**1.1 MFC 文档视图结构程序结构总揽**

当我们使用 MFC AppWizard 生成一个 MFC 程序，选用所有默认的设置（当然也是 Multiple Documents ，本文讨论主要基于 Multiple Documents ，对于 Single Document 情况仅以简单表述提及，皆因后者和前者很多相似相同之处，但前者更为复杂，并且更加常用。），假设你的程序名称为 A ，则你会得到 CMainFrame 、 CChildFrame 、 CAboutDlg 、 CADoc 、 CAView 、 CAApp 6 个类（ Single Document 仅少一个 CChildFrame 类，其余均同）。这些类的具体含义将在后面给出，这里先要给出一个 MFC 支持文档视图结构程序（以下简称 App ）的主要组成：

一个 App （对应类 CAApp ）可以包含多个文档模版（ CDocTemplate ），但是 MFC AppWizard （无论是 SDI 还是 MDI ）都只是默认生成一个。但是在实际开发中一个文档模版不够，需要自己手工添加（在后面实际项目例子提供示例）。这个结构是通过 MFC 中 CWinApp 的成员变量 CDocManager\* m\_pDocManager 实现的，我们的 CAApp 正是继承自 MFC 提供的 CWinApp 类。

CDocManager 类则拥有一个指针链表 CPtrList m\_templateList 来维护这些文档模版。这些文档模版都是在 CAApp ：： InitInstance （）中通过 AddDocTemplate(pDocTemplate) 。

CDocTemplate 拥有 3 个成员变量，分别保存着 Document 、 View 、 Frame 的 CRuntimeClass 指针，另外持有成员变量 m\_nIDResource ，用来指定这个 Document 显示时候采用的菜单资源。这 4 份数据都在 CAApp ：： InitInstance （）中 CDocTemplate 的构造函数中指定。在 Document 中拥有一个回指 CDocTemplate 的指针（ m\_pDocTemplate ）。

一个文档可以有多个视图，由 Document 中成员变量 CPtrList m\_ViewList 维护。

CFrameWnd 拥有一个成员变量 CView\* m\_pActiveView 指向当前活动的 View 。

CView 中拥有一个成员变量 CDocument\* m\_pDocument 指向该视图相关的 Document 。

[ 注解 ] ： ① MFC SDI/MDI 程序默认都默认生成了一个文档模版，并将这个文档模版 Add 到其文档模版的链表中，由于这是 MFC 默认提供的，因此这个文档模版会被插入到文档模版的第一个位置，而 MFC 也是通过这个文档模版的特定位置可以确定的。默认情况下，当我们点击 File （ Open ） / File （ New ）的时候，这个文档模版会被启用。

除了侯捷先生在《深入浅出 MFC 中列出的以上的深入分析，我们还应该（很大程度上更加重要）掌握以下的关于 MFC SDI/MDI 的知识：

**文档的本质**：文档是用来保存数据以及关于数据的处理的，每当 MFC SDI/MDI 响应 File （ Open ） / File （ New ）的时候都会打开一份文档。文档可以拥有多个视图。文档和视图的关系可以这样理解：文档是被视图观察的对象。

**视图本质**：视图在 Windows 中就是一个窗口，也就是一个可视化的矩形区域。视图是用来表示文档的数据的。但是每个视图必需依附于一个框架（ SDI 中是 MainFrame ， MDI 是 ChildFrame ）。当然你可以自己去 Create 一个视图，并且去显示它。

**框架的本质**：框架实际是也是一个 Windows 窗口。但是在框架上可以放置菜单、工具栏、状态栏等。而视图则放在框架的客户区。因此 MFC 中我们看到的窗口实际上 Frame 和 View 共同作用的结果。

在某一时刻，程序中只有一个活动的文档、框架和视图，即当前的文档、框架、视图。

**1.2 MFC SDI/MDI 各个类之间的互访**

在实际项目开发中用的最多就是各个类之间的互访问，这里将网络上和书籍中提到的做了一个总结，也是笔者在实际开发中都用到过的。

|  |  |  |
| --- | --- | --- |
| 访问对象 | 访问位置 | 访问实现 |
| 应用程序 App | 任何位置 | ① AfxGetApp();  ② 在要使用应用程序 App 的文件中加入：  extern CAApp theApp ，然后直接使用全局的 theApp 变量。 |
| 主框架窗口 | 任何位置 | ① AfxGetMainWnd();  ② AfxGetApp()->m\_pMainWnd; |
| 视图 | 框架类中 | GetActiveView();// 当前的活动视图 |
| 文档类中 | GetFirstViewPosition （）； // 可以获取全部视图  GetNextView （）； |
| 文档 | 文档类中 | GetDocument() ； |
| 文当模版类中 | GetFirstDocPosition(); // 该文档模版对应全部文档  GetNextDoc(); |
| 框架类中 | GetActiveDocument(); // 当前活动文当 |
| 子框架类（MDI中） | 主框架类中 | ① MDIGetActive （）；  ② GetActiveFrame （）； |
| 视图类中 | GetParentFrame(); |
| 文档模版 | 文档类中 | GetDocTemplate(); |
| 应用程序 App 中 | GetFirstDocTemplatePosition();  GetNextDocTemplate(); |

说明： 1 ）以上给出的都是方法，实际访问中可能还要进行以下简单的处理，如类型转换，循环遍历等；

            2 ）可能没有列举完所有可能位置的互访问，但可以通过他们的组合得到。

**2 文档、视图、框架之间的关联**

MFC SDI/MDI 中的核心就在于文档、视图、框架之间的关联，形成了一个有机的可运作的整体。 MFC 提供了默认的关联关系，但是在实际的项目开发中很多时候需要动态进行他们的之间的关联。

**2.1 文档和视图间的关联**

使用 MFC AppWizard 声称 MFC SDI/MDI 程序，在 App 类的 InitInstance （）方法中有如下代码（假设 Project 名称均为 Test ）：

SDI 中

|  |
| --- |
| CSingleDocTemplate\* pDocTemplate;  pDocTemplate = new CSingleDocTemplate(  IDR\_MAINFRAME,  RUNTIME\_CLASS(CTestDoc),  RUNTIME\_CLASS(CMainFrame),// main SDI frame window  RUNTIME\_CLASS(CTestView));  AddDocTemplate(pDocTemplate); |

MDI 中

|  |
| --- |
| CMultiDocTemplate\* pDocTemplate;  pDocTemplate = new CMultiDocTemplate(  IDR\_TESTTYPE,  RUNTIME\_CLASS(CTestDoc),  RUNTIME\_CLASS(CChildFrame), // custom MDI child frame  RUNTIME\_CLASS(CTestView));  AddDocTemplate(pDocTemplate); |

这里通过 CDocTemplate （无论是 SDI 中的 CSingleDocTemplate 还是 MDI 中的 CMultiDocTemplate ）的构造函数，将文当、视图和框架（ SDI 中与主框架， MDI 中与自框架）关联在一起了，形成了一个整体。

**手工实现文档和视图的关联**

在实际的项目开发时候仅仅依靠 MFC AppWizard 生成的文当和视图、框架是不够的，因此我们需要掌握手工进行这种关联。手工进行文当和视图的关联可以有以下两种实现方式：

模仿 MFC AppWizard 实现，使用 CDocTemplate 的构造函数：在上面的分析中我们可以看到通过 CDocTemplate （无论是 SDI 中的 CSingleDocTemplate 还是 MDI 中的 CMultiDocTemplate ）的构造函数我们可以获得文档、视图和框架的关联。因此可以通过模拟这种方式进行关联。具体实现方法如下：

1 ） 创建新的文档、视图和框架类，方法是使用 VC 中的 Insert MFC Class

实现。注意到框架类选择 CMDIChildWnd 作为基类，文档类选择 CDocument 作为基类，而视图类则可以根据需要选择 CView 或其子类（ CEditView ）等作为基类。

2 ） 为该框架添加菜单资源，方法是在 VC 资源窗口 Menu 菜单下添加新的菜

单，当然可以通过复制 VC 提供默认菜单进行修改。

3 ） 在 App 类的 InitInstance （）中添加如下类似代码：

|  |
| --- |
| CMultiDocTemplate\* m\_pDocTemplate;  m\_pDocTemplate = new CMultiDocTemplate(  IDR\_TESTTYPE,// 改为你新建的菜单资源 ID  RUNTIME\_CLASS(CTestDoc),// 改为你新建的文档类  RUNTIME\_CLASS(CChildFrame), // 改为你新建的框架类  RUNTIME\_CLASS(CTestView));// 改为你新建的视图类  AddDocTemplate(m\_pDocTemplate); |

