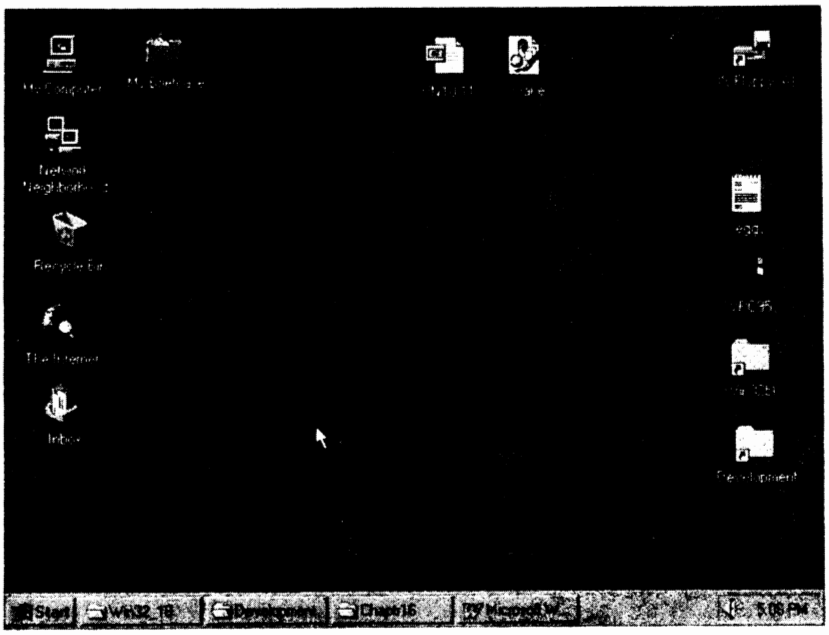


图 16-1　系统外壳 EXPLORER. EXE 由两个组件组成：任务栏和桌面

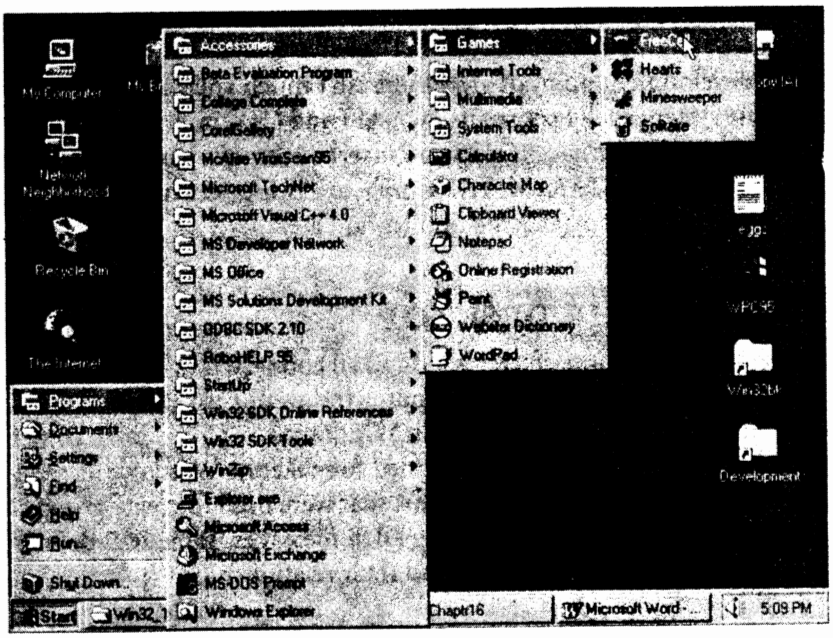


图 16-2　Start 菜单以及另外一些次级菜单

从程序开发的眼光来看，我们对任务栏几乎没有什么需要做的。尽管系统外壳提供了它自己的 API 和 C++接口，但是我们基本上只有 Shell_NotifyIcon () 函数可供利用——它能在任务栏的通知区域内插入或者删除小图标。在本书附带 CD 的 Listing 16. 3 里，大家能找到

一个名为 STARTMENU 的示范程序，它向我们展示了怎样与这种通知区域进行交互作用。

表 16-1 组成任务栏的窗口

窗口	父窗口	类	风格	ID
0x00000100	HWND_DESKTOP	Shell_TrayWnd	WS_POPUP	无
0x00000104	0x00000100	BUTTON	WS_CHILD	130
0x00000108	0x00000100	TrayNotifyWnd	WS_CHILD	12F
0x0000010C	0x00000108	TrayClockWClass	WS_CHILD	12F
0x00000118	0x00000100	MSTaskSwWClass	WS_CHILD	0
0x0000010C	0x00000118	SysTabControl32	WS_CHILD	1

16.2 桌面的深入探索

桌面是一系列四个窗口的集合，这些窗口都连接到一个分级结构里。在这个树形结构的顶部是一个弹出式窗口，它属于 PROGMAN 类，这个类与 Windows 3.x 里的程序管理器是相同的。这并不是一种巧合，相反，这是微软工程师们有意设计后的产物。它在这儿的目的是让 Windows 95 外壳与老式的 Win16 程序兼容，因为某些 Win16 程序需要通过 DDE 消息在程序管理器里创建程序组和图标。表 16-2 为我们总结了构成桌面的各种窗口，以及它们之间的关系。

表 16-2 组成系统桌面的窗口

窗口	父窗口	类	风格	ID
0x000000EC	HWND_DESKTOP	PROGRAM	WS_POPUP	0
0x000000F0	0x000000EC	SHELLDLL_DefView	WS_CHILD	0
0x000000F4	0x000000F0	SysListView32	WS_CHILD	1
0x000000F8	0x000000F4	Sysheader32	WS_CHILD	0

从属于任务栏以及桌面的所有这些窗口都是在 EXPLORER.EXE 的同一个线程里创建的。这种选择能给我们带来一些有益的提示：我们以前的知识都是基于如何在几个独立线程的窗口之间实现多线程方案，现在这种情况无疑是对以前知识的一种补充。系统外壳内的所有窗口都从属于相同的线程，这是为了避免在这些窗口间交换信息时产生性能的退化。

用户理解为桌面的那个东西其实就是一个列表视窗控件，当前这个控件正处于图标视窗模式下面——该对象只支持这种视窗模式。这种列表视窗窗口的作用很简单，它可以用属于不同类型的图标把各种对象可视化地表示出来。这个窗口在系统外壳 OLE 领域内的一种表现就是 SHELLDLL_DefView 类。这一类拥有系统外壳面向对象的全部逻辑，而列表视窗窗口只是用于表达数据的一种方式而已。相同的组合——一个列表视窗加上属于 SHELLDLL_DefView 类的一个窗口——也是通用对话框的基础。假如我们用 Spy++程序仔细地检查一个通用对话框的中心部分，就可以发现构成外壳桌面的相同两个窗口。这个窗口类是在 SHELL32.DLL 模块里注册的，然而具体的细节在所有文档资料里都没有提到。就我个人来看，我认为这不能不说是一种遗憾，因为通过它也许可以立刻让一个窗口拥有 OLE 功能，这样就能减少我们实现这种特性所花的时间。

桌面显示的几个图标分别指定了不同类型的对象。在图 16-1 里，大家可以注意到右上角用于 A 驱动器的一个快捷，诸如 My Computer，Network Neighborhood 以及 Recycle Bin 的某些系统对象沿着屏幕左侧摆放，另外还有针对经常用到的文件夹的一些快捷。这些对象的一部分提供了一个物理性备份——存储于\WINDOWS\DESKTOP 文件夹内的一个文件，如图 16-3 所示。

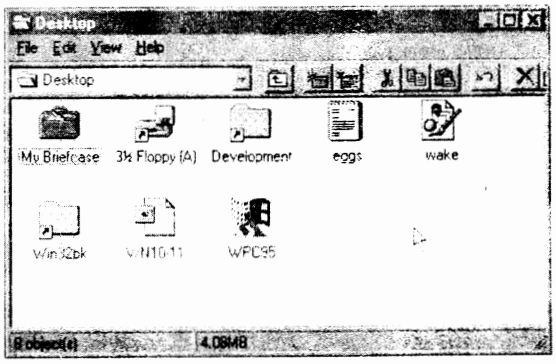


图 16-3 \WINDOWS\DESKTOP 文件夹的内容

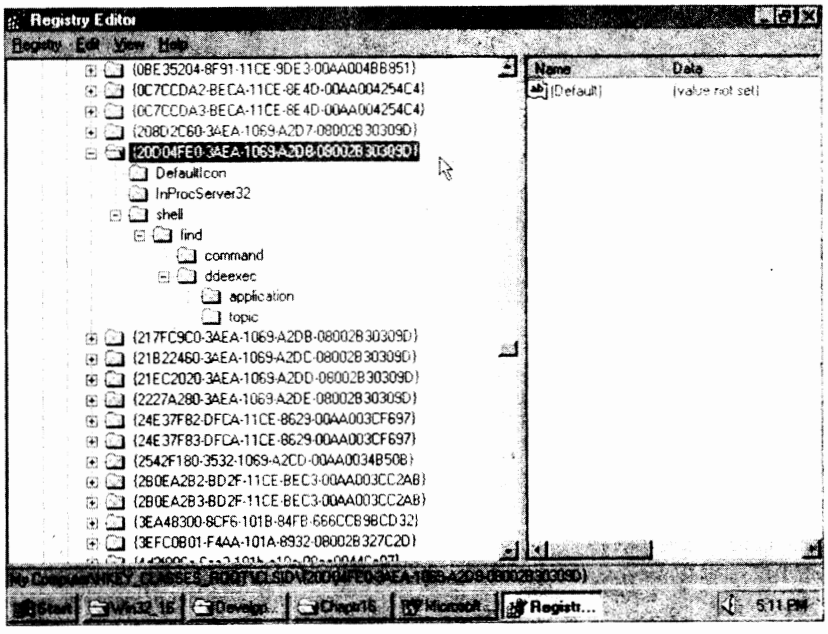


图 16-4 REGEDIT 显示了与 My Computer 这个 OLE 对象有关的所有信息

通过比较图 16-1 和图 16-3，我们会注意到后者缺少了一些对象。DESKTOP 文件夹内列出了所有快捷，因为这些快捷是用扩展名 .LNK 标识的一种物理性文件。而 My Computer，Network Neighborhood，Recycle Bin，Inbox 以及其他对象都不在这个文件夹中。这是怎么回事呢？原因在于某些外壳组件实际上是 OLE 对象，甚至可以说它们是注册表数据库内列出的

OLE 类的复制品。图 16-4向我们展示了注册表数据库的一部分，其中指定 My Computer 的类已经注册好了。

RecycleBin从属于另外一个OLE类，即：{645FF040-5081-101B-9F08-00AA002F954E}。这种数字是宽度为 128 位的一个 ID，利用它可以对每个 Windows 95 系统内的这种 OLE 类进行标识。对于这种 ID 来说，更为常见的一种称呼是 CLSID。在注册表数据库里，HKEY_CLASSES_ROOT 下面带有相同名字的子键内列出了迄今为止在系统里注册的所有类。这个列表显然是可以扩展的，每个新的 Windows 95 兼容产品都应该根据不同的需求注册一个或者多个新类。

随同开发环境提供了一个名为 UUIDGEN.EXE 的工具软件，利用它可以生成一个新的 ID，用这个 ID 可以唯一性标识多计算机系统范围内的一个 OLE 类。当然，在多台计算机范围内有效的前提是计算机已经配备了一块网卡；否则，它生成的数字只有在当前系统内才是唯一的。假如执行这样一条命令：UUIDGEN /?，就可以看到如下所示的帮助信息：

```
usage: uuidgen [-isonvh?]
i — Output UUID in an IDL interface template
s — Output UUID as an initialized C struct
o〈filename〉 — redirect output to a file, specified immediately after o
n〈number〉 — Number of UUIDs to generate, specified immediately after n
v — display version information about uuidgen
h,? — Display comand option summary
```

下面是在我的 PC 上生成的两个 CLSID：

```
9D0FDE40-2755-11CF-ABC2-00608C115A4F
BC43D820-2755-11CF-ABC2-00608C115A4F
```

这些数字组合对于新 OLE 类的注册是很有用的，这种注册过程需要涉及到几个中间步骤才能完成，对这些步骤的详细讨论将在本章的后面部分进行。当我们返回 Recycle Bin 以后，可以把它的 CLSID 拷贝到剪贴板内。最简单的方法就是选定它，然后选择 Edit 菜单内的 Rename 菜单项。进入编辑模式以后，我们只需简单地按下 Ctrl+Ins（或者 Ctrl+C）就可以把它拷贝到剪贴板内，如图 16-5所示。

在这个时候，我们可以在桌面或者其他任何现在文件夹内建立一个新的文件夹，然后对它进行更名处理。实际上，我们必须做的仅仅是把对象的 CLSID 简单地添加到文件夹名字的后面，就好象这种 CLSID 是它最后的扩展名一样。例如，假设我们现在建立了一个名为 Garbage 的文件夹。这时，它的新的完整名字就会变成：Garbage.{645FF040-5081-101B-9F08-00AA002F954E}，注意花括号对是不可缺少的。确认了这种编辑结果以后，对象就会自动改变标准的文件夹图标，并用 Recycle Bin 图标取代，如图 16-6 所示。

经过这个例子以后，我相信对 CLSID 子键下面的注册表数据库进行探索是很必要的，这种方式有助于我们更好地理解系统外壳是如何实现的。

16.2.1 关于复活节彩蛋

由于可以建立这种特殊的对象，使其属于某种奇妙的类，所以 Windows 95 的开发者们获

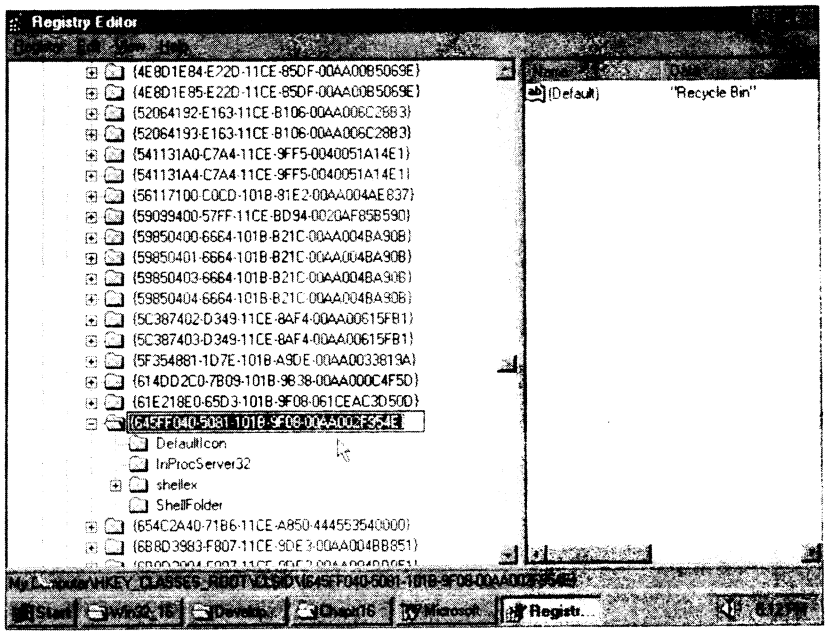


图 16-5　把一个 CLSID 直接拷贝到剪贴板内，这样就可以避免出现输入错误

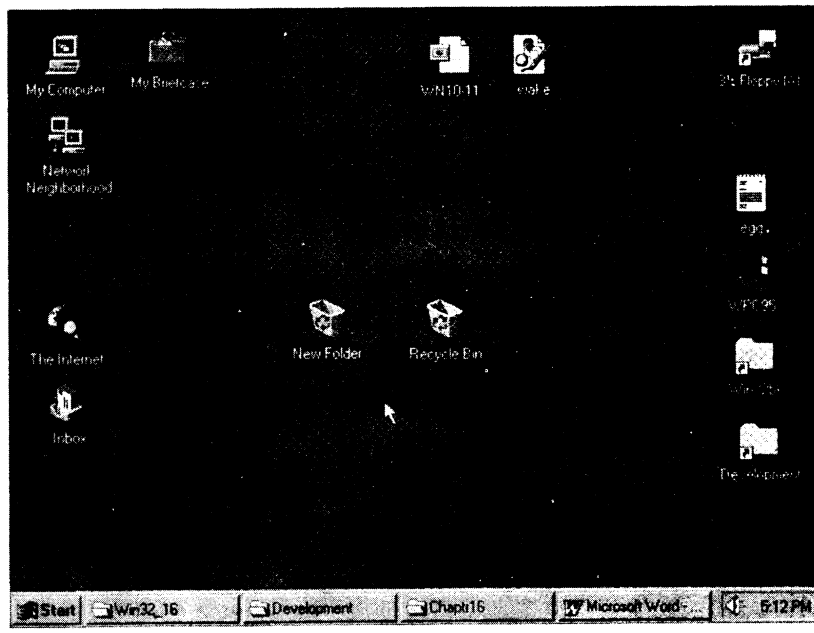


图 16-6　系统现在提供了两个类型为 Recycle Bin 的对象。
注意它们中的任何一个都不是快捷

得了灵感，他们建立了著名的“复活节彩蛋”(Easter Eggs)。这个术语是指一种特殊的应用程序，它们在系统内部运行，可以显示出涉及一个软件项目开发的所有人士。为了访问 MS

Windows 95 的复活节彩蛋，我们必须在桌面建立一个文件夹，然后接连三次改变它的名字，为它们分配下面列出的这三个字符串（请保证正确地输入了大小写形式）：

and now, the moment you've all been waiting for

we proudly present for your viewing pleasure

The Microsoft Windows 95 Product Team!

经过这样的操作以后，我们就有幸看到图 16-7 显示的窗口。这个窗口的背景是起伏的水面，上面显示出所有我们或多或少熟悉的一些人的名字，同时伴随着震撼人心的摇滚乐。

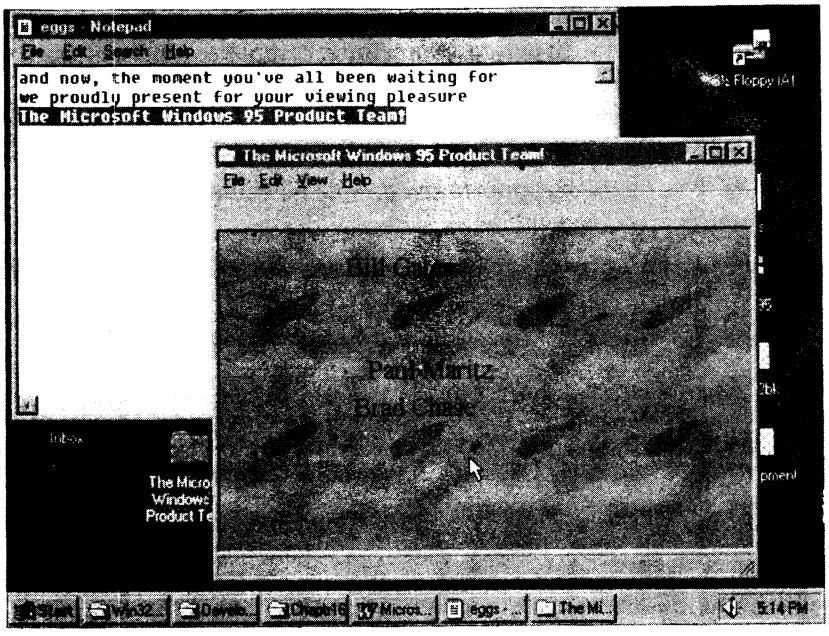


图 16-7　MS Windows 95 的复活节彩蛋

16.2.2　外壳命名空间

由于外壳桌面里需要显示不同类型的对象，这样就为 Windows 95 的开发者们带来了一些新的问题和挑战。把应用程序源代码与外壳集成到一起看来是必由之路。事实上，为了把微软公司的徽标散布到世界各地，这种方案几乎是不可或缺的。

由系统外壳管理的对象不再是 Windows 3.x 里的文件，而是文件与 OLE 对象的组合。微软公司之所以首次引入这种具有革命性的概念，是为了让计算机更易于使用，即使那些非专业人士也都能操纵自如。正是由于存在外壳对象的不同属性，所以就导致了“外壳命名空间”这个概念的诞生。

在 Windows 95 的术语表里，外壳命名空间的英文原文是“Shell Name Space”，它表示一系列特殊对象的集合，这些对象都将用于对整个系统进行管理。这些对象是用一种分级结构组织起来的，这种结构可以包含属于不同类别的对象。文件系统目录和文件构成了 Windows 95 命名空间对象的主体，另外还包括诸如虚拟文件夹、打印机以及共享资源的一些对象。

命名空间分级结构的根部就是桌面。对于这种概念，我们并不是第一次碰到了，以前学习 Win32 API 以及 Windows 95 组件的时候就曾经数次遇到过这样的问题。在这种情况下，桌

面是指外壳桌面内显示的一系列对象的集合，它们与图 16-3 内显示的对象是完全不同的。正如我们通过图 16-8 看到的那样，命名空间桌面是对外壳桌面的一种正确的表达，尽管它是限制在一个窗口里的。我们甚至可以尝试用详细列表模式来显示这种信息，看看会发生什么情况。如图 16-9 所示，我们注意到没有物理性备份的所有对象都被标识成 System Folder（系统文件夹）。

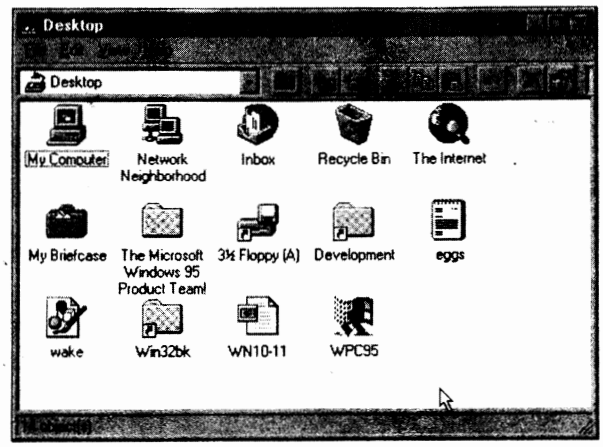


图 16-8　命名空间的根部是桌面，它由外壳桌面显示的一系列对象组成

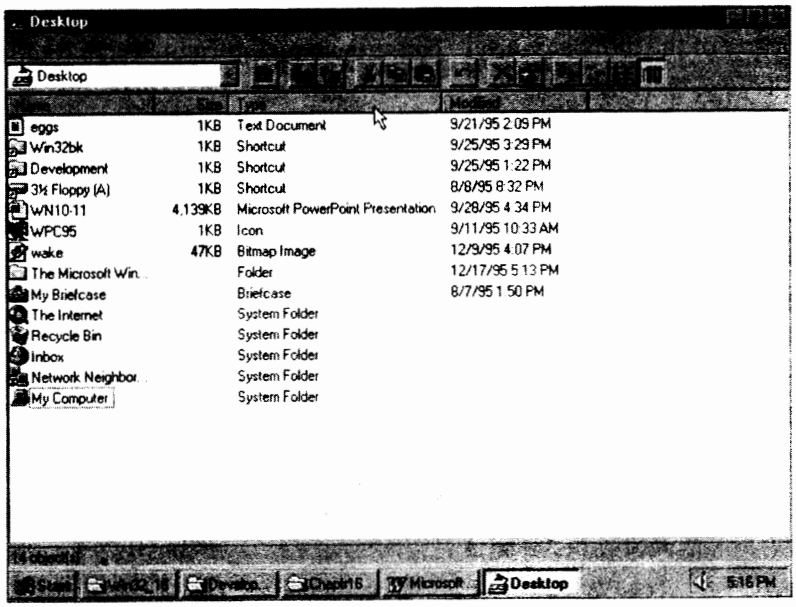


图 16-9　以详细列表视窗显示的命名空间根部

作为一名开发者，我们必须对系统信息的这种新的表达和管理方式引起足够的重视。新

的应用程序应该考虑到命名空间对自己带来的影响。为了对外壳命名空间进行探索，我们首先可以编写一个简单的程序，它将利用 SHBrowseForFolder () 这个 API 函数在应用程序源代码内对命令空间进行表示。

1. 浏览文件夹

Find 引擎以及一个标准的外壳文件夹之间有什么联系呢？它们都是系统外壳的两个组件，并且它们都依赖于外壳命名空间来实现自己的任务。查找一个对象的时候，用户可以选择一个命名空间分支，从而实现对任务的限制。为了达到这种目的，用户必须选择主窗口内的 Browse 按钮。这样会马上显示出来一个对话框，其中用一个列表视窗的形式列出了整个命名空间的样子。如图 16-10 所示。

