深入剖析 WTL—WTL 框架窗口分析 （1）

开发者在线 Builder.com.cn 更新时间:2007-11-16 作者：建新 来源:赛迪网
本文关键词： 窗口分析 WTL

WTL 的基础是 ATL。WTL 的框架窗口是 ATL 窗口类的继承。因此，先介绍一下 ATL 对 Windows 窗口的封装。

由第一部分介绍的 Windows 应用程序可以知道创建窗口和窗口工作的逻辑是：

1 注册一个窗口类

2 创建该类窗口

3 显示和激活该窗口

4 窗口的消息处理逻辑在窗口函数中。该函数在注册窗口类时指定。

从上面的逻辑可以看出，要封装窗口主要需解决怎样封装窗口消息处理机制。

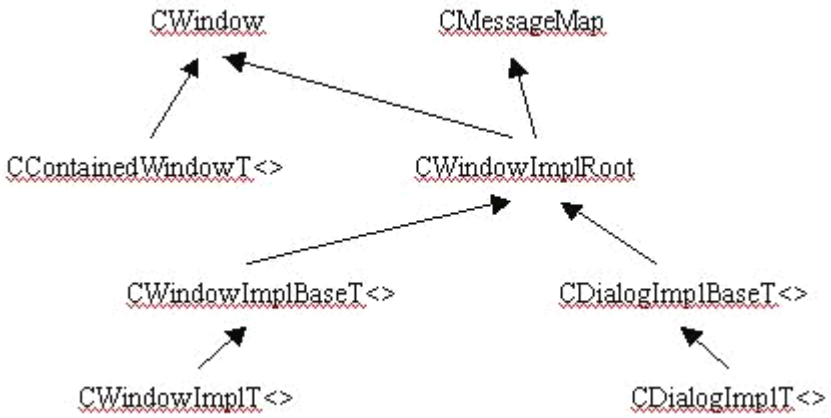
对于窗口消息处理机制的封装存在两个问题。

一是，为了使封装好的类的窗口函数对外是透明的，我们就会想到，要将窗口函数的消息转发到不同的类的实例。那么怎样将窗口函数中的消息转发给封装好的类的实例？因为所有封装好的类窗口的窗口函数只有一个，即一类窗口只有一个窗口函数。而我们希望的是将消息发送给某个类的实例。问题是窗口函数并不知道是哪个实例。它仅仅知道的是 HWND，而不是类的实例的句柄。因此，必须有一种办法，能通过该 HWND，找到与之相对应的类的实例。

二是，假设已经解决了上面的问题。那么怎样将消息传递给相应的类的实例。通常的办法是采用虚函数。将每个消息对应生成一个虚函数。这样，在窗口处理函数中，对于每个消息，都调用其对应的虚函数即可。 但这样，会有很多虚函数，使得类的虚函数表十分巨大。 为此，封装窗口就是要解决上面两个基本问题。对于第二个问题，ATL 是通过只定义一个虚函数。然后，通过使用宏，来生成消息处理函数。对于第一个问题，ATL 通过使用动态改变 HWND 参数方法来实现的。

ATL 对窗口的封装

图示是 ATL 封装的类的继承关系图。从图中可以看到有两个最基本的类。一个是 CWindow，另一个是 CMessageMap。



CWindows 是对 Windows 的窗口 API 的一个封装。它把一个 Windows 句柄封装了起来，并提供了对该句柄所代表的窗口的操作的 API 的封装。 CWindow 的实例是 C++语言中的一个对象。它与实际的 Windows 的窗口通过窗口句柄联系。创建一个 CWindow 的实例时并没有创建相应的 Windows 的窗口，必须调用 CWindow.Create()来创建 Windows 窗口。也可以创建一个 CWindow 的实例，然后将它与已经存在的 Windows 窗口挂接起来。

CMessageMap 仅仅定义了一个抽象虚函数——ProcessWindowMessage()。所有的包含消息处理机制的窗口都必须实现该函数。 通常使用 ATL 开发程序，都是从 CWindowImplT 类派生出来的。从类的继承图可以看出，该类具有一般窗口的操作功能和消息处理机制。

在开发应用程序的时候，你必须在你的类中定义"消息映射"。

```
BEGIN_MSG_MAP(CMainFrame)
        MESSAGE_HANDLER(WM_CREATE, OnCreate)
        COMMAND_ID_HANDLER(ID_APP_EXIT, OnFileExit)
        COMMAND_ID_HANDLER(ID_FILE_NEW, OnFileNew)
        COMMAND_ID_HANDLER(ID_VIEW_TOOLBAR, OnViewToolBar)
        COMMAND_ID_HANDLER(ID_VIEW_STATUS_BAR, OnViewStatusBar)
        COMMAND_ID_HANDLER(ID_APP_ABOUT, OnAppAbout)
      CHAIN_MSG_MAP(CUpdateUI<CMainFrame>)
      CHAIN_MSG_MAP(CFrameWindowImpl<CMainFrame>)
END_MSG_MAP()
```

我们知道一个窗口函数的通常结构就是许多的 case 语句。ATL 通过使用宏，直接形成窗口函数的代码。 消息映射是用宏来实现的。通过定义消息映射和实现消息映射中的消息处理句柄，编译器在编译时，会为你生成你的窗口类的 ProcessWindowMessage()。 消息映射宏包含消息处理宏和消息映射控制宏。

BEGIN_MSG_MAP()和 END_MSG_MAP()

