```
UINT ThreadFunc (LPVOID pParam)
{
  CRect * pRect = (CRect * ) pParam;
  int nArea = pRect -> Width () * pRect -> Height ();
  return 0;
}
```

但是以下程序会引发断言错误：

```
CWinThread * pThread = AfxBeginThread (ThreadFunc, pDoc);
  .
  .
  .
UINT ThreadFunc (LPVOID pParam)
{
  CDocument * pDoc = pParam;
  pDoc -> UpdateAllViews (NULL);
  return 0;
}
```

即使是一些好像是没问题的函数，如 AfxGetMainWnd，当在应用程序的主线程以外任何地方调用它们时，它们都不会正常工作。

最基本的要求是在调用由其他线程创建的 MFC 对象中的成员函数之前，“您必须理解它们隐含的意思”。要理解隐含意思的唯一方法就是研究 MFC 源程序，去了解特定成员函数的行为。另外要记住：MFC 并不是线程安全的，在本章的以后部分将进一步讨论。因此即使一个成员函数看上去是安全的，也要问问自己如果线程 B 访问一个由线程 A 创建的对象，在访问过程中线程 A 抢先了线程 B 该怎么办。

要把这些函数都挑出来非常困难，只会增加编写多线程应用程序的复杂性。这就是为什么在实际工作中多线程 MFC 应用程序趋向于将大量的用户界面工作交给主线程来执行的原因。如果后台线程想要更新用户界面，它会将消息发送或公布给主线程，让主线程执行更新工作。在本章的样例程序中您会看到这种消息处理过程的应用。

17.1.11　您的第一个多线程应用程序

图 17-1 所示的应用程序说明了在设计和实现多线程应用程序时涉及到的基本原则。Sieve 是一个基于对话框的应用程序，它使用了著名的 Eratosthenes 素数算法来计算 2 和所指定的数之间存在的素数的数量。单击 Start 按钮后开始执行计算，当结果数目出现在窗口中的框中时结束。由于计算素数是资源密集型操作，所以

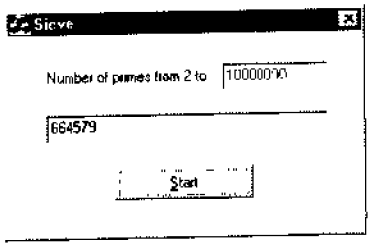

图 17-1　Sieve 窗口

Sieve 在后台线程中执行了大部分计算。(要理解计算素数是资源密集型操作,可以让 Sieve 计算 2 和 100 000 000 之间的素数数量。除非您有巨大的内存,否则就要等一会儿才能得到结果。)如果用主线程来计算素数,Sieve 就会在此过程中冻结输入。但是由于将计算素数的任务委派给了工作者线程,所以无论占用多少时间 Sieve 都会维持对用户输入的响应。

执行计算的线程是由 Start 按钮的 ON_BN_CLICKED 处理程序 OnStart 启动的。图 17-2 中给出了它的源程序。下面是启动线程的语句:

```
THREADPARMS * ptp = new THREADPARMS;
ptp -> nMax = nMax;
ptp -> hWnd = m_hWnd;
AfxBeginThread (ThreadFunc, ptp);
```

OnStart 在名为 THREADPARMS 的应用程序定义的数据结构中将数据传递给了工作者线程。结构中的一个项目是用户在对话框中输入的上限(nMax)。另一个是对话框窗口的句柄。上限值传递给 Sieve 函数执行实际的计算。对话框窗口的句柄用来在工作者线程得到结果后就给应用程序的主窗口公布消息:

```
int nCount = Sieve (nMax);
::PostMessage (hWnd, WM_USER_THREAD_FINISHED, (WPARAM) nCount, 0);
```

在 SieveDlg.h 中,WM_USER_THREAD_FINISHED 是用户定义的消息 ID。主窗口的 WM_USER_THREAD_FINISHED 处理程序从消息的 wParam 中接收结果并在窗口中显示它。

注意,THREADPARMS 结构(通过地址传递给了线程函数)的存储单元是在主线程中分配而在工作者线程中释放的,如下所示:

```
// In the primary thread
THREADPARAMS * ptp = new THREADPARMS;
  .
  .
  .
AfxBeginThread (ThreadFunc, ptp);

// In the worker thread
THREADPARMS * ptp = (THREADPARMS * ) pParam;
  .
  .
  .
delete ptp;
```

为什么要在一个线程中创建结构而在另一个线程中删除呢?因为如果在主线程中的堆栈上创建结构,结构就可能会在其他线程有机会访问它之前超出有效范围而失效。这是一件讨厌的琐碎工作,但是如果没有对它进行恰当的处理,就可能会造成好像是随机发生的错误。

用 new 分配结构确保了越界问题不会发生，也保证了在一个线程中分配内存而在另一个线程中删除它是无害的。使结构作为一个类的数据成员或将其声明为全局有效同样可以确保它不会过早地失效。

在应用程序的主线程结束以后，进程就会结束，并且属于此进程的其他线程也将结束。多线程 SDK 应用程序通常在结束之前并不去终止线程，但是 MFC 应用程序如果没有终止运行着的后台线程就结束的话，会蒙受内存漏损（memory leaks），因为线程的 CWinThread 对象并不会自动删除。由于操作系统几乎会立刻清除它们，所以这种漏损不会很大。但是，如果您不愿意留下任何漏损的机会，那么就应该在应用程序即将结束以前删除现存的 CWinThread 从而避免内存漏损。删除运行的 CWinThread 不会有什么损害，但是要记住您再也不能调用已删除 CWinThread 中的 CWinThread 函数了。

Sieve.h

```
// Sieve.h : main header file for the SIEVE application
//

#if !defined(AFX_SIEVE_H__6DF40C9B_7EA1_11D1_8E53_E4D9F9C00000__INCLUDED_)
#define AFX_SIEVE_H__6DF40C9B_7EA1_11D1_8E53_E4D9F9C00000__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#ifndef __AFXWIN_H__
#error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"       // main symbols
/////////////////////////////////////////////////////////////////////////////
// CSieveApp:
// See Sieve.cpp for the implementation of this class
//

class CSieveApp : public CWinApp
{
public:
  CSieveApp();

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CSieveApp)
  public:
  virtual BOOL InitInstance();
  //}}AFX_VIRTUAL
```

```
// Implementation

  //{{AFX_MSG(CSieveApp)
    // NOTE - the ClassWizard will add and remove member functions here.
    //   DO NOT EDIT what you see in these blocks of generated code !
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations
// immediately before the previous line.
#endif
// !defined(AFX_SIEVE_H__6DF40C9B_7EA1_11D1_8E53_E4D9F9C00000__INCLUDED_)
```

Sieve.cpp

```
// Sieve.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "Sieve.h"
#include "SieveDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CSieveApp

BEGIN_MESSAGE_MAP(CSieveApp, CWinApp)
  //{{AFX_MSG_MAP(CSieveApp)
    // NOTE - the ClassWizard will add and remove mapping macros here.
    //   DO NOT EDIT what you see in these blocks of generated code!
  //}}AFX_MSG
  ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CSieveApp construction

CSieveApp::CSieveApp()
{
```

```
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CSieveApp object

CSieveApp theApp;

/////////////////////////////////////////////////////////////////////////////
// CSieveApp initialization

BOOL CSieveApp::InitInstance()
{
    // Standard initialization
    // If you are not using these features and wish to reduce the size
    // of your final executable, you should remove from the following
    // the specific initialization routines you do not need.

    CSieveDlg dlg;
    m_pMainWnd = &dlg;
    int nResponse = dlg.DoModal();
    if (nResponse == IDOK)
    {
        // TODO: Place code here to handle when the dialog is
        // dismissed with OK
    }
    else if (nResponse == IDCANCEL)
    {
        // TODO: Place code here to handle when the dialog is
        // dismissed with Cancel
    }

    // Since the dialog has been closed, return FALSE so that we exit the
    // application, rather than start the application's message pump.
    return FALSE;
}
```

SieveDlg.h

```
// SieveDlg.h : header file
//

#if !defined(
    AFX_SIEVEDLG_H__6DF40C9D_7EA1_11D1_8E53_E4D9F9C00000__INCLUDED_)
#define AFX_SIEVEDLG_H__6DF40C9D_7EA1_11D1_8E53_E4D9F9C00000__INCLUDED_
#if _MSC_VER >= 1000
```

```
#pragma once
#endif //_MSC_VER >= 1000

#define WM_USER_THREAD_FINISHED WM_USER + 0x100

UINT ThreadFunc (LPVOID pParam);
int Sieve (int nMax);

typedef struct tagTHREADPARMS {
  int nMax;
  HWND hWnd;
} THREADPARMS;

/////////////////////////////////////////////////////////////////////////////
// CSieveDlg dialog

class CSieveDlg : public CDialog
{
// Construction
public:
  CSieveDlg(CWnd* pParent = NULL);  // standard constructor

// Dialog Data
  //{{AFX_DATA(CSieveDlg)
  enum { IDD = IDD_SIEVE_DIALOG };
    // NOTE: the ClassWizard will add data members here
  //}}AFX_DATA

  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CSieveDlg)
  protected:
  virtual void DoDataExchange(CDataExchange* pDX);  // DDX/DDV support
  //}}AFX_VIRTUAL

// Implementation
protected:
  HICON m_hIcon;

  // Generated message map functions
  //{{AFX_MSG(CSieveDlg)
  virtual BOOL OnInitDialog();
  afx_msg void OnPaint();
  afx_msg HCURSOR OnQueryDragIcon();
  afx_msg void OnStart();
  //}}AFX_MSG
  afx_msg LONG OnThreadFinished (WPARAM wParam, LPARAM lParam);
  DECLARE_MESSAGE_MAP()
};
```

```
//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations
// immediately before the previous line.

#endif
// !defined(
//   AFX_SIEVEDLG_H__6DF40C9D_7EA1_11D1_8E53_E4D9F9C00000__INCLUDED_)
```

SieveDlg.cpp

```
// SieveDlg.cpp : implementation file
//

#include "stdafx.h"
#include "Sieve.h"
#include "SieveDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CSieveDlg dialog

CSieveDlg::CSieveDlg(CWnd* pParent /* = NULL */)
  : CDialog(CSieveDlg::IDD, pParent)
{
  //{{AFX_DATA_INIT(CSieveDlg)
    // NOTE: the ClassWizard will add member initialization here
  //}}AFX_DATA_INIT
  // Note that LoadIcon does not require a subsequent
  // DestroyIcon in Win32
  m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CSieveDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CSieveDlg)
    // NOTE: the ClassWizard will add DDX and DDV calls here
  //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CSieveDlg, CDialog)
```

```
  //{{AFX_MSG_MAP(CSieveDlg)
  ON_BN_CLICKED(IDC_START, OnStart)
  //}}AFX_MSG_MAP
  ON_MESSAGE(WM_USER_THREAD_FINISHED, OnThreadFinished)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CSieveDlg message handlers

BOOL CSieveDlg::OnInitDialog()
{
  CDialog::OnInitDialog();
  SetIcon(m_hIcon, TRUE);
  SetIcon(m_hIcon, FALSE);
  return TRUE;
}

void CSieveDlg::OnStart()
{
  int nMax = GetDlgItemInt(IDC_MAX);
  if (nMax < 10) {
    MessageBox(_T("The number you enter must be 10 or higher"));
    GetDlgItem(IDC_MAX)->SetFocus();
    return;
  }

  SetDlgItemText(IDC_RESULT, _T(""));
  GetDlgItem(IDC_START)->EnableWindow(FALSE);

  THREADPARMS* ptp = new THREADPARMS;
  ptp->nMax = nMax;
  ptp->hWnd = m_hWnd;
  AfxBeginThread(ThreadFunc, ptp);
}

LONG CSieveDlg::OnThreadFinished(WPARAM wParam, LPARAM lParam)
{
  SetDlgItemInt(IDC_RESULT, (int) wParam);
  GetDlgItem(IDC_START)->EnableWindow(TRUE);
  return 0;
}

/////////////////////////////////////////////////////////////////////////////
// Global functions

UINT ThreadFunc(LPVOID pParam)
{
```

```
    THREADPARMS * ptp = (THREADPARMS *) pParam;
    int nMax = ptp->nMax;
    HWND hWnd = ptp->hWnd;
    delete ptp;

    int nCount = Sieve (nMax);
    ::PostMessage (hWnd, WM_USER_THREAD_FINISHED, (WPARAM) nCount, 0);
    return 0;
}

int Sieve(int nMax)
{
    PBYTE pBuffer = new BYTE[nMax + 1];
    ::FillMemory (pBuffer, nMax + 1, 1);

    int nLimit = 2;
    while (nLimit * nLimit < nMax)
        nLimit++;

    for (int i = 2; i <= nLimit; i++) {
        if (pBuffer[i]) {
            for (int k = i + i; k <= nMax; k += i)
                pBuffer[k] = 0;
        }
    }

    int nCount = 0;
    for (i = 2; i <= nMax; i++)
        if (pBuffer[i])
            nCount++;

    delete[] pBuffer;
    return nCount;
}
```

图 17-2　Sieve 应用程序

17.2　线程同步

在现实世界中，您通常都不具有开始一个线程而只是让它随意运行的条件。更多的是，那个线程必须将它的动作与应用程序中的其他线程相协调。例如，如果两个线程共享一个链表，那么对该链表的访问必须被串性化，这样两个线程才不会在同一时间试图修改链表。只是让一个线程运行和做它自己的事情，可能会导致在测试中随机出现的各种类型的同步

问题,这些同步问题对应用程序来说可能是致命的。

Windows 支持 4 种类型的同步对象,可以用来同步由并发运行的线程所执行的操作:

- 临界区
- 互斥量
- 事件
- 信号量

MFC 在名为 CCriticalSection、CMutex、CEvent 和 CSemaphore 的类中封装了这些对象。MFC 还包含了名为 CSingleLock 和 CMultiLock 的一对类。在下面的小节中,我将描述如何使用这些类来同步并发执行的线程的操作。

17.2.1 临界区

最简单类型的线程同步对象就是临界区。临界区用来串行化对由两个或者多个线程共享的链表、简单变量、结构和其他资源的访问。这些线程必须属于相同的进程,因为临界区不能跨越进程的边界工作。

临界区背后的思想就是,每个独占性地访问一个资源的线程可以在访问那个资源之前锁定临界区,访问完成之后解除锁定。如果线程 B 试图锁定当前由线程 A 锁定的临界区,线程 B 将阻塞直到该临界区空闲。阻塞时,线程 B 处在一个十分有效的等待状态等待,它不消耗处理器时间。

CCriticalSection::Lock 锁定临界区,而 CCriticalSection::Unlock 解除对临界区的锁定。让我们举个例子来说明,有一个包含了从 MFC 的 CList 类创建的链表数据成员的文档类和两个使用该链表的独立线程。一个线程写入到链表,另一个线程从链表中读取。为防止这两个线程在完全相同的时间访问链表,您可以使用临界区来保护该链表。下面的示例使用一个全局声明的 CCriticalSection 对象来演示其中的方法(我已经在示例中使用了全局的同步对象,以确保这些对象对于进程中的所有线程都是同样可见的,但是同步对象不一定必须具有全局范围)。

```
// Global data
CCriticalSection g_cs;
   .
   .
   .
// Thread A
g_cs.Lock ();
// Write to the linked list.
g_cs.Unlock ();
   .
   .
   .
```

```
// Thread B
g_cs.Lock ();
// Read from the linked list.
g_cs.Unlock ();
```

现在，线程 A 和线程 B 要同时访问该链表是不可能的，因为它们都使用相同的临界区保护了链表。图 17-3 演示了通过串行化线程的操作，临界区是如何防止重叠的读取和写入访问的。

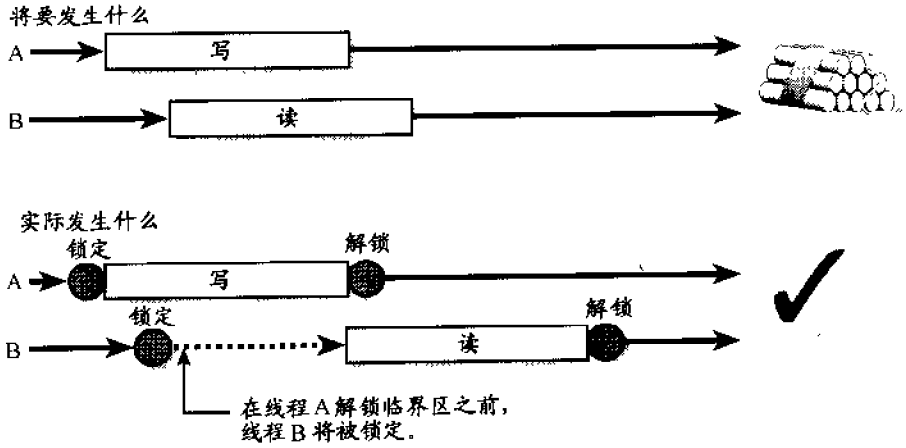

图 17-3　使用临界区保护共享资源

CCriticalSection::Lock 的另一种形式接受超时值，而某些 MFC 文档声明，如果您给 Lock 传递一个超时值，而且如果在临界区空闲之前该超时时间段到期了，Lock 将返回。该说明是错的。如果需要，您可以指定一个超时值，但 Lock 在临界区解除锁定之前是不会返回的。

为什么链表应当防止并发线程的访问是很显然的，但是对于简单变量呢？例如，假定线程 A 使用下面的语句增大一个变量：

```
nVar++;
```

而线程 B 对该变量进行了其他的操作。nVar 是否应当使用临界区来保护呢？一般来说，回答是肯定的。在 C++ 程序（即使是一个简单的 ++ 操作符的应用程序）中一个看来十分简单的原子操作也可能要编译成一个具有多个机器指令的序列。而且在任何两个机器指令之间一个线程可以剥夺另一个线程的机器指令。通常，防止任何数据遭受同时发生的写入访问或者同时发生的读取和写入访问是不错的想法。临界区就是进行这项工作的完美工具。

Win32 API 包含一组名为 ::InterlockedIncrement、::InterlockedDecrement、::InterlockedExchange、::InterlockedCompareExchange 和 ::InterlockedExchangeAdd 的函数，您可以用它们来

安全地操作 32 位的值,而不需显式使用同步对象。例如,如果 nVar 是 UINT、DWORD 或者其他的 32 位数据类型,那么您可以使用下面的语句使之递增:

```
::InterlockedIncrement (&nVar);
```

系统将确保其他使用 Interlocked 函数执行的对 nVar 的访问不会重叠。nVar 应当在 32 位边界上对齐,或者 Interlocked 函数在多处理器 Windows NT 系统上可能会失败。此外,::InterlockedCompareExchange和::InterlockedExchangeAdd 仅在 Windows NT 4.0 或更高版本以及 Windows 98 上支持。

17.2.2 互斥量

Mutex 是单词 mutually 和 exclusive 的缩写。与临界区一样,互斥量也是用来获得对由两个或者更多线程共享的资源的独占性访问的。与临界区不同的是,互斥量可以用来同步在相同进程或者不同进程上运行的线程。对于进程内线程同步的需要,临界区一般要优于互斥量,因为临界区更快,但是如果您希望同步在两个或者多个不同进程上运行的线程,那么互斥量就更合适了。

假定两个应用程序使用一块共享内存来交换数据。在该共享内存的内部是一个必须防止并发线程访问的链表。临界区就不能用了,因为它不能跨越进程的边界,但是互斥量可以相当出色地完成工作。下面就是在读取或者写入链表之前您在每个进程上所要做的:

```
// Global data
CMutex g_mutex (FALSE,_T ("MyMutex"));
  .
  .
  .
g_mutex.Lock ();
// Read or write the linked list.
g_mutex.Unlock ();
```

传递给 CMutex 构造函数的第一个参数指定互斥量的初始状态是锁定(TRUE)或者没有锁定(FALSE)。第二个参数指定互斥量的名称,如果该互斥量是用来同步在两个不同进程上的线程,就需要这个名称。您可以选择名称,但这两个进程必须指定相同的名称,这样两个 CMutex 对象将引用 Windows 内核中相同的互斥量对象。显然,Lock 在由另一个线程锁定的互斥量上将阻塞,而 Unlock 释放互斥量,以便其他的进程可以锁定它。

默认情况下,Lock 将永远等待直到互斥量变为没有锁定。您可以通过指定一个以毫秒为单位的最大等待时间来建立一个安全失败机制。在下面的示例中,线程在访问由互斥量保护的资源前等待可多达 1 秒时间。

```
g_mutex.Lock (60000);
// Read or write the linked list.
g_mutex.Unlock ();
```

Lock 的返回值告诉您函数调用返回的原因。一个非 0 的返回值表示互斥量空闲，而 0 表示超时时间段首先到期了。如果 Lock 返回 0，不访问共享资源通常是比较谨慎的，因为访问共享资源可能会导致重叠访问。因此，使用 Lock 的超时特性的代码一般如下面这样构造：

```
if (g_mutex.Lock (60000)) {
  // Read or write the linked list.
  g_mutex.Unlock ();
}
```

互斥量和临界区之间还有另外一个差别。如果一个线程锁定了临界区而终止时没有解除临界区的锁定，那么等待临界区空闲的其他线程将无限期地阻塞下去。然而，如果锁定互斥量的线程不能在其终止前解除互斥量的锁定，那么系统将认为互斥量被"放弃"了并自动释放该互斥量，这样等待进程就可以继续执行。

17.2.3　事件

MFC 的 CEvent 类封装了 Win32 事件对象。一个事件不只是操作系统内核中的一个标记。在任何特定的时间，事件只能处在两种状态中的一种：引发（设置）或者调低（重置）。设置状态事件也可以认为是处于信号状态，重置状态事件也可以认为是处于非信号状态。CEvent::SetEvent 设置一个事件，而 CEvent::ResetEvent 将事件重置。相关函数 CEvent::PulseEvent 可以在一次操作中设置和清除一个事件。

有时事件被描述为"线程触发器"。一个线程调用 CEvent::Lock 在一个事件上阻塞，等待该事件变为设置状态。另一个线程设置事件，从而唤醒该等待线程。设置事件就像按下触发器，它解除等待线程的阻塞并允许该线程继续执行。一个事件可能有一个或者多个在事件上阻塞的线程，如果您的代码编写正确，那么当该事件变为设置状态时，所有的等待线程都将被唤醒。

Windows 支持两种不同类型的事件：自动重置事件和手动重置事件。它们之间的差别非常细微，但其意义却是深远的。当在自动重置事件上阻塞的线程被唤醒时，该事件被自动重置为信号状态。手动重置事件不能自动重置，它必须使用编程方式重置。用于选择自动重置事件还是手动重置事件——以及一旦您做出选择之后如何使用它们——的规则如下所示：

- 如果事件只触发一个线程，那么使用自动重置事件和使用 SetEvent 唤醒等待线程，这里不需要调用 ResetEvent，因为线程被唤醒的那一刻事件将被自动重置
- 如果事件将触发两个或者多个线程，那么使用手动重置线程和使用 PulseEvent 唤醒

所有的等待线程。而且,您不需要调用 ResetEvent,因为 PulseEvent 在唤醒线程后将为您重置事件。

使用手动重置事件来触发多个线程是至关重要的。为什么?因为自动重置事件将在其中一个线程被唤醒的那一刻被重置,因此它只触发一个线程。使用 PulseEvent 来按下手动重置事件上的触发器也是相当重要的。如果您使用 SetEvent 和 ResetEvent,您就不能保证所有的等待线程将被唤醒。PulseEvent 不仅能够设置和重置事件,而且还确保了所有在事件上等待的线程在重置事件之前被唤醒。

通过构造一个 CEvent 对象来创建事件。CEvent::CEvent 接受 4 个参数,它们都是可默认的。其原形如下所示:

```
CEvent (BOOL bInitiallyOwn = FALSE,
  BOOL bManualReset = FALSE, LPCTSTR lpszName = NULL,
  LPSECURITY_ATTRIBUTES lpsaAttribute = NULL)
```

第 1 个参数 bInitiallyOwn 指定事件对象被初始化为信号状态(TRUE)还是非信号状态(FALSE)。在大多数情况下取默认值即可。bManualReset 指定事件为手动重置事件(TRUE)还是自动重置事件(FALSE)。第 3 个参数 lpszName 给事件对象指定一个名称。与互斥量相同的是,事件可以用来协调在不同进程上运行的线程,对于跨越进程边界的事件,必须给它指定一个名称。如果使用事件的多个线程属于同一个进程,那么 lpszName 应当为 NULL。第 4 个参数 lpsaAttribute 是一个指向 SECURITY_ATTRIBUTES 结构的指针,它描述了事件对象的安全属性。NULL 表示接受默认的安全属性,它适用于大部分的应用程序。

那么您怎样使用事件来同步线程呢?下面的示例包括一个向缓冲区填充数据的线程(线程 A)和另一个对数据进行处理的线程(线程 B)。假定线程 B 必须等待来自线程 A 的一个信号(缓冲区已初始化并准备工作)。自动重置事件是完成这项工作的绝好工具:

```
// Global data
CEvent g_event; // Autoreset, initially nonsignaled
  .
  .
  .
// Thread A
InitBuffer (&buffer); // Initialize the buffer.
g_event.SetEvent (); // Release thread B.
  .
  .
  .
// Thread B
g_event.Lock ();  // Wait for the signal.
```

线程 B 调用 Lock 来阻塞事件对象。当线程 A 准备唤醒线程 B 时调用 SetEvent。图 17-4 表

明了发生的结果。

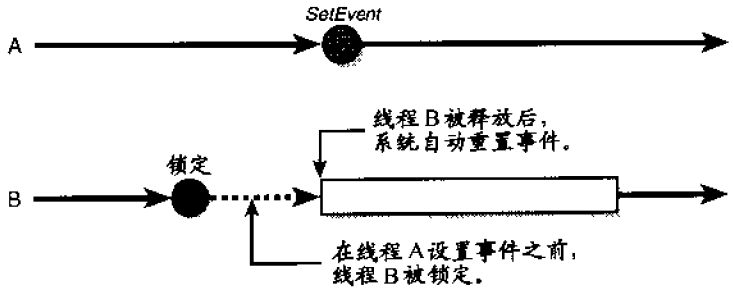

图 17-4　使用自动重置事件触发线程

传递给 Lock 的单个参数指定了调用者愿意等待的时间长度，以毫秒为单位。默认值是 INFINITE，表示需要等多长时间就等多长时间。非 0 返回值表示 Lock 成功返回，因为对象变为信号状态；0 表示超时时间段到期或者发生错误。这里 MFC 没有做任何奇特的事。它只是重新构造内核中的线程同步对象以及 API 函数，它们以更加面向对象的模式操作这些对象。

自动重置事件适用于触发单线程，但如果与线程 B 平行运行的线程 C 对缓冲的数据进行了完全不同的操作，那会怎么样呢？您需要手动重置事件一同唤醒线程 B 和 C，因为自动重置事件只能唤醒其中的一个或者另一个，而不能都唤醒。下面就是使用手动重置事件触发两个或者更多线程的代码：

```
// Global data
CEvent g_event (FALSE, TRUE); // Nonsignaled, manual-reset
  .
  .
  .
// Thread A
InitBuffer (&buffer); // Initialize the buffer.
g_event.PulseEvent (); // Release threads B and C.
  .
  .
  .
// Thread B
g_event.Lock ();   // Wait for the signal.
  .
  .
  .
// Thread C
g_event.Lock ();   // Wait for the signal.
```

注意,线程 A 使用 PulseEvent 按下触发器,符合上面规定的两条规则中的第二条。图 17-5 演示了使用手动重置事件来触发两个线程的效果。

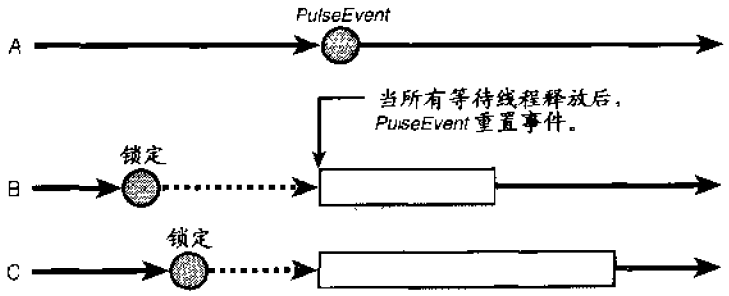

图 17-5 使用手动重置事件触发两个线程

再次重申,请用自动重置事件和 CEvent::SetEvent 释放在事件上阻塞的单个线程;用手动重置事件和 CEvent::PulseEvent 释放多个线程。如果遵从这些简单原则,事件就会可信赖地、有效地为您服务。

有时不用事件做触发器,而是用它作原始的信号发生机制。例如:或许线程 B 想知道线程 A 是否已经完成任务,但是如果答案是否定的,它不希望被阻塞。通过传递给::WaitForSingleObject 事件句柄和超时值 0,线程 B 可以避免被阻塞并检查事件的状态。事件句柄可以从 CEvent 的 m_hObject 数据成员中提取:

```
if (::WaitForSingleObject (g_event.m_hObject, 0) == WAIT_OBJECT_0) {
  // The event is signaled.
}else {
  // The event is not signaled.
}
```

这样使用事件时要注意一个警告:如果在事件变成设置状态之前,线程要反复检查事件,则一定要确信事件是手动重置事件,而不是自动重置事件。否则,检查事件本身就会使它重置。

17.2.4 信号量

第 4 种,即最后一种同步化对象是信号量。如果任何一个线程锁定了事件、临界区和互斥对象,Lock 就会阻塞它们。在这个意义上,这 3 种对象具有这样的特性:"要么有,要么什么都没有"。信号量则不同,它始终保存有代表可用资源数量的资源数。锁定信号量会减少资源数,释放信号量则增加资源数。只有在线程试图锁定资源数为 0 的信号量时,线程才会被阻塞。在这种情况下,直到另一个线程释放信号量,资源数随之增加时,或者直到指定的

超时时间期满时，该线程才被释放。信号量可以用来同步化同一进程中的线程，也可以同步化不同进程中的线程。

MFC 用类 CSemaphore 的实例代表信号量。语句

```
CSemaphore g_semaphore(3, 3);
```

构造了一个信号量对象，其初始资源数为 3（参数 1），最大资源数也为 3（参数 2）。如果信号量用于同步化不同进程中的线程，则要添加第三个参数，并赋予它信号量名。可选的第四个参数指向 SECURITY_ATTRIBUTES 结构（默认值等于 NULL）。每个线程可以这样访问由信号量控制的资源：

```
g_semaphore.Lock();
// Access the shared resource.
g_semaphore.Unlock();
```

只要同时访问资源的线程不超过 3 个，则 Lock 不会暂停线程。但是如果信号量已被 3 个线程锁定，而又有第 4 个线程调用 Lock，则该线程只有等到其他三个线程中的一个调用 Unlock 时才被释放（参见图 17-6）。为了限制 Lock 等待信号量资源数变为零的时间，可以传递给 Lock 函数一个最大等待时间（用毫秒表示）。

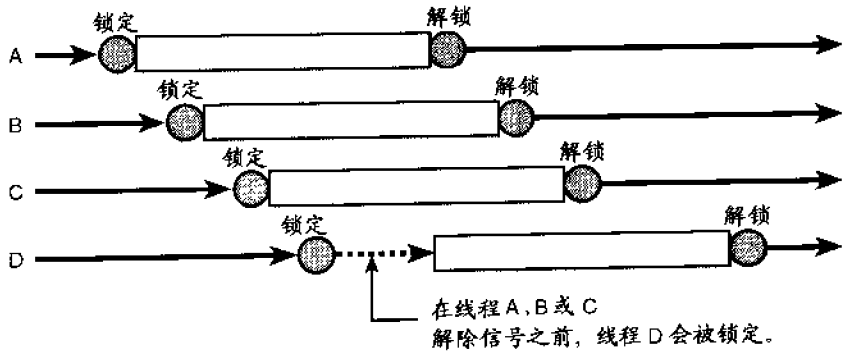

图 17-6　使用信号量保护共享资源

CSemaphore::Unlock 可用大于 1 的增量增加资源数，并在调用 Unlock 之前确定资源数的大小。例如：假定同一线程连续两次调用 Lock，请求使用由信号量保护的两个资源，则为了解除锁定，线程不必调用 Unlock 两次，可以这样：

```
LONG lPrevCount;
g_semaphore.Unlock(2, &lPrevCount);
```

除了 CSemaphore::Unlock 和它在 API 中的等价函数::ReleaseSemaphore 外，在 MFC 或

API 中没有其他函数能返回信号量的资源数。

信号量的传统用法是：它允许一组线程(m 个)共享 n 个资源，其中 m 比 n 大。例如：假定要启动 10 个线程，并且每个线程都要收集数据。不管何时线程用数据填充缓冲区，线程都要通过打开的套接字传送数据，清空缓冲区并重新开始收集数据。现在假定在任意时刻只有三个套接字可用。如果用信号量保护该套接字池，其中资源数为 3，然后编写各线程代码，使它们在请求套接字前锁定信号量，则线程在等待套接字被释放的过程中就不会占用 CPU 时间。

17.2.5 CSingleLock 和 CMultiLock 类

MFC 包含了一对类：CSingleLock 和 CMultiLock，它们具有自己的 Lock 和 Unlock 函数。您可以在 CSingleLock 对象中包装临界区、互斥对象、事件或信号量，并使用 CSingleLock::Lock 来实现锁定，如下所示：

```
CCriticalSection g_cs;
  .
  .
  .
CSingleLock lock (&g_cs); // Wrap it in a CSingleLock.
lock.Lock ();     // Lock the critical section.
```

用此方法锁定临界区而不是直接调用 CCriticalSection 对象的 Lock 函数有好处吗？有时有。请考虑一下，当下列程序在调用 Lock 和 Unlock 之间产生异常时会发生什么情况？

```
g_cs.Lock ();
  .
  .
  .
g_cs.Unlock ();
```

如果异常发生，临界区会永远维持锁定状态，因为对 Unlock 的调用被绕开了。但是如果您按下列方法构造程序，那么会发生什么情况？

```
CSingleLock lock (&g_cs);
lock.Lock ();
  .
  .
  .
lock.Unlock ();
```

临界区并不会永久地被锁定。为什么？因为 CSingleLock 对象是在堆栈上创建的，如果异常

出现就会调用它的析构函数。CSingleLock 的析构函数会调用被包含的同步化对象中的 Unlock。换句话说，CSingleLock 是一个便于使用的工具，用来确保即使面对偶然产生的异常，已锁定的同步化对象也可以得到解锁。

CMultiLock 则完全不同。通过使用 CMultiLock，一个线程可以一次阻塞至多 64 个同步化对象。并且根据它调用 CMultiLock::Lock 方式的不同，线程可以处于阻塞状态直到同步化对象之一获得自由或直到所有对象被释放。下列代码说明了一个线程同时阻塞了两个事件和一个互斥对象。要注意事件、互斥对象和信号量可以封装在 CMultiLock 对象中，而临界区则不行。

```
CMutex g_mutex;
CEvent g_event[2];
CSyncObject * g_pObjects[3] = { &g_mutex, &g_event[0], &g_event[1] };
  .
  .
  .
// Block until all three objects become signaled.
CMultiLock multiLock (g_pObjects, 3);
multiLock.Lock ();
  .
  .
  .
// Block until one of the three objects becomes signaled.
CMultiLock multiLock (g_pObjects, 3);
multiLock.Lock (INFINITE, FALSE);
```

CMultiLock::Lock 接受 3 个参数，它们都是可选的。第 1 个指定等待时间值（默认为 INFINITE）。第 2 个参数指定线程应该被唤醒的时间，是在同步化对象之一解锁（FALSE）之后还是在所有对象解锁（默认为 TRUE）之后。第 3 个参数是“唤醒掩码”，指定了唤醒线程的其他条件，例如 WM_PAINT 消息或鼠标键消息。默认唤醒掩码的值为 0，它防止因任何原因唤醒线程，除非是同步化对象被释放或等待时间已经期满。

在调用 CMultiLock::Lock 使线程处于阻塞状态，直到一个同步化对象进入信号发出状态为止，如果线程脱离了阻塞状态，一般情况下线程需要知道是哪个对象处于了信号发出状态。从 Lock 的返回值中可以弄清答案：

```
CMutex g_mutex;
CEvent g_event[2];
CSyncObject * g_pObjects[3] = { &g_mutex, &g_event[0], &g_event[1] };
  .
  .
  .
```

```
CMultiLock multiLock (g_pObjects, 3);
DWCRD dwResult = multiLock.Lock (INFINITE, FALSE);
DWCRD nIndex = dwResult - WAIT_OBJECT_0;
if (nIndex == 0) {
  // The mutex became signaled.
}
else if (nIndex == 1) {
  // The first event became signaled.
}
else if (nIndex == 2) {
  // The second event became signaled.
}
```

注意,如果给 Lock 传递的等待时间值不是 INFINITE,就应该在减去 WAIT_OBJECT_0 之前将返回值与 WAIT_TIMEOUT 进行比较,以防 Lock 在等待期满后返回。另外,如果 Lock 返回是因为已放弃的互斥对象进入了信号已发出状态,那么您必须从返回值中减去 WAIT_ABANDONED_0 而不是 WAIT_OBJECT_0。想了解更详细的内容,请参见有关 CMultiLock::Lock 的文献。

下面给出的例子说明了可以使用 CMultiLock 的情况。假设 3 个线程,线程 A、B 和 C,共同来准备缓冲区中的数据。一旦数据准备好,线程 D 就会通过套接字来传送数据或将它写入文件。但是在线程 A、B、C 全部完成它们的工作之前不能调用线程 D。怎样解决此问题呢?创建独立的时间对象代表线程 A、B、C,让线程 D 使用 CMultiLock 对象而进入阻塞状态,直到所有 3 个事件都处于信号发出状态。当每个线程都完成它们的工作时,它就会将相应的事件对象设置为信号发出状态。所以线程 D 会维持阻塞状态,直到 3 个线程中的最后一个被设置为信号发出状态。

17.2.6 编写线程安全类

MFC 类在类层次是线程安全的而在对象层次上却不是。换句话说,这意味着两个线程可以安全地访问同一个类的两个独立的实例,但是如果允许两个线程同时访问同一个实例就会产生问题。MFC 的设计者出于程序运行效率的原因没有选择将线程安全设置在对象层次。简单的锁定已解锁的临界区的动作可能会在 Pentium 处理器上消耗成百上千个时钟周期。如果对 MFC 类对象的每个访问都锁定一个临界区,单线程应用程序的运行效率就会遭受不必要的损害。

要说明对于类而言线程安全的含义,让我们考虑一下,如果两个线程使用同一个 CString 对象,而它们的操作没有进行同步化处理时会发生什么情况。假设线程 A 要设置一个字符串,字符串名字为 g_strFileName,它等于由 pszFile 引用的文本字符串:

```
g_strFileName = pszFile;
```

几乎同时，线程 B 决定要把 g_strFileName 传递给 CDC::TextOut 以在屏幕上显示字符串：

```
pDC -> TextOut (x, y, g_strFileName);
```

那么会显示什么样的内容呢？g_strFileName 中以前的值还是现在的值？可能都不是。将文本复制给 CString 对象是一种多步操作，涉及到分配缓冲空间来保存文本；执行 memcpy 来复制字符串；设置 CString 数据成员用来保存字符串长度，使它等于被复制的字符串数量；并在最后添加一个结束标记 0，等等。如果线程 B 在不合适的时刻中断了此过程，那么在传递给 TextOut 时，CString 的状态就说不清会是什么样子了。输出可能会被错误地截断。或者 TextOut 可能显示无用信息或产生非法访问。

同步化对 g_strFileName 访问的一种方法是用临界区保护它，如下所示：

```
// Global data
CCriticalSection g_cs;
  .
  .
  .
// Thread A
g_cs.Lock ();
g_strFileName = pszFile;
g_cs.Unlock ();
  .
  .
  .
// Thread B
g_cs.Lock ();
pDC -> TextOut (x, y, g_strFileName);
g_cs.Unlock ();
```

另一种方法是从 CString 派生一个类，通过将其生成于临界区内来实现派生类的线程安全性，该临界区要在任何访问发生的时刻被自动锁定。这样对象自身就确保了访问是在线程安全的状态下进行的，并且再也不用将同步化线程操作的责任依赖于使用对象的应用程序了。

派生一个类并使它具有线程安全性基本上就是要覆盖所有读取或写入对象数据的成员函数，并用锁定和解锁作为派生类成员的同步化对象的函数调用来封装基类中成员函数的调用。同样，对于那些不是通过派生而是通过组合得到的线程安全类来说，要给类添加 CCriticalSection 或 CMutex 数据成员，并在任何访问发生之前锁定和解锁同步化对象。

不可能使类完全成为线程安全的。例如：如果一个线程使用了 GetBuffer 或 LPCTSTR 运算符来获取指向 CString 文本的指针，CString 本身就无法控制调用者对该指针的使用了。在

这种情况下,仍然是使用 CString 对象的线程具有与其他线程协调访问的责任。

从以上讨论的所有内容中要把握一点:默认状态下对象不具有线程安全性。您可以以线程安全的方式使用同步化对象来访问其他对象,也可以通过控制对该类所创建的对象的访问来开发本质上具有线程安全性的类。但是在一个线程修改对象数据时允许另一个线程读取数据,或者相反,都是造成灾难的根源。有时情况会更糟,这种特点的错误在测试阶段通常会随机地出现。您可能会运行应用程序 1 000 次而从来不会感到运行结果有什么不同。但是可以确信某种可能性是存在的,您的应用程序的某个用户有可能在某个最坏时刻经历到这种双重访问带来的灾难,从而使整个应用程序(也可能是整个操作系统)陷入瘫痪。

17.2.7 ImageEdit 应用程序

图 17-7 所示的应用程序是第 15 章中的 Vista 应用程序的增强版本,它在背景中使用了独立的线程执行复杂的图像处理任务。当您从 Effects 菜单中选择了 Convert To Gray Scale 命令之后,ImageEdit 会逐个像素地扫描当前位图,并将每个像素都转换为相应的灰度,并通过调整调色板来显示原始彩色图像转换为灰度等级后的效果。

图 17-7 ImageEdit 窗口

转换函数很适合工作者线程来执行,因为根据位图的大小、CPU 的速度以及其他因素的不同,转换工作所花费的时间也从几秒到几分钟不等。执行转换的程序决非最优的程序;实际上,如果重新编写程序使转换直接作用在位图位上而不是对每个像素都调用 CDC::

GetPixel和 CDC::SetPixel，那么执行速度可能会提高 10 倍或更高。但是出于演示的目的，现在这样也可以了。并且，使用 CDC 像素函数来获取和设置像素颜色，我们可以用 20 行左右的程序代码就能实现转换功能，而如果重写 ImageEdit 使它能够处理原始位图数据，那就至少需要几百行语句。

在图 17-8 中给出了大量的 ImageEdit 源程序。在本章中我想提供一个多线程文档/视图程序，因为在多线程 SDK 应用程序或其他不使用文档/视图的 MFC 应用程序中并不存在编写多线程文档/视图应用程序所特有的问题。例如：文档对象启动工作者线程来处理文档数据并不是很少见。但是后台线程又如何让文档对象知道处理已经结束了呢？它不能给文档公布消息，因为文档不是窗口。让文档阻塞在一个事件上等待线程完成任务也不是个好主意，因为这样的话就暂停了应用程序的主线程，实际上也中止了消息循环。而文档通常确实是需要知道线程的结束时间以便更新视图。问题是：怎么办？

MainFrm.h

```
// MainFrm.h : interface of the CMainFrame class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(
  AFX_MAINFRM_H__9D77AEE8_AA14_11D2_8E53_006008A82731__INCLUDED_)
#define AFX_MAINFRM_H__9D77AEE8_AA14_11D2_8E53_006008A82731__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class CMainFrame : public CFrameWnd
{

protected: // create from serialization only
  CMainFrame();
  DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:

// Operations
public:

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CMainFrame)
  virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
```

```
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CMainFrame();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected: // control bar embedded members
    CSpecialStatusBar m_wndStatusBar;

// Generated message map functions
protected:
    int m_nPercentDone;

    //{{AFX_MSG(CMainFrame)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg BOOL OnQueryNewPalette();
    afx_msg void OnPaletteChanged(CWnd* pFocusWnd);
    //}}AFX_MSG
    afx_msg LRESULT OnUpdateImageStats (WPARAM wParam, LPARAM lParam);
    afx_msg LRESULT OnThreadUpdate (WPARAM wParam, LPARAM lParam);
    afx_msg LRESULT OnThreadFinished (WPARAM wParam, LPARAM lParam);
    afx_msg LRESULT OnThreadAborted (WPARAM wParam, LPARAM lParam);
    DECLARE_MESSAGE_MAP()
};

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

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations
// immediately before the previous line.

#endif
//!defined(AFX_MAINFRM_H__9D77AEE8_AA14_11D2_8E53_006008A82731__INCLUDED_)
```

MainFrm.cpp

```
// MainFrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"
#include "ImageEdit.h"
```

```
#include "ImageEditDoc.h"
#include "SpecialStatusBar.h"
#include "MainFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMainFrame

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)

  ON_WM_CREATE()
  ON_WM_QUERYNEWPALETTE()
  ON_WM_PALETTECHANGED()
  //}}AFX_MSG_MAP
  ON_MESSAGE (WM_USER_UPDATE_STATS, OnUpdateImageStats)
  ON_MESSAGE (WM_USER_THREAD_UPDATE, OnThreadUpdate)
  ON_MESSAGE (WM_USER_THREAD_FINISHED, OnThreadFinished)
  ON_MESSAGE (WM_USER_THREAD_ABORTED, OnThreadAborted)
END_MESSAGE_MAP()

static UINT indicators[] =
{
  ID_SEPARATOR,
  ID_SEPARATOR,
  ID_SEPARATOR
};

/////////////////////////////////////////////////////////////////////////////
// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
  m_nPercentDone = -1;
}

CMainFrame::~CMainFrame()
{
}

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
```

```
}
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

  if (!m_wndStatusBar.Create(this))
  {
    TRACE0("Failed to create status bar\n");
    return -1;   // fail to create
  }
  return 0;
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
  if( !CFrameWnd::PreCreateWindow(cs) )
    return FALSE;
  return TRUE;
}
/////////////////////////////////////////////////////////////////////////////
// CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
  CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
  CFrameWnd::Dump(dc);
}

#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CMainFrame message handlers

BOOL CMainFrame::OnQueryNewPalette()
{
  CDocument * pDoc = GetActiveDocument ();
  if (pDoc != NULL)
    GetActiveDocument () -> UpdateAllViews (NULL);
  return TRUE;
}

void CMainFrame::OnPaletteChanged(CWnd * pFocusWnd)
{
```

```
    if (pFocusWnd != this) {
      CDocument * pDoc = GetActiveDocument ();
      if (pDoc != NULL)
        GetActiveDocument () -> UpdateAllViews (NULL);
    }
}

LRESULT CMainFrame::OnUpdateImageStats (WPARAM wParam, LPARAM lParam)
{
  m_wndStatusBar.SetImageStats ((LPCTSTR) lParam);
  return 0;
}

LRESULT CMainFrame::OnThreadUpdate (WPARAM wParam, LPARAM lParam)
{
  int nPercentDone = ((int) wParam * 100) / (int) lParam;
  if (nPercentDone != m_nPercentDone) {
    m_wndStatusBar.SetProgress (nPercentDone);
    m_nPercentDone = nPercentDone;
  }
  return 0;
}

LRESULT CMainFrame::OnThreadFinished (WPARAM wParam, LPARAM lParam)
{
  CImageEditDoc * pDoc = (CImageEditDoc * ) GetActiveDocument ();
  if (pDoc != NULL) {
    pDoc -> ThreadFinished ();
    m_wndStatusBar.SetProgress (0);
    m_nPercentDone = -1;
  }
  return 0;
}

LRESULT CMainFrame::OnThreadAborted (WPARAM wParam, LPARAM lParam)
{
  CImageEditDoc * pDoc = (CImageEditDoc * ) GetActiveDocument ();
  if (pDoc != NULL) {
    pDoc -> ThreadAborted ();
    m_wndStatusBar.SetProgress (0);
    m_nPercentDone = -1;

  return 0;
}
```

ImageEditDoc.h

```
// ImageEditDoc.h : interface of the CImageEditDoc class
//
/////////////////////////////////////////////////////////////////////////////
#if !defined(
  AFX_IMAGEEDITDOC_H__9D77AEEA_AA14_11D2_8E53_006008A82731__INCLUDED_)
#define AFX_IMAGEEDITDOC_H__9D77AEEA_AA14_11D2_8E53_006008A82731__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
UINT ThreadFunc (LPVOID pParam);
LOGPALETTE* CreateGrayScale ();

class CImageEditDoc : public CDocument
{
protected: // create from serialization only
  CImageEditDoc();
  DECLARE_DYNCREATE(CImageEditDoc)

// Attributes
public:

// Operations
public:

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CImageEditDoc)
  public:
  virtual BOOL OnNewDocument();
  virtual BOOL OnOpenDocument(LPCTSTR lpszPathName);
  virtual void DeleteContents();
  //}}AFX_VIRTUAL

// Implementation
public:
  void ThreadAborted();
  void ThreadFinished();
  CPalette* GetPalette();
  CBitmap* GetBitmap();
  virtual ~CImageEditDoc();
#ifdef_DEBUG
  virtual void AssertValid() const;
  virtual void Dump(CDumpContext& dc) const;
#endif
```

```
protected:

// Generated message map functions
protected:
  CCriticalSection m_cs;
  CEvent m_event;
  HANDLE m_hThread;
  BOOL m_bWorking;
  CPalette m_palette;
  CBitmap m_bitmap;
  //{{AFX_MSG(CImageEditDoc)
  afx_msg void OnGrayScale();

  afx_msg void OnUpdateGrayScale(CCmdUI * pCmdUI);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

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

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations
// immediately before the previous line.

#endif
// !defined(
//   AFX_IMAGEEDITDOC_H__9D77AEEA_AA14_11D2_8E53_006008A82731__INCLUDED_)
```