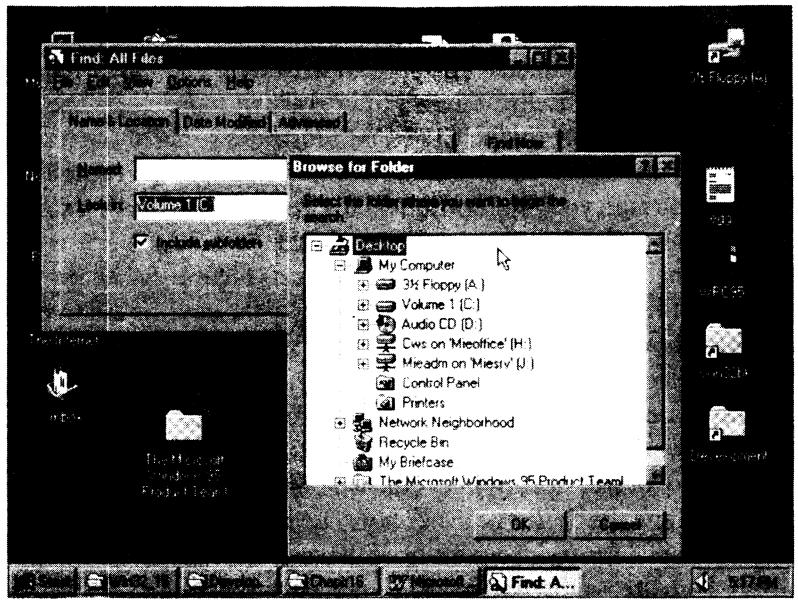


图 16-10　Find 主页内的 Browse 按钮引入了一个对话框，
其中列出了整个命名空间

一个标准的文件夹也要以外壳命名空间为基础。工具栏左侧的组合框是在命名空间内漫游的一种简便和有效的方式，如图 16-11 所示。

无论在 Browse for folder 对话框里显示的信息，还是在文件夹组合框内显示的信息，这些信息对于应用程序来说也是可以访问的。我们只需要依靠一些外壳 API 函数就可以达到这种目的，这些 API 是专门针对与外壳的交互作用而设计的。本书附带 CD 的 Listing 16.1 里提供了一个名为 SHELLOBJ 的示范程序，它向我们阐述了如何在自己的应用程序里仿真某些外壳组件，以及实现几种相关的命名空间特性。这个程序看起来就像一个标准的 Win32 窗口。假如选择 Actions 下拉式菜单，我们就可以对不同的外壳交互操作进行试验，如图 16-12 所示。

假如选择 Browse for folder 菜单项，应用程序就会显示出如图 16-13 所示的一个对话框。整个命名空间都在其中表达出来了，用户可在其中选择一个分支。

与 Find 显示的对话框不同，在这种情况下，标题栏下方显示了两个正文串，这就证明了

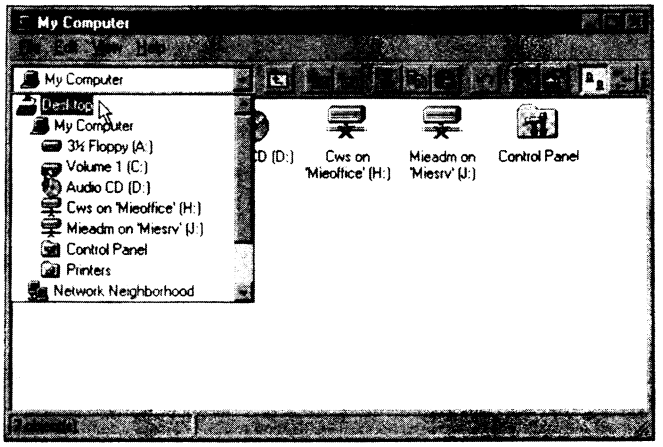


图 16-11　为了显示出与命名空间组合框关联在一起的列表框，
我们只需要在显示出来的项目上面单击就可以了，
从而节省了尝试单击下箭头按钮所需的时间

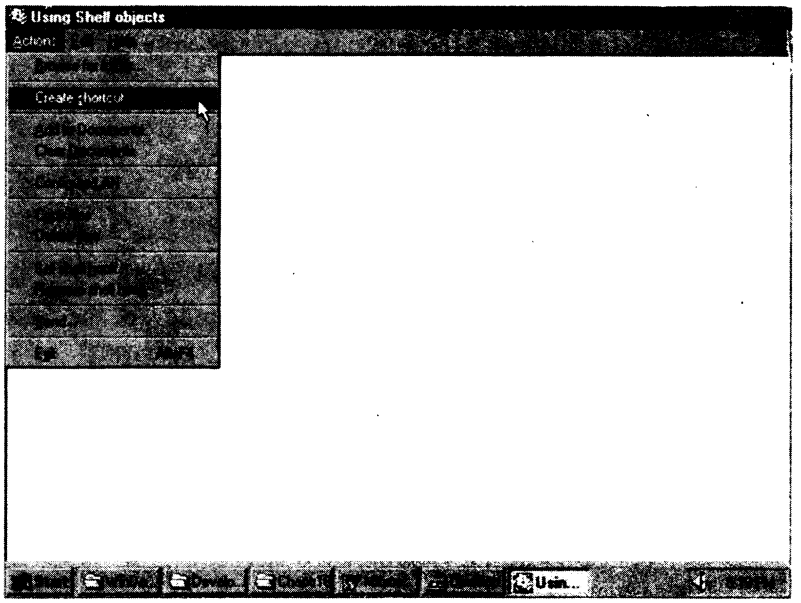


图 16-12　SHELLOBJ 示范程序向我们展示了用于控制
和实现外壳特性的几种方法

这个系统组件现在正处于某个定制应用程序的控制之下。Browse for folder 对话框是通过调用 SHBrowseForFolder（）这个 API 函数生成的（表 16-10 内包含了对外壳函数的一份完整总结，另外还有简短的说明）。在真正调用这个函数之前，我们需要先完成两件准备工作。首先，我们需要对一个 ITEMIDLIST 数据结构进行初始化，其中将填写与命名空间节点有关的

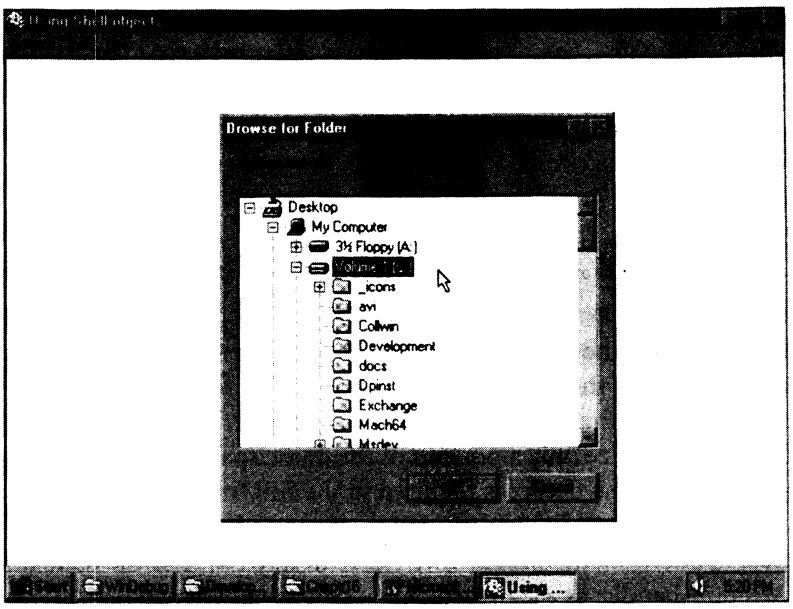


图 16-13　Browse for folder 对话框是由 SHELLOBJ 程序控制和管理的

所有信息，这些节点将当作列表视窗的根部使用。另外，在调用 SHBrowseForFolder（）之前，我们还要在一个 BROWSEINFO 数据结构里填写某些适当的数据。

```
#include〈shlobj.h〉
HRESULT WINAPI SHGetSpecialFolderLocation(HWND hwndOwner,
                                          int nFolder,
                                          LPITEMIDLIST * ppidl);
```

参数	说明
HWND hwndOwner	浏览对话框的物主窗口的句柄
int nFolder	特殊的文件夹 ID
LPITEMIDLIST * ppidl	一个 ITEMIDLIST 数据结构的地址
返回值	在正文里讨论
HRESULT	假如函数调用成功，就返回一个 NOERROR 值；否则返回一个 OLE 错误代码

窗口句柄指定了最终要在其中使用这个 API 函数，从而显示一条警告消息的那个应用程序窗口。特殊的文件夹 ID 是表 16-2 内列出的某种定义，它的作用是对一个命名空间组件进行标识。

第三个也是最后一个参数是一个 ITEMIDLIST 数据结构的地址，这个结构内将填写与那个文件夹有关的信息。下面这个代码段向我们展示了怎样获得与桌面——命名空间根部有关的信息：

```
...
// get the pidl for the desktop to initialize the folder browser
SHGetSpecialFolderLocation(NULL, CSIDL_DESKTOP, &pidlDesktop);
...
```

表 16-3 为大家列出了用于标识特殊文件夹的所有 ID。桌面是整个命名空间的根，它代表了所有文件夹起始的地方。作为另外一种选项，我们也可以挑选列于下面的另外某个特殊文件夹 ID，从而只显示出整个命名空间的一部分。

在 BROWSEINFO 数据结构里需要用到 ITEMIDLIST 结构，它的作用是指示 API 函数 SHBrowseForFolder () 建立和显示命名空间的树形结构。除此以外，BROWSEINFO 数据结构还要求用于对话框物主窗口的句柄，这和一个标准的对话框是没有区别的 (hwndOwner)。

表 16-3 特殊文件夹 ID 位置

特殊文件夹 ID	值	说 明
CSIDL_DESKTOP	0x0000	“桌面”(Desktop) 文件夹
CSIDL_PROGRAMS	0x0002	与 Programs Files 目录对应的文件夹
CSIDL_CONTROLS	0x0003	“控制面板”(Control Panel) 文件夹
CSIDL_PRINTERS	0x0004	打印机 (Printers) 文件夹
CSIDL_PERSONAL	0x0005	指定\My Documents 文件夹
CSIDL_FAVORITES	0x0006	指定\Windows\Favorites 文件夹
CSIDL_STARTUP	0x0007	指定“启动”(Startup) 文件夹
CSIDL_RECENT	0x0008	“最近常用”(Recent documents) 文件夹
CSIDL_SENDTO	0x0009	Send To 文件夹
CSIDL_BITBUCKET	0x000a	“回收站”(Recycle Bin) 文件夹
CSIDL_STARTMENU	0x000b	“开始菜单”(Start menu) 文件夹
CSIDL_DESKTOPDIRECTORY	0x0010	指定\WINDOWS\DESKTOP 目录
CSIDL_DRIVES	0x0011	虚拟文件夹，其中包含了本地计算机内的所有信息：存储设备、打印机以及控制面板。文件夹也许还包含了映射过来的网络驱动器
CSIDL_NETWORK	0x0012	用于表示网络分级结构的顶部级别的虚拟文件夹
CSIDL_NETHOOD	0x0013	文件系统目录，其中包含了显示于“网上邻居”内的对象
CSIDL_FONTS	0x0014	“字体”(Fonts) 文件夹
CSIDL_TEMPLATES	0x0015	“模板”(Templates) 文件夹

一旦撤消了对话框的显示以后，对话框内已选中的项目标签就会填写到 pszDisplayName 缓冲区内。而 lpszTitle 则代表了一个正文串，它将在对话框标题栏下方显示出来。通常，这种正文应该为用户提供恰当的线索，帮助他们作出选择。它可以是任何类型的正文，甚至可以超出一个单一的正文行的限制。

表 16-4 为我们列出了所有 BIF_标志，这些标志可用于对 Browse for folder 对话框的整体行为进行控制。为了显示出整个命名地址空间的树形结构，我们只需要为 ulFlags 项分配一个零值就可以了。在各种 BIF_标志实现的不同的效果里，BIF_SETSTATUSTEXT 可以在对话框的注释正文下面以及列表视窗控件的上面增加一个新 的输出区域。如果想在这外区域内插入正文，我们应该使用表 16-5 内列出的 BFFM_SETSTATUSEXT 消息。

```
typedef struct _browseinfo
{
    HWND hwndOwner;
    LPCITEMIDLIST pidlRoot;
    LPSTR pszDisplayName;
    LPCSTR lpszTitle;
    UINT ulFlags;
    BFFCALLBACK lpfn;
    LPARAM lParam;
    int iImage;
} BROWSEINFO, *PBROWSEINFO, *LPBROWSEINO;
```

表 16-4 BROWSEINFO 数据结构使用的标志

BROWSEINFO 标志	值	说 明
BIF_RETURNONLYFSDIRS	0x0001	用于查找一个文件夹，从而启动文档搜索
BIF_DONTGOBELOWDOMAIN	0x0002	用于启动 Find Computer
BIF_STATUSTEXT	0x0004	在对话框内包含一个状态区域。回调函数可以通过向对话框发送消息从而实现对状态正文的设置
BIF_RETURNFSANCESTORS	0x0008	返回文件系统级别内的前一个组件
BIF_BROWSEFORCOMPUTER	0x1000	浏览计算机
BIF_BROWSEFORPRINTER	0x2000	浏览打印机

在 BROWSEINFO 数据结构的 lParam 项内，开发者可以存储一些由应用程序定义的数据。Browse for folder 对话框要求用到由应用程序提供的一个回调函数，调用这个函数可以对选中项目有关的信息进行传递。这个函数的形式如下所示：

```
int BrowseCallbackProc (HWND hwnd,
                        UINT msg,
                        LPARAM lParam,
                        LPARAM lpData);
```

参数	说明
HWND hwnd	Browse 对话框的句柄
UINT msg	表 16-4 内列出的一条消息
LPARAM lParam	与消息有关的特定信息
LPARAM lpData	由 BROWSEINFO 数据结构的 lParam 项提供的一些信息
返回值	在正文里讨论
int	通常为零值

在表 16-5 内列出的五条消息中，只有前两条需要实际发送给回调函数，从而反映出由用

户采取的一些物理性行动。BFFM_INITIALIZED 使我们有机会在对话框真正显示出来之前完成一些准备工作，而 BFFM_SELCHANGED 则用于指出用户已经选择了一个不同的项目。

表 16-5 Browse 对话框使用的消息

BROWSE 对话框消息	值	说 明
BFFM_INITIALIZED	1	Browse 对话框已经初始化好了
BFFM_SELCHANGED	2	已经选择了一个新项目
BFFM_SETSTATUSTEXT	(WM_USER+100)	设置对话框状态区内的正文
BFFM_ENABLEOK	(WM_USER+101)	激活或者屏蔽 OK 按钮
BFFM_SETSELECTION	(WM_USER+102)	设置当前的选定内容、

最后，iImage 项内包含了系统图象列表内的图标索引编号，这些图标与选中的项目是对应的。因此，应用程序不仅要从控件内接收项目正文，还要访问和显示相关的图标。怎样对系统图象列表进行访问将在本章的后面部分进行详细的讲述。

下面这个代码段是从 SHELLOBJ 示范程序里摘录下来的。用适当的信息填写了 BROWSEINFO 数据结构以后，应用程序会调用 API 函数 SHBrowseForFolder ()，从而最终建立和显示对话框：

```
...
// fill the BROWSEINFO data structure
bi.hwndOwner = hwnd;
bi.pidlRoot = pidlDesktop;
bi.pszDisplayName = szText;
bi.lpszTitle = pszTitle;
bi.ulFlags = 0;
bi.lpfn = BrowseProc;
bi.lParam = 0;
bi.iImage = 0;

// start browsing
pidl = SHBrowseForFolder(&bi);
...
```

图 16-14 向我们显示了 Browse for folder 对话框。定制标签有力地证明了这个对话框是由一个定制应用程序建立的。

最初在没有任何 BIF_标志的前提下建立对话框的时候，OK 按钮将处于活动状态。它的状态可以通过在回调函数里发出 BFFM_ENABLEOK 消息来进行改变。假如想激活它，只需在那条消息的 wParam 里设置 TRUE；屏蔽它则可以设置 FALSE。同时，lParam 总是设置成 0 值：

```
SendMessage (hwnd, BFFM_ENABLEOK, TRUE lParam);
```

我们可以通过 BFFM_SETSTATUSTEXT 消息在对话框的状态区内插入正文，这条消

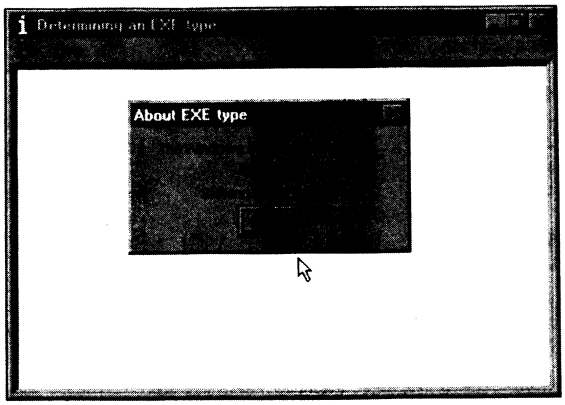


图 16-14　在 SHELLOBJ 示范程序里的 Browse for folder 对话框

息也是从回调函数里发出的。相应的正文串则通过那条消息的 lParam 进行传递，如下所示：

SendMessage （hwnd， BFFM_ SETSTATUSTEXT， 0， （LPARAM）" TORONTO"）；

为了强制性地选择一个特定的项目，我们可以发出 BFFM_SETSELECTION 消息。同时，假如这条消息的 wParam 设置成 TRUE，就表示 lParam 里包含了一个路径名；假如设置成 FALSE，就表示 lParam 是指向一个 ID 列表的指针。

SendMessage （hwnd，BFFM_ SETSELECTION，TRUE，（lPARAM）" C：\\ WINDOWS\\SYSTEM"）；

在前面的那个例子里，我们选中的是 System 目录。当用户按下 OK 按钮撤消对话框的显示时，BROWSEINFO 数据结构的 pszDiaplayName 项内包含了与选中项目对应的正文串。iImage 的作用是标识图标的索引，这些图标与系统图象列表内的项目是关联起来的。只要在知道了系统图象列表的信息的情况下，这种索引才是有用的。为了获取这种信息，我们需要调用另外一个外壳 API，即 SHGetFileInfo （），如下所示：

```
include 〈shlobj. h〉
DWORD WINAPI SHGetFileInfo (LPCSTR pszPath,
                            DWORD dwFileAtributes,
                            SHFILEINFO * psfi,
                            UINT cbFileInfo,
                            UINT uFlags);
```

参数	说明
LPCSTR pszPath	对象路径命名或者指针
DWORD dwFileAtributes	文件属性
SHFILEINFO * psfi	一个 SHFILEINFO 数据结构的地址
UINT cbFileInfo	SHFILEINFO 数据结构的长度

UINT uFlags	一个或者多个 SHGFI_标志
返回值	在正文里讨论
DWORD	返回值的含义将随着调用时使用的 SHGFI_的标志发生改变。假如使用了 SHGFI_EXETYPE 标志，返回值就是执行文件的类型。SHGFI_ ICON 和 SHGFI_SYSINCONINDEX 是指示函数返回系统图象列表的句柄，这个图象列表里包含了大图标图象（SHGFI_SMALLICON 则返回小图标图象）。假如前面提到的任何一个标志都没有使用，那么如果函数调用成功，就返回非零值；假如失败，则返回零值。

这个函数将返回与文件系统对象有关的信息，具体是什么信息则取决于为 SHGetFileInfo() 的 uFlags 参数分配的 SHGFI_标志，请大家参考表 16-6。准备检查的对象是在第一个参数 pszPath 里定义的，在缺省情况下，这是一个完整或者部分完整的文件名。假如设置了 SHGFI_PIDL 标志，这个参数就会用一个指针取代。

表 16-6 SHGetFileInfo（）函数使用的标志

SHFILEINFO 标志	值	说 明
SHGFI_ICON	0x000000100	返回一个图标的句柄，这个图标用于在屏幕上代表相应的文件
SHGFI_DISPLAYNAME	0x000000200	获得文件显示名称
SHGFI_TYPENAME	0x000000400	返回对文件类型进行说明的字符串
SHGFI_ATTRIBUTES	0x000000800	获取文件属性
SHGFI_ICONLOCATION	0x000001000	获取文件名，这个文件包含了用于代表文件的图标
SHGFI_EXETYPE	0x000002000	返回 EXE 类型
SHGFI_SYSICONINDEX	0x000004000	返回系统图象列表内的图标索引
SHGFI_LINKOVERLAY	0x000008000	增加覆盖于文件图标的链接
SHGFI_SELECTED	0x000010000	让图标显示于选定状态
SHGFI_LARGEICON	0x000000000	返回文件大图标
SHGFI_SMALLICON	0x000000001	返回文件小图标
SHGFI_OPENICON	0x000000002	取得文件打开图标
SHGFI_SHELLICONSIZE	0x000000004	返回外壳尺寸图标
SHGFI_PIDL	0x000000008	提醒函数 pszPath 是一个指针
SHGFI_USEFILEATTRIBUTES	0x000000010	函数应该使用包含于 dwFileAttributes 参数内的信息

通过使用恰当的 SHGFI_标志，我们就能取得一些与文件系统对象有关的信息，甚至还能够判断它的本质。SHGFI_EXETYPE 标志让我们有机会测试一个可执行模块，从而判断它是一个新执行模块（NE）、一个可移植模块（PE）、一个 MS-DOS 应用程序还是一个 Win32 控制台程序。返回值将按照下列的方案标识 .EXE 的类型：

返回值	**EXE 类型**
0	非执行文件或者一个出错条件
LOWORD=0x454e	NE，Win16 应用程序

LOWORD=0x5A4D　　Microsoft MS-DOS 的 .EXE，.COM 或者 .BAT 文件
LOWORD=0x0004550　基于 Win32 的控制台应用程序

本书附带 CD 的 Listing 16.2 里列举了一个名为 EXETYPE 的示范程序，它向我们展示了怎样判断一个可执行模块的类型。表 16-7 列出了 SHGetFileInfo () 函数第二个参数能够选用的所有文件属性。

SHELLAPI.H 包容文件里包含了对 SHFILEINFO 数据结构的说明和定义，这个数据结构的形式如下所示，这也是 SHGetFileInfo () 里的第三个参数。结构的每一项都根据 SHGetFileInfo () 返回时指定的标志填写了恰当的信息。

```
typedef struct _SHFILEINFOW
{
    HICON hIcon;
    int iIcon;
    DWORD dwAttributes;
    WCHAR szDisplayName [MAX_PATH];
    WCHAR szTypeName [80];
} SHFILEINFO;
```

表 16-7　由 SHGetFileInfo () 的 dwFileAttributs 参数使用的文件属性标志

文件属性	值	说　明
FILE_ATTRIBUTE_READONLY	0x00000001	只读文件
FILE_ATTRIBUTE_HIDDEN	0x00000002	隐藏文件
FILE_ATTRIBUTE_SYSTEM	0x00000004	系统文件
FILE_ATTRIBUTE_DIRECTORY	0x00000010	文件系统目录
FILE_ATTRIBUTE_ARCHIVE	0x00000020	归档文件
FILE_ATTRIBUTE_NORMAL	0x00000080	没有特定属性的文件
FILE_ATTRIBUTE_TEMPORARY	0x00000100	临时性存储文件
FILE_ATTRIBUTE_COMPRESSED	0x00000800	压缩文件

表 16-8 为大家列出了 SHFILINFO 数据结构的 dwAttributes 项使用的所有 SFGAO_标志，它能更好地对所检查对象的本质进行限定。

表 16-8　由 SHFILEINFO 数据结构的 dwAttributes 项返回的 SFGAO_标志

标　志	值	说　明
SFGAO_CANCOPY	DROPEFFECT_COPY	待检查的对象可以拷贝
FGAO_CANMOVE	DROPEFFECT_MOVE	待检查的对象可以移动
SFGAO_CANLINK	DROPEFFECT_LINK	待检查的对象可以生成一个快捷
SFGAO_CANRENAME	0x00000010L	待检查的对象可以更名
SFGAO_CANDELETE	0x00000020L	待检查的对象可以删除

（续）

标 志	值	说 明
SFGAO_HASPROPSHEET	0x00000040L	对象可以有一个属性表
SFGAO_DROPTARGET	0x00000100L	待检查的对象可以作为一个拖放终点
SFGAO_CAPABILITYMASK	0x00000177L	用于功能标志的屏蔽值
SFGAO_LINK	0x00010000L	待检查的对象是一个快捷
SFGAO_SHARE	0x00020000L	待检查的对象是一个共享对象
SFGAO_READONLY	0x00040000L	待检查的对象是只读的
SFGAO_GHOSTED	0x00080000L	待检查的对象应该显示一个暗淡（低亮度）的图标
SFGAO_DISPLAYATTRMASK	0x000F0000L	用于显示属性的屏蔽值
SFGAO_FILESYSANCESTOR	0x10000000L	待检查的对象由一个或者多个文件目录组成
SFGAO_FOLDER	0x20000000L	待检查的对象是一个文件夹
SFGAO_FILESYSTEM	0x40000000L	待检查的对象是文件系统的一部分
SFGAO_HASSUBFOLDER	0x80000000L	待检查的对象提供了子文件夹
SFGAO_CONTENTSMASK	0x01000000L	用于内容属性的屏蔽值
SFGAO_VALIDATE	0x02000000L	对缓存信息进行有效性检查
SFGAO_REMOVABLE	0x00000000L	待检查的对象驻留于可移动的媒体里

如下所示的代码段是从本书附带 CD-ROM Listing-16.2 的 EXETYPE 示范程序里摘录下来的。其中，SHGetFileInfo（）的作用是对可执行模块的返回类型进行查询，这种信息将封装于 iType 标识符的低字里。

```
...
// determine the EXE nature
iType = (int)SHGetFileInfo(szFileName,
                           0,
                           &sfi,
                           sizeof(SHFILEINFO),
                           SHGFI_EXETYPE);

// determine the type
switch(LOWORD(iType))
{
    // NE
    case 0x454e:
    {
        MessageBox(hwnd,"It is a Win16 application",szFileName,
                   MB_OK);
    }
        break;

    // MZ - MS-DOS
    case 0x5A4D:
```

```
        {
            MessageBox(hwnd, "It is an MS-DOS application", szFileName,
                        MB_OK);
        }
            break;

        // VXD
        case 0x454C:
        {
            MessageBox(hwnd, "It is a VxD module", szFileName, MB_OK);
        }
            break;

        // PE — Win32
        case 0x00004550:
        {
            MessageBox(hwnd, "It is a Win32 application", szFileName, MB_OK);
        }
            break;
    }
    ...
```