每个消息映射都由 BEGIN_MSG_MAP()宏开始。我们看一下这个宏的定义：

```
#define BEGIN_MSG_MAP(theClass)
public:
        BOOL ProcessWindowMessage(HWND hWnd, UINT uMsg, WPARAM wParam,
        LPARAM lParam, LRESULT& lResult, DWORD dwMsgMapID = 0)
        {
                BOOL bHandled = TRUE;
                hWnd;
                uMsg;
                wParam;
                lParam;
                lResult;
                bHandled;
                switch(dwMsgMapID)
                {
                case 0:
```

一目了然，这是函数 ProcessWindowMessage()的实现。与 MFC 的消息映射相比，ATL 的是多么的简单。记住越是简单越吸引人。 需要注意的是 dwMsgMapID。每个消息映射都有一个 ID。后面会介绍为什么要用这个。 于是可以推断，消息处理宏应该是该函数的函数体中的某一部分。也可以断定 END_MSG_MAP()应该定义该函数的函数结尾。

我们来验证一下：

```
#define END_MSG_MAP()
                break;
        default:
                ATLTRACE2(atlTraceWindowing, 0, _T("Invalid message
map ID (%i)n"),
dwMsgMapID);
                ATLASSERT(FALSE);
                break;
                }
                return FALSE;
        }
```

下面看一下消息映射中的消息处理宏。

深入剖析 WTL—WTL 框架窗口分析 （2）

开发者在线 Builder.com.cn 更新时间:2007-11-16 作者：建新 来源:赛迪网

本文关键词： 窗口分析 WTL

ATL 的消息处理宏

消息映射的目的是实现 ProcessWindowMessage()。ProcessWindowMessage()函数是窗口函数的关键逻辑。 一共有三种消息处理宏，分别对应三类窗口消息——普通窗口消息（如 WM_CREATE），命令消息（WM_COMMANS）和通知消息（WM_NOTIFY）。 消息处理宏的目的是将消息和相应的处理函数（该窗口的成员函数）联系起来。

· 普通消息处理宏

有两个这样的宏：MESSAGE_HANDLER 和 MESSAGE_RANGE_HANDLER。

第一个宏将一个消息和一个消息处理函数连在一起。第二个宏将一定范围内的消息和一个消息处理函数连在一起。

消息处理函数通常是下面这样的：

```
LRESULT MessageHandler(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled);
```

注意最后一个参数 bHandled。它的作用是该处理函数是否处理该消息。如果它为 FALSE，消息 MAP 的其它处理函数会来处理这个消息。

我们看一下 MESSAGE_HANDLE 的定义：

```
#define MESSAGE_HANDLER(msg, func)
        if(uMsg == msg)
        {
                bHandled = TRUE;
                lResult = func(uMsg, wParam, lParam, bHandled);
                if(bHandled)
                        return TRUE;
        }
```

在上面的代码中，首先判断是否是想要处理的消息。如果是的，那么调用第二个参数表示的消息处理函数来处理该消息。 注意 bHandled，如果在消息处理函数中设置为 TRUE，那么，在完成该消息处理后，会进入 return TRUE 语句，从 ProcessWindowMessage()函数中返回。 如果 bHandled 在调用消息处理函数时，设置为 FALSE，则不会从 ProcessWindowMessage 中返回，而是继续执行下去。

· 命令消息处理宏和通知消息处理宏

命令消息处理宏有五个——COMMAND_HANDLER，COMMAND_ID_HANDLER，COMMAND_CODE_HANDLER，COMMAND_RANGE_HANDLER 和 COMMAND_RANGE_CODE_HANDLER。

通知消息处理宏有五个--NOTIFY_HANDLER，NOTIFY_ID_HANDLER，NOTIFY_CODE_HANDLER，NOTIFY_RANGE_HANDLER 和 NOTIFY_RANGE_CODE_HANDLER

我们不再详细分析。

通过上面的分析，我们知道了 ATL 是怎样实现窗口函数逻辑的。那么 ATL 是怎样封装窗口函数的呢？为了能理解 ATL 的封装方法，还必须了解 ATL 中的窗口 subclass 等技术。我们将在分析完这些技术之后，再分析 ATL 对窗口消息处理函数的封装。

扩展窗口类的功能

我们知道 Windows 窗口的功能由它的窗口函数指定。通常在创建 Windows 应用程序时，我们要开发一个窗口函数。通过定义对某些消息的相应来实现窗口的功能。

在每个窗口处理函数的最后，我们一般用下面的语句：

```
        default:
          return DefWindowProc(hWnd, message, wParam, lParam);
```

它的意思是，对于没有处理的消息，我们将它传递给 Windows 的确省窗口函数。Windows 除了提供这个缺省的窗口函数，还为某些标准的控制提供了一些预定义的窗口函数。

我们在注册窗口类的时候，指定了该窗口类的窗口处理函数。

扩展窗口类的功能，就是要改变窗口函数中对某些消息的处理逻辑。

下面我们来看几种扩展窗口功能的技术，以及看看 ATL 是怎样实现的。

派生和 CHAIN_MSG_MAP()

很自然，我们会在一个窗口类的基础上派生另一个。然后通过定义不同的消息处理函数。

下面是一个简单的实例（该例子摘自 MSDN）。

```
class CBase: public CWindowImpl< CBase >
// simple base window class: shuts down app when closed
{
   BEGIN_MSG_MAP( CBase )
      MESSAGE_HANDLER( WM_DESTROY, OnDestroy )
   END_MSG_MAP()

   LRESULT OnDestroy( UINT, WPARAM, LPARAM, BOOL& )
   {
      PostQuitMessage( 0 );
      return 0;
   }
```

```
};
class CDerived: public CBase
// derived from CBase; handles mouse button events
{
   BEGIN_MSG_MAP( CDerived )
      MESSAGE_HANDLER( WM_LBUTTONDOWN, OnButtonDown )
      CHAIN_MSG_MAP( CBase ) // chain to base class
   END_MSG_MAP()

   LRESULT OnButtonDown( UINT, WPARAM, LPARAM, BOOL& )
   {
      ATLTRACE( "button downn" );
      return 0;
   }
};
```

