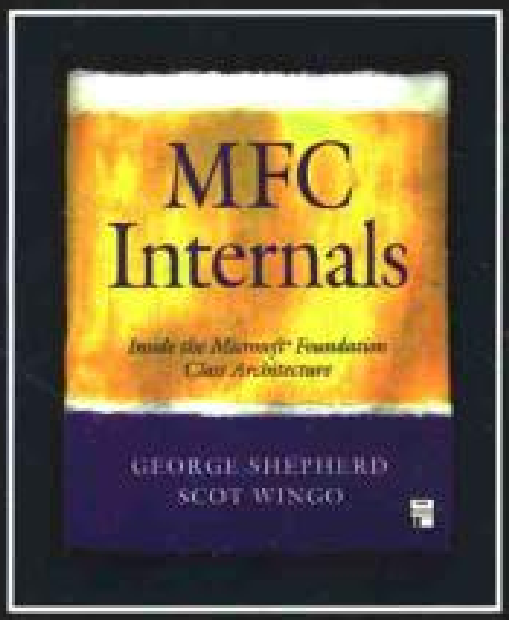
开 发 大 师 系 列

MFC Internals

Inside the Microsoft Foundation Class Architecture

深入解析MFC

[美] George Shepherd, Scot Wingo 著
赵剑云 卿瑾 译



非常简单，本书是所有严谨的MFC开发人员的必备之作。

——Dean McCrory，MFC开发组负责人

中国电力出版社
www.infopower.com.cn

MFC Internals
Inside the Microsoft Foundation Class Architecture

深入解析 MFC

"本书绝对不是对现有文档的重新组织，也不是一本教你如何去做的书，而是一本告诉你MFC是如何运作的书。"

——Dean McCrory，MFC开发组负责人

这是一本填补"使用向导"类的Visual C++书籍、产品文档以及MFC源代码之间空隙的MFC书籍。本书是了解MFC内幕的向导，提供了关于那些没有文档记录的MFC类、实用函数和数据成员的独一无二并且透彻的信息，介绍了有用的编码技巧，并对MFC各个类之间的协作方式进行了重要的分析。

本书的第一部分包含了核心的MFC图形用户界面类以及支持它们的类，第二部分包含了像OLE这种扩展基本Windows支持的主题。如果做到以下几点，你就可以成为一位透彻理解MFC实现细节的专家：

- 探索MFC文档/视图结构的内幕，从而学习视图同步、打印和打印预览
- 更深入地了解MFC序列化中那些没有文档记录的方面和一些没有文档记录的类，例如CPreview、CPreviewDC、CMirrorFile以及CDockBar等等
- 最后理解MFC和OLE是如何共同运作的，以及OLE控件是如何实现的
- 积累技巧，学会自己研究和理解MFC源代码

本书所有示例代码和MFC FAQ文件均在www.infopower.com.cn中提供。附录A是一个便利的MFC源代码指南。

对于理解MFC丰富而强大的应用程序框架以及将MFC的高级知识应用到世界级的Windows应用程序，本书是最基本的向导。

George Shepherd 是DevelopMentor的资深计算机科学家，他为使用MFC和OLE的开发人员开发并发布了很多课件。

Scot Wingo 是Stingray Software公司的创始人之一，该公司主要从事MFC扩展工作。同时，他还维护着MFC FAQ（http://www.unx.com/~scot/mfc_faq.html）站点。

ISBN 7-5083-1800-5

9 787508 318004 >

责任编辑/乔 晶
封面设计/王红柳

Addison Wesley

ISBN 7-5083-1800-5
定价：59.00元

开 发 大 师 系 列

MFC Internals

Inside the Microsoft Foundation Class Architecture

深入解析 MFC

[美] George Shepherd, Scot Wingo 著
赵剑云 卿瑾 译

中国电力出版社

MFC Internals: Inside the Microsoft Foundation Class Architecture
(ISBN 0-201-40721-3)
George Shepherd, Scot Wingo
Authorized translation from the English language edition, entitled MFC Internals ,published by Addison Wesley, Copyright©1996

CHINESE SIMPLIFIED language edition published by China Electric Power Press

本书由美国培生集团授权出版。
北京市版权局著作权合同登记号 图字：01-2002-0713 号

图书在版编目（CIP）数据

深入解析 MFC /（美）谢泼德，（美）温勾著；赵剑云，卿璋译. —北京：中国电力出版社，2003
（开发大师系列）
ISBN 7-5083-1800-5
Ⅰ.深… Ⅱ.①谢…②温…③赵…④卿… Ⅲ.C 语言—程序设计 Ⅳ.TP312
中国版本图书馆 CIP 数据核字（2003）第 069624 号

责任编辑：乔晶
丛 书 名：开发大师系列
书　　名：深入解析MFC
编　　著：(美)George Shepherd, Scot Wingo
翻　　译：赵剑云　卿璋
出版发行：中国电力出版社
地址：北京市三里河路6号　邮政编码：100044
电话：（010）88515918　传真：（010）88518169
本书如有印装质量问题，我社负责退换
印　　刷：北京丰源印刷厂
开　　本：787×1092 1/16　　**印　　张**：33.75　　**字　　数**：765千字
书　　号：ISBN 7-5083-1800-5
版　　次：2003年10月北京第一版
印　　次：2003年10月第 一次印刷
定　　价：59.00 元

前　言

自从 MFC 1.0 于 1990 年发布以来，微软基础类（Microsoft Foundation Class）（现在是 4.0 版了）已经发展成为性能完全、健壮、口碑甚佳、使用广泛而且非常复杂的应用程序框架。对于我们这些一直跟着它发展的人，凭借使用以前版本 MFC 的历史和经验还能帮着减轻一点复杂度。而对于今天刚开始学习 MFC 的新手，学起来就太困难了。进入 MFC 的天堂需要不懈的努力——使用 MFC 写代码，并且大量地阅读。

当然，Visual C++（和其他获得 MFC 许可的 C++产品）都带有介绍 MFC 库的参考文档和概念信息。也有很多第三方写的书讲授如何使用 MFC 创建 Windows 应用程序。然而，所有这些知识源都把焦点集中在如何使用 MFC 上：调用什么、重载什么、怎样激活特定的特性。在几个流行的在线论坛上，已经就是否有必要在文档之上理解 MFC 的实现这个问题展开过几次有趣的讨论。我的观点是，要成为一个真正的 MFC 专家，你必须理解 MFC 的实现。对任何问题的最终回答常常是在源代码本身中找到的。这也是 MFC 提供源码的主要原因之一。

实际上，阅读源代码是我开始学习 MFC 的方法。我这样做部分是因为有必要，部分是出于我的天性。我介入 MFC 是从它的一个内部 Alpha 版开始的，这还是在 Microsoft C/C++ 7.0 开发过程中发布的。当时还没有 MFC 的文档，甚至 CodeView 也还不能处理 C++代码。文档的匮乏没有带给我很大麻烦，因为我很想知道一切到底是怎么工作的。我一直问类似这样的问题："MFC 在处理 Windows 消息时怎么调用我的 C++成员函数？"，"MFC 的异常处理是如何实现的？"（当时 Microsoft C++的编译器还不支持异常处理）。因此，我开始浏览代码，试图理解它。但是，是不是每个使用 MFC 的人都必须这么做？是不是每个人都有这么多时间？可能不是——有的人需要享受生活！

即使你有足够的时间来研究一摞一摞的 MFC 源代码，这本书也可以引导你了解哪里有些什么。Scot 和 George 做了相当出色的工作，分析了 MFC 的源码，在本书中对 MFC 的很多关键的实现细节提供了解释。实际上，很多次我曾经试图写这样的一本书，但是我总是忙于挖掘 MFC 的新版本，没有这么多时间来创作这样一本书。幸运的是，我还有时间看完这本书，并且提供一点我的看法。

本书绝对不是对现有文档的一个重新组织，也不是一本教你如何去做的书，而是一本告诉你 MFC 是如何运作的书。很多章节都以类似这样的话开始："下面的声明已经去掉了所有有文档说明的成员函数，我们将注意力集中在没有文档说明的实现细节上。"这些实现细节会令你明白某个类内在的一些性质和限制，即使这些可能在类的公共接口中不出现。这些细节以及它们如何影响你的应用程序的知识会影响你在自己代码中使用这些类的方式。当你真正

坐下来写一个应用程序时，这些都是非常重要的细节。此外，有些细节可能只是为了趣味或展示一些你可能希望在自己的程序中使用的 C++技术（或者是你要避免的技术）。不过作者并没有揭示所有有趣的东西，有些东西留给读者自己去挖掘。在读了本书中的大量分析之后，你应该已经有能力自己探究这里没讲到的主题。掌握这个技能十分重要——很多在流行的在线服务上问到的问题（甚至在微软内部问到的问题）都可以通过简单地浏览 MFC 代码而得到解答。

简而言之，本书是所有严谨的 MFC 开发人员的必备之作。实际上，微软内部的 MFC 开发、质量保证、用户培训小组的所有新老成员都应该读一读这本书。

Dean D. McCrory

致　谢

Scot Wingo：首先感谢读者，感谢你有兴致读这样一本书。这本书写出来就像一股冲击波。感谢 Claire Horne，因为他签约雇用了一个没有任何写书和 MFC 经验的人（开玩笑的）。感谢 McCrory 院长，因为他敏锐的洞察力，他有趣的 AFX 组笑话，当然还有他提供的原始技术反馈。

尤其要感谢我的狗 Mack，因为每写一页的时候，它都给予我道义上的支持，而且还在空闲时间和我一起玩飞盘。

最后感谢我的伙伴 Stingrays，因为他是一个出色的同伴；感谢我的父母，因为他们生育了我；感谢 1203 Belle Mead 的那些人，因为好吃的红辣椒餐；也感谢可口可乐公司，因为它发明并分发了 Mello Yello——写本书过程中的最佳提神饮料；感谢 Superchunk、Small、StereoLab 和 Juliana Hatfield，因为他们写出了伟大的曲调；感谢 Dad、Todd McFarlane、John Walker、Frank Miller 和 Jim Lee，因为他们是企业家角色的榜样；最后感谢 George，因为他让我参与这个计划，从而登台成为一位大师。

George Shepherd：每当我读一本关于软件和编程的书，就不可避免地看到书中写着类似“感谢我的配偶，因为在我写这本书时她容忍着我的离开”这样的话。我过去认为他们是在开玩笑，或者只是善意地这样做。但是，现在我不这样想了。完成这本书是我所做过的事情中最艰难的一件。确实，我应该感谢我的妻子 Sandy，因为每当我消失在办公室埋头苦干这个项目的时候，她都给予了很大的耐心。另外 Sandy 也是一个很好的评论家，她帮助我澄清那些令人混淆的文字。既然现在我有了时间，从现在开始，我要为自己离开的时间和她给予我的帮助回报她。

感谢我的儿子 Teddy，因为他理解我为什么有时不能陪他去动物园。

感谢 Don Box，因为他以一种易于理解的方式向我解释 OLE；感谢 Mary Kirtland，因为她帮助我使得这个项目可以正常运作；感谢 Andrew Schulman，因为他提供了反馈意见并且推荐了能接收本书的 Addison-Wesley；感谢 Jeff Duntemann，因为他发表了我的前两篇文章，帮助我开始写作；感谢 J.D.Hildebrand 和 Oakley Publishing 的人们，因为他们帮助过我鼓励过我；感谢 Addison-Wesley 的 Claire Horne，因为在我们因仔细推敲敲而影响进度（至少我们没有按她喜欢的速度写出来）的时候她仍然保持了极大的耐心；感谢 Dean McCrory，因为他编辑了本书的技术部分；感谢我的兄弟 Patrick，因为他是一个关于想法的宣传者；感谢我的父母，因为他们激励我去尽最大努力做到最好。最后我还要感谢 Scot Wingo，因为他是一个伟大而又饱含激情的写作伙伴。

简　介

这是又一本关于 MFC（Microsoft Foundation Class，Microsoft 基础类）的书，里面又写了些什么呢？

当你在喜爱的书店浏览 C++和 Windows 方面的书时，你会发现有很多书都是关于 Microsoft Visual C++和 MFC 的。其中大部分书籍都集中介绍了向导代码生成工具，并包含了使用 MFC 的基本方法。这本书是不同的——你不会发现一本和它相似的书。它的不同之处就在于它是关于 MFC 内幕的。它会告诉你 MFC 是如何整合在一起，以及在表象后面发生的事件。

为什么需要本书？

为什么你需要一本关于 MFC 内幕的书？为了回答这个问题，首先让我们来看一个典型的 MFC 编程例子。

使用 AppWizard，点击某些按钮，做出一些选择，在 10 分钟之内，你就可以得到一个 Windows 应用程序。这个应用程序支持 MDI（Multiple Document Interface，多文档界面）和 OLE（Object Linking and Embedding，对象链接和嵌入），还有一个浮动的工具栏、一个状态栏、打印对话框、打印预览对话框和关于对话框。在利用 MFC 编程的早期，事实上你已经成为超级程序员——比一个快速的 C 程序员更快，而且能够在一个小范围内跳过 OLE。

下面你决定修改应用程序内容，从而完成一些类似于处理多文档类型的事情：例如，增加一个 GIF/BMP 图像浏览器。你感觉像突然撞到一堵砖墙。MFC 将文档链表放在哪里了呢？将它们的类型添加到“打开文件”对话框需要做什么呢？你的第一反应可能是检查 MFC 的文档，但你会发现里面没有任何这样的细节内容。幸运的是，微软将 MFC 的源代码封装在类库中，所以你现在要做的就是在类库中查找它并弄清楚，对吗？

错，很快你就会意识到 MFC 的代码十分复杂，仅是一个单独的源文件都有超过 120000 行的代码，还不算头文件和.H 扩展文件。所以，你要再花一周左右的时间增加一个新特性，而在你掌握了 AppWizard 经验后只花几分钟就可以完成。现在你已经失去了那种神速超级程序员的感觉，你需要花上几个白天（或晚上）专心理解 MFC 的源代码并学习有关如何让 MFC 按照你的意愿为你服务的知识。

有些情况下你需要更多地了解 MFC 内幕。这里给出了你在用 MFC 编程时可能已经遇到的几个例子：

- 当你调试应用程序时，会经常停在 MFC 的源代码中，其中都是你所不熟悉的类和没有文档说明的内部结构。知道这些没有文档说明的 MFC 结构正在做什么，并知

道它们如何影响了你所跟踪的问题不是更好吗？

- 在 MFC 中，许多类都是用来继承的。换句话说，你不需要直接创建它们的实例，但需要创建一个派生类，并将这个派生类实例化。在这一过程中，你必须覆盖基类的一些虚函数，这样才能在你的派生类中获得一些需要的行为。问题就在于 MFC 的文档没有清晰地说明何时你应该完全替代函数、何时你应该第一次调用这个函数以及何时增加自己的代码。例如，当创建 CDialog 的一个派生类时，需要重载 OnInitDialog()来完成一些初始化工作，那么你是应该先调用 CDialog::OnInitDialog()呢？还是完全替代这个函数而不去理会那些冲突呢？显然，要做这个决定，你必须理解 CDialog::OnInitDialog()都做了些什么——只有清楚了这些才能使得是否调用基类的函数变得一目了然。
- MFC 对 OLE 的支持非常完备，但如果你还想要扩展它，或者写自己的 COM 对象，那么就要放弃已有记录的 MFC OLE 类的安全性，还必须在自己的类中指出 MFC OLE 代码。

本书要解决的问题

尽管 MFC 隐藏了细节，并很好地组织了 Windows 代码的功能，但是它没有削减了解其内部的需要。如果你想要创建热门的商用应用程序，程序不只是弹出一些对话框那么简单，你就必须知道 MFC 是如何工作的。有许多工作组致力于研究 MFC 内幕。其中许多还不十分明显，有一些甚至没有记录。这些也就是我们这本书所要探讨的。

我们已经搜索了 MFC 的源代码，以求找到能帮助解决 MFC 问题的有用信息。我们已经发现了很多有趣的内容，例如没有文档说明的类、功能函数和数据成员，有趣的 C++技术，有用的代码技术和若干关于各种 MFC 类如何工作以及它们如何互相合作的细节。我们将总结所有的关键事实和至关重要的发现，以便于你能快速地解决在 MFC 编程中所遇到的问题（再次成为超级程序员！）。

本书读者对象

本书是为那些决定使用 MFC 2.0 或更高版本的、认真的 Windows 开发人员而编写的（我不十分关心你是否认真，只要你想学）。

我们对读者假设以下两件事情：

1. 理解 C++。MFC 是 C++的类库，所以你需要掌握 C++工作的方式。你必须能够读懂这些代码，遵从语法，并理解底层的面向对象编程的封装、继承和多态性等基本思想。MFC 在它的实现中十分依赖这些概念。按照这种方法，如果我们找到任何 MFC 使用的十分高级或有趣的 C++技术，我们就将它指出，以便你能使用这种技术。

2. 从开发的角度看，你必须掌握基本的 Windows。你需要理解一些概念，例如 window 句柄、消息、设备环境以及矩形等等。

如何使用本书

本书关注服务于 Windows 95 和 Windows NT 的 32 位 MFC 4.0。如果你所使用的 MFC 的版本较低，请不必担心，因为经过这么多年，大多数的内部概念都没有发生变化。如果有的发生了变化，我们会附带一个关于版本的提示。

贯穿本书，我们介绍了那些在标准文档中漏掉的重要概念。我们还指出了那些没有文档说明的类、特性和成员函数等等，我们还会告诉你如何在你的应用程序中使用它们 www.infopower.com.cn 中有一些关于如何使用 MFC 的示例。

本书的第一部分（第 1 章~ 第 8 章）集中讨论了核心的 Windows 图形用户界面类和支持类。在第二部分（第 9 章~ 第 14 章），我们介绍了对基本 Windows 支持的扩展内容，如 OLE。最后，在第 15 章，我们给出了 MFC 的高级内幕。

我们设计这本书的目标有很大弹性。你必须一页一页地读这本书，或者将它作为一本参考书。无论哪种情况，如果你从来没有看过 MFC 的源代码，那么附录 A 可能是一个好的起点。它回顾了 MFC 源代码的组织，还包含了一些微软的编码方针。

如果你是一个 MFC 初级程序员，你可能想从第 1 章开始，然后在读其他部分之前先读第 2 章和第 5 章。第 2 章阐述 MFC 如何封装 Windows，第 5 章讨论大多数 MFC 类的基类 CObject 如何工作，以及它在原有的 Windows 基础上增加了哪些内容。

如果你是一个 MFC 中级程序员，你可以简单地复习前 5 章，然后直接进入你最感兴趣的题目。例如，如果你需要一些关于如何实现 MFC 文档/视图结构的信息，就可以直接跳进第 7 章。

最后，如果你是一个高级 MFC 程序员，你可能想要阅读那些包含你尚未了解的内幕的章节。

警　告

微软没有将这里所指出的内容归档也是有原因的，他们希望这些内容能够发生变化。事实上，在写书的过程中，我们必须修改很多章节，因为 MFC 变化得太快（一年发布 4 次）。如果你决定在自己的应用程序中使用我们给出的任何内幕，请注意要使用注释，或者使用#ifdef，这样在新版本的 MFC 破坏你的用法时，你能够知道究竟发生了什么变化。例如，在 MFC 以前的版本中，文档和视图的链表保存在文档模板中。在 4.0 版本中，微软将这些链表移动到新类 CDocManager 中。如果你已经使用了以前链表的位置，你就不得不重写这段代码以使用新的类。

联系作者

我们已经尽了最大努力提供那些适时的实际信息。如果你发现书中有问题，或者你有什么意见以及想要讨论某个话题，你可以访问我们的网页 http://www.stingsoft.com/mfc_internals 并反馈给我们。这个站点还拥有对本书的修改和其他一些有趣的 MFC 站点的链接。如果你不能通过网络访问，还可以通过 e-mail：mfc_internals@stingsoft.com 联系我们。我们会尽快地回复你。

现在，冒险开始了。

目　录

CHAPTER

1 MFC的概念性总括

你如果理解了MFC设计者的意图，就会更清楚地看到MFC的整体框架结构。掌握了它的框架结构，你就能更好地利用MFC的性能。这一章从提供一个MFC的高层的概念性总括开始，介绍了MFC的一些历史。我们首先回顾一下OOP（Object-Oriented Programming，面向对象编程）的基本原则和MFC的设计目标（如果你很熟悉OOP的原则，就可以直接跳到“应用程序框架与MFC”一节）。然后我们会展现一副更大的图像：MFC的组件以及这些基本部分是如何共同工作的。

面向对象编程的一些背景

目前，似乎每个软件包的广告都宣称它们的程序中使用了对象。当然，其中一些确实是面向对象的，而另一些却不是。但说一个程序是面向对象的意味着什么呢？MFC是面向对象的。为了更彻底地理解MFC，你需要理解OOP的基本原理，包括抽象、封装、继承、多态性和模块化等概念。本节将介绍OOP的总括性的背景和一些面向对象的概念。

结构化设计、分析和编程在20世纪70年代中期到80年代后期很流行。它是每个人从计算机课程中所学到的。它也帮助实现了无数的系统，其中有许多至今还在使用。但随着软件开发者所处理的工程越来越大，结构化技术逐渐显露出它的缺点。这并不是说它们已经没有用了，只是目前更大、更复杂的系统需要新的技术和概念来辅助控制系统的复杂性。OOP可以对现实世界中不可能使用结构化技术解决的问题建模。因而，许多开发人员开始使用面向对象分析、设计和编程来帮助他们管理这样庞大而复杂的系统。

面向对象编程术语

通常用来描述一个面向对象系统的术语是抽象、封装、继承、多态性和模块化。一种编程语言只有支持了这些特性才能称之为面向对象编程语言。下面给出了每个概念的细节描述。

抽象（abstraction）

抽象定义了一个对象区别于其他对象的重要特性，它与用户的观点有关。抽象从外部观察对象，忽略了那些关于对象是如何实现的细节。抽象还为对象类之间的交互定义了约定和

协议。选择适当的抽象或类是面向对象设计的主要目标。

封装（encapsulation）

封装就是将代码和数据组合成一个整体的能力。抽象和封装二者紧密结合，没有封装，就不能定义任何用于对现实世界对象建模的抽象。在C++中，封装的单元是类（class）和结构（struct）。理想情况下，应该能在没有看到对象的实现前使用对象。了解对象的实现会增加复杂度，因为你必须把它看作是类的一个客户。

然而，这个世界不是理想的。正如Grady Booch（《Object-Oriented Analysis and Design》的作者）所说：在实际情况下，一个人要研究一个类的实现从而理解它的意义需要很长时间，尤其是在缺少额外的文档时（这也就是为什么有了这本关于MFC的书的原因）。

继承（inheritance）

继承就是从已存在类的基础上获得新的类的能力。继承使得你能够建立类层次结构，从而使你能够重用已有的代码。在C++中，继承一个类很简单，新的类具有基类的所有功能。一旦你有了所有的好功能在手，你就可以按照自己的需要来修改。

多态性（polymorphism）

多态性的本意就是“有很多形状”。这意味着在OOP中，在类层次结构中，一个类可以向上或向下共享指定的函数名，虽然每个特定的类所表现出的行为是不同的。例如，假设使用OOP来开发一些画各种形状图形的类。每种形状的图形有自己的绘画方法。即使每个对象的绘画方法看起来很相似，你还是希望每个图形有自己的绘画行为。你需要一个圆对象来画圆，需要一个正方形对象来画正方形。每个对象以自己的绘画方法完成绘画的能力就是多态性的例子。在C++中，多态性是通过虚拟成员函数来完成的。虚拟成员函数是在运行时而不是在编译时与类绑定在一起的（像一般的C函数一样）。多态性之所以存在于C++中，是因为究竟选择哪个类的成员函数来调用是在运行时决定的。

模块化（modularity）

虽然类形成了一个系统的组件，但它们每个都不能组成一个完整的系统。为了定义系统的体系结构，需要将这些类分成模块。模块对于控制复杂度很重要（尤其是对大系统），所以确定如何将你的类分成模块就和选择类一样难。模块化实质上是一种封装，它在你将相关的类组合在一起以提供更高层次的行为时发生。一个好的模块功能结合紧密，并提供了满足客户需求的最小接口。

通常的对象

对象就是一个有属性并展现一定行为的实体。以你每天使用的计算机为例，它就是一个对象：它有属性并且展现一定的行为。即使计算机内部的工作情况很复杂，也不防碍你使用它。在你和计算机之间有一层抽象。计算机的实现细节被安全地封装在组成机器的金属、塑

CHAPTER

1 MFC的概念性总括

你如果理解了MFC设计者的意图，就会更清楚地看到MFC的整体框架结构。掌握了它的框架结构，你就能更好地利用MFC的性能。这一章从提供一个MFC的高层的概念性总括开始，介绍了MFC的一些历史。我们首先回顾一下OOP（Object-Oriented Programming，面向对象编程）的基本原则和MFC的设计目标（如果你很熟悉OOP的原则，就可以直接跳到“应用程序框架与MFC”一节）。然后我们会展现一副更大的图像：MFC的组件以及这些基本部分是如何共同工作的。

面向对象编程的一些背景

目前，似乎每个软件包的广告都宣称它们的程序中使用了对象。当然，其中一些确实是面向对象的，而另一些却不是。但说一个程序是面向对象的意味着什么呢？MFC是面向对象的。为了更彻底地理解MFC，你需要理解OOP的基本原理，包括抽象、封装、继承、多态性和模块化等概念。本节将介绍OOP的总括性的背景和一些面向对象的概念。

结构化设计、分析和编程在20世纪70年代中期到80年代后期很流行。它是每个人从计算机课程中所学到的。它也帮助实现了无数的系统，其中有许多至今还在使用。但随着软件开发者所处理的工程越来越大，结构化技术逐渐显露出它的缺点。这并不是说它们已经没有用了，只是目前更大、更复杂的系统需要新的技术和概念来辅助控制系统的复杂性。OOP可以对现实世界中不可能使用结构化技术解决的问题建模。因而，许多开发人员开始使用面向对象分析、设计和编程来帮助他们管理这样庞大而复杂的系统。

面向对象编程术语

通常用来描述一个面向对象系统的术语是抽象、封装、继承、多态性和模块化。一种编程语言只有支持了这些特性才能称之为面向对象编程语言。下面给出了每个概念的细节描述。

抽象（abstraction）

抽象定义了一个对象区别于其他对象的重要特性，它与用户的观点有关。抽象从外部观察对象，忽略了那些关于对象是如何实现的细节。抽象还为对象类之间的交互定义了约定和

CHAPTER

1 MFC的概念性总括

你如果理解了MFC设计者的意图，就会更清楚地看到MFC的整体框架结构。掌握了它的框架结构，你就能更好地利用MFC的性能。这一章从提供一个MFC的高层的概念性总括开始，介绍了MFC的一些历史。我们首先回顾一下OOP（Object-Oriented Programming，面向对象编程）的基本原则和MFC的设计目标（如果你很熟悉OOP的原则，就可以直接跳到“应用程序框架与MFC”一节）。然后我们会展现一副更大的图像：MFC的组件以及这些基本部分是如何共同工作的。

面向对象编程的一些背景

目前，似乎每个软件包的广告都宣称它们的程序中使用了对象。当然，其中一些确实是面向对象的，而另一些却不是。但说一个程序是面向对象的意味着什么呢？MFC是面向对象的。为了更彻底地理解MFC，你需要理解OOP的基本原理，包括抽象、封装、继承、多态性和模块化等概念。本节将介绍OOP的总括性的背景和一些面向对象的概念。

结构化设计、分析和编程在20世纪70年代中期到80年代后期很流行。它是每个人从计算机课程中所学到的。它也帮助实现了无数的系统，其中有许多至今还在使用。但随着软件开发者所处理的工程越来越大，结构化技术逐渐显露出它的缺点。这并不是说它们已经没有用了，只是目前更大、更复杂的系统需要新的技术和概念来辅助控制系统的复杂性。OOP可以对现实世界中不可能使用结构化技术解决的问题建模。因而，许多开发人员开始使用面向对象分析、设计和编程来帮助他们管理这样庞大而复杂的系统。

面向对象编程术语

通常用来描述一个面向对象系统的术语是抽象、封装、继承、多态性和模块化。一种编程语言只有支持了这些特性才能称之为面向对象编程语言。下面给出了每个概念的细节描述。

抽象（abstraction）

抽象定义了一个对象区别于其他对象的重要特性，它与用户的观点有关。抽象从外部观察对象，忽略了那些关于对象是如何实现的细节。抽象还为对象类之间的交互定义了约定和

受外部资源随意影响的数据组成的。它是由许多自治的实体组成，它们之间互相通信，例如一个公司由很多一起工作的雇员组成。这也就是对象在程序内部所做的。每个对象都知道如何很好地完成自己的工作，而且还同其他的对象（这些对象也知道如何很好地完成自己的工作）协作来达到一个共同的目标。

使用 C++作为面向对象语言（C++不仅仅是比 C 语言更好的 C 语言）并使用面向对象技术，不仅增加了代码的重用性，还增加了设计的重用性，从而使开发过程加快，也使长期维护的开销减少。总地来说，用对象模型搭建的系统易于修改，这是因为系统中的对象模拟了现实世界中问题空间里的对象。最终，使用对象建模激发了软件开发过程的演化，它将独立的组件整合，逐渐搭建成复杂的系统，从而减少了开发风险。OOP 的一个特殊的用处就是创建了可重用的应用程序框架。

应用程序框架与 MFC

MFC 是一个应用程序框架，它是专门为微软的 Windows 操作系统创建应用程序而设计的。而我们也确实需要这样一个应用程序框架。

Microsoft SDK（Software Development Kit，软件开发工具包）文档按字母顺序记录了所有的 Windows API（Application Programming Interface，应用程序编程接口）。微软对 API 的记载非常详尽，寻找有关信息十分简单（首先你要提供所要查找的关键字），但是你不能从这些文档中区分哪些 API 更重要，也不知道这些 API 之间是如何合作的。直至最近，还没有这些函数的“see also”附录，各个函数之间的关系也很难理清。MFC 只是使用抽象、封装、继承、多态性和模块化的面向对象原则，在逻辑上将 Windows API 分类。

不太久远的历史

在 1989 年，微软成立了应用程序框架技术开发组（application framework technology development group），又叫做 AFX 组。这个小组为 Windows 应用程序开发人员创建了 C++和面向对象工具（据说 AF 听起来不够强大，所以他们在后面增加了字母 X，X 并没有什么含义）。小组成员都是精选出来的经验丰富的微软开发人员，他们各自有不同的背景。有一些来自设计并实现微软应用程序共享组件的小组，另外一些来自原来的 Windows 和 Presentation Manager 组，还有一些来自应用程序划分组。AFX 组的宗旨是：“使用最新的面向对象技术来为市场上编写最先进的 GUI 应用程序的程序员提供工具和库。”

不幸的是，第一个应用程序框架原形被证明与 Windows 无关。经过一年的工作，AFX 组又创建出一个新的应用程序框架，这个框架完全包含了 Windows 系统、图形子系统、用户控件体系结构、对象数据库、一个泛化的容器层次结构和一个垃圾回收策略！但是当 AFX 小组试图使用这个框架来写应用程序时，他们发现这个框架太复杂，与 Windows 本身有太大区别。所以，正如 Sinofsky 所报告的，他们放弃了这个原型，并将他们的宗旨修改为：“AFX 将向程序员传递面向对象解决方案的力量，使他们能用 C++搭建出世界级的基于 Windows 的应用程序。”（大概与此同时，Windows 3.0 发布了，Windows 3.0 是第一个 Windows 的可工作版本。）新的焦点就在于用 C++为 Windows 开发人员创建一个简单的、一流的、在技术上

因为已有的这些 Windows 程序的代码在你使用 MFC 时仍然是有用的。

一个牢固的基础

Windows API 就像滚雪球一样，越来越大，因为它还在不停地搜集新的资料。微软不断地向 Windows 中增加新的 API，包括 OLE 2.0 和 ODBC。Windows NT 的出现又带来了一个新的有更多特性的 32 位 API。尽管初始框架不支持每个新的 Windows API，但 MFC 的设计能保证它的可扩展性。每个新的 API 都向开发人员引入了新的内容，但是 MFC 将 API 的细节封装到一系列类中，从而减轻了开发人员的负担。例如，为了在一个应用程序中能使用 OLE，往往需要写上千行的代码，但是如果使用 MFC 中的 OLE 类，就可以大大减少代码的行数。

保持小而快的框架

AFX 组希望框架保持小而快。他们设计 MFC 的最初目标就是使用 MFC 框架只需 20K 的内存代价，而且所开发的应用程序不比同等的 C/SDK 应用程序慢。这一限制对 MFC 的整体设计和实现都有很大影响。抽象层次比较高的类库和许多虚拟函数都将产生大而慢的应用程序。为了保证速度更快，规模更小，AFX 发明了其他机制来处理 Windows 消息。MFC 在没有大量虚拟函数的情况下保持了灵活性。

第一个发布版本

1992 年，微软发布了 MFC 的 1.0 版本，同时带有他们的 C/C++ 7.0 版本和与 Windows NT 一起出售的 SDK。MFC 1.0 中包含了 60 多个 C++类，有些类用于 Windows 应用程序开发，还有一些时间类、字符串类、集合类（vector、list、map）、文件类、永久存储类、内存管理类、异常类和诊断类等通用类。这些类都是经过优化处理的，以便在需要很小内存的情况下仍能快速执行。MFC 不是 Windows 开发的一个全新的高层次的抽象，它提供了一个 Windows API 的小而有效的 C++转换。

不足之处

MFC 1.0 受到了表扬，而且被一些人认为是 Windows 的标准应用程序框架。然而，没有事情是完美的，AFX 小组也收到了很多关于 MFC 1.0 的建设性反馈，这些反馈来自那些希望改进 MFC 的现实世界的开发人员。从他们所收到的数以百计的意见和要求特性中，小组决定应该为下一个发布版本 MFC 2.0 增加两个主要特性：（1）高层体系结构的支持；（2）封装组件。

AFX 小组增加高层体系结构的支持和一系列封装组件来简化 Windows 开发。对于那些遵循微软的应用程序接口设计指南中提供的用户接口原则的 Windows 应用程序、那些与 big-name 应用程序[例如工具栏（toolbar）、状态栏（status bar）、打印预览（print preview）和与上下文相关的帮助（context-sensitive help）]具有相同通用特性的应用程序，MFC 增加的这些内容使开发人员搭建以上两种 Windows 应用程序变得更容易。MFC 的体系结构还为开发人员定制和扩展一般应用程序提供了 hook（钩子）。然而，MFC 2.0 不要求开发人员必须使用这些高层次的抽象。开发人员仍然可以在低层次访问框架，就像 MFC 1.0，或者直接访问 Windows API。

在设计高层体系结构时，AFX 组紧紧记住 Windows 自己将来的方向。Windows 应该在 OLE 2.0 的基础上继续向面向对象操作系统方向发展。尽管 OLE 2.0 在 MFC 2.0 发布时还没有发布，但是体系结构的设计已经使对 OLE 2.0 的支持成为已有体系结构的自然扩展。

AFX 组还考虑了一些额外的限制：兼容性和可移植性。使用 MFC 1.0 写的应用程序与使用 MFC 2.0 写的应用程序是兼容的。开发人员不希望为了使用 MFC 2.0 的一些新特性而重写 MFC 1.0 应用程序。此外，用 MFC 写的应用程序在不同的应用程序目标平台之间要可移植。将使用 Windows 3.1 的 MFC 2.0 写的应用程序移植到 Windows NT 或 Win32 上很简单，就像使用 32 位的 MFC 2.0 重新编译一遍一样。

1993 年 4 月，微软发布了 Windows 的 Visual C++ 1.0，它所带的 MFC 的版本是 2.0。随着 Windows NT 的发布，NT 的 Visual C++ 1.0 带的 MFC 是 2.1 版本。这两个版本有相同的特性集合，虽然它们的底层实现稍有不同。MFC 2.0 包括大多数的基本 Windows API 和 OLE 1.0。它包含了 100 多个类（几乎 60000 行的标准 C++代码）。虽然 MFC 1.0 中的通用 Windows 类得到了增强，但这一发布版本的主要特性是新的应用程序体系结构和高层次抽象。应用程序体系结构类提供了一般 Windows 特性的标准实现，例如文档与视图、打印和命令处理。高层次的抽象为通用用户界面元素（例如工具栏和状态栏）提供了一系列预创建好的组件。

目前进展

在 1993 年底，微软发布了 Visual C++ 1.5，其中带有 MFC 2.5。MFC 2.5 搭建在 MFC 2.0 中引入的高层系统结构之上，又增加了对 OLE 2.0 和 ODBC 的支持。1994 年底，微软又发布了 MFC 3.0，这一次的主要改进是增加了线程的安全性。1995 年初，微软在 MFC 3.1 中增加了对简单的 MAPI（Messaging Application Programming Interface，消息接发应用程序编程接口）和 WinSock 的支持。接着，在 1995 年底，微软又同时发布了 VC++ 4.0（开发环境）和 MFC 4.0（框架），这样就大大改善了开发环境，将问题集中在可重用性上。

目前，微软已经为使用 MFC 增加了另一个理由：他们希望看到 MFC 成为 Windows 编程的 C++接口。在将来，因为微软的支持，你会看到更多的 MFC/C++代码。所以，要尽可能多地学习 MFC。本书就是一个重要的起点。

MFC 要点之旅

旅程的第一站就是看看 MFC 是由哪些类组成的，以及它们之间是如何配合工作的。MFC 很大，而它又必须那么大，因为它封装了大多数的 Windows API 并提供了一个健壮的应用程序框架。然而它可以被分成几个可管理的块。首先，微软将这些类分成 4 组：

- 通用（general-purpose）类。
- Windows API 类。
- 应用程序框架类。
- 高层抽象。

通用类提供那些类似字符串处理的类、集合类和异常类。Windows API 类则封装了所有的 Windows API。在这里你会发现窗口类、对话框类、设备环境类等等。应用程序框架类处

理整个应用程序中大的部分。它封装了消息泵（message pump）逻辑、打印、在线帮助和 MFC 的文档/视图结构。高层抽象包括那些非基础的特性，例如工具栏、分割的窗口、状态栏。最后 MFC 为几个操作系统扩展提供了支持，例如 OLE、ODBC、简单 MAPI 和 WinSock。本章的剩余部分就是仔细地研究每个组。

通用类

当考虑为 Windows 写一个应用程序框架时，你可能首先想到的就是窗口类、应用程序类和控件类。但是 MFC 包含了很多与可视化元素或应用程序框架的明显结构元素无关的通用类。这些通用类对 MFC 依然很重要，没有它们，MFC 就不能像现在这样很好地运行（在某些情况下，不是所有情况）。本质上，通用类帮助框架粘合在一起。通用类包括 CObject、异常处理类、集合类、动态字符串、文件类、日期和时间类以及几个混合类。

CObject：（几乎）所有类的父类

许多类库都提供一个或两个抽象类，类库中其他类都从这一个或两个类中派生出来。这也是 MFC 的策略。MFC 的根类就是 CObject。从 CObject 派生出来的类继承了一些有用的特性，包括派生类的运行时类型标识、序列化（即持久性）、诊断函数以及对动态对象创建的支持。这些特性是可选的，而且在定义新的派生类时可以启用也可以禁用。

为了达到这些目的，CObject 使用了几个其他 MFC 类的服务。

- CArchive：与 CObject 联合处理对象持久性。
- CDumpContext：诊断函数调用它来输出关于对象的信息。
- CRuntimeClass：与每个从 CObject 派生出来的类相关，并且包含关于类的信息。

异常处理类

异常就是那些在程序可控制范围外发生的不正常情况，包括低内存或 I/O 错误。MFC 提供了一系列异常处理类来解决异常。许多 MFC 函数都能抛出异常，你可以在自己的程序外面调用那些能通过异常处理函数抛出异常的 MFC 函数，这样就可以保护程序。

MFC 预定义的异常处理类包括：

- CException——异常的基类。
- CArchiveException——处理存档时发生的异常。
- CFileException——处理文件操作时发生的异常。
- CMemoryException——处理内存异常。
- CNotSupportedException——处理程序在试图使用一个不支持的特性时而抛出的异常。
- COleException——处理 OLE 客户和服务器的异常。
- CDBException——处理数据访问过程中的异常。
- CResourceException——处理读取 Windows 资源失败时抛出的异常。
- CUserException——处理应用程序特定的异常。

内存诊断类

CMemoryState 结构提供了一系列方便的方法，通过在程序中设置内存断点来跟踪程序中的内存泄漏，它还提出了一些控制和查询内存断点的成员函数。

集合类

一个集合类管理一组对象。所有的 MFC 集合类都包含这样的方法：在集合中增加和删除元素，从集合中获得某些元素，遍历整个集合。所有动态改变数组大小和维护链表（linked list）的代码都可以由 MFC 的集合类来处理。集合类包括以下几种：

作为数组实现的集合

- CByteArray——字节数组。
- CWordArray——字数组。
- CDWordArray——双字数组。
- CPtrArray——指针数组。
- CObArray——从 CObject 派生的对象数组。
- CStringArray——CString 对象数组。
- CUintArray——无符号整数数组。

作为链表实现的集合

- CObList——从 CObject 派生的对象链表。
- CPtrList——指针链表。
- CStringList——CString 对象链表。

MFC 还支持一种叫做映射表（map）的集合类。映射表可以带两个对象，而且二者是成对的。只要你知道了一个对象，查找另一个对象就很容易。例如，如果你想创建一个数字字典和它们的拼写，你可以使用“字到字符串”（word-to-string）映射表将一个数字和它的拼写配成一对。MFC 支持以下的映射表：

- CMapPtrToWord——将一个指针映射为一个字。
- CMapPtrToPtr——在两个指针间映射。
- CMapStringToOb——将一个字符串映射为一个从 CObject 派生的类。
- CMapStringToPtr——将一个字符串映射为一个指针。
- CMapStringToString——在两个字符串间映射。
- CMapWordToOb——将一个字映射为从 CObject 派生的类。
- CMapWordToPtr——将一个字映射为一个指针。

动态字符串

如果你对 C 的不可思议的字符串函数感到灰心，你可以使用 MFC 的动态字符串。使用 CString 类来创建字符串很简单。MFC 的 CString 类提供了一些基本操作，例如连接、比较和赋值。

文件类

MFC 的文件类封装了文件 I/O。CFile 处理标准的二进制文件 I/O，CStdioFile 处理缓冲的 I/O 文件（例如文本文件）。最后 CMemFile 处理内存文件。这些都是二进制文件，它们的行为使它们看似在磁盘上，而实际它们在内存中。

时间和时间跨度类

CTime 和 CTimeSpan 两个类使你在程序中使用时间值变得简单。CTime 类代表了绝对时

间，它提供了成员函数来获取当前时间，并将特定格式的时间转化成字符串，还能提取特定元素（例如分钟、秒、月和年）。CTime 还提供了加、减、比较、赋值等几种操作。

CTimeSpan 以秒为单位表示时间，它主要用来计时。它提供几种方法从其中提取元素（例如天、小时、秒、总天数、总时数和总秒数），还提供了加、减、比较和赋值几种操作。

几个混合类

它们所封装的通用类是一些作为对象实现的 Windows 结构：

- CPoint——封装了 POINT 结构。
- CSize——封装了 SIZE 结构。
- CRect——封装了 RECT 结构。

Windows API 类

MFC 所做的最重要的事情之一就是封装了 Windows API，隐藏了那些你不想知道的难懂的代码。Windows API 类还封装了某些 Windows 对象，例如设备环境。这不是一件容易的事，而 MFC 却完成了它。这里给出了 MFC 中 Windows API 类的纲要。一些类深深地交织在高层体系结构里，但其他的类可以独立使用（例如，你可以在一般的 C/SDK 程序中使用控件类）。

你的应用程序：CCmdTarget、CCmdUI、CWinThread 和 CWinApp 类

在标准的 C/SDK 应用程序中你需要写 WinMain()函数。这个函数会从消息队列中取出消息，然后将消息传递给窗口过程（也叫消息处理函数）。你所写的消息处理函数通常是一个 case 语句，可以利用这个语句截取自己需要的消息并处理它，然后跳过其他消息。

MFC 对此的解决方案是消息映射，由 CCmdTarget 和 CWnd 类来完成。消息映射是一种机制，它将事件映射为类中处理该事件的方法。任何从 CCmdTarget 派生的类都有一个与之相关的消息映射表，它将命令传递给从 CCmdTarget 派生的类。派生类使用各种成员函数来处理消息。

CCmdUI 类提供了更新用户界面对象（例如菜单或复选框控件）状态的函数。例如，CCmdUI 类会让你使用消息来产生命令目标对象，从而自动更新菜单状态，而不是调用 EnableMenuItem() 更新菜单状态。在点击菜单之后以及菜单项显示之前，MFC 会给应用程序中的命令目标发送一个命令更新消息。如果在命令目标对象的消息映射表中有这个更新消息的内容，MFC 会给 CCmdUI 对象传递一个代表菜单项的指针，也就是命令目标对象所更新的内容。

CWinThread 代表在 MFC 程序内执行的线程。虽然 MFC 早期的版本不是线程安全的，但 3.0 和之后的版本都是线程安全的。这是通过支持两种不同的线程来完成的：辅助线程（worker thread）（真实的、抢占式的 Win32 线程）和用户界面线程［只是一个使用了 GetMessage()..DispatchMessage()的标准消息泵（message pump）］。

在标准的 C/SDK 中，应用程序的核心是消息循环，如下所示：

```
while (GetMessage((LPMSG)&msg, NULL, 0, 0)) {
   /* Check for menu accelerator message */
   if (!TranslateAccelerator(hwndMain, hAccel, &msg)) {
      TranslateMessage((LPMSG)&msg);
      DispatchMessage((LPMSG)&msg);
   }
}
```

CWinThread 封装了这一行为。这也就是为什么在你的 MFC 代码中找不到 WinMain()函数的原因。组成标准 WinMain()函数的功能被隐藏在 CWinThread 类和组成 MFC 的库中。

CWinApp 是从 CWinThread 派生的类，它代表标准的 Windows 应用程序。CWinApp 有一些可覆盖的函数，你可以用它们来初始化你的应用程序，或者根据需要在结束时做一些收尾工作。因为 CWinApp 是从 CWinThread 派生出来的，所以它也有一个 Run()方法，这个方法负责执行应用程序。

同步对象类

MFC 4.0 增加了一套用于简化编写多线程应用程序的类，这通过在不同的 Win32 同步方法周围提供各种包装层来实现。这些类包括：

- CSyncObject——同步对象类的基类。
- CCriticalSection——一个同步类，它只允许单个进程中的一个线程访问一个对象。
- CSemaphore——一个同步类，它允许对一个对象有 1 个到某个指定数目的之间个数的同步访问。
- CMutex——一个同步类，它只允许任何数目进程中的一个线程访问对象。
- CEvent——一个同步类，当某个事件发生时，它会通知某个应用程序。
- CSingleLock——线程安全的类的成员函数中用来锁住一个同步对象的对象。
- CMultiLock——线程安全的类的成员函数中用来锁住一个或更多个同步对象的对象，锁住的对象来自一个同步对象数组。

相关的类

另外两个在应用程序中广泛使用的类是：

- CCommandLineInfo——分析程序启动时的命令行。
- CWaitCursor——在屏幕上显示一个等待光标，在时间较长的操作过程中使用。

CWnd：所有窗口类的父类

所有其他窗口类（例如对话框、控件和框架窗口）都是 CWnd 的派生类。因为 CWnd 是从 CCmdTarget 派生的类，所以它可以接收和处理消息。在 CWnd 内部，你会发现特定窗口的句柄、大多数以窗口为操作对象的公用 API 函数（例如 ShowWindows()，MessageBox()和 MoveWindow()），以及一些 MFC 的特有的新 API 函数。

框架窗口

主窗口通常就是应用程序中第一个接收消息的窗口。CFrameWnd 就是 SDI（Single Document Interface，单文档接口）应用程序的主窗口的基类。CMDIFrameWnd 则为 MDI（Multiple Document Interface，多文档接口）应用程序提供了主框架窗口，CMDIChildWnd 为 MDI 应用程序提供了子窗口。

CWnd 的派生类：对话框和控件

CWnd 是所有 Windows 可视界面对象的滋生地。CDialog 和所有的控件（CButton、CListBox、CEdit 等等）都是从 CWnd 派生出来的，所以它们和一般的窗口具有相同的功能，另外还有自己特殊的功能。例如，CListBox 除了有标准的窗口方法外，还有填满列表框的方

法、得到当前选项的方法以及设置一个选项的方法等等。

对话框

CDialog 类支持 MFC 的对话框，你可以创建模态的(modal)对话框或非模态的(modeless)对话框。除了可以创建自己的对话框，MFC 还提供了一套封装了 Windows 公用对话框的类，这样你就不必再自己重新写了。这些公用对话框在不同的应用程序中有相似的外表，它们包括：

- CFileDialog——从某个目录下选择一个文件。
- CColorDialog——选择一个指定的颜色。
- CFontDialog——选择一种字体。
- CPrintDialog——处理打印机的安装和打印。
- CFindReplaceDialog——为查找和替换选择正文。

对话框数据交换和验证

当在一般的 C/SDK 程序中编写对话框时，不得不编写初始化控件的代码和 OK 按钮被点击时从控件采集数据的代码。MFC 提供了一种初始化对话框控件和自动从对话框端提取数据的方法。对话框数据的交换和验证（DDX/DDV）都是通过 CDataExchange 类实现的。

属性页

使用 Visual C++，你可能已经发现了属性页（property sheet）或者标签对话框（tabbed dialog）。它们对在你的应用程序中减少对话框个数很有帮助，你可以将它们按层次放在一个对话框中。MFC 通过两个类来支持属性页：CPropertySheet 和 CPropertyPage。

控件

MFC 将每个基础 Windows 控件封装为它们自己的类。所有从 CWnd 对象派生出来的控件如下：

- CButton 和 CBitmapButton——标准的 Windows 按钮和带有位图的定制按钮。
- CComboBox——标准的 Windows 组合框。
- CEdit——标准的 Windows 编辑控件。
- CScrollBar——标准的 Windows 滚动条。
- CListBox——标准的 Windows 列表框。
- CCheckListBox——一个包含了按钮链表的特定列表框。
- CStatic——标准的 Windows 静态文字控件。
- CVBControl——Visual Basic 控件（只有 16 位的 MFC 才有）。

除了这些标准控件之外，MFC 还在 CWnd 类中增加了自绘画（owner draw）的方法，方便了创建自绘画控件。

MFC 还包括那些封装了新 Windows 公共控件的控件类。这些类有：

- CAnimateCtrl——播放动画的控件。
- CDragListBox——CListBox 的派生类，你可以在这个列表框中拖动和去掉选项。
- CHeaderCtrl——和 CListCtrl 一起来显示柱状信息。
- CHotKeyCtrl——为从用户获得键序列提供接口（Alt-Backspace-Delete）。

- CImageList——一个 CObject 的派生类，它为你维护图像集合。
- CListCtrl——显示一个链表项的图形链表（类似 Explorer）。
- CProgressCtrl——显示一个进度条。
- CRichEditCtrl——一个丰富的编辑控件，它理解一些 RTF 格式的概念，而且允许使用多字体、多颜色等等。
- CSliderCtrl——一个在某个值范围内进行选择的滚动条。
- CSpinButtonCtrl——微调控制项。
- CStatusBarCtrl——状态栏。
- CTabCtrl——属性页控件。
- CToolBarCtrl——实现一个工具栏。
- CToolTipCtrl——提供工具提示。
- CTreeCtrl——一个类似 Explorer 的树控件。

菜单

Windows 的菜单由 CMenu 类来处理。所有的菜单功能都封装在 CMenu 中，包括菜单句柄和菜单载入方法、更新方法、跟踪方法以及关闭菜单的方法。CMenu 类还支持自绘画菜单（owner-draw menu）。

GDI 支持和绘画对象

MFC 还用大量的类支持 Windows GDI（Graphics Device Interface，图形设备接口）。GDI 的核心是 DC（Device Context，设备环境），在 MFC 中用 CDC 类表示设备环境。在 MFC 中创建一个设备环境和创建 CDC 的一个实例一样简单。设备环境类使用起来很方便，因为实际的设备环境会在调用设备环境的析构函数（destructor）时被释放。这对于那些常常在工作完成时忘记释放设备环境的人（当然，我们从来没有过）来说是个好消息。你会发现与设备环境相关的 API 函数都是 CDC 的成员函数。除了 CDC 以外，MFC 还提供了几个额外的设备环境类，它们有特殊用途：

- CPaintDC——封装了处理 WM_PAINT 消息时所要使用的 BeginPaint()和 EndPaint()两个调用。
- CWindowDC——封装了与整个窗口相关的设备环境。
- CClientDC——封装了与窗口中客户区有关的设备环境。
- CMetaFileDC——为元文件（metafile）封装了设备环境。

所有的 Windows 绘画对象在 MFC 中也以类似的形式表示。CFont、CPen、CBrush、CBitmap、CPalette 和 CRgn 这些类都是从 CGdiObject 类中派生出来的，CGdiObject 为所有的 GDI 对象提供了基础功能。

应用程序框架类

现在我们看以下最麻烦的部分：框架类。这些新的特性（MFC 1.0 中没有）很大程度上改善了作为应用程序框架的 MFC。MFC 1.0 帮助将 Windows API 组织成可管理的块，而 MFC 2.0 提供了更系统、更健壮的方法，使用实际框架来创建标准的 Windows 应用程序。应用程序框架类是 MFC 2.0 的内容。在之后的 MFC 的演化过程中，它们仍然是框架的重要部分。

文档/视图结构

文档/视图（document/view）结构可能是 MFC 1.0 和 MFC 2.0 之间最大的区别。文档/视图结构背后的概念是将数据和数据的表示形式分隔开。换句话说，对于某个数据集合，你可以为它创建多个视图。电子数据表就是一个很典型的例子。一个简单的电子数据表包含一些行和列，你可以从这些数据产生不同的图形。当修改电子数据表中的某些内容时，它就会反应到图形上。MFC 提供了几个实现了文档/视图结构的类，包括：

- CDocTemplate、CSingleDocTemplate 和 CMultiDocTemplate——文档模板是将文档和其视图粘合在一起的黏合剂。它是文档、视图以及框架窗口之间的联络点。CSingleDocTemplate 在一个时刻只支持一个文档，而 CMultiDocTemplate 同时支持多个文档。它们都是从 CDocTemplate 类中派生出来的。
- CDocument ——处理应用程序中数据的类。典型的文档可能包括电子数据表、数据库和文字处理过程。任何使用 File/Open 命令打开的东西都是你的文档（虽然 CDocument 不仅仅限于基于磁盘的数据）。当从 CDocument 派生一个新类时，你要增加一些变量保存数据，增加代码来读取和修改数据，还要增加一些代码来将数据写回到磁盘上。
- CView——CView 代表在屏幕上看到的窗口。CView 是从 CWnd 派生出来的，它也是你向外界显示数据的方式。视图将数据交给输出设备，例如屏幕和打印机。在 C/SDK 程序中，当你的应用程序接收到 WM_PAINT 消息时，你需要处理屏幕绘画，而在其他情况下则处理打印。MFC 将这些虚化了，所以无论视图被传送到哪里，都会调用 CView 的 OnDraw 的方法。有了 CView，使得在屏幕上绘制数据的代码和在打印机上打印和打印预览时显示图像的代码是完全相同的，事实上它们只是使用了不同的设备环境。因为 CView 也是从 CCmdTarget 派生出来的，所以它还可以接收用户输入。

上下文相关帮助

以前，在市场上，上下文相关帮助（context-sensitive help）将你的应用程序和其他人的区分开来。现在却不再是这样。提供在线帮助逐渐成为一种义务，所以 MFC 有很多在线的上下文相关帮助。MFC 支持以下两种获取在线帮助的方法：

- Help 菜单——从菜单中调用帮助。
- F1 键——为目前的任务寻求帮助。
- Shift F1——为用户点击的下一个项寻求帮助。

高层次抽象

微软在 MFC 2.0 中增加了许多有用的特性。其中大多数你不需要深入了解——只有当你要研究整个 MFC 的文档/视图结构时才需要。

增强的视图

根据自己的数据去建立新的视图对你来说不是必需的。MFC 提供了很多封装好的特殊 CView，包括滚动视图（scrolling view）、窗体视图（form view）和编辑视图（edit view）。

滚动视图

如果你的数据太大，不能全在屏幕中显示，就可以使用 CScrollView。CScrollView 就像一页视窗，你可以通过操纵滚动条来选择要看的内容。你还可以使视图随着显示它的窗口的大小而改变，或者也可以自己实现滚动视图的逻辑。

窗体视图

Windows 对话框管理器的一大限制就是，对于窗体处理过程中所需要的很多内容它都不支持。一个对话框中控件的个数必须要适合一屏（或者 255 个控件，开始时是这样的）。而且，对话框不支持滚动、打印和对同一数据的多窗体。这的确是一个问题，因为 Windows 的一个与日俱增的使用便是保存在线数据项（host on-line data-entry）应用程序。大量的数据项操作（例如保险公司）需要在同时输入几页的数据。MFC 利用窗体视图解决了这个问题，CFormView 是一个合并了对话框模板的滚动视图。就像一般的对话框那样，你可以在上面放置很多控件（例如编辑控件、列表框、单选按钮等等）。因为 CFormView 是从 CScrollView 派生出来的，所以它具有 CScrollView 所有的性质，包括滚动、命令处理和通过文档模板管理文档等。

控件视图

MFC 4.0 还增加了几个新的 CView 的派生类，它们都是基本的控件，同时结合了 CView 的特性，例如打印和打印预览特性。

- CEditView——MFC 的 CEditView 就像一个增加了新特性的编辑控件，它完成编辑控件所能完成的一切任务，而且还可以做得更多。CEditView 有编辑控件的所有特性，并且还可以执行类似剪切、拷贝、粘贴、撤消和查找/替换等操作。因为它是从 CView 派生出来的，所以还共享了一般视图的功能。CEditView 使得在你的应用程序中添加一个文本编辑器变得很容易。（这个视图在 MFC 2.0 中就有，而其他几个视图都是 MFC 4.0 新增加的。）
- CListView——CListView 使得在一个视图中可以有一个链表。它可以是字符串链表或者你所希望的任何东西的链表。
- CRichEditView——流行的 Rich Text Edit 通用控件的 CView 版本。
- CTreeView——在视图中提供了一个类似 Explorer 的树。

分割窗口（Splitter Window）

MFC 解决了同一数据在同一窗口内有不同视图的问题，它使用了独立的、可缩放的窗格（pane）。CSplitterWnd 类支持这个界面。有两种分割窗口：静态的和动态的。静态的分割窗口有预定义的窗格数目，而且窗格的数目和排列是不能修改的，虽然每个窗格可以显示不同类型的视图。而在动态的分割窗口中，窗格的数目和顺序都是可以变化的，而且每个窗格必须显示同一类型的视图。

控制栏

MFC 在 CControlBar 类中增加了一整套新的界面对象。因为它们都是从 CCmdUI 派生出来的，所以它们都能够接收命令消息。控制栏有 3 种：工具栏（toolbar）、状态栏（status bar）和对话栏（dialog bar）。

工具栏

CToolBar 代表应用程序工具栏（也可能有其他各种名字）。工具栏就是一行图标，它和菜单一样都是一种选择。MFC 的 CToolBar 使得在应用程序中增加一个工具栏很容易，这与 Windows 里 Word 的工具栏和 Visual C++的工具栏差不多。

状态栏

除了工具栏，大多数应用程序还有一个状态栏。通常状态栏显示一些信息，例如某些特定键（Caps Lock、Scroll Lock、Num Lock 等等）的状态和对当前菜单选择的简短文字介绍，状态栏还可以定制为显示其他信息甚至是位图。

对话栏

除了工具栏和状态栏，对话栏也是 Visual C++新增加的。它和工具栏很像，但它包含的是对话框控件，而不是位图。

操作系统扩展

自从 2.5 版开始，MFC 支持一些操作系统扩展，包括 OLE、ODBC、简单 MAPI 和 WinSock。下面我们来看一些因为支持这些扩展而引入的类。

OLE 支持：OLE 文档

使用 MFC 的一个重要理由就是它支持 OLE。OLE 越来越重要，尤其在微软开始强调在它的操作系统中使用 OLE 时。MFC 为 OLE 技术提供了很多支持：复合文档（compound document）、自动化（automation）和 OLE 控件。MFC 还将 OLE 数据传输很好地封装起来。这里给出了在创建一个支持 OLE 复合文档的文档时需要使用的类。

- CDocItem——MFC 的 COleClientItem 和 COleServerItem 类的基类。
- COleServerItem——表示与嵌入或链接的 OLE 项的连接的服务器端。
- COleClientItem——表示与嵌入或链接的 OLE 项的连接的容器（container）端。
- COleDocument——是 MFC 对复合文档支持的核心。除了维护应用程序的本地数据之外，它还维护了一个 CDocItem 对象链表。Document 是从这个类派生的，而且可以保存嵌入和链接的 OLE 项。
- COleLinkingDoc——包含一些链接，这些链接指向嵌入在其他地方的项。
- COleServerDoc——由复合文档综合体的服务器端应用程序使用。
- COleIPFrameWnd——为了成为复合文档服务器，应用程序必须有两种不同的框架窗口：（1）通常的框架窗口（当应用程序在本地模式下运行时显示在屏幕上的窗口）；（2）应用程序在恰当位置显示时所使用的框架窗口（也就是当用户调用一个复合文档内部的可视化编辑操作时）。COleIPFrameWnd 封装了复合文档服务器的部分功能。

OLE 支持：类厂（class factory）

每个要对外暴露接口的 OLE 对象都要有一个类厂。类厂位于 OLE 服务器中，它会创建一个 OLE 对象的实例来代表服务器。组成 OLE 类厂支持的两个类是 COleObjectFactory 和 COleTemplateServer。

- COleObjectFactory——为那些需要类厂，但又不是面向文档的 MFC 应用程序实现

类厂。

- COleTemplateServer——从 COleObjectFactory 直接派生出来的类，它为那些面向文档的、能使用 OLE 的 MFC 应用程序（例如复合文档服务器）实现类厂。

OLE 支持：自动化

自动化是 OLE 的一个非常重要的特性。使用自动化，开发人员可以将对象开放，对象的属性和方法都可以被那些使用比 C++更松弛的开发环境（例如 Visual Basic）的开发者所使用。MFC 通过 CCmdTarget 类，使用一种“分发映射表（dispatch map）”的机制来支持自动化。

OLE 支持：统一数据传输

OLE 数据传输（又称统一数据传输）由任何实现了 IDataObject 接口的对象完成。如果你用 MFC 编程，则不必自己实现 IDataObject，而只要使用 MFC 对统一数据传输的支持。

- COleDataSource——每次数据传输由两部分组成：源和目的地。在 MFC 程序中，数据传输由 COleDataSource 类完成初始化。这个类可以用于剪贴板（clip-board）传输，也可以用于拖放（drag-and-drop）传输。
- COleDataObject——数据传输的另一端，目的地，通常使用 COleDataObject 表示。
- COleDropSource——定制“拖放”操作时有用。
- COleDropTarget——每当对创建一个接收拖放数据的窗口感兴趣时，你不得不用 MFC 注册这个窗口的兴趣，这就是 COleDropTarget 的用处。

OLE 支持：OLE 控件

OLE 控件正在迅速成为最重要的 OLE 新技术之一。VBX（Visual Basic 控件）已经过时了，很快就要为 OLE 控件所替代。OLE 控件实现了大部分 OLE 技术。它们是可以嵌在 OLE 文档内部的自动化对象。OLE 控件不但支持自动化输入接口（incoming interface），还实现了大量将事件传给宿主进程的输出接口（outgoing interface）。MFC 有几个新增加的类来支持 OLE 控件。

- COleControl——从 CWnd 派生出来，是 OLE 控件的基类。它扩展了基本的 CWnd 功能，能处理与控件相关的特定问题。
- COlePropertyPage——从 Dialog 派生出来，是 OLE 属性页的基类，用于修改控件的属性。
- COleControlModule——从 CWinApp 派生出来，是保存 OLE 控件的动态链接库（dynamic link library）的基类。它与 MFC 程序中的 CWinApp 很类似，负责执行初始化和 OLE 控件特有的各种任务。
- COleObjectFactoryEx——它从 COleClassFactory 派生出来，并扩展了 COleClassFactory。它完成了一般类厂对象所完成的一切，还支持认证。在 MFC 3.0 中，它是独立的类，但当 MFC 的控件开发包扩展到 MFC，并合并到 MFC 的核心部分时，它才合并到 COleObjectFactory 中。
- COleConnectionPoint——从 CCmdTarget 派生出来，代表到其他 OLE 对象的输出接口，用于事件触发和向容器发出修改通知。
- CPropExchange——它与 MFC 中用于标准 DDX/DDV 的 CDataExchange 类似，为属

性交换建立环境，并在控件与容器之间帮助交换属性。

- CFontHolder——该类封装了 Windows 的字体类。它实现了 OLE 的 IFont 接口，用于 Font 的常备属性。
- CPictureHolder——实现了“图像属性”。它以多态的方式封装了一个位图、图标或元文件。

ODBC 支持

从 2.5 版本开始，MFC 提供了 ODBC（Open Database Connectivity，开放式数据库互联）支持。下面是 ODBC 类的链表：

- CDatabase——封装了对数据源的连接，通过它你可以对数据源进行操作。
- CRecordset——封装了一些从数据源中选出的记录。记录集允许从记录到记录的滚动，更新记录（增加、编辑、删除），使用过滤器选择记录，对选出的记录进行排序，根据在运行时获得或计算得到的信息将选出的记录参数化。
- CFieldExchange——提供环境信息来支持 RFX（Record Field Exchange，记录域交换）。RFX 会在记录集对象的域数据成员以及参数数据成员与数据源的相应的链表之间交换数据。
- CLongBinary——封装了一个句柄，以便存储大的二进制对象（或 BLOB），例如位图。它主要用于管理存储于数据库表中的大数据对象。
- CRecordView——提供一个连接到记录集对象的窗体视图。DDX 机制负责在记录集和记录视图的控件之间交换数据。

DAO 支持

在 MFC 4.0 中，微软推出另一个数据库访问策略，叫做 DAO（Data Access Object，数据访问对象）。它与 ODBC 不同，因为它针对 Microsoft Jet 数据库引擎，而 ODBC 是针对所有数据库的。用于支持 DAO 的 MFC 类包括：

- CDaoWorkspace——管理命名的、有密码保护的数据库会话。
- CDaoDatabase——连接到某个数据库上，你可以通过它访问数据库。
- CDaoRecordset——从数据源中选出的记录集。
- CDaoRecordView——在控件中显示数据库记录。
- CDaoQueryDef——一个查询定义，通常存放在数据库里。
- CDaoTableDef——一个基表（base table）或附加表（attached table）的存储定义。
- CDaoException——DAO 类产生的异常情况。
- CDaoFieldExchange——支持由 DAO 数据库类使用的 DAO 记录域交换例程。

MAPI 支持

其实，没有完全支持 MAPI 的 MFC 类。它是嵌在 CDocument 内部的，你可以调用 CDocument::OnUpdateFileSendMail()成员函数来通过电子邮件发送一个文档。

WinSock 支持

有两个类包装了 Windows 套接字（Socket）接口。CAsyncSocket 类是一个简单的包装层。CSocket 提供高层抽象来处理套接字。

Pen Windows 支持

MFC 用两个新的编辑类来支持 Pen Windows：CHEdit 和 CBEdit。前者在 Microsoft Windows 中为 Pen Computing 封装了手写编辑控件，后者与前者类似，只是它会显示一个方格，表示每个字母应输入的位置。这使得识别器（recognizer）不容易认出用户输写的内容。Pen 支持只限于 16 位平台。

结 语

现在你已经对 MFC 做了一次短暂的旅行。你会发现微软不仅仅包含了 Windows API，还增加了文档/视图结构和几种很有用的高层抽象。我们下一步就是比较使用两种方式写的同一个程序：一种是用 C 和 SDK 写的，另一种是用 C++和 MFC 写的。这样，你会发现 MFC 有那么多的样本代码，而且它支持的内容是你在使用 C 和 SDK 时所不曾想到的。下一章讲 MFC 对 Windows 应用程序的基本支持，包括 WinMain()、消息泵和 MFC 的消息路由选择机制。

CHAPTER

2 基本的 Windows 支持

在上一章结束后请稍作休息。MFC 的确很广，不是吗？它有 200 多个类，而且有着丰富的特性。在概括性地浏览这些类后，你可能已经知道哪些类是你常用的，哪些只是偶尔使用，可能还有一些你永远不会使用的类。

MFC 很大，惟一可以把它彻底弄懂的办法就是将它分成小块。这一章主要讨论 MFC 对 Windows 基本应用程序的支持。无论你怎么分，一个 Windows 程序终究还是一个 Windows 程序。它与你使用的语言和框架无关。不管一个 Windows 程序是由 C、C++或 Delphi 写的，也不管它是否使用了类似 MFC 或 Object Window Library（OWL）这样的应用程序框架来创建，所有的 Windows 程序都做着相同的基本事情——例如，注册窗口类和启动消息循环，当然这些还仅是 Windows 程序的一部分任务。本章将介绍 MFC 如何实现这些基本任务。

你已经见到了类的概要，现在应该知道 MFC 的层次了。下面介绍 MFC 的基本 Windows 支持中最基本的方面：WinMain()和消息处理。我们会自上而下逐步挖掘整个框架。这样，我们就会多次访问同一个类，例如，CWnd 除了简单地封装大多数面向窗口的 API 函数外，还要处理文档/视图结构，并支持 OLE 2.0。为了完成这些，CWnd 就有了很多方面。我们会分别地看每个方面，在其他章中讨论 MFC 的文档/视图结构和 OLE 支持时，还会提到 CWnd。

首先让我们看一下 MFC 程序和 C/SDK 程序都做了些什么。根据 MFC 实际封装 Windows 应用程序的方式，你会看到 MFC 是如何处理以下问题的：

- 应用程序自身。
- Windows。
- 消息处理。
- GDI（Graphics Device Interface，图形设备接口）。

为了更清楚地观察，我们比较一个只监视鼠标左键点击的基本 Windows 程序的两个版本。一个版本是用 C/SDK 写的，另一个是用 MFC 写的。

MFC 与 C/SDK

Windows 程序只能有一种写法，程序自身惟一可能发生变化的是它处理消息的方式。在很多方面，这句话是对的，但程序要想运行在 Windows 下必须要满足一些要求，这些要求是

一致的。每个 Windows 应用程序都有两个主要部分：主应用程序自身和至少一个能进行消息处理的窗口。而且大部分代码不会随应用程序而发生变化。事实上，SDK 文档中带的 Guide to Programming 手册在考虑所举的一个例子 GENERIC.C 时就得到一个结论："你可以使用 Generic.c 作为搭建应用程序的模板，而只需要拷贝源文件，重命名，然后修改相关的函数名，插入新的代码。"

可见，微软也鼓励使用剪贴这种 Windows 开发方式。这种方法有时确实很有用。一部分使用 SDK 用 C++写的 Windows 程序（那些只是将窗口显示在屏幕上而不做任何其他事的程序）都有大约 80 行的样本代码。即使你喝了很多咖啡而且打字足够快，80 行的样本代码量还是不小的，而且输入那么多代码难免会出错，但是这些代码又是必需的，所以你最好还是从别处找到运行正确的代码，拷贝过来，再加以修改。

然而这样做也很耗时间，效率也不高。我们现在已经是 OOP 时代了，应该采用不同的方法，而不应该再在应用程序之间剪切粘贴。从现在开始，你要使用 C++类——继承应用程序的一部分，并且只修改那些需要修改的地方。

所有的样本代码

为什么这些代码是必需的？Windows 应用程序都用它做了些什么？最直接的答案是事件驱动的操作系统强加的。Windows 应用程序需要大量的安装，这都是因为 Windows 是一个事件驱动的环境，应用程序需要大量代码来接收和处理事件。

Windows 位于硬件和应用程序之间，每当有某个应用程序所感兴趣的事情发生时，Windows 都会通知应用程序。例如，假设鼠标左键被按下，一旦 Windows 发现了这个事件，就有必要让那些应用程序知道按键事件。这样应用程序就可以用各种方法处理事件或者忽略事件。

Windows 编程中引入的这些代码是用于处理消息。考虑一下要完成的任务：

- Windows 应用程序需要建立一个消息处理函数（Message Handler），并把它插入到 Windows 中（通过 RegisterClass()这个 API 函数），这样 Windows 就知道对于特定的应用程序，将消息送到哪里处理。
- Windows 需要记录应用程序的特定实例。
- 应用程序向 Windows 请求消息队列中的下一个消息，并将消息分发给合适的消息处理函数，然后再请求下一个消息。
- 这样的过程持续到应用程序结束为止（无论什么原因结束）。

记住这些，你就会明白为什么一个 Windows 程序要有这么多代码，但成为 Windows 应用程序之前，应用程序都需要做什么？

- 应用程序首先要有一个 WinMain()函数，就像 C 程序都有一个 main()函数一样。每个 Windows 程序都必须有一个 WinMain()函数，它是程序的入口点，应用程序就是通过它从 Windows 获得运行所必要的信息的。这些必要信息包括应用程序的当前实例的句柄、应用程序的上一个运行实例的句柄、所有的命令行参数以及如何显示窗口等（最小、最大和适中等等）。
- 应用程序至少要注册一个窗口类作为主窗口，无法想像没有窗口的 Windows 程序会是什么样。将用户界面显示在屏幕上是 Windows 的工作。

- 应用程序要建立消息循环。如果消息是 Windows 应用程序的血液，那消息循环就是它的心脏。在 Windows 里，应用程序要负责建立消息泵。也就是说，应用程序需要建立循环来提取消息。
- 大多数应用程序需要一些初始化和安装。初始化有两种形式：针对应用程序的初始化和针对实例的初始化。应用程序应该提供任何必要的初始化步骤。
- 应用程序必须提供消息处理函数。这个窗口过程（window procedure）函数至少要处理 WM_DESTROY 消息（除非应用程序永远不终止）。

把这些代码封装成一个类将是一件很好的事，这样你就不用总是对着键盘敲那么多代码。这也就是 MFC 所做的。

比较代码

看 MFC 程序如何工作的最好方法就是比较用 MFC 写的程序和用标准 C/SDK 写的程序。程序清单 2-1 给出了一个小的 C/SDK 应用程序，你可能从前也写过这样的代码。它只是初始化了应用程序，启动了一个主窗口，启动了消息循环，并且等待 WM_LBUTTONDOWN 消息和 WM_DESTROY 消息。虽然这是一个 Windows 程序，但是它说明了这种程序的基本点。你可以将它作为研究 MFC 的起点。

程序清单 2-1　程序 MINIMAL.C（用 C/SDK 编写）

```
/*********************************************************************
** Minimal.c— This is a minimal Windows program. It initializes
** the program, starts a message loop, watches for WM_LBUTTONDOWN
** messages, and waits for a WM_DESTROY message.
*/
#include "windows.h"

HANDLE hInst;  /* current instance            */

long CALLBACK __export MainWndProc(HANDLE hWnd,
                                   UINT message,
                                   WPARAM wParam,
                                   LPARAM lParam) {

  switch(message) {
     case WM_LBUTTONDOWN:
        MessageBox(hWnd, "Left mouse button clicked...", NULL, MB_OK);
        break;
     case WM_DESTROY:
        PostQuitMessage(0);
        break;

  default:    /* Passes it on if unproccessed    */
      return (DefWindowProc(hWnd, message, wParam, lParam));
  }
  return (NULL);
}

BOOL InitApplication(HANDLE hInstance) {
```

```
30   WNDCLASS  wc;
31
32   wc.style = NULL;                /* Class style(s).                    */
33   wc.lpfnWndProc = MainWndProc; /* Function to receive messages for */
34                                  /* windows of this class.             */
35   wc.cbClsExtra = 0;              /* No per-class extra data.           */
36   wc.cbWndExtra = 0;              /* No per-window extra data.          */
37   wc.hInstance = hInstance;       /* Application that owns the class. */
38   wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
39   wc.hCursor = LoadCursor(NULL, IDC_ARROW);
40   wc.hbrBackground = GetStockObject(WHITE_BRUSH);
41   wc.lpszMenuName =  NULL;    /* Name of menu */
42   wc.lpszClassName = "MinimalWClass"; /* Name of window class */
43
44   return (RegisterClass(&wc));
45 }
46
47 BOOL InitInstance(HANDLE hInstance, int nCmdShow) {
48   HWND            hWnd;               /* Main window handle.  */
49
50   hInst = hInstance;
51
52   hWnd = CreateWindow(
53       "MinimalWClass",      /* Window Class                    */
54       "Minimal",            /* Caption                         */
55       WS_OVERLAPPEDWINDOW,  /* Window style.                    /
56       CW_USEDEFAULT,        /* Default horizontal pos.          /
57       CW_USEDEFAULT,        /* Default vertical pos.           */
58       CW_USEDEFAULT,        /* Default width.                  */
59       CW_USEDEFAULT,        /* Default height.                 */
60       NULL,                 /* No parent.                       */
61       NULL,                 /* Use the window class menu.      */
62       hInstance,            /* This instance owns the window.*/
63       NULL);                /* Not needed.                      */
64
65   if (!hWnd)
67      return (FALSE);
68
69   /* Show window, update window */
70
71   ShowWindow(hWnd, nCmdShow);
72   UpdateWindow(hWnd);
73   return (TRUE);
74 }
75
76 int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
77                    LPSTR lpCmdLine, int nCmdShow) {
78   MSG msg;                        /* message              */
79
80   if (!hPrevInstance) {      /* First instance of app?  */
81      if (!InitApplication(hInstance))  /* Shared stuff */
82          return (FALSE);  /* cannot initialize */
83   }
84
```

```
85    if (!InitInstance(hInstance, nCmdShow))
86        return (FALSE);
87
88    while (GetMessage(&msg, NULL, NULL, NULL)) {
89       TranslateMessage(&msg);
90       DispatchMessage(&msg);
91    }
92
93    return (msg.wParam);      /* Returns the value from PostQuitMessage */
94 }
```

好多代码！用 MFC 写同样的程序，见程序清单 2-2。

程序清单 2-2　用 C++/MFC 写的程序 MINIMAL.CPP

```
 1 #include <afxwin.h>
 2
 3 // Define an application class derived from CWinApp
 4 class CGenericApp : public CWinApp {
 5 public:
 6    virtual BOOL InitInstance();
 7 };
 8
 9 class CGenericWindow : public CFrameWnd {
10 public:
11    CGenericWindow() {
12       Create(NULL, "Generic");
13    }
14
15    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
16    DECLARE_MESSAGE_MAP()
17 };
18
19 BEGIN_MESSAGE_MAP(CGenericWindow, CFrameWnd)
20    ON_WM_LBUTTONDOWN()
21 END_MESSAGE_MAP()
22
23 void CGenericWindow::OnLButtonDown(UINT nFlags, CPoint point) {
24    MessageBox( "Left mouse button pressed...", NULL, MB_OK);
25 }
26
27 BOOL CGenericApp::InitInstance() {
28    m_pMainWnd = new CGenericWindow();
29    m_pMainWnd->ShowWindow(m_nCmdShow);
30    m_pMainWnd->UpdateWindow();
31    return TRUE;
32 }
33
34 CGenericApp GenericApp;
```

这样更好了，不是吗？去掉了那些样本代码后，源代码减少为原来的 60%。

究竟这些应用程序发生了什么变化呢？首先，检查一下 C/SDK 的例子。你可以清楚地看到都发生了些什么。它作为一个 Windows 应用程序显然完成了任务。它有一个 WinMain()函数，窗口类在 InitInstance()中注册，主窗口创建了，并显示出来。最后，主窗口有一个窗口

过程（叫做 MainWndProc()），它处理鼠标左键点击事件。这些都没有什么奇怪的。

现在再看一下 MFC 的例子，看起来它似乎少了点儿什么，比如 WinMain()函数没有了，而且好像没有注册任何窗口类，甚至没有消息处理函数，更不像是创建了一个消息循环，但它运行的时候和 C/SDK 版本一样。这看起来不可能，那些代码都跑到哪里去了呢？其实，代码还在哪里，只不过被隐藏了

用 MFC 或者其他应用程序框架（例如 Borland 的 OWL）写 Windows 程序，在很多方面都很实用，尤其当你有很强的 C 和 SDK 背景时。使用 C 和 SDK，一切看起来都很清楚，很容易看到程序的动作。它的缺点是你必须写大量的代码，MFC 的优点就在于它减少了你要写的代码量。事实上，为了运行一个 Windows 小程序，你所要做的就是实例化 CWinApp 对象，创建一个主窗口对象。简单地实例化 CWinApp 就可以使程序自动运行。困难的是你必须相信所有的代码已经在那里，并会像广告里说的那样发挥作用。当你看到 MFC 的内部时，你就会发现所有必要的代码。

当然，在你将 DOS 程序移植为 Windows 程序时，你就必须做一定的修改。因为操作系统一直是你的应用程序的一部分。考虑一下，当在 DOS 下编程时，你或多或少会凌驾于系统之上。DOS 是你的奴隶（虽然一些人会认为整个一代程序员是 DOS 的奴隶），但在 Windows 编程过程中，Windows 和应用程序之间需要变得更加协作。Windows 和应用程序进入一个控制循环彼此相互传递消息。

将程序移植为 MFC 程序也需要进行一次类似的修改。处理一般应用程序的 99%代码都在 MFC 中，然而它不是建立 Windows 过程来捕获消息，而是当事件发生时，便会通知部分应用程序框架。也就是说，消息捕获功能已经安装，你需要做的就是决定重写哪部分代码。

下面让我们来仔细看一下 MFC 应用程序，并找出所有东西在哪里。

WinMain()函数

为了在 Windows 下运行，应用程序必须有 WinMain()函数作为程序的入口点。这也是程序执行针对应用程序和针对实例的初始化以及启动应用程序消息循环的地方。

在 MINIMAL.C 中，你可以在第 76 行到第 94 行看到该函数。如果这是应用程序的第一个实例，WinMain()会调用 InitApplication()，该函数会将窗口类注册为 MinimalWClass。然后 WinMain()调用 InitInstance()，这个函数会根据 MinimalWClass 模型创建并显示一个窗口。最后 WinMain()建立消息循环，该循环从队列中取消息，再分发给合适的窗口，直至接收到 WM_QUIT 消息。

下面再仔细看看 MINIMAL.CPP。它没有 WinMain()函数，那该函数在哪里呢？WinMain()函数实际上已经成为应用程序框架的一部分，你连接框架代码时就已经将它连入了。MFC 的 WinMain()函数和从前的 WinMain()函数一样完成相同的工作，包括初始化应用程序和启动消息循环。

初始化应用程序的一个特定实例

对于每个要运行的应用程序的实例来说，都有一些初始化工作要做。通常是创建和显示主窗口。

在 MINIMAL.C 中，初始化是在第 47 行到第 74 行之间的 InitInstance 函数中完成的（由

WinMain()调用)。InitInstance 在全局变量 hInst 中保存了程序当前实例的句柄，这个值在装载资源时有用。因为 Windows 只在程序开始处把当前实例的句柄交给应用程序，所以如果你要使用它，就只好在这里保存。然后，InitInstance()会创建主窗口，并且按照在命令行中传入的 nCmdShow 参数所指定的方式来在屏幕上显示窗口。

在 MINIMAL.CPP 中，针对实例的初始化在 27 行和 32 行之间，虽然你在这里看不到，但是 CGenericApp 会在 WinMain()中调用它的成员函数 InitInstance()。InitInstance 会创建一个新的主窗口并通知它显示自己。

消息循环

消息循环就是应用程序接收消息和分发消息的地方。在 MINIMAL.C 中，消息循环位于 88 行与 91 行之间。这并不奇怪：循环不停地调用 GetMessage()和 DispatchMessage()，直至遇到了 WM_QUIT 消息。

在 MINIMAL.CPP 中，消息循环又是框架的一部分。它在 CWinApp 的 Run()成员函数被调用时启动，也是 WinMain()的最后一步。MFC 的消息处理函数还提供了其他功能，例如，在队列中没有消息时执行 idle 进程，解释特定的消息时提供 PreTranslateMessage()。然而，在很低的层次上，MFC 会以传统的模式(调用 GetMessage()、TranslateMessage()和 DispatchMessage())通过系统转储和分发消息，然后由 MFC 的命令路由选择结构来处理消息。

消息处理

Windows 应用程序中每个类都要有一个消息处理过程。窗口通过消息处理函数来产生特定的行为。每当 Windows 检测到与某个窗口有关的事件时，它就会产生一条消息，并调用窗口的消息处理函数，其中参数是关于该事件的信息。例如，在 MINIMAL.C 中，主窗口的行为包括对 WM_LBUTTONDOWN 和 WM_DESTROY 消息的响应。注意，消息处理函数（第 10 行到第 27 行）只是用 switch 语句过滤这些消息，如果消息不是 WM_LBUTTONDOWN 和 WM_DESTROY 就调用 DefWindowProc()。其他应用程序可能还要响应菜单命令或鼠标移动命令，这样 Switch 语句就要增加适当的语句来处理新消息。

MINIMAL.CPP 的源代码几乎没有窗口过程。在相应的位置上只有两样东西：

1. 主窗口的一个成员函数，OnLButtonDown（第 23 行到第 25 行）.

2. 消息映射表（Message Map）（第 16 行和第 19 行到 21 行）。MFC 程序没有给窗口指定一个特定的消息处理函数（C/SDK 程序会指定），它使用消息映射表获得命令目标的命令和消息（例如，CWnd 是一个命令目标，即是命令和消息的目标）。消息映射表将特定的命令、消息与处理该消息的成员函数联系在一起。

如果没有窗口过程，MFC 将不能运作。事实上，有一个与 CWnd 相关的窗口过程。如果仔细研究 APPINIT.CPP（它的 MFC 源代码在 Visual C++中），你会发现它注册了 4 个窗口类。尽管有 4 个不同的窗口类，但它们都使用相同的窗口过程（叫做 DefWindowProc()）。这听起来不可信，不是吗？等一下，你会看到 MFC 的另一个窗口过程 AfxWndProc()。尽管这个函数不是通过窗口类注册而直接安装的，但它最终还是使用了 Windows 钩子（我们很快就会看到如何实现）。CWnd 类有一个 WindowProc()成员函数。AfxWndProc()在调用 CWinAPP 的 m_pMainWnd->WindowProc()时结束，这个函数将消息路由到正确的命令目标那里。

所以，尽管窗口类没有显式的窗口过程，但命令路由选择（command-routing）和消息分发（message-dispatch）结构仍然有效地处理了消息。MFC 通过命令路由选择机制处理 WM_COMMAND，该机制会为相应的命令目标得到 WM_COMMAND 消息。MFC 的消息分发机制为相应窗口得到窗口消息，在该窗口中由相应的成员函数处理。

基本的 MFC 应用程序组件

在 MFC 中，就基本结构而言，所有的 Windows 应用程序与前面的 C/SDK 例子很类似。正如前面提到的那样，一个 Windows 应用程序至少由两部分组成：消息泵和窗口过程。MFC 通过将 Windows 应用程序分成两块来支持这一基本 Windows 程序结构。这两块是代表应用程序的一个类和代表窗口的一个类。这是处理问题的一种逻辑方式——翻一下 C/SDK 的源代码你就会发现。C/SDK 程序都包括这两个不同的部分：（1）针对应用程序的部分，它负责初始化，创建一个或多个窗口，维护 GetMessage()..DispatchMessage()循环；（2）针对窗口的部分，负责在窗口上绘画，处理消息等等。在 MFC 中，这两个部分分别对应 CWinApp 和 CWnd 类。

在看完这些基本内容后，我们来进一步看一个 MFC 程序的结构。

CWinApp:应用程序对象

显然，CWinApp 是一个很大的类——它需要完成很多工作。程序清单 2-3 是这个类的精简内容（完整的定义在 AFXWIN.H 中）。

程序清单 2-3 CWinAPP 的伪代码（在 AFXWIN.H 中）

```
class CWinApp : public CWinThread
{
public:

// Constructor
    CWinApp(LPCTSTR lpszAppName = NULL);  // app name defaults to EXE name

// Attributes
    // Startup args (do not change)
    HINSTANCE m_hInstance;
    HINSTANCE m_hPrevInstance;
    LPTSTR m_lpCmdLine;
    int m_nCmdShow;
    LPCTSTR m_pszAppName;  // human readable name
        // advanced attributes omitted

public:  // set in constructor to override default
    LPCTSTR m_pszExeName;        // executable name (no spaces)
    LPCTSTR m_pszHelpFilePath;  // default based on module path
    LPCTSTR m_pszProfileName;   // default based on app name
       // Initialization Operations omitted

public:
    // Cursor wrappers
```

```
    // Icon wrappers
    // INI and profile wrappers

// Running Operations - to be done on a running application
    // Functions for dealing with document templates
    // Functions for dealing with files
    // Printer helpers
    // Command line parsing functions

// Overridables
    // hooks for your initialization code
    virtual BOOL InitApplication();

    // Functions to use when exiting
    // Advanced: MessageBox/Cursor overrides
    // Advanced: Help support

// Command Handlers
protected:
    // Command handlers for OnFileNew(), OnFileOpen(), and OnFilePrintSetup()
    // Context sensitive help functions

protected:
     // General attributes
     // Document/view, profile, and printer attributes and helpers

public: // public for implementation access
    CCommandLineInfo* m_pCmdInfo;

      // other attributes
      // Advanced OLE members

    void SetCurrentHandles();

    // helpers for standard commdlg dialogs
    // overrides for implementation
    virtual BOOL InitInstance();
    virtual int ExitInstance(); // return app exit code
    virtual int Run();
    virtual BOOL OnIdle(LONG lCount); // return TRUE if more idle processing
    virtual LRESULT ProcessWndProcException(CException* e, const MSG* pMsg);

public:
    virtual ~CWinApp();
protected:
    //{{AFX_MSG(CWinApp)
    afx_msg void OnAppExit();
    afx_msg void OnUpdateRecentFileMenu(CCmdUI* pCmdUI);
    afx_msg BOOL OnOpenRecentFile(UINT nID);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

CWinApp 中全是成员函数和成员变量，下面列出了一些重要的函数和变量：

- CWinApp 保存了一些传递给 WinMain()的命令行参数，这些参数包括当前实例的句柄（m_hInstance）、前一个实例的句柄（m_hPrevInstance）、命令行参数（m_lpCmdLine）

以及显示窗口标志（m_nCmdShow）。

- CWinApp 在 m_pszAppName 中保存了应用程序名字的拷贝。
- CWinApp 保存了指向可执行文件名字的指针（m_pszExeName）、指向应用程序帮助文件路径的指针（m_pszHelpFilePath）以及指向应用程序配置文件（profile）名字的指针。我们一会儿会看到如何初始化它们。
- CWinApp 还使用一个叫 CCommandLineInfo 的结构来保存命令行参数。这么多年过去了，已经有了一些命令行参数标准，例如，当 OLE 应用程序作为对象嵌入到 OLE 文档时，Windows 会用命令行参数“/Embedding”启动这个应用程序。如果想让应用程序执行类似打开新文件、打印已有文件和运行自动化代码的操作，就会有标准的命令行参数可以使用。CCommandLineInfo 在一个地方保存了所有标准参数，它是一个小的实用类，但它在你需要得到命令行参数时很有用。它的定义如程序清单 2-4 所示。

程序清单 2-4 CCommandLineInfo 的定义

```
class CCommandLineInfo : public CObject
{
public:
    // Sets default values
    CCommandLineInfo();

    virtual void ParseParam(const char* pszParam, BOOL bFlag, BOOL bLast);

    BOOL m_bShowSplash;
    BOOL m_bRunEmbedded;

      BOOL m_bRunAutomated;
      enum { FileNew, FileOpen, FilePrint, FilePrintTo, FileDDE }
          m_nShellCommand;

      // not valid for FileNew
      CString m_strFileName;

      // valid only for FilePrintTo
      CString m_strPrinterName;
      CString m_strDriverName;
      CString m_strPortName;

      ~CCommandLineInfo();
  };
```

- 每个应用程序都需要进行针对实例的初始化。在 MFC 应用程序中，这部分功能由 CWinApp::InitInstance()完成。MFC 在应用程序中首先调用 InitInstance()，例如，通常是显示主窗口。
- 相应地，大多数应用程序还要执行关闭程序和清除资源的代码。在 MFC 应用程序中，该工作由 ExitInstance()完成。
- 在 MFC 应用程序中，消息泵是 CWinApp 的一部分。调用 CWinApp::Run()会启动标

准的 GetMessage()..DispatchMessage()循环。CWinApp 的消息循环还支持后台处理。

- 像 Windows 这样的事件驱动环境的一大特点是，应用程序运行期间队列中可能没有任何消息。MFC 此时会调用 CWinApp 类的一个叫做 OnIdle()的函数。每当消息队列内容为空时，CWinApp::Run()都会调用 OnIdle()。

这是 CWinApp 的概要，下面让我们看一下 CWnd——所有 MFC 窗口的基类。

CWnd:窗口基类

除了从 CWinApp 派生的对象，MFC Windows 应用程序还包含表示屏幕上窗口的对象，这些对象都是从 CWnd 派生出来的。CWnd 太大了，所以这里就不再显示它的代码，如果你想看它的定义，可以看 AFXWIN.H 提供的两方面功能：（1）包装的一般 Windows API 函数（像 Create()和 ShowWindow()）；（2）提供高层与 MFC 相关的功能，像默认消息处理函数。

包装 Windows API

包装 Windows API 是很容易的部分。CWnd 保存了一个名为 m_hWnd 的成员变量，它表示 API 级的窗口句柄（就是 HWND）。对于初学者而言，CWnd 封装了所有带有窗口句柄的 Windows API 函数，这些函数包括 ShowWindow 和 MoveWindow()。任何时候，如果你看到了某个函数的第一个参数是一个窗口句柄（HWND），你就可以断言它是 CWnd 的一个成员函数。如果某段客户代码调用了 CWnd 派生类的 API 函数，函数的 CWnd 版本会将对象的窗口句柄（m_hWnd）传给标准 API 函数。然而，在这个层次使用 Windows API，你就不会看到那些你应该注意的地方。例如，这里是一段用 C/SDK 写的代码，它负责显示一个窗口：

```
HWND hWnd
...
ShowWindow(hWnd, SW_SHOWNORMAL);
```

使用 MFC 来做相同的事情会如下所示：

```
CWnd* pWnd;
...
pWnd->ShowWindow(SW_SHOWNORMAL);
```

MFC 源代码包括一个叫 AFXWIN2.INL[INL 扩展名表示内嵌（inline）]的文件，这个文件包含了 CWnd 默认行为的代码。CWnd 的大多数 API 包装都是内嵌的。一些 API 不是内嵌的，是因为当 CWnd 指向一个 OLE 控件时，它们需要特殊的处理，因为 OLE 控件太大了而不能内嵌。如果 CWnd 不是一个 OLE 控件，那它的成员函数都是简单的 Win32 API 调用。每个映射成 Windows API 函数的 CWnd 成员都简单地使用 m_hWnd 来调用 API 函数。

然而 CWnd 不仅仅是一些 API 函数的包装层，它还有一些其他特性。例如，它为 MFC 基本程序封装默认窗口行为。

CWnd 的其他特性

首先，注意一下 CWnd 是如何派生的：

```
CObject->CCmdTarget->CWnd
```

CWnd 的父类是 CCmdTarget，CCmdTarget 又是从 CObject 派生出来的。从 CObject 派生使 CWnd 保留了一些强制的属性，例如动态运行时信息与序列化。我们将在第 5 章讨论这些。从 CCmdTarget 派生使 CWnd 能够深入 MFC 的消息路由选择方案，这也是本章的重点。

另外，CWnd 还定义了 MFC 对大多数窗口消息的默认响应。再看一下程序清单 2-1 中 C/SDK 例子的 switch 语句，你会发现消息从不被处理，MINIMAL 的窗口过程将消息处理委托给默认窗口过程 DefWindowProc()。MFC 处理不需要的窗口消息时采用相同的方法。请记住，MFC 处理消息的方式与标准的 SDK 程序不同。MFC 将消息映射到处理器函数。CWnd 有处理多数窗口消息的成员变量和成员函数。它们的一般形式是“On...()”。例如，CWnd 处理 WM_PAINT 消息的函数叫做 OnPaint()。简单地查看一下 AFXWIN2.INL，你会发现 CWnd 的消息处理函数只是简单地委托给 CWnd::Default()。这样最终就会调用 CWnd::DefWindowProc()。这个函数也是大多数消息的终点。

在更深入地研究 MFC 内幕之前，我们来看使用 MFC 进行 Windows 开发的一个经常被误解的方面：窗口句柄与封装了它的 C++类之间的区别。

将窗口句柄转化成窗口对象

理解本地窗口句柄（HWND）和表示窗口的 MFC 对象之间的区别很重要。在最高级别上，MFC 是与 CWnd 对象共同工作的。然而本地的 Windows 代码还使用窗口句柄。例如，当 Windows 调用一个窗口过程时，Windows 会将一个窗口句柄作为第一个参数传递给窗口过程。而 MFC 的消息分发机制又会与 CWnd 的派生类合作。为了使消息分发机制更好地工作，MFC 必须指明对于特定句柄有哪个 CWnd 派生对象与之关联。MFC 使用了一个叫做 CHandleMap 的类将 CWnd 派生对象与窗口句柄关联在一起。

CHandMap:没有文档说明的窗口句柄映射表类

这个类将窗口句柄映射成 MFC 的 Windows 对象。这就意味着，如果给定了一个句柄，你就可以找到与之关联的对象。MFC 需要一个这样的机制来简化如下的难题：Windows 使用句柄，而 MFC 使用对象。当 Windows 调用一个回调（call back）函数时，它会将窗口句柄作为一个参数传入。MFC 需要将这个窗口句柄转化成它能使用的东西：一个 CWnd 的派生类。

对于 MFC 对象来说，尽可能地将本地窗口句柄包装起来（这也正是 MFC 所做的）很重要。因为 MFC 经常把本地句柄和 MFC 包装层混合在一起，所以框架就需要一个窗口句柄与包装它的 C++对象之间的统一映射。

CHandleMap 有两个类型为 CMapPtrToPtr 的成员变量，分别是 m_permanentMap 和 m_temporaryMap。CHandleMap 使用 CMapPtrToPtr 的性能来维护窗口句柄与关联的 MFC 对象之间的关系。m_permanentMap 表示永久映射表，它保存了程序运行过程中句柄/对象映射表。m_temporaryMap 表示临时映射表，它仅在消息存在的过程中才存在。

永久映射表保存了那些由开发人员显式创建的 C++对象。每当创建一个 CWnd 的派生类时，MFC 都会在永久目录下插入一条映射记录。调用 CWnd 的 OnNcDestroy()时就会从永久目录下删除一条映射记录。

前面已经提到，MFC 中使用的是 CWnd 的派生类，而不是 HWND。有时句柄可能会传递给一个不在永久目录下的 MFC 接口（也就是说，它不是由开发人员显式创建的）。在这种

情况下，MFC 会分配一个临时对象来包装句柄，并将这个映射保存到临时目录下。这些临时包装对象在空闲时间里会被从临时映射表中删除。

CWnd::GetActiveWindow()是一个这方面的好例子。它是 CWnd 的一个静态成员函数，它包装了 GetActiveWindow() API 函数。然而它不仅返回当前活动窗口的窗口句柄（如::GetActiveWindow()所做的），它还会返回一个指向 CWnd 对象的指针。当然当前的活动窗口可能是另一个应用程序，这时 CWnd::GetActiveWindow()不得不创建一个临时 CWnd 对象来包装窗口句柄。CWnd::GetActiveWindow()只是简单地调用了全局版本的 GetActiveWindow()来得到一个窗口句柄，然后，::GetActiveWindow()会使用 CWnd::FromHandle()来创建一个包装了活动窗口句柄的 CWnd 对象。CWnd::FromHandle()首先在永久映射表中查找窗口句柄，然后到临时映射表中查找。如果 CWnd::FromHandle()在任何一个表中找到了句柄，它就会分配一个 CWnd 对象，然后将句柄与这个对象关联起来。

除了 CWnd 派生对象与 HWND 之间的映射以外，还有 4 个从 MFC 类到本地窗口句柄的映射。MFC 定义了 4 个映射表，使它们与 MFC 中每个基于句柄的类关联在一起。句柄映射表位于 AFX_MODULE_THREAD_STATE 结构中，这个结构我们会在后面详细讨论。这里给出所有的 MFC 对象映射表的句柄：

- m_pmapHWND：窗口句柄与 CWnd 对象之间的映射表。
- m_pmapHMENU：菜单句柄到 CMenu 对象的映射表。
- m_pmapHDC：设备环境句柄到 CDC 对象的映射表。
- m_pmapHGDIOBJ：GDI 对象句柄到 CGDI 对象的映射表。
- m_mapHIMAGELIST：图象链表句柄到 CImageList 对象的映射表。

在给定本地窗口句柄的情况下，找到这些 MFC 类中的一个很容易。每个参与了句柄映射方案的 MFC 类都包括一个 FromHandle 函数，这个函数将 CHandleMap::FromHandle()包装了起来，而 CHandleMap::FromHandle()才真正完成了查找，并将一个本地句柄与一个 C++对象相关联。CWnd::FromHandle()会返回一个适当类型的对象。例如，如果你有一个本地窗口句柄，而你想要得到相应的 CWnd 对象，你就只要创建一个 CWnd 对象，然后调用 FromHandle()即可。这样，在 Windows 中给定一个本地句柄类型时，调用 FromHandle()，你总是能得到相应的 C++对象。

关联窗口句柄与分离窗口句柄

为了完成窗口句柄与 C++对象之间的映射机制，CWnd 有一对函数来联系窗口句柄和 CWnd 派生对象。这两个函数是 Attach()和 Detach()。

CWnd::Attach()很直接。给定一个窗口句柄，Attach()会将 CWnd::m_hWnd 赋值为已有的窗口句柄，并将这对关系存放在 MFC 的永久窗口句柄映射表中。CWnd::Detach()则完成相反的工作。它将窗口句柄与 CWnd 派生对象之间的关系从窗口句柄映射表中删除，并将 CWnd::m_hWnd 的值设为 NULL。

使用 Attach()和 Detach()方法来设置成员变量 m_hWnd 更恰当。请记住句柄与 C++对象之间的关系是由系统管理的，所以你应该使用 MFC 的函数来在窗口句柄和 C++对象之间建立关系。

此时，对 CWinAPP 和 CWnd 的了解已经足够你理解剩下的部分——MFC 应用程序如何

启动和如何初始化。下面该进一步挖掘了。

现在，找到 WinMain()

CWinApp 和 CWnd 自身做的事情很少。这两个类之间只同工作就会得到一个能运作的 Windows 应用程序。现在，你会在一个示例程序（程序清单 2-2）中看到 CWnd 和 CWinApp，但你还要确定它是否满足了所有其他要求。你可能会问自己："究竟窗口在哪里注册的呢？消息循环从什么时候开始的呢？消息处理函数在哪里？"事实上，这些在程序里都有（否则它就不能运行）。这里就给出了 MFC 的内幕和它如何实现了 Windows 应用程序的基本元素。

在 MFC 程序里，去掉了 WinMain()函数。再看一下程序清单 2-2 中的 MFC 小程序。最明显的就是 MINIMAL.CPP 中没有了 WinMain()函数。因为 MFC 程序能正常运作，所以 WinMain()肯定在某个地方，它只是被埋在了 MFC 内部。

你可以在 APPMODUL.CPP 中找到 WinMain()函数。这里是 MFC 在你的 MFC 应用程序中包含的 WinMain()函数：

```
_tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine, int nCmdShow)
{
   return AfxWinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow);
}
```

从代码中可以看到，WinMain()将处理委托给一个叫做 AfxWinMain()的函数。如果再看 WINMAIN.CPP，你会看到 AfxWinMain()函数。它和一般的 WinMain()函数很相似，而且会看到它的行为和 WinMain()很像。程序清单 2-5 给出了 AfxWinMain。

程序清单 2-5 AfxWinMain 的伪代码（在 WINMAIN.CPP 中）

```
int AFXAPI AfxWinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine, int nCmdShow)
{
   ASSERT(hPrevInstance == NULL);

   int nReturnCode = -1;
   CWinApp* pApp = AfxGetApp();

   // AFX internal initialization
   if (!AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow))
      goto InitFailure;

   // App global initializations (rare)
   ASSERT_VALID(pApp);
   if (!pApp->InitApplication())
      goto InitFailure;
   ASSERT_VALID(pApp);

   // Perform specific initializations
   if (!pApp->InitInstance())
   {
```

```
        if (pApp->m_pMainWnd != NULL)
        {
           TRACE0("Warning: Destroying non-NULL m_pMainWnd\n");
           pApp->m_pMainWnd->DestroyWindow();
        }
        nReturnCode = pApp->ExitInstance();
        goto InitFailure;
    }
    ASSERT_VALID(pApp);

    nReturnCode = pApp->Run();
    ASSERT_VALID(pApp);

InitFailure:

    AfxWinTerm();
    return nReturnCode;
}
```

比较这个 WinMain 函数和其他的 WinMain()函数（例如 SDK 示例），你会发现这里没什么特殊的。参数还是那些，功能也是相同的。你会注意到在 WinMain 的前面有“_t”。“_t”表示支持 Unicode。下面在研究 WinMain()之前，我们先看看框架使用的一些重要结构。

MFC 状态信息

对于应用程序来说，在程序运行期间保存某些信息很有帮助。例如，大多数标准 C/SDK 程序会用全局变量保存实例的句柄。这样，应用程序用这个实例句柄可以获得关联到 EXE 文件的资源，而且这也是保存句柄的标准方法。MFC 也是这样做的，不过它使用各种结构来保存数据。

AFX_MODULE_STATE:没有文档说明的状态信息

与大多数 C/SDK 程序相比，除了实例句柄之外，MFC 应用程序还有很多其他状态信息。作为一个应用程序框架，MFC 要尽可能地一般化——它必须比一个用 C 和 SDK 写的一般程序能覆盖更广的内容。不过和其他 Windows 应用程序一样，MFC 也保存了各种项，包括主窗口句柄、资源和模块句柄。考虑一下 MFC 所提供的一切，包括内存分配跟踪、ODBC 支持、OLE 支持和异常处理。为了支持这些特性，MFC 必须比一般的窗口应用程序维护更多的信息。为了达到这个目的，MFC 定义了一个叫做 AFX_MODULE_STATE 的结构，它位于 AFXSTAT_.H 文件中，内容见程序清单 2-6。

程序清单 2-6　AFX_MODULE_STATE 的伪代码（在 AFXSTAT.H 中）

```
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

    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
    BYTE m_bSystem; // TRUE if module is a "system" module, FALSE if not
    BYTE m_bReserved[2]; // padding

    short m_fRegisteredClasses;// flags for registered window classes

    // 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

    // define process local/thread local portions of module state
    PROCESS_LOCAL(AFX_MODULE_PROCESS_STATE, m_process)

    THREAD_LOCAL(AFX_MODULE_THREAD_STATE, m_thread)
};
```

AFX_MODULE_STATE 包括很多关于模块的核心信息——在不考虑类型（EXE 或者 DLL）的情况下所有 MFC 模块所需要的信息。结构中有模块实例句柄、从中获得资源的模块实例、指向模块中 CWinApp 派生类的指针、应用程序的名字、指向应用程序的运行时类信息结构链表中的第一个节点的指针以及指向异常处理函数的指针。异常处理函数只有当编译器没有异常处理功能时才有用。总而言之，对于现代的编译器（例如 VC++ 2.x 及以上版本），它是没有用处的

这里给出了 AFX_MODULE_STATE 的一些重要的成员：

- m_pCurrentWinApp——一个指向 CWinApp 的指针。
- m_hCurrentInstanceHandle——该模块的实例句柄。

- m_hCurrentResourceHandle——代表保存了模块资源的实例句柄。
- m_lpszCurrentAppName——指向应用程序名字的指针。
- m_bDLL——表示模块是一个动态链接库还是一个可执行文件。
- m_classList——指向应用程序的 CRuntimeClass 结构链表中的第一个运行时类的指针。
- m_factoryList——指向应用程序的 COleObjectFactory 结构链表中的第一个运行时类的指针。
- m_nObjectCount——OLE 服务器的引用计数，表示服务器是否有未完成的 COM 对象。
- m_bUserCtrl——表示用户是否在使用 OLE 服务器的标志。
- m_szUnregisterList[4096]——维护一个注册了的窗口类的链表，以便 MFC 在这些类结束时可以将它们注销。
- m_pfnAfxWndProc——指向 MFC 的标准窗口过程。
- m_fRegisteredClasses——表示哪些 MFC 窗口类已经注册了。

开发人员提示

再仔细地看一下 AFX_MODULE_STATE，你会发现这个结构里有两个实例句柄：一个是应用程序实例句柄，另一个是资源实例句柄，当你想将自己的应用程序国际化时，这一点是非常有用的。当你想应用一个用户接口资源（例如一个对话框模板或一个字符串表）时，MFC 就会使用模块的资源实例（AFX_MODULE_STATE::m_hCurrentResourceHandle）来获得资源。通常情况下，资源句柄都指向应用程序的可执行文件。但你不能使用实例句柄（可能是一个 DLL 的句柄）来获得为本地定制的资源。

你还会发现 AFX_MODULE_STATE 包含一个指向 CRuntimeClass 结构链表的第一个节点的指针。这个链表在模块启动时就被初始化了，并且在有新的类从 CObject 派生时，它就会增加。如果你需要为你的应用程序枚举运行时类信息，可以从这里开始。当然使用时要小心（这个结构在 AFXSTAT_H 中）。这些东西在将来可能都会被修改。虽然保存这些状态结构对调试很有帮助，但是还是不要依靠这些结构来写代码。

你会看到 MFC 保存的信息不仅仅是一个实例句柄。既然你已经知道了这些结构，那么如果你想访问这些关于应用程序状态信息中的某些位就知道到哪里去找了。AFX_MODULE_STATE 中尤其有用的是资源实例句柄和运行时类信息链表。AFX_MODULE_STATE 中大部分成员都可以通过类似 AfxGetInstanceHandle()这样的文档函数来访问。现在，请记住这些结构中的数据，我们在后面还会用到这些成员。

当然，还需要注意一点：这些都是没有文档说明的结构。它们是可能改变的。例如，MFC 1.5 的 CRuntimeClass 结构中包含了指向应用程序的运行时类链表的指针，而你在程序清单 2-6 中会看到这个指针已被移到了 AFX_MODULE_STATE 结构中。你可以使用这些结构，但是要谨慎——尤其是在给某些域赋值时。

版本注释

MFC 3.0 将这些信息分成了独立的结构：AFX_CORE_STATE、AFX_WIN_STATE 和 AFX_OLE_STATE。MFC 3.0 也有一个叫做 AFX_MODULE_STATE 的结构，它只是将 3 个结构组合在一起。

现在你知道了 MFC 中保存的状态信息，让我们继续来比较 MFC 写的程序和 C/SDK 写的程序吧。

回到 WinMain()

理解了 MFC 用来保存模块状态信息的结构，AfxWinMain()就变得容易了些。下面我们再来看一个小的 MFC 程序的运行情况。AfxWinMain()首先声明一个类型为 CWinApp 的指针变量，然后将它赋值为 AfxGetApp()。AfxGetApp()是自描述的：它将返回与程序相关的应用程序对象（每个 MFC 程序都需要一个 CWinApp 的派生对象）。但是，如果 WinMain()是 Windows 应用程序的入口点，而应用程序又不是在 WinMain()中创建的，那么框架应该如何得到应用程序对象呢？

答案在于 C++对构建全局对象的处理方式。C++程序在做任何事之前（甚至是在 main()函数或 WinMain()函数调用之前），首先构建全局对象。只要你在程序中有一个全局的 CWinApp 派生对象，就可以确保 WinMain()执行时全局变量已经存在。

构造 CWinApp

CWinApp 的构造函数代码在 APPCORE.CPP 中。CWinApp 的构造函数的主要工作就是初始化 CWinApp 的成员变量。CWinApp 的构造函数只有一个参数：程序的名字。CWinApp 会将 m_pszAppName 变量设置为传入的值。这个参数默认情况下为 NULL——如果你想提供应用程序的名字也可以。然后 CWinApp 的构造函数会初始化模块的线程状态结构和模块状态结构（我们会在第 10 章再讨论 AFX_THREAD_STATE 结构）。CWinApp 的构造函数将 AFX_MODULE_STATE 的 m_pCurrentWinApp 初始化为正在构造的 CWinApp。CWinApp 的所有其他成员（句柄、指针和字符串）都被置为 NULL（也就是实例句柄、指向主窗口的指针和应用程序的名字等等）。

初始化框架：AfxWinInit()

下一步，AfxWinMain()会调用 AfxWinInit()来初始化框架。AfxWinInit()和 WinMain()所带的 4 个参数一样：当前实例句柄、前一个实例句柄、命令行参数和显示命令。

AfxWinInit()调用 SetErrorMode()来为应用程序设置错误模式。这用来指明什么会导致应用程序失败。SetErrorMode()带有一个位图标志。MFC 使用 SEM_FAILCRITICALERRORS 和 SEM_NOOPENFILEERRORBOX 标志设置错误模式。MFC 还使用 SEM_ FAILCRITICALERRORS 来通知窗口对于关键错误不要显示关键错误处理函数（critical-error-handler）消息框，而只是将错误传回给调用进程。SEM_NOOPENFILEERRORBOX 标志则告诉窗口在没有找到文件时不要显示消息框，而将错误返回给调用进程。

然后 AfxWinInit()会调用 AfxGetModuleState()来得到模块的 AFX_MODULE_STATE 结构。AfxWinInit()将模块的实例句柄和资源句柄存放在 AFX_MODULE_STATE::m_hCurrentInstanceHandle 和 AFX_MODULE_STATE::m_hCurrentResourceHandle 中。此时，这两个句柄都指向模块的实例句柄。

每当 Windows 应用程序启动时，Windows 都会传给应用程序 4 个参数：（1）当前模块实例的句柄；（2）前一个模块实例的句柄；（3）任何命令行参数；（4）显示窗口标志。CWinApp 将这些参数拷贝到成员变量里。AfxWinInit()会设置 CWinApp::m_hInstance、CWinApp::m_

hPrevInstance、CWinApp::lpCmdLine 和 CWinApp::nCmdShow 的值。然后 AfxWinApp()会调用应用程序对象的 SetCurrentHandles()函数，而再次设置 AFX_MODULE_STATE 的句柄。这个多余步骤有历史性的原因。过去没有模块状态结构——它是在解决 OLE 控件时引入的。也许以后 CWinApp 中的模块状态结构也会消失，这些都依赖于向后兼容问题。

下一步，CWinApp::SetCurrentHandles()会用 CWinApp 的内容初始化应用程序名字和路径变量：首先它调用 GetModuleFileName()来获得模块文件名。GetModuleFileName()是一个 Win32 API 函数，它会返回模块的运行实例的全名。SetCurrentHandles()初始化 CWinApp::m_pszExeName，将它设置为可执行文件的名字（不带 EXE 扩展名）。例如你写了一个名为“FILEVIEW.EXE”的 MFC 应用程序，CWinApp::m_pszExeName 的值就是“FILEVIEW”。SetCurrentHandles()然后将 m_pszAppName 初始化为应用程序的标题。如果你使用了 AppWizard 来创建工程，MFC 会使用你工程的资源文件中特定的字符串来代替 ID AFX_IDS_APP_ TITLE。如果找不到资源，m_pszAppName 就会被初始化为 m_pszExeName 的值。SetCurrentHandles()还会把应用程序的 AFX_MODULE_STATE::m_lpszCurrentAppName 的值设置为 m_pszAppName 的值。

SetCurrentHandles()还会填写 CWinApp 的帮助文件和 profile 字符串：m_pszHelpFilePath 和 m_pszProfileName。SetCurrentHandles()将 m_pszHelpFilePath 初始化为 GetModuleFileName()返回的值，但会使用 HLP 作为扩展名，它还将 m_pszProfileName 初始化为应用程序的名字，扩展名为 INI。

如果 AfxWinInit()完成了上面的所有工作，应用程序和框架就完全初始化了。

句柄和文件名都正确地初始化了，MFC 就继续初始化应用程序的其他部分。AfxWinMain()会调用应用程序的 InitApplication()函数。

InitApplication()

InitApplication()的工作是执行那些针对整个应用程序的初始化。CWinApp 的 InitApplication 版本初始化应用程序文档管理器（你会在第 7 章看到更多关于文档/视图结构的内容）。当 Windows 升级为 32 位后，InitApplication()就基本上没用了。所有的初始化都在 InitInstance()中完成了。InitApplication()只在 16 位的 Windows 中有用。在 16 位的 Windows 中，应用程序的第一个实例和后面的实例不同（你可以通过检查 hPrevInstance 发现）。Win32 就不区别第一个实例和后面的实例。

AfxWinMain()然后调用 InitInstance()初始化应用程序的特定实例。

InitInstance()

程序启动后，为程序的特定实例进行初始化很有必要。CWnd::InitInstance()就是负责完成这一任务的。CWinApp 的默认 InitInstance()实现不做任何事情，它只是返回 TRUE。但 InitInstance()是虚函数，所以你可以放心地覆盖它。InitInstance()内部的代码应该包含这样的任务：为应用程序设置所有的文档和显示主窗口。因为 InitInstance()的默认版本不做任何事，所以确保窗口显示在屏幕上都是你的事。

准备好消息泵：CWinApp::Run()

WinMain()所做的最后一件事就是调用 CWinApp 派生类的 Run()函数。Run()函数会启动

消息循环。我们立即会看到 Run()函数会比一般的 GetMessage()..DispatchMessage()循环完成更多的任务。

一些其他隐藏的信息

在检查消息泵之前，我们先考虑两个问题。其中比较明显的是窗口类：所有的 Windows 应用程序都要注册至少一个窗口类。不那么明显的问题是 Windows hook 和 MFC 窗口过程是如何引入的。现在我们先看 MFC 的窗口类（这是窗口类，它和 C++类有所不同）。

注册窗口类

在 Windows 应用程序显示窗口之前，应用程序必须先在操作系统那里注册至少一个窗口类。MFC 应用程序和其他 Windows 应用程序一样，所以它也需要注册至少一个窗口类。窗口类还定义了窗口的一些基本内容，例如它的显示（通过一些标志）以及它的行为（通过回调函数）。MFC 实际上注册了 4 个标准窗口类，这 4 个窗口类分别为：（1）一般的子窗口；（2）一个控制栏窗口；（3）一个 MDI 框架窗口；（4）一个 SDI 窗口或 MDI 的子窗口。它们的名字都带有 MFC 的版本号。而且，MFC 还用关于应用程序是否静态链接和应用程序是调试版本还是发布版本的信息来修饰类名。下面这 4 个修饰后的窗口类名字用于非静态的调试版本的应用程序：

- AfxFrameOrView40d。
- AfxControlBar40d。
- AfxWnd40d。
- AfxMDIFrame40d。

MFC 还为一般控件准备了一个窗口类，我们会在第 6 章看到。

MFC 注册窗口的方法很有趣。它在 AFXIMPL.H 中定义一个叫做 AfxDeferRegisterClass()的宏：

