第 2 章　控 制 条

从第1章中可以注意到，通用控件窗口是由 Windows API提供，用于创建按钮、列表框和编辑框等控件的特定子窗口。虽然工具栏可能看起来像这些控件窗口的集合，但实际上它只是一种较长的控件窗口，该控件窗口在本窗口中绘制按钮。状态栏也只是一个较长的窗口，不过它可以在自己的边界以内绘制其窗格。工具栏和状态栏都代表了叫做通用控制条的一类特定通用控件窗口。

本章仅探讨如何使用Windows API创建通用控制条。然后研究利用 MFC类通过API所提供的许多功能来创建它们。最后则看看由这些控制条所绘制的按钮和窗格的可用风格。

2.1　通用控制条

通用控制条有3种：工具栏、状态栏和伸缩条。工具栏和状态栏在 Windows的各个版本中都可以见到。另一方面，伸缩条 (Rebar)只能在Windows 98或者以后版本中以及系统安装了IE4.0后才可以使用，本章最后将讨论伸缩条。工具栏是人们熟悉的控件窗口，它沿着应用主窗口的顶端和四周布局并包含了多个按钮。状态栏通常位于主窗口的底部，它指示键盘状态并显示帮助消息。

初看起来，用户可能会觉得工具栏上的按钮是一排使用通用控件类 BUTTON创建的子窗口，而状态栏则可能被认为是 Static控件的集合。实际上，这两者都是自己绘制并控制这些按钮图像，请参见图2-1。

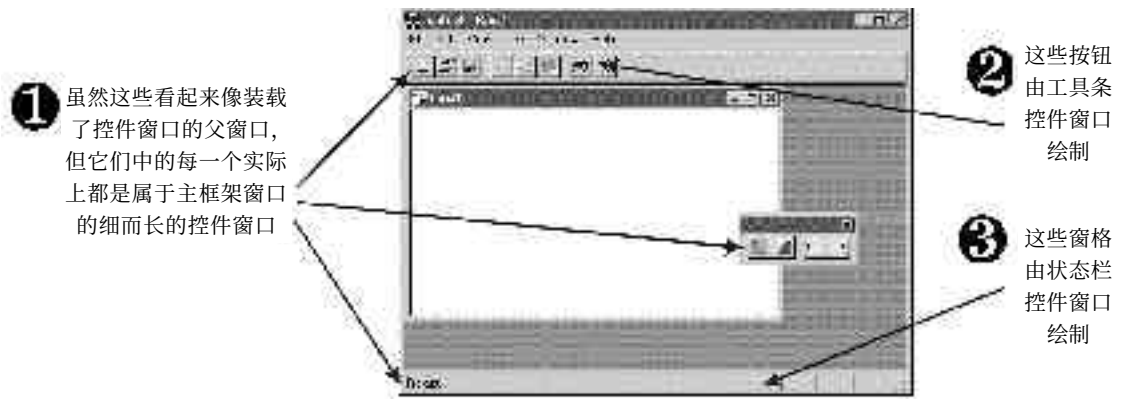


图2-1　工具栏和状态栏控制窗口

那么，为什么控制条要不厌其烦地重复创建这个功能而不使用已有的通用控件呢？答案是控制条自己做这些工作可以节约许多开销。在创建窗口时，　50%以上的时间花费在背景定位和资源初始化上。具有 10个或者10个以上按钮的工具栏采用 1个绘图处理器比在10个处理器中画得快得多。

2.2 用API创建控制条

现在看看只采用API该如何创建并初始化控制条。因为这个工作的大部分可以由 MFC来完成，这里的介绍就不拘泥于细节了。

创建工具栏或者状态栏的Windows API与创建通用控件的API是一样的，只是其窗口类结构不同。可以使用下列语句创建工具栏：

```
HWND hWnd = :: CreateWindowEx(..., "ToolbarWindow32" ,...);
```

工具栏控件可以由一个定义了一套按钮外观的位图对象来初始化。为了定义这个位图对象，可以向这个控件发送一个如下所示的 TB_ADDBITMAP窗口消息：

```
::SendMessage (hWnd, TB_ADDBITMAP, nNumButtons,&tbab);
```

这里使用的 tbab参数是一个 TBADDBITMAP结构，该结构包含了位图对象的句柄。nNumButtons参数则告诉工具栏该位图代表了多少个按钮表面。一旦该位图被定义，就可以发送如下的TB_ADDBUTTONS消息以告诉工具栏绘制什么按钮以及在哪里绘制它们：

```
::SendMessage (hWnd, TB_ADDBUTTONS, nNumButtons, &tbbutton);
```

这里使用的tbbutton参数代表一个定义了每个按钮的 TBBUTTON结构数组。该结构指出针对具体的按钮绘制什么样的按钮表面以及按钮被单击之后产生什么命令消息 ID。

可以使用下列语句创建状态栏：

```
HWND hWnd = :: CreateWindowEx (..., "msctls_statusbar32" ,...);
```

状态栏由通常显示文本的窗格组成。可以使用 SB_SETPARTS窗口消息创建这些窗格：

```
::SendMessage (hWnd, SB_SETPARTS, nParts, Widths);
```

nParts参数告诉控件要创建多少个窗格，而 Widths参数则是一个整型数数组，它定义了以像素为单位计算的每个窗格的大小。可以使用 SB_SETTEXT消息在窗格上添加文本：

```
::SendMessage (hWnd, SB_SETTEXT, nPane,  "pane text");
```

nPane告诉状态栏控件在哪个窗格上添加文本。

控制条风格

对所有通用控件，工具栏和状态栏都可以用标准的 Windows风格(WS_CHILD、WS_VISIBLE等)创建，也可以使用它们自己的风格来创建。工具栏风格有一个 TBSTYLE_前缀，而状态栏风格则具有SBARS_前缀，但现在只用一种状态栏风格。这两种风格的实例可以参见表 2-1。除了这些控件风格以外，这些控制条都还共享了一套名叫 Common Control Styles(通用控件风格)的通用风格。这些风格具有前缀CCS_并且倾向于包括所有控制窗口共有的风格。然而，因为控件窗口的特征不同，这些风格多数仅被控件采用。通用控件风格可以参见表2-2，了解这些风格如何影响控制条请参见图 2-2。

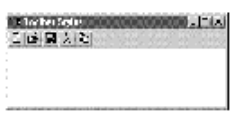
除了这些控件风格外，所有这些控件还有一个公共的风格集合，称为 common Control Styles (通用控件风格)。这些风格的前缀也为 CCS-，其目的是要包含所有控件窗口的所有公共风格。但是，因为控件窗口的不同特性，这些风格大多由控件条使用。表 2-2中列出了这些通用控件风格。要想了解这些风格对控件条的影响，请参见图 2-2。

表2-1 一些工具栏和状态栏风格

风 格	使 用 说 明
TBSTYLE_FLAT	使工具栏上的按钮表面平坦，意味着按钮的边界不会被绘制出来，除非鼠标指针指在上面
TBSTYLE_LIST	按钮文本显示在按钮的左边
TBSTYLE_TOOLTIPS	如果用户将鼠标指针停留在按钮上较长时间就使按钮产生 TTN_NEEDTEXT通知消息。程序员需要负责为这个工具提示提供说明文本
TBSTYLE_WRAPABLE	使控件将其按钮排成多行以适应当前的控件大小
SBARS_SIZEGRIP	唯一的状态栏风格，使状态栏在控件底部显示熟悉的把手条。因为控件正常情况下定位于应用窗口的底部，所以该状态栏看起来影响了整个窗口

表2-2 通用控件风格

风 格	使 用 说 明
CCS_NORESIZE	被创建的控制条的大小由::CreateWindow() API调用来指定。该风格忽略以下4种风格：这些风格则忽略指定的大小
CCS_TOP	使控制条一开始沿其父窗口框架的顶部/底部对齐和延伸。高度则被设置为系统标准。CreateWindos() API调用中指定的大小则被忽略。注意：如果父窗口被重置大小，则必须自己重置控制条大小
CCS_BOTTOM	
CCS_LEFT	同上，但控制条必须沿父框架左/右对齐并且其宽度设置为系统标准
CCS_RIGHT	
CCS_NODIVIDER	通用控制条自动地沿其顶部绘制一条直线：通常用来将工具栏和菜单分隔开
CCS_ADJUSTABLE	允许用户动态地配置其工具栏，这并不是 Developer Studio提供的良好方式，请参看实例9以了解这一Developer Studio方法
CCS_NOMOVEY	这些风格或者是没有明显的效果或者是重复了以上所述的风格
CCS_NOMOVEEX	
CCS_NOPARENTALIGN	