深入剖析 WTL—WTL 框架窗口分析 （3）

开发者在线 Builder.com.cn 更新时间:2007-11-16 作者：建新 来源:赛迪网
本文关键词： 窗口分析 WTL

在上面的例子中，CDerived 从 CBase 中派生出来。

CDerived 类通过定义一 WM_LBUTTONDOWN 消息处理函数来改变 CBase 类代表的窗口的功能。 这样，CBase 类的消息映射定义了一个 ProcessWindowMessage()函数，而 CDerived 类的消息映射也定义了一个 ProcessWindowMessage()函数。 那么，我们在窗口处理函数逻辑中怎样把这两个类的 ProcessWindowMessage()连起来呢？（想想为什么要连起来？） 在 CDerived 的消息映射中，有一个宏 CHAIN_MSG_MAP()。它的作用就是把两个类对消息的处理连起来。

看一下这个宏的定义：

```
#define CHAIN_MSG_MAP(theChainClass)
{
    if(theChainClass::ProcessWindowMessage(hWnd, uMsg, wParam,
lParam, lResult))
     return TRUE;
}
```

很简单，它仅仅调用了基类的 ProcessWindowMessage()函数。

也就是说，CDerived 类的 ProcessWindowMessage()包含两部分，一部分是调用处理 WM_LBUTTONDOWN 的消息处理函数，该函数是该类的成员函数。第二部分是调用 CBase 类的 ProcessWindowMessage()函数，该函数用于处理 WM_DESTROY 消息。

在后面对窗口函数的封装中，我们会知道，对于其他消息处理，CDerived 会传递给缺省窗口函数。

派生和 ALT_MSG_MAP()

如果我们希望在 CBase 类上再派生一个新的窗口类。该类除了要对 WM_RBUTTONDOWN 做不同的处理外，还希望 CBase 对 WM_DESTROY 消息的响应与前一个例子不同。比如希望能提示

关闭窗口信息。 那怎么处理呢？ATL 提供了一种机制，它由 ALT_MSG_MAP()实现。它使得一个类的消息映射能处理多个 Windows 窗口类。

下面是具体的示例：

```
// in class CBase:
   BEGIN_MSG_MAP( CBase )
      MESSAGE_HANDLER( WM_DESTROY, OnDestroy1 )
      ALT_MSG_MAP( 100 )
      MESSAGE_HANDLER( WM_DESTROY, OnDestroy2 )
   END_MSG_MAP()
```

ALT_MSG_MAP()将消息映射分成两个部分。每个部分的消息映射都有一个 ID。上面的消息映射的 ID 分别为 0 和 100。

分析一下 ALT_MSG_MAP()：

```
#define ALT_MSG_MAP(msgMapID)
               break;
               case msgMapID:
```

很简单，它结束了前面的 msgMapID 的处理，开始进入另一个 msgMapID 的处理。

那么，在 CDerived 类的消息映射中，是怎样将两个类的 ProcessWindowMessage()函数的逻辑连在一起的呢？

```
// in class CDerived:
   BEGIN_MSG_MAP( CDerived )
      CHAIN_MSG_MAP_ALT( CBase, 100 )
   END_MSG_MAP()
```

这里使用 CHAIN_MSG_MAP_ALT()宏。它的具体定义如下：

```
#define CHAIN_MSG_MAP_ALT(theChainClass, msgMapID)
{
if(theChainClass::ProcessWindowMessage(hWnd, uMsg, wParam, lParam,
lResult,
msgMapID))
        return TRUE;
        }
```

不再分析其原理。请参考前面对 CHAIN_MSG_MAP（）宏的分析。

深入剖析 WTL—WTL 框架窗口分析 （4）

开发者在线 Builder.com.cn 更新时间:2007-11-13 作者：建新 来源:赛迪网

本文关键词： 分析 窗口 WTL

superclass

superclass 是一种生成新的窗口类的方法。它的中心思想是依靠现有的窗口类，克隆出另一个窗口类。被克隆的类可以是 Windows 预定义的窗口类，这些预定义的窗口类有按钮或下拉框控制等等。也可以是一般的类。克隆的窗口类使用被克隆的类（基类）的窗口消息处理函数。

克隆类可以有自己的窗口消息处理函数，也可以使用基类的窗口处理函数。

需要注意的是，superclass 是在注册窗口类时就改变了窗口的行为。即通过指定基类的窗口函数或是自己定义的窗口函数。这与后面讲到的 subclass 是不同的。后者是在窗口创建完毕后，通过修改窗口函数的地址等改变一个窗口的行为的。

请看示例（摘自 MSDN）：

```
class CBeepButton: public CWindowImpl< CBeepButton >
{
public:
   DECLARE_WND_SUPERCLASS( _T("BeepButton"), _T("Button") )
   BEGIN_MSG_MAP( CBeepButton )
      MESSAGE_HANDLER( WM_LBUTTONDOWN, OnLButtonDown )
   END_MSG_MAP()
   LRESULT OnLButtonDown( UINT, WPARAM, LPARAM, BOOL& bHandled )
   {
      MessageBeep( MB_ICONASTERISK );
      bHandled = FALSE; // alternatively: DefWindowProc()
      return 0;
   }
}; // CBeepButton
```

