前言

现在流行的 Windows 下的编程语言实在不少，所以在 BBS 上常常有人会问：我应该使用什么编程语言呢？其中，有一个大家认可的答案：真正的程序员使用 Visual C++。

的确，Visual C++是一个功能强大、灵活、方便的编程工具，可以完成其他编程语言所无法完成的任务，可以让程序员方便地实现自己的设计，尽情的发挥自己地创造性。

Visual C++的强大无比的功能除了得益于 C++的特性之外，更重要的是它具有体系完整、机制灵活、功能丰富的 MFC 类库。

所以，要讲 Visual C++，必须讲 MFC 类库。

MFC 的类库可以分两个层次，首先是实现 MFC 编程框架体系的核心 MFC 类库，然后是建立在核心 MFC 类库基础之上的扩展类库，例如，支持 COM 的类库，实现网络功能的类库，等等。随着 Visual C++的不断升级，MFC 类库的功能越来越丰富，越来越强大，但是，MFC 核心类库是相对稳定的，特别是从 Visual C++ 4.2 开始到现在的 Visual C++6.0。

本书的中心就是深入浅出地解析 MFC 类库，分析怎么使用 MFC 类库以及 MFC 类库的内部实现，揭开 MFC 复杂、深奥的面纱，让读者对 MFC 有一个全面、透彻、清晰的理解。关于 MFC 的核心实现，主要有以下几个方面。

首先，MFC 采用 C++的面向对象的特征封装了 Windows 的对象和 Win32 函数，一定程度上隐蔽了底层 Win32 的复杂性。

其次，MFC 采用消息映射的方法来处理 Windows 消息和事件，隐藏了 Windows 窗口的窗口过程，简化了消息处理的复杂性和烦琐性。

还有，MFC 提供了一个以文档-视为中心的编程模式，并实现了以文档-视为中心的编程框架，简化了数据处理的过程。

而且，MFC 提出了模块状态、线程状态、模块线程状态来支持多线程的编程设计和 DLL 的编程。

本书分别从使用 MFC 的角度和 MFC 内部设计及实现的角度讨论了上述内容，分析了 MFC 核心的设计和实现；然后，在此基础上，进一步讨论了 MFC 对一些常用类的实现。有关章节的内容如下：

第一章，MFC 概述。

第二章，解释 MFC 对 Win32 API 和 Windows 对象的封装，讨论各类 MFC 对象的使用，分析 MFC 对象和 Windows 对象的关系。

第三章，讨论 CObject 的特性及其实现，包括动态类信息、动态创建、序列化的实现等内容。

第四章，讨论 MFC 的消息映射机制，分析 MFC 对各类消息的处理，例如对 Windows 消息、控制通知消息、命令消息、状态更新消息、反射消息的处理等；并揭示了 MFC 通过消息映射手段实现 C++虚拟函数机制的原理。

第五章和第六章，分析 MFC 编程框架启动和关闭一个应用程序的过程，揭示 MFC 框架的内幕，剖析以文档模板为核心创建基于文档-视的应用程序的过程，展示 MFC 框架处理消息和调用虚拟函数的时机和位置。

第七、八、九章，介绍 MFC 的动态链接库、进程、线程等概念，以及 MFC 动态链接库的种类和使用，讨论 MFC 下多线程编程的问题。并且进一步阐述 MFC 的核心概念之一：状态（模块状态、线程状态、模块线程状态），揭示 MFC 对多线程的支持机制，MFC 实现规则 DLL 和扩展 DLL 的内幕。

第十章，阐述 MFC 下的调试手段。

第十一章，讨论 CFile 类，主要分析了 CFile 的使用和它对 Win32 文件函数的封装。

第十二章，讨论模式和无模式对话框，分析 MFC 如何设计和实现这两种对话框的功能，分析 CDialog 和 CFormView 为实现有关功能而设计的虚拟函数、消息处理函数等。

第十三章，讨论 MFC 工具栏和状态栏的设计及其实现，分析 MFC 是如何以 CControlBar 为基础，派生出 CStatusBar、CToolBar、CDialogBar 等子类，实现 MFC 工具栏和状态栏标准处理。

第十四章，讨论 MFC 的 Socket 类。

第一章到第十章介绍了 MFC 的核心概念以及实现。在此基础上，第十一章到第十四章讨论了 MFC 一些常用类的实现。

本书的内容对 MFC 的初学者（最好对 Visual C++和 Windows 有所了解）和提高者都是很有帮助的。

如果您是一个初学者，可以读第一至第六章。主要目的是建立对 MFC 的全面理解，了解 MFC 框架是如何支持程序员编程的。如果有读不懂的地方，可以跳过，直接阅读有关分析的结论。特别是第五章和第六章，可以重点阅读，了解 MFC 是怎样来处理有关消息、调用有关虚拟函数的。

然后，还可以读第十章，第十一至第十四章。特别第十二章，可以重点阅读，它是 MFC 从 CWnd 或者 CView 派生出特定的类实现特定功能的例子，可以帮助您进一步理解 MFC，并且学习如何设计和实现一个特定的类。

如果您对 MFC 有一定的掌握，可以进一步阅读第八和第九章，了解 MFC 处理 DLL 和线程的知识。对于第一至第六章、第十至第十四章，应该把重点放在 MFC 的设计和实现的分析上。这样，可以深化您对 MFC 和 Windows 编程的理解与掌握。

如果您可以较熟练地使用 MFC，建议您进一步阅读第九章，并且对所有有关章节的设计和实现分析作重点阅读，这样，不仅可以帮助您深入的理解和掌握 MFC，而且，从 MFC 的有关内部设计和实现上，必然可以提高您的程序设计和编写能力。

由于成书仓促，书中可能存在一些缺点和错误，恳请您不吝赐教！作者的电子邮箱：ljjin@public.szonline.net。

作者 李久进
1999 年 6 月

约定和说明

1．图解或者说明的流程都是 MFC 的缺省实现。

2．对于 Win32 全局函数，用“::”为前缀，以区分 MFC 的成员函数；如果从上下文可以明确判定一个函数是全局函数，则省掉“::”前缀。

3．本书图解时，使用灰色框表示注释。如果注释某个函数是某个类的成员函数，则表示该类是定义该函数的最上层的类。

4．流程图所描述的函数的流程不一定是该函数的程序流程，可能是该函数运行时的执行流程。比如：函数 A 图解为函数 B→函数 C，可以是函数 A 先调用函数 B，然后调用函数 C；或者函数 A 调用函数 B，函数 B 调用函数 C。

5．流程图省略了对 OLE 的处理。

6．流程图中表示调用了某个成员函数，使用了类限制符号。如果用正体表示类名和函数名（形式为 ClassName::FunctionName），则程序源码中没有类名约束，分几种情况：如果是虚拟函数则表示该函数动态约束的结果是调用了指定类的函数；如果是消息处理函数，表示指定类的消息处理函数被调用；如果不是上述两种情况，表示该函数调用了指定类的实现。如果用斜体表示表示类名和函数名（形式为 *ClassName::FunctionName*），则程序的源码明确使用了类的限制符号来调用函数。

7．动态连接到 MFC DLL 定义了_AFXDLL 符号。引用的 MFC 源码如果定义了该符号，则表示在动态连接情况下使用。

8．“MFC DLL”指 MFC 的核心 DLL，即 MFCXX.DLL。

9．某个类的对象表示该类的一个实例。例如，CWinApp 对象表示 CWinApp 类的一个实例。“类的实例”和“类的对象”两种说法可以互换。

10．MFC 对象是 C++对象，即一个 C++类的实例；Windows 对象是 Windows 操作系统定义的数据结构的实例。一个 Window 对象对应一个 MFC 对象。

11．“类的成员数据”和“类的成员变量” 两种说法可以互换，都表示类的属性。

12．文档和文件，使用“文件”的地方可以用“文档”代替，表示一个操作系统的文件；使用“文档”的地方一般还有“文档”对象的含义。

13．MFC 窗口类表示 CWnd 或其派生类；Windows“窗口类”表示程序注册的 Windows Window 对象的类，书中表示 Windows“窗口类”的地方用引号加以区分。

第1章　MFC 概述

1.1　MFC 是一个编程框架

MFC (Microsoft Foundation Class Library)中的各种类结合起来构成了一个应用程序框架，它的目的就是让程序员在此基础上来建立 Windows 下的应用程序，这是一种相对 SDK 来说更为简单的方法。因为总体上，MFC 框架定义了应用程序的轮廓，并提供了用户接口的标准实现方法，程序员所要做的就是通过预定义的接口把具体应用程序特有的东西填入这个轮廓。Microsoft Visual C++提供了相应的工具来完成这个工作：AppWizard 可以用来生成初步的框架文件（代码和资源等）；资源编辑器用于帮助直观地设计用户接口；ClassWizard 用来协助添加代码到框架文件；最后，编译，则通过类库实现了应用程序特定的逻辑。

1.1.1 封装

构成 MFC 框架的是 MFC 类库。MFC 类库是 C++类库。这些类或者封装了 Win32 应用程序编程接口，或者封装了应用程序的概念，或者封装了 OLE 特性，或者封装了 ODBC 和 DAO 数据访问的功能，等等，分述如下。

（1）对 Win32 应用程序编程接口的封装

用一个 C++ Object 来包装一个 Windows Object。例如：class CWnd 是一个 C++ window object，它把 Windows window(HWND)和 Windows window 有关的 API 函数封装在 C++ window object 的成员函数内，后者的成员变量 m_hWnd 就是前者的窗口句柄。

（2）对应用程序概念的封装

使用 SDK 编写 Windows 应用程序时，总要定义窗口过程，登记 Windows Class，创建窗口，等等。MFC 把许多类似的处理封装起来，替程序员完成这些工作。另外，MFC 提出了以文档-视图为中心的编程模式，MFC 类库封装了对它的支持。文档是用户操作的数据对象，视图是数据操作的窗口，用户通过它处理、查看数据。

（3）对 COM/OLE 特性的封装

OLE 建立在 COM 模型之上，由于支持 OLE 的应用程序必须实现一系列的接口（Interface），因而相当繁琐。MFC 的 OLE 类封装了 OLE API 大量的复杂工作，这些类提供了实现 OLE 的更高级接口。

（4）对 ODBC 功能的封装

以少量的能提供与 ODBC 之间更高级接口的 C++类，封装了 ODBC API 的大量的复杂的工作，提供了一种数据库编程模式。

1.1.2 继承

首先，MFC 抽象出众多类的共同特性，设计出一些基类作为实现其他类的基础。这些类中，最重要的类是 CObject 和 CCmdTarget。CObject 是 MFC 的根类，绝大多数 MFC 类是其派生的，包括 CCmdTarget。CObject 实现了一些重要的特性，包括动态类信息、动态创建、对

象序列化、对程序调试的支持，等等。所有从CObject派生的类都将具备或者可以具备CObject所拥有的特性。CCmdTarget 通过封装一些属性和方法，提供了消息处理的架构。MFC 中，任何可以处理消息的类都从 CCmdTarget 派生。
针对每种不同的对象，MFC 都设计了一组类对这些对象进行封装，每一组类都有一个基类，从基类派生出众多更具体的类。这些对象包括以下种类：窗口对象，基类是 CWnd；应用程序对象，基类是 CwinThread；文档对象，基类是 Cdocument，等等。
程序员将结合自己的实际，从适当的 MFC 类中派生出自己的类，实现特定的功能，达到自己的编程目的。

1.1.3 虚拟函数和动态约束

MFC 以“C++”为基础，自然支持虚拟函数和动态约束。但是作为一个编程框架，有一个问题必须解决：如果仅仅通过虚拟函数来支持动态约束，必然导致虚拟函数表过于臃肿，消耗内存，效率低下。例如，CWnd 封装 Windows 窗口对象时，每一条 Windows 消息对应一个成员函数，这些成员函数为派生类所继承。如果这些函数都设计成虚拟函数，由于数量太多，实现起来不现实。于是，MFC 建立了消息映射机制，以一种富有效率、便于使用的手段解决消息处理函数的动态约束问题。
这样，通过虚拟函数和消息映射，MFC 类提供了丰富的编程接口。程序员继承基类的同时，把自己实现的虚拟函数和消息处理函数嵌入 MFC 的编程框架。MFC 编程框架将在适当的时候、适当的地方来调用程序的代码。本书将充分的展示 MFC 调用虚拟函数和消息处理函数的内幕，让读者对 MFC 的编程接口有清晰的理解。

1.1.4 MFC 的宏观框架体系

如前所述，MFC 实现了对应用程序概念的封装，把类、类的继承、动态约束、类的关系和相互作用等封装起来。这样封装的结果对程序员来说，是一套开发模板（或者说模式）。针对不同的应用和目的，程序员采用不同的模板。例如，SDI 应用程序的模板，MDI 应用程序的模板，规则 DLL 应用程序的模板，扩展 DLL 应用程序的模板，OLE/ACTIVEX 应用程序的模板，等等。
这些模板都采用了以文档-视为中心的思想，每一个模板都包含一组特定的类。典型的 MDI 应用程序的构成将在下一节具体讨论。
为了支持对应用程序概念的封装，MFC 内部必须作大量的工作。例如，为了实现消息映射机制，MFC 编程框架必须要保证首先得到消息，然后按既定的方法进行处理。又如，为了实现对 DLL 编程的支持和多线程编程的支持，MFC 内部使用了特别的处理方法，使用模块状态、线程状态等来管理一些重要信息。虽然，这些内部处理对程序员来说是透明的，但是，懂得和理解 MFC 内部机制有助于写出功能灵活而强大的程序。

总之，MFC 封装了 Win32 API，OLE API，ODBC API 等底层函数的功能，并提供更高一层的接口，简化了 Windows 编程。同时，MFC 支持对底层 API 的直接调用。
MFC 提供了一个 Windows 应用程序开发模式，对程序的控制主要是由 MFC 框架完成的，而且 MFC 也完成了大部分的功能，预定义或实现了许多事件和消息处理，等等。框架或者由其本身处理事件，不依赖程序员的代码；或者调用程序员的代码来处理应用程序特定的事件。

MFC 是 C++类库，程序员就是通过使用、继承和扩展适当的类来实现特定的目的。例如，继承时，应用程序特定的事件由程序员的派生类来处理，不感兴趣的由基类处理。实现这种功能的基础是 C++对继承的支持，对虚拟函数的支持，以及 MFC 实现的消息映射机制。

1.2 MDI 应用程序的构成

本节解释一个典型的 MDI 应用程序的构成。

用 AppWizard 产生一个 MDI 工程 t（无 OLE 等支持），AppWizard 创建了一系列文件，构成了一个应用程序框架。这些文件分四类：头文件（.h），实现文件(.cpp)，资源文件(.rc)，模块定义文件(.def)，等。

1.2.1 构成应用程序的对象

图 1-1 解释了该应用程序的结构，箭头表示信息流向。

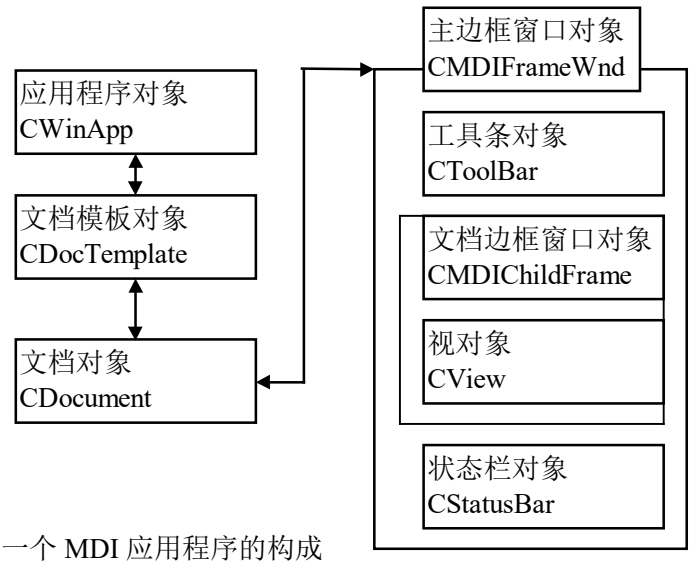


图 1-1 一个 MDI 应用程序的构成

从 CWinApp、CDocument、CView、CMDIFrameWnd、CMDIChildWnd 类对应地派生出 CTApp、CTDoc、CTView、CMainFrame、CChildFrame 五个类，这五个类的实例分别是应用程序对象、文档对象、视对象、主框架窗口对象和文档边框窗口对象。主框架窗口包含了视窗口、工具条和状态栏。对这些类或者对象解释如下。

（1）应用程序

应用程序类派生于 CWinApp。基于框架的应用程序必须有且只有一个应用程序对象，它负责应用程序的初始化、运行和结束。

（2）边框窗口

如果是 SDI 应用程序，从 CFrameWnd 类派生边框窗口类，边框窗口的客户子窗口(MDIClient)直接包含视窗口；如果是 MDI 应用程序，从 CMDIFrameWnd 类派生边框窗口类，边框窗口的客户子窗口(MDIClient)直接包含文档边框窗口。

如果要支持工具条、状态栏，则派生的边框窗口类还要添加 CToolBar 和 CStatusBar 类型的成员变量，以及在一个 OnCreate 消息处理函数中初始化这两个控制窗口。

边框窗口用来管理文档边框窗口、视窗口、工具条、菜单、加速键等，协调半模式状态（如上下文的帮助(SHIFT+F1 模式)和打印预览）。

（3）文档边框窗口

文档边框窗口类从 CMDIChildWnd 类派生，MDI 应用程序使用文档边框窗口来包含视窗口。

（4）文档

文档类从 CDocument 类派生，用来管理数据，数据的变化、存取都是通过文档实现的。视窗口通过文档对象来访问和更新数据。

（5）视

视类从 CView 或它的派生类派生。视和文档联系在一起，在文档和用户之间起中介作用，即视在屏幕上显示文档的内容，并把用户输入转换成对文档的操作。

（6）文档模板

文档模板类一般不需要派生。MDI 应用程序使用多文档模板类 CMultiDocTemplate；SDI 应用程序使用单文档模板类 CSingleDocTemplate。

应用程序通过文档模板类对象来管理上述对象（应用程序对象、文档对象、主边框窗口对象、文档边框窗口对象、视对象）的创建。

1.2.2 构成应用程序的对象之间的关系

这里，用图的形式可直观地表示所涉及的 MFC 类的继承或者派生关系，如图 1-2 所示意。图 1-2 所示的类都是从 CObject 类派生出来的；所有处理消息的类都是从 CCmdTarget 类派生的。如果是多文档应用程序，文档模板使用 CMultiDocTemplae，主框架窗口从 CMdiFarmeWnd 派生，它包含工具条、状态栏和文档框架窗口。文档框架窗口从 CMdiChildWnd 派生，文档框架窗口包含视，视从 CView 或其派生类派生。

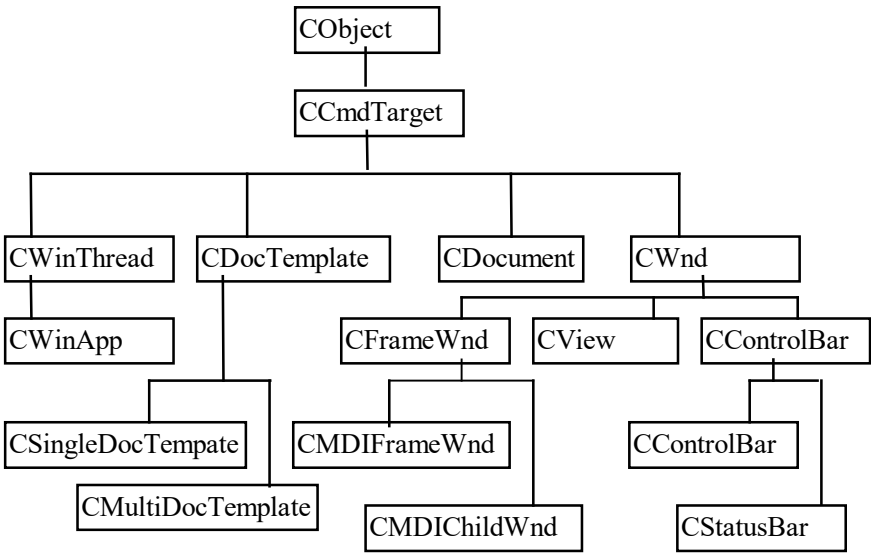


图 1-2　一些 MFC 类的层次

1.2.3 构成应用程序的文件

通过上述分析，可知 AppWizard 产生的 MDI 框架程序的内容，所定义和实现的类。下面，从文件的角度来考察 AppWizard 生成了哪些源码文件，这些文件的作用是什么。表 1-1 列出了 AppWizard 所生成的头文件，表 1-2 列出了了 AppWizard 所生成的实现文件及其对头文

件的包含关系。

表1-1 AppWizard所生成的头文件

头文件	用途
stdafx.h	标准 AFX 头文件
resource.h	定义了各种资源 ID
t.h	#include "resource.h" 定义了从 CWinApp 派生的应用程序对象 CTApp
childfrm.h	定义了从 CMDIChildWnd 派生的文档框架窗口对象 CTChildFrame
mainfrm.h	定义了从 CMDIFrameWnd 派生的框架窗口对象 CMainFrame
tdoc.h	定义了从 CDocument 派生的文档对象 CTDoc
tview.h	定义了从 CView 派生的视图对象 CTView

表1-2 AppWizard所生成的实现文件

实现文件	所包含的头文件	实现的内容和功能
stdafx.cpp	#include "stdafx.h"	用来产生预编译的类型信息。
t.cpp	# include "stdafx.h" # include "t.h" # include "MainFrm.h" # include "childfrm.h" #include "tdoc.h" #include "tview.h"	定义 CTApp 的实现，并定义 CTApp 类型的全局变量 theApp。
childfrm.cpp	#inlcude "stdafx.h" #include "t.h" #include “childfrm.h”	实现了类 CChildFrame
childfrm.cpp	#inlcude "stdafx.h" #include "t.h" #include "childfrm.h"	实现了类 CMainFrame
tdoc.cpp	# include "stdafx.h" # include "t.h" # include "tdoc.h"	实现了类 CTDoc
tview.cpp	# include "stdafx.h" # include "t.h" # include "tdoc.h" # include "tview.h"	实现了类 CTview

从表 1-2 中的包含关系一栏可以看出：

CTApp 的实现用到所有的用户定义对象，包含了他们的定义；CView 的实现用到 CTdoc；其他对象的实现只涉及自己的定义；

当然，如果增加其他操作，引用其他对象，则要包含相应的类的定义文件。

对预编译头文件说明如下：

所谓头文件预编译，就是把一个工程(Project)中使用的一些 MFC 标准头文件(如 Windows.H、Afxwin.H)预先编译，以后该工程编译时，不再编译这部分头文件，仅仅使用预编译的结果。这样可以加快编译速度，节省时间。

预编译头文件通过编译 stdafx.cpp 生成，以工程名命名，由于预编译的头文件的后缀是“pch”，所以编译结果文件是 projectname.pch。

编译器通过一个头文件 stdafx.h 来使用预编译头文件。stdafx.h 这个头文件名是可以在 project 的编译设置里指定的。编译器认为，所有在指令#include "stdafx.h"前的代码都是预编译的，它跳过#include "stdafx. h"指令，使用 projectname.pch 编译这条指令之后的所有代码。

因此，所有的 CPP 实现文件第一条语句都是：#include "stdafx.h"。

另外，每一个实现文件 CPP 都包含了如下语句：

```
#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif
```

这是表示，如果生成调试版本，要指示当前文件的名称。__FILE__是一个宏，在编译器编译过程中给它赋值为当前正在编译的文件名称。

第2章　MFC 和 Win32

2.1　MFC Object 和 Windows Object 的关系

MFC 中最重要的封装是对 Win32 API 的封装，因此，理解 Windows Object 和 MFC Object (C++对象，一个 C++类的实例)之间的关系是理解 MFC 的关键之一。所谓 Windows Object（Windows 对象）是 Win32 下用句柄表示的 Windows 操作系统对象；所谓 MFC Object (MFC 对象)是 C++对象，是一个 C++类的实例，这里（本书范围内）MFC Object 是有特定含义的，指封装 Windows Object 的 C++ Object，并非指任意的 C++ Object。

MFC Object 和 Windows Object 是不一样的，但两者紧密联系。以窗口对象为例：

一个 MFC 窗口对象是一个 C++ CWnd 类（或派生类）的实例，是程序直接创建的。在程序执行中它随着窗口类构造函数的调用而生成，随着析构函数的调用而消失。而 Windows 窗口则是 Windows 系统的一个内部数据结构的实例，由一个“窗口句柄”标识，Windows 系统创建它并给它分配系统资源。Windows 窗口在 MFC 窗口对象创建之后，由 CWnd 类的 Create 成员函数创建，“窗口句柄”保存在窗口对象的 m_hWnd 成员变量中。Windows 窗口可以被一个程序销毁，也可以被用户的动作销毁。MFC 窗口对象和 Windows 窗口对象的关系如图 2-1 所示。其他的 Windows Object 和对应的 MFC Object 也有类似的关系。

图 2-1 MFC 窗口对象和 Windows 窗口对象的关系

下面，对 MFC Object 和 Windows Object 作一个比较。有些论断对设备描述表(MFC 类是 CDC，句柄是 HDC)可能不适用，但具体涉及到时会指出。

（1）　从数据结构上比较

MFC Object 是相应 C++类的实例，这些类是 MFC 或者程序员定义的；

Windows Object 是 Windows 系统的内部结构，通过一个句柄来引用；

MFC 给这些类定义了一个成员变量来保存 MFC Object 对应的 Windows Object 的句柄。对于设备描述表 CDC 类，将保存两个 HDC 句柄。

（2）　从层次上讲比较

MFC Object 是高层的，Windows Object 是低层的；

MFC Object 封装了 Windows Object 的大部分或全部功能，MFC Object 的使用者不需要直接应用 Windows Object 的 HANDLE（句柄）使用 Win32 API，代替它的是引用相应的 MFC Object 的成员函数。

（3） 从创建上比较

MFC Object 通过构造函数由程序直接创建；Windows Object 由相应的 SDK 函数创建。

MFC 中，使用这些 MFC Object，一般分两步：

首先，创建一个 MFC Object，或者在 STACK 中创建，或者在 HEAP 中创建，这时，MFC Object 的句柄实例变量为空，或者说不是一个有效的句柄。

然后，调用 MFC Object 的成员函数创建相应的 Windows Object，MFC 的句柄变量存储一个有效句柄。

CDC(设备描述表类)的创建有所不同，在后面的 2.3 节会具体说明 CDC 及其派生类的创建和使用。

当然，可以在 MFC Object 的构造函数中创建相应的 Windows 对象，MFC 的 GDI 类就是如此实现的，但从实质上讲，MFC Object 的创建和 Windows Object 的创建是两回事。

（4） 从转换上比较

可以从一个 MFC Object 得到对应的 Windows Object 的句柄；一般使用 MFC Object 的成员函数 GetSafeHandle 得到对应的句柄。

可以从一个已存在的 Windows Object 创建一个对应的 MFC Object； 一般使用 MFC Object 的成员函数 Attach 或者 FromHandle 来创建，前者得到一个永久性对象，后者得到的可能是一个临时对象。

（5） 从使用范围上比较

MFC Object 对系统的其他进程来说是不可见、不可用的；而 Windows Object 一旦创建，其句柄是整个 Windows 系统全局的。一些句柄可以被其他进程使用。典型地，一个进程可以获得另一进程的窗口句柄，并给该窗口发送消息。

对同一个进程的线程来说，只可以使用本线程创建的 MFC Object，不能使用其他线程的 MFC Object。

（6） 从销毁上比较

MFC Object 随着析构函数的调用而消失；但 Windows Object 必须由相应的 Windows 系统函数销毁。

设备描述表 CDC 类的对象有所不同，它对应的 HDC 句柄对象可能不是被销毁，而是被释放。

当然，可以在 MFC Object 的析构函数中完成 Windows Object 的销毁，MFC Object 的 GDI 类等就是如此实现的，但是，应该看到：两者的销毁是不同的。

每类 Windows Object 都有对应的 MFC Object，下面用表格的形式列出它们之间的对应关系，如表 2-1 所示：

表2-1 MFC Object和Windows Object的对应关系

描述	Windows 句柄	MFC Object
窗口	HWND	CWnd and CWnd-derived classes
设备上下文	HDC	CDC and CDC-derived classes
菜单	HMENU	CMenu
笔	HPEN	CGdiObject 类，CPen 和 CPen-derived classes
刷子	HBRUSH	CGdiObject 类，CBrush 和 CBrush-derived classes
字体	HFONT	CGdiObject 类，CFont 和 CFont-derived classes
位图	HBITMAP	CGdiObject 类，CBitmap 和 CBitmap-derived classes

调色板	HPALETTE	CGdiObject 类，CPalette 和 CPalette-derived classes
区域	HRGN	CGdiObject 类，CRgn 和 CRgn-derived classes
图像列表	HIMAGELIST	CImageList 和 CImageList-derived classes
套接字	SOCKET	CSocket,CAsynSocket 及其派生类

表 2-1 中的 OBJECT 分以下几类：
Windows 对象，
设备上下文对象，
GDI 对象（BITMAP，BRUSH，FONT，PALETTE，PEN，RGN），
菜单，
图像列表，
网络套接字接口。
从广义上来看，文档对象和文件可以看作一对 MFC Object 和 Windows Object，分别用 CDocument 类和文件句柄描述。
后续几节分别对前四类作一个简明扼要的论述。

2.2 Windows Object

用 SDK 的 Win32 API 编写各种 Windows 应用程序，有其共同的规律：首先是编写 WinMain 函数，编写处理消息和事件的窗口过程 WndProc，在 WinMain 里头注册窗口（Register Window），创建窗口，然后开始应用程序的消息循环。
MFC 应用程序也不例外，因为 MFC 是一个建立在 SDK API 基础上的编程框架。对程序员来说所不同的是：一般情况下，MFC 框架自动完成了 Windows 登记、创建等工作。
下面，简要介绍 MFC Window 对 Windows Window 的封装。

2.2.1 Windows 的注册

一个应用程序在创建某个类型的窗口前，必须首先注册该“窗口类”(Windows Class)。注意，这里不是 C++类的类。Register Window 把窗口过程、窗口类型以及其他类型信息和要登记的窗口类关联起来。

（1） “窗口类”的数据结构

“窗口类”是 Windows 系统的数据结构，可以把它理解为 Windows 系统的类型定义，而 Windows 窗口则是相应“窗口类”的实例。Windows 使用一个结构来描述“窗口类”，其定义如下：

```
typedef struct _WNDCLASSEX {
    UINT       cbSize; //该结构的字节数
    UINT       style;   //窗口类的风格
    WNDPROC lpfnWndProc; //窗口过程
    int        cbClsExtra;
    int        cbWndExtra;
    HANDLE    hInstance; //该窗口类的窗口过程所属的应用实例
    HICON     hIcon; //该窗口类所用的像标
```

```
    HCURSOR hCursor; //该窗口类所用的光标
    HBRUSH    hbrBackground; //该窗口类所用的背景刷
    LPCTSTR lpszMenuName; //该窗口类所用的菜单资源
    LPCTSTR lpszClassName; //该窗口类的名称
    HICON     hIconSm; //该窗口类所用的小像标
} WNDCLASSEX;
```

从“窗口类”的定义可以看出，它包含了一个窗口的重要信息，如窗口风格、窗口过程、显示和绘制窗口所需要的信息，等等。关于窗口过程，将在后面消息映射等有关章节作详细论述。

Windows 系统在初始化时，会注册(Register)一些全局的“窗口类”，例如通用控制窗口类。应用程序在创建自己的窗口时，首先必须注册自己的窗口类。在 MFC 环境下，有几种方法可以用来注册“窗口类”，下面分别予以讨论。

（2） 调用 AfxRegisterClass 注册

AfxRegisterClass 函数是 MFC 全局函数。AfxRegisterClass 的函数原型：

BOOL AFXAPI AfxRegisterClass(WNDCLASS *lpWndClass);

参数 lpWndClass 是指向 WNDCLASS 结构的指针，表示一个“窗口类”。

首先，AfxRegisterClass 检查希望注册的“窗口类”是否已经注册，如果是则表示已注册，返回 TRUE，否则，继续处理。

接着，调用::RegisterClass(lpWndClass)注册窗口类；

然后，如果当前模块是 DLL 模块，则把注册“窗口类”的名字加入到模块状态的域 m_szUnregisterList 中。该域是一个固定长度的缓冲区，依次存放模块注册的“窗口类”的名字（每个名字是以“\n\0”结尾的字符串）。之所以这样做，是为了 DLL 退出时能自动取消(Unregister)它注册的窗口类。至于模块状态将在后面第 9 章详细的讨论。

最后，返回 TRUE 表示成功注册。

（3） 调用 AfxRegisterWndClass 注册

AfxRegisterWndClass 函数也是 MFC 全局函数。AfxRegisterWndClass 的函数原型：

```
LPCTSTR AFXAPI AfxRegisterWndClass(UINT nClassStyle,
    HCURSOR hCursor, HBRUSH hbrBackground, HICON hIcon)
```

参数 1 指定窗口类风格；

参数 2、3、4 分别指定该窗口类使用的光标、背景刷、像标的句柄，缺省值是 0。

此函数根据窗口类属性动态地产生窗口类的名字，然后，判断是否该类已经注册，是则返回窗口类名；否则用指定窗口类的属性（窗口过程指定为缺省窗口过程），调用 AfxRegisterCalss 注册窗口类，返回类名。

动态产生的窗口类名字由以下几部分组成（包括冒号分隔符）：

如果参数 2、3、4 全部为 NULL，则由三部分组成。

“Afx”+“:”+模块实例句柄”+“:”+“窗口类风格”

否则，由六部分组成：

“Afx”+“:”+模块实例句柄+“:”+“窗口类风格”+“:”+光标句柄+“:”+背景刷句柄+“:”+像标句柄。比如：“Afx:400000:b:13de:6:32cf”。

该函数在 MFC 注册主边框或者文档边框“窗口类”时被调用。具体怎样用在 5.3.3.3 节会指出。

（4） 隐含的使用 MFC 预定义的的窗口类

MFC4.0 以前的版本提供了一些预定义的窗口类，4.0 以后不再预定义这些窗口类。但是，MFC 仍然沿用了这些窗口类，例如：

用于子窗口的“AfxWnd”；
用于边框窗口(SDI 主窗口或 MDI 子窗口)或视的“AfxFrameOrView”；
用于 MDI 主窗口的“AfxMDIFrame”；
用于标准控制条的“AfxControlBar”。
这些类的名字就 是“AfxWnd”、“AfxFrameOrView”、“AfxMdiFrame”、 “AfxControlBar”加上前缀和后缀（用来标识版本号或是否调试版等）。它们使用标准应用程序像标、标准文档像标、标准光标等标准资源。为了使用这些“窗口类”，MFC 会在适当的时候注册这些类：或者要创建该类的窗口时，或者创建应用程序的主窗口时，等等。
MFC 内部使用了函数
BOOL AFXAPI AfxEndDeferRegisterClass(short fClass)
来帮助注册上述原 MFC 版本的预定义“窗口类”。参数 fClass 区分了那些预定义窗口的类型。根据不同的类型，使用不同的窗口类风格、窗口类名字等填充 WndClass 的域，然后调用 AfxRegisterClass 注册窗口类。并且注册成功之后，通过模块状态的 m_fRegisteredClasses 记录该窗口类已经注册，这样该模块在再次需要注册这些窗口类之前可以查一下 m_fRegisteredClasses，如果已经注册就不必浪费时间了。为此，MFC 内部使用宏
AfxDeferRegisterClass(short fClass)
来注册“窗口类”，如果 m_fRegisteredClasses 记录了注册的窗口类，返回 TRUE，否则，调用 AfxEndDeferRegisterClass 注册。
注册这些窗口类的例子：
MFC 在加载边框窗口时，会自动地注册“AfxFrameOrView”窗口类。在创建视时，就会使用该“窗口类”创建视窗口。当然，如果创建视窗口时，该“窗口类”还没有注册，MFC 将先注册它然后使用它创建视窗口。
不过，MFC 并不使用”AfxMDIFrame”来创建 MDI 主窗口，因为在加载主窗口时一般都指定了主窗口的资源，MFC 使用指定的像标注册新的 MDI 主窗口类（通过函数 AfxRegisterWndClass 完成，因此“窗口类”的名字是动态产生的）。
MDI 子窗口类似于上述 MDI 主窗口的处理。
在 MFC 创建控制窗口时，如工具栏窗口，如果“AfxControlBar”类还没有注册，则注册它。注册过程很简单，就是调用::InitCommonControl 加载通用控制动态连接库。

（5） 调用::RegisterWndClass。

直接调用 Win32 的窗口注册函数::RegisterWndClass 注册“窗口类”，这样做有一个缺点：如果是 DLL 模块，这样注册的“窗口类”在程序退出时不会自动的被取消注册(Unregister)。所以必须记得在 DLL 模块退出时取消它所注册的窗口类。

（6） 子类化

子类化（Subclass）一个“窗口类”，可自动地得到它的“窗口类”属性。

2.2.2 MFC 窗口类 CWnd

在 Windows 系统里，一个窗口的属性分两个地方存放：一部分放在“窗口类”里头，如上所述的在注册窗口时指定；另一部分放在 Windows Object 本身，如：窗口的尺寸，窗口的位置（X，Y 轴），窗口的 Z 轴顺序，窗口的状态（ACTIVE，MINIMIZED，MAXMIZED，RESTORED…），和其他窗口的关系（父窗口，子窗口…），窗口是否可以接收键盘或鼠标消息，等等。
为了表达所有这些窗口的共性，MFC 设计了一个窗口基类 CWnd。有一点非常重要，那就是 CWnd 提供了一个标准而通用的 MFC 窗口过程，MFC 下所有的窗口都使用这个窗口过

程。至于通用的窗口过程却能为各个窗口实现不同的操作，那就是 MFC 消息映射机制的奥秘和作用了。这些，将在后面有关章节详细论述。

CWnd 提供了一系列成员函数，或者是对 Win32 相关函数的封装，或者是 CWnd 新设计的一些函数。这些函数大致如下。

（1）窗口创建函数

这里主要讨论函数 Create 和 CreateEx。它们封装了 Win32 窗口创建函数::CreateWindowEx。Create 的原型如下：

```
BOOL CWnd::Create(LPCTSTR lpszClassName,
    LPCTSTR lpszWindowName, DWORD dwStyle,
    const RECT& rect,
    CWnd* pParentWnd, UINT nID,
    CCreateContext* pContext)
```

Create 是一个虚拟函数，用来创建子窗口（不能创建桌面窗口和 POP UP 窗口）。CWnd 的基类可以覆盖该函数，例如边框窗口类等覆盖了该函数以实现边框窗口的创建，视类则使用它来创建视窗口。

Create 调用了成员函数 CreateEx。CWnd::CreateEx 的原型如下：

```
BOOL CWnd::CreateEx(DWORD dwExStyle, LPCTSTR lpszClassName,
    LPCTSTR lpszWindowName, DWORD dwStyle,
    int x, int y, int nWidth, int nHeight,
    HWND hWndParent, HMENU nIDorHMenu, LPVOID lpParam)
```

CreateEx 有 11 个参数，它将调用::CreateWindowEx 完成窗口的创建，这 11 个参数对应地传递给::CreateWindowEx。参数指定了窗口扩展风格、“窗口类”、窗口名、窗口大小和位置、父窗口句柄、窗口菜单和窗口创建参数。

CreateEx 的处理流程将在后面 4.4.1 节讨论窗口过程时分析。

窗口创建时发送 WM_CREATE 消息，消息参数 lParam 指向一个 CreateStruct 结构的变量，该结构有 11 个域，其描述见后面 4.4.1 节对窗口过程的分析，Windows 使用和 CreateEx 参数一样的内容填充该变量。

（2）窗口销毁函数

例如：

DestroyWindow 函数 销毁窗口

PostNcDestroy( )，销毁窗口后调用，虚拟函数

（3）用于设定、获取、改变窗口属性的函数，例如：

SetWindowText(CString tiltle) 设置窗口标题

GetWindowText() 得到窗口标题

SetIcon(HICON hIcon, BOOL bBigIcon)；设置窗口像标

GetIcon(BOOL bBigIcon) ;得到窗口像标

GetDlgItem(int nID)；得到窗口类指定 ID 的控制子窗口

GetDC(); 得到窗口的设备上下文

SetMenu(CMenu *pMenu); 设置窗口菜单

GetMenu()；得到窗口菜单

…

（4）用于完成窗口动作的函数

用于更新窗口，滚动窗口，等等。一部分成员函数设计成或可重载(Overloaded）函数，或虚拟(Overridden)函数，或 MFC 消息处理函数。这些函数或者实现了一部分功能，或者仅仅是一个空函数。如：

- 有关消息发送的函数：

SendMessage(UINT message,WPARAM wParam = 0, LPARAM lParam = 0);

给窗口发送发送消息，立即调用方式

PostMessage((UINT message,WPARAM wParam = 0, LPARAM lParam = 0);

给窗口发送消息，放进消息队列

…

- 有关改变窗口状态的函数

MoveWindow(LPCRECT lpRect, BOOL bRepaint = TRUE);

移动窗口到指定位置

ShowWindow(BOOL)；显示窗口，使之可见或不可见

….

- 实现 MFC 消息处理机制的函数：

virtual LRESULT WindowProc(UINT message, WPARAM wParam, LPARAM lParam)；窗口过程，虚拟函数

virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam)；处理命令消息

…

- 消息处理函数：

OnCreate(LPCREATESTRUCT lpCreateStruct);MFC 窗口消息处理函数，窗口创建时由 MFC 框架调用

OnClose();MFC 窗口消息处理函数，窗口创建时由 MFC 框架调用

…

- 其他功能的函数

CWnd 的导出类是类型更具体、功能更完善的窗口类，它们继承了 CWnd 的属性和方法，并提供了新的成员函数（消息处理函数、虚拟函数、等等）。

常用的窗口类及其层次关系见图 1-1。

2.2.3 在 MFC 下创建一个窗口对象

MFC 下创建一个窗口对象分两步，首先创建 MFC 窗口对象，然后创建对应的 Windows 窗口。在内存使用上，MFC 窗口对象可以在栈或者堆(使用 new 创建)中创建。具体表述如下：

- 创建 MFC 窗口对象。通过定义一个 CWnd 或其派生类的实例变量或者动态创建一个 MFC 窗口的实例，前者在栈空间创建一个 MFC 窗口对象，后者在堆空间创建一个 MFC 窗口对象。
- 调用相应的窗口创建函数，创建 Windows 窗口对象。

例如：在前面提到的 AppWizard 产生的源码中，有 CMainFrame（派生于 CMDIFrame(SDI)或者 CMDIFrameWnd(MDI)）类。它有两个成员变量定义如下：

```
CToolBar m_wndToolBar;
CStatusBar m_wndStatusBar;
```

当创建 CMainFrame 类对象时，上面两个 MFC Object 也被构造。

CMainFrame 还有一个成员函数

OnCreate（LPCREATESTRUCT lpCreateStruct)，

它的实现包含如下一段代码，调用 CToolBar 和 CStatusBar 的成员函数 Create 来创建上述两个 MFC 对象对应的工具栏 HWND 窗口和状态栏 HWND 窗口：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
…
    if (!m_wndToolBar.Create(this) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;          // fail to create
    }

    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
          sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Failed to create status bar\n");
        return -1;          // fail to create
    }
…
}
```

关于工具栏、状态栏将在后续有关章节作详细讨论。

在 MFC 中，还提供了一种动态创建技术。动态创建的过程实际上也如上所述分两步，只不过 MFC 使用这个技术是由框架自动地完成整个过程的。通常框架窗口、文档框架窗口、视使用了动态创建。介于 MFC 的结构，CFrameWnd 和 CView 及其派生类的实例即使不使用动态创建，也要用 new 在堆中分配。理由见窗口的销毁（2.2.5 节）。

至于动态创建技术，将在下一章具体讨论。

在 Windows 窗口的创建过程中，将发送一些消息，如：

在创建了窗口的非客户区(Nonclient area)之后，发送消息 WM_NCCREATE；

在创建了窗口的客户区（client area）之后，发送消息 WM_CREATE；

窗口的窗口过程在窗口显示之前收到这两个消息。

如果是子窗口，在发送了上述两个消息之后，还给父窗口发送 WM_PARENATNOTIFY 消息。

其他类或风格的窗口可能发送更多的消息，具体参见 SDK 开发文档。

2.2.4 MFC 窗口的使用

MFC 提供了大量的窗口类，其功能和用途各异。程序员应该选择哪些类来使用，以及怎么使用他们呢？

直接使用 MFC 提供的窗口类或者先从 MFC 窗口类派生一个新的 C++类然后使用它，这些在通常情况下都不需要程序员提供窗口注册的代码。是否需要派生新的 C++类，视 MFC 已有的窗口类是否能满足使用要求而定。派生的 C++类继承了基类的特性并改变或扩展了它的

功能，例如增加或者改变对消息、事件的特殊处理等。
主要使用或继承以下一些 MFC 窗口类（其层次关系图见图 1-1）：
框架类 CFrameWnd，CMdiFrameWnd；
文档框架 CMdiChildWnd；
视图 CView 和 CView 派生的有特殊功能的视图如：列表 CListView，编辑 CEditView，树形列表 CTreeView，支持 RTF 的 CRichEditView，基于对话框的视 CFormView 等等。
对话框 CDialog。
通常，都要从这些类派生应用程序的框架窗口和视窗口或者对话框。

工具条 CToolBar
状态条 CStatusBar
其他各类控制窗口，如列表框 CList，编辑框 CEdit，组合框 CComboBox，按钮 Cbutton 等。
通常，直接使用这些类。

2.2.5 在 MFC 下窗口的销毁

窗口对象使用完毕，应该销毁。在 MFC 下，一个窗口对象的销毁包括 HWND 窗口对象的销毁和 MFC 窗口对象的销毁。一般情况下，MFC 编程框架自动地处理了这些。

（1）对 CFrameWnd 和 CView 的派生类

这些窗口的关闭导致销毁窗口的函数 DestroyWindow 被调用。销毁 Windows 窗口时，MFC 框架调用的最后一个成员函数是 OnNcDestroy 函数，该函数负责 Windows 清理工作，并在最后调用虚拟成员函数 PostNcDestroy。CFrameWnd 和 CView 的 PostNcDestroy 调用 delete this 删除自身这个 MFC 窗口对象。
所以，对这些窗口，如前所述，应在堆（Heap）中分配，而且，不要对这些对象使用 delete 操作。

（2）对 Windows Control 窗口

在它们的析构函数中，将调用 DestroyWidnow 来销毁窗口。如果在栈中分配这样的窗口对象，则在超出作用范围的时候，随着析构函数的调用，MFC 窗口对象和它的 Windows window 对象都被销毁。如果在堆（Heap）中分配，则显式调用 delete 操作符，导致析构函数的调用和窗口的销毁。
所以，这种类型的窗口应尽可能在栈中分配，避免用额外的代码来销毁窗口。如前所述的 CMainFrame 的成员变量 m_wndStatusBar 和 m_wndToolBar 就是这样的例子。

（3）对于程序员直接从 CWnd 派生的窗口

程序员可以在派生类中实现上述两种机制之一，然后，在相应的规范下使用。
后面章节将详细的讨论应用程序退出时关闭、清理窗口的过程。

2.3 设备描述表

2.3.1 设备描述表概述

当一个应用程序使用 GDI 函数时，必须先装入特定的设备驱动程序，然后为绘制窗口准备设备描述表，比如指定线的宽度和颜色、刷子的样式和颜色、字体、剪裁区域等等。不像其他 Win32 结构，设备描述表不能被直接访问，只能通过系列 Win32 函数来间接地操作。
如同 Windows“窗口类”一样，设备描述表也是一种 Windows 数据结构，用来描述绘制窗口所需要的信息。它定义了一个坐标映射模式、一组 GDI 图形对象及其属性。这些 GDI 对象包括用于画线的笔，绘图、填图的刷子，位图，调色板，剪裁区域，及路径(Path)。
表 2-2 列出了设备描述表的结构和各项缺省值，表 2-3 列出了设备描述表的类型，表 2-4 显示设备描述表的类型。

表2-2 设备描述表的结构

属性	缺省值
Background color	Background color setting from Windows Control Panel (typically, white)
Background mode	OPAQUE
Bitmap	None
Brush	WHITE_BRUSH
Brush origin	(0,0)
Clipping region	Entire window or client area with the update region clipped, as appropriate. Child and pop-up windows in the client area may also be clipped
Palette	DEFAULT_PALETTE
Current pen position	(0,0)
Device origin	Upper left corner of the window or the client area
Drawing mode	R2_COPYPEN
Font	SYSTEM_FONT (SYSTEM_FIXED_FONT for applications written to run with Windows versions 3.0 and earlier)
Intercharacter spacing	0
Mapping mode	MM_TEXT
Pen	BLACK_PEN
Polygon-fill mode	ALTERNATE
Stretch mode	BLACKONWHITE
Text color	Text color setting from Control Panel (typically, black)
Viewport extent	(1,1)
Viewport origin	(0,0)
Window extent	(1,1)
Window origin	(0,0)

表2-3 设备描述表的分类

Display	显示设备描述表，提供对视频显示设备上的绘制操作的支持
Printer	打印设备描述表，提供对打印机、绘图仪设备上的绘制操作的支持
Memory	内存设备描述表，提供对位图操作的支持
Information	信息设备描述表，提供对操作设备信息获取的支持

表 2-3 中的显示设备描述表又分三种类型，如表 2-4 所示。

表2-4 显示设备描述表的分类

名称	特点	功能
Class Device Contexts	提供对 Win16 的向后兼容	
Common Device Contexts	在 Windows 系统的高速缓冲区，数量有限	Applicaion 获取设备描述表时，Windows 用缺省值初始化该设备描述表，Application 使用它完成绘制操作，然后释放
Private Device Contexts	没有数量限制，用完不需释放一次获取，多次使用	多次使用过程中，每次设备描述表属性的任何修改或变化都会被保存，以支持快速绘制

（1）使用设备描述表的步骤

要使用设备描述表，一般有如下步骤：

- 获取或者创建设备描述表；
- 必要的话，改变设备描述表的属性；
- 使用设备描述表完成绘制操作；
- 释放或删除设备描述表。

Common 设备描述表通过::GetDC，::GetDCEx，::BeginPaint 来获得一个设备描述表，用毕，用::ReleaseDC 或::EndPaint 释放设备描述表；

Printer 设备描述表通过::CreateDC 创建设备描述表，用::DeleteDC 删除设备描述表。

Memory 设备描述表通过::CreateCompatibleDC 创建设备描述表，用::DeleteDC 删除。

Information 设备描述表通过::CreateIC 创建设备描述表，用::DeleteDC 删除。

（2）改变设备描述表属性的途径

要改变设备描述表的属性，可通过以下途径：

用::SelectObject 选入新的除调色板以外的 GDI Object 到设备描述表中；

对于调色板，使用::SelectPalette 函数选入逻辑调色板，并使用::RealizePalette 把逻辑调色板的入口映射到物理调色板中。

用其他 API 函数改变其他属性，如::SetMapMode 改变映射模式。

2.3.2 设备描述表在 MFC 中的实现

MFC 提供了 CDC 类作为设备描述表类的基类，它封装了 Windows 的 HDC 设备描述表对象和相关函数。

（1） CDC 类

CDC 类包含了各种类型的 Windows 设备描述表的全部功能，封装了所有的 Win32 GDI 函数和设备描述表相关的 SDK 函数。在 MFC 下，使用 CDC 的成员函数来完成所有的窗口绘

制工作。
CDC 类的结构示意图 2-2 所示。

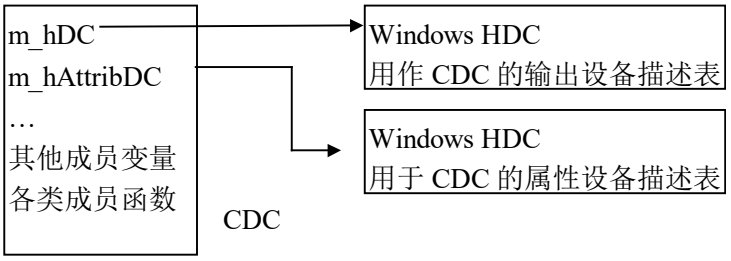


图 2-2　CDC 类和 Windows DC 的关系

CDC 类有两个成员变量：m_hDC，m_hAttribDC，它们都是 Windows 设备描述表句柄。CDC 的成员函数作输出操作时，使用 m_Hdc；要获取设备描述表的属性时，使用 m_hAttribDC。在创建一个 CDC 类实例时，缺省的 m_hDC 等于 m_hAttribDC。如果需要的话，程序员可以分别指定它们。例如，MFC 框架实现 CMetaFileDC 类时，就是如此：CMetaFileDC 从物理设备上读取设备信息，输出则送到元文件（metafile）上，所以 m_hDC 和 m_hAttribDC 是不同的，各司其责。还有一个类似的例子：打印预览的实现，一个代表打印机模拟输出，一个代表屏幕显示。
CDC 封装::SelectObject(HDC hdc，HGDIOBJECT hgdiobject)函数时，采用了重载技术，即它针对不同的 GDI 对象，提供了名同而参数不同的成员函数：
SelectObject(CPen *pen)用于选入笔；
SelectObject(CBitmap* pBitmap)用于选入位图；
SelectObject(CRgn *pRgn)用于选入剪裁区域；
SelectObject(CBrush *pBrush)用于选入刷子；
SelectObject(CFont *pFont)用于选入字体；
至于调色板，使用 SelectPalette(CPalette *pPalette，BOOL bForceBackground)选入调色板到设备描述表，使用 RealizePalletter()实现逻辑调色板到物理调色板的映射。

（2） 从 CDC 派生出功能更具体的设备描述表

从 CDC 派生出四个功能更具体的设备描述表类。层次如图 2-3 所示。

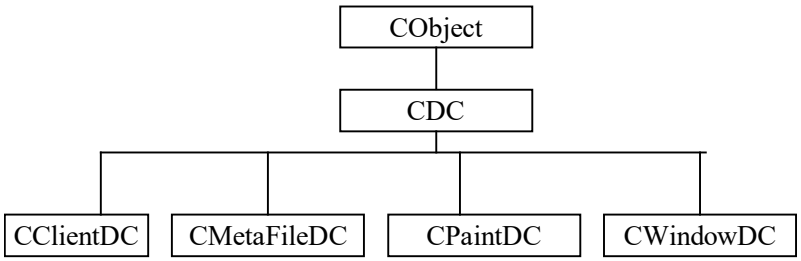


图 2-3　设备描述表类的层次

下面，分别讨论派生出的四种设备描述表。

- CCientDC

代表窗口客户区的设备描述表。其构造函数 CClientDC(CWnd *pWin)通过::GetDC 获取指定窗口的客户区的设备描述表 HDC，并且使用成员函数 Attach 把它和 CClientDC 对象捆绑在

一起；其析构函数使用成员函数 Detach 把设备描述表句柄 HDC 分离出来，并调用::ReleaseDC 释放设备描述表 HDC。

- CPaintDC

仅仅用于响应 WM_PAINT 消息时绘制窗口，因为它的构造函数调用了::BeginPaint 获取设备描述表 HDC，并且使用成员函数 Attach 把它和 CPaintDC 对象捆绑在一起；析构函数使用成员函数 Detach 把设备描述表句柄 HDC 分离出来，并调用::EndPaint 释放设备描述表 HDC，而::BeginPaint 和::EndPaint 仅仅在响应 WM_PAINT 时使用。

- CMetaFileDC

用于生成元文件。

- CWindowDC

代表整个窗口区(包括非客户区)的设备描述表。其构造函数 CWindowDC(CWnd *pWin)通过::GetWindowDC 获取指定窗口的客户区的设备描述表 HDC，并使用 Attach 把它和 CWindowDC 对象捆绑在一起；其析构函数使用 Detach 把设备描述表 HDC 分离出来，调用::ReleaseDC 释放设备描述表 HDC。

2.3.3 MFC 设备描述表类的使用

（1） 使用 CPaintDC、CClientDC、CWindowDC 的方法

首先，定义一个这些类的实例变量，通常在栈中定义。然后，使用它。

例如，MFC 中 CView 对 WM_PAINT 消息的实现方法如下：

```
void CView::OnPaint()
{
    // standard paint routine
    CPaintDC dc(this);
    OnPrepareDC(&dc);
    OnDraw(&dc);
}
```

在栈中定义了 CPaintDC 类型的变量 dc，随着构造函数的调用获取了设备描述表；设备描述表使用完毕，超出其有效范围就被自动地清除，随着析构函数的调用，其获取的设备描述表被释放。

如果希望在堆中创建，例如

```
CPaintDC *pDC;
pDC = new CPaintDC(this)
```

则在使用完毕时，用 delete 删除 pDC:

```
delete pDC;
```

（2） 直接使用 CDC

需要注意的是：在生成 CDC 对象的时候，并不像它的派生类那样，在构造函数里获取相应的 Windows 设备描述表。最好不要使用::GetDC 等函数来获取一个设备描述表，而是创建一个设备描述表。其构造函数如下：

```
CDC::CDC()
{
    m_hDC = NULL;
    m_hAttribDC = NULL;
```

```
        m_bPrinting = FALSE;
    }
```

其析构函数如下：

```
    CDC::~CDC()
    {
        if (m_hDC != NULL)
        ::DeleteDC(Detach());
    }
```

在 CDC 析构函数中，如果设备描述表句柄不空，则调用 DeleteDC 删除它。这是直接使用 CDC 时最好创建 Windows 设备描述表的理由。如果设备描述表不是创建的，则应该在析构函数被调用前分离出设备描述表句柄并用::RealeaseDC 释放它，释放后 m_hDC 为空，则在析构函数调用时不会执行::DeleteDC。当然，不用担心 CDC 的派生类的析构函数调用 CDC 的析构函数，因为 CDC::~CDC()不是虚拟析构函数。

直接使用 CDC 的例子是内存设备上下文，例如：

```
    CDC dcMem;   //声明一个 CDC 对象
    dcMem.CreateCompatibleDC(&dc); //创建设备描述表
    pbmOld = dcMem.SelectObject(&m_bmBall);//更改设备描述表属性
    …//作一些绘制操作

    dcMem.SelectObject(pbmOld);//恢复设备描述表的属性
    dcMem.DeleteDC();   //可以不调用，而让析构函数去删除设备描述表
```

2.4 GDI 对象

在讨论设备描述表时，已经多次涉及到 GDI 对象。这里，需强调一下：GDI 对象要选入 Windows 设备描述表后才能使用；用毕，要恢复设备描述表的原 GDI 对象，并删除该 GDI 对象。

一般按如下步骤使用 GDI 对象：

```
    Create or get a GDI OBJECT hNewGdi;

    hOldGdi =    ::SelectObject(hdc, hNewGdi)
    ……
    ::SelectObject(hdc, hOldGdi)
    ::DeleteObject(hNewGdi)
```

先创建或得到一个 GDI 对象，然后把它选入设备描述表并保存它原来的 GDI 对象；用毕恢复设备描述表原来的 GDI 对象并删除新创建的 GDI 对象。

需要指出的是，如果 hNewGdi 是一个 Stock GDI 对象，可以不删除（删除也可以）。通过

```
    HGDIOBJ GetStockObject(
        int fnObject     // type of stock object
      );
```

来获取 Stock GDI 对象。

（1） MFC GDI 对象

MFC 用一些类封装了 Windows GDI 对象和相关函数，层次结构如图 2-4 所示：

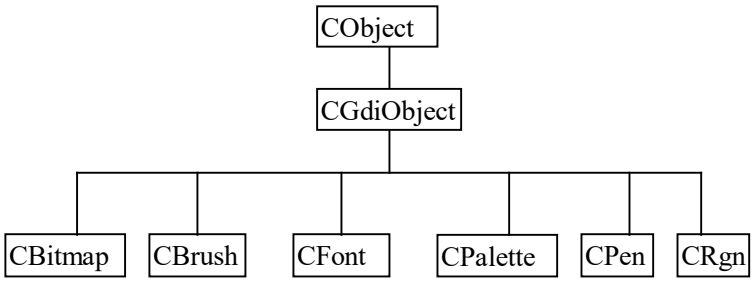


图 2-4　GDI 类的层次

CGdiObject 封装了 Windows GDI Object 共有的特性。其派生类在继承的基础上，主要封装了各类 GDI 的创建函数以及和具体 GDI 对象相关的操作。

CGdiObject 的构造函数仅仅让 m_hObject 为空。如果 m_hObject 不空，其析构函数将删除对应的 Windows GDI 对象。MFC GDI 对象和 Windows GDI 对象的关系如图 2-5 所示。

（2）　使用 MFC GDI 类的使用

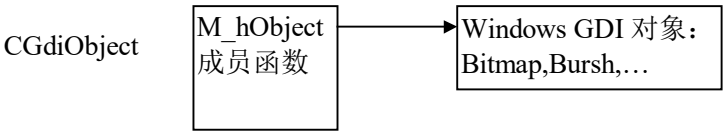


图 2-5　MFC GDI 对象和 Windows GDI 的关系

首先创建 GDI 对象，可分一步或两步创建。一步创建就是构造 MFC 对象和 Windows GDI 对象一步完成；两步创建则先构造 MFC 对象，接着创建 Windows GDI 对象。然后，把新创建的 GDI 对象选进设备描述表，取代原 GDI 对象并保存。最后，恢复原 GDI 对象。例如：

```
void CMyView::OnDraw(CDC *pDC)
{
    CPen penBlack; //构造 MFC CPen 对象
    if (penBlack.CreatePen(PS_SOLID, RGB(0, 0, 0)))
    {
        CPen *pOldPen   = pDC->SelectObject(&penBlack)); //选进设备表，保存原笔
        …
        pDC->SelectObject(pOldPen); //恢复原笔
    }else
    {
        …
    }
}
```

和在 SDK 下有一点不同的是：这里没有 DeleteObject。因为执行完 OnDraw 后，栈中的 penBlack 被销毁，它的析构函数被调用，导致 DeleteObject 的调用。

还有一点要说明：

pDC->SelectObject(&penBlack)返回了一个 CPen *指针，也就是说，它根据原来 PEN 的句柄创建了一个 MFC CPen 对象。这个对象是否需要删除呢？不必要，因为它是一个临时对象，MFC 框架会自动地删除它。当然，在本函数执行完毕把控制权返回给主消息循环之前，该对象是有效的。

关于临时对象及 MFC 处理它们的内部机制，将在后续章节详细讨论。

至此，Windows 编程的核心概念：窗口、GDI 界面（设备描述表、GDI 对象等）已经陈述清楚，特别揭示了 MFC 对这些概念的封装机制，并简明讲述了与这些 Windows Object 对应的 MFC 类的使用方法。还有其他 Windows 概念，可以参见 SDK 开发文档。在 MFC 的实现上，基本上仅仅是对和这些概念相关的 Win32 函数的封装。如果明白了 MFC 的窗口、GDI 界面的封装机制，其他就不难了。

第3章　CObject 类

CObject 是大多数 MFC 类的根类或基类。CObject 类有很多有用的特性：对运行时类信息的支持，对动态创建的支持，对串行化的支持，对象诊断输出，等等。MFC 从 CObject 派生出许多类，具备其中的一个或者多个特性。程序员也可以从 CObject 类派生出自己的类，利用 CObject 类的这些特性。

本章将讨论 MFC 如何设计 CObject 类的这些特性。首先，考察 CObject 类的定义，分析其结构和方法（成员变量和成员函数）对 CObject 特性的支持。然后，讨论 CObject 特性及其实现机制。

3.1　CObject 的结构

以下是 CObject 类的定义：

```
class CObject
{
public:

//与动态创建相关的函数
virtual CRuntimeClass* GetRuntimeClass() const;
析构函数
virtual ~CObject();   // virtual destructors are necessary

//与构造函数相关的内存分配函数，可以用于 DEBUG 下输出诊断信息
void* PASCAL operator new(size_t nSize);
void* PASCAL operator new(size_t, void* p);
void PASCAL operator delete(void* p);
#if defined(_DEBUG) && !defined(_AFX_NO_DEBUG_CRT)
      void* PASCAL operator new(size_t nSize, LPCSTR lpszFileName, int nLine);
#endif

//缺省情况下，复制构造函数和赋值构造函数是不可用的
//如果程序员通过传值或者赋值来传递对象，将得到一个编译错误
protected:
//缺省构造函数
CObject();
private:
//复制构造函数，私有
CObject(const CObject& objectSrc);              // no implementation
//赋值构造函数，私有
void operator=(const CObject& objectSrc);       // no implementation

// Attributes
```

```
public:
//与运行时类信息、串行化相关的函数
BOOL IsSerializable() const;
BOOL IsKindOf(const CRuntimeClass* pClass) const;
// Overridables
virtual void Serialize(CArchive& ar);
//  诊断函数
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;

// Implementation
public:
//与动态创建对象相关的函数
static const AFX_DATA CRuntimeClass classCObject;
#ifdef _AFXDLL
static CRuntimeClass* PASCAL _GetBaseClass();
#endif
};
```

由上可以看出，CObject 定义了一个 CRuntimeClass 类型的静态成员变量：

CRuntimeClass classCObject

还定义了几组函数：

构造函数析构函数类，

诊断函数，

与运行时类信息相关的函数，

与串行化相关的函数。

其中，一个静态函数：_GetBaseClass；五个虚拟函数：析构函数、GetRuntimeClass、Serialize、AssertValid、Dump。这些虚拟函数，在 CObject 的派生类中应该有更具体的实现。必要的话，派生类实现它们时可能要求先调用基类的实现，例如 Serialize 和 Dump 就要求这样。

静态成员变量 classCObject 和相关函数实现了对 CObjet 特性的支持。

3.2 CObject 类的特性

下面，对三种特性分别描述，并说明程序员在派生类中支持这些特性的方法。

（1） 对运行时类信息的支持

该特性用于在运行时确定一个对象是否属于一特定类（是该类的实例），或者从一个特定类派生来的。CObject 提供 IsKindOf 函数来实现这个功能。

从 CObject 派生的类要具有这样的特性，需要：

- 定义该类时，在类说明中使用 DECLARE_DYNAMIC（CLASSNMAE）宏；
- 在类的实现文件中使用 IMPLEMENT_DYNAMIC(CLASSNAME，BASECLASS)宏。

（2） 对动态创建的支持

前面提到了动态创建的概念，就是运行时创建指定类的实例。在 MFC 中大量使用，如前所

述框架窗口对象、视对象，还有文档对象都需要由文档模板类(CDocTemplate)对象来动态的创建。

从 CObject 派生的类要具有动态创建的功能，需要：

- 定义该类时，在类说明中使用 DECLARE_DYNCREATE（CLASSNMAE）宏；
- 定义一个不带参数的构造函数（默认构造函数）；
- 在类的实现文件中使用 IMPLEMENT_DYNCREATE（CLASSNAME，BASECLASS）宏；
- 使用时先通过宏 RUNTIME_CLASS 得到类的 RunTime 信息，然后使用 CRuntimeClass 的成员函数 CreateObject 创建一个该类的实例。

例如：

```
CRuntimeClass* pRuntimeClass = RUNTIME_CLASS(CNname)
//CName 必须有一个缺省构造函数
CObject* pObject = pRuntimeClass->CreateObject();
//用 IsKindOf 检测是否是 CName 类的实例
Assert( pObject->IsKindOf(RUNTIME_CLASS(CName));
```

（3） 对序列化的支持

“序列化”就是把对象内容存入一个文件或从一个文件中读取对象内容的过程。从 CObject 派生的类要具有序列化的功能，需要：

- 定义该类时，在类说明中使用 DECLARE_SERIAL（CLASSNMAE）宏；
- 定义一个不带参数的构造函数（默认构造函数）；
- 在类的实现文件中使用 IMPLEMENT_SERIAL（CLASSNAME，BASECLASS）宏；
- 覆盖 Serialize 成员函数。（如果直接调用 Serialize 函数进行序列化读写，可以省略前面三步。）

对运行时类信息的支持、动态创建的支持、串行化的支持层（不包括直接调用 Serailize 实现序列化），这三种功能的层次依次升高。如果对后面的功能支持，必定对前面的功能支持。支持动态创建的话，必定支持运行时类信息；支持序列化，必定支持前面的两个功能，因为它们的声明和实现都是后者包含前者。

（4） 综合示例：

定义一个支持串行化的类 CPerson：

```
class CPerson : public CObject
{
public:
DECLARE_SERIAL( CPerson )
// 缺省构造函数
CPerson(){};{};

CString m_name;
WORD     m_number;

void Serialize( CArchive& archive );

// rest of class declaration
```

```
};
```

实现该类的成员函数 Serialize，覆盖 CObject 的该函数：

```
void CPerson::Serialize( CArchive& archive )
{
    // 先调用基类函数的实现
    CObject::Serialize( archive );

    // now do the stuff for our specific class
    if( archive.IsStoring() )
        archive << m_name << m_number;
    else
        archive >> m_name >> m_number;
}
```

使用运行时类信息：

```
CPerson a;
ASSERT( a.IsKindOf( RUNTIME_CLASS( CPerson ) ) );
ASSERT( a.IsKindOf( RUNTIME_CLASS( CObject ) ) );
```

动态创建：

```
CRuntimeClass* pRuntimeClass = RUNTIME_CLASS(CPerson)
//Cperson 有一个缺省构造函数
CObject* pObject = pRuntimeClass->CreateObject();
Assert( pObject->IsKindOf(RUNTIME_CLASS(CPerson));
```

3.3 实现 CObject 特性的机制

由上，清楚了 CObject 的结构，也清楚了从 CObject 派生新类时程序员使用 CObject 特性的方法。现在来考察这些方法如何利用 CObjet 的结构，CObject 结构如何支持这些方法。
首先，要揭示 DECLARE_DYNAMIC 等宏的内容，然后，分析这些宏的作用。

3.3.1 DECLARE_DYNAMIC 等宏的定义

MFC 提供了 DECLARE_DYNAMIC、DECLARE_DYNCREATE、DECLARE_SERIAL 声明宏的两种定义，分别用于静态链接到 MFC DLL 和动态链接到 MFC DLL。对应的实现宏 IMPLEMNET_XXXX 也有两种定义，但是，这里实现宏就不列举了。

MFC 对这些宏的定义如下：

```
#ifdef _AFXDLL //动态链接到 MFC DLL
#define DECLARE_DYNAMIC(class_name) \
protected: \
    static CRuntimeClass* PASCAL _GetBaseClass(); \
public: \
```

```
        static const AFX_DATA CRuntimeClass class##class_name; \
        virtual CRuntimeClass* GetRuntimeClass() const; \

#define _DECLARE_DYNAMIC(class_name) \
protected: \
        static CRuntimeClass* PASCAL _GetBaseClass(); \
public: \
        static AFX_DATA CRuntimeClass class##class_name; \
        virtual CRuntimeClass* GetRuntimeClass() const; \

#else
#define DECLARE_DYNAMIC(class_name) \
public: \
        static const AFX_DATA CRuntimeClass class##class_name; \
        virtual CRuntimeClass* GetRuntimeClass() const; \

#define _DECLARE_DYNAMIC(class_name) \
public: \
        static AFX_DATA CRuntimeClass class##class_name; \
        virtual CRuntimeClass* GetRuntimeClass() const; \

#endif

// not serializable, but dynamically constructable
#define DECLARE_DYNCREATE(class_name) \
        DECLARE_DYNAMIC(class_name) \
        static CObject* PASCAL CreateObject();

#define DECLARE_SERIAL(class_name) \
        _DECLARE_DYNCREATE(class_name) \
        friend CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb);
```

由于这些声明宏都是在 CObect 派生类的定义中被使用的，所以从这些宏的上述定义中可以看出，DECLARE_DYNAMIC 宏给所在类添加了一个 CRuntimeClass 类型的静态数据成员 class##class_name(类名加前缀 class，例如，若类名是 CPerson，则该变量名称是 classCPerson)，且指定为 const；两个（使用 MFC DLL 时，否则，一个）成员函数：虚拟函数 GetRuntimeClass 和静态函数_GetBaseClass（使用 MFC DLL 时）。
DECLARE_DYNCREATE 宏包含了 DECLARE_DYNAMIC，在此基础上，还定义了一个静态成员函数 CreateObject。
DECLARE_SERIAL 宏则包含了_DECLARE_DYNCREATE，并重载了操作符“>>”（友员函数）。它和前两个宏有所不同的是 CRuntimeClass 数据成员 class##class_name 没有被指定为 const。

对应地，MFC 使用三个宏初始化 DECLARE 宏所定义的静态变量并实现 DECLARE 宏

所声明的函数：IMPLEMNET_DYNAMIC，IMPLEMNET_DYNCREATE，IMPLEMENT_SERIAL。
首先，这三个宏初始化 CRuntimeClass 类型的静态成员变量 class#class_name。IMPLEMENT_SERIAL 不同于其他两个宏，没有指定该变量为 const。初始化内容在下节讨论 CRuntimeClass 时给出。
其次，它实现了 DECLARE 宏声明的成员函数：

- _GetBaseClass()

返回基类的运行时类信息，即基类的 CRuntimeClass 类型的静态成员变量。这是静态成员函数。

- GetRuntimeClass()

返回类自己的运行类信息，即其 CRuntimeClass 类型的静态成员变量。这是虚拟成员函数。
对于动态创建宏，还有一个静态成员函数 CreateObject，它使用 C++操作符和类的缺省构造函数创建本类的一个动态对象。

- 操作符的重载

对于序列化的实现宏 IMPLEMENT_SERIAL，还重载了操作符<<和定义了一个静态成员变量

```
static const AFX_CLASSINIT _init_##class_name(RUNTIME_CLASS(class_name));
```

比如，对 CPerson 来说，该变量是_init_Cperson，其目的在于静态成员在应用程序启动之前被初始化，使得 AFX_CLASSINIT 类的构造函数被调用，从而通过 AFX_CLASSINIT 类的构造函数在模块状态的 CRuntimeClass 链表中插入构造函数参数表示的 CRuntimeClass 类信息。至于模块状态，在后文有详细的讨论。
重载的操作符函数用来在序列化时从文档中读入该类对象的内容，是一个友员函数。定义如下：

```
CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb)
{
pOb = (class_name*) ar.ReadObject(
RUNTIME_CLASS(class_name));
return ar;
 }
```

回顾 CObject 的定义，它也有一个 CRuntimeClass 类型的静态成员变量 classCObject，因为它本身也支持三个特性。
以 CObject 及其派生类的静态成员变量 classCObject 为基础，IsKindOf 和动态创建等函数才可以起到作用。
这个变量为什么能有这样的用处，这就要分析 CRuntimeClass 类型变量的结构和内容了。下面，在讨论了 CRuntimeClass 的结构之后，考察该类型的静态变量被不同的宏初始化之后的内容。

3.3.2 CruntimeClass 类的结构与功能

从上面的讨论可以看出，在对 CObject 特性的支持上，CRuntimeClass 类起到了关键作用。下面，考查它的结构和功能。

（1） CRuntimeClass 的结构

CruntimeClass 的结构如下：

```
Struct CRuntimeClass
```

```
{
LPCSTR m_lpszClassName;//类的名字
int m_nObjectSize;//类的大小
UINT m_wSchema;
CObject* (PASCAL* m_pfnCreateObject)();
//pointer to function, equal to newclass.CreateObject()
//after IMPLEMENT
CRuntimeClass* (PASCAL* m_pfnGetBaseClass)();
CRumtieClass* m_pBaseClass;

//operator:
CObject *CreateObject();
BOOL IsDerivedFrom(const CRuntimeClass* pBaseClass) const;
...
}
```

CRuntimeClass 成员变量中有两个是函数指针，还有几个用来保存所在 CruntimeClass 对象所在类的名字、类的大小（字节数）等。

这些成员变量被三个实现宏初始化，例如：

m_pfnCreateObject，将被初始化指向所在类的静态成员函数 CreateObject。CreateObject 函数在初始化时由实现宏定义，见上文的说明。

m_pfnGetBaseClass，如果定义了_AFXDLL，则该变量将被初始化指向所在类的成员函数_GetBaseClass。_GetBaseClass 在声明宏中声明，在初始化时由实现宏定义，见上文的说明。

下面，分析三个宏对 CObject 及其派生类的 CRuntimeClass 类型的成员变量 class##class_name 初始化的情况，然后讨论 CRuntimeClass 成员函数的实现。

（2） 成员变量 class##class_name 的内容

IMPLEMENT_DYNCREATE 等宏将初始化类的 CRuntimeClass 类型静态成员变量的各个域，表 3-1 列出了在动态类信息、动态创建、序列化这三个不同层次下对该静态成员变量的初始化情况：

表3-1 静态成员变量class##class_name的初始化

CRuntimeClass 成员变量	动态类信息	动态创建	序列化
m_lpszClassName	类名字符串	类名字符串	类名字符串
m_nObjectSize	类的大小（字节数）	类的大小（字节数）	类的大小（字节数）
m_wShema	0xFFFF	0xFFFF	1、2 等，非 0
m_pfnCreateObject	NULL	类的成员函数 CreateObject	类的成员函数 CreateObject
m_pBaseClass	基类的 CRuntimeClass 变量	基类的 CRuntimeClass 变量	基类的 CRuntimeClass 变量
m_pfnGetBaseClass	类的成员函数 _GetBaseClass	类的成员函数 _GetBaseClass	类的成员函数 _GetBaseClass
m_pNextClass	NULL	NULL	NULL

m_wSchema 类型是 UINT，定义了序列化中保存对象到文档的程序的版本。如果不要求支持序列化特性，该域为 0XFFFF，否则，不能为 0。
Cobject 类本身的静态成员变量 classCObject 被初始化为：
{ "CObject", sizeof(CObject), 0xffff, NULL, &CObject::_GetBaseClass, NULL };
对初始化内容解释如下：
类名字符串是“CObject”，类的大小是 sizeof(CObject)，不要求支持序列化，不支持动态创建。

（3） 成员函数 CreateObject

回顾 3.2 节，动态创建对象是通过语句 pRuntimeClass->CreateObject 完成的，即调用了 CRuntimeClass 自己的成员函数，CreateObject 函数又调用 m_pfnCreateObject 指向的函数来完成动态创建任务，如下所示：