诸如外壳 Find 引擎的模块对符合搜索标准的每个项目都进行了简短的描述。EXE 被定义成应用程序，而 DLL 则被定义成应用程序扩展。这些字符串是与外壳对象关联在一起的一些说明的例子。SHGFI_TYPENAME 标志可用于返回这种类型的信息。

针对 SHGetFileInfo()这个 API 函数，我们已经对所有参数以及可能的标志组合进行了一番尽管冗长，然而很有必要的描述。现在让我们把注意力放在这样的一个问题上面：怎样返回系统图象列表。下面这个代码段向我们展示了返回系统图象列表有多么容易，同时，我们还图标的索引编号存储到了 SHFILEINFO 的 iIcon 项里：

```
    ...
    himlSmall = (HIMAGELIST)SHGetFileInfo("C:\\",
                                          0,
                                          &sfi,
                                          sizeof(SHFILEINFO),
                                                SHGFI_SYSICONINDEX |
                                                SHGFI_SMALLICON);

    himlLarge = (HIMAGELIST)SHGetFileInfo("C:\\",
```

```
                                    0,
                                    &sfi,
                                    sizeof(SHFILEINFO),
                                        SHGFI_SYSICONINDEX |
                                        SHGFI_LARGEICON);
...
```

在上面摘录的代码段里，没有准确说明的信息是一个具体的路径名，其中存储了准备访问的系统图象列表。正如我们看到的那样，我们假定这个路径是C:驱动器的根目录，它间接标识了系统图象列表。这个API将调用两次，从而分别取得小图标和大图标图象列表的句柄。随后，这些信息将用于从图象列表位图里提取出相应的图标，然后把它的值存储到HICON标识符里。在这以后，我们就可以在屏幕上的任何一个地方显示出图标。

```
...
hicon = ImageList_GetIcon(himlLarge, bi.iImage, ILD_NORMAL);
...
```

大家现在也许已经开始考虑利用系统图象列表能够做什么。其中，我们能做最聪明的一件事情就是允许用户通过Browse for older对话框选择一个系统外壳对象。当用户确认了他(她)的选择以后，就可以通过早先描述的机制取得对象图标、访问系统图象列表以及对与那个项目关联在一起的图标进行改动。

```
...
// load a fake icon
hicon = LoadIcon(hInstance, "shellobj");
ImageList_ReplaceIcon(himlLarge, bi.iImage, hicon);
...
```

正如我们从图16-15里看到的那样，标准的My Computer图标已经被SHELLOBJ示范程序的图标取代了。

Microsoft Plus! 可以帮助我们替换掉所有系统图标，用新的位图来代替，这看起来采用的好象是同一种办法。但是事实上，它采用的是另外一种方案，这种方案不需要用户进行干涉。在本书附带CD-ROM的Listing 6.4内，大家可以找到一个名为CLSID的示范程序，该程序向我们展示了这第二种技术。

2. 命名空间项目

现在让我们继续对命名空间进行探索。接下来，让我们把注意力放在系统如何对每个命名空间项目进行管理上面。根据与命令空间概念关联在一起的新术语定义，目录就是一个文件夹，其中包含了一系列命名空间项目。尽管目录肯定就是文件夹，但文件夹却并不一定就是目录。

正如我们以前提到的那样，Windows 95提供了一些特殊的文件夹以及几个虚拟文件夹。特殊文件夹是文件系统目录，它在系统的整体功能管理上扮演了一种重要的角色。这11个文

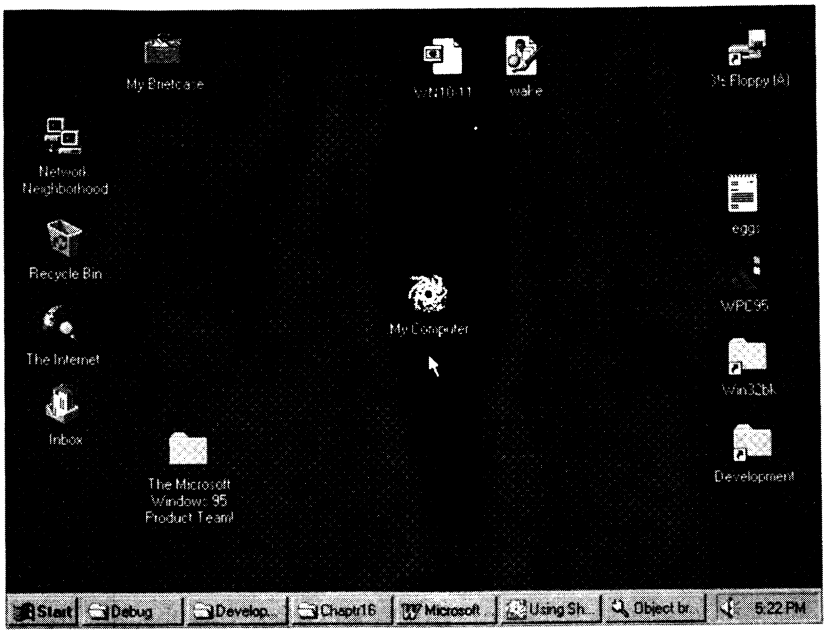


图 16-15　My Computer 现在提供了一个崭新的图标，这是由用户在 SHELLOBJ 示范程序里间接分配的

件夹的列表存储于注册表数据库的 HKEY_CURRENT_USER 根键下方。完整的路径是：

```
HKEY_CURRENT_USER
  Software
    Microsoft
      Windows
        CurrentVersion
          Explorer
            Shell Folders
```

图 16-6 向我们展示了注册表数据库里的这个分支。请大家也参考一下表 16-3，其中列出了 CLSID 的一份完整列表，其中有一部分引用的就是特殊文件夹。

虚拟文件夹在文件系统里并没有一个物理性的备份。控制面板、打印机以及字体文件夹都属于这种虚拟文件夹类别。它们可以当作几个项目的容器进行使用，这些项目对于控制面板和字体来说是一些特殊性的文件，然而这些文件夹在真正的文件系统里却是找不到的。因此，一个 Windows 95 文件夹最好定义成一个 OLE“组件对象模式”（COM）的对象。文件夹最主要的用途就是列举出自己特定的内容，同时实现一些相关的操作。文件夹内包含了项目和其他一些文件夹。对父文件夹内的一个项目进行标识的信息被定义成“项目标识符”（Item Identifier）。从程序开发的角度来看，项目标识符是当作一个长度可变的数据结构实现的，这个数据结构从属于 SHITEMID 类型。需要注意的是，该结构提供的信息只能用于对它父文件

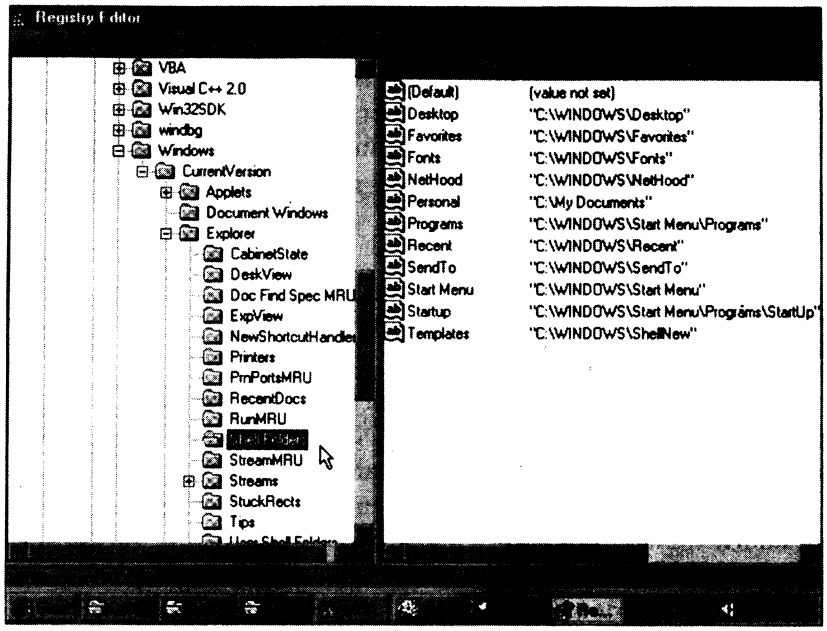


图 16-16　REGEDIT 显示了 11 个特殊文件夹

夹内的项目进行标识；离开这种应用场合以后，它就完全失去了自己的意义。

```
typedef struct _SHITEMID
{     //mkid
    USHORT cb;
    BYTE abID [1];
} SHITEMID, * LPSHITEMID;
```

唯一有用和可以辨认的信息是它的第一个参数:cb,这个参数里包含了数据结构的整体尺寸。剩余的数据是一个易变的信息块，它只能由项目驻留的那个父文件夹进行管理。

对于整个命名空间里的一个项目来说，为了唯一性地把它标识出来，系统定义了一个“项目标识符列表”(Item Identifier List)，这是另外一种数据结构，由一系列 SHITEMID 结构组成。项目标识符列表内的每个项目都是由一个 ITEMIDLIST 数据结构指定的，就象下面这样：

```
typedef struct _ITEMIDLIST
{     // idl
    SHITEMID mkid;
} ITEMIDLIST, * LPITEMIDLIST;
typedef const ITEMIDLIST * LPCITEMIDLIST;
```

其中，唯一用到的参数是一个 SHITEMID 类型。项目标识符列表是由一个或者多个连续的

ITEMIDLIST 数据结构组成的，这些结构都沿着字节的边界排列，后面跟随一个 16 位的零值。根据封装于 SHITEMID 数据结构里的、至今尚未具体文档化说明的信息，外壳可以在整个列表内向后搜索，从而唯一性地标识出一个项目。应用程序拥有对一个项目标识符列表的访问权。总而言之，项目标识符提供了某个对象在它的父文件夹内的信息，而项目标识符列表则提供了更为广泛的途径对整个命名空间内的一个项目进行标识。

为了知道与每个项目有关的某些附加信息，我们需要实现 IShellFolder 编程接口。在这种情况下，接口就是指向一个对象的指针，这种处理是通过一个名为“虚拟函数表”（Virtual Function Table）的中间层实现的。在虚拟函数表内驻留了一些“方法”（Method，通常也可称为成员函数），它们与那个特定的对象有关。通过对虚拟函数表的检查，我们就可以列出属于那个接口的所有“方法”。

考虑到 SHITEMID 数据结构只有在它的父文件夹内才有意义，所以 IShellFolder 接口就成了应用程序与外壳 COM（组件对象模式）之间的一座桥梁。所有 IShellFolder“方法”都列于表 16-9 内。

表 16-9　IShellFolder 接口

IShellFolder 成员函数	说　明
BindToObject (pidl, pbc, riid, ppvOut)	返回一个子文件夹的实例，这个子文件夹由 IDList (pidl) 指定
BindToStorage (pidl, pbc, riid, ppvObj)	返回子文件夹的一个存储实例，这个子文件夹由 IDList (pidl) 指定。在 Win95 的第一个版本里，外壳不会调用这个成员函数
CompareIDs (lParam, pidl1, pidl2)	对两个 IDLists 进行比较，并且返回比较结果。对于开发者来说，应该通过 lParam 传递 0 值，它表示“按名字排序”。假如两个 ID 指定了相同的对象，就会返回 0；假如 pidl1 应该放置于 pidl2 之前，就返回负值；假如 pidl2 应该放置于 pidl1 之前，就返回正值
CreateViewObject (hwndOwner, riid, ppvOut)	这个函数建立了文件夹本身的一个视窗对象。视窗对象是与外壳文件夹对象不同的一种实例。外壳保留这个函数
GetAttributesOf (cidl, apidl, prgflnOut)	返回那个文件夹内指定对象的属性。“cidl”和“apidl”指定了对象。“apidl”内只包含了简单的 IDLists。开发者可以用一系列标志对 * prgflnOut 进行初始化。外壳文件夹可以通过拒绝返回未指定的标志，从而对操作进行优化
GetUIObjectOf (hwndOwner, cidl, apidl, riid, prgflnOut, ppvOut)	这个函数可以建立一个 UI 对象，将其用于指定的对象。外壳开发者在 riid 里既可以传递 IID_IDataobject（用于传输操作），也可以传递 IID_IContextMenu（用于关联菜单的操作）
GetDiaplayNameOf ()	返回指定对象的显示名称。假如 ID 内包含了显示名称（在本地字符集里），它就会返回名称的偏移量。否则，它就会返回指向显示名称串（UNI_CODE）的一个指针，这个指针是由任务分配模块分配的，或者是在一个缓冲区内填写了的
SetNameOf ()	设置指定对象的显示名称。假如它也改变了 ID，就会返回新的 ID，这个 ID 是由任务分配模块分配的

除了表 16-9 内列出的“方法”以外，与其他所有 OLE 接口一样，IShellFolder 也提供了 QueryInterface，AddRef 以及 Release 成员函数。每个 OLE 对象和接口里都提供了 IUnknown。它的主要任务是获得由那个对象发出的所有接口的信息。实际上，这种任务是由 QueryInterface 方法完成的，它返回了指向自己支持的一个接口的指针。

HRESULT IUnknown:: QueryInterface (REFIID iid, void * * ppvObject);

其中，iid 代表接口 ID（即 Interface ID），而 ppvObject 则是一个间接性的指针，它指向自己假如支持的那个接口。所有接口 ID 的一份完整列表是相当长的，并且需要涉及几个包容文件。SHLGUID. H 里包含了与外壳进行交互作用时最常用的一些 IID，对此大家请参考表 16-10。

表 16-10 由 SHLGUID. H 定义的接口 ID

接口 ID	
IID_IContextMenu	IID_IShellLink
IID_IShellFolder	IID_IShellCopyHook
IID_IShellExtInit	IID_IFileViewer
IID_IShellPropSheetExt	IID_IEnumIDList
IID_IExtractIcon	IID_IFileViewerSite

接口 ID 是一种 128 位长的数字，它的结构与我们以前讨论地的 CLSID 类似。Win95 外壳已经分配了一个相应的信息块，其中包含了 256 个 GUID。这些 GUID 的格式如下所示：

000214xx-0000-0000-C000-0000000000

DEFINE_SHLGUID 宏可用于对 IID 进行管理，如下所示：

```
#define DEFINE_SHLGUID (name, l, w1, w2) \
        DEFINE_GUID (name, l, w1, w2, 0xC0, 0, 0, 0, 0, 0, 0,
                0, x46
```

例如，IShellFolder 是象下面这样定义的：

DEFIN_SHLGUID (IID_IShellFolder, 0x000214E6L, 0, 0);

对于一个应用程序，假如它需要对 IShellFolder 接口进行访问，就意味着它是与文件夹约束在一起的。为了解除这种约束，我们可以调用 Release 成员函数。与桌面文件夹约束起来以后，获得外壳命名空间的根项目就相当容易了，这是通过调用 API 函数 SHGetDesktopFolder () 来实现的。表 16-11 为大家列出了构成外壳开发工具的所有函数。

表 16-11 SH API

外壳 API	说 明
void WINAPI SHChangeNotify	通知系统外壳一个事件已经发生
SHNAMEMAPPING SHGetNameMappingPtr	取得一个 SHNAMEMAPPING 数据结构的地址，这个结构包含于一个文件名映射对象里
int SHGetNameMappingCount	取得一个文件名映射对象里的 SHNAMEMAPPING 数据结构的数量

（续）

外壳 API	说　明
BOOL WINAPI Shell_NotifyIcon	在任务栏通知区域内放置和管理一个小图标
BOOL WINAPI SHGetPathFromDList	把一个项目标识符列表转换成文件系统路径
DWORD WINAPI SHGetFileInfo	取得与文件内某个对象有关的信息，这些对象包括文件、目录或者驱动器等等
HRESULT WINAPI SHGetDesktopFolder	取得用于桌面文件夹的 IShellFolder 接口，桌面文件夹是外壳命名空间的根
HRESULT WINAPI SHGetInstanceExplorer	取得 Explorer 的 IUnknown 接口地址
HRESULT WINAPI SHGetMalloc	返回指向一个内存块的指针，这个内存块是在外壳里分配的
HRESULT WINAPI SHGetSpecialFolderLocation	取得特殊文件夹的位置
HRESULT WINAPI SHLoadInProc	在外壳的处理环境中，建立特殊对象类的一个实例
int WINAPI SHFileOperation	针对一个文件系统对象执行拷贝、移动、更名或者删除操作
LPITEMIDLIST WINAPI SHBrowseForFolder	强迫显示一个系统对话框，以便在系统命名空间内浏览
UINT APIENTRY SHAppBarMessage	把应用程序栏消息发送给外壳
void WINAPI SHAddToRecentDocs	在 Documents 次级菜单内增加一个新文档
void WINAPI SHFreeNameMappings	释放一个命名空间映射对象，这个对象是由 SHFileOperation 函数取得的

这些 C 函数是我们访问接口成员函数的一种简便方法，它把所有任务都封装到一次单独的调用里完成了。SHGetDesktopFolder（）就是这方面的一个例子，下面请大家看看这个函数的语法结构：

```
#include <shlobj.h>
HRESULT SHGetDesktopFolder (LPSHELLFOLDER * ppshf);
```

其中，唯一的参数是指向一个 IShellFolder 接口的指针，这个接口是按照下面这种形式定义的：

```
typedef IShellFolder * LPSHELLFOLDER;
```

SHGetDesktopFolder（）允许我们建立一个未初始化的对象，这个对象属于 CLSID_DESKTOP 类，用它可以返回自己希望的接口指针——一个普通的 void 指针：

```
CoCreateInstance (&CLSID_DESKTOP, NULL, CLSCTX_INPROC, &IID_IShellFolder, &pshf);
```

表 16-12　一个类对象的执行环境

标　志	值	说　明
CLSCTX_ALL	(CLSCTX_INPROC_SERVER | CLSCTX_INPROC_HANDLER | CLSCTX_LOCAL_SERVER)	对于建立和管理这一类对象的代码来说，运行它们的进程与指定类现场的那个函数的调用进程是相同的
CLSCTX_INPROC	(CLSCTX_INPROC_SERVER | CLSCTX_INPROC_HANDLER)	对这类对象进行管理的代码是一个进程内句柄。这是在客户进程内运行的一个 DLL，并且在通过远程访问这一类的实例时，实现了这种类的客户端结构
CLSCTX_SERVER	(CLSCTX_INPROC_SERVER | CLSCTX_LOCAL_SERVER)	用于建立和管理这一类对象的 EXE 代码是在独立的进程空间内载入的（运行于同一台机器，但却位于不同的进程）

BASE.H 内包含了所有 CLSCTX 定义，这些定义如表 16-12 所示，它们指出了当时的执行环境是什么。

在这个时候，我们明显可以看出 SHGetDesktopFolder () 是 SHGetSpecialLocation () 函数应用范围更广、更灵活的一种表现形式。一旦我们知道怎样与桌面文件夹（或者使用 SHGetSpecialFolderLocation () 和近似 CLSID 的其他任何一个文件夹）进行接口，就可以利用 IShellFolder 接口成员函数获得一些附加的信息。举个例子来说，IShellFolder::EnumObjects () 可以建立一个项目列举对象（一个 IEnumIDList 接口），从而列举出某个文件夹的内容。

```
#include <shlobj.h>
HRESULT IShellFolder::EnumObjects(LPSHELLFOLDER pIface,
                                  HWND hwndOwner,
                                  DWORD grfFlags,
                                  LPENUMIDLIST *ppenumIDL);
```

参数	说明
LPSHELLFOLDER pIface	IShellFolder 指针
HWND hwndOwner	假如准备显示一个消息框，这个参数指定的就是那个消息框的物主窗口的句柄（物主窗口通常就是应用程序的主窗口）
DWORD grfFlags	表 16-13 内列出的某个 SHCONTF 值
LPENUMIDLIST *ppenumIDL	用于接收一个 IEnumIDList 接口地址的指针
返回值	在正文里讨论
HRESULT	假如函数调用成功，就返回一个 NOERROR；否则就返回由 OLE 定义的一个出错代码

第一个参数是指向某个 IShellFolder 接口的指针，这个接口是前面通过调用 SHGetDesktopFolder ()函数已经返回的。搜索的标准可以用表 16-3 内列出的一个或者几个 SHCONTF_标志进行定义。对于第四个也是最后一个参数来说，它可以接收指向 IEnumIDList 接口的一个指针。IShellFolder 和 IEnumIDList 是两个基本的外壳接口。

表 16-13　用于 EnumObjects 成员函数的指针

列举标志	说　明
SHCONTF_FOLDERS	列出所有文件夹
SHCONTF_NONFOLDERS	列出非文件夹对象
SHCONTF_INCLUDEHIDDEN	列出隐藏/系统对象

IEnumIDList 有四个成员函数 (clone，next，reset 和 skip)，它们特别设计用于对项目标识符进行列举，如表 16-14 所示。

表 16-14　IEnumIDList 接口使用的四个成员函数

IEnumIDList 函数	说　明
clone	以给定的对象为基础，建立一个新的项目列举对象
next	移至下一个项目标识符
reset	返回列举序列的起始处
skip	跳过列举序列的一个或者多个项目

最后，其他两个 IShellFolder 成员函数是 GetDisplayNameOf () 和 GetAttributesOf ()，利用它们可以取得每个对象正在显示的名字以及为每个对象分配 SFGAO_属性，请参考表 16-8。除此以外，整个外壳命名空间内的漫游过程是由 IShellFolder:: BindToObject () 成员函数来实现的，它允许我们与某个给定文件夹的任何一个子文件夹约束到一起。

通过利用 IShellFolder 和 IEnumIDList 接口，以及利用 SH API，我们可以让自己的应用程序具备由系统外壳提供的几乎任何一种功能。通过 SHELLOBJ 示范程序，我们可以对自己在这个方向上的开发潜力有一个初步的了解。现在让我们把注意力转移到 EXETYPE 示范程序上面。图 16-17 显示了它运行以后的显示外观。

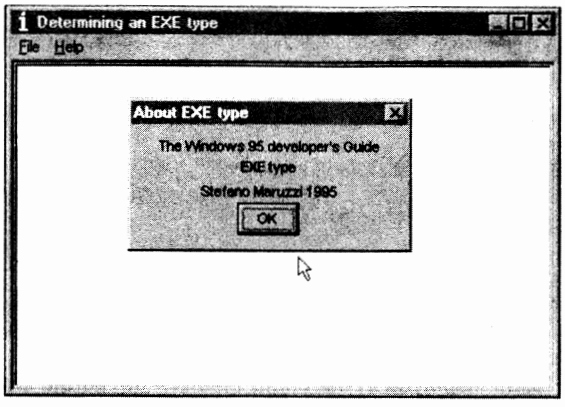


图 16-17　EXETYPE 示范程序向我们展示了怎样在应用程序里包含一些系统外壳对象

这个示范程序的用途是向用户报告一个可执行模块的本质。这种功能是通过拖放协议来实现的。用户应该先从屏幕上的任何位置（桌面或者一个打开的文件夹）选择一个 EXE 模块，然后拖动它经过 EXETYPE 的客户区域。这时就会显示出一个小的消息框，其中列出了选定 EXE 的类型，如图 16-18 所示。

File 下拉式菜单里列出了 Drives 菜单项。选择这个菜单以后，应用程序客户区里就会发生一些有趣的事情，如图 16-19 所示。所有文件系统驱动器都会用它们的标准图标在客户区内显示出来。

客户区由一个列表视窗窗口覆盖，这个区域最开始的时候是一片空白。选择了 Drives 菜单项以后，应用程序源代码内就会采取两种操作。首先，应用程序将通过调用 SHGetDesktopFolder () 获得 IShellInterface 指针。返回值将利用 FAILED 宏进行测试，这个宏在 WINERROR.H 头文件里是按照下面这种形式定义的：

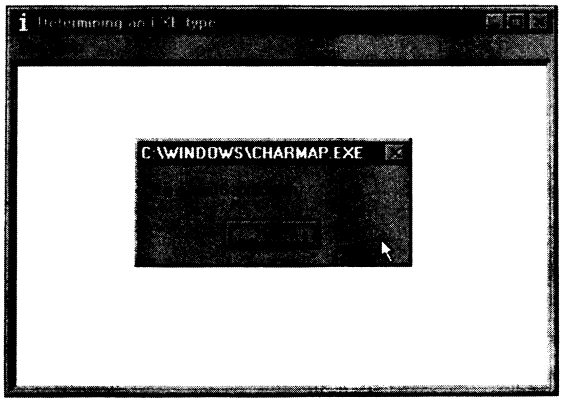


图 16-18　Character Map（字符映射表）仍然是一个 Win16 程序

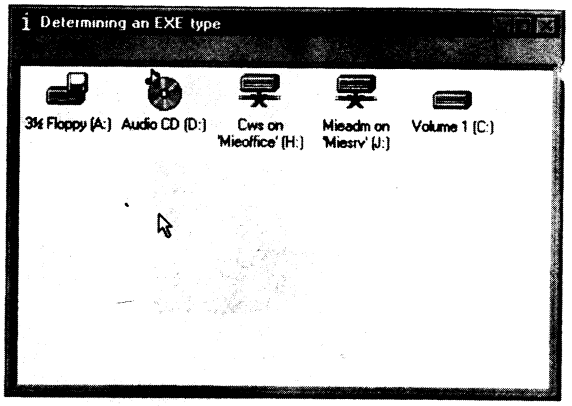


图 16-19　客户区内显示的图标是 My Computer 文件夹里的真正的外壳对象

```
#define FAILED(Status)((HRESULT)(Status)<0)
```