```
#define AfxDeferRegisterClass(fClass) \
   ((afxRegisteredClasses & fClass) ? TRUE :
AfxEndDeferRegisterClass(fClass))
```

AfxDeferRegisterClass 首先查找是否已经注册了一个窗口类。全局变量 afxRegisteredClasses 保存表示 MFC 窗口类的位图。下面的值表示 MFC 的窗口类：

```
AFX_WND_REG              (0x0001)
AFX_WNDCONTROLBAR_REG    (0x0002)
AFX_WNDMDIFRAME_REG      (0x0004)
AFX_WNDFRAMEORVIEW_REG   (0x0008)
```

MFC 使用全局变量 afxRegisteredClasses 来优化窗口注册。这一点很重要，因为 MFC 试图在多个地点注册窗口类。AfxDeferRegisterClass()首先检测前一个注册类的指示器（indicator），判断是否需要注册窗口类。如果窗口已经注册，那么 AfxDeferRegisterClass()只返回 TRUE。否则它会注册刚刚用位图表示的窗口类。

版本注释

MFC 3.0 在运行 AfxWinInit()时注册所有的窗口类。

AfxEndDeferRegisterClass()

AfxEndDeferRegisterClass()会调用 memset 来将 WNDCLASS 结构清空，从而除了那些显式设置的域之外，其他域都是 NULL 或零。AfxEndDeferRegisterClass()先初始化 WNDCLASS::lpfnWndProc，将它指向 DefWindowProc()，就是那个位于消息循环末尾处理与你的应用程序无关的消息的 DefWindowProc()。虽然看起来它会阻止你的应用程序接收消息，而实际上 MFC 有办法将消息直接传递给你的应用程序，关于这一点你会在第 3 章找到有关内容。窗口类的句柄 hInstance 是当前的实例句柄。AfxDeferRegisterClass()将窗口类的光标设置为一般的箭头光标。这些值对于所有的 MFC 窗口类来说都是一样的。

一旦类结构被清空，域也被初始化之后，AfxEndDeferRegisterClass()会开始填充域，这些值由注册的窗口类决定。如果 AfxEndDeferRegisterClass()尝试注册一个一般的 plain-vanilla 窗口，窗口就叫做“AfxWndXXXX”（用恰当的类名来修饰它）。它会使用类风格的最小集：CS_DBLCLKS、CS_HREDRAW 和 CS_VREDRAW。所有其他的域都是 NULL 或零。这个窗口类可以用于所有用 CWnd::Create()创建的 CWnd 对象。

如果调用者要注册一个工具栏，AfxEndDeferRegisterClass 会注册一个名为“AfxControlBarXXXX”的类。MFC 会将类风格对应的位关掉（风格位被设为零）。也就是说 CS_HREDRAW 和 CS_VREDRAW 都被关掉了，因为控制栏（工具栏和状态栏）在接收到 WM_SIZE 消息时不需要重画整个窗口。这也减少了在重画窗口时带来的闪烁。而且控制栏也不用处理双击事件。最后，窗口的背景画笔被设置为按钮表面的颜色。

如果调用者要注册一个 MDI 框架窗口类，那么窗口类的名字会是“AfxMDIFrameXXXX”。AfxEndDeferRegisterClass()会打开双击标志（CS_DBLCLKS），并将背景画笔设置为 NULL。注册窗口类使用的默认 MDI 框架图标是由 AppWizard 提供的，图标的标识符是 AFX_IDI_MDI_FRAME。如果你要修改 MDI 框架窗口的图标，就要修改 AFX_IDI_STD_ MDIFRAME 图标。

最后，SDI 框架窗口或 MDI 子窗口或视图的窗口类注册时的名字是“AfxFrameOrView XXXX”。它的风格是 CS_DBLCLKS、CS_HREDRAW 和 CS_VREDRAW。背景画笔的颜色是窗口的默认颜色。窗口类会用默认的 SDI 框架图标，叫做 AFX_IDI_STD_FRAME。和 MDI 框架窗口类一样，如果你想修改框架图标，需要在应用程序中修改 AFX_IDI_STD_FRAME 图标。

这些窗口类是在什么时候注册的呢？MFC 早期的版本在启动时注册这些类。下面的链表显示了哪些函数负责注册窗口类和它们都注册了哪些类：

- CDialogBar::Create()——注册 AfxControlBar 类。
- CDockBar::Create()——注册 AfxControlBar 类。
- COleResizeBar::Create()——注册 AfxControlBar 类。
- CView::PreCreateWindow()——注册 AfxFrameOrView 类。
- CWnd::Create()——注册 AfxWnd 类。
- CFrameWnd::PreCreateWindow()——注册 AfxFrameOrView 类。

- CFrameWnd::LoadFrame()——注册 AfxFrameOrView 类。
- CMDIFrameWindow::PreCreateWindow()——注册 AfxMDIFrame 类。
- CSplitterWnd::CreateCommon()——注册 AfxMDIFrame 类。

微软决定在窗口创建期间注册这些窗口类，而不是在应用程序的开始注册这些窗口类，是因为 Wnd 类注册需要时间。MFC 避免了注册那些没有必要的窗口类。在一些情况下只有一小部分类是有用的（例如对于一个 OLE 控件或没有用户界面的简单 DLL）。这样就优化了启动过程。

MFC 的窗口 hook

在应用程序的初始化过程中，AfxWinInit()完成了一个很有趣的部分：它安装了两个 hook。第一个 hook 是一个消息过滤器 hook（在调用 SetWindowsHooEx()时由 WH_MSGFILTER 标志来标记）。消息过滤器 hook 监听那些在对话框、消息框、菜单或滚动条中作为输入事件的结果而生成的消息。MFC 安装_AfxMsgFilterHook()函数作为消息 hook。在下一章当我们讨论命令和消息路由选择结构时，使用这个 hook 的理由就显示出来了。

第二个 hook 是支持基于计算机训练的应用程序。MFC 使用 WH_CBT 标志调用 SetWindowsHookEx()来设置这个 hook。

MFC 不特别支持基于计算机的训练。MFC 安装了 hook，从而就可以在窗口被创建时立刻进入。MFC 不能等到 CreateWindowEx()返回时再进入，因为那时已经有一些消息被发送出来了。CWnd 对象必须在任何消息被送给窗口之前让 MFC 进入。WH_CBT hook 就完成了这个工作（虽然这不是它原始的目的）。

很显然，MFC 执行这一步是为了确保应用程序在 Windows 下运行。它提供了 WinMain()入口点，提供了一个消息处理函数，而且还注册了一些窗口类。MFC 应用程序惟一要做的事就是启动消息泵。这些在 CWinApp::Run()中完成。

MFC 的消息泵：CWinApp::Run()

MFC 3.0（及以后版本）提供的一个主要特性就是安全的用户的接口线程（UI 线程）（在过去，你不能在用户接口代码内部使用线程）（更多的细节参考第 10 章）。为了获得线程安全性，微软定义了两种不同的线程：（1）标准的 Win32 线程（也叫做辅助线程）；（2）MFC 的 UI 线程。这两种线程都是由 CWinThread 类封装的。辅助线程与 UI 线程的主要区别就是辅助线程是抢先式的。它们是由 CreateThread() API 函数创建的 Win32 线程。另一方面，UI 线程只是消息泵（也就是一般的 Windows GetMessage()..DispatchMessage()循环）。因为 CWinApp 是从 CWinThread 派生出来的，所以 CWinApp 也从 CWinThread 那里继承了一个消息泵。你会在 CWinApp 的 Run()函数中找到实现了的消息泵。

回忆一下，WinMain()的最后一步就是调用应用程序的 Run()函数。此时，CWinApp 的派生类只是使用了 CWinThread 的 Run()函数来启动消息泵。如果你想找到 CWinThread::Run()是如何工作的，请跳到第 10 章。现在你所需要知道的就是处理消息的方式和其他 Windows 程序一样：使用 GetMessage()、TranslateMessage()和 DispatchMessage()。然而，MFC 处理消息路由选择的方式有所不同。它没有使用庞大的 switch 语句来过滤消息，而是使用了消息映射表，我们会在下一章讨论消息映射表。

从现在开始，我们来看 Windows 编程的另一个基本概念：GDI（Graphics Device Interface，图形设备接口）。

MFC 对 GDI 的支持

对于开发人员来说，使用 Windows 写应用程序的一个很重要的理由就是它支持图形。Windows 总是带有大量的图形工具，再加上 Win32 的 bezier 和路径函数，它对图形的支持越来越好。

Windows 最初的设计目标之一就是保证设备独立性。想法就是在简单的 API 后面隐藏图形代码。这样使开发人员只需要使用一些图形函数，而不用考虑实际的代码。例如，开发者应该能在屏幕、绘图仪或打印机上用相同的函数绘画。这个 API 就是 GDI。

在 GDI 后隐藏的基本概念就是操作系统为在不同的设备上绘画提供了一系列的函数和数据结构。在最高层次上使用 GDI 就是应用程序本身了。应用程序使用 GDI 函数来在一些设备（通常是屏幕和打印机）上绘画。然而，Windows 并不将绘画命令直接传递给设备，而是传递给设备驱动器——一个位于 Windows 和物理设备之间的软件。当设备驱动器接收到绘画命令时，它在物理设备上执行所需要执行的代码。然而，用 GDI 编程不是件简单的事。事实上，它很麻烦。MFC 的追求目标就是帮助隐藏 GDI 的细节和一些矛盾，从而简化 GDI 编程。

设备环境

GDI 的核心就是设备环境（device context）。在 Windows 下产生输出，其背后隐藏的思想就是向标准的 API（图形设备接口函数）中写内容，然后由 Windows 处理其他事情。Windows 图形 API 都在一个叫做设备环境的结构里。在 Windows 产生输出时，是直接输出到设备环境中。设备环境可能表示一个屏幕（通常情况下都是屏幕），也可能是其他设备，例如打印机。从编程的角度讲，向屏幕设备环境输出和向打印机设备环境输出所使用的函数是一样的。

软件模式：Windows GDI 和 facade 模式

软件开发领域中最令人激动的思想就是软件模式，在《Design Patterns: Elements of Reusable Object-Oriented Software》（Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 编著，（Addison- Wesley，1995）一书中已经有详细讨论。这本书由两部分组成：（1）关于软件模式是什么的描述，以及它们是如何起作用的；（2）23 种软件设计模式的目录。

简而言之，软件模式就是已成功设计的规范化的描述。软件设计被认为是“第一原则”。例如，一些有经验的软件设计者尝试将他们的代码隐藏在一个坚固的、定义良好的接口后面。使用这种方法，他们可以修改实现或者写类似的能插入的组件。这也就是 Windows GDI 后面隐藏的观点。有了 GDI，你就可以使用一系列函数来对任何数量的媒质[屏幕、打印机、元文件（metafile）等等]进行写操作。Windows 然后会将处理过程委托给设备驱动器，再由设备驱动器和物理设备打交道。事实上，Windows GDI 后面的概念就是《Design Patterns》一书中描述的“facade”模式。记得要查看一下。

设备环境定义了一系列图形对象和与它们相关的属性[例如画笔（pen）、画刷（brush）和

位图（bitmap）]以及影响输出的图形模式（像背景模式和映射模式）。

当然，MFC 还有一些表示设备环境的类。MFC 定义了几种不同的设备环境：一个用于绘画（CPaintDC），一个用来表示窗口的整个屏幕区域（CWindowDC），一个用来表示窗口中的客户区域（CClientDC），一个用来表示特殊的元文件（CMetaFileDC）。

MFC 对设备环境的支持和它对一般窗口句柄的支持很类似。也就是说，MFC 设备环境类（CDC、CClientDC、CPaintDC、CWindowDC 和 CMetaFileDC）都包装有一个 Windows 设备环境句柄（一个 HDC）。和与窗口句柄相关的 API 函数一样，每当你看到一个 API 函数的第一个参数是 HDC 时，这个 API 函数很可能就是设备环境类的成员。

MFC 还保存了一个 HDC 与它们对应的 CDC 驱动对象之间的映射表。这个表是 AFX_MODULE_THREAD_STATE 的一个成员。CDC 还有一个成员函数叫 FromHandle()，它返回与某个特定的 HDC 相匹配的 C++对象。

CDC 类内部

正如你所猜测的，CDC 类中有一个类型为 HDC 的成员变量，它表示一个实际设备环境的句柄。然而，CDC 还有另一个 HDC，即 m_hAttribDC。这个设备环境和 m_hDC 一样。向设备环境请求信息的 CDC 函数通常使用 m_hAttribDC。

CDC 还有一个 BOOL 类型的成员变量，表示是否输出到打印机。

CDC 的构造函数很简单：它只是将成员变量（m_hDC、m_hAttribDC 和 m_bPrinting）初始化为 NULL（或者将 m_bPrinting 初始化为 FALSE）。Attach()将 CDC::m_hDC 设置为特定的 HDC，而且在映射表中增加 HDC/C++对象关系。CDC::Detach()从映射表中删除关系。

除了包装一些与设备环境有关的 API 函数以外，CDC 还有一些与 HDC 无关的函数。例如，用于打印的 Windows API 函数也是 CDC 的成员函数。这些函数包括 StartDoc()、EndDoc()和 Escape()。你会在第 8 章的文档/视图结构中看到更多的打印函数。

最后，CDC 的析构函数调用 DeleteDC()。因为设备环境的数量有限，所以在你使用完后删除 DC 很重要。使用 MFC 设备环境类后要注意在完成时删除设备环境。

CDC 的派生类

MFC 从 CDC 派生了几个特殊的设备环境：CPaintDC、CWindowDC、CClientDC 和 CMetaFileDC。

- CPaintDC——在需要绘画时要使用这个类。它和调用 BeginPaint()得到的设备环境一样。CPaintDC 的构造函数需要一个指向 CWnd 的指针，它会调用 BeginPaint()而创建一个设备环境，然后将它写入设备环境映射表。CPaintDC 的析构函数是 EndPaint()并且从映射表中删除设备环境句柄。
- CWindowDC——这个结构表示窗口内的整个屏幕区域（包括用户区域和框）。CWindowDC 的构造函数需要一个指向 CWnd 的指针，它调用 GetWindowDC()来为整个窗口创建一个设备环境。CWindowDC 的析构函数是 ReleaseDC()。
- CClientDC——这个设备环境表示窗口中的用户区域。它的构造函数也需要一个指向 CWnd 的指针。CClientDC 调用 GetClientDC()来为窗口中的用户区域创建一个设备环境。CClientDC 的析构函数是 ReleaseDC()。
- CMetaFileDC——Windows 的元文件有一个 GDI 命令序列，利用这个命令序列可以

重新画出图像。MFC 定义了一个类——CMetaFileDC，用它来包装元文件设备环境。创建一个元文件 DC 需要两步：首先，构造 CMetaFileDC（在栈中分配一个或声明一个），然后调用 CMetaFile::Create()来创建一个元文件并初始化 CMetaFile 的成员变量。CMetaFileDC 的析构函数是 DeleteMetaFile()。

设备环境还只是一半。要想使用图形，还需要指定要使用哪种工具来绘画。这些工具在 MFC 的 GDI 对象类中有所描述。

图形对象

设备环境保存了在 Windows 中绘画所需要的所有信息，包括映射模式等等。GDI 的另一个问题就是设备环境使用的图形对象。设备环境还保存了关于在设备上绘画所使用工具的信息。这些绘画工具包括字体、画笔、画刷。

MFC 对 GDI 对象的支持包括一个名为 CGDIObject 的基类和几个表示绘画工具的类，例如 CPen、CBrush、CFont、CBitmap、CPalette 和 CRgn。

CGDIObject

这个类代表了所有 GDI 对象的基类。MFC 对 GDI 对象的支持和对窗口句柄的支持很类似。GDI 对象也是通过句柄定义的，MFC 维护了一个 GDI 对象和表示 GDI 对象的 C++对象之间的映射表。CGDIObject 还有管理和使用这些映射表的成员函数。CGDIObject 有 FromHandle()函数、Attach()函数和 Detach()函数。

CGDIObject 有一个叫 m_hObject 的成员变量，它就是 GDI 对象的句柄。该句柄属于画刷、画笔、字体、位图、调色板和区域。

和窗口句柄及设备环境句柄一样，MFC 还保存了 GDI 对象句柄与包装它的 C++对象之间的关系。你还会在 CGDIObject 中找到管理这种关系和获得句柄的函数。幸运的是，这些函数的名字都是一样的。也就是说，如果你想得到包装了特定 GDI 句柄的 GDI 对象，只要调用 FromHandle()函数即可。GDI 对象/句柄映射表保存在前面提到的 AFX_MODULE_STATE 结构中。

在使用 GDI 对象的过程中，你有多少次忘记删除它们了呢？泄漏资源是不对的。即使现在你有一个很大的资源池用来分配 GDI 对象，你也应该在使用之后释放它们。这包括当你使用完 GDI 对象时释放它们。幸运的是，CGDIObject 的析构函数会调用 DeleteObject()。

MFC 从 CGDIObject 中派生了 6 个 GDI 对象：CPen、CBrush、CFont、CBitmap、CPalette 和 CRgn。

- CPen——包装了 Windows 画笔对象（一个 HPEN），还包含了那些创建画笔的 API 函数作为成员函数。
- CBrush——包装了 Windows 画刷对象（一个 HBRUSH）和创建画刷的 API 函数。
- CFont——包装了 Windows 字体对象（一个 HFONT）以及创建和管理字体的 API 函数。
- CBitmap——包装了 Windows 位图对象（一个 HBITMAP）以及创建、装载和管理位图的 API 函数。
- CPalette——包装了 Windows 调色板对象（一个 HPALETTE）以及创建并使用调色

板的函数。

- CRgn——包装了 Windows 区域对象（一个 HRGN）和使用区域的函数。

要使用这些 GDI 对象，只要将它们选入活动的设备环境当中，然后就可以使用了。例如，如果你想在窗口中的用户区域画蓝色的线，可以这样使用 OnPaint()：

```
CMyWnd::OnPaint() {
      CPaintDC paintDC(this);
      CPen* pOldPen;
      CPen bluePen(PS_SOLID, 25, RGB(0, 0, 255));

      pOldPen = paintDC.SelectObject(&bluePen);
      paintDC.MoveTo(1, 1);
      paintDC.LineTo(100, 100);
      paintDC.SelectObject(pOldPen);
}
```

结 语

现在你已经掌握了 MFC 对基本 Windows 应用程序的支持。所有的 Windows 程序如果要正确运行都要执行许多相同的操作。而且，在不考虑应用程序是用什么开发的情况下，这些步骤都是相同的。MFC 处理了所有的样本代码：80 行的 C/SDK 代码在 C++和 MFC 代码中只要 20 行。MFC 在 CWinApp 和 CWnd 类中处理了 Windows 应用程序的两个基本部分。

然而，MFC 所做的不仅仅是隐藏样本代码。MFC 是一个成熟的应用程序框架，它的众多优点可以让软件开发人员的工作变得更容易。下一章主要讲 MFC 的消息映射表结构。在后面的章节，我们会看到 MFC 的实用类（第 4 章）、CObject 类（第 5 章）、MFC 对对话框和控件的支持（第 6 章）、MFC 的文档/视图结构（第 7 章和第 8 章）和 MFC 对用户接口的支持。

CHAPTER 3

MFC 中的消息处理

现在，我们已经了解了 Windows 程序的第一个主要部分：主应用程序自身。这部分主要完成程序的各种初始化工作：显示窗口、启动消息泵。我们下面还要学习第二部分：实际的消息处理。

在一般的 C/SDK 程序中很容易看出消息是如何处理的，因为处理过程都在窗口过程的大 switch 语句里。但在 MFC 程序里你却找不到这样的 switch 语句。那么 MFC 又是如何处理消息的呢？应用程序初始化之后，每个窗口类都会调用相同的回调函数——DefWindowProc()。为什么一个应用程序中所有的窗口都使用同一个回调函数呢？而且 DefWindowProc()并没有做什么——它的主要用途就是处理那些没人需要的消息。为什么这个过程要和 MFC 窗口类一起注册呢？这些问题的答案会随着我们进一步研究 MFC 消息处理的过程中得到。

CCmdTarget 和消息映射表

MFC 的消息处理结构由两个基本部分组成：CCmdTarget 类和消息映射表。这两个部分协作来完成和一般 C/SDK 应用程序一样的消息处理能力。一旦你理解了消息映射表，你也就掌握了 MFC 实现 COM（Component Object Model，组件对象模型）的方式、自动化以及 OLE 控件事件，因为它们都使用了类似的技术。

在 MFC 的第一个版本和 Microsoft C 版本 7 一起推出的时候，消息映射表就是 MFC 的一个特性。消息映射表把消息和处理消息的代码关联在一起。它替代了用 C 和 SDK 开发的 Windows 程序中的那个大的 switch 语句。

理解 MFC 的消息映射表机制很重要，因为：

1. 它可以满足你对 MFC 内幕的好奇心。

2. 你会知道为什么一个 MFC 程序是那样的（例如，你会知道为什么要在 MFC 类的头文件上写 DECLARE_MESSAGE_MAP，并在调用的实现中写 BEGIN_MESSAGE_MAP/END_MESSAGE_MAP）。

3. 你可以更好地理解消息映射表的语法。

4. 你对 MFC 的整体把握会更好。

我们把 MFC 的消息映射表分成几块，以便你能理解以下几点：

1. 消息映射表是如何替代 C/SDK 中的 switch 语句的。

2. 消息映射表由哪些结构组成。
3. 消息映射表如何工作。
4. 消息在应用程序框架中如何流动。

为了完全理解消息映射表和它们的开发原理，知道窗口消息本身和 Windows 消息处理如何演化就很重要。我们首先会看窗口消息本身的结构以及在 C/SDK 应用程序中如何处理窗口消息。然后，我们会看到为什么 MFC 用消息映射表处理窗口消息而不是用虚函数（virtual function）。最后我们把消息映射表分成几块，并且实际地在框架中跟踪一条命令消息（菜单项或其他用户界面对象产生的）和一个一般窗口消息。看看消息映射表如何工作。

窗口消息

Windows 是一个事件驱动的操作环境：它在计算机上观察硬件，看是否有事情发生，一旦发生了一些有趣的事，例如鼠标移动了，Windows 就会检测到它，并将信息传递给对消息感兴趣的应用程序。这个关于事件的信息就是一条消息。在上面举的例子中就是 WM_MOUSEMOVE 消息。如果这个消息对屏幕上某个窗口来说很重要，窗口就会处理这个消息。例如，如果 Windows 检测到屏幕上某个窗口的菜单项被选中了，它会把这个信息传递给窗口的消息处理函数。窗口的消息处理函数会处理菜单选项事件的。

所有窗口消息的基本形式都是一样的。它有 3 个部分：（1）一个无符号的整数，它包含了消息的实际内容；（2）WPARAM——一个 4 字节的参数（WPARAM 在 Win32 下是 32 位的）；（3）LPARAM——一个 4 字节的参数。微软这样定义消息是因为 Windows 不知道每条消息的特殊情况。也就是说每条消息必须有多方面的行为。换句话说就是窗口消息有一个具有多种不同行为的接口。

在窗口消息的 3 个组成部分中，无符号整数是实际消息。WPARAM 和 LPARAM 可以意味着很多事。事实上，LPARAM 常常包含了额外数据或指向某个处理消息所需要的数据结构的指针。

例如像 WM_COMMAND 这样的典型消息。当某个菜单项被选中、控件向它的父窗口发送通知（notification）代码或翻译快捷键时，窗口都会收到 WM_COMMAND 消息。参数 WPARAM 的低 16 位表示一个能够标志控件的号码。高 16 位是通知代码，表示像用户是否双击了控件这样的事件。LPARAM 表示发送消息的控件的窗口句柄。

这些就是 WM_COMMAND 消息的内容。其他消息的 WPARAM 和 LPARAM 参数中有不同的信息。显然要记录消息和参数的意义很难。

Windows 每纳秒产生一条消息。这样程序就需要有办法过滤消息，以便它们能做出正确的反应。在使用 SDK 时，这些都是通过使用一个带有 switch 语句（用来解释消息）的窗口过程来完成的。

使用 C 和 SDK 进行消息处理

任何 Windows 程序（甚至是基于 MFC 的程序）的核心都是消息泵。消息泵就是一个循环，它取出消息，并将消息送给恰当的窗口消息处理函数。下面是一个典型的消息泵：

```
while (GetMessage(&msg, NULL, NULL, NULL)) {
   TranslateMessage(&msg);/* Translates virtual key codes       */
   DispatchMessage(&msg); /* Dispatches message to window       */
}
return (msg.wParam);       /* Returns the value from PostQuitMessage */
```

一个 C/SDK 程序开始运行时，它会立即调用 RegisterClass() API 函数并使用一个 WNDCLASS 结构来注册至少一个窗口类。WNDCLASS 结构中包含了窗口的图标和背景画刷，还包含了一个处理窗口消息的回调函数。每当该类的一个窗口有消息时，Windows 都会调用 WNDCLASS 结构的 lpfnWndProc 域所指定的函数。

作为一个典型的 C/SDK 程序，消息处理函数往往是一个 switch 语句，它将窗口感兴趣的消息过滤出来。例如，如果你的应用程序是个绘画程序，你就会对鼠标移动事件感兴趣，如果你的窗口发现鼠标在移动，或者鼠标左键被按下，那么你的应用程序可能就会解释成“用户正在试图画一条线——在该处画一个点”。

典型的 C/SDK 程序中的 switch 语句很繁琐，通常就像程序清单 3-1 所示。

程序清单 3-1　一个典型的 C/SDK 窗口过程

```
LRESULT EXPORT WINAPI WndProc(HWND hwnd,
                              UINT message,
                              WPARAM wParam,
                              LPARAM lParam) {
   LONG  lRet  = 0L;

   switch (message) {
      case WM_CREATE:
         HandleCreate(hWnd, wParam, lParam);
         break;

      case WM_COMMAND: {
         switch (wParam) {
            case IDM_ABOUT:
               HandleAbout(hWnd);
               break;

            case IDM_EXIT:
               DestroyWindow(hwnd);
               break;

            default:
               break;
            }
         break;
         }
      case WM_PAINT: {
         PAINTSTRUCT ps;
         HDC         hdc;

         hdc = BeginPaint(hwnd, &ps);
         EndPaint(hwnd, &ps);
         }
```

```
            break;

        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        default:
            lRet = DefWindowProc(hwnd, message, wParam, lParam);
            break;

      return lRet;
   }
}
```

你会发现 switch 语句是分层次的。例如，一旦你发现有 WM_COMMAND 消息，就需要检查 wParam 以找到更多的关于命令的消息。

很少有人愿意写这么庞大的一个 switch 语句，它很快就会变得难以控制。当你想增加一条新消息时，就很难找到该把 case 语句放在哪儿，尤其是在 switch 语句跨了几页的情况下，而且丢了一个 break 语句就会使事情变得更糟糕。

这就是 C/SDK Windows 程序解释消息的方法。下面让我们看一看在 C++程序中如何处理消息。

Windows 与 C++

考虑一下你会如何使用 C++类来处理 Windows 消息。当一个 C++程序员听到“多态性”这个词时，他（她）通常会立刻想到虚函数。乍看起来，用虚函数处理窗口消息似乎不错，你只要创建一个窗口基类，并为窗口可能接收到的每个消息提供一个虚成员函数。

虽然这一方法符合 C++和 OOP 风格，但它仍存在问题。请记住，虚函数是用类的虚函数表（virtual function table）实现的，而每个派生类都会带一个虚函数表的拷贝。虚函数表中每个入口都是一个 4 字节的指针。数一下虚函数要处理的消息的个数，将这个数目乘 4（因为每个指针要占 4 个字节），这样每个类就会在虚函数表中带来大量的额外字节。

用虚函数实现多态性行为还有一个问题。窗口消息的个数和种类会随时发生变化。在消息改变时，使用虚函数做消息处理函数就会导致代码废弃。

为了克服用虚函数处理消息所带来的困难，微软开发了高效的、可扩展的、不针对编译器的消息路由选择和处理系统。解决方案就是消息映射表。在最高层次上，消息映射表就是将窗口消息和命令与类的成员函数绑定在一起。这就是消息映射表背后的故事和它们在你的 MFC 应用程序是如何工作的。

MFC 消息映射内幕

MFC 的消息映射技术由两部分组成：CCmdTarget 类和消息映射表。CCmdTarget 是所有接收窗口消息和命令的对象的基类。消息映射表就是将窗口消息与处理消息的类成员函数关联起来的机制。消息映射表数据结构和消息映射宏是消息映射系统的另外两个重要

的方面。

CCmdTarget 类

为了能接收消息，类必须从 CCmdTarget 派生。CCmdTarget 的派生类都有必要处理和使用消息映射表。当你浏览 MFC 的层次图时，你会发现很多类都是它的派生类，包括 CWnd、CDocument 和 CWinApp。

消息映射表数据结构

在看到实际的消息映射表之前，我们先看两个用于实现 MFC 消息映射的数据结构。第一个是 AFX_MSGMAP_ENTRY，这个结构表示消息映射表的实际入口：

```
struct AFX_MSGMAP_ENTRY
{
   UINT nMessage;
   UINT nCode;
   UINT nID;
   UINT nLastID;
   UINT nSig;
   AFX_PMSG pfn;
};
```

第一个域（nMessage）表示 Windows 消息。它和出现在 C/SDK 程序的 switch 语句中的消息一样。第二个域（nCode）表示控件代码或 WM_NOTIFY 代码。这个域是 MFC 3.0 新增加的。第三个域（nID）是产生消息的控件 ID。第四个参数（nLastID）也是 MFC 3.0 新增加的。参数 nSig 表示用于处理消息的函数的签名（signature）（本章后面讨论）。最后一个参数（pfn）指向处理消息的函数。

第二个结构是 AFX_MSGMAP。它表示实际的消息映射表：

```
struct AFX_MSGMAP
{
   const AFX_MSGMAP* pBaseMap;
   const AFX_MSGMAP_ENTRY* lpEntries;
};
```

这个结构有两个部分：指向另一个 AFX_MSGMAP 结构的指针（实际上是基类的消息映射表）；一个 AFX_MSGMAP_ENTRY 数组。请注意这个结构是如何建立链表关系的。消息映射表基本上就是 AFX_MSGMAP_ENTRY 结构数组。一个应用程序的类层次结构中维护了一个消息映射表的链表。这也表明了 MFC 如何用消息映射表实现继承。如果类自身的消息映射表不能处理一个消息，框架会检查基类的消息映射表。基本上，框架沿着消息映射表向上找，直至找到能处理这个消息的函数。

消息映射宏

MFC 提供了以下几个宏来产生消息映射表：DECLARE_MESSAGE_MAP、BEGIN_MESSAGE_MAP 和 END_MESSAGE_MAP。这些宏都可以扩展成为 CCmdTarget 派生类定义和实现消息映射表的代码。

要在类中用消息映射表，最基本的方法是在类定义中包括 DECLARE_MESSAGE_MAP

（在.H 文件中），然后在 CPP 文件中增加 BEGIN_MESSAGE_MAP、END_MESSAGE_ MAP 和消息映射信息。

DECLARE_MESSAGE_MAP 定义在文件 AFXWIN.H 中，如下：

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

在类定义中使用 DECLARE_MESSAGE_MAP 意味着在类中定义了 3 种东西：（1）一个叫做_messageEntries 的 AFX_MSGMAP_ENTRY 结构数组；（2）一个名为 messageMap 的 AFX_MSGMAP；（3）一个获得类的消息映射表的函数（GetMessageMap()）。消息映射表入口（_messageEntries）和消息映射表结构（messageMap）是类的静态成员。这意味着在类里对于所有的对象只能有一个_messageEntries 数组和一个 messageMap 成员变量。

下面是一个 CView 的派生类 CTestView，它有一个消息映射表。预处理程序用消息映射宏来产生支持消息映射的代码。下面是 C++代码：

```
class CTestView : public CView
{
        ...
protected:
   //{{AFX_MSG(CTestView)
   afx_msg void OnLButtonDblClk(UINT nFlags, CPoint point);
   //}}AFX_MSG
   DECLARE_MESSAGE_MAP()
};
```

预处理程序会产生如下类定义：

```
class CTestView : public CView
{
        ...
protected:
        afx_msg void OnLButtonDblClk(UINT nFlags, CPoint point);
private:
        static const AFX_MSGMAP_ENTRY _messageEntries[];
protected:
        static const AFX_MSGMAP messageMap;
        virtual const AFX_MSGMAP* GetMessageMap() const;
};
```

一旦类定义了，就有两个宏来定义消息映射表：BEGIN_MESSAGE_MAP 和 END_MESSAGE_MAP。

正如名字暗示的那样，BEGIN_MESSAGE_MAP 宏标志着消息映射表的开始，在 AFXWIN.H 中，宏定义如下：

```
#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[] = \
        {
```

END_MESSAGE_MAP 宏表示消息映射表的结束，在 AFXWIN.H 中，它的定义如下：

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

当这些宏在实现文件（一个 C++文件）中一起使用时，这些宏就实现了消息映射表。对如下代码实验：

```
BEGIN_MESSAGE_MAP(CTestView, CView)
   //{{AFX_MSG_MAP(CTestView)
   ON_COMMAND(ID_STRING_CENTER, OnStringCenter)
   ON_WM_LBUTTONDBLCLK()
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

预处理程序会产生如下代码：

```
const AFX_MSGMAP* CTestView::GetMessageMap() const
{ return &CTestView::messageMap; }

AFX_DATADEF const AFX_MSGMAP CTestView ::messageMap =
{ &CView::messageMap, &CTestView::_messageEntries[0] };

const AFX_MSGMAP_ENTRY CTestView::_messageEntries[] =
{
   ON_COMMAND(ID_STRING_CENTER, OnStringCenter)
   ON_WM_LBUTTONDBLCLK()
   {0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 }
};
```

首先，宏产生了一个 GetMessageMap()函数，这个函数会返回类的消息映射表的指针。框架程序会用 GetMessageMap()来获得类的消息映射表。

然后宏会产生填充 AFX_MSGMAP 结构的代码。第一个域指向基类的消息映射表（在这种情况下是 CView 的消息映射表），同时创建一条到根对象的链表。这使得框架在找不到消息处理函数时能检查基类是否有消息处理函数。这个链表表明了 MFC 如何使用消息映射表实现继承。第二个域是指向类自己的第一个消息映射表的入口。

最后，宏会产生实际的消息表。消息表实际上就是 AFX_MSGMAP_ENTRY 结构的表。

注意，消息映射表中还有一些增加的宏，MFC 的文档中记录了一些这样的宏。它们都是以“ON_”开头的，并且都将窗口消息映射到对应的消息处理函数。

BEGIN_MESSAGE_MAP 和 END_MESSAGE_MAP 之间的内容是一系列消息映射表的入口宏。这些宏扩展后可以填充类的消息映射表_messageEntries。MFC 定义各种消息映射表的入口宏。如表 3-1 所示。

表 3-1 MFC 的消息映射表入口宏

消息类型	宏形式	参数
预定义的窗口消息	ON_WM_XXXX	None
命令	ON_COMMAND	Command ID, Handler name
更新命令	ON_UPDATE_COMMAND_UI	Command ID, Handler name
控件通知	ON_XXXX	Control ID, Handler name
用户定义的消息	ON_MESSAGE	User-defined message ID, Handler name
注册的窗口消息	ON_REGISTERED_MESSAGE	Registered message ID variable, Handler name
命令 ID 的范围	ON_COMMAND_RANGE	Start and end of a contiguous range of command IDs
用于更新的命令 ID 的范围	ON_UPDATE_COMMAND_UI_RANGE	Start and end of a contiguous range of command IDs
控件 ID 的范围	ON_CONTROL_RANGE	A control-notification code and the start and end of a contiguous range of command IDs

就上面的示例而言，消息映射表使用了两个消息映射表入口宏：ON_COMMAND 和 ON_WM_LBUTTONDBLCLK。这两个宏都可以在 AFXMSG_.H 中找到。如程序清单 3-2 所示。

程序清单 3-2 ON_COMMAND 和 ON_WM_LBUTTONDBLCLK 宏（在 AFXMSG.H 中）

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

ON_WM_LBUTTONDBLCLK macro:
#define ON_WM_LBUTTONDBLCLK() \
    { WM_LBUTTONDBLCLK, \
      0, \
      0, \
      0, \
      AfxSig_vwp, \
      (AFX_PMSG)(AFX_PMSGW) \
        (void (AFX_MSG_CALL CWnd::*)(UINT, CPoint))OnLButtonDblClk },
```

下面再回头看一下 AFX_MSGMAP_ENTRY 结构，看一下宏是如何填充这个结构的。第五个域指定了用于处理消息的函数的签名。框架程序用签名的值来确定返回类型和消息处理函数的参数。

MFC 如何使用消息映射表

现在你知道了消息映射表的组成和构造过程。下面我们看一下它是如何使用的。

一个 MFC 程序能处理以下两种消息：一般的窗口消息（像 WM_MOUSEMOVE）和命令（从菜单和控件产生的消息或由 WM_COMMAND 表示的消息）。消息映射表也可以处理这两种消息。

理解消息映射结构的最好方法就是在应用程序中跟踪一些消息。首先，我们会在应用程序框架中跟踪一条 WM_COMMAND 消息，看它都到了哪里，然后再跟踪一条窗口消息，我们看它从哪里开始到哪里结束。

MFC 窗口如何与 WndProc 建立关联

在 MFC 的 16 位版本（2.5）中，MFC 为所有的窗口（除了开发人员注册的非 MFC 窗口）注册了一个叫做 AfxWndProc()的消息处理过程。和其他的 Windows 应用程序一样，MFC 应用程序的消息也存放在窗口过程中。对于 16 位版本的 MFC 应用程序，这个函数是 AfxWndProc()。因为 MFC 窗口类和 AfxWndProc()一起注册为消息处理函数，所以很显然 AfxWndProc()是进行消息处理的地方。在系统中跟踪消息就变得很容易：你只要在 AfxWndProc()中设置一个断点。

对于 32 位版本的 MFC 来说，情况却不同。MFC 窗口类不再和 AfxWndProc()一起注册，而且和 DefWindowProc()一起注册为消息处理函数。如果你查看 MFC 的源代码，会发现还有 AfxWndProc()。它现在的任务是向各种 CWnd 派生对象发送消息。这些消息是如何在 AfxWndProc()那里结束的呢？

MFC 有两个自己的消息 hook：AfxMsgFilterHook()和_AfxCbtFilterHook()。因为消息是首先交给 hook 的，所以对于某些消息可以在这两个 hook 中的一个那里得到解释。MFC 用基于计算机训练的 hook 函数将 AfxWndProc()连到 MFC 窗口（在窗口创建时）。下面说明 AfxWndProc()是如何连入到 MFC 窗口的。

当一个新的 CWnd 派生类创建时，在调用 CWnd::CreateEx()过程中，MFC 都会安装 AfxCbtFilterHook()。在 CWnd::CreateEx()调用 CreateWindowEx() API 函数之前，CWnd::CreateEx()会调用 AfxHookWindowCreate()。它会在 hook 链中插入_AfxCbtFilterHook()，因为_AfxCbtFilterHook()是一个基于计算机训练的 hook，所以 Windows 在激活、创建、销毁、最小化、最大化、移动窗口或改变窗口大小之前，会调用_AfxCbtFilterHook()。

Windows 还在完成系统命令之前，从系统消息队列中删除一条鼠标、键盘事件之前，在设置键盘焦点之前，在与系统消息队列同步之前，调用_AfxCbtFilterHook()。

AfxCbtFilterHook 会像刚刚所说的那样接收消息，它会忽略所有的窗口消息，直至接收到 HCBT_CREATEWND。接收到 HCBT_CREATEWND 意味着窗口要创建了。_AfxCbtFilterHook()会调用_AfxStandardSubClass()。

_AfxStandardSubClass()会调用 SetWindowLong()将 AfxWndProc()放入窗口内。从现在开始，所有该窗口的消息都会经过 AfxWndProc()。在这个函数里，命令路由选择结构会处理消息。所以说，即使窗口开始注册了 DefWindowProc()，将它作为消息处理过程，框架程序还是会在 CWnd 派生类创建后由 AfxWndProc()来处理。

微软不将 AfxWndProc()作为注册窗口过程的原因是使用 DefWindowProc()可以支持 3D 控件。这些都在微软的 CTL3D.DLL 中。如果系统具有 CTL3D 功能已经是一种迫切需要，那应用程序就要覆盖 CTL3D 的功能（在处理 WM_CTLCOLOR 消息方面）。为了确保这一点，

MFC 必须保证按以下顺序调用：AfxWndProc()、CTL3D 的 WndProc()和最后的 DefWindowProc()。可见为了确保这一点，微软不得不允许 CTL3D 在 AfxWndProc()之前分类（subclass），这就意味着延迟 AfxWndProc()的引入，直至 CTL3DSubclassDlgEx 被调用。所以 MFC 在 DefWndProc()中完成注册，然后在_AfxStandardSubclass()中引入 CTL3D，最后在 CTL3D 之上安装 AfxWndProc 作为窗口过程的最后一步。为了保持向后兼容，微软仍然支持在启动时用 AfxWndProc()注册。

处理消息

MFC 用两种方式表示窗口：（1）用统一的系统定义的窗口句柄；（2）用表示窗口的 C++类。窗口句柄由 CWnd 和 CWnd 的派生类包装。因为窗口句柄是 CWnd 类的成员变量，所以从 CWnd 对象得到窗口句柄很容易。但从 CWnd 对象得到窗口句柄还需要一些额外工作。

在前面已经看到，MFC 用 CMapPtrToPtr 对象将句柄映射成 CWnd 对象。MFC 在窗口存在期间维护这个链接。如果使用 CWnd 创建一个窗口，窗口句柄就会和 CWnd 对象关联在一起，也就是说二者通过句柄映射表关联在一起，MFC 这样做就使得框架代码可以使用 C++对象，而不是使用窗口句柄。

现在回到窗口过程。AfxWndProc()处理一个特定消息：WM_QUERYAFXWNDPROC。如果消息是 WM_QUERYAFXWNDPROC，AfxWndProc 会返回数字 1。应用程序可以通过发送 WM_QUERYAFXWNDPROC 消息来检查该窗口是否是使用 MFC 消息映射系统的 MFC 窗口。CWnd 的 Dump 过程还用 WM_QUERYAFXWNDPROC 消息的返回值验证信息。如果消息不是 WM_QUERYAFXWNDPROC，AfxWndProc()会继续处理消息。

在 AfxWndProc()内部，框架程序会调用 CWnd::FromHandlePermanent()来获得与窗口句柄关联的 C++对象。然后，框架程序会调用 AfxCallWndProc()。注意，即使第一个参数指向一个 CWnd 对象，第二个参数仍是一个窗口句柄。这样 AfxCallWndProc()就可以在处理异常和调试时维护上一条消息的记录。下面是 AfxCallWndProc()的原型。可以看出它和其他的窗口过程很类似，只是它还还包含了一个 CWnd 指针。

```
LRESULT PASCAL _AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT message,
                               WPARAM wParam, LPARAM lParam);
```

AfxCallWndProc()首先检查消息，看它是否是一个 WM_INITDIALOG 消息。如果是，它就调用_AfxHandleInitDialog)。_AfxHandleInitDialog()使对话框自动置于窗口中间。如果放在窗口中间很合适（窗口还不可见，也没有移动），MFC 就会自动地把对话框放在父窗口的中间。AfxCallWndProc()函数会在线程状态的 m_lastSentMsg 成员变量中保存窗口句柄、消息、WPARAM 和 LPARAM。然后 AfxCallWndProc()会调用窗口对象的窗口过程。下面是 WinProc()的原型：

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

它和一般的窗口过程参数一致，因为此时 MFC 已经将窗口句柄映射为已有的 CWnd 派生类。

另外，CWnd::WindowProc()是虚函数，你可以覆盖它。覆盖它的一个理由是如果你想在 MFC 查看某个消息之前处理这个消息，例如，你可能有一个类不使用消息映射系统——可能

为了提高性能。

CWnd::WindowProc 调用 CWnd::OnWndMsg()。如果 OnWndMsg()返回 FALSE，然后 CWnd::WindowProc()会调用 CWnd::DefWindowProc()。

实际的处理过程在 CWnd::OnWndMsg()中，程序清单 3-3 显示了源代码。

程序清单 3-3 CWnd::OnWndMsg()代码的缩减版本

```
BOOL CWnd::OnWndMsg(UINT message, WPARAM wParam, LPARAM lParam, LRESULT*
pResult) {
   if (message == WM_COMMAND)
      call OnCommand(wParam, lParam) and return;

   if (message == WM_NOTIFY)
      call OnNotify(wParam, lParam, &lResult) and return;

   if (message == WM_ACTIVATE)
      call _AfxHandleActivate(this, wParam,
                              CWnd::FromHandle((HWND)lParam)) and return;

   if (message == WM_SETCURSOR) &&
      call _AfxHandleSetCursor(this,
                               (short)LOWORD(lParam),
                               HIWORD(lParam))) and return;

   // Find the message map entry in a message cache using its hash value

if (the message is in the cache ) {
      if (message map entry is NULL)
         return FALSE;

      if (message < 0xC000)   // Registered Windows message
         goto LDispatch;  // Call the function
      else
         goto LDispatchRegistered; // Call the function for a registered
                                   // message
   } else {
      // not in cache, look for it
      msgCache.nMsg = message;
      msgCache.pMessageMap = pMessageMap;

      for (/* pMessageMap already init'ed */; pMessageMap != NULL;
         pMessageMap = pMessageMap->pBaseMap) {

         if (message < 0xC000) {
            // constant window message
            if ((lpEntry = AfxFindMessageEntry(pMessageMap->lpEntries,
                                               message, 0, 0)) != NULL) {
                 msgCache.lpEntry = lpEntry;
                 goto LDispatch;
            }
         } else {
            // registered windows message
            lpEntry = pMessageMap->lpEntries;
```

```
            while ((lpEntry = AfxFindMessageEntry(lpEntry, 0xC000, 0, 0))
                  != NULL) {
               UINT* pnID = (UINT*)(lpEntry->nSig);
               // must be successfully registered
               if (*pnID == message) {
                  msgCache.lpEntry = lpEntry;
                  goto LDispatchRegistered;
               }
               lpEntry++;      // keep looking past this one
            }
         }
      }
      msgCache.lpEntry = NULL;
      return FALSE;
   }
LDispatch:
   ASSERT(message < 0xC000);
   union MessageMapFunctions mmf;
   mmf.pfn = lpEntry->pfn;

   switch (lpEntry->nSig) {
   default:
      ASSERT(FALSE);
      break;

   // Examine the signature type, calling the
   //   message handler with the appropriate parameters
   }
   goto LReturnTrue;

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

LReturnTrue:
   if (pResult != NULL)
      *pResult = lResult;
   return TRUE;
}
```

这个函数很大，就是它替代了原来的那个 switch 语句。在跟踪消息之前，我们先大致看一下 OnWndMsg()。首先，它要过滤出那些特定的消息：WM_COMMAND、WM_NOTIFY，WM_ACTIVATE 和 WM_SETCURSOR。框架程序有处理这些消息的特殊方法。如果消息不是这几个之一，OnWndMsg()会在消息映射表中查找消息。MFC 维护着一个消息映射表入口缓存，可以通过散列值（hash value）来访问它。这是一种优化，因为在散列表（hash table）中查询值比遍历消息映射表快。

这也就是命令和一般窗口消息的处理方式的不同。如果是命令消息（nMsg==WM_COMMAND），那么 CWnd::OnWndMsg 会调用 OnCommand()，否则它会检索窗口对象的消息映射表来处理消息。让我们首先看一下命令消息路由选择。

处理 WM_COMMAND

命令消息到命令目标的第一站就是 CWnd::OnCommand()。

CWnd::OnCommand()

OnCommand()是一个虚函数，所以框架程序必须要调用正确版本的 OnCommand()。在这个例子里，因为消息是为主窗口产生的，所以框架程序会调用 CFrameWnd 版本的 OnCommand()。

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

此时，消息剩下的只是它的 WPARAM 和 LPARAM。如果消息是在线帮助请求，则框架程序会发送给框架窗口一个 WM_COMMANDHELP 消息，否则消息会交给基类的 OnCommand()——CWnd::OnCommand()。

OnCommand 首先检查表示控件的 LPARAM。如果命令消息是由控件产生的，LPARAM 就会包含控件窗口。如果消息是控件通知（例如 LBN_CHANGSEL），框架程序就会执行特定的处理过程。如果消息是为某个控件产生的，OnCommand()会将消息直接发送给控件，然后 OnCommand()返回。

否则，CWnd::OnCommand()要确保产生命令的用户界面元素没有被禁用（例如菜单项），然后将消息传递给 OnCmdMsg()（也是虚函数）。因为框架窗口还要尝试处理消息，所以 CFrameWnd::OnCmdMsg 被调用。这个函数在 WINFRM.CPP 中。

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

CFrameWnd::OnCmdMsg()在调用时，pExtra 和 pHandlerInfo 是 NULL，因为处理命令不需要这一信息。

CFrameWnd::OnCmdMsg()将取出的消息按以下顺序经过应用程序的各个部分：

- 活动视图。
- 活动视图的文档。
- 主窗口。
- 应用程序。

为了将命令路由到活动视图，CFrameWnd::OnCmdMsg 首先调用 CWnd::GetActiveView()得到活动视图。如果 CFrameWnd::OnCmdMsg()找到了活动视图，它就可以调用活动视图的 OnCmdMsg()。如果活动视图不能处理该命令，文档就会获得命令。如果 CFrameWnd::OnCmdMsg()没找到活动视图，或者视图和文档都不能处理消息，主窗口就有机会处理消息。最后，如果主窗口不想处理消息，应用程序就要对这个消息进行处理——CFrameWnd::OnCmdMsg()会调用应用程序的 OnCmdMsg()函数。

在这个例子里，消息先到达了活动视图和 CView::OnCmdMsg()函数。该函数在 VIEWCORE.CPP 中：

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

框架程序给视图的窗框部分一个对消息做出反应的机会，这通过调用 CWndTarget::OnCmdMsg()实现。如果视图不能处理消息，就会转移到文档。

因为 CWnd 不能覆盖 OnCmdMsg()，所以命令就会直接到达 CCmdTarget::OnCmdMsg()，如 CMDTARG.CPP 所示。

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

CCmdTarget::OnCmdMsg()会遍历消息映射表（如果有必要，还要追溯到基类），从而寻找消息处理函数。如果找到了，它就会调用那个函数；如果没找到，CCmdTarget::OnCmdMsg()会返回 FALSE，文档就有机会处理消息。如果文档不对消息做任何处理，消息就由 CWnd 的 DefWindowProc()处理。

如果 CCmdTarget::OnCmdMsg()在消息映射表中找到了消息处理函数，它就会调用 DispatchCmdMsg()，该函数也在 CMDTARG.CPP 中：

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

这个函数是静态的，也就是说，它仅在 CMDTARG.CPP 内可见。一个参数是函数签名，这个签名来自消息映射表入口自身（记住，消息映射表的一个域是函数声明）。还有一个指向消息处理函数的指针（pfn）做为参数。这个函数指针在消息映射表入口中也有。

DispatchCmdMsg()根据函数签名是一般命令的函数签名、扩展命令的函数签名、命令用户接口处理函数的函数签名或 Visual Basic 控件的函数签名而执行不同的操作。如果是一般的菜单命令，签名就会是 AfxSig_xx（返回为空，参数链表为空）。DispatchMessage()会立刻调用消息处理函数。

如果 CCmdTarget::OnCmdMsg()在消息映射表中没有找到消息处理函数，它会返回 FALSE，最终会由 CWnd::DefWindowProc()来处理。

表 3-2 给出了各种 MFC 函数的签名、参数和返回值

表 3-2 消息映射中使用的函数签名

签名	返回类型	参数
AfxSig_bD	BOOL	CDC*
AfxSig_bb	BOOL	BOOL
AfxSig_bWww	BOOL	CWnd*, UINT, UINT
AfxSig_hDWw	HBRUSH	CDC*, CWnd*, UINT
AfxSig_iwWw	int	UINT, CWnd*, UINT
AfxSig_iWww	int	CWnd*, UINT, UINT
AfxSig_is	int	LPTSTR
AfxSig_lwl	LRESULT	WPAPAM, LPARAM
AfxSig_lwwM	LRESULT	UINT, UINT, CMenu*
AfxSig_vv	void	void
AfxSig_vw	void	UINT
AfxSig_vww	void	UINT, UINT
AfxSig_vvii	void	int, int (wParam is ignored)
AfxSig_vwww	void	UINT, UINT, UINT

续表

签名	返回类型	参数
AfxSig_vwii	void	UINT, int, int
AfxSig_vwl	void	UINT, LPARAM
AfxSig_vbWW	void	BOOL, CWnd*, CWnd*
AfxSig_vD	void	CDC*
AfxSig_vM	void	CMenu*
AfxSig_vMwb	void	CMenu*, UINT, BOOL
AfxSig_vW	void	CWnd*
AfxSig_vWww	void	CWnd*, UINT, UINT
AfxSig_vWh	void	CWnd*, HANDLE
AfxSig_vwW	void	UINT, CWnd*
AfxSig_vwWb	void	UINT, CWnd*, BOOL
AfxSig_vwwW	void	UINT, UINT, CWnd*
AfxSig_vs	void	LPTSTR
AfxSig_vOWNER	void	int, LPTSTR(force return TRUE)
AfxSig_iis	int	int, LPTSTR
AfxSig_wp	UINT	CPoint
AfxSig_wv	UINT	void
AfxSig_vPOS	void	WINDOWPOS*
AfxSig_vCALC	void	NCALCSIZE_PARAMS*
AfxSig_vNMHDRpl	void	NMHDR*, LRESULT*
AfxSig_vwNMHDRpl	void	UINT, NMHDR*, LRESULT*
AfxSig_bwNMHDRpl	BOOL	UINT, NMHDR*, LRESULT*
特定于 CCmdTarget 的签名		
AfxSig_cmdui	void	CCmdUI*
AfxSig_cmduiw	void	CCmdUI*, UINT
AfxSig_vpv	void	void*
AfxSig_bpv	BOOL	void*
基于实现的其他别名		
AfxSig_vwwh	void	UINT, UINT, HANDLE
AfxSig_vwp	void	UINT, CPoint
AfxSig_bw = AfxSig_bb	BOOL	UINT
AfxSig_bh = AfxSig_bb	BOOL	HANDLE
AfxSig_iw = AfxSig_bb	int	UINT
AfxSig_ww = AfxSig_bb	UINT	UINT
AfxSig_bv = AfxSig_wv	BOOL	void

续表

签名	返回类型	参数
AfxSig_hv = AfxSig_wv	HANDLE	void
AfxSig_vb = AfxSig_vw	void	BOOL
AfxSig_vbh = AfxSig_vww	void	BOOL, HANDLE
AfxSig_vbw = AfxSig_vww	void	BOOL, UINT
AfxSig_vhh = AfxSig_vww	void	HANDLE, HANDLE
AfxSig_vh = AfxSig_vw	void	HANDLE
AfxSig_end = 0	[消息映射表的结尾标记]	

MFC 对所有的 CCmdTarget 派生类使用该命令路由选择方案。这些 CCmdTarget 类包括从 CWnd、CDocument、CView 和 CFrameWnd 派生的类。命令到达最终目的地的路径很有趣，所有命令消息的前3步都是一样的。这3步是消息首先进入 AfxWndProc()，其中 AfxWndProc() 从 HWND 参数中得到 CWnd 对象，并且调用 AfxCallWndProc()。AfxCallWndProc()调用 CWnd 派生类的 WindowProc()，从这开始，消息就到达了它的目的地。下面是命令消息在 MFC 应用程序各个组成部分中路径的总结。

到达框架窗口的命令

下面是 WM_COMMAND 消息到达应用程序的框架窗口的路径。在所有的窗口消息经过 MFC 程序时，第一站都是 AfxWndProc()。它会调用 AfxCallWndProc()，最后到达某个窗口的窗口过程。从这里开始，命令消息就会被送到适当的命令目标那里。

```
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CWnd::OnWndMsg()
CFrameWnd::OnCommand()
CWnd::OnCommand()
CFrameWnd::OnCmdMsg()
CCmdTarget::OnCmdMsg()
DispatchCmdMsg()
CMainFrame::OnFrameAfamecommand()
```

到达文档的命令

下面是 WM_COMMAND 消息到达应用程序的文档的路径。

```
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CWnd::OnWndMsg()
CFrameWnd::OnCommand()
CWnd::OnCommand()
CFrameWnd::OnCmdMsg()
CView::OnCmdMsg()
CDocument::OnCmdMsg()
CCmdTarget::OnCmdMsg()
DispatchCmdMsg()
CSdiappDoc::OnDoccommandAdoccommand()
```

到达视图的命令

下面是 WM_COMMAND 消息到达应用程序的视图的路径。

```
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CWnd::OnWndMsg()
CFrameWnd::OnCommand()
CWnd::OnCommand()
CFrameWnd::OnCmdMsg()
CView::OnCmdMsg()
CCmdTarget::OnCmdMsg()
DispatchCmdMsg()
CSdiappView::OnViewAviewcommand()
```

到达应用程序的命令

下面是 WM_COMMAND 消息到达应用程序的 CWinApp 派生类的路径。

```
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CWnd::OnWndMsg()
CFrameWnd::OnCommand()
CWnd::OnCommand()
CFrameWnd::OnCmdMsg()
CCmdTarget::OnCmdMsg()
DispatchCmdMsg()
CSdiappApp::OnAppAnappcommand()
```

到达对话框的命令

对话框也能接收命令消息。下面是 WM_COMMAND 消息到达对话框的路径。

```
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CWnd::OnWndMsg()
CWnd::OnCommand()
CDialog::OnCmdMsg()
CCmdTarget::OnCmdMsg()
DispatchCmdMsg()
CAboutDlg::OnAButton()
```

这就是命令消息经过框架的过程。你可以看到，消息在不同的类之间运动。处理一般的窗口消息（像 WM_SIZE）就相对简单一些。

处理一般窗口消息

和命令消息一样，窗口消息也是从 AfxWndProc()开始的。AfxWndProc()也会将窗口句柄转化成 CWnd 对象。然后 AfxWndProc()函数会调用 AfxCallWndProc()，AfxCallWndProc()又会调用 CWnd 对象的 WindowProc()。WindowProc()会调用 OnWndMsg()，OnWndMsg()会调用 AfxFindMessageEntry()来为消息找到处理函数。

AfxFindMessageEntry()

AfxFindMessageEntry()是一个有趣的函数。它有两个版本，一个是用汇编语言写的，另一个是用C语言写的。如果应用程序和Intel的机器一起编译，AfxFindMessageEntry()就会用汇编语言版本。否则它就会使用C语言版本。

为了在表中找到消息映射表入口，OnWndMsg()首先获得CWnd对象的消息映射表。然后AfxFindMessageEntry()根据消息映射表中的第一个入口，遍历消息映射表入口数组，直至在消息映射表中找到消息或遍历到了以AfxSig_end签名标记的消息映射表的结尾（这也就是END_MESSAGE_MAP宏所做的）。

当OnWndMsg()找到一个处理函数时，它就会调用这个处理函数。如果消息映射表中没有这个消息的处理函数，框架程序会调用窗口的默认窗口过程。所以，和命令不一样的是，命令要被送到多个地方，而窗口消息只要送到需要它的窗口。例如，看一下WM_SIZE消息到目的地（一个CWnd派生类）的路径：

将WM_SIZE消息送给视图

下面是WM_SIZE消息到达应用程序的视图的路径。你会看到窗口消息到CWnd派生类是一条直线。

```
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CWnd::OnWndMsg()
CSdiappView::OnSize()
```

调用成员函数

在离开消息映射表之前，我们先看一下当MFC在消息映射表中找到入口后，它如何为特定的消息调用成员函数。可以回想一下，消息映射表入口结构有一个成员变量是指向处理窗口消息的函数的。为了解决这个问题，OnWndMsg()会在栈顶声明一个MessageMapFunction联合（union）。程序清单3-4给出了这个联合的一部分：

程序清单3-4 MFC的消息处理函数的签名

```
union MessageMapFunctions
{
   AFX_PMSG pfn;   // generic member function pointer

   // specific type safe variants
   BOOL    (AFX_MSG_CALL CWnd::*pfn_bD)(CDC*);
   BOOL    (AFX_MSG_CALL CWnd::*pfn_bb)(BOOL);
   BOOL    (AFX_MSG_CALL CWnd::*pfn_bWww)(CWnd*, UINT, UINT);
   BOOL    (AFX_MSG_CALL CWnd::*pfn_bHELPINFO)(HELPINFO*);
   HBRUSH  (AFX_MSG_CALL CWnd::*pfn_hDWw)(CDC*, CWnd*, UINT);
   HBRUSH  (AFX_MSG_CALL CWnd::*pfn_hDw)(CDC*, UINT);
   int     (AFX_MSG_CALL CWnd::*pfn_iwWw)(UINT, CWnd*, UINT);
   int     (AFX_MSG_CALL CWnd::*pfn_iww)(UINT, UINT);
   int     (AFX_MSG_CALL CWnd::*pfn_iWww)(CWnd*, UINT, UINT);
   int     (AFX_MSG_CALL CWnd::*pfn_is)(LPTSTR);
   LRESULT (AFX_MSG_CALL CWnd::*pfn_lwl)(WPARAM, LPARAM);
   LRESULT (AFX_MSG_CALL CWnd::*pfn_lwwM)(UINT, UINT, CMenu*);
   void    (AFX_MSG_CALL CWnd::*pfn_vv)(void);
```

```
    void    (AFX_MSG_CALL CWnd::*pfn_vw)(UINT);
    void    (AFX_MSG_CALL CWnd::*pfn_vww)(UINT, UINT);
    void    (AFX_MSG_CALL CWnd::*pfn_vwii)(UINT, int, int);
...
};
```

MessageMapFunction 联合里有一个 AFX 函数指针，它后面都是可能用做消息处理函数的各种函数的原型。OnWndMsg()将 MessageMapFunction 联合设置为消息处理函数的地址：

```
mmf.pfn = lpEntry->pfn;
```

下一步，OnWndMsg()会继续为 WM_SIZE 消息找到匹配的签名。一旦 OnWndMsg()找到了一个匹配的签名，它就会从 WPARAM 和 LPARAM 中取出必要的参数，并使用 MessageMapFunction 结构中的原型调用处理函数。

例如，下面是为 WM_SIZE 执行处理函数的一行代码：

```
case AfxSig_vwii:
   (this->*mmf.pfn_vwii)(wParam, LOWORD(lParam), HIWORD(lParam));
   break;
```

很老套，不是吗？有些人说看 MFC 就像是在访问香肠工厂。香肠的味道很好，而工厂的工作秩序又很好，但你永远看不到香肠是怎么做的。MFC 的消息映射系统的代码很难理解。但是它却工作得很好，而且它已经是 Visual C++的向导技术的重要组成部分。

除了 WM_COMMAND 和一般的窗口消息，MFC 还以特殊方式处理另外几种窗口消息。

其他类型的消息

你的应用程序所接收到的消息中最常见的就是 WM_COMMAND 和各种窗口消息（像 WM_SIZE 和 WM_MOVE）。但是，MFC 还有特殊的方法来处理 WM_NOTIFY、WM_ACTIVATE 和 WM_SETCURSOR 消息。下面我们来简单地看一下这些特殊情况。

WM_NOTIFY 和消息反射（Message Reflection）

在 Windows 的早期版本中，WM_COMMAND 是一个复杂的消息，它表示来自菜单项的命令或者子窗口发出的通知。随着 Windows 95 和新的通用控件的出现，又有一个新的消息叫做 WM_NOTIFY。WM_NOTIFY 是一个总括的控件的通知。WM_NOTIFY 也是一个通知，WM_COMMAND 可以是通知也可以是命令。

CWnd::OnWndMsg()用 CWnd::OnNotify()处理 WM_NOTIFY 消息。WM_NOTIFY 没有传入 WPARAM 和 LPARAM 参数，而是在 LPARAM 中打包了一个 NMHDR 结构。

```
struct NMHDR {
   HWND hwndFrom;      // control that sent notification
   UINT idFrom;        // ID of control
   UINT code;          // notification code
};
```

OnNotify()会使用 hwndFrom 域并结束调用虚 CWnd 函数 OnChildNotify()。然后，OnChildNotify()会直接将消息送给控件，以便控件处理消息（毕竟，这是控件的责任）。

除了处理新控件的 WM_NOTIFY，MFC 还提供了一种新的机制——消息反射（Message Reflection）。回忆一下控件通知在 MFC 早期版本中是如何工作的：每当有控件感兴趣的事情

发生时，控件都会通知父类，由父类来做一些处理。一个很好的例子就是在对话框中对控件着色。你可以在对话框中放置控件，但是如果你想修改它的颜色，应该由对话框来对WM_CTLCOLOR消息作出反应，同时对控件着色。当然，作这样的事情这不是最好的方法，因为它不是自包含的。在理想世界里，控件应该自己负责绘画，在MFC 4.0中，消息反射就提供了这一性能。

MFC为了处理反射的消息而定义了一些消息映射宏，它们和标准的消息映射宏类似，只是它们都带有标签“REFLECT”。例如WM_CTLCOLOR消息的标准处理函数是ON_WM_CTLCOLOR。ON_WM_CTLCOLOR有两个参数，第一个参数是一个函数，这个函数的参数是一个指向CDC的指针，第二个参数是一个无符号整数，它会返回一个HBRUSH。在消息映射表中增加这个宏会使父窗口在收到WM_CTLCOLOR时通知控件。这样控件就可以处理消息，而不至于依赖父窗口进行重新着色。

WM_ACTIVATE

MFC的消息结构还处理WM_ACTIVATE。OnWndMsg()会调用非C++成员函数_AfxHandleActivate()。这个函数会检查最高层窗口是否是WM_ACTIVATE。如果是，_AfxHandleActivate()会向最高层窗口发送WM_ACTIVATETOPLEVEL消息。

WM_SETCURSOR

MFC在_AfxHandleSetCursor()函数中处理WM_SETCURSOR消息。_AfxHandleSetCursor()会检查是否有鼠标按钮按下。如果鼠标按钮按下，_AfxHandleSetCursor()会激活最后一个活动窗口，这涉及到Windows的一个bug。如果窗口是被禁用的，而某个模态对话框的父窗口是这个被禁用的窗口，那么当用户点击禁用的窗口时，应用程序就不能正确地激活自己。用户应该直接点击对话框。如果对话框很小，就很容易覆盖。一般说来，在禁用的窗口中点击会使DefWindowProc()产生警鸣。MFC修改了这个bug。当你点击一个带有活动模态对话框的被禁用窗口时，MFC会激活对话框（这样就将应用程序的窗口放在最前面）。

进入消息循环：PreTranslateMessage()

MFC还有另一个重要特性：你可以在消息还没到达命令目标或窗口时插入消息循环中来处理消息。PreTranslateMessage()用于完成这一任务。MFC为你提供了两个插入点：CWinApp::PreTranslateMessage()和CWnd::PreTranslateMessage()。CWinApp::PreTranslateMessage()使你能够在窗口消息在到达应用程序的任何窗口或命令目标之前处理消息。

在消息由TranslateMessage()和DispatchMessage()处理之前，CWinApp::Run()会调用CWinApp::PreTranslateMessage()。CWndApp::PreTranslateMessage()只带一个参数：一个指向MSG结构的指针。默认情况下，框架程序会检查消息。如果消息是鼠标点击或鼠标双击，则CWnd::PreTranslateMessage()会调用CancelToolTip()来从屏幕中清除所有工具提示。

然后，CWinApp::PreTranslateMessage()会从目标窗口（由消息结构中的窗口句柄指定）到应用程序的主窗口调用每个窗口。CWnd::PreTranslateMessage()会执行程序清单3-5所示的函数。

程序清单 3-5 CWnd 的 WalkPreTranslateTree()函数

```
BOOL PASCAL CWnd::WalkPreTranslateTree(HWND hWndStop, MSG* pMsg)
{
   ASSERT(hWndStop == NULL || ::IsWindow(hWndStop));
   ASSERT(pMsg != NULL);

   // walk from the target window up to the hWndStop window checking
   //  if any window wants to translate this message

   for (HWND hWnd = pMsg->hwnd; hWnd != NULL; hWnd = ::GetParent(hWnd))
   {
      CWnd* pWnd = CWnd::FromHandlePermanent(hWnd);
      if (pWnd != NULL)
      {
         // target window is a C++ window
         if (pWnd->PreTranslateMessage(pMsg))
            return TRUE; // trapped by target window (eg: accelerators)
      }

      // got to hWndStop window without interest
      if (hWnd == hWndStop)
         break;
   }
   return FALSE;       // no special processing
}
```

在调用 WalkPreTranslateTree() 之前，CWinApp::PreTranslateMessage() 会调用 AfxGetMainWnd()来获得应用程序主窗口的窗口句柄。WalkPreTranslateTree()会利用这个句柄知道何时停止调用 PreTranslateMessage()。然后从内部开始（就是从目标窗口开始），WalkPreTranslateTree()会联系每个窗口，试图找到一个对消息感兴趣的窗口。对于树中的每个窗口，WalkPreTranslateTree()会调用每个 CWnd 对象的 PreTranslateMessage()，而从窗口句柄映射表中获得 CWnd 对象。WalkPreTranslateTree()会为每个窗口完成这样的过程，直至窗口处理了消息或 WalkPreTranslateTree()到达了应用程序的主窗口。

CWnd::PreTranslateMessage()只有一个参数：一个指向 MSG 结构的指针。当你覆盖 PreTranslateMessage()时，只要检查 MSG 结构。如果你对这个消息有兴趣，只管在 PreTranslateMessage()里处理，然后返回 TRUE。

如果 CWinApp::PreTranslateMessage()找到一个对该消息感兴趣的窗口，它就会返回 TRUE。如果 CWinApp::PreTranslateMessage()返回 TRUE，CWinApp 的消息泵就不会分发消息。所以 PreTranslateMessage()有效地吞掉了消息。

结 语

MFC 的标准消息映射技术与使用虚类成员函数实现消息处理相比是一个很好的选择。它减少了大量的虚函数表，而且很有效，又独立于编译器。

你应该已经很好地掌握了 MFC 如何处理应用程序（初始化和消息泵）以及 MFC 如何处理 Windows 应用程序的窗口方面的问题（消息处理）。现在我们开始看框架中的各个不同的组成部分。首先我们要看实用类。

CHAPTER

4 MFC 实用类

MFC 实用类不提供任何绘画功能，也不提供庞大的应用程序特性。但是，它们对 MFC 程序员来说很重要。开始时看这些类如何工作似乎作用不大，但是，因为你写程序用的类很多都来自实用类，所以学习 MFC 在实用类中采用的技术会帮助你为自己的工程写出更好的实用类。

在本章中，我们会先看一些简单类的内部，例如 CString、CTime、CTimeSpan 和窗口结构包装函数（例如 CPoint、CSize 和 CRect）。在看到 MFC 如何实现这些简单类之后，我们会检查各种集合类，看它们如何工作。然后，我们将剖析 CFile 文件实用类和它的一些派生类。最后，我们会看到 MFC 通过 CException 和它的派生类所提供的高级错误处理支持。

这样，我们就能指出那些有趣的技术和一些没有文档说明的信息。

简单值类型

简单值类型实用类封装了已有的 C++类型和 Windows 类型，例如 char *和 RECT 结构。这些类除了提供被封装类型的 C++接口以外，还增加了一些扩展的特性。

简单值类型类提供了 C++操作，这些操作可以更安全、更容易地管理类型。其中一些值类型类还增加了内存管理、国际化（internationalization）和其他有帮助的成员函数来增强基本类型性能。

一个典型的封装基本类型的类就是 CString。下面我们先看一下 CString，然后再看其他的简单值类型。

类 CString

CString 封装了一个字符串，而且提供了一些高级特性，包括：

- 国际化。
- 内存分配与回收。
- 操作符（例如+、＝和＝＝）。
- 字符串搜索。
- 字符串操作（例如反转和连接）。

- 格式化（与 printf 类似的输出）。

使用 CString

在看 MFC 如何实现 CString 之前，我们先看一下如何使用 CString 的特性。

CString 提供了很多构造函数，这样你可以从一个 char *、另一个 CString 甚至是一个你想多次重复的字符创建一个 CString。下面是 CString 的一些构造样例：

```
//Create a CString from a char *
CString myString1("This is a string");
//Create a CString from a CString
CString myString2(myString1);
//Create a string with 80 Z characters
CString myString3('Z',80);
//Create an empty CString
CString myString4;
```

CString 操作符

CString 的操作符包括相加、比较、赋值和访问字符串中的字符。程序清单 4-1 给出了 CString 的一些操作符：

程序清单 4-1 CString 的一些操作符

```
CString myString1("string 1");
CString myString2("string 2");
CString myString3("string 3");

// ** Addition (concatenation) and assignment operators **
myString1 += myString2;
// result is "string 1string2" in myString1

myString3 = "This is " + myString2;
// result is "This is string 2" in myString3

myString2 += 'Z';
// result is "string 2Z" in myString2

// ** Comparison operators **
if (myString1 == myString2) // Like strcmp
... // Do something

if (myString1 < myString2)
... // Do something

if (myString1 != myString2)
... // Do something

// ** Operator [] (access char in string) **
// Get the eighth character (0 based array)
if (myString1[7] == '1')
... // Do something
```

CString 还为搜索字符串提供了很多成员函数，如下所示：

- Find()——在字符串中查找子串或字符。如果找到，返回在字符串中的索引，否则返

回-1.

- FindOneOf()——参数是 char *，在 CString 字符串中查找参数字符串中出现的第一个字符。如果找到，就返回第一个出现字符在字符串（CString）中的索引，否则返回 1。
- ReverseFind()——和 Find()一样，但它从字符串的右端开始，向左查找，如果找到，返回在字符串中的索引，否则返回-1。

CString 提供了一个 Format()成员函数，这个函数的参数和 printf 的很像，它会将结果变成 CString。

例如，下面的代码段：

```
CString myString1; myString1.Format("There are %d at %x !",1000,0xffff);
```

得到的结果就是“There are 1000 at 0xfff inside myString1”。

你可以剪切字符串、将字符全部变为大写或小写、取出子串以及在 CString 上执行很多其他操作。程序清单 4-2 给出了一些 CString 上的操作。

程序清单 4-2 一些 CString 上的操作

```
CString myString("This is a string that I'm going to manipulate!")
// Chop out Y (13) characters starting at X (22) and put it into
// newString.
CString newString = myString.Mid(22,5);
// yields: "I'm going to "
// Return the right ten characters
newString = myString.Right(10);
// yields: "This is a ".

// Nuke white space
myString.TrimRight();
// yields: "ThisisastringthatI'mgoingtomanipulate!"

// Upper case the string.
myString.MakeUpper();
// yields: "THISISASTRINGTHATI'MGOINGTOMANIPULATE!"

// Reverse the string
myString.MakeRevers();
// yields: "!ETALUPINAMOTGNIOGM'ITAHTGNIRTSASISIHT"
```

CString 的引用计数

在 MFC 4.0 中，微软在 CString 类中增加了引用计数——一个很友好的特性。引用计数确保只有在完全有必要时才复制字符串数据。CString 引用计数是自动发生的，但是你可以调用 LockBuffer()成员函数锁定字符串缓冲区，并用 UnlockBuffer()来解锁。引用计数减少了内存的使用，而且提高了性能，因为字符串的拷贝变少了。

CString 内幕

CString 的特性很有趣，所以下面让我们看 CString 类是如何实现的。

CString 文件

CString 的声明在 AFX.H 文件中，具体实现分布在 AFX.INL（内嵌文件）、STRCORE.CPP 和 STREX.CPP 中。

CString 的声明很长，因为它支持的操作符和成员函数很多。程序清单 4-3 给出了 CString 声明的缩写版本，它突出了类中那些针对实现和没有文档说明的部分。

程序清单 4-3 CString 声明的缩写版本（在文件 AFX.H 中）

```
class CString
{
// NOTE: Lots of operators and member functions
// omitted for brevity..

// Implementation
public:
    ~CString();
    int GetAllocLength() const;

protected:
    LPTSTR m_pchData;    // pointer to ref counted string data
// implementation helpers
    CStringData* GetData() const;
    void Init();
    void AllocCopy(CString& dest, int nCopyLen, int nCopyIndex, int
                   nExtraLen) const;
    void AllocBuffer(int nLen);
    void AssignCopy(int nSrcLen, LPCTSTR lpszSrcData);
    void ConcatCopy(int nSrc1Len, LPCTSTR lpszSrc1Data, int nSrc2Len,
                    LPCTSTR lpszSrc2Data);
    void ConcatInPlace(int nSrcLen, LPCTSTR lpszSrcData);
    void FormatV(LPCTSTR lpszFormat, va_list argList);
    void CopyBeforeWrite();
    void AllocBeforeWrite(int nLen);
    void Release();
    static void Release(CStringData* pData);
    static int SafeStrlen(LPCTSTR lpsz);
};
```

m_pchData：理解 CString 的关键

在程序清单 4-3 中可以看到，CString 只有一个数据成员——LPTSTR m_pchData，它作为指针指向 CString 封装的字符串。但是如果 CString 是带有引用计数的，这么计数放在哪里呢？字符串的长度如何？当然，不是每次操作都需要计算字符串的长度。

仔细看 CString 的源代码，你会发现一些关于 CString 的没有文档说明的有趣事实，从而可以回答这些问题。

CStringData 辅助结构

在 AFX.H 中，CString 的声明之前，你会看到 CStringData 辅助结构的声明。这个结构如程序清单 4-4 所示。

程序清单 4-4 CStringData 辅助结构的声明

```
struct CStringData
{
    long nRefs;         // reference count
    int nDataLength;
    int nAllocLength;
    // TCHAR data[nAllocLength]
```

```
    TCHAR* data()
    { return (TCHAR*)(this+1); }
};
```

这个辅助结构包含了引用计数需要的 CString 信息，但是不在 AFX.H 中使用。下面看一下 CString::AllocBuffer()中哪部分分配 m_pchData。在 CString 需要为字符串申请空间时，会调用 AllocBuffer()。程序清单 4-5 包含了这个没有文档说明的 CString 例程的伪代码。

程序清单 4-5 CString::AllocBuffer()的实现

```
void CString::AllocBuffer(int nLen)
{
    if (nLen == 0)
        Init();
    else{
        CStringData* pData =
          (CStringData*)new BYTE[sizeof(CStringData) +
          (nLen+1)*sizeof(TCHAR)];
        pData->nRefs = 1;
        pData->data()[nLen] = '\0';
        pData->nDataLength = nLen; pData->nAllocLength = nLen;
        m_pchData = pData->data();
    }
}
```

AllocBuffer()会分配一块内存，内存的大小是 CStringData 的大小加上要申请的字符串的大小。然后，AllocBuffer()会初始化 CStringData 成员，最后在将 CStringData 复制到新分配的内存块后面，将 m_pchData 指针设置为新分配的内存。如果没有指定长度，AllocBuffer()会调用 Init()，其中 Init()会将 m_pchData 设置为特殊的“空”CStringData，以便函数能够避免检查到 NULL。“空”字符串是提前锁定的（它的引用计数是-1），这样可以避免对引用计数的操作。图 4-1 显示了 AllocBuffer()中 CStringData 的分配。

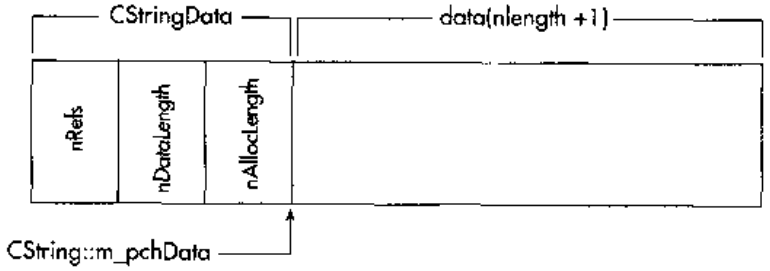


图 4-1 AllocBuffer()分配的内存

你可以看到，在 m_pchData 指向分配的数据之前，CString 隐藏了它用于引用计数的信息。现在，很多问题都是关于为什么 MFC 组有这样古怪的行为？为什么不在 CString 中增加一些新的成员，而冒险使用类型不安全的指针？

挖掘 CStringData

我们问过 MFC 的权威人士 Dean McCrory 关于这样做的动机，他回答说：

> 如果 CString 在二进制形式上很像 TCHAR*，这样就会很方便（也就是说，你逐个对照组成这两种对象的各个位时会发现它们都是一样的）。引用计数信息不在 CString 中，因为这些内容是

数据，而不是有引用计数的 CString 本身（在引用对象中的引用没有很多好处——引用计数必须在被引用的对象中）。这两个长度成员（nDataLength 和 nAllocLength）在 CStringData 对象中的原因与上面类似：它们描述了实际的数据。它们应该和每个 CString 在一起，但是对引用同一数据的 CString 对象来说是多余的。

微软本来可以将 m_pchData 改为 CStringData 指针，并以一种更安全的方式访问数据，但遗留下的 MFC 应用程序（引用计数是从 MFC 4.0 才开始使用的）应该有从 CString 派生类和通过 protect 成员访问 m_pchData 缓冲区指针的代码，如果将 m_pchData 改为 CStringData，就势必要修改这些代码，影响了向后兼容性，如程序清单 4-6 所示。

程序清单 4-6　在 MFC 4.0 中可能受到影响的代码

```
//My string adds some new string functionality
class CMyString : public CString {
public:
    MyNeatFunction();
    //...

};
CMyString::MyNeatFunction()
{
    char buffer[256];
    strcpy(buffer, "I'm a WILD MFC HACKER! ");
    strcat(buffer,myString.m_pchData);
}
```

将 m_pchData 修改为 CStringData 指针会影响所有通过 protect 成员变量 m_pchData 访问 CString 缓冲区的 MFC 类。MFC 小组为了不破坏已有的 MFC 程序付出的代价是很大的。

根据 Dean McCrory 所说，隐藏 CStringData 还有第三个原因：

另一个原因是支持问题。对于人们来说，写这样一段代码再普通不过：

```
printf("The string %s was not found in file %s.",strSearch,
    strFileName);
```

在以前版本的 MFC 中，这种代码必定要被放大，因为 sizeof(CString)比 4 要大，所以 printf 可能会将一些不是指针的数据当作指针使用。正确的代码应该是：

```
printf("The string %s was not found in file
    %s.",(LPCTSTR)strSearch,LPCTSTR)strFileName);
```

虽然问题看起来很简单，但要解决它也不容易。当我们使用引用计数时，我们才意识到可以解决，前面受影响的代码也能正常运行。

所以，隐藏 CStringData 的 3 个原因是：

- 使 TCHAR 方便使用。
- 保持向后兼容性。
- 解决一个常见的 CString 的问题。

揭示引用计数

引用计数是 CString 中最有趣的部分。引用计数限制了字符串数据的拷贝个数，使得只有在必须进行数据拷贝时才拷贝（这些操作包括写字符串和修改字符串）。例如，考虑下面的

代码：

```
CString myString1("This is a string");
CString myString2(myString1);
CString myString3 = myString1;
```

只要我们不对字符串进行写操作，我们就可以只使用一个缓冲区，缓冲区的内容是“This is a string”，并将 myString2 和 myString3 指向 myString1，如图 4-2 所示。

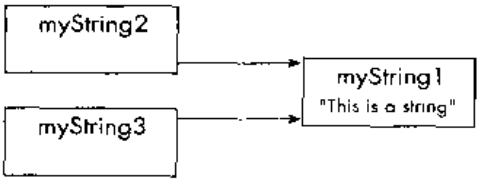


图 4-2 CString 的引用计数

引用计数是 MFC 编程技巧中很有用的一个，因为你可以使用类似的引用计数系统来优化自己的 MFC 类。事实上，引用计数在我们讨论 OLE 时会成为一个很重要的话题。

CStringData::nRefs 引用计数是引用计数算法的关键。引用计数的初始化值是 0，字符串每增加一个引用，计数就加 1，删除一个引用，计数就会减 1。如果引用计数为 0，就表示没有任何引用了，这时，就可以把字符串删除了。如果 CString 发现有对引用字符串的写操作，它会自动地拷贝引用字符串，并将原有字符串的引用计数减 1。

仔细想一想，你会发现，就像操作系统中的很多系统级资源一样，所有 CString 引用的数据都可以看作是一种临界资源。如果有两个线程在运行，而两个线程又都要同时增加/删除同一个引用会怎么样呢？

MFC 用两个 Win32 API 函数解决这个问题：InterlockedIncrement() 和 InterlockedDecrement()。这两个 API 还解决了所有操作系统级别的共享资源问题。它们确保 nRefs 每次都只增加 1 或减少 1，无论多少进程同时访问这个值。

下面看一个示例，我们可以从中看出引用计数算法是如何运行的，数字 1~5 表示这些行代码的执行顺序：

```
1.  CString myString1("This is my string1!");
2.  CString myString2(myString1);
3.  CString myString3; myString = myString2;
4.  // Write something!
5.  myString3.MakeUpper();
```

从 MFC 引用计数算法的角度讲，前三行代码不需要拷贝数据，此时 myString1、myString2、myString3 都指向同一个字符串“This is a string1!”。在第五行，CString 发现有写操作，于是它创建了新数据，此时就有两个数据对象“This is my string1!”的两个引用（myString1，myString2）以及“THIS IS MY STRING1!”的一个引用（myString3）。

下面我们针对这个例子跟踪 CString 的源代码，看一下引用计数如何运行。

第一行代码是调用 CString 的构造函数，参数是一个 char *。CString 的这个构造函数在 STRCORE.CPP 中，如程序清单 4-7 所示。

程序清单 4-7 第一行代码调用的 CString 构造函数

```
CString::CString(LPCTSTR lpsz)
{
    Init();
    int nLen = SafeStrlen(lpsz);
    if (nLen != 0)
        AllocBuffer(nLen); memcpy(m_pchData, lpsz, nLen*sizeof(TCHAR));
}
```

Init()成员函数将 nRefs 初始化为-1，将所有其他 CStringData 成员清零。然后，构造函数确定参数的长度，接着调用 AllocBuffer()分配一块内存（见程序清单 4-5）。最后 CString 的构造函数将参数内容拷贝到新分配的内存块中。

此时，为 CString 变量 myString1 调用 AllocBuffer()，使 nRefs 的值变为 1。

然后，在第二行调用 CString 的另一个构造函数，参数为 CString 引用，程序清单 4-8 是这个构造函数的伪代码。

程序清单 4-8 第二行调用的 CString 构造函数

```
CString::CString(const CString& stringSrc)
{
    if (stringSrc.GetData()->nRefs >= 0){
        m_pchData = stringSrc.m_pchData;
        InterlockedIncrement(&GetData()->nRefs);
    }else{
        Init();
        *this = stringSrc.m_pchData;
    }
}
```

如果 stringSrc 参数的引用计数是 0 或更大，这个构造函数会将对象的数据指针指向 stringSrc 的数据。注意下面这行代码：

```
m_pchData = stringSrc.m_pchData;
```

这行代码也使得字符串完全共享 CStringData 的信息和数据。因此，调用 InterlockedIncrement()会对 myString1 和 myString2 引用的计数加 1。“else”子句对理解 LockBuffer()和 UnlockBuffer()的作用也很重要。如果一个 CString 引用的数据被加锁，那么它引用就不能被其他 CString 引用。

而且，加锁的 CString 不能再引用其他字符串。如果你想得到一个特殊的字符串实例，即这个字符串实例不与其他字符串共享数据，也不使用其他 CString 的数据——换句话说，如果你想创建一个非引用计数的（non-reference-counted）CString 实例，那么加锁就变得很重要了。

例如，在考虑到 MFC ODBC 类时，你必须给 ODBC 一个绑定的地址。在你移动数据库内容时，它会将数据转移到这个地址。这个地址不能够修改，也不能收回空间。为了达到这一目的，我们在绑定到这个地址（*m_pchData*）之前调用 CString::LockBuffer()。

现在，对于 myString1 和 myString2 来说，nRefs 都是 2。两个 CString 都指向同一个数据，这个数据是由第一次调用 AllocBuffer()时分配的。

示例中的第三行代码是第一次调用 myString3 的默认构造函数。这个构造函数将所有数

据初始化为-1，然后调用了赋值操作符。程序清单4-9给出了CString赋值操作符的伪代码。

程序清单4-9 示例的第三行调用的CString赋值操作符

```
const CString& CString::operator=(const CString& stringSrc)
{
    if (m_pchData != stringSrc.m_pchData)    {
        if ((GetData()->nRefs < 0 && GetData() != afxDataNil) ||
          stringSrc.GetData()->nRefs < 0){
        // actual copy necessary since one of the strings is locked
              AssignCopy(stringSrc.GetData()->nDataLength,
                  stringSrc.m_pchData);
        }else{  // can just copy references around
            Release();
            m_pchData = stringSrc.m_pchData;
            InterlockedIncrement(&GetData()->nRefs);
        }
    }
    return *this;
}
```

示例运行时会进入赋值操作符的“else”分支，如果有必要就先释放掉缓冲区。在我们的示例中，调用Release()不会做任何事，因为引用计数不是0或更小。然后赋值操作符会创建一个指向myString1数据的引用，然后将nRefs引用计数增加到3。

如果字符串被加锁（nRefs=-1），则赋值操作符的“if”分支拷贝缓冲区。

在示例中，此时运行到了第四行。在myString3被构造之后以及MakeUpper()被调用之前，3个CString对象（myString1、myString2和myString3）都指向使用AllocBuffer()分配的初始数据。

在第五行，示例调用了CString::MakeUpper()成员函数，也就是对共享缓冲区进行了写操作。这样CString就拷贝了缓冲区，以免3个字符串都指向修改了的大写字符串。程序清单4-10显示了MakeUpper()的伪代码。

程序清单4-10 MakeUpper()成员函数（在示例中的第五行被调用）

```
void CString::MakeUpper()
{
    CopyBeforeWrite();
    ::CharUpper(m_pchData);
}
```

CString::CopyBeforeWrite()成员函数拷贝缓冲区，并更新引用计数，所以让我们看看这个函数。

程序清单4-11显示了CopyBeforeWrite()的伪代码。

程序清单4-11 CopyBeforeWrite()成员函数，它负责在写操作执行之前拷贝字符串数据

```
void CString::CopyBeforeWrite()
{
    if (GetData()->nRefs > 1){
        CStringData* pData = GetData();
        Release();
        AllocBuffer(pData->nDataLength);
        memcpy(m_pchData, pData->data(),
```

```
            (pData->nDataLength+1)*sizeof(TCHAR));
    }
}
```

首先，CopyBeforeWrite()要验证引用计数大于 1，以免进行没必要的拷贝。然后，它调用 Release()成员函数释放了一个引用，然后为数据分配了新的缓冲区，同时将 nRefs 置为 1。最后，CopyBeforeWrite()将数据从前面的字符串拷贝到新的缓冲区中。

总结一下引用计数的这个示例。一旦示例代码运行就会得到两套字符串数据：一个是由 myString1 和 myString2 共享的字符串数据，另一个是 myString3 引用的全部大写的字符串。

在看 CString 的另一个特性之前，我们先看一下 Release()成员函数，看它是如何删除一个引用的。程序清单 4-12 中显示了 CString::Release()的伪代码。

程序清单 4-12 CString::Release()，删除 Cstring 引用的成员函数

```
void CString::Release()
{
    if (GetData() != afxDataNil){
        ASSERT(GetData()->nRefs != 0);
        if (InterlockedDecrement(&GetData()->nRefs) <= 0)
            delete[] (BYTE*)GetData();
        Init();
    }
}
```

在检查到字符串不是 NULL 以及引用确实有需要释放的之后，Release()函数调用了 InterlockedDecrement()。如果引用计数在减 1 之后变成 0 或者更小，就说明再没有指针指向数据了（包括 CString 指针“this”），而且可以释放数据了。在前面的示例中，Release()调用会删除数据。CString 的析构函数也会做同样的检查，以便在 CString 被删除时缓冲区不被破坏。

包装引用计数

一旦你知道了 MFC 用于隐藏引用计数的小技巧，你就会发现 CString 的引用计数算法其实很简单。作为练习，你应该浏览 STRCORE.CPP 文件，并注意那些调用 CopyBeforeWrite()的函数。而且你应该试试找出为什么+=操作符不调用 CopyBeforeWrite()。

其他有趣的 CString 内幕

大部分 CString 成员函数依赖_t 国际化的字符串函数之一。例如程序清单 4-13 所示的 CString::Find()的伪代码。

程序清单 4-13 CString::Find()——一个 CString 调用_t 字符串例程的示例

```
int CString::Find(TCHAR ch) const
{
    // find first single character
    LPTSTR lpsz = _tcschr(m_pchData, ch);

    // return -1 if not found and index otherwise
    return (lpsz == NULL) ? -1 : (int)(lpsz - m_pchData);
}
```

Find()函数将参数和内部的字符串指针传递给_tcschr()，_tcschr()会映射到我们熟悉的处

理单字节字符串的 strchr()函数，在调用_tcschr()函数之后，Find()会修改调用的结果。

继续……

现在我们已经讨论了值类型类中最复杂的一个，下面看一些其他值类型类的有趣内幕。

一些其他简单类

当然，CString 不是 MFC 中惟一的简单类，但它和其他简单类的不同之处在于它完全是内嵌的，没有在封装类中增加过多的特性。

下面列出了所有非 CString 的简单类、它们封装的结构以及实现它们的源文件：

值的类型	结构	源文件
CPoint	POINT(struct tagPoint)	afxwin1.inl
CRect	RECT(struct tagRECT)	afxwin1.inl, wingdix.cpp
CSize	SIZE(struct tagSIZE)	afxwin1.inl
CTime	time_t operations	afx.inl, timecore.cpp
CTimeSpan	time_t math	afx.inl, timecore.cpp

我们先看 CRect 类，因为它的实现和其他几个很类似。

CRect 类

CRect 的定义在 AFXWIN.H 里，它是 tagRECT 的派生类：

```
class CRect : public tagRECT
```

tagRECT 是一个标准的 Windows 结构，它在 WINDEF.H 文件中，声明如下：

```
typedef struct tagRECT
{
    LONG    left;
    LONG    top;
    LONG    right;
    LONG    bottom;
} RECT, *PRECT, NEAR *NPRECT, FAR *LPRECT;
```

MFC 通过使 CRect 从 tagRECT 派生出来，从而维护了 CRect 和 tagRECT 之间的兼容性。这意味着，如果函数的参数链表中有 CRect，你就可以用 tagRect 结构。而且，MFC 在 tagRECT 的基础上又增加了新的性能，以避免复制 left/top/right/bottom 等数据成员。

CRect 的所有成员函数都以各种方式对这 4 个数据成员进行操作。例如，DeflateRect()成员函数会根据参数中的矩形参数减小当前的矩形。WINGDIX.CPP 文件中的 DeflectRect()的伪代码如程序清单 4-14 所示。

程序清单 4-14 CRect::DeflateRect()，一个典型的简单类的成员函数

```
void CRect::DeflateRect(LPCRECT lpRect)
{
    left += lpRect->left;
    top += lpRect->top;
    right -= lpRect->right;
    bottom -= lpRect->bottom;
}
```

和其他大部分简单类的成员函数一样，这个简洁的成员函数避免了程序员总是访问那 4 个数据成员。

如果你已经看懂一个简单类，就等于掌握了所有的简单类。所以我们下面看 MFC 如何实现下一种实用类——集合类。

MFC 的集合类

MFC 提供了 3 种用来存储数据的封装的抽象数据结构。对于每种形式的集合，MFC 提供了几种无模板的类来保存像 WORD、int、BYTE、CString 以及 CObject 这样的类型。MFC 还为每种形式提供了有模板的版本，这样你可以在自己的类中使用类型安全的集合。

MFC 集合类的形式

MFC 支持的 3 种形式是数组（array）、链表（list）和映射表（map）。数组和 C++的数组一样，只是它处理所有的内存分配操作。MFC 的链表是双链表。映射表，有时也称字典（dictionary），它保存了一对值，这样你可以利用键（key）立刻得到它对应的值。

表 4-1 给出了所有 MFC 集合类的名字、形式以及它们存储的类型。

表 4-1 所有的 MFC 集合类

名字	形式	类型 1	类型 2
CByteArray	Array	BYTE	
CDWordArray	Array	DWORD	
CUintArray	Array	unsigned int	
CObArray	Array	CObject	
CStringArray	Array	CString	
CWordArray	Array	WORD	
CObList	List	CObject*	
CPtrList	List	void*	
CStringList	List	CString	
CMapPtrToPtr	List	void*	void*
CMapStringToOb	Map	CString	CObject*
CMapStringToPtr	Map	CString	void*
CMapStringToString	Map	CString	CString
CMapWordToOb	Map	WORD	CObject*
CMapWordToPtr	Map	WORD	void*

在本节，我们会介绍每种形式的 MFC 集合类，从数组开始，然后是链表，最后是映射表。每个 MFC 集合类不仅仅是新的抽象数据类型，而且还具有一些可以应用于 C++类的有趣的技术。

MFC 数组集合类

我们首先回顾一个示例来说明如何使用 MFC 数组集合类。然后我们再看数组是如何实现的。

数组回顾

使用 MFC 数组很简单。只要用 SetSize()成员函数初始化数组，再调用 SetAt()成员函数在数组中增加元素即可。如果要获得数组中的值，就要调用 GetAt()方法或[]操作符。可以调用 DeleteAt()来从数组中删除元素。

程序清单 4-15 中的示例将无符号的整数数组中每个元素置为其索引的两倍。

程序清单 4-15 使用数组的示例

```
void ArrayExample()
{
    CUintArray myIntArray;
    //Initialize size
    myIntArray.SetSize(100);

    //Fill it up with 2 times index.
    for (int i = 0; i < 100;i++)
        myIntArray.SetAt(i,2*i);
    // Now access item 50
    UINT bogus = myIntArray[50];
    //Alternative access
    bogus = myIntArray.GetAt(50);
    //Delete item 50
    myIntArray.DeleteAt(50);
}
```

MFC 数组内幕

非模板的数组类的声明在 AFXCOLL.H 头文件中。它的实现在 ARRAY_X.CPP 源文件中，其中 X 表示数组的类型。例如，CStringArray 就位于 ARRAY_S.CPP 中。所有的 MFC 模板集合类位于 AFXTEMPL.H 头文件中。

各种 CArray 的实现是完全相同的，仅数据类型不同。我们先看 CWordArray 类（一个 Word 的数组），并用它作为例子来说明 MFC 如何实现这些数组集合类。

程序清单 4-16 有 CWordArray 声明的一个缩写版本，其中有文档说明的构造函数、属性和操作等都省略了，这样我们就可以着重关心//Implementation 部分。

程序清单 4-16 CWordArray 的声明，在 AFXCOLL.H 中（有文档说明的成员被省略了）

```
class CWordArray : public CObject
{

DECLARE_SERIAL(CWordArray) public:
// Construction **omitted
// Attributes **omitted
```

```
// Operations **omitted
// ...