```
CObject* CRuntimeClass::CreateObject()
{
if (m_pfnCreateObject == NULL) //判断函数指针是否空
{
    TRACE(_T("Error: Trying to create object which is not ")
    _T("DECLARE_DYNCREATE \nor DECLARE_SERIAL: %hs.\n"),
    m_lpszClassName);
    return NULL;
}
//函数指针非空，继续处理
CObject* pObject = NULL;
TRY
{
    pObject = (*m_pfnCreateObject)(); //动态创建对象
}
END_TRY
return pObject;
}
```

（4） 成员函数 IsDerivedFrom

该函数用来帮助运行时判定一个类是否派生于另一个类，被 CObject 的成员函数 IsKindOf 函数所调用。其实现描述如下：

如果定义了_AFXDLL 则，成员函数 IsDerivedFrom 调用成员函数 m_pfnGetBaseClass 指向的函数来向上逐层得到基类的 CRuntimeClass 类型的静态成员变量，直到某个基类的 CRuntimeClass 类型的静态成员变量和参数指定的 CRuntimeClass 变量一致或者追寻到最上层为止。

如果没有定义_AFXDLL，则使用成员变量 m_pBaseClass 基类的 CRuntimeClass 类型的静态成员变量。

程序如下所示：

```
BOOL CRuntimeClass::IsDerivedFrom(
const CRuntimeClass* pBaseClass) const
{
ASSERT(this != NULL);
```

```
ASSERT(AfxIsValidAddress(this, sizeof(CRuntimeClass), FALSE));
ASSERT(pBaseClass != NULL);
ASSERT(AfxIsValidAddress(pBaseClass, sizeof(CRuntimeClass), FALSE));

// simple SI case
const CRuntimeClass* pClassThis = this;
while (pClassThis != NULL)//从本类开始向上逐个基类搜索
{
    if (pClassThis == pBaseClass)//若是参数指定的类信息
        return TRUE;
        //类信息不符合，继续向基类搜索
        #ifdef _AFXDLL
        pClassThis = (*pClassThis->m_pfnGetBaseClass)();
        #else
        pClassThis = pClassThis->m_pBaseClass;
        #endif
    }
return FALSE;          // 搜索完毕，没有匹配，返回 FALSE。
}
```

由于 CRuntimeClass 类型的成员变量是静态成员变量，所以如果两个类的 CruntimeClass 成员变量相同，必定是同一个类。这就是 IsDerivedFrom 和 IsKindOf 的实现基础。

（5） RUNTIME_CLASS 宏

RUNTIME_CLASS 宏定义如下：

```
#define RUNTIME_CLASS(class_name)  (&class_name::class##class_name)
```

为了方便地得到每个类（Cobject 或其派生类）的 CRuntimeClass 类型的静态成员变量，MFC 定义了这个宏。它返回对类 class_name 的 CRuntimeClass 类型成员变量的引用，该成员变量的名称是“class”加上 class_name（类的名字）。例如：

RUNTIME_CLASS(CObject)得到对 classCObject 的引用；

RUNTIME_CLASS(CPerson)得到对 class CPerson 的引用。

3.3.3 动态类信息、动态创建的原理

MFC 对 Cobject 动态类信息、动态创建的实现原理：

动态类信息、动态创建都建立在给类添加的 CRuntimeClass 类型的静态成员变量基础上，总结如下。

C++不支持动态创建，但是支持动态对象的创建。动态创建归根到底是创建动态对象，因为从一个类名创建一个该类的实例最终是创建一个以该类为类型的动态对象。其中的关键是从一个类名可以得到创建其动态对象的代码。

在一个类没有任何实例之前，怎么可以得到该类的创建动态对象的代码？借助于 C++的静态成员数据技术可达到这个目的：

- 静态成员数据在程序的入口(main 或 WinMain)之前初始化。因此，在一个静态成员数据里存放有关类型信息、动态创建函数等，需要的时候，得到这个成员数据就可以了。
- 不论一个类创建多少实例，静态成员数据只有一份。所有的类的实例共享一个静态成员数据，要判断一个类是否是一个类的实例，只须确认它是否使用了该类的这个静态数据。

从前两节的讨论知道，DECLARE_CREATE 等宏定义了一个这样的静态成员变量：类型是 CRuntimeClass，命名约定是“calss”加上类名；IMPLEMENT_CREATE 等宏初始化该变量；RUNTIME_CLASS 宏用来得到该成员变量。

动态类信息的原理在分析 CRuntimeClass 的成员函数 IsDerivedFrom 时已经作了解释。

动态创建的过程和原理了，用图表示其过程如下：

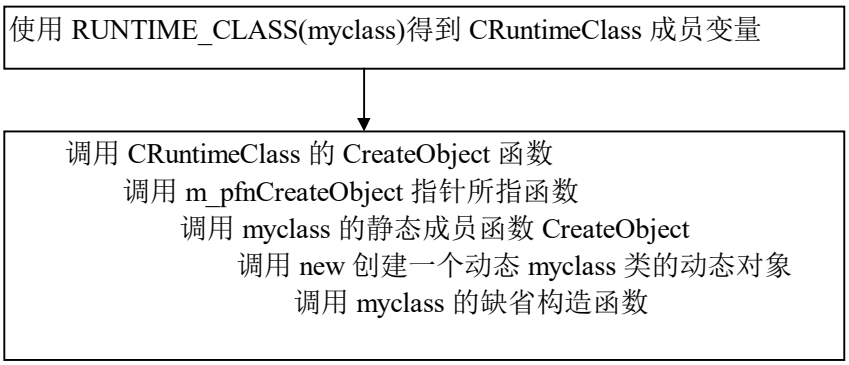


图 3-1　MFC 动态创建的过程和原理

注：下面一个方框内的逐级缩进表示逐层调用关系。

3.3.4 序列化的机制

由上所述可知，一个类要支持实现序列化，使得它的对象可以保存到文档中或者可以从文档中读入到内存中并生成对象，需要使用动态类信息，而且，需要覆盖基类的 Serialize 虚拟函数来完成其对象的序列化。

仅仅有类的支持是不够的，MFC 还提供了一个归档类 CArchive 来支持简单类型的数据和复杂对象的读写。

CArchive 在文件和内存对象之间充当一个代理者的角色。它负责按一定的顺序和格式把内存对象写到文件中，或者读出来，可以被看作是一个二进制的流。它和文件类 CFile 的关系如图 3-2 所示：

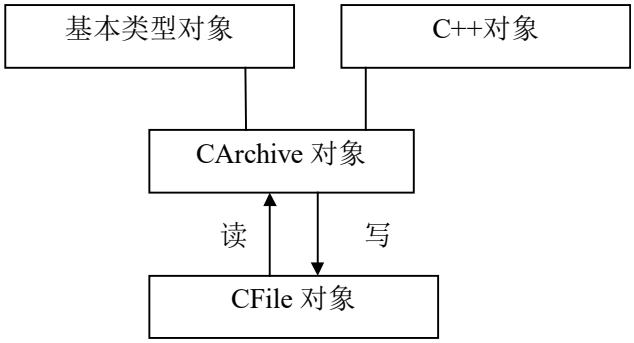


图 3-2　序列化的机制

一个 CArchive 对象在要序列化的对象和存储媒体(storage medium，可以是一个文件或者一个 Socket)之间起了中介作用。它提供了系列方法来完成序列化，不仅能够把 int、float 等简

单类型数据进行序列化，而且能够把复杂的数据如 string 等进行序列化，更重要的是它能把复杂的对象(包括复合对象)进行序列化。这些方法就是重载的操作符>>和<<。对于简单类型，它针对不同类型直接实现不同的读写操作；对于复杂的对象，其每一个支持序列化的类都重载了操作符>>，从前几节可以清楚地看到这点：IMPLEMENT_SERIAL 给所在类重载了操作符>>。至于<<操作，就不必每个序列化类都重载了。
复杂对象的“<<”操作，先搜索本模块状态的 CRuntimeClass 链表看是否有“<<”第二个参数指定的对象类的运行类信息(搜索过程涉及到模块状态，将在 9.5.2 节描述)，如果有（无，则返回），则先使用这些信息动态的创建对象（这就是是序列化类必须提供动态类信息、支持动态创建的原因），然后对该对象调用 Serilize 函数从存储媒体读入对象内容。
复杂对象的“>>”操作先把对象类的运行类信息写入存储媒体，然后对该对象调用 Serilize 函数把对象内容写入存储媒体。
在创建 CArchive 对象时，必须有一个 CFile 对象，它代表了存储媒介。通常，程序员不必做这个工作，打开或保存文档时 MFC 将自动的创建 CFile 对象和 CArchive 对象并在适当的时候调用序列化类的 Serialize 函数。在后面讨论打开(5.3.3.2 节)或者关闭(6.1 节)文档时将会看到这样的流程。
CArchive 对象被创建时，需要指定它是用来读还是用来写，即指定序列化操作的方向。Serialize 函数适用 CArchive 的函数 IsStoring 判定 CArchive 是用于读出数据还是写入数据。

在解释实现序列化的方法时，曾经提到如果程序员直接调用 Serilize 函数完成序列化，而不借助 CArchive 的>>和<<操作，则可以不需要动态类信息和动态创建。从上文的论述可以看出，没有 CArchive 的>>和<<操作，的确不需要动态类信息和动态创建特性。

第4章　消息映射的实现

4.1　Windows 消息概述

Windows 应用程序的输入由 Windows 系统以消息的形式发送给应用程序的窗口。这些窗口通过窗口过程来接收和处理消息，然后把控制返还给 Windows。

4.1.1 消息的分类

（1）　队列消息和非队列消息

从消息的发送途径上看，消息分两种：队列消息和非队列消息。队列消息送到系统消息队列，然后到线程消息队列；非队列消息直接送给目的窗口过程。

这里，对消息队列阐述如下：

Windows 维护一个系统消息队列（System message queue），每个 GUI 线程有一个线程消息队列（Thread message queue）。

鼠标、键盘事件由鼠标或键盘驱动程序转换成输入消息并把消息放进系统消息队列，例如 WM_MOUSEMOVE、WM_LBUTTONUP、WM_KEYDOWN、WM_CHAR 等等。Windows 每次从系统消息队列移走一个消息，确定它是送给哪个窗口的和这个窗口是由哪个线程创建的，然后，把它放进窗口创建线程的线程消息队列。线程消息队列接收送给该线程所创建窗口的消息。线程从消息队列取出消息，通过 Windows 把它送给适当的窗口过程来处理。

除了键盘、鼠标消息以外，队列消息还有 WM_PAINT、WM_TIMER 和 WM_QUIT。

这些队列消息以外的绝大多数消息是非队列消息。

（2）　系统消息和应用程序消息

从消息的来源来看，可以分为：系统定义的消息和应用程序定义的消息。

系统消息 ID 的范围是从 0 到 WM_USER-1，或 0X80000 到 0XBFFFF；应用程序消息从 WM_USER（0X0400）到 0X7FFF，或 0XC000 到 0XFFFF；WM_USER 到 0X7FFF 范围的消息由应用程序自己使用；0XC000 到 0XFFFF 范围的消息用来和其他应用程序通信，为了 ID 的唯一性，使用::RegisterWindowMessage 来得到该范围的消息 ID。

4.1.2 消息结构和消息处理

（1）　消息的结构

为了从消息队列获取消息信息，需要使用 MSG 结构。例如，::GetMessage 函数（从消息队列得到消息并从队列中移走）和::PeekMessage 函数(从消息队列得到消息但是可以不移走)都使用了该结构来保存获得的消息信息。

MSG 结构的定义如下：

```
typedef struct tagMSG {        // msg
    HWND     hwnd;
    UINT     message;
```

```
    WPARAM wParam;
    LPARAM lParam;
    DWORD    time;
    POINT    pt;
} MSG;
```

该结构包括了六个成员，用来描述消息的有关属性：

接收消息的窗口句柄、消息标识（ID）、第一个消息参数、第二个消息参数、消息产生的时间、消息产生时鼠标的位置。

（2） 应用程序通过窗口过程来处理消息

如前所述，每个“窗口类”都要登记一个如下形式的窗口过程：

```
LRESULT CALLBACK MainWndProc (
    HWND hwnd,//  窗口句柄
    UINT msg,//  消息标识
    WPARAM wParam,//消息参数 1
    LPARAM lParam//消息参数 2
)
```

应用程序通过窗口过程来处理消息：非队列消息由 Windows 直接送给目的窗口的窗口过程，队列消息由::DispatchMessage 等派发给目的窗口的窗口过程。窗口过程被调用时，接受四个参数：

a window handle（窗口句柄）；

a message identifier（消息标识）；

two 32-bit values called message parameters（两个 32 位的消息参数）；

需要的话，窗口过程用::GetMessageTime 获取消息产生的时间，用::GetMessagePos 获取消息产生时鼠标光标所在的位置。

在窗口过程里，用 switch/case 分支处理语句来识别和处理消息。

（3） 应用程序通过消息循环来获得对消息的处理

每个 GDI 应用程序在主窗口创建之后，都会进入消息循环，接受用户输入、解释和处理消息。

消息循环的结构如下：

```
while (GetMessage(&msg, (HWND) NULL, 0, 0)) {//从消息队列得到消息
    if (hwndDlgModeless == (HWND) NULL ||
        !IsDialogMessage(hwndDlgModeless, &msg) &&
        !TranslateAccelerator(hwndMain, haccel, &msg)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg); //发送消息
}
}
```

消息循环从消息队列中得到消息，如果不是快捷键消息或者对话框消息，就进行消息转换和派发，让目的窗口的窗口过程来处理。

当得到消息WM_QUIT，或者::GetMessage出错时，退出消息循环。

（4） MFC 消息处理

使用 MFC 框架编程时，消息发送和处理的本质也如上所述。但是，有一点需要强调的是，所有的 MFC 窗口都使用同一窗口过程，程序员不必去设计和实现自己的窗口过程，而是通过 MFC 提供的一套消息映射机制来处理消息。因此，MFC 简化了程序员编程时处理消息的复杂性。

所谓消息映射，简单地讲，就是让程序员指定要某个 MFC 类（有消息处理能力的类）处理某个消息。MFC 提供了工具 ClassWizard 来帮助实现消息映射，在处理消息的类中添加一些有关消息映射的内容和处理消息的成员函数。程序员将完成消息处理函数，实现所希望的消息处理能力。
如果派生类要覆盖基类的消息处理函数，就用ClassWizard在派生类中添加一个消息映射条目，用同样的原型定义一个函数，然后实现该函数。这个函数覆盖派生类的任何基类的同名处理函数。

下面几节将分析 MFC 的消息机制的实现原理和消息处理的过程。为此，首先要分析 ClassWizard 实现消息映射的内幕，然后讨论 MFC 的窗口过程，分析 MFC 窗口过程是如何实现消息处理的。

4.2 消息映射的定义和实现

4.2.1 MFC 处理的三类消息

根据处理函数和处理过程的不同，MFC 主要处理三类消息：

- Windows 消息，前缀以“WM_”打头，WM_COMMAND 例外。Windows 消息直接送给 MFC 窗口过程处理，窗口过程调用对应的消息处理函数。一般，由窗口对象来处理这类消息，也就是说，这类消息处理函数一般是 MFC 窗口类的成员函数。
- 控制通知消息，是控制子窗口送给父窗口的 WM_COMMAND 通知消息。窗口过程调用对应的消息处理函数。一般，由窗口对象来处理这类消息，也就是说，这类消息处理函数一般是 MFC 窗口类的成员函数。

需要指出的是，Win32 使用新的 WM_NOFITY 来处理复杂的通知消息。WM_COMMAND 类型的通知消息仅仅能传递一个控制窗口句柄(lparam)、控制窗 ID 和通知代码(wparam)。WM_NOTIFY 能传递任意复杂的信息。

- 命令消息，这是来自菜单、工具条按钮、加速键等用户接口对象的 WM_COMMAND 通知消息，属于应用程序自己定义的消息。通过消息映射机制，MFC 框架把命令按一定的路径分发给多种类型的对象（具备消息处理能力）处理，如文档、窗口、应用程序、文档模板等对象。能处理消息映射的类必须从 CCmdTarget 类派生。

在讨论了消息的分类之后，应该是讨论各类消息如何处理的时候了。但是，要知道怎么处理消息，首先要知道如何映射消息。

4.2.2 MFC 消息映射的实现方法

MFC 使用 ClassWizard 帮助实现消息映射，它在源码中添加一些消息映射的内容，并声明和实现消息处理函数。现在来分析这些被添加的内容。
在类的定义（头文件）里，它增加了消息处理函数声明，并添加一行声明消息映射的宏 DECLARE_MESSAGE_MAP。
在类的实现（实现文件）里，实现消息处理函数，并使用 IMPLEMENT_MESSAGE_MAP 宏实现消息映射。一般情况下，这些声明和实现是由 MFC 的 ClassWizard 自动来维护的。

看一个例子：
在 AppWizard 产生的应用程序类的源码中，应用程序类的定义（头文件）包含了类似如下的代码：

```
//{{AFX_MSG(CTttApp)
afx_msg void OnAppAbout();
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
```

应用程序类的实现文件中包含了类似如下的代码：

```
BEGIN_MESSAGE_MAP(CTApp, CWinApp)
//{{AFX_MSG_MAP(CTttApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

头文件里是消息映射和消息处理函数的声明，实现文件里是消息映射的实现和消息处理函数的实现。它表示让应用程序对象处理命令消息 ID_APP_ABOUT，消息处理函数是 OnAppAbout。
为什么这样做之后就完成了一个消息映射？这些声明和实现到底作了些什么呢？接着，将讨论这些问题。

4.2.3 在声明与实现的内部

（1） DECLARE_MESSAGE_MAP 宏：
首先，看 DECLARE_MESSAGE_MAP 宏的内容：

```
#ifdef _AFXDLL
#define DECLARE_MESSAGE_MAP() \
private: \
        static const AFX_MSGMAP_ENTRY _messageEntries[]; \
protected: \
static AFX_DATA const AFX_MSGMAP messageMap; \
static const AFX_MSGMAP* PASCAL _GetBaseMessageMap(); \
virtual const AFX_MSGMAP* GetMessageMap() const; \

#else
#define DECLARE_MESSAGE_MAP() \
private: \
        static const AFX_MSGMAP_ENTRY _messageEntries[]; \
protected: \
static AFX_DATA const AFX_MSGMAP messageMap; \
virtual const AFX_MSGMAP* GetMessageMap() const; \

#endif
```

DECLARE_MESSAGE_MAP 定义了两个版本，分别用于静态或者动态链接到 MFC

DLL 的情形。

（2） BEGIN_MESSAE_MAP 宏

然后，看 BEGIN_MESSAE_MAP 宏的内容：

```
#ifdef _AFXDLL
#define BEGIN_MESSAGE_MAP(theClass, baseClass) \
const AFX_MSGMAP* PASCAL theClass::_GetBaseMessageMap() \
{ return &baseClass::messageMap; } \
const AFX_MSGMAP* theClass::GetMessageMap() const \
{ return &theClass::messageMap; } \
AFX_DATADEF const AFX_MSGMAP theClass::messageMap = \
    { &theClass::_GetBaseMessageMap, &theClass::_messageEntries[0] }; \
const AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
{ \

#else
#define BEGIN_MESSAGE_MAP(theClass, baseClass) \
const AFX_MSGMAP* theClass::GetMessageMap() const \
{ return &theClass::messageMap; } \
AFX_DATADEF const AFX_MSGMAP theClass::messageMap = \
{ &baseClass::messageMap, &theClass::_messageEntries[0] }; \
const AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
{ \

#endif

#define END_MESSAGE_MAP() \
        {0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
    }; \
```

对应地，BEGIN_MESSAGE_MAP 定义了两个版本，分别用于静态或者动态链接到 MFC DLL 的情形。END_MESSAGE_MAP 相对简单，就只有一种定义。

（3） ON_COMMAND 宏

最后，看 ON_COMMAND 宏的内容：

```
#define ON_COMMAND(id, memberFxn) \
{\
    WM_COMMAND,\
    CN_COMMAND,\
    (WORD)id,\
    (WORD)id,\
    AfxSig_vv,\
    (AFX_PMSG)memberFxn\
};
```

4.2.3.1 消息映射声明的解释

在清楚了有关宏的定义之后，现在来分析它们的作用和功能。
消息映射声明的实质是给所在类添加几个静态成员变量和静态或虚拟函数，当然它们是与消息映射相关的变量和函数。

（1） 成员变量

有两个成员变量被添加，第一个是_messageEntries，第二个是 messageMap。

- 第一个成员变量的声明：

AFX_MSGMAP_ENTRY _messageEntries[]

这是一个 AFX_MSGMAP_ENTRY 类型的数组变量，是一个静态成员变量，用来容纳类的消息映射条目。一个消息映射条目可以用 AFX_MSGMAP_ENTRY 结构来描述。

AFX_MSGMAP_ENTRY 结构的定义如下：

```
struct AFX_MSGMAP_ENTRY
{
//Windows 消息 ID
UINT nMessage;
//控制消息的通知码
UINT nCode;
//Windows Control 的 ID
UINT nID;
//如果是一定范围的消息被映射，则 nLastID 指定其范围
UINT nLastID;

UINT nSig;//消息的动作标识
//响应消息时应执行的函数(routine to call (or special value))
AFX_PMSG pfn;
};
```

从上述结构可以看出，每条映射有两部分的内容：第一部分是关于消息 ID 的，包括前四个域；第二部分是关于消息对应的执行函数，包括后两个域。

在上述结构的六个域中，pfn 是一个指向 CCmdTarger 成员函数的指针。函数指针的类型定义如下：

```
typedef void (AFX_MSG_CALL CCmdTarget::*AFX_PMSG)(void);
```

当使用一条或者多条消息映射条目初始化消息映射数组时，各种不同类型的消息函数都被转换成这样的类型：不接收参数，也不返回参数的类型。因为所有可以有消息映射的类都是从 CCmdTarge 派生的，所以可以实现这样的转换。
nSig 是一个标识变量，用来标识不同原型的消息处理函数，每一个不同原型的消息处理函数对应一个不同的 nSig。在消息分发时，MFC 内部根据 nSig 把消息派发给对应的成员函数处理，实际上，就是根据 nSig 的值把 pfn 还原成相应类型的消息处理函数并执行它。

- 第二个成员变量的声明

AFX_MSGMAP messageMap;

这是一个 AFX_MSGMAP 类型的静态成员变量，从其类型名称和变量名称可以猜出，它是一个包含了消息映射信息的变量。的确，它把消息映射的信息（消息映射数组）和相关函数

打包在一起，也就是说，得到了一个消息处理类的该变量，就得到了它全部的消息映射数据和功能。AFX_MSGMAP 结构的定义如下：

```
struct AFX_MSGMAP
{
//得到基类的消息映射入口地址的数据或者函数
#ifdef _AFXDLL
//pfnGetBaseMap 指向_GetBaseMessageMap 函数
const AFX_MSGMAP* (PASCAL* pfnGetBaseMap)();
#else
//pBaseMap 保存基类消息映射入口_messageEntries 的地址
const AFX_MSGMAP* pBaseMap;
#endif
//lpEntries 保存消息映射入口_messageEntries 的地址
const AFX_MSGMAP_ENTRY* lpEntries;
};
```

从上面的定义可以看出，通过 messageMap 可以得到类的消息映射数组_messageEntries 和函数_GetBaseMessageMap 的地址（不使用 MFC DLL 时，是基类消息映射数组的地址）。

（2） **成员函数**

- _GetBaseMessageMap()

用来得到基类消息映射的函数。

- GetMessageMap()

用来得到自身消息映射的函数。

4.2.3.2 消息映射实现的解释

消息映射实现的实质是初始化声明中定义的静态成员函数_messageEntries 和 messageMap，实现所声明的静态或虚拟函数 GetMessageMap、_GetBaseMessageMap。

这样，在进入 WinMain 函数之前，每个可以响应消息的 MFC 类都生成了一个消息映射表，程序运行时通过查询该表判断是否需要响应某条消息。

（1） 对消息映射入口表(消息映射数组)的初始化

如前所述，消息映射数组的元素是消息映射条目，条目的格式符合结构 AFX_MESSAGE_ENTRY 的描述。所以，要初始化消息映射数组，就必须使用符合该格式的数据来填充：如果指定当前类处理某个消息，则把和该消息有关的信息（四个）和消息处理函数的地址及原型组合成为一个消息映射条目，加入到消息映射数组中。

显然，这是一个繁琐的工作。为了简化操作，MFC 根据消息的不同和消息处理方式的不同，把消息映射划分成若干类别，每一类的消息映射至少有一个共性：消息处理函数的原型相同。对每一类消息映射，MFC 定义了一个宏来简化初始化消息数组的工作。例如，前文提到的 ON_COMMAND 宏用来映射命令消息，只要指定命令 ID 和消息处理函数即可，因为对这类命令消息映射条目，其他四个属性都是固定的。ON_COMMAND 宏的初始化内容如下：

```
{WM_COMMAND,
CN_COMMAND,
(WORD)ID_APP_ABOUT,
```

(WORD)ID_APP_ABOUT,
AfxSig_vv,
(AFX_PMSG)OnAppAbout
}

这个消息映射条目的含义是：消息 ID 是 ID_APP_ABOUT，OnAppAbout 被转换成 AFX_PMSG 指针类型，AfxSig_vv 是 MFC 预定义的枚举变量，用来标识 OnAppAbout 的函数类型为参数空(Void)、返回空(Void)。

在消息映射数组的最后，是宏 END_MESSAGE_MAP 的内容，它标识消息处理类的消息映射条目的终止。

（2） 对 messageMap 的初始化

如前所述，messageMap 的类型是 AFX_MESSMAP。

经过初始化，域 lpEntries 保存了消息映射数组_messageEntries 的地址；如果动态链接到 MFC DLL，则 pfnGetBaseMap 保存了_GetBaseMessageMap 成员函数的地址；否则 pBaseMap 保存了基类的消息映射数组的地址。

（3） 对函数的实现

_GetBaseMessageMap()

它返回基类的成员变量 messagMap（当使用 MFC DLL 时），使用该函数得到基类消息映射入口表。

GetMessageMap()：

它返回成员变量 messageMap，使用该函数得到自身消息映射入口表。

顺便说一下，消息映射类的基类 CCmdTarget 也实现了上述和消息映射相关的函数，不过，它的消息映射数组是空的。

既然消息映射宏方便了消息映射的实现，那么有必要详细的讨论消息映射宏。下一节，介绍消息映射宏的分类、用法和用途。

4.2.4 消息映射宏的种类

为了简化程序员的工作，MFC 定义了一系列的消息映射宏和像 AfxSig_vv 这样的枚举变量，以及标准消息处理函数，并且具体地实现这些函数。这里主要讨论消息映射宏，常用的分为以下几类。

（1）用于 Windows 消息的宏，前缀为“ON_WM_”。

这样的宏不带参数，因为它对应的消息和消息处理函数的函数名称、函数原型是确定的。MFC 提供了这类消息处理函数的定义和缺省实现。每个这样的宏处理不同的 Windows 消息。例如：宏 ON_WM_CREATE()把消息 WM_CREATE 映射到 OnCreate 函数，消息映射条目的第一个成员 nMessage 指定为要处理的 Windows 消息的 ID，第二个成员 nCode 指定为 0。

（2）用于命令消息的宏 ON_COMMAND

这类宏带有参数，需要通过参数指定命令 ID 和消息处理函数。这些消息都映射到 WM_COMMAND 上，也就是将消息映射条目的第一个成员 nMessage 指定为 WM_COMMAND，第二个成员 nCode 指定为 CN_COMMAND(即 0)。消息处理函数的原型是 void (void)，不带参数，不返回值。

除了单条命令消息的映射，还有把一定范围的命令消息映射到一个消息处理函数的映射宏

ON_COMMAND_RANGE。这类宏带有参数，需要指定命令 ID 的范围和消息处理函数。这些消息都映射到 WM_COMMAND 上，也就是将消息映射条目的第一个成员 nMessage 指定为 WM_COMMAND，第二个成员 nCode 指定为 CN_COMMAND(即 0)，第三个成员 nID 和第四个成员 nLastID 指定了映射消息的起止范围。消息处理函数的原型是 void (UINT)，有一个 UINT 类型的参数，表示要处理的命令消息 ID，不返回值。

（3）用于控制通知消息的宏

这类宏可能带有三个参数，如 ON_CONTROL，就需要指定控制窗口 ID，通知码和消息处理函数；也可能带有两个参数，如具体处理特定通知消息的宏 ON_BN_CLICKED、ON_LBN_DBLCLK、ON_CBN_EDITCHANGE 等，需要指定控制窗口 ID 和消息处理函数。控制通知消息也被映射到 WM_COMMAND 上，也就是将消息映射条目的第一个成员的 nMessage 指定为 WM_COMMAND，但是第二个成员 nCode 是特定的通知码，第三个成员 nID 是控制子窗口的 ID，第四个成员 nLastID 等于第三个成员的值。消息处理函数的原型是 void (void)，没有参数，不返回值。

还有一类宏处理通知消息 ON_NOTIFY，它类似于 ON_CONTROL，但是控制通知消息被映射到 WM_NOTIFY。消息映射条目的第一个成员的 nMessage 被指定为 WM_NOTIFY，第二个成员 nCode 是特定的通知码，第三个成员 nID 是控制子窗口的 ID，第四个成员 nLastID 等于第三个成员的值。消息处理函数的原型是 void (NMHDR*, LRESULT*)，参数 1 是 NMHDR 指针，参数 2 是 LRESULT 指针，用于返回结果，但函数不返回值。

对应地，还有把一定范围的控制子窗口的某个通知消息映射到一个消息处理函数的映射宏，这类宏包括 ON__CONTROL_RANGE 和 ON_NOTIFY_RANGE。这类宏带有参数，需要指定控制子窗口 ID 的范围和通知消息，以及消息处理函数。

对于 ON__CONTROL_RANGE，是将消息映射条目的第一个成员的 nMessage 指定为 WM_COMMAND，但是第二个成员 nCode 是特定的通知码，第三个成员 nID 和第四个成员 nLastID 等于指定了控制窗口 ID 的范围。消息处理函数的原型是 void (UINT)，参数表示要处理的通知消息是哪个 ID 的控制子窗口发送的，函数不返回值。

对于 ON__NOTIFY_RANGE，消息映射条目的第一个成员的 nMessage 被指定为 WM_NOTIFY，第二个成员 nCode 是特定的通知码，第三个成员 nID 和第四个成员 nLastID 指定了控制窗口 ID 的范围。消息处理函数的原型是 void (UINT, NMHDR*, LRESULT*)，参数 1 表示要处理的通知消息是哪个 ID 的控制子窗口发送的，参数 2 是 NMHDR 指针，参数 3 是 LRESULT 指针，用于返回结果，但函数不返回值。

（4）用于用户界面接口状态更新的 ON_UPDATE_COMMAND_UI 宏

这类宏被映射到消息 WM_COMMND 上，带有两个参数，需要指定用户接口对象 ID 和消息处理函数。消息映射条目的第一个成员 nMessage 被指定为 WM_COMMAND，第二个成员 nCode 被指定为-1，第三个成员 nID 和第四个成员 nLastID 都指定为用户接口对象 ID。消息处理函数的原型是 void (CCmdUI*)，参数指向一个 CCmdUI 对象，不返回值。

对应地，有更新一定 ID 范围的用户接口对象的宏 ON_UPDATE_COMMAND_UI_RANGE，此宏带有三个参数，用于指定用户接口对象 ID 的范围和消息处理函数。消息映射条目的第一个成员 nMessage 被指定为 WM_COMMAND，第二个成员 nCode 被指定为-1，第三个成员 nID 和第四个成员 nLastID 用于指定用户接口对象 ID 的范围。消息处理函数的原型是 void (CCmdUI*)，参数指向一个 CCmdUI 对象，函数不返回值。之所以不用当前用户接口对象 ID 作为参数，是因为 CCmdUI 对象包含了有关信息。

（5）用于其他消息的宏

例如用于用户定义消息的 ON_MESSAGE。这类宏带有参数，需要指定消息 ID 和消息处理函数。消息映射条目的第一个成员 nMessage 被指定为消息 ID，第二个成员 nCode 被指定为

0，第三个成员 nID 和第四个成员也是 0。消息处理的原型是 LRESULT (WPARAM, LPARAM)，参数 1 和参数 2 是消息参数 wParam 和 lParam，返回 LRESULT 类型的值。

（6）扩展消息映射宏

很多普通消息映射宏都有对应的扩展消息映射宏，例如：ON_COMMAND 对应的 ON_COMMAND_EX，ON_ONTIFY 对应的 ON_ONTIFY_EX，等等。扩展宏除了具有普通宏的功能，还有特别的用途。关于扩展宏的具体讨论和分析，见 4.4.3.2 节。

作为一个总结，下表列出了这些常用的消息映射宏。

表4-1 常用的消息映射宏

消息映射宏	用途
ON_COMMAND	把 command message 映射到相应的函数
ON_CONTROL	把 control notification message 映射到相应的函数。MFC 根据不同的控制消息，在此基础上定义了更具体的宏，这样用户在使用时就不需要指定通知代码 ID，如 ON_BN_CLICKED。
ON_MESSAGE	把 user-defined message.映射到相应的函数
ON_REGISTERED_MESSAGE	把 registered user-defined message 映射到相应的函数，实际上 nMessage 等于 0x0C000，nSig 等于宏的消息参数。nSig 的真实值为 Afxsig_lwl。
ON_UPDATE_COMMAND_UI	把 user interface user update command message 映射到相应的函数上。
ON_COMMAND_RANGE	把一定范围内的 command IDs 映射到相应的函数上
ON_UPDATE_COMMAND_UI_RANGE	把一定范围内的 user interface user update command message 映射到相应的函数上
ON_CONTROL_RANGE	把一定范围内的 control notification message 映射到相应的函数上

在表 4-1 中，宏 ON_REGISTERED_MESSAGE 的定义如下：

```
#define ON_REGISTERED_MESSAGE(nMessageVariable, memberFxn) \
{ 0xC000, 0, 0, 0,\
(UINT)(UINT*)(&nMessageVariable), \
/*implied 'AfxSig_lwl'*/ \
(AFX_PMSG)(AFX_PMSGW)(LRESULT\
(AFX_MSG_CALL CWnd::*)\
(WPARAM, LPARAM))&memberFxn }
```

从上面的定义可以看出，实际上，该消息被映射到 WM_COMMAND(0XC000)，指定的 registered 消息 ID 存放在 nSig 域内，nSig 的值在这样的映射条目下隐含地定为 AfxSig_lwl。由于 ID 和正常的 nSig 域存放的值范围不同，所以 MFC 可以判断出是否是 registered 消息映射条目。如果是，则使用 AfxSig_lwl 把消息处理函数转换成参数 1 为 Word、参数 2 为 long、返回值为 long 的类型。

在介绍完了消息映射的内幕之后，应该讨论消息处理过程了。由于 CCmdTarge 的特殊性和重要性，在 4.3 节先对其作一个大略的介绍。

4.3 CcmdTarget 类

除了 CObject 类外，还有一个非常重要的类 CCmdTarget。所有响应消息或事件的类都从它派生。例如，CWinapp，CWnd，CDocument，CView，CDocTemplate，CFrameWnd，等等。CCmdTarget 类是 MFC 处理命令消息的基础、核心。MFC 为该类设计了许多成员函数和一些成员数据，基本上是为了解决消息映射问题的，而且，很大一部分是针对 OLE 设计的。在 OLE 应用中，CCmdTarget 是 MFC 处理模块状态的重要环节，它起到了传递模块状态的作用：其构造函数获取当前模块状态，并保存在成员变量 m_pModuleState 里头。关于模块状态，在后面章节讲述。

CCmdTarget 有两个与消息映射有密切关系的成员函数：DispatchCmdMsg 和 OnCmdMsg。

（1） 静态成员函数 DispatchCmdMsg

CCmdTarget 的静态成员函数 DispatchCmdMsg，用来分发 Windows 消息。此函数是 MFC 内部使用的，其原型如下：

```
static BOOL DispatchCmdMsg(
    CCmdTarget* pTarget,
    UINT nID,
    int nCode,
    AFX_PMSG pfn,
    void* pExtra,
    UINT nSig,
    AFX_CMDHANDLERINFO* pHandlerInfo)
```

关于此函数将在 4.4.3.2 章节命令消息的处理中作更详细的描述。

（2） 虚拟函数 OnCmdMsg

CCmdTarget 的虚拟函数 OnCmdMsg，用来传递和发送消息、更新用户界面对象的状态，其原型如下：

```
OnCmdMsg(
    UINT nID,
    int nCode,
    void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo)
```

框架的命令消息传递机制主要是通过该函数来实现的。其参数描述参见 4.3.3.2 章节 DispacthCMdMessage 的参数描述。

在本书中，命令目标指希望或者可能处理消息的对象；命令目标类指命令目标的类。

CCmdTarget 对 OnCmdMsg 的默认实现：在当前命令目标(this 所指)的类和基类的消息映射数组里搜索指定命令消息的消息处理函数（标准 Windows 消息不会送到这里处理）。

这里使用虚拟函数 GetMessageMap 得到命令目标类的消息映射入口数组_messageEntries，然后在数组里匹配指定的消息映射条目。匹配标准：命令消息 ID 相同，控制通知代码相同。因为 GetMessageMap 是虚拟函数，所以可以确认当前命令目标的确切类。

如果找到了一个匹配的消息映射条目，则使用 DispachCmdMsg 调用这个处理函数；

如果没有找到，则使用_GetBaseMessageMap 得到基类的消息映射数组，查找，直到找到或搜寻了所有的基类（到 CCmdTarget）为止；

如果最后没有找到，则返回 FASLE。

每个从 CCmdTarget 派生的命令目标类都可以覆盖 OnCmdMsg，利用它来确定是否可以处理某条命令，如果不能，就通过调用下一命令目标的 OnCmdMsg，把该命令送给下一个命令目标处理。通常，派生类覆盖 OnCmdMsg 时，要调用基类的被覆盖的 OnCmdMsg。
在 MFC 框架中，一些 MFC 命令目标类覆盖了 OnCmdMsg，如框架窗口类覆盖了该函数，实现了 MFC 的标准命令消息发送路径。具体实现见后续章节。
必要的话，应用程序也可以覆盖 OnCmdMsg，改变一个或多个类中的发送规定，实现与标准框架发送规定不同的发送路径。例如，在以下情况可以作这样的处理：在要打断发送顺序的类中把命令传给一个非 MFC 默认对象；在新的非默认对象中或在可能要传出命令的命令目标中。
本节对 CCmdTarget 的两个成员函数作一些讨论，是为了对 MFC 的消息处理有一个大致印象。后面 4.4.3.2 节和 4.4.3.3 节将作进一步的讨论。

4.4 MFC 窗口过程

前文曾经提到，所有的消息都送给窗口过程处理，MFC 的所有窗口都使用同一窗口过程，消息或者直接由窗口过程调用相应的消息处理函数处理，或者按 MFC 命令消息派发路径送给指定的命令目标处理。
那么，MFC 的窗口过程是什么？怎么处理标准 Windows 消息？怎么实现命令消息的派发？这些都将是下文要回答的问题。

4.4.1 MFC 窗口过程的指定

从前面的讨论可知，每一个“窗口类”都有自己的窗口过程。正常情况下使用该“窗口类”创建的窗口都使用它的窗口过程。
MFC 的窗口对象在创建 HWND 窗口时，也使用了已经注册的“窗口类”，这些“窗口类”或者使用应用程序提供的窗口过程，或者使用 Windows 提供的窗口过程（例如 Windows 控制窗口、对话框等）。那么，为什么说 MFC 创建的所有 HWND 窗口使用同一个窗口过程呢？在 MFC 中，的确所有的窗口都使用同一个窗口过程：AfxWndProc 或 AfxWndProcBase（如果定义了_AFXDLL）。它们的原型如下：

```
LRESULT CALLBACK
AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam)

LRESULT CALLBACK
AfxWndProcBase(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam)
```

这两个函数的原型都如 4.1.1 节描述的窗口过程一样。
如果动态链接到 MFC DLL(定义了_AFXDLL)，则 AfxWndProcBase 被用作窗口过程，否则 AfxWndProc 被用作窗口过程。AfxWndProcBase 首先使用宏 AFX_MANAGE_STATE 设置正确的模块状态，然后调用 AfxWndProc。

下面，假设不使用 MFC DLL，讨论 MFC 如何使用 AfxWndProc 取代各个窗口的原窗口过程。窗口过程的取代发生在窗口创建的过程时，使用了子类化(Subclass)的方法。所以，从窗口的创建过程来考察取代过程。从前面可以知道，窗口创建最终是通过调用 CWnd::CreateEx 函数完成的，分析该函数的流程，如图 4-1 所示。

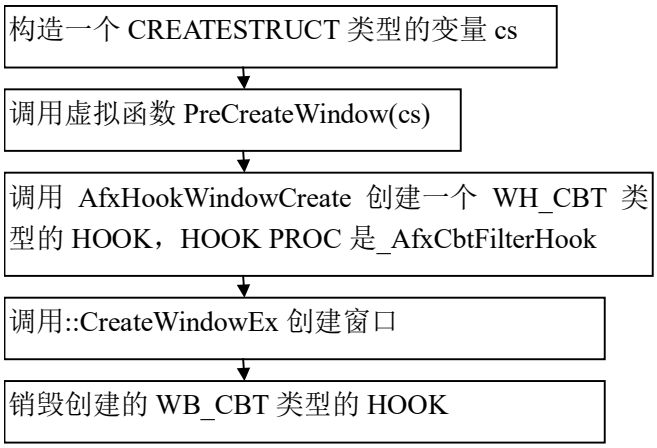


图 4-1　指定 MFC 的窗口过程

图 4-1 中的 CREATESTRUCT 结构类型的变量 cs 包含了传递给窗口过程的初始化参数。CREATESTRUCT 结构描述了创建窗口所需要的信息，定义如下：

```
typedef struct tagCREATESTRUCT {
LPVOID      lpCreateParams; //用来创建窗口的数据
HANDLE      hInstance; //创建窗口的实例
HMENU       hMenu; //窗口菜单
HWND        hwndParent; //父窗口
int         cy; //高度
int         cx; //宽度
int         y; //原点 Y 坐标
int         x;//原点 X 坐标
LONG        style; //窗口风格
LPCSTR      lpszName; //窗口名
LPCSTR      lpszClass; //窗口类
DWORD       dwExStyle; //窗口扩展风格
} CREATESTRUCT;
```

cs 表示的创建参数可以在创建窗口之前被程序员修改，程序员可以覆盖当前窗口类的虚拟成员函数 PreCreateWindow，通过该函数来修改 cs 的 style 域，改变窗口风格。这里 cs 的主要作用是保存创建窗口的各种信息，::CreateWindowEx 函数使用 cs 的各个域作为参数来创建窗口，关于该函数见 2.2.2 节。

在创建窗口之前，创建了一个 WH_CBT 类型的钩子（Hook）。这样，创建窗口时所有的消息都会被钩子过程函数_AfxCbtFilterHook 截获。

AfxCbtFilterHook 函数首先检查是不是希望处理的 Hook——HCBT_CREATEWND。如果是，则先把 MFC 窗口对象（该对象必须已经创建了）和刚刚创建的 Windows 窗口对象捆绑在一起，建立它们之间的映射（见后面模块-线程状态）；然后，调用::SetWindowLong 设置窗口过程为 AfxWndProc，并保存原窗口过程在窗口类成员变量 m_pfnSuper 中，这样形成一个窗口过程链。需要的时候，原窗口过程地址可以通过窗口类成员函数 GetSuperWndProcAddr 得到。

这样，AfxWndProc 就成为 CWnd 或其派生类的窗口过程。不论队列消息，还是非队列消息，都送到 AfxWndProc 窗口过程来处理（如果使用 MFC DLL，则 AfxWndProcBase 被调用，然后是 AfxWndProc）。经过消息分发之后没有被处理的消息，将送给原窗口过程处理。

最后，有一点可能需要解释：为什么不直接指定窗口过程为 AfxWndProc，而要这么大费周折呢？这是因为原窗口过程（“窗口类”指定的窗口过程）常常是必要的，是不可缺少的。接下来，讨论 AfxWndProc 窗口过程如何使用消息映射数据实现消息映射。Windows 消息和命令消息的处理不一样，前者没有消息分发的过程。

4.4.2 对 Windows 消息的接收和处理

Windows 消息送给 AfxWndProc 窗口过程之后，AfxWndProc 得到 HWND 窗口对应的 MFC 窗口对象，然后，搜索该 MFC 窗口对象和其基类的消息映射数组，判定它们是否处理当前消息，如果是则调用对应的消息处理函数，否则，进行缺省处理。

下面，以一个应用程序的视窗口创建时，对 WM_CREATE 消息的处理为例，详细地讨论 Windows 消息的分发过程。

用第一章的例子，类 CTview 要处理 WM_CREATE 消息，使用 ClassWizard 加入消息处理函数 CTview::OnCreate。下面，看这个函数怎么被调用：

视窗口最终调用::CreateEx 函数来创建。由 Windows 系统发送 WM_CREATE 消息给视的窗口过程 AfxWndProc，参数 1 是创建的视窗口的句柄，参数 2 是消息 ID（WM_CREATE)，参数 3、4 是消息参数。图 4-2 描述了其余的处理过程。图中函数的类属限制并非源码中所具有的，而是根据处理过程得出的判断。例如，“CWnd::WindowProc”表示 CWnd 类的虚拟函数 WindowProc 被调用，并不一定当前对象是 CWnd 类的实例，事实上，它是 CWnd 派生类 CTview 类的实例；而“CTview::OnCreate”表示 CTview 的消息处理函数 OnCreate 被调用。下面描述每一步的详细处理。

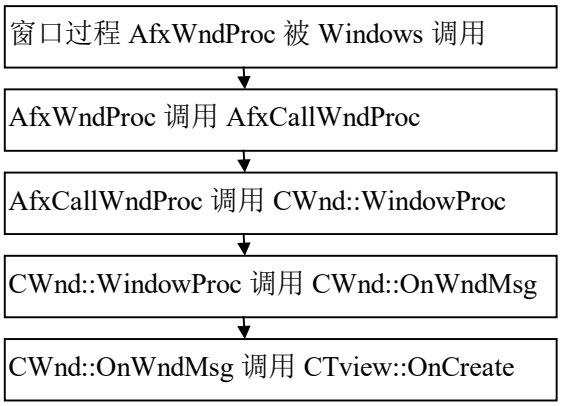


图 4-2　标准 Windows 消息的处理

4.4.2.1　从窗口过程到消息映射

首先，分析 AfxWndProc 窗口过程函数。

- AfxWndProc 的原型如下：

```
LRESULT AfxWndProc(HWND hWnd,
        UINT nMsg, WPARAM wParam, LPARAM lParam)
```

如果收到的消息 nMsg 不是 WM_QUERYAFXWNDPROC（该消息被 MFC 内部用来确认窗口过程是否使用 AfxWndProc），则从 hWnd 得到对应的 MFC Windows 对象（该对象必须已

存在，是永久性<Permanent>对象）指针 pWnd。pWnd 所指的 MFC 窗口对象将负责完成消息的处理。这里，pWnd 所指示的对象是 MFC 视窗口对象，即 CTview 对象。
然后，把 pWnd 和 AfxWndProc 接受的四个参数传递给函数 AfxCallWndProc 执行。

- AfxCallWndProc 原型如下：

```
LRESULT AFXAPI AfxCallWndProc(CWnd* pWnd, HWND hWnd,
    UINT nMsg, WPARAM wParam = 0, LPARAM lParam = 0)
```

MFC 使用 AfxCallWndProc 函数把消息送给 CWnd 类或其派生类的对象。该函数主要是把消息和消息参数(nMsg、wParam、lParam)传递给 MFC 窗口对象的成员函数 WindowProc（pWnd->WindowProc）作进一步处理。如果是 WM_INITDIALOG 消息，则在调用 WindowProc 前后要作一些处理。

WindowProc 的函数原型如下：

```
LRESULT CWnd::WindowProc(UINT message,
    WPARAM wParam, LPARAM lParam)
```

这是一个虚拟函数，程序员可以在 CWnd 的派生类中覆盖它，改变 MFC 分发消息的方式。例如，MFC 的 CControlBar 就覆盖了 WindowProc，对某些消息作了自己的特别处理，其他消息处理由基类的 WindowProc 函数完成。
但是在当前例子中，当前对象的类 CTview 没有覆盖该函数，所以 CWnd 的 WindowProc 被调用。
这个函数把下一步的工作交给 OnWndMsg 函数来处理。如果 OnWndMsg 没有处理，则交给 DefWindowProc 来处理。
OnWndMsg 和 DefWindowProc 都是 CWnd 类的虚拟函数。

- OnWndMsg 的原型如下：

```
BOOL CWnd::OnWndMsg( UINT message,
    WPARAM wParam, LPARAM lParam,RESULT*pResult );
```

该函数是虚拟函数。
和 WindowProc 一样，由于当前对象的类 CTview 没有覆盖该函数，所以 CWnd 的 OnWndMsg 被调用。
在CWnd中，MFC使用OnWndMsg来分别处理各类消息：
如果是WM_COMMAND消息，交给OnCommand处理；然后返回。
如果是WM_NOTIFY消息，交给OnNotify处理；然后返回。
如果是WM_ACTIVATE消息，先交给_AfxHandleActivate处理（后面5.3.3.7节会解释它的处理），再继续下面的处理。
如果是WM_SETCURSOR消息，先交给_AfxHandleSetCursor处理；然后返回。
如果是其他的Windows消息（包括WM_ACTIVATE），则
首先在消息缓冲池进行消息匹配，
若匹配成功，则调用相应的消息处理函数；
若不成功，则在消息目标的消息映射数组中进行查找匹配，看它是否处理当前消息。这里，消息目标即CTview对象。
如果消息目标处理了该消息，则会匹配到消息处理函数，调用它进行处理；
否则，该消息没有被应用程序处理，OnWndMsg返回FALSE。

关于 Windows 消息和消息处理函数的匹配，见下一节。

缺省处理函数 DefWindowProc 将在讨论对话框等的实现时具体分析。

4.4.2.2 Windows 消息的查找和匹配

CWnd或者派生类的对象调用OnWndMsg搜索本对象或者基类的消息映射数组，寻找当前消息的消息处理函数。如果当前对象或者基类处理了当前消息，则必定在其中一个类的消息映射数组中匹配到当前消息的处理函数。
消息匹配是一个比较耗时的任务，为了提高效率，MFC设计了一个消息缓冲池，把要处理的消息和匹配到的消息映射条目（条目包含了消息处理函数的地址）以及进行消息处理的当前类等信息构成一条缓冲信息，放到缓冲池中。如果以后又有同样的消息需要同一个类处理，则直接从缓冲池查找到对应的消息映射条目就可以了。
MFC用哈希查找来查询消息映射缓冲池。消息缓冲池相当于一个哈希表，它是应用程序的一个全局变量，可以放512条最新用到的消息映射条目的缓冲信息，每一条缓冲信息是哈希表的一个入口。
采用AFX_MSG_CACHE结构描述每条缓冲信息，其定义如下：

```
struct AFX_MSG_CACHE
{
UINT nMsg;
const AFX_MSGMAP_ENTRY* lpEntry;
const AFX_MSGMAP* pMessageMap;
};
```

nMsg存放消息ID，每个哈希表入口有不同的nMsg。
lpEnty存放和消息ID匹配的消息映射条目的地址，它可能是this所指对象的类的映射条目，也可能是这个类的某个基类的映射条目，也可能是空。
pMessageMap存放消息处理函数匹配成功时进行消息处理的当前类（this所指对象的类）的静态成员变量messageMap的地址，它唯一的标识了一个类（每个类的messageMap变量都不一样）。
this所指对象是一个CWnd或其派生类的实例，是正在处理消息的MFC窗口对象。
哈希查找：使用消息ID的值作为关键值进行哈希查找，如果成功，即可从lpEntry获得消息映射条目的地址，从而得到消息处理函数及其原型。
如何判断是否成功匹配呢？有两条标准：
第一，当前要处理的消息 message 在哈希表（缓冲池）中有入口；第二，当前窗口对象（this 所指对象）的类的静态变量 messageMap 的地址应该等于本条缓冲信息的 pMessagMap。MFC 通过虚拟函数 GetMessagMap 得到 messageMap 的地址。

如果在消息缓冲池中没有找到匹配，则搜索当前对象的消息映射数组，看是否有合适的消息处理函数。

如果匹配到一个消息处理函数,则把匹配结果加入到消息缓冲池中，即填写该条消息对应的哈希表入口：

```
nMsg=message;
pMessageMap=this->GetMessageMap;
```

lpEntry=查找结果

然后，调用匹配到的消息处理函数。否则（没有找到），使用_GetBaseMessageMap 得到基类的消息映射数组，查找和匹配；直到匹配成功或搜寻了所有的基类（到 CCmdTarget）为止。如果最后没有找到，则也把该条消息的匹配结果加入到缓冲池中。和匹配成功不同的是：指定 lpEntry 为空。这样 OnWndMsg 返回，把控制权返还给 AfxCallWndProc 函数，AfxCallWndProc 将继续调用 DefWndProc 进行缺省处理。

消息映射数组的搜索在 CCmdTarget::OnCmdMsg 函数中也用到了，而且算法相同。为了提高速度，MFC 把和消息映射数组条目逐一比较、匹配的函数 AfxFindMessageEntry 用汇编书写。

```
const AFX_MSGMAP_ENTRY* AFXAPI
AfxFindMessageEntry(const AFX_MSGMAP_ENTRY* lpEntry,
    UINT nMsg, UINT nCode, UINT nID)
```

第一个参数是要搜索的映射数组的入口；第二个参数是Windows消息标识；第三个参数是控制通知消息标识；第四个参数是命令消息标识。

对Windows消息来说，nMsg是每条消息不同的，nID和nCode为0。

对命令消息来说，nMsg固定为WM_COMMAND，nID是每条消息不同，nCode都是CN_COMMAND（定义为0）。

对控制通知消息来说，nMsg固定为WM_COMMAND或者WM_NOTIFY，nID和nCode是每条消息不同。

对于Register消息，nMsg指定为0XC000，nID和nCode为0。在使用函数AfxFindMessageEntry得到匹配结果之后，还必须判断nSig是否等于message，只有相等才调用对应的消息处理函数。

4.4.2.3 Windows 消息处理函数的调用

对一个 Windows 消息，匹配到了一个消息映射条目之后，将调用映射条目所指示的消息处理函数。

调用处理函数的过程就是转换映射条目的 pfn 指针为适当的函数类型并执行它：MFC 定义了一个成员函数指针 mmf，首先把消息处理函数的地址赋值给该函数指针，然后根据消息映射条目的 nSig 值转换指针的类型。但是，要给函数指针 mmf 赋值，必须使该指针可以指向所有的消息处理函数，为此则该指针的类型是所有类型的消息处理函数指针的联合体。

对上述过程，MFC 的实现大略如下：

```
union MessageMapFunctions mmf;
mmf.pfn = lpEntry->pfn;
swithc (value_of_nsig){
    …
    case AfxSig_is:   //OnCreate 就是该类型
        lResult = (this->*mmf.pfn_is)((LPTSTR)lParam);
        break;
    …
default:
    ASSERT(FALSE); break;
}
```

```
…
LDispatchRegistered:      // 处理 registered windows messages
    ASSERT(message >= 0xC000);
    mmf.pfn = lpEntry->pfn;
    lResult = (this->*mmf.pfn_lwl)(wParam, lParam);
    …
```

如果消息处理函数有返回值，则返回该结果，否则，返回TRUE。
对于图 4-1 所示的例子，nSig 等于 AfxSig_is，所以将执行语句

```
(this->*mmf.pfn_is)((LPTSTR)lParam)
```

也就是对 CTview::OnCreate 的调用。
顺便指出，对于Registered窗口消息，消息处理函数都是同一原型，所以都被转换成lwl型（关于Registered窗口消息的映射，见4.4.2节）。

综上所述，标准Windwos消息和应用程序消息中的Registered消息，由窗口过程直接调用相应的处理函数处理：
如果某个类型的窗口（C++类）处理了某条消息（覆盖了CWnd或直接基类的处理函数），则对应的HWND窗口（Winodws window）收到该消息时就调用该覆盖函数来处理；如果该类窗口没有处理该消息，则调用实现该处理函数最直接的基类（在C++的类层次上接近该类）来处理，上述例子中如果CTview不处理WM_CREATE消息，则调用上一层的CWnd::OnCreate处理；
如果基类都不处理该消息，则调用DefWndProc来处理。

4.4.2.4 消息映射机制完成虚拟函数功能的原理

综合对Windows消息的处理来看，MFC使用消息映射机制完成了C++虚拟函数的功能。这主要基于以下几点：

- 所有处理消息的类从 CCmdTarget 派生。
- 使用静态成员变量_messageEntries 数组存放消息映射条目，使用静态成员变量 messageMap 来唯一地区别和得到类的消息映射。
- 通过 GetMessage 虚拟函数来获取当前对象的类的 messageMap 变量，进而得到消息映射入口。
- 按照先底层，后基层的顺序在类的消息映射数组中搜索消息处理函数。基于这样的机制，一般在覆盖基类的消息处理函数时，应该调用基类的同名函数。

以上论断适合于 MFC 其他消息处理机制，如对命令消息的处理等。不同的是其他消息处理有一个命令派发/分发的过程。
下一节，讨论命令消息的接受和处理。

4.4.3 对命令消息的接收和处理

4.4.3.1 MFC 标准命令消息的发送

在 SDI 或者 MDI 应用程序中，命令消息由用户界面对象（如菜单、工具条等）产生，然后送给主边框窗口。主边框窗口使用标准 MFC 窗口过程处理命令消息。窗口过程把命令传递给 MFC 主边框窗口对象，开始命令消息的分发。MFC 边框窗口类 CFrameWnd 提供了消息分发的能力。

下面，还是通过一个例子来说明命令消息的处理过程。

使用 AppWizard 产生一个单文档应用程序 t。从 help 菜单选择“About”，就会弹出一个 ABOUT 对话框。下面，讨论从命令消息的发出到对话框弹出的过程。

首先，选择“About”菜单项的动作导致一个 Windows 命令消息 ID_APP_ABOUT 的产生。Windows 系统发送该命令消息到边框窗口，导致它的窗口过程 AfxWndProc 被调用，参数 1 是边框窗口的句柄，参数 2 是消息 ID（即 WM_COMMAND），参数 3、4 是消息参数，参数 3 的值是 ID_APP_ABOUT。接着的系列调用如图 4-3 所示。

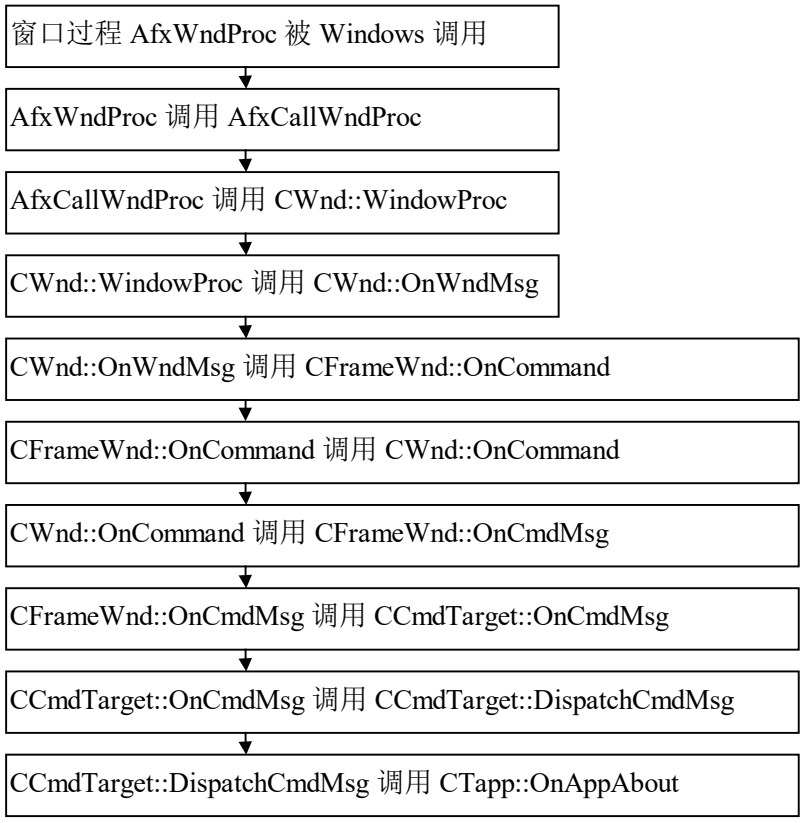


图 4-3　命令消息的处理

下面分别讲述每一层所调用的函数。

前 4 步同对 Windows 消息的处理。这里接受消息的 HWND 窗口是主边框窗口，因此，AfxWndProc 根据 HWND 句柄得到的 MFC 窗口对象是 MFC 边框窗口对象。

在 4.2.2 节谈到，如果 CWnd::OnWndMsg 判断要处理的消息是命令消息(WM_COMMAND)，就调用 OnCommand 进一步处理。由于 OnCommand 是虚拟函数，当前 MFC 窗口对象是边框窗口对象，它的类从 CFrameWnd 类导出，没有覆盖 CWnd 的虚拟函数 OnCommand，而 CFrameWnd 覆盖了 CWnd 的 OnCommand，所以，CFrameWnd 的 OnCommand 被调用。换句话说，CFrameWnd 的 OnCommand 被调用是动态约束的结果。接着介绍的本例子的有关调用，也是通过动态约束而实际发生的函数调用。

接着的有关调用，将不进行为什么调用某个类的虚拟或者消息处理函数的分析。

（1）CFrameWnd 的 OnCommand 函数

BOOL CFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam)

参数 wParam 的低阶 word 存放了菜单命令 nID 或控制子窗口 ID；如果消息来自控制窗口，高阶 word 存放了控制通知消息；如果消息来自加速键，高阶 word 值为 1；如果消息来自菜单，高阶 word 值为 0。

如果是通知消息，参数 lParam 存放了控制窗口的句柄 hWndCtrl，其他情况下 lParam 是 0。

在这个例子里，低阶 word 是 ID_APP_ABOUT，高阶 word 是 1；lParam 是 0。

MFC 对 CFrameWnd 的缺省实现主要是获得一个机会来检查程序是否运行在 HELP 状态，需要执行上下文帮助，如果不需要，则调用基类的 CWnd::OnCommand 实现正常的命令消息发送。

（2）CWnd 的 OnCommand 函数

BOOL CWnd::OnCommand(WPARAM wParam, LPARAM lParam)

它按一定的顺序处理命令或者通知消息，如果发送成功，返回 TRUE，否则，FALSE。处理顺序如下：

如果是命令消息，则调用 OnCmdMsg(nID, CN_UPDATE_COMMAND_UI, &state, NULL)测试 nID 命令是否已经被禁止，如果这样，返回 FALSE；否则，调用 OnCmdMsg 进行命令发送。关于 CN_UPDATE_COMMAND_UI 通知消息，见后面用户界面状态的更新处理。

如果是控制通知消息，则先用 ReflectLastMsg 反射通知消息到子窗口。如果子窗口处理了该消息，则返回 TRUE；否则，调用 OnCmdMsg 进行命令发送。关于通知消息的反射见后面 4.4.4.3 节。OnCommand 给 OnCmdMsg 传递四个参数：nID，即命令消息 ID；nCode，如果是通知消息则为通知代码，如果是命令消息则为 NC_COMMAND(即 0)；其余两个参数为空。

（3）CFrameWnd 的 OnCmdMsg 函数

BOOL CFrameWnd::OnCmdMsg(UINT nID, int nCode, void* pExtra,

AFX_CMDHANDLERINFO* pHandlerInfo)

参数 1 是命令 ID；如果是通知消息（WM_COMMAND 或者 WM_NOTIFY），则参数 2 表示通知代码，如果是命令消息，参数 2 是 0；如果是 WM_NOTIFY，参数 3 包含了一些额外的信息；参数 4 在正常消息处理中应该是空。

在这个例子里，参数 1 是命令 ID，参数 2 为 0，参数 3 空。

OnCmdMsg 是虚拟函数，CFrameWnd 覆盖了该函数，当前对象（this 所指）是 MFC 单文档的边框窗口对象。故 CFrameWnd 的 OnCmdMsg 被调用。CFrameWnd::OnCmdMsg 在 MFC 消息发送中占有非常重要的地位，MFC 对该函数的缺省实现确定了 MFC 的标准命令发送路径：

第一，送给活动（Active）视处理，调用活动视的 OnCmdMsg。由于当前对象是 MFC 视对象，所以，OnCmdMsg 将搜索 CTview 及其基类的消息映射数组，试图得到相应的处理函数。

第二，如果视对象自己不处理，则视得到和它关联的文档，调用关联文档的 OnCmdMsg。由于当前对象是 MFC 视对象，所以，OnCmdMsg 将搜索 CTdoc 及其基类的消息映射数组，试图得到相应的处理函数。

第三，如果文档对象不处理，则它得到管理文档的文档模板对象，调用文档模板的 OnCmdMsg。由于当前对象是 MFC 文档模板对象，所以，OnCmdMsg 将搜索文档模板类及其基类的消息映射数组，试图得到相应的处理函数。

第四，如果文档模板不处理，则把没有处理的信息逐级返回：文档模板告诉文档对象，文档对象告诉视对象，视对象告诉边框窗口对象。最后，边框窗口得知，视、文档、文档模

板都没有处理消息。

第五，CFrameWnd 的 OnCmdMsg 继续调用 *CWnd::OnCmdMsg*（斜体表示有类属限制）来处理消息。由于 CWnd 没有覆盖 OnCmdMsg，故实际上调用了函数 CCmdTarget::OnCmdMsg。由于当前对象是 MFC 边框窗口对象，所以 OnCmdMsg 函数将搜索 CMainFrame 类及其所有基类的消息映射数组，试图得到相应的处理函数。CWnd 没有实现 OnCmdMsg 却指定要执行其 OnCmdMsg 函数，可能是为了以后 MFC 给 CWnd 实现了 OnCmdMsg 之后其他代码不用改变。

这一步是边框窗口自己尝试处理消息。

第六，如果边框窗口对象不处理，则送给应用程序对象处理。调用 CTApp 的 OnCmdMsg，由于实际上 CTApp 及其基类 CWinApp 没有覆盖 OnCmdMsg，故实际上调用了函数 CCmdTarget::OnCmdMsg。由于当前对象是 MFC 应用程序对象，所以 OnCmdMsg 函数将搜索 CTApp 类及其所有基类的的消息映射入口数组，试图得到相应的处理函数

第七，如果应用程序对象不处理，则返回 FALSE，表明没有命令目标处理当前的命令消息。这样，函数逐级别返回，OnCmdMsg 告诉 OnCommand 消息没有被处理，OnCommand 告诉 OnWndMsg 消息没有被处理，OnWndMsg 告诉 WindowProc 消息没有被处理，于是 WindowProc 调用 DefWindowProc 进行缺省处理。

本例子在第六步中，应用程序对 ID_APP_ABOUT 消息作了处理。它找到处理函数 CTApp::OnAbout，使用 DispatchCmdMsg 派发消息给该函数处理。

如果是 MDI 边框窗口，标准发送路径还有一个环节，该环节和第二、三、四步所涉及的 OnCmdMsg 函数，将在下两节再次具体分析。

4.4.3.2 命令消息的派发和消息的多次处理

（1） 命令消息的派发

如前 3.1 所述，CCmdTarget 的静态成员函数 DispatchCmdMsg 用来派发命令消息给指定的命令目标的消息处理函数。

```
static BOOL DispatchCmdMsg(CCmdTarget* pTarget,
    UINT nID, int nCode,
    AFX_PMSG pfn, void* pExtra, UINT nSig,
    AFX_CMDHANDLERINFO* pHandlerInfo)
```

前面在讲 CCmdTarget 时，提到了该函数。这里讲述它的实现：

第一个参数指向处理消息的对象；第二个参数是命令 ID；第三个是通知消息等；第四个是消息处理函数地址；第五个参数用于存放一些有用的信息，根据 nCode 的值表示不同的意义，例如当消息是 WM_NOFITY，指向一个 NMHDR 结构（关于 WM_NOTIFY，参见 4.4.4.2 节通知消息的处理）；第六个参数标识消息处理函数原型；第七个参数是一个指针，指向 AFX_CMDHANDLERINFO 结构。前六个参数（除了第五个外）都是向函数传递信息，第五个和第七个参数是双向的，既向函数传递信息，也可以向调用者返回信息。

关于 AFX_CMDHANDLERINFO 结构：

```
struct AFX_CMDHANDLERINFO
{
CCmdTarget* pTarget;
void (AFX_MSG_CALL CCmdTarget::*pmf)(void);
};
```

第一个成员是一个指向命令目标对象的指针，第二个成员是一个指向 CCmdTarget 成员函数的指针。

该函数的实现流程可以如下描述：
首先，它检查参数 pHandlerInfo 是否空，如果不空，则用 pTarget 和 pfn 填写其指向的结构，返回 TRUE；通常消息处理时传递来的 pHandlerInfo 空，而在使用 OnCmdMsg 来测试某个对象是否处理某条命令时，传递一个非空的 pHandlerInfo 指针。若返回 TRUE，则表示可以处理那条消息。
如果 pHandlerInfo 空，则进行消息处理函数的调用。它根据参数 nSig 的值，把参数 pfn 的类型转换为要调用的消息处理函数的类型。这种指针转换技术和前面讲述的 Windows 消息的处理是一样的。

（2） 消息的多次处理

如果消息处理函数不返回值，则 DispatchCmdMsg 返回 TRUE；否则，DispatchCmdMsg 返回消息处理函数的返回值。这个返回值沿着消息发送相反的路径逐级向上传递，使得各个环节的 OnCmdMsg 和 OnCommand 得到返回的处理结果：TRUE 或者 FALSE，即成功或者失败。
这样就产生了一个问题，如果消息处理函数有意返回一个 FALSE，那么不就传递了一个错误的信息？例如，OnCmdMsg 函数得到 FALSE 返回值，就认为消息没有被处理，它将继续发送消息到下一环节。的确是这样的，但是这不是 MFC 的漏洞，而是有意这么设计的，用来处理一些特别的消息映射宏，实现同一个消息的多次处理。
通常的命令或者通知消息是没有返回值的（见 4.4.2 节的消息映射宏），仅仅一些特殊的消息处理函数具有返回值，这类消息的消息处理函数是使用扩展消息映射宏映射的，例如：
ON_COMMAND 对应的 ON_COMMAND_EX
扩展映射宏和对应的普通映射宏的参数个数相同，含义一样。但是扩展映射宏的消息处理函数的原型和对应的普通映射宏相比，有两个不同之处：一是多了一个 UINT 类型的参数，另外就是有返回值（返回 BOOL 类型）。回顾 4.4.2 章节，范围映射宏 ON_COMMAND_RANGE 的消息处理函数也有一个这样的参数，该参数在两处的含义是一样的，例如：命令消息扩展映射宏 ON_COMMAND_EX 定义的消息处理函数解释该参数是当前要处理的命令消息 ID。有返回值的意义在于：如果扩展映射宏的消息处理函数返回 FALSE，则导致当前消息被发送给消息路径上的下一个消息目标处理。
综合来看，ON_COMMAND_EX 宏有两个功能：
一是可以把多个命令消息指定给一个消息处理函数处理。这类似于 ON_COMMAND_RANGE 宏的作用。不过，这里的多条消息的命令 ID 或者控制子窗口 ID 可以不连续，每条消息都需要一个 ON_COMMAND_EX 宏。
二是可以让几个消息目标处理同一个命令或者通知或者反射消息。如果消息发送路径上较前的命令目标不处理消息或者处理消息后返回 FALSE，则下一个命令目标将继续处理该消息。

对于通知消息、反射消息，它们也有扩展映射宏，而且上述论断也适合于它们。例如：
ON_NOTIFY 对应的 ON_NOTIFY_EX
ON_CONTROL 对应的 ON_CONTROL_EX
ON_CONTROL_REFLECT 对应的 ON_CONTROL_REFLECT_EX
等等。

范围消息映射宏也有对应的扩展映射宏，例如：

ON_NOTIFY_RANGE 对应的 ON_NOTIFY_EX_RANGE
ON_COMMAND_RANGE 对应的 ON_COMMAND_EX_RANGE
使用这些宏的目的在于利用扩展宏的第二个功能：实现消息的多次处理。

关于扩展消息映射宏的例子，参见 13.2..4.4 节和 13.2.4.6 节。

4.4.3.3 一些消息处理类的 OnCmdMsg 的实现

从以上论述知道，OnCmdMsg 虚拟函数在 MFC 命令消息的发送中扮演了重要的角色，CFrameWnd 的 OnCmdMsg 实现了 MFC 的标准命令消息发送路径。
那么，就产生一个问题：如果命令消息不送给边框窗口对象，那么就不会有按标准命令发送路径发送消息的过程？答案是肯定的。例如一个菜单被一个对话框窗口所拥有，那么，菜单命令将送给 MFC 对话框窗口对象处理，而不是 MFC 边框窗口处理，当然不会和 CFrameWnd 的处理流程相同。
但是，有一点需要指出，一般标准的 SDI 和 MDI 应用程序，只有主边框窗口拥有菜单和工具条等用户接口对象，只有在用户与用户接口对象进行交互时，才产生命令，产生的命令必然是送给 SDI 或者 MDI 程序的主边框窗口对象处理。
下面，讨论几个 MFC 类覆盖 OnCmdMsg 虚拟函数时的实现。这些类的 OnCmdMsg 或者可能是标准 MFC 命令消息路径的一个环节，或者可能是一个独立的处理过程（对于其中的 MFC 窗口类）。
从分析 CView 的 OnCmdMsg 实现开始。

- CView 的 OnCmdMsg

```
CView::OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo)
```

首先，调用 *CWnd::OnCmdMsg*，结果是搜索当前视的类和基类的消息映射数组，搜索顺序是从下层到上层。若某一层实现了对命令消息 nID 的处理，则调用它的实现函数；否则，调用 m_pDocument->OnCmdMsg，把命令消息送给文档类处理。m_pDocument 是和当前视关联的文档对象指针。如果文档对象类实现了 OnCmdMsg，则调用它的覆盖函数；否则，调用基类(例如 CDocument)的 OnCmdMsg。
接着，讨论 CDocument 的实现。

- CDocument 的 OnCmdMsg

```
BOOL CDocument::OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo)
```

首先，调用 *CCmdTarget::OnCmdMsg*，导致当前对象(this)的类和基类的消息映射数组被搜索，看是否有对应的消息处理函数可用。如果有，就调用它；如果没有，则调用文档模板的 OnCmdMsg 函数（m_pTemplate->OnCmdMsg）把消息送给文档模板处理。
MFC 文档模板没有覆盖 OnCmdMsg，导致基类 CCmdTarget 的 OnCmdMsg 被调用，看是否有文档模板类或基类实现了对消息的处理。是的话，调用对应的消息处理函数，否则，返回 FALSE。从前面的分析知道，CCmdTarget 类的消息映射数组是空的，所以这里返回 FALSE。

- CDialog 的 OnCmdMsg

```
BOOL CDialog::OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo)
```

第一，调用 *CWnd::OnCmdMsg*，让对话框或其基类处理消息。

第二，如果还没有处理，而且是控制消息或系统命令或非命令按钮，则返回 FALSE，不作进一步处理。否则，调用父窗口的 OnCmdmsg(GetParent()->OnCmdmsg)把消息送给父窗口处理。

第三，如果仍然没有处理，则调用当前线程的 OnCmdMsg(GetThread()->OnCmdMsg)把消息送给线程对象处理。

第四，如果最后没有处理，返回 FALSE。

- CMDIFrameWnd 的 OnCmdMsg

对于 MDI 应用程序，MDI 主边框窗口首先是把命令消息发送给活动的 MDI 文档边框窗口进行处理。MDI 主边框窗口对 OnCmdMsg 的实现函数的原型如下：

```
BOOL CMDIFrameWnd::OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo)
```