4 ） 为了记录这个文档模版，你可以在 App 类中添加一个 CMultiDocTemplate\*类型变量来维持这个文档模版。

上面给出了通过 CDocTemplate 的构造函数将文档、视图、和框架关联起来，但是有时候我们并不想创建一个新的文档模版，我们只是想给同一个数据提供不同的结果显示，或者说是为同一个文档添加一个新的视图，并提供他们之间的一个切换。还有一种可能就是我们本来不是一个文档视图结构支持的程序，想为视图添加一个文档，更好进行业务逻辑和表示层的一个分离。第一种方法的实现方法：

Step 1 ：使用 VC 6.0 新建一个 Project ，命名为： MultiView 。除选择单文档属性外，一切使用“默认”方式。于是你可以获得五个类： CMainFrame ， CMultiViewApp ， CMultiViewDoc ，CMultiViewView ，和 CAboutDlg ；

Step 2 ：新建一个新的视图 View ，添加一个新的 MFC Class （ Insert － >New Class ），基类为CView （或者 CView 的派生子类，如 CEditView 等）。类的名字为 CAnotherView ，这就是新的视图；并为 CAnotherView 添加 GetDocument 的实现：

|  |
| --- |
| CMultiViewDoc\* CAnotherView::GetDocument()  {  return (CMultiViewDoc\*)m\_pDocument;  } |

Step 3 ：在 CMultiViewApp 添加成员变量记录这两个视图：

|  |
| --- |
| private:  CView\* m\_pFirstView;  CView\* m\_pAnotherView; |

给程序菜单 IDR\_MAINFRAME 添加一个菜单项目“视图”，该菜单项有两个子菜单“视图一”和“视图二”，添加相应函数（ void CMultiViewApp :: OnShowFirstview （）和 void CMultiViewApp ::OnShowSecondview （））；

Step 4 ：创建新的视图：在 BOOL CMultiViewApp :: InitInstance () 中添加代码：

|  |
| --- |
| …….  // 创建一个新的视图  CView\* m\_pActiveView = ((CFrameWnd\*)m\_pMainWnd)->GetActiveView();  m\_pFirstView = m\_pActiveView;  m\_pAnotherView = new CAnotherView();  // 文档和视图关联  CDocument\* m\_pDoc = ((CFrameWnd\*)m\_pMainWnd)->GetActiveDocument();  CCreateContext context;  context.m\_pCurrentDoc = m\_pDoc;  // 创建视图  UINT m\_IDFORANOTHERVIEW = AFX\_IDW\_PANE\_FIRST + 1;  CRect rect;  m\_pAnotherView->Create(NULL,NULL,WS\_CHILD,rect,m\_pMainWnd,  m\_IDFORANOTHERVIEW,&context);      …… |

Step 5 ：现在已经创建了视图，并且都和文档关联起来了。现在要作的就是视图间的转换。在void CMultiViewApp :: OnShowFirstview （）中添加实现代码：

|  |
| --- |
| void CMultiViewApp::OnShowFirstview()  {  // TODO: Add your command handler code here  UINT temp = ::GetWindowLong(m\_pAnotherView->m\_hWnd, GWL\_ID);      ::SetWindowLong(m\_pAnotherView->m\_hWnd, GWL\_ID, ::GetWindowLong(m\_pFirstView->m\_hWnd, GWL\_ID));      ::SetWindowLong(m\_pFirstView->m\_hWnd, GWL\_ID, temp);  m\_pAnotherView->ShowWindow(SW\_HIDE);  m\_pFirstView->ShowWindow(SW\_SHOW);  ((CFrameWnd\*)m\_pMainWnd)->SetActiveView(m\_pFirstView);  ((CFrameWnd\*) m\_pMainWnd)->RecalcLayout();      m\_pFirstView->Invalidate();  } |

在 void CMultiViewApp :: OnShowSecondview （）中添加实现代码：

|  |
| --- |
| void CMultiViewApp::OnShowSecondview()  {  // TODO: Add your command handler code here  UINT temp = ::GetWindowLong(m\_pAnotherView->m\_hWnd, GWL\_ID);      ::SetWindowLong(m\_pAnotherView->m\_hWnd, GWL\_ID, ::GetWindowLong(m\_pFirstView->m\_hWnd, GWL\_ID));      ::SetWindowLong(m\_pFirstView->m\_hWnd, GWL\_ID, temp);  m\_pFirstView->ShowWindow(SW\_HIDE);  m\_pAnotherView->ShowWindow(SW\_SHOW);  ((CFrameWnd\*)m\_pMainWnd)->SetActiveView(m\_pAnotherView);  ((CFrameWnd\*) m\_pMainWnd)->RecalcLayout();      m\_pAnotherView->Invalidate();  } |

Step 6 ：为了演示，这里将不同的视图给予一个标记，在 CMultiViewView 和 CAnotherView 的OnDraw 方法中分别添加以下代码：

|  |
| --- |
| pDC->TextOut(400,300,"First View");  pDC->TextOut(400,320,pDoc->GetTitle()); |

和

|  |
| --- |
| pDC->TextOut(400,300,"Another View");  pDC->TextOut(400,320,pDoc->GetTitle()); |

至此就大功告成了，但是实现过程中有 几 点说明：

1）   实现中由于使用到相关的类，因此在必要的地方要 include 相关的头文件，这里省略；CAnotherView 的默认构造函数是 Protected 的，需要将其改为 Public ，或者提供一个产生CAnotherView 对象的方法（因要创建视图对象）；

2）   这里给出的是一个示例代码，实际开发中可以通过参考实现获得自己想要实现的具体应用情况（例如视图类的不同、数量不同，更重要的还有业务逻辑的不同实现等）；

第二种视图和文档关联的方法：我们使用 CCreateContext 类进行他们之间的关联，具体实现为：

|  |
| --- |
| m\_pAnotherView = new CAnotherView(); //new 一个新的视图，可以改为你新建的视图  // 获取一个已有的文档，可以是你新建的文档  CDocument\* m\_pDoc = ((CFrameWnd\*)m\_pMainWnd)->GetActiveDocument();  // 文档和视图关联  CCreateContext context;  context.m\_pCurrentDoc = m\_pDoc;  // 创建视图  UINT m\_IDFORANOTHERVIEW = AFX\_IDW\_PANE\_FIRST + 1; // 创建视图的 ID 号，你可以自己设置  CRect rect;  m\_pAnotherView->Create(NULL, NULL, WS\_CHILD, rect, m\_pMainWnd, m\_IDFORANOTHERVIEW, &context); |

在框架和视图关联的时候进行设置，具体见一下框架和视图关联部分。

**2.2 框架和视图的关联**

在第一部分分析我们知道，框架和视图其实都是 windows 窗口，不过框架提供了菜单、标题栏、状态栏等资源，而视图则只是一个矩形区域。 MFC 程序中视图决定大多数时候要依附于一个框架（ SDI 中的 MainFrame 和 MDI 中的子框架窗口），可以这样理解，框架相当于一个窗口容器（当然它本身也是一个 windows 窗口），而视图则正好是放置在框架内客户区域的内容。

框架和视图的关联也可以通过 模仿 MFC AppWizard 实现，使用 CDocTemplate 的构造函数实现，即和 2.1 中文档和视图间的关联方式相同，这里就不再给出，参看上面的详细实现即可。

同上面的分析，在很多的时候我们并不是需要提供一个新的文档模版，我们只是需要显示一个新的窗口（ MDI 程序），例如我们在作 MIS 系统界面管理的时候，经常出现的情况就是用户点击一个菜单选项，即弹出一个处理窗口。而要显示一个新的窗口，我们可以通过 CDocTemplate 的OpenDocumentFile （）方法打开一个文档实现，这样就建立一套的文档、视图和框架的体系。上面已经分析到，我们看到 MFC 的窗口实际上框架和视图的一个结合体，我们并不一定要提供文档、视图、框架的整个体系，我们只需要框架和视图的结合即可实现窗口的显示，这就要经过两个步骤实现：第一步将视图和框架关联，第二步显示框架（也就是一个 windows 窗口的显示）。以下给出框架和视图关联的具体实现：

|  |
| --- |
| CChildFrame \* pFrm = new CChildFrame();// 框架可以是你新建或者定制的框架类  CCreateContext context;  context.m\_pNewViewClass = RUNTIME\_CLASS(CDemoView);// 视图可以是你想显示的视图  pFrm->LoadFrame(IDR\_TEST2TYPE,WS\_CHILD  WS\_OVERLAPPEDWINDOW, this, &context);// 菜单资源你可以修改  pFrm->ShowWindow(SW\_SHOW);// 显示窗口  pFrm->InitialUpdateFrame(NULL,true);// 调用视图的 OnInitialUpdate （）和框架的 ActiveFrame （），你可以在这里设置窗口的标题 |

