MICROSOFT

2008

MFC程序员的WTL指南

开发文档

wzj

[键入公司地址]

[序言 3](#_Toc207269932)

[第一章ATL 界面类 4](#_Toc207269933)

[1.1、对本书的总体介绍 4](#_Toc207269934)

[1.2、对这一章的简单介绍 5](#_Toc207269935)

[1.2.1、ATL 背景知识 ATL 和 WTL 的发展历史 5](#_Toc207269936)

[1.2.2、ATL风格模板 6](#_Toc207269937)

[1.3、ATL 窗口类 8](#_Toc207269938)

[1.3.1、定义一个窗口的实现 8](#_Toc207269939)

[1.3.2、填写消息映射链 9](#_Toc207269940)

[1.3.3、高级消息映射链和嵌入类 11](#_Toc207269941)

[1.3.4、ATL程序的结构 13](#_Toc207269942)

[1.3.5、ATL中的对话框 15](#_Toc207269943)

[第二章WTL 界面基类 18](#_Toc207269944)

[2.1、WTL 总体印象 18](#_Toc207269945)

[2.2、开始写WTL程序 18](#_Toc207269946)

[2.2.1、WTL 对消息映射的增强 20](#_Toc207269947)

[2.2.2、从WTL的应用程序生成向导能得到什么 24](#_Toc207269948)

[第三章 工具条与状态条 35](#_Toc207269949)

[3.1、主窗口的工具条和状态条 35](#_Toc207269950)

[第四章 对话框与控件 48](#_Toc207269951)

[第五章 高级对话框用户界面类 67](#_Toc207269952)

[第六章 包容ActiveX控件 82](#_Toc207269953)

[第七章 分隔窗口 96](#_Toc207269954)

[第八章 属性页与向导 112](#_Toc207269955)

[第九章 GDI类，通用对话框，初始化类 127](#_Toc207269956)

[**9.1、GDI 封装类** 127](#_Toc207269957)

[**9.1.1、封装类的通用函数** 128](#_Toc207269958)

[**9.2.3、与 MFC 封装类的不同之处** 130](#_Toc207269959)

[**9.3、资源装载（Resource-Loading）函数** 130](#_Toc207269960)

[**9.4、使用通用对话框** 133](#_Toc207269961)

[**9.4.1、CFileDialog类** 134](#_Toc207269962)

[**9.4.2、CFolderDialog类** 137](#_Toc207269963)

[**9.5、其它有用的类和全局函数** 138](#_Toc207269964)

[**9.5.1、对结构的封装** 138](#_Toc207269965)

[**9.5.2、处理双类型参数的类** 138](#_Toc207269966)

[**9.6、其它工具类** 139](#_Toc207269967)

[**9.7、全局函数** 141](#_Toc207269968)

[**9.8、宏** 142](#_Toc207269969)

[**9.9、例子工程** 143](#_Toc207269970)

[第十章 支持拖放操作 147](#_Toc207269971)

# 序言

我一直在寻找这样一个类库：他对Windows的窗口提供面向对象的封装，有灵活的消息响应机制和比较完备的界面框架解决方案，对标准控件提供简练实用的封装，支持操作系统的新特性，支持功能扩充和二次开发，有代码自动生成向导机制，生成的程序使用较少的系统资源，最后是有完全的代码支持和文档支持。

　　你会说那就用MFC吧！

　　是的，我一直使用MFC，但我对MFC已经越来越厌倦了。陈旧的类库使得它无法支持操作系统的新特性(MFC的类库从4.21版之后就没有更新了，而那时是1998年，人们使用Windows 95和windows NT4)，臃肿的消息映射机制和为了兼容性而保留下来的代码使得程序效率低下，面面俱到的框架结构使得生成的应用程序庞大并占用过多的系统资源。当一个功能简单的程序使用动态链接也超过200K，占用3%-4%的系统资源时，我决定放弃MFC，寻找一个新的功能类似的类库。我研究过很多类似的代码，不是过于简单，无法用于应用程序的开发,就是缺乏代码和文档的支持。在CodeProject上有一个名为Class的类库，我也研究过它的代码，具备了基本的界面框架，对控件也有了简单的封装，但是不实用，庞大的虚函数机制使得对象非常臃肿，无法减少对资源的占用。我甚至仿照MFC做了一个简单的类库 miniGUI，形成了基本的框架解决方案，但是最后放弃了，原因很简单：无法用于应用程序的开发。一个应用程序界面框架错综复杂，要考虑的事情太多，开发者不可能在应用程序和界面框架两线作战。就在我即将绝望的时候，我遇到了WTL。

　　由于工作的需要经常开发一些COM组件，在要求不能使用MFC的场合就是用ATL。ATL提供了对窗口的面向对象地封装和简单的消息映射机制，但是ATL过于简单，用它开发应用程序几乎不可能。要想让 ATL具备界面框架解决方案的功能还需要做很多事情，幸运的是WTL就做了这些事情。WTL是个很奇特的东西，它由微软公司一群热情的程序员维护，它从未出现在微软的官方产品名单上，但可以从微软的官方网站下载最新的WTL。它没有正式的文档支持，用WTL做关键字在MSDN中检索只能得到0个结果，但是全世界的开发网站上都有针对WTL的讨论组和邮件列表，任何问题都会得到热情的解答。我认真地对比了MFC和WTL，发现二者有很多相通之处，MFC的功能几乎都能在WTL中实现，只是方法不同而已。我几乎不费吹灰之力就将以前写的一个MFC程序用WTL改写了，使用静态链接的WTL程序比使用动态链接的 MFC程序还要小，资源占用只有MFC程序的一半。

　　但是一时的热情不能解决文档缺乏的困扰，虽然网上有很多使用WTL的例子和说明文章，几乎把MFC能实现的各种稀奇古怪的效果都实现了。就在这个时候我看到了迈克尔.敦 (Michael Dunn)的“WTL for MFC Programmers”系列文章，我的感觉和1995年我第一次见到MSDN时一样，几乎是迫不及待地将其读完，同时也萌发了将其翻译成汉语的冲动。于是给Michael写了封邮件，希望能够得到授权将他的文章翻译成汉语(事实上在这之前我已经翻译了两章了)。在得到授权确认后才发现这个工作是多么的困难，但为时已晚，只能硬着头皮撑下去。

# 第一章ATL 界面类

　　你需要开发平台SDK。你要使用WTL不能没有它，你可以使用在线升级安装开发平台SDK，也可以下载全部文件后在本地安装。在使用之前要将SDK的包含文件（.h头 文件）和库文件（.Lib文件）路径添加到VC的搜索目录，SDK有现成的工具完成这个工作，这个工具位于开发平台SDK程序组的“Visual Studio Registration”文件夹里（这是针对VC++6.0的）。

　　你需要安装 WTL。你可以从微软的网站上下载WTL的8.0版，在安装之前可以先查看“Intro duction to WTL - Part 1”和“Easy installation of WTL”这两篇文章，了解一下所要安装的文件的信息，虽然现在这些文章有些过时，但还是可以提供很多有用的信息。有一件我认为不该在本篇文章中提到的事是告诉VC如何搜索WTL的包含文件路径，如果你用的VC6，用鼠标点击 Tools\Options，转到Directories标签页，在显示路径的列表框中选择Include Files，然后将WTL的包含文件的存放路径添加到包含文件搜索路径列表中。

　　你需要了解MFC。很好地了解MFC将有助于你理解后面提到的有关消息映射的宏并能够编辑那些标有“不要编辑（DO NOT EDIT）”的代码而不会出现问题。

　 　你需要清楚地知道如何使用Win32 API编程。如果你是直接从MFC开始学习Windows编程，没有学过API级别的消息处理方式，那很不幸你会在使用WTL时遇到麻烦。如果不了解 Windows消息中WPARAM参数和LPARAM参数的意义，则需要读一些这方面的文章（在CodeProject有大量的此类文章）。

　　你需要知道 C++ 模板的语法，你可以到VC Forum FAQ 相关的连接寻求答案。

我只是讨论了一些涵盖VC 6的特点，不过据我了解所有的程序都可以在VC 7上使用。由于我不使用VC 7，我无法对那些在VC 7中出现的问题提供帮助，不过你还是可以放心的在此张贴你的问题，因为其他的人可能会帮助你。

## 1.1、对本书的总体介绍

　 　WTL具有两面性，确实是这样的。虽然它没有MFC的界面类库那样强大的功能，但是能够生成很小的应用程序。如果你象我一样使用MFC进行界面编程，你会觉得MFC提供的界面控件封装类使用起来非常舒服，更不用说MFC内置的消息处理机制。当然，如果你也象我一样不希望自己的程序仅仅因为使用了MFC的框架就增加几百K的大小的话，WTL就是你的选择。当然，我们还要克服一些障碍：

1. ATL样式的模板类初看起来有点怪异 ；
2. 没有类向导的支持，所以要手工处理所有的消息映射（名叫VisualFC的第三方插件可以提供相应向导）；
3. MSDN没有正式的文档支持，你需要到处去收集有关的文档，甚至是查看WTL的源代码；
4. 买不到参考书籍；
5. 没有微软的官方支持；
6. ATL/WTL的窗口与MFC的窗口有很大的不同，你所了解的有关MFC的知识并不完全适用于WTL。

　　 从另一方面来讲，WTL也有它自身的优势：

1. 不需要学习或掌握复杂的文档/视图框架；
2. 具有MFC的基本的界面特色，比如DDX/DDV和命令状态UI更新功能（比如菜单的Check标记和Enable标记等）；
3. 增强了一些MFC的特性（比如更加易用的分隔窗口）；
4. 可生成比静态链接的MFC程序更小的应用程序（WTL的所有源代码都是静态链接到你的程序中的，除了CRT之类）；
5. 你可以修正自己使用的WTL中的BUG，而不会影响其他的应用程序(相比之下，如果你修正了有BUG的MFC/CRT动态库就可能会引起其它应用程序的崩溃；
6. 如果你仍然需要使用MFC，MFC的窗口和ATL/WTL的窗口可以“和平共处”。

　　在本文中，我将首先介绍ATL的窗口类，毕竟WTL是构建于ATL之上的，并附加了一系列类，所以需要很好地了解ATL的窗口类。介绍完ATL之后我将介绍WTL的特性，并展示它是如何使界面编程变得轻而易举。

## 1.2、对这一章的简单介绍

　　WTL是个很酷的工具，在理解这一点之前需要首先介绍ATL。WTL是构建与ATL之上的，并附加了一系列类。如果你是个使用MFC的程序员，你可能没有机会接触到ATL的界面类，所以请容忍我在开始WTL之前先罗索一些别的东西，绕道来介绍一下ATL是很有必要地。

　　在本文的第一部分，我将给出一点ATL的背景知识，包括一些编写ATL代码所必须知道的基础知识，快速地解释一些令人不知所措的ATL模板类和基本的ATL窗口类。

### 1.2.1、ATL 背景知识 ATL 和 WTL 的发展历史

ATL “活动模板库”是一个很古怪的名字。那些年纪大的人可能还记得，它最初被称为“网络组件模板库”，这可能是它更准确的称呼，因为ATL的目的就是使编写组件对象和ActiveX控件更容易一些（ATL是在微软开发新产品ActiveX-某某的过程中开发的，那些ActiveX-某某现在被称为某某. NET）。由于ATL只是为了便于编写组件对象而存在的，所以仅提供了简单的界面类，相当于MFC的窗口类（CWnd）和对话框类（CDialog）。 幸运地是，这些类非常的灵活，能够在其基础上构建象WTL这样的附加类。

WTL现在已经是第三次修正版了，最初的版本是3.1，现在的最新版本是8.0（WTL的版本号之所以这样选择，是为了与ATL的版本匹配，所以不存在1和2这 样的版本号）。WTL 3.1可以与VC 6和VC 9一起使用，但是在VC 9下需要定义几个预处理标号。WTL 8向下兼容WTL 3.1，并且不作任何修改就可以与VC 9一起使用，现在看来没有任何理由还使用3.1来进行新的开发工作。

### 1.2.2、ATL风格模板

　　即使你能够毫不费力地阅读C++的模板类代码，仍然有两件事可能会使你有些头晕，以下面这个类的定义为例：

|  |
| --- |
| class　CMyWnd : public CWindowImpl<CMyWnd>  {  ...  }; |

　　这样作是合法的，因为C++的语法解释说即使CMyWnd类只是被部分定义，类名CMyWnd已经被列入递归继承列表，是可以使用的。将类名作为模板类的参数是因为ATL要做另一件诡秘的事情，那就是编译期间的虚函数调用机制。

如果你想要了解它是如何工作地，请看下面的例子：

|  |
| --- |
| template <class T>  class B1  {  public:  void SayHi()  {  T\* pT = static\_cast<T\*>(this);　 // HUH?? 我将在下面解释  pT->PrintClassName();  }  protected:  void PrintClassName() { cout << "This is B1"; }  };  class D1 : public B1<D1>  {  // No overridden functions at all  };  class D2 : public B1<D2>  {  protected:  　　void PrintClassName() { cout << "This is D2"; }  };  main()  {  D1 d1;  D2 d2;  d1.SayHi();　　// 打印"This is B1"  d2.SayHi();　　// 打印"This is D2"  } |

　 　这句代码static\_cast<T\*>(this) 就是窍门所在。它根据函数调用时的特殊处理将指向B1类型的指针this指派为D1或D2类型的指针，因为模板代码是在编译其间生成的，所以只要编译器生成正确的继承列表，这样指派就是安全的。如果你写成：

|  |
| --- |
| class D3 : public B1<D2> |

　　就会有麻烦。之所以安全是因为this对象只可能是指向D1或D2（在某些情况下）类型的对象，不会是其他的东西。注意这很像C++的多态性，只是SayHi()方法不是虚函数。

　　要解释这是如何工作的，首先看对每个SayHi()函数的调用，在第一个函数调用，对象B1被指派为D1，所以代码被解释成：

|  |
| --- |
| void B1<D1>::SayHi()  {  D1\* pT = static\_cast<D1\*>(this);  pT->PrintClassName();  } |

　　由于D1没有重载PrintClassName()，所以查看基类B1，B1有PrintClassName()，所以B1的Print ClassName()被调用。

　　现在看第二个函数调用SayHi()，这一次对象被指派为D2类型，SayHi()被解释成：

|  |
| --- |
| void B1<D2>::SayHi()  {  D2\* pT = static\_cast<D2\*>(this);  pT->PrintClassName(); } |

　　这一次，D2含有PrintClassName()方法，所以D2的PrintClassName()方法被调用。

　　这种技术的有利之处在于：

1. 不需要使用指向对象的指针；
2. 节省内存，因为不需要虚函数表；
3. 因为没有虚函数表所以不会发生在运行时调用空指针指向的虚函数；
4. 所有的函数调用在编译时确定（区别于C++的虚函数机制使用的动态链接），有利于编译程序对代码的优化。

　　节省虚函数表在这个例子中看起来无足轻重（每个虚函数只有4个字节），但是设想一下如果有15个基类，每个类含有20个方法，加起来就相当可观了。

## 1.3、ATL 窗口类

　　好了，关于ATL的背景知识已经讲的够多了，该正式讲ATL的了。ATL在设计时，接口定义和实现是严格区分开的，这在窗口类的设计中最为明显。这一点类似于COM，COM的接口定义和实现是完全分开的（或者可能有多个实现）。

ATL有一个专门为窗口设计的类，可以做全部的窗口操作，这就是CWindow。它实际上就是对HWND操作的包装类，对几乎所有以HWND句柄为第一个参数的窗口API的进行了封装，例如：SetWindowText() 和 DestroyWindow()。CWindow类有一个公有成员m\_hWnd，使你可以直接对窗口的句柄操作，CWindow还有一个操作符HWND， 你可以将CWindow对象传递给以HWND为参数的函数，但这与CWnd::GetSafeHwnd()（译者加：MFC的方法）没有任何相同之处。

　 　CWindow 与 MFC 的CWnd类有很大的不同，创建一个CWindow对象占用很少的资源，因为只有一个数据成员，没有MFC窗口中的对象链。MFC内部维持这个对象链， 此对象链将HWND映射到CWnd对象。还有一点与MFC的CWnd类不同的是：当一个CWindow对象超出了作用域，它所关联的窗口并不被销毁掉，这意味 着你并不需要随时记得分离你所创建的临时CWindow对象。

　　在ATL类中对窗口过程的实现是CWindowImpl。 CWindowImpl 包含有所有窗口实现代码，例如：窗口类的注册，窗口的子类化，消息映射以及基本的WindowProc()函数。可以看出这与MFC的设计有很大的不同， MFC将所有的代码都放在CWnd类中。

　　另外两个独立的类是对话框的实现，它们分别是CDialogImpl 和 CAxDialogImpl，CDialogImpl 用于实现普通的对话框，而CAxDialogImpl实现含有ActiveX控件的对话框。

### 1.3.1、定义一个窗口的实现

　　任何非对话框窗口都是从CWindowImpl 派生的，你的新类需要实现三样东西：

1. 定义窗口类（通过**DECLARE\_WND\_CLASS**或**DECLARE\_WND\_CLASS\_EX**宏实现）；
2. 生成消息映射链（通过**BEGIN\_MSG \_MAP**和**END\_MSG\_MAP**宏实现）；
3. 窗口使用的默认窗口类型（称为window traits）。

　 　窗口类的定义通过**DECLARE\_WND\_CLASS**宏或**DECLARE\_WND\_CLASS\_EX**宏来实现。这两个宏定义了一个 CWndClassInfo结构，这个结构封装了WNDCLASSEX结构。DECLARE\_WND\_ CLASS宏让你指定窗口类的类名，其他参数使用默认设置，而DECLARE\_WND\_CLASS\_EX宏还允许你指定**窗口类的类型和窗口的背景颜色**。你也可以用NULL作为类名，ATL会自动为你生成一个类名。

让我们开始定义一个新类，在后面的章节我们会逐步完成这个类的定义。

|  |
| --- |
| class CMyWindow : public CWindowImpl<CMyWindow>  {  public:  DECLARE\_WND\_CLASS(\_T("My Window Class"))  }; |

　 　接下来是生成消息映射链，ATL的消息映射链比MFC简单得多，ATL的消息映射链被展开为switch语句，switch语句正确的消息处理者并调用相 应的函数。使用消息映射链的宏是BEGIN\_MSG \_MAP 和 END\_MSG\_MAP，让我们为我们的窗口添加一个空的消息映射链。

|  |
| --- |
| class CMyWindow : public CWindowImpl<CMyWindow>  {  public:  DECLARE\_WND\_CLASS(\_T("My Window Class"))  BEGIN\_MSG\_MAP(CMyWindow)  END\_MSG\_MAP()  }; |

　 　我将在下一节展开讲如何如何添加消息处理到消息映射链。最后，我们需要为我们的窗口类定义窗口的特征，窗口的特征就是窗口类型和扩展窗口类型的联合体， 用于创建窗口时指定窗口的类型。窗口类型被指定为参数模板，所以窗口的调用者不需要为指定窗口的正确类型而烦心，下面是是同ATL类CWinTraits 定义窗口类型的例子：

|  |
| --- |
| Typedef CWinTraits<WS\_OVERLAPPEDWINDOW|WS\_CLIPCHILDREN,WS\_EX\_APPWINDOW> CMyWindowTraits;  class CMyWindow : public CWindowImpl<CMyWindow, CWindow, CMyWindowTraits>  {  public:  DECLARE\_WND\_CLASS(\_T("My Window Class"))  BEGIN\_MSG\_MAP(CMyWindow)  END\_MSG\_MAP()  }; |

　　调用者可以重载CMyWindowTraits的类型定义，但是一般情况下这是没有必要的，ATL提供了几个预先定义的特殊的类型，其中之一就是CFrameWinTraits，一个非常棒的框架窗口：

|  |
| --- |
| typedef CWinTraits<WS\_OVERLAPPEDWINDOW | WS\_CLIPCHILDREN | WS\_CLIPSIBLINGS,  WS\_EX\_APPWINDOW | WS\_EX\_WINDOWEDGE>　CFrameWinTraits; |

### 1.3.2、填写消息映射链

　 　ATL的消息映射链是对开发者不太友好的部分，也是WTL对其改进最大的部分。类向导至少可以让你添加消息响应，然而ATL没有消息相关的宏和象MFC那样的参数自动展开功能，在ATL中只有三种类型的消息处理，一个是WM\_NOTIFY，一个是WM\_COMMAND，第三类是其他窗口消息宏MESSAGE\_HANDLER。让我们开始为我们的窗口添加WM\_CLOSE 和 WM\_DESTROY的消息处理函数。

|  |
| --- |
| class CMyWindow : public CWindowImpl<CMyWindow, CWindow, CFrameWinTraits>  {  public:  DECLARE\_WND\_CLASS(\_T("My Window Class"))  BEGIN\_MSG\_MAP(CMyWindow)  　　 MESSAGE\_HANDLER(WM\_CLOSE, OnClose)  　　 MESSAGE\_HANDLER(WM\_DESTROY, OnDestroy)  END\_MSG\_MAP()  LRESULT OnClose(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)  {  　　 DestroyWindow();  　　 return 0;  }  LRESULT OnDestroy(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)  {  　　 PostQuitMessage(0);  　　 return 0;  }  }; |

你可能注意到消息响应函数得到的是原始的WPARAM 和 LPARAM值，你需要自己将其展开为相应的消息所需要的参数。还有第四个参数bHandled，这个参数在消息相应函数调用前被ATL设置为TRUE，如果在你的消息响应处理完之后需要ATL调用默认的WindowProc()处理该消息，你可以将bHandled设置为FALSE。这与MFC不同， MFC是显式地调用基类的响应函数来实现的默认的消息处理的。

让我们再添加一个对WM\_COMMAND消息的处理，假设我们的窗口有一个ID为IDC\_ABOUT的About菜单：

|  |
| --- |
| class CMyWindow : public CWindowImpl<CMyWindow, CWindow, CFrameWinTraits>  {  public:  DECLARE\_WND\_CLASS(\_T("My Window Class"))  BEGIN\_MSG\_MAP(CMyWindow)  　　 MESSAGE\_HANDLER(WM\_CLOSE, OnClose)  　　 MESSAGE\_HANDLER(WM\_DESTROY, OnDestroy)  　　 COMMAND\_ID\_HANDLER(IDC\_ABOUT, OnAbout)  END\_MSG\_MAP()  LRESULT OnClose(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)  {  　　 DestroyWindow();  　　 return 0;  }  LRESULT OnDestroy(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)  {  　　 PostQuitMessage(0);  　　 return 0;  }  LRESULT OnAbout(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled)  {  　　 MessageBox ( \_T("Sample ATL window"), \_T("About MyWindow") );  　　 return 0;  }  }; |

　　需要注意的是COMMAND\_HANDLER宏已经将WM\_COMMAND消息的参数展开了，同样，NOTIFY\_ HANDLER宏也将WM\_NOTIFY消息的参数展开了。

### 1.3.3、高级消息映射链和嵌入类

　　ATL的另一个显著不同之处就是任何C++类都可以响应消息，而MFC只是将消息响应任务分给了CWnd类和CCmdTarget类，外加几个有**PreTranslateMessage()**方法的类。ATL的这种特性允许我们编写所谓的“嵌入类”，为我们的窗口添加特性只需将该类添加到继承列表中就行了，就这么简单！

一个基本的带有消息映射链的类通常是模板类，将派生类的类名作为模板的参数，这样它就可以访问派生类中的成员，比如m\_hWnd（CWindow类中的HWND成员）。让我们来看一个嵌入类的例子，这个嵌入类通过响应WM\_ERASEBKGND消息来画窗口的背景。

|  |
| --- |
| template <class T, COLORREF t\_crBrushColor>  class CPaintBkgnd : public CMessageMap  {  public:  CPaintBkgnd() { m\_hbrBkgnd = CreateSolidBrush(t\_crBrushColor); }  ~CPaintBkgnd() { DeleteObject ( m\_hbrBkgnd ); }  BEGIN\_MSG\_MAP(CPaintBkgnd)  　　 MESSAGE\_HANDLER(WM\_ERASEBKGND, OnEraseBkgnd)  END\_MSG\_MAP()  LRESULT OnEraseBkgnd(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)  {  　　 T\*　 pT = static\_cast<T\*>(this);  　　 HDC　dc = (HDC) wParam;  　　 RECT rcClient;  　　 pT->GetClientRect ( &rcClient );  　　 FillRect (dc, &rcClient, m\_hbrBkgnd );  　　 return 1;　　// we painted the background  }  protected:  HBRUSH m\_hbrBkgnd;  }; |

　 　让我们来研究一下这个新类。首先，CPaintBkgnd有两个模板参数：使用CPaintBkgnd的派生类的名字和用来画窗口背景的颜色（t\_前缀通常用来作为模板类的模板参数的前缀）。CPaintBkgnd也是从CMessageMap派生的，这并不是必须的，因为所有需要响应消息的类只需使用BEGIN\_MSG\_MAP宏就足够了，所以你可能看到其他的一些嵌入类的例子代码，它们并不是从该基类派生的。

构造函数和析构函数都相当简单，只是创建和销毁Windows画刷，这个画刷由参数t\_crBrushColor决定颜色。接着是消息映射链，它响应WM\_ERASEBKGND消息，最后由响应函数OnEraseBkgnd()使用构造函数创建的画刷填充窗口的背景。在OnEraseBkgnd()中有两件事需要注意：一是它使用了一个派生的窗口类的方法（即GetClientRect()），我们如何知道派生类中有GetClientRect()方法呢？如果派生类中没有这个方法我们的代码也不会有任何问题，由编译器确认派生类T是从CWindow派生的。另一个是OnEraseBkgnd()没有将消息参数wParam展开为设备上下文（DC）。

想要使用这个嵌入类有两件事需要做：

首先，将它加入到继承列表：

|  |
| --- |
| class CMyWindow : public CWindowImpl<CMyWindow, CWindow, CFrameWinTraits>,  　　　　　　　　public CPaintBkgnd<CMyWindow, RGB(0,0,255)> |

　　其次，需要CMyWindow将消息链入CPaintBkgnd类，在CMyWindow的消息映射链中加入CHAIN \_MSG\_MAP宏：

|  |
| --- |
| class CMyWindow : public CWindowImpl<CMyWindow, CWindow, CFrameWinTraits>,  　　　　　　　　public CPaintBkgnd<CMyWindow, RGB(0,0,255)>  {  ...  typedef CPaintBkgnd<CMyWindow, RGB(0,0,255)> CPaintBkgndBase;  BEGIN\_MSG\_MAP(CMyWindow)  　　 MESSAGE\_HANDLER(WM\_CLOSE, OnClose)  　　 MESSAGE\_HANDLER(WM\_DESTROY, OnDestroy)  　　 COMMAND\_HANDLER(IDC\_ABOUT, OnAbout)  　　 CHAIN\_MSG\_MAP(CPaintBkgndBase)  END\_MSG\_MAP()  ...  }; |

任何CMyWindow没有处理的消息都被传递给CPaintBkgnd。应该注意的是WM\_CLOSE，WM\_ DESTROY和IDC\_ABOUT消息将不会传递，因为这些消息一旦被处理消息映射链的查找就会中止。使用typedef是必要地，因为宏是预处理宏，只能有一个参数，如果我们将 CPaintBkgnd<CMyWindow, RGB(0,0,255)>作为参数传递，那个“,”会使预处理器认为我们使用了多个参数。

你可以在继承列表中使用多个嵌入类，每一个嵌入类使用一个CHAIN\_MSG\_MAP宏，这样消息映射链就会将消息传递给它。这与MFC不同，MFC的CWnd派生类只能有一个基类，MFC自动将消息传递给基类。

### 1.3.4、ATL程序的结构

到目前为止我们已经有了一个完整的主窗口类（即使不完全有用），让我们看看如何在程序中使用它。一个ATL程序包含一个CComModule类型的全局变量\_Module，这和MFC的程序都有一个CWin App类型的全局变量theApp有些类似，唯一不同的是在ATL中这个变量必须命名为 \_Module。

　　下面是stdafx.h文件的开始部分：

|  |
| --- |
| // stdafx.h:  #define STRICT  #define VC\_EXTRALEAN  #include <atlbase.h>　　　　 // 基本的ATL类  extern CComModule \_Module;　// 全局\_Module  #include <atlwin.h>　　　　 // ATL窗口类 |

　　atlbase.h是包含最基本的Window编程的头文件，所以我们不需要再包含windows.h，tchar.h之类的头文件。在CPP文件中声明了\_Module变量：

|  |
| --- |
| // main.cpp:  CComModule \_Module; |

　　CComModule含有程序的初始化和关闭函数，需要在WinMain()中显示的调用。让我们从这里开始：

|  |
| --- |
| // main.cpp:  CComModule \_Module;  int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hInstPrev, LPSTR szCmdLine, int nCmdShow)  {  \_Module.Init(NULL, hInst);  \_Module.Term();  } |

　 　Init()的第一个参数只有COM的服务程序才有用，由于我们的EXE不含有COM对象，我们只需将NULL传递给Init()就行了。ATL不提供自己的WinMain()和类似MFC的消息队列，所以我们需要创建CMyWindow对象并添加消息队列才能使我们的程序运行。

|  |
| --- |
| // main.cpp:  #include "MyWindow.h"  CComModule \_Module;  int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hInstPrev,LPSTR szCmdLine, int nCmdShow)  {  　　 \_Module.Init(NULL, hInst);  　　 CMyWindow wndMain;  　　 MSG msg;  　　 // 创建和显示主窗口  　　 if ( NULL == wndMain.Create ( NULL, CWindow::rcDefault, \_T("My First ATL Window") ))  　　 {  　　　　 // 窗口失败  　　　　 return 1;  　　 }  　　 wndMain.ShowWindow(nCmdShow);  　　 wndMain.UpdateWindow();  　　 // 运行消息循环  　　 while ( GetMessage(&msg, NULL, 0, 0) > 0 )  　　 {  　　　　 TranslateMessage(&msg);  　　　　 DispatchMessage(&msg);  　　 }  　　 \_Module.Term();  　　 return msg.wParam;  } |

上面的代码唯一需要说明的是：CWindow::rcDefault，它是CWindow中的成员（静态数据成员），数据类型是RECT。与调用 CreateWindow() 时使用CW\_USEDEFAULT指定窗口的宽度和高度一样，ATL使用rcDefault作为窗口的最初大小。

　　在ATL代码内部，ATL使用了一些类似汇编语言的方法将窗口的句柄与相应的窗口对象联系起来。从外部来看，就可以毫无问题的在线程之间传递CWindow对象，而MFC的CWnd却不能这样做。图1就是我们的窗口：



图1 ATL的应用程序

我们得承认这确实没有什么激动人心的地方。我们将添加一个About菜单并显示一个对话框，主要是为它增加一些情趣。

### 1.3.5、ATL中的对话框

　　我们在前面提到过，ATL有两个对话框类，我们的About对话框使用CDialogImpl。生成一个新对话框和生成一个主窗口几乎一样，只有两点不同：

1. 窗口的基类是CDialogImpl，而不是CWindowImpl。
2. 你需要定义名称为IDD的公有成员，用来保存对话框的资源ID。

　　现在开始为About对话框定义一个新类：

|  |
| --- |
| class CAboutDlg : public CDialogImpl<CAboutDlg>  {  public:  enum { IDD = IDD\_ABOUT };  BEGIN\_MSG\_MAP(CAboutDlg)  END\_MSG\_MAP()  }; |

ATL没有在内部实现对“OK”和“Cancel”两个按钮的响应处理，所以我们需要自己添加这些代码，如果用户用鼠标点击标题栏的关闭按钮，WM\_CLOSE的响应函数就会被调用。我们还需要处理WM\_INITDIALOG消息，这样我们就能够在对话框出现时正确的设置键盘焦点，下面是完整的类定义和消息响应函数。

|  |
| --- |
| class CAboutDlg : public CDialogImpl<CAboutDlg>  {  public:  enum { IDD = IDD\_ABOUT };  BEGIN\_MSG\_MAP(CAboutDlg)  　　 MESSAGE\_HANDLER(WM\_INITDIALOG, OnInitDialog)  　　 MESSAGE\_HANDLER(WM\_CLOSE, OnClose)  　　 COMMAND\_ID\_HANDLER(IDOK, OnOKCancel)  　　 COMMAND\_ID\_HANDLER(IDCANCEL, OnOKCancel)  END\_MSG\_MAP()  LRESULT OnInitDialog(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)  {  　　 CenterWindow();  　　 return TRUE;　　// let the system set the focus  }  LRESULT OnClose(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)  {  　　 EndDialog(IDCANCEL);  　　 return 0;  }  LRESULT OnOKCancel(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled)  {  　　 EndDialog(wID);  　　 return 0;  }  }; |

　　我使用一个消息响应函数同时处理“OK”和“Cancel”两个按钮的WM\_COMMAND消息，因为命令响应函数的wID参数就已经指明了消息是来自“OK”按钮还是来自“Cancel”按钮。

显示对话框的方法与MFC相似，创建一个新对话框类的实例，然后调用DoModal()方法。现在我们回到主窗口，添加一个带有About菜单项的菜单用来显示我们的对话框，这需要再添加两个消息响应函数，一个是响应WM\_CREATE，另一个是响应菜单的IDC\_ABOUT命令。

|  |
| --- |
| class CMyWindow : public CWindowImpl<CMyWindow, CWindow, CFrameWinTraits>,  　　　　　 public CPaintBkgnd<CMyWindow,RGB(0,0,255)>  {  public:  BEGIN\_MSG\_MAP(CMyWindow)  　　 MESSAGE\_HANDLER(WM\_CREATE, OnCreate)  　　 COMMAND\_ID\_HANDLER(IDC\_ABOUT, OnAbout)  　　 // ...  　　 CHAIN\_MSG\_MAP(CPaintBkgndBase)  END\_MSG\_MAP()  LRESULT OnCreate(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)  {  　　 HMENU hmenu = LoadMenu ( \_Module.GetResourceInstance(), MAKEINTRESOURCE(IDR\_MENU1) );  　　 SetMenu(hmenu);  　　 return 0;  }  LRESULT OnAbout(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled)  {  　　 CAboutDlg dlg;  　　 dlg.DoModal();  　　 return 0;  }  // ...  }; |

　　在指定对话框的父窗口的方式上有些不同，MFC是通过构造函数将父窗口的指针传递给对话框,而在ATL中是将父窗口的指针作为DoModal()方法的第一个参数传递给对话框的，如果象上面的代码一样没有指定父窗口，ATL会使用GetActiveWindow()得到的窗口（也就是我们的主框架窗口）作为对话框的父窗口。

　 　对LoadMenu()方法的调用展示了CComModule的另一个方法－GetResourceInstance()，它返回你的EXE的 HINSTANCE实例，和MFC的AfxGetResourceHandle()方法相似。（当然还有CCom Module::GetModuleInstance()，它相当于MFC的AfxGetInstanceHandle()。）

　　图2就是主窗口和对话框的显示效果：

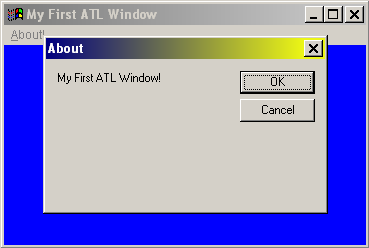


图2 带对话框的应用程序主窗口

# 第二章WTL 界面基类

　　在这一部分我们讲的内容包括生成一个基本的主窗口和WTL提供的一些友好改进，比如UI界面的更新（如菜单上的选择标记）和更好的消息映射机制。为了更好地掌握本章的内容，你应该安装WTL并将WTL库的头文件目录添加到VC的搜索目录中，还要将WTL的应用程序生成向导复制到正确的位置。WTL的发布版本中有文档具体介绍如何做这些设置，如果遇到困难可以查看这些文档。

## 2.1、WTL 总体印象

　　WTL的类大致可以分为几种类型：

1. 主框架窗口的实现- CFrameWindowImpl, CMDIFrameWindowImpl
2. 控件的封装- CButton, CListViewCtrl
3. GDI 对象的封装- CDC, CMenu
4. 一些特殊的界面特性 - CSplitterWindow, CUpdateUI, CDialogResize, CCustomDraw
5. 实用的工具类和宏- CString, CRect, BEGIN\_MSG\_MAP\_EX

　　本篇文章将深入地介绍框架窗口类，还将简要地讲一下有关的界面特性类和工具类，界面特性类和工具类中绝大多数都是独立的类，尽管有一些是嵌入类，例如：CDialogResize。

## 2.2、开始写WTL程序

　　如果你没有用WTL的应用程序生成向导也没关系(我将在后面介绍这个向导的用法), WTL的程序的代码结构很像ATL的程序，本章使用的例子代码有别于第一章的，主要是为了显示WTL的特性，没有什么实用价值。

　　这一节我们将在WTL生成的代码基础上添加代码，生成一个新的程序，程序主窗口的客户区显示当前的时间。stdafx.h的代码如下：

|  |
| --- |
| #define STRICT  #define WIN32\_LEAN\_AND\_MEAN  #define \_WTL\_USE\_CSTRING  #include <atlbase.h>　　　 // 基本的ATL类  #include <atlapp.h>　　　　 // 基本的WTL类  extern CAppModule \_Module; // WTL 派生的CComModule版本  #include <atlwin.h>　　　　 // ATL 窗口类  #include <atlframe.h>　　　 // WTL 主框架窗口类  #include <atlmisc.h>　　　 // WTL 实用工具类，例如：CString  #include <atlcrack.h>　　　 // WTL 增强的消息宏 |

**atlapp.h**是你的工程中第一个包含的头文件，这个文件内定义了有关消息处理的类和CApp Module，CAppModule是从CComModule派生的类。如果你打算使用CString类，你需要手工定义\_WTL \_USE\_ CSTRING标号，因为CString类是在**atlmisc.h**中定义的，而许多包含 在atlmisc.h之前的头文件都会用到CString，定义**\_WTL\_USE\_CSTRING**之后，atlapp.h就会向前声明CString类，其他的头文件就知道CString类的存在，从而避免编译器为此报错。

接下来定义框架窗口。我们的SDI窗口是从CFrameWindowImpl派生的，在定义窗口类时使用**DECLARE\_FRAME\_WND\_CLASS**代替前面使用的DECLARE\_WND\_CLASS。下面时MyWindow.h中窗口定义的开始部分：

|  |
| --- |
| class CMyWindow : public CFrameWindowImpl<CMyWindow>  {  public:  DECLARE\_FRAME\_WND\_CLASS(\_T("First WTL window"), IDR\_MAINFRAME);  BEGIN\_MSG\_MAP(CMyWindow)  　　 CHAIN\_MSG\_MAP(CFrameWindowImpl<CMyWindow>)  END\_MSG\_MAP()  }; |

DECLARE\_FRAME\_WND\_CLASS有两个参数，窗口类名（类名可以是NULL，ATL会替你生成一个类名）和资源ID，创建窗口时WTL 用这个ID装载图标，菜单和快捷键表。我们还要象CFrame WindowImpl中的消息处理（例如WM\_SIZE和WM\_DESTROY消息）那样将消息链入窗口的消息中。

　　现在来看看WinMain()函数，它和第一部分中的例子代码中的WinMain()函数几乎一样，只是创建窗口部分的代码略微不同。

|  |
| --- |
| // main.cpp:  #include "stdafx.h"  #include "MyWindow.h"  **CAppModule \_Module;**  int APIENTRY WinMain(HINSTANCE hInst, HINSTANCE hPrevInst,LPSTR lpCmdLine, int nCmdShow )  {  \_Module.Init ( NULL, hInstance );  CMyWindow wndMain;  MSG msg;  // 创建主窗口  if ( NULL == wndMain.CreateEx())  　　 return 1;　　　 // Window creation failed  // 显示窗口  wndMain.ShowWindow ( nCmdShow );  wndMain.UpdateWindow();  // 标准Win32消息循环  while ( GetMessage ( &msg, NULL, 0, 0 ) > 0 )  {  　　 TranslateMessage ( &msg );  　　 DispatchMessage ( &msg );  }  \_Module.Term();  return msg.wParam;  } |

CFrameWindowImpl中的CreateEx()函数的参数使用了常用的默认值，所以我们不需要特别指定任何参数。正如前面介绍的， CFrameWindowImpl会处理资源的装载，你只需要使用IDR\_MAIN FRAME作为ID定义你的资源就行了（译者注：主要是图标，菜单和快捷键表），你也可以直接使用本章的例子代码。

　　如果你现在就运行程序，你会看到主框架窗口，事实上它没有做任何事情。我们需要手工添加一些消息处理，所以现在是介绍WTL的消息映射宏的最佳时间。

### 2.2.1、WTL 对消息映射的增强

　 　将Win32 API通过消息传递过来的WPARAM和LPARAM数据分解出来是一件麻烦的事情，并且很容易出错，不幸得是ATL并没有为我们提供更多的帮助，我们仍然需要从消息中分解这些数据，当然WM\_COMMAND和WM\_NOTIFY消息除外。但是WTL的出现拯救了这一切！

　　WTL的增强消息映射宏定义在**atlcrack.h**（这个名字来源于“消息解密者”，是一个与windowsx.h的宏所使用的相同术语）中。首先将BEGIN\_MSG\_MAP改为**BEGIN\_MSG\_MAP\_EX**，带\_EX的版本产生“解密”消息的代码。

|  |
| --- |
| class CMyWindow : public CFrameWindowImpl<CMyWindow>  {  public:  BEGIN\_MSG\_MAP\_EX(CMyWindow)  　　 CHAIN\_MSG\_MAP(CFrameWindowImpl<CMyWindow>)  END\_MSG\_MAP()  }; |

　　对于我们的时钟程序，我们需要处理WM\_CREATE消息来设置定时器，WTL的消息处理使用MSG\_作为前缀，后面是消息名称，例如MSG\_WM\_CREATE。这些宏只是代表消息响应处理的名称，现在我们来添加对WM\_CREATE消息的响应函数：

|  |
| --- |
| class CMyWindow : public CFrameWindowImpl<CMyWindow>  {  public:  BEGIN\_MSG\_MAP\_EX(CMyWindow)  　　 MSG\_WM\_CREATE(OnCreate)  　 CHAIN\_MSG\_MAP(CFrameWindowImpl<CMyWindow>)  END\_MSG\_MAP()  // OnCreate(...) ?  }; |

WTL的消息处理函数看起来有点象MFC，每一个处理函数根据消息传递的参数不同也有所不同。由于我们没有向导自动添加消息响应，所以我们需要自己查找正确的消息处理函数（第三插件VisualFC对生成正确的消息处理函数提供支持）。幸运地是，VC可以帮我们的忙，将鼠标光标移到“MSG\_WM\_CREATE”宏的文字上按F12键就可以来到这个宏的定义代码处。如果是第一次使用这个功能，VC会要求从新编译全部文件以建立浏览信息数据库（browse info database），建立了这个数据库之后，VC会打开atlcrack.h并将代码定位到MSG\_WM\_CREATE的定义位置：

|  |
| --- |
| #define MSG\_WM\_CREATE(func) \  if (uMsg == WM\_CREATE) \  { \  　　 SetMsgHandled(TRUE); \  　　 lResult = (LRESULT)**func((LPCREATESTRUCT)lParam)**; \  　　 if(IsMsgHandled()) \  　　　　 return TRUE; \  } |