// Implementation  -The undocumented parts!!
protected:
    WORD* m_pData;   // the actual array of data
    int m_nSize;     // # of elements (upperBound - 1)
    int m_nMaxSize;  // max allocated
    int m_nGrowBy;   // grow amount

public:
    ~CWordArray();
    void Serialize(CArchive&);
};
```

所有的 CArray 数据成员都是没有文档说明的，所以下面简单描述了每个数据成员的作用：

- m_pData——指向数组中实际数据的指针。对于不同的数组，这个成员的类型有所不同。例如，在 CStringArray 集合类里它是一个 CString 指针。
- m_nSize——数组当前的大小，值由 SetSize()设置。
- m_nMaxSize——因为数组内每次增长的量大于一个 WORD，所以有一些元素是分配后没有初始化的。m_nMaxSize 跟踪数组的物理大小，而 m_nSize 跟踪数组的逻辑大小或者说是对开发人员可见的大小。
- m_nGrowBy——每次分配内存时，一大块内存中可以存放的元素的个数。MFC 提供了最佳猜测增长尺寸（best-guess grow-by size）策略来减少内存碎片。

CWordArray 的实现

你会在 ARRAY_W.CPP（W 表示 WORD）文件中找到 CWordArray 的实现。MFC 数组最有趣的部分是它管理内存的方法。

在开始之前请注意一点：每个访问数组的 CArray 操作会先检查待访问的索引处的内存是否已经分配。如果没有分配，CArray 会调用 SetSize()成员函数分配内存。

数组如何增长：SetSize()成员函数

SetSize()是理解 CArray 内存处理的关键，所以我们先看它。它的具体代码如程序清单 4-17 所示。

程序清单 4-17　CWordArray::SetSize()实现的伪代码

```
void CWordArray::SetSize(int nNewSize, int nGrowBy)
{
    if (nGrowBy != -1)
        m_nGrowBy = nGrowBy;  // set new size
    if (nNewSize == 0){ //shrink to nothing
        delete[] (BYTE*)m_pData;
        m_pData = NULL;
        m_nSize = m_nMaxSize = 0;
    } else if (m_pData == NULL){ //create on with exact size
        m_pData = (WORD*) new BYTE[nNewSize * sizeof(WORD)];
        // zero fill m_nSize = m_nMaxSize = nNewSize;
        memset(m_pData, 0, nNewSize * sizeof(WORD));
```

```
    } else if (nNewSize <= m_nMaxSize){ //use an already allocated block
                                        //of memory
        if (nNewSize > m_nSize)
            memset(&m_pData[m_nSize], 0, (nNewSize-m_nSize) * sizeof(WORD));
        m_nSize = nNewSize;
    }
    else{ // Grow array
        int nGrowBy = m_nGrowBy;
        if (nGrowBy == 0)
            nGrowBy = min(1024, max(4, m_nSize / 8));
        int nNewMax;
        if (nNewSize < m_nMaxSize + nGrowBy)
            nNewMax = m_nMaxSize + nGrowBy;  // granularity
        else
            nNewMax = nNewSize;  // no slush
        WORD* pNewData = (WORD*) new BYTE[nNewMax * sizeof(WORD)];
        // copy new data from old
        memcpy(pNewData, m_pData, m_nSize * sizeof(WORD));
        // construct remaining elements
        memset(&pNewData[m_nSize], 0, (nNewSize-m_nSize) * sizeof(WORD));
        // get rid of old stuff (note: no destructors called)
        delete[] (BYTE*)m_pData;
        m_pData = pNewData;
        m_nSize = nNewSize;
        m_nMaxSize = nNewMax;
    }
}
```

SetSize()成员函数可以分成以下 5 个部分：

1. 初始化部分——如果 m_nGrowBy 的值不是默认的-1，则 SetSize()会先设置它的值。

2. 缩减到无的部分——如果用户调用 SetSize(0)，就会执行“if (nNewSize == 0)”部分。m_pData 就会被释放，所有的成员变量都会被设置成零。

3. 第一次分配内存的部分——如果前一个大小不是 0，就会执行“else if (m_pData == NULL)”部分，这也就是第一次调用 SetSize()的情况。它会首先分配一块和申请的大小一样大的内存块，然后利用 memset()将它初始化为 0。接着代码块会把 m_nSize 和 m_nMax 设置为新的大小。例如，如果调用 SetSize(0)，然后调用 SetSize(100)，这部分代码就会被执行。

4. 从额外空间分配内存的部分——“else if (nNewSize <= m_nMaxSize)”部分会检查申请的新块的大小是否超过了数组的物理大小，如果是，它会调用 memset()把那些元素初始化为 0，然后更新 m_nSize 来反应新的逻辑大小。

5. 增长数组的部分——最后一个块从“else”开始。如果数组已经存在，但需要增长时，就会调用这部分代码。实际上，这块也完成了大部分工作。

首先，如果没有指定 nGrowBy，“else”部分会使用下面计算的结果赋值给 nGrowBy：

```
min(1024, max(4,m_nSize/8);
```

这一步计算使内存每次增长的部分都大于 3，从而帮助避免堆碎片。使用 1024 有些武断，因为它分配的空间你可能用不完。微软认为 1023 个多余元素的限制是默认情况下最好的。你可以根据自己的数据来修改它的值。

这个块的下一部分会将 nNewMax 的值设置成所有申请的大小和增长的大小的和。然后

分配新的数据，再调用 memcpy 把旧的数据从 m_pData 拷贝到新分配的区域里。位于申请大小范围内的元素被初始化为 0，旧的数据被清除掉了。m_pData 数据成员也会得到更新，指向新分配的内存。最后，m_nSize 和 m_nMaxSize 成员变量也都被更新了。

代码的最后部分代价很大，因为它不得不分配、拷贝和释放大块的数据。在每次分配内存后增加的额外内存更加突出了这块代码的影响。只要数组的增长还在可增长范围内，就不会再调用分配/拷贝/释放内存的操作。

在数组中增加元素：SetAt()和 InsertAt()

我们已经知道数组是如何被分配内存的，下面我们看 MFC 如何在数组中增加元素。增加元素的过程在 SetAt()和 InsertAt()的实现中。你可以调用 SetAt()在数组中放置一个元素，也可以调用 InsertAt()在数组中插入一个元素。

SetAt()的实现很短，所以它嵌入在 AFXCOLL.INI 文件中。下面是 SetAt()的实现（只有一行）：

```
m_pData[nIndex] = newElement;
```

SetAt()直接访问数组的内部元素，就像访问一般的 C++数组一样。

InsertAt()的代码稍长，代价也大一些，因为它要将所有大于等于目的索引的所有元素向后移动一位。程序清单 4-18 是 InsertAt()的伪代码。

程序清单 4-18　CWordArray::InsertAt()的实现

```
void CWordArray::InsertAt(int nIndex, WORD newElement, int nCount)
{
    if (nIndex >= m_nSize) // adding after the end of the array
        SetSize(nIndex + nCount);  // grow so nIndex is valid
    else {    // inserting in the middle of the array
        int nOldSize = m_nSize;
        SetSize(m_nSize + nCount);  // grow it to new size
        // shift old data up to fill gap
        memmove(&m_pData[nIndex+nCount], &m_pData[nIndex],
              (nOldSize-nIndex) * sizeof(WORD));
        // re-init slots we copied from
        memset(&m_pData[nIndex], 0, nCount * sizeof(WORD));
    }
    // insert new value in the gap
    while (nCount--)
        m_pData[nIndex++] = newElement;
}
```

如果用户在数组的尾部插入元素，InsertAt()会先调用 SetSize()增长数组的大小，然后插入新值。

如果用户在数组的中部插入元素，InsertAt()会增加数组的大小，然后调用 memmove 将数组元素向后移动。最后新值就会插入。记住这些细节可以帮助你避免过度使用代价昂贵的 InsertAt()方法。

使用 GetAt()和操作符[]从数组中获得元素

在讨论 MFC 的链表之前，先看 MFC 如何支持通过 GetAt()和操作符[]访问数组。这两个成员函数在内嵌文件 AFXCOLL.INL 中。程序清单 4-19 显示了这两个函数的伪代码。

程序清单 4-19 MFC 获得数组元素的成员函数

```
WORD CWordArray::operator[](int nIndex) const
{
return GetAt(nIndex);
}

WORD CWordArray::GetAt(int nIndex) const
{
return m_pData[nIndex];
}
```

你可以从代码中看出，操作符[]实际上也调用了 GetAt()，它其实也是访问 m_pData 数据成员的一般方法。通过使所有对数组的访问都经过 GetAt()，MFC 小组在后来的实现中修改了 GetAt()的代码。这也是你使用自己的 C++类时很重要的数据抽象技术。

总而言之，MFC 数组封装了 C++数组类型的大部分。它们还增加了一些有趣的内存处理函数（我们会在 SetAt()成员函数中看到）。因为有了这些内存处理函数，所以应该避免使用某些成员函数，例如 InsertAt()。你还可以提供自己的增长因子，使得它更适用于你对数组的使用，从而调整内存处理方案。

我们下一个要讨论的 MFC 集合类是声名狼藉的链表。

链表

本节，我们首先看如何使用 MFC 的链表，然后分解 CStringList 类来揭示一个没有文档说明的类——CPlex。

MFC 的链表是双链表，它支持如下几种操作：

- AddHead()——在链表的头部增加一个元素。
- Find()——在链表中搜索某个元素。
- GetCount()——获得链表中元素的个数。
- GetHead()——得到链表中的第一个元素（头元素）。
- GetNext()——得到链表中位置为 POSITION 的元素的下一个元素。
- GetTail()——得到链表中的最后一个元素。
- IsEmpty()——判断链表是否已经空。
- RemoveAll()——从链表中删除所有元素。
- RemoveHead()——删除链表中的第一个元素。
- RemoveTail()——删除链表中的最后一个元素。
- InsertAfter()——在给定位置 POSITION 后插入一个元素。
- InsertBefore()——在给定位置 POSITION 前插入一个元素。

可以通过 POSITION 遍历器（iterator）来访问链表中的元素。它提供了用来获得特定元素的 POSITION 的成员函数，还提供了用来获得特定 POSITION 的元素的成员函数。例如，下面是一段遍历一个 CStringList 的代码：

```
// ... m_strList is the string list.
POSITION pos = m_strList.GetHeadPosition();     while (pos != NULL)
CString tmpString = m_strList.GetNext(pos);
```

MFC 链表内幕

下面我们将 CStringList（字符串链表）作为示例来讨论链表。其他的链表都很类似，只不过它们所包含的是 CObject 指针或 void 指针，而不是 CString。CStringList 的声明在 AFXCOLL.H 头文件中。具体的实现在 LIST_S.CPP 源文件中。

CStringList 的声明

程序清单 4-20 给出了 CStringList 的声明。和 CWordArray 一样，声明中有一部分在 MFC 中已经有文档说明了，所以在这里就忽略了这部分声明，而集中精力讨论那些没有文档说明的部分。

程序清单 4-20 CStringList 的声明（在文件 AFXCOLL.H 中）

```
class CStringList : public CObject
{
DECLARE_SERIAL(CStringList)
protected:
    struct CNode {
        CNode* pNext;
        CNode* pPrev;
        CString data;
    };
public:
// Construction  **omitted.
// Attributes (head and tail) **omitted
// Operations **omitted
// iteration ** omitted
// Implementation
protected:
    CNode* m_pNodeHead;
    CNode* m_pNodeTail;
    int m_nCount;
    CNode* m_pNodeFree;
    struct CPlex* m_pBlocks;
    int m_nBlockSize;

    CNode* NewNode(CNode*, CNode*);
    void FreeNode(CNode*);

public:
    ~CStringList();
    void Serialize(CArchive&);
};
```

这个类包含 CNode 结构的嵌入声明。CNode 结构就是实际的元素，也是链表中的节点。因为 MFC 的链表是双链表，所以 CNode 由一个 CString 数据域和一对指针（pNext/pPrev）组成。

在 CStringList 的//Implementation 部分，我们可以找到一些典型的链表项：

- m_pNodeHead——一个指向链表中第一个元素的 CNode 指针（头节点）。
- m_pNodeTail——一个指向链表中最后一个元素的 Cnode 指针（尾节点）。
- m_nCount——链表中节点的个数。

在 CStringList 中有 3 个没有文档说明的数据成员：m_pNodeFree、m_pBlocks 和 m_nBlockSize。有 2 个没有文档说明的成员函数：NewNode()和 FreeNode()。

为了理解这些成员函数的作用，我们先看一个 m_pBlocks 指向的新的辅助结构 CPlex。

CPlex：*没有文档说明的集合类的内存辅助结构*

CPlex 结构的声明在私有头文件 AFXPLEX_.H 中，如程序清单 4-21 所示。CPlex 的具体实现在 MFC 的 PLEX.CPP 源文件中。

程序清单 4-21 MFC 集合类的内存管理所使用的没有文档说明的 CPlex 结构(在文件 AFXPLEX_.H 中)

```
struct CPlex
{
    CPlex* pNext;
    UINT nMax;
    UINT nCur;
    void* data() { return this+1; }

    static CPlex* PASCAL Create(CPlex*& head, UINT nMax, UINT cbElement);
    void FreeDataChain();
};
```

从它的声明可以看出，CPlex 也是节点的定义。它只有 pNext 指针，所以说这实际上是单链表。数据占用的内存是在类之后（this+1）直接分配的。为了理解如何使用这种新的节点类型，我们先看 CStringList::NewNode() 成员函数。不要着急，在知道了 CStringList::NewNode()中为什么使用 CPlex 之后，我们会返回到 CPlex 的成员变量 nMax、nCur 以及成员函数 Create()和 FreeDataChain()。

CStringList::NewNode()

如果浏览 LIST_S.CPP 文件，你会发现 NewNode()是 CStringList 内存管理的关键部分。所有的 Add()和 Insert()成员函数都会先调用 NewNode()分配一个节点，然后再将节点插入链表。NewNode()也是我们研究的惟一一个和 m_pBlock 这个 CPlex 指针打交道的成员函数。CStringList::NewNode()的伪代码在程序清单 4-22 中。

程序清单 4-22 所有链表分配的基础：CStringList::NewNode()的实现（在文件 LIST_S.CPP 中）

```
CStringList::CNode*
CStringList::NewNode(CStringList::CNode* pPrev, CStringList::CNode* pNext)
{
    if (m_pNodeFree == NULL){// add another block
        CPlex* pNewBlock = CPlex::Create(m_pBlocks, m_nBlockSize,
            sizeof(CNode));
        // chain them into free list
        CNode* pNode = (CNode*) pNewBlock->data();
        // free in reverse order to make it easier to debug
        pNode += m_nBlockSize - 1;
        for (int i = m_nBlockSize-1; i >= 0; i--, pNode--)
            pNode->pNext = m_pNodeFree; m_pNodeFree = pNode;

    }

    CStringList::CNode* pNode = m_pNodeFree;
    m_pNodeFree = m_pNodeFree->pNext;
```

```
    pNode->pPrev = pPrev;
    pNode->pNext = pNext;
    m_nCount++;
    ASSERT(m_nCount > 0);  // make sure we don't overflow
    ConstructElement(&pNode->data);
    return pNode;
}
```

NewNode()首先检查 m_pNodeFree 是否指向了某个空闲的 CNode 结构。如果没有空闲的的 CNode 节点，NewNode()会调用 CPlex::Create()成员函数，参数是 m_nBlockSize 和 sizeof(CNode)，分别表示块的大小和元素的大小。CPlex::Create()会返回一个 CPlex 指针，指针指向大小为 m_nBlockSize 个 CNode 元素的内存空间。

MFC 维护了一个空闲 CNode 元素的单链表，理解在 NewNode()中 CPlex::Create()调用之后的代码可以帮助你知道这一点。m_pNodeFree 指向这个单链表的头节点，链表中的元素利用 CNode->pNext 链接在一起。MFC 在以下两种情况下使用 pNext 指针：链表被使用时和 MFC 维护一个空闲链表时。

在 Create()之后的代码遍历了新分配的内存块，从后到前地将每个 CNode 节点放到空闲链表 m_pNodeFree 中。

在分配完内存并且将新分配的这些 CNode 放到空闲链表中后，NewNode()会从空闲链表中取出一个 CNode 节点，并且更新空闲链表的头节点（如果有必要还要更新尾节点）。然后，NewNode()更新这个 CNode 节点的 pPre 指针和 pNext 指针，使其分别指向 pPrev 参数和 pNext 参数。最后 NewNode()增加了链表内部的节点计数，并且调用 ConstructElement，它用于清除 CNode 节点中的数据部分。

注意：CString 集合类使用了很多没有文档说明的实用函数，位于 SRC\ELEMENTS.H 头文件中。虽然我们不打算很详细地讨论这些函数，但一定要检查这些实用函数是否可用，因为有些在你自己的 CString 集合类中很有用。

回到 CPlex

我们从 NewNode()的代码可以看出，CPlex 是与 CNodes 密切相关的一个基本内存块。我们先介绍 CPlex 的 Create()函数，看 CPlex 的成员函数 nMax、nCur 和 pNext 是如何使用的。程序清单 4-23 给出了 CPlex::Create()的伪代码，它的定义在文件 PLEX.CPP 中。

程序清单 4-23　CPlex::Create()的实现（在文件 PLEX.CPP 中）

```
CPlex* PASCAL CPlex::Create(CPlex*& pHead, UINT nMax, UINT cbElement)
{
    CPlex* p = (CPlex*) new BYTE[sizeof(CPlex) + nMax * cbElement];
    // may throw exception
    p->nMax = nMax;
    p->nCur = 0;
    p->pNext = pHead;
    pHead = p;  // change head (adds in reverse order for simplicity)
    return p;
}
```

仔细看这个函数，我们会发现 pNext 将各个节点用 CPlex 链接在一起，以得到一个单链

表。这个链表按逆序（与分配顺序相反）存储了分配的 CPlex 块。

揭开 CPlex 的神秘面纱

CPlex 的神秘之处在于它有 nMax 和 nCur 两个成员变量。这两个变量只在 Create()中出现，在 MFC 其他源代码中都没有出现过。我们就这个问题请教了 MFC 的权威人士 Dean McCrory，他回答说："这两个变量看起来没什么用，我认为惟一的原因要追溯到'old AFX'——在 MFC 之前试图创建的框架（由同一 MFC 小组创建）。"

在我们向 Dean McCrory 提出这两个没用的成员变量之后，他认为应该把这两个变量从 MFC 中去掉！所以在 MFC 4.0 中你会在 AFX.H 中看到以下代码：

```
#define CPlex CPlexNew
```

而且在文件 PLEX.CPP 的头部增加了如下注释：

```
// CPlexNew - CPlex without nCur and nMax members
```

这样的处理方法（重定义 CPlex 类的名字）可以保证不破坏旧的 MFC 应用程序。

为什么要去掉这两个变量呢？首先，CPlex 是链表和映射表的关键内存机制，每个 CPlex 都分配了两个无用的整数。因为 MFC 内部还使用了链表和映射表，所以你的 MFC 应用程序中就有成千（或更多）的 CPlex 块浪费了两个整数。

考虑一下节省下来的内存，MFC 的内幕就是：改变 MFC 编程，让它变得更好。

释放 CPlex

在浏览 LIST_S.CPP 文件时，你会发现链表中的节点从来不真正释放。如果元素从链表中删除了，MFC 只是把它放到空闲链表中，而并不释放。内存的释放只在 CStringList::RemoveAll()中完成。下面我们看这个成员函数是如何释放元素的。程序清单 4-24 显示了所有伪代码。

程序清单 4-24 CStringList::RemoveAll()：MFC 如何从链表中释放所有节点

```
void CStringList::RemoveAll()
{
    CNode* pNode;
    for (pNode = m_pNodeHead; pNode != NULL; pNode = pNode->pNext)
        DestructElement(&pNode->data);
    m_nCount = 0;
    m_pNodeHead = m_pNodeTail = m_pNodeFree = NULL;
    m_pBlocks->FreeDataChain();
    m_pBlocks = NULL;
}
```

首先，RemoveAll()遍历了链表中的所有元素，并使元素释放自己（注意：DestructElement()也是一个有用的函数，它的定义在文件 ELEMENTS.H 中）。然后，RemoveAll()清除了计数变量 m_nCount 和所有指针，最后又调用了 CPlex::FreeDataChain()成员函数，并将已分配内存的 CPlex 的链表指针 m_pBlocks 赋值为空。

在 RemoveAll()中释放内存的位置还不是很明显，所以我们先看一下 FreeDataChain()成员函数。程序清单 4-25 列出了 CPlex::FreeDataChain()成员函数，这个函数在文件 PLEX.CPP 中。

程序清单 4-25 CPlex::FreeDataChain()：所有分配的 CPlex 内存块被释放的位置

```
void CPlex::FreeDataChain()
{
    CPlex* p = this;
    while (p != NULL){
        BYTE* bytes = (BYTE*) p;
        CPlex* pNext = p->pNext;
        delete[] bytes;
        p = pNext;
    }
}
```

FreeDataChain()遍历分配的内存块 CPlex 链表，并且删除链表中的每个 CPlex 节点。你会发现 FreeDataChain()在删除节点之前都会先获取下一个节点，这样它才能沿着链表继续向下删除。

CList 内存管理总结

记住 CList 内存管理的关键之一是要知道 MFC 为 CList 维护了 3 个链表：

1. 一个实际使用的链表。它是 CNode 的双链表，头指针是 m_pNodeHead，尾指针是 m_pNodeTail。

2. 一个空闲节点的链表，它是单链表，头指针是 m_pNodeFree。

3. 所有分配的内存块链表，或者说是 CPlex 链表，头指针是 m_pBlocks。

另一个关于 MFC 链表的关键点是在你删除元素时，内存并没有释放，而是在空闲链表中等待重复利用。这样如果你做了很多增加和删除节点的操作，那么你的链表就会增长得很大。RemoveAll()函数会利用 CPlex::FreeDataChain()彻底释放所有的空间。

现在你知道 MFC 是如何为链表进行内存管理的了。如果你的数据结构也是基于节点的，你就可以使用这种方法来解决问题了。而且，使用这种方法不必分配/释放很细小的空间（例如一个 CNode 与一个 CPlex），否则代价很高。

关于 MFC 链表中的位置

回顾一下如何使用 MFC 链表，你可能会注意到一个含糊的类型——POSITION。POSITION 类型的定义在文件 AFX.H 中：

```
// abstract iteration position
struct __POSITION { int unused; };
typedef __POSITION* POSITION;
```

POSITION 和 void 指针很相似，它实际上是指向一个 CNode 节点的指针。但是因为实现是可以改变的，所以它依赖于类的使用者。

另一个没有解决的难题！？

为什么把 POSITION 定义成一个指向那样一个结构（只包含了一个无用的整数）的指针？这实在很不清楚。在 MFC 4.0 以前的版本中，POSITION 实际上是一个 void 指针。

我们向 MFC 的权威人士 Dean McCrory 请教了这个问题，他的回答是：

为了类型安全。我们发现对于 POSITION 参数，人们很容易传错，尤其当链表和映射表中的元素是 void* 时。因为 POSITION 的类型和集合元素的类型是一样的，所以你可能在不需要 POSITION 的地方传入了一个 POSITION，而在需要 POSITION 的地方传入了一个 void*。使用上面的技术（类似于 Win31 的 WINDOWS.H 中增加的 STRICT）使编译器能够进行类型检查。

为了证明 POSITION 是指向 CNode 的指针，下面是内嵌文件 AFXCOLL.INL 中的 GetHeadPosition()的实现：

```
POSITION CPtrList::GetHeadPosition() const
{
    return (POSITION) m_pNodeHead;
}
```

AFXCOLL.INL 文件中还有 CStringList::GetNext()成员函数：

```
CString& CStringList::GetNext(POSITION& rPosition)
{
    CNode* pNode = (CNode*) rPosition;
    rPosition = (POSITION) pNode->pNext;
    return pNode->data;
}
```

更多信息

CStringList 还有其他进行插入、删除和查找元素的成员函数，这些操作都是直接对链表进行的。现在你已经知道 POSITION 如何使用，你也就可以检查一下这些函数。你会发现你刚刚看到的 NewNode()函数和 FreeNode()函数有多么重要！

MFC 映射表集合类

本节，我们将看 MFC 如何实现映射表集合类。但我们首先回顾一下使用映射表的基本点。

映射表（map），有时也称字典（dictionary），存储了成对的信息，每个对有一个键（key）和一个值（value）。键是用来查找值的。MFC 使用散列表来存储键，因此，在散列表中查找内容比在链表中查找内容快。

MFC 把这个键/值对称为关系（association）。程序清单 4-26 显示了一些使用 CMapStringToString 映射表集合类的代码。

程序清单 4-26 使用 CMapStringToString 的例子

```
//...
CMapStringToString myStringMap;

//Let's create a spanish dictionary!
myStringMap.SetAt("one", "uno");
myStringMap.SetAt("two", "dos");
myStringMap.SetAt("three","tres");
//... sometime later, the user wants to convert 'one' to spanish..
CString strAnswer;
if (myStringMap.LookUp("one",answer))
    // got it, should be uno.
else
    // oops, wasn't in there?!

// Iterate through the map
POSITION pos = GetStartPosition();
while (pos != NULL){
```

```
    CString key, value;
    GetNextAssoc(pos, key,value);
    //Down here, key and value have been
    //Updated to hold the next association
}
```

除了 CMap 例子中给出的操作，还有其他很多操作，例如删除关系、得到关系的数量和执行其他集合操作。

CMap 的实现

下面我们看 MFC 如何实现键的类型和值的类型不同的映射表，例如 CMapWordToPtr。

CMapWordToPtr 的声明在文件 AFXCOLL.H 中，它的实现在文件 MAP_WP.CPP 中。CMap 的源文件的命名规则是 MAP_XY.CPP，其中 X 是映射表中的键，Y 是映射表中的值。在我们举的例子中，X（键）是 WORD，Y（值）是 PTR（void*）。

程序清单 4-27 给出了 CMapWordToPtr 声明的缩写版本，在文件 AFXCOLL.H 中。

程序清单 4-27 CMapWordToPtr 的声明（在文件 AFXCOLL.H 中）

```
class CMapWordToPtr : public CObject
{
DECLARE_DYNAMIC(CMapWordToPtr) protected:
// Association
    struct CAssoc{
        CAssoc* pNext;
        UINT nHashValue;  // needed for efficient iteration WORD key;
        void* value;
    };
public:
// Construction **omitted
// Attributes **omitted
// Operations **some omitted
// advanced features for derived classes
    UINT GetHashTableSize() const;
    void InitHashTable(UINT hashSize, BOOL bAllocNow = TRUE);

    UINT HashKey(WORD key) const;
// Implementation
protected:
    CAssoc** m_pHashTable;
    UINT m_nHashTableSize;
    int m_nCount;
    CAssoc* m_pFreeList;
    struct CPlex* m_pBlocks;
    int m_nBlockSize;

    CAssoc* NewAssoc();
    void FreeAssoc(CAssoc*);
    CAssoc* GetAssocAt(WORD, UINT&) const;

public:
    ~CMapWordToPtr();
};
```

如果查看 MFC 对 CMap 类的文档说明，你会发现程序清单 4-27 中有很多成员函数和成员数据都是没有文档说明的。这意味着我们有更多的东西需要探索。

首先，和 CList::CNode 很类似，所有的映射表集合类中都有一个嵌入的 CAssoc 结构的声明，它描述了映射表中的元素。CAssoc 结构中有键和值。CAssoc 看起来还通过 pNext 指针链接成链表。我们后面还会讨论 nHashValue。

类似的情况

映射表的一些没有文档说明的成员函数和我们看到的 MFC 链表的一些成员函数很相似。映射表和链表很相似，它们使用同样的 CPlex 内存辅助结构和算法来管理内存。所以，我们完全可以猜测出这些变量的作用：

- m_nCount——映射表中元素的个数。
- m_pBlocks——一个指向已分配的 CPlex 块链表的头节点的指针。
- m_pFreeList——一个指向从 CPlex 分配出来的空闲 CAssoc 链表的指针。
- m_nBlockSize——分配的 CPlex 块的大小。

因为我们已经仔细研究过了 CPlex 类，所以我们现在直接跳到映射表中那些没有文档说明的部分。

MFC 映射表中的散列

如果你不熟悉散列，我们会先介绍一下。散列主要用于提高查找速度。用一个散列函数作用于键，会得到数据应该存储在散列表中的位置（索引）。如果要获得值，你要再用散列函数作用于键，从而得到索引，这样不需要检索就能查到数据。

MFC 映射表的散列是散列应用的一个典型例子。首先，我们先看一个散列表，以及何时、如何分配散列表。然后再看键是如何散列到散列表中的。最后再看如何从散列表中取到值。

所有 MFC 映射表都通过调用 InitHashTable()来分配内部的散列表。程序清单 4-28 给出了 InitHashTable()成员函数的伪代码，在 MAP_WP.CPP 中。

程序清单 4-28 映射表的散列表的初始化在 InitHashTable()中完成

```
void CMapWordToPtr::InitHashTable(UINT nHashSize, BOOL bAllocNow)
{
    if (m_pHashTable != NULL) {
        // free hash table
        delete[] m_pHashTable;
        m_pHashTable = NULL;
    }

    if (bAllocNow) {
        m_pHashTable = new CAssoc* [nHashSize];
        memset(m_pHashTable, 0, sizeof(CAssoc*) * nHashSize);
    }
    m_nHashTableSize = nHashSize;
}
```

首先，InitHashTable()函数检查 m_pHashTable 是否已经指向了一个散列表。如果是，它会先删除旧的散列表，以便为新的散列表让出空间。然后如果 bAllcoNow 标志是 TRUE，InitHashTable()会分配一个 CAssoc 指针数组，并将 m_pHashTable 指向它，再用 memset 清除

这个数组。

最后，InitHashTable()更新了成员变量 m_nHashTableSize，以免值超出散列数组的大小。

注意：了解 InitHashTable()很有帮助。如果你已经知道映射表大小的上界，例如 1017，就可以像下面这样初始化散列表：

```
myMap.InitHashTable(1017,TRUE);
```

这样就可以避免后来再进行内存分配，也可以管理自己的映射表。但是要注意，对一个完成的映射表调用这个函数会删除映射表中的所有内容，所以你要在恰当的地方调用。

散列表的默认大小是 17。散列表的大小通常是质数，因为对质数做取模操作会使值分布得更好。

散列表就是 CAssoc 指针数组。下一个问题是如何使关系散列到表中。HashKey()函数用来对键散列。MFC 映射表中有两个 HashKey()函数，一个是字和指针的散列函数，另一个是字符串的散列函数。程序清单 4-29 给出了两个 HashKey()函数。

程序清单 4-29 CMap 的散列函数

```
inline UINT CMapWordToPtr::HashKey(WORD key) const
{
    return ((UINT)(void*)(DWORD)key) >> 4;
}
inline UINT CMapStringToString::HashKey(LPCTSTR key) const
{
    UINT nHash = 0;
    while (*key)
        nHash = (nHash<<5) + nHash + *key++;
    return nHash;
}
```

第一个 HashKey()函数用来散列 WORD 和 void 指针。它将散列值向右移动 4 位，从而增加了高位的重要性。

第二个 HashKey()函数用来散列字符串。它遍历字符串，对字符串中的每个字符相加得到 nHash。每次将 nHash 向左移动 5 位，从而增加了字符串中那些独特位的重要性，这样就使得分布更均匀。

这两个函数的目的就是使 WORD、指针或字符串在散列表中的分布更均匀。不过非模板的 MFC 的映射表不允许你提供自己的 HashKey()函数，但是你可以用模板映射表类来创建自己的散列函数。

碰撞（collision）

散列表的一个问题就是解决碰撞，也就是解决两个值映射到散列表中同一个索引的情况。

MFC 通过在链表中增加碰撞项来解决碰撞。回头看看程序清单 4-27，你会发现实际上 CAssoc 结构已经包含了一个 pNext 指针。CMap 就用这个指针链接那些冲突的关系。

为了看清楚 MFC 映射表如何散列，我们先看如何在映射表中增加节点。所有的插入操作都要用到操作符[]。SetAt()成员函数就是直接利用它插入新值的。下面是内嵌函数 SetAt()的具体实现（在 AFXCOLL.INL 文件中）：

```
void CMapWordToPtr::SetAt(WORD key, void* newValue)
{
    (*this)[key] = newValue;
}
```

SetAt()调用[]操作符对 key 操作，然后返回结果值。为了理解下面的内容，我们先看CMapWordToPtr::operator[]。程序清单 4-30 给出了这个操作符的伪代码。

程序清单 4-30 CMapWordToPtr::operator[]：在映射表中增加关系的“键”

```
void*& CMapWordToPtr::operator[](WORD key)
{
    UINT nHash;
    CAssoc* pAssoc;
    if ((pAssoc = GetAssocAt(key, nHash)) == NULL) {
        if (m_pHashTable == NULL)
            InitHashTable(m_nHasTableSize);
        // it doesn't exist, add a new Association
        pAssoc = NewAssoc();
        pAssoc->nHashValue = nHash;
        pAssoc->key = key;
        // 'pAssoc->value' is a constructed object, nothing more
        // put into hash table
        pAssoc->pNext = m_pHashTable[nHash];
        m_pHashTable[nHash] = pAssoc;
    }
    return pAssoc->value;  // return new reference
}
```

操作符[]扮演着双重角色，它既是散列表的存取器也是它的存储辅助结构。从程序清单4-30 中不能很清楚地看出散列过程在哪里发生。实际上这一过程是在 GetAssocAt()函数中完成的。下面我们会看 GetAssocAt()函数，但我们首先分析操作符[]都做了些什么。调用GetAssocAt()既返回了 nHash 的值（散列的索引值）又检查了这个入口在散列表中是否出现过。如果关系已经在散列表中，“if”部分就不执行了，操作符就会返回得到的值。

如果散列表中没有这个关系，操作符首先会检查散列表，确保散列表已经创建而且初始化过。然后它创建一个新的关系（CMap::NewAssoc()和 CList::NewNode()一样），并且存储散列索引和键。此时，你会发现还没有设置值。

最后，操作符[]将新的关系插入到散列表中，其索引就是 nHash，并返回对值指针的引用。值得注意的是“if”分支的最后两个语句解决了碰撞问题。

操作符返回一个对值的引用，并且 SetAt()成员函数存储值。

CMapWordToPtr::GetAssocAt()：散列的瓶颈

为了更清楚地了解操作符[]，我们需要分析 GetAssocAt()函数。它负责在散列表中定位入口（entry）。程序清单 4-31 给出了没有文档说明 GetAssocAt()函数。

程序清单 4-31 CMapWordToPtr::GetAssocAt()：一个没有文档说明的映射表成员函数，它负责为查找和存储操作完成散列过程

```
CMapWordToPtr::CAssoc* CMapWordToPtr::GetAssocAt(WORD key, UINT& nHash)
const
```

```
{
    nHash = HashKey(key) % m_nHashTableSize;
    if (m_pHashTable == NULL)
        return NULL;
    // see if it exists
    CAssoc* pAssoc;
    for (pAssoc = m_pHashTable[nHash]; pAssoc != NULL; pAssoc =
            pAssoc->pNext) {
        if (pAssoc->key == key)
            return pAssoc;
    }
    return NULL;
}
```

在我们分析它的源代码之前，要记住如果键在散列表中不存在，GetAssocAt()的工作是返回 NULL 和一个新的散列索引。如果键在散列表中，就应该返回这个键所对应的关系，当然也需要通过参数 nHash 返回散列索引。

在程序清单 4-31 中，GetAssocAt()的第一行是 MFC 映射表的散列表的核心。GetAssocAt()首先调用 HashKey()来运行散列函数，并将得到的值 Mod 散列表的大小，从而得到新的散列索引。所有对散列表的访问（除了删除键以外）都使用这个函数来散列键。

一旦计算出新的散列索引，GetAssocAt()会验证散列表是否已经初始化过，然后获得散列索引对应的关系。GetAssocAt()然后遍历对应这个散列索引的链表，看是否能找到键。如果找到了键，就返回这个关系，否则返回 NULL。

MFC 映射表总结

关于 MFC 映射表，我们已经学习了很多有趣的内容：

- 它们和 MFC 链表使用了相同的 CPlex 内存管理技术。
- 内部使用散列表加快了查找关系的速度。
- 可以指定散列表的大小，从而在应用程序中提高映射表的性能。

更多信息

学习了以上内容之后你可能会问一个问题：为什么关系中要存储散列键呢？看起来任何访问关系的代码都会知道键，所以说关系中存储了复制的信息。答案就在 CMapWordToPtr::GetNextAssoc()函数里。另一个有趣的成员函数是 RemoveKey()。

以上就是我们对 MFC 集合类的介绍。如果你不想仅限于此，可以看看模板集合类，这些内容在 AFXTEMP1.H 文件中。

CFile 家族：MFC 对文件的访问

为了让 MFC 程序员更容易访问文件，MFC 提供了一套有层次的实用类。这些 MFC 文件类层次结构中的根节点是 CFile 类，它实现了一些基本的非缓冲的文件访问，而且只是 Windows 文件 API 的一个简单包装。CStdioFile 提高了 CFile 的性能，它提供了带缓冲的 I/O。因为 CStdioFile 是带缓冲的，所以读写非二进制的 ASCII 文件就变得很容易：你可以在缓冲

区中向前读到一整行。

CMemFile 类还为通过 CFile 接口实现共享内存提供了基类。不幸的是，CMemFile 没有提供所有的功能，但它是个开始。

另外还有一个没有文档说明的 CFile 的派生类——CSharedFile，本节后面会介绍这个类。它在 CMemFile 的基础上又增加了一些关于内存共享的内容。如果你想将一些数据序列化到内存中，然后再保存，就可以使用 CMemFile。

使用 CFile

在创建一个 CFile 对象时，需要指定文件的名字和文件的打开模式。可以在它的构造函数中指出这些内容，也可以在调用 CFile::Open()时指定。这里的模式就是访问模式和共享模式的组合，通过 OR 将多个模式连接在一起。表 4-2 给出了各种可以使用的 CFile 模式。

表 4-2 CFile 的访问和共享标志

模式	描述
modeCreate	创建文件，如果文件已经存在就将文件大小设置为 0
modeNoTruncate	创建文件，但是并不截去文件
modeRead	打开文件，为了进行读操作
modeReadWrite	打开文件，为了进行读操作和写操作
modeWrite	打开文件，为了进行写操作
modeNoInherit	防止子进程继承文件
shareDenyNone	打开文件，但不阻止其他进程读或写文件
shareDenyRead	打开文件，但阻止其他进程读文件
shareDenyWrite	打开文件，但阻止其他进程写文件
shareExclusive	打开文件，但阻止其他进程读文件和写文件
shareCompat	以兼容模式打开文件，允许其他进程打开任意次
typeText	设置文本模式，对回车换行对进行特殊处理（只在 CStdioFile 中使用）
typeBinary	设置二进制模式（只在 CStdioFile 中使用）

CFile 文件中定义了这些模式，所以你只需要指定 CFile 作用域就能访问它们，例如 CFile::shareDenyRead、CFile::typeText 等等。

下面这行代码说明了如何以创建和写模式打开一个 CFile：

```
CFile myFile ("myfile.sct", CFile::modeCreate | CFile::modeWrite);
```

CFile 有很多成员函数，你可以用这些函数读文件、写文件、锁定文件、定位文件、重命名文件、删除文件以及获取文件的状态。

CFile 内幕

因为使用 CFile 和使用标准的 Windows 文件 API 一样，所以我们直接分析 CFile 是如何

实现的，然后再看它的一些特殊的派生类。

和大多数 MFC 实用类一样，CFile 的声明也在 AFX.H 文件中。它的实现在两个文件里：FILECORE.CPP 和 FILEST.CPP。前者包含了除了获得文件状态之外的所有成员函数，而后者只包含了获得文件状态的成员函数。

CFile 的声明

程序清单 4-32 给出了 CFile 声明的缩减版本，而且主要列出了那些没有文档说明的成员。

程序清单 4-32 缩减版本的 CFile 的声明

```
class CFile : public CObject
{
DECLARE_DYNAMIC(CFile)
public:
// Flag values **omitted
// Constructors  **omitted
// Attributes  **mostly omitted
    UINT m_hFile;
// Operations  **omitted
// Overridables  **omitted
// Implementation  **omitted

public:
    virtual ~CFile();
    enum BufferCommand { bufferRead, bufferWrite, bufferCommit,
         bufferCheck };
    virtual UINT GetBufferPtr(UINT nCommand, UINT nCount = 0,
            void** ppBufStart = NULL, void** ppBufMax = NULL);
protected:
    BOOL m_bCloseOnDelete;
    CString m_strFileName;
};
```

CFile 中第一个有趣的声明是 public 变量 m_hFile，它存储了 Windows 文件的句柄。CFile 提供这样一个成员变量的目的是使 CFile 用户可以在直接进行 Windows 文件 API 调用时直接获得文件句柄。例如，CFile 没有提供任何 Windows NT 文件安全方面的 API，但你可以用 CFile::m_hFile 作为文件句柄调用它们。

另一些有趣的成员是 BufferCommand 枚举和 GetBufferPtr()成员函数。在 CFile 里，这些成员什么都不做，只是 CMemFile 继承类会提供 GetBufferPtr()函数的实现。MFC 序列化机制利用这个函数来优化 CMemFile 的序列化。

m_bCloseOnDelete 成员指定了在 CFile 被破坏后是否关闭文件句柄。m_strFileName 成员用来存储当前文件的文件名。

CFile 操作

CFile 的大多数成员函数都是直接调用 Windows API，所以其中没有太多关于成员变量的源代码。表 4-3 列出了各种 CFile 成员函数和它们所基于的 Windows API。

表 4-3 CFile 成员函数和 Windows 文件 API 之间的映射

CFile 成员函数	封装的 Windows 文件 API
Open()	::Createfile()
Duplicate()	::DuplicateHandle()
Read()	::ReadFile()
Write()	::WriteFile()
Seek()	::SetFilePointer()
GetPosition()	::SetFilePointer()
Flush()	::FlushFileBuffers()
Close()	::CloseHandle()
LockRange()	::LockFile()
UnlockRange()	::UnlockFile()
SetLength()	::SetEndOfFile()
Rename()	::MoveFile()
Remove	::DeleteFile()
GetStatus()	::GetFileTime(), ::GetFileSize()

有趣的没有文档说明的辅助函数

看一下 FILECORE.CPP 文件，你会发现有一些专门辅助管理文件名的全局辅助函数。了解这些函数很有好处，因为这些函数可以使你在管理文件名时节省时间。

- AfxResolveShortcut()——查找一个 Windows 95 的快捷键，并将它转化成全路径名。
- AfxFullPath——将一个文件路径转化成绝对路径。
- AfxGetRoot——解析一个 UNC（Uniform Nameing Convention，统一命名标准）路径或一个旧式路径，得到卷标名。
- AfxComparePath()——比较两个路径，判断它们是否一样。而且它为你处理所有的 DBCS（Double Byte CharacterSet，双字节字符集）/UNICODE 问题。
- AfxGetFileTitle()——从路径中解析出文件名。

如果你对使用这些成员函数有兴趣，应该将它们的代码从 FILECORE.CPP 文件中拷贝到自己的函数里，因为这些没有文档说明的函数可能会随着 MFC 版本的更新而被修改或删除。

因为 CFile 是文件处理的一个简单包装层，所以它的内部不是很复杂。CStdioFile 相对而言就更有趣一些，因为它在 CFile 中增加了缓冲。下面就让我们来看一下 CFile 的这个派生类，看它如何支持文件 I/O 的缓冲。

CStdio File

使用 CStdioFile 类和使用 CFile 类很类似，只不过它增加了两个函数：ReadString()和 WriteString()。在 CStdioFile 以文本模式打开时用这两个成员函数读写字符串。

ReadString()会一直读文件，直至以下情况之一满足：

1. ReadString()读到指定数目的字符。
2. ReadString()读到一个回车/换行符。

3. ReadString()读到了文件的末尾。

下面的代码段显示了如何使用 CStdioFile 对文本文件中的行数计数：

```
int nLineCount = 0;
char buffer[256];
CStdioFile myFile("countme.txt", CFile::typeText |CFile::modeRead);
while (csfTipFile.ReadString(buffer,256) != 0)
    nLineCount++;
```

CStdioFile 不支持 CFile 中的一些函数，因为它使用了缓冲。不支持的 CFile 成员函数包括 Duplicate()、LockRange()和 UnlockRange()。

CStdioFile 内幕

从外部看，CStdioFile 看起来在 CFile 中增加了一些函数，而在内部 CStdioFile 只是增加了额外的逻辑来支持文件缓冲。CFile 封装了 Windows 文件 API，而 CStdioFile 封装了 C 运行时库（C run-time library）的流文件函数，例如 fopen、fread、fputs、fgets 和 fseek，从而使 CStdioFile 能够进行文件缓冲和文本操作。为了使用这些函数，CStdioFile 除了有一个从 CFile 继承的文件句柄外，不得不再维护一个 FILE 指针或流指针。下面我们看 CStdioFile 的声明以及它如何存储这些额外信息。

CStdioFile 的声明

和 CFile 一样，CStdioFile 的声明也在文件 AFX.H 中，它的实现在 FILETXT.CPP 中。程序清单 4-33 给出了 CStdioFile 声明的缩写版本。

程序清单 4-33　CStdioFile 声明的缩写版本

```
class CStdioFile : public CFile
{
DECLARE_DYNAMIC(CStdioFile)
public:
// Constructors **omitted
// Attributes
    FILE* m_pStream;     // stdio FILE
// m_hFile from base class is _fileno(m_pStream)
// Operations
    virtual void WriteString(LPCTSTR lpsz);
    virtual LPTSTR ReadString(LPTSTR lpsz, UINT nMax);
    BOOL ReadString(CString& rString);
// Implementation
public:
    virtual ~CStdioFile();
    virtual DWORD GetPosition() const;
    virtual BOOL Open(LPCTSTR lpszFileName, UINT nOpenFlags,
            CFileException* pError = NULL);
    virtual UINT Read(void* lpBuf, UINT nCount);
    virtual void Write(const void* lpBuf, UINT nCount);
    virtual LONG Seek(LONG lOff, UINT nFrom);
    virtual void Abort();
    virtual void Flush();
    virtual void Close();
// Unsupported APIs
    virtual CFile* Duplicate() const;
```

```
    virtual void LockRange(DWORD dwPos, DWORD dwCount);
    virtual void UnlockRange(DWORD dwPos, DWORD dwCount);
};
```

从 CStdioFile 的声明可以看出，它在 m_pStream 数据成员中保存了一个 FILE 指针，而且其中还包括了 WriteString()和 ReadString()这两个新成员函数。你还可以看到它有几个成员函数封装了 C 运行时流文件调用。

ReadString()

为了理解 CStdioFile 如何运作，我们首先看 CStdioFile::ReadString()的伪代码。这些代码在 STRTXT.CPP 中，如程序清单 4-34 所示。

程序清单 4-34 CStdioFile::ReadString()的实现（在文件 STRTXT.CPP 中）

```
LPTSTR CStdioFile::ReadString(LPTSTR lpsz, UINT nMax)
{
    LPTSTR lpszResult = _fgetts(lpsz, nMax, m_pStream);
    if (lpszResult == NULL && !feof(m_pStream))    {
        clearerr(m_pStream);
        AfxThrowFileException(CFileException::generic, _doserrno,
            m_strFileName);
    }
    return lpszResult;
}
```

从它的实现，你会发现大多数 CStdioFile 操作都封装了 C 运行时调用（例如_fgetts），然后执行了一些额外的错误检查。

表 4-4 显示了 CStdioFile 成员函数与 C 运行时函数之间的映射。

表 4-4 CStdioFile 成员函数与 C 运行时文件流调用之间的映射

CStdioFile 成员	封装的 C 运行时 API
Open()	_fdopen()
Read()	fread()
Write()	fwrite()
ReadString()	_fgetts()
WriteString()	_fputts()
Seek()	fseek()
GetPosition()	ftell()
Flush()	fflush()
Close()	fclose()

到目前为止，CFile 和 CStdioFile 没有提供任何有趣的 MFC 内幕。下面让我们分析 CMemFile 和它的派生类，还有没有文档说明的 CSharedFile 类。

CMemFile

因为 CMemFile 为一块内存提供了 CFile 接口，所以它不用保存文件名和 m_hFile 文件句柄。

CMemFile——一个真实的故事

许多 MFC 用户认为使用 CMemFile 可以通过 Win32 内存映射文件和其他共享内存机制来访问共享内存，但这是不对的。CMemFile 使用的内存是标准内存而不是共享内存。你可以从 CMemFile 派生一个类，再增加共享内存。

MFC 提供 CMemFile 的一个主要原因是可以将一块内存序列化(serialization)(关于 MFC 序列化的更多信息，请见第 5 章)。例如，你可以使用 CArchive/CMemFile 来产生对象的内存映像，然后再将内存映像作为二进制数据写入注册表。Visual C++常用这种技术。

在序列化中使用 CMemFile 是面向对象多态性的一个很好的例子。因为序列化的逻辑就是使用 CFile 接口，所以只要表现的行为相同，CMemFile 不管写入的是文件还是一块内存。

CMemFile 自动分配内存、扩大内存以及释放内存。你还可以自己分配一块内存，然后调用 Attach()将 CMemFile 附加在它后面。

CMemFile 用 malloc()、realloc()、memcpy()和 free()等内存函数来管理内部内存块，你还可以通过 CMemFile 的构造函数中的 nGrowBytes 参数来指定内存每次增长的大小。你还可以重载 Alloc()、Free()、Realloc()、Memcpy()和 GrowFile()等函数，自己管理内存。默认情况下，这些函数会调用相应的 C 运行时内存函数。

对于内存块来说，CFile 的许多函数都不起作用，所以 CMemFile 对 CFile 的一些函数不支持。这些函数包括 Duplicate()、LockRange()和 UnlockRange()。

下面我们看一下 CMemFile 为内存块所提供的 CFile 接口。

CMemFile 内幕

CMemFile 与 CFile 和 CStdioFile 不一样，它有非常出色的内部细节，而且它将一个文件接口映射到一块内存上，这的确是一件很麻烦的事。

CMemFile 的声明也在文件 AFX.H 中，它的实现在文件 FILEMEM.CPP 中，CMemFile::GetStatus()成员函数在文件 FILEST.CPP 中。

CMemFile 的声明

程序清单 4-35 简略地给出了 CMemFile 的声明。

程序清单 4-35 CMemFile 声明的缩略版本（在文件 AFX.H 中）

```
class CMemFile : public CFile
{
DECLARE_DYNAMIC(CMemFile) public:
// Constructors
    CMemFile(UINT nGrowBytes = 1024);
    CMemFile(BYTE* lpBuffer, UINT nBufferSize, UINT nGrowBytes = 0);
// Operations
    void Attach(BYTE* lpBuffer, UINT nBufferSize, UINT nGrowBytes = 0);
    BYTE* Detach();
// Advanced Overridables **omitted
protected:

// Implementation
protected:
    UINT  m_nGrowBytes;
```

```
    DWORD m_nPosition;
    DWORD m_nBufferSize;
    DWORD m_nFileSize;
    BYTE* m_lpBuffer;
    BOOL  m_bAutoDelete;

public:
    virtual ~CMemFile();
    virtual DWORD GetPosition() const;
    BOOL GetStatus(CFileStatus& rStatus) const;
    virtual LONG Seek(LONG lOff, UINT nFrom);
    virtual void SetLength(DWORD dwNewLen);
    virtual UINT Read(void* lpBuf, UINT nCount);
    virtual void Write(const void* lpBuf, UINT nCount);
    virtual void Abort();
    virtual void Flush();
    virtual void Close();
    virtual UINT GetBufferPtr(UINT nCommand, UINT nCount = 0,
                void** ppBufStart = NULL, void** ppBufMax = NULL);
// Unsupported APIs **omitted
};
```

CMemFile 增加了几个新的成员变量：

- m_nGrowBytes——指明在必须再分配新的内存时，CMemFile 内部缓冲区增长的大小。
- m_nPosition——指明内存中的模拟文件指针的位置。它由 Seek()等成员函数控制，指出读/写操作在内存块中的位置。
- m_nBufferSize——实际分配的缓冲区的大小。
- m_nFileSize——指明内存文件的逻辑大小，它可以比缓冲区小，因为内存每次分配的大小是 m_nGrowBytes。
- m_nlpBuffer——表示指向内存文件的实际内存的指针。
- m_bAutoDelete——指明如果调用了析构函数，内存是否应该释放。

CMemFile 的内存管理

CMemFile 的内存管理和 CArray 类似，都是在需要时才管理和增加内存块。所有的 CMemFile 成员函数都会首先检查操作是否在当前内存块的界内，如果不在界内，它就会调用 CMemFile::GrowFile()成员函数来分配新的内存或重新分配足够的内存。

程序清单 4-36 给出了 CMemFile::GrowFile()成员函数的伪代码，在文件 FILEMEM.CPP 中。

程序清单 4-36 CMemFile::GrowFile()的伪代码（在文件 FILEMEM.CPP 中）

```
void CMemFile::GrowFile(DWORD dwNewLen)
{
    if (dwNewLen > m_nBufferSize)  {
        // grow the buffer
        DWORD dwNewBufferSize = (DWORD)m_nBufferSize;
        // determine new buffer size
        while (dwNewBufferSize < dwNewLen)
            dwNewBufferSize += m_nGrowBytes;
        // allocate new buffer
        BYTE* lpNew;
```

```
            if (m_lpBuffer == NULL)
                lpNew = Alloc(dwNewBufferSize);
            else
                lpNew = Realloc(m_lpBuffer, dwNewBufferSize);
            m_lpBuffer = lpNew;
            m_nBufferSize = dwNewBufferSize;
        }
    }
```

GrowFile()只带一个参数，这个参数指定了缓冲区的新长度，如果需要分配内存，GrowFile()会计算新的内存大小（应该能被 m_nGrowBytes 整除）。如果还没有分配缓冲区，GrowFile()会调用 Alloc()。如果缓冲区已经存在，它会调用 ReAlloc()，参数是原缓冲区的指针和新的大小。最后，所有的数据成员都随着新增加的内存块而更新。

访问内存块

为了分析访问内存块都涉及到哪些内容，我们首先看 CMemFile::Read()的实现。程序清单 4-37 给出了这个成员函数的伪代码。

程序清单 4-37 CMemFile::Read()：访问内存文件

```
UINT CMemFile::Read(void* lpBuf, UINT nCount)
{
    if (nCount == 0)
        return 0;
    if (m_nPosition > m_nFileSize)
        return 0;
    UINT nRead;
    if (m_nPosition + nCount > m_nFileSize)
        nRead = (UINT)(m_nFileSize - m_nPosition);
    else
        nRead = nCount;
    memcpy((BYTE*)lpBuf, (BYTE*)m_lpBuffer + m_nPosition, nRead);
    m_nPosition += nRead;
    return nRead;
}
```

首先，Read()检查要读的大小是否为 0，以及内部指针的位置是否正确（例如，超出了文件大小）。

然后，Read()计算出要读的字节数，以确保这个字节数加上当前位置不会超过内存块的大小（否则就要读到文件末尾之后）。如果字节数加上当前位置超出了文件的大小，Read()就会减少所要读的字节数，以免读内存块时越界。

最后，Read()调用 Memcpy()将字节从内存缓冲区拷贝到目的内存缓冲区中，然后返回实际读的字节数。

CMemFile::Read()反映了 CMemFile 其他函数是如何实现的。从外部看来，它提供了一个类似文件的接口，而就其内部而言，它只是对一块内存进行操作。

高级主题：CFile::GetBufferPtr()与 MFC 序列化

CMemFile 提供了一个名为 GetBufferPtr()的函数，这个函数仅由 CArchive 类使用。CArchive 将在下一章讨论。如果有必要，CArchive 会提供自己的缓冲方法，但它首先会调用 CFile::GetBufferPtr()来看文件是否支持缓冲。如果 CFile 支持缓冲，CArchive 就使用这种对缓

冲的支持，而不使用自己的。

CArchive 通过指定一个 BufferCommand 枚举类型来与文件通信，BufferCommand 中每个枚举含义如下：

- bufferCheck——CArchive 询问 CFile 是否支持缓冲技术，CFile 和 CStdioFile 回答否，CMemFile 回答是。
- bufferRead——CArchive 要从缓冲区中读取数据，要指定起始位置、大小和最大值。
- bufferWrite——CArchive 要向缓冲区中写数据。
- bufferCommit——CArchive 要向缓冲区提交修改。

记住这些缓冲命令，下面我们来看 CMemFile::GetBufferPtr() 的实现（在文件 FILEMEM.CPP 中），见程序清单 4-38，我们来看 CMemFile 和 CArchive 是如何共同工作的。

程序清单 4-38 CMemFile::GetBufferPtr()成员函数（在文件 FILEMEM.CPP 中）

```
UINT CMemFile::GetBufferPtr(UINT nCommand, UINT nCount, void** ppBufStart,
void**ppBufMax)
{
    if (nCommand == bufferCheck)
        return 1;
    if (nCommand == bufferCommit)    {
        // commit buffer
        m_nPosition += nCount;
        if (m_nPosition > m_nFileSize)
        m_nFileSize = m_nPosition; return 0;
    }

    // when storing, grow file as necessary to satisfy buffer request
    if (nCommand == bufferWrite && m_nPosition + nCount > m_nBufferSize)
        GrowFile(m_nPosition + nCount);
    // store buffer max and min
    *ppBufStart = m_lpBuffer + m_nPosition;

    // end of buffer depends on whether you are reading or writing
    if (nCommand == bufferWrite)
        *ppBufMax = m_lpBuffer + min(m_nBufferSize, m_nPosition + nCount);
    else
        *ppBufMax = m_lpBuffer + min(m_nFileSize, m_nPosition + nCount);
        m_nPosition += LPBYTE(*ppBufMax) - LPBYTE(*ppBufStart);
    }

    // return number of bytes in returned buffer space (may be <= nCount)
    return LPBYTE(*ppBufMax) - LPBYTE(*ppBufStart);
}
```

如果用 bufferCheck 调用 GetBufferPtr()，它会返回 1，因为 CMemFile 支持缓冲。如果是 bufferCommit，GetBufferPtr()会更新内部指针来包括 CArchive 更新后要提交的字节。

如果 CArchive 用 bufferWrite 命令调用 GetBufferPtr()函数，GetBufferPtr()会在必要时增长内存文件，然后返回参数 ppBufStart 和 ppBufMax，分别指向内存块的起点和终点。需要注意的是，在此过程中 m_nPosition 是不更新的。CMemFile 会一直等待，直至 CArchive 进行

提交，以免用户决定取消写操作。

如果 CArchive 用 bufferRead 命令调用 GetBufferPtr()，也会进行类似的计算，但内部指针会发生变化，以体现读操作的效果。

GetBufferPtr()在返回的起始地址和结束地址上体现变化，它也反映了缓冲区的大小。

在读过第 5 章中关于 CArchive 的内容之后，你会发现这部分代码很有用，而且你还可以看到 CArchive 如何实现缓冲。

CSharedFile

如果仅限于 MFC 的文档，你会认为关于 CFile 的讨论已经结束了，但其实还有一个没有文档说明的 CFile 的派生类：CSharedFile。

CSharedFile 是 CMemFile 的派生类，它使用了 Windows 全局内存 API，例如 GlobalAlloc()、GlobalReAlloc()、GlobalLock()和 GlobalFree()，而没有使用 alloc/realloc/free 等等。这个类很隐蔽，它的定义在 MFC 头文件 AFXPRIV.H 中。它的实现在源文件 FILESHRD.CPP 中。

程序清单 4-39 中包含了 AFXPRIV.H 中 CSharedFile 的声明。

程序清单 4-39　没有文档说明的 CSharedFile 类的声明

```
class CSharedFile : public CMemFile
{
DECLARE_DYNAMIC(CSharedFile) public:
// Constructors
    CSharedFile(UINT nAllocFlags = GMEM_DDESHARE|GMEM_MOVEABLE,
    UINT nGrowBytes = 4096);
// Attributes
    HGLOBAL Detach();
    void SetHandle(HGLOBAL hGlobalMemory, BOOL bAllowGrow = TRUE);
// Implementation
public:
    virtual ~CSharedFile(); protected:
    virtual BYTE* Alloc(DWORD nBytes);
    virtual BYTE* Realloc(BYTE* lpMem, DWORD nBytes);
    virtual void Free(BYTE* lpMem);
    UINT m_nAllocFlags;
    HGLOBAL m_hGlobalMemory;
    BOOL m_bAllowGrow;
};
```

首先，构造函数的默认标志指定了 GMEM_DDESHARE 和 GMEM_MOVEABLE，表示全局内存块可以共享，也可以由操作系统移动。

和 CMemFile::Attach()类似，CSharedFile 提供了一个 SetHandle()函数，它将一个类附加到一个已经分配了的全局内存句柄后面。

就实现而言，CSharedFile 和 CMemFile 十分类似，只是它使用了一个全局句柄，而不是一个指针。

使用 CSharedFile 的情况

我们发现了这个没有文档说明的类，但问题是什么时候使用它呢？

MFC 使用 CSharedFile 时用到的惟一两个实例在 MFC 的 OLE 类源文件 OLEDOBJ1.CPP 和 OLEDOBJ2.CPP 中。在这两个实例中，它都是用来从剪贴板中读取全局句柄，这个句柄包

含了图片的一个内存块（DIB）。

如果你发现自己用到了一个来自剪贴板的全局句柄，你现在就知道可以在这块内存后面粘贴一个 CSharedFile，然后通过 CFile 接口访问它。

事实上，如果你浏览了一些 MFC 样例（例如 drawcli），你会发现微软也在例子中使用了这个没有文档说明的类。

CFile 总结

CFile 类和 CStdioFile 类在已有的 C 文件 API 之外提供了一个简单的包装层。CMemFile 为一块内存提供了 CFile 接口。如果你想共享某些序列化后的数据，这一点很重要。最后我们介绍了 CSharedFile 类。它和 CMemFile 类似，但它为由一个 Windows 全局内存句柄创建和引用的内存块提供了 CFile 接口。

我们要介绍的最后一个 MFC 实用类就是处理异常的类——CException 家族。

CException：提供更好的错误处理

为了提高原有 C 模型检查函数返回值错误的能力，C++的始祖们提出了一种更好的解决方案——异常。简单地说，你只要将可能产生错误的代码放在“try”块里，然后在“catch”部分处理错误。一旦出现了错误，就会通过“throw”关键字抛出错误，这样控制流就跳到“catch”部分，然后由“catch”部分执行错误处理，如果有必要就清除错误。

例如，程序清单 4-40 就检查了分配内存时可能出现的异常。

程序清单 4-40 关于 C++异常的例子

```
//...
char * myPtr = NULL;
try
  {
    myPtr = (char *)malloc(256);
    if (myPtr == NULL) //ERROR!!
        throw "Error, could not allocate memory!";
} //end try
catch(char * ptr) //Catch block for pointers
{
    TRACE(ptr);
    //Perform any cleanup here if necessary
}
```

在 2.0 版本推出之前，MFC 用宏来实现这个语言特性，但现在 MFC 宏使用 C++语言的异常处理。大多数 MFC 程序没有直接使用 C++关键字，而仍然使用 MFC 异常处理宏。MFC 异常处理宏与 C++关键字略有不同。表 4-5 显示了 C++异常关键字与 MFC 宏之间的映射。

为了弄清楚这些宏和关键字，我们再看一下它们在 AFX.H 中的定义，如程序清单 4-41 所示。

表 4-5 MFC 异常宏与 C++关键字之间的映射

MFC 宏	C++关键字	注释
TRY	try	—
CATCH	catch	—
AND_CATCH	catch	用于二次捕获或额外捕获
END_TRY	none	标记 try 块的结束
THROW	throw	—
CATCH_ALL	catch(CException* e)	捕获所有可能的异常
THROW_LAST	throw	—

程序清单 4-41 MFC 异常宏

```
#define TRY { AFX_EXCEPTION_LINK _afxExceptionLink; try {

#define CATCH(class, e) } catch (class* e) \
{ ASSERT(e->IsKindOf(RUNTIME_CLASS(class))); \
_afxExceptionLink.m_pException = e;
#define AND_CATCH(class, e) } catch (class* e) \
{ ASSERT(e->IsKindOf(RUNTIME_CLASS(class))); \
_afxExceptionLink.m_pException = e;

#define END_CATCH } }

#define THROW(e) throw e

#define THROW_LAST() (AfxThrowLastCleanup(), throw)
// Advanced macros for smaller code

#define CATCH_ALL(e) } catch (CException* e) \
{ { ASSERT(e->IsKindOf(RUNTIME_CLASS(CException))); \
_afxExceptionLink.m_pException = e;

#define AND_CATCH_ALL(e) } catch (CException* e) \
{ { ASSERT(e->IsKindOf(RUNTIME_CLASS(CException))); \
_afxExceptionLink.m_pException = e;

#define END_CATCH_ALL } } }

#define END_TRY } catch (CException* e) \
{ ASSERT(e->IsKindOf(RUNTIME_CLASS(CException))); \
_afxExceptionLink.m_pException = e; } }
```

从程序清单 4-41 中可以看出，MFC 宏通常映射到一个_afxExceptionLink 声明。_afxExceptionLink 是 C++关键字，后面的括号表示作用域。

MFC 使用一个叫做 AFX_EXCEPTION_LINK 的结构来进行存储并将 Catch 块链接起来，这样 MFC 在遍历 catch 块链表时就可以找到与抛出的异常类型匹配的处理函数。

版本注释

MFC 后来的版本都使用了 C++关键字。这样就不需要跟踪异常，但是是否调用 e->Delete()删除异常

取决于你。

通过使用这些宏，你可以确定自己是在使用 MFC 的 CException 类。CException 类在原有的 C++异常处理机制的基础上有很大提高。

版本注释

在 Visual C++支持异常之前，这些宏都调用 setjmp/longjmp。但那些旧的代码在 MFC 新版中仍有保留，目的是为了支持旧的编译器。如果你定义了_AFX_OLD_Exception 再重新创建 MFC，你就会用到旧版本的 MFC。这些异常的一个缺点就是在被捕获后不删除任何对象［称为展开堆栈（unwinding the stack）］。这些都依赖于具体实现，它可能会造成内存泄漏。

CException 内幕

CException 是一个抽象基类，它为专门处理各种异常的 CException 派生类定义了接口。它们处理的异常类和错误类型包括：

- CArchiveException——序列化异常。
- CDaoException——DAO（数据访问对象）异常。
- CDBException——数据库异常。
- CFileException——文件异常。
- CMemoryException——内存异常。
- CNotSupportedException——某些内容不支持。
- COleDispatchException——OLE 分发（自动化）异常。
- COleException——OLE 异常。
- CResourceException——Windows 资源问题。
- CUserException——用户产生的异常。

CException 有两个主要成员函数：GetErrorMessage()和 ReportError()。前者会返回一个缓冲区，缓冲区内存放着描述异常的字符串；后者会用一个 Windows 消息框显示异常信息字符串。

在 CException 内只有一个没有文档说明的数据成员 m_bAutoDelete，它决定了 CException 是否在构造函数中删除自己。

类 CException 的定义在文件 AFX.H 中，具体实现在文件 EXCEPT.CPP 中。因为 CException 是一个抽象基类，所以它没有什么需要分析的函数。为了使你能够体会到 CException 的强大，我们看一下 CFileException 类。这个类处理与文件相关的错误。

CFileException

看一下 AFX.H 中 CFileException 的定义，你会发现一个枚举列出了所有可能出现的文件异常：

- none——没有错误发生。
- generic——发生了未定义的错误。
- fileNotFound——不能定位文件。
- badPath——路径的全部或部分无效。
- tooManyOpenFiles——打开文件的个数过多。

- accessDenied——不能访问到文件。
- invalidFile——尝试使用无效的文件句柄。
- removeCurrentDir——目前的工作目录不能删除。
- directoryFull——不能有更多的目录项。
- badSeek——设置文件指针时出现错误。
- hardIO——硬件错误。
- sharingViolation——SHARE.EXE 未载入，或者共享区域被锁定。
- lockViolation——尝试锁定一个已被锁定的区域。
- diskFull——磁盘已满。
- endOfFile——到达了文件的结尾。

在看 FILEX.CPP 中的 CFileException 内幕之前，我们先看一个例子。CFile 派生类如何使用 CFileException 通知用户产生了上面枚举的错误之一。

例如，如果用户尝试在内存块界外定位，CMemFile::Seek()成员函数会抛出一个“badSeek”异常，如果异常被检测到，CMemFile::Seek()会调用：

```
AfxThrowFileException(CFileException::badSeek);
```

AfxThrowFileException()是一个全局辅助函数，它的定义在 FILEX.CPP 中。它的伪代码如程序清单 4-42 所示。

程序清单 4-42　CFileException 的全局辅助函数 AfxThrowFileException（在文件 FILEX.CPP 中）

```
void AFXAPI AfxThrowFileException(int cause, LONG lOsError,LPCTSTR
     lpszFileName )
{
#ifdef _DEBUG
    LPCSTR lpsz;
    if (cause >= 0 && cause < _countof(rgszCFileExceptionCause))
        lpsz = rgszCFileExceptionCause[cause];
    else
        lpsz = szUnknown;
    TRACE3("CFile exception: %hs, File %s, OS error information = %ld.\n",
        lpsz, (lpszFileName == NULL) ? "Unknown" : lpszFileName, lOsError);
#endif
    THROW(new CFileException(cause, lOsError, lpszFileName));
}
```

如果调试版本发生了异常，AfxThrowFileException()会打印出一条辅助信息，包括异常类型、文件名和使用 TRACE3 宏得到的 Visual C++调试窗口的操作系统错误号。

在显示这些信息之后，AfxThrowFileException()会调用“throw”来触发异常，并传入一个新的 CFileException（带有关于错误的信息）。

例如，我们可以如程序清单 4-43 所示调用 CMemFile::Seek()。

程序清单 4-43　使用 CExceptioin 捕获错误

```
//...
CMemFile myMemFile;
try
{
    //...
```

```
}
catch (CFileException, e)
{
    // If we get here, there was an error and e will contain
    // information about it.
    //Check for a seek problem
    if (e->m_cause == CFileException::badSeek)
        //uh-oh, a bad seek, do something...
    // Delete the exception now that it isn't needed
        e->Delete();
  }
END_CATCH
```

CFileException 有 3 个帮助描述 CFile 异常的数据成员：m_cause 包含了枚举的 CFile 错误；m_lOsError 包含了::GetLastError()返回的 Win32 错误函数；m_strFileName 包含了发生异常的 CFile 的名字。任何异常处理函数的 CATCH 部分都可以访问这些方法来帮助处理异常。

CFileException 还有一些实用的用于转化的成员函数：在枚举错误类型和字符串之间进行转化；从操作系统错误号到 CFileException 异常原因进行转化。

CException 的每个派生类和 CFileExceptioin 都很类似。它们都增加了描述可能的错误情况的信息，增加了成员变量来保存帮助异常处理的错误状态。它们通常还有抛出异常的辅助结构，这些结构以 CException 派生类要保存的异常状态作为参数。

没有文档说明的异常类

没有文档说明的异常类只有一个，如果你看看 AFX.H 文件中 CMemoryException 和 CNotSupportedException 的定义，你会发现它们与 MFC 文档中说明的不一样。这两个类不是从 CException 派生的，而是从一个没有文档说明的 CSimpleException 类派生的。

CSimpleException 是用于 CMemoryException 和 CNotSupportedException 的一个辅助基类，它有空间，也有成员函数来存储错误信息。

更多信息

如果你对学习 CSimpleException 有兴趣，可以浏览 EXCEPT.CPP 文件，看它在 CException 基础上增加了什么，使得 CMemoryException 和 CNotSupportedException 从它派生而不是从 CException 派生。

结　语

本章我们学习了 MFC 实用类，例如单值类型类、集合类、文件类和异常类。我们介绍了一些有用的技术，例如 CString 的引用技术和 CArray 的内存管理，你可以在自己的 MFC 类中使用这些技术。我们还学了一些没有文档说明的类、成员函数和成员数据。如果你对此一无所知，这些内容会对你有帮助，如果你要从这些 MFC 实用类之一派生自己的类，这些就更有用了。

下一章我们会介绍 MFC 类层次结构的根：CObject。我们会看到它为形成 MFC 其他部分的基础提供了哪些特性。

CHAPTER 5

CObject

如果你像我们这样把 MFC 的类层次结构图贴在墙上，你会注意到有一个类几乎位于每个类层次结构的顶部：CObject。这样做有一个很好的理由：CObject 定义并实现了大多数 MFC 类在与框架中其他部分合作时所需要的功能。在你自己写 MFC 类或用类向导（Class Wizard）生成 MFC 类的时候，你应该确定 CObject 在你自己的类层次结构中，从而可以使用它的特性。

使用 CObject 的代价

关于将 MFC 所有的类都从 CObject 派生这样的设计（C++的语言学家称它为树状结构）是否友好的问题有过很多讨论。有些人认为这样的设计会产生大量的虚函数表，从而降低了运行时的性能，还有人认为这是一个很糟糕的面向对象的设计。

当我们揭示 CObject 的所有内幕之后，我们会回顾这个问题，并对 CObject 带来的功能与增加 CObject 虚表项对性能的影响加以比较。

CObject 的特性

CObject 就像是所有 MFC 类的切割机。一旦你从 CObject 派生一个类，你的类就能够自动拥有 4 种重要的 MFC 特性。这 4 种特性分别是：

- 运行时类信息。
- 动态创建。
- 持久性（序列化）。
- 运行时对象诊断。

CObject 的这些特性建立在彼此之上，所以我们会先逐个地分析这些特性，然后再看它们是如何共同工作的。

在进一步讨论之前，我们先看一下 CObject 的声明。对于 MFC 内部的类来说，清楚地知道 CObject 都提供了哪些函数以及这些函数如何工作很重要。程序清单 5-1 列出了 CObject 的声明（在文件 AFX.H 中）。

程序清单 5-1 MFC 树的根节点 CObject 的声明

```
class CObject
{
public:
    virtual CRuntimeClass* GetRuntimeClass() const;
    virtual ~CObject();  // virtual destructors are necessary
// Diagnostic allocations
    void* operator new(size_t, void* p);
    void* operator new(size_t nSize);
    void operator delete(void* p);

#ifdef _DEBUG
    // for file name/line number tracking using DEBUG_NEW
    void*  operator new(size_t nSize, LPCSTR lpszFileName, int nLine);
#endif
protected:
    CObject();
private:
    CObject(const CObject& objectSrc);              // no implementation
    void operator=(const CObject& objectSrc);       // no implementation
// Attributes
public:
    BOOL IsSerializable() const;
    BOOL IsKindOf(const CRuntimeClass* pClass) const;
// Overridables
    virtual void Serialize(CArchive& ar);
// Diagnostic Support
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
// Implementation
public:
    static AFX_DATA CRuntimeClass classCObject;
};
```

CObject 的声明中运用了很多 C++的小技巧。首先，析构函数是虚函数，这样就可以通过多态机制调用析构函数。换句话说，MFC 打算自己创建和删除很多从 CObject 派生的类。这样，在不需要确保调用了正确的删除操作符和析构函数的情况下，它就能删除一个对象。

然后，你会发现拷贝构造函数 CObject(const CObject& objectSrc)和操作符＝是私有的（private）。MFC 的设计者这样做的目的是，当用户尝试将一个 CObject 用＝操作符赋值给另一个 CObject 时，或者尝试用拷贝构造函数从另一个 CObject 对象创建一个 CObject 对象时，强迫编译器报错。

现在，你可以忽略像 AFX_DATA 和 AFX_DATADEF 这样的宏。它们在 MFC 的许多 DLL 中才会使用，而且我们会在第 10 章详细讨论。

在我们分析 CObject 的特性时，我们会不断地返回来看 CObject 的声明。你最好在这一页做一个标记，这样可以很快地翻回来。

宏的介绍

MFC 中常见的一种模式就是使用一对宏（DECLARE/IMPLEMENT）在类中增加各种特性。

DECLARE 宏通常定义一些成员变量和成员函数，IMPLEMENT 宏通常实现那些成员函数。DECLARE 宏一般在头文件中，而 IMPLEMENT 宏一般在 C++文件中。

一旦有机会，我们就会把宏扩展出来，看它里面都有些什么，但是记住 DECLARE 宏是在类中增加成员，而 IMPLEMENT 是为 DECLARE 宏中定义的成员函数提供实现。

运行时类的信息

CObject 的 RTCI（run-time class information，运行时类的信息）特性使开发人员在运行时可以确定关于类的信息，例如类的名字和父类的名字。

即使已经使用了 MFC 一段时间，你可能还没有意识到正在类中使用 RTCI。这是因为支持 RTCI 的代码都在以下这些宏里：

DECLARE_DYNAMIC/IMPLEMENT_DYNAMIC

DECLARE_DYNCREATE/IMPLEMENT_DYNCREATE

DECLARE_SERIAL/IMPLEMENT_SERIAL

在其他节中我们将讨论第二个和第三个宏集合。类向导会自动地在你的类声明（.H）文件中添加这些声明宏，相应的实现宏也会被添加到类的实现文件（.CPP）中。

一旦你已经将这些宏加入到 CObject 的派生类中，就可以调用 IsKindOf()来测试这个类的类型。IsKindOf()函数有一个参数，这个参数由另一个宏 RUNTIME_CLASS 创建。

程序清单 5-2 显示如何使用 MFC RTCI。

程序清单 5-2　MFC RTCI 的例子

```
//This lives in .h file
class CMyClass : public CObject
{
    DECLARE_DYNAMIC(CMyClass);  //Mystery macro 1
public:
    CMyClass();
    //...
}
//This lives in .cpp file
IMPLEMENT_DYNAMIC(CMyClass, CObject) //Mystery macro 2
DemoRTCI()
{
    CObject * pObject = new CMyClass;
    //Mystery macro 3
    if ( pObject->IsKindOf(RUNTIME_CLASS(CMyClass))){
        CMyClass * pMyObject = (CMyClass *)pObject;
    }
}
```

这个例子说明了如何使用 RTCI 来检查到派生类的强制转化是否安全。

为什么不使用 C++ RTTI?

此时你可能会问为什么 MFC 开发人员不使用 C++的 RTTI（run-time type information language，运行时类型信息语言）这个特性，而要使用这些复杂的宏。MFC 组和语言组是分离开的，在 MFC 早期他们确实需要 RTTI 的支持。但是后来他们没有等待语言组增加 C++ RTTI 支持（最终在 Visual C++的 4.0 版本中实现了），而是自己实现了这方面的支持。最近有传言说 MFC 未来版本会用 C++ RTTI 特性替代 RTCI 宏的实现。这并不影响大多数应用程序，因为这些宏和实现已经脱离开来。

RTCI 如何运作

既然你已经看到了一个如何使用 RTCI 的例子，就让我们再看 MFC 组如何实现这种支持，你也可以看看其中是否有你在程序中能用得到的内容。

RTCI 宏在文件 AFX.H 中，它对 RTCI 的支持的程序清单在 MFC 源文件 OBJCORE.CPP 中，下面我们首先分解这些宏。

当你在自己的类的头文件中增加了 DECLARE_DYNAMIC 宏时，需要将类的名字作为参数传递给它。让我们看看 DECLARE_DYNAMIC 的定义，然后将它应用到例子中的类里。

下面是 DECLARE_DYNAMIC 宏的声明：

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

这段代码中有了一个小技巧，它使用了预处理连接操作符“##”。操作符“##”告诉预处理器将操作符右侧的部分和左侧的部分连接在一起。例如 MFC##Internals 经过预处理器后的结果就是 MFCInternals。如果参数出现在其中，就会将参数替代进去再连接。你最好习惯这些用法，因为 MFC 在很多地方都使用这些技术。

将 CMyClass 插入到 DECLARE_DYNAMIC 宏中，再运行预处理器，就会使如下代码添加到 CObject 的派生类的定义中：

```
public:
  static CRuntimeClass classCMyClass;
  virtual CRuntimeClass * GetRuntimeClass() const;
```

第一行创建了一个静态 CRuntimeClass 成员变量，叫做 classCMyClass。回忆一下静态类变量如何运作：不管你创建了这个类的多少个实例，静态方法的拷贝都只有一个。

如果回头看程序清单 5-1 中 CObject 的声明，你就会发现 CMyClass::GetRuntimeClass() 成员方法的声明覆盖了虚函数 CObject::GetRuntimeClass()。这样，CMyClass::GetRuntimeClass() 会返回正确的运行时类信息。

另一个类

正如你所知，这里还有一个类：CRuntimeClass。在看这个宏的 IMPLEMENTATION 部分之前，让我们先仔细看一下静态 CRuntimeClass 的声明。

程序清单 5-3 给出了 CRuntimeClass 的声明（在 AFX.H 中）。

程序清单 5-3 CRuntimeClass 辅助结构的声明

```
struct CRuntimeClass
{
// Attributes
    LPCSTR m_lpszClassName;
    int m_nObjectSize;
    UINT m_wSchema; // schema number of the loaded class
    //Use to be m_pfnConstruct in MFC 2.x
    void (PASCAL* m_pfnCreateObject)(void* p);
    // if NULL => abstract base class
    CRuntimeClass* m_pBaseClass;
// Operations
    CObject* CreateObject();
// Implementation
    BOOL ConstructObject(void* pThis);
    void Store(CArchive& ar);
    static CRuntimeClass* PASCAL Load(CArchive& ar, UINT* pwSchemaNum);
    // CRuntimeClass objects linked together in simple list
    CRuntimeClass* m_pNextClass;       // linked list of registered classes
// special debug diagnostics
#ifdef _DEBUG
    BOOL IsDerivedFrom(const CRuntimeClass* pBaseClass) const;
#endif
};
```

你会发现其实这里没有类。它只是一个结构。它的名字“CRuntimeClass”似乎取错了。其实 C++的结构也是类，只不过它的所有属性在默认情况下都是 public 的。换句话说，结构可以有成员函数和其他类似于类的特性，但是它的默认成员域都是 public，这样你就不必要写：

```
class CMyClass {
public:
    // stuff
};
```

因为 CRuntimeClass 是一个结构，所以它的所有成员都是 public 的。这样使用起来很方便，因为它使得你可以很容易地得到 CRuntimeClass 的成员。

IMPLEMENT_DYNAMIC

为了进一步理解 CRuntimeClass，下面看 IMPLEMENT_DYNAMIC(classname,base_classname)宏都完成了些什么。

```
#define IMPLEMENT_DYNAMIC(class_name, base_class_name) \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, NULL)
#define _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, pfnNew) \
    AFX_DATADEF CRuntimeClass class_name::class##class_name = { \
    #class_name, sizeof(class_name), wSchema, pfnNew, \
    RUNTIME_CLASS(base_class_name), NULL }; \
    static const AFX_CLASSINIT \
    _init_##class_name(&class_name::class##class_name); \
    CRuntimeClass* class_name::GetRuntimeClass() const \
    { return &class_name::class##class_name; } \
```

即使我们已经少写了部分内容，但这个宏看起来还是很复杂。这里还使用了另一个有趣的预处理器宏："#"，或者叫做字符串化的宏。这个宏将它右侧的部分转化成用引号包括的字符串。下面我们用参数 CMyClass 和 CObject 来在 IMPLEMENT_DYNAMIC 上运行预处理器。

```
AFX_DATADEF CRuntimeClass CMyClass::classCMyClass = {
    "CMyClass", sizeof(CMyClass),0xFFFF, NULL, RUNTIME_CLASS(CObject),
        NULL };
static const AFX_CLASSINIT _init_CMyClass(&CMyClass::classCMyClass);
CRuntimeClass * CMyClass::GetRuntimeClass() const
{

    return &CMyClass::classCMyClass;
}
```

看到这些生成的代码，你就知道这个宏完成了 3 件事：

1. 它初始化了静态 CRuntimeClass 数据成员变量 classCMyClass（你可能需要再回忆一下 CRuntimeClass 声明）：

```
m_lpszClassName = "CMyClass";
m_nObjectSize = sizeof(CMyClass);
m_wSchema = 0xFFFF;
m_pfnCreateObject = NULL;
m_pBaseClass = RUNTIME_CLASS(CObject);
```

2. 它创建了一个静态 AFX_CLASSINIT 结构。

这段代码看起来可能不是很清晰，不过在你看到了 AFX_CLASSINIT 的声明之后一切就清楚了。AFX_CLASSINIT 是一个结构，它只有一个构造函数。

```
struct AFX_CLASSINIT
{ AFX_CLASSINIT(CRuntimeClass* pNewClass); };
```

IMPLEMENT_DYNAMIC 创建了一个静态的 AFX_CLASSINIT 接口，并调用了它的构造函数，参数是 CMyClass::classCMyClass 这个 CRuntimeClass 静态结构。这使得 CMyClass::classCMyClass 被添加到 MFC 的一个内部状态链表中。关于这个链表我们会在第 9 章中讨论（在本章的"内存诊断"中我们还会讨论这个语句）。

3. IMPLEMENT_DYNAMIC 宏会生成这个覆盖成员函数 GetRuntimeClass()。GetRuntimeClass()会返回静态 CRuntimeClass 成员变量 classCMyClass 的地址。

最后一个要分解的关于运行时信息的宏就是 RUNTIME_CLASS，它的定义如下：

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

RUNTIME_CLASS 宏会返回静态 CRuntimeClass 成员数据，DECLARE_DYNAMIC 宏会将 CRuntimeClass 成员变量放置在你的类中（类似于 GetRuntimeClass()）。

CObject::IsKindOf()内幕

以上的讨论会带来几个关于 CRuntimeClass 的问题。像 m_wSchema 和 m_pfnCreateObject 这样的变量是用来做什么的？为什么 CRuntimeClass 有一个链表？这个链表是做什么的？为了帮助回答这些问题，让我们来看 CObject::IsKindOf()是如何实现的。

程序清单 5-4 显示了 CObject::Is kind of ()的伪代码，它位于 OBJCORE.CPP 中。

程序清单 5-4 CObject::IsKindOf()的实现

```
BOOL CObject::IsKindOf(const CRuntimeClass* pClass) const
{
    CRuntimeClass* pClassThis = GetRuntimeClass();
    while (pClassThis != NULL){
        if (pClassThis == pClass)
            return TRUE;
        pClassThis = pClassThis->m_pBaseClass;
    }
    return FALSE;       // walked to the top, no match
}
```

IsKindOf()有一个参数，这个参数是 RUNTIME_CLASS 返回的静态 CRuntimeClass 指针，接着它会为当前对象调用 GetRuntimeClass()函数。然后比较 CRuntimeClass 指针，看它们是否指向同一个静态结构。如果它们是同一个对象，就匹配了。否则，IsKindOf()就会遍历继承树，查找一个匹配的结构。

注意一下程序清单 5-2 中对 IsKindOf()的调用。在这种情况下，IsKindOf()会从 CMyClass 开始，没有找到匹配的结构，然后就沿着继承树向上一层，从 CMyClass 到 CObject 来查找匹配的结构。

再检查一下程序清单 5-1。CObject 运行时类 classCObject 定义在类的底部。你可能在初始化的时候遗漏了它，因为 DECLARE_DYNAMIC 不负责隐藏这些内容。

CRuntimeClass 中没有文档说明的函数：IsDerivedFrom()

CRuntimeClass 中还有很多我们没有分析的函数。我们会在后面分析这些函数，并且回答一些关于 CRuntimeClass 的问题。但是现在我们先看一个没有文档说明的函数：IsDerivedFrom()。

程序清单 5-5 列出了 CRuntimeClass::IsDerivedFrom()的实现。

程序清单 5-5 CRuntimeClass::IsDerivedFrom()的代码

```
BOOL CRuntimeClass::IsDerivedFrom(const CRuntimeClass* pBaseClass) const
{
    const CRuntimeClass* pClassThis = this;
    while (pClassThis != NULL){
        if (pClassThis == pBaseClass)
            return TRUE;
        pClassThis = pClassThis->m_pBaseClass;
    }
    return FALSE;       // walked to the top, no match
}
```

看起来很眼熟，不是吗？它的代码和 IsKindOf()的代码是一样的，但是你通过 CRuntimeClass 对象调用它。MFC 用 CRuntimeClass::IsDerivedFrom()来对开发人员是否传入了一个合法的派生对象进行双重检查（例如，在一个分割窗口中 MFC 会检查分割窗口是否是一个合法的 CWnd 派生类）。在很多这种情况下，对象还没有创建，但是 MFC 仍需要知道它是否指向了一个正确的类型（是否是 CRuntimeClass 的一个派生类）。如果你自己也遇到类似情况，就会发现这个函数很方便。

下面让我们看 CObject 对动态创建的支持，其中我们还可以学习到关于 CRuntimeClass 结构的内容。

动态创建

动态创建是这里要专门讨论的，一般的学校是不教这个的。开玩笑而已。为了在你的 CObject 派生类中增加对动态创建的支持，要使用 DECLARE_DYNCREATE 宏和 IMPLEMENT_DYNCREATE 宏。这些宏在 DECLARE_DYNAMIC 宏和 IMPLEMENT_DYNAMIC 宏的基础上创建，所以不用在你的类中同时使用这两种宏。事实上，你同时使用它们会导致编译器报错。

一旦已经添加了这些宏，就可以在这些 CRuntimeClass 信息的基础上创建对象了。要完成这些，你只要调用 CRuntimeClass 的成员函数 CreateObject()即可。

程序清单 5-6 列出了动态创建的一个例子。这段代码假设 CMyClass 和程序清单 5-2 中列出的一样，但是用 DYNCREATE 宏替代了 DYNAMIC 宏。

程序清单 5-6 如何使用动态创建的例子

```
CRuntimeClass* pRuntimeClass = RUNTIME_CLASS( CMyClass );
CObject* pObject = pRuntimeClass->CreateObject();
ASSERT( pObject->IsKindOf( RUNTIME_CLASS( CMyClass ) ) );
```

MFC 使用动态创建的范围很广。你最熟悉的一个例子可能就是 MFC 的 CMultiDocTemplate 类，它的创建代码如下：

```
pDocTemplate = new CMultiDocTemplate(IDR_SCRIBTYPE,
               RUNTIME_CLASS(CScribDoc),
               RUNTIME_CLASS(CMDIChildWnd),
               RUNTIME_CLASS(CScribView));
```

MFC 首先存储了 CRuntimeClass 结构，然后调用 CRuntimeClass::CreateObject()函数初始化对象（关于文档模板的更多信息和文档模板如何工作请参考第 7 章）。

动态创建如何完成？

只要你理解了 RTCI 是如何实现的，理解 CObject 对动态创建的支持就很容易。下面我们就分析宏的动态创建，看它们是如何工作的。

动态创建的关键在于 CRuntimeClass 结构和它的 CreateObject()方法。DECLARE_DYNCREATE 宏与 DECLARE_DYNAMIC 宏的输出只在一行上有区别。

```
static CObject * CreateObject();
```

正如你所猜测的那样，IMPLEMENT_DYNCREATE 宏不但生成了一般的 IMPLEMENT_DYNAMIC，还生成了 CreateObject()成员函数。生成的 CreateObject()的实现如下：

```
CObject* PASCAL CMyClass::CreateObject()
{
    return new CMyClass;
}
```

IMPLEMENT_DYNCREATE 宏还用不同的参数调用了_IMPLEMENT_RUNTIMECLASS 宏。然后把 CMyClass::CreateObject()作为参数传入，用来初始化 CRuntimeClass::m_

pfnCreateObject 函数指针。静态 CRuntimeClass 结构会用 m_pfnCreateObject 成员中保存的指向 CMyClass::CreateObject()的指针来初始化，然后用 m_pfnCrateObject 在运行时动态创建一个 CMyClass 类型的对象。

CRuntimeClass::CreateObject()实现

CreateObject()的实现在文件 OBJCORE.CPP 中，它的伪代码如程序清单 5-7 所示。

程序清单 5-7 CRuntimeClass::CreateObject，MFC 动态创建的核心

```
CObject* CRuntimeClass::CreateObject()
{
    if (m_pfnCreateObject == NULL) {
        //Error!  User did not use correct macro
        return NULL;
    }
    CObject* pObject = NULL;
    TRY {
        pObject = (*m_pfnCreateObject)();
    }
    END_TRY
    return pObject;
}
```

CRuntime::CreateObject()首先会进行检查，确定用户是否指定了正确的宏，然后调用 CRuntimeClass::m_pfnCreateObject 成员变量指向的函数。在我们的例子中，这个指针指向了 CMyClass::CreateObject()，这个函数最终会调用“new CMyClass”。这个对象会通过 CreateObject()调用被返回给 CRuntimeClass::CreateObject()的调用者。

CRuntimeClass 剩下的部分

关于 CRuntimeClass 的成员数据和成员函数，我们几乎已经了解了大部分，除了以下几个：

```
LPCSTR m_lpszClassName;
int m_nObjectSize;
UINT m_wSchema;
void Store(CArchive& ar) const;
static CRuntimeClass* PASCAL Load(CArchive& ar, UINT* pwSchemaNum);
CRuntimeClass* m_pNextClass;       // linked list of registered classes
```

下面一节我们会讨论关于 MFC 如何支持持续性（persistence）。

MFC 中的持续性

持续性就是指存储对象，并在之后的某个时刻恢复对象的能力。这个特性在 MFC 中被称为序列化（serialization）。使用持续性，你可以快速地实现文件读写，而不需要担心所操作文件的格式。你只要在文件后面连接对象，并按相反顺序读出。

回头看一下程序清单 5-1 中 CObject 的声明，你会发现 CObject 定义了两个与序列化有关的成员函数：IsSerializable()和 Serialize()。下面我们看看如何在你的类中增加序列化代码，然后分析 MFC 如何实现序列化。

在类中增加序列化代码

为了在你的 CObject 派生类中支持序列化，你必须做两件事：

1. 使用 DECLARE_SERIAL/IMPLEMENT_SERIAL 宏。IMPLEMENT_SERIAL 宏带一个 WORD 作为参数，它定义了关于你的对象的版本信息。你可以利用它定义对象格式的版本。如果在读入数据的时候发现版本不匹配，MFC 就不会读入对象。

2. 覆盖 CObject::Serialize()方法来读/写自己的成员数据。

下面是一个新的 CMyClass 声明，它增加了对序列化的支持，还添加了一些新的成员来说明序列化。

```
class CMyClass : public CObject
{
    DECLARE_SERIAL(CMyClass);
public:
    CMyClass();
    WORD   m_wType;
    DWORD  m_dwData;
    CPoint m_ptMiddle;
    void Serialize(CArchive &);
}
```

为了实现成员函数 CMyClass::Serialize()，我们使用插入操作符(<<)和提取操作符(>>)：

```
IMPLEMENT_SERIAL(CMyClass, CObject, 0xabcd)
void CMyClass::Serialize(CArchive & ar)
{
    if (ar.IsStoring()){
        ar << m_wType;
        ar << m_dwData;
        ar << m_ptMiddle;
    }
    else{
    ar >> m_wType;
    ar >> m_dwData;
    ar >> m_ptMiddle;
    }
}
```

序列化不能再简单了！对于基本类型，只要使用提取/插入操作符。对于更复杂的类型，要调用它们的 Serialize()方法，而且要记住传递 CArchive 参数给它。

当然，如果一件事情过于简单，那它后面必然隐藏着很多内幕。对于 MFC 序列化来说也是这样。

序列化如何运作

本节我们会给出在 AFX.H 和 AFXWIN.H 中的序列化声明。序列化的实现在 MFC 的源文件 ARCCORE.CPP 和 ARCOBJ.CPP 中。

为了全面理解 MFC 序列化，我们会先研究序列化的辅助类 CArchive。然后我们会分析 DECLARE_SERIAL/IMPLEMENT_SERIAL 宏。最后我们会在框架中跟踪类的存储和读取，从而看一下序列化如何运作。

CArchive 辅助类

CArchive 类没有实现一般的 ASCII C++流（或输入/输出流），而是实现了一个二进制流，它和一个 CFile 指针绑定在一起，而且为了获得更高的性能而实现了数据缓冲。

让我们看一下 CArchive 类是如何实现的。它的定义在 AFX.H 中，如程序清单 5-8 所示。

程序清单 5-8　CArchive 的声明

```
class CArchive
{
public:
    // Flag values
    enum Mode { store = 0, load = 1, bNoFlushOnDelete = 2, bNoByteSwap = 4 };
    CArchive(CFile* pFile, UINT nMode, int nBufSize = 512, void* lpBuf =
        NULL);
    ~CArchive();
// Attributes
    BOOL IsLoading() const;
    BOOL IsStoring() const;
    BOOL IsByteSwapping() const;
    BOOL IsBufferEmpty() const;
    CFile* GetFile() const;
    UINT GetObjectSchema(); // only valid when reading a CObject*
    void SetObjectSchema(UINT nSchema);
    CDocument* m_pDocument;
// Operations
    UINT Read(void* lpBuf, UINT nMax);
    void Write(const void* lpBuf, UINT nMax);
    void Flush();
    void Close();
    void Abort();    // close and shutdown without exceptions
// reading and writing strings
    void WriteString(LPCTSTR lpsz);
    LPTSTR ReadString(LPTSTR lpsz, UINT nMax);
    BOOL ReadString(CString& rString);

public:
    friend CArchive& operator<<(CArchive& ar, const CObject* pOb);
    friend CArchive& operator>>(CArchive& ar, CObject*& pOb);
    friend CArchive& operator>>(CArchive& ar, const CObject*& pOb);
// insertion operations
    CArchive& operator<<(BYTE by);
    CArchive& operator<<(WORD w);
    CArchive& operator<<(LONG l);
    CArchive& operator<<(DWORD dw);
    CArchive& operator<<(float f);
    CArchive& operator<<(double d);
    CArchive& operator<<(int i);
    CArchive& operator<<(short w);
    CArchive& operator<<(char ch);
// extraction operations
    CArchive& operator>>(BYTE& by);
    CArchive& operator>>(WORD& w);
    CArchive& operator>>(DWORD& dw);
    CArchive& operator>>(LONG& l);
    CArchive& operator>>(float& f);
    CArchive& operator>>(double& d);
    CArchive& operator>>(int& i);
```

```
    CArchive& operator>>(short& w);
    CArchive& operator>>(char& ch);
// object read/write
    CObject* ReadObject(const CRuntimeClass* pClass);
    void WriteObject(const CObject* pOb);
// advanced object mapping (used for forced references)
    void MapObject(const CObject* pOb);
// advanced versioning support
    void WriteClass(const CRuntimeClass* pClassRef);
    CRuntimeClass* ReadClass(const CRuntimeClass* pClassRefRequested = NULL,
                UINT* pSchema = NULL, DWORD* pObTag = NULL);
    void SerializeClass(const CRuntimeClass* pClassRef);
// advanced operations (used when storing/loading many objects)
    void SetStoreParams(UINT nHashSize = 2053, UINT nBlockSize = 128);
    void SetLoadParams(UINT nGrowBy = 1024);
// Implementation
public:
    BOOL m_bForceFlat;  // for COleClientItem implementation (default TRUE)
    BOOL m_bDirectBuffer;   // TRUE if m_pFile supports direct buffering
    void FillBuffer(UINT nBytesNeeded);
    void CheckCount();    // throw exception if m_nMapCount is too large
// special functions for reading and writing (16-bit compatible) counts
    DWORD ReadCount();
    void WriteCount(DWORD dwCount);
// public for advanced use
    UINT m_nObjectSchema;
    CString m_strFileName;
protected: // archive objects cannot be copied or assigned
    CArchive(const CArchive& arSrc);
    void operator=(const CArchive& arSrc);
    BOOL m_nMode;
    BOOL m_bUserBuf;
    int m_nBufSize;
    CFile* m_pFile;
    BYTE* m_lpBufCur;
    BYTE* m_lpBufMax;
    BYTE* m_lpBufStart;
// array/map for CObject* and CRuntimeClass* load/store
    UINT m_nMapCount;
    union {
        CPtrArray* m_pLoadArray;
        CMapPtrToPtr* m_pStoreMap;
    };
// map to keep track of mismatched schemas
    CMapPtrToPtr* m_pSchemaMap;
// advanced parameters (controls performance with large archives)
    UINT m_nGrowSize;
    UINT m_nHashSize;
};
```

可能在刚刚我说在任何简单事情后面的内幕都是颇为复杂的时候你还认为我很夸张，现在相信了吧。

CArchive 中大部分都没有文档说明

如果你查看 MFC 的参考手册，会发现关于 CArchive 的记录很少。下面是 CArchive 中使

用的一些不是很明显的成员：

- m_nMode——指定了文档是否正在读和写。还可以用它在删除的时候关闭刷新。MFC 的 Macintosh 版本使用了 bNoByteSwap 模式来在序列化过程中进行字节交换。
- IsLoading()、IsStoring()、IsByteSwapping()和 IsBufferEmpty()——这些函数只访问模式，用来确定 CArchive 对象的状态。
- 操作符——CArchive 为 CObject 派生类、Windows 数据类型和 C++类型定义了插入和提取操作符。
- Map 成员——进行写操作时，CArchive 维护了一个映射表，以便它能迅速访问到类似对象的类信息。CArchive 还利用这个映射表来确保对一个特定的类只写一次 CRuntimeClass 信息，然后在序列化流中引用这个信息。

  进行读操作时，CArchive 会维护已经创建的对象的数组，并在数组中存储那些已经读出的 CRuntimeClass 结构信息。这样，每当 CArchive 查找一个写入到序列化流的引用时，都可以查找该数组。

  MapObject()、m_nMapCount、m_pLoadArray、m_pStoreMap、m_pSchemaMap 和 m_nGrowSize 这些成员在为读和写实现映射表和数组时都使用到了。
- 类成员函数——WriteClass()、ReadClass()和 SerializeClass()等成员函数用来序列化 CRuntimeClass 结构信息（过一会儿会讨论更多关于这方面的内容）。

其他方法实现了标准的缓冲，并存储了关于该 CArchive 使用的 CFile 的信息。

CArchive 操作符的内幕

为了更进一步了解我们在使用 CArchive 时它都做了些什么，下面来看一下 WORD 类型的提取和插入操作符。这些操作符都是内嵌的，在 INCLUDE\AFX.INL 内嵌文件中。

下面是插入操作符的实现代码：

```
_AFX_INLINE CArchive& CArchive::operator<<(WORD w)
{
    if (m_lpBufCur + sizeof(WORD) > m_lpBufMax)
        Flush();
    *(WORD*)m_lpBufCur = w;
    m_lpBufCur += sizeof(WORD);
    return *this;
}
```

插入操作符所完成的就是检查它是否需要刷新内部 CArchive 缓冲区，然后将 WORD 放入缓冲区，最后增长缓冲区的指针。插入操作符会返回一个 CArchive 引用，这样就可以反复使用插入操作符。

下面再看一下 WORD 类型的提取操作符：

```
_AFX_INLINE CArchive& CArchive::operator>>(WORD& w)
{
    if (m_lpBufCur + sizeof(WORD) > m_lpBufMax)
        FillBuffer(sizeof(WORD) - (UINT)(m_lpBufMax - m_lpBufCur));
    w = *(WORD*)m_lpBufCur;
    m_lpBufCur += sizeof(WORD);
    return *this;
}
```

提取操作符所完成的就是检查它是否需要从文件中截取更多的数据放入内部缓冲区中。然后将缓冲区中的数据取出按照 WORD 类型放入 word 引用参数中。最后它会增长缓冲区指针，为下一次提取操作做准备。

序列化宏

既然我们已经对 CArchive 做什么有了一定的了解，下面再看序列化宏 DECLARE_SERIAL 和 IMPLEMENT_SERIAL 宏都做了些什么。

下面是 DECLARE_SERIAL 的定义：

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

DECLARE_SERIAL 宏只是在我们前面提到的 RTCI 和动态创建部分中增加一行代码。下面这段代码是宏被预处理之后在你的类声明中的样子：

```
friend CArchive& operator>> (CArchive& ar, CMyClass* &pOb);
```

这一行声明了一个全局提取操作符，并将它作为类的友元。

再看一下 IMPLEMENT_SERIAL 就会发现更多：

```
#define IMPLEMENT_SERIAL(class_name, base_class_name, wSchema) \
    CObject* class_name::CreateObject() \
    { return new class_name; } \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, \
    class_name::CreateObject) \
    CArchive& operator>>(CArchive& ar, class_name* &pOb) \
    { pOb = (class_name*) ar.ReadObject(RUNTIME_CLASS(class_name)); \
    return ar; } \
```

IMPLEMENT_SERIAL 宏传递了一个新的参数给_IMPLEMENT_RUNTIMECLASS 实用宏。这个新的参数就是 wSchema，它会赋值给 CRuntimeClass::m_wSchema 成员（过一会和会讨论更多关于这方面的内容）。

IMPLEMENT_SERIAL 生成了全局提取宏的实现：

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

IMPLEMENT_SERIAL 重载了操作符>>，这样它就可以用 RUNTIME_CLASS 返回的 CRuntimeClass 成员调用 CArchive::ReadObject()。让我们看看 ReadObject()。

CArchive::ReadObject()

这是一个非常复杂的函数。程序清单 5-9 列出了它的伪代码，并突出了我们感兴趣的部分。

程序清单 5-9 CArchive::ReadObject()成员函数的实现

```
CObject* CArchive::ReadObject(const CRuntimeClass* pClassRefRequested)
{
    UINT nSchema;
    DWORD obTag;
    CRuntimeClass* pClassRef = ReadClass(pClassRefRequested, &nSchema,
```

```
                                                &obTag);
        // allocate a new object based on the class just acquired
        pOb = pClassRef->CreateObject();
        // Serialize the object with the schema number set in the archive
        UINT nSchemaSave = m_nObjectSchema;
        m_nObjectSchema = nSchema;
        pOb->Serialize(*this);
        m_nObjectSchema = nSchemaSave;
        return pOb;
    }
```

在 CArchive::ReadObject()中，首先从文件中读取 CRuntimeClass 结构。如果已经读取过这种类型的一个类，就会已经有一个引用，ReadObject()就会在内部数组中查找它。然后 ReadObject()会使用动态方法创建一个对象，最后调用对象的 Serialize()方法，并将当前的 CArchive 引用（pOb->Serialize(*this)）传入。

为什么没有插入操作符？

此时，你可能想问为什么序列化宏没有定义插入操作符。因为在 AFX.INL 中有一个全局的插入操作符：

```
CArchive& operator<<(CArchive& ar, const CObject* pOb)
{
    ar.WriteObject(pOb);
    return ar;
}
```

WriteObject() 和 ReadObject() 类似。它调用了 WriteClass()，WriteClass() 会写入 CRuntimeClass 信息，然后 WriteObject()又会调用对象的 Serialize()成员函数。

既然 WriteObject()不需要任何特定的 CRuntimeClass 信息，因此 CObject 插入操作符也就足够了，也就不需要为 CObject 的派生类再定义新的插入操作符了。

CObject 和序列化

既然我们已经知道序列化宏都完成了什么，就让我们再看一下 CObject 的两个序列化函数：IsSerializable()和 Serialize()。

CObject::IsSerializable()

这个函数的实现在 OBJCORE.CPP 中：

```
BOOL CObject::IsSerializable() const
{
    return (GetRuntimeClass()->m_wSchema != 0xffff);
}
```

这个函数就是为了确保用户已经为序列化指定了正确的宏，它会检查 CRuntimeClass 结构的 m_wSchema 是不是 0xffff。

MFC 手册中提到，在执行针对类的序列化之前，应该先调用被覆盖的 CObject::Serialize()。下面说明了为什么。

CObject::Serialize()

令人惊讶的是 CObject::Serialize()里没有任何代码，这个函数位于 AFX.INL 文件中。如

果你想自己查看一下，可以查找这个文件。这个只是 MFC 目前的版本中 Serialize()函数的占位符。也许在未来微软会修改这个函数，在里面增加一些在调用针对你的类的 Serialize()代码之前必须调用的代码。调用 CObject::Serialize()这种形式很好，但是现在你知道它实际上不做任何事。

跟踪 CObject 派生对象的读和写

为了帮助理解序列化如何运行，我们在框架中跟踪一次写操作和读操作的处理过程。为了让事情变得有趣一些，我们假设正在使用文档/视图结构（第 7 章会详细讨论），而且我们的文档中包含了一个名为 m_pMyClass 的 CMyClass 成员指针。下面是我们的 CDocument 派生类 CMyDocument 的 Serialize()方法。

```
void CMyDocument::Serialize(CArchive & ar)
{
    if (ar.IsStoring()){
        ar << m_pMyClass ;
    } else {
    ar >> m_pMyClass;
    }
}
```

CMyClass::Serialize()方法序列化了一个 WORD、DWORD 和一个 CPoint。图 5-1 显示了一旦用户选择了文档要写入的文件的名字，MFC 和你的 CDocument/CObject 派生类要采取的步骤。

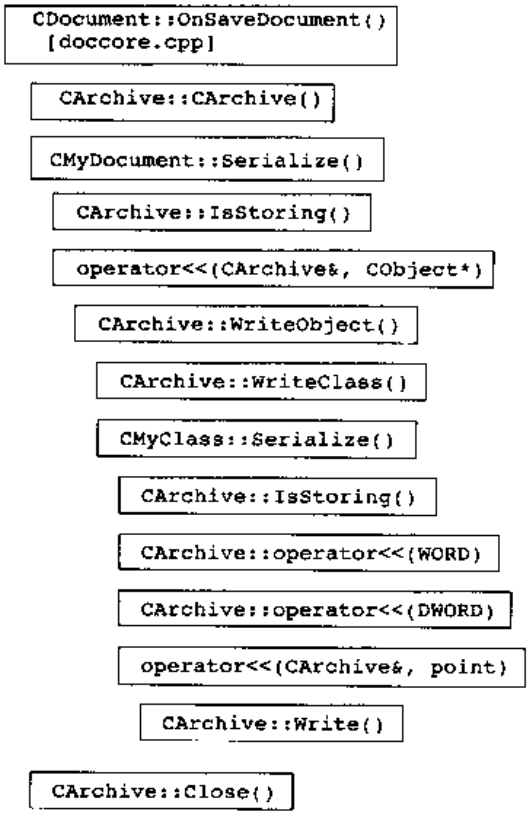


图 5-1 序列化存储操作的步骤

在第 1 步中，CDocument::OnSaveDocument()以存储模式创建了一个 CArchive 对象，然后将它贴到由对话框(pFile)打开的文件后面。然后，OnSaveDocument()会调用文档的 Serialize()方法，最后关闭文档。下面是 CDocument::OnSaveDocument()中相关的代码：

```
    // ...
CArchive saveArchive(pFile, CArchive::store |
                     CArchive::bNoFlushOnDelete);
saveArchive.m_pDocument = this;
TRY {
    Serialize(saveArchive);     // save me
saveArchive.Close();
ReleaseFile(pFile, FALSE);
}
// ...
```

在第 2~5 步中，CMyDocument::Serialize()方法被调用。它检查文档是否是通过 IsStoring()方法保存的，而且为写操作调用插入操作符（第 5 步）。

在第 6~8 步中，插入操作符又调用了 CArchive::WriteObject()函数。这个函数又调用了 WriteClass()，然后序列化 CMyDocument 对象。

第 9~13 步才将实际数据写入输出缓冲区中，最后，文档在第 14 步由 CDocument::OnSaveDocument()关闭。

下面再看一下同一对象的读操作，如图 5-2 所示。

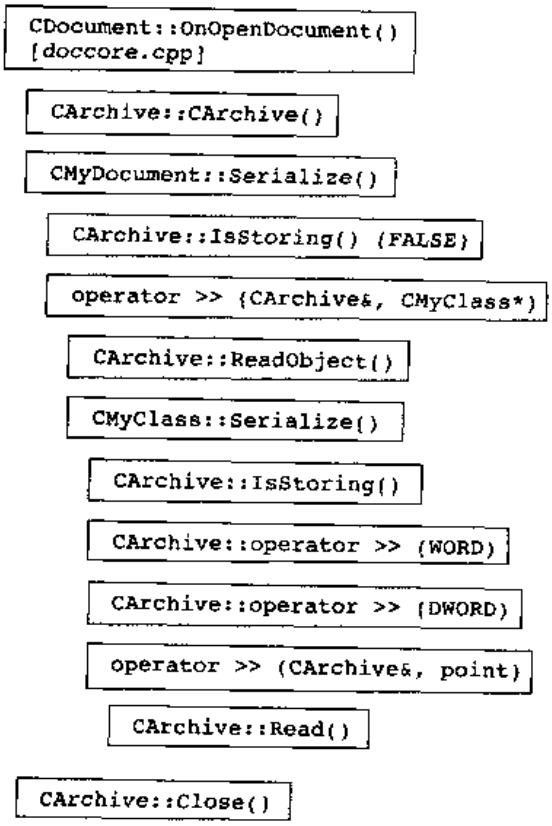


图 5-2 序列化读操作的步骤

基本上，在进行读操作时，框架“展开”（unwind）了它在保存操作过程中写入的内容。

利用这种方法，框架可以确保数据以写入的顺序读出。

序列化性能

回忆我们前面讨论的 CArchive 声明，我们简短地提到了 MFC 在序列化过程中使用的映射表和数组。下面我们仔细看看这些映射表和数组的作用。

考虑一个应用程序，它不得不写入上千个某种类型（例如 CPoint）的对象。将所有关于 CPoint 的信息写入几千次是不是很浪费？MFC 会通过对第一个对象类型序列化 CRuntime 成员来解决这个问题。对于后面相同类型的对象，MFC 会只写入一个引用而不会写入整个 CRuntime 信息（例如，1 表示第一个对象）。因为 MFC 会在映射表中存储引用和对象名，所以当所有相同类型的其他对象被写入时就只要存储引用。

在读入的时候，CArchive 会维护一个已经读出的对象数组。当它读入一个 CRuntime 引用，而不是一个 CRuntimeClass 结构时，CArchive 会在数组中查找这个引用，并使用数组中存储的 CRuntimeClass 信息。换句话说，CArchive 保存了一个数组，以便它能将引用解码成 CRuntimeClass 信息。

另一个关于序列化性能的考虑就是序列化大量数据的性能影响。如果数据大于 16KB，还有两个 API 可以帮助你提高 MFC 序列化的性能：SetLoadParams()和 SetStoreParams()。

这些值可以帮助你压缩存储映射表的散列表的大小，并且可以使读入数组每次增长的大小适合你的需要。如果你不记得如何用这些值修改 MFC 的集合类，就请回忆第 4 章的内容。

序列化和抽象基类

如果你尝试使用带有抽象基类的序列化宏，则在编译的时候就会收到一条关于 CreateObject()方法的错误信息。为了解决这个问题，应该定义自己的 IMPLEMENT_SERIAL 宏，就像下面这样：

```
#define IMPLEMENT_SERIAL_ABC(class_name, base_clas_name, wSchema) \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, \
    NULL) \
CArchive&  operator>>(CArchive& ar, class_name* &pOb) \
    { pOb = (class_name*) ar.ReadObject(FRUNTIME_CLASS(class_name)); \
    return ar; }
```

这个新的宏取消了 CreateObject()引用，使你能在一个抽象基类里使用宏。

更新 CRuntimeClass 的状态

前面我们讨论了 CObject 序列化，从而对于 CRuntimeClass 结构的成员有了更多的了解。下面让我们再回顾一下 CRuntimeClass，看看有哪些成员。

- LPCSTR m_lpszClassName——这是类的名字。在序列化过程中，它作为类状态的一部分被写入或读出。
- UNIT m_wSchema——也是类的一个状态。它给出了类的版本信息。
- void Store(CArchive& ar) const——CArchive::WriteClass()调用这个成员函数来写出关于类的 CRuntimeClass 结构信息。
- static CRuntimeClass * Load(CArchive& ar, UINT * pwSchemaNum); ——CArchive::ReadClass()会调用 Load()来读入由 Store()写入的 CRuntimeClass 信息。

到现在为止，只剩下 CRuntimeClass 的两个成员没有讨论。（你可能不会想到挖掘 MFC

的内幕竟然像在考古，不是吗？）

```
int m_nObjectSize;
CRuntimeClass * m_pNextClass;
```

下面让我们看一下 CObject 提供的功能的最后一部分，即对诊断（diagnostic）的支持，并研究一下 MFC 如何使用这两个成员。

CObject 对诊断的支持

CObject 对诊断的支持包括两个层次：基本诊断和高级内存诊断，例如内存泄漏的检测和内存统计。

在看这些特性如何实现之前，我们先看如何使用这些特性。

诊断的输出

诊断输出和旧式的“printf-debugging”一样。MFC 会提供一个类似 printf 的宏 TRACE，还提供一个更复杂的特性，你可以利用这个特性在运行时转储（dump）C++对象的状态。

TRACE 宏

TRACE 宏处理类似于 printf 的输出。要打开跟踪，就必须运行 MFC 的跟踪程序。一旦打开，所有的输出就都会输出到你的 Visual C++调试器窗口。例如：

```
TRACE("At this point, my CPoint is: %d %d\n",mypoint.x,mypoint.y);
```

对象转储

所有从 CObject 派生的 MFC 类都有一个名为 Dump()的成员函数，调用它可以打印类的状态。例如，下面就是我们可能为 CMyClass 实现的转储函数：

```
#ifdef _DEBUG
void CMyClass::Dump(CDumpContext& dc)
{
    //Dump base class first
    CObject::Dump(dc);
    dc << "Type is: " << m_wType << "\n"
    << "Data is: " << m_dwData << "\n"
    << "Point is: " << m_ptMiddle.x << "  "
    << m_ptMiddle.y << "\n";
}
#endif // end _DEBUG
```

注意，通常 Dump()函数只出现在应用程序的调试版本中。

运行时检查

MFC 为在运行时通过断言（assertion）来检查特定的错误状态提供了一些方便的方法。

断言

ASSERT 宏是一个调试版本的宏，你可以用它来检查运行时一些变量的状态。如果被检

测的变量出现错误，它会自动显示一些警告消息框。如果你编过很多 MFC 程序，就会对它很熟悉。图 5-3 显示了一个典型的断言。

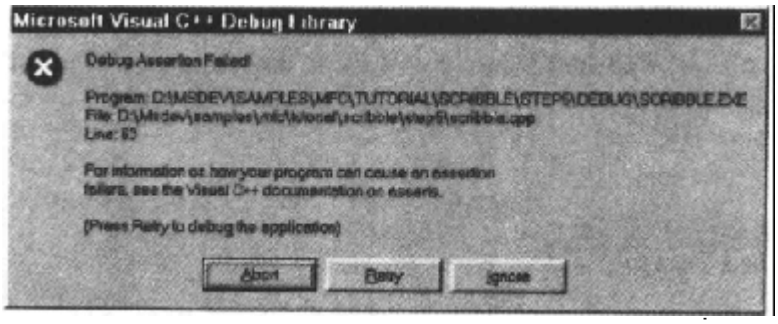


图 5-3 ASSERT 消息框

为了使用 ASSERT 宏，需要传入一个待测试的 Boolean 变量。如果参数的结果是 false，则宏就会弹出对话框警告。否则，这个宏不会有任何动作，并且继续执行下面的语句。

顺便说一下，MFC 自身就使用了超过 5000 个 ASSERT 宏！你可以说微软在调试工具中放入了很多树桩。当然这对你来说也是一个练习。每当你要假设一些事情的时候都可以 ASSERT 任何内部变量的状态。例如你可以在使用一个窗口句柄之前调用 ASSERT 看它是否合法。

对象合法化检查

除了 Dump()成员函数，大多数 CObject 派生类还要实现一个 AssertValid()成员函数。利用这个函数，对象可以在运行时检查成员的合法化。其他对象也可以调用 AssertValid()来确认对象是否处于安全状态。如果一个对象没有处于安全状态，则 AssertValid()应该调用一个断言，尽快通知开发人员。

下面是一个实现 CMyClass::AssertValid()方法的例子：

```
#ifdef _DEBUG
void CMyClass::AssertValid()
{
    //Call inherited first
    CObject::AssertValid();
    ASSERT(m_wType > 0xff00);
    ASSERT(m_dwData != 0);
    ASSERT(m_ptMiddle.x != 0 && m_ptMiddle.y != 0);
}
#endif //End _DEBUG
```

MFC 还提供了一个宏 ASSERT_VALID，你可以对任何 CObject 派生类调用这个宏。这个宏会调用 AssertValid()，还会做一些额外的检查。例如，如果我们有 CMyClass 的一个文档，则以在文档的 AssertValid()成员函数中用下面这个语句：

```
ASSERT_VALID(m_myclass);
```

内存诊断

你可能见过当调试退出时 MFC 程序的输出，就像程序清单 5-10 中所示。

程序清单 5-10 内存诊断输出的示例

```
Detected memory leaks!
Dumping objects ->
{100} a CDialog at $659328

m_hWnd = 0x0m_lpDialogTemplate = 130
m_hDialogTemplate = 0x0
m_pParentWnd = $0
m_nIDHelp = 0x82

{96} a CDialog at $658A10
m_hWnd = 0x0m_lpDialogTemplate = 130
m_hDialogTemplate = 0x0
m_pParentWnd = $0
m_nIDHelp = 0x82

{92} a CDialog at $658454
m_hWnd = 0x0m_lpDialogTemplate = 130
m_hDialogTemplate = 0x0
m_pParentWnd = $0
m_nIDHelp = 0x82

{88} a CDialog at $657B28
m_hWnd = 0x0m_lpDialogTemplate = 130
m_hDialogTemplate = 0x0
m_pParentWnd = $0
m_nIDHelp = 0x82

{84} a CDialog at $657190
m_hWnd = 0x0m_lpDialogTemplate = 130
m_hDialogTemplate = 0x0
m_pParentWnd = $0
m_nIDHelp = 0x82

Object dump complete.
Process 0xFFF35B63 terminated, exit code 0 (0x0).
```

这就是 MFC 的内存泄漏检测在起作用。尽管这是一个很方便的特性，但 MFC 还是提供了更强大的高级内存诊断功能。例如，你可以使用内存诊断来查明内存泄漏，确定程序在生存周期中使用了多少内存，并且打印关于内存使用的统计数据。

检测内存泄漏

使用 CMemoryState 类的 Checkpoint()可以将你要检测的区域包装起来，从而检测内存泄漏情况。然后你就可以使用 Difference()函数来比较两个断点，查看两个调用 Checkpoint()之间是否有泄漏。

程序清单 5-11 显示了使用这一技术的例子：

程序清单 5-11 如何检测内存泄漏

```
//...
#ifdef _DEBUG
    CMemoryState startMemState, stopMemState, diffMemState;
```

```
    startMemState.Checkpoint(); //Start checking here
#endif //End _DEBUG
    CString abc = "Free me!";
//...
#ifdef _DEBUG
    stopMemState.Checkpoint();
    if (diffMemState.Difference(startMemState,stopMemState)) {
        TRACE("DANGER DANGER! Memory leak! ALERT!");
    }
#endif //End _DEBUG
```

检测最大内存使用

通过在 afxMemDF 全局变量中增加按位或（OR）的枚举值 delayFreeMemDF，就可以停止使用内存释放函数（释放和删除）。

例如：

```
afxMemDF |= delayFreeMemDF;
```

这样做，你就可以使用内存统计特性来看你的程序使用了多大内存。

检查内存

如果你想检测所有的内存分配情况和释放情况，只要与 checkAlwaysMemDF 的值进行或操作。例如：

```
afxMemDF |= checkAlwaysMemDF.
```

这样框架就会在分配和释放内存时检测内存。

内存统计

CMemoryState 类还有一个成员函数——DumpStatistics()，它会产生如下的输出：

```
0 bytes in 0 Free Blocks.
52 bytes in 1 Object Blocks.
0 bytes in 0 Non-Object Blocks.
Largest number used: 52 bytes.
Total allocations: 52 bytes.
```

转储所有对象

你还可以利用 CMemoryState 类转储所有泄漏对象的行、地址和大小。这项功能很强大。调用 DumpAllObjectsSince()成员函数会产生类似程序清单 5-12 所示的输出。

程序清单 5-12 所有对象的转储

```
Dumping objects ->
{34} strcore.cpp(78) : non-object block at $00651BE0, 8 bytes long
{31} plex.cpp(28) : non-object block at $00651AD4, 132 bytes long
{30} a CObjfunView object at $00651A78, 52 bytes long
{28} a CMDIChildWnd object at $0065192C, 200 bytes long
{27} plex.cpp(28) : non-object block at $00651880, 132 bytes long
{26} a CObjfunDoc object at $006517F8, 96 bytes long
{25} array_p.cpp(111) : non-object block at $006517BC, 20 bytes long
{24} array_p.cpp(72) : non-object block at $00651790, 4 bytes long
{23} winfrm2.cpp(66) : a CDockBar object at $006516E4, 132 bytes long
```

```
{22} array_p.cpp(72) : non-object block at $006516B8, 4 bytes long
{21} winfrm2.cpp(66) : a CDockBar object at $0065160C, 132 bytes long
{20} array_p.cpp(72) : non-object block at $006515E0, 4 bytes long
{19} winfrm2.cpp(66) : a CDockBar object at $00651534, 132 bytes long
{17} winfrm2.cpp(66) : a CDockBar object at $0065145C, 132 bytes long
{16} bardock.cpp(491) : non-object block at $006513AC, 136 bytes long
{13} plex.cpp(28) : non-object block at $00650FB0, 132 bytes long
{12} strcore.cpp(78) : non-object block at $00650B40, 7 bytes long
{10} a CMainFrame object at $00650920, 456 bytes long
{9} plex.cpp(28) : non-object block at $00650874, 132 bytes long
{8} strcore.cpp(78) : non-object block at $00650818, 49 bytes long
{7} a CMultiDocTemplate object at $00650768, 136 bytes long
{5} strcore.cpp(78) : non-object block at $00650704, 7 bytes long
{4} strcore.cpp(78) : non-object block at $006506C8, 17 bytes long
{3} non-object block at $0065066C, 52 bytes long
{2} non-object block at $00650618, 44 bytes long
```

CObject 的诊断支持内幕

我们对如何使用CObject诊断支持已经非常了解了，下面看它在MFC内部是如何实现的。随着序列化的出现，内部有很多工作需要完成。

在我们开始之前，先说一些关于MFC版本的事情。在MFC 4.0中，很多检查被移植到了C运行库中，这使得诊断在所有的malloc和free中被应用得更广泛。在MFC 3.x或更低的版本中，诊断只是针对MFC对象的。在合适情况下，会标注一些针对版本的说明。

首先，我们会看简单的诊断是如何实现的。然后我们看MFC如何实现高级诊断。

本节中提到的大多数宏和函数都定义在文件 INCLUDE\AFX.H 中。程序清单在AFXMEM.CPP或AFXTLS.CPP中。

诊断输出

我们首先从AFX.H中的TRACE宏声明开始：

