COM/ActiveX 实用技巧

蔡孟哲◎编著

概念讲解与原理剖析

ATL WINDOW类介绍

ATL WINDOW消息处理机制解说

COM、COM Threading Model剖析

COM组件开发向导

Variant、BSTR、Smart Pointer使用说明

Connection Point（连接点）介绍

实用技巧与精彩实作

ActiveX控件与COM、设计与执行时间授权数字认证、数字签名

用MFC开发ActiveX控件

用ATL开发ActiveX控件

对ActiveX控件给予设计与执行时间授权

对ActiveX控件做数字认证、数字签名

用Jave打开ActiveX控件

内附本书范例光盘

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

COM/ActiveX 实用技巧

蔡孟哲　编著

中国铁道出版社

2001年·北京

（京）新登字 063 号

北京市版权局著作权合同登记号： 01-2001-1432 号

版 权 声 明

图书在版编目（CIP）数据

COM/Active X 实用技巧/蔡孟哲编著. —北京：中国铁道出版社，2001.6

ISBN 7-113-04195-7

Ⅰ.C... Ⅱ.蔡... Ⅲ.软件工程 Ⅳ.TP311.5

中国版本图书馆 CIP 数据核字（2001）第 027457 号

书　　名：COM/Active X 实用技巧
作　　者：蔡孟哲
出版发行：中国铁道出版社（100054，北京市宣武区右安门西街 8 号）
策划编辑：严晓舟
特邀编辑：李晓霞
封面设计：孙天昭
印　　刷：北京兴顺印刷厂
开　　本：787×1092　1/16　印张：12.5　字数：294 千
版　　本：2001 年 6 月第 1 版　2001 年 6 月第 1 次印刷
印　　数：1～5000 册
书　　号：ISBN 7-113-04195-7/TP・555
定　　价：24.00 元

出版说明

本书从ATL定义的Window Class开始，向您详细地介绍了COM的相关概念，其中包括Interface、Threading Model以及它的生命周期等内容。通过具体的实例，讲解如何使用ATL制作COM组件，如何用MFC与ATL开发Active X控件等方法，同时还对Active X控件的特性、Active X与COM的关系、数字签名与数字认证、Design-Time Lincense，以及Run-Time License等作了彻底的分析。为了方便读者学习，本书还附赠一张范例光盘供参考，希望能给您一定的帮助。

本书由台湾松岗电脑图书资料股份有限公司提供版权，经中国铁道出版社计算机图书项目中心审选，杨志刚、何伟、邓雄、王涛、李谨、严尔顺、廖康良、孟丽花、陈贤淑等同志完成整稿及编排工作。

中国铁道出版社

2001 年 6 月

目　录

COM/Active X

第一篇

ATL Overview

实用技巧

原书空白页

第 1 章

ATL 简介

ATL 是 Active Template Library（或 ActiveX Template Library）的缩写，是一组 C++定义好的模板类库(template-based C++ classes)，程序设计师可以用它快速的开发出 COM 组件。在 ATL 里，对于 COM，提供以下一系列支持 Iunknown、IclassFactory、IClassFactory2 与 IDispatch 等的接口，Dual interfaces，Standard COM enumerator interfaces，Connection Points，Tear-off interfaces，以及 ActiveX controls。我们可以使用 ATL 来开发 Single-Threaded objects，Apartment-Model objects，Free-Threaded Model objects，Both Threading objects（本书后续章节将介绍各种 Threading Model）。

1-1 ATL 3.0 Window Classes 入门

虽然 ATL 的主要功能在于以模板（template）支持 COM 应用程序的开发，但用 ATL 开发窗口应用程序具有很高的效率。在 Visual C++ 6.0 以后的版本中，ATL 3.0 修正了许多以前的 bug，并提供几个类支持 Windows Programming，当然还是以 template 为主。介绍 Window 类的原因是日后我们将以这些类以及相关的技术来实作 ActiveX 控件。表 1-1 是 ATL 3.0 提供的 Window Class 以及其功能简介：

表 1-1

CWindow	我们可以将窗口的HANDLE新增至Cwindow，通过它对该窗口做出某些特殊的控制，比方说将窗口移至屏幕正中央。
CWindowImpl	在这个class里，程序员可以对特定的消息做处理，主要以Superclass或Subclass对已有的Window类做延伸处理。
CDialogImpl	通过它，我们可以作 modal 或modeless 的对话框（dialog box）并且可以用消息映射表（message map）处理特定消息。
CcontainedWindow	主要用来支持Contained Window。Contained Window的消息映射并不存在于定义Window的类里，而是被定义在另一个类中。使用CContainedWindow 最大的好处是可以将所有的消息处理集中在同一个类里。
CaxDialogImpl	允许程序设计人员开发包含ActiveX Control的对话框。
CsimpleDialog	允许程序设计人员开发具有最基本功能的对话框。
CAxWindow	通过这个模板类，设计人员可以作出包含ActiveX control的窗口应用程序。