当然你可以在这里添加视图和文档的关联，具体实现是添加以下代码：

|  |
| --- |
| context.m\_pCurrentDoc = m\_pDoc;// m\_pDoc 就是你要关联到的文档对象 |

**3 代码实例**

本部分将以一个通用的 MDI 项目界面设计开发为例，将上面的分析附诸实践。

**3.1 情景描述**

一个常见界面逻辑为：用户打开一个系统，显示基本的菜单，用于用户的登录、注销和用户管理（当然这也可以通过一个用户登录的对话框实现）。用户正确登录后，显示系统的功能操作界面，当用户点击一个菜单项后（对应一个或者多个业务逻辑），弹出一个处理操作界面（并非对话框）。各个功能操作界面可共存于一个框架内，可以最大化、最小化或者关闭。

3.2 代码实现

下面就将上面的提到的情景用前面提到的技术，给出详细的实现方案。

Step 1 ：新建一个 MFC 项目，名称为 Demo ，选择不用 Document/View Architecture 支持（第二步去掉默认的复选框即可）。这样系统为你默认生成 5 个类： CDemoApp 、 CMainFrame 、 CChildFrame 、 CAboutDlg 、 CDemoView 。各个类的含义上面已经分析了，不罗嗦。另外系统还提供了两个默认的菜单： IDR\_DEMOTYPE 和 IDR\_MAINFRAME ，将 IDR\_MAINFRAME 的菜单的“文件”改名为“开始”（好像更加专业，不该也没有什么，本来就是 Demo ），然后将这个菜单项的子菜单中改为“登录”、“注销”，并使用默认的子项，并将前两者的 ID 号改为：“ ID\_LOGIN”和“ ID\_LOGOUT ”。拷贝（ Ctrl + C ）粘贴（ Ctrl + V ），则得到一个名称为 IDR\_MAINFRAME1 的菜单资源，删除原有的 IDR\_DEMOTYPE菜单（注一先保存名称），再将 IDR\_MAINFRAME1 的名称改为 IDR\_DEMOTYPE 。再为 IDR\_DEMOTYPE 添加一个菜单项“功能”，添加两个子菜单项“业务逻辑一”和“业务逻辑二”， ID 号分别为： ID\_FUNC\_ONE 和 ID\_FUNC\_TWO 。

Step 2 ：给 CDemoApp 添加两个变量，保存菜单资源：

|  |
| --- |
| HMENU m\_hOPMenu;  HMENU m\_hInitMenu; |

并在 BOOL CDemoApp::InitInstance()中添加代码：

|  |
| --- |
| m\_hInitMenu = ::LoadMenu(hInst, MAKEINTRESOURCE(IDR\_MAINFRAME));  m\_hOPMenu = ::LoadMenu(hInst, MAKEINTRESOURCE(IDR\_DEMOTYPE)); |

为了显示效果，在 BOOL CDemoApp::InitInstance()中添加代码：

|  |
| --- |
| pFrame->SetWindowText(" 未登录 ");  m\_nCmdShow = SW\_SHOWMAXIMIZED;  pFrame->ShowWindow(m\_nCmdShow);// 系统提供  pFrame->UpdateWindow(); |

为菜单项“登录”添加响应函数（使用 Class Wizard ， Class Name 选择 CDemoApp ）

|  |
| --- |
| void CDemoApp::OnLogin()  {  // TODO: Add your command handler code here  SetMenu(AfxGetApp()->m\_pMainWnd->m\_hWnd,m\_hOPMenu);  AfxGetApp()->m\_pMainWnd->SetWindowText(" 已登录 ");  } |

同上，为菜单项“注销”添加响应函数：

|  |
| --- |
| void CDemoApp::OnLogout()  {  // TODO: Add your command handler code here  if(((CMainFrame \*)AfxGetMainWnd())->m\_pFuncOneFrame != NULL)  ((CMainFrame \*)AfxGetMainWnd())->m\_pFuncOneFrame->SendMessage(WM\_CLOSE);  if(((CMainFrame \*)AfxGetMainWnd())->m\_pFuncTwoFrame != NULL)  ((CMainFrame \*)AfxGetMainWnd())->m\_pFuncTwoFrame->SendMessage(WM\_CLOSE);  SetMenu(AfxGetApp()->m\_pMainWnd->m\_hWnd,m\_hInitMenu);  AfxGetApp()->m\_pMainWnd->SetWindowText(" 未登录 ");  } |

这就实现了登录和注销的功能（当然实际中可能还会有一个验证用户权限和合法性的对话框，这里从略），并实现了登录注销时刻用户操作菜单的转变。注意：这里 OnLogout中前两行代码是在注销的时候要把已经打开的窗口关闭而添加的， m\_pFuncOneFrame和 m\_pFuncTwoFrame的定义和作用请参见后面定义。

Step 3 ：添加一个新的子框架类 CDemoFrame ，其基类为 CMDIChildWnd 。添加两个新的视图类 CFuncOneView 和 CFuncTwoView 类，前者的基类为 CView ，后者为 CFormView 。当然为了添加 CFuncTwoView 类，需要先 Insert 一个对话框资源，并将 ID 改为 IDD\_FUNC\_TWO\_DLG，属性 Style 修改为“ child ”（默认为 Popup ）。这样在新建 CFuncTwoView 的时候选择该 DialogID 为 IDD\_FUNC\_TWO\_DLG 。注意将CDemoFrame 的构造函数改为 public （默认是 protected ）。

Step 4 ：为 CMainFrame 添加两个成员变量记录各个业务逻辑对应的窗口，在 MainFrm.h 中添加：

|  |
| --- |
| CDemoFrame\* m\_pFuncOneFrame;  CDemoFrame \* m\_pFuncTwoFrame; |

并在 CMainFrame::CMainFrame() 中初始化：

|  |
| --- |
| CMainFrame::CMainFrame()  {  // TODO: add member initialization code here  m\_pFuncOneFrame = NULL;  m\_pFuncTwoFrame = NULL;  } |

Step 5 ：为“业务逻辑一”添加响应函数（ CMianFrame 中）：

|  |
| --- |
| void CMainFrame::OnFuncOne()  {  // TODO: Add your command handler code here  if (m\_pFuncOneFrame != NULL)  {  m\_pFuncOneFrame->MDIActivate();  return ;  }  m\_pFuncOneFrame = new CDemoFrame();  CCreateContext context;  context.m\_pNewViewClass = RUNTIME\_CLASS(CFuncOneView);  m\_pFuncOneFrame->LoadFrame(IDR\_MAINFRAME, WS\_MAXIMIZEWS\_OVERLAPPEDWINDOW, this, &context);  m\_pFuncOneFrame->SetWindowText(" 业务逻辑一 ");  m\_pFuncOneFrame->ShowWindow(SW\_SHOWMAXIMIZED);  m\_pFuncOneFrame->InitialUpdateFrame(NULL,true);  } |

同上，为“业务逻辑二”添加响应函数：

|  |
| --- |
| void CMainFrame::OnFuncTwo()  {  // TODO: Add your command handler code here  if (m\_pFuncTwoFrame != NULL)  {  m\_pFuncTwoFrame->MDIActivate();  return ;  }  m\_pFuncTwoFrame = new CDemoFrame();  CCreateContext context;  context.m\_pNewViewClass = RUNTIME\_CLASS(CFuncTwoView);  m\_pFuncTwoFrame->LoadFrame(IDR\_MAINFRAME, WS\_MAXIMIZEWS\_OVERLAPPEDWINDOW, this, &context);  m\_pFuncTwoFrame->SetWindowText(" 业务逻辑二 ");  m\_pFuncTwoFrame->ShowWindow(SW\_SHOWMAXIMIZED);  m\_pFuncTwoFrame->InitialUpdateFrame(NULL,true);  } |

这样，上述的需求情景基本是做到了，但是需要说明的是：

1 ） 需要在适当的地方加入适当的头文件，就是说在使用类的时候要 include 其实现的头文件。

2 ） 上面其实是很多的 MIS 管理系统的通用界面操作模版，大家可以在实际的项目开发中作相应的修改（主要是实现相应的业务逻辑等）；

3 ） 这里提供的默认的主框架窗口比较简单（默认）；

4 ） 上面的例子中，是让其生成不支持 MFC Document/View Architecture ，如果已经有了默认的支持 MFC Document/View Architecture 的程序，请将 App 中模仿上面的实现即可。

5 ） 上面可以使用 MFC AppWizard 生成的 CChildFrame 类代替 CDemoFrame 类，但是建议新建框架类，业务逻辑一和业务逻辑二可以使用不同的框架类，模仿实现即可。