ImageEditDoc.cpp

```
// ImageEditDoc.cpp : implementation of the CImageEditDoc class
//

#include "stdafx.h"
#include "ImageEdit.h"

#include "ImageEditDoc.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CImageEditDoc
```

```
IMPLEMENT_DYNCREATE(CImageEditDoc, CDocument)

BEGIN_MESSAGE_MAP(CImageEditDoc, CDocument)
  //{{AFX_MSG_MAP(CImageEditDoc)
  ON_COMMAND(ID_EFFECTS_GRAY_SCALE, OnGrayScale)
  ON_UPDATE_COMMAND_UI(ID_EFFECTS_GRAY_SCALE, OnUpdateGrayScale)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
////////////////////////////////////////////////////////////////////////////
// CImageEditDoc construction/destruction

CImageEditDoc::CImageEditDoc() :
  m_event(FALSE, TRUE) // Manual-reset event, initially unowned
{
  m_hThread = NULL;
  m_bWorking = FALSE;
}

CImageEditDoc::~CImageEditDoc()
{
}

BOOL CImageEditDoc::OnNewDocument()
{
  if (!CDocument::OnNewDocument())
    return FALSE;
  return TRUE;
}

////////////////////////////////////////////////////////////////////////////
// CImageEditDoc diagnostics

#ifdef _DEBUG
void CImageEditDoc::AssertValid() const
{
  CDocument::AssertValid();
}

void CImageEditDoc::Dump(CDumpContext& dc) const
{
  CDocument::Dump(dc);
}
#endif //_DEBUG

////////////////////////////////////////////////////////////////////////////
// CImageEditDoc commands

BOOL CImageEditDoc::OnOpenDocument(LPCTSTR lpszPathName)
```

```
{
  //
  // Return now if an image is being processed.
  //
  if (m_bWorking) {
    AfxMessageBox (_T ("You can't open an image while another is "\
      "being converted"));
    return FALSE;
  }

  //
  // Let the base class do its thing.
  //
  if (!CDocument::OnOpenDocument (lpszPathName))
    return FALSE;

  //
  // Open the file and create a DIB section from its contents.
  //
  HBITMAP hBitmap = (HBITMAP) ::LoadImage (NULL, lpszPathName,
    IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE|LR_CREATEDIBSECTION);

  if (hBitmap == NULL) {
    CString string;
    string.Format (_T ("%s does not contain a DIB"), lpszPathName);
    AfxMessageBox (string);
    return FALSE;
  }

  m_bitmap.Attach (hBitmap);

  //
  // Return now if this device doesn't support palettes.
  //
  CClientDC dc (NULL);
  if ((dc.GetDeviceCaps (RASTERCAPS) & RC_PALETTE) == 0)
    return TRUE;

  //
  // Create a palette to go with the DIB section.
  //
  if ((HBITMAP) m_bitmap != NULL) {
    DIBSECTION ds;
    m_bitmap.GetObject (sizeof (DIBSECTION), &ds);

    int nColors;
```

```
    if (ds.dsBmih.biClrUsed != 0)
      nColors = ds.dsBmih.biClrUsed;
    else
      nColors = 1 << ds.dsBmih.biBitCount;

    //
    // Create a halftone palette if the DIB section contains more
    // than 256 colors.
    //
    if (nColors > 256)
      m_palette.CreateHalftonePalette (&dc);

    //
    // Create a custom palette from the DIB section's color table
    // if the number of colors is 256 or less.
    //
    else {
      RGBQUAD* pRGB = new RGBQUAD[nColors];

      CDC memDC;
      memDC.CreateCompatibleDC (&dc);
      CBitmap* pOldBitmap = memDC.SelectObject (&m_bitmap);
      ::GetDIBColorTable ((HDC) memDC, 0, nColors, pRGB);
      memDC.SelectObject (pOldBitmap);

      UINT nSize = sizeof (LOGPALETTE) +
        (sizeof (PALETTEENTRY) * (nColors - 1));
      LOGPALETTE* pLP = (LOGPALETTE*) new BYTE[nSize];

      pLP->palVersion = 0x300;
      pLP->palNumEntries = nColors;

      for (int i=0; i<nColors; i++) {
        pLP->palPalEntry[i].peRed = pRGB[i].rgbRed;
        pLP->palPalEntry[i].peGreen = pRGB[i].rgbGreen;
        pLP->palPalEntry[i].peBlue = pRGB[i].rgbBlue;
        pLP->palPalEntry[i].peFlags = 0;
      }

      m_palette.CreatePalette (pLP);
      delete[] pLP;
      delete[] pRGB;
    }
  }
  return TRUE;
}
```

```
void CImageEditDoc::DeleteContents()
{
  if ((HBITMAP) m_bitmap != NULL)
    m_bitmap.DeleteObject ();

  if ((HPALETTE) m_palette != NULL)
    m_palette.DeleteObject ();

  CDocument::DeleteContents();
}

CBitmap * CImageEditDoc::GetBitmap()
{
  return ((HBITMAP) m_bitmap == NULL) ? NULL : &m_bitmap;
}

CPalette * CImageEditDoc::GetPalette()
{
  return ((HPALETTE) m_palette == NULL) ? NULL : &m_palette;
}

void CImageEditDoc::ThreadFinished()
{
  ASSERT (m_hThread != NULL);
  ::WaitForSingleObject (m_hThread, INFINITE);
  ::CloseHandle (m_hThread);
  m_hThread = NULL;
  m_bWorking = FALSE;

  //
  // Replace the current palette with a gray scale palette.
  //
  if ((HPALETTE) m_palette != NULL) {
    m_palette.DeleteObject ();
    LOGPALETTE * pLP = CreateGrayScale ();
    m_palette.CreatePalette (pLP);
    delete[] pLP;
  }

  //
  // Tell the view to repaint.
  //
  UpdateAllViews (NULL);
}

void CImageEditDoc::ThreadAborted()
{
```

```
    ASSERT (m_hThread != NULL);
    ::WaitForSingleObject (m_hThread, INFINITE);
    ::CloseHandle (m_hThread);
    m_hThread = NULL;
    m_bWorking = FALSE;
  }

  void CImageEditDoc::OnGrayScale()
  {
    if (!m_bWorking) {
      m_bWorking = TRUE;
      m_event.ResetEvent ();

      //
      // Package data to pass to the image processing thread.
      //
      THREADPARMS * ptp = new THREADPARMS;
      ptp->pWnd = AfxGetMainWnd ();
      ptp->pBitmap = &m_bitmap;
      ptp->pPalette = &m_palette;
      ptp->pCriticalSection = &m_cs;
      ptp->pEvent = &m_event;

      //
      // Start the image processing thread and duplicate its handle.
      //
      CWinThread * pThread = AfxBeginThread (ThreadFunc, ptp,
        THREAD_PRIORITY_NORMAL, 0, CREATE_SUSPENDED);

      ::DuplicateHandle (GetCurrentProcess (),
        pThread->m_hThread, GetCurrentProcess (), &m_hThread,
        0, FALSE, DUPLICATE_SAME_ACCESS);

      pThread->ResumeThread ();
    }
    else
      //
      // Kill the image processing thread.
      //
      m_event.SetEvent ();
  }

  void CImageEditDoc::OnUpdateGrayScale(CCmdUI * pCmdUI)
  {
    if (m_bWorking) {
      pCmdUI->SetText (_T ("Stop &Gray Scale Conversion"));
```

```
    pCmdUI->Enable();
  }
  else {
    pCmdUI->SetText(_T("Convert to &Gray Scale"));
    pCmdUI->Enable((HBITMAP) m_bitmap != NULL);
  }
}

////////////////////////////////////////////////////////////////////////////
// Thread function and other globals

UINT ThreadFunc(LPVOID pParam)
{
  THREADPARMS* ptp = (THREADPARMS*) pParam;
  CWnd* pWnd = ptp->pWnd;
  CBitmap* pBitmap = ptp->pBitmap;
  CPalette* pPalette = ptp->pPalette;
  CCriticalSection* pCriticalSection = ptp->pCriticalSection;
  CEvent* pKillEvent = ptp->pEvent;
  delete ptp;

  DIBSECTION ds;
  pBitmap->GetObject(sizeof(DIBSECTION), &ds);
  int nWidth = ds.dsBm.bmWidth;
  int nHeight = ds.dsBm.bmHeight;

  //
  // Initialize one memory DC (memDC2) to hold a color copy of the
  // image and another memory DC (memDC1) to hold a gray scale copy.
  //
  CClientDC dc(pWnd);
  CBitmap bitmap1, bitmap2;
  bitmap1.CreateCompatibleBitmap(&dc, nWidth, nHeight);
  bitmap2.CreateCompatibleBitmap(&dc, nWidth, nHeight);

  CDC memDC1, memDC2;
  memDC1.CreateCompatibleDC(&dc);
  memDC2.CreateCompatibleDC(&dc);
  CBitmap* pOldBitmap1 = memDC1.SelectObject(&bitmap1);
  CBitmap* pOldBitmap2 = memDC2.SelectObject(&bitmap2);

  CPalette* pOldPalette1 = NULL;
  CPalette* pOldPalette2 = NULL;
  CPalette grayPalette;

  if (pPalette->m_hObject != NULL) {
    LOGPALETTE* pLP = CreateGrayScale();
```

```
    grayPalette.CreatePalette (pLP);
    delete[] pLP;

    pOldPalette1 = memDC1.SelectPalette (&grayPalette, FALSE);
    pOldPalette2 = memDC2.SelectPalette (pPalette, FALSE);
    memDC1.RealizePalette ();
    memDC2.RealizePalette ();
  }

  //
  // Copy the bitmap to memDC2.
  //
  CDC memDC3;
  memDC3.CreateCompatibleDC (&dc);
  pCriticalSection->Lock ();
  CBitmap* pOldBitmap3 = memDC3.SelectObject (pBitmap);
  memDC2.BitBlt (0, 0, nWidth, nHeight, &memDC3, 0, 0, SRCCOPY);
  memDC3.SelectObject (pOldBitmap3);
  pCriticalSection->Unlock ();

  //
  // Convert the colors in memDC2 to shades of gray in memDC1.
  //
  int x, y;
  COLORREF crColor;
  BYTE grayLevel;

  for (y = 0; y < nHeight; y++) {
    for (x = 0; x < nWidth; x++) {
      crColor = memDC2.GetPixel (x, y);
      grayLevel = (BYTE)
        (((((UINT) GetRValue (crColor)) * 30) +
        (((UINT) GetGValue (crColor)) * 59) +
        (((UINT) GetBValue (crColor)) * 11)) / 100);
      memDC1.SetPixel (x, y,
        PALETTERGB (grayLevel, grayLevel, grayLevel));
    }

    //
    // Kill the thread if the pKillEvent event is signaled.
    //
    if (::WaitForSingleObject (pKillEvent->m_hObject, 0) ==
      WAIT_OBJECT_0) {

      memDC1.SelectObject (pOldBitmap1);
      memDC2.SelectObject (pOldBitmap2);
```

```
        if (pPalette->m_hObject != NULL) {
          memDC1.SelectPalette (pOldPalette1, FALSE);
          memDC2.SelectPalette (pOldPalette2, FALSE);
        }
        pWnd->PostMessage (WM_USER_THREAD_ABORTED, y + 1, 0);
        return (UINT) -1;
      }
      pWnd->SendMessage (WM_USER_THREAD_UPDATE, y + 1, nHeight);
    }

    //
    // Copy the gray scale image over the original bitmap.
    //
    CPalette* pOldPalette3 = NULL;
    if (pPalette->m_hObject != NULL) {
      pOldPalette3 = memDC3.SelectPalette (&grayPalette, FALSE);
      memDC3.RealizePalette ();
    }
    pCriticalSection->Lock ();
    pOldBitmap3 = memDC3.SelectObject (pBitmap);
    memDC3.BitBlt (0, 0, nWidth, nHeight, &memDC1, 0, 0, SRCCOPY);
    memDC3.SelectObject (pOldBitmap3);
    pCriticalSection->Unlock ();

    //
    // Clean up the memory DCs.
    //
    memDC1.SelectObject (pOldBitmap1);
    memDC2.SelectObject (pOldBitmap2);

    if (pPalette->m_hObject != NULL) {
      memDC1.SelectPalette (pOldPalette1, FALSE);
      memDC2.SelectPalette (pOldPalette2, FALSE);
      memDC3.SelectPalette (pOldPalette3, FALSE);
    }

    //
    // Tell the frame window we're done.
    //
    pWnd->PostMessage (WM_USER_THREAD_FINISHED, 0, 0);
    return 0;
  }

  LOGPALETTE* CreateGrayScale ()

    UINT nSize = sizeof (LOGPALETTE) + (sizeof (PALETTEENTRY) * 63);
```

```
    LOGPALETTE * pLP = (LOGPALETTE * ) new BYTE[nSize];

    pLP -> palVersion = 0x300;
    pLP -> palNumEntries = 64;

    for (int i = 0; i < 64; i ++ ) {
      pLP -> palPalEntry[i].peRed = i * 4;
      pLP -> palPalEntry[i].peGreen = i * 4;
      pLP -> palPalEntry[i].peBlue = i * 4;
      pLP -> palPalEntry[i].peFlags = 0;
    }
    return pLP;
}
```

ImageEditView.h

```
// ImageEditView.h : interface of the CImageEditView class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(
  AFX_IMAGEEDITVIEW_H__9D77AEEC_AA14_11D2_8E53_006008A82731__INCLUDED_)
#define
  AFX_IMAGEEDITVIEW_H__9D77AEEC_AA14_11D2_8E53_006008A82731__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif //_MSC_VER > 1000

class CImageEditView : public CScrollView
{
protected: // create from serialization only
  CImageEditView();
  DECLARE_DYNCREATE(CImageEditView)

// Attributes
public:
  CImageEditDoc * GetDocument();

// Operations
public:

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CImageEditView)
  public:
  virtual void OnDraw(CDC * pDC); // overridden to draw this view
```

```
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    protected:
    virtual void OnInitialUpdate(); // called first time after construct
    //{{AFX_VIRTUAL

  // Implementation
  public:
    virtual ~CImageEditView();
  #ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
  #endif

  protected:

  // Generated message map functions
  protected:
    //{{AFX_MSG(CImageEditView)
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
  };

  #ifndef _DEBUG // debug version in ImageEditView.cpp
  inline CImageEditDoc* CImageEditView::GetDocument()
  { return (CImageEditDoc*)m_pDocument; }
  #endif

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

  //{{AFX_INSERT_LOCATION}}
  // Microsoft Visual C++ will insert additional declarations
  // immediately before the previous line.

  #endif
  // !defined(
  //   AFX_IMAGEEDITVIEW_H__9D77AEEC_AA14_11D2_8E53_006008A82731__INCLUDED_)
```

ImageEditView.cpp

```
// ImageEditView.cpp : implementation of the CImageEditView class
//

#include "stdafx.h"
#include "ImageEdit.h"

#include "ImageEditDoc.h"
#include "ImageEditView.h"
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CImageEditView

IMPLEMENT_DYNCREATE(CImageEditView, CScrollView)

BEGIN_MESSAGE_MAP(CImageEditView, CScrollView)
  //{{AFX_MSG_MAP(CImageEditView)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CImageEditView construction/destruction

CImageEditView::CImageEditView()
{
}

CImageEditView::~CImageEditView()
{
}

BOOL CImageEditView::PreCreateWindow(CREATESTRUCT& cs)
{
  return CScrollView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CImageEditView drawing

void CImageEditView::OnDraw(CDC* pDC)
{
  CImageEditDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  CBitmap* pBitmap = pDoc->GetBitmap();

  if (pBitmap != NULL) {
    CPalette* pOldPalette;
    CPalette* pPalette = pDoc->GetPalette();

    if (pPalette != NULL) {
      pOldPalette = pDC->SelectPalette(pPalette, FALSE);
      pDC->RealizePalette();
```

```
    }

    DIBSECTION ds;
    pBitmap->GetObject(sizeof(DIBSECTION), &ds);

    CDC memDC;
    memDC.CreateCompatibleDC(pDC);
    CBitmap* pOldBitmap = memDC.SelectObject(pBitmap);

    pDC->BitBlt(0, 0, ds.dsBm.bmWidth, ds.dsBm.bmHeight, &memDC,
      0, 0, SRCCOPY);

    memDC.SelectObject(pOldBitmap);

    if (pPalette != NULL)
      pDC->SelectPalette(pOldPalette, FALSE);
  }
}

void CImageEditView::OnInitialUpdate()
{
  CScrollView::OnInitialUpdate();

  CString string;
  CSize sizeTotal;
  CBitmap* pBitmap = GetDocument()->GetBitmap();

  //
  // If a bitmap is loaded, set the view size equal to the bitmap size.
  // Otherwise, set the view's width and height to 0.
  //
  if (pBitmap != NULL) {
    DIBSECTION ds;
    pBitmap->GetObject(sizeof(DIBSECTION), &ds);
    sizeTotal.cx = ds.dsBm.bmWidth;
    sizeTotal.cy = ds.dsBm.bmHeight;
    string.Format(_T("\t%d x %d, %d bpp"), ds.dsBm.bmWidth,
      ds.dsBm.bmHeight, ds.dsBmih.biBitCount);
  }
  else {
    sizeTotal.cx = sizeTotal.cy = 0;
    string.Empty();
  }

  AfxGetMainWnd()->SendMessage(WM_USER_UPDATE_STATS, 0,
    (LPARAM)(LPCTSTR)string);
  SetScrollSizes(MM_TEXT, sizeTotal);
}
```

```
/////////////////////////////////////////////////////////////////////////////
// CImageEditView diagnostics

#ifdef _DEBUG
void CImageEditView::AssertValid() const
{
  CScrollView::AssertValid();
}

void CImageEditView::Dump(CDumpContext& dc) const
{
  CScrollView::Dump(dc);
}

CImageEditDoc* CImageEditView::GetDocument() // non-debug version is inline
{
  ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CImageEditDoc)));
  return (CImageEditDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CImageEditView message handlers
```

SpecialStatusBar.h

```
// SpecialStatusBar.h: interface for the CSpecialStatusBar class.
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(
  AFX_SPECIALSTATUSBAR_H__4BA7D301_AA24_11D2_8E53_006008A82731__INCLUDED_)
#define
  AFX_SPECIALSTATUSBAR_H__4BA7D301_AA24_11D2_8E53_006008A82731__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif //_MSC_VER > 1000

class CSpecialStatusBar : public CStatusBar
{
public:
  void SetProgress(int nPercent);
  void SetImageStats(LPCTSTR pszStats);
```

```
    CSpecialStatusBar();
    virtual ~CSpecialStatusBar();

protected:
    CProgressCtrl m_wndProgress;
    afx_msg int OnCreate(LPCREATESTRUCT lpcs);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    DECLARE_MESSAGE_MAP()
};

#endif
// !defined(
//AFX_SPECIALSTATUSBAR_H__4BA7D301_AA24_11D2_8E53_006008A82731__INCLUDED_)
```

SpecialStatusBar.cpp

```
// SpecialStatusBar.cpp: implementation of the CSpecialStatusBar class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "ImageEdit.h"
#include "SpecialStatusBar.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

BEGIN_MESSAGE_MAP(CSpecialStatusBar, CStatusBar)
    ON_WM_CREATE()
    ON_WM_SIZE()
END_MESSAGE_MAP()

CSpecialStatusBar::CSpecialStatusBar()
{
}

CSpecialStatusBar::~CSpecialStatusBar()
{
}
```

```
int CSpecialStatusBar::OnCreate (LPCREATESTRUCT lpcs)
{
  static UINT nIndicators[] =
  {
    ID_SEPARATOR,
    ID_SEPARATOR,
    ID_SEPARATOR
  };

  if (CStatusBar::OnCreate (lpcs) == -1)
    return -1;

  //
  // Add panes to the status bar.
  //
  SetIndicators (nIndicators, sizeof (nIndicators) / sizeof (UINT));

  //
  // Size the status bar panes.
  //
  TEXTMETRIC tm;
  CClientDC dc (this);
  CFont * pFont = GetFont ();

  CFont * pOldFont = dc.SelectObject (pFont);
  dc.GetTextMetrics (&tm);
  dc.SelectObject (pOldFont);

  int cxWidth;
  UINT nID, nStyle;
  GetPaneInfo (1, nID, nStyle, cxWidth);
  SetPaneInfo (1, nID, nStyle, tm.tmAveCharWidth * 24);
  GetPaneInfo (2, nID, nStyle, cxWidth);
  SetPaneInfo (2, nID, SBPS_NOBORDERS, tm.tmAveCharWidth * 24);

  //
  // Place a progress control in the rightmost pane.
  //
  CRect rect;
  GetItemRect (2, &rect);
  m_wndProgress.Create (WS_CHILD|WS_VISIBLE|PBS_SMOOTH,
    rect, this, -1);
  m_wndProgress.SetRange (0, 100);
  m_wndProgress.SetPos (0);
  return 0;
}
```

```
void CSpecialStatusBar::OnSize (UINT nType, int cx, int cy)
{
  CStatusBar::OnSize (nType, cx, cy);

  //
  // Resize the rightmost pane to fit the resized status bar.
  //
  CRect rect;
  GetItemRect (2, &rect);
  m_wndProgress.SetWindowPos (NULL, rect.left, rect.top,
    rect.Width (), rect.Height (), SWP_NOZORDER);
}

void CSpecialStatusBar::SetImageStats(LPCTSTR pszStats)
{
  SetPaneText (1, pszStats, TRUE);
}

void CSpecialStatusBar::SetProgress(int nPercent)
{
  ASSERT (nPercent >= 0 && nPercent <= 100);
  m_wndProgress.SetPos (nPercent);
}
```

图 17-8　ImageEdit 应用程序

ImageEdit 程序是一个实际的例子，它解决了工作者线程如何让文档对象知道它何时结束的问题。当从 Effects 菜单中选择了 Convert To Gray Scale 命令后，文档的 OnGrayScale 函数将启动执行 ThreadFunc 函数的后台线程。ThreadFunc 处理位图并在结束之前给应用程序的主窗口发布 WM_USER_THREAD_FINISHED 消息。接下来主窗口调用文档的 ThreadFinished 函数通知文档位图转换已经完成，而且 ThreadFinished 将调用 UpdateAllViews。

给主窗口发布消息并向下调用文档对象与让线程直接调用文档对象中的函数不同，因为 PostMessage 调用会将控制虚拟地传送给主线程。如果 ThreadFunc 自己调用了文档对象，那么就会在后台线程的描述体中调用 UpdateAllViews 并导致失败。

一个好方法是，每次完成位图中另一行的转换时，ThreadFunc 都会给主窗口发送一个 WM_USER_THREAD_UPDATE 消息。主窗口响应时会把进度控件嵌入状态栏，这样用户就不会对何时将出现已转换的图像而感到迷惑了。WM_USER_THREAD_UPDATE 消息是被发送而不是被发布的，这就确保了可以实时更新进度控件。如果 WM_USER_THREAD_UPDATE 消息被发布而不是被发送，那么后台线程发布消息的速度可能会快于在很快的 CPU 上主窗口处理消息的速度。

ImageEdit 使用了两个线程同步化对象：CEvent 对象 m_event 和 CCriticalSection 对象 m_cs。这两个都是文档类中的成员，都是在 THREADPARMS 结构中通过地址传递给线程函数的。在用户从 Effects 菜单中选择了 Stop Gray Scale Conversion 命令而希望停止灰度转换过程时，事件对象用来结束工作者线程。为了结束线程，主线程将事件设置为了信号发出状态：

```
m_event.SetEvent ();
```

在每次完成一行扫描之后，ThreadFunc 中的转换例程就会检查事件对象，如果事件进入了信号发出状态，则结束线程：

```
if (::WaitForSingleObject (pKillEvent->m_hObject, 0) ==
  WAIT_OBJECT_0) {
  .
  .
  .
  pWnd->PostMessage (WM_USER_THREAD_ABORTED, y + 1, 0);
  return (UINT) -1;
}
```

WM_USER_THREAD_ABORTED 消息警告主窗口线程已经中止。主窗口会调用 CImageEditDoc::ThreadAborted 来通知文档，只有在线程没有完全终止时，ThreadAborted 才会阻塞在线程句柄上。然后复位内部标志，表明线程不再运行。