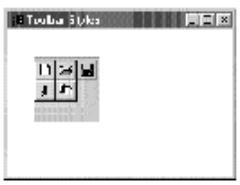
CCS_TOP

CCS_BOTTOM

TBSTYLE_FLAT

CCS_LEFT
即使是左对齐，工具条仍然垂直绘制

CCS_NORESIZE
工具条的尺寸和位置可在创建时指定



工具条中的按钮被设定为特定控件尺寸，但CCS_WRAPABLE不能与CCS_LEFT一起使用

图2-2 工具栏风格实例

如上表和上图所示，控制条可用窗口风格是相当少的，即便加上通用控件风格也是这样。控制条在控件的顶部(而不是在其底部或是左边、右边)自动地绘制一条直线，而且也没有垂直放置通用控制条的风格(可以沿主窗口的左边或者右边对齐控制条，但按钮会消失在控件之外)。此外，创建的控制条不能自动地与其他控制条共享主窗口的客户区。事实上，必须手工重置其大小并移动它们以使视和控制条可见。通用工具栏的另一个难题是用户不能移动它们。现在的多数应用程序中的工具栏可以移动到应用主窗口的任意一边或者浮动在其自己特定的窗口框架上。然而，由 Windows API提供的控制条不支持以上任何功能。为了弥补这个功能上的缺陷，MFC提供了几个类来封装并增强应用程序的控制条功能。

2.3 用MFC创建控制条

MFC提供的两种类封装了创建工具栏和状态栏的 Windows API(与在第1章中所述的CWnd封装Windows API 创建窗口一样)。这些类是CToolBarCtrl和CStatusBarCtrl类。

2.3.1 CToolBarCtrl和CStatusBarCtrl

为了用MFC创建工具栏，使用以下代码：

```
CToolBarCtrl tb;
tb.Create ( DWORD dwStyle, const RECT& rect, CWnd* pParentWnd,
   UINT nID );
```

为了向该工具栏中增加位图，使用如下语句：

```
tb.AddBitmap(int nNumButtons, CBitmap* pBitmap );
```

最后，按照如下语句定义一个实际工具栏按钮：

```
tb.AddButton(int nNumButtons, LPTBBUTTON lpButtons );
```

lpButtons参数与以上用API定义按钮的TBBUTTONS数组是一样的。为了创建状态栏，使用如下代码：

```
CStatusBarCtrl sb;
sb.Create( DWORD dwStyle, const RECT& rect, CWnd* pParentWnd,
   UINT nID );
```

可以使用如下代码增加窗格和文本：

```
sb.SetParts( int nParts, int* pWidths );
sb.SetText( LPCTSTR lpszText, int nPane, int nType );
```

pWidths和nPane参数在以上API版本中已经用过。

正如所看到的，这两个类立即带来的好处就是使得 C++程序中与API对话变得容易多了。然而，由于它们只是简单地封装了API而没有增加附加的功能，所以由它们所创建的工具栏和状态栏不会比API版本增加更多的功能。它们也不能移动或被用户停靠或者垂直对齐。它们仅能用于在控制条内更方便地设置并控制按钮和窗格，所以，为了得到这些所希望的功能就必须创建自己的控制条，这就要用到另外两个 MFC类:CToolBar和CStatusBar。

2.3.2 CToolBar和CStatusBar

为了使用CToolBar创建工具栏，需要再次使用下列熟悉的 Create()成员函数：

```
CToolBar tb;
tb.Create( CWnd* pParentWnd, DWORD dwStyle, UINT nID );
```

如上所示，创建工具栏已经是一件很容易的事，而且不需要考虑 rect参数。但最大的方便还在于装载位图和定义按钮的时候：

```
tb.LoadToolBar(UINT id);
```

此时不用编程定义位图和每个按钮，使用 Toolbar Editor创建一个将要引用的资源即可。LoadToolBar 函数接下来负责将资源翻译为API调用以设置该工具栏。

状态栏类也很容易使用，如下代码所示：

```
CStatusBar sb;
sb.Create( CWnd* pParentWnd, DWORD dwStyle, UINT nID );
sb.SetIndicators( const UINT* lpIDArray, int nIDCount );
```

这时不用为状态栏定义文本和窗格的数目，可以使用 String Editor创建一组在IDArray数组中定义的字符串，该数组则被传送给 SetIndicators()函数。

使用这些类不仅使装载和配置控制条变得更容易，而且还可以得到下列增强的功能：

■ 使用CToolBar类，工具栏中使用的位图的颜色自动地随当前按钮颜色而变化。淡灰色是标准的Windows按钮表面色，但是，如果用户确定为自己的桌面选定一种新的色彩方案 (例如淡紫红色)，那么如果工具栏仍保持淡灰色，按钮看起来就好像偏离了本来的位置。因此，CToolBar将它们转变成当前的按钮颜色，所有这些替代颜色如表 2-3所示

表2-3 CToolBar位图颜色转换

工具栏位图颜色	系 统 颜 色	工具栏位图颜色	系 统 颜 色
黑	COLOR_BTNTEXT	亮灰	COLOR_BTNFACE
暗灰	COLOR_BTNSHADOW	白色	COLOR_BTNHIGHLIGHT

■ 由CToolBar和CStatusBar提供的其他成员函数则建立在 Windows API的功能之上。例如，CToolBar::SetButtonText()函数允许直接设置工具栏按钮的颜色而不用关心所有做出这一改变的API调用。

■ 这些类还自动提供了启用或者禁用工具栏按钮和状态栏窗格或者对其设置复选状态的能力。换句话说，除非工具栏是用 CToolbar创建的，否则OnCmdUI()消息处理器不会启用或者禁用工具栏按钮。

按照C++的继承思想，可以认为 CToolBar和CStausBar类是从 CToolBarCtrl和CStatusBarCtrl类派生的，因此可以得到这两者的功能。但是，微软决定从一种名叫CControlBar的新类来派生CToolBar和CStausBar，这种新类包含两者共同的新功能。那是否意味着微软必须在CToolBar中复制CToolBarCtrl类的功能呢？答案是“不”，这是因为CToolBar和CToolBarCtrl都通过发送消息给一个控件的窗口句柄来同控件对话，这样，事实上可以同时拥有一个CToolBar和CtoolBarCtrl类的实例来控制同样的工具栏并共享一个窗口句柄。实际上，CToolBar类的一个名叫GetToolBarCtrl()的成员函数将自动地创建一个 CToolBarCtrl类的实例并指向同一个工具栏。状态栏也提供了类似的函数 GetStatusBarCtrl()。请参见图2-3了解它们是如何协同工作的。