该类实现一个按钮，在点击它时，会有响声。

该类的消息映射处理 WM_LBUTTONDOWN 消息。其它的消息由 Windows 缺省窗口函数处理。在消息映射前面，有一个宏--DECLARE_WND_SUPERCLASS()。它的作用就是申明 BeepButton 是 Button 的一个 superclass。

分析一下这个宏：

```
#define DECLARE_WND_SUPERCLASS(WndClassName, OrigWndClassName)
static CWndClassInfo& GetWndClassInfo()
{
        static CWndClassInfo wc =
        {
                { sizeof(WNDCLASSEX), 0, StartWindowProc,
                  0, 0, NULL, NULL, NULL, NULL, NULL, WndClassName,
NULL },
                OrigWndClassName, NULL, NULL, TRUE, 0, _T("")
        };
        return wc;
}
```

这个宏定义了一个静态函数 GetWndClassInfo()。这个函数返回了一个窗口类注册时用到的数据结构 CWndClassInfo。该结构的详细定义如下：

```
struct _ATL_WNDCLASSINFOA
{
        WNDCLASSEXA m_wc;
        LPCSTR m_lpszOrigName;
        WNDPROC pWndProc;
        LPCSTR m_lpszCursorID;
```

```
        BOOL m_bSystemCursor;
        ATOM m_atom;
        CHAR m_szAutoName[13];
        ATOM Register(WNDPROC* p)
        {
                return AtlModuleRegisterWndClassInfoA(&_Module, this,
p);
        }
};
struct _ATL_WNDCLASSINFOW
{
 … …
        {
                return AtlModuleRegisterWndClassInfoW(&_Module, this,
p);
        }
};
typedef _ATL_WNDCLASSINFOA CWndClassInfoA;
typedef _ATL_WNDCLASSINFOW CWndClassInfoW;
#ifdef UNICODE
#define CWndClassInfo CWndClassInfoW
#else
#define CWndClassInfo CWndClassInfoA
#endif
```

这个结构调用了一个静态函数 AtlModuleRegisterWndClassInfoA(&_Module, this, p);。这个函数的用处就是注册窗口类。 它指定了 WndClassName 是 OrigWdClassName 的 superclass。

subclass

subclass 是普遍采用的一种扩展窗口功能的方法。它的大致原理如下。

在一个窗口创建完了之后，将该窗口的窗口函数替换成新的窗口消息处理函数。这个新的窗口函数可以对某些需要处理的特定的消息进行处理，然后再将处理传给原来的窗口函数。 注意它与 superclass 的区别。 Superclass 是以一个类为原版，进行克隆。既在注册新的窗口类时，使用的是基类窗口的窗口函数。 而 subclass 是在某一个窗口注册并创建后，通过修改该窗口的窗口消息函数的地址而实现的。它是针对窗口实例。

看一个从 MSDN 来的例子：

```
class CNoNumEdit: public CWindowImpl< CNoNumEdit >
{
   BEGIN_MSG_MAP( CNoNumEdit )
      MESSAGE_HANDLER( WM_CHAR, OnChar )
   END_MSG_MAP()
   LRESULT OnChar( UINT, WPARAM wParam, LPARAM, BOOL& bHandled )
   {
      TCHAR ch = wParam;
```

```
      if( _T('0') <= ch && ch <= _T('9') )
         MessageBeep( 0 );
      else
         bHandled = FALSE;
      return 0;
   }
};
```

这里定义了一个只接收数字的编辑控件。即通过消息映射，定义了一个特殊的消息处理逻辑。

然后，我们使用 CWindowImplT. SubclassWindow()来 subclass 一个编辑控件。

```
class CMyDialog: public CDialogImpl<CMyDialog>
{
public:
   enum { IDD = IDD_DIALOG1 };
   BEGIN_MSG_MAP( CMyDialog )
      MESSAGE_HANDLER( WM_INITDIALOG, OnInitDialog )
   END_MSG_MAP()
   LRESULT OnInitDialog( UINT, WPARAM, LPARAM, BOOL& )
   {
      ed.SubclassWindow( GetDlgItem( IDC_EDIT1 ) );
      return 0;
   }

   CNoNumEdit ed;
};
```

上述代码中，ed.SubclassWindow(GetDlgItem(IDC_EDIT1))语句是对 IDC_EDIT1 这个编辑控件进行 subclass。该语句实际上是替换了编辑控件的窗口函数。 由于 SubClassWindows()实现的机制和 ATL 封装窗口函数的机制一样，我们会在后面介绍 ATL 是怎么实现它的。

深入剖析 WTL—WTL 框架窗口分析 （5）

开发者在线 Builder.com.cn 更新时间:2007-11-13 作者：建新 来源:赛迪网
本文关键词： 分析 窗口 WTL

ATL 对窗口消息处理函数的封装

在本节开始部分谈到的封装窗口的两个难题，其中第一个问题是怎样解决将窗口函数的消息转发到 HWND 相对应的类的实例中的相应函数。 下面我们来看一下，ATL 采用的是什么办法来实现的。我们知道每个 Windows 的窗口类都有一个窗口函数。

LRESULT WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

在类 CWindowImplBaseT 中，定义了两个类的静态成员函数。

```
template <class TBase = CWindow, class TWinTraits = CControlWinTraits>
class ATL_NO_VTABLE CWindowImplBaseT : public CWindowImplRoot< TBase >
{
public:
  … …
static LRESULT CALLBACK StartWindowProc(HWND hWnd, UINT uMsg, WPARAM
wParam,
LPARAM lParam);
static LRESULT CALLBACK WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam,
LPARAM lParam);
 … …
}
```