　　标记为红色的那一行非常重要，就是在这里调用实际的消息响应函数，他告诉我们消息响应函数有一个LPCREATESTRUCT类型的参数，返回值的类型是LRESULT。请注意这里没有ATL的宏所用的 bHandled 参数，SetMsgHandled()函数代替了该参数，我会为此作些简要的介绍。

　　现在为我们的窗口类添加OnCreate()响应函数:

|  |
| --- |
| class CMyWindow : public CFrameWindowImpl<CMyWindow>  {  public:  BEGIN\_MSG\_MAP\_EX(CMyWindow)  　　 MSG\_WM\_CREATE(OnCreate)  **CHAIN\_MSG\_MAP(CFrameWindowImpl<CMyWindow>)**  END\_MSG\_MAP()  LRESULT OnCreate(LPCREATESTRUCT lpcs)  {  　　 SetTimer ( 1, 1000 );  　　 SetMsgHandled(false);  　　 return 0;  }  }; |

　　CFrameWindowImpl 是直接从CWindow类派生的, 所以它继承了CWindow类的所有方法，如Set Timer()。这使得对窗口API的调用有点象MFC的代码，只是MFC使用CWnd类包装这些API。

我们使用SetTimer()函数创建一个定时器，它每隔一秒钟(1000毫秒)触发一次。由于我们需要让CFrameWindowImpl也处理 WM\_CREATE消息，所以我们调用SetMsgHandled(false)，让消息通过**CHAIN\_MSG\_MAP**宏链入基类，这个调用代替了ATL宏使用的bHandled参数。（即使CFrameWindow Impl类不需要处理WM\_CREATE消息，调用**SetMsgHandled (false)**让消息流入基类是个好的习惯，因为这样我们就不必总是记着哪个消息需要基类处理那些消息不需要基类处理，这和VC的类向导产生的代码相似，大多数的派生类的消息处理函数的开始或结尾都会调用基类的消息处理函数）。

为了能够停止定时器我们还需要响应WM\_DESTROY消息，添加消息响应的过程和前面一样，MSG\_WM\_DESTROY宏的定义是这样的：

|  |
| --- |
| #define MSG\_WM\_DESTROY(func) \  if (uMsg == WM\_DESTROY) \  { \  　　 SetMsgHandled(TRUE); \  　　 func(); \  　　 lResult = 0; \  　　 if(IsMsgHandled()) \  　　　 　return TRUE; \  } |

　　OnDestroy()函数没有参数也没有返回值，CFrameWindowImpl也要处理WM\_DESTROY消息，所以还要调用SetMsgHandled(false)：

|  |
| --- |
| class CMyWindow : public CFrameWindowImpl<CMyWindow>  {  public:  BEGIN\_MSG\_MAP\_EX(CMyWindow)  　　MSG\_WM\_CREATE(OnCreate)  　　MSG\_WM\_DESTROY(OnDestroy)  　　CHAIN\_MSG\_MAP(CFrameWindowImpl<CMyWindow>)  END\_MSG\_MAP()  void OnDestroy()  {  　　KillTimer(1);  　　SetMsgHandled(false);  }  }; |

　　接下来是响应WM\_TIMER消息的处理函数，它每秒钟被调用一次。你应该知道怎样使用F12键的窍门了，所以我直接给出响应函数的代码：

|  |
| --- |
| class CMyWindow : public CFrameWindowImpl<CMyWindow>  {  public:  BEGIN\_MSG\_MAP\_EX(CMyWindow)  　　 MSG\_WM\_CREATE(OnCreate)  　　 MSG\_WM\_DESTROY(OnDestroy)  　　 MSG\_WM\_TIMER(OnTimer)  　　 CHAIN\_MSG\_MAP(CFrameWindowImpl<CMyWindow>)  END\_MSG\_MAP()  void OnTimer ( UINT uTimerID, TIMERPROC pTimerProc )  {  　 　if ( 1 != uTimerID )  　　 　　SetMsgHandled(false);  　 　else  　　 RedrawWindow();  }  }; |

　　这个响应函数只是在每次定时器触发时重画窗口的客户区。最后我们要响应WM\_ERASE BKGND消息，在窗口客户区的左上角显示当前的时间。

|  |
| --- |
| class CMyWindow : public CFrameWindowImpl<CMyWindow>  {  public:  　　 BEGIN\_MSG\_MAP\_EX(CMyWindow)  　　　　 MSG\_WM\_CREATE(OnCreate)  　　　　 MSG\_WM\_DESTROY(OnDestroy)  　　　　 MSG\_WM\_TIMER(OnTimer)  　　　　 MSG\_WM\_ERASEBKGND(OnEraseBkgnd)  　　　　 CHAIN\_MSG\_MAP(CFrameWindowImpl<CMyWindow>)  　　 END\_MSG\_MAP()  　　 LRESULT OnEraseBkgnd ( HDC hdc )  　　 {  　　 CDCHandle　dc(hdc);  　　 CRect　　　rc;  　　 SYSTEMTIME st;  　　 CString　　sTime;  　　 // 获取客户区域  　　 GetClientRect ( rc );  　　 // 构造字符串来显示在窗口  　　 GetLocalTime ( &st );  　　 sTime.Format ( \_T("The time is %d:%02d:%02d"), st.wHour, st.wMinute, st.wSecond );  // 保存DC，绘制文本  　　 dc.SaveDC();  　　 dc.SetBkColor ( RGB(255,153,0);  　　 dc.SetTextColor ( RGB(0,0,0) );  　　 dc.ExtTextOut ( 0, 0, ETO\_OPAQUE, rc, sTime, sTime.GetLength(), NULL );  　　 // 恢复DC.  　　 dc.RestoreDC(-1);  　 return 1;　　// 我们清除背景（ExtTextOut 已经做了）  }  }; |

这个消息处理函数不仅使用了CRect和CString类，还使用了一个GDI包装类CDCHandle。对于CString类，我想说的是它等同于MFC的CString类。我在后面的文章中还会介绍这些包装类，现在你只需要知道CDCHandle是对HDC的简单封装就行了，使用方法与MFC的 CDC类相似，只是CDCHandle的实例在超出作用域后不会销毁它所操作的设备上下文。

所有的工作完成了，现在看看我们的窗口是什么样子：

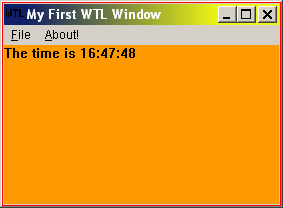


图3 简单WTL应用程序

例子代码中还使用了WM\_COMMAND响应菜单消息，在这里我不作介绍，但是你可以查看例子代码，看看WTL的COMMAND\_ID\_HANDLER\_EX宏是如何工作的。

### 2.2.2、从WTL的应用程序生成向导能得到什么

WTL的发布版本附带一个很棒的应用程序生成向导，让我们以一个SDI 应用为例看看它有什么特性。

**2.2.2.1、使用向导的整个过程**

　　在VC的IDE环境下单击File|New菜单，从列表中选择ATL/WTL AppWizard，我们要重写时钟程序，所以用WTLClock作为项目的名字：

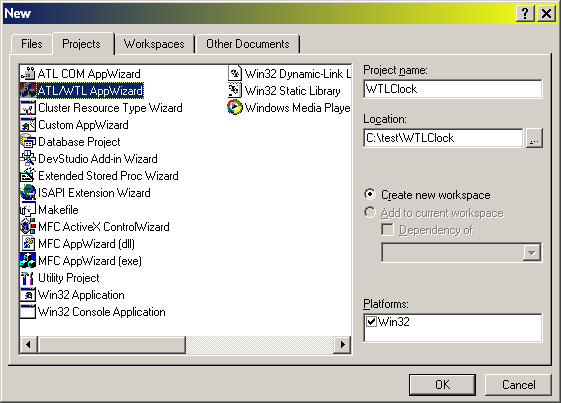


图4 WTL的应用程序向导

　　在下一页你可以选择项目的类型：SDI，MDI或者是基于对话框的应用，当然还有其它选项，如下图所示设置这些选项，然后点击“下一步”：



图5 WTL应用程序向导的应用程序类型选项

　　在最后一页你可以选择是否使用toolbar，rebar和status bar，为了简单起见，取消这些选项并单击“结束”。



图6 WTL应用程序的UI特性选项

**2.2.2.2、查看生成的代码**

向导完成后，在生成的代码中有三个类：CMainFrame, CAboutDlg, 和CWTLClockView，从名字上就可以猜出这些类的作用。虽然也有一个是视图类，但它仅仅是从CWindowImpl派生出来的一个简单的窗口类，没有象MFC那样的文档/视图结构。

　　还有一个\_tWinMain()函数，它先初始化COM环境，公用控件和\_Module，然后调用全局函数Run()。Run()函数创建主窗口并开始消息循环，Run()调用CMessageLoop::Run()，消息队列实际上是位于 CMessageLoop::Run()内，我将在下一个章节介绍CMessageLoop的更多细节。

　　CAboutDlg是CDialogImpl的派生类，它对应于ID IDD\_ABOUTBOX资源，我在第一部分已经介绍过对话框，所以你应该能看懂CAboutDlg的代码。

　 　CWTLClockView是我们程序的视图类，它的作用和MFC的视图类一样，没有标题栏，覆盖整个主窗口的客户区。CWTLClockView类有一个PreTranslateMessage()函数，也和MFC中的同名函数作用相同，还有一个WM\_PAINT的消息响应函数。这两个函数都没有什么特别之处，只是我们会填写OnPaint()函数来显示时间。

　　最后是我们的CMainFrame类，它有许多有趣的新东西，这是这个类的定义缩略版本：

|  |
| --- |
| class CMainFrame : public CFrameWindowImpl<CMainFrame>,  　　　　　　　　public CUpdateUI<CMainFrame>,  　　　　　　　　public CMessageFilter,  　　　　　　　　public CIdleHandler  {  public:  DECLARE\_FRAME\_WND\_CLASS(NULL, IDR\_MAINFRAME)  CWTLClockView m\_view;  virtual BOOL PreTranslateMessage(MSG\* pMsg);  virtual BOOL OnIdle();  BEGIN\_UPDATE\_UI\_MAP(CMainFrame)  END\_UPDATE\_UI\_MAP()  BEGIN\_MSG\_MAP(CMainFrame)  　　　　// ...  　　 CHAIN\_MSG\_MAP(CUpdateUI<CMainFrame>)  　　 CHAIN\_MSG\_MAP(CFrameWindowImpl<CMainFrame>)  END\_MSG\_MAP()  }; |

CMessageFilter是一个嵌入类，它提供PreTranslateMessage()函数，CIdleHandler也是一个嵌入类，它提供了OnIdle()函数。CMessageLoop，CIdleHandler和CUpdateUI三个类互相协同，完成界面元素的UI状态更新，就像MFC中的ON\_UPDATE\_COMMAND\_UI宏一样。

　　CMainFrame::OnCreate()中创建了视图窗口，并保存了这个窗口的句柄，当主窗口改变大小时视图窗口的大小也会随之改变。OnCreate ()函数还将CMainFrame对象添加到由CAppModule维护的消息过滤器队列和空闲处理队列，我将在稍后介绍这些。

|  |
| --- |
| LRESULT CMainFrame::OnCreate(UINT /\*uMsg\*/, WPARAM /\*wParam\*/,  　　　　　　　　　　　　　　 LPARAM /\*lParam\*/, BOOL& /\*bHandled\*/)  {  　　m\_hWndClient = m\_view.Create(m\_hWnd, rcDefault, NULL, | WS\_CHILD | WS\_VISIBLE |  WS\_CLIPSIBLINGS | WS\_CLIPCHILDREN, WS\_EX\_CLIENTEDGE);  // 为消息过滤和空闲更新登记对象  CMessageLoop\* pLoop = \_Module.GetMessageLoop();  pLoop->AddMessageFilter(this);  pLoop->AddIdleHandler(this);  return 0;  } |

　　m\_hWndClient是CFrameWindowImpl对象的一个成员变量，当主窗口大小改变时此窗口的大小也将改变。

在生成的CMainFrame中还添加了对File|New，File|Exit和 Help|About菜单的命令处理。我们的时钟程序不需要这些默认的菜单项，但是现在将它们留在代码中也没有害处。现在可以编译并运行向导生成的代码，不过这个程序确实没有什么用处。如果你感兴趣的话可以深入CMain Frame::CreateEx()函数的内部看看主窗口和它的资源是如何被加载和创建的。

　　我们的下一步WTL之旅是CMessageLoop，它掌管消息队列和空闲处理。

**2.2.2.3、CMessageLoop 的内部实现**

　 　CMessageLoop为我们的应用程序提供一个消息队列，除了一个标准的DispatchMessage/Translate Message循环外，它还通过调用PreTranslateMessage()函数实现了消息过滤机制，并通过调用OnIdle()实现了空闲处理功能。下面是Run()函数的伪代码:

|  |
| --- |
| int Run()  {  MSG msg;  for(;;)  {  　　 while ( !PeekMessage(&msg) )  　　　　 DoIdleProcessing();  　　 if ( 0 == GetMessage(&msg) )  　　　　 break;　　//从队列中返回WM\_QUIT  　　 if ( !PreTranslateMessage(&msg) )  　　 {  　　　　 TranslateMessage(&msg);  　　　　 DispatchMessage(&msg);  　　 }  }  return msg.wParam;  } |

那些需要过滤消息的类只需要象CMainFrame::OnCreate()函数那样调用CMessageLoop::Add MessageFilter()函数就行了，CMessageLoop就会知道该调用那个PreTranslateMessage()函数，同样， 如果需要空闲处理还可调用CMessageLoop::AddIdleHandler()函数。

　　需要注意得是：在这个消息循环中没有调用TranslateAccelerator()或IsDialogMessage()函数，因为CFrameWindowImpl在这之前已经做了处理。但是如果你在程序中使用了非模式对话框，那你就需要在CMainFrame:: PreTranslateMessage()函数中添加对IsDialogMessage()函数的调用。

**2.2.2.4、CFrameWindowImpl 的内部实现**

　 　CFrameWindowImpl 和它的基类 CFrameWindowImplBase提供了对工具条，rebars, 状态条,工具条按钮的提示和菜单项的掠过式帮助的支持，这些也是MFC的CFrameWnd类的基本特征。我会逐步介绍这些特征，完整的讨论 CFrameWindowImpl类需要再写两篇文章，但是现在看看CFrame WindowImpl是如何处理WM\_SIZE和它的客户区就足够了。需要记住一点前面提到的东西，m\_hWnd Client是CFrame WindowImplBase类的成员变量，它存储主窗口内的“视图”窗口的句柄。

　　CFrameWindowImpl类处理了WM\_SIZE消息：

|  |
| --- |
| **LRESULT OnSize(UINT /\*uMsg\*/, WPARAM wParam, LPARAM /\*lParam\*/, BOOL& bHandled)**  **{**  **if(wParam != SIZE\_MINIMIZED)**  **{**  **T\* pT = static\_cast<T\*>(this);**  **pT->UpdateLayout();**  **}**  **bHandled = FALSE;**  **return 1;**  **}** |

它首先检查窗口是否最小化，如果不是就调用UpdateLayout()，下面是UpdateLayout():

|  |
| --- |
| void UpdateLayout(BOOL bResizeBars = TRUE)  {  RECT rect;  GetClientRect(&rect);  // 定位条和偏移尺寸  UpdateBarsPosition(rect, bResizeBars);  // 调整客户窗口  if(m\_hWndClient != NULL)  　　 ::SetWindowPos(m\_hWndClient, NULL, rect.left, rect.top,rect.right - rect.left,  rect.bottom - rect.top, SWP\_NOZORDER | SWP\_NOACTIVATE);  } |

注意这些代码是如何使用m\_hWndClient的，既然m\_hWndClient是一般窗口的句柄，它就可能是任何窗口，对这个窗口的类型没有限制。 这一点不像MFC，MFC在很多情况下需要CView的派生类(例如分隔窗口类)。如果你回过头看看CMainFrame::OnCreate()就会看到它创建了一个视图窗口并赋值给m\_hWndClient，由m\_hWndClient确保视图窗口被设置为正确的大小。

**2.2.2.5、回到前面的时钟程序**

　　现在我们已经看到了主窗口类的一些细节，现在回到我们的时钟程序。视图窗口用来响应定时器消息并负责显示时钟，就像前面的CMyWindow类。下面是这个类的部分定义:

|  |
| --- |
| class CWTLClockView : public CWindowImpl<CWTLClockView>  {  public:  DECLARE\_WND\_CLASS(NULL)  BOOL PreTranslateMessage(MSG\* pMsg);  BEGIN\_MSG\_MAP\_EX(CWTLClockView)  　　 MESSAGE\_HANDLER(WM\_PAINT, OnPaint)  　　 MSG\_WM\_CREATE(OnCreate)  　　 MSG\_WM\_DESTROY(OnDestroy)  　　 MSG\_WM\_TIMER(OnTimer)  　　 MSG\_WM\_ERASEBKGND(OnEraseBkgnd)  END\_MSG\_MAP()  }; |

　　使用BEGIN\_MSG\_MAP\_EX代替BEGIN\_MSG\_MAP后，ATL的消息映射宏可以和WTL的宏混合使用，前面的例子在OnEraseBkgnd()中显示(画)时钟，现在被搬到了OnPaint()中。新窗口看起来是这个样子的：

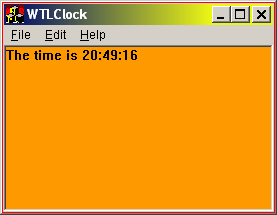


图7 显示的时钟程序

最后为我们的程序添加UI更新功能，为了演示这些用法，我们为窗口添加Start菜单和Stop菜单用于开始和停止时钟，Start菜单和Stop菜单将被适当的设置为可用和不可用。

**2.2.2.6、UI更新**

空闲时间的UI更新是几件事情协同工作的结果：CMessageLoop对象，嵌入类CIdleHandler 和 CUpdateUI。CMainFrame类从这两个嵌入类派生，当然还有CMainFrame类中的UPDATE\_UI\_MAP宏。CUpdateUI 能够操作5种不同的UI元素：**顶级菜单项(就是菜单条本身)，弹出式菜单的菜单项，工具条按钮，状态条的格子和子窗口(如对话框中的控件)**。每一种界面元素都对应CUpdateUIBase类的一个常量：

|  |  |
| --- | --- |
| UI元素 | 宏 |
| 菜单条项 | UPDUI\_MENUBAR |
| 弹出式菜单项 | UPDUI\_MENUPOPUP |
| 工具条按钮 | UPDUI\_TOOLBAR |
| 状态条格子 | UPDUI\_STATUSBAR |
| 子窗口 | UPDUI\_CHILDWINDOW |

　　CUpdateUI可以设置enabled状态，checked状态和文本（当然不是所有的界面元素都支持所有状态，比如一个子窗口是编辑框它就不能被check）。菜单项可以被设置为默认状态，这样它的文字会被加重显示。

　　要使用UI 更新需要做四件事：

1. 主窗口需要从CUpdateUI 和 CIdleHandler派生；
2. 将 CMainFrame 的消息链入 CUpdateUI类；
3. 将主窗口添加到模块的空闲处理队列；
4. 在主窗口中添加 UPDATE\_UI\_MAP 宏。

　　向导生成的代码已经为我们做了三件事，现在我们只需要决定哪个菜单项需要更新，他们是什么时候可用什么时候不可用。

**2.2.2.7、添加控制时钟的新菜单项**

　　在菜单条添加一个Clock菜单，它有两个菜单项：IDC\_START 和IDC\_STOP：

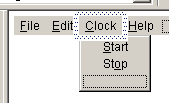


图8 添加菜单项

　　然后在UPDATE\_UI\_MAP宏中为每个菜单项添加一个入口：

|  |
| --- |
| class CMainFrame : public ...  {  public:  // ...  BEGIN\_UPDATE\_UI\_MAP(CMainFrame)  　　 UPDATE\_ELEMENT(IDC\_START, UPDUI\_MENUPOPUP)  　　 UPDATE\_ELEMENT(IDC\_STOP, UPDUI\_MENUPOPUP)  END\_UPDATE\_UI\_MAP()  // ...  }; |

　　我们只需要调用CUpdateUI::UIEnable()就可以改变这两个菜单项的任意一个的使能状态。UIEnable()有两个参数，一个是界面元素的ID，另一个是标志界面元素是否可用的bool型变量(true表示可用，false表示不可用)。

　　这套体系比MFC的ON\_UPDATE\_COMMAND\_UI体系笨拙一些，在MFC中我们只需编写处理函数，由MFC选择界面元素的显示状态，在WTL中我们需要告诉WTL界面元素的状态在何时改变。当然，这两个库都是在菜单将要显示的时候才应用菜单状态的改变。

**2.2.2.8、调用 UIEnable()**

　　现在返回到OnCreate()函数看看是如何设置Clock菜单的初始状态。

|  |
| --- |
| LRESULT CMainFrame::OnCreate(UINT /\*uMsg\*/, WPARAM /\*wParam\*/,LPARAM /\*lParam\*/, BOOL& bHandled)  {  m\_hWndClient = m\_view.Create(...);  // register object for message filtering and idle updates  // [omitted for clarity]  // 设置Clock菜单项的初始状态  UIEnable ( IDC\_START, false );  UIEnable ( IDC\_STOP, true );  return 0;  } |

　　我们的程序开始时Clock菜单是这样的：

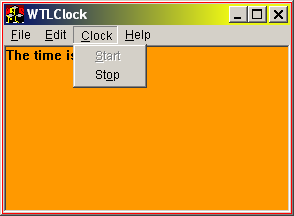


图9 UI更新显示效果

CMainFrame现在需要处理两个新菜单项，在视图类调用它们开始和停止时钟时处理函数需要翻转这两个菜单项的状态。这是MFC的内建消息处理机制所无法想象的地方之一。在MFC的程序中，所有的界面更新和命令消息处理必须完整的放在视图类中，但是在WTL中，主窗口类和视图类通过某种方式沟通； 菜单由主窗口拥有，主窗口获得这些菜单消息并做相应的处理，要么响应这些消息，要么发送给视图类。

　　这种沟通是通过 PreTranslateMessage()完成的，当然CMainFrame仍然要调用UIEnable()。CMain Frame可以将this指针传递给视图类，这样视图类也可以通过这个指针调用UIEnable()。在这个例子中，我选择的这种解决方案会导致主窗口和视图成为紧密耦合体，但是我发现这很容易理解(和解释!)。

|  |
| --- |
| class CMainFrame : public ...  {  public:  BEGIN\_MSG\_MAP\_EX(CMainFrame)  　　 // ...  　　 COMMAND\_ID\_HANDLER\_EX(IDC\_START, OnStart)  　　 COMMAND\_ID\_HANDLER\_EX(IDC\_STOP, OnStop)  END\_MSG\_MAP()  // ...  void OnStart(UINT uCode, int nID, HWND hwndCtrl);  void OnStop(UINT uCode, int nID, HWND hwndCtrl);  };  void CMainFrame::OnStart(UINT uCode, int nID, HWND hwndCtrl)  {  // Stop使能，Start无效  UIEnable ( IDC\_START, false );  UIEnable ( IDC\_STOP, true );  // 告诉视图启动时钟  m\_view.StartClock();  }  void CMainFrame::OnStop(UINT uCode, int nID, HWND hwndCtrl)  {  // Start使能，Stop无效  UIEnable ( IDC\_START, true );  UIEnable ( IDC\_STOP, false );  // 告诉视图停止时钟  m\_view.StopClock();  } |

　　每个处理函数都更新Clock菜单，然后在视图类中调用一个方法，选择在视图类中使用是因为时钟是由视图类控制得。StartClock() 和 StopClock()得代码没有列出，但可以在这个工程得例子代码中找到它们。

**2.2.2.9、消息映射链中最后需要注意的地方**

　　如果你使用VC 6，你会注意到将BEGIN\_MSG\_MAP改为BEGIN\_MSG\_MAP\_EX后ClassView显得有些杂乱无章：

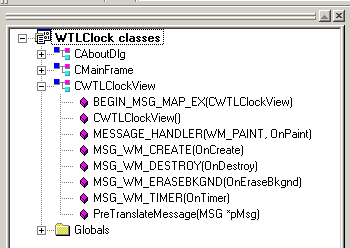


图10 改为BEGIN\_MSG\_MAP\_EX的类视图显示效果

　　出现这种情况是因为ClassView不能解释BEGIN\_MSG\_MAP\_EX宏，它以为所有得WTL消息映射宏是函数定义。你可以将宏改回为BEGIN\_MSG\_MAP并在stdafx.h文件得结尾处添加这两行代码来解决这个问题：

|  |
| --- |
| #undef BEGIN\_MSG\_MAP  　　#define BEGIN\_MSG\_MAP(x) BEGIN\_MSG\_MAP\_EX(x) |

# 第三章 工具条与状态条

工具条和状态条自从作为Windows 95的通用控件出现以来，就变成了很普遍的事物。由于MFC支持浮动的工具条，从而使它们更受欢迎。随着通用控件的更新，Rebars（最初被称为Coollbar）使得工具条有了另一种展示方式。在这一章，我将介绍WTL对这些控制条的支持和如何在你的程序中使用它们。

## 3.1、主窗口的工具条和状态条

CFrameWindowImpl有三个HWND类型的成员变量在窗口创建时被初始化，我们已经见过m\_hWndClient，它是填充主窗口客户区的“视图”窗口的句柄，现在我们介绍另外两个：

|  |  |
| --- | --- |
| 成员变量 | 说明 |
| m\_hWndToolBar | 工具条或Rebar的窗口句柄 |
| m\_hWndStatusBar | 状态条的窗口句柄 |

CFrameWindowImpl只支持一个工具条，也没有像MFC那样的可多点停靠的工具条，如果你想使用多个工具条又不想修改 CFrameWindowImpl的内部代码，你就需要使用Rebar。我将介绍它们二者并演示如何使用应用程序向导添加工具条和Rebar。

CFrameWindowImpl::OnSize()消息处理函数调用了UpdateLayout()，UpdateLayout()做两件事：重新定位所有控制条；改变视图窗口的大小使之填充整个客户区。实际工作是由UpdateBarsPosition()完成的，UpdateLayout()只是调用了该函数。实现的代码相当简单，向工具条和状态条发送WM\_SIZE消息，由这些控制条的默认窗口处理过程将它们定位到主窗口的顶部或底部。

　　当你告诉应用程序向导给你的窗口添加工具条和状态条时，向导就在CMainFrame::OnCreate()中添加了创建它们的代码。现在我们来看看这些代码，当然是为了再写一个时钟程序。

**3.1.1、向导为工具条和状态条生成代码**

　　我们将开始一个新的工程，让向导为主窗口创建工具条和状态条。首先创建一个名为WTLClock2的新工程，在向导的第一页，选SDI并使“生成CPP文件”检查框被选中：



图11 WTL应用程序向导：应用程序类型选项

　　在第二页，取消Rebar使向导仅仅创建一个普通的工具条：



图12 WTL应用程序向导：UI选项

　　从第二部分的程序中复制相应的代码，新程序看起来是这样的：

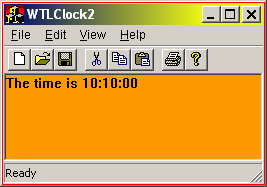


图13 新的应用程序

**3.1.2、如何创建工具条和状态条**

　　在这个例子中，向导在CMainFrame::OnCreate()函数中添加了更多的代码，这些代码的作用就是创建控制条，通知CUpdateUI更新工具条上的按钮。

|  |
| --- |
| LRESULT CMainFrame::OnCreate(UINT /\*uMsg\*/, WPARAM /\*wParam\*/,LPARAM /\*lParam\*/, BOOL& bHandled)  {  **CreateSimpleToolBar();**  **CreateSimpleStatusBar();**  m\_hWndClient = m\_view.Create(...);  // ...  // 为消息过滤和空闲更新登记对象  CMessageLoop\* pLoop = \_Module.GetMessageLoop();  ATLASSERT(pLoop != NULL);  pLoop->AddMessageFilter(this);  pLoop->AddIdleHandler(this);  return 0;  } |

这是新添加的代码的开始部分，CFrameWindowImpl::CreateSimpleToolBar()函数使用资源 IDR\_ MAINFRAME创建工具条并将其句柄赋值给m\_hWndToolBar，下面是CreateSimpleToolBar()函数的代码：

|  |
| --- |
| BOOL CFrameWindowImpl::CreateSimpleToolBar(UINT nResourceID = 0,DWORD dwStyle =  ATL\_SIMPLE\_TOOLBAR\_STYLE,　UINT nID = ATL\_IDW\_TOOLBAR)  {  ATLASSERT(!::IsWindow(m\_hWndToolBar));  if(nResourceID == 0)  　　 nResourceID = T::GetWndClassInfo().m\_uCommonResourceID;  m\_hWndToolBar = T::CreateSimpleToolBarCtrl(m\_hWnd, nResourceID, TRUE,dwStyle, nID);  return (m\_hWndToolBar != NULL);  } |

参数说明：

|  |  |
| --- | --- |
| 参数 | 说 明 |
| nResourceID | 工具条资源得ID。如果使用默认值0作为参数，程序将使用DECLARE\_FRAME\_ WND\_CLASS宏指定得资源，这里使用的 IDR\_MAINFRAME是向导生成的代码。 |
| dwStyle | 指出工具条的类型或样式。默认值ATL\_SIMPLE\_TOOLBAR\_STYLE被定义为TBS TYLE\_TOOLTIPS，子窗口和可见三种风格的结合， 这使得鼠标移到按钮上时工具条会弹出工具提示。 |
| nID | 工具条的窗口ID，通常都会使用默认值。 |

　 　CreateSimpleToolBar()首先检查是否已经创建了一个工具条，然后调用CreateSimpleToolBarCtrl()函数创建工具条控件，CreateSimpleToolBarCtrl()把返回的工具条控件句柄保存在m\_hWndToolBar中。

CreateSimpleToolBarCtrl()负责读出资源并创建相应的工具条按钮，然后返回工具条窗口的句柄。这部分的代码相当长，我不在这里做具体介绍，如果你对此感兴趣得话何以在atlframe.h中找到这些代码。

　　OnCreate()函数接下来会调用CFrameWindowImpl::CreateSimpleStatusBar()函数，此函数创建状态条，将句柄存在m\_hWndStatusBar，下面是该函数的代码：

|  |
| --- |
| BOOL CFrameWindowImpl::CreateSimpleStatusBar(UINT nTextID = ATL\_IDS\_IDLEMESSAGE,  DWORD dwStyle = ... SBARS\_SIZEGRIP,UINT nID = ATL\_IDW\_STATUS\_BAR)  {  TCHAR szText[128];　　// max text lentgth is 127 for status bars  szText[0] = 0;  ::LoadString(\_Module.GetResourceInstance(), nTextID, szText, 128);  return CreateSimpleStatusBar(szText, dwStyle, nID);  } |

显示在状态条的文字是从字符串资源中装载的，这个函数的参数说明如下：

|  |  |
| --- | --- |
| 参数 | 说明 |
| nTextID | 用于在状态条上显示的字符串的资源ID，向导生成的ATL\_IDS\_IDLEMESSAGE对应的字符串是“Ready”。 |
| dwStyle | 状态条的样式。默认值包含了SBARS\_SIZEGRIP风格，这使得状态条的右下角会显示一个改变窗口大小的标志。 |
| nID | 状态条的窗口ID，通常都会使用默认值。 |

　　CreateSimpleStatusBar()调用另外一个重载函数创建状态条：

|  |
| --- |
| BOOL CFrameWindowImpl::CreateSimpleStatusBar(LPCTSTR lpstrText,DWORD dwStyle = ...  SBARS\_SIZEGRIP,　UINT nID = ATL\_IDW\_STATUS\_BAR)  {  ATLASSERT(!::IsWindow(m\_hWndStatusBar));  m\_hWndStatusBar = ::CreateStatusWindow(dwStyle, lpstrText, m\_hWnd, nID);  return (m\_hWndStatusBar != NULL); } |

　　这个重载的版本首先检查是否已经创建了状态条，然后调用CreateStatusWindow()创建状态条，状态条的句柄存放在m\_hWndStatusBar中。

**3.1.3、显示和隐藏工具条和状态条**

　 　CMainFrame类也有一个视图菜单，它有两个命令：显示/隐藏工具条和状态条，它们的ID是ID\_ VIEW\_TOOLBAR和 ID\_VIEW\_STATUS\_BAR。CMainFrame类有这两个命令的响应函数，分别显示和隐藏相应的控制条，下面是OnViewToolBar ()函数的代码：

|  |
| --- |
| LRESULT CMainFrame::OnViewToolBar(WORD /\*wNotifyCode\*/, WORD /\*wID\*/,HWND /\*hWndCtl\*/,  BOOL& /\*bHandled\*/)  {  BOOL bVisible = !::IsWindowVisible(m\_hWndToolBar);  ::ShowWindow(m\_hWndToolBar, bVisible ? SW\_SHOWNOACTIVATE : SW\_HIDE);  UISetCheck(ID\_VIEW\_TOOLBAR, bVisible);  UpdateLayout();  return 0; } |

　　这些代码翻转控制条的显示状态，相应的翻转View|Toolbar菜单上的检查标记，然后调用Update Layout()重新定位控制条并改变视图窗口的大小。

**3.1.4、工具条和状态条的内在特征**

　　MFC的框架提供了很多好的特性，例如工具条按钮的工具提示和菜单项的掠过式帮助。WTL中相对应的功能实现在CFrameWindowImpl类中。下面的屏幕截图显示了工具提示和掠过式帮助。

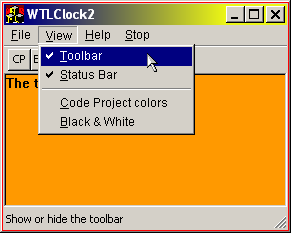
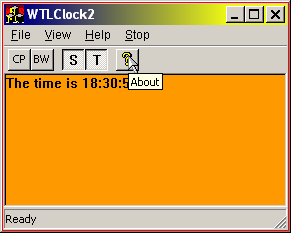


图14 工具条按钮提示与菜单项掠过式帮助

　 　CFrameWindowImplBase类有两个消息处理函数用来实现这些功能，OnMenuSelect()处理WM\_ MENUSELECT消息，它像MFC那样查找掠过式帮助的字符串：首先装载与菜单资源ID相同的字符串资源，在字符串中查找 \n 字符，使用\n之前的内容作为掠过帮助的内容。OnToolTipTextA() 和 On ToolTipTextW() 函数分别响应 TTN\_GETDISPINFOA消息和TTN\_GETDISPINFOW消息，提供工具条按钮的工具提示。这两个处理函数和OnMenuSelect ()函数一样装载相应的字符串，只是使用\n后面的字符串（注：OnMenuSelect()和OnToolTipTextA()函数对于DBCS字符是不安全的，因为它在查找\n字符时没有检查DBCS字符串的头部和尾部）。下面是工具条及其关联的帮助字符串的例子：

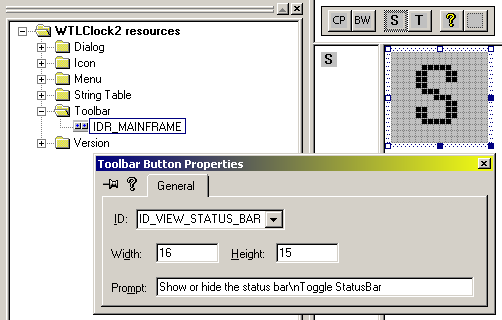


图15工具条按钮提示与菜单项掠过式帮助的创建

**3.1.5、创建不同样式的工具条**

如果你不喜欢在工具条上显示3D按钮（尽管从可用性观点来看平面的界面元素是件糟糕的事情），你可以通过改变CreateSimpleToolBar ()函数的参数来改变工具条的样式。例如，你可以在CMain Frame::OnCreate()使用如下代码创建一个IE风格的工具条：

|  |
| --- |
| CreateSimpleToolBar ( 0, ATL\_SIMPLE\_TOOLBAR\_STYLE |TBSTYLE\_FLAT | TBSTYLE\_LIST ); |

　　如果你使用向导为你的程序添加了manifest文件，它就会在Windows XP系统上使用6.0版的通用控件，你不能选择按钮的类型，工具条会自动使用平面按钮即使你创建工具条时没有添加TBS TYLE\_ FLAT风格。

**3.1.6、工具条编辑器**

　　正如我们前面所见，向导为我们的程序创建了几个默认的按钮，当然只有About按钮有事件处理。你可以像在MFC的工程中那样，使用工具条编辑器修改工具条资源，CreateSimpleToolBarCtrl()用这个工具条资源创建工具条。下面是向导生成的工具条在编辑器中的样子：

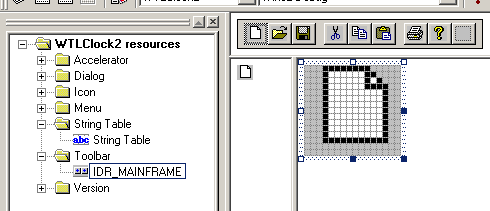


图16 编辑工具条资源

　　对于我们的时钟程序，我们添加四个按钮，两个按钮用来改变视图窗口的颜色，另外两个用来显示/隐藏工具条和状态条。下面是我们的新工具条：

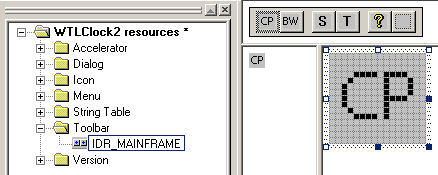


图17 添加工具条上的资源

　　这些按钮是：

* IDC\_CP\_COLORS：将视图窗口颜色改为CodeProject网站的颜色
* IDC\_BW\_COLORS：将视图窗口颜色改为黑白颜色
* ID\_VIEW\_STATUS\_BAR：显示或隐藏状态条
* ID\_VIEW\_TOOLBAR：显示或隐藏工具条

前面两个按钮都有相应的菜单项，它们都调用视图类的一个新函数SetColor()，向这个函数传递前景颜色和背景颜色，视图窗口用这两个参数改变窗口的显示。响应这两个按钮的处理函数与响应相应的菜单项的处理函数在使用COMMAND\_ID\_HANDLER\_EX宏上没有区别，你可以查看例子工程的代码，了解这些消息处理的细节。在下一节我将介绍状态条和工具条按钮的UI状态更新，使它们能够反映工具条或状态条当前的状态。

**3.1.7、工具条按钮的UI状态更新**

　　向导生成的代码已经为CMainFrame添加了对View|Toolbar和View|Status Bar两个菜单项的Check和Uncheck的UI更新处理。这和第二章的程序一样：对CMainFrame类的两个命令使用UI更新的宏：

|  |
| --- |
| BEGIN\_UPDATE\_UI\_MAP(CMainFrame)  UPDATE\_ELEMENT(ID\_VIEW\_TOOLBAR, UPDUI\_MENUPOPUP)  UPDATE\_ELEMENT(ID\_VIEW\_STATUS\_BAR, UPDUI\_MENUPOPUP)  END\_UPDATE\_UI\_MAP() |

　　我们的时钟程序的工具条按钮与对应的菜单项有相同的ID，所以第一步就是为每个宏添加UPDUI\_TOOLBAR标志：

|  |
| --- |
| BEGIN\_UPDATE\_UI\_MAP(CMainFrame)  UPDATE\_ELEMENT(ID\_VIEW\_TOOLBAR, UPDUI\_MENUPOPUP | UPDUI\_TOOLBAR)  UPDATE\_ELEMENT(ID\_VIEW\_STATUS\_BAR, UPDUI\_MENUPOPUP | UPDUI\_TOOLBAR)  END\_UPDATE\_UI\_MAP() |

　　还需要添加两个函数响应工具条按钮的更新。幸运地是,向导已经为我们做了，所以如果此时编译这个程序，菜单项和工具条按钮都会更新。

**3.1.8、使一个工具条支持UI状态更新**

　　如果查看CMainFrame::OnCreate()的代码你就会发现一段新的代码，这段代码设置了两个菜单项的初始状态：

|  |
| --- |
| LRESULT CMainFrame::OnCreate( ... )  {  // ...  m\_hWndClient = m\_view.Create(...);  UIAddToolBar(m\_hWndToolBar);  UISetCheck(ID\_VIEW\_TOOLBAR, 1);  UISetCheck(ID\_VIEW\_STATUS\_BAR, 1);  // ...  } |

　　UIAddToolBar()将工具条的窗口句柄传给CUpdateUI，所以当需要更新按钮的状态时CUpdateUI会向这个窗口发消息。另一个重要的调用位于OnIdle()中：

|  |
| --- |
| BOOL CMainFrame::OnIdle()  {  **UIUpdateToolBar();**  return FALSE;  } |

当消息队列中没有消息等待时CMessageLoop::Run()就会调用OnIdle()，UIUpdateToolBar()遍历UI更新表，寻找那些带有UPDUI\_TOOLBAR标志又被UISetCheck()之类的函数改变了状态的的界面元素(当然是工具条)，相应的改变按钮的状态。需要注意的是：如果更新弹出式菜单的状态就不需要做以上两步，因为CUpdateUI响应WM\_INITMENUPOPUP消息，只有接到此消息时才更新菜单状态。

　　如果查看例子代码就会发现，它也演示了如何更新框架窗口的菜单条上的顶级菜单项的状态。有一个菜单项是执行Start和Stop命令，起到开始和停止时钟的作用，当然这需要做一些不平常的事情：菜单条上的菜单项总是处于弹出状态。为了完整的介绍CUpdateUI我将它们也加进例子代码中，要了解它们可以查找对UIAddMenuBar()和UIUpdateMenuBar()两个函数的调用。

**3.1.9、使用Rebar代替简单的工具条**

　　CFrameWindowImpl也支持使用Rebar控件，使你的程序看起来像IE，使用 Rebar也是在程序中使用多个工具条的一个方法(译者加：前面讲过，另一个方法就是修改WTL的源代码)。要使用Rebar需要在向导的第二页选上支持 Rebar的检查框，如下所示：



图18 在向导中增加Rebar

　　第二个例子工程WTLClock3就使用了Rebar控件，如果你正在跟着例子代码学习，那现在就打开WTLClock3。

　　你首先会注意到创建工具条的代码有些不同，出现这种感觉是因为我们在程序中使用了rebar。以下是相关的代码：

|  |
| --- |
| LRESULT CMainFrame::OnCreate(...)  {  HWND hWndToolBar = CreateSimpleToolBarCtrl ( m\_hWnd,IDR\_MAINFRAME, FALSE,  　 ATL\_SIMPLE\_TOOLBAR\_PANE\_STYLE );  CreateSimpleReBar(ATL\_SIMPLE\_REBAR\_NOBORDER\_STYLE);  AddSimpleReBarBand(hWndToolBar);  // ...  } |