当然，你可能需要为视图添加文档以实现业务逻辑和表现层的松耦合，下面就为 CFuncOneView添加文档视图结构支持。

Step 6 ：添加一个文档类 CDemoDoc ，基类为 CDocument 并将 CDemoDoc 的构造函数改为 public （默认为 protected ），为了演示，为CDemoDoc 添加函数 GetData （）：

|  |
| --- |
| CString CDemoDoc::GetData()  {  return "Hello world";  } |

Step 7 ：为 CFuncOneView添加函数 GetDocument ：

|  |
| --- |
| CDemoDoc\* CFuncOneView::GetDocument()  {  return (CDemoDoc\*)m\_pDocument;  } |

并修改 OnDraw 函数：

|  |
| --- |
| void CFuncOneView::OnDraw(CDC\* pDC)  {  //CDocument\* pDoc = GetDocument();  // TODO: add draw code here  CDemoDoc\* pDoc = GetDocument();  pDC->TextOut(50,50,pDoc->GetData());  } |

Step 8 ：将 CDemoDoc 和 CFuncTwoView 关联：修改 void CMainFrame::OnFuncOne()函数为：

|  |
| --- |
| void CMainFrame::OnFuncOne()  {  // TODO: Add your command handler code here  if (m\_pFuncOneFrame != NULL)  {  m\_pFuncOneFrame->MDIActivate();  return ;  }  m\_pFuncOneFrame = new CDemoFrame();  CDemoDoc\* m\_pDoc = new CDemoDoc();  CCreateContext context;  context.m\_pNewViewClass = RUNTIME\_CLASS(CFuncOneView);  context.m\_pCurrentDoc = m\_pDoc;  m\_pFuncOneFrame->LoadFrame(IDR\_MAINFRAME, WS\_MAXIMIZEWS\_OVERLAPPEDWINDOW, this, &context);  m\_pFuncOneFrame->SetWindowText(" 业务逻辑一 ");  m\_pFuncOneFrame->ShowWindow(SW\_SHOWMAXIMIZED);  m\_pFuncOneFrame->InitialUpdateFrame(NULL,true);  } |

至此，就完成了整个过程。需要说明的是：

1 ） 需要在适当的地方加入适当的头文件，就是说在使用类的时候要 include 其实现的头文件。

2 ） 如果是已经有了文档视图支持的程序，上面的文档可以使用系统中提供的文档；

上面提供了一个简单通用的界面操作的实现，大家可以参照实现，例如添加更多的视图、实现自己的业务逻辑等。

1、模板、文档、视图、框架的关系  
  
　　连载1~5我们各个击破地讲解了文档、文档模板、视图和框架类，连载1已经强调这些类有着亲密的内部联系，总结1~5我们可以概括其联系为：  
  
　　（1）文档保留该文档的视图列表和指向创建该文档的文档模板的指针；文档至少有一个相关联的视图，而视图只能与一个文档相关联。  
  
　　（2）视图保留指向其文档的指针，并被包含在其父框架窗口中；  
  
　　（3）文档框架窗口（即包含视图的MDI子窗口）保留指向其当前活动视图的指针；  
  
　　（4）文档模板保留其已打开文档的列表，维护框架窗口、文档及视图的映射；  
  
　　（5）应用程序保留其文档模板的列表。   
  
　　我们可以通过一组函数让这些类之间相互可访问，表6-1给出这些函数。  
  
　　表6-1 文档、文档模板、视图和框架类的互相访问

|  |  |
| --- | --- |
| 从该对象 | 如何访问其他对象 |
| 全局函数 | 调用全局函数AfxGetApp可以得到CWinApp应用类指针 |
| 应用 | AfxGetApp()->m\_pMainWnd为框架窗口指针；用CWinApp::GetFirstDocTemplatePostion、CWinApp::GetNextDocTemplate来遍历所有文档模板 |
| 文档 | 调用CDocument::GetFirstViewPosition，CDocument::GetNextView来遍历所有和文档关联的视图；调用CDocument:: GetDocTemplate 获取文档模板指针 |
| 文档模板 | 调用CDocTemplate::GetFirstDocPosition、CDocTemplate::GetNextDoc来遍历所有对应文档 |
| 视图 | 调用CView::GetDocument 得到对应的文档指针； 调用CView::GetParentFrame 获取框架窗口 |
| 文档框架窗口 | 调用CFrameWnd::GetActiveView 获取当前得到当前活动视图指针； 调用CFrameWnd::GetActiveDocument 获取附加到当前视图的文档指针 |
| MDI 框架窗口 | 调用CMDIFrameWnd::MDIGetActive 获取当前活动的MDI子窗口（CMDIChildWnd） |

　　我们列举一个例子，综合应用上表中的函数，写一段代码，它完成遍历文档模板、文档和视图的功能：

|  |
| --- |
| CMyApp \*pMyApp = (CMyApp\*)AfxGetApp(); //得到应用程序指针 POSITION p = pMyApp->GetFirstDocTemplatePosition();//得到第1个文档模板 while (p != NULL) //遍历文档模板 { 　CDocTemplate \*pDocTemplate = pMyApp->GetNextDocTemplate(p); 　POSITION p1 = pDocTemplate->GetFirstDocPosition();//得到文档模板对应的第1个文档 　while (p1 != NULL) //遍历文档模板对应的文档 　{ 　　CDocument \*pDocument = pDocTemplate->GetNextDoc(p1); 　　POSITION p2 = pDocument->GetFirstViewPosition(); //得到文档对应的第1个视图 　　while (p2 != NULL) //遍历文档对应的视图 　　{ 　　　CView \*pView = pDocument->GetNextView(p2); 　　} 　} } |

　　由此可见，下面的管理关系和实现途径都是完全类似的：  
  
　　（1）应用程序之于文档模板；  
  
　　（2）文档模板之于文档；  
  
　　（3）文档之于视图。  
  
　　图6.1、6.2分别给出了一个多文档/视图框架MFC程序的组成以及其中所包含类的层次关系。

|  |
| --- |
| 9cb19283a889b3e90df4d274 图6.1 多文档/视图框架MFC程序的组成 |

|  |
| --- |
| 83321ba310ca52e6caefd074 图6.2 文档/视图框架程序类的层次关系 |

　　关于文档和视图的关系，我们可进一步细分为三类：  
  
　　（1）文档对应多个相同的视图对象，每个视图对象在一个单独的 MDI 文档框架窗口中；  
  
　　（2）文档对应多个相同类的视图对象，但这些视图对象在同一文档框架窗口中（通过"拆分窗口"即将单个文档窗口的视图空间拆分成多个单独的文档视图实现）；   
  
　　（3）文档对应多个不同类的视图对象，这些视图对象仅在一个单独的 MDI 文档框架窗口中。在此模型中，由不同的类构造成的多个视图共享单个框架窗口，每个视图可提供查看同一文档的不同方式。例如，一个视图以字处理模式显示文档，而另一个视图则以"文档结构图"模式显示文档。  
  
　　图6.3显示了对应三种文档与视图关系应用程序的界面特点。

|  |
| --- |
| 3d71faacd5d115424a36d674 图6.3文档/视图的三种关系 |

2. 消息流动机制  
  
　　在基于"文档/视图"架构的MFC程序中，用户消息（鼠标、键盘输入等）会先发往视图，如果视图未处理则会发往框架窗口。所以，一般来说，消息映射宜定义在视图中。另外，如果一个应用同时拥有多个视图而当前活动视图没有对消息进行处理则消息也会发往框架窗口。  
  
　　下面我们来看实例，我们利用Visual C++向导创建一个单文档/视图架构的MFC程序，在其中增加一个菜单项为"自定义"（ID为IDM\_SELF，如图6.4）。

|  |
| --- |
| fa6abd086b08cfe42fddd474 图6.4 含"自定义"菜单的单文档/视图架构MFC程序 |

　　我们分别在视图类和框架窗口类中为"自定义"菜单添加消息映射，代码如下：

|  |
| --- |
| //视图中的消息映射和处理函数 BEGIN\_MESSAGE\_MAP(CExampleView, CView) 　//{{AFX\_MSG\_MAP(CExampleView) 　　ON\_COMMAND(IDM\_SELF, OnSelf) 　//}}AFX\_MSG\_MAP END\_MESSAGE\_MAP()  void CExampleView::OnSelf()  { 　// TODO: Add your command handler code here 　AfxMessageBox("消息在视图中处理"); }  //框架窗口中的消息映射和处理函数 BEGIN\_MESSAGE\_MAP(CMainFrame, CFrameWnd) 　//{{AFX\_MSG\_MAP(CMainFrame) 　　ON\_COMMAND(IDM\_SELF, OnSelf) 　//}}AFX\_MSG\_MAP END\_MESSAGE\_MAP()  void CMainFrame::OnSelf()  { 　// TODO: Add your command handler code here 　AfxMessageBox("消息在框架窗口中处理"); } |