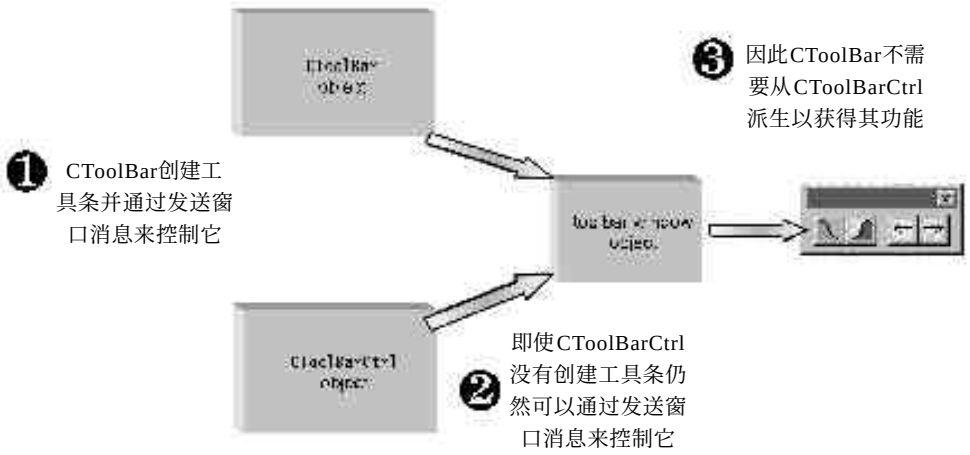


图2-3 控制条的类层次

2.3.3 CControlBar

CToolBar和CStatusBar类都从CControlBar类派生，这就可以允许微软将下列功能加入CControlBar：

■ 如果状态栏或者工具栏不处理下列任何窗口消息， CControlBar就将它们发回给控制条的宿主窗口处理。

WM_NOTIFY	WM_DELETEITEM
WM_COMMAND	WM_COMPAREITEM
WM_DRAWITEM	WM_VKEYTOITEM
WM_MEASUREITEM	WM_CHARTOITEM

通常情况下，窗口消息如果不由所关注的窗口处理则将被丢弃。这个功能允许工具栏将按钮命令传回给框架窗口以便这些消息可以由与处理菜单命令相同的命令处理器来处理。

■ 正如上面提到的， Windows API提供的控制条只能沿它们的顶部绘制边界。CControlBar则可以在工具栏和状态栏的任何一边绘制边界。

■ 同上所述，Windows API提供的控制条不能垂直确定工具栏。状态栏可以沿窗口的左边或者右边对齐，但它们的按钮将会水平地消失在控件之外。即便控制条采用了TBSTYLE_WRAPABLE风格也不会作为单列按钮出现。 CControlBar类则可以创建一个按钮单列。

注意 状态栏不能垂直排列，即使采用了CControlBar也是如此，不过，垂直的状态栏也不怎么符合用户的审美习惯。

■ 同上所述，以前用户必须自己保证Windows API控制条不会互相覆盖或者覆盖窗口视。而使用CControlBar类以及以上所述两种控制类，将会自动地保证控制条不会互相覆盖。对于其他类以及它们是如何表现的，读者可以参考2.7节“视和控制条如何共享客户区”。

■ CControlBar类还为工具栏上的按钮和状态栏上的窗格提供工具提示和在线帮助。工具提示(Tool tip)帮助，又称为气泡(bubble) 帮助，在鼠标指针停留在一个按钮或者窗格之上时将出现一个小小的白色帮助窗口，这就是工具提示帮助。若在鼠标指针停留在菜单选项或者工

具栏按钮之上时则会在状态栏上出现在线帮助。

使用CControlBar类可以具有表2-4所示的几种新的控制条风格，除了 CCS_ADJUSTABLE以外，程序员应该对 CToolBar和CStatusBar控制条使用这些风格而放弃以前所述的 CCS_类别风格。TBSTYLE_和SBARS_风格则可以继续使用。

表2-4 MFC控制条风格

风 格	使 用 说 明
CBRS_ALIGN_TOP	控制条沿父窗口框架的顶部/底部对齐并延伸。如果控制条是可停靠的，则允许它停靠于父窗口的顶部/底部
CBRS_ALIGN_BOTTOM	
CBRS_ALIGN_LEFT	同上，但在父窗口的左边/右边
CBRS_ALIGN_RIGHT	
CBRS_ORIENT_HORZ	使用停靠控制条并允许它们停靠于父窗口的顶部或者底部
CBRS_ORIENT_VERT	同上，但在父窗口的左边或者右边
CBRS_ALIGN_ANY	仅用于停靠控制条的情况，允许该条停靠于父窗口框架的任何一边
CBRS_ORIENT_ANY	
CBRS_TOOLTIPS	在用户将鼠标指针停留在工具栏按钮或者状态栏窗格之上时使工具提示出现
CBRS_FLYBY	在用户将鼠标指针停留在工具栏按钮或者状态栏窗格之上时使在线帮助出现
CBRS_BORDER_TOP	MFC控制条类内部使用
CBRS_BORDER_LEFT	
CBRS_BORDER_RIGHT	
CBRS_BORDER_BOTTOM	
CBRS_BORDER_ANY	
CBRS_BORDER_3D	
CBRS_HIDE_INPLACE	
CBRS_NOALIGN	
CBRS_FLOATING	

CControlBar类能够保证工具栏和状态栏免于互相覆盖并且不会遮盖窗口视。但是，它所采用的方法是成套的，没有两个工具栏可以共享同一行或者列。这样，用户还是不能移动它们或者将其浮动。为了得到这些附加的功能就需要使用另外两个 MFC类：CDockBar和CDockContext。

2.4 停靠栏

为了让用户能够使用其鼠标移动工具栏， MFC应用程序沿其主窗口创建了4种统一的控制条，即停靠栏。用户可能没有注意过它们，但正是它们绘制了包围可移动工具栏的空间。更常见的是，停靠栏往往沿着应用程序主窗口缩小到不可见，以等待以后用户在其所处位置停靠工具栏。一旦工具栏被停靠，它们就展开并包围工具栏。

CFrameWnd类提供了一个叫EnableDocking()的成员函数，它可以沿窗口框架外沿创建多达4个停靠栏。CFrameWnd的EnableDocking()函数使用CDockBar类创建停靠栏。CDockBar本身派生于CControlBar类，就如同工具栏和状态栏一样可以使用其共享功能，包括 CControlBar绘制控制条周边的功能。

请参见图2-4观察停靠栏。

注意 状态栏即使是从CControlBar派生而来并受其控制，也并不支持停靠功能。

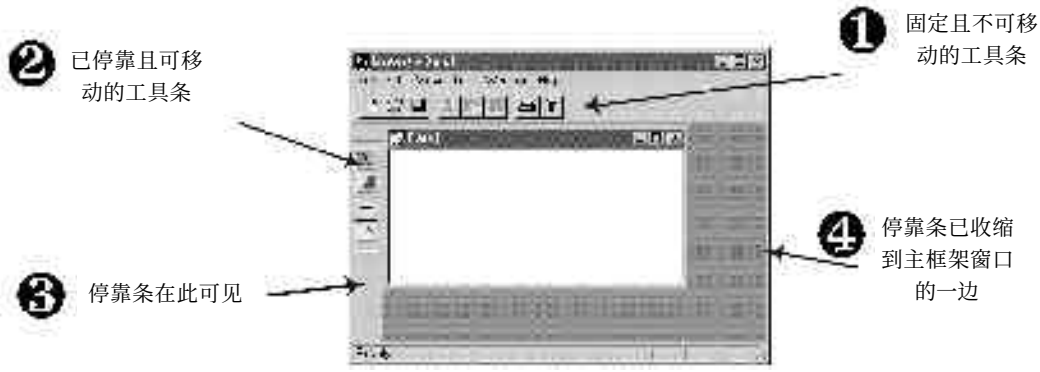


图2-4 停靠栏

2.4.1 设置停靠功能

停靠功能并不是自动地被 MFC应用程序启用的，为了启用停靠功能必须在使用AppWizard创建工程时选择 Docking toolbars选项，或者由自己动手将同样的 3行代码增加到CMainFrame类的OnCreate()函数之中，如下所示：

- CFrameWnd::EnableDocking() 该函数创建停靠栏。
- CControlBar::EnableDocking() 该函数向控制条中增加停靠和移动功能。
- CFrameWnd::DockControlBar() 该函数将控制条停靠到停靠栏。

下面就讨论这些函数所做的工作。

1. CFrameWnd::EnableDocking()

为了使主框架或者子框架窗口具备停靠功能，使用如下所示代码：

```
EnableDocking( DWORD dwStyle          // which side to create
                                      //    a docking bar on:
//    BRS_ALIGN_TOP                   // for the top
//    CBRS_ALIGN_BOTTOM               // for the bottom
//    CBRS_ALIGN_LEFT                 // for the left side
//    CBRS_ALIGN_RIGHT                // for the right side
//    CBRS_ALIGN_ANY                  // for all four sides
   );
```

可以将这些风格用 OR联合起来以提供附加效果。例如，使用 CBRS_ALIGN_LEFT | CBRS_ALIGN_TOP可以防止用户停靠在其框架窗口的底部或者右边，这是因为该位置没有停靠栏可接受它们。

CFrameWnd::EnableDocking()函数可以同时创建4个停靠栏，每个停靠栏都采用CDockBar类来创建。CDockBar类与其他MFC控件类一样，为了创建该控件，必须首先创建一个该类的实例然后再调用 Create()成员函数来创建其窗口。为了创建停靠窗口， CDockBar使用了AfxWndControlBar窗口类。