```
#define TRACE           ::AfxTrace
```

可以看出它只是调用了全局函数 AfxTrace()。AfxTrace()的实现在文件 DUMPOUT.CPP中，如程序清单5-13所示。

程序清单5-13 AfxTrace()的实现（在文件DUMPOUT.CPP中）

```
void AfxTrace(LPCTSTR lpszFormat, ...)
{
#ifdef _DEBUG // all AfxTrace output is controlled by afxTraceEnabled
    if (!afxTraceEnabled)
        return;
#endif
    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);
}
```

afxTraceEnabled是一个Boolean变量，它的定义位于文件AFX.H中，文件DUMPINTT.CPP中有一个静态对象_AFX_DEBUG_STATE，afx TraceEnabled的初始化就是在这个静态对象的构造函数中完成的。因为它是一个静态对象，所以它的构造函数会在应用程序启动运行之前被调用。

```
_AFX_DEBUG_STATE::_AFX_DEBUG_STATE()
{
    afxTraceEnabled =
     ::GetPrivateProfileInt(szDiagSection, szTraceEnabled,
     !afxData.bWin31,szIniFile);
    afxTraceFlags =
      ::GetPrivateProfileInt(szDiagSection, szTraceFlags,0, szIniFile);
    //...
}
```

在DUMPINIT.CPP的前面，你会发现下面这段GetPrivateProfileInt调用的声明：

```
static const TCHAR szIniFile[] = _T("AFX.INI");
static const TCHAR szDiagSection[] = _T("Diagnostics");
static const TCHAR szTraceEnabled[] = _T("TraceEnabled");
static const TCHAR szTraceFlags[] = _T("TraceFlags");
```

所以afxTraceEnabled和afxTraceFlags两个全局变量只是从AFX.INI这个配置文件中读出的。MFC的跟踪应用程序所做的就是确定文件中的值。

一旦TRACE确定它应该检查afxTraceEnabled并产生输出时，它就会遍历变量表，将vstprintf()（它提供了所有的printf功能）的结果输出给afxDump。

下面让我们看一下afxDump，弄清楚它如何处理所有的<<操作符，并将输出传送到你的VC++调试窗口中。

afxDump和CDumpContext

afxDump的声明在文件AFX.H中，内容如下：

```
#ifdef _DEBUG
    extern AFX_DATA CDumpContext afxDump;
#endif //End _DEBUG
```

CDumpContext辅助类的声明也在AFX.H中，如程序清单5-14所示。

程序清单5-14 CDumpContext类的声明

```
class CDumpContext
{
public:
    CDumpContext(CFile* pFile = NULL);
// Attributes
    int GetDepth() const;       // 0 => this object, 1 => children objects
```

```
    void SetDepth(int nNewDepth);
// Operations
    CDumpContext& operator<<(LPCTSTR lpsz);
    CDumpContext& operator<<(LPCSTR lpsz);  // automatically widened
    CDumpContext& operator<<(const void* lp);
    CDumpContext& operator<<(const CObject* pOb);
    CDumpContext& operator<<(const CObject& ob);
    CDumpContext& operator<<(BYTE by);
    CDumpContext& operator<<(WORD w);
    CDumpContext& operator<<(UINT u);
    CDumpContext& operator<<(LONG l);
    CDumpContext& operator<<(DWORD dw);
    CDumpContext& operator<<(float f);
    CDumpContext& operator<<(double d);
    CDumpContext& operator<<(int n);
    void HexDump(LPCTSTR lpszLine, BYTE* pby, int nBytes, int nWidth);
    void Flush();
// Implementation
protected:
    // dump context objects cannot be copied or assigned
    CDumpContext(const CDumpContext& dcSrc);
    void operator=(const CDumpContext& dcSrc);
    void OutputString(LPCTSTR lpsz);
    int m_nDepth;
public:
    CFile* m_pFile;
};
```

CDumpContext 为 C++和 Windows 中类似 CArchive 的类型定义了操作符，这样你可以对这些类型使用<<操作符。我们先看 CDumpContext 的一个操作符，看它是如何实现的，以便了解这个类的功能。下面是 WORD 类型的操作符：

```
CDumpContext& CDumpContext::operator<<(WORD w)
{
    TCHAR szBuffer[32];
    wsprintf(szBuffer, _T("%u"), (UINT) w);

    OutputString(szBuffer);
    return *this;
}
```

所有的操作符都很类似，它们都调用了 wsprintf()来将值以字符串的形式输出，然后再将字符串传递给 CDumpContext::OutputString()，由这个函数完成真正的输出。OutputString 的具体实现在文件 DUMPCONT.CPP 中，如程序清单 5-15 所示。

程序清单 5-15　CDumpContext::OutputString()的伪代码（在文件 DUMPCONT.CPP 中）

```
void CDumpContext::OutputString(LPCTSTR lpsz)
{
#ifdef _DEBUG
    // all CDumpContext output is controlled by afxTraceEnabled
    if (!afxTraceEnabled)
        return;
#endif
    // use C-runtime/OutputDebugString when m_pFile is NULL
```

```
    if (m_pFile == NULL){
        AfxOutputDebugString(lpsz);
        return;
    }
    // otherwise, write the string to the file
    m_pFile->Write(lpsz, lstrlen(lpsz)*sizeof(TCHAR));
}
```

创建了全局变量 afxDump 之后，m_pFile 是 NULL（默认值），所以它将自己的所有输出传递给 AfxOutputDebugString()函数。如果你创建了自己的 CDumpContext，并在构造函数中传递给它一个 CFile 指针，则 OutputString()会将字符串写到文件中，这样你就可以得到一个输出的日志。

版本注释

在 MFC 3.x 和更低的版本中，CDumpContext::OutputString()会调用全局函数 OutputString()，而不是调用 AfxOutputDebugString()。没有 OutputDebugString()的程序清单，因为它是一个 Windows API。

AfxOutputDebugString()的声明位于文件 AFX.H 中，它实际上是一个宏：

```
#ifdef _UNICODE
#define AfxOutputDebugString(lpsz) \
    do \
        { \
    int _convert; _convert = 0; \
    _RPT0(_CRT_WARN, W2CA(lpsz)); \
    } while (0)
#else
    #define AfxOutputDebugString(lpsz) _RPT0(_CRT_WARN, lpsz)
#endif
```

AfxOutputDebugString()调用 C 运行时库中的内容，所以我们不得不假设在 C 运行时库中的底层某处，微软将 CDumpContext 的输出送给调试窗口。

对象转储

现在我们已经理解了 TRACE 宏的功能，并且知道了它如何使用 CDumpContext 辅助类以及如何从 profile 字符串中初始化之后，下面我们就看一下对象转储的内部机制。

再看一下程序清单 5-1 中 CObject 的声明。你会发现 CObject 有一个虚拟的 Dump()函数，你可以重载它。下面是文件 OBJCORE.CPP 中 CObject::Dump()的实现：

```
#ifdef _DEBUG
void CObject::Dump(CDumpContext& dc) const
{
    dc << "a " << GetRuntimeClass()->m_lpszClassName << " at " <<
          (void*)this << "\n";
}
#endif //_DEBUG
```

CObject::Dump()只是转储了类名（存储在 CRuntimeClass 类中）和 this 指针的地址。Dump()输出还使用了 TRACE 宏所使用的 CDumpContext 操作符。

AssertValid()和断言

下面看 MFC 如何处理断言。特别地，我们会看一下 AssertValid()运行时的对象检查功能是如何实现的。

ASSERT 宏的声明位于 AFX.H 中：

```
#define ASSERT(f) \
    do \
        { \
        if (!(f) && AfxAssertFailedLine(THIS_FILE, __LINE__)) \
        AfxDebugBreak(); \
    } while (0) \
```

AfxAssertFailedLine()全局函数在文件 OBJCORE.CPP 中。它首先阻塞所有活动的线程，然后显示熟悉的消息框，上面有文件名和行数号。

现在你知道 ASSERT 都完成了些什么，你就可以在启动调试器之前在 AfxAssertFailedLine()那里设置一个断点。因为这样在调试中会在消息框弹出之前将断言以断点方式报告，同时在这里设置一个断点会使得调试类似于 WM_ACTIVATE 的情况（以及某些多线程情况）变得更简单。

你可能在 MFC 的源文件中见过这样的声明：

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

它为当前的文件名创建了空间，并将文件名存放进去。编译器会自动为你替代_FILE_和_LINE_。

当 ASSERT 宏调用 AfxDebugBreak()时，控制会返回给 Visual C++的调试器，这样你就可以交互式地调试断言。

AssertValid()

和 Dump()一样，AssertValid()在 CObject 中也被声明为虚函数。CObject::AssertValid()的实现可以在文件 OBJCORE.CPP 中找到。它只是简单地验证了 this 指针不是 NULL。

```
void CObject::AssertValid() const
{
    ASSERT(this != NULL);
}
```

让我们再看看 ASSERT_VALID 宏，看它还完成了什么。它的定义如下：

```
#define ASSERT_VALID(pOb)  (::AfxAssertValidObject(pOb, THIS_FILE, __LINE__))
```

AfxAssertValidObject()全局函数位于 OBJCORE.CPP 中。程序清单 5-16 列出了它的伪代码：

程序清单 5-16　AfxAssertValidObject()的伪代码

```
void AfxAssertValidObject(const CObject* pOb,LPCSTR lpszFileName, int nLine)
{
    if (pOb == NULL) {
        TRACE0("ASSERT_VALID fails with NULL pointer.\n");
        if (AfxAssertFailedLine(lpszFileName, nLine))
            AfxDebugBreak();
```

```
            return;     // quick escape
        }
        if (!AfxIsValidAddress(pOb, sizeof(CObject))) {
            TRACE0("ASSERT_VALID fails with illegal pointer.\n");
            if (AfxAssertFailedLine(lpszFileName, nLine))
                AfxDebugBreak();
            return;     // quick escape
        }
        // check to make sure the VTable pointer is valid
        ASSERT(sizeof(CObject) == sizeof(void*));
        if (!AfxIsValidAddress(*(void**)pOb, sizeof(void*), FALSE))    {
            TRACE0("ASSERT_VALID fails with illegal vtable pointer.\n");
            if (AfxAssertFailedLine(lpszFileName, nLine))
                AfxDebugBreak();
            return;     // quick escape
        }
        if (!AfxIsValidAddress(pOb, pOb->GetRuntimeClass()->m_nObjectSize)) {
            TRACE0("ASSERT_VALID fails with illegal pointer.\n");
            if (AfxAssertFailedLine(lpszFileName, nLine))
                AfxDebugBreak();
            return;     // quick escape
        }
        pOb->AssertValid();
    }
```

首先，AfxAssertValidObject()检查对象是否是NULL，然后再确认对象的大小是否正确以及它的地址是否合法，最后它要确认对象的大小和CRuntimeClass::m_nObjectSize中存储的信息是否一致，并且调用对象的AssertValid()成员函数。对象可以有自定义的状态信息。因为AssertValid()是虚函数，所以为了断言特定于你的对象的情况，你必须覆盖它。

虽然这个宏开销很大，但在你调试时会知道它仍然很有用。如果对象用不同的测试值执行这个函数，你就可以确定问题不在于对象。

在ASSERT_VALID中，AfxIsValidAddress()看起来是MFC内存检查的关键，下面让我们看看在你的MFC程序中如何使用它。

AfxIsValidAddress()

这个辅助函数的定义在AFX.H中：

```
BOOL AfxIsValidAddress(const void* lp, UINT nBytes, BOOL bReadWrite = TRUE);
```

它的实现位于文件VALIDADD.CPP中：

```
BOOL AfxIsValidAddress(const void* lp, UINT nBytes, BOOL bReadWrite)
{
    return (lp != NULL && !IsBadReadPtr(lp, nBytes) &&
        (!bReadWrite || !IsBadWritePtr((LPVOID)lp, nBytes)));
}
```

这个函数使用基本的Win32 API（IsBadReadPtr()和IsBadWritePtr()）检查指针。它还用bReadWrite参数（默认情况下是TRUE）来检查指针是否可写。

如果你仔细看MFC的程序清单，你会发现对AfxIsValidAddress()的调用多达400次，足以说明它是一个非常重要的函数。记得在你的程序中使用这个函数。

通常情况下，MFC会用如下的ASSERT宏包装AfxIsValidAddress()：

```
ASSERT(AfxIsValidAddress(lpLogFont, sizeof(LOGFONT), FALSE));
```

使用 AfxIsValidAddress()来测试传递给成员函数的每个参数指针，从而确保开发人员没有传入一个垃圾指针。和 ASSERT 宏一样，使用 AfxIsValidAddress()也会降低你编译链接的速度，但是这些宏在发布版本中都是编译好的。

它包装了简单的输出诊断。下面让我们看 MFC 如何实现高级内存诊断，例如内存泄漏的检测。

高级内存诊断

版本注释

内存诊断的实现在 MFC 3.x 和 4.x 中有很大不同。正如前面提到的，4.x 版本中大多数都使用了 C 运行时库的函数，而 MFC 3.x 则使用了自己的函数。

在本节，我们会分析 MFC 3.x 和 MFC 4.x，以便帮助你理解使用 C 运行时库时都发生了什么。

CMemoryState

当使用高级诊断时，你可能会注意到另外一个名为 CMemoryState 的辅助类。这个辅助类也是没有文档说明的。程序清单 5-17 中给出了它的声明，以及一些关键的成员函数，下面看看这个辅助类是如何运作的。

程序清单 5-17 CMemoryState 辅助类的声明

```
struct CMemoryState
{
// Attributes
    enum blockUsage {
        freeBlock,    // memory not used
        objectBlock,  // contains a CObject derived class object
        bitBlock,     // contains ::operator new data
        crtBlock,     // New in MFC 4.
        ignoredBlock, // New in MFC 4.
        nBlockUseMax  // total number of usages
    };
    _CrtMemState m_memState;
    //Note: MFC 3.x had: CBlockHeader * m_pBlockHeader; instead
    LONG m_lCounts[nBlockUseMax];
    LONG m_lSizes[nBlockUseMax];
    LONG m_lHighWaterCount;
    LONG m_lTotalCount;
    CMemoryState();
// Operations
    void Checkpoint();  // fill with current state
    BOOL Difference(const CMemoryState& oldState,
    const CMemoryState& newState);  // fill with difference
    void UpdateData();
// Output to afxDump
    void DumpStatistics() const;
    void DumpAllObjectsSince() const;
};
```

版本注释

因为 MFC 4.0 将 CMemoryState 的大多数功能放入 C 运行时库，所以对 CMemoryState 的解释就是基于 MFC 3.x 代码的，MFC 3.x 的代码更好理解。

CMemoryState 记录的每个已分配的内存块是以下几个之一：

- freeBlock——由于使用 delayFreeMemDF 而导致释放被延迟的块。
- objectBlock——保留的或者已经泄漏的 CObject 对象的数量。
- bitBlock——已分配内存的非 CObject 对象的个数。
- crtBlock（只在 MFC 4.0 中）——已分配的 C 运行时块的个数。
- ignoredBlock（只在 MFC 4.0 中）——MFC 检查所忽略的块的个数。

CMemoryState 将分配的次数保存在 m_lCounts 数组中，将已分配的各种类型的大小存放在 m_lSizes 成员数组中。

m_lHighWaterCount 成员存储了最大分配数，m_lTotalCount 记录了分配的总次数。

m_memState 和 m_pBlockHeader 成员变量都指向了一个内存块链表的头节点。

我们已经知道 CMemoryState 所存储的内容，最大的问题就是如何让 CMemoryState 知道所有的对象分配？MFC 通过 C++中的覆盖操作符功能记录了所有对象的分配和释放。

调试 new 操作符

你在 ClassWizard 生成的源文件中会发现以下几行代码：

```
#ifdef _DEBUG
#define new DEBUG_NEW
#endif
```

定义_DEBUG 会将你的所有 new 操作符重定向到调用 DEBUG_NEW。

下面是定义在 AFX.H 中的 DEBUG_NEW：

```
#define DEBUG_NEW new(THIS_FILE, __LINE__)
```

这个宏会使得调试版本的代码中使用不同的 new 操作符，这个操作符就记录了所有的内存分配情况，并存储了文件名和代码行。

下面是被重载的全局 new 操作符的新声明：

```
void* operator new(size_t nSize, LPCSTR lpszFileName, int nLine);
```

从程序清单 5-1 中，你会想起 CObject 定义了几个 new 操作符和 delete 操作符。为了理解为什么使用了两个不同的 new 操作符（一个是全局的，另一个是专门为 CObject 使用的），下面让我们看这两个操作符的实现。这些实现位于文件 AFXMEM.CPP 中。程序清单 5-18 是这两个操作符的伪代码。

程序清单 5-18 两个 new 操作符（你能发现它们的区别吗？）

```
void*  operator new(size_t nSize, LPCSTR lpszFileName, int nLine)
{
    // memory corrupt before global new
    if (afxMemDF & checkAlwaysMemDF)
        ASSERT(AfxCheckMemory());
    void* p = AfxAllocMemoryDebug(nSize, FALSE, lpszFileName, nLine);
    if (p == NULL) {
```

```
        TRACE1("::operator new(%u) failed - throwing exception.\n", nSize);
        AfxThrowMemoryException();
    }
    return p;
}
void* CObject::operator new(size_t nSize, LPCSTR lpszFileName, int nLine)
{
    // memory corrupt before 'CObject::new'
    if (afxMemDF & checkAlwaysMemDF)
        ASSERT(AfxCheckMemory());
    void* p = AfxAllocMemoryDebug(nSize, TRUE, lpszFileName, nLine);
    if (p == NULL) {
        TRACE1("CObject::operator new(%u) failed - throwing exception.\n",
            nSize);
        AfxThrowMemoryException();
    }
    return p;
}
```

这两个操作符的区别很小，你发现了吗？下面是3条暗示：

1. 看一下对AfxAllocMemoryDebug()的调用。
2. 注意第二个参数。
3. 参数一个是TRUE，另一个是FALSE。

我们知道对于debug版本的代码有专门的new操作符，下面就让我们看看这个操作符是如何存储内存分配信息的。我们会检查全局辅助函数AfxAllocMemoryDebug()。

程序清单5-19是这个辅助函数的伪代码。

程序清单5-19 AfxAllocMemoryDebug()的伪代码

```
void* AfxAllocMemoryDebug(size_t nSize, BOOL bIsObject, LPCSTR
lpszFileName, int nLine)
{
    if (nSize == 0)
        TRACE("Warning: Allocating zero length memory block.\n");
    if (nSize > (size_t)SIZE_T_MAX - nNoMansLandSize -
            sizeof(CBlockHeader)) {
        TRACE1("Error: memory allocation: tried to allocate %u bytes --\n",
            nSize);
        TRACE0("\tobject too large or negative size.\n");
    AfxThrowMemoryException();
    }
   // keep track of total amount of memory allocated
   lTotalAlloc += nSize; lCurAlloc += nSize;
   if (lCurAlloc > lMaxAlloc)
       lMaxAlloc = lCurAlloc;
   struct CBlockHeader* p =
      (struct CBlockHeader*) malloc(sizeof(CBlockHeader) + nSize +
          nNoMansLandSize);
   if (p == NULL)
       return NULL;
   LockDebugMemory();  // block other threads
   if (pFirstBlock)
       pFirstBlock->pBlockHeaderPrev = p;
```

```
    p->pBlockHeaderNext = pFirstBlock;
    p->pBlockHeaderPrev = NULL;
    p->lpszFileName = lpszFileName;
    p->nLine = nLine;
    p->nDataSize = nSize;
    p->use = bIsObject ? CMemoryState::objectBlock : CMemoryState::bitBlock;
    p->lRequest = lRequest;
    // fill in gap before and after real block
    memset(p->gap, bNoMansLandFill, nNoMansLandSize);
    memset(p->pbData() + nSize, bNoMansLandFill, nNoMansLandSize);
    // fill data with silly value (but non-zero)
    memset(p->pbData(), bCleanLandFill, nSize);
    // link blocks together
    pFirstBlock = p;
    UnlockDebugMemory();    // release other threads
    return (void*)p->pbData();
}
```

这个复杂的函数是理解 MFC 记录内存方式的关键。下面让我们按步骤分解 AfxAllocMemoryDebug()。

1. AfxAllocMemoryDebug()首先确定大小不是负数，而且不是特别大。

2. 然后，AfxAllocMemoryDebug()更新了总共分配计数，如果需要的内存块的大小大于上一次分配的内存块，最大分配计数也需要更新。

3. 然后 AfxAllocMemoryDebug()会分配内存，注意调用 malloc()时使用的大小：

```
malloc(sizeof(CBlockHeader) + nSize + nNoMansLandSize);
```

MFC 会分配比需要更大的内存，并在内存块前面增加一个 CBlockHeader 结构，这样就可以跟踪这块内存。在 CBlockHeader 结构中，最后一项是字节缝隙。MFC 还在数据后面留下一些空字节，称为“无主地”。MFC 在实际数据的两端使用这个空间来写一些不同寻常的内存模式，也可以叫做“哨（sentinel）”。通过检查“哨”，MFC 可以迅速地判断出程序是否对已分配内存进行了下溢写操作或上溢写操作，从而关于有关问题警告用户。

图 5-4 用图形的形式显示了 MFC 如何分配内存。

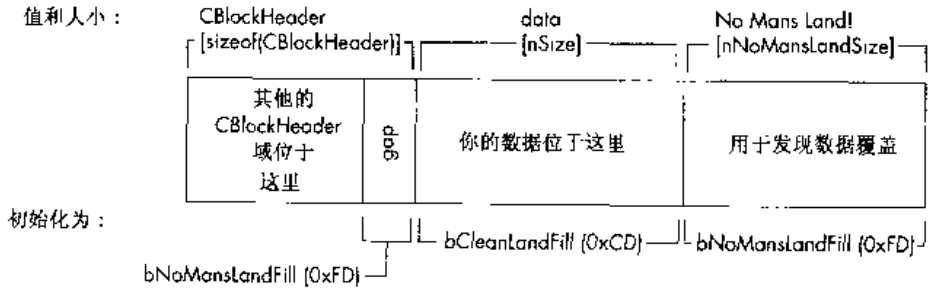


图 5-4 在调试模式下 MFC 如何分配内存

4. 在分配内存之后，AfxAllocMemoryDebug()会锁定所有的其他线程。

5. 接下来，它将当前的内存块加入到块的链表（pFirstBlock->pBlockHeaderPrev = p）中。

6. 然后将所有关于内存的信息写入 CBlockHeader 中，包括文件名、行号、大小，并根据 bIsObject 参数（请再看一下全局变量和 CObject 调试版本的 new 操作符）来确定它是一个

对象还是按位分配。

7. 然后你会看到对 memset()的两次调用，它将前面第 3 步提到的“哨”的值拷贝到内存块的前面和后面。

8. 下一个 memset()调用将已分配的内存中的值都置为非 0，这样你就可以确认你的程序没有做关于内存初始化为 0 的假设。

9. 最后，函数会返回指向内存块（CBlockStructure 之外）的指针。这样你的程序就可以继续运行了。

回到 CMemoryState

我们理解了 MFC 如何存储内存分配情况，下面让我们看一下 CMemoryState 的一些重要成员函数是如何实现的。

首先，我们看 CMemoryState::Checkpoint()的代码（在文件 AFXMEM.CPP 中），如程序清单 5-20 所示。

程序清单 5-20 CMemoryState::Checkpoint()的实现（在文件 AFXMEM.CPP 中）

```
void CMemoryState::Checkpoint()
{
    if (!(afxMemDF & allocMemDF))
        return;     // can't do anything
    LockDebugMemory();  // block other threads
    m_pBlockHeader = pFirstBlock;
    for (int use = 0; use < CMemoryState::nBlockUseMax; use++)
        m_lCounts[use] = m_lSizes[use] = 0;
    struct CBlockHeader* p;
    for (p = pFirstBlock; p != NULL; p = p->pBlockHeaderNext) {
        if (p->lRequest == lNotTracked)
            // ignore it for statistics
        else if (p->use >= 0 && p->use < CMemoryState::nBlockUseMax){
            m_lCounts[p->use]++;
            m_lSizes[p->use] += p->nDataSize;
        }
        else{
            TRACE1("Bad memory block found at $%08lX.\n", (BYTE*)p);
        }
    }
    m_lHighWaterCount = lMaxAlloc;
    m_lTotalCount = lTotalAlloc;
    UnlockDebugMemory();    // release other threads
}
```

CMemoryState::Checkpoint()首先初始化了两个计数数组 m_lCounts 和 m_lSizes。然后 CheckPoint()遍历内存块的链表，并在 m_lCounts 数组和 m_lSize 数组的恰当目录中添加内存大小和计数。最后，它根据全局变量 lMaxAlloc 和 lTotalAlloc 中设置的值（如果有新的最大值就会由 AfxAllocMemoryDebug()设置）更新了 m_lHighWaterCount 和 m_lTotalCount 成员。

下面我们看一下 CMemoryState::Difference()，看它如何使用存储在两个含有断点的 CMemoryState 对象的信息确定内存是否泄漏。程序清单 5-21 给出了伪代码。

程序清单 5-21 CMemoryState::Difference()

```
// fills 'this' with the difference, returns TRUE if significant
BOOL CMemoryState::Difference(const CMemoryState& oldState,
                              const CMemoryState& newState)
{
    BOOL bSignificantDifference = FALSE;
    for (int use = 0; use < CMemoryState::nBlockUseMax; use++){
        m_lSizes[use]  = newState.m_lSizes[use] - oldState.m_lSizes[use];
        m_lCounts[use] = newState.m_lCounts[use] - oldState.m_lCounts[use];
        if ((m_lSizes[use] != 0 || m_lCounts[use] != 0) &&
           use != CMemoryState::freeBlock)
                bSignificantDifference = TRUE;
    }
    m_lHighWaterCount = newState.m_lHighWaterCount -
        oldState.m_lHighWaterCount;
    m_lTotalCount     = newState.m_lTotalCount - oldState.m_lTotalCount;
    return bSignificantDifference;
}
```

对于每个内存目录，Difference()会从每个 CMemoryState 参数中减去积累的值，再存入 CMemoryState 的 m_lSizes 数组和 m_lCounts 数组中。如果差值不是 0，而且目录不是 CMemoryState::freeBlock，就会标记出不同，然后返回给调用者 TRUE，表示发现了问题。

既然对 CMemoryState 已经有了感觉，你就可以看程序清单 5-22 中 DumpStatistics()的实现了。

程序清单 5-22 CMemoryState::DumpStatistics()的实现

```
void CMemoryState::DumpStatistics() const
{
    for (int use = 0; use < CMemoryState::nBlockUseMax; use++){
        TRACE3("%ld bytes in %ld %hs Blocks.\n", m_lSizes[use],
              m_lCounts[use], blockUseName[use]);
    }
    TRACE1("Largest number used: %ld bytes.\n", m_lHighWaterCount);
    TRACE1("Total allocations: %ld bytes.\n", m_lTotalCount);
}
```

CMemoryState::DumpAllObjectsSince()会遍历 CBlockHeader 结构链表，检查所有的内存地址，并产生和对象相应的输出。这些信息包括文件名和分配所位于的行号。

AfxCheckMemory()

你可能想知道 MFC 在何处检查这些由 AfxAllocMemoryDebug()放置的“哨”。AfxCheckMemory()全局辅助函数的任务就是检查这些“哨”，它会针对每个发现的问题调用 TRACE。如果你正在调试，而且你怀疑某些内存被重写了，就可以调用 AfxCheckMemory()函数来快速地查找出错误所在。

AfxDumpMemoryLeaks()

你可能还想知道 MFC 在何处以及如何生成内存泄漏链表。在 MFC 4.x 中，这些都是由 C 运行时退出模块完成的，这个模块会在退出时自动调用_CrtDumpMemoryLeaks()。

在 MFC 3.x 中，要调用的代码位于文件 WINMAIN.CPP 中的 CWinApp::Run()之后，这

段代码通过调用全局辅助函数 AfxDumpMemoryLeaks()而被调用。程序清单 5-23 是 MFC 3.x 中的 AfxDumpMemoryLeaks()函数。

程序清单5-23 MFC 3.x中的AfxDumpMemoryLeaks(),相当于MFC 4.x中的_CrtDumpMemoryLeaks()

```
BOOL AfxDumpMemoryLeaks()
{
    // only dump leaks when there are in fact leaks
    CMemoryState msNow;
    msNow.Checkpoint();
    if (msNow.m_lCounts[CMemoryState::objectBlock] != 0 ||
      msNow.m_lCounts[CMemoryState::bitBlock] != 0){
        // dump objects since empty state since difference detected.
        TRACE0("Detected memory leaks!\n");
        afxDump.SetDepth(1);     // just 1 line each
        CMemoryState msEmpty;    // construct empty memory state object
        msEmpty.DumpAllObjectsSince();
        return TRUE;
    }
    return FALSE;    // no leaked objects
}
```

在 MFC 4.x 中，AfxDumpMemoryLeaks()只是调用了_CrtDumpMemoryLeaks()。但是你可以在程序启动之后的任何时刻调用它来查看是否发生了内存泄漏。

AFX 辅助函数

在调试时可以使用两个有趣的全局 AFX 辅助函数：AfxDoForAllObjects()和 AfxDoForAllClasses()。

AfxDoForAllObjects()

AfxDoForAllObjects()带有两个参数，一个是指向一个函数的指针，另一个是 void*指针。AfxDoForAllObjects()遍历内存对象链表，并将它们作为 CObject 指针传递给用户提供的函数，其中该函数的另一个参数就是 AfxDoForAllObjects()的另一个参数。在 MFC 3.x 中，这个函数会遍历 CBlockHeader 链表。在 MFC 4.x 中，它完全由 C 运行时函数实现。它的代码位于文件 AFXMEM.CPP 中。

AfxDoForAllClasses()

AfxDoForAllClasses()和 AfxDoForAllObjects()很类似，只是它会遍历 CRuntimeClass 结构，然后将这些结构传递给一个函数。下面是它的实现：

```
void AfxDoForAllClasses(void (*pfn)(const CRuntimeClass*, void*),
                        void* pContext)
{
    // just walk through the simple list of registered classes
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();

    AfxLockGlobals();
    for (CRuntimeClass* pClass = pModuleState->m_classList; pClass != NULL;
      pClass = pClass->m_pNextClass)
        (*pfn)(pClass, pContext);
}
```

CRuntimeClass 结构的最后一个成员出现了，那就是 m_pNextClass。MFC 通过 AFX_CLASSINT 结构的构造函数将对象存储在这个链表中。AFX_CLASSINT 的构造函数是由前面提到的 DECLARE/IMPLEMENT 宏生成的。

组合在一起

我们已经知道了 CObject 的所有特性，下面就回到前面 CObject 的声明，来了解为什么 CObject 按这种方式组合，以及那些成员都做了些什么。

为了帮助将这些特性组合在一起，我们快速地浏览一下 CRuntimeClass 结构中每个成员变量的作用：

- m_lpszClassName——在序列化时使用，用来存储类的状态。
- m_nObjectSize——调试实用函数 AfxAssertValidObject()用该变量来验证类的大小与它的 CRuntimeClass 等价物匹配。当内存中出现问题时，这一点可能不能保证。
- m_wSchema——序列化用这个 WORD 类型的变量给类一个版本号。如果不支持序列化，m_wSchema 的值就是 0xffff。你可以通过 IMPLEMENT_SERIAL 宏指定自己的版本（第三个参数），或者在运行时修改它。
- m_pfnCreateObject——用于动态创建。DECLARE/IMPLEMENT_DYNCREATE 宏会将这个成员函数指针指向 CMyClass::CreateObject()成员函数。当你调用 CRuntimeClass::CreateObject()时，它会通过 m_pfnCreateObject 调用 CMyClass::CreateObject()。CMyClass::CreateObject()会返回一个新的 CMyClass。
- m_pBaseClass——保存一个指向基类的 CRuntimeClass 结构的指针。IsKindOf()和 IsDerivedFrom()会用它来确定对象的“多态类型”。
- m_pNextClass——维护一个已经创建的对象的简单链表。

投入使用

前面提到的 CObject 特性都很有趣，但是是否有用呢？答案当然是有用。在调试下你会发现它有多有用。你可以感觉到打印出前面那些没有文档说明的辅助类的成员并理解它们会很舒服。你还可以使用本章中提到的其他信息。

例如，我们已经实现了一个调试对话框。这个对话框显示了 CRuntimeClass 链表中的所有类，以及所有当前创建的对象。图 5-5 显示了一个标准的文档/视图 MDI 应用程序的类链表。图 5-6 中显示了同一应用程序在某一时刻的所有对象。

这个例子的代码在 www.infopower.com.cn 中，位于目录 chap05 下，你可以看它是如何实现的，并根据自己的需要修改代码。

是否值得

现在我们已经知道从 CObject 那里能继承到什么特性，接下来就让我们再回到本章开始

的那个问题：“MFC 使用这种以 CObject 为根的结构是否会导致性能问题？”

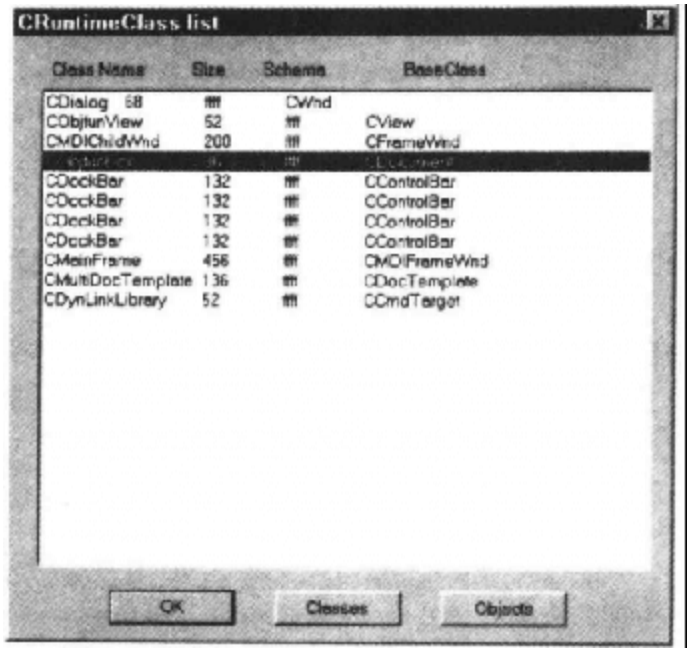


图 5-5　在你的应用程序中访问 CRuntimeClass 链表的例子

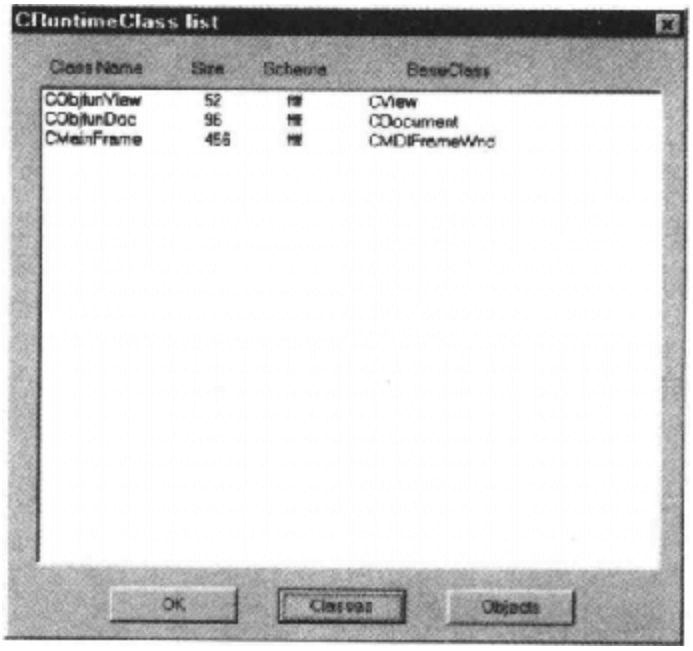


图 5-6　在运行期间某一时刻所有对象的快照（snapshot）

C++必须在运行时索引类的虚函数表才能确定要执行哪个函数。我们在前面提到 CObject 有 5 个虚函数，其中 2 个专门在调试时使用，所以它们不会影响发布版本的性能。使用虚函数的最大开销是在虚函数表中增加的实例大小。如果你打算使用虚函数，CObject 不会增加任何额外的开销。

因此，从 CObject 派生的权衡就在于，在类的虚表中增加三个额外的虚函数与另外为

RTCI、动态创建、序列化和内存诊断增加虚函数之间选择。当然是否使用这些函数由你来决定。MFC 只有极少的类不是从 CObject 派生的（第 4 章提到的）。这些类多数都很小而且没有虚函数。

结 语

到此为止，你已经能够打开非 GUI 的 MFC 头文件 AFX.H，并能理解其中的每个类、宏和辅助函数。

现在可以尝试复习已学的知识，并为学习下一章做准备。下一章我们要学习 MFC 如何实现对话框、控制栏、文档/视图以及一些高级用户界面类（例如分割窗口）。

CHAPTER

6 MFC 对话框和控件类

在这一章我们会挖掘 MFC 对话框和控件类的内幕。因为所有的这些类都是从 CObject、CCmdTarget 和 CWnd 派生的，所以你要确定在学习这一章内容之前你已经对前面介绍的 MFC 内幕掌握得很好。

首先，我们先了解 MFC 如何实现 Windows 对话框，以及 MFC 如何进行动态数据交换和数据确认（缩写为 DDX/DDV）。然后我们研究 MFC 如何包装 Windows 的一般对话框和 OLE 对话框。然后，我们看 MFC 的属性页和标签式对话框。

在分析完关于 MFC 对话框的内幕之后，我们再查看 MFC 控件类。

CDialog：模态 MFC 对话框和非模态 MFC 对话框

MFCCDialog 类既支持模态对话框（modal dialog）又支持非模态对话框（modeless dialog）。在本节，我们将简要地浏览如何使用 CDialog，然后再看 MFC 如何实现这个类。我们还会看到 MFC 如何初始化、交换以及确认对话框数据。

使用 CDialog

为了使用 CDialog 类，你需要创建 CDialog 的一个派生类（通常使用类向导），使得它成为你所希望的对话框。通常你要将 CDialog 派生类和一个对话框资源通过一个整数对话框 ID 或一个资源对话框名字联系在一起。图 6-1 显示了如何使用类向导完成这些。

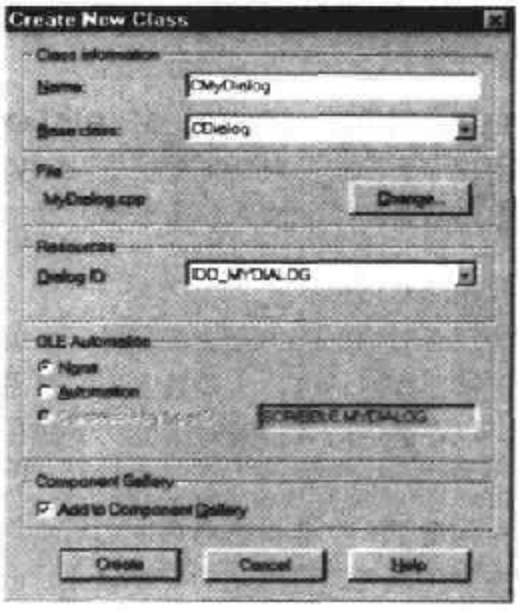


图 6-1 类向导

CDialog 类支持模态对话框和非模态对话框的创建和操作。模态对话框会“冻结”应用程序的其他部分，强制用户必须完成一些操作。模态对话框的一个很好的例子就是通用颜色对话框（common color dialog box）。

一旦对话框窗口出现在屏幕上，你惟一可以关闭它的办法就是按“OK”按钮。对于非模态的对话框，用户可以在对话框显示的同时继续使用应用程序。通常用户必须要手动地关闭对话框。非模态对话框的一个很好的例子就是 Word 中的拼写检查对话框（spell check dialog box）。下面让我们看如何在 MFC 中使用模态 CDialog 和非模态的 CDialog。

历史注释

追溯到 MFC 1.0，共有两个对话框类：CDialog 和 CModalDialog。在 MFC 1.0 之后的版本中，MFC 将这两个合并成一个类 CDialog。为了保持向后的兼容性，MFC 定义了这样的#define：

```
#define CModalDialog CDialog
```

在文件 AFXWIN.H 中，这样对于任何使用了 CModalDialog 的 MFC 1.0 应用程序都可以正常运作了。

模态对话框

为了创建一个模态对话框，你必须创建 CDialog 派生类的一个本地对象，然后调用它的 DoModal()成员函数。DoModal()会为你初始化、创建、显示并销毁该对话框。

当 DoModal()返回时，你可以检查它的返回值，看用户是选择了“OK”还是“Cancel”。在 DoModal()返回之后，你仍然可以访问存放在 CDialog 派生类中的信息。下面时一个典型的 MFC 模态对话框：

```
void CMyView::DisplayOrderDialog()
{
    CMyDialog myDialog(ID_DLG_MYDIALOG);
    int nRetVal = myDialog.DoModal();
    if (nRetVal == IDOK){
        //Do OK processing, maybe retrieve a value from myDialog?
    }
    else {
    //Do Cancel processing, maybe reset some values?
    }
}
```

在这个例子里，ID_DLG_MYDIALOG（被传递给 CMyDialog 的构造函数）引用一个位于应用程序资源中的对话框模板资源。为了方便，CDialog 还提供了另外一个构造函数，这个构造函数带两个参数——一个整数 ID 和一个对话框模板的名字。

非模态对话框

非模态对话框的生存期通常比一次函数调用长，所以它是从堆上利用 new 操作符创建的。为了能够跟踪对话框，大多数应用程序需要在成员函数中存储指向非模态对话框的指针，这样它可以在适当时候销毁对话框（例如在应用程序退出时或者当用户通过菜单选项关闭对话框时）。

在构造了一个非模态对话框之后，你必须显式地调用 Create()成员函数来显示对话框。非模态对话框要通过继承的 CWnd::DestroyWindow()成员函数来完成。

下面的代码显示了如何创建、显示并销毁一个非模态的对话框。

```
//Construct it, store in a member variable.
m_pDlgMyDlgPtr = new CMyDialog;
//Display it, pass in the template reference
m_pDlgMyDlgPtr->Create(ID_DLG_MYDIALOG);
//Sometime later, nuke it.
m_pDlgMyDlgPtr->DestroyWindow();
m_pDlgMyDlgPtr = NULL;
```

对话框操作

除了模态创建和非模态创建之外，CDialog 类还支持一些对话框操作。例如，调用 NextDlgCtrl()、PrevDlgCtrl()或 GotoDlgCtrl()，你就可以遍历对话框中的所有控件。它还支持设置对话框中按钮的辅助 ID 和默认 ID 的操作。

CDialog 还允许覆盖 OK 按钮和 Cancel 按钮方法，也允许覆盖对话框初始化方法。

CDialog 的某些派生类还可以覆盖特定的功能。例如，CFontDlg 具有覆盖的方法来指定字体。

一旦你知道了如何创建对话框模板以及如何利用类向导创建 CDialog 派生类，你就会发现 CDialog 的使用很方便。下面一节是关于 MFC 如何实现 CDialog 的，我们会看到 MFC 为了给我们提供方法遇到多大的麻烦。

CDialog 内幕

有一些有用的 Win32 API 可以用来创建对话框：

- CreateDialog()——从模板资源创建一个非模态的对话框。
- CreateDialogIndirect()——从模板指针创建一个非模态的对话框。
- DialogBox()——从模板资源创建一个模态的对话框。
- DialogBoxIndirect()——从模板指针创建一个模态的对话框。

CDialog 只调用 CreateDialogIndirect() API，并自己实现了模态化，而没有调用 DialogBox 的例程。因为 MFC 调用 API 的“Indirect”版本，所以它会在对话框模板的装载过程中对资源增加额外的检查。

在 MFC 的早期版本中，CDialog 只是调用 CreateDialogIndirect()。在后来的版本中，又增加了一些新的特性，CDialog 变得更复杂。例如，增加了 OLE 控件就使得 CDialog 的代码量加倍，因为要增加对 OLE 控件的支持就需要很多 OLE 接口。

我们会在第 15 章提到 OLE 控件异常，而且会在那一章中集中讨论 CDialog 如何包装并增强 Win32 对话框 API。

CDialog 类的声明位于 AFXWIN.H 头文件中，它的实现在文件 DLGCORE.CPP 中。CDialog 还有一些成员在 MFC 的内嵌文件 AFXWIN2.INL 中。

缩减后的 CDialog 的声明

为了弄清 CDialog 如何运作，我们首先看一个缩减后的 CDialog 的声明，如程序清单 6-1 所示。简而言之，有文档说明的 CDialog 成员都被忽略掉了，以便我们可以集中研究那些没有文档说明的类的实现细节。

程序清单 6-1 缩减后的 CDialog 的声明（在文件 AFXWIN.H 中）

```
class CDialog : public CWnd
{
    DECLARE_DYNAMIC(CDialog)
// Construction **omitted
// Attributes  **omitted
// Operations  **omitted
// Overridables **omitted
// Implementation **The good stuff!!
public:
    virtual ~CDialog();
    virtual BOOL PreTranslateMessage(MSG* pMsg);
    virtual BOOL OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo);
    virtual BOOL CheckAutoCenter();
protected:
    UINT m_nIDHelp;          // Help ID (0 for none, see HID_BASE_RESOURCE)
    // parameters for 'DoModal'
    LPCTSTR m_lpszTemplateName;    // name or MAKEINTRESOURCE
    HGLOBAL m_hDialogTemplate;     // indirect (m_lpDialogTemplate == NULL)
    LPCDLGTEMPLATE m_lpDialogTemplate;  // indirect if (m_lpszTemplateName
                                        // == NULL)
    void* m_lpDialogInit;      // DLGINIT resource data
    CWnd* m_pParentWnd;        // parent/owner window
    HWND m_hWndTop;            // top level parent window (may be disabled)

    _AFX_OCC_DIALOG_INFO* m_pOccDialogInfo;
    virtual BOOL SetOccDialogInfo(_AFX_OCC_DIALOG_INFO* pOccDialogInfo);
    virtual void PreInitDialog();
// implementation helpers
    HWND PreModal();
    void PostModal();

protected:   // ** some omitted
    afx_msg LRESULT OnCommandHelp(WPARAM wParam, LPARAM lParam);
    afx_msg LRESULT HandleInitDialog(WPARAM, LPARAM);
    afx_msg LRESULT HandleSetFont(WPARAM, LPARAM);
    DECLARE_MESSAGE_MAP()
};
```

这是没有文档说明的成员函数和成员变量的一个纲要。我们会在后面讨论 CDialog 的各种特性时详细地探索其中一部分。

CDialog 中没有文档说明的数据成员

CDialog 的数据成员用来存储传递给构造函数的信息和 CDialog 的初始化信息。这些成员都是受保护的，这样你就只能在派生类中访问它们。

- m_nIDHelp——按钮的辅助 ID，由模板 ID 确定。
- m_lpszTemplateName——资源模板的名字。如果指定了 ID，m_lpszTemplateName 就是 MAKELONG(0,ID)。
- m_hDialogTemplate——一旦装载之后，是资源模板句柄。

- m_lpDialogInit——一个指向用来初始化对话框的数据的指针。用来初始化对话框的数据是从资源文件中装载的，通过一个资源 ID 与对话框相联系。因为它控制了对话框的初始化，所以我们会很详细地讨论这个成员变量。
- m_pParentWnd——一个指向父窗口或主窗口的 CWnd 指针。
- m_hWndTop——最顶层的父窗口。
- m_pOccDialogInfo——存储 OLE 控件所需要的信息。

CDialog 的一些有趣的成员函数

- virtual PreTranslateMessage()——为特殊情况过滤消息，例如工具提示和 SHIFT-F1 环境帮助。
- virtual OnCmdMsg()——将命令消息路由给所有者和当前的 CWinThread 对象。忽略控件通知。
- virtual CheckAutoCenter()——检查用户是否已经指定了对话框自动位于中心：DS_CENTER、DS_CENTERMOUSE 或 DS_ABSALIGN。（注释：对话框还有 Windows 95 扩展风格）而且 x 和 y 必须是 0。
- virtual SetOccDialogInfo——用于设置 m_pOccDialogInfo 成员变量。
- virtual PreInitDialog()——CDialog 派生类的一个占位符。PreInitDialog()会在 WM_INITDIALOG(OnInitDialog())之前被调用。而且如果你想要更早执行初始化，那么它也很有用。
- PreModal()——通过准备将自己粘贴到已创建的对话框窗口上，为 DoModal()逻辑准备 CDialog。
- PostModal()——在 DoModal()逻辑之后将 CDialog 分离。

在我们研究 CDialog 的关键成员函数的过程中，你还会时常看到这些没有文档说明的成员函数。

模态对话框的创建

首先，我们来跟踪一个模态对话框的生存周期，以便对实际对话框何时创建以及设计了哪些实现细节有所了解。

正如前面提到的，创建一个本地的 CDialog 派生对象，然后调用它的 DoModal()方法。这里涉及到两个方法——构造函数和 DoModal()。

CDialog 的构造函数

CDialog 有两个模态构造函数：一个以字符串模板名为第一个参数，另一个以资源 ID 为第一个参数。资源 ID 使用得更频繁，所以我们来看这个构造函数的实现。程序清单 6-2 中是模态构造函数的伪代码，在文件 DLGCORE.CPP 中。

程序清单 6-2 CDialog 的构造函数（在文件 DLGCORE.CPP 中）

```
CDialog::CDialog(UINT nIDTemplate, CWnd* pParentWnd)
{
    AFX_ZERO_INIT_OBJECT(CWnd);
    m_pParentWnd = pParentWnd; m_lpszTemplateName =
        MAKEINTRESOURCE(nIDTemplate);
    m_nIDHelp = nIDTemplate;
}
```

从程序清单 6-2 中可以看出，模态 CDialog 的构造函数会先将实例清空（使用 AFX_ZERO_INIT_OBJECT 宏），然后将参数填入相应的 CDialog 数据成员。

AFX_ZERO_INIT_OBJECT 宏

这个宏位于文件 AFX.H 中，它可以用来对你的类（有数据成员）清零。它的声明如下：

```
#define AFX_ZERO_INIT_OBJECT(base_class) \
    memset(((base_class*)this)+1, 0, sizeof(*this) - \
    sizeof(class base_class));
```

“+1”是为了确保 vtable 没有被覆盖。

需要注意的是，如果你的类有任何带有构造语义的嵌入成员，比如嵌入的 CString 变量和其他具有特定构造函数的对象，或者具有虚函数的对象（它们的 vtbl 指针就会被覆盖），则使用宏不是一个好主意。

DoModal()

一旦 CDialog 的派生类构造好了，你就可以调用 DoModal()。DoModal()会处理对话框的创建、显示和销毁。你可以认为这个函数会很大，因为它需要完成的任务很多，事实也是这样。程序清单 6-3 列出了 CDialog::DoModal()的伪代码，在文件 DLGCORE.CPP 中。

程序清单 6-3 CDialog::DoModal()的伪代码（在文件 DLGCORE.CPP 中）

```
int CDialog::DoModal()
{
    // Code Block 1
    LPCDLGTEMPLATE lpDialogTemplate = m_lpDialogTemplate;
    HGLOBAL hDialogTemplate = m_hDialogTemplate;
    if (m_lpszTemplateName != NULL){
        HINSTANCE hInst = AfxFindResourceHandle(m_lpszTemplateName,
            RT_DIALOG);
        HRSRC hResource = ::FindResource(hInst, m_lpszTemplateName,
            RT_DIALOG);
        hDialogTemplate = LoadResource(hInst, hResource);
    }
    if (hDialogTemplate != NULL)
        lpDialogTemplate = (LPCDLGTEMPLATE)LockResource(hDialogTemplate);
    // return -1 in case of failure to load the dialog template resource
    if (lpDialogTemplate == NULL)
        return -1;
    // Code Block 2
    HWND hWndParent = PreModal();
    AfxUnhookWindowCreate();
    CWnd* pParentWnd = CWnd::FromHandle(hWndParent);
    BOOL bEnableParent = FALSE;
    if (hWndParent != NULL && ::IsWindowEnabled(hWndParent))
        ::EnableWindow(hWndParent, FALSE);
    bEnableParent = TRUE;
    // Code Block 3
    AfxHookWindowCreate(this);
    if (CreateDlgIndirect(lpDialogTemplate, CWnd::FromHandle(hWndParent)))
    {
        if (m_nFlags & WF_CONTINUEMODAL){
            // enter modal loop
            DWORD dwFlags = MLF_SHOWONIDLE;
            if (GetStyle() & DS_NOIDLEMSG) dwFlags |= MLF_NOIDLEMSG;
```

```
                VERIFY(RunModalLoop(dwFlags) == m_nModalResult);
            }
            SetWindowPos(NULL, 0, 0, 0, 0,
               SWP_HIDEWINDOW|SWP_NOSIZE|SWP_NOMOVE|SWP_NOACTIVATE|SWP_NOZORDER);
        }

        if (bEnableParent)
            ::EnableWindow(hWndParent, TRUE);
        if (hWndParent != NULL && ::GetActiveWindow() == m_hWnd)
            ::SetActiveWindow(hWndParent);
        // Code Block 4
        DestroyWindow();
        PostModal();
        if (m_lpszTemplateName != NULL || m_hDialogTemplate != NULL)
            UnlockResource(hDialogTemplate);
        if (m_lpszTemplateName != NULL)
            FreeResource(hDialogTemplate);
        return m_nModalResult;
    }
```

因为 CDialog::DoModal()是一个很长很复杂的函数，所以我们在其中增加了很多注释（例如//Code Block 1），从而将它划分成容易理解的几个部分。

- Code Block 1：载入对话框资源。

  DoModal()的第一部分会使用对话框模板名来从应用程序的资源文件中查找、载入并锁定对话框模板。如果 DoModal()不能定位资源，它会返回错误代码-1。

- Code Block 2：准备创建对话框。

  DoModal()的第二部分首先调用了 PreModal()。PreModal()会执行一些安全检查（例如，确保没有对一个非模态对话框调用 DoModal()），然后为对话框查找所需的父句柄。它会调用 CWnd::GetSafeOwner()来完成这些，然后将结果存储在成员变量 m_hWnd 中。

  在调用 PreModal()之后，它又调用了 EnableWindow(FALSE）来将对话框的父窗口禁用，从而父窗口不能再接收消息。这样就“冻结”了父窗口（通常是主窗口）并强制实现了模态化。

- Code Block 3：创建并显示对话框。

  在 DoModal()的第三部分中，它首先准备将一个 MFC 对象粘贴到对话框中，这通过调用 AfxHookWindowCreate()完成。然后又调用了 CWnd::CreateDlgIndirect()。这个没有文档说明的 CWnd 成员函数在文件 DLGCORE.CPP 中。CWnd::CreateDlgIndirect()会做一些错误检查，然后设置对话框的字体。在设置默认字体之后，CreateDlgIndirect()会用 WF_CONTINUEMODAL 对 CWnd::m_nFlags 成员做 OR 操作，然后调用 Win32 API CreateDialogIndirect()，参数是对话框模板、父窗口句柄和 AfxDlgProc()（作为对话框过程）。如果创建过程顺利，CWnd::CreateDlgIndirect()返回 TRUE，否则返回 FALSE。

  如果对 CWnd::CreateDlgIndirect()的调用成功，DoModal()就会调用 CWnd::RunModalLoop()。CWnd::RunModalLoop()会处理消息直至用户按下了“OK”按钮或“Cancel”按钮。OnOK()和 OnCancel()成员函数都会调用 EndDialog()，EndDialog()

又调用 CWnd::EndModalLoop()，这样就结束了 CWnd::RunModalLoop()的处理过程，将控制流转给 DoModal()。

一旦 RunModalLoop()返回了，DoModal()会调用 SetWindowPos()隐藏目前已经死掉的对话框。对话框被隐藏之后，DoModal()会调用 EnableWindow(TRUE)来激活父窗口，再调用 SetActivateWindow()确保父窗口成为新的激活窗口。

理解 CDialog::DoModal()代码块的第三部分对全面掌握 CDialog 来说十分重要。你需要在继续向下读之前确保已经掌握了这一段。启动 VC++，并在示例中的 about 对话框中的 CDialog::DoModal()那里设置一个断点，跟踪一下，这会帮助你理解。

- Code Block 4：收尾。

  到了 Code Block 4，模态对话框已经显示过，用户也已经点击过"OK"按钮或"Cancel"按钮，对话框也已经被隐藏。现在 DoModal()会调用 DestroyWindow()来销毁 Windows 对话框对象，然后调用 PostModal()。PostModal()会将第一步中粘贴到 Windows 对象的 C++对象分离，然后将 m_hWndTop 设置为 NULL。

  最后，DoModal()会对一些资源解锁，释放那些需要释放的资源，然后将 CWnd::RunModalLoop()的返回值返回。

这些就是 DoModal()所完成的。总结起来，DoModal()函数的关键就在于它调用了 Win32 API CreateDialogIndirect()，然后使用 CWnd::RunModalLoop()处理所有的消息，并强制模态化。因为 CWnd 中有模态循环逻辑，所以如果你需要，可以写自己的 CWnd 派生类，并使得它模态化，而不用从 CDialog 派生。我们在后面要提到的 MFC 的属性页就是这种情况。

非模态对话框的创建

我们已经知道了模态对话框是如何创建的，下面让我们看看非模态对话框是如何创建的。你要记住的是创建一个非模态对话框需要使用 *new* 操作符，然后调用 CDialog::Create()来初始化并显示对话框。你可以调用 CDialog::EndDialog()来结束对话框。

CDialog::Create()

CDialog 的默认构造函数会将类初始化为 0，所以我们可以直接跳到 CDialog::Create()成员函数。它位于文件 DLGCORE.CPP 中，如程序清单 6-4 所示。

程序清单 6-4 *CDialog::Create()*的实现（在文件 *DLGCORE.CPP* 中）

```
BOOL CDialog::Create(LPCTSTR lpszTemplateName, CWnd* pParentWnd)
{
    m_lpszTemplateName = lpszTemplateName;
    // used for help
    if (HIWORD(m_lpszTemplateName) == 0 && m_nIDHelp == 0)
       m_nIDHelp = LOWORD((DWORD)m_lpszTemplateName);
#ifdef _DEBUG
    if (!_AfxCheckDialogTemplate(lpszTemplateName, FALSE)){
        ASSERT(FALSE);          // invalid dialog template name
        PostNcDestroy();        // cleanup if Create fails too soon
        return FALSE;
    }
#endif //_DEBUG
    HINSTANCE hInst = AfxFindResourceHandle(lpszTemplateName, RT_DIALOG);
    HRSRC hResource = ::FindResource(hInst, lpszTemplateName, RT_DIALOG);
```

```
    HGLOBAL hTemplate = LoadResource(hInst, hResource);
    BOOL bResult = CreateIndirect(hTemplate, pParentWnd);
    FreeResource(hTemplate);
    return bResult;
}
```

首先，Create()在相应的成员变量中存储了模板的名字和辅助 ID。然后它会检查对于当前对话框的情况，模板是否合法。如果所有检查都通过，它会查找资源并装载资源。

如果资源的查找和装载顺利，Create()就会调用 CDialog::CreateIndirect()来创建对话框。

CreateIndirect()会将父窗口设置为 AfxGetMainWnd()的返回值，然后调用我们总是提到的 CWnd::CreateDlgIndirect()。这个函数最终又调用了 Win32 API CreateDialogIndirect()。最终，Create()会调用 FreeResource()释放已经装载的资源，并返回 CreateIndirect()调用的结果。

到此为止，一个非模态的对话框已经显示出来了，终端用户就可以开始交互。非模态对话框会一直显示在屏幕上，直至用户点击了“OK”按钮或“Cancel”按钮。无论点击哪个按钮都会最终调用 EndDialog()。当然用户也可以自己调用 EndDialog()直接关闭非模态对话框。

Create()中一个有趣的技术

在 CDialog::Create()中，注意一下在 DEBUG 版本中有一个特殊检查。Create()会调用诊断辅助函数 _AfxCheckDialogTemplate()。这个函数会执行一系列检查，确认用户提供了一个合法的对话框模板。你可以将这一技术应用到自己的类中。

如果有些数据需要进行额外检查，你就可以将检查转移到诊断辅助函数中。如果在后面你发现需要进行新的检查，只要将它增加到辅助函数中就行了。

CDialog::EndDialog(): 终结函数

我们在关于模态对话框的讨论中简要地提到了 EndDialog()，我们在这里会仔细地看这个函数。程序清单 6-5 中是 CDialog::EndDialog()的代码，在文件 DLGCORE.CPP 中。

程序清单 6-5 CDialog::EndDialog: CDialog 的终结函数

```
void CDialog::EndDialog(int nResult)
{
    if (m_nFlags & (WF_MODALLOOP|WF_CONTINUEMODAL))
    EndModalLoop(nResult);
    ::EndDialog(m_hWnd, nResult);
}
```

在非模态的情况下，不会设置 WF_MODALLOOP 标志，所以也就不会调用 EndModalLoop()。Win32 API EndDialog()会确保对话框已经销毁。

CDialog 控件的初始化

MFC CDialog 控件初始化很透明，以至于你从来不用考虑它是如何实现的。如果你用 VC++的资源编辑器创建一个对话框，并且在对话框里是一个复选框，你可以在上面添加字符串“one”、“two”和“three”。

MFC 如何获得这些信息呢？快速检查一下生成的代码，我们没有找到类似下面的调用：

```
m_pCombo->AddString("one");
m_pCombo->AddString("two");
m_pCombo->AddString("three");
```

可能我们跟踪了 CDialog 的初始化就可以发现 MFC 如何实现它的 CDialog 初始化。

WM_INITDIALOG

一旦创建了对话框，Windows 就在窗口显示之前发送一个 WM_INITDIALOG 消息，这样应用程序就可以在对话框在屏幕上显示之前初始化对话框中的控件。

CDialog 消息映射表将 WM_INITDIALOG 映射到 CDialog::HandleInitDialog()。HandleInitDialog()会执行 OLE 控件的一些初始化，从而调用 CDialog::OnInitDialog()。CDialog::OnInitDialog()也就是 WM_INITDIALOG 句柄的另一个名字。

CDialog::OnInitDialog()会调用 CWnd::ExecuteDlgInit()，由后者完成最终对话框的初始化。

为什么是 CWnd::ExecuteDlgInit()?

到此为止，你可能会问“为什么这么多针对 CDialog 的方法都在 CWnd 中呢？”这个问题问的好。原因就是还有其他 CWnd 的派生类（不是从 CDialog 派生的）需要使用这些方法，所以为了避免代码复制，MFC 组就将这些代码都放在了公共父类 CWnd 中。这一点看来有些缺乏面相对象的思想，但它解决了复制大量处理模态化和初始化的代码的问题。在我们后面讨论到 MFC 属性页时这些就会显得更有意义。

CWnd::ExecuteDlgInit()

实际上有两个 CWnd::ExecuteDlgInit()函数，第一个的参数是对话框模板的名字。

你可能会问“为什么要这样做呢？”。看起来 ExecuteDlgInit()要对对话框模板做一些处理，但实际上不是这样的。如果你看 VC++生成的.RC 文件，你就会发现一个资源会像程序清单 6-6 中列出的那样：

程序清单 6-6 DLGINIT 资源的数据

```
IDD_ABOUTBOX DLGINIT
BEGIN
IDC_COMBO1, 0x403, 4, 0
0x6e6f, 0x0065,
IDC_COMBO1, 0x403, 4, 0
0x7774, 0x006f,
IDC_COMBO1, 0x403, 6, 0
0x6874, 0x6572, 0x0065,
IDC_COMBO1, 0x403, 5, 0
0x6f66, 0x7275, "\000"
IDC_COMBO1, 0x403, 5, 0
0x6966, 0x6576, "\000"
IDC_COMBO1, 0x403, 4, 0
0x6973, 0x0078,
IDC_COMBO1, 0x403, 6, 0
0x6573, 0x6576, 0x006e,
0
END
```

你可以用这个用户定义的资源将一些数据存放到资源文件中，资源的名字由你指定。在上面的例子中，资源的名字是 DLGINIT，资源的 ID 和 DLGINIT 数据所属的对话框的 ID 一样，是 IDD_ABOUTBOX。如果 MFC 需要用这个数据，第一个 ExecuteDlgInit()会为指定名字的对话框查找到 DLGINIT 资源。如果指定名字的 DLGINIT 资源存在，ExecuteDlgInit()会

装载资源并锁定资源，从而获得指向数据的指针。有了指针之后，它会调用第二个ExecuteDlgInit()函数，这个函数负责处理指针指向的数据。

ExecuteDlgInit(): 处理 DLGINIT 数据

现在我们知道了 MFC 将初始化信息存放在哪里，剩下最大的问题就是它的格式是什么样的。下面让我们看看 ExecuteDlgInit()的代码，看它如何处理 DLGINIT 数据。程序清单 6-7 显示了 ExecuteDlgInit()的缩减版本（在文件 WINCORE.CPP 中）。你会发现在这里任何初始化 OLE 控件的代码都被忽略了。

程序清单 6-7 CWnd::ExecuteDlgInit()（在文件 WINCORE.CPP 中）

```
BOOL CWnd::ExecuteDlgInit(LPVOID lpResource)
{
    BOOL bSuccess = TRUE;
    UNALIGNED WORD* lpnRes = (WORD*)lpResource;
    while (bSuccess && *lpnRes != 0){
        WORD nIDC = *lpnRes++;
        WORD nMsg = *lpnRes++;
        DWORD dwLen = *((UNALIGNED DWORD*&)lpnRes)++;
        if (nMsg == LB_ADDSTRING || nMsg == CB_ADDSTRING){
            if (::SendDlgItemMessageA(m_hWnd, nIDC, nMsg, 0,
                (LONG)lpnRes) == -1)
                bSuccess = FALSE;
        }
        // skip past data lpnRes = (WORD*)((LPBYTE)lpnRes + (UINT)dwLen);
    }
    return bSuccess;
}
```

ExecuteDlgInit()使用一个 WORD 指针 lpnRes 来遍历数据。首先它将一个 ID 压缩成一个 nIDC，然后将消息压缩成 nMsg，最后将一个 DWORD 压缩成 dwLen。

如果压缩后的 nMsg 是一个 LB_ADDSTRING 或 CB_ADDSTRING，则 ExecuteDlgInit()会用这些压缩后的数据做参数调用 SendDlgItemMessage()。

组合在一起

我们现在已经掌握了所有关于 MFC 对话框控件初始化的问题，现在该将这些内容组合起来。Visual C++将初始化信息作为数据存储在资源文件中，名字是 DLGINIT，并且资源 ID 与它所属于的对话框 ID 一样。

在响应 WM_INITDIALOG 消息创建对话框时，MFC 会装载这个资源（资源中包含了控件的信息）。ExecuteDlgInit()会遍历数据，解码这些消息，然后将消息发送给已经创建好但是还没有显示的对话框控件。

这样做的一个好处就是字符串在可执行文件的数据段中不占用任何内存，它们可以更好地定位，因为这些数据在资源文件中。

请记住 OLE 控件也使用相同技术进行初始化，我们会在第 15 章详细讨论 OLE 控件的内幕。

MFC 还支持另外一种初始化对话框控件的方法，你可以利用这种方法更方便地在数据成员和控件之间交换信息，并验证用户端的对话框控件输入。这个特性就是动态数据交换和对

话框数据确认，即 DDX/DDV。

DDX/DDV：CDialog 数据交换与验证

为了看 DDX/DDV 是如何实现的，我们首先看如何在应用程序中使用 DDX/DDV，然后我们再看 MFC 的一些小片断代码。这些小片断代码就解释了这个方便的特性在内部是如何运作的。

使用 DDX/DDV

正如名字所示，DDX/DDV 是两个特性。首先就是动态数据交换，DDX 将一个数据成员和一个控件联系起来，这样在对话框显示的时候。你就可以在数据成员中指定它的初始化值，并且由 MFC 将初始值拷贝给控件。当对话框被关闭时，控件的状态又会被拷贝回给相应的成员变量。例如，如果你有一个双状态的复选框，就可以将这个复选框和一个 Boolean 类型的成员变量联系起来，这样你就不用总是调用 GetCheck()来保存值。

为了在你的对话框中使用 DDX，你需要声明一个 DoDataExchange()成员函数。这个成员函数的参数是一个 CDataExchange 指针。然后你要对特定于控件的函数（这些函数将控件映射到成员变量）进行调用。程序清单 6-8 列出了一个例子，这个对话框中有两个编辑控件、一个复选框和一个单选按钮。

程序清单 6-8 带有 DoDataExchange()成员函数的 CDialog 派生类的一个例子

```
void CMyDlg::DoDataExchange(CDataExchange * pDX)
{
    CDialog::DoDataExchange(pDX); //Call base first.
    DDX_Text(pDX, IDC_EDIT1, m_strEdit1);
    DDX_Text(pDX, IDC_EDIT2, m_uEdit2);
    DDX_Check(pDX, IDC_CHECK, m_bCheckState);
    DDX_Radio(pDX, IDC_RADIO, m_nRadioState);
}
```

在上面的例子中，资源 ID 为 IDC_EDIT1 的编辑器被映射到一个 CString 成员变量 m_strEdit1 中。ID 为 IDC_EDIT2 的编辑器被映射到一个无符号整数成员变量 m_uEdit2 中。类似地，资源 ID 为 IDC_CHECK 的复选框控件被映射到一个 Boolean 类型的成员变量 m_bCheckState 中，资源 ID 为 IDC_RADIO 的单选按钮被映射到成员变量 m_nRadioState 中。

对于每种 Windows 控件至少要有一个 DDX 函数，另外对于每种控件还可以有若干重载函数，这样你就可以对控件使用不同类型的成员变量。程序清单 6-8 中第二个 DDX_Text 调用是重载的 DDX 函数的例子，它自动为你将编辑器中的字符串转换成一个整数。

看起来写 DoDataExchange()函数很枯燥无味，确实是这样。幸运的是，Visual C++的类向导已经能够增加 DoDataExchange()成员函数，而且能够调用将控件映射到成员变量的函数。你只要在类向导中选择“Member variable”标签，你就会看到如何在 DoDataExchange()成员函数中增加一个入口。图 6-2 显示了类向导的 Member Variable 标签。

一旦你实现了自己的 DoDataExchange()成员函数，MFC 就会帮你完成其他的工作。如果你想初始化一些控件，只要在你的 CDialog 派生类的构造函数中初始化相应的成员数据即可，这样在显示时，控件就会有相应的数据。如果你的对话框是通过点击“OK”按钮关闭的，那

么这些成员变量中就会包含对话框中控件的值。

MFC 对话框的第二个特性就是对话框数据验证。这个特性和 DDX 共同合作来验证已经修改的数据。例如，如果你想要用户在编辑框中输入一个介于 1~234 的数，DDV 会确保用户所输入的是一个整数而且位于 1~234 之间。

为了使用 DDV 支持，你要使用 DDV 的一个函数，用这个函数来指定你对控件/数据成员对的限制。

下面我们扩展前面的例子，强制用户在第一个编辑控件中输入指定数目的字符，在第二个编辑控件中输入一个 1~234 的数字。程序清单 6-9 中列出了我们该如何做。

程序清单 6-9　在 DDX 示例中增加 DDV 控件验证

```
void CMyDlg::DoDataExchange(CDataExchange * pDX)
{
    CDialog::DoDataExchange(pDX); //Call base first.
    DDX_Text(pDX, IDC_EDIT1,m_strEdit1);
    DDV_MaxChars(pDX, m_strEdit1, 20);
    DDX_Text(pDX, IDC_EDIT2, m_nEdit2Integer);
    DDV_MinMaxUInt(pDX, m_uEdit2,1,234);
    DDX_Check(pDX, IDC_CHECK, m_bCheckState);
    DDX_Radio(pDX, IDC_RADIO, m_nRadioState);
}
```

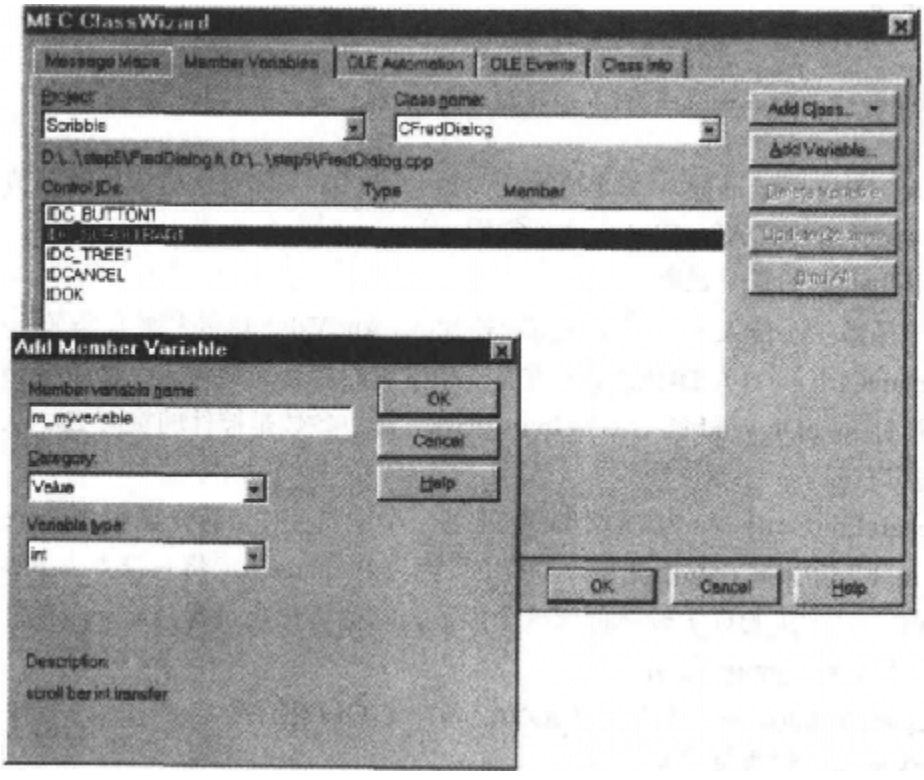


图 6-2　类向导的 Member Variable 标签

这些就是如果你要限制用户所需要做的。下面我们看 MFC 如何实现 DDX/DDV 的所有功能。

DDX/DDV 如何工作

浏览了如何使用 DDX/DDV 就带来了一个很大的问题——它内部如何运作：

- CDataExchange 扮演了什么样的角色？
- MFC 何时、何地、以何种方式从成员变量那里获得控件的信息，又何时、何地、以何种方式将这些信息传递给成员变量？
- DDX/DDV 函数都做了些什么？

带着这些问题，我们开始分解这个新的辅助类 CDataExchange。

DDX/DDV 的辅助类 CDataExchange

DDX/DDV 辅助类 CDataExchange 位于文件 AFXWIN.H 中，如程序清单 6-10 所示。

程序清单 6-10 DDX/DDV 辅助类 CDataExchange 的声明

```
class CDataExchange
{
// Attributes
public:
    BOOL m_bSaveAndValidate;    // TRUE => save and validate data
    CWnd* m_pDlgWnd;            // container usually a dialog
// Operations (for implementors of DDX and DDV procs)
    HWND PrepareCtrl(int nIDC);      // return HWND of control
    HWND PrepareEditCtrl(int nIDC); // return HWND of control
    void Fail();                     // will throw exception
    CWnd* PrepareOleCtrl(int nIDC); // for OLE controls in dialog
// Implementation
    CDataExchange(CWnd* pDlgWnd, BOOL bSaveAndValidate);
    HWND m_hWndLastControl;     // last control used (for validation)
    BOOL m_bEditLastControl;    // last control was an edit item
};
```

下面是对 CDataExchange 中每个成员的简短介绍。如果你对这些成员的细节感兴趣，可以在 MFC 源文件 DLGDATA.CPP 中找到它们。

CDataExchange 的成员函数

- 构造函数——传入一个对话框指针和 bSaveAndValidate 的初始化设置。
- PrepareCtrl()——为 DDX/DDV 准备一个非编辑控件。通常这个函数会调用内部对话框指针 m_pDlgWnd 的 GetDlgItem()方法，其中参数是控件的资源 ID，从而获得控件的窗口句柄。
- PrepareEditCtrl()——为 DDX/DDV 准备一个编辑控件。首先调用 PrepareCtrl()，然后将 m_bEditLastControl 设置为 TRUE，因为 PrepareCtrl()将它设置为 FALSE。
- Fail()——如果验证失败就调用这个函数。它会将焦点设置到前一个控件，并抛出一个 CUserException 异常。
- PrepareOleCtrl()——为 DDX/DDV 准备一个 OLE 控件。

CDataExchange 的成员变量

- m_bSaveAndValidate——这个成员变量指定了 DDX 的方向，以及是否应该做 DDV。如果它是 FALSE，数据就从成员变量流到控件，如果它是 TRUE，数据就从控件流向成员变量。DDV 只发生在它为 TRUE 的时候。
- m_pDlgWnd——一个指向对话框的 CWnd 指针，它包含了需要交换和验证的控件。这个值通过 CDataExchange 构造函数传入。

- m_hWndLastControl——m_hWndPreviousControl 可能是这个 CDataExchange 成员变量的更好的名字，它记录了前一个控件的句柄。
- m_bEditLastControl——一个 Boolean 变量，它指定了前一个控件是否是编辑器。

随着我们对 DDX/DDV 内幕的分析，CDataExchange 的这些成员所扮演的角色就会越来越清晰。

DDX/DDV 例程的内幕

我们已经对 CDataExchange 的成员有了一定的了解，下面就让我们看一些 DDX/DDV 的函数，看它们都做了些什么。

我们首先看程序清单 6-9 中所使用到的函数。首先，我们用一个 CString 来调用 DDX_Text，以便在一个 CString 和一个编辑控件之间交换数据。程序清单 6-11 中列出了这个函数的伪代码，它位于文件 DLGDATA.CPP 中。文件 DLGDATA.CPP 中包含了大多数 DDX/DDV 函数。

程序清单 6-11 DDX_Text()函数的伪代码（在文件 DLGDATA.CPP 中）

```
void DDX_Text(CDataExchange* pDX, int nIDC, CString& value)
{
    HWND hWndCtrl = pDX->PrepareEditCtrl(nIDC);
    if (pDX->m_bSaveAndValidate){
        int nLen = ::GetWindowTextLength(hWndCtrl);
        ::GetWindowText(hWndCtrl, value.GetBufferSetLength(nLen), nLen+1);
        value.ReleaseBuffer();
    }
    else
        AfxSetWindowText(hWndCtrl, value);
}
```

每个 DDX 函数都会先调用 PrepareEditCtrl()或 PrepareCtrl()，至于调用哪一个依赖于控件的类型。然后 DDX_Text()检查 Boolean 变量 m_bSaveAndValidate 的值。如果这个标志的值是 TRUE（表示数据从控件流向成员变量），然后 DDX_Text()调用 Win32 API，从而从编辑控件获得字符串，并将字符串存放在“value”参数中，在我们的例子就是 m_strEdit1 成员变量。

如果 CDataExchange::m_bSaveAndValidate 是 FALSE（表示数据从成员变量流向控件），DDX_Text()会调用 AFX 的函数，从而使用 CString 参数的内容更新编辑控件。图 6-3 显示了 DDX/DDV 数据的流向。

AfxSetWindowText()

贯穿 MFC，你会发现它们使用了 AfxSetWindowText()函数，而没有直接调用 SetWindowText()。这样做是为什么呢？你可以看 WINUTIL.CPP 中 AfxSetWindowText()的实现，你会发现它在设置文本之前首先调用 GetWindowText()检查文本是否发生了变化，这样做可以减少很耗时的更新操作，使得控件不用总更新。你可以考虑在类似情况下使用这个函数。

然后我们看在文件 DLGDATA.CPP 中的 DDV_MaxChars()验证函数，它的代码在程序清单 6-12 中，这个函数用来确认所输入的字符数不会超过指定大小。

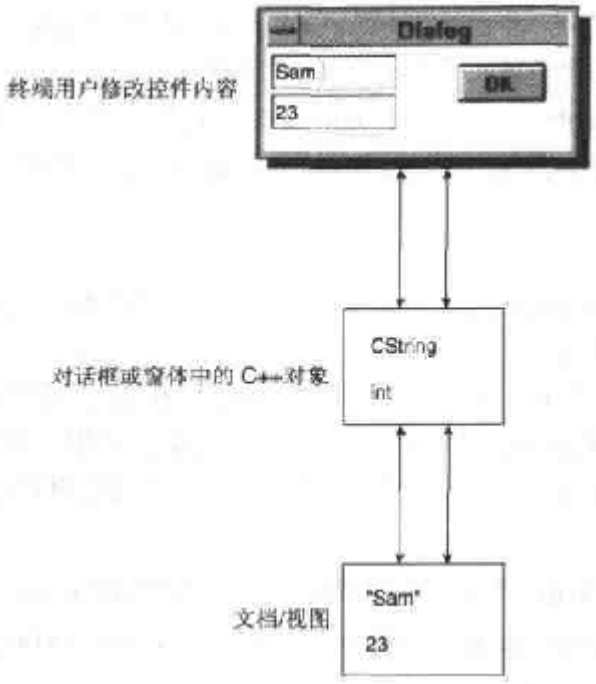


图 6-3 控件的 DDX/DDV 流

程序清单 6-12 DDV_MaxChars()的伪代码（在文件 DLGDATA.CPP 中）

```
void DDV_MaxChars(CDataExchange* pDX, CString const& value, int nChars)
{
    if (pDX->m_bSaveAndValidate && value.GetLength() > nChars){
        TCHAR szT[32]; wsprintf(szT, "%d", nChars);
        CString prompt;
        AfxFormatString1(prompt, AFX_IDP_PARSE_STRING_SIZE, szT);
        AfxMessageBox(prompt, MB_ICONEXCLAMATION, AFX_IDP_PARSE_STRING_SIZE);
        prompt.Empty(); // exception prep
        pDX->Fail();
    }
    else if (pDX->m_hWndLastControl != NULL && pDX->m_bEditLastControl)
        ::SendMessage(pDX->m_hWndLastControl, EM_LIMITTEXT, nChars, 0);
}
```

首先，它检查 CDataExchange::m_bSaveAndValidate 标志，以确保它是 TRUE，这就意味着现在正在进行验证。DDV_MaxChars()然后根据指定的大小检查字符串的长度。如果字符串的长度合法，就不做任何事，如果字符串的长度超过了上界，就执行“if”语句下面的代码块。

这块代码会弹出一个消息框（通过调用 AfxMessageBox()）来提示用户字符串太长了。实际显示的信息由 AfxFormatString1 函数从资源中载入。在这个例子中，这个字符串的 ID 是 AFX_IDP_PARSE_STRING_SIZE，位于文件 AFXRES.RC 中，它的内容是“Please enter no more than %1 characters”。MFC 会将终端用户的消息存放在资源中，这样就可以很快地定位资源。AfxFormatString1()会自动地用指定大小替代%1，在我们的例子中就是 szT 或 20。

一旦消息对话框显示出来，DDV_MaxChars() 会调用 CDataExchange::Fail()，CDataExchange::Fail()将交点转至有问题的控件，并抛出一个 CUserException 异常。

DDV_MaxChars()的第二部分验证了它所拥有的指向编辑器的句柄是非空的，然后使用 EM_LIMITTEXT 编辑消息来通知编辑器要限制输入的大小。这样就将限制字符数的任务转嫁给 Windows 控件，而不是 DDV 函数。DDX/DDV 的许多其他函数也都采用类似方法。

另一个重要的辅助函数是 DDX_TextWithFormat()。这个函数用于将字符串转化成整数和将整数转化成字符串。它的实现在文件 DLGDATA.CPP 中。试试看你能否发现 AfxSimpleScanf()函数是如何工作的，并找出它的限制条件。

未回答的问题

一旦你知道了 CDataExchange 类的辅助成员所扮演的角色，DDX/DDV 函数就不再是深不可测的了。但关于 DDX/DDV 仍然有些问题尚未解答：

- 何时调用 CDialog::DoDataExchange()？谁来调用？
- CDataExchange 在何处创建？
- 什么引发了 DDX/DDV？
- 谁来捕获 CDataExchange::Fail()抛出的异常？

要回答这些问题就要实际跟踪一个关键的成员函数。

发掘答案

我们知道有很多地方都需要调用 DoDataExchange()。这些都是在对话框关闭之前，而且也都是对话框初始化过程使用成员函数的信息对控件初始化。

看一下 CDialog 的成员函数 OnInitDialog()和 OnOK()，你会发现它们分别以 FALSE 和 TRUE 为参数调用了 CWnd::UpdateData()。

程序清单 6-13 列出了默认情况下 CDialog::OnOK()的伪代码，这些代码在文件 DLGCORE.CPP 中。

程序清单 6-13 CDialog::OnOK()的伪代码（在文件 DLGCORE.CPP 中）

```
void CDialog::OnOK()
{
    if (!UpdateData(TRUE))
        TRACE0("UpdateData failed during dialog termination.\n"); return;
    EndDialog(IDOK);
}
```

这似乎就是解释其他关于 DDX/DDV 实现问题的答案。OnOK()调用了 UpdateData (TRUE)，如果返回结果不是 TRUE，它就不关闭对话框。下面我们再看 UpdateData()的代码，看它如何解决 DDX/DDV 的难题。

CWnd::UpdateData()

CWnd::UpdateData()又是一个放入 CWnd 类中的针对对话框的函数，它方便了其他含有控件的 MFC CWnd 的派生类支持 DDX/DDV（这样的函数还有 CWnd::RunModalLoop()和 CWnd::ExecuteDlgInit()）。

下面让我们看 CWnd::UpdateData()的伪代码（在文件 WINCORE.CPP 中），看这个核心函数都调用了什么。程序清单 6-14 列出了它的伪代码。

程序清单 6-14 CWnd::UpdateData()的伪代码（在文件 WINCORE.CPP 中）

```
BOOL CWnd::UpdateData(BOOL bSaveAndValidate)
{
```

```
    // Code Block 1
    CDataExchange dx(this, bSaveAndValidate);
    // Code Block 2
    _AFX_THREAD_STATE* pThreadState = AfxGetThreadState();
    HWND hWndOldLockout = pThreadState->m_hLockoutNotifyWindow;
    pThreadState->m_hLockoutNotifyWindow = m_hWnd;
    // Code Block 3
    BOOL bOK = FALSE;       // assume failure
    TRY {
        DoDataExchange(&dx);
        bOK = TRUE;         // it worked
    }
    CATCH(CUserException, e) {
        // Code Block 4
        ASSERT(bOK == FALSE);
    }
    AND_CATCH_ALL(e) {
        //Code Block 5
        e->ReportError(MB_ICONEXCLAMATION, AFX_IDP_INTERNAL_FAILURE);
        ASSERT(!bOK);
        DELETE_EXCEPTION(e);
    }
    END_CATCH_ALL
    //Code Block 6
    pThreadState->m_hLockoutNotifyWindow = hWndOldLockout;
    return bOK;
}
```

这个函数看起来很长，所以需要简要地解释一下。我们将代码分成几个部分加以说明：

- Code Block 1：首先，它创建了一个 CDataExchange 实例（这也就是 CDataExchange 实例被创建的地方），将 this 作为 CDialog 指针传入，并继续传入 bSaveAndValidate 标志。（记住，OnDialogInit()调用这个函数时标志值是 FALSE，表示数据从成员变量流向控件；OnOK()调用时标志值是 TRUE，表示数据从控件流向成员变量而且需要验证）。
- Code Block 2：下一步，UpdateData()会阻塞当前线程，使线程不能接收控件的消息（第 10 章会深入研究线程）。
- Code Block 3：本地 Boolean 变量 bOK 用来存储 DDX/DDV 代码的返回值。一旦 UpdateData()将 bOK 设置为 FALSE（一个不太好的解决办法），它会调用你的 CDialog 派生类的 DoDataExchange 方法，同时将 CDataExchange 实例作为参数传入。这个调用在一个 TRY 块内，以免在某个 DDV 函数内调用了 CDataExchange::Fail()而抛出异常。如果没有异常，bOK 就被设置为 TRUE，这个值最终返回给 OnUpdate()，表示所有域的交换和验证都没有发生错误。
- Code Block 4：当抛出 CUserException 异常时，就会调用这块代码，也就是当 CDataExchange::Fail()被调用时。此时 bOK 会被设置为 FALSE，异常会通过下面的代码，最终 UpdateData()会返回 0。

  CUserException 异常是一个特殊异常，它表示用户已经接收到了有关错误，也就是说在 catch 语句那里不需要使用消息对话框。

在 Code Block 4 中，你还会发现 ASSERT(bOK == FALSE)。根据函数的逻辑，这一句似乎永远不会出错。MFC 用这个 ASSERT 语句为了确保万无一失，为了确保函数的逻辑流程不出现错误。你可能也需要在自己的 MFC 代码中增加类似的检查，尤其当你有一组程序员对同一块代码操作时。

- Code Block 5：如果 DDX/DDV 处理过程中没有任何异常抛出，就会执行这一部分。它会报告已经发生的一系列错误，也有可能导致返回 FALSE。
- Code Block 6：最后这一块用于重置通知窗口，从而按常规方法处理控件通知，最终返回 FALSE 或 TRUE（依赖于 DoDataExchange()的返回值）。

对于模态对话框，要在 OnInitDialog()和 OnOK()中调用 UpdateData()，但对于非模态对话框，你可能希望在用户点击一个按钮（例如“Apply”）或者响应终端用户交互时发生 DDX/DDV。现在你知道对于非模态对话框只要调用 UpdateData(TRUE)就可以激发 DDX/DDV。

组合在一起

为了将 DDX/DDV 的这些内容组合起来，我们想像如下的场景：

假设有一个模态对话框，上面有一个编辑域，它负责接收介于 1~234 之间的整数。这个对话框的 DoDataExchange()成员函数的实现包含了对 DDX_Text 和 DDV_MinMaxUInt()的调用。DDX 调用将编辑控件与 m_uEditValue 成员变量绑定在一起。

在上面的情况下会引起以下事件：

在对话框的创建过程中，会调用 OnInitDialog()，同时 OnInitDialog()会调用 UpdateData(FALSE)。UpdateData()会调用对话框的 DoDataExchange()，并将 CDataExchange::m_bSaveAndValidate 成员变量赋值为 FALSE。然后会调用 DDX_Text()过程。DDX_Text()会检查 m_bSaveAndValidate 编辑控件，同时会发现应该用 m_uEditValue 的内容初始化编辑控件的值，它会调用 AfxSetWindowText()来达到这一目的。

然后会调用 DDV_MinMaxUInt()方法。这个方法会检查 m_bSaveAndValidate，但因为它的值是 FALSE，所以不作任何处理。

到目前为止，所有的 DDX/DDV 都已经完成，DoDataExchange()完成了，对话框也已经创建好了。

现在假设用户在编辑框中输入了“356”。一旦用户点击了“OK”按钮，就会调用 CDialog::OnOK()，CDialog::OnOK()又会调用 UpdateData(TRUE)。

UpdateData()会调用 DoDataExchange()，DoDataExchange()会调用我们的 DDX_Text()函数，DDX_Text()会检查 m_bSaveAndValidate 是否为 TRUE，然后将字符串“356”转化成一个无符号的整数，接着将值拷贝到成员变量 m_uEditValue 中。然后 DoDataExchange()中的 DDV_MinMaxUInt()会被调用，DDV_MinMaxUInt()发现它应该进行验证，并发现出现了问题。为了解决这个问题，DDV_MinMaxUInt()会显示一个消息对话框“Please enter an integer between 1 and 234”。在用户关闭这个消息对话框之后，DDV_MinMaxUInt()函数会调用 CDataExchange::Fail()。

CDataExchange::Fail()重新将焦点设置为编辑框，然后抛出一个异常。这个异常会通知 UpdateData()已经出现了错误，然后给 OnOK()返回 FALSE。当 OnOK()意识到

UpdateData(TRUE)失败了，它不会调用 EndDialog()，而是仍然显示对话框，这样终端用户必须修改刚刚发生验证错误的控件中的值并重新输入新值。

这么简单的 DDX/DDV 函数所完成的工作很多，不是吗？

我们已经了解了 CDialog 类中有趣的部分，下面让我们看一下 CDialog 的一些派生类，也就是 MFC 公用对话框类，看它们如何使用并增强现在的 CDialog 接口。

MFC 公用对话框

微软在 Windows 3.1 中介绍了很多公用对话框，目的是为公用操作（例如保存/打开文件、选择字体、查找/替换字符串、打印和选择颜色）提供一个标准的用户界面。随着 Windows 95 的发布，公用对话框又有了新的风格，现在主要就是 Windows 95 Explorer 风格。

MFC 公用对话框封装了 Windows 公用对话框 API，并给它们一个 CDialog 接口（例如 DoModal()和 OnInitDialog()）。表 6-1 列出了 MFC 公用对话框类和它们所执行的操作。

表 6-1 MFC 公用对话框类

MFC 类名	操作
CColorDialog	让用户选择一种颜色
CFileDialog	用于打开和保存文件
CFindReplaceDialog	允许用户查找和替代项
CFontDialog	提供了选择字体的对话框
CPrintDialog	让用户选择打印机、页数、打印方向
CPageSetupDialog	一个 Win95 的公用对话框，它替代了 CPrintDialog，并增加了一些页面设置选项，例如页眉等

版本注释

自 MFC 1.0 以来，MFC 公用对话框就在 MFC 中（记录是从 2.0 版本开始的）。CPageSetupDialog 是一个特例，它是在 MFC 4.0 中出现的。

下面我们看如何使用公用对话框，然后看 MFC 中如何实现公用对话框。

使用公用对话框类

使用公用对话框的步骤如下：

1. 调用构造函数，传入恰当的参数使得公用对话框适合你的需要。（例如，你可以指定文件过滤器和一系列标志）。

2. 调用 DoModal()。

3. 如果 DoModal()的返回值是 IDOK，你就可以调用这个类的其他成员函数从而获得用户指定的信息。

程序清单 6-15 是一个简单的示例，显示了如何创建一个 File Open 对话框。

程序清单 6-15 MFC 公用对话框类的使用示例

```
CFileDialog myDialog(TRUE,NULL,NULL,OFN_FILEMUSTEXIST);
if (myDialog.DoModal() == IDOK) {
    //Now we can get the filename, path and others.
    CString strPath = myDialog.GetPathName();
    CString strFile = myDialog.GetFileName();
}
else
    //User selected cancel...
```

虽然这个例子中主要使用了 CFileDialog，但其他的 MFC 公用对话框也是这样使用，只不过返回颜色、字体和相应的其他信息等等。

提供一个新的对话框模板，指定一个截取关键消息的 hook（回调），你就可以自定制 Windows 3.x 公用对话框。Windows 95 Explorer 风格的公用对话框也有这一特性，但你不用提供一个完全新的对话框模板，你只要创建一个模板，该模板中包含了对话框内的控件。

为了使用 MFC 公用对话框类进行高级定制，你要创建公用对话框的一个派生类，并在构造函数中设置 OPENFILENAME 结构中的标志。例如，你要设置 OFN_ENABLETEMPLATE 来指定你是否要使用自己的模板。下一步，你要将 OPENFILE 结构的一个域指向你的对话框模板的句柄。

你也可以不提供 hook，只要覆盖一些虚函数来解释各种事件。例如，如果你想要检查一个文件的名字，可以覆盖 OnFileNameOK()，并在对话框关闭之前查看是否有文件名字。

以上就是如何使用 MFC 公用对话框类。下面我们看这些对话框的 MFC 代码。

公用对话框内幕

为了理解 MFC 公用对话框类如何运作，你需要知道 Windows 公用对话框如何从 Win32 API 搭建而成。每个对话框都有一个结构，这个结构的域描述了对话框的设置。每个对话框都有一个相应的 API，这个 API 以这个结构做参数，并负责显示对话框、与用户交互并返回。例如，对于打开文件对话框，这个结构是 OPENFILENAME，对应的 API 是 GetOpenFileName (LPOPENFILENAME)。

每个这样的结构和函数在几乎每本 Windows 编程书中都有详细的记录，所以我们不会详细讨论这部分。我们所关心的是 Win32 公用对话框函数的内部都发生了什么。记住了这一点，就让我们看看 MFC 公用对话框类扮演了什么样的角色。

MFC 公用对话框类的声明在头文件 AFXDLGS.H 中，它的实现分布在很多个源文件中。表 6-2 列出了每个 MFC 公用对话框类的实现所在的文件。

表 6-2 公用对话框类的源文件

类名	源文件名
CColorDialog	DLGCLR.CPP
CFileDialog	DLGFILE.CPP
CFindReplaceDialog	DLGFR.CPP
CFontDialog	DLGFNT.CPP
CPrintDialog	DLGPRNT.CPP
CPageSetupDialog	DLGPRNT.CPP

在看 AFXDLGS.H 头文件的时候，你可能会发现所有的公用对话框类都是从 CCommonDialog 派生的，而不是直接从 CDialog 派生的。下面让我们先看看这个公用对话框基类。

CCommonDialog:公用对话框基类

CCommonDialog 很小，它定义了两个虚函数。每个公用对话框类都会继承这两个虚函数。程序清单 6-16 列出了 CCommonDialog 的声明，在文件 AFXDLGS.H 中。

程序清单 6-16 CCommonDialog 的声明（在文件 AFXDLGS.H 中）

```
class CCommonDialog : public CDialog
{
public:
    CCommonDialog(CWnd* pParentWnd);
// Implementation
protected:
    virtual void OnOK();
    virtual void OnCancel();
};
```

CCommonDialog 为所有公用对话框的 OnOK()和 OnCancel()处理预留了位置。现在我们看 CCommonDialog:: OnOK()函数的实现（在文件 DLGCOMM.CPP 中），看它完成了什么（程序清单 6-17）。

程序清单 6-17 CCommonDialog::OnOK()的实现

```
void CCommonDialog::OnOK()
{
    if (!UpdateData(TRUE)) {
        TRACE0("UpdateData failed during dialog termination.\n");
        // the UpdateData routine will set focus to correct item return;
    }
    // Common dialogs do not require ::EndDialog
    Default();
}
```

CCommonDialog::OnOK()首先调用我们前面总是提到的 UpdateData()，因为 MFC 的 CCommonDialog 派生类都没有 DoDataExchange()成员函数，所以 UpdateData()调用要确定你的 CCommonDialog 派生类的 DoDataExchange()方法被调用。调用 Default()使得公用对话框确定它们是否有必要调用 EndDialog()。

探索 CFileDialog

公用对话框都有类似的实现，所以我们不会看每个类，只看 CFileDialog 这个类。我们在 CFileDialog 中发现的细节对于其他公用对话框类来说也是成立的。程序清单 6-18 中列出了 CFileDialog 的缩减后的声明，在文件 AFXDLGS.H 中。

程序清单 6-18 CFileDialog 的声明（在文件 AFXDLGS.H 中）

```
class CFileDialog : public CCommonDialog
{
public:
// Attributes
```

```
OPENFILENAME m_ofn; // open file parameter block
// Constructors **omitted
// Operations **omitted
// Overridable callbacks
protected:
    friend UINT CALLBACK _AfxCommDlgProc(HWND, UINT, WPARAM, LPARAM);
    virtual UINT OnShareViolation(LPCTSTR lpszPathName);
    virtual BOOL OnFileNameOK();
    virtual void OnLBSelChangedNotify(UINT nIDBox, UINT iCurSel, UINT nCode);
    // only called back if OFN_EXPLORER is set
    virtual void OnInitDone();
    virtual void OnFileNameChange();
    virtual void OnFolderChange();
    virtual void OnTypeChange();

// Implementation
protected:
    BOOL m_bOpenFileDialog;     // TRUE for file open, FALSE for file save
    CString m_strFilter;          // filter string
    TCHAR m_szFileTitle[64];      // contains file title after return
    TCHAR m_szFileName[_MAX_PATH]; // contains full path name after return
    OPENFILENAME*  m_pofnTemp;
    virtual BOOL OnNotify(WPARAM wParam, LPARAM lParam, LRESULT* pResult);
};
```

在 CFileDialog 的//Attribute 部分，你会发现有一个 public 的 OPENFILENAME 成员变量 m_ofn。这个结构就是传递给 GetOpenFileName()的结构。CFileDialog 的大部分代码都是以各种形式对这个结构进行操作。

CFileDialog 的另外一个有趣的部分就是 protected 的虚函数集。如果你将这个链表和已有文档说明（至少在 MFC 4.0 中）的链表相比较，你会发现这个类中有很多没有文档说明的回调函数。我们会在后面详细讨论这些函数。你还会发现一个辅助函数_AfxCommDlgProc()声明为这个类的友元。

最后在//Implementation 部分，我们看到有很多标准缓冲区。这些缓冲区中存储了类似过滤器、文件头、文件名之类的东西。在调用 DoModal()之后，当用户调用类似 GetFileName()之类的方法时，这些缓冲区中就会包含请求的信息。

在//Implementation 部分还有一个神秘的 OPENFILENAME 指针 m_pofnTemp 和一个虚函数 OnNotify()。

在我们探索 CFileDialog 的这些秘密之前，我们先看构造函数和 DoModal()的实现，这样我们就会对公用对话框的创建和显示有所了解。

CFileDialog 的构造函数和 DoModal()

CFileDialog 的构造函数初始化它的成员变量，然后处理构造函数的参数，并将它们转成 m_ofn 这个 OPENFILENAME 结构中相应的设置。

除此之外，如果构造函数是在 Windows 95 下被调用的，它还会设置 OFN_EXPLORER 标志和 OFN_ENABLEHOOK 标志。OFN_EXPLORER 确保创建的对话框是 Windows 95 风格的。OFN_ENABLEHOOK 标志使得 MFC 能够为公用对话框提供一个 hook 函数。在设置 OFN_ENABLEHOOK 标志之后，CFileDialog 的构造函数就将 m_ofn 中的 hook 域设置为

_AfxCommDlgProc()。

因为 m_ofn 结构是 public 的，所以你既可以在构造函数中设置它们，也可以直接访问 CFileDialog 的实例来修改它们。

记住 CFileDialog 的构造函数的行为，下面让我们看看程序清单 6-19 中列出的 CFileDialog::DoModal()实现的伪代码。

程序清单 6-19 CFileDialog::DoModal()的伪代码（在文件 DLGFILE.CPP 中）

```
int CFileDialog::DoModal()
{
    int nResult;
    BOOL bEnableParent = FALSE;
    m_ofn.hwndOwner = PreModal();
    AfxUnhookWindowCreate();
    if (m_ofn.hwndOwner != NULL && ::IsWindowEnabled(m_ofn.hwndOwner)){
        bEnableParent = TRUE;
        ::EnableWindow(m_ofn.hwndOwner, FALSE);
    }
    if (m_bOpenFileDialog)
        nResult = ::GetOpenFileName(&m_ofn);
    else
        nResult = ::GetSaveFileName(&m_ofn);
    if (bEnableParent)
        ::EnableWindow(m_ofn.hwndOwner, TRUE);
    PostModal();

    return nResult ? nResult : IDCANCEL;
}
```

首先，CFileDialog::DoModal()调用了 CDialog::PreModal()，然后将父窗口的句柄存储在 m_ofn 的 hWndOwner 域中。

然后 DoModal()禁用了父窗口（如果有父窗口），并强制模态化。

在禁用父窗口之后，DoModal()检查由 CFileDialog 构造函数设置的 m_bOpenFileDialog 成员变量。如果是 TRUE，就调用 Win32 API ::GetOpenFileName()，因为用户想使用打开文件对话框。

如果 m_bOpenFileDialog 的值是 FALSE，表示调用 DoModal()的用户想用保存文件对话框，所以它就调用 Win32 API ::GetSaveFileName()。DoModal()将 m_ofn 成员变量作为参数传递给这两个 API。

到此为止，对话框已经显示了，直至用户点击了 OK 按钮或 Cancel 按钮。GetOpenFileName()/GetSaveFileName()的返回值会被存储在本地变量 nResult 中。

一旦公用对话框的 API 返回了，并且如果父窗口被禁用，则 DoModal()就会激活父窗口，调用 PostModal()，然后如果返回结果，并且如果返回结果大于 0，就直接返回这个结果，否则返回 IDCANCEL。

关键地方在哪里？

到现在 CFileDialog 的内容都是可以预见到的。构造函数设置了 m_ofn 的域，DoModal()调用了公用对话框 API。除此之外还有一些没有文档说明的东西很有趣，还有我们在 CFileDialog 声明中看到的那些虚回调函数。

回调函数的关键是 hook 过程_AfxCommDlgProc()，它在 CFileDialog 的构造函数中被创建（其他公用对话框也是如此）。

因为_AfxCommDlgProc()是所有公用对话框的 hook 过程，所以你可以想像到它是一个很长的函数。程序清单 6-20 中包含了_AfxCommDlgProc()函数的缩减版本，这些代码在文件 DLGCOMM.CPP 中。

程序清单 6-20 MFC 公用对话框的 hook 函数_AfxCommDlgProc

```
UINT CALLBACK _AfxCommDlgProc(HWND hWnd, UINT message, WPARAM wParam,
                              LPARAM lParam)
{
    //Lots of code omitted, including pDlg declaration.
    if (message == nMsgSHAREVI)
        return ((CFileDialog*)pDlg)->OnShareViolation((LPCTSTR)lParam);
    else if (message == nMsgFILEOK) {
        if (afxData.bWin4)
            ((CFileDialog*)pDlg)->m_pofnTemp = (OPENFILENAME*)lParam;
            BOOL bResult = ((CFileDialog*)pDlg)->OnFileNameOK();
            ((CFileDialog*)pDlg)->m_pofnTemp = NULL;
            return bResult;
        }
    else if (message == nMsgLBSELCHANGE){
        ((CFileDialog*)pDlg)->OnLBSelChangedNotify(wParam, LOWORD(lParam),
        HIWORD(lParam));
        return 0;
    }
    else if (message == _afxNMsgSETRGB)
        return 0;
    return 0;
}
```

AfxCommDlgProc()检查消息参数的所有可能值并对相应的类型调用适当的虚回调函数。对于某些虚回调函数，例如 OnShareViolation()，AfxCommDlgProc()只要调用函数，并传入一个参数即可。而对于其他回调函数，例如 OnFileNameOK()，就需要做一些额外的工作。

对于 OnFileNameOK()，_AfxCommDlgProc()先将 lParam 参数所指向的 OPENFILENAME 保存在 m_pofnTemp 变量中，然后才调用 OnFileNameOK()。通常在 OnFileNameOK()中你可以查看 m_ofn.lpszFileName，那为什么微软还要在 m_pofnTemp 中保存 OPENFILENAME 指针呢？看起来微软这样做要么是 Windows 的一个 bug，要么是因为保持和 Macintosh 的兼容。

这些听起来还很平淡无奇，但是更大的问题是："nMsgSHAREVI"消息是什么？_AfxCommDlgProc()正在检查的其他值是什么？

如果你看 DLGCOMM.CPP 的前面，你会发现答案。MFC 根据公用对话框定义的字符串注册了很多消息。例如下面就是一个消息的声明，这个消息指明有一个共享违规：

```
static const UINT nMsgSHAREVI = ::RegisterWindowMessage(SHAREVISTRING);
```

SHAREVISTRING 是一个宏，它由 Win32 公用对话框定义。这些消息在每个 MFC 程序的起始位置定义（即使你不使用公用对话框），_AfxCommDlgProc()使用这些宏来对公用对话框发送的消息解码，并将它们转化成对公用对话框类的虚回调函数的调用。

但是还有一个问题。_AfxCommDlgProc()调用了 3 个虚回调函数：OnShareViolation()、OnFileNameOK()和 OnLBSelChangedNotify()。那么那些没有文档说明的虚回调函数（OnInitDone()、OnFileNameChange()、OnFolderChange()和 OnTypeChange()）呢？

答案就在 Windows 95 的公用对话框里。这些对话框使用不同的机制在回调函数和应用程序之间通信。Windows 95 的公用对话框没有使用我们前面提到的注册消息机制，而是发送 WM_NOTIFY 消息。如果不处理 WM_NOTIFY 消息，就会发送那些注册了的消息。

Windows 95 回调函数的生成者：CFileDialog::OnNotify()

没有文档说明的 CFileDialog::OnNotify()成员完成了将 Windows 95 的 WM_NOTIFY 消息转化成没有文档说明的虚回调函数的大部分工作。程序清单 6-21 列出了 CFileDialog::OnNotify()代码的缩减版本，在文件 DLGFILE.CPP 中。

版本注释

这个机制只在 MFC 4.0 及更高版本中才有。

程序清单 6-21 CDialog::OnNotify()的伪代码（在文件 DLGFILE.CPP 中）

```
BOOL CFileDialog::OnNotify(WPARAM wParam, LPARAM lParam, LRESULT* pResult)
{
    OFNOTIFY* pNotify = (OFNOTIFY*)lParam;
    switch(pNotify->hdr.code)
        {
        case CDN_INITDONE:
            OnInitDone();
            return TRUE;
        case CDN_SELCHANGE:
            OnFileNameChange();
            return TRUE;
        case CDN_FOLDERCHANGE:
            OnFolderChange();
            return TRUE;
        case CDN_SHAREVIOLATION:
            *pResult = OnShareViolation(pNotify->pszFile);
            return TRUE;
        case CDN_FILEOK:
            *pResult = OnFileNameOK();
            return TRUE;
        case CDN_TYPECHANGE:
            OnTypeChange();
            return TRUE;
    }
    return FALSE;   // not handled
}
```

首先，OnNotify()从 lParam 参数中取出 OFNOTIFY 指针。它最感兴趣的是 pNotify->hdr.code，这个结构中包含了通知代码。然后，OnNotify()关闭了通知代码，调用了相应的虚回调函数。如果已经处理了消息，这些函数就会返回 TRUE。

这里有一个有趣的问题：为什么_AfxCommDlgProc()没有使用 switch 语句，而是使用复杂的 if/else 结构呢？

_AfxCommDlgProc()中检查的值在编译时是不可知的，所以它不得不用 if/else 语句，而没有用 switch 语句。

与原有的_AfxCommDlgProc()方法相比，CDialog::OnNotify()函数处理了更多的通知，这样也就更容易定制 Windows 95 公用对话框。

MFC CFileDialog 类还有另外一个没有文档说明的方法供那些想要定制文件对话框的开发人员使用。这就是成员函数 SetTemplate()。这个函数有两个参数，都是对话框模板，一个是旧式的文件对话框模板，另一个是新式的文件对话框模板。这个没有文档说明的函数负责为你确定并使用正确的对话框模板，这样你就不用在代码中担心这些问题了。如果你还想支持旧式的对话框，这一点很方便。

现在你应该对公用对话框中使用的小技巧很了解了。在我们继续之前，让我们再看一个成员函数，这个函数负责从 CFileDialog 中提取信息。这个函数就是 GetFileName()，它从对话框中提取用户输入的文件名，并不带目录。程序清单 6-22 列出了这个函数的实现，在文件 DLGFILE.CPP 中。例如，如果我们输入了 C:\SCOT\BOOK\CHAP6.DOC，GetFileName()就会返回字符串“CHAP6.DOC”。

程序清单 6-22 CFileDialog::GetFileName()的实现（在文件 DLGFILE.CPP 中）

```
CString CFileDialog::GetFileName() const
{
    if ((m_ofn.Flags & OFN_EXPLORER) && m_hWnd != NULL){
        CString strResult;
        if (GetParent()->SendMessage(CDM_GETSPEC, (WPARAM)MAX_PATH,
          (LPARAM)strResult.GetBuffer(MAX_PATH)) < 0)
            strResult.Empty();
        else{
            strResult.ReleaseBuffer(); return strResult;
        }
    }
    return m_ofn.lpstrFileTitle;
}
```

GetFileName()首先检查对话框是否是 Explorer 风格（OFN_EXPLORER），然后确定它有一个合法的窗口句柄。如果条件满足，它就使用新对话框的特性来发送消息（CDM_GETSPEC），以获取文件的名字。如果 CDM_GETSPEC 消息返回了大于零的值，就表示成功，GetFileName()就会返回一个字符串。如果 SendMessage()执行失败了，或者对话框不是 Explorer 风格的，GetFileName()就会转化成旧式的方法，并返回 m_ofn 的 lpstrFileTitle 域。这个域也是由公用对话框设置的。

在 Windows 95 公用对话框中，发送消息给对话框更可靠；微软鼓励 Windows 95 开发人员使用这种新技术，而不要再检查 m_ofn 的域。

MFC 公用对话框的包装

我们已经查看过 CFileDialog 类的很有有趣的部分，下面我们再看一下 OLE 对话框。在我们继续之前，有几个关于 MFC 公用对话框的要点需要记住：

- 所有的 MFC 公用对话框都是从 CCommonDialog 派生的，CCommonDialog 提供了

OnOK()和 OnCancel()两个函数。

- 公用对话框是 Win32 结构和 API 的一个包装层。在我们的例子中，就是 OPENFILE 结构和 GetOpenFile()/GetSaveFile() API 的包装层。
- MFC 在内部会检查 OFN_EXPLORER 标志，以区分旧式对话框和新式对话框，然后根据结果调用适当的方法。如果有 hook 和对话框模板，这一点就很有用。
- 在公用对话框的 hook 函数中还调用了很多没有文档说明的回调函数。在考虑使用你自己的 hook 函数之前，请对你的 MFC 版本做双重检查，从而看一下你的 hook 函数是否是会被处理但却是没有文档说明的。

其他信息

不幸的是，我们不能在这里分析每个 MFC 公用对话框。但是，如果你对这个有兴趣，下面我们会给你指引方向。

如果你看过 Scribble 指南，你会回忆起来这个应用程序从第一步开始就有一个文件打开对话框，但是在源文件中没有任何指向 CFileDialog 的引用。

考虑下面几个问题：

- CFileDialog 在哪儿呢？
- 如果你有 CFileDialog 的一个派生类，它用于在应用程序中完成特定的任务，你如何让 Scribble 创建一个你的 CFileDialog 派生类的实例，而不是创建 CFileDialog 的一个实例呢？
- 你知道为什么 CFileDialog 有 ofnTemp 这个数据成员吗？

看另外一个公用对话框的实现，看你是否发现了什么没有文档说明的函数和成员变量。你能推断出这些没有文档说明的成员是做什么的吗？下面是帮助回答这个问题的几个问题：

- 为什么 CFontDialog 有一个没有文档说明的 LOGFONT 成员变量（m_lf）？
- 为什么 CColorDialog 需要有一个 OnCtlColor 成员？它是做什么的？
- _AFX_COLOR_STATE 是什么？
- CPageSetupDialog::PaintHookProc()又完成了什么？为什么需要它？
- 你认为 DLGPRNT.CPP 中的 AfxCreateDC()函数的功能是什么？

在 MFC 程序清单下隐藏的是信息的金矿。这些只是帮助你挖掘这些信息的引子。

OLE 对话框

OLE 动态链接库（OLEDLG.DLL）中包含了几个“OLE 公用对话框”，它为用户完成了特定的 OLE 任务。例如，当用户插入一个新的对象时，就会有一个封装好的对话框要求用户从已注册的对象类型中选择一个。

版本注释

在 MFC 3.x 及更老的版本中，OLE 对话框的代码在文件 MFCUIA32.DLL 中。随着 Windows 95 和 NT 3.51 的出现，这些函数就和操作系统合并到一起了。

通过使用公用的 OLE 对话框（类似一般的公用对话框），你可以确保终端用户在他们的

Windows（MFC）应用程序中见到的是类似的 OLE 接口。

下面是 OLE 对话框的小结，以及它们的功能：

- COleDialog——所有 OLE 对话框的基类。
- COleInsertDialog——提示用户要插入的对象的类型；通常是从菜单中选择 Edit/Insert Object 选项时创建的对话框。
- COleConvertDialog——使得用户可以修改嵌入对象的类型。
- COleChangeIconDialog——允许用户修改特定类型的嵌入对象的图标。
- COlePasteSpecialDialog——允许用户选择他（或她）是否想在文档中粘贴一个对象、链接文档或向文档中拷贝内容。通常通过选择 Edit/Paste Special 菜单显示。
- COleLinksDialog——允许用户修改 OLE 链接。
- COleUpdateDialog——允许用户更新 OLE 链接。
- COleBusyDialog——当 OLE 服务器忙而不能对请求作出响应时显示。
- COlePropertiesDialog——为 OLE 对象显示属性页。
- COleChangeSourceDialog——允许用户修改链接的源和目的地。

使用 OLE 对话框

在大多数情况下，MFC OLE 类处理 OLE 对话框的创建工作。MFC OLE 对话框的使用方法和公用对话框一样。你只要创建它的一个实例，然后调用 DoModal()。最后如果 DoModal() 的返回值是 IDOK，就获取用户选择的信息。

MFC OLE 对话框类内幕

OLE 对话框类的声明在文件 AFXODLGS.H 文件中。对话框的实现在 3 个文件中，分别是 OLEDLGS1.CPP、OLEDLGS2.CPP 和 OLEDLGS3.CPP。下面是哪个类位于哪个文件的一个指南：

- OLEDLGS1.CPP——COleInsertDialog、COleConvertDialog、COleChangeIconDialog、COleLinksDialog、COleUpdateDialog、COlePasteSpecialDialog。
- OLEDLGS2.CPP——COleDialog、COleBusyDialog。
- OLEDLGS3.CPP——COlePropertiesDialog、COleChangeSourceDialog。

OLE 对话框还有一些成员的实现在内嵌文件 AFXOLE.INL 中。

OLE 对话框和这一章前面提到的公用对话框很相似。每个 OLE 对话框都有一个结构和一个 API。结构定义了对话框的选项，API 负责使用结构中的域设置显示对话框。例如，COlePasteSpecialDialog 这个 OLE 对话框类封装了 OLEUIPASTESPECIAL 结构和 OleUIPasteSpecial()这个 OLE API。

MFC 公用对话框和 OLE 对话框类之间的最大区别就在于，在继承关系中有一个新的类介于 CCommonDialog 和每个 OLE 对话框之间，那就是 COleDialog。

COleDialog 内幕：OLE 对话框的基类

程序清单 6-23 列出了 COleDialog 声明（在文件 AFXODLGS.H 中）的缩减版本。

程序清单 6-23 COleDialog 这个 OLE 对话框基类的声明（在文件 AFXODLGS.H 中）

```
class COleDialog : public CCommonDialog
{
    DECLARE_DYNAMIC(COleDialog)
// Attributes
public:
    UINT GetLastError() const;
// Implementation
public:
    int MapResult(UINT nResult);
    COleDialog(CWnd* pParentWnd);

protected:
    UINT m_nLastError;
protected:
    friend UINT CALLBACK _AfxOleHookProc(HWND, UINT, WPARAM, LPARAM);
};
```

COleDialog 是一个很小的基类，它只增加了两个成员：一个是 UINT 类型的 m_nLastError 成员变量，用于保存错误码；另一个是 MapResult()成员函数，用于将一些 OLE 返回码映射为 MFC 更熟悉的 IDOK 和 IDCANCEL。GetLastError()成员函数值返回 m_nLastError 的内容。

它的 hook 过程和_AfxCommDlgProc()、_AfxOleHookProc()一样，在基类中都声明为友元，这样每个派生类都不用将_AfxCommDlgProc()声明为友元。

OLE 对话框总结

关于 OLE 对话框的总结如下：

- 它们和公用对话框类在使用上和实现上都很相似。
- OLE 对话框都是从 COleDialog 派生的，COleDialog 只是增加了一些错误处理和一些成员函数。
- MFC 已经创建了很多标准的对话框，你可以根据应用程序的环境决定是否使用它们。

更多的信息

如果你是 OLE 的忠实跟随者，如果你想要学到更多的东西，下面是为你提供的问题，你可以带着这些问题翻阅 MFC 程序清单：

- 你认为_AfxOlePropertiesEnabled()和_AfxParseDisplayName()的功能是什么？
- 有一个没有文档说明的类 COleUILinkInfo。它是用来干什么的？为什么它是没有文档说明的呢？你能在自己的应用程序中使用这个类吗？
- 你能推测出 COlePropertiesDialog 的每个成员的功能吗？
- COlePropertyPage 和 COlePropertiesDialog 之间的关系是什么？

到目前为止，我们几乎看到了 CDialog 的每个派生类。但是如果你有 MFC 的一个继承图表，你会发现 CDialog 还有一个派生类 CPropertyPage。这个派生类是生成 MFC 属性页（也称带标签的对话框）的关键。

属性页（也称带标签的对话框）

MFC 属性页对话框是从 MFC 3.0 开始引入的。图 6-4 显示了属性页的一个示例。属性页后来越来越普遍是因为它使得用户可以在一个对话框中完成大量操作，而不用多次进行菜单/对话框组合的切换。例如，Visual C++的 Project/Options 这个属性页。如果每个标签对应的对话框都独立出来的话，想像一下那样修改选项要花费多长时间！

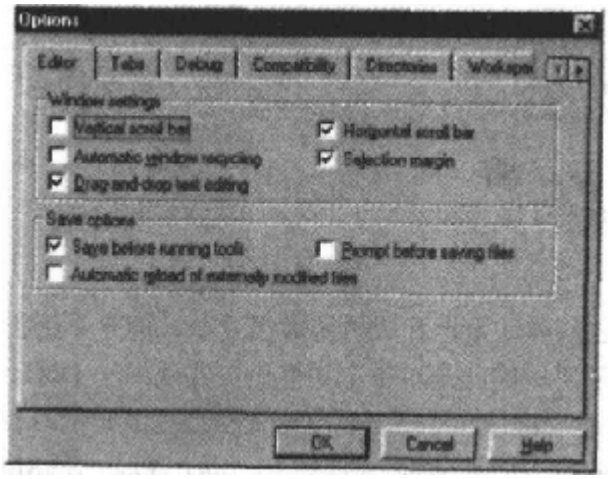


图 6-4 属性页的一个示例

使用 MFC 属性页

MFC 中属性对话框的实现基于两个类：CPropertySheet 和 CPropertyPage。CPropertySheet 是“带标签的对话框”类，CPropertyPage 则在带标签的对话框中封装了“标签”或页的类。

为了使用 MFC 属性页，你要按照下面的步骤操作：

1. 创建多个对话框模板，使模板能描述你想要在属性页中布置的控件和布局。

2. 为每个对话框模板创建 CPropertyPage 的一个派生类，并在类中增加成员变量来存储属性页的状态。

3a. 对于一个模态的属性页，只要创建它的 CPropertySheet 实例，然后调用 CPropertySheet::AddPage()将前两步创建的单个属性页加入到属性页中。增加了这些属性单页之后再调用 CDialog::DoModal()来显示属性页。MFC 会为你自动增加 OK/Apply/Cancel 按钮。

3b. 如果你想创建一个非模态的属性页，那么你所创建的 CPropertySheet 派生类必须要增加一些东西，以便能够关闭对话框。如果你创建了一个非模态的对话框，则 MFC 不会为你自动增加 OK/Apply/Cancel 按钮。然后创建你的 CPropertySheet 派生类的一个示例，并调用它的 Create()来显示非模态属性页。

程序清单 6-24 就是一个带有 3 个标签的属性页。3 个属性单页分别是 CPageOne、CPageTwo 和 CPageThree，它们都是 CPropertyPage 的派生类。

程序清单 6-24 如何创建一个带有 3 个标签的模态属性页的示例

```
//...