它们都是窗口函数。之所以定义为静态成员函数，是因为每个类必须只有一个窗口函数，而且，窗口函数的申明必须是这样的。

在前面介绍的消息处理逻辑过程中，我们知道怎样通过宏生成虚函数 ProcessWindowsMessage(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam, LRESULT& lResult, DWORD dwMsgMapID)。

现在的任务是怎样在窗口函数中把消息传递给某个实例（窗口）的 ProcessWindowsMessage()。这是一个难题。窗口函数是类的静态成员函数，因此，它不象类的其它成员函数，参数中没有隐含 this 指针。 注意，之所以存在这个问题是因为 ProcessWindowsMessage()是一个虚函数。而之所以用虚函数是考虑到类的派生及多态性。如果不需要实现窗口类的派生及多态性，是不存在这个问题的。通常想到的解决办法是根据窗口函数的 HWND 参数，寻找与其对应的类的实例的指针。然后，通过该指针，调用该实例的消息逻辑处理函数 ProcessWindowsMessage()。

这样就要求存储一个全局数组，将 HWND 和该类的实例的指针一一对应地存放在该数组中。

ATL 解决这个问题的方法很巧妙。该方法并不存储这些对应关系，而是使窗口函数接收 C++类指针作为参数来替代 HWND 作为参数。

具体步骤如下：

· 在注册窗口类时，指定一个起始窗口函数。

· 创建窗口类时，将 this 指针暂时保存在某处。

· Windows 在创建该类的窗口时会调用起始窗口函数。它的作用是创建一系列二进制代码（thunk）。这些代码用 this 指针的物理地址来取代窗口函数的 HWND 参数，然后跳转到实际的窗口函数中。这是通过改变栈来实现的。

· 然后，用这些代码作为该窗口的窗口函数。这样，每次调用窗口函数时都对参数进行转换。

· 在实际的窗口函数中，只需要将该参数 cast 为窗口类指针类型。

详细看看 ATL 的封装代码。

1. 注册窗口类时，指定一个起始窗口函数。

在 superclass 中，我们分析到窗口注册时，指定的窗口函数是 StartWindowProc()。

2. 创建窗口类时，将 this 指针暂时保存在某处。

```
template <class TBase, class TWinTraits>
HWND CWindowImplBaseT< TBase, TWinTraits >::Create(HWND hWndParent, RECT& rcPos,
LPCTSTR szWindowName,
DWORD dwStyle, DWORD dwExStyle, UINT nID, ATOM atom, LPVOID lpCreateParam)
{
ATLASSERT(m_hWnd == NULL);
```

```
if(atom == 0)
return NULL;
_Module.AddCreateWndData(&m_thunk.cd, this);
if(nID == 0 && (dwStyle & WS_CHILD))
nID = (UINT)this;
HWND hWnd = ::CreateWindowEx(dwExStyle, (LPCTSTR)MAKELONG(atom, 0), szWindowName,
dwStyle, rcPos.left, rcPos.top, rcPos.right - rcPos.left,
rcPos.bottom - rcPos.top, hWndParent, (HMENU)nID,
_Module.GetModuleInstance(), lpCreateParam);
ATLASSERT(m_hWnd == hWnd);
return hWnd;
}
```

该函数用于创建一个窗口。它做了两件事。第一件就是通过 _Module.AddCreateWndData(&m_thunk.cd, this);语句把 this 指针保存在_Module 的某个地方。

第二件事就是创建一个 Windows 窗口。

3. 一段奇妙的二进制代码

下面我们来看一下一段关键的二进制代码。它的作用是将传递给实际窗口函数的 HWND 参数用类的实例指针来代替。

ATL 定义了一个结构来代表这段代码：

```
#pragma pack(push,1)
struct _WndProcThunk
{
        DWORD   m_mov; // mov dword ptr [esp+0x4], pThis (esp+0x4 is
hWnd)
        DWORD   m_this;       //
        BYTE    m_jmp;        // jmp WndProc
        DWORD   m_relproc;    // relative jmp
};
```

#pragma pack(pop)

#pragma pack(push,1)的意思是告诉编译器，该结构在内存中每个字段都紧紧挨着。因为它存放的是机器指令。

这段代码包含两条机器指令：

```
mov dword ptr [esp+4], pThis
jmp WndProc
```

MOV 指令将堆栈中的 HWND 参数（esp+0x4）变成类的实例指针 pThis。JMP 指令完成一个相对跳转到实际的窗口函数 WndProc 的任务。注意，此时堆栈中的 HWND 参数已经变成了 pThis，也就是说，WinProc 得到的 HWND 参数实际上是 pThis。