1-2 CWindow Class

CWindow 是 ATL 最基本的 window class ，里面封装了 window handle1e，同时也封装了一个窗口应用程序正常运行所需要的 Windows API。以下是标准的窗口程序码：

```
HWND hWnd = ::CreateWindow( "button", "Click on me",
    WS_CHILD, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT, NULL, NULL, hInstance, NULL );
::ShowWindow( hWnd, nCmdShow );
::UpdateWindow( hWnd );
若以 ATL Cwindow 类来改写则可以改成：
CWindow win;
win.Create( "button", NULL, CWindow::rcDefault, "Click on me",
    WS_CHILD );
win.ShowWindow( nCmdShow );
win.UpdateWindow();
```

窗口（window）， 在本文是泛指一组被 Window OS 用来管理屏幕上某个区域的数据结构。而 ATL CWindow object 只是一个 C++实例,本身并不知道窗口应用程序在屏幕的位置。相对的它使用一个变量 m_Hwnd 作为 CWindow Object 与窗口程序连接，通过 m_Hwnd 使得 CWindow 可以对该窗口程序做控制。一般来说，先创建一个 Window 窗口，然后再把 CWindow Object 的 m_Hwnd 附加到先前产生的窗口，接着再通过 m_Hwnd 来对该 window 控制。程序代码如下：

```
HWND hWnd = CreateWindow( szWndClass, "Main window",
    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT, NULL, NULL, hInstance, NULL );
//      CWindow win( hWnd );        // attach via constructor
// or
//      CWindow win;
//      win = hWnd;                 // attach via operator=
// or
//      CWindow win;
//      win.Attach( hWnd );         // attach via Attach()
win.CenterWindow();         // now can work with win in place of hWnd
hwnd
win.ShowWindow( nCmdShow );
win.UpdateWindow();
```

由以上的程序代码可以发现，我们可以在 CWindow 的构造函数中附加(attach)一个 Window Handle（也就是先前产生的 Window 窗口），或是以 CWindow 提供的 Attach 来附加，之后控制改变先前产生的 Window 的行为。的确，CWindow 使得工程人员改变 Window 的行为更容易。但它却有一个缺点，那就是不提供消息处理机制。虽然 CWindow 可以改变 Window

的外观，甚至传送消息，但是所有的消息处理都只会被先前产生的 Window 处理。因此 ATL 3.0 另外定义了一个模板 CWindowImpl。

1-3 CWindowImpl

CWindowImpl 继承自 CWindow，不同的是它具备了消息处理的能力。在 ATL 里，消息处理与MFC所使用的方法是一样的，都是通过消息映射(message map)。首先，自CwindowImpl派生，定义我们的类如下，并在其中加入消息映射：

```
class CMyWindow : public CWindowImpl<CMyWindow>
{
.....
BEGIN_MSG_MAP(CMyWindow)
    MESSAGE_HANDLER(WM_PAINT,OnPaint)
    MESSAGE_HANDLER(WM_CREATE,OnCreate)
    MESSAGE_HANDLER(WM_DESTROY,OnDestroy)
END_MSG_MAP()
.....
LRESULT OnPaint(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled )
{...
 }
LRESULT OnCreate(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled )
{...
 }
LRESULT OnDestroy(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled )
{...
 } // CMyWindow
}
```

其中 MESSAGE_HANDLER(WM_CREATE,OnCreate)是指当 Window 收到 WM_CREATE 消息时，立刻启动 CMyWindow::OnCreate 方法。而 OnPaint(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)， OnCreate(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)，OnDestroy(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)则是定义处理消息的函数。mMsg 是消息代号，wParam 与 lParam 视函数要处理的消息而定，而 bHandled 用来表示这个消息是否在这个消息函数里已经被处理还是仍然需要再进一步处理。如果我们需要新增一个继承自 CWindowImpl 的对象，只需要下列代码：

```
CMyWindow wnd;          // constructs a CMyWindow object
wnd.Create( NULL, CWindow::rcDefault, _T("Hello"),
    WS_OVERLAPPEDWINDOW|WS_VISIBLE );
```

值得注意的是 CWindowImpl 的 Create 与 CWindow 的 Create 并不一样，在 CWindow 里，

新增类必须要派生自一个已经注册的 Window 类，而 CWindowImpl 并不需要，因此以 CWindowImpl 来新增一个 Window 对象可以新增一个 Window 类，不需要从一个已经注册的 Window 类派生。