第一，如果有激活的文档边框窗口，则调用它的 OnCmdMsg(MDIGetActive（）->OnCmdMsg)把消息交给它进行处理。MFC 的文档边框窗口类并没有覆盖 OnCmdMsg 函数，所以基类 CFrameWnd 的函数被调用，导致文档边框窗口的活动视、文档边框窗口本身、应用程序对象依次来进行消息处理。

第二，如果文档边框窗口没有处理，调用 CFrameWnd::OnCmdMsg 把消息按标准路径发送，重复第一次的步骤，不过对于 MDI 边框窗口来说不存在活动视，所以省却了让视处理消息的必要；接着让 MDI 边框窗口本身来处理消息，如果它还没有处理，则让应用程序对象进行消息处理——虽然这是一个无用的重复。

除了 CView、CDocument 和 CMDIFrameWnd 类，还有几个 OLE 相关的类覆盖了 OnCmdMsg 函数。OLE 的处理本书暂不涉及，CDialog::OnCmdMsg 将在对话框章节专项讨论其具体实现。

4.4.3.4 一些消息处理类的 OnCommand 的实现

除了虚拟函数 OnCmdMsg，还有一个虚拟函数 OnCommand 在命令消息的发送中占有重要地位。在处理命令或者通知消息时，OnCommand 被 MFC 窗口过程调用，然后它调用 OnCmdMsg 按一定路径传送消息。除了 CWnd 类和一些 OLE 相关类外，MFC 里主要还有 MDI 边框窗口实现了 OnCommand。

```
BOOL CMDIFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam)
```

第一，如果存在活动的文档边框窗口，则使用 AfxCallWndProc 调用它的窗口过程，把消息送给文档边框窗口来处理。这将导致文档边框窗口的 OnCmdMsg 作如下的处理：

活动视处理消息→与视关联的文档处理消息→本文档边框窗口处理消息→应用程序对象处理消息→文档边框窗口缺省处理

任何一个环节如果处理消息，则不再向下发送消息，处理终止。如果消息仍然没有被处理，就只有交给主边框窗口了。

第二，第一步没有处理命令，继续调用 *CFrameWnd::OnCommand*，将导致 CMDIFrameWnd 的 OnCmdMsg 被调用。从前面的分析知道，将再次把消息送给 MDI 边框窗口的活动文档边框窗口，第一步的过程除了文档边框窗口缺省处理外都将被重复。具体的处理过程见前文的 CMDIFrameWnd::OnCmdMsg 函数。

第三，对于 MDI 消息，如果主边框窗口还不处理的话，交给 CMDIFrameWnd 的 DefWindowProc 作缺省处理。

第四， 消息没有处理，返回 FALSE。

上述分析综合了 OnCommand 和 OnCmdMsg 的处理，它们是在 MFC 内部 MDI 边框窗口处理命令消息的完整的流程和标准的步骤。整个处理过程再次表明了边框窗口在处理命令消息时的中心作用。从程序员的角度来看，可以认为整个标准处理路径如下：

活动视处理消息→与视关联的文档处理消息→本文档边框窗口处理消息→应用程序对象处理消息→文档边框窗口缺省处理→MDI 边框窗口处理消息→MDI 边框窗口缺省处理

任何一个环节如果处理消息，不再向下发送消息，急处理终止。

4.4.4 对控制通知消息的接收和处理

4.4.4.1 WM_COMMAND 控制通知消息的处理

WM_COMMAND 控制通知消息的处理和 WM_COMMAND 命令消息的处理类似，但是也有不同之处。

首先，分析处理 WM_COMMAND 控制通知消息和命令消息的相似处。如前所述，命令消息和控制通知消息都是由窗口过程给 OnCommand 处理（参见 CWnd::OnWndMsg 的实现），OnCommand 通过 wParam 和 lParam 参数区分是命令消息或通知消息，然后送给 OnCmdMsg 处理（参见 CWnd::OnCommnd 的实现）。

其次，两者的不同之处是：

- 命令消息一般是送给主边框窗口的，这时，边框窗口的 OnCmdMsg 被调用；而控制通知消息送给控制子窗口的父窗口，这时，父窗口的 OnCmdMsg 被调用。
- OnCmdMsg 处理命令消息时，通过命令分发可以由多种命令目标处理，包括非窗口对象如文档对象等；而处理控制通知消息时，不会有消息分发的过程，控制通知消息最终肯定是由窗口对象处理的。

不过，在某种程度上可以说，控制通知消息由窗口对象处理是一种习惯和约定。当使用 ClassWizard 进行消息映射时，它不提供把控制通知消息映射到非窗口对象的机会。但是，手工地添加消息映射，让非窗口对象处理控制通知消息的可能是存在的。例如，对于 CFormView，一方面它具备接受 WM_COMMAND 通知消息的条件，另一方面，具备把 WM_COMMAND 消息派发给关联文档对象处理的能力，所以给 CFormView 的通知消息是可以让文档对象处理的。

事实上，BN_CLICKED 控制通知消息的处理和命令消息的处理完全一样，因为该消息的通知代码是 0， ON_BN_CLICKED(id， memberfunction) 和 ON_COMMAND(id，memberfunction)是等同的。

此外，MFC 的状态更新处理机制就是建立在通知消息可以发送给各种命令目标的基础之上的。关于 MFC 的状态更新处理机制，见后面 4.4.4.4 节的讨论。

- 控制通知消息可以反射给子窗口处理。OnCommand 判定当前消息是 WM_COMAND 通知消息之后，首先它把消息反射给控制子窗口处理，如果子窗口处理了反射消息，OnCommand 不会继续调用 OnCmdMsg 让父窗口对象来处理通知消息。

4.4.4.2 WM_NOTIFY 消息及其处理：

（1）WM_NOTIFY 消息

还有一种通知消息 WM_NOTIFY，在 Win32 中用来传递信息复杂的通知消息。WM_NOTIFY 消息怎么来传递复杂的信息呢？WM_NOTIFY 的消息参数 wParam 包含了发送通知消息的控制窗口 ID，另一个参数 lParam 包含了一个指针。该指针指向一个 NMHDR 结构，或者更大的结构，只要它的第一个结构成员是 NMHDR 结构。

NMHDR 结构：

```
typedef struct tagNMHDR {
HWND hwndFrom;
UINT idFrom;
UINT code;
} NMHDR;
```

上述结构有三个成员，分别是发送通知消息的控制窗口的句柄、ID 和通知消息代码。

举一个更大、更复杂的结构例子：列表控制窗发送 LVN_KEYDOWN 控制通知消息，则 lParam 包含了一个指向 LV_KEYDOWN 结构的指针。其结构如下：

```
typedef struct tagLV_KEYDOWN {
NMHDR hdr;
WORD wVKey;
UINT flags;
}LV_KEYDOWN;
```

它的第一个结构成员 hdr 就是 NMHDR 类型。其他成员包含了更多的信息：哪个键被按下，哪些辅助键(SHIFT、CTRL、ALT 等)被按下。

（2）WM_NOTIFY 消息的处理

在分析 CWnd::OnWndMsg 函数时，曾指出当消息是 WM_NOTIFY 时，它把消息传递给 OnNotify 虚拟函数处理。这是一个虚拟函数，类似于 OnCommand，CWnd 和派生类都可以覆盖该函数。OnNotify 的函数原型如下：

```
BOOL CWnd::OnNotify(WPARAM, LPARAM lParam, LRESULT* pResult)
```

参数 1 是发送通知消息的控制窗口 ID，没有被使用；参数 2 是一个指针；参数 3 指向一个 long 类型的数据，用来返回处理结果。

WM_NOTIFY 消息的处理过程如下：

第一，反射消息给控制子窗口处理。

第二，如果子窗口不处理反射消息，则交给 OnCmdMsg 处理。给 OnCmdMsg 的四个参数分别如下：第一个是命令消息 ID，第四个为空；第二个高阶 word 是 WM_NOTIFY，低阶 word 是通知消息；第三个参数是指向 AFX_NOTIFY 结构的指针。第二、三个参数有别于 OnCommand 送给 OnCmdMsg 的参数。

AFX_NOTIFY 结构：

```
struct AFX_NOTIFY
{
LRESULT* pResult;
NMHDR* pNMHDR;
};
```

pNMHDR 的值来源于参数 2 lParam，该结构的域 pResult 用来保存处理结果，域 pNMHDR

用来传递信息。

OnCmdMsg 后续的处理和 WM_COMMAND 通知消息基本相同，只是在派发消息给消息处理函数时，DispatchMsdMsg 的第五个参数 pExtra 指向 OnCmdMsg 传递给它的 AFX_NOTIFY 类型的参数，而不是空指针。这样，处理函数就得到了复杂的通知消息信息。

4.4.4.3 消息反射

（1）消息反射的概念

前面讨论控制通知消息时，曾经多次提到了消息反射。MFC 提供了两种消息反射机制，一种用于 OLE 控件，一种用于 Windows 控制窗口。这里只讨论后一种消息反射。

Windows 控制常常发送通知消息给它们的父窗口，通常控制消息由父窗口处理。但是在 MFC 里头，父窗口在收到这些消息后，或者自己处理，或者反射这些消息给控制窗口自己处理，或者两者都进行处理。如果程序员在父窗口类覆盖了通知消息的处理（假定不调用基类的实现），消息将不会反射给控制子窗口。这种反射机制是 MFC 实现的，便于程序员创建可重用的控制窗口类。

MFC 的 CWnd 类处理以下控制通知消息时，必要或者可能的话，把它们反射给子窗口处理：

WM_CTLCOLOR，

WM_VSCROLL，WM_HSCROLL，

WM_DRAWITEM，WM_MEASUREITEM，

WM_COMPAREITEM，WM_DELETEITEM，

WM_CHARTOITEM，WM_VKEYTOITEM，

WM_COMMAND、WM_NOTIFY。

例如，对 WM_VSCROLL、WM_HSCROLL 消息的处理，其消息处理函数如下：

```
void CWnd::OnHScroll(UINT, UINT, CScrollBar* pScrollBar)
{
    //如果是一个滚动条控制，首先反射消息给它处理
    if (pScrollBar != NULL && pScrollBar->SendChildNotifyLastMsg())
        return;        //控制窗口成功处理了该消息

    Default();
}
```

又如：在讨论 OnCommand 和 OnNofity 函数处理通知消息时，都曾经指出，它们首先调用 ReflectLastMsg 把消息反射给控制窗口处理。

为了利用消息反射的功能，首先需要从适当的 MFC 窗口派生出一个控制窗口类，然后使用 ClassWizard 给它添加消息映射条目，指定它处理感兴趣的反射消息。下面，讨论反射消息映射宏。

上述消息的反射消息映射宏的命名遵循以下格式：“ON”前缀+消息名+“REFLECT”后缀，例如：消息 WM_VSCROLL 的反射消息映射宏是 ON_WM_VSCROLL_REFECT。但是通知消息 WM_COMMAND 和 WM_NOTIFY 是例外，分别为 ON_CONTROL_REFLECT 和 ON_NOFITY_REFLECT 。状态更新通知消息的反射消息映射宏是

ON_UPDATE_COMMAND_UI_REFLECT。
消息处理函数的名字和去掉“WM_”前缀的消息名相同 ，例如 WM_HSCROLL 反射消息处理函数是 Hscroll。
消息处理函数的原型这里不一一列举了。
这些消息映射宏和消息处理函数的原型可以借助于 ClassWizard 自动地添加到程序中。ClassWizard 添加消息处理函数时，可以处理的反射消息前面有一个等号，例如处理 WM_HSCROLL 的反射消息，选择映射消息“=EN_HSC ROLL”。ClassWizard 自动的添加消息映射宏和处理函数到框架文件。

（2）消息反射的处理过程

如果不考虑有 OLE 控件的情况，消息反射的处理流程如下图所示：

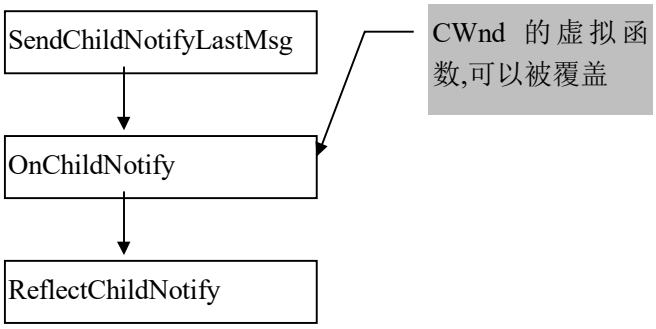


图 4-4　消息反射的处理流程

首先，调用 CWnd 的成员函数 SendChildNotifyLastMsg，它从线程状态得到本线程最近一次获取的消息（关于线程状态，后面第 9 章会详细介绍）和消息参数，并且把这些参数传递给函数 OnChildNotify。注意，当前的 CWnd 对象就是 MFC 控制子窗口对象。
OnChlidNofify 是 CWnd 定义的虚拟函数，不考虑 OLE 控制的话，它仅仅只调用 ReflectChildNotify。OnChlidNofify 可以被覆盖，所以如果程序员希望处理某个控制的通知消息，除了采用消息映射的方法处理通知反射消息以外，还可以覆盖 OnChlidNotify 虚拟函数，如果成功地处理了通知消息，则返回 TRUE。
ReflectChildNotify 是 CWnd 的成员函数，完成反射消息的派发。对于 WM_COMMAND，它直接调用 *CWnd::OnCmdMsg* 派发反射消息 WM_REFLECT_BASE+WM_COMMAND；对于 WM_NOTIFY ， 它 直 接 调 用 *CWnd::OnCmdMsg* 派 发 反 射 消 息 WM_REFLECT_BASE+WM_NOFITY；对于其他消息，则直接调用 *CWnd::OnWndMsg*（即 *CmdTarge::OnWndMsg*）派发相应的反射消息，例如 WM_REFLECT_BASE+WM_HSCROLL。
注意：ReflectChildNotify 直接调用了 CWnd 的 OnCmdMsg 或 OnWndMsg，这样反射消息被直接派发给控制子窗口，省却了消息发送的过程。
接着，控制子窗口如果处理了当前的反射消息，则返回反射消息被成员处理的信息。

（3）一个示例

如果要创建一个编辑框控制，要求它背景使用黄色，其他特性不变，则可以从 CEdit 派生一个类 CYellowEdit，处理通知消息 WM_CTLCOLOR 的反射消息。CYellowEdit 有三个属性，定义如下：

```
CYellowEdit::CYellowEdit()
{
m_clrText = RGB( 0, 0, 0 );
```

```
m_clrBkgnd = RGB( 255, 255, 0 );
m_brBkgnd.CreateSolidBrush( m_clrBkgnd );
}
```

使用 ClassWizard 添加反射消息处理函数：

函数原型：

```
afx_msg void HScroll();
```

消息映射宏：

```
ON_WM_CTLCOLOR_REFLECT()
```

函数的框架

```
HBRUSH CYellowEdit::CtlColor(CDC* pDC, UINT nCtlColor)
{
// TODO:添加代码改变设备描述表的属性
// TODO:  如果不再调用父窗口的处理，则返回一个非空的刷子句柄
return NULL;
}
```

添加一些处理到函数 CtlColor 中，如下：

```
pDC->SetTextColor( m_clrText );//设置文本颜色
pDC->SetBkColor( m_clrBkgnd );//设置背景颜色
return m_brBkgnd;                    //返回背景刷
```

这样，如果某个地方需要使用黄色背景的编辑框，则可以使用 CYellowEdit 控制。

4.4.5 对更新命令的接收和处理

用户接口对象如菜单、工具条有多种状态，例如：禁止，可用，选中，未选中，等等。这些状态随着运行条件的变化，由程序来进行更新。虽然程序员可以自己来完成更新，但是 MFC 框架为自动更新用户接口对象提供了一个方便的接口，使用它对程序员来说可能是一个好的选择。

4.4.5.1 实现方法

每一个用户接口对象，如菜单、工具条、控制窗口的子窗口，都由唯一的 ID 号标识，用户和它们交互时，产生相应 ID 号的命令消息。在 MFC 里，一个用户接口对象还可以响应 CN_UPDATE_COMMAND_UI 通知消息。因此，对每个标号 ID 的接口对象，可以有两个处理函数：一个消息处理函数用来处理该对象产生的命令消息 ID，另一个状态更新函数用来处理给该对象的 CN_UPDATE_COMMAND_UID 的通知消息。

使用 ClassWizard 可把状态更新函数加入到某个消息处理类，其结果是：

在类的定义中声明一个状态函数；

在消息映射中使用 ON_UPDATE_COMMAND_UI 宏添加一个映射条目；

在类的实现文件中实现状态更新函数的定义。

ON_UPDATE_COMMAND_UI 给指定 ID 的用户对象指定状态更新函数，例如：

ON_UPDATE_COMMAND_UI(ID_EDIT_COPY, OnUpdateEditCopy)

映射标识号 ID 为 ID_EDIT_COPY 菜单的通知消息 CN_UPDATE_COMMAND_UI 到函数

OnUpdateEditCopy。用于给 EDIT（编辑菜单）的菜单项 ID_EDIT_COPY（复制）添加一个状态处理函数 OnUpdateEditCopy，通过处理通知消息 CN_UPDATE_COMMAND_UI 实现该菜单项的状态更新。
状态处理函数的原型如下：
afxmsg void ClassName::OnUpdateEditPaste(CCmdUI* pCmdUI)
CCmdUI 对象由 MFC 自动地构造。在完善函数的实现时，使用 pCmdUI 对象和 CmdUI 的成员函数实现菜单项 ID_EDIT_COPY 的状态更新，让它变灰或者变亮，也就是禁止或者允许用户使用该菜单项。

4.4.5.2 状态更新命令消息

要讨论 MFC 的状态更新处理，先得了解一条特殊的消息。MFC 的消息映射机制除了处理各种 Windows 消息、控制通知消息、命令消息、反射消息外，还处理一种特别的“通知命令消息”，并通过它来更新菜单、工具栏（包括对话框工具栏）等命令目标的状态。
这种“通知命令消息”是 MFC 内部定义的，消息 ID 是 WM_COMMAND，通知代码是 CN_UPDATE_COMMAND_UI（0XFFFFFFFF）。
它不是一个真正意义上的通知消息，因为没有控制窗口产生这样的通知消息，而是 MFC 自己主动产生，用于送给工具条窗口或者主边框窗口，通知它们更新用户接口对象的状态。
它和标准 WM_COMMAND 命令消息也不相同，因为它有特定的通知代码，而命令消息通知代码是 0。
但是，从消息的处理角度，可以把它看作是一条通知消息。如果是工具条窗口接收该消息，则在发送机制上它和 WM_COMMAND 控制通知消息是相同的，相当于让工具条窗口处理一条通知消息。如果是边框窗口接收该消息，则在消息的发送机制上它和 WM_COMMAND 命令消息是相同的，可以让任意命令目标处理该消息，也就是说边框窗口可以把该条通知消息发送给任意命令目标处理。
从程序员的角度，可以把它看作一条“状态更新命令消息”，像处理命令消息那样处理该消息。每条命令消息都可以对应有一条“状态更新命令消息”。ClassWizard 也支持让任意消息目标处理“状态更新命令消息”（包括非窗口命令目标），实现用户接口状态的更新。
在这条消息发送时，通过 OnCmdMsg 的第三个参数 pExtra 传递一些信息，表示要更新的用户接口对象。pExtra 指向一个 CCmdUI 对象。这些信息将传递给状态更新命令消息的处理函数。
下面讨论用于更新用户接口对象状态的类 CCmdUI。

4.4.5.3 类 CCmdUI

CCmdUI 不是从 CObject 派生，没有基类。

（1） 成员变量

m_nID	用户接口对象的 ID
m_nIndex	用户接口对象的 index
m_pMenu	指向 CCmdUI 对象表示的菜单
m_pSubMenu	指向 CCmdUI 对象表示的子菜单
m_pOther	指向其他发送通知消息的窗口对象
m_pParentMenu	指向 CCmdUI 对象表示的子菜单

（2） 成员函数

Enable(BOOL bOn = TRUE) 禁止用户接口对象或者使之可用

SetCheck(int nCheck = 1) 标记用户接口对象选中或未选中

SetRadio(BOOL bOn = TRUE)

SetText(LPCTSTR lpszText)

ContinueRouting()

还有一个 MFC 内部使用的成员函数：

DoUpdate(CCmdTarget* pTarget, BOOL bDisableIfNoHndler)

其中，参数 1 指向处理接收更新通知的命令目标，一般是边框窗口；参数 2 指示如果没有提供处理函数（例如某个菜单没有对应的命令处理函数），是否禁止用户对象。

DoUpdate 作以下事情：

首先，发送状态更新命令消息给参数 1 表示的命令目标：调用 pTarget->OnCmdMsg（m_nID, CN_UPDATE_COMMAND_UI, this, NULL) 发送 m_nID 对象的通知消息 CN_UPDATE_COMMAND_UI。OnCmdMsg 的参数 3 取值 this，包含了当前要更新的用户接口对象的信息。

然后，如果参数 2 为 TRUE，调用 pTarget->OnCmdMsg(m_nID, CN_COMMAND, this, &info) 测试命令消息 m_nID 是否被处理。这时，OnCmdMsg 的第四个参数非空，表示仅仅是测试，不是真的要派发消息。如果没有提供命令消息 m_nID 的处理函数，则禁止用户对象 m_nID，否则使之可用。

从上面的讨论可以知道：通过其结构，一个 CCmdUI 对象标识它表示了哪一个用户接口对象，如果是菜单接口对象，pMenu 表示了要更新的菜单对象；如果是工具条，pOther 表示了要更新的工具条窗口对象，nID 表示了工具条按钮 ID。

所以，由参数上状态更新消息的消息处理函数就知道要更新什么接口对象的状态。例如，第 1 节的函数 OnUpdateEditPaste，函数参数 pCmdUI 表示一个菜单对象，需要更新该菜单对象的状态。

通过其成员函数，一个 CCmdUI 可以更新、改变用户接口对象的状态。例如，CCmdUI 可以管理菜单和对话框控制的状态，调用 Enable 禁止或者允许菜单或者控制子窗口，等等。所以，函数 OnUpdateEditPaste 可以直接调用参数的成员函数（如 pCmdUI->Enable）实现菜单对象的状态更新。

由于接口对象的多样性，其他接口对象将从 CCmdUI 派生出管理自己的类来，覆盖基类的有关成员函数如 Enable 等，提供对自身状态更新的功能。例如管理状态条和工具栏更新的 CStatusCmdUI 类和 CToolCmdUI 类。

4.4.5.4 自动更新用户接口对象状态的机制

MFC 提供了分别用于更新菜单和工具条的两种途径。

（1） 更新菜单状态

当用户对菜单如 File 单击鼠标时，就产生一条 WM_INITMENUPOPUP 消息，边框窗口在菜单下拉之前响应该消息，从而更新该菜单所有项的状态。

在应用程序开始运行时，边框也会收到 WM_INITMENUPOPUP 消息。

（2） 更新工具条等状态

当应用程序进入空闲处理状态时，将发送 WM_IDLEUPDATECMDUI 消息，导致所有的工具条用户对象的状态处理函数被调用，从而改变其状态。WM_IDLEUPDATECMDUI 是 MFC

自己定义和使用的消息。
在窗口初始化时，工具条也会收到 WM_IDLEUPDATECMDUI 消息。

（3） 菜单状态更新的实现

MFC 让边框窗口来响应 WM_INITMENUPOPUP 消息，消息处理函数是 OnInitMenuPopup，其原型如下：

```
afx_msg void CFrameWnd::OnInitMenuPopup( CMenu* pPopupMenu,
    UINT nIndex, BOOL bSysMenu );
```

第一个参数指向一个 CMenu 对象，是当前按击的菜单；第二个参数是菜单索引；第三个参数表示子菜单是否是系统控制菜单。
函数的处理：
如果是系统控制菜单，不作处理；否则，创建 CCmdUI 对象 state，给它的各个成员如 m_pMenu，m_pParentMenu，m_pOther 等赋值。
对该菜单的各个菜单项，调函数 state.DoUpdate，用 CCmdUI 的 DoUpdate 来更新状态。DoUpdate 的第一个参数是 this，表示命令目标是边框窗口；在 CFrameWnd 的成员变量 m_bAutoMenuEnable 为 TRUE 时（表示如果菜单 m_nID 没有对应的消息处理函数或状态更新函数，则禁止它），把 DoUpdate 的第二个参数 bDisableIfNoHndler 置为 TRUE。
顺便指出，m_bAutoMenuEnable 缺省时为 TRUE，所以，应用程序启动时菜单经过初始化处理，没有提供消息处理函数或状态更新函数的菜单项被禁止。

（4） 工具条等状态更新的实现

图 4-5 表示了消息空闲时 MFC 更新用户对象状态的流程：
MFC 提供的缺省空闲处理向顶层窗口(框架窗口)的所有子窗口发送消息 WM_IDLEUPDATECMDUI；MFC 的控制窗口（工具条、状态栏等）实现了对该消息的处理，导致用户对象状态处理函数的调用。

虽然两种途径调用了同一状态处理函数，但是传递的 CCmdUI 参数从内部构成上是不一样的：第一种传递的 CCmdUI 对象表示了一菜单对象，(pMenu 域被赋值)；第二种传递了一个窗口对象(pOther 域被赋值)。同样的状态改变动作，如禁止、允许状态的改变，前者调用了 CMenu 的成员函数 EnableMenuItem，后者使用了 CWnd 的成员函数 EnabelWindow。但是，这些不同由 CCmdUI 对象内部区分、处理，对用户是透明的：不论菜单还是对应的工具条，用户都用同一个状态处理函数使用同样的形式来处理。

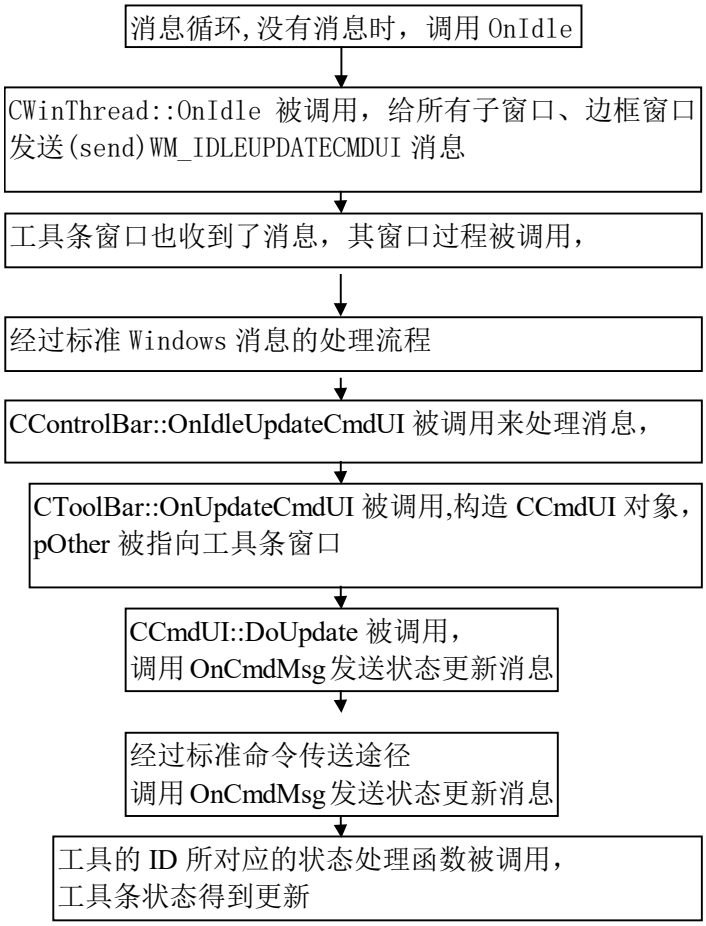


图 4-5　MFC 命令目标状态的更新

这一节分析了用户界面更新的原理和机制。在后面第 13 章讨论工具条和状态栏时，将详细的分析这种机制的具体实现。

4.5　消息的预处理

到现在为止，详细的讨论了 MFC 的消息映射机制。但是，为了提高效率和简化处理，MFC 提供了一种消息预处理机制，如果一条消息在预处理时被过滤掉了（被处理），则不会被派发给目的窗口的窗口过程，更不会进入消息循环了。

显然，能够进行预处理的消息只可能是队列消息，而且必须在消息派发之前进行预处理。因此，MFC 在实现消息循环时，对于得到的每一条消息，首先送给目的窗口、其父窗口、其祖父窗口乃至最顶层父窗口，依次进行预处理，如果没有被处理，则进行消息转换和消息派发，如果某个窗口实现了预处理，则终止。有关实现见后面关于 CWinThread 线程类的章节，CWinThread 的 Run 函数和 PreTranslateMessage 函数以及 CWnd 的函数 WalkPreTranslateTree 实现了上述要求和功能。这里要讨论的是 MFC 窗口类如何进行消息预处理。

CWnd 提供了虚拟函数 PreTranslateMessage 来进行消息预处理。CWnd 的派生类可以覆盖该函数，实现自己的预处理。下面，讨论几个典型的预处理。

首先，是 CWnd 的预处理：

预处理函数的原型为：

BOOL CWnd::PreTranslateMessage(MSG* pMsg)

CWnd 类主要是处理和过滤 Tooltips 消息。关于该函数的实现和 Tooltips 消息，见后面第 13 章关于工具栏的讨论。

然后，是 CFrameWnd 的预处理：

CFrameWnd 除了调用基类 CWnd 的实现过滤 Tooltips 消息之外，还要判断当前消息是否是键盘快捷键被按下，如果是，则调用函数::TranslateAccelerator(m_hWnd, hAccel, pMsg)处理快捷键。

接着，是 CMDIChildWnd 的预处理：

CMDIChildWnd 的预处理过程和 CFrameWnd 的一样，但是不能依靠基类 CFrameWnd 的实现，必须覆盖它。因为 MDI 子窗口没有菜单，所以它必须在 MDI 边框窗口的上下文中来处理快捷键，它调用了函数::TranslateAccelerator(GetMDIFrame()->m_hWnd, hAccel, pMsg)。

讨论了 MDI 子窗口的预处理后，还要讨论 MDI 边框窗口：

CMDIFrameWnd 的实现除了 CFrameWnd 的实现的功能外，它还要处理 MDI 快捷键（标准 MDI 界面统一使用的系统快捷键）。

在后面，还会讨论 CDialog、CFormView、CToolBar 等的消息预处理及其实现。

至于 CWnd::WalkPreTranslateTree 函数，它从接受消息的窗口开始，逐级向父窗回溯，逐一对各层窗口调用 PreTranslateMessage 函数，直到消息被处理或者到最顶层窗口为止。

4.6 MFC 消息映射的回顾

从处理命令消息的过程可以看出，Windows 消息和控制消息的处理要比命令消息的处理简单，因为查找消息处理函数时，后者只要搜索当前窗口对象(this 所指)的类或其基类的消息映射入口表。但是，命令消息就要复杂多了，它沿一定的顺序链查找链上的各个命令目标，每一个被查找的命令目标都要搜索它的类或基类的消息映射入口表。

MFC 通过消息映射的手段，以一种类似 C++虚拟函数的概念向程序员提供了一种处理消息的方式。但是，若使用 C++虚拟函数实现众多的消息，将导致虚拟函数表极其庞大；而使用消息映射，则仅仅感兴趣的消息才加入映射表，这样就要节省资源、提高效率。这套消息映射机制的基础包括以下几个方面：

第一、消息映射入口表的实现：采用了 C++静态成员和虚拟函数的方法来表示和得到一个消息映射类(CCmdTarget 或派生类)的映射表。

第二、消息查找的实现：从低层到高层搜索消息映射入口表，直至根类 CCmdTarget。

第三、消息发送的实现：主要以几个虚拟函数为基础来实现标准 MFC 消息发送路径：OnComamnd、OnNotify、OnWndMsg 和 OnCmdMsg。、

OnWndMsg 是 CWnd 类或其派生类的成员函数，由窗口过程调用。它处理标准的 Windows 消息。

OnCommand 是 CWnd 类或其派生类的成员函数，由 OnWndMsg 调用来处理 WM_COMMAND 消息，实现命令消息或者控制通知消息的发送。如果派生类覆盖该函数，则必须调用基类的实现，否则将不能自动的处理命令消息映射，而且必须使用该函数接受的参数（不是程序员给定值）调用基类的 OnCommand。

OnNotify 是 CWnd 类或其派生类的成员函数，由 OnWndMsg 调用来处理 WM_NOTIFY 消息，实现控制通知消息的发送。
OnCmdMsg 是 CCmdTarget 类或其派生类的成员函数。被 OnCommand 调用，用来实现命令消息发送和派发命令消息到命令消息处理函数。

自动更新用户对象状态是通过 MFC 的命令消息发送机制实现的。
控制消息可以反射给控制窗口处理。
队列消息在发送给窗口过程之前可以进行消息预处理，如果消息被 MFC 窗口对象预处理了，则不会进入消息发送过程。

第5章　MFC 对象的创建

前面几章介绍了 MFC 的核心概念和思想，即介绍了 MFC 对 Windows 对象的封装方法和特点；MFC 对象的动态创建、序列化；MFC 消息映射机制。
现在，考查 MFC 的应用程序结构体系，即以文档-视为核心的编程模式。学习本章，应该弄清楚以下问题：
MFC 中诸多 MFC 对象的关系：应用程序对象，文档对象，边框窗口对象，文档边框窗口对象，视对象，文档模板对象等。
MFC 对象的创建和销毁：由什么对象创建或销毁什么对象，何时创建，何时销毁？
MFC 提供了那些接口来支持其编程模式？

5.1　MFC 对象的关系

5.1.1 创建关系

这里讨论应用程序、文档模板、边框窗口、视、文档等的创建关系。图 5-1 大略地表示了创建顺序，但表 5-1 更直接地显示了创建与被创建的关系。

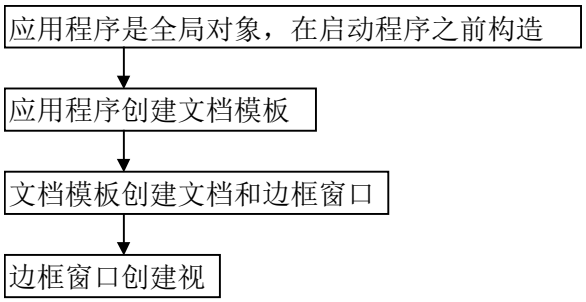


图 5-1 应用程序创建 MFC 对象的顺序

表5-1 MFC对象的创建关系

创建者	被创建的对象
应用程序对象	文档模板
文档模板	文档
文档模板	边框窗口
边框窗口	视

5.1.2 交互作用关系

应用程序对象有一个文档模板列表，存放一个或多个文档模板对象；文档模板对象有一个打开文档列表，存放一个或多个已经打开的文档对象；文档对象有一个视列表，存放显示该文档数据的一个或多个视对象；还有一个指针指向创建该文档的文档模板对象；视有一个指向其关联文档的指针，视是一个子窗口，其父窗口是边框窗口（或者文档边框窗口）；文档边框窗口有一个指向其当前活动视的指针；文档边框窗口是边框窗口的子窗口。

Windows 管理所有已经打开的窗口，把消息或事件发送给目标窗口。通常，命令消息发送给主边框窗口。

图 5-2 大略地表示了上述关系：

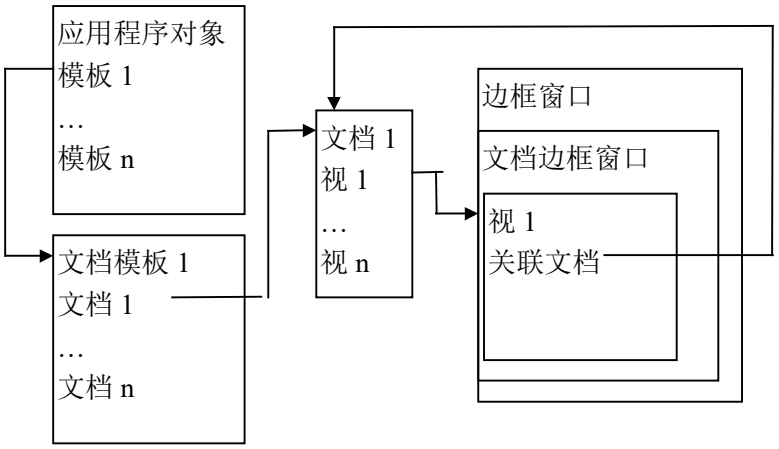


图 5-2　构成一个应用程序的对象

MFC 提供了一些函数来维护这些关系。

表 5-2 列出了从一个对象得到相关对象的方法。

表5-2 从一个对象得到另一个对象的方法

本对象	要得到的对象	使用的成员函数
CDocument 对象	视列表	GetFirstViewPosition GetNextView
	文档模板	GetDocTemplate
CView 对象	文档对象	GetDocument
	边框窗口	GetParentFrame
CMDIChildWnd 或 CFrameWnd 对象	活动视	GetActiveView
	活动视的文档	GetActiveDocument
CMDIFrameWnd 对象	活动文档边框窗口	MDIGetActive

表5-3 从一个对象通知另一个对象的方法：

本对象	要通知的对象/动作	使用的成员函数
CView 对象	通知文档更新所有视	CDocument::UpdateAllViews
CDocument 对象	更新一个视	CView::OnUpdate
CFrameWnd 或 CMDIFrameWnd 对象	通知一个视为活动视	CView::OnActivateView
	设置一个视为活动视	SetActivateView

可以通过表 5-2 得到相关对象，再调用表 5-3 中相应的函数。例如：视在接受了新数据或者数据被修改之后，使用表 5-2 中的函数 GetDocument 得到关联文档对象，然后调用表 5-3 中

的文档函数 UpdateAllViews 更新其他和文档对象关联的视。
在表 5-2 和表 5-3 中，CView 对象指 CView 或派生类的实例；成员函数列中如果没有指定类属，就是第一列对象的类的成员函数。

5.2 MFC 提供的接口

MFC 编程就是把一些应用程序特有的东西填入 MFC 框架。MFC 提供了两种填入的方法：一种就是使用前一章论述的消息映射，消息映射给应用程序的各种对象处理各种消息的机会；另一种就是使用虚拟函数，MFC 在实现许多功能或者处理消息、事件的过程中，调用了虚拟函数来完成一些任务，这样就给了派生类覆盖这些虚拟函数实现特定处理的机会。
下面两节将列出两类接口，有两个目的：一是为了让读者获得整体印象，二是后文将涉及到或者讨论其中的许多函数时，不显得突兀。

5.2.1 虚拟函数接口

几乎每一个 MFC 类都定义和使用了虚拟成员函数，程序员可以在派生类中覆盖它们。一般，MFC 提供了这些函数的缺省实现，所以覆盖函数应该调用基类的实现。这里给出一个 MFC 常用虚拟函数的总览表(见表 5-4)，更详细的信息或它们的缺省实现动作参见 MFC 文档。由于基类的虚拟函数被派生类继承，所以在派生类中不作重复说明。
覆盖基类的虚拟函数可以通过 ClassWizard 进行，不过，并非所有的函数都可以这样，有的必须手工加入函数声明和实现。

表5-4 常见MFC类的虚拟函数接口

类	虚拟函数	覆盖的目的和功能
CCmdTarget	OnCmdMsg	发送、派发命令消息
	OnFinalRelease	OLE 用途，引用为 0 时作清理工作
CWinThread	ExitInstance	在线程退出时作清理工作
	InitInstance	在线程开始时作初始化
	OnIdle	执行 thread-specific idle-time 处理
	PreTranslateMessage	在消息送给 Windows 函数 TranslateMessage and DispatchMessage.之前进行消息过滤
	IsIdleMessage	检查是否是某个特别的消息
	ProcessWndProcException	截获线程消息/命令处理中的例外
	ProcessMessageFilter	线程消息过滤
	Run	实现线程特定的消息循环
CWinApp	HideApplication	关闭所有的窗口之前隐藏应用程序
	CloseAllDocument	退出程序之前关闭所有文档

转下页

续表

	SaveModifiedDocument	框架窗口关闭时用来保存文档
	DoMessageBox	实现客户化的 messagebox
	DoWaitCursor	关闭或打开等待光标
	OnDDeCommand	响应 DDE 命令
	WinHelp	调用 WinHelp 函数
CWnd	WindowProc	提供一个窗口过程
	DefWindowProc	为应用程序不处理的消息提供缺省处理
	PostNcDestroy	在窗口销毁之后被消息处理函数 OnNcDestroy 调用
	OnNotify	处理通知消息 WM_NOTIFY
	OnChildNotify	父窗口调用它给控制子窗口一个机会来处理通知反射消息
	DoDataExchange	Updata 调用它来进行对话框数据交换和验证
CFrameWnd	GetMessageBar	返回一个指向框架窗口的状态条的指针
	OnCreateClient	创建框架的客户窗口
	OnSetPreviewMode	设置程序的主框架窗口进入或退出打印预览模式
	NegotiateBorderSpace	协调边框窗口的边框空间的大小(OLE 用途)
CMDIFrameWnd	CreateClient	创建 CMDIFrameWnd 的 MDICLIENT 窗，被 CWnd 的消息处理函数 OnCreate 调用.

转下页

续表

	GetWindowMenuPopup	返回窗口的弹出式菜单
CDialog	OnInitDialog	对话框窗口的初始化
	OnSetFont	设置对话框控制的文本字体
	OnOK	模式对话框的 OK 按钮按下后进行的处理
	OnCancel	模式对话框的 CANCEL 按钮按下后进行的处理
CView	IsSelected	测试是否有一个文档被选择(OLE 支持)
	OnActivateView	视窗口激活时调用
	OnActivateFrame	当包含视窗口的框架窗口变成活动或非活动窗口时调用
	OnBeginPrinting	打印工作开始时调用，用来分配 GDI 资源
	OnDraw	用来屏幕显示、打印、打印预览文档内容
	OnEndPrinting	打印工作结束时调用，释放 GDI 资源
	OnEndPrintPreview	退出打印预览模式时调用
	OnPrepareDC	OnDraw 或 OnPrint 之前调用，用来准备设备描述表
	OnPreparePrinting	文档打印或者打印预览前调用，可用来初始化打印对话框
	OnPrint	用来打印或打印预览文档
	OnUpdate	用来通知一个视的关联文档内容已经变化
CDocTemplate	MatchDocType	确定文档类型和文档模板匹配时的可信程度

转下页

续表

	CreateNewDocument	创建一个新的文档
	CreateNewFrame	创建一个包含文档和视的框架窗口
	InitialUpdateFrame	初始化框架窗口，必要时使它可见
	SaveAllModified	保存所有和模板相关的而且修改了的文档
	CloseAllDocuments	关闭所有和模板相关的文档
	OpenDocumentFile	打开指定路径的文件
	SetDefaultTitle	设置文档窗口缺省显示的标题
CDocument	CanCloseFrame	在关闭显示该文档的边框窗口之前调用
	DeleteContents	用来清除文档的内容
	OnChangedViewList	在与文档关联的视图被移走或新加入时调用
	OnCloseDocument	用来关闭文档
	OnNewDocument	用来创建新文档
	OnOpenDocument	用来打开文档
	OnSaveDocument	以来保存文档
	ReportSaveLoadException	处理打开、保存文档操作失败时的例外
	GetFile	返回一个指向 Cfile 对象的指针
	ReleaseFile	释放一个文件以便其他应用程序可以使用
	SaveModified	用来询问用户文档是否需要保存
	PreCloseFrame	在框架窗口关闭之前调用

5.2.2 消息映射方法和标准命令消息

窗口对象可以响应以“WM_”为前缀的标准 Windows 消息，消息处理函数名称以“ON”为前缀。不同类型的 Windows 窗口处理的 Windows 消息是有所不同的，因此，不同类型的 MFC 窗口实现的消息处理函数也有所不同。例如，多文档边框窗口能处理 WM_MDIACTIVATE 消息，其他类型窗口就不能。程序员从一定的 MFC 窗口派生自己的窗口类，对感兴趣的消息，覆盖基类的消息处理函数，实现自己的消息处理函数。

所有的命令目标（CCmdTarger 或导出类对象）可以响应命令消息，程序员可以指定应用程序对象、框架窗口对象、视对象或文档对象等来处理某条命令消息。一般地，尽量由与命令消息关系密切的对象来处理，例如隐藏/显示工具栏由框架窗口处理，打开文件由应用程序对象处理，数据变化的操作由文档对象处理。

对话框的控制子窗口可以响应各类通知消息。

对于命令消息，MFC 实现了一系列标准命令消息处理函数。标准命令 ID 在 afxres.h 中定义。表 5-5 列出了 MFC 标准命令的实现，从 ID 或者函数名可以大致地看出该函数的目的、功用，具体的实现有的后续章节会讲解，详细参见 MFC 技术文档。

程序员可以自己来处理这些标准消息，也可以通过不同的类或从不同的类导出自己的类来处理这些消息，不过最好遵循 MFC 的缺省实现。比如处理 ID_FILE_NEW 命令，最好由 CWinApp 的派生类处理。

表5-5 标准命令消息处理函数

ID	函数	实现函数的类
ID_FILE_NEW	OnFileNew	CWinApp
ID_FILE_OPEN	OnFileOpen	CWinApp
ID_FILE_CLOSE	OnFileClose	CDocument
ID_FILE_SAVE	OnFileSave	CDocument
ID_FILE_SAVE_AS	OnFileSaveAs	CDocument
ID_FILE_SAVE_COPY_AS	OnFileSaveCopyAs	COleServerDoc
ID_FILE_UPDATE	OnUpdateDocument	COleServerDoc
ID_FILE_PAGE_SETUP	OnFilePrintSetup	CWinApp

转下页

续表

ID_FILE_PRINT	OnFilePrint	CView
ID_FILE_PRINT_PREVIEW	OnFilePrintPreview	CView
ID_FILE_MRU_FILE1...FILE16	OnUpdateRecentFileMenu	CWinApp
ID_EDIT_CLEAR		CView 没有实现，但是，如果有实现函数，就是派生类 CEditView 的实现函数
ID_EDIT_CLEAR_ALL		
ID_EDIT_COPY		
ID_EDIT_CUT		
ID_EDIT_FIND		
ID_EDIT_PASTE_LINK		
ID_EDIT_PASTE_SPECIAL		
ID_EDIT_REPEAT		
ID_EDIT_REPLACE		
ID_EDIT_SELET_ALL		
ID_EDIT_UNDO		
ID_WINDOW_NEW	OnWindowNew	CMDIFrameWnd
ID_WINDOW_ARRANGE	OnMDIWindowCmd	CMDIFrameWnd
ID_WINDOW_CASCADE		
ID_WINDOW_TILE_HORZ		
ID_WINDOW_TILE_VERT		
ID_WINDOW_SPLIT		CSplitterWnd
ID_APP_ABOUT		
ID_APP_EXIT	OnAppExit	CWinApp
ID_HELP_INDEX	OnHelpIndex	CWinApp
ID_HELP_USING	OnHelpUsing	CWinApp
ID_CONTEXT_HELP	OnContextHelp	CWinApp

转下页

续表

ID_HELP	OnHelp	CWinApp
ID_DEFAULT_HELP	OnHelpIndex	CWinApp
ID_NEXT_PANE	OnNextPaneCmd	CSplitterWnd
ID_PREV_PANE	OnNextPaneCmd	CSplitterWnd
ID_OLE_INSERT_NEW		
ID_OLE_EDIT_LINKS		
ID_OLE_VERB_FIRST...LAST		
ID_VIEW_TOOLBAR		CFrameWnd
ID_VIEW_STATUS_BAR		CFrameWnd
ID_INDICATOR_CAPS ID_INDICATOR_NUM ID_INDICATOR_SCRL ID_INDICATOR_KANA	OnUpdateKeyIndicator	CFrameWnd

5.3 MFC 对象的创建过程

应用程序使用 MFC 的接口是把一些自己的特殊处理填入 MFC 框架，这些处理或者在应用程序启动和初始化的时候被调用，或者在程序启动之后和用户交互的过程中被调用，或者在程序退出和作清理工作的时候被调用。这三个阶段中，和用户交互阶段是各个程序自己的事情，自然都不一样，但是程序的启动和退出两个阶段是 MFC 框架所实现的，是 MFC 框架的一部分，各个程序都遵循同样的步骤和规则。显然，清楚 MFC 框架对这两个阶段的处理是很有必要的，它可以帮助深入理解 MFC 框架，更好地使用 MFC 框架，更有效地实现应用程序特定的处理。

MFC 程序启动和初始化过程就是创建 MFC 对象和 Windows 对象、建立各种对象之间的关系、把窗口显示在屏幕上的过程，退出过程就是关闭窗口、销毁所创建的 Windows 对象和 MFC 对象的过程。所以，下面要讨论几种常用 MFC 对象的结构，它们是构成一个文档-视模式应用程序的重要部件。

5.3.1 应用程序中典型对象的结构

本节将主要分析应用程序对象、文档对象、文档模板等的数据结构。通过考察类的结构，特别是成员变量结构，弄清它的功能、目的以及和其他类的关系；另外，在后续有关分析中必定会提到这些成员变量，这里先作个说明，到时也不会显得突兀。

下面几节以表格的形式来描述各个类的成员变量。表格中，第一列打钩的表示是 MFC 类库文档有说明的；没打钩的在文档中没有说明，如果是 public，则可以直接访问，但随着 MFC 版本的变化，以后 MFC 可能不支持这些成员；第二列是访问属性；第三列是成员变量名称；第四列是成员变量的数据类型；第五列是对成员变量的功能、用途的简要描述。

5.3.1.1 应用程序类的成员变量

应用程序对象的数据成员表由两部分组成，第一部分是 CWinThread 的成员变量，如表 5-6 所示，CWinApp 继承了 CWinThread 的数据成员。第二部分是 CWinApp 自己定义的成员变量，如表 5-7 所示。

表5-6 CwinThread的成员变量

	访问限制	变量名称	类型	解释
√	public	m_bAutoDelete	BOOL	指定线程结束时是否销毁线程对象本身
√	public	m_hThread	HANDLE	当前线程的句柄
√	public	m_nThreadID	UINT	当前线程的 ID
√	public	m_pMainWnd	CWnd*	指向应用程序主窗口的指针
√	public	m_pActiveWnd	CWnd*	当 OLE SERVER 就地激活时指向客户程序主窗口的指针
	public	m_msgCur	MSG	当前消息(MSG 结构)
	public	m_pThreadParams	LPVOID	传递给线程开始函数的参数
	public	m_pfnThreadProc	函数指针 1	线程开始函数，AFX_THREADPROC 类型
	public	m_lpfnOleTermOrFreeLib	函数指针 2	OLE 用途，void (AFXAPI * fn)(BOOL,BOOL)
	public	m_pMessageFilter	指针	OLE 消息过滤，指向 COleMessageFilter 对象
	protected	m_ptCursorLast	CPoint	最新鼠标位置
	protected	m_nMsgLast	UINT	消息队列中最新接收到的消息

表5-7 CWinApp的成员变量

	访问限制	变量名称	类型	解释
√	public	m_pszAppName	LPCTSTR	应用程序名称
√	public	m_hInstance	HINSTANCE	标志应用程序当前实例句柄
√	public	m_hPrevInstance	HINSTANCE	32 位程序设为空
√	public	m_lpCmdLine	LPTSTR	指向应用程序的命令行字符串
√	public	m_nCmdShow	int	指定窗口开始的显示方式
√	public	m_bHelpMode	BOOL	标识用户是否在上下文帮助模式
√	public	m_pszExeName	LPCTSTR	应用程序的模块名
√	public	m_pszHelpFilePath	LPCTSTR	应用程序的帮助文件名，缺省时同模块名
√	public	m_pszProfileName	LPCTSTR	应用程序的 INI 文件名，缺省

				时同应用程序名
√	public	m_pszRegistryKey	LPCTSTR	Register 入口，如果不指定，使用 INI 文件。
	public	m_pDocManager;	CDocManager *	指向一个文档模板管理器
	protected	m_hDevMode	HGLOBAL	打印设备模式
	protected	m_hDevNames	HGLOBAL	打印设备名称
	protected	m_dwPromptContext	DWORD	被 MESSAGE BOX 覆盖的帮助上下文
	protected	m_nWaitCursorCount	int	等待光标计数
	protected	m_hcurWaitCursorRestore	HCURSOR	保存的光标，在等待光标之后恢复
	protected	m_pRecentFileList	指针	指向 CRecentFileList 对象，最近打开的文件列表
	public	m_atomApp	ATOM	DDE 用途
	public	m_atomSystemTopic	m_atomApp	DDE 用途
	public	m_nNumPreviewPages	UINT	缺省被打印的页面
	public	m_nSafetyPoolSize	size_t	理想尺寸
	public	m_lpfnDaoTerm	函数指针	DAO 初始化设置时使用

5.3.1.2 CDocument 的成员变量

表5-8 文档对象的属性。

	访问限制	变量名称	类型	解释
	protected	m_strTitle	CString	文档标题
	protected	m_strPathName	CString	文档路径
	protected	m_pDocTemplate	CDocTemplate*	指向文档模板的指针
	protected	m_viewList	CPtrList	关联的视窗口列表
	protected	m_bModified	BOOL	文档是否有变化、需要存盘
	public	m_bAutoDelete	BOOL	关联视都关闭时是否删除文档对象
	public	m_bEmbedded	BOOL	文档是否由 OLE 创建

5.3.1.3 文档模板的属性

表 5-9 列出了文档模板的成员变量，5-10 列出了单文档模板的成员变量，5-11 列出了多文档模板的成员变量。单、多文档模板继承了文档模板的成员变量。

表5-9 文档模板的数据成员

	访问限制	变量名称	类型	解释
	public	m_bAutoDelete	BOOL	
	public	m_pAttachedFactory	CObject *	
	public	m_hMenuInPlace	HMENU	就地激活时，OLE 客户程序的菜单
	public	m_hAccelInPlace	HACCEL	就地激活时，OLE 客户程序的快捷键
	public	m_hMenuEmbedding	HMENU	
	public	m_hAccelEmbedding	HACCEL	
	public	m_hMenuInPlaceServer	HMENU	
	public	m_hAccelInPlaceServer	HACCEL	
	protected	m_nIDResource	UINT	框架、菜单、快捷键等的资源 ID
	protected	m_nIDServerResource	UINT	
	public	m_nIDEmbeddingResource	UINT	
	public	m_nIDContainerResource	UINT	
	public	m_pDocClass	CRuntimeClass*	指向文档类的动态创建信息
	public	m_pFrameClass	CRuntimeClass*	指向框架类的动态创建信息
	public	m_pViewClass	CRuntimeClass*	指向视类的动态创建信息，由字符串 m_nIDResource 描述
	public	m_pOleFrameClass	CRuntimeClass*	指向 OLD 框架类的动态创建信息
	public	m_pOleViewClass	CRuntimeClass*	
	public	m_strDocStrings	CString	描述该文档类型的字符串

表5-10 单文档模板的成员变量

	访问限制	变量名称	类型	解释
	protected	m_pOnlyDoc	CDocment*	指向唯一的文档对象

表5-11 单文档模板的成员变量

	访问限制	变量名称	类型	解释
	public	m_hMenuShared	HMENU	该模板的 MDI 子窗口的菜单
	public	m_hAccelTable	HACCEL	该模板的 MDI 子窗口的快捷键
	protected	m_docList	CPtrList	该模板的文档列表
	protected	m_nUntitledCount	UINT	用来生成文件名的数字，如”untitled0”的 0。

5.3.2 WinMain 入口函数

5.3.2.1 WinMain 流程

现在讨论 MFC 应用程序如何启动。
WinMain 函数是 MFC 提供的应用程序入口。进入 WinMain 前，全局应用程序对象已经生成。WinMain 流程如图 5-3 所示。图中，灰色框是对被调用的虚拟函数的注释，程序员可以或必须覆盖它以实现 MFC 要求的或用户希望的功能；大括号所包含的图示是相应函数流程的细化，有应用程序对象 App 的初始化、Run 函数的实现、PumpMessage 的流程，等等。

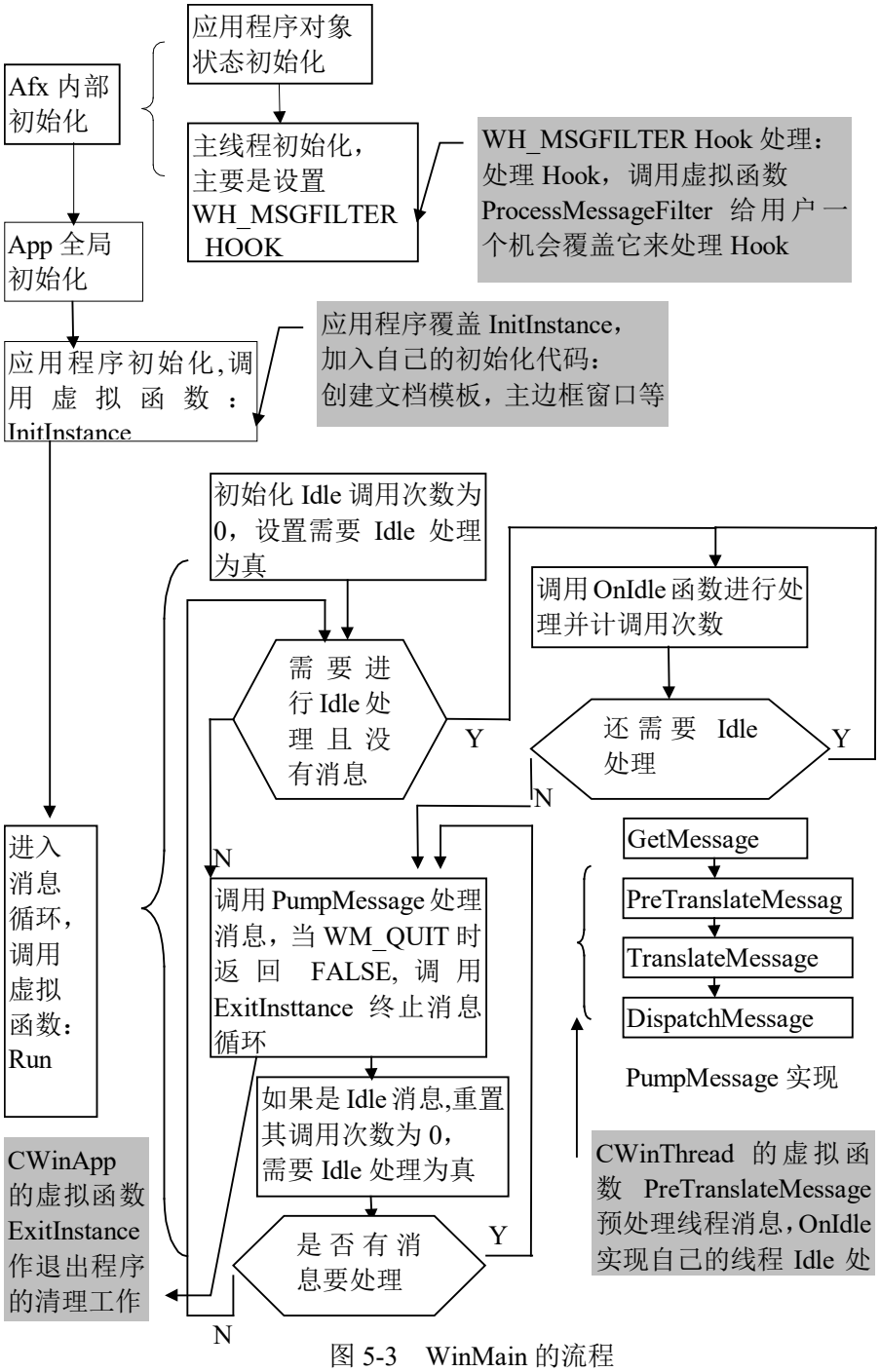


图 5-3　WinMain 的流程

从图中可以看出：

（1）一些虚拟函数被调用的时机

对应用程序类（线程类）的 InitIntance、ExitInstance、Run、ProcessMessageFilter、OnIdle、PreTranslateMessage 来说，InitInstance 在应用程序初始化时调用，ExitInstance 在程序退出时调用，Run 在程序初始化之后调用导致程序进入消息循环，ProcessMessageFilter、OnIdle、PreTranslateMessage 在消息循环时被调用，分别用来过滤消息、进行 Idle 处理、让窗口预处理消息。

（2）应用程序对象的角色

首先，应用程序对象的成员函数 InitInstance 被 WinMain 调用。对程序员来说，它就是程序的入口点（真正的入口点是 WinMain，但 MFC 向程序员隐藏了 WinMain 的存在）。由于

MFC 没有提供 InitInstance 的缺省实现，用户必须自己实现它。稍后将讨论该函数的实现。其次，通过应用程序对象的 Run 函数，程序进入消息循环。实际上，消息循环的实现是通过 CWinThread::Run 来实现的，图中所示的是 CWinThread::Run 的实现，因为 CWinApp 没有覆盖 Run 的实现，程序员的应用程序类一般也不用覆盖该函数。

（3）Run 所实现的消息循环

它调用 PumpMessage 来实现消息循环，如果没消息，则进行空闲(Idle)处理。如果是 WM_QUIT 消息，则调用 ExitInstance 后退出消息循环。

（4）CWinThread::PumpMessage

该函数在 MFC 函数文档里没有描述，但是 MFC 建议用户使用。它实现获取消息，转换(Translate)消息，发送消息的消息循环。在转换消息之前，调用虚拟函数 PreTranslateMessage 对消息进行预处理，该函数得到消息目的窗口对象之后，使用 CWnd 的 WalkPreTranslateTree 让目的窗口及其所有父窗口得到一个预处理当前消息的机会。关于消息预处理，见消息映射的有关章节。如果是 WM_QUIT 消息，PumpMessage 返回 FALSE；否则返回 TRUE。

5.3.2.2 MFC 空闲处理

MFC 实现了一个 Idle 处理机制，就是在没有消息可以处理时，进行 Idle 处理。Idle 处理的一个应用是更新用户接口对象的状态。更新用户接口状态的内容见消息映射的章节。

（1） 空闲处理由函数 OnIdle 完成，其原型为 BOOL OnIdle（int）。参数的含义是当前空闲处理周期已经完成了多少次 OnIdle 调用，每个空闲处理周期的第一次调用，该参数设为 0，每调用一次加 1；返回值表示当前空闲处理周期是否继续调用 OnIdle。

（2） MFC 的缺省实现里，CWinThread::OnIdle 完成了工具栏等的状态更新。如果覆盖 OnIdle，需要调用基类的实现。

（3） 在处理完一个消息或进入消息循环时，如果消息队列中没有消息要处理，则 MFC 开始一个新的空闲处理周期；

（4） 当 OnIdle 返回 FASLE，或者消息队列中有消息要处理时，当前的空闲处理周期结束。

从图 5-3 中 Run 的流程上可以清楚的看到 MFC 空闲处理的情况。

本节描述了应用程序从 InitInstance 开始初始化、从 Run 进入消息循环的过程，下面将就 SDI 应用程序的例子描述该过程中创建各个所需 MFC 对象的流程。

5.3.3 SDI 应用程序的对象创建

如前一节所述，程序从 InitInstance 开始。在 SDI 应用程序的 InitInstance 里，至少有以下语句：

```
//第一部分，创建文档模板对象并把它添加到应用程序的模板链表
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(CTDoc),
    RUNTIME_CLASS(CMainFrame),    // main SDI frame window
    RUNTIME_CLASS(CTView));
```

```
AddDocTemplate(pDocTemplate);

//第二部分，动态创建文档、视、边框窗口等 MFC 对象和对应的 Windows 对象
//Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);
// Dispatch commands specified on the command line
if (!ProcessShellCommand(cmdInfo))
return FALSE;

//第三部分，返回 TRUE，WinMain 下一步调用 Run 开始消息循环，
//否则，终止程序
return TRUE;
```

对于第二部分，又可以分解成许多步骤。
下面将解释每一步。

5.3.3.1 文档模板的创建

第一步是创建文档模板。
文档模板的作用是动态创建其他 MFC 对象，它保存了要动态创建类的动态创建信息和该文档类型的资源 ID。这些信息保存在文档模板的成员变量里：m_nIDResource(资源 ID)、m_pDocClass(文档类动态创建信息)、m_pFrameClass(边框窗口类动态创建信息)、m_pViewClass(视类动态创建信息)。
资源 ID 包括菜单、像标、快捷键、字符串资源的 ID，它们都使用同一个 ID 值，如 IDR_MAINFRAME。其中，字符串资源描述了文档类型，由七个被“\n”分隔的子字符串组成，各个子串可以通过 CDocTemplate 的成员函数 GetDocString(CString& rString, enum DocStringIndex index)来获取。DocStringIndex 是 CDocTemplate 类定义的枚举变量以区分七个子串，描述如下（英文是枚举变量名称）。
WindowTitle 应用程序窗口的标题。仅仅对 SDI 程序指定。
DocName 用来构造缺省文档名的字符串。当用 File 菜单的菜单项 new 创建新文档时，缺省文档名由该字符串加一个数字构成。如果空，使用“unitled”。
FileNewName 文档类型的名称，在打开 File New 对话框时显示。
FilterName 匹配过滤字符串，在 File Open 对话框用来过滤要显示的文件。如果不指定，File Open 对话框的文件类型(file style)不可访问。
FilterExt 该类型文档的扩展名。如果不指定，则不可访问对话框的文件类型(File Style)。
RegFileTypeId 文档类型在 Windows 注册库中的存储标识。
RegFileTypeName 文档类型在 Windows 注册库中的类型名称。

文档模板被应用程序对象创建和管理。应用程序类 CWinApp 有一个 CDocManager 类型的成员变量 m_pDocManager，通过该变量来管理应用程序的文档模板列表，把一些相关的操作委派给 CDocManager 对象处理。
CDocManager 使用 CPtrList 类型的 m_templateList 变量来存储文档模板，并提供了操作文档

模板列表的系列函数。

从语句 pDocTemplate = new CSingleDocTemplate(…)可以看出应用程序对象创建模板时传递一个资源 ID 和三个类的动态创建信息给它：
IDR_MAINFRAME，资源 ID
RUNTIME_CLASS(CTDoc)，文档类动态创建信息
RUNTIME_CLASS(CMainFrame),边框窗口类动态创建信息
RUNTIME_CLASS(CTView)，视类动态创建信息
文档模板对象接收这些信息并把它们保存到对应的成员变量里头。然后 AddDocTemplate 实际调用 m_pDocManager->AddDocTemplate，把创建的模板对象加入到文档模板管理器的模板列表中，也就是应用程序对象的文档模板列表中。

5.3.3.2 文件的创建或者打开

第二步是创建或者打开文件。
对于 SDI 程序，MFC 对象的动态创建过程是在创建或者打开文件中发生的。但是为什么没有看到文件操作相关的语句呢？

（1） CCommandLineInfo

首先，需要弄清楚类 CcommandLineInfo，它是用来处理命令行信息的类，CWinApp::PareCommandLine 调用 CCommandLineInfo 的成员函数 ParseParm 分析启动程序时的参数，把分析结果保存在 CCommandLineInfo 对象的成员变量里。CCommandLineInfo 的定义如下：

```
class CCommandLineInfo : public CObject
{
BOOL m_bShowSplash;
BOOL m_bRunEmbedded;
BOOL m_bRunAutomated;

enum { FileNew, FileOpen, FilePrint, FilePrintTo, FileDDE,
AppUnregister, FileNothing = -1 } m_nShellCommand;
// not valid for FileNew
CString m_strFileName;
// valid only for FilePrintTo
CString m_strPrinterName;
CString m_strDriverName;
CString m_strPortName;
};
```

由上述定义可以看出，分析结果分几类：是否 OLE 激活；应该执行什么动作（FileNew、FileOpen 等）；传递的参数（打开或打印的文件名，打印设备、端口等）。
当命令行空时，执行 FileNew 命令。原因在于 CCommandLineInfo 的缺省构造函数：

```
CCommandLineInfo::CCommandLineInfo()
{
m_bShowSplash = TRUE;
m_bRunEmbedded = FALSE;
```

```
m_bRunAutomated = FALSE;
m_nShellCommand = FileNew;//指定了 SHELL 命令操作
}
```

缺省构造把应该执行的动作指定为 FileNew。

（2） 处理命令行命令

其次，分析 CWinApp::ProcessShellCommand(CCommandLineInfo& rCmdInfo)的流程，它处理命令行的命令，流程如图 5-3 所示。

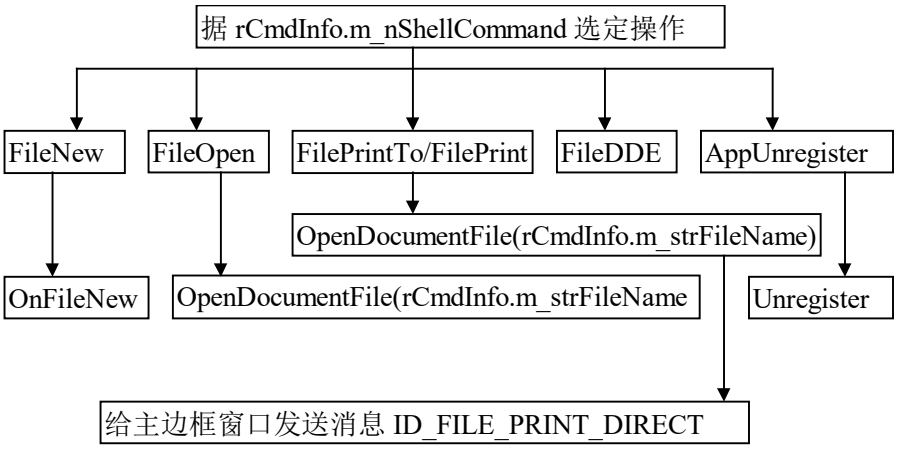


图 5-4 MFC 对命令行的处理

图 5-4 第三层表示根据命令类型进一步调用的函数，都是 CWinApp 或者派生类的成员函数。对于 FILEDDE 类型没有进一步的调用。

命令类型是 FILENEW 时，调用的函数就是标准命令 ID_FILE_NEW 对应的处理函数 OnFileNew；命令类型是 FILEOPEN 时调用的函数是 OpenDocumentFile，标准命令 ID_FILE_OPEN 的处理函数 OnFileOpen 的工作实际上就是由 OpenDocumentFile 完成的。函数 FileNew、OpenDocumentFile 导致了窗口、文档的创建。

（3） OnFileNew

接着，分析 CWinApp::OnFileNew 流程，如图 5-5 所示。

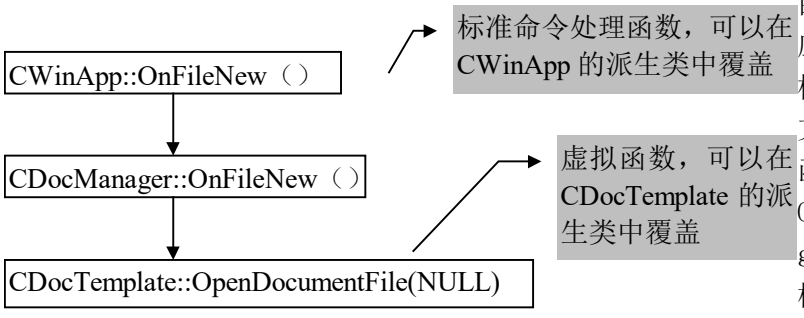


图 5-5 CWinApp 的 OnFileNew 函数

图 5-5 的说明：

应用程序对象得到文档模板管理器指针，调用文档模板管理器的成员函数 OnFileNew(m_pDocManager->OnFileNew())；模板管理器获取文档模板对象指针，调用文档模板对象的 OpenDocumentFile 函数（pTemplate->OpenDocumentFile(NULL)）。如果模板管理器发现有多个文档模板，就弹出一个对话框让用户选择文档模板。这里和后面的图解中类似于 CWinApp::、CDocManager::、CDocTemplate::等的函数类属限制并不表示直接源码中有这样的限制，而是通过指针或者指

针的动态约束可以认定调用了某个类的成员函数，其正确性仅仅限于本书图解的 MFC 的缺省实现。

如图 5-5 所示，程序员可以覆盖有关虚拟函数或命令处理函数：如果程序员在自己的应用程序类中覆盖了 OnFileNew，则可以实现完全不同的处理流程；一般情况下，不会从文档模板类派生新类，如果派生的话，可以覆盖 CDocTemplate 的虚拟函数。

（4） OnFileOpen

分析了 OnFileNew 后，现在分析 CWinApp::OnFileOpen()，其流程如图 5-6 所示。

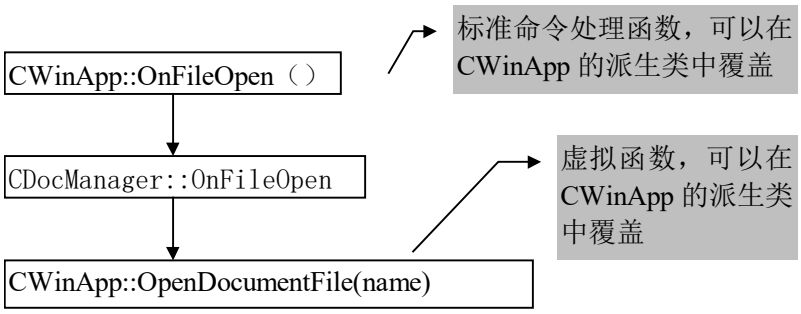


图 5-6 CWinApp 的 OnFileOpen 函数

CWinApp::OnFileOpen和OnFileNew类似，不过，第二步须得到一个要打开的文件的名称，第三步调用的是应用程序对象的OpenDocumentFile，而不是文档模板对象的该函数。

（5） 应用程序对象的 OpenDocumentFile

分析应用程序的打开文档函数：CWinApp::OpenDocumentFile(LPCSTR name)，其流程如图 5-7 所示。

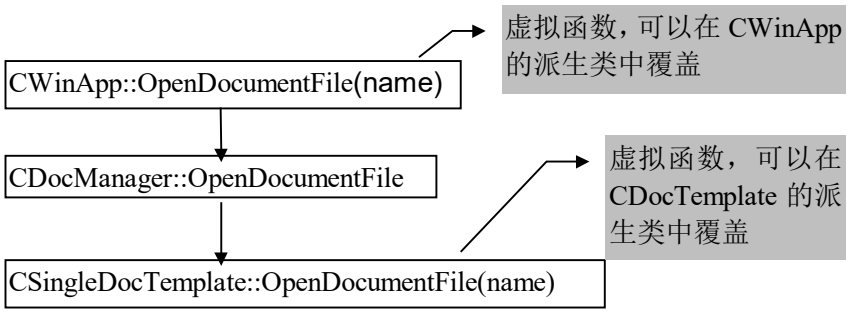


图 5-7 CWinApp 的 OpenDocumentFile 函数

应用程序对象把打开文件操作委托给文档模板管理器，后者又委托给文档模板对象来执行。如果是SDI程序，则委托给单文档对象；如果是MDI程序，则委托给多文档对象——这是由指针所指对象的实际类型决定的，因为该函数是一个虚拟函数。

（6） 文档模板的 OpenDocumentFile

不论是 FileNew 还是 FileOpen，最后的操作都归结到由文档模板来打开文件（文件名空则创建文件）。

CSingleDocTemplate::OpenDocumentFile(lpcstr name，BOOL visible)的流程见图 5-8。有一点需要指出的是：创建了一个文档对象，并不等于打开了一个文档(件)或者创建了一个新文档(件)。

图5-8显示的流程大致可以描述如下：

如果已经有文档打开，则保存当前的文档；否则，文档对象还没有创建，需要创建一个新的

文档对象。因为这时边框窗口还没有生成，所以还要创建边框窗口对象（MFC 对象）和边框窗口。MFC 边框窗口对象动态创建，HWND 边框窗口由 LoadFrame 创建。MFC 边框窗口被创建时，CFrameWnd 的缺省构造函数被调用，它把正创建的对象(this 所指)加入到模块-线程状态的边框窗口列表 m_frameList 之首。
边框窗口创建过程中由 CreateView 动态创建 MFC 视对象和 HWND 视窗口。
接着，如果没有指定要打开的文件名，则创建一个新的文件；否则，则打开文件，并使用序列化机制读入文件内容。
通过上述过程，动态地创建了 MFC 边框窗口对象、视对象、文档对象以及对应的 Windows 对象，并填写了有关对象的成员变量，建立起这些 MFC 对象的关系。

（7） 打开文件过程中所涉及的消息处理函数和虚拟函数

图 5-8 描述的整个过程中系列消息处理函数和虚拟函数被调用。例如：在 Windwos 边框窗口和视窗口被创建时会产生 WM_CREATE 等消息，导致 OnCreate 等消息处理函数的调用，CFrameWnd 和 CView 都覆盖了该函数，所以在边框窗口和视窗口的创建中，同样的消息调用了不同的处理函数 CFrameWnd::OnCreate 和 CView::OnCreate。
图 5-8 涉及的几个虚拟函数的流程分别由图 5-9、图 5-10 图解。图 5-9 表示 CDocument 的 OnNewDocument 的流程；图 5-10 表示 CDocument 的 OpenDocument 的流程。这两个函数分别在创建新文档或者打开一个文档时被调用。从流程可以看出，对于 OpenDocument 函数，MFC 的缺省实现主要用来设置修改标识、序列化读入打开文档的内容。图 5-10 显示了序列化的操作过程：
首先，使用文档对象打开或者创建的文件句柄创建一个用于读出数据的 CArchive 对象 loadarchive；然后使用它通过 Serialize 进行序列化操作，完毕，CArchive 对象被自动销毁，文件句柄被关闭。
从这些图中可以看到何时、何处调用了什么消息处理函数和虚拟函数，这些函数用来作了什么事情。必要的话，程序员可以在派生类覆盖它们。
在创建工作完成之后，进行初始化，使用文档对象的数据来更新视和显示窗口。