CPropertySheet mySheet("My Cool Property Sheet!",this);
// Now create the property pages for the sheet.
CPageOne myPage1;
CPageTwo myPage2;
CPageThree myPage3;

// and insert them into the sheet
mySheet.AdPage(&myPage1);
mySheet.AdPage(&myPage2);
mySheet.AdPage(&myPage3);

// DoModal, gotta love it!
mySheet.DoModal();

//...
```

在不同的属性单页之间移动数据和成员函数很复杂，但是不用担心：你可以对属性页使用 DDX 和 DDV。你所需要做的就是对每个属性单页实现一个 DoDataExchange()成员函数。MFC 属性页还有一个有趣的特性。用户可以在对话框还在显示的时候使修改生效，而不需要关闭对话框。为了启用 Apply 按钮，你必须调用 CPropertyPage::SetModified(TRUE)通知 MFC 有属性被修改了。当用户点击 Apply 按钮时，你的成员变量就会更新为新的值，然后你就要相应地更新用户界面。每个属性页面有一个修改标志，所以当用户点击 Apply 按钮时，所有的内容不是立刻都能应用的，只有当前属性单页能应用。

对于很多 MFC 编程人员来说属性页是一个很重要的工具。下面我们看属性页是如何实现的，并发掘 MFC 属性页和属性单页类内部隐藏了什么。

属性页和属性单页内幕

现在，MFC 中的两个属性页类 CPropertySheet 和 CPropertyPage 的地位出现危机。因为属性页并不是从 CDialog 派生的，而属性单页却是从 CDialog 派生的。

CPropertySheet 是从 CWnd 派生的，而不是从 CDialog 派生的，因为 CDialog 是对 Windows 对话框的封装，而 Windows 对话框是和对话框模板资源绑定在一起的。又因为 CPropertySheets 与对话框模板没有丝毫关系，所以它应该是一个窗口，即 CWnd 的派生类。事实上，当 CPropertySheet 出现时，MFC 小组将 CDialog 的很多方法（例如 RunModalLoop()）都放到了 CWnd 里。这使得同样的函数不用重复出现：CDialog::RunModalLoop()和 CPropertySheet::RunModalLoop()。

这也就解释了为什么 CPropertyPage 从 CDialog 派生。因为 CPropertyPages 封装了一个对话框模板，所以在 MFC 层次结构中恰好在 CDialog 下面。

源文件——关于版本的一些话

你可以在源文件 AFXDLGS.H 中找到 CPropertySheet 和 CPropertyPage 的声明。这两个类的实现都在文件 DLGPROP.CPP 中，还有一些内嵌函数在文件 AFXDLGS.INL 中。

随着 MFC 4.0 的发布，CPropertySheet 和 CPropertyPage 的内部都发生了很大变化。在

MFC 3.0 中，属性页的实现是完全包含在 MFC 中的。MFC 负责完成所有标签的绘制、窗口的创建、标签的管理等等。那时，Windows 通用控件（一系列从 Windows 95 演变出来的新控件）还没有发布，所以 MFC 组走在了前面，它在 MFC 内部实现了自己的属性页。

在 MFC 4.0 中，也就是 Windows 95 发布后的一个主要修改版本，MFC 又重写了属性页，这一次他们使用了 Windows 通用控件中的 Win 32 属性页的实现（这个实现使用了公用标签控件）。现在 MFC 不用再管理属性页的每个方面，Windows 通用控件处理了大部分工作。MFC 只是为新的 Windows 通用控件提供了自 3.0 版本以来相同的接口。除了我们前面提到的接口，MFC 4.0 还增加了一个“向导”模式，这种模式可以引导你使用类似的 MFC 属性页类创建向导。这个功能也来自封装的 Windows 通用控件。

MFC 属性页不是在这个过程中惟一重写的类。我们在第 9 章中提到的一些类（例如工具栏和状态栏）也使用新的 Windows 通用控件重写了一遍。

为了使事情复杂化，还有一些 MFC 类提供了对 Windows 通用控件的简单封装，其中包括一个对属性页封装的类 CTabCtrl。我们会在后面说这个 CTabCtrl 类，但是在大多数情况下，对于属性页来说，使用 CPropertySheet/CPropertyPage 这两个接口比使用更低级更难使用的 CTabCtrl 方便。

MFC 小组用 Windows 通用控件重写了 CPropertySheet/CPropertyPage 之后，这两个类的代码量大大减少。如果你有一个旧版本的 MFC，可以打开 AFXDLGS.H，将里面的 CPropertySheet 和 MFC 4.0 以后版本的这个类进行比较。后者的成员大概只有前者的 1/3。但是不用担心，它的功能没受到影响。

回顾 Windows 属性页公用控件

既然 MFC 属性页封装了属性页 Windows 公用控件，那么为了理解 MFC 的 CPropertySheet/ CPropertyPage 这两个类，我们需要回顾如何使用属性页控件。

使用 Windows 属性页控件和使用公用的对话框差不多。首先，你要填写 PROPSHEETHEADER 结构的一些域。这个结构定义了属性对话框的设置，包括属性单页的信息。一旦你填好了这个结构，你就可以调用 Win 32 API PropertyPage()，这个函数以这个结构为参数，负责创建属性页。

PROPSHEETHEADER 结构很复杂，但是它的很多域对理解 MFC 属性页类很重要。其中一个重要的域就是 ppsp。这个域包含了一个指针，这个指针指向属性单页结构——PROPSHEETPAGE 数组。每个 PROPSHEETPAGE 结构定义了属性页中的一个页或标签。

PROPSHEETHEADER 还有一个域 nPages，用来告诉控件这个数组中有多少个属性单页结构。

这个结构的另一个重要域是 dwFlags，你可以利用它设置很多标志，使属性页处于不同的操作模式。属性页控件可以以两种方式显示标签“stacked”和“scrolled”。“scrolled”标签在属性页中是彼此相连的，需要时你可以用小滚动条上下滚动。

属性页标志都以“PSH”开头，下面是一些 MFC 属性页类所使用的标志链表：

- PSH_MODELESS——使属性页成为非模态的，而不是模态的。
- PSH_PROPSHEETPAGE——通知属性页有属性单页结构。它会忽略这些属性单页结构，除非设置了这个标志。

- PSH_MULTILINETABS——显示“stacked”标签而不显示“scrolled”标签。
- PSH_WIZARD——将属性页控件放入向导模式。在向导模式下，控件不会显示标签，而只是显示“Next”和“Previous”两个按钮，用户可以利用这两个按钮换页。

在使用 PropertySheet() API 创建了属性页之后，属性页会发送给你的应用程序 WM_NOTIFY 消息，因为用户开始和它交互了。这些通知也都会以“PSN”开始。下面是一些例子：

- PSN_APPLY——Apply 按钮被按下了。
- PSN_WIZBACK——在向导模式下，用户按下了 Previous 按钮
- PSN_WIZNEXT——在向导模式下，用户按下了 Next 按钮。
- PSN_WIZFINISH——在向导模式下，用户按下了 Finish 按钮。

如果你想获取信息（响应刚刚提到的通知），你也可以向属性页空间发送消息。这些消息以“PSM“开头。下面是一些例子：

- PSM_SETTITLE——设置属性页的标题。
- PSM_ADDPAGE——在属性页中增加一个单页。这个消息的 lParam 参数是一个指向已创建的属性单页（由 CreatePropertySheetPage()创建）的句柄。
- PSM_REMOVEPAGE——在属性页中删除一个属性单页。

简短地回顾已经足以说明 CPropertySheet/CPropertyPage 类的功能了。如果你想定制 MFC 属性页，或者学习更多关于 Windows 通用控件的内容，请查看完全版本的在线信息，并查看关于这方面的书。

现在我们要看 MFC 属性页类的内幕了。我们会自顶向下地发掘，首先看 CPropertySheet，然后再看 CPropertyPage。

CPropertySheet 内幕

CPropertySheet 是 CWnd 的派生类，它的任务是为属性页控件提供一个类似 CDialog 的接口。在我们下面的分析过程中，请注意 MFC 是如何将原有的 MFC 3.0 的 CPropertySheet 接口映射成新的 Windows 通用控件的。

程序清单 6-25 列出了 CPropertySheet 声明的一个缩减版本，在文件 AFXDLGS.H 中。

程序清单 6-25 CPropertySheet 声明的缩减版本（在文件 AFXDLGS.H 中）

```
class CPropertySheet : public CWnd
{
// Construction  **omitted
// Attributes ** mostly omitted
public:
    PROPSHEETHEADER m_psh;
// Operations ** omitted
public:
// Implementation  **some omitted
public:
    void CommonConstruct(CWnd* pParentWnd, UINT iSelectPage);
    void EnableStackedTabs(BOOL bStacked);
    virtual void BuildPropPageArray();
    virtual BOOL OnInitDialog();
protected:
```

```
    CPtrArray m_pages;      // array of CPropertyPage pointers
    CString m_strCaption;   // caption of the pseudo-dialog
    CWnd* m_pParentWnd;     // parent window of property sheet
    BOOL m_bStacked;        // EnableStackedTabs sets this
    BOOL m_bModeless;       // TRUE when Create called instead of DoModal

// Generated message map functions  **omitted
    friend class CPropertyPage;
};
```

和往常一样，我们只保留了我们感兴趣的//Implementation 部分，而忽略了其他部分。

首先注意 public //Attributes 部分，这里定义了一个名为 m_psh 的 PROPSHEETHEADER 结构。如果你刚刚看过 CFileDialog 的描述，你可能会有似曾相识的感觉。CFileDialog 和其他公用对话框都使用了同样的办法来维护一个 public 成员，所以在应用程序中需要你可以访问并修改对话框/控件结构。

我们直接进入//Implementation 部分，这里有很多成员函数和成员数据，简要说明如下：

- CommonConstruct()——CPropertySheet 有很多构造函数。它们都只是接收参数，然后传递给 CommonConstruct()，这样对于每个构造函数来说，就不用复制相同的构造代码了。CommonConstruct()首先设置了 m_psh 所指向的 PSH_PROPSHEETPAGE 结构的 dwFlags 域，然后初始化了 m_psh 所指向结构的几个其他域。CommonConstruct()还将 m_bStacked 设置为 TRUE，将 m_bModeless 设置为 FALSE。
- EnableStackedTabs()——将 CPropertySheet 设置为“stacked”模式而不是“scrolled”模式。这个内嵌函数只修改 m_bStacked 成员变量。
- BuildPropPageArray()——我们马上又会看到这个成员函数。我们会在后面介绍它。
- OnInitDialog()——一个很长的成员函数，它的执行步骤如下：

  1. 如果不是处于“向导”模式，就将整个属性页传递出去，以免控件绑在一起。
  2. 根据 m_bStacked 的值修改标签的风格。通过调用 EnableStackedTabs()完成。
  3. 如果属性页是非模态的，而且不处于“向导”状态，OnInitDialog()就删除标准的按钮（前面提到过 MFC 属性页没有 OK/Cancel/Apply 按钮）。
  4. 调用 CWnd::CenterWindow()，使属性页处于窗口的正中间。

- m_pages——一个指针数组，是 CPtrArray 的实例，它保存了指向属性页的所有 CPropertyPage 类的指针。
- m_strCaption——属性页的说明。
- m_pParentWnd——指向父窗口的指针。
- m_bStacked——一个 Boolean 变量，记录了用户指定的标签的风格是“stacked”还是“scrolled”。
- m_bModeless——一个 Boolean 变量，记录了用户希望属性页是模态的还是非模态的。

CPropertySheet::DoModal()

为了看所有这些成员如何协作，我们先看 CPropertySheet::DoModal()。程序清单 6-26 列出了 CPropertySheet::DoModal()的伪代码，在文件 DLGPROP.CPP 中。

程序清单 6-26 CPropertySheet::DoModal()的伪代码（在文件 DLGPROP.CPP 中）

```
int CPropertySheet::DoModal()
{
    // register common controls
    VERIFY(AfxDeferRegisterClass(AFX_WNDCOMMCTLS_REG));
    // finish building PROPSHEETHEADER structure
    BuildPropPageArray();
    pParentWnd->EnableWindow(FALSE);
    HWND hWnd = (HWND)::PropertySheet(&m_psh);
    nResult = RunModalLoop(dwFlags);
    // cleanup
    DestroyWindow();
    return nResult;
}
```

首先，DoModal()调用了一个神秘的函数 AfxDeferRegisterClass()。从而导致 MFC 调用了 InitCommonControls()函数，这个函数初始化了 Windows 通用控件 DLL。

然后，DoModal()调用了 BuildPropPageArray()。我们过一会儿会详细讨论这个函数。

在调用了 BuildPropPageArray()之后，DoModal()与 CDialog 和 CFileDialog 中的 DoModal()做同样的事。它会禁用主窗口，强制实现模态化，然后调用::PropertySheet() API 创建并显示公用属性页控件。然后，DoModal()调用 CWnd::RunModalLoop()在模态化过程中处理消息泵。在 RunModalLoope()完成后，DoModal()会销毁窗口，重新激活父窗口，然会返回 RunModalLoop()的返回值。我们忽略了其中一些关于 RunModalLoop()调用的细节，如果你要了解其中细节，请查看代码。

关于CPropertySheet需要记住的一个要点就是，直到DoModal()（对于模态属性页）或Create()（对于非模态属性页）被调用，类都不能映射到一个 Windows 窗口句柄。在::PropertySheet() API 被调用之后，Windows 窗口句柄才被创建，并和 MFC CPropertySheet 对象绑定在一起。后面你会发现 CPropertySheet 的成员执行了 m_hWnd!=NULL 检查。这就意味着，当对话框已经创建与对话框没有绑定到一个窗口句柄相比，有一些操作是不一样的。通常对于非模态的属性页来说这是个问题，因为它们的生存周期比模态属性页要长。

CPropertySheet::AddPage()

下面我们会分解 AddPage()函数，看 CPropertySheet 如何增加一个属性单页。程序清单 6-27 列出了 CPropertySheet::AddPage()的伪代码，在文件 DLGPROP.CPP 中。

程序清单 6-27 CPropertySheet::AddPage()的伪代码（在文件 DLGPROP.CPP 中）

```
void CPropertySheet::AddPage(CPropertyPage* pPage)
{
    m_pages.Add(pPage);
    if (m_hWnd != NULL) {
        HPROPSHEETPAGE hPSP = CreatePropertySheetPage(&pPage->m_psp);
        if (!SendMessage(PSM_ADDPAGE, 0, (LPARAM)hPSP)){
            DestroyPropertySheetPage(hPSP);
            AfxThrowMemoryException();
        }
    }
}
```

首先，AddPage()在它的内部变量 m_pages（属性单页指针数组）中保存了 CPropertyPage 指针。然后在检查了 m_hWnd 非空之后，AddPage()调用了::CreatePropertySheetPage()，并发送 PSM_ADDPAGE 消息给属性页控件，从而让它将 CreatePropertySheetPage 所创建的属性单页加进去。

你可能想知道为什么 CPropertySheet 维护了 m_pages 数组，它完全可以使用保存在控件中的属性单页。这其实只是时间的问题。在大多数情况下，m_hWnd != NULL 这个检查都会返回 FALSE，因为通常 MFC 用户都会为每个属性单页调用 AddPage()，然后调用 DoModal()或 Create()。m_pages 数组将所有在属性页创建之前加入的属性单页都缓冲了起来。

可能正如你所猜测的那样，BuildPropPageArray()就负责将 m_pages 数组转化成 PROPSHEETPAGE 结构，因为 m_psh 需要使用这个结构。下面我们来看看 BuildPropPageArray()，看转化过程中都用到了什么。

CPropertySheet::BuildPropPageArray()

程序清单 6-28 列出了 BuildPropPageArray()的伪代码，在文件 DLGPROP.CPP 中。

程序清单 6-28 BuildPropPageArray()的伪代码（在文件 DLGPROP.CPP 中）

```
void CPropertySheet::BuildPropPageArray()
{
    delete[] (PROPSHEETPAGE*)m_psh.ppsp;
    m_psh.ppsp = NULL;
    LPPROPSHEETPAGE ppsp = new PROPSHEETPAGE[m_pages.GetSize()];
    m_psh.ppsp = ppsp;
    for (int i = 0; i < m_pages.GetSize(); i++) {
        CPropertyPage* pPage = (CPropertyPage*)m_pages[i];
        memcpy(&ppsp[i], &pPage->m_psp, sizeof(PROPSHEETPAGE));
        // **omitted - find/load/lock ppsp[i] resource
        // of psp.pszTemplate into pTemplate
        _ChangePropPageFont(pTemplate);
    }
    m_psh.nPages = m_pages.GetSize();
}
```

首先，它删除了 m_psh.ppsp 域所指向的数组，并将它设置为 NULL，从而为新的数组保留控件。新的数组来自内部 CPtrArray 数组 m_pages。

然后，它为本地变量 ppsp 分配了一个新数组（数组的基类型是 PROPSHEETPAGES），并将 m_psh.ppsp 指向同一块内存。这样每次修改.ppsp 域时就不用访问 m_psh 结构了：它可以通过使用 ppsp 这个局部变量完成。

在设置了 ppsp 域后，它遍历 m_pages 数组中的所有 CPropertyPages。对于每个页，它将 CPropertyPage 的内部 PROPSHEETPAGE 结构拷贝到 PROPSHEETHEADER 数组 ppsp 中。拷贝结束后，它又调用了_ChangePropPageFont()来修改所有控件的字体，参数是一个指向对话框模板的指针，这样控件就可以和属性页使用相同的字体了。

一旦所有的页都放入了 m_psh.ppsp，它就会更新 m_psh.nPages 这个单页计数，以便和 m_pages 的大小相匹配。

CProperSheet 通知

CProperSheet 将几个属性页控件通知映射为可覆盖的回调函数。例如，在向导模式下，CProperSheet 会在它的 OnNotify() 处理函数中接收 WM_NOTIFY 消息（例如 PSN_WIZNEXT）。在调用的时候，它会调用虚函数 OnWizardNext()表示 Next 按钮被按下了。

CPropertyPage

和 CProperSheet 一样，CPropertyPage 也封装了一个结构，那就是 PROPSHEETPAGE。PROPSHEETPAGE 结构很直接，CPropertyPage 只是这个结构的一个很小的包装层：它在 PROPSHEETPAGE 之上增加了一些逻辑。

正因为这样，我们将 CPropertyPage 的探索留给了读者。和 CPropertySheet 一样，CPropertyPage 的声明在文件 AFXDLGS.H 中，实现在文件 DLGPROP.CPP 中。

属性页小结

我们已经知道的关于 MFC 属性页的基本内容如下：

- MFC 属性页类在属性页控件上提供了一个 CDialog 接口。为了这样做，它们就不得不做一些神秘的转化。
- 和公用对话框一样，这些类也封装了一个 Win32 结构和 API。这个结构是 public 的，而且类的任何使用者都可以操纵它。利用这些知识，你可以根据应用程序的需要在结构中设置各种标志。

更多的信息

如果你是属性页的忠实爱好者，下面这些问题会引导你达到一个有趣的领域：

- 探索 AfxDeferRegisterClass()。
- _ChangePropPageFont()这个静态函数如何知道 MFC 将使用哪种字体作为属性页的字体？
- CPropertySheet 有一个成员函数 GetTabControl()，它返回一个 MFC 控件类指针 CTabCtrl。这个指针指向 CPropertySheet 所使用的底层 Windows 属性页控件。
  1. GetTabCtrl()如何得到这个指针？
  2. 你为什么认为它们没有在 CPropertySheet 中使用 CTabCtrl，而认为它直接调用了 API？
- 看一下 MFC 4.0 之前版本的 CPropertyPage 和 CPropertySheet。
  1. 你能找到这个版本中它是如何绘制自己的吗？
  2. 这里面有没有文档说明的辅助类吗？
  3. 你认为哪个代码更简洁，MFC 4.0 之前的版本，还是 MFC 4.0？

现在我们已经探索了 MFC CDialog 的派生类。正如前面建议的，我们该分析 MFC 控件类了。

MFC 控件类

随着 Windows 通用控件类被引入 MFC，MFC 的控件类就分成了 3 个组：

- 旧式类，这些类封装了 6 个基本控件（自从 Windows 出现就有的六个控件：按钮、复选框、编辑框、列表框、滚动条和静态文本）。在 MFC 4.0 中，微软引入了 2 个扩展的控件：下拉链表（实际上是通用控件）和复选列表框。MFC 自从 2.0 版本后还增加了位图按钮类。
- 新式类，这些类封装了 Windows 通用控件。
- OLE 控件。OLE 控件有很多有趣的内幕，所以针对 OLE 控件单独做一章讨论（第 15 章），我们这里就不详细讲了。

在这部分我们首先看基本的控件类，然后看新的通用控件类的内部。

“旧式”Windows 控件类

基本的控件类包括：

CBitmap。

CBitmapButton。

CComboBox。

CEdit。

CListBox。

CDragListBox。

CCheckListBox。

CScrollBar。

CStatic。

这些类的名字都能很好地解释类的功能。使用这些 MFC 基本控件类的方法就是从对话框中已有的控件创建它：

```
CButton * pButton = (CButton *)pDialog->GetDlgItem(IDC_CHECK1);
ASSERT(pButton != NULL);
pButton->SetCheck(m_bShowState);
```

另一个使用这些类的办法是定义一个显示变量，然后调用它的 Create()成员函数。例如，下面的代码就在 CView 中创建了一个按钮：

```
CRect rectButton(10,10,50,50);
CButton * pButton = new CButton;
pButton->Create("Button text", WS_CHILD|BS_PUSHBUTTON,
rectButton, pView, ID_BUTTON);
// Later that same day...
pButton->DestroyWindow();
delete pButton;
```

在控件已经创建之后，对于每个控件你有很多方法可以调用。关于详细内容请查看 MFC 文档。

基本控件类内幕

MFC 控件类都是 CWnd 的派生类，它们的声明在文件 AFXWIN.H 中，实现在如下几个文件中：

- AFXWIN2.INL——所有控件类的内嵌函数。
- WINCTRL1.CPP——CStatic、CButton、CListBox、CComboBox、CEdit 和 CScrollBar。
- WINCTRL2.CPP——CDragListBox。
- WINCTRL3.CPP——CCheckListBox。
- WINBTN.CPP——CBitmapButton。

最基本的 6 个控件的实现已经很完美。当调用 Create()成员函数进行创建工作时，MFC 会自动将这个控件粘贴到相应的 Windows 对象上。其中，所有的成员函数会映射到每个控件所支持的消息中。例如，当你通过 CListBox::AddString()在 CListBox 上添加一个字符串时，AFXWIN2.INL 中如下代码就会被调用：

```
ASSERT(::IsWindow(m_hWnd));
return (int)::SendMessage(m_hWnd, LB_ADDSTRING, 0, LPARAM)lpszItem);
```

实际上，如果你打开 WINCTRL1.CPP，看看这些类的//Implementation 部分，你就会发现里面什么都没有。

不要失望：如果我们再看进去一些，我们就会发现一些有趣的内幕。CCheckListBox 的实现很有趣，而且它是一个演示如何定制一个已有控件类的好例子。

在我们看 CCheckListBox 的声明之前，先做一个小简介：一个复选列表框就是复选框的链表。如果有选项链表（就像你在安装程序中常见到的），这些控件都是方便的。使用 CCheckListBox 和使用 CListBox 差不多，只是增加了一些新的成员函数来处理复选框状态。例如，SetCheck()成员函数用于对一个复选框设置“选中”，GetCheck()成员函数用来获得复选框是否选中的信息。

你还可以指定复选列表框中所有按钮的风格：BS_CHECKBOX、BS_AUTOCHECKBOX、BS_AUTO3STATE 和 BS_3STATE。

探索 CCheckListBox

程序清单 6-29 包含了 CCheckListBox 声明的一个缩减版本，在文件 AFXWIN.H 中。

程序清单 6-29 CCheckListBox 声明的缩减版本（在文件 AFXWIN.H 中）

```
class CCheckListBox : public CListBox
{
// Constructors **omitted
// Attributes **omitted
// Overrideables **omitted
// Implementation
protected:
    void PreDrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
    void PreMeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct);
    int PreCompareItem(LPCOMPAREITEMSTRUCT lpCompareItemStruct);
    void PreDeleteItem(LPDELETEITEMSTRUCT lpDeleteItemStruct);
    virtual BOOL OnChildNotify(UINT, WPARAM, LPARAM, LRESULT*);
    int CalcMinimumItemHeight();
    void InvalidateCheck(int nIndex);
    void InvalidateItem(int nIndex);
    int m_cyText;
    UINT m_nStyle;
// Message map functions **some omitted
```

```
protected:
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg LRESULT OnLBSetItemData(WPARAM wParam, LPARAM lParam);
    afx_msg LRESULT OnLBSetItemHeight(WPARAM wParam, LPARAM lParam);
    DECLARE_MESSAGE_MAP()
};
```

CCheckList 使用标准列表框类的自绘（owner-draw）特性实现了复选框链表。前 5 个成员函数（PreXXX）都负责应答由处于“自绘”模式的列表框发送的消息。OnChildNotify()负责将“自绘”消息映射成成员函数。

下面是一些实用函数。CalcMinimumItemHeight()使用字体矩阵来指出链表中每个项的最小大小。InvalidateCheck()和 InvalidateItem()负责使一个项的选择区域或整个项失效。

CCheckListBox 有两个成员数据。m_cyText 存储了复选链表中正文的高度，m_nStyle 保存了复选列表框中按钮的风格。

CCheckListBox 还有消息映射表，利用这个消息映射表对标准的 CListBox 消息进行分类。例如，OnLButtonDown()需要指出用户是否正要选中某个复选框，并且要更新按钮的状态。通常 OnLButtonDown 只是在列表框中选择一个项。

下面我们直接看 CCheckListBox 的实现，如程序清单 6-30 所示，先看的是 CCheckListBox::SetCheck 成员函数，看一下关于是否选中的信息存储在哪里了。这个实现在文件 WINCTRL3.CPP 中。

程序清单 6-30 CCheckListBox::SetCheck()实现的缩减版本（在文件 WINCTRL3.CPP 中）

```
void CCheckListBox::SetCheck(int nIndex, int nCheck)
{
    LRESULT lResult = DefWindowProc(LB_GETITEMDATA, nIndex, 0);
    AFX_CHECK_DATA* pState = (AFX_CHECK_DATA*)lResult;
    if (pState == NULL)
        pState = new AFX_CHECK_DATA;
    pState->m_nCheck = nCheck;
    VERIFY(DefWindowProc(LB_SETITEMDATA, nIndex, (LPARAM)pState) != LB_ERR);
    InvalidateCheck(nIndex);
}
```

首先，SetCheck()调用了 DefWindowProc()，参数是 LB_GETITEMDATA，用来查看是否已经为 item 创建了一个 AFX_CHECK_DATA 指针。如果没有，SetCheck()就会继续执行，并且为它分配一个。

然后，SetCheck()修改了 AFX_CHECK_DATA 状态指针的 m_nCheck 域，并调用 DefWindowProc(LB_ SETITEMDATA)将新指针重新和 item 联系。

最后，SetCheck()调用 InvalidateCheck()。InvalidateCheck()会使选中失效，这样就会产生 WM_DRAWITEM 调用。PreDrawItem()成员函数重画这个条目，以反映出新的 nCheck 状态。

AFX_CHECK_DATA 看起来使保存了每个条目的状态。下面是它的声明，在文件 WINCTRL3.CPP 中：

```
struct AFX_CHECK_DATA
{
public:
    int m_nCheck;
    BOOL m_bEnabled;
    DWORD m_dwUserData;
};
```

所以，对于复选链表中的每个条目（每个条目都是复选框），CCheckList 都会维护三个值：状态（m_nCheck）、是否被启用（m_bEnabled）和一个 DWORD（m_dwUserData），这个 DWORD 是为了让类的使用者能将一些数据和这个 item 联系在一起而增加的。

CCheckList::PreDrawItem()成员函数是整个类的关键。它完成了 CCheckList 的所有绘画工作。不幸的是，这个函数太长了，所以不能把它全列出来，即使是只列出伪代码。下面是它的功能的纲要（如果你打开 WINCTRL3.CPP，像读这些描述那样读代码，你会从中获得更多的东西）：

1. 获得一个指向 static _AFX_CHECKLIST_STATE 的指针，名字为 pChecklistState。

2. 根据提供的区域，确定是否需要重画整个复选框（选中和正文）或只画一部分（选中或正文）。

3. 如果选中需要重画，它就创建一个兼容的 DC，并选中 pChecklistState 指针指向的位图，然后用它对选中做 BitBlt 操作。

4. 现在 PreDrawItem 会调用 DrawItem。

CCheckListBox::DrawItem()会在选择了相应的字体和其他 DC 对象之后，调用 ExtTextOut()来画复选框条目的正文。

这个 CListBox 派生类使用的一个有趣的技术是 AFX_CHECKLIST_STATE。下面让我们在继续之前仔细看一下这个结构。

_AFX_CHECKLIST_STATE 结构的声明在文件 WINCTRL3.CPP 中：

```
class _AFX_CHECKLIST_STATE : public CNoTrackObject
{
public:
    _AFX_CHECKLIST_STATE();
    virtual ~_AFX_CHECKLIST_STATE();
    HBITMAP m_hbitmapCheck;
    CSize m_sizeCheck;
};
```

它从 CNoTrackObject 派生，表示这个类不希望被内存诊断跟踪。因为它是一个静态对象，所以直到程序退出时才会销毁，所以 MFC 可能会误认为这是内存泄漏。

你会发现它有一个 m_hbitmapCheck 位图句柄数据成员。在 PreDrawItem()中曾使用到了这个成员和 m_sizeCheck（保存位图的大小）。下面是_AFX_CHECKLIST_STATE 的构造函数的摘要，反映了它如何得到这个句柄和位图的大小。

```
CBitmap bitmap;
if (afxData.bWin4 || AfxGetCtl3dState()->m_pfnSubclassDlgEx != NULL)
    VERIFY(bitmap.LoadBitmap(AFX_IDB_CHECKLISTBOX_95));
else
```

```
    VERIFY(bitmap.LoadBitmap(AFX_IDB_CHECKLISTBOX_NT));
BITMAP bm;
bitmap.GetObject(sizeof (BITMAP), &bm);
m_sizeCheck.cx = bm.bmWidth / 3;
m_sizeCheck.cy = bm.bmHeight;
m_hbitmapCheck = (HBITMAP)bitmap.Detach();
```

_AFX_CHECKLIST_STATE 构造函数根据它是否运行在 Windows 95 下而装载不同的位图。一旦它从资源装载了位图，它会调用 GetObject 来确定宽度和高度，然后存储在 m_sizeCheck 中。最后，它将 CBitmap::Detach()返回的位图句柄赋值给 m_hbitmapCheck。

这些位图是 MFC\INCLUDE\RES\NTCHECK.BMP 和 MFC\INCLUDE\RES\95CHECK.BMP。图 6-5 和 6-6 显示了两个位图，这两个位图用来显示不同的 check 类型。每一段都表示一个状态。你会在三状态（AUTO_3STATE）复选框中看到最后一个状态。PreDrawItem 根据状态选择正确的位图。我们看一下 PreDrawItem()的程序清单，看它如何选择位图。

图 6-5 选中位图的 NT 版本

图 6-6 选中位图的 Windows 95 版本

“新式”的 MFC Windows 通用控件类

和基本控件类一样，大多数通用控件类都是从 CWnd 派生的。下面是这些通用控件类的链表和一些简短描述。如果类不是从 CWnd 派生的，则会注明它的基类。

- CAnimateCtrl——一个显示动画的控件。
- CDragListBox——CListBox 的派生类，你可以在列表框中拖放条目。
- CHeaderCtrl——和一个 CListCtrl 一起使用，用来显示柱状信息。
- CHotKeyCtrl——为从用户那里获得键序列（Alt-Backspace-Delete）提供了一个接口。
- CImageList——CObject 的派生类，它保存了一个图像的集合。
- CListCtrl——显示链表条目的图像链表（Explorer 风格的）。
- CProgressCtrl——显示一个进度条。
- CRichEditCtrl——一个丰富的编辑控件，它理解一些基本的 RTF 格式，允许设置多种字体和颜色。
- CSliderCtrl——在某个范围内选择值的滚动条。
- CSpinButtonCtrl——一个微调控制器（spinner）。
- CStatusBarCtrl——一个状态栏。
- CTabCtrl——属性页控件。
- CToolBarCtrl——实现了一个工具栏。
- CToolTipCtrl——提供了一个小工具条。
- CTreeCtrl——一个 Explorer 风格的树状控件。

使用通用控件类

使用 MFC 通用控件类和使用基本控件类一样。你可以使用资源编辑器在对话框中放置

控件，也可以将控件看作是一个子窗口，用::Create 创建它，用 DestroyWindow 销毁它。

主要的区别就是通用控件类有一些新的成员函数可以用。看一下 CTreeCtrl，它有 30~40 个成员函数，你可以利用这些函数控制树状控件。

MFC 通用控件类内幕

通用控件类的声明在头文件 AFXCMN.H 中。它的实现在下面几个文件里：

- AFXCMN.INL——通用控件类的很多内嵌成员在这里。
- WINCTRL2.CPP——所有的通用控件类，除了 CRichEditCtrl。
- WINCTRL4.CPP——CRichEditCtrl。

通用控件类只是相应的控件上一个简单的包装层。和基本的控件类一样，它们也是在创建的时候绑定到对话框上，而且创建后每个成员函数如果要设置信息或提取信息，都会发送消息给控件。

例如，CTreeCtrl::InsertItem()成员函数在文件 AFXCMN.INL 中，扩展如下：

```
HTREEITEM CTreeCtrl::InsertItem(LPTV_INSERTSTRUCT lpInsertStruct)
{
    ASSERT(::IsWindow(m_hWnd));
    return (HTREEITEM)::SendMessage(m_hWnd, TVM_INSERTITEM, 0,
        (LPARAM)lpInsertStruct);
}
```

所以除了增加了一些断言以及处理一些特殊参数，公用对话框类实在没有什么特殊的内幕。

但是快速地浏览 AFXCMN.H、AFXCMN.INL、WINCTRL2.CPP 和 WINCTRL4.CPP 还是值得的，这样你可以对 MFC 如何封装控件有所了解。说不定哪天，你需要从控件发送自己的消息或者获取特定的通知，这时了解 MFC 如何完成同样工作就很有帮助了。

结　语

在下一章我们会继续沿着 MFC 树向上爬，探索文档/视图结构的内幕。因为文档/视图结构是完全在 MFC 内部实现的，所以有很多成熟的内幕可以发掘。

CHAPTER 7

MFC 的文档/视图结构

作为一个应用程序框架，MFC 内容丰富，功能强大。除了为 Windows SDK 提供一个方便使用的包装，MFC 还提供了其他一些特性，使得开发人员要做的事情相当简单。这些特性包括易于实现的工具栏和状态栏、逻辑命令和消息路由选择、预建立的集合类（collection）以及定义了赋值和连接操作的字符串类。MFC 的最重要的特性之一就是它能将管理应用程序数据的代码和产生这些数据的代码分离开来。这个特殊的性质（分离应用程序的数据管理代码和用户界面代码）由 MFC 的文档/视图结构（Document/View architecture）提供。

文档/视图结构是 MFC 的基石之一，理解它是有效使用 MFC 的一个关键。例如，MFC 中 OLE 复合文档的实现强烈地依赖于它的文档/视图结构。你要使用 MFC 实现复合文档，就要对文档/视图结构有深刻的理解。

为什么要用文档/视图

几乎所有的软件开发都要以某种形式处理数据管理。在整个计算机工业的历史中，寻找管理应用程序的数据的方法一直是一个大问题。毕竟，信息和数据管理是计算机技术的主要用途之一。

分离数据与用户界面显示代码涉及许多方面，包括：（1）决定应用程序的哪个部分拥有数据；（2）确定应用程序的哪个部分负责更新数据；（3）确定如何显示数据的多个图示；（4）协调数据更新；（5）构想某种存储数据的方法；（6）管理用户界面。最后这一条可能很有技巧性（特别是当涉及多种文档类型的时候），因为这经常要求更新工具栏和菜单。

如果你的背景是 C/SDK，可能对管理数据的困难有深刻的了解。由于 SDK 程序的结构，所以决定把数据管理代码放在哪里总是很伤脑筋。

其他原因

分离数据和 GUI 代码当然是一个重要的目的，不过 MFC 的文档/视图结构还有其他的好处。一个很重要的收获是内建的打印和打印预览支持。有谁能够抵御第一次使用文档/视图结构写 MFC 应用程序时就可以感受到的强大功能：只需要几个小时的时间，你的应用程序就能具有不逊色于大部分商业化的 Windows 应用程序的打印和打印预览支持。

当你书写更高级的应用程序时，可能会发现需要支持 OLE 特性，例如复合文档和本地编辑等。MFC 对这些 OLE 特性的支持很大程度上依赖于文档/视图结构。

旧的方法

你是否还记得当年不得不使用 C 和 SDK 书写应用程序的经历（甚或是使用 MFC 版本 1，那时 Microsoft 还没有增加所有的高级抽象）？你必须想出一种表示数据的方式。然后你不得不找到办法在屏幕上产生数据。如果存在不止一种方式表示数据，你还肩负着协调应用程序的数据及其数据的不同显示的任务。这是一个艰难的过程，可以实现数据管理的方式的数量几乎和程序员的数量一样多。对于 C/SDK 程序员，有多种方法可供选择来解决这些问题。

如果你的应用程序只使用一种数据（例如，一个简单的记事本类型编辑器只需要编辑文本行），这个问题不是那么严重。要解决这样一个不复杂的任务，一个全局数据定义就足够了。对于一个简单的文本编辑器，数据可以就是一个表示文本的字符串数组。

但是如果你想写一个使用多种类型的数据的应用程序怎么办？一个管理多种不同类型数据库的应用程序就是一个这样的例子。可以想像，每种类型的文档都需要自己的一套菜单资源和工具栏。除了管理数据库的代码，你还要自己写代码管理每种不同的文档类型及其资源的用户界面。

下一步，设想你要创建一个应用程序，以多种不同的方式产生数据。可能应用程序要编辑一个电子表格或数据库，把其中的数据表示成表格以及各种类型的图。更进一步地设想，你可以从任意一种数据表现形式修改数据，而且对其中一种显示所做的修改会引起其他显示的更新。现在，你需要改进你的软件，在管理数据显示的同时协调用户界面交互和管理应用程序数据。

你可以很会地发现问题是如何积累起来的。幸运的是，现在有一种通用的方法可以解决上面的所有问题：它就叫文档/视图结构。

任何数据管理机制背后的基本思想是将数据的管理工作分割成两个逻辑部分：（1）数据管理；（2）用户界面管理。这就是文档/视图结构做的事情。

体系结构

支持应用程序数据和数据显示的协调是一个框架能完成的完美工作。当微软在 1993 年初发布 MFC 2.0 时，就在 MFC 中包含了文档/视图结构。MFC 的文档/视图结构处理了如上所述的数据管理和用户界面问题。

从本质而言，MFC 文档/视图结构并不是一个新概念。它第一次由 Xerox PARC（图形用户界面也是由这家公司发明的）创建，是 Smalltalk 环境的一个关键部分。Smalltalk 版本的文档/视图结构叫做模型-视图控制器（model-view-controller，MVC），其中的模型等于现在所说的文档。Smalltalk 的 MVC 体系结构有一个控制器（类似于 MFC 中的 CDocTemplate），作为文档和视图之间的屏障，使它们不会过分互相依赖。

文档和视图

MFC 文档/视图结构为协调应用程序数据及其显示提供了一种一致的方法。在 MFC 中，文档处理数据管理，而应用程序的视图处理用户界面的管理。实际操作中，应用程序的数据集中放在一个地方，用户界面代码单独封装。

文档/视图结构的主要优点是，它鼓励分离应用程序数据（文档）和数据的表示（视图）。术语“文档”（document）可能有一点误导，因为看起来它似乎是指以某种类似于文字处理器和电子表格形式出现的数据。也许称之为“数据集”或者“数据源”会更好一些。文档其实就是一个保存数据的地方，视图则是显示数据的地方。

这个分离十分重要——特别是如果你采用了不止一种方法显示你的数据。如果你写代码的时候把输出数据的代码和管理数据的代码混在一起，会费很大的功夫。当然你也可以自己写代码达到分隔的效果，但是文档/视图结构提供给你一个单一的、一致的方法。这个框架为文档和视图的管理提供了基本的底层架构，包括文件管理、在屏幕上显示、管理资源等功能。你可以定制（或者忽略）不适合你的需要的部分。

如果没有这个实现分离的框架，你不得不处理很多问题。例如，如果你有多种方法来看数据，就必须想办法保证当前的表示（视图）正确地访问到了当前的数据。如果用户可以从一种或多种表示中编辑数据，你必须确保所有的表示都更新了。MFC 的文档/视图结构已经完成了所有这些工作！

文档/视图组件

MFC 文档/视图结构有 4 个主要的组件：（1）文档；（2）视图；（3）文档/视图框架；（4）文档模板。下面几个小节分别解释每个组件。

文档

MFC 文档/视图结构的核心就是文档，你可以把它理解成任何一种数据源。

例如，如果你在开发一个应用程序来管理实时数据集合，没有任何理由说文档不能是实时数据提供器本身。天然地，文档管理可能需要涉及读或写某种永久存储设备（例如硬盘），但这并不是绝对必需的。

在 MFC 中，文档是由 CDocument 类体现的。当书写你自己的 MFC 应用程序时，你可以从 CDocument 派生你的文档类。实际上，当使用 AppWizard 生成一个应用程序时，AppWizard 就会创建一个派生于 CDocument 的类。CDocument 有各种各样的功能，用于完成管理 I/O、更新数据显示等操作。

CDocument 类自身其实没什么用。当你在你的 MFC 应用程序中实现一个文档时，通常都需要往你的 CDocument 类里添加数据成员，以表示文档中的数据。例如，如果你写一个电子表格类型的应用程序，你可能要创建一个文档类处理一个数组，数组的每个元素是代表一个电子表格单元的结构。如果你在开发一个画图应用程序，你可能会把你的数据表示成一个数组或链表，元素是描述屏幕上的不同图形的结构。当然，文档的一个经典例子可以在微软自己提供的 MFC 引导例子中找到，那就是 Scribble 程序。Scribble 程序将数据表示成屏幕上点的位置（坐标）的数组。

MFC 的 CDocument 类在 MFC 层次结构中的深度是两层。它派生自 CCmdTarget，后者又派生自 CObject。因为它派生自 CObject，所以 CObject 类提供的所有支持，即运行时类型信息（run-time type information）、动态创建（dynamic creation）和序列化（serialization），对 CDocument 都是可见的。此外，CDocument 类还派生自 CCmdTarget 类，所以它可以接收命令消息（即 WM_COMMAND 消息）。最后，CDocument 类可以支持组件对象模型（COM）接口和 OLE 自动化，这也是从 CcmdTarget 类继承来的。这一点十分重要，因为你有时候可能希望数据源自身也支持用户界面。例如，设想你在写一个应用程序，从某个科学仪器收集实时数据。因为 CDocument 类派生自 CCmdTarget，所以你可以向仪器（即数据源，或称文档）发送诸如“启动仪器”、“关闭仪器”之类的命令。

除了发送程序式的命令，另一个好处是它允许把用户界面直接和文档中的功能绑定。换而言之，你可以在文档自身中实现一个命令处理器。根据命令路由的定义，命令在传播时先经过文档，再传播到用户界面元素，例如工具栏和菜单，这些东西实际上属于完全不同的对象。这个事实可能被滥用，导致对文档的不安全的访问；但如果使用正确，可以形成非常有用的技术。

视图

只有一个文档的应用程序一点都不好玩。我们还需要在屏幕上显示数据并且提供操作数据的用户界面。这个功能是由 MFC 的 CView 类提供的。

在一个 MFC 的文档/视图结构中，视图负责显示文档的数据。CView 派生自 CWnd，后者又派生于 CCmdTarget，再上一层的父类是 CObject（听起来感觉像是圣经，是不是？）。因为 CView 派生于 CCmdTarget，所以它能够接收来自菜单和控件的消息（这一点很不错，因为视图一般都要同时负责处理用户界面）。因为 CView 派生于 CWnd，所以它也能接收窗口消息（诸如 WM_PAINT 和 WM_SIZE）。

在一个标准的 C/SDK 程序中，一般通过处理 WM_PAINT 消息显示数据。通过调用 BeginPaint()创建一个设备环境（context）的句柄（一个 HDC）。一旦获得了设备环境，就可以调用很多函数在窗口中画图了。

标准的 MFC 应用程序使用文档/视图结构在视图中画图。CView 中包含一个 OnDraw()函数，当视图需要更新的时候框架就调用这个函数。

MFC 视图其实就是一个无边界的窗口，用来提供数据显示。但是，当你使用一个基于 MFC 的应用程序时，图像好像出现在一个带有边界和菜单的窗口中。这个窗口就叫做框架窗口，它是文档/视图结构正常工作的必要条件。

从所有实际效用来看，CView 和子 CWnd 差不多，只是多了几个额外的函数。所以，如果你的 CView 派生类写得正确，它可以用在某些特异的环境中（例如，对话框中的一个子窗口）。一般来说，CView 中没有什么东西要求父窗口一定是 CFrameWnd，虽然当父窗口不是框架时，有些特性的确会被禁用。

文档/视图框架

迄今为止，我们已经描述了文档/视图结构将数据管理和显示分离在两个不同地方的办法——文档和视图。不过，这个机制还有另一个重要的方面：为每个独立的视图提供一个不同的用户界面（即一套不同的菜单和控件）的能力。

假设现在编写一个电子表格应用程序，只有一个文档（就是电子表格自身），但是要以两种不同方式表示数据。也许这个应用程序将数据显示在一个网格（第一个视图）和一张图表（第二个视图）中。这种情况下，你可能希望每个视图有不同的菜单。也许你希望在网格视图中包含诸如排序和查询数据的菜单选项；而在图表中，可能希望提供诸如修改图的颜色和格式之类的菜单选项。幸运的是，每个视图的框架用来处理这个用户界面代码。

第一次接触的时候，你可能会奇怪为什么菜单管理要由一个独立的框架来处理，而不是视图本身。这是一个好的设计模式，实际经验告诉我们，最好把这类功能分离出来，而不是紧密地耦合。这样可以减少依赖，使得我们能更容易地一次处理一个单独的代码段。这个技术也是区分 SDI、MDI 以及 OLE 定点编辑间的差别的基础。将 CView 和框架分离使得 CView 更加灵活，你可以在其他情况下使用它们。

图 7-1 显示了框架和视图之间的关系。

图 7-1 一个框架窗口中的视图

SDI（Single Document Interface，单文档接口）和 MDI（Multiple Document Interface，多文档接口）应用程序处理它们的文档、视图和框架时有细微的不同。当然，SDI 应用程序一个时刻只需要关心一个文档。微软的画笔和写字板（Windows 中带的两个应用程序）是这类程序的典型例子。你一次只可以看一个文档。当你在一个文档已经打开的情况下试图打开一个新文档时，程序会在打开第二个文档之前先关闭第一个文档。使用 MDI 应用程序可以同时打开多个文档，例如微软的 Excel 和 Novell 的 Quattro Pro。

SDI 应用程序使用一个派生于 CFrameWnd 的类作为包含视图的框架。MDI 应用程序从 CMDIChildWnd 派生出一个类作为包含视图的框架。

文档模板

文档/视图结构的一个有趣的地方是前述 3 个组件在操作时作为一个单元处理。也就是说，文档、显示和用户界面的概念作为一个整体一起处理。文档模板（Document Template）将整个机制捆绑在一起。这 3 个组件由一个称为 CDocTemplate 的类管理。

文档模板：你必须使用它吗？

你可以不依赖于 CDocTemplate 或其任何一个派生类就使用框架/视图/文档。CDocTemplate 实际上并没有实现任何特别的功能，只是对文档/视图框架的一些常用功能进行一下包装。特殊的应用程序常常需要特殊的文档模板类，或完全不同的什么东西。另外，MFC 中很多命令的实现都依赖于文档模板链表，例如文件|新建和文件|打开。不过你也可以自己实现这些命令的一个版本。

你的应用程序对所支持的每种类型的文档都应该有一个文档模板。例如，如果某个应用程序同时支持电子表格和数据库，那么这个应用程序就有两个文档模板对象。每个文档模板负责创建和管理该类型的所有文档。

CDocTemplate 是一个抽象基类，它定义了处理文档、框架和视图的基本操作。GetFirstDocPosition()、GetNextDocPosition()、OpenDocumentFile()是纯虚函数，这使得 CDocTemplate 是一个抽象基类。MFC 定义了其他两种文档模板类，可以在应用程序中直接使用。它们是 CSingleDocTemplate 和 CMultiDocTemplate。

CSingleDocTemplate

对于只使用一种文档的应用程序，MFC 定义了 CSingleDocTemplate 类。CSingleDocTemplate 的构造函数有 4 个参数：（1）一个资源 ID，用于标识所有绑定到该文档模板的各种资源；（2）运行时的 CDocument 派生类；（3）运行时的视图框架；（4）运行时的文档视图。

资源 ID 也标识了应用程序字符串表的一个条目。它指定一个字符串，其中该字符串包含了关于该文档的 7 个方面的信息：（1）窗口标题；（2）文档名称；（3）当创建一个新文档时使用的文档类型描述符；（4）对标准的打开文件对话框中文件类型的描述；（5）文件扩展名过滤器；（6）文件管理器使用的文件类型描述符；（7）在 Windows 注册表中注册的 ProgID。

CMultiDocTemplate

对于那些使用多于一种文档的应用程序，MFC 定义了一个名为 CMultiDocTemplate 的类。CMultiDocTemplate 和 CSingleDocTemplate 十分相似。它的构造函数具有相同的参数。主要的不同之处在于 CMultiDocTemplate 可以支持一组文档的链表，而 CSingleDocTemplate 只能支持一种文档。

CWinApp 的角色

MFC 定义了两个类来管理文档/视图/框架的组合：CMultiDocTemplate 和 CSingleDocTemplate。但是，谁管理文档模板呢？在一个 MFC 应用程序中，CWinApp 对象管理文档模板。

CWinApp::InitInstance()是通常用来创建文档模板的地方。AppWizard 一开始就会为你创建好一个文档模板，同时还带有一个文档、一个框架和一个视图。文档模板构造好以后，InitInstance()紧接着将这个文档模板添加到 CWinApp 的文档模板链表中去，使用的函数是 CWinApp::AddDocTemplate()。

CWinApp 如何使用文档模板呢？在标准的 MFC 应用程序中，应用程序对象（派生于 CWinApp）处理文件|新建和文件|打开命令。你可能会认为框架直接创建文档。实际上，创建新文件和打开已有文件的工作是由文档模板处理的。

CWinApp 包含一些函数，相应于文件处理的菜单项。它们是 OnFileNew()和 OnFileOpen()。你可以在 CWinApp 类中找到这些函数。要使它们工作，只需要加些水——添加一些消息映射项即可。当然，如果使用 AppWizard 创建你的应用程序，则 AppWizard 会为你添加消息映射。一旦有了 OnFileNew()和 OnFileOpen()的消息映射，CWinApp 就会处理剩下的事情。这些实现可能不适合你的应用程序的要求。如果这样，记住它们只不过是普通的消息处理函数，可以像其他命令处理函数一样被重载或替换。

现在我们已经对 MFC 的文档/视图框架有了一个总体的认识，下面来看看它内部是如何工作的。

文档/视图结构内幕

在本节中，我们将深入文档/视图结构的类（使用 CWinApp 作为跳板），并展示它们是如何结合在一起工作的。然后，我们将涉及一些高级的文档/视图问题，例如打印和打印预览，以及 CView 派生类内幕。在很多文档/视图结构类之间有千丝万缕的联系，理解这些细节能够使你有效地使用文档/视图。

CWinApp 管理文档模板，文档模板管理框架/视图/文档。理解文档/视图结构的第一步是学习 CWinApp 和 CDocTemplate 之间的接口。

CWinApp/CDocTemplate 接口：CDocManager

因为 MFC 应用程序可以注册多于一种的文档模板，所以框架必须维护一个文档模板的链表。这带来一个很重要的问题：这个链表放在哪里？把链表放在 CWinApp 里看起来很符合逻辑，但是 CWinApp 的声明（在 AFXWIN.H 中）却没有文档模板链表。

不过，在这个声明中，有一个数据成员体现了 CWinApp 和 CDocTemplate 之间的关联。头文件中与之有关的部分是：

```
CDocManager * m_pDocManager;
```

CDocManager 目前（在 MFC 4.0 中）是一个没有文档说明的类，所以让我们来看看它的声明，弄清楚它是做什么的。程序清单 7-1 包含了 CDocManager 的声明。

程序清单 7-1　没有文档说明的 CDocManager 声明（在文件 AFXWIN.H 中）

```
class CDocManager : public CObject
{
    DECLARE_DYNAMIC(CDocManager) public:
// Constructor
    CDocManager();
//Document functions
    virtual void AddDocTemplate(CDocTemplate* pTemplate);
    virtual POSITION GetFirstDocTemplatePosition() const;
    virtual CDocTemplate* GetNextDocTemplate(POSITION& pos) const;
    virtual void RegisterShellFileTypes(BOOL bWin95);
    virtual CDocument* OpenDocumentFile(LPCTSTR
                                        lpszFileName); // open named file
    virtual BOOL SaveAllModified(); // save before exit
    virtual void CloseAllDocuments(BOOL bEndSession); // close documents
before exit
    virtual int GetOpenDocumentCount();
// helper for standard commdlg dialogs
    virtual BOOL DoPromptFileName(CString& fileName, UINT nIDSTitle,
                                  DWORD lFlags, BOOL bOpenFileDialog,
                                  CDocTemplate* pTemplate);
//Commands ** Some omitted.
    virtual void OnFileNew();
```

```
    virtual void OnFileOpen();
// Implementation
protected:
    CPtrList m_templateList;
public:
    virtual ~CDocManager();
};
```

在 CDocManager 中，如果你直接跳到//Implementation 部分，可以发现一个名为 m_templateList 的 CPtrList（一个 CObject 指针链表）。这就是丢失的文档模板链表！CDocManager 中的大部分成员函数都用于管理这个链表，并向 CWinApp 提供一个 CDocTemplate 的接口。

版本注释

MFC 4.0 之前的版本中，都是在 CWinApp 中维护文档模板链表的。微软看起来好像正在将它抽象到 CDocManager 中，虽然我们不是很清楚他们为什么要这样做。也许他们在为新的 MDI 选择做铺垫。谁知道呢——也许在 CWinApp 中管理这个链表太累赘了，微软希望在 4.0 中简化 CWinApp。

MFC 内部人士是这样说的：

我们将这个实现放到一个单独的类里，主要原因是：这样，不使用文档/视图（例如，只需要实现简单功能）的应用程序不必负担额外的开销。以前，把所有这些功能放在 CWinApp 中是可以的，因为它相当简单，不会影响应用程序的大小，即使应用程序不怎么需要使用它。渐渐地，我们增加了一些新的特性，它的实现越来越庞大。在 MFC 4.0 中，我们把这部分功能分离成一个新的类，带有所有的虚函数。如果一个 CDocManager 对象永远不被创建，它的虚表永远不被访问，结果是它的函数不会被链接进来。你还可以在其他几个情形下（希望某个特性及有关代码是可选的）看到这个技术，例如 CDocManager 和 CRecentFileList。当然，如果你使用该 DLL，那么所有的代码都会在那里，这个技巧并没有起多大作用。如果你选择静态链接，则结果应用程序的大小就会有差别了（我们注意示例程序 HelloApp 在做这些优化后的大小）。而且，还有一个边缘效果，程序员可以用自己的版本代替 CDocManager 的实现。将来，我们希望减少 CWinApp 中的东西（现在它已经超出控制范围了），这是向该目标前进的一个例子。

因为 CDocManager 的大部分功能以前都在 CWinApp 中，所以很多 CWinApp 的成员函数（特别是其中处理文档模板的那些）现在都通过 m_pDocManager 调用。

例如，这里是 CWinApp::AddDocTemplate()的实现，来自 APPUI2.CPP：

```
void CWinApp::AddDocTemplate(CDocTemplate* pTemplate)
{
    if (m_pDocManager == NULL)
        m_pDocManager = new CDocManager;
    m_pDocManager->AddDocTemplate(pTemplate);
}
```

下面是一些 CWinApp 的成员函数链表，这些函数都可以通过 CDocManager 指针 CWinApp:: m_pDocManager 直接调用。

- CWinApp::AddDocTemplate()。
- CWinApp::CloseAllDocuments()。
- CWinApp::DoPromptFileName()。
- CWinApp::GetFirstDocTemplatePosition()。
- CWinApp::GetNextDocTemplate()。

- CWinApp::GetOpenDocumentCount()。
- CWinApp::OnFileNew()。
- CWinApp::OnFileOpen()。
- CWinApp::OpenDocumentFile()。
- CWinApp::RegisterShellFileTypes()。
- CWinApp::SaveAllModified()。

以上这些成员函数大部分都有很好的文档，所以现在我们直接跳到 CDocManager 的数据成员。不必担心，当我们学习如何创建文档/视图组件的时候，还会用到其中一些 CDocManager 成员函数。

- m_templateList——如前所述，这个指针维护一个文档模板链表。其中很有意思的一点是这个数据成员是 protected 的。要访问它，你的对象或者是友元，或者是 CDocManager 的派生类。不幸的是，这使得你几乎不可能直接在你的 MFC 应用程序中访问这个指针。你可以很容易地创建一个 CDocManager 派生类，但是困难的是如何用你自己的 CDocManager 派生类代替 CWinApp 中的 m_pDocManager 指针。

所幸，GetFirstDocTemplatePosition()和 GetNextDocTemplate()这两个 CWinApp 成员函数使得你可以遍历这个模板链表。没有什么情形（至少我们想不出来）必须直接访问 m_pDocManager，因为你可以通过成员函数访问它。

CDocManager::OnFileNew()内幕

CDocManager 的一个关键成员函数是 OnFileNew()。OnFileNew()负责创建一个新的文档模板（接着就是文档/模板中涉及的所有其他对象）。让我们看看 CDocManager::OnFileNew()的伪代码，见程序清单 7-2。

程序清单 7-2 CDocManager::OnFileNew()伪代码（在文件 DOCMGR.CPP 中）

```
void CDocManager::OnFileNew()
{
    CDocTemplate* pTemplate = (CDocTemplate*)m_templateList.GetHead();
    if (m_templateList.GetCount() > 1) {
        CNewTypeDlg dlg(&m_templateList);
        int nID = dlg.DoModal();
        if (nID == IDOK)
            pTemplate = dlg.m_pSelectedTemplate;
        else
            return;     // none - cancel operation
    }
    pTemplate->OpenDocumentFile(NULL);
}
```

首先，OnFileNew()检查 m_templateList 中是否有多于一个文档模板。如果是，则 OnFileNew()创建一个 CNewTypeDlg 对话框，列出一个文档链表以供用户选择。

一旦用户选择了适当的文档模板，则 OnFileNew()调用 CDocTemplate::OpenDocumentFile()。我们学习 CDocTemplate 时会再来看这个 CDocTemplate::OpenDocumentFile()。

注意在 OnFileNew()中，CNewTypeDlg 的构造函数读入一个指向文档模板指针链表的指针，即 m_templateList。并注意在 DoModal()之后，OnFileNew() 从 CNewTypeDlg::m_

pSelectedTemplate 中得到选中的对话框模板。

CNewTypeDlg 是一个有趣的例子，快速而且简便地向终端用户提供一个选项（这个例子中是文档类型）链表。让我们稍微离一点题，看看这个对话框时怎么实现的。

CDocManager：没有文档说明的辅助类 CNewTypeDlg

图 7-2 显示了一个工作中的 CNewTypeDlg。CNewTypeDlg 的声明位于 DOCMGR.CPP 中，如程序清单 7-3 所示。

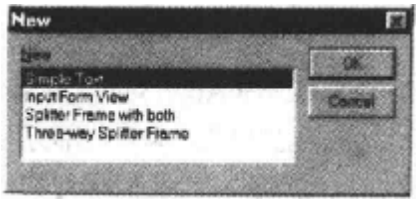


图 7-2 CNewTypeDlg

程序清单 7-3 CNewTypeDlg 声明（来自 DOCMGR.CPP）

```
class CNewTypeDlg : public CDialog
{
protected:
    CPtrList*   m_pList;
public:
    CDocTemplate*   m_pSelectedTemplate;
    enum { IDD = AFX_IDD_NEWTYPEDLG };
    CNewTypeDlg(CPtrList* pList) : CDialog(CNewTypeDlg::IDD){
       m_pList = pList;
       m_pSelectedTemplate = NULL;
    }
    ~CNewTypeDlg() { }
protected:
    BOOL OnInitDialog();
    void OnOK();
    DECLARE_MESSAGE_MAP()
};
```

从 CNewTypeDlg 的声明中，我们可以看到 CNewTypeDlg 是 CDialog 的一个派生类。它有两个数据成员：m_pList 和 m_pSelectedTemplate。这个对话框有一个资源 ID，是 AFX_IDD_ NEWTYPEDLG。这个对话框资源在 INCLUDE\AFXRES.RC 中定义，它的布局见图 7-3。

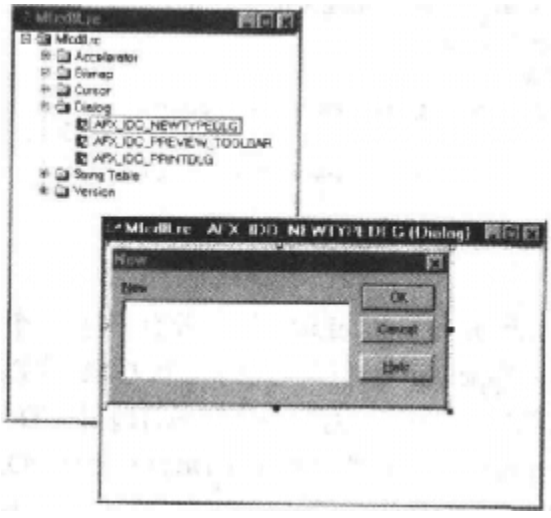


图 7-3 新的对话框资源

进一步看这个内嵌构造函数，我们可以看到 m_pList 指向的是传入构造函数的文档模板链表。

现在让我们看看 CNewTypeDlg 如何显示文档模板链表并返回一个选择。CNewTypeDlg::OnInitDialog()的伪代码在程序清单 7-4 中列出。

程序清单 7-4 CNewTypeDlg::OnInitDialog()伪代码（在文件 DOCMGR.CPP 中）

```
BOOL CNewTypeDlg::OnInitDialog()
{
    CListBox* pListBox = (CListBox*)GetDlgItem(AFX_IDC_LISTBOX);
    POSITION pos = m_pList->GetHeadPosition();
    while (pos != NULL) {
        CDocTemplate* pTemplate = (CDocTemplate*)m_pList->GetNext(pos);
        CString strTypeName;
        if (pTemplate->GetDocString(strTypeName,
            CDocTemplate::fileNewName) &&
            !strTypeName.IsEmpty()) {
            // add it to the listbox
            int nIndex = pListBox->AddString(strTypeName);
            pListBox->SetItemDataPtr(nIndex, pTemplate);
        } // end if
    } // end while
    return CDialog::OnInitDialog();
}
```

首先，OnInitDialog()调用 GetDlgItem()得到一个指向 CListBox 的指针，并将之类型转换成 CListBox 指针。

得到 ClistBox 指针后，OnInitDialog()遍历它的文档模板链表。在遍历的过程中，OnInitDialog()调用 CDocTemplate::GetDocString()得到文档类型的字符串版本。OnInitDialog()获取每个文档的这个字符串，并将之加到链表中，然后调用 CListBox::SetItemDataPtr()将这个文档模板指针附加到这个项上。

OnInitDialog()一旦完成将文档模板加入链表的工作，就返回到 CDialog::OnInitDialog()。

在 CNewType::OnOK()中，当用户从链表中选择了一项并按下 OK 按钮时，CNewType::OnOK()从链表中得到被选中的项，并调用 CListBox::SetItemDataPtr()获得指向该文档模板的一个指针。OnOK()将这个被选中的模板指针放在 pSelectedTemplate 中，这样 DoModal()返回之后就可以获取这个指针。

现在，如果你发现自己再遇到类似的情况，就知道一种快速而简单的方法，可以在模态对话框中显示一些选择了。

到了这里，我们已经穷尽了有趣的 CDocManager 的内幕，所以让我们沿着文档/视图的命令链，到下一站——文档模板。

CDocTemplate: CDocument、CView 和 CFrameWnd 管理器

如前所述，CDocTemplate 行使将文档/视图/框架三元体绑定在一起的职能。CDocTemplate 的声明可以在 AFXWIN.H 中找到，实现在 DOCTEMPL.CPP 中。记住，CDocTemplate 是一个抽象基类，你的 MFC 应用程序实际上使用的是 CDocTemplate 的两个派生类之一：CSingleDocTemplate 或 CMultiDocTemplate。CSingleDocTemplate 在 DOCSINGL.CPP 中，

CMultiDocTemplate 在 DOCMULTI.CPP 中。

让我们先观察 AFXWIN.H 中的 CDocTemplate 的声明（见程序清单 7-5），看看它是如何工作的。注意：CDocTemplate 的声明非常长，为了简洁，我们已经去掉了很多 OLE 和其他成员。

程序清单 7-5 缩减后的 CDocTemplate 声明

```
class CDocTemplate : public CCmdTarget
{
    DECLARE_DYNAMIC(CDocTemplate)
    // Constructors ** omitted.
    // Attributes ** omitted
    // Operations ** omitted
    // Overridables ** omitted
    // Implementation ** some omitted
protected:
    UINT m_nIDResource;                 // IDR_ for frame/menu/accel as well
    CRuntimeClass* m_pDocClass;         // class for creating new documents
    CRuntimeClass* m_pFrameClass;       // class for creating new frames
    CRuntimeClass* m_pViewClass;        // class for creating new views
    CString m_strDocStrings;            // '\n' separated names
};
```

缩减后的 CDocTemplate 声明突出了没有文档说明的 CDocTemplate 数据成员。每个数据成员的简要描述和用途如下：

- m_nIDResource —— 存储传给 CDocTemplate 构造函数的第一个参数，其实是一个资源 ID，指向绑定到该文档模板的各种资源。
- m_pDocClass——一个指针，指向该文档模板的 CDocument（或其派生类）的 CRuntimeClass 结构。在构造函数中设定。
- m_pFrameClass——一个指针，指向该文档模板的 CFrameWnd（或其派生类）的 CRuntimeClass 结构。在构造函数中设定。
- m_pViewClass——一个指针，指向该文档模板的 CView（或其派生类）的 CRuntimeClass 结构。在构造函数中设定。
- m_strDocStrings——一个 CString，包含用来定义与每个文档模板联系的 7 个字符串的字符串表资源（参考前面的介绍）。在 LoadTemplate()中初始化。

当我们查看 CDocTemplate 的关键成员函数时，会更详细地看到这些数据成员是怎么使用的。

CDocTemplate 如何创建新的文档/视图/框架？

这是一个很重要的问题，答案在两个成员函数中：CDocTemplate:: CreateNewDocument()和 CDocTemplate::CreateNewFrame()。

正如它们的名字所示，CreateNewDocument()创建一个文档，而 CreateNewFrame()创建一个框架。但是还有一个视图没有处理。在什么地方创建视图呢？我们现在先把这个问题搁一搁。

先让我们看看 CreateNewDocument()，然后再看 CreateNewFrame()。程序清单 7-6 列出了 CreateNewDocument()的伪代码。

程序清单 7-6 CDocTemplate:: CreateNewDocument()的伪代码（来自 DOCTEMPL.CPP）

```
CDocument* CDocTemplate::CreateNewDocument()
{
```

```
    CDocument* pDocument = (CDocument*)m_pDocClass->CreateObject();
    if (pDocument == NULL)    {
        TRACE1("Warning: Dynamic create of document type %hs failed.\n",
        m_pDocClass->m_lpszClassName);
        return NULL;
    }
    AddDocument(pDocument);
    return pDocument;
}
```

首先，CreateNewDocument()通过 CRuntimeClass 指针的数据成员 m_pDocClass 调用 CRuntimeClass::CreateObject()创建一个新的文档。如果你忘了 MFC 动态创建的工作方式，可以参看第 5 章。MFC 文档/视图体系结构在很多地方用到动态创建。

在 pDocument 中保存了新的 CDocument（或其派生类）指针之后，CreateNewDocument()验证 CreateObject()已经工作了。如果 CreateObject()调用失败，CreateNewDocument()会产生一些调试输出，并且返回 NULL，表示失败。如果 CreateObject()调用成功，CreateNewDocument()通过调用 AddDocument()将该文档加到打开文档链表中。最后，CreateNewDocument()返回指向新文档的指针。

好了，现在很清楚 CreateNewDocument()没有创建视图。现在我们继续来看 CreateNewFrame()，看看能不能找到什么线索。程序清单 7-7 包含了 CreateNewFrame()的伪代码。

程序清单 7-7　CDocTemplate:: CreateNewFrame()的伪代码（来自 DOCTEMPL.CPP）

```
CFrameWnd* CDocTemplate::CreateNewFrame(CDocument* pDoc, CFrameWnd*
pOther)
{
    CCreateContext context;
    context.m_pCurrentFrame = pOther;
    context.m_pCurrentDoc = pDoc;
    context.m_pNewViewClass = m_pViewClass;
    context.m_pNewDocTemplate = this;
    CFrameWnd* pFrame = (CFrameWnd*)m_pFrameClass->CreateObject();
    // create new frame from resource
    if (!pFrame->LoadFrame(m_nIDResource,
    WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE, // default frame styles
    NULL, &context)){
        TRACE0("Warning: CDocTemplate couldn't create a frame.\n");
        return NULL;
    }
    return pFrame;
}
```

首先，CreateNewFrame()填充一个 CCreateContext（细节见下一小节），其信息与新创建的框架有关。pOther 只在 MDI 中使用，与现在讨论的东西关系不大。

下一步，CreateNewFrame()通过 m_pFrameClass 中的 CRuntimeClass 结构调用 CRuntimeClass::CreateObject()创建一个框架。创建了框架之后，CreateNewFrame()调用 LoadFrame()装入这个框架的资源，并根据这些值创建该框架。最后，CreateNewFrame()返回一个指向新框架的指针。

在继续探求如何创建视图之前，我们先偏离主题去看看 CCreateContext。

CCreateContext: 创建的辅助类

CCreateContext 是一个辅助结构，框架用它来存放创建各种文档/视图结构元素所需的关键信息。程序清单 7-8 给出了 CCreateContext 的声明。

程序清单 7-8 CCreateContext 声明（在文件 AFXEXT.H 中）

```
struct CCreateContext   // Creation information helper
{
    CRuntimeClass* m_pNewViewClass;
    CDocument* m_pCurrentDoc;
    CDocTemplate* m_pNewDocTemplate;
    CView* m_pLastView;
    CFrameWnd* m_pCurrentFrame;
// Implementation
    CCreateContext();
};
```

以下是 CCreateContext 的各个域存放信息的总结：

- m_pNewViewClass —— 一个 CRuntimeClass 指针，用来创建视图；
- m_pCurrentDoc —— 指向当前文档对象的指针；
- m_pCurrentFrame —— 指向当前框架对象的指针；
- m_pNewDocTemplate —— 如果有多个文档，指向最后一个文档；
- m_pLastView —— 如果有多个视图，指向最后一个视图；

这里当然有一些有用的信息，但是都已经在文档模板里有了，为什么还要费事把这些都放到 CCreateContext 结构里呢？这是因为 MFC 需要在文档模板、框架、视图和文档之间传递这些信息。MFC 并不将之处理成一个全局变量，而是存放在这样一个结构中并利用指向在 CDocTemplate::CreateNewFrame()中创建的实例的指针传递这些信息。在讲到各种文档/结构的专题时，你会看到 CCreateContext 被一次又一次地提到。有时候 MFC 还要在一个消息的 LPARAM 中传递一个指向 CCreateContext 结构的指针，所以当你看到类似下面的文字时不要大惊小怪：

```
int CView::OnCreate(LPCREATESTRUCT lpcs)
{
    //...
    CCreateContext* pContext = (CCreateContext*)lpcs->lpCreateParams;
    //...
}
```

CView 的创建？

CCreateContext::m_pNewViewClass 给了我们第一个关于视图创建的线索。这个线索最终会把我们引向 CFrameWnd 类，但是现在我们还有一些 CDocTemplate 的细节需要学习。我们将稍后在 CFrameWnd 内幕中继续讨论 CView 创建的问题。

CSingleDocTemplate 和 CMultiDocTemplate

在我们讨论了 CreateNewDocument()和 CreateNewFrame()之后，你可能会问，打开文档的链表放在哪里呢？绝对不是在 CDocTemplate 的声明中。文档链表对于两种 CDocTemplate 的派生类（CSingleDocTemplate 和 CMultiDocTemplate）不太一样。

在 CSingleDocTemplate 的声明中，可以找到这样一个 CDocument 指针类型的数据成员：

```
CDocument* m_pOnlyDoc;
```

（一个听起来有点孤独的数据成员，不是吗？）

另一方面，CMultiDocTemplate 必须维护多个打开的文档，所以它的声明中有这个：

```
CPtrList m_docList;          // open documents of this type
```

这是一个 CObject 派生类，或者说得更准确一点，一些打开的 CDocument。

在 CSingleDocTemplate 中，AddDocument()将 m_pOnlyDoc 设置成参数；而 CMultiDocTemplate:: AddDocument()通过调用 m_docList.AddTail(pDocument)将新的文档加入自己的链表中。

除了 m_docList，CMultiDocTemplate 还有一个数据成员：

```
int m_nUntitledCount;
```

这个数据成员用来记录没有标题的窗口数。当用户创建多个未命名的文档时，MFC 会自动将之命名为 Untitled[X]，其中 X 是未命名文档数。

CMultiDocTemplate::OpenDocumentFile()

下面我们来看看 CMultiDocTemplate::OpenDocumentFile()，理解 CDocTemplate 是如何打开文档的。CMultiDocTemplate 与 CSingleDocTemplate 相比有一些更有趣的性质，所以我们决定分析 CMultiDocTemplate::OpenDocumentFile()。程序清单 7-9 包含了 OpenDocumentFile()的伪代码，来自 DOCMULTI.CPP。

程序清单 7-9　CMultiDocTemplate::OpenDocumentFile()伪代码（来自 DOCMULTI.CPP）

```
CDocument* CMultiDocTemplate::OpenDocumentFile(LPCTSTR lpszPathName,
BOOL bMakeVisible)
{
    // Hey reader-> Remember these from earlier?!
    CDocument* pDocument = CreateNewDocument();
    CFrameWnd* pFrame    = CreateNewFrame(pDocument, NULL);
    if (lpszPathName == NULL) { // create a new document - with default
                                // document name
        SetDefaultTitle(pDocument);
        if (!pDocument->OnNewDocument()) {
            pFrame->DestroyWindow();
            return NULL;
        }

        m_nUntitledCount++;
    }
    else {    // open an existing document
        CWaitCursor wait;
        if (!pDocument->OnOpenDocument(lpszPathName)){
            pFrame->DestroyWindow();
            return NULL;
        }
        pDocument->SetPathName(lpszPathName);
    }
    InitialUpdateFrame(pFrame, pDocument, bMakeVisible); return pDocument;
}
```

OpenDocumentFile()调用CreateNewDocument()创建一个新的空文档(一个CDocument 派生对象)。下一步，一个新的框架通过调用 CreateNewFrame()被创建，它的 lpszPathName 参数决定下一步做什么。

如果 lpszPathName 为 NULL，则 OpenDocumentFile()知道要创建的是一个空的新文档。它调用 SetDefaultTitle()完成这个功能。该函数为新文档确定名称[使用 Untitled（或其他指定名称）与 m_nUntitledCount 中未命名计数的组合]，然后通过新文档指针调用 CDocument::SetTitle()为该文档设置名字。最后，OpenDocumentFile()调用 OnOpenDocumentFile()，并将未命名计数的数据成员加 1。

如果 lpszPathName 不等于 NULL，则 OpenDocumentFile()尝试打开指定的文档。首先，OpenDocumentFile()显示一个等待光标，然后调用 CDocument:: OnOpenDocumentFile()，将文件名传入作为参数。如果CDocument::OnOpenDocumentFile()调用失败，则OpenDocumentFile()将销毁新创建的框架，并返回一个 NULL 表示失败。如果 CDocument::OnOpenDocumentFile()成功，则 OpenDocumentFile()调用 CDocument::SetPathName()更新 CDocument 的路径。

不管 lpszPathName 的状态是什么，OpenDocumentFile()最后都要调用 CDocTemplate::InitializeUpdateFrame()，后者又调用 CFrameWnd::InitialUpdateFrame()，并传递一个指向新文档的指针。

虽然我们在第 2 章中已经说了很多关于 CFrameWnd 的东西，我们还是来看看这个类中与文档/视图结构紧密相关的部分，从而理解它是如何嵌入到大框架里的。

CFrameWnd 内幕

有好几个地方 CFrameWnd 会与 CDocument 和 CView 类交互（至少从 CFrameWnd 的角度来看如此）：

1. CView 的创建。（终于等到这一天了……）我们现在还不能揭示出它如何工作。

2. CFrameWnd 有一个 CView 指针对象成员 m_pViewActive，CFrameWnd 用它来与视图交互，激活或使它失活。这个成员变量可以通过 CFrameWnd::GetActiveView()获取，通过 CFrameWnd::SetActiveView()设置。

3. 成员函数 InitialUpdateFrame()使所有活动视图的 CView::OnInitialUpdate()成员函数被调用。（第 9 章中讲到的分割窗口允许在一个框架中有多个视图同时被激活）

CFrameWnd 和 CView 的创建

现在终于到了你一直等待着的时刻：CView 在什么时候、如何创建？在此之前，我们最后一次说到这个问题时，已经知道了 CDocTemplate::CreateNewDocument()创建文档，CDocTemplate::CreateNewFrame()创建框架窗口。此外，CreateNewFrame()将视图的 CRuntimeClass 信息放在 CCreateContext::m_pNewViewClass 中，并将这个结构传给 CFrameWnd::LoadFrame()。

在 WINFRM.CPP 中，只有一个成员函数使用了 CCreateContext::m_pNewViewClass，那就是 CFrameWnd::CreateView()。

下面我们就来看看这个函数。

视图的创建：CFrameWnd::CreateView()

程序清单 7-10 给出了 CFrameWnd::CreateView()的伪代码。

程序清单 7-10 CFrameWnd::CreateView()的伪代码（来自 WINFRM.CPP）

```
CWnd* CFrameWnd::CreateView(CCreateContext* pContext, UINT nID)
{
    CWnd* pView = (CWnd*)pContext->m_pNewViewClass->CreateObject();
    if (pView == NULL)
        return NULL;
    if (!pView->Create(NULL, NULL, AFX_WS_DEFAULT_VIEW,
         CRect(0,0,0,0), this, nID, pContext))
        return NULL;        // can't continue without a view
    return pView;
}
```

第一步，CreateView()使用 CCreateContext::m_pNewViewClass 中存储的 CRuntimeClass 创建一个视图。创建之后，它调用 CWnd::Create()完成 MFC 对象创建的第二阶段工作。

在所有准备工作之后，真正的创建非常简单。但是……

等一等，还有更多！

现在有一个很严重的问题：谁调用 CreateView()？我们说过，CreateNewDocument()和 CreateNewFrame() 是被 C[Single/Multi]DocTemplate::OpenDocumentFile() 调用的。但是 OpenDocumentFile()不调用 CreateView()。

为了找到答案，我们沿着调用堆栈往上走。搜索 WINFRM.CPP，其中惟一调用了 CreateView() 的函数是 OnCreateView()。OnCreateView() 被 OnCreateHelper() 调用。OnCreateHelper()只被 OnCreate()调用。

CFrameWnd::OnCreate()是一个消息映射表条目，当框架收到 WM_CREATE 时被调用。所以当文档模板在 CreateNewFrame()中创建框架时，Windows 发出一个 WM_CREATE，并进行如下调用：

```
OnCreate() -> OnCreateHelper() -> OnCreateClient() -> CreateView().
```

（实际上分割窗口也调用 CreateView()，但你得等到第 9 章才能知道为什么。）

现在你已经知道视图是如何创建的，接着该继续 CFrameWnd 的讨论，看它是如何嵌入在视图/文档结构中工作的。我们已经在本节的引言部分介绍了 m_pActiveView 的作用，现在剩下 CFrameWnd::InitialUpdateFrame()。

CFrameWnd::InitialUpdateFrame()

回忆一下，InitialUpdateFrame()使得框架中的所有视图都被更新。让我们来看看它是如何实现这个功能的。程序清单 7-11 包含 CFrameWnd::InitialUpdateFrame()的伪代码。

程序清单 7-11 CFrameWnd::InitialUpdateFrame()伪代码（来自 WINFRM.CPP）

```
void CFrameWnd::InitialUpdateFrame(CDocument* pDoc, BOOL bMakeVisible)
{
    CView* pView = // ** omitted it determines the active view or gets
    // first splitter window - See Chapter 7.
    if (bMakeVisible) {
        SendMessageToDescendants(WM_INITIALUPDATE, 0, 0, TRUE, TRUE);
        ActivateFrame();
        pView->OnActivateView(TRUE, pView, pView);
    }
    // update frame counts and frame title (may already have been visible)
```

```
    if (pDoc != NULL)
        pDoc->UpdateFrameCounts();
    OnUpdateFrameTitle(TRUE);
}
```

首先，InitialUpdateFrame()获得一个活跃视图的指针，并将之存放在 pView 中。下一步，如果 bMakeVisible 参数为 TRUE，则 InitialUpdateFrame()发出一个 MFC 特有的消息（WM_INITIALUPDATE）给它的所有后代（这包括了所有视图）。现在，你只要知道这个动作将使得该框架窗口内的所有视图获得 WM_INITIALUPDATE 消息。MFC 将这个消息映射到 CView::OnInitialUpdate()。发送了 WM_INITIALUPDATE 消息之后，InitialUpdateFrame()激活框架和视图。视图的激活通过调用 CView::OnActiveView()实现。

下一步，InitialUpdateFrame()更新文档中的框架计数（以后我们还会更多地讲到它），最后 InitialUpdateFrame()更新框架的名字，以反映文档的名字。

现在，等式中只剩下 CDocument 和 CView 两个变量了。我们先看 CDocument，最后以 CView 内核结束。

CDocument 内幕

CDocument 中有大量的没有文档说明的数据成员、成员函数和有趣的性质。我们首先看 AFXWIN.H 中的 CDocument 的声明，见程序清单 7-12。CDocument 的程序清单在 DOCCORE.CPP 中。

程序清单 7-12 简化的 CDocument 声明（在文件 AFXWIN.H 中）

```
class CDocument : public CCmdTarget
{
    DECLARE_DYNAMIC(CDocument)
// Constructors **omitted
// Attributes **omitted
// Operations **omitted
// Overridables **omitted
// Implementation **Some ommitted
protected:
    CString m_strTitle;
    CString m_strPathName;
    CDocTemplate* m_pDocTemplate;
    CPtrList m_viewList;                    // list of views
    BOOL m_bModified;                       // changed since last saved
public:
    BOOL m_bAutoDelete;     // TRUE => delete document when no more views
    virtual ~CDocument();
// implementation helpers
    virtual BOOL DoSave(LPCTSTR lpszPathName, BOOL bReplace = TRUE);
    virtual BOOL DoFileSave();
    virtual void UpdateFrameCounts();
    void DisconnectViews();
    void SendInitialUpdate();
    friend class CDocTemplate;
// Message handlers **omitted
    DECLARE_MESSAGE_MAP()
};
```

哇！在 CDocument 中出现了前所未有多的没有文档说明的成员。我们不可能对每一个成员都详细阐述，以下是一个简要的链表：

- m_strTitle——文档的名字。用 SetTitle()设置。保存时用它来设定框架窗口的标题和文件的名字。
- m_strPathName——当前文档的路径（包括文件名）。用 SetPathname()设置。这个成员是对 MRU 菜单的补充，在保存文件时会用到。
- m_pDocTemplate——指向文档的文档模板的指针（见后面的说明）。在 CDocTemplate::AddDocument()中设置。可通过 CDocTemplate::GetDocTemplate()访问该成员。
- m_viewList——每个 MFC 文档都可以拥有无限多个视图。m_viewList 记录在该文档下打开的所有视图的指针。当文档改变时，CDocument 通过 m_viewList 通知视图，从而使之更新以反映文档的变化。视图通过 AddView()被加入到 m_viewList 中，通过 RemoveView()被删除。要遍历 m_viewList，可以用 GetFirstViewPosition()获得第一个视图，然后不断调用 GetNextView()。文档也用 m_viewList 作为访问框架窗口的一种手段。请看下面的 UpdateFrameCount()的描述。
- m_bModified——“修改”标志位，当文档被用户修改就设为 1。如果这个标志位为 1，框架就会在用户关闭文档时提示。可以用 IsModified()查看状态，用 SetModifiedFlag()设置。
- m_bAutoDelete——该标志位可以被框架设置成 TRUE，表示当它的所有视图关闭时该文档对象要被删除。开发人员可以将它设置成 FALSE，这样即使在该文档上没有任何打开的视图，该文档对象仍会保留。
- DisconnectViews()——该成员函数遍历 m_viewList 中保存的视图链表，并将 CView::m_pDocument 指针设置为 NULL，从而将所有视图从该文档上断开。在 CDocument 的析构函数中会调用 DisconnectViews()。
- DoSave()——这个成员函数包含保存文档的“核心”代码。CDocument 成员 OnFileSaveAs()和 OnFileSave()在提示输入文件名或验证文件名之后，最终都调用 DoSave()。DoSave()中调用 OnSaveDocument()。
- DoFileSave()——CDocument::OnFileSave()调用这个成员函数。DoFileSave()先检查文档要被存储到的文件的一些性质，然后调用 DoSave()。
- UpdateFrameCounts()——这是一个非常有趣的 CDocument 成员，但是内容太多，不可能在这里介绍它的细节。强烈建议读者花一点时间读一读 DOCCORE.CPP 中它的源码。UpdateFrameCounts()计数为该文档的所有视图打开的框架，并且根据新的计数通知它们更新它们的窗口标题。我们说 UpdateFrameCounts()很有趣，是因为它展示了文档是如何遍历所有视图和框架的。

  另外，UpdateFrameCounts()还展示了另一个有趣的算法，叫做“标记并清除（mark and sweep）”。它遍历两遍框架窗口。第一遍标记所有的框架，第二遍根据第一遍获得的计数更新所有标记过的框架。

- SendInitialUpdate()——遍历视图链表并调用 CView::OnInitialUpdate()。

“嘿，我死了！”

文档/视图类中一个有趣的问题是它们如何互相标记。例如，如果用户关闭了某个文档，文档模板如何知道这个文档现在不存在了？对此，MFC 文档/视图体系采取的一个常用解决方案是维护一个指向某个类的指针，当文档“死”的时候必须通知它（像一个讣告）。在我们这个例子中，CDocument 维护一个指向 CDocTemplate 的指针（成员变量 m_pDocTemplate）。这个指针是创建者设置的，被创建对象有义务在死的时候通知创建它的人。在 CDocument/CDocTemplate 例子中，CDocTemplate 在它的 AddDocument()成员函数中设置 CDocument::m_pDocTemplate。在 CDocument 析构函数中，CDocument 检查 m_pDocTemplate，如果不为 NULL，就调用 m_pDocTemplate ->RemoveDocument(this)，通知 CDocTemplate 自己被销毁了。

文档和视图采用相同的技术。每个 CView 有一个指针指向和它相联系的 CDocument，并用这个指针通知 CDocument 这个 CView 正在被销毁。

现在我们已经向你介绍了大部分之前没有文档说明的 CDocument 实现细节，现在将关注 CDocument 的 3 个关键方面：

1. 创建文档（包括创建新文档和从文件中打开）。
2. 对文档排序。
3. 与视图交互。

创建文档

文档或者作为空文档创建，或者从一个文件打开。如果创建一个空的文档，会调用 CDocument::OnNewDocument()。这个成员函数首先调用 DeleteContents()清空文档。DeleteContents()的默认实现什么都不做，如果你想在新文档创建的时候执行特殊的清空工作，要在你自己的 CDocument 派生类中重载 DeleteContents()。调用了 DeleteContents()之后，OnNewDocument()设置修改标志位为 FALSE，并确保 m_strPathName 为空。

CDocument::OnOpenDocument()从一个文件创建文档。在 CDocTemplate 内核一节中，我们说过 C[Multi/Single]DocTemplate::OpenDocumentFile()调用 CDocument:: OnOpenDocument()。OnOpenDocument()是一个关键的成员函数，所以我们要看一看它的伪代码，了解它做了些什么。程序清单 7-13 包含了 CDocument::OnOpenDocument()的伪代码。

程序清单 7-13 CDocument::OnOpenDocument()伪代码（在文件 DOCCORE.CPP 中）

```
BOOL CDocument::OnOpenDocument(LPCTSTR lpszPathName)
{
    CFileException fe;
    CFile* pFile = GetFile(lpszPathName,
            CFile::modeRead|CFile::shareDenyWrite, &fe);
// **omitted, checks on pFile, assume it worked..
    DeleteContents();
    SetModifiedFlag();  // dirty during de-serialize
    CArchive loadArchive(pFile, CArchive::load | CArchive::bNoFlushOnDelete);
    loadArchive.m_pDocument = this;
    loadArchive.m_bForceFlat = FALSE;
    TRY    {
        CWaitCursor wait;
```

```
    CFile* pFile = NULL;
    pFile = GetFile(lpszPathName, CFile::modeCreate |
                CFile::modeReadWrite | CFile::shareExclusive, &fe);
// **omitted, checks on pFile, assume it's cool.
    CArchive saveArchive(pFile, CArchive::store |
        CArchive::bNoFlushOnDelete);
    saveArchive.m_pDocument = this;
    saveArchive.m_bForceFlat = FALSE;
    TRY {
        CWaitCursor wait;
        Serialize(saveArchive);     // save me
        saveArchive.Close();
        ReleaseFile(pFile, FALSE);
    }
    CATCH_ALL(e) {
        //**omitted, handling of archive exception.
    }
    END_CATCH_ALL
    SetModifiedFlag(FALSE);     // back to unmodified
    return TRUE;        // success
}
```

OnSaveDocument()和 OnOpenDocument()两个函数的相似性不言自明：它们是对方的完美映像！

就像在 OnOpenDocument()中做的那样，OnSaveDocument()通过 GetFile() 打开一个文件，但使用标志指定文件是打开用来保存的，而不是用来读的（CFile::mode ReadWrite|CFile::modeCreate）。接着，OnSaveDocument()创建 CArchive，并将之传递给 Serialize()成员函数。

最后，如果一切都顺利地序列化了，则 CArchive 被关闭，标志位设置成 FALSE，并且 OpenDocument()返回 TRUE，表示文档已经成功地被保存了。

现在让我们看看 CDocument 内核的第三个有趣的方面：它如何与视图通信。

与视图通信

在 CDocument 声明概述中已经提到，m_viewList 是 CDocument 与正在“视（viewing）这个文档的视图进行交互的中介。

实际上只有很少的情况文档需要与它的视图交互，包括这样一些：

1. 通知视图文档被销毁。CDocument 析构函数调用 DisconnectView()。DisconnectView()遍历视图的链表，将 CView::m_pDocument 设置成 NULL。

2. 从视图出发，通过调用 CView::GetParentFrame()访问框架，例如 UpdateFrameCounts()、CanCloseFrame()以及 OnCloseDocument()都这样做。

3. 通知视图文档被修改、需要更新，UpdateAllViews()完成这一功能。

为了对 CDocument 和 CView 交互的工作过程有一个感性认识，我们来看看 CDocument::UpdateAllViews()的伪代码，见程序清单 7-15。

程序清单 7-15 CDocument::UpdateAllViews()伪代码（在文件 DOCCORE.CPP 中）

```
void CDocument::UpdateAllViews(CView* pSender, LPARAM lHint, CObject*
pHint)
{
```

```
    POSITION pos = GetFirstViewPosition();
    while (pos != NULL) {
        CView* pView = GetNextView(pos);
        if (pView != pSender)
            pView->OnUpdate(pSender, lHint, pHint);
    }
}
```

调用 UpdateAllViews()是 CDocument 或者 CView 的责任。如果它在视图中被调用，则第一个参数（pSender）一般就指向调用它的视图。

首先，UpdateAllViews()通过调用 GetFirstViewPosition()获得 m_viewList 的第一个视图；然后，UpdateAllViews()遍历整个视图链表，如果视图不是那个发送者，就调用 CView::OnUpdate()。它不调用发送者的 CView::OnUpdate()，因为发送者自己负责更新自己。

我们要从 UpdateAllViews()学习的要点是，文档不得不与它的视图链表进行通信，而这正是它的典型做法。

现在，在 MFC 内核范围，我们只有一个类没有看了，那就是 CView。

CView 内幕

CView 类是视图类体系的根，有多个派生类。这一节中，我们只关注 CView。到第 8 章，我们会介绍 CView 的一些高级特性，例如打印和打印预览，还有 CView 的派生类。

CView 的声明位于 AFXWIN.H 中。因为 CView 的文档十分详尽，所以这里没有必要列出它的声明。只有一个没有文档说明的数据成员 m_pDocument。m_pDocument 是一个 CDocument 指针，与 CDocument::m_pDocTemplate 类似。说它们相似，是因为与 CDocument::m_pDocTemplate 一样，m_pDocument 被 CView 用来通知文档自己被销毁。这个 m_pDocument 指针有时候也用于消息路由选择。可以通过成员函数 CView::GetDocument()获得 m_pDocument 指针。

在 CFrameWnd 内核一节中，我们已经看到了视图是如何创建的。为了了解视图如何工作，我们现在来看框架如何画并更新 CView。

CView 绘图

在文档/视图介绍中，曾说过必须提供 CView 的一个派生类，实现 OnDraw()成员函数以完成所有的画图功能。让我们看看 OnDraw()在何时以及如何被调用，从而使得视图重画。

Windows 中画图的常规机制是处理 WM_PAINT——或者在 MFC 中 WM_PAINT 的消息映射表条目 OnPaint()。下面我们来看 CView::OnPaint()消息处理函数，了解它是如何处理重画的。程序清单 7-16 包含了 CView::OnPaint()的伪代码。

程序清单 7-16 CView::OnPaint()伪代码（在文件 VIEWCORE.CPP 中）

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

CView::OnPaint()首先创建一个 CPaintDC，其中包含了要重画的区域。下一步，OnPaint()将这个 paint DC 传递给 OnPrepareDC()，最后将 paint DC 传给 OnDraw()。

CView 中默认的 OnPrepareDC()和 OnDraw()实现实际上不做任何事情，你的 CView 派生类要负责提供这个行为。

我们将 CView 和 CView 派生类的其他有趣的内幕推迟到第 8 章介绍。现在，趁着读者对 CDocTemplate、CDocManager、CFrameWnd、CDocument 和 CView 内核还记忆犹新，我们回顾一下到现在为止学了些什么。

文档/视图内幕再览

哇！这一章里我们已经覆盖了很多类的内核：CDocTemplate、CDocManager、CFrameWnd、CDocument、CView。为了帮助你理解和消化这一切，现在我们快速地回顾一遍。

首先，我们将再次强调文档/视图体系中各个类之间的相互依赖关系。然后，我们回顾所有类的创建。最后，我们以在框架中创建一个新文档（通过 CWinApp::OnFileNew()）这个例子为主线，把所有内容串在一起。

文档/视图相互依赖关系

以下是前面已经介绍过的文档/视图体系中的一些相互依赖关系：

- CWinApp 包含一个 CDocManager 指针。
- CDocManager 维护一个文档模板链表。该应用程序支持的每一类文档都有一个文档模板相对应。开发者应该负责在 CWinApp::InitInstance()中创建并加入这些模板。
- 文档模板将这 3 个类绑定在一起：CView/CDocument/CFrameWnd。这些类都根据传递给文档模板构造函数的 CRuntimeClass 信息创建。
- MDI 专有的文档模板维护一个打开文档链表，而 SDI 专有的文档模板只有一个指向打开文档的指针。
- 文档维护一个打开视图的链表。它们使用这个链表与视图通信、更新视图。
- 文档有一个指针指向它们的文档模板。它们用这个指针设置标题，进行命令路由选择，并在被删除的时候通知文档模板。
- 框架有一个指针指向当前的活跃视图。
- 在被创建的时候，框架得到一个 CCreateContext 结构，其中包含在文档模板中可以找到的 CRuntimeClass 信息和一个指向文档的指针。
- 视图有一个指针指向它的文档。当该视图被删除的时候（RemoveView()），用该指针通知文档。命令路由选择也要使用它。
- 视图可以通过调用 CView::GetParentFrame()获得它的框架窗口。
- 如果文档要访问它的所有视图的框架，可以遍历它的视图链表，并调用 CView::GetParentFrame()。

如果你在文档/视图体系中工作，那么了解这些组件如何协同工作是非常重要的。你可能需要反复阅读这个表，以确保已经百分之百地理解了。

创建信息

这里是各个类在何处创建的索引：

- CDocTemplate——在 CWinApp 中创建，保存在 CWinApp::m_pDocManager 中。
- CDocManager——在 CWinApp::InitInstance()中创建。
- CFrameWnd——在 CDocTemplate::CreateNewFrame()中创建。
- CDocument——在 CDocTemplate::CreateNewDocument()中创建。
- CView——通过 CFrameWnd::OnCreate()创建。

组合到一起

从这样低的一个层次观察 MFC 的文档/视图体系结构，就好比在一根消防水管下喝水。现在给你歇一口气的机会，我们将一起经历从框架开始创建一个新文档的过程。

因为创建一个空文档和打开一个文档十分相似，所以我们将这两个情况一起讨论。注意我们讨论的出发点是默认的消息映射表和成员函数。如果你覆盖了这些默认行为，下面的讨论可能不适用于你的应用程序。

开始

文档的创建过程开始于你的 CWinApp 派生类。如果你用旧版的应用程序生成向导产生了你的应用程序，那么可以在你的 CWinApp 派生类的实现（文件名一般是应用程序名字加上.CPP 后缀）中看到这样两行代码：

```
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
```

这两个消息映射表条目使得当用户在应用程序菜单中选择“新建”时，CWinApp::OnFileNew()将被调用；当选择“打开”时 CWinApp::OnFileOpen()被调用。

这两个成员函数都会通过 CWinApp 中的 CDocManager 指针 m_pDocManager 调用相应的函数。

“新建”之后

新建和打开操作从这一点开始分道扬镳。我们先看新建的情形，待会儿再说打开。

CDocManager::OnFileNew()检查它的模板链表（该链表在你的 CWinApp 派生类的 OnInitInstance()成员函数中初始化）。如果有多于一个文档模板可供选择，则 CDocManager::OnFileNew()生成一个模态对话框 CNewTypeDlg，让用户选择要创建的新文档的类型。一旦用户选择了文档类型，CDocManager::OnFileNew()调用 CDocTemplate::OpenDocumentFile(NULL)，其中用到用户选择文档类型返回的指针。记住，这个指针在 CDocManager 文档模板链表中——m_templateList。

“打开”之后

如果是“打开”，CDocManager::OnFileOpen()创建一个普通对话框，要求用户输入要打开的文件的名字。当用户选择一个文件之后，CDocManager::OnFileOpen()调用 CWinApp::OpenDocumentFile(file name)。CWinApp::OpenDocumentFile()又回头来调用 CDocManager::OpenDocumentFile()。

CDocManager::OpenDocumentFile()根据用户在 CDocManager::OnFileOpen()中输入的文件扩展名选择正确的文档模板。找到正确的文档模板指针之后，CDocManager::OnFileOpen()调用 CDocTemplate::OpenDocumentFile(file name)。

这个时候，打开和新建操作又回到了同一条路径，只有一点不同：新建操作调用 CDocTemplate::OpenDocumentFile()时使用参数 NULL，而打开操作调用该函数时传入文件名。

图 7-4 用图示说明了我们现在所处的位置。

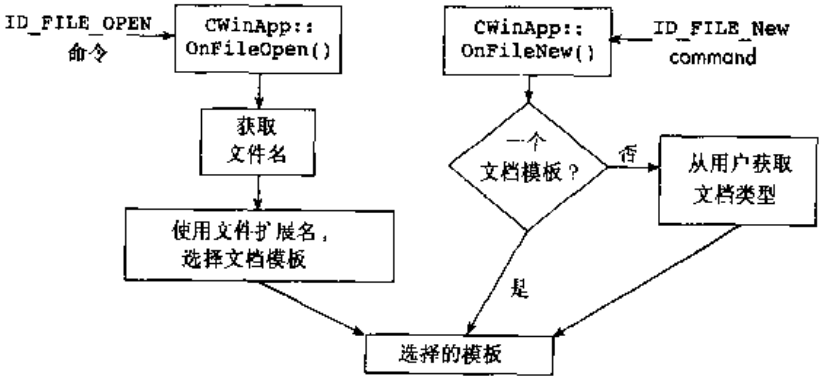


图 7-4 创建/打开文档过程的第一步，目标是选择一个文档模板

汇合路径：CDocTemplate::OpenDocumentFile()

根据应用程序类型的不同（SDI 或 MDI），现在或者调用 CSingleDocTemplate::OpenDocumentFile()，或者调用 CMultiDocTemplate::OpenDocumentFile()。本例中假设调用的是 CMultiDocTemplate::OpenDocumentFile()（MDI 情形）。

首先 CMultiDocTemplate::OpenDocumentFile()调用 CDocTemplate::CreateNewDocument()创建一个新文档，后者根据文档模板中记录的运行时信息创建一个文档对象。下一步，CMultiDocTemplate::OpenDocumentFile()调用 CDocTemplate::CreateNewFrame()创建一个框架。这个函数根据文档模板中记录的运行时信息创建一个框架对象。

新框架的创建引发一系列的事件（WM_CREATE、CFrameWnd::OnCreate()、CFrameWnd::OnCreateHelper()、CFrameWnd::OnCreateClient()、CFrameWnd::CreateView()），最后的结果是一个新的视图被创建，根据的信息是运行时信息。其中运行时信息最初时被传递给文档模板，后来被放在 CCreateContext 辅助类中。

现在，如果调用 CDocTemplate::OpenDocumentFile()时指定了文件名，就调用 CMyDoc::OnOpenDocument()读入序列化的文档。读入这个文档（反序列化过程）之后，打开和新建操作又统一到相同的路径。

最后，CDocTemplate::OpenDocumentFile()调用 CMyDoc::OnNewDocument()完成初始化。

现在，所有 3 个文档/视图对象都已经创建好了，图 7-5 展示了打开/新建操作的最后一部分流程。

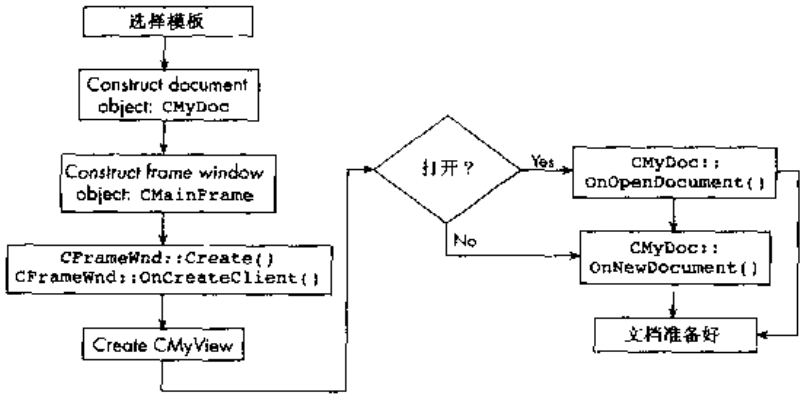


图 7-5 选择模板之后的控制流程

最后，为了保证你理解了这个非常重要的控制流程，图 7-6 展示了真正的函数调用及其源文件，你可以自己走一遍。

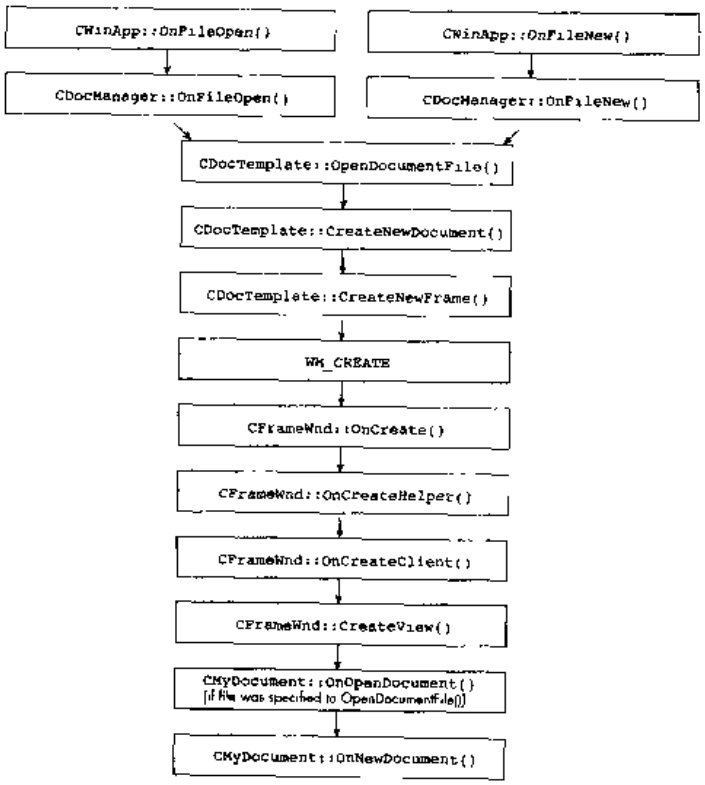


图 7-6 创建文档/视图体系中所有对象过程中调用的函数

结 语

现在，你已经对 MFC 文档/视图体系有了一个较好的理解，并且知道了各个部分如何组装在一起。下一章将继续深入这一话题。我们将揭示的一些文档/视图特性包括没有文档说明的 CDocument 辅助类、CMirrorFile 以及高级 CView 主题，例如打印、打印预览、派生类等。

CHAPTER

8 高级文档/视图结构内幕

在讨论文档/视图的最后一章里，我们看所有的这些问题是如何协作的：这是文档/视图启迪的第一步。在这一章里，我们会集中分析那些利用文档/视图结构可以获得的特性，以及MFC 如何实现这些特性。这也是文档/视图启迪的第二步。

首先，我们从前面提到过的 CDocument 的一个很有趣的内幕开始。然后我们先看 CView 对打印和打印预览的支持。最后我们再看一些有趣的 CView 派生类（这些派生类提供了对滚动的支持、对窗体的支持）以及 CView 对 Windows 控件提供的接口。

我们已经知道的 CDocument 内幕是什么呢？

CMirrorFile

MFC 4.0 对 CDocument 类做了很大改进。它增加了两个虚函数 GetFile()和 ReleaseFile()，从而开发人员可以在 CDocument 的派生类中指定一个特定的 CFile 派生类。

只要 CDocument 需要对文件做操作，它就会使用文件名、文件权限和一些其他参数调用 GetFile()函数。GetFile()会返回一个 CFile 指针，这个指针可以指向 CFile 的任何派生类。

下面我们看一下 CDocument::GetFile()的默认实现，如程序清单 8-1 所示。这些代码都在文件 DOCCORE.CPP 中。

程序清单 8-1　CDocument::GetFile 的实现（在文件 DOCCORE.CPP 中）

