第14章　多　任　务

由于Windows是多任务操作系统，应用程序可以以多种方式运行。在应用程序内部通过创建同时运行的处理线程可以实现多任务。此时还可以从自己的应用程序来执行其他应用程序，但是它们实际上将与用户自己的应用程序同时运行。 MFC提供了一种使用方便的成员函数，将其重载便可以处理后台程序。本章的实例具体包括：

实例53　后台处理，本例中将演示在没有命令需要处理时，一种连续清除后台应用程序临时分配内存的方式。

实例54　运行其他应用程序。本例中将从自己的应用程序来运行其他应用程序。

实例55　改变优先级，当没有其他活跃的应用程序时，改变应用程序或者线程的优先级，使它运行得更快或者更慢。

实例56　应用程序(工作者线程)内部的多任务处理，本例中将创建一个工作者线程 (Worker Thread)来完成数学运算功能，与此同时由应用程序查询用户的需要。工作者线程与后面例子中的用户界面线程不同，后者意味着面向任务存在，任务完成后则终止。

实例57　应用程序 (用户界面线程)内部的多任务处理，本例将创建用户界面线程 (User Interface Thread)，该线程具有运行另外一个应用程序的功能，同时又具有与应用程序在同一地址空间的好处，这将允许线程和应用程序之间轻而易举地共享数据。

实例58　向线程发送消息，本例将实现与用户界面线程通信。

实例59　与线程共享数据，本例将演示一种在多个线程之间无冲突地共享同一个数据地址空间的方式。

14.1　实例53：后台处理

1. 目标

在应用程序处于空闲时，清除应用程序所占的零碎内存，在本例中，将在后台清除临时分配的内存。

2. 策略

使用ClassWizard重载应用程序类的OnIdle()函数，只要没有用户命令需要处理，应用程序便调用它。

3. 步骤

后台处理

用Class Wizard重载应用程序类的OnIdle()成员函数。

在本例中，使用重载的OnIdle()函数删除临时对象的列表:

```
BOOL CWzdApp::OnIdle( LONG lCount )
{
   // clean up temporary objects
   while ( !m_TempList.IsEmpty() )
   {
```

```
        delete m_TempList.RemoveHead();
    }
    return CWinApp::OnIdle( lCount );
}
```

4. 注意

■ 确保重载的OnIdle()连续调用CWinApp::OnIdle(),否则应用程序将中断工作。

■ 一个在其“外壳(shell)”中运行了多个其他应用程序的应用程序可以使用该实例以确保它产生的应用程序仍在运行，否则，它可以再次运行它们。

■ OnIdle()不是进行冗长处理过程的理想场所，例如数学处理。如果 OnIdle()不能迅速返回，则由于鼠标和键盘的消息被延迟响应而使应用程序将看起来显得反应迟钝，因此最好在其他应用程序或者线程中完成密集运算，详见下面的实例。

5. 使用光盘时注意

运行随书附带光盘上的工程时，在Wzd.cpp的OnIdle()函数中设置断点(在条件语句中设置，以免不停地产生中断)。单击Wzd菜单中的Test命令，填满一个全局列表集合。注意 OnIdle()函数将清除这个集合。

14.2 实例54：运行其他应用程序

1. 目标

从一个应用程序运行其他应用程序。

2. 策略

由于MFC类库中目前没有可以运行其他应用程序的 MFC类，因此本例使用两个WindowsAPI调用。第一个是 ::WinExec ()，它是一个简单、直接的 API，微软不推荐使用。::WinExec()调用了第二个API，::CreateProcess ()，它提供了对新生成程序的更广泛控制。

3. 步骤

1) 使用::WinExec ()运行应用程序

使用::WinExec ()运行应用程序可以参阅下面的代码。在本例中，运行了批处理文件Wzd.bat。

```
CString str;
str = "Wzd.bat";

// execute a batch file
if ( ::WinExec(
    str,                    // command line
    SW_NORMAL )             // see ShowWindow for other options
    >31 )                   // numbers lower then 31 are failures
{
    AfxMessageBox( "Successfully created." );
}
else
{
    AfxMessageBox( "Failed to create process." );
}
```