CFrameWnd类在一个名为m_listControlBars的指针链表里为它所创建的每个 CDockBar实例存储了一个指针。该链表是CPtrList类型的非文档公共成员变量。

如果要从CDockBar派生出自己的停靠栏则必须重写 CFrameWnd::EnableDocking()函数，因为很难在EnableDocking()函数中编写代码CDockBar，读者可以参考该主题下的“ 控制条趣

味编程”一节。

2. CControlBar::EnableDocking()

为了向控制条内增加必要的功能以允许其停靠，应该进行如下调用：

```
m_wndToolBar.EnableDocking( dwStyle          // which side can this bar
                                             //    be docked to:
//    CBRS_ALIGN_LEFT                        // for left side only
//    CBRS_ALIGN_RIGHT                       // for right side only
//    CBRS_ALIGN_TOP                         // for top only
//    CBRS_ALIGN_BOTTOM                      // for bottom only
//    CBRS_ALIGN_ANY                         // for any side
   );
```

这些风格可以再次使用 OR联合以获得预期的效果，例如 CBRS_ALIGN_LEFT |CBRS_ALIGN_TOP将告诉工具栏它只能停靠于主窗口的顶部或者左边，哪怕在窗口四周都具有停靠栏。

EnableDocking()函数增加到控制条内的停靠功能被包含在另一个 MFC类实例CDockContext中。只要用户双击或者拖动工具栏， CDockContext内的函数就会起作用。当用户释放一个被拖动的工具栏时， CDockContext将确定哪一个停靠栏起停靠作用以及停靠于什么位置。

3. CFrameWnd::DockControlBar()

为了确实将控制条停靠于停靠栏内，可使用如下代码调用：

```
DockControlBar(
   ( CWnd * )&m_wndToolBar          // a pointer to the CControlBar object
   LPRECT lpRect                    // location to dock the bar (optional)
   );
```

其中CFrameWnd的DockControlBar()函数确定使用什么停靠栏并将控制条停靠在该停靠栏。如何选取一个停靠栏取决于由 lpRect参数确定的最接近位置，同时还取决于如何启用控制条。例如，如果在启用控制条的时候仅指定了 CBRS_ALIGN_LEFT风格，那么CFrameWnd的DockControlBar ()函数将把控制条停靠在应用窗口的左边。没有指定位置的 CBRS_ALIGN_ANY风格则使得CFrameWnd的DockControlBar()函数自动选择首先存在的停靠栏，一般是顶部停靠栏。

CFrameWnd的DockControlBar()函数通过调用CDockBar的DockControlBar()函数停靠控制条。虽然它们名字一样，但是只有 CDockBar的这一函数才接受控制条停靠。它将指向控制条的一个指针存放在一个内部数组中。此外它还完成一些在以后章节将要讨论的工作。

1) CDockBar::DockControlBar()

对CDockBar的DockControlBar()函数调用如下所示：

```
DockControlBar(CWnd *pBar, LPRECT lpRect);
```

用户并不需要自己调用该函数， CFrameWnd的DockControlBar()函数便可完成这些工作。调用CFrameWnd的DockControlBar()函数所需要的控制条指针和位置矩形被传送给该函数以后，将被存储在CdockBar中的一个叫做m_arrBars的成员数组中。只要控制条开始停靠于其停靠栏上，该数组就起作用。该数组内的一个零指针告诉 CDockBar开始一个新行。如果在调用DockControlBar()函数的时候不精确指定位置，它只是将其添加到该数组末尾并创建一个新行

(在末尾添零)。如果指定了一个位置， DockControlBar()将识别是哪一行，然后保存它并将其插入。

2) CDockContext::DockControlBar()

CDockContext类在需要停靠控制条的时候也调用该函数。它首先确定停靠到哪个停靠栏(类似CFrameWnd的DockControlBar函数)，然后调用该控制条的CDockBar::DockcontrolBar()函数传递用户当前的鼠标位置。

2.4.2 自动改变大小和移动

用户不仅能在停靠栏内移动工具条，而且当用户重置应用程序大小的时候，如果一个或多个控制条消失在窗口框架以外，停靠栏本身也能够自动地移动其控制条，而部分被遮盖的控制条并不这样。但是，如果一个控制条完全不可见，它将会被移动到一个新行 (或新列 —— 如果该条是垂直对齐方式)。可见，将控制条停靠到特定位置是很快捷的。停靠栏区域则可以是固定的。

本主题的更多内容可参见2.7节“ 视和控制条如何共享客户区 ”。

请参见图2-5回顾停靠控制条的过程。

❶ 使用CFrameWrd ::EnableDocking()函数将停靠条添加到主窗口的各边

❷ 使用CControlBar:: EnableDocking()函数为每一个工具条加入可移动的功能

❸ 使用CFraneWnd:: DockControlBar()函数对工具条进行初始停靠

❹ CFraneWnd::DockControl Bar()函数根据邻近程度选择停靠条并调用它的CControlBar::DockControl Bar()函数

图2-5 停靠栏回顾

2.4.3 停靠栏小结

停靠控制条采用了两种MFC类以及4个函数：

1) CDockBar类沿应用程序的主窗口四周，创建至多 4个停靠栏以控制和包含其他控制条。如果停靠栏没有包含其他控制条，它就将自身缩小到看起来仿佛已经从应用程序中消失，然后等待新的控制条放置在自己上面。

2) CDockContext类处理用户鼠标单击并拖动控制条于停靠栏上这一事件，该类通过CControlBar::EnableDocking()函数为控制条创建。

3) CFrameWnd的EnableDocking()函数为窗口的每一边创建CDockBar的实例。

4) CControlBar的EnableDocking()函数创建一个CDockContext实例以允许控制条处理用户鼠标输入移动或者停靠其控制条等事件。

5) CFrameWnd的DockControlBar()函数确定将控制条停靠到哪个停靠栏。

6) CDockBar的DockControlBar()函数用于停靠栏控制其上的控制条。

以上便是用户移动并停靠其工具栏的过程。

如果用户熟悉MFC工具栏，可能会注意到在拖动工具栏离开主窗口后，它会突然被自己的框架包围，换句话说就是浮动。对这个功能，MFC提供了一个最新的叫做CDockMiniFrameWnd的类。

2.5 浮动条

CDockMiniFrameWnd类本身是从CFrameWnd派生的(请看注解)。基本上，它是一种简单的框架窗口，该窗口仅包含了一个停靠栏。因为该框架窗口没有视，所以这种情况下不需要 4个停靠栏。当CDockContext类实现了用户拖放其工具栏到空白区域的任务时，它将相应创建一个CDockMiniFrameWnd实例并将工具栏停靠于该框架内部自动创建的停靠栏上。请参见图2-6的浮动条实例。

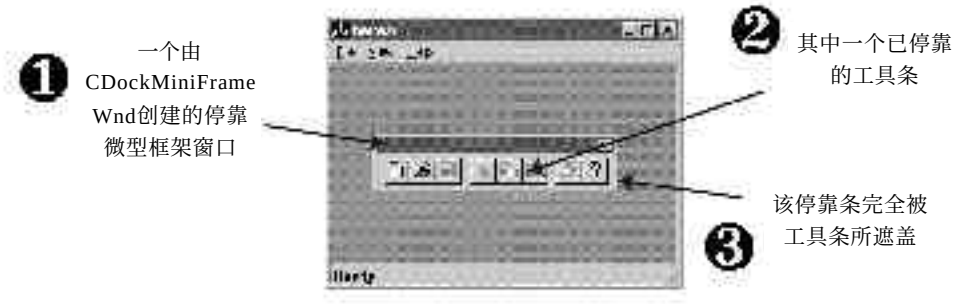


图2-6 浮动工具栏

注解 CDockMiniFrameWnd实际上派生于CMiniFrameWnd，而CMiniFrameWnd又派生于CFrameWnd。CMiniFrameWnd因而具备CFrameWnd的全部功能但比起主框架或者子框架来又更容易适应定制的应用程序。它可以有自己的菜单和工具栏，在效果上类似于一个正好处于当前应用程序中间的微型SDI应用程序。然而，不能使用AppWizard来创建该微型 SDI应用程序，而且在同一工程内混合视类和文档类可能引起麻烦。因此，最好创建一个单独的SDI应用程序而不是在自己的应用程序中创建。CMiniFrameWnd所做的最好工作可能就仅仅是支持CDockMiniFrameWnd以及浮动工具栏。