代码从创建工具条开始，只是使用了不同的风格，也就是ATL\_SIMPLE\_TOOLBAR\_PANE\_ STYLE，它定义在atlframe.h文件 中，与ATL\_SIMPLE\_TOOLBAR\_STYLE风格相似，只是附加了一些诸如CCS\_NOPARENTALIGN之类的风格，这是使工具条作为 Rebar的子窗口能够正常工作所必需的风格。

　　下一行代码是调用CreateSimpleReBar()函数，该函数创建Rebar控件并将句柄存到m\_hWnd ToolBar中。接下来调用AddSimpleReBarBand()函数为Rebar创建一个条位并告诉Rebar这个条位上是一个工具条。

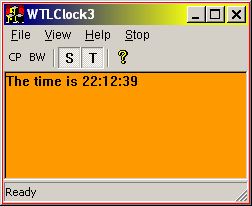


图19 Rebar的效果

　　CMainFrame::OnViewToolBar()函数也有些不同，它只隐藏Rebar上工具条所在的条位而不是隐藏m\_hWndToolBar(如果隐藏m\_hWndToolBar将隐藏整个Rebar而不仅仅是工具条)。

如果你使用多个工具条，只需像向导为我们生成的关于第一个工具条的代码那在OnCreate()创建它们并调用AddSimpleReBarBand ()添加到Rebar就行了。CFrameWindowImpl使用标准的Rebar控件，不像MFC那样支持可停靠的工具条，你所能作得就是排列这些工具条在Rebar中的位置。

**3.1.10、多窗格的状态条**

　　WTL有一个状态条类实现多窗格的状态条，与MFC的默认的状态条一样有CAPS，LOCK和NUM LOCK指示器，这个类就是CMultiPaneStatusBarCtrl。在WTLClock3例子工程中演示了如何使用这个类。这个类支持有限的UI更新，当弹出式菜单被显示时有“Default”属性的窗格会延伸到整个状态条的宽度用于显示菜单的掠过式帮助。

　　第一步就是在CMainFrame中声明一个CMultiPaneStatusBarCtrl类型的成员变量：

|  |
| --- |
| class CMainFrame : public ...  {  //...  protected:  CMultiPaneStatusBarCtrl m\_wndStatusBar;  }; |

　　接着在OnCreate()中创建状态条，并支持UI更新：

|  |
| --- |
| m\_hWndStatusBar = m\_wndStatusBar.Create ( \*this );  UIAddStatusBar ( m\_hWndStatusBar ); |

　　就像CreateSimpleStatusBar()函数做得那样，我们也将状态条的句柄存放在m\_hWndStatusBar中。

　　下一步就是调用CMultiPaneStatusBarCtrl::SetPanes()函数建立窗格：

|  |
| --- |
| BOOL SetPanes(int\* pPanes, int nPanes, bool bSetText = true); |

参数说明：

|  |  |
| --- | --- |
| 参数 | 说 明 |
| pPanes | 存放窗格ID的数组 |
| nPanes | 窗格ID数组中元素的个数(译者加：就是窗格数) |
| bSetText | 如果是true，所有的窗格被立即设置文字，这一点将在下面解释。 |

窗格ID可以是ID\_DEFAULT\_PANE，此ID用于创建支持掠过式帮助的窗格，窗格ID也可以是字符串资源ID。对于非默认的窗格WTL装载这个ID对应的字符串并计算宽度，并将窗格设置为相应的宽度，这和MFC使用的逻辑是一样的。

　　bSetText控制着窗格是否立即显示相关的字符串，如果是true，SetPanes()显示每个窗格的字符串，否则窗格就被置空。

下面是我们对SetPanes()的调用：

|  |
| --- |
| // 创建状态条窗格  int anPanes[] = { ID\_DEFAULT\_PANE, IDPANE\_STATUS,IDPANE\_CAPS\_INDICATOR };  m\_wndStatusBar.SetPanes ( anPanes, 3, false ); |

IDPANE\_STATUS对应的字符串是“@@@@”，这样应该有足够的宽度(希望是)显示两个时钟状态字符串“Running”和 “Stopped”。和MFC一样，你需要自己估算窗格的宽度，IDPANE\_CAPS\_ INDICATOR对应的字符串是“CAPS”。

**3.1.11、窗格的UI状态更新**

　　为了更新窗格上的文本，我们需要将相应的窗格添加到UI更新表：

|  |
| --- |
| BEGIN\_UPDATE\_UI\_MAP(CMainFrame)  　　　　//...  　　　　UPDATE\_ELEMENT(1, UPDUI\_STATUSBAR)　// 时钟状态  　　　　UPDATE\_ELEMENT(2, UPDUI\_STATUSBAR)　// CAPS指示器  END\_UPDATE\_UI\_MAP() |

　　这个宏的第一个参数是窗格的索引而不是ID。这很不幸，因为如果你重新排列了窗格，你要记得更新UI更新表。

　　由于我们在调用SetPanes()是第三个参数是false，所以窗格初始是空的。我们下一步要做得就是将时钟状态窗格的初始文本设为“Running”

|  |
| --- |
| // 设置时钟状态格的初始文本  　　UISetText ( 1, \_T("Running") ); |

　　和前面一样，第一个参数是窗格的索引。UISetText()是状态条唯一支持的UI更新函数。

　　最后，在CMainFrame::OnIdle()中添加对UIUpdateStatusBar()函数的调用，使状态条的窗格能够在空闲时间被更新：

|  |
| --- |
| BOOL CMainFrame::OnIdle()  {  UIUpdateToolBar();  UIUpdateStatusBar();  return FALSE;  } |

当你使用UIUpdateStatusBar()时，CUpdateUI的一个问题就暴露出来了－－菜单项的文本在调用UISetText()后没有改变！如果你在看WTLClock3工程的代码，时钟的开始/停止菜单项被移到了Clock菜单，在菜单项命令的响应处理函数中设置菜单项的文本。无论如何，如果当前调用的是UIUpdate StatusBar()，那么对UISetText()的调用就不会起作用。我没有研究这个问题是否可以被修复，所以如果你打算改变菜单的文本，你需要留意这个地方。

　　最后，我们需要检查CAPS LOCK键的状态，更新相应的两个窗格。这些代码是通过OnIdle()被调用的，所以程序会在每次空闲时间检查它们的状态。

|  |
| --- |
| BOOL CMainFrame::OnIdle()  {  // Check the current Caps Lock state, and if it is on, show the  // CAPS indicator in pane 2 of the status bar.  if ( GetKeyState(VK\_CAPITAL) & 1 )  　　 UISetText ( 2, CString(LPCTSTR(IDPANE\_CAPS\_INDICATOR)) );  else  　　　　UISetText ( 2, \_T("") );  UIUpdateToolBar();  UIUpdateStatusBar();  return FALSE;  } |

　　第一次调用UISetText()时将从字符串资源中装载“CAPS”字符串，但是在CString的构造函数中使用了一个灵巧的窍门(有充分的文档说明)。

　　在完成所有的代码之后，状态条看起来是这个样子：

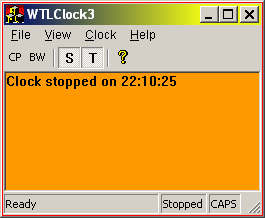


图20 新的状态条

# 第四章 对话框与控件

MFC 的对话框和控件的封装真的可以节省你很多时间和功夫。如果没有MFC对控件的封装，你要操作控件就得耐着性子填写各种结构，写很多的SendMessage()调用。MFC还提供了对话框动态数据交换(DDX)，它可以在控件和变量之间传输数据。WTL 当然也提供了这些功能，并对控件的封装做了很多改进。本章将着眼于一个基于对话框的程序来演示你以前用MFC实现的功能，除此之外还有WTL消息处理的增强功能。第五章将介绍高级界面特性和WTL对新控件的封装。

**4.1、回顾一下ATL的对话框**

　　现在回顾一下第一章提到的两个对话框类，CDialogImpl 和 CAxDialogImpl。CAxDialogImpl用于包含ActiveX控件的对话框。本文不准备介绍ActiveX控件，所以只使用CDialogImpl。

**4.1.1创建一个对话框需要做三件事**

1> 创建一个对话框资源；

2> 从CDialogImpl类派生一个新类；

3> 添加一个公有成员变量IDD，将它设置为对话框资源的ID。

　　然后就像主框架窗口那样添加消息处理函数，WTL没有改变这些，不过确实添加了一些其他能够在对话框中使用得特性。

**4.2、通用控件的封装类**

　　WTL有许多控件的封装类对你应该比较熟悉，因为它们使用与MFC相同(或几乎相同)的名字。控件的方法的命名也和MFC一样，所以你可以参照MFC的文档使用这些WTL的封装类。不足之处是F12键不能方便地跳到类的定义代码处。

下面是Windows内建控件的封装类：

|  |  |
| --- | --- |
| 控件类型 | 封装类 |
| 标准控件 | CStatic, CButton, CListBox, CComboBox, CEdit, CScrollBar, CDragListBox |
| 通用控件 | CImageList, CListViewCtrl (CListCtrl), CTreeViewCtrl (CTreeCtrl), CHeaderCtrl, CToolBarCtrl, CStatusBarCtrl, CTabCtrl, CToolTipCtrl, CTrackBarCtrl (CSliderCtrl), CUpDownCtrl (CSpinButtonCtrl), CProgressBarCtrl, CHotKeyCtrl, CAnimateCtrl, CRichEditCtrl, CReBarCtrl, CComboBoxEx, CDateTimePickerCtrl, CMonthCalendarCtrl, CIPAddressCtrl（刮号内的MFC类） |
| MFC中没有的封装类 | CPagerCtrl, CFlatScrollBar, CLinkCtrl (可点击的超联接，只在XP可用) |

还有一些是WTL特有的类：CBitmapButton, CCheckListViewCtrl (带检查选择框的list控件), CTree ViewCtrlEx 和 CTreeItem (通常一起使用， CTreeItem 封装了HTREEITEM)， CHyperLink (类似于网页上的超链接对象，支持所有操作系统)。

　　需要注意得一点是大多数封装类都是基于CWindow的，它们封装了HWND并对控件的消息进行了封装(例如，CListBox:: GetCurSel()封装了LB\_GETCURSEL消息)。所以和CWindow一样，创建一个控件的封装对象并将它与已经存在的控件关联起来只占用很少的资源，当然也和CWindow一样，控件封装对象销毁时不销毁控件本身。也有一些例外，如CBitmapButton, CCheck ListViewCtrl和CHyperLink。

　　由于本书定位于有经验的MFC程序员，就不浪费时间介绍这些封装类，它们和MFC相应的控件封装相似。当然我会介绍WTL的新类：CBitmapButton。CBitmapButton类与MFC的同名类有很大的不同，CHyperLink则完全是新事物。

**4.2.1、用应用程序向导生成基于对话框的程序**

运行VC并启动WTL应用向导，相信你在做时钟程序时已经用过它了，为我们的新程序命名为ControlMania1。在向导的第一页选择基于对话框的应用，还要选择是使用模式对话框，还是使用非模式对话框。它们有很大的区别，我将在第五章介绍它们的不同，现在我们选择简单的一种：模式对话框。如下所示选择模式对话框和生成CPP文件选项：



图21 选择对话框和生成CPP文件的应用程序向导

　　第二页上所有的选项只对主窗口是框架窗口时有意义，现在它们是不可用状态，单击"Finish"，再单击"OK"完成向导。

　　正如你想的那样，向导生成的基于对话框程序的代码非常简单。\_tWinMain()函数在Control Mania1.cpp中，下面是重要的部分：

|  |
| --- |
| int WINAPI \_tWinMain (HINSTANCE hInstance, HINSTANCE /\*hPrevInstance\*/,  LPTSTR lpstrCmdLine, int nCmdShow )  {  HRESULT hRes = ::CoInitialize(NULL);  AtlInitCommonControls(ICC\_COOL\_CLASSES | ICC\_BAR\_CLASSES);  hRes = \_Module.Init(NULL, hInstance);  int nRet = 0;  // BLOCK: Run application  {  　　 CMainDlg dlgMain;  　　 nRet = dlgMain.DoModal();  }  \_Module.Term();  ::CoUninitialize();  return nRet;  } |

在代码中，首先初始化COM并创建一个单线程工程，这对于使用ActiveX控件的对话框是有必要得，接着调用WTL的函数 AtlInitCommonControls()，这个函数是对InitCommonControlsEx()的封装。全局对象\_Module被初始化， 主对话框显示出来（注意所有使用DoModal()创建的ATL对话框实际上是模式的，这不像MFC，MFC的所有对话框是非模式的，MFC通过在代码中禁用对话框的父窗口来模拟模式对话框的行为）。最后，\_Module和COM被释放，DoModal()的返回值被用来作为程序的结束码。

　 将CMainDlg变量放在一个区块中是很重要的，因为CMainDlg可能有成员使用了ATL和WTL的特性，这些成员在析构时也会用到 ATL/WTL的特性，如果不使用区块，CMainDlg将在\_Module.Term ()（这个函数完成ATL/WTL的清理工作）调用之后调用析构函数销毁自己（和成员），并试图使用ATL/ WTL的特性，这将导致程序出现诊断错误崩溃（WTL 3的向导生成的代码没有使用区块，使得我的一些程序在结束时崩溃）。

　　你现在可以编译并运行这个程序，尽管它只是一个简陋的对话框：

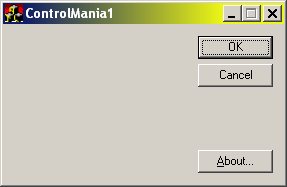


图22 简单对话框应用程序

　　CMainDlg 的代码处理了WM\_INITDIALOG, WM\_CLOSE和三个按钮的消息，如果你喜欢可以浏览一下这些代码，你应该能够看懂CMainDlg的声明，以及它的消息映射和它的消息处理函数。

　　这个简单的工程还演示了如何将控件和变量联系起来，这个程序使用了几个控件。在接下来的讨论中你可以随时回来查看这些图表。

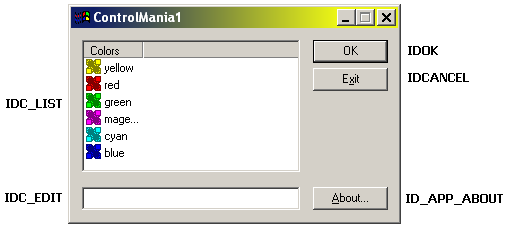


图23 为简单对话框添加新的成员

　　由于程序使用了list view控件，所以对AtlInitCommonControls()的调用需要作些修改，将其改为：

|  |
| --- |
| AtlInitCommonControls ( ICC\_WIN95\_CLASSES ); |

　　虽然这样注册的控件类比我们用到的多，但是当我们向对话框添加不同类型的控件时就不用随时记得添加名为ICC\_\*的常量。

**4.3、使用控件的封装类**

有几种方法将一个变量和控件建立起关联，可以使用CWindows（或其它Window接口类，如CListViewCtrl），也可以使用 CWindowImpl的派生类。如果只是需要一个临时变量就用CWindow，如果需要子类化一个控件，并处理发送给该控件的消息就需要使用 CWindowImpl。

**4.3.1、ATL 方式1 ----Attach方法**

连接一个CWindow对象最简单的方法是声明一个CWindow或其它窗口接口类，然后调用Attach ()方法，还可以使用CWindow的构造函数直接将变量与控件的HWND关联起来。

　　下面的代码三种方法将变量和一个list控件联系起来：

|  |
| --- |
| HWND hwndList = GetDlgItem(IDC\_LIST);  CListViewCtrl wndList1 (hwndList);　// 使用构造函数  CListViewCtrl wndList2, wndList3;  wndList2.Attach ( hwndList );　　 // 使用Attach方法  wndList3 = hwndList;　　　　　　　 // 使用赋值操作 |

记住CWindow的析构函数并不销毁控件窗口，所以在变量超出作用域时不需要将其脱离控件，如果你愿意的话还可以将其作为成员变量使用：你可以在OnInitDialog()处理函数中建立变量与控件的联系。

**4.3.2、ATL 方式2 ----包容器窗口(CContainedWindow)**

CContainedWindow是介于CWindow和CWindowImpl之间的类，它可以子类化控件，在控件的父窗口中处理控件的消息，这使得所有的消息处理都放在对话框类中，不需要为每个控件生成一个单独的CWindowImpl派生类对象。需要注意的是：不能用 CContainedWindow 处理WM\_COMMAND、WM\_ NOTIFY和其他通知消息，因为这些消息是发给控件的父窗口的。

CContainedWindow只是CContainedWindowT定义的一个数据类型，CContainedWindowT才是真正的类，它是一个模板类，使用窗口接口类的类名作为模板参数。这个特殊的CContainedWindowT <CWindow>和CWindow 功能一样，CContainedWindow只是它定义的一个简写名称，要使用不同的window接口类只需将该类的类名作为模板参数就行了，例如CContainedWindowT<CListViewCtrl>。

　 钩住一个CContainedWindow对象需要做四件事：

1. 在对话框中创建一个CContainedWindowT 成员变量；
2. 将消息处理添加到对话框消息映射的ALT\_MSG\_MAP小节；
3. 在对话框的构造函数中调用CContainedWindowT 构造函数并告诉它哪个ALT\_MSG\_MAP小节的消息需要处理；
4. 在OnInitDialog()中调用CContainedWindowT::SubclassWindow() 方法与控件建立关联。

　　在ControlMania1中，我对三个按钮分别使用了一个CContainedWindow，对话框处理发送到每一个按钮的WM\_SETCURSOR消息，并改变鼠标指针形状。

　　现在仔细看看这一步，首先，我们在CMainDlg中添加了CContainedWindow成员。

|  |
| --- |
| class CMainDlg : public CDialogImpl<CMainDlg>  {  // ...  protected:  CContainedWindow m\_wndOKBtn, m\_wndExitBtn;  }; |

　　其次，我们添加了ALT\_MSG\_MAP小节，OK按钮使用1小节，Exit按钮使用2小节。这意味着所有发送给OK按钮的消息将由ALT\_MSG\_MAP(1)小节处理，所有发给Exit按钮的消息将由ALT\_MSG\_MAP(2)小节处理。

|  |
| --- |
| class CMainDlg : public CDialogImpl<CMainDlg>  {  public:  BEGIN\_MSG\_MAP\_EX(CMainDlg)  　　 MESSAGE\_HANDLER(WM\_INITDIALOG, OnInitDialog)  　　 COMMAND\_ID\_HANDLER(ID\_APP\_ABOUT, OnAppAbout)  　　 COMMAND\_ID\_HANDLER(IDOK, OnOK)  　　 COMMAND\_ID\_HANDLER(IDCANCEL, OnCancel)  ALT\_MSG\_MAP(1)  　　 MSG\_WM\_SETCURSOR(OnSetCursor\_OK)  ALT\_MSG\_MAP(2)  　　 MSG\_WM\_SETCURSOR(OnSetCursor\_Exit)  END\_MSG\_MAP()  LRESULT OnSetCursor\_OK(HWND hwndCtrl, UINT uHitTest, UINT uMouseMsg);  LRESULT OnSetCursor\_Exit(HWND hwndCtrl, UINT uHitTest, UINT uMouseMsg);  }; |

　　接着，我们调用每个CContainedWindow的构造函数，告诉它使用ALT\_MSG\_MAP的哪个小节。

|  |
| --- |
| CMainDlg::CMainDlg() : m\_wndOKBtn(this, 1)  , m\_wndExitBtn(this, 2)  {  } |

　　构造函数的参数是消息映射链的地址和ALT\_MSG\_MAP的小节号码，第一个参数通常使用this，就是使用对话框自己的消息映射链，第二个参数告诉对象将消息发给ALT\_MSG\_MAP的哪个小节。

　　最后，我们将每个CContainedWindow对象与控件关联起来。

|  |
| --- |
| LRESULT CMainDlg::OnInitDialog(...)  {  // ...  // Attach CContainedWindows to OK and Exit buttons  m\_wndOKBtn.SubclassWindow (GetDlgItem(IDOK));  m\_wndExitBtn.SubclassWindow (GetDlgItem(IDCANCEL));  return TRUE;  } |

　　下面是新的WM\_SETCURSOR消息处理函数：

|  |
| --- |
| LRESULT CMainDlg::OnSetCursor\_OK (HWND hwndCtrl, UINT uHitTest, UINT uMouseMsg )  {  static HCURSOR hcur = LoadCursor ( NULL, IDC\_HAND );  if ( NULL != hcur )  {  　　 SetCursor ( hcur );  　　 return TRUE; 　　}  else  {  　　 SetMsgHandled(false); 　　　　return FALSE; 　　}  }  LRESULT CMainDlg::OnSetCursor\_Exit ( HWND hwndCtrl, UINT uHitTest, UINT uMouseMsg )  {  static HCURSOR hcur = LoadCursor ( NULL, IDC\_NO );  if ( NULL != hcur )  {  　　 SetCursor ( hcur );  　　 return TRUE;  }  else  {  　　 SetMsgHandled(false);  　　 return FALSE;  }  } |

　　如果你还想使用按钮类的特性，你需要这样声明变量：

|  |
| --- |
| CContainedWindowT<CButton> m\_wndOKBtn; |

　　这样就可以使用CButton类的方法。

　　当你把鼠标光标移到这些按钮上就可以看到WM\_SETCURSOR消息处理函数的作用结果：

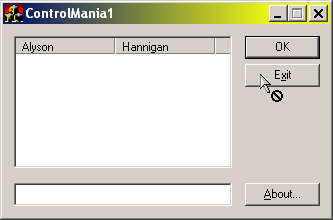
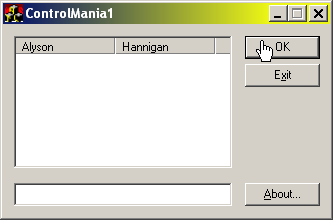


图24 用包容器窗口处理WM\_SETCURSOR消息的效果

**4.3.3、ATL 方式3 ---- 子类化(Subclassing)**

　　第三种方法创建一个CWindowImpl派生类并用它子类化一个控件。这和第二种方法有些相似，只是消息处理放在CWindowImpl类内部而不是对话框类中。

　　ControlMania1使用这种方法子类化主对话框的About按钮。下面是CButtonImpl类，他是从CWindowImpl类派生的，处理WM\_SETCURSOR消息：

|  |
| --- |
| class CButtonImpl : public CWindowImpl<CButtonImpl, CButton>  {  BEGIN\_MSG\_MAP\_EX(CButtonImpl)  　　 MSG\_WM\_SETCURSOR(OnSetCursor)  END\_MSG\_MAP()  LRESULT OnSetCursor(HWND hwndCtrl, UINT uHitTest, UINT uMouseMsg)  {  　　 static HCURSOR hcur = LoadCursor ( NULL, IDC\_SIZEALL );  　　 if ( NULL != hcur )  　　 {  　　　　 SetCursor ( hcur );  　　　　 return TRUE;  　　 }  　　 else  　　 {  　　　 　SetMsgHandled(false);  　　　 　return FALSE;  　 　}  }  }; |

　　接着在主对话框声明一个CButtonImpl成员变量：

|  |
| --- |
| class CMainDlg : public CDialogImpl<CMainDlg>  {  // ...  protected:  CContainedWindow m\_wndOKBtn, m\_wndExitBtn;  CButtonImpl m\_wndAboutBtn;  }; |

　　最后，在OnInitDialog()种子类化About按钮。

|  |
| --- |
| LRESULT CMainDlg::OnInitDialog(...)  {  // ...  // Attach CContainedWindows to OK and Exit buttons  m\_wndOKBtn.SubclassWindow ( GetDlgItem(IDOK) );  m\_wndExitBtn.SubclassWindow ( GetDlgItem(IDCANCEL) );  // CButtonImpl: subclass the About button  m\_wndAboutBtn.SubclassWindow ( GetDlgItem(ID\_APP\_ABOUT) );  return TRUE;  } |

**4.3.4、WTL 方式----动态数据交换(DDX)**

WTL的DDX(对话框数据交换)很像MFC，可以使用很简单的方法将变量和控件关联起来。首先，和前面的例子一样你需要从CWindowImpl派生一个新类，这次我们使用一个新类CEditImpl，因为这次我们使用的是Edit控件。你还需要将#include <atlddx.h> 添加到stdafx.h中，这样就可以使用DDX代码。

　　要使对话框支持DDX，需要将CWinDataExchange添加到继承列表中：

|  |
| --- |
| class CMainDlg : public CDialogImpl<CMainDlg>,  　　　　　　　public CWinDataExchange<CMainDlg>  {  //...  }; |

接着在对话框类中添加DDX链，这和MFC的类向导使用的DoDataExchange()函数功能相似。对于不同类型的数据可以使用不同的DDX宏， 我们使用DDX\_CONTROL用来连接变量和控件，这次我们使用CEditImpl处理WM\_CONTEXTMENU消息，使它能够在你右键单控件时做一些事情。

|  |
| --- |
| class CEditImpl : public CWindowImpl<CEditImpl, CEdit>  {  BEGIN\_MSG\_MAP\_EX(CEditImpl)  　　 MSG\_WM\_CONTEXTMENU(OnContextMenu)  END\_MSG\_MAP()  void OnContextMenu ( HWND hwndCtrl, CPoint ptClick )  {  　　 MessageBox("Edit control handled WM\_CONTEXTMENU");  }  };  class CMainDlg : public CDialogImpl<CMainDlg>,  　　　　　　　public CWinDataExchange<CMainDlg>  {  //...  BEGIN\_DDX\_MAP(CMainDlg)  　　 DDX\_CONTROL(IDC\_EDIT, m\_wndEdit)  END\_DDX\_MAP()  protected:  CContainedWindow m\_wndOKBtn, m\_wndExitBtn;  CButtonImpl m\_wndAboutBtn;  CEditImpl　 m\_wndEdit;  }; |

最后，在OnInitDialog()中调用DoDataExchange()函数，这个函数是继承自CWinDataExchange。 DoDataExchange()第一次被调用时完成相关控件的子类化工作，所以在这个例子中，DoDataExchange()子类化ID为 IDC\_EDIT的控件，将其与m\_wndEdit建立关联。

|  |
| --- |
| LRESULT CMainDlg::OnInitDialog(...)  {  // ...  // Attach CContainedWindows to OK and Exit buttons  m\_wndOKBtn.SubclassWindow ( GetDlgItem(IDOK) );  m\_wndExitBtn.SubclassWindow ( GetDlgItem(IDCANCEL) );  // CButtonImpl: subclass the About button  m\_wndAboutBtn.SubclassWindow ( GetDlgItem(ID\_APP\_ABOUT) );  // First DDX call, hooks up variables to controls.  DoDataExchange(false);  return TRUE;  } |

　　DoDataExchange()的参数与MFC的UpdateData()函数的参数意义相同，我会在下一节详细介绍。

　　现在运行ControlMania1程序，可以看到子类化的效果。鼠标右键单击编辑框将弹出消息框，当鼠标通过按钮上时鼠标形状会改变。

**4.3.5、DDX的详细内容**

　　当然，DDX是用来做数据交换的，WTL支持在Edit控件和字符串之间交换数据，也可以将字符串解析成数字，转换成整型或浮点型变量，还支持Check box和Radio button组的状态与int型变量之间的转换。

**4.3.6、DDX 宏**

DDX可以使用6种宏，每一种宏都对应一个CWinDataExchange类的方法支持其工作，每一种宏都用相同的形式：DDX\_FOO(控件ID, 变量)，每一种宏都可以支持多种类型的变量，例如DDX\_ TEXT的重载就支持多种类型的数据。

|  |  |
| --- | --- |
| DDX宏 | 说 明 |
| DDX\_TEXT | 在字符串和edit box控件之间传输数据，变量类型可以是CString, BSTR, CComBSTR或者静态分配的字符串数组，但是不能使用new动态分配的数组。 |
| DDX\_INT | 在edit box控件和数字变量之间传输int型数据。 |
| DDX\_UINT | 在edit box控件和数字变量之间传输无符号int型数据。 |
| DDX\_FLOAT | 在edit box控件和数字变量之间传输浮点型(float)数据或双精度型数据(double)。 |
| DDX\_CHECK | 在check box控件和int型变量之间转换check box控件的状态。 |
| DDX\_RADIO | 在radio buttons控件组和int型变量之间转换radio buttons控件组的状态。 |
| DDX\_FLOAT | 在edit box控件和数字变量之间传输float型数据。 |

　　宏有一些特殊，要使用DDX\_FLOAT宏需要在stdafx.h文件的所有WTL头文件包含之前添加一行定义：

|  |
| --- |
| #define \_ATL\_USE\_DDX\_FLOAT |

　　这个定义是必要的，因为默认状态为了优化程序的大小而不支持浮点数。

**4.3.7、有关 DoDataExchange()的详细内容**

　　调用DoDataExchange()方法和在MFC中使用UpdateData()一样，DoDataExchange()的函数原型是：

|  |
| --- |
| BOOL DoDataExchange ( BOOL bSaveAndValidate = FALSE, UINT nCtlID = (UINT)-1 ); |

参数说明：

|  |  |
| --- | --- |
| 参数 | 说 明 |
| bSaveAndValidate | 指示数据传输方向的标志。TRUE表示将数据从控件传输给变量，FALSE表示将数据从变量传输给控件。 |
| nCtlID | 使用-1可以更新所有控件，如果只想DDX宏作用于一个控件就使用控件的ID。 |

需要注意得是：参数bSaveAndValidate的默认值是FALSE，而 MFC的UpdateData()函数的默认值是TRUE。为了方便记忆，你可以使用DDX\_SAVE 和 DDX\_LOAD标号(它们分别被定义为TRUE和FALSE)。

如果控件更新成功DoDataExchange()会返回 TRUE，如果失败就返回FALSE，对话框类有两个重载函数处理数据交换错误。一个是OnDataExchangeError()，无论什么原因的错误都会调用这个函数，这个函数的默认实现在CWinDataExchange中，它仅仅是驱动PC喇叭发出一声蜂鸣，并将出错的控件设为当前焦点。另一个函数是OnDataValidateError()，但是要到本文的第五章介绍DDV时才用得到。

**4.3.8、使用DDX**

　　在CMainDlg中添加几个变量，演示DDX的使用方法。

|  |
| --- |
| class CMainDlg : public ...  {  //...  BEGIN\_DDX\_MAP(CMainDlg)  　　 DDX\_CONTROL(IDC\_EDIT, m\_wndEdit)  　　 DDX\_TEXT(IDC\_EDIT, m\_sEditContents)  　　 DDX\_INT(IDC\_EDIT, m\_nEditNumber)  END\_DDX\_MAP()  protected:  // DDX variables  CString m\_sEditContents;  int　　 m\_nEditNumber;  }; |

　　在OK按钮的处理函数中，我们首先调用DoDataExchange()将将edit控件的数据传送给我们刚刚添加的两个变量，然后将结果显示在列表控件中。

|  |
| --- |
| LRESULT CMainDlg::OnOK ( UINT uCode, int nID, HWND hWndCtl )  {  CString str;  // Transfer data from the controls to member variables.  if ( !DoDataExchange(true) )  　　 return;  m\_wndList.DeleteAllItems();  m\_wndList.InsertItem ( 0, \_T("DDX\_TEXT") );  m\_wndList.SetItemText ( 0, 1, m\_sEditContents );  str.Format ( \_T("%d"), m\_nEditNumber );  m\_wndList.InsertItem ( 1, \_T("DDX\_INT") );  m\_wndList.SetItemText ( 1, 1, str );  } |

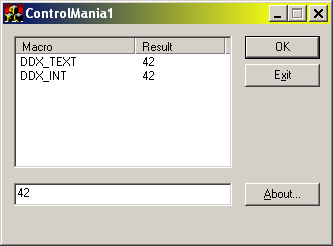


图25 使用WTL的DDX

　　如果编辑控件输入的不是数字，DDX\_INT将会失败并触发OnDataExchangeError()的调用，CMain Dlg重载了OnDataExchangeError()函数显示一个消息框：

|  |
| --- |
| void CMainDlg::OnDataExchangeError ( UINT nCtrlID, BOOL bSave )  {  CString str;  str.Format ( \_T("DDX error during exchange with control: %u"), nCtrlID );  MessageBox ( str, \_T("ControlMania1"), MB\_ICONWARNING );  ::SetFocus ( GetDlgItem(nCtrlID) );  } |

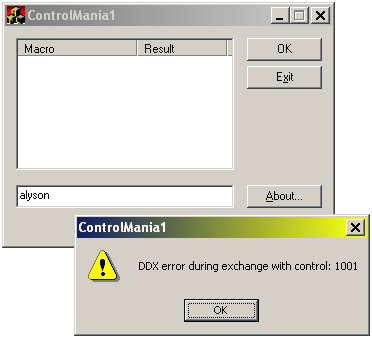


图26 DDV验证出错

作为最后一个使用DDX的例子，我们添加一个check box演示DDX\_CHECK的使用：

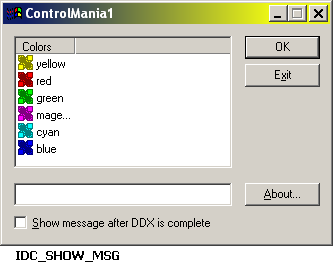


图27 演示DDX\_CHECK宏

　 　DDX\_CHECK使用的变量类型是int型，它的可能值是0，1，2，分别对应check box的未选择状态，选择状态和不确定状态。你也可以使用常量BST\_UNCHECKED，BST\_CHECKED，和 BST\_ INDETERMINATE代替，对于check box来说只有选择和未选择两种状态，你可以将其视为布尔型变量。

　　以下是为使用check box的DDX而做的改动：

|  |
| --- |
| class CMainDlg : public ...  {  //...  BEGIN\_DDX\_MAP(CMainDlg)  　　 DDX\_CONTROL(IDC\_EDIT, m\_wndEdit)  　　 DDX\_TEXT(IDC\_EDIT, m\_sEditContents)  　　 DDX\_INT(IDC\_EDIT, m\_nEditNumber)  　　 DDX\_CHECK(IDC\_SHOW\_MSG, m\_nShowMsg)  END\_DDX\_MAP()  protected:  // DDX variables  CString m\_sEditContents;  int　　 m\_nEditNumber;  int　　 m\_nShowMsg;  }; |

　　在OnOK()的最后，检查m\_nShowMsg的值看看check box是否被选中。

|  |
| --- |
| void CMainDlg::OnOK ( UINT uCode, int nID, HWND hWndCtl )  {  // Transfer data from the controls to member variables. 　　if ( !DoDataExchange(true) )  　　 return;  //...  if ( m\_nShowMsg )  　　 MessageBox ( \_T("DDX complete!"), \_T("ControlMania1"), MB\_ICONINFORMATION );  } |

　　使用其它DDX\_\*宏的例子代码包含在例子工程中。

**4.3.9、处理控件发送的通知消息**

在WTL中处理通知消息与使用API方式编程相似，控件以WM\_COMMAND 或 WM\_NOTIFY 消息的方式向父窗口发送通知事件，父窗口相应并做相应处理。少数其它的消息也可以看作是通知消息，例如：WM\_DRAWITEM，当一个自画控件需要画自己时就会发送这个消息，父窗口可以自己处理这个消息，也可以再将它反射给控件，MFC采用得就是消息反射方式，使得控件能够自己处理通知消息，提高了代码 的封装性和可重用性。

**4.3.10、在父窗口中响应控件的通知消息**

以WM\_NOTIFY 和WM\_COMMAND消息形式发送的通知消息包含各种信息。WM\_ COMMAND消息的参数包含发送通知消息的控件ID，控件的窗口句柄和通知代码， WM\_NOTIFY消息的参数还包含一个NMHDR数据结构的指针。ATL和WTL有各种消息映射宏用来处理这些通知消息，我在这里只介绍WTL宏，因为本文就是讲WTL的。使用这些宏需要在消息映射链中使用BEGIN\_MSG\_MAP\_EX，并包含atlcrack.h文件。

**4.3.11、消息映射宏**

　　要处理WM\_COMMAND通知消息需要使用COMMAND\_HANDLER\_EX宏：

|  |  |
| --- | --- |
| 宏 | 描 述 |
| COMMAND\_HANDLER\_EX(id, code, func) | 处理从某个控件发送得某个通知代码。 |
| COMMAND\_ID\_HANDLER\_EX(id, func) | 处理从某个控件发送得所有通知代码。 |
| COMMAND\_CODE\_HANDLER\_EX(code, func) | 处理某个通知代码得所有消息，不管是从那个控件发出的。 |
| COMMAND\_RANGE\_HANDLER\_EX(idFirst, idLast, func) | 处理ID在idFirst和idLast之间得控件发送的所有通知代码。 |
| COMMAND\_RANGE\_CODE\_HANDLER\_EX(idFirst, idLast, code, func) | 处理ID在idFirst和idLast之间得控件发送的某个通知代码。 |

例子：

|  |  |
| --- | --- |
| 例子 | 说明 |
| COMMAND\_HANDLER\_EX(IDC\_USERNAME, EN\_CHANGE, OnUsernameChange) | 处理从ID是IDC\_USERNAME的edit box控件发出的EN\_CHANGE通知消息。 |
| COMMAND\_ID\_HANDLER\_EX(IDOK, OnOK) | 处理ID是IDOK的控件发送的所有通知消息。 |
| COMMAND\_RANGE\_CODE\_HANDLER\_EX  (IDC\_MONDAY, IDC\_FRIDAY, BN\_CLICKED, OnDayClicked) | 处理ID在IDC\_MONDAY和IDC\_FRIDAY之间控件发送的BN\_CLICKED通知消息。 |

还有一些宏专门处理WM\_NOTIFY消息，和上面的宏功能类似，只是它们的名字开头以“NOTIFY \_”代替“COMMAND\_”。

　　WM\_COMMAND 消息处理函数的原型是：

|  |
| --- |
| void func ( UINT uCode, int nCtrlID, HWND hwndCtrl ); |

WM\_COMMAND通知消息不需要返回值，所以处理函数也不需要返回值。

WM\_NOTIFY消息处理函数的原型是：

|  |
| --- |
| LRESULT func ( NMHDR\* phdr ); |

消息处理函数的返回值用作消息相应的返回值，这不同于MFC，MFC的消息响应通过消息处理函数的LRESULT\*参数得到返回值。发送通知消息的控件的窗口句柄和通知代码包含在NMHDR结构中，分别是code和hendFrom成员。与MFC一样的是，如果通知消息发送的不是普通的NMHDR结构，你的消息处理函数应该将phdr参数转换成正确的类型。

　　我们将为CMainDlg添加LVN\_ITEMCHANGED通知的处理函数，处理从list控件发出的这个通知，在对话框中显示当前选择的项目，先从添加消息映射宏和消息处理函数开始：

|  |
| --- |
| class CMainDlg : public ...  {  BEGIN\_MSG\_MAP\_EX(CMainDlg)  　　 NOTIFY\_HANDLER\_EX(IDC\_LIST, LVN\_ITEMCHANGED, OnListItemchanged)  END\_MSG\_MAP()  LRESULT OnListItemchanged(NMHDR\* phdr);  //...  }; |

　　下面是消息处理函数：

|  |
| --- |
| LRESULT CMainDlg::OnListItemchanged ( NMHDR\* phdr )  {  NMLISTVIEW\* pnmlv = (NMLISTVIEW\*) phdr;  int nSelItem = m\_wndList.GetSelectedIndex();  CString sMsg;  // If no item is selected, show "none". Otherwise, show its index.  if ( -1 == nSelItem )  　　 sMsg = \_T("(none)");  else  　　 sMsg.Format ( \_T("%d"), nSelItem );  SetDlgItemText ( IDC\_SEL\_ITEM, sMsg );  return 0;　 // retval ignored  } |

　　该处理函数并未用到phdr参数，我将他强制转换成NMLISTVIEW\*只是为了演示用法。

**4.3.12、反射通知消息**

　　如果你是用CWindowImpl的派生类封装控件，比如前面使用的CEditImpl，你可以在类的内部处理通知消息，而不是在对话框中，这就是通知消息的反射，它和MFC的消息反射相似。不同的是在WTL中父窗口和控件都可以处理通知消息，而在MFC中只有控件能处理通知消息（除非你重载 WindowProc函数，在MFC反射这些消息之前截获它们）。

　　如果需要将通知消息反射给控件封装类，只需在对话框的消息映射链中添加**REFLECT\_ NOTIFICATIONS()**宏：

|  |
| --- |
| class CMainDlg : public ...  {  public:  BEGIN\_MSG\_MAP\_EX(CMainDlg)  　　 //...  　　 NOTIFY\_HANDLER\_EX(IDC\_LIST, LVN\_ITEMCHANGED, OnListItemchanged)  **REFLECT\_NOTIFICATIONS()**  END\_MSG\_MAP()  }; |

这个宏向消息映射链添加了一些代码处理那些未被前面的宏处理的通知消息，它检查消息传递的HWND窗口句柄是否有效并将消息转发给这个窗口，当然，消息代码的数值被改变成OLE控件所使用的值，OLE控件有与之相似的消息反射系统。新的消息代码值用OCM\_xxx代替了WM\_xxx，但是消息的处理方式和未反射前一样。

有18种被反射的消息：

|  |  |
| --- | --- |
| 消息类型 | 消息 |
| 控件通知消息 | WM\_COMMAND, WM\_NOTIFY, WM\_PARENTNOTIFY |
| 自画消息 | WM\_DRAWITEM, WM\_MEASUREITEM, WM\_COMPAREITEM, WM\_DELETEITEM |
| List box 键盘消息 | WM\_VKEYTOITEM, WM\_CHARTOITEM |
| 其它 | WM\_HSCROLL, WM\_VSCROLL, WM\_CTLCOLOR\* |

在你想添加反射消息处理的控件类内，不要忘了使用**DEFAULT\_REFLECTION\_HANDLER(**)宏， DEFAULT\_REFLECTION\_HANDLER()宏确保将未被处理的消息交给DefWindowProc()正确处理。 下面的例子是一个自画按钮类，它相应了从父窗口反射的WM\_DRAWITEM消息。

|  |
| --- |
| class CODButtonImpl : public CWindowImpl<CODButtonImpl, CButton>  {  public:  BEGIN\_MSG\_MAP\_EX(CODButtonImpl)  　　 MSG\_OCM\_DRAWITEM(OnDrawItem)  　　 DEFAULT\_REFLECTION\_HANDLER()  END\_MSG\_MAP()  void OnDrawItem ( UINT idCtrl, LPDRAWITEMSTRUCT lpdis )  {  　　 // do drawing here...  }  }; |

**4.3.13、用来处理反射消息的WTL宏**

我们现在只看到了WTL的消息反射宏中的一个：MSG\_OCM\_DRAWITEM，还有17个这样的反射宏。由于WM\_NOTIFY和 WM\_COMMAND消息带的参数需要展开，WTL提供了特殊的宏MSG\_ OCM\_COMMAND和MSG\_OCM\_NOTIFY做这些事情。这些宏所作的工作与COMMAND\_ ANDLER \_EX和NOTIFY\_HANDLER\_EX宏相同，只是前面加了“REFLECTED\_”，例如，一个树控件类可能存在这样的消息映射链：

|  |
| --- |
| class CMyTreeCtrl : public CWindowImpl<CMyTreeCtrl, CTreeViewCtrl>  {  public:  BEGIN\_MSG\_MAP\_EX(CMyTreeCtrl)  REFLECTED\_NOTIFY\_CODE\_HANDLER\_EX(TVN\_ITEMEXPANDING, OnItemExpanding)  　　 DEFAULT\_REFLECTION\_HANDLER()  END\_MSG\_MAP()  LRESULT OnItemExpanding ( NMHDR\* phdr );  }; |

在ControlMania1对话框中用了一个树控件，和上面的代码一样处理TVN\_ITEMEXPANDING消息，CMainDlg类的成员 m\_wndTree使用DDX连接到控件上，CMainDlg反射通知消息，树控件的处理函数OnItemExpanding()是这样的：

|  |
| --- |
| LRESULT CBuffyTreeCtrl::OnItemExpanding ( NMHDR\* phdr )  {  NMTREEVIEW\* pnmtv = (NMTREEVIEW\*) phdr;  if ( pnmtv->action & TVE\_COLLAPSE )  　　 return TRUE;　　// don''t allow it  else  　　 return FALSE;　 // allow it  } |

　　运行ControlMania1，用鼠标点击树控件上的+/-按钮，你就会看到消息处理函数的作用－节点展开后就不能再折叠起来。

**4.3.14、容易出错和混淆的地方: 对话框的字体**

　　如果你像我一样对界面非常讲究，并且正在使用windows 2000或XP，你就会奇怪为什么对话框使用MS Sans Serif字体而不是Tahoma字体，因为VC6太老了，它生成的资源文件在NT 4上工作的很好，但是对于新的版本就会有问题。你可以自己修改，需要手工编辑资源文件，据我所知VC 7以止版本不存在这个问题。

在资源文件中对话框的入口处需要修改3个地方：

|  |  |
| --- | --- |
| 待修改处 | 修改内容 |
| 对话框类型 | 将DIALOG改为DIALOGEX |
| 窗口类型 | 添加DS\_SHELLFONT |
| 对话框字体 | 将MS Sans Serif改为MS Shell Dlg |

不幸的是前两个修改会在每次保存资源文件时丢失(被VC又改回原样)，所以需要重复这些修改，下面是改动之前的代码：

|  |
| --- |
| IDD\_ABOUTBOX DIALOG DISCARDABLE　0, 0, 187, 102  STYLE DS\_MODALFRAME | WS\_POPUP | WS\_CAPTION | WS\_SYSMENU  CAPTION "About"  FONT 8, "MS Sans Serif"  BEGIN  　...  END |

　　这是改动之后的代码：

|  |
| --- |
| IDD\_ABOUTBOX DIALOGEX DISCARDABLE　0, 0, 187, 102  STYLE DS\_SHELLFONT | DS\_MODALFRAME | WS\_POPUP | WS\_CAPTION | WS\_SYSMENU  CAPTION "About"  FONT 8, "MS Shell Dlg"  BEGIN  　...  END |

这样改了之后，对话框将在新的操作系统上使用Tahoma字体，而在老的操作系统上仍旧使用MS Sans Serif字体。

**4.3.15、\_ATL\_MIN\_CRT**

本文的论坛 FAQ已经做过解释，ATL包含的优化设置让你创建一个不使用C运行库(CRT)的程序，使用这个优化需要在预处理设置中添加\_ATL\_MIN\_CRT标号，向导生成的代码在 Release配置中默认使用了这个优化。由于我写程序总是会用到CRT函数，所以我总是去掉这个标号，如果你在CString类或DDX中用到了浮点运算特性，你也要去掉这个标号。

# 第五章 高级对话框用户界面类

　　在上一章我们介绍了一些与对话框和控件有关的WTL的特性，它们和MFC相应类的作用相同。本文将介绍一些新类，用来实现高级界面特性：控件自画和自定控件外观，新的WTL控件，UI更新和对话框数据验证(DDV)。

**5.1、特别的自画和外观定制类**

由于自画和定制控件外观在图形用户界面中是很常用的手段，所以WTL提供了几个嵌入类来完成这些令人厌烦的工作。我接着就会介绍它们，事实上我们在上一个例子工程ControlMania2的结尾部分已经这么做了。如果你正随着我的讲解，用应用程序向导创建新工程，请不要忘了使用无模式对话框。为了使正常工作必须使用无模式对话框，我会在对话框中控件的UI 更新部分详细解释为什么这样做。

**5.1.1、COwnerDraw**

　　控件的自画需要响应四个消息：WM\_MEASUREITEM, WM\_DRAWITEM, WM\_COMPAREITEM 和WM\_DELETEITEM，在**atlframe.h**头文件中定义的COwnerDraw类可以简化这些工作，使用这个类就不需要处理这四个消息，你只需将消息链入COwnerDraw，它会调用你的类中的重载函数。

　　如何将消息链入COwnerDraw取决于：你是否将消息反射给控件，两种方法有些不同。下面是COwnerDraw类的消息映射链，它使得两种方法的差别更加明显：

|  |
| --- |
| template <class T>  class COwnerDraw  {  public:  BEGIN\_MSG\_MAP(COwnerDraw<T>)  　　 MESSAGE\_HANDLER(WM\_DRAWITEM, OnDrawItem)  　　 MESSAGE\_HANDLER(WM\_MEASUREITEM, OnMeasureItem)  　　 MESSAGE\_HANDLER(WM\_COMPAREITEM, OnCompareItem)  　　 MESSAGE\_HANDLER(WM\_DELETEITEM, OnDeleteItem)  ALT\_MSG\_MAP(1)  MESSAGE\_HANDLER(OCM\_DRAWITEM, OnDrawItem)  MESSAGE\_HANDLER(OCM\_MEASUREITEM, OnMeasureItem)  MESSAGE\_HANDLER(OCM\_COMPAREITEM, OnCompareItem)  MESSAGE\_HANDLER(OCM\_DELETEITEM, OnDeleteItem)  END\_MSG\_MAP()  }; |

　　注意，消息映射链的主要部分处理WM\_\*消息，而ALT部分处理反射的消息OCM\_\*。自画的通知消息就像WM\_NOTIFY消息一样，你可以在父窗口处理它们，也可以将它们反射回控件，如果你使用前一种方法，消息被直接链入COwnerDraw：

|  |
| --- |
| class CSomeDlg : public COwnerDraw<CSomeDlg>, ...  {  BEGIN\_MSG\_MAP(CSomeDlg)  //...  CHAIN\_MSG\_MAP(COwnerDraw<CSomeDlg>)  END\_MSG\_MAP()  　 void DrawItem ( LPDRAWITEMSTRUCT lpdis );  }; |

　　当然，如果你想要控件自己处理这些消息，你需要使用CHAIN\_MSG\_MAP\_ALT宏将消息链入ALT\_MSG\_MAP(1)部分：

|  |
| --- |
| class CSomeButtonImpl : public COwnerDraw<CSomeButtonImpl>, ...  {  BEGIN\_MSG\_MAP(CSomeButtonImpl)  　　 //...  　　 CHAIN\_MSG\_MAP\_ALT(COwnerDraw<CSomeButtonImpl>, 1)  DEFAULT\_REFLECTION\_HANDLER()  END\_MSG\_MAP()  void DrawItem ( LPDRAWITEMSTRUCT lpdis );  }; |

COwnerDraw类将对消息传递的参数展开，然后调用你的类中的实现函数。上面的例子中，我们自己的类实现DrawItem()函数，当有WM\_DRAWITEM或OCM\_DRAWITEM消息被链入COwner Draw时，这个函数就会被调用。你可以重载的方法有：

|  |
| --- |
| void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);  void MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct);  int　CompareItem(LPCOMPAREITEMSTRUCT lpCompareItemStruct);  void DeleteItem(LPDELETEITEMSTRUCT lpDeleteItemStruct); |

如果你不想处理某个消息，你可以调用SetMsgHandled(false)，消息会被传递给消息映射链中的其他响应者。SetMsgHandled ()事实上是COwnerDraw类的成员函数，但是它的作用和在BEGIN\_MSG \_ MAP\_EX()中使用SetMsgHandled()一样。

　　对于ControlMania2，它从ControlMania1中的树控件开始，添加了自画按钮处理反射的WM\_ DRAWITEM消息，下面是资源编辑器中的新按钮：

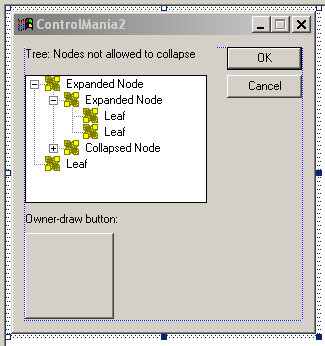


图28 添加自画按钮

　　现在我们需要新类来实现自画按钮：

|  |
| --- |
| class CODButtonImpl : public CWindowImpl<CODButtonImpl, CButton>,  public COwnerDraw<CODButtonImpl>  {  public:  BEGIN\_MSG\_MAP\_EX(CODButtonImpl)  　　 CHAIN\_MSG\_MAP\_ALT(COwnerDraw<CODButtonImpl>, 1)  　　 DEFAULT\_REFLECTION\_HANDLER()  END\_MSG\_MAP()  void DrawItem ( LPDRAWITEMSTRUCT lpdis );  }; |

　　DrawItem()使用了像BitBlt()这样的GDI函数向按钮的表面画位图，代码应该很容易理解，因为WTL使用的类名和函数名都和MFC类似。

|  |
| --- |
| void CODButtonImpl::DrawItem ( LPDRAWITEMSTRUCT lpdis )  {  // NOTE: m\_bmp is a CBitmap init''ed in the constructor.  CDCHandle dc = lpdis->hDC;  CDC dcMem;  dcMem.CreateCompatibleDC ( dc );  dc.SaveDC();  dcMem.SaveDC();  // Draw the button''s background, red if it has the focus, blue if not.  if ( lpdis->itemState & ODS\_FOCUS )  　　 dc.FillSolidRect ( &lpdis->rcItem, RGB(255,0,0) );  else  　　 dc.FillSolidRect ( &lpdis->rcItem, RGB(0,0,255) );  // Draw the bitmap in the top-left, or offset by 1 pixel if the button is clicked.  dcMem.SelectBitmap ( m\_bmp );  if ( lpdis->itemState & ODS\_SELECTED )  　　 dc.BitBlt ( 1, 1, 80, 80, dcMem, 0, 0, SRCCOPY );  else  　　 dc.BitBlt ( 0, 0, 80, 80, dcMem, 0, 0, SRCCOPY );  dcMem.RestoreDC(-1);  dc.RestoreDC(-1);  } |

　　我们的按钮看起来是这个样子：

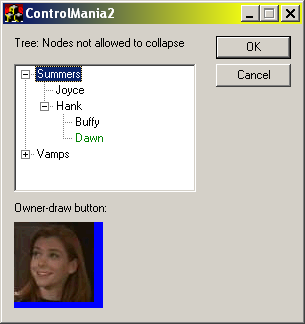


图29 自画的位图按钮

**5.1.2、CCustomDraw**

　　CCustomDraw类使用和COwnerDraw类相同的方法处理NM\_CUSTOMDRAW消息，对于自定绘制的每个阶段都有相应的重载函数：

|  |
| --- |
| DWORD OnPrePaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD);  DWORD OnPostPaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD);  DWORD OnPreErase(int idCtrl, LPNMCUSTOMDRAW lpNMCD);  DWORD OnPostErase(int idCtrl, LPNMCUSTOMDRAW lpNMCD);  DWORD OnItemPrePaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD);  DWORD OnItemPostPaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD);  DWORD OnItemPreErase(int idCtrl, LPNMCUSTOMDRAW lpNMCD);  DWORD OnItemPostEraset(int idCtrl, LPNMCUSTOMDRAW lpNMCD);  DWORD OnSubItemPrePaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD); |

　　这些函数默认都是返回CDRF\_DODEFAULT，如果想自画控件或返回一个不同的值，就需要重载这些函数：

你可能注意到上面的屏幕截图将“道恩”(Dawn：女名)显示成绿色，这是因为CBuffyTreeCtrl将消息链入CCustomDraw并重载了OnPrePaint()和OnItemPrePaint()方法。向树控件中添加节点时，节点的itemdata字段被设置成1，OnItemPrePaint()检查这个值，然后改变文字的颜色。

|  |
| --- |
| DWORD CBuffyTreeCtrl::OnPrePaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD)  {  return CDRF\_NOTIFYITEMDRAW;  }  DWORD CBuffyTreeCtrl::OnItemPrePaint(int idCtrl,LPNMCUSTOMDRAW lpNMCD)  {  if ( 1 == lpNMCD->lItemlParam )  　　 pnmtv->clrText = RGB(0,128,0);  return CDRF\_DODEFAULT; } |

　　CCustomDraw类也有SetMsgHandled()函数，你可以像在COwnerDraw类那样使用这个函数。

**5.2、WTL的新控件**

　　WTL有几个新控件，它们要么是其他封装类的扩展(像 CTreeViewCtrlEx)，要么是提供Windows标准控件没有的新功能(像 CHyperLink)。

**5.2.1、CBitmapButton类**

　 　WTL的CBitmapButton类声明在atlctrlx.h中，它比MFC的同名类使用起来要简单的多。WTL的CBitmapButton类使用图像列表而不是单个的位图资源，你可以将多个按钮的图像放到一个位图文件中，减少GDI资源的占用。这对于使用很多图片，并需要在Windows 9X系统上运行的程序很有好处，因为使用太多的单个位图将会很快耗尽GDI资源并导致系统崩溃。

　　CBitmapButton是一个CWindowImpl派生类，它有很多特色：自动调整控件的大小，自动生成3D边框，支持热点跟踪，每个按钮可以使用多个图像分别表示按钮的不同状态。

在ControlMania2中，我们对前面的例子创建的自画按钮使用CBitmapButton类。现在CMainDlg对话框类中添加 CBitmapButton类型的变量m\_wndBmpBtn，调用SubclassWindow()函数或使用DDX将其和控件联系起来，将位图装载到图像列表，并告诉按钮使用该图像列表，还要告诉按钮每个图像分别对应按钮的什么状态。下面是OnInitDialog()函数中建立和使用这个按钮的代码段：

|  |
| --- |
| // Set up the bitmap button  CImageList iml;  iml.CreateFromImage(IDB\_ALYSON\_IMGLIST,81,1, CLR\_NONE,IMAGE\_BITMAP, LR\_CREATEDIBSECTION );  m\_wndBmpBtn.SubclassWindow ( GetDlgItem(IDC\_ALYSON\_BMPBTN) );  m\_wndBmpBtn.SetToolTipText ( \_T("Alyson") );  m\_wndBmpBtn.SetImageList ( iml );  m\_wndBmpBtn.SetImages ( 0, 1, 2, 3 ); |

默认情况下，按钮只是引用图像列表，所以OnInitDialog()不能删除它所创建的图像列表。下面显示的是新按钮的一般状态，注意控件是如何根据图像的大小来调整自己的大小。

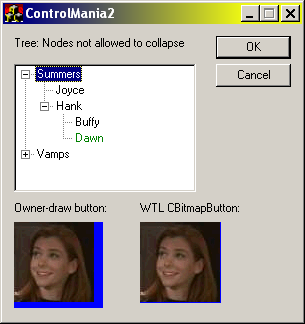


图30 使用WTL的CBitmapButton类

　　因为CBitmapButton是一个非常有用的类，我想介绍一下它的公有方法。

**5.2.1.1、方法**

　　CBitmapButtonImpl类包含了实现一个按钮的所有代码，除非你想重载某个方法或消息处理，你可以对控件直接使用CBitmapButton类。

**5.2.1.1.1、构造函数**

|  |
| --- |
| CBitmapButtonImpl(DWORD dwExtendedStyle = BMPBTN\_AUTOSIZE,HIMAGELIST hImageList = NULL) |

　　构造函数可以指定按钮的扩展样式(这与窗口的样式不冲突)和图像列表，通常使用默认参数就足够了，因为可以使用其他的方法设定这些属性。

**5.2.1.1.2、窗口对象子类化**

|  |
| --- |
| BOOL SubclassWindow(HWND hWnd) |

　　SubclassWindow()是个重载函数，主要完成控件的子类化和初始化控件类保有的内部数据。

**5.2.1.1.3、位图按钮扩展风格**

|  |
| --- |
| DWORD GetBitmapButtonExtendedStyle()  DWORD SetBitmapButtonExtendedStyle(DWORD dwExtendedStyle, DWORD dwMask = 0) |

CBitmapButton支持一些扩展样式，这些扩展样式会对按钮的外观和操作方式产生影响：

|  |  |
| --- | --- |
| 风格 | 说明 |
| BMPBTN\_HOVER | 使用热点跟踪，当鼠标移到按钮上时按钮被画成焦点状态 |
| BMPBTN\_AUTO3D\_SINGLE, BMPBTN\_AUTO3D\_DOUBLE | 在按钮图像周围自动产生一个三维边框，当按钮拥有焦点时会显示一个表示焦点的虚线矩形框。另外如果你没有指定按钮按下状态的图像，将会自动生成一个。**BMPBTN\_AUTO3D\_DOUBLE**样式生成的边框稍微粗一些，其他特征和BMPBTN\_AUTO3D\_SINGLE一样 |
| BMPBTN\_AUTOSIZE | 按钮调整自己的大小以适应图像大小，这是默认样式 |
| MPBTN\_SHAREIMAGELISTS | 如果指定这个样式，按钮不负责销毁按钮使用的图像列表，如果不使用这个样式，析构函数会销毁按钮使用的图像列表 |
| BMPBTN\_AUTOFIRE | 如果设置这个样式，在按钮上按住鼠标左键不放将会产生连续的WM\_COM MAND消息 |

　　调用SetBitmapButtonExtendedStyle()时，dwMask参数控制着那个样式将被改变，默认值是0，意味着用新样式完全替换旧的样式。

**5.2.1.1.4、图像列表管理**

|  |
| --- |
| HIMAGELIST GetImageList()  HIMAGELIST SetImageList(HIMAGELIST hImageList) |

调用SetImageList()设置按钮使用的图像列表。

**5.2.1.1.5、工具提示管理**

|  |
| --- |
| int　GetToolTipTextLength()  bool GetToolTipText(LPTSTR lpstrText, int nLength)  bool SetToolTipText(LPCTSTR lpstrText) |

　　CBitmapButton支持显示工具提示(tooltip)，调用SetToolTipText()指定显示的文字。

**5.2.1.1.6、设置使用的图像**

|  |
| --- |
| void SetImages(int nNormal, int nPushed = -1,int nFocusOrHover = -1, int nDisabled = -1) |

　　调用SetImages()函数告诉按钮分别使用图像列表的哪一个图像表示哪个状态。nNormal是必须的，其它是可选的，使用－1表示对应的状态没有图像。

**5.2.2、CCheckListViewCtrl类**

　 　CCheckListViewCtrl类在atlctrlx.h中定义，是一个CWindowImpl派生类，实现了一个带检查框的list view控件。它和MFC的CCheckListBox不同，CCheckListBox只是一个list box，不是list view。CCheckListViewCtrl类非常简单，只添加了很少的函数，当然，它使用了一个新的辅助类 CCheckList ViewCtrlImplTraits，它和CWinTraits类的作用类似，只是第三个参数是list view控件的扩展样式属性，如果你没有定义自己的CCheckListViewCtrlImplTraits，它将使用默认的样式：

|  |
| --- |
| LVS\_EX\_CHECKBOXES | LVS\_EX\_FULLROWSELECT。 |

　　下面是一个定义list view扩展样式属性的例子，加入了一个使用这个样式的新类（注意，扩展属性必须包含LVS\_EX\_CHECKBOXES，否则会因起断言错误消息）。

|  |
| --- |
| typedef CCheckListViewCtrlImplTraits<WS\_CHILD | WS\_VISIBLE | LVS\_REPORT,  WS\_EX\_CLIENTEDGE,LVS\_EX\_CHECKBOXES | LVS\_EX\_GRIDLINES | LVS\_EX\_UNDERLINEHOT |  LVS\_EX\_ONECLICKACTIVATE> CMyCheckListTraits;  class CMyCheckListCtrl : public CCheckListViewCtrlImpl<CMyCheckListCtrl, CListViewCtrl,CMyCheckListTraits>  {  private:  typedef CCheckListViewCtrlImpl<CMyCheckListCtrl, CListViewCtrl, CMyCheckListTraits> baseClass;  public:  BEGIN\_MSG\_MAP(CMyCheckListCtrl)  　　 CHAIN\_MSG\_MAP(baseClass)  END\_MSG\_MAP()  }; |

**5.2.2.1方法**

**5.2.2.1.1、子类化窗口**

　　当子类化一个已经存在的list view控件时，SubclassWindow()查看CCheckListViewCtrlImplTraits的扩展样式属性并将之应用到控件上。未用到前两个参数(窗口样式和扩展窗口样式)。

**5.2.2.1.2、SetCheckState() 和GetCheckState()**

　　这些方法实际上是在CListViewCtrl中，SetCheckState()使用行的索引和一个布尔类型参数，该布尔参数的值表示是否check这一行。GetCheckState()以行索引为参数，返回该行的checked状态。

**5.2.2.1.3、CheckSelectedItems()**

这个方法使用item的索引作为参数，它翻转这个item的check状态，这个item必须是被选定的，同时还将其他所有被选择的item设置成相应状态(译者加：多选状态下)。你大概不会用到这个方法，因为CCheckListViewCtrl会在check box被单击或用户按下了空格键时设置相应的item的状态。

　　下面是ControlMania2中的CCheckListViewCtrl的样子：

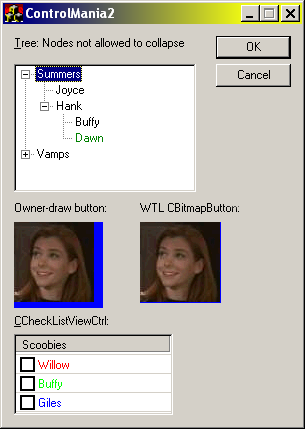


图31 演示CCheckListViewCtrl类

**5.2.3、CTreeViewCtrlEx 类和CTreeItem类**

有两个类使得树控件的使用简化了很多：CTreeItem类封装了HTREEITEM，一个CTreeItem对象含有一个HTREEITEM和一个指向包含这个HTREEITEM的树控件的指针，使你不必每次调用都引用树控件；CTreeViewCtrlEx和CTreeViewCtrl一样，只是它的方法操作CTreeItem而不是HTREE ITEM。例如，InsertItem()函数返回一个CTreeItem而不是HTREEITEM，你可以使用CTreeItem操作新添加的item。下面是一个例子：

|  |
| --- |
| // Using plain HTREEITEMs:  HTREEITEM hti, hti2;  hti = m\_wndTree.InsertItem ( "foo", TVI\_ROOT, TVI\_LAST );  hti2 = m\_wndTree.InsertItem ( "bar", hti, TVI\_LAST );  m\_wndTree.SetItemData ( hti2, 100 );  // Using CTreeItems:  CTreeItem ti, ti2;  **ti = m\_wndTreeEx.InsertItem ( "foo", TVI\_ROOT, TVI\_LAST );**  **ti2 = ti.AddTail ( "bar", 0 );**  **ti2.SetData ( 100 );** |

　 　CTreeViewCtrl对HTREEITEM的每一个操作，CTreeItem都有与之对应的方法，正像每一个关于HWND的API都有一个 CWindow方法与之对应一样。查看ControlMania2的代码可以看到更多的CTreeViewCtrlEx和CTreeItem类的方法的演示。

**5.2.4、CHyperLink类**

　　CHyperLink是一个CWindowImpl派生类，它子类化一个静态编辑控件，使之变成可点击的超链接。CHyperLink根据用户的IE使用的颜色画链接对象，还支持键盘导航。CHyperLink类的构造函数没有参数，下面是其它的公有方法。

**5.2.4.1、方法**

　　CHyperLinkImpl类内含实现一个超链接的全部代码，如果不需要重载它的方法或处理消息的话，你可以直接使用CHyperLink类。

**5.2.4.1.1、子类化窗口**

|  |
| --- |
| BOOL SubclassWindow(HWND hWnd) |

　　重载函数SubclassWindow()完成控件子类化，然后初始化该类保有的内部数据。

**5.2.4.1.2、文字标签管理**

|  |
| --- |
| bool GetLabel(LPTSTR lpstrBuffer, int nLength)  bool SetLabel(LPCTSTR lpstrLabel) |

　　获得或设置控件显示的文字，如果不指定显示文字，控件会显示资源编辑器指定给控件的静态字符串。

**5.2.4.1.3、超链接管理**

|  |
| --- |
| bool GetHyperLink(LPTSTR lpstrBuffer, int nLength)  bool SetHyperLink(LPCTSTR lpstrLink) |

　　获得或设置控件关联超链接的URL，如果不指定超链接URL，控件会使用显示的文字字符串作为URL。

**5.2.4.1.4、导航**

|  |
| --- |
| bool Navigate() |

　　导航到当前超链接的URL，该URL或者是由SetHyperLink()函数指定的URL，或者就是控件的窗口文字。

**5.2.4.1.5、工具提示管理**

没有公开的方法设置工具提示，所以需要直接使用CToolTipCtrl成员m\_tip。

下图显示的就是ControlMania2对话框中的超链接控件：

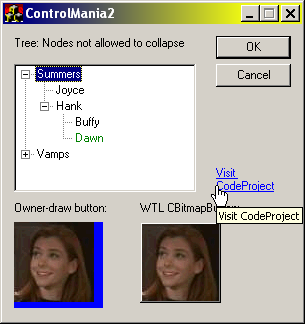


图32 演示CHyperLink类

　　在OnInitDialog()函数中设置URL：

|  |
| --- |
| **m\_wndLink.SetHyperLink ( \_T("http://www.codeproject.com/") );** |

**5.3、对话框中控件的UI更新**

　　对话框中的的UI更新控制比MFC中简单得多。在MFC中，你需要响应未公开的WM\_KICKIDLE消息，处理这个消息并触发控件的更新，在WTL中，没有这个设计，不过向导存在一个BUG，需要手工添加一行代码解决这个问题。

　 首先需要记住的是对话框必须是无模式的，因为CUpdateUI需要在程序的消息循环控制下工作。如果对话框是模式的，系统处理消息循环，我们程序的空闲处理函数就不会被调用，由于CUpdateUI是在空闲时间工作的，所以没有空闲处理就没有UI更新。

　　ControlMania2的对话框是非模式的，类定义的开始部分很像是一个框架窗口类：

|  |
| --- |
| class CMainDlg : public CDialogImpl<CMainDlg>, public CUpdateUI<CMainDlg>,  　　　　　　　　 public CMessageFilter, public CIdleHandler  {  public:  enum { IDD = IDD\_MAINDLG };  virtual BOOL PreTranslateMessage(MSG\* pMsg);  virtual BOOL OnIdle();  BEGIN\_MSG\_MAP\_EX(CMainDlg)  　　 MSG\_WM\_INITDIALOG(OnInitDialog)  　　 COMMAND\_ID\_HANDLER\_EX(IDOK, OnOK)  　　 COMMAND\_ID\_HANDLER\_EX(IDCANCEL, OnCancel)  　　 COMMAND\_ID\_HANDLER\_EX(IDC\_ALYSON\_BTN, OnAlysonODBtn)  END\_MSG\_MAP()  BEGIN\_UPDATE\_UI\_MAP(CMainDlg)  END\_UPDATE\_UI\_MAP()  //...  }; |

　　注意CMainDlg类从CUpdateUI派生并含有一个更新UI链。OnInitDialog()做了这些工作，这和前面介绍的框架窗口中的代码很相似：

|  |
| --- |
| // register object for message filtering and idle updates  CMessageLoop\* pLoop = \_Module.GetMessageLoop();  ATLASSERT(pLoop != NULL);  pLoop->AddMessageFilter(this);  pLoop->AddIdleHandler(this);  UIAddChildWindowContainer(m\_hWnd); |

只是这次我们不是调用UIAddToolbar()或UIAddStatusBar()，而是调用UIAddChildWindow Container ()，它告诉CUpdateUI我们的对话框含有需要更新的子窗口，只要看看OnIdle()，你会怀疑少了写什么：

|  |
| --- |
| BOOL CMainDlg::OnIdle()  {  return FALSE;  } |

　　你可能猜想这里应该调用另一个CUpdateUI的方法做一些实在的更新工作，你是对的，应该是这样的，向导在OnIdle()中漏掉了一行代码，现在加上：

|  |
| --- |
| BOOL CMainDlg::OnIdle()  {  **UIUpdateChildWindows();**  return FALSE;  } |

　　为了演示UI更新，我们设定鼠标点击左边的位图按钮，使得右边的按钮变得可用或禁用。先在update UI链中添加一个消息入口，使用UPDUI\_CHILDWINDOW标志表示此入口是子窗口类型：

|  |
| --- |
| BEGIN\_UPDATE\_UI\_MAP(CMainDlg)  **UPDATE\_ELEMENT(IDC\_ALYSON\_BMPBTN, UPDUI\_CHILDWINDOW)**  END\_UPDATE\_UI\_MAP() |

　　在左边的按钮的单击事件处理中，我们调用UIEnable()来翻转另一个按钮的使能状态：

|  |
| --- |
| void CMainDlg::OnAlysonODBtn ( UINT uCode, int nID, HWND hwndCtrl )  {  static bool s\_bBtnEnabled = true;  s\_bBtnEnabled = !s\_bBtnEnabled;  UIEnable ( IDC\_ALYSON\_BMPBTN, s\_bBtnEnabled );  } |

**5.4、DDV**

WTL的对话框数据验证(DDV)比MFC简单些，在MFC中你需要分别使用DDX(对话框数据交换)宏和DDV(对话框数据验证)宏，在WTL中只需一个宏就可以了，WTL包含基本的数据验证支持，在DDV链中可以使用三个宏：

|  |  |
| --- | --- |
| DDV宏 | 说 明 |
| DDX\_TEXT\_LEN | 和DDX\_TEXT一样，只是还要验证字符串的长度(不包含结尾的空字符)小于或等于限制长度。 |
| DDX\_INT\_RANGE  DDX\_UINT\_RANGE | 和DDX\_INT，DDX\_UINT一样，还加了对数字的最大最小值的验证。 |
| DDX\_FLOAT\_RANGE | 除了像DDX\_FLOAT一样完成数据交换之外，还验证数字的最大最小值。 |

　　ControlMania2有一个ID是IDC\_FAV\_SEASON的edit box，它和成员变量m\_nSeason相关联。

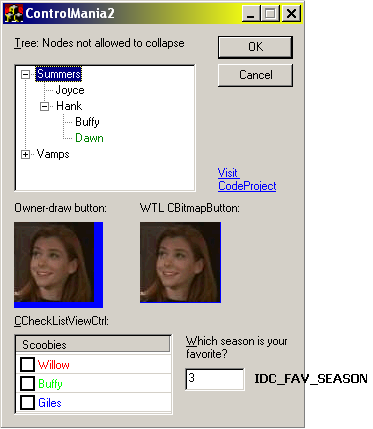


图33 演示DDV验证

　　由于有效的值是1到7，所以使用这样的数据验证宏：

|  |
| --- |
| BEGIN\_DDX\_MAP(CMainDlg)  //...  **DDX\_INT\_RANGE(IDC\_FAV\_SEASON, m\_nSeason, 1, 7)**  END\_DDX\_MAP() |

　　OnOK()调用DoDataExchange()获得season的数值，并验证是在1到7之间。

**5.4.1、处理DDV验证失败**

　　如果控件的数据验证失败，CWinDataExchange会调用重载函数**OnDataValidateError()**，默认的处理是驱动PC喇叭发出声音，你可能想给出更友好的错误指示。OnDataValidateError()的函数原型是：

|  |
| --- |
| void OnDataValidateError ( UINT nCtrlID, BOOL bSave, \_XData& data ); |

　　\_XData是一个WTL的内部数据结构，CWinDataExchange根据输入的数据和允许的数据范围填充这个数据结构。下面是这个数据结构的定义：

|  |
| --- |
| struct \_Xdata  {  \_XDataType nDataType;  Union  　　{  　　 \_XTextData textData;  　　 \_XIntData intData;  　　 \_XFloatData floatData;  };  }; |

　　nDataType指示联合中的三个成员哪个是有意义的，nDataType 的取值可以是：

|  |
| --- |
| enum \_XdataType  {  ddxDataNull = 0,  ddxDataText = 1,  ddxDataInt = 2,  ddxDataFloat = 3,  ddxDataDouble = 4  }; |

　　在我们的例子中，nDataType的值是ddxDataInt，这表示\_XData中的\_XIntData成员是有效的，\_XIntData是个简单的数据结构：

|  |
| --- |
| struct \_XIntData  {  long nVal;  long nMin;  long nMax;  }; |

　　我们重载OnDataValidateError()函数，显示错误信息并告诉用户允许的数值范围：

|  |
| --- |
| void CMainDlg::OnDataValidateError ( UINT nCtrlID, BOOL bSave, \_XData& data )  {  CString sMsg;  sMsg.Format ( \_T("Enter a number between %d and %d"),  data.intData.nMin, data.intData.nMax );  MessageBox ( sMsg, \_T("ControlMania2"), MB\_ICONEXCLAMATION );  ::SetFocus ( GetDlgItem(nCtrlID) );  } |

　　\_XData中的另外两个结构\_XTextData和\_XFloatData的定义在atlddx.h中，感兴趣的话可以打开这个文件查看一下。

**5.5、改变对话框的大小**

　 　WTL引起我的注意的第一件事是对可调整大小对话框的内建的支持。在这之前我曾写过一篇关于这个主题的文章，详情请参考这篇文章。简单的说就是将 CDialogResize类添加到对话框的集成列表，在OnInitDialog()中调用DlgResize\_Init()，然后将消息链入 CDialogResize。

# 第六章 包容ActiveX控件

我们将介绍ATL对话框中使用ActiveX控件的支持，由于ActiveX控件就是ATL的专业，所以WTL没有添加其他的辅助类。不过，在ATL中使用ActiveX控件与在MFC中有很大的不同，所以需要重点介绍。我们将介绍如何包容一个控件并处理控件的事件，开发ATL应用程序相对于MFC的类向导来说有点不方便。在WTL程序中自然可以使用ATL对包容ActiveX控件的支持。

　　例子工程演示如何使用IE的浏览器控件，我选择浏览器控件有两个好处：

1. 每台计算机都有这个控件；
2. 它有很多方法和事件，是个用来做演示的好例子。

　　我当然无法与那些花了大量时间编写基于IE浏览器控件的定制浏览器的人相比，不过，当你读完本篇文章之后，你就知道如何开始编写自己定制的浏览器！

**6.1、使用向导创建工程**

　　WTL的向导可以创建一个支持包容ActiveX控件的程序，我将开始一个名为IEHoster的新工程。我们像上一章一样使用无模式对话框，只是这次要选上支持ActiveX控件包容(Enable ActiveX Control Hosting)，如下图：



图34 向导创建支持ActiveX控件的对话框应用程序

　 　选上这个check box将使我们的对话框从CAxDialogImpl派生，这样就可以包容ActiveX控件。在向导的第二页还有一个名为包容ActiveX控件的 check box，但是选择这个好像对最后的结果没有影响，所以在第一页就可以点击“Finish”结束向导。

**6.2、向导生成的代码**

　　在这一节我将介绍一些以前没有见过的新代码(由向导生成的)，下一节介绍ActiveX包容类的细节。

　　首先要看的文件是stdafx.h，它包含了这些文件：

|  |
| --- |
| #include <atlbase.h>  #include <atlapp.h>  extern CAppModule \_Module;  **#include <atlcom.h>**  **#include <atlhost.h>**  #include <atlwin.h>  #include <atlctl.h>  // .. other WTL headers ... |

**atlcom.h**和**atlhost.h**是很重要的两个，它们含有一些COM相关类的定义(比如智能指针CComPtr)，还有可以包容控件的窗口类。

　　接下来看看maindlg.h中声明的CMainDlg类：

|  |
| --- |
| class CMainDlg : public **CAxDialogImpl<CMainDlg>**,  　　　　　　　public CUpdateUI<CMainDlg>,  　　　　　　　public CMessageFilter, public CIdleHandler |

　　CMainDlg现在是从CAxDialogImpl类派生的，这是使对话框支持包容ActiveX控件的第一步。

　　最后，看看WinMain()中新加的一行代码：

|  |
| --- |
| int WINAPI \_tWinMain(...)  {  //...  \_Module.Init(NULL, hInstance);  **AtlAxWinInit();**  int nRet = Run(lpstrCmdLine, nCmdShow);  \_Module.Term();  return nRet;  } |

**AtlAxWinInit()**注册了一个类名为AtlAxWin的窗口类，ATL用它创建ActiveX控件的包容窗口。

**6.3、使用资源编辑器添加控件**

　　和MFC的程序一样，ATL也可以使用资源编辑器向对话框添加控件。首先，在对话框编辑器上点击鼠标右键，在弹出的菜单中选择“Insert ActiveX control”：

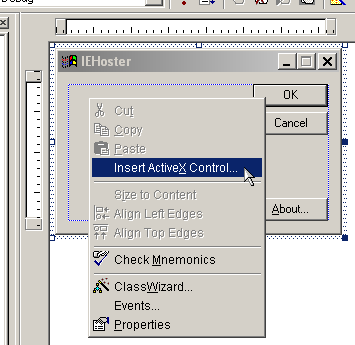


图35 在资源中添加ActiveX控件

　　VC将系统安装的控件显示在一个列表中，滚动列表选择“Microsoft Web Browser”，单击Insert按钮将控件加入到对话框中。查看控件的属性，将ID设为IDC\_IE。对话框中的控件显示应该是这个样子的：

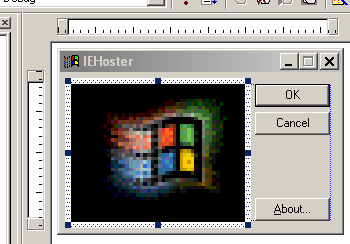


图36 在对话框中添加ActiveX控件

　　如果现在编译运行程序，你会看到对话框中的浏览器控件，它将显示一个空白页，因为我们还没有告诉它到哪里去。

　　在下一节，我将介绍与创建和包容ActiveX控件有关的ATL类，同时我们也会明白这些类是如何与浏览器交换信息的。

**6.4、ATL中使用控件的类**

　　在对话框中使用ActiveX控件需要两个类协同工作：**CAxDialogImpl和CAxWindow**。它们处理所有控件容器必须实现的接口方法，提供通用的功能函数，例如查询控件的某个特殊的COM接口。

**6.4.1、CAxDialogImpl类**

　 　第一个类是CAxDialogImpl，你的对话框要能够包容控件就必须从CAxDialogImpl类派生而不是从CDialogImpl类派生。CAxDialogImpl类重载了Create()和DoModal()函数，这两个函数分别被全局函数AtlAxCreateDialog()和AtlAxDialogBox()调用。既然IEHoster对话框是由Create()创建的，我们看看Atl AxCreateDialog()到底做了些什么工作。

　　AtlAxCreateDialog()使用辅助类\_DialogSplitHelper装载对话框资源，这个辅助类遍历所有对话框的控件，查找由资源编辑器创建的特殊的入口，这些特殊的入口表示这是一个ActiveX控件。例如，下面是IEHoster.rc文件中浏览器 控件的入口：

|  |
| --- |
| CONTROL "",IDC\_IE,"{8856F961-340A-11D0-A96B-00C04FD705A2}",  　　　　WS\_TABSTOP,7,7,116,85 |

第一个参数是窗口文字(空字符串)，第二个是控件的ID，第三个是窗口的类名。DialogSplitHelper:: SplitDialogTemplate()函数找到以''{''开始的窗口类名时就知道这是一个ActiveX控件的入口。它在内存中创建了一个临时对话框模板，在这个新模板中，这些特殊的控件入口被创建的AtlAxWin窗口代替，新的入口是在内存中的等价体：

|  |
| --- |
| CONTROL "{8856F961-340A-11D0-A96B-00C04FD705A2}",IDC\_IE,"AtlAxWin",  　　　　WS\_TABSTOP,7,7,116,85 |

　　结果就是创建了一个相同ID的AtlAxWin窗口，窗口的标题是ActiveX控件的GUID。所以你调用GetDlgItem(IDC\_IE)返回的值是AtlAxWin窗口的句柄，而不是ActiveX控件本身。

　　SplitDialogTemplate()函数完成工作后，AtlAxCreateDialog()接着调用CreateDialogIndirectParam()函数使用修改后的模板创建对话框。

**6.4.2、AtlAxWin类和CAxWindow类**

正如上面讲到的，AtlAxWin实际上是ActiveX控件的宿主窗口，AtlAxWin还会用到一个特殊的窗口接口类：CAxWindow，当 AtlAxWin从模板创建一个对话框后，AtlAxWin的窗口处理过程AtlAx WindowProc()，就会处理WM\_CREATE消息并创建相应的ActiveX控件。ActiveX控件还可以在运行其间动态创建，不需要对话框模板，我会在后面介绍这种方法。

　　 WM\_CREATE的消息处理函数调用全局函数AtlAxCreateControl()，将AtlAxWin窗口的窗口标题作为参数传递给该函数，大家应该记得那实际就是浏览器控件的GUID。AtlAxCreateControl()有会调用一堆其他函数，不过最终会用到 CreateNormalizedObject()函数，这个函数将窗口标题转换成GUID，并最终调用CoCreateInstance()创建 ActiveX控件。

　　由于ActiveX控件是AtlAxWin的子窗口，所以对话框不能直接访问控件，当然CAxWindow提供了这些方法与控件通信，最常用的一个是QueryControl()，这个方法调用控件的QueryInterface()方法。例如，你可以使用 QueryControl()从浏览器控件得到IWebBrowser2接口，然后使用这个接口将浏览器引导到指定的URL。

**6.4.2.1、调用控件的方法**

既然我们的对话框有一个浏览器控件，我们可以使用COM接口与之交互。我们做得第一件事情就是使用IWebBrowser2接口将其引导到一个新URL处。在OnInitDialog()函数中，我们将一个CAx Window变量与包容控件的AtlAxWin联系起来。

|  |
| --- |
| CAxWindow wndIE = GetDlgItem(IDC\_IE); |

　　然后声明一个IWebBrowser2的接口指针并查询浏览器控件的这个接口，使用CAx Window::QueryControl()：

|  |
| --- |
| CComPtr<IWebBrowser2> pWB2;  HRESULT hr;  hr = **wndIE.QueryControl ( &pWB2 )**; |

　　QueryControl()调用浏览器控件的QueryInterface()方法，如果成功就会返回IWebBrowser2接口，我们可以调用Navigate()：

|  |
| --- |
| if ( pWB2 )  {  CComVariant v;　// empty variant  **pWB2->Navigate** ( CComBSTR("http://www.codeproject.com/"), &v, &v, &v, &v );  } |

**6.4.2.2、响应控件触发的事件**

　　从浏览器控件得到接口非常简单，通过它可以单向的与控件通信。通常控件也会以事件的形式与外界通信，ATL有专用的类包装连接点和事件相应，所以我们可以从控件接收到这些事件。为使用对事件的支持需要做四件事：

1. 将CMainDlg变成COM对象；
2. 添加IDispEventSimpleImpl到CMainDlg的继承列表；
3. 填写事件映射链，它指示哪些事件需要处理；
4. 编写事件响应函数。

**6.4.2.3、CMainDlg的修改**

将CMainDlg转变成COM对象的原因是事件应该是基于IDispatch的，为了让CMainDlg暴露这个接口，它必须是个COM对象。IDispEventSimpleImpl提供了IDispatch接口的实现和建立连接点所需的处理函数，当事件发生时 IDispEventSimpleImpl还调用我们想要接收的事件的处理函数。

　　以下的类需要添加到CMainDlg的集成列表中，同时COM\_MAP列出了CMainDlg暴露的接口：

|  |
| --- |
| #include <exdisp.h>　　// browser control definitions  #include <exdispid.h>　// browser event dispatch IDs  class CMainDlg : public CAxDialogImpl<CMainDlg>,  　　　　　　　public CUpdateUI<CMainDlg>,  　　　　　　　public CMessageFilter, public CIdleHandler,  **public CComObjectRootEx<CComSingleThreadModel>,**  **public CComCoClass<CMainDlg>,**  **public IDispEventSimpleImpl<37, CMainDlg, &DIID\_DWebBrowserEvents2>**  {  ...  **BEGIN\_COM\_MAP(CMainDlg)**  **COM\_INTERFACE\_ENTRY2(IDispatch, IDispEventSimpleImpl)**  **END\_COM\_MAP()**  }; |

CComObjectRootEx类和CComCoClass类共同使CMainDlg成为一个COM对象，IDispEvent SimpleImpl的模板参数是事件的ID，我们的类名和连接点接口的IID。事件ID可以是任意正数，连接点对象的IID是DIID\_DWebBrowserEvents2，可以在浏览器控件的相关文档中找到这些参数，也可以查看**exdisp.h**。

**6.4.2.4、填写事件映射链**

　　下一步是给CMainDlg添加事件映射链，这个映射链将我们感兴趣的事件和我们的处理函数联系起来。我们要处理的第一个事件是DownloadBegin，当浏览器开始下载一个页面时就会触发这个事件，我们响应这个事件显示“please wait”信息给用户，让用户知道浏览器正在忙。在MSDN中可以查到DWebBrowserEvents2::DownloadBegin事件的原型：

|  |
| --- |
| void DownloadBegin(); |

这个事件没有参数，也不需要返回值。为了将这个事件的原型转换成事件响应链，我们需要写一个\_ATL\_FUNC\_INFO结构，它包含返回值，参数的个数和参数类型。由于事件是基于IDispatch的，所以所有的参数都用VARIANT表示，这个数据结构的描述相当长(支持很多个数据类型)，以下是常用的几个：

|  |  |
| --- | --- |
| 类型 | 说明 |
| VT\_EMPTY | void |
| VT\_BSTR | BSTR 格式的字符串 |
| VT\_I4 | 4字节有符号整数，用于long类型的参数 |
| VT\_DISPATCH | IDispatch\* |
| VT\_VARIANT | VARIANT |
| VT\_BOOL | VARIANT\_BOOL (允许的取值是VARIANT\_TRUE和VARIANT\_FALSE) |

　　另外，标志VT\_BYREF表示将一个参数转换成相应的指针。例如，VT\_VARIANT|VT\_BYREF表示VARIANT\*类型。下面是\_ATL\_FUNC\_INFO的定义：

|  |
| --- |
| #define \_ATL\_MAX\_VARTYPES 8  struct \_ATL\_FUNC\_INFO  {  CALLCONV cc;  VARTYPE　vtReturn;  SHORT 　 nParams;  VARTYPE　pVarTypes[\_ATL\_MAX\_VARTYPES];  }; |

　　参数：

|  |  |
| --- | --- |
| **参数** | **描述** |
| cc | 我们的事件响应函数的调用方式约定，这个参数必须是CC\_STDCALL，表示是\_\_stdcall方式 |
| vtReturn | 事件响应函数的返回值类型 |
| nParams | 事件带的参数个数 |
| pVarTypes | 相应的参数类型，按从左到右的顺序 |

　　了解这些之后，我们就可以填写DownloadBegin事件处理的\_ATL\_FUNC\_INFO结构：

|  |
| --- |
| \_ATL\_FUNC\_INFO DownloadInfo = { CC\_STDCALL, VT\_EMPTY, 0 }; |

　　现在，回到事件响应链，我们为每一个我们想要处理的事件添加一个SINK\_ENTRY\_INFO宏，下面是处理DownloadBegin事件的宏：

|  |
| --- |
| class CMainDlg : public ...  {  ...  **BEGIN\_SINK\_MAP(CMainDlg)**  **SINK\_ENTRY\_INFO(37, DIID\_DWebBrowserEvents2, DISPID\_DOWNLOADBEGIN,**  **OnDownloadBegin, &DownloadInfo)**  **END\_SINK\_MAP()**  }; |

这个宏的参数是事件的ID(37，与我们在IDispEventSimpleImpl的继承列表中使用的ID一样)，事件接口的IID，事件的分派ID（可以在MSDN或exdispid.h头文件中查到），事件处理函数的名字和指向描述这个事件处理的\_ATL\_FUNC\_INFO结构的指针。

**6.4.2.5、编写事件处理函数**

　　好了，等了这么长时间(吹个口哨！)，我们可以写事件处理函数了：

|  |
| --- |
| void \_\_stdcall CMainDlg::OnDownloadBegin()  {  // show "Please wait" here...  } |

　　现在来看一个复杂一点的事件，比如BeforeNavigate2，这个事件的原型是：

|  |
| --- |
| void BeforeNavigate2 (IDispatch\* pDisp, VARIANT\* URL, VARIANT\* Flags,  VARIANT\* TargetFrameName, VARIANT\* PostData,VARIANT\* Headers, VARIANT\_BOOL\* Cancel ); |

　　此方法有7个参数，对于VARIANT类型参数可以从MSDN查到它到底传递的是什么类型的数据，我们感兴趣的是URL，是一个BSTR类型的字符串。

　　描述BeforeNavigate2事件的\_ATL\_FUNC\_INFO结构是这样的：

|  |
| --- |
| **\_ATL\_FUNC\_INFO BeforeNavigate2Info = { CC\_STDCALL, VT\_EMPTY, 7,**  **{ VT\_DISPATCH, VT\_VARIANT|VT\_BYREF, VT\_VARIANT|VT\_BYREF,**  **VT\_VARIANT|VT\_BYREF, VT\_VARIANT|VT\_BYREF, VT\_VARIANT|VT\_BYREF,**  **VT\_BOOL|VT\_BYREF }**  **};** |

　　和前面一样，返回值类型是VT\_EMPTY表示没有返回值，nParams是7，表示有7个参数。接着是参数类型数组，这些类型前面介绍过了，例如VT\_DISPATCH表示IDispatch\*。

　　事件响应链的入口与前面的例子很相似：

|  |
| --- |
| BEGIN\_SINK\_MAP(CMainDlg)  SINK\_ENTRY\_INFO(37, DIID\_DWebBrowserEvents2, DISPID\_DOWNLOADBEGIN,  　　　　　　　　　　OnDownloadBegin, &DownloadInfo)  SINK\_ENTRY\_INFO(37, DIID\_DWebBrowserEvents2, DISPID\_BEFORENAVIGATE2,  　　　　　　　　　　OnBeforeNavigate2, &BeforeNavigate2Info)  END\_SINK\_MAP() |

　　事件处理函数是这个样子：

|  |
| --- |
| void \_\_stdcall CMainDlg::OnBeforeNavigate2(IDispatch\* pDisp, VARIANT\* URL, VARIANT\* Flags,  VARIANT\* TargetFrameName, VARIANT\* PostData,VARIANT\* Headers, VARIANT\_BOOL\* Cancel )  {  CString sURL = URL->bstrVal;  // ... log the URL, or whatever you''d like ...  } |

　　我打赌你现在是越来越喜欢ClassWizard了，因为当你向MFC的对话框插入一个ActiveX控件时ClassWizard自动为你完成了所有工作。

　　将CMainDlg转换成对象需要注意几件事情，首先必须修改全局函数Run()，现在CMainDlg是个COM对象，我们必须使用CComObject创建CMainDlg：

|  |
| --- |
| int Run(LPTSTR /\*lpstrCmdLine\*/ = NULL, int nCmdShow = SW\_SHOWDEFAULT)  {  CMessageLoop theLoop;  \_Module.AddMessageLoop(&theLoop);  **CComObject<CMainDlg> dlgMain;**  **dlgMain.AddRef();**  if ( dlgMain.Create(NULL) == NULL )  {  　　 ATLTRACE(\_T("Main dialog creation failed!\n"));  　　 return 0;  }  dlgMain.ShowWindow(nCmdShow);  int nRet = theLoop.Run();  \_Module.RemoveMessageLoop();  return nRet;  } |

另一个可替代的方法是不使用CComObject，而使用CComObjectStack类，并删除dlgMain.AddRef ()这一行代码， CComObjectStack对IUnknown的三个方法的实现有些微不足道（它们只是简单的从函数返回），因为它们不是必需的－－这样的COM对象 可以忽略对引用的计数，因为它们仅仅是创建在栈中的临时对象。

　 当然这并不是完美的解决方案，CComObjectStack用于短命的临时对象，不幸的是只要调用它的任何一个IUnknown方法都会引发断言错误。因为CMainDlg对象在开始监听事件时会调用AddRef，所以CComObjectStack不适用于这种情况。

　　解决这个问题要么坚持使用CComObject，要么从CComObjectStack派生一个CComObjectStack2类，允许对IUnknow方法调用。CComObject的那个不必要的引用计数并无大碍－－人们不会注意到它的发生－－但是如果你必须节省那个CPU时钟周期的话，你可以使用本章的例子工程代码中的CComObjectStack2类。

**6.4.2.6、回顾例子工程**

现在我们已经看到事件响应如何工作了，再来看看完整的IEHoster工程，它包容了一个浏览器控件并响应了6个事件，它还显示了一个事件列表，你会对浏览器如何使用它们提供带进度条的界面有个感性的认识，程序处理了以下几个事件：

|  |  |
| --- | --- |
| 事件 | 说 明 |
| BeforeNavigate2  NavigateComplete2 | 这些事件让程序可以控制URL的导航，如果你响应了BeforeNavigate2事件，你可以在事件的处理函数中取消导航 |
| DownloadBegin  DownloadComplete | 程序使用这些事件控制“wait”消息，这表示浏览器正在工作。一个更优美的程序会像IE一样在此期间使用一段动画 |
| CommandStateChange | 这个事件告诉程序向前和向后导航命令何时可用，应用程序将相应的按钮变为可用或不可用 |
| StatusTextChange | 这个事件会在几种情况下触发，例如鼠标移到一个超链接上。这个事件发送一个字符串，应用程序响应这个事件，将这个字符串显示在浏览器窗口下的静态控件上 |

　　程序有四个按钮控制浏览器工作：向后，向前，停止和刷新，它们分别调用IWebBrowser2相应的方法。

　　事件和伴随事件发送的数据都被记录在列表控件中，你可以看到事件的触发，你还可以关闭一些事件记录而仅仅观察其中的一二个事件。为了演示事件处理的重要作用，我们在 BeforeNavigate2事件处理函数中检查URL，如果发现“doubleclick.net”就取消导航。广告和弹出窗口过滤器等一些IE的插件使用的就是这个方法而不是HTTP代理，下面就是做这些检查的代码。

|  |
| --- |
| void \_\_stdcall CMainDlg::OnBeforeNavigate2 (IDispatch\* pDisp, VARIANT\* URL, VARIANT\* Flags,  VARIANT\* TargetFrameName, VARIANT\* PostData, VARIANT\* Headers, VARIANT\_BOOL\* Cancel )  {  USES\_CONVERSION;  CString sURL;  sURL = URL->bstrVal; 　　// You can set \*Cancel to VARIANT\_TRUE to stop the  // navigation from happening. For example, to stop  // navigates to evil tracking companies like doubleclick.net:  if ( sURL.Find ( \_T("doubleclick.net") ) > 0 )  　　 \*Cancel = VARIANT\_TRUE;  } |

　　下面就是我们的程序工作起来的样子：

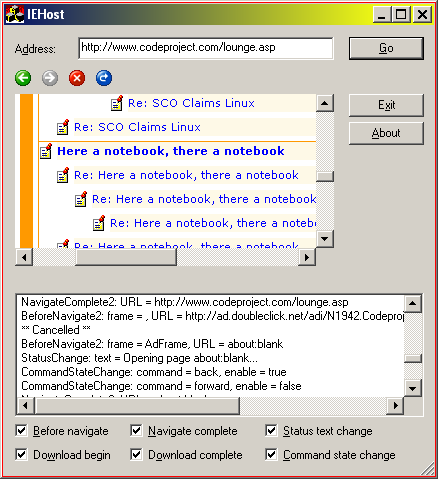


图37 IE控件的事件处理

　　IEHoster还使用了前几章介绍过得类：CBitmapButton(用于浏览器控制按钮)，CListViewCtrl(用于事件记录)，DDX (跟踪checkbox的状态)和CDialogResize.

**6.5、运行时创建ActiveX控件**

　　除了使用资源编辑器，还可以在运行其间动态创建ActiveX控件。About对话框演示了这种技术。对话框编辑器预先放置了一个group box用于浏览器控件的定位：

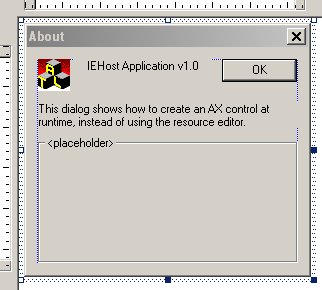


图38 动态创建ActiveX控件的例子

　　在OnInitDialog()函数中我们使用 CAxWindow创建了一个新AtlAxWin，它定位于我们预先放置好的group box的位置上(这个group box随后被销毁)：

|  |
| --- |
| LRESULT CAboutDlg::OnInitDialog(...)  {  CWindow wndPlaceholder = GetDlgItem ( IDC\_IE\_PLACEHOLDER );  CRect rc;  CAxWindow wndIE;  // Get the rect of the placeholder group box, then destroy  // that window because we don''t need it anymore.  wndPlaceholder.GetWindowRect ( rc );  ScreenToClient ( rc );  wndPlaceholder.DestroyWindow();  // Create the AX host window.  wndIE.Create ( \*this, rc, \_T(""), WS\_CHILD | WS\_VISIBLE | WS\_CLIPCHILDREN );  } |

接下来我们用CAxWindow方法创建一个ActiveX控件，有两个方法可以选择：CreateControl()和CreateControlEx()。CreateControlEx()用一个额外的参数返回接口指针，这样就不需要再调用Query Control ()函数。我们感兴趣的两个参数是第一个和第四个参数，第一个参数是字符串形式的浏览器控件的GUID，第四个参数是一个IUnknown\*类型的指针，这个指针指向ActiveX控件的IUnknown接口。创建控件后就可以查询IWebBrowser2接口，然后就可以像前面一样控制它导航到某个URL。

|  |
| --- |
| CComPtr<IUnknown> punkCtrl;  CComQIPtr<IWebBrowser2> pWB2;  CComVariant v;  // Create the browser control using its GUID.  **wndIE.CreateControlEx ( L"{8856F961-340A-11D0-A96B-00C04FD705A2}",NULL, NULL, &punkCtrl );**  // Get an IWebBrowser2 interface on the control and navigate to a page.  pWB2 = punkCtrl;  **pWB2->Navigate ( CComBSTR("about:mozilla"), &v, &v, &v, &v );**  } |

　　对于有ProgID的ActiveX控件可以传递ProgID给CreateControlEx()，代替GUID。例如，我们可以这样创建浏览器控件：

|  |
| --- |
| // 使用控件的ProgID: 创建Shell.Explorer:  **wndIE.CreateControlEx ( L"Shell.Explorer", NULL,　NULL, &punkCtrl );** |

CreateControl()和CreateControlEx()还有一些重载函数用于一些使用浏览器的特殊情况，如果你的应用程序使用WEb页面作为HTML资源，你可以将资源ID作为第一个参数，ATL会创建浏览器控件并导航到这个资源。IEHoster包含一个ID为 IDR\_ABOUTPAGE的WEB页面资源，我们在About对话框中使用这些代码显示这个页面：

|  |
| --- |
| wndIE.CreateControl ( IDR\_ABOUTPAGE ); |

　　这是显示结果：

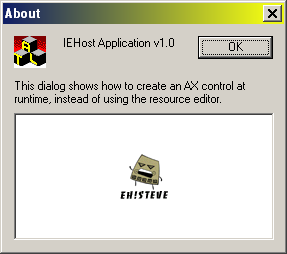


图39 动态生成ActiveX控件

　　例子代码对上面提到的三个方法都用到了，你可以查看CAboutDlg::OnInitDialog()中的注释和未注释的代码，看看它们分别是如何工作的。

**6.6、键盘事件处理**

最后一个但是非常重要的细节是键盘消息。ActiveX控件的键盘处理非常复杂，因为控件和它的宿主程序必须协同工作以确保控件能够看到它感兴趣的消息。例如，浏览器控件允许你使用TAB键在链接之间切换。MFC自己处理了所有工作，所以你永远不会意识到让键盘完美并正确的工作需要多么大的工作量。

不幸的是向导没有为基于对话框的程序生成键盘处理代码，当然，如果你使用Form View作为视图类的SDI程序，你会看到必要的代码已经被添加到PreTranslateMessage()中。当程序从消息队列中得到鼠标或键盘消息时，就使用ATL的WM\_FORWARDMSG消息将此消息传递给当前拥有焦点的控件。它们通常不作什么事情，但是如果是ActiveX控件， WM\_FORWARDMSG消息最终被送到包容这个控件的AtlAxWin，AtlAxWin识别WM\_FORWARDMSG消息，并采取必要的措施看看是否控件需要亲自处理这个消息。

　　如果拥有焦点的窗口没有识别WM\_FORWARDMSG消息，PreTranslateMessage()就会接着调用IsDialogMessage()函数，使得像TAB这样的标准对话框的导航键能正常工作。

　　例子工程的PreTranslateMessage()函数中含有这些必需的代码，由于PreTranslateMessage()只在无模式对话框中有效，所以如果你想在基于对话框的应用程序中正确使用键盘就必须使用无模式对话框。 继续在下一章，我们将回到框架窗口并介绍如何使用分隔窗口。

# 第七章 分隔窗口

　　随着使用两个分隔的视图管理文件系统的资源管理器在Windows 95中第一次出现，分隔窗口逐渐成为一种流行的界面元素。MFC也有一个复杂的功能强大的分隔窗口类，但是要掌握它的用法确实有点难，并且它和文档/视图 框架联系紧密。在本章我们将介绍WTL的分隔窗口，它比MFC的分隔窗口要简单一些。WTL的分隔窗口没有MFC那么多特性，但是易于使用和扩展。

　　本章的例子工程是用WTL重写的ClipSpy，如果你对这个程序不太熟悉，现在可以快速浏览一下本章内容，因为我只是复制了ClipSpy的功能，而没有深入的解释它是如何工作的，毕竟这篇文章的重点是分隔窗口，不是剪贴板。

**7.1、WTL 的分隔窗口**

　　头文件**atlsplit.h**包含所有WTL的分隔窗口类，一共有三个类：CSplitterImpl，CSplitterWindowImpl和CSplitterWindowT，不过你通常只会用到其中的一个。下面将介绍这些类和它们的基本方法。

**7.1.1、相关的类**

CSplitterImpl是一个有两个参数的模板类，一个是窗口界面类的类名，另一个是布尔型变量表示分隔窗口的方向：true表示垂直方向， false表示水平方向。CSplitterImpl类包含了几乎所有分隔窗口的实现代码，它的许多方法是可重载的，重载这些方法可以自己绘制分隔条的外观或者实现其它的效果。CSplitterWindowImpl类是从CWindowImpl和CSplitterImpl两个类派生出来的，但是它的代 码不多，有一个空的WM\_ERASEBKGND消息处理函数和一个WM\_SIZE处理函数用于重新定位分隔窗口。

　　最后一个是 CSplitterWindowT类，它从CSplitterImpl类派生，它的窗口类名是“WTL\_Splitter Window”。还有两个自定义数据类型通常用来取代上面的三个类：CSplitterWindow用于垂直分隔窗口，CHorSplitterWindow用于水平分隔窗口。

**7.1.2、创建分割窗口**

由于CSplitterWindow是从CWindowImpl类派生的，所以你可以像创建其他子窗口那样创建分隔窗口。分隔窗口将存在于整个主框架窗口的生命周期，应该在CMainFrame类添加一个CSplitter Window类型的变量。在CMainFrame::OnCreate()函数内，你可以将分隔窗口作为主窗口的子窗口创建，然后将其设置为主窗口的客户区窗口：

|  |
| --- |
| LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )  {  // ...  const DWORD dwSplitStyle = WS\_CHILD | WS\_VISIBLE | WS\_CLIPCHILDREN | WS\_CLIPSIBLINGS;  const DWORD dwSplitExStyle = WS\_EX\_CLIENTEDGE;  m\_wndSplit.Create ( \*this, rcDefault, NULL, dwSplitStyle, dwSplitExStyle );  m\_hWndClient = m\_wndSplit;  } |

创建分隔窗口之后，你就可以为每个窗格指定窗口或者做其他必要的初始化工作。

7.1.2.1、**基本方法**

|  |
| --- |
| bool SetSplitterPos(int xyPos = -1, bool bUpdate = true)  int GetSplitterPos() |

可以调用SetSplitterPos()函数**设置分隔条的位置**，这个位置表示分割条距离分隔窗口的上边界（水平分隔窗口）或左边界（垂直分隔窗口）有多少个象素点。你可以使用默认值－1将分隔条设置到分隔窗口的中间，使两个窗格大小相同。通常传递true给bUpdate参数表示在移动分隔条之后相应 的改变两个窗格的大小。GetSplitterPos()返回当前分隔条的位置，这个位置也是相对于分隔窗口的上边界或左边界。

|  |
| --- |
| bool SetSinglePaneMode(int nPane = SPLIT\_PANE\_NONE)  int GetSinglePaneMode() |

调用SetSinglePaneMode()函数可以**改变分隔窗口的模式**使单窗格模式还是双窗格模式，在单窗格模式下，只有一个窗格使可见的并且隐藏了分隔条，这和MFC的动态分隔窗口相似（只是没有那个小钳子形状的手柄，它用于重新分隔分隔窗口）。对于nPane参数可用的值是 SPLIT\_PANE\_LEFT，SPLIT\_ PANE\_RIGHT，SPLIT\_PANE\_TOP，SPLIT\_PANE\_BOTTOM，和 SPLIT\_PANE\_NONE，前四个指示显示那个窗格（例如，使用SPLIT\_PANE\_LEFT参数将显示左边的窗格，隐藏右边的窗格），使用 SPLIT \_PANE\_NONE表示两个窗格都显示。GetSinglePaneMode()返回五个SPLIT\_PANE\_\*值中的一个表示当前的模式。

|  |
| --- |
| DWORD SetSplitterExtendedStyle(DWORD dwExtendedStyle, DWORD dwMask = 0)  DWORD GetSplitterExtendedStyle() |

分隔窗口有自己的样式用于控制当整个分隔窗口改变大小时如何移动分隔条。有以下几种样式：

|  |  |
| --- | --- |
| 样式宏 | 说明 |
| SPLIT\_PROPORTIONAL | 两个窗格一起改变大小 |
| SPLIT\_RIGHTALIGNED | 右边的窗格保持大小不变，只改变左边的窗格大小 |
| SPLIT\_BOTTOMALIGNED | 下部的窗格保持大小不变，只改变上边的窗格大小 |
| SPLIT\_NONINTERACTIVE | 分隔条不能被移动并且不相应鼠标 |

如果既没有指定SPLIT\_PROPORTIONAL，也没有指定SPLIT\_RIGHTALIGNED/SPLIT\_ BOTTOMALIGNED，则分隔窗口会变成左对齐或上对齐。如果将SPLIT\_PROPORTIONAL和 SPLIT\_ RIGHTALIGNED/SPLIT\_BOTTOMALIGNED一起使用，则优先选用SPLIT\_PROPORTIONAL样式。扩展样式的默认值是 SPLIT\_PROPORTIONAL。

|  |
| --- |
| bool SetSplitterPane(int nPane, HWND hWnd, bool bUpdate = true)  void SetSplitterPanes(HWND hWndLeftTop, HWND hWndRightBottom, bool bUpdate = true)  HWND GetSplitterPane(int nPane) |

可以调用SetSplitterPane()为分隔窗口的窗格指派子窗口，nPane是一个SPLIT\_PANE\_\*类型的值，表示设置哪一个窗格。hWnd是子窗口的窗口句柄。你可以使用SetSplitterPane()将一个子窗口同时指定给两个窗格，对于bUpdate参数通常使用默认值，也就是告诉分隔窗口立即调整子窗口的大小以适应窗格的大小。可以调用GetSplitterPane()得到某个窗格的子窗口句柄，如果窗格没有指派子窗口则Get SplitterPane()，返回NULL。

|  |
| --- |
| bool SetActivePane(int nPane)  int GetActivePane() |

SetActivePane()函数将分隔窗口中的某个子窗口设置为当前焦点窗口，nPane是SPLIT\_PANE\_\*类型的值，表示需要激活哪个窗格，这个函数还可以设置默认的活动窗格(后面介绍)。GetActivePane()函数查看所有拥有焦点的窗口，如果拥有焦点的窗口是窗格或窗格的子窗口就返回一个SPLIT\_ PANE\_\*类型的值，表示是哪个窗格。如果当前拥有焦点的窗口不是窗格的子窗口，那么GetActivePane ()返回 SPLIT\_PANE\_NONE。

|  |
| --- |
| bool ActivateNextPane(bool bNext = true) |

如果分隔窗口是单窗格模式，焦点被设到可见的窗格上，否则的话，ActivateNextPane()函数将调用GetActivePane()查看拥有焦点的窗口。如果一个窗格(或窗格内的子窗口)拥有检点，分隔窗口就将焦点设给另一个窗格，否则ActivateNextPane()将判断bNext 的值，如果是true就激活left/top窗格，如果是false则激活right/bottom窗格。

|  |
| --- |
| bool SetDefaultActivePane(int nPane)  bool SetDefaultActivePane(HWND hWnd)  int GetDefaultActivePane() |

调用SetDefaultActivePane()函数可以设置默认的活动窗格，它的参数可以是SPLIT\_PANE\_\*类型的值，也可以是窗口的句 柄。如果分隔窗口自身得到的焦点，可以通过调用SetFocus()将焦点转移给默认窗格。GetDefaultActivePane()函数返回 SPLIT\_PANE\_\*类型的值表示哪个窗格是当前默认的活动窗格。

|  |
| --- |
| void GetSystemSettings(bool bUpdate) |

GetSystemSettings()读取系统设置并相应的设置数据成员。分隔窗口在OnCreate()函数中自动调用这个函数，你不需要自己调用这个函数。当然，你的主框架窗口应该响应WM\_SETTINGCHANGE并将它传递给分隔窗口，CSplitterWindow在WM\_SETTINGCHANGE消息的处理函数中调用Get SystemSettings()。传递true给 bUpdate参数，分隔窗口会根据新的设置重画自己。

**7.1.2.2、数据成员**

其他的一些特性可以通过直接访问CSplitterWindow的公有成员来设定，只要GetSystemSettings()被调用了，这些公有成员也会相应的被重置。

|  |  |
| --- | --- |
| 数据成员 | 说 明 |
| m\_cxySplitBar | 控制分隔条的宽度(垂直分隔条)和高度(水平分隔条)。默认值是通过调用GetSystemMetrics (SM\_CXSIZEFRAME)(垂直分隔条)或GetSystemMetrics(SM\_CY SIZEFRAME)(水平分隔条)得到的 |
| m\_cxyMin | 控制每个窗格的最小宽度(垂直分隔)和最小高度(水平分隔)，分隔窗口不允许拖动比这更小的宽度或高度。如果分隔窗口有 WS\_EX\_CLIENTEDGE扩展属性，则这个变量的默认值是0，否则其默认值是2\*GetSystemMetrics(SM\_CXEDGE)(垂直 分隔)或2\*GetSystemMetrics(SM\_CYEDGE)(水平分隔) |
| m\_cxyBarEdge | 控制画在分隔条两侧的3D边界的宽度(垂直分隔)或高度(水平分隔)，其默认值刚好和m\_cxyMin相反 |
| m\_bFullDrag | 如果是true，当分隔条被拖动时窗格大小跟着调整，如果是false，拖动时只显示一个分隔条的影子，直到拖动停止才调整窗 格的大小。默认值是调用System ParametersInfo (SPI\_GET DRAGFULLWINDOWS)函数的返回值。 |

**7.1.3、开始一个例子工程**

　　既然我们已经对分隔窗口有了基本的了解，我们就来看看如何创建一个包含分隔窗口的框架窗口。使用WTL向导开始一个新工程，在第一页选择SDI Application并单击Next，在第二页，如下图所示取消工具条并选择不使用视图窗口：



图40 创建新工程

　　我们不使用视图窗口是因为分隔窗口和它的窗格将作为“视图窗口”，在CMainFrame类中添加一个CSplitterWindow类型的数据成员：

|  |
| --- |
| class CMainFrame : public ...  {  //...  protected:  CSplitterWindow　m\_wndVertSplit;  }; |

　　接着在OnCreate()中创建分隔窗口并将其设为视图窗口：

|  |
| --- |
| LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )  {  //...  // Create the splitter window  const DWORD dwSplitStyle = WS\_CHILD | WS\_VISIBLE | WS\_CLIPCHILDREN | WS\_CLIPSIBLINGS,  　　const DWORD dwSplitExStyle = WS\_EX\_CLIENTEDGE;  m\_wndVertSplit.Create ( \*this, rcDefault, NULL,dwSplitStyle, dwSplitExStyle );  // Set the splitter as the client area window, and resize  // the splitter to match the frame size.  **m\_hWndClient = m\_wndVertSplit;**  UpdateLayout();  // Position the splitter bar.  m\_wndVertSplit.SetSplitterPos ( 200 );  return 0;  } |

需要注意的是在设置分隔窗口的位置之前要先设置m\_hWndClient并调用CFrameWindowImpl:: UpdateLayout()函数，UpdateLayout()将分隔窗口设置为初始时的大小。如果跳过这一步，分隔窗口的大小将不确定，可能小于200个象素点的宽度，最终导致 SetSplitterPos()出现意想不到的结果。还有一种不调用UpdateLayout()函数的方法，就是先得到框架窗口的客户区坐标，然后使用这个客户区坐标替换rcDefault坐标创建分隔窗口。使用这种方式创建的分隔窗口一开始就在正确的初始位置上，随后对位置调整的函数(例如 SetSplitterPos())都可以正常工作。

　　现在运行我们的程序就可以看到分隔条，即使没有创建任何窗格窗口它仍具有基本的行为。你可以拖动分隔条，用鼠标双击分隔条使其移到窗口的中间位置。

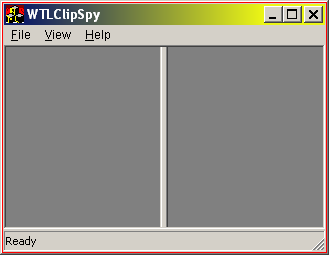


图41 分割窗口视图应用程序

　　为了演示分隔窗口的不同使用方法，我将使用一个CListViewCtrl派生类和一个简单的CRichEdit Ctrl，下面是从CClipSpyListCtrl类摘录的代码，我们在左边的窗格使用这个类：

|  |
| --- |
| typedef CWinTraitsOR<LVS\_REPORT | LVS\_SINGLESEL | LVS\_NOSORTHEADER>　CListTraits;  class CClipSpyListCtrl : public CWindowImpl<CClipSpyListCtrl, CListViewCtrl, CListTraits>,  public CCustomDraw<CClipSpyListCtrl>  {  public:  DECLARE\_WND\_SUPERCLASS(NULL, WC\_LISTVIEW)  BEGIN\_MSG\_MAP(CClipSpyListCtrl)  　　 MSG\_WM\_CHANGECBCHAIN(OnChangeCBChain)  　　 MSG\_WM\_DRAWCLIPBOARD(OnDrawClipboard)  　　 MSG\_WM\_DESTROY(OnDestroy)  　　 CHAIN\_MSG\_MAP\_ALT(CCustomDraw<CClipSpyListCtrl>, 1)  　　 DEFAULT\_REFLECTION\_HANDLER()  END\_MSG\_MAP()  //...  }; |

　　如果你看过前面的几篇文章就会很容易读懂这个类的代码。它响应WM\_CHANGECBCHAIN消息，这样就可以知道是否启动和关闭了其它剪贴板查看程序，它还响应WM\_DRAWCLIPBOARD消息，这样就可以知道剪贴板的内容是否改变。

　　由于分隔窗口窗格内的子窗口在程序运行其间一直存在，我们也可以将它们设为CMainFrame类的成员：

|  |
| --- |
| class CMainFrame : public ...  {  //...  protected:  CSplitterWindow　m\_wndVertSplit;  CClipSpyListCtrl m\_wndFormatList;  CRichEditCtrl　　m\_wndDataViewer;  }; |

**7.2、创建一个窗格内的窗口**

　　既然已经有了分隔窗口和子窗口的成员变量，填充分隔窗口就是一件简单的事情了。先创建分隔窗口，然后创建两个子窗口，使用分隔窗口作为它们的父窗口：

|  |
| --- |
| LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )  {  //...  // Create the splitter window  const DWORD dwSplitStyle = WS\_CHILD | WS\_VISIBLE | WS\_CLIPCHILDREN | WS\_CLIPSIBLINGS,  　　const DWORD dwSplitExStyle = WS\_EX\_CLIENTEDGE;  m\_wndVertSplit.Create ( \*this, rcDefault, NULL,dwSplitStyle, dwSplitExStyle );  // Create the left pane (list of clip formats)  m\_wndFormatList.Create ( m\_wndVertSplit, rcDefault );  // Create the right pane (rich edit ctrl)  const DWORD dwRichEditStyle =WS\_CHILD | WS\_VISIBLE | WS\_HSCROLL | WS\_VSCROLL |  　　 ES\_READONLY | ES\_AUTOHSCROLL | ES\_AUTOVSCROLL | ES\_MULTILINE;  m\_wndDataViewer.Create ( m\_wndVertSplit, rcDefault, NULL, dwRichEditStyle );  m\_wndDataViewer.SetFont ( AtlGetStockFont(ANSI\_FIXED\_FONT) );  // Set the splitter as the client area window, and resize  // the splitter to match the frame size.  m\_hWndClient = m\_wndVertSplit;  UpdateLayout();  m\_wndVertSplit.SetSplitterPos ( 200 );  return 0;  } |

　　注意两个类的Create()函数都用m\_wndVertSplit作为父窗口，RECT参数无关紧要，因为分隔窗口会重新调整它们的大小，所以可以使用CWindow::rcDefault。

　　最后就是将窗口的句柄传递给分隔窗口的窗格，这一步也需要在UpdateLayout()调用之前完成，这样最终所有的窗口都有正确的大小。

|  |
| --- |
| LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )  {  //...  m\_wndDataViewer.SetFont ( AtlGetStockFont(ANSI\_FIXED\_FONT) );  // Set up the splitter panes  m\_wndVertSplit.SetSplitterPanes ( m\_wndFormatList, m\_wndDataViewer );  // Set the splitter as the client area window, and resize  // the splitter to match the frame size.  m\_hWndClient = m\_wndVertSplit;  UpdateLayout();  m\_wndVertSplit.SetSplitterPos ( 200 );  return 0;  } |

　　现在，list控件上增加了几栏，结果看起来是这个样子：

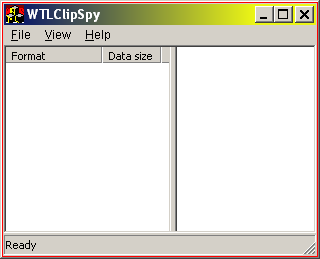


图42 添加了list控件的分割窗口

　　需要注意的是分隔窗口对放进窗格的窗口类型没有限制，不像MFC那样必须是CView的派生类。窗格窗口只要有WS\_CHILD样式就行了，没有任何其他限制。

**7.3、消息处理**

　　由于在主框架窗口和我们的窗格窗口之间加了一个分隔窗口，你可能想知道现在通知消息是如何工作的，比如，主框架窗口是如何收到NM\_CUSTOMDRAW通知消息并将它反射给list控件的？答案就在CSplitterWindowImpl的消息链中：

|  |
| --- |
| BEGIN\_MSG\_MAP(thisClass)  　　MESSAGE\_HANDLER(WM\_ERASEBKGND, OnEraseBackground)  　　MESSAGE\_HANDLER(WM\_SIZE, OnSize)  　　CHAIN\_MSG\_MAP(baseClass)  **FORWARD\_NOTIFICATIONS()**  END\_MSG\_MAP() |

最后的哪个FORWARD\_NOTIFICATIONS()宏最重要，回忆一下第四章，有一些通知消息总是被发送的子窗口的父窗口， FORWARD\_NOTIFICATIONS()就是做了这些工作，它将这些消息转发给分隔窗口的父窗口。也就是说，当list窗口发送一个 WM\_NOTIFY消息给分隔窗口时（它是list的父窗口），分隔窗口就将这个WM\_NOTIFY消息转发给主框架窗口(它是分隔窗口的父窗口)。当主框架窗口反射回消息时，会将消息反射给WM\_NOTIFY消息的最初发送者，也就是list窗口，所以分隔窗口并没有参与消息反射。

　　在list窗口和主框架窗口之间的这些消息传递，并不影响分隔窗口的工作。这使得在程序中添加和移除分隔窗口非常容易，因为子窗口不需要做任何改变就可以继续工作。

**7.4、窗格容器**

　　WTL还有一个被称为窗格容器的构件，它就像Explorer中左边的窗格那样，顶部有一个可以显示文字的区域，还有一个可选择是否显示的Close按钮：

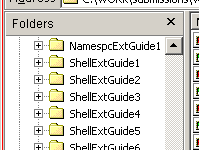


图43 窗格窗口显示Close按钮

　　就像分隔窗口管理两个窗格窗口一样，这个窗格容器也管理一个子窗口，当容器窗口的大小改变时，子窗口也相应的改变大小以便能够填充容器窗口的内部空间。

**7.5、相关的类**

　 　这个窗格容器的实现需要两个类：CPaneContainerImpl和CPaneContainer，它们都在atlctrlx.h中声明。CPaneContainerImpl是一个CWindowImpl派生类，它含有窗格容器的完整实现，CPane Container只是提供了一个类名，除非重载CPaneContainerImpl的方法或改变容器的外观，一般使用CPaneContainer就够了。

**7.5.1、基本方法**

|  |
| --- |
| HWND Create(HWND hWndParent, LPCTSTR lpstrTitle = NULL,DWORD dwStyle = WS\_CHILD | WS\_VISIBLE |  WS\_CLIPSIBLINGS | WS\_CLIPCHILDREN, DWORD dwExStyle = 0, UINT nID = 0, LPVOID lpCreateParam = NULL)  HWND Create(HWND hWndParent, UINT uTitleID, DWORD dwStyle = WS\_CHILD | WS\_VISIBLE | WS\_CLIPSIBLINGS |  WS\_CLIPCHILDREN, DWORD dwExStyle = 0, UINT nID = 0, LPVOID lpCreateParam = NULL) |

　　创建一个CPaneContainer窗口和创建其它子窗口一样。有两个Create()函数，它们的区别仅仅是第二个参数不同。第一个函数需要传递一个字符串作为容器顶部区域显示的文字，第二个参数需要需要传一个字符串的资源ID，其他参数只要使用默认值就行了。

|  |
| --- |
| DWORD SetPaneContainerExtendedStyle(DWORD dwExtendedStyle, DWORD dwMask = 0)  DWORD GetPaneContainerExtendedStyle() |

CPaneContainer还有一些扩展样式用来控制容器窗口上Close按钮的布局方式：

|  |  |
| --- | --- |
| 扩展样式 | 说 明 |
| PANECNT\_NOCLOSEBUTTON | 使用样式去掉顶部的Close按钮 |
| PANECNT\_VERTICAL | 设置这个样式后，顶部的文字区域将沿着容器窗口的左边界垂直放置 |

　　扩展样式的默认值是0，表示容器窗口是水平放置的，还有一个Close按钮。

|  |
| --- |
| HWND SetClient(HWND hWndClient)  HWND GetClient() |

调用SetClient()可以将一个子窗口指派给窗格容器，这和调用CSplitterWindow类的SetSplitter Pane()方法作用类似。SetClient()同时返回原来的客户区窗口句柄，而调用GetClient()则可以得到当前的客户区窗口句柄。

|  |
| --- |
| BOOL SetTitle(LPCTSTR lpstrTitle)  BOOL GetTitle(LPTSTR lpstrTitle, int cchLength)  int GetTitleLength() |

　　调用SetTitle()可以改变容器窗口顶部显示的文字，调用GetTitle()可以得到当前窗口顶部区域显示的文字，调用GetTitleLength()可以得到当前显示的文字的字符个数（不包括结尾的空字符）。

|  |
| --- |
| BOOL EnableCloseButton(BOOL bEnable) |

　　如果窗格容器使用的Close按钮，你可以调用EnableCloseButton()来控制这个按钮的状态。

**7.5.2、在分隔窗口中使用窗格容器**

为了说明窗格容器的使用方法，我们将向ClipSpy的分隔窗口的左窗格添加一个窗格容器，我们将一个窗格容器指派给左窗格取代原来使用的list控件，而将list控件指派给窗格容器。下面是在CMainFrame::OnCreate()中为支持窗格容器而添加的代码。

|  |
| --- |
| LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )  {  //...  m\_wndVertSplit.Create ( \*this, rcDefault, NULL, dwSplitStyle, dwSplitExStyle );  // Create the pane container.  m\_wndPaneContainer.Create ( m\_wndVertSplit, IDS\_LIST\_HEADER );  // Create the left pane (list of clip formats)  m\_wndFormatList.Create ( m\_wndPaneContainer, rcDefault );  //...  // Set up the splitter panes  m\_wndPaneContainer.SetClient ( m\_wndFormatList );  m\_wndVertSplit.SetSplitterPanes ( m\_wndPaneContainer, m\_wndDataViewer ); |

　　注意，现在list控件的父窗口是m\_wndPaneContainer，同时m\_wndPaneContainer被设定成分隔窗口的左窗格。

下面是修改后的左窗格的外观，由于窗格容器在顶部的文本区域自己画了一个三维边框，所以我还要稍微修改一下边框的样式。这样看起来不是很好看，你可以自己调整样式知道你满意为止（当然，你需要在Windows XP 上测试一下哪个界面主题可以使得分隔窗口看起来“更有意思”）。

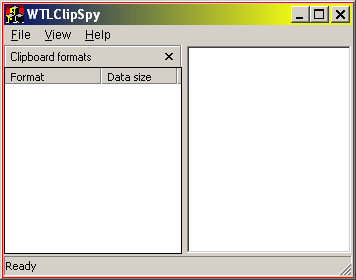


图44 分割窗口的新式样

**7.5.3、关闭按钮和消息处理**

当用户用鼠标单击Close按钮时，窗格容器向父窗口发送一个WM\_COMMAND消息，命令的ID是ID\_PANE\_CLOSE。如果你在分隔窗口中使用了窗格容器，你需要响应整个消息，调用SetSingle PaneMode()隐藏这个窗格（但是，不要忘了提供用户一个重新显示窗格的方法）。

CPaneContainer的消息链也用到了FORWARD\_NOTIFICATIONS()宏，和CSplitterWindow一样，窗格容器在 客户窗口和它的父窗口之间传递通知消息。在ClipSpy这个例子中，在list控件和主框架窗口之间隔了两个窗口(窗格容器和分隔窗口)，但是 FORWARD\_NOTIFICATIONS()宏可以确保所有的通知消息被送到主框架窗口。

**7.6、高级功能**

　　在这一节，我将介绍一些如何使用WTL的高级界面特性。

**7.6.1、嵌套的分隔窗口**

　　如果你要编写一个email的客户端程序，你可能需要使用嵌套的分隔条，一个水平的和一个垂直的分隔条。使用WTL很容易做到这一点：创建一个分隔窗口作为另一个分隔窗口的子窗口。

为了演示这种效果，我将为ClipSpy添加一个水平分隔窗口。首先，添加一个名为m\_wndHorz Splitter的 CHorSplitterWindow类型的成员，像创建垂直分隔窗口m\_wndVertSplitter那样创建这个水平分隔窗口，使水平分隔窗口 m\_wndHorzSplitter成为顶层窗口，将m\_wndVertSplitter创建成m\_wnd HorzSplitter的子窗口。最后将 m\_hWndClient设置为m\_wndHorzSplitter，因为现在水平分隔窗口占据整个主框架窗口的客户区。

|  |
| --- |
| LRESULT CMainFrame::OnCreate()  {  //...  // Create the splitter windows.  m\_wndHorzSplit.Create ( \*this, rcDefault, NULL, dwSplitStyle, dwSplitExStyle );  m\_wndVertSplit.Create( m\_wndHorzSplit, rcDefault, NULL,dwSplitStyle, dwSplitExStyle );  //...  // Set the horizontal splitter as the client area window.  m\_hWndClient = m\_wndHorzSplit;  // Set up the splitter panes  m\_wndPaneContainer.SetClient ( m\_wndFormatList );  m\_wndHorzSplit.SetSplitterPane ( SPLIT\_PANE\_TOP, m\_wndVertSplit );  m\_wndVertSplit.SetSplitterPanes ( m\_wndPaneContainer, m\_wndDataViewer );  //...  } |

最终的结果是这个样子的：

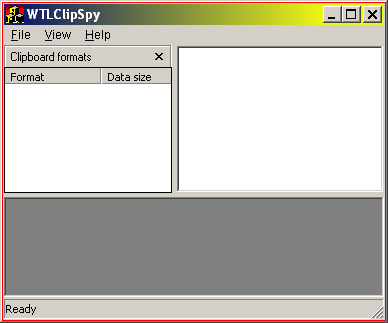


图45 嵌套分割窗口

**7.6.2、在窗格中使用ActiveX控件**

　　在分隔窗口的窗格中使用ActiveX控件与在对话框中使用ActiveX控件类似，使用CAxWindow类的方法在运行是创建控件，然后将这个CAxWindow指定给分隔窗口的窗格。下面演示了如何在水平分隔窗口下面的窗格中使用浏览器控件：

|  |
| --- |
| // Create the bottom pane (browser)  CAxWindow wndIE;  const DWORD dwIEStyle=WS\_CHILD| WS\_VISIBLE | WS\_CLIPCHILDREN |WS\_HSCROLL | WS\_VSCROLL;  wndIE.Create( m\_wndHorzSplit, rcDefault,\_T("http://www.codeproject.com"), dwIEStyle );  // Set the horizontal splitter as the client area window.  m\_hWndClient = m\_wndHorzSplit;  // Set up the splitter panes  m\_wndPaneContainer.SetClient ( m\_wndFormatList );  m\_wndHorzSplit.SetSplitterPanes ( m\_wndVertSplit, wndIE );  m\_wndVertSplit.SetSplitterPanes ( m\_wndPaneContainer, m\_wndDataViewer ); |

**7.6.3、特殊绘制**

如果你想改变分隔条的外观，例如在上面使用一些材质，你可以从CSplitterWindowImpl派生新类并重载DrawSplitterBar ()函数。如果你只是想调整一下分隔条的外观，可以复制CSplitterWindow Impl类的函数，然后稍做修改。下面的例子就在分隔条中使用了斜交叉线图案。

|  |
| --- |
| template <bool t\_bVertical = true>  class CMySplitterWindowT : public CSplitterWindowImpl<CMySplitterWindowT<t\_bVertical>, t\_bVertical>  {  public:  DECLARE\_WND\_CLASS\_EX(\_T("My\_SplitterWindow"), CS\_DBLCLKS, COLOR\_WINDOW)  // Overrideables  void DrawSplitterBar(CDCHandle dc)  {  　　 RECT rect;  　　 if ( m\_br.IsNull() )  　　　 m\_br.CreateHatchBrush(HS\_DIAGCROSS,t\_bVertical?RGB(255,0,0):RGB(0,0,255) );  　　 if ( GetSplitterBarRect ( &rect ) )  　　 {  　　　　 dc.FillRect ( &rect, m\_br );  　　　　 // draw 3D edge if needed  　　　　 if ( (GetExStyle() & WS\_EX\_CLIENTEDGE) != 0)  　　　　　　 dc.DrawEdge(&rect, EDGE\_RAISED,　t\_bVertical ? (BF\_LEFT | BF\_RIGHT)  　　　　　　　　 : (BF\_TOP | BF\_BOTTOM));  　　 }  }  protected:  CBrush m\_br;  };  typedef CMySplitterWindowT<true>　　CMySplitterWindow;  typedef CMySplitterWindowT<false>　 CMyHorSplitterWindow; |

　　这就是结果（将分隔条变宽是为了更容易看到效果）：

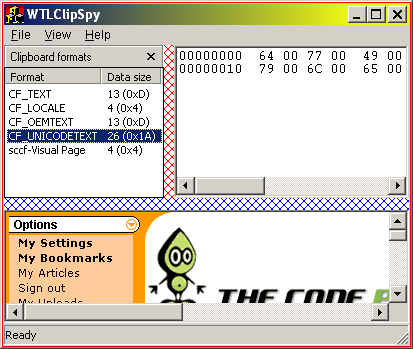


图46 在分隔条中使用了斜交叉线图案

**7.6.4、窗格容器内的特殊绘制**

　　CPaneContainer也有几个函数可以重载，用来改变窗格容器的外观。你可以从CPaneContainerImpl派生新类并重载你需要的方法，例如：

|  |
| --- |
| class CMyPaneContainer :　public CPaneContainerImpl<CMyPaneContainer>  {  public:  DECLARE\_WND\_CLASS\_EX(\_T("My\_PaneContainer"), 0, -1)  //... overrides here ...  }; |

　　一些更有意思的方法是：

|  |
| --- |
| void CalcSize() |

调用CalcSize()函数只是为了设置m\_cxyHeader，这个变量控制着窗格容器的顶部区域的宽度和高度。不过 SetPaneContainerExtendedStyle()函数中有一个BUG，导致窗格从水平切换到垂直时没有调用派生类的CalcSize() 方法，你可以将CalcSize()调用改为pT->CalcSize()来修补这个BUG。

|  |
| --- |
| HFONT GetTitleFont() |

这个方法返回一个HFONT，它被用来画顶部区域的文字，默认的值是调用GetStockObject ( DEFAULT\_GUI\_FONT)得到的字体，也就是MS Sans Serif。如果你想改为更现代的Tahoma字体，你可以重载GetTitleFont()方法，返回你创建的Tahoma字体。

|  |
| --- |
| BOOL GetToolTipText(LPNMHDR lpnmh) |

重载这个方法提供鼠标移到Close按钮时弹出的提示信息，这个函数实际上是TTN\_GETDISP INFO的相应函数，你可以将lpnmh转换成 NMTTDISPINFO\*,并设置这个数据结构内相应的成员变量。记住一点，你必须检查通知代码，它可能是TTN\_GETDISPINFO或 TTN\_GETDISPINFOW，你需要有区别的访问这两个数据结构。

|  |
| --- |
| void DrawPaneTitle(CDCHandle dc) |

　　你可以重载这个方法自己画顶部区域，你可以用GetClientRect()和m\_cxyHeader来计算顶部区域的范围。下面的例子演示了在水平容器的顶部区域画一个渐变填充的背景：

|  |
| --- |
| void CMyPaneContainer::DrawPaneTitle ( CDCHandle dc )  {  RECT rect;  GetClientRect(&rect);  TRIVERTEX tv[] = {　{ rect.left, rect.top, 0xff00 },  　　 { rect.right, rect.top + m\_cxyHeader, 0, 0xff00 }};  GRADIENT\_RECT gr = { 0, 1 };  dc.GradientFill ( tv, 2, &gr, 1, GRADIENT\_FILL\_RECT\_H );  } |

　　例子工程代码中演示了对这几个方法的重载，使得结果看起来是这个样子的：

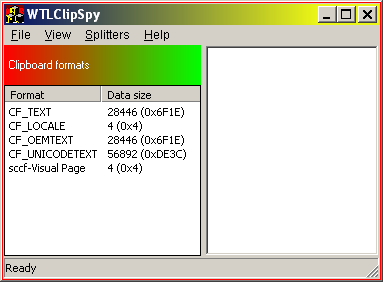


图47 在顶部区域画一个渐变填充的背景

　　从上面的图中可以看到，这个演示程序有一个Splitters菜单，通过它可以在各种风格的分隔条(包括自画风格)和窗格容器之间切换，比较它们之间的异同。你还可以锁定分隔条的位置，这是通过设置和取消SPLIT\_NONINTERACTIVE扩展风格来实现的。

**7.6.5、在状态栏显示进度条**

　　正如我在前几篇文章中做得保证那样，新的ClipSpy也演示了如何在状态条上创建进展条，它和MFC版本得功能一样，几个相关得步骤是：

1. 得到状态条第一个窗格得坐标范围RECT；
2. 创建一个进展条作为状态条得子窗口，窗口大小就是哪个状态条窗格得大小；
3. 随着edit控件被填充的同时更新进展条的位置。

　　这些代码在CMainFrame::CreateProgressCtrlInStatusBar()函数中。

# 第八章 属性页与向导

　　甚至在成为Windows 95的通用控件之前，使用属性表来表示一些选项就已经成为一种很流行的方式。向导模式的属性表通常用来引导用户安装软件或完成其他复杂的工作。WTL对这 两种方式的属性表都提供了很好的支持，可以使用前面介绍的与对话框相关的特性，如DDX和DDV。在本章我将演示如何创建一个基本的属性表和向导，如何处理属性页发送的通知消息和事件。

**8.1、WTL 的属性表类**

　　实现一个属性表需要CPropertySheetWindow和 CPropertySheetImpl两个类联合使用，它们都定义在atldlgs.h头文件中。CPropertySheetWindow类是一个窗口接口类(也就是说是一个CWindow派生类)，CPropertySheetImpl有消息映射链和窗口的完整实现，这和ATL的基本窗口类相似，它需要CWindow和CWindowImpl两个类联合使用。

　　CPropertySheetWindow类封装了对各种PSM\_\* 消息的处理，例如，SetActivePageByID()封装了PSM\_SETCURSELID消息。CPropertySheetImpl类管理一个 PROPSHEETHEADER结构和一个HPROPSHEETPAGE类型的数组，CPropertySheetImpl类还提供了一些方法用来填充PROPSHEET HEADER结构，添加或删除属性页，你也可以使用m\_psh成员变量直接操作PROPSHEETHEADER结构。

　　最后，CPropertySheet类是CPropertySheetImpl类的一个特例，你可以直接使用它而不需要定制整个属性表。

**8.1.1、CPropertySheetImpl 的方法**

　　下面是CPropertySheetImpl类的一些重要方法。由于许多方法仅仅是对窗口消息的封装，所以就不在这里列出，你可以查看atldlgs.h中完整的函数清单。

|  |
| --- |
| CPropertySheetImpl(\_U\_STRINGorID title = (LPCTSTR) NULL,UINT uStartPage = 0, HWND hWndParent = NULL) |

　 　CPropertySheetImpl类的构造函数允许你使用一些常用的属性(默认值)，所以就不需要在调用其他的方法设置它们。title指定显示在属性表的标题栏的文字，\_U\_STRINGorID是一个WTL的工具类，它可以自动转换LPCTSTR和资源ID，例如，下面的两行代码都是正确的：

|  |
| --- |
| CPropertySheetImpl mySheet ( IDS\_SHEET\_TITLE );  CPropertySheetImpl mySheet ( \_T("My prop sheet") ); |

　　IDS\_SHEET\_TITLE 是字符串的ID。 uStartPage 是属性表启动时激活的属性页，是一个从0开始的索引。hWndParent 是属性表的父窗口的句柄。

|  |
| --- |
| BOOL AddPage(HPROPSHEETPAGE hPage)  BOOL AddPage(LPCPROPSHEETPAGE pPage) |

　 　添加一个属性页。如果这个属性页已经创建了，你可以使用第一个重载函数，使用属性页的句柄(HPROPSHEETPAGE)作为参数。通常是使用第二个重载函数，使用这个重载函数只需设置一个PROPSHEETPAGE数据结构(后面会讲到，它和CPropertyPageImpl一起协同工作)， CPropertySheet Impl会为你创建并管理这个属性页。

|  |
| --- |
| BOOL RemovePage(HPROPSHEETPAGE hPage)  BOOL RemovePage(int nPageIndex) |

　　移除一个属性页，可以使用属性页的句柄或索引。

|  |
| --- |
| BOOL SetActivePage(HPROPSHEETPAGE hPage)  BOOL SetActivePage(int nPageIndex) |

　　设置属性表的活动页面。可以使用属性页的句柄或索引。你可以在属性表创建(显示)之前使用这个方法动态的设置处于激活的属性页。

|  |
| --- |
| void SetTitle(LPCTSTR lpszText, UINT nStyle = 0) |

设置属性表窗口的标题文字。nStyle可以是0或PSH\_PROPTITLE，如果是PSH\_PROPTITLE，则属性表就具有 PSH\_PROPTITLE样式，这样系统会在你通过lpszText参数指定的窗口标题前添加字符串“Properties for”。

|  |
| --- |
| void SetWizardMode() |

设置PSH\_WIZARD样式，将属性表改称向导模式，这个函数必须在属性表显示之前调用。

|  |
| --- |
| void EnableHelp() |

　　设置PSH\_HASHELP样式，将在属性表中添加帮助按钮。需要注意的是你还要在每个属性页中使帮助按钮可用，并提供帮助才能使之生效。

|  |
| --- |
| INT\_PTR DoModal(HWND hWndParent = ::GetActiveWindow()) |

　　创建并显示一个模式的属性表，返回正值表示操作成功，有关PropertySheet() API的帮助文档有有关返回值的详细解释，如果发生错误，属性表无法创建，DoModal()返回－1。

|  |
| --- |
| HWND Create(HWND hWndParent = NULL) |

　　创建并显示一个无模式的属性表，返回值是窗口的句柄，如果发生错误，属性表无法创建，Create()返回NULL。

**8.2、WTL 的属性页类**

　　WTL对属性页的封装类与属性表的封装类相似，有一个窗口接口类 CPropertyPageWindow 和一个实现类 CPropertyPageImpl 。CPropertyPageWindow 很小，包含最常用的需要在作为父窗口的属性表中调用的方法。

　 　CPropertyPageImpl 是从 CDialogImplBaseT派生，由于属性页是从对话框资源中创建的，这就意味着所有可以在对话框中使用的WTL的特性都可以在属性页中使用，如 DDX和DDV。CProperty PageImpl 有两个主要作用：管理一个PROPSHEETPAGE数据结构(保存在成员变量m\_psp中)，处理所有PSN\_开头的通知消息。对于很简单的属性页可以直接使用CPropertyPage类，这个类只适合与用户没有任何交互的属性页，例如“关于”页面或者向导中的介绍页面

　　也可以创建含有ActiveX控件的属性页。首先，这需要在stdafx.h文件中添加对atlhost.h的包含，还要使用CAxPropertyPageImpl 代替CPropertyPageImpl。对于简单的页面可以使用CAxPropertyPage代替CPropertyPage。

**8.2.1、CPropertyPageImpl 的方法**

CPropertyPageImpl 管理着一个 PROPSHEETPAGE 结构，也就是公有成员 m\_psp。CPropertyPage Impl还重载了PROPSHEETPAGE\*操作符，所以你可以将CPropertyPageImpl传递给需要LPPROP SHEETPAGE 或 LPCPROPSHEETPAGE 类型的参数的方法，例如CPropertySheetImpl::AddPage()。

　　CPropertyPageImpl的构造函数允许你设置页面的标题，标题通常显示在页面的Tab标签上：

|  |
| --- |
| CPropertyPageImpl(\_U\_STRINGorID title = (LPCTSTR) NULL) |

　　如果你不想让属性表创建属性页面而是想手工创建页面，你可以调用Create():

|  |
| --- |
| HPROPSHEETPAGE Create() |

　 　Create() 只是调用用m\_psp做参数调用了 CreatePropertySheetPage() 。如果你向一个已经创建的属性表添加属性页或者向另一个不在控制的属性表添加属性页(例如，处理系统Shell扩展的属性表)，那就只需要调用 Create()函数。

　　下面的三个方法用于设置属性页的各种标题文本：

|  |
| --- |
| void SetTitle(\_U\_STRINGorID title)  void SetHeaderTitle(LPCTSTR lpstrHeaderTitle)  void SetHeaderSubTitle(LPCTSTR lpstrHeaderSubTitle) |

　　第一个方法改变页面标签的文字，另外几个用来设置Wizard97样式的向导中属性页顶部的文字。

|  |
| --- |
| void EnableHelp() |

　　设置m\_psp中的PSP\_HASHELP标志，当本页面激活时使属性表的帮助按钮可用。

**8.2.2、处理通知消息**

　　CPropertyPageImpl有一个消息映射处理WM\_NOTIFY。如果通知代码是PSN\_\*的值，OnNotify()就会调用相应的通知处理函数。这使用了编译阶段虚函数机制，从而使得派生类可以很容易的重载这些处理函数。

　　由于WTL 3和WTL 7设计的改变，从而存在两套不同的通知处理机制。在WTL 3中通知处理函数返回的值与PSN\_\*消息的返回值不同，例如，WTL 3是这样处理PSN\_WIZFINISH的：

|  |
| --- |
| case PSN\_WIZFINISH:  　　 lResult = !pT->OnWizardFinish();  　　 break; |

OnWizardFinish()期望返回TRUE结束向导，FALSE阻止关闭向导。这个方法很简陋，但是IE5的通用控件对 PSN\_WIZFINISH处理的返回值添加了新解释，他返回需要获得焦点的窗口的句柄。WTL 3的程序将不能使用这个特性，因为它对所有非0的返回值都做相同的处理。

　　在WTL 7中，OnNotify() 没有改变 PSN\_\* 消息的返回值，处理函数返回任何文档中规定的合法数值和正确的行为。当然，为了向前兼容，WTL 3 仍然使用当前默认的工作方式，要使用WTL 7的消息处理方式，你必须在中including atldlgs.h一行之前添加一行定义：

|  |
| --- |
| #define \_WTL\_NEW\_PAGE\_NOTIFY\_HANDLERS |

　　编写新的代码没有理由不使用WTL 7的消息处理函数，所以这里就不介绍WTL 3的消息处理方式。

CPropertyPageImpl 为所有消息提供了默认的通知消息处理函数，你可以重载与你的程序有关的消息处理函数完成特殊的操作。默认的消息处理函数和相应的行为如下：

|  |  |
| --- | --- |
| 事件 | 说明 |
| int OnSetActive() | 允许页面成为激活状态 |
| BOOL OnKillActive() | 允许页面成为非激活状态 |
| int OnApply() | 返回 PSNRET\_NOERROR 表示应用操作成功完成 |
| void OnReset() | 无相应的动作 |
| BOOL OnQueryCancel() | 允许取消操作 |
| int OnWizardBack() | 返回到前一个页面 |
| int OnWizardNext() | 进行到下一个页面 |
| INT\_PTR OnWizardFinish() | 允许向导结束 |
| void OnHelp() | 无相应的动作 |
| BOOL OnGetObject(LPNMOBJECTNOTIFY lpObjectNotify) | 无相应的动作 |
| int OnTranslateAccelerator(LPMSG lpMsg) | 返回 PSNRET\_NOERROR 表示消息没有被处理 |
| HWND OnQueryInitialFocus(HWND hWndFocus) | 返回 NULL 表示将按Tab Order顺序的第一个控件设为焦点状态 |

**8.3、创建一个属性表**

　　关于这些类的解释就全部讲完了，现在需要一个例子程序演示如何使用它们。本章的例子工程是一个简单的SDI程序，它在客户区显示一幅图片并使用一种颜色填充背景，使用的图片和颜色可以通过一个选项对话框(一个属性表)来设置，还有一个向导(稍后会介绍)。

**8.3.1、最简单的属性表**

　　首先用WTL的向导创建一个SDI工程，然后为关于对话框添加一个属性表。首先改变向导创建的关于对话框样式，使它用起来像个属性页。

　　第一步就是**去除OK按钮**，因为属性表不希望属性页自己关闭。在Style Tab中，将对话框样式改为**Child**，在Thin Border，选择Title Bar，在More Styles tab，选择Disabled。

　　第二步(也是最后一步)是在OnAppAbout()的处理函数中创建一个属性表，我们使用非定制的CPropertySheet 和 CPropertyPage类：

|  |
| --- |
| LRESULT CMainFrame::OnAppAbout(...)  {  CPropertySheet sheet ( \_T("About PSheets") );  CPropertyPage<IDD\_ABOUTBOX> pgAbout;  sheet.AddPage ( pgAbout );  sheet.DoModal();  return 0;  } |

　　结果看起来向下面这样：

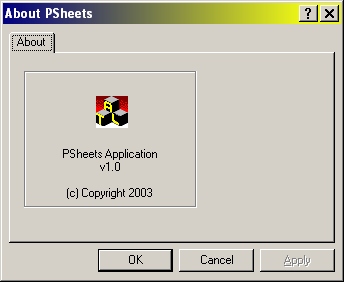


图48 简单属性表

**8.3.2、创建一个有用的属性页**

　　并不是每一个属性表中的每一个属性页都像关于对话框这么简单，大多数属性页需要使用CPropertyPageImpl的派生类，所以我们现在就看一个这样的类。我们创建了一个新的属性页用来设置客户区背景显示的图片，它是这个样子的：

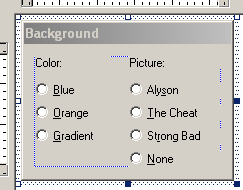


图49 有用的属性页

　　这个对话框的样式和关于页面相同，我们需要一个新类来和这个属性页协同工作，我们将其命名为CBackgroundOptsPage。这个类是从CPropertyPageImpl类派生的，它有一个CWinDataExchange来支持DDX。

|  |
| --- |
| class CBackgroundOptsPage : public CPropertyPageImpl<CBackgroundOptsPage>,  public CWinDataExchange<CBackgroundOptsPage>  {  public:  enum { IDD = IDD\_BACKGROUND\_OPTS };  // Construction  CBackgroundOptsPage();  ~CBackgroundOptsPage();  // Maps  BEGIN\_MSG\_MAP(CBackgroundOptsPage)  　　 MSG\_WM\_INITDIALOG(OnInitDialog)  　　 CHAIN\_MSG\_MAP(CPropertyPageImpl<CBackgroundOptsPage>)  END\_MSG\_MAP() 　　BEGIN\_DDX\_MAP(CBackgroundOptsPage)  　　 DDX\_RADIO(IDC\_BLUE, m\_nColor)  　　 DDX\_RADIO(IDC\_ALYSON, m\_nPicture)  END\_DDX\_MAP()  // Message handlers  BOOL OnInitDialog ( HWND hwndFocus, LPARAM lParam );  // Property page notification handlers  int OnApply();  // DDX variables  int m\_nColor, m\_nPicture;  }; |

　　关于这个类需要注意几点：

1. 有一个名为IDD的公有成员将对话框与资源联系起来。
2. 消息映射链和CDialogImpl相似。
3. 消息映射链将消息链入CPropertyPageImpl，从而使我们能够处理与属性表相关的通知消息。
4. 有一个OnApply()处理函数在单击属性表中的OK按钮时保存用户的选择。

　　OnApply() 非常简单，它调用 DoDataExchange() 更新 DDX 变量，然后返回一个代码标识是否可以关闭这个属性表：

|  |
| --- |
| int CBackgroundOptsPage::OnApply()  {  return DoDataExchange(true) ? PSNRET\_NOERROR : PSNRET\_INVALID;  } |

　　我们还要在主窗口添加一个Tools|Options菜单来打开属性表，这个菜单的处理函数创建一个属性表，但是添加了一个新属性页CBackgroundOptsPage。

|  |
| --- |
| void CMainFrame::OnOptions ( UINT uCode, int nID, HWND hwndCtrl )  {  CPropertySheet sheet ( \_T("PSheets Options"), 0 );  CBackgroundOptsPage pgBackground;  CPropertyPage<IDD\_ABOUTBOX> pgAbout;  pgBackground.m\_nColor = m\_view.m\_nColor;  pgBackground.m\_nPicture = m\_view.m\_nPicture;  sheet.m\_psh.dwFlags |= PSH\_NOAPPLYNOW;  sheet.AddPage ( pgBackground );  sheet.AddPage ( pgAbout );  if ( IDOK == sheet.DoModal() )  　　 m\_view.SetBackgroundOptions ( pgBackground.m\_nColor, pgBackground.m\_nPicture );  } |

属性表的构造函数的第二个参数是0，表示将索引是0的页面初始是可见的，你可以将其设为1，使得属性表第一次显示时显示关于页面。既然是演示代码，我就偷个懒，使用一个公有变量与CBack groundOptsPage属性页的radio button建立关联，在主窗口中直接为其赋初始值，当用户单击属性表的OK按钮时在将其读出来。

　　如果用户点击OK按钮，DoModal()发送IDOK，我们通知视图窗口使用新的图片和背景颜色。下面是几个屏幕截图显示几个不同的样式的视图：

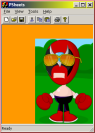


图50 不同样式的视图

**8.3.3、创建一个更好的属性表类**

　　在OnOptions()中创建属性表是个好主意，但是在这里使用很多初始化代码却非常糟糕，这不是CMainFrame应该做得事情。更好的方法是从CPropertySheetImpl派生一个新类，在这个类中完成这些任务。

|  |
| --- |
| #include "BackgroundOptsPage.h"  class CAppPropertySheet : public CPropertySheetImpl<CAppPropertySheet>  {  public:  // Construction  CAppPropertySheet ( \_U\_STRINGorID title = (LPCTSTR) NULL,UINT uStartPage = 0, HWND hWndParent = NULL );  // Maps  BEGIN\_MSG\_MAP(CAppPropertySheet)  　　 CHAIN\_MSG\_MAP(CPropertySheetImpl<CAppPropertySheet>)  END\_MSG\_MAP()  // Property pages  CBackgroundOptsPage　　　　 m\_pgBackground;  CPropertyPage<IDD\_ABOUTBOX> m\_pgAbout;  }; |

　　我们使用这个类封装属性表中各个属性页的细节，将初始化代码移到属性表内部完成，构造函数完成添加页面，并设置其他必需的标志：

|  |
| --- |
| CAppPropertySheet::CAppPropertySheet(\_U\_STRINGorID title,UINT uStartPage,HWND ndParent ) :  CPropertySheetImpl<CAppPropertySheet> ( title, uStartPage, hWndParent )  {  m\_psh.dwFlags |= PSH\_NOAPPLYNOW;  AddPage ( m\_pgBackground );  AddPage ( m\_pgAbout );  } |

　　这样一来，OnOptions()处理函数就变得简单了一些：

|  |
| --- |
| void CMainFrame::OnOptions ( UINT uCode, int nID, HWND hwndCtrl )  {  CAppPropertySheet sheet ( \_T("PSheets Options"), 0 );  sheet.m\_pgBackground.m\_nColor = m\_view.m\_nColor;  sheet.m\_pgBackground.m\_nPicture = m\_view.m\_nPicture;  if ( IDOK == sheet.DoModal() )  　　 m\_view.SetBackgroundOptions ( sheet.m\_pgBackground.m\_nColor, sheet.m\_pgBackground.m\_nPicture );  } |

**8.3.4、创建一个向导样式的属性表**

创建一个向导和创建一个属性表很相似，这并不奇怪，只需稍做修改添加“上一步”和“下一步”按钮就行了。和MFC一样，你需要重载 OnSetActive()函数，并调用SetWizardButtons()使相应的按钮可用。我们先从一个简单的介绍页面开始，它的ID是 IDD\_WIZARD\_INTRO：

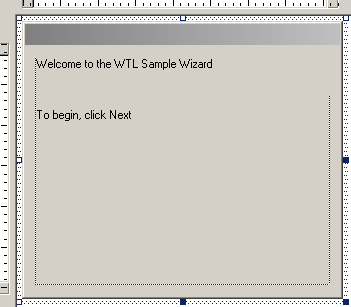


图51 向导式属性页

　　注意这个页面没有标题栏文字，因为向导中的所有的页面通常都有相同的标题，我更愿意在CPropertySheetImpl的构造函数中设置这些文字，然后每个页面使用这个字符串资源。这就是为什么我只需要改变一个字符串就能改变所有页面标题文字的原因。

　　关于这个页面的实现代码在CWizIntroPage类中：

|  |
| --- |
| class CWizIntroPage : public CPropertyPageImpl<CWizIntroPage>  {  public:  enum { IDD = IDD\_WIZARD\_INTRO };  // Construction  CWizIntroPage();  // Maps  BEGIN\_MSG\_MAP(COptionsWizard)  　　 CHAIN\_MSG\_MAP(CPropertyPageImpl<CWizIntroPage>)  END\_MSG\_MAP()  // Notification handlers  int OnSetActive();  }; |

构造函数使用(引用)一个字符串资源ID来设置页面的文字：

|  |
| --- |
| CWizIntroPage::CWizIntroPage() : CPropertyPageImpl<CWizIntroPage>( IDS\_WIZARD\_TITLE )  {  } |

　　当这个页面激活时，字符串IDS\_WIZARD\_TITLE ("PSheets Options Wizard")将出现在向导的标题栏。OnSetActive()仅仅使“下一步”按钮可用：

|  |
| --- |
| int CWizIntroPage::OnSetActive()  {  SetWizardButtons ( PSWIZB\_NEXT );  return 0; } |

　　为了实现一个向导，我们需要创建一个类COptionsWizard，还要在主窗口添加菜单Tools|Wizard。COptionsWizard类的构造函数和CAppPropertySheet类的构造函数一样，只是设置必要的样式标志和添加页面。

|  |
| --- |
| class COptionsWizard : public CPropertySheetImpl<COptionsWizard>  {  public:  // Construction  COptionsWizard ( HWND hWndParent = NULL );  // Maps  BEGIN\_MSG\_MAP(COptionsWizard)  　　 CHAIN\_MSG\_MAP(CPropertySheetImpl<COptionsWizard>)  END\_MSG\_MAP()  // Property pages  CWizIntroPage m\_pgIntro;  };  COptionsWizard::COptionsWizard ( HWND hWndParent ) :  CPropertySheetImpl<COptionsWizard> ( 0U, 0, hWndParent )  {  SetWizardMode();  AddPage ( m\_pgIntro );  } |

　　CMainFrame类的Tools|Wizard菜单处理函数是这个样子：

|  |
| --- |
| void CMainFrame::OnOptionsWizard ( UINT uCode, int nID, HWND hwndCtrl )  {  COptionsWizard wizard;  wizard.DoModal();  } |

　　这就是向导的效果：

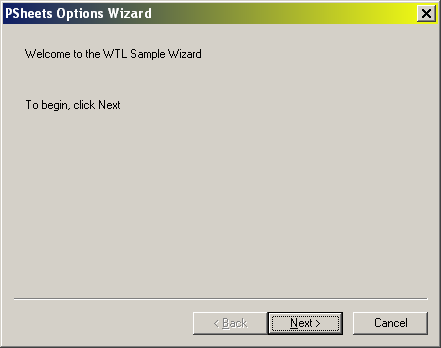


图52 演示向导式属性页

**8.3.5、添加更多的属性页，使用DDV**

　　为了使这个向导能够有点用处，我们要为其添加一个设置视图背景颜色的页面。这个页面还将有一个checkbox演示如何处理DDV验证失败并阻止向导进行到下一页。下面就是新的页面，ID是IDD\_WIZARD\_BKCOLOR：

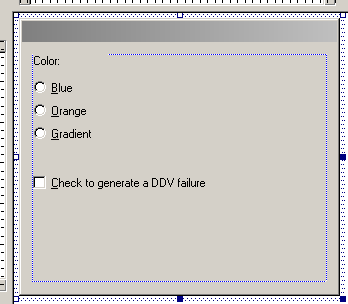


图53 使用DDV的向导式属性页

　　这个类的实现代码在CWizBkColorPage类中，下面是相关的部分代码

|  |
| --- |
| class CWizBkColorPage : public CPropertyPageImpl<CWizBkColorPage>,  public CWinDataExchange<CWizBkColorPage>  {  public:  // some stuff removed for brevity...  BEGIN\_DDX\_MAP(CWizBkColorPage)  　　 DDX\_RADIO(IDC\_BLUE, m\_nColor)  　　 DDX\_CHECK(IDC\_FAIL\_DDV, m\_bFailDDV)  END\_DDX\_MAP()  // Notification handlers  int OnSetActive();  BOOL OnKillActive();  // DDX vars  int m\_nColor;  protected:  int m\_bFailDDV;  }; |

OnSetActive()的工作和前面的介绍页面相同，它使“上一步”和“下一步”按钮可用。OnKillActive()是个新的处理函数，它触发 DDV，然后检查m\_bFailDDV的值，如果是TRUE就表示checkbox处于选中状态，OnKillActive()将阻止向导进行到下一页。

|  |
| --- |
| int CWizBkColorPage::OnSetActive()  {  **SetWizardButtons ( PSWIZB\_BACK | PSWIZB\_NEXT );**  return 0;  }  int CWizBkColorPage::OnKillActive()  {  if ( !DoDataExchange(true) )  　　 return TRUE;　　// prevent deactivation  if ( m\_bFailDDV )  {  　　 MessageBox (\_T("Error box checked, wizard will stay on this page."), \_T("PSheets"), MB\_ICONERROR );  　 return TRUE;　　// prevent deactivation  }  return FALSE;　　　 // allow deactivation  } |

需要注意的是OnKillActive()中做的事情也可以在OnWizardNext()中完成，因为这两个处理函数都可以使向导维持在当前页面。它们的不同之处在于OnKillActive()在用户单击“上一步”和“下一步”按钮时被调用，而OnWizardNext()只是在用户单击“下一步” 按钮时被调用。OnWizardNext()还被用来完成其它目的，比如，它可以直接将向导引导到指定的页面而不是按顺序的下一页。

　　例子工程的向导还有另外两个页面，CWizBkPicturePage 和 CWizFinishPage，由于它们和前面的两个页面相似，我就不再详细介绍它们，想了解它们的细节可以查看源代码。

**8.3.6、其他的界面考虑----置中一个属性表**

　　属性页和向导的默认位置是出现在父窗口的左上角：

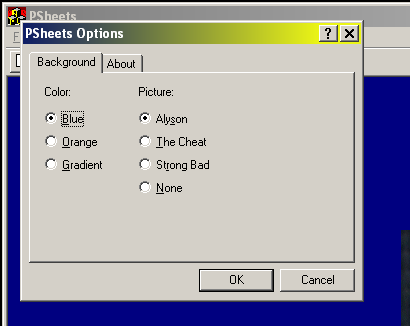


图54 默认属性页的显示

这看起来有点不爽，还好有方法可以补救。第一种方法是重载CPropertySheetImpl::PropSheetCall back()函数，在这个函数中将属性表置中。PropSheetCallback()是MSDN中介绍的PropSheetProc()的回调函数，操作系统在属性表创建时调用这个函数，WTL也是利用在这时子类化属性表窗口的。所以我们的第一种尝试是：

|  |
| --- |
| class CAppPropertySheet : public CPropertySheetImpl<CAppPropertySheet>  {  //...  static int CALLBACK PropSheetCallback(HWND hWnd, UINT uMsg, LPARAM lParam)  {  　　 int nRet = CPropertySheetImpl<CAppPropertySheet>::PropSheetCallback (hWnd, uMsg, lParam );  　　　　if ( PSCB\_INITIALIZED == uMsg )  　　　　{  　　　　　　// center sheet... somehow?  　　　　}  　　　　return nRet;  }  }; |

正如你看到的，我们遇到了棘手的问题。PropSheetCallback()是一个静态方法，不能使用this指针访问属性表窗口。那将这些代码从 CPropertySheetImpl::PropSheetCallback()中拷贝出来，然后添加我们自己的方法行不行呢？撇开刚才将代码和特定版本的WTL联系在一起的方法（这已经被证明不是种好方法），现在代码应该是这样的：

|  |
| --- |
| class CAppPropertySheet : public CPropertySheetImpl<CAppPropertySheet>  {  //...  static int CALLBACK PropSheetCallback(HWND hWnd, UINT uMsg, LPARAM)  {  　　 if(uMsg == PSCB\_INITIALIZED)  　　 {  　　　　 // Code copied from WTL and tweaked to use CAppPropertySheet  　　　　 // instead of T:  　　　　 ATLASSERT(hWnd != NULL);  　　　　 CAppPropertySheet\* pT = (CAppPropertySheet\*)\_Module.ExtractCreateWndData();  　　　 　// subclass the sheet window  　　　 　pT->SubclassWindow(hWnd);  　　　 　// remove page handles array  　　　 　pT->\_CleanUpPages();  　　　 　// Our own code follows:  　　　 　pT->CenterWindow ( pT->m\_psh.hwndParent );  　　 }  　　 return 0;  }  }; |

　　这从理论上讲很完美，但是我试过，属性表的位置并未改变。显然，通用控件的代码在我们调用CenterWindow()之后又改变了属性表窗口的位置。

　　必须放弃这个将代码封装到属性表类的方法，尽管它是个好的解决方案。我又回到原来的方案，即使用属性页窗口和属性表窗口相互协作是属性表窗口置中。我添加了一个用户定义消息UWM\_ CENTER\_SHEET：

|  |
| --- |
| #define UWM\_CENTER\_SHEET WM\_APP |

　　CAppPropertySheet 在它的消息映射链中处理这个消息：

|  |
| --- |
| class CAppPropertySheet : public CPropertySheetImpl<CAppPropertySheet>  {  //...  BEGIN\_MSG\_MAP(CAppPropertySheet)  　　 MESSAGE\_HANDLER\_EX(UWM\_CENTER\_SHEET, OnPageInit)  　　 CHAIN\_MSG\_MAP(CPropertySheetImpl<CAppPropertySheet>)  END\_MSG\_MAP()  // Message handlers  LRESULT OnPageInit ( UINT, WPARAM, LPARAM );  protected:  bool m\_bCentered;　// set to false in the ctor  };  LRESULT CAppPropertySheet::OnPageInit ( UINT, WPARAM, LPARAM )  {  if ( !m\_bCentered )  {  　　 m\_bCentered = true;  　　 CenterWindow ( m\_psh.hwndParent );  }  return 0;  } |

　　然后，每个属性页的OnInitDialog() 方法发送这个消息到属性表窗口：

|  |
| --- |
| BOOL CBackgroundOptsPage::OnInitDialog ( HWND hwndFocus, LPARAM lParam )  {  GetPropertySheet().SendMessage ( UWM\_CENTER\_SHEET );  DoDataExchange(false);  return TRUE;  } |

　　添加m\_bCentered标志确保属性表窗口只响应收到的第一个UWM\_CENTER\_SHEET消息。

**8.3.7、在属性页中添加图标**

如果要使用属性表和属性页的未被成员函数封装的特性，就需要直接访问相关的数据结构：CPropertySheetImpl类中的 PROPSHEETHEADER类型(结构)成员m\_psh和CPropertyPageImpl类中的PROPSHEETPAGE类型(结构)成员 m\_psp。

　　例如：为例子中Option属性表中的Background页面添加一个图标，就需要添加一个标志并设置属性页的PROPSHEETPAGE结构中的几个成员：

|  |
| --- |
| CBackgroundOptsPage::CBackgroundOptsPage()  {  **m\_psp.dwFlags |= PSP\_USEICONID;**  **m\_psp.pszIcon = MAKEINTRESOURCE(IDI\_TABICON);**  **m\_psp.hInstance = \_Module.GetResourceInstance();**  } |

　　下面是这些代码的效果：

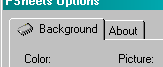


图55 为属性页的标签添加图标

# 第九章 GDI类，通用对话框，初始化类

WTL还有很多封装类和工具类在本系列文章前八篇中并没有介绍，例如CString和CDC，WTL还提供了对GDI对象的良好封装，还包括一些有用的资源装载函数以及WIN32通用对话框的封装类，使得通用对话框更加容易使用。现在，在本章中，我将介绍一些最常用的工具类。

本文将讨论以下内容：

1. GDI 封装类
2. 资源装载(Resource-loading)函数
3. 使用打开文件和选择文件夹的通用对话框
4. 其它有用的类和全局函数

**9.1、GDI 封装类**

WTL使用了与MFC截然不同方式封装GDI对象，WTL的方法是为每种GDI对象设计一个模板类，每个模板都有一个名为t\_bManaged的bool类型模板参数，这个参数控制着这些类是否“管理”它所封装的GDI对象。如果t\_bManaged是false，表示这个C++对象并不管理GDI对象的生命周期，它只是围绕着这个GDI对象句柄的简单封装；如果t\_bManaged是true，就产生了两点不同之处：

1. 如果封装的句柄不为NULL，析构会调用DeleteObject()函数释放资源。
2. 如果封装的句柄不为NULL，Attach()会在捆绑其它新句柄之前调用DeleteObject()释放当前封装的句柄

这种设计与ATL窗口类的封装风格是一致的，ATL的CWindow就是对HWND句柄的一个简单的封装类，而CWindowImpl则负责管理窗口的整个生命周期。

GDI的封装类都定义在**atlgdi.h**中（译者注：注意是“定义在”而不是通常说的 “声明在”头文件中，这是因为ATL/WTL使用的是包含编译模式，所有的代码都在头文件中）, 只有CMenuT例外，它定义在**atluser.h**中。你不需要直接包含这些头文件，因为它们已经包含在*atlapp.h* 中了。每种模板类都由很容易记忆的名字：

|  |  |  |  |
| --- | --- | --- | --- |
| **封装 GDI 对象** | **模板类** | **可管理对象封装** | **简单的句柄封装** |
| Pen | CPenT | CPen | CPenHandle |
| Brush | CBrushT | CBrush | CBrushHandle |
| Font | CFontT | CFont | CFontHandle |
| Bitmap | CBitmapT | CBitmap | CBitmapHandle |
| Palette | CPaletteT | CPalette | CPaletteHandle |
| Region | CRgnT | CRgn | CRgnHandle |
| Device context | CDCT | CDC | CDCHandle |
| Menu | CMenuT | CMenu | CMenuHandle |

相对于MFC围绕着对象指针封装的方法，我更喜欢WTL的方法：你不用总是担心得到一个NULL指针(封装的句柄也可能是NULL，但这是另一回事)，也不用总是注意操作临时对象这种特殊情况（译者注：MFC有一个回收机制，总是试图释放使用FromHandle得到的临时对象），还有一点就是使用这种形式封装的类实例只占用很少的资源，因为类只有一个成员变量，就是其封装的句柄。象这种类似CWindow的封装形式，你可以在不同的线程之间传递封装类对象而不用担心线程安全问题，正是因为这样，WTL也不需要象MFC那样的线程特殊性映射。

下面的设备上下文封装类用于一些特殊的绘制场景:

|  |  |
| --- | --- |
| DC封装类 | 说明 |
| CClientDC | 封装了GetDC()和ReleaseDC（），用于Windows客户区的绘制。 |
| CWindowDC | 封装了GetWindowDC() 和ReleaseDC()，用于在整个Windows上下文中绘制。 |
| CPaintDC | 封装了BeginPaint() 和 EndPaint()，用户WM\_PAINT 消息响应函数。 |

每个类的构造函数都有一个HWND（窗口句柄）参数，类的行为和MFC的同名类相似。三个类都是CDC的派生类，它们自己管理设备上下文。

**9.1.1、封装类的通用函数**

所有的GDI封装类使用的是相同的设计理念，所以它们的使用方法都大同小异，为了简单起见，这里只介绍一下CBitmapT类。

**9.1.1.1、封装 GDI 对象句柄**

每个类都由一个公有成员变量，也就是C++对象封装的GDI对象句柄，CBitmapT有一个HBITMAP类型成员，名为m\_hBitmap。

**9.1.1.2、构造函数**

构造函数有一个HBITMAP类型的参数，默认值是NULL，m\_hBitmap将被初始化为这个值。

**9.1.1.3、析构函数**

如果t\_bManaged 是true，并且m\_hBitmap不是NULL，那么析构函数会调用 DeleteObject()释放这个bitmap。

**9.1.1.4、Attach() 和 operator =**

这两个函数都有一个HBITMAP类型的参数，如果t\_bManaged是true，并且m\_hBitmap不为NULL，它们会调用 DeleteObject()释放这个CBitmapT对象管理的bitmap，然后将m\_hBitmap的值设为作为参数传递进来的那个HBITMAP。

**9.1.1.5、Detach()**

Detach()将m\_hBitmap的值设为NULL，然后返回m\_hBitmap的值，Detach()调用以后， CBitmapT对象就不再关联GDI bitmap了。

**9.1.1.6、创建 GDI 对象的函数**

CBitmapT 封装了几个用来创建位图的WIN32 API ：LoadBitmap()，LoadMappedBitmap()、 CreateBitmap()、CreateBitmapIndirect ()、CreateCompatibleBitmap()、 CreateDiscardable Bitmap()、CreateDIBitmap()、CreateDIBSection()。这些方法将保证m\_hBitmap不是NULL然后返回一个，如果要将这个CBitmapT对象用于其它GDI bitmap，需要首先调用DeleteObject()或Detach() 函数。

**9.1.1.7、DeleteObject()**

DeleteObject() 销毁GDI bitmap对象，将m\_hBitmap设为NULL，调用这个函数时 m\_hBitmap应该不为NULL，否则将会出现断言错误。

**9.1.1.8、IsNull()**

如果m\_hBitmap不为NULL，IsNull() 返回true，否则返回false。使用这个函数可以测试CBitmapT对象是否关联了一个GDI bitmap。

**9.1.1.9、operator HBITMAP**

这个转换操作符返回m\_hBitmap，这样你就可以将CBitmapT对象传递给那些使用HBIMAP句柄作为参数的函数或Win32 API。这个转换操作符还可用在测试这个对象合法性的布尔表达式中，其值与IsNull()刚好相反。例如，下面两个if语句时等价的：

|  |
| --- |
| CBitmapHandle bmp = /\* some HBITMAP value \*/;  if ( !bmp.IsNull() ) { do something... }  if ( bmp ) { do something more... } |

**9.1.1.10、GetObject()封装**

CBitmapT对Win32 API GetObject()有一个类型安全的封装：GetBitmap()，它有两个重载形式，一个使用 LOGBITMAP\* 类型的参数并直接调用GetObject()；另一个使用 LOGBITMAP& 类型的参数并返回一个bool值表示操作是否成功，后一个版本比较容易使用，例如：

|  |
| --- |
| CBitmapHandle bmp2 = /\* some HBITMAP value \*/;  LOGBITMAP logbmp = {0};  bool bSuccess;  if ( bmp2 )  bSuccess = bmp2.GetLogBitmap ( logbmp ); |

**9.1.1.11、对于操作GDI 对象的API的封装**

CBitmapT 封装了操作HBITMAP 的API：GetBitmapBits()、SetBitmapBits ()、GetBitmap Dimension()、SetBitmapDimension()、GetDIBits()和 SetDIBits()，这些函数都对m\_hBitmap是否是NULL进行断言。

**9.1.1.12、其它有用的方法**

CBitmapT 有两个很有用操作m\_hBitmap的函数：LoadOEMBitmap ()和 GetSize()。

**9.2、使用CDCT**

CDCT 与其它类稍有不同，所以这里单独介绍一下这个类。

**9.2.1、方法有哪些不同**

销毁DC时调用DeleteDC() 而不是DeleteObject()。

**9.2.2、将对象选入DC**

MFC的CDC类一大诟病就是给DC选入设备时容易出错，MFC的CDC类对SelectObject()函数有几个不同的重载形式，每种重载都使用一个指向不同GDI封装类的指针作为参数，(CPen\*, CBitmap\*, 等等)。如果你传递一个C++对象而不是对象指针给SelectObject()函数，代码最终将调用一个使用 HGDIOBJ作为参数的重载形式（这个重载形式并未见诸于正式文档），这将导致错误发生。

WTL的 CDCT 采用一种稍好一点的方法，它的几个select函数都是直接使用对应类型的GDI对象：

|  |
| --- |
| HPEN SelectStockPen(int nPen)  HBRUSH SelectStockBrush(int nBrush)  HFONT SelectStockFont(int nFont)  HPALETTE SelectStockPalette(int nPalette, BOOL bForceBackground) |

**9.2.3、与 MFC 封装类的不同之处**

**9.2.3.1、较少的构造函数：**

这些封装类缺少创建新GDI对象的构造函数，例如：MFC的CBrush类可以创建使用实心填充方式和模式填充方式的画刷对象。在WTL中，你必须调用创建方法来创建一个GDI对象。

**9.2.3.2、向DC选入对象做得比较好：**

*可以参考上面 Using CDCT* 一节

**9.2.3.3、没有**< B>m\_hAttribDC：

WTL的CDCT 没有m\_hAttribDC成员。

**9.2.3.4、一些函数的参数稍有不同：**

例如：MFC中的CDC::GetWindowExt() 返回一个CSize ；而WTL版本的函数返回一个bool值，size的值通过输出参数返回。

**9.3、资源装载（Resource-Loading）函数**

WTL有几个全局函数对于装载各种资源很有帮助，在了解这些函数之前先要了解一下这个工具类：\_U\_STRINGorID。

在 Win32平台上，有集中资源既可以用字符串标识（LPCTSTR），也可以用无符号整形数标识（UINT），那些使用资源的API 都使用一个LPCTSTR类型的参数作为资源标识，如果你想传递一个UINT，你需要使用 MAKEINTRESOURCE宏将其转换成LPCTSTR。\_U\_STRINGorID就是扮演一个转换资源标识类型的角色，它隐藏了两种类型的区别，函数调用者只需要直接传递UINT或 LPCTSTR 就可以了，这个函数在必要的时候使用CString 装载字符串：

|  |
| --- |
| void somefunc ( \_U\_STRINGorID id )  {  CString str ( id.m\_lpstr );  // use str...  }    void func2()  {  // Call 1 - using a string literal  somefunc ( \_T("Willow Rosenberg") );    // Call 2 - using a string resource ID  somefunc ( IDS\_BUFFY\_SUMMERS );  } |

这样使用之所以有效是因为CString的构造函数会检查LPCTSTR参数是不是一个字符串ID，如果是就从字符串资源表中装载这个字符串并赋值给CString。

在 VC 6版本中\_U\_STRINGorID由WTL提供，定义在 *atlwinx.h*中，在 VC 7版本中，\_U\_STRINGorID 成为ATL的一部分，不过，对于使用没有区别，因为无论何种方式它都已经包含在你的ATL/WTL项目的头文件中了。

本节中介绍的函数从本地资源句柄装载资源，这个本地资源句柄保存在全局变量\_Module（in VC 6）或者是全局变量\_AtlBaseModule(in VC 7)中。使用其它模块的资源已经超出了本文的范畴，这里就不再介绍了。不过请记住，这些函数只能装载代码所在的EXE或DLL中的资源，它们仅仅是使用 \_U\_STRINGorID带来的简化手段调用Win32的API而已。

|  |
| --- |
| HACCEL AtlLoadAccelerators(\_U\_STRINGorID table) |

调用 LoadAccelerators()。

|  |
| --- |
| HMENU AtlLoadMenu(\_U\_STRINGorID menu) |

调用 LoadMenu()。

|  |
| --- |
| HBITMAP AtlLoadBitmap(\_U\_STRINGorID bitmap) |

调用 LoadBitmap()。

|  |
| --- |
| HCURSOR AtlLoadCursor(\_U\_STRINGorID cursor) |

调用 LoadCursor()。

|  |
| --- |
| HICON AtlLoadIcon(\_U\_STRINGorID icon) |

调用 LoadIcon()。注意，这个函数和LoadIcon()一样只装载32x32的图标。

|  |
| --- |
| int AtlLoadString(UINT uID, LPTSTR lpBuffer, int nBufferMax)  bool AtlLoadString(UINT uID, BSTR& bstrText) |

调用LoadString()。字符串可以返回在 TCHAR中，也可以指定给一个BSTR，这要看你调用的是哪一个重载版本。注意，这个函数只接受 UINT类型的资源ID，因为字符串表资源不能用字符串作为标识。

下面这一组函数封装了对LoadImage()的调用，使用一个额外的参数传递给LoadImage()。

|  |
| --- |
| HBITMAP AtlLoadBitmapImage(\_U\_STRINGorID bitmap, UINT fuLoad = LR\_DEFAULTCOLOR) |

调用LoadImage()，使用IMAGE\_BITMAP类型，通过fuLoad传递标志字段。

|  |
| --- |
| HCURSOR AtlLoadCursorImage( \_U\_STRINGorID cursor, UINT fuLoad = LR\_DEFAULTCOLOR | LR\_DEFAULTSIZE,  int cxDesired = 0, int cyDesired = 0) |

调用LoadImage()，使用IMAGE\_CURSOR 类型，使用fuLoad 标志。由于一个图标可能有几个不同大小的图标组成，所以另外两个参数用于指定所要装载图标的大小。

|  |
| --- |
| HICON AtlLoadIconImage(\_U\_STRINGorID icon, UINT fuLoad = LR\_DEFAULTCOLOR | LR\_DEFAULTSIZE,  int cxDesired = 0, int cyDesired = 0) |

调用LoadImage()，使用IMAGE\_ICON 类型，通过fuLoad 传递标志字段。cxDesired 和cyDesired 参数与AtlLoadCursorImage()函数作用一样。

下面这一组函数封装了一些操作系统定义资源的API(例如，标准的手形鼠标指针)。默认情况下并不包含其中的一些资源ID (大多数是位图资源)，你需要在stdafx.h 中定义OEMRESOURCE 标号才能使用它们。

|  |
| --- |
| HBITMAP AtlLoadSysBitmap(LPCTSTR lpBitmapName) |

使用NULL资源句柄调用LoadBitmap()，使用这个函数可以装载LoadBitmap()帮助文档中列举的一些 OBM\_\*位图。

|  |
| --- |
| HCURSOR AtlLoadSysCursor(LPCTSTR lpCursorName) |

使用NULL资源句柄调用LoadCursor()，使用这个函数可以装载 LoadCursor()帮助文档中列举的IDC\_\*图标

|  |
| --- |
| HICON AtlLoadSysIcon(LPCTSTR lpIconName) |

使用NULL资源句柄调用LoadIcon()，使用这个函数可以装载 LoadIcon() 帮助文档中列举的一些IDI\_\* 图标。需要注意的是这个函数和LoadIcon()一样只装载 32x32 大小的图标。

|  |
| --- |
| HBITMAP AtlLoadSysBitmapImage(WORD wBitmapID, UINT fuLoad = LR\_DEFAULTCOLOR) |

使用NULL资源句柄调用LoadImage()，装载类型为IMAGE\_BITMAP ，可以使用这个函数装载AtlLoadSysBitmap()能够装载的位图。

|  |
| --- |
| HCURSOR AtlLoadSysCursorImage(\_U\_STRINGorID cursor, UINT fuLoad = LR\_DEFAULTCOLOR|  LR\_DEFAULTSIZE, int cxDesired = 0, int cyDesired = 0) |

使用NULL资源句柄调用LoadImage() ，装载类型为IMAGE\_CURSOR，可以使用这个函数装载AtlLoadSysCursor())能够装载的图标。

|  |
| --- |
| HICON AtlLoadSysIconImage(\_U\_STRINGorID icon,UINT fuLoad = LR\_DEFAULTCOLOR | LR\_DEFAULTSIZE,  int cxDesired = 0, int cyDesired = 0) |

使用NULL资源句柄调用LoadImage() ，装载类型为IMAGE\_ICON，可以使用这个函数装载AtlLoadSysIcon ()能够装载的图标，不过只能指定不同的大小，比如16x16。

最后这组函数提供了对GetStockObject() API的类型安全封装。

|  |
| --- |
| HPEN AtlGetStockPen(int nPen)  HBRUSH AtlGetStockBrush(int nBrush)  HFONT AtlGetStockFont(int nFont)  HPALETTE AtlGetStockPalette(int nPalette) |

这个函数首先将指定的参数转换成对GDI对象有效的类型 (比如AtlGetStockPen() 只接受WHITE\_PEN和BLACK\_PEN, 等等)，然后直接调用GetStockObject()。

**9.4、使用通用对话框**

WTL还提供了一些类用于简化对Win32 通用对话框地使用，这些类响应通用对话框发送的消息和回调函数，依次调用重载的消息处理函数。这种设计方式和属性页非常相似，你需要为每个属性页提供单独的通知消息响应函数（比如，OnWizardNext() 处理 PSN\_WIZNEXT），它们在必要的时候由CPropertyPageImpl调用。

WTL为每个通用对话框提供两个类，例如：选择文件夹对话框由CFolderDialogImpl 和 CFolder Dialog两个类封装。如果你想改变它们的默认行为或为某个消息定制一个独特的响应函数，就需要从CFolderDialogImpl派生一个新类，在类中做相应的修改。如果默认的 CFolderDialogImpl够用了，你就可以使用CFolderDialog。

通用对话框和对应的WTL类：

|  |  |  |  |
| --- | --- | --- | --- |
| 通用对话框 | 相应的 Win32 API | 实现类 | 非定制化类 |
| File Open and File Save | GetOpenFileName(),  GetSaveFileName() | CFileDialogImpl | CFileDialog |
| Choose Folder | SHBrowseForFolder() | CFolderDialogImpl | CFolderDialog |
| Choose Font | ChooseFont() | CFontDialogImpl, CRichEditFontDialogImpl | CFontDialog, CRichEditFontDialog |
| Choose Color | ChooseColor() | CColorDialogImpl | CColorDialog |
| Printing and Print Setup | PrintDlg() | CPrintDialogImpl | CPrintDialog |
| Printing (Windows 2000 and later) | PrintDlgEx() | CPrintDialogExImpl | CPrintDialogEx |
| Page Setup | PageSetupDlg() | CPageSetupDialogImpl | CPageSetupDialog |
| Text find and replace | FindText(),  ReplaceText() | CFindReplaceDialogImpl | CFindReplaceDialog |

介绍所有的类会使本文变成超级长文章，本文只介绍前两个最经常使用的类。

**9.4.1、CFileDialog类**

CFileDialog类和 CFileDialogImpl类（译者注：一个是接口类，一个是实现类）用于显示文件打开和保存对话框，CFileDialogImpl类中最重要的两个成员是m\_ofn和m\_szFileName。m\_ofn 是一个OPENFILENAME结构，和MFC一样， CFileDialogImpl 使用一些有意义的默认值填充这个结构，如果有必要你可以直接操作这个成员修改其中的属性。m\_szFileName 是一个TCHAR 数组，用来保存选择的文件名。(CFileDialogImpl只有一个字符串缓冲区保存文件名，如果要选择多个文件需要指定你自己的缓冲区)。

使用CFileDialog 的基本步骤：

1. 创建一个CFileDialog对象，通过构造函数传递一些初始数据。
2. 调用 DoModal()。
3. 如果 DoModal() 返回IDOK，在m\_szFileName中得到文件名。

下面是CFileDialog类的构造函数：

|  |
| --- |
| CFileDialog::CFileDialog ( BOOL bOpenFileDialog, LPCTSTR lpszDefExt = NULL,LPCTSTR lpszFileName = NULL,  DWORD dwFlags = OFN\_HIDEREADONLY |OFN\_OVERWRITEPROMPT, LPCTSTR lpszFilter = NULL,  HWND hWndParent = NULL ) |

创建打开文件对话框需要指定bOpenFileDialog为true(CFileDialog将调用GetOpenFile Name()显示对话框)，创建文件保存对话框需要指定bOpenFileDialog为false (CFileDialog 调用GetSaveFileName())。其它参数对应着m\_ofn结构中的成员，它们是可选的参数，因为你可以在调用DoModal()之前直接操作m\_ofn修改这些值。

与MFC的CFileDialog有一点显著的不同，那就是lpszFilter参数必须是用null字符分隔的字符串列表(格式和 OPENFILENAME 文档中说明的一样)，而不是用“|”分隔的字符串列表。

下面的例子演示了使用带有filter的CFileDialog选择 Word 12 文件(\*.docx)（译者注： office 2007）：

|  |
| --- |
| CString sSelectedFile;  CFileDialog fileDlg ( true, \_T("docx"), NULL,OFN\_HIDEREADONLY | OFN\_FILEMUSTEXIST,  \_T("Word 12 Files\0\*.docx\0All Files\0\*.\*\0") );    if ( IDOK == fileDlg.DoModal() )  sSelectedFile = fileDlg.m\_szFileName; |

CFileDialog 类对本地化的支持不是很好，那是因为构造函数使用 LPCTSTR类型的参数，不仅如此，filter字符串处理起来也很蹩脚。有两个解决方案，一是直接操作m\_ofn，另一个是从 CFile DialogImpl派生新类。这里我们采用第二种方式，派生一个新类，然后做如下修改：

1. 构造函数中的字符串参数使用\_U\_STRINGorID 代替LPCTSTR。
2. 和MFC一样，filter 字符串改用“|”分隔，而不是null字符。
3. 对话框相对于父窗口自动居中。

我们开始编写一个新类，它的构造函数的参数和CFileDialogImpl的构造函数相似：

|  |
| --- |
| class CMyFileDialog : public CFileDialogImpl<CMyFileDialog>  {  public:  // Construction  CMyFileDialog ( BOOL bOpenFileDialog, \_U\_STRINGorID szDefExt = 0U, \_U\_STRINGorID szFileName = 0U,  DWORD dwFlags = OFN\_HIDEREADONLY | OFN\_OVERWRITEPROMPT,  \_U\_STRINGorID szFilter = 0U,HWND hwndParent = NULL );    protected:  LPCTSTR PrepFilterString ( CString& sFilter );  CString m\_sDefExt, m\_sFileName, m\_sFilter;  }; |

构造函数初始化三个CString 成员，必要时可能从资源中装载字符串：

|  |
| --- |
| CMyFileDialog::CMyFileDialog ( BOOL bOpenFileDialog, \_U\_STRINGorID szDefExt, \_U\_STRINGorID szFileName,  DWORD dwFlags, \_U\_STRINGorID szFilter, HWND hwndParent ) :  CFileDialogImpl<CMyFileDialog>(bOpenFileDialog, NULL, NULL, dwFlags, NULL, hwndParent),  m\_sDefExt(szDefExt.m\_lpstr), m\_sFileName(szFileName.m\_lpstr), m\_sFilter(szFilter.m\_lpstr)  {  } |

注意一点，这三个字符串在调用基类的构造函数的时候都是空的，这是因为基类的构造函数是在三个字符串初始化之前调用的，要设置m\_ofn中的字符串数据，我们需要添加一些代码将 CFileDialogImpl 构造函数中的初始化步骤重做一遍：

|  |
| --- |
| CMyFileDialog::CMyFileDialog(...)  {  **m\_ofn.lpstrDefExt = m\_sDefExt;**  **m\_ofn.lpstrFilter = PrepFilterString ( m\_sFilter );**    **// setup initial file name**  **if ( !m\_sFileName.IsEmpty() )**  **lstrcpyn ( m\_szFileName, m\_sFileName, \_MAX\_PATH );**  } |

PrepFilterString() 是一个辅助函数，将的filter字符串中的“|”分隔转换成null字符，结果就是将“|”分隔的filter字符串转换成 OPENFILENAME所需要的格式。

|  |
| --- |
| LPCTSTR CMyFileDialog::PrepFilterString(CString& sFilter)  {  LPTSTR psz = sFilter.GetBuffer(0);  LPCTSTR pszRet = psz;    while ( '\0' != \*psz )  {  if ( '|' == \*psz )  \*psz++ = '\0';  else  psz = CharNext ( psz );  }    return pszRet;  } |

这些转换简化了字符串的处理。要实现窗口的自动居中显示，我们需要重载OnInitDone()，这需要添加消息映射(这样我们能够链接到基类的通知消息)，下面是我们的OnInitDone()处理函数：

|  |
| --- |
| class CMyFileDialog : public CFileDialogImpl<CMyFileDialog>  {  public:  // Construction  CMyFileDialog(...);    // Maps  BEGIN\_MSG\_MAP(CMyFileDialog)  CHAIN\_MSG\_MAP(CFileDialogImpl<CMyFileDialog>)  END\_MSG\_MAP()    // Overrides  void OnInitDone ( LPOFNOTIFY lpon )  {  GetFileDialogWindow().CenterWindow(lpon->lpOFN->hwndOwner);  }    protected:  LPCTSTR PrepFilterString ( CString& sFilter );  CString m\_sDefExt, m\_sFileName, m\_sFilter;  }; |

关联到CMyFileDialog 对象的窗口实际上是文件打开对话框的一个子窗口，因为我们需要窗口队列的顶层窗口，所以调用 GetFileDialogWindow()得到这个顶层窗口。

**9.4.2、CFolderDialog类**

CFolderDialog和CFolderDialogImpl类用来显示一个浏览文件夹的对话框，Windows的文件夹浏览对话框能够查看整个外壳名字空间（shell namespace）的任何位置，但是CFolderDialog，只支持浏览文件系统。CFolderDialogImpl中最重要的两个数据成员是m\_bi 和 m\_sz FolderPath，m\_bi 一个 BROWSEINFO 类型的数据结构，它由CFolderDialogImpl 负责维护并作为参数传递给SHBrowseForFolder() API，必要时可以直接修改这个数据结构，m\_szFolderPath 是一个TCHAR 类型的数组，它存放选中的文件夹全名。

使用CFolderDialog的步骤是：

1. 创建一个CFolderDialog对象，通过构造函数传递初始数据。
2. 调用 DoModal()。
3. 如果DoModal() 返回IDOK，就可以从m\_szFolderPath获得文件夹名称。

下面是CFolderDialog的构造函数：

|  |
| --- |
| CFolderDialog::CFolderDialog (HWND hWndParent = NULL, LPCTSTR lpstrTitle = NULL,  UINT uFlags = BIF\_RETURNONLYFSDIRS ) |

hWndParent 是浏览对话框的拥有者窗口，可以通过构造函数在创建时指定拥有者窗口，也可以在调用DoModal() 时通过这个函数的参数指定拥有者窗口。lpstrTitle 是显示在浏览窗口中树控件上方的文字标签，uFlags 是一个标志，它决定了浏览对话框的行为。uFlag应该总是包括 BIF\_RETURNONLYFSDIRS属性，这样树控件就只显示文件系统的目录，有关这个标志的其它情况可以查阅关于 BROWSEINFO数据结构的帮助文档，不过有一点需要了解，那就是并不是所有的标志属性都会产生好的作用，比如 BIF\_BROWSEFORPRINTER。不过与UI相关的一些标志工作的很好，比如BIF\_USENEWUI。还有一点就是构造函数中的 lpstrTitle参数在使用时会有点小问题。

下面是使用CFolderDialog选择目录的例子：

|  |
| --- |
| CString sSelectedDir;  CFolderDialog fldDlg ( NULL, \_T("Select a dir"), BIF\_RETURNONLYFSDIRS|BIF\_NEWDIALOGSTYLE );    if ( IDOK == fldDlg.DoModal() )  sSelectedDir = fldDlg.m\_szFolderPath; |

现在演示一下如何使用定制的CFolderDialog，我们从CFolderDialogImpl类派生一个新类并设置初始选择，由于这个对话框的回调不使用Windows消息，所以新类也不需要消息映射链，只需重载OnInitialized()函数即可，这个函数在基类接收到 BFFM\_INITIALIZED 通知消息时被调用，On Initialized()调用CFolderDialogImpl::SetSelection()改变对话框的初始选择。

|  |
| --- |
| class CMyFolderDialog : public CFolderDialogImpl<CMyFolderDialog>  {  public:  // 构造函数  CMyFolderDialog ( HWND hWndParent = NULL, \_U\_STRINGorID szTitle = 0U,  UINT uFlags = BIF\_RETURNONLYFSDIRS ) :  CFolderDialogImpl<CMyFolderDialog>(hWndParent, NULL, uFlags), m\_sTitle(szTitle.m\_lpstr)  {  m\_bi.lpszTitle = m\_sTitle;  }    **// 重载**  **void OnInitialized()**  **{**  **// 设置Windows目录的初始选项**  **TCHAR szWinDir[MAX\_PATH];**    **GetWindowsDirect ory ( szWinDir, MAX\_PATH );**  **SetSelection ( szWinDir );**  **}**    protected:  CString m\_sTitle;  }; |

**9.5、其它有用的类和全局函数**

**9.5.1、对结构的封装**

和MFC一样，WTL 也对SIZE、POINT和RECT数据结构进行了封装，分别是CSize、 CPoint和CRect类。

**9.5.2、处理双类型参数的类**

就像前面提到的那样，你可以使用\_U\_STRINGorID 去自适应那些参数是数字或字符串资源ID的函数，WTL中还有两个类和这个类的作用类似：

|  |  |
| --- | --- |
| 类型 | 说明 |
| \_U\_MENUorID | 这个类型支持UINT 或 HMENU，通常用在CreateWindow() 的封装中函数中，hMenu 参数在某些情况下是菜单句柄，但是在创建子窗口时它又是一个窗口ID，\_U\_MENUorID 用来消除（隐藏）这些差异，\_U\_MENUorID 有一个m\_hMenu成员，用来向CreateWindow() 或 CreateWindowEx()传递hMenu参数。 |
| \_U\_RECT | 这个类可以从LPRECT 或RECT&构建，可以将RECT 数据结构，RECT指针或象CRect那样的封装类转换成很对函数需要的 RECT类型参数。 |

和\_U\_STRINGorID一样，\_U\_MENUorID 和\_U\_RECT 也已经随着其它头文件包含在你的工程中了。

**9.6、其它工具类**

**9.6.1、CString类**

WTL 的CString和MFC的CString类似，所以这里就不用详细介绍了, 不过，WTL的CString 还多了了一些额外的特性，这些特性在你使用\_ATL\_MIN\_CRT方式编译代码的时候就显得十分有用，比如，\_cstrchr() 和 \_cstrstr()函数。当不使用 \_ATL\_MIN\_CRT方式编译时它们会被相应的CRT函数取代，不会产生额外的代码。（译者注： 使用\_ATL\_MIN\_CRT方式编译是为了减少对CRT库的依赖，从而产生较小的可执行文件，不过这样就不能使用很多CRT的标准函数，比如 strstr、strchr等等，在这种情况下WTL的CString会使用自己的函数代替，从而保证CString能够正常工作，当不使用 \_ATL\_MIN\_CRT方式时，CString会直接调用CRT库函数）

**9.6.2、CFindFile类**

CFindFile 封装了FindFirstFile()和FindNextFile() APIs，它比MFC的CFileFind还要容易使用一些，使用方式可以参考下面的模式：

|  |
| --- |
| CFindFile finder;  CString sPattern = \_T("C:\\windows\\\*.exe");    if ( finder.FindFirstFile ( sPattern ) )  {  do  {  // act on the file that was found  }while ( finder.FindNextFile() );  }    finder.Close(); |

如果FindFirstFile() 返回true，就表示至少有一个文件匹配查找模式，在循环内，你可以访问CFindFile 的公有成员m\_fd，这是一个WIN32\_FIND\_DATA 数据结构，包含了这个文件的信息，循环可以一直继续直到FindNextFile() 返回false，这表示你已经把所有的文件遍历了一遍。

为了使用方便，CFindFile还提供了一些操作m\_fd的函数，这些函数的返回值只在成功调用了 FindFirstFile或FindNextFile()之后才有意义。

|  |
| --- |
| ULONGLONG GetFileSize() |

返回文件的大小，数据类型时64位无符号整形数。

|  |
| --- |
| BOOL GetFileName(LPTSTR lpstrFileName, int cchLength)  CString GetFileName() |

得到查找到文件的名字和扩展名(从m\_fd.cFileName复制数据)。

|  |
| --- |
| BOOL GetFilePath(LPTSTR lpstrFilePath, int cchLength)  CString GetFilePath() |

返回查找到文件的全路径。

|  |
| --- |
| BOOL GetFileTitle(LPTSTR lpstrFileTitle, int cchLength)  CString GetFileTitle() |

返回文件的标题 (就是没有扩展名)。

|  |
| --- |
| BOOL GetFileURL(LPTSTR lpstrFileURL, int cchLength)  CString GetFileURL() |

创建一个file:// URL，包含文件的全路径。（译者注：比如“file://d:\doc\sss.doc”）

|  |
| --- |
| BOOL GetRoot(LPTSTR lpstrRoot, int cchLength)  CString GetRoot() |

得到文件所在的目录。

|  |
| --- |
| BOOL GetLastWriteTime(FILETIME\* pTimeStamp)  BOOL GetLastAccessTime(FILETIME\* pTimeStamp)  BOOL GetCreationTime(FILETIME\* pTimeStamp) |

这些函数从m\_fd的数据成员ftLastWriteTime、ftLastAccessTime和ftCreation Time 复制数据。

CFindFile 还有一些辅助函数用于检查文件的属性。

|  |
| --- |
| BOOL IsDots() |

如果文件是"." 或 ".." 目录就返回true。

|  |
| --- |
| BOOL MatchesMask(DWORD dwMask) |

将文件的属性和dwMask (通常是一些FILE\_ATTRIBUTE\_\* 属性的组合)比较，看看查找到的文件是是否有指定的属性如果文件属性包含dwMask指定的位掩码，就返回true。

|  |
| --- |
| BOOL IsReadOnly()  BOOL IsDirectory()  BOOL IsCompressed()  BOOL IsSystem()  BOOL IsHidden()  BOOL IsTemporary()  BOOL IsNormal()  BOOL IsArchived() |

这些函数是MatchesMask() 函数的更直观的替代者，它们通常只是测试属性中的某一个，比如，IsReadOnly() 就是调用MatchesMask(FILE\_ATTRIBUTE\_READONLY)。

**9.7、全局函数**

WTL 还有一些很有用的全局函数，比如检查DLL版本或者显示一个消息框窗口。

|  |
| --- |
| bool AtlIsOldWindows() |

判断Windows的版本是否太老，如果是Windows 95、 98、 NT 3 或 NT 4这样的系统就会返回true。

|  |
| --- |
| HFONT AtlGetDefaultGuiFont() |

返回值和调用GetStockObject(DEFAULT\_GUI\_FONT)的返回值相同，在英文版的Windows 2000 或更新的版本 (也包括一些使用拉丁字母的单字节语言)中，这个字库的名字（face name）是“MS Shell Dlg”。这种字体对于对话框效果很好，但是如果你要在界面上创建自己的字体，这就不是一个很好的选择。MS Shell Dlg 就是 MS Sans Serif的别名, 而不是使用新字体Tahoma。要避免使用MS Sans Serif，你可以通过消息框得到字体：

|  |
| --- |
| NONCLIENTMETRICS ncm = { sizeof(NONCLIENTMETRICS) };  CFont font;    if ( SystemParametersInfo ( SPI\_GETNONCLIENTMETRICS, 0, &ncm, false ) )  font.CreateFontIndirect ( &ncm.lfMessageFont ); |

另一种方法是检查AtlGetDefaultGuiFont()返回的字体名称，如果是“MS Shell Dlg”就将其改成“MS Shell Dlg 2”，它将使用新字体 Tahoma。

|  |
| --- |
| HFONT AtlCreateBoldFont(HFONT hFont = NULL) |

创建指定字体的加粗版本，如果hFont是NULL，AtlCreateBoldFont()就创建一个通过Atl GetDefaultGuiFont()得到的字体的加粗版本。

|  |
| --- |
| BOOL AtlInitCommonControls(DWORD dwFlags) |

这是对InitCommonControlsEx()API的封装，使用一些指定的标志初始化INITCOMMON CONTROLSEX 结构，然后调用InitCommonControlsEx()。

|  |
| --- |
| HRESULT AtlGetDllVersion(HINSTANCE hInstDLL, DLLVERSIONINFO\* pDllVersionInfo)  HRESULT AtlGetDllVersion(LPCTSTR lpstrDllName, DLLVERSIONINFO\* pDllVersionInfo) |

这两个函数在指定的模块种查找名为DllGetVersion()的导出函数，如果函数存在就调用这个函数，如果调用成功就返回一个DLLVERSIONINFO结构的版本信息。

|  |
| --- |
| HRESULT AtlGetCommCtrlVersion(LPDWORD pdwMajor, LPDWORD pdwMinor) |

返回comctl32.dll的主版本号和次版本号。

|  |
| --- |
| HRESULT AtlGetShellVersion(LPDWORD pdwMajor, LPDWORD pdwMinor) |

返回shell32.dll的主版本号和次版本号。

|  |
| --- |
| bool AtlCompactPath(LPTSTR lpstrOut, LPCTSTR lpstrIn, int cchLen) |

将一个文件名截断，从而使其长度小于cchLen，在结尾添加省略号，它和shlwapi.dll中的PathCompactPath()和PathSetDlgItemPath()功能相似。

|  |
| --- |
| int AtlMessageBox(HWND hWndOwner, \_U\_STRINGorID message, \_U\_STRINGorID title = NULL,  UINT uType = MB\_OK | MB\_ICONINFORMATION) |

和MessageBox()一样，显示一个消息框，但是使用了\_U\_STRINGorID 参数，这样你就可以传递字符串资源ID作为参数，AtlMessageBox()会自动装载字符串资源。

**9.8、宏**

在WTL 的头文件中你会看到各种各样的预处理宏，大多数的宏都可以在编译选项中设置，从而改变WTL代码的某些行为。

以下几个宏与编译设置紧密相关，在WTL的代码中到处都有它们的身影：

**9.8.1、\_WTL\_VER**

对于 WTL 7.1 被定义为 0x0710。

**9.8.2、\_ATL\_MIN\_CRT**

如果这个宏被定义了，ATL将不链接C标准库，由于很多WTL的类（尤其是CString）都要使用C标准库函数，很多特殊的代码被编译进来以代替CRT函数。

**9.8.3、\_ATL\_VER**

对于VC6，这个版本是0x0300，对于VC7，这个版本是0x0700，对于VC8，这个版本是 0x0800。

**9.8.4、\_WIN32\_WCE**

是否编译的是 Windows CE 二进制映象，一些WTL代码因为Windows CE不支持相应的特性而不可用。

下面的宏默认是不定义的，要使用它们需要在 *stdafx.h*文件中所有#include语句之前定义它们。

**9.8.5、\_ATL\_NO\_OLD\_NAMES**

这个宏只在维护WTL 3的代码时起作用，它增加了很多编译检测用于识别两个老的类名和函数名：CUpdateUIObject 已经改名为 CIdleHandler，DoUpdate()也改名为OnIdle()。

**9.8.6、\_ATL\_USE\_CSTRING\_FLOAT**

定义这个标号可以使CString支持浮点运算，它不能和\_ATL\_MIN\_CRT一起使用，如果想在 CString::Format()函数中使用%I64格式，就必须定义这个选项。如果定义了 \_ATL\_USE\_CSTRING\_ FLOAT 标号，CString::Format()将调用\_vstprintf()，这个函数能够识别%I64 格式。

**9.8.7、\_ATL\_USE\_DDX\_FLOAT**

定义这个标号可以使 DDX 代码支持浮点运算，它不能和\_ATL\_MIN\_CRT同时使用。

**9.8.8、\_ATL\_NO\_MSIMG**

定义这个标号将使编译器忽略#pragma comment(lib, "msimg32") 代码行，也就是禁止CDCT 使用 msimg32 函数: AlphaBlend()、TransparentBlt()和GradientFill()。

**9.8.9、\_ATL\_NO\_OPENGL**

定义这个标号编译器将忽略#pragma comment(lib, "opengl32") 代码行，也就是禁止CDCT 使用 OpenGL。

**9.8.10、\_WTL\_FORWARD\_DECLARE\_CSTRING**

已经废弃，使用\_WTL\_USE\_CSTRING 代替。

**9.8.11、\_WTL\_USE\_CSTRING**

定义这个标号将向前声明CString类，这样，那些在 *atlmisc.h* 文件之前包含的头文件也可以使用CString。

**9.8.12、\_WTL\_NO\_CSTRING**

定义这个标号将不能使用WTL::CString。

**9.8.13、\_WTL\_NO\_AUTOMATIC\_NAMESPACE**

定义这个标号将阻止直接使用WTL命名空间（namespace）。

**9.8.14、\_WTL\_NO\_AUTO\_THEME**

定义这个标号阻止CMDICommandBarCtrlImpl 使用 XP 主题。

**9.8.15、\_WTL\_NEW\_PAGE\_NOTIFY\_HANDLERS**

定义这个标号可以在CPropertyPage中使用较新的PSN\_\* 通知消息响应函数，因为老的WTL 3的消息响应函数已经废弃掉了，这个标号应该总是被定义，除非你是在维护老的WTL3 代码。

**9.8.16、\_WTL\_NO\_WTYPES**

定义这个标号将禁止使用WTL封装的CSize、CPoint和CRect类。

**9.8.17、\_WTL\_NO\_THEME\_DELAYLOAD**

在VC6中编译代码时，定义这个标号将阻止*uxtheme.dll* 被自动标记为延时加载。

注意: 如果\_WTL\_USE\_CSTRING 和\_WTL\_NO\_CSTRING 同时定义，产生的结果就只能在*atlmisc.h* 文件包含之后使用CString。

**9.9、例子工程**

本文的演示工程是一个下载工具，这个名为Kibbles的下载工具演示了几个本文介绍的类。这个下载工具使用了BITS（[background intelligent transfer service](http://msdn.microsoft.com/library/en-us/bits/bits/bits_start_page.asp)）组件，Windows 2000及其以后的操作系统都支持这个组件，也就是说这个程序只能运行在基于NT技术的操作系统上，所以我就将其创建成了Unicode工程。

这个程序有一个视图窗口，这个视图窗口用来显示下载过程，它使用了很多GDI函数，也包括专门画饼图的Pie()函数。第一次运行时，程序的初始界面是这样的：



图56 程序初始界面

你可以从浏览器中拖一个链接到这个窗口中，程序会创建一个新的BITS并将链接指定的目标下载到“我的文档”文件夹。当然也可以单击工具栏上第三个按钮直接添加一个URL，工具栏上的第四个按钮则允许你修改默认的下载文件存放位置。

当一个下载任务正在进行，Kibbles会显示一些下载任务的细节，下载的过程显示如下：

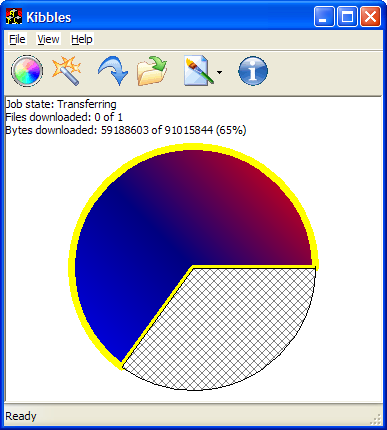


图57 显示下载细节

工具栏上的前两个按钮用来修改过程显示的颜色，第一个按钮会打开一个选项对话框，在选项对话框中可以设置过程饼图中各部分的颜色：

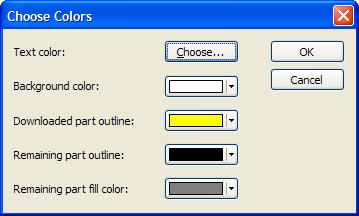


图58 设置过程饼图各部分的颜色

对话框中使用了Tim Smith的文章“Color Picker for WTL with XP themes”中介绍的一个很棒的按钮类，可以查看Kibbles工程中的CChooseColorsDlg 类的代码了解这个按钮类是如何工作的。*“Text color”* 按钮是一个普通的按钮，它的响应函数OnChooseTextColor()演示了如何使用WTL的 CColorDialog类。第二个工具栏按钮的功能是使用随即颜色显示下载过程。

第五个工具栏按钮用来设置背景图片，Kibbles使用这个图片在饼图上显示下载过程。默认的图片程序资源中包含的一个图片，你也可以选择任何BMP位图文件作为背景图片：

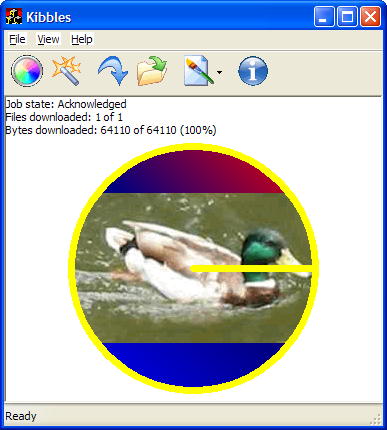


图59 BMP位图背景

CMainFrame::OnToolbarDropdown() 的代码响应按钮的press事件并显示一个弹出式菜单，这个函数还使用了CFindFile 类遍历“我的文档”文件夹。关于各种GDI函数的用法可以查看CKibblesView::OnPaint()函数的代码。

关于工具栏有一点需要特别注意：这个工具栏使用了256色的位图，不过VC的工具栏编辑器只支持16色位图，如果你使用工具栏编辑器修改过工具栏位图，你的工具栏位图就会被转换成16色。我的建议是，在另一个目录中保存这个高彩色的位图，使用图像编辑工具直接编辑这个图片，然后在 “res”目录中另存一个256色的版本。

# 第十章 支持拖放操作

**10.1、简介**

拖放是许多流行应用的特性之一。尽管实现一个放下目标相当简单，但拖动源却要复杂的多。MFC 中有两个类COleDataObject和COleDropSource可以帮助管理拖动源所必须提供的数据，但WTL 中没有这种辅助类。对于我们这些 WTL 用户来说，幸运的是，Raymond Chen 在 2000 年的时候在 MSDN 上写过一篇文章（“The Shell Drag/Drop Helper Object Part 2”），其中有IDataObject的纯 C++ 实现，这对于为 WTL 应用编制一个完整的拖放源提供了巨大的帮助。

本文的示例工程是一个CAB文件查看器，可以使你从CAB中提取文件，只要把它们从查看器里拖到资源浏览器窗口中即可。本文还会讨论几个新的框架窗口话题，例如对文件打开的处理以及类似于 MFC 的文档/视图框架的数据管理。我还会演示WTL的 MRU文件列表类，以及第六版的列表视图控件的几个新的UI特性。

**重要提示**：你需要从微软下载并安装CAB SDK来编译示例代码。在KB文章Q310618 中有此SDK的链接。示例工程假定SDK位于和源代码相同的路径下的名为“cabsdk”的目录中。

记住，如果你在安装WTL或者编译示例代码时遇到了问题，请在这儿提问之前先阅读第一部分的readme一节。

**10.1.1、开始工程**

要开始我们的CAB查看器应用，需要运行WTL应用程序向导并创建一个名为 *WTLCabView* 的工程。它应该是一个SDI应用，所以在第一页中选择SDI 应用程序：

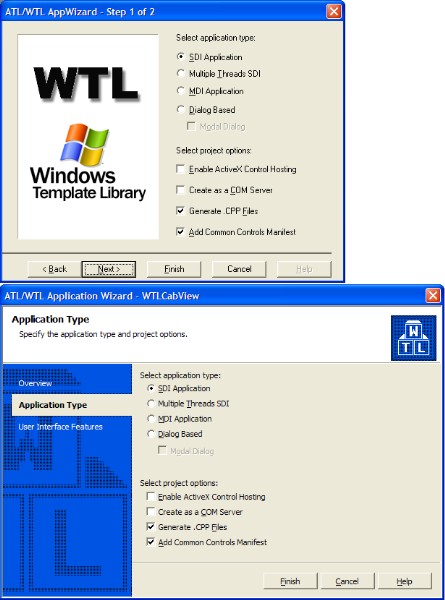


图60 WTL应用程序向导

在下一页里，去掉命令条，并把视图类型改为列表视图。向导会为我们的视图窗口创建一个派生自CListViewCtrl的C++类。

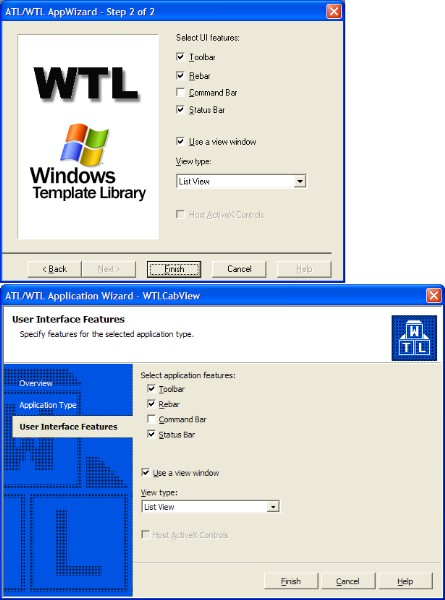


图61 选择列表视图

视图窗口类看起来就是这样：

|  |
| --- |
| class CWTLCabViewView : public CWindowImpl<CWTLCabViewView, CListViewCtrl>  {  public:  DECLARE\_WND\_SUPERCLASS(NULL, CListViewCtrl::GetWndClassName())    // 构造函数  CWTLCabViewView();    // 消息映射  BEGIN\_MSG\_MAP(CWTLCabViewView)  END\_MSG\_MAP()    // ...  }; |

就像我们在第二章里使用视图类一样，我们可以使用CWindowImpl的第三个模板参数设置缺省的窗口风格：

|  |
| --- |
| **#define VIEW\_STYLES \**  **(LVS\_REPORT | LVS\_SHOWSELALWAYS | \**  **LVS\_SHAREIMAGELISTS | LVS\_AUTOARRANGE )**  **#define VIEW\_EX\_STYLES (WS\_EX\_CLIENTEDGE)**    class CWTLCabViewView : public CWindowImpl<CWTLCabViewView, CListViewCtrl,  **CWinTraitsOR<VIEW\_STYLES,VIEW\_EX\_STYLES>** >  {  //...  }; |

由于在WTL中没有文档/视图框架，视图类需要做双份的工作，既是UI，也是存放有关CAB信息的地方。在拖放操作中传递的数据结构为CDraggedFileInfo：

|  |
| --- |
| struct CDraggedFileInfo  {  // 在拖放开始时设置的数据  CString sFilename; // 存放CAB的文件名  CString sTempFilePath; // 解析CAB文件的目录  int nListIdx; // 列表控件中该项的索引    // 当解析文件时设置数据  bool bPartialFile; // 如果文件在另一个CAB文件中继续为true  CString sCabName; // CAB文件名  bool bCabMissing; // 如果文件只部分存放在这个CAB，而继续部分的CAB文件没找到时，为true，  // 这意味着该文件不可被解析    CDraggedFileInfo ( const CString& s, int n ) : sFilename(s), nListIdx(n), bPartialFile(false), bCabMissing(false)  { }  }; |

视图类中还有如下方法：初始化、管理文件列表，以及在拖放操作开始的时候准备一个 CDraggedFile Info列表。由于本文是讲关于拖放的，所以我不会深入到UI工作的内部去，需要了解所有细节的话可以检视示例工程中的 *WTLCabViewView.h*。

**10.1.2、文件打开处理**

要查看一个CAB文件，用户可以使用*File-Open*命令并选择一个CAB文件。向导为CMainFrame生成的代码包含了*File-Open*菜单项的一个处理器：

|  |
| --- |
| BEGIN\_MSG\_MAP(CMainFrame)  COMMAND\_ID\_HANDLER\_EX(ID\_FILE\_OPEN, OnFileOpen)  END\_MSG\_MAP() |

OnFileOpen()使用了CMyFileDialog类，该类是在第九章里介绍到的WTL的CFileDialog的增强版本，用以显示一个标准的文件打开对话框。

|  |
| --- |
| void CMainFrame::OnFileOpen ( UINT uCode, int nID, HWND hwndCtrl )  {  CMyFileDialog dlg ( true, \_T("cab"), 0U, OFN\_HIDEREADONLY|OFN\_FILEMUSTEXIST,  IDS\_OPENFILE\_FILTER, \*this );    if ( IDOK == dlg.DoModal(\*this) )  ViewCab ( dlg.m\_szFileName );  } |

OnFileOpen()调用了辅助函数ViewCab()：

|  |
| --- |
| void CMainFrame::ViewCab ( LPCTSTR szCabFilename )  {  if ( EnumCabContents ( szCabFilename ) )  m\_sCurrentCabFilePath = szCabFilename;  } |

EnumCabContents()相当的复杂，它使用CAB SDK调用来枚举在OnFileOpen()中选中的文件的内容，并填充视图窗口。不过ViewCab()现在还不完善，我们后面会给它加入支持MRU列表的代码。下面是查看器的样子，其中显示着 Windows 98 的某个CAB文件的内容：

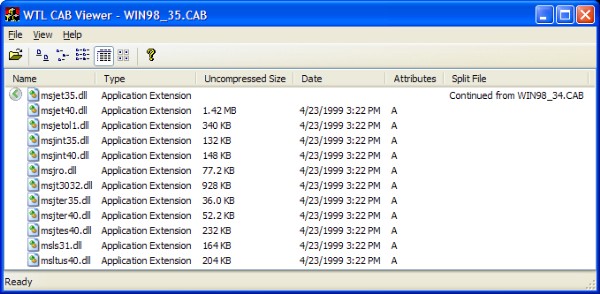


图62 查看的结果

EnumCabContents()使用了视图类中的两个方法来填充UI：AddFile()和AddPartial File()。Add PartialFile()在当一个文件被部分存储在CAB中时被调用，因为其头部是在前面的CAB里。在上面的截图里，列表中的第一个文件就是一个部分文件。其余的文件是通过AddFile()添加的。这两个方法都会为要添加的文件分配一个数据结构，从而视图可以知道其显示的每个文件的所有相关细节。

如果EnumCabContents()返回真，则代表所有的枚举以及UI设置工作成功完成。如果我们只是写一个简单的CAB查看器，我们就可以收手了，尽管此应用不那么有趣。为了使它真正地有用，我们将对它添加拖放支持，以使用户可以从CAB中提取文件。

**10.1.3、拖动源**

拖放源是一个COM对象，它实现了两个接口：IDataObject和IDropSource。IDataObject用来存放在拖放操作中客户端需要传递的任意数据，在我们这种情况下，此数据应该是一个HDROP结构，其中列出了要从CAB中提取的文件。IDropSource的方法会由OLE调用，用以在拖放操作中向源通知事件。

**10.1.4、拖动源接口**

实现了我们的拖放源的C++类为CDragDropSource。该类以我在简介中提到过的MSDN文章中的IDataObject实现为起始。在该文中你可以找到所有代码相关的细节，因此我在这儿就不重复了。然后我们再向类中加入IDropSource及其两个方法：

|  |
| --- |
| class CDragDropSource : public CComObjectRootEx<CComSingleThreadModel>,  public CComCoClass<CDragDropSource>,  public IDataObject,  **public IDropSource**  {  public:  // 构造函数  CDragDropSource();    // 消息映射  BEGIN\_COM\_MAP(CDragDropSource)  COM\_INTERFACE\_ENTRY(IDataObject)  **COM\_INTERFACE\_ENTRY(IDropSource)**  END\_COM\_MAP()    // IDataObject的方法没有显示...    **// IDropSource**  **STDMETHODIMP QueryContinueDrag ( BOOL fEscapePressed, DWORD grfKeyState );**  **STDMETHODIMP GiveFeedback ( DWORD dwEffect );**  }; |

**10.1.5、用于调用者的辅助方法**

CDragDropSource使用几个辅助方法封装了IDataObject的管理以及拖放的通信。一个拖放操作遵循以下模式：

1. 主框架得到用户开始拖放操作的通知。
2. 主框架调用视图窗口来创建一个被拖动的文件的列表。视图在一个vector<CDraggedFileInfo>向量中返回此信息。
3. 主框架创建一个CDragDropSource对象并将上述向量传递给它，以使它得知要从CAB中提取哪些文件。
4. 主框架开始拖放操作。
5. 如果用户在一个适当的拖放目标上放下，则 CDragDropSource对象提取文件。
6. 主框架更新UI以标示不能被提取的文件。

步骤 3 到 6 由辅助方法处理。初始化在Init()方法中完成：

|  |
| --- |
| bool Init(LPCTSTR szCabFilePath, vector<CDraggedFileInfo>& vec); |

Init()将数据复制到保护成员中，填入到一个HDROP结构，并使用IDataObject方法将该结构保存到数据对象中。Init()还作了另一个重要的步骤：它在TEMP目录下为每个拖动的文件创建了一个零字节的文件。例如，如果用户从CAB文件中拖动*buffy.txt*和*willow.txt*，Init()将在TEMP目录下使用这两个名字创建两个文件。这是为了预防，万一拖放目标要验证从HDROP读入的文件名，如果文件不存在，则目标有可能会拒绝放下。

接下来的方法是DoDragDrop()：

|  |
| --- |
| HRESULT DoDragDrop(DWORD dwOKEffects, DWORD\* pdwEffect); |

DoDragDrop()接受dwOKEffects中的一组DROPEFFECT\_\*标志，这些标志表明了源所允许的那些动作。它会查询必要的接口，然后调用DoDragDrop() API。如果拖放成功，\*pdwEffect就被设置为用户希望执行的DROPEFFECT\_\*值。

最后一个方法是GetDragResults()：

|  |
| --- |
| const vector<CDraggedFileInfo>& GetDragResults(); |

CDragDropSource对象维护的vector<CDraggedFileInfo>会在拖放操作过程中被更新。如果某个文件被发现还连着另一个CAB，或者是不能被提取，则CDraggedFileInfo结构会被执行必要的更新。主框架调用GetDragResults()来获取此向量，查找错误并相应更新 UI。

**10.1.6、IDropSource 的方法**

第一个IDropSource方法是GiveFeedback()，它通知源，用户是想采取哪种操作（移动、复制或者链接）。如果愿意的话源可以改变光标。CDragDropSource对操作保持了跟踪，并告诉OLE要使用缺省的拖放光标。

|  |
| --- |
| STDMETHODIMP CDragDropSource::GiveFeedback(DWORD dwEffect)  {  m\_dwLastEffect = dwEffect;  return DRAGDROP\_S\_USEDEFAULTCURSORS;  } |

另一个IDropSource方法是QueryContinueDrag()。OLE在用户把光标移来移去时调用此方法，并告诉源哪个鼠标键，以及键盘键，被按下了。下边是大多数QueryContinueDrag()的实现所采用的样板代码：

|  |
| --- |
| STDMETHODIMP CDragDropSource::QueryContinueDrag ( BOOL fEscapePressed, DWORD grfKeyState )  {  // 如果按下了ESC键，取消拖动操作；如果左键释放，做放下操作  if ( fEscapePressed )  return DRAGDROP\_S\_CANCEL;  else if ( !(grfKeyState & MK\_LBUTTON) )  {  // 如果在GiveFeedback()得到最后的DROPEFFECT是DROPEFFECT\_NONE，  // 我们中断操作，因为源与目标不匹配  if ( DROPEFFECT\_NONE == m\_dwLastEffect )  return DRAGDROP\_S\_CANCEL;    // TO DO：在这里加入从CAB中提取文件...    return DRAGDROP\_S\_DROP;  }  else  return S\_OK;  } |

当我们发现左键被释放了，就到了我们要从CAB中提取选中的文件的地方了。

|  |
| --- |
| STDMETHODIMP CDragDropSource::QueryContinueDrag ( BOOL fEscapePressed, DWORD grfKeyState )  {  // 如果按下了ESC键，取消拖动操作；如果左键释放，做放下操作  if ( fEscapePressed )  return DRAGDROP\_S\_CANCEL;  else if ( !(grfKeyState & MK\_LBUTTON) )  {  // 如果在GiveFeedback()得到最后的DROPEFFECT是DROPEFFECT\_NONE，  // 我们中断操作，因为源与目标不匹配  if ( DROPEFFECT\_NONE == m\_dwLastEffect )  return DRAGDROP\_S\_CANCEL;    **// 如果放下被接受接受，在这里提取文件，所以，当我们返回时，**  **// 文件在temp目录中，并准备用Explorer来复制**  **if ( ExtractFilesFromCab() )**  **return DRAGDROP\_S\_DROP;**  **else**  **return E\_UNEXPECTED;**  }  else  return S\_OK;  } |

CDragDropSource::ExtractFilesFromCab()是另一个复杂点的代码，它使用CAB SDK把文件提取到 TEMP目录下，覆盖掉我们先前创建的零字节的文件。当QueryContinueDrag()返回DRAGDROP\_S\_ DROP时，也即告诉了OLE完成此拖放操作。如果拖放目标是一个资源浏览器窗口，资源浏览器会把文件从TEMP目录复制到发生拖放的目录。

**10.1.7、从查看器中拖放**

我们已经看过了实现拖放操作逻辑的类，现在，我们来看一下我们的查看器应用是怎样使用这个类的。当主框架窗口接收到LVN\_BEGINDRAG通知消息时，它会调用视图以获取选中文件的列表，而后设置CDragDropSource对象：

|  |
| --- |
| LRESULT CMainFrame::OnListBeginDrag(NMHDR\* phdr)  {  vector<CDraggedFileInfo> vec;  CComObjectStack<CDragDropSource> dropsrc;  DWORD dwEffect = 0;  HRESULT hr;    // Get a list of the files being dragged (minus files  // that we can't extract from the current CAB).  if ( !m\_view.GetDraggedFileInfo(vec) )  return 0; // do nothing    // Init the drag/drop data object.  if ( !dropsrc.Init(m\_sCurrentCabFilePath, vec) )  return 0; // do nothing    // Start the drag/drop!  hr = dropsrc.DoDragDrop(DROPEFFECT\_COPY, &dwEffect);    return 0;  } |

第一个调用的是视图的GetDraggedFileInfo()方法，用以得到选中文件的列表。此方法返回一个vector<CDraggedFileInfo>，我们要用它来初始化CDragDropSource对象。GetDraggedFileInfo()在选定的文件都不能被提取的情况下（例如文件被分块存放在不同的CAB文件中）有可能失败。如果发生了这种情况，则OnListBeginDrag()也静静地失败，不做任何事情就返回。最后，我们调用DoDragDrop()来开始操作，并让CDragDropSource处理剩余的事情。

上面列出的步骤 6 提到了拖放结束后对UI的更新。因为有可能在CAB末尾的一个文件仅仅是部分存储于此CAB中，而剩余的则在后续的一个CAB里。（这在 Windows 9x 的安装文件里非常普遍，在那儿CAB需要能符合软盘的大小）当我们试图提取这样的一个文件时，CAB SDK会告诉我们含有该文件剩余部分的CAB的名字。它还会在原始CAB所在的相同目录下寻找那个CAB，如果存在的话则从中提取文件的剩余部分。

因为我们要在视图窗口中标示分块文件，所以OnListBeginDrag()会检查拖放的结果，看是否找到了分块文件：

|  |
| --- |
| LRESULT CMainFrame::OnListBeginDrag(NMHDR\* phdr)  {  //...    // Start the drag/drop!  hr = dropsrc.DoDragDrop(DROPEFFECT\_COPY, &dwEffect);    **if ( FAILED(hr) )**  **ATLTRACE("DoDragDrop() failed, error: 0x%08X\n", hr);**  **else**  **{**  **// If we found any files continued into other CABs, update the UI.**  **const vector<CDraggedFileInfo>& vecResults = dropsrc.GetDragResults();**  **vector<CDraggedFileInfo>::const\_iterator it;**    **for ( it = vecResults.begin(); it != vecResults.end(); it++ )**  **{**  **if ( it->bPartialFile )**  **m\_view.UpdateContinuedFile ( \*it );**  **}**  **}**    return 0;  } |

我们调用GetDragResults()来获取反映了拖放操作结果的更新过的vector<CDraggedFileInfo>。如果结构中的bPartialFile成员为true，则表示该文件仅部分存在于此CAB中。我们再调用视图方法UpdateContinuedFile()，并将信息结构传递给它，因而它可以相应地文件列表视图中的项。下面就是当发现有后续CAB时，应用程序如何标示分块文件：

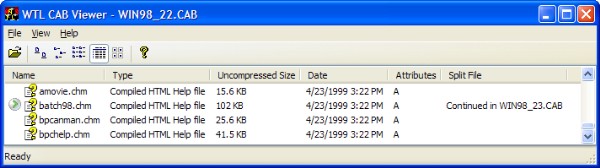


图63 标示分块文件

如果找不到后续的CAB，应用会通过设置LVIS\_CUT风格来标示该文件不可以被提取，所以图标看起来是虚的：

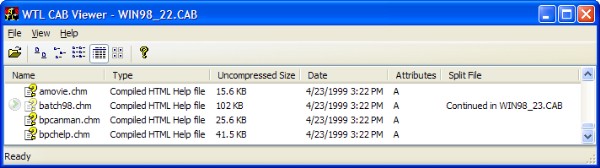


图64 无后续文件显示虚图标

出于安全考虑，应用把提取出的文件留在了TEMP目录里，而不是在拖放操作结束后立刻清除它们。在CDragDropSource::Init()创建零字节的临时文件时，它同时也把文件名添加到了全局向量 g\_ vecsTempFiles中。临时文件在主框架窗口关闭时才删除。

**10.1.8、加入MRU列表**

我们将要看到的另一个文档/视图风格的特性是最近使用文件列表。WTL 的MRU实现为一个模板类 CRecentDocumentListBase。如果你不需要覆盖MRU任何的缺省行为（当然缺省通常已经足够用了），你可以使用派生类 CRecentDocumentList。

|  |
| --- |
| template <class T, int t\_cchItemLen = MAX\_PATH, int t\_nFirstID = ID\_FILE\_MRU\_FIRST,  int t\_nLastID = ID\_FILE\_MRU\_LAST> CRecentDocumentListBase |

CRecentDocumentListBase模板具有以下参数：

|  |  |
| --- | --- |
| 参 数 | 说 明 |
| T | 特化 CRecentDocumentListBase 的派生类的名字。 |
| t\_cchItemLen | 以TCHAR为单位的存储在 MRU 项中的字符串长度。至少必须为 6。 |
| t\_nFirstID | 用于 MRU 项的 ID 范围的最小 ID。 |
| t\_nLastID | 用于 MRU 项的 ID 范围的最大 ID。此值必须大于 t\_nFirstID。 |

要把 MRU 特性添加到我们的应用里，需要做以下几步：

1. 在我们希望 MRU 菜单项出现的地方插入一个ID为ID\_FILE\_MRU\_FIRST的菜单项。此项的文本当列表为空时会显示出来。
2. 添加一个ID为ATL\_IDS\_MRU\_FILE 的字符串表项。此字符串用作MRU项被选中时的动态帮助。如果你使用WTL AppWizard，则此字符串已经帮你创建好了。
3. 添加一个CRecentDocumentList对象到CMainFrame中。
4. 在CMainFrame::Create()里初始化该对象。
5. 处理命令ID介于ID\_FILE\_MRU\_FIRST和ID\_FILE\_MRU\_LAST之间（含）的WM\_COMMAND消息。
6. 当打开一个CAB文件时更新MRU列表。
7. 应用关闭时保存MRU列表。

记住，如果ID\_FILE\_MRU\_FIRST和ID\_FILE\_MRU\_LAST不适合你的应用的话，你还是可以更改ID范围的，只要生成CRecentDocumentListBase的一个新的特化版本就可以。

**10.1.9、设置MRU对象**

第一步就是添加一个菜单项，以标明MRU项应该处于什么位置。通常是在*File*菜单下，这也是我们的应用用到的。下面就是占位菜单项：

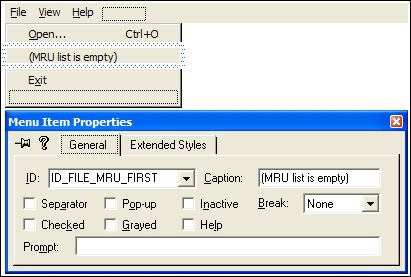


图65 添加菜单资源

AppWizard已经把ATL\_IDS\_MRU\_FILE字符串添加到了字符串表里，我们将之改为读作“打开此CAB文件”。接下来，我们向CMainFrame中加入一个名为m\_mru的CRecentDocumentList类型成员变量并在OnCreate()中初始化它：

|  |
| --- |
| #define APP\_SETTINGS\_KEY \  \_T("software\\Mike's Classy Software\\WTLCabView");    LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )  {  HWND hWndToolBar = CreateSimpleToolBarCtrl(...);    CreateSimpleReBar ( ATL\_SIMPLE\_REBAR\_NOBORDER\_STYLE );  AddSimpleReBarBand ( hWndToolBar );    CreateSimpleStatusBar();    m\_hWndClient = m\_view.Create ( m\_hWnd, rcDefault );  m\_view.Init();    **// Init MRU list**  **CMenuHandle mainMenu = GetMenu();**  **CMenuHandle fileMenu = mainMenu.GetSubMenu(0);**    **m\_mru.SetMaxEntries(9);**  **m\_mru.SetMenuHandle ( fileMenu );**  **m\_mru.ReadFromRegistry ( APP\_SETTINGS\_KEY );**    // ...  } |

前两个方法设置了我们想在维持的项的数目（缺省为16）以及包含占位项的菜单句柄。ReadFrom Registry()从注册表中把MRU列表读出。它接收我们传递给它的键名，并在其下创建一个新键来保存列表。当我们这里，键为HKCU\Software\Mike's Classy Software\WTLCabView\RecentDocument List。

加载文件列表之后，ReadFromRegistry()调用另一个CRecentDocumentList方法，即UpdateMenu()，该方法会找到占位菜单项病将之以实际的MRU项代替。

**10.1.10、处理 MRU 命令并更新列表**

当用户选择了某个MRU项时，主框架会收到一个WM\_COMMAND消息，命令ID等于菜单项的ID。我们可以在消息映射中用一个宏来处理这些命令：

|  |
| --- |
| BEGIN\_MSG\_MAP(CMainFrame)  COMMAND\_RANGE\_HANDLER\_EX( ID\_FILE\_MRU\_FIRST, ID\_FILE\_MRU\_LAST, OnMRUMenuItem)  END\_MSG\_MAP() |

消息处理其从MRU对象处获取该项的全路径，然后调用ViewCab()以使应用显示该文件的内容。

|  |
| --- |
| void CMainFrame::OnMRUMenuItem ( UINT uCode, int nID, HWND hwndCtrl )  {  CString sFile;    if ( m\_mru.GetFromList ( nID, sFile ) )  ViewCab ( sFile, nID );  } |

如上文提到的，我们要扩展ViewCab()以使之感知MRU对象，并在必要时更新文件列表。新的原型为：

|  |
| --- |
| void ViewCab ( LPCTSTR szCabFilename, int nMRUID = 0 ); |

如果nMRUID为 0，则ViewCab()是被OnFileOpen()调用的，否则的话，则是用户选择了某个MRU菜单项，nMRUID就是OnMRUMenuItem()接收到的命令ID。下面是更新过的代码：

|  |
| --- |
| void CMainFrame::ViewCab ( LPCTSTR szCabFilename, int nMRUID )  {  if ( EnumCabContents ( szCabFilename ) )  {  m\_sCurrentCabFilePath = szCabFilename;    **// If this CAB file was already in the MRU list,**  **// move it to the top of the list. Otherwise,**  **// add it to the list.**  **if ( 0 == nMRUID )**  **m\_mru.AddToList ( szCabFilename );**  **else**  **m\_mru.MoveToTop ( nMRUID );**  **}**  **else**  **{**  **// We couldn't read the contents of this CAB file,**  **// so remove it from the MRU list if it was in there.**  **if ( 0 != nMRUID )**  **m\_mru.RemoveFromList ( nMRUID );**  **}**  } |

当EnumCabContents()成功了，我们就根据CAB文件是如何被选中的以不同的方式更新MRU。如果是通过*File-Open*选中的，我们就调用AddToList()来把文件名加入到MRU列表里；如果是通过MRU菜单项选中的，我们就用MoveToTop()把该项移动到列表的顶部。如果EnumCabContents()失败，我们就用RemoveFromList()把文件名从MRU列表中移除。所有这几个方法都会在内部调用UpdateMenu ()，因此*File*菜单会自动被更新。

**10.1.11、保存 MRU 列表**

在应用关闭时，我们把MRU列表保存回注册表中。这事简单，只需要一行：

|  |
| --- |
| m\_mru.WriteToRegistry ( APP\_SETTINGS\_KEY ); |

这行放到了CMainFrame对WM\_DESTROY和WM\_ENDSESSION的消息处理器里。

**10.1.12、其他 UI Goodies**

**10.1.12.1、透明的拖动图像**

Windows 2000及以后有一个内建的COM对象，叫做拖放助手，其目的是在拖放操作中提供良好的透明拖动图像。拖动源通过IDragSourceHelper接口使用此对象。下面是我们加入到OnListBeginDrag ()中的使用此助手对象的额外代码，用粗体标示：

|  |
| --- |
| LRESULT CMainFrame::OnListBeginDrag(NMHDR\* phdr)  {  **NMLISTVIEW\* pnmlv = (NMLISTVIEW\*) phdr;**  **CComPtr<IDragSourceHelper> pdsh;**  vector<CDraggedFileInfo> vec;  CComObjectStack<CDragDropSource> dropsrc;  DWORD dwEffect = 0;  HRESULT hr;    if ( !m\_view.GetDraggedFileInfo(vec) )  return 0; // do nothing    if ( !dropsrc.Init(m\_sCurrentCabFilePath, vec) )  return 0; // do nothing    **// Create and init a drag source helper object**  **// that will do the fancy drag image when the user drags**  **// into Explorer (or another target that supports the**  **// drag/drop helper interface).**  **hr = pdsh.CoCreateInstance ( CLSID\_DragDropHelper );**    **if ( SUCCEEDED(hr) )**  **{**  **CComQIPtr<IDataObject> pdo;**    **if ( pdo = dropsrc.GetUnknown() )**  **pdsh->InitializeFromWindow ( m\_view, &pnmlv->ptAction, pdo );**  **}**    // Start the drag/drop!  hr = dropsrc.DoDragDrop(DROPEFFECT\_COPY, &dwEffect);    // ...  } |

我们从创建此拖放助手COM对象开始。如果成功，我们就调用InitializeFromWindow()并传递三个参数：拖动源的窗口HWND，光标位置，和一个基于我们的CDragDropSource对象的IDataObject接口。拖放助手使用此接口保存其数据，而且，如果拖放目标也使用助手对象的话，此数据用来生成拖动图像。

要使InitializeFromWindow()可以工作，拖放源的窗口需要处理DI\_GETDRAGIMAGE消息，而且在对该消息的效应里，要创建一个用作拖动图像的位图。对我们而言幸运的是，列表视图控件支持这一特性，因此我们仅需很少的工作就可以得到拖动图像。下面是拖动图像的样子：

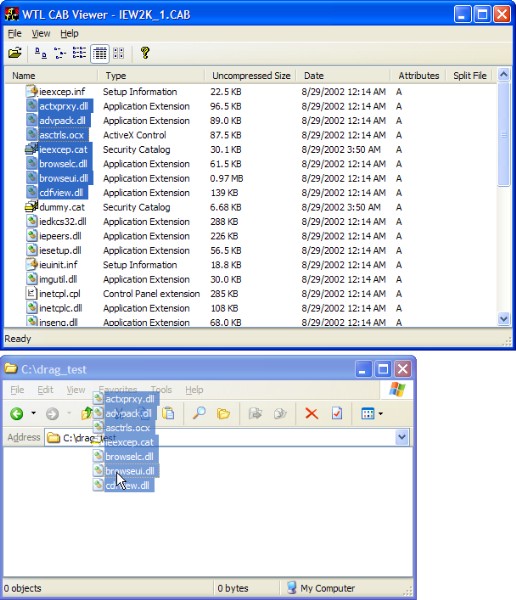


图66 拖放助手效果

如果我们使用一些别的不处理DI\_GETDRAGIMAGE的窗口做我们的视图类，我们就要自己来创建拖动图像并调用InitializeFromBitmap()来把图像保存到拖放助手对象中。

**10.1.12.2、透明的选择矩形**

从Windows XP开始，列表视图控件可以显示一个透明的选择框。此特性缺省是关闭的，但通过给控件设置LVS\_EX\_DOUBLEBUFFER风格即可启用。我们的应用把这件事在CWTLCabViewView:: Init()中作为视图窗口初始化工作的一部分来做。下面是成果：

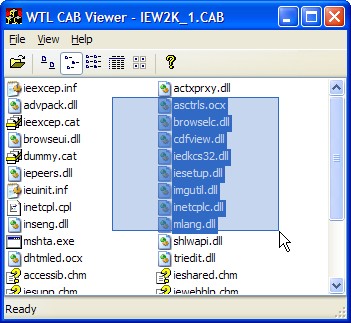


图67 透明选择矩形

如果没有显示出透明选择框，那就需要检查你的系统属性，确保下面的特性是启用了的：

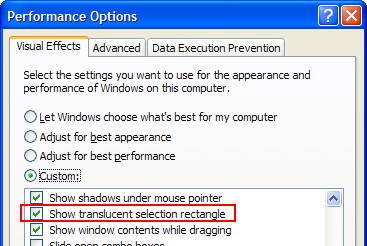


图68 启用特效

**10.1.12.3、标示排序的列**

在Windows XP及之后，详细信息模式的列表视图控件有一个被选中的列，以不同的背景颜色显示。这一特性通常用来标示那一列是被排序了的，而这也正是我们的CAB查看器要做的。标头控件也有了两个新的格式风格，使得在一列中标头可以显示一个向上或者向下的箭头。这通常用来显示排序的方向。

视图类在LVN\_COLUMNCLICK处理器中处理了排序。显示排序列的代码用粗体作了加亮：

|  |
| --- |
| LRESULT CWTLCabViewView::OnColumnClick ( NMHDR\* phdr )  {  int nCol = ((NMLISTVIEW\*) phdr)->iSubItem;    // If the user clicked the column that is already sorted,  // reverse the sort direction. Otherwise, go back to  // ascending order.  if ( nCol == m\_nSortedCol )  m\_bSortAscending = !m\_bSortAscending;  else  m\_bSortAscending = true;    **if ( g\_bXPOrLater )**  **{**  **HDITEM hdi = { HDI\_FORMAT };**  **CHeaderCtrl wndHdr = GetHeader();**    **// Remove the sort arrow indicator from the**  **// previously-sorted column.**  **if ( -1 != m\_nSortedCol )**  **{**  **wndHdr.GetItem ( m\_nSortedCol, &hdi );**  **hdi.fmt &= ~(HDF\_SORTDOWN | HDF\_SORTUP);**  **wndHdr.SetItem ( m\_nSortedCol, &hdi );**  **}**    **// Add the sort arrow to the new sorted column.**  **hdi.mask = HDI\_FORMAT;**  **wndHdr.GetItem ( nCol, &hdi );**  **hdi.fmt |= m\_bSortAscending ? HDF\_SORTUP : HDF\_SORTDOWN;**  **wndHdr.SetItem ( nCol, &hdi );**  **}**    // Store the column being sorted, and do the sort  m\_nSortedCol = nCol;    SortItems ( SortCallback, (LPARAM)(DWORD\_PTR) this );    **// Indicate the sorted column.**  **if ( g\_bXPOrLater )**  **SetSelectedColumn ( nCol );**    return 0;  } |

加亮代码的第一小节去除了先前的排序列的排序箭头。如果没有排序的列的话，就省略这一步。然后，把箭头添加到用户刚刚点击的列上。如果以升序排序则箭头朝上，降序则朝下。排序结束后，我们调用 SetSelectedColumn() ——对LVM\_SETSELECTEDCOLUMN消息的一个封装——来把选中列设置为我们刚刚排序的列。

以下是文件以大小排序后的列表控件的样子：

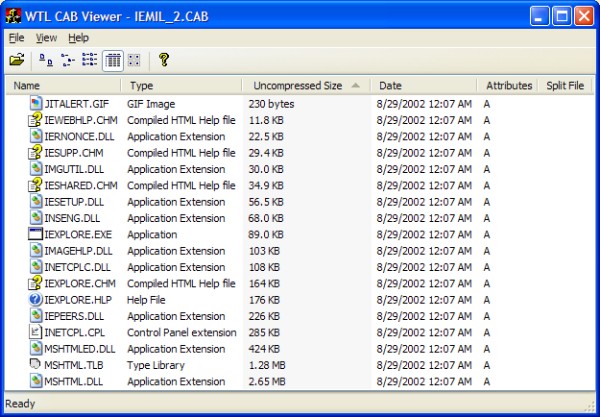


图69 标示排序的列

**10.1.12.4、使用平铺视图模式**

在Windows XP及之后，列表视图控件还有一种新的风格称为*平铺视图模式*。作为视图窗口初始化的一部分，如果应用是运行于XP或之后上，它就会使用SetView()（对LVM\_SETVIEW消息的一个封装）把列表视图的模式设置为平铺模式。然后再填充一个LVTILEVIEWINFO结构来设置一些控制如何平铺绘制的属性。cLines属性被设为了2，表示要在图标旁边显示两行附加文本。dwFlags成员设置为了LVTVIF\_AUTOSIZE，这使得控件自身的大小改变时同时也改变平铺区域的大小。

|  |
| --- |
| void CWTLCabViewView::Init()  {  // ...  // On XP, set some additional properties of the list ctrl.  if ( g\_bXPOrLater )  {  // Turning on LVS\_EX\_DOUBLEBUFFER also enables the  // transparent selection marquee.  SetExtendedListViewStyle ( LVS\_EX\_DOUBLEBUFFER, LVS\_EX\_DOUBLEBUFFER );    **// Default to tile view.**  **SetView ( LV\_VIEW\_TILE );**    **// Each tile will have 2 additional lines (3 lines total).**  **LVTILEVIEWINFO lvtvi = { sizeof(LVTILEVIEWINFO), LVTVIM\_COLUMNS };**    **lvtvi.cLines = 2;**  **lvtvi.dwFlags = LVTVIF\_AUTOSIZE;**  **SetTileViewInfo ( &lvtvi );**  }  } |

**10.1.12.5、设置平铺视图的图像列表**

在平铺视图模式下，我们会使用特大系统图形列表（在缺省的显示设置下图标为48x48大小）。我们使用SHGetImageList() API获取此图像列表。SHGetImageList()不同于SHGetFileInfo()的是它返回一个基于图像列表对象的COM接口。视图窗口有两个成员变量用来管理此图像列表：

|  |
| --- |
| CImageList m\_imlTiles; // the image list handle  CComPtr<IImageList> m\_TileIml; // COM interface on the image list |

视图窗口在InitImageLists()中获取特大图像列表：

|  |
| --- |
| HRESULT (WINAPI\* pfnGetImageList)(int, REFIID, void\*\*);  HMODULE hmod = GetModuleHandle ( \_T("shell32") );    (FARPROC&) pfnGetImageList = GetProcAddress(hmod, "SHGetImageList");    hr = pfnGetImageList ( SHIL\_EXTRALARGE, IID\_IImageList, (void\*\*) &m\_TileIml );    if ( SUCCEEDED(hr) )  {  // HIMAGELIST and IImageList\* are interchangeable,  // so this cast is OK.  m\_imlTiles = (HIMAGELIST)(IImageList\*) m\_TileIml;  } |

如果SHGetImageList()成功，我们可以把IImageList\*接口转型为一个HIMAGELIST，再像使用其它图像列表一样使用它。

**10.1.12.6、使用平铺视图的图像列表**

由于列表控件并不能为平铺视图模式持有单独的图像列表，所以我们需要在运行时，当用户选择大图标或者平铺视图模式时改变图像列表。视图类有一个SetViewMode()方法来处理改变图像列表和视图风格的事宜：

|  |
| --- |
| void CWTLCabViewView::SetViewMode ( int nMode )  {  if ( g\_bXPOrLater )  {  if ( LV\_VIEW\_TILE == nMode )  SetImageList ( m\_imlTiles, LVSIL\_NORMAL );  else  SetImageList ( m\_imlLarge, LVSIL\_NORMAL );    SetView ( nMode );  }  else  {  // omitted - no image list changing necessary on  // pre-XP, just modify window styles  }  } |

如果控件即将进入平铺视图模式，那我们就把控件的图像列表设置为48x48的那个，否则设置为32x32的那个。

**10.1.12.7、设置附加的文本行**

在初始化时，我们将平铺效果设置为要显示附加的两行文本。第一行总是项的文字，就像在大图标和小图标模式里一样。显示在两行附加的文本里的文字取自于子项，类似于详细信息模式下的列。我们可以为每个图标设置单独的子项。下面就是在AddFile()中视图是怎样设置文本的：

|  |
| --- |
| // Add a new list item.  int nIdx;    nIdx = InsertItem ( GetItemCount(), szFilename, info.iIcon );  SetItemText ( nIdx, 1, info.szTypeName );  SetItemText ( nIdx, 2, szSize );  SetItemText ( nIdx, 3, sDateTime );  SetItemText ( nIdx, 4, sAttrs );    **// On XP+, set up the additional tile view text for the item.**  **if ( g\_bXPOrLater )**  **{**  **UINT aCols[] = { 1, 2 };**  **LVTILEINFO lvti = { sizeof(LVTILEINFO), nIdx, countof(aCols), aCols };**    **SetTileInfo ( &lvti );**  **}** |

aCols数组保存有子项，其文本将会被显示出来，在这儿我们显示子项1（文件类型）以及2（文件大小）。下面是在平铺视图模式下查看器的样子：

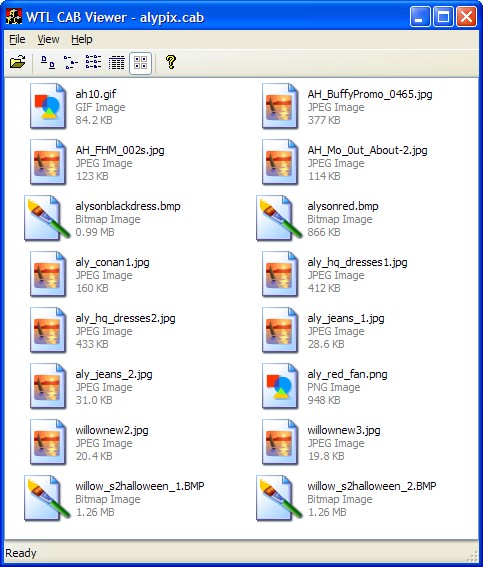


图70 设置附加的文本行

注意，附加行当你在详细信息模式下排序某列之后会改变。当使用LVM\_SETSELECTEDCOLUMN设置了选中列时，该子项的文字总是最先显示，覆盖了我们传递到LVTILEINFO结构中的子项设置。