至此，本节描述了 MFC 的 SDI 程序从分析命令行到创建或打开文件的处理过程，文档对象已经动态创建。总结如下：
命令行分析→应用程序的 FileNew→文档模板的 OpenDocumentFile(NULL)→文档的 OnNewDocument
命令行分析→应用程序的 FileOPen→文档模板的 OpenDocumentFile(filename)→文档的 OpenDocument
边框窗口对象、视对象的动态创建和对应 Windows 对象的创建从 LoadFrame 开始，这些将在下一节论述。

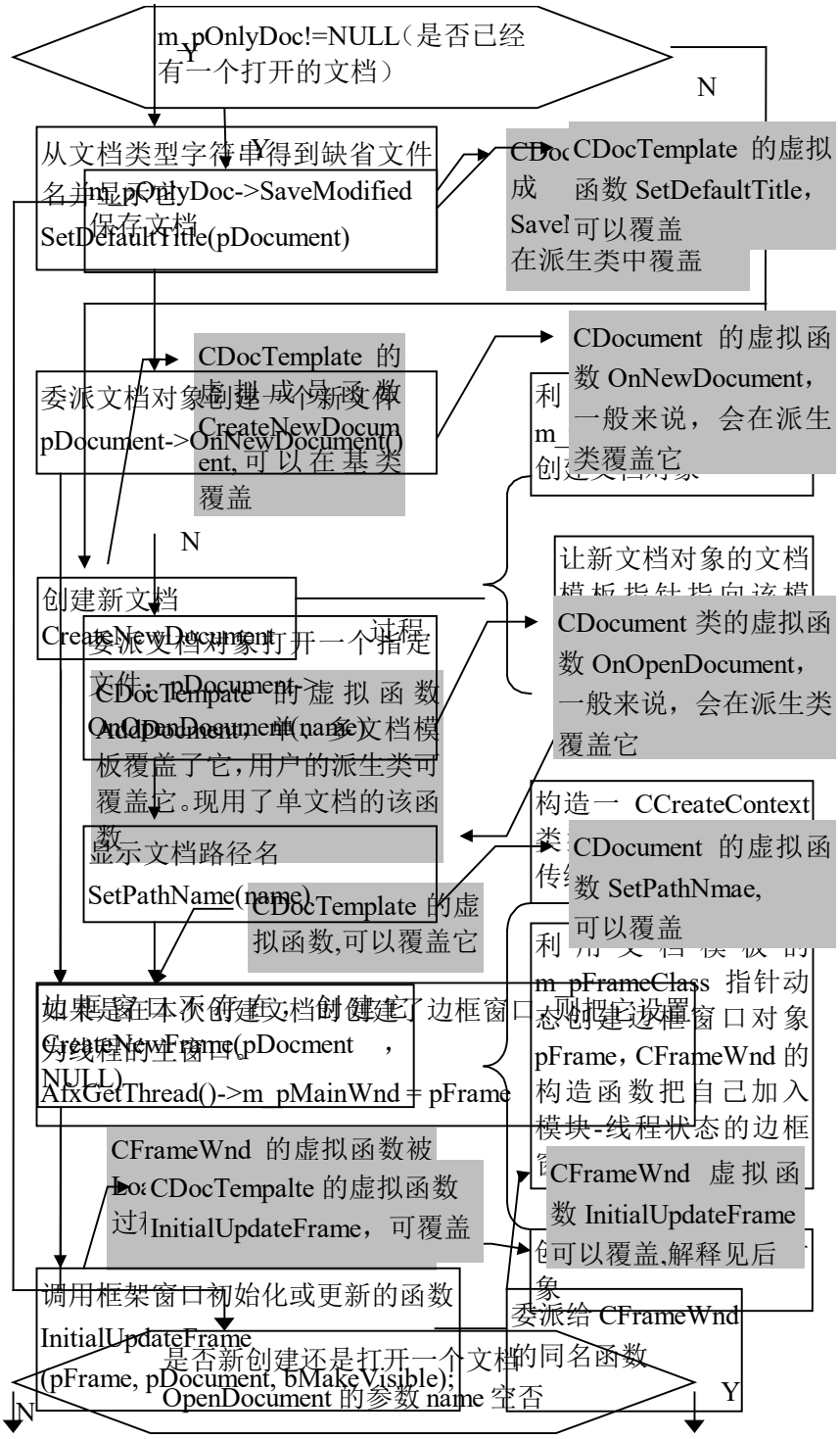


图 5-8　CSingleDocTemplate 的 OpenDocumentFile 函数

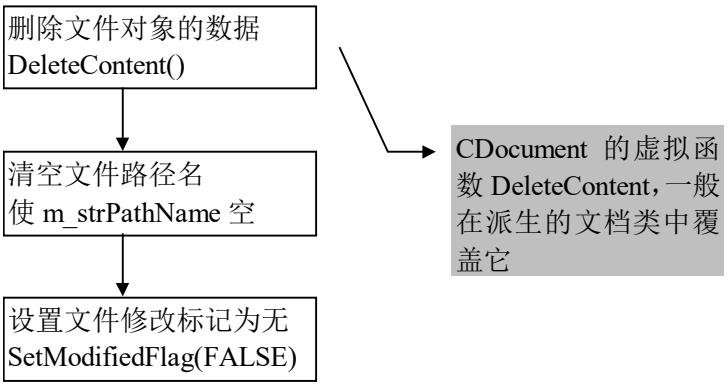


图 5-9　CDocument 的 OnNewDocment 函数

5.3.3.3　SDI 边框窗口的创建

第三步是创建 SDI 边框窗口。

图 5-8 已经分析了创建 SDI 边框窗口的时机和创建方法，下面，从 LoadFrame 开始分析整个窗口创建过程。

（1）　CFrameWnd::LoadFrame

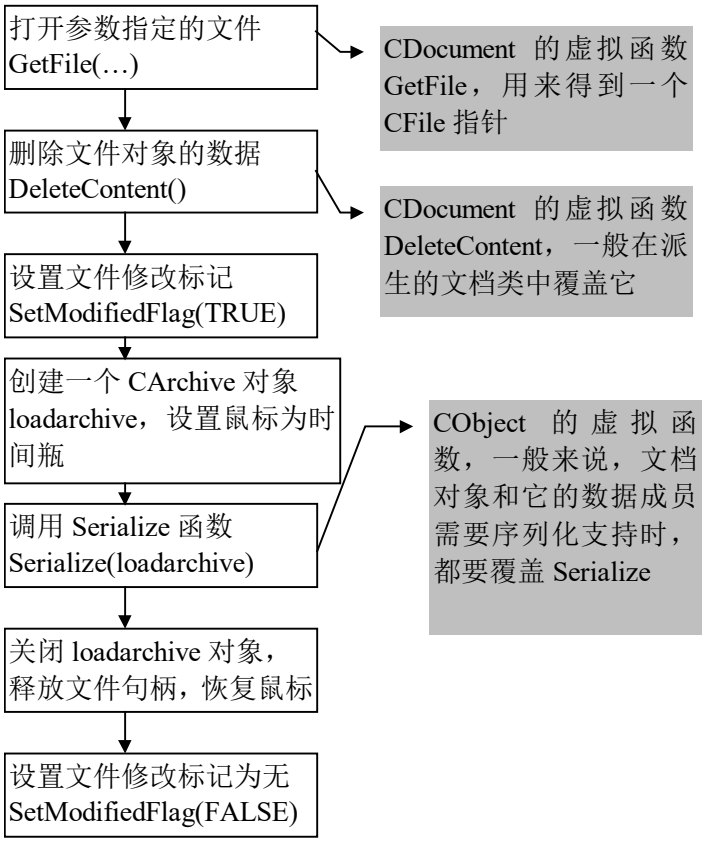


图 5-10　　CDocument 的 OpenDocument 函数

CFrameWnd::LoadFrame 的流程如图 5-11 所示，其原型如下：

```
BOOL CFrameWnd::LoadFrame(UINT nIDResource,
    DWORD dwDefaultStyle,
```

```
        CWnd* pParentWnd,
        CCreateContext* pContext)
```

第一个参数是和该框架相关的资源 ID，包括字符串、快捷键、菜单、像标等；
第二个参数指定框架窗口的“窗口类”和窗口风格；此处创建 SDI 窗口时和缺省值相同，为 WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE；
第三个参数指定框架窗口的父窗口，此处和缺省值相同，为 NULL；
第四个参数指定创建的上下文，如图 5-8 所示由 CreateNewFrame 生成了该变量并传递给 LoadFrame。其缺省值为 NULL。
创建上下文结构的定义：

```
struct CCreateContext
{
CRuntimeClass* m_pNewViewClass; //View 的动态创建信息
CDocument*m_pCurrentDoc;//指向一文档对象，将和新创建视关联

//用来创建 MDI 子窗口的信息(C MDIChildFrame::LoadFrame 使用)
CDocTemplate* m_pNewDocTemplate;

// for sharing view/frame state from the original view/frame
CView* m_pLastView;
CFrameWnd* m_pCurrentFrame;
};
```

这里，传递给 LoadFrame 的 CCreateContext 变量是：
(视的动态创建信息，新创建的文档对象，当前文档模板，NULL，NULL)。
其中，“新创建的文档对象”就是图 5-8 中创建的那个文档对象。从此图中还可以看到，LoadFrame 被 CreateNewFrame 调用，CreateNewFrame 是文档模板的成员函数，被文档模板的成员函数 OpenDocumentFile 所调用，所以，LoadFrame 间接地被文档模板调用，“当前文档模板”就是调用它的模板对象。顺便指出，对 SDI 程序来说是这样的，对 MDI 程序有所不同。“视的动态创建信息”也是文档模板传递过来的。

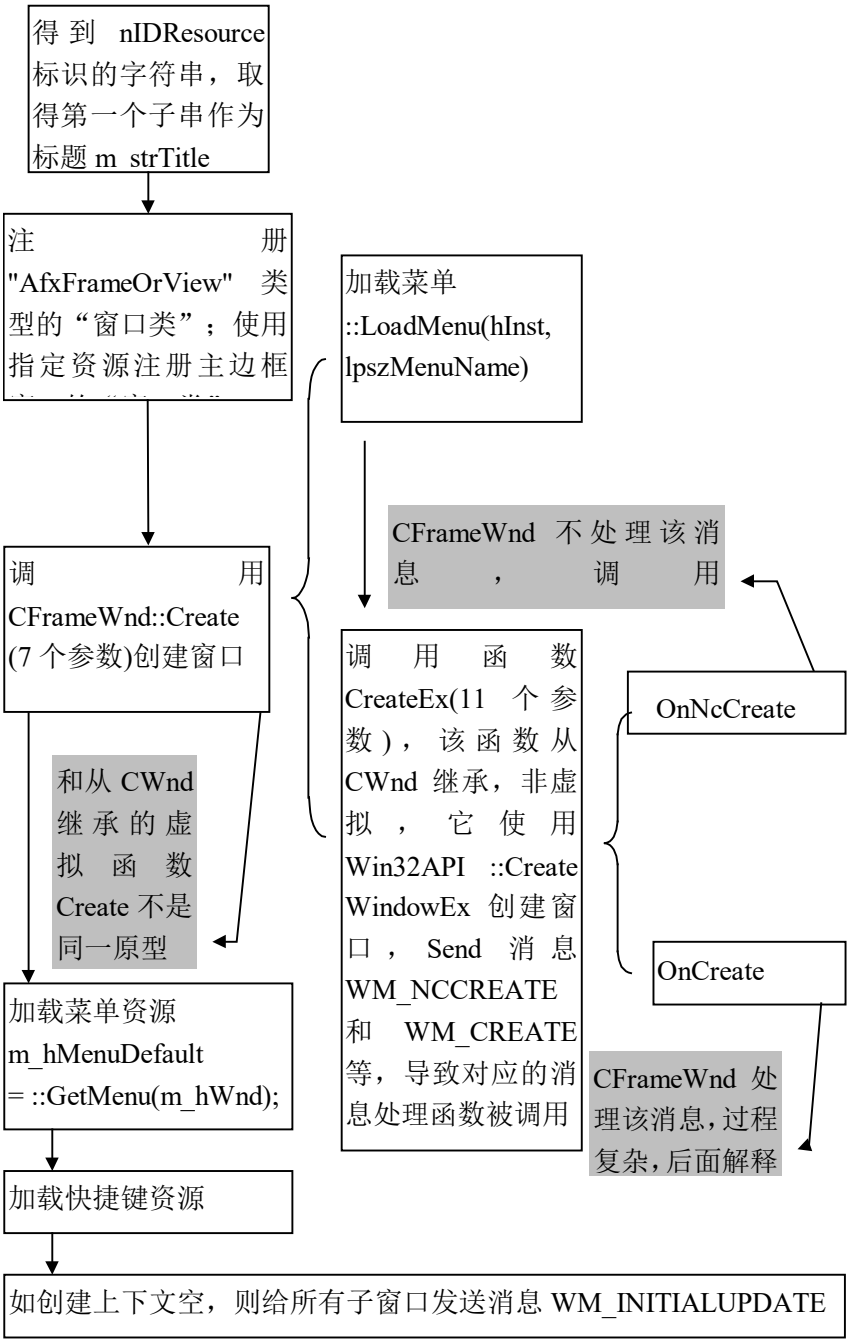


图 5-11　CFrameWnd 的 LodFrame 函数

对图 5-11 的说明：

在创建边框窗口之前，先注册“窗口类”。LoadFrame 注册了两个“窗口类”，一个为边框窗口，一个为视窗口。关于“窗口类”注册，见 2.2.1 节。

注册窗口类之后，创建边框窗口，并加载资源。创建边框窗口使用了 CFrameWnd 的 Create 虚拟函数，最终调用::CreateEx 创建窗口。::CreateEx 有 11 个参数，其最后一个参数就是文档模板传递给 LoadFrame 的 CCreateContext 类型的指针，该指针将被传递给窗口过程，进一步由 Windows 传递给 OnCreate 函数。顺便指出，创建完毕的边框窗口的窗口过程是统一的 MFC 窗口过程。

创建边框窗口时，发送消息 WM_NCCREATE 和 WM_CREATE，导致对应的消息处理函数 OnNcCreate 和 OnCreate 被调用。CWnd 提供了 OnNcCreate 处理非客户区创建消息，CFrameWnd 没有处理该消息，但是提供了 OnCreate 处理消息 WM_CREATE。OnCreate 将创建视对象和视窗口。

（2） CFrameWnd::OnCreate

按创建工作的进度，现在要讨论边框窗口创建消息（WM_CREATE）的处理了，处理函数是 CFrameWnd 的 OnCreate，其原型如下：

```
int CFrameWnd::OnCreate(LPCREATESTRUCT lpcs)
```

其中，参数指向一个 CreateStruct 结构（关于 CreateStruct 的描述见 4.4.1 节），它包含了窗口创建参数的副本，也就是说 CreaeEx 窗口创建函数的 11 个参数被对应地复制到该结构的 11 个域，例如它的第一个成员就可以转换成 CCreateContext 类型的指针。

函数 OnCreate 处理 WM_CREATE 消息，它从 lpcs 指向的结构中分离出 lpCreateParams 并把它转换成为 CCreateContext 类型的指针 pContext，然后，调用 OnCreateHelp(lpcs，pContext)，把创建工作委派给它完成。

CFrameWnd::OnCreateHelp 的原型如下，流程见图 5-11。

```
int CFrameWnd::OnCreateHelp(LPCREATESTRUCT lpcs,
    CCreateContext* pContext)
```

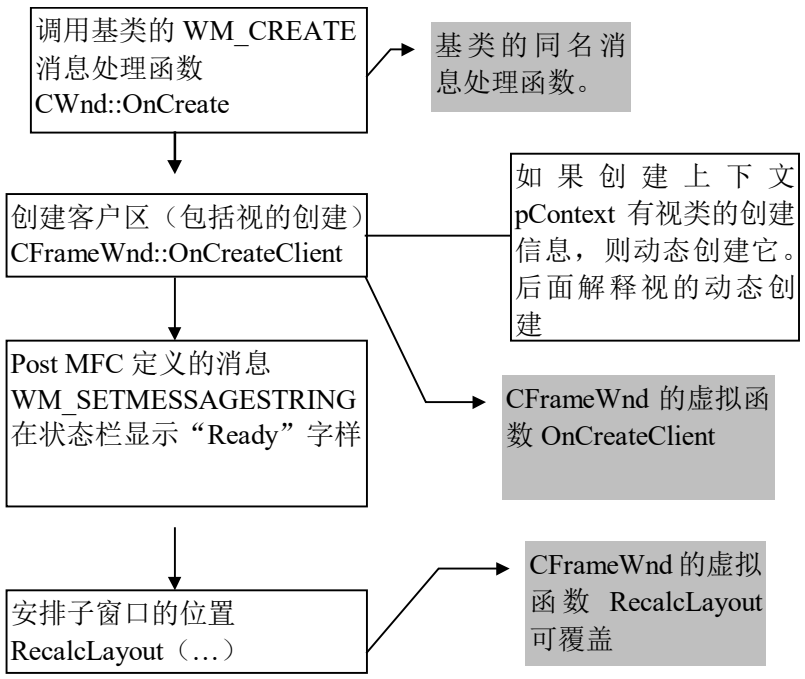


图 5-12 CFrameWnd 的 OnCreateHelp 函

说明：由于 CFrameWnd 覆盖了消息处理函数 OnCreate 来处理 WM_CREATE 消息，所以 CWnd 就失去了处理该消息的机会，除非 CFrameWnd::OnCreate 主动调用基类的该消息处理函数。图 5-11 展示了对 CWnd::OnCreate 的调用。

在边框窗口被创建之后，调用虚拟函数 OnCreateClient(lpcs，pContext)，它的缺省实现将创建视对象和视窗口。

最后，在状态栏显示“Ready”字样，调用 RecalcLayout 安排工具栏等的位置。关于 WM_SETMESSAGESTRING 消息和 RecalcLayout 函数，见工具栏有关 13.2.3 节。

到此，SDI 的边框窗口已经被创建。下一节将描述视的创建。

5.3.3.4 视的创建

第四步，创建视。

如前一节所述，若 CFrameWnd::OnCreateClient(lpcs，pContext)判断 pContext 包含了视的动态创建信息，则调用函数 CreateView 创建视对象和视窗口。CreateView 的原型如下，其流程如图 5-13 所示。

```
CWnd * CFrameWnd::CreateView(CCreateContext* pContext, UINT nID)
```

其中：

第一个参数是创建上下文；

第二个参数是创建视(子窗口)的 ID，缺省是 AFX_IDW_PANE_FIRST，这里等同缺省值。

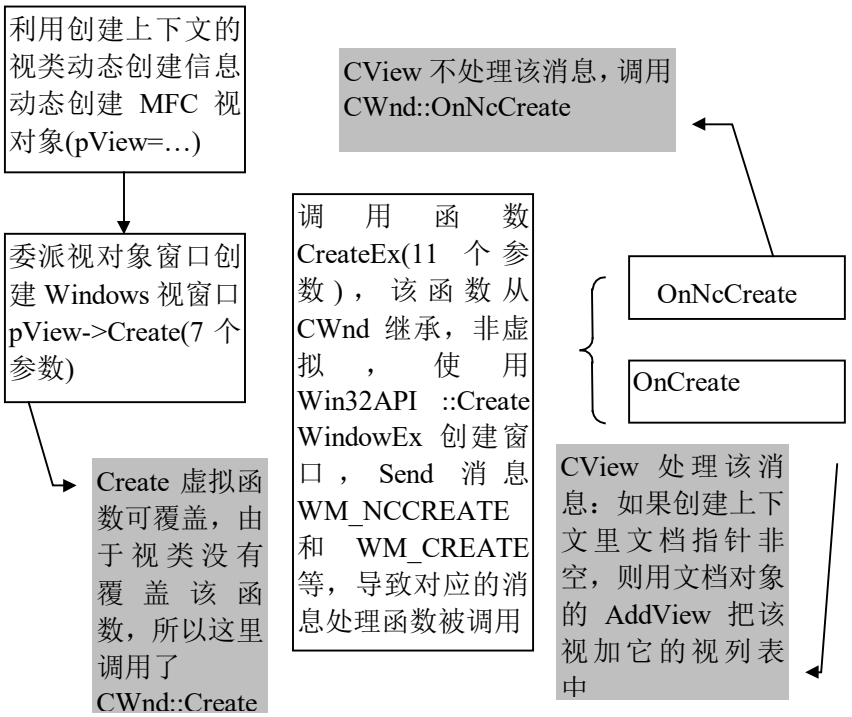


图 5-13　CFrameWnd 的 CreateView 函数

说明：CreateView 调用了 CWnd 的 Create 函数创建 HWND 视窗口，视的子窗口 ID 是 AFX_IDW_PANE_FIRST，父窗口是创建它的边框窗口。创建视窗口时的 WM_CREATE、WM_NCCREATE 消息分别被 CView、CWnd 的相关消息处理函数处理。处理情况如图 5-13 所述，这里不作进一步的讨论。

到此，文档对象、边框窗口对象、视窗口对象已经创建，文件已经打开或者创建，边框窗口、视窗口已经创建。现在，是显示和定位窗口、显示文档数据的时候了，这些通过调用 CFrameWnd 的虚拟函数 InitialUpdateFrame 完成，如图 5-8 所示。

5.3.3.5 窗口初始化

这是第五步，初始化边框窗口、视窗口等。

InitialUpdateFrame 的原型如下：

```
void CFrameWnd::InitialUpdateFrame(CDocument* pDoc, BOOL bMakeVisible)
```

其中：

第一个参数是当前的文档对象；

第二个参数表示边框窗口是否应该可见并激活。
该函数是在文档模板的 OpenDocumentFile 中调用的，传递过来的第一个参数是刚才创建的文档，第二个参数是 TRUE，见图 5-8。
InitialUpdateFrame 的处理过程参见图 5-14，解释如下：
首先，如果当前边框窗口没有活动视，则获取 ID 为 AFX_IDW_PANE_FIRST 的视 pView。如果该视不存在，则 pView=NULL；否则(pView!=NULL)，调用成员函数 SetActiveView(pView，FALSE)把它设置为活动视，参数 2 为 FALSE 表示并不通知它成为活动视（见图 5-14）。
然后，如果 InitialUpdateFrame 的参数 bMakeVisible 为 TRUE，则给所有边框窗口的视发送 WM_INITIALUPDATE 消息，通知它们在显示之前使用文档数据来初始化视。这导致视类的虚拟函数 OnInitUpdate 被调用，该函数又调用 OnUpdate 来完成初始化。其他子窗口（如状态栏、工具栏）也将收到 WM_INITIALUPDATE 消息，导致它们更新状态。
其三，调用 pView->OnActivateFrame（WA_INACTIVE，this）给活动视（若存在的话）一个机会来保存当前焦点。这里，解释这个函数：

```
void CView::OnActivateFrame( UINT nState,CFrameWnd* pFrameWnd );
```

其中，参数 1 取值为 WA_INACTIVE/WA_ACTIVE/WA_CLICKACTIVE，具体见消息 WM_ACTIVE 的解释；参数 2 指向被激活的框架窗口。
视对象通过该虚拟函数在它所属的边框窗口被激活或者失去激活时作一些特别的处理，例如，CFormView 用它来保存或者恢复输入焦点控制。
其四，在 OnActivateFrame 之后，调用成员函数 ActivateFrame 激活框架窗口。这个过程将产生一个消息 WM_ACTIVE（处理该消息的过程在下一节作解释），它导致 OnActiveTopLevel 和 OnActive 被调用。接着，如果活动视非空，则调用成员函数 OnActivateView 激活它。
至此，参数 bMakeVisible 为 TRUE 时显示窗口的处理完毕。
最后，如果参数 pDoc 非空，则更新边框窗口计数，更新边框窗口的标题。更新边框窗口计数是为了在一个文档对应多个视的情况下，给显示同一文档的不同文档边框窗口编号，编号从 1 开始，保存在边框窗口的成员变量 m_nWindow 里。例如有两个边框对应一个文档 tt1，则它们的标题分别为“tt1:1”、“tt1:2”。如果只有一个文档只对应一个边框窗口，则成员变量 m_nWindow 等于-1，标题不含编号，如“tt1”。
当然，对于 SDI 应用程序，不存在一个文档对应多个视的情况。上述情况是针对 MDI 应用程序而言的。SDI 应用程序执行该过程时，相当于 MDI 程序的一个特例。
图 5-14 涉及的一些函数由图 5-15、5-15 图解。

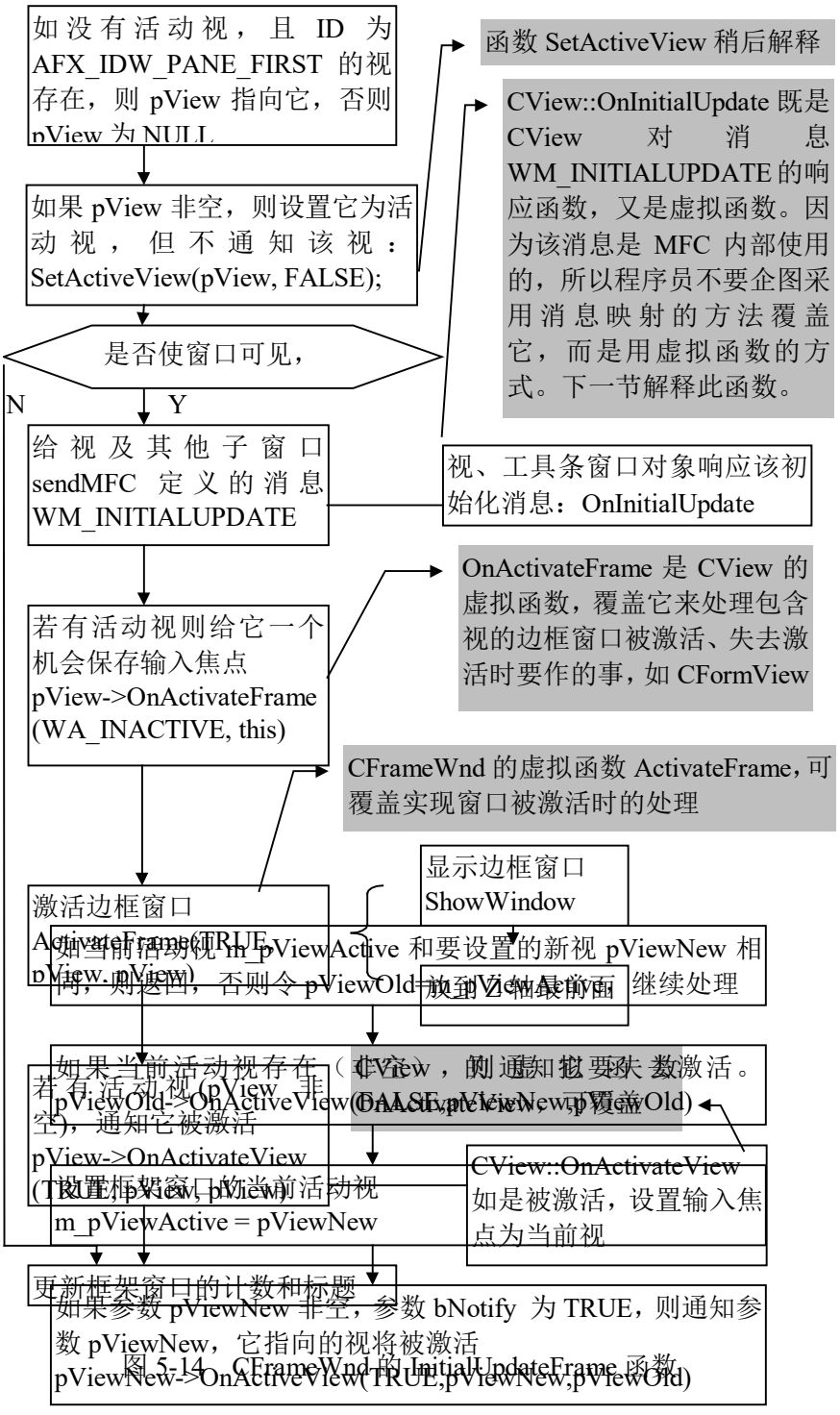


图 5-15　CFrameWnd 的 SetActiveView 函

图 5-14 中的函数 SetActiveView 的图解如图 5-15 所示，其原型如下，：

void CFrameWnd::SetActiveView(CView * pViewNew, BOOL bNotify = TRUE)

其中：

参数 1 指向被设置的视对象，若非视类型的对象，则为 NULL；

参数 2 表示是否通知被设置的视。

图 5-15 中的变量 m_pViewActive 是 CFrameWnd 的成员变量，用来保存边框窗口的活动视。

图 5-15 中的流程可以概括为：Deactivate 当前视（m_pViewActive 非空时）；设置当前活动视；若参数 bNotify 为 TRUE，通知 pViewNew 被激活。

图 5-14 中的函数 ActivateFrame 图

图 5-14　CFrameWnd 的 InitialUpdateFrame 函数

解如图 5-16 所示，其原型如下，：

void CFrameWnd::ActivateFrame(UINT nCmdShow)

参数 nCmdShow 用来传递给 CWnd::ShowWindow，指定显示窗口的方式。参数缺省为 1，图 5-14 调用时设置为-1。

该函数用来激活(Activate)和恢复(Restore)边框窗口，使得它对用户可见可用。在初始化、OLE 事件、DDE 事件等需要显示边框窗口的地方调用。图 5-16 表示的 MFC 缺省实现是激活边

框窗口并把它放到顶层。
程序员可以覆盖该虚拟函数 ActivateFrame 来控制边框窗口怎样被激活。
图 5-16 中的函数 BringToTop 是 CFrameWnd 内部使用的成员函数(protected)。它调用::BringWindowToTop 把窗口放到 Z 轴上的顶层。
至此，边框窗口初始化的过程已经描述清楚，视的初始化见下一节。

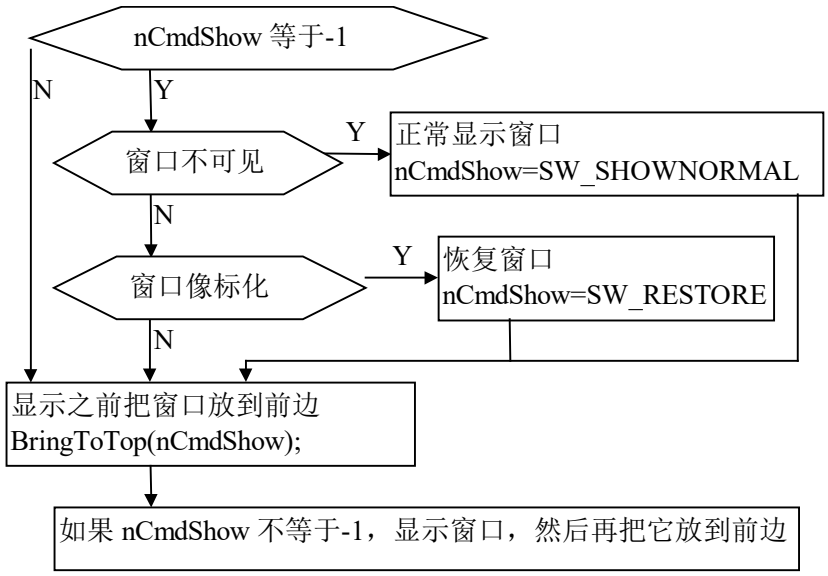


图 5-16　CFrameWnd 的 ActivateFrame 函数

5.3.3.6　视的初始化

第六步，在边框窗口初始化之后，初始化视。
如图 5-14 所示，视、工具条窗口处理消息 WM_INITAILUPDATE(MFC 内部消息)，完成初始化。这里只讨论视的消息处理函数，其原型如下：

```
void CView::OnInitialUpdate()
```

图 5-14 对该函数的注释说明了该函数的特殊之处。其缺省实现是调用 OnUpdate(NULL, 0, NULL)更新视。可以覆盖 OnInitialUpdate 实现自己的初始化。

OnUpdate 是一个虚拟函数，其原型如下：

```
void CView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
```

其中：
参数 1 指向修改文档数据的视；若更新所有的视，设为 NULL；
参数 2 是一个包含了修改信息的 long 型变量；
参数 3 指向一个包含修改信息的对象（从 CObject 派生的对象）。
参数 2、参数 3 是在文档更新对应视的时候传递过来的。
该函数用来更新显示视窗口，反映文档的变化，在 MFC 中，它为函数 CView::OnInitialUpdate 和 CDocument::UpdateAllViews 所调用。其缺省实现是使整个客户区无效。在下一次收到 WM_PAINT 消息时，重绘无效区。
工具条的初始化见讨论第 13 章。

5.3.3.7　激活边框窗口(处理 WM_ACTIVE)

第七步，在窗口初始化完成之后，激活并显示出来。
下面讨论边框窗口激活时的处理（对 WM_ACTIVE 的处理）。

（1）　WM_ACTIVE 的消息参数

wParam 的低阶 word 指示窗口是被激活还是失去激活：WA_ACTIVE，被鼠标点击以外的方法激活；WA_CLICKACTIVE，由鼠标点击激活；WA_INACTIVE，失去激活；
wParam 的高阶 word 指示窗口是否被最小化；非零表示最小化；
lPararm 表示将激活的窗口句柄(WA_INACTIVE)，或者将失去激活的窗口句柄(WA_CLICKACTIVE、WA_ACTIVE)。

在标准 Windows 消息处理的章节中，曾指出处理 WM_ACTIVE 消息时，先要调用一个函数_AfxHandleActivate，此函数的原型如下：

```
static void AFXAPI _AfxHandleActivate(CWnd* pWnd,
    WPARAM nState,CWnd* pWndOther)
```

其中：
参数 1 是接收消息的窗口；
参数 2 是窗口状态，为 WM_ACTIVE 的消息参数 wParam;
参数 3 是 WM_ACTIVE 的消息参数 lParam 表示的窗口。
_AfxHandleActivate 是 MFC 内部使用的函数，声明和实现均在 WinCore.CPP 文件中。实现如下：
如果 pWnd 指向的窗口不是子窗口，而且 pWnd 和 pWndOther 窗口的顶级父窗口(TopLevelParent)不是同一窗口，则发送 MFC 定义的消息 WM_ACTIVATETOPLEVEL 给 pWnd 的顶级窗口，消息参数 wParam 是 nState，消息参数 lParam 指向一个长度为二的数组，数组里存放 pWnd 和 pWndOther 所指窗口的句柄。否则，_AfxHandleActivate 不作什么。
从这里可以看出：只有顶层的主边框窗口能处理 WM_ACTIVE 消息，事实上，Windows 系统只会给顶层的非子窗口发送 WM_ACTIVE 消息。

（2） WM_ACTIVATETOPLEVEL 消息的处理

CWnd 及派生类 CFrameWnd 实现了对 WM_ACTIVATETOPLEVEL 消息的处理，分别解释如下：
消息处理函数 CWnd::OnActivateTopLevel 如果失去激活，则取消工具栏的提示（TOOLTIP）。
消息处理函数 CFrameWnd::OnActivateTopLevel 调用 CWnd 的 OnActivateTopLevel；如果接收 WM_ACTIVE 消息的窗口是线程主窗口，则使得其活动的视窗口变成非活动的（OnActive(FALSE, pActiveView,pActiveView)。
从这里可以知道，在顶层窗口接收到 WM_ACTIVE 消息后，MFC 会进行一些固定的处理，然后才调用 WM_ACTIVE 消息处理函数。

（3） WM_ACTIVE 消息的处理

在_AfxHandleActivate 和 WM_ACTIVATETOPLEVEL 消息处理完之后，才是对 WM_ACTIVE 的处理。CWnd 和 CFrameWnd 都实现了消息处理。
CWnd 的消息处理函数：

```
void CWnd::OnActive(UINT nState, CWnd* pWndOther, BOOL bMinimized)
```

其中：
参数 1 取值为 WA_INACTIVE/WA_ACTIVE/WA_CLICKACTIVE；
参数 2 指向激活或者失去激活的窗口，具体同 WM_ACTIVE 消息；
参数 3 表示是否最小化。
此函数的实现是调用 Default()，作缺省处理。

CFrameWnd 的消息处理函数：

```
void CFrameWnd::OnActive(UINT nState,CWnd* pWndOther, BOOL bMinimized)
```

首先调用 CWnd::OnActivate。
如果活动视非空，消息是 WA_ACTIVE/WA_CLICKACTIVE，并且不是最小化，则重新激活当前视，调用了以下函数：
pActiveView->OnActiveView(TRUE，pActiveView，pActiveView)；
并且，如果活动视非空，通知它边框窗口状态的变化（激活/失去激活），调用以下函数：
pActiveView->OnActivateFrame(nState, this)。

5.3.3.8 SDI 流程的回顾

从 InitialInstance 开始，首先应用程序对象创建文档模板，文档模板创建文档对象、打开或创建文件；然后，文档模板创建边框窗口对象和边框窗口；接着边框窗口对象创建视对象和视窗口。这些创建是以应用程序的文档模板为中心进行的。在创建这些 MFC 对象的同时，建立了它们之间的关系。创建这些之后，进行初始化，激活主边框窗口，把边框窗口、视窗口显示出来。
这样，一个 SDI 应用程序就完成了启动过程，等待着用户的交互或者输入。
5.3.4 节将在 SDI 程序启动流程的基础之上，介绍 MDI 应用程序的启动流程。两者的相同之处可以这样类比：创建 SDI 边框窗口、视、文档的过程和创建 MDI 文档边框窗口、视、文档的过程类似。不同之处主要表现在：主边框窗口的创建不一样；MDI 有文档边框窗口的创建，SDI 没有；SDI 只能一个文档、一个视；MDI 可能多文档、多个视。

5.3.4 MDI 程序的对象创建

MDI 应用程序对象的 InitialInstance 函数一般含有以下代码：

```
//第一部分：创建和添加模板
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
        IDR_TTTYPE,
        RUNTIME_CLASS(CTtDoc),
        RUNTIME_CLASS(CChildFrame),//custom MDI child frame
        RUNTIME_CLASS(CTtView));
AddDocTemplate(pDocTemplate);

//第二部分：创建 MFC 框架窗口对象和 Windows 主边框窗口
// 创建主 MDI 边框窗口
CMainFrame* pMainFrame = new CMainFrame;
if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
        return FALSE;
m_pMainWnd = pMainFrame;

//第三部分：处理命令行，命令行空则执行 OnFileNew 创建新文档
//分析命令行
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);
```

```
// 处理命令行命令
if (!ProcessShellCommand(cmdInfo))
    return FALSE;

第四部分：显示和更新主框架窗口
// 主窗口已被初始化，现在显示和更新主窗口
pMainFrame->ShowWindow(m_nCmdShow);
pMainFrame->UpdateWindow();
```

SDI 应用程序对象的 InitialInstance 和 SDI 应用程序对象的 InitialInstance 比较，有以下的相同和不同之处。相同之处在于：
创建和添加模板；处理命令行。
不同之处在于：

- 创建的模板类型不同。SDI 使用单文档模板，边框窗口类从 CFrameWnd 派生；MDI 使用多文档模板，边框窗口类从 CMDIChildWnd 派生.
- 主窗口类型不同。SDI 的是从 CFrameWnd 派生的类；MDI 的是从 CMDIFrameWnd 派生的类。
- 主框架窗口的创建方式不同。SDI 在创建或者打开文档时动态创建主窗口对象，然后加载主窗口(LoadFrame)并初始化；MDI 使用第二部分的语句来创建动态主窗口对象和加载主窗口，第四部分语句显示、更新主窗口。
- 命令行处理的用途不一样。SDI 一定要有命令行处理部分的代码，因为它导致了主窗口的创建；MDI 可以去掉这部分代码，因为它的主窗口的创建、显示等由第二、四部分的语句来处理。

5.3.4.1 有别于 SDI 的主窗口加载过程

和 SDI 应用程序一样，MDI 应用程序使用 LoadFrame 加载主边框窗口，但因为 LoadFrame 的虚拟属性，所以 MDI 调用了 CMDIFrameWnd 的 LoadFrame 函数，而不是 CFrameWnd 的 LoadFrame。
LoadFrame 的参数 1 指定了资源 ID，其余几个参数取缺省值。和 SDI 相比，第四个创建上下文参数为 NULL，因为 MDI 主窗口不需要文档、视等的动态创建信息。
图 5-17 图解了 CMdiFrameWnd::LoadFrame 的流程：

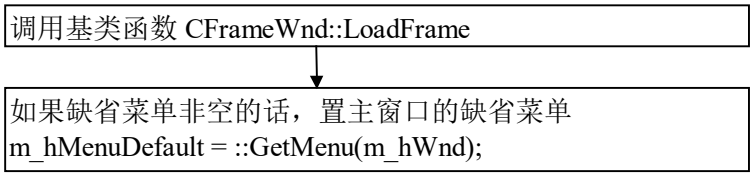


图 5-17 CMDIFrameWnd 的 LoadFrame 函数

首先，用同样的参数调用基类 CFrameWnd 的 LoadFrame，其流程如图 5-11 所示，但由于参数 4 表示的创建上下文为空，所以，CFrameWnd::LoadFrame 在加载了菜单和快捷键之后，给所有子窗口发送 WM_INITUPDATE 消息。
另外，WM_CREATE 消息怎样处理呢？由于 CMDIFrameWnd 没有覆盖 OnCreate，所以还是由基类 CFrameWnd::OnCreate 处理。但是它调用虚拟函数 OnCreateClient（见图 5-12）时，

由于 CMDIFrameWnd 覆盖了该函数，所以动态约束的结果是 CMDIFrameWnd::OnCreateClient 被调用，它和基类的 OnCreateClient 不同，后者 CreateView 创建 MFC 视对象和视窗口，前者调用虚拟函数 CreateClient 创建 MDI 客户窗口。MDI 客户窗口负责创建和管理 MDI 子窗口。

CreateClient 是 CMDIFrameWnd 的虚拟函数，其原型如下：

```
BOOL CMDIFrameWnd::CreateClient(
    LPCREATESTRUCT lpCreateStruct, CMenu* pWindowMenu);
```

该函数主要用来创建 MDI 客户区窗口。它使用 Windows 系统预定义的“mdiclient”窗口类来创建客户区窗口，保存该窗口句柄在 CMDIFrameWnd 的成员变量 m_hWndMDIClient 中。调用::CreateWindowEx 创建客户窗口时传递给它的窗口创建数据参数（第 11 个参数）是一个 CLIENTCREATESTRUCT 结构类型的参数，该结构指定了一个菜单和一个子窗口 ID：

```
typedef struct tagCLIENTCREATESTRUCT{
HMENU       hWindowMenu;
UINT        idFirstChild;
}CLIENTCREATESTRUCT;
```

hWindowMenu 表示主边框窗口菜单栏上的“Windows 弹出菜单项”的句柄。MDICLIENT 类客户窗口在创建 MDI 子窗口时，把每一个子窗口的标题加在这个弹出菜单的底部。idFirstChild 是第一个被创建的 MDI 子窗口的 ID 号，第二个 MDI 子窗口 ID 号为 idFirstChild+1，依此类推。

这里，hWindowMenu 的指定不是必须的，程序员可以在 MDI 子窗口激活时进行菜单的处理；idFirstChild 的值是 AFX_IDM_FIRST_MDICHILD。

综合地讲，CMDIFrameWnd::LoadFrame 完成创建 MDI 主边框窗口和 MDI 客户区窗口的工作。

创建了 MDI 边框窗口和客户区窗口之后，接着是处理 WM_INITUPDATE 消息，进行初始化。但是按 SDI 应用程序的讨论顺序，下一节先讨论 MDI 子窗口的创建。

5.3.4.2 MDI 子窗口、视、文档的创建

和 SDI 应用程序类似，MDI 应用程序通过文档模板来动态创建 MDI 子窗口、视、文档对象。不同之处在于：这里使用了多文档模板，调用的是 CMDIChildWnd（或派生类）的消息处理函数和虚拟函数，如果它覆盖了 CFrameWnd 的有关函数的话。

还是以处理标准命令消息 ID_FILE_NEW 的 OnFileNew 为例。

表示 OnFileNew 的图 5-5、表示 OnFileOpen 的图 5-6 在多文档应用程序中仍然适用，但表示 OpenDocumentFile 的图 5-8 有所不同，其第三步中地单文档模板应当换成多文档模板，关于这一点，参阅图 5-8 的说明。

（1）多文档模板的 OpenDocumentFile

MDI 的 OpenDocumentFile 的原型如下：

```
CDocument* CMultiDocTemplate::OpenDocumentFile(
    LPCTSTR lpszPathName, BOOL bMakeVisible);
```

它的原型和单文档模板的该函数原型一样，但处理流程比图 5-8 要简单些：

第一，不用检查是否已经打开了文档；

第二，不用判断是否需要创建框架窗口或者文档对象，因为不论新建还是打开文档都需

要创建新的文档框架窗口(MDI 子窗口)和文档对象。
除了这两点，其他处理步骤基本相同，调用同样名字的函数来创建文档对象和 MDI 子窗口。虽然是名字相同的函数，但是参数的值可能有异，又由于 C++的虚拟机制和 MFC 消息映射机制，这些函数可能来自不同层次类的成员函数，因而导致有不同的处理过程和结果，即 SDI 创建了 CFrameWnd 类型的对象和边框窗口；MDI 则创建了 CMDIChildWnd 类型的对象和边框窗口。不同之处解释如下：

（2）CMDIChildWnd 的虚拟函数 LoadFrame

CMDIChildWnd::LoadFrame 代替了图 5-8 中的 CFrameWnd::LoadFrame，两者流程大致相同，可以参见图 5-11。但是它们用来创建窗口的函数不同。前者调用了函数 CMDIChildWnd::Create(参数 1…参数 6)；后者调用了 CFrameWnd::Create(参数 1…参数 7)。这两个窗口创建函数，虽然都是虚拟函数，但是有很多不同之处：

- 前者是 CMDIChildWnd 定义的虚拟函数，后者是 CWnd 定义的虚拟函数；
- 前者在参数中指定了父窗口，即主创建窗口，后者的父窗口参数为 NULL；
- 前者指定了 WS_CHILD 风格，创建的是子窗口，后者创建一个顶层窗口；
- 前者给客户窗口 m_hWndMDIClient(CMDIFrameWnd 的成员变量)发送 WM_MDICREATE 消息让客户窗口来创建 MDI 子窗口（主边框窗口的子窗口是客户窗口，客户窗口的子窗口是 MDI 子窗口），后者调用::CreateEx 函数来创建边框窗口；
- 前者的窗口创建数据是指向 MDICREATESTRUCT 结构的指针，该结构的最后一个域存放一个指向 CCreateContext 结构的指针，后者是指向 CCreateContext 结构的指针。

MDICREATESTRUCT 结构的定义如下：

```
typedef struct tagMDICREATESTRUCT { // mdic
LPCTSTR szClass;
LPCTSTR szTitle;
HANDLE  hOwner;
int     x;
int     y;
int     cx;
int     cy;
DWORD   style;
LPARAM  lParam;
}MDICREATESTRUCT;
```

该结构的用处和 CREATESTRUCT 类似，只是它仅用于 MDI 子窗口的创建上，用来保存创建 MDI 子窗口时的窗口创建数据。域 lParam 保存一个指向 CCreateContext 结构的指针。

（4） WM_CREATE 的处理函数不同

创建 MDI 子窗口时发送的 WM_CREATE 消息由 CMDIChildWnd 的成员函数 OnCreate(LPCREATESTRUCT lpCreateStruct)处理。

OnCreate 函数仅仅从 lpCreateStruct 指向的数据中取出窗口创建数据，即指向 MDICREATESTRUCT 结构的指针，并从该结构得到指向 CCreateContext 结构的指针 pContext，然后调用虚拟函数 OnCreateHelper(lpCreateStruct，pContext)。

此处动态约束的结果是调用了 CFrameWnd 的成员函数 OnCreateHelper。SDI 应用程序的 OnCreate 也调用了 CFrameWnd::OnCreateHelper，所以后面的处理（创建视等）可参见 SDI 的流程了。

待 MDI 子窗口、视、文档对象创建完毕，多文档模板的 OpenDocumentFile 也调用 InitialUpdateFrame 来进行初始化。

5.3.4.3 MDI 子窗口的初始化和窗口的激活

（1）MDI 子窗口的初始化

完成了 MDI 子窗口、视、文档的创建之后，多文档模板的 OpenDocumenFile 调用边框窗口的虚拟函数 InitialUpdateFrame 进行初始化，该函数流程参见图 5-14。不过，这里 this 指针指向 CMDIChildWnd 对象，由于 C++虚拟函数的动态约束，初始化过程调用了 CMDIChildWnd 的 ActivateFrame 函数（不是 CFrameWnd 的 ActivateFrame），来显示 MDI 子窗口，更新菜单等等，见图 5-18。

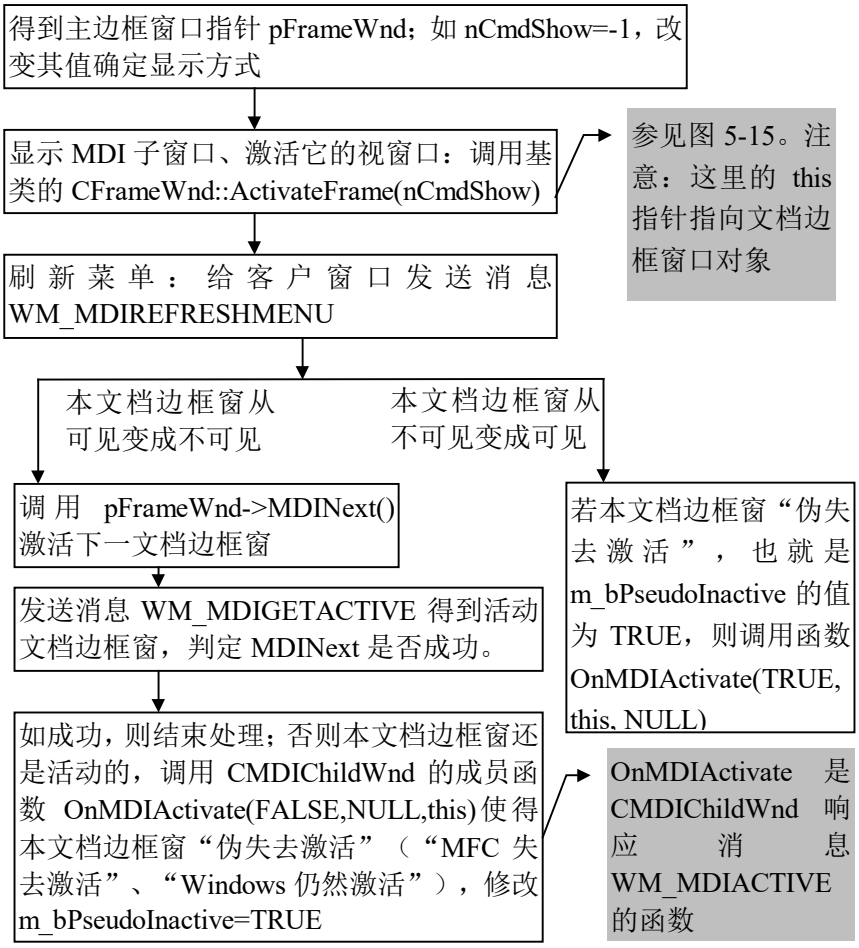


图 5-18　CMDIChildWnd 的 OnActiveFrame 函数

图 5-18 的说明：

第一，调用基类 CFrameWnd 的 ActivateFrame 显示窗口时，由于当前窗口是文档边框窗口，所以没有发送 WM_ACTIVATE 消息，而是发送消息 WM_MDIACTIVATE。

第二，由于 Windows 不处理 MDI 子窗口的激活，所以必须由 MFC 或者程序员来完成。当一个激活的 MDI 子窗口被隐藏后从可见变成不可见，但它仍然是活动的，这时需要把下一文档边框窗口激活以便用户看到的就是激活的窗口。在没有其他文档边框窗口时，则把该隐藏的文档边框窗口标记为“伪失去激活”。当一个文档边框窗口从不可见变成可见时，检查变量 m_bPseudoInactive，若真则该窗口从 Windows 角度看仍然是激活的，只需要调用

OnMDIActivate 把它改成“MFC 激活”。OnMDIActivate 把变量 m_bPseudoInactive 的值改变为 FALSE。

至此，MDI 子窗口初始化调用描述完毕。初始化将导致 MDI 窗口被显示、激活。下面讨论 MDI 子窗口的激活。

（2）MDI 子窗口的激活

通过给客户窗口发送消息 WM_MDIACTIVATE 来激活文档边框窗口。客户窗口发送 WM_MDIACTIVATE 消息给将被激活或者取消激活的 MDI 子窗口(文档边框窗口)，这些子窗口调用消息处理函数 OnMDIActivate 响应该消息 WM_MDIACTIVATE。关于 MDI 消息，见表 5-12。

用户转向一个子窗口（包括文档边框窗口）导致它的顶层(TOP LEVEL)边框窗口收到 WM_ACTIVATE 消息而被激活，子窗口是文档边框窗口的话将收到 WM_MDIACTIVATE 消息。

但是，一个边框窗口被其他方式激活时，它的文档边框窗口不会收到 WM_MDIACTIVATE 消息，而是最近一次被激活的文档边框窗口收到 WM_NCACTIVATE 消息。该消息由 CWnd::Default 缺省处理，用来重绘文档边框窗口的标题栏、边框等等。

MDI 子窗口用 OnMDIActiveate 函数处理 WM_MDIACTIVATE 消息。其原型如下：

```
void CMDIChildWnd::OnMDIActivate( BOOL bActivate,
    CWnd* pActivateWnd,CWnd* pDeactivateWnd );
```

其中：

参数 1 表示是激活(TRUE)，还是失去激活(FALSE)；

参数 2 表示将被激活的 MDI 子窗口；

参数 3 表示将被失去激活的 MDI 子窗口；

简单地说，该函数把 m_bPseudoInactive 的值改变为 FALSE，调用成员函数 OnActivateView 通知失去激活的子窗口的视它将失去激活，调用成员函数 OnActivateView 通知激活子窗口的视它将被激活。

至于 MDI 主边框窗口，它还是响应 WM_ACTIVATE 消息而被激活或相反。CMDIFrameWnd 没有提供该消息的处理函数，它调用基类 CFrameWnd 的处理函数 OnActivate。

现在，MDI 应用程序的启动过程描述完毕。

表5-12　MDI消息

消息	说明
WM_MDIACTIVATE	激活MDI Child窗口
WM_MDICASCADE	CASCADE排列MDI Child窗口
WM_MDICREATE	创建MDI Child窗口
WM_MDIDESTROY	销毁MDI Child窗口
WM_MDIGETACTIVE	得到活动的MDI Child窗口
WM_MDIICONARRANGE	安排最小化了的MDI Child窗口

WM_MDIMAXIMIZE	MDI Child窗口最大化
WM_MDINEXT	激活Z轴顺序的下一MDI Child窗口
WM_MDIREFRESHMENU	根据当前MDI Child窗口更新菜单
WM_MDIRESTORE	恢复MDI Child窗口
WM_MDISETMENU	根据当前MDI Child窗口设置菜单
WM_MDITITLE	TITLE安排MDI Child窗口

第6章　应用程序的退出

一个 Windows 应用程序启动之后，一般是进入消息循环，等待或者处理用户的输入，直到用户关闭应用程序窗口，退出应用程序为止。

例如，用户按主窗口的关闭按钮，或者选择执行系统菜单“关闭”，或者从“文件”菜单选择执行“退出”，都会导致主窗口被关闭。

当用户从“文件”菜单选择执行“退出”时，将发送 MFC 标准命令消息 ID_APP_EXIT。MFC 实现了函数 CWinApp::OnAppExit()来完成对该命令消息的缺省处理。

```
void CWinApp::OnAppExit()
{
// same as double-clicking on main window close box
ASSERT(m_pMainWnd != NULL);
m_pMainWnd->SendMessage(WM_CLOSE);
}
```

可以看出，其实现是向主窗口发送 WM_CLOSE 消息。主窗口处理完 WM_CLOSE 消息之后，关闭窗口，发送 WM_QUIT 消息，退出消息循环（见图 5-3），进而退出整个应用程序。

6.1　边框窗口对 WM_CLOSE 的处理

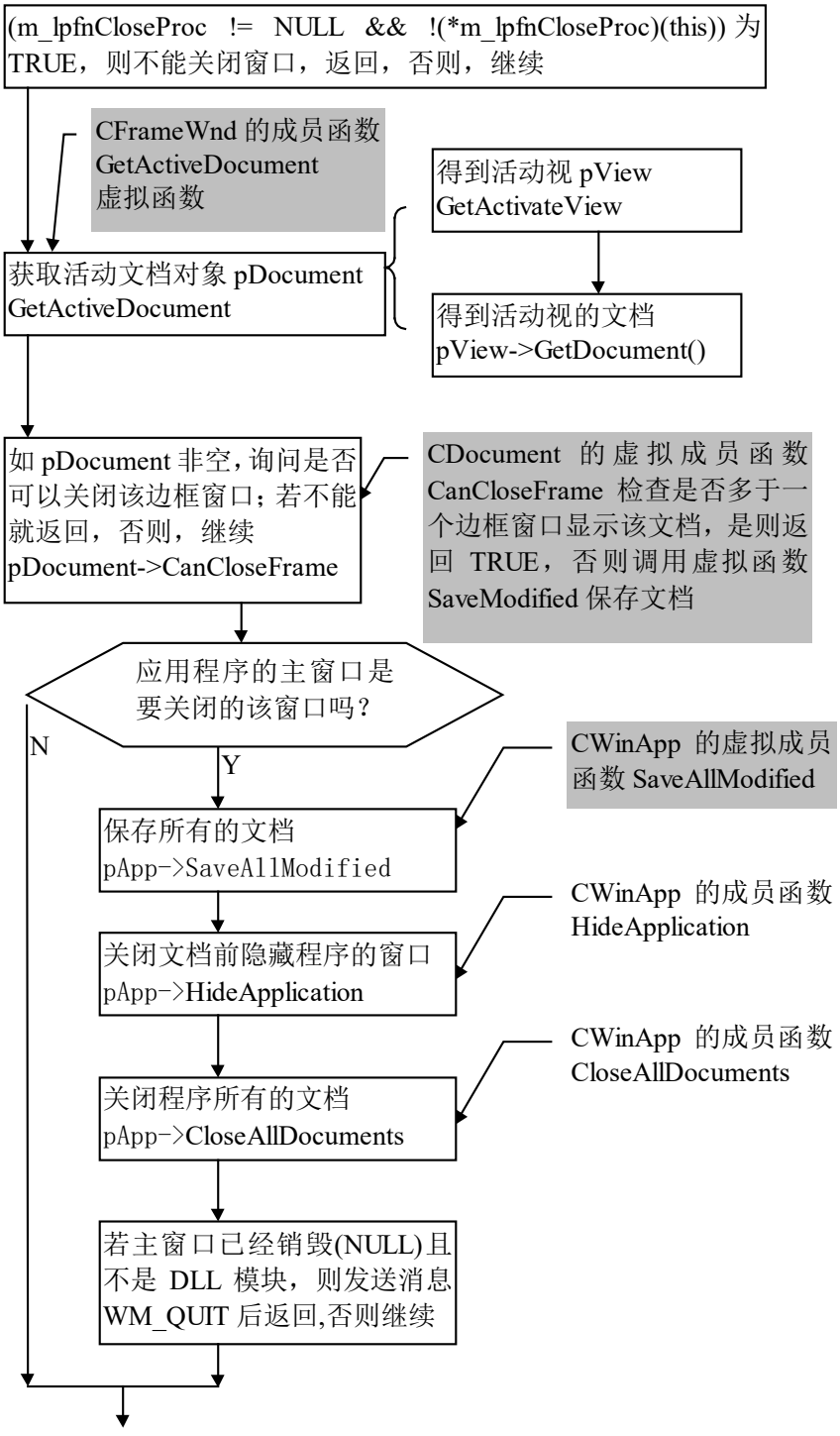
MFC 提供了函数 CFrameWnd::OnClose 来处理各类边框窗口的关闭：不仅包括 SDI 的边框窗口(从 CFrameWnd 派生)，而且包括 MDI 的主边框窗口（从 CMDIFrameWnd 派生）或者文档边框窗口(从 CMDIChildWnd 派生)的关闭。

该函数的原型如下，流程如图 6-1 所示：

```
void CFrameWnd::OnClose()
```

从图 6-1 中可以看出，它首先判断是否可以关闭窗口（m_lpfnCloseProc 是函数指针类型的成员变量，用于打印预览等情况下），然后，根据具体情况进行处理：

- 如果是主窗口被关闭，则关闭程序的所有文档，销毁所有窗口，退出程序；
- 如果不是主窗口被关闭，则是文档边框窗口被关闭，又分两种情况：若该窗口所显示的文档被且仅被该窗口显示，则关闭文档和文档窗口并销毁窗口；若该窗口显示的文档还被其他文档边框窗口所显示，则仅仅关闭和销毁文档窗口。



下面是处理 WM_CLOSE 消息中涉及的一些函数。

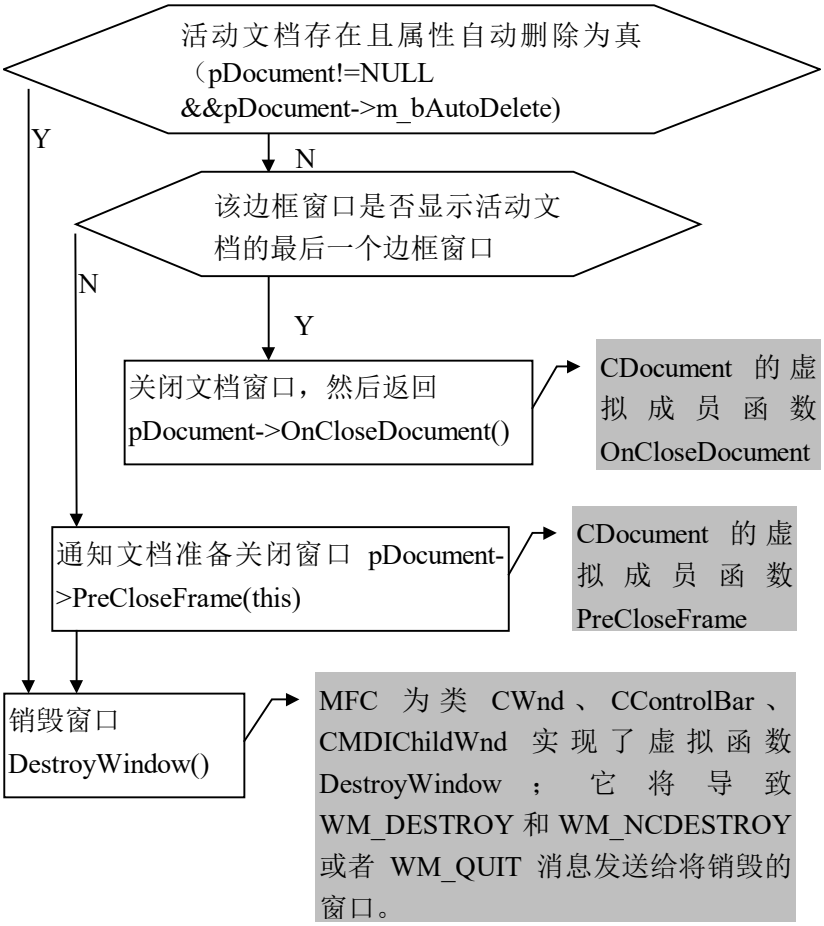


图 6-1　CFrameWnd 的 OnClose 函数

- BOOL CDocument::SaveModified()

该虚拟函数的缺省实现：首先调用 IsModifed 判断文档是否被修改，没有修改就返回，否则继续。

当询问用户是否保存被修改的文档时，若用户表示“cancel”，返回 FALSE；若用户表示“no”，则返回 TRUE；若用户表示“yes”，则存盘失败返回 FALSE，存盘成功返回 TRUE。存盘处理首先要得到被保存文件的名称，然后调用虚拟函数 OnSaveDocument 完成存盘工作，并使用 SetModifidFlag(FALSE)设置文档为没有修改。

- BOOL CDocument::OnSaveDocument(LPCTSTR lpszPathName)

该函数是虚拟函数，用来保存文件。其实现的功能和 OpOpenDocument 相反，但处理流程类似，描述如下：

根据 lpszPathName 打开文件 pFile；

使用 pFile 构造一个用于写入数据的 CArchive 对象，此对象用来保存数据到文件；

设置鼠标为时间瓶形状；

使用 Serialize 函数完成序列化写；

完毕，恢复鼠标的形状。

- CWinApp::SaveAllModified()

  CWinApp::CloseAllDocuments(BOOL bEndSession)

这两个函数都遍历模板管理器列表，并分别对列表中的模板管理器对象逐个调用

CDocManager 的同名成员函数：
CDocManager::SaveAllModified()
CDocManager::CloseAllDocuments(BOOL bEndSession)
这两个函数都遍历其文档模板列表，并分别对列表中的模板对象逐个调用 CDocTemplate 的同名成员函数：
CDocTemplate::SaveAllModified()
CDocTemplate::CloseAllDocuments(BOOL bEndSession)
这两个函数都遍历其文档列表，并分别对列表中的文档对象逐个调用 CDocuemnt 的成员函数：
CDocument::SaveModified()
CDocument::OnCloseDocument()

- CDocument::SaveModified()
  CDocument::OnCloseDocument()
  CDocument::SaveModified 前面已作了解释。OnCloseDocument 是一个虚拟函数，其流程如图 6-2 所示。

通过文档对象所对应的视，得到所有显示该文档的边框窗口的指针：在 SDI 程序关闭窗口时，获取的是主边框窗口；在 MDI 程序关闭窗口时，获取的是 MDI 子窗口。
然后，关闭并销毁对应的边框窗口。
如果文档对象的 m_bAutoDelete 为真，则销毁文档对象自身。

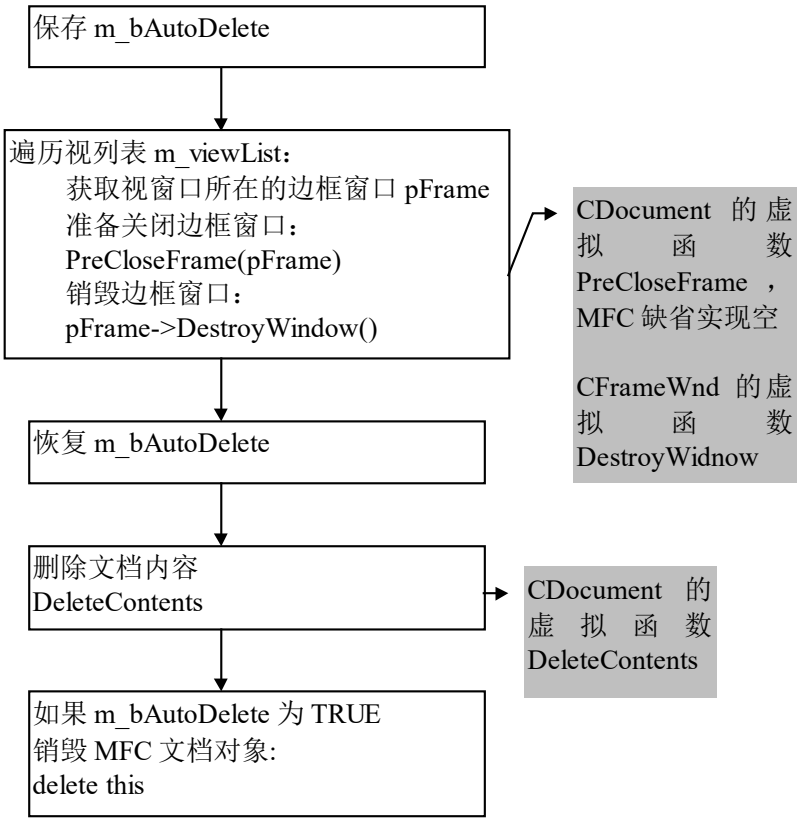


图 6-2　CDocument 的 OnCloseDocument 函数

6.2 窗口的销毁过程

6.2.1 DestroyWindow

从图 6-1、图 6-2 可以看出，销毁窗口是通过调用 DestroyWindow 来完成的。DestroyWindow 是 CWnd 类的一个虚拟函数。CWnd 实现了该函数，而 CMDIChildWnd 覆盖了该函数。

（1）CWnd::DestroyWindow()

主要就是调用::DestroyWindow 销毁 m_hWnd(必须非空)，同时销毁其菜单、定时器，以及完成其他清理工作。

::DestroyWindow 使将被销毁的窗口失去激活、失去输入焦点，并发送 WM_DESTROY、WM_NCDESTROY 消息到该窗口及其各级子窗口。如果被销毁的窗口是子窗口且没有设置 WM_NOPARENTNOTFIY 风格，则给其父窗口发送 WM_PARENTNOFITY 消息。

（2）CMDIChildWnd::DestroyWindow()

因为 MDI 子窗口不能使用::DestroyWindows 来销毁，所以 CMdiChildWnd 覆盖了该函数，CMDIChildWnd 主要是调用成员函数 MDIDestroy 给客户窗口(父窗口)发送消息 WM_MDIDESTROY，让客户窗口来销毁自己。

6.2.2 处理 WM_DESTROY 消息

消息处理函数 OnDestroy 处理 WM_DESTROY 消息，CFrameWnd、CMDIChildWnd、CWnd、CView 及其派生类（如 CEditView 等等）、CControlBar 等都提供了对该消息的处理函数。这里，主要解释边框、文档边框、视类的消息处理函数 OnDestroy。

（1） CWnd::OnDestroy()

调用缺省处理函数 Default()。

（2） CFrameWnd::OnDestroy()

首先，销毁工具栏的窗口；然后，设置菜单为缺省菜单；接着，如果要销毁的是主边框窗口，则通知 HELP 程序本应用程序将退出，没有其他程序使用 WINHELP 则关闭 WINHELP；最后调用 CWnd::OnDestroy。

（3） CMDIFrameWnd::OnDestroy()

首先，调整客户窗口的边界类型；然后，调用基类 CframeWnd 的 OnDestroy。这时，MDI 子窗口的工具栏窗口列表为空，故没有工具栏窗口可以销毁。

（4） CView::OnDestroy()

首先，判断自身是否是边框窗口的活动视，如果是则调用边框窗口的 SetActivateView 使自己失去激活；然后，调用基类 CWnd 的 OnDestroy。

6.2.3 处理 WM_NCDESTROY 消息

窗口的非客户区被销毁时，窗口接收 WM_NCDESTROY 消息，由 OnNcDestroy 处理 WM_NCDESTROY 消息。在 MFC 中，OnNcDestroy 是 Windows 窗口被销毁时调用的最后一个成员函数。

CWnd、CView 的某些派生类提供了对该消息的处理函数，这里只讨论 CWnd 的实现。

（1） CWnd::OnNcDestroy()

首先判断当前线程的主窗口是否是该窗口，如果是且模块非 DLL，则发送 WM_QUIT 消息，使得程序结束；

然后，判断当前线程的活动窗口是否是该窗口，如果是则设置活动窗口为 NULL；

接着，清理 Tooltip 窗口，调用 Default 由 Windows 缺省处理 WM_NCDESTROY 消息，UNSUBCLASS，把窗口句柄和 MFC 窗口对象分离(Detach)；

最后，调用虚拟函数 PostNcDestoy。

（2） PostNcDestoy

CWnd、CFrameWnd、CView、CControlBar 等都覆盖了该函数。文档边框窗口和边框窗口都使用 CFrameWnd::PostNcDestroy。

- CWnd::PostNcDestroy()

MFC 缺省实现空

- void CFrameWnd::PostNcDestroy()

调用 delete this 销毁自身这个 MFC 对象。

- void CView::PostNcDestroy()

调用 delete this 销毁自身这个 MFC 对象。

（3） 析构函数

delete this 导致析构函数的调用。需要提到的是 CFrameWnd 和 CView 的析构函数。

- CFrameWnd::~CFrameWnd()

边框窗口在创建时，把自身加入到模块-线程状态的边框窗口列表 m_frameList 中。现在，从列表中移走该窗口对象。

必要的话，删除 m_phWndDisable 数组。

- CView::~CView()

在视创建时，把自身加入到文档对象的视列表中。现在，从列表中移走该视对象。

应用程序调用 CloseAllDocument 关闭文档时。参数为 FALSE，它实际上并没有把视从列表中清除，而最后的清除是由析构函数来完成的。

至此，边框窗口关闭的过程讨论完毕。下面，结合具体情况——SDI 窗口的关闭、MDI 主窗口的关闭、MDI 子窗口的关闭——描述对 WM_CLOSE 消息的处理。

6.3 SDI 窗口、MDI 主、子窗口的关闭

参考图 6-1 分析 SDI 窗口、MDI 主、子窗口的关闭流程。

（1） SDI 窗口的关闭

在这种情况下，主窗口将被关闭。首先，关闭应用程序的文档对象。文档对象的虚拟函数 OnCloseDocument 调用时销毁了主窗口（Windows 窗口和 MFC 窗口对象），同时也导致视、工具条窗口的销毁。主窗口销毁后，应用程序的主窗口对象为空，故发送 WM_QUIT 消息结束程序。

（2） MDI 主窗口的关闭

首先，关闭应用程序的所有文档对象。文档对象的 OnCloseDocument 函数关闭文档时，将销毁文档对象对应的文档边框窗口和它的视窗口。这样，所有的 MDI 子窗口（包括其子窗口视）被销毁，但应用程序的主窗口还在。接着，调用 DestroyWindow 成员函数销毁主窗口自身，DestroyWindow 发现被销毁的是应用程序的主窗口，于是发送 WM_QUIT 消息结束

程序。

（3） MDI 子窗口（文档边框窗口）的关闭

在这种情况下，被关闭的不是主窗口。判断与该文档边框窗口对应的文档对象是否还被其他一个或者多个文档边框窗口使用，如果是，则仅仅销毁该文档边框窗口（包括其子窗口视）；否则，关闭文档，文档对象的 OnCloseDocument 将销毁该文档边框窗口（包括其子窗口视）。

第7章　MFC 的 DLL

一般的，在介绍 Windows 编程的书中讲述 DLL 的有关知识较多，而介绍 MFC 的书则比较少地提到。即使使用 MFC 来编写动态链接库，对于初步接触 DLL 的程序员来说，了解 DLL 的背景知识是必要的。另外，MFC 提供了新的手段来帮助编写 DLL 程序。所以，本节先简洁的介绍有关概念。

7.1　DLL 的背景知识

（1）　静态链接和动态链接

当前链接的目标代码(.obj)如果引用了一个函数却没有定义它，链接程序可能通过两种途径来解决这种从外部对该函数的引用：

- 静态链接

链接程序搜索一个或者多个库文件(标准库.lib)，直到在某个库中找到了含有所引用函数的对象模块，然后链接程序把这个对象模块拷贝到结果可执行文件(.exe)中。链接程序维护对该函数的所有引用，使它们指向该程序中现在含有该函数拷贝的地方。

- 动态链接

链接程序也是搜索一个或者多个库文件(输入库.lib)，当在某个库中找到了所引用函数的输入记录时，便把输入记录拷贝到结果可执行文件中，产生一次对该函数的动态链接。这里，输入记录不包含函数的代码或者数据，而是指定一个包含该函数代码以及该函数的顺序号或函数名的动态链接库。

当程序运行时，Windows 装入程序，并寻找文件中出现的任意动态链接。对于每个动态链接，Windows 装入指定的 DLL 并且把它映射到调用进程的虚拟地址空间（如果没有映射的话）。因此，调用和目标函数之间的实际链接不是在链接应用程序时一次完成的（静态），相反，是运行该程序时由 Windows 完成的（动态）。

这种动态链接称为加载时动态链接。还有一种动态链接方式下面会谈到。

（2）　动态链接的方法

链接动态链接库里的函数的方法如下：

- 加载时动态链接(Load_time dynamic linking)

如上所述。Windows 搜索要装入的 DLL 时，按以下顺序：

应用程序所在目录→当前目录→Windows SYSTEM 目录→Windows 目录→PATH 环境变量指定的路径。

- 运行时动态链接(Run_time dynamic linking)

程序员使用 LoadLibrary 把 DLL 装入内存并且映射 DLL 到调用进程的虚拟地址空间（如果已经作了映射，则增加 DLL 的引用计数）。首先，LoadLibrary 搜索 DLL，搜索顺序如同加载时动态链接一样。然后，使用 GetProcessAddress 得到 DLL 中输出函数的地址，并调用它。最后，使用 FreeLibrary 减少 DLL 的引用计数，当引用计数为 0 时，把 DLL 模块从当前进程的虚拟空间移走。

（3）　输入库(.lib)：

输入库以.lib 为扩展名，格式是 COFF(Common object file format)。COFF 标准库（静态链接库）的扩展名也是.lib。COFF 格式的文件可以用 dumpbin 来查看。