浮动工具栏具有两个停靠工具栏所不具备的特点：第一，用户仅需要拖动微型框架窗口一角就能够改变工具栏的大小；第二，用户还可以在同一个微型框架窗口内停靠一个或者多个附加工具栏。读者请参见图 2-7以了解这些特点。

注意：一个可调整大小的工具条不能和另一个位于微型框架窗口的工具条一起停靠

图2-7 改变大小和微型浮动工具栏

启用或者关闭浮动条功能需要两种额外的控制条风格，如表 2-5所示。

表2-5 浮动控制条的MFC风格

风 格	使 用 说 明
CBRS_SIZE_FIXED	防止用户或者系统重置控制条的大小
CBRS_SIZE_DYNAMIC	允许用户或者系统重置控制条大小，虽然用户可能只是改变浮动地工具栏大小
CBRS_FLOAT_MULTI	允许多个控制条被停靠于同一微型框架窗口内

可以通过调用CFrameWnd的FloatControlBar()函数将工具栏浮动。

2.6 MFC的高级控制条类小结

现在我们已经了解，CToolBar、CStatusBar类允许控制条和视类共享客户区以及许多其他功能。CDockBar和CDockContext类允许用户移动其工具栏。而 CDockMiniFrameWnd类则可以使用户浮动它们的工具栏。

接下来讨论这些类型的控制条如何与主窗口共享同一客户区。

2.7 视和控制条如何共享客户区

应用程序的客户区允许两种类型的子窗口争夺空间：视窗口和控制条窗口。在 MDI应用程序的情况下，视窗口就是 MDIClient区域。在SDI应用程序的情况下，视窗口则由 CView类创建。通过一个循环过程可以确定每个窗口占据了多少屏幕实际空间。首先调用主窗口的 GetClientRect()函数得到整个可用空间的大小。然后这个空间大小作为一个矩形参数被传送给主窗口的每个子窗口。每个窗口则为自己划分出一块空间并在该处定位，然后向下一个窗口传送一个减小的矩形空间，这样直到所有的窗口都找到空间。剩下的空间则留给视窗口。

并不是所有的子窗口都参与了这一过程，只有那些由 CControlBar的派生类所创建的窗口才这样做。这就意味着像 CToolBarCtrl这样的类或者直接通过 API调用而创建的窗口不会得到一块空间。由于视窗口并不是由 CControlBar类创建的，它也不能挖出一块场地来，但它确实得到了所有剩余的空间。由于有的时候没有什么空间留下，于是视就只好消失了。

这整个过程开始于应用程序的主窗口发生变化的时候 —— 要么是添加或者删除了工具栏，要么就是移动或者改变大小。对浮动的工具栏，用户可以改变它的形状或者增加一个新的控制条。这些功能中的任何一个通常都会导致调用 CFrameWnd的RecalcLayout()。举个例子，CFrameWnd中的OnSize()消息处理器将直接调用该函数。可以按以下方式编程调用该函数。

2.7.1 CFrameWnd::RecalcLayout()

该函数的语法表达是：

```
CFrameWnd::RecalcLayout(BOOL bNotify);
```

如果框架包括OLE框架并且要让框架重新计算其布局，那么这里的 bNotify参数就应该设为TRUE。

该函数除了调用基类函数 CWnd::RepositionBars()以外几乎什么也没有做，这事实上就是循环开始。RecalcLayout()实际完成的一件工作是为 RepositionBars()识别视窗口。如果没有视窗口，或者没有浮动的工具栏，那么 RecalcLayout()就使用RepositionBars()函数来关闭包围其

内部任何控制条的框架。

2.7.2 CWnd::RepositionBars()

该函数的语法表达如下：

```
CWnd::RepositionBars(
   UINT nIDFirst, UINT nIDLast,        // child windows to poll
   UINT nIDLeftOver,                   // id of leftover window
   UINT nFlags,                        // can be set to simply inquire
                                       //    how much space the
                                       //    control bars need
   LPRECT lpRectParam,                 // can return inquired size
   LPCRECT lpRectClient,               // substitute client area
   BOOL bStretch                       // cause bars to stretch to fit
                                       //    client area
   );
```

RepositionBars()首先确定其窗口的客户区，换句话说，它必须计算出有多少空间来放置视和控制条。为了得到这个尺寸，它只调用 CWnd::GetClientRect()函数。然后RepositionBars() 向它的任何子窗口发送一个 WM_SIZEPARENT窗口消息(特别为MFC创建的)，而这些子窗口则落在由nIDFirst和nIDLast参数限定的范围之内(是nIDLast 而不是nIDLeftOver结尾：并不一定仅仅是剩余的窗口)。传递给这些窗口的参数只有两个：客户区矩形以及每个控制条是否应当延伸以填充客户区。每一个窗口的OnSizeParent()消息处理器负责确定它需要多少客户区，并将该区域从客户区矩形中减去，使减去的空间对下一个窗口不可用。

当全部子窗口(包括控制条)都划分出自己的区域以后， RepositionBars()使用剩余的空间设置剩余窗口的大小和位置。在这种情况下即为视窗口。

虽然有好几类控制条，但WM_SIZEPARENT窗口消息则只由CControlBar中的一个消息处理器处理。这就是为什么只有用 CControlBar创建的控制条才可以划出区域的原因。 接下来讨论CControlBar的OnSizeParent()消息处理器如何确定划分出多少空间。

2.7.3 CControlBar::OnSizeParent()

该消息处理器的语法表达如下：

```
LRESULT CControlBar::OnSizeParent(WPARAM, LPARAM lParam);
```

其中lParam包含了客户区矩形。

OnSizeParent()消息处理器接收传递给每个窗口的矩形参数，确定它自己的控制条需要多少空间，从矩形中减去这一片空间然后将该参数传递给下一个窗口。为确定自己的控制条需要多少空间， OnSizeParent()函数调用了CalcDynamicLayout()函数。

CalcDynamicLayout()函数本身被每个由 CControlBar派生的控制条类 (如CToolBar、CStatusBar等)重载以确定相应类型控制条所需要的空间尺寸大小。对于可停靠工具栏，CalcDynamicLayout()在计算出其大小之前可能改变工具栏的形状。

CalcDynamicLayout()返回给OnSizeParent()函数的尺寸与其说是需求不如说是请求。例如，延伸的水平工具栏将请求一个 32 767像素的宽度。但显然没有多少监视器处理这样的分辨率，实际上可以说没有。但CalcDynamicLayout()真正要OnSizeParent()去做的是使该宽度达到当前

可用空间的最大尺寸。或者说，使控制条延伸跨越整个主窗口的客户区。

因此，OnSizeParent()最后判断它的控制条将拥有多少客户区，它总是尽力调整其控制条，使它们沿窗口框架四周占据最小的区域。一旦该函数做出了决定，控制条就移动到指定位置，其占用的大小也如上所述从客户区中扣除。

下面讨论CalcDynamicLayout()如何计算出控制条大小和改变其形状。

2.7.4 CalcDynamicLayout()和CalcFixedLayout()

实际上有两个函数可用于帮助OnSizeParent()计算控制条的大小：

CalcDynamicLayout() —— 用于对具有CBRS_SIZE_DYNAMIC风格的工具栏布局并计算其大小。

CalcFixedLayout() —— 计算其余控制条的大小。

OnSizeParent()函数并不直接调用CalcFixedLayout()，它调用的是CalcDynamicLayout()。该函数确定是否是动态工具栏，如果不是，CalcDynamicLayout()则调用CalcFixedLayout()函数。

CalcDynamicLayout()的语法表达如下：

```
virtual CSize CCalDynamicLayout( int nLength, DWORD dwMode);
```

这里的nLength参数指定控制条请求所需的长度，而dwMode则是表2-6所示的布局模式的联合。这两种参数都仅能用于具有CBRS_DYNAMIC_SIZE风格的工具栏。