2) 使用::CreateProcess()运行应用程序

以另外的方式运行相同的批处理文件，但具有更多的选项，代码如下所示。

```
    CString str;
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    // specify command
    str = "Wzd.bat";

    // zero out and initialize STARTUPINFO
    memset( &si, 0, sizeof( si ) );
    si.cb = sizeof( si );
    si.dwFlags = STARTF_USESHOWWINDOW;
    si.wShowWindow = SW_SHOW;

    if ( CreateProcess(
       NULL,                          // can be name of process unless
                                      //    batch file, else must be
                                      //    in command line:
       ( char * )LPCSTR( str ),       // command line
       NULL,NULL,                     // security options
       FALSE,                         // if true will inherit all
                                      //    inheritable handles
                                      //    from this process
       NORMAL_PRIORITY_CLASS,         // can also be HIGH_PRIORITY_CLASS
                                      //    or IDLE_PRIORITY_CLASS
       NULL,                          // inherit this process's
                                      // environment block
       NULL,                          // specifies working directory
                                      //    of created process
       &si,                           // STARTUPINFO specified above
       &pi                            // PROCESS_INFORMATION returned
       ) )
    {
       AfxMessageBox( "Successfully created." );
       HANDLE pH = pi.hProcess;

       // wait until application is ready for input
       if ( !WaitForInputIdle( pH,1000 ) )
       {
          // send messages, etc.
       }

       // kill process with 0 exit code
       TerminateProcess( pH, 0 );

    }
    else
    {
       AfxMessageBox( "Failed to create process." );
    }
}
```

3) 获取应用程序的目录

另外一个Windows API调用::GetModuleFileName()将通知应用程序其执行文件所在磁盘的

子目录。当自己的应用程序与在它运行时需要访问的其他文件安装在同一目录中时，以上这一点将特别有用。

为寻找应用程序所在的目录，使用以下代码：

```
// change directory to this application's .exe file
char szBuffer[128];
::GetModuleFileName( AfxGetInstanceHandle(), szBuffer,
   sizeof( szBuffer ) );
char *p = strrchr( szBuffer, '\\' );
*p = 0;
```

SZBuffer变量包括应用程序的主目录名。

为了改变应用程序的当前工作目录到该路径，使用：

```
_chdir(szBuffer);
```

一旦这个目录成为当前目录，应用程序就可以无需指定路径而打开任何文件。

4. 注意

■ 传递给WinExec()的第二个参数与调用 ShowWindow()的参数相同。它包括 SW_HIDE, SW_MAXMIZE, SW_MINIMIZE等。此标记影响正在执行的应用程序的主窗口。

■ 如果在API的命令行参数中写入了路径，确保周围加上了双引号。其中的原因是由于有间隔符“ \ ”，所以以下将认为是三个独立的命令行参数。

```
C:\Program Files\Microsoft Office\prog.exe
```

而使用双引号，则被认为是一个参数。

```
"C:\Program Files\Microsoft Office\prog.exe"
```

■ 当运行批处理文件时，实际上是在运行 CMD.EXE并将批处理文件的名字作为命令行参数传递给它。这就是批处理文件在使用 ::CreateProcess()时不能作为第一个参数的原因。

■ 在本例中，使用了 TerminateProcess()删除已运行的文件。然而，这种方法可能不妥——应用程序在未结束前不能被清除。一种更好的方法是向应用程序发送消息使它关闭。由于MFC没有提供相应的函数，因此需要自己创建 (参考实例58)。如果应用程序无法关闭，则使用TerminateProcess()。

■ 与其运行其他应用程序，倒不如在当前的应用程序中创建简单的处理线程。应用程序本身就是可以创建其他线程的线程。由于 Windows是多任务处理系统，因此每个创建的线程都可以同时运行，而且在它们完成自己的任务时将通知应用程序。由于线程与应用程序在相同的程序地址空间中运行，所以应用程序和线程之间的数据传递与应用程序之间的数据传递相比要容易得多。创建线程而非应用程序的例子请参考实例 57。

■ 与新创建的应用程序进行通信请参考实例 49。若与它共享数据则请参考实例 50。

5. 使用光盘时注意

运行随书附带光盘上的工程时，在 Mainfrm.cpp中的OnWzd1Test()或者OnWzd2Test()函数中设置断点。单击Test菜单的Wzd1或者Wzd2，注意函数如何运行批处理文件 wzd.bat。