The opposite is SUCCEEDED, always defined in WINERROR.H:

```
#define SUCCEEDED(Status)((HRESULT)(Status) >=0)

...
case MN_DRIVES:
{
    HRESULT hres;
    LPITEMIDLIST pid;
    LPSHELLFOLDER pshf, pshfDrv;
```

```
// get the IShellFolder interface
hres = SHGetDesktopFolder(&pshf);
if(FAILED(hres))
    return FALSE;
...
```

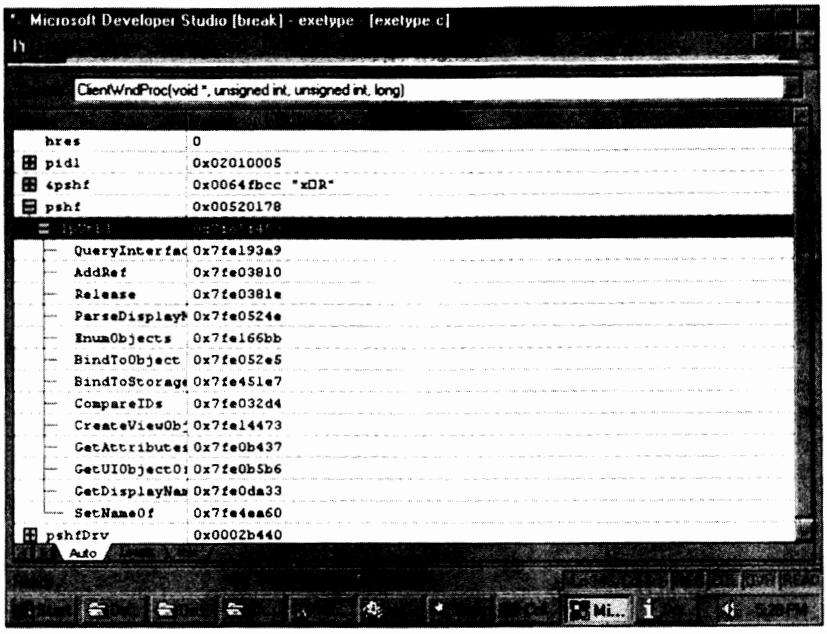


图 16-20　IShellFolder 接口使用的所有成员函数的一份完整列表

一旦我们获得了对 IShellFolder 的访问权限，接下来就可以列出支持的所有成员函数。图 16-20 向我们显示的是 MS Visual C++ 4.0 调试环境。在这以后，应用程序将发出请求，从而获得指向 CSIDL_DRIVES 文件夹的项目标识符列表的指针，这个列表里指定了 My Computer 对象。

```
...
// get the CLSID_DRIVES pidlle
hres = SHGetSpecialFolderLocation(hwnd, CSIDL_DRIVES, &pidl);
if(FAILED(hres))
    return FALSE;
...
```

随后，指向 IShellFolder 接口的指针将用于建立指向 CSIDL_DRIVES 子文件夹的一个 IShellFolder 指针。

```
...
// bind to the CSLID_DRIVES subfolder
hres = pshf -> lpVtbl -> BindToObject(pshf,
                                      pidl,
                                      0,
                                      &IID_IShellFolder,
                                      (LPVOID *)&pshfDrv);
if(FAILED(hres))
    return FALSE;
...
```

从这个时候开始，应用程序就会引用指向 IShellFolder 接口的 pshfDrv 指针，因为它只是要求对包含于这个文件夹内的信息进行检查。应用程序接下来将对用于填写列表视窗控件的所有信息进行处理，这些信息包括 My Computer 里提供的所有存储媒体的图标。这个任务是由PopulateListView()例程来完成的，它可以接收列表视窗的句柄、CSIDL_DRIVES 指针以及指向同一个文件夹的 IShellFolder 接口指针。

```
...
// populate the listview
PopulateListView(hwndLV, pidl, pshfDrv);
// release the memory block
pshfDrv -> lpVtbl -> Release(pshfDrv);
pshf -> lpVtbl -> Release(pshf);
}
    break;
...
```

实际上，EXETYPE 示范程序里的 PopulateListView () 例程使用的程序代码是非常简短的，它的主要用途是取得普通和小型的系统图象列表，然后把它们分配给覆盖于客户区上方的列表视窗。通过前面对 SHELLOBJ 示范程序进行的探讨，我们发现这种信息需要由 SHGetFileInfo () 返回。

由 PopulateListView () 完成的第二项任务是对外壳进行查询，从而取得与 CSIDL_DRIVES 特殊文件夹有关的每个对象的信息。为了完成这项任务，我们需要对活动的对象列表进行依次检查，只选出提供了物理存储设备的那些对象——换言之即系统连接的所有驱动器。EnumObjects () 成员函数可以建立一个列举对象，同时通过自己的第四个参数 (peidl) 返回指向 IEnumIDList 接口的一个指针。第三个参数则是表 16-13 里列出的一个或者多个 SHCONTF_定义。

```
    ...
    // enumerates the objects in My Computer
     hres = pshfDrv -> lpVtbl -> EnumObjects (pshfDrv, hwnd, SHCONTF_
FOLDERS, &peidl);
    if(FAILED(hres))
        return FALSE;
    ...
```

列举进程的第二项操作要求分配一个内存块，用它包含应用程序一些特定的信息，这些信息与不同的列表视窗项目是关联起来的。我们在这儿的目标是当用户选择客户区内显示的某个项目以后，通过一种简便的方式，从而对外壳接口以及数据结构进行访问。SHGetMalloc () 函数可以返回指向 IMalloc 接口的一个指针。

```
    ...
    hres = SHGetDesktopFolder(&pshf);
    if(FAILED(hres))
        return FALSE;
    ...
```

在这以后，应用程序就进入了一个 while 循环，这个循环持续下去的前提是 IEnumIDList 接口的 Next () 函数调用成功。

```
    #include <shlobj.h>
    HRESULT IEnumIDList:: Next (IEnumIDList *pEnumIDList,
                                ULONG celt,
                                LPITEMIDLIST *rgelt,
                                ULONG *pceltFetched);
```

参数	说明
IEnumIDList *pEnumIDList	指向 IEnumIDList 的一个指针
ULONG celt	由 rgelt 参数指定的数组元素数量
LPITEMIDLIST *rgelt	一个项目标识符列表的地址
ULONG *pceltFetched	一个无符号长标识符的地址，其中填写了由 reglt 返回的标识符的实际数量
返回值	在正文里讨论
HRESULT	假如函数调用成功，就返回一个 NOERROR；否则就返回由 OLE 定义的出错代码

在 EXETYPE 示范程序里，每调用一次 Next () 成员函数，就会取回一个项目，如下所示：

```
...
while(S_OK == peidl -> lpVtbl -> Next(peidl, 1, &pidl, &ulFetched))
{
    // get the attributes
    ulAttrs = SFGAO_FILESYSTEM;
    pshfDrv -> lpVtbl -> GetAttributesOf(pshfDrv, 1, &pidl, &ulAttrs);

    // check if it is a file system object
    if(! (ulAttrs & SFGAO_FILESYSTEM))
    {
        continue;
    }

    // OK, let's get some memory for our ITEMDATA struct
      plvid = (LPLVITEMDATA)lpMalloc -> lpVtbl -> Alloc(lpMalloc,
              sizeof(LVITEMDATA));
    if(! plvid)
        continue;

    // now get the friendly name that we'll put in the treeview...
    GetName(pshfDrv, pidl, SHGDN_NORMAL, szBuff);
...
```

在 while 循环里，应用程序将分配一个内存块，用它来存放 LVITEMDATA 数据结构，然后对刚才用 Next() 成员函数获得的项目的属性进行查询。在这以后，我们将调用 IShellFolder 的一个成员函数 GetAttributesOf ()，从而判断正在检查的那个项目的属性是什么。调用 GetAttributesOf () 函数的时候，我们可以指定自己希望检查的属性是什么。表 16-8 里列出的 SFGAO_ 标志不仅可用于 SHGetFileInfo () 外壳函数，也可以在这儿用于 GetAttributesOf () 函数。考虑到 EXETYPE 程序感兴趣的只是系统内的所有执行模块，所以这种搜索将限制在文件系统对象身上，不对虚拟文件夹进行处理。这样一来，诸如控制面板以及打印机等等就不会在列表视窗内显示出来。

这一部分代码最后需要调用 GetName () 例程，这个例程里封装了 GetDisplayNameOf () 成员函数。dwFlags 参数将设置成 SHGDN_NORMAL（参考表 16-15），而第四个参数则一个类型为 STRRET 的标识符。

```
...
if(NOERROR ! = pshfDrv —〉lpVtbl —〉GetDisplayNameOf(pshfDrv, pidl, d
wFlags,&str))
    return FALSE;
...
```

表 16-15　用于 IShellFllder：GetDisplayNameOf（）的标志

GetDisplayNameOf（）标志	值	说　明
SHGDN_NORMAL	0	缺省的显示名字
SHGDN_INFOLDER	1	与文件夹内显示的一个文件对象对应的显示名字
SHGDN_FORPARSING	0x8000	可以传递给 ParseDisplayName（）成员函数的显示名字

STRRET 数据结构是在 SHLOBJ.H 里定义的，它的形式如下所示：

```
typedef struct _STRREET
{ // str
    UINT uType;
    union
    {
        LPWSTR pOleStr;
        UINT uOffset;
        char cStr[MAX_PATH];
    } DUMMYUNIONNAME;
} STRRET, * LPSTRRET;
```

应用程序应该仔细地检查 uType 项，从而判断字符串在 STRRET 数据结构里的位置，请大家参考表 16-16。

表 16-16　用于 STRRET 数据结构的 uType 项的值

字符串类型	值	说　明
STRRET_WSTR	0x0000	字符串用 cStr 返回
STRRET_OFFSET	0x0001	字符串位于从项目标识符列表开头计算的 uOffset 偏移字节处
STRRET_CSTR	0x0002	字符串位于由 pOleStr 指定的地址处

应用程序的这一部分最后还要通过 SHGetFileInfo（）函数取得对象图标，然后向列表视窗控件传递一个 LV_ITEM 数据结构，从而让它在屏幕上显示出来。

```
...
lvi.iItem = iCtr++;
lvi.iSubItem = 0;
```

```
lvi.pszText = szBuff;
lvi.cchTextMax = MAX_PATH;
lvi.iImage = GetIcon(pifqThisItem, SHGFI_PIDL | SHGFI_SYSICONINDEX|
                     SHGFI_SMALLICON);

plvid -> pshf = pshfDrv;
pshfDrv -> lpVtbl -> AddRef(pshfDrv);

// now, make a copy of the ITEMIDLIST
plvid -> pidl = CopyITEMID(lpMalloc, pidl);
// store in lParam the LITEMDATA pointer
lvi.lParam = (LPARAM)plvid;

// add the item to the listview
if(ListView_InsertItem(hwndLV, &lvi) == -1)
 return FALSE;
...
```

假如大家仍不相信合并到应用程序里的图标真的与系统外壳对象对应，那么可以用鼠标右键单击C:驱动器按钮。这时会在屏幕上显示出与C:驱动器对象关联在一起的弹出式菜单，这样有力证明了这个图标具有相同的C:驱动器逻辑和特性，如图16-21所示。

我们仍然觉得不够。只需要再多加几行代码，就可以把外壳对象“嵌入”自己的定制应用程序里，并且在不改变功能的前提下实现对象的控制。事实上，假如双击应用程序客户区里的任何一个驱动器图标，就能得到缺省的响应。这很奇妙，不是吗，就像变魔术一样，我们自己就能模仿出系统外壳的行为。请大家参考图16-22。

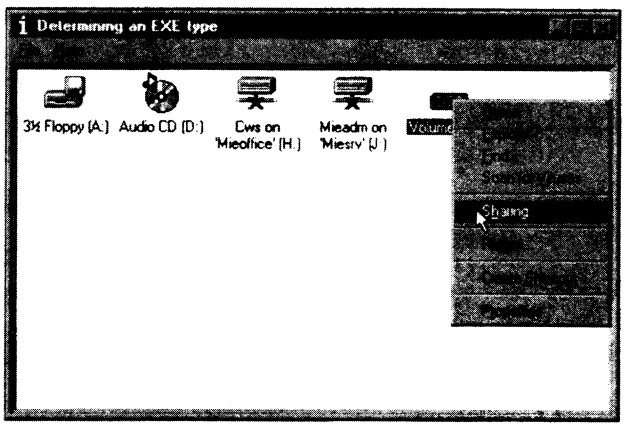


图16-21 EXETYPE示范程序里显示的每个对象都类似于等价的外壳对象

另外一个外壳接口在这种“魔术”的背后起到了关键性的作用。当我们访问与某个外壳

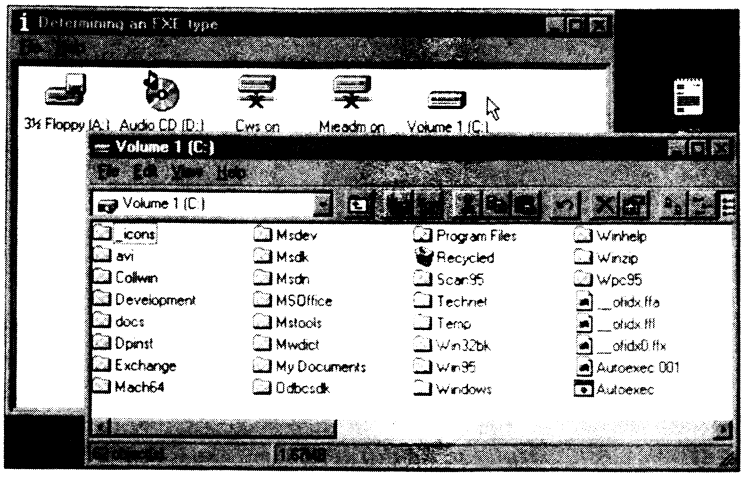


图 16-22　假如双击 C：驱动器图标，应用程序就会打开对应的文件夹

对象对应的关联菜单时，IContextMenu 接口以及它的三个成员函数——QueryContext ()，InvokeCommand () 和 GetCommandString () 就可以对这种访问进行控制。GetUIObjectOf () 成员函数则可用于提供指向 IContextMenu 接口的一个指针，如下所示：

```
#include <shlobj.h>
HRESULT IShellFolder::GetUIObjectOf(LPSHELLFOLDER pIface,
                                    HWND hwndOwner,
                                    UINT cidl,
                                    LPCITEMIDLIST *apidl,
                                    REFIID riid,
                                    UINT *prgfReserved
                                    LPVOID *ppvOut);
```

参数	说明
LPSHELLFOLDER pIface	指向 IShellFolder 接口的指针
HWND hwndOwner	一个窗口的句柄，假如显示出一个消息框，这个窗口就会作为那个消息框的物主使用
UINT cidl	在 apidl 参数里指定的对象数目
LPCITEMIDLIST *apidl	一个 ITEMIDLIST 数据结构数组的地址
REFIID riid	一个接口 ID（IID）
UINT *prgfReserved	NULL
LPVOID *ppvOut	与需要的那个 IID 有关的数据结构的地址
返回值	在正文里讨论
HRESULT	假如函数调用成功，就返回一个 NOERROR 值；假如失败，

就返回由 OLE 定义的一个出错代码

下面这个代码段向大家阐述了怎样访问与对象对应的关联菜单。首先，我们需要在第一个参数里传递指向 IShellFolder 接口的一个正确、有效的指针。最后一个参数则是 IContextMenu 接口的地址：

```
...
hres = lpshfParent -> lpVtbl -> BindToObject(pshf, hwnd,
                                             1, &pidl,
                                             &IID_IContextMenu,
                                             0,
                                             (LPVOID *)&lpcm);
if(FAILED(hres))
    return FALSE;
...
```

到现在为止，我们已经掌握了对关联菜单进行查询以及获得它的句柄时需要用到的所有工具。QueryContextMenu () 函数最后将按照下述的形式完成它的任务：

```
#include <shlobj.h>
HRESULT IContextMenu::QueryContextMenu(LPCONTEXTMENU pIface,
                                       HMENU hmenu,
                                       UINT indexMenu,
                                       UINT idCmdFirst,
                                       UINT idCmdLast,
                                       UINT uFlags);
```

参数	说明
LPCONTEXTMENU pIface	指向 IContextMenu 接口的指针
HMENU hmenu	菜单句柄
UINT indexMenu	菜单项索引位置
UINT idCmdFirst	最小的新菜单项标识符
UINT idCmdLast	最大的新菜单项标识符
UINT uFlags	零值，或者选用表 16-17 列出的 CMF_标志
返回值	在正文里讨论
HRESULT	假如函数调用成功，就返回一个 NOERROR；假如失败，就返回由 OLE 定义的一个出错代码

就 IShellFolder 的成员函数 QueryContextMenu () 来说，它的本质是对外壳的一种扩展，

该函数可以在标准的集合里增加新的菜单项。

表 16-17 QueryContextMenu () 函数使用的标志

QueryContextMenu 标志	值	说 明
CMF_NORMAL	0x00000000	常规操作
CMF_DEFAULTONLY	0x00000001	用户激活了缺省行动
CMF_VERBSONLY	0x00000002	只能用于快捷对象
CMF_EXPLORE	0x00000004	对 Explorer 程序的左侧显示的一个项目才有效

在 EXETYPE 示范程序里，我们将依赖这个函数提供的服务简化对菜单句柄 hmenu 标识符的初始化工作，这个句柄是与某个特定的外壳对象关联在一起的。

```
...
hres = lpcm -> lpVtbl -> QueryContextMenu(lpcm, hMenu,
                                          0,
                                          1,
                                          0x7fff,
                                          CMF_EXPLORE);
...
```

到这时为止，应用程序已经拥有了涉及对象关联菜单显示的所有信息。对 TrackPopupMenu () 的一次调用可以强迫弹出式菜单在恰当的屏幕位置显示出来。该函数的返回值与选中的菜单项 ID 对应。当需要执行由用户选择的命令时，这种信息就有用武之地了。执行这种命令时，我们需要调用 IContextMenu 的成员函数 InvokeCommand ()。

```
...
// create a popup menu
idCmd = TrackPopupMenu(hMenu,
                       TPM_LEFTALIGN | TPM_RETURNCMD |
                       TPM_RIGHTBUTTON,
                       lppt -> x,
                       lppt -> y,
                       0,
                       hwnd,
                       NULL);

if(! idCmd)
    return FALSE;

// fill the CMINVOKECOMMANDINFO structure
```

```
cmi.cbSize = sizeof(CMINVOKECOMMANDINFO);
cmi.fMask = 0;
cmi.hwnd = hwnd;
cmi.lpVerb = MAKEINTRESOURCE(idCmd - 1);
cmi.lpParameters = NULL;
cmi.lpDirectory = NULL;
cmi.nShow = SW_SHOWNORMAL;
cmi.dwHotKey = 0;
cmi.hIcon = NULL;

// invoke command
hres = lpcm -> lpVtbl -> InvokeCommand(lpcm, &cmi);
...
```

这段代码背后的逻辑其实是相当简单的。调用 QueryContextMenu () 的目的是获得与某个外壳项目关联在一起的菜单的句柄。由用户选择的命令并不能产生任何效果，因为相关的 WM_COMMAND 消息会到达物主窗口，而物主窗口里没有与之对应的代码。这就解释了为什么应用程序会捕获 TrackPopupMenu()的返回值，这些返回值与用户选中的项目是对应的。在这个时候，应用程序会把命令传递给 IContextMenu:: InvokeComand () 函数，从而对应的操作。作为另外一种选择，也可以调用 IContextMenu:: GetCommandString ()，从而取得与语言无关的某个特定项目的命令正文串。

16.3 对象的移动、拷贝、删除和更名

对于一个面向对象的用户界面来说，一种经常需要执行的任务就是拖动图标。我们可以在桌面到处拖动图标、把图标拖动到文件夹里、把图标从文件夹拖动到桌面或者把图标从一个文件夹拖动到另外一个文件夹里。某些时候，所有这些拖动操作都不过是简单的移动——通常是一种拷贝或者删除行动，有些时候则需要对一个外壳对象进行更名。外壳 API 针对这些操作提供了一些强有力的工具，利用它们可以在自己的应用程序里实现这些行动——就好象是由外壳本身执行的一样。删除文件是一种很常规的操作，Win32 API 函数 DeleteFile () 可以帮助我们实现文件的删除。但是，由于外壳对象没有自己对应的物理性拷贝，所以情况要显得稍微复杂一些。在这种情况下，DeleteFile () 不再是我们正确的选择了。SHFileOperation () 函数帮助我们完成对象的几种维护工作，如下所示：

```
#include <shlobj.h>
int SHFileOperation (LPSHFILEOPSTRUCT FileOp);
```

其中，唯一的参数就是某个 LPSHFILEOPSTRUCT 数据结构的地址，这个结构的构成如下所示：

```
typedef struct _SHFILEOPSTRUCT
{
```

```
    HWND hwnd;
    UINT wFunc;
    LPCWSTR pFrom;
    LPCWSTR pTo;
    FILEOP_FLAGS fFlags;
    BOOL fAnyOperationsAborted;
    LPVOID hNameMappings;
    LPCWSTR lpszProgressTitle;
} SHFILEOPSTRUCT, *LPSHFILEOPSTRUCT;
```

其中，hwnd 项标识了当一个消息框在屏幕中显示出来时，当作这个消息框的物主使用的一个应用程序窗口。当我们删除文件或者把几个对象从一个文件夹拷贝到另外一个时，那些好看的动画对话框就会在屏幕上显示出来。然而，大家同时要记住这样一点："Copying…"对话框只有在一个文件从某个驱动器传输到另外一个驱动器的时候才会显示出来。wFunc 项可以选用表 16-18 里列出的某个值，这些值指定了准备完成的操作。

表 16-18　SHFileOperation（）函数在 wFunc 项里支持的行动

SHFILEOPSTRUCT 行动标志	值	说　明
FO_MOVE	0x0001	移动一个或者多个对象
FO_COPY	0x0002	拷贝一个或者多个对象
FO_DELETE	0x0003	删除一个或者多个对象
FO_RENAME	0x0004	更名一个或者多个对象

第三和第四个参数是 pFrom 和 pTo，它们分别包含了用于指定源文件夹和目标文件夹的一个正文串。执行诸如此类的命令时，这种基本信息是相当重要的。删除一个或者多个文件的时候，pForm 就会分配准备删除那个对象的完整路径名。

SHFILEOPSTRUCT 的其余设置项提供了对操作进行定制的一些附加信息。表 16-9 为大家列出了用于 fFlags 的所有标志。这些标志的作用是定义用户与应用程序操作之间的交互作用。比如，FOF_ALLOWUNDO 可以尝试激活 Undo（撤消）特性，而 FOF_SILENT 则可以用于禁止进展指示窗的显示。

表 16-19　对调用 SHFileOperation（）时所发命令产生影响的标志

SHFILEOPERSTRUT 标志	值	说　明
FOF_MULTIDESTFILES	0x0001	pTo 项指定了多个目标文件，而不是一个单一的目录
FOF_CONFIRMMOUSE	0x0002	还没有实现
FOF_SILENT	0x0004	禁止显示进展指示窗
FOF_RENAMEONCOLLISION	0x0008	碰到有抵触的名字时，自动分配 Copy #x 前缀
FOF_NOCONFIRMATION	0x0010	如有必要，不经过用户确认便可直接选择 Yes to All 按钮
FOF_WANTMAPPINGHANDLE	0x0020	填写 hNameMappings 项
FOF_ALLOWUNDO	0x0040	尝试为一次后续的"撤消"（Undo）操作保存信息
FOF_FILESONLY	0x0080	只能对文件进行控制
FOF_SIMPLEPROGRESS	0x0100	显示一个简单的进展指示窗
FOF_NOCONFIRMMKDIR	0x0200	需要建立一个新目录时不要求用户进行确认

SHELLOBJ 示范程序允许用户把几个图标从安装示范程序的目录拷贝到 A：驱动器里。因此，首先准备一张空白磁盘，把它插入软盘驱动器，然后在 Actions 菜单里选择 Copy files 菜单项即可。图 16-23 向我们展示了动画对话框以及目标文件夹的外观。

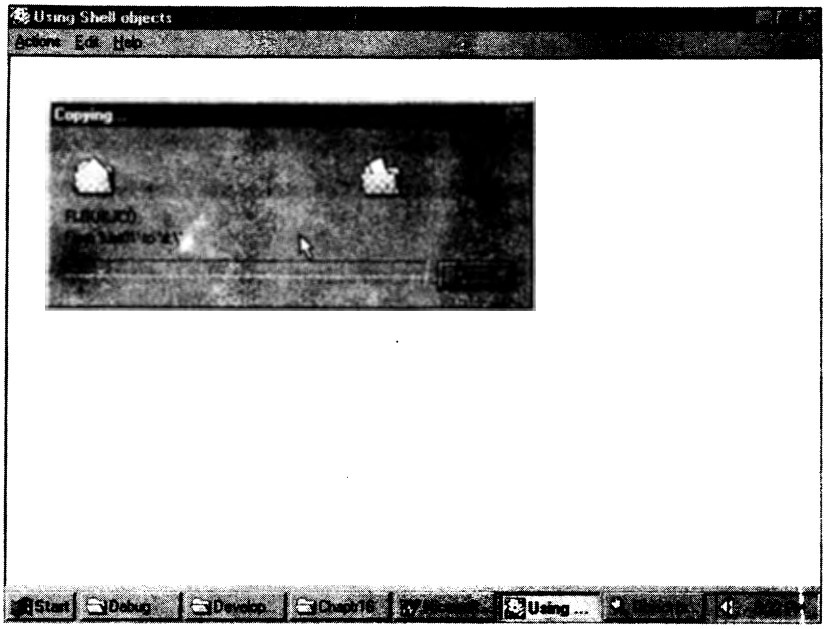


图 16-23　对文件和对象进行拷贝、移动、删除或者更名操作时，Win32 应用程序应该调用 API 函数 SHFileOperation（），从而实现与系统外壳相同的视觉效果