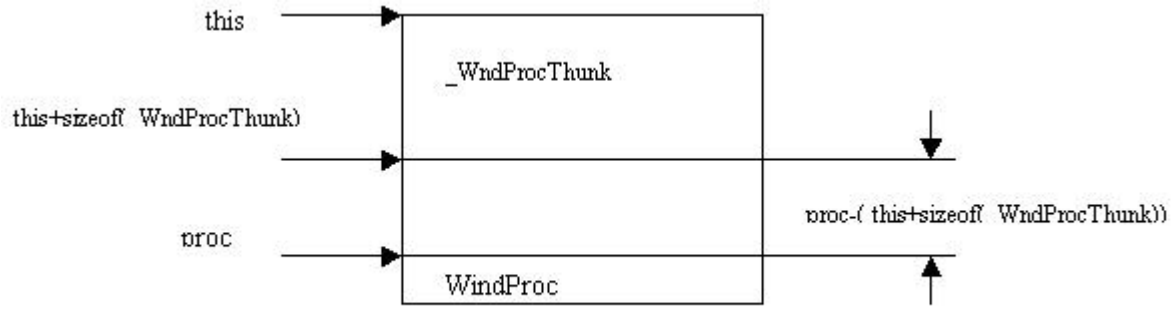
上面最关键的问题是计算出 jmp WndProc 的相对偏移量。

我们看一下 ATL 是怎样初始化这个结构的。

```
class CWndProcThunk
{
public:
```

```
        union
        {
                _AtlCreateWndData cd;
                _WndProcThunk thunk;
        };
        void Init(WNDPROC proc, void* pThis)
        {
                thunk.m_mov = 0x042444C7;  //C7 44 24 0C
                thunk.m_this = (DWORD)pThis;
                thunk.m_jmp = 0xe9;
                thunk.m_relproc = (int)proc -
((int)this+sizeof(_WndProcThunk));
                // write block from data cache and
                //  flush from instruction cache
                FlushInstructionCache(GetCurrentProcess(), &thunk,
sizeof(thunk));
        }
};
```

ATL 包装了一个类并定义了一个 Init()成员函数来设置初始值的。在语句 thunk.m_relproc = (int)proc - ((int)this+sizeof(_WndProcThunk)); 用于把跳转指令的相对地址设置为(int)proc - ((int)this+sizeof(_WndProcThunk))。



上图是该窗口类的实例（对象）内存映象图，图中描述了各个指针及它们的关系。很容易计算出相对地址是(int)proc - ((int)this+sizeof(_WndProcThunk))。

4. StartWindowProc()的作用

```
template <class TBase, class TWinTraits>
LRESULT CALLBACK CWindowImplBaseT< TBase, TWinTraits >::StartWindowProc(HWND hWnd,
UINT uMsg, WPARAM wParam, LPARAM lParam)
{
CWindowImplBaseT< TBase, TWinTraits >* pThis = (CWindowImplBaseT< TBase,
TWinTraits >*)_Module.ExtractCreateWndData();
          ATLASSERT(pThis != NULL);
          pThis->m_hWnd = hWnd;
          pThis->m_thunk.Init(pThis->GetWindowProc(), pThis);
          WNDPROC pProc = (WNDPROC)&(pThis->m_thunk.thunk);
```

```
          WNDPROC pOldProc = (WNDPROC)::SetWindowLong(hWnd, GWL_WNDPROC, (LONG)pProc);
#ifdef _DEBUG
          // check if somebody has subclassed us already since we discard it
          if(pOldProc != StartWindowProc)
                    ATLTRACE2(atlTraceWindowing, 0, _T("Subclassing through a hook
discarded.n"));
#else
          pOldProc; // avoid unused warning
#endif
          return pProc(hWnd, uMsg, wParam, lParam);
}
```

该函数做了四件事：

一是调用_Module.ExtractCreateWndData()语句，从保存 this 指针的地方得到该 this 指针。

二是调用 m_thunk.Init(pThis->GetWindowProc(), pThis)语句初始化 thunk 代码。

三是将 thunk 代码设置为该窗口类的窗口函数。

```
WNDPROC pProc = (WNDPROC)&(pThis->m_thunk.thunk);
        WNDPROC pOldProc = (WNDPROC)::SetWindowLong(hWnd, GWL_WNDPROC,
(LONG)pProc);
```

这样，以后的消息处理首先调用的是 thunk 代码。它将 HWND 参数改为 pThis 指针，然后跳转到实际的窗口函数 WindowProc()。

四是在完成上述工作后，调用上面的窗口函数。

由于 StartWindowProc()在创建窗口时被 Windows 调用。在完成上述任务后它应该继续完成 Windows 要求完成的任务。因此在这里，就简单地调用实际的窗口函数来处理。

5. WindowProc()窗口函数

下面是该函数的定义：

```
template <class TBase, class TWinTraits>
LRESULT CALLBACK CWindowImplBaseT< TBase, TWinTraits >::WindowProc(HWND hWnd,
UINT uMsg, WPARAM wParam, LPARAM lParam)
{
CWindowImplBaseT< TBase, TWinTraits >* pThis = (CWindowImplBaseT< TBase,
TWinTraits >*)hWnd;
// set a ptr to this message and save the old value
MSG msg = { pThis->m_hWnd, uMsg, wParam, lParam, 0, { 0, 0 } };
const MSG* pOldMsg = pThis->m_pCurrentMsg;
pThis->m_pCurrentMsg = &msg;
// pass to the message map to process
LRESULT lRes;
BOOL bRet = pThis->ProcessWindowMessage(pThis->m_hWnd, uMsg, wParam, lParam, lRes,
0);
// restore saved value for the current message
ATLASSERT(pThis->m_pCurrentMsg == &msg);
pThis->m_pCurrentMsg = pOldMsg;
// do the default processing if message was not handled
```

```
if(!bRet)
{
		if(uMsg != WM_NCDESTROY)
			lRes = pThis->DefWindowProc(uMsg, wParam, lParam);
			else
			{
				// unsubclass, if needed
				LONG pfnWndProc = ::GetWindowLong(pThis->m_hWnd,
GWL_WNDPROC);
				lRes = pThis->DefWindowProc(uMsg, wParam, lParam);
				if(pThis->m_pfnSuperWindowProc != ::DefWindowProc &&
::GetWindowLong(pThis->m_hWnd, GWL_WNDPROC) == pfnWndProc)
					::SetWindowLong(pThis->m_hWnd, GWL_WNDPROC,
(LONG)pThis->m_pfnSuperWindowProc);
				// clear out window handle
				HWND hWnd = pThis->m_hWnd;
				pThis->m_hWnd = NULL;
				// clean up after window is destroyed
				pThis->OnFinalMessage(hWnd);
			}
		}
		return lRes;
}
```