输入库包含了 DLL 中的输出函数或者输出数据的动态链接信息。当使用 MFC 创建 DLL 程序时，会生成输入库(.lib)和动态链接库(.dll)。

（4） 输出文件(.exp)

输出文件以.exp 为扩展名，包含了输出的函数和数据的信息，链接程序使用它来创建 DLL 动态链接库。

（5） 映像文件(.map)

映像文件以.map 为扩展名，包含了如下信息：

模块名、时间戳、组列表（每一组包含了形式如 section::offset 的起始地址，长度、组名、类名）、公共符号列表(形式如 section::offset 的地址，符号名，虚拟地址 flat address，定义符号的.obj 文件)、入口点如 section::offset、fixup 列表。

（6） lib.exe 工具

它可以用来创建输入库和输出文件。通常，不用使用 lib.exe，如果工程目标是创建 DLL 程序，链接程序会完成输入库的创建。

更详细的信息可以参见 MFC 使用手册和文档。

（7） 链接规范（Linkage Specification ）

这是指链接采用不同编程语言写的函数(Function)或者过程(Procedure)的链接协议。MFC 所支持的链接规范是“C”和“C++”，缺省的是“C++”规范，如果要声明一个“C”链接的函数或者变量，则一般采用如下语法：

```
#if defined(__cplusplus)
extern "C"
{
#endif

//函数声明（function declarations）
…
//变量声明(variables declarations)
#if defined(__cplusplus)
}
#endif
```

所有的C标准头文件都是用如上语法声明的，这样它们在C++环境下可以使用。

（8） 修饰名(Decoration name)

“C”或者“C++”函数在内部（编译和链接）通过修饰名识别。修饰名是编译器在编译函数定义或者原型时生成的字符串。有些情况下使用函数的修饰名是必要的，如在模块定义文件里头指定输出“C++”重载函数、构造函数、析构函数，又如在汇编代码里调用“C””或“C++”函数等。

修饰名由函数名、类名、调用约定、返回类型、参数等共同决定。

7.2 调用约定

调用约定(Calling convention)决定以下内容：函数参数的压栈顺序，由调用者还是被调用者把参数弹出栈，以及产生函数修饰名的方法。MFC 支持以下调用约定：

（1） _cdecl

按从右至左的顺序压参数入栈，由调用者把参数弹出栈。对于“C”函数或者变量，修饰名是在函数名前加下划线。对于“C++”函数，有所不同。

如函数 void test(void)的修饰名是_test；对于不属于一个类的“C++”全局函数，修饰名是?test@@ZAXXZ。
这是 MFC 缺省调用约定。由于是调用者负责把参数弹出栈，所以可以给函数定义个数不定的参数，如 printf 函数。

（2） _stdcall

按从右至左的顺序压参数入栈，由被调用者把参数弹出栈。对于“C”函数或者变量，修饰名以下划线为前缀，然后是函数名，然后是符号“@”及参数的字节数，如函数 int func(int a, double b)的修饰名是_func@12。对于“C++”函数，则有所不同。
所有的 Win32 API 函数都遵循该约定。

（3） _fastcall

头两个 DWORD 类型或者占更少字节的参数被放入 ECX 和 EDX 寄存器，其他剩下的参数按从右到左的顺序压入栈。由被调用者把参数弹出栈，对于“C”函数或者变量，修饰名以“@”为前缀，然后是函数名，接着是符号“@”及参数的字节数，如函数 int func(int a, double b)的修饰名是@func@12。对于“C++”函数，有所不同。
未来的编译器可能使用不同的寄存器来存放参数。

（4） thiscall

仅仅应用于“C++”成员函数。this 指针存放于 CX 寄存器，参数从右到左压栈。thiscall 不是关键词，因此不能被程序员指定。

（5） naked call

采用 1-4 的调用约定时，如果必要的话，进入函数时编译器会产生代码来保存 ESI，EDI，EBX，EBP 寄存器，退出函数时则产生代码恢复这些寄存器的内容。naked call 不产生这样的代码。
naked call 不是类型修饰符，故必须和_declspec 共同使用，如下：

```
__declspec( naked ) int func( formal_parameters )
{
    // Function body
}
```

（6） 过时的调用约定

原来的一些调用约定可以不再使用。它们被定义成调用约定_stdcall 或者_cdecl。例如：

```
#define CALLBACK      __stdcall
#define WINAPI        __stdcall
#define WINAPIV       __cdecl
#define APIENTRY      WINAPI
#define APIPRIVATE    __stdcall
#define PASCAL        __stdcall
```

表 7-1 显示了一个函数在几种调用约定下的修饰名（表中的“C++”函数指的是“C++”全局函数，不是成员函数），函数原型是 void CALLTYPE test(void)，CALLTYPE 可以是_cdecl、_fastcall、_stdcall。

表7-1 不同调用约定下的修饰名

调用约定	extern “C”或.C 文件	.cpp, .cxx 或/TP 编译开关
_cdecl	_test	?test@@ZAXXZ

_fastcall	@test@0	?test@@YIXXZ
_stdcall	_test@0	?test@@YGXXZ

7.2.1 MFC 的 DLL 应用程序的类型

（1） 静态链接到 MFC 的规则 DLL 应用程序

该类 DLL 应用程序里头的输出函数可以被任意 Win32 程序使用，包括使用 MFC 的应用程序。输入函数有如下形式：

```
extern "C" EXPORT YourExportedFunction( );
```

如果没有 extern “C”修饰，输出函数仅仅能从 C++代码中调用。

DLL 应用程序从 CWinApp 派生，但没有消息循环。

（2） 动态链接到 MFC 的规则 DLL 应用程序

该类 DLL 应用程序里头的输出函数可以被任意 Win32 程序使用，包括使用 MFC 的应用程序。但是，所有从 DLL 输出的函数应该以如下语句开始：

```
AFX_MANAGE_STATE(AfxGetStaticModuleState( ))
```

此语句用来正确地切换 MFC 模块状态。关于 MFC 的模块状态，后面第 9 章有详细的讨论。其他方面同静态链接到 MFC 的规则 DLL 应用程序。

（3） 扩展 DLL 应用程序

该类 DLL 应用程序动态链接到 MFC，它输出的函数仅可以被使用 MFC 且动态链接到 MFC 的应用程序使用。和规则 DLL 相比，有以下不同：

第一， 它没有一个从 CWinApp 派生的对象；

第二， 它必须有一个 DllMain 函数；

第三， DllMain 调用 AfxInitExtensionModule 函数，必须检查该函数的返回值，如果返回 0，DllMmain 也返回 0；

第四， 如果它希望输出 CRuntimeClass 类型的对象或者资源(Resources)，则需要提供一个初始化函数来创建一个 CDynLinkLibrary 对象。并且，有必要把初始化函数输出。

第五， 使用扩展 DLL 的 MFC 应用程序必须有一个从 CWinApp 派生的类，而且，一般在 InitInstance 里调用扩展 DLL 的初始化函数。

为什么要这样做和具体的代码形式，将在后面 9.4.2 节说明。

MFC 类库也是以 DLL 的形式提供的。通常所说的动态链接到 MFC 的 DLL，指的就是实现 MFC 核心功能的 MFCXX.DLL 或者 MFCXXD.DLL（XX 是版本号，XXD 表示调试版）。至于提供 OLE（MFCOXXD.DLL 或者 MFCOXX0.DLL）和 NET（MFCNXXD.DLL 或者 MFCNXX.DLL）服务的 DLL 就是动态链接到 MFC 核心 DLL 的扩展 DLL。

其实，MFCXX.DLL 可以认为是扩展 DLL 的一个特例，因为它也具备扩展 DLL 的上述特点。

7.3 DLL 的几点说明

（1） DLL 应用程序的入口点是 DllMain。

对程序员来说，DLL 应用程序的入口点是 DllMain。

DllMain 负责初始化(Initialization)和结束(Termination)工作，每当一个新的进程或者该进程的新的线程访问 DLL 时，或者访问 DLL 的每一个进程或者线程不再使用 DLL 或者结束时，

都会调用 DllMain。但是，使用 TerminateProcess 或 TerminateThread 结束进程或者线程，不会调用 DllMain。
DllMain 的函数原型符合 DllEntryPoint 的要求，有如下结构：

```
BOOL    WINAPI    DllMain (HANDLE hInst,
            ULONG ul_reason_for_call,LPVOID lpReserved)
{
switch( ul_reason_for_call ) {
    case DLL_PROCESS_ATTACH:
    ...
    case DLL_THREAD_ATTACH:
    ...
    case DLL_THREAD_DETACH:
    ...
    case DLL_PROCESS_DETACH:
    ...
}
return TRUE;
}
```

其中：
参数 1 是模块句柄；
参数 2 是指调用 DllMain 的类别，四种取值：新的进程要访问 DLL；新的线程要访问 DLL；一个进程不再使用 DLL(Detach from DLL)；一个线程不再使用 DLL(Detach from DLL)。
参数 3 保留。
如果程序员不指定 DllMain，则编译器使用它自己的 DllMain，该函数仅仅返回 TRUE。
规则 DLL 应用程序使用了 MFC 的 DllMain，它将调用 DLL 程序的应用程序对象(从 CWinApp 派生)的 InitInstance 函数和 ExitInstance 函数。
扩展 DLL 必须实现自己的 DllMain。

（2） _DllMainCRTStartup

为了使用“C”运行库(CRT，C Run time Library)的 DLL 版本（多线程），一个 DLL 应用程序必须指定_DllMainCRTStartup 为入口函数，DLL 的初始化函数必须是 DllMain。
_DllMainCRTStartup 完成以下任务：当进程或线程捆绑(Attach)到 DLL 时为“C”运行时的数据(C Runtime Data)分配空间和初始化并且构造全局“C++”对象，当进程或者线程终止使用 DLL(Detach)时，清理 C Runtime Data 并且销毁全局“C++”对象。它还调用 DllMain 和 RawDllMain 函数。
RawDllMain 在 DLL 应用程序动态链接到 MFC DLL 时被需要，但它是静态的链接到 DLL 应用程序的。在讲述状态管理时解释其原因。

（3） DLL 的函数和数据

DLL 的函数分为两类：输出函数和内部函数。输出函数可以被其他模块调用，内部函数在定义它们的 DLL 程序内部使用。
虽然 DLL 可以输出数据，但一般的 DLL 程序的数据仅供内部使用。

（4） DLL 程序和调用其输出函数的程序的关系

DLL 模块被映射到调用它的进程的虚拟地址空间。
DLL 使用的内存从调用进程的虚拟地址空间分配，只能被该进程的线程所访问。

DLL 的句柄可以被调用进程使用；调用进程的句柄可以被 DLL 使用。
DLL 使用调用进程的栈。
DLL 定义的全局变量可以被调用进程访问；DLL 可以访问调用进程的全局数据。使用同一 DLL 的每一个进程都有自己的 DLL 全局变量实例。如果多个线程并发访问同一变量，则需要使用同步机制；对一个 DLL 的变量，如果希望每个使用 DLL 的线程都有自己的值，则应该使用线程局部存储(TLS，Thread Local Strorage)。

7.4 输出函数的方法

（1） 传统的方法

在模块定义文件的 EXPORT 部分指定要输入的函数或者变量。语法格式如下：

entryname[=internalname] [@ordinal[NONAME]] [DATA] [PRIVATE]

其中：

entryname 是输出的函数或者数据被引用的名称；

internalname 同 entryname;

@ordinal 表示在输出表中的顺序号(index)；

NONAME 仅仅在按顺序号输出时被使用（不使用 entryname）；

DATA 表示输出的是数据项，使用 DLL 输出数据的程序必须声明该数据项为 _declspec(dllimport)。

上述各项中，只有 entryname 项是必须的，其他可以省略。

对于“C”函数来说，entryname 可以等同于函数名；但是对“C++”函数（成员函数、非成员函数）来说，entryname 是修饰名。可以从.map 映像文件中得到要输出函数的修饰名，或者使用 DUMPBIN /SYMBOLS 得到，然后把它们写在.def 文件的输出模块。DUMPBIN 是 VC 提供的一个工具。

如果要输出一个“C++”类，则把要输出的数据和成员的修饰名都写入.def 模块定义文件。

（2） 在命令行输出

对链接程序 LINK 指定/EXPORT 命令行参数，输出有关函数。

（3） 使用 MFC 提供的修饰符号_declspec(dllexport)

在要输出的函数、类、数据的声明前加上_declspec(dllexport)的修饰符，表示输出。MFC 提供了一些宏，就有这样的作用，如表 7-2 所示。

表7-2 MFC定义的输入输出修饰符

宏名称	宏内容
AFX_CLASS_IMPORT	__declspec(dllexport)
AFX_API_IMPORT	__declspec(dllexport)
AFX_DATA_IMPORT	__declspec(dllexport)
AFX_CLASS_EXPORT	__declspec(dllexport)
AFX_API_EXPORT	__declspec(dllexport)
AFX_DATA_EXPORT	__declspec(dllexport)
AFX_EXT_CLASS	#ifdef _AFXEXT AFX_CLASS_EXPORT #else AFX_CLASS_IMPORT
AFX_EXT_API	#ifdef _AFXEXT AFX_API_EXPORT

	#else AFX_API_IMPORT
AFX_EXT_DATA	#ifdef _AFXEXT AFX_DATA_EXPORT #else AFX_DATA_IMPORT
AFX_EXT_DATADEF	

像 AFX_EXT_CLASS 这样的宏，如果用于 DLL 应用程序的实现中，则表示输出（因为 _AFX_EXT 被定义，通常是在编译器的标识参数中指定该选项/D_AFX_EXT）；如果用于使用 DLL 的应用程序中，则表示输入（_AFX_EXT 没有定义）。

要输出整个的类，对类使用_declspec(_dllexpot)；要输出类的成员函数，则对该函数使用_declspec(_dllexport)。如：

```
class AFX_EXT_CLASS CTextDoc : public CDocument
{
    …
}

extern "C" AFX_EXT_API void WINAPI InitMYDLL();
```

这几种方法中，最好采用第三种，方便好用；其次是第一种，如果按顺序号输出，调用效率会高些；最次是第二种。

在“C++”下定义“C”函数，需要加 extern “C”关键词。输出的“C”函数可以从“C”代码里调用。

第8章　MFC 的进程和线程

8.1　Win32 的进程和线程概念

进程是一个可执行的程序，由私有虚拟地址空间、代码、数据和其他操作系统资源（如进程创建的文件、管道、同步对象等）组成。一个应用程序可以有一个或多个进程，一个进程可以有一个或多个线程，其中一个是主线程。

线程是操作系统分时调度分配 CPU 时间的基本实体。一个线程可以执行程序的任意部分的代码，即使这部分代码被另一个线程并发地执行；一个进程的所有线程共享它的虚拟地址空间、全局变量和操作系统资源。

之所以有线程这个概念，是因为以线程而不是进程为调度对象效率更高：

- 由于创建新进程必须加载代码，而线程要执行的代码已经被映射到进程的地址空间，所以创建、执行线程的速度比进程更快。
- 一个进程的所有线程共享进程的地址空间和全局变量，所以简化了线程之间的通讯。

8.2　Win32 的进程处理简介

因为 MFC 没有提供类处理进程，所以直接使用了 Win32 API 函数。

8.2.1 进程的创建

调用 CreateProcess 函数创建新的进程，运行指定的程序。CreateProcess 的原型如下：

```
BOOL CreateProcess(
LPCTSTR lpApplicationName,
LPTSTR lpCommandLine,
LPSECURITY_ATTRIBUTES lpProcessAttributes,
LPSECURITY_ATTRIBUTES lpThreadAttributes,
BOOL bInheritHandles,
DWORD dwCreationFlags,
LPVOID lpEnvironment,
LPCTSTR lpCurrentDirectory,
LPSTARTUPINFO lpStartupInfo,
LPPROCESS_INFORMATION lpProcessInformation
);
```

其中：

lpApplicationName 指向包含了要运行模块名字的字符串。

lpCommandLine 指向命令行字符串。

lpProcessAttributes 描述进程的安全性属性，NT 下有用。

lpThreadAttributes 描述进程初始线程（主线程）的安全性属性，NT 下有用。

bInHeritHandles 表示子进程（被创建的进程）是否可以继承父进程的句柄。可以继承的句柄有线程句柄、有名或无名管道、互斥对象、事件、信号量、映像文件、普通文件和通讯端口等；还有一些句柄不能被继承，如内存句柄、DLL 实例句柄、GDI 句柄、URER 句柄等等。子进程继承的句柄由父进程通过命令行方式或者进程间通讯（IPC）方式由父进程传递给它。
dwCreationFlags 表示创建进程的优先级类别和进程的类型。创建进程的类型分控制台进程、调试进程等；优先级类别用来控制进程的优先级别，分 Idle、Normal、High、Real_time 四个类别。
lpEnviroment 指向环境变量块，环境变量可以被子进程继承。
lpCurrentDirectory 指向表示当前目录的字符串，当前目录可以继承。
lpStartupInfo 指向 StartupInfo 结构，控制进程的主窗口的出现方式。
lpProcessInformation 指向 PROCESS_INFORMATION 结构，用来存储返回的进程信息。

从其参数可以看出创建一个新的进程需要指定什么信息。

从上面的解释可以看出，一个进程包含了很多信息。若进程创建成功的话，返回一个进程信息结构类型的指针。进程信息结构如下：

```
typedef struct _PROCESS_INFORMATION {
HANDLE hProcess;
HANDLE hThread;
DWORD dwProcessId;
DWORD dwThreadId;
}PROCESS_INFORMATION;
```

进程信息结构包括进程句柄，主线程句柄，进程 ID，主线程 ID。

8.2.2 进程的终止

进程在以下情况下终止：

- 调用 ExitProcess 结束进程；
- 进程的主线程返回，隐含地调用 ExitProcess 导致进程结束；
- 进程的最后一个线程终止；
- 调用 TerminateProcess 终止进程。
- 当要结束一个 GDI 进程时，发送 WM_QUIT 消息给主窗口，当然也可以从它的任一线程调用 ExitProcess。

8.3 Win32 的线程

8.3.1 线程的创建

使用 CreateThread 函数创建线程，CreateThread 的原型如下：

```
HANDLE CreateThread(
LPSECURITY_ATTRIBUTES lpThreadAttributes,
DWORD dwStackSize,
LPTHREAD_START_ROUTINE lpStartAddress,
```

```
LPVOID lpParameter,
DWORD dwCreationFlags,  // creation flags
LPDWORD lpThreadId
);
```

其中：

lpThreadAttributes 表示创建线程的安全属性，NT 下有用。

dwStackSize 指定线程栈的尺寸，如果为 0 则与进程主线程栈相同。

lpStartAddress 指定线程开始运行的地址。

lpParameter 表示传递给线程的 32 位的参数。

dwCreateFlages 表示是否创建后挂起线程(取值 CREATE_SUSPEND)，挂起后调用 ResumeThread 继续执行。

lpThreadId 用来存放返回的线程 ID。

- 线程的优先级别

进程的每个优先级类包含了五个线程的优先级水平。在进程的优先级类确定之后，可以改变线程的优先级水平。用 SetPriorityClass 设置进程优先级类，用 SetThreadPriority 设置线程优先级水平。

Normal 级的线程可以被除了 Idle 级以外的任意线程抢占。

8.3.2 线程的终止

以下情况终止一个线程：

- 调用了 ExitThread 函数；
- 线程函数返回：主线程返回导致 ExitProcess 被调用，其他线程返回导致 ExitThread 被调用；
- 调用 ExitProcess 导致进程的所有线程终止；
- 调用 TerminateThread 终止一个线程；
- 调用 TerminateProcess 终止一个进程时，导致其所有线程的终止。

当用TerminateProcess或者TerminateThread终止进程或线程时，DLL的入口函数DllMain不会被执行（如果有DLL的话）。

8.3.3 线程局部存储

如果希望每个线程都可以有线程局部(Thread local)的静态存储数据，可以使用 TLS 线程局部存储技术。TLS 为进程分配一个 TLS 索引，进程的每个线程通过这个索引存取自己的数据变量的拷贝。

TLS 对 DLL 是非常有用的。当一个新的进程使用 DLL 时，在 DLL 入口函数 DllMain 中使用 TlsAlloc 分配 TLS 索引，TLS 索引就作为进程私有的全局变量被保存；以后，当该进程的新的线程使用 DLL 时(Attahced to DLL)，DllMain 给它分配动态内存并且使用 TlsSetValue 把线程私有的数据按索引保存。DLL 函数可以使用 TlsGetValue 按索引读取调用线程的私有数据。

TLS 函数如下：

- DWORD TlsAlloc()

在进程或 DLL 初始化时调用，并且把返回值（索引值）作为全局变量保存。

- BOOL TlsSetValue(
    DWORD dwTlsIndex, //TLS index to set value for
    LPVOID lpTlsValue //value to be stored
  );

其中：
dwTlsIndex 是 TlsAlloc 分配的索引。
lpTlsValue 是线程在 TLS 槽中存放的数据指针，指针指向线程要保存的数据。
线程首先分配动态内存并保存数据到此内存中，然后调用 TlsSetValue 保存内存指针到 TLS 槽。

- LPVOID TlsGetValue(
  DWORD dwTlsIndex // TLS index to retrieve value for
  );

其中：
dwTlsIndex 是 TlsAlloc 分配的索引。
当要存取保存的数据时，使用索引得到数据指针。

- BOOL TlsFree(
  DWORD dwTlsIndex // TLS index to free
  );

其中：
dwTlsIndex 是 TlsAlloc 分配的索引。
当每一个线程都不再使用局部存储数据时，线程释放它分配的动态内存。在 TLS 索引不再需要时，使用 TlsFree 释放索引。

8.4 线程同步

同步可以保证在一个时间内只有一个线程对某个资源（如操作系统资源等共享资源）有控制权。共享资源包括全局变量、公共数据成员或者句柄等。同步还可以使得有关联交互作用的代码按一定的顺序执行。
Win32 提供了一组对象用来实现多线程的同步。
这些对象有两种状态：获得信号(Signaled)或者没有或则信号(Not signaled)。线程通过 Win32 API 提供的同步等待函数（Wait functions）来使用同步对象。一个同步对象在同步等待函数调用时被指定，调用同步函数地线程被阻塞(blocked)，直到同步对象获得信号。被阻塞的线程不占用 CPU 时间。

8.4.1 同步对象

同步对象有：Critical_section（关键段），Event（事件），Mutex（互斥对象），Semaphores（信号量）。
下面，解释怎么使用这些同步对象。
（1） 关键段对象：
首先，定义一个关键段对象 cs：
CRITICAL_SECTION cs;
然后，初始化该对象。初始化时把对象设置为 NOT_SINGALED，表示允许线程使用资源：

InitializeCriticalSection(&cs);
如果一段程序代码需要对某个资源进行同步保护，则这是一段关键段代码。在进入该关键段代码前调用 EnterCriticalSection 函数，这样，其他线程都不能执行该段代码，若它们试图执行就会被阻塞。
完成关键段的执行之后，调用 LeaveCriticalSection 函数，其他的线程就可以继续执行该段代码。如果该函数不被调用，则其他线程将无限期的等待。

（2） 事件对象

首先，调用 CreateEvent 函数创建一个事件对象，该函数返回一个事件句柄。然后，可以设置（SetEvent）或者复位(ResetEvent)一个事件对象，也可以发一个事件脉冲（PlusEvent），即设置一个事件对象，然后复位它。复位有两种形式：自动复位和人工复位。在创建事件对象时指定复位形式。。
自动复位：当对象获得信号后，就释放下一个可用线程（优先级别最高的线程；如果优先级别相同，则等待队列中的第一个线程被释放）。
人工复位：当对象获得信号后，就释放所有可利用线程。
最后，使用 CloseHandle 销毁创建的事件对象。

（3） 互斥对象

首先，调用 CreateMutex 创建互斥对象；然后，调用等待函数，可以的话利用关键资源；最后，调用 RealseMutex 释放互斥对象。
互斥对象可以在进程间使用，但关键段对象只能用于同一进程的线程之间。

（4） 信号量对象

在 Win32 中，信号量的数值变为 0 时给以信号。在有多个资源需要管理时可以使用信号量对象。
首先，调用 CreateSemaphore 创建一个信号量；然后，调用等待函数，如果允许的话，则利用关键资源；最后，调用 RealeaseSemaphore 释放信号量对象。

（5） 此外，还有其他句柄可以用来同步线程：

文件句柄（FILE HANDLES）
命名管道句柄（NAMED PIPE HANDELS）
控制台输入缓冲区句柄（CONSOLE INPUT BUFFER HANDLES）
通讯设备句柄（COMMUNICTION DEVICE HANDLES）
进程句柄（PROCESS HANDLES）
线程句柄（THREAD HANDLES）
例如，当一个进程或线程结束时，进程或线程句柄获得信号，等待该进程或者线程结束的线程被释放。

8.4.2 等待函数

Win32 提供了一组等待函数用来让一个线程阻塞自己的执行。等待函数分三类：

（1） 等待单个对象的(FOR SINGLE OBJECT):

这类函数包括：

SignalObjectAndWait
WaitForSingleObject
WaitForSingleObjectEx

函数参数包括同步对象的句柄和等待时间等。
在以下情况下等待函数返回：

同步对象获得信号时返回；
等待时间达到了返回：如果等待时间不限制(Infinite)，则只有同步对象获得信号才返回；如果等待时间为 0，则在测试了同步对象的状态之后马上返回。

（2） 等待多个对象的(FOR MULTIPLE OBJECTS)

这类函数包括：

WaitForMultipleObjects

WaitForMultipleObjectsEx

MsgWaitForMultipleObjects

MsgWaitForMultipleObjectsEx

函数参数包括同步对象的句柄，等待时间，是等待一个还是多个同步对象等等。
在以下情况下等待函数返回：
一个或全部同步对象获得信号时返回（在参数中指定是等待一个或多个同步对象）；
等待时间达到了返回：如果等待时间不限制(Infinite)，则只有同步对象获得信号才返回；如果等待时间为 0，则在测试了同步对象的状态之后马上返回。

（3） 可以发出提示的函数(ALTERABLE)

这类函数包括：

MsgWaitForMultipleObjectsEx

SignalObjectAndWait

WaitForMultipleObjectsEx

WaitForSingleObjectEx

这些函数主要用于重叠(Overlapped)的 I/O（异步 I/O）。

8.5 MFC 的线程处理

在 Win32 API 的基础之上，MFC 提供了处理线程的类和函数。处理线程的类是 CWinThread，函数是 AfxBeginThread、AfxEndThread 等。
表 5-6 解释了 CWinThread 的成员变量和函数。
CWinThread 是 MFC 线程类，它的成员变量 m_hThread 和 m_hThreadID 是对应的 Win32 线程句柄和线程 ID。
MFC 明确区分两种线程：用户界面线程(User interface thread)和工作者线程(Worker thread)。用户界面线程一般用于处理用户输入并对用户产生的事件和消息作出应答。工作者线程用于完成不要求用户输入的任务，如耗时计算。
Win32 API 并不区分线程类型，它只需要知道线程的开始地址以便它开始执行线程。MFC 为用户界面线程特别地提供消息泵来处理用户界面的事件。CWinApp 对象是用户界面线程对象的一个例子，CWinApp 从类 CWinThread 派生并处理用户产生的事件和消息。

8.5.1 创建用户界面线程

通过以下步骤创建一个用户界面线程：

- 从 CWinThread 派生一个有动态创建能力的类。使用 DECLARE_DYNCREATE 和 IMPLEMENT_DYNCREATE 宏来支持动态创建。
- 覆盖 CWinThread 的一些虚拟函数，可以覆盖的函数见表 5-4 关于 CWinThread 的部分。其中，函数 InitInstance 是必须覆盖的，ExitInstance 通常是要覆盖的。

- 使用 AfxBeginThread 创建 MFC 线程对象和 Win32 线程对象。如果创建线程时没有指定 CREATE_SUSPENDED，则开始执行线程。
- 如果创建线程是指定了 CREATE_SUSPENDED，则在适当的地方调用函数 ResumeThread 开始执行线程。

8.5.2 创建工作者线程

程序员不必从 CWinThread 派生新的线程类，只需要提供一个控制函数,由线程启动后执行该函数。
然后，使用 AfxBeginThread 创建 MFC 线程对象和 Win32 线程对象。如果创建线程时没有指定 CREATE_SUSPENDED（创建后挂起），则创建的新线程开始执行。
如果创建线程是指定了 CREATE_SUSPENDED，则在适当的地方调用函数 ResumeThread 开始执行线程。
虽然程序员没有从 CWinThread 派生类，但是 MFC 给工作者线程提供了缺省的 CWinThread 对象。

8.5.3 AfxBeginThread

用户界面线程和工作者线程都是由 AfxBeginThread 创建的。现在，考察该函数：MFC 提供了两个重载版的 AfxBeginThread，一个用于用户界面线程，另一个用于工作者线程，分别有如下的原型和过程：

（1） 用户界面线程的 AfxBeginThread

用户界面线程的 AfxBeginThread 的原型如下：

```
CWinThread* AFXAPI AfxBeginThread(
    CRuntimeClass* pThreadClass,
    int nPriority,
    UINT nStackSize,
    DWORD dwCreateFlags,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs)
```

其中：

参数 1 是从 CWinThread 派生的 RUNTIME_CLASS 类；
参数 2 指定线程优先级，如果为 0，则与创建该线程的线程相同；
参数 3 指定线程的堆栈大小，如果为 0，则与创建该线程的线程相同；
参数 4 是一个创建标识，如果是 CREATE_SUSPENDED，则在悬挂状态创建线程，在线程创建后线程挂起，否则线程在创建后开始线程的执行。
参数 5 表示线程的安全属性，NT 下有用。

（2） 工作者线程的 AfxBeginThread

工作者线程的 AfxBeginThread 的原型如下：

```
CWinThread* AFXAPI AfxBeginThread(
    AFX_THREADPROC pfnThreadProc,
    LPVOID pParam,
    int nPriority,
    UINT nStackSize,
```

DWORD dwCreateFlags,

LPSECURITY_ATTRIBUTES lpSecurityAttrs)

其中：

参数 1 指定控制函数的地址；

参数 2 指定传递给控制函数的参数；

参数 3、4、5 分别指定线程的优先级、堆栈大小、创建标识、安全属性，含义同用户界面线程。

（3） AfxBeginThread 创建线程的流程

不论哪个 AfxBeginThread，首先都是创建 MFC 线程对象，然后创建 Win32 线程对象。在创建 MFC 线程对象时，用户界面线程和工作者线程的创建分别调用了不同的构造函数。用户界面线程是从 CWinThread 派生的，所以，要先调用派生类的缺省构造函数，然后调用 CWinThread 的缺省构造函数。图 8-1 中两个构造函数所调用的 CommonConstruct 是 MFC 内部使用的成员函数。

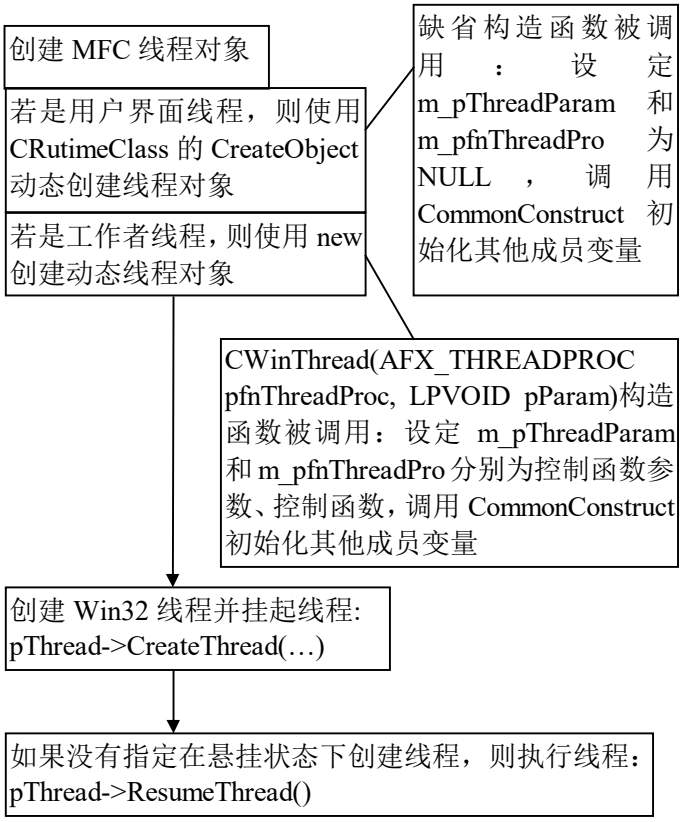


图 8-1 AfxBeginThread 创建线程的流程

8.5.4 CreateThread 和_AfxThreadEntry

MFC 使用 CWinThread::CreateThread 创建线程，不论对工作者线程或用户界面线程，都指定线程的入口函数是_AfxThreadEntry。_AfxThreadEntry 调用 AfxInitThread 初始化线程。CreateThread 和_AfxThreadEntry 在线程的创建过程中使用同步手段交互等待、执行。CreateThread 由创建线程执行，_AfxThreadEntry 由被创建的线程执行，两者通过两个事件对象(hEvent 和 hEvent2)同步：在创建了新线程之后，创建线程将在 hEvent 事件上无限等待直到新线程给出创建结果；新线程在创建成功或者失败之后，触发事件 hEvent 让父线程运行，并且在 hEven2 上无限等待直到父线程退出 CreateThread 函数；父线程(创建线程)因为 hEvent 的置位结束等待，继续执行，退出 CreateThread 之前触发 hEvent2 事件；新线程（子线程）因为 hEvent2 的置位结束等待，开始执行控制函数（工作者线程）或者进入消息循环（用户界面线程）。

MFC 在线程创建中使用了如下数据结构：

```
struct _AFX_THREAD_STARTUP
{
//传递给线程启动的参数（IN）
```

```
_AFX_THREAD_STATE* pThreadState;//父线程的线程状态
CWinThread* pThread;      //新创建的 MFC 线程对象
DWORD dwCreateFlags;      //线程创建标识
_PNH pfnNewHandler;       //新线程的句柄
HANDLE hEvent;      //同步事件，线程创建成功或失败后置位
HANDLE hEvent2;     //同步事件，新线程恢复执行后置位

//返回给创建线程的参数，在新线程恢复执行后赋值
BOOL bError;      //如果创建发生错误，TRUE
};
```

该结构作为线程开始函数的参数被传递给_beginthreadex 函数来创建和启动线程。_beginthreadex 函数是“C”的线程创建函数，具有如下原型：

```
unsigned long _beginthreadex(
void *security,
unsigned stack_size,
unsigned ( __stdcall *start_address )( void * ),
void *arglist,
unsigned initflag,
unsigned *thrdaddr );
```

图 8-2 描述了上述过程。图中表示，_AfxThreadEntry 在启动线程时，将创建本线程的线程状态，并且继承父线程的模块状态。关于 MFC 状态，见第 9 章。

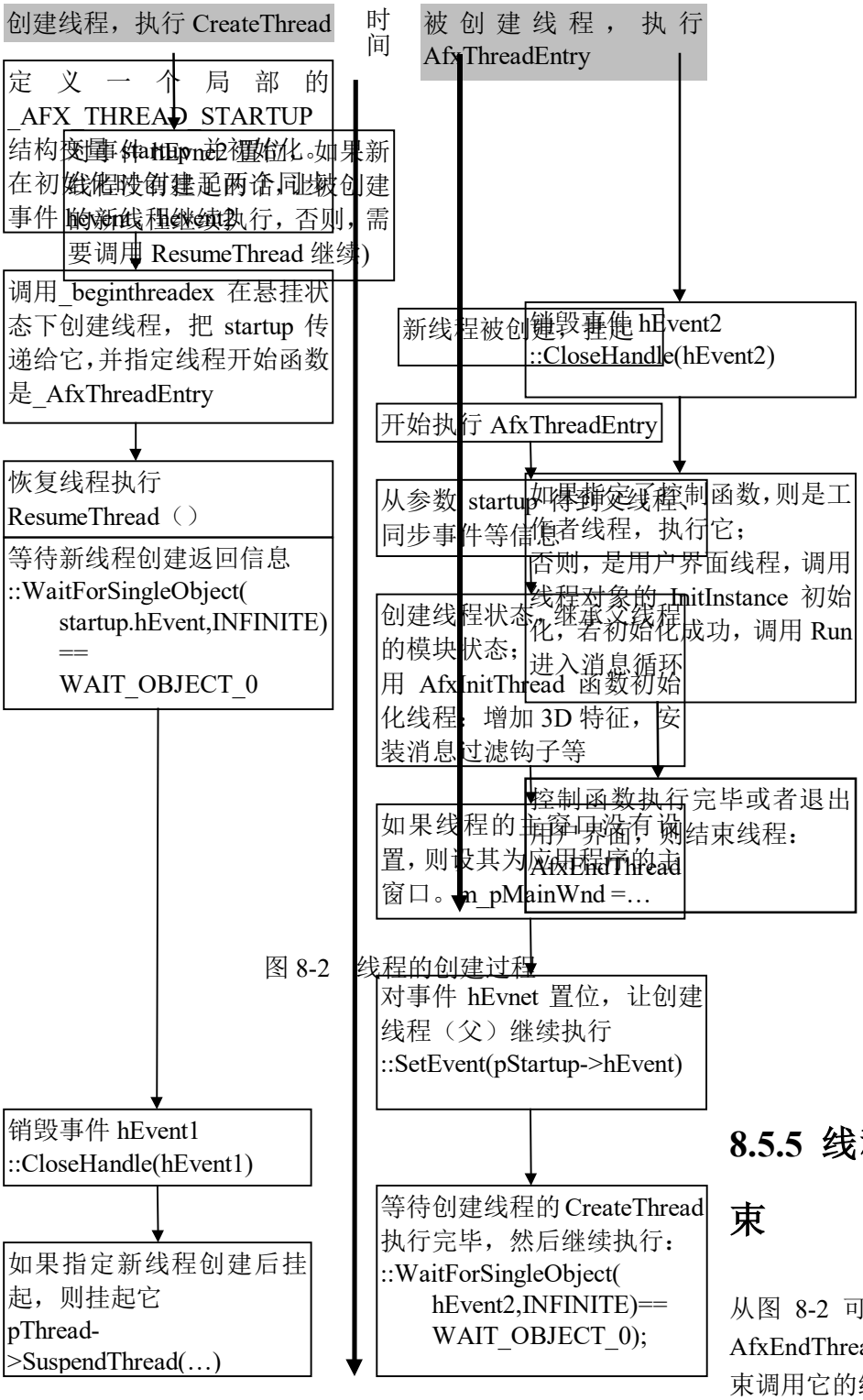


图 8-2　线程的创建过程

8.5.5 线程的结束

从图 8-2 可以看出，AfxEndThread 用来结束调用它的线程：它将清理本线程创建的 MFC 对象和释放线程局部存储分配的内存空间；调用 CWinThread 的虚拟函数 Delete；调用“C”的结束线程函数_endthreadex 释放分配给线程的资源，但是不关闭线程句柄。

CWinThread::Delete 的缺省实现是：如果本线程的成员函数 m_bDelete 为 TRUE，则调用“C”运算符号 delete 销毁 MFC 线程对象自身（delete this），这将导致线程对象的析构函数被调用。若析构函数检测线程句柄非空则调用 CloseHandle 关闭它。

通常，让 m_bDelete 为 TRUE 以便自动地销毁线程对象，释放内存空间（MFC 内存对象在

堆中分配）。但是，有时候，在线程结束之后（Win32 线程已经不存在）保留 MFC 线程对象是有用的，当然程序员自己最后要记得销毁该线程对象。

8.5.6 实现线程的消息循环

在 MFC 中，消息循环是由线程完成的。一般地，可以使用 MFC 缺省的消息循环（即使用函数 CWindThrad::Run），但是，有些时候需要程序员自己实现一个线程的消息循环，比如在用户界面线程进行一个长时间计算处理或者等待另一个线程时。一般有如下形式：

```
while ( bDoingBackgroundProcessing)
{
MSG msg;
while ( ::PeekMessage( &msg, NULL,0, 0, PM_NOREMOVE ) )
{
    if ( !PumpMessage( ) )
    {
        bDoingBackgroundProcessing = FALSE;
        ::PostQuitMessage( );
        break;
    }
}
// let MFC do its idle processing
LONG lIdle = 0;
while ( AfxGetApp()->OnIdle(lIdle++ ) );
// Perform some background processing here
// using another call to OnIdle
}
```

该段代码的解释参见图 5-3 对线程的 Run 函数的图解。

程序员实现线程的消息循环有两个好处，一是顾及了 MFC 的 Idle 处理机制；二是在长时间的处理中可以响应用户产生的事件或者消息。

在同步对象上等待其他线程时，也可以使用同样的方式，只要把条件

```
bDoingBackgroundProcessing
```

换成如下形式：

```
WaitForSingObject(hHandleOfEvent,0) == WAIT_TIMEOUT
```

即可。

MFC 处理线程和进程时还引入了一个重要的概念：状态，如线程状态(Thread State)、进程状态(Process State)、模块状态(Module State)等。由于这个概念在 MFC 中占有重要地位，涉及的内容比较多，所以专门在下一章来讲述它。

第9章　MFC 的状态

MFC 定义了多种状态信息，这里要介绍的是模块状态、进程状态、线程状态。这些状态可以组合在一起，例如 MFC 句柄映射就是模块和线程局部有效的，属于模块-线程状态的一部分。

9.1　模块状态

这里模块的含义是：一个可执行的程序或者一个使用 MFC DLL 的 DLL，比如一个 OLE 控件就是一个模块。

一个应用程序的每一个模块都有一个状态，模块状态包括这样一些信息：用来加载资源的 Windows 实例句柄、指向当前 CWinApp 或者 CWinThread 对象的指针、OLE 模块的引用计数、Windows 对象与相应的 MFC 对象之间的映射。只有单一模块的应用程序的状态如图 9-1 所示。

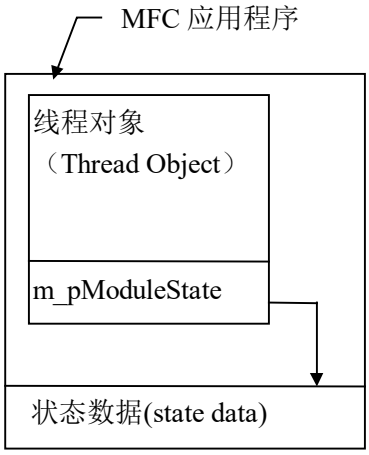


图 9-1 单模块状态示意图

m_pModuleState 指针是线程对象的成员变量，指向当前模块状态信息（一个 AFX_MODULE_STATE 结构变量）。当程序运行进入某个特定的模块时，必须保证当前使用的模块状态是有效的模块状态——是这个特定模块的模块状态。所以，每个线程对象都有一个指针指向有效的模块状态，每当进入某个模块时都要使它指向有效模块状态，这对维护应用程序全局状态和每个模块状态的完整性来说是非常重要的。为了作到这一点，每个模块的所有入口点有责任实现模块状态的切换。模块的入口点包括：DLL 的输出函数；OLE/COM 界面的成员函数；窗口过程。

在讲述窗口过程和动态链接到 MFC DLL 的规则 DLL 时，曾提到了语句 AFX_MANAGE_STATE(AfxGetStaticModuleState( ))，它就是用来在入口点切换模块状态的。其实现机制将在后面 9.4.1 节讲解。

多个模块状态之间切换的示意图如图 9-2 所示。

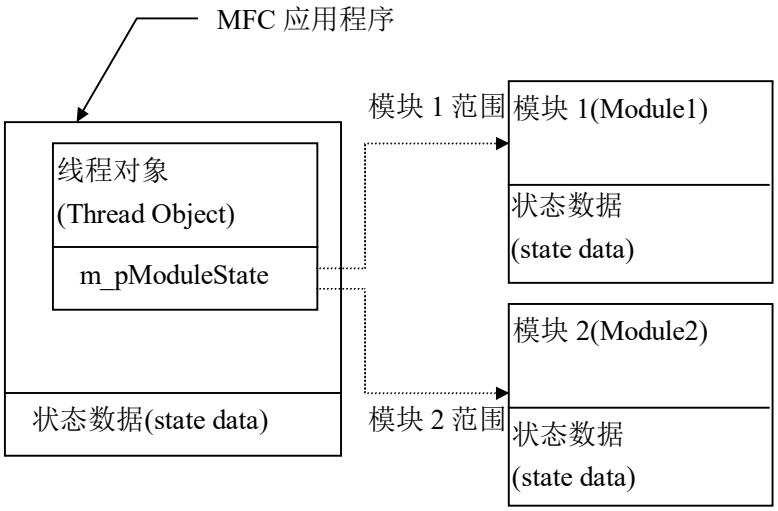


图 9-2 多模块的状态示意图

图 9-2 中，m_pModuleState 总是指向当前模块的状态。

9.2 模块、进程和线程状态的数据结构

MFC 定义了一系列类或者结构，通过它们来实现状态信息的管理。这一节将描述它们的关系，并逐一解释它们的数据结构、成员函数等。

9.2.1 层次关系

图 9-3 显示了线程状态、模块状态、线程-模块状态等几个类的层次关系：

线程状态用类_AFX_THREAD_STATE 描述，模块状态用类 AFX_MODULE_STATE 描述，模块-线程状态用类 AFX_MODULE_THREAD_STATE 描述。这些类从类 CNoTrackObject 派生。进程状态类用 _AFX_BASE_MODULE_STATE 描述，从模块状态类 AFX_MODULE_STATE 派生。进程状态是了一个可以独立执行的 MFC 应用程序的模块状态。还有其他状态如 DLL 的模块状态等也从模块状态类_AFX_MODULE_STATE 派生。

图 9-4 显示了这几个类的交互关系。

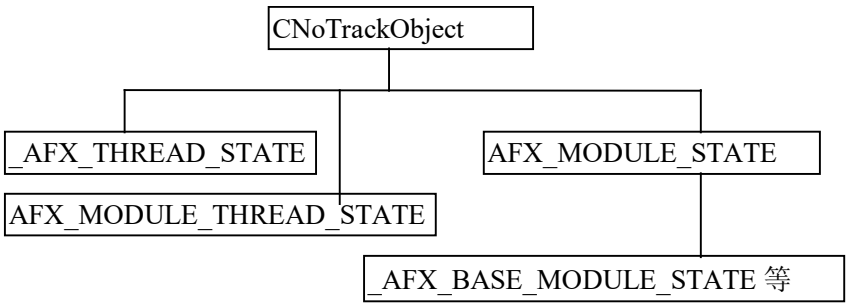


图 9-3 MFC 状态类的层次

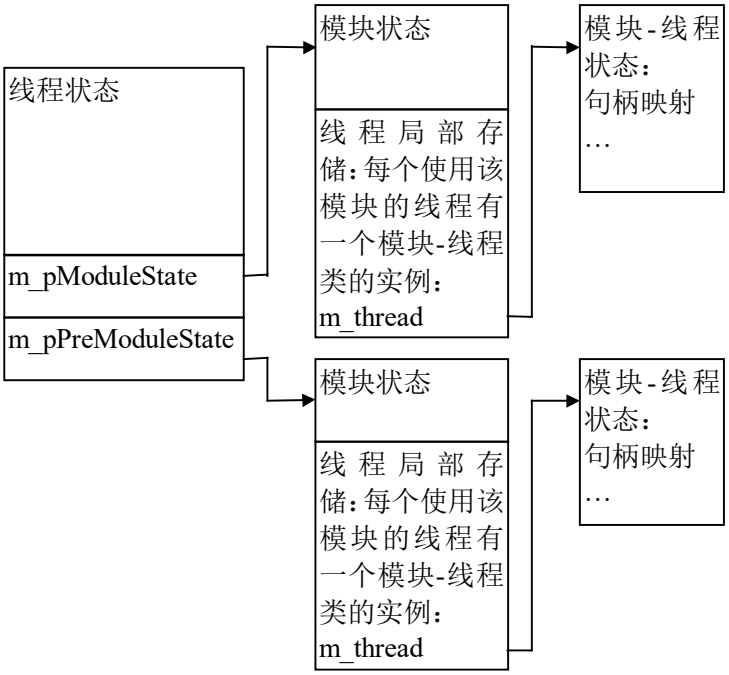


图 9-4　模块、线程、模块-线程状态的关系

从图 9-4 可以看出：首先，每个线程有一个线程状态，线程状态的指针 m_pModuleState 和 m_pPreModuleState 分别指向线程当前运行模块的状态或前一运行模块的状态；其次，每一个模块状态都有一个线程局部的变量用来存储模块-线程状态。
下面各小节列出状态信息管理所涉及的各个类的定义。

9.2.2 CNoTrackObject 类

在图 9-3 中， CnoTrackObject 是根类，所有状态类都是从它这里派生的，其定义如下：

```
class CNoTrackObject
{
public:
void* PASCAL operator new(size_t nSize);
void PASCAL operator delete(void*);

#if defined(_DEBUG) && !defined(_AFX_NO_DEBUG_CRT)
    void* PASCAL operator new(size_t nSize, LPCSTR, int);
#endif
virtual ~CNoTrackObject() { }
};
```

该类的析构函数是虚拟函数；而且，CNoTrackObject重载new操作符用来分配内存，重载delete操作符号用来释放内存，内部通过LocalAlloc/LocalFree提供了一个低层内存分配器（Low_level alloctor)。

9.2.3 AFX_MODULE_STATE 类

AFX_MODULE_STATE 类的定义如下：

```
// AFX_MODULE_STATE (global data for a module)
class AFX_MODULE_STATE : public CNoTrackObject
{
public:
#ifdef _AFXDLL
    AFX_MODULE_STATE(BOOL bDLL,WNDPROC pfnAfxWndProc,
    DWORD dwVersion);
    AFX_MODULE_STATE(BOOL bDLL, WNDPROC pfnAfxWndProc,
    DWORD dwVersion,BOOL bSystem);
#else
    AFX_MODULE_STATE(BOOL bDLL);
#endif
~AFX_MODULE_STATE();

CWinApp* m_pCurrentWinApp;
HINSTANCE m_hCurrentInstanceHandle;
HINSTANCE m_hCurrentResourceHandle;
LPCTSTR m_lpszCurrentAppName;
BYTE m_bDLL;// TRUE if module is a DLL, FALSE if it is an EXE
//TRUE if module is a "system" module, FALSE if not
BYTE m_bSystem;
BYTE m_bReserved[2]; // padding

//Runtime class data:
#ifdef _AFXDLL
    CRuntimeClass* m_pClassInit;
#endif
CTypedSimpleList<CRuntimeClass*> m_classList;

// OLE object factories
#ifndef _AFX_NO_OLE_SUPPORT
    #ifdef _AFXDLL
        COleObjectFactory* m_pFactoryInit;
    #endif
    CTypedSimpleList<COleObjectFactory*> m_factoryList;
#endif

// number of locked OLE objects
long m_nObjectCount;
BOOL m_bUserCtrl;
```

```
    // AfxRegisterClass and AfxRegisterWndClass data
    TCHAR m_szUnregisterList[4096];
    #ifdef _AFXDLL
        WNDPROC m_pfnAfxWndProc;
        DWORD m_dwVersion;    // version that module linked against
    #endif

    // variables related to a given process in a module
    //    (used to be AFX_MODULE_PROCESS_STATE)
    #ifdef _AFX_OLD_EXCEPTIONS
    // exceptions
        AFX_TERM_PROC m_pfnTerminate;
    #endif
    void (PASCAL *m_pfnFilterToolTipMessage)(MSG*, CWnd*);

    #ifdef _AFXDLL
    // CDynLinkLibrary objects (for resource chain)
        CTypedSimpleList<CDynLinkLibrary*> m_libraryList;

        // special case for MFCxxLOC.DLL (localized MFC resources)
        HINSTANCE m_appLangDLL;
    #endif

    #ifndef _AFX_NO_OCC_SUPPORT
        // OLE control container manager
        COccManager* m_pOccManager;
        // locked OLE controls
        CTypedSimpleList<COleControlLock*> m_lockList;
    #endif

    #ifndef _AFX_NO_DAO_SUPPORT
        _AFX_DAO_STATE* m_pDaoState;
    #endif

    #ifndef _AFX_NO_OLE_SUPPORT
        // Type library caches
        CTypeLibCache m_typeLibCache;
        CMapPtrToPtr* m_pTypeLibCacheMap;
    #endif

    // define thread local portions of module state
    THREAD_LOCAL(AFX_MODULE_THREAD_STATE, m_thread)
    };
```

从上面的定义可以看出，模块状态信息分为如下几类：

模块信息，资源信息，对动态链接到 MFC DLL 的支持信息，对扩展 DLL 的支持信息，对 DAO 的支持信息，对 OLE 的支持信息，模块-线程状态信息。
模块信息包括实例句柄、资源句柄、应用程序名称、指向应用程序的指针、是否为 DLL 模块、模块注册的窗口类，等等。其中，成员变量 m_fRegisteredClasses、m_szUnregisterList 曾经在讨论 MFC 的窗口注册时提到过它们的用处。
在“#ifdef _AFXDLL…#endif”条件编译范围内的是支持 MFC DLL 的数据；
在“#ifndef _AFX_NO_OLE_SUPPOR…#endif”条件编译范围内的是支持 OLE 的数据；
在“#ifndef _AFX_NO_OCC_SUPPOR…#endif”条件编译范围内的是支持 OLE 控件的数据；
在“#ifndef _AFX_NO_DAO_SUPPORT”条件编译范围内的是支持 DAO 的数据。
THREAD_LOCAL 宏定义了线程私有的模块-线程类型的变量 m_thread。

9.2.4 _AFX_BASE_MODULE_STATE

该类定义如下：

```
class _AFX_BASE_MODULE_STATE : public AFX_MODULE_STATE
{
public:
#ifdef _AFXDLL
        _AFX_BASE_MODULE_STATE() : AFX_MODULE_STATE(TRUE,
          AfxWndProcBase, _MFC_VER)
#else
        _AFX_BASE_MODULE_STATE() : AFX_MODULE_STATE(TRUE)
#endif
{ }
};
```

由定义可见，该类没有在_AFX_MODULE_STATE 类的基础上增加数据。它类用来实现一个 MFC 应用程序模块的状态信息。

9.2.5 _AFX_THREAD_STATE

该类定义如下：

```
class _AFX_THREAD_STATE : public CNoTrackObject
{
public:
_AFX_THREAD_STATE();
virtual ~_AFX_THREAD_STATE();

// override for m_pModuleState in _AFX_APP_STATE
AFX_MODULE_STATE* m_pModuleState;
AFX_MODULE_STATE* m_pPrevModuleState;

// memory safety pool for temp maps
void* m_pSafetyPoolBuffer;      // current buffer
```

```
// thread local exception context
AFX_EXCEPTION_CONTEXT m_exceptionContext;

// CWnd create, gray dialog hook, and other hook data
CWnd* m_pWndInit;
CWnd* m_pAlternateWndInit;          // special case commdlg hooking
DWORD m_dwPropStyle;
DWORD m_dwPropExStyle;
HWND m_hWndInit;
BOOL m_bDlgCreate;
HHOOK m_hHookOldCbtFilter;
HHOOK m_hHookOldMsgFilter;

// other CWnd modal data
MSG m_lastSentMsg;                  // see CWnd::WindowProc
HWND m_hTrackingWindow;         // see CWnd::TrackPopupMenu
HMENU m_hTrackingMenu;
TCHAR m_szTempClassName[96];       // see AfxRegisterWndClass
HWND m_hLockoutNotifyWindow;       // see CWnd::OnCommand
BOOL m_bInMsgFilter;

// other framework modal data
CView* m_pRoutingView;         // see CCmdTarget::GetRoutingView
CFrameWnd*m_pRoutingFrame;//see CmdTarget::GetRoutingFrame

// MFC/DB thread-local data
BOOL m_bWaitForDataSource;

// common controls thread state
CToolTipCtrl* m_pToolTip;
CWnd* m_pLastHit;             // last window to own tooltip
int m_nLastHit;                // last hittest code
TOOLINFO m_lastInfo;       // last TOOLINFO structure
int m_nLastStatus;           // last flyby status message
CControlBar* m_pLastStatus; // last flyby status control bar

// OLE control thread-local data
CWnd* m_pWndPark;             // "parking space" window
long m_nCtrlRef;              // reference count on parking window
BOOL m_bNeedTerm;             // TRUE if OleUninitialize needs to be called
};
```

从定义可以看出，线程状态的成员数据分如下几类：

指向模块状态信息的指针，支持本线程的窗口创建的变量，MFC 命令和消息处理用到的信

息，处理工具条提示信息(tooltip)的结构，和处理 OLE 相关的变量，等等。

9.2.6 AFX_MODULE_THREAD_STATE

该类定义如下：

```
// AFX_MODULE_THREAD_STATE (local to thread *and* module)
class AFX_MODULE_THREAD_STATE : public CNoTrackObject
{
public:
AFX_MODULE_THREAD_STATE();
virtual ~AFX_MODULE_THREAD_STATE();

// current CWinThread pointer
CWinThread* m_pCurrentWinThread;

// list of CFrameWnd objects for thread
CTypedSimpleList<CFrameWnd*> m_frameList;

// temporary/permanent map state
DWORD m_nTempMapLock;          // if not 0, temp maps locked
CHandleMap* m_pmapHWND;
CHandleMap* m_pmapHMENU;
CHandleMap* m_pmapHDC;
CHandleMap* m_pmapHGDIOBJ;
CHandleMap* m_pmapHIMAGELIST;

// thread-local MFC new handler (separate from C-runtime)
_PNH m_pfnNewHandler;

#ifndef _AFX_NO_SOCKET_SUPPORT
    // WinSock specific thread state
    HWND m_hSocketWindow;
    CMapPtrToPtr m_mapSocketHandle;
    CMapPtrToPtr m_mapDeadSockets;
    CPtrList m_listSocketNotifications;
#endif
};
```

模块-线程状态的数据成员主要有：

指向当前线程对象(CWinThread 对象)的指针 m_pCurrentWinThread；

当前线程的框架窗口对象(CFrameWnd 对象)列表 m_frameList（边框窗口在创建时（见图 5-8）把自身添加到 m-frameList 中，销毁时则删除掉，通过列表 m_frameList 可以遍历模块所有的边框窗口）；

new 操作的例外处理函数 m_pfnNewHandler；

临时映射锁定标识 m_nTempMapLock，防止并发修改临时映射。

系列 Windows 对象-MFC 对象的映射，如 m_pmapHWND 等。
这些数据成员都是线程和模块私有的。

下一节讨论 MFC 如何通过上述这些类来实现其状态的管理。

9.3 线程局部存储机制和状态的实现

MFC 实现线程、模块或者线程-模块私有状态的基础是 MFC 的线程局部存储机制。MFC 定义了 CThreadSlotData 类型的全局变量_afxThreadData 来为进程的线程分配线程局部存储空间：
CThreadSlotData* _afxThreadData;
在此基础上，MFC 定义了变量_afxThreadState 来管理线程状态，定义了变量_afxBaseModuleState 来管理进程状态。
THREAD_LOCAL(_AFX_THREAD_STATE, _afxThreadState)
PROCESS_LOCAL(_AFX_BASE_MODULE_STATE, _afxBaseModuleState)
对于每个THREAD_LOCAL宏定义的变量，进程的每个线程都有自己独立的拷贝，这个变量在不同的线程里头可以有不同的取值。
对于每个PROCESS_LOCAL宏定义的变量，每个进程都有自己独立的拷贝，这个变量在不同的进程里头可以有不同的取值。
分别解释这三个变量。

9.3.1 CThreadSlotData 和_afxThreadData

9.3.1.1 CThreadSlotData 的定义

以 Win32 线程局部存储机制为基础，MFC 设计了类 CThreadSlotData 来提供管理线程局部存储的功能，MFC 应用程序使用该类的对象——全局变量_afxThreadData 来管理本进程的线程局部存储。CThreadSlotData 类的定义如下：

```
class CThreadSlotData
{
public:
CThreadSlotData();

//Operations
int AllocSlot();
void FreeSlot(int nSlot);
void* GetValue(int nSlot);
void SetValue(int nSlot, void* pValue);
// delete all values in process/thread
void DeleteValues(HINSTANCE hInst, BOOL bAll = FALSE);
// assign instance handle to just constructed slots
```

```
void AssignInstance(HINSTANCE hInst);

// Implementation
DWORD m_tlsIndex;// used to access system thread-local storage
int m_nAlloc;     // number of slots allocated (in UINTs)
int m_nRover; // (optimization) for quick finding of free slots
int m_nMax;    // size of slot table below (in bits)
CSlotData* m_pSlotData; // state of each slot (allocated or not)
//list of CThreadData structures
CTypedSimpleList<CThreadData*> m_list;
CRITICAL_SECTION m_sect;
// special version for threads only!
void* GetThreadValue(int nSlot);
void* PASCAL operator new(size_t, void* p){ return p; }
void DeleteValues(CThreadData* pData, HINSTANCE hInst);
~CThreadSlotData();
};
```

通过 TLS 索引 m_tlsIndex，CThreadSlotData 对象（_afxThreadData）为每一个线程分配一个线程私有的存储空间并管理该空间。它把这个空间划分为若干个槽，每个槽放一个线程私有的数据指针，这样每个线程就可以存放任意个线程私有的数据指针。

9.3.1.2 CThreadSlotData 的一些数据成员

在 CThreadSlotData 类的定义中所涉及的类或者结构定义如下：

（1）m_sect

m_sect 是一个关键段变量，在_afxThreadData 创建时初始化。因为_afxThreadData 是一个全局变量，所以必须通过 m_sect 来同步多个线程对该变量的并发访问。

（2）m_nAlloc 和 m_pSlotData

m_nAlloc 表示已经分配槽的数目，它代表了线程局部变量的个数。每一个线程局部变量都对应一个槽，每个槽对应一个线程局部变量。槽使用 CSlotData 类来管理。

CSlotData 的定义如下：

```
struct CSlotData{
DWORD dwFlags;          // slot flags (allocated/not allocated)
HINSTANCE hInst;       // module which owns this slot
};
```

该结构用来描述槽的使用：

域 dwFlags 表示槽的状态，即被占用或者没有；

域 hInst 表示使用该槽的模块的句柄。

m_pSlotData 表示一个 CSlotData 类型的数组，用来描述各个槽。该数组通过成员函数 AllocSlot 和 FreeSlot 来动态地管理，见图 9-6。

（3）m_list

先讨论 CThreadData 类。CThreadData 定义如下：

```
struct CThreadData : public CNoTrackObject{
CThreadData* pNext; // required to be member of CSimpleList
```

```
int nCount;              // current size of pData
LPVOID* pData;           // actual thread local data (indexed by nSlot)
};
```

该结构用来描述 CThreadSlotData 为每个线程管理的线程局部空间：

域 pNext 把各个线程的 CThreadData 项目链接成一个表，即把各个线程的线程私有空间链接起来；

域 nCount 表示域 pData 的尺寸，即存储了多少个线程私有数据；

pData 表示一个 LPVOID 类型的数组，数组中的每一个元素保存一个指针，即线程私有数据指针，该指针指向一个在堆中分配的真正存储线程私有数据的地址。数组元素的个数和槽的个数相同，每个线程局部变量（THREAD_LOCAL 定义的变量）都有一个对应的槽号，用该槽号作为下标来引用 pData。

m_list 表示一个 CThreadData 类型的指针数组，数组中的各项指向各个线程的线程私有空间，每个线程在数组中都有一个对应项。该数组通过 GetValue、SetValue、DeleteValues 等成员函数来管理，见图 9-6。

9.3.1.3 _afxThreadData

_afxThreadData 仅仅定义为一个 CThreadSlotData 类型的指针，所指对象在第一次被引用时创建，在此之前该指针为空。下文_afxThreadData 含义是它所指的对象。图 9-5、9-6 图解了 MFC 的线程局部存储机制的实现。

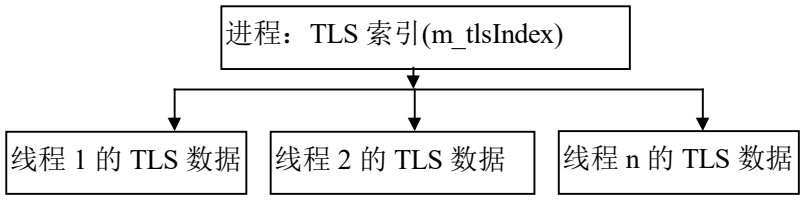


图 9-5　MFC 的线程存储机制 1

图 9-5 表示_afxTheadData 使用 TLS 技术负责给进程分配一个 TLS 索引，然后使用 TLS 索引为进程的每一个线程分配线程局部存储空间。

图 9-6 表示每个线程的的局部存储空间可以分多个槽，每个槽可以放一个线程私有的数据指针。_afxThreadData 负责给线程局部变量分配槽号并根据槽号存取数据。图的左半部分描述了管理槽的 m_pSlotData 及类 CSlotData 的结构，右半部分描述了管理 MFC 线程私有空间的 m_list 及类 CThreadData 的结构。

结合图 9-6，对 MFC 线程局部存储机制总结如下：

- 每个线程局部变量（宏 THREAD_LOCAL 定义）占用一个槽，并有一个槽号。。
- 每个线程都有自己的 MFC 局部存储空间（下文多次使用“线程的 MFC 局部存储空间”，表示和此处相同的概念）。
- 通过 TLS 索引得到的是一个指针 P1，它指向线程的 MFC 局部存储空间。
- 通过指针 P1 和线程局部变量在空间所占用的槽号，得到该槽所存储的线程私有的数据指针，即真正的线程私有数据的地址 P2；
- 从地址 P2 得到数据 D。

这个过程相当于几重间接寻址：先得到 TLS 线程私有数据指针，从 TLS 线程私有数据指针

得到线程的 MFC 线程局部存储空间，再从 MFC 局部存储空间的对应槽得到一个线程私有的数据指针，从该指针得到最终的线程私有数据。如果没有这种机制，使用 Win32 TLS 只要一次间接寻址：得到 TLS 线程私有数据指针，从该指针得到最终的线程私有数据。

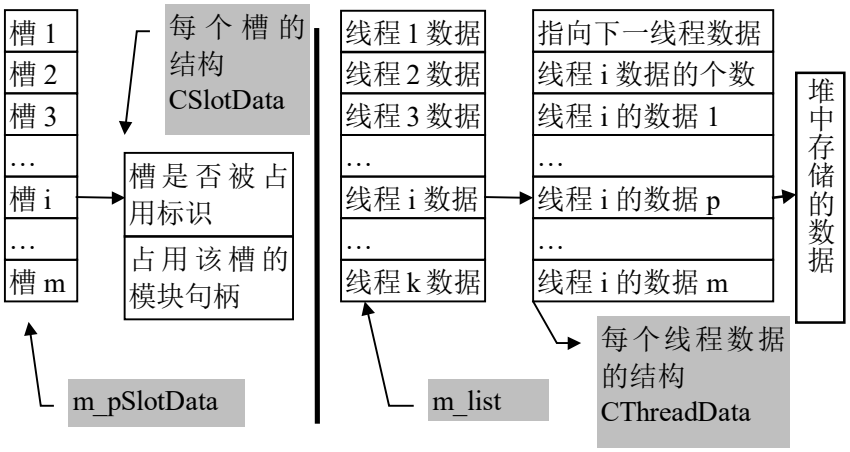


图 9-6　MFC 的线程存储机制 2

9.3.2 线程状态_afxThreadState

从上一节知道了 MFC 的线程局部存储机制。但有一点还不清楚，即某个线程局部变量所占用的槽号是怎么保存的呢？关于这点可从线程局部的线程状态变量_afxThreadState 的实现来分析 MFC 的作法。变量_afxThreadState 的定义如下：

```
THREAD_LOCAL(_AFX_THREAD_STATE, _afxThreadState)
```

THREAD_LOCAL 是一个宏，THREAD_LOCAL(class_name, ident_name)宏展开后如下：

```
AFX_DATADEF CThreadLocal<class_name> ident_name;
```

这里，CThreadLocal 是一个类模板，从 CThreadLocalObject 类继承。

CThreadLocalObject 和 CThreadLocal 的定义如下：

```
class CThreadLocalObject
{
public:
// Attributes
CNoTrackObject* GetData(CNoTrackObject* (AFXAPI*
pfnCreateObject)());
CNoTrackObject* GetDataNA();

// Implementation
int m_nSlot;
~CThreadLocalObject();
};
```

CThreadLocalObject 用来帮助实现一个线程局部的变量。成员变量 m_nSlot 表示线程局部变量在 MFC 线程局部存储空间中占据的槽号。GetDataNA 用来返回变量的值。GetData 也可以返回变量的值，但是如果发现还没有给该变量分配槽号(m_slot=0)，则给它分配槽号并在线程的 MFC 局部空间为之分配一个槽；如果在槽 m_nSlot 还没有数据（为空），则调用参数 pfnCreateObject 传递的函数创建一个数据项，并保存到槽 m_nSlot 中。

```
template<class TYPE>
class CThreadLocal : public CThreadLocalObject
{
// Attributes
public:
inline TYPE* GetData()
{
        TYPE* pData = (TYPE*)CThreadLocalObject::GetData(&CreateObject);
        ASSERT(pData != NULL);
        return pData;
}
inline TYPE* GetDataNA()
{
        TYPE* pData = (TYPE*)CThreadLocalObject::GetDataNA();
        return pData;
}
inline operator TYPE*()
{ return GetData(); }
inline TYPE* operator->()
{ return GetData(); }

// Implementation
public:
static CNoTrackObject* AFXAPI CreateObject()
{ return new TYPE; }
};
```

CThreadLocal 模板用来声明任意类型的线程私有的变量，因为通过模板可以自动的正确的转化(cast)指针类型。程序员可以使用它来实现自己的线程局部变量，正如 MFC 实现线程局部的线程状态变量和模块-线程变量一样。

CThrealLocal 的成员函数 CreateObject 用来创建动态的指定类型的对象。成员函数 GetData 调用了基类 CThreadLocalObject 的同名函数，并且把 CreateObject 函数的地址作为参数传递给它。

另外，CThreadLocal 模板重载了操作符号“*”、“->”，这样编译器将自动地进行有关类型转换，例如:

_AFX_THREAD_STATE *pStata = _afxThreadState

是可以被编译器接收的。

现在回头来看_afxThreadState 的定义：

从以上分析可以知道，THREAD_LOCAL(class_name, ident_name)定义的结果并没有产生一个名为 ident_name 的 class_name 类的实例，而是产生一个 CThreadLocal 模板类（确切地说，是其派生类）的实例，m_nSlot 初始化为 0。所以，_afxThreadState 实质上是一个 CThreadLocal 模板类的全局变量。每一个线程局部变量都对应了一个全局的 CThreadLoacl 模板类对象，模板对象的 m_nSlot 记录了线程局部变量对象的槽号。

9.3.3 进程模块状态 afxBaseModuleState

进程模块状态定义如下：

```
PROCESS_LOCAL(_AFX_BASE_MODULE_STATE, _afxBaseModuleState)
```

表示它是一个_AFX_BASE_MODULE_STATE 类型的进程局部(process local)的变量。
进程局部变量的实现方法主要是为了用于 Win32s 下。在 Win32s 下，一个 DLL 模块如果被多个应用程序调用，它将让这些程序共享它的全局数据。为了 DLL 的全局数据一个进程有一份独立的拷贝，MFC 设计了进程私有的实现方法，实际上就是在进程的堆(Heap)中分配全局数据的内存空间。
在 Win32 下，DLL 模块的数据和代码被映射到调用进程的虚拟空间，也就是说，DLL 定义的全局变量是进程私有的；所以进程局部变量的实现并不为 Win32 所关心。但是，不是说 afxBaseModuleState 不重要，仅仅是采用 PROCESS_LOCAL 技术声明它是进程局部变量不是很必要了。PROCESS_LOCAL(class_name, ident_name)宏展开后如下：

```
AFX_DATADEF CProcessLocal<class_name> ident_name;
```

这里，CProcessLocal 是一个类模板，从 CProcessLocalObject 类继承。
CProcessLocalObject 和 CProcessLocal 的定义如下：

```
class CProcessLocalObject
{
public:
// Attributes
CNoTrackObject* GetData(CNoTrackObject* (AFXAPI*
pfnCreateObject)());

// Implementation
CNoTrackObject* volatile m_pObject;
~CProcessLocalObject();
};

template<class TYPE>
class CProcessLocal : public CProcessLocalObject
{
// Attributes
public:
inline TYPE* GetData()
{
    TYPE* pData =(TYPE*)CProcessLocalObject::GetData(&CreateObject);
    ASSERT(pData != NULL);
    return pData;
```

```
}
inline TYPE* GetDataNA()
{ return (TYPE*)m_pObject; }
inline operator TYPE*()
{ return GetData(); }
inline TYPE* operator->()
{ return GetData(); }

// Implementation
public:
static CNoTrackObject* AFXAPI CreateObject()
{ return new TYPE; }
};
```

类似于线程局部对象，每一个进程局部变量都有一个对应的全局 CProcessLocal 模板对象。

9.3.4 状态对象的创建

9.3.4.1 状态对象的创建过程

回顾前一节的三个定义：

```
CThreadSlotData* _afxThreadData;
THREAD_LOCAL(_AFX_THREAD_STATE, _afxThreadState)
PROCESS_LOCAL(_AFX_BASE_MODULE_STATE, _afxBaseModuleState)
```

第一个仅仅定义了一个指针；第二和第三个定义了一个模板类的实例。相应的CThreadSlotData对象（全局）、_AFX_THREAD_STATE对象（线程局部）以及_AFX_BASE_MODULE_STATE对象（进程局部）并没有创建。当然，模块状态对象的成员模块-线程对象也没有被创建。这些对象要到第一次被访问时，才会被创建，这样做会提高加载DLL的速度。

下面以一个动态链接到MFC DLL的单模块应用程序为例，说明这些对象的创建过程。

当第一次访问状态信息时，比如使用AfxGetModuleState得到模块状态，导致系列创建过程的开始，如图9-7所示。

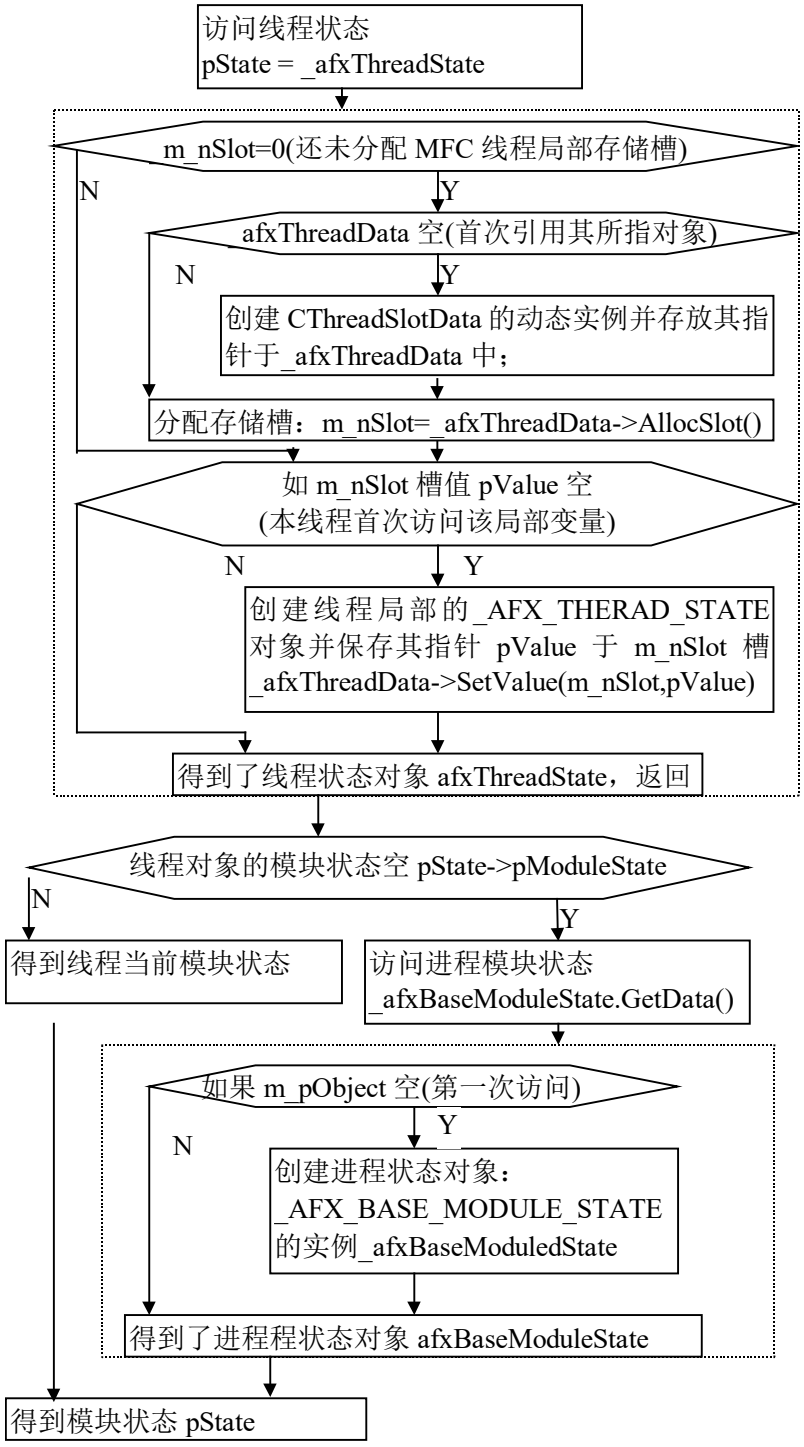


图 9-7　线程、模块状态的创建过程

首先分析语句 pState=_afxThreadState 。如果_afxThreadData、线程状态和模块状态还没有创建，该语句可以导致这些数据的创建。

pState 声明为 CNoTrackObject 对象的指针，_afxThreadState 声明为一个模板 CThreadLocal 的实例，pState=_afxThreadData 为什么可以通过编译器的检查呢？因为 CThreadLocal 模板重载了操作符“”*”和“->”，这两个运算返回 CNoTrackObject 类型的对象。回顾 3.2 节 CThreadLocalObject 、CThreadLocal 的定义，这两个操作符运算到最后都是调用 CThreadLocalObject 的成员函数 GetData。