14.3 实例55：改变优先级

1. 目标

改变应用程序或者线程的优先级，使其在后台运行得更快或者更慢。

2. 策略

改变应用程序的优先级，使用::SetPriority() API。改变线程的优先级则使用::SetThreadPriority() API。

3. 步骤

1) 改变应用程序的优先级

改变应用程序的优先级，使用：

```
::SetPriorityClass(
   ::GetCurrentProcess(),              // process handle
   // REALTIME_PRIORITY_CLASS          // highest: thread must run
                                       //    immediately before any
                                       //    other system task
   // HIGH_PRIORITY_CLASS              // high: time-critical threads
   // NORMAL_PRIORITY_CLASS            // normal: thread with equal
                                       //    importance to other
                                       //    system applications
   IDLE_PRIORITY_CLASS                 // low: threads that can run in
                                       //    the background of the
                                       //    entire system
   );
```

获得当前的优先级：

```
DWORD priority =
   ::GetPriorityClass(
      ::GetCurrentProcess()            // process handle
      );
```

2) 改变线程的优先级

如下所示，在线程中改变线程的优先级：

```
SetThreadPriority(
   THREAD_PRIORITY_TIME_CRITICAL       // highest priority
   THREAD_PRIORITY_HIGHEST             // next highest
   THREAD_PRIORITY_ABOVE_NORMAL        // etc....
   THREAD_PRIORITY_NORMAL
   THREAD_PRIORITY_BELOW_NORMAL
   THREAD_PRIORITY_LOWEST
   THREAD_PRIORITY_IDLE                // lowest priority
   );
```

4. 注意

设置优先级并不总是有助于多任务处理。与常识相反，应该为密集占用 CPU的应用程序或者线程赋予较低的优先级。而几乎不需要处理的应用程序或者线程应该赋予较高的优先级。使用较低的优先级，自己可以不必把密集占用 CPU的应用程序分割成CPU处理段，使它在后台运行。给予实时应用程序较高的优先级，因为它们需要立刻处理所接收的消息，例如键盘按下或者从串行设备来的消息。

5. 使用光盘时注意

将Wzd.cpp中的SetPriorityClass()改变为较高或者较低的优先级。设置较高的优先级，然后创建应用程序并运行它。使用 File/Open菜单命令打开对话框。接着在系统中运行另外一个

应用程序，例如用Windows Explorer来查找文件。在运行中，注意当拖动应用程序的对话框时，可以暂停其他的应用程序。

14.4 实例56：应用程序内部的多任务——工作者线程

1. 目标

创建程序线程处理数学运算或者其他需要 CPU集中处理的函数，该线程将单独与应用程序同时运行。

2. 策略

使用框架的AfxBeginThread()函数创建线程。为了在线程完成任务时通知主应用程序，需创建自己的Windows消息，线程在任务完成后将该消息发送给应用程序。

3. 步骤

1) 设置工作者线程

定义用于向线程传递数据的数据结构，一个实例结构可以如下所示：

```
typedef struct t_THREADDATA
{
   HWND hDoneWnd;              // window handle of main app to send messages
   int nData;                  // data to process
} THREADDATA;
```

用下述语法编写工作者线程函数。参数 pParam是前面定义的数据结构的指针。参考程序清单——工作者线程，可以查看其完整代码列表：

```
UINT WzdThread( LPVOID pParam )
{
   // get data from thread creator
   THREADDATA *pData = ( THREADDATA * )pParam;

   // do calulations
   for ( int i = pData -> nData;i < 1000000;i++ )
   {
   }

   // save data back to thread creator
   pData -> nData = i;

   // tell creator we're done
   ::SendMessage( pData -> hDoneWnd, WM_DONE, 0, 0 );

   return 0;
}
```

返回的值由用户定义。父应用程序可以使用 GetExitThread()检索到这一代码。使用AfxEndThread(arg)将终止工作者线程，arg的值也将由用户自己定义。

2) 创建工作者线程

为创建线程首先使用适当的数据初始化数据结构，然后调用 AfxBeginThread()指定上面创建的线程函数。确保在调用的类中嵌入了数据结构，这样在创建线程以后它可以驻留下来:

```
THREADDATA m_ThreadData;
```

```
: : :
ThreadData.nData = 123;
ThreadData.hDoneWnd = m_hWnd;
AfxBeginThread(
   WzdThread,                    // static thread process declare
   &m_ThreadData                 // data to send to thread
   );
```