```
CFile* CDocument::GetFile(LPCTSTR lpszFileName, UINT nOpenFlags,
CFileException* pError)
{
    CMirrorFile* pFile = new CMirrorFile;
    if (!pFile->Open(lpszFileName, nOpenFlags, pError)) {
        delete pFile;
        pFile = NULL;
    }
    return pFile;
}
```

GetFile()的实现和你想像中的差不多，只不过第一行有点特殊。这个没有文档说明的 CMirrorFile 类的功能是什么？为什么文档中使用了它，而没有使用 CFile？这对你的程序有

什么影响呢？CMirrorFile 是和微软的网络相连接而将你的文档发送给微软吗？

只有一个办法能够回答这些问题，那就是继续发掘内幕。

CMirrorFile 内幕

CMirrorFile 的声明在头文件 AFXPRIV.H 中。程序清单 8-2 中列出了这个 CFile 派生类的声明，在文件 AFXPRIV.H 中。

程序清单 8-2 CMirrorFile 的声明（在文件 AFXPRIV.H 中）

```
class CMirrorFile : public CFile
{
// Implementation
public:
    virtual void Abort();
    virtual void Close();
    virtual BOOL Open(LPCTSTR lpszFileName, UINT nOpenFlags,
                      CFileException* pError = NULL);
protected:
    CString m_strMirrorName;
};
```

这是 CFile 派生类中最小的一个。除了覆盖 Open()、Close()和 Abort()，它还增加了一个数据成员 m_strMirrorName。

CMirrorFile::Open()

CMirrorFile 的实现和 CDocument 的大部分程序清单都在 DOCCORE.CPP 中。程序清单 8-3 列出了 CMirrorFile::Open()的伪代码（你可能希望在继续看下去之前休息一下，因为你会被下面的内容吓倒）。

程序清单 8-3 CMirrorFile::Open()的伪代码（在文件 DOCCORE.CPP 中）

```
BOOL CMirrorFile::Open(LPCTSTR lpszFileName, UINT nOpenFlags,
CFileException* pError)
{
    m_strMirrorName.Empty();
    CFileStatus status;
    if (nOpenFlags & CFile::modeCreate) {
        if (CFile::GetStatus(lpszFileName, status)){
            CString strRoot;
            AfxGetRoot(lpszFileName, strRoot);
            DWORD dwSecPerClus, dwBytesPerSec, dwFreeClus, dwTotalClus;
            int nBytes = 0;
            if (GetDiskFreeSpace(strRoot, &dwSecPerClus, &dwBytesPerSec,
            &dwFreeClus, &dwTotalClus)){
                nBytes = dwFreeClus*dwSecPerClus*dwBytesPerSec;
                if (nBytes > 2*status.m_size){
                    // get the directory for the file TCHAR
                    szPath[_MAX_PATH];
                    LPTSTR lpszName;
                    GetFullPathName(lpszFileName,_MAX_PATH, szPath,
                    &lpszName);
                    *lpszName = NULL;
                    GetTempFileName(szPath, _T("MFC"), 0,
```

```
                        m_strMirrorName.GetBuffer(_MAX_PATH+1));
                    m_strMirrorName.ReleaseBuffer();
                }
            }
        }
        if (!m_strMirrorName.IsEmpty() &&
            CFile::Open(m_strMirrorName, nOpenFlags, pError)){
                m_strFileName = lpszFileName;
                FILETIME ftCreate, ftAccess, ftModify;
                if (::GetFileTime((HANDLE)m_hFile, &ftCreate, &ftAccess,
                     ftModify)) {
                    AfxTimeToFileTime(status.m_ctime, &ftCreate);
                    SetFileTime((HANDLE)m_hFile, &ftCreate, &ftAccess,
                        &ftModify);
        }
        DWORD dwLength = 0;
        PSECURITY_DESCRIPTOR pSecurityDescriptor = NULL;
        GetFileSecurity(lpszFileName, DACL_SECURITY_INFORMATION,
            NULL, dwLength, &dwLength);
        pSecurityDescriptor = (PSECURITY_DESCRIPTOR) new BYTE[dwLength];
        if (::GetFileSecurity(lpszFileName, DACL_SECURITY_INFORMATION,
            pSecurityDescriptor, dwLength, &dwLength)){
            SetFileSecurity(m_strMirrorName, DACL_SECURITY_INFORMATION,
                pSecurityDescriptor);
        }
        delete[] (BYTE*)pSecurityDescriptor;
        return TRUE;
    }
    m_strMirrorName.Empty();
    return CFile::Open(lpszFileName, nOpenFlags, pError);
}
```

CMirrorFile::Open()在逻辑上可以分成两个部分。在第一部分里，Open()先检查modeCreate，看调用者是想创建一个新文件还是想连接在已有的文件后面。然后，它调用了CFile::GetStatus()。如果文件非空，GetStatus()会返回一个非零的数（表示用户希望连接在已有文件的后面）。如果文件存在，Open()接着会调用 GetDiskFreeSpace()，确定驱动器上有多少个字节可以用。然后将这个结果和已有文件大小的两倍进行比较。如果前者大于后者，Open()就会创建一个临时文件，并将文件名存储在变量 m_strMirrorName 中。

Open()的第二部分仅当 *m_strMirrorName* 不为空时才被执行，也就是说第一部分运行正常，而且获得了一个临时文件。Open()会调用 CFile::Open()打开镜像文件。然后将文件的时间和文件的权限从源文件拷贝给镜像文件。如果执行了第二部分，Open()会返回 TRUE。如果第二部分没有执行，Open()会调用 CFile::Open()，返回调用的结果。

简而言之，如果有写操作正在进行（CFile::modeCreate）或者有文件被覆盖，CMirrorFile::Open()就打开一个和指定文件不同的文件。

例如，在 Scribble 中（它使用了文档/视图结构），如果你写了一个新的 scribble，并将它保存在 MFCNTRNL.SCR 中，这些代码不会被执行。但是，如果你装载这个文件并做些修改，那它就可以执行了，Scribble 也就能向镜像文件中写数据了。

CMirrorFile::Close()

下面再看 CMirror::Close()。程序清单 8-4 列出了 CMirrorFile::Close()的伪代码，这些代码在文件 DOCCORE.CPP 中。

程序清单 8-4 CMirrorFile::Close()的实现（在文件 DOCCORE.CPP 中）

```
void CMirrorFile::Close()
{
    CString m_strName = m_strFileName;
    CFile::Close();
    if (!m_strMirrorName.IsEmpty()) {
        CFile::Remove(m_strName);
        CFile::Rename(m_strMirrorName, m_strName);
    }
}
```

首先，Close()在 m_strName 中存储了文件的名字，因为 CFile::Close()调用会清除 m_strFileName 的内容。

在调用 CFile::Close()之后，它会检查是否使用了镜像文件。如果使用了镜像文件，Open()就会删除指定文件，将镜像文件拷贝为指定文件。

CMirrorFile 小结

总之，CMirrorFile 会通过保存原有文件和对临时文件进行写操作来保护你的文档。这样，如果写操作出现问题，源文件也是安全的，如果没有出现问题，CMirrorFile 就会将新文件拷贝成源文件。CMirrorFile 还会确保源文件的安全性以及文件创建信息能被正确地拷贝。

你可能不赞同 MFC 的这种做法。如果你不喜欢 CDocument 使用 CMirrorFile，可以覆盖 GetFile()文件，使它始终返回一个旧式的 CFile。

CMirrorFile 实际上只是 CDocument 的内幕。因为你的数据都在 CDocument 中，所以它的内幕也就取决于开发人员。

CView 就不同了。因为它的任务是显示文档中的数据，所以就会涉及到很多有趣的 Windows API、GDI 技巧和 CView 派生类。我们先看 CView 如何支持打印。

CView 打印

CView 是 MFC 对打印和打印预览支持的关键。通过对传递给 CView 派生类的 OnDraw()成员函数的设备环境进行操作，框架对应用程序增加了对打印和打印预览的支持。如果你对打印和打印预览的默认实现很满意，就没必要在应用程序中再增加关于打印/打印预览的代码。

如果你要在一个非 MFC 的 Windows 应用程序中增加打印功能，可能就想知道如何完成底层的工作。所以我们首先看 CView 在打印过程中用到的成员函数，看 CView 用了哪些打印技巧。

CView 打印概述

为了打印，CView 提供了几个可以覆盖的虚函数。框架通过调用这些函数在视图内支持

打印。这些函数包括 OnPreparePrinting()、OnBeginPrinting()、OnPrint()、OnEndPrinting()和 OnPrepareDC()。

OnPreparePrinting()在打印工作启动之前由框架调用。它只有一个参数：一个 CPrintInfo 结构。CPrintInfo 包含了文档中的页数、要打印的页码范围等信息。你可以用 OnPreparePrinting()来指定文档中的页数。

打印工作开始后，框架会调用 OnBeginPrinting()函数。MFC 会提供一个指向已选中的打印机的设备环境的指针，和一个指向 CPrintInfo 结构的指针作为 OnBeginPrinting()的参数。可以覆盖 OnBeginPrinting()为打印操作分配任何特定的 GDI 资源。如果页数受打印设备环境的某些性质的影响，你也可以利用 OnBeginPrinting()设置打印页数。

框架用 CView::OnPrint()打印文档的特定部分。如果你希望打印输出和屏幕输出有所区别，就需要覆盖这个函数。如果你希望有些输出不出现在屏幕上（例如有标题的页面），也可以使用这个函数。默认情况下，OnPrint()会将实际打印委托给 CView 的 OnDraw()函数，并将打印机的设备环境传递给它。

如果打印结束，框架会调用 OnEndPrinting()。这个函数的目的是清理所有在 OnBeginPrinting()里分配的资源。

最后，如果有什么特殊工作需要做，MFC 就使用 OnPrepareDC()准备设备环境。需要覆盖 OnPrepareDC()的理由可能有如下 3 个：

1. 为单独的页设置映射模式或设备环境的其他属性。

2. 在打印文档时判断是否到了文档的末尾。虽然你可能使用 OnPreparePrinting()指定了文档的长度，但是你可能提前并不知道文档的长度。在这种情况下，你就需要 OnPrepareDC()测试文档的末尾。

3. 在打印时执行一些其他的打印函数。

CView 还有一个辅助函数，DoPreparePrinting()。根据选入对话框的打印机，框架会使用这个函数创建一个设备环境，显示打印对话框。

CPrintInfo 结构

MFC 的打印结构中另一个重要组成部分是名为 CPrintInfo 的结构。这个类是有文档说明的（开发人员和框架都可以使用它）。CPrintInfo 维护了单个打印任务所需要的所有参数。CPrintInfo 中有 CPrintDialog 的一个实例，以及从 CPrintDialog 中获取信息的函数。CPrintInfo 还携带了关于打印任务的信息，包括输出是送往打印机还是打印预览窗口，以及打印任务的当前页。你一会儿也会看到，这个结构贯穿了 MFC 的打印周期。程序清单 8-5 列出了 CPrintInfo 结构的声明，在文件 AFXEXT.H 中。

程序清单 8-5 CPrintInfo 结构（在文件 AFXEXT.H 中声明）

```
struct CPrintInfo // Printing information structure
{
    CPrintInfo();
    ~CPrintInfo();
    CPrintDialog* m_pPD;
    BOOL m_bPreview;
    BOOL m_bDirect;
    BOOL m_bContinuePrinting;
```

```
    UINT m_nCurPage;
    UINT m_nNumPreviewPages;
    CString m_strPageDesc;
    LPVOID m_lpUserData;
    CRect m_rectDraw;
    void SetMinPage(UINT nMinPage);
    void SetMaxPage(UINT nMaxPage);
    UINT GetMinPage() const;
    UINT GetMaxPage() const;
    UINT GetFromPage() const;
    UINT GetToPage() const;
};
```

下面是这个结构中每个域的功能：

- m_pPD　一个指向打印对话框的指针。
- m_bPreview——如果是预览模式就是 TRUE，否则是 FALSE。
- m_bDirect——如果是 TRUE 就忽略对话框，否则就显示对话框。
- m_bContinuePrinting——设置为 FALSE 表示停止打印。
- m_nCurPage——当前页。
- m_nNumPreviewPages——预览中显示的页数。
- m_strPageDesc——显示页码的字符串格式。
- m_lpUserData——指向用户创建的结构。
- m_rectDraw——一个定义了当前可用的页区域的矩形。
- CPrintInfo()——创建一个 CPrintDialog，并在 m_pPD 中存储它。初始化成员变量和 m_pPD->m_pd PRINTDLG 结构的部分域，例如最小页和最大页。
- SetMinPage()——设置 m_pPD->m_pd PRINTDLG 结构的最小页这个域。
- SetMaxPage()——设置 m_pPD->m_pd PRINTDLG 结构的最大页这个域。
- GetMinPage()、GetMaxPage()、GetFromPage()、GetToPage()——从 m_pPD->m_pd PRINTDLG 结构中获得相关域。

我们会在下一节看 CPrintInfo 的这些域。

CView 打印内幕

当用户选择 File/Print 时，打印便开始了。CView 消息映射表入口是这样的：

```
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
```

让我们从 OnFilePrint()开始，顺着 CView 的打印踪迹去探索。注意：CView 打印的代码很多都在 VIEWPRNT.CPP 中。

CView::OnFilePrint()

OnFilePrint()是一个很长的成员函数，所以我们把它分成几个部分。我们首先看它的第一部分，它负责准备打印。我们会在下一个部分看它实际上如何循环打印。

程序清单 8-6 是 OnFilePrint()的第一部分代码，在文件 VIEWPRNT.CPP 中。

程序清单 8-6 CView::OnFilePrint()的第一部分代码（在文件 VIEWPRNT.CPP 中）

```
void CView::OnFilePrint()
{
    CPrintInfo printInfo;
    // **omitted - OnFilePrint() checks the command line to see if it
    // should display a CPrintDialog or not (m_bDirect data member
    //  is set.
    if (OnPreparePrinting(&printInfo)) {
        //*omitted - Will get filename if user is printing to file
        // by displaying a CFileDialog.
        CString strTitle;
        CDocument* pDoc = GetDocument();
        strTitle = pDoc->GetTitle();
        // setup the printing DC and DOCINFO
        DOCINFO docInfo;
        docInfo.cbSize = sizeof(DOCINFO);
        docInfo.lpszDocName = strTitle;
        CDC dcPrint;
        dcPrint.Attach(printInfo.m_pPD->m_pd.hDC);
        dcPrint.m_bPrinting = TRUE;
        OnBeginPrinting(&dcPrint, &printInfo);
        dcPrint.SetAbortProc(_AfxAbortProc);
        AfxGetMainWnd()->EnableWindow(FALSE);
        CPrintingDialog dlgPrintStatus(this);
        // *omitted - CPrintingDialog setup
        dlgPrintStatus.ShowWindow(SW_SHOW);
        dlgPrintStatus.UpdateWindow();
        // .. To be continued in next chunk
```

OnFilePrint()首先在栈上声明了一个 CPrintInfo 结构，并构造了它。在它的构造函数里，CPrintInfo 分配了一个打印对话框，并把它赋值给 m_pPD 变量。CPrintInfo 将对话框的最小页码设置为 1，最大页码设置为 0xffff（65565）——这样页码就必须在这个范围之内。CPrintInfo 还初始化了其他成员变量：m_nCurPage 设置为 1，m_lpUserData 设置为 NULL，m_bPreview 设置为 FALSE，m_bContinuePrinting 设置为 TRUE。

一旦 OnFilePrint()构造了一个 CPrintInfo 结构，它就调用 OnPreparePrinting()。这是一个虚函数，这样控制就交给了派生类（如果这个函数被覆盖）。如果你使用 AppWizard 生成程序，则 CMyView::OnPreparePrinting()会调用 CView::DoPreparePrinting()。

通过显示 CPrintInfo 结构的 CPrintDialog，DoPreparePrinting()会创建一个打印机 DC。当用户从打印对话框中选择各种打印选项之后，打印机 DC 就存放在 pInfo->m_pPD->m_pd.hDC 中。

调用了 OnPreparePrinting()之后，OnFilePrint()获得了文档的题目，并创建了一个 DOCINFO 结构。DOCINFO 结构中包含了文档的名字、输出设备的名字。这个结构会在后面做为 Win32 API StartDoc()的参数。

然后，OnFilePrint()在栈顶创建了一个本地 CDC，并将它和 DoPreparePrinting()创建的 DC 句柄（保存在 PRINTDIALOG 结构中）绑定在一起。一旦 CDC 创建好了，OnFilePrint()会将 DC 和 CPrintInfo 结构一起发送给 CView::OnBeginPrinting()。OnBeginPrinting()的默认实现什么也不做，但是如果你想修改用于打印的 DC，就需要重载 OnBeginPrinting()。

在调用了 OnBeginPrinting()之后，OnFilePrint()开始准备打印。它为 DC 准备了一个退出

过程——_AfxAbortProc()，然后禁用主窗口。然后 OnFilePrint()创建、初始化并显示了一个 CPrintingDialog。

CPrintingDialog 是什么？CPrintingDialog 是一个很小的打印状态对话框，它会显示页码，让用户能够取消打印操作。图 8-1 显示了 MFC 应用程序 Scribble 中的 CPrintingDialog。

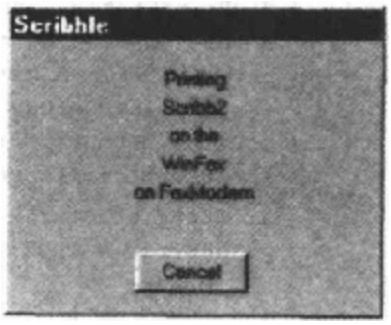


图 8-1 Scribble 中的 CPrintingDialog

如果用户希望取消打印，CPrintingDialog::OnCancel()会将一个全局退出标志设置为 TRUE。在打印期间，_AfxAbortProc()会检查这个标志，如果是 TRUE 就退出打印工作。

此时，在 OnFilePrint()中，已经创建了 CPrintInfo 结构，CPrintingDialog 也已经显示出来了，打印 DC 也已经创建。现在，我们就可以开始打印了。下面再看 OnFilePrint()的其他代码，看它如何使用第一部分已经创建的对象。

为什么要修改打印对话框？

CPrintingDialog 是一个很枯燥的打印对话框，如果有一个表示进度的指示器或一个打印机的图片就会更好。所以考虑一下修改 OnFilePrint()所使用的对话框。不幸的是，修改起来很复杂。它和方便的 CDocument::GetFile()不一样，没有 CView::GetPrintDlg()。所以你可能需要采取一些强制措施。下面就是如何实现你自己的打印对话框：

1. 在你的 CView 派生类中增加 CMyView::OnFilePrint()。
2. 修改消息映射表，使得在调用 CView 的 OnFilePrint()的地方变成调用你的 OnFilePrint()。
3. 把 CView::OnFilePrint()的代码拷贝到你的 OnFilePrint()中。
4. 修改 CPrintingDialog 调用附近的代码，让它调用你的 CDialog 派生类，而不是调用 CPrintingDialog。
5. 确定你的打印对话框的 Cancel 和 CPrintingDialog::OnCancel()一样。
6. 确保你已经修改了所有 CPrintingDialog 出现的地方，尤其是在 OnFilePrint()的末尾（更新对话框的地方）。

希望 MFC 未来的版本中会增加更多的定制函数，这样就不需要这么麻烦地修改了。

CView::OnFilePrint()的打印循环

程序清单 8-7 中是 OnFilePrint()的最后一部分伪代码。

程序清单 8-7 CView::OnFilePrint()的其余伪代码

```
// start document printing process
if (dcPrint.StartDoc(&docInfo) == SP_ERROR)
    // **omitted, some error handling that cleans and returns
int nStep = (nEndPage >= nStartPage) ? 1 : -1;
nEndPage = (nEndPage == 0xffff) ? 0xffff : nEndPage + nStep;
// begin page printing loop
BOOL bError = FALSE;
for (printInfo.m_nCurPage = nStartPage;printInfo.m_nCurPage != nEndPage;
    printInfo.m_nCurPage += nStep){
      OnPrepareDC(&dcPrint, &printInfo);
      // check for end of print
      if (!printInfo.m_bContinuePrinting)
```

```
            break;
        TCHAR szBuf[80];
        wsprintf(szBuf, strTemp, printInfo.m_nCurPage);
        dlgPrintStatus.SetDlgItemText(AFX_IDC_PRINT_PAGENUM, szBuf);
        // set up drawing rect to entire page (in logical coordinates)
        printInfo.m_rectDraw.SetRect(0, 0, dcPrint.GetDeviceCaps(HORZRES),
            dcPrint.GetDeviceCaps(VERTRES));
        dcPrint.DPtoLP(&printInfo.m_rectDraw);
        // attempt to start the current page
        if (dcPrint.StartPage() < 0){
            bError = TRUE;
            break;
        }

        OnPrint(&dcPrint, &printInfo);
        if (dcPrint.EndPage() < 0 || !_AfxAbortProc(dcPrint.m_hDC, 0)){
            bError = TRUE;
            break;
        }
    } //end page for loop here.
    // cleanup document printing process
    if (!bError)
        dcPrint.EndDoc();
    else
        dcPrint.AbortDoc();
    AfxGetMainWnd()->EnableWindow();    // enable main window

    OnEndPrinting(&dcPrint, &printInfo);    // clean up after printing
    dlgPrintStatus.DestroyWindow();
    dcPrint.Detach();   // will be cleaned up by CPrintInfo destructor
    }
}
```

在 OnFilePrint()的这一部分，首先调用了 CDC::StartDoc()来启动 Windows 打印引擎。然后它计算一步（-1 表示一页，1 表示多页），确定最后一页，并将 bError 设置为 FALSE。

然后，OnFilePrint()进入 for 循环，开始进行打印。这里的 for 循环对文档中的每一页做如下操作：

1. 调用 OnPrepareDC()，它的默认实现是什么都不做。
2. 更新 CPrintingDialog 上的页号。
3. 设置 CPrintInfo.m_rectDraw 矩形，表示页面的大小。
4. 调用 CDC::StartPage()，表示将要打印新的一页。
5. 调用 CView::OnPrint()，参数是 DC 指针和 CPrintInfo 结构。CView::OnPrint()调用 CView::OnDraw()，在这个时候，你的 CView 派生类会在 DC 中绘画，然后生成打印机输出。
6. 调用 EndPage()。

每个页面都打印之后，OnFilePrint()会调用 EndDoc()，激活窗口，并且调用 OnEndPrinting()。OnEndPrinting()的默认实现是什么都不做。如果你在打印过程中分配了什么，就可以覆盖这个虚函数，释放那些分配的资源。

最后，OnFilePrint()销毁了 CPrintingDialog 打印状态对话框，并将它和 DC 分离。

总而言之，OnFilePrint()驱动的打印过程如下：

- 准备打印。
- 打印每个页面。
- 清理资源。

贯穿整个打印过程，可覆盖的虚函数为开发人员提供了定制打印过程的机会。图 8-2 以流程图的方式显示了 MFC 的打印过程。

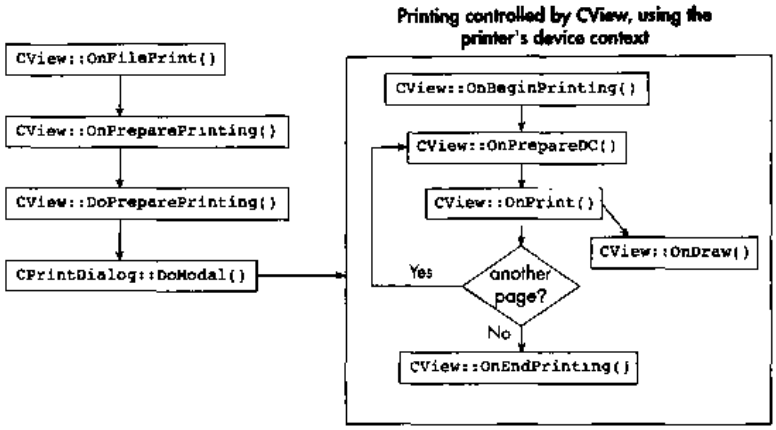


图 8-2 MFC 打印过程流程图

CView 的打印预览的工作原则和这个类似，但是因为在屏幕上模拟已打印的页是一项很复杂的工作，所以它必须做得更复杂。

CView 对打印预览支持的内幕

和打印一样，打印预览也调用 CMyView::OnDraw()函数生成输出。但是和打印有所区别的是它的过程更复杂。如果你看到如图 8-3 所示的 MFC 打印预览窗口，就会有很多疑问。

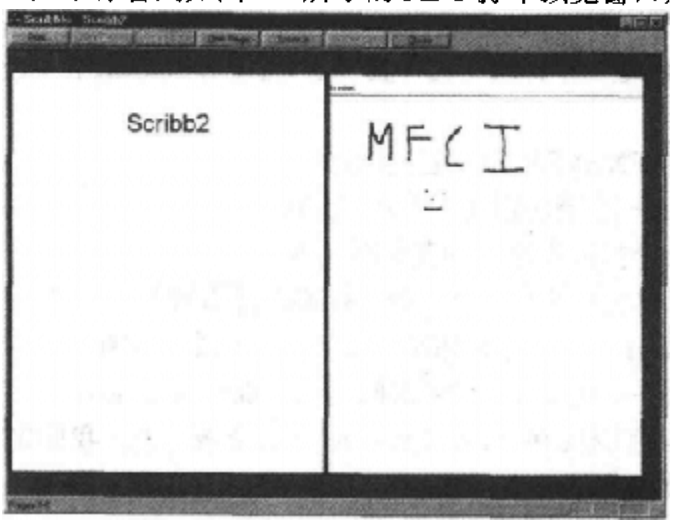


图 8-3 打印预览的示例

- MFC 如何用打印预览窗口替换你的 Windows 窗口？

- 它如何画出页面的轮廓？
- 它如何支持缩放和滚动？
- 它如何在显示器上模拟打印输出？
- 工具条从哪里来？

我们会回答这些问题。事实证明，在 CView 对打印预览的支持中，有很多有价值的没有文档说明的类和辅助结构。下面我们就看这些没有文档说明的类。

CView::OnFilePrintPreview()

和打印一样，打印预览也是通过 File/Print Preview 菜单选项的消息映射表入口开始的：

```
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
```

下面我们看一看 CView::OnFilePrintPreview()下一步做了什么。程序清单 8-8 是这个函数的伪代码。注意：所有关于 CView 对打印预览的支持都在文件 VIEWPREV.CPP 中。

程序清单 8-8 CView::OnFilePrintPreview()（在文件 VIEWPREV.CPP 中）

```
void CView::OnFilePrintPreview()
{
    CPrintPreviewState* pState = new CPrintPreviewState;
    if (!DoPrintPreview(AFX_IDD_PREVIEW_TOOLBAR, this,
        RUNTIME_CLASS(CPreviewView), pState)){
        delete pState;    // preview failed to initialize, delete State now
    }
}
```

和 OnFilePrint()相比，OnFilePrintPreview()很小。它只是从堆上创建了一个 CPrintPreviewState 对象，然后传递给 DoPrintPreview()。DoPrintPreview()的参数还包括打印预览工具条的资源 ID（AFX_IDD_PREVIEW_TOOLBAR）、一个 this 指针和 RUNTIME_CLASS（CPreviewView）。然后，我们看一下 CPrintPreviewState 和 DoPrintPreview()，我们后面再讨论 CPreviewView。

CPrintPreviewState

CPrintPreviewState 是一个辅助结构（类似于 CCreateContext 和 CPrintInfo），它存储了关于打印预览的信息。

它的声明在文件 AFXEXT.H 中，它的域有：

- nIDMainPane——隐藏的主窗口的资源 ID。
- hMenu——一个指向保存了的菜单的句柄。
- dwStates——控制栏的状态位，表示控制栏是否可见。
- pViewActiveOld——在预览期间，保存前一个活动视图。
- hAccelTable——一个保存了的加速表（accelerator table）。

我们会在后面详细描述 CPrintPreviewState 的各个域。现在我们先看 DoPrintPreview()这个函数，它由 OnFilePrintPreview()调用。

DoPrintPreview()内幕

CView::DoPrintPreview()有很多参数。程序清单 8-9 是它的伪代码，在文件 VIEWPREV.

CPP 中。

程序清单 8-9 CView::DoPrintPreview()的伪代码（在文件 VIEWPREV.CPP 中）

```
BOOL CView::DoPrintPreview(UINT nIDResource, CView* pPrintView,
                           CRuntimeClass* pPreviewViewClass,
                           CPrintPreviewState* pState)
{
    CFrameWnd* pParent = (CFrameWnd*)AfxGetThread()->m_pMainWnd;
    CCreateContext context;
    context.m_pCurrentFrame = pParent;
    context.m_pCurrentDoc = GetDocument();
    context.m_pLastView = this;
    // Create the preview view object
    CPreviewView* pView = (CPreviewView*)pPreviewViewClass->CreateObject();
    pView->m_pPreviewState = pState;         // save pointer
    pParent->OnSetPreviewMode(TRUE, pState);     // Take over Frame Window

    // Create the toolbar from the dialog resource
    pView->m_pToolBar = new CDialogBar;
    if (!pView->m_pToolBar->Create(pParent,
                                   MAKEINTRESOURCE(nIDResource),CBRS_TOP,
                                   AFX_IDW_PREVIEW_BAR)){
        TRACE0("Error: Preview could not create toolbar dialog.\n");
        return FALSE;
    }
    pView->m_pToolBar->m_bAutoDelete = TRUE;     // automatic cleanup
    if (!pView->Create(NULL, NULL, AFX_WS_DEFAULT_VIEW,
        CRect(0,0,0,0), pParent, AFX_IDW_PANE_FIRST, &context)) {
        TRACE0("Error: couldn't create preview view for frame.\n");
        return FALSE;
    }
    pState->pViewActiveOld = pParent->GetActiveView();
    CView* pActiveView = pParent->GetActiveFrame()->GetActiveView();
    pActiveView->OnActivateView(FALSE, pActiveView, pActiveView);
    pView->SetPrintView(pPrintView);
    pParent->SetActiveView(pView);  // set active view - even for MDI
    // update toolbar and redraw everything
    pView->m_pToolBar->SendMessage(WM_IDLEUPDATECMDUI, (WPARAM)TRUE);
    pParent->RecalcLayout();            // position and size everything
    pParent->UpdateWindow();
    return TRUE;
}
```

DoPrintPreview()首先将主窗口的指针存放在 pParent 中，然后创建了一个局部的 CCreateContext 结构，并为它的各个域赋值。

然后，它使用动态创建函数（以 CRuntimeClass 指针做参数）创建了一个 CPreviewView 的实例。调用 CFrameWnd::OnSetPreviewMode()之后，根据工具栏资源的参数创建一个工具栏。

创建了工具栏，并将工具栏保存在 CPreviewView 指针之后，DoPrintPreview()创建了视图，并激活了它。在设置 CPreviewView 为活动视图时，DoPrintPreview()在 CPrintPreviewState 结构中保存了前一个活动视图。

一旦 CPreview 和工具栏都创建好了，而且激活了，DoPrintPreview()会更新它们，最终返回 TRUE。

到此为止，我们已经讨论很多关于打印预览的内容。一个有趣的地方就是 DoPrintPreview()在主窗口中如何设置活动视图，从而从 MDI 或 SDI 窗口切换到打印预览视图。如果你想有一个全窗口模式或者想产生类似的效果，就可以在你的 MFC 应用程序中使用这一技术。如果想知道它如何运作，请查看 CFrameWnd::OnSetPreviewMode()。

目前我们的大多数问题还都没有解决。所有的线索都指向了同一个地方：没有文档说明的类 CPreviewView。

CPreviewView：一个没有文档说明的打印预览类，CView 的派生类

在 MFC 最秘密的头文件 AFXPRIV.H 中，我们找到了 CPreviewView 的声明，它揭示了这个类的大部分内容。程序清单 8-10 中是它的声明的缩减版本。

程序清单 8-10 CPreviewView 的声明（在文件 AFXPRIV.H 中）

```
class CPreviewView : public CScrollView
{
    DECLARE_DYNCREATE(CPreviewView)
// Constructors
public:
    CPreviewView();
    BOOL SetPrintView(CView* pPrintView);
// Attributes
protected:
    CView* m_pOrigView;
    CView* m_pPrintView;
    CPreviewDC* m_pPreviewDC;  // Output and attrib DCs Set, not created
    CDC m_dcPrint;             // Actual printer DC
// Operations *omitted
// Overridables *omitted
// Implementation * some omitted
public:
    virtual void OnPrepareDC(CDC* pDC, CPrintInfo* pInfo = NULL);
protected:
    afx_msg void OnPreviewClose();
    afx_msg int  OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg void OnDraw(CDC* pDC);
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg BOOL OnEraseBkgnd(CDC* pDC);
    afx_msg void OnNextPage();
    afx_msg void OnPrevPage();
    afx_msg void OnPreviewPrint();
    afx_msg void OnZoomIn();
    afx_msg void OnZoomOut();
    void DoZoom(UINT nPage, CPoint point);
    void SetScaledSize(UINT nPage);
    CSize CalcPageDisplaySize();
    CPrintPreviewState* m_pPreviewState; // State to restore
    CDialogBar* m_pToolBar; // Toolbar for preview
    struct PAGE_INFO {
```

```
        CRect rectScreen; // screen rect (screen device units)
        CSize sizeUnscaled; // unscaled screen rect (screen device units)
        CSize sizeScaleRatio; // scale ratio (cx/cy)
        CSize sizeZoomOutRatio; // scale ratio when zoomed out (cx/cy)
    };
    PAGE_INFO* m_pPageInfo; // Array of page info structures
    PAGE_INFO m_pageInfoArray[2]; // Embedded array for the default
                                  // implementation
    BOOL m_bPageNumDisplayed;  // Flags whether or not page number has yet
    // been displayed on status line
    UINT m_nZoomOutPages; // number of pages when zoomed out
    UINT m_nZoomState;
    UINT m_nMaxPages; // for sanity checks
    UINT m_nCurrentPage;
    UINT m_nPages;
    int m_nSecondPageOffset; // used to shift second page position
    HCURSOR m_hMagnifyCursor;
    CSize m_sizePrinterPPI; // printer pixels per inch
    CPoint m_ptCenterPoint;
    CPrintInfo* m_pPreviewInfo;
    DECLARE_MESSAGE_MAP()
};
```

CPreviewView 这个类可是够大的。你会发现它是 CScrollView 的派生类。我们还没有讨论过 CScrollView 这个派生类，但是从名字能看出来它提供了对滚动的支持。如果你在预览的时候缩小窗口，就会看到显示出来的滚动条：这就是 CScrollView 这个派生类的工作。

CPreviewView 有很多有趣的内幕。下面列出的是它的一些有趣的数据成员：

- m_pOrigView——指向前一个活动窗口的指针。
- m_pPrintView——指向待打印的视图的指针。
- m_pPreviewDC——指向 CPreviewDC 的指针（什么是 CPreviewDC？我们后面会解释）。
- m_dcPrint——一个指向打印机 DC 的指针。
- PAGE_INFO——一个定义了预览页面的大小的方便结构。
- m_pPageInfo——指向 PAGE_INFO 结构的指针。
- m_pageInfoArray——一个两页信息结构数组。
- m_bPageNumDisplayed——一个表示页码是否显示的标志。
- m_nZoomOutPages——存储了在打印预览视图中待显示的页数。
- m_nZoonState——可以是以下四种缩放状态之一（通过 SetZoomState()设置）：
  - ZOOM_OUT——默认值，显示的比例略小于 1:1。
  - ZOOM_MIDDLE——缩放级别在放大和缩小之间。
  - ZOOM_IN——显示比例略大于 1:1。
  - ZOOM_OFF——没有缩放。
- m_nMaxPages——指定了页数。对于 CPreviewView 来说通常是 2。
- m_nCurrentPage——当前显示的页面。
- m_nPages——页数。在 CPreviewView 中默认值是 1 或 2。
- m_nSecondPageOffset——指定了第二个预览页的位置的左坐标。
- m_hMagnifyCursor——指向放大酒杯光标的句柄。

- m_sizePrinterPPI——存储通过调用 GetDeviceCaps(LOGPIXELSX)而得到的打印机上页面的大小。
- m_ptCenterPoint——从来没使用过。
- m_pPreviewInfo—— 一个指向 CPrintInfo 结构的指针。

看了这些数据成员你可能会有些疑问：

- ZOOM_MIDDLE 是什么？为什么要使用它？
- m_ptCenterPoint 又是干什么的？为什么从来不用还要定义？

对于 CPreviewView 的探索，我们从 SetPrintView()成员函数开始。回忆一下，这个函数曾经在 CView::DoPrintPreview()（在程序清单 8-9 中）中被调用过。

CPreviewView::SetPrintView()内幕

SetPrintView 对于 CPreviewView 而言是一个非常重要的函数。它负责初始化打印预览视图，并准备启动打印预览进程。在你浏览它的伪代码时，请记住这一点。它的伪代码在程序清单 8-11 中，在文件 VIEWPREV.CPP 里。

程序清单 8-11 CPreviewView::SetPrintView()的伪代码（在 VIEWPREV.CPP 文件中）

```
BOOL CPreviewView::SetPrintView(CView* pPrintView)
{
    m_pPrintView = pPrintView;
    m_pPreviewInfo = new CPrintInfo;
    m_pPreviewInfo->m_pPD->SetHelpID(AFX_IDD_PRINTSETUP);
    m_pPreviewInfo->m_pPD->m_pd.Flags |= PD_PRINTSETUP;
    m_pPreviewInfo->m_pPD->m_pd.Flags &= ~PD_RETURNDC;
    m_pPreviewInfo->m_bPreview = TRUE;  // signal that this is preview
    m_pPreviewDC = new CPreviewDC;      // must be created before any
    if (!m_pPrintView->OnPreparePrinting(m_pPreviewInfo))
        return FALSE;
    m_dcPrint.Attach(m_pPreviewInfo->m_pPD->m_pd.hDC);
    m_pPreviewDC->SetAttribDC(m_pPreviewInfo->m_pPD->m_pd.hDC);
    m_pPreviewDC->m_bPrinting = TRUE;
    m_dcPrint.m_bPrinting = TRUE;
    m_dcPrint.SaveDC();     // Save pristine state of DC
    HDC hDC = ::GetDC(m_hWnd);
    m_pPreviewDC->SetOutputDC(hDC);
    m_pPrintView->OnBeginPrinting(m_pPreviewDC, m_pPreviewInfo);
    m_pPreviewDC->ReleaseOutputDC();
    ::ReleaseDC(m_hWnd, hDC);
    m_dcPrint.RestoreDC(-1);    // restore to untouched state
    // Get Pixels per inch from Printer
    m_sizePrinterPPI.cx = m_dcPrint.GetDeviceCaps(LOGPIXELSX);
    m_sizePrinterPPI.cy = m_dcPrint.GetDeviceCaps(LOGPIXELSY);
    m_nPages = m_pPreviewInfo->m_nNumPreviewPages;
    m_nZoomOutPages = m_nPages;
    SetScrollSizes(MM_TEXT, CSize(1, 1));  // initialize mapping mode only
    if (m_pPreviewInfo->GetMaxPage() < 0x8000 &&
       m_pPreviewInfo->GetMaxPage() - m_pPreviewInfo->GetMinPage() <=
       32767U)
          SetScrollRange(SB_VERT, m_pPreviewInfo->GetMinPage(),
```

```
            m_pPreviewInfo->GetMaxPage(), FALSE);
    else
        ShowScrollBar(SB_VERT, FALSE);
    SetCurrentPage(m_pPreviewInfo->m_nCurPage, TRUE);
    return TRUE;
}
```

SetPrintView()首先设置了 m_pPrintView 的值，使它指向文档的视图。记住，文档的视图在 OnDraw()函数中完成显示。因为 CPreviewView 使用视图的 OnDraw()函数来截取打印预览窗口中的数据，所以 CPreviewView 必须保存文档的活动视图的备份。然后，SetPrintView()又创建了一个新的 CPrintInfo 对象。从 CView 打印的角度看，我们发现 CPrintInfo 的构造函数又创建了一个 CPrintDialog。在创建了 CPrintInfo 对象之后，SetPrintView()设置了一些标志。最终的一个标志就是告诉 CPrintDialog 不要返回一个 DC。我们一会儿会看为什么。

在创建了 CPrintInfo 对象之后，它又创建了一个 CPreviewDC 对象。你可能会问，CPreviewDC 是什么？它是另一个没有文档说明的 MFC 类。MFC 的 CDC 类有两个句柄指向设备环境：m_hDC 和 m_hAttribDC。m_hDC 成员变量表示输出的设备环境，m_hAttribDC 用于保存信息（又称“只读”DC）。通常这些设备环境都表示相同的事物——屏幕或打印机。

CPreviewDC 有两个不同的设备环境：一个是屏幕（m_hDC），另一个是打印机（m_hAttribDC）。CPreviewDC 这样实现是因为即使输出是在打印机上，打印预览也必须画些什么，但是在屏幕上显示得要正确。SetPrintView()设置了 CPreviewDC——将 m_hDC 指向视图的 hDC，将 m_hAttribDC 指向打印机的 hDC。为了学习更多关于 CPreviewDC 如何将打印机的输出显示在屏幕上这个问题，请查看 MFC 头文件 AFXPRIV.H 和源文件 DCPREV.CPP。

绘画打印预览

我们探索打印预览的路径越来越清晰了。首先，我们看了 OnFilePrintPreview()中如何使用 CPreviewView，然后沿着这个方向看了函数 SetPrintView()。但是我们使用没有看到打印预览的绘画过程是如何完成的。

如果你回头看一下程序清单 8-9（CView::DoPrintPreview()），你会发现这个函数强制刷新了一个窗口，这个窗口中包含了一个新创建的 CPreviewView。和其他的 MFC 视图一样，这会导致调用 CPreviewView 的 OnDraw()函数。

CPreviewView::OnDraw()是一个复杂的函数，所以我们在这里就不显示它的代码了。我们只是推荐你打开 VIEWPREV.CPP 看一下它，以便继续讨论。首先，OnDraw()创建了两个画笔（pen）：rectPen——用于绘制表示页面的框；shadowPen——用于绘制页面周围的阴影（为了制造 3D 效果）。

然后，对于每个页面，它进入了一个 for 循环。在这个 for 循环中，执行的步骤如下：

1. 用 paint DC 绘制页面的轮廓和阴影。paint DC 是作为 OnDraw()的参数传入的。它会用一系列的 MoveTo()/LineTo()调用来绘制矩形。然后 for 循环调用 FillRect()在白色页面中画矩形。

2. 调用 OnDisplayPageNumber()显示页码。

3. 一旦画了新的页面，OnDraw()会调用 m_pPrintView->OnPrint()，参数是 CPreviewDC，这样就产生了打印预览输出。

在 for 循环迭代了所有预览的页面之后，OnDraw()会释放它所创建的画笔。

打印预览的封装

总而言之：

- 完成 MFC 打印预览的主要是两个类：CPreviewView 和 CPreviewDC。
- 为了解决打印预览问题，CView 将自己和 CPreviewView 交换，直至预览结束。
- 从此，CPreviewView 负责处理打印预览工具栏、缩放和打印预览的其他方面。
- CPreviewDC 提供了一种方法在屏幕上模拟打印机的输出。

更多的信息

我们可以继续一章又一章地讨论 CPreviewView 和 CPreviewDC 的细节，但是我们还要讨论 CView 中一些有趣的内容。对于那些想要继续探索的读者来说，下面是值得你考虑的一些问题：

- 看你是否能找到打印预览的缩放工作原理。(提示：从 CPreviewView::DoZoom()开始，看一下 SetScaledSize()。)
- 页码是如何画出的？哪个类画它？
- 为什么 CPreviewDC 需要知道 GDI 函数 DrawText()和 TextOut()的版本？
- 为了在屏幕上显示 4 个页面，你需要对 MFC 的打印预览做什么修改？

下面我们看 CView 的一些其他派生类，从而继续我们的 CView 高级内幕之旅。

CView 的派生类：CScrollView

你第一次看到 CScrollView 可能是在 MFC 的 Scribble 指南中。CView 的这个派生类使得在视图中增加滚动功能变得很容易。为了使用 CScrollView，你可以让你的应用程序视图从 CScrollView 中派生，然后调用 SetScrollSizes()，从而让 CScrollView 知道你的“逻辑”视图的大小。

CScrollView 获得了大小的信息，然后操纵传递给 OnDraw()的 DC，从而使得你的应用程序自动支持滚动。CScrollView 还有一个有趣的特性，它会自动重画你的视图，以便用户视图总能适合客户区域。MFC 称之为“scale to fit”模式。

一切看起来都很好，但是 CScrollView 是如何完成这些的呢？为了找到答案，我们首先看 CScrollView 如何运作。

注意：这是一个提高你的 Petzold GDI 映射模式和视口（viewport）概念的好机会，因为 CScrollView 经常用到这些东西。我们会假设你已经对这些 API 有所了解，对于那些比较难的，我们会在遇到的时候加以解释。

CScrollView 如何运作

为了指出 CScrollView 是如何实现的，我们先看 CScrollView 声明中的一些实现细节。它的声明在文件 AFXWIN.H 中，它的实现在文件 VIEWSCRL.CPP 中。程序清单 8-12 包含了它的一个缩减后的声明。

程序清单 8-12 CScrollView 的声明（在文件 VIEWSCRL.CPP 中）

```
class CScrollView : public CView
{
    DECLARE_DYNAMIC(CScrollView)
// Constructors ** omitted.
// Attributes **omitted.
// Operations **omitted.
// Implementation **some omitted
protected:
    int m_nMapMode;
    CSize m_totalLog;
    CSize m_totalDev;
    CSize m_pageDev;
    CSize m_lineDev;
    BOOL m_bCenter;
    void CenterOnPoint(CPoint ptCenter);
    void ScrollToDevicePosition(POINT ptDev);
protected:
    void UpdateBars();
    BOOL GetTrueClientSize(CSize& size, CSize& sizeSb);
    void GetScrollBarSizes(CSize& sizeSb);
    void GetScrollBarState(CSize sizeClient, CSize& needSb,
        CSize& sizeRange, CPoint& ptMove, BOOL bInsideClient);
public:
    virtual void CalcWindowRect(LPRECT lpClientRect,
        UINT nAdjustType = adjustBorder);
    virtual void OnPrepareDC(CDC* pDC, CPrintInfo* pInfo = NULL);
    virtual BOOL OnScroll(UINT nScrollCode, UINT nPos,
        BOOL bDoScroll = TRUE);
    virtual BOOL OnScrollBy(CSize sizeScroll, BOOL bDoScroll = TRUE);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg void OnHScroll(UINT nSBCode, UINT nPos,
        CScrollBar* pScrollBar);
    afx_msg void OnVScroll(UINT nSBCode, UINT nPos,
        CScrollBar* pScrollBar);
    DECLARE_MESSAGE_MAP()
};
```

下面是 CScrollView 实现的成员的一个纲要，按照它们在声明中出现的顺序列出：

CScrollView 实现中的数据成员：

- m_nMapMode——在 SetScrollSizes()中，你可以为应用程序指定一个映射模式。默认情况下是 MM_NONE。CScrollView 定义了一个“自定义”映射模式（MM_SCALETOFIT），我们后面会解释这个模式。用户可以指定任何映射模式，除了 MM_ISOTROPIC 和 ANISOTROPIC。这些映射模式很灵活，CScrollView 不可能使用所有模式。
- m_totalLog——逻辑坐标中视图的大小。这个值是通过调用 SetScrollSizes()成员函数传递给 CScrollView 的。
- m_totalDev——设备坐标中视图的大小。
- m_pageDev——设备坐标中一个页的大小。

- m_lineDev——设备坐标中一条线的大小。
- m_bCenter——CPreviewView 使用这个数据成员进入用户窗口中的视图。

CScrollView 实现中的成员函数：

- CenterOnPoint()——将视图集中于一点。这个函数由 CPreviewView 调用（见 m_bCenter）。
- ScrollToDevicePosition()——正如名字所暗示的那样，这个函数负责滚动视图。它通过调用::SetScrollPos()和::ScrollWindow()更新滚动条，从而完成视图的滚动。
- UpdateBars()——CScrollView 在初始化的时候和窗口大小发生改变的时候调用这个辅助函数。UpdateBars()的责任是根据 GetScrollBarState()的返回信息隐藏、显示和初始化滚动条。
- GetTrueClientSize()——这个成员函数用来确定用户视图是否足够大（大到需要使用滚动条）。只有 UpdateBars()才会调用 GetTrueClientSize()。
- GetScrollBarSizes()——确定滚动条的宽度和高度，考虑了窗口的风格和边框的宽度。GetScrollBarState 和 GetTrueClientSize()在计算过程中都需要使用滚动条的大小。
- GetScrollBarState()——获取 CScrollView 所需要的关于视图的状态。例如，它要计算视图是否需要滚动条以及滚动条的当前位置等等。
- CalcWindowRect()——计算窗口矩形的大小，考虑滚动条和其他窗口饰物的大小。
- OnPrepareDC()——CScrollView/CView 交互的关键。我们会在后面详细看这个函数。
- OnScroll()——OnHScroll()和 ONVScroll()这两个滚动条消息处理函数都调用了 OnScroll()，其中传入的信息来自滚动信息（scroll message）。OnScroll()会解码滚动信息，并根据页面大小和行的大小确定需要滚动的量。一旦需要滚动的量计算好了，OnScroll()会调用 OnScrollBy()。
- OnScrollBy()——这个成员函数会检查需要滚动的量是否超出了滚动条的范围和视图的逻辑大小。如果都没有，OnScrollBy()会调用::SetScrollPos()移动滚动条，然后调用::ScrollWindow()。
- OnSize()——如果 CScrollView 不是处于“scale-to-fit”模式下，OnSize()会调用 UpdateBars()。如果是处于“scale-to-fit”模式下，它会调用 SetScaleToFitSize()。
- OnHScroll()——调用 OnScroll()的消息处理函数。
- OnVScroll()——调用 OnScroll()的消息处理函数。

如果你仔细看前面这些内容，你可能已经发现在实现的这些辅助函数中有重复代码。ScrollToDevicePosition()和 OnScrollBy()最后都做了同样的事情：调用 SetScrollPos()和 ScrollWindow()进行滚动。为什么有重复的代码呢？

原因就是 OnScrollBy()是通过用户交互调用的，所以它执行了很多检查以确定所有输入都是合法的。而 ScrollToDevicePosition()是由开发人员通过调用 CScrollView 的 ScrollToPosition()成员函数而被调用的。在某些情况下，开发人员不希望进行 OnScrollBy()中的那些检查。为了满足这样的开发人员，MFC 就对同样的功能提供了两个函数。

让我们先看一下你在 CScrollView 派生类中初始化 CScrollView 时必须调用的成员函数：SetScrollSizes()。

CScrollView::SetScrollSizes()

程序清单 8-13 中是 CScrollView::SetScrollSizes()的伪代码，这些代码在文件 VIEWSCRL.CPP 中。

程序清单 8-13 CScrollView::SetScrollSizes()的伪代码（在文件 VIEWSCRL.CPP 中）

```
void CScrollView::SetScrollSizes(int nMapMode, SIZE sizeTotal, const SIZE&
     sizePage, const SIZE& sizeLine)
{
    int nOldMapMode = m_nMapMode;
    m_nMapMode = nMapMode;
    m_totalLog = sizeTotal;

    {
        CWindowDC dc(NULL);
        dc.SetMapMode(m_nMapMode);
        m_totalDev = m_totalLog;
        dc.LPtoDP((LPPOINT)&m_totalDev);
        m_pageDev = sizePage;
        dc.LPtoDP((LPPOINT)&m_pageDev);
        m_lineDev = sizeLine;
        dc.LPtoDP((LPPOINT)&m_lineDev);
        if (m_totalDev.cy < 0)
            m_totalDev.cy = -m_totalDev.cy;
        if (m_pageDev.cy < 0)
            m_pageDev.cy = -m_pageDev.cy;
        if (m_lineDev.cy < 0)
            m_lineDev.cy = -m_lineDev.cy;
    }
    if (m_pageDev.cx == 0)
    m_pageDev.cx = m_totalDev.cx / 10;
    if (m_pageDev.cy == 0)
        m_pageDev.cy = m_totalDev.cy / 10;
    if (m_lineDev.cx == 0)
        m_lineDev.cx = m_pageDev.cx / 10;
    if (m_lineDev.cy == 0)
        m_lineDev.cy = m_pageDev.cy / 10;
    if (m_hWnd != NULL){
        UpdateBars();
        if (nOldMapMode != m_nMapMode)
            Invalidate(TRUE);
    }
}
```

首先，SetScrollSizes()将 m_nMapMode 初始化为提供的新映射模式。另外，m_totalLog 的初始化值是 sizeTotal 参数。然后，SetScrollSizes()会在栈上创建一个 CWindowDC，并且在设置了映射模式之后，它使用 DC 来计算视图大小、页面大小和行大小的设备坐标。

在将逻辑坐标转化成设备坐标之后，SetScrollSizes()会检查，以确保用户提供了一个非默认的值。如果指定了默认值（0 是默认值），SetScrollSizes()会将页面大小设置为视图大小的 1/10，行的大小是页大小的 1/10 或视图大小的 1/100。

最后，如果需要，它会调用 UpdateBars()和 Invalidate()。UpdateBars()根据新的大小设置

滚动条，如果修改了映射模式，Invalidate 会导致重画。

对 DC 分块！

乍一看，在程序清单 8-13 中，在 CWindowDC 创建之前和设备计算之后的大括号（{、}）可能令你吃惊。微软为什么在这里用一个额外的大括号呢？答案与缺乏 Windows 资源（设备环境）有关。设备环境是一个代价很大的资源，MFC 要减小它的开销，所以它比默认情况提前几行释放了设备环境。

在栈上创建的 DC（类似 SetScrollSizes()中的 CWindowDC）外面增加两个括号会导致创建另一个堆栈框架。当编译器遇到这个嵌套的堆栈框架的末尾时，它会析构所有的堆栈变量，这样也就释放了前面创建的 DC。没有这些额外的大括号，DC 就只能在 SetScrollSizes()的末尾释放。这样加上对 UpdateBars()和 Invalidate()的调用就要花费很多时间。

我们得出的结论就是，如果你在类中使用了栈中创建的 DC，则应该将它们用大括号分块，从而减少运行时间。

在 CScrollView 中最常见到的代码块可能是：

```
//...
CWindowDC dc(NULL);
dc.SetMapMode(m_nMapMode);
dc.LPtoDP(...);
//...
```

因为你的 CScrollView 派生类可以在很多模式下绘制视图，所以 CScrollView 就必须仔细地将逻辑坐标转化成设备坐标，或者相反。在计算中使用 DC 是最好的办法。

正如前面提到的，CScrollView 和 CView 通过可覆盖的函数交互，这个函数就是 CScrollView::OnPrepareDC()。下面我们看这个可覆盖的函数。

CScrollView::OnPrepareDC()

在这一章的前面和前一章，我们都提到如果要定制用于显示和打印的设备环境，你要覆盖 CView::OnPrepareDC()。CScrollView 就是这样做的。下面我们看 CScrollView 的 OnPrepareDC()函数，看它对 DC 做了哪些处理。程序清单 8-14 中是相关代码，这些代码来自文件 VIEWSCRL.CPP。

程序清单 8-14 CScrollView::OnPrepareDC()的伪代码（来自文件 VIEWSCRL.CPP）

```
void CScrollView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    switch (m_nMapMode){
        case MM_SCALETOFIT:
            pDC->SetMapMode(MM_ANISOTROPIC);
            pDC->SetWindowExt(m_totalLog);  // window is in logical
                                            // coordinates
            pDC->SetViewportExt(m_totalDev);
            break;
        default:
            pDC->SetMapMode(m_nMapMode);
            break;
    }
```

```
    CPoint ptVpOrg(0, 0);       // assume no shift for printing
    if (!pDC->IsPrinting()) {
        ptVpOrg = -GetDeviceScrollPosition();
        // **omitted, some m_bCenter stuff.
    }
    pDC->SetViewportOrg(ptVpOrg);

    CView::OnPrepareDC(pDC, pInfo);     // For default Printing behavior
}
```

在我们解释 OnPrepareDC()的功能之前，要记住对它的调用都是在调用 OnDraw()之前，且传递给它的 DC 就是传递给 OnDraw()的。

首先，它设置了映射模式。如果 CScrollView 是在“scale-to-fit”模式下，OnPrepareDC()会将模式设置为 ANISOTROPIC，然后对窗口和视图端口操作，直至视图的大小和客户窗口匹配。这些映射模式的结果是，当你在 OnDraw()中绘画时，GDI 会自动缩放，使图像和客户窗口匹配。当 ScrollView 不是在 scale-to-fit 模式下，OnPrepareDC()会调用 SetMapMode()，参数是用户通过 SetScrollSizes()指定的映射模式。

然后在这段代码里，CScrollView 使用了一个小技巧，所以你要仔细看好了。OnPrepareDraw()在栈上创建了一个 CPoint（viewport origin 的缩写）。ptVpOrg 的初始值是 0,0。如果视图没有在打印，OnPrepareDC()会从视图的原点中减去当前滚动条的位置。然后 OnPrepareDC()会调用 CDC::SetViewportOrg()，参数是 ptVpOrg（如果我们正在打印，它的值就是 0,0），然后调用被覆盖了的 CView::OnPrepareDC()。

如果你对 GDI 不是很熟悉，这对你来说可能很神秘。基本上，OnPrepareDC()是对设备环境操作，你的 OnDraw()函数只是从滚动视图的某个起始位置开始绘画。这样 OnDraw()不知道自己所绘画的地方的偏移是多少。

例如，如果你调用 SetPixel(10,10)，并且滚动条位于(10,5)，那么这个点实际上画在了(0,5)。这个原始坐标要加到 GDI 坐标。在这个例子中就是：10+(−10)=0 和 10+(−5)=5。

CScrollView 总结

一旦你理解了 CScrollView 成员的用途，以及 SetScrollSizes()和 OnPrepareDC()的功能，你就知道 CScrollView 实现的 90%了。回忆起 SetScrollSizes()初始化了映射模式、页面大小和行大小。然后，如果需要，调用 UpdateBars()来创建并显示滚动条。还有 OnPrepareDC()会对 DC 的视口的原点进行操作，从而为实现视图的滚动做准备。实际的滚动是由 OnScrollBy()和 ScrollToDevicePosition()实现，这两个函数又调用了 ScrollWindow()。

更多信息

下面是关于 CScrollView 的更多信息：

- 从 OnSize()开始，看你是否能指出视图是如何重画的，使得在“scale-to-fit”模式下重画后恰好适合新的客户窗口大小。
- 根据 m_bCenter 在 OnPrepareDC()和 GetDeviceScrollPosition()中的用法，你能推断出 m_bCenter 的功能吗？
- 浏览 UpdateBars()。
  - 对 m_bInsideUpdate 的用法有什么猜测？

- 关于 WM_RECALCPARENT 消息的功能有什么想法？

既然我们已经了解了 CScrollView 的内部，让我们看它的一个派生类，CFormView。

CFormView

CFormView 使得你可以创建一个视图，这个视图由从对话框模板创建的控件组成。如果你有很多基于窗口的用户界面，这个类很有帮助。开发人员会从对话框模板和文档/视图结构的结合中获益。结果是得到了一个带有滚动条、支持 DDX/DDV 并且可以用于分割窗口（后面一章会讨论）的窗体。不幸的是，CFormView 不支持自动的打印和打印预览，因为它取决于开发人员认为窗体的打印版本是什么样。

使用 CFormView 和使用 CDialog 类似。首先你要使用 VC++的资源编辑器用控件创建一个对话框模板，然后创建一个 CFormView 的派生类，然后将它和对话框模板绑定（通过将模板的资源 ID 传递给 CFormView 构造函数）。

使用 CFormView 和使用 CView 的一个有趣的区别就是你不需要覆盖 OnDraw()。对话框模板中的控件只负责显示自己，不需要你去管理。

总而言之，CFormView 的大多数代码都是用控件执行 DDX/DDV。（我们在前面的第 6 章已经讲解了 DDX/DDV，如果需要你可以翻回去看看。）

CFormView 内幕

CFormView 的声明在文件 AFXEXT.H 中，实现在文件 VIEWFORM.CPP 中。它的声明没有在 CScrollView 中增加太多内容，只是覆盖了 CScrollView 的一些成员，并增加了一些新的数据成员。下面是新的数据成员和覆盖的一些有趣的内容：

- m_lpszTemplateName——对话框模板资源的名字。
- m_pCreateContext——指向 CCreateContext 的指针。CFormView 不直接用这个变量，但是会在 OnCreate()函数中将它传递给框架。
- m_hWndFocus——在 OnSetFocus()中，CFormView 将焦点设置为窗口句柄，除非它是非法的。OnActivateView()和 OnActivateFrame()都负责维护这个成员变量。
- OnDraw()——什么都不做。Windows 控件都会重画自己，所以这个函数什么都不用做。
- PreTranslateMessage()——CFormView 要完成一些消息路由的任务，将控件、视图和窗口的消息送往正确的目的地。所有这些都是在 PreTranslateMessage()中完成的。
- SaveFocusControl()——将 m_hWndFocus 设置为::GetFocus()的返回结果，也就是当前处于焦点的控件。
- OnActivateFrame()——当视图被禁用时，调用 SaveFocusControl()确保当焦点回来时，处于焦点的控件仍有控制。这些发生在函数 OnSetFocus()中。
- OnActivateView()——设置 m_hWndFocus 变量。
- OnCreate()——将 m_pCreateContext 传递给 LPCREATESTRUCT 结构指针的 lpCreateParams 域。
- OnSetFocus()——如果 m_hWndFocus 指向一个合法的窗口，OnSetFocus()就会将焦点设置为这个控件。如果 m_hWndFocus 指向的不是一个合法的控件，OnSetFocus()就将焦点设置为源窗口。

其中有一个函数很值得继续探索，那就是 CFormView::Create()。

CFormView::Create()内幕

CWnd::Create()这个覆盖函数完成了 CFormView 90%的工作。让我们看看 Create()如何创建并初始化 CFormView。程序清单 8-15 中是 CFormView::Create()的伪代码。

程序清单 8-15 CFormView::Create()的伪代码

```
BOOL CFormView::Create(LPCTSTR , LPCTSTR , DWORD dwRequestedStyle,
                       const RECT& rect, CWnd* pParentWnd,
                       UINT nID, CCreateContext* pContext)
{
    m_pCreateContext = pContext;
    VERIFY(AfxDeferRegisterClass(AFX_WNDCOMMCTLS_REG));
    CREATESTRUCT cs;
    memset(&cs, 0, sizeof(CREATESTRUCT));
    if (dwRequestedStyle == 0)
        dwRequestedStyle = AFX_WS_DEFAULT_VIEW;
    cs.style = dwRequestedStyle;
    if (!PreCreateWindow(cs))
        return FALSE;
    if (!CreateDlg(m_lpszTemplateName, pParentWnd))
        return FALSE;
    m_pCreateContext = NULL;
    ModifyStyle(WS_BORDER|WS_CAPTION, dwRequestedStyle &
      (WS_BORDER|WS_CAPTION));
    ModifyStyleEx(WS_EX_CLIENTEDGE, cs.dwExStyle & WS_EX_CLIENTEDGE);
    SetDlgCtrlID(nID);
    CRect rectTemplate;
    GetWindowRect(rectTemplate);
    SetScrollSizes(MM_TEXT, rectTemplate.Size());
    if (!ExecuteDlgInit(m_lpszTemplateName))
        return FALSE;
    SetWindowPos(NULL, rect.left, rect.top, rect.right - rect.left,
            rect.bottom - rect.top, SWP_NOZORDER|SWP_NOACTIVATE);
    if (dwRequestedStyle & WS_VISIBLE)
        ShowWindow(SW_NORMAL);
    return TRUE;
}
```

首先，Create()在 m_pCreateContext 中存储了 CCreateContext 参数，这样就可以在调用 OnCreate()时传入。然后 Create()确保 Windows 通用控件都是注册过的。

在后面一块代码里（从 CREATESTRUCT 到 PreCreateWindow()调用），Create()调用了 PreCreateWindow 来确定用户指定的扩展风格（存储在 cs.dwExStyle 中）。在确定了扩展风格后，Create()会调用 CWnd::CreateDlg()。CreateDlg()会从应用程序的资源中装载对话框模板，并调用 CWnd::CreateDlgIndirect()（关于这个函数的细节请参考第 5 章）。CreateDlg()会创建一个非模态的对话框。

在调用了 CreateDlg()之后，Create 将 m_pCreateContext 设置为 NULL，因为后面不再需要它了。然后它修改了窗口的标准风格和扩展风格。对于标准风格而言，它确保了边界和提示与 dwRequestedStyle 中要求的一致。对于扩展风格而言，它确保 WS_EX_CLIENTEDGE 和传递给 PreCreateWindow()的 CREATESTRUCT 结构中指定的一致。然后，Create()将窗口

控件的标识符设置为 nID 参数。然后调用 SetScrollSizes()来初始化 ScrollView，所采用的映射模式是 MM_TEXT，这样就创建了一个和对话框模板一样大的逻辑视图。

除了从应用程序资源中装载对话框资源，Create()还调用了 CWnd::CreateDlg()。CreateDlg()是 CWnd 的一个成员函数，这个成员函数调用 CreateDlgIndirect 创建了一个非模态的对话框。既然控件已经创建，滚动条也已经在适当的位置，如果资源文件中指定了某些初始值，Create()会调用 ExecuteDlgInit()（第 5 章中有这个函数）初始化窗体中的控件。

最后，Create()强制要求视图的大小和 rect 参数的值匹配，然后如果在 PreCreateWindow()调用中指定了 WS_VISIBLE，就调用 ShowWindow()。一旦创建了 CFormView，惟一有趣的实现细节就是 DDX/DDV，这个我们在前面已经讨论过。接下来 CFormView::OnInitialUpdate()调用了 UpdateData(FALSE)来初始化窗体中的控件。从此之后，DDX/DDV 会认为 CFormView 就是一个对话框。

CFormView 总结

CFormView 类是 CScrollView 的一个很小的派生类，它从对话框资源模板创建了一个非模态的对话框，并对对话框提供了一个 CView 接口。

在我们即将看到的 CView 派生类的下一个层次中，概念都是类似的，但是它没有为对话框提供一个 CView 接口，而是为单个控件提供了 CView 接口。

CView 的另一个派生类：CCtrlView

CCtrlView 这个 CView 的派生类可以使一个 Windows 控件成为一个视图。你的应用程序不能以 CCtrlView 为基类，但是你可以使用它的派生类：CEditView、CListView、CTreeView 和 CRichEditView。

在这一部分，我们首先看 CCtrlView 如何实现控件到视图的映射，然后看 CCtrlView 的一个派生类的内部，看它如何利用 CCtrlView 接口运作。

版本注释

CCtrlView 是在 MFC 4.0 中发布的。在 MFC 4.0 之前，惟一的控件视图就是 CEditView，它是从 CView 派生的。正是因为增加了三个新的 CView 控件类，所以才要创建一个新的基类，用它定义 CView 到控件的接口。在 MFC 4.0 中，CEditView 的派生类直接从 CView 改成了 CCtrlView。

CCtrlView 如何运作

CCtrlView 的声明在文件 AFXWIN.H 中，它的实现在文件 VIEWCORE.CPP 中。它的声明很短，所以我们只是把它的声明列在程序清单 8-16 中，而不加以详细讨论。

程序清单 8-16 CCtrlView 的声明（在文件 AFXWIN.H 中）

```
class CCtrlView : public CView
{
    DECLARE_DYNCREATE(CCtrlView)
public:
    CCtrlView(LPCTSTR lpszClass, DWORD dwStyle);
```

```
// Attributes
protected:
    CString m_strClass;
    DWORD m_dwDefaultStyle;
// Overrides
    virtual void OnDraw(CDC*);
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
// Implementation ** some debug stuff omitted
protected:
    afx_msg void OnPaint();
    DECLARE_MESSAGE_MAP()

};
```

CCtrlView 确实没有做什么，它的主要目标是为它的派生类提供一个接口。记住了这些，下面就是每个成员的纲要：

- m_strClass——这个 CString 成员变量包含了控件的窗口类的名字，通常是“view-ized”。m_strClass 是在 CCtrlView 的构造函数中设置的，通过构造函数的参数传入。
- m_dwDefaultStyle——视图类的默认风格。这个成员数据也是通过 CCtrlView 构造函数的一个参数传入的。
- CCtrlView()——构造函数，负责设置 m_strClass 和 m_dwDefaultStyle。
- OnDraw()——从来不被调用！它调用了 ASSERT(FALSE)来确保不被调用。见 OnPaint()。
- PreCreateWindow()——设置 CREATESTRUCT 的 lpszClass 域，设置成 m_strClass 成员数据，这样就创建了控件。它还对 CREATESTRUCT 的风格域操作，以反应存储在 m_dwDefaultStyle 中的值。
- OnPaint()——调用 Default()。Default()会将前一个消息（WM_PAINT）发送给 DefWindowProc()。CCtrlView 这样做是因为控件可以自己重画，而不需要其他帮助。

好了，我们已经发现 CCtrlView 不像我们所想像的那么有趣。CCtrlView 中最有趣的部分就是它的派生类对这些接口的实现。

下面我们就看一个 CCtrlView 的新派生类：CTreeView。

CTreeView：控件视图的例子/CCtrlView 的派生类

CTreeView 的声明在文件 AFXCVIEW.H 中，它的实现在文件 VIEWCMN.CPP 和 AFXCVIEW.INL 中。程序清单 8-17 中列出了 CTreeView 的声明。

程序清单 8-17 CTreeView 的声明（在文件 AFXCVIEW.H 中）

```
class CTreeView : public CCtrlView
{
    DECLARE_DYNCREATE(CTreeView)
// Construction
public:
    CTreeView();
// Attributes
public:
    CTreeCtrl& GetTreeCtrl() const;
```

```
protected:
    void RemoveImageList(int nImageList);
public:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    afx_msg void OnDestroy();
    DECLARE_MESSAGE_MAP()
};
```

CTreeView 基本上是为我们前面讨论的 CCtrlView 的成员提供实现，并增加一些新函数。这些新函数包括：

- GetTreeCtrl()——返回一个 CTreeCtrl 引用，类的使用者可以用这个引用直接调用 CTreeCtrl，对树控件操作。
- RemoveImageList()——一个内部辅助函数，它将控件的图像链表清空。这个辅助函数会在 OnDestroy()中被调用，在 OnDestroy()函数中还会清空 LSVIL_NORMAL 和 LVSIL_STATE 两个图像链表。这两个图像链表分别是大图标的链表和状态对象的链表。

CTreeView::CTreeView()

下面是 CTreeView()的构造函数的实现，在文件 AFXCVIEW.INL 中。

```
CTreeView::CTreeView() :
    CCtrlView(WC_TREEVIEW, AFX_WS_DEFAULT_VIEW) { }
```

构造函数所做的就是将类的名字传递给基类 CCtrlView，而类的名字被#define 为 WC_TREEVIEW。CTreeView()还将窗口的默认风格位（AFX_WS_DEFAULT_VIEW）传递给基类。

CTreeView::PreCreateWindow()

PreCreateWindow()的任务就是确保 Windows 普通控件在调用 CCtrlView::PreCreateWindow()之前都已经初始化了。

CTreeView::GetTreeCtrl()

最后，GetTreeCtrl()的实现如下：

```
CTreeCtrl& CTreeView::GetTreeCtrl() const
{
    return *(CTreeCtrl*)this;
}
```

为什么要这样做？它这样做是因为 CTreeCtrl 和 CWnd 有类似的二进制表示，因为 CTreeView 是从 CWnd 派生的。换句话说，一个 CTreeCtrl 只不过是一个 CWnd（从二进制的角度看），当然，CWnd 也可以是一个 CTreeCtrl。

CCtrlView 的包装

CCtrlView 的派生类很小，尤其是 CListView 和 CTreeView。CRichEditView 有一些内幕很有趣，这些内幕是关于对 OLE 的支持的，但是这些在本章讨论范围之外。CEditView 还有一些其他有趣的内幕，但是其中大部分来自传统的编辑控件的限制。

更多信息

如果你对探索 CCtrlView 的其他派生类感兴趣，可以查看表 8-1 所示的类的声明和实现。

表 8-1 其他的 CCtrlView 派生类所在文件

	声明	实现
CEditView	AFXEXT.H	AFXEXT.INL/VIEWEDIT.CPP
CListView	AFXCVIEW.H	AFXCVIEW.INL/VIEWCMN.CPP
CRichEditView	AFXRICH.H	AFXRICH.INL/VIEWRICH.CPP

下面是给你的一些问题：

- CEditView——这个 CCtrlView 的派生类如何支持打印？
- CEditView 是可序列化的。但是为什么你要序列化一个视图呢？
- CListView——它如何支持自画的链表视图呢？
- CRichEditView——探索 CRichEditDoc 类和 CRichEditCntrItem 类，你能指出它们是做什么的吗？为什么需要它们？
- 没有文档说明的 CReObject 的功能是什么？

结 语

我们已经学习了 MFC 的文档/视图结构的一些基本知识和一些高级内容，下面让我们看看 MFC 的一些已增强的用户界面类，例如控制栏和分割窗口。分隔窗口依赖于这里提到的很多 CView 的细节，所以准备迎接对 CView 的考试吧。

CHAPTER

9 MFC 的增强型用户界面类

到目前为止，我们已经学习了几种不同的 MFC 用户界面类。除了 Windows 的包装类（例如 CWnd、CDialog 以及所有的控件类）外，还有一种并没有对已有的 Windows 对象进行包装的用户界面类。我们将它们称为增强型用户界面类。既然上一章的 CView 对你还是记忆犹新（希望如此！），那么在本章中让我们从分割窗口（splitter window）开始。分割窗口与视图（view）的联系非常紧密。

下面，让我们看一下 CControlBar 类家族，它包括对话框条、工具栏、以及状态栏。我们也一定会向读者阐明 MFC 灵巧的可停放工具栏（dockable toolbar）是如何工作的。在本章的最后我们将讲述 MFC 的 CMiniFrame、CFrameWnd 等派生类。

因为这些类的实现绝大多数全部在 MFC 中（只有 MFC 4.0 中的 CToolBar 和 CStatusBar 是个例外），所以有许多非常有趣的内容可以让你来探索。首先让我们来看看 CSplitterWnd 是如何实现的。

CSplitterWnd：MFC 分割窗口

CSplitterWnd 是所有的 MFC 类中最复杂、因此也是最让人糊涂的类之一。CSplitterWnd 之所以复杂，是因为它要提供非常高级的界面。读完本节后，你应该能明白 CSplitterWnd 到底是如何工作的，这样，你就能在 MFC 应用程序中比较轻松地使用它。

如果你曾经使用过比较流行的微软窗口产品，可能已经遇到过某些类型的分割窗口。静态分割窗口能将窗口逻辑地分成两个可调整大小的区域。Windows 95 的浏览器和 Visual C++的位图编辑器就是静态分割窗口的两个例子。动态分割窗口可以让用户将窗口任意地分割成多个视图，而且都有相同的数据。Visual C++代码编辑器和微软的 Word 使用了动态分割技术。

在深入研究 CSplitterWnd 之前，首先让我们来分割一个窗口。

剖析分割窗口

图 9-1 显示了分割窗口的主要部分。

分割窗口就像现实中的窗口一样，有窗格（pane）。在下图中仅有 2 个窗格，但在 MFC 中，可以有 2 个或者 4 个窗格。窗格通常是 CView，但 CSplitterWnd 非常灵活，所以允许窗

格可以是任何 CWnd 的派生类。

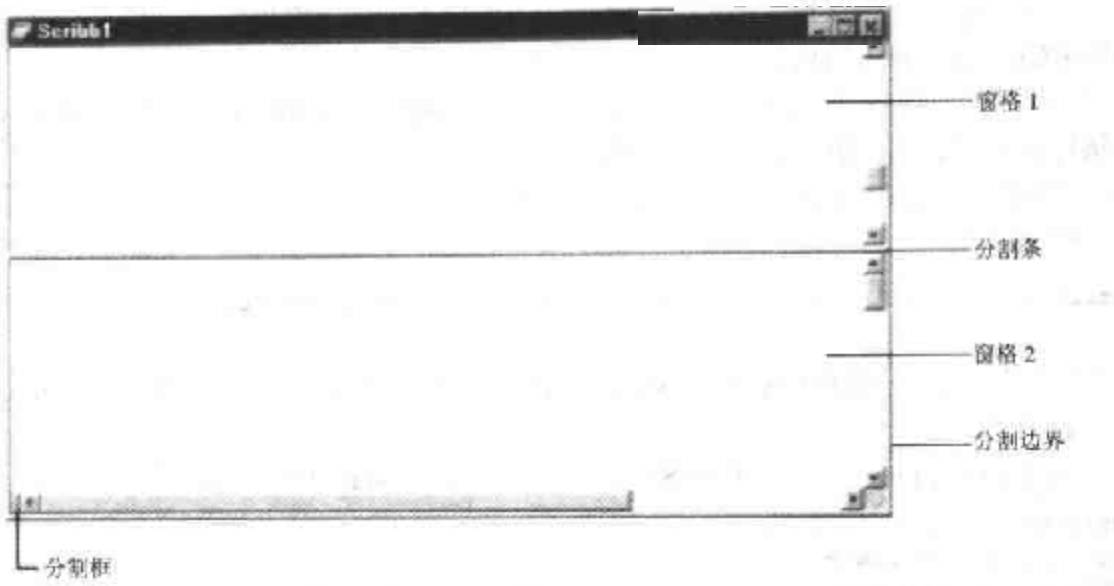


图 9-1 分割窗口组件

用以下的方法可以创建新的窗格："抓住"分割框（splitter box）（见图 9-1），将它拖拉到你所希望分割条（splitter bar）所在的位置。当有 4 个窗格时，即有 2 个分割条时，2 个分割条相交的点称为分割交点（splitter intersection）（图中没有画出）。

当我们学习 MFC 是如何画出所有这些分割条组件时，我们就会探究神秘的分割边界（splitter border）了。

分割窗口在内部用行和列的形式保存这些窗格。例如，图 9-1 中的窗格是两行；如果分割条是竖直的，那么将有两列。最后，如果既有水平的划分，也有竖直的划分，那么将有两行和两列。

因为分割窗口将视图当作窗格，所以当用户修改文档数据时，CDocument::UpdateAllViews()机制会被调用，从而导致每个窗格都被自动更新。如果你将 CWnd 的派生类当作窗口，那么同步更新的任务将是你的了。

最后，注意分割窗口都有它们自己的滚动条。因为滚动条经常会同时影响两个窗格，所以分割窗口需要有它们自己的滚动条逻辑。例如，在图 9-1 中，底部的水平滚动条会同时影响窗格 1 和窗格 2。

复习：如何使用 CSplitterWnd

正如前面提到的，有两种方法可以使用 CSplitterWnd：动态的和静态的。

在你的 MFC 应用程序中创建动态分割窗口，需要遵从下面的步骤：

1. 在子框架的派生类中加上一个 CSplitterWnd 数据成员（你使用的是 MDI 还是 SDI 会影响这一步）。例如：

```
CSplitterWnd  m_wndSplitter;
```

2. 在 CMyChildFrame::OnCreateClient()处理程序，添加对 CSplitterWnd::Create()的调用。

Create()带有好几个参数。第一个参数是指向父框架的指针（通常是 this）。第二个和第三个参数指定了最大行数和最大列数。第四个参数指定了所允许的最小窗格大小。最后，第五个参数是指向 CCreateContext 的指针。

下面的例子创建了一个分割窗口，它最多有两行、两列。最小的窗格大小是 10×10，并且环境指针直接被传递过去，没有任何修改。

```
BOOL CChildFrame::OnCreateClient(LPCREATESTRUCT , CCreateContext*
    pContext)
{
return m_wndSplitter.Create(this,2, 2, CSize(10, 10), pContext);
}
```

在 MFC 里，动态分割窗口被限制成只能有两行、两列，所以行和列参数经常被指定为 1 来限制行数和列数。

3. 将框架窗口的 RUNTIME_CLASS 信息传递给 CDocTemplate 构造函数：

```
CMultiDocTemplate * pDocTemplate =
new CMultiDocTemplate(IDR_MYAPPTYPE,
RUNTIME_CLASS(CMyDocClass), RUNTIME_CLASS(CMyChildFrameClass),
RUNTIME_CLASS(CMyViewClass));
AddDocTemplate(pDocTemplate);
```

这就是在 MFC 应用程序中添加动态分割窗口所要做的全部事情！当用户对窗口进行分割后，MFC 会自动用窗格（视图）来填充分割部分。在文档模板里（在这个例子中是 CMyViewClass），这些窗格和分割窗口被紧密地联系在一起。

要创建静态分割窗口，可以用 CreateStatic()来代替 Create()。为每个静态窗口创建视图也是开发人员的任务，因为 MFC 并不知道要创建哪种类型的视图。你可以调用 CSplitterWnd::CreateView()来创建新的窗格。

静态分割窗口并没有动态分割窗口 2×2 的限制。你也许正在这样问自己："既然 CSplitterWnd 可以静态地画出任意多的行和列，那么为什么它不能动态地做这些呢？"问题出来了！当我们开始探究 CSplitterWnd 内部时，我们将进一步讨论这个问题。

下面是创建一个 4×4 静态分割窗口的例子：

```
BOOL CChildFrame::OnCreateClient(LPCREATESTRUCT , CCreateContext* pContext)
{
    int nRow;
    int nCol;
    if (!m_wndSplitter.CreateStatic(this,4,4))
        return FALSE;
    for (nRow = 0; nRow < 4; nRow++)
        for (nCol = 0; nCol < 4; nCol++)
            m_wndSplitter.CreateView(nRow,nCol,RUNTIME_CLASS(CMyView),
              CSize(10,10), pContext);
    return TRUE;
}
```

一旦视图被创建了，你就可以调用 CSplitterWnd 的成员函数来指定静态分割部分的大小，例如 SetColumnInfo()、SetRowInfo()。

已经有足够多的概述了！现在让我们来看看 MFC 到底是如何实现这个实在是非常灵活的用户界面小配件。

CSplitterWnd 的内部实现

回想一下前一节所学的内容，你也许会意识到 CSplitterWnd 其实是最容易使用的 MFC 类之一。即使是对话框和控件，你也要创建一些资源或者类似的东西。作为一个熟悉 MFC 内部实现的成员（在第 9 章你已经做到了这一点！），你会意识到下面的事实：CSplitterWnd 肯定在后台做了大量的工作才取得界面的简单性。

在 CSplitterWnd 内部有许多非常有意思的实现，所以我们将采取和其他 MFC 类稍微不同的方法来探究 CSplitterWnd。通常，我们首先一头扎进程序清单，并从主要的函数开始跟踪。对于 CSplitterWnd，我们将采取从顶至下的方法，并看看 MFC 中 CSplitterWnd 实现的以下部分：

- 声明——我们将从 CSplitterWnd 所特有的声明部分开始。
- 初始化——下面，我们将依次看看 Create()、CreateStatic()以及 CreateView()成员函数，从而探究当你的 MFC 应用程序在调用这些函数时，到底发生了哪些事情。
- 窗格的管理——在明白 CSplitterWnd 是如何进行初始化后，我们将进一步考查它是如何管理窗格的。
- 画图——下面，我们将更深一步地研究 MFC 到底是如何为分割窗口画出那些组成部分的，包括分割框、分割条、分割交点以及其他所有的分割窗口组成部分。
- 点击测试/跟踪——在明白了分割窗口是如何被画出来后，我们将看看另外一件有趣的实现细节：点击测试和跟踪。CSplitterWnd 通过点击测试来推断出用户正在与分割窗口的哪个部分进行交互。在许多 MFC 的窗口中你都可以使用点击测试技术。当用户“抓住”分割框、并将分割框拖拉到所希望到达的地方时，跟踪技术就被用来实现用户界面了。
- 实现分割——最后，我们将实际分割一个框架，将以上学到的所有内容组合到一起，并看看 CSplitterWnd 的所有特性是如何一起工作的。

贯穿整个对 CSplitterWnd 的讨论，我们并没有对静态分割和动态分割做很大的区分，因为实现它们的代码有 99%是一样的。当有什么东西对于动态分割或者静态分割来说比较特殊时，我们一定会指出来。

CSplitterWnd 声明

CSplitterWnd 在 AFXEXT.H 文件里声明。程序清单 9-1 列出了在声明中与实现相关的部分。

程序清单 9-1 缩减后的 CSplitterWnd 声明（在文件 AFXEXT.H 中）

```
class CSplitterWnd : public CWnd
{
    DECLARE_DYNAMIC(CSplitterWnd)
// Construction ** omitted.
// Attributes **omitted.
// Operations
public:
    virtual void RecalcLayout();    // call after changing sizes
// Overridables **some omitted.
protected:
```

```
    enum ESplitType { splitBox, splitBar, splitIntersection, splitBorder };
    virtual void OnDrawSplitter(CDC* pDC, ESplitType nType,
        const CRect& rect);
    virtual void OnInvertTracker(const CRect& rect);
public:
    virtual BOOL CreateScrollBarCtrl(DWORD dwStyle, UINT nID);
public:
// high level command operations - called by default view implementation
    virtual BOOL CanActivateNext(BOOL bPrev = FALSE);
    virtual void ActivateNext(BOOL bPrev = FALSE);
    virtual BOOL DoKeyboardSplit();
// synchronized scrolling
    virtual BOOL DoScroll(CView* pViewFrom, UINT nScrollCode,
        BOOL bDoScroll = TRUE);
    virtual BOOL DoScrollBy(CView* pViewFrom, CSize sizeScroll,
        BOOL bDoScroll = TRUE);
// Implementation  **some omitted for brevity
public:
    struct CRowColInfo {
        int nMinSize;       // below that try not to show
        int nIdealSize;     // user set size
        int nCurSize;       // 0 => invisible, -1 => nonexistent
    };
protected:
    CRuntimeClass* m_pDynamicViewClass;
    int m_nMaxRows, m_nMaxCols;
// implementation attributes which control layout of the splitter
    int m_cxSplitter, m_cySplitter;         // size of splitter bar
// space on either side of splitter
    int m_cxBorderShare, m_cyBorderShare;
// amount of space between panes
    int m_cxSplitterGap, m_cySplitterGap;
    int m_cxBorder, m_cyBorder;             // borders in client area
    // current state information
    int m_nRows, m_nCols;
    BOOL m_bHasHScroll, m_bHasVScroll;
    CRowColInfo* m_pColInfo;
    CRowColInfo* m_pRowInfo;
// Tracking info - only valid when 'm_bTracking' is set
    BOOL m_bTracking, m_bTracking2;
    CPoint m_ptTrackOffset;
    CRect m_rectLimit;
    CRect m_rectTracker, m_rectTracker2;
    int m_htTrack;
// implementation routines
    BOOL CreateCommon(CWnd* pParentWnd, SIZE sizeMin, DWORD dwStyle,
        UINT nID);
    int HitTest(CPoint pt) const;
    void GetInsideRect(CRect& rect) const;
    void GetHitRect(int ht, CRect& rect);
    void TrackRowSize(int y, int row);
    void TrackColumnSize(int x, int col);
    void DrawAllSplitBars(CDC* pDC, int cxInside, int cyInside);
    void SetSplitCursor(int ht);
```

```
    CWnd* GetSizingParent();
// starting and stopping tracking
    virtual void StartTracking(int ht);
    virtual void StopTracking(BOOL bAccept);
    afx_msg BOOL OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT message);
    afx_msg void OnMouseMove(UINT nFlags, CPoint pt);
    afx_msg void OnPaint();
    afx_msg void OnLButtonDown(UINT nFlags, CPoint pt);
    afx_msg void OnLButtonDblClk(UINT nFlags, CPoint pt);
    afx_msg void OnLButtonUp(UINT nFlags, CPoint pt);
    afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg void OnHScroll(UINT nSBCode, UINT nPos,
        CScrollBar* pScrollBar);
    afx_msg void OnVScroll(UINT nSBCode, UINT nPos,
        CScrollBar* pScrollBar);
    afx_msg BOOL OnNcCreate(LPCREATESTRUCT lpcs);
    afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx_msg void OnDisplayChange();
    DECLARE_MESSAGE_MAP()
};
```

你会注意到在程序清单 9-1 中，我们在“//Implementation”部分上面的声明里放置了许多成员。尽管这些成员有文档说明，但很好地理解它们对学习 CSplitterWnd 的实现有很大的帮助。MFC 在 CSplitterWnd 的“虚拟化”上做了很多工作，使得用户可以创建 CSplitterWnd 的派生类，并改变 CSplitterWnd 的默认行为。在你对 CSplitterWnd 的实现了解更多后，实现自己的派生类会更加容易。

下面是 CSplitterWnd 中比较有意思的成员链表。这些成员中有一些有文档说明，但有一些没有。当我们开始仔细研究 CSplitterWnd 成员函数时，还会再次看到许多这些成员的。

封装的数据类型

- ESplitType——定义要画出的分割器的类型，属于枚举类型。有效的类型有分割框、分割条、分割交点以及分割边界（回顾图 9-1 中列出的分割窗口的各个组成部分）。
- CRowColInfo——记录行或列的最小尺寸、理想尺寸以及当前的尺寸，属于内部结构。当描述的是行时，这些值记录的是行的高度；当描述的是列时，这些值记录的是列的宽度。

创建/布局数据成员

这些数据成员记录了创建分割窗口的窗格所需的信息，以及窗格布局的信息。

- m_pDynamicViewClass——指向由 CSplitterWnd 动态创建的视图（窗格）的 CRuntimeClass 信息的指针。对于动态分割窗口，该成员是在 Create()里定义的；对于静态分割窗口，该成员是在 CreateView()里定义的。
- m_nMaxRows/m_nMaxCols——在调用 Create()或 CreateStatic()时指定的最大行数和列数。
- m_nRows/m_nCols——当前在 CSplitterWnd 里显示的行数和列数。
- m_bHasHScroll/m_bHasVScroll——表明行滚动条或列滚动条是否已经被创建的标记。

- m_pColInfo——是 CRowInfo 的数组，每个元素对应 CSplitterWnd 的一列。在静态分割窗口里，这个值是固定的；而在动态分割窗口里，这个值是变化的。
- m_pRowInfo——是 CRowColInfo 的数组，每个元素对应 CSplitterWnd 的一行。在静态分割窗口里，这个值是固定的；而在动态分割窗口里，这个值是变化的。

修饰的数据成员

这些比较“神秘”的数据成员（又称为“捏造因素”）都是在 CSplitterWnd 的构造函数里初始化的。因为它们没有相应的文档，所以我们不得不假设它们的默认值在绝大多数分割窗口的绘画中都是好的。如果出于兴趣，你也可以创建 CSplitterWnd 的派生类，并在派生类中修改这些值来看看会有什么发生。

- m_cxSplitter/m_cySplitter——分割框和分割器的宽度和高度。在 Windows 95 下这个值是 7，在 Windows NT 下这个值是 4。
- m_cxBorderShare/m_cyBorderShare——如果分割窗口正在画分割窗口的边界，则这些值为 0（在 Windows 95 下），或者为 1（在 Windows NT 下）。
- m_cxSplitterGap/m_cySplitterGap——分割框/分割条和滚动条/边界之间的距离。在 Windows NT 里，这个值是 6。在 Windows 95 里，这个值是 7。
- m_cxBorder/m_cyBorder——分割边界的边界宽度。在 NT 里这个值是 0，在 Windows 95 里是 2。

跟踪数据成员

下面的数据成员用于点击测试和跟踪。

- m_bTracking——如果为真，则用户正在拖动一个分割条。
- m_bTracking2——如果为真，则用户正在拖动两个分割条（m_bTracking 也为真）。什么时候用户能同时拖动两个分割条呢？当用户拖动分割交点时。
- m_ptTrackOffset——点击测试中的“选取”尺寸。也就是说，点击测试并不是局限于分割框、分割条的精确宽度和精确高度。m_ptTrackOffset 允许用户有所偏差。
- m_rectLimit——跟踪时窗格的大小，用来确定被跟踪的分割条的高度。
- m_rectTracker——跟踪时用来画分割条的矩形。
- m_rectTracker2——跟踪时用来画第二个分割条的矩形（如果用户要拖动两个分割条，则要拖动它们的交点）。
- m_htTrack——被 CSplitterWnd 的点击跟踪机制设置成一个枚举值，这个枚举值用来描述分割窗口的哪个部分被点击了（在以后的章节中将详细介绍）。

通用成员函数

- CreateCommon()——当 Create()和 CreateStatic()初始化完 CSplitterWnd 的动态成员或静态成员时会调用该函数。我们将在后面的章节中详细介绍。
- CreateScrollBarCtrl()——创建带有指定风格和标识符的滚动条。
- DoScroll()——对滚动条消息做出反应。DoScroll()能够同步适当的窗格。
- DoScrollBy()——以指定的数量滚动相应的窗格。
- DoKeyboardSplit()——在程序里调用该函数会使得窗口被分割。例如，在 Visual C++ 里，你可以从菜单里选择 Window/Split 来分割编辑窗口。
- CanActivateNext()——调用该函数来确定下一个窗格是否能被激活。也就是说，它是

否能得到焦点。该函数被 CView 类调用（将在后面的章节详细介绍）。

- ActivateNext()——用来激活下一个窗格。通常在一个窗格被删除时调用，这样焦点就能在该窗格消失之前被改变。

布局成员函数

- RecalcLayout()——这是个很重要的函数，它会维护所有分割窗口的位置。当一个窗格被创建时，RecalcLayout()会被调用；当窗格被删除时，RecalcLayout()也会被调用。我们将在后面的章节详细介绍 RecalcLayout。
- TrackRowSize()——更新指定行的 m_pRowInfo 数组信息。同时确定是否有足够的空间来存储该行。如果没有足够的空间，TrackRowSize()会将它删除（只在动态分割窗口里）。
- TrackColumnSize()——和上面的相同，但是针对列的。
- GetSizingParent()——搜索大小可变的父窗口（仅仅在 Windows 95 里）。

绘画成员函数

- DrawAllSplitBars()——该函数“驱动”分割窗口的绘画进程，方法是为分割窗口中每个需要绘画的组件调用 OnDrawSplitter。
- OnDrawSplitter()——OnDrawSplitter()为分割窗口的各个组件进行绘画。OnDrawSplitter()是虚函数，所以如果你愿意，也可以修改分割窗口的外观。
- OnPaint()——对 WM_PAINT 消息做出响应。

点击测试成员函数

- HitTest()——选取某个点，然后返回这个点的点击测试值。
- GetInsideRect()——该成员函数与 GetClientRect()比较类似，只是它会考虑共享的滚动条。
- GetHitRect()——为某一指定的分割窗口组件检索点击矩形。
- SetSplitCursor()——使用点击测试来确定要显示哪一种光标。如果鼠标移过分割框或者分割条，SetSplitCursor()会显示改变大小的箭头，如果鼠标移过交点，SetSplitCursor()会显示 4 个方向的箭头，依此类推。

跟踪成员函数

- StartTracking()——基于点击测试值开始跟踪操作。该函数由 DoKeyboardSplit()和 OnLButtonDown()调用。
- StopTracking()——停止一个跟踪操作。当 OnLButtonDblClk()、OnLButtonUp()以及其他函数由于某种原因要停止跟踪时就会调用该函数。
- OnInvertTracker()——在改变边框大小的过程中反转跟踪器。

消息处理程序

- OnNcCreate()——CSplitterWnd 处理 WM_NCCREATE 消息，所以它可以移走 WS_EX_CLIENTEDGE 的扩展风格位（extended style bit）。CSplitterWnd 自己画维边界，这样分割器才会看上去是边界的一部分。
- OnPaint()——画分割窗口的各个组件。
- OnDisplayChange()——当用户改变显示器的分辨率时该函数被调用。OnDisplayChange()为新的设置调用 RecalcLayout()来更新分割窗口。

- OnSize()——当用户改变窗口大小时调用 RecalcLayout()。
- OnMouseMove()——在分割窗口组件上执行点击测试。
- OnLButtonDown()——同样在分割窗口组件上执行点击测试，而且当用户点击分割窗口的某个组件时启动跟踪。
- OnLButtonDblClk()——对双击进行点击测试，而且当用户双击分割框时，它会自动将窗口分成两半。如果用户双击分割条 3 分割就会被取消。
- OnLButtonUp()——停止跟踪。
- OnKeyDown()——提供某些键序列来控制分割窗口的跟踪。

这里有太多的成员函数，你不可能一下子就全部掌握它们。现在，你只需尽力熟悉它们就可以了。当我们学习特定的 CSplitterWnd 成员函数时，你可能会回到本章节来看看到底这些成员函数的功能是什么。

现在让我们看看分割窗口是如何初始化的。注意，所有的 CSplitterWnd 都保存在 WINSPLIT.CPPMFC 源文件中。你可以用你最喜欢的代码编辑器来打开这个文件，并做你喜欢做的事。

CSplitterWnd 的初始化

CSplitterWnd 有点儿让人烦。它将 m_cxSplitter、m_cySplitter、m_cxBorderShare、m_cyBorderShare、m_cxSplitterGap、m_cySplitterGap、m_cxBorder 以及 m_cyBorder 初始化为我们前面提到的值。让我们跳过它们，直接来看看 Create()和 CreateStatic()成员函数。

Create()和 CreateStatic()都会最终调用 CreateCommon()，所以我们也会研究这个函数。最后，我们会看看窗格是如何通过 CreateView()成员函数来创建的。

记住，在概述里我们说过 Create()创建的是动态分割窗口。程序清单 9-2 包含的是 Create()的伪代码，来自文件 WINSPLIT.CPP。

程序清单 9-2 CSplitterWnd::Create()（在文件 WINSPLIT.CPP 中）

```
BOOL CSplitterWnd::Create(CWnd* pParentWnd, int nMaxRows, int nMaxCols,
    SIZE sizeMin, CCreateContext* pContext, DWORD dwStyle, UINT nID)
{
    ASSERT(nMaxRows >= 1 && nMaxRows <= 2);
    ASSERT(nMaxCols >= 1 && nMaxCols <= 2);
    ASSERT(nMaxCols > 1 || nMaxRows > 1);       // 1x1 is not permitted
    m_nMaxRows = nMaxRows;
    m_nMaxCols = nMaxCols;
    m_nRows = m_nCols = 1;      // start off as 1x1
    if (!CreateCommon(pParentWnd, sizeMin, dwStyle, nID))
        return FALSE;
    m_pDynamicViewClass = pContext->m_pNewViewClass;
    if (!CreateView(0, 0, m_pDynamicViewClass, sizeMin, pContext))
        DestroyWindow(); // will clean up child windows return FALSE;
    m_pColInfo[0].nIdealSize = sizeMin.cx;
    m_pRowInfo[0].nIdealSize = sizeMin.cy;
    return TRUE;
}
```

Create()开始先执行一些严格的断言。这些断言语句使得 CSplitterWnd()所创建的窗口的

nMaxCols 和 nMaxRows 有一个或者两个都是 2。如果最大的行数和列数都是 1，那么就没有必要对窗口进行分割了。因为动态分割器只能有两列和两行，所以调用 Create()时实际上只有 3 种合法的参数值：1,2；2,1；2,2。

如果参数通过了这次检查，Create()会将 nMaxRows 和 nMaxCols 参数存储到相应的数据成员里。接着，Create()将当前的行数和列数初始化为 1(在动态分割器里，m_nRows 和 m_nCols 只能是 1 或者 2)。只有 1 行、1 列的窗口看上去和只有 1 个视图的窗口是一样的。

在初始化当前行数、列数和最大行数、列数后，Create()就将父窗口指针、最小大小、窗口风格以及 ID 传递给 CreateCommon()。我们马上就会学习 CreateCommon()了。

在调用 CreateCommon()后，Create()会将窗格视图的 CRuntimeClass 信息存储到 m_pDynamicViewClass 里。然后 Create()就会将该指针作为参数传递给 CreateView()。我们马上就会看到这点。

最后，Create()将 sizeMin 的值存储到当前窗格 pane(0,0)的 m_pColInfo 和 m_pRowInfo 里。这里有一点比较奇怪，我们一直没有看到这些数组是在哪里分配内存的。这些内存的分配一定是在 CreateCommon()或者是在 CreateView()里进行的。

让我们将 Create()和 CreateStatic()进行比较。程序清单 9-3 包含了在文件 WINSPLIT.CPP 中的 CreateStatic()的伪代码。

程序清单 9-3 CSplitterWnd::CreateStatic()的伪代码（在文件 WINSPLIT.CPP 中）

```
BOOL CSplitterWnd::CreateStatic(CWnd* pParentWnd, int nRows, int nCols,
    DWORD dwStyle, UINT nID)
{
    ASSERT(nRows >= 1 && nRows <= 16);
    ASSERT(nCols >= 1 && nCols <= 16);
    ASSERT(nCols > 1 || nRows > 1);
    ASSERT(!(dwStyle & SPLS_DYNAMIC_SPLIT));
    m_nRows = m_nMaxRows = nRows;
    m_nCols = m_nMaxCols = nCols;
    if (!CreateCommon(pParentWnd, CSize(0, 0), dwStyle, nID))
        return FALSE;
    return TRUE;
}
```

在 CreateStatic()的伪代码里，最开始的断言语句表明静态窗口最多有 16 列、16 行（也就是 256 个窗口！）。断言语句也会检查是否行数大于 1，或者列数大于 1（否则我们不需要分割窗口）。阴影部分对最后的断言语句做了解释。

在对参数做了彻底检查后，CreateStatic()将 m_nRows/m_nMaxRows（在静态窗口里，这两个值通常是相等的）设置成参数 nRows 的值。CreateStatic()对数据成员中列（column）的相应部分也会做同样的操作。

最后，CreateStatic()调用 CreateCommon()，参数依次为父窗口指针、(0,0) 的尺寸、风格以及整数 ID。

注意，与 Create()不同的是，CreateStatic()并不调用 CreateView()。创建所有的视图是静态分割窗口用户的责任，因为 CreateStatic()并不能知道运行时的信息。

你可以选择 SPLS_DYNAMIC_SPLIT 风格，也就是说：

在声明 Create()时，默认风格位中的一个被设置成了 SPLS_DYNAMIC_SPLIT。在 AFXEXT.H 中是这样定义的：

```
#define SPLS_DYNAMIC_SPLIT  0X0001
```

如果你需要确定某个 CSplitterWnd 到底是动态的还是静态的，这是惟一进行区分的方法。下面的示例代码说明了如何确定静态分割窗口和动态分割窗口：

```
if (m_wndSplitter.GetStyle() & SPLS_DYNAMIC_SPLIT){
    // It's dynamic!
    // ..
} else {
    // It's static!
    // ...
}
```

马上我们会看到 MFC 自己会做这样的检查。

Create()和 CreateStatic()比较相像，除了它们在参数的限制上有所不同，以及 Create()会调用 CreateView()，而 CreateStatic()不会。

两个函数都会调用 CreateCommon()，所以让我们看看这个函数会做哪些初始化。程序清单 9-4 包含了取自 WINSPLIT.CPP 的 CreateCommon()的伪代码。

程序清单 9-4 CSplitterWnd::CreateCommon()（在文件 WINSPLIT.CPP 中）

```
BOOL CSplitterWnd::CreateCommon(CWnd* pParentWnd,SIZE sizeMin,
                                DWORD dwStyle, UINT nID)
{
    DWORD dwCreateStyle = dwStyle & ~(WS_HSCROLL|WS_VSCROLL);
    if (afxData.bWin4)
        dwCreateStyle &= ~WS_BORDER;
    if (!AfxDeferRegisterClass(AFX_WNDMDIFRAME_REG))
        return FALSE;
    if (!CreateEx(0, _afxWndMDIFrame, NULL, dwCreateStyle,
        0, 0, 0, 0, pParentWnd->m_hWnd, (HMENU)nID, NULL))
        return FALSE;       // create invisible
    m_pColInfo = new CRowColInfo[m_nMaxCols];
    for (int col = 0; col < m_nMaxCols; col++){
        m_pColInfo[col].nMinSize = m_pColInfo[col].nIdealSize = sizeMin.cx;
        m_pColInfo[col].nCurSize = -1; // will be set in RecalcLayout
    }
    m_pRowInfo = new CRowColInfo[m_nMaxRows];
    for (int row = 0; row < m_nMaxRows; row++){
        m_pRowInfo[row].nMinSize = m_pRowInfo[row].nIdealSize = sizeMin.cy;
        m_pRowInfo[row].nCurSize = -1; // will be set in RecalcLayout
    }
    SetScrollStyle(dwStyle);
    return TRUE;
}
```

CreateCommon()首先会创建一个临时的风格变量 dwCreateStyle，并将初始的 dwStyle 参数减去 WS_HSCROLL、WS_VSCROLL 的值存放到该变量中。CreateCommon()这样做的原因是它不希望 Windows 窗口有滚动条——滚动条应该由分割窗口管理，而不是由 Windows

窗口管理。

在调整好风格标记后，如果应用程序运行在 Windows 95 下，CreateCommon()会减去 WS_BORDER 标记。

接着，CreateCommon()会调用 AfxDeferRegisterClass()（参看第 2 章），然后是 CWnd::CreateEx()，参数依次为无扩展风格、_afxWndMDIFrame（MDI 框架窗口类——不擦背景处理）、记录在 pParentWnd 中的父窗口句柄以及作为标识符的 nID。CreateEx()创建 Windows 窗口，所以我们在这里还是最终做了一些创建！

在调用完 CreateEx()后，CreateCommon()为 m_pColInfo 和 m_pRowInfo 数组分配空间，并对它们做初始化，将 m_nMaxCols/Rows 作为数组的大小。CreateCommon()在循环里依次访问 CRowColInfo 数组，并做如下的操作：

1. nMinSize 和 nIdealSize 都被设置成参数 sizeMin 的值。

2. nCurSize 被初始化为-1，这说明当窗格的尺寸被初始化（RecalcLayout()）时，该值应该被设置。

在初始化完 CRowColInfo 的行和列数组后，CreateCommon()调用 SetScrollStyle()，将 m_bHasHScroll 和 m_bHasVScroll 进行初始化，然后返回 TRUE。

AFX_IDW_PANE_FIRST：它与什么有关？

静态分割窗口和动态分割窗口的默认 ID 都是 AFX_IDW_PANE_FIRST（或者是 0xE900，如果你喜欢 16 进制的话）。MFC 也定义了窗格 ID 的最大值，即 AFX_IDW_PANE_LAST（0xE9ff），它将窗格的数目限制为 256。当所有窗格的 ID 都在这个范围时，MFC 可以很容易地检查一个窗口是否是一个窗格，这只要检查它的 ID 是否在这个范围就可以了。

什么时候会创建其 nID 不是 AFX_IDW_PANE_FIRST 的分割窗口呢？很好的问题！如果你有嵌套的分割窗口时会怎么样呢？让我们假设你希望创建有 2 个窗格的静态分割窗口：左边的窗格是一个视图，而右边的窗格是一个动态分割窗口。在这种情况下，你可以用 AFX_IDW_PANE_FIRST+1 来创建这个动态分割窗口——表明它不是第一个窗格，而只是第二个窗格（第一个窗格是父静态分割窗口的普通视图）。

MFC 的灵活性还不止这些！

现在我们已经看到真正的分割窗口已经被创建了（在 CreateCommon()里），下面让我们看看窗格是如何被 CreateView()创建的。程序清单 9-5 列出了它的程序清单，它来自文件 WINSPLIT.CPP。

程序清单 9-5 CSplitterWnd::CreateView()（在文件 WINSPLIT.CPP 中）

```
BOOL CSplitterWnd::CreateView(int row, int col, CRuntimeClass* pViewClass,
    SIZE sizeInit, CCreateContext* pContext)
{
    m_pColInfo[col].nIdealSize = sizeInit.cx;
    m_pRowInfo[row].nIdealSize = sizeInit.cy;
    BOOL bSendInitialUpdate = FALSE;
    CCreateContext contextT;
    if (pContext == NULL) {
        CView* pOldView = (CView*)GetActivePane();
        if (pOldView != NULL && pOldView->IsKindOf(RUNTIME_CLASS(CView))){
            contextT.m_pLastView = pOldView;
```

```
            contextT.m_pCurrentDoc = pOldView->GetDocument();
            if (contextT.m_pCurrentDoc != NULL)
                contextT.m_pNewDocTemplate =
                        contextT.m_pCurrentDoc->GetDocTemplate();
        }
    pContext = &contextT;
    bSendInitialUpdate = TRUE;
    }
    CWnd* pWnd = (CWnd*)pViewClass->CreateObject();
    DWORD dwStyle = AFX_WS_DEFAULT_VIEW;
    if (afxData.bWin4)
        dwStyle &= ~WS_BORDER;
    // Create with the right size (wrong position)
    CRect rect(CPoint(0,0), sizeInit);
    if (!pWnd->Create(NULL, NULL, dwStyle, rect, this,
            IdFromRowCol(row, col),
        pContext))
        return FALSE;
    // send initial notification message
    if (bSendInitialUpdate)
        pWnd->SendMessage(WM_INITIALUPDATE);
    return TRUE;
}
```

CreateView()最初将 sizeInit 参数存储在 CRowColInfo 相应的数组下标（array index）里。接下来，CreateView()设置一个局部标记 bSendInitialUpdate，值为 FALSE。

接下来的代码真的是很酷。如果出于某种原因 CCreateContext 指针为空，CreateView()会创建一个局部的 CCreateContext()，并且以自己的最大努力将每个元素初始化为比较完整的值。它调用 GetActivePane()来确定 m_pLastViewCView 指针。一旦 CreateView()有了 m_pLastView，它就可以通过调用 GetDocument()来确定 CCreateContext 其他域的值，然后调用 CDocument::GetDocTemplate()。在它找到所有这些元素后，CreateView()使得 pContext 指针指向它们，并将 bSendInitialUpdate 设置为 TRUE，这样，它们就能在后面的函数中被正确地初始化了。

接着，CreateView()调用 CreateObject()（记得它是第 5 章里的吗？）为 CRuntimeClass 信息创建一个窗格对象（CWnd 的派生对象）。在创建窗格对象后，CreateView()设置风格和定位矩形，它们是被传递到 Create()的参数。IdFromRowCol()函数基于窗格的行和列计算窗格的 ID。公式是 AFX_IDW_PANE_FIRST+row*16+col。所以，如果窗格的行是 0，列是 5，则 ID 就是 AFX_IDW_PANE_FIRST+5。如果窗格的行是 5，列是 0，ID 就是 AFX_IDW_PANE_FIRST+80。这个公式能保证所有 256 个可能的窗格都会有惟一的标识符，并且取值范围是从 AFX_IDW_PANE_FIRST 到 AFX_IDW_PANE_LAST（参看前面的阴影部分）。

如果对 Create()的调用失败了，CreateView()会返回 FALSE。如果 Create()成功了，当 bSendInitialUpdate 标记被设置为 TRUE 时，CreateView()会向窗格发送 WM_INITIALUPDATE 消息。仅当 CreateView()不得不自己得到 CCreateContext 信息时，该标记才会被设置。最后，CreateView()返回 TRUE。

现在，你已经看到了初始化分割窗口和创建分割窗口所需要做的全部事情了。在分割窗口和一个或多个窗格被 Create()或者 CreateStatic()创建后，所有的分割窗口行为都是对窗口消

息和用户交互的响应。

在研究 MFC 到底是如何画出这些精致的分割窗口组件之前，让我们先来看看 MFC 是如何管理这些不同的窗格的。

窗格管理

在分割窗口里有两种形式的窗格管理。首先，让我们看看窗格是如何在动态分割窗口里被动态创建的。然后，我们会看看当用户改变包含有分割窗口的框架大小时，窗格是如何改变大小以及如何定位的。

有两个 CSplitterWnd 成员函数：SplitRow()和 SplitColumn()。它们被调用来动态创建窗格。我们随机挑选一个，SplitColumn()，从而看看它是如何动态创建新的一列窗格的。

SplitColumn()有一个参数，即分割的位置，并且当用户创建一个垂直的分割时被调用。SplitColumn()伪代码在程序清单 9-6 中，来自文件 WINSPLIT.CPP。

程序清单 9-6 CSplitterWnd:: SplitColumn() (在文件 WINSPLIT.CPP 中)