类似地，Delete files 菜单项可用于从 A：软盘里删除已经拷贝的图标，并且把它们发配到回收站（Recycle Bin）里。这时，标准的外壳对话框会再次在屏幕上显示出来，这些都是我们使用 SHFileOperation（）函数获得的效果，如图 16-24 所示。

与外壳进行交互作用时，另外还有一个 API 函数是相当有用的，即：SHChangeNotify（）。这个函数可以通知系统一个特定的事件已经发生了。

```
#include〈slobj.h〉
void SHChangeNotify（LONG wEventId，
                     UINT uFlags，
                     LPCVOID dwItem1，
                     LPCVOID dwItem2）；
```

参数	说明
LONG wEventId	表 16-20 列出的某种事件定义
UINT uFlags	表 16-21 列出的某个标志
LPCVOID dwItem1	一个四字节长的值，它的含义取决于 uFlags 参数
LPCVOID dwItem2	一个四字节长的值，它的含义取决于 uFlags 参数

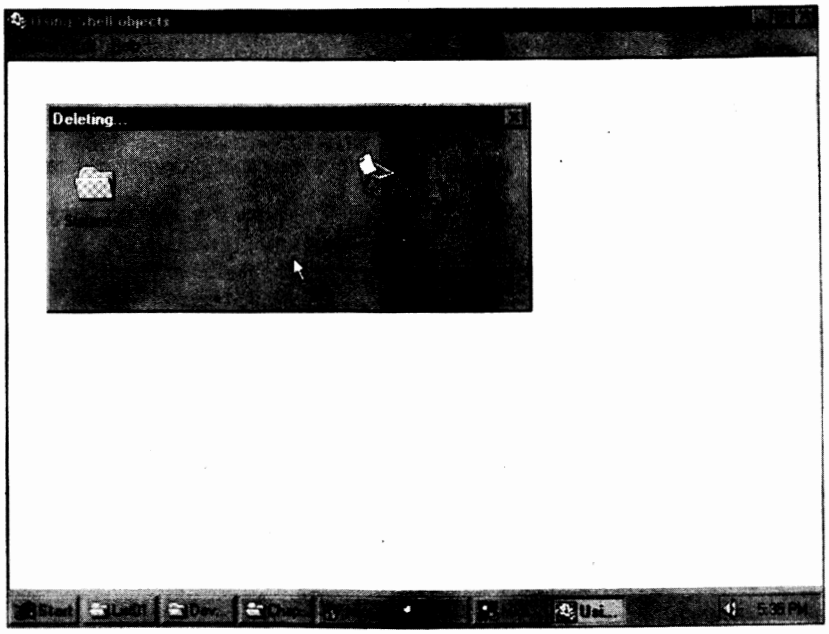


图 16-24　用 SHFileOperation（）函数删除文件可以模拟外壳的行为

返回值	在正文里讨论
void	没有返回值

SHChangeNotify（）背后的逻辑是相当简单和直观的。无论什么时候，只要应用程序执行了一个或许会对外壳产生视觉影响的操作，我们就可以利用对 SHChangeNotify（）的调用来通知外壳将要发生的那种行动。表 16-20 内列出的事件代表了也许会对外壳的显示和行为产生影响的行动。

表 16-20　通过 SHChangeNotify（）向外壳发出的事件信号

SHChangeNotify（）事件	值	说　明
SHCNE_RENAMEITEM	0x00000001L	一个项目已被更名
SHCNE_CREATE	0x00000002L	已经建立了一个文件
SHCNE_DELETE	0x00000004L	已经删除了一个文件
SHCNE_MKDIR	0x00000008L	一个新目录已经建立起来了
SHCNE_RMDIR	0x00000010L	已经删除了一个目录
SHCNE_MEDIAINSERTED	0x00000020L	已经插入了某种类型的可移动媒体
SHCNE_MEDIAREMOVED	0x00000040L	已经撤消了某种类型的可移动媒体
SHCNE_DRIVEREMOVED	0x00000080L	已经撤消了一个驱动器
SHCNE_DRIVEADD	0x00000100L	已经增添了一个驱动器
SHCNE_NETSHARE	0x00000200L	对网络上的某种资源进行共享
SHCNE_NETUNSHARE	0x00000400L	撤消与网络上某种资源的连接
SHCNE_ATTRIBUTES	0x00000800L	文件属性已经改变了
SHCNE_UPDATEDIR	0x00001000L	目录的内容已经发生了更新
SHCNE_UPDATEITEM	0x00002000L	打印机或者文件的属性已经改变了
SHCNE_SERVERDISCONNECT	0x00004000L	撤消了与一台网络服务器的连接

（续）

SHChangeNotify（）事件	值	说　明
SHCNE_UPDATEIMAGE	0x00008000L	系统图象列表内的一幅图象已经改变了
SHCNE_DRIVEADDGUI	0x00010000L	从图形显示上反映，一个网络驱动器已经增添到系统上了
SHCNE_RENAMEFOLDER	0x00020000L	已经删除了一个文件夹
SHCNE_FREESPACE	0x00040000L	发生了某种类型的事件
SHCNE_ASSOCCHANGED	0x08000000L	一种关联关系已经改变了
SHCNE_DISKEVENTS	0x0002381FL	发生了一个磁盘事件
SHCNE_GLOBALEVENTS	0x0C0581E0L	发生了一个全局事件
SHCNE_ALLEVENTS	0x7FFFFFFFL	所有事件
SHCNE_INTERRUPT	0x80000000L	发生了一次中断

根据不同的通知事件信息，我们还应该在dwItem1和dwItem2参数里传递某些附加的信息。这两个参数是指向某个void数据类型的一个常规的指针。第二个参数是uFlags，它定义了这两个存储类的本质，请大家参考表16-21。

表16-21　对SHChangeNotify（）第二个参数的内容进行指定的标志

用于**SHChangeNotify**（）的标志	值	说　明
SHCNF_IDLIST	0x0000	dwItem1和dwItem2是项目标识符列表
SHCNF_PATH	0x0001	dwItem1和dwItem2是路径名称
SHCNF_PRINTER	0x0002	dwItem1和dwItem2是打印机名称
SHCNF_DWORD	0x0003	dwItem1和dwItem2是双字
SHCNF_FLUSH	0x1000	冲洗系统事件缓冲区
SHCNF_FLUSHNOWAIT	0x2000	冲洗系统事件缓冲区，并且不等待操作的结果

当应用程序建立一个新目录的时候，就应该通过这条命令向外壳发出通知：

```
...
// notify the shell that you made a chage
SHChangeNotify(SHCNE_MKDIR, SHCNF_PATH, lpszFolder, 0);
...
```

其中，lpszFolder是新文件夹的完整路径名。在下面的另外一个例子里，应用程序将通知外壳一个关联关系已经发生了改变：

```
...
// ask the shell to refresh the icon list
SHChangeNotify (SHCNE_ASSOCCHANGED, SHCNF_FLUSHNOWAIT,
                0, 0);
...
```

不幸的是，微软至今还没有对针对不同事件类型分配给dwItem1和dwItem2参数的这种值进行具体的文档化。因此，我们必须不断地进行试验，从而找出和通知消息有关的正确的外壳响应。

16.4 最近常用文档的管理

对于一个设计良好的 Win32 应用程序来说，每次当它打开、建立或者管理了一个文档文件时，都应该把文档最近的状态通知给外壳。假如我们打开 Start 菜单，就会发现 Documents 菜单项可以引入一个次级菜单，其中的内容将不时地发生变化，如图 16-25 所示。

实际上，对于列表这个次级菜单内的所有文档来说，它们都是一些直接从外壳载入的文件，这种载入是用户双击某个相应对象图标的结果。我们在这儿的想法是为用户提供一种简便的方法，使他们可以避免复杂和耗时的搜索工作，从而直接打开上一次已打开过的某个文档，这样便可立即继续对文档的处理。根据 Windows 95 的程序设计准则，一个应用程序应该实现这种行为。比如，MS Word 7.0 for Windows 95 就实现了这种行为，但这个程序只是众多软件产品的一小部分而已，其他程序都应该根据这种思路进行设计。事实上，这种效果相当容易就可以达到，实现它只是举手之劳。在这儿，所有相关的操作都需要用到 SHAddToRecentDoscs () 这个 API 函数，如下所示：

```
#include <shlobj.h>
void SHAddRecentDocs(UINT uFlags, LPCVOID pv);
```

参数	说明
UINT uFlags	假如为 SHARD_PATH，表明第二个参数是一个文件路径名称；假如为 SHARD_PIDL，就表明它是一个指针
LPCVOID pv	要么是个路径名，要么是个指针
返回值	在正文里讨论
void	没有返回值

下面这个例子向大家揭示了怎样在列表内增加一个 MS Word 文档：

```
...
// add a document to the Documents list
SHAddToRecentDocs(SHARD_PATH,"c:\\win32bk\\win32_16\\list01\\
                  readme.doc");
...
```

为了清除最近常用的文档列表，我们只需要在调用 SHAddToRecentDocs () 的时候用它的第二个参数传递一个 NULL 值就可以了：

```
...
// clear the Documents list
SHAddToRecentDocs(SHARD_PATH, NULL);
...
```

为了增加由自己的项目标识符列表指定的一个对象，我们首先必须用 SHGetSpecialFolderLocation () 获得一个指针，然后把它传递给 SHAddToRecentDocs () 的第二个参数。

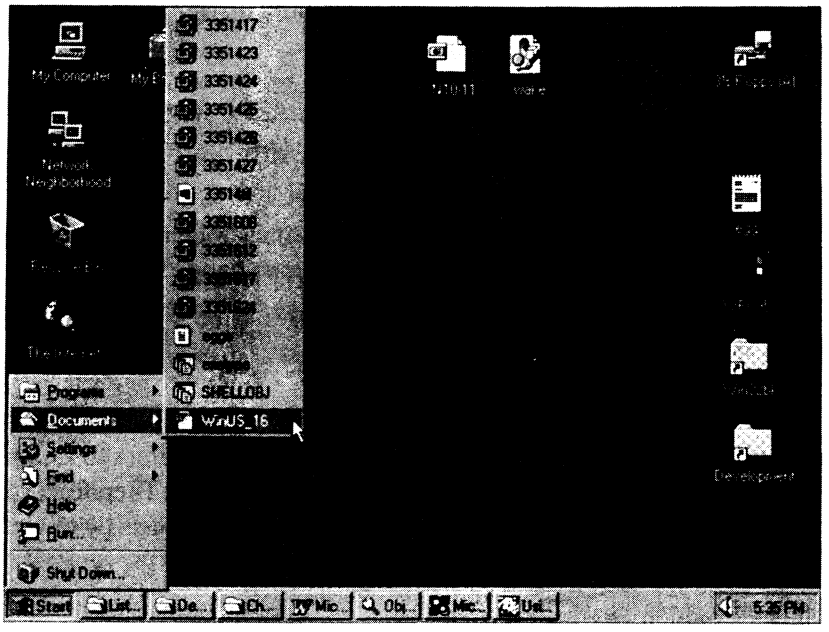


图 16-25　Documents 次级菜单列出了由外壳激活的所有文档

16.5　快捷的建立和推敲

对于 Windows 95 外壳来主，最有趣的一种特性也许要算快捷（Shortcut）。“快捷”这个术语也许显得并不是很适当，它的本质就是带有 .LNK 扩展名的一个文件系统对象。快捷的作用相当于对另外一个对象的引用（或者说链接）。由于快捷的本质是文件，所以它所引用的对象可以是任何一个命名空间项目：文件、文件夹、特殊文件夹以及虚拟文件夹等等，甚至还可以引用另外一个快捷。除此以外，引用的项目也可以驻留于远程系统。在这种情况下，对象路径将按照“统一命名规范”（UNC，Universal Naming Convention）的语法进行定义。在 UNC 里，计算机的名字将用起始的两个反斜杠标识，比如：\\MIESRV\TWENY\TOPO.LNK。

通常情况下，快捷是一个具有有限尺寸的文件，大约只有几百个字节长。另外，文件内的信息是用一种二进制格式存储的。一个外壳接口 IShellLink 将对与快捷关联起来的所有操作进行管理。根据这个接口定义的术语，由一个快捷引用的对象被称为“链接目标”（target）。

为了建立一个快捷，我们需要用到两种信息：链接目标的名字和路径，以及快捷驻留的目的地。在快捷的建立过程中，第一个步骤是取得指向 IShellLini 接口的一个指针。没有一个外壳 API 可以帮助我们自动完成这一工作，所以必须调用更普通的一个 OLE 函数 CoCreateInstance（）。

```
STDAPI CoCreateInstance (REFCLSID rclsid,
                         LPUNKNOWN pUnkOuter,
```

DWORD dwClsContext，
REFIID riid)；

参数	说明
REFCLSID rclsid	一个 CLISD_定义
LPUNKNOWN pUnkOuter	假如为 NULL，表明它不是某个集合的一部分
DWORD dwClsContext	一个 CLSCTX_定义（请参考表 16-12）
REFIID riid	一个接口 ID
LPVOID * ppv	指向接口的一个指针
返回值	在正文里讨论
STDAPI	假如函数调用成功，就返回一个 S_OK；假如失败，就返回由 OLE 定义的一个出错代码

对一个快捷进行引用的一个类 ID 是 CLSID_ShellLink，而接口 ID 则是表 16-10 中列出的 IID_IShellLink。下面列出的那个 CreateLink（）例程是从 SHELLOBJ 示范程序里摘录下来的，它向我们展示了怎样建立一个快捷。这个接口提供了几个成员函数，如图 16-26 所示。

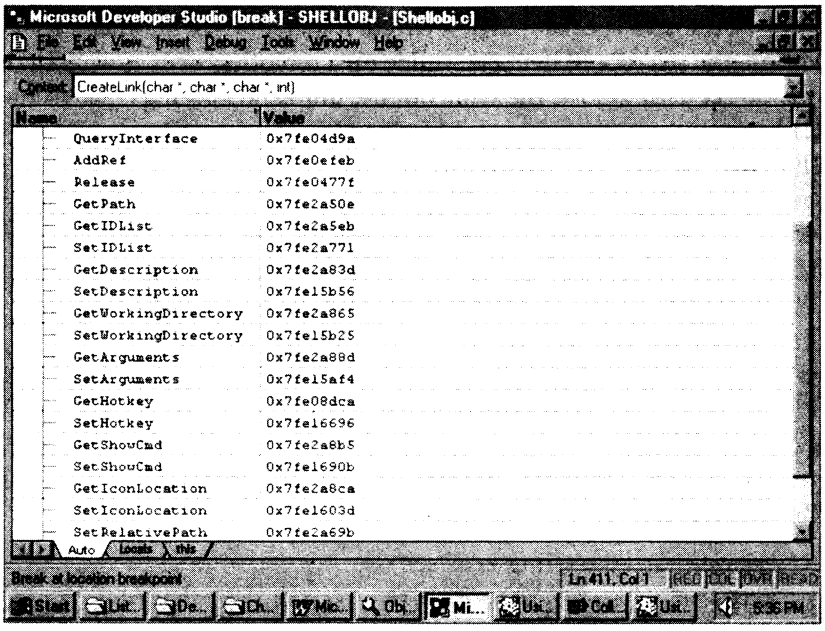


图 16-26　MS Visual C++ 4.0 的调试器可以让我们对 IShellLink 接口的所有成员函数进行检查

在这些成员函数里，与快捷建立有关的只有 SetPath（）函数。正如通过它的名字可以看出的那样，这种成员函数要求用到目标对象的完整路径名。在 CreateLink（）例程里，我们调用了 SetDescription（）和 SetHotkey（），这两个函数分别用于把一个描述性的正文串与链接关联在一起，以及为它分配一个键盘加速键。

```
...
HRESULT CreateLink (LPCSTR pszPathObj, LPSTR pszPathLink, LPSTR
                    pszDesc,
                    BOOL fPIDL)
{
    HRESULT hres;
    IShellLink * psl;
    IPersistFile * ppf;
    WORD wsz[MAX_PATH];

    // get a pointer to the IShellLink interface
    hres = CoCreateInstance(&CLSID_ShellLink,
                            NULL,
                            CLSCTX_INPROC_SERVER,
                            &IID_IShellLink,
                            &psl);
    if(FAILED(hres))
        return FALSE;
    // set the path to the shortcut target, and add the description
    psl -> lpVtbl -> SetPath(psl, pszPathObj);
    psl -> lpVtbl -> SetDescription(psl, pszDesc);
    psl -> lpVtbl -> SetHotkey(psl, MAKEWORD('R', HOTKEYF_SHIFT |
                               HOTKEYF_CONTROL));

    // query IShellLink for the IPersistFile interface for saving the shortcut
    hres = psl -> lpVtbl -> QueryInterface(psl, &IID_IPersistFile, &ppf);
    if(FAILED(hres))
        return FALSE;

    // ensure that that string is ANSI
    MultiByteToWideChar(CP_ACP, 0, pszPathLink, -1, wsz, MAX_PATH);

    // save the link by calling IPersistFile::Save
    hres = ppf -> lpVtbl -> Save(ppf, wsz, STGM_READWRITE);

    // release the IPersistFile interface
    ppf -> lpVtbl -> Release(ppf);
    // release the IShellLink interface
    psl -> lpVtbl -> Release(psl);
```

```
        return hres;
    }
    ...
```

.LNK 文件的物理性建立是通过 IPersistFile 接口完成的，这个接口的指针将通过 QueryInterface ()成员函数发出调用请求。IPersistFile：：Save () 成员函数的作用是对所有实际的输出操作进行管理。在这以后，为了建立一个快捷，我们最终还要做两件事情——对两个接口进行访问，这两个接口分别是 IPersistFile 和 IShellLink。

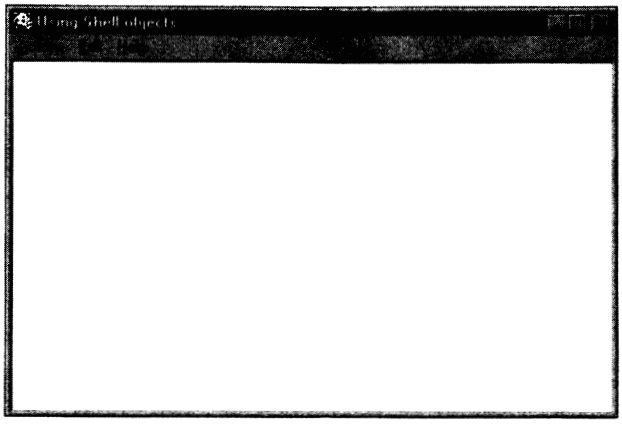


图 16-27　Word 文档快捷已经由 SHELLOBJ 示范程序建立起来了

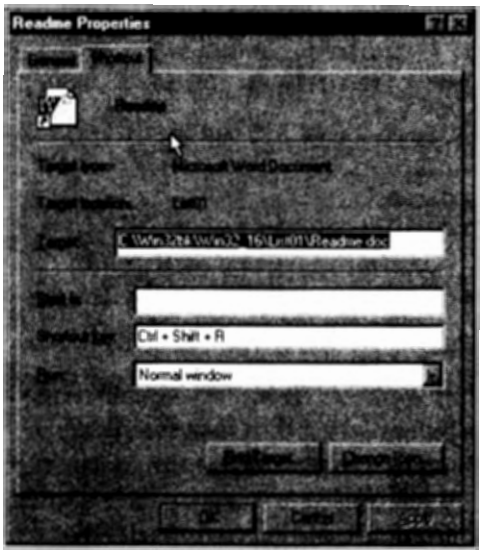


图 16-28　与 SHELLOBJ 建立的快捷关联起来的属性表

通过从 Actions 菜单内选择 Create Shortcut 菜单项，SHELLOBJ 建立了针对桌面上一个名为 README. DOC 文档的快捷，如图 16-27 所示。假如我们对这个链接的属性进行检查，就能发现其中列出了加速键组合：Ctrl + Shift + R。这种加速键是我们在程序中通过 IShellLink：：SetHotkey（）成员函数分配的，如图 16-28 所示。假如双击这个图标，就能启动 MS Word 7.0，然后强迫它载入目标文档。

在 SHELLOBJ 程序里，一个简单的例程 CreateLink（）就足以帮助我们应付大多数常见的情况。假如我们需要建立针对某个外壳项目的快捷，而不是针对文件系统对象的快捷，就应该调用 IShellLink：：SetIDList（）成员函数，用它取代以前的 IShellLink：：SetPath（）。通过下面这个代码段，我们就可以了解建立针对某个外壳对象（在这种情况下是控制面板）的快捷时需要对 SetPath（）调用进行替换的指令：

```
...
hres = SHGetSpecialFolderLocation(HWND_DESKTOP, CSIDL_CONTROLS,
                                  &pidl);
if(FAILED(hres))
    return FALSE;

// set pidl
hres = psl -> lpVtbl -> SetIDList(psl, pidl);
...
```

事实上，在 Start 菜单下面的次级菜单里，列出的所有菜单项都是针对系统内其他对象的快捷。在 Windows 3.x 里，一个新应用程序安装时它会自动建立一个程序管理器组，并且通过“动态数据交换”(DDE) 规范在其中放入恰当的图标。Windows 95 外壳仍然支持这种 DDE 规范，然而，出于性能方面的考虑，我们应该考虑另外一种方案，也就是使用快捷。实际上，由设置进程和外壳的交互作用而生成的对象无非就是一个快捷。与 DDE 协议的复杂程度比较起来，快捷的建立要简单得多。考虑到这方面的原因，所以建立快捷是目前为止最简单的方法。

16.6 发送文档

在任何 Win32 应用程序里应该提供的另外一项有趣的特性是 Send…菜单项。如果我们仔细观察 SHELLOBJ 示范程序，如图 16-29 所示，就能发现 Actions 顶级菜单下的 Send…菜单。

Send…菜单项的目的是让用户把当前文档与一条邮件消息或者其他任何一种目标设备关联起来。这种目标设备可以是传真软件或者公文包，甚至可以是一个系统文件夹。为了实现这种功能，我们必须使用 API 的 MAPI 集合。MAPI 是“报文处理 API”简称。利用 MAPI，多个应用程序就可以跨越许多硬件平台与多个报文处理系统进行无缝连接，从而实现两者之间的交互作用。幸运的是，为了让应用程序把自己的当前文档与一条邮件消息联系起来，我们只需要调用 MAPISendDocuments（）这个 API 函数就足够了。SHELLOBJ 示范程序向我们展示了这种特性如何在 MN_SEND 菜单项命令下实现。在源代码的起始处，我们首先需要

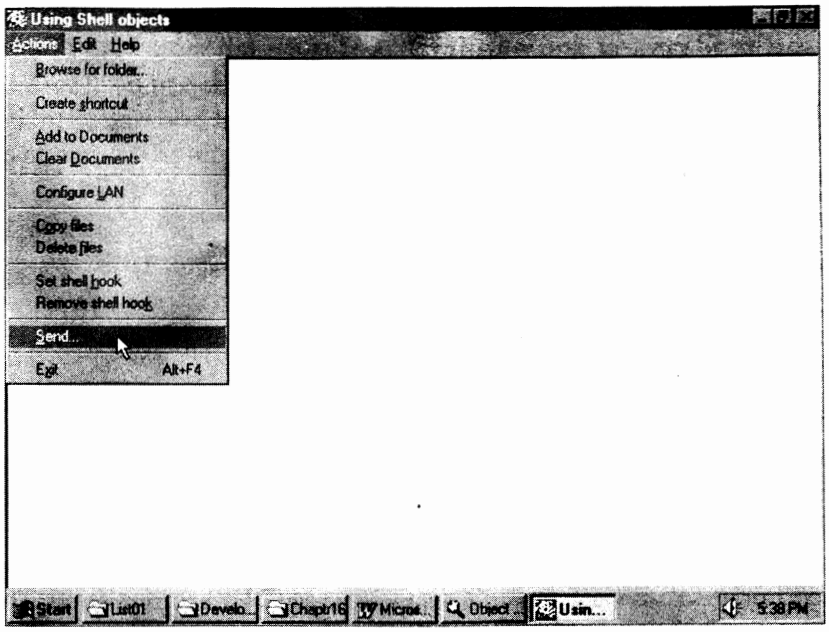


图 16-29　一个 Win32 应用程序应该提供 Send…菜单项，
以便使当前文档与一条邮件消息联系起来

定义一种新的数据类型，将其当作指向 MAPISendDocuments（）函数的一个指针使用。

```
...
typedef ULONG (WINAPI *PFNMAPISENDDOCUMENTS)(HWND, LPSTR,
                    LPSTR, LPSTR, ULONG);
...
```

