第二部分　用户界面实例

本部分的实例将把重点放在应用程序用户界面的各个方面，这些界面可以由 Developer Studio、MFC和Visual C++来创建。用户界面开发工具是本部分密切关注的内容，所以该部分内容包括了本书的大部分实例。本部分的各个章节是：

第4章　应用程序和环境

本章的实例覆盖了应用程序与其所处环境交互的各个方面，内容包括应用程序如何运行以及应用程序的外观如何显现。实例的范围包括寻找空间添加标识以及阻止同时运行同一个应用程序等。

第5章　菜单和控件条

这一章的内容涉及应用程序的菜单和控制条，这也许是用户同应用程序之间进行交互的主要方式。AppWizard自动地为应用程序添加基本的菜单、工具栏和状态栏，但是这些菜单和控制条是很简单的，连 Developer Studio的都比不上。自己动手费点力可以为自己的应用程序增加与Developer Studio类似的外观。

第6章　视

如果创建SDI或者MDI应用程序，则此时应用程序的视将是用户与应用程序，特别是与正在编辑中的属于应用程序的文档之间进行交互的主要方式。本章所有的实例都与视相关，从创建一个属性表、打印视到拖放文件到视中等等。

第7章　对话框和对话条

对话框和对话条用于提示用户使用不同类型的控件，比如按钮和列表框等等。 MFC和Developer Studio自动创建对话框和对话框类。本章将探讨手工修改对话框的各种方式，使之更加符合用户的要求。

第8章　控件窗口

对话框上添加了按钮和编辑框等控件窗口(操作系统提供的子窗口)。不仅可以在对话框中采用它们，还可以把它们放在视、控制条和任何窗口之内。程序员甚至可以自己绘制它们。

第9章　绘图

位图和图标可以为自己的应用程序增加颜色和风格。因为所有的窗口界面都类似，所以标识符和splash屏幕才是真正区别不同程序外观的唯一方式。显而易见，绘图对创建自己的控件以及在CAD应用程序中显示图表也是很重要的。

第10章 帮助

比起阅读手册里的各种命令，在线帮助有助于大量减少学习应用程序所需要的时间和困难，用户可以直接查询控件，本章实例描述了如何实现 3种类型的在线帮助。

第11章 普通窗口

视、对话框和控制条、控件窗口和工具栏等组成了 MFC应用程序的界面，但它们都是建立在低级窗口的基础之上。为什么在其他元素都很复杂的情况下却采用这样类型的窗口呢？这是因为，在这个级别上编写程序可以达到高级窗口才能达到的效果。

第12章 特定的应用程序

本章用一些专门的 MFC应用程序来概述全部分。本章的实例包括一对简单的文本编辑器和两个数据库编辑器。其中一个实例提供了创建自己的浏览器风格应用程序的全部内容。最后，本章还包括了如何创建简单的 Wizard的内容。

第 4 章 应用程序和环境

本章探讨应用程序和环境之间在各方面如何交互。本章的内容包括所有应用程序的运行方式以及它们向用户呈现出的外观表现。这里的实例有：寻找新地方来添加标识符，在环境中添加图标以及防止同时运行两个相同的应用程序。这些实例具体包括：

实例1 在工具栏中添加一个静态标识符，即在工具栏中添加一个程序附属的图片，使之一直可见。

实例2 在工具栏中添加一个动态标识符 ,即上例中静态标识符的动态版本。

实例3 只启动一个实例，本例将防止同时运行应用程序中的多个实例。

实例4 创建一个对话框/MDI混合式应用程序，本例将创建这两类标准的应用程序类型。

实例5 在系统托盘(System Tray)中添加图标，本例将单击系统托盘中用户自己的图标(系统托盘是一个位于桌面上的图标集，与时间显示相邻)。

实例6 在主窗口标题栏上显示标识符，本例利用应用程序的标题栏来显示标识符或其他位图。

4.1 实例1：在工具栏中添加静态标识符

1. 目标

如图4-1所示，在应用程序工具栏的右上角处显示一个标识符。

2. 策略

工具栏是单控件窗口，可以在其客户区内绘制各类按钮。当用户单击某个按钮时，工具栏就会根据该按钮的图像来触发相应的命令。这里只是简单地给应用程序的工具栏添加一个子窗口，而且该子窗口远离其他按钮。所添加的子窗口是一个普通窗口，它可以显示一幅位图，并且随着工具栏尺寸的改变而不断移动。

3. 步骤

1) 创建一个用来绘制标识符的普通窗口

用Class Wizard 创建一个来自generic CWnd的通用窗口。

在该通用窗口中，用光盘中为本例提供的位图类嵌入一个位图变量，如下所示：

```
CWzdBitmap m_bitmap；
```

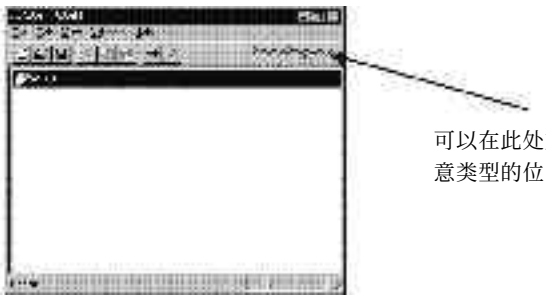

图4-1　在工具栏中添加一个静态标识符

注意　在本系列的第一本书中，创建位图类时不仅装入了一个位图资源，而且也装入了位图的调色板,其目的是用来准确地修改位图的颜色。

在该类中添加一个CreateBX()成员函数，该函数将装入位图标识符，并同时在工具栏中创建一个实际的窗口:

```
void CWzdLogo::CreateBX( CWnd *pWnd, UINT nBitmapID, UINT nChildID )
{
   m_bitmap.LoadBitmapEx( nBitmapID,TRUE );
   CRect rect( 0,0,0,0 );    // will be resizing constantly anyways
   Create( NULL,_T( " " ),WS_CHILD|WS_VISIBLE, rect, pWnd, nChildID );
}
```