- 创建_afxThreadData 所指对象和线程状态

CThreadLocalObject::GetData 用来获取线程局部变量（这个例子中是线程状态）的值，其参数用来创建动态的线程局部变量。图 9-7 的上面的虚线框表示其流程：

它检查成员变量 m_nSlot 是否等于 0（线程局部变量是否曾经被分配了 MFC 线程私有空间槽位），检查全局变量_afxTheadData 指针是否为空。如果_afxThreadData 空，则创建一个 CThreadSlotData 类对象，让_afxThreadData 指向它，这样本程序的 MFC 线程局部存储的管理者被创建。如果 m_nSlot 等于 0，则让_afxThreadDtata 调用 AllocSlot 分配一个槽位并把槽号保存在 m_nSlot 中。

得到了线程局部变量（线程状态）所占用的槽位后，委托_afxThreadData 调用 GetThreadValue(m_nSlot)得到线程状态值（指针）。如果结果非空，则返回它；如果结果是 NULL，则表明该线程状态还没有被创建，于是使用参数创建一个动态的线程状态，并使用

SetValue 把其指针保存在槽 m_nSlot 中，返回该指针。

- 创建模块状态

得到了线程状态的值后，通过它得到模块状态 m_pModuleState。如果 m_pModuleState 为空，表明该线程状态是才创建的，其许多成员变量还没有赋值，程序的进程模块状态还没有被创建。于是调用函数_afxBaseModule.GetData，导致进程模块状态被创建。
图 9-7 的下面一个虚线框表示了 CProcessLocalObject::GetData 的创建过程：
_afxBaseModule 首先检查成员变量 m_pObject 是否空，如果非空就返回它，即进程模块状态指针；否则，在堆中创建一个动态的_AFX_BASE_MODULE_STATE 对象，返回。
从上述两个 GetData 的实现可以看出，CThreadLocal 模板对象负责线程局部变量的创建和管理(查询，修改，删除)；CProcessLocal 模板对象负责进程局部变量的创建和管理(查询，修改，删除)。

- 模块-线程状态的创建

模块状态的成员模块-线程状态 m_thread 的创建类似于线程状态的创建：当第一次访问 m_thread 所对应的 CThreadLocal 模板对象时，给 m_thread 分配 MFC 线程局部存储的私有槽号 m_nSlot，并动态地创建_AFX_MODULE_THREAD_STATE 对象，保存对象指针在 m_nSlot 槽中。

9.3.4.2 创建过程所涉及的几个重要函数的算法

创建过程所涉及的几个重要函数的算法描述如下：

（1） AllocSlot

AllocSlot 用来分配线程的 MFC 私有存储空间的槽号。由于该函数要修改全局变量_afxThreadData，所以必须使用 m_sect 关键段对象来同步多个线程对该函数的调用。

```
CThreadSlotData::AllocSlot()
{
    进入关键段代码（EnterCriticalSection(m_sect);）
    搜索 m_pSlotData，查找空槽（SLOT）
    如果不存在空槽（第一次进入时，肯定不存在）
        分配或再分配内存以创建新槽，
        指针 m_pSlotData 指向分配的地址。
    得到新槽(SLOT)
    标志该 SLOT 为已用
    记录最新可用的 SLOT 到成员变量 m_nRover 中。
    离开关键段代码（LeaveCriticalSection(m_sect);）
    返回槽号
}
```

（2） GetThreadValue

GetThreadValue 用来获取调用线程的第 slot 个线程局部变量的值。每一个线程局部变量都占用一个且只一个槽位。

```
CThreadSlotData::GetThreadValue(int slot)
{
    //得到一个 CThreadData 型的指针 pData
    //pData 指向 MFC 线程私有存储空间。
```

```
        //m_tlsIndex 在_afxThreadData 创建时由构造函数创建
        pData=(CThreadData*)TlsGetValue(m_tlsIndex)，。
        如果指针空或 slot>pData->nCount,  则返回空。
        否则，返回 pData
    }
```

（3） SetValue

SetValue 用来把调用线程的第 slot 个线程局部变量的值（指针）存放到线程的 MFC 私有存储空间的第 slot 个槽位。

```
    CThreadSlotData::SetValue(int slot, void *pValue)
    {
        //通过 TLS 索引得到线程的 MFC 私有存储空间
        pData = (CThreadData*)TlsGetValue(m_tlsIndex)

        //没有得到值或者 pValue 非空且当前槽号，即
        //线程局部变量的个数
        //大于使用当前局部变量的线程个数时
        if （pData NULL or slot > pData->nCount && pValue！=NULL）
        {
            if  pData NULL //当前线程第一次访问该线程局部变量
            {
                创建一个 CThreadData 实例；
                添加到 CThreadSlotData::m_list；
                令 pData 指向它；
            }
            按目前为止，线程局部变量的个数为 pData->pData 分配或重分配内存，
            用来容纳指向真正线程数据的指针
            调用 TlsSetValue(pData)保存 pData
        }

        //把指向真正线程数据的 pValue 保存在 pData 对应的 slot 中
        pData->pData[slot] = pValue
    }
```

9.4 管理状态

在描述了 MFC 状态的实现机制之后，现在来讨论 MFC 的状态管理和相关状态的作用。

9.4.1 模块状态切换

模块状态切换就是把当前线程的线程状态的 m_pModuleState 指针指向即将运行模块的模块状态。

MFC 使用 AFX_MANAGE_STATE 宏来完成模块状态的切换，即进入模块时使用当前模板的模板状态，并保存原模板状态；退出模块时恢复原来的模块状态。这相当于状态的压栈和

出栈。实现原理如下。
先看 MFC 关于 AFX_MANAGE_STATE 的定义：

```
#ifdef _AFXDLL
    struct AFX_MAINTAIN_STATE
    {
    AFX_MAINTAIN_STATE(AFX_MODULE_STATE* pModuleState);
    ~AFX_MAINTAIN_STATE();
    protected:
    AFX_MODULE_STATE* m_pPrevModuleState;
    };
//AFX_MANAGE_STATE 宏的定义：
    #define AFX_MANAGE_STATE(p)    AFX_MAINTAIN_STATE _ctlState(p);
#else    // _AFXDLL
    #define AFX_MANAGE_STATE(p)
#endif //!_AFXDLL
```

如果使用 MFC DLL，MFC 提供类 AFX_MAINTAIN_STATE 来实现状态的压栈和出栈，AFX_MANAGE_SATATE 宏的作用是定义一个 AFX_MAINTAIN_STATE 类型的局部变量_ctlState。
AFX_MAINTAIN_STATE 的构造函数在其成员变量 m_pPrevModuleState 中保存当前的模块状态对象，并把参数指定的模块状态设定为当前模块状态。所以该宏作为入口点的第一条语句就切换了模块状态。
在退出模块时，局部变量_ctlState 将自动地销毁，这导致 AFX_MAINTAIN_STATE 的析构函数被调用，析构函数把保存在 m_pPrevModuleState 的状态设置为当前状态。

AFX_MANAGE_SATATE 的参数在不同场合是不一样的，例如，

- DLL 的输出函数使用

```
AFX_MANAGE_SATATE(AfxGetStaticModuleState());
```

其中，AfxGetStaticModuleState 返回 DLL 的模块状态 afxModuleState。

- 窗口函数使用

```
AFX_MANAGE_STATE(_afxBaseModuleState.GetData());
```

其中，_afxBaseModuleState.GetData()返回的是应用程序的全局模块状态。
OLE 使用的模块切换方法有所不同，这里不作讨论。

上面讨论了线程执行行不同模块的代码时切换模块状态的情况。在线程创建时怎么处理模块状态呢？

- 一个进程(使用 MFC 的应用程序)的主线程创建线程模块状态和进程模块状态，前者是_AFX_THREAD_STATE 类的实例，后者是_AFX_BASE_MODULE_STATE 类的实例。
- 当进程的新的线程被创建时，它创建自己的线程状态，继承父线程的模块状态。在线程的入口函数_AfxThreadEntry 完成这样的处理，该函数的描述见 8.5.3 节。

9.4.2 扩展 DLL 的模块状态

7.3.1 节指出扩展 DLL 的实现必须遵循五条规则，为此，首先在扩展 DLL 实现文件里头，定义 AFX_EXTENSION_MODULE 类型的静态扩展模块变量，然后在 DllMain 入口函数里

头使用 AfxInitExtension 初始化扩展模块变量，并且实现和输出一个初始化函数供扩展 DLL 的使用者调用。
使用者必须具备一个 CWinApp 对象，通常在它的 InitInstance 函数中调用扩展 DLL 提供的初始化函数。
一般用以下的几段代码完成上述任务。首先是扩展模块变量的定义和初始化：

```
static AFX_EXTENSION_MODULE extensionDLL;
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID)
{
if (dwReason == DLL_PROCESS_ATTACH)
{
    // Extension DLL one-time initialization
    if (!AfxInitExtensionModule(extensionDLL,hInstance))
        return 0;
    ……
}
}
```

然后是扩展DLL的初始化函数，假定初始化函数命名为InitMyDll，InitMyDll被定义为“C”链接的全局函数，并且被输出。

```
// wire up this DLL into the resource chain
extern “C” void WINAPI InitMyDll()
{
CDynLinkLibrary* pDLL = new
CDynLinkLibrary(extensionDLL, TRUE);
ASSERT(pDLL != NULL);
...
}
```

最后是调用者的处理，假定在应用程序对象的 InitInstance 函数中调用初始化函数：

```
BOOL CMyApp::InitInstance()
{
    InitMyMyDll();
    …
}
```

上述这些代码只有在动态链接到 MFC DLL 时才有用。下面，对这些代码进行分析和解释

9.4.2.1 _AFX_EXTENSION_MODULE

在分析代码之前，先讨论描述扩展模块状态的_AFX_EXTENSION_MODULE 类。_AFX_EXTENSION_MODULE 没有基类，其定义如下：

```
struct AFX_EXTENSION_MODULE
{
BOOL bInitialized;
HMODULE hModule;
HMODULE hResource;
CRuntimeClass* pFirstSharedClass;
```

```
COleObjectFactory* pFirstSharedFactory;
};
```

其中：

第一个域表示该结构变量是否已经被初始化了；

第二个域用来保存扩展 DLL 的模块句柄；

第三个域用来保存扩展 DLL 的资源句柄；

第四个域用来保存扩展 DLL 要输出的 CRuntimeClass 类；

第五个域用来保存扩展 DLL 的 OLE Factory。

该结构用来描述一个扩展 DLL 的模块状态信息，每一个扩展 DLL 都要定义一个该类型的静态变量，例如 extensionDLL。

在 DllMain 中，调用 AfxInitExtensionModule 函数来初始化本 DLL 的静态变量该变量(扩展模块状态)，如 extensionDLL。函数 AfxInitExtensionModule 原型如下：

```
BOOL AFXAPI AfxInitExtensionModule(
    AFX_EXTENSION_MODULE& state, HMODULE hModule)
```

其中：

参数 1 是 DllMain 传递给它的扩展 DLL 的模块状态，如 extensionDLL；

参数 2 是 DllMain 传递给它的模块句柄。

AfxInitExtensionModule 函数主要作以下事情：

（1）把扩展 DLL 模块的模块句柄 hModule、资源句柄 hModule 分别保存到参数 state 的成员变量 hModule、hResource 中；

（2）把当前模块状态的 m_classList 列表的头保存到 state 的成员变量 pFirstSharedClass 中，m_classInit 的头设置为模块状态的 m_pClassInit。在扩展 DLL 模块进入 DllMain 之前，如果该扩展模块构造了静态 AFX_CLASSINIT 对象，则在初始化时把有关 CRuntimeClass 信息保存在当前模块状态（注意不是扩展 DLL 模块，而是应用程序模块）的 m_classList 列表中。因此，扩展 DLL 模块初始化的 CRuntimeClass 信息从模块状态的 m_classList 中转存到扩展模块状态 state 的 pFirstSharedClass 中，模块状态的 m_classInit 恢复被该 DLL 改变前的状态。关于 CRuntimeclass 信息和 AFX_CLASSINIT 对象的构造，在 3.3.1 节曾经讨论过。一个扩展 DLL 在初始化时，如果需要输出它的 CRuntimeClass 对象，就可以使用相应的 CRuntimeClass 对象定义一个静态的 AFX_CLASSINIT 对象，而不一定要使用 IMPLEMENT_SERIAL 宏。当然，可以序列化的类必定导致可以输出的 CRuntimeClass 对象。

（3）若支持 OLE 的话，把当前模块状态的 m_factoryList 的头保存到 state 的成员变量 pFirstSharedFactory 中。m_factoryList 的头设置为模块状态的 m_m_pFactoryInit。

（4）这样，经过初始化之后，扩展 DLL 模块包含了扩展 DLL 的模块句柄、资源句柄、本模块初始化的 CRuntimeClass 类等等。

扩展 DLL 的初始化函数将使用扩展模块状态信息。下面，讨论初始化函数的作用。

9.4.2.2 扩展 DLL 的初始化函数

在初始化函数 InitMyDll 中，创建了一个动态的 CDynLinkLibrary 对象，并把对象指针保存在 pDLL 中。CDynLinkLibrary 类从 CCmdTarget 派生，定义如下：

```
class CDynLinkLibrary : public CCmdTarget
{
DECLARE_DYNAMIC(CDynLinkLibrary)
```

```
public:
// Constructor
CDynLinkLibrary(AFX_EXTENSION_MODULE& state,
BOOL bSystem = FALSE);

// Attributes
HMODULE m_hModule;
HMODULE m_hResource;                    // for shared resources
CTypedSimpleList<CRuntimeClass*> m_classList;
#ifndef _AFX_NO_OLE_SUPPORT
    CTypedSimpleList<COleObjectFactory*> m_factoryList;
#endif
BOOL m_bSystem;                         // TRUE only for MFC DLLs

// Implementation
public:
CDynLinkLibrary* m_pNextDLL;            // simple singly linked list
virtual ~CDynLinkLibrary();

#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif // _DEBUG
};
```

CDynLinkLibrary 的结构和 AFX_EXTENSION_MODULE 有一定的相似性，存在对应关系。CDynLinkLibrary 构造函数的第一个参数就是经过 AfxInitExtensionModule 初始化后的扩展 DLL 的模块状态，如 extensionDLL，第二个参数表示该 DLL 模块是否是系统模块。

创建 CDynLinkLibrary 对象导致 CCmdTarget 和 CDynLinkLibrary 类的构造函数被调用。CCmdTarget 的构造函数将获取模块状态并且保存在成员变量 m_pModuleState 中。CDynLinkLibrary 的构造函数完成以下动作：

构造列表 m_classList 和 m_factoryList；

把参数 state 的域 hModule、hResource 复制到对应的成员变量 m_hModule、m_hResource 中；

把 state 的 pFirstSharedClass、pFirstSharedFactory 分别插入到 m_classList 列表、m_factoryList 列表的表头；

把参数 2 的值赋值给成员变量 m_bSystem 中；

至此，CDynLinkLibrary 对象已经构造完毕。之后，CDynLinkLibrary 构造函数把 CDynLinkLibrary 对象自身添加到当前模块状态（调用扩展 DLL 的应用程序模块或者规则 DLL 模块）的 CDynLinkLibrary 列表 m_libraryList 的表头。为了防止多个线程修改模块状态的 m_libraryList，访问 m_libraryList 时使用了同步机制。

这样，调用模块执行完扩展模块的初始化函数之后，就把该扩展 DLL 的资源、CRuntimeClass 类、OLE Factory 等链接到调用者的模块状态中，形成一个链表。图 9-8 表明了这种关系链。

综合以上分析，可以知道：

扩展 DLL 的模块仅仅在该 DLL 调用 DllMain 期间和调用初始化函数期间被使用，在这些初

始化完毕之后，扩展 DLL 模块被链接到当前调用模块的模块状态中，因此它所包含的资源信息等也就被链接到调用扩展 DLL 的应用程序或者规则 DLL 的模块状态中了。扩展 DLL 扩展了调用者的资源等，这是“扩展 DLL”得名的原因之一。
也正因为扩展 DLL 没有自己的模块状态（指 AFX_MODULE_STATE 对象，扩展 DLL 模块状态不是），而且必须由有模块状态的模块来使用，所以只有动态链接到 MFC 的应用程序或者规则 DLL 才可以使用扩展 DLL 模块的输出函数或者输出类。

9.4.3 核心 MFC DLL

所谓核心 MFC DLL，就是 MFC 核心类库形成的 DLL，通常说动态链接到 MFC，就是指核心 MFC DLL。
核心 MFC DLL 实际上也是一种扩展 DLL，因为它定义了自己的扩展模块状态 coreDLL，实现了自己的 DllMain 函数，使用 AfxInitExtensionModule 初始化核心 DLL 的扩展模块状态 coreDLL，并且 DllMain 还创建了 CDynLinkLibrary，把核心 DLL 的扩展模块状态 coreDLL 链接到当前应用程序的模块状态中。所有这些，都符合扩展 DLL 的处理标准。
但是，核心 MFC DLL 是一种特殊的扩展 DLL，因为它定义和实现了 MFC 类库，模块状态、线程状态、进程状态、状态管理和使用的机制就是核心 MFC DLL 定义和实现的。例如核心 MFC DLL 定义和输出的模块状态变量，即_afxBaseModuleState，就是动态链接到 MFC 的 DLL 的应用程序的模块状态。
但是 MFC DLL 不作为独立的模块表现出来，而是把自己作为一个扩展模块来处理。当应用程序动态链接到 MFC DLL 时，MFC DLL 把自己的扩展模块状态 coreDLL 链接到模块状态 - afxBaseModuleState，模块状态的成员变量 m_hCurrentInstanceHandle 指定为应用程序的句柄。当规则 DLL 动态链接到 MFC DLL 时，由规则 DLL 的 DllMain 把核心 MFC DLL 的扩展模块状态 coreDLL 链接到规则 DLL 的模块状态 afxModuleState 中，模块状态 afxModuleState 的 m_hCurrentInstanceHandle 指定为规则 DLL 的句柄。
关于 afxModuleState 和规则 DLL 的模块状态，见下一节的讨论。

9.4.4 动态链接的规则 DLL 的模块状态的实现

在本节中，动态链接到 MFC DLL（定义了_AFXDLL）的规则 DLL 在下文简称为规则 DLL。
（1）规则 DLL 的模块状态的定义
规则 DLL 有自己的模块状态_afxModuleState，它是一个静态变量，定义如下：

```
static _AFX_DLL_MODULE_STATE afxModuleState;
```

_AFX_DLL_MODULE_STATE 的基类是 AFX_MODULE_STATE。
在前面的模块状态切换中提到的 AfxGetStaticModuleState 函数，其定义和实现如下：

```
_AFX_MODULE_STATE* AFXAPI AfxGetStaticModuleState()
{
AFX_MODULE_STATE* pModuleState = &afxModuleState;
return pModuleState;
}
```

它返回规则 DLL 的模块状态 afxModuleState。
规则 DLL 的内部函数使用 afxModuleState 作为模块状态；输出函数在被调用的时候首先切换到该模块状态，然后进一步处理。

（2）规则 DLL 的模块状态的初始化

从用户角度来看，动态链接到 MFC DLL 的规则 DLL 不需要 DllMain 函数，只要提供 CWinApp 对象即可。其实，MFC 内部是在实现扩展 DLL 的方法基础上来实现规则 DLL 的，它不仅为规则 DLL 提供了 DllMain 函数，而且规则 DLL 也有扩展 DLL 模块状态 controlDLL。顺便指出，和扩展 DLL 相比，规则 DLL 有一个 CWinApp（或其派生类）应用程序对象和一个模块状态 afxModuleState。应用程序对象是全局对象，所以在进入规则 DLL 的 DllMain 之前已经被创建，DllMain 可以调用它的初始化函数 InitInstance。模块状态 afxModuleState 是静态全局变量，也在进入 DllMain 之前被创建，DllMain 访问模块状态时得到的就是该变量。扩展 DLL 是没有 CWinApp 对象和模块状态的，它只能使用应用程序或者规则 DLL 的 CWinApp 对象和模块状态。

由于核心 MFC DLL 的 DllMain 被调用的时候，访问的必定是应用程序的模块状态，要把核心 DLL 的扩展模块状态链接到规则 DLL 的模块状态中，必须通过规则 DLL 的 DllMain 来实现。

规则 DLL 的 DllMain（MFC 内部实现）把参数 1 表示的模块和资源句柄通过 AfxWinInit 函数保存到规则 DLL 的模块状态中。顺便指出，WinMain 也通过 AfxWinInit 函数把资源和模块句柄保存到应用程序的模块状态中。

然后，该 DllMain 还创建了一个 CDynLinkLibrary 对象，把核心 MFC DLL 的扩展模块 coreDLL 链接到本 DLL 的模块状态 afxModuleState。

接着，DllMain 得到自己的应用程序对象并调用 InitInstance 初始化。

之后，DllMain 创建另一个 CDynLinkLibrary 对象，把本 DLL 的扩展模块 controlDLL 链接到本 DLL 的模块状态 afxModuleState。

（3）使用规则 DLL 的应用程序可不需要 CwinApp 对象

规则 DLL 的资源等是由 DLL 内部使用的，不存在资源或者 CRuntimeClass 类输出的问题，这样调用规则 DLL 的程序不必具有模块状态，不必关心规则 DLL 的内部实现，不一定需要 CwinApp 对象，所以可以是任意 Win32 应用程序，

还有一点需要指出，DllMain 也是规则 DLL 的入口点，在它之前，调用 DllMain 的 RawDllMain 已经切换了模块状态，RawDllMain 是静态链接的，所以不必考虑状态切换。

9.5 状态信息的作用

在分析了 MFC 模块状态的实现基础和管理机制之后，现在对状态信息的作用进行专门的讨论。

9.5.1.1 模块信息的保存和管理

传统上，线程状态、模块状态等包含的信息是全局变量，但是为了支持 Win32s、多线程、DLL 等，这些变量必须是限于进程或者线程范围内有效，或者限于某个模块内有效。也就是，不再可能把它们作为全局变量处理。因此，MFC 引入模块、线程、模块-线程状态等来保存和管理一些重要的信息。

例如：一个模块注册了一个“窗口类”之后，应用程序要保存“窗口类”的名字，以便在模块退出时取消注册的“窗口类”。因此，模块状态使用成员变量 m_szUnregisterList 在注册成功之后保存的“窗口类”名字。窗口注册见 2.2.1 节。

又如：Tooltip 窗口是线程相关的，每个线程一个，所以线程状态用成员变量 m_pToolTip 来

保存本线程的 MFC Tooltip 窗口对象。Tooltip 窗口见 13.2.4.4 节。
还有，MFC 对象是线程和模块相关的，所以模块线程中有一组变量用来管理本线程的 MFC 对象到 Windows 对象的映射关系。关于 MFC 对象和 Windows 对象的映射，见稍后的讨论。

模块状态、线程状态、模块线程状态的每个成员变量都有自己存在的必要和作用，这里就不一一论述了，在此，只是强调模块状态自动地实现对模块句柄和资源句柄等信息的保存和管理，这对 MFC 应用程序是非常重要的。
SDK 下的应用程序或者 DLL，通常使用一个全局变量来保存模块/资源句柄。有了模块状态之后，程序员就不必这么作了。规则 DLL 或者应用程序的模块和资源句柄在调用 DllMain 或 WinMain 时被保存到了当前模块的模块状态中。如果是扩展 DLL，则其句柄被保存到扩展模块状态中，并通过 CDynLinkLibrary 对象链接到主模块的模块状态。
图 9-8 示意了 MFC 模块状态对资源、CRuntimeClass 对象、OLE 工厂等模块信息的管理。
图 9-8 的说明：

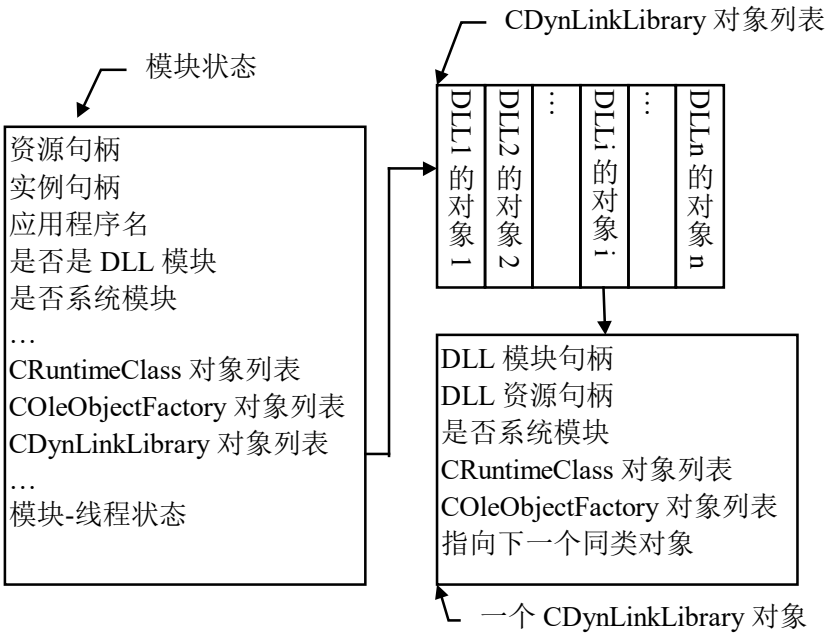


图 9-8　模块状态的一个用途

左边的主模块状态表示动态链接到 MFC DLL 的应用程序或者规则 DLL 的模块状态，其资源句柄和模块句柄用来查找和获取资源，资源句柄一般是应用程序的模块句柄；CRuntimeClass 对象列表和 COleObjectFactory 对象列表分别表示该模块初始化了的 CRuntimeClass 对象和该模块的 OLE 工厂对象；CDynLinkLibrary 列表包含了它引用的系列扩展 DLL 的扩展模块状态（包括核心 MFC DLL 的状态），链表中的每一个 CDynLinkLibrary 对象对应一个扩展模块状态，代表了创建该对象的扩展 DLL 的有关资源、信息。
MFC 查找资源、CRuntimeClass 类、OLE 工厂时，首先查找模块状态，然后，遍历 CDynLinkLibrary 表搜索相应的对象。下面两节举例说明。

9.5.2 MFC 资源、运行类信息的查找

MFC 内部使用的资源查找函数是：

HINSTANCE AfxFindResourceHandle(LPCTSTR lpszName, LPCTSTR lpszType)：

其中：

参数 1 是要查找的资源名称，参数 2 是要查找的资源类型。

返回包含指定资源的模块的句柄。

上述函数的查找算法如下：

第一， 如果进程模块状态（主模块）不是系统模块，则使用::FindResource（下同）搜索它，成功则返回；

第二， 如果没有找到，则遍历 CDynLinkLibrary 对象列表，搜索所有的非系统模块，成功则返回；

第三， 如果没有找到，则检查主模块的语言资源，成功则返回；

第四， 如果没有找到，并且主模块是系统模块，则搜索它，成功则返回；

第五， 如果没有找到，则遍历 CDynLinkLibrary 对象列表，搜索所有的系统模块，成功则返回；

第六， 如果没有找到，则使用 AfxGetResourceHanlde 返回应用程序的资源。

需要指出的是，遍历 CDynLinkLibrary 对象列表时，必须采取同步措施，防止其他线程改变链表。MFC 是通过锁定全局变量 CRIT_DYNLINKLIST 来实现的，类似的全局变量 MFC 定义了多个。

运行时类信息的查找算法类似。

3.3.4 节指出，对象进行“<<”序列化操作时，首先需要搜索到指定类的运行时信息，方法如下：

```
CRuntimeClass* PASCAL CRuntimeClass::Load(
    CArchive& ar, UINT* pwSchemaNum)
```

第一， 遍历主模块的 CRuntimeClass 对象列表 m_classList，搜索主模块是否实现了指定的 CRuntimeClass 类；

第二， 遍历 CDynLinkLibrary 对象列表 m_libraryList；对每一个 CDynLinkLibrary 对象，遍历它的 CRuntimeClass 对象列表 m_classList。这样，所有的扩展 DLL 模块的 CRuntimeClass 对象都会被搜索到。

9.5.3 模块信息的显示

遍历模块状态和 CDynLinkLibrary 列表，可以显示模块状态及其扩展模块状态的有关信息。下面，给出一个实现，它显示程序的当前模块名称、句柄和初始化的 CRuntimeClass 类，然后显示所有扩展模块的名称名称、句柄和初始化的 CRuntimeClass 类。

```
#ifdef _DEBUG
    AFX_MODULE_STATE* pState = AfxGetModuleState();
    //显示应用程序的名称和句柄
    TRACE("APP %s HANDLE %x\r\n", pState->m_lpszCurrentAppName,
```

```
        pState->m_hCurrentInstanceHandle);
    TCHAR szT[256];
    int nClasses;
    nClasses=0;
    //显示 CRuntimeClass 类信息
    AfxLockGlobals(CRIT_RUNTIMECLASSLIST);
    for (CRuntimeClass* pClass = pModuleState->m_classList;
        pClass != NULL;pClass = pClass->m_pNextClass)
    {
        nClasses++;
        TRACE("CRuntimeClass: %s\r\n",pClass->m_lpszClassName, );
    }
    AfxUnlockGlobals(CRIT_RUNTIMECLASSLIST);
    TRACE("all %d classes\r\n", nClasses);

    //遍历 CDynLinkLibrary 列表
    AfxLockGlobals(CRIT_DYNLINKLIST);
    for (CDynLinkLibrary* pDLL = pState->m_libraryList; pDLL != NULL;
        pDLL = pDLL->m_pNextDLL)
    {
        // 得到模块名并且显示
        TCHAR szName[64];
        GetModuleFileName(pDLL->m_hModule, szName, sizeof(szName));
        TRACE("MODULE %s HANDLE IS %x    \r\n", szName, pDLL->m_hModule);

        //得到 CRuntimeClass 信息并显示
        nClasses = 0;
        for (CRuntimeClass* pClass = pDLL->m_classList;
            pClass != NULL; pClass = pClass->m_pNextClass)
        {
            nClasses++;
            TRACE("CRuntimeClass: %s\r\n",pClass->m_lpszClassName, );
        }
        wsprintf(szT, _T("      Module %s has %d classes"),szName, nClasses);
    }
    AfxUnlockGlobals(CRIT_DYNLINKLIST);
    #endif
```

使用 MFC 提供的调试函数 AfxDoForAllClasses 可以得到 DLL 模块的输出 CRuntimeClass 类的信息。上述实现类似于 AfxDoForAllClasses 函数的处理，只不过增加了模块名和模块句柄信息。

9.5.4 模块-线程状态的作用

由模块-线程状态类的定义可知，一个模块-线程状态包含了几类 Windows 对象—MFC 对象

的映射。下面讨论它们的作用。

9.5.4.1 只能访问本线程 MFC 对象的原因

MFC 规定：

（1） 不能从一个非 MFC 线程创建和访问 MFC 对象

如果一个线程被创建时没有用到 CWinThread 对象，比如，直接使用“C”的_beginthread 或者_beginthreadex 创建的线程，则该线程不能访问 MFC 对象；换句话说，只有通过 CWinThread 创建 MFC 线程对象和 Win32 线程，才可能在创建的线程中使用 MFC 对象。

（2） 一个线程仅仅能访问它所创建的 MFC 对象

这两个规定的原因是：

为了防止多个线程并发地访问同一个 MFC 对象，MFC 对象和 Windows 对象之间有一个一一对应的关系，这种关系以映射的形式保存在创建线程的当前模块的模块-线程状态信息中。当一个线程使用某个 MFC 对象指针 P 时，ASSERT_VALID(P)将验证当前线程的当前模块是否有 Windows 句柄和 P 对应，即是否创建了 P 所指的 Windows 对象，验证失败导致 ASSERT 断言中断程序的执行。如果一个线程要使用其他线程的 Windows 对象，则必须传递 Windows 对象句柄，不能传递 MFC 对象指针。

当然一般来说，MFC 应用程序仅仅在 Debug 版本下才检查这种映射关系，所以访问其他线程的 MFC 对象的程序在 Realease 版本下表面上不会有问题，但是 MFC 对象被并发访问的后果是不可预见的。

9.5.4.2 实现 MFC 对象和 Windows 对象之间的映射

MFC 提供了几个函数完成 MFC 对象和 Windows 对象之间的映射或者解除这种映射关系，以及从 MFC 对象得到 Windows 对象或者从 Windows 对象得到或创建相应的 MFC 对象。每一个 MFC 对象类都有成员函数 Attach 和 Detach，FromHandle 和 FromHandlePermanent，AssertValid。这些成员函数的形式如下：

- Attach(HANDLE Windows_Object_Handle)

例如：CWnd 类的是 Attach(HANLDE hWnd)，CDC 类的是 Attach(HDC hDc)。

Attach 用来把一个句柄永久性(Perment)地映射到一个 MFC 对象上：它把一个 Windows 对象捆绑(Attach)到一个 MFC 对象上，MFC 对象的句柄成员变量赋值为 Windows 对象句柄，该 MFC 对象应该已经存在，但是句柄成员变量为空。

- Detach（）

Detach 用来取消 Windows 对象到 MFC 对象的永久性映射。如果该 Windows 对象有一个临时的映射存在，则 Detach 不理会它。MFC 让线程的 Idle 清除临时映射和临时 MFC 对象。

- FromHandle(HANDLE Windows_Object)

它是一个静态成员函数。如果该 Windows 对象没有映射到一个 MFC 对象，FromHandle 则创建一个临时的 MFC 对象，并把 Windows 对象映射到临时的 MFC 对象上，然后返回临时 MFC 对象。

- FromHandlePermanent(HANDLE Windows_Object)

它是一个静态成员函数。如果该 Windows 对象没有永久地映射到一个 MFC 对象上，则返回 NULL，否则返回对应的 MFC 对象。

- AssertValid()

它是从 CObject 类继承来的虚拟函数。MFC 覆盖该函数，实现了至少一个功能：判断当前 MFC 对象的指针 this 是否映射到一个对应的可靠的 Windows 对象。

图 9-9 示意了 MFC 对映射结构的实现层次，对图 9-9 解释如下。

图 9-9　MFC 对象和 Windows 对象之间的映射的实现

图中上面的虚线框表示使用映射关系的高层调用，包括上面讲述的几类函数。MFC 和应用程序通过它们创建、销毁、使用映射关系。

图中中间的虚线框表示 MFC 使用 CHandleMap 类实现对映射关系的管理。一个 CHandleMap 对象可以通过两个成员变量来管理两种映射数据：临时映射和永久映射。模块-线程状态给每一类 MFC 对象分派一个 CHandleMap 对象来管理其映射数据(见模块-线程类的定义)，例如 m_pmapHWND 所指对象用来保存 CWnd 对象（或派生类对象）和 Windows window 之间的映射。

下面的虚线框表示映射关系的最底层实现，MFC 使用通用类 CMapPtrToPtr 来管理 MFC 对象指针和 Windows 句柄之间的映射数据。

对本节总结如下：

- MFC 的映射数据保存在模块-线程状态中，是线程和模块局部的。每个线程管理自己映射的数据，其他线程不能访问到本线程的映射数据，也就不允许使用本线程的 MFC 对象。
- 每一个 MFC 对象类（CWnd、CDC 等）负责创建或者管理这类线程-模块状态的对应 CHandleMap 类对象。例如，CWnd::Attach 创建一个永久性的映射保存在 m_pmapHwnd 所指对象中，如果 m_pmapHand 还没有创建，则使用 AfxMapHWND 创建相应的 CHandleMap 对象。
- 映射分两类：永久性的或者临时的。

9.5.4.3 临时对象的处理

在 2.4 节就曾经提到了临时对象，现在是深入了解它们的时候了。

第一， 临时对象指 MFC 对象，是 MFC 或者程序员使用 FromHandle 或者 SelectObject 等从一个 Windows 对象句柄创建的对应的 MFC 对象。

第二， 在模块-线程状态中，临时 MFC 对象的映射是和永久映射分开保存的。

第三， 临时 MFC 对象在使用完毕后由 MFC 框架自动删除，MFC 在线程的 Idle 处理中删除本线程的临时 MFC 对象，为了防止并发修改，通过线程状态 m_nTempMapLock（等于 0，可以修改，大于 0，等待）来同步。所以，临时 MFC 对象不能保存备用。

9.6 状态对象的删除和销毁

至此，本章讨论了 MFC 的线程局部存储机制，MFC 状态的定义、实现和用途。在程序或者 DLL 退出之前，模块状态被销毁；在线程退出时，线程状态被销毁。状态对象被销毁之前，它活动期间所动态创建的对象被销毁，动态分配的内存被释放。

先解释几个函数：

AfxTermExtensionModule(HANDLE hInstanceOfDll，BOOL bAll);

若 bAll 为真，则该函数销毁本模块(hInstanceOfDll 标识的模块)的模块状态的 m_libraryList 列表中所有动态分配的 CDynLinkLibrary 对象，否则，该函数清理本 DLL 动态分配的 CDynLinkLibrary 对象，并调用 AfxTerLocalData 释放本 DLL 模块为当前线程的线程局部变量分配的堆空间。

AfxTermLocalData(HANDLE hInstance, BOOL bAll);

若 bAll 为真，则删除 MFC 线程局部存储的所有槽的指针所指的对象，也就是销毁当前线程的全部局部变量，释放为这些线程局部变量分配的内存；否则，仅仅删除、清理当前线程在 hInstance 表示的 DLL 模块中创建的线程局部变量。

参与清理工作的函数有多种、多个，下面结合具体情况简要描述它们的作用。

（1）对动态链接到 MFC DLL 的应用程序

动态链接到 MFC DLL 的应用程序退出时，将在 DllMain 和 RawDllMain 处理进程分离时清理状态对象，该 DllMain 和 RawDllMain 是核心 MFC DLL 的入口和出口，在 DLLINIT.CPP 文件中实现，和进程分离时完成如下动作：

DllMain 调用 AfxTermExtensionModule(coreDll)清理核心 MFC DLL 的模块状态；调用 AfxTermExtensionModule(coreDll, TRUE) 清理 OLE 私有的模块状态；调用 AfxTermLocalData(NULL, TRUE)释放本进程或者线程所有的局部变量。

RawDllMain 在 DllMain 之后调用，它调用 AfxTlsRealease；AfxTlsRealease 减少对 _afxThreadData 的引用计数，如果引用数为零，则调用对应的 CThreadSlotData 析构函数清理_afxThreadData 所指对象。

（2）对静态链接到 MFC DLL 的应用程序

如果是静态链接到 MFC DLL 的应用程序，由于 RawDllMain 和 DllMain 不起作用，将由一个静态变量析构时完成状态的清除：

有一个 AFX_TERM_APP_STATE 类型的静态变量，在程序结束时将被销毁，导致析构函数被调用，析构函数完成以下动作：

调用 AfxTermLocalData(NULL, TRUE)释放本进程（主线程）的所用局部数据。

（3）对于动态链接到 MFC DLL 的规则 DLL

对于动态链接到 MFC DLL 的规则 DLL，将在 RawDllMain 和 DllMain 中清理状态对象。这两个函数在 DllModule.cpp 中定义，是规则 DLL 的入口和出口。当和进程分离时，分别有如下动作：

DllMain 清除该模块的模块-线程状态中的所有临时映射，清除临时 MFC 对象；调用 AfxWinTerm；调用 AfxTermExtensionModule(controlDLL, TRUE)，释放本 DLL 模块状态 m_libraryList 中的所有 CDynLinkLibrary 对象。

RawDllMain 设置线程状态的模块状态指针，使它指向线程状态的 m_PrevModuleState 所指状态。

（4）对于静态链接到 MFC DLL 的 DLL

对于静态链接到 MFC DLL 的 DLL，只有 DllMain 会被调用，执行以下动作：

清除该模块的模块-线程状态中的所有临时映射，清除临时 MFC 对象；调用 AfxWinTerm；调用 AfxTermLocalData(hInstance, TRUE)清理本 DLL 模块的当前线程的线程局部数据。

另外，它定义一个_AFX_TERM_DLL_STATE 类型的静态变量，在 DLL 退出时该变量被销毁，导致其析构函数被调用。析构函数完成如下动作：

调用 AfxTermateLocalData(NULL, TRUE); 调用 AfxCriticlTerm 结束关键变量；调用 AfxTlsRealease。

（5）线程终止时

当使用 AFxBeginThread 创建的线程终止时，将调用 AfxTermThread(HANDLE hInstance)作结束线程的清理工作（参数为 NULL）：销毁临时 MFC 对象，销毁本线程的线程局部变量，等等。

另外，当 DLL 模块和 AfxBeginThread 创建的线程分离时，也调用 AfxTermThread(hInstance)，参数是模块的句柄，销毁临时 MFC 对象，销毁本线程在本 DLL 创建的线程局部变量，等等。所以，AfxTermThread 可能被调用两次。

最后，CThreadLocal 和 CProcessLocal 的实例将被销毁，析构函数被调用：如果 MFC 线程局部存储空间的槽 m_nSlot 所指的线程局部对象还没有销毁，则销毁它。

_afxThreadData 在 MFC DLL 的 RawDllMain 或者随着_AFX_TERM_APP_STATE 析构函数的调用，_afxThreadData 所指对象被销毁。_afxThreadData 所指对象销毁之后，所有的状态相关的内存都被释放。

第10章　内存分配方式和调试机制

10.1　M 内存分配

10.1.1 内存分配函数

MFCWin32 或者 C 语言的内存分配 API，有四种内存分配 API 可供使用。

（1）　Win32 的堆分配函数

每一个进程都可以使用堆分配函数创建一个私有的堆——调用进程地址空间的一个或者多个页面。DLL 创建的私有堆必定在调用 DLL 的进程的地址空间内，只能被调用进程访问。HeapCreate 用来创建堆；HeapAlloc 用来从堆中分配一定数量的空间，HeapAlloc 分配的内存是不能移动的；HeapSize 可以确定从堆中分配的空间的大小；HeapFree 用来释放从堆中分配的空间；HeapDestroy 销毁创建的堆。

（2）　Windows 传统的全局或者局部内存分配函数

由于 Win32 采用平面内存结构模式，Win32 下的全局和局部内存函数除了名字不同外，其他完全相同。任一函数都可以用来分配任意大小的内存（仅仅受可用物理内存的限制）。用法可以和 Win16 下基本一样。

Win32 下保留这类函数保证了和 Win16 的兼容。

（3）　C 语言的标准内存分配函数

C 语言的标准内存分配函数包括以下函数：

malloc，calloc，realloc，free，等。

这些函数最后都映射成堆 API 函数，所以，malloc 分配的内存是不能移动的。这些函数的调式版本为

malloc_dbg，calloc_dbg，realloc_dbg，free_dbg，等。

（4）　Win32 的虚拟内存分配函数

虚拟内存 API 是其他 API 的基础。虚拟内存 API 以页为最小分配单位，X86 上页长度为 4KB，可以用 GetSystemInfo 函数提取页长度。虚拟内存分配函数包括以下函数：

- LPVOID VirtualAlloc(LPVOID lpvAddress,
  DWORD cbSize,
  DWORD fdwAllocationType,
  DWORD fdwProtect);

该函数用来分配一定范围的虚拟页。参数 1 指定起始地址；参数 2 指定分配内存的长度；参数 3 指定分配方式，取值 MEM_COMMINT 或者 MEM_RESERVE；参数 4 指定控制访问本次分配的内存的标识，取值为 PAGE_READONLY、PAGE_READWRITE 或者 PAGE_NOACCESS。

- LPVOID VirtualAllocEx(HANDLE process,
  LPVOID lpvAddress,
  DWORD cbSize,
  DWORD fdwAllocationType,

DWORD fdwProtect);

该函数功能类似于 VirtualAlloc，但是允许指定进程 process。VirtaulFree、VirtualProtect、VirtualQuery 都有对应的扩展函数。

- BOOL VirtualFree(LPVOID lpvAddress,
  DWORD dwSize,
  DWORD dwFreeType);

该函数用来回收或者释放分配的虚拟内存。参数 1 指定希望回收或者释放内存的基地址；如果是回收，参数 2 可以指向虚拟地址范围内的任何地方，如果是释放，参数 2 必须是 VirtualAlloc 返回的地址；参数 3 指定是否释放或者回收内存，取值为 MEM_DECOMMINT 或者 MEM_RELEASE。

- BOOL VirtualProtect(LPVOID lpvAddress,
  DWORD cbSize,
  DWORD fdwNewProtect,
  PDWORD pfdwOldProtect);

该函数用来把已经分配的页改变成保护页。参数 1 指定分配页的基地址；参数 2 指定保护页的长度；参数 3 指定页的保护属性，取值 PAGE_READ、PAGE_WRITE、PAGE_READWRITE 等等；参数 4 用来返回原来的保护属性。

- DWORD VirtualQuery(LPCVOID lpAddress,
  PMEMORY_BASIC_INFORMATION lpBuffer,
  DWORD dwLength
  );

该函数用来查询内存中指定页的特性。参数 1 指向希望查询的虚拟地址；参数 2 是指向内存基本信息结构的指针；参数 3 指定查询的长度。

- BOOL VirtualLock(LPVOID lpAddress,DWORD dwSize);

该函数用来锁定内存，锁定的内存页不能交换到页文件。参数1指定要锁定内存的起始地址；参数2指定锁定的长度。

- BOOL VirtualUnLock(LPVOID lpAddress,DWORD dwSize);

参数1指定要解锁的内存的起始地址；参数2指定要解锁的内存的长度。

10.1.2 C++的 new 和 delete 操作符

MFC 定义了两种作用范围的 new 和 delete 操作符。对于 new，不论哪种，参数 1 类型必须是 size_t，且返回 void 类型指针。

（1） 全局范围内的 new 和 delete 操作符

原型如下：

```
void _cdecl ::operator new(size_t nSize);
void __cdecl operator delete(void* p);
```

调试版本：

```
void* __cdecl operator new(size_t nSize, int nType,
LPCSTR lpszFileName, int nLine)
```

（2） 类定义的 new 和 delete 操作符

原型如下：

```
void* PASCAL classname::operator new(size_t nSize);
```

void PASCAL classname::operator delete(void* p);

类的 operator new 操作符是类的静态成员函数，对该类的对象来说将覆盖全局的 operator new。全局的 operator new 用来给内部类型对象（如 int）、没有定义 operator new 操作符的类的对象分配内存。

new 操作符被映射成 malloc 或者 malloc_dbg，delete 被映射成 free 或者 free_dbg。

10.2 调试手段

MFC 应用程序可以使用 C 运行库的调试手段，也可以使用 MFC 提供的调试手段。两种调试手段分别论述如下。

10.2.1 C 运行库提供和支持的调试功能

C 运行库提供和支持的调试功能如下：

（1） 调试信息报告函数

用来报告应用程序的调试版本运行时的警告和出错信息。包括：

_CrtDbgReport 用来报告调试信息；

_CrtSetReportMode 设置是否警告、出错或者断言信息；

_CrtSetReportFile 设置是否把调试信息写入到一个文件。

（2） 条件验证或者断言宏：

断言宏主要有：

assert 检验某个条件是否满足，不满足终止程序执行。

验证函数主要有：

_CrtIsValidHeapPointer 验证某个指针是否在本地堆中；

_CrtIsValidPointer 验证指定范围的内存是否可以读写；

_CrtIsMemoryBlock 验证某个内存块是否在本地堆中。

（3） 内存（堆）调试：

malloc_dbg 分配内存时保存有关内存分配的信息，如在什么文件、哪一行分配的内存等。有一系列用来提供内存诊断的函数：

_CrtMemCheckpoint 保存内存快照在一个_CrtMemState 结构中；

_CrtMemDifference 比较两个_CrtMemState；

_CrtMemDumpStatistics 转储输出一_CrtMemState 结构的内容；

_CrtMemDumpAllObjectsSince 输出上次快照或程序开始执行以来在堆中分配的所有对象的信息；

_CrtDumpMemoryLeaks 检测程序执行以来的内存漏洞，如果有漏洞则输出所有分配的对象。

10.2.2 MFC 提供的调试手段

MFC 在 C 运行库提供和支持的调试功能基础上，设计了一些类、函数等来协助调试。

（1） MFC 的 TRACE、ASSERT

ASSERT

使用 ASSERT 断言判定程序是否可以继续执行。

TRACE

使用 TRACE 宏显示或者打印调试信息。TRACE 是通过函数 AfxTrace 实现的。由于 AfxTrace 函数使用了 cdecl 调用约定，故可以接受个数不定的参数，如同 printf 函数一样。它的定义和实现如下：

```
void AFX_CDECL AfxTrace(LPCTSTR lpszFormat, ...)
{
#ifdef _DEBUG // all AfxTrace output is controlled by afxTraceEnabled
    if (!afxTraceEnabled)
        return;
#endif

    //处理个数不定的参数
    va_list args;
    va_start(args, lpszFormat);

    int nBuf;
    TCHAR szBuffer[512];

    nBuf = _vstprintf(szBuffer, lpszFormat, args);
    ASSERT(nBuf < _countof(szBuffer));

    if ((afxTraceFlags & traceMultiApp) && (AfxGetApp() != NULL))
        afxDump << AfxGetApp()->m_pszExeName << ": ";
    afxDump << szBuffer;

    va_end(args);
}
#endif //_DEBUG
```

在程序源码中，可以控制是否显示跟踪信息，显示什么跟踪信息。如果全局变量 afxTraceEnabled 为 TRUE，则 TRACE 宏可以输出；否则，没有 TRACE 信息被输出。如果通过 afxTraceFlags 指定了跟踪什么消息，则输出有关跟踪信息，例如为了指定“Multilple Application Debug”，令 AfxTraceFlags|=traceMultiApp。可以跟踪的信息有：

```
enum AfxTraceFlags
{
    traceMultiApp = 1,          // multi-app debugging
    traceAppMsg = 2,            // main message pump trace (includes DDE)
    traceWinMsg = 4,            // Windows message tracing
    traceCmdRouting = 8,        // Windows command routing trace
                                    //(set 4+8 for control notifications)
    traceOle = 16,              // special OLE callback trace
    traceDatabase = 32,         // special database trace
    traceInternet = 64          // special Internet client trace
```

```
};
```

这样，应用程序可以在需要的地方指定 afxTraceEnabled 的值打开或者关闭 TRACE 开关，指定 AfxTraceFlags 的值过滤跟踪信息。

Visual C++提供了一个 TRACE 工具，也可以用来完成上述功能。

为了显示消息信息，MFC 内部定义了一个 AFX_MAP_MESSAG 类型的数组 allMessages，储存了 Windows 消息和消息名映射对。例如：

allMessages[1].nMsg = WM_CREATE,

allMessages[1].lpszMsg = “WM_CREATE”

MFC 内部还使用函数_AfxTraceMsg 显示跟踪消息，它可以接收一个字符串和一个 MSG 指针，然后，把该字符串和 MSG 的各个域的信息组合成一个大的字符串并使用 AfxTrace 显示出来。

allMessages 和函数_AfxTraceMsg 的详细实现可以参见 AfxTrace.cpp。

（2） MFC 对象内容转储

对象内容转储是 CObject 类提供的功能，所有从它派生的类都可以通过覆盖虚拟函数 DUMP 来支持该功能。在讲述 CObject 类时曾提到过。

虚拟函数 Dump 的定义：

```
class ClassName : public CObject
{
public:
#ifdef _DEBUG
    virtual void Dump( CDumpContext& dc ) const;
#endif
…
};
```

在使用 Dump 时，必须给它提供一个 CDumpContext 类型的参数，该参数指定的对象将负责输出调试信息。为此，MFC 提供了一个预定义的全局 CDumpContext 对象 afxDump，它把调试信息输送给调试器的调试窗口。从前面 AfxTrace 的实现可以知道，MFC 使用了 afxDump 输出跟踪信息到调试窗口。

CDumpContext 类没有基类，它提供了以文本形式输出诊断信息的功能。

例如：

```
CPerson* pMyPerson = new CPerson;
// set some fields of the CPerson object...
//...
// now dump the contents
#ifdef _DEBUG
    pMyPerson->Dump( afxDump );
#endif
```

（3） MFC 对象有效性检测

对象有效性检测是 CObject 类提供的功能，所有从它派生的类都可以通过覆盖虚拟函数 AssertValid 来支持该功能。在讲述 CObject 类时曾提到过。

虚拟函数 AssertValid 的定义：

```
class ClassName : public CObject
{
```

```
public:
#ifdef _DEBUG
    virtual void AssertValid(    ) const;
#endif
…
};
```

使用 ASSERT_VALID 宏判断一个对象是否有效，该对象的类必须覆盖了 AssertValid 函数。形式为：ASSERT_VALID(pObject)。

另外，MFC 提供了一些函数来判断地址是否有效，如：

AfxIsMemoryBlock，AfxIsString，AfxIsValidAddress。

10.2.3 内存诊断

MFC 使用 DEBUG_NEW 来跟踪内存分配时的执行的源码文件和行数。

把#define new DEBUG_NEW 插入到每一个源文件中，这样，调试版本就使用_malloc_dbg 来分配内存。MFC Appwizard 在创建框架文件时已经作了这样的处理。

（1） AfxDoForAllObjects

MFC 提供了函数 AfxDoForAllObjects 来追踪动态分配的内存对象，函数原型如下：

```
void AfxDoForAllObjects( void (*pfn)(CObject* pObject,
    void* pContext), void* pContext );
```

其中：

参数 1 是一个函数指针，AfxDoForAllObjects 对每个对象调用该指针表示的函数。

参数 2 将传递给参数 1 指定的函数。

AfxDoForAllObjects 可以检测到所有使用 new 分配的 CObject 对象或者 CObject 类派生的对象，但全局对象、嵌入对象和栈中分配的对象除外。

（2） 内存漏洞检测

仅仅用于 new 的 DEBUG 版本分配的内存。

完成内存漏洞检测，需要如下系列步骤：

- 调用 AfxEnableMemoryTracking(TRUE/FALSE)打开/关闭内存诊断。在调试版本下，缺省是打开的；关闭内存诊断可以加快程序执行速度，减少诊断输出。
- 使用 MFC 全局变量 afxMemDF 更精确地指定诊断输出的特征，缺省值是 allocMemDF，可以取如下值或者这些值相或：

  afxMemDF，delayFreeMemDF，checkAlwaysMemDF

  其中：allocMemDF 表示可以进行内存诊断输出；delayFreeMemDF 表示是否是在应用程序结束时才调用 free 或者 delete，这样导致程序最大可能的分配内存；checkAlwaysMemDF 表示每一次分配或者释放内存之后都调用函数 AfxCheckMemory 进行内存检测（AfxCheckMemory 检查堆中所有通过 new 分配的内存（不含 malloc））。

  这一步是可选步骤，非必须。
- 创建一个 CMemState 类型的变量 oldMemState，调用 CMemState 的成员函数 CheckPoint 获得初次内存快照。
- 执行了系列内存分配或者释放之后，创建另一个 CMemState 类型变量 newMemState，调用 CMemState 的成员函数 CheckPoint 获得新的内存快照。
- 创建第三个 CMemState 类型变量 difMemState，调用 CMemState 的成员函数 Difference 比较 oldMemState 和 newMemState，结果保存在变量 difMemState 中。如果没有不同，则返

回 FALSE，否则返回 TRUE。

- 如果不同，则调用成员函数 DumpStatistics 输出比较结果。

例如：

```
// Declare the variables needed
#ifdef _DEBUG
    CMemoryState oldMemState, newMemState, diffMemState;
    oldMemState.Checkpoint();
#endif

// do your memory allocations and deallocations...
CString s = "This is a frame variable";
// the next object is a heap object
CPerson* p = new CPerson( "Smith", "Alan", "581-0215" );

#ifdef _DEBUG
    newMemState.Checkpoint();
    if( diffMemState.Difference( oldMemState, newMemState ) )
    {
        TRACE( "Memory leaked!\n" );
        diffMemState.DumpStatistics();
        //or diffMemState.DumpAllObjectsSince();
    }
#endif
```

MFC在应用程序（调试版）结束时，自动进行内存漏洞检测，如果存在漏洞，则输出漏洞的有关信息。

第11章　MFC 下的文件类

11.1　文件操作的方法

使用 Visual C++编程，有如下方法进行文件操作：

（1）使用标准 C 运行库函数，包括 fopen、fclose、fseek 等。

（2）使用 Win16 下的文件和目录操作函数，如 lopen、lclose、lseek 等。不过，在 Win32 下，这些函数主要是为了和 Win16 向后兼容。

（3）使用 Win32 下的文件和目录操作函数，如 CreateFile，CopyFile，DeleteFile，FindNextFile，等等。

Win32 下，打开和创建文件都由 CreateFile 完成，成功的话，得到一个 Win32 下的句柄，这不同于“C”的 fopen 返回的句柄。在 Win16 下，该句柄和 C 运行库文件操作函数相容。但在 Win32 下，“C”的文件操作函数不能使用该句柄，如果需要的话，可以使用函数 _open_osfhandle 从 Win32 句柄得到一个“C”文件函数可以使用的文件句柄。

关闭文件使用 Win32 的 CloseHandle。

在 Win32 下，CreateFile 可以操作的对象除了磁盘文件外，还包括设备文件如通讯端口、管道、控制台输入、邮件槽等等。

（4）使用 CFile 和其派生类进行文件操作。CFile 从 CObject 派生，其派生类包括操作文本文件的 CStdioFile，操作内存文件的 CmemFile，等等。

CFile 是建立在 Win32 的文件操作体系的基础上，它封装了部分 Win32 文件操作函数。

最好是使用 CFile 类（或派生类）的对象来操作文件，必要的话，可以从这些类派生自己的文件操作类。统一使用 CFile 的界面可以得到好的移植性。

11.2　MFC 的文件类

MFC 用一些类来封装文件访问的 Win32 API。以 CFile 为基础，从 CFile 派生出几个类，如 CStdioFile，CMemFile，MFC 内部使用的 CMiororFile，等等。

11.2.1 CFile 的结构

11.2.1.1　CFile 定义的枚举类型

CFile 类定义了一些和文件操作相关的枚举类型，主要有四种：OpenFlags，Attribute，SeekPosition，hFileNull。下面，分别解释这些枚举类型。

（1） OpenFlags

OpenFlags 定义了 13 种文件访问和共享模式：

```
enum OpenFlags {
//第一（从右，下同）至第二位，打开文件时访问模式，读/写/读写
modeRead =                0x0000,
modeWrite =               0x0001,
modeReadWrite =           0x0002,
shareCompat =             0x0000, //32 位 MFC 中没用
//第五到第七位，打开文件时的共享模式
shareExclusive =          0x0010,//独占方式，禁止其他进程读写
shareDenyWrite =          0x0020,//禁止其他进程写
shareDenyRead =           0x0030,//禁止其他进程读
shareDenyNone =           0x0040,//允许其他进程写
//第八位，打开文件时的文件继承方式
modeNoInherit =           0x0080,//不允许子进程继承
//第十三、十四位，是否创建新文件和创建方式
modeCreate =              0x1000,//创建新文件，文件长度 0
modeNoTruncate = 0x2000,//创建新文件时如文件已存在则打开
//第十五、十六位，文件以二进制或者文本方式打开，在派生类 CStdioFile 中用
typeText =                0x4000,
typeBinary =      (int)0x8000
};
```

（2） Attribute

Attribute 定义了文件属性：正常、只读、隐含、系统文件，文件或者目录等。

```
enum Attribute {
normal =        0x00,
readOnly =    0x01,
hidden =        0x02,
system =        0x04,
volume =        0x08,
directory =     0x10,
archive =       0x20
}
```

（3） SeekPosition

SeekPosition 定义了三种文件位置：头、尾、当前：

```
enum SeekPosition{
begin = 0x0,
current = 0x1,
end = 0x2
};
```

（4） hFileNull

hFileNull 定义了空文件句柄

```
enum { hFileNull = -1 };
```

11.2.1.2 CFile 的其他一些成员变量

CFile 除了定义枚举类型，还定义了一些成员变量。例如：

```
UINT m_hFile
```

该成员变量是 public 访问属性，保存::CreateFile 返回的操作系统的文件句柄。MFC 重载了运算符号 HFILE 来返回 m_hFile，这样在使用 HFILE 类型变量的地方可以使用 CFile 对象。

```
BOOL m_bCloseOnDelete;
CString m_strFileName;
```

这两个成员变量是 protected 访问属性。m_bCloseOnDelete 用来指示是否在关闭文件时删除 CFile 对象；m_strFileName 用来保存文件名。

11.2.1.3 CFile 的成员函数

CFile 的成员函数实现了对 Win32 文件操作函数的封装，完成以下动作：打开、创建、关闭文件，文件指针定位，文件的锁定与解锁，文件状态的读取和修改，等等。其中，用到了 m_hFile 文件句柄的一般是虚拟函数，和此无关的一般是静态成员函数。一般地，成员函数被映射到对应的 Win32 函数，如表 11-1 所示。

表11-1　CFile函数对Win32文件函数的封装

虚拟	静态	成员函数	对应的 Win32 函数
文件的创建、打开、关闭			
√		Abort	CloseHandle
√		Duplicate	DuplicateHandle
√		Open	CreateFile
√		Close	CloseHandle
文件的读写			
√		Read	ReadFile
		ReadHuge（向后兼容）	调用 Read 成员函数
√		Write	WriteFile
		WriteHuage(向后兼容)	调用 Write 成员函数
√		Flush	FlushFileBuffers
文件定位			
√		Seek	SetFilePointer
		SeekToBegin	调用 Seek 成员函数
		SeekToEnd	调用 Seek 成员函数
√		GetLength	调用 Seek 成员函数
√		SetLength	SetEndOfFile
文件的锁定/解锁			
√		LockRange	LockFile
√		UnlockRange	UnlockFile

文件状态操作函数			
√		GetPosition	SetFilePointer
		GetStatus(CFileStatus&)	GetFileTime,GetFileSize 等
	√	GetStatus(LPSTR lpszFileName CFileStatus&)	FindFirstFile
√		GetFileName	不是简单地映射到某个函数
√		GetFileTitle	
√		GetFilePath	
√		SetFilePath	
	√	SetStatus	
改名和删除			
	√	Rename	MoveFile
	√	Remove	DeleteFile

11.2.2 CFile 的部分实现

这里主要讨论 CFile 对象的构造函数和文件的打开/创建的过程。

（1） 构造函数

CFile 有如下几个构造函数：

- CFile()

缺省构造函数，仅仅构造一个 CFile 对象，还必须使用 Open 成员函数来打开文件。

- CFile(int hFile)

已经打开了一个文件 hFile，在此基础上构造一个 CFile 对象来给它打包。HFile 将被赋值给 CFile 的成员变量 m_hFile。

- CFile(LPCTSTR lpszFileName, UINT nOpenFlags)

指定一个文件名和文件打开方式，构造 CFile 对象，调用 Open 打开/创建文件，把文件句柄保存到 m_hFile。

（2） 打开/创建文件

Open 的原型如下：

```
BOOL CFile::Open(LPCTSTR lpszFileName, UINT nOpenFlags,
    CFileException* pException)
```

Open 调用 Win32 函数::CreateFile 打开文件，并把文件句柄保存到成员变量 m_hFile 中。CreateFile 函数的原型如下：

```
HANDLE CreateFile(
LPCTSTR lpFileName,// pointer to name of the file
DWORD dwDesiredAccess,// access (read-write) mode
DWORD dwShareMode,// share mode
LPSECURITY_ATTRIBUTES lpSecurityAttributes, //pointer to security descriptor
DWORD dwCreationDistribution,// how to create
DWORD dwFlagsAndAttributes,// file attributes
HANDLE hTemplateFile// handle to file with attributes to copy
);
```

显然，Open 必须把自己的两个参数 lpszFileName 和 nOpenFlags 映射到 CreateFile 的七个参

数上。
从 OpenFlags 的定义可以看出，(nOpenFlags & 3)表示了读写标识，映射成变量 dwAccess，可以取值为 Win32 的 GENERIC_READ、GENERIC_WRITE、GENERIC_READ|GENERIC_WRITE。
(nOpenFlags & 0x70)表示了共享模式，映射成变量 dwShareMode，可以取值为 Win32 的 FILE_SHARE_READ、FILE_SHARE_WRITE、FILE_SHARE_WRITE|FILE_SHARE_READ。
Open 定义了一个局部的 SECURITY_ATTRIBUTES 变量 sa，(nOpenFlags & 0x80)被赋值给 sa.bInheritHandle。
(nOpenFlags & modeCreate)表示了创建方式，映射成变量 dwCreateFlag，可以取值为 Win32 的 OPEN_ALWAYS、CREATE_ALWAYS、OPEN_EXISTING。
在生成了上述参数之后，先调用::CreateFile：

```
HANDLE hFile =::CreateFile(lpszFileName,
    dwAccess, dwShareMode, &sa,
    dwCreateFlag, FILE_ATTRIBUTE_NORMAL, NULL);
```

然后，hFile 被赋值给成员变量 m_hFile，m_bCloseOnDelete 被设置为 TRUE。

由上可以看出，CFile 打开(创建)一个文件时大大简化了:: CreateFile 函数的复杂性，即只需要指定一个文件名、一个打开文件的参数即可。若该参数指定为 0，则表示以只读方式打开一个存在的文件，独占使用，不允许子进程继承。
在 CFile 对象使用时，如果它是在堆中分配的，则应该销毁它；如果在栈中分配的，则 CFile 对象将被自动销毁。销毁时析构函数被调用，析构函数是虚拟函数。若 m_bCloseOnDelete 为真且 m_hFile 非空，则析构函数调用 Close 关闭文件。
至于其他 CFile 成员函数的实现，这里不作分析了。

11.2.3 CFile 的派生类

这里主要简要地介绍 CStdioFile 和 CmemFile 及 CFileFind。
（1） CStdioFile
CStdioFile 对文本文件进行操作。
CStdioFile 定义了新的成员变量 m_pStream，类型是 FILE*。在打开或者创建文件时，使用_open_osfhandle 从 m_hFile(Win32 文件句柄)得到一个“C”的 FILE 类型的文件指针，然后，在文件操作中，使用“C”的文件操作函数。例如，读文件使用_fread，而不是::ReadFile，写文件使用了_fwrite，而不是::WriteFile，等等。m_hFile 是 CFile 的成员变量。
另外，CStdioFile 不支持 CFile 的 Dumplicate、LockRange、UnlockRange 操作，但是实现了两个新的操作 ReadString 和 WriteString。
（2） CMemFile
CMemFile 把一块内存当作一个文件来操作，所以，它没有打开文件的操作，而是设计了 Attach 和 Detach 用来分配或者释放一块内存。相应地，它提供了 Alloc、Free 虚拟函数来操作内存文件，它覆盖了 Read、Write 来读写内存文件。
（3） CFileFind
为了方便文件查找，MFC 把有关功能归结成为一个类 CFileFind。CFileFind 派生于 CObject 类。首先，它使用 FindFile 和 FineNextFile 包装了 Win32 函数::FindFirstFile 和::FindNextFile；其次，它提供了许多函数用来获取文件的状态或者属性。
使用 CFileStatus 结构来描述文件的属性，其定义如下：

```
struct CFileStatus
{
CTime m_ctime;              // 文件创建时间
CTime m_mtime;              // 文件最近一次修改时间
CTime m_atime;              // 文件最近一次访问时间
LONG m_size;                // 文件大小
BYTE m_attribute;           // 文件属性
BYTE _m_padding;            // 没有实际含义，用来增加一个字节
TCHAR m_szFullName[_MAX_PATH]; //绝对路径

#ifdef _DEBUG
    //实现 Dump 虚拟函数，输出文件属性
    void Dump(CDumpContext& dc) const;
#endif
};
```

例如：

```
CFileStatus status;
pFile->GetStatus(status);
#ifdef _DEBUG
    status.dump(afxDump);
#endif
```

第12章　对话框和对话框类 CDialog

对话框经常被使用，因为对话框可以从模板创建，而对话框模板是可以使用资源编辑器方便地进行编辑的。

12.1　模式和无模式对话框

对话框分两种类型，模式对话框和无模式对话框。

12.1.1 模式对话框

一个模式对话框是一个有系统菜单、标题栏、边线等的弹出式窗口。在创建对话框时指定 WS_POPUP, WS_SYSMENU, WS_CAPTION 和 DS_MODALFRAME 风格。即使没有指定 WS_VISIBLE 风格，模式对话框也会被显示。
创建对话框窗口时，将发送 WM_INITDIALOG 消息（如果指定对话框的 DS_SETFONT 风格，还有 WM_SETFONT 消息）给对话框过程。
对话框过程(Dialog box procedure)不是对话框窗口的窗口过程(Window procedure)。在 Win32 里，对话框的窗口过程由 Windows 系统提供，用户在创建对话框窗口时提供一个对话框过程由窗口过程调用。
对话框窗口被创建之后，Windows 使得它成为一个激活的窗口，它保持激活直到对话框过程调用::EndDialog 函数结束对话框的运行或者 Windows 激活另一个应用程序为止，在激活时，用户或者应用程序不可以激活它的所属窗口（Owner window）。
从某个窗口创建一个模式对话框时，Windows 自动地禁止使用（Disable）这个窗口和它的所有子窗口，直到该模式对话框被关闭和销毁。虽然对话框过程可以 Enable 所属窗口，但是这样做就失去了模式对话框的作用，所以不鼓励这样做。
Windows 创建模式对话框时，给当前捕获鼠标输入的窗口（如果有的话）发送消息 WM_CANCLEMODE。收到该消息后，应用程序应该终止鼠标捕获(Release the mouse capture)以便于用户能把鼠标移到模式对话框；否则由于 Owner 窗口被禁止，程序将失去鼠标输入。
为了处理模式对话框的消息，Windows 开始对话框自身的消息循环，暂时控制整个应用程序的消息队列。如果 Windows 收到一个非对话框消息时，则它把消息派发给适当的窗口处理；如果收到了 WM_QUIT 消息，则把该消息放回应用程序的消息队列里，这样应用程序的主消息循环最终能处理这个消息。
当应用程序的消息队列为空时，Windows 发送 WM_ENTERIDLE 消息给 Owner 窗口。在对话框运行时，程序可以使用这个消息进行后台处理，当然应该注意经常让出控制给模式对话框，以便它能接收用户输入。如果不希望模式对话框发送 WM_ENTERIDlE 消息，则在创建模式对话框时指定 DS_NOIDLEMSG 风格。
一个应用程序通过调用::EndDialog 函数来销毁一个模式对话框。一般情况下，当用户从系统菜单里选择了关闭（Close）命令或者按下了确认（OK）或取消（CANCLE）按钮，::EndDialog 被对话框过程所调用。调用::EndDialog 时，指定其参数 nResult 的值，Windows 将在销毁对话框窗口后返回这个值，一般，程序通过返回值判断对话框窗口是否完成了任务或者被用户取消。

12.1.2 无模式对话框

一个无模式对话框是一个有系统菜单、标题栏、边线等的弹出式窗口。在创建对话框模板时指定 WS_POPUP、WS_CAPTION、WS_BORDER 和 WS_SYSMENU 风格。如果没有指定 WS_VISIBLE 风格，无模式对话框不会自动地显示出来。
一个无模式对话框既不会禁止所属窗口，也不会给它发送消息。当创建一个模式对话框时，Windows 使它成为活动窗口，但用户或者程序可以随时改变和设置活动窗口。如果对话框失去激活，那么即使所属窗口是活动的，在 Z 轴顺序上，它仍然在所属窗口之上。
应用程序负责获取和派发输入消息给对话框。大部分应用程序使用主消息循环来处理，但是为了用户可以使用键盘在控制窗口之间移动或者选择控制窗口，应用程序应该调用::IsDialogMessage 函数。
这里，顺便解释::IsDialogMessage 函数。虽然该函数是为无模式对话框设计的，但是任何包含了控制子窗口的窗口都可以调用它，用来实现类似于对话框的键盘选择操作。
当::IsDialogMessage 处理一个消息时，它检查键盘消息并把它们转换成相应对话框的选择命令。例如，当 Tab 键被压下时，下一个或下一组控制被选中，当 Down Arrow 键按下后，一组控制中的下一个控制被选择。
::IsDialogMessage 完成了所有必要的消息转换和消息派发，所以该函数处理的消息一定不要传递给 TranslateMessage 和 DispatchMessage 处理。
一个无模式对话框不能像模式对话框那样返回一个值给应用程序。但是对话框过程可以使用::SendMessage 给所属窗口传递信息。
在应用程序结束之前，它必须销毁所有的无模式对话框。使用::DestroyWindow 销毁一个无模式对话框，不是使用::EndDiaLog。一般来说，对话框过程响应用户输入，如用户选择了“取消”按钮，则调用::DestroyWindow；如果用户没有有关动作，则应用程序必须调用::DestroyWindow。

12.2 对话框的 MFC 实现

在 MFC 中，对话框窗口的功能主要由 CWnd 和 CDialog 两个类实现。

12.2.1 CDialog 的设计和实现

MFC 通过 CDialog 来封装对话框的功能。CDialog 从 CWnd 继承了窗口类的功能（包括 CWnd 实现的有关功能），并添加了新的成员变量和函数来处理对话框。

12.2.1.1 CDialog 的成员变量

CDialog 的成员变量有：

```
protected:
UINT m_nIDHelp; // Help ID (0 for none, see HID_BASE_RESOURCE)

LPCTSTR m_lpszTemplateName;    // name or MAKEINTRESOURCE
```

```
HGLOBAL m_hDialogTemplate; // indirect (m_lpDialogTemplate == NULL)
// indirect if (m_lpszTemplateName == NULL)
LPCDLGTEMPLATE m_lpDialogTemplate;
void* m_lpDialogInit;                // DLGINIT resource data
CWnd* m_pParentWnd;                  // parent/owner window
HWND m_hWndTop;          // top level parent window (may be disabled)
```

成员变量保存了创建对话框的模板资源、对话框父窗口对象、顶层窗口句柄等信息。三个关于模板资源的成员变量 m_lpszTemplateName、m_hDialogTemplate、m_lpDialogTemplate 对应了三种模板资源，但在创建对话框时，只要一个模板资源就可以了，可以使用其中的任意一类。

12.2.1.2 CDialog 的成员函数：

（1） 构造函数：

```
CDialog( LPCTSTR lpszTemplateName, CWnd* pParentWnd = NULL );
CDialog( UINT nIDTemplate, CWnd* pParentWnd = NULL );
CDialog( );
```

CDialog 重载了三个构造函数。其中，第三个是缺省构造函数；第一个和第二个构造函数从指定的对话框模板资源创建，pParentWnd 指定了父窗口或所属窗口，若空则设置父窗口为应用程序主窗口。

（2） 初始化函数

```
BOOL Create( LPCTSTR lpszTemplateName, CWnd* pParentWnd = NULL );
BOOL Create( UINT nIDTemplate, CWnd* pParentWnd = NULL );
BOOL CreateIndirect( LPCDLGTEMPLATE lpDialogTemplate, CWnd* pParentWnd = NULL );
BOOL CreateIndirect( HGLOBAL hDialogTemplate, CWnd* pParentWnd = NULL );
BOOL InitModalIndirect( LPCDLGTEMPLATE lpDialogTemplate, CWnd* pParentWnd = NULL );
BOOL InitModalIndirect( HGLOBAL hDialogTemplate, CWnd* pParentWnd = NULL );
```

Create用来根据模板创建无模式对话框；CreateInDirect用来根据内存中的模板创建无模式对话框；InitModalIndirect用来根据内存中的模板创建模式对话框。它们都提供了两个重载版本。

（3） 对话框操作函数

```
void MapDialogRect( LPRECT lpRect ) const;
void NextDlgCtrl( ) const;
void PrevDlgCtrl( ) const;
void GotoDlgCtrl( CWnd* pWndCtrl );
void SetDefID( UINT nID );
void SetHelpID( UINT nIDR );
void EndDialog( int nResult );
```

（4） 虚拟函数

```
virtual int DoModal( );
virtual    BOOL OnInitDialog( );
virtual    void OnSetFont( CFont* pFont );
virtual void OnOK( );
virtual void OnCancel( );
```