在这以后，我们将通过 LoadLibrary（）把 MAPI32.DLL 库载入进程地址空间，并且把 API 函数 MAPISendDocuments（）的地址存储到类型适当的指针里。正如这个函数的名字暗示的那样，它可以一次性把多个文档与一条单一的邮件消息连接到一起。第二个参数是用作分界标志的一个字符，该字符将在第三个参数里对文档的名字进行分隔。在下面这个例子里，只有一个文档与一条邮件消息连接到了一起，同时提供了这个文档的完整路径名。

```
...
case MN_SEND:
{
    HANDLE hLibrary;
    PFNMAPISENDDOCUMENTS lpfnMAPISendDocuments;

    // load the MAPI library
    if((hLibrary = LoadLibrary("MAPI32")) < (HANDLE)32)
        return FALSE;
```

```
        if((lpfnMAPISendDocuments = (PFNMAPISENDDOCUMENTS)
                                    GetProcAddress
                                    (hLibrary, "MAPISendDocuments")) ==
                                    NULL)
            return FALSE;

        (*lpfnMAPISendDocuments)(hwnd, ";",
                                 "c:\\win32bk\\win32_16\\list01\\readme.doc",
                                 "readme.doc", 0);
        FreeLibrary(hLibrary);
    }
        break;
...
```

在离开这一部分代码之前，应用程序将卸载库，从而释放未使用资源。图 16-30 向大家展示了最终的结果。一条邮件消息已经建立好了，同时 README.DOC 文档也在编辑区域显示出来了，它是作为一种附着物插入的。

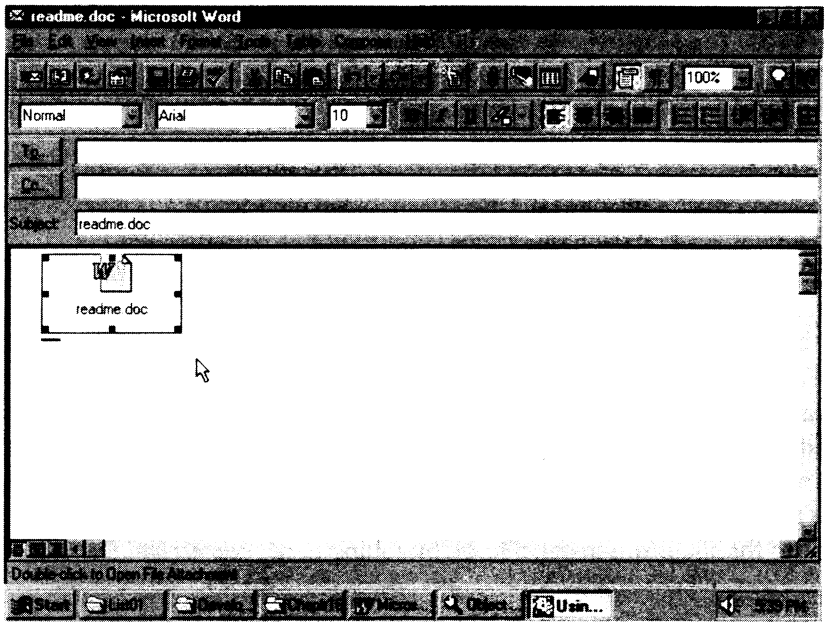


图 16-30　利用 MAPISendDocuments ()，一个文档与
一条邮件消息的连接是很直接的操作

与文档发送证明有关的是 Send to 菜单项。几个外壳对象的关联菜单内都可以找到这个选项，如图 16-31 所示。它的次级菜单列出了几个目的地，比如一个报文处理应用程序、一个文件夹、A 驱动器或者公文包（Briefcase）等。按照某种方式，Send to 菜单项会强迫选中对

象的一个拷贝进入目标位置。假如这个目标位置是MSExchange，操作就会转换成为文件的连接。

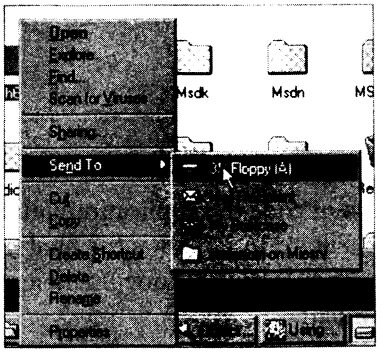


图 16-31　Send to 菜单项引入了选择范围很宽的一系列目标位置

16.7　外壳的挂接

为了获得对系统外壳更广泛的控制权，我们必须安装一个特殊的挂接模块，我们将其称为“外壳挂接模块”(shell hook)。通常情况下，挂接模块就是一个过滤器，放置于系统消息队列的级别上——也就是说，在一条消息抵达自己的目标线程消息队列之前。通过一个挂接模块，应用程序就可以拦截针对系统任何一个窗口的每一种事件。挂接模块在这儿扮演了一个检查官的角色。这种方案的缺点也是很明显的。由于几乎每时每刻都要进行附加的检查工作，所以导致了性能的降低。从另外一方面来说，通过激活一个挂接模块，应用程序就能访问一些以前有可能废弃的信息。在各种系统挂接模块之间有一个特殊的 WH_SHELL，它明确设计用于拦截与外壳相关的消息。我们可以通过调用 SetWindowHookEx () 函数来实现挂接模块的安装：

```
#include 〈winuser.r〉
HHOOK SetWindowsHookEx(int idHook,
                       HOKPROC hkprc,
                       HINSTANCE hMod,
                       DWORD dwThreadID);
```

参数	说明
int idHook	挂接模块类型(请参考表 16-22)
HOKPROC hkprc	挂接例程
HINSTANCE hMod	驻留挂接例程的模块的句柄
DWORD dwThreadID	与挂接模块关联在一起的线程 ID
返回值	在正文里讨论
HHOOK	一个挂接句柄；假如函数调用失败，就返回一个 NULL 值

我们通常把一个挂接模块安装到一个 DLL 里，以便让系统内的每个进程都能对其进行访问。SetWindowsHookEx（）的第三个参数就是那个 DLL 的句柄。假如只想在一个线程里访问挂接模块，就可以把挂接例程调整到安装挂接模块的那个相同的线程里，同时在第三个参数里传递 hInstance。

表 16-22　挂接模块的类型

挂接类型	值	说　明
WH_MSGFILTER	(-1)	对话框挂接模块
WH_JOURNALRECORD	0	记录输入消息
WH_JOURNALPLAYBACK	1	回放以前通过 WH_JOURNALRECORD 挂接模块记录下来的输入消息
WH_KEYBOARD	2	键盘挂接模块
WH_GETMESSAGE	3	用于张贴消息的挂接模块
WH_CALLWNDPROC	4	窗口进程挂接模块
WH_CBT	5	以计算机为基础的培训挂接模块
WH_SYSMSGFILTER	6	系统消息挂接模块（对话框、消息框、菜单或者滚动条）
WH_MOUSE	7	鼠标挂接模块
WH_HARDWARE	8	硬件和即插即用挂接模块
WH_DEBUG	9	调试挂接模块
WH_SHELL	10	外壳挂接模块
WH_FOREGROUNDIDLE	11	前台空闲窗口挂接模块
WH_CALLWNDPROCRET	12	消息已在窗口进程里进块行处理以后，用于接收这些消息的挂接例程

挂接例程是一种特殊的函数，只要发生了相应种类的事件，系统就会调用那个函数。外壳进程可以接收一个挂接代码（如表 16-23 所示）和两个附加的参数，这两个参数用于对进程的行为进行限制：

LRESULT CALLBACK ShellProc（int nCode，WPARAM wParam，LPARAM lParam）；

表 16-23 里的值与一个常规的外壳有关，并不特别地针对 EXPLORER.EXE。因此，在 MS Windows 95 里，并不是所有外壳代码都可以到达一个外壳挂接例程。

表 16-23　在一个外壳例程里拦截的外壳挂接代码

外壳挂接代码	值	说　明
HSELL_WINDOWCREATED	1	一个新的顶级窗口已经建立起来了
HSELL_WINDOWDESTROYED	2	一个顶级窗口已经消除了
HSELL_ACTIVATESHELLWINDOW	3	外壳应该激活它的主窗口
HSELL_WINDOWACTIVATED	4	一个独立的顶级窗口已经激活了
HSELL_GETMINRECT	5	一个顶级窗口正处于最小化显示状态，并且系统需要那个窗口使用的最小化矩形区域的大小
HSELL_REDRAW	6	任务栏内的窗口标题已经进行了重画
HSELL_TASKMAN	7	用户已经选择了任务列表
HSELL_LANGUAGE	8	已经安装了一种新的键盘语言

在 SHELLOBJ 示范程序里，我们安装了一个外壳挂接模块，以便对 Set shell hook 菜单项提供响应。以后再通过调用 UnhookWindowsHookEx（）函数删除这个模块：

```
...
case MN_SETHOOK:
{
    HHOOK hhook;

    // set a shell hook
    hhook = SetWindowsHookEx(WH_SHELL, ShellProc, hInstance,
GetCurrentThreadId());
    if(! hhook)
        MessageBeep(0);
}
    break;
...
```

下面这个代码段也是从SHELLOBJ示范程序里摘录下来的，它向我们揭示了怎样建立一个挂接例程。假如没有捕获到一个外壳代码，它的返回值就应该是FALSE。wParam和lParam的内容将根据外壳代码的不同而发生变化。

```
LRESULT CALLBACK ShellProc(int nCode, WPARAM wParam, LPARAM
                                    lParam)
{
    switch(nCode)
    {
        case HSHELL_ACTIVATESHELLWINDOW:
            return TRUE;

        case HSHELL_WINDOWACTIVATED:
            return TRUE;

        case HSHELL_TASKMAN:
            return TRUE;

        case HSHELL_WINDOWCREATED:
            return TRUE;

    }
    return FALSE;
}
```

无论挂接模块是如何安装的，在应用程序中断之前都要记住将其删除，这样可以恢复消息的正确行为和流向。

16.8 外壳对象和定制应用程序

SHELLOBJ 和 EXETYPE 示范程序都实现了 Windows 95 里一些新的外壳接口。表 16-24 为大家列出了这两个程序里用到的外壳接口，以及对外壳进行扩展开发时需要用到的接口。

表 16-24 外壳接口的完整集合，其中包括外壳扩展接口

外壳接口	说 明
IShellFolder	外壳接口
IEnumIDList	这个接口用于列举文件夹内的项目
IContextMenu	这个接口用于管理与某个项目对应的关联菜单
ICopyHook	这个接口用于对外壳项目的直接维护进行控制
IExtractIcon	用于取得文件对象所用图标的外壳接口
IMalloc	用于对内存分配进行管理的接口
IShellPropSheetExt	用于增加或者替换属性表的卡片的接口
IShellExtInit	这个接口用于初始化属性表扩展、关联菜单扩展或者拖放控制模块
IShellLink	用于建立和分析外壳链接的接口

遗憾的是，微软公司现在还没有提供能对 Start 弹出式菜单进行维护的接口。Start 菜单是一种标准的组件，它只能通过某些系统策略进行修改和控制。我们期望 Windows 的下一个版本能提供更多的接口和成员函数，以便对这方面的问题进行控制。

在本章的另外三个示范程序里——STARTMENU，CLSID 和 BROWSER，我们提供了一些很有价值的提示和建议，它们有助于对系统外壳的接口进行扩展。

16.8.1 任务栏通知区域

任务栏最有价值的特性之一就是它能在通知区域里显示几个小图标，这个通知区域位于任务栏的右侧。在我自己的系统里，数字时钟左侧唯一显示的就是一个用于代表音量控制的图标。这只是任务栏能显示的十一个标准图标中的一个。这些图标的完整列表请大家参考表 16-25。

表 16-25 可以在任务栏通知区域内显示的预定义对象

图标	对象
Battery Meter（电池尺）	电池电量级别
HP Jet Direct Printer	HP JetAdmin 工具软件
PC Card Status（PC 卡状态）	PC 卡控制器
Printer	常规打印机
Fax Rendering（传真报告）	传真
Fax Status（传真状态）	传真
Keybord Layout（键盘布局）	键盘基本语言
Modem（调制解调器）	调制解调器状态
Microsoft Network（MSN）	Microsoft Network 主页
Send/Receive Mail（邮件收发）	MS Exchange
Volume Control（音量控制）	声音卡

在某些情况下，特别是一台笔记本计算机里，通知区域将同时显示出几个图标。除了标准控件以外，一个应用程序还可以增加定制图标，从而实现附加的功能。这也就是我们准备用 STARTMENU 示范程序完成的工作。启动这个程序以后，它会在通知区域内的数字时钟旁边放置一个小图标，用它代表美国本土的 48 个州。如果用鼠标左键单击这个图标，就会在屏幕上显示出一个叠置式窗口，如图 16-32 所示。

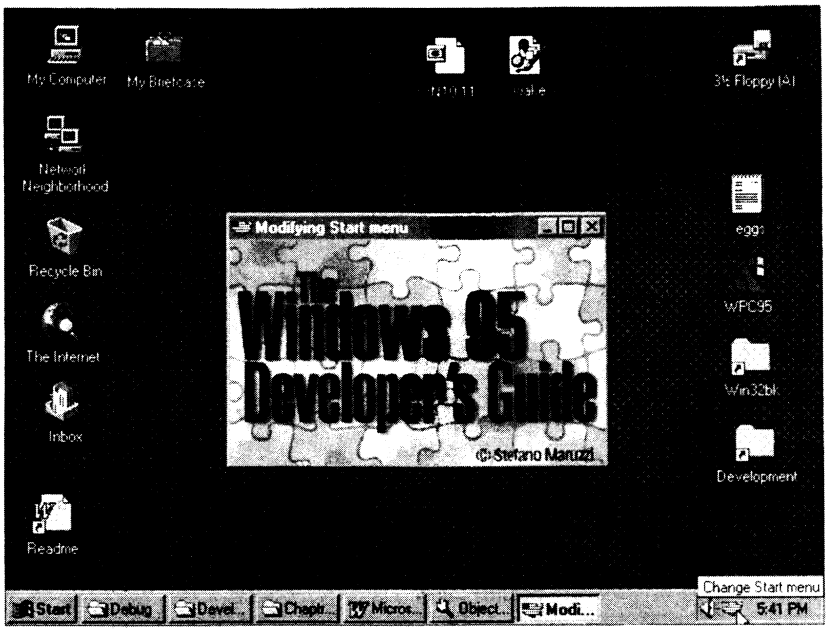


图 16-32　STARTMENU 的主窗口显示了一幅位图

对于这个窗口来说，我们在它上面没有什么可做的。该窗口的主要用途是提供一些视觉上的反馈效果，以及证明两个鼠标按钮（左键和右键）在任务栏通知区域内某个图标上的单击都能被捕获到。更有趣的，假如在那个美国地图图标上面单击鼠标右键，一个小的弹出式菜单就会显示出来，其中列出了五个菜单项，如图 16-33 所示。

图 16-33　STARTMENU 程序允许用户在 Start 菜单的菜单集合里增加或者删除单一的项目或者整个菜单组

假如选择 Add group 菜单项，标准的 Browse for Folders 对话框就会显示出来，同时从与 SHELLOBJ 示范程序里看到的不同的一个位置处开始对外壳命名空间进行检查。在这个时候，应用程序将从 Program 项目处开始检查命名空间。Programs 项目是位于 Start 菜单对象下方的第一个项目，它位于 Start 按钮使用的整个菜单系统的根部，如图 16-34 所示。在这以

后，一个定制对话框将显示出来，它要求用户为新组别分配一个名字。假如所有操作都正确地完成了，Start 按钮的次级菜单就会提供如图 16-35 所示的一个附加组别。

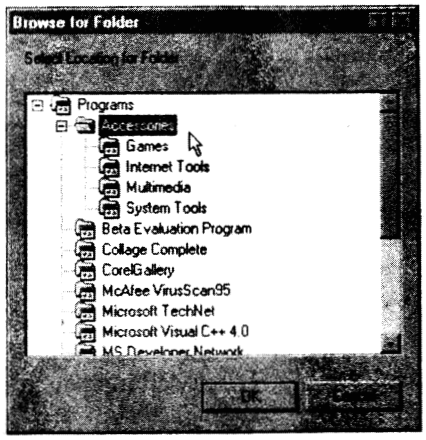


图 16-34　在 STARTMENU 程序里，用户可以选择一个位置，用于增加一个新组别

图 16-35　ANTONELLA 组别现在已经增添到菜单组的原始集合里了

根据相同的逻辑，用户可以增加或者删除单一的项目，甚至可以删除整个菜单组别。对所有这些操作的管理都是在 STARTMENU 源代码里进行的，它们主要依赖于本章前面介绍过的一些 SH API 函数。每个组实际上都是一个系统目录，所有目录都建立于 Start Menu 文件夹下方，CreateDirectory（）这个 Win32 API 可以帮助我们实现目录的创建。单一的菜单

项是针对执行模块的一些快捷，这些快捷则是通过我们以前讲述过的 CreateLink () 例程建立的。

STARTMENU 实现的新特性是与任务栏的通知区域进行的交互作用。

```
#include <shellapi.h>
BOOL WINAPI Shell_NotifyIcon(DWORD dwMessage, PNOTIFYICONDATA
                             pnid);
```

参数	说明
DWORD dwMessage	表 16-26 列出的其中一条 NIM_消息
PNOTIFYICONDATA pnid	一个 NOTIFYICONDATA 数据结构的地址
返回值	在正文里讨论
BOOL	假如函数调用成功，就返回一个 TRUE 值；假如失败，则返回一个 FALSE 值

这个函数的所有逻辑都要以 NOTIFYICONDATA 数据结构为基础，那种结构的构造情况如下所示：

```
typedef struct _NOTIFYICONDATA
{
    DWORD cbSize;
    HWND hWnd;
    UINT Uid;
    UINT uFlags;
    UINT uCallbackMessage;
    HICON hIcon;
    WCHAR szTip[64];
} NOTIFYICONDATA, *PNOTIFYICONDATA;
```

下面讲述整体的运行机制。应用程序在任务栏通知区域放置一个图标的同时就定义了一条定制回调消息。不管什么时候，只要用户单击了那个图标，应用程序（或者在 NOTIFYICONDATA 结构里指定的窗口）就会接收到定制消息，并且判断用户对那个图标实际进行了什么操作。从这个时候开始，应用程序就会继续它的常规执行流程。在中断以前，图标将从通知区域删除，并且恢复通知区域以前的状态。

表 16-26 Shell_NotifyIcon () API 使用的消息

Shell_NotifyIcon () 消息	值	说 明
NIM_ADD	0x00000000	在任务栏通知区域插入一个图标
NIM_MODIFY	0x00000001	从任务栏通知区域删除一个图标
NIM_DELETE	0x00000002	对任务栏通知区域的图标进行修改

这样一来，NOTIFYICONDATA 的 hWnd 项就显得相当重要了，因为它间接性地定义了一个代码段。每次当用户单击图标的时候，那个代码段就会激活。这就解释了在通知区域放置对象的一个应用程序需要一个主窗口。图标是用 Uid 项和图标句柄（hIcon）定义的，Uid 内包含了一个唯一性的 ID。uFlags 项可用于判断其余三个项内是否包含了有效的信息。表 16-27 内列出的三个 NIF_标志分别定义了那三个设置项。

表 16-27　NOTIFYICONDATA 数据结构使用的标志

NOTIFYICONDATA 标志	值	说　明
NIF_MESSAGE	0x00000001	uCallbackMessage 项内包含了有效的信息
NIF_ICON	0x00000002	hIcon 项内包含了有效的信息
NIF_TIP	0x00000004	szTip 项内包含了有效的信息

在通知区域插入一个图标的时候，NIM_ADD 消息就成了 Shell_NotifyIcon () 的第一个参数，同时 NOTIFYICONDATA 结构的 uFlags 项内将设置好全部三条 NIF_标志。uCallbackMessage 项内包含了由应用程序定义的一条消息，图标将利用这条消息把信息传回由 hWnd 项指定的窗口。在 STARTMENU 程序里，我们按照下述的格式建立了 WM_TRAYNOTIFY 消息：

```
#define WM_TRAYNOTIFY WM_USER + 100
while the icon ID is set to 17:
#define TRAY_ICON117
```

利用前两条定义，应用程序把 WM_TRAYNOTIFY 消息指定成定制消息，不管什么时候，只要用户与通知区域内的图标进行交互作用，就会发出这条消息，那个图标的 ID 是 17。

```
...
// place the icon on the Taskbar
tnd.cbSize = sizeof(NOTIFYICONDATA);
tnd.hWnd = hwnd;
tnd.uID = TRAY_ICON1;

tnd.uFlags = NIF_MESSAGE | NIF_ICON | NIF_TIP;
tnd.uCallbackMessage = WM_TRAYNOTIFY;
tnd.hIcon = LoadIcon(hInstance, "start");
strcpy(tnd.szTip, "Change Start menu");

Shell_NotifyIcon(NIM_ADD, &tnd);
...
```

从理论上说，Shell_NotifyIcon () 和 NOTIFYICONDATA 数据结构的语法可以让我们

在任务栏通知区域内建立和定位数量无限的图标。然而，根据我们的经验，图标的总数应该限制在几个以内，数量众多的图标会使用户产生混淆。

用户在通知区域内单击某个应用程序图标以后，WM_TRAYNOTIFY 定制消息就会到达主窗口的窗口进程。这条消息的 wParam 内包含了图标 ID (STARTMENU 里设置为 17)，而 lParam 内则填写了一条 WM_消息，用于对触发这条消息的行动进行说明。在下面的代码段里，我们没有对 wParam 进行处理，因为应用程序在通知区域内只放置了一个图标。假如用户用鼠标单击那个图标，就会显示出相应的弹出式菜单。

```
...
case WM_TRAYNOTIFY:
    switch(lParam)
    {
        case WM_RBUTTONDOWN:
        {
            POINT pt;
            int iCmd;

            // get the mouse location on the screen
            GetCursorPos(&pt);
            // setFocus(hwnd);
            // SetActiveWindow(hwnd);
            iCmd = TrackPopupMenu(hpopup,
                    TPM_BOTTOMALIGN | TPM_LEFTBUTTON |TPM_
                    RETURNCMD,
                    pt.x, SYS(SM_CYSCREEN), 0, hwnd, NULL);

            // simulate a WM_COMMAND
            if(iCmd)
                SendMessage(hwnd, WM_COMMAND, MAKEWPARAM
                        (iCmd, 0), 0);
}
    break;
...
```

应用程序中断之前，图标将从通知区域删除，这种操作可以通过再次调用 Shell_NotifyIcon () 来实现，只需传递一条 NIM_DELETE 消息就可以了：

```
...
tnd.cbSize = sizeof(NOTIFYICONDATA);
```

```
tnd.hWnd = hwnd;
tnd.uID = TRAY_ICON1;

tnd.uFlags = NIF_MESSAGE|NIF_ICON;
tnd.uCallbackMessage = WM_TRAYNOTIFY;
tnd.hIcon = LoadIcon(hInstance, "start");
*tnd.szTip = '\0';

// remove the icon
Shell_NotifyIcon(NIM_DELETE, &tnd);
...
```

假如应用程序需要对当前显示的图标进行修改，就可以调用 Shell_NotifyIcon ()，同时传递 NIM_MODIFY 消息。膝上型计算机系统的电源采用的就是这种原理。

16.8.2 深入探索“类”

在注册表数据库里，大家是否注意到了 CLSID 子键下面列出了多少类呢？这个列表的长度几乎是无限的，并且坦白地说，通过对 128 位的 ID 进行检查并不足以对类进行判断。

在本书附带 CD 的 Listing 16.4 里，大家可以找到一个名为 CLSID 的示范程序，它用一种不同的形式向我们提供了系统内注册的所有类的一份列表。File 菜单内的 Browse 菜单项可以强迫应用程序通过 Reg () 这个 API 访问注册表数据库，取得它们的名字，并在覆盖于整个客户区上方的列表视窗内显示出这些类，如图 16-36 所示。

尽管在列表视窗控件的顶部仍然存在传统的标题按钮，但是列出的信息并不能进行排序处理，因为每个列表视窗项目内的 lParam 在这个示范程序里都扮演了一种不同的角色（CD-ROM 提供的另外一个例子阐述了在列表视窗内的排序技术）。第一栏内的数字只是一种简单的连续性 ID，通过它只能了解注册表内定义了多少个类。考虑类的数量和类型会随着系统的不同发生变化，所以假如 CLSID 在你的系统上得到了一个不同的输出结果，不应该对此感到意外。

只要单击左栏内列出的一个数字，一个次级窗口就会显示出来，该窗口位于屏幕的左上角，如图 16-37 所示。例如，Recycle Bin 在 DefaultIcon 条目中提供了三个按钮。第一个是缺省按钮，第二个代表 Recycle Bin 置空时的样子，第三个则代表其中堆满了废纸时的样子。

DefaultIcon 子键内的每个条目都使用了完整的路径名，后面接一个逗号以及一个数字，哪个数字用于指定针对哪个对象显示哪个图标。例如，一个有效的定义可能是：C:\WINDOWS\SYSTEM|SHELL32.DLL，31。缺省的空回收站使用的是 SHELL32.DLL 的资源段内的第 32 个图标。CLSID 程序将从每一类键集合内读取信息，然后从源模块内把图标提取出来。这是程序不断调用 Reg () API、ExtractIcon () 以及与弹出式列表视窗进行交互作用的结果。假如在弹出式列表视窗内的任何一个图标上方单击鼠标右键，就会显示出一个关联菜单。其中的两个菜单项允许用户改变图标的图象，或者恢复原始图象。第二个菜单项最初是屏蔽起来的，这样就限制用户刚开始的时候只能对缺省图标进行替换。