添加另一个成员函数，该函数使得工具栏可以移动窗口:

```
void CWzdLogo::MoveLogo( int nWidth, int nHeight )
{
   MoveWindow( nWidth-m_bitmap.m_Width,0,m_bitmap.m_Width,nHeight );
}
```

应用Class Wizard添加一个在窗口中绘制位图的WM_PAINT消息处理函数句柄，并以此来结束该类:

```
void CWzdLogo::OnPaint()
{
   CPaintDC dc( this );    // device context for painting

   // get bitmap colors
   CPalette *pOldPal =
      dc.SelectPalette( m_bitmap.GetPalette(), FALSE );
   dc.RealizePalette();

   // get device context to select bitmap into
   CDC dcComp;
   dcComp.CreateCompatibleDC( &dc );
   dcComp.SelectObject( &m_bitmap );

   // draw bitmap
   dc.BitBlt( 0,0,m_bitmap.m_Width, m_bitmap.m_Height, &dcComp,
      0,0,SRCCOPY );
   // reselect old palette
```

```
    dc.SelectPalette( pOldPal, FALSE );
}
```

要查看该普通窗口类的总体情况，请参考本实例结尾处的程序清单——标识符类。

2) 创建一个包含普通窗口类的自定义工具栏

使用Class Wizard 从CToolBarCtrl中创建一个新类。编辑所有生成的 .h和.cpp文件，以修改所有从CToolBarCtrl 到CToolBar的引用(ClassWizard不允许从CToolBar类派生一个类)。

在新的工具栏中嵌入新的普通窗口类，如下所示：

```
CWzdLogom_Logo;
```

使用Class Wizard给工具栏添加一个WM_CREAT消息处理函数句柄。用该句柄调用普通窗口类的CreatBX()函数:

```
int CWzdToolbar::OnCreate( LPCREATESTRUCT lpCreateStruct )
{
    if ( CToolBar::OnCreate( lpCreateStruct ) = -1 )
        return -1;

    m_Logo.CreateBX( this,IDB_LOGO,-1 );

    return 0;
}
```

再一次使用Class Wizard来添加一个WM_SIZE消息处理函数，这里要调用普通窗口类的MoveLogo()函数：

```
void CWzdToolbar::OnSize( UINT nType, int cx, int cy )
{
    CToolBar::OnSize( nType, cx, cy );

    m_Logo.MoveLogo( cx,cy );
}
```

要看该工具栏类的总体，请参考本实例结尾处的程序清单——工具栏类。

3) 在应用程序中使用新的工具栏类

现在，在MainFrm.h文件中用新的工具栏类替换CToolBar:

```
// change CToolBar to CWzdToolBar
CWzdToolbar    m_wndToolBar;
```

不幸的是，本实例对可停靠的工具栏就不起作用了。这是因为可停靠的工具栏仅仅与它所包括的最后一个按钮长度相同。而该实例却依赖于具有大量额外空间的工具栏，以便在结尾处粘贴普通窗口。因此必须在注释工具栏处将其设为非停靠式 (un-dockable)，或者在CmainFrame::OnCreat()中删除以下各列：

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

    // comment out or delete these lines
//  m_wndToolBar.EnableDocking( CBRS_ALIGN_ANY );
//  EnableDocking( CBRS_ALIGN_ANY );
//  DockControlBar( &m_wndToolBar );
```

```
    return 0;
}
```

4. 注意

■ 实际上，可以利用本实例在父窗口的任何空白区粘贴标识符。只要在父窗口的类中嵌入这个窗口，并使用 Class Wizard 添加一个WM_CREAT消息处理函数句柄就可以创建标识符窗口。

■ 如果想在可停靠的工具栏中粘贴一个标识符，请参考第 2章。在第2章中，你会发现：通过在CToolBar 类中重载CalCDynamicLayout()可以扩大工具栏的尺寸使之包含你的标识符。无论如何都要确保提供该标识符的水平和垂直两个版本。

5. 使用光盘时注意

运行随书附带光盘上的工程时，会注意到右手方向的工具栏上写有 Your Company的字样。

6. 程序清单 —— 标识符类

```
#if !defined WZDLOGO_H
#define WZDLOGO_H

// WzdLogo.h : header file
//
#include "WzdBitmap.h"
/////////////////////////////////////////////////////////////////////////////
// CWzdLogo window

class CWzdLogo : public CWnd
{
// Construction
public:
    CWzdLogo();

// Attributes
public:

// Operations
public:
    void CreateBX( CWnd *pWnd, UINT nBitmapID, UINT nChildID );
    void MoveLogo( int nWidth, int nHeight );

// Overrides
    // ClassWizard generated virtual function overrides
    // {{AFX_VIRTUAL( CWzdLogo )
    // }}AFX_VIRTUAL

// Implementation
public:
    virtual ~CWzdLogo();

    // Generated message map functions
```

```
protected:
    // {{AFX_MSG(CWzdLogo)
    afx_msg void OnPaint();
    // }}AFX_MSG
    DECLARE_MESSAGE_MAP()
private:
    CWzdBitmap m_bitmap;
};

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

#endif

// WzdLogo.cpp : implementation file
//

#include "stdafx.h"
#include "WzdLogo.h"

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

/////////////////////////////////////////////////////////////////////////////
// CWzdLogo

CWzdLogo::CWzdLogo()
{
}

CWzdLogo::~CWzdLogo()
{
}


BEGIN_MESSAGE_MAP( CWzdLogo, CWnd )
    // {{AFX_MSG_MAP( CWzdLogo )
    ON_WM_PAINT()
    // }}AFX_MSG_MAP
END_MESSAGE_MAP()


/////////////////////////////////////////////////////////////////////////////
// CWzdLogo message handlers

void CWzdLogo::OnPaint()
{
    CPaintDC dc( this );    // device context for painting
```

```
    // get bitmap colors
    CPalette *pOldPal = dc.SelectPalette( m_bitmap.GetPalette(),FALSE );
    dc.RealizePalette();

    // get device context to select bitmap into
    CDC dcComp;
    dcComp.CreateCompatibleDC( &dc );
    dcComp.SelectObject( &m_bitmap );

    // draw bitmap
    dc.BitBlt( 0, 0, m_bitmap.m_Width,m_bitmap.m_Height, &dcComp, 0, 0, SRCCOPY );

    // reselect old palette
    dc.SelectPalette(pOldPal,FALSE);
}

void CWzdLogo::CreateBX( CWnd *pWnd, UINT nBitmapID, UINT nChildID )
{
    m_bitmap.LoadBitmapEx( nBitmapID,TRUE );
    CRect rect( 0,0,0,0 );
    Create( NULL,_T( " " ),WS_CHILD|WS_VISIBLE,rect,pWnd,nChildID );
}


void CWzdLogo::MoveLogo( int nWidth, int nHeight )
{
    MoveWindow( nWidth-m_bitmap.m_Width,0,m_bitmap.m_Width,nHeight );
}
```

7. 程序清单 —— 工具栏类

```
#if !defined WZDTOOLBAR_H
#define WZDTOOLBAR_H

// WzdToolbar.h : header file
//
#include "WzdLogo.h"
/////////////////////////////////////////////////////////////////////////////
// CWzdToolbar window

class CWzdToolbar : public CToolBar
{

// Construction
public:
    CWzdToolbar();

// Attributes
public:

// Operations
public:
```

```
// Overrides
    // ClassWizard generated virtual function overrides
    // {{AFX_VIRTUAL(CWzdToolbar)
    // }}AFX_VIRTUAL

// Implementation
public:
    virtual ~CWzdToolbar();

    // Generated message map functions
protected:
    // {{AFX_MSG(CWzdToolbar)
    afx_msg void OnSize( UINT nType, int cx, int cy );
    afx_msg int OnCreate( LPCREATESTRUCT lpCreateStruct );
    // }}AFX_MSG

    DECLARE_MESSAGE_MAP()
private:
    CWzdLogo m_Logo;
};

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

#endif
// WzdToolbar.cpp : implementation file
//

#include "stdafx.h"
#include "wzd.h"
#include "WzdToolbar.h"

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

/////////////////////////////////////////////////////////////////////////////
// CWzdToolbar

CWzdToolbar::CWzdToolbar()
{
}

CWzdToolbar::~CWzdToolbar()
{
}


BEGIN_MESSAGE_MAP( CWzdToolbar, CToolBar )
```

```
    // {{AFX_MSG_MAP(CWzdToolbar)
    ON_WM_SIZE()
    ON_WM_CREATE()
    // }}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////////
// CWzdToolbar message handlers

int CWzdToolbar::OnCreate( LPCREATESTRUCT lpCreateStruct )
{
    if ( CToolBar::OnCreate(lpCreateStruct) = -1 )
        return -1;

    m_Logo.CreateBX( this,IDB_LOGO,-1 );

    return 0;
}

void CWzdToolbar::OnSize( UINT nType, int cx, int cy )
{
    CToolBar::OnSize( nType, cx, cy );

    m_Logo.MoveLogo( cx,cy );
}
```

4.2　实例2：在工具栏中添加动态标识符

1. 目标

如图4-2所示，要在应用程序工具栏的右上角处播放一个 AVI文件 (动态位图)：

2. 策略

创建自己的工具栏，并在其较远的一角粘贴一个动画控件。从MFC的CTool Bar类中产生工具栏，并用MFC的CAnitmateCtrl 类创建动画控件。

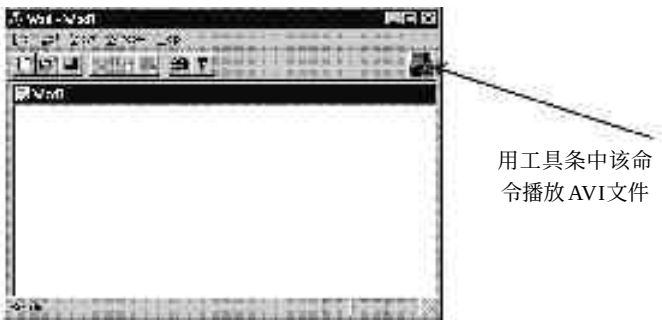

图4-2　在工具栏添加一个动态标识符

3. 步骤

1) 给应用程序资源添加一个AVI文件

单击Developer Studio的Insert和Resources菜单命令，将会产生 Insert Resources对话框。单击Import并定位要找的AVI文件。当提示资源类型时，输入 AVI。该命令会把指定的 AVI文件拷贝到项目的 \res文件，并同时在项目资源中添加一个AVI文件夹。使用鼠标右键单击资源 ID号，给它起一个恰当的名字。本实例将它命名为：IDR_MFC2。

2) 创建一个自定义工具栏类

使用Class Wizard创建一个来自CToolBarCtrl的新类。用文本编辑器修改所有CToolBarCtrl对CToolBar的引用(注意：Class Wizard通常不允许从CToolBar中派生类)。

在新的工具栏类中嵌入一个CAnimateCtrl类：

```
.h:
CAnimateCtrl m_AnimateCtrl;
```

使用Class Wizard给新的工具栏类中添加一个WM_CREART消息处理函数。使用该句柄创建上一步中要嵌入的动画控件窗口。同时装载前面创建的 AVI资源，并将其初始化为播放三次:

```
int CWzdToolbar::OnCreate( LPCREATESTRUCT lpCreateStruct )
{
   if ( CToolBar::OnCreate(lpCreateStruct) = -1 )
      return -1;

   m_AnimateCtrl.Create( WS_CHILD| WS_VISIBLE| ACS_CENTER,
      CRect( 0,0,0,0 ), this, IDC_ANIMATE_CTRL );
   m_AnimateCtrl.Open( IDR_MFC2 );
   m_AnimateCtrl.Play( 0,-1,3 );    // play three times initially

   return 0;
}
```

注意在初始时创建的动画控件窗口是没有尺寸的 (CRect(0,0,0,0))。这是因为程序员无论如何都会修改它的尺寸，因此在这里尺寸就不重要了。

使用Class Wizard给工具栏类添加一个WM_SIZE消息处理函数。在WM_SIZE消息处理函数中将确定工具栏的当前尺寸，并将动画控件移动到左边。程序代码如下：

```
void CWzdToolbar::OnSize( UINT nType, int cx, int cy )
{
   CToolBar::OnSize( nType, cx, cy );

   CRect rect;
   GetWindowRect( &rect );
   ScreenToClient( &rect );
   rect.left = rect.right-32;
   m_AnimateCtrl.MoveWindow( rect );
}
```

下面还要给工具栏添加两个辅助函数，以便允许其他类来播放这个 AVI文件：

```
void CWzdToolbar::PlayLogo()
{
   m_AnimateCtrl.Play( 0,-1,-1 );
}

void CWzdToolbar::StopLogo()
{
   m_AnimateCtrl.Stop();
}
```

要看该工具栏类的总体，请参考本实例结尾处的程序清单 —— 工具栏类。

3) 在CMainFrame中使用新的工具栏类

用新的工具栏类替换当前由 MainFrame使用的类:

```
// use CWzdToolbar for toolbar in Mainfrm.h
CWzdToolbar    m_wndToolBar;
```

同样必须通过注释MainFrm.cpp中的以下程序行将工具栏的浮动和停靠性能置为禁用。否则，工具栏将会自动改变其尺寸，使得它仅能包含它自己的按钮:

```
// disable toolbar docking in Mainfrm.cpp
   // TODO: Delete these three lines if you don't want the toolbar to
   //    be dockable
//  m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
//  EnableDocking(CBRS_ALIGN_ANY);
//  DockControlBar(&m_wndToolBar);
```

4. 注意

■ 动画控件不能扩展到工具栏控件实际的边界。这是因为工具栏控件在其周围绘制了一个空白边界。如果想将动画控件扩展到工具栏控件的实际边界，请参考站点 www.wdj.com上的WDJ 5/99中的文章“使MFC停靠栏很酷”。

■ AVI文件是由两个或多个已转换成为一个 .avi文件框架的位图文件创建的。进行此转换的应用程序可以在Internet的共享软件包中找到。

5. 使用光盘时注意

运行随书附带光盘上的工程时，会注意到右手方向的工具栏上正在播放一个 AVI文件。

6. 程序清单 —— 工具栏类

```
#if !defined WZDTOOLBAR_H
#define WZDTOOLBAR_H

// WzdToolbar.h : header file
//
/////////////////////////////////////////////////////////////////////////////
// CWzdToolbar window

class CWzdToolbar : public CToolBar
{
// Construction
public:
   CWzdToolbar();

// Attributes
public:

// Operations
public:
   void PlayLogo();
   void StopLogo();

// Overrides
   // ClassWizard generated virtual function overrides
   // {{AFX_VIRTUAL( CWzdToolbar )
```

```
    // }}AFX_VIRTUAL

// Implementation
public:
    virtual ~CWzdToolbar();

    // Generated message map functions
protected:
    // {{AFX_MSG( CWzdToolbar )
    afx_msg void OnSize( UINT nType, int cx, int cy );
    afx_msg int OnCreate( LPCREATESTRUCT lpCreateStruct );
    // }}AFX_MSG

    DECLARE_MESSAGE_MAP()
private:
    CAnimateCtrl m_AnimateCtrl;
};

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

#endif
// WzdToolbar.cpp : implementation file
//

#include "stdafx.h"
#include "wzd.h"
#include "WzdToolbar.h"

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

/////////////////////////////////////////////////////////////////////////////
// CWzdToolbar

CWzdToolbar::CWzdToolbar()
{
}

CWzdToolbar::~CWzdToolbar()
{
}

BEGIN_MESSAGE_MAP( CWzdToolbar, CToolBar )
    // {{AFX_MSG_MAP(CWzdToolbar)
    ON_WM_SIZE()
    ON_WM_CREATE()
    // }}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```
//////////////////////////////////////////////////////////////////////////
// CWzdToolbar message handlers

int CWzdToolbar::OnCreate( LPCREATESTRUCT lpCreateStruct )
{
    if ( CToolBar::OnCreate( lpCreateStruct ) == -1 )
        return -1;

    m_AnimateCtrl.Create(
        WS_CHILD|WS_VISIBLE|WS_DLGFRAME|WS_EX_CLIENTEDGE|ACS_CENTER,
        CRect( 0,0,0,0 ), this, IDC_ANIMATE_CTRL );
    m_AnimateCtrl.Open( IDR_MFC2 );
    m_AnimateCtrl.Play( 0,-1,3 );

    return 0;
}

void CWzdToolbar::OnSize( UINT nType, int cx, int cy )
{
    CToolBar::OnSize( nType, cx, cy );

    CRect rect;
    GetWindowRect( &rect );
    ScreenToClient( &rect );
    rect.left = rect.right-32;
    m_AnimateCtrl.MoveWindow( rect );
}

void CWzdToolbar::PlayLogo()
{
    m_AnimateCtrl.Play(0,-1,-1);
}

void CWzdToolbar::StopLogo()
{
    m_AnimateCtrl.Stop();
}
```

4.3　实例3：只启动一个实例

1. 目标

防止系统在任何时刻运行一个应用程序的多个实例。

2. 策略

MFC和Windows API通常不支持该功能，因此只能自己“动手”来实现。微软推荐的方法是为应用程序在系统中创建一些唯一的东西，这些唯一的东西在运行之前可以检查出来。例如，在本例中将要创建一个名为互斥体 (mutex)的唯一资源。互斥体通常用来同步两个或多个正在使用同一数据区的进程。在实例 59中使用的是封装互斥体的 MFC类，然而，由于必须

设置互斥体的确切名称，因此本例中将直接使用 Windows API。

3. 步骤

1) 设置应用程序

在应用程序类中使用 #define定义一个唯一的名字。确保该名字唯一的办法是使用为 COM 接口提供唯一 ID号的GUID程序发生器，用户可以在 VC++的\BIN目录下找到一个名为GUIDGEN.EXE的程序，它就是GUID程序发生器。使用GUIDGEN.EXE定义唯一的名字的一个实例如下：

```
#define UNIQE_NAME "{F5EFF561-ECB3-11d1-A18D-DCB3C85EBD34}"
```

2) 在InitInstance()中创建互斥体

使用上面在应用程序类的InitInstance()函数的开始处定义的名字来创建一个互斥体，并保存该互斥体的句柄 —— 在关闭互斥体时还要用到该句柄。如果应用程序的另一个实例已经存在，::CreatMutex()函数就会给由其他实例创建的互斥体返回一个句柄，而不是再创建一个新的句柄。为了确定该句柄是否为应用程序已经运行的实例的互斥体的句柄，可以调用GetLastError()函数，如果是，则该函数将返回一个 ERROR_ALREADY_EXISTS错误。可以通过退出InitInstance()函数返回一个FALSE值来终止应用程序的运行。以下代码说明了上述过程是如何完成的：

```
BOOL CWzdApp::InitInstance()
{
   m_hOneInstance = ::CreateMutex( NULL,FALSE,UNIQUE_NAME );
   if ( GetLastError() = ERROR_ALREADY_EXISTS )
   {
      AfxMessageBox( "Application already running!" );
      return FALSE;
   }

   :  :  :
}
```

3) 关闭互斥体

使用Class Wizard 重载应用程序类的ExitInstance()函数，同时关闭该互斥体的句柄：

```
int CWzdApp::ExitInstance()
{
   CloseHandle( m_hOneInstance );
   return CWinApp::ExitInstance();
}
```

4. 注意

■ 不仅可以显示应用程序的实例已经运行的错误信息，而且还可以向其他的实例广播消息，告诉它们将其窗口放置在前台。怎样向其他应用程序广播消息，可以参考实例 49。

■ 关闭句柄是十分有必要的。当程序运行结束时， Windows会自动清除它所创建的所有互斥体。这使得这种方法更具有可靠性 —— 如果应用程序非正常终止，则它就不会有机会调用CloseHandle()函数，也就不会留下阻止应用程序再次运行的互斥体。

■ 另一个确定应用程序是否有其他实例正在运行的方法是使用 CWnd::FindWindow()函数。FindWindow()用一个特殊的窗口标题和窗口类名来查找上一层的所有窗口。用户可以创

建一个唯一但隐藏的普通窗口 (参见实例38)，并使该窗口可以被以后的应用程序实例查找；或者也可以查找应用程序的主窗口。如果主窗口没有唯一且不变的标题，就可以给它起一个唯一的窗口类名以便查找。可以在 CmainFrame 的PreCreatWindow()函数中创建这个唯一的窗口类名。这种方法的缺点是在创建实例和创建主窗口之间存在着一定的时间延迟 —— 这样应用程序的第二个实例可能会逃过检查而被启动，因此微软不推荐这种方法。

5. 使用光盘时注意

运行光盘上的项目两次。在试着运行第二次时，就会出现应用程序已经运行的消息。

4.4　实例4：创建对话框／MDI 混合式应用程序

1. 目标

创建一个由命令行标志所决定的对话框或 MDI应用程序。

2. 策略

使用AppWizard创建一个MDI 应用程序。该应用程序会检查 InitInstance()函数中/d标志，如果该标志置位，则创建的将是一个有模式对话框，而不是 MDI框架窗口。一旦该对话框关闭，则使InitInstance()函数返回一个引起应用程序结束的 FALSE值。

3. 步骤

1) 设置应用程序

使用AppWizard创建一个MDI应用程序。

使用对话框编辑器(Dialog Editor)和ClassWizard创建对话框的模板和类。

使用ClassWizard创建自己的CCommandLineInfo类版本。修改该类以检查/d标志。

2) 修改InitInstance()来创建上述两种应用程序中的一种

用新的CCommandLineInfo 类代替原有的应用程序类：

```
CWzdCommandLineInfo cmdInfo;
ParseCommandLine( cmdInfo );
```

如果命令行中出现 /d，将使用新的对话框类创建有模式对话框。在对话框关闭时，将从InitInstance()返回一个引起应用程序结束的 FALSE值。具体情况请参见以下代码：

```
// if user started with /d flag, start dialog app instead
if ( cmdInfo.m_bDialogFg )
{
   CWzdDialog dlg;
   dlg.DoModal();
   return FALSE;
}
```

4. 注意

本实例的另一类型是创建一个基于命令行选项的　MDI或SDI应用程序。开始时可以用AppWizard创建一个MDI应用程序，接着再根据某个特定标志的设置与否来定义 SDI或MDI文档模板。特别地，可使用 AppWizard创建SDI应用程序，并将它的 InitInstance()结合到MDI应用程序的InitInstance()中。

5. 使用光盘时注意

当运行光盘上没有 /d选项的项目时，就会出现一个 MDI应用程序。使用 /d选项，则会出现

一个对话框应用程序。

6. 程序清单 —— CCommandLineInfo类

```
#if !defined WZDCOMMANDLINEINFO_H
#define WZDCOMMANDLINEINFO_H

// WzdCommandLineInfo.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CWzdCommandLineInfo window

class CWzdCommandLineInfo : public CCommandLineInfo
{
// Construction
public:
    CWzdCommandLineInfo();
// Attributes
public:
    BOOL m_bDialogFg;

// Operations
public:
    void ParseParam( const TCHAR* pszParam,BOOL bFlag,BOOL bLast );

// Overrides

// Implementation
public:
    virtual ~CWzdCommandLineInfo();

};

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

#endif

// WzdCommandLineInfo.cpp : implementation file
//

#include "stdafx.h"
#include "wzd.h"
#include "WzdCommandLineInfo.h"

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

/////////////////////////////////////////////////////////////////////////////
// CWzdCommandLineInfo

CWzdCommandLineInfo::CWzdCommandLineInfo()
```

```
{
    m_bDialogFg = FALSE;
}

CWzdCommandLineInfo::~CWzdCommandLineInfo()
{
}

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

void CWzdCommandLineInfo::ParseParam( const TCHAR* pszParam,BOOL bFlag,
    BOOL bLast )
{
    CString sArg( pszParam );
    if ( bFlag )
    {
        m_bDialogFg = !sArg.CompareNoCase( "d" );
    }

    CCommandLineInfo::ParseParam( pszParam,bFlag,bLast );
}
```

4.5　实例5：在系统托盘中添加图标

1. 目标

在系统托盘(system tray)中添加图标(如图4-3所示，系统托盘是一个出现在桌面右下方的图标集合)。

2. 策略

使用::Shell_NotifyIcon()Window API调用给系统托盘添加一个图标。为了在用户单击图标时能够接收到从该图标返回的消息，需要创建自己的窗口消息，并手工添加自己的消息处理函数。

放在桌面右下角的系统托盘中的用户自己的图标

图4-3　系统托盘中的图标

3. 步骤

1) 设置应用程序

系统托盘图标通过向应用程序窗口发送窗口消息来向应用程序报告返回。在 MDI 或SDI应用程序中，系统托盘图标是一种典型的主窗口。因此，需要给 CMainFrame类添加自己的系统托盘逻辑。在对话框应用程序中，使用自己的 CDialog类。

如下所示在CMainFrame 或CDialog类中的包含文件中定义一个新的窗口消息。当用户单击图标时，这个消息就会发送到该类所在的窗口:

```
// define new window message to indicate user has clicked on icon
//    in system tray
#define WM_SYSTEMTRAY WM_USER+1
```

2) 在系统托盘中创建图标

在CMainFrame 或CDialog派生类中添加以下代码以创建图标。其中的 m_hWnd变量是属于该类的窗口句柄。使用 ID编辑器将ID_SYSTEMTRAY加入ID。如果应用程序显示的图标有

好几个，这里就会用不同的图标来表示。参见下一步了解如何添加 WM_SYSTEMTRAY消息用于当用户单击图标时向应用程序发送该消息:

```
// put icon in system tray
NOTIFYICONDATA nid;
nid.cbSize = sizeof( NOTIFYICONDATA );
nid.hWnd = m_hWnd;                          // handle of window that will receive
                                            //     messages from icon
nid.uID = ID_SYSTEMTRAY;                    // id for this icon
nid.uFlags = NIF_MESSAGE|NIF_ICON|NIF_TIP;
                                            // the next three parameters are valid
nid.uCallbackMessage = WM_SYSTEMTRAY;
                                            // message that icon sends when clicked
nid.hIcon = AfxGetApp()->LoadIcon( IDI_SYSTEMTRAY_ICON );
                                            // icon
strcpy( nid.szTip, "System Tray Tip" );
                                            // bubble help message for icon
::Shell_NotifyIcon( NIM_ADD,&nid );
```

3) 从图标获取消息

在CMainFrame 或CDialog派生类中手工添加一个新的消息处理函数。由于是手工添加，因此必须确保将消息映射宏放在{{}}括号之外，使Class Wizard不能删除它:

```
BEGIN_MESSAGE_MAP( CMainFrame, CMDIFrameWnd )
   // {{AFX_MSG_MAP( CMainFrame )
   :   :   :
   // }}AFX_MSG_MAP
   ON_MESSAGE( WM_SYSTEMTRAY,OnSystemTray )
```

如下所示的代码用于处理来自图标的消息。其中 1Param参数包含实际的鼠标消息内容:

```
// handle system tray message
LRESULT CMainFrame::OnSystemTray( WPARAM wParam,LPARAM lParam)
{
   // wParam = the nid.uID defined above
   //     (useful if you have more then one icon in tray)
   // lParam = mouse message
   if ( wParam = ID_SYSTEMTRAY )
   {
      switch( lParam )
      {
      case WM_LBUTTONDOWN:
         break;

      case WM_RBUTTONDOWN:
         break;

      case WM_LBUTTONDBLCLK:
         break;

      }
   }
   return 1;
}
```

4) 从系统托盘中删除图标

在应用程序结束之前，确保使用以下代码从系统托盘中删除自己的图标。否则该图标将一直存在，直到用户重新启动计算机时才消失。删除图标的理想位置是在创建图标的窗口的WM_CLOSE消息处理函数中，代码如下所示:

```
// delete icon from system tray
NOTIFYICONDATA nid;
nid.cbSize = sizeof( NOTIFYICONDATA );
nid.hWnd = m_hWnd;
nid.uID = ID_SYSTEMTRAY;
nid.uFlags = 0;
::Shell_NotifyIcon( NIM_DELETE,&nid );
```

注意　如果是在调试器中终止应用程序，那就没有机会删除系统托盘中的图标。但是，如果拖动鼠标光标到这个“流浪的”图标上，系统将会迅速判定该图标不再需要并将其自动删除。

4. 注意

系统托盘的典型用途是配置一个应用程序并注册自己的图标，然后使应用程序隐藏。为了创建这种应用程序，可以使用 AppWizard来创建一个对话框应用程序。如果乐意的话，还可以将对话框转换为具有标签的属性表 —— 如实例46所示，但不要设置 Wizard的模式。一旦应用程序在系统托盘中注册好一个图标，它将使用 Show_Window(SW_HIDE)来使自己隐藏。如果用户单击系统托盘中的相应图标，就会出现一个弹出式菜单或用 Show Window(SW_HIDE)产生的应用程序对话框。

注意　隐藏应用程序的主窗口将同时使主窗口从任务栏中消失。

5. 使用光盘时注意

当运行随书附带光盘上的工程时，在系统托盘中就会出现一个新的图标。可以在MainFrame.cpp中的OnSystemTray()中设置断点，并观察用鼠标单击图标的动作给应用程序所报告的消息。

4.6　实例6：主菜单状态栏中的标记

1. 目标

如图4-4所示，在主窗口的标题栏中显示一个标记或其他位图。

图4-4　标题栏中的标识符

2. 策略

标题栏通常是在窗口收到 WM_NCPAINT消息时由系统绘制的。然而，标题栏也可以用来反映某个窗口的激活与否 —— 通过窗口背景颜色由灰到亮的变化来判断。因此，必须截获发送给主窗口的3个窗口消息：WM_NCPAINT、WM_ACTIVATE和WM_NCACTIVATE，然后再绘制自己的位图。

3. 步骤

1) 创建并装载位图

创建一个较小的位图——确保背景为灰色，这个位图将成为新的标题。将该位图分两次添加到资源中——一次是作为 IDB_ACTIVE_CAPTION_BITMAP，另一次是作为 IDB_INACTIVE_CAPTION_BITMAP。可以在两个标题中使用同一位图，这是因为稍后将为它们替换灰色背景。

使用位图类CWzdBitmap(在前面书中的实例中已经介绍过)分两次装载该位图。实际上，CWzdBitmap所做的只是用位图的灰色替换装载位图时用户自己所指定的颜色。在CMainFrame类中嵌入两个CWzdBitmap变量：

```
CWzdBitmap m_bitmapActive;
CWzdBitmap m_bitmapInactive;
```

现在分两次装载在第一步创建的位图，告诉 CWzdBitmap首先用灰色替换当前系统标题栏的颜色，先对激活窗口，然后是非激活窗口：

```
m_bitmapActive.LoadBitmapEx( IDB_ACTIVE_CAPTION_BITMAP,
   ::GetSysColor( COLOR_ACTIVECAPTION ) );
m_bitmapInactive.LoadBitmapEx( IDB_INACTIVE_CAPTION_BITMAP,
   ::GetSysColor( COLOR_INACTIVECAPTION ) );
```

2) 截获在非客户区的绘制消息

使用Class Wizard给CMainFrame类添加三个消息处理函数： WM_NCPAINT、WM_ACITIVETA和WM_NCATIVATE。由于该类归属于应用程序的主窗口，因此它将接收导致主窗口被绘制的消息。

如下所示，填充这些消息处理函数。注意，必须给 CMainFrame类添加一个m_bActive变量，以跟踪主窗口是否被激活。同时也要注意用下面所给的方法调用 DrawTitle()辅助函数来进行绘制:

```
// WM_NCPAINT message handler
void CMainFrame::OnNcPaint()
{
   CMDIFrameWnd::OnNcPaint();

   // draw title
   DrawTitle();
}

// WM_ACTIVE message handler
void CMainFrame::OnActivate( UINT nState, CWnd* pWndOther,
   BOOL bMinimized )
{
   CMDIFrameWnd::OnActivate( nState, pWndOther, bMinimized );

   // set state and draw title
   BOOL m_bActive;
   m_bActive = ( nState! = WA_INACTIVE );
   DrawTitle();
}

// WM_NCACTIVATE message handler
BOOL CMainFrame::OnNcActivate( BOOL bActive )
```

```
{
    BOOL b = CMDIFrameWnd::OnNcActivate( bActive );

    // set state and draw title
    m_bActive = bActive;
    DrawTitle();

    return b;
}
```

3) 绘制标题栏

给CMainFrame类添加一个 DrawTitle()函数，该函数的起始处应确定是否有标题需要绘制:

```
void CMainFrame::DrawTitle()
{
    // if window isn't visible or is minimized, skip
    if ( !IsWindowVisible() || IsIconic())
        return;
```

根据窗口是否激活，将合适的位图对象选入内存设备环境：

```
    CDC memDC;
    CDC* pDC = GetWindowDC();
    memDC.CreateCompatibleDC( pDC );
    memDC.SelectObject( m_bActive ? &m_bitmapActive:&m_bitmapInactive );
```

计算绘制位图的位置。既要避免在最左方的图标上绘制，又要避免在最右方的窗口按钮(关闭，最小化等按钮)上绘制。另外还要避免在窗口的边界上绘制:

```
    CRect rect, rectWnd;
    GetWindowRect( &rect );
    rect.top += GetSystemMetrics( SM_CYFRAME )+1;
        // for small caption use SM_CYDLGFRAME
    rect.bottom = rect.top + GetSystemMetrics( SM_CYSIZE )-4;
        // for small caption use SM_CYSMSIZE
    rect.left += GetSystemMetrics( SM_CXFRAME ) +
        // for small caption use SM_CXDLGFRAME
        GetSystemMetrics( SM_CXSIZE );
            // for small caption use SM_CXSMSIZE
    rect.right -= GetSystemMetrics( SM_CXFRAME ) -
        // for small caption use SM_CXDLGFRAME
        ( 3 *
            // set to number of buttons already in caption + 1
        GetSystemMetrics( SM_CXSIZE ) )-1;
          // for small caption use SM_CXSMSIZE
    GetWindowRect( rectWnd );
    rect.OffsetRect( -rectWnd.left, -rectWnd.top );
```

绘制位图然后再清除:

```
    pDC->BitBlt( rect.left, rect.top, rect.Width(), rect.Height(),
        &memDC, 0, 0, SRCCOPY );

    memDC.DeleteDC();
    ReleaseDC( pDC );
}
```

4. 注意

在该位图填充标题栏以前，旧标题仍然将被绘制，这样在填充时会引起一些闪烁。为减少这种闪烁可以通过在 CMainFrame的PreCreate Window()函数中添加如下代码来防止文档标题包含在旧标题中:

```
BOOL CMainFrame::PreCreateWindow( CREATESTRUCT& cs )
{
   cs.style &= ~ FWS_ADDTOTITLE;
   return CMDIFrameWnd::PreCreateWindow( cs );
}
```

也可以完全删除旧标题，但是这个标题仍然在应用程序任务栏中出现。

5. 使用光盘时注意

当运行随书附带光盘上的工程时，会注意到在应用程序的标题栏中通常的文字已被位图所代替。