注意 如果调用CalcDynamicLayout()时具有有效的长度参数，控制条必须被停靠并且不能使用LM_MRUWIDTH或者LM_COMMIT布局模式。

表2-6 CalcDynamicLayout()布局模式

模 式	使 用 说 明
LM_LENGTHY	如果不是宽度而是一个指示长度的长度参数则需要设置该模式
LM_COMMIT	告诉函数将其计算的尺寸存储于 Most Recently Used变量里，该变量可以用下一个模式得到。它还锁定工具栏的方向和形状，请参考以下的讨论
LM_MRUWIDTH	当LM_COMMIT模式被使用时，告诉函数返回其计算的最后尺寸
LM_STRETCH	告诉函数返回被延伸控制条的大小。如果使用了 LM_HORZ，控制条将水平延伸，否则将垂直延伸。控制条被延伸到填充控制条和主窗口之间的空间。由于停靠栏填充任何空白的区域，因此停靠和浮动条不能被延伸
LM_HORZ	指示控制条水平或者垂直布局
LM_HORZDOCK	告诉函数为已经停靠的工具栏返回一个尺寸
LM_VERTDOCK	停靠工具栏只有一行或者一列按钮

CalcFixedLayout()的语法表达如下：

```
CSize CalcFixedLayout(BOOL bStretch, BOOL bHorz );
```

如果需要一个延伸的尺寸而不是控制条真正的尺寸，便需要在此设置 bStretch参数。而bHorz参数则仅适用于垂直延伸的工具栏。

CControlBar中CalcDynamicLayout()函数的缺省行为只是调用 CalcFixedLayout()，而CControlBar中CalcFixedLayout()的缺省行为只是为水平延伸控制条返回尺寸数 32767,0，为垂直延伸控制条返回0,32767或者为不延伸的控制条返回 0,0。显而易见，不从CControlBar派生

的类总是不可见的。

对每种控制条(工具栏、停靠栏等)，CalcFixedLayout()和CalcDynamicLayout()的重载版本将在下面更详细讨论。

2.7.5 CToolBar::CalcFixedLayout()和CToolBar::CalcDynamicLayout()

CToolBar的CalcFixedLayout()和CalcDynamicLayout()都调用同样的辅助函数CalcLayout()函数来确定工具栏大小。CalcLayout()函数询问工具栏有多少按钮以及配置如何。如果工具栏具有CBRS_SIZE_FIXED风格，CalcLayout()将只返回基于按钮风格的工具栏的当前大小，以及这个控制条是一行还是多行。

但是，如果工具栏具有CBRS_SIZE_DYNAMIC风格，CalcLayout()就可能在计算其大小之前改变工具栏的形状。它放置工具栏的方法取决于下列顺序中的布局模式：

■ 如果工具栏处于浮动状态或者使用了LM_MRUWIDTH模式，工具栏将返回到最近使用的大小并返回该尺寸。一般这种模式常用于恢复浮动工具栏的大小(在其停靠的情况下)。

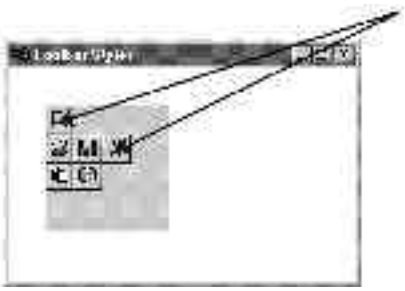
■ 如果工具栏垂直或者水平停靠，CalcDynamicLayout()将改变工具栏形状为单行或者单列按钮并请求客户区内可用的最大宽度或高度。

■ 如果长度不是-1，那么，工具栏在停靠并且没有采用LM_COMMIT模式的情况下就置为该长度。如果设置了LM_LENGTHY布局模式，那么Length指的就是宽度。

为了确认布局，必须使用LM_COMMIT布局模式。否则，由CalcDynamicLayout()返回的大小会对被请求的布局有效，但是在操作它之前工具栏本身又要保持其形状。如果使用了LM_MRUWIDTH模式，LM_COMMIT还将导致CalcDynamicLayout()存储它所计算的尺寸以供今后使用。

2.7.6 工具栏布局

为了改变工具栏的形状(使它垂直或者行数加倍以创建特定大小)，CalcDynamicLayout()在开始新的一行之前对工具栏按钮应用TBSTATE_WRAP按钮状态。正如所看到的，这是一种按钮状态而不是工具栏状态。没有可靠的方式来告诉工具栏按照特定按钮封装。可以将工具栏设置为TBSTYLE_WRAPABLE风格以便它在宽度太小的情况下自动地封装成多行。但是，这不可能获得直接使用TBSTATE_WRAP所能达到的控制程度。事实是，为了使工具栏垂直仅需要为工具栏上的每个按钮应用TBSTATE_WRAP状态即可。请参见图2-8了解TBSTATE_WRAP状态是如何用于创建多行的。



TBSTATE_WRAP工具条状态应用于该单个工具条控件窗口中的两个按钮

图2-8 封装工具栏各行

注意 为什么CalcDynamicLayout()函数会愚蠢到为了确定其工具栏形状而在其他操作正在进行的过程中来获得其大小呢？为什么工具栏不事先建立自己的外形呢？这是因为，在工具栏的邻居们都在同时执行操作的这一过程中，根本没有更合适的时间来重新塑造工具栏的外观。仅仅改变一个控制条的外观将会导致在其原位置出现一个漏洞，

在其新位置上却会引起覆盖操作。因而，只要工具栏需要被重新塑造外观，系统只改变其风格然后调用RecalcLayout()函数来执行实际改变外观的工作。

2.7.7 CStatusBar::CalcFixedLayout()

由于状态栏不能重置大小或者改变其对齐方向，所以其 CalcDynamicLayout()函数仅仅调用CalcFixedLayout()函数即可。CalcFixedLayout()函数的状态栏版本随后只是返回该状态栏在其边界大小和在所用字体尺寸的基础上所需的高度。该宽度总是返回 32 767像素值，以使状态栏延伸到窗口框架的两边。

2.7.8 CDockBar::CalcFixedLayout()

由于停靠栏也从不改变大小或者对齐方向，其 CalcFixedLayout()函数也仅仅调用CalcFixedLayout()函数。CalcFixedLayout()函数的停靠栏版本确定它需要多大空间来包围所有的被停靠控制条。但在进行这个计算之前， CalcFixedLayout()函数必须为自己进行一些布局方面的工作。

CalcFixedLayout()所做的工作就是遍历所有被停靠于其上的控制条 (记住m_arrBars包含了该列表)，并调用它们每一个的CalcDynamicLayout()函数。这与先前学习的CToolBar的布局函数是完全一样的。其返回的请求尺寸随后用于计算添加控制条的停靠栏所需的空间。如果控制条离开了父窗口的左边或者底部， CalcFixedLayout()则将使其移回原处。如果控制条离开父框架很远则不可见， CalcFixedLayout()就会创建新行(如果是垂直布局则创建新列)并把控制条移动到新行去。一旦 CalcFixedLayout()函数为停靠栏中的每一个控制条完成了这项工作，它就会计算自己在父框架内需要多大空间并将这个数值返回给 OnSizeParent()，该函数则按规矩从可用空间中扣除掉这一块。

注意 位于停靠栏上的控制条从不接收WM_SIZEPARENT消息。因为它们是停靠栏的子窗口，以上讨论的RepositionBar()函数“看不见”它们，只有它们的父窗口，也就是停靠栏才能看见。停靠栏的CalcFixedLayout()函数完成一些“冒充”的工作，为这些控制条代理调用OnSizeParent()函数以保证它们之间不相互覆盖。

2.7.9 共享客户区小结

下面介绍应用程序主窗口中的控制条如何互相共享客户区：

1) 当窗口上的内容改变时， CFrameWnd的RecalcLayout()函数即被调用。

2) RecalcLayout()随后调用CWnd的RepositionBars()函数并告诉它允许窗口中所有的控制条占用其所需空间，然后将剩余空间留给客户区。

3) RepositionBars()遍历窗口中的所有子窗口并向它们发送 WM_SIZEPARENT消息来完成这项工作。它给框架中每个还可见的控制条传递给一个逐渐缩小的矩形区域。