在本章早些时候，我们曾经讨论了一个成员函数，利用它可以通过对外壳图象列表的直

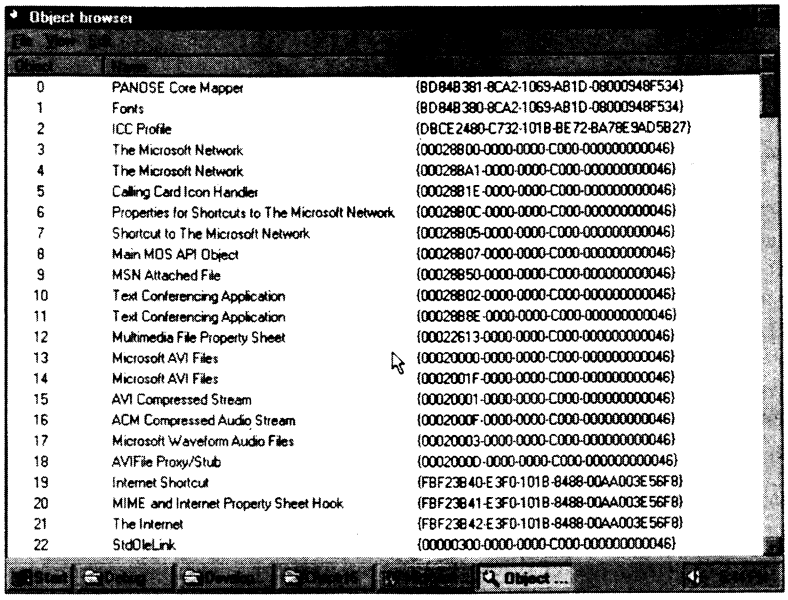


图 16-36　CLSID 示范程序列出了注册表数据库内定义的所有类

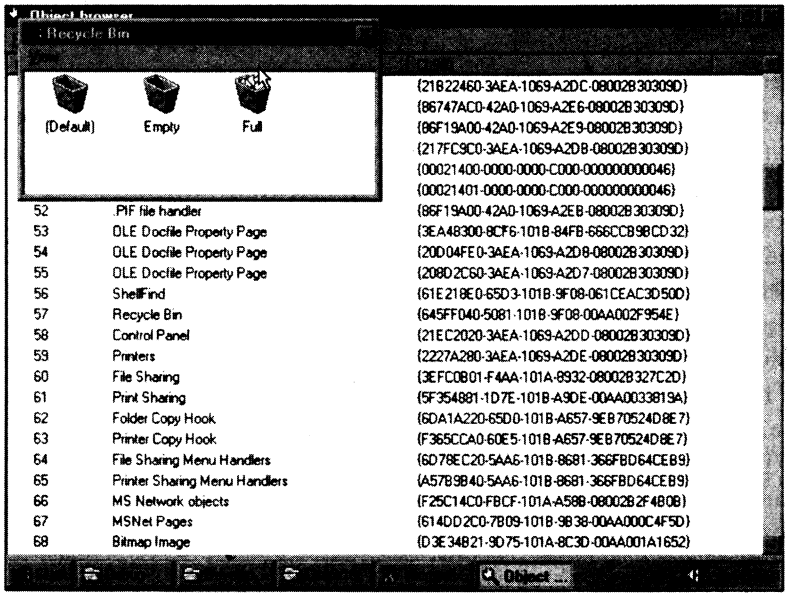


图 16-37　假如一个类拥有关联在一起的一个或者多个图标，
这些图标就会在一个次级窗口里显示出来

接维护，从而改变外壳桌面图标。这种技术能够获得很好的效果，并且简便、有效。然而，它也有自己的缺点。改变的图标在下一次系统启动以后仍然会恢复成缺省图象。这种行为并不让我们感到意外，因为系统图象列表是在最初引导结束以后 CLSID 子键内所有 DefaultIcon

条目的一个“瞬态图”。

动态修改并不能在注册表数据库内反映出来，所以作出的修改只是暂时的。然而，从另一方面来说，CLSID 示范程序却能实现我们的要求。利用这个工具，我们可以直接和永久性地修改与特定对象类关联在一起的图标，把改动后的结果存储到注册表数据库里。为了实现这个目的，我们需要在注册表的 HKEY_CURRENT_USER 根键下面建立一个新的树形分支。在标准的 Software 子键下方，CLSID 程序会建立 ObjBrowser 键，并在更深的一级放置 CLSID 键，亦即：Software\ObjBrowser\CLSID。在 CLSID 键下方，应用程序将存储 128 位长的类 ID 以及一个新的 DefaultIcon 键，并在其中存储与那个对象类关联在一起的原始图标的信息。如图 16-38 所示。

CLSID 的源代码相当长，大约有 1800 行。请大家仔细地研究它，把它当作建立一个更复杂的应用程序的起点，从而实现注册表数据库更完美的处理。

16.8.3 更多的浏览

本书附带 CD-ROM 的 Listing 16.5 内提供了一个名为 BROWSER 的示范程序，利用它可以对 Windows 95 外壳进行更加深入的分析。利用这个超过 1200 行的代码段，我们将完成外壳对象在一个定制应用程序里最后的封装处理。整个外壳命名空间的分级结构都是用一个列表视窗控件代表的。为什么要用列表视窗呢，这一着是从 EXPLORER.EXE 那里学来的，列表视窗显然是表示分级结构项目的最佳工具。节点既可以展开，也可以折叠。假如在一个项目名称上方单击鼠标左键，就会在一个连接起来的列表视窗内填写选中文件夹内含的对象。图 16-39 向大家展示了选中 Network Neighborhood 项目时，应用程序的显示情况。

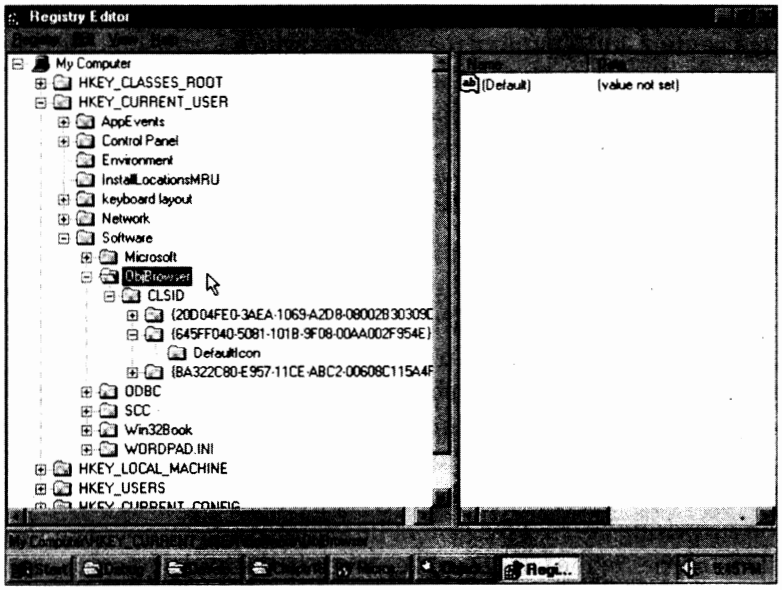


图 16-38 被修改的每一个类的原始图标都在 ObjBrowser 键下面进行了备份

这个程序其实还应该对某些虚拟文件夹进行特殊的考虑。这些文件夹包括控制面板以及打印机等等，它们都要求对关联起来的图标进行特殊的考虑。具体的编程工作留待读者自己去完成。

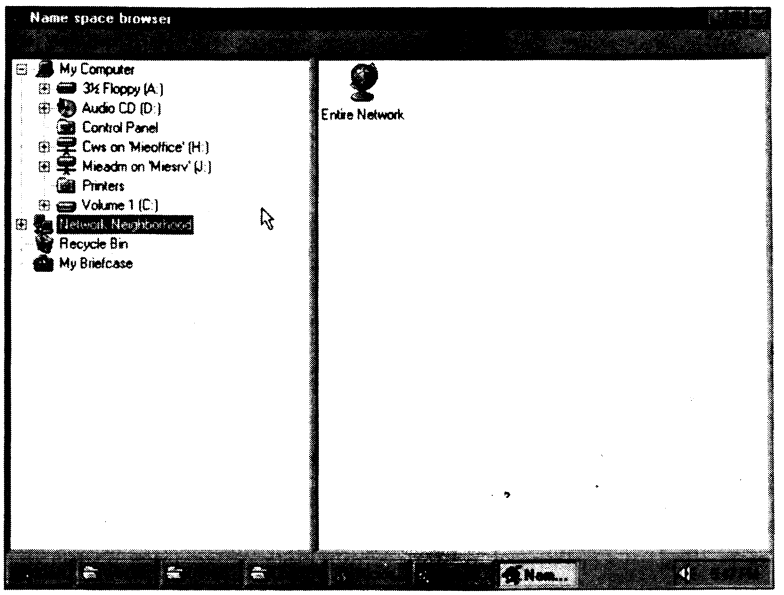


图 16-39　BROWSER 示范程序再次展示了怎样把外壳对象合并到一个定制应用程序里

16.9　应用程序栏

接下来，我们准备讨论一个新主题，亦即从技术角度称呼的应用程序桌面工具栏，简称为“应用程序栏”(appbar)。在这儿，应用程序栏是一种特殊的窗口，它的行为类似于系统任务栏。MS Office for Windows 95 为我们提供了这样的一个应用程序栏，它是最初在 Windows 3.x 内实现的 Microsoft Office Manager 工具的一个改进版本。

“应用程序栏”窗口可以沿着任何一个屏幕边界摆放，这种窗口没有提供传统的标题栏，所以用户不能利用标题栏把窗口拖动到一个定制的位置处。这样一来，为了给应用程序栏分配一个新位置，用户既可以在选择一个空白地点以后拖动它，也可以通过发出一条特定的命令来实现。本书附带 CD 的 Listing 1.6 内提供了一个名为 TASKBAR 的示范程序，它向大家揭示了怎样建立一个定制的应用程序桌面工具栏。

系统将从整体的屏幕显示空间内减去由一个应用程序栏占据的区域。整体显示空间减去由任务栏以及活动的应用程序栏占据的空间以后，最后得到的就是工作区。这个区域是一个最大化顶级窗口最多能够占用的一个矩形表面。在缺省情况下，系统会通过 WM_GETMINMAXINFO 消息把这种信息传递给每个窗口，我们毋需在应用程序源代码内进行任何干涉。正如大家自己就可以测试的那样，如果对任务栏进行伸缩处理，一个最大化显示的窗口就会自动调整自己的新显示矩形。屏幕上放置的应用程序栏数量是没有什么限制的——尽管屏幕的边界一共只有四个。事件上，只要自己愿意，完全可以在某个屏幕边界放置两个应用程序栏。

根据我们的经验，应该严格限制每个单一进程使用的应用程序栏数量。否则，用户就会碰到很大的麻烦。应用程序栏应该帮助用户通过简便和直接的方式完成特定的任务。但是假

如同时存在多个应用程序栏，就会得到相反的效果。

应用程序栏的建立

考虑到应用程序栏扮演的特殊角色，系统本身已经针对它提供了一些帮助和信息，利用它们可以对这种对象进行管理。因此，当我们建立一个应用程序栏时，第一个步骤就是通过系统对其进行注册。新的应用程序栏信息会增添到一个内部列表里，对这个列表的管理是由外壳实现的。在应用程序里，我们可以通过一些特殊的应用程序栏消息，从而间接性地对这种对象进行访问。表 16-28 为大家列出了 MS Windows 95 支持的所有应用程序栏消息。

表 16-28 通过 SHAppBarMessage（）发送的应用程序栏消息

应用程序栏消息	值	说 明
ABM_NEW	0x00000000	注册一个新应用程序栏，并指定相应的消息标识符，系统将利用这个标识符向应用程序栏发送通知消息
ABM_REMOVE	0x00000001	撤消对一个应用程序栏的注册，将应用程序栏从系统的内部列表内删除
ABM_QUERYPOS	0x00000002	请求获得应用程序栏的大小和屏幕位置
ABM_SETPOS	0x00000003	设置应用程序的大小和屏幕位置
ABM_GETSTATE	0x00000004	取得 Windows 任务栏的“自动隐藏”和“总是位于最前面”状态
ABM_GETTASKBARPOS	0x00000005	取得 Windows 任务栏的限制矩形
ABM_ACTIVATE	0x00000006	通知系统：一个应用程序栏已被激活了
ABM_GETAUTOHIDEBAR	0x00000007	取得一个自动隐藏应用程序栏的句柄，这个应用程序栏与屏幕的某个边界是关联在一起的
ABM_SETAUTOHIDEBAR	0x00000008	针对屏幕的某个边界，注册或者撤消注册一个自动隐藏应用程序栏
ABM_WINDOWPOSCHANGED	0x00000009	通知系统：应用程序的位置已发生了变化

为了发出任何一条这样的消息，我们都必须使用 API 函数 SHAppBarMessage（），它的语法结构从概念上说是与 Shell_NotifyIcon（）完全一致的，如下所示：

```
#include <shellapi.h>
UINT APIENTRY SHAppBarMessage(DWORD dwMessage, PAPPBARDATA pabd);
```

参数	说明
DWORD dwMessage	应用程序栏消息(参考表 16-28)
PAPPBARDATA pabd	APPBARDATA 数据结构的地址
返回值	在正文里讨论
UINT	返回值取决于发出的消息

事实上，第一个参数是一条消息，接下来是一个特殊的数据结构——APPBARDATA 的地址：

```
typedef struct _AppBarData
{
    DWORD cbSize;
    HWND hWnd;
    UINT uCallbackMessage;
    UINT uEdge;
    RECT rc;
} APPBARDATA, *PARRBARDATA
```

应用程序的表现与任务栏通知区域很类似。它们都需要通过一条定制消息与管理自己的窗口进行通信，这种消息是在初始化阶段从应用程序传递到外壳对象内的。从那个时候开始，只要有必要与管理自己的窗口进行通信，应用程序栏（或者任务栏的通知区域）就会发出这条消息。同样地，通知区域图标运用的整体运作机制亦可适用于 APPBARDATA 数据结构。除了这个结构的长度以外，建立应用程序栏的程序还必须定义一条回调消息，系统可以利用这条消息把信息传回目标窗口进程。uEdge 项是一个数值定义（参考表 16-29），它与放置应用程序栏的那个屏幕边界是对应的。

表 16-29　与四个屏幕边界对应的定义

屏幕边界	值	说　明
ABE_LEFT	0	左边框
ABE_TOP	1	顶部边框
ABE_RIGHT	2	右边框
ABE_BOTTOM	3	底部边框

类型为 RECT 的那个项定义了屏幕的一个矩形部分，应用程序将放置于这个矩形内。显然，为了计算出最终的坐标，起点应该位于整个屏幕空间的左上角。APPBARDATA 的最后一个设置项是一个四字节的内存区域，利用它可以传递与特定 ABM_消息有关的附加信息。

为了注册一个新应用程序栏，应用程序将发出 ABM_NEW 消息，其中的 WM_APPBAR 是由应用程序定义的一条消息，如下所示：

```
...
APPBARDATA abd;

abd.cbSize = sizeof(APPBARDATA);
abd.hWnd = hwnd;
abd.lParam = 0;

SHAppBarMessage(ABM_ACTIVATE, &abd);
...
```

为了对应用程序栏进行定位，应用程序首先必须发出一条 ABM_QUERYPOS 消息，用它指出一个屏幕边界以及它应该占据的目标矩形。这条消息返回以后，APPBARDATA 的 rc 项内就包含了应用程序栏在屏幕上的位置和尺寸，这些数据已经根据屏幕的当前状态进行了调整。举个例子来说，假如选中的屏幕边界已经由任务栏占据了，系统会确保任务栏总是最外部的一个对象。通过 ABM_SETPOS，应用程序可以定义一个应用程序栏最终的位置和尺寸。这样一来，SHAppBarMessage () 函数返回以后，rc 设置项也许包含了经过修改的值，从而反映出应用程序栏在屏幕上的最佳位置。实际的定位可以通过调用 MoveWindow () 或者 SetWindowPos () 函数实现。利用这两个函数，就可以实现应用程序栏在屏幕上的拖动，同时利用由外壳提供的某些帮助，从而判断最合适的位置。

应用程序栏改变自己的位置或者被激活以后，它就明显需要与系统进行交互作用。作为一个矩形的窗口，应用程序栏可以在重定位操作结束以后收到 WM_WINDOWPOSCHANGED 消息。为了对这条消息提供响应，系统应该接收 WM_WINDOWPOSCHANGED 消息，以保证应用程序栏在 Z 轴上的正确位置。相同的调整也适用于 WM_ACTIVATE 消息，为了对这条消息进行处理，必须发出一条 ABM_ACTIVATE。

“自动隐藏”是一种很有趣的特性，它的含义是指只有鼠标指针定位于以前由应用程序栏占据的屏幕边界的时候，这个应用程序栏才会显示出来。Windows 任务栏本身就支持这种特性，为了激活它，用户可以选择 Settings 菜单项，在其中作具体的设置。通过把 APPBARDATA 数据结构的 lParam 分别设置成 TRUE 或者 FALSE，ABM_SETAUTOHIDEBAR 消息就可以实现对一个应用程序栏“自动隐藏”特性的注册或者撤消注册；假如为 TRUE，意味着它是一个自动隐藏应用程序栏；假如为 FALSE，就表示这一特性没有实现。

在其他消息中间，ABM_GETTASKBARPOS 有些时候是相当有用的，它甚至能在其他场合下使用——判断系统任务栏在屏幕上的位置和尺寸。

任务栏消息可以向系统报告某些特定事件的发生，以及指示应用程序栏执行某种特定的任务。为了向自己的控制器（注册时在 hWnd 项里指定的那个窗口）发出通知信息，应用程序栏必须发出自己注册时事先在 uCallbackMessage 项里指定好的消息，同时在 wParam 里存储一个通知代码（请参考表 16-30）。

表 16-30　APPBAR 通知代码

APPBAR 通知代码	值	说　明
ABN_STATECHANGE	0x0000000	“自动隐藏”或者“总是位于最顶部”状态已经发生了变化
ABN_POSCHANGED	0x0000001	应用程序栏的大小或者位置已经改变了
ABN_FULLSCREENAPP	0x0000002	准备打开或者关闭一个全屏幕应用程序
ABN_WINDOWARRANGE	0x0000003	已经从任务栏的关联菜单内选择了 Cascade，Horizontally 或者 Tile Vertically 命令

图 16-40 向大家展示了应用程序栏沿着屏幕右边界摆放时，TASKBAR 示范程序的运行情况。MS Word 7.0 处于最大化显示状态，然而它的整体尺寸已经减小了，这是考虑到要为应用程序的桌面工具栏留下显示空间。

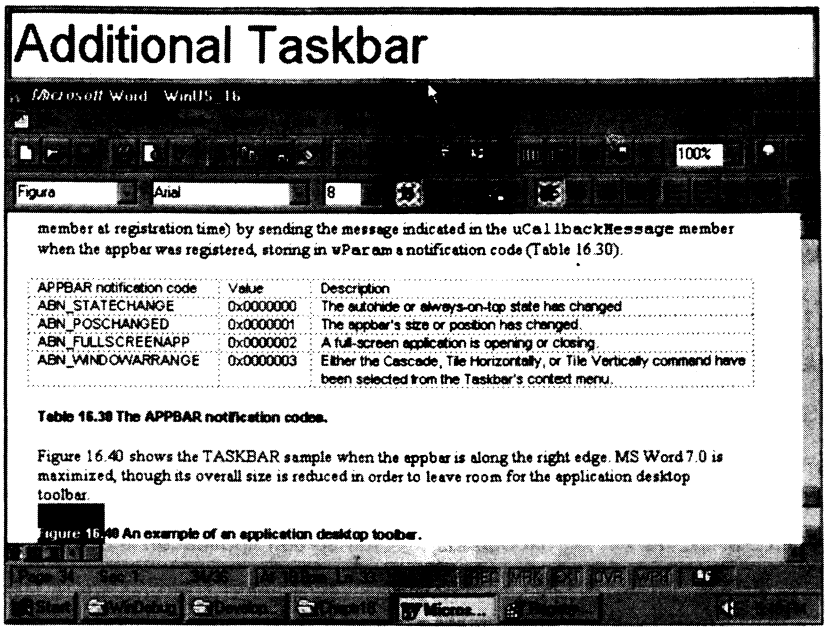


图 16-40　一个应用程序桌面工具栏的例子

16.10　拖动至外壳

在 Win32 应用程序里，最有趣的特性之一就是我们能通过一种简单的直接维护方式把数据传输到外壳上面。对于一个面向对象的外壳来说，拖放机制是它的核心。在 Windows 95 里，移动项目、删除文件或者拷贝文件夹都可以通过指点设备来完成。在第 11 章“图形设备接口示例”里，我们阐述了把信息从外壳拖到一个定制应用程序里时需要用到的技术。现在我们将深入这种讨论。事实上，为了完成类似的任务，使用 OLE 的拖放规范是唯一可行的方法。在本书附带 CD 的 Listing 16.7 里，大家可以找到一个名为 SHELLDRAG 的示范程序，它向我们揭示了怎样利用系统外壳提供的 OLE 支持，从而实现拖放功能，如图 16-41 所示。

图 16-41　SHELLDRAG 示范程序可以通过拖放协议把数据传输到外壳上

拖放操作涉及到两个对象：一个源对象以及一个目标对象。源对象内包含了准备拖动的数据，而目标对象则用于对拖动的数据进行接收。在 SHELLDRAG 程序里，源对象就是应用

程序代码，而目标对象则是外壳本身。在这种情况下，很重要的一点要求是源对象必须根据相应的协议对所有信息进行正确的传递，以便让目标对象最终获得正确的拖动对象。为了实现正确的拖放操作，源对象和目标对象都应该进行正确的处理。对于源对象来说，它的任务主要是下面这些：

▶准备需要传输的数据。
▶访问 IDropSource 接口。
▶调用 DoDragDrop（），执行实际的拖动操作。
▶通过 IDropSource：：GiveFeedback（）对鼠标指针的形状进行相应的改动，从而向用户提供正确的反馈信息。

另外一方面，目标对象的响应则显得稍微复杂一些：
▶访问 IDropTarget 接口。
▶通过调用 RegisterDragDrop（）函数，从而把 IDropTarget 对象与一个窗口关联起来。
▶当鼠标指针覆盖了目标窗口的一个显示象素时，就对由 IDropTarget：DragEnter（）和 IDropTaget：：DragOver（）函数负载的信息进行有效性检查。
▶通过 IDropTarget：：Drop（）成员函数，从而接收拖动的对象。

在拖放操作中，源对象与目标对象之间的一座桥梁是 OLE 函数 DoDragDrop（）。这个函数能从源对象处接收信息，并且可以搜索一个潜在的目标对象——检查某个对象是否有可能对自己负载的信息进行接收。这个阶段在内部是通过一个循环监视机制来实现的，它能对来自鼠标和键盘的所有信息进行检查。事实上，当我们把一个文件从一个文件夹拖动另一个文件夹的时候，鼠标是我们执行这种操作的最基本的工具。相反，键盘的用处并不是很大。通过按下 Ctrl 或者 Shift 键，我们也许可以修改正在进行的行动，将其从一个标准的移动转变成一个快捷的建立或者拷贝。为了实现这种功能，DoDragDrop（）需要调用 IDropSource 接口的两个成员函数，即 QueryContinueDrag 和 GiveFeedback。第一个函数的作用是指出 Esc 键是否已经按下，以及 Ctrl 和 Shift 键的状态，如下所示：

```
#include 〈oleidl.h〉
HRESULT IDropSource:QueryContinueDrag(BOOL fEscapePressed, DWORD
    grfKeyState);
```

假如拖动操作应该继续下去，QueryContinueDrag（）就会返回一个S_OK。相反，假如按下了Esc键，就会返回DRAGDROP_S_CANCEL。拖动操作抵达到了自己的目的地以后，QueryContinueDrag（）就会返回一个DRAGDROP_S_DROP。下面这个代码段是从SHELLDRAG程序里摘录下来的，它向我们展示了怎样用C语言实现QueryContinueDrag（）成员函数：

```
...
HRESULT CDropSource_QueryContinueDrag(IDropSource *pdsrc,
                                      BOOL fEscapePressed,
```

```
                                                DWORD grfKeyState)
    {
        CDropSource * this = IToClass(CDropSource, dsrc, pdsrc);

        if(fEscapePressed)
            return DRAGDROP_S_CANCEL;

        // initialize ourself with the drag begin button
        if(this -> grfInitialKeyState == 0)
| MK_RBUTTON ) grfInitialKeyState = (grfKeyState & (MK_LBUTTON
                                                    MK_MBUTTON));

        if (! (grfKeyState & this -> grfInitialKeyState))
            return DRAGDROP_S_DROP;
        else
        return S_OK;
    }
    ...
```

GiveFeedback()的作用是根据拖动操作当前的状态而提供恰当的视觉反馈信息。正是由于这个函数的作用如此，所以每当前面对 IDropTarget::DragEnter()，IDropTarget::DragOver()和 IDropTarget::Drop()的调用(这些调用的目的是对新的潜在的目标对象进行侦测)而返回 DoDragDrop()的时候，就需要调用到 GiveFeedback()函数。如下所示：

```
#include <oleidl.h>
HRESULT GiveFeedback(DWORD dwEffect);
```

接下来，让我们一起来检查一下 DoDragDrop()函数的语法结构。除了 IDropSource 接口以外，它还需要用到指向 IDataObject 接口的一个指针。这个接口的基本用途是提供数据传输功能和针对数据的改变发出通知。

```
#include <oleidl.h>
HRESULT DoDragDrop(IDataObject * pDataObject,
                   IDropSource * pDropSource,
                   DWORD dwOKEffect,
                   DWORD * pdwEffect);
```

其中，第一个参数是指向 IDataObject 接口的指针，这个指针通常是通过调用 IClassFactory::CreateInstance()成员函数获得的。第三个参数是一个或者多个 DROPEFFECT 定义——用于定义拖动操作(或者拷贝、移动以及链接)的最初状态。最后一个参数用于接收当 DoDragDrop()返回时拖动操作的结果。唯一没有提到的信息是如何对数据封装和传输进行完