```
BOOL CSplitterWnd::SplitColumn(int cxBefore)
{
ASSERT(GetStyle() & SPLS_DYNAMIC_SPLIT);
cxBefore -= m_cxBorder;
int colNew = m_nCols;
int cxNew = CanSplitRowCol(&m_pColInfo[colNew-1], cxBefore, m_cxSplitter);
if (cxNew == -1)
return FALSE;   // too small to split
// create the scroll bar first (so new views can see that it is there)
if (m_bHasHScroll &&
!CreateScrollBarCtrl(SBS_HORZ, AFX_IDW_HSCROLL_FIRST + colNew))    {
    TRACE0("Warning: SplitRow failed to create scroll bar.\n");
    return FALSE;
    }

m_nCols++;  // bump count during view creation
// create new views to fill the new column (RecalcLayout will position)
    for (int row = 0; row < m_nRows; row++)    {
        CSize size(cxNew, m_pRowInfo[row].nCurSize);
        if (!CreateView(row, colNew, m_pDynamicViewClass, size, NULL))   {
            TRACE0("Warning: SplitColumn failed to create new column.\n");
            // delete anything we partially created 'col' = # columns
            // created
            while (row > 0)
                DeleteView(--row, colNew);
            if (m_bHasHScroll)
                GetDlgItem(AFX_IDW_HSCROLL_FIRST + colNew)->DestroyWindow();
                m_nCols--;      // it didn't work out return FALSE;
        }
    }
    // new parts created - resize and re-layout
    m_pColInfo[colNew-1].nIdealSize = cxBefore;
    m_pColInfo[colNew].nIdealSize = cxNew;
    RecalcLayout();
    return TRUE;
}
```

SplitColumn()不允许用作静态分割窗口，所以它会首先通过断言语句检查SPLS_DYNAMIC_SPLIT标记来确定当前分割窗口是动态的。

接着，SplitColumn()从边界的大小中减去列的大小，并且将局部变量colNew设置成列数。SplitColumn()然后将局部变量cxNew（新的列宽）设置成调用CanSplitRowCol()的返回值。CanSplitRowCol()基于列信息、cxBefore值以及分割器的宽度计算窗格的新宽度。如果用户创建了一个尺寸小于最小窗格大小的分割，CanSplitRowCol()会返回－1，指明该窗格不能被创建。SplitColumn()会对这种情况进行检查，如果窗格不能被创建，则返回FALSE。CanSplitRowCol()会在调试时产生一条TRACE语句（你也许从来就没有注意过这一点，但是，当分割窗口处在调试状态时，如果你运行Scribble指南，并且创建一个非常小的分割，你就会看到CanSplitRowCol()会拒绝新的画板尺寸，并将分割窗口重新恢复成未被分割的窗口）。

在CanSplitRowCol()函数成功返回后，如果m_bHasHScroll的值为TRUE，SplitColumn()则为该控件创建滚动条。m_bHasHScroll和m_bHasVScroll的设置都是在SetScrollStyle()里进行的，并且基于WS_VSCROLL/WS_HSCROLL风格位。你也许还记得SetScrollStyle()是在CreateCommon()里被调用的。一旦滚动条被创建好了，SplitColumn()就会将列数（m_nCols）增加1。

SplitColumn()的下一块代码是真正创建新的窗格的。注意，这实际上是一个“for”循环，循环次数为列数。SplitColumn()这样做的原因是如果有两行，并且用户自己创建一个新的列，那么就会有两个窗格、而不是一个窗格需要被创建。

在SplitColumn()以行数进行“for”循环并为每一行创建新的窗格时，它会调用CreateView()，并且以cxNew为宽度，以当前列的高度为高度。如果CreateView()失败了，SplitColumn在做完清理工作后返回FALSE。

一旦“for”循环结束后，SplitColumn()会用分割后的列的宽度来更新当前和以前的CRowColInfo数组元素。SplitColumn()最后调用RecalcLayout()来使得分割窗口的所有窗格和构件都被重画。

浏览一下CSplitterWnd的程序清单，你会发现在几乎每一个处理窗格布局的成员函数的最后，都会调用RecalcLayout()。RecalcLayout()从字面就可以看出它会控制分割窗口的每一个构件的布局。理解RecalcLayout()对于理解分割窗口是如何工作的以及如何进行更新的很关键。

正如你可以想像的，RecalcLayout()是一个非常长的成员函数，所以在这里我们不能把它列出来。相反，我们会对RecalcLayout()的运行给出一个大概的描述。在读下面这些步骤时，打开WINSPLIT.CPP文件是个很不错的主意。

1. RecalcLayout()通过调用::GetClientRect()来得到用户区的矩形，并且通过调用CSplitterWnd::GetInsideRect()来获得内部的矩形。

2. RecalcLayout()通过调用LayoutRowCol()来计算m_pColInfo/m_pRowInfo中行和列的布局情况。LayoutRowCol()根据尺寸参数和存储在CRowColInfo参数里的尺寸线索计算行和列应该放置在什么地方。RecalcLayout()会调用LayoutRowCol()两次：一次是为了列，并且基于内部矩形的宽度；另一次是为了行，基于内部矩形的高度。

3. 接着，RecalcLayout()调用::BeginDeferWindowPos()来建立起Begin/Defer/EndWindowPos的调用“三重唱”。RecalcLayout()使用行和列的个数来计算将要被移动的窗口数

目，::BeginDeferWindowPos()返回的句柄被存放在局部变量 layout 中，类型为 AFX_SIZEPARENTPARAMS 结构（如果要了解更多的关于私有 MFC 结构的信息，可以参考阴影部分）。

4. RecalcLayout()接着计算滚动条的尺寸，并通过调用 DeferClientPos()来改变它们的位置。DeferClientPos()得到尺寸，然后调用 AfxRepositionWindows()并将 AFX_SIZEPARENTPARAMS 结构的指针作为参数传递过去。AfxRepositionWindow()位于 WINCORE.CPP 中。它仍会做更多的检查，并且最后使用 AFX_SIZEPARENTPARAMS 结构里的句柄来调用 DeferWindowPos。

5. 在将所有的滚动条进行重定位后，RecalcLayout()对所有的窗格进行循环，为每一个窗格调用 DeferClientPos()，其中传递的参数有相应的 CRowColInfo 数组元素中的尺寸信息、对分割间距所作的微小的调整以及其他一些东西。

6. 一旦所有的滚动条和窗格被重定位后，RecalcLayout()调用::EndDeferWindowPos()来快速地、一次性地移动所有的窗口。

7. 最后，RecalcLayout()会用一个空的 DC 来调用 DrawAllSplitBars()，使得分割条无效，从而使得它们可以在新的位置上被重画。分割条将基于新的行和列的位置被重新绘制。

AFX_SIZEPARENTPARAMS 私有结构

这个私有结构是在 AFXPRIV.H 里声明的，如下所示：

```
struct AFX_SIZEPARENTPARAMS
{
    HDWP hDWP;              // handle for DeferWindowPos
    RECT rect;              // parent client rectangle (trim as appropriate)
    SIZE sizeTotal;         // total size on each side as layout proceeds
    BOOL bStretch;          // should stretch to fill all space
};
```

AFX_SIZEPARENTPARAMS 通过 MFC 的窗口消息 WM_SIZEPARENT 来向父窗口传递大小信息。这个结构很重要，因为在本章中的很多地方我们都会用它来帮助对一组兄弟窗口进行布局。

现在我们已经了解了 CSplitterWnd 是如何对行和列进行布局的，下面让我们看看分割条是如何被画出来的。

CSplitterWnd 的绘画

RecalcLayout()成员函数以调用 DrawAllSplitBars()为结束，所以让我们从那里开始进行对绘画的讨论。DrawAllSplitBars()又是一个很长的例程，所以我们会要求你使用自己的代码编辑器来跟上我们的讨论。

在详细分析 DrawAllSplitBars()到底做了些什么之前，注意这个函数实际上并没有做任何“绘画”。OnDrawSplitter()成员函数做了所有实际的绘画操作。OnDrawSplitter()的参数有一个 DC 指针、一个 ESplitTye 类型的枚举参数以及一个矩形。在分析 DrawAllSplitBars()是如何驱动该函数之后，我们再来看看 OnDrawSplitter()。

首先，DrawAllSplitBars()遍历所有的列，并且画出垂直的分割条，其中循环次数为 m_nCols-1 次。DrawAllSplitBars()基于分割器的宽度、列的 CRowColInfo 以及其他一些用于

界面的数据成员计算出分割条的尺寸。

接着，DrawAllSplitBars()遍历所有的行，并且执行和上面相同的操作，但是现在是用分割窗口的高度和行的信息来画出行的分割条。

最后，如果程序运行在 Windows 95 下，DrawAllSplitBars()会在每个窗格的周围都画上 3D 的边界。这样分割条和边界才会很好地联系在一起，而不是看上去像是不同的程序画出来的。

现在，我们期待已久的时刻到了！让我们看看 CSplitterWnd 到底是如何画出所有这些神奇的分割框和分割条的。正如我们在 DrawAllSplitBars()的讨论中提到，OnDrawSplitter()做了分割窗口所有的绘画操作。

程序清单 9-7 包含了该成员函数的伪代码，它在文件 WINSPLIT.CPP 中。

程序清单 9-7 CSplitterWnd:: OnDrawSplitter() (在文件 WINSPLIT.CPP 中)

```
void CSplitterWnd::OnDrawSplitter(CDC* pDC, ESplitType nType,
     const CRect& rectArg)
{
    if (pDC == NULL)
        RedrawWindow(rectArg, NULL, RDW_INVALIDATE|RDW_NOCHILDREN);
        return;
    CRect rect = rectArg;
    switch (nType)    {
        case splitBorder:
            pDC->Draw3dRect(rect, afxData.clrBtnShadow,
                afxData.clrBtnHilite);
            rect.InflateRect(-CX_BORDER, -CY_BORDER);
            pDC->Draw3dRect(rect, afxData.clrWindowFrame,
                afxData.clrBtnFace);
            rect.InflateRect(-CX_BORDER, -CY_BORDER);
            return;
        case splitIntersection:
            break;
        case splitBox:
            if (afxData.bWin4){
                pDC->Draw3dRect(rect, afxData.clrBtnFace,
                    afxData.clrWindowFrame);
                rect.InflateRect(-CX_BORDER, -CY_BORDER);
                pDC->Draw3dRect(rect, afxData.clrBtnHilite,
                    afxData.clrBtnShadow);
                rect.InflateRect(-CX_BORDER, -CY_BORDER);
                break;
            }
        // fall through...
        case splitBar:
            if (!afxData.bWin4){
                pDC->Draw3dRect(rect, afxData.clrBtnHilite,
                    afxData.clrBtnShadow);
                rect.InflateRect(-CX_BORDER, -CY_BORDER);
            }
        break;
       default:
           ASSERT(FALSE);  // unknown splitter type
}
```

```
        pDC->FillSolidRect(rect, afxData.clrBtnFace);
    }
```

如果传递给 OnDrawSplitter()的 CDC 指针是空的，这表明调用者希望使得矩形无效。在这种情况下，OnDrawSplitter() 会调用 CWnd::RedrawWindow()，并以矩形和 RDW_INVALIDATE 标志集为参数。

另一方面，如果 CDC 指针非空，OnDrawSplitter()会基于 ESplitType 参数进行切换。从 CSplitterWnd 头文件我们知道，它是各种分割窗口构件的一个枚举类型（例如分割边界、分割交点以及分割框）。

在我们分析 switch 语句不同的情况之前，要注意在 switch 语句之后，OnDrawSplitter() 会调用 FillSolidRect()，其中参数为局部矩形变量和按钮的表面颜色。知道这之后，我们能推断出在 FillSolidRect()之前的 switch 语句会在矩形中添加一些装饰。下面让我们分析 switch 语句块，看看 OnDrawSplitter()做了些什么使得被填充的矩形看起来更神奇。

switch 语句的第一种情况是 splitBorder。DrawAllSplitBars()以 splitBorder 类型为参数调用 OnDrawSplitter()的惟一情况就是当应用程序运行在 Windows 95 上。当 OnDrawSplitter()画边界时，它会调用 CDC::Draw3dRect()两次。第一次调用画出边界的外部，第二次调用画出边界的内部，偏移量移为 CX_BORDER 和 CY_BORDER。这为窗格的内部创造出了窗口的边界效果。

switch 语句的第二种情况是 splitIntersection，它仅仅跳出 switch 语句。

splitBox 是第三种情况。让我们用这个例子来看看这些 GDI 调用到底做了些什么，并且是如何画出这么酷的分割框的。

图 9-2 显示了 Windows 95 分割框的一个特写镜头，说明了每个 GDI 调用画的是分割框的哪个部分。

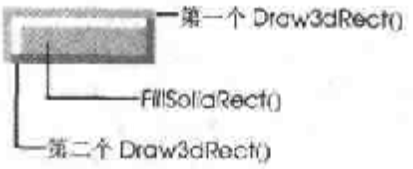


图 9-2 Windows 95 分割框的特写镜头

如果应用程序运行在 Windows 95 下，OnDrawSplitter()会画出三维的分割框，这通过调用两次 CDC::Draw3dRect()成员函数。第一次调用画出边界的外部，第二次调用画出边界的内部。在图 9-2 里，你会看到这种效果。注意 Draw3dRect()是如何用两个颜色参数的，它用一种颜色画出了矩形的顶部和左部，并用另一种颜色画出了矩形的底部和右部。结果就是 Windows 95 有了轮廓分明的 3D 界面。如果你曾经努力完成同样的界面，Draw3dRect()可以帮你。在 Windows 95 下，对于分割框这种情况，程序跳出 switch 语句后就直接到了 FillSolidRect()。FillSolidRect()填充分割框的中心区域，从而完成了分割框的绘画，如图 9-2 所示。

图 9-3 Windows NT 分割框的组件

如果应用程序运行在 Windows NT（或者 Windows 32）下，对于 splitBox 这种情况，程序会运行到 splitBar 这种情况。图 9-3 显示了 Windows NT 的分割框。注意，在这张图中，黑色的外框是为按钮提供对比的。这个边界不是 OnDrawSplitter()画的。

对于 splitBar 这种情况，只有在程序不是运行在 Windows 95 下时，OnDrawSplitter()才会画出某些 3D 的阴影。记住，在 NT 和 Win 32 下，分割框和分割条的代码是一样的。NT 的

分割框和分割条在画出一个 3D 的矩形后就直接将矩形传给了 FillSolidRect()。图 9-3 显示了每个 GDI 调用对分割器界面的影响。

现在比较一下图 9-2 中 Windows 95 的分割框和图 9-3 中 Windows NT 的分割框的区别。Windows 95 的分割框只是多了一个倾斜的矩形就看起来特别“深”，这真是很让人惊奇。

现在我们快要完成 CSplitterWnd 的界面了。记住，DrawAllSplitBars()只会关心分割条和分割边界的绘画。这留下了很大的一个问题，是哪个函数调用 OnDrawSplitter()来画出分割框的呢？答案是 OnPaint()。在程序清单 9-8 中你可以看到 OnPaint()的伪代码，它在文件 WINSPLIT.CPP 中。

程序清单 9-8 CSplitterWnd:: OnPaint()的伪代码（在文件 WINSPLIT.CPP 中）

```
void CSplitterWnd::OnPaint()
{
    CPaintDC dc(this);
    CRect rectClient;
    GetClientRect(&rectClient);
    rectClient.InflateRect(-m_cxBorder, -m_cyBorder);
    CRect rectInside;
    GetInsideRect(rectInside);
    if (m_bHasVScroll && m_nRows < m_nMaxRows)
        OnDrawSplitter(&dc, splitBox, CRect(rectInside.right +
        afxData.bNotWin4, rectClient.top, rectClient.right,
        rectClient.top + m_cySplitter));
if (m_bHasHScroll && m_nCols < m_nMaxCols)
    OnDrawSplitter(&dc, splitBox,
        CRect(rectClient.left, rectInside.bottom + afxData.bNotWin4,
        rectClient.left + m_cxSplitter, rectClient.bottom));
DrawAllSplitBars(&dc, rectInside.right, rectInside.bottom);
if (!afxData.bWin4) {
    // draw splitter intersections (inside only)
    GetInsideRect(rectInside);
    dc.IntersectClipRect(rectInside);
    CRect rect;
    rect.top = rectInside.top;
    for (int row = 0; row < m_nRows - 1; row++) {
        rect.top += m_pRowInfo[row].nCurSize + m_cyBorderShare;
        rect.bottom = rect.top + m_cySplitter;
        rect.left = rectInside.left;
        for (int col = 0; col < m_nCols - 1; col++) {
            rect.left += m_pColInfo[col].nCurSize + m_cxBorderShare;
            rect.right = rect.left + m_cxSplitter;
            OnDrawSplitter(&dc, splitIntersection, rect);
            rect.left = rect.right + m_cxBorderShare;
        }
        rect.top = rect.bottom + m_cxBorderShare;
    }
}
```

OnPaint()通过得到用户矩形和内部矩形来准备绘画。OnPaint()调用 OnDrawSplitter()来画出水平的和垂直的分割框。在画完分割框后，OnPaint()调用 DrawAllSplitBars()来画出分割条和分割边界。最后，如果应用程序不是运行在 Windows 95 下，OnPaint()会做许多工作来画出

分割条的交点。

这段代码会引出一个非常有趣的问题：CSplitterWnd 为什么不对 Windows 95 下的交点做一些特殊的处理（记住，OnDrawSplitter()在 splitIntersection 这种情况下只是直接跳出 switch 语句）？

要找到这个答案，让我们将 Windows 95 和 Windows NT 的交点都放到很容易得到的 MFC 内部显微镜下。Windows 95 的分割交点在图 9-4 中显示，Windows NT 的分割交点在图 9-5 中显示。每个图都说明了画出分割交点这个部分的命令。现在你明白了为什么 MFC 不在 Windows 95 下画额外的 3D 矩形了吧。

你猜对了！因为 MFC 会为 Windows 95 的应用程序画出分割窗口边界，而边界的绘画代码实际上也使得分割条看上去也有 3D 的效果。因为分割条是凹进去的（而且更宽），所以分割交点也就自动被画出来了。

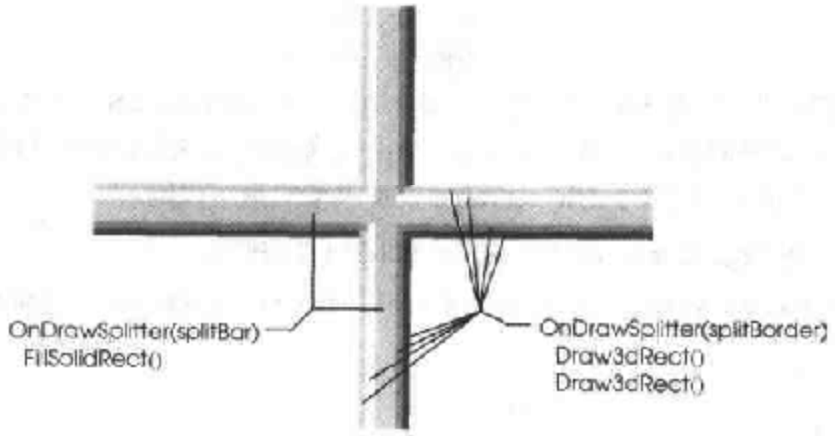


图 9-4 Windows 95 分割交点的特写镜头

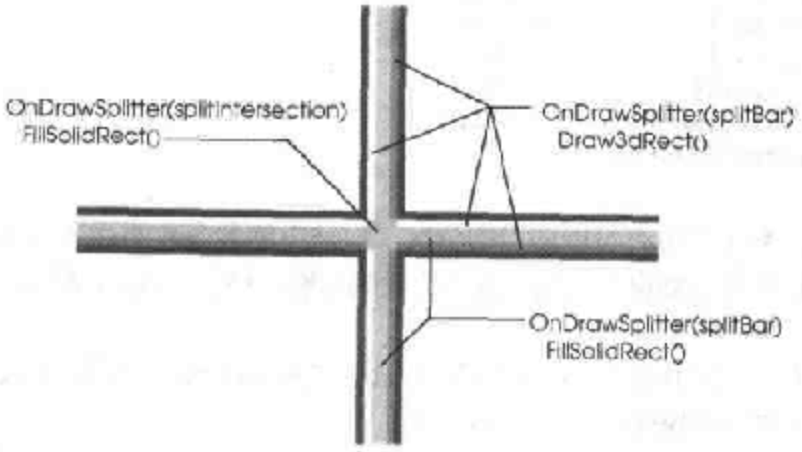


图 9-5 在显微镜下的 Windows NT 分割交点

另一方面，Windows NT 应用程序不画边界，所以它画的分割条的 3D 效果就更好了。当 NT 应用程序做这些时，分割交点看上去很小。MFC 为了使得分割交点显得更轮廓分明，就在那里画了一个小的矩形（如图 9-5 所示）。

现在你应该明白了每个分割窗口的构件是如何被画出来了。对于你来说，这可能像是

MFC 的一些琐碎小事，但一旦你希望或者需要改变 MFC 分割窗口的界面，那么你刚学习到的分割窗口的绘画就真的是太有用了！

从学习 CSplitterWnd 到现在，我们还有两个有趣的问题没有弄清：点击测试和跟踪。让我们首先来看看点击测试是什么，以及为什么 CSplitterWnd 需要它。

分割窗口的点击测试

如果你曾经写过 Windows 控件或者 MFC 控件，那么点击测试的概念对你来说应该是已经非常熟悉了。如果你还不知道点击测试，那么还需要学习。简而言之，点击测试就是这样的一种技术，能够让你用程序来确定用户的界面操作是否影响了某个控件。对于分割窗口来说，分割窗口需要对以下的用户操作进行点击测试：

- 鼠标移动，这样窗口就可以在鼠标移过分割条和分割框时改变鼠标。
- 按钮被按下，这样窗口就可以拖动框条、创建框条等等。
- 按钮被放开，这样窗口就可以停止拖动、停止创建等等。

即使你以前写过一些点击测试的代码，你仍要学习：MFC 的 CSplitterWnd 非常有趣。

让我们用 WINSPLIT.CPP 中的一个很重要的内部枚举类型来开始对 CSplitterWnd 点击的测试分析。程序清单 9-9 包含了 HitTestValue 枚举类型的声明代码。

程序清单 9-9 HitTestValue 枚举类型（在文件 WINSPLIT.CPP 中）

```
// HitTest return values (values and spacing between values is important)
enum HitTestValue
{
    noHit                   = 0,
    vSplitterBox            = 1,
    hSplitterBox            = 2,
    bothSplitterBox         = 3,
    vSplitterBar1           = 101,
    vSplitterBar15          = 115,
    hSplitterBar1           = 201,
    hSplitterBar15          = 215,
    splitterIntersection1   = 301,
    splitterIntersection225 = 525
};
```

HitTestValue 列举了所有可能的点击情况。枚举中绝大多数的意义确实是很容易猜出来的。例如，noHit 表明分割窗口的构件都不会受到影响，vSplitterBox 表明一个垂直的分割框被点击了。

在枚举中第一个不是很明显的值是 bothSplitterBox。当开发人员想通过调用 DoKeyboardSplit()来分割窗口时会使用这个值。

在枚举中还有一些比较让人困惑的名字和值。例如，vSplitterBar1 和 vSplitterBar15。你能猜出这些值为什么取这样的名字，以及为什么它们的值是 101 和 115 吗？

记住，在静态分割窗口中行的最高限度是 16。枚举中的这些值使得点击测试在工作时能有一个逻辑上的点击测试“范围”。对于列来说也是一样的。因为最多只能有 256 个交点。HitTestValue 的取值范围是从 splitterIntersection1（301）到 splitterIntersection225（525）。在我们分析 HitTest()时我们会看看 MFC 是如何使用这个取值范围的。

分析 MFC 点击测试的关键在于 HitTest()成员函数。HitTest()以一个点为参数，并返回 HitTestValue 枚举中的一个。程序清单 9-10 包含了 HitTest()的伪代码，它在文件 WINSPLIT.CPP 中。

程序清单 9-10 CSplitterWnd:: HitTest()的伪代码（在文件 WINSPLIT.CPP 中）

```
int CSplitterWnd::HitTest(CPoint pt) const
{
    CRect rectClient;
    GetClientRect(&rectClient);
    rectClient.InflateRect(-m_cxBorder, -m_cyBorder);
    CRect rectInside;
    GetInsideRect(rectInside);
    if (m_bHasVScroll && m_nRows < m_nMaxRows &&
        CRect(rectInside.right, rectClient.top, rectClient.right,
        rectClient.top + m_cySplitter - afxData.bWin4).PtInRect(pt))
            return vSplitterBox;
    if (m_bHasHScroll && m_nCols < m_nMaxCols && CRect(rectClient.left,
        rectInside.bottom, rectClient.left + m_cxSplitter - afxData.bWin4,
        rectClient.bottom).PtInRect(pt))
            return hSplitterBox;
    CRect rect;
    rect = rectClient;
    for (int col = 0; col < m_nCols - 1; col++) {
        rect.left += m_pColInfo[col].nCurSize;
        rect.right = rect.left + m_cxSplitterGap;
        if (rect.PtInRect(pt))
            break;
        rect.left = rect.right;
    }
    rect = rectClient;
    for (int row = 0; row < m_nRows - 1; row++) {
        rect.top += m_pRowInfo[row].nCurSize;
        rect.bottom = rect.top + m_cySplitterGap;
        if (rect.PtInRect(pt))
            break;
        rect.top = rect.bottom;
    }
    // row and col set for hit splitter (if not hit will be past end)
    if (col != m_nCols - 1) {
        if (row != m_nRows - 1)
        return splitterIntersection1 + row * 15 + col;
        return hSplitterBar1 + col;
    }
    if (row != m_nRows - 1)
        return vSplitterBar1 + row;
    return noHit;
}
```

HitTest()首先得到用户矩形和内部矩形。接着，HitTest()为每个分割窗口构件创建临时的 CRect 对象，并且以 CPoint 为参数调用 CRect::PtInRect()来确定指定的点是否在分割窗口构件对应的矩形里。

你也许会奇怪 HitTest()为什么不检查分割框的窗口句柄并使用 GetWindowFromPoint()或

者一些其他类似的技术。记住，所有的分割窗口构件都是被 CSplitterWnd 创建的，所以它们都有相同的窗口。HitTest()不得不求助于很粗暴的矩形计算，因为这是它做检查时惟一能得到的信息。

在检查完分割框后，HitTest()遍历各个列来寻找点击。如果点击被发现是在某个行或列上发生的，HitTest()就会跳出行或者列的循环。在行和列的循环之后，HitTest()通过检查“for”循环（行和列）的当前下标是否是 m_nRows-1 或者 m_nCols-1 来检测行点击或者列点击。如果下标不是在行数和列数的最后，说明这确实是行点击或者列点击。

如果某个列被点击了，HitTest()还会检查行点击。如果行和列都被点击了，那么这肯定是一个交点。如果真的是这种情况，HitTest()返回 vSplitterIntersection1+row*15+col。这个公式为 16×16 个行列交点提供了不同的标识符。

如果只有行点击或者只有列点击，HitTest()会为列点击返回 hSplitterBar1 并加上列数，为行点击返回 vSplitterBar1 并加上行数。如果 HitTest()的 CPoint 参数没有落到任何 CSplitterWnd 构件矩形里，noHit 的值将被返回。

点击测试实际上是对大量矩形的检查。CSplitterWnd 通过应用它和跟踪技术来实现分割窗口用户界面。

分割窗口跟踪

分割窗口跟踪指的是由 CSplitterWnd 实现的“拖放”的用户界面效果。例如，当用户点击了一个可变大小的框条、将分割条的阴影条拖动到所希望的地方然后放下阴影条（放开鼠标的按钮）时，跟踪就开始了。在跟踪过程中 CSplitterWnd 需要执行多个操作。在本节里，我们将看看其中的几个是如何实现的。

让我们通过下面的情况示例来看看跟踪到底是如何进行的：用户在分割框上按下了鼠标按钮，拖动分割条的阴影条，然后松开按钮。处理的操作开始于 CSplitterWnd 的 OnLButtonDown()消息处理函数中。程序清单 9-11 包含了在文件 WINSPLIT.CPP 中的 OnLButtonDown()的代码（记住，数据成员 m_bTracking 表明用户正在拖动分割条）。

程序清单 9-11 CSplitterWnd:: OnLButtonDown()的伪代码（在文件 WINSPLIT.CPP 中）

```
void CSplitterWnd::OnLButtonDown(UINT /*nFlags*/, CPoint pt)
{
    if (m_bTracking)
        return;
    StartTracking(HitTest(pt));
}
```

如果 m_bTracking 表明用户已经开始跟踪了，OnLButtonDown()会直接返回，因为用户不可能同时激活两个跟踪操作。如果用户没有开始跟踪，OnLButtonDown()会调用 StartTracking()，并以 HitTest(pt)的返回值为参数，这里的 pt 是鼠标的左键被放开的地方。

StartTracking()的代码太长了，这里就不把它列出来了。它的功能就是用 HitTest()的返回值来初始化一个跟踪操作。如果 HitTest()返回 noHit，StartTracking()马上返回，并不采取任何行动。如果点击是交点点击，StartTracking()将为拖动两个分割条的阴影条做好准备：将 m_bTracking2 设为 TRUE，并且初始化 m_rectTracker 和 m_rectTracker2。这些矩形是分割条的阴影条的开始矩形。

如果分割是由于程序的运行（bothSplitterBox）导致的，StartTracking()会将它当作交点并采取同样的操作，只是阴影条的开始位置不是当前交点的当前位置，而是开始于分割窗口的中心位置。

如果 StartTracking()运行到了这里，说明或者是分割框被点击了，或者是一个分割条被点击了。如果是这种情况，StartTracking()会将 m_rectTracker 初始化为最开始的阴影矩形。

在 StartTracking()在该点的基础上已经做了初始化后，它会通过调用 SetCapture()来激活跟踪操作，并将鼠标的输入锁定在分割窗口内部。接着，StartTracking()将 m_bTracking 设置为 TRUE，并且如果 m_bTracking 被设置了，StartTracking()接着以 m_rectTracker 和 m_rectTracker2 为参数调用 OnInvertTracker()。

最后，StartTracking()将点击测试的值存放到 m_htTrack 里，接着调用 SetSplitCursor()来强制鼠标被更新。

这时，StartTracking 就已经完成了。现在轮到另外两个 CSplitterWnd 消息处理函数来完成跟踪操作：OnMouseMove()和 OnLButtonUP()。在分析这两个跟踪消息处理函数之前，让我们看看 OnInvertTracker()函数做了些什么来画出这么漂亮的分割条的阴影条。程序清单 9-12 显示了在文件 WINSPLIT.CPP 中的 OnInvertTracker()的伪代码。

程序清单 9-12　CSplitterWnd:: OnInvertTracker()的伪代码（在文件 WINSPLIT.CPP 中）

```
void CSplitterWnd::OnInvertTracker(const CRect& rect)
{
    CDC* pDC = GetDC();
    CBrush* pBrush = CDC::GetHalftoneBrush();
    HBRUSH hOldBrush = NULL;
    if (pBrush != NULL)
        hOldBrush = (HBRUSH)SelectObject(pDC->m_hDC, pBrush->m_hObject);
    pDC->PatBlt(rect.left, rect.top, rect.Width(), rect.Height(), PATINVERT);
    if (hOldBrush != NULL)
        SelectObject(pDC->m_hDC, hOldBrush);
    ReleaseDC(pDC);
}
```

OnInvertTracker()实际上是一个基本的 rubber-banding 程序。它选择了半色调的画刷（一种有颤动效果的画刷）后，接着以 PAINTVERT 为参数调用 CDC::PatBlt()。带有 PAINTVERT 参数的 PatBlt()实际上做的是异或操作，它自动擦去原来的 PatBlt()来给出 rubber-banding 的效果。在 PatBlting 后，OnInvertTracker()选择原来的旧画刷，并且释放 DC。

在 OnLButtonDown()之后，跟踪在 OnMouseMove()里继续着。如果 m_bTracking 没有被设置，OnMouseMove()调用 HitTest()，并将 SetSplitCursor()的返回值作为参数传递过去。这样，当用户将鼠标移过某个分割窗口构件时，鼠标就能被更新了。

如果 m_bTracking 被设置了（这是我们更加关心的部分），OnMouseMove()将跟踪移到当前鼠标所在的位置，方法是用以前的矩形为参数调用 OnInvertTracker()来擦去原来的分割条的阴影条 。在旧的阴影条被擦去后，OnMouseMove()再次调用 OnInvertTracker()，从而在鼠标移动到的新位置上画出新的阴影条。

当用户松开鼠标按钮时，跟踪就结束了，而且 CSplitterWnd 的 OnLButtonUp()消息处理函数会被调用。OnLButtonUp()用 TRUE 为参数调用 StopTracking()。

StopTracking()会释放鼠标的捕获，并通过最后一次调用 OnInvertTracker()擦去所有的阴影。下面的 StopTracking()操作依赖于被跟踪的组件，保存 HitTest()返回值的 m_htTrack 暗示了是哪个组件。下面的链表显示了不同点击测试的可能情况，以及 StopTracking()为它们所作的不同操作。

- vSplitterBox——调用 SplitRow()来执行动态垂直分割。
- hSplitterBox——调用 SplitColumn()来执行动态水平分割。
- vSplitterBarX——调用 TrackRowSize()，用新的大小信息来更新内部的 CRowColInfo 数组，接着调用 RecalcLayout()。
- hSplitterBarX——调用 TrackColumnSize()，用新的大小信息来更新内部的 CRowColInfo 数组，接着调用 RecalcLayout()。
- splitterIntersectionX——调用 TrackRowSize()和 TrackColumnSize()来为可变大小的行和列更新内部行和列数据。在更新完这些信息后，StopTracking()会调用 RecalcLayout()。
- bothSplitterBox——调用 SplitRow()和 SplitColumn()在分割窗口里创建水平分割和垂直分割。

一旦 StopTracking()结束后，分割窗口的跟踪操作也就完成了。

现在，你已经理解了 CSplitterWnd 的各种细节信息，从初始化到绘画，到跟踪。让我们通过用户创建动态分割时 CSplitterWnd 所执行的操作来将所有的知识放到一起。

窗格一定是视图吗？

在整个对 CSplitterWnd 的分析中，你也许已经注意到 CSplitterWnd 对成员的命名好像暗示着窗格一定是视图——例如，CreateView()、m_pNewViewClass 以及其他等等。

这些成员也能够被命名为 CreatePane()、m_pNewPaneClass，因为任何 CWnd 的派生类都可用作 CSplitterWnd 窗格。当然，会有一些警告，而且要使得一切正常工作要花费很多精力。例如，CSplitterWnd 的同步滚动和激活逻辑是特定于视图的。而且，如果你决定用 CWnd 的派生类来当作窗格，而不是用 CView，那么你会失去视图非常容易实现的文档，视图自动更新功能，它会使得分割非常容易。

CSplitterWnd 总结

为了总结，我们会举出一个分割的例子来浏览分割窗口函数，并将每一个我们详细分析了的分割窗口特性紧紧地联系到一起。在这个例子中，让我们假设用户在一个以前没有做过分割的动态分割窗口上做了垂直分割。在这个分割之后，就会有两个窗格（也就是说，两列、一行）。

下面是一系列发生的步骤，完全按照程序执行的时间顺序排列（看看你是否能够在读到它们之前就能猜到这些步骤）：

1. CSplitterWnd 用户通过在 CMDIChildFrame 派生类的 OnCreateClient()成员函数里调用 Create()来初始化动态分割。

2. Create()调用 CreateCommon()，CreateCommon()接着调用 CWnd::CreateEx()，从而为分割窗口创建 Windows 窗口。

3. Create()接着调用 CreateView()，其中 CreateView()用 CRuntimeClass 信息来创建初始的

窗格。

4. 一旦一切都创建了，Windows 就会给窗口发送一个大小消息。然后 RecalcLayout()被调用，并布局好所有的尺寸条和这个窗格。

5. 接着，接收到一个绘画消息，它导致 OnPaint()被调用。OnPaint()通过最终调用 OnDrawSplitter()画出所有的分割窗口的构件。

6. 在分割窗口被创建、大小被改变以及被重新绘制后，什么都不会发生，直到用户和界面进行交互为止。在这个例子中，用户在左下端的分割框上按下了按钮来创建一个垂直分割。

7. OnLButtonDown()处理函数使用点击测试来确定用户已经点击了垂直分割框。一旦 OnLButtonDown()探测到这个情况以后，它就通过调用相应的函数 StartTracking()来激活跟踪。

8. 在用户四处移动鼠标来放置垂直分割条时，OnMouseMove()会被调用，它在跟踪时负责处理分割窗口中的阴影条的绘制。

9. 当用户放下分割条时，OnLButtonUp()消息处理函数调用 StopTracking()来停止所有的跟踪，并调用 SplitColumn()。

10. SplitColumn()首先检查是否有足够的内存空间，然后通过调用 CreateView()来创建一个新的滚动条，并在分割条的右边创建一个新的窗格。在新的窗格被创建后，SplitColumn()会调用 RecalcLayout()。

11. RecalcLayout()更新原来窗格的位置（它被分成了两半）和所有其他分割窗口组件的位置。RecalcLayout()使得分割条无效，从而引发绘画消息。

12. OnPaint()被调用，所有的新的分割组件在它们的正确位置上被重新绘制。RecalcLayout()作了所有的工作。

我敢打赌，在你第一次通读 Scribble 指南时，你决不会想到 MFC 在背后为你做了这么多的工作。

更多的信息

到目前为止，本章已经介绍了大量的 CSplitterWnd 信息来让你慢慢消化。如果你真的对 CSplitterWnd 非常感兴趣，并且想知道更多的东西，可以遵照下面的建议来进一步深入了解 CSplitterWnd：

- CSplitterWnd 实现中另一个比较有趣的部分是同步滚动。要分析同步滚动，可以从 DoScroll 和 DoScrollBy()开始。
- 当用户双击分割条时会发生什么事情呢？
- 当用户双击分割框时会发生什么事情呢？
- 什么键序列用于停止跟踪？又是什么键序列会用在跟踪上呢？
- 除了 Create()里的断言语句，什么是限制动态分割窗口最多只能有 4 个窗格的技术限制呢？

现在，我们已经明白了 MFC 是如何实现它最复杂的增强型用户界面类了，让我们继续下一组类的学习：MFC 控制栏。

MFC 的 CControlBar 体系结构

回到 MFC 4.0 以前的那些日子，那时的 CControlBar 家族比现在的要有趣得多。那是在微软为了使用 Windows 通用控件而重新编写 CStatusBar 类和 CToolBar 类之前。MFC 曾经为工具栏和状态栏做了所有的绘画、窗格的管理、位图控制以及其他很多工作；现在这些有趣的工作都由通用控件来做了。

但是不要惊慌！我们已经在这些类的新的实现里发现，CControlBar 家族内部的一些方面其实和原来的实现同样有趣。在本节里我们会主要来看看 CControlBar，以及它是如何为它的派生类提供停放（docking）和状态存储特性。

实际上，在深入分析 MFC 的程序清单后，我们会惊奇地发现 MFC 在内部用了几个（4个）没有文档说明的类来实现这些特性。但是 CControlBar 并不只是应用了这些没有文档说明的类：还有许多内部使用的有趣的技术，我们一定会一一说明。

已经有足够的积累了！让我们开始分析吧。

开始学习 CControlBar

正如已经提到的，在本节里我们将分析 CControlBar 是如何提供停放功能和工具栏的“可持续性”。在我们深入研究 CControlBar 之前，首先简要介绍一下 CControlBar 家族，并简单地说明如何使用控制栏。

使用 CControlBar 及其家族

CControlBar 是为控制栏定义通用操作的基类。一个定义良好的控制栏通常是沿着框架窗口的一边对齐的特殊窗口。一些控制栏可以被停放，这意味着它们可以从原来的位置上被拖动，并放置到框架窗口里面或者外面的任何地方。控制栏可以看作是窗口的包容器，就像是按钮、复合框或者是让应用程序进行绘画的一块区域。

在 MFC 里，控制栏能够将它们的位置信息存放到应用程序的 INI 文件里，或者存放到注册表里。当应用程序重新启动时，控制栏就会自动恢复到用户最后一次退出应用程序时的状态，这样，用户所作的定制就能被保存下来了。

你也许还从来没有听说过 CControlBar，因为它的派生类比这个基类本身要通用得多。CControlBar 的派生类如下所示：

- CDialogBar——用来显示一组使用对话框模版创建的控件。MFC 的打印预览 CPreviewView（参看第 8 章）中的按钮条就是 CDialogBar 的一个例子。在这种情况下，CDialogBar 就像 Windows 窗口的包容器一样。
- COleResizeBar——让用户重新设置 OLE 对象的大小。
- CStatusBar——包含了与用户进行交互的状态信息的“框架”。CStatusBar 框架由 Windows 通用控件来绘制并被其控制。
- CToolBar——包含被位图化的按钮和分离器（separator）。

AppWizard 在默认情况下会为你的应用程序“自动”加上一个工具栏和状态栏。AppWizard

按照下面的步骤来添加控制栏：

1. AppWizard 在你的“main”框架窗口中加入数据成员：

```
protected:  // control bar embedded members
    CStatusBar  m_wndStatusBar;
    CToolBar    m_wndToolBar;
```

2. 在 Frame 派生类的 OnCreate()里，AppWizard 创建了控制栏。工具栏自动将 IDR_MAINFRAME 资源中用 VC++资源编辑器创建的位图图形装载进来：

```
if (!m_wndToolBar.Create(this) || !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    return -1;      // fail to create
```

状态栏框架或者指示器都是通过由 AppWizard 放置在框架派生类的.CPP 文件开头的静态数组被初始化的。在 Frame 派生类的 OnCreate()里调用 Create()后，这个静态数组然后就被传递给了 CStaticBar。

```
if (!m_wndStatusBar.Create(this) ||
    !m_wndStatusBar.SetIndicators(indicators, sizeof(indicators)/
        sizeof(UINT)))
    return -1;      // fail to create
```

3. 然后，AppWizard 为工具栏设置某些风格位标志。在下面的例子中，工具提示(tool tip)、即时状态栏更新以及动态尺寸都与原来的风格进行了或操作。

```
m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
    CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);
```

4. 最后，AppWizard 通过加入 3 行代码启用停放功能。开始的两个调用看上去似乎是重复的，但第一个是 CToolBar::EnableDocking()，第二个是 CFrameWnd::EnableDocking()。CBRS_ALIGH_ANY 是另一个控制栏风格，它表明你希望控制栏可以停放到框架窗口的任何地方。

```
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);
```

好了！4 个步骤之后，你就添加了一个可停放的工具栏和状态栏。除了可停放，工具栏还支持工具提示，并且在状态栏里还会对工具栏按钮显示一些解释（CBRS_FLYBY）。如果你真的愿意坚持到最后，在另外两个步骤之后就可以存储工具栏的状态了。

5. 在 CFrame 派生类的 OnClose()里以一个键（key）为参数调用 SaveBarState()，将工具栏的状态保存下来：

```
CMainFrame::OnClose()
{
    //...
    SaveBarState("ControlBarState");
    //...
}
```

6. 在 OnCreate()里装载工具栏状态（当然是在创建了工具栏之后）。一定要使用同 SaveBarState()里一样的关键字。

```
CMainFrame::OnCreate()
{
    //... create status/toolbar, etc..
    LoadBarState("ControlBarState");
    // ...
}
```

你甚至可以通过调用 DockControlBar()和 FloatControlBar()来动态使得控制栏可停放或者不可停放。

这只是 CControlBar 家族的一个简要的介绍。在在线文档里你会读到更多关于它们的信息。让我们首先来看看在在线文档里你不会找到的一些东西。

剖析 CControlBar

只要你好好想想，就会发现如果可停放功能要你自己实现，实在会是一件非常复杂的事情。好几个让你头疼的问题马上就会出现：

- MFC 是如何让框条浮出主窗口的呢？
- MFC 是如何决定将框条停放在哪里的呢？
- MFC 是如何实现控制栏的阴影区随着不同的目标而切换方向这种很巧妙的效果的呢？
- MFC 是如何将窗口四处移动，从而使得它和一个最近被停放的框条进行匹配呢？
- MFC 是如何在控制栏被停放时画出那些精巧的、轮廓分明的边界的呢？

要回答这些以及其他许多关于控制栏的问题，让我们首先来看看控制栏的停放是如何被实现的。

在深入了解停放功能之前，让我们快速地看看以前从没看过的 CControlBar 的声明部分。CControlBar 是在 AFXEXT.H 中声明的；程序清单 9-13 包含了声明中比较有趣的部分。顺便提一句，在所有 MFC 类中，CControlBar 是实现得最分散的类之一。实现中的绝大部分在 BARCORE.CPP 里，但 CControlBar 还有一些部分实现在 BARDOCK.CPP、DOCKSTAT.CPP 以及 WINPERM.CPP 里。

表 9-1 是与 CControlBar 实现相关的其他的一些源文件。

表 9-1 与 CControlBar 实现相关的其他源文件

文件	内容
BARDLG.CPP	CDialogBar 实现
BARDOCK.CPP	CControlBar 停放辅助程序
BARSTAT.CPP	CStatusBar 实现
BARTOOL.CPP	CToolBar 实现
DOCKSTAT.CPP	CControlBar 停放/持续性辅助程序
DOCKCONT.CPP	CControlBar 拖放辅助程序

程序清单 9-13 CControlBar 的实现细节（在文件 AFXEXT.H 中）

```
class CControlBar : public CWnd
{
// Construction **omitted
// Attributes **omitted
// Operations **omitted
// Overridables **omitted
// Implementation **some omitted
public:
    int m_cxLeftBorder, m_cxRightBorder;
    int m_cyTopBorder, m_cyBottomBorder;
    int m_cxDefaultGap;          // default gap value
    UINT m_nMRUWidth;   // For dynamic resizing.
    int m_nCount;
    void* m_pData;
    enum StateFlags { delayHide = 1, delayShow = 2, tempHide = 4,
        statusSet = 8 };
    UINT m_nStateFlags;
    // support for docking
    DWORD m_dwStyle;    // creation style (used for layout)
    DWORD m_dwDockStyle;// indicates how bar can be docked
    CFrameWnd* m_pDockSite;
    CDockBar* m_pDockBar;
    CDockContext* m_pDockContext;
    virtual void DoPaint(CDC* pDC);
    void DrawBorders(CDC* pDC, CRect& rect);
    void EraseNonClient();
    void GetBarInfo(CControlBarInfo* pInfo);
    void SetBarInfo(CControlBarInfo* pInfo, CFrameWnd* pFrameWnd);
    void ResetTimer(UINT nEvent, UINT nTime);
// Message handlers
    afx_msg void OnTimer(UINT nIDEvent);
    DECLARE_MESSAGE_MAP()
    friend class CFrameWnd;
    friend class CDockBar;
};
```

再看看程序清单 9-13。第一眼看过去，你也许会错过 3 个没有文档说明的类：CDockBar、CDockContext 以及 CControlBarInfo。我们会在后面详细介绍它们（还有 2 个没有文档说明的类没有看到）。在我们继续之前，想首先向你介绍一些数据成员，这样在以后的分析中，当你看到它们跳出来时就不会很奇怪了。

数据成员

- m_cxLeftBorder/ m_cyLeftBorder——在构造函数里用硬编码（hard-coded）将此变量赋为 6。各种 CControlBar 的派生类为了有不同的界面效果会改变这些值。
- m_cyTopBorder/ m_cyBottomBorder——在构造函数里用硬编码将此变量赋为 1。各种 CControlBar 的派生类为了有不同的界面效果会改变这些值。
- m_cxDefaultGap——在构造函数里用硬编码将此变量赋为 2。不会被改变。
- m_nMRUWidth——在构造函数里此变量被赋为 32767。该数据成员会被更新为控制栏的最后宽度。控制栏的布局逻辑在某些情况下会使用该成员。不同的布局模式在

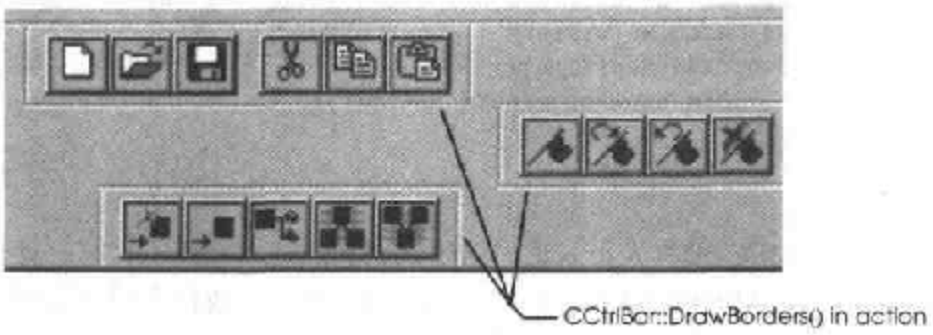


图 9-6 控制栏边界的特写镜头

CControlBar 对停放功能的实现

由 AppWizard 添加的特定于停放功能的代码行看起来是开始进行分析的一个好地方，所以，让我们从这里开始吧。AppWizard 添加的第一行代码是以 CBRS_ALIGH_ANY 为参数调用 CControlBar::EnableDocking()。程序清单 9-14 包含了这个成员函数的伪代码。

程序清单 9-14 CControlBar::EnableDocking()的伪代码（在文件 BARDOCK.CPP 中）

```
void CControlBar::EnableDocking(DWORD dwDockStyle)
{
    //**omitted asserts on argument styles
    m_dwDockStyle = dwDockStyle;
    if (m_pDockContext == NULL)
        m_pDockContext = new CDockContext(this);
    if (m_hWndOwner == NULL)
        m_hWndOwner = ::GetParent(m_hWnd);
}
```

EnableDocking()初始化一些数据成员，从而为 CControlBar 的停放做好准备。首先，EnableDocking()将停放风格参数存储到 m_dwDockStyle 数据成员里。

接着，EnableDocking()创建一个新的 CDockContext 对象，并将它存储到 m_pDockContext 数据成员里。最后，EnableDocking()将::GetParent()的返回结果存放到 m_hWndOwner 里。我们知道你也许对 CDockContext 有点好奇，但现在讨论这个类还为时过早。现在，你只要记住它是在 EnableDocking()里被初始化的，这意味着它在以后的章节中非常重要。

在调用 CControlBar:: EnableDocking()后，AppWizard 接着调用 CFrameWnd:: EnableDocking()。让我们首先看看第二个 EnableDocking()做了些什么吧。程序清单 9-15 里包含了它的伪代码。

程序清单 9-15 CFrameWnd:: EnableDocking()的伪代码（在文件 WINFRM2.CPP 中）

```
void CFrameWnd::EnableDocking(DWORD dwDockStyle)
{
    m_pFloatingFrameClass = RUNTIME_CLASS(CMiniDockFrameWnd);
    for (int i = 0; i < 4; i++){
        if (dwDockBarMap[i][1] & dwDockStyle & CBRS_ALIGN_ANY){
            CDockBar* pDock = (CDockBar*)GetControlBar(dwDockBarMap[i][0]);
            if (pDock == NULL){
                pDock = new CDockBar;
```

```
            if (!pDock->Create(this, WS_CLIPSIBLINGS|WS_CLIPCHILDREN|
                WS_CHILD|WS_VISIBLE | dwDockBarMap[i][1],
                dwDockBarMap[i][0]))
                    AfxThrowResourceException();
        }
    }
}
```

对于绝大多数的 MFC 编程人员来说，这段代码真的是太疯狂了！第一行代码存储没有文档说明的 MFC 类（CMiniDockFrameWnd）的 RTCI 信息。我们在后面会详细介绍它。接着，EnableDocking()对 dwDockBarMap 进行从 0~3 的循环，并得到 CDockBar（又是一个没有文档说明的类）的指针。如果没有指针，EnableDocking()会从堆里创建一个，然后接着调用 Create()成员函数。

你也许还是头疼。WINFRM2.CPP 里的二维数组对于你应该有所帮助：

```
const DWORD CFrameWnd::dwDockBarMap[4][2] =
{
    { AFX_IDW_DOCKBAR_TOP,      CBRS_TOP    },
    { AFX_IDW_DOCKBAR_BOTTOM,   CBRS_BOTTOM },
    { AFX_IDW_DOCKBAR_LEFT,     CBRS_LEFT   },
    { AFX_IDW_DOCKBAR_RIGHT,    CBRS_RIGHT  },
};
```

有了这些知识，你就可以看到 EnableDocking()正在对这个表进行遍历，并且根据 dwDockStyle 标志检查 CBRS_XXX（第二列）。在我们的例子中，dwDockStyle 是 CBRS_ALIGH_ANY，4 个 CDockBars 都被创建了。如果我们改变 EnableDocking()的 CBRS_ALIGH_ANY 参数，例如 CBRS_TOP，那么就只有一个 CDockBars 能被创建。

EnableDocking()这样写的好处在于，如果 4 个 CDockBars 都已经被创建了，它们不会被重新创建，因为 GetControlBar()会返回这个框条，并且 Create()就不会被调用了。

你已经推断出了 CDockBar 是做什么的吗？猜对了：CDockBar 是为可停放工具栏准备的“着陆框条”。图 9-7 显示了 4 个 CDockBars（ID 是 AFX_IDW_DOCKBARTOP，依次类推）。在后面我们还会详细介绍 CDockBar。

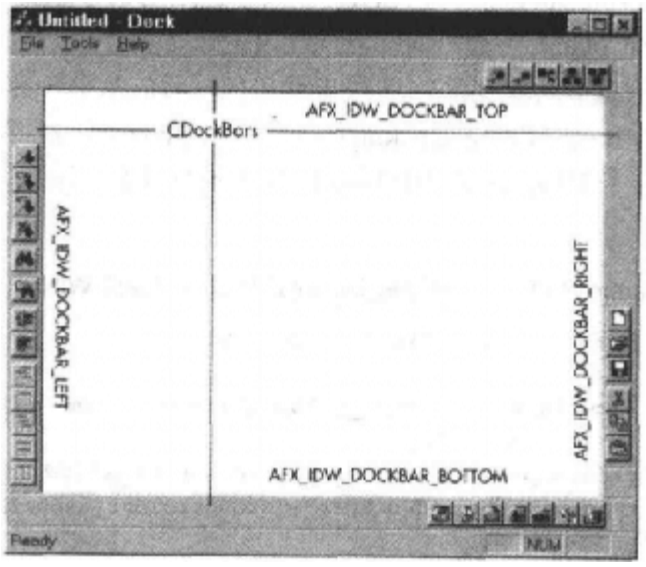


图 9-7 CDockBar 的位置以及 ID

这个函数中让人不安的一点是，“新的”CDockBar 对象被分配内存，然后创建了这个对象，接着这个对象就好像丢失了一样，因为没有任何成员函数指向它们。实际上，CFrameWnd 有一个 m_listControlBars，它是 CObject 指针的程序清单。CFrameWnd 用这个程序清单来为它存储所有的控制栏。控制栏是在 CControlBar::OnCreate()里被加到这个程序清单上来的。所以，不要慌张，毕竟 MFC 会对所有的 CDockBars 进行跟踪。

现在，让我们看看 AppWizard 增加的第三行代码，并且看看我们发现了其他什么非常酷的实现机制。在调用 CFrame::EnableDocking()后，AppWizard 会添加对 CFrameWnd::DockControlBar()的调用，如下面代码所示：

```
DockControlBar(&m_wndToolBar);
```

程序清单 9-16 包含了 CFrameWnd:: DockControlBar()成员函数的伪代码。

程序清单 9-16　两个 CFrameWnd:: DockControlBar()成员函数（在文件 WINFRM2.CPP 中）

```
void CFrameWnd::DockControlBar(CControlBar* pBar, UINT nDockBarID,
    LPCRECT lpRect)
{
    CDockBar* pDockBar = (nDockBarID == 0) ? NULL :
        (CDockBar*)GetControlBar(nDockBarID);
    DockControlBar(pBar, pDockBar, lpRect);
}
void CFrameWnd::DockControlBar(CControlBar* pBar, CDockBar* pDockBar,
    LPCRECT lpRect)
{
    if (pDockBar == NULL){
        for (int i = 0; i < 4; i++){
            if ((dwDockBarMap[i][1] & CBRS_ALIGN_ANY) == (pBar->m_dwStyle &
               CBRS_ALIGN_ANY)){
                pDockBar = (CDockBar*)GetControlBar(dwDockBarMap[i][0]);
                break;
            }
        }
    }
    pDockBar->DockControlBar(pBar, lpRect);
}
```

AppWizard 生成的 DockControlBar()调用会导致对程序清单 9-16 中的第一个 DockControlBar()的调用。nDockBarID 的默认值为 0，lpRect 的默认值为 NULL。第一个 DockControlBar()会对这种情况进行检查，并在调用第二个 DockControlBar()之前将 pDockBar 设置为 NULL。

第二个 DockControlBar()再一次对 dwDockBarMap 表进行遍历，并且通过调用 GetControlBar()来得到 CDockBar 类型的指针。一旦指针得到了，DockControlBar()会调用 CDockBar:: DockControlBar()，并将 CControlBar 指针和矩形参数传递过去。

现在，我们已经学了比较多的东西了！我们已经知道了使得 CControlBar 具有可停放功能所需要的一切“催化剂”。我们也知道了 CDockBar（着陆场）、CCtrlBar 指针（飞机）甚至指定目标的矩形（跑道号？）。CDockBar:: DockControlBar()真的是非常长。所以程序清单 9-17 里只包含了一个相当简略的版本。实际代码在 BARDOCK.CPP 文件里。

程序清单 9-17 CDockBar:: DockControlBar()的简略版本（在文件 BARDOCK.CPP 中）

```
void CDockBar::DockControlBar(CControlBar* pBar, LPCRECT lpRect)
{
    CRect rectBar;
    int nPos = -1;
    pBar->GetWindowRect(&rectBar);
    // **omitted, does some stuff with styles here to 'mark' it as
    //  a docked bar and not a floating one.
    if (lpRect != NULL) {
        // insert into appropriate row
        CRect rect(lpRect);
        ScreenToClient(&rect);
        CPoint ptMid(rect.left + rect.Width()/2, rect.top + rect.Height()/2);
        nPos = Insert(pBar, rect, ptMid);
        // position at requested position
        pBar->SetWindowPos(NULL, rect.left, rect.top, rect.Width(),
            rect.Height(), SWP_NOZORDER|SWP_NOACTIVATE|SWP_NOCOPYBITS);
    }
    else {    // always add on current row, then create new one
        m_arrBars.Add(pBar);
        m_arrBars.Add(NULL);
        // align off the edge initially
        pBar->SetWindowPos(NULL, -afxData.cxBorder2, -afxData.cyBorder2,
            0, 0, SWP_NOSIZE|SWP_NOZORDER|SWP_NOACTIVATE|SWP_NOCOPYBITS);
    }
    // attach it to the docking site
    if (pBar->GetParent() != this)
        pBar->SetParent(this);
    if (pBar->m_pDockBar == this)
        pBar->m_pDockBar->RemoveControlBar(pBar, nPos);
    else if (pBar->m_pDockBar != NULL)
        pBar->m_pDockBar->RemoveControlBar(pBar);
    pBar->m_pDockBar = this;
    // remove any place holder for pBar in this dockbar
    RemovePlaceHolder(pBar);
    // get parent frame for recalc layout
    CFrameWnd* pFrameWnd = GetDockingFrame();
    pFrameWnd->DelayRecalcLayout();
}
```

现在要讨论最大的问题了！我们一直都在想，MFC 是如何决定将用户四处拖动的可停放框条停放在哪里，以及是如何停放的呢？DockControlBar()是问题的关键。在通过 GetWindowRect()得到要停放的框条的尺寸后，DockControlBar()有两种不同的放置方案。

在第一种方案里，DockControlBar()被调用时，参数是有效的 lpRect，它是对停放目的地所给出的一个提示。DockControlBar 得到该矩形，将它转换到用户坐标里，然后计算出一个中间点，接着调用 Insert()。在调用 Insert()后，DockControlBar()调用 SetWindowPos()将控制栏放置到新算出来的位置上。

在第二种方案里，lpRect 是空的，这也是我们正在跟踪的当前方案。记住，在 OnCreate()里，调用 DockControlBar()时用的是 lpRect 的默认值，该默认值为 NULL。当 lpRect 参数为 NULL 时，DockControlBar()将控制栏和 NULL 加到 m_arrBars（意味着一个框条数组）数据

成员里。接着调用 SetWindowPos()将框条放到窗口的边缘。如果你运行的是 AppWizard 生成的应用程序，你会注意到在最开始时，工具栏都是放在左边的。这就是 DockControlBar()的第二种方案。

非常重要！现在我们要学习本节里最重要的内容了，所以注意力一定要非常集中。在任何一种方案完成之后，要被停放的框条被正确地放置好了，但是它仍然是浮动的。要将浮动的控制栏并入 CDockBar，DockControlBar()会调用 CWnd::SetParent()。SetParent()会将 CControlBar 从浮动窗口的子窗口转换成 CDockBar 的子窗口。

如果你仔细想想，你会发现 SetParent()API 函数的功能实在是非常强大。它能够让你花费非常小的代价来随意改变应用程序中窗口的层次关系。SetParent()对于每个 MFC 开发人员来说都是一个非常重要的工具，是应该保存在你脑海中的 MFC 工具框里的。

在 SetParent()之上的检查是为了处理下面的这种情况：用户将控制栏四处拖动，但实际上还是在同一个 CDockBar 里，这时就不用重新放置控制栏了。

在调用 SetParent()结束后，DockControlBar()会检查 m_pDockBar，看是否用户将控制栏从一个CDockBar拖拉到了另一个CDockBar(例如将框条从CDockBar的左边拖拉到了右边)。如果确实是这种情况，DockControlBar()会调用 RemoveControlBar()将控制栏从原来的 CDockBar 里删除。

在快结束时，DockControlBar()会调用 RemovePlaceHolder()，它会对一些内部的 CDockBar 结构进行清理。最后，DockControlBar()调用 DelayRecalcLayout()，参数为 GetDockingFrame()函数返回的 CFrameWnd 指针。

现在，你知道了 MFC 是如何实现可停放工具栏的停放功能了！在我们继续学习控制栏是如何从停放状态转换到浮动状态之前，让我们来看看已经非常熟悉的 CDockBar 的声明。CDockBar 的声明在我们喜爱的 MFC 头文件 AFXPRIV.H 里，为了方便，你也可以直接看程序清单 9-18 中的代码。

程序清单 9-18 CDockBar 的声明（在文件 AFXPRIV.H 中）

```
class CDockBar : public CControlBar
{
    DECLARE_DYNAMIC(CDockBar)
// Construction and Attributes
public:
    CDockBar(BOOL bFloating = FALSE);  //true if attached to
                                       //CMiniDockFrameWnd
    BOOL Create(CWnd* pParentWnd, DWORD dwStyle, UINT nID);
    BOOL m_bFloating;
// Operations
    void DockControlBar(CControlBar* pBar, LPCRECT lpRect = NULL);
    void ReDockControlBar(CControlBar* pBar, LPCRECT lpRect = NULL);
    BOOL RemoveControlBar(CControlBar*, int nPosExclude = -1);
    void RemovePlaceHolder(CControlBar* pBar);
// Implementation
public:
    virtual CSize CalcFixedLayout(BOOL bStretch, BOOL bHorz);
    virtual void DoPaint(CDC* pDC);
    void GetBarInfo(CControlBarInfo* pInfo);
    void SetBarInfo(CControlBarInfo* pInfo, CFrameWnd* pFrameWnd);
```

```
    int FindBar(CControlBar* pBar, int nPosExclude = -1);
    void ShowAll(BOOL bShow);
protected:
    CPtrArray m_arrBars;    // each element is a CControlBar
    BOOL m_bLayoutQuery;
    CRect m_rectLayout;
```

数据成员

- m_bFloating——这个标志值说明了 CDockBar 是否是浮动的。到目前为止，我们仅仅看到了非浮动的 CDockBars。
- m_arrBars——从它的名字我们就可以猜出，这个数据成员是一个 CPtrArray，其中数组的指针是 CControlBars。
- m_bLayoutQuery——MFC 窗口的布局逻辑所使用的一个标志。
- m_rectLayout——MFC 窗口的布局逻辑所使用的一个矩形。

成员函数

- Create()——初始化风格，然后调用 CWnd::Create()，参数为_afxWndControlBar 类名。
- DockControlBar()——你知道它的功能！
- ReDockControlBar()——该成员函数快速地将一个浮动的控制栏移回到原来的停放状态和停放位置，反之亦然。当用户在一个没有子窗口也没有子框架的区域内用鼠标双击时，该成员函数会被调用（在 VC++里试一试，你会看到运行中的重定位）。
- RemoveControlBar()——正如名字所蕴涵的，该函数从 CDockBar 里删除一个控制栏。
- RemovePlaceHolder()——CDockBar 将以前的位置信息（占位符）保存在 m_arrBars 程序清单里，以防用户取消拖放操作。在这种情况下，CDockBar 能将框条恢复到原来的位置。我们可以将移动当作一种两次提交操作。
- CalcFixedLayout()——在一个 CDockBar 里将所有的控制栏布置好。需要做一些很复杂的几何运算，以所要求的矩形、实际的矩形以及其他一些因素为基础。CalcFixedLayout()通过调用 DeferWindowPos() API 函数来快速地移走所有的控制栏。该函数由 MFC 窗口布局逻辑调用。
- GetBarInfo()与 SetBarInfo()——在后面会详细介绍。
- FindBar()——在内部的 m_arrBars 数组里寻找某个框条。
- ShowAll()——遍历 m_arrBars 数组，并且显示所有的控制栏。
- Insert()——将一个新的 CControlBar 插入到 m_arrBars 数组里。这些 CControlBars 会以某种特定的顺序进行存储，并且 Insert()会保持这种顺序关系。CDockBar 在该数组中会保存 NULL 指针来表示新的一列。Insert()会考虑保留这种格式。

到这里为止，我们已经分析了 CControlBar 停放功能实现中的 65%的内容。还有两个内容我们还没有分析到：

1. 可停放工具栏是如何以相反的形式进行转换，即从 CDockBar 的子窗口转换到一个浮动的工具栏？

2. 实际的拖拉以及阴影条的绘画是如何实现的，是哪些类做了这些工作？

让我们首先来解决第一个问题。你能猜到是哪个 API 函数来做这些工作的吗？好的，这

实际上是一个很简单的问题。你能猜到当 CControlBar 处于浮动状态时，哪个类是它的父类吗？稍安毋躁，答案会让你大吃一惊！

CControlBar 的浮动实现

每当用户希望某个控制栏处于浮动状态时，CFrameWnd::FloatControlBar()成员函数都会被调用。程序清单 9-19 包含了该函数的伪代码。

程序清单 9-19 CFrameWnd::FloatControlBar()的伪代码（在文件 WINFRM2.CPP 中）

```
void CFrameWnd::FloatControlBar(CControlBar* pBar, CPoint point,
                                DWORD dwStyle)
{
    // **omitted special case where dragging between floating bars
    // **omitted ~ some style manipulations.
    CMiniDockFrameWnd* pDockFrame = CreateFloatingFrame(dwStyle);
    pDockFrame->SetWindowPos(NULL, point.x, point.y, 0, 0,
        SWP_NOSIZE|SWP_NOZORDER|SWP_NOACTIVATE);
    CDockBar* pDockBar =
        (CDockBar*)pDockFrame->GetDlgItem(AFX_IDW_DOCKBAR_FLOAT);
    pDockBar->DockControlBar(pBar);
    pDockFrame->RecalcLayout(TRUE);
    pDockFrame->ShowWindow(SW_SHOWNA);
    pDockFrame->UpdateWindow();
}
```

注意，FloatControlBar()带有 3 个参数：（1）CControlBar 指针，指向将要被浮动的框条；（2）一个点，它是浮动框条的左上角；（3）风格，指定了当控制栏处在浮动状态时的某些风格位。

首先，FloatControlBar()调用 CreateFloatingFrame()，传递的参数是用来创建 CMiniDockFrameWnd 的风格位（又一次是这样！）。CreateFloatingFrame()的基本功能是创建一个 CMiniDockFrameWnd，并且调用 Create()函数。暂且把这些放置起来，我们以后会详细分析 CMiniDockFrameWnd。现在，可以把 CMiniDockFrameWnd 当作一个浮动框架窗口（正如它的名字所蕴涵的）。

在创建 CMiniDockFrameWnd 后，FloatControlBar()将它的位置设置成点参数所指定的地方，使用的函数是 SetWindowPos()。接着，从 CMiniDockFrameWnd 那里得到 CDockBar 指针，并将它存储到 pDockBar 里。在得到 CDockBar 指针后，FloatControlBar()会以类似的方法来调用 DockControlBar()。

好的！这肯定会是一个惊奇。下面是理解到底发生了什么的关键：CMiniDockFrameWnd 有一个子窗口，一个旧的 CDockBar。也就是说，CMiniDockFrameWnd 是浮动窗口的着陆块——就像一艘航空母舰。

通过对固定的和浮动的着陆块使用相同的类，MFC 将停放的主要功能放在 CDockBar::DockControlBar()里实现。在 FloatControlBar()里，SetParent()会被调用，从而使得父窗口从固定的 CDockBar 转换成 CMiniDockFrameWnd 的浮动 CDockBar。

作个总结，FloatControlBar()通过调用 RecalcLayout()来实现 CMiniDockFrameWnd 的 MFC 逻辑布局，而且它也会显示并刷新浮动窗口。从 FloatControlBar()我们可以知道，固定的工具栏与浮动的工具栏之间的惟一区别在于框架窗口。对于固定的 CDockBar，框架是原来的普

通 CFrameWnd。而对于浮动的 CDockBar，框架是神秘的 CMiniDockFrameWnd。

CMiniDockFrameWnd 是 CMiniFrameWnd 的派生类。我们会在本章的末尾介绍 CMiniFrameWnd。现在我们已经知道 CMiniFrameWnd 的功能是在控制栏的周围画出小的边界。下面让我们看看 CMiniDockFrameWnd 在 CMiniFrameWnd 里加进了些什么。

CMiniDockFrameWnd 与 CDockBar 一样，都是在我们喜欢的 MFC 头文件 AFXPRIV.H 里声明的。为了方便起见，在程序清单 9-20 中列出了这些声明。

程序清单 9-20 CMiniDockFrameWnd 的声明（在文件 AFXPRIV.H 中）

```
class CMiniDockFrameWnd : public CMiniFrameWnd
{
    DECLARE_DYNCREATE(CMiniDockFrameWnd)
public:
// Construction
    CMiniDockFrameWnd();
    virtual BOOL Create(CWnd* pParent, DWORD dwBarStyle);
// Operations
    virtual void RecalcLayout(BOOL bNotify = TRUE);
// Implementation
public:
    CDockBar m_wndDockBar;
    //message map entries omitted -- covered in CMiniFrameWnd section later
};
```

从程序清单 9-20 可以看到，CMiniDockFrameWnd 是 CMiniFrameWnd 的一个派生类，它增加了一个定制的 Create()、一个成员函数 RecalcLayout()和一个 CDockBar 类型的成员数据 m_wndDockBar。

我们首先迅速浏览 CMiniDockFrameWnd::Create ()，看 CDockBar 是如何创建的。程序清单 9-21 给出了 Create ()的一个缩减版本。

程序清单 9-21 CMiniDockFrameWnd::Create()的缩减版本（在文件 BARDOCK.CPP 中）

```
BOOL CMiniDockFrameWnd::Create(CWnd* pParent, DWORD dwBarStyle)
{
    //*omitted - recalclayout logic, style settings, etc..
    if (!CMiniFrameWnd::CreateEx(dwExStyle, NULL, &afxChNil, dwStyle,
        rectDefault, pParent))
        return FALSE;
    CMenu* pSysMenu = GetSystemMenu(FALSE);
    pSysMenu->DeleteMenu(SC_SIZE, MF_BYCOMMAND);
    pSysMenu->DeleteMenu(SC_CLOSE, MF_BYCOMMAND);
    pSysMenu->AppendMenu(MF_STRING|MF_ENABLED, SC_CLOSE, strHide);
    // must initially create with parent frame as parent
    if (!m_wndDockBar.Create(pParent, WS_CHILD | WS_VISIBLE | dwStyle,
        AFX_IDW_DOCKBAR_FLOAT))
            return FALSE;
    m_wndDockBar.SetParent(this);
    return TRUE;
}
```

Create()首先调用 CMiniFrameWnd::CreateEx()来创建一个 CMiniFrameWnd 对象。接着，Create()通过改变菜单的元素来定制 CMiniFrameWnd 对象。最后，Create()调用 SetParent()将 CDockBar 的父窗口修改成 CMiniDockFrameWnd。

一旦知道了浮动操作其实是将着陆点从 CFrameWnd 的 CDockBar 改变到 CMiniFrameWnd 的 CDockBar，浮动操作就非常简单了——感谢功能强大的 CDockBar 辅助类。

到目前为止，我们已经覆盖了控制栏停靠功能实现的 90%的内容。现在我们开始学习剩下的 10%：控制栏是如何被拖动的。

CControlBar 的拖动实现

当用户将鼠标移过控制栏并按下鼠标左键时，控制栏就被抓住了，从而拖动操作也就开始了。这将我们带到了 CControlBar::OnLButtonDown()，见程序清单 9-22 所示。

程序清单 9-22 CControlBar::OnLButtonDown()的伪代码（在文件 BARCORE.CPP 中）

```
void CControlBar::OnLButtonDown(UINT nFlags, CPoint pt)
{
    if (m_pDockBar != NULL && OnToolHitTest(pt, NULL) == -1) {
        ClientToScreen(&pt);
        m_pDockContext->StartDrag(pt);
    }
    else
        CWnd::OnLButtonDown(nFlags, pt);
}
```

OnLButtonDown()调用 OnToolHitTest()来证实用户确实是点击了合法的拖动起始点，而不是点击了控制栏的子窗口或者子框架。如果确实是合法的点击，OnLButtonDown()将鼠标的点击点转换为屏幕的坐标，然后调用 CDockContext::StartDrag()。如果用户还没有开始拖动，OnLButtonDown()将被传递到 CWnd。

在程序清单 9-14 的 CControlBar::EnableDocking()里我们看到 CDockContext 被初始化，并被赋给 m_pDockContext 变量。CDockContext 将为 CControlBar 处理拖动操作。

CDockContext 对拖动和跟踪的实现与 CSplitterWnd 的跟踪实现实在是非常类似。主要的不同点在于 CDockContext 在从桌面窗口创建的 DC 上绘画，并且调用 CDC::DrawDragRect()来画出控制栏的阴影条，而不是调用 PatBlt()。CDockContext 也会注意在特定的时候翻转阴影条，向用户显示用户停止拖动操作时工具栏将被放置的位置。

这里我们不会深入地讲解 CDockContext，这将是留给用户的一个练习。CDockContext 还是在 AFXPRIV.H 里声明的。CDockContext 的实现在 DOCKCONT.CPP 里。CDockContext 会记住控制栏原来所处的位置，以防用户取消拖动操作。“MRU”成员数据就是用于这个目的的。

现在你已经完全明白了 CControlBar 的停放功能的实现，下面让我们看看 MFC 是如何实现 CControlBar 状态的保持功能的。

CControlBar 的状态保持

你应该还记得，在 CControlBar 的简要介绍里，我们讲过如果用户程序需要保存并恢复 CControlBar 的状态，用户需要调用两个成员函数。让我们先来看看保存操作。对 CFrameWnd::SaveBarState()的调用会初始化一次保存操作。所有关于 CControlBar 的状态保持代码（包括 SaveBarState()）都在 DOCKSTAT.CPP 里。程序清单 9-23 列出了 SaveBarState()的伪代码。

程序清单 9-23 CFrameWnd::SaveBarState()的伪代码（在文件 DOCKSTAT.CPP 中）

```
void CFrameWnd::SaveBarState(LPCTSTR lpszProfileName) const
{
    CDockState state;
    GetDockState(state);
    state.SaveState(lpszProfileName);
}
```

首先，SaveBarState()创建一个局部的 CDockState 对象，并将它传递给 GetDockState()函数。在调用 GetDockState()得到状态后，SaveBarState()通过调用 CDockState::SaveState()将信息写到磁盘上去。

好的！这里看上去又出现了一个没有文档说明的辅助类！在我们揭开 CDockState 的面纱之前，让我们来看看 CFrameWnd::GetDockState()，从而对 CDockState 的用途有一个感性的认识。程序清单 9-24 包含了伪代码。

程序清单 9-24 CFrameWnd:: GetDockState()的伪代码（在文件 DOCKSTAT.CPP 中）

```
void CFrameWnd::GetDockState(CDockState& state) const
{
    state.Clear();
    POSITION pos = m_listControlBars.GetHeadPosition();
    while (pos != NULL) {
        CControlBar* pBar = (CControlBar*)m_listControlBars.GetNext(pos);
        CControlBarInfo* pInfo = new CControlBarInfo;
        pBar->GetBarInfo(pInfo);
        state.m_arrBarInfo.Add(pInfo);
    }
}
```

从 GetDockState()里我们能推断出 CDockState 的许多内容。首先，GetDockState()对 state 参数作一次清理操作，从而使它可以接收一些新的控制栏实现信息。接着，GetDockState()在控制栏的 CFrameWnd 程序清单中循环。

对于框架窗口中的每一个控制栏，都会有一个新的 CControlBarInfo 对象被创建，并被传递到 CControlBar::GetBarInfo()。在 GetBarInfo()调用完成后，CControlBarInfo 对象被加入到 CDockState 内部的 CControlBarInfo 数组里。

天哪！又是一个没有文档说明的 CControlBar 辅助类。从目前我们所见到的小的代码片断来看，CDockState 似乎是用来记录多个控制栏的状态信息的。各个控制栏的独立的状态信息看来是被记录在 CControlBarInfo 辅助类里。也就是说，对于每个框架来说都有一个 CDockState，而对于框架里的每个 CControlBar，都会有一个 CControlBarInfo。

揭示 CDockState 与 CControlBarInfo

首先，我们会看看这两个新类的声明，然后会分析 CControlBar::GetBarInfo()和 CDockState::SaveState()都做了些什么。程序清单 9-25 中是缩减后 CDockState 的声明，在文件 AFXPRIV.H 中。

程序清单 9-25 CDockState 的缩减声明（在文件 AFXPRIV.H 中）

```
class CDockState : public CObject
{
```

```
    DECLARE_SERIAL(CDockState)
    CDockState();
public:
// Attributes
    CPtrArray m_arrBarInfo;
public:
// Operations
    void LoadState(LPCTSTR lpszProfileName);
    void SaveState(LPCTSTR lpszProfileName);
    void Clear(); //deletes all the barinfo's
    DWORD GetVersion();
    // Scaling Operations
    void ScalePoint(CPoint& pt);
    void ScaleRectPos(CRect& rect);
    CSize GetScreenSize();
    void SetScreenSize(CSize& size);
// Implementation
protected:
    BOOL m_bScaling;
    CRect m_rectDevice;
    CRect m_rectClip;
    CSize m_sizeLogical;
    DWORD m_dwVersion;
public:
    ~CDockState();
    virtual void Serialize(CArchive& ar);
};
```

注意，在程序清单 9-25 里，CDockState 是 CObject 的派生类，它支持序列化（serialization）（关于序列化的详细内容请参考第 5 章）。记住，这个类包含了 CFrameWnd 存储一组控制栏的状态时所需的所有信息。这个状态由一组数据成员来表示，并被一组成员函数所控制。下面是每个成员的功能的一个简略说明。

首先，CDockState 的数据成员：

- m_arrBarInfo——一个 CPtrArray，用来存储框架中的每个控制栏所对应的 CControlBarInfo。
- m_bScaling——用来指明 CDockState 是否应调整它的值的一个标志（在后面会更详细地解释）。
- m_rectDevice——存储显示器的大小。显示器的大小由::GetSystemMetrics()调用决定，参数为 SM_CXSCREEN/CYSCREEN。
- m_rectClip——存储显示器减去图标的大小。图标的大小由::GetSystemMetrics()调用决定，参数为 S_CXICON/CYICON。m_rectClip 在调整大小时会被用到。
- m_sizeLogical——存储 m_rectDevice 被调整大小之前的逻辑窗口大小。
- m_dwVersion——序列化所用到的版本号。在 MFC 4.0 里，版本号为 2。

在我们分析 CDockState 的成员函数之前，先来看看调整大小。为什么 CDockState 需要能够改变大小呢？如果用户在显示器设置成 VGA 时运行基于 MFC 的应用程序，并将控制栏存储到某一个地方。然后，当下一次用户又运行该程序，并且将显示器改成 1024*780。这时如果 CDockState 将所有的控制栏都恢复到它们原来所在的位置，它们会被挤在显示器的一个

角落里，因为这就是原来在 VGA 分辨率下的坐标，而现在是在一个更高的分辨率下运行。

CDockState 通过存储“另存为(saved as)”分辨率来解决这个问题。这能够让 CDockState 判断出分辨率已经改变了，需要重新动态调整控制栏的位置，使得它们可以适应显示器的新坐标。在我们的这个例子里，如果用户在 VGA 显示器的中央有一个浮动的控制栏，CDockState 的调整大小功能会自动将该浮动条放置到 1024*780 显示器的中央。相当酷的功能！

现在让我们看看 CDockState 的成员函数：

- LoadState()——将 CDockState 的信息存储到一个 INI 文件里或存储到注册表中。
- SaveState()——从 INI 文件或注册表中检索 CDockState 信息。
- Clear()——通过清除 m_arrBarInfo 里所有的 CControlBarInfo 元素来清空 CDockState。
- GetVersion()——得到 m_dwVersion 数据成员。
- ScalePoint()——用旧的分辨率和新的分辨率来调整指定点。m_rectClip 的作用是使得该点不会被调整到可见的窗口之外。
- ScaleRectPos()——用类似 ScalePoint()的算法来调整一个矩形。
- GetScreenSize()——从 m_rectDevice 里获取显示器的大小。
- SetScreenSize()——当该函数在 CDockState 被恢复的过程中被调用时，如果被恢复的 m_rectDevice 和当前的显示器的大小不一样，则调整大小开关 m_bScaling 将被打开。
- Serialize()——将 CDockState 序列化到某个文档中。如果希望将某些信息存储到某个文档，而不是存储到 INI 文件或者是注册表时，这个函数将被派上用场。Serialize()会将所有的数据成员序列化出去。当 m_arrBarInfo 数组被写出时，CPtrArray 将为每个 CControlBarInfo 调用:: Serialize()函数。正如我们将会看到的，CControlBarInfo 也支持序列化。

好的！一旦开始分析 CDockState，你会发现它是一个功能相当强大的、小的辅助类。现在，让我们也来看看 CControlBarInfo，并且来看看它是否有一些新颖的功能。

程序清单 9-26 列出了 CControlBarInfo 的声明。正如你能猜到的，CControlBarInfo 的声明同样在文件 AFXPRIV.H 里。

程序清单 9-26 CControlBarInfo 缩减后的声明（在文件 AFXPRIV.H 中）

```
class CControlBarInfo
{
public:
// Implementation
    CControlBarInfo();
// Attributes
    UINT m_nBarID;       // ID of this bar
    BOOL m_bVisible;     // visibility of this bar
    BOOL m_bFloating;    // whether floating or not

    BOOL m_bHorz;        // orientation of floating dockbar
    BOOL m_bDockBar;     // true if a dockbar
    CPoint m_pointPos;   // topleft point of window
    UINT m_nMRUWidth;    // MRUWidth for Dynamic Tool bars
    BOOL m_bDocking;     // TRUE if this bar has a DockContext
    UINT m_uMRUDockID;   // most recent docked dockbar
    CRect m_rectMRUDockPos;  // most recent docked position
    DWORD m_dwMRUFloatStyle; // most recent floating orientation
```

```
    CPoint m_ptMRUFloatPos;  // most recent floating position
    CPtrArray m_arrBarID;    // bar IDs for bars contained within this one
    CControlBar* m_pBar;     // bar which this refers to (transient)
    void Serialize(CArchive& ar, CDockState* pDockState);
    BOOL LoadState(LPCTSTR lpszProfileName, int nIndex,
        CDockState* pDockState);
    BOOL SaveState(LPCTSTR lpszProfileName, int nIndex);
};
```

在程序清单 9-26 里，注意，CControlBarInfo 并不是 CObject 的派生类。下面是它的各个成员的功能的简介：

- m_nBarID——控制栏的 ID。
- m_bVisible——指明控制栏是否可见的标志。
- m_bFloating——指明控制栏是被停放还是处于浮动状态的标志（也就是说，它的父窗口是 CFrameWnd 还是 CMiniDockFrameWnd？）。
- m_bHorz——指明控制栏是水平的还是垂直的标志。
- m_bDockBar——它是一个 CDockBar 吗？
- m_pointPos——窗口左上点的坐标。
- m_nMRUWidth——镜像 CControlBar::m_nMRUWidth 数据成员。
- m_bDocking——如果 CControlBar 有 CDockContext，该变量为 TRUE。
- m_uMRUDockID——最经常使用的被停放工具栏的 ID。
- m_rectMRUDockPos——最经常使用的被停放工具栏的位置。
- m_dwMRUFloatStyle——最经常使用的浮动风格。
- m_ptMRUFloatPos——最经常使用的浮动位置。
- m_arrBarID——被包含的所有框条的 ID。例如，被停放的框条能包含多个 CControlBar。
- m_pBar——指向 CControlBarInfo 描述的控制栏的指针。
- LoadState()/SaveState()——将 CControlBarInfo 的数据成员存储到 INI 文件中或者注册表中。
- Serialize()——将 CControlBarInfo 的数据成员存储到某个文档中。

现在我们已经知道了 CControlBarInfo 数据成员有哪些了，这时一个相当大的体系结构问题出现了。看起来 CControlBarInfo 是重复了 CControlBar 的数据成员。为什么不是在 CControlBar 中加入 LoadState()/SaveState()以及 Serialize()成员函数，而要创建一个“镜像”类，每当 CControlBar 变化时，它都需要更新？

CControlBar 的状态保持功能是如何工作的：存储

现在你已经了解了 CDockState 和 CControlBarInfo 的全部内容，下面让我们看看 MFC 是如何使用它们的。

在程序清单 9-24 里，CFrameWnd::GetDockState()创建了 CControlBarInfo 对象，然后将它传递给了 CControlBar::GetBarInfo()。下面让我们看看 GetBarInfo()对 CControlBarInfo 对象做了些什么操作。程序清单 9-27 列出了 GetBarInfo()的伪代码。

程序清单 9-27 CControlBar::GetBarInfo()的伪代码（在文件 DOCKSTAT.CPP 中）

```
void CControlBar::GetBarInfo(CControlBarInfo* pInfo)
{
    pInfo->m_nBarID = _AfxGetDlgCtrlID(m_hWnd);
    pInfo->m_pBar = this;
    pInfo->m_bVisible = IsVisible(); // handles delayed showing and hiding
    pInfo->m_nMRUWidth = m_nMRUWidth;
    if (m_pDockBar != NULL) {
        CRect rect;
        GetWindowRect(&rect);
        m_pDockBar->ScreenToClient(&rect);
        pInfo->m_pointPos = rect.TopLeft();
        pInfo->m_bDocking = TRUE;
        pInfo->m_uMRUDockID = m_pDockContext->m_uMRUDockID;
        pInfo->m_rectMRUDockPos = m_pDockContext->m_rectMRUDockPos;
        pInfo->m_dwMRUFloatStyle = m_pDockContext->m_dwMRUFloatStyle;
        pInfo->m_ptMRUFloatPos = m_pDockContext->m_ptMRUFloatPos;
    }
}
```

正如你所猜到的，GetBarInfo()基本上是将 CControlBar 的状态信息（数据成员）拷贝到 CControlBarInfo 类中。

在 GetDockState()创建了 CControlBarInfo 对象后，为表格里的每个 CControlBar 调用 GetBarInfo()将 CControlBar 的状态信息拷贝到该对象里。在调用 GetDockState()后，CFrameWnd::SaveBarState()（参考程序清单 9-23）调用 DockState::SaveState()来保存状态信息。

在程序清单 9-23 里，我们看到在调用 GetDockState()后，CFrameWnd 调用了 CDockState::SaveState()。

下面让我们看看 CDockState::SaveState()做了些什么。程序清单 9-28 包含了它的伪代码。

程序清单 9-28 CDock State:: SaveState()缩减后的代码（在文件 DOCKSTAT.CPP 中）

```
void CDockState::SaveState(LPCTSTR lpszProfileName)
{
    CWinApp* pApp = AfxGetApp();
    int nIndex = 0;
    for (int i = 0;i < m_arrBarInfo.GetSize(); i++)    {
        CControlBarInfo* pInfo = (CControlBarInfo*)m_arrBarInfo[i];
            if (pInfo->SaveState(lpszProfileName, nIndex))
                nIndex++;
    }
    TCHAR szSection[256];
    wsprintf(szSection, szSummarySection, lpszProfileName);
    pApp->WriteProfileInt(szSection, szBars, nIndex);
    CSize size = GetScreenSize();
    pApp->WriteProfileInt(szSection, szScreenCX, size.cx);
    pApp->WriteProfileInt(szSection, szScreenCY, size.cy);
}
```

SaveState()首先在 m_arrBarInfo 数组里循环，并为每个 CControlBarInfo 结构调用 CControlBarInfo::SaveState()。在保存了每个控制栏后，SaveState()通过调用 WriteProfileInt()将它自己的状态信息存储到 INI 文件里或者注册表里。

CControlBarInfo:: SaveState()所做的事情基本上与 CDockState::SaveState()一样。它通过调用 WriteProfileInt()将 CControlBarInfo 的数据成员写出去。

下面通过一个简单的例子来让我们能对所发生的事情有一个更真实的感觉。假设我们有一个普通的 AppWizard 应用程序，它有一个状态栏和一个工具栏。我们将一个 SaveBarState（"ControlBarState"）加到 CMainFrame::OnClose()里。这会使得 3 个框条（状态栏、工具栏以及工具栏的停放条）被保存到应用程序的 INI 文件里。

退出时，INI 文件如下面代码所示：

```
[ControlBarState-Summary]
Bars=3
ScreenCX=1280
ScreenCY=1024

[ControlBarState-Bar0]
BarID=59392
XPos=-2
YPos=-2
Docking=1
MRUDockID=0
MRUDockLeftPos=218
MRUDockTopPos=257
MRUDockRightPos=206
MRUDockBottomPos=251
MRUFloatStyle=8196
MRUFloatXPos=218
MRUFloatYPos=257

[ControlBarState-Bar1]
BarID=59393

[ControlBarState-Bar2]
BarID=59419
Bars=3
Bar#0=65536
Bar#1=59392
Bar#2=65536
```

CDockState::SaveState()会存储"概述"（Summary）条目，它包括显示器的大小以及控制栏的数目。紧跟在概述后面的是对应各个控制栏的 ID、位置等条目。

通过将它们转换成 16 进制并和 AFXRES.H 里的定义相对照，你可以推断出各个控制栏的类型。例如：

Bar0 – 59392 = E800 = AFX_IDW_TOOLBAR = 工具栏

Bar1 – 59393 = E801 = AFX_STATUS_BAR = 状态栏

Bar2 – 59419 = E81B = AFX_DOCKBAR_TOP = 工具栏的 CDockBar

注意，停放条将它包含的一串控制栏也存储了。65536 指明在该条上没有其他框条。

相当简洁，是吧？如果你希望使用注册表，可以在应用程序的 InitInstance()里调用 CWinApp::SetRegistryKey()，然后这些信息就会以类似的格式被存储到注册表中。

CControlBar 的状态保持功能是如何工作的：读取

当应用程序启动，并且调用 LoadBarState（"ControlBarState"）时，将会创建一个 CDockState 对象，这个对象会读取概述里的信息。然后使用框条的数目信息，并进行这么多次的循环，从而每个框条创建和装载 CControlBarInfo 结构。因为读取和写入 CControlBar 状态的过程很相似，所以就将分析 DOCKSTAT.CPP 的细节内容留给读者。

CControlBar 需要被分析的最后一部分是 MFC 是如何定位控制栏的。

CControlBar 的布局管理

到目前为止，我们已经学习了 CControlBar 是如何被创建、停放以及如何在程序执行过程中保留状态信息。剩下的问题是 MFC 如何对所有的控制栏进行定位。如果用户重新设置了窗口的大小，是哪个类负责重置工具栏、状态栏以及其他控制栏的大小呢？当创建一个控制栏时，又是哪个类负责给它分配空间呢？

如果要完全详细地说明 MFC 是如何工作的至少要两本这样的书，所以我们只向你介绍 MFC 工作流程的简略过程，而且会告诉你哪些部分比较复杂。

每当框架窗口的状态发生变化时（例如，一个新的控制栏被加进来了，或者窗口被重置了大小），MFC 都会调用 CFrameWnd::RecalcLayout()。程序清单 9-29 包含了该函数的伪代码。

程序清单 9-29 CFrameWnd::RecalcLayout()的伪代码（在文件 WINFRM.CPP 中）

```
void CFrameWnd::RecalcLayout(BOOL bNotify)
{
    if (GetStyle() & FWS_SNAPTOBARS) {
        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
        RepositionBars(0, 0xffff, AFX_IDW_PANE_FIRST, reposExtra,
&m_rectBorder);
}
```

RecalcLayout()基本上是调用 CWnd::RepositionBars()来设置 MFC 的逻辑布局的。RepositionBars()对窗口的子窗口进行循环处理，从而找到它们的当前大小以及它们期望的大小。然后 RepositionBars()将这些框条四处移动，并且记录下剩留的空间以及其他几何学管理

上的限制。

在 RepositionBars()计算完新子窗口的位置后，它会调用 DeferWindowPos()将所有的子窗口装在一个大组里一起移动，而不是分别地一个一个移动。RepositionBars()然后重新设置 MDI 客户窗口，使得它可以适应没有被控制栏占据的剩余空间。

当然，这个过程做了很多简化，但如果你希望自己能在这个方面进行独立钻研的话，它至少给你开了一个头。在学习 CMiniFrameWnd 之前，让我们来复习已经学到的关于 CControlBar 的知识。

CControlBar 摘要

作为摘要，我们列出关于 CControlBar 的以下重要部分：

- CControlBar 是由 Windows 的通用控件进行绘画的。MFC 在通用控件里加入了停放功能、状态保持功能以及逻辑布局管理。
- CDockBars 是可停放工具栏的“着陆条”。它们在停放的和浮动的控件条下正常工作。
- 浮动框条通过调用 SetParent()变成被停放框条。
- 被停放框条也是通过调用 SetParent()，从而将父窗口从 CFrameWnd 里的 CDockBar 变成 CMiniDockFrameWnd 里的 CDockBar 而变成浮动框条。
- CMiniDockFrameWnd 其实是定制的 CMiniFrameWnd 加上了一个内部的 CDockBar。它可以让 CMiniDockFrameWnd 记录所有的控制栏，并赋予它们浮动的能力——就像航空母舰一样。
- CControlBar 的拖动是由没有文档说明的 CDockContext 辅助类来实现的。
- CControlBar 状态信息的保存和恢复是由没有文档说明的 CDockState 和 CControlBarInfo 辅助类来实现的。CDockState 里记录了多个 CControlBarInfo 对象的信息。对于每个 CControlBar 来说都有一个 CControlBarInfo 对象。有两个类可以对 INI 文件和注册表进行存储和读取，从而使得 CControlBar 可以在程序执行过程中保存状态信息。
- MFC 控制栏由相当复杂的 MFC 窗口布局函数进行定位。CWnd::RepositionBars()是组织中的核心函数。

更多信息

如果你希望自己可以像我们一样研究 CControlBar，下面是取得更多信息的一些线索：

- 布局逻辑。分析下面的函数可以知道关于控制栏布局的更多信息：
  - CDockBar::CalcFixedLayout()。
  - CWnd::RepositionBars()。
  - CControlBar::OnSizeParent()。
- 如果你对 MFC 的 Macintosh 版本感兴趣，可以在 BAR*.CPP 文件里搜索#ifdef _MAC 来看看 MFC 代码是如何针对 Macintosh 进行调整的。
- 在 BARSTAT.CPP 里检查 CStatusBar 的实现：
  - AFX_STATUSPANE 是什么，以及它是在哪里被存储的（提示：在前面我们已经告诉过你了）？

- SBPF_UPDATE 是做什么用的？
- 状态栏窗格是如何被更新的？
- CStatusCmdUI 是什么？

- 分析 CDialogBar 的实现：
  - 文档模版是在哪里被存储的？
  - 子控件和 OLE 控件是如何被初始化的？
- 通读 DOCKCONT.CPP 可以对控制栏的拖动和跟踪有更多的认识。
- 通读 DOCKSTAT.CPP 可以对控制栏的状态保持有更多的认识。

在结束 MFC 增强型用户界面之前，我们还有最后的一个类要学习。它很有意思，所以请你继续读下去。

CMiniFrameWnd

在 CControlBar 一节里，我们已经对没有文档说明的 CMiniFrameWnd 有了一点初步的认识，但我们还不清楚 MFC 是如何在那么小的框架里创建一个窗口的。

图 9-8 显示了 CMiniFrameWnd 的一个例子。注意，它的标题条比一个普通的窗口要短得多。而且左边的菜单图标没有了，还有右边的一些按钮也没有了。

图 9-8 CMiniFrameWnd 窗口

下面让我们来看看 MFC 是如何实现这么“小”的界面的。从 AFXWIN.H 里 CMiniFrameWnd 的声明中，我们注意到 CMiniFrameWnd 处理了几个“不寻常”的消息：

```
afx_msg BOOL OnNcActivate(BOOL bActive);
afx_msg void OnNcCalcSize(BOOL bCalcValidRects,
    NCCALCSIZE_PARAMS* lpParams);
afx_msg UINT OnNcHitTest(CPoint point); afx_msg void OnNcPaint();
afx_msg void OnNcLButtonDown(UINT nHitTest, CPoint pt );
afx_msg BOOL OnNcCreate(LPCREATESTRUCT lpcs);
```

如果你不知道这些消息是做什么用的，不要着急。即使是一些经验丰富的 Windows 程序员也未必知道它们——也就是这一点使得它们更有趣了！

这些消息被称为“非用户”消息。窗口的客户区是在边界和标题条包围下的区域。每当有些事情发生在窗口的用户区之外时，Windows 便会发送这些非用户消息。例如，当 Windows 开始绘制你的窗口时，它会发送 WM_NCPAINT 消息来使得标题条和边界被绘制。通常，DefWindowProc()负责这条消息，并且为用户绘制框条。然而，Windows 非常灵活，它可以让用户获取这条消息并自己绘制标题（你以前也许曾疑惑微软是如何在 Windows 95 Office 里加进微软图标的，现在你应该知道了吧）。

这个技术也正是 CMiniFrameWnd 用来绘制小型化的窗口外表所采用的技术。当然，除了绘制小型化的标题条，CMiniFrameWnd 还必须处理相当多的所有其他非用户消息，例如激活（标题条应该随着它被激活还是没有被激活而不同）、标题条里的按钮点击以及其他。

除了非用户消息，实现的剩下部分是一大堆 GDI 的调用，它们绘制各种不同的窗口组件。

我们将继续分析它们的任务留给读者。你会发现在 WINMINI.CPP 里所有的代码都被整理得非常清晰。以 OnNCPaint()函数为起点，开始你的分析之旅吧。祝你旅途愉快！

在我们进入下一章之前，还想揭示最后一个 MFC 用户界面特性：最近使用的文件链表。

MFC 的 MRU 文件链表实现

MFC 的 MRU（most recently used，最近使用）文件链表使得用户最近打开的文件（或者文档）很容易被再次访问，用户只需点击菜单元素，而不用再次重复打开文件的整个过程。

在我们分析 MFC 是如何实现 MRU 文件之前，让我们来看看在 MFC 应用程序里，你会如何使用它们。AppWizard 自动为你产生实现 MRU 文件所需的代码，这通过在 CWinApp::InitInstance()函数里加进对 CWinApp::LoadStdProfileSettings()的调用。LoadStdProfileSettings()的参数指明了你所希望记录的最多文件个数。默认值是 4，并且最大值是 16。

下面，你所要做的事情就是告诉 MFC 你希望 MRU 文件链表放置在什么地方，方法是在 Visual C++的资源编辑器里往菜单里加进菜单 ID 为 ID_FILE_MRU_FILE1 的菜单元素。

如果你的文档类是从 CDocumnet 继承来的，那么它们会自动将文件名保存到 MRU 文件链表里。你可以通过调用公用函数 CDocument::SetPathName()和 CWinApp::AddToRecentFileList()来手动管理该链表。

MFC 是如何实现 MRU 文件链表的

MRU 文件链表中的文件名保存在应用程序的 INI 文件里，在“Recently File List”部分名里。

当你的应用程序调用 CWinApp::LoadStdProfileSettings()时，它会初始化一个 CRecentFileList 对象，并将它存储到名为 m_pRecentFileList 的 CWinApp 数据成员里。接着，LoadStdProfileSettings()会调用 CRecentFileList::ReadList()。

等等！这里又有一个没有文档说明的类！不管你怎么搜索，你也不会找到任何一点与该类有关的线索。让我们分析地深入一点，看看 CRecentFileList 做了些什么。

该类的声明在程序清单 9-30 里，它位于文件 AFXPRIV.H 中！

程序清单 9-30 CRecentFileList 的声明（在文件 AFXPRIV.H 中）

```
class CRecentFileList
{
// Constructors
public:
    CRecentFileList(UINT nStart, LPCTSTR lpszSection,
        LPCTSTR lpszEntryFormat, int nSize, int nMaxDispLen =
        AFX_ABBREV_FILENAME_LEN);
// Attributes
    int GetSize() const;
    CString& operator[](int nIndex);
// Operations
    virtual void Remove(int nIndex);
    virtual void Add(LPCTSTR lpszPathName);
    BOOL GetDisplayName(CString& strName, int nIndex, LPCTSTR lpszCurDir,
```

```
        int nCurDir, BOOL bAtLeastName = TRUE) const;
    virtual void UpdateMenu(CCmdUI* pCmdUI);
    virtual void ReadList();     // reads from registry or ini file
    virtual void WriteList();    // writes to registry or ini file
// Implementation
    virtual -CRecentFileList();
    int m_nSize;                 // contents of the MRU list
    CString* m_arrNames;
    CString m_strSectionName;    // for saving
    CString m_strEntryFormat;
    UINT m_nStart;               // for displaying
    int m_nMaxDisplayLength;
    CString m_strOriginal;       // original menu item contents
);
```

下面是该私有类里各个成员的概述。

数据成员

- m_nSize——MRU 文件链表的表项数。
- m_arrNames——CString 数组，记录文件名。
- m_strSectionName——INI 文件里记录 MRU 文件链表信息的部分名。默认值为“Recent File List”。
- m_strEntryFormat——该字符串记录 MRU 部分的关键字。默认值为“File%d”，这里的%d 是 MRU 菜单里文件名的位置。
- m_nStart——指定开始放置菜单元素的位置。默认值为零。
- m_nMaxDisplayLength——MRU 文件链表表项的最大个数，从 LoadStdProfile-Settings()里传递。
- m_strOriginal——该 CString 对象记录被某个 MRU 菜单元素替换掉的原始菜单字符串。

成员函数

- CRecentFileList()——初始化所有的数据成员。
- GetSize()——获取内部的 MRU 文件名数组的当前元素数目。
- operator[]——提供对内部 MRU 文件名数组的类似于数组的访问。
- Remove()——从内部 MRU 链表中删除一个文件名。
- Add()——向 MRU 链表中增加一个文件名，并检查是否有重复。
- GetDisplayName()——创建要显示的文件名，在必要的时候简化文件名。通过调用 MFC 函数 AbbreviateName()来实现简化。该函数在需要的时候会加进圆形括号。
- UpdateMenu()——该函数被调用时会向应用程序的菜单里加进一个 MRU 链表元素。
- ReadList()——从应用程序的 INI 文件或者注册表里读取 MRU 链表。
- WriteList()——向应用程序的 INI 文件或者注册表写入 MRU 链表。

在 CRecentFileList 得到菜单链表后，MFC 调用 UpdateMenu()将这些项加到菜单里。

当用户希望装载一个 MRU 文件时，他或者她选择该菜单，从而使得其 ID 在 ID_FILE_MRU_ FILE1 与 ID_FILE_MRU_FILE16 之间的一个命令消息被创建。CWinApp 在它的消息映射表里有如下的项来处理这些命令：

```
ON_COMMAND_EX_RANGE(ID_FILE_MRU_FILE1, ID_FILE_MRU_FILE16,
                    OnOpenRecentFile)
```

CWinApp::OnOpenRecentFile()使用该命令的 ID 在 CRecentFileList 里查找该文件名，然后调用 OpenDocumentFile()函数，参数为该文件名。

如果你正在使用文档/视图结构，你的文档会通过 CDocument::DoSave()自动被加到 MRU 菜单里，它会以文档名为参数调用 CWinApp::AddToRecentFileList()。如果没有使用文档/视图结构，则可以自己调用该函数。

最后，当应用程序退出以及 CWinApp::ExitInstance()被调用时，CWinApp 会调用 SaveStdProfileSettings()。它会调用 m_pRecentFileList->WriteList()来将 CRecentFileList 的内容写到 INI 文件或者注册表里。

结　语

现在，你应该已经对 MFC 如何实现它的许多 GUI 类（以及一些非 GUI 类）有了非常清晰的理解吧。而且，如果查看 AFXPRIV.H，你会发现里面的类你已经非常熟悉了。如果你能说出下面的类是干什么用的，那么你已经成为了一名 MFC 内核高手了。

- CSharedFile。
- CPreviewDC。
- CPreviewView。
- CMiniDockFrameWnd。
- CRecentFileList。
- CDockState。

在下一章里我们会分析 MFC 里的线程类，并且会学习 MFC 提供的不同的 DLL 选项。

在学习线程和 DLL 后，我们会开始学习 MFC 的 OLE 类。只要你理解了这些内容，你就会成为一名真正的 MFC 内核高手，所以请继续学习吧。

CHAPTER

10 MFC 的 DLL 与线程

在本章里我们将学习两个似乎没有什么联系的内容：MFC 的 DLL 与线程。我们会讲述对于 DLL MFC 是如何工作的，以及 MFC 是如何让用户可以方便地使用类库来实现自己的扩展 DLL。本章还会讲述 MFC 线程实现的许多方面，并且说明它是如何与 Win32 线程 API 联系到一起的。

在我们深入研究这些内容之前，让我们来看看 MFC 的 DLL 与线程实现之间相同的概念：状态。

理解状态

你也许还记得在第 2 章里，我们接触过 AFX_MODULE_STATE 结构。在本章里我们将在 DLL 和线程的环境里再一次学习该结构，并且向你介绍一些新的状态结构。

回到 Win16 编程的那些日子（在 Win32 多线程之前），修改程序的数据仅仅是 DLL 所要考虑的事情。在 Win16 里，如果两个应用程序改变了 DLL 的一个变量，它们都会看到这个变化。

在 Win32 多线程应用程序里，数据的访问变得更加复杂。Win32 提供了“进程局部”的数据，从而解决了 Win16 里 DLL 的问题。然而，线程引入了新的数据访问复杂性，而且，微软不得不在 Win32 上支持 MFC。它没有进程局部的数据，所以他们也不得不考虑 Win16 的 DLL 所要考虑的问题。

要解决这些问题，MFC 实现了独立的状态对象，它们包含了不同的 MFC 结构所需的状态信息。

MFC 状态解释

有 3 种类型的 MFC 状态信息：模块状态、进程状态以及线程状态。在 Win32 里，一个模块就是独立于应用程序其他部分而操作的可执行代码。例如 DLL 与 OLE 控件。当 MFC 被用作 DLL 时，它也是一个模块。进程与 Win32 进程的概念一致。

进程状态与线程状态非常类似：它们都有传统的全局数据，但它们必须与 Win32 或者多线程对应，从而具体到某个进程或者线程。

模块的状态比较特殊，因为它既可以包含真正的全局状态，也可以包含进程局部或者线程局部的状态，而且它可以被快速地切换。例如，如果 3 个程序都在使用 MFC 的

DLL，那么每个应用程序都必须包含自己的模块状态信息，为每个用户保存 MFC 的 DLL 状态。

可以把 MFC 状态理解成应用程序不同部分的局部数据。例如，进程状态包含局部于进程和某个模块的数据。模块的状态信息包含局部于该模块的数据，线程的状态信息包含局部于该线程的数据。

MFC 的状态是一个非常容易被混淆的概念，所以让我们停止分析抽象的概念吧，下面我们将读读具体的代码！下面的 3 个小节分别对 3 个重要的 MFC 状态进行了分析，从非常通用的（进程）直到非常具体的（线程）。

注意，当我们分析不同的状态时，会看到代码中有#ifdef AFXDLL 部分。当它是 DLL 时，AFXDLL 被打开，而当它被静态链接时，AFXDLL 被关闭。在不同的环境下，MFC 需要不同的信息，因为静态环境和共享环境都有不同的要求。当我们深入分析 DLL 时，我们会对它们作详细解释。现在，你也许希望能对它们的区别有个大致的了解。

MFC 进程状态

什么样的信息才能使得进程局部化呢？程序清单 10-1 显示了 AFX_MODULE_PROCESS_STATE 的定义，来自 AFXSTAT_.H。

程序清单 10-1 AFX_MODULE_PROCESS_STATE（在文件 AFXSTAT_.H 中）

```
class AFX_MODULE_PROCESS_STATE : public CNoTrackObject
{
public:
    AFX_MODULE_PROCESS_STATE();
    virtual ~AFX_MODULE_PROCESS_STATE();
    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
    // OLE control container manager
    COccManager* m_pOccManager;
    // locked OLE controls
    CTypedSimpleList<COleControlLock*> m_lockList;
};
```

除了构造函数和析构函数，AFX_MODULE_PROCESS_STATE 还包含下面的内容：

- m_pfnFilterToolTipMessage——一个函数指针，指向用于过滤工具提示消息的函数。
- m_libraryList——附加的 MFC 扩展 DLL 链表（在后面我们将详细解释）。
- m_appLangDLL——当前被局部化的资源的实例句柄。
- m_pOccManager——指向 OLE 控件管理对象的指针（在第 15 章里我们将详细解释）。
- m_lockList——被锁定的 OLE 控件链表（在第 15 章里我们将详细解释）。

所以，在进程状态这个层次上，实在是没有什么让 MFC 程序开发人员感到非常有趣的事情。但是知道它的存在还是有好处的，毕竟你会偶然遇到它。我们将在本章里重点讲述模块和线程状态，因为它们与我们每天的编程更有联系。在分析了各个状态的内容后，

我们将分析它们之间的联系。你可以做个有趣的试验，统一一下我们用到“状态”这个词的次数。

剖析 MFC 模块状态

什么样的信息才能使得一个模块局部化呢？每个 DLL 都有资源，所以在保存所有这些资源的方式上会有所限制。而且，当一个 DLL 被初始化时，它都会被传递给一个 hInstance。

程序清单 10-2 显示了 AFX_MODULE_STATE 的声明，这样你就可以直接了解 MFC 需要什么信息来使得一个模块局部化。

程序清单 10-2 AFX_MODULE__STATE 的定义（在文件 AFXSTAT_.H 中）

```
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
    CWinApp* m_pCurrentWinApp;
    HINSTANCE m_hCurrentInstanceHandle;
    HINSTANCE m_hCurrentResourceHandle;
    LPCTSTR m_lpszCurrentAppName;
    BYTE m_bDLL;
    BYTE m_bSystem;
    BYTE m_bReserved[2];
    short m_fRegisteredClasses;
#ifdef _AFXDLL
    CRuntimeClass* m_pClassInit;
#endif
    CTypedSimpleList<CRuntimeClass*> m_classList;
#ifdef _AFXDLL
    COleObjectFactory* m_pFactoryInit;
#endif
    CTypedSimpleList<COleObjectFactory*> m_factoryList;
    long m_nObjectCount;
    BOOL m_bUserCtrl;
    TCHAR m_szUnregisterList[4096];
#ifdef _AFXDLL
    WNDPROC m_pfnAfxWndProc;
    DWORD m_dwVersion;
#endif
    PROCESS_LOCAL(AFX_MODULE_PROCESS_STATE, m_process)
    THREAD_LOCAL(AFX_MODULE_THREAD_STATE, m_thread)
};
```

MFC 在 AFX_MODULE_STATE 里保存了下面的数据：

- m_pCurrentWinApp——指向该模块的 CWinApp 对象的指针。
- m_hCurrentInstanceHandle——模块的实例句柄。
- m_hCurrentResourceHandle——资源的实例句柄。如果你希望有一个局部于语言的资

源，它将有所变化。

- m_lpszCurrentAppName——当前应用程序的名字。
- m_bDLL——指明该模块是否是 DLL 的一个标志。
- m_bSystem——指明该模块是否是系统模块的一个标志。
- m_fRegisteredClasses——用于模块的延迟注册类的位标志。
- m_pClassInit——指向第一个类的 CRuntimeClass 信息的指针（通常是 m_classList 的头）。
- m_classList——模块里各个对象的 CRuntimeClass 信息链表。
- m_pFactoryInit——指向第一个 COleObjectFactory 对象的指针（通常是 m_factoryList 的头）。
- m_factoryList——模块里所有 COleObjectFactory 对象的链表。
- m_nObjectCount——被锁住的 OLE 对象的数目。
- m_bUserCtrl——如果用户有控制权，则设置为 TRUE，否则设置为 FALSE。
- m_szUnregisterList——未被注册的类链表。
- m_pfnAfxWndProc——指向模块所有的 AfxWndProc 的指针。
- m_dwVesion——模块链接所使用的 MFC 版本号。
- m_process——进程状态信息。
- m_thread——线程状态信息。

注意，m_process 与 m_thread 声明被 PROCESS_LOCAL 与 THREAD_LOCAL 宏所包装。这些宏把数据与将这些数据放到不同地方的类包装到一起。在 THREAD_LOCAL 里，数据由使用了 Win32 TLS（Thread Local Storage，线程局部存储）API 的类所包装，例如 TlsSetValue()、TlsAlloc()、TlsFree()以及 TlsGetValue()。MFC 在 CThreadSlotData 类里包含了 TLS 值的一个内部数组。我们把深入了解该类与分析使用了 TLS 的 MFC 实现的任务留给了读者，它们的定义在 AFXTLS_.H 文件里，实现在 AFXTLS.CPP 文件里。

PROCESS_LOCAL 使用了 TLS 来保护运行在 Win32 上的 DLL 里的数据，Win32 并不在操作系统级别上提供进程局部数据。

下面让我们来看看线程状态信息的内容，然后就开始将 MFC 的状态问题一起解决。

MFC 线程状态

什么样的信息才能使得一个线程局部化呢？记住，线程是执行线程（可以把线程想像一个成人的代码噬虫，在整个过程中，它不停得吃着代码）。在 MFC 的线程里，需要跟踪各种各样的信息。

记住，线程随时可以被打断，所以 MFC 不得不非常小心。它将所有的数据都放到线程状态里，这样被中断的 MFC 调用就可以被恢复了。

程序清单 10-3 包含了 MFC 的_AFX_THREAD_STATE 类的声明。

程序清单 10-3 _AFX_THREAD_STATE 的声明（在文件 AFXSTAT_.H 中）

```
class _AFX_THREAD_STATE : public CNoTrackObject
{
public:
    _AFX_THREAD_STATE();
```