临界区防止了应用程序的两个线程试图同时将位图选入设备描述表。在视图需要更新时主线程将位图选入设备描述表，后台线程在灰度转换开始时将位图选入内存设备描述表一次而在转换结束时又选入一次。位图只能一次选入一个设备描述表，因此任何一个线程如果在另一个线程已经将位图选入设备描述表以后还要试图选入，那么其中的一个线程就会失败。(但是，另一方面调色板可以同时被选入几个设备描述表中，并且在可调色板化的设备上执行灰度转换时 ThreadFunc 就利用了这一点。)两个线程试图同时选用位图的情况出现的几率很小，但是使用临界区可以保证程序在执行对 SelectObject 调用的过程中不会被其他线程调用 SelectObject 而打断。位图在被选入的设备描述表中呆的时间很短，是不会被感觉到的，因此如果临界区被锁定，线程也不会等待很长时间。

ImageEdit 还说明了如何将进度控件放入状态栏。ImageEdit 的状态栏是 CSpecialStatusBar 的一个实例，该类是从 CStatusBar 派生来的。CSpecialStatusBar::OnCreate 给状态栏添加了三个窗格。然后它创建了一个进度控件并把它非常合适地放置在了最右边的窗格里。由于在状态栏缩放时状态栏中窗格的大小和位置也会改变，所以 CSpecialStatusBar 还包含了一个 OnSize 处理程序，用来将进度控件始终都调整在最右边的窗格中。结果是进度控件看上去

好像是一个普通的状态栏窗格，只有调用了 CProgressCtrl::SetPos 后它才会出现。

17.3　小知识点

关于多任务和多线程，这里还有些小知识点或许对您有用。

17.3.1　消息泵

编程人员常误以为多线程可以使应用程序运行得更快。在单处理机上，多线程不会加快应用程序的运行速度；然而，它确实使应用程序的"响应更快"了。有一种方法可以显示多线程对响应速度的作用。编写一个应用程序，让它响应某菜单命令，画几千个椭圆。如果由主线程画椭圆，则主线程便不能抽空检查消息队列和分派等待消息，因而输入就会被冻结，直到画椭圆循环执行完毕。如果应用程序中画椭圆部分由独立线程完成，则在画椭圆时，应用程序还能响应用户的输入。

然而在这样简单的应用程序中使用多线程未免小题大做。另一个解决方案是：主线程画椭圆时，可以使用"消息泵"保证消息的流动。假定画椭圆的消息处理程序是这样的：

```
void CMainWindow::OnStartDrawing()
{
  for (int i = 0; i < NUMELLIPSES; i++)
    DrawRandomEllipse();
}
```

如果 NUMELLIPSES 是很大的数字，则 for 语句循环一旦启动，程序就会停滞较长的时间。针对这种情况，可以试着添加另一个菜单命令，设置一个标志并截断 for 语句循环，如下所示：

```
void CMainWindow::OnStartDrawing()
{
  m_bQuit = FALSE;
  for (int i = 0; i < NUMELLIPSES && !m_bQuit; i++)
    DrawRandomEllipse();
}

void CMainWindow::OnStopDrawing()
{
  m_bQuit = TRUE;
}
```

但是这还不行。为什么呢？因为只要 OnStartDrawing 中的 for 循环语句不经提取消息就执行

命令,则用来激活 OnStopDrawing 的 WM_COMMAND 消息也无法起作用。实际上,在 for 循环语句执行时,菜单是打不开的。

如果使用消息泵,这个问题就能很好解决。在单线程 MFC 程序中,最好用下面这种方法执行较冗长的过程:

```
void CMainWindow::OnStartDrawing ()
{
  m_bQuit = FALSE;
  for (int i = 0; i < NUMELLIPSES && !m_bQuit; i++) {
    DrawRandomEllipse ();
    if (!PeekAndPump ())
      break;
  }
}

void CMainWindow::OnStopDrawing ()
{
  m_bQuit = TRUE;
}

BOOL CMainWindow::PeekAndPump ()
{
  MSG msg;
  while (::PeekMessage (&msg, NULL, 0, 0, PM_NOREMOVE)) {
    if (!AfxGetApp ()->PumpMessage ()) {
      ::PostQuitMessage (0);
      return FALSE;
    }
  }
  LONG lIdle = 0;
  while (AfxGetApp ()->OnIdle (lIdle++));
  return TRUE;
}
```

PeekAndPump 在一个消息循环中制定另一个消息循环。它在 OnStartDrawing 中 for 语句的循环终点处被调用。如果::PeekMessage 指示队列中有消息等待,则 PeekAndPump 首先调用 CWinThread::PumpMessage 提取和分派消息。如果 PumpMessage 返回 0,则表示提取和分派的最后一个消息是 WM_QUIT 消息。而因为只有用"主"消息循环提取 WM_QUIT 消息,应用程序才能结束,所以该消息要求特殊处理。因此如果 PumpMessage 返回 0,PeekAndPump 就会把另一个 WM_QUIT 消息发往队列;如果 PeekAndPump 返回 0,OnStartDrawing 中的 for 语句循环就会失败。如果 WM_QUIT 消息不提示提前退出,那么通过在返回之前调用应用程

序对象的 OnIdle 函数，PeekAndPump 就可模仿主结构的闲置机制。

如果画椭圆的循环中有了 PeekAndPump，用来激活 OnStopDrawing 的 WM_COMMAND 消息就能被正常提取和分派了。因为 OnStopDrawing 把 m_bQuit 设置为 TRUE，所以在画下一个椭圆之前，该循环就会失败。

17.3.2　执行其他进程

Win32 进程执行其他进程和执行线程一样容易。下列语句执行 c:\Windows\Notepad.exe。

```
STARTUPINFO si;
::ZeroMemory (&si, sizeof (STARTUPINFO));
si.cb = sizeof (STARTUPINFO);
PROCESS_INFORMATION pi;

if (::CreateProcess (NULL,_T ("C: \\ Windows \\ Notepad"), NULL,
  NULL, FALSE, NORMAL_PRIORITY_CLASS, NULL, NULL, &si, &pi)) {
  ::CloseHandle (pi.hThread);
  ::CloseHandle (pi.hProcess);

}
```

::CreateProcess 是一个通用函数，它获取可执行文件的名字（和路径），然后加载并执行它。如果可执行文件名中的驱动器和目录名被省略，则系统自动在 Windows 目录、Windows 系统目录、当前路径下的所有目录和选中的其他位置中搜索该文件。文件名也可以包含命令行参数，如：

```
"C: \\ Windows \\ Notepad C: \\ Windows \\ Desktop \\ Ideas.txt"
```

::CreateProcess 将进程的关键信息填充在 PROCESS_INFORMATION 结构中。相关信息包括：进程句柄（hProcess）和进程中主线程的句柄（hThread）。在进程启动后，要用::CloseHandle 关闭这些句柄。如果不再用这些句柄，只要::CreateProcess 一返回，就可以关闭它们。

如果::CreateProcess 返回非零值，则意味着进程启动成功。因为 Win32 是异步启动、异步执行的，所以::CreateProcess 不必等到进程结束后再返回。如果您希望启动另一个进程，并暂停当前进程直到该进程启动的进程结束，则您可以对该进程句柄调用::WaitForSingleObject，如下所示：

```
STARTUPINFO si;
::ZeroMemory (&si, sizeof (STARTUPINFO));
si.cb = sizeof (STARTUPINFO);
PROCESS_INFORMATION pi;
```

```
if (::CreateProcess (NULL,_T ("C: \\ Wincows \\ Notepad"), NULL,
  NULL, FALSE, NORMAL_PRIORITY_CLASS, NULL, NULL, &si, &pi)) {
  ::CloseHandle (pi.hThread);
  ::WaitForSingleObject (pi.hProcess, INFINITE);
  ::CloseHandle (pi.hProcess);
}
```

进程和线程一样都有退出代码。如果::WaitForSingleObject 没有返回 WAIT_FAILED,则可以调用::GetExitCodeProcess 获取进程的退出代码。

有时需要启动进程并等待足够长的时间后,才能确保进程已开始并响应用户输入。例如:如果进程 A 启动进程 B,而进程 B 又创建了一个窗口,这时,如果进程 A 要给窗口发送消息,它就不得不等到::CreateProcess 返回,留给进程 B 足够的时间创建窗口并开始处理消息。通过 Win32 ::WaitForInputIdle 函数则可以轻松地解决这个问题:

```
STARTUPINFO si;
::ZeroMemory (&si, sizeof (STARTUPINFO));
si.cb = sizeof (STARTUPINFO);
PROCESS_INFORMATION pi;

if (::CreateProcess (NULL,_T ("C: \\ Windows \\ Notepad"), NULL,
  NULL, FALSE, NORMAL_PRIORITY_CLASS, NULL, NULL, &si, &pi)) {
  ::CloseHandle (pi.hThread);
  ::WaitForInputIdle (pi.hProcess, INFINITE);
  // Get B's window handle and send or post a message.
  ::CloseHandle (pi.hProcess);
}
```

直到指定进程开始处理消息并清空它的消息队列,::WaitForInputIdle 才允许当前进程继续执行。因为实现进程句柄到窗口句柄转换的 MFC 或 API 函数都很复杂,所以本书没有提供搜索窗口句柄的程序代码。还有另一种方法可用,但是它要求您必须根据该进程的已知特性,借助于::EnumWindows、::FindWindow 或相关函数搜索该进程的窗口。

17.3.3 文件改变通知

在本章前面,提到传递给::WaitForSingleObject 的 HANDLE 参数可以是"文件改变通知句柄"。Win32 API 包含一个函数::FindFirstChangeNotification,只要给定目录或它的了目录发生了变化,例如文件被重命名或删除,或创建了一个新目录,该函数都能返回一个句柄,通过该句柄您可以启动一个被阻塞的线程。

如果想改善第 11 章中的 Wanderer 应用程序,使文件系统的变化能立刻反映在左面或右面的窗格中。为此,最有效的方法是启动一个后台线程,并使它在一个或多个文件改变通知

句柄上处于阻塞状态。下面是用来监视驱动器 C: 的线程的线程函数：

```
UINT ThreadFunc (LPVOID pParam)
{
  HWND hwnd = (HWND) pParam; // Window to notify
  HANDLE hChange = ::FindFirstChangeNotification ( _T("C:\\"),
    TRUE, FILE_NOTIFY_CHANGE_FILE_NAME | FILE_NOTIFY_CHANGE_DIR_NAME);

  if (hChange == INVALID_HANDLE_VALUE) {
    TRACE (_T("Error: FindFirstChangeNotification failed\n"));
    return (UINT) -1;
  }

  while (...) {
    ::WaitForSingleObject (hChange, INFINITE);
    ::PostMessage (hwnd, WM_USER_CHANGE_NOTIFY, 0, 2);
    ::FindNextChangeNotification (hChange); // Reset
  }
  ::FindCloseChangeNotification (hChange);
  return 0;
}
```

传递给::FindFirstChangeNotification 的第 1 个参数指定了待监视目录，第 2 个参数确定要监视指定目录(FALSE)还是监视指定目录和它的所有子目录(TRUE)，第 3 个参数指定线程应接收到的几个变化种类。本样例中，文件在 C: 驱动器的任意地方被创立、重命名或删除(FILE_NOTIFY_CHANGE_FILE_NAME)时，该线程就会被启动；目录被创建、重命名或删除(FILE_NOTIFY_CHANGE_DIR_NAME)时，该线程也被启动。当该线程被启动时，它向窗口发送用户自定义消息，其中窗口的句柄是用 pParam 传递的。该消息的 lParam 保存驱动器号(驱动器 C: 的编号是 2)。假定接收该消息的窗口是应用程序的顶层框架窗口，则它可以通过更新自己的视图响应这个消息。记住：如果线程是由文件改变而被通知启动的，那么它不会接收到与改变的性质以及改变发生所在地有关的信息。所以如果它想知道引发文件改变通知的原因，必须扫描文件系统。

如果想让线程不只监视一个驱动器，那么通过构造线程就可以实现。您所要做的是对每个驱动器调用一次::FindFirstChangeNotification，获得每个驱动器的不同文件改变通知句柄，而后使用::WaitForMultipleObjects 同时在所有文件改变通知上进入阻塞状态。::WaitForMultipleObjects是 CMultiLock::Lock 在 Win32 API 中的等价函数。在 WaitForMultipleObjects 的第三个参数为 FALSE 时，调用它则等于指示系统：如果被线程阻塞的对象接到通知，则启动线程。

第Ⅳ部分

COM、OLE 和 ActiveX

第 18 章　MFC 和组件对象模型

当 MFC 还在其初始阶段时，那些偏好 MFC 而离开 Microsoft Windows API 的 C++ 程序员就希望在开发 Windows 应用程序时可以得到类库的帮助。当时传统的有识之士认为 MFC 可以使 Windows 程序设计更容易，但事实上 Windows 程序设计依旧是 Windows 程序设计。MFC 简化了开发过程的某些方面，也减轻了将 16 位 Windows 应用程序移植到 32 位系统中的痛苦，而这一点在早些时候几乎没有哪个程序员可以预见到。但是即使是 MFC 也不能声称它减少了著名的学习 Windows 的困难。当时是这样，今天依然是如此。

今天有了一个更有说服力的要使用 MFC 的理由。如果您开发的应用程序用到了 COM、OLE 或 ActiveX，那么 MFC 可以极大地简化开发过程。我的意思是可以大量地减少开发一个应用程序（或软件组成部分）所需要的时间。现在因为有了非常好的类库可以使用，已经没有什么站得住脚的理由来从头开始开发某种软件了。COM、OLE 和 ActiveX 由于过分复杂和极端地难于理解受到了很多批评，但是无论如何它总是生存下来了，并且如果您想编写 Windows 程序的话，极有可能在将来您会成为 COM 程序员。

那么什么是 COM、OLE 以及 ActiveX 呢？MFC 又如何使得对其进行编程更容易了呢？很高兴您提出这样的问题，在本章的剩余部分我们将讨论 MFC 对 COM 所给予的全部支持。在本章中，首先从定义 COM、OLE 和 ActiveX 开始，然后将介绍一些 MFC 用来捆绑它们的独特而有趣的方法。在以后各章中，我们将解决具体的基于 COM 的技术问题，如 Automation 和 ActiveX 控件，您会看到如何用 MFC 来实现它们。

18.1　组件对象模型

COM 是组件对象模型（Component Object Model）的首字母缩略词。简单地说，COM 是一种方法，用于创建独立于任何编程语言的对象。如果您想得到详细资料，可以从 Microsoft 的 Web 站点下载 COM 的详细说明书。但不要急于打开您的浏览器：如果您是第一次接触 COM，详细规范就有些过于繁杂了。最好的办法是慢慢地开始，让自己有时间理解它的总体内容，而不是冒险去让自己纠缠于那些不重要的细节中。

C++ 程序员习惯于编写其他 C++ 程序员可以使用的类。而使用这些类的问题是“只有”其他 C++ 程序员可以使用它们。COM 告诉了我们如何用任意一种编程语言编写在任何编程语言中都能使用的类，换句话说，COM 超越了创建可重用对象的特定语言限制方式，对于对象体系结构来说，它给我们提供了真正的二进制标准。

C++ 类具有成员函数,而 COM 对象具有"方法"。方法被分别组织在"接口"中,并通过"接口指针"调用它们。接口将语义相关的方法组合在一起。例如,假设您正在编写一个 COM 类,它具有名为 Add、Subtract 和 CheckSpelling 的方法。此时不应该把这三个方法都作为同一个接口的成员,而是应该将 Add 和 Subtract 分配给接口 IMath,将 CheckSpelling 分配给接口 ISpelling。(在接口名字前面加一个大写的 I 来代表"接口"已经成为通用的 COM 程序设计习惯了。)Microsoft 已经预定义了 100 多个接口,任何 COM 对象都可以支持它们。这些接口被称为"标准接口"。用户定义的接口如 IMath 和 ISpelling 叫做"自定义接口"。COM 对象可以使用标准接口、自定义接口以桨两种接口的组合。

每一种 COM 对象都实现了一个名为 IUnknown 的接口。IUnknown 只包含 3 个方法,如表 18-1 所示。

表 18-1 IUnknown 的 3 个方法

方法名称	说 明
QueryInterface	返回指向另一个接口的指针
AddRef	增加对象的引用计数
Release	减小对象的引用计数

COM 的一条规则规定:给定一个指向接口的指针,客户就可以通过该指针调用任何一个 IUnknown 方法,以及所有属于该接口的特殊方法。换句话说,所有接口除了支持它们自己的方法以外还必须支持这 3 个 IUnknown 方法。这就意味着如果定义了一个带有方法 Add 和 Subtract 的 IMath 接口,该接口实际上已经包含 5 个方法了:QueryInterface、AddRef、Release、Add 以及 Subtract。大多数对象并不单独实现 IUnknown 接口。因为所有接口都包含有 IUnknown 的方法,对于大多数接口而言,如果要请求一个 IUnknown 指针,它们就会简单地返回一个指向其他接口的指针。

图 18-1 是一个简单 COM 对象的示意图。图中的细棒,有时称为"棒棒糖",代表对象的接口。通常都会忽略 IUnknown 棒棒糖,因为大家都知道所有 COM 对象都实现 IUnknown 了。

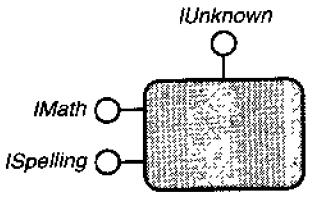

图 18-1 一个简单的 COM 对象

我一直使用易于理解的名字如 IMath 来指代接口,但实际上接口是用数字而不是用名字标识的。"接口标识"(或称为 IID)是一个 128 位的值,它唯一地标识了每一个接口。有如此众多不同的 128 位数值,您和我任意选择而得到相同 IID 的概率实际上等于 0。所以,如果地球上不同地方的两个人定义了自定义接口 IMath 的不兼容的版本,那也没关系。关键是两个 IMath 接口具有不同的 IID。

Microsoft Visual C++ 带有两个工具用来生成 IID。一个是命令行工具 Uuidgen;另一个是 GUI 应用程序 Guidgen。即使将某些变量如您的网卡 Ethernet ID 以及日期等考虑进去,两个工具也都会使它们生成的 128 位数值具有最大的任意性。也可以使用 COM API 函数

CoCreateGuid通过编程生成 IID。CoCreateGuid 中的 Guid 代表“全局唯一标识符”，是描述 128 位标识的一般术语。IID 仅仅是一个特殊的 GUID。

18.1.1　实例化 COM 对象

COM 类像接口一样也是由 128 位的值标识的。标识类的 GUID 被称为“类 ID”或 CLSID。为实例化一个对象，客户仅需要知道该对象的 CLSID 就可以了。COM 具有自己的 API，其中包括“激活函数”用来创建对象实例。最常用的激活函数是 CoCreateInstance，它接受一个 CLSID 并返回指向对象的接口指针。下列语句实例化了一个 CLSID 为 CLSID_Object 的 COM 类，并将指向对象的 IMath 接口的指针保存在了 pMath 中：

```
IMath * pMath;
CoCreateInstance (CLSID_Object, NULL,
 CLSCTX_SERVER, IID_IMath, (void * *) &pMath);
```

其中的 IID_IMath 只是一个变量，用来保存 IMath 的 128 位接口 ID。

一旦有了接口指针，C++ 客户就可以使用 -> 运算符来调用该接口上的方法了。下列语句调用 IMath::Add 相加一对数值：

```
int sum;
pMath -> Add (2, 2, &sum);
```

Add 并不直接返回两个输入值的和；它将结果复制到由调用者指定的地址中，在本例中是变量 sum 的地址。这是因为 COM 方法要返回专门的 32 位值，称为 HRESULT。HRESULT 可以告诉调用者调用是成功还是失败。如果调用不成功的话，它还能提供有关失败的详细的信息。您或许会认为像 Add 这样简单的方法永远不会失败，可是实际上失败是可能的。如果实现方法的对象正运行在远程网络服务器上，而且由于电缆没有连接致使客户不能与服务器取得连接，那操作就会失败。如果这种情况发生，系统就会介入并返回 HRESULT 通知调用者调用没有成功。

COM 通常使初学者感到困惑的一个方面是每个对外可创建的 COM 类（也就是可以通过把 CLSID 传递给 CoCreateInstance 进行实例化的 COM 类）都伴随有一个“类对象”。类对象也是一个 COM 对象。它生存的唯一目的是为了创建其他 COM 对象。把 CLSID 传递给 CoCreateInstance 好像是直接实例化了对象，但在内部，CoCreateInstance 首先要实例化对象的类对象，然后再请求类对象去创建对象。大多数类对象都实现了一个特殊的 COM 接口 IClassFactory（或是 IClassFactory2，该接口的新版本，在功能上它是 IClassFactory 的扩展集）。实现 IClassFactory 的类对象被称为“类工厂”。给定一个 IClassFactory 接口指针，客户就可以通过调用 IClassFactory::CreateInstance 来创建对象实例了。这个方法（CreateInstance）可以看做是在 COM 中与 C++ 中的 new 运算符等价。

并不是所有的 COM 类都是对外可创建的。一些类专门供私有使用，不能用CoCreateInstance

进行初始化,因为它们没有 CLSID 也没有类工厂。C++ 程序员通过在实现对象的C++ 类上调用 new 来实例化这些对象。通常,这些对象参与实现基于 COM 的协议(如拖放数据传输)。一些 MFC 的 COM 类符合这个特点。在我们讨论 MFC 所支持的多种 COM 和 ActiveX 技术时您就会进一步理解了。

18.1.2 对象生存期

C++ 程序员习惯于使用 C++ new 运算符来创建基于堆的对象。他们还习惯于调用 delete 来删除用 new 创建的对象。在此方面 COM 与 C++ 不同,客户创建对象实例但却不去删除它们。相反是由 COM 对象来删除它们。为什么?

假设有两个以上的客户正在使用对象的同一个实例。客户 A 创建了对象,客户 B 通过申请接口指针也与该对象相联结。如果客户 A 意识不到客户 B 的存在而删除了该对象,那么就给客户 B 留下一个不再指向任何对象的接口指针。由于 COM 客户通常并不知道(也不关心)它是否是对象的唯一使用者,所以 COM 就担负起了删除对象的责任。在由对象保存的内部引用计数降低到 0 时就开始执行删除操作。引用计数记录了保存有指向对象中接口指针的客户的数量。

对于在 C++ 中实现的 COM 类,引用计数通常都保存在成员变量中。引用计数在调用 AddRef 时递增,在调用 Release 时递减。(别忘了 AddRef 和 Release 是 IUnknown 方法,它们可以通过任何一个接口指针被调用。)AddRef 和 Release 的实现通常很简单,如下所示:

```
ULONG__stdcall CComClass::AddRef()
{
  return ++m_lRef;
}

ULONG__stdcall CComClass::Release()
{
  if(--m_lRef == 0){
    delete this;
    return 0;
  }
  return m_lRef;
}
```

在本例中,CComClass 是代表 COM 类的 C++ 类。m_lRef 是用来保存对象引用计数的成员变量。如果在完成使用接口以后每个客户都调用 Release,那么在最后一个客户调用Release时对象就会很方便地删除自身。

对于 AddRef 和 Release 的使用有一些协议。在任何提供接口指针的时候都要调用 AddRef,这是对象而不是客户的责任,而调用 Release 则是客户的责任。客户有时也调用 AddRef,说明它们正在复制接口指针。在这种情况下,当接口指针不再需要时,调用Release

仍然是客户(或那些得到客户提供的接口指针的拷贝者)的责任。

18.1.3　获得接口指针

前文我们研究的 CoCreateInstance 例子中创建了一个对象并且申请了一个 IMath 接口指针。现在假设对象还实现了 ISpelling。那么一个具有 IMath 指针的客户如何向对象申请 ISpelling指针呢?