1-4 Message Maps

可以将消息分为以下三大类：

1. Message handlers ： WM_CREATE, WM_PAINT 等。
2. Command handlers: WM_COMMAND messages (通常是由 Button 或 menu item 发出)。
3. Notification handlers: WM_NOTIFY messages (通常是由 status bars 或 list view controls 发出)。

1-5 Message Handler Macros

有两类消息处理宏：

1. MESSAGE_HANDLER: 对应特定的消息到特定的消息处理函数。
2. MESSAGE_RANGE_HANDLER: 对应某个特定范围的消息到某个函数。

1-6 Command Handler Macros

1. COMMAND_HANDLER: 对应特定命令至消息处理函数。
2. COMMAND_ID_HANDLER: 对应任何命令消息至消息处理函数。
3. COMMAND_CODE_HANDLER: 对应从任何控件所发出的特定命令至消息处理函数。
4. COMMAND_RANGE_HANDLER: 对应从特定范围的控件所发出的任何命令至消息处理函数。
5. COMMAND_RANGE_CODE_HANDLER: 对应从特定范围的控件所发出的特定命令范围至消息处理函数。

1-7 Notification Handler Macros

用来处理通知类型的消息。

1. NOTIFY_HANDLER
2. NOTIFY_ID_HANDLER
3. NOTIFY_CODE_HANDLER
4. NOTIFY_RANGE_HANDLER
5. NOTIFY_RANGE_CODE_HANDLER

1-8 Adding Functionality to Existing Window Classes（对已有 Window 类的新增功能）

有几种不同方法可以对已有的 window 类新增功能函数。若这个类是 ATL Window 类，可以继承它，以基类链结（Base Class Chaining，稍后做说明）的方式处理消息映射。 如果我们要延伸已有 Window 类（button 或 list box 控件）的适用性或是加强它的延展性，可以对已有的窗口控件类使用超类（superclass）。也就是将控件放入容器对象，可以改变它的外形、颜色数量、位置等。有时候我们想要更细微的在窗口程序执行时改变它所包含的控件（对话框中的按钮或下拉式菜单）的某些行为，可以使用 ATL Window 类对已有的控件派生子类（subclasses）的方式定义一个子类，而控件的消息处理则在新定义的子类里定义。

除此之外，也可以把控件变成 Contained Window，使它包含在一个容器（Container）里。Contained Window 是指一个控件，把它所接收到的消息传送到包含它的容器里，也就是说容器可以定义特殊消息处理函数。

最后，让我们来研究消息反射（Message Reflection），Message Reflection 是指当窗口（Window）程序接收到消息的时并不处理它，而是将它再传递回去给发出消息的控件。但该控件必须要自行处理所发送出去的消息。

1-9 Base Class Chaining

让我们先了解一下 Base Class Chaining。 先定义一个 ATL Windowing 的基类，里面包含一些功能函数。然后再继承它，定义另一个子类。以下是其程序代码：

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

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

class CDerived: public Cbase_mjtsai
// derived from Cbase_mjtsai; handles mouse button events
{
   BEGIN_MSG_MAP( CDerived )
```

```
    MESSAGE_HANDLER( WM_LBUTTONDOWN, OnButtonDown )
  END_MSG_MAP()

  LRESULT OnButtonDown( UINT, WPARAM, LPARAM, BOOL& )
  {
    ATLTRACE( "button down\n" );
    return 0;
  }
};

// in WinMain():
  ...
  CDerived win;
  win.Create( NULL, CWindow::rcDefault, "derived window" );
```

前面曾提到 CWindowImpl 与 CWindow 最大的不同在于 CWindowImpl 可以自行定义新的窗口类，因此本书先定义 Cbase_mjtsai 类。不过当我们以 debug mode 来执行这个程序时，会发现在窗口内用鼠标轻点一下按钮，会出现 button down 小画面，而当我们选择关闭 Cderived 窗口时，即使我们在 Cbase_mjtsai 里定义了 WM_DESTROY 消息的处理函数，程序将不会停止执行。虽然 CDerived 类继承自 Cbase_mjtsai，但是 CDerived 类并未将 WM_DESTROY 消息处理链接（chain）到 Cbase_mjtsai 父类，因此当 CDerived 结束时，没有函数处理 WM_DESTROY 消息。以下是其消息处理函数的修改：

```
BEGIN_MSG_MAP( CDerived )
  MESSAGE_HANDLER( WM_LBUTTONDOWN, OnButtonDown )
  CHAIN_MSG_MAP( Cbase_mjtsai) // chain to base class
END_MSG_MAP()
```

只要加入 CHAIN_MSG_MAP(包含消息处理函数的窗口父类)就可以解决这个的问题。在 CDerived 发出 WM_DESTROY 后，该消息会被自动导向 Cbase_mjtsai 父类。

1-10 Alternate Message Maps

在这里我们讨论一个问题：为什么 Base Class Chaining 机制在子类继承父类时不自动的去帮我们完成呢？因为在多重继承(multiple inheritance)里，一个子类有可能继承多个父类，没有有效的方法可以得知子类中哪一个消息要对应到哪个父类的处理函数。因此这个工作就留给程序员。虽然 base class chaining 可以将消息有效地导向父类处理，但是不同的子类可能有相同的消息要处理，而不同的子类所要求的处理又不尽相同，这该怎么办呢？没关系，在 ATL 里提供了 alternate message maps。我们可以在 message map 里，将相同的消息分割放置在不同的区域，方法如下：

```
// in class Cbase_mjtsai:
    BEGIN_MSG_MAP( Cbase_mjtsai )
        MESSAGE_HANDLER( WM_CREATE, OnCreate1 )
        MESSAGE_HANDLER( WM_PAINT, OnPaint1 )
        ALT_MSG_MAP( 100 )
        MESSAGE_HANDLER( WM_CREATE, OnCreate2 )
        MESSAGE_HANDLER( WM_PAINT, OnPaint2 )
        ALT_MSG_MAP( 101)
        MESSAGE_HANDLER( WM_CREATE, OnCreate3 )
        MESSAGE_HANDLER( WM_PAINT, OnPaint3 )
    END_MSG_MAP()
```

Cbase_mjtsai 的消息映射表里，预设的消息映射区码由 0 开始，在上面的代码中我们看到了另外两个不同消息映射区域是 100 与 101。当我们由子类将消息处理机制链接到父类时，可以指明设定子类中的消息希望由父类中消息映射表里某个区段中的函数来处理。我们可以由下面的代码看到 Cderived 子类希望父类消息映射表里 100 号的区块来处理消息映射，当 CDerived 接收到 WM_PAINT 消息时，Cbase_mjtsai::OnPaint2 就会被自动执行。

```
class CDerived: public Cbase_mjtsai {
    BEGIN_MSG_MAP( CDerived )
        CHAIN_MSG_MAP_ALT( Cbase_mjtsai, 100 )
    END_MSG_MAP()
...
```

1-11 Window Superclassing

Superclassing 可以对先前定义好的窗口类（按钮或是下拉式菜单）延伸，新增函数等。下面的范例 superclass 一个按钮控件，当按钮被按下时就会发出哔声：

```
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
```

DECLARE_WND_SUPERCLASS 宏声明了类名为 BeepButton，以及要被包裹以延伸功能的控件是 Button 类。消息映射表里只对应 WM_LBUTTONDOWN 到 OnLButtonDown 函数。其他消息都将被窗口系统处理。除了发出哔声外，CbeepButton 的表现也应该要像一个正常的按钮控件，因此将 bHandle 设为 FALSE，在函数处理完发出哔声后，预设的系统消息处理函数接着接手处理 WM_LBUTTONDOWN。

截至目前为止，我们只是定义好了一个完整的 CBeepButton 类，现在要做的是新增（create）真实的 CBeepButton。底下的范例类里包含了两个 CbeepButton 类型的成员，当这个程序类被创建（create）出来时，它会有两个 CBeepButton 子窗口。

```
const int ID_BUTTON1 = 101;
const int ID_BUTTON2 = 102;

class CMyWindow: public CWindowImpl< CMyWindow, CWindow,
CWinTraits<WS_OVERLAPPEDWINDOW|WS_VISIBLE> >
{
    CBeepButton b1, b2;

    BEGIN_MSG_MAP( CMyWindow )
        MESSAGE_HANDLER( WM_CREATE, OnCreate )
        COMMAND_CODE_HANDLER( BN_CLICKED, OnClick )
    END_MSG_MAP()
    LRESULT OnClick(WORD wNotifyCode, WORD wID, HWND hWndCtl,
        BOOL& bHandled)
    {
        ATLTRACE( "Control %d clicked\n", wID );
        return 0;
    }
    LRESULT OnCreate( UINT, WPARAM, LPARAM, BOOL& )
    {
        RECT r1 = { 10, 10, 250, 80 };
        b1.Create(*this, r1, "beep1", WS_CHILD|WS_VISIBLE, 0, ID_BUTTON1);
        RECT r2 = { 10, 110, 250, 180 };
        b2.Create(*this, r2, "beep2", WS_CHILD|WS_VISIBLE, 0, ID_BUTTON2);
        return 0;
    }
}; // CMyWindow
```

1-12 Window Subclassing

Subclassing 允许我们改变已有 Window 控件的行为，通常用新增一个消息映射表来改变原有控制对该消息的预设处理。举个例子来说，假如我们现在有一个对话框(Dialog Box)里面