12.2.2 MFC 模式对话框的实现

从前面的介绍可以知道，Win32 SDK编程下的模式对话框使用了Windows提供给对话框窗口的窗口过程和自己的对话框过程，对话框过程将被窗口过程调用。但在MFC下，所有的窗口类都使用了同一个窗口过程，CDialog也不例外。CDialog对象在创建Windows对话框时，采用了类似于CWnd的创建函数过程，采用子类化的手段将Windows提供给对话框的窗口过程取代为AfxWndProc或者AfxBaseWndProc，同时提供了对话框过程AfxDlgProc。那么，这些“过程”是如何实现或者协调的呢？下文将予以分析。

12.2.2.1 MFC 对话框过程

MFC对话框过程AfxDlgProc的原型和实现如下：

```
BOOL CALLBACK AfxDlgProc(HWND hWnd,
UINT message, PARAM, LPARAM)
{
if (message == WM_INITDIALOG)
{
    //处理 WM_INITDIALOG 消息
    CDialog* pDlg = DYNAMIC_DOWNCAST(CDialog,
    CWnd::FromHandlePermanent(hWnd));
        if (pDlg != NULL)
            return pDlg->OnInitDialog();
        else
            return 1;
}
return 0;
}
```

由上可以看出，MFC的对话框函数AfxDlgProc仅处理消息WM_INITDIALOG，其他都留给对话框窗口过程处理。因此，它不同于SDK编程的对话框过程。程序员在SDK的对话框过程处理消息和事件，实现自己的对话框功能。
AfxDlgProc处理WM_INITDIALOG消息时调用虚拟函数OnInitDialog，给程序员一个机会处理对话框的初始化。

12.2.2.2 模式对话框窗口过程

本小节讨论对话框的窗口过程。
AfxWndProc是所有的MFC窗口类使用的窗口过程，它取代了模式对话框原来的窗口过程（Windows提供），那么，MFC如何完成Win32下对话框窗口的功能呢？
考查模式对话框的创建过程。CDialog::DoModal用来创建模式对话框窗口并执行有关任务，和DoModal相关的是MFC内部使用的成员函数CDialog::PreModal和CDialog::PostModal。下面分别讨论它们的实现。

```
HWND CDialog::PreModal()
{
// cannot call DoModal on a dialog already constructed as modeless
ASSERT(m_hWnd == NULL);

// allow OLE servers to disable themselves
AfxGetApp()->EnableModeless(FALSE);

// 得到父窗口
CWnd* pWnd = CWnd::GetSafeOwner(m_pParentWnd, &m_hWndTop);

// 如同 CWnd 处理其他窗口的创建，设置一个窗口创建 HOOK
AfxHookWindowCreate(this);

//返回父窗口的句柄
return pWnd->GetSafeHwnd();
}

void CDialog::PostModal()
{
//取消窗口创建前链接的 HOOK
AfxUnhookWindowCreate();      // just in case
//MFC 对话框对象和对应的 Windows 对话框窗口分离
Detach();                     // just in case

// m_hWndTop 是当前对话框的父窗口或所属窗口，则恢复它
if (::IsWindow(m_hWndTop))
        ::EnableWindow(m_hWndTop, TRUE);
m_hWndTop = NULL;

AfxGetApp()->EnableModeless(TRUE);
}

int CDialog::DoModal()
{
// can be constructed with a resource template or InitModalIndirect
ASSERT(m_lpszTemplateName != NULL ||
        m_hDialogTemplate != NULL || m_lpDialogTemplate != NULL);

//加载对话框资源
LPCDLGTEMPLATE lpDialogTemplate = m_lpDialogTemplate;
HGLOBAL hDialogTemplate = m_hDialogTemplate;
HINSTANCE hInst = AfxGetResourceHandle();
//查找资源（见 9.5.2 节），找到了就加载它
```

```
if (m_lpszTemplateName != NULL)
{
	hInst = AfxFindResourceHandle(m_lpszTemplateName, RT_DIALOG);
	HRSRC hResource =
		::FindResource(hInst, m_lpszTemplateName, RT_DIALOG);
	hDialogTemplate = LoadResource(hInst, hResource);
}

//锁定加载的资源
if (hDialogTemplate != NULL)
	lpDialogTemplate = (LPCDLGTEMPLATE)LockResource(hDialogTemplate);

// return -1 in case of failure to load the dialog template resource
if (lpDialogTemplate == NULL)
	return -1;

//创建对话框前禁止父窗口，为此要调用 PreModal 得到父窗口句柄
HWND hWndParent = PreModal();
AfxUnhookWindowCreate();
CWnd* pParentWnd = CWnd::FromHandle(hWndParent);
BOOL bEnableParent = FALSE;
if (hWndParent != NULL && ::IsWindowEnabled(hWndParent))
{
	::EnableWindow(hWndParent, FALSE);
	bEnableParent = TRUE;
}

//创建对话框，注意是无模式对话框
TRY
{
//链接一个 HOOK 到 HOOK 链以处理窗口创建，
//如同 4.4.1 节描述的 CWnd 类窗口创建一样
AfxHookWindowCreate(this);
//CreateDlgIndirect 间接调用::CreateDlgIndirect，
//最终调用了::CreateWindowEX 来创建对话框窗口。
//HOOK 过程_AfxCbtFilterHook 用子类化的方法
//取代原来的窗口过程为 AfxWndProc。
if (CreateDlgIndirect(lpDialogTemplate, CWnd::FromHandle(hWndParent), hInst))
{
	if (m_nFlags & WF_CONTINUEMODAL)
	{
		// enter modal loop
		DWORD dwFlags = MLF_SHOWONIDLE;
		//RunModalLoop 接管整个应用程序的消息处理
```

```
            if (GetStyle() & DS_NOIDLEMSG)
                dwFlags |= MLF_NOIDLEMSG;
                VERIFY(RunModalLoop(dwFlags) == m_nModalResult);
            }

            // hide the window before enabling the parent, etc.
            if (m_hWnd != NULL)
                SetWindowPos(NULL, 0, 0, 0, 0, SWP_HIDEWINDOW|
                SWP_NOSIZE|SWP_NOMOVE|
                SWP_NOACTIVATE|SWP_NOZORDER);
        }
    }
    CATCH_ALL(e)
    {
        DELETE_EXCEPTION(e);
        m_nModalResult = -1;
    }
    END_CATCH_ALL

    //Enable 并且激活父窗口
    if (bEnableParent)
        ::EnableWindow(hWndParent, TRUE);
    if (hWndParent != NULL && ::GetActiveWindow() == m_hWnd)
        ::SetActiveWindow(hWndParent);

    //::EndDialog 仅仅关闭了窗口，现在销毁窗口
    DestroyWindow();

    PostModal();

    // 必要的话，解锁/释放资源
    if (m_lpszTemplateName != NULL || m_hDialogTemplate != NULL)
        UnlockResource(hDialogTemplate);
    if (m_lpszTemplateName != NULL)
        FreeResource(hDialogTemplate);

    return m_nModalResult;
    }
```

从DoModal的实现可以看出：

它首先Disable对话框窗口的父窗口；然后使用::CreateIndrectDialog创建对话框窗口，使用子类化的方法用AfxWndProc（或者AfxBaseProc）替换了原来的窗口过程，并把原来的窗口过程保存在CWnd的成员变量m_pfnSuper中。原来的窗口过程就是::DialogBox等创建对话框窗口时指定的，是Windows内部提供的对话框“窗口类”的窗口过程。取代(Subclass)原来“窗口类”的窗口过程的方法如同 4.4.1节描述的CWnd::Create。

在::CreateIndirectDialog创建对话框窗口后，会发送WM_INITDIALOG消息给对话框的对话框过程（必要的话，还有WM_SETFONT消息）。但是MFC取代了原来的对话框窗口过程，这两个消息如何送给对话框过程呢？处理方法如下节所描述。

12.2.2.3 使用原对话框窗口过程作消息的缺省处理

对话框的消息处理过程和其他窗口并没有什么不同。这里主要分析的是如何把一些消息传递给对话框原窗口过程处理。下面，通过解释MFC对WM_INITDIALOG消息的处理来解释MFC窗口过程和原对话框窗口过程的关系及其协调作用。

MFC提供了WM_INITDIALOG消息的处理函数CDialog::HandleInitDialog，WM_INITDIALOG消息按照标准Windows的处理送给HandleInitDialog处理。

HandleInitDialog调用缺省处理过程Default，导致CWnd的Default函数被调用。

CWnd::Default的实现如下：

```
LRESULT CWnd::Default()
{
// call DefWindowProc with the last message
_AFX_THREAD_STATE* pThreadState = _afxThreadState.GetData();
return DefWindowProc(pThreadState->m_lastSentMsg.message,
        pThreadState->m_lastSentMsg.wParam,
        pThreadState->m_lastSentMsg.lParam);
}
```

顺便指出，从Default的实现可以看出线程状态的一个用途：它把本线程最新收到和处理的消息记录在成员变量m_lastSentMsg中。

在Default的实现中，CWnd的DefWindowsProc被调用，其实现如下：

```
LRESULT CWnd::DefWindowProc(UINT nMsg,
        WPARAM wParam, LPARAM lParam)
{
        //若“窗口超类（SuperClass）”的窗口过程 m_pfnSuper 非空，则调用它
if (m_pfnSuper != NULL)
        return ::CallWindowProc(m_pfnSuper, m_hWnd, nMsg, wParam, lParam);

//在 MFC 中，GetSuperWndProcAddr 的作用就是返回 m_pfnSuper，为什么还
//要再次调用呢？因为虽然该函数现在是 Obsolete，但原来曾经是有用的。如
//果返回非空，就调用该窗口过程进行处理，否则，由 Windows 进行缺省处理。
WNDPROC pfnWndProc;
if ((pfnWndProc = *GetSuperWndProcAddr()) == NULL)
        return ::DefWindowProc(m_hWnd, nMsg, wParam, lParam);
else
        return   ::CallWindowProc(pfnWndProc, m_hWnd, nMsg, wParam, lParam);
}
```

综合上述分析，HandleInitDialog 最终调用了窗口过程 m_pfnSuper，即 Windows 提供给“对话框窗口类”的窗口过程，于是该窗口过程调用了对话框过程 AfxDlgProc，导致虚拟函数 OnInitDialog 被调用。

顺便提一下，CWnd::AfxCallWndProc 在处理 WM_INITDIALOG 消息之前和之后都会有一些特别的处理，如尝试把对话框放到屏幕中间。具体实现这里略。
OnInitDialog 的 MFC 缺省实现主要完成三件事情：
调用 ExecInitDialog 初始化对话框中的控制；调用 UpdateData 初始化对话框控制中的数据；确定是否显示帮助按钮。所以，程序员覆盖该函数时，一定要调用基类的实现。

MFC 采用子类化的方法取代了对话框的窗口过程，实现了 12.1 节描述的模式对话框窗口的一些特性，原来 SDK 下对话框过程要处理的东西大部分转移给 MFC 窗口过程处理，如处理控制窗口的控制通知消息等。如果不能处理或者必须借助于原来的窗口过程的，则通过缺省处理函数 Default 传递给原来的窗口过程处理，如同这里对 WM_INITDIALOG 的处理一样。

12.2.2.4 Dialog 命令消息和控制通知消息的处理

通过覆盖 CWnd 的命令消息发送函数 OnCmdMsg，CDialog 实现了自己的命令消息发送路径。在 4.4.3.3 节，曾经分析了 CDialog::OnCmdMsg 函数，这里给出其具体实现：

```
BOOL CDialog::OnCmdMsg(UINT nID, int nCode, void* pExtra,
        AFX_CMDHANDLERINFO* pHandlerInfo)
{
//首先，让对话框窗口自己或者基类处理
if (CWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
        return TRUE;
//如果还未处理，且是控制通知消息或者状态更新消息或者系统命令
//则停止进一步的发送
if ((nCode != CN_COMMAND && nCode != CN_UPDATE_COMMAND_UI) ||
                !IS_COMMAND_ID(nID) || nID >= 0xf000)
{
        return FALSE;             // not routed any further
}

//尝试给父窗口处理
CWnd* pOwner = GetParent();
if (pOwner != NULL)
{
#ifdef _DEBUG
        if (afxTraceFlags & traceCmdRouting)
                TRACE1("Routing command id 0x%04X to owner window.\n", nID);
#endif
        ASSERT(pOwner != this);
        if (pOwner->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
                return TRUE;
}

// 最后，给当前线程对象处理
```

```
CWinThread* pThread = AfxGetThread();
if (pThread != NULL)
{
#ifdef _DEBUG
    if (afxTraceFlags & traceCmdRouting)
        TRACE1("Routing command id 0x%04X to app.\n", nID);
#endif
    if (pThread->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
        return TRUE;
}

#ifdef _DEBUG
    if (afxTraceFlags & traceCmdRouting)
    {
        TRACE2("IGNORING command id 0x%04X sent to %hs dialog.\n", nID,
            GetRuntimeClass()->m_lpszClassName);
    }
#endif
return FALSE;
}
```

从上述实现可以看出，CDialog 处理命令消息遵循如下顺序：

对话框自身→父窗口→线程对象

例如，模式对话框产生的 WM_ENTERIDLE 消息就发送给父窗口处理。

从实现中还看到，MFC 根据 TRACE 过滤标识 afxTraceFlags 的值，把有关命令消息的派发显示到调试窗口。

CDialog::OnCmdMsg 不仅适用于模式对话框，也适用于无模式对话框。

12.2.2.5 消息预处理和 Dialog 消息

另外，对话框窗口的消息处理还有一个特点，就是增加了对 Dialog 消息的处理，如同在介绍::IsDialogMessage 函数时所述。如果是 Dialog 消息，MFC 框架将不会让它进入下一步的消息循环。为此，MFC 覆盖了 CDialog 的虚拟函数 PreTranslateMessage，该函数的实现如下：

```
BOOL CDialog::PreTranslateMessage(MSG* pMsg)
{
// 用于无模式或者模式对话框的处理
ASSERT(m_hWnd != NULL);

//过滤 tooltip messages
if (CWnd::PreTranslateMessage(pMsg))
    return TRUE;

//在 Shift+F1 帮助模式下，不转换 Dialog messages
CFrameWnd* pFrameWnd = GetTopLevelFrame();
if (pFrameWnd != NULL && pFrameWnd->m_bHelpMode)
```

```
        return FALSE;

    //处理 Escape 键按下的消息
    if (pMsg->message == WM_KEYDOWN &&
        (pMsg->wParam == VK_ESCAPE || pMsg->wParam == VK_CANCEL) &&
        (::GetWindowLong(pMsg->hwnd, GWL_STYLE) & ES_MULTILINE) &&
        _AfxCompareClassName(pMsg->hwnd, _T("Edit")))
    {
        HWND hItem = ::GetDlgItem(m_hWnd, IDCANCEL);
        if (hItem == NULL || ::IsWindowEnabled(hItem))
        {
            SendMessage(WM_COMMAND, IDCANCEL, 0);
            return TRUE;
        }
    }
    // 过滤来自控制该对话框子窗口的送给该对话框的 Dialog 消息
    return PreTranslateInput(pMsg);
    }
```

从其实现可以看出，如果是 Tooltip 消息或者 Dialog 消息，这些消息将在 PreTranslateMessage 中被处理，不会进入消息发送的处理。

PreTranslateInput 是 CWnd 的成员函数，它调用::IsDialogMessage 函数来处理 Dialog 消息。

PreTranslateMessage 的实现不仅用于模式对话框，而且用于无模式对话框。

12.2.2.6 模式对话框的消息循环

从 DoModal 的实现可以看出，DoModal 调用 CreateDlgIndirect 创建的是无模式对话框，MFC 如何来接管和控制应用程序的消息队列，实现一个模式对话框的功能呢？

CDialog 调用了 RunModalLoop 来实现模式窗口的消息循环。RunModalLoop 是 CWnd 的成员函数，它和相关函数的实现如下：

```
    int CWnd::RunModalLoop(DWORD dwFlags)
    {
    ASSERT(::IsWindow(m_hWnd)); //窗口必须已经创建且不在模式状态  ASSERT(!(m_nFlags  &
WF_MODALLOOP));

    // 以下变量用于 Idle 处理
    BOOL bIdle = TRUE;
    LONG lIdleCount = 0;
    BOOL bShowIdle = (dwFlags & MLF_SHOWONIDLE) &&
            !(GetStyle() & WS_VISIBLE);
    HWND hWndParent = ::GetParent(m_hWnd);
    m_nFlags |= (WF_MODALLOOP|WF_CONTINUEMODAL);
    MSG* pMsg = &AfxGetThread()->m_msgCur;

    //获取和派发消息直到模式状态结束
```

```
for (;;)
{
ASSERT(ContinueModal());

//第一阶段，判断是否可以进行 Idle 处理
while (bIdle &&!::PeekMessage(pMsg, NULL, NULL, NULL, PM_NOREMOVE))
{
ASSERT(ContinueModal());

//必要的话，当 Idle 时显示对话框窗口
if (bShowIdle)
{
        ShowWindow(SW_SHOWNORMAL);
        UpdateWindow();
        bShowIdle = FALSE;
}

//  进行 Idle 处理
//必要的话发送 WM_ENTERIDLE 消息给父窗口
if (!(dwFlags & MLF_NOIDLEMSG) &&hWndParent != NULL && lIdleCount == 0)
{
        ::SendMessage(hWndParent, WM_ENTERIDLE,
        MSGF_DIALOGBOX, (LPARAM)m_hWnd);
}
//必要的话发送 WM_KICKIDLE 消息给父窗口
if ((dwFlags & MLF_NOKICKIDLE) ||
        !SendMessage(WM_KICKIDLE, MSGF_DIALOGBOX, lIdleCount++))
{
        //终止 Idle 处理
        bIdle = FALSE;
}
}

//第二阶段，发送消息
do
{
ASSERT(ContinueModal());

//  若是 WM_QUIT 消息，则发送该消息到消息队列，返回；否则发送消息。
if (!AfxGetThread()->PumpMessage())
{
        AfxPostQuitMessage(0);
        return -1;
}
```

```
//必要的话，显示对话框窗口
if (bShowIdle &&
        (pMsg->message == 0x118 || pMsg->message == WM_SYSKEYDOWN))
{
        ShowWindow(SW_SHOWNORMAL);
        UpdateWindow();
        bShowIdle = FALSE;
}

if (!ContinueModal())
        goto ExitModal;

//在派发了“正常 ”消息后，重新开始 Idle 处理
if (AfxGetThread()->IsIdleMessage(pMsg))
{
        bIdle = TRUE;
        lIdleCount = 0;
}
} while (::PeekMessage(pMsg, NULL, NULL, NULL, PM_NOREMOVE));
}

ExitModal:
m_nFlags &= ~(WF_MODALLOOP|WF_CONTINUEMODAL);
return m_nModalResult;
}

BOOL CWnd::ContinueModal()
{
return m_nFlags & WF_CONTINUEMODAL;
}

void CWnd::EndModalLoop(int nResult)
{
        ASSERT(::IsWindow(m_hWnd));

        // this result will be returned from CWnd::RunModalLoop
        m_nModalResult = nResult;

        // make sure a message goes through to exit the modal loop
        if (m_nFlags & WF_CONTINUEMODAL)
        {
                m_nFlags &= ~WF_CONTINUEMODAL;
                PostMessage(WM_NULL);
```

```
        }
    }
```

和 CWinThread::Run 的处理过程比较，RunModalLoop 也分两个阶段进行处理。不同之处在于，这里不同于 Run 的 Idle 处理，RunModalLoop 是给父窗口发送 WM_ENTERIDLE 消息（如果需要的话）；另外，当前对话框的父窗口被 Disabled，是不接收用户消息的。

RunModalLoop 是一个实现自己的消息循环的示例，消息循环的条件是模式化状态没有结束。实现线程自己的消息循环见 8.5.6 节。

当用户按下按钮“取消”、“确定”时，将导致 RunModalLoop 退出消息循环，结束对话框模式状态，并调用::EndDialog 关闭窗口。有关关闭对话框的处理如下：

```
void CDialog::EndDialog(int nResult)
{
ASSERT(::IsWindow(m_hWnd));

if (m_nFlags & (WF_MODALLOOP|WF_CONTINUEMODAL))
EndModalLoop(nResult);

::EndDialog(m_hWnd, nResult);
}
void CDialog::OnOK()
{
if (!UpdateData(TRUE))      {
      TRACE0("UpdateData failed during dialog termination.\n");
      // the UpdateData routine will set focus to correct item
      return;
}
EndDialog(IDOK);
}

void CDialog::OnCancel()
{
EndDialog(IDCANCEL);
}
```

上述函数 OnOk、OnCancle、EndDialog 都可以用来关闭对话框窗口。其中：

OnOk 首先进行数据交换，获取对话框中各个控制子窗口的数据，然后调用 EndDialog 结束对话框。

OnCancle 直接 EndDialog 结束对话框。

EndDialog 首先修改 m_nFlag 的值，表示结束模式循环，然后调用::EndDialog 关闭对话框窗口。

12.2.3 对话框的数据交换

对话框数据交换指以下两种动作，或者是把内存数据写入对应的控制窗口，或者是从控制窗口读取数据并保存到内存变量中。MFC 为了简化这些操作，以 CDataExchange 类和一些数据交换函数为基础，提供了一套数据交换和校验的机制。

12.2.3.1 数据交换的方法

首先，定义保存数据的内存变量——给对话框添加成员变量，每个控制窗口可以对应一个成员变量，或者是控制窗口类型，或者是控制窗口表示的数据的类型。例如，对于对话框的一个编辑控制窗口，可以定义一个 CEdit 类型的成员变量，或者一个 CString 类型的成员变量。其次，覆盖对话框的虚拟函数 DoDataExchange，实现数据交换和验证。

ClassWizard 可以协助程序员自动地添加成员变量，修改 DoDataExchange。例如，一个对话框有两个控制窗口，其中的一个编辑框表示姓名，ID 是 IDC_NAME，另一个编辑框表示年龄，ID 是 IDC_AGE，ClassWizard 添加如下的成员变量：

```
// Dialog Data
//{{AFX_DATA(CExDialog)
enum { IDD = IDD_DIALOG2 };
CEditm_name;
int          m_iAge;
//}}AFX_DATA
```

使用 ClassWizard 添加成员变量中，一个定义为 CEdit，另一个定义为 int。这些定义被“//{{AFX_DATA”和“//}}AFX_DATA”引用，表示是 ClassWizard 添加的，程序员不必修改它们。

相应的 DoDataExchange 的实现如下：

```
void CExDialog::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CFtpDialog)
DDX_Control(pDX, IDC_NAME, m_name);
DDX_Text(pDX, IDC_AGE, m_nAge);
DDV_MinMaxInt(pDX, m_nAge, 1, 100);
//}}AFX_DATA_MAP
}
```

DDX_ Control 表示把 IDC_NAME 子窗口的内容传输到窗口 m_name，或者相反。

DDX_ Text 表示把 IDC_AGE 子窗口的内容按整数类型保存到 m_nAge，或者相反。

DDV_MinMaxInt 表示 m_nAge 应该在 1 和 100 之间取值。

12.2.3.2 CDataExchange

上文中提到 DDX_Xxxxx 数据交换函数可以进行双向的数据交换，那么它们如何知道数据传输的方向呢？这通过 DDX_Xxxxx 函数的第一个参数 pDX（也就是 DoDataEx change 的参数 pDX）所指的 CDataExchange 对象来决定，pDX 指向一个 CdataExchange 对象。CDataExchange 定义如下：

```
class CDataExchange
{
// Attributes
public:
BOOL m_bSaveAndValidate;    // TRUE 则 保存和验证数据
```

```
CWnd* m_pDlgWnd;                    // 指向一个对话框

// Operations (for implementors of DDX and DDV procs)
HWND PrepareCtrl(int nIDC);         //返回指定 ID 的控制窗口的句柄
HWND PrepareEditCtrl(int nIDC); //返回指定 ID 的编辑控制窗口句柄
void Fail();                        // 用来扔出例外

#ifndef _AFX_NO_OCC_SUPPORT //OLE 控制
    CWnd* PrepareOleCtrl(int nIDC); // 用于对话框中的 OLE 控制窗口
#endif

// Implementation
CDataExchange(CWnd* pDlgWnd, BOOL bSaveAndValidate);

HWND m_hWndLastControl;     // last control used (for validation)
BOOL m_bEditLastControl;    // last control was an edit item
};
```

DoDataExchange 类似于 Serialize 函数，CDataExchange 类似于 CArchive。CDataExchange 使用成员变量 m_pDlgWnd 保存要进行数据交换的对话框，使用成员变量 m_bSaveAndValidate 指示数据传输的方向，如果该变量真，则从控制窗口读取数据到成员变量，如果假，则从成员变量写数据到控制窗口。

在构造一个 CDataExchange 对象时，将保存有关信息在对象的成员变量中。构造函数如下：

```
CDataExchange::CDataExchange(CWnd* pDlgWnd, BOOL bSaveAndValidate)
{
    ASSERT_VALID(pDlgWnd);
    m_bSaveAndValidate = bSaveAndValidate;
    m_pDlgWnd = pDlgWnd;
    m_hWndLastControl = NULL;
}
```

构造函数参数指定了进行数据交换的对话框 pDlgWnd 和数据传输方向 bSaveAndValidate。

12.2.3.3 数据交换和验证函数

在进行数据交换或者验证时，首先使用 PrePareCtrl 或者 PrePareEditCtrl 得到控制窗口的句柄，然后使用::GetWindowsText 从控制窗口读取数据，或者使用::SetWindowsText 写入数据到控制窗口。下面讨论几个例子：

- static void AFX_CDECL DDX_TextWithFormat(CDataExchange* pDX,
    int nIDC,LPCTSTR lpszFormat, UINT nIDPrompt, ...)

```
{
va_list pData; //用来处理个数可以变化的参数
va_start(pData, nIDPrompt);//得到参数

//得到编辑框的句柄
HWND hWndCtrl = pDX->PrepareEditCtrl(nIDC);
```

```
TCHAR szT[32];
if (pDX->m_bSaveAndValidate) //TRUE，从编辑框读出数据
{
    // the following works for %d, %u, %ld, %lu
    //从编辑框得到内容
    ::GetWindowText(hWndCtrl, szT, _countof(szT));
    //转换编辑框内容为指定的格式，支持“ %d, %u, %ld, %lu”
    if (!AfxSimpleScanf(szT, lpszFormat, pData))
    {
        AfxMessageBox(nIDPrompt);
        pDX->Fail();            //数据交换失败
    }
}
else //FALSE，写入数据到编辑框
{
    //把要写的内容转换成指定格式
    wvsprintf(szT, lpszFormat, pData);//不支持浮点运算
    //设置编辑框的内容
    AfxSetWindowText(hWndCtrl, szT);
}

va_end(pData);//结束参数分析
}
```

DDX_TextWithFormat 用来按照一定的格式把数据写入或者读出编辑框。首先，它得到编辑框的句柄 hWndCtrl，然后，根据传输方向从编辑框读出内容并转换成指定格式（读出时），或者转换内容为指定格式后写入编辑框（写入时）。本函数可以处理个数不定的参数，是多个数据交换和验证函数的基础。

- void AFXAPI DDX_Text(CDataExchange* pDX, int nIDC, long& value)

```
{
if (pDX->m_bSaveAndValidate)
    DDX_TextWithFormat(pDX, nIDC, _T("%ld"), AFX_IDP_PARSE_INT, &value);
else
    DDX_TextWithFormat(pDX, nIDC, _T("%ld"), AFX_IDP_PARSE_INT, value);
}
```

上述 DDX_TEXT 用来在编辑框和 long 类型的数据成员之间交换数据。MFC 提供了 DDX_TEXT 的多个重载函数处理编辑框和不同类型的数据成员之间的数据交换。

- void AFXAPI DDX_LBString(CDataExchange* pDX, int nIDC,CString& value)

```
{
//得到列表框句柄
HWND hWndCtrl = pDX->PrepareCtrl(nIDC);
if (pDX->m_bSaveAndValidate)//TRUE，读取数据
{
```

```
        //确定列表框当前被选择的条目
        int nIndex = (int)::SendMessage(hWndCtrl, LB_GETCURSEL, 0, 0L);
        if (nIndex != -1) //列表框有一个条目被选中
        {
            //得到当前条目的长度
            int nLen = (int)::SendMessage(hWndCtrl, LB_GETTEXTLEN, nIndex, 0L);
            //读取当前条目的内容到 value 中
            ::SendMessage(hWndCtrl, LB_GETTEXT, nIndex,
                (LPARAM)(LPVOID)value.GetBufferSetLength(nLen));
        }
        else //当前列表框没有条目被选中
        {
            value.Empty();
        }
        value.ReleaseBuffer();
    }
    else//FALSE，写内容到列表框
    {
        // 把 value 字符串写入当前选中的条目
        if (::SendMessage(hWndCtrl, LB_SELECTSTRING,
            (WPARAM)-1,(LPARAM)(LPCTSTR)value) == LB_ERR)
        {
            // no selection match
            TRACE0("Warning: no listbox item selected.\n");
        }
    }
    }
```

DDX_LBString 用来在列表框和 CString 类型的成员数据之间交换数据。首先，得到列表框的句柄，然后，调用 Win32 的列表框操作函数读取或者修改列表框的内容。

- 下面的 DDX_Control 用于得到一个有效的控制类型窗口对象(MFC 对象)。

```
    void AFXAPI DDX_Control(CDataExchange* pDX, int nIDC, CWnd& rControl)
    {
    if (rControl.m_hWnd == NULL)       // 还没有子类化
    {
        ASSERT(!pDX->m_bSaveAndValidate);
        //得到控制窗口句柄
        HWND hWndCtrl = pDX->PrepareCtrl(nIDC);
        //把 hWndCtrl 窗口和 MFC 窗口对象 rControl 捆绑在一起
        if (!rControl.SubclassWindow(hWndCtrl))
        {
            ASSERT(FALSE);          //不允许两次子类化
            AfxThrowNotSupportedException();
        }
        #ifndef _AFX_NO_OCC_SUPPORT//OLE 控制相关的操作
```

```
        else
        {
            // If the control has reparented itself (e.g., invisible control),
            // make sure that the CWnd gets properly wired to its control site.
            if (pDX->m_pDlgWnd->m_hWnd != ::GetParent(rControl.m_hWnd))
                rControl.AttachControlSite(pDX->m_pDlgWnd);
            }
            #endif //!_AFX_NO_OCC_SUPPORT

        }
    }
```

DDX_Control 用来把控制窗口（Windows 窗口）和一个对话框成员（MFC 窗口对象）捆绑在一起，这个过程是通过 SubclassWindow 函数完成的。这样，程序员就可以通过成员变量来操作控制窗口，读、写、修改控制窗口的内容。

MFC 还提供了许多其他数据交换函数（“DDX_”为前缀）和数据验证函数(“DDV_”为前缀)。DDV 函数和 DDX 函数类似，这里不再多述。
程序员可以创建自己的数据交换和验证函数并使用它们，可以手工加入这些函数到 DoDataExchange 中，如果要 Classwizard 使用这些函数，可以修改 DDX.CLW 文件，在 DDX、DDV 函数入口中加入自己创建的函数。

12.2.3.4 UpdateData 函数

有了数据交换类和数据交换函数，怎么来使用它们呢？MFC 设计了 UpdateData 函数来完成上述数据交换和验证的处理。
首先，UpdateData 创建 CDataExchange 对象，然后调用 DoDataExchange 函数。其实现如下：

```
BOOL CWnd::UpdateData(BOOL bSaveAndValidate)
{
ASSERT(::IsWindow(m_hWnd)); // calling UpdateData before DoModal?

//创建 CDataChange 对象
CDataExchange dx(this, bSaveAndValidate);

//防止在 UpdateData 期间派发通知消息给该窗口
_AFX_THREAD_STATE* pThreadState = AfxGetThreadState();
HWND hWndOldLockout = pThreadState->m_hLockoutNotifyWindow;
ASSERT(hWndOldLockout != m_hWnd);     // must not recurse
pThreadState->m_hLockoutNotifyWindow = m_hWnd;

BOOL bOK = FALSE;             // assume failure
TRY
{
    //数据交换
    DoDataExchange(&dx);
```

```
        bOK = TRUE;             // it worked
    }
    CATCH(CUserException, e)//例外
    {
        // validation failed - user already alerted, fall through
        ASSERT(bOK == FALSE);
        // Note: DELETE_EXCEPTION_(e) not required
    }
    AND_CATCH_ALL(e)
    {
        // validation failed due to OOM or other resource failure
        e->ReportError(MB_ICONEXCLAMATION, FX_IDP_INTERNAL_FAILURE);
        ASSERT(!bOK);
        DELETE_EXCEPTION(e);
    }
    END_CATCH_ALL

    //恢复原来的值
    pThreadState->m_hLockoutNotifyWindow = hWndOldLockout;
    return bOK;
    }
```

UpdataDate 根据参数创建 CDataExchange 对象 dx，如果参数为 TRUE，dx 用来写数据，否则 dx 用来读数据；然后调用 DoDataExchange 进行数据交换。在数据交换期间，为了防止当前窗口接收和处理命令通知消息，在当前线程的线程状态中记录该窗口的句柄，用来防止给该窗口发送通知消息。

使用 MFC 的数据交换和验证机制，大大简化了程序员的工作。通常在 OnInitDialog 中，MFC 调用 UpdateData(FALSE)把数据送给控制窗口显示；在 OnOk 中，调用 UpdateData(TRUE)从控制窗口中读取数据。

12.3 无模式对话框

CFormView是MFC使用无模式对话框的一个典型例子。CFormView是基于对话框模板创建的视，它的直接基类是CSrcollView，CSrcollView的直接基类才是CView。所以，这里先对CScorllView作一个简要的介绍。

12.3.1 CScrollView

CScrollView 继承了 CView 的特性，并且增加了如下的功能：
（1）管理映射模式、窗口尺寸、视口尺寸(Map mode、Window and Viewport size)。Window and Viewport size 用来完成页面空间到设备空间的转换。
（2）自动管理滚动条，响应滚动条消息。
为了实现这些功能，CScrollView 覆盖 CView 或者 CWnd 的一些虚拟函数和消息处理函数，

添加了一些新的函数，当然也设计了新的成员变量。

- CscrollView 新的成员变量

```
protected:
int m_nMapMode;
CSize m_totalLog;                  // total size in logical units (no rounding)
CSize m_totalDev;                  // total size in device units
CSize m_pageDev;                   // per page scroll size in device units
CSize m_lineDev;                   // per line scroll size in device units

BOOL m_bCenter;                    // Center output if larger than total size
BOOL m_bInsideUpdate;              // internal state for OnSize callback
```

- CScrollView 新的成员函数，用来完成和滚动操作、滚动条等有关的功能

```
void SetScaleToFitSize(SIZE sizeTotal);
void SetScrollSizes(int nMapMode, SIZE sizeTotal,
const SIZE& sizePage = sizeDefault,
const SIZE& sizeLine = sizeDefault);
```

这两个函数中的尺寸大小按逻辑单位计算。

SetScaleToFitSize 设置视口尺寸为当前的窗口尺寸，这样，在没有滚动条时，逻辑视的内容被放大或者缩小到正好窗口大小。

SetScrollSizes 设置窗口的映射模式，窗口尺寸，页和行尺寸。sizeDefualt 被定义为（0，0）。

- 下面几个函数用来实现滚动或者得到滚动条相关的信息

```
void ScrollToPosition(POINT pt);       // set upper left position
void FillOutsideRect(CDC* pDC, CBrush* pBrush);
void ResizeParentToFit(BOOL bShrinkOnly = TRUE);
CPoint GetScrollPosition() const;              // upper corner of scrolling
CSize GetTotalSize() const;                    // logical size
```

- 下面两个函数使用了设备坐标单位

```
CPoint GetDeviceScrollPosition() const;
void GetDeviceScrollSizes(int& nMapMode, SIZE& sizeTotal,
SIZE& sizePage, SIZE& sizeLine) const;
```

- 覆盖的消息处理函数

处理 WM_SIZE 的 OnSize；

处理 WM_HSCROLL 的 OnHScroll；

处理 WM_VSCROLL 的 OnVScroll；

- 覆盖的虚拟函数

CWnd 的 CalcWindowRect

CView 的 OnPrepareDC、OnScroll、OnScrollBy

- 用于 DEBUG 的 Dump 和 AssertValid

这里，覆盖的消息处理函数和虚拟函数共同完成对滚动条、滚动消息的处理。

在 CSrcollView 的实现涉及到许多和 Windows 映射模式、坐标转换等相关的函数的使用。这里，不作具体讨论。

12.3.2 CFormView

CFormView 派生于 CSrcollView，本身没有增加新的函数，但覆盖了一些基类的虚拟函数，增加了几个成员变量（以下列出的不包含 OLE 处理）。

（1） 增加的成员变量

```
LPCTSTR m_lpszTemplateName;
CCreateContext* m_pCreateContext;
HWND m_hWndFocus;     // last window to have focus
```

m_lpszTemplateName 用来保存创建视图的对话框模板的名称，_pCreateContext 用来保存创建上下文，m_hWndFocus 用来保存最近一次拥有焦点的控制窗口。在构造 CFormView 对象时，构造函数把有关信息保存到成员变量中，如下所示：

```
CFormView::CFormView(LPCTSTR lpszTemplateName)
{
m_lpszTemplateName = lpszTemplateName;
m_pCreateContext = NULL;
m_hWndFocus = NULL;         // focus window is unknown
}
```

（2） 覆盖的虚拟函数

```
virtual void OnDraw(CDC* pDC);          // MFC 缺省处理空
virtual BOOL Create(LPCTSTR, LPCTSTR, DWORD,
        const RECT&, CWnd*, UINT, CCreateContext*);
virtual BOOL PreTranslateMessage(MSG* pMsg);
virtual void OnActivateView(BOOL, CView*, CView*);
virtual void OnActivateFrame(UINT, CFrameWnd*);
```

创建基于对话框的视窗口，不同于创建普通视窗口（前者调用 CWnd::CreateEx，后者调用 CWnd::CreateDlg），故需要覆盖 Create 虚拟函数。

覆盖 PreTranslateMessage 是为了过滤对话框消息，把一些消息让 CFormView 对象来处理。

（3） 覆盖了两个消息处理函数：

```
afx_msg int OnCreate(LPCREATESTRUCT lpcs);
afx_msg void OnSetFocus(CWnd* pOldWnd);
```

下面，分析几个函数作。Create 函数解释了 MFC 如何使用一个对话框作为视的方法，PreTranslateMessage 显示了 CFormView 不同于 CDialog 的实现。

12.3.2.1 CFormView 的创建

设计 CFormView 的创建函数，必须考虑两个问题：

首先，CFormView 是一个视，其创建函数必须是一个虚拟函数，原型必须和 CWnd::Create(LPSTR…pContext)函数一致，见图 5-13 视的创建。其次，CFormView 使用了对话框创建函数和对话框“窗口类”来创建视，但必须作一些处理使得该窗口具备视的特征。Create 的实现如下：

```
BOOL CFormView::Create(LPCTSTR /*lpszClassName*/,
        LPCTSTR /*lpszWindowName*/,
      DWORD dwRequestedStyle, const RECT& rect, CWnd* pParentWnd, UINT nID,
```

```
        CCreateContext* pContext)
{
ASSERT(pParentWnd != NULL);
ASSERT(m_lpszTemplateName != NULL);

m_pCreateContext = pContext;        // save state for later OnCreate

#ifdef _DEBUG
        // dialog template must exist and be invisible with WS_CHILD set
        if (!_AfxCheckDialogTemplate(m_lpszTemplateName, TRUE))
        {
                ASSERT(FALSE);                  // invalid dialog template name
                PostNcDestroy();                // cleanup if Create fails too soon
                return FALSE;
        }
#endif //_DEBUG

//若 common control window 类还没有注册，则注册
VERIFY(AfxDeferRegisterClass(AFX_WNDCOMMCTLS_REG));

// call PreCreateWindow to get prefered extended style
CREATESTRUCT cs; memset(&cs, 0, sizeof(CREATESTRUCT));
if (dwRequestedStyle == 0)
        dwRequestedStyle = AFX_WS_DEFAULT_VIEW;
cs.style = dwRequestedStyle;
if (!PreCreateWindow(cs))
        return FALSE;

//::CreateDialogIndirect 间接被调用来创建一个无模式对话框
if (!CreateDlg(m_lpszTemplateName, pParentWnd))
        return FALSE;
//创建对话框时，OnCreate 被调用，m_pCreateContext 的作用结束了
m_pCreateContext = NULL;

// we use the style from the template - but make sure that
//    the WS_BORDER bit is correct
// the WS_BORDER bit will be whatever is in dwRequestedStyle
ModifyStyle(WS_BORDER|WS_CAPTION, cs.style & (WS_BORDER|WS_CAPTION));
ModifyStyleEx(WS_EX_CLIENTEDGE, cs.dwExStyle & WS_EX_CLIENTEDGE);

SetDlgCtrlID(nID);

CRect rectTemplate;
GetWindowRect(rectTemplate);
```

```
SetScrollSizes(MM_TEXT, rectTemplate.Size());

// initialize controls etc
if (!ExecuteDlgInit(m_lpszTemplateName))
        return FALSE;

// force the size requested
SetWindowPos(NULL, rect.left, rect.top,
        rect.right - rect.left, rect.bottom - rect.top,
        SWP_NOZORDER|SWP_NOACTIVATE);

// make visible if requested
if (dwRequestedStyle & WS_VISIBLE)
        ShowWindow(SW_NORMAL);

return TRUE;
}
```

从 Create 的实现过程可以看出，CreateDialog 在创建对话框时使用了 Windows 预定义的对话框“窗口类”，PreCreateWindow 返回的 cs 在创建对话框窗口时并没有得到体现，所以在 CFormView::Create 调用 PreCreateWindow 让程序员修改“窗口类”的风格之后，还要调用 ModifyStyle 和 ModifyStyleEx 来按 PreCreateWindow 返回的 cs 的值修改窗口风格。

回顾视窗口的创建过程，Create 函数被 CFrameWnd::CreateView 所调用，参数 nID 取值 AFX_IDW_PANE_FIRST。由于 CreateDlg 设置对话框窗口的 ID 为对话框模板的 ID，所以需要调用函数 SetDlgCtrlID(nID)设置视窗口 ID 为 nID(即 AFX_IDW_PANE_FIRST)。

由于 CFormView 是从 CScrollView 继承，所以调用 SetScrollSize 设置映射模式，窗口尺寸等。

完成上述动作之后，初始化对话框的控制子窗口。

最后，必要的话，显示视窗口。

这样，一个无模式对话框被创建，它被用作当前 MDI 窗口或者 MDI 子窗口的视。如同 CDialog 的消息处理一样，必要时，消息或者事件将传递给视原来的窗口过程(无模式对话框的原窗口过程)处理，其他的消息处理和通常视一样。

由于是调用对话框创建函数创建视窗口，所以不能向::CreateWindowEX 传递创建上下文指针，于是把它保存到成员变量 m_pCreateContext 中，在 OnCreate 时使用。OnCreate 的实现如下：

```
int CFormView::OnCreate(LPCREATESTRUCT lpcs)
{
//既然不能通过 CreateDialog 使用参数传递的方法得到创建上下文
//参数，则使用一个成员变量来传递
return CScrollView::OnCreate(lpcs);
}
```

12.3.2.2 CFormView 的消息预处理

现在，讨论 CFormView 的 PreTranslateMessage 函数。CDialog 覆盖函数 PreTranslateMessage 的主要目的是处理 Tooltip 消息、Escape 键盘消息和 Dialog 消息。CFormView 覆盖该函数的目的是处理 Tooltip 消息和 Dialog 消息。CFormView 和 CDialog 不同之处在于 CFormView 是一个视，故在把键盘消息当 Dialog 消息处理之前，必须优先让其父窗口检查按下的键是否是快捷键。PreTranslateMessage 函数实现如下：

```
BOOL CFormView::PreTranslateMessage(MSG* pMsg)
{
    ASSERT(pMsg != NULL);
    ASSERT_VALID(this);
    ASSERT(m_hWnd != NULL);

    //过滤 Tooltip 消息
    if (CView::PreTranslateMessage(pMsg))
        return TRUE;

    //SHIFT+F1 上下文帮助模式下，不处理 Dialog 消息
    CFrameWnd* pFrameWnd = GetTopLevelFrame();
    if (pFrameWnd != NULL && pFrameWnd->m_bHelpMode)
        return FALSE;

    //既然 IsDialogMessage 将把窗口快捷键解释成 Dialog 消息
    //所以在此先调用所有父边框窗口的消息预处理函数
    pFrameWnd = GetParentFrame();   // start with first parent frame
    while (pFrameWnd != NULL)
    {
        // allow owner & frames to translate before IsDialogMessage does
        if (pFrameWnd->PreTranslateMessage(pMsg))
            return TRUE;

        // try parent frames until there are no parent frames
        pFrameWnd = pFrameWnd->GetParentFrame();
    }

    // 过滤来自子窗口的消息或者给对话框的消息
    return PreTranslateInput(pMsg);
}
```

由于 CFormView 是一个视，不是模式对话框，所以它首先要把消息给父窗口（MDI 子窗口或者 MDI 窗口）预处理，如果它们不能处理，则调用 PreTranslateInput 来过滤 Dialog 消息。

12.3.2.3 CFormView 的输入焦点

CFormView 另一个特性是：在和用户交互中，如果用户离开视窗口，则必须保存 CFormView 视的哪个控制子窗口拥有输入焦点，以便在重新激活视窗口时，原来的那个窗口重新获得输入焦点。所以，CFormView 覆盖了虚拟函数 OnActivateView 和 OnActiveFrame，以便在视窗口失去激活时把它的当前输入焦点保存到成员变量 m_hWndFocus 中。

为了在适当时候恢复输入焦点，CFormView 覆盖了消息处理函数 OnSetFocus，以便在视获得输入焦点时把输入焦点传递给 m_hWndFocus（如果非空）。

至此，MFC 实现对话框的处理分析完毕。

在后面要讨论的工具条等控制窗口，类似于对话框也具备由 Windows 提供的窗口过程，MFC 在 SDK 的特定控制窗口创建函数的基础上，提供了 MFC 的窗口创建函数，使用 MFC 的窗口过程取代了它们原来的窗口过程，然后在必要的时候调用 Default 把有关消息和事件传递给原来的窗口过程处理。

第13章　MFC 工具条和状态栏

13.1　Windows 控制窗口

Windows(Windows 95 或者以上版本)提供了系列通用控制窗口，其中包括工具条(ToolBar)、状态栏(StatusBar)、工具条提示窗口(ToolTip)。

Windows 在一个 DLL 加载时注册个控制窗口的“窗口类”。例如，工具条的“窗口类”是“ToolbarWindow32”，状态栏的“窗口类”是“msctls_statusbar32”，工具条提示窗口的“窗口类”是“tooltips_class32”。为了保证该 DLL 被加载，使用控制“窗口类”前，应该首先调用函数 InitCommonControl。MFC 在窗口注册函数 AfxDeferRegisterClass 中实现了这一点。见 2.2.1 节 MFC 下窗口的注册。

创建通用控制窗口，可以使用专门的创建函数，如创建工具条的函数::CreateToolBarEx，创建状态栏的函数::CreateStatusBarEx。也可以调用窗口创建函数::CreateWindowEx，但是需要指定预定义的“窗口类”，必要的话还要其他步骤，如使用“ToolbarWindow32”“窗口类”创建工具栏后，还需要在工具栏中添加或者插入按钮。

一般，通用控制可以指定控制窗口风格(Style)。例如，具备风格 CCS_TOP，表示该控制窗口放到父窗口客户区的顶部，具备 CCS_BOTTOM，表示该控制窗口在客户区的底部。具体的控制窗口类可以有特别的适合于自己的风格，例如，TTS_ALWAYSTIP 表示只要光标落在工具栏的按钮上，ToolTip 窗口不论激活与否都会显示出来。

每一控制窗口类都有自己的窗口过程来处理自己的窗口消息，实现特定的功能。控制窗口类的窗口过程由 Windows 提供。

- 工具条

工具条的窗口过程处理了必要的消息，提供了标准工具条的功能，例如，工具条对客户化特征提供内在的支持，用户可以通过一个客户化对话框来添加、修改、删除或者重新安排工具条按钮。这些特征是否可以被用户所用或者用到什么地步是可以由程序控制的。

工具条的窗口过程将自动设置工具条的尺寸大小和位置，如果指定了控制窗口风格 CCS_TOP 或者 CCS_BOTTOM，则窗口过程把工具条放到父窗口客户区的顶部或者底部。窗口过程任何时候只要收到 WM_SIZE 或者 TB_AUTOSIZE 消息就自动地调整工具条的大小和位置。

工具条的按钮被选中后，会产生一个命令消息，它的窗口过程把该消息送给父窗口的窗口过程处理。

工具条中的按钮并不以子窗口的形式出现，而是以字符或者位图按钮的方式显示，每个按钮大小相同，缺省是 24*22 个像素。每个按钮都有一个索引，索引编号从 0 开始。每个按钮包括如下属性：

按钮的字符串索引，位图索引，风格，状态，命令 ID

按钮可以有两种风格 TBSTYLE_BUTTON 和 TBSTYLE_CHECK，前者像一个标准按钮那样响应用户的按击，后者响应每一次按击，在按下和跳起两种状态之间切换。按钮响应用户的动作，给父窗口发送一个包含了该按钮对应命令 ID 的命令消息。一般一个按钮的命令 ID 对应一个菜单项。

工具条维护两个列表，分别用来存放工具条按钮使用的字符串或者位图，列表中的位图或者

字符串从 0 开始编号，编号和按钮的索引相对应。
工具条可以是 Dockable（泊位）或者 Floatable（漂浮）的。
工具条可以有 TBSTYLE_TOOLTIPS 风格，如果具有这种风格，则创建和管理一个 Tooltip 控制，这是一个小的弹出式窗口，用来显示描述按钮的文本，平时该窗口隐藏，当鼠标落到按钮上面并停留约一秒后才弹出，在鼠标附近显示。
由于 Tooltip 窗口平时是隐藏的，所以不能接收鼠标消息来决定何时显示本窗口。这样，接收鼠标的窗口必须把鼠标消息送给 Tooltip 窗口，这是通过给 Tooptip 窗口发送消息 TTM_RELAYEVENT 来实现的。

- 状态栏

状态栏类似于工具条，有自己的窗口过程，可以泊位、漂浮。不过，习惯上状态栏都位于屏幕底部。每个状态条分成若干格（Status bar panes），每格从 0 开始编号，编号作为格的索引。每一个格，如同工具条的按钮一样，并不是一个 Windows 窗口。

13.2 MFC 的工具条和状态栏类

MFC 使用 CToolBarCtrl、CStatusBarCtrl 和 CToolTipCtrl 窗口类分别对工具条、状态栏、Tooltip 控制窗口进行了封装。
但是，直接使用这些类还不是很方便。MFC 提供了 CToolBar、CStatusBar 来处理状态栏和工具条，CToolBar、CStatusBar 功能更强大，灵活。这两个类都派生于 CControlBar。
在 MFC 下，建议这些控制条子窗口 ID 介于 AFX_IDW_TOOLBARFIRST(0xE800)和 AFX_IDW_CONTROLBAR_LAST（0Xe8FF）之间。这 256 个 ID 中，前 32 个又有其特殊性，用于 MFC 的打印预览中。
CControlBar 派生于 CWnd 类，是控制条窗口类的基类，它派生出 CToolBar、CStatusBar、CDockBar、CDialogBar、COleResizeBar 类。CControlBar 实现了以下功能：

- 和父窗口（边框窗口）的顶部或者底部或者其他边对齐。
- 可以包含子条目，这些条目或者是基于 HWND 的子窗口，或者是基于非 HWND 的条目。负责分配条目数组。
- 支持 CBRS_TOP（缺省，控制条放在顶部），CBRS_BOTTOM（放在底部），CBRS_NOALIGN（父窗口大小变化时不重新放置控制条）等几种控制风格。
- 支持派生类的实现。几个派生类有一定的共性，或者其中两个有一定的共性，这样 CControlBar 实现的函数一部分只适用于某个派生类，一部分适用于两个或者多个派生类，还有一部分适用于所有的派生类。所谓适用，这里指派生类直接继承了 CControlBar 的实现，或者覆盖了其实现但是建立在扩展其实现的基础上。类似地，CControlBar 的成员变量也不是为所有派生类所共同适用的。

CStatusBar 和 CControlBar 一方面建立在 CControlBar 的基础之上，另一方面以 Windows 的通用控制状态栏和工具条为基础。它们继承了 CControlBar 类的特性，但是所封装的窗口句柄是相应的 Windows 控制窗口的句柄，如同 CFormView 继承了 CSrcollView 的视类特性，但是其窗口句柄是无模式对话框窗口句柄一样。

典型地，如果在使用 AppWizard 生成应用程序时，指定了要求工具条和状态栏的支持，则在主边框窗口的 OnCreate 函数中包含一段如下的代码，用来创建工具条、状态栏和设置一些特性。

```
//创建工具栏
    if (!m_wndToolBar.Create(this) ||!m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
```

```
{
    TRACE0("Failed to create toolbar\n");
    return -1;          // fail to create
}
//创建状态栏
if (!m_wndStatusBar.Create(this) ||
    !m_wndStatusBar.SetIndicators(indicators, sizeof(indicators)/sizeof(UINT)))
{
    TRACE0("Failed to create status bar\n");
    return -1;          // fail to create
}

// TODO: Remove this if you don't want tool tips or a resizeable toolbar
//对工具栏设置 Tooltip 特征
m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
    CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

//使得工具栏可以泊位在边框窗口
// TODO: Delete these three lines if you don't want the toolbar to
//  be dockable
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);
```

工具条除了 Tooltip，Resizeable，Dockable 特性外，还可以是 Floatable。应用程序可以使用 CFrameWnd::SaveBarState 保存边框窗口的控制条的有关信息到 INI 文件或者 Windows Register 库，使用 LoadBarSate 从 INI 文件或者 Register 库中读取有关信息并恢复各个控制条的设置。

下文，将讨论工具条等的创建、销毁，从中分析 CControlBar 和派生类的关系，讨论 CControlBar 如何实现共性，如何支持派生类的特定要求，派生类又如何实现自己的特定需求等。

13.2.1 控制窗口的创建

创建工具条、状态条、对话框工具栏的方法是不同的，所以必须给每个派生类 CToolBar、CStatusBar、CDialogBar 设计和实现自己的窗口创建函数 Create。但是，它们是也是有共性的，共性由 CControlBar 的 PreCreateWindow 处理。在窗口创建之后，各个派生类都要进行的处理（共性）由 CControlBar 的 OnCreate 完成，特别的处理通过派生类的 OnNcCreate 完成。

13.2.1.1 PreCreateWindow

首先，讨论 CControlBar 类的 PreCreateWindow 的实现。

```
BOOL CControlBar::PreCreateWindow(CREATESTRUCT& cs)
{
if (!CWnd::PreCreateWindow(cs))
	return FALSE;

//修改窗口风格，强制适用 clipsliblings，以防重复绘制
cs.style |= WS_CLIPSIBLINGS;

//default border style translation for Win4
//(you can turn off this translation by setting CBRS_BORDER_3D)
if (afxData.bWin4 && (m_dwStyle & CBRS_BORDER_3D) == 0)
{
	DWORD dwNewStyle = 0;
	switch (m_dwStyle & (CBRS_BORDER_ANY|CBRS_ALIGN_ANY))
	{
		case CBRS_LEFT: //控制条在边框窗口的左边显示
		dwNewStyle = CBRS_BORDER_TOP|CBRS_BORDER_BOTTOM;
		break;
		case CBRS_TOP://控制条在边框窗口的顶部显示
		dwNewStyle = CBRS_BORDER_TOP;
		break;
		case CBRS_RIGHT://控制条在边框窗口的右边显示
		dwNewStyle = CBRS_BORDER_TOP|CBRS_BORDER_BOTTOM;
		break;
		case CBRS_BOTTOM://控制条在边框窗口的底部显示
		dwNewStyle = CBRS_BORDER_BOTTOM;
		break;
	}

	// set new style if it matched one of the predefined border types
	if (dwNewStyle != 0)
	{
		m_dwStyle &= ~(CBRS_BORDER_ANY);
		m_dwStyle |= (dwNewStyle | CBRS_BORDER_3D);
	}
}
return TRUE;
}
```

其中，afxData 是一个全局变量，MFC 用它来记录系统信息，如版本信息等。这里 afxData.bWin4 表示 Windows 版本是否高于 4.0。

CToolBar 的 PreCreateWindow 函数修改了窗口风格，也修改状态栏、工具栏等的 CBRS_风格。CBRS_风格的改变不会影响窗口风格。因为这些 CBRS_风格被保存在成员变量 m_dwStyle 中。

除了上述在程序中用到的影响工具条、状态栏等显示位置的 CBRS_风格外，还有和泊位相

关的 CBRS_风格，CBRS_ALIGN_LEFT、CBRS_ALIGN_RIGHT、CBRS_ALIGN_BOTTOM、CBRS_ALIGN_TOP、CBRS_ALIGN_ANY，分别表示工具条可以在停泊在边框窗口的左边、右边、底部、顶部或者所有这些位置；和漂浮相关的 CBRS_风格 CBRS_FLOAT_MULTI，表示多个工具条可以漂浮在一个微型边框窗口中；和 Tooltips 相关的 CBRS_风格 CBRS_TOOLTIPS 和 CBRS_FLYBY。
派生类如果没有特别的要求，可以不覆盖 PreCreateWindow 函数。CStatusBar 因为有更具体和特殊的风格要求，所以它覆盖了 PreCreateWindow。CStatusBar 的覆盖实现调用了 CControlBar 的实现。
派生类也可以在覆盖实现中修改 PreCreateWindow 参数 cs，改变窗口风格；修改 m_dwStyle，改变 CBRS_风格。

13.2.1.2 控制条的窗口创建

CControlBar 派生类实现了自己的窗口创建函数 Create，CControlBar 的 PreCreateWindow 被派生类的 Create 函数直接或者间接地调用。以 CToolBar 为例讨论窗口创建函数和创建过程。
（1） CToolBar 的窗口创建函数 Create
Create 函数实现如下：

```
BOOL CToolBar::Create(CWnd* pParentWnd, DWORD dwStyle, UINT nID)
{
ASSERT_VALID(pParentWnd);     // must have a parent
ASSERT (!((dwStyle & CBRS_SIZE_FIXED) &&
        (dwStyle & CBRS_SIZE_DYNAMIC)));

//  保存 dwStyle 指定的 CBRS_风格
m_dwStyle = dwStyle;
if (nID == AFX_IDW_TOOLBAR)
      m_dwStyle |= CBRS_HIDE_INPLACE;

//去掉参数 dwStyle 包含的 CBRS_风格
dwStyle &= ~CBRS_ALL;
//设置窗口风格
dwStyle |=
CCS_NOPARENTALIGN|CCS_NOMOVEY|CCS_NODIVIDER|CCS_NORESIZE;

//初始化通用控制，可以导致 InitCommonControl 的调用
VERIFY(AfxDeferRegisterClass(AFX_WNDCOMMCTLS_REG));

//创建窗口，将调用 PreCreateWindow,OnCreate, OnNcCreate 等
CRect rect; rect.SetRectEmpty();
if (!CWnd::Create(TOOLBARCLASSNAME, NULL, dwStyle,
      rect, pParentWnd, nID))
return FALSE;

// Note: Parent must resize itself for control bar to be resized
```

```
return TRUE;
}
```

其中：

Create 函数的参数 1 表示工具条的父窗口。参数 2 指定窗口风格和 CBRS_风格，缺省值为 WS_CHILD | WS_VISIBLE | CBRS_TOP，其中 WS_CHILD 和 WS_VISIBLE 是窗口风格，CBRS_TOP 是 CBRS_风格。参数 3 指定工具条 ID，缺省值为 AFX_IDW_TOOLBAR（0X0E800 或者 59392）。如果还有多个工具栏要显示，在创建它们时则必须给每个工具栏指明 ID。

首先，Create 函数把参数 2（dwStyle）指定的窗口风格和 CBRS_风格分离出来，窗口风格保留在 dwStyle 中，CBRS_风格保存到成员变量 m_dwStyle 中。CToolBar::PreCreateWindow 将进一步修改这些风格。

接着，Create 函数调用了函数 AfxDeferRegisterClass。它如果没有注册 TOOLBARCLASSNAME 表示的“窗口类”，就注册该类；否则，返回 TRUE，表示已经注册。TOOLBARCLASSNAME 表示的字符串是“ToolbarWindow32”，即“窗口类”名称。

然后，调用 CWnd::Create(7 个参数)使用“ToolbarWindow32”“窗口类”创建工具栏。

Create 在创建窗口的过程中，用 MFC 的标准窗口过程取代原来的窗口过程，如同 CFormView 和 CDialog 窗口创建时窗口过程被取代一样，并发送 WM_CREATE 和 WM_NCCREATE 消息。

至于添加向工具栏添加按钮，则由函数 LoadToolBar 完成。在分析 LoadToolBar 函数之前，先讨论 OnCreate、OnNcCreate 等函数。

（2） 处理 WM_CREATE 消息

CControlBar 提供了消息处理函数 OnCreate 来处理 WM_CREATE 消息。

```
int CControlBar::OnCreate(LPCREATESTRUCT lpcs)
{
//调用基类的实现
if (CWnd::OnCreate(lpcs) == -1)
return -1;
//针对工具栏，是否有 Tooltip 特性
if (m_dwStyle & CBRS_TOOLTIPS)
    EnableToolTips();
//得到父窗口，并添加自身到其控制条列表中
CFrameWnd *pFrameWnd = (CFrameWnd*)GetParent();
if (pFrameWnd->IsFrameWnd())
{
    m_pDockSite = pFrameWnd;
    m_pDockSite->AddControlBar(this);
}
return 0;
}
```

如果需要支持 Tooltips，则 OnCreate 调用 EnableTooltips。

m_pDockSite 是 CControlBar 的和泊位相关的成员变量，这里把它初始化为拥有工具栏的父边框窗口，该边框窗口把控制条加入其控制条列表 m_listControlBars 中。

在处理 WM_CREATE 之前，派生类先处理消息 WM_NCCREAE。例如，CToolBar 覆盖了 OnNcCreate 函数。

（3） 处理 WM_NCCREATE 消息

CToolBar 对 WM_NCCREATE 消息的处理如下：

```
BOOL CToolBar::OnNcCreate(LPCREATESTRUCT lpCreateStruct)
{
if (!CControlBar::OnNcCreate(lpCreateStruct))
    return FALSE;
// if the owner was set before the toolbar was created, set it now
if (m_hWndOwner != NULL)
    DefWindowProc(TB_SETPARENT, (WPARAM)m_hWndOwner, 0);

DefWindowProc(TB_BUTTONSTRUCTSIZE, (WPARAM)sizeof(TBBUTTON), 0);
return TRUE;
}
```

CToolBar 覆盖 CcontrolBar 的该函数用来设置工具条的所属窗口和描述工具条按钮结构的大小，这两个动作都是通过给工具条窗口发送消息来实现的。因为这些消息被送给控制窗口类的窗口过程（Windows 提供的）来处理，所以直接调用 DefWindowProc，省却了消息发送的过程。

在控制窗口创建之后，对于工具条来说，下一步就是向工具栏添加按钮。

（4） 向工具栏添加按钮

通过函数 LoadToolBar 完成向工具栏添加按钮的任务，其实现如下：

```
BOOL CToolBar::LoadToolBar(LPCTSTR lpszResourceName)
{
ASSERT_VALID(this);
ASSERT(lpszResourceName != NULL);

//查找并确认按钮位图、字符串等资源的位置
HINSTANCE hInst = AfxFindResourceHandle(lpszResourceName, RT_TOOLBAR);
HRSRC hRsrc = ::FindResource(hInst, lpszResourceName, RT_TOOLBAR);
if (hRsrc == NULL)
    return FALSE;

//锁定资源
HGLOBAL hGlobal = LoadResource(hInst, hRsrc);
if (hGlobal == NULL)
    return FALSE;

CToolBarData* pData = (CToolBarData*)LockResource(hGlobal);
if (pData == NULL)
    return FALSE;
ASSERT(pData->wVersion == 1);

//复制与各个位图对应的命令 ID 到数组 pItem
UINT* pItems = new UINT[pData->wItemCount];
for (int i = 0; i < pData->wItemCount; i++)
```

```
        pItems[i] = pData->items()[i];
    //添加按钮到工具栏，指定各个按钮对应的 ID
    BOOL bResult = SetButtons(pItems, pData->wItemCount);
    delete[] pItems;

    //设置按钮的位图
    if (bResult)
    {
        // set new sizes of the buttons
        CSize sizeImage(pData->wWidth, pData->wHeight);
        CSize sizeButton(pData->wWidth + 7, pData->wHeight + 7);
        SetSizes(sizeButton, sizeImage);

        // load bitmap now that sizes are known by the toolbar control
        bResult = LoadBitmap(lpszResourceName);
    }

    UnlockResource(hGlobal);
    FreeResource(hGlobal);

    return bResult;
    }
```

LoadToolBar 函数的参数指定了资源。ToolBar 资源的类型是 RT_TOOLBAR，ToolBar 位图资源的类型是 RT_BITMAP。

在 RT_TOOLBAR 类型的资源读入内存之后，可以用 CToolBarData 结构描述。一个这样的结构包括了 ToolBar 资源的如下信息：

工具条位图的版本，宽度，高度，个数，各个位图对应的命令 ID。

然后，LoadToolBar 把这些命令 ID 被复制到数组 pItem 中；根据位图宽度、高度形成按钮尺寸 sizeButton 和位图尺寸 sizeImage。

接着，调用 SetBottons 添加按钮到工具栏，把各个按钮和命令 ID 对应起来；调用 SetSizes 设置按钮和位图的尺寸大小；调用 LoadBitmap 添加或者取代工具条的位图列表。这些动作都是调用工具栏“窗口类”的窗口过程完成的。例如，SetButtons 的实现：

```
    BOOL CToolBar::SetButtons(const UINT* lpIDArray, int nIDCount)
    {
    ASSERT_VALID(this);
    ASSERT(nIDCount >= 1);  // must be at least one of them
    ASSERT(lpIDArray == NULL ||
    AfxIsValidAddress(lpIDArray, sizeof(UINT) * nIDCount, FALSE));

    //首先，删除工具条中现有的按钮
    int nCount = (int)DefWindowProc(TB_BUTTONCOUNT, 0, 0);
    while (nCount--)
    VERIFY(DefWindowProc(TB_DELETEBUTTON, 0, 0));
```

```
if (lpIDArray != NULL)//命令 ID 数组非空
{
	//添加新按钮
	TBBUTTON button; memset(&button, 0, sizeof(TBBUTTON));
	int iImage = 0;
	for (int i = 0; i < nIDCount; i++)
	{
		button.fsState = TBSTATE_ENABLED;
		if ((button.idCommand = *lpIDArray++) == 0)
		{
			//按钮之间分隔
			button.fsStyle = TBSTYLE_SEP;
			//按钮之间隔 8 个像素
			button.iBitmap = 8;
		}
		else
		{
			//有位图和命令 ID 的按钮
			button.fsStyle = TBSTYLE_BUTTON;
			button.iBitmap = iImage++;//设置位图索引
		}
		//添加按钮
		if (!DefWindowProc(TB_ADDBUTTONS, 1, (LPARAM)&button))
			return FALSE;
	}
}
else//命令 ID 数组空，添加空按钮
{
	TBBUTTON button; memset(&button, 0, sizeof(TBBUTTON));
	button.fsState = TBSTATE_ENABLED;
	for (int i = 0; i < nIDCount; i++)
	{
		ASSERT(button.fsStyle == TBSTYLE_BUTTON);
		if (!DefWindowProc(TB_ADDBUTTONS, 1, (LPARAM)&button))
			return FALSE;
	}
}
//记录按钮个数到成员变量 m_nCount 中
m_nCount = (int)DefWindowProc(TB_BUTTONCOUNT, 0, 0);

//稍后放置按钮
m_bDelayedButtonLayout = TRUE;

return TRUE;
```

}
函数的参数 1 是一个数组，数组的各个元素就是命令 ID；参数 2 是按钮的个数。首先，SetButtons 删除工具条原来的按钮；然后，添加新的按钮，若命令 ID 数组非空，则把每一个按钮和命令 ID 对应并分配位图索引，否则设置空按钮并返回 FALSE；最后，记录按钮个数。
从 SetButtons 的实现可以看出，对工具条的所有操作都是通过工具条“窗口类”的窗口过程完成的，SetSizes、LoadBitmap 也是如此，这里不作讨论。
（5） 状态栏和对话框工具栏的创建
至此，分析了 MFC 创建工具条窗口的过程。对于状态栏和对话框工具栏有类似的步骤，但也有不同之处。
CStatusBar 的 Create 使用“msctls_statusbar32”“窗口类”创建状态栏，窗口 ID 为 AFX_IDW_STATUS_BAR(0XE801)，然后通过成员函数 SetIndictors 给状态栏分格，类似于给工具条添加按钮的过程，它实际上是通过状态栏“窗口类”的窗口过程完成的。
CDialogBar 的 Create 使用 CreateDlg 创建对话框工具栏，类似于 CFormView 的过程。在工具栏窗口创建之后，要添加到父窗口的工具栏列表中，这通过 CControlBar::OnCreate 完成。这样创建的结果导致窗口过程使用 MFC 的统一的窗口过程，相应“窗口类”的窗口过程也将在缺省处理中被调用，这一点如同 CFormView 和 CDialog 中所描述的。在初始化对话框的时候完成了各个控制按钮的添加。
CStatusBar 和 CdialogBar 都没有处理消息 WM_NCCREATE。
关于 CStautsBar 和 CDialogBar 创建过程的具体实现，这里不作详细讨论了。

13.2.2 控制条的销毁

描述了控制条的创建，顺便考察其销毁的设计。
工具条、状态栏等这些控制窗口都要使用 DestroyWindow 来销毁，所有有关操作集中由 CControlBar 处理。CControlBar 覆盖了虚拟函数 DestroyWindow、PostNcDestroy 和消息处理函数 OnDestroy。
当然，各个派生类的虚拟析构函数被实现。如果成员变量 m_bAutoDelete 为 TRUE，则动态创建的 MFC 窗口将自动销毁。

13.2.3 处理控制条的位置

13.2.3.1 计算控制条位置的过程和算法

工具条等控制条是作为一个子窗口在父边框窗口内显示的。为了处理控制条的布置(Layout)，首先需要计算出控制条的尺寸大小，这个工作被委派给工具条等控制窗口自己来完成。为此，CControlBar 提供了两个函数来达到这个目的：CalcFixLayout，CalcDynamicLayout。这两个函数都是虚拟函数。各个派生类都覆盖了这两个或者其中一个函数，用来计算自身的尺寸大小。这些计算比较琐碎，在此不作详细讨论。其次，在父窗口位置或者大小变化时，控制条的大小和位置要作相应的调整。
下面，描述 MFC 确定或者更新工具条、状态栏等位置的步骤：

（1）边框窗口在必要的时候调用虚拟函数 RecalcLayout 来重新放置它的控制条和客户窗口，例如在创建窗口时、响应消息 WM_SIZE 时（见 5.3.3.5 节)边框窗口的初始化）。
（2）CFrameWnd::RecalcLayout 调用 CWnd 的成员函数 RepositionBars 完成控制条窗口的重新放置。
（3）CWnd::RepositionBars 作如下的处理：
RepositionBars 首先给各个控制子窗口发送（Send）MFC 内部使用的消息 WM_SIZEPARENT，把窗口客户区矩形指针传递给它们，给它们一个机会来确认自己的尺寸。
然后，各个控制子窗口用 OnSizeParent 响应 WM_SIZEPARENT 消息；ControlBar 实现了消息处理函数 OnSizeParent，它调用 CalcDynamicLayout 等函数确定本窗口的大小，并从客户区矩形中减去自己的尺寸。
在所有的控制子窗口处理了 OnSizeParent 消息之后，RepositonBars 利用返回的信息调用函数 CalcWindowRect 计算客户区窗口（MDI 客户窗口、View 等）的大小。
最后，调用::EndDeferWindowPos 或者::SetWindowPos 放置所有的窗口（控制子窗口和客户窗口）。
在窗口被放置的时候，发送消息 WM_WINDOWPOSCHANGING 和 WM_WINDOWPOSCHANGED。MFC 的实现中，控制窗口响应了前一个消息，消息处理函数是 OnWindowPosChanging。CControlBar、CToolBar 和 CStatusBar 等实现了消息处理函数 OnWindowPosChanging。

上述处理过程所涉及的这些函数中，RecalcLayout 是 CFrameWnd 定义的虚拟函数；RepostionBars 是 CWnd 的成员函数；CalcaWindowRect 是 CWnd 的虚拟函数；OnSizeParent 是 CControlBar 定义的消息处理函数；OnWindowPosChanging 是 CToolbar、CStatusBar、CDockBar 等 CControlBar 派生类定义的消息处理函数。
下面，对其中两个函数 RecalcLayout 和 RepositionBars 作一些分析。

13.2.3.2 CFrameWnd 的虚拟函数 RecalcLayout

RecalcLayout 的实现如下：

```
void CFrameWnd::RecalcLayout(BOOL bNotify)
{
//RecalcLayout 是否正在被调用
if (m_bInRecalcLayout)
        return;

m_bInRecalcLayout = TRUE;
// clear idle flags for recalc layout if called elsewhere
if (m_nIdleFlags & idleNotify)
        bNotify = TRUE;
m_nIdleFlags &= ~(idleLayout|idleNotify);

//与 OLE 相关的处理
#ifndef _AFX_NO_OLE_SUPPORT
        // call the layout hook -- OLE support uses this hook
if (bNotify && m_pNotifyHook != NULL)
```

```
        m_pNotifyHook->OnRecalcLayout();
    #endif

    //是否包含浮动(floating)控制条的边框窗口(CMiniFrameWnd 类)
    if (GetStyle() & FWS_SNAPTOBARS)
    {
        //计算控制条和边框窗口的位置、尺寸并设置它们的位置
        CRect rect(0, 0, 32767, 32767);
        RepositionBars(0, 0xffff, AFX_IDW_PANE_FIRST, reposQuery,
            &rect, &rect, FALSE);
        RepositionBars(0, 0xffff, AFX_IDW_PANE_FIRST, reposExtra,
            &m_rectBorder, &rect, TRUE);
        CalcWindowRect(&rect);
        SetWindowPos(NULL, 0, 0, rect.Width(), rect.Height(),
        SWP_NOACTIVATE|SWP_NOMOVE|SWP_NOZORDER);
    }
    else
        //是普通边框窗口，则设置其所有子窗口的位置、尺寸
        RepositionBars(0, 0xffff, AFX_IDW_PANE_FIRST,
            reposExtra, &m_rectBorder);

    //本函数处理完毕
    m_bInRecalcLayout = FALSE;
    }
```

该函数主要的目的是调用 RepositionBars 函数，它分两种情况来调用 RepositionBars 函数。一种情况是当前边框窗口为浮动控制条的包容窗口（微型边框窗口）时；另一种情况是当前边框窗口为普通边框窗口时。

13.2.3.3 CWnd 的成员函数 RepositionBars

RepositionBars 的实现如下：

```
    void CWnd::RepositionBars(UINT nIDFirst, UINT nIDLast, UINT nIDLeftOver,
        UINT nFlags, LPRECT lpRectParam, LPCRECT lpRectClient, BOOL bStretch)
    {
    ASSERT(nFlags == 0 || nFlags == reposQuery || nFlags == reposExtra);

    AFX_SIZEPARENTPARAMS layout;
    HWND hWndLeftOver = NULL;

    layout.bStretch = bStretch;
    layout.sizeTotal.cx = layout.sizeTotal.cy = 0;
    if (lpRectClient != NULL)
        layout.rect = *lpRectClient;       //从参数 6 得到客户区
    else
```

```
        //参数 lpRectClient 空，得到客户区域
        GetClientRect(&layout.rect);

if (nFlags != reposQuery)
        //准备放置各个子窗口(layout)
        layout.hDWP = ::BeginDeferWindowPos(8); // reasonable guess
else
        layout.hDWP = NULL; // not actually doing layout

//按一定顺序给各个控制条发送父窗口 resize 的消息；
//各个控制条窗口收到消息后，从客户区中扣除自己使用的区域；
//并且必要的话每个控制窗口调用::DeferWindowPos
//剩下的区域留给 nIDLeftOver 子窗口
for (HWND hWndChild = ::GetTopWindow(m_hWnd); hWndChild != NULL;
        hWndChild = ::GetNextWindow(hWndChild, GW_HWNDNEXT))
{
        UINT nIDC = _AfxGetDlgCtrlID(hWndChild);
        CWnd* pWnd = CWnd::FromHandlePermanent(hWndChild);
        //如果是指定的 nIDLeftOver 子窗口，则保存其窗口句柄；
        //否则，是控制条窗口，给它们发送 WM_SIZEPARENT 消息
        if (nIDC == nIDLeftOver)
                hWndLeftOver = hWndChild;
        else if (nIDC >= nIDFirst && nIDC <= nIDLast && pWnd != NULL)
                //如果 layout->hDWP 非空， OnSizeParent 则将执行窗口布置的操作
                ::SendMessage(hWndChild, WM_SIZEPARENT, 0, (LPARAM)&layout);
        }

        //如果是 reposQuery，则得到客户区矩形，返回
        if (nFlags == reposQuery)
        {
                ASSERT(lpRectParam != NULL);
                if (bStretch)
                        ::CopyRect(lpRectParam, &layout.rect);
                else
                {
                        lpRectParam->left = lpRectParam->top = 0;
                        lpRectParam->right = layout.sizeTotal.cx;
                        lpRectParam->bottom = layout.sizeTotal.cy;
                }
        return;
}

//其他情况下（reposDefault、reposExtra），则需要执行 Layout 操作
```

```
//处理 hWndLeftOver(nIDLeftOver 子窗口)
if (nIDLeftOver != 0 && hWndLeftOver != NULL)
{
    CWnd* pLeftOver = CWnd::FromHandle(hWndLeftOver);
    // allow extra space as specified by lpRectBorder
    if (nFlags == reposExtra)
    {
        ASSERT(lpRectParam != NULL);
        layout.rect.left += lpRectParam->left;
        layout.rect.top += lpRectParam->top;
        layout.rect.right -= lpRectParam->right;
        layout.rect.bottom -= lpRectParam->bottom;
    }
    //基于 layout.rect 表示的客户尺寸计算出窗口尺寸
    pLeftOver->CalcWindowRect(&layout.rect);
    //导致函数::DeferWindowPos 的调用
    AfxRepositionWindow(&layout, hWndLeftOver, &layout.rect);
}

//给所有的窗口设置尺寸、位置(size and layout)
if (layout.hDWP == NULL || !::EndDeferWindowPos(layout.hDWP))
    TRACE0("Warning: DeferWindowPos failed - low system resources.\n");
}
```

RepositionBars 用来改变客户窗口中控制条的尺寸大小或者位置，其中：
参数 1 和参数 2 定义了需要重新放置的子窗口 ID 的范围，一般是 0 到 0xFFFF。
参数 3 指定了一个子窗口 ID，它拥有客户窗口剩下的空间，一般是 AFX_IDW_PANE_FIRST，表示视的窗口 ID。
参数 4 指定了操作类型，缺省是 CWnd::ReposDefault，表示执行窗口放置操作，参数 5 不会用到；若取值 CWnd::ReposQuery，则表示尝试进行窗口放置（Layout），但最后不执行这个操作，只是把参数 5 初始化成客户区的尺寸大小；若取值 CWnd::ReposExtra，则把参数 5 的值加到参数 2 表示的子窗口的客户区域，并执行窗口放置操作。
参数 6 表示传递给函数的可用窗口客户区的尺寸，如果空则使用窗口客户区尺寸。
如果执行 layout 操作的话，该函数的核心处理就是：
首先，调用::BeginDeferWindowPos 初始化一个 Windows 内部的多窗口位置结构（Multiple-window - position structure）hDWP；
然后，让各个子窗口逐个调用::DeferWindowPos，更新 hDWP。在调用::DeferWindowPos 之前，要作一个确定子窗口大小的工作。这些工作通过给各个控制子窗口发送消息 WM_SIZEPARENT 来完成。
控制子窗口通过函数 OnSizeParent 响应 WM_SIZEPARENT 消息，先确定自己的尺寸，然后，如果需要进行窗口布置(WM_SIZEPARENT 消息参数 lParam 包含了一个非空的 HDWP 结构（lpLayout->hDWP）)，则 OnSizeParent 将调用 AfxRepositionWindow 函数计算本控制窗口的位置，结果保存到 hDWP 中。
在所有的控制窗口尺寸确定之后，剩下的留给窗口 hWndLeftOver（如果存在的话）。确定了 hWndLeftOver 的大小之后，调用 AfxRepositionWindow 函数计算其位置，结果保存到 hDWP

中。
上面提到的函数 AfxRepositionWindow 间接调用了::DeferWindowPos。
最后，::EndDeferWindowPos，使用 hDWP 安排所有子窗口的位置和大小。

至于其他函数，如 OnSizeparent、OnWindowPosChanging、CalcWindowRect，这里不作进一步的分析。

13.2.4 工具条、状态栏和边框窗口的接口

13.2.4.1 应用程序在状态栏中显示信息

MFC 内部通过给边框窗口发送消息 WM_SETMESSAGESTRING 、WM_POPMESSAGESTRING 的方式在状态栏中显示信息。这两个消息在 afxpriv.h 里头定义。WM_SETMESSAGESTRING 消息表示在状态栏中显示和某个 ID 对应的字符串信息或者指定的字符串信息，消息参数 wParam 指定了字符串资源 ID，消息参数 lParam 指定了字符串指针，两个消息参数只有一个有用。一般，一个命令 ID 对应了一个字符串 ID，对应的字符串是命令 ID 的说明。
消息 WM_POPMESSAGESTRING 用来重新设置状态栏。
这两个消息对应的消息处理函数分别是 OnSetMessageString 和 OnPopMessageString，OnSetMessageString 和 OnPopMessageString 分别实现如下：
（1） OnSetMessageString

```
LRESULT CFrameWnd::OnSetMessageString(WPARAM wParam, LPARAM lParam)
{
//最近一次被显示的消息字符串 IDS（一个消息对应的字符串）
UINT nIDLast = m_nIDLastMessage;
m_nFlags &= ~WF_NOPOPMSG;

//得到状态栏
CWnd* pMessageBar = GetMessageBar();
if (pMessageBar != NULL)
{
    LPCTSTR lpsz = NULL;
    CString strMessage;

    //设置状态栏文本
    if (lParam != 0) //指向一个字符串
    {
        ASSERT(wParam == 0);    // can't have both an ID and a string
        lpsz = (LPCTSTR)lParam; // set an explicit string
    }
    else if (wParam != 0)//一个字符串资源 IDS
    {
```

```
            //打印预览时映射 SC_CLOSE 成 AFX_IDS_PREVIEW_CLOSE;
            if (wParam == AFX_IDS_SCCLOSE && m_lpfnCloseProc != NULL)
                wParam = AFX_IDS_PREVIEW_CLOSE;

            //得到资源 ID 所标识的字符串
            GetMessageString(wParam, strMessage);
            lpsz = strMessage;
        }
        //在状态栏中显示文本
        pMessageBar->SetWindowText(lpsz);

        // 根据最近一次选择的消息更新状态条所属窗口的有关记录
        CFrameWnd* pFrameWnd = pMessageBar->GetParentFrame();
        if (pFrameWnd != NULL)
        {
            //记录最近一次显示的消息字符串
            pFrameWnd->m_nIDLastMessage = (UINT)wParam;
            //记录最近一次 Tracking 的命令 ID 和字符串 IDS
            pFrameWnd->m_nIDTracking = (UINT)wParam;
        }
    }

    m_nIDLastMessage = (UINT)wParam;       // new ID (or 0)
    m_nIDTracking = (UINT)wParam;          // so F1 on toolbar buttons work
    return nIDLast;
    }
```

OnSetMessageString 函数直接或者从 ID 从字符串资源中得到字符串指针。如果是从 ID 得到字符串指针，则函数 GetMessageString 被调用。

和命令 ID 对应的字符串由两部分组成，前一部分用于在状态栏显示，后一部分用于 Tooltip 显示，分隔符号是“\n”。例如，字符串 ID_APP_EXIT（对应“退出”菜单、按钮）是“Exit Application\nExit”，当鼠标落在“退出”按钮上时，状态栏显示“Exit Application”，Tooltip 显示“Exit”。根据这种格式，GetMessageString 分离出第一部分的文本信息。至于第二部分的用途将在讨论 Tooltip 的章节将用到。

得到了字符串之后，OnSetMessageString 调用状态栏的 SetWindowText 函数。SetWindowText 导致消息 WM_SETTEXT 消息发送给状态栏，状态栏的消息处理函数 OnSetText 被调用，实际上等于调用了 SetPaneText(0, lpsz)，即在状态栏的第 0 格中显示字符串 lpsz 的信息。对于工具栏来说，SetWindowText 可以认为是 SetPaneText(0, lpsz)的简化版本。

顺便指出，pMessageBar->GetParentFrame()返回主边框窗口，即使 pMessageBar 指向漂浮的工具条。关于泊位和漂浮，见后面 13.2.5 节的描述。

关于 OnSetText，其实现如下：

```
LRESULT CStatusBar::OnSetText(WPARAM, LPARAM lParam)
{
ASSERT_VALID(this);
ASSERT(::IsWindow(m_hWnd));
```

```
int nIndex = CommandToIndex(0); //返回 0
if (nIndex < 0)
    return -1;

return SetPaneText(nIndex, (LPCTSTR)lParam) ? 0 : -1;
}
```

（2） OnPopMessageString

```
LRESULT CFrameWnd::OnPopMessageString(WPARAM wParam,
        LPARAM lParam)
{
//WF_NOPOPMSG 表示边框窗口不处理 WM_POPMESSAGESTRING
if (m_nFlags & WF_NOPOPMSG)
    return 0;

//调用 OnSetMessageString
return SendMessage(WM_SETMESSAGESTRING, wParam, lParam);
}
```

一般，在清除状态栏消息时，发送 WM_POPMESSAGESTRING，通过消息参数 wParam 指定一个字符串资源，其 ID 为 AFX_IDS_IDLEMESSAGE，对应的字符串是“Ready”。

13.2.4.2 状态栏显示菜单项的提示信息

状态栏的一个重要作用是显示菜单命令或者工具条按钮的提示信息。本节讨论如何显示菜单命令的提示信息，关于工具条按钮在这方面的讨论见后面 13.2.4.4 章节。

显示菜单命令的提示信息，就是每当一个菜单项被选中之后，在状态栏显示该菜单的功能、用法等信息。这些信息以字符串资源的形式保存，字符串 ID 对应于菜单项的命令 ID。

所以，必须处理菜单选择消息 WM_MENUSELECT。CFrameWnd 实现了消息处理函数 OnMenuSelect，其实现如下：

```
void CFrameWnd::OnMenuSelect(UINT nItemID,
    UINT nFlags, HMENU /*hSysMenu*/)
{
CFrameWnd* pFrameWnd = GetTopLevelFrame();
ASSERT_VALID(pFrameWnd);

//跟踪被选中的菜单项
if (nFlags == 0xFFFF)
{
    //取消菜单操作
    m_nFlags &= ~WF_NOPOPMSG;
    if (!pFrameWnd->m_bHelpMode)
        m_nIDTracking = AFX_IDS_IDLEMESSAGE;
    else
        m_nIDTracking = AFX_IDS_HELPMODEMESSAGE;
```

```
        //在状态栏显示
        SendMessage(WM_SETMESSAGESTRING, (WPARAM)m_nIDTracking);
        ASSERT(m_nIDTracking == m_nIDLastMessage);

        // update right away
        CWnd* pWnd = GetMessageBar();
        if (pWnd != NULL)
            pWnd->UpdateWindow();
        }
        else
        {
            //选中分隔栏、Popup 子菜单或者没有选中一个菜单项
            if (nItemID == 0 || nFlags & (MF_SEPARATOR|MF_POPUP))
            {
                // nothing should be displayed
                m_nIDTracking = 0;
            }
            else if (nItemID >= 0xF000 && nItemID < 0xF1F0) // max of 31 SC_s
            {
                //系统菜单的菜单项被选中
                m_nIDTracking = ID_COMMAND_FROM_SC(nItemID);
                ASSERT(m_nIDTracking >= AFX_IDS_SCFIRST &&
                m_nIDTracking < AFX_IDS_SCFIRST + 31);
            }
            else if (nItemID >= AFX_IDM_FIRST_MDICHILD)
            {
                //如果选中的菜单项表示一个 MDI 子窗口
                m_nIDTracking = AFX_IDS_MDICHILD;
            }
            else
            {
                //选中了一个菜单项
                m_nIDTracking = nItemID;
            }
            pFrameWnd->m_nFlags |= WF_NOPOPMSG;
        }

    // when running in-place, it is necessary to cause a message to
    //    be pumped through the queue.
    if (m_nIDTracking != m_nIDLastMessage && GetParent() != NULL)
        PostMessage(WM_KICKIDLE);
    }
```

OnMenuSelect 的作用在于跟踪当前选中的菜单项，把菜单项对应的 ID 保存在 CFrameWnd 的成员变量 m_nIDTracking 中。

如果菜单项没有选中，或者选中的是一个子菜单，则设置 nIDTracking 为 0。
如果选中的是系统菜单，则把系统菜单 ID 转换成一个对应的命令 ID；保存该值到 nIDTracking 中。
如果选中的菜单是 MDI 子窗口创建时添加的（用来表示 MDI 子窗口），则转换菜单 ID 为 AFX_IDS_MDICHILD，所有对应 MDI 子窗口的菜单项都使用 AFX_IDS_MDICHILD，保存该值到 nIDTracking 中。
其他情况，就是选中菜单项的 ID，把它保存到 nIDTracking 中。

跟踪被选择的菜单项并保存其 ID 在 m_nIDTracking 中，OnEnterIdle 将用到 m_nIDTracking。OnEnterIlde 是消息 WM_ENTERIDLE 的处理函数，CFrameWnd 的实现如下。

```
void CFrameWnd::OnEnterIdle(UINT nWhy, CWnd* pWho)
{
CWnd::OnEnterIdle(nWhy, pWho);

//若不是因为菜单选择进入该函数
//或者当前跟踪到的菜单项 ID 是最近一次处理的，则返回
if (nWhy != MSGF_MENU || m_nIDTracking == m_nIDLastMessage)
    return;

//将发送消息 WM_SETMESSAGETEXT
//在状态栏显示 m_nIDTracking 对应的字符串
SetMessageText(m_nIDTracking);
ASSERT(m_nIDTracking == m_nIDLastMessage);
}
```

当一个对话框或者菜单被显示的时候，Windows 发送 WM_ENTERIDLE 消息。消息参数 wParam 取值为 MSGF_DIALOGBOX 或者 MSGF_MENU。前者表示显示对话框时发送该消息，这时消息参数 lParam 表示对话框的句柄；后者表示显示菜单时发送该消息，这时消息参数 lParam 表示菜单的句柄。
经过消息映射，wParam 的值传递给 OnEnterIdle 的参数 nWhy，参数 lParam 的值传给参数 who。如果参数 1 取值为 MSGF_MENU，并且 OnEnterIdle 最近一次在菜单显示被调用时的菜单 ID 不同于这一次，则调用 SetMessageText 在状态栏显示对应 ID 命令的字符串，并且记录当前菜单 ID 到变量 m_nIDTracking 中（见消息处理函数 OnSetMessageText）。
这样，在菜单选择期间，用户选择的菜单项 ID 被 OnMenuSelect 记录，在消息 WM_ENTERIDLE 处理时在状态栏显示 ID 命令的提示。

13.2.4.3 控制条的消息分发处理

工具条（包括对话框工具条）是一个子窗口，它们可以响应各种消息。如果按标准的 Windows 消息和命令消息的分发途径，一些消息不能送到拥有工具条的边框窗口，因为这些消息都将被工具条（对话框工具条）处理掉。所以，CControlBar 覆盖了虚拟函数 PreTranslateMessage 和 WindowProc 以便实现特定的消息分发路径。
（1） WindowProc
CControlBar 的 WindowProc 实现了如下的消息分发路径：

用户对控制条的输入消息或者分发给 CControlBar 及其派生类处理，或者送给拥有控制条的边框窗口处理，或者送给 Windows 控制“窗口类”的窗口过程处理。
WindowProc 的实现如下：

```
LRESULT CControlBar::WindowProc(UINT nMsg,
        WPARAM wParam, LPARAM lParam)
{
ASSERT_VALID(this);

LRESULT lResult;
switch (nMsg)
{
        //本函数处理以下消息
        case WM_NOTIFY:
        case WM_COMMAND:
        case WM_DRAWITEM:
        case WM_MEASUREITEM:
        case WM_DELETEITEM:
        case WM_COMPAREITEM:
        case WM_VKEYTOITEM:
        case WM_CHARTOITEM:
        //首先，工具条处理上述消息，如果没有处理，则接着给所属边框窗口处理
        if (OnWndMsg(nMsg, wParam, lParam, &lResult))
                return lResult;
        else
                return GetOwner()->SendMessage(nMsg, wParam, lParam);
        }
}
// 最后，给基类 CWnd，按缺省方式处理
lResult = CWnd::WindowProc(nMsg, wParam, lParam);
return lResult;
}
```

从上述实现可以看出，对于 case 范围内的一些消息，如 WM_COMMAND、WM_NOTIFY 等，控制条如果不能处理，则优先分发给其父窗口（边框窗口）处理，然后进入缺省处理，对于其他消息直接调用基类 CWnd 的实现（缺省处理）。基于这样的机制，可以把用户对工具条按钮或者对话框工具条内控制的操作解释成相应的命令消息，执行对应的命令处理。
对于工具条，当用户选中某个按钮时（鼠标左键弹起，消息是 WM_LBUTTONUP），工具条窗口接收到 WM_LBUTTONUP 消息，该消息不在 CControlBar::WindowProc 特别处理的消息范围内，于是进行缺省处理，也就是说，把该消息派发给控制条对应的 Windows 控制的窗口过程处理（即被 MFC 统一窗口过程所取代的原窗口过程），该窗口过程则把该消息转换成一条命令消息 WM_COMMAND，命令 ID 就是选中按钮对应的 ID，然后，发送该命令消息给拥有工具条的边框窗口，导致相应的命令处理函数被调用。
对于对话框工具条，当工具条的某个控制子窗口被选中之后，则产生一条命令通知消息 WM_COMMAND，wParam 是控制子窗口的 ID。CControlBar::WindowProc 处理该消息。WindowProc 首先调用 OnWndMsg 把消息发送给对话框工具条或者对话框工具条的基类处

理，如果没有被处理的话，则 OnWndMsg 返回 FALSE。接着，WindowPoc 把命令消息传递给父窗口（边框窗口）处理。由于工具条的控制窗口的 ID 对应的是命令 ID，所以，这条 WM_COMMAND 消息传递给边框窗口时，被解释成一个命令消息，相应的命令处理函数被调用。

（2） PreTranslateMessage

CControlBar 覆盖 PreTranslateMessage 函数，主要是为了在光标落在工具条按钮上时显示 FLYBY 信息，并且让对话框工具条过滤 Dialog 消息。

```
BOOL CControlBar::PreTranslateMessage(MSG* pMsg)
{
ASSERT_VALID(this);
ASSERT(m_hWnd != NULL);

//过滤 Tooltip 消息
if (CWnd::PreTranslateMessage(pMsg))
        return TRUE;        //是 Tooltip 消息，已经被处理

UINT message = pMsg->message;
//控制条的父窗口，对工具条和对话框工具条，总是创建它的边框窗口
CWnd* pOwner = GetOwner();

//必要的话，在状态条显示工具栏按钮的提示
if (((m_dwStyle & CBRS_FLYBY) ||
        message == WM_LBUTTONDOWN || message == WM_LBUTTONUP) &&
        ((message >= WM_MOUSEFIRST && message <= WM_MOUSELAST) ||
          (message >= WM_NCMOUSEFIRST &&
              message <= WM_NCMOUSELAST)))
{
        _AFX_THREAD_STATE* pThreadState = AfxGetThreadState();

        //确认鼠标在工具栏的哪个按钮上
        CPoint point = pMsg->pt;
        ScreenToClient(&point);
        TOOLINFO ti; memset(&ti, 0, sizeof(TOOLINFO));
        ti.cbSize = sizeof(TOOLINFO);
        int nHit = OnToolHitTest(point, &ti);
        if (ti.lpszText != LPSTR_TEXTCALLBACK)
                free(ti.lpszText);
        BOOL bNotButton =
        message == WM_LBUTTONDOWN && (ti.uFlags & TTF_NOTBUTTON);
        if (message != WM_LBUTTONDOWN && GetKeyState(VK_LBUTTON) < 0)
                nHit = pThreadState->m_nLastStatus;

        //更新状态栏的提示信息
        if (nHit < 0 || bNotButton)
```

```
        {
            if (GetKeyState(VK_LBUTTON) >= 0 || bNotButton)
            {
            SetStatusText(-1);
            KillTimer(ID_TIMER_CHECK);
            }
        }
        else
        {
            if (message == WM_LBUTTONUP)
            {
                SetStatusText(-1);
                ResetTimer(ID_TIMER_CHECK, 200);
            }
        else
        {
            if ((m_nStateFlags & statusSet) || GetKeyState(VK_LBUTTON) < 0)
            SetStatusText(nHit);
        else if (nHit != pThreadState->m_nLastStatus)
            ResetTimer(ID_TIMER_WAIT, 300);
        }
}
pThreadState->m_nLastStatus = nHit;
}

// don't translate dialog messages when in Shift+F1 help mode
CFrameWnd* pFrameWnd = GetTopLevelFrame();
if (pFrameWnd != NULL && pFrameWnd->m_bHelpMode)
        return FALSE;

//在 IsDialogMessage 之前调用边框窗口的 PreTranslateMessage，
//给边框窗口一个处理快捷键的机会
while (pOwner != NULL)
{
        // allow owner & frames to translate before IsDialogMessage does
        if (pOwner->PreTranslateMessage(pMsg))
            return TRUE;

        // try parent frames until there are no parent frames
        pOwner = pOwner->GetParentFrame();
}

//过滤给对话框的消息和来自子窗口的消息
return PreTranslateInput(pMsg);
```

```
}
```

函数 PreTranslateMessage 主要是针对工具栏的，用来处理工具栏的 CBRS_FLYBY 特征。对于对话框工具栏，也可以有 CBRS_FLYBY 特征。但在这种情况下，还需要把一些用户键盘输入解释成对话框消息。为了防止快捷键被解释成对话框消息，在调用函数 PreTranslateInput 之前，必须调用所有父边框窗口的 PreTranslateMessage，给父边框窗口一个机会处理用户的输入消息，判断快捷键是否被按下。
关于 Tooltip 和本 PreTranslateMessage 函数处理 Tooltip 的详细解释见下一节的讨论。

13.2.4.4 Tooltip

工具条（或对话框工具条）如果指定了 CBRS_TOOLTIPS 风格（创建时指定或者创建后用 SetBarStyle 设置），则当鼠标落在某个按钮上（或者对话框的子控制窗口）时，在鼠标附近弹出一个文本框——Tooltip 提示窗口。
如果还指定了 CBRS_FLYBY 风格，则还在状态栏显示和按钮（或子控制窗口）ID 对应的字符串信息。当然，鼠标左键在某个按钮（或子控制窗口）按下时，也要在状态栏显示按钮的提示信息，当左键弹起时，则重置状态栏的状态。
如前所述，Tooltip 窗口是 Windows 控制窗口。MFC 使用了 CToolTipCtrl 类封装 Tooltip 的 HWND 窗口。在一个线程的生存期间，至多拥有一个 Tooltip 窗口，该窗口对象的指针保存在线程状态的成员变量 m_pToolTip 中。线程状态类 AFX_THREAD_STATE 的析构函数如果检测到 m_pToolTip，则销毁 MFC 窗口对象和相应的 Windows 窗口对象。
（1） CWnd 对 Tooltip 消息的预处理
为了支持 Tooltip 显示，CWnd 提供了以下函数：EnableTooltip，CancelTooltip，PreTranslateMessage，FilterTooltipMessage，OnToolHitTest。
EnableTooltip 设置 CBRS_TOOLTIP 风格，相反 CancelTootip 取消这种风格。
PreTranslateMessage 调用了 FilterTooltipMessage 过滤 Tooltip 消息。
OnToolHitTest 是一个由 CWnd 定义的虚拟函数。CToolBar 通过覆盖该函数，来检测对话框工具栏的控制子窗口或者工具栏按钮是否被选中、哪个被选中。

CWnd 的 PreTranslateMessage 在 4.5 节讨论过，它的实现如下：

```
BOOL CWnd::PreTranslateMessage(MSG* pMsg)
{
//处理 Tooltip 消息
AFX_MODULE_STATE* pModuleState = _AFX_CMDTARGET_GETSTATE();
if (pModuleState->m_pfnFilterToolTipMessage != NULL)
//导致调用 FilterTooltipMessage
(*pModuleState->m_pfnFilterToolTipMessage)(pMsg, this);

    //不是 Tooltip 消息
    return FALSE;
}
```

至于为什么 MFC 在模块状态中保存一个处理 Tooltip 消息的函数地址，通过该函数调用 FilterTooltipMessage，是因为 Tooltip 窗口是模块线程局部有效的。

FilterTooltipMessage 检测是否是 Tooltip 消息。如果是，在必要时创建一个 CTooltipCtrl 对象和对应的 HWND，调用 OnToolHitTest 确定被选中的按钮或者控制的 ID，接着弹出 Tooltip 窗口。

其他函数和 CTooltipCtrl 这里不作详细论述了。

（2） 处理 TTN_NEEDTEXT 通知消息

Tooltip 窗口在弹出之前，它给工具条（或者对话框工具栏）的父窗口发送通知消息 TTN_NEEDTEXT，请求得到要显示的文本。

CFrameWnd 类处理了 TTN_NEEDTEXT 通知消息，消息处理函数是 OnToolTipText。

消息映射的定义：

```
ON_NOTIFY_EX_RANGE(TTN_NEEDTEXTW, 0, 0xFFFF, OnToolTipText)
ON_NOTIFY_EX_RANGE(TTN_NEEDTEXTA, 0, 0xFFFF, OnToolTipText)
```

这里，使用了扩展消息映射宏把子窗口 ID 在 0 和 0xFFFF 之间的控制条窗口的通知消息 TTN_NEEDTEXTA 和 TTN_NEEDTEXTW 映射到函数 OnToolTipText。

消息映射的实现：

```
BOOL CFrameWnd::OnToolTipText(UINT, NMHDR* pNMHDR, LRESULT* pResult)
{
ASSERT(pNMHDR->code == TTN_NEEDTEXTA ||
      pNMHDR->code == TTN_NEEDTEXTW);

//让上一层的边框窗口优先处理该消息
if (GetRoutingFrame() != NULL)
      return FALSE;

//分 ANSI and UNICODE 两个处理版本
TOOLTIPTEXTA* pTTTA = (TOOLTIPTEXTA*)pNMHDR;
TOOLTIPTEXTW* pTTTW = (TOOLTIPTEXTW*)pNMHDR;
TCHAR szFullText[256];
CString strTipText;
UINT nID = pNMHDR->idFrom;
//如果 idFrom 是一个子窗口，则得到其 ID。
if (pNMHDR->code == TTN_NEEDTEXTA &&
      (pTTTA->uFlags & TTF_IDISHWND) ||
      pNMHDR->code == TTN_NEEDTEXTW &&
       (pTTTW->uFlags & TTF_IDISHWND))
{
      //idFrom 是工具条的句柄
      nID = _AfxGetDlgCtrlID((HWND)nID);
}

if (nID != 0) //若是 0，为一分隔栏，不是按钮
{
      //得到 nID 对应的字符串
      AfxLoadString(nID, szFullText);
      //从上面得到的字符串中取出 Tooltip 使用的文本
```

```
        AfxExtractSubString(strTipText, szFullText, 1, '\n');
    }
    //复制分离出的文本
    #ifndef _UNICODE
        if (pNMHDR->code == TTN_NEEDTEXTA)
        lstrcpyn(pTTTA->szText, strTipText, _countof(pTTTA->szText));
    else
        _mbstowcsz(pTTTW->szText, strTipText, _countof(pTTTW->szText));
    #else
        if (pNMHDR->code == TTN_NEEDTEXTA)
        _wcstombsz(pTTTA->szText, strTipText, _countof(pTTTA->szText));
    else
        lstrcpyn(pTTTW->szText, strTipText, _countof(pTTTW->szText));
    #endif
    *pResult = 0;

    //显示 Tooltip 窗口
    ::SetWindowPos(pNMHDR->hwndFrom, HWND_TOP, 0, 0, 0, 0,
        SWP_NOACTIVATE|SWP_NOSIZE|SWP_NOMOVE);

    return TRUE;      //消息处理完毕
    }
```

OnToolTipText 是一个扩展映射宏定义的消息处理函数，所以有一个 UINT 参数并且返回 BOOL 类型的值。不过，由于第二个参数（NMHDR）的 idFrom 域包含了有关信息，所以第一个 UINT 类型的参数没有用上。

OnToolTipText 也是一个处理通知消息的例子。其中，通知参数 wParam 的结构如 4.4.4.2 节所述，具体如下：

```
typedef struct {
    NMHDR        hdr;   //WM_NOTIFY 消息要求的头
    LPTSTR       lpszText;    //接收工具条按钮对应文本的缓冲区
    WCHAR        szText[80]; //接收 Tooltip 显示文本的缓冲区
    HINSTANCE hinst;          //包含了 szText 的实例句柄
    UINT         uflags;       //标识了 NMHDR 的 idFrom 成员的意义
} TOOLTIPTEXT, FAR *LPTOOLTIPTEXT;
```

uflags 如果等于 TTF_IDISHWND，则表示通知消息来自对话框工具条的一个子窗口，而不是包含工具条按钮。

OnToolTipText 根据子窗口 ID 或者工具条按钮对应的 ID，得到字符串 ID。如前所述，字符串 ID 由两部分组成，第二部分用于 Tooltip 显示，分隔符号是“\n”。根据这种格式 OnToolTipText 分离出 Tooltip 文本。

得到了 Tooltip 文本之后，可以有三种方法返回文本信息：把文本信息复制到 szText 缓冲区；把文本地址复制到 lpszText；复制字符串资源的 ID 到 lpszText、复制包含资源的实例句柄到 hint。本函数采用了第一种方法。

在得到了返回的 Tooltip 文本之后，该文本在 Tooltip 窗口中被显示出来。

其他的 OnToolHist 等函数的实现不作详细的解释了。下面，讨论 CBRS_FLYBY 风格的实现。

（3） CBRS_FLYBY 风格的实现

CBRS_FLYBY 是 MFC 提供的特征。当鼠标落在工具条按钮（或者对话框工具条的子窗口）上且稳定 300ms 后，在状态栏显示对应的提示信息。如果选中某个按钮或者子窗口（鼠标左键按下），则在相应命令消息处理之前在状态栏显示有关提示信息，之后（鼠标左键弹起），重新设置状态栏的状态信息。

为了支持这种特征，CControlBar 覆盖虚拟函数 PreTranslateMessage 来处理和 CBRS_FLYBY 相关的消息，该函数前面已经讨论过，这里解释它如何处理 CBRS_FLYBY 特征。

如果同时具备

条件 1：控制条具有 CBRS_FLYBY 特征或者当前消息是 WM_LBUTTONUP 或者 WM_LBUTTONDOWN。

条件 2：当前消息是鼠标消息（在 WM_MOUSEFIRST 和 WM_MOUSELAST 之间或者在 WM_NCMOUSEFIRST 和 WM_NCMOUSELAST 之间）。

则进行 FLYBY 处理。

首先，调用 OnToolHitTest 测试用户是否选中了工具条的按钮或者子窗口；

如果没有按钮或者子窗口被选中，则重新设置状态栏的状态，取消曾经设置的 Check 定时器。重置状态栏的状态时调用了 SetStatusText（int nHit）函数，它是 CControlBar 内部使用的函数，若 nHit 等于-1，它向父窗口发送 WM_POPMESSAGETEXT，消息参数是 AFX_IDS_IDLEMESSAGE，结果导致状态栏显示“Ready”字样；否则发送 WM_SETMESSAGETEXT 消息，wParm 设置为 nHit，结果导致在状态栏显示 ID 为 nHit 的字符串。

如果有按钮或者子窗口被选中，若左键弹起，则重新设置状态栏信息，取消 Wait 定时器，并重新设置 Check 定时器，定时是 200ms；若左键按下，则在状态栏显示消息 ID 对应的提示信息；若是其他鼠标消息，如果当前鼠标所在按钮（子窗口）不同于最近一次，则取消 Check 定时器，重新设置 Wait 定时器，定时 300 毫秒。

CControlBar 覆盖了消息处理函数 OnTimer，在指定时间之后，检查鼠标位置，如果鼠标还在某个按钮或者子窗口上，则在状态条显示提示信息。Wait 定时器在等待之后准备在状态条显示信息，触发一次后被取消；Check 定时器在等待之后，判断是否需要取消状态条当前显示的信息，重新设置状态条，若这样的话，同时也取消 Check 定时器。

注意，这些鼠标消息被处理之后，并没有终止，它们将继续被发送给控制条的窗口过程处理。

至此，CBRS_FLYBY 特征的支持实现描述完毕。

13.2.4.5 禁止和允许

在 MFC 下，工具条、状态条还有一个重要的特征，就是自动地根据条件禁止或者允许使用某个按钮、窗格等。在 4.4.5 节命令消息的处理中，曾详细讨论了其实现原理，现在，详细地分析所涉及函数是如何实现的。有关的消息处理函数和虚拟函数如下。

处理 WM_INITIALUPDATE 消息的 OnInitialUpdate；

处理 WM_IDLEUPDATECMDUI 消息的 OnIdleUpdateCmdUI；

虚拟函数 OnUpdateCmdUI。

回顾 5.3.3.5 节，在边框窗口的创建之后，给所有的子窗口发送初始化消息，控制子窗口用 OnInitialUpdate 响应它，调用 OnIdleUpdateCmdUI 完成状态的初始化。

OnIdleUpdateCmdUI 还在 IDLE 处理时进行状态的更新处理，它生成用于处理状态更新消息的命令目标 pTarget，然后调用虚拟函数 OnUpdateCmdUI(pTarget，…)来更新工具栏或者状

态栏的状态。

CControlBar 的子类都实现了自己的 OnUpdateCmdUI 函数，用该函数生成适当的 CCmdUI 对象 state，然后调用 CCmdUI 的 DoUpdate（pTarget，…）给 pTarget 所指对象发送状态更新消息。为了完成具体的状态更新，从 CCmdUI 派生出 CToolCmdUI 和 CStatusCCmUI，它们实现了自己的 Enable、SetCheck 等等。

（1） 初始化控制窗口

CControlBar 使用 OnInitialUpdate 消息处理函数初始化控制窗口的状态。

```
void CControlBar::OnInitialUpdate()
{
//在窗口显示之前，更新状态
OnIdleUpdateCmdUI(TRUE, 0L);
}
```

CControlBar 实现了 OnInitialUpdate 函数，通过它来处理 WM_INITIALUPDATE 消息。各个子类不必覆盖该消息处理函数。

（2） 处理 Idle 消息更新工具条状态

CControlBar 使用 OnIdleUpdateCmdUI 消息处理函数处理 IDLE 消息。

```
LRESULT CControlBar::OnIdleUpdateCmdUI(WPARAM wParam, LPARAM)
{
// handle delay hide/show
BOOL bVis = GetStyle() & WS_VISIBLE;
UINT swpFlags = 0;
if ((m_nStateFlags & delayHide) && bVis)
        swpFlags = SWP_HIDEWINDOW;
else if ((m_nStateFlags & delayShow) && !bVis)
        swpFlags = SWP_SHOWWINDOW;
m_nStateFlags &= ~(delayShow|delayHide);
if (swpFlags != 0)
{
        SetWindowPos(NULL, 0, 0, 0, 0, swpFlags|
        SWP_NOMOVE|SWP_NOSIZE|SWP_NOZORDER|SWP_NOACTIVATE);
}

// the style must be visible and if it is docked
// the dockbar style must also be visible
if ((GetStyle() & WS_VISIBLE) &&
        (m_pDockBar == NULL || (m_pDockBar->GetStyle() & WS_VISIBLE)))
{
        //得到父边框窗口，状态更新消息将发送给它
        CFrameWnd* pTarget = (CFrameWnd*)GetOwner();
        if (pTarget == NULL || !pTarget->IsFrameWnd())
        pTarget = GetParentFrame();
        if (pTarget != NULL)
                OnUpdateCmdUI(pTarget, (BOOL)wParam);
}
```

```
return 0L;
}
```

OnIdleUpdateCmdUI 或者在初始化时被 OnInitialUpdate 调用，或者作为消息处理函数来处理 WM_IDLEUPDATECMDUI 消息。

CControlBar 实现了 OnIdleUpdateCmdUI 函数，把具体的用户界面更新动作委托给虚拟函数 OnUpdateCmdUI 完成。

由于各个用户界面的特殊性，所以 CControlBar 本身没有实现 OnUpdateCmdUI，而是留给各个派生类去实现。例如，CToolBar 覆盖了 OnUpdateCmdUI，其实现如下：

```
void CToolBar::OnUpdateCmdUI(CFrameWnd* pTarget, BOOL bDisableIfNoHndler)
{
//定义一个 CCmdUI 对象，CToolCmdUI 派生于 CCmdUI
CToolCmdUI state;
//给 CCmdUI 的各个成员赋值
state.m_pOther = this;

//得到总的按钮数目
state.m_nIndexMax = (UINT)DefWindowProc(TB_BUTTONCOUNT, 0, 0);
//逐个按钮进行状态更新
for (state.m_nIndex = 0; state.m_nIndex < state.m_nIndexMax; state.m_nIndex++)
{
    //获取按钮状态信息
    TBBUTTON button;
    _GetButton(state.m_nIndex, &button);
    //得到按钮的 ID
    state.m_nID = button.idCommand;

    // ignore separators
    if (!(button.fsStyle & TBSTYLE_SEP))
    {
        //优先让 CToolBar 对象处理状态更新消息
        if (CWnd::OnCmdMsg(state.m_nID,
            CN_UPDATE_COMMAND_UI, &state, NULL))
            continue;//处理了更新消息，更新下一个按钮

        //CToolBar 没有处理，将发送给 pTarget 处理状态更新消息
        //第二个参数 bDisableIfNoHndler 往下传
        state.DoUpdate(pTarget, bDisableIfNoHndler);
    }
}

//更新加到控制条中的对话框控制的状态
UpdateDialogControls(pTarget, bDisableIfNoHndler);
}
```

CToolBar 的 OnUpdateCmdUI 函数完成工具条按钮的状态更新。它接受两个参数，参数 1 表

示接收状态更新命令消息的对象，由 CControlBar 的函数 OnIdleUpdateCmdUI 传递过来，一般是边框窗口对象；参数 2 表示如果某条命令消息没有处理函数时，对应的用户接口对象是否被禁止。

OnUpdateCmdUI 通过发送状态更新通知消息，逐个更新按钮的状态。更新消息首先让工具条对象处理，如果没有处理的话，送给边框窗口对象处理，导致状态更新命令消息的处理函数被调用，参见 4.4.5 节。

CStatusBar 的 OnUpdateCmdUI 类似于此。

CDialogBar 的 OnUpdateCmdUI 则调用了虚拟函数 UpdateDialogControls 来进行状态更新，CWnd 提供了该函数的实现，过程类似于 CToolBar 的函数 OnUpdateCmdUI。

（3） 菜单项的自动更新

那么，菜单项的自动更新如何实现的呢？OnInitMenuPopup 在菜单项状态的自动更新中曾经被提到，其实现如下：

```
void CFrameWnd::OnInitMenuPopup(CMenu* pMenu, UINT, BOOL bSysMenu)
{
AfxCancelModes(m_hWnd);

if (bSysMenu)
        return;         // don't support system menu

ASSERT(pMenu != NULL);
// check the enabled state of various menu items

CCmdUI state;
state.m_pMenu = pMenu;
ASSERT(state.m_pOther == NULL);
ASSERT(state.m_pParentMenu == NULL);

//判断菜单是否在顶层菜单(top level menu)中弹出，如果这样
//则设置 m_pParentMenu 指向顶层菜单，否则 m_pParentMenu
//为空，表示它是一个二级弹出菜单
HMENU hParentMenu;
//是否是浮动式的弹出菜单（floating pop up menu）
if (AfxGetThreadState()->m_hTrackingMenu == pMenu->m_hMenu)
        state.m_pParentMenu = pMenu;        // parent == child for tracking popup
else if ((hParentMenu = ::GetMenu(m_hWnd)) != NULL)//
{
        CWnd* pParent = GetTopLevelParent();
        // child windows don't have menus -- need to go to the top!
        //得到顶层窗口的菜单
        if (pParent != NULL &&
                (hParentMenu = ::GetMenu(pParent->m_hWnd)) != NULL)
        {
                int nIndexMax = ::GetMenuItemCount(hParentMenu);
                //确定顶层窗口的菜单是否包含本菜单项
```

```
            for (int nIndex = 0; nIndex < nIndexMax; nIndex++)
            {
                if (::GetSubMenu(hParentMenu, nIndex) == pMenu->m_hMenu)
                {
                    //顶层窗口菜单是本菜单的父菜单
                    state.m_pParentMenu = CMenu::FromHandle(hParentMenu);
                    break;
                }
            }
        }
    }

    //本菜单的菜单项(menu item)数量
    state.m_nIndexMax = pMenu->GetMenuItemCount();
    //对所有菜单项逐个进行状态更新
    for (state.m_nIndex = 0; state.m_nIndex < state.m_nIndexMax;
       state.m_nIndex++)
    {
        state.m_nID = pMenu->GetMenuItemID(state.m_nIndex);
        if (state.m_nID == 0)
            continue; // menu separator or invalid cmd - ignore it

        ASSERT(state.m_pOther == NULL);
        ASSERT(state.m_pMenu != NULL);
        if (state.m_nID == (UINT)-1)
        {
            // 可能是一个 popup 菜单，得到其第一个子菜单项目
            state.m_pSubMenu = pMenu->GetSubMenu(state.m_nIndex);
            if (state.m_pSubMenu == NULL ||
                (state.m_nID = state.m_pSubMenu->GetMenuItemID(0)) == 0 ||
                state.m_nID == (UINT)-1)
            {
                continue;           // 找不到 popup 菜单的子菜单项
            }
            //popup 菜单不会被自动的禁止
            state.DoUpdate(this, FALSE);
        }
        else
        {
            //正常的菜单项，若边框窗口的 m_bAutoMenuEnable 设置为
            //TURE 且菜单项非系统菜单，则自动 enable/disable 该菜单项
            state.m_pSubMenu = NULL;
            state.DoUpdate(this, m_bAutoMenuEnable && state.m_nID < 0xF000);
        }
```

```
        //经过菜单状态的更新处理，可能增加或删除了一些菜单项
        UINT nCount = pMenu->GetMenuItemCount();
        if (nCount < state.m_nIndexMax)
        {
            state.m_nIndex -= (state.m_nIndexMax - nCount);
            while (state.m_nIndex < nCount &&
                pMenu->GetMenuItemID(state.m_nIndex) == state.m_nID)
            {
                state.m_nIndex++;
            }
        }
        state.m_nIndexMax = nCount;
        }
    }
```

菜单弹出之前，发送 WM_INITMENUPOPUP 消息，OnInitMenuPopup 消息处理函数被调用，逐个更新菜单项目（menu item）的状态。程序员可以处理它们对应的状态更新消息，禁止/允许菜单项目被使用（disable/enable)，在菜单项目上打钩或者取消（checked/unchecked），等等。

13.2.4.6 显示或者隐藏工具栏和状态栏

这里讨论显示或者隐藏工具栏、状态栏的操作，以及工具栏、状态栏被显示/隐藏时，相关的两个菜单项 ID_VIEW_STATUS_BAR、ID_VIEW_TOOLBAR 的状态更新。这两个菜单命令及对应的状态更新命令是标准命令消息所包含的。MFC 边框窗口实现了菜单命令消息的处理和菜单项状态的更新。

CFrameWnd 提供了 OnBarCheck 来响应与 ID_VIEW_STATUS_BAR、ID_VIEW_TOOLBAR 菜单项对应的命令。

消息映射：

```
ON_COMMAND_EX(ID_VIEW_STATUS_BAR, OnBarCheck)
ON_COMMAND_EX(ID_VIEW_TOOLBAR, OnBarCheck)
```

这里，使用了扩展命令消息映射宏把 ID_VIEW_STATUS_BAR 和 ID_VIEW_TOOLBAR 命令映射给同一个函数 OnBarCheck 处理。

OnBarCheck 函数的实现：

```
BOOL CFrameWnd::OnBarCheck(UINT nID)
{
ASSERT(ID_VIEW_STATUS_BAR == AFX_IDW_STATUS_BAR);
ASSERT(ID_VIEW_TOOLBAR == AFX_IDW_TOOLBAR);

//得到工具条或者状态条
CControlBar* pBar = GetControlBar(nID);
if (pBar != NULL)
{
    //若控制条可见，则隐藏它；否则，显示它
```

```
        ShowControlBar(pBar, (pBar->GetStyle() & WS_VISIBLE) == 0, FALSE);
        //处理完毕
        return TRUE;
    }
    //可以让下一个命令目标继续处理
    return FALSE;
    }
```

由于是扩展映射宏定义的消息处理函数，所以 OnBarCheck 函数有一个 UINT 类型的参数和一个 BOOL 返回值。

当用户从“View”菜单选择打了钩的“Toolbar”时，消息处理函数 OnBarCheck 被调用，参数就是菜单项的 ID 号 ID_VIEW_TOOLBAR，它等于工具条的子窗口 IDAFX_IDW_TOOLBAR。处理结果，工具条被隐藏；当再次选择该菜单项则工具条被显示。处理状态条的过程类似于工具条的处理。

ShowControlBar 是 CFrameWnd 的成员函数，参数 1 表示控制条对象指针，参数 2 表示显示(TRUE)或者隐藏(FALSE)，参数 3 表示是立即显示（FALSE）或者延迟显示（TRUE）。

如果工具条或者状态条被隐藏，则相应的菜单项 ID_VIEW_STATUS_BAR 或者 ID_VIEW_TOOLBAR 变成 uncheked（菜单项被标记为没有选择），否则，checked（菜单项被标记选择）。CFrameWnd 实现了这两个菜单项的状态更新处理，列举其中一个如下：

声明处理 ID_VIEW_TOOLBAR 的状态更新消息：

```
    ON_UPDATE_COMMAND_UI(ID_VIEW_TOOLBAR, OnUpdateControlBarMenu)
```

函数的实现：

```
    void CFrameWnd::OnUpdateControlBarMenu(CCmdUI* pCmdUI)
    {
    ASSERT(ID_VIEW_STATUS_BAR ==
    AFX_IDW_STATUS_BAR);
    ASSERT(ID_VIEW_TOOLBAR == AFX_IDW_TOOLBAR);

    CControlBar* pBar = GetControlBar(pCmdUI->m_nID);
    //存在工具栏
    if (pBar != NULL)
    {
        //工具条窗口被显示则 checked，被隐藏则 uncheked
        pCmdUI->SetCheck((pBar->GetStyle() & WS_VISIBLE) != 0);
        return;
    }
    pCmdUI->ContinueRouting();
    }
```

GetControlBar 是 CFrameWnd 的成员函数，用来返回边框窗口的指定 ID 的控制条对象（指定 ID 是控制条的子窗口 ID）。

13.2.5 泊位和漂浮

工具条可以泊位在边框窗口的任一边(上、下、左、右)，或者漂浮在屏幕上的任何地方。

（1） 实现泊位的方法

首先，边框窗口调用 CFrameWnd::EnableDocking 函数使控制条泊位在边框窗口中有效，指明在边框窗口的哪边接受泊位。如果想在任何边都可以泊位，则使用参数 CBRS_ALIGN_ANY。

然后，工具条调用 ControlBar::EnableDocking 使泊位对工具条有效，如果在调用 ControlBar::EnableDocking 时指定的泊位目的边和边框窗口能够泊位的边不符合，那么工具条不能泊位，它将漂浮。

最后，边框窗口调用 CFrameWnd::DockControlBar 泊位工具条。

（2） 泊位后形成窗口层次关系

边框窗口、泊位条、工具条的包含关系如下：

边框窗口

泊位条 1

工具条 1

工具条 2

…

泊位条 2

…

边框窗口包含 1 到 4 个泊位条子窗口，每个泊位条包含若干个控制条子窗口。

（3） 泊位的实现

CFrameWnd::EnableDocking 指定哪边接受泊位，则为泊位准备一个泊位条。泊位条用 CDockBar 描述，派生于 CControlBar。如果指定任何边都可以泊位，则创建四个 CDockBar 对象和对应的 HWND 窗口。然后，调用 ControlBar::EnableDocking 在对应的泊位条内安置工具条。

MFC 设计了 CDockBar 类和 CFrameWnd 的一些函数来实现泊位，具体代码实现在此不作详细讨论。

（4） 实现漂浮工具条的方法：

边框窗口调用 FloatControlBar 实现工具条的漂浮。

（5） 漂浮的实现：

首先，创建一个微型漂浮边框窗口，该边框窗口有一个泊位条。

然后，在微型边框窗口的泊位条内放置工具条。

MFC 设计了微型边框类 CMiniFrameWnd，在此基础上派生出微型泊位边框窗口类 CMiniDockFrameWnd。CMiniDockFrameWnd 增加了一个 CDockBar 类型成员变量 m_wndDockBar，即泊位条。

在 CMiniDockFrameWnd 对象被创建时，创建泊位条 m_wndDockBar。泊位条 m_wndDockBar 的父窗口如同 CMiniDockFrameWnd 的父窗口一样，是调用 FloatControlBar 的边框窗口，而不是微型泊位边框窗口。微型边框窗口和泊位条创建完成之后，调用 ControlBar::DockControlBar 泊位工具条在 CMiniDockFrameWnd 窗口。

具体的代码实现略。

第14章　SOCKET类的设计和实现

14.1　WinSock基本知识

这里不打算系统地介绍socket或者WinSock的知识。首先介绍WinSock API函数，讲解阻塞/非阻塞的概念；然后介绍socket的使用。

14.1.1 WinSock API

Socket接口是网络编程（通常是TCP/IP协议，也可以是其他协议）的API。最早的Socket接口是Berkeley接口，在Unxi操作系统中实现。WinSock也是一个基于Socket模型的API，在Microsoft Windows操作系统类中使用。它在Berkeley接口函数的基础之上，还增加了基于消息驱动机制的Windows扩展函数。Winscok1.1只支持TCP/IP网络，WinSock2.0增加了对更多协议的支持。这里，讨论TCP/IP网络上的API。
Socket接口包括三类函数：
第一类是WinSock API包含的Berkeley socket函数。这类函数分两部分。第一部分是用于网络I/O的函数，如
accept、Closesocket、connect、recv、recvfrom、Select、Send、Sendto
另一部分是不涉及网络I/O、在本地端完成的函数，如
bind、getpeername、getsockname、getsocketopt、htonl、htons、inet_addr、inet_nton
ioctlsocket、listen、ntohl、ntohs、setsocketopt、shutdow、socket等
第二类是检索有关域名、通信服务和协议等Internet信息的数据库函数，如
gethostbyaddr、gethostbyname、gethostname、getprotolbyname
getprotolbynumber、getserverbyname、getservbyport。
第三类是Berkekley socket例程的Windows专用的扩展函数，如gethostbyname对应的WSAAsynGetHostByName（其他数据库函数除了gethostname都有异步版本），select对应的WSAAsynSelect，判断是否阻塞的函数WSAIsBlocking，得到上一次Windsock API错误信息的WSAGetLastError，等等。
从另外一个角度，这些函数又可以分为两类，一是阻塞函数，一是非阻塞函数。所谓阻塞函数，是指其完成指定的任务之前不允许程序调用另一个函数，在Windows下还会阻塞本线程消息的发送。所谓非阻塞函数，是指操作启动之后，如果可以立即得到结果就返回结果，否则返回表示结果需要等待的错误信息，不等待任务完成函数就返回。
首先，异步函数是非阻塞函数；
其次，获取远地信息的数据库函数是阻塞函数（因此，WinSock提供了其异步版本）；
在Berkeley socket函数部分中，不涉及网络I/O、本地端工作的函数是非阻塞函数；
在Berkeley socket函数部分中，网络I/O的函数是可阻塞函数，也就是它们可以阻塞执行，也可以不阻塞执行。这些函数都使用了一个socket，如果它们使用的socket是阻塞的，则这些函数是阻塞函数；如果它们使用的socket是非阻塞的，则这些函数是非阻塞函数。
创建一个socket时，可以指定它是否阻塞。在缺省情况下，Berkerley的Socket函数和WinSock都创建“阻塞”的socket。阻塞socket通过使用select函数或者WSAAsynSelect函数在指定

操作下变成非阻塞的。WSAAsyncSelect 函数原型如下。

```
int WSAAsyncSelect(
    SOCKET s,
    HWND hWnd,
    u_int wMsg,
    long lEvent
   );
```

其中，参数 1 指定了要操作的 socket 句柄；参数 2 指定了一个窗口句柄；参数 3 指定了一个消息，参数 4 指定了网络事件，可以是多个事件的组合，如：

FD_READ　　准备读
FD_WRITE　　准备写
FD_OOB　　带外数据到达
FD_ACCEPT　　收到连接
FD_CONNECT　　完成连接
FD_CLOSE　　关闭 socket。

用 OR 操作组合这些事件值，如 FD_READ|FD_WRITE

WSAAsyncSelect 函数表示对 socket s 监测 lEvent 指定的网络事件，如果有事件发生，则给窗口 hWnd 发送消息 wMsg。

假定应用程序的一个 socket s 指定了监测 FD_READ 事件，则在 FD_READ 事件上变成非阻塞的。当 read 函数被调用时，不管是否读到数据都马上返回，如果返回一个错误信息表示还在等待，则在等待的数据到达后，消息 wMsg 发送给窗口 hWnd，应用程序处理该消息读取网络数据。

对于异步函数的调用，以类似的过程最终得到结果数据。以 gethostbyname 的异步版本的使用为例进行说明。该函数原型如下：

```
HANDLE WSAAsyncGetHostByName(
    HWND hWnd,
    u_int wMsg,
    const char FAR *name,
    char FAR *buf,
    int buflen
   );
```

在调用 WSAAsyncGetHostByName 启动操作时，不仅指定主机名字 name，还指定了一个窗口句柄 hWnd，一个消息 ID wMsg，一个缓冲区及其长度。如果不能立即得到主机地址，则返回一个错误信息表示还在等待。当要的数据到达时，WinSock DLL 给窗口 hWnd 发送消息 wMsg 告知得到了主机地址，窗口过程从指定的缓冲区 buf 得到主机地址。

使用异步函数或者非阻塞的 socket，主要是为了不阻塞本线程的执行。在多进程或者多线程的情况下，可以使用两个线程通过同步手段来完成异步函数或者非阻塞函数的功能。

14.1.2 Socket 的使用

WinSock 以 DLL 的形式提供，在调用任何 WinSock API 之前，必须调用函数 WSAStartup 进行初始化，最后，调用函数 WSACleanUp 作清理工作。

MFC 使用函数 AfxSocketInit 包装了函数 WSAStartup，在 WinSock 应用程序的初始化函数 IninInstance 中调用 AfxSocketInit 进行初始化。程序不必调用 WSACleanUp。

Socket 是网络通信过程中端点的抽象表示。Socket 在实现中以句柄的形式被创建，包含了进行网络通信必须的五种信息：连接使用的协议，本地主机的 IP 地址，本地进程的协议端口，远地主机的 IP 地址，远地进程的协议端口。

要使用 socket，首先必须创建一个 socket；然后，按要求配置 socket；接着，按要求通过 socket 接收和发送数据；最后，程序关闭此 socket。

- 为了创建 socket，使用 socket 函数得到一个 socket 句柄：

```
socket_handle = socket(protocol_family. Socket_type, protocol);
```

其中：protocol_family 指定 socket 使用的协议，取值 PF_INET，表示 Internet(TCP/IP)协议族；Socket_type 指 socket 面向连接或者使用数据报；第三个参数表示使用 TCP 或者 UDP 协议。当一个 socket 被创建时，WinSock 将为一个内部结构分配内存，在此结构中保存此 socket 的信息，到此，socket 连接使用的协议已经确定。

- 创建了 socket 之后，配置 socket。

对于面向连接的客户，WinSock 自动保存本地 IP 地址和选择协议端口，但是必须使用 connect 函数配置远地 IP 地址和远地协议端口：

```
result = connect(socket_handle, remote_socket_address, address_length)
```

remote_socket_address 是一个指向特定 socket 结构的指针，该地址结构为 socket 保存了地址族、协议端口、网络主机地址。

面向连接的服务器则使用 bind 指定本地信息，使用 listen 和 accept 获取远地信息。

使用数据报的客户或者服务器使用 bind 给 socket 指定本地信息，在发送或者接收数据时指定远地信息。

bind 给 socket 指定一个本地 IP 地址和协议端口，如下：

```
result = bind( socket_hndle, local_socket_address, address_length)
```

参数类型同 connect。

函数 listen 监听 bind 指定的端口，如果有远地客户请求连接，使用 accept 接收请求，创建一个新的 socket，并保存信息。

```
socket_new = accept(socket_listen, socket_address, address_length)
```

- 在 socket 配置好之后，使用 socket 发送或者接收数据。

面向连接的 socket 使用 send 发送数据，recv 接收数据；

使用数据报的 socket 使用 sendto 发送数据，recvfrom 接收数据。

14.2 MFC 对 WinSockt API 的封装

MFC 提供了两个类 CAsyncSocket 和 CSocket 来封装 WinSock API，这给程序员提供了一个更简单的网络编程接口。

CAsyncSocket 在较低层次上封装了 WinSock API，缺省情况下，使用该类创建的 socket 是非阻塞的 socket，所有操作都会立即返回，如果没有得到结果，返回 WSAEWOULDBLOCK，表示是一个阻塞操作。

CSocket 建立在 CAsyncSocket 的基础上，是 CAsyncSocket 的派生类。也就是缺省情况下使用该类创建的 socket 是非阻塞的 socket，但是 CSocket 的网络 I/O 是阻塞的，它在完成任务之后才返回。CSocket 的阻塞不是建立在“阻塞”socket 的基础上，而是在“非阻塞”socket 上实现的阻塞操作，在阻塞期间，CSocket 实现了本线程的消息循环，因此，虽然是阻塞操作，但是并不影响消息循环，即用户仍然可以和程序交互。

14.2.1 CAsyncSocket

CAsyncSocket封装了低层的WinSock API，其成员变量m_hSocket保存其对应的socket句柄。使用CAsyncSocket的方法如下：

首先，在堆或者栈中构造一个CAsyncSocket对象，例如：

CAsyncSocket sock；或者

CAsyncSocket *pSock = new CAsyncSocket；

其次，调用Create创建socket，例如：

使用缺省参数创建一个面向连接的socket

sock.Create()

指定参数参数创建一个使用数据报的socket，本地端口为30

pSocket.Create(30, SOCK_DGRM)；

其三，如果是客户程序，使用Connect连接到远地；如果是服务程序，使用Listen监听远地的连接请求。

其四，使用成员函数进行网络I/O。

最后，销毁CAsyncSocket，析构函数调用Close成员函数关闭socket。

下面，分析CAsyncSocket的几个函数，从中可以看到它是如何封装低层的WinSock API，简化有关操作的；还可以看到它是如何实现非阻塞的socket和非阻塞操作。

14.2.2 socket对象的创建和捆绑

（1）Create函数

首先，讨论Create函数，分析socket句柄如何被创建并和CAsyncSocket对象关联。Create的实现如下：

```
BOOL CAsyncSocket::Create(UINT nSocketPort, int nSocketType,
        long lEvent, LPCTSTR lpszSocketAddress)
{
if (Socket(nSocketType, lEvent))
{
        if (Bind(nSocketPort,lpszSocketAddress))
                return TRUE;
        int nResult = GetLastError();
        Close();
        WSASetLastError(nResult);
}
return FALSE;
}
```

其中：

参数1表示本socket的端口，缺省是0，如果要创建数据报的socket，则必须指定一个端口号。

参数2表示本socket的类型，缺省是SOCK_STREAM，表示面向连接类型。

参数3是屏蔽位，表示希望对本socket监测的事件，缺省是FD_READ | FD_WRITE | FD_OOB

| FD_ACCEPT | FD_CONNECT | FD_CLOSE。
参数 4 表示本 socket 的 IP 地址字符串，缺省是 NULL。
Create 调用 Socket 函数创建一个 socket，并把它捆绑在 this 所指对象上，监测指定的网络事件。参数 2 和 3 被传递给 Socket 函数，如果希望创建数据报的 socket，不要使用缺省参数，指定参数 2 是 SOCK_DGRM。
如果上一步骤成功，则调用 bind 给新的 socket 分配端口和 IP 地址。

（2）Socket 函数

接着，分析 Socket 函数，其实现如下：

```
BOOL CAsyncSocket::Socket(int nSocketType, long lEvent,
int nProtocolType, int nAddressFormat)
{
ASSERT(m_hSocket == INVALID_SOCKET);

m_hSocket = socket(nAddressFormat,nSocketType,nProtocolType);
if (m_hSocket != INVALID_SOCKET)
{
    CAsyncSocket::AttachHandle(m_hSocket, this, FALSE);
    return AsyncSelect(lEvent);
}
return FALSE;
}
```

其中：
参数 1 表示 Socket 类型，缺省值是 SOCK_STREAM。
参数 2 表示希望监测的网络事件，缺省值同 Create，指定了全部事件。
参数 3 表示使用的协议，缺省是 0。实际上，SOCK_STREAM 类型的 socket 使用 TCP 协议，SOCK_DGRM 的 socket 则使用 UDP 协议。
参数 4 表示地址族（地址格式），缺省值是 PF_INET（等同于 AF_INET）。对于 TCP/IP 来说，协议族和地址族是同值的。
在 socket 没有被创建之前，成员变量 m_hSocket 是一个无效的 socket 句柄。Socket 函数把协议族、socket 类型、使用的协议等信息传递给 WinSock API 函数 socket，创建一个 socket。如果创建成功，则把它捆绑在 this 所指对象。

（3）捆绑（Attatch）

捆绑过程类似于其他 Windows 对象，将在模块线程状态的 WinSock 映射中添加一对新的映射：this 所指对象和新创建的 socket 对象的映射。
另外，如果本模块线程状态的“socket 窗口”没有创建，则创建一个，该窗口在异步操作时用来接收 WinSock 的通知消息，窗口句柄保存到模块线程状态的 m_hSocketWindow 变量中。函数 AsyncSelect 将指定该窗口为网络事件消息的接收窗口。
函数 AttachHandle 的实现在此不列举了。

（4）指定要监测的网络事件

在捆绑完成之后，调用 AsyncSelect 指定新创建的 socket 将监测的网络事件。AsyncSelect 实现如下：

```
BOOL CAsyncSocket::AsyncSelect(long lEvent)
{
ASSERT(m_hSocket != INVALID_SOCKET);
```

```
_AFX_SOCK_THREAD_STATE* pState = _afxSockThreadState;
ASSERT(pState->m_hSocketWindow != NULL);

return WSAAsyncSelect(m_hSocket, pState->m_hSocketWindow,
WM_SOCKET_NOTIFY, lEvent) != SOCKET_ERROR;
}
```

函数参数 lEvent 表示希望监视的网络事件。

_ afxSockThreadState 得到的是当前的模块线程状态，m_ hSocketWindow 是本模块在当前线程的“socket 窗口”，指定监视 m_hSocket 的网络事件，如指定事件发生，给窗口 m_hSocketWindow 发送 WM_SOCKET_NOTIFY 消息。

被指定的网络事件对应的网络 I/O 将是异步操作，是非阻塞操作。例如：指定 FR_READ 导致 Receive 是一个异步操作，如果不能立即读到数据，则返回一个错误 WSAEWOULDBLOCK。在数据到达之后，WinSock 通知窗口 m_hSocketWindow，导致 OnReceive 被调用。

指定 FR_WRITE 导致 Send 是一个异步操作，即使数据没有送出也返回一个错误 WSAEWOULDBLOCK。在数据可以发送之后，WinSock 通知窗口 m_hSocketWindow，导致 OnSend 被调用。

指定 FR_CONNECT 导致 Connect 是一个异步操作，还没有连接上就返回错误信息 WSAEWOULDBLOCK，在连接完成之后，WinSock 通知窗口 m_hSocketWindow，导致 OnConnect 被调用。

对于其他网络事件，就不一一解释了。

所以，使用 CAsyncSocket 时，如果使用 Create 缺省创建 socket，则所有网络 I/O 都是异步操作，进行有关网络 I/O 时则必须覆盖以下的相关函数：

OnAccept、OnClose、OnConnect、OnOutOfBandData、OnReceive、OnSend。

（5）Bind 函数

经过上述过程，socket 创建完毕，下面，调用 Bind 函数给 m_hSocket 指定本地端口和 IP 地址。Bind 的实现如下：

```
BOOL CAsyncSocket::Bind(UINT nSocketPort, LPCTSTR lpszSocketAddress)
{
USES_CONVERSION;

//使用 WinSock 的地址结构构造地址信息
SOCKADDR_IN sockAddr;
memset(&sockAddr,0,sizeof(sockAddr));

//得到地址参数的值
LPSTR lpszAscii = T2A((LPTSTR)lpszSocketAddress);
//指定是 Internet 地址类型
sockAddr.sin_family = AF_INET;

if (lpszAscii == NULL)
//没有指定地址，则自动得到一个本地 IP 地址
//把 32 比特的数据从主机字节序转换成网络字节序
```

```
        sockAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    else
    {
        //得到地址
        DWORD lResult = inet_addr(lpszAscii);
        if (lResult == INADDR_NONE)
        {
            WSASetLastError(WSAEINVAL);
            return FALSE;
        }
        sockAddr.sin_addr.s_addr = lResult;
    }

    //如果端口为 0，则 WinSock 分配一个端口（1024—5000）
    //把 16 比特的数据从主机字节序转换成网络字节序
    sockAddr.sin_port = htons((u_short)nSocketPort);

    //Bind 调用 WinSock API 函数 bind
    return Bind((SOCKADDR*)&sockAddr, sizeof(sockAddr));
    }
```

其中：函数参数 1 指定了端口；参数 2 指定了一个包含本地地址的字符串，缺省是 NULL。函数 Bind 首先使用结构 SOCKADDR_IN 构造地址信息。该结构的域 sin_family 表示地址格式（TCP/IP 同协议族），赋值为 AF_INET（Internet 地址格式）；域 sin_port 表示端口，如果参数 1 为 0，则 WinSock 分配一个端口给它，范围在 1024 和 5000 之间；域 sin_addr 是表示地址信息，它是一个联合体，其中 s_addr 表示如下形式的字符串，“28.56.22.8”。如果参数没有指定地址，则 WinSock 自动地得到本地 IP 地址（如果有几个网卡，则使用其中一个的地址）。

（6）总结 Create 的过程

首先，调用 socket 函数创建一个 socket；然后把创建的 socket 对象映射到 CAsyncSocket 对象（捆绑在一起），指定本 socket 要通知的网络事件，并创建一个“socket 窗口”来接收网络事件消息，最后，指定 socket 的本地信息。

下一步，是使用成员函数 Connect 连接远地主机，配置 socket 的远地信息。函数 Connect 类似于 Bind，把指定的远地地址转换成 SOCKADDR_IN 对象表示的地址信息（包括网络字节序的转换），然后调用 WinSock 函数 Connect 连接远地主机，配置 socket 的远地端口和远地 IP 地址。

14.2.3 异步网络事件的处理

当网络事件发生时，“socket 窗口”接收 WM_SOCKET_NOTIFY 消息，消息处理函数 OnSocketNotify 被调用。“socket 窗口”的定义和消息处理是 MFC 实现的，这里不作详细的讨论。

OnSocketNotify 回调 CAsyncSocket 的成员函数 DoCallBack，DoCallBack 调用事件处理函数，如 OnRead、OnWrite 等。摘录 DoCallBack 的一段代码如下：

```
    switch (WSAGETSELECTEVENT(lParam))
```

```
{
    case FD_READ:
    {
        DWORD nBytes;
        //得到可以一次读取的字节数
        pSocket->IOCtl(FIONREAD, &nBytes);
        if (nBytes != 0)
            pSocket->OnReceive(nErrorCode);
    }
    break;
    case FD_WRITE:
        pSocket->OnSend(nErrorCode);
        break;
    case FD_OOB:
        pSocket->OnOutOfBandData(nErrorCode);
        break;
    case FD_ACCEPT:
        pSocket->OnAccept(nErrorCode);
        break;
    case FD_CONNECT:
        pSocket->OnConnect(nErrorCode);
        break;
    case FD_CLOSE:
        pSocket->OnClose(nErrorCode);
        break;
```

lParam 是 WM_SOCKET_NOFITY 的消息参数，OnSocketNotify 传递给函数 DoCallBack，表示通知事件。

函数 IOCtl 是 CAsyncSocket 的成员函数，用来对 socket 的 I/O 进行控制。这里的使用表示本次调用 Receive 函数至多可以读 nBytes 个字节。

从上面的讨论可以看出，从创建 socket 到网络 I/O，CAsyncSocket 直接封装了低层的 WinSock API，简化了 WinSock 编程，实现了一个异步操作的界面。如果希望某个操作是阻塞操作，则在调用 Create 时不要指定该操作对应的网络事件。例如，希望 Connect 和 Send 是阻塞操作，在任务完成之后才返回，则可以使用如下的语句：

```
pSocket->Create(0, SOCK_STREAM,
    FR_WRITE|FR_OOB|FR_ACCEPT|FR_CLOSE);
```

这样，在 Connect 和 Send 时，如果是用户界面线程的话，可能阻塞线程消息循环。所以，最好在工作者线程中使用阻塞操作。

14.3 CSocket

如果希望在用户界面线程中使用阻塞 socket，则可以使用 CSocket。它在非阻塞 socket 基础之上实现了阻塞操作，在阻塞期间实现了消息循环。

对于 CSocket，处理网络事件通知的函数 OnAccept、OnClose、OnReceive 仍然可以使用，

OnConnect、OnSend 在 CSocket 中永远不会被调用，另外 OnOutOfBandData 在 CSocket 中不鼓励使用。

CSocket 对象在调用 Connect、Send、Accept、Close、Receive 等成员函数后，这些函数在完成任务之后（连接被建立、数据被发送、连接请求被接收、socket 被关闭、数据被读取）之后才会返回。因此，Connect 和 Send 不会导致 OnConnect 和 OnSend 被调用。如果覆盖虚拟函数 OnReceive、OnAccept、OnClose，不主动调用 Receive、Accept、Close，则在网络事件到达之后导致对应的虚拟函数被调用，虚拟函数的实现应该调用 Receive、Accept、Close 来完成操作。下面，就一个函数 Receive 来考察 CSocket 如何实现阻塞操作和消息循环的。

```
int CSocket::Receive(void* lpBuf, int nBufLen, int nFlags)
{
//m_pbBlocking 是 CSocket 的成员变量，用来标识当前是否正在进行
//阻塞操作。但不能同时进行两个阻塞操作。
if (m_pbBlocking != NULL)
{
    WSASetLastError(WSAEINPROGRESS);
    return    FALSE;
}
//完成数据读取
int nResult;
while ((nResult = CAsyncSocket::Receive(lpBuf, nBufLen, nFlags))
    == SOCKET_ERROR)
{
    if (GetLastError() == WSAEWOULDBLOCK)
    {
        //进入消息循环，等待网络事件 FD_READ
        if (!PumpMessages(FD_READ))
            return SOCKET_ERROR;
        }
        else
        return SOCKET_ERROR;
    }
    return nResult;
}
```

其中：

参数 1 指定一个缓冲区保存读取的数据；参数 2 指定缓冲区的大小；参数 3 取值 MSG_PEEK（数据拷贝到缓冲区，但不从输入队列移走），或者 MSG_OOB（处理带外数据），或者 MSG_PEEK|MSG_OOB。

Receive 函数首先判断当前 CSocket 对象是否正在处理一个阻塞操作，如果是，则返回错误 WSAEINPROGRESS；否则，开始数据读取的处理。

读取数据时，如果基类 CAsyncSocket 的 Receive 读取到了数据，则返回；否则，如果返回一个错误，而且错误号是 WSAEWOULDBLOCK，则表示操作阻塞，于是调用 PumpMessage 进入消息循环等待数据到达（网络事件 FD_READ 发生）。数据到达之后退出消息循环，再次调用 CAsyncSocket 的 Receive 读取数据，直到没有数据可读为止。

PumpMessages 是 CSocket 的成员函数，它完成以下工作：

（1）设置 m_pbBlocking，表示进入阻塞操作。
（2）进行消息循环，如果有以下事件发生则退出消息循环：收到指定定时器的定时事件消息 WM_TIMER，退出循环，返回 TRUE；收到发送给本 socket 的消息 WM_SOCKET_NOTIFY，网络事件 FD_CLOSE 或者等待的网络事件发生，退出循环，返回 TRUE；发送错误或者收到 WM_QUIT 消息，退出循环，返回 FALSE；
（3）在消息循环中，把 WM_SOCKET_DEAD 消息和发送给其他 socket 的通知消息 WM_SOCKET_NOFITY 放进模块线程状态的通知消息列表 m_listSocketNotifications，在阻塞操作完成之后处理；对其他消息，则把它们送给目的窗口的窗口过程处理。

14.4 CSocketFile

MFC 还提供了一个网络编程模式，可以充分利用 CSocket 的特性。该模式的基础是 CSocketFile 类。使用方法如下：
首先，构造一个 CSocket 对象；调用 Create 函数创建一个 socket 对象（SOCK_STREAM 类型）。
接着，如果是客户程序，调用 Connect 连接到远地主机；如果是服务器程序，先调用 Listen 监听 socket 端口，收到连接请求后调用 Accept 接收请求。
然后，创建一个和 CSocket 对象关联的 CSocketFile 对象，创建一个和 CSocketFile 对象关联的 CArchive 对象，指定 CArchive 对象是用于读或者写。如果既要读又要写，则创建两个 CArchive 对象。
创建工作完成之后，使用 CArchive 对象在客户和服务器之间传送数据
使用完毕，销毁 CArchive 对象、CSocketFile 对象、CSocket 对象。
从前面的章节可以知道，CArchive 可以以一个 CFile 对象为基础，通过<<和>>操作符完成对文件的二进制流的操作。所以可以从 CFile 派生一个类，实现 CFile 的操作界面（Read 和 Write）。由于 CSocket 提供了阻塞操作，所以完全可以像读写文件一样读写 socket 数据。
下面，分析 CSocketFile 的设计和实现。

（1）CSocketFile 的构造函数和析构函数的实现

- 构造函数的实现

```
CSocketFile::CSocketFile(CSocket* pSocket, BOOL bArchiveCompatible)
{
m_pSocket = pSocket;
m_bArchiveCompatible = bArchiveCompatible;

#ifdef _DEBUG
ASSERT(m_pSocket != NULL);
ASSERT(m_pSocket->m_hSocket != INVALID_SOCKET);

int nType = 0;
int nTypeLen = sizeof(int);
ASSERT(m_pSocket->GetSockOpt(SO_TYPE,&nType,&nTypeLen));
ASSERT(nType == SOCK_STREAM);
#endif // _DEBUG
}
```

其中：

构造函数的参数 1 指向关联的 CSocket 对象，被保存在成员变量 m_pSocket 中；
参数 2 指定该对象是否和一个 CArchive 对象关联（不关联则独立使用），被保存在成员变量 bArchiveCompatible 中。
Degug 部分用于检测 m_pSocket 是否是 SOCK_STREAM 类型。

- 析构函数的实现

```
CSocketFile::~CSocketFile()
{
}
```

（2）CSocketFile 的读写的实现

分析 CSocketFile 如何用文件的读写实现网络 I/O。

- 文件读的实现

```
UINT CSocketFile::Read(void* lpBuf, UINT nCount)
{
ASSERT(m_pSocket != NULL);

int nRead;

//CSocketFile 对象独立使用
if (!m_bArchiveCompatible)
{
    int nLeft = nCount;
    PBYTE pBuf = (PBYTE)lpBuf;

    //读完 nCount 个字节的数据
    while(nLeft > 0)
    {
        //CSocket 的 Receive，阻塞操作，读取到数据才继续
        nRead = m_pSocket->Receive(pBuf, nLeft);
        if (nRead == SOCKET_ERROR)
        {
            int nError = m_pSocket->GetLastError();
            AfxThrowFileException(CFileException::generic, nError);
            ASSERT(FALSE);
        }
        else if (nRead == 0)
        {
            return nCount - nLeft;
        }

        nLeft -= nRead;
        pBuf += nRead;
    }
    return nCount - nLeft;
}
```

```
//和一个 CArchive 对象关联使用
//读取数据，能读多少是多少
nRead = m_pSocket->Receive(lpBuf, nCount, 0);
if (nRead == SOCKET_ERROR)
{
        int nError = m_pSocket->GetLastError();
        AfxThrowFileException(CFileException::generic, nError);
        ASSERT(FALSE);
}
return nRead;
}
```

- 文件写的实现

```
void CSocketFile::Write(const void* lpBuf, UINT nCount)
{
ASSERT (m_pSocket!=NULL);

//CSocket 的函数 Send，阻塞操作，发送完毕才继续
int nWritten = m_pSocket->Send(lpBuf, nCount);
if (nWritten == SOCKET_ERROR)
{
        int nError = m_pSocket->GetLastError();
        AfxThrowFileException(CFileException::generic, nError);
}
}
```

从 CSockefFile 的读写实现可以看出，CSocketFile 如果独立使用，在 Read 操作时可能出现无限等待，因为数据是分多个消息多次送达的，没有读取到指定长度的数据并不表示数据读取完毕。但是和 CArchive 配合使用，则仅仅读取到数据就返回。至于数据是否读取完毕，可以使用 CArchive 的 IsBufferEmpty 函数来判断。

其他 CFile 界面，CSocketFile 没有实现。

从 CScocketFile 的设计和实现来看，CSocketFile 是使用 CSocket 的一个很好的例子，也是使用 CFile 的一个例子。