3) 当线程完成任务时通知应用程序

为了在线程完成任务时通知主应用程序，需要创建如下的 Windows消息：

```
#define WM_DONE WM_USER + 1
```

如下所示，当线程完成任务时，发送该消息：

```
// tell creator we're done
::SendMessage( pData -> hDoneWnd, WM_DONE, 0, 0 );
```

为了让应用程序接收该消息，需要手工添加消息处理函数。首先在接收窗口的消息映射中添加ON_MESSAGE_VOID()消息宏。确保这些代码在{{}}括号内，否则ClassWizard不会处理它:

```
BEGIN_MESSAGE_MAP( CWzdView, CView )
   // {{AFX_MSG_MAP( CWzdView )
   // }}AFX_MSG_MAP
   ON_MESSAGE_VOID( WM_DONE,OnDone )
END_MESSAGE_MAP()
```

如下添加消息处理函数：

```
void CWzdView::OnDone()
{
}
```

确保在类的.h文件中也定义了这个消息处理函数。使用 ON_MESSAGE_VOID() 而不是ON_MESSAGE()宏以避免处理后者所需的多余参数。

4. 注意

■ 由于线程和应用程序占用了相同的地址空间，因此可通过数据结构向线程传递指针。但是不能够向CWnd结构传递指针。这是因为应用程序跟踪每一个它创建的实例和它指向的窗口句柄，因而可以立即把 CWnd指针转换为窗口句柄。同时每个线程也跟踪它所创建的窗口。但是如果向线程传递CWnd对象指针，线程由于没有其窗口句柄的记录，不能为窗口进行转换。此时可以传递的只能是窗口句柄本身。如果用户愿意，仍然可以在线程内部把窗口指针封装在CWnd对象中。

■ 为避免两个线程同时写入相同的数据区域请参考实例 59。

■ AfxBeginThread()函数仅仅是一个辅助函数，它创建 CWinThread类的实例，并从线程中调用用户所提供的函数。当函数返回或者调用 AfxEndThread()时，辅助程序将调用::PostQuitMessage ()终止线程。

■ 工作者线程的典型用途包括电子数据表格中每一栏的数学计算功能或者文件应用程序中的后台拼写检查。请参考实例51和52了解涉及同步套接字和端口通信的有关内容。

■ 如果需要一次运行几个线程，用户在定义用于通知应用程序线程已执行完毕其任务的all done消息中，可能将同时包括线程标识号，为此，需要在消息处理函数中使用 ON_

MESSAGE ()消息宏：

```
LRESULT CMainFrame::OnDone( WPARAM wParam,LPARAM lParam )
{
    return 0;
}
```

5. 使用光盘时注意

当运行随书附带光盘上的工程时，在 Wzdview.cpp中的OnTestWzd()和OnDone()中设置断点。单击Test和Wzd菜单命令。观察如何创建线程和报告线程结束。

```
#ifndef WZDTHREAD_H
#define WZDTHREAD_H

#define WM_DONE WM_USER + 1

typedef struct t_THREADDATA
{
    HWND hDoneWnd;
    int nData;
} THREADDATA;

UINT WzdThread( LPVOID pParam );

#endif
// WzdThread.cpp : thread process
//

#include "stdafx.h"
#include "WzdThread.h"

////////////////////////////////////////////////////////////////////////////
// CWzdThread

UINT WzdThread( LPVOID pParam )
{
    // get data from thread creator
    THREADDATA *pData = ( THREADDATA * )pParam;

    // do calulations
    for ( int i = pData -> nData;i < 1000000;i++ )
    {
    }

    // save data back to thread creator
    pData -> nData = i;

    // tell creator we're done
    ::SendMessage( pData -> hDoneWnd, WM_DONE, 0, 0 );

    return 0;
// return value up to you—parent can retrieve with GetExitCodeThread();
// can also call AfxEndThread(0) where the meaning of the argument is up to you
}
```

14.5 实例57：应用程序内部的多任务——用户界面线程

1. 目标

创建一个单独执行某功能，且和应用程序同时运行的线程，而且该线程需要自己的用户界面，例如在文档应用程序中的查询和替换功能。

2. 策略

和创建工作者线程一样，需要使用框架中的 AfxBegin Thread ()函数来创建用户界面线程。这一次将对线程具有完全控制权，而不是像实例 56那样，只具拥有对工作者线程的部分控制权。因此本例将创建自己的CWinThread派生线程类。