4) CControlBar的OnSizeParent()消息处理器为所有的控制条类型处理 WM_SIZEPARENT消息，它采取的方式是询问控制条它们需要多大的空间并随后从可用空间中将它们所需的空间减去。

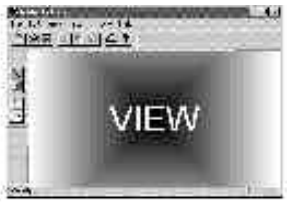
5) OnSizeParent()通过调用CalcDynamicLayout()函数计算出其控制条所需的空间，这个函

数被每个基于不同类型的控制条重载以确定这个所需的尺寸。

6) 停靠栏在返回请求尺寸之前为自己完成一点布局工作。

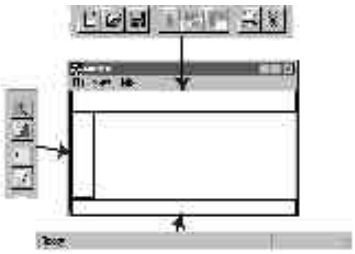
以上所述就是本节主题的全部过程，请参见图 2-9以深入学习该过程。

❶ 主框架窗口中发生一些事件，CFrameWnd::RecalcLayout()函数被调用



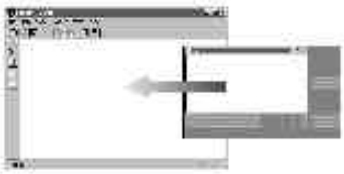
❷ RecalcLayout()获得视察窗口的ID号并调用Wnd::RepositionBars ()，在这种情况下，视是一个MDIClient窗口

❸ RepositionBars()以原始状态启动并发送给该框架窗口中的每一个子窗口WM_SIZEPARENT消息



❹ 每一个CControlBar类通过使用CalcDynamicLayout()来获取各自工具条的尺寸，然后在框架中寻找一个区域

❺ 可调整大小的工具条中的CalcDynamicLayout()甚至可能首先改变工具条的布局



❻ RepositionBars()将剩余的空间留给视窗口

图2-9 共享客户区小结

2.8 对话条

对话条是工具栏和无模式对话框相结合的产物。如同 CToolBar类创建了工具栏并控制其子窗口一样， CDialogBar类则创建对话条并控制其无模式的子对话框。 CDialogBar类本身派生于CControlBar类并再次使用了其绘制边界、停靠和浮动等通用的功能。

对话条由一个具有子风格、无边界的对话框模板产生。出于风格统一的考虑，它应该具有工具栏或者状态栏的一般尺度。例如，它应该长而窄或者高而苗条等等。外观较粗大的对话条可能使应用程序的外观显得不清晰明快。

虽然对话条可以浮动或者停靠，但它们并不能由用户自动改变其大小。但是程序员可以重载其CalcFixedLayout()函数并控制其大小。

CDialogBar::CalcFixedLayout()

如同CToolBar和CDockBar， CDialogBar也有自己的CalcFixedLayout()派生函数以占用一块主窗口区域。在这种情况下，该函数仅仅返回一些修改的 m_sizeDefault数值，该值是创建对话条的对话框模板所定义的原始尺寸。例如，假设对话条已停靠并延伸到填充应用程序窗口的顶部，那么CalcFixedLayout()将返回32767，也就是m_sizeDeafault.cy值。

如果打算控制该函数，仅需重载CDialogBar派生类的CalcFixedLayout()函数并提供自己的定义值即可。

2.9 伸缩条

应用程序最后一种可用的控制条就是伸缩条（Rebar），虽然它也被称为 Cool Bar，术语rebar可能是Resize Bar的缩写。

伸缩条具有它自己的 Windows API对等函数，可以以如下方式创建，虽然它仅仅在Windows 98版本或者安装了Internet Explorer 4.01及其以上版本以后才可用，但在介绍其内容时不必拘泥于这些细节：

```
HWND hwnd=::CreateWindowEx( ..., "ReBarWindow32",..);
```

实际上伸缩条看起来像是一种 windows API，这种API试图提供一些以前仅在MFC停靠栏中才可用的功能，以便使非 MFC应用程序也可以移动它们的工具栏。

但是，当它终于能够移动工具栏的时候，伸缩条的功能已经远远超过了简单的停靠栏。它不仅可以移动工具栏而且还可以改变其大小。不只工具栏可以得到这样的支持，任何控件窗口都可以得到其支持，例如编辑框、按钮、组合框、列表框和动画控件等等。

每种控件窗口包含在伸缩条的段(band)中，每段一个控件，但是每个伸缩条可能有无限的段。一个段增加一个提手条(grriper bar)，程序员自己可以拥有其标题并用自己的位图绘制其背景。请参见图2-10了解伸缩条的布局。

图2-10 伸缩条控件窗口

正如工具栏和状态栏一样，可以使用 3种风格来影响伸缩条的外观和行为：标准窗口风格(WS_VISIBLE,....)、通用控件风格(CCS_VERT,...)以及一种针对伸缩条的特有风格等等，该特定风格具有RBS_前缀。伸缩条风格如表2-7所示。

表2-7 伸缩控制条风格

风 格	使 用 说 明
RBS_BANDBORDERS	伸缩条控件绘制相邻段之间的线条
RBS_VERTICALGRIPPER	提手条在垂直伸缩条中垂直显示(Win98和IE4.01中才有)
RBS_AUTOSIZE	当控制条的大小改变时，伸缩条改变段的布局(Win98和IE4.01才有)

伸缩条与其停靠栏不能一起使用，虽然可以在应用程序中创建许多个伸缩条而且可以将它们垂直竖起来，但就是不能使它们停靠。所以还得创建停靠栏以停靠应用中的其他工具栏。

为了给伸缩条增加新的控制窗口，可使用如下代码：

```
::SendMessage(hWnd, RB_INSERTBAND, uIndex, (LPARAM)prbbi);
```

这里uIndex是插入段的位置，而prbbi则是指REBARBANDINFO结构，该结构定义了放进伸缩条段中的子窗口。

MFC近来只在其6.0版本中才提供了对伸缩条的支持，但这项支持与对工具栏和状态栏的支持是一样的：使用一种类简单地封装 API，一种类则提供更强大的功能。

2.9.1 CReBar和CReBarCtrl

CReBarCtrl类封装用于创建伸缩条的Windows API的方式与CToolBarCtrl封装工具栏的创建类似 。为了创建伸缩条并使用CReBarCtrl类增加一个新的段，可以使用如下代码：

```
CReBarCtrl rb;
rb.Create( DWORD dwStyle, const RECT& rect, CWnd* pParentWnd,
   UINT nID );
rb.InsertBand( UINT uIndex, REBARBANDINFO* prbbi );
```

这里uIndex和prbbi参数的含义与使用API 插入一个段时的含义是一样的。

当然，不仅仅只是封装API ，CReBar类还具有一些更先进的特点。例如，由于CReBar加入了在其他的MFC类中(如CToolBar、CStatusBar和CDialogBar)才能发现的功能。程序员可以向由CReBar创建的伸缩条中直接增加由这些类所创建的控制条，如下所示：

```
CReBar rb;
rb.Create( CWnd* pParentWnd, DWORD dwCtrlStyle, DWORD dwStyle,
   UINT nID );
CToolBar tb;
tb.Create( CWnd *pParentWnd );
tb.LoadToolBar( UINT nID );
rb.AddBar( &tb, LPCTSTR lpszText, CBitmap* pbmp, DWORD dwStyle );
```

AddBar()的lpszText和pbm参数是可选段的文本和位图。

如果打算在段中包括其他的窗口类型，例如动画控件，仅需将该控件放进对话框模板中并创建对话条，然后利用CReBar::AddBar()将其加入该条中。

这种直接装载其他CControlBar的功能使得CReBar类特别有用，但是该类没有CReBarCtrl类的附加功能。

2.9.2 CReBar::CalcFixedLayout()

如同CToolBar和CdialogBar一样，CreBar具有自己的CalcFixedLayout()派生函数，该函数返回一个请求大小值给OnSizeParent()。然而，在这种情况下，CReBar的CalcFixedLayout()函数仅仅将其全部段的大小加在一起组成一个整体传递。因为伸缩条不能被停靠或者浮动，用户当然就不能改变其大小，所以CReBar并没有CalcDynamicLayout()函数。

2.10 命令条

命令条(Command Bar)是本章最后要进行深入讨论的控制条，这是因为命令条在好几种微软应用程序中都得到了使用，其中包括Developer Studio。命令条实际上不过是一种具有扁平外观按钮的工具栏，它可以使用TBSTYLE_FLAT工具栏风格来创建。命令条的前面还有一个提手条，以帮助用户抓住该条。否则从扁平的按钮周围找到工具栏的边界再拖动将比较困难。请参考实例8以进一步了解对命令条外观进行初始化的方法。

Developer Studio内的菜单也是命令条。换句话说，Developer Studio内的菜单实际上是一个带有按钮文本的扁平工具栏，而不是按钮图标。当这些按钮之一被按下时，在该按钮之下就会直接出现一个弹出菜单，该类按钮则表现出标准下拉菜单的外观。

以这种方式使用工具栏虽然并不会为应用程序增加任何附加功能，而且实际上需要花费

额外的工作来实现这种方式，但这样将使应用程序看起来与众不同。

2.11 控制条窗口小部件风格

每种类型的控制条都具有各自的风格，这些风格使得控件具有不同的外观。

2.11.1 工具栏按钮风格

一些更有趣的工具栏按钮风格包括：

■ TBSTYLE_SEP，一般用于创建工具栏按钮之间的间隙 (gap)，以便建立按钮分组。指向按钮位图表面的索引成为以像素计算的间隙大小。

■ TBSTYLE_AUTOSIZE，仅用于Windows 98或者安装了IE 4.01以上版本的应用程序。该风格中每个按钮的大小基于其中的文本计算，它经常用于模仿菜单条的工具栏。

■ TBSTYLE_DROPDOWN，使工具栏产生TBN_DROPDOWN控件通知消息而不是产生来自按钮的命令消息。它具有 TBSTYLE_AUTOSIZE类似的使用局限性，也用于实现菜单工具栏。

■ TBSTYLE_EX_DRAWDDARROWS，与TBSTYLE_DROPDOWN联合用于创建一种按钮，这种按钮中嵌入了一个下拉箭头。该风格应用时必须使用 TB_ SETEXTENDEDSTYLE窗口消息。

用户可以为自己的工具栏按钮应用好几种风格以改变它们的外观。但是除了上面讨论的TBSTATE_WRAP状态之外，其他的多数状态已经通过 OnCmdUI消息处理器由MFC处理，其中包括禁用按钮或者为按钮设置复选标记。

2.11.2 状态栏窗格风格

状态栏窗格的一些更有趣的风格如下所示：

■ SBPS_NOBORDERS，导致窗格凹进控制条而不出现。该风格自动应用于 MFC出现帮助消息的应用程序状态栏的第一个窗格。如果要让该窗格也凹进去，那就需要为窗格 0使用接下来的两种风格。

■ SBPS_NORMAL，导致窗格出现下凹并且不延伸。

■ SBPS_STRETCH，导致窗格延伸并填充延伸状态栏的空白区域。该风格只能使用于一个窗格，通常是第一个，用于提供在那里出现的帮助信息。

■ SBPS_POPOUT，导致窗格呈现突起。使用用于更新用户界面的 CCmdUI的成员函数SetCheck(TRUE)可以得到同样的效果。

■ SBPS_OWNERDRAW，允许程序员绘制单独的窗格，为了做到这一点，必须从CStatusBar派生自己的类并重载DrawItem()成员函数。可以使用该风格在状态窗格中显示图标或者使用颜色。

2.11.3 伸缩条段风格

一些更有趣的伸缩条段的风格如下：

■ RBBS_BREAK，导致控制条开始新行/列。

■ RBBS_CHILDEDGE，导致段在其子窗口的顶部和底部绘制边界。

■ RBBS_FIXEDSIZE，防止段被改变大小，因此不显示提手条。

■ RBBS_NOVERT，在伸缩条垂直对齐的情况下隐藏该段。

■ RBBS_GRIPPERALWAYS和RBBS_NOGRIPPER，导致提手条总是(或是从不)显示。仅在Windows 98和IE 4.01及其以上系统中才可用。

2.12 设计自己的控制条

虽然MFC控制条类已经减轻了程序员使用控制条的许多麻烦和头痛，但是它们是已经实现了的，这就使得有时，如果不依赖一些其他未列出的特点，是不可能修改它们的一些缺省行为表现的。所以在某些情况下甚至需要重写 MFC功能，虽然这种情况不多见。

以下是一些修改和增强控制条功能的方法。

2.12.1 重载CControlBar::CalcDynamicLayout()

这也许是增强控制条功能最普通的方法。重载 CalcDynamicLayout()函数可以确定控制条的大小。因为当前MFC并不支持重置对话条大小，所以首先可以自己增加该功能，以向用户提供一种请求新尺寸的机制。然后将新的尺寸参数返回给 CalcDynamicLayout()函数。无法自动定义像Dialog Editor那样具有多个列的工具栏，替代的办法就是在自己的工具栏派生类中重载CalcDynamicLayout()函数。在这个类中必须对适当的按钮使用 TBSTATE_WRAP风格然后返回合适的尺寸。你甚至可以在基于工具栏所停靠的停靠栏的基础之上重新对齐工具栏。

2.12.2 增加WM_SIZEPARENT消息处理器

正如前面所提到的，应用程序中的主窗口的所有子窗口都接收 WM_SIZEPARENT窗口消息，但目前只有CControlBar类具有针对该消息的处理器函数，它允许 CControlBar窗口在客户区划分出一块小区域。因此，对这个过程增加一个非控制条窗口仅仅意味着为自己的窗口类增加了一个WM_SIZEPARENT消息处理器。然而，既然添加到这一处理过程的其他类型的窗口可能位于对话条内，而对话条可以自然地获得 WM_SIZEPARENT消息处理器，所以程序员也许从来不会需要自己来创建该处理器。

2.12.3 重载CMainFrame::RecalcLayout()

控制条在应用程序主窗口中的布局如何得到控制呢？这可以通过重载CMainFrame::RecalcLayout()函数或者CChildFrame::RecalcLayout()函数来完成。通过手工进行布局，可以：

■ 确定哪个条首先得到分配区域。

■ 总是在特定的位置放置特定的控制条。

然而 MFC已经使这种过程变得不可能，除非重写函数 CFrameWnd::RecalcLayout()和CWnd::RepositionBars()。因为这些函数的功能将来会由微软增强，而这些函数现在又不得不由程序员自己来实现，所以，如果对重写 MFC函数感到棘手，那么大可放心：这都是些内容不大可能改变的小函数。

如果修改这些函数可以带来很大的好处，那么程序员完全可以通过从 MFC的\src目录下拷贝RecalcLayout()和CWnd::RepositionBars()函数版本来开始这项工作。这些函数包含在

wincore.cpp和winfrm.cpp中，而且它们的内容通俗易懂，这些函数如何修改就留给读者自己领会了。

2.12.4 从CDockBar派生

为了修改并增强自己的停靠栏，需要再次重写 MFC函数。但是该函数规模较小，而且在未来MFC版本中也不可能会改变。停靠栏一般由 CFrameWnd的EnableDocking()函数内部的CDockBar类创建。但是当 EnableDocking()函数创建这个类的时候，使用了固定代码名字CDockBar，这就不可能替换自己的 CDockBar派生类。因此，如果真想要建立自己的CDockBar版本，就需要在CMainFrame内部重新创建EnableDocking()的功能。同样的，也可以从拷贝MFC的\src目录下winfrm.cpp内的原始代码开始着手这项工作。

2.13 实例

本书与控制条相关的例子包括：

- 实例1——为工具栏增加标记(logo)。
- 实例2——为工具栏增加动画标记。
- 实例3——创建命令条。
- 实例4——可编程工具栏。

2.14 总结

本章讨论了工具栏和状态栏，它们都是长而窄的控件窗口，这类控件窗口在自身内部绘制按钮和窗格。本章还讨论了如何利用 Windows API和MFC类创建控制条，而MFC类方法则使得这项工作简单化。然后讨论了控制条如何停靠以及它们之间以及与视窗口之间如何共享主窗口的客户区。最后讨论了修改并增强控制条的几种可能的方式。

接下来将要讨论MFC应用程序如何与其所处的环境通信。