这就体现出第 3 个 IUnknown 方法的用处了。给定一个指针,客户就可以通过该指针调用 QueryInterface 来获得对象所支持的其他所有接口的指针。程序代码如下:

```
IMath * pMath;
HRESULT hr = CoCreateInstance (CLSID_Object, NULL,
 CLSCTX_SERVER, IID_IMath, (void * *) &pMath);

if (SUCCEEDED (hr)) { // CoCreateInstance worked.
   .
   .
   .
   ISpelling * pSpelling;
   hr = pMath -> QueryInterface (IID_ISpelling, (void * *) &pSpelling);
   if (SUCCEEDED (hr)) {
      // Got the interface pointer!
      .
      .
      .
      pSpelling -> Release ();
   }
   pMath -> Release ();
}
```

注意,此时客户要检查由 CoCreateInstance 返回的 HRESULT,以确保激活请求成功。在对象创建后的某时刻,客户使用 QueryInterface 来请求一个 ISpelling 指针,它将再次检查 HRESULT 而不是简单地假定该指针有效。(SUCCEEDED 宏告诉客户 HRESULT 代码是成功还是失败。相关的宏 FAILED 可以用来检测失败。)在不再需要它们时,两个接口都会被释放。当通过 IMath指针调用 Release 时,如果没有其他客户拥有接口指针的话,对象将删除自己。

注意,不存在可以用来枚举对象所支持的所有接口的 COM 函数。假定客户知道对象所支持的接口,因此它可以使用 QueryInterface 来获取任意接口的指针。使用称为"类型信息"的机制,对象可以公布所支持接口的列表。一些 COM 对象允许它的客户使用类型信息,一些对象则不允许。某些 COM 对象类型,其中包括 ActiveX 控件,要求公布其类型信息。在第 21 章讨论 ActiveX 控件体系结构时您就会明白其中的原因了。

18.1.4 COM 服务器

如果在响应激活请求时 COM 要创建对象,那么它必须知道在哪里可以找到每个对象的可执行文件。实现 COM 对象的可执行文件就叫做"COM 服务器"。注册表中的 HKEY_CLASSES_ROOT\CLSID 部分所包含的信息将 CLSID 和可执行文件联在了一起。例如,如果有一个名为 MathSvr.exe 的服务器用来实现 Math 对象,客户用 Math 的 CLSID 来调用 CoCreateInstance,COM 就会在注册表中查找 CLSID,然后得到 MathSvr.exe 的路径并启动 EXE。接下来 EXE 会给 COM 提供一个类工厂,COM 将调用类工厂的 CreateInstance 方法来创建 Math 对象的实例。

COM 服务器具有两种基本类型:进程内类型和进程外类型。进程内类型服务器(通常称为 in-proc 服务器)是动态链接库(DLL)类型。它们被叫做 in-proc 的原因是由于在 Win32 环境下,DLL 的加载和运行都在与其客户相同的地址空间中。相反,EXE 类型则运行在分开的地址空间中,这些地址空间实际上是彼此隔离的。在大多数情况下,调用 in-proc 对象非常快速,它们只是调用内存中的其他地址。通过 in-proc 对象调用一个方法类似于在自己的应用程序中调用子例程。

进程外服务器(也称为 out-of-proc 服务器)是 EXE 类型。将 COM 对象封装在 EXE 中的好处是,运行在两个不同进程中的客户和对象可以彼此保护以防其中一个崩溃。缺点是速度问题。调用其他进程中的对象比调用 in-proc 对象大约要慢 1 000 倍,这是因为跨越进程边界调用方法会增加额外开销。

Microsoft Windows NT 4.0 引入了分布式 COM (DCOM),它给 out-of-proc 服务器提供了在远程网络服务器上运行的自由。可以很容易地在本地编写、测试以及调试一个 out-of-proc 服务器,然后再把它提供给网络上使用。(对于 Windows NT 4.0 Service Pack 2,in-proc 服务器也可以通过使用代理 EXE 程序来调用 DLL 的机制实现远程运行。)CoCreateInstance 和其他 COM 激活函数完全可以创建驻留在网络其他地方的对象。即使在 DCOM 出现以前编写的 COM 服务器,也可以通过少量的注册表修改而实现远程操作。

为区分运行在同一个设备上的服务于对象的 out-of-proc 服务器和运行在远程机器上的 out-of-proc 服务器,COM 程序员使用了术语"本地服务器"和"远程服务器"。本地服务器是 EXE 程序,运行在客户所运行的设备上;与它相比,远程服务器运行在网络上的其他地方。虽然在 in-proc 和 out-of-proc 服务器之间有重大结构上的差别,但是在本地和远程服务器之间却没有差别。用 DCOM 设计的对象专注于操作系统根本上的安全模型或是专注于改善执行效率。但是除了优化操作问题以外,本地服务器和远程服务器共同拥有相同的服务器和对象体系结构。

18.1.5 定位透明度

COM 的最强大的功能之一是定位透明度。简单地讲,定位透明度是指客户既不知道也

不关心对象的位置。无论对象是运行在客户的地址空间中还是运行在另一个进程或甚至另一台机器上，都可以用完全相同的指令序列通过该对象来调用方法。为实现定位透明度功能使用了大量的奇妙技巧，但是 COM 在其中做了大量的工作。

当调用其他线程或其他机器上的对象中的方法时，COM 会进行“远程调用”。在远程调用过程中，COM 将“调度”方法的参数和返回值。调度过程有多种形式，但最常见的一种调度类型基本上是将调用者的栈帧在调用接收者的地址空间中复制一份。“名称解析代理”和“桩基模块”(stub)执行了大部分的调度和远程处理工作。当给客户一个指向对象的接口指针时，如果该对象所运行的进程不是客户的进程，COM 就会在客户进程中创建一个接口名称解析代理，在服务器进程中创建一个接口桩基模块。客户拥有的接口指针实际上是指向名称解析代理的接口指针，它实现了与真实的对象相同的接口和方法。当客户调用对象中的方法时，调用传到名称解析代理，名称解析代理将使用某种进程间通信(IPC)将调用提交给桩基模块。桩基模块展开方法参数并调用对象，然后将返回值调度回名称解析代理。图 18-2 说明了客户、对象、名称解析代理以及桩基模块间的关系。

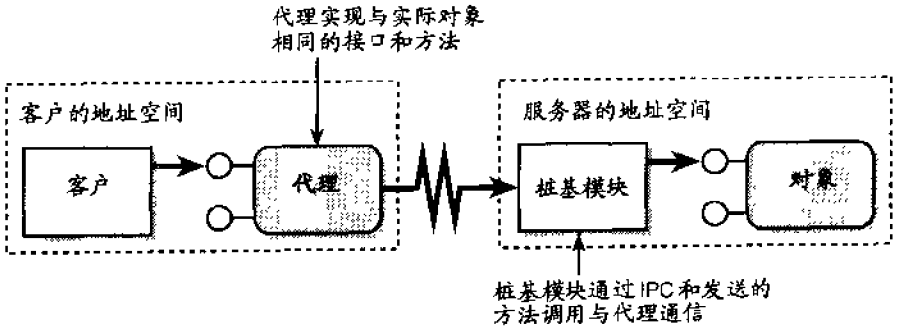

图 18-2　名称解析代理和桩基模块

名称解析代理和桩基模块是从哪里来的呢？如果对象仅仅使用了标准的接口，COM 就会提供名称解析代理和桩基模块。如果对象使用了自定义接口，那就应该由对象的实现者以名称解析代理/桩基模块 DLL 的形式来提供了。好消息是您几乎可以不必手工编写名称解析代理/桩基模块 DLL。Visual C++ 带有一个称为 MIDL (Microsoft Interface Definition Language，微软接口定义语言)的编译器，可以用来“编译”IDL (Interface Definition Language，接口定义语言)文件、为名称解析代理/桩基模块 DLL 生成源程序。坏消息是现在您又不得不学习另一种新语言 IDL 了。IDL 被叫做是 COM 的“通用语”。您越是熟悉 IDL，就越是有能力优化本地和远程服务器的执行效率。要想避免使用 IDL 和 MIDL，您可以选择另外一种调度策略“自定义调度”，但是自定义调度很难正确地实现，所以除非您有明确而有力的理由，否则还是应该选择名称解析代理和桩基模块方式。还有其他方法可以避免编写名称解析代理和桩基模块，但是您必须在灵活性和运行效率方面付出代价。Automation 就是其中的一种方

法,我们将在第 20 章讨论。

定位透明度的关键是当客户与运行在其他进程中的对象通信时,它们并不知道实际上是通过名称解析代理和桩基模块实现通信的。客户所知道的只是它有一个接口指针,并且通过此接口指针可以调用方法。现在您明白了吧?

18.1.6 对象链接和嵌入

在出现 COM 之前,已经存在对象的链接和嵌入了,更常提到的名字是其首字母缩略词 OLE。OLE 允许您将由一个应用程序创建的"内容对象"放入由另一个应用程序创建的文档中。OLE 的一种应用就是将 Excel 电子表格放在 Word 文档中(参见图 18-3)。在这种情况下,Excel 作为 OLE 服务器提供了嵌入或链接的电子表格对象(内容对象),Word 作为 OLE 容器来管理该对象。

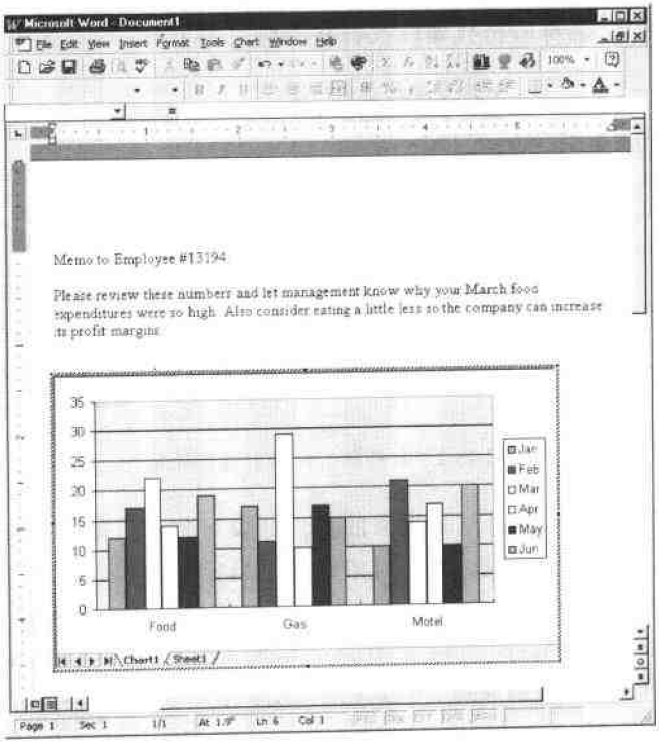

图 18-3 嵌入在 Microsoft Word 文档中的 Microsoft Excel 图表

OLE 是一种复杂的软件协议,描述了 OLE 服务器与 OLE 容器之间的对话过程。Microsoft在动态数据交换(DDE)的基础上创建了 OLE 1.0。DDE 被证明不是一个很理想的 IPC 机制,因此 Microsoft 发明了 COM 来作为 OLE 2.0 的基础 IPC 机制。在一段很长的时间

中，Microsoft 给所有新的 COM 技术都贴上了 OLE 标签：Automation 成为 OLE Automation、ActiveX控件被命名为 OLE 控件，等等。Microsoft 甚至想把 OLE 当做一个“词”，而不是首字母缩略形式。可是到了 1995 年由于术语 ActiveX 的诞生，Microsoft 开始反悔了，并且说：“我们改主意了；OLE 还是代表对象链接与嵌入。”尽管有了这样明确的反悔，可是许多程序员仍然（错误地）将术语 COM 和 OLE 相互替代使用。其实，它们并不是同义词。COM 是对象模型，形成了所有 OLE 和 ActiveX 技术的基础。OLE 则是一种技术，允许您将 Excel 电子表格放置在 Word 文档中。熟悉它们之间的关系可以使您避免许多程序员所具有的困扰。

那么 OLE 是如何使用 COM 的呢？当 OLE 服务器（如 Excel）给容器（如 Word）提供电子表格时，它将创建一个或多个 COM 对象来实现某种标准接口（如 IOleObject 和 IViewObject）。Word 也要创建 COM 对象来遵守公布的规则。这种结构具有一般性，它并不局限于 Word 和 Excel；其他任何应用程序都可以成为 OLE 容器或服务器，甚至同时既是容器又是服务器。容器和服务器通过指针调用方法来进行通信。虽然为了处理 Windows 的某种限制，一些 OLE 的 COM 接口必须在进程内实现，但是因为有了定位透明度，所以容器和服务器运行在不同的进程中也就没有关系了。因为设备描述表句柄在进程间是不可移植的（例如，当容器请求服务器在容器的窗口中绘制一个对象时），所以服务器的这部分功能必须在进程内实现。

图 18-4 显示了一个简单的嵌入容器的示意图。对于每个嵌入容器文档中的内容对象，容器都要实现一个“现场对象”（site object）。现场对象至少必须实现 COM 接口 IOleClientSite 和 IAdviseSink。为了与容器进行对话，服务器要通过指向这些接口的指针来调用方法。这个简单的图表掩饰了实际链接和嵌入服务器的内部复杂性，但是它却也说明了 COM 作为一种可用的技术所起的作用。

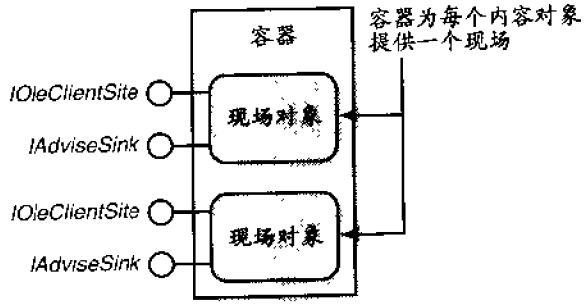

图 18-4　简单的嵌入容器

实际上，链接对象和嵌入对象根本上是不同的。嵌入对象和容器自己的文档数据一起被保存在容器的文档文件中。而另一方面，链接对象保存在外部文件中。容器的文档只是保存了对象的“链接”，简单形象地讲就是容器保存了包含对象数据的文件的名称和路径。实际上，链接要比这复杂得多。例如，如果您创建了一个对 Excel 电子表格中一组单元的链

接,那么链接包含的信息就不仅是文件的路径,还要有标识此单元范围的信息。

18.1.7 Active Documents

在我的观念里,OLE 是所有 Microsoft 定义的 COM 技术中最没趣味的一个,因此在本书中不会对它进一步讨论了。(如果您想了解它的更多知识,可以从 Visual C++ 附带的Scribble 教程中开始学习 OLE。)但是一个从 OLE 发展来的基于 COM 的技术至少值得在这里谈一下,因为它具有潜在的重要用途。这个技术就是"Active Documents"(活动文档)。

Active Documents 协议是对象链接与嵌入的超集。它允许 Active Document 容器(如 Microsoft Internet Explorer)打开由 Active Document 服务器(如 Word 和 Excel)创建的文档文件。曾经注意过在 Internet Explorer 中打开 Word DOC 文件或 Excel XLS 文件时的情况吗? Internet Explorer 好像是理解 Word 和 Excel 文件格式似的。其实并非如此。实际上,Internet Explorer 通过 COM 接口与 Word 或 Excel 进行了对话。Word 或 Excel 运行在背景上(在Internet Explorer 中打开 DOC 或 XLS 文件时只要查看一下任务列表就可以证实),并且基本上占据了 Internet Explorer 的窗口内部。您实际上是在使用 Word 或 Excel,虽然看上去不是这样。

Active Documents 实际上在您将 Word 或 Excel 文档发布到 Web 站点上时起了很大的作用。如果 Internet Explorer 运行的计算机上安装了 Word 和 Excel,就可以很容易像查看 HTML 网页一样查看 DOC 和 XLS 文件了。这就是 Active Documents 的功能。

18.1.8 ActiveX

首先是 OLE,接下来是 COM,然后才出现 ActiveX。在 1995 年 Microsoft 把注意力集中在 Internet 上时,软件天才们就创造了术语 ActiveX,它指代一套基于 COM 的技术,用来使 Internet(特别是 World Wide Web)更具有交互性。ActiveX 控件可能是最有名的 ActiveX 技术,但除此而外还有别的技术。如果将"Active"作为名字的一部分,称为 ActiveX 技术的有 ActiveX控件、ActiveX 数据对象(ADO)、动态服务器页面(ASP)以及 Active Documents 等等。这仅仅是一小部分。每天这个名单都在增长。

所有 ActiveX 技术的一个共同之处是它们都是基于 COM 的。例如 ActiveX 控件就是 COM 对象,它遵守在 Microsoft 的 OLE 控件(OCX)规范中提出的行为规则。管理 ActiveX 控件的应用程序也要实现 COM 接口。正式地讲,它们被称为 ActiveX 控件容器。

编写一个完整的 ActiveX 控件(可以被插入 Web 网页或对话框中)并不是一个小任务。ActiveX 控件体系结构很复杂。典型的 ActiveX 控件要实现 10 多个 COM 接口,其中一些甚至还包含 20 多个方法。即使看上去较简单的工作,如将 ActiveX 控件插入对话框中,都要比大多数人想像的复杂。要拥有一个 ActiveX 控件,对话框就必须成为 ActiveX 控件容器,并且容器必须实现许多它们自己的 COM 接口。

幸运的是,MFC 为封装 ActiveX 控件和控件容器做了许多精彩的工作。在 AppWizard 中

选中一个复选框，任何一个对话框就会立即成为控件容器。您不必编写哪怕是一行代码，因为 MFC 已经为您提供了所有必要的基础结构。MFC 还简化了 ActiveX 控件的开发过程。从头开始编写一个 ActiveX 控件可能至少需要两个月的开发时间。但是，如果使用 MFC，您就可以在几小时之内编写出一个功能完整的控件。为什么？因为 MFC 提供了常备的 COM 的 ActiveX 控件接口的实现。您所要做的就是在某些地方覆盖一些虚函数和添加一些元素，以使自己的控件有所不同。

18.2　MFC 和 COM

MFC 使 COM、OLE 和 ActiveX 程序设计更容易的主要原因是它在类（如 COleControl 和 COleControlSite 类）中提供了对公用 COM 接口的封装。COM 被称为“空心 API”，意思是 Microsoft定义了接口和方法并告诉您这些方法的用途，但是却留待您（对象实现者）去编写程序代码。这样的好处是只要您是编写 MFC 明确支持的 ActiveX 控件、Automation 服务器或是其他类型的组件，MFC 都会为您实现必要的接口。

在接下来的三章中，您将会熟悉许多实现了 COM 接口的 MFC 类。现在，我希望能让您理解 MFC 类是如何实现 COM 接口的。要实现这个目的，您必须理解 COM 程序员用来编写代表 COM 对象的 C++ 类的两种技术。首先是“多重继承”，其次是“嵌套类”。MFC 只用到了嵌套类，但是为了比较两者的特性，还是让我们看一下这两种技术。

18.2.1　多重继承

C++ 程序员使用下列语法规则定义 COM 接口：

```
interface IUnknown
{
    virtual HRESULT__stdcall QueryInterface (REFIID riid, void** ppv) = 0;
    virtual ULONG __stdcall AddRef () = 0;
    virtual ULONG__ stdcall Release () = 0;
};
```

关键字 interface 是 struct 的别名。所以对于 C++ 程序员，接口的定义就是一组纯粹的虚函数按逻辑捆绑在一起作为公用结构的成员。并且由于在 C++ 中，结构和类基本上被同等看待，所以从一个接口派生另一个接口完全合法，如下所示：

```
interface IMath : public IUnknown
{
    virtual HRESULT__stdcall Add (int a, int b, int * pResult) = 0;
    virtual HRESULT__stdcall Subtract (int a, int b, int * pResult) = 0;
};
```

在开发 C++ 类时,由于接口的定义只是一组纯粹虚拟函数的集合,所以您可以利用这个特点。例如,可以声明一个实现 IMath 的类:

```
class CComClass : public IMath
{
protected:
    long m_lRef;    // Reference count
public:
    CComClass ();
    virtual ~CComClass ();
    // IUnknown methods
    virtual HRESULT __stdcall QueryInterface (REFIID riid, void * * ppv);
    virtual ULONG __stdcall AddRef ();
    virtual ULONG __stdcall Release ();
    // IMath methods
    virtual HRESULT __stdcall Add (int a, int b, int * pResult);
    virtual HRESULT __stdcall Subtract (int a, int b, int * pResult);
};
```

使用这种设置,您就可以把 QueryInterface、AddRef、Release、Add 和 Subtract 作为 CComClass 类的成员函数而实现它们了。

现在,假设您希望用 CComClass 不仅仅实现一个 COM 接口,而是想实现两个。那该怎么办呢?一种方法是通过多重继承从 IMath 和另一个接口派生 CComClass 类,如下所示:

```
class CComClass : public IMath, public ISpelling
{
protected:
    long m_lRef;    // Reference count
public:
    CComClass ();
    virtual CComClass ();
    // IUnknown methods
    virtual HRESULT __stdcall QueryInterface (REFIID riid, void * * ppv);
    virtual ULONG __stdcall AddRef ();
    virtual ULONG __stdcall Release ();
    // IMath methods
    virtual HRESULT __stdcall Add (int a, int b, int * pResult);
    virtual HRESULT __stdcall Subtract (int a, int b, int * pResult);
    // ISpelling methods
    virtual HRESULT __stdcall CheckSpelling (wchar_t * pString);
};
```

这种方法有几个优点。首先,它很简单。要声明实现了 n 个接口的类,只要在基类的类列表中包含 n 个接口就可以。其次,需要实现一次 IUnknown。如果每个接口事实上是单独实现的,就必须为每个都实现 QueryInterface、AddRef 和 Release。但是通过多重继承,所有接口的

所有方法可以根本上融合在一个实现中。

使用多重继承编写 COM 类的更有趣的一个方面发生在客户调用 QueryInterface 请求接口指针时。假设客户要请求一个 IMath 指针，返回接口指针的恰当方法是将 this 指针强制转换为 IMath *：

```
*ppv = (IMath *) this;
```

如果客户请求的是一个 ISpelling 指针，那么就将其转换为 ISpelling *：

```
*ppv = (ISpelling *) this;
```

如果没有进行强制类型转换，那么程序虽然会得到正常的编译，但是在使用两个接口中的一个时可能就会出现问题。为什么？因为使用多重继承生成的类包含多个 vtable 和多个 vtable 指针，如果不进行强制类型转换，就不知道 this 指针引用的是哪个 vtable。换句话说，以上给出的两个强制转换即使在 this 永远不变的情况下也会返回不同的数字值。如果客户请求一个 ISpelling 指针，而您却返回一个无类型（未强制转换）的指针，并且如果 this 碰巧引用了 IMath 的 vtable，客户就会通过调用 IMath vtable 来调用 ISpelling 方法。这就是产生灾难的原因，也是多重继承总是要进行强制转换来获取合适的 vtable 指针的原因。

18.2.2　嵌套类

使用多重继承实现 COM 类有什么问题呢？完全没有，如果不存在两个接口具有相同名字和标记的方法的话。如果 IMath 和 ISpelling 都包含名为 Init 的方法，它具有相同的参数列表但要求单独的实现，那么您就不能够使用多重继承来定义实现它们的类了。为什么呢？因为使用了多重继承，类就只能具有一个名为 Init 的成员函数。因此就不可能单独地为 IMath 和 ISpelling 实现 Init 了。

这种限制正是 MFC 使用嵌套类方法实现 COM 接口的原因。嵌套类与多重继承相比稍微有点麻烦和不直观，但是它也具有相当的一般性。在 C++ 中使用嵌套类方法时，无论接口的特性如何，都可以实现任何 COM 接口的组合。下面说明其工作原理。

假设 CComClass 实现了 IMath 和 ISpelling，并且两个接口都具有方法 Init，它们不接受任何参数。

```
virtual HRESULT __stdcall Init () = 0;
```

在这种情况下不能使用多重继承，因为 C++ 不能在一个类中支持语义上相同的函数。因此，您需要定义两个子类，每个都实现一个接口：

```
class CMath : public IMath
{
protected:
    CComClass * m_pParent;    // Back pointer to parent
```

```
public:
    CMath ();
    virtual CMath ();
    // IUnknown methods
    virtual HRESULT__stdcall QueryInterface (REFIID riid, void * * ppv);
    virtual ULONG__stdcall AddRef ();
    virtual ULONG__stdcall Release ();
    // IMath methods
    virtual HRESULT__stdcall Add (int a, int b, int * pResult);
    virtual HRESULT _stdcall Subtract (int a, int b, int * pResult);
    virtual HRESULT_ _stdcall Init () = 0;
};

class CSpelling : public ISpelling
{
protected:
    CComClass * m_pParent;    // Back pointer to parent
public:
    CSpelling ();
    virtual CSpelling ();
    // IUnknown methods
    virtual HRESULT __stdcall QueryInterface (REFIID riid, void * * ppv);
    virtual ULONG__stdcall AddRef ();
    virtual ULONG__stdcall Release ();
    // ISpelling methods
    virtual HRESULT_ _stdcall CheckSpelling (wchar_t * pString);
    virtual HRESULT_ _stdcall Init () = 0;
};
```

为使 CMath 和 CSpelling 成为嵌套类,就要在 CComClass 中声明它们。然后在 CComClass 中包含一对作为 CMath 和 CSpelling 实例的数据成员:

```
class CComClass : public IUnknown
{
protected:
    long m_lRef;              // Reference count
    class CMath : public IMath
    {
        [...]
    };
    CMath m_objMath;          // CMath object
    class CSpelling : public ISpelling
    {
        [...]
    };
    CSpelling m_objSpell;     // CSpelling object
```

```
public:
    CComClass ();
    virtual ~CComClass ();
    // IUnknown methods
    virtual HRESULT__stdcall QueryInterface (REFIID riid, void * * ppv);
    virtual ULONG__stdcall AddRef ();
    virtual ULONG__stdcall Release ();
};
```

注意，现在 CComClass 只从 IUnknown 得到了派生。它没有从 IMath 或 ISpelling 进行派生是因为嵌套类提供了两个接口的实现。如果客户调用 QueryInterface 来请求 IMath 指针，CComClass 会简单地给它传递一个指向 CMath 对象的指针：

```
*ppv = (IMath *) &m_objMath;
```

与此相似，如果请求 ISpelling 指针，CComClass 会返回指向 m_objSpell 的指针：

```
*ppv = (ISpelling *) &m_objSpell;
```

理解嵌套类方法的关键是子对象必须将对它们的 IUnknown 方法的调用委派给父类中同等的方法。注意在保存引用计数成员变量的地方，每个嵌套类都保存了一个 CComClass 指针。该指针是一个“回溯指针”，指向子对象的父亲。通过利用回溯指针调用 CComClass 的 IUnknown 方法实现了委派。通常，由父亲的构造函数初始化回溯指针：

```
CComClass::CComClass ()
{
    [...]        // Normal initialization stuff goes here.
    m_objMath.m_pParent = this;
    m_objSpell.m_pParent = this;
}
```

嵌套类的 IUnknown 实现如下：

```
HRESULT__stdcall CComClass::CMath::QueryInterface (REFIID riid, void * * ppv)
{
    return m_pParent -> QueryInterface (riid, ppv);
}

ULONG__stdcall CComClass::CMath::AddRef ()
{
    return m_pParent -> AddRef ();
}

ULONG__stdcall CComClass::CMath::Release ()
{
    return m_pParent -> Release ();
}
```

这种类型的委派是必要的。首先,如果客户调用由子对象实现的接口中的 AddRef 或 Release,父对象的引用计数就会被调整,而不是子对象的引用计数被调整。其次,如果客户调用一个子对象中的 QueryInterface,父对象就必须接管调用,因为只有父对象知道存在哪个嵌套类以及它实现了哪个接口。

18.2.3 MFC 和嵌套类

如果您浏览 MFC 类(如 COleControl)的源程序,并不会看到与上一小节中相似的程序代码。这是因为 MFC 在宏背后隐藏了嵌套类。

MFC 的 COleDropTarget 类就是一个例子。它是较简单的 MFC COM 类中的一个,只实现了一个接口(标准接口 IDropTarget)。如果看一下 Afxole.h 中的代码,在 COleDropTarget 的类声明结尾处会看到如下代码:

```
BEGIN_INTERFACE_PART(DropTarget, IDropTarget)
    [...]
    STDMETHOD(DragEnter)(LPDATAOBJECT, DWORD, POINTL, LPDWORD);
    STDMETHOD(DragOver)(DWORD, POINTL, LPDWORD);
    STDMETHOD(DragLeave)();
    STDMETHOD(Drop)(LPDATAOBJECT, DWORD, POINTL pt, LPDWORD);
END_INTERFACE_PART(DropTarget)
```

MFC 的 BEGIN_INTERFACE_PART 宏定义了实现一个 COM 接口的嵌套类。类的名称是通过在宏的参数列表中的第一个参数前加上一个前缀 X 来定义的。在本例中,嵌套类的名称为 XDropTarget。END_INTERFACE_PART 宏声明了一个成员变量,它是一个嵌套类的实例。下面是由预处理器生成的程序代码:

```
class XDropTarget : public IDropTarget
{
public:
    STDMETHOD_(ULONG, AddRef)();
    STDMETHOD_(ULONG, Release)();
    STDMETHOD(QueryInterface)(REFIID iid, LPVOID* ppvObj);
    STDMETHOD(DragEnter)(LPDATAOBJECT, DWORD, POINTL, LPDWORD);
    STDMETHOD(DragOver)(DWORD, POINTL, LPDWORD);
    STDMETHOD(DragLeave)();
    STDMETHOD(Drop)(LPDATAOBJECT, DWORD, POINTL pt, LPDWORD);
} m_xDropTarget;
friend class XDropTarget;
```

看出预处理器的输出结果与前面讨论的嵌套类的例子的相似之处了吗?注意嵌套类实例的名称为 m_x 加上宏的参数列表中的第一个参数,本例中是 m_xDropTarget。

嵌套类实现了 3 个 IUnknown 方法,外加 BEGIN_INTERFACE_PART 和 END_INTERFACE_

PART 之间列出的方法。IDropTarget 具有四个方法：DragEnter、DragOver、DragLeave 以及 Drop，这些方法在前面的程序列表中进行了命名。下面给出 MFC 源程序文件 Oledrop2 .cpp中的一段摘录，说明在嵌套 XDropTarget 类中 IDropTarget 的方法是如何被实现的：

```
STDMETHODIMP_(ULONG) COleDropTarget::XDropTarget::AddRef()
{
    [...]
}

STDMETHODIMP_(ULONG) COleDropTarget::XDropTarget::Release()
{
    [...]
}

STDMETHODIMP COleDropTarget::XDropTarget::QueryInterface(...)
{
    [...]
}

STDMETHODIMP COleDropTarget::XDropTarget::DragEnter(...)
{
    [...]
}

STDMETHODIMP COleDropTarget::XDropTarget::DragOver(...)
{
    [...]
}

STDMETHODIMP COleDropTarget::XDropTarget::DragLeave(...)
{
    [...]
}

STDMETHODIMP COleDropTarget::XDropTarget::Drop(...)
{
    [...]
}
```

代码中方法的实现方式现在并不重要。关键是在一个 MFC 源程序代码列表中一些看上去无用的宏却转变成了实现完整 COM 接口的嵌套类。为每个接口包含一个 BEGIN_INTERFACE_PART/END_INTERFACE_PART 结构块就可以创建实现几个 COM 接口的类。另外，如果两个以上的接口包含相同的方法，您也不必为可能发生的冲突担心，因为嵌套类技术允许每个接口(及其方法)可以独立地实现。

18.2.4　MFC 实现 IUnknown 的方法

让我们返回去仔细研究一下 COleDropTarget 中 QueryInterface、AddRef 和 Release 的实现。

下面是完整没有删节的版本:

```
STDMETHODIMP_(ULONG) COleDropTarget::XDropTarget::AddRef()
{
    METHOD_PROLOGUE_EX_(COleDropTarget, DropTarget)
    return pThis->ExternalAddRef();
}

STDMETHODIMP_(ULONG) COleDropTarget::XDropTarget::Release()
{
    METHOD_PROLOGUE_EX_(COleDropTarget, DropTarget)
    return pThis->ExternalRelease();
}

STDMETHODIMP COleDropTarget::XDropTarget::QueryInterface(
    REFIID iid, LPVOID* ppvObj)
{
    METHOD_PROLOGUE_EX_(COleDropTarget, DropTarget)
    return pThis->ExternalQueryInterface(&iid, ppvObj);
}
```

注意,MFC 的工作再次隐藏在了宏背后。宏 METHOD_PROLOGUE_EX_创建了一个名为 pThis 的堆栈变量,指向 XDropTarget 的父亲,也就是 COleDropTarget 对象(XDropTarget 对象是它的成员)。了解了这一点,您就可以明白 XDropTarget 的 IUnknown 方法被委派给了 COleDropTarget。这样就产生了两个问题:COleDropTarget 的 ExternalAddRef、ExternalRelease 和 ExternalQueryInterface函数是做什么用的?它们来自何处?

第 2 个问题要易于回答一些。所有 3 个函数都是 CCmdTarget 的成员,而 COleDropTarget 是从 CCmdTarget 派生来的。要回答第 1 个问题,我们应该先看一下 CCmdTarget 中函数的具体实现。下面给出了从 MFC 源程序文件 Oleunk.cpp 中摘录的程序:

```
DWORD CCmdTarget::ExternalAddRef()
{
    [...]
    return InternalAddRef();
}

DWORD CCmdTarget::ExternalRelease()
{
    [...]
    return InternalRelease();
}

DWORD CCmdTarget::ExternalQueryInterface(const void* iid,
    LPVOID* ppvObj)
{
    [...]
```

```
    return InternalQueryInterface(iid, ppvObj);
}
```

ExternalAddRef、ExternalRelease 和 ExternalQueryInterface 调用了另一组 CCmdTarget 函数：InternalAddRef、InternalRelease 和 InternalQueryInterface。“内部”函数比较复杂，但是如果研究一下它们，您就会发现它们执行的正是 AddRef、Release 和 QueryInterface 要做的工作，不同处是以 MFC 的方式。现在我们知道了嵌套类的 IUnknown 方法委派给了父类，父类则从 CCmdTarget 中继承了这些类的实现。让我们继续。

18.2.5　接口映射

最有趣的“内部”函数是 InternalQueryInterface。如果看一下 Oleunk.cpp，您会发现它调用了一个不为人知的函数 GetInterface，它属于一个不太为人所知的类 CUnknown。GetInterface 在表格中进行查找来确定类是否支持指定的接口。它然后再检索指向实现接口的嵌套类的指针并将其返回给 InternalQueryInterface。因此 MFC 是使用表格驱动机制来实现 QueryInterface 的。但表格是从哪里来的呢？

我们再看一下 COleDropTarget 的例子。就在 COleDropTarget 类声明的结尾处是语句：

```
DECLARE_INTERFACE_MAP()
```

并且在 COleDropTarget 中有以下一组相关语句：

```
BEGIN_INTERFACE_MAP(COleDropTarget, CCmdTarget)
    INTERFACE_PART(COleDropTarget, IID_IDropTarget, DropTarget)
END_INTERFACE_MAP()
```

DECLARE_INTERFACE_MAP 是一个 MFC 宏，它声明了接口映射，此映射是一个表格，以每个由类（实际上是嵌套类）实现的接口为表目。BEGIN_INTERFACE_MAP 和 END_INTERFACE_MAP 也是宏。它们定义了接口映射的内容。正如同消息映射告诉 MFC 类为哪个消息提供了处理程序一样，接口映射告诉 MFC 类支持哪些 COM 接口，以及哪个嵌套类提供了接口的实现。出现在 BEGIN_INTERFACE_MAP 和 END_INTERFACE_MAP 之间的每个 INTERFACE_PART 宏都组成表格中的一个输入项。在本例中，INTERFACE_PART 语句告诉 MFC：接口映射是 COleDropTarget 的成员，COleDropTarget 实现了 IDropTarget 接口，而包含实际的 IDropTarget 实现的嵌套类是 XDropTarget。INTERFACE_PART 使用与 BEGIN_INTERFACE_PART 同样的方式给类名前添加一个 X。

因为接口映射可以包含任意数量的 INTERFACE_PART 宏，所以 MFC 类并没有被限制于每个类只能有一个 COM 接口，实际上它们可以有好几个接口。对于出现在类的接口映射表中的每个 INTERFACE_PART 输入项，在类声明中都有一个 BEGIN_INTERFACE_PART/END_INTERFACE_PART 声明块。看一下 Ctlcore.cpp 中的 COleControl 的接口映射表以及

AfxCtl.h 中的多个 BEGIN_INTERFACE_PART/END_INTERFACE_PART 声明块,您就明白我所指的意思了。

18.2.6 MFC 和聚合

是不是有些奇怪,CCmdTarget 有两组在其名字中都有 QueryInterface、AddRef 和 Release 的函数?在我给出 External 函数的源程序时,为了明白起见,忽略了解释原因的部分。下面再次给出其没有缩略的形式:

```
DWORD CCmdTarget::ExternalAddRef()
{
    // delegate to controlling unknown if aggregated
    if (m_pOuterUnknown != NULL)
        return m_pOuterUnknown->AddRef();

    return InternalAddRef();
}

DWORD CCmdTarget::ExternalRelease()
{
    // delegate to controlling unknown if aggregated
    if (m_pOuterUnknown != NULL)
        return m_pOuterUnknown->Release();

    return InternalRelease();
}

DWORD CCmdTarget::ExternalQueryInterface(const void* iid,
    LPVOID* ppvObj)
{
    // delegate to controlling unknown if aggregated
    if (m_pOuterUnknown != NULL)
        return m_pOuterUnknown->QueryInterface(*(IID*)iid, ppvObj);

    return InternalQueryInterface(iid, ppvObj);
}
```

注意,只有在 m_pOuterUnknown 等于 NULL 值时,"外部"函数才调用"内部"函数。m_pOuterUnknown 是 CCmdTarget 成员变量,保存着对象的 controlling unknown。如果 m_pOuterUnknown 不为 NULL,"外部"函数就通过保存在 m_pOuterUnknown 中的指针进行委派。如果您熟悉 COM 聚合,那么您可能已经猜到了其中的具体工作。但是如果聚合对您而言还是新内容,那前面的代码就需要进一步解释了。

COM 从来不会按 C++ 的方式支持继承。换句话说,我们不可以按照从 C++ 中的一个类派生另一个类那样的方式从一个 COM 对象派生另一个。但是,COM 的对象重用支持两种机制——"包容"和"聚合"。

包容较简单。为说明其工作原理，我们假定要编写一个包含一对方法 Add 和 Subtract 的对象。现在假设别人也编写了一个包含 Multiply 和 Divide 方法的 COM 对象，并且您希望将这两个方法合并到自己的对象中。一种"借用"其他对象的方法的途径是让自己的对象用 CoCreateInstance 创建其他对象，然后再按需求调用方法。您的对象是外部对象，另一个对象是内部对象，并且如果 m_pInnerObject 保存了指向实现 Multiply 和 Divide 的内部对象上的接口的指针，那么您可能还要在自己的对象中包含 Multiply 和 Divide 方法，它们的实现如下所示：

```
HRESULT__stdcall CComClass::Multiply(int a, int b, int * pResult)
{
    return m_pInnerObject -> Multiply(a, b, pResult);
}

HRESULT__stdcall CComClass::Divide(int a, int b, int * pResult)
{
    return m_pInnerObject -> Divide(a, b, pResult);
}
```

这就是简单的包容。图 18-5 说明了内部对象和外部对象之间的关系。注意，内部对象的接口只提供给了外部对象，而不是外部对象的客户。

聚合则完全不同。当两个对象进行聚合时，聚合对象提供了内部对象和外部对象的接口（参见图 18-6）。客户并不知道该对象实际上是两个以上对象的聚合。

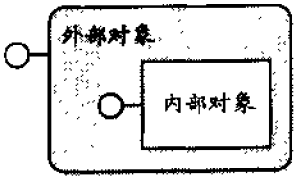

图 18-5　包容

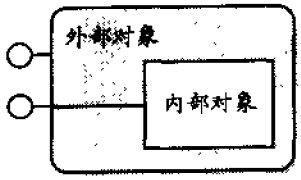

图 18-6　聚合

聚合与包容的相似之处在于都是由外部对象创建内部对象，但是相似也仅此而已。聚合在工作时，内部对象和外部对象必须合作，以建立一种它们实际上是一个对象的错觉。两个对象必须遵守一组严格的行为规则。这些规则之一就是外部对象必须传递自己的 IUnknown 指针给内部对象。该指针成为内部对象的 controlling unknown。如果客户调用内部对象的 IUnknown 方法，内部对象就必须通过 controlling unknown 调用 QueryInterface、AddRef 或 Release 来实现对外部对象的委派。这正是 CCmdTarget 的"外部"函数通过 m_pOuterUnknown 调用 QueryInterface、AddRef 或 Release 时执行的操作。如果对象是被聚合的，m_pOuterUnknown 则为非 NULL 值并且"外部"函数会委派给外部对象。否则，对象不被聚合，而且相应调用"内部"函数。

包容和聚合的主要区别在于任何对象都可以被另一个对象包容，但是只有那些被指定

为支持聚合的对象才能进行聚合处理。MFC 使得聚合易于实现,是因为它内置了可自由使用的聚合支持。

18.2.7 MFC 和类厂

任何对 COM 进行了友好封装的类库都应该支持类厂。COM 类厂通常包含许多样本程序,它们在应用程序彼此之间变化不大,因此是隐藏在 C++ 类中的最佳选择。

MFC 在 COleObjectFactory 中提供了 COM 类的封装实现。MFC 的 COleObjectFactory 类实现了两个 COM 接口:IClassFactory 和 IClassFactory2。IClassFactory2 是 IClassFactory 的超集,它支持所有的 IClassFactory 方法,并增加了一个授权方法,该函数主要由 ActiveX 控件使用。

当创建 COleObjectFactory 时,要给它的构造函数提供 4 个重要的信息。第 1 个是类厂创建的对象的 CLSID。第 2 个是 RUNTIME_CLASS 指针,标识实现该类型对象的 C++ 类。第 3 个是 BOOL 值,告诉 COM,这个服务器(如果是 EXE 的话)能否创建多个对象实例。如果此参数为 TRUE,那么 10 个客户调用该服务器实现的 COM 类中的 CoCreateInstance,就会启动 10 个不同的 EXE 实例。如果参数为 FALSE,那么只有 1 个 EXE 实例服务 10 个客户。COleObjectFactory 构造函数的第 4 个也是最后一个参数是类厂创建的对象的 ProgID。ProgID 是 Program ID 的缩写形式,是供人阅读的名称(例如 Math.Object),可以用在用到 CLSID 的地方来标识 COM 类。下面的程序片段创建了一个 COleObjectFactory,它在 CLSID_Math 被传递给 COM 激活函数时初始化 CComClass:

```
COleObjectFactory cf (
    CLSID_Math,                 // The object's CLSID
    RUNTIME_CLASS (CComClass),  // Class representing the object
    FALSE,                      // Many clients, one EXE
    _T ("Math.Object")          // The object's ProgID
);
```

大多数 MFC 应用程序并不显式地声明一个 COleObjectFactory 的实例;它们使用 MFC 的 DECLARE_OLECREATE 和 IMPLEMENT_OLECREATE 宏。当预处理器遇到以下代码时

```
// In the class declaration
DECLARE_OLECREATE (CComClass)

// In the class implementation
IMPLEMENT_OLECREATE (CComClass, "Math.Object", 0x708813ac,
    0x88d6, 0x11d1, 0x8e, 0x53, 0x00, 0x60, 0x08, 0xa8, 0x27, 0x31)
```

它输出

```
// In the class declaration
public:
    static COleObjectFactory factory;
```

```
    static const GUID guid;

// In the class implementation
COleObjectFactory CComClass::factory(CComClass::guid,
    RUNTIME_CLASS(CComClass), FALSE,_T("Math.Object"));
const GUID CComClass::guid =
    { 0x708813ac, 0x88d6, 0x11d1, { 0x8e, 0x53, 0x00,
      0x60, 0x08, 0xa8, 0x27, 0x31 } };
```

OLECREATE 宏的一个缺陷是它们包含了对 COleObjectFactory 的硬代码引用。如果您从 COleObjectFactory 中派生了一个类并希望在应用程序中使用它，那您必须或者是放弃那些宏，手工编写派生类的引用代码，或者是编写自己的 OLECREATE 宏。程序员有时的确会发现从 COleObjectFactory 派生自己的类来修改类厂的行为是很有用的。例如，通过覆盖虚 OnCreateObject 函数，您可以创建一个“singleton”类厂，该类厂在第一次调用 IClassFactory::CreateInstance 时创建一个对象，并在响应以后的激活请求时为已存在的对象提供指针。

在内部，MFC 维持一个应用程序创建的所有 COleObjectFactory 对象的链接列表。（看一下 COleObjectFactory 构造函数的内部，您会看到用来将每个新实例化的对象添加到列表中的程序。）COleObjectFactory 包含一些便于使用的函数，用来通过操作系统注册一个应用程序的所有类厂，以及注册类厂在系统注册表中创建的对象。语句

```
COleObjectFactory::UpdateRegistryAll();
```

给注册表添加创建对象所要求的所有信息，应用程序的类厂给该对象提供了服务。这项功能很强大，因为别的方法需要编写依赖于 Win32 注册表函数来更新注册表自身的低级程序。

18.2.8　总结

到目前为止，本章覆盖的内容使您足以了解 MFC 和 COM 之间的关系了吗？几乎不可能。在以后的 3 章中，您还会发现许多内容。但是本章为以后的学习打下了基础。现在当您再看到类似图 18-1 那样的图表时，就会知道自己在看什么，同时对实现 MFC 的方法有了一个清晰的概念。另外，当您看到向导生成的源程序列表时，或是深入到 MFC 源程序中时，就会了解像 INTERFACE_PART 和 IMPLEMENT_OLECREATE 这样的语句的含义。

如果 COM 对您而言是新内容，那么到现在您可能就会感到一些不知所措了。但是不要气馁。学习 COM 就和学习 Windows 编程一样：在开始真正理解它之前，您必须强制性地忍受半年精神上的困惑。一个好消息是您不必成为 COM 的专家就可以用 MFC 去创建基于 COM 的应用程序。实际上，您根本不必知道太多 COM 的知识。但是如果您（和我一样）相信最好的程序员就应该理解内部的实际操作，那么本章提供的内容就为您踏上漫长征途做了充分的准备。

第 19 章　剪贴板和 OLE 拖放

从 1.0 版本开始，Microsoft Windows 就支持数据通过剪贴板传送了。剪贴板处于传送路径的中心位置，一个应用程序中的数据可以保存在那里，然后再传送到另一个应用程序或同一个应用程序的其他部分。即使是初学者也能很快掌握剪贴板简单的剪切、复制或粘贴操作：选择数据，将它剪切或复制到剪贴板，然后再粘贴到其他地方。许多应用程序在【编辑】菜单中都有【剪切】、【复制】和【粘贴】命令，并且很多的 Windows 用户已经知道了 Ctrl-X、Ctrl-C和 Ctrl-V 与这些基本剪贴板命令等价。

原始剪贴板(指的是“传统剪贴板”或简单地称为“剪贴板”)在 Windows 98 和 Windows 2000 中仍然存在，但是要鼓励程序员放弃它而使用更新的、功能更强大的剪贴板，称为“OLE 剪贴板”。OLE 剪贴板对传统剪贴板向下兼容，这意味着一个应用程序可以将数据复制到 OLE 剪贴板中，而另一个应用程序可以从传统剪贴板中得到它，或是相反。更重要的是，OLE 剪贴板允许程序员完成一些用传统剪贴板无法完成的工作。例如，要使用传统剪贴板传送一个大型位图，就必须分配足够的内存来保存整个位图。但是使用 OLE 剪贴板传送相同的位图，则可以将位图保存在更合理的存储媒体中，如硬盘上的文件中。

与 OLE 剪贴板紧密相关的是一个传送数据的虚拟方法，称为“OLE 拖放”。OLE 拖放由于取消了中间程序而简化了剪切/复制/粘贴操作。它不需要一个命令来将数据移动到剪贴板，然后再用另一个命令将其粘贴到文档中，OLE 拖放允许您用鼠标抓住数据的一部分然后拖放到希望的目的地。在内部有一种与 OLE 剪贴板类似的机制帮助实现了数据传送。而对于用户而言，传送好像是无缝的，非常直观。

对于涉及到传统剪贴板的操作，MFC 并没有提供明确的支持，但是它却支持 OLE 剪贴板和 OLE 拖放。如果从头编写使用这两种数据传送机制的程序，那不是一件容易的事情，但是有了 MFC 的帮助，问题就简单多了。

要理解 OLE 剪贴板和 OLE 拖放的最好办法是首先理解传统剪贴板的工作方式。为了实现最终目的，在本章一开始我们将回顾一下传统剪贴板和用来访问它们的 Windows API 函数。然后将介绍 OLE 剪贴板和 OLE 拖放，并说明如何使用 MFC 开发直接支持它们的应用程序。

19.1　传统剪贴板

数据传送到传统剪贴板或从其中传出是通过使用一个小型 Windows API 函数子集实现

的。表 19-1 简要地总结了这些函数。

表 19-1　剪贴板 API 函数

函数	说　明
OpenClipboard	打开剪贴板
CloseClipboard	关闭剪贴板
EmptyClipboard	删除当前剪贴板中内容
GetClipboardData	从剪贴板中取得数据
SetClipboardData	将数据传送到剪贴板

将数据放置到剪贴板中需要 4 个步骤：

1. 用::OpenClipboard 打开剪贴板。
2. 用::EmptyClipboard 放弃目前保存在剪贴板中的任何数据。
3. 使用::SetClipboardData 将包含剪贴板数据的全局内存块或其他对象(例如位图句柄)的所有权传送给剪贴板。
4. 用::CloseClipboard 关闭剪贴板。

"全局内存块"是用::GlobalAlloc API 函数分配的一块内存。::GlobalAlloc 返回类型为 HGLOBAL 的句柄，在 Win32 应用程序中被作为一般的 HANDLE。一个相关函数::GlobalLock 得到 HGLOBAL 并返回指向内存块的指针。Windows 程序员已经不怎么使用::GlobalAlloc 了，因为在 Win32 API 中::HeapAlloc 取代了它。但是，因为剪贴板要求一个内存句柄而不是指针，所以对于剪贴板程序设计，::GlobalAlloc 还是很有用的。

下列程序将文本字符串放置到剪贴板中，方法是将文本字符串复制到全局内存块中并将内存块移交给了剪贴板：

```
char szText[] = "Hello, world"; // ANSI characters
if (::OpenClipboard (m_hWnd))
    ::EmptyClipboard ();

    HANDLE hData = ::GlobalAlloc (GMEM_MOVEABLE, ::lstrlen (szText) + 1);
    LPSTR pData = (LPSTR) ::GlobalLock (hData);
    ::lstrcpy (pData, szText);
    ::GlobalUnlock (hData);

    ::SetClipboardData (CF_TEXT, hData);
    ::CloseClipboard ();
}
```

一旦内存块被移交给了剪贴板，分配该块的应用程序就不应该使用或删除它了。现在剪贴板就拥有了自己的内存，并将在适当的时候释放它(具体地说，在应用程序下次调用::EmptyClipboard的时候)。

传递给::OpenClipboard 的唯一参数是窗口的句柄，该窗口是剪贴板打开时拥有剪贴板

的窗口。当然在 MFC 应用程序中,可以从 m_hWnd 数据成员中得到 CWnd 的窗口句柄。如果另一个应用程序已经使剪贴板打开了,那么::OpenClipboard 就会失败。必须使每个应用程序都在使用剪贴板之前打开它,Windows 用这种方法来同步化对共享资源的访问,确保应用程序使用时剪贴板中的内容不会被修改。

从剪贴板获取数据非常简单。执行步骤如下:

1. 用::OpenClipboard 打开剪贴板。
2. 用::GetClipboardData 来获取包含剪贴板数据的全局内存块或其他对象的句柄。
3. 对数据进行本地拷贝,从全局内存块中复制数据。
4. 用::CloseClipboard 关闭剪贴板。

下列程序可以获取上例中保存在剪贴板中的文本字符串:

```
char szText[BUFLEN];
if (::OpenClipboard (m_hWnd)) {
    HANDLE hData = ::GetClipboardData (CF_TEXT);
    if (hData != NULL) {
        LPCSTR pData = (LPCSTR) ::GlobalLock (hData);
        if (::lstrlen (pData) < BUFLEN)
            ::lstrcpy (szText, pData);
        ::GlobalUnlock (hData);
    }
    ::CloseClipboard ();
}
```

如果在剪贴板中文本字符串有效,那么此例程结束时 szText 中就会保存有它的副本。

19.1.1 剪贴板格式

::SetClipboardData 和::GetClipboardData 都接受一个指定剪贴板类型的整数值,它标识了传送中涉及到的数据类型。前一节中的例子使用了 CF_TEXT,它表示数据为 ANSI 文本。Windows 使用单独的剪贴板格式 ID 来标识 Unicode 文本。(这就是在两个例子中都使用 char 数据类型而不是 TCHAR 的原因。)CF_TEXT 是 Windows 支持的几个预定义格式之一。在表 19-2 中给出了部分剪贴板格式的列表。

表 19-2 常用剪贴板格式

格式	数据类型
CF_BITMAP	Windows 位图
CF_DIB	设备无关位图
CF_ENHMETAFILE	GDI 增强型元文件
CF_METAFILEPICT	旧式(非增强型)GDI 元文件,带有附加的尺寸和映射方式信息
CF_HDROP	HDROP 格式中文件名称列表

续表

格式	数据类型
CF_PALETTE	GDI 调色板
CF_TEXT	8 位 ANSI 字符组成的文本
CF_TIFF	TIFF 格式的位图
CF_UNICODETEXT	16 位 Unicode 字符组成的文本
CF_WAVE	WAV 格式的音频数据

您可以使用预定义的剪贴板格式来传送位图、调色板、增强型元文件以及其他对象，像传送文本一样容易。例如，如果 m_bitmap 为保存有位图的 CBitmap 数据成员，下面就给出了一种方法来将该位图复制并放置到剪贴板上：

```
if (::OpenClipboard (m_hWnd)) {
    // Make a copy of the bitmap.
    BITMAP bm;
    CBitmap bitmap;
    m_bitmap.GetObject (sizeof (bm), &bm);
    bitmap.CreateBitmapIndirect (&bm);

    CDC dcMemSrc, dcMemDest;
    dcMemSrc.CreateCompatibleDC (NULL);
    CBitmap * pOldBitmapSrc = dcMemSrc.SelectObject (&m_bitmap);
    dcMemDest.CreateCompatibleDC (NULL);
    CBitmap * pOldBitmapDest = dcMemDest.SelectObject (&bitmap);

    dcMemDest.BitBlt (0, 0, bm.bmWidth, bm.bmHeight, &dcMemSrc,
        0, 0, SRCCOPY);
    HBITMAP hBitmap = (HBITMAP) bitmap.Detach ();

    dcMemDest.SelectObject (pOldBitmapDest);
    dcMemSrc.SelectObject (pOldBitmapSrc);

    // Place the copy on the clipboard.
    ::EmptyClipboard ();
    ::SetClipboardData (CF_BITMAP, hBitmap);
    ::CloseClipboard ();
}
```

要从剪贴板取回位图，可以调用::GetClipboardData 并给它传递一个 CF_BITMAP 参数：

```
if (::OpenClipboard (m_hWnd)) {
    HBITMAP hBitmap = (HBITMAP) ::GetClipboardData (CF_BITMAP);
    if (hBitmap != NULL)
        // Make a local copy of the bitmap.
    {
```

```
    ::CloseClipboard();
}
```

注意这里采用的模式。将数据放置到剪贴板的应用程序告诉 Windows 数据类型。获取数据的应用程序则申请特定的数据类型。如果该数据格式无效,::GetClipboardData 返回 NULL。在上例中,如果剪贴板没有包含 CF_BITMAP 类型的数据,那么::GetClipboardData 将返回 NULL,而且不会执行复制位图的程序段。

在调用::GetClipboardData 时,系统默默地将一些剪贴板格式转换为相关的数据类型。例如,如果应用程序 A 将一个 ANSI 文本字符串复制到了剪贴板中(CF_TEXT),而应用程序 B 则调用::GetClipboardData 来请求 Unicode 文本(CF_UNICODETEXT),Windows 2000 就会将文本转换为 Unicode,而且::GetClipboardData 将返回有效的内存句柄。位图也可以进行这种数据转换。在 Windows 98 和 Windows 2000 中将 CF_BITMAP 位图转换为了 CF_DIB,反过来转换也可以。这样就增添了一种受人欢迎的可移植方法,剪贴板格式可以代表相同基本数据类型的不同形式。

CF_HDROP 剪贴板格式

更有意思的(至少被证明是这样)剪贴板格式是 CF_HDROP。当您从剪贴板中检索 CF_HDROP 格式的数据时,得到一个 HDROP,它实际上是一个全局内存块的句柄。在内存块中是一个文件名称的列表。不要通过分析内存块中的内容来阅读文件名称,您可以使用::DragQueryFile 函数。下列程序从剪贴板中得到 HDROP 并将所有文件名称都装入一个由 CListBox 指针 pListBox 引用的列表中:

```
if (::OpenClipboard (m_hWnd)) {
    HDROP hDrop = (HDROP) ::GetClipboardData (CF_HDROP);
    if (hDrop != NULL) {
        // Find out how many file names the HDROP contains.
        int nCount = ::DragQueryFile (hDrop, (UINT) -1, NULL, 0);
        // Enumerate the file names.
        if (nCount) {
            TCHAR szFile[MAX_PATH];
            for (int i = 0; i < nCount; i++) {
                ::DragQueryFile (hDrop, i, szFile,
                    sizeof (szFile) / sizeof (TCHAR));
                pListBox -> AddString (szFile);
            }
        }
    }
    ::CloseClipboard();
}
```

从 HDROP 中提取文件名称很容易,但要插入它们却有些麻烦。HDROP 引用的内存块

包含一个 DROPFILES 结构，后面跟着文件名称列表，最后是两个连接的 NULL 字符。在 Shlobj.h 中 DROPFILES 定义如下：

```
typedef struct _DROPFILES {
    DWORD pFiles;            // Offset of file list
    POINT pt;                // Drop coordinates
    BOOL fNC;                // Client or nonclient area
    BOOL fWide;              // ANSI or Unicode text
} DROPFILES, FAR * LPDROPFILES;
```

要创建自己的 HDROP，您可以分配一个全局内存块，在其中初始化 DROPFILES 结构，并附带文件名称列表。需要初始化的 DROPFILES 字段是 pFiles，它保存了文件名称列表中第一个字符相对于内存块开始处的偏移量；以及 fWide，它说明文件名称是由 ANSI (fWide = FALSE) 字符还是由 Unicode (fWide = TRUE) 字符组成的。为便于理解，下面的程序创建了一个包含两个文件名称的 HDROP，并将 HDROP 放置到了剪贴板中：

```
TCHAR szFiles[3][32] = {
     _T("C:\\My Documents\\Book\\Chap20.doc"),
     _T("C:\\My Documents\\Book\\Chap21.doc"),
     _T("")
};

if (::OpenClipboard(m_hWnd)) {
    ::EmptyClipboard();
    int nSize = sizeof(DROPFILES) + sizeof(szFiles);
    HANDLE hData = ::GlobalAlloc(GHND, nSize);
    LPDROPFILES pDropFiles = (LPDROPFILES)::GlobalLock(hData);
    pDropFiles->pFiles = sizeof(DROPFILES);

#ifdef UNICODE
     pDropFiles->fWide = TRUE;
#else
     pDropFiles->fWide = FALSE;
#endif

     LPBYTE pData = (LPBYTE)pDropFiles + sizeof(DROPFILES);
     ::CopyMemory(pData, szFiles, sizeof(szFiles));
     ::GlobalUnlock(hData);
     ::SetClipboardData(CF_HDROP, hData);
     ::CloseClipboard();
}
```

在本例中传递给::GlobalAlloc 的 GHND 参数组合了 GMEM_MOVEABLE 和 GMEM_ZEROINIT 标志。GMEM_ZEROINIT 告诉::GlobalAlloc 将内存块中的所有字节都初始化为 0，这样就确

保了 DROPFILES 结构的未初始化的成员被设置为 0。另外,如果您在 Win32 环境下给剪贴板递交了一个分配的全局内存块,GMEM_MOVEABLE 标志就不再是必需的了,尽管一些文档中说是。在这里使用它是出于对 16 位 Windows 的考虑,它要求用 GMEM_MOVEABLE 和 GMEM_DDESHARE 标志分配剪贴板内存。

HDROP 的这种传送文件名称的方式好像有一些奇怪。但是 Windows 98 和 Windows 2000 命令解释器却使用此格式剪切、复制和粘贴文件。您可以简单地做一个实验,看看命令解释器是如何使用 HDROP 的。将示例程序复制到应用程序,并修改文件名称使它引用硬盘上真实的文件。执行程序将 HDROP 传送给剪贴板。然后在硬盘文件夹上打开窗口并选择该窗口【编辑】菜单中的【粘贴】命令。命令解释器就会将名称出现在 HDROP 中的文件移到文件夹中。

19.1.2 私有剪贴板格式

CF_TEXT、CF_BITMAP 和其他预定义的剪贴板格式包含了一个很广的数据类型范围,但是它们不可能包括每种应用程序要通过剪贴板传送的数据类型。因此,Windows 允许您注册私有的剪贴板格式,并用它们代替标准格式,或与标准格式一起使用它们。

假设您正在编写一个 Widget 应用程序用来创建饰件。您可能会希望您的用户能够将饰件剪切或复制到剪贴板并粘贴到文档的其他地方(或是完全不同的文档中)。为支持这样的功能,可以调用 Win32 API 函数::RegisterClipboardFormat 为饰件注册私有剪贴板格式:

```
UINT nID = ::RegisterClipboardFormat (_T("Widget"));
```

您得到的 UINT 是私有剪贴板格式的 ID。要将饰件复制到剪贴板,就要将定义饰件所需的所有数据都复制到全局内存块中,然后用私有剪贴板格式 ID 和内存句柄调用::SetClipboardData:

```
::SetClipboardData (nID, hData);
```

要从剪贴板中取得饰件,应将饰件的剪贴板格式传递给::GetClipboardData:

```
HANDLE hData = ::GetClipboardData (nID);
```

然后锁定内存块获得指针并根据内存块中的数据重新构造饰件。这里的关键是如果 10 个不同的应用程序(或同一个应用程序的 10 个不同实例)用相同的格式名调用::RegisterClipboardFormat,所有 10 个应用程序都会得到相同的剪贴板格式 ID。这样,如果应用程序 A 将饰件复制到剪贴板,应用程序 B 去获取它,则只要在调用::RegisterClipboardFormat 时两个应用程序指定了相同的格式名称,操作就会顺利进行。

19.1.3 以多种格式提供数据

只要每个项目都代表不同的格式,那么将多个项目放置在剪贴板中就是合法的。应用

程序经常这样做。这是将数据提供给多个应用程序(甚至是那些不理解您的私有剪贴板格式的应用程序)的有效方法。

Microsoft Excel 是一个很好的例子,它使用了多个剪贴板格式。如果您在 Excel 中选择了一个范围内的电子表格单元格并将所选内容复制到剪贴板中,那么 Excel 会在剪贴板中放置至多 30 个项目。其中之一使用了私有剪贴板格式,代表 Excel 自己的电子表格数据。另一个是表格单元格的 CF_BITMAP 译文。Windows 带有的 Paint 程序并不理解 Excel 的私有剪贴板格式,但它却可以将 Excel 电子表格单元格粘贴到位图中。至少看上去好像 Paint 可以粘贴电子表格单元格。实际上,它粘贴的是这些单元格的位映射图像而不是电子表格单元格。您甚至还可以将 Excel 数据粘贴到 Notepad 中,因为 Excel 放置在剪贴板中的格式之一是 CF_TEXT。通过使电子表格数据在多种格式下有效,Excel 提高了其剪贴板数据的可移植性。

通过什么方法将两个以上的项目放入剪贴板呢?很简单,只要为每种格式调用一次::SetClipboardData就可以了:

```
::SetClipboardData (nID, hPrivateData);
::SetClipboardData (CF_BITMAP, hBitmap);
::SetClipboardData (CF_TEXT, hTextData);
```

现在,如果应用程序调用::GetClipboardData 来申请 CF_TEXT 格式、CF_BITMAP 格式或由 nID 指定的私有格式下的数据,调用就不会失败,并且调用者将收到一个返回的非 NULL 数据句柄。

19.1.4　查询有效数据格式

确定在特定格式下剪贴板数据是否有效的一种方法是调用::GetClipboardData 并检查返回值是否是 NULL。但是,有时您可能想预先知道调用::GetClipboardData 是否会成功,或是想知道通过枚举得到的当前所有有效格式,以便可以选择最适合自己的一种。表19-3中的 Win32 API 函数可以帮助您实现这一切。

表 19-3　Win32 API 函数

函数	说　明
CountClipboardFormats	返回当前有效的格式总数
EnumClipboardFormats	枚举所有有效的剪贴板格式
IsClipboardFormatAvailable	说明特定格式下的数据是否有效
GetPriorityClipboardFormat	给定一个具有优先级别的格式列表,说明哪个是首先可以使用的

::IsClipboardFormatAvailable 是 4 个函数中最简单的一种。为了确定是否可以使用 CF_TEXT 格式下的数据,可以调用::IsClipboardFormatAvailable,如下所示:

```
if (::IsClipboardFormatAvailable (CF_TEXT)) {
    // Yes, it's available.
}
else {
    // No, it's not available.
}
```

此函数通常用来实现【编辑】菜单中【粘贴】命令的更新处理程序。可以参考第 7 章中说明其用法的例子。

即使剪贴板没有打开,::IsClipboardFormatAvailable 也会正常工作。但别忘了如果剪贴板没有打开,剪贴板中的数据就很容易变动。不要在编程时犯如下错误:

```
if (::IsClipboardFormatAvailable (CF_TEXT)) {
    if (::OpenClipboard (m_hWnd)) {
        HANDLE hData = ::GetClipboardData (CF_TEXT);
        LPCSTR pData = (LPCSTR) ::GlobalLock (hData);
          .
          .
          .
        ::CloseClipboard ();
    }
}
```

这个程序的错误在于在多任务环境下,会发生出现概率很小但确实存在的情况,即在调用::GetClipboardData之前如果执行了::IsClipboardFormatAvailable,剪贴板上的数据就可能被别的数据取代。您只要在调用::IsClipboardFormatAvailable 之前打开剪贴板就可以避免这种风险:

```
if (::OpenClipboard (m_hWnd)) {
    if (::IsClipboardFormatAvailable (CF_TEXT)) {
        HANDLE hData = ::GetClipboardData (CF_TEXT);
        LPCSTR pData = (LPCSTR) ::GlobalLock (hData);
          .
          .
          .
    }
    ::CloseClipboard ();
}
```

这个程序就不会出什么问题,因为只有打开剪贴板的应用程序才能修改剪贴板中的内容。

可以使用::EnumClipboardFormats 以迭代方式对所有有效剪贴板格式的列表进行操作,如下所示:

```
if (::OpenClipboard (m_hWnd)) {
```

```
    UINT nFormat = 0; // Must be 0 to start the iteration.
    while (nFormat = ::EnumClipboardFormats (nFormat)) {
        // Next clipboard format is in nFormat.
    }
    ::CloseClipboard ();
}
```

在到达列表末尾时::EnumClipboardFormats 会返回 0,因此在检索了最后一个有效格式之后就会退出循环。如果您只是想知道有多少数据格式在剪贴板上有效,则可以调用::CountClipboardFormats。

最后一个检查剪贴板数据有效性的函数是::GetPriorityClipboardFormat,它简化了检查多个(不只是一个)剪贴板格式的过程。假设您的应用程序能够粘贴以下格式的数据:保存在 nID 中的私有格式、CF_TEXT 格式或 CF_BITMAP 格式。您首先选用私有格式,如果该格式无效,则选择 CF_TEXT,而如果这两个都失败,则最后选择 CF_BITMAP。但编程时不能写为:

```
if (::OpenClipboard (m_hWnd)) {
    if (::IsClipboardFormatAvailable (nID)) {
        // Perfect!
    }
    else if (::IsClipboardFormatAvailable (CF_TEXT)) {
        // Not the best, but I'll take it.
    }
    else if (::IsClipboardFormatAvailable (CF_BITMAP)) {
        // Better than nothing.
    }
    ::CloseClipboard ();
}
```

您应该这样编写:

```
UINT nFormats[3] = {
    nID, // First choice
    CF_TEXT, // Second choice
    CF_BITMAP // Third choice
};

if (::OpenClipboard (m_hWnd)) {
    UINT nFormat = ::GetPriorityClipboardFormat (nFormats, 3);
    if (nFormat > 0) {
        // nFormat holds nID, CF_TEXT, or CF_BITMAP.
    }
    ::CloseClipboard ();
}
```

::GetPriorityClipboardFormat 的返回值是当前有效格式列表中第一个格式的 ID。如果没有有效的格式存在,::GetPriorityClipboardFormat 将返回 -1,如果剪贴板为空则返回 0。

19.1.5 延时再现

传统剪贴板的一个缺陷是它上面的所有数据都要保存到内存中。对于文本和其他简单的数据类型,基于内存的数据传送既快速又效率高。但是假设某人要复制一个 10MB 位图到剪贴板。在剪贴板被清空之前,位图会占用 10 MB 的 RAM。并且如果没有人粘贴该位图,分配给它的内存空间就会被闲置。

为了避免这种浪费,Windows 支持"延时再现"。延时再现使得应用程序可以说:"我可以通过剪贴板提供数据,但是必须等到有人请求时才将它复制到剪贴板。"延时再现是如何工作的呢?首先用有效的剪贴板格式 ID 而不是 NULL 数据句柄来调用::SetClipboardData。然后通过用::SetClipboardData 将数据实际放到剪贴板上来响应 WM_RENDERFORMAT 消息。当应用程序调用::GetClipboardData 来请求该特定格式的数据时,就发送 WM_RENDERFORMAT 消息。如果没有人申请数据,那么消息永远不会被发送,您也不必去分配一个 10MB 内存。要记住 WM_RENDERFORMAT 消息处理程序不应调用::OpenClipboard 和::CloseClipboard,因为在消息接收时,接收消息的窗口已经隐含地拥有了剪贴板。

处理 WM_RENDERFORMAT 消息的应用程序必须也处理 WM_RENDERALLFORMATS 消息。如果应用程序在剪贴板具有 NULL 数据句柄(应用程序放置在那里的)时结束,就会发送 WM_RENDERALLFORMATS 消息。相应消息处理程序的工作是打开剪贴板,将应用程序通过延时再现机制承诺提供的数据传送到剪贴板,然后关闭剪贴板。将数据放在剪贴板中可以确保使用延时再现的应用程序结束以后还能给其他应用程序提供该数据。

第 3 个剪贴板消息 WM_DESTROYCLIPBOARD 在延时再现中也起到了作用。此消息通知应用程序它没有责任再提供延时再现的数据了。当另一个应用程序调用::EmptyClipboard 时发送此消息。它还会在 WM_RENDERALLFORMATS 消息之后被发送。如果您为了响应 WM_RENDERFORMAT 和 WM_RENDERALLFORMATS 消息而占据着一些资源,那么当 WM_DESTROYCLIPBOARD 消息到达时就可以安全地释放这些资源了。

下面给出一个 MFC 应用程序使用延时再现将位图放置在剪贴板中的方法:

```
// In CMyWindow's message map
ON_COMMAND (ID_EDIT_COPY, OnEditCopy)
ON_WM_RENDERFORMAT ()
ON WM_ RENDERALLFORMATS ()
   .
   .
   .
// Elsewhere in CMyWindow
void CMyWindow::OnEditCopy ()
```

```
}
    ::SetClipboardData (CF_BITMAP, NULL);
}

void CMyWindow::OnRenderFormat (UINT nFormat)
{
    if (nFormat == CF_BITMAP) {
        // Make a copy of the bitmap, and store the handle in hBitmap.
            .
            .
            .
        ::SetClipboardData (CF_BITMAP, hBitmap);
    }


void CMyWindow::OnRenderAllFormats ()
{
    ::OpenClipboard (m_hWnd);
    OnRenderFormat (CF_BITMAP);
    ::CloseClipboard ();
```

这个例子并不完全实用，因为在复制到剪贴板和取回之间的时间段内，位图还存在被修改的可能性（如果应用程序是位图编辑器并且位图已经打开进行编辑时，可能性会更大），此时 OnEditCopy 将被迫复制当前状态下的位图。如果 OnEditCopy 复制了位图，那么使用延时再现的目的就失败了。延时再现就是用来保存内存的工具，但是如果应用程序被迫对每个项目都进行复制（这些项目是为延时再现而“复制”到剪贴板的），那为什么不直接把项目复制到剪贴板呢？

其实并不是必须如此。快照可以被保存在硬盘上。下面给出了程序的修改版本，说明即使数据易于变动延时再现也可以保存内存：

```
// In CMyWindow's message map
ON_COMMAND (ID_EDIT_COPY, OnEditCopy)
ON_WM_RENDERFORMAT ()
ON_WM_RENDERALLFORMATS ()
ON_WM_DESTROYCLIPBOARD ()
   .
   .
   .
// Elsewhere in CMyWindow
void CMyWindow::OnEditCopy ()
{
    // Save the bitmap to a temporary disk file.
```

```
            .
            .
            .
        ::SetClipboardData (CF_BITMAP, NULL);
    }

    void CMyWindow::OnRenderFormat (UINT nFormat)
    {
        if (nFormat == CF_BITMAP) {
            // Re-create the bitmap from the data in the temporary file.
                .
                .
                .
            ::SetClipboardData (CF_BITMAP, hBitmap);
        }
    }

    void CMyWindow::OnRenderAllFormats ()
    {
        ::OpenClipboard (m_hWnd);
        OnRenderFormat (CF_BITMAP);
        ::CloseClipboard ();
    }

    void CMyWindow::OnDestroyClipboard ()
    {
        // Delete the temporary file.
    }
```

这里的思路是将位图的副本保存在 OnEditCopy 中的文件中，并从 OnRenderFormat 中的文件中重新创建。磁盘空间要比 RAM 便宜很多，因此在大多数情况下这种折衷是可接受的。

19.1.6 创建可重用剪贴板类

考虑到剪贴板的特性，您可能会奇怪地发现 MFC 并没有提供 CClipboard 类来封装这些剪贴板 API。您不费多大力就可以创建自己的剪贴板类，但实际上没有必要去这么做。为什么呢？因为 OLE 剪贴板可以完成传统剪贴板所做的所有工作，并且 MFC 已经完成了对 OLE 剪贴板的封装了。涉及到 OLE 剪贴板的操作要比涉及传统剪贴板的操作复杂得多，但是 MFC 给我们降低了难度。实际上，在 MFC 的帮助下使用 OLE 剪贴板并不比使用传统剪贴板困难多少。在以后的几节中将解释原因。

19.2 OLE 剪贴板

OLE 剪贴板是传统剪贴板的现代版本。它也是向下兼容的。多亏有了 OLE 库中那些

奇妙的技术，您可以将文本字符串、位图或其他一些项目放在 OLE 剪贴板中，而另外一个对 OLE 一无所知的应用程序却可以像从传统剪贴板中那样从其中粘贴项目。反过来，一个应用程序也可以使用 OLE 剪贴板从传统剪贴板中获取数据。

OLE 剪贴板有什么不同之处呢？为什么它比旧的剪贴板高级呢？它们之间主要有两个不同点。首先，OLE 剪贴板完全是基于 COM 的；所有的数据都是通过指向 COM 接口的指针调用方法来传送的。其次，OLE 剪贴板支持存储媒体而不是全局内存。相反，传统剪贴板使用内存来进行所有数据的传送，实际上可用的内存限制了通过剪贴板传送项目的大小。由于传统剪贴板不能使用媒体而只能用内存进行数据传送，所以传统剪贴板和 OLE 剪贴板之间的兼容性就受到了限制，只有通过内存传送的项目才能被复制到一个剪贴板或从另一个剪贴板中获得。

仅有第 1 个原因还不足以放弃传统剪贴板。COM 虽然很流行，使用对象也很好，但是如果没有 MFC，那么编写与 OLE 剪贴板交互的程序要比编写与传统剪贴板交互的程序复杂得多。第 2 个原因（可以自由地使用可选的存储媒体）正是要使用 OLE 剪贴板的理由。使用传统剪贴板不可能传送 4GB 的位图，因为 Windows 当前版本并不支持这样大的内存对象。但是使用 OLE 剪贴板，您可以传送任何适合于硬盘的数据。事实上，利用一些精巧的设计，您可以传送任何项目，即使这些项目太大而不适合放在硬盘上。由于当今的许多应用程序不得不处理大量的信息，所以实际上 OLE 剪贴板已成为了非常方便的工具。

19.2.1　OLE 剪贴板基础

要理解 OLE 剪贴板，最重要的基本概念是当您将一个数据项目放在剪贴板上时，实际上您放置的并不是数据本身，而是封装了数据的 COM 数据对象。“数据对象”是一种 COM 对象，它实现了一个 IDataObject 接口。IDataObject 具有两个在 OLE 剪贴板操作中起主要作用的方法：SetData 和 GetData。假设数据对象是一般的数据仓库（与自定义的用来处理特殊数据的对象相对），数据提供者用 IDataObject::SetData 将数据填入数据对象，然后用 ::OleSetClipboard将对象放在 OLE 剪贴板上。数据使用者会调用::OleGetClipboard 来获得剪贴板数据对象的 IDataObject 指针，然后调用 IDataObject::GetData 来获取数据。

图 19-1 给出了 OLE 剪贴板操作的概念性示意图。这是个简化了的视图，其中由 ::OleGetClipboard返回的 IDataObject 指针实际上并不是传递给::OleSetClipboard 的 IDataObject 指针，而是指向由系统提供的剪贴板数据对象实现的 IDataObject 接口的指针，该数据对象封装了提供给::OleSetClipboard 的数据对象，并允许消费者可以访问传统剪贴板上的数据。幸运的是这些间接性一点儿也不影响您的程序编写，您只要用 IDataObject 接口与数据对象交互即可，系统会完成剩下的工作。

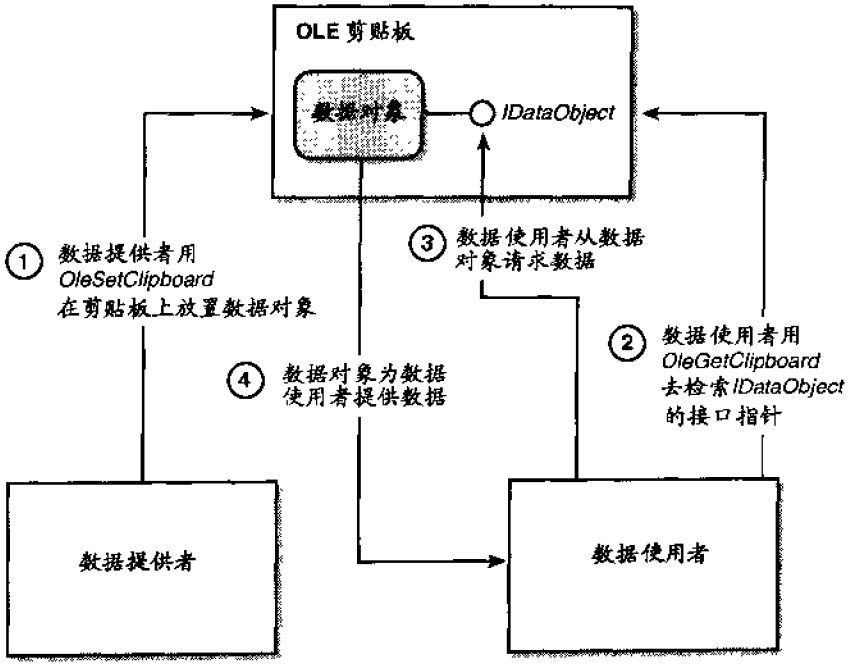

图 19-1 通过 OLE 剪贴板传送数据

使用 OLE 剪贴板听起来好像很简单,但是只要是涉及到 COM 和 OLE,就不会有什么简单的事情存在了。最困难的地方是为数据对象编写程序,而且不仅要实现 IDataObject::GetData 和 IDataObject::SetData,还要实现其他 IDataObject 方法。但是首要的问题我们要放在第一位。假设您已经实现了一个数据对象,pdo 保存着指向对象 IDataObject 接口的指针,下面给出一种将文本字符串放置在 OLE 剪贴板上的方法:

```
// Copy the text string to a global memory block.
char szText[] = "Hello, world";
HANDLE hData = ::GlobalAlloc (GMEM_MOVEABLE, ::lstrlen (szText) + 1);
LPSTR pData = (LPSTR) ::GlobalLock (hData);
::lstrcpy (pData, szText);
::GlobalUnlock (hData);

// Initialize a FORMATETC structure and a STGMEDIUM structure that
// describe the data and the location at which it's stored.
FORMATETC fe;
fe.cfFormat = CF_TEXT;              // Clipboard format = CF_TEXT
fe.ptd = NULL;                      // Target device = Screen
fe.dwAspect = DVASPECT_CONTENT;     // Level of detail = Full content
fe.lindex = -1;                     // Index = Not applicable
fe.tymed = TYMED_HGLOBAL;           // Storage medium = Memory
```

```
STGMEDIUM stgm;
stgm.tymed = TYMED_HGLOBAL;          // Storage medium = Memory
stgm.hGlobal = hData;                // Handle to memory block
stgm.pUnkForRelease = NULL;          // Use ReleaseStgMedium

// Place the data object on the OLE clipboard.
pdo->SetData(&fe, &stgm, FALSE);
::OleSetClipboard(pdo);
pdo->Release();
```

最后一条语句中的 Release 调用假定创建数据对象的应用程序在把它提交给 OLE 剪贴板后就不再使用它了。调用数据对象上的 Release 不会促使对象进行自我删除，因为::OleSetClipboard 执行了传递给它的 IDataObject 指针上的 AddRef。

获取文本字符串就稍微简单一些，因为我们不必去创建数据对象。但是整个过程还是不像从传统剪贴板中获取字符串那样直接：

```
char szText[BUFLEN];
IDataObject * pdo;
STGMEDIUM stgm;

FORMATETC fe = {
    CF_TEXT, NULL, DVASPECT_CONTENT, -1, TYMED_HGLOBAL
};

if (SUCCEEDED(::OleGetClipboard(&pdo))) {
    if (SUCCEEDED(pdo->GetData(&fe, &stgm) && stgm.hGlobal != NULL)) {
        LPCSTR pData = (LPCSTR)::GlobalLock(stgm.hGlobal);
        if (::lstrlen(pData) < BUFLEN)
            ::lstrcpy(szText, pData);
        ::GlobalUnlock(stgm.hGlobal);
        ::ReleaseStgMedium(&stgm);
    }
    pdo->Release();
}
```

如果数据对象不能提供所要求的数据，它将返回一个 HRESULT 表明操作失败。在本例中使用的 SUCCEEDED 宏与第 18 章中用来测试 HRESULT 时用到的宏相同。

在 SetData 和 GetData 操作中有两个起重要作用的结构：FORMATETC 和 STGMEDIUM。FORMATETC 描述了数据格式并标识了保存数据的“存储媒体”(例如全局内存块或文件)的类型。下面给出了 Objidl.h 中 FORMATETC 的定义：

```
typedef struct tagFORMATETC {
    CLIPFORMAT cfFormat;          // Clipboard format
    DVTARGETDEVICE * ptd;         // Target device
    DWORD dwAspect;               // Level of detail
```

```
    LONG lIndex;                // Page number or other index
    DWORD tymed;                // Type of storage medium
} FORMATETC;
```

两个最重要的字段是 cfFormat 和 tymed。cfFormat 保存着剪贴板格式 ID。该 ID 可以是标准剪贴板格式 ID 如 CF_TEXT 或 CF_BITMAP,也可以是私有剪贴板格式 ID。tymed 标识了存储媒体的类型,可以是表 19-4 中列出的任一个值。大多数 OLE 剪贴板数据传送仍旧使用老式全局内存块,但是您可以明白地看出 FORMATETC 也支持其他媒体类型。

表 19-4 IDataObject 存储媒体类型

tymed 标志	存储媒体类型
TYMED_HGLOBAL	全局内存块
TYMED_FILE	文件
TYMED_ISTREAM	流对象(实现接口 IStream)
TYMED_ISTORAGE	存储对象(实现接口 IStorage)
TYMED_GDI	GDI 位图
TYMED_MFPICT	元文件图像
TYMED_ENHMF	GDI 增强型元文件

FORMATETC 指定了存储类型,而 STGMEDIUM 结构标识了存储媒体自身。例如,如果数据保存在全局内存块中,STGMEDIUM 结构会具有一个 HGLOBAL。相反,如果数据在文件中,STGMEDIUM 就会含有一个指向指定文件名的字符串的指针。STGMEDIUM 还包含有其他信息。下面给出该结构的定义:

```
typedef struct tagSTGMEDIUM {
    DWORD tymed;
    union {
        HBITMAP hBitmap;                // TYMED_GDI
        HMETAFILEPICT hMetaFilePict;    // TYMED_MFPICT
        HENHMETAFILE hEnhMetaFile;      // TYMED_ENHMF
        HGLOBAL hGlobal;                // TYMED_HGLOBAL
        LPOLESTR lpszFileName;          // TYMED_FILE
        IStream *pstm;                  // TYMED_STREAM
        IStorage *pstg;                 // TYMED_STORAGE
    };
    IUnknown *pUnkForRelease;
} STGMEDIUM;
```

这里,tymed 保存了一个 TYMED 值,用来标识存储媒体的类型,就像 FORMATETC 的 tymed 字段一样。hBitmap、hMetaFilePict 和其他内置联合的成员标识了实际的存储媒体。最后,pUnkForRelease 保存了一个指向 COM 接口的指针,其 Release 方法可以释放存储媒体。当应用程序使用 IDataObject::GetData 从 OLE 剪贴板中获取到一个项目时,该应用程序就有责任在不

需要时将存储媒体释放。对于内存块，“释放”意味着释放掉内存块；对于文件而言，就意味着删除文件。COM 提供了一个 API 函数::ReleaseStgMedium，应用程序可以调用它来释放存储媒体。如果在初始化 STGMEDIUM 时简单地把 pUnkForRelease 设置为 NULL，那么::ReleaseStgMedium 就会使用与存储媒体类型相应的逻辑手段来释放存储媒体。

有关这些数据结构还有许多可讲的内容，但是这里提供的介绍已经足够您用来理解上一节中给出的例子了。第 1 个例子初始化了一个 FORMATETC 结构，以描述一个保存在全局内存块（tymed = TYMED_HGLOBAL）中的 ANSI 文本字符串（cfFormat = CF_TEXT）。它还用 STGMEDIUM（hGlobal = hData 和 tymed = TYMED_HGLOBAL）封装了内存块。这两个结构都是用指向 IDataObject::SetData 的地址传递的。

在第 2 个例子中，用相同的参数对 FORMATETC 结构进行了初始化，而没对 STGMEDIUM 结构进行初始化。它们都被传递给 IDataObject::GetData 去获取文本字符串。在本例中，FORMATETC 结构中的参数告诉数据对象，调用者想要哪种数据和哪类存储媒体。IDataObject::GetData返回后，STGMEDIUM 结构就保存了 HGLOBAL，通过它可以访问数据。

现在，您可能开始理解编写 OLE 剪贴板程序要比编写传统剪贴板程序需要投入更多工作的原因了。但是，您其实还没看到所有工作的一半，因为我还没有为数据对象提供程序代码呢。要记住，数据对象是一个实现了 IDataObject 接口的 COM 对象。IDataObject 是基于 COM 的数据传送协议，该协议被 Microsoft 命名为 Uniform Data Transfer（统一数据传输），或 UDT。在前面我已经提到过，GetData 和 SetData 是您必须对付的两个 IDataObject 方法。表 19-5 给出了完整的方法列表。

表 19-5　IDataObject 方法

方法	说　明
GetData	从数据对象中获取数据（对象提供了存储媒体）
GetDataHere	从数据对象中获取数据（调用者提供了存储媒体）
QueryGetData	确定特定格式的数据是否有效
GetCanonicalFormatEtc	创建一个不同的但逻辑上等价的 FORMATETC
SetData	给数据对象提供数据
EnumFormatEtc	用于枚举有效的数据格式
DAdvise	给数据对象建立一个咨询连接
DUnadvise	结束咨询连接
EnumDAdvise	枚举存在的咨询连接

为执行一个简单的剪贴板数据传送，您不必实现所有这些方法（一些方法只是返回特殊的 COM 出错代码 E_NOTIMPL），但是实现 IDataObject 仍然是个不小的工作。给传统剪贴板复制一个简单的文本字符串仅仅需要几行代码。而给 OLE 剪贴板复制同样的字符串却要求几百行程序代码，主要是为实现完整的 COM 数据对象而增加的额外开销。

如果您感觉为了将一个字符串复制到剪贴板而不得不编写几百行代码显得很傻的话，

请不要失望。MFC 为您极大地简化了工作,它为您提供了数据对象,并将其封装在友好的 C++ 类中,隐藏了 FORMATETC 结构和 STGMEDIUM 结构以及 IDataObject 接口的其他低级部分。一般而论,在 MFC 应用程序中使用 OLE 剪贴板并不比使用传统剪贴板困难多少,特别是当您使用全局内存块作为存储媒体时。您还保留有使用文件和其他存储媒体替代全局内存的权利。综合考虑,MFC 的 OLE 剪贴板抽象使得程序员受益匪浅。下面让我们看看您是否同意此观点。

19.2.2 MFC、全局内存和 OLE 剪贴板

MFC 对 OLE 剪贴板的支持主要集中体现在两个类上。首先,COleDataSource 作为剪贴板操作提供者端的模型。其次是 COleDataObject 作为消费者端的模型。换句话说,您要使用 COleDataSource 把数据放置在 OLE 剪贴板上,而用 COleDataObject 把它取回。不必奇怪,COleDataSource 也包含了 COM 的 IDataObject 接口的一般实现。您可以在 MFC 源程序文件 Oledobj2.cpp 中找到它的实现程序。如果您对 MFC 类实现 COM 接口的方式还不熟悉,可以在阅读源程序之前回顾一下第 18 章的内容。

如果让 COleDataSource 执行这些工作的话,将保存在全局内存中的项目放置在 OLE 剪贴板上是很容易的。以下是具体步骤:

1. 在堆上(不是在堆栈上)创建 COleDataSource 对象。
2. 调用 COleDataSource::CacheGlobalData 将 HGLOBAL 递交给 COleDataSource 对象。
3. 调用 COleDataSource::SetClipboard 将对象放置在 OLE 剪贴板上。

下例中使用 COleDataSource 在 OLE 剪贴板上提供了 ANSI 文本字符串:

```
char szText[] = "Hello, world"; // ANSI characters
HANDLE hData = ::GlobalAlloc (GMEM_MOVEABLE, ::lstrlen (szText) + 1);
LPSTR pData = (LPSTR) ::GlobalLock (hData);
::lstrcpy (pData, szText);
::GlobalUnlock (hData);

COleDataSource * pods = new COleDataSource;
pods -> CacheGlobalData (CF_TEXT, hData);
pods -> SetClipboard ();
```

注意,COleDataSource 对象是在堆上而不是在堆栈上创建的。这一点很重要,因为对象必须保留在内存中,直到调用 IUnknown::Release 将对象引用计数降低到 0,到那时对象会执行自我删除。如果您在堆栈上创建了 COleDataSource,对象就会在超出有效范围的时候被删除。

MFC 的 COleDataObject 提供了从 OLE 剪贴板获取项目的机制。下面给出获取保存在全局内存中的项目的过程:

1. 创建 COleDataObject 对象。
2. 调用 COleDataObject::AttachClipboard 将 COleDataObject 连接到 OLE 剪贴板。

3．　使用 COleDataObject::GetGlobalData 获取项目。

4．　释放由 GetGlobalData 返回的全局内存块。

下例说明了使用 COleDataObject 获取上例中放置在 OLE 剪贴板中的文本字符串的方法：

```
char szText[BUFLEN];
COleDataObject odo;
odo.AttachClipboard();
HANDLE hData = odo.GetGlobalData(CF_TEXT);

if (hData != NULL) {
    LPCSTR pData = (LPCSTR) ::GlobalLock(hData);
    if (::lstrlen(pData) < BUFLEN)
        ::lstrcpy(szText, pData);
    ::GlobalUnlock(hData);
    ::GlobalFree(hData);
}
```

AttachClipboard 函数创建了 COleDataObject 和 OLE 剪贴板之间的逻辑连接。一旦连接建立，MFC 就会将对 GetGlobalData 的调用和对其他 COleDataObject 数据获取函数的调用变换为通过由::OleGetClipboard 返回的 IDataObject 指针所指向的对 GetData 的调用。别忘了，释放由 GetGlobalData 返回的全局内存块是您自己的责任。这种要求正是上例中调用::GlobalFree 的原因。

19.2.3　使用其他存储媒体

在本章中到目前为止所有提供的例子都使用了全局内存作为传送媒体。但是别忘了 OLE 剪贴板还支持其他媒体类型。COleDataSource::CacheGlobalData 和 COleDataObject::GetGlobalData 与全局内存块密不可分。但是您可以使用更一般的 COleDataSource::CacheData 和 COleDataObject::GetData 函数在其他类型的媒体中传送数据。

下面的例子说明了如何使用文件作为传送媒体，通过剪贴板来传送文本字符串。字符串首先复制到临时文件中。然后用描述文件的信息和文件包含的数据初始化 FORMATETC 和 STGMEDIUM 结构。最后信息被传送给 COleDataSource::CacheData，并用 COleDataSource::SetClipboard 将数据对象放置在剪贴板上：

```
char szText[] = "Hello, world";
TCHAR szPath[MAX_PATH], szFileName[MAX_PATH];
::GetTempPath(sizeof(szPath) / sizeof(TCHAR), szPath);
::GetTempFileName(szPath, _T("tmp"), 0, szFileName);

CFile file;
if (file.Open(szFileName, CFile::modeCreate | CFile::modeWrite)) {
```

```
    file.Write (szText, ::lstrlen (szText) + 1);
    file.Close ();

    LPWSTR pwszFileName =
        (LPWSTR) ::CoTaskMemAlloc (MAX_PATH * sizeof (WCHAR));

# ifdef UNICODE
    ::lstrcpy (pwszFileName, szFileName);
# else
    ::MultiByteToWideChar (CP_ACP, MB_PRECOMPOSED, szFileName, -1,
        pwszFileName, MAX_PATH);
# endif

    FORMATETC fe = {
        CF_TEXT, NULL, DVASPECT_CONTENT, -1, TYMED_FILE
    };

    STGMEDIUM stgm;
    stgm.tymed = TYMED_FILE;
    stgm.lpszFileName = pwszFileName;
    stgm.pUnkForRelease = NULL;

    COleDataSource * pods = new COleDataSource;
    pods -> CacheData (CF_TEXT, &stgm, &fe);
    pods -> SetClipboard ();
}
```

在调用 CacheData 之前,其地址被复制给 STGMEDIUM 结构的文件的名称必须由 Unicode 字符组成。这一点总是正确的,即使在 Windows 98 中。您还必须使用 COM 函数::CoTaskMem Alloc分配文件名缓冲区。这就确保了在::ReleaseStgMedium 通过缓冲区指针调用::CoTaskMemFree时可以恰当地释放缓冲区。

在消费者端,您可以使用 COleDataObject::GetData 从剪贴板获取字符串:

```
char szText[BUFLEN];
STGMEDIUM stgm;

FORMATETC fe = {
    CF_TEXT, NULL, DVASPECT_CONTENT, -1, TYMED_FILE
};

COleDataObject odo;
odo.AttachClipboard ();

if (odo.GetData (CF_TEXT, &stgm, &fe) && stgm.tymed == TYMED_FILE) {
    TCHAR szFileName[MAX_PATH];

# ifdef UNICODE
    ::lstrcpy (szFileName, stgm.lpszFileName);
# else
```

```
        ::WideCharToMultiByte (CP_ACP, 0, stgm.lpszFileName,
            -1, szFileName, sizeof (szFileName) / sizeof (TCHAR), NULL, NULL);
#endif

        CFile file;
        if (file.Open (szFileName, CFile::modeRead)) {
            DWORD dwSize = file.GetLength ();
            if (dwSize < BUFLEN)
                file.Read (szText, (UINT) dwSize);
            file.Close ();
        }
        ::ReleaseStgMedium (&stgm);
    }
```

在使用 COleDataObject::GetData 取回数据之后，就应将存储媒体释放，这就是在本例最后调用::ReleaseStgMedium 的原因。

当然，通过文件而不是全局内存块传送文本字符串并没有多大的意义。但是如果传送的项目是大型位图时，这种传送就很有意义了，特别是如果位图已经保存在了磁盘上时。本节的例子中我使用文本字符串主要是为了使程序代码尽可能地简单明了，但是例子中说明的原则却适用于所有数据类型。

19.2.4　将 OLE 剪贴板看作 CFile

MFC 的 COleDataObject::GetFileData 函数提供了一个便于使用的 OLE 剪贴板抽象，用户可以把剪贴板当作普通的 CFile，对保存在下列任一个存储媒体中的数据进行获取操作：

- TYMED_HGLOBAL
- TYMED_FILE
- TYMED_MFPICT
- TYMED_ISTREAM

如果成功，则 GetFileData 将返回指向 CFile 对象的指针，该对象封装了从剪贴板获取到的项目。您可以通过该指针调用 CFile::Read 来读取数据。

下例说明了使用 GetFileData 从 OLE 剪贴板中获取字符串的方法：

```
char szText[BUFLEN];
COleDataObject odo;
odo.AttachClipboard ();

CFile* pFile = odo.GetFileData (CF_TEXT);

if (pFile != NULL) {
    DWORD dwSize = pFile->GetLength ();
    if (dwSize < BUFLEN)
        pFile->Read (szText, (UINT) dwSize);
```

```
    delete pFile; // Don't forget this!
}
```

注意,必须亲自去删除 CFile 对象,GetFileData 返回了该对象的地址。如果忘了去释放它,就会出现内存泄漏。

上面给出的代码在功能上与上一小节的 GetData 例子等价,但是它却有两个优点。一个是它更简单。第二个优点是无论数据保存在全局内存、文件或是流中,它都可以正常工作,以不变应万变。为了得到与 GetData 相同的结果,您必须做以下工作:

```
char szText[BUFLEN];
STGMEDIUM stgm;

COleDataObject odo;
odo.AttachClipboard();

FORMATETC fe = {
    CF_TEXT, NULL, DVASPECT_CONTENT, -1,
    TYMED_FILE | TYMED_HGLOBAL | TYMED_ISTREAM
};

if (odo.GetData (CF_TEXT, &stgm, &fe)) {
    switch (stgm.tymed) {

    case TYMED_FILE:
        // Read the string from a file.
            .
            .
            .
            break;

        case TYMED_HGLOBAL:
            // Read the string from a global memory block.
            .
            .
            .
            break;

        case TYMED_ISTREAM:
            // Read the string from a stream object.
            .
            .
            .
            break;
        }
        ::ReleaseStgMedium (&stgm);
}
```