3. 步骤

1) 创建新的线程类

使用ClassWizard创建CWinThread派生线程类。例如创建无模式对话框的线程类，请参考程序清单 —— 用户界面线程类。在本例中创建无模式对话框而不是有模式对话框的原因是，允许消息从主应用程序连续地转发到线程。

2) 创建用户界面线程

为启动线程可以使用如下代码：

```
CWinThread *pThread = AfxBeginThread( RUNTIME_CLASS( CWzdThread ) );
```

线程需要调用::PostQuitMessage(arg)来终止，这里的arg参数需要用户自己定义。应用程序为了获得arg的值，可以调用如下代码：

```
int arg = pThread -> GetExitCodeThread();
```

注意 对于应用程序直接结束线程没有推荐的方式。线程必须自己退出并允许将自身清除。用户需要做的是创建Windows消息来通知线程终止。线程通过调用::PostQuitMessage (arg)来处理消息。请参考实例58以了解如何向线程发送消息。

4. 注意

■ 工作者线程倾向于琐碎的处理，与它不同的是，用户界面线程具有自己的界面而且实际上类似于运行其他应用程序。创建线程而不是其他应用程序的好处是线程可与应用程序共享程序空间，这样可以简化线程与应用程序共享数据的功能。

■ 典型情况是用户界面线程用于完成查询和替换等功能，或者是其他不希望占用主应用程序大量处理时间但是需要一个界面的功能或服务，或者用户也可完全不考虑界面，将这种类型的线程用于窗口消息服务器作为一种传递其消息的方式，以避免使自己因占用处理时间过多而陷入困境。

■ 在时间要求严格的应用程序 (例如实时应用程序)中，不希望因为工作者线程启动而等待，这时可将工作者线程中的控制逻辑内置到用户界面线程中并提前创建线程。当需要处理事务时，向用户界面线程发送消息，此时用户界面线程已经运行并且在等待指令。

5. 使用光盘时注意

运行随书附带光盘上的工程时，单击 Test和Wzd菜单命令，创建包含无模式对话框的线程。

6. 程序清单 —— 用户界面线程类

```
#if !defined( AFX_WZDTHREAD_H__411AE4C2_E515_11D1_9B80_00AA003D8695__INCLUDED_ )
```

```
#define AFX_WZDTHREAD_H__411AE4C2_E515_11D1_9B80_00AA003D8695__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif    // _MSC_VER >= 1000
// WzdThread.h : header file
//

#include "WzdDialog.h"

/////////////////////////////////////////////////////////////////////////////
// CWzdThread thread

class CWzdThread : public CWinThread
{
    DECLARE_DYNCREATE( CWzdThread )
protected:
    CWzdThread();    // protected constructor used by dynamic creation

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    // {{AFX_VIRTUAL( CWzdThread )
public:
    virtual BOOL InitInstance();
    virtual int ExitInstance();
    // }}AFX_VIRTUAL

// Implementation
protected:
    virtual ~CWzdThread();

    // Generated message map functions
    // {{AFX_MSG( CWzdThread )
    // NOTE - the ClassWizard will add and remove member functions here.
    // }}AFX_MSG

    DECLARE_MESSAGE_MAP()
private:
    CWzdDialog m_dlg;
};

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

// {{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately
```

```
//     before the previous line.

#endif
    // !defined( AFX_WZDTHREAD_H__411AE4C2_E515_11D1_9B80_00AA003D8695__INCLUDED_ )
// WzdThread.cpp : implementation file
//

#include "stdafx.h"
#include "wzd.h"
#include "WzdThread.h"

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

/////////////////////////////////////////////////////////////////////////////
// CWzdThread

IMPLEMENT_DYNCREATE( CWzdThread, CWinThread )

CWzdThread::CWzdThread()
{
}

CWzdThread::~CWzdThread()
{
}

BOOL CWzdThread::InitInstance()
{
    m_dlg.Create( IDD_WZD_DIALOG );
    m_dlg.ShowWindow( SW_SHOW );
    m_pMainWnd = &m_dlg;

    return TRUE;    // can end thread by returning FALSE here
}

int CWzdThread::ExitInstance()
{
    m_dlg.DestroyWindow();
    return CWinThread::ExitInstance();
}

BEGIN_MESSAGE_MAP( CWzdThread, CWinThread )
    // {{AFX_MSG_MAP( CWzdThread )
    // NOTE - the ClassWizard will add and remove mapping macros here.
    // }}AFX_MSG_MAP
END_MESSAGE_MAP()
```