　　这时候，我们单击"自定义"菜单，弹出对话框显示"消息在视图中处理"；如果我们删除框架窗口中的消息映射，再单击"自定义"菜单，弹出对话框也显示"消息在视图中处理"；但是，若我们将视图中的消息映射删除了，就会显示"消息在框架窗口中处理"！这验证了我们关于消息处理顺序论述的正确性。  
  
　　欲深入理解消息流动过程，还需认真分析CFrameWnd::OnCmdMsg、CView::OnCmdMsg函数的源代码：

|  |
| --- |
| BOOL CFrameWnd::OnCmdMsg(UINT nID, int nCode, void\* pExtra, AFX\_CMDHANDLERINFO\* pHandlerInfo) { 　// pump through current view FIRST 　CView\* pView = GetActiveView(); 　if (pView != NULL && pView->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo)) 　　return TRUE;  　// then pump through frame 　if (CWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo)) 　　return TRUE;  　// last but not least, pump through app 　CWinApp\* pApp = AfxGetApp(); 　if (pApp != NULL && pApp->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo)) 　　return TRUE;  　return FALSE; }  BOOL CView::OnCmdMsg(UINT nID, int nCode, void\* pExtra, AFX\_CMDHANDLERINFO\* pHandlerInfo) { 　// first pump through pane 　if (CWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo)) 　　return TRUE;  　// then pump through document 　BOOL bHandled = FALSE; 　if (m\_pDocument != NULL) 　{ 　　// special state for saving view before routing to document 　　\_AFX\_THREAD\_STATE\* pThreadState = AfxGetThreadState(); 　　CView\* pOldRoutingView = pThreadState->m\_pRoutingView; 　　pThreadState->m\_pRoutingView = this; 　　bHandled = m\_pDocument->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo); 　　pThreadState->m\_pRoutingView = pOldRoutingView; 　}  　return bHandled; } |

　　分析上述源代码可知，WM\_COMMAND消息的实际流动顺序比前文叙述的"先视图，后框架窗口"要复杂得多，文档和应用程序都参与了消息的处理过程。如果我们再为文档和应用添加消息映射和处理函数：

|  |
| --- |
| //文档的消息映射和处理函数 BEGIN\_MESSAGE\_MAP(CExampleDoc, CDocument) 　//{{AFX\_MSG\_MAP(CExampleDoc) 　　ON\_COMMAND(IDM\_SELF, OnSelf) 　//}}AFX\_MSG\_MAP END\_MESSAGE\_MAP()  void CExampleDoc::OnSelf()  { 　// TODO: Add your command handler code here 　AfxMessageBox("消息在文档中处理"); }  //应用的消息映射和处理函数 BEGIN\_MESSAGE\_MAP(CExampleApp, CWinApp) //{{AFX\_MSG\_MAP(CExampleApp)  ON\_COMMAND(IDM\_SELF, OnSelf) //}}AFX\_MSG\_MAP END\_MESSAGE\_MAP()  void CExampleApp::OnSelf()  { 　// TODO: Add your command handler code here 　AfxMessageBox("消息在应用中处理"); } |

　　屏蔽掉视图和框架窗口的消息映射，再单击"自定义"菜单，弹出对话框显示"消息在文档中处理"；再屏蔽掉文档中的消息映射，弹出对话框显示"消息在应用中处理"！由此可见，完整的WM\_COMMAND消息的处理顺序是"视图――文档――框架窗口――应用"！  
  
　　实际上，关于MFC的消息流动是一个很复杂的议题，陷于篇幅的原因，我们不可能对其进行更详尽的介绍，读者可自行寻找相关资料。

**深入浅出MFC文档/视图架构之框架**

2010-11-19 11:14

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 从前文可知，在MFC中，文档是真正的数据载体，视图是文档的显示界面，对应同一个文档，可能存在多个视图界面，我们需要另外一种东东来将这些界面管理起来，这个东东就是框架。  　　MFC创造框架类的初衷在于：把界面管理工作独立出来！框架窗口为应用程序的用户界面提供结构框架，它是应用程序的主窗口，负责管理其包容的窗口。一个应用程序启动时会创建一个最顶层的框架窗口。  　　MFC提供二种类型的框架窗口：单文档窗口SDI和多文档窗口MDI（你可以认为对话框是另一种框架窗口）。单文档窗口一次只能打开一个文档框架窗口，而多文档窗口应用程序中可以打开多个文档框架窗口，即子窗口（Child Window）。这些子窗口中的文档可以为同种类型，也可以为不同类型。  　　在Visual C++ AppWizard的第一个对话框中，会让用户选择应用程序是基于单文档、多文档还是基于对话框的，如图5.1。   |  | | --- | | 49753c370cf01a7a0b55a962 图5.1 在AppWizard中选择框架窗口 |   　　MFC提供了三个类CFrameWnd、CMDIFrameWnd、CMDIChildWnd用于支持单文档窗口和多文档窗口，这些类的层次结构如图5.2。   |  | | --- | | c6865a8e7adf6bb1503d9262 图5.2 CFrameWnd、CMDIFrameWnd、CMDIChildWnd类的层次 |   　　（1）CFrameWnd类用于SDI应用程序的框架窗口，SDI框架窗口既是应用程序的主框架窗口，也是当前文档对应的视图的边框；CFrameWnd类也作为CMDIFrameWnd和CMDIChildWnd类的父类，而在基于SDI的应用程序中，AppWizard会自动为我们添加一个继承自CFrameWnd类的CMainFrame类。  　　CFrameWnd类中重要的函数有Create（用于创建窗口）、LoadFrame（用于从资源文件中创建窗口）、PreCreateWindow（用于注册窗口类）等。Create函数第一个参数为窗口注册类名，第二个参数为窗口标题，其余几个参数指定了窗口的风格、大小、父窗口、菜单名等，其源代码如下：   |  | | --- | | BOOL CFrameWnd::Create(LPCTSTR lpszClassName, LPCTSTR lpszWindowName, DWORD dwStyle, const RECT &rect, CWnd \*pParentWnd, LPCTSTR lpszMenuName, DWORD dwExStyle, CCreateContext \*pContext) { 　HMENU hMenu = NULL; 　if (lpszMenuName != NULL) 　{ 　　// load in a menu that will get destroyed when window gets destroyed 　　HINSTANCE hInst = AfxFindResourceHandle(lpszMenuName, RT\_MENU); 　　if ((hMenu = ::LoadMenu(hInst, lpszMenuName)) == NULL) 　　{ 　　　TRACE0("Warning: failed to load menu for CFrameWnd.\n"); 　　　PostNcDestroy(); // perhaps delete the C++ object 　　　return FALSE; 　　} 　}  　m\_strTitle = lpszWindowName; // save title for later  　if (!CreateEx(dwExStyle, lpszClassName, lpszWindowName, dwStyle, rect.left, rect.top, rect.right - rect.left, rect.bottom - rect.top, pParentWnd ->GetSafeHwnd(), hMenu, (LPVOID)pContext)) 　{ 　　TRACE0("Warning: failed to create CFrameWnd.\n"); 　　if (hMenu != NULL) 　　　DestroyMenu(hMenu); 　　return FALSE; 　} 　return TRUE; } |   　　LoadFrame函数用于从资源文件中创建窗口，我们通常只需要给其指定一个参数，LoadFrame使用该参数从资源中获取主边框窗口的标题、图标、菜单、加速键等，其源代码为：   |  | | --- | | BOOL CFrameWnd::LoadFrame(UINT nIDResource, DWORD dwDefaultStyle, CWnd \*pParentWnd, CCreateContext \*pContext) { 　// only do this once 　ASSERT\_VALID\_IDR(nIDResource); 　ASSERT(m\_nIDHelp == 0 || m\_nIDHelp == nIDResource);  　m\_nIDHelp = nIDResource; // ID for help context (+HID\_BASE\_RESOURCE)  　CString strFullString; 　if (strFullString.LoadString(nIDResource)) 　　AfxExtractSubString(m\_strTitle, strFullString, 0); 　　// first sub-string  　VERIFY(AfxDeferRegisterClass(AFX\_WNDFRAMEORVIEW\_REG));  　// attempt to create the window 　LPCTSTR lpszClass = GetIconWndClass(dwDefaultStyle, nIDResource); 　LPCTSTR lpszTitle = m\_strTitle; 　if (!Create(lpszClass, lpszTitle, dwDefaultStyle, rectDefault, pParentWnd, MAKEINTRESOURCE(nIDResource), 0L, pContext)) 　{ 　　return FALSE; // will self destruct on failure normally 　}  　// save the default menu handle 　ASSERT(m\_hWnd != NULL); 　m\_hMenuDefault = ::GetMenu(m\_hWnd);  　// load accelerator resource 　LoadAccelTable(MAKEINTRESOURCE(nIDResource));  　if (pContext == NULL) 　　// send initial update 　　SendMessageToDescendants(WM\_INITIALUPDATE, 0, 0, TRUE, TRUE);  　return TRUE; } |   　　在SDI程序中，如果需要修改窗口的默认风格，程序员需要修改CMainFrame类的PreCreateWindow函数，因为AppWizard给我们生成的CMainFrame::PreCreateWindow仅对其基类的PreCreateWindow函数进行调用，CFrameWnd::PreCreateWindow的源代码如下：   |  | | --- | | BOOL CFrameWnd::PreCreateWindow(CREATESTRUCT &cs) { 　if (cs.lpszClass == NULL) 　{ 　　VERIFY(AfxDeferRegisterClass(AFX\_WNDFRAMEORVIEW\_REG)); 　　cs.lpszClass = \_afxWndFrameOrView; // COLOR\_WINDOW background 　}  　if ((cs.style &FWS\_ADDTOTITLE) && afxData.bWin4)cs.style |= FWS\_PREFIXTITLE;  　if (afxData.bWin4) 　　cs.dwExStyle |= WS\_EX\_CLIENTEDGE; 　return TRUE; } |   （2）CMDIFrameWnd类用于MDI应用程序的主框架窗口，主框架窗口是所有MDI文档子窗口的容器，并与子窗口共享菜单；CMDIFrameWnd类相较CFrameWnd类增加的重要函数有：MDIActivate（激活另一个MDI子窗口）、MDIGetActive（得到目前的活动子窗口）、MDIMaximize（最大化一个子窗口）、MDINext（激活目前活动子窗口的下一子窗口并将当前活动子窗口排入所有子窗口末尾）、MDIRestore（还原MDI子窗口）、MDISetMenu（设置MDI子窗口对应的菜单）、MDITile（平铺子窗口）、MDICascade（重叠子窗口）。  　　Visual C++开发环境是典型的MDI程序，其执行MDI Cascade的效果如图5.3。   |  | | --- | | 10ba730f51db37ac37d1226d 图5.3 MDI Cascade的效果 |   　　而执行MDI Tile的效果则如图5.4。   |  | | --- | | 4a699fe61ae64c65b838206d 图5.4 MDI Tile的效果 |   　　MDISetMenu函数的重要意义体现在一个MDI程序可以为不同类型的文档（与文档模板关联）显示不同的菜单，例如下面的这个函数Load一个菜单，并将目前主窗口的菜单替换为该菜单：   |  | | --- | | void CMdiView::OnReplaceMenu()  { 　// Load a new menu resource named IDR\_SHORT\_MENU 　CMdiDoc\* pdoc = GetDocument(); 　pdoc->m\_DefaultMenu = ::LoadMenu(AfxGetResourceHandle(), MAKEINTRESOURCE(IDR\_SHORT\_MENU)); 　if (pdoc->m\_DefaultMenu == NULL) 　　return;  　// Get the parent window of this view window. The parent window is 　// a CMDIChildWnd-derived class. We can then obtain the MDI parent  　// frame window using the CMDIChildWnd\*. Then, replace the current  　// menu bar with the new loaded menu resource. 　CMDIFrameWnd\* frame = ((CMDIChildWnd \*) GetParent())->GetMDIFrame(); 　frame->MDISetMenu(CMenu::FromHandle(pdoc->m\_DefaultMenu), NULL); 　frame->DrawMenuBar(); } |   　　CMDIFrameWnd类另一个不讲"不足以服众"的函数是OnCreateClient，它是子框架窗口的创造者，其实现如下：   |  | | --- | | BOOL CMDIFrameWnd::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext\*) { 　CMenu\* pMenu = NULL; 　if (m\_hMenuDefault == NULL) 　{ 　　// default implementation for MFC V1 backward compatibility 　　pMenu = GetMenu(); 　　ASSERT(pMenu != NULL); 　　// This is attempting to guess which sub-menu is the Window menu. 　　// The Windows user interface guidelines say that the right-most 　　// menu on the menu bar should be Help and Window should be one 　　// to the left of that. 　　int iMenu = pMenu->GetMenuItemCount() - 2;  　　// If this assertion fails, your menu bar does not follow the guidelines 　　// so you will have to override this function and call CreateClient 　　// appropriately or use the MFC V2 MDI functionality. 　　ASSERT(iMenu >= 0); 　　pMenu = pMenu->GetSubMenu(iMenu); 　　ASSERT(pMenu != NULL); 　} 　return CreateClient(lpcs, pMenu); } |   　　从CMDIFrameWnd::OnCreateClient的源代码可以看出，其中真正起核心作用的是对函数CreateClient的调用：   |  | | --- | | BOOL CMDIFrameWnd::CreateClient(LPCREATESTRUCT lpCreateStruct, CMenu\* pWindowMenu) { 　ASSERT(m\_hWnd != NULL); 　ASSERT(m\_hWndMDIClient == NULL); 　DWORD dwStyle = WS\_VISIBLE | WS\_CHILD | WS\_BORDER | WS\_CLIPCHILDREN | WS\_CLIPSIBLINGS | MDIS\_ALLCHILDSTYLES; // allow children to be created invisible 　DWORD dwExStyle = 0; 　// will be inset by the frame  　if (afxData.bWin4) 　{ 　　// special styles for 3d effect on Win4 　　dwStyle &= ~WS\_BORDER; 　　dwExStyle = WS\_EX\_CLIENTEDGE; 　}  　CLIENTCREATESTRUCT ccs; 　ccs.hWindowMenu = pWindowMenu->GetSafeHmenu(); 　// set hWindowMenu for MFC V1 backward compatibility 　// for MFC V2, window menu will be set in OnMDIActivate 　ccs.idFirstChild = AFX\_IDM\_FIRST\_MDICHILD;  　if (lpCreateStruct->style & (WS\_HSCROLL|WS\_VSCROLL)) 　{ 　　// parent MDIFrame's scroll styles move to the MDICLIENT 　　dwStyle |= (lpCreateStruct->style & (WS\_HSCROLL|WS\_VSCROLL));  　　// fast way to turn off the scrollbar bits (without a resize) 　　ModifyStyle(WS\_HSCROLL|WS\_VSCROLL, 0, SWP\_NOREDRAW|SWP\_FRAMECHANGED); 　}  　// Create MDICLIENT control with special IDC 　if ((m\_hWndMDIClient = ::CreateWindowEx(dwExStyle, \_T("mdiclient"), NULL, dwStyle, 0, 0, 0, 0, m\_hWnd, (HMENU)AFX\_IDW\_PANE\_FIRST, AfxGetInstanceHandle(), (LPVOID)&ccs)) == NULL) 　{ 　　TRACE(\_T("Warning: CMDIFrameWnd::OnCreateClient: failed to create MDICLIENT.") \_T(" GetLastError returns 0x%8.8X\n"), ::GetLastError()); 　　return FALSE; 　} 　// Move it to the top of z-order 　::BringWindowToTop(m\_hWndMDIClient);  　return TRUE; } |   　　（3）CMDIChildWnd类用于在MDI主框架窗口中显示打开的文档。每个视图都有一个对应的子框架窗口，子框架窗口包含在主框架窗口中，并使用主框架窗口的菜单。  　　CMDIChildWnd类的一个重要函数GetMDIFrame()返回目前MDI客户窗口的父窗口，其实现如下：   |  | | --- | | CMDIFrameWnd \*CMDIChildWnd::GetMDIFrame() { 　HWND hWndMDIClient = ::GetParent(m\_hWnd); 　CMDIFrameWnd \*pMDIFrame; 　pMDIFrame = (CMDIFrameWnd\*)CWnd::FromHandle(::GetParent(hWndMDIClient)); 　return pMDIFrame; } |   　　利用AppWizard生成的名为"example"的MDI工程包含如图5.5所示的类。   |  | | --- | | 0754e42f970ff81b4ec2266d 图5.5 一个MDI工程包含的类 |   　　其中的CMainFrame继承自CMDIFrameWnd，CChildFrame类继承自CMDIChildWnd类，CExampleView视图类则负责在CMDIChildWnd类对应的子框架窗口中显示文档的数据。   　　文中只是对CMDIFrameWnd的CreateClient成员函数进行了介绍，实际上，CFrameWnd、CMDIChildWnd均包含CreateClient成员函数。我们经常通过重载CFrameWnd:: CreateClient、CMDIChildWnd:: CreateClient函数的方法来实现"窗口分割"，例如：   |  | | --- | | BOOL CChildFrame::OnCreateClient(LPCREATESTRUCT lpcs,  CCreateContext \*pContext) { 　… 　if (!m\_wndSplitter.Create(this, 2, 2, // 分割的行、列数 　　　CSize(10, 10), // 最小化尺寸 　　　pContext)) 　{ 　　TRACE0("创建分割失败"); 　　return FALSE;  　} 　… 　return TRUE; } | |

**深入浅出MFC文档/视图架构之视图**

2010-11-19 11:12

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **视图类CView** 　　在MFC"文档/视图"架构中，CView类是所有视图类的基类，它提供了用户自定义视图类的公共接口。在"文档/视图"架构中，文档负责管理和维护数据；而视图类则负责如下工作：  　　（1） 从文档类中将文档中的数据取出后显示给用户；  　　（2） 接受用户对文档中数据的编辑和修改；  　　（3） 将修改的结果反馈给文档类，由文档类将修改后的内容保存到磁盘文件中。  　　文档负责了数据真正在永久介质中的存储和读取工作，视图呈现只是将文档中的数据以某种形式向用户呈现，因此一个文档可对应多个视图。  　　下面我们来看看CView类的声明：   |  | | --- | | class CView : public CWnd { 　DECLARE\_DYNAMIC(CView)  　// Constructors protected: 　CView();  　// Attributes public: 　CDocument\* GetDocument() const;  　// Operations public: 　// for standard printing setup (override OnPreparePrinting) 　BOOL DoPreparePrinting(CPrintInfo\* pInfo);  　// Overridables public: 　virtual BOOL IsSelected(const CObject\* pDocItem) const; // support for OLE  　// OLE scrolling support (used for drag/drop as well) 　virtual BOOL OnScroll(UINT nScrollCode, UINT nPos, BOOL bDoScroll = TRUE); 　virtual BOOL OnScrollBy(CSize sizeScroll, BOOL bDoScroll = TRUE);  　// OLE drag/drop support 　virtual DROPEFFECT OnDragEnter(COleDataObject\* pDataObject,DWORD dwKeyState, CPoint point); 　virtual DROPEFFECT OnDragOver(COleDataObject\* pDataObject,DWORD dwKeyState, CPoint point); 　virtual void OnDragLeave(); 　virtual BOOL OnDrop(COleDataObject\* pDataObject,DROPEFFECT dropEffect, CPoint point); 　virtual DROPEFFECT OnDropEx(COleDataObject\* pDataObject, 　DROPEFFECT dropDefault, DROPEFFECT dropList, CPoint point); 　virtual DROPEFFECT OnDragScroll(DWORD dwKeyState, CPoint point);  　virtual void OnPrepareDC(CDC\* pDC, CPrintInfo\* pInfo = NULL);  　virtual void OnInitialUpdate(); // called first time after construct  protected: 　// Activation 　virtual void OnActivateView(BOOL bActivate, CView\* pActivateView,CView\* pDeactiveView); 　virtual void OnActivateFrame(UINT nState, CFrameWnd\* pFrameWnd);  　// General drawing/updating 　virtual void OnUpdate(CView\* pSender, LPARAM lHint, CObject\* pHint); 　virtual void OnDraw(CDC\* pDC) = 0;  　// Printing support 　virtual BOOL OnPreparePrinting(CPrintInfo\* pInfo); 　// must override to enable printing and print preview  　virtual void OnBeginPrinting(CDC\* pDC, CPrintInfo\* pInfo); 　virtual void OnPrint(CDC\* pDC, CPrintInfo\* pInfo); 　virtual void OnEndPrinting(CDC\* pDC, CPrintInfo\* pInfo);  　// Advanced: end print preview mode, move to point 　virtual void OnEndPrintPreview(CDC\* pDC, CPrintInfo\* pInfo, POINT point,CPreviewView\* pView);  　// Implementation public: 　virtual ~CView(); 　#ifdef \_DEBUG 　　virtual void Dump(CDumpContext&) const; 　　virtual void AssertValid() const; 　#endif //\_DEBUG  　// Advanced: for implementing custom print preview 　BOOL DoPrintPreview(UINT nIDResource, CView\* pPrintView,CRuntimeClass\* pPreviewViewClass, CPrintPreviewState\* pState);  　virtual void CalcWindowRect(LPRECT lpClientRect,UINT nAdjustType = adjustBorder); 　virtual CScrollBar\* GetScrollBarCtrl(int nBar) const; 　static CSplitterWnd\* PASCAL GetParentSplitter(const CWnd\* pWnd, BOOL bAnyState);  protected: 　CDocument\* m\_pDocument;  public: 　virtual BOOL OnCmdMsg(UINT nID, int nCode, void\* pExtra,AFX\_CMDHANDLERINFO\* pHandlerInfo); protected: 　virtual BOOL PreCreateWindow(CREATESTRUCT& cs); 　virtual void PostNcDestroy();  　// friend classes that call protected CView overridables 　friend class CDocument; 　friend class CDocTemplate; 　friend class CPreviewView; 　friend class CFrameWnd; 　friend class CMDIFrameWnd; 　friend class CMDIChildWnd; 　friend class CSplitterWnd; 　friend class COleServerDoc; 　friend class CDocObjectServer;  　//{{AFX\_MSG(CView) 　　afx\_msg int OnCreate(LPCREATESTRUCT lpcs); 　　afx\_msg void OnDestroy(); 　　afx\_msg void OnPaint(); 　　afx\_msg int OnMouseActivate(CWnd\* pDesktopWnd, UINT nHitTest, UINT message); 　　// commands 　　afx\_msg void OnUpdateSplitCmd(CCmdUI\* pCmdUI); 　　afx\_msg BOOL OnSplitCmd(UINT nID); 　　afx\_msg void OnUpdateNextPaneMenu(CCmdUI\* pCmdUI); 　　afx\_msg BOOL OnNextPaneCmd(UINT nID);  　　// not mapped commands - must be mapped in derived class 　　afx\_msg void OnFilePrint(); 　　afx\_msg void OnFilePrintPreview(); 　//}}AFX\_MSG 　DECLARE\_MESSAGE\_MAP() }; |   　　CView类首先要维护文档与视图之间的关联，它通过CDocument\* m\_pDocument保护性成员变量记录关联文档的指针，并提供CView::GetDocument接口函数以使得应用程序可得到与视图关联的文档。而在CView类的析构函数中，需将对应文档类视图列表中的本视图删除：   |  | | --- | | CView::~CView() { 　if (m\_pDocument != NULL) 　　m\_pDocument->RemoveView(this); } |   　　CView中地位最重要的函数是virtual void OnDraw(CDC\* pDC) = 0;从这个函数的声明可以看出，CView是一个纯虚基类。这个函数必须被重载，它通常执行如下步骤：  　　（1） 以GetDocument()函数获得视图对应文档的指针；  　　（2） 读取对应文档中的数据；  　　（3） 显示这些数据。  　　以MFC向导建立的一个初始"文档/视图"架构工程将这样重载OnDraw()函数，注意注释中的"add draw code for native data here（添加活动数据的绘制代码）"：   |  | | --- | | ///////////////////////////////////////////////////////////////////////////// // CExampleView drawing void CExampleView::OnDraw(CDC\* pDC) { 　CExampleDoc\* pDoc = GetDocument(); 　ASSERT\_VALID(pDoc); 　// TODO: add draw code for native data here } CView::PreCreateWindow负责View的初始化： ///////////////////////////////////////////////////////////////////////////// // CView second phase construction - bind to document BOOL CView::PreCreateWindow(CREATESTRUCT & cs) { 　ASSERT(cs.style & WS\_CHILD);  　if (cs.lpszClass == NULL) 　{ 　　VERIFY(AfxDeferRegisterClass(AFX\_WNDFRAMEORVIEW\_REG)); 　　cs.lpszClass = \_afxWndFrameOrView; // COLOR\_WINDOW background 　}  　if (afxData.bWin4 && (cs.style & WS\_BORDER)) 　{ 　　cs.dwExStyle |= WS\_EX\_CLIENTEDGE; 　　cs.style &= ~WS\_BORDER; 　}  　return TRUE; } |   　　CView::OnUpdate函数在文档的数据被改变的时候被调用（即它被用来通知一个视图的关联文档的内容已经被修改），它预示着我们需要重新绘制视图以显示变化后的数据。其中的Invalidate(TRUE)将整个窗口设置为需要重绘的无效区域，它会产生WM\_PAINT消息，这样OnDraw将被调用：   |  | | --- | | void CView::OnUpdate(CView\* pSender, LPARAM /\*lHint\*/, CObject\* /\*pHint\*/) { 　ASSERT(pSender != this); 　UNUSED(pSender); // unused in release builds  　// invalidate the entire pane, erase background too 　Invalidate(TRUE); } |   　　假如文档中的数据发生了变化，必须通知所有链接到该文档的视图，这时候文档类的UpdateAllViews函数需要被调用。  　　此外，CView类包含一系列函数用于进行文档的打印及打印预览工作：  　　（1）CView::OnBeginPrinting在打印工作开始时被调用，用来分配GDI资源；  　　（2）CView::OnPreparePrinting函数在文档打印或者打印预览前被调用，可用来初始化打印对话框；  　　（3）CView::OnPrint用来打印或打印预览文档；  　　（4）CView::OnEndPrinting函数在打印工作结束时被调用，用以释放GDI资源；  　　（5）CView::OnEndPrintPreview在退出打印预览模式时被调用。  　　**CView派生类**  　　MFC提供了丰富的CView派生类，各种不同的派生类实现了对不同种类控件的支持，以为用户提供多元化的显示界面。这些CView派生类包括：  　　(1)CScrollView：提供滚动支持；  　　(2)CCtrlView：支持tree、 list和rich edit控件；  　　(3)CDaoRecordView：在dialog-box控件中显示数据库记录；  　　(4)CEditView：提供了一个简单的多行文本编辑器；  　　(5)CFormView：包含dialog-box控件，可滚动，基于对话框模板资源；   　　(6)CListView：支持list控件；  　　(7)CRecordView：在dialog-box控件中显示数据库记录；  　　(8)CRichEditView：支持rich edit控件；  　　(9)CTreeView：支持tree控件。  　　其中，CRichEditView、CTreeView及CListView均继承自CCtrlView类；CFormView继承自CScrollView类；CRecordView、CDaoRecordView则进一步继承自CFormView类。  　　下图描述了CView类体系的继承关系：   |  | | --- | | 7ecfeba5a0096abc9052ee99 | |

**深入浅出MFC文档/视图架构之文档模板(3)**

2007-08-27 10:15

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CDocTemplate类的AddDocument、RemoveDocument成员函数使得CDocument\* pDoc参数所指向的文档归属于本文档模板（通过将this指针赋值给pDoc所指向CDocument对象的m\_pDocTemplate成员变量）或脱离与本文档模板的关系：   |  | | --- | | void CDocTemplate::AddDocument(CDocument\* pDoc) { 　ASSERT\_VALID(pDoc); 　ASSERT(pDoc->m\_pDocTemplate == NULL); // no template attached yet 　pDoc->m\_pDocTemplate = this; } void CDocTemplate::RemoveDocument(CDocument\* pDoc) { 　ASSERT\_VALID(pDoc); 　ASSERT(pDoc->m\_pDocTemplate == this); // must be attached to us 　pDoc->m\_pDocTemplate = NULL; } |   　　而CDocTemplate类的CreateNewDocument成员函数则首先调用CDocument运行时类的CreateObject函数创建一个CDocument对象，再调用AddDocument成员函数将其归属于本文档模板类：   |  | | --- | | CDocument\* CDocTemplate::CreateNewDocument() { 　// default implementation constructs one from CRuntimeClass 　if (m\_pDocClass == NULL) 　{ 　　TRACE0("Error: you must override CDocTemplate::CreateNewDocument.\n"); 　　ASSERT(FALSE); 　　return NULL; 　} 　CDocument\* pDocument = (CDocument\*)m\_pDocClass->CreateObject(); 　if (pDocument == NULL) 　{ 　　TRACE1("Warning: Dynamic create of document type %hs failed.\n",m\_pDocClass->m\_lpszClassName); 　　return NULL; 　} 　ASSERT\_KINDOF(CDocument, pDocument); 　AddDocument(pDocument); 　return pDocument; } |   　　文档类对象由文档模板类构造生成，单文档模板类CSingleDocTemplate只能生成一个文档类对象，并用成员变量 m\_pOnlyDoc 指向该对象；多文档模板类可以生成多个文档类对象，用成员变量 m\_docList 指向文档对象组成的链表。  　　CSingleDocTemplate的构造函数、AddDocument及RemoveDocument成员函数都在CDocTemplate类相应函数的基础上增加了对m\_pOnlyDoc指针的处理：   |  | | --- | | CSingleDocTemplate::CSingleDocTemplate(UINT nIDResource, CRuntimeClass\* pDocClass, CRuntimeClass\* pFrameClass, CRuntimeClass\* pViewClass) : CDocTemplate(nIDResource, pDocClass, pFrameClass, pViewClass) { 　m\_pOnlyDoc = NULL; } void CSingleDocTemplate::AddDocument(CDocument\* pDoc) { 　ASSERT(m\_pOnlyDoc == NULL); // one at a time please 　ASSERT\_VALID(pDoc);  　CDocTemplate::AddDocument(pDoc); 　m\_pOnlyDoc = pDoc; } void CSingleDocTemplate::RemoveDocument(CDocument\* pDoc) { 　ASSERT(m\_pOnlyDoc == pDoc); // must be this one 　ASSERT\_VALID(pDoc);  　CDocTemplate::RemoveDocument(pDoc); 　m\_pOnlyDoc = NULL; } |   　　同样，CMultiDocTemplate类的相关函数也需要对m\_docList所指向的链表进行操作（实际上AddDocument和RemoveDocument成员函数是文档模板管理其所包含文档的函数）：   |  | | --- | | // CMultiDocTemplate document management (a list of currently open documents) void CMultiDocTemplate::AddDocument(CDocument\* pDoc) { 　ASSERT\_VALID(pDoc);  　CDocTemplate::AddDocument(pDoc); 　ASSERT(m\_docList.Find(pDoc, NULL) == NULL); // must not be in list 　m\_docList.AddTail(pDoc); } void CMultiDocTemplate::RemoveDocument(CDocument\* pDoc) { 　ASSERT\_VALID(pDoc);  　CDocTemplate::RemoveDocument(pDoc); 　m\_docList.RemoveAt(m\_docList.Find(pDoc)); } |   　　由于CMultiDocTemplate类可包含多个文档，依靠其成员函数GetFirstDocPosition和GetNextDoc完成对文档链表m\_docList的遍历：   |  | | --- | | POSITION CMultiDocTemplate::GetFirstDocPosition() const { 　return m\_docList.GetHeadPosition(); } CDocument\* CMultiDocTemplate::GetNextDoc(POSITION& rPos) const { 　return (CDocument\*)m\_docList.GetNext(rPos); } |   　　而CSingleDocTemplate的这两个函数实际上并无太大的意义，仅仅是MFC要玩的某种"招数"，这个"招数"高明吗？相信看完MFC的相关源代码后你或许不会这么认为，实际上CSingleDocTemplate的GetFirstDocPosition、GetNextDoc函数仅仅只能判断m\_pOnlyDoc的是否为NULL：   |  | | --- | | POSITION CSingleDocTemplate::GetFirstDocPosition() const { 　return (m\_pOnlyDoc == NULL) ? NULL : BEFORE\_START\_POSITION; }  CDocument\* CSingleDocTemplate::GetNextDoc(POSITION& rPos) const { 　CDocument\* pDoc = NULL; 　if (rPos == BEFORE\_START\_POSITION) 　{ 　　// first time through, return a real document 　　ASSERT(m\_pOnlyDoc != NULL); 　　pDoc = m\_pOnlyDoc; 　} 　rPos = NULL; // no more 　return pDoc; } |   　　笔者认为，MFC的设计者们将GetFirstDocPosition、GetNextDoc作为基类CDocTemplate的成员函数是不合理的，一种更好的做法是将GetFirstDocPosition、GetNextDoc移至CMultiDocTemplate派生类。  　　CDocTemplate还需完成对其对应文档的关闭与保存操作：   |  | | --- | | BOOL CDocTemplate::SaveAllModified() { 　POSITION pos = GetFirstDocPosition(); 　while (pos != NULL) 　{ 　　CDocument\* pDoc = GetNextDoc(pos); 　　if (!pDoc->SaveModified()) 　　　return FALSE; 　} 　return TRUE; } void CDocTemplate::CloseAllDocuments(BOOL) { 　POSITION pos = GetFirstDocPosition(); 　while (pos != NULL) 　{ 　　CDocument\* pDoc = GetNextDoc(pos); 　　pDoc->OnCloseDocument(); 　} } | |