首先，该函数把 hWnd 参数 cast 到一个类的实例指针 pThis。

然后调用 pThis->ProcessWindowMessage(pThis->m_hWnd, uMsg, wParam, lParam, lRes, 0);语句，也就是调用消息逻辑处理，将具体的消息处理事务交给 ProcessWindowMessage()。

接下来，如果 ProcessWindowMessage()没有对任何消息进行处理，就调用缺省的消息处理。

注意这里处理 WM_NCDESTROY 的方法。这和 subclass 有关，最后恢复没有 subclass 以前的窗口函数。

WTL 对 subclass 的封装

前面讲到过 subclass 的原理，这里看一下是怎么封装的。

```
template <class TBase, class TWinTraits>
BOOL CWindowImplBaseT< TBase, TWinTraits >::SubclassWindow(HWND hWnd)
{
	ATLASSERT(m_hWnd == NULL);
	ATLASSERT(::IsWindow(hWnd));
	m_thunk.Init(GetWindowProc(), this);
	WNDPROC pProc = (WNDPROC)&(m_thunk.thunk);
	WNDPROC pfnWndProc = (WNDPROC)::SetWindowLong(hWnd,
GWL_WNDPROC,
(LONG)pProc);
	if(pfnWndProc == NULL)
		return FALSE;
	m_pfnSuperWindowProc = pfnWndProc;
```

```
        m_hWnd = hWnd;
        return TRUE;
}
```

没什么好说的，它的工作就是初始化一段 thunk 代码，然后替换原先的窗口函数。

深入剖析 WTL—WTL 框架窗口分析 （6）

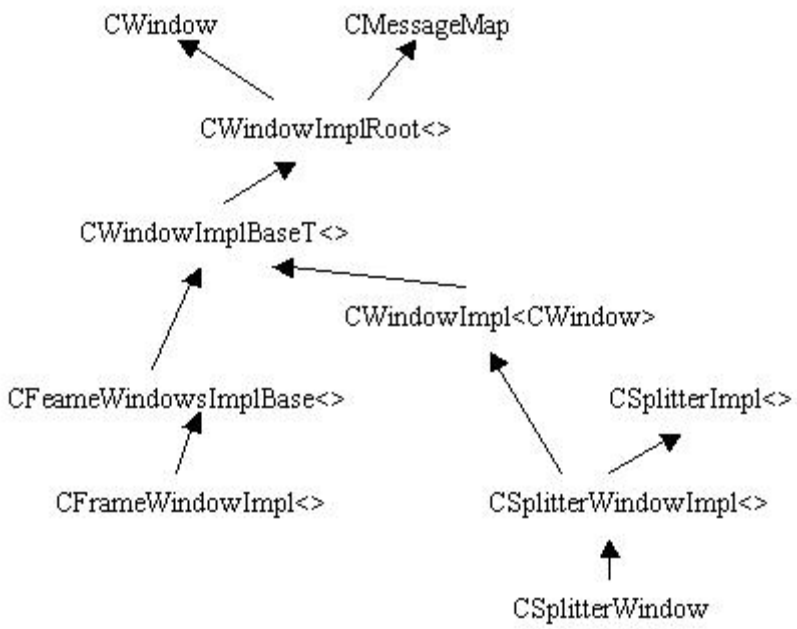
开发者在线 Builder.com.cn 更新时间:2007-11-13 作者：建新 来源:赛迪网
本文关键词： 分析 窗口 WTL

WTL 对框架窗口的封装

ATL 仅仅是封装了窗口函数和提供了消息映射。实际应用中，需要各种种类的窗口，比如，每个界面线程所对应的框架窗口。WTL 正是在 ATL 基础上，为我们提供了框架窗口和其他各种窗口。