```
    virtual ~_AFX_THREAD_STATE();
    AFX_MODULE_STATE* m_pModuleState;
    AFX_MODULE_STATE* m_pPrevModuleState;
    void* m_pSafetyPoolBuffer;    // current buffer
    AFX_EXCEPTION_CONTEXT m_exceptionContext;
    CWnd* m_pWndInit;
    CWnd* m_pAlternateWndInit;      // special case commdlg hooking
    DWORD m_dwPropStyle;
    DWORD m_dwPropExStyle;
    HWND m_hWndInit;
    BOOL m_bDlgCreate;
    HHOOK m_hHookOldSendMsg;
    HHOOK m_hHookOldCbtFilter;
    HHOOK m_hHookOldMsgFilter;
    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;
    CView* m_pRoutingView;          // see CCmdTarget::GetRoutingView
    CFrameWnd* m_pRoutingFrame;     // see CCmdTarget::GetRoutingFrame
    BOOL m_bWaitForDataSource;
    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
};
```

_AFX_THREAD_STATE 包含下面的成员：

- m_pModuleState——指向当前模块状态的指针。
- m_pPrevModuleState——指向前一模块状态的指针。
- m_pSafetyPoolBuffer——指向安全缓冲区的指针，它支持强壮的临时对象内存分配。
- m_exceptionContext——当前的异常环境。
- m_pWndInit——一个窗口指针，指向最近 hook 的窗口。
- m_pAlternateWndInit——指向最近被 hook 的公用对话框窗口的指针。
- m_dwPropStyle——属性页的风格。
- m_dwPropExStyle——属性页的扩展风格。
- m_hWndInit——m_pWndInit 的匹配句柄。
- m_bDlgCreate——表明某个对话框被创建了。MFC 为对话框绘制与普通窗口不同的背景颜色。
- m_hHookOldSendMsg——由::SetWindowsHookEx（WH_CALLWNDPROC）返回的前一句柄的句柄。
- m_hHookOldCbtFilter——由::SetWindowsHookEx（WH_CBT）返回的前一句柄的句柄。
- m_hHookOldMsgFilter——由::SetWindowsHookEx（WH_MSGFILTER）返回的前一

句柄的句柄。

- m_lastSentMsg——发送的最后一个消息。
- m_hTrackingWindow——当前跟踪窗口的句柄。
- m_hTrackingMenu——当前跟踪菜单的句柄。
- m_szTempClassName——在 AfxRegisterWndClass()中使用的缓冲区。
- m_hLockoutNotifyWindow——被锁定（没有 OLE 控件）的窗口句柄，如果存在一个的话。
- m_bInMsgFilter——表明该线程在一个消息过滤器中的标志。
- m_pRoutingView——在将消息发送给文档之前，视图首先将自己保存到该变量中。
- m_bWaitForDataSource——指明 ODBC 正在等待数据。
- m_pToolTip——指向当前 CToolTipCtrl 的指针。
- m_pLastHit——指向拥有工具提示控件的最后一个窗口的指针。
- m_nLastHit——最后的点击测试代码（用于工具提示点击测试）。
- m_lastInfo——最后的工具提示 TOOLINFO 结构。
- m_nLastStatus——最后的浮动状态代码。
- m_pLastStatus——指向最后的浮动状态控制条的指针。

MFC 状态之间的联系

现在，你已经对 MFC 状态结构里存储的信息类型有了一个初步的了解，下面让我们来看看它们之间是如何被联系起来的，以线程为起点直到进程。

当 MFC 需要到达当前的_AFX_THREAD_STATE 时，它会调用 AfxGetThreadState()，它的实现如下所示：

```
_AFX_THREAD_STATE* AFXAPI AfxGetThreadState()
{
    return _afxThreadState.GetData();
}
```

在 AFXSTATE.CPP 的前面是一个全局的声明：

```
THREAD_LOCAL(_AFX_THREAD_STATE, _afxThreadState)
```

这行代码会为每个线程的 TLS 创建一个名为_afxThreadState 的_AFX_THREAD_STATE 类，你可以通过调用 AfxGetThreadState()来对它进行访问操作。

在程序清单 10-3 里，_AFX_THREAD_STATE 记录了指向当前模块状态的指针，名为 m_pModuleState。在 MFC 里，AFX_MODULE_STATE 通过调用 AfxGetModuleState()函数来得到，它的实现如下所示：

```
AFX_MODULE_STATE* AFXAPI AfxGetModuleState()
{
    _AFX_THREAD_STATE* pState = _afxThreadState;
    AFX_MODULE_STATE* pResult;
    if (pState->m_pModuleState != NULL)
        pResult = pState->m_pModuleState;
    else
        pResult = AfxGetAppModuleState(); return pResult;
}
```

在绝大多数情况下，AfxGetModuleState()会得到_afxThreadState.m_pModuleState 的 AFX_MODULE_STATE。如果它为 NULL，那么将有一个全局的进程局部模块状态，定义如下所示：

```
PROCESS_LOCAL(_AFX_BASE_MODULE_STATE, _afxBaseModuleState)
```

AfxGetModuleState()的实现如下所示：

```
AFX_MODULE_STATE* AFXAPI AfxGetAppModuleState()
{
    return _afxBaseModuleState.GetData();
}
```

在内部的 MFC 链表里搜索到数据所对应的元素后，GetData()通过调用::TlsGetValue()来获得 TLS 的信息。图 10-1 显示了模块与线程是如何联系到一起的。

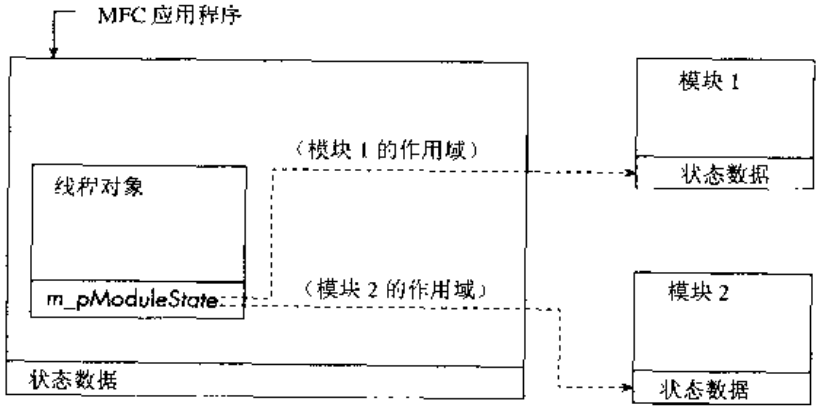


图 10-1 模块与线程的联系

当 MFC 是一个 DLL 时，它会通过调用 AfxSetModuleState()来改变模块的状态，它的实现如下所示：

```
AFX_MODULE_STATE* AFXAPI AfxSetModuleState(AFX_MODULE_STATE* pNewState)
{
    _AFX_THREAD_STATE* pState = _afxThreadState;
    AFX_MODULE_STATE* pPrevState = pState->m_pModuleState;

    pState->m_pModuleState = pNewState;
    return pPrevState;
}
```

现在，我们已经了解了 MFC 状态的概念，下面我们可以开始分析真正的 DLL 和线程了。我们首先将分析 DLL，然后再分析 MFC 线程。在分析过程中，你会了解到这些状态结构更多的信息，所以你一定要对正在学习的知识有很好的把握。

MFC 的 DLL

MFC 对 DLL 的支持从 MFC 2.0 开始一直在逐步地升级。在以前的日子里，有两种 DLL 方案：USRDLL 与 AFXDLL（以它们的用途来命名）。

USRDLL 能被静态地“粘贴”到 DLL 上。这意味着你可以在 DLL 里使用 MFC，而不必

要求使用该 DLL 的应用程序也是用 MFC 写的。USRDLL 的问题在于它们都非常大，因为你是在把 MFC 抓到你的 DLL 上。如果你有 3 个或者 4 个这样的 DLL，它会变得很庞大。而且 USRDLL 也丧失了 DLL 的主要功能：减少代码的拷贝。

AFXDLL 是使用了 MFC 的 DLL。MFC 在 DLL 里，或者被静态地附着在应用程序上。AFXDLL 只能用于 MFC 应用程序（这也正是 USRDLL 产生的原因）。

在今天的 MFC 4.x 里，USRDLL 已经逐渐被淘汰了，并且被叫做普通 DLL。也就是说，它们是有着一个输出链表的普通 Win32 DLL。它们可以被静态地链接，这也正是原来 USRDLL 的工作方式，或者它们也可以被动态链接、共享（这是 4.0 的一个新特性）。通常只有在下面的情况下该设置才会被用到：DLL 的编写者希望将 DLL 的函数输出到一个非 MFC 应用程序，但还是希望能在 DLL 里使用 MFC。AFXDLL 同样也有了一个新的、更友好的名字：扩展 DLL。

在本节里，我们将集中讨论 MFC 的扩展 DLL 的实现。我们会着重讲述 MFC 的 DLL 链接与非 DLL 链接之间的区别。首先，让我们来学习资源。

DLL 的资源问题

你也许已经注意到，在 MFC 的源代码里，每当某个资源被装载时，微软不会直接装载资源，而是通过调用 AfxFindResourceHandle()来实现。这个辅助函数会负责搜索一串已被装载进来的扩展 DLL 来寻找该资源。在我们看 AfxFindResourceHandle()的伪代码之前，让我们首先来看看 MFC 是如何保存一串扩展 DLL 的，以及它为什么要这样做。

实际上，在 DLL 里有 3 种类型的信息，它们必须串到一起，以使 MFC 在运行时能找到所需的信息：资源、静态的 CRuntimeClass（它是 CObject 的派生类，参考第 5 章）指针以及 OLE 对象厂（object factory）。

下面让我们来看看 MFC 是如何对这些信息进行存储、查找以及恢复操作的。我们将着重讲述资源的问题，CRuntimeClass 信息与 OLE 对象厂的信息的存储和链接是以通用的方式进行的。

MFC 的 DLL 版本程序的起点在 DllMain()（当 DLL 被装载时该函数被调用）里，在这里我们将看到程序清单 10-4 所示的神秘的代码行。

程序清单 10-4　DllMain()的代码片断

```
AfxInitExtensionModule(coreDLL,hInstance)
CDynLinkLibrary* pDLL = new CDynLinkLibrary(coreDLL, TRUE);
```

实际上，如果你曾经写过 MFC 的扩展 DLL，你将不得不将这些代码加到你的 DllMain()里。AfxInitExtensionModule()与 CDynLinkLibrary 结构函数的参数都是 AFX_EXTENSION_MODULE。它是在 AFXDLL_.H 里定义的一个小结构，包含每个 MFC 扩展 DLL 所需的信息。AFX_EXTENSION_MODULE 的声明如下所示：

```
struct AFX_EXTENSION_MODULE
{
    BOOL bInitialized;
    HMODULE hModule;
    HMODULE hResource;
    CRuntimeClass* pFirstSharedClass;
```

```
    COleObjectFactory* pFirstSharedFactory;
};
```

下面让我们来看看 AfxInitExtensionModule()与 CDynLinkLibrary 对 AFX_EXTENSION_MODULE 做了哪些操作，以及它们是如何与资源关联到一起的。

剖析 AfxInitExtensionModule()

程序清单 10-5 显示了 AfxInitExtensionModule()的伪代码。

程序清单 10-5 AfxInitExtensionModule()的伪代码（在文件 DLLINIT.CPP 中）

```
BOOL AFXAPI AfxInitExtensionModule(AFX_EXTENSION_MODULE& state, HMODULE
hModule)
{
// only initialize once
    if (state.bInitialized){
        AfxInitLocalData(hModule);
        return TRUE;
    }
    state.bInitialized = TRUE;
    // save the current HMODULE information for resource loading
    state.hModule = hModule;
    state.hResource = hModule;
    // save the start of the runtime class list
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
    state.pFirstSharedClass = pModuleState->m_classList.GetHead();
    pModuleState->m_classList.m_pHead = pModuleState->m_pClassInit;
    // save the start of the class factory list
    state.pFirstSharedFactory = pModuleState->m_factoryList.GetHead();
    pModuleState->m_factoryList.m_pHead = pModuleState->m_pFactoryInit;
    return TRUE;
}
```

首先，AfxInitExtensionModule()通过检查 AFX_EXTENSION_MODULE 状态参数的 bInitialized 域来检查该扩展 DLL 是否已经被初始化了。如果该模块以前就已经被初始化了，AfxInitLocalData()将被调用来更新 TLS 使用的模块句柄。如果 DLL 以前没有被初始化，AfxInitExtensionModule()继续执行，并且将状态的 bInitialized 域设置为 TRUE，接着开始初始化过程。

接着，AfxInitExtensionModule()将状态参数的 hModule 和 hResource 设置成 hModule 参数。然后，AfxInitExtensionModule()将当前的模块状态信息提取到 pModuleState 里，并且得到模块状态 CRuntimeClass 信息链表的头。结果被放置到状态 pFirstSharedClass 里。接着，模块状态类链表的头被设置成模块状态类的初始化信息。

接着，AfxInitExtensionModule()对模块的对象厂链表做了类似的操作。对象厂链表的头被存储到模块状态里，然后，在模块状态里被赋值给 m_pFactoryInit 指针。

MFC 在这里都做了些什么呢？当_AFX_CLASSINIT 结构被初始化时（有辅助类将 CRuntimeClass 对象放到一个链表中），它们被放置到模块状态的全局链表中。当有扩展 DLL 时，微软希望将这个全局链表转换出来，并且将它转换到 CDynLinkLibrary 对象里。如果没有对原来的链表进行跟踪，微不能执行这个转换操作。跟踪链表头的任务是由

AFX_EXTENSION_MODULE 成员来完成的。它们需要在另一个进程运行时跟踪链表的头，并且需要在它的链表里记录一个类似的 CDynLinkLibrary，因为 CDynLinkLibrary 对象全部都是进程局部的。

无可否认，在这里发生了什么事情并不是很明显。我们再多忍受几节后就会明白它们是如何被组合在一起的。下一个要分析的问题是 CDynLinkLibrary 辅助类。

剖析 CDynLinkLibrary

CDynLinkLibrary 是一个令人讨厌的没有文档说明的类（如果你曾经写过 MFC 的扩展 DLL，你会明白我的意思）。每当 MFC 文档告诉你去创建一个实例时，微软并没有告诉你这个类做了些什么，以及为什么要这样做。

程序清单 10-6 包含了 CDynLinkLibrary 的声明。注意，CDynLinkLibrary 是 CCmdTarget 的派生类。这有一点奇怪，因为 CDynLinkLibrary 好像从来就没有利用过它的父类的特性。也就是说，它好像并没有处理任何消息。

也许 MFC 的组员觉得这样的一种情况是可能的：某个非窗口/文档相关的消息可能需要被发送给某个扩展 DLL，例如一个命令消息。从理论上讲，你有了可动态装载的命令处理函数。这个消息将留给用户来作为练习，希望能将 MFC 的这个小的、非常好的技巧运用到真实的问题上去。

程序清单 10-6 CDynLinkLibrary 的声明（在文件 AFXDLL_.H 中）

```
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;
    CTypedSimpleList<COleObjectFactory*> m_factoryList;
    BOOL m_bSystem;                         // TRUE only for MFC DLLs
// Implementation
public:
    CDynLinkLibrary* m_pNextDLL;            // simple singly linked list
    virtual ~CDynLinkLibrary();
};
```

现在我们能看到 CDynLinkLibrary 的第二个参数指明了该扩展 DLL 是否是一个系统 DLL。回头看看程序清单 10-4 里 DllMain()的两行代码，MFC 将这个参数设置成 TRUE。在 MFC 的文档里，他们告诉你这个参数被设置成 FALSE。最终的结论是当 MFC 是扩展 DLL 时，它被当作了“系统”DLL，并且当有任何其他扩展 DLL 被初始化时，它又被当作了非系统 DLL。

在我们分析系统 DLL 与非系统 DLL 的实现之前，注意在 CDynLinkLibrary 里有许多在 AFX_EXTENSION_MODULE 里能找到的信息。惟一的比较大的增添是名为 m_pNextDLL 的 CDynLinkLibrary 指针。

下面让我们来看看 CDynLinkLibrary 的构造函数，以得到更多的答案。

CDynLinkLibrary:: CDynLinkLibrary()

程序清单 10-7 包含了 CDynLinkLibrary 构造函数的伪代码。

程序清单 10-7 CDynLinkLibrary 构造函数的伪代码（在文件 DLLINIT.CPP 中）

```
CDynLinkLibrary::CDynLinkLibrary(AFX_EXTENSION_MODULE& state, BOOL
bSystem)
{
    m_factoryList.Construct(offsetof(COleObjectFactory, m_pNextFactory));
    m_classList.Construct(offsetof(CRuntimeClass, m_pNextClass));
    // copy info from AFX_EXTENSION_MODULE
    struct m_hModule = state.hModule;
    m_hResource = state.hResource;
    m_classList.m_pHead = state.pFirstSharedClass;
    m_factoryList.m_pHead = state.pFirstSharedFactory;
    m_bSystem = bSystem;
    // insert at the head of the list (extensions will go in front of
    // core DLL)
    AFX_MODULE_PROCESS_STATE* pState = AfxGetModuleProcessState();
    AfxLockGlobals(CRIT_DYNLINKLIST); pState->m_libraryList.AddHead(this);
    AfxUnlockGlobals(CRIT_DYNLINKLIST);
}
```

首先，构造函数构造 m_factoryList 和 m_classList。然后构造函数将这个信息从描述当前 MFC 扩展 DLL 的 AFX_EXTENSION_MODULE 状态参数里拷贝到对应的 CDynLinkLibrary 数据成员里。

最后，CDynLinkLibrary 把它自己加到当前的进程状态里。向进程状态中添加内容这一动作是被锁定的，所以能够保持对 DLL 的正确的添加顺序。下面我们将看看状态链表是如何被搜索的。

剖析 AfxFindResourceHandle()

所有这些辅助类的最终结果是一串 CDynLinkLibrary 对象（每个扩展 DLL 对应一个对象）被保存到了模块状态里。扩展 DLL 将它自己的信息传递给 CDynLinkLibrary 的构造函数，它会把自己加到模块链表上去。现在你已经掌握了所有的背景知识，现在可以看看 AfxFindResourceHandle()函数是如何搜索资源了。程序清单 10-8 里包含了 AfxFind-ResourceHandle()的伪代码。

程序清单 10-8 AfxFindResourceHandle()的伪代码（在文件 DLLINIT.CPP 中）

```
HINSTANCE AFXAPI AfxFindResourceHandle(LPCTSTR lpszName, LPCTSTR lpszType)
{
    HINSTANCE hInst;
    // first check the main module state
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
    if (!pModuleState->m_bSystem){
        hInst = AfxGetResourceHandle();
        if (::FindResource(hInst, lpszName, lpszType) != NULL)
        return hInst;
```

```
    }
    // check for non-system DLLs in proper order
    AFX_MODULE_PROCESS_STATE* pState = AfxGetModuleProcessState();
    AfxLockGlobals(CRIT_DYNLINKLIST);
    for (CDynLinkLibrary* pDLL = pState->m_libraryList; pDLL != NULL;
        pDLL = pDLL->m_pNextDLL){
        if (!pDLL->m_bSystem && pDLL->m_hResource != NULL &&
            ::FindResource(pDLL->m_hResource, lpszName, lpszType) != NULL){
            AfxUnlockGlobals(CRIT_DYNLINKLIST);
            return pDLL->m_hResource;
        }
    }
    AfxUnlockGlobals(CRIT_DYNLINKLIST);
    // check language specific resource next
    hInst = pState->m_appLangDLL;
    if (hInst != NULL && ::FindResource(hInst, lpszName, lpszType) != NULL)
        return hInst;
    // check the main system module state
    pModuleState = AfxGetModuleState();
    if (pModuleState->m_bSystem){
        hInst = AfxGetResourceHandle();
        if (::FindResource(hInst, lpszName, lpszType) != NULL)
        return hInst;
    }
    // check for system DLLs in proper order
    AfxLockGlobals(CRIT_DYNLINKLIST);
    for (pDLL = pState->m_libraryList; pDLL != NULL;
            pDLL = pDLL->m_pNextDLL){
        if (pDLL->m_bSystem && pDLL->m_hResource != NULL &&
           ::FindResource(pDLL->m_hResource, lpszName, lpszType) != NULL)
              AfxUnlockGlobals(CRIT_DYNLINKLIST); return pDLL->m_hResource;
    }
    AfxUnlockGlobals(CRIT_DYNLINKLIST);
    // if failed to find resource, return application resource
    return AfxGetResourceHandle();
}
```

正如你在程序清单 10-8 的代码里可以看到的，AfxFindResourceHandle()在寻找资源时采用了传统的、好的反复试验的方法。它所选择的路径对于定义 MFC 扩展 DLL 的行为来说非常重要。

首先，如果当前模块不是系统模块，AfxFindResourceHandle()在当前模块的资源句柄（由 AfxGetResourceHandle()函数返回）里查找资源。

接着，AfxFindResourceHandle()会在进程状态 m_libraryList 里遍历 CDynLinkLibrary 链表。如果扩展 DLL 不是系统 DLL，FindResource()被调用，它会尽力寻找资源。AfxFindResourceHandle()然后在被进程状态指向的与特定语言相关的 DLL 里寻找该信息。

多余的代码

你能在程序清单 10-8 里发现多余的代码吗？

嗯，你猜对了。AfxGetModuleState()被调用了两次，但实际上它只需要被调用一次。微软已经发现这个问题了，所以在未来的新的 MFC 版本里你应该会看到多余的一次已经被删除了。

接着，AfxFindResourceHandle()检查当前的模块是否是系统模块。如果是，AfxFindResourceHandle()会调用 FindResource()来搜索资源。最后，AfxFindResourceHandle()再次遍历 CDynLinkLibrary 链表，并在系统扩展 DLL 里查找该资源。

如果这些查找都没有发现任何东西，那么 AfxFindResourceHandle()会返回 AfxGetResourceHandle()，它仅仅是返回了进程的当前资源句柄。AfxGetResourceHandle()是 MFC 如何访问各种状态结构的一个很好的例子。它也显示了系统 DLL 与非系统 DLL 之间的差别。

现在我们已经学习了 MFC 的 DLL 是如何处理资源的，下面让我们来学习当 MFC 作为扩展 DLL 运行时，它会如何初始化自身。

扩展 DLL 初始化与清除

在原来的 Win16 的日子里，DLL 必须有 LibMain()和 WEP()（一个 Windows 退出函数）。LibMain()负责 DLL 的初始化，WEP()负责清除工作。在 Win32 里，DllMain()函数对这两个任务都要负责。Win32 调用 DllMain()时，用不同的 DWORD 类型的参数 dwReason 来指定到底要执行哪种操作。两种通常被处理的行为是 DLL_PROCESS_ATTACH 与 DLL_PROCESS_DETACH。尽管阅读 MFC 的 DllMain()确实还比较有趣（在 DLLINIT.CPP 文件里），但我们已经覆盖了与 DLL 初始化相关的许多内容。剩余的代码就是要处理各种错误检查以及各种预防措施。

扩展 DLL 的另一个有趣的方面是一些小的技巧，在下面的 MFC 消息映射里有所体现。

AFXDLL 与宏

到了要完全明白的时候了。以前，当我们讲述消息映射与 CObject 宏时，我们向你隐藏了宏的声明里的一些细节（我们实在是不愿意把读者弄得精力太分散）。

MFC 里的许多宏都有两个版本。我们将以 DECLARE_DYNAMIC()为例来说明。程序清单 10-9 包含了 DECLARE_DYNAMIC 的两个版本。

程序清单 10-9 DECLARE_DYNAMIC 的 DLL 与非 DLL 版本（在文件 AFX.H 中）

```
#ifdef _AFXDLL
#define DECLARE_DYNAMIC(class_name) \
    protected: \
    static CRuntimeClass* _GetBaseClass(); \
    public: \
    static AFX_DATA CRuntimeClass class##class_name; \
    virtual CRuntimeClass* GetRuntimeClass() const; \
#else //!_AFXDLL
#define DECLARE_DYNAMIC(class_name) \
    public: \
    static AFX_DATA CRuntimeClass class##class_name; \
    virtual CRuntimeClass* GetRuntimeClass() const; \
#endif //end !_AFXDLL
```

在程序清单 10-9 里，前面的 DECLARE_DYNAMIC 在 DLL 中使用，而下面的版本在非 DLL 的版本中使用。惟一的区别在于下面的两行代码：

```
protected:
static CRuntimeClass * _GetBaseClass();
```

让我们来看看 IMPLEMENT_DYNAMIC 是否也是不同的。程序清单 10-10 包含了它的伪代码（注意，IMPLEMENT_DYNAMIC 直接调用_IMPLEMENT_RUNTIMECLASS。为了读者查阅的方便，我们也列出了它的代码）。

程序清单 10-10 IMPLEMENT_DYNAMIC 声明的 DLL 与非 DLL 版本，（在文件 AFX.H 中）

```
#define IMPLEMENT_DYNAMIC(class_name, base_class_name) \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, NULL)
#ifdef _AFXDLL
#define _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema,
pfnNew) \
    CRuntimeClass* class_name::_GetBaseClass() \
    { return RUNTIME_CLASS(base_class_name); } \
    AFX_DATADEF CRuntimeClass class_name::class##class_name = { \
    #class_name, sizeof(class class_name), wSchema, pfnNew, \
    &class_name::_GetBaseClass, NULL }; \
    static const AFX_CLASSINIT \
    _init_##class_name(&class_name::class##class_name); \
    CRuntimeClass* class_name::GetRuntimeClass() const \
    { return &class_name::class##class_name; } \
#else
#define _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema,
pfnNew) \
    AFX_DATADEF CRuntimeClass class_name::class##class_name = { \
    #class_name, sizeof(class class_name), wSchema, pfnNew, \
    RUNTIME_CLASS(base_class_name), NULL }; \
    static const AFX_CLASSINIT \
    _init_##class_name(&class_name::class##class_name); \
    CRuntimeClass* class_name::GetRuntimeClass() const \
    { return &class_name::class##class_name; } \
#endif
```

在程序清单 10-10 里，代码变得更复杂了。DLL 版本与非 DLL 版本的区别非常大。首先，DLL 版本生产下面的成员函数：

```
CRuntimeClass * CMyClass::_GetBaseClass()
{
    return RUNTIME_CLASS(CMyBaseClass);
}
```

而且要注意，当 IMPLEMENT DYNAMIC 生成初始化代码时（这段代码结束标志着静态 CRuntimeClass 对象的初始化结束了）（如果要了解这些宏的细节，请参考第 5 章。这里，我们将着重考虑与 DLL 相关的问题），它将 GetBaseClass 函数的地址传递过去，而不仅仅是 RUNTIME_CLASS（CMyBaseClass），如同非 DLL 版本一样。

你可能已经猜到了，CRuntimeClass 结构允许使用 ifdef 而带来的差异：

```
#ifdef _AFXDLL
    CRuntimeClass* (PASCAL* m_pfnGetBaseClass)();
#else
    CRuntimeClass* m_pBaseClass;
#endif
```

记住，RUNTIME_CLASS 返回静态 CRuntimeClass 的地址，由 DECLARE_DYNAMIC 宏将它嵌入到你的类里。

作个总结，DLL 与非 DLL 之间的差别在于 MFC 必须调用能返回静态 CRuntimeClass 成员地址的函数，而不是直接存储、访问静态 CRuntimeClass 数据成员的地址。

为什么在 DLL 版本里 MFC 必须走这个弯路呢？我们询问了微软 MFC 内幕组，McCrory 院士作了如下的回答：

> 在 Win32 里导出/导入（export/import）的数据在使用前必须由 C 运行时初始化。这就不得不依赖 Win32 中导出数据的实现。这样就有了额外的一层间接关系，而且基本上指针必须在启动时刻被初始化。指针在对象的构造期间被 C 运行时初始化。实际上，它认为类似 CRuntimeClass 的类有一个构造函数，即使实际上它没有，而且是在编译时刻被初始化的。因为这种特性是用普通的静态对象构造方法构造对象来实现的，所以 CRuntimeClass 对象被构造的顺序也就不可预知了，因为还有其他的静态对象。这样，我们发现，如果我们仅仅将数据直接引入，用户会写出直接访问未必初始化的 CRuntimeClass 结构的代码（例如，某个对象的构造函数是在全局范围里进行初始化的）。要解决这个问题，我们只有通过函数来得到导出数据，它不需要任何特殊的初始化工作。这使得代码更短小了（不用对 CRuntimeClass 对象进行特殊的构造了），而且也解决了在静态对象初始化过程中过早地对 CRuntimeClass 对象进行访问的问题。

你会注意到，几乎所有的 MFC 宏都有 AFXDLL 和非 AFXDLL 版本，包括消息映射宏。

现在，相对于大多数的使用 MFC 扩展 DLL 的编程人员来说，你已经知道了更多的 MFC 扩展 DLL 的知识。现在我们可以继续前进了，下面就学习 MFC 线程。

MFC 线程

在本节里，我们将看到 MFC 是如何实现它的两个封装线程：辅助线程（worker thread）与 UI 线程（user-interface thread）。通常，我们都会指出一些运行中的、有趣的 Win32 API。可现在，我们认为你已经了解线程了，所以你也许需要查阅你周围的高级 Win32 编程书籍来进行复习，或者花一些时间来学习 Visual C++的在线书籍。

而且，我们会看到 MFC 的状态类会遍布在我们查看的线程代码的四周。如果你直接从索引或者目录直接跳到了这里，也许在继续学习之前，你需要回到本章的第一节，它对 MFC 的状态做了初步的讲解。

正如我们已经提到的，MFC 支持两种不同类型的线程：辅助线程与 UI 线程。因为线程已经是一个相当复杂的问题了，所以我们将从这两者之间较容易的一个出发，辅助线程，然后我们会继续分析 UI 线程。

MFC 用户通过调用 AfxBeginThread()来启动两种类型的线程。实际上对 AfxBeginThread()有两个重载。对于辅助线程来说，你将一个 CWinThread 对象和一个指针传递给控制函数（控制函数负责工作的完成）。对于 UI 线程，你需要创建一个 CWinThread，并且将它的 CRuntimeClass 信息传递给 AfxBeginThread()。

AfxBeginThread()的两个版本会接收你能想得到的创建线程所需的公共参数，例如线程的优先级以及安全方面的特性。

现在让我们来看看 MFC 是如何实现辅助线程的。

MFC 的辅助线程

让我们首先来看看 AfxBeginThread()的辅助线程版本是如何实现的。程序清单 10-11 包含了 AfxBeginThread()的伪代码。

程序清单 10-11 AfxBeginThread()的辅助线程版本（在文件 THRDCORE.CPP 中）

```
CWinThread* AFXAPI AfxBeginThread(AFX_THREADPROC pfnThreadProc,
    LPVOID pParam, int nPriority, UINT nStackSize, DWORD dwCreateFlags,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs)
{
    CWinThread* pThread = new CWinThread(pfnThreadProc, pParam);
    if (!pThread->CreateThread(dwCreateFlags|CREATE_SUSPENDED,
       nStackSize, lpSecurityAttrs)) {
        pThread->Delete();
        return NULL;
    }
    VERIFY(pThread->SetThreadPriority(nPriority));
    if (!(dwCreateFlags & CREATE_SUSPENDED))
        VERIFY(pThread->ResumeThread() != (DWORD)-1);
    return pThread;
}
```

首先，AfxBeginThread()在堆里初始化一个 CWinThread 对象，并将辅助函数指针作为参数传递进去，另外还有 pParam（传递给辅助函数的参数）。

接着，AfxBeginThread()用 CWinThread 对象来激活某个原来被暂停的线程（CREATE_SUSPENDED），并且将线程的特性参数传递给 CWinThread::CreateThread()。如果 CWinThread::CreateThread()的调用失败了，CWinThread 对象会被删除，并且 NULL 将被返回。

在调用 CWinThread::CreateThread()之后，AfxBeginThread()通过调用 CWinThread::SetThreadPriority()来设置优先级。最后，如果调用者没有设置 CREATE_SUSPENDED 标志，AfxBeginThread()会通过调用 CWinThread::ResumeThread()来使得线程处于运行状态。AfxBeginThread()会返回指向新 CWinThread 对象的指针。

CWinThread

从 AfxBeginThread()的实现可以看出，有一件事情相当明显：理解 MFC 的 CWinThread 类是理解 MFC 的线程实现的关键。所以首先让我们打开这个迷吧！程序清单 10-12 包含了 CWinThread 声明中更加有趣的部分。我们已经删除了许多有文档说明的成员函数，所以如果你对它们还不太熟悉，可以参考在线文档。

程序清单 10-12 CWinThread 的缩减后的声明（在文件 AFXWIN.H 中）

```
class CWinThread : public CCmdTarget
{
DECLARE_DYNAMIC(CWinThread)
public:
// Constructors *omitted
// Attributes
    CWnd* m_pMainWnd; // main window (usually same AfxGetApp()->m_pMainWnd)
```

```
    CWnd* m_pActiveWnd;  // active main window (may not be m_pMainWnd)
    BOOL m_bAutoDelete;  // enables 'delete this' after thread termination
    HANDLE m_hThread;        // this thread's HANDLE
    DWORD m_nThreadID;       // this thread's ID

// Operations *omitted
// Overridables *omitted
// Implementation

public:
    void CommonConstruct();
    virtual void Delete();
    MSG m_msgCur;                    // current message
public:
    // constructor used by implementation of
    AfxBeginThread CWinThread(AFX_THREADPROC pfnThreadProc, LPVOID pParam);
    // valid after construction
    LPVOID m_pThreadParams; // generic parameters passed to starting
                            // function
    AFX_THREADPROC m_pfnThreadProc;
protected:
    CPoint m_ptCursorLast;       // last mouse position
    UINT m_nMsgLast;             // last mouse message
};
```

在对各个成员的实现作了概述后，我们将分析一组重要的 CWinThread 函数的实现，来看看在强大的 MFC 类里到底发生了什么事情。首先，要注意到，CWinThread 是一个 CCmdTarget，这表明消息函数将会牵扯到它（在本章的后面将详细讲述）。

CWinThread 数据成员

- m_pMainWnd——指向应用程序主窗口的指针，CWinThread 需要用该指针来正确地对消息进行处理。
- m_pActiveWnd——当 OLE 服务器在某个地方处于活动状态时，该指针指向包含应用程序的主窗口。
- m_bAutoDelete——当该变量为 TRUE 时，CWinThread 会在线程结束时杀掉自己。
- m_hThread——被 CWinThread 封装的 Win32 线程的句柄。在证明它处于活动状态后，如果你愿意的话，可以在 Win32 的线程 API 里使用它。在后面我们会看看它是如何被创建以及如何被 CWinThread 的内部使用的。
- m_hThreadID——被 CWinThread 封装的线程的 Win32 线程 ID。
- m_msgCur——缓冲当前正被 CWinThread 消息发送器所处理的消息（在后面将详细讲述）。
- m_pThreadParams——保存 pParam 参数，它会继续传递给辅助函数（worker function）。
- m_pfnThreadProc——指向辅助函数的指针。
- m_ptCursorLast——最后的鼠标移动消息中的 CPoint。它被用来在 CWinThread 的空闲时刻过滤掉多余的鼠标移动消息。
- m_nMsgLast——与 m_ptCursorLast 类似，用来探测两个连续的消息（例如，当前的和最近的）。

CWinThread 成员函数的实现

- CommonConstruct()——两个构造函数（辅助线程和 UI 线程）都会最终调用该函数。我们不会查看它的伪代码，因为它仅仅是将数据成员设置成正确的默认值。
- Delete()——如果 m_bAutoDelete 值为 TRUE，Delete()会调用“delete this”，从而使得该线程将自己杀死（自杀！）。

现在让我们通过 CWinThread::CreateThread()来看看在 MFC 内部，线程到底是如何被创建的。

线程的创建——剖析 CWinThread::CreateThread()

CreateThread()相当复杂，所以我们将它分成两个部分。首先，通过使用一些内部的 MFC 功能结构和功能函数来创建线程。接着，MFC 对线程和 CWinThread 做一些有趣的初始化工作。

当 MFC 创建一个 Win32 线程时，它会调用_beginthreadex()运行时库函数。该函数带有好几个参数。我们要讨论的比较重要的参数如下所示：

- 第三个参数——指向开始执行新线程的函数地址（开始地址）的指针。对于 CWinApp::CreateThread()，该值为_AfxThreadEntry。
- 第四个参数——指向传递给线程函数的参数的指针（看第三个参数）。MFC 定义了一个内部结构，_AFX_THREAD_STARTUP，我们马上就会看到它被使用。
- 第六个参数——新线程的地址。CWinThread::CreateThread()将 m_nThreadID 的地址作为参数传递进去，它将被设置成线程的地址（ID）。

让我们带着预先准备的信息来看看 CWinThread::CreateThread()的第一部分。程序清单 10-13 包含了 CWinThread::CreateThread()的伪代码。

程序清单 10-13 CWinThread::CreateThread()缩减后代码的第一部分(在文件 THRDCORE.CPP 中)

```
BOOL CWinThread::CreateThread(DWORD dwCreateFlags, UINT nStackSize,
     LPSECURITY_ATTRIBUTES lpSecurityAttrs)
{
    // setup startup structure for thread initialization
    _AFX_THREAD_STARTUP startup;
    memset(&startup, 0, sizeof(startup));
    startup.pThreadState = AfxGetThreadState();
    startup.pThread = this;
    startup.hEvent = ::CreateEvent(NULL, TRUE, FALSE, NULL);
    startup.hEvent2 = ::CreateEvent(NULL, TRUE, FALSE, NULL);
    startup.dwCreateFlags = dwCreateFlags;
    // **some event checking and cleanup omitted for brevity.
    // create the thread (it may or may not start to run)
    m_hThread = (HANDLE)_beginthreadex(lpSecurityAttrs, nStackSize,
        &_AfxThreadEntry, &startup, dwCreateFlags | CREATE_SUSPENDED,
        (UINT*)&m_nThreadID);
    if (m_hThread == NULL)
    return FALSE;
```

CreateThread()首先创建一个局部的_AFX_THREAD_STARTUP，并且将与线程有关的有价值的信息存放到该变量里，该变量随后在线程的执行过程中被传递给线程函数。这些信息

包含以下内容：

- pThreadState——在程序清单 10-3 里提到的_AFX_THREAD_STATE。
- pThread——指向 CWinThread 的后向指针（back pointer）。
- hEvent/hEvent2——内部 CWinThread 引发的两个事件句柄。在线程被成功创建后，hEvent 被触发。在线程被重新启动后，hEvent2 被触发。
- dwCreateFlags——指定线程是否应该被挂起以及其他信息的创建标志。线程也许在后面的执行中需要检查这些，所以它们要被保存下来。

初始化信息（例如安全特性）没有被保存在_AFX_THREAD_STARTUP 中，因为或者线程在运行中并不需要这样的信息，或者可以通过 pThread 指针得到这些信息。

在用这些信息初始化局部的_AFX_THREAD_STARTUP 结构后，CreateThread()通过调用_beginthreadex()来创建一个线程——正如我们所预料到的！CreateThread()传递的参数有安全特性、栈的大小、_AfxThreadEntry、&startup、创建标志以及线程 ID 的地址。

线程会被成功创建，除非这些参数中有某一个不合要求，或者系统资源问题（例如没有内存了）不能创建新的线程。

记住，CreateThread()调用_beginthreadex()时的参数为 CREATE_SUSPENDED，所以尽管新的线程被创建了，它仍然处于睡眠状态，直到 MFC 决定将它唤醒。

CreateThread()的其他伪代码在程序清单 10-14 里。

程序清单 10-14 CWinThread::CreateThread()的简略代码（在线程创建之后的部分），来自文件 THRDCORE.CPP

```
    //CWinThread::CreateThread() continued...
    ResumeThread();
    ::WaitForSingleObject(startup.hEvent, INFINITE);
    ::CloseHandle(startup.hEvent);
    // if created suspended, suspend it until resume thread wakes it up
    if (dwCreateFlags & CREATE_SUSPENDED)
        ::SuspendThread(m_hThread);
    if (startup.bError)
    //**Error cleanup omitted - lots of frees and closes <g>
    // allow thread to continue, once resumed (it may already be resumed)
    ::SetEvent(startup.hEvent2); return TRUE;
}
```

这里的代码变得有点奇怪，所以请你仍然有耐心地坐在椅子上，并且保持很好的思考状态。CreateThread()调用 ResumeThread()（它又会调用::ResumeThread(m_hThread)）。

这时，我们创建的线程开始噼里啪啦响了。这是线程的一生中最脆弱的时候。它还没有_AFX_THREAD_STARTUP 指针、m_pMainWnd 指针以及一个真正的 MFC 线程所应该具备的其他重要信息。

而且，我们现在其实有两个线程！一个是我们最初调用 CreateThread()得到的线程（父线程），还有一个是处在幼稚期的线程。

现在，CreateThread()使得父线程暂停下来，使得它等待_AFX_THREAD_START::hEvent。此时，处在幼稚期的线程开始运行了！当它开始时，它会从 CreateThread()_beginthreadex()调用告诉它的位置开始：_AfxThreadEntry()。

_AfxThreadEntry()是一个非常长的函数，但是我们没有办法做到既能将它简略许多，同时又能不丢掉它所做的事情的环境。下面是当我们的幼稚期线程开始执行_AfxThreadEntry()时所做事情的一个详细的描述：

1. _AfxThreadEntry()（以后简记为_ATE）马上将它的_AFX_THREAD_STARTUP 参数的 pThread 域保存到真正的 CWinThread 指针里，这样它就不用总要很麻烦地引用 _AFX_THREAD_STARTUP 指针了。

2. 然后_ATE 调用 AfxGetThreadState()，它会返回父线程的状态。子线程会完整不动地使用绝大部分父线程的状态信息。它所改变的惟一一个域是模块状态，所用的值来自 _AFX_THREAD_STARTUP 参数。新的子线程从父线程那里“继承”到了模块状态信息（你也许希望能看看 THRDCORE.CPP 里的代码，来看看它到底是如何工作的）。

3. 接着，_ATE 调用 AfxInitThread()，该函数做了更多的状态处理工作。不同的是，AfxInitThread()会为该线程创建一个消息队列（SetMessageQueue）以及一些消息过滤器（还记得在程序清单 10-3 里讲到的消息过滤器处理函数吗？）。

4. 在 AfxInitThread()执行完成后，_ATE 创建一个局部的 CWnd 对象，并将它贴附到当前的主窗口里。方法是将它贴附到 CWinApp::m_pMainWnd::m_hWnd，然后使得 CWinThread::m_pMainWnd 指针指向它的地址。

5. 这时，处在幼稚期的线程已经在我们的眼皮底下逐渐成长起来了，并进入了青少年期！在进入成年期之前的最后一刻，当父线程将它推出家巢，使得它进入残酷的线程生活的真正世界里时，hEvent2 句柄被从_AFX_STARTUP_THREAD 指针里拷贝出来，因为该指针很快就要变成干烤面包了。

6. 最后，处在青少年期的线程准备变成一个成年线程了，并且准备通知父线程它已经长成一个大人了（记住，在这整个期间，父线程一直都在耐心地等待这个消息）。信号被转换为事件，它会使得父线程里的::WaitForSingleObject()被启动。真正的函数调用是::SetEvent（pStartup->hEvent）。

7. 现在，新的成年线程准备开始有所行动了，但是在最后说再见之前，父线程还有一个操作希望能完成。_ATE 调用::WaitForSingleObject(hEvent2，INFINITE)将新的成年线程放到冰川上，直到父线程向它发出最后的生命信号。

这时，在调用::WaitForSingleObject()后，让我们回到父线程，并且回到程序清单 10-14。父线程通过调用::CloseHandle()为它的子女清除 hEventHandle，并且如果调用者指定子线程以暂停的方式被创建，它会在子线程上调用::SuspendThread()。

现在，父线程能够确定指定的暂停状态已经被满足了。它通过在第二个事件的基础上调用 SetEvent 来将它的子线程送上它自己的旅途。

等等！现在就是真正有趣的地方了。此时，父线程走上了它欢乐的旅程，但子线程或者被暂停了，或者正在_ATE 里挣扎。让我们假设它正在那里挣扎，现在我们从离开的地方来重新开始_ATE。

8. 在 WaitForSingleObject()调用完成后，子线程向它的父线程最后说声再见，并且关闭了 hEvent2 记录的句柄。

9. 现在，如果 CWinThread::m_pfnThreadProc 是非空的，子线程会意识到它自己是辅助线程，并且通过将控制传递给 m_pfnThreadProc 来开始工作。记住，我们从调用

AfxBeginThread()时就将该参数传递进来了，而直到现在它才重见天日。

10. 如果子线程的 m_pfnThreadProc 为空，它会意识到在它的有生之年它将是一个 UI 线程。

我们在_AfxThreadEntry()处停止分析了线程，而向读者介绍一些与 UI 线程有关的概念。图 10-2 显示了两个线程所发生的事情，这样读者就能有一个清晰的概念了。

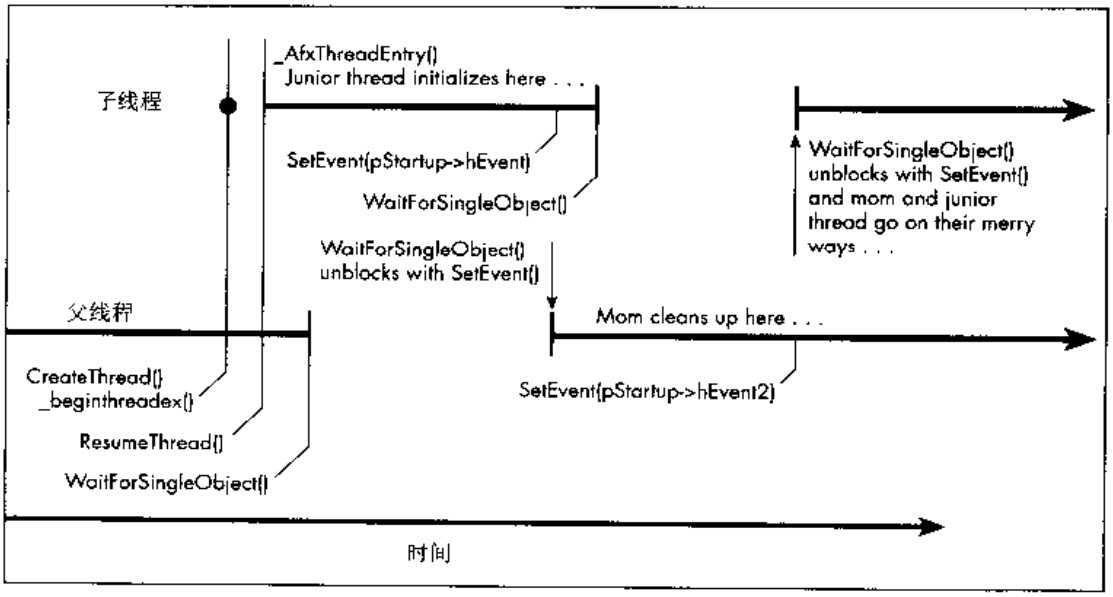


图 10-2 父线程与子线程的生命线

MFC UI 线程

辅助线程很容易理解：它们只有一个函数运行，并且这就是它们的全部。而 UI 线程就需要花费更多的精力，而且要做更多的事情，例如响应用户事件以及处理用户的输入。

辅助线程仅仅需要向 CWinThread 提供一个指向某个函数的指针，而 UI 线程需要从 CWinThread 派生出来，而且提供几个可重载的成员函数来使得它可以正确工作。被重载的函数如下所示：

- ExitInstance()——当线程结束时执行一些清理工作。
- InitInstance()——执行线程实例的初始化工作。
- OnIdle()——执行与线程有关的空闲时的处理工作。
- PreTranslateMessage()——消息过滤器。
- Run()——消息泵（这里就是非 MFC 消息的循环处理所位于的地方）。

CWinApp：最终的 UI 线程

听起来很熟悉吧？是的，你猜对了：CWinApp 是一个 UI 线程。由你的 CWinApp 的派生类来负责提供有用的 InitInstance()。CWinThread 提供了默认的 Run()和 OnIdle()函数。CWinApp 的其他工作就是要与用户打交道，而且它做得非常好。

我们会详细分析 Run()和 OnIdle()。但在我们开始分析之前，让我们首先来继续子线程的故事。在我们最后一次见到子线程时，它通过检查 m_pfnThreadProc 已经意识到了自己是 UI 线程，而不是辅助线程。

回到_AfxThreadEntry()和子线程

一旦子线程意识到自己是 UI 线程，它就会启动 UI 线程进程。程序清单 10-15 列出了_AfxThreadEntry()中与 UI 有关的部分。

程序清单 10-15 _AfxThreadEntry()中与 UI 有关的部分（在文件 THRDCORE.CPP 中）

```
//Initialization and synchronization with mom before here.
if (pThread->m_pfnThreadProc != NULL)
    nResult = (*pThread->m_pfnThreadProc)(pThread->m_pThreadParams);
else if (!pThread->InitInstance())
    nResult = pThread->ExitInstance();
else

    nResult = pThread->Run();
// cleanup and shutdown the thread
threadWnd.Detach();
AfxEndThread(nResult);
return 0;    // not reached
```

在程序清单 10-15 中，首先，我们会检查 m_fnThreadProc，然后_ATE 通过调用 m_fnThreadProc 来启动辅助线程。在“else if”语句里，有一个对 CWinThread::InitInstance()的调用，它会被用户界面的 CWinThread 的派生类所重载，用来做一些事情，并且不会返回 FALSE。如果它返回 FALSE，ExitInstance 会被调用，并且线程会退出。

接着，如果 InitInstance()返回 TRUE，CWinThread::Run()会被调用。它通常不会被用户界面的 CWinThread 的派生类所重载，所以真正的 CWinThread::Run()会被调用（你可以自由地重载 Run()，但好像没有许多情况需要你这样做）。

我们马上就会分析 Run()了，但是正如它的名字所暗示的，它将线程踢进了另一种模式，在这种模式下，它基本被挂起，而且为事件做出响应。

接着的就是_ATE 清理代码。当辅助线程的 m_fnThreadProc 返回时，或者 InitInstance()返回时，或者 Run()返回时该代码会被执行。清理代码将局部的 CWnd 主窗口对象分离出来，并且调用 AfxEndThread()。该函数会做更多的清理工作，并且最终会调用_endthreadex()（_beginthreadex()的对立函数）。

现在你已经知道了辅助线程与 UI 线程的整个生命的全部故事。然而，在这些故事里还有一些漏洞。最大的漏洞就是在子线程的运行阶段发生了哪些事情——在它生命的初期。

Run，Thread.Run！

在第 2 章里我们曾经许诺要详细分析 CWinThread::Run()，现在你的坚持将要得到回报了。在 MFC GUI 类的旅途中，我们实在是绕了一个很完整的圈子。

如果一定要我们说的话，CWinThread::Run()是 MFC 代码里惟一最重要的代码块。在 MFC 里没有其他任何代码比这 30 行代码定义了更多的关于 MFC 应用程序是如何工作的信息。我们已经看到了关于空闲进程、消息处理以及其他数不清的问题。所有这些都可以由 CWinThread::Run()回答，它的代码在程序清单 10-16 里。

程序清单 10-16 CWinThread::Run()——MFC 中最重要的 30 行代码（在文件 THRDCORE.CPP 中）

```
int CWinThread::Run()
{
```

```
        BOOL bIdle = TRUE;
        LONG lIdleCount = 0;
        for (;;) {
            // phase1: check to see if we can do idle work
            while (bIdle &&
               !::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE)) {
                if (!OnIdle(lIdleCount++))
                    bIdle = FALSE; // assume "no idle" state
            }
            // phase2: pump messages while available
            do {
                if (!PumpMessage())
                    return ExitInstance();
                if (IsIdleMessage(&m_msgCur)) {
                    bIdle = TRUE;
                    lIdleCount = 0;
                }
            } while (::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE));
        }
    }
```

首先要提醒的是：Run()仅仅有 30 行代码，但它实在是非常复杂，而它却往往隐藏了这一点。它所包含的内容比一眼看过去所能得到的内容要多得多。继续吧……

CWinThread::Run()是一个被清晰地分成两个部分的无穷循环：

- 阶段 1——它是一个“while”循环，当线程的消息队列里没有任何消息时，它处在闲置状态（也就是说，::PeekMessage()返回 FALSE）。
- 阶段 2——它是一个“do-while”循环，当有消息需要被分发时，它进行消息的分发（也就是说，::PeekMessage()返回 TRUE）。

这两个循环被 bIdle 标志和 LONG 型变量 lIdleCount 所“连接”。

让我们来详细分析这两个阶段。

空闲处理

在刚开始时，bIdle 为 TRUE，并且 lIdleCount 初始值为 0。bIdle 会一直为 TRUE，直到::PeekMessage()返回 FALSE，指示在队列里没有任何等待被处理的消息，从而线程进入了第一阶段的“while”块。在这个块里，OnIdle()会被调用。

Run()通过调用 OnIdle()来告诉线程用户没有执行任何操作，这样线程就可以做一些清理工作，或者一些它想要执行的其他操作。当线程处理完闲置操作后，它会返回 0。OnIdle()的参数用来指示线程已经有多长时间处在非活动状态了，这样 OnIdle()就可以对空闲处理做一些优先排序工作。当静止计数较低时，执行高优先级的空闲处理，当静止计数较高时，执行低优先级的空闲处理。

OnIdle()可以是默认的 CWinThread::OnIdle()，它可以通过 CWinApp::OnIdle()来直接进行调用。或者如果你在你的 CWinApp 派生类里重载了 CWinApp:: OnIdle()，该调用将会是你的 OnIdle()，而且你必须调用 CWinApp::OnIdle()。不管你是否选择进行空闲处理，CWinThread:: OnIdle()将被调用。

程序清单 10-17 包含了 CWinThread:: OnIdle()的一些缩减后的伪代码。

程序清单 10-17 CWinThread:: OnIdle()的伪代码（在文件 THRDCORE.CPP 中）

```
BOOL CWinThread::OnIdle(LONG lCount)
{
    if (lCount <= 0) {
        // send WM_IDLEUPDATECMDUI to the main window
        // **omitted
        // send WM_IDLEUPDATECMDUI to all frame windows
        // **omitted
    }
    else if (lCount >= 0) {
        // Call AfxLockTempMaps/AfxUnlockTempMaps
        // to free maps, DLLS, etc..
        // ** code omitted
    }
    return lCount < 0;  // nothing more to do if lCount >= 0
}
```

尽管我们不得不在程序清单 10-17 里省略掉了主要的代码块，你仍然可以看清 CWinThread::OnIdle()的主要控制流程。注意，如果调用了 OnIdle(-1)，它会强制进行一次 UI 更新。如果你正在等待某些很长的操作（串行 I/O ——数据库的访问），而且希望能在等待的过程中刷新 MFC 应用程序的 UI 组件，这个功能将非常有用。

CWinApp:: OnIdle()（在 APPCORE.CPP 里）处理空闲操作，并且当 lCount 为 0 或者为 1 时加进了一些处理函数。你的 OnIdle()处理函数应该为 MFC 保留这些“高优先级”的计数，而从 2 或者更高的地方开始空闲操作。

现在，我们已经学习了空闲操作，让我们回到::Run()。第一阶段继续调用 OnIdle()，当队列里没有消息时增加 lIdleCount 的计数。如果所有的空闲操作都已经完成了，bIdle 被设置为 FALSE，并且我们将进入到第二阶段。而且，如果某个消息在这个过程中到达了，一旦 OnIdle()返回后，我们也会进入到第二阶段（PeekMessage()在 while 循环里有执行的机会，并且可以返回 TRUE）。

消息的分发：CWinThread::Run()的第二阶段

在第二阶段时，线程进入“do-while”循环，并且一直待在里面，直到线程的消息队列里再没有消息为止。在“do-while”循环里，线程调用 PumpMessage()，它会导致消息被分发，然后它会调用 IsIdleMessage()。IsIdleMessage()确定当前正在被处理的、在 PumpMessage()里被设置好了的消息（在程序清单 10-12 里提到过）是否是空闲消息。空闲消息基本上是 Windows 发送的、指示用户当前在线程里没有做任何事情的消息。

例如，当发生一堆基于不同点的鼠标移动消息或者 WM_COMMAND 消息（指示某些程序状态已经改变的消息）时，OnIdle()需要被调用（想想将空闲进程转到运行状态的空闲消息）。某些消息，例如在同一点的鼠标移动消息、绘画消息，一般都不会改变程序的状态（WM_PAINT 响应程序状态的改变，但通常并不改变它），所以也就没有必要为这些消息调用 OnIdle()了。如果有一些排在消息队列里的窗口消息你不希望它们引起空闲操作，你也可以重载 OnIdle()。

如果是这种情况，bIdle 就会被设置成 TRUE（仅当 OnIdle()指示空闲操作完成时它才有可能被设置为 FALSE），并且 lIdleCount 被设置为 0。

空闲操作意味着忙等待吗？

关于 CWinThread 空闲操作的一个常见的问题就是应用程序会“忙等待”吗？也就是说，MFC 应用程序是否会占据处理器，并且使得 Windows 处理全部的空闲操作。

再看看 Run()，最终队列里将不会有任何消息，并且 OnIdle()会返回 FALSE。这会导致对 PumpMessage()的调用，它会在 GetMessage()上被阻塞。不要太关心空闲操作会占用处理器的几个周期，你仅仅要确信的是你的 OnIdle()执行的是优先级非常小的操作，而且它最终会返回 FALSE。

退出这个无穷循环的惟一途径是 PostMessage()返回 FALSE。仅仅当 PostQuitMessage()被调用，而且线程接收到 WM_QUIT 消息时，这种情况才会发生。在这种情况下，第二个阶段会调用 ExitInstance()并且返回结果。然后，线程的控制流就将回到_AfxThreadEntry，并且 UI 线程会被清除掉。

关于线程我们还有最后的一点需要说明，这样，在写多线程 MFC 应用程序时，你就会意识到潜在的一些危险。

线程、句柄以及对象

关于 MFC 的线程实现，要记住的一个关键点是它自己的消息队列。这看上去很简单，但实际上它所蕴涵的内容却很多。退回到第 2 章。还记得 MFC 通过将某些消息归类来将它的对象“贴附”到 Windows 上去吗？

因为每个线程都有它自己的消息队列，所以每个线程也都有它自己的 MFC 对象到 Windows 句柄的映射关系。在写 UI 线程时，记住这一点，并且尽量将线程局部化到线程不得不访问的少数几个用户界面对象。你可以通过在线程间不断使用旧的::PostMessage()来避开这个限制，但这样做太麻烦了。

现在，你已经了解了许多关于 MFC 线程实现的关键内容。这些知识能帮助你使用 MFC 写出更好的多线程程序，并且能节约好多时间和精力。在进入下一章之前，我们想对在本章里讲到的关键内容做个总结，并向读者指出可以进一步学习的 MFC 领域。

结 语

在本章里，我们对 MFC 的 DLL 学习了以下内容：

- DLL 与线程的实现依赖于内部 MFC 状态类。MFC 的状态将数据分到不同的逻辑范围中，从而使得线程和 DLL 不会破坏对方的数据。
- 在 MFC 4.0 里，有 3 个 MFC 的 DLL 模式：
  - 普通 DLL，静态链接（以前称为_USRDLL）。
  - 普通 DLL，动态链接（以前并不存在）。
  - 扩展 DLL，一般采用动态链接（没有改变名字）。

  两种通用 DLL 都可以被非 MFC 应用程序所调用。扩展 DLL 仅仅用来扩展已存在的 MFC 应用程序。MFC 应用程序也必须使用 MFC DLL。一个 DLL 有可能在模式 2 和模式 3（MFC 4.0 的 DLL 是一个例子）下工作，但这种做法比较罕见，而且也比较麻烦。
- 在 MFC 里，扩展 DLL 被创建时要使用_AFXDLL 标志。
- 扩展 DLL 有一些资源和其他信息需要在运行时被检索。CDynLinkLibrary 是它的辅

助类，并且在实现中起到了关键性的作用。

- MFC 不得不对一些访问 CRuntimeClass 信息的宏进行特殊处理，使得在扩展 DLL 里静态对象能被正确创建。

另外，我们也学习了 MFC 线程的以下知识：

- 它们可以有两种类型：辅助线程与 UI 线程。两者都是用_beginthreadex()来创建，以_endthreadex()来结束。
- MFC 的 CWinThread 类提供了所有对线程的支持。CWinThread 最重要的部分是 CreateThread()成员函数。
- CWinThread::CreateThread 创建线程，并且使用_AfxThreadEntry()来为线程提供执行路径。
- UI 线程（例如 CWinApp）将它的生命的绝大部分都花费在了::Run()里。在那里，它们对消息进行响应，并且处理空闲操作。
- 每个 MFC 开发人员需要注意的 MFC 里最关键的代码行是 CWinThread::Run()。CWinThread::Run()是 MFC 的核心——它通过用户界面的通道分发消息，向用户发送信息，并从用户那里接收信息。

更多的信息

如果你希望能对本章所讲述的内容做进一步研究，下面是我们的一些建议：

- MFC 其实是几个 DLL，每个都有不同的初始化操作。可以通过阅读以下的源文件对每个 DLL 的要求和实现有更多的了解：DLLDB.CPP、DLLINIT.CPP、DLLMODULE.CPP、DLLNET.CPP、DLLOLE.CPP 以及 OLEDLL.CPP。
- 分析 AFXTLS_.H 与 AFXTLS.CPP，然后回答以下的问题：
  - CThreadSlotData 做了些什么，并且是如何实现的？
  - 推断出下面的调用做了些什么，并且是如何实现的？

```
AfxLockGlobals(CRIT_DYNLINKLIST);
```

- 分析 WINHAND_.H 与 WINHAND.CPP，然后回答以下的问题：
  - CHandleMap 为什么要有 CWinThread 这个友元？
  - 为什么 CHandleMap 有两个映射？
- 在 CWinApp::Run()里用调试器设置断点，并且通过对几个消息的运行跟踪使得对它的工作方式有更好的了解。
- 分析 AFXMT.H、MTCORE.CPP 以及 MTEX.CPP 里 MFC 同步类的实现，回答以下的问题：
  - CSyncObject 基类提供了哪些功能？
  - CSingleLock 与 CMultiLock 之间的根本区别在哪里？
  - 对每个同步类的真正的 Win32 API 调用发生在哪里？

下一章

在下一章里，我们将开始分析 MFC 里 OLE 的实现。我们将从最基本的知识开始（MFC 如何支持 COM），直到各种 OLE 组件。

CHAPTER

11 用 MFC 实现 COM

Visual C++对 OLE 的支持是众所周知的事情。MFC 通过基本的 AppWizard 对 OLE 复合文档（compound document）和自动化服务器（automation server）提供支持，还通过基本的 ControlWizard 对 OLE 控件提供支持。除此之外，它还提供一组类（class），通过这些类，可以为 MFC 应用程序添加各种不同的 OLE 特性。虽然 MFC 提供的对 OLE 的支持范围很广，但对 OLE 的功能的支持却很粗糙。

在高层包括 OLE 特性[例如容器（container）支持、服务器（server）支持和自动化（automation）支持]很简单：只要打开 AppWizard，选定你所需要的选项，让它自动给你生成代码，然后在这个框架的基础上直接加入自己的应用程序代码即可。到此，就已经得到了一个复合文档或是一个自动化对象的实例了。（虽然 AppWizard 不能帮你生成全部实例，但是比起全部都用手做的情况了已经简单多了。）

然后，有的时候你必须在更低的层次下编程。例如，你要开发一个用来实现客户定制接口的 COM（Component Object Model，组件对象模型）类，就必须抛开标准的框架，自己直接编写 COM 代码。又或者，你想从 MFC 的 OLE 类中派生（derive）出一个子类，而这个子类要实现另外一个接口，也必须依靠 MFC 提供的对更低层的支持。

本章的内容就是描述 MFC 如何实现 COM。首先，我们将复习 OLE 和 COM 的概况。然后我们介绍 MFC 对 COM 的支持，这部分内容将包括 CCmdTarget、接口映射表（interface map）和 COleObjectFactory，还有一些没有文档说明的方法。这些方法是用来实现 in-proc 服务器的。最后，我们将会看见如何写一个 COM 类。这个 COM 类实现了一个微软定义的接口（IPersist）和一个客户定义的接口（IMath）。

MFC 和 OLE

当你进入 OLE 编程的世界时候，会发现 MFC 给你提供了很多必要的 OLE“样板”代码。AppWizard 会帮我们生成基本的 OLE 代码，我们所要做的全部工作就是在这个框架的基础上，加入应用程序所要实现的功能。在很多方面，MFC 和 OLE 之间的关系很类似于编译器和汇编语言之间的关系。你能够在更高程度的抽象的基础上做绝大部分你想做的事情，当然，你也可以一头扎进汇编语言一级的编程里。

当我们编写支持 OLE 的应用程序的时候，可以利用 MFC 提供的高层次的支持做大部分工作。然而，和高级语言一样，有些时候，我们还是得深入到系统一级去工作。这种情况在

OLE 中，就是表示要使用 COM。记住，MFC 是一个应用程序框架——一台已经运行良好的机器，你只要加入你所需要的功能即可。对于实现标准的 OLE 特性，例如 OLE 文档容器、服务器和自动化服务器，这就已经足够了。然而，如果需要设计一个客户接口或是实现一个 MFC 之外的接口，就需要在底层利用 COM 进行编程了。

幸运的是，MFC 也提供了对 COM 编程的支持。MFC 对 COM 的支持主要体现在 CCmdTarget、COleObjectFactory 和 MFC 接口映射宏上。在很多 OLE 类中，MFC 不遗余力地支持了 UDT（Uniform Data Transfer，统一数据传输）、OLE 文档应用程序、自动化服务器和 OLE 控件。我们将会很快看到这些类。但是在我们看 MFC 对 OLE 特性的支持之前，我们先要对 MFC 对 COM 的支持有个大致的认识。我们将从简要概述 COM 开始。

COM

OLE 是微软的对象技术的名字。OLE 是一种整合软件的统一方法，它使得软件可以随时间而演变，是一种整合技术。OLE 的主要目标就是能够使得应用程序被外界所认识和理解。也就是说，使用 COM 编程的目的是使得软件从源代码一级的重用上升到二进制一级的重用。举个例子，想像一下现在人们是如何编写应用程序的：人们通常只使用一种语言（C++当然是其中一种）来编程，当你写完应用程序，程序通常只包含一个可执行文件，也许还有一些相关的动态链接库（DLL）。

在很多的情况下，上面的方法都可以完成地非常好。然而，使用这种方法就意味着你的应用程序是封闭的：它只有面向自己的入口。要想共享应用程序的功能和数据只有一种方法，那就是通过一些不一致的特殊的方法，比如通过共享 DLL 或使用 DDE。

COM 通过引入一个一致的协议解决了这个问题，那就是允许软件组件互相通信。微软的组件对象模型将在未来的 Windows 实现中扮演十分重要的角色，所以值得我们去研究一下这种技术。

COM 消除了 Jeff Duntemann（Visual Developer 杂志的编辑）称之为“边界”的东西，也就是说 COM 调整了应用程序，使得它们之间知道怎么互相通信。使用 COM 编程，可以保证在一个系统下的所有组件之间通过一致的方法进行通信。在当前，共享数据和功能的机制是不一致的。在很多情况下，共享功能需要各个参与的应用程序都必须相互了解，又或需要一些依赖于编译器的信息。因为 COM 定义了二进制一级的标准，它的编译器和编程语言是相互独立的，所以 COM 有效地定义了一个对象之间在二进制一级相互通信的协议。

COM 编程产生了一些传统编程所没有情况。读者需要熟悉这些条款才能很好地了解 COM。这些思想如下所述：

- 定义 COM 类。
- 通过二进制防火墙提供功能。
- 标识 COM 对象（在编译以后就确定，跟语言无关）。
- 管理一个对象的生存期。
- 实现运行时发现功能（discoverability）。

- 包装 COM 对象（在 DLL 和 EXE 服务器中）。
- 从客户的角度使用 OLE 类。

这些概念将在下面的部分涉及到。

何为 COM 类

在学习 COM 的时候，必须牢记一点，那就是：COM 所做的全部工作就是在二进制一级共享软件。这一点和面向对象语言（比如 C++）所体现的思想是完全不同的，前者正好是后者的补充。

让我们先来回顾一下 C++的一些基础知识。C++中的类是数据以及作用在这些数据上的成员函数的集合。例如程序清单 11-1 和 11-2 中的一个简单的 C++ string 类。

程序清单 11-1 C++ string 类的头文件

```
class string {
   char m_achString[256];

public:
   string();
   char * GetString(char *);
   void SetString(char *);
   int GetLength();
};
```

程序清单 11-2 C++ string 类的源文件

```
string::string() {
   m_achString[0] = '\0';
}

string::GetString(char *psz) {
   if(psz) {
      strncpy(psz, m_achString,  255);
   }
   return
      psz;
}

void string::SetString(char * psz)  {
   if(psz) {
      memset(m_achString, 0, 256);
      strncpy(m_achString, psz, 255);
   }
}

int string::GetLength() {
   return strlen(m_achString);
}
```

这个类只有一个接口（也就是所有的公共成员函数）。在这个类的定义下，客户所要做的全部工作就是用 new 操作符创建这个类的一个对象即可，然后就可以调用你所需要的成员

函数。当使用完毕之后，调用 delete 删除这个对象。

C++类是组织代码的一种很好的方法，但是，这种方法不能在二进制一级共享代码。如果想共享程序清单 11-1 和 11-2 中的 string 类，该如何做呢？一种解决办法当然是导出这个类。但是，这样做的话，程序员就必须考虑名字分割（name-mangling）的问题，而这是与编译器相关的。如果编译器提供商决定改变名字分割体系，又或是客户决定使用另外一个编译器，你就有麻烦了。也许你可以直接定义一组 C 的 API 函数，然后导出它们，但是这样的话，就不能利用类所带来的好处了。如果使用 MFC，可以将类放在扩展名为 DLL 的 AFX 中，但是如果将来你想改变 m_achString 的大小或是加入一个成员变量的时候，就必须重新发布这个 DLL。不幸的是，并没有统一的方法能够在不用的应用程序之间共享 C++类。

这就是为什么要有 COM，COM 就是为了解决这些迫切的问题而被引入的。让我们首先来看看 C++类和 COM 类之间有什么区别：

1. C++使用操作符 new 来实例化一个对象，而 COM 对象是通过 API 的函数来创建的。

2. C++中使用 delete 来删除一个对象，而 COM 中使用引用计数来控制对象的生存期。当引用计数为 0 时（此时没有客户使用这个对象），COM 对象自动删除。

3. C++使用特殊的机制来控制运行时类型信息（run-time type information）和类型转换（typecasting），而 COM 类通过成员函数来支持运行时类型信息和类型转换。

4. C++类能够提供成员函数和数据，以供外部访问；而在 COM 中，类只能提供成员函数。

5. COM 类没有规定具体的布局。

6. COM 引入了一套接口的规范来形式化对象指针的概念。我们将在下面详细介绍这个。

总之，从根本上来说，就是 COM 是在二进制一级的，而 C++是在源代码一级的。使用 COM 对象基本上要经过以下几个步骤：（1）给操作系统发送实例化一个 COM 对象的请求；（2）得到对象的一个接口指针；（3）使用这个接口；（4）最后释放这个接口。如果你接触过 OLE，那就至少听说过“接口”这个术语。现在的一个主要问题是：究竟何为一个接口？

COM 接口

接口规范是 COM 最为重要的特性之一。从严格意义上来说，一个接口是一组（语意上相关的）函数。一旦一组函数能实现某个功能，它们就能被组合在一起称之为一个接口。从实现的技术上来说，一个接口就是一组等待被实现的纯虚函数。

认识接口的另外一个很好的方法是把接口看作是一组相关的函数的捆绑。例如，在 OLE 中有一个接口叫 IDropTarget（实际上这个接口是微软定义的），它包含了以下一些函数：DragEnter()、DragOver()、DragLeave()、Drop()。我们会发现这些函数都是用来实现拖放数据的传输操作的。因为这些函数是相关的，所以它们被放在同一个接口里面。

COM 接口有别于 COM 类。COM 类将接口提供给客户，客户通过接口与 COM 对象进行通信。COM 接口作为 COM 类和客户之间联系的桥梁，它定义了客户和 COM 类之间通信的方法，所以是强类型的。通过使用接口作为协议，不同的类可以用各自不同的方法实现同一个接口，而且可以在二进制形式下互相交换。如果所有的操作都按照接口的定义统一进行，

那么将不会有任何问题。例如，如果有几个类都实现了自动化（automation），也就是实现了IDispatch接口，这其中也许有两个类使用了完全不同的方法实现了IDispatch。如果接口是按照预期的方法操作的，那么这几个COM类就可以相互转换，或者可以相互替代而不会有什么问题。

每一个COM接口都是惟一的。这就导致了接口的另外一条很重要的特性：接口是不能被改变的。一旦一个接口被定义了并且被发布，它就不能被改变。一个接口的新的版本就作为一个全新的接口，并且被分配一个新的标识符——即使仅仅是给接口中的某个函数加了一个参数。这就意味着一个新的接口永远不会和旧的接口产生冲突。

客户和COM类进行通信的惟一途径就是通过接口。接口指针对客户屏蔽了COM类内部的所有实现细节。给COM接口增加数据会出现问题，因为加入数据是与编译器相关的。

有关接口的最后一个要点是COM类可以实现多个接口，也就是说一个COM类可以提供多个服务。例如，一个实现了复合文档服务器的应用程序必须提供多个接口，包括IPersistStorage、IOleObject、IDataObject、IOleInPlaceObject、IOleInPlaceActiveObject（现在不必确切地知道这些接口都是做什么用的，只要知道是让OLE文档能起作用即可）。MFC通过它的文档/视图结构来支持OLE文档技术。MFC在COleServerDoc中实现这些接口。

我们会发现所有的COM接口的名字都是以字母I开头的。例如IDataObject，它包含的方法在实现统一数据传输中会用到。一个叫IDispatch的接口用来处理OLE自动化。也许，当你的代码不能通过编译而让你感到万分恼火的时候，你会想把接口名字定义为ICaramba。

GUID

在进入下一部分之前，让我们先来简要地谈谈GUID（Globally Unique Identifier，全球惟一标识符）。因为在COM中，它是无所不在的。

让我们先考虑一下标准的C++类。当你书写一个C++类的时候，你给它起了个名字（比如“class foo”或别的什么名字），成员函数和成员变量同样的也有自己的名字。编译器通过这些信息编译和链接你的应用程序或库。这些标识对编译器来说是很重要的，因为它要通过这些标识来解决在整个程序中变量地址的定位和函数的调用。然而，当可执行文件或是库一旦生成之后，这些标识就消失了（惟一的例外是导出的函数）。在一个面向对象的系统中，客户能够使用不同的语言来实例化一个对象并且使软件能够在二进制一级进行通信。要做到这一点，就要求标识系统在编译和链接完成之后还能保持不变。

COM使用称之为全球惟一标识符（GUID）的128位数字来标识OLE类（以及接口，这在后面将会看到）。一个GUID一般是下面的形式：

```
{00020900-0000-C000-000000000046}
```

实际上，这个GUID代表了Windows文档中一个字。它是作为OLE文档所包含的一个对象存在的。

用这些数字定义OLE类和接口贯穿着整个系统。由于它们足够大（从$0\sim2^{128}$），所以这些数字之间的冲突就可以被有效地避免了。事实上，微软的COM规范上说即使是用来产生

GUID 的最普通的算法在理论上“也能够在未来的 3240 年内以每台机器每秒 10000000 的速度分配 GUID”。

COM 使用这些数字来惟一地标识 COM 类和接口。有一点区别是，COM 类的 ID 用“CLSID”作为前缀，而接口 ID 使用“IID”作为前缀。比如，我们将使用的一个 COM 类叫 CoMath，它用 CLSID_CoMath 来标识。而一个微软定义的接口叫 IUnknown，却是用符号 IID_IUnknown 来标识。

剖析 IUnknown 接口

在所有定义的接口中，也许 IUnknown 是其中最重要的一个。它是所有接口的共同祖先，并且在 COM 中规定，所有 COM 类和 COM 接口都必须实现 IUnknown。读者将很快看到 IUnknown 怎么样来影响一个 COM 对象。可以在 OLE 的头文件中找到 IUnknown 的定义，其定义如下：

```
struct IUnknown {
   virtual HRESULT QueryInterface(IID& iid, void **ppvObj) = 0;
   virtual ULONG AddRef() = 0;
   virtual ULONG Release() = 0;
};
```

注意：微软使用类似于 DECLARE_INTERFACE 和 STDMETHOD 等宏将以上的定义给隐藏了，目的是为了处理 16 位和 32 位实现中的不同的地方。为了更加清晰，就省略了它们。HRESULT 拥有丰富的错误类型代码，能够反映一个函数返回值的各种信息。

注意到 IUnknown 仅仅是定义了 3 个纯虚函数的结构。任何实现 IUnknown 接口的类都必须实现这个 3 个函数。这几个函数支持所有 COM 类所需要的两方面功能：对象的生存期管理和接口协商（negotiation）（也就是运行时发现功能）。

对象生存期管理

COM 对象的一个很重要的特性就是它们能够自己控制生存期。COM 客户可能拥有一个对象的多个接口指针（也就是 COM 对象可能一次有多个接口在使用）。另外一个需要考虑的情况就是也许一个 COM 对象在同一时刻可能被多处引用。如果 COM 对象还在使用，它们就必须还保留在内存中。也就是说，COM 对象必须控制自己的生存期，当它知道自己没有被使用的时候（也就是外面没有地方保留了它的任何接口指针），必须能够把自己删除。COM 对象使用自析构而不是依靠客户将它们删除。为了实现这个功能，COM 对象中保留了引用计数（很像在 DLL 中保留的引用计数）。只要引用计数大于零，就表示 COM 对象还在使用中。

COM 对象的引用计数由 IUnknown 接口的 AddRef()和 Release()这两个函数来管理。任何一个客户得到了一个对象的接口指针，这个对象的引用计数就要调用 IUnknown 的 AddRef()来加 1。当客户使用完了一个接口，就调用 Release()将引用计数减 1。当对象不再被使用的时候（也就是引用计数为 0），对象可以从内存中将自己删除。

为了更好地说明这个，让我们来看个简单的例子：一个 COM 对象 CoSomeObject，它只

支持一个接口 IUnknown。你可以像程序清单 11-3 那样用 C++来定义这个类：

程序清单 11-3 CoSomeObject：一个简单的 COM 类

```
class CoSomeObject : public IUnknown {
public:
   HRESULT QueryInterface(IID& riid, void FAR* FAR*ppvObj);
   ULONG AddRef();
   ULONG Release();

   CoSomeObject();

private:
   DWORD m_dwRefCount;
};
```

很自然，我们会在构造函数里面将引用计数初始化为零。另外，实现 AddRef()和 Release()也是比较简单的。AddRef()和 Release()函数的实现如程序清单 11-4 所示。

程序清单 11-4 CoSomeObject 的实现代码

```
CoSomeObject::CoSomeObj() {
   m_dwRefCount = 0;
}

ULONG CoSomeObject::AddRef() {
   return ++m_dwRefCount;
}

ULONG CoSomeObject::Release() {
   DWORD dw;
   dw = m_dwRefCount--;
   if (m_dwRefCount == 0)
      delete this;

   return dw;
 }
```

程序清单 11-3 和 11-4 中的代码是实现 AddRef()和 Release()的标准代码。注意一下在 Release 中删除了对象的 this 指针，这也许看着有点奇怪，但是这确实是 C++的合法语法，而且确实做到我们想做的——让对象自己控制生存期。因为只有对象自己知道它是否还在使用，所以对象就必须自己来删除自己。

IUnknown 所解决的第二个问题是接口协商（也就是运行时发现功能）。这个功能由 QueryInterface()函数来实现。

接口协商

如果想有效地完成一个任务，COM 类就必须能够支持多个接口（因为 IUnknown 是必须要有的一个，所以 OLE 必须实现另外一个接口，比如 IDataSource 或是 IOleObject）。当一个客户拥有了对象的一个接口指针，该客户就应该能够发现这个对象所支持的别的接口。在 COM 中使用 QueryInterface()这个函数实现这个功能（这个函数是通过 IUnknown 接口提供的）。QueryInterface()的原型如下：

```
HRESULT QueryInterface(REFIID riid, LPVOID far* ppvObj);
```

QueryInterface 有两个参数：(1) 一个惟一的 128 字节的数字，用来标识所要得到的接口（也是一个 GUID）；(2) 一个用来存放接口指针的地方。当客户调用一个接口的 QueryInterface 时，COM 对象就用请求的接口 ID 和它所支持的所有接口 ID 逐个比较。如果 QueryInterface 发现有一个接口满足请求的 ID，它就返回一个指针给请求的接口，同时调用 AddRef()对 COM 对象的引用计数加 1。由于 COM 对象分发了一个接口，这个是它的标准的应用方式，所以必须对它的引用计数加 1。

现在，我们假设 CoSomeObject 支持另外一个叫 IPersist 的接口（其实 IPersist 是微软定义的一个实际上存在的接口）。程序清单 11-5 和 11-6 显示了一个 QueryInterface 的实现，它根据 riid 参数的请求返回一个接口指针。

程序清单 11-5 CoSomeObject 的定义

```
class CoSomeObject : public IUnknown, public IPersist {
   // IUnknown methods
   virtual DWORD   AddRef(void);
   virtual DWORD   Release(void);
   virtual HRESULT QueryInterface(REFIID,
                                  LPVOID FAR*);

   // IPersist methods
   virtual HRESULT GetClassID(LPCLSID pclsid);

};
```

程序清单 11-6 CoSomeObject 的成员函数

```
HRESULT
   CoSomeObject::QueryInterface(IID& riid,
                                void FAR * FAR * ppvObj) {
    HRESULT hr = ResultFromSCode(E_NOINTERFACE);
    *ppvObj = NULL;
   if (riid == IID_IUnknown) {
     *ppvObj = (IPersist *)this;
      hr = NOERROR;
   } else if(riid == IID_IPersist) {
      *ppvObj = (IPersist *)this;
      hr = NOERROR;
   }
   return hr;
}
```

这是实现 IUnknown::QueryInterface 的一个方法（我们将在以后看到别的实现方式）。让我们看一下它是如何发现接口的：QueryInterface 用 riid 参数去检查这个对象的所有可用的接口，直到找到一个匹配的（或是返回一个错误信息）。这个类的 this 指针被转换为 IPersist*类型（因为 IPersist 是从 IUnknown 接口派生的，所以用那个指针可以调用 IUnknown 的函数），并且接口指针通过 ppvObj 参数返回。这个实现方案适合那种通过多继承实现 COM 类的类。正如我们即将看到的，多继承仅仅是实现 COM 类的一种方式。和 AddRef()和 Release() 一样，这个仅仅是实现 COM 接口查询的一个标准代码，当然不是实现接口查询的惟一方法。我们

可以使用查询表或是别的更优化的方法。而实际上 MFC 使用类似于消息映射的机制来实现 QueryInterface()。

从客户的角度来看整个过程：调用/使用/释放

从客户的角度来看，IUnknown 的对象生存期控制机制给 COM 接口的使用强加了一个标准的协议。使用 COM 对象的调用/使用/释放的规则大致如下：

1. 调用某个方法得到一个接口（通过初始化一个对象或是通过一个存在的接口调用 QueryInterface()）。
2. 随你所愿使用接口的函数。
3. 当使用完接口后，通过接口指针调用 Release()对对象的引用计数减 1。

例如，OLE 通过一个叫 IMalloc 的接口进行内存分配。得到 IMalloc 指针的一个方法是通过 COM 的一个叫 CoGetMalloc()的辅助方法。如果通过 IMalloc 得到了一个 OLE 的任务内存分配器指针，你就可以调用它的任何方法，包括 Alloc()和 Free()。然后当你使用完了接口指针后，只要调用 IMalloc 的 Release()方法。由于与 IMalloc 指针相对应的一个对象保存了一个引用计数，所以它将在必要的时候将自己从内存中删除。程序清单 11-7 表示了调用/使用/释放的协议。

程序清单 11-7 使用 IMalloc 接口的一个例子

```
HRESULT AllocMem() {
     LPMALLOC pMalloc;
     LPSTR lpsz;

     ::CoGetMalloc(MEMCTX_TASK, &pMalloc);
     if (lpsz = LPSTR(pMalloc->Alloc(256)))
        result = NOERROR;
     else
        result = ResultFromScode(E_OUTOFMEMORY);

     if(lpsz)
        pMalloc->Free(lpsz);

     pMalloc->Release();

     return result;
}
```

让我们来看一下调用/使用/释放协议是如何工作的：CoGetMalloc()得到一个 IMalloc 指针。IMalloc 中有几个方法是管理内存用的，包括 Alloc()、Free()、Realloc()和 DidAlloc()。随你所愿使用这些方法。别忘了在使用完接口后，要释放接口指针。必须注意无论函数的执行成功与否，都必须释放接口指针。

COM 对象服务器

很自然，COM 类也必须存活于某处：或是 DLL，或是 EXE 文件中。COM 类存在于服务器(server)中。COM 服务器通常分为两种：进程内(in-proc)服务器和进程外(out-proc)

服务器。进程内服务器是 DLL 形式的，它们和客户在同一个进程空间内。进程外服务器作为 EXE 文件形式存在，和客户在不同的进程空间内（有时候甚至在不同的机器中）。

进程内服务器（DLL）

进程内服务器在 Windows 系统下通过 DLL 来实现。一个进程内服务器能够被载入客户的进程空间内并且能够作为“进程内对象”来使用。

进程内服务器的最大的好处是它的速度：它们通常都是很快的。在 EXE 和 DLL 之间进行接口函数调用是很快的，因为程序和 DLL 共享同一个进程空间。然而，进程内服务器可能造成危险：如果 DLL 崩溃了，也许整个进程空间都有可能随之崩溃。

进程外服务器（EXE）

进程外服务器在独立的进程空间内运行（或是在同一台机器上，或是在不同的机器上）。与客户在同一台机器上的进程外服务器称之为“本地服务器”，而和客户在不同机器上的进程外服务器称之为“远程服务器”。进程外服务器以 EXE 文件的形式存在。

EXE 服务器比 DLL 有更强的鲁棒性。如果客户代码在另外一个进程空间内调用对象的接口函数，客户和服务器将会被很好地各自保护起来，不会影响对方。如果服务器由于某种原因崩溃了，客户进程将不会受到什么影响。但是，尽管有更强的鲁棒性这个好处，但使用进程外服务器也要付出额外的代价，那就是列集（marshaling）。

这儿有一个问题：一个进程空间内的地址（数据的地址和函数指针）在另外一个进程空间内将不会起作用。所以，在另外一个进程空间内的客户代码如何调用函数以及传递参数就成了一个问题。解决的方法是用远程过程调用（RPC）和列集（marshaling）。为了理解它的工作原理，让我们稍微深入了解一下列集的机制。

所有的函数调用和参数传递都必须在同一个进程空间内进行，这是所有在 Windows 系统中运行的程序所必须遵循的原则。但是，COM 的结构允许任何客户通过接口指针与任何对象通信，而无论这个对象是否和客户在同一个进程空间内。为了支持这个，OLE 使用了远程过程调用机制。下面是它的运行机制：

对于每一个接口，都在客户代码中有一个代理（proxy）。一个代理是一个接口的进程内实现。因为代理代码是和客户在同一个进程空间内的，所以客户代码能够很容易调用它。但是，客户还是不能直接调用真正的服务器代码。

在服务器端，对于每一个特定的接口都建立了一份存根（stub）代码（在服务器进程空间内）。存根是 RPC 传输的接收端，它把代理传过来的函数调用和参数传递映射为服务器进程空间内的函数调用和参数传递。当客户通过接口进行函数调用时，代理将参数打包为一个 32 位的可移植的数据结构。也许在拷贝缓冲区、字符串、数字和其他参数的时候都要经过这个过程。当代理将参数打包在一起之后，它就对服务器进程产生一个远程过程调用。在服务器这端，存根保存着真正的接口指针、函数和相关的数据结构。存根将参数从包中解开以在本进程空间内调用。当一个函数调用返回时，存根将返回值和其它包含返回信息的参数打包，然后发送回给代理。代理解开这个包，取出返回值和传出参数供客户使用。术语“列集”就是指在不同的进程空间内对参数进行打包和解包的过程。

图 11-1 显示了代理/存根机制是如何工作的。

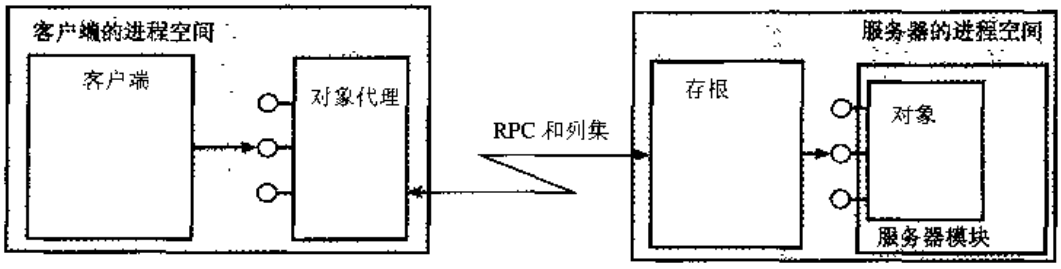


图 11-1 COM 对象在不同进程空间内的通信

类厂

要使 COM 成为真正的二进制一级的标准，它的服务器还必须能够生成代表客户端的对象。在 C++中，新的对象只能由 new 操作符来产生。而在 COM 中，已经把软件放到二进制一级的层次来考虑。COM 的一个主要的目的就是使形形色色的各种软件能够互相混合着使用。new 操作符是“C++特性的”，在客户用来建立客户端软件的语言也许根本就不支持 new 操作符。比如，Visual Basic，一种经典的 OLE 自动化的客户端语言，根本不关心如何分配内存。要使 COM 在各种进程空间内能协调工作，对象的实例化必须通过一个 API 函数来实现。所以要使 COM 能顺利工作，必须在服务器端建立一个代理，用来响应客户对 COM 对象的创建请求。

让我们先来回顾一下 C++中是如何分配对象的。C++对象通常是由客户来分配的（在很多时候就是程序自己）。而 C++程序在用完对象后，应该将对象清除。程序清单 11-8 就是一个例子。

程序清单 11-8 在标准 C++中的对象生存期控制

```
class Foo {
   char m_szName;*
public:
   Foo(char * szData);
   ShowData() {
      cout <<  m_szName << "\n";
   }
};

int main() {
   Foo *pFoo;

   pFoo = new Foo("This is the Foo Class");
   pFoo->ShowData();
   delete pFoo;

   return 0;
}
```

实际上分配和删除对象的整个过程都是在某个程序内进行的，而且产生对象的代码可以直接访问类的所有的公共（public）成员变量和成员函数。

相对应地，客户通过一个标准的 API 函数发出请求，服务器根据请求分配 COM 对象，而且这是产生对象的惟一途径。OLE 的目标是成为二进制标准。因为客户端和服务器可能在不同的进程空间内，所以使用 new 操作符创建对象是行不通的。这个任务就落到了另外一个 COM 类的头上，那就是类厂（class factory）。

COM 通过类厂来创建对象。对于进程空间内的每一个 COM 类都有一个类厂与之相对应。然而，也许类厂这个名字也许会有点误导嫌疑：类厂真正产生的是对象，而不是类。

类厂仅仅是一个比较特殊的 COM 类。和别的 COM 类一样，类厂也实现 COM 接口。这里面有一个很重要的接口是 IClassFactory。IClassFactory 包含两个函数：CreateInstance()和 LockServer()。客户通过调用 CreateInstance 来创建 OLE 类的实例。LockServer()增加整个服务器的引用计数。通过使用引用计数，服务器就能在内存中保存一份实例，直到没有客户使用为止。下面是 IClassFactory 的大致结构：

```
struct IClassFactory {

   // IUnknown methods
   virtual HRESULT QueryInterface(IID& iid, void **ppvObj) = 0;
   virtual ULONG AddRef() = 0;
   virtual ULONG Release() = 0;

   // IClassFactory methods
   virtual HRESULT CreateInstance(IUnknown * pUnkOuter,
                                  REFIID riid,
                                  VOID ** ppvObject) = 0;
   virtual HRESULT LockServer(BOOL fLock) = 0;
};
```

IClassFactory 跟别的 COM 接口没有什么区别。它也实现了 IUnknown 接口。

每一个 COM 类都有它自己的类厂（在源代码一级）。比如，在图 11-2 中，一个 COM 服务器包含了一个叫 CoSomeObject 的 COM 类。COM 服务器包含了两个 COM 对象：CoSomeObject 和它的类厂。

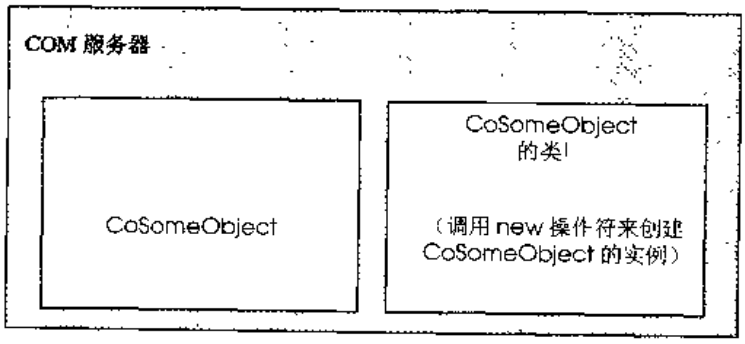


图 11-2　在 COM 服务器中 CoSomeObject 和它的类厂

当客户需要使用 COM 类时，它是如何得到类厂的呢？进程内服务器通过一个预先定义的函数将类厂提供给客户，而进程外服务器将类厂注册到 Windows 的注册表中。

也许你已经注意到 IClassFactory 这个名字不是那么确切。正如一个玩具工厂生产玩具，而一个 IClassFactory 却并不产生类（产生类是程序员做的工作）。IClassFactory 真正产生的是

对象，即给定类的实例。也许，它的名字改为 IObjectFactory 更为贴切。这个也许是微软为什么把它的 MFC 类厂实现称为 COleObjectFactory 的原因。

在进程内服务器中提供类厂

进程内服务器都是 DLL，所以很容易导出函数。进程内服务器通过导出函数（exported function）来提供它们的类厂。这个函数叫 DLLGetClassObject()。

客户可以通过 CoGetClassObject()或 CoCreateInstance()这两个 COM API 函数的任何一个来创建一个 COM 对象。无论客户通过哪个方法创建了一个 COM 对象，COM 都调用了服务器的 DLLGetObject()函数来得到相应的 COM 类的类厂接口指针。客户有了类厂的接口指针后，就可以调用类厂的 CreateInstance()来创建一个类的实例并且得到它的一个接口指针。下面是 DLLGetClassObject()的一个原型：

```
STDAPI DllGetClassObject(REFCLSID rclsid,
                         REFIID riid,
                         LPVOID FAR*ppv)
```

这个函数有 3 个参数：对要实例化的类的 GUID 的一个引用、想要得到的接口的 GUID 以及用来存放返回接口指针的变量。第一个参数是很简单的，比如你想得到 CoSomeObject 的类厂，第一个参数就是 CoSomeObject 的 GUID 的值。第二个参数通常是 IID_IClassFactory。但是，如果类厂还实现了别的一些接口（除了 IClassFactory），客户也可以得到那些特殊的接口。第三个参数和 IUnknown::QueryInterface()中功能是一样的：它是一个指向接口指针的指针，一个用来存放请求的接口指针地方。

我们将很快看到在一个实际的 DLL 中，DLLGetClassObject()是如何工作的。

在进程外服务器中提供类厂

很自然，进程外服务器和进程内服务器有点区别。我们不能简单地通过导出函数来得到类厂，需要使用别的机制。实际上，进程外服务器在运行时在注册表里注册类厂。OLE 提供一个 API 函数来实现这个功能，这个函数叫 CoRegisterClassObject()：

```
HRESULT CoRegisterClassObject(REFCLSID clsid,
                              IUnknown *pUnk,
                              DWORD grfContext,
                              DWORD grfFlags,
                              LPDWORD pdwRegister);
```

进程外服务器通常在初始化的时候使用 CoRegisterClassObject 来注册它的 COM 类。这个函数要比导出 DLL 函数稍微复杂一点。第一个参数是要注册的类的 GUID。第二个参数是对象的 IUnknown 指针。第三个参数用来标识服务器的环境——可执行的服务器代码是一个 DLL、一个本地服务器还是一个远程服务器。环境的枚举是按位映射的，能够通过按位取或来得到所有的服务器环境。

```
enum tagCLSCTX
{ CLSCTX_INPROC_SERVER    = 1,
  CLSCTX_INPROC_HANDLER   = 2,
  CLSCTX_LOCAL_SERVER     = 4,
  CLSCTX_INPROC_SERVER16  = 8
}   CLSCTX;
```

第四个参数是一个叫 grfFlags 的 DWORD，用来表示客户是用何种方式跟服务器建立连接的。这个参数的值可以是 REGCLS 枚举中的一个，规定了一个类厂所能创建的对象的数量：

```
typedef enum tagREGCLS {
   REGCLS_SINGLEUSE = 0,
   REGCLS_MULTIPLEUSE = 1,
   REGCLS_MULTI_SEPARATE = 2
} REGCLS;
```

最后一个参数是一个指向 DWORD 的指针，它所指向的值在类厂注册的时候由 COM 来填写。这个值标识了类厂的注册信息，客户以后可以通过它然后调用 CoRevokeClassObject()来取消类厂的注册。

一旦一个类厂在 COM 中注册了，客户就是可以通过这个类厂来创建一个类的实例了。

卸载进程内服务器

我们已经看到在 COM 中 DLL 是如何被装载进来的（通常是通过调用 CoGetClassObject()或 CoCreateInstance()）。但是，DLL 又是怎么被卸载的呢？

因为 COM DLL 是有程序加载的，所以也有程序卸载。如果一个 DLL 没有被卸载，它将驻留在内存中，这是不允许的。由于 DLL 是被动的，所以它们必须得由外面的某个调用者来卸载。幸运的是，COM 中一个 API 函数 CoFreeUnusedLibraries()用来实现这个功能。COM 客户端必须经常调用这个函数，通常是在闲置时间（idle time）。作为垃圾清除机制，CoFreeUnusedLibraries()检查每一个通过 CoLoadLibrary()装载进来的 DLL，询问它们是否能够被卸载。但是它是如何询问 DLL 是否能够被卸载的呢？

OLE 进程内服务器提供一个 DLLCanUnloadNow()函数来实现这个功能。COM 在客户端调用 CoFreeUnusedLibraries()的时候就调用 DLL 的 DLLCanUnloadNow()函数。当 DLL 处于以下任何一种情况下的时候，DLLCanUnloadNow()都返回 S_FALSE：

1. 还存在对 DLL 类厂对象的引用。
2. 还存在对类厂创建的对象的引用。
3. 有任何明确的对 LockServer(TRUE)的调用，而没有通过 LockServer(FALSE)取消。

否则，DLLCanUnloadNow()返回 S_OK。如果 DLLCanUnloadNow()返回的值是 TRUE，COM 就卸载这个 DLL，否则，COM 在内存中保留这个 DLL。

卸载进程外服务器

和进程内服务器一样，进程外服务器也需要保存引用计数来表示服务器是否被锁定或者是否有客户创建了 COM 对象。进程外服务器所遵循的规则和进程内服务器一样，除了第一条以外：进程外服务器没有把类厂对象的引用计数作为判断是否要卸载服务器的依据。

然而，因为进程外服务器是主动活动着的，所以跟进程内服务器还是有区别的。进程外服务器不是等着 COM 来卸载它们，它们自己卸载自己。也就是说，在客户使用服务器以及它们提供的对象的时候，服务器必须检查自身的状态，看是否有客户在使用它的对象或接口。服务器必须在下面两种情况下检查状态：（1）当客户调用了一个类厂的 LockServer()函数；（2）客户释放接口指针。服务器在两种情况下可以卸载自己：

- 当客户减少服务器的引用计数（通过调用 IClassFactory::LockServer(FALSE)），而且

外部没有对象了。

- 当客户释放最后一个对象的最后一个接口指针，并且服务器的锁计数（lock count）是 0。

进程外服务器还有值得我们思考的地方。因为它们本身是一个完整的可执行程序，所以客户就可以在任何时候控制它们。比如，我们在 Windows 中打开 Word，并且想链接进 Excel 的电子表格（spreadsheet）。这个是很容易实现的：作为一个客户端，Word 让 Windows 自己去注册表中找到 Excel，激活它，然后用工作表（worksheet）的类厂创建一个工作表的对象实例。然后用 Excel 再打开一个工作表。现在，Excel 有两个打开的工作表：一个连接到 OLE 客户，另外一个是用户打开的。如果用户没有打开一个新的工作表，Excel 可以在 Word 关闭的同时关闭自己。然而在这个例子中，由于用户的介入，如果有人还在编辑文档的时候 Excel 就被强制关闭了，事情就变得不那么美妙了。

由于这个原因，进程外服务器需要维持一些信息来存储它的状态。也就是说，进程外服务器必须要有一个标识位来标识是否另外有用户控制程序（比如，打开另外一个文档）。这样，服务器就可以文明地关闭（比如警告用户）或是保持运行直到用户来关闭。

在注册表中注册类

在服务器中得到类只不过是第一步，客户还必须知道如何找到服务器。Windows 在系统注册表中保存类的信息。注册表是一个有层次的数据库，保留着所有的各种信息，包括机器上与 OLE 相关的和不相关的各种应用程序的信息。在注册表中，HKEY_CLASSES_ROOT 部分下面有一列 GUID 值（见图 11-3）。每一个实现了由某个 GUID 代表的类的服务器在这个 GUID 下面都有一个条目。比如，如果 CoSomeObject 同时在一个进程内服务器和一个进程外服务器中实现，则在注册表中这个 GUID 下面有两项：“InprocServer32”和“LocalServer32”。这些项的值是每一个服务器的真正的路径。

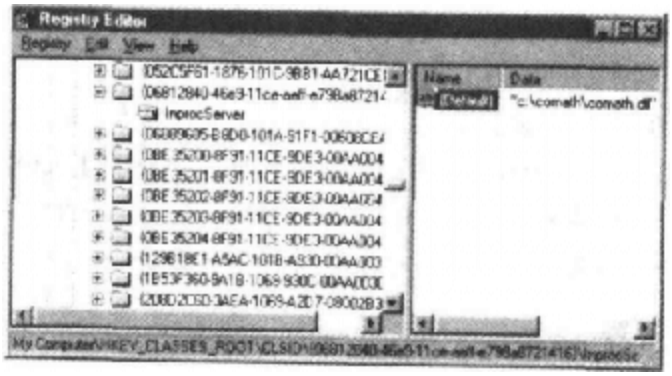


图 11-3 用 REGEDT32 浏览 CoMath 的 InprocServer32 注册表项

当一个 OLE 客户想实例化一个对象时，OLE 在注册表中寻找对应的 GUID。如果请求是实例化一个进程外服务器中的对象，则 OLE 寻找“LocalServer32”得到 EXE 文件的路径，从而执行服务器代码。如果请求是进程内的，服务器就使用和“InprocServer32”相关的路径。

创建 COM 类的实例

COM 客户使用一个叫 CoGetClassObject()的 OLE API 函数来得到一个 COM 类的类厂。当客户得到了一个类的 IClassFactory 接口指针后，就可以调用 IClassFactory::CreateInstance()

来创建一个 COM 对象，同时得到它的一个接口指针。

另外，客户也可以调用一个更方便的函数 CoCreateInstance()来实例化 COM 类。CoCreateInstance()执行以下几个步骤：

1. 在系统注册表中查询服务器的路径。
2. 装载 DLL 或是执行 EXE 服务器。
3. 得到类厂的指针。
4. 使用 IClassFactory 接口创建类的一个实例。
5. 返回请求接口的指针。

图 11-4 显示了 OLE 客户端和进程内服务器是如何工作的。

图 11-5 显示了 OLE 客户端和进程外服务器是如何工作的。

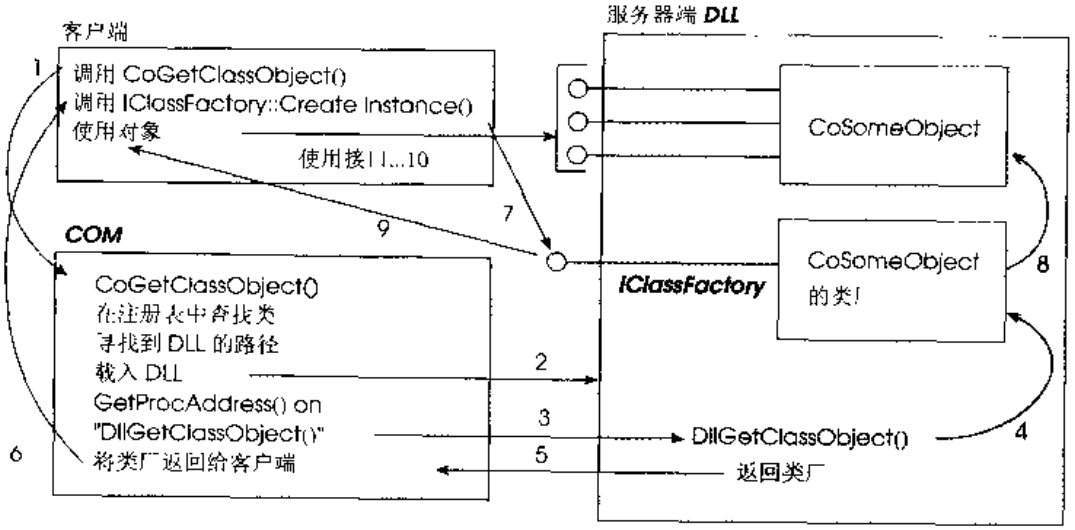


图 11-4 使用一个进程内服务器创建 COM 对象

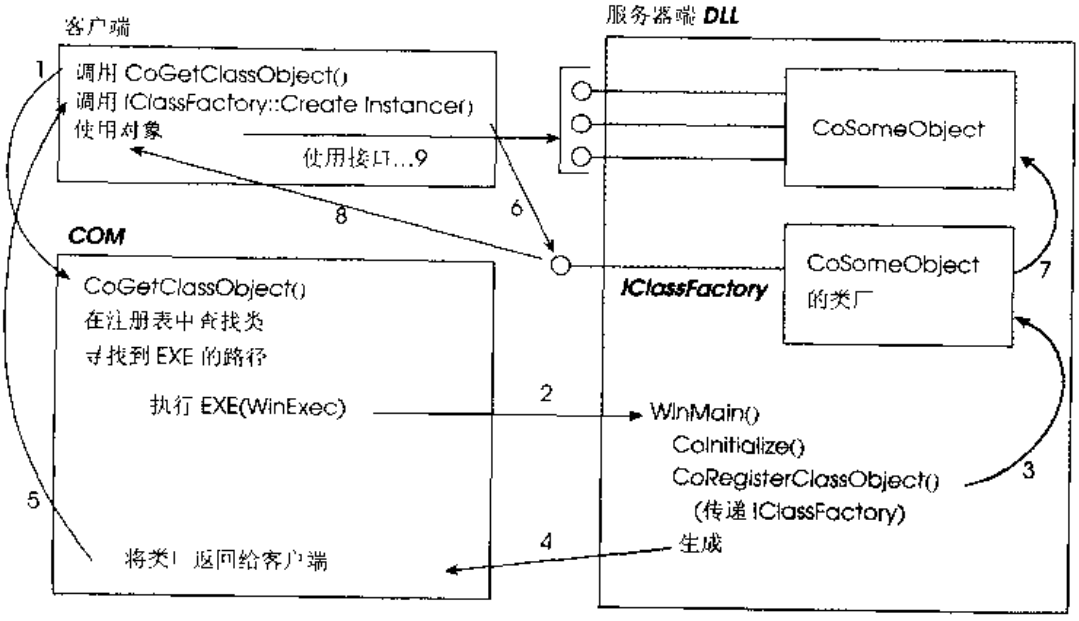


图 11-5 使用一个进程外服务器创建 COM 对象

进程内服务器和进程外服务器基本上使用相同的方式工作，其中最基本的区别是它们提供类厂接口的方式以及它们被卸载的方式。

拥有多个接口的 COM 类

COM 的工作方式可以说是相当漂亮的，让我们看个例子。为了理解有多个接口的真正的 COM 类，我们假定有一个叫 CoMath 的 COM 类。CoMath 提供 3 个接口：IUnknown、IPersist 以及 IMath。IUnknown 接口我们已经很熟悉了，它是所有 COM 类所必须要有的接口，用来处理对象的生存期和运行时动态接口发现。为了支持运行时接口发现，就必须使 COM 类支持另外的功能，也就是支持另外的接口。这个新的 COM 类（CoMath）除了支持 IUnknown 之外，还将支持 IPersist 和 IMath 接口。IPersist 接口微软已经定义，它里面只有一个函数：GetClassID()，它返回实现这个接口的 COM 类的 GUID。这个接口在这儿就是为了说明的目的。总的来说，这是一个很容易实现的接口，因为它只有一个比较容易实现的函数。这个类实现的第三个接口是 IMath。IMath 是一个客户接口，也就是说，它不是微软定义的标准接口（至少目前还不是）。IMath 是拥有两个函数的接口，这两个函数分别是 Add()和 Subtract()。Add()返回两个数的和，而 Subtract()返回两个数的差。

程序清单 11-9 显示了实际的 IMath 接口。图 11-6 是 CoMath 类的图形化表示。

程序清单 11-9 IMath 接口

```
DECLARE_INTERFACE_(IMath, IUnknown)
{
    // *** IUnknown methods ***//
    STDMETHOD(QueryInterface) (THIS_ REFIID riid,
                               LPVOID FAR* ppvObj) PURE;
    STDMETHOD_(ULONG,AddRef) (THIS)  PURE;
    STDMETHOD_(ULONG,Release) (THIS) PURE;

    // *** IMath methods ***//
    STDMETHOD(Add) (THIS_ INT, INT, LPLONG) PURE;
    STDMETHOD(Subtract) (THIS_ INT, INT, LPLONG) PURE;
};
```

在 C++中，实现具有多接口的 COM 类的最常用的两种方法是：（1）使用多继承；（2）使用嵌套类。

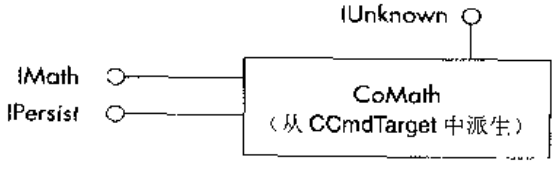


图 11-6 使用微软的插件符号表示的 CoMath

使用多继承实现 CoMath

实现类的最直接的方法是使用多继承机制。定义一个类，然后使这个类从你想使用的接口中派生即可。我们可以这样做是因为接口本身就是真正的 C++结构，而且接口里面只有纯虚函数。程序清单 11-10 和程序清单 11-11 显示了如何使用多继承来实现 CoMath。

程序清单 11-10 CoMath 的头文件（使用多继承）

```
#ifndef _COMATH_H
#define _COMATH_H
#include "IMath.h"
class CoMath: public IPersist, IMath {
private:
   DWORD m_dwRefCount;

public:
   CoMath();
   virtual ~CoMath();

   // IUnknown methods
   STDMETHODIMP_(DWORD) AddRef(void);
   STDMETHODIMP_(DWORD) Release(void);
   STDMETHODIMP QueryInterface(REFIID riid,
                               LPVOID FAR* ppv);

   // IPersist methods
   STDMETHODIMP GetClassID(LPCLSID pclsid);

   // IMath methods
   STDMETHODIMP Add(INT x, INT y, LPLONG lpResult);
   STDMETHODIMP Subtract(INT x, INT y, LPLONG lpResult);
};


#endif //_COMATH_H
```

程序清单 11-11 使用多继承实现 CoMath

```
CoMath::CoMath() {
   m_dwRefCount = 0;
}

CoMath::~CoMath() {
}


STDMETHODIMP_(DWORD) CoMath::AddRef(void) {
  return ++m_dwRefCount;
}

STDMETHODIMP_(DWORD) CoMath::Release(void) {
  DWORD dwResult = --m_dwRefCount;

  if(!dwresult)
    return dwResult;
}

STDMETHODIMP CoMath::QueryInterface(REFIID riid, LPVOID FAR *ppv) {
  *ppv = 0;
```

```
    // Test for each interface here...
    if (riid == IID_IUnknown)
      *ppv = LPPERSIST(this);
    else if (riid == IID_IMath)
      *ppv = LPMATH(this);
    else if (riid == IID_IPersist)
      *ppv = LPPERSIST(this);

    // If the Interface was available, AddRef on the way out
    if (*ppv) {
      LPUNKNOWN(*ppv)->AddRef();

      return NOERROR;
    }

    // Otherwise return the "No interface" error
    return ResultFromScode(E_NOINTERFACE);
}

STDMETHODIMP CoMath::GetClassID(LPCLSID pclsid) {
    *pclsid = CLSID_CoMath;
    return NOERROR;
}

STDMETHODIMP CoMath::Add(INT n1, INT n2, LPLONG lpResult) {
    *lpResult = n1 + n2;
    return NOERROR;
}

STDMETHODIMP CoMath::Subtract(INT n1, INT n2, LPLONG lpResult) {
    *lpResult = n1 - n2;
    return NOERROR;
}
```

使用多继承来创建 COM 类很直观，也很自然。注意 CoMath 是如何从几个不同的接口派生的。这几个接口分别是 IUnknown、IMath 和 IPersist。CoMath 继承了这 3 个接口的成员函数。我们知道接口的成员函数是纯虚函数，它们在 IMath 中定义，在 CoMath 中实现。

注意到在 CoMath 中有一个成员变量叫 dwRefCount。这个就是对象的引用计数，IUnknown 接口的成员函数使用它来控制对象的生命周期。

实现 IUnknown 是很直接也很简单的一件工作。其中 AddRef()增加对象的引用计数，而 Release()减少对象的引用计数。而 QueryInterface()在对象支持的接口中检查请求接口，然后返回恰当的接口指针。编译器会对返回的函数表进行处理，以保证返回的结果是正确的。

IPersist 是微软定义的另外一个接口。相对来说，这个接口比 IUnknown 更容易实现：这个接口的惟一成员函数返回 CoMath 的 GUID。我们只要将 CoMath 的 GUID 放到 LPCLSID 类型的参数中，然后返回 NOERROR 即可。

另外一个接口是 IMath，它有两个成员函数：一个对两个数进行加法，一个对两个数进行减法。Add()有两个整型参数和一个 LONG 类型的指针，名字叫 lpResult，用来存放返回结

果。Add()计算两个整数的和然后将返回结果存放在 lpResult 中。Subtract()与之类似。它也有两个整型参数和一个 LONG 类型的指针（也叫 lpResult），用来存放结果。Subtract()计算两个整数的差，然后将返回结果存放在 lpResult 中。

所有的这些操作都是在一个 COM 类中进行的。这个 COM 类从 IUnknown、IPersist 和 IMath 中继承。在这个例子中，每一个接口的成员函数都在 CoMath 中直接实现。

使用嵌套类实现 CoMath

COM 类支持多接口的第二种方法是通过类的合成。这种方法比起多继承来，可能没那么直观，但是，它也能很好地工作。

使用类的合成的基本思想是：只用一个类来表示 COM 类，这个类里面嵌套了一个或多个类，而接口的真正实现放在嵌套类里面。

程序清单 11-12 和 11-13 向我们显示了如何使用嵌套类实现 CoMath。

程序清单 11-12 CoMath 的头文件（使用嵌套类）

```
#ifndef _COMATHN_H
#define _COMATHN_H
#include "IMath.h"


class CoMath : public IUnknown {
private:
   DWORD m_dwRefCount;

public:
   CoMath();
   virtual ~CoMath();

   // IUnknown methods
   STDMