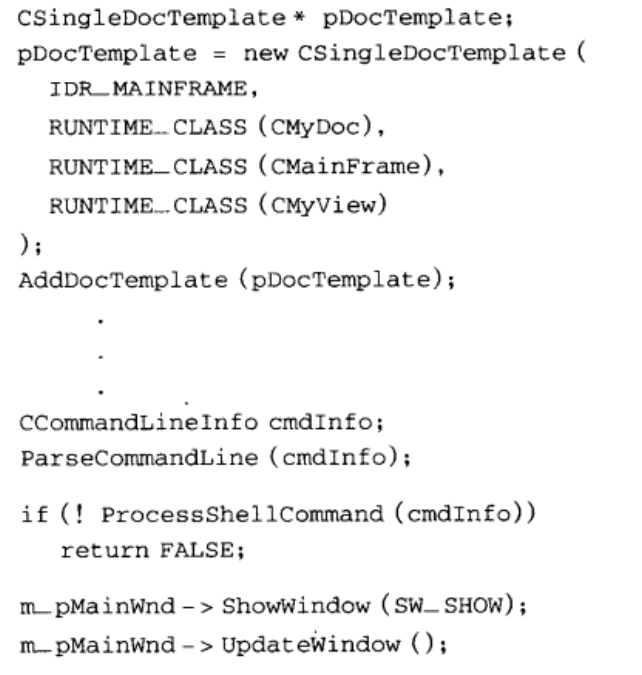
文档视图体系结构

# 文档、视图和单文档界面

## 文档视图基础知识

### InitInstance函数

SDI应用程序的InitInstance函数，含有以下内容：



首先，从MFC的CSingleDocTemplate类创建了一个SDI文档模板。该模板是SDI文档/视图应用程序的最主要的部分。它表示了用来管理应用程序数据的文档类、包含数据视图的框架窗口类、用来绘制可视数据表示的视图类。文档模板还保存了资源ID，主结构用它来加载菜单、加速键以及其他形成应用程序用户界面的资源。

其次，使用AddDocTemplate将它添加到由应用程序对象保存的文档模板列表中。用此方法注册的每个模板都定义了一个应用程序支持的文档类型。SDI应用程序只注册一个文档类型，MDI应用程序可以注册多个类型。

然后，通过CWinApp::ParseCommandLine并用反映命令行输入参数的值来初始化CCommandLineInfo对象，其中通常包含文档文件名。

然后，使用ProcessShellCommand处理命令行参数。它首先调用CWinApp::OnFileNew来启动应用程序，如果文件名没有在命令行上输入就使用空文档，或者在指定了文件名的情况下使用CWinApp::OpenDocumentFile来加载一个文档。正是在程序执行的这个阶段，主结构用保存在文档模板中的信息来创建文档、框架窗口和视图对象。初始化成功返回TRUE，否则返回FALSE。

最后在屏幕上显示应用程序的框架窗口。

在应用程序被启动，文档、框架窗口和视图被创建之后，消息循环就开始工作了。

### 文档对象

文档是程序数据的抽象表示，在数据的保存和给用户提供数据之间划分了清晰的界限。通常，文档对象为其他对象，特别是视图，提供了公用成员函数，使用它可以访问文档的数据。

CDocument的主要操作包括：

|  |  |
| --- | --- |
| 函数 | 说明 |
| GetFirstViewPosition | 返回POSITION，用来传递给GetNextView，以开始列举与文档关联的视图 |
| GetNextView | 返回CView指针，指向与文档关联的视图列表中的下一个视图 |
| GetPathName | 检索文档的文件名称和路径，例如C:\Documents\MyFile.doc；如果文档还未命名则返回空字符串 |
| GetTitle | 检索文档的标题，例如MyFile；如果文档没命名则返回空字符串 |
| IsModified | 如果文档包含未保存的数据就返回非零值，否则为零 |
| SetModifiedFlag | 设置或清除文档中已修改的标志，该标志说明文档是否包含没有保存的数据 |
| UpdateAllViews | 通过调用每个视图的OnUpdate函数来更新与文档关联的所有视图 |

每次修改了文档数据之后都要调用SetModifiedFlag。

OnUpdate是CView的保护型成员函数，所以在AfxWin.h中将CDocument声明为CView的友元。

CDocument主要的可覆盖函数：

|  |  |
| --- | --- |
| 函数 | 说明 |
| OnNewDocument | 在新文档被创建时由主结构调用。覆盖它是为了每次创建新文档时都对文档对象应用专门的初始化 |
| OnOpenDocument | 在从磁盘上装载文档时由主结构调用。覆盖它是为了每次装载新文档时都对文档对象进行专门的初始化 |
| DeleteContents | 主结构调用它来删除文档的内容。覆盖它是为了在文档关闭之前释放分配给文档的内存和其他资源 |
| Serialize | 主结构调用它在文档和磁盘之间串行化输出和输入。覆盖它是为了提供针对文档的串行化程序以便文档可以被装载和保存 |
| OnCloseDocument | 在文档关闭时调用 |
| OnSaveDocument | 在文档保存时调用 |
| SaveModified | 在包含未保存数据的文档被关闭之前调用，用来询问用户是否保存已做的改动 |
| ReportSaveLoadException | 串行化过程中有错误发生时调用 |

在SDI应用程序中，MFC只在应用程序启动时实例化文档对象一次，因此文档类的构造函数也只执行一次初始化。如果想随时在文档被创建或打开时进行某种初始化工作，就必须覆盖OnNewDocument或OnOpenDocument。

MFC提供了OnNewDocument和OnOpenDocument的默认实现，所以在覆盖这个两个函数时，应该调用基类中相同的函数。

一般来说，MFC应用程序更经常覆盖OnNewDocument而不是OnOpenDocument。因为OnOpenDocument间接调用了Serialize函数，它使用从文档文件中检索到的值初始化文档的永久数据成员。只有非永久数据成员才需要在OnOpenDocument中初始化。

SDI程序在新文档被创建或打开之前，主结构调用文档对象的虚拟DeleteContents函数来删除文档中已存在的数据。因此可覆盖DeleteContents并利用它释放分配给文档的任意资源，还可以执行其他必要的清理工作为重新使用文档对象做准备。MDI程序通常也是这样，不同之处在于，它们是在用户打开和关闭文档时各自创建和销毁的。

### 视图对象

视图类的GetDocument函数返回m\_pDocument，m\_pDocument数据成员中保存了指向视图相关文档对象的指针。

CView的可覆盖函数：

|  |  |
| --- | --- |
| 函数 | 说明 |
| OnDraw | 被调用来绘制输出文档的数据。覆盖是为了绘制文档视图 |
| OnInitialUpdate | 视图第一次附加到文档时被调用。覆盖是为了每次在文档被创建或加载时都初始化视图对象 |
| OnUpdate | 在文档的数据已经修改或视图需要更新时调用。覆盖是为了实现有效更新功能，只重画视图中需要重画的部分而不重画整个视图 |

文档试图应用程序中，WM\_PAINT消息由OnPaint函数处理，它使用CPaintDC对象来完成绘制工作，并用指向CPaintDC对象的指针来调用视图的OnDraw函数。OnDraw函数使用的是主结构提供的设备描述表指针而不是初始化自己的设备描述表，这样可以使用相同的程序实现向窗口输出、打印和打印预览。

SDI应用程序中，视图和文档一样，只构造一次然后重复使用。每当文档被打开或新建时都要调用视图的OnInitialUpdate函数。OnInitialUpdate的默认实现要调用OnUpdate，而OnUpdate的默认实现将使视图客户区无效并执行重绘。在覆盖后的OnInitialUpdate中应该调用基类的版本，否则新文档被打开或创建时视图不会被更新。使用OnInitialUpdate来初始化视图类的数据成员，并在单文档基础上执行其他与视图相关的初始化。

MDI应用程序中，覆盖OnActiveView视图可确定何时激活何时无效。如果视图被激活，OnActiveView的第一个参数是非零值，如果将要无效则为零。第2个和第3个参数分别标识即将被激活或无效的视图。

### 框架窗口对象

SDI应用程序只有一个框架窗口CFrameWnd，它被用作应用程序的顶层窗口并用来包含视图；MDI应用程序使用CMDIFrameWnd用作顶层窗口，而CMDIChildWnd窗口在顶层窗口中浮动用来包含应用程序文档的视图。

CFrameWnd类提供了OnClose和OnQueryEndSession处理程序，确保用户有机会在应用程序被关闭或Windows关闭之前保存未保存的数据。

### 动态对象创建

MFC的DECLARE\_DYNCREATE和IMPLEMENT\_DYNCREATE宏提供了创建动态创建的类的功能。需要做的工作：

* 从CObject派生对象；
* 在类声明中调用DECLARE\_DYNCREATE。DECLARE\_DYNCREATE只接收一个参数，即可动态创建类的名字；
* 在类声明的外部调用IMPLEMENT\_DYNCREATE。IMPLEMENT\_DYNCREATE接收两个参数：可动态创建类的名字和其基类的名字。

用类似下列语句使用这些宏来创建类实例：

RUNTIME\_CLASS(CMyClass)->CreateObject();

DECLARE\_DYNCREATE宏在类声明中添加了3个成员：一个静态CRuntimeClass数据成员、一个名为GetRuntimeClass的虚拟函数以及一个静态函数CreateObject。

DECLARE\_DYNCREATE(CMyClass)

等价于

Public:

static const AFX\_DATA CRuntimeClass classCMyClass;

virtual CRuntimeClass\* GetRuntimeClass() const;

static CObject\* PASCAL CreateObject();

IMPLEMENT\_DYNCREATE通过类名和类实例大小等这样的信息来初始化CRuntimeClass结构，它还提供了GetRuntimeClass和CreateObject函数。CreateObject的实现方式：

CObject\* PASCAL CMyClass::CreateObject()

{

return new CMyClass;

}

### SDI文档模板的其他内容

SDI应用程序创建文档模板对象时第一个参数是一个等于IDR\_MAINFRAME的整数值。它用来表示下面四个资源：

* 应用程序图标；
* 应用程序的菜单；
* 伴随菜单的加速键列表；
* 文档字符串，首先指定应用程序为文档生成的默认文件扩展名以及未命名文档的默认名字。

SDI程序中，主结构创建顶层窗口的过程为：先用保存在文档模板中的运行时创建类的信息来创建一个框架窗口对象，然后再调用该对象的LoadFrame函数。LoadFrame创建一个框架窗口并一次装载相关的菜单、加速键和图标。因此应给所有的资源分配相同的ID，并把ID传给LoadFrame。

文档字符串是一个字符串资源，由7个用”\n”字符分割的字符串组成。对SDI应用程序，子字符串从左到右依次具有如下含义：

1. 出现在框架窗口标题栏的标题。通常是应用程序的名称。
2. 分配给新文档的标题。如果被忽略，默认值是”Untitled”。
3. 一个描述文档类型的名字，在注册了两个以上文档类型的MDI应用程序中，当用户选中New命令后，它将和其他文档类型一起出现在对话框中。SDI应用程序中不使用此字符串。
4. 一个描述文档类型的名字，带有包含文件扩展名的通配符文件说明。例如“Drawing Files(\*.drw)”在Open和Save As对话框中使用次字符串。
5. 对于某类型文档的默认文件扩展名，例如”.drw”。
6. 不带空格的名字用来标识注册时的文档类型，例如“Draw.Document”。如果应用程序调用CWinApp::RegisterShellFileTypes来注册此文档类型，此字符串就成了以文档的文件扩展名命名的HKEY\_CLASSES\_ROOT子键的默认值。
7. 一个描述文档类型的名字。例如“Microsoft Draw Document”。与文档字符串中的前一个字符串不同，此字符串中可以包含空格。如果应用程序使用CWinApp::RegisterShellFileTypes来注册此文档类型，此字符串就是命令解释器在属性页中显示的便于人们阅读的名字。

如果需要忽略个别字符串，在”\n”后紧跟一个”\n”即可。

可以使用CDocTemplate::GetDocString函数从文档字符串中检索各个子字符串，如：

CString strDefExt;

pDocTemplate->GetDocString(strDefExt, CDocTemplate::filterExt);

将文档的默认文件扩展名复制到strDefExt中。

### 用操作系统命令解释器注册文档类型

要实现通过应用程序打开被双击的文档等类似操作，应用程序必须用操作系统命令解释器来注册其文档类型，这就需要在注册表的HKEY\_CLASSES\_ROOT部分添加一系列输入项，用来标识每个文档的文件扩展名以及用来打开和打印该类文件的命令。

可以使用两种方法进行这种注册：一种是提供REG文件，用户可以把它合并到注册表中；另一种是使用::RegCreateKey、::RegSetValue和其他win32注册函数通过编程将必要的输入项添加到注册表中。

MFC应用程序简化了这种处理，调用CWinApp::RegisterShellTypes并在调用AddDocTemplate之后传递一个TRUE参数可以在应用程序、其创建的文档及操作系统命令解释器之间铸就关键的链接。

CWinApp函数EnableShellOpen给MDI应用程序添加了漂亮的特性。如果MDI应用程序使用RegisterShellOpen和EnableShellOpen注册了其文档类型，在应用程序运行时如果用户双击了文档图标，命令解释器并不会自动启动第二个应用程序实例；它会使用动态数据交换DDE给已存在的实例发送“打开”命令及文档的文件名称。这样，文档就出现在顶层MDI框架内的新窗口中。

### 命令传送

SDI应用程序中“命令消息”的传送过程为（只有命令消息和用户界面更新才遵循传送机制）：



习惯上，New、Open和Exit命令由应用程序对象处理，CWinApp提供了OnFileNew、OnFileOpen、OnAppExit命令处理程序。Save、SaveAs命令通常由文档对象处理，它提供了Document::OnFileSave和Document::OnFileSaveAs的默认处理程序。显示和隐藏工具栏和状态栏的命令由框架窗口使用CFrameWnd成员函数来处理。其他大多数命令由文档或视图处理。

标准Windows消息如WM\_CHAR、WM\_LBUTTONDOWN、WM\_CREATE以及WM\_SIZE必须在接收这些消息的窗口中进行处理。鼠标和键盘消息通常传给视图，大多数其他消息则传给框架窗口。文档对象和应用程序对象从不接收非命令消息。

## MFC提供的视图类

常用的视图类：

|  |  |
| --- | --- |
| 类名 | 说明 |
| CView | 所有视图类的基类 |
| CCtrlView | CEditView、CRichEditView、CListView和CTreeView的基类，用来创建基于其他控件类型的视图类 |
| CEditView | 封装了多行编辑控件，并添加了打印、查找及查找替换功能 |
| CRichEditView | 封装了多功能编辑控件 |
| CListView | 封装了列表视图控件 |
| CTreeView | 封装了树视图控件 |
| CHtmlView | 从HTML文件和其他Microsoft Internet Explorer WebBrowser控件支持的媒体创建视图 |
| CScrollView | 给CView添加了滚动功能，是CFormView的基类 |
| CFormView | 实现可滚动的从对话框模板创建的“窗体”视图 |
| CRecordView | CFormView的派生类，用来显示从ODBC数据库获得的记录 |
| CDaoRecordView | CRecordView的DAO版本 |
| COleDBRecordView | CRecordView的OLE DB版本 |

### 滚动视图CScrollView

CScrollView靠自己处理滚动任务，除了实现OnDraw以外可以不对它做任何工作。实现OnDraw的方法与在CView中一样。除了希望有效使用它优化滚动操作。

要使用CScrollView，除指定CScrollView为基类外，还需覆盖视图类的OnInitialUpdate，并调用SetScrollSizes来指定视图的逻辑尺寸。通过此方式告诉MFC可滚动视图占据的区域大小。

CScrollView::SetScrollSizes接受4个参数，其中2个为可选参数，参数按顺序为：

* 指定映射模式的整数(必要，可以指定除MM\_ISOTROPIC和MM\_ANISOTROPIC以外的任何模式)
* 指定视图逻辑尺寸的SIZE结构或CSize对象(必要)
* SIZE结构或CSize对象，用来指定“页尺寸”，即单击滚动轴时MFC对视图的滚动量(可选)
* SiZE结构或CSize对象，用来指定行尺寸，即单击滚动条箭头时MFC对视图的滚动量(可选)

使用CScrollView时应该记住两个原则：

* 如果在视图中的OnDraw函数之外绘制输出，就要调用CScrollView::OnPrepareDC让MFC在输出中考虑映射模式和滚动位置的影响。
* 如果响应鼠标消息时执行命中测试，就要使用CDC::DPtoLP将单击处的坐标从设备坐标转换为逻辑坐标，从而在命中测试中考虑到映射模式和滚动位置的影响。

CSCrollView的工作方式为：当滚动事件发生时，CScrollView就用OnVScroll和OnHScroll消息处理程序捕获随后的消息，并调用::ScrollWindow来水平或垂直滚动视图。稍后，视图的OnPaint函数将被调用，以绘制由::ScrollWindow造成的失效窗口的一部分。CScrollView的OnPaint程序：

CPaintDC dc(this);

OnPrepareDC(&dc);

OnDraw(&dc);

CScrollView覆盖OnPrepareDC并在其中调用CDC::SetMapMode来设置映射模式，调用CDC::SetViewportOrg将视口原点转换为等于水平和垂直滚动位置的量。因此，在OnDraw重绘视图时滚动位置就被自动考虑进去了。

#### CScrollView的常用操作函数：

|  |  |
| --- | --- |
| 函数 | 说明 |
| GetScrollPosition | 检索当前水平或垂直滚动位置，返回CPoint |
| ScrollToPosition | 滚动到给定位置 |
| GetTotalSize | 测量视图的逻辑宽度和高度，返回CSize |
| SetScaleToFitSize | 将整个逻辑视图缩放到物理视图中。要恢复视图的默认滚动方式，再次调用SetScrollSizes即可。 |

#### 优化滚动操作

优化OnDraw的关键是CDC函数GetClipBox。它在传递给OnDraw的设备描述表对象中被调用，GetClipBox用无效矩形的逻辑坐标下的尺寸和位置来初始化RECT结构或CRect对象，如下所示：

CRect rect;

pDC->GetClipBox(&rect);

初始化的CRect可以区分需要重绘的视图部分。如何使用此信息完全取决于应用程序。可以将返回的坐标转换为行列号等等。

如果OnDraw视图绘制整个视图，GDI就会通过剪切无效矩形外的像素点来消除不必要的输出。但是剪切需要时间，结果是根据OnDraw视图绘制无效矩形外的内容而浪费的CPU周期的不同，滚动操作的效果可能很好，也可能非常糟糕。

### 树形视图CTreeView

CTreeCtrl给树形视图控件CTreeView提供程序接口。CTreeView函数GetTreeCtrl返回该控件的CTreeCtrl引用。

#### 树形视图样式

修改树形视图样式，只要在传递给PreCreateWindow的CREATESTRUCT结构中的style字段对它们进行或运算即可。

|  |  |
| --- | --- |
| 样式 | 说明 |
| TVS\_HASLINES | 添加线段，将子项目和其父项目连接起来 |
| TVS\_LINESATROOT | 添加线段，将分层结构的顶层或称为根的项目连接起来。只有指定了TVS\_HASLINES，此样式才有效 |
| TVS\_HASBUTTONS | 给具有子项目的项目添加带有加号或减号的按钮。单击该按钮可以展开或折叠相关子树 |
| TVS\_EDITLABELS | 使置换式标签编辑通知有效 |
| TVS\_DISABLEDDRAGDROP | 使拖放通知无效 |
| TVS\_SHOWSELALWAYS | 指定当前选中的项目总被加亮显示。默认状态下，控件失去输入焦点时加亮显示将被消除 |

#### 设置图形

想要在树形视图中添加图形，必须创建和初始化一个图形列表，并使用CTreeCtrl::SetImageList把它分配给树形视图。

MFC中使用CImageList表示图形列表。可以把图形列表当做具有相似尺寸位图的集合，其中每个位图都是由一个从0开始的索引号标识。

CImageList il;

il.Create(IDB\_IMAGES, 16, 1, RGB(255, 0, 255));

从包含多个图形的位图资源IDB\_IMAGES中创建一个图形列表。第2个参数说明每个图形都具有16像素宽。最后一个参数指定图形列表的透明颜色。当图形列表中的图形在树形视图中被显示时，只有非指定颜色像素才出现。

GetTreeCtrl().SetImageList(&il, TVSIL\_NORMAL);

将图形列表和树形视图联系在一起。TVSIL\_NORMAL告诉树形视图，图形列表中的图形将被用来表示选中和未选中的项目。可以将单独的TVSIL\_STATE图形列表分配给图形视图，来表示具有由应用程序定义的状态的项目。注意在树视图被销毁之前，不能销毁图形列表。否则，图形将从控件中消失。

#### 插入项目

CTreeCtrl::InsertItem给树形视图控件添加一个项目。根项目是通过指定TVI\_ROOT作为父亲来创建的。

#### 树形视图控件常用操作

|  |  |
| --- | --- |
| 样式 | 说明 |
| DeleteItem | 从控件中删除项目 |
| DelteAllItems | 删除所有的项目 |
| Expand | 展开或折叠一个子树 |
| SetItemText | 修改一个项目的标签 |
| GetItemText | 检索标签 |
| SortChildren | 排列子树中的项目 |
| GetSelectedItem | 检索所选项目的句柄 |
| GetParentItem | 获取父项目，其他类似接口GetChildItem、GetNextItem、GetNextSiblingItem |

#### 树形视图通知

通知以WM\_NOTIFY消息的形式出现，在大多数情况下，lParam指向一个NM\_TREEVIEW结构，其中包含有关产生消息的事件的附加信息。

|  |  |
| --- | --- |
| 通知 | 发送条件 |
| TVN\_BEGINDRAG | 用鼠标左键开始拖放操作。如果控件具有TVS\_DISABLEDRAGDROP样式则不发送 |
| TVN\_BEGINRDRAG | 用鼠标右键开始拖放操作。如果控件具有TVS\_DISABLEDRAGDROP样式则不发送 |
| TVN\_BEGINLABELEDIT | 开始标签编辑操作。只有在控件具有TVS\_EDITLABELS样式时才发送 |
| TVN\_ENDLABELEDIT | 完成标签编辑操作。只有在控件具有TVS\_EDITLABELS样式时才发送 |
| TVN\_GETDISPINFO | 控件需要别的信息来显示项目。如果项目文本是LPSTR\_TEXTCALLBACK或图形索引是I\_IMAGECALLBACK时才发送 |
| TVN\_DELETEITEM | 一个项目将被删除 |
| TVN\_ITEMEXPANDED | 子树已经展开或折叠。NM\_TREEVIEW结构中的action字段表明是展开还是折叠 |
| TVN\_ITEMEXPANDING | 子树即将展开或折叠 |
| TVN\_KEYDOWN | 控件具有输入焦点时一个键被按下 |
| TVN\_SELCHANGED | 选中项目已经更改 |
| TVN\_SELCHANGING | 选中项目即将更改 |
| TVN\_SETDISPINFO | TV\_DISPINFO结构中的信息需要更新 |

### MFC中的本地文件浏览功能

Win32::GetLogicalDrives函数标识系统中的逻辑驱动器。它返回一个DWORD值，其中各二进制位标识驱动器的有效性：位0与驱动器A对应，位1与驱动器B对应，等等。

向::GetDriveTypes函数提供一个指向到驱动器根目录的路径的字符串，返回一个标识驱动器类型的UNIT值，可能的返回值为：

|  |  |
| --- | --- |
| 返回值 | 含义 |
| DRIVE\_UNKNOWN | 驱动器类型未知 |
| DRIVE\_NO\_ROOT\_DIR | 驱动器没有根目录 |
| DRIVE\_REMOVABLE | 驱动器是可移动的(对于软驱和其他可移动媒体驱动器，如Zip驱动器，返回此值) |
| DRIVE\_FIXED | 驱动器是固定的(对硬盘返回此值) |
| DRIVE\_REMOTE | 驱动器是远程驱动器(对网络驱动器返回此值) |
| DRIVE\_CDROM | 驱动器是CD-ROM驱动器 |
| DRIVE\_RAMDISK | 驱动器是RAM驱动器 |

int nPos = 0;

int nDrivesAdded = 0;

CString string = \_T("?:\\");

DWORD dwDriveList = ::GetLogicalDrives();

while (dwDriveList) {

if (dwDriveList & 1) {

string.SetAt(0, \_T('A') + nPos);

UINT nType = ::GetDriveType(pszDrive);

switch (nType) {

case DRIVE\_REMOVABLE:

......

}

}

dwDriveList >>= 1;

nPos++;

}

使用::FindFirstFile和::FindNextFile枚举目录的代码：

HANDLE hFind;

WIN32\_FIND\_DATA fd;

CString strPath = pszPath;

if (strPath.Right(1) != \_T("\\")) {

strPath += \_T("\\");

}

strPath += \_T("\*.\*");

if ((hFind = ::FindFirstFile(strPath, &fd)) == INVALID\_HANDLE\_VALUE) {

return 0;

}

do {

if (fd.dwFileAttributes & FILE\_ATTRIBUTE\_DIRECTORY) {

CString strComp = (LPCTSTR)&fd.cFileName;

if ((strComp != \_T(".") && strComp != \_T(".."))) {

CString strNewPath = pszPath;

if (strNewPath.Right(1) != \_T("\\")) {

strNewPath += \_T("\\");

}

strNewPath += (LPCTSTR)&fd.cFileName;

......

}

}

} while (::FindNextFile(hFind, &fd));

::FindClose(hFind);

### 列表视图CListView

列表视图支持四种表现样式：大图标、小图标、列表、报表模式。列表视图中的项目可以包含文本和图形，这些项目可以具有只能是文本的“子项目”，其中包含有关相关项目的其他信息。当控件处于报表模式时子项目可见。

列表视图通过CListView::GetListCtrl获得列表视图内部的控件的CListCtrl的引用。

#### 修改列表视图样式

在派生类中覆盖PreCreateWindow并给视图应用一个以上的默认样式。列表视图样式包括的部分样式：

|  |  |
| --- | --- |
| 样式 | 说明 |
| LVS\_ICON | 选择大图标模式 |
| LVS\_SMALLICON | 选择小图标模式 |
| LVS\_LIST | 选择列表模式 |
| LVS\_REPORT | 选择报表模式 |
| LVS\_NOCOLUMNHEADER | 删除通常出现在报表模式中的头标控件 |
| LVS\_NOSORTHEADER | 使LVN\_COLUMNCLICK通知失效，默认状态下该通知在报表模式中的列头被单击时发送 |
| LVS\_ALIGNLEFT | 在大图标和小图标模式下沿左边框对齐项目 |
| LVS\_ALIGNTOP | 在大图标和小图标模式下沿顶边对齐项目 |
| LVS\_AUTOARRANGE | 在大图标和小图标模式下自动按行列排列项目 |
| LVS\_EDITLABELS | 使原位编辑标签有效 |
| LVS\_NOLABELWRAP | 在大图标模式下限制标签为单行显示 |
| LVS\_NOSCROLL | 使滚动失效。默认时有效。 |
| LVS\_OWNERDRAWFIXED | 指定响应WM\_DRAWITEM消息时控件所有者将绘制项目 |
| LVS\_SHAREIMAGELISTS | 防止列表视图在自身被删除时去自动删除与它关联的图形列表 |
| LVS\_SINGLESEL | 使多重选择支持无效 |
| LVS\_SHOWSELALWAYS | 指定当前选中的项目总被加亮显示。默认状态下，控件失去输入焦点时加亮显示将被消除 |
| LVS\_SORTASCENDING | 指定项目按字母升序排列A->Z |
| LVS\_DESCENDING | 指定项目按字母降序排列Z->A |

可以使用CListCtrl::GetStringWidth来获得由字符宽度所决定的列宽度，这里的字符是控件字体下的字符，并以此列宽度为基准将文本字符串用像素来计数。

#### 修改表现样式

在PreCreateWindow中使用的是默认表现样式，但是通过修改表现模式可以自由的选择显示模式。例如选择小图标模式：

ModifyStyle(LVS\_TYPEMASK, LVS\_SMALLICON);

ModifyStyle是一个CWnd函数，第1个参数指定某样式位关闭，第2个参数指定某样式位打开。LVS\_TYPEMASK可以屏蔽所有4个表现样式。

LVS\_ICON、LVS\_SMALLICON、LVS\_LIST、LVS\_REPORT不是真正的位标志，因此可以使用LVS\_TYPEMASK来查询列表视图当前的表现样式：

DWORD dwStyle = GetStyle() & LVS\_TYPEMASK;

if(dwStyle == LVS\_ICON)

……

#### 在列表视图中排序

在一个不带LVS\_NOCOLUMNHEADER样式的列表视图中选择了报表模式时，它会自动显示一个头标控件，即带有形似按钮而作为每列标题的“头标项”。除非列表视图具有样式LVS\_NOSORTHEADER，否则单击头标项会给列表视图的父亲发送一个LVN\_COLUMNCLICK通知。消息的lParam指向NM\_LISTVIEW结构，该结构的iSubItem字段保存从0开始编号的索引号，用来标识被单击的列。

应用程序通常对LVN\_COLUMNCLICK通知的响应是调用CListCtrl::SortItems来给列表视图项目排序。但有时需要提供一个回调函数，自定义排序。回调比较函数接收3个参数：两个被比较项目的32位lParam值和一个应用程序定义的lParam值，该值等于传递给SortItems的第2个参数。

可以在InsertItem的调用中或在单独对CListCtrl::SetItemData的调用中赋给项目的lParam值。除非应用程序保存了每个项目数据的私有副本，并在lParam中存放了一个值允许项目的数据被检索，否则比较函数得不到执行。虽然应用程序给每个项目分配内存并将指针填入项目的lParam中并不困难，但是由于事后必须释放内存，所以把事情搞复杂了。而且如果一个应用程序将文本字符串赋给列表视图的项目和子项目，而又要保存自己的项目数据，那么在内存使用方面它的效率就不高了，因为数据最终会被保存两次。要避免这种浪费，可以对项目和子项目文本指定LPSTR\_TEXTCALLBACK，并且在响应LVN\_GETDISPINFO通知时给列表视图控件提供文本。

//// .h

typedef struct tagITEMINFO

{

CString strFileName;

DWORD nFileSizeLow;

FILETIME ftLastWriteTime;

} ITEMINFO;

//// .cpp

ON\_WM\_DESTROY()

ON\_NOTIFY\_REFLECT(LVN\_GETDISPINFO, OnGetDispInfo)

ON\_NOTIFY\_REFLECT(LVN\_COLUMNCLICK, OnColumnClick)

void CFileView::OnDestroy()

{

FreeItemMemory();

CListView::OnDestroy();

}

void CFileView::FreeItemMemory()

{

int nCount = GetListCtrl().GetItemCount();

if (nCount)

{

for (int i = 0; i < nCount; i++)

{

delete (ITEMINFO\*)GetListCtrl().GetItemData(i);

}

}

}

BOOL CFileView::AddItem(int nIndex, WIN32\_FIND\_DATA\* pfd)

{

// Allocate a new ITEMINFO structure and initialize it with information about the item

ITEMINFO\* pItem;

try

{

pItem = new ITEMINFO;

}

catch (CMemoryException\* e)

{

e->Delete();

return FALSE;

}

pItem->strFileName = pfd->cFileName;

pItem->nFileSizeLow = pfd->nFileSizeLow;

pItem->ftLastWriteTime = pfd->ftLastWriteTime;

// Add the item to the list view

LV\_ITEM lvi;

lvi.mask = LVIF\_TEXT | LVIF\_IMAGE | LVIF\_PARAM;

lvi.iItem = nIndex;

lvi.iSubItem = 0;

lvi.iImage = 0;

lvi.pszText = LPSTR\_TEXTCALLBACK;

lvi.lParam = (LPARAM)pItem;

if (GetListCtrl().InsertItem(&lvi) == -1)

{

return FALSE;

}

return TRUE;

}

void CFileView::OnGetDispInfo(NMHDR\* pNMHDR, LRESULT\* pResult)

{

CString string;

LV\_DISPINFO\* pDispInfo = (LV\_DISPINFO\*)pNMHDR;

if (pDispInfo->item.mask & LVIF\_TEXT)

{

ITEMINFO\* pItem = (ITEMINFO\*)pDispInfo->item.lParam;

switch (pDispInfo->item.iSubItem)

{

case 0: // File Name

{

::lstrcpy(pDispInfo->item.pszText, pItem->strFileName);

break;

}

case 1: // File Size

{

string.Format(\_T("%u"), pItem->nFileSizeLow);

::lstrcpy(pDispInfo->item.pszText, string);

break;

}

......

}

}

\*pResult = 0;

}

void CFileView::OnColumnClick(NMHDR\* pNMHDR, LRESULT\* pResult)

{

NM\_LISTVIEW\* pNMListView = (NM\_LISTVIEW\*)pNMHDR;

GetListCtrl().SortItems(CompareFunc, pNMListView->iSubItem);

\*pResult = 0;

}

int CALLBACK CFileView::CompareFunc(LPARAM lParam1, LPARAM lParam2, LPARAM lParamSort)

{

ITEMINFO\* pItem1 = (ITEMINFO\*)lParam1;

ITEMINFO\* pItem2 = (ITEMINFO\*)lParam2;

int nResult;

switch (lParamSort)

{

case 0: // File Name

nResult = pItem1->strFileName.CompareNoCase(pItem2->strFileName);

break;

case 1: // File Size

nResult = pItem1->nFileSizeLow - pItem2->nFileSizeLow;

break;

case 2: // Date and time

nResult = ::CompareFileTime(&pItem1->ftLastWriteTime, &pItem2->ftLastWriteTime);

break;

default:

break;

}

return nResult;

}

#### 列表视图的命中测试

可以使用CListCtrl::HitTest对列表视图中的项目进行命中测试。给定一个点的坐标，HitTest返回该点处项目的索引号，或者该点与项目不相对应时返回-1。例如：

ON\_NOTIFY\_REFLECT(NM\_DBLCLK, OnDoubleClick)

Void CMyListView::OnDoubleClick(NMHDR\* pNMHDR, LRESULT\* pResult)

{

DWORD dwPos = ::GetMessagPos();

CPoint point((int)LOWORD(dwPos), (int)HIWORD(dwPos));

GetListCtrl().ScreenToClient(&point);

Int nIndex;

If((nIndex = GetListCtrl().HitTest(point)) != -1) { … … }

}

NM\_DBLCLK通知中并不包含鼠标指针的坐标，因此可以用::GetMessagePos来检索指针位置。由::GetMessagePos返回的屏幕坐标被转换为列表视图中的客户区坐标并传递给了CListCtrl::HitTest。

### 自制控件视图

使用CCtrlView做基类，可以创建封装了其他Windows控件的控件视图。

例如CCtrlView派生类定义了一个标签视图，它是一种封装了Win32标签控件的简单视图。该视图被显示时，看上去与普通视图相似，就是在顶部有类似于属性页上的标签。

//// .h

class CTabView : public CCtrlView

{

DECLARE\_DYNCREATE(CTabView)

public:

CTabView() :

CCtrlView(\_T("SysTabControl32"), AFX\_WS\_DEFAULT\_VIEW)

{}

CTabCtrl& GetTabCtrl() const {

return \*(CTabCtrl\*)this;

}

virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

virtual void OnInitialUpdate();

};

//// .cpp

IMPLEMENT\_DYNCREATE(CTabView, CCtrlView)

BOOL CTabView::PreCreateWindow(CREATESTRUCT& cs)

{

::InitCommonControls();

if (!CCtrlView::PreCreateWindow(cs))

{

return FALSE;

}

cs.style |= TCS\_FIXEDWIDTH; // Fixed-width tabs

return TRUE;

}

void CTabView::OnInitialUpdate()

{

static CString strLabel[] = {\_T("Tab No.1"), \_T("Tab No.2"), \_T("Tab No.3")};

// Set the tab width to 96 pixels

GetTabCtrl().SetItemSize(CSize(96, 0));

// Add three tabs

TC\_ITEM item;

item.mask = TCIF\_TEXT;

for (int i = 0; i < 3; i++)

{

item.pszText = (LPTSTR)(LPCTSTR)strLabel[i];

item.cchTextMax = strLabel[i].GetLength();

GetTabCtrl().InsertItem(i, &item);

}

}

## 多文档和多视图

MDI和SDI应用程序结构上的主要区别：

* MDI应用程序从CMDIFrameWnd而不是CFrameWnd派生它们的顶层框架窗口类。
* MDI应用程序使用基于CMDIChildWnd的类来代表包含文档视图的子框架窗口。
* MDI应用程序使用基于CMultiDocTemplate而不是CSingleDocTemplate来创建文档模板。在CMultiDocTemplate的构造函数中引用的框架窗口类是子框架窗口类而不是顶层框架窗口类。
* MDI应用程序至少具有两个菜单资源，而SDI只有一个。其中一个在没有文档打开时显示，另一个在至少有一个文档打开时显示。

### 同步文档的多个视图

当从Windows菜单选择了New Windows命令后，MFC提供的命令处理程序会调出文档模板，提取标识视图类和框架窗口类的CRunTimeClass指针，并用它实例化一个新视图和新的框架窗口类(自框架，而非顶层框架)。实际上，第二个视图的地址被添加在了由文档对象维护的视图链接列表中，以便文档可以明了有它的两个独立的视图出现在了屏幕上。如果任一个视图要求重绘，都要调用GetDocument来获取指向文档对象的指针，向文档查询需要的数据进行重绘。

主结构以CDocument::UpdateAllViews和CView::OnUpdate的形式提供了在一个视图中编辑了文档，其他视图也要被更新的机制。UpdateAllViews迭代处理与文档关联的视图列表，调用每个视图的虚拟OnUpdate函数。

CView提供了一个很小的OnUpdate实现，它使视图无效并促使调用OnDraw。如果希望整体重绘，则没有必要覆盖OnUpdate。如果希望使更新操作尽可能提高效率，只重绘视图中修改过的部分，就要在视图类中覆盖OnUpdate并使用传递给UpdateAllViews的提示信息。

UpdateAllView的原型如下：

void UpdateAllViews(CView\* pSender, LPARAM lHint = 0L, CObject\* pHint = NULL);

OnUpdate的原型如下：

void OnUpdate (CView\* pSender, LPARAM lHint, CObject\* pHint);

lHint和pHint将提示信息从UpdateAllViews带到OnUpdate。如何使用这些参数取决于应用程序。提示信息的一个简单用途是传递RECT结构的地址或是指定视图中需要更新部分的CRect对象的指针。OnUpdate可以在调用InvalidateRect中使用此信息。

// In the document views

UpdateAllViews(NULL, 1, (CObject\*)pRect);

// In the view class

Void CMyView::OnUpdate(CView\* pSender, LPARAM lHint, CObject\* pHint)

{

If(lHint == 1)

{

CRect\* pRect = (CRect\*)pHint;

InvalidateRect(pRect);

return;

}

CView::OnUpdate(pSender, lHint, pHint);

}

MFC有时自己会用lHint等于0来调用OnUpdate，可以对lHint使用任意的非零值，甚至可以定义多个提示集，给pHint赋予不同的含义并使用lHint来标识提示类型。

UpdateAllViews的第一个参数pSender指定了从更新循环操作中忽略的一个视图。

### 支持多个文档类型

MDI应用程序除了可以支持多个文档实例外，还可以支持多个文档类型，每一个的特性都由文档模板来定义。具体实现方法：

* 派生一个新文档类和一个新视图类作为新文档类型。
* 为新文档在项目中添加4个新资源：图标、菜单、加速键(可选)和文本字符串，赋给4个资源相同的资源ID。
* 修改InitInstance来创建新的文档模板，其中包含资源ID和为文档、视图以及框架窗口类设置的CRuntimeClass指针。然后调用AddDocTemplate并传递文档模板对象的指针。

### MDI之外的其他选择

MDI程序设计模型以外的3种可选模型：

基于工作区的模型，将相关的文档组合在称为”工作区”的对象中，并允许包含在工作区中的文档可以在类似MDI文档窗口中被查看和编辑。

工作手册模型，每个视图都占据顶层框架窗口的整个客户区，但任何时候只有一个视图可见。外观与MDI应用程序中的最大化文档框架类似。每个视图都做了一个标签，用户可以通过点击按钮而在视图间切换，视图好像是属性页中的页面一样。

项目模型，将相关的文档组合在一个项目中，但只允许各个文档在类似SDI的框架窗口中编辑。项目模型、MDI和工作区模型三者之间的主要区别在于，在项目模型中没有提供顶层框架窗口来包含文档框架。

## 拆分窗口

对于SDI应用程序，呈现一个文档具有两个以上同步视图的最好方法是使用基于MFC CSplitterWnd类的拆分窗口。拆分窗口的每个窗格都包含文档数据的一个视图。视图是拆分窗口的子窗口，通常拆分窗口又是框架窗口(SDI的顶层框架窗口、MDI的文档框架窗口)的子窗口。位于拆分窗口内的视图使用CView::GetParentFrame来获得指向其父框架窗口的指针。

MFC支持两种类型的拆分窗口：静态的和动态的。

静态拆分窗口的行列数在拆分窗口被创建时就设置好了，用户不能修改，但可以缩放各行各列。最多可以包含16行16列。

动态拆分窗口的最大行列数是在创建拆分窗口时指定的，最多可以有2行2列，但它们可以相互拆分和合并。动态拆分窗口中的视图并不是彼此完全独立的：水平方向拆分的两行窗口具有各自独立的垂直滚动条但公用一个水平滚动条；垂直方向拆分的两列窗口具有各自独立的水平滚动条但公用一个垂直滚动条。

选择静态或动态拆分窗口的一个准则是是否希望用户能够交互式地修改拆分窗口的行列配置，如果希望就选用动态拆分窗口；另外一个准则是计划在拆分窗口中使用的视图种类，静态拆分窗口可以使用两个以上不同种类的视图，动态拆分窗口除非从CSplitterWnd派生一个新类并修改拆分窗口的默认操作功能，否则所有视图使用的都是相同的视图类。

### 动态拆分窗口

动态拆分窗口使用MFC的CSplitterWnd::Create函数创建，步骤为：

* 给框架窗口类添加一个CSplitterWnd数据成员；
* 覆盖框架窗口的虚拟OnCreateClient函数，并调用CSplitterWnd::Create在框架窗口的客户区创建动态拆分窗口；

BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext\* pContext)

{

return m\_wndSplitter.Create(this, 2, 1, CSize(1,1), pContext);

}

第一个参数标识了拆分窗口的父窗口，它是框架窗口。第二个和第三个参数指定了窗口能够拆分的最大行列数。第四个参数以像素为单位指定了每个窗格的最小宽度和最小高度，主结构使用这些值来确定在拆分条移动过程中何时创建和销毁窗格。第五个参数是指向主结构的CCreateContext结构的指针，结构中m\_pNewViewClass成员标识了用来创建拆分窗格中视图的视图类。主结构创建一个最初的视图并把它应用到第一个窗格中。其他相同类的视图在别的窗格被生成时会自动地创建。

Create还支持第六个和第七个参数，分别指定拆分窗口的样式和其子窗口ID。默认的子窗口ID为AFX\_IDW\_PANE\_FIRST，用它可以使框架窗口标识与其关联的拆分窗口。只有在已经包含拆分窗口的框架窗口中创建第二个拆分窗口时才需要修改此ID。

CSplitterWnd提供一些成员函数，用来向拆分窗口查询一些相关信息：查询当前显示窗口的行列数、查询一行或一列的高度或宽度、查询指向特定行列窗格中视图的CView指针。

如果想在应用程序的菜单中添加Split命令，需要一个ID为ID\_WINDOW\_SPLIT的菜单项。此ID已经预置在CView消息映射表中的CView::OnSplitCmd和CView::OnUpdateSplitCmd中。在内部，CView::OnSplitCmd会调用CSplitterWnd::DoKeyboardSplit开始跟踪处理，并允许使用箭头键上下移动拆分条。在Enter键按下时跟踪结束，接受新的拆分位置，在ESC键按下时取消操作。

### 静态拆分窗口

静态拆分窗口使用CSplitterWnd::CreateStatic创建，函数返回后创建视图。步骤为：

给框架窗口添加一个CSplitterWnd数据成员；

覆盖框架窗口的OnCreateClient函数，并调用CSplitterWnd::CreateStatic创建静态拆分窗口；

使用CSplitterWnd::CreateView在每个静态拆分窗口的窗格中创建视图；

BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext\* pContext)

{

if(!m\_wndSplitter.CreateStatic(this, 1, 2) ||

!m\_wndSplitter.CreateView(0, 0, RUNTIME\_CLASS(CTextView), CSize(128, 0), pContext) ||

!m\_wndSplitter.CreateView(0, 1, RUNTIME\_CLASS(CPictureView), CSize(0, 0), pContext) ||)

{

return FALSE;

}

return TRUE;

}

传递给CreateStatic的参数指定了拆分窗口的父窗口以及拆分窗口包含的行列数。对每个窗格调用一次CreateView。视图并不是直接实例化得到的，而是由MFC创建的。所以，要将指向CreateView的指针而不是指向已存在的CView对象的指针传递给CRuntimeClass。

传递给CreateView的CSize对象指定了窗格的初始尺寸。本例中CTextView初始宽度为128像素，CPictureView将占据剩余的窗口宽度。指定宽度或高度为0， 主结构会忽略它们。

### 自定义命令传送

默认的命令传送机制不能满足要求时，可以自定义命令传送路径。只要覆盖合适的OnCmdMsg函数，就可以在命令传送顺序中的任何地方进行处理。一般情况下，应该从覆盖函数中调用基类的OnCmdMsg以确保默认命令传送继续有效。要小心处理所调用的OnCmdMsg函数，如果调用不当可能会陷入递归循环。

### 嵌套拆分窗口

#### 嵌套静态拆分窗口

BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext\* pContext)

{

if (!m\_wndSplitter1.CreateStatic(this, 1, 2) ||

!m\_wndSplitter1.CreateView(0, 0, RUNTIME\_CLASS(CTextView), CSize(128, 0), pContext) ||

!m\_wndSplitter2.CreateStatic(&m\_wndSplitter1, 2, 1, WS\_CHILD | WS\_VISIBLE, m\_wndSplitter1.IdFromRowCol(0, 1)) ||

!m\_wndSplitter2.CreateView(0, 0, RUNTIME\_CLASS(CPictureView), CSize(0, 128), pContext) ||

!m\_wndSplitter2.CreateView(1, 0, RUNTIME\_CLASS(CPictureView), CSize(0, 0), pContext))

return FALSE;

return TRUE;

}

* 第一个拆分窗口是根据m\_wndSplitter1调用CreateStatic而创建的，包含一行两列；
* 使用CreateView将CTextView添加到m\_wndSplitter1的第一个窗格中；
* 通过调用m\_wndSplitter2的CreateStatic函数在第一个拆分窗口的第二个窗格中创建第二个拆分窗口。m\_wndSplitter2的父窗口是m\_wndSplitter1而不是框架窗口，并且它得到一个子窗口ID，将其标识为第0行第1列的窗格。调用CSplitterWnd::IdFromRowCol可以得到适当的ID，该函数使用简单的换算将行列编号转换为添加给AFX\_IDW\_PANE\_FIRST的数字偏移量；
* 调用CreateView两次给m\_wndSplitter2窗格添加CPictureView；

#### 嵌套动态拆分窗口

由于MFC对给动态创建的拆分窗口窗格生成要填入的新视图做了一些假定，所以对m\_wndSplitter2使用动态拆分窗口稍微复杂一些。

如果使用以下代码嵌套动态拆分视图，可能会发生访问错误：

BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext\* pContext)

{

if (!m\_wndSplitter1.CreateStatic(this, 1, 2) ||

!m\_wndSplitter1.CreateView(0,0,RUNTIME\_CLASS(CTextView),

CSize(128, 0), pContext) ||

!m\_wndSplitter2.Create(&m\_wndSplitter1, 2, 1, CSize(1,1), pContext, WS\_CHILD | WS\_VISIBLE | WS\_HSCROLL | WS\_VSCROLL | SPLS\_DYNAMIC\_SPLIT, m\_wndSplitter1.IdFromRowCol(0, 1)))

return FALSE;

return TRUE;

}

错误原因根植于主结构中。当动态拆分窗口拆分时，CSplitterWnd用一个NULL pContext指针来调用CreateView为新窗格创建一个新视图。由于pContext为NULL值，CreateView就要向框架窗口查询指向活动视图的指针，并使用该视图作为新视图的模型。如果碰巧在拆分时CTextView窗口是活动视图，主结构看到该视图不是动态拆分窗口的子视图就会创建一个与文档对象无关的“空”视图。这样在视图第一次访问其文档时，访问错误就会发生。

成功地在静态拆分窗口中嵌套动态拆分窗口的秘密：

* 从CSplitterWnd中派生类并用以下代码替换派生类中的CSplitterWnd::SplitRow

BOOL CNestedSplitterWnd::SplitRow(int cyBefore)

{

GetParentFrame()->SetActiveView((CView\*)GetPane(0, 0));

return CSplitterWnd::SplitRow(cyBefore);

}

* 使要嵌套的动态拆分窗口成为派生类的实例而不是CSplitterWnd的实例；

拖动水平拆分条创建新窗格时就会调用SplitRow虚拟函数。以上实现使得动态拆分窗口内最顶层窗格中的视图在拆分进行之前成为了活动视图，就解决了由于活动视图是静态拆分窗口的子视图而造成的动态视图创建中的问题。

覆盖函数使用GetParentFrame而不是GetParent是因为动态拆分窗口的父亲实际上是静态拆分窗口而不是框架窗口，而且是框架窗口函数设置活动视图的。

### 带有多种视图类型的动态拆分窗口

从CSplitterWnd中派生一个类，并覆盖CreateView，然后用指向所选视图的CRuntimeClass指针调用CSplitterWnd::CreateView。可以创建动态拆分窗口，在它的不同窗格中显示不同类型的视图。

BOOL CDynaSplitterWnd::CreateView(int row, int col, CRuntimeClass\* pViewClass, SIZE sizeHint, CCreateContext\* pContext)

{

if((row == 1) && (col == 0))

{

return CSplitterWnd::CreateView(row, col, RUNTIME\_CLASS(CTextView), sizeHint, pContext);

}

return CSplitterWnd::CreateView(row, col, pViewClass, sizeHint, pContext);

}

覆盖后的函数不管第0行第0列的视图类型是什么，都将使第1行第0列的窗格具有CTextView视图。