所有的应用程序类型中，每个界面线程都有一个框架窗口(Frame)和一个视(View)。它们的概念和 MFC 中的一样。

图示是 WTL 的窗口类的继承图。



WTL 框架窗口为我们提供了：

一个应用程序的标题，窗口框架，菜单，工具栏。

视的管理，包括视的大小的改变，以便与框架窗口同步。

提供对菜单，工具栏等的处理代码。

在状态栏显示帮助信息等等。

WTL 视通常就是应用程序的客户区。它通常用于呈现内容给客户。

WTL 提供的方法是在界面线程的逻辑中创建框架窗口，而视的创建由框架窗口负责。后面会介绍，框架窗口在处理 WM_CREATE 消息时创建视。

如果要创建一个框架窗口，需要：

从 CFrameWindowImpl 类派生你的框架窗口。

加入 DECLARE_FRAME_WND_CLASS，指定菜单和工具栏的资源 ID。

加入消息映射，同时把它与基类的消息映射联系起来。同时，加入消息处理函数。
下面是使用 ATL/WTL App Wizard 创建一个 SDI 应用程序的主框架窗口的申明。

```
class CMainFrame : public CFrameWindowImpl<CMainFrame>,
public CUpdateUI<CMainFrame>,
                              public CMessageFilter, public
CIdleHandler
{
public:
        DECLARE_FRAME_WND_CLASS(NULL, IDR_MAINFRAME)
     // 该框架窗口的视的实例
        CView m_view;
     // 该框架窗口的命令工具行
        CCommandBarCtrl m_CmdBar;
        virtual BOOL PreTranslateMessage(MSG* pMsg);
        virtual BOOL OnIdle();
        BEGIN_UPDATE_UI_MAP(CMainFrame)
                UPDATE_ELEMENT(ID_VIEW_TOOLBAR, UPDUI_MENUPOPUP)
                UPDATE_ELEMENT(ID_VIEW_STATUS_BAR, UPDUI_MENUPOPUP)
        END_UPDATE_UI_MAP()
        BEGIN_MSG_MAP(CMainFrame)
                MESSAGE_HANDLER(WM_CREATE, OnCreate)
                COMMAND_ID_HANDLER(ID_APP_EXIT, OnFileExit)
                COMMAND_ID_HANDLER(ID_FILE_NEW, OnFileNew)
                COMMAND_ID_HANDLER(ID_VIEW_TOOLBAR, OnViewToolBar)
                COMMAND_ID_HANDLER(ID_VIEW_STATUS_BAR,
OnViewStatusBar)
                COMMAND_ID_HANDLER(ID_APP_ABOUT, OnAppAbout)
                CHAIN_MSG_MAP(CUpdateUI<CMainFrame>)
                CHAIN_MSG_MAP(CFrameWindowImpl<CMainFrame>)
        END_MSG_MAP()
// Handler prototypes (uncomment arguments if needed):
//      LRESULT MessageHandler(UINT /*uMsg*/, WPARAM /*wParam*/, LPARAM
/*lParam*/,
BOOL& /*bHandled*/)
//      LRESULT CommandHandler(WORD /*wNotifyCode*/, WORD /*wID*/, HWND
/*hWndCtl*/,
BOOL& /*bHandled*/)
//      LRESULT NotifyHandler(int /*idCtrl*/, LPNMHDR /*pnmh*/, BOOL&
/*bHandled*/)
        LRESULT OnCreate(UINT /*uMsg*/, WPARAM /*wParam*/, LPARAM
/*lParam*/, BOOL&
/*bHandled*/);
        LRESULT OnFileExit(WORD /*wNotifyCode*/, WORD /*wID*/, HWND
/*hWndCtl*/,
```

```
BOOL& /*bHandled*/);
        LRESULT OnFileNew(WORD /*wNotifyCode*/, WORD /*wID*/, HWND
/*hWndCtl*/,
BOOL& /*bHandled*/);
        LRESULT OnViewToolBar(WORD /*wNotifyCode*/, WORD /*wID*/, HWND
/*hWndCtl*/,
BOOL& /*bHandled*/);
        LRESULT OnViewStatusBar(WORD /*wNotifyCode*/, WORD /*wID*/,
HWND
/*hWndCtl*/, BOOL& /*bHandled*/);
        LRESULT OnAppAbout(WORD /*wNotifyCode*/, WORD /*wID*/, HWND
/*hWndCtl*/,
BOOL& /*bHandled*/);
};
```

DECLARE_FRAME_WND_CLASS()宏是为框架窗口指定一个资源 ID，可以通过这个 ID 和应用程序的资源联系起来，比如框架的图标，字符串表，菜单和工具栏等等。

WTL 视

通常应用程序的显示区域分成两个部分。一是包含窗口标题，菜单，工具栏和状态栏的主框架窗口。另一部分就是被称为视的部分。这部分是客户区，用于呈现内容给客户。

视可以是包含 HWND 的任何东西。通过在框架窗口处理 WM_CREATE 时，将该 HWND 句柄赋植给主窗口的 m_hWndClien 成员来设置主窗口的视。

比如，在用 ATL/WTL App Wizard 创建了一个应用程序，会创建一个视，代码如下：

```
class CTestView : public CWindowImpl<CTestView>
{
public:
        DECLARE_WND_CLASS(NULL)

        BOOL PreTranslateMessage(MSG* pMsg);

        BEGIN_MSG_MAP(CTestView)
                MESSAGE_HANDLER(WM_PAINT, OnPaint)
        END_MSG_MAP()

// Handler prototypes (uncomment arguments if needed):
//      LRESULT MessageHandler(UINT /*uMsg*/, WPARAM /*wParam*/, LPARAM
/*lParam*/,
BOOL& /*bHandled*/)
//      LRESULT CommandHandler(WORD /*wNotifyCode*/, WORD /*wID*/, HWND
/*hWndCtl*/,
BOOL& /*bHandled*/)
//      LRESULT NotifyHandler(int /*idCtrl*/, LPNMHDR /*pnmh*/, BOOL&
/*bHandled*/)
LRESULT OnPaint(UINT /*uMsg*/, WPARAM /*wParam*/, LPARAM /*lParam*/,
BOOL&
```

```
/*bHandled*/);
};
```

这个视是一个从 CWindowImpl 派生的窗口。

在主窗口的创建函数中，将该视的 HWND 设置给主窗口的 m_hWndClient 成员。

```
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);
    … …
        return 0;
}
```

上述代码为主窗口创建了视。

到此为止，我们已经从 Win32 模型开始，到了解 windows 界面程序封装以及 WTL 消息循环机制，详细分析了 WTL。通过我们的分析，您是否对 WTL 有一个深入的理解，并能得心应手的开发出高质量的 Windows 应用程序？别急，随后，我们还将一起探讨开发 WTL 应用程序的技巧。

查看本文来源