14.6 实例58：向用户界面线程发送消息

1. 目标

向用户界面线程发送消息。

2. 策略

一般通过使用CWnd::SendMessage()来发送消息。但是只能使用 SendMessage()向窗口发送消息，而不是线程。因此本例中使用了另外一个函数 CWinThread：：PostThreadMessage()来发送消息。PostThreadMessage()允许直接访问线程类中的消息映射。在本例中，将创建自定义的窗口消息，它会导致用户界面线程终止。

注意 仍然可以用SendMessage ()向线程发送消息，但仅限于线程创建的窗口。

3. 步骤

向线程发送消息

如下所示定义窗口消息：

```
#define WM_WZDKILLTHREAD WM_USER + 1
```

按通常方式启动用户界面线程，但保留一个指向由它创建的 CWinThead对象的指针：

```
m_pThread = AfxBeginThread( RUNTIME_CLASS( CWzdThread ) );
```

向线程发送消息，如下所示使用 PostThreadMessage()：

```
m_pThread  -> PostThreadMessage( WM_WZDKILLTHREAD, 0, 0 );
```

为了让线程能截取消息，在线程类的消息映射中手工添加下列消息映射宏，确保在 {{}} 括号外边添加代码，以便ClassWizard不会被它弄糊涂:

```
BEGIN_MESSAGE_MAP( CWzdThread, CWinThread )
   // {{AFX_MSG_MAP( CWzdThread )
   // }}AFX_MSG_MAP
   ON_THREAD_MESSAGE( WM_WZDKILLTHREAD,OnKillThread )
END_MESSAGE_MAP()
```

使用下列语法编写消息处理函数。在本例中，还要调用 ::PostQuitMessage()函数来终止线程:

```
LRESULT CWzdThread::OnKillThread( WPARAM wParam,LPARAM lParam )
{
   ::PostQuitMessage( 0 );
   return 0;                    // returned to PostThreadMessage()
}
```

同时确保在线程类的头文件中定义了这个消息处理函数。

4. 注意

■ 由于不能访问工作者线程的消息映射，因此不能使用 PostThreadMessage()向它发送消息。通过传递给它的结构中的数据单元，可以非正式地与工作者线程通信。q例如，用户可以在这个结构中包含bKill标记，当由应用程序设置该标志时将导致线程过早地异常中断。

■ 用户界面线程能够有选择地截获已注册消息 (有关已注册消息，请参考实例 49)。为处理注册消息，可使用以下消息宏：

```
ON_REGISTER_THREAD_MESSAGE( registered_message_id, process )
```

由于线程被认为是创建应用程序中的一部分，考虑到每个注册消息总是要占用一定资源，因此使用已注册消息与用户界面线程通信将可能会导致它们负担过重。

5. 使用光盘时注意

当运行随书附带光盘上的工程时，单击 Test和Wzd菜单命令，创建包含无模式对话框的线程。接着单击End向线程发送消息来结束线程。

14.7 实例59：线程间的数据共享

1. 目标

在线程之间进行应用程序数据共享。同时避免由于两个线程同时访问相同的数据而引发的冲突。

2. 策略

使用三种MFC类：CMutex、CSingleLock和CMultiLock来同步多个线程对一个数据类的同时访问。

3. 步骤

数据对象防火墙

在线程中确定共享的数据类。在每个类定义中嵌入 CMutex对象，如下所示：

```
class CWzdData : public CObject
{
   :  :  :
   // synchronization protection
   CMutex m_mutex;
   :  :  :
};
```

如果数据类没有访问其数据的成员函数，这一步将添加它们。这些函数如下所示:

```
void CWzdData::GetData( int *pInt,float *pFloat,DWORD *pWord )
{
   *pInt = m_nInt;
   *pFloat = m_fFloat;
   *pWord = m_dwWord;
}

void CWzdData::SetData( int nInt,float fFloat,DWORD dwWord )
{
   m_nInt = nInt;
   m_fFloat = fFloat;
   m_dwWord = dwWord;
}
```

在引用已嵌入 CMutex变量的 SetData()函数堆栈上创建 CSingleLock类的实例。使用CSingleLock的Lock()函数避免在函数内部对数据多重访问，如下所示：

```
BOOL CWzdData::SetData( int nInt,float fFloat,DWORD dwWord )
{
   CSingleLock slock( &m_mutex );

   if ( slock.Lock( 1000 ) )          // timeout in milliseconds,
```

```
                                        //     default = INFINITE
    {
        // set values—can also be lists and arrays
        m_nInt = nInt;
        m_fFloat = fFloat;
        m_dwWord = dwWord;
        return TRUE;
    }
    return FALSE;                       // timed out!

    // unlocks on return or you can call slock.Unlock();
}
```

如果其他的线程同时访问这个数据， Lock()将立刻返回。否则， Lock()在指定的毫秒数内等待，直到超时并返回FALSE。

如果在这个类中保存的数据与其他类中保存的数据相关，则在两个类中嵌入 CMutex变量，两边都用CMultiLock等待，如下所示：

```
CMutex mutex[2];
mutex[0] = &mutex1;
mutex[1] = &mutex2;
CMultiLock mlock( mutex,2 );      // where 2 is the number of mutexes
if ( mlock.Lock( 1000 ) )
{
}
```

要查看经过这些修改后的数据类实例，请参考本实例结尾的程序清单 —— 实例数据类。

4. 注意

■ 要了解更多的有关 CMutex和相关类的知识，请参阅第 1章。CMutex使用了实例 3中的::CreateMutex()Windows API以避免若干应用程序的实例同时运行。 ::CreateMutex()函数的功能并不仅仅只是追踪应用程序的实例。在该实例中只是简单使用其中的部分功能。

■ Mutex实际应用的例子可参考实例 51。通过Windows套接字线程， CObList可以安全地实现多重访问。

5. 使用光盘时注意

当运行随书附带光盘上的工程时，在 WzdData.cpp的GetData()中设置断点。单击 Test和Wzd菜单命令，观察三个线程如何试图访问同一个数据。

6. 程序清单 —— 数据类

```
#ifndef WZDDATA_H
#define WZDDATA_H

#include <afxmt.h>

class CWzdData : public CObject
{
public:
    DECLARE_SERIAL( CWzdData )

    CWzdData();
```

```
    BOOL GetData( int *pInt,float *pFloat,DWORD *pWord );
    BOOL SetData( int nInt,float fFloat,DWORD dwWord );

    // synchronization protection
    CMutex m_mutex;

    // result data
    int m_nInt;
    float m_fFloat;
    DWORD m_dwWord;

};
#endif
// WzdData.cpp : implementation of the CWzdData class
//

#include "stdafx.h"
#include "WzdData.h"

////////////////////////////////////////////////////////////////////////////
// CWzdData

IMPLEMENT_SERIAL( CWzdData, CObject, 0 )


CWzdData::CWzdData()
{
    m_nInt = 0;
    m_fFloat = 0.0f;
    m_dwWord = 0;
}

BOOL CWzdData::GetData( int *pInt,float *pFloat,DWORD *pWord )
{
    // we lock here too so that we'll never read half written data
    CSingleLock slock( &m_mutex );

    if ( slock.Lock( 1000 ) )    // timeout in milliseconds, default = INFINITE
    {
        // get values—can also be lists and arrays
        *pInt = m_nInt;
        *pFloat = m_fFloat;
        *pWord = m_dwWord;
        return TRUE;
    }
    return FALSE;    // timed out!

    // unlocks on return or you can call slock.Unlock();
}
```

```
BOOL CWzdData::SetData( int nInt,float fFloat,DWORD dwWord )
{
   CSingleLock slock(&m_mutex);    // or with CMultiLock can specify several
                                   //    m_mutex's for waiting on several
                                   //    data items

   if ( slock.Lock( 1000 ) )       // timeout in milliseconds,
                                   //    default = INFINITE
   {
      // set values—can also be lists and arrays
      m_nInt = nInt;
      m_fFloat = fFloat;
      m_dwWord = dwWord;
      return TRUE;
   }
   return FALSE;    // timed out!

   // unlocks on return or you can call slock.Unlock();
}
```