整控制，这个主题已经超出了本书的范围。

SHELLDRAG 程序提供了一个 Edit 控件窗口，这个窗口覆盖了程序的整个客户区。在其中键入了一些无意义的单词以后，可以按下鼠标左键不放，然后把它拖动到桌面。在这个时候，与光标关联在一起的图象将发生变化，下方将增加一个小矩形框，框内显示了一个加号。假如此时松开鼠标按钮，拖放操作就结束了，同时把 Edit 窗口内包含的数据传输到外壳桌面一个崭新的对象内，如图 16-42所示。在这个时候，我们可以把这个对象与一条邮件消息联系起来、把它扔进回收站、拷贝或者移动至另外一个文件夹里，就像一个标准的外壳项目那样。

图 16-42　拖动操作结束以后，一个新的正文文档就会在屏幕上的相关位置处建立起来

16.11　总结

Windows 95 外壳本身就是一个崭新的开发环境；新的 32 位应用程序就是在它的基础上建立起来的。外壳不久以后将在 NT 内的引入以及 OLE 开发模式向真正多线程环境的迈进，它们代表了未来两种主要的发展趋势。

通过这本书，大家掌握了 Win32 应用程序开发背后涉及到的所有理论，这些理论是从最基础的知识开始讲述的。为了最终成为一名出色的、有经验的 Windows 开发者，大家还需要长时间的努力；这也许是许多年辛勤努力的结果。随着 32 位计算环境的引入，一个全新的世界在我们面前打开了，Windows 程序的设计已经变得不像以前那么困难。我们需要做的一切就是学习更多的 API、接口、消息、风格、标志、结构以及技术。我希望这本书至少可以帮助大家朝最终的成功迈出可喜的第一步！

附录 A　窗口消息

此附录包含了窗口消息的两份完整列表。第一份列表根据对应值排序，第二份列表则按其名称排序。

A.1　按值排序的窗口消息

以下是按对应值排序的窗口消息的一份完整列表。

消息	对应值
WM_NULL	0x0000
WM_CREATE	0x0001
WM_DESTROY	0x0002
WM_MOVE	0x0003
WM_SIZE	0x0005
WM_ACTIVATE	0x0006
WM_SETFOCUS	0x0007
WM_KILLFOCUS	0x0008
WM_ENABLE	0x000A
WM_SETREDRAW	0x000B
WM_SETTEXT	0x000C
WM_GETTEXT	0x000D
WM_GETTEXTLENGTH	0x000E
WM_PAINT	0x000F
WM_CLOSE	0x0010
WM_QUERYENDSESSION	0x0011
WM_QUIT	0x0012
WM_QUERYOPEN	0x0013
WM_ERASEBKGND	0x0014
WM_SYSCOLORCHANGE	0x0015
WM_ENDSESSION	0x0016
WM_SHOWWINDOW	0x0018
WM_WININICHANGE	0x001A
WM_DEVMODECHANGE	0x001B
WM_ACTIVATEAPP	0x001C
WM_FONTCHANGE	0x001D
WM_TIMECHANGE	0x001E
WM_CANCELMODE	0x001F
WM_SETCURSOR	0x0020
WM_MOUSEACTIVATE	0x0021
WM_CHILDACTIVATE	0x0022
WM_QUEUESYNC	0x0023
WM_GETMINMAXINFO	0x0024
WM_PAINTICON	0x0026

WM_ICONERASEBKGND	0x0027
WM_NEXTDLGCTL	0x0028
WM_SPOOLERSTATUS	0x002A
WM_DRAWITEM	0x002B
WM_MEASUREITEM	0x002C
WM_DELETEITEM	0x002D
WM_VKEYTOITEM	0x002E
WM_CHARTOITEM	0x002F
WM_SETFONT	0x0030
WM_GETFONT	0x0031
WM_SETHOTKEY	0x0032
WM_GETHOTKEY	0x0033
WM_QUERYDRAGICON	0x0037
WM_COMPAREITEM	0x0039
WM_COMPACTING	0x0041
WM_COMMNOTIFY	0x0044
WM_WINDOWPOSCHANGING	0x0046
WM_WINDOWPOSCHANGED	0x0047
WM_POWER	0x0048
WM_COPYDATA	0x004A
WM_CONCELJOURNNAL	0x004B
WM_NOTIFY	0x004E
WM_INPUTLANGCHANGEREQUEST	0x0050
WM_INPUTLANGCHANGE	0x0051
WM_TCARD	0x0052
WM_HELP	0x0053
WM_USERCHANGED	0x0054
WM_NOTIFYFORMAT	0x007A
WM_CONTEXTMENU	0x007B
WM_CONTEXTMENU	0x007B
WM_STYLECHANGING	0x007C
WM_STYLECHANGED	0x007D
WM_DISPLAYCHANGE	0x007E
WM_GETICON	0x007F
WM_SETICON	0x0080
WM_NCCREAT	0x0081
WM_NCDSTROY	0x0082
WM_NCCALCSIZE	0x0083
WM_NCHITTEST	0x0084
WM_NCPAINT	0x0085
WM_NCACTIVATE	0x0086
WM_GETDLGCODE	0x0087
WM_NCMOUSEMOVE	0x00A0
WM_NCLBUTTONDOWN	0x00A1
WM_NCLBUTTONUP	0x00A2

WM_NCLBUTTONDBLCLK	0x00A3
WM_NCRBUTTONDOWN	0x00A4
WM_NCRBUTTONUP	0x00A5
WM_NCRBUTTONDBLCLK	0x00A6
WM_NCMBUTTONDOOWN	0x00A7
WM_NCMBUTTONUP	0x00A8
WM_NCMBUTTONDBLCLK	0x00A9
WM_KEYFIRST	0x0100
WM_KEYDOWN	0x0100
WM_KEYUP	0x0101
WM_CHAR	0x0102
WM_READCHAR	0x0103
WM_SYSKEYDOWN	0x0104
WM_SYSKEYUP	0x0105
WM_SYSCHAR	0x0106
WM_SYSREADCHAR	0x0107
WM_KEYLAST	0x0108
WM_IME_STARTCOMPOSITION	0x0I0D
WM_IME_ENDCOMPOSITION	0x010E
WM_IME_COMPOSITION	0x010F
WM_IME_KEYLAST	0x010F
WM_INITDIALOG	0x0110
WM_COMMON	0x0111
WM_SYSCOMMAND	0x0112
WM_TIMER	0x0113
WM_HSCROLL	0x0114
WM_VSCROLL	0x0115
WM_INITMENU	0x0116
WM_INITMENUPOPUP	0x0117
WM_MENUSELECT	0x011F
WM_MENUCHAR	0x0120
WM_ENTERIDLE	0x0121
WM_CTLCOLORMSGBOX	0x0132
WM_CTLCOLOREDIT	0x0133
WM_CTLCOLORLISTBOX	0x0134
WM_CTLCOLORBTN	0x0135
WM_CTLCOLORDLG	0x0136
WM_CTLCOLORSCROLLBAR	0x0137
WM_CTLCOLORSTATIC	0x0138
WM_MOUSEFIRST	0x0200
WM_MOUSEMOVE	0x0200
WM_LBUTTONDOWN	0x0201
WM_LBUTTONUP	0x0202
WM_LBUTTONDBLCLK	0x0203
WM_RBUTTONDOWN	0x0204

WM_RBUTTONUP	0x0205
WM_RBUTTONDBLCLK	0x0206
WM_MBUTTONDOWN	0x0207
WM_MBUTTONUP	0x0208
WM_MBUTTONDBLCLK	0x0209
WM_MOUSELAST	0x0209
WM_PARENTNOTIFY	0x0210
WM_ENTERMENULOOP	0x0211
WM_EXITMENULOOP	0x0212
WM_NEXTMENU	0x0213
WM_SIZING	0x0214
WM_CAPTRUECHANGED	0x0215
WM_MOVING	0x0216
WM_POWERBROARDCAST	0x0218
WM_DEVICECHANGE	0x0219
WM_MDICREATE	0x0220
WM_MDIDESTROY	0x0221
WM_MDIACTIVATE	0x0222
WM_MDIRESTORE	0x0223
WM_MDINEXT	0x0224
WM_MDIMAXIMIZE	0x0225
WM_MDITILE	0x0226
WM_MDICASCADE	0x0227
WM_MDIICONARRANGE	0x0228
WM_MDIGETACTIVE	0x0229
WM_MDISETMENU	0x0230
WM_ENTERSIZEMOVE	0x0231
WM_EXITSIZEMOVE	0x0232
WM_DROPFILES	0x0233
WM_MDIREFRESHMENU	0x0234
WM_IME_SETCONTEXT	0x0281
WM_IME_NOTIFY	0x0282
WM_IME_CONTROL	0x0283
WM_IME_COMPOSTIONFULL	0x0284
WM_IME_SELECT	0x0285
WM_IME_CHAR	0x0286
WM_IME_KEYDOWN	0x0290
WM_IME_KEYUP	0x0291
WM_CUT	0x0300
WM_COPY	0x0301
WM_PASTE	0x0302
WM_CLEAR	0x0303
WM_UNDO	0x0304
WM_RENDERFORMAT	0x0305
WM_RENDERALLFORMATS	0x0306

WM_DESTROYCLIPBOARD	0x0307
WM_DRAWCLIPBOARD	0x0308
WM_PAINTCLIPBOARD	0x0309
WM_VSCROLLCLIPBOARD	0x030A
WM_SIZECLIPBOARD	0x030B
WM_ASKCBFORMATNAME	0x030C
WM_CHANGECBCHAIN	0x030D
WM_HSCROLLCLIPBOARD	0x030E
WM_QUERYNEWPALETTE	0x030F
WM_PALETTEISCHANGING	0x0310
WM_PALETTEISCHANGED	0x0311
WM_HOTKEY	0x0312
WM_PRINT	0x0317
WM_PRINTCLIENT	0x0318
WM_HANDHELDFIRST	0x0358
WM_HANDHELDLAST	0x035F
WM_AFXFIRST	0x0360
WM_AFXLAST	0x037F
WM_PENWINFIRST	0x0380
WM_PENWINLAST	0x038F
WM_USER	0x0400
WM_APP	0x8000
WM_SETTINGCHANGE	WM_WININICHANGE

A.2 按名称排序的窗口消息

以下是按名称（即按字母顺序）排序窗口消息的一份完整列表。

消息	对应值
WM_ACTIVATE	0x0006
WM_ACTIVATEAPP	0x001C
WM_AFXFIRST	0x0360
WM_AFXLAST	0x037F
WM_APP	0x8000
WM_ASKBFORMATNAME	0x030C
WM_CANCELJOURNAL	0x004B
WM_CANCLEMODE	0x001F
WM_CAPTURECHANGED	0x0215
WM_CHANGECBCHAIN	0x030D
WM_CHAR	0x0102
WM_CHARTOITEM	0x002F
WM_CHILDACTIVATE	0x0022
WM_CLEAR	0x0303
WM_CLOSE	0x0010
WM_COMMAND	0x0111
WM_COMMNOTIFY	0x0044

WM_COMPACTING	0x0041
WM_COMPAREITEM	0x0039
WM_CONTEXTMENU	0x007B
WM_CONTEXTMENU	0x007B
WM_COPY	0x0301
WM_COPYDATA	0x004A
WM_CREATE	0x0001
WM_CTLCOLORBTN	0x0135
WM_CTLCOLORDLG	0x0136
WM_CTLCOLOREDIT	0x0133
WM_CTLCOLORLISTBOX	0x0134
WM_CTLCOLORMSGBOX	0x0132
WM_CTLCOLORSCROLLBAR	0x0137
WM_CTLCOLORSTATIC	0x0138
WM_CUT	0x0300
WM_DEADCHAR	0x0103
WM_DELETEITEM	0x002D
WM_DESTROY	0x0002
WM_DESTROYCLIPBOARD	0x0307
WM_DEVICECHANGE	0x0219
WM_DEVMODECHANGE	0x001B
WM_DISPLAYCHANGE	0x007E
WM_DRAWCLIPBOARD	0x0308
WM_DRAWITEM	0x002B
WM_DROPFILES	0x0233
WM_ENABLE	0x000A
WM_ENDSESSION	0x0016
WM_ENTERIDLE	0x0121
WM_ENTERMENULOOP	0x0211
WM_ENTERSIZEMOVE	0x0231
WM_ERASEBKGND	0x0014
WM_EXITMENULOOP	0x0212
WM_EXITSIZEMOVE	0x0232
WM_FONTCHANGE	0x001D
WM_GETDLGCODE	0x0087
WM_GETFONT	0x0031
WM_GETHOTKEY	0x0033
WM_GETICON	0x007F
WM_GETMINMAXINFO	0x0024
WM_GETTEXT	0x000D
WM_GETTEXTLENGTH	0x000E
WM_HANDHELDFIRST	0x0358
WM_HANDHELDLAST	0x035F
WM_HELP	0x0053
WM_HOTKEY	0x0312

WM_HSCROLL	0x0114
WM_HSCROLLCLIPBOARD	0x030E
WM_ICONERASEBKGND	0x0027
WM_IME_CHAR	0x0286
WM_IME_COMPOSITION	0x010F
WM_IME_COMPOSITIONFULL	0x0284
WM_IME_CONTROL	0x0283
WM_IME_ENDCOMPOSITION	0x010E
WM_IME_KEYDOWN	0x0290
WM_IME_KEYLAST	0x010F
WM_IME_KEYUP	0x0291
WM_IME_NOTIFY	0x0282
WM_IME_SELECT	0x0285
WM_IME_SETCONTEXT	0x0281
WM_IME_STARTCOMPOSITION	0x010D
WM_INITDIALOG	0x0110
WM_INITMENU	0x0116
WM_INITMENUPOPUP	0x0117
WM_INPUTLANGCHANGE	0x0051
WM_INPUTLANGCHANGEREQUEST	0x0050
WM_KEYDOWN	0x0100
WM_KEYFIRST	0x0100
WM_KEYLAST	0x0108
WM_KEYUP	0x0101
WM_KILLFOCUS	0x0008
WM_LBUTTONDBLCLK	0x0203
WM_LBUTTONDOWN	0x0201
WM_LBUTTONUP	0x0202
WM_MBUTTONDBLCLK	0x0209
WM_MBUTTONDOWN	0x0207
WM_MBUTTONUP	0x0208
WM_MDIACTIVATE	0x0222
WM_MDICASCADE	0x0227
WM_MDICREATE	0x0220
WM_MDIDESTROY	0x0221
WM_MDIGETACTIVE	0x0229
WM_MDIICONARRANGE	0x0228
WM_MDIMAXIMIZE	0x0225
WM_MDINEXT	0x0224
WM_MDIREFRESHMENU	0x0234
WM_MDIRESTORE	0x0223
WM_MDISETMENU	0x0230
WM_MDITILE	0x0226
WM_MEASUREITEM	0x002C
WM_MENUCHAR	0x0120

WM_MENUSELECT	0x011F
WM_MOUSEACTIVE	0x0021
WM_MOUSEFITST	0x0200
WM_MOUSELAST	0x0209
WM_MOUSEMOVE	0x0200
WM_MOVE	0x0003
WM_MOVING	0x0216
WM_NCACTIVATE	0x0086
WM_NCCALCSIZE	0x0083
WM_NCCREATE	0x0081
WM_NCDESTROY	0x0082
WM_NCHITTEST	0x0084
WM_NCLBUTTONDBLCLK	0x00A3
WM_NCLBUTTONDOWN	0x00A1
WM_NCLBUTTONUP	0x00A2
WM_NCMBUTTONDBLCLK	0x00A9
WM_NCMBUTTONDOWN	0x00A7
WM_NCMBUTTONUP	0x00A8
WM_NCMOUSEMOVE	0x00A0
WM_NCPAINT	0x0085
WM_NCRBUTTONDBLCLK	0x00A6
WM_NCRBUTTONDOWN	0x00A4
WM_NCRBUTTONUP	0x00A5
WM_NEXTDLGCTL	0x0028
WM_NEXTMENU	0x0213
WM_NOTIFY	0x004E
WM_NOTIFYFORMAT	0x0055
WM_NULL	0x0000
WM_PAINT	0x000F
WM_PAINTCLIPBOARD	0x0309
WM_PAINTICON	0x0026
WM_PALETTECHANGED	0x0311
WM_PALETTECHANGING	0x0310
WM_PARENTNOTIFY	0x0210
WM_PASTE	0x0302
WM_PENWINFIRST	0x0380
WM_PENWINLAST	0x038F
WM_POWER	0x0048
WM_POWERBROADCAST	0x0218
WM_PRINT	0x0317
WM_PRINTCLIENT	0x0318
WM_QUERYDRAGICON	0x0037
WM_QUERYENDSESSION	0x0011
WM_QUERYNEWPALETTE	0x030F
WM_QUERYOPEN	0x0013

WM_QUEUESYNC	0x0023
WM_QUIT	0x0012
WM_RBUTTONDBLCLK	0x0206
WM_RBUTTONDOWN	0x0204
WM_RBUTTONUP	0x0205
WM_RENDERALLFORMATS	0x0306
WM_RENDERFORMAT	0x0305
WM_SETCURSOR	0x0020
WM_SETFOCUS	0x0007
WM_SETFONT	0x0030
WM_SETHOTKEY	0x0032
WM_SETICON	0x0080
WM_SETREDRAW	0x000B
WM_SETTEXT	0x000C
WM_SETTINGCHANGE	WM_WININICHANGE
WM_SHOWWINDOW	0x0018
WM_SIZE	0x0005
WM_SIZECLIPBOARD	0x030B
WM_SIZING	0x0214
WM_SPOOLERSTATUS	0x002A
WM_STYLECHANGED	0x007D
WM_STYLECHANGING	0x007C
WM_SYSCHAR	0x0106
WM_SYSCOLORCHANGE	0x0015
WM_SYSCOMMAND	0x0112
WM_SYSDEADCHAR	0x0107
WM_SYSKEYDOWN	0x0104
WM_SYSKEYUP	0x0105
WM_TCARD	0x0052
WM_TIMECHANGE	0x001E
WM_TIMER	0x0113
WM_UNDO	0x0304
WM_USER	0x0400
WM_USERCHANGED	0x0054
WM_VKEYTOITEM	0x002E
WM_VSCROLL	0x0115
WM_VSCROLLCLIPBOARD	0x030A
WM_WINDOWPOSCHANGED	0x0047
WM_WINDOWPOSCHANGING	0x0046
WM_WININICHANGE	0x001A

附录 B　本书示范程序的安装

随同《Microsoft Windows 95 Developer's Guide》一书，我们附带了一张 CD-ROM 光盘，其中包含了全书 16 章里讲述的所有示范程序的源代码。所有示范程序都是用 Visual C++ 4.0 开发环境开发出来，并且全部用 ANSI C 编写，它们都通过了运行测试。因此，读者可以用任何一个 32 位 C 编译器对它们进行重新编译，前提是这个编译器支持 Win32 软件开发包（SDK）。

CD-ROM 内包含了 16 个文件夹，每章占用一个。另外，每个文件夹都包含了大量子文件夹——每个示范程序占用一个。这样可以方便大家对 CD 的内容进行检查。大家可以在其中找到所有源文件、过渡文件以及由 MS Visual C++ 4.0 开发环境针对每个示范程序生成的最终文件。除此以外，大家还可以找到一个名为 EXESONLY 的独立文件夹，其中列出了单纯的可执行文件。这些可执行文件也是按照刚才提到的那种以章节为基础的分级文件夹结构排列的。由于我们已经在单独的树形结构里提供了所有可执行模块，所以大家可以考虑不把整套软件安装到自己的 PC 上，从而直接在光盘上运行和测试它们。这样省下大量宝贵的硬盘空间。除此以外，UPDATE 文件夹提供了用于 MS Windows 95 的一些工具以及“补丁”程序。如果想了解这些文件的一份完整列表，请大家参考本附录后面的“升级文件”小节。

CD-ROM 中几乎所有示范程序均可在任何 MS Windows 95 或者 MS Windows NT 3.51 系统中运行。然而，有些示范程序在 NT 里无法正常运行，这是由于 NT 缺少 MS Windows 95 系统外壳的缘故。

B.1　运行设置程序

把 CD-ROM 插入 CD 驱动器以后，它将自动启动，这是由于 MS Windows 95 支持的 AutoPlay 特性造成的。这种特性表现在两个方面：首先，My Computer 文件夹里的 CD 驱动器会用一个更好看的图标取代。其次，安装程序——SETUP.EXE 模块——会在屏幕中央自动启动和显示出来。这是一个 Win32 程序，其中包含了将近 3000 行代码。我用 MS Visual C++ 4.0 建立了这个程序，将它特别设计用于把所有示范程序文件从 CD-ROM 传送到与系统连接起来的硬盘驱动器上（可以是本地或者网络驱动器）。

在 MS Windows 95 系统中，AutoPlay 特性在缺省情况下将处于活动状态。通过对 CD-ROM 驱动器的属性表进行检查，我们可以发现 Auto insertion notification（自动插入通知）核选框，撤消了对这个核选框的选择以后，就可以屏蔽这个特性。另外，我们还可以采用更简单的一种方法——关闭 CD 驱动器门的时候按住 Shift 键不放。这种操作可以避免系统运行 AutoPlay 特性。

SETUP.EXE 是一个非常简单的程序。菜单栏内提供了两个顶级菜单：File 和 Help，这两个菜单提供的菜单项都是相当有限的。与 File 关联在一起的下拉式菜单也可以通过在应用程序的客户区内单击鼠标右键来激活。除了 Exit 菜单项以外，还提供了另外两个选项，其中只有 Install 是用正常的颜色显示的。因此，我们的选择范围很有限；要么选择 Install 命令，把示范程序安装到硬盘上面；要么选择退出安装程序。安装好软件以后，如果再次进入 SETUP.EXE，就能发现 Install 和 Uninstall 之间多出了一个选项，同时 Uninstall 也将处于活动状态。

Install 菜单项可以启动一个定制向导，它能引导我们完成整个安装过程。此时，应用程序的主窗口将临时性地消失，从而为向导留下显示空间。在每张向导卡片里，我都尝试把需要提供的信息减少到最低限度。一张初始卡片以后，将显示出一个列表视窗控件，其中列出了当前与系统连接在一起的所有存储设备。Network 按钮可以自动启动 Network Neighborhood（网上邻居）这个外壳对象，以便让用户对网络以及与当前系统连接起来的其他 PC 进行访问。

为了进行完整安装，至少需要 50MB 的空闲空间，其中只有 12MB 用于存储可执行模块。假如选择的一个驱动器不符合这种条件，或者选择了一个只读设备，一个警告图标以及对应的警告消息就会在列表视窗下方显示出来。

在第三张卡片里，设置向导会提示用户输入一些常规的信息，包括计算机名称和用户名，并且在一个只读编辑控件里显示出已经侦测到的 MS Visual C++ 4.0 编译器的文件夹名字。左侧的组合框列出了包含包容文件的所有目录，这些包容文件是由开发环境引用的；所有这些信息都是从注册表数据库里提取出来的。如果系统内没有安装 MS Visual C++，尽管这不会对安装进程产生影响，然而会妨碍 SETUP 自动把 WIN32BK.H 头文件拷贝到能由编译器访问的某个包容文件目录内。

第四张也是最后一张卡片允许用户决定目标文件夹的名字以及安装类型。用户可以选择安装所有源文件和执行文件，也可以决定只安装 EXE 文件。按下 Finish 按钮就会强制 Setup Wizard 开始从光盘上把相应的文件拷贝到目标驱动器上面。

拷贝操作是由外壳完成的，这对于一个设置应用程序来说是很少见的。就我个人来说，我认为一个 Win32 进程应该利用由系统提供的所有服务，其中包括一些很不错的功能，比如拷贝、移动、更名以及删除文件等等。

整个安装过程花不到几分钟。经历的时间主要取决于目标驱动器的类型、处理器的速度以及空闲内存的大小。在最后，SETUP 主窗口会再次显示出来，同时目录窗口也会在屏幕的中央显示出来。

在拷贝过程中，系统将建立和更新注册表数据库 HKEY_LOCAL_MACHINE 键下面的一些条目。这种信息对于示范程序以后的反安装是很有用的。为了从目标驱动器里删除安装好的文件，我们只需简单地再次运行 SETUP.EXE，然后选择 File 下拉式菜单内的 Uninstall 菜单项即可。这种操作可以清除注册表数据库的相关内容，删除以前在桌面上生成的快捷（如果它们还在那里的话），并且把 Start 菜单内增加的程序选项删除掉。

安装进程完整的源代码可以在 CD-ROM 的 BOOKINST 文件夹内找到。

B.2 补充文件

UPDATE 文件夹里包含了一些实用工具以及补丁程序，利用它们可以更新某些系统组件，并且可以为这些组件带来新的特性。迄今为止，微软公司已经发布了多种系统更新程序和补丁程序，它们有些可以修复系统外壳（SHELL32.EXE）的一些小错误，有些则可以改进 Novell NetWare 以及 MS Windows NT 的客户请求器（NWSRVUPD 和 VSERUPD）。

除此以外，MS Internet Explorer 2.0 是 MS Plus 提供的 Internet Explorer 的一个新的 32 位 Internet Explorer 的版本。这个新版本支持一些新的特性，比如表格、刻度线以及影象剪辑等等。

最后，POWERTOY 是一系列工具软件的集合，它们可以通过某些新组件对外壳的功能进行扩展，以便更好地管理自己的资源以及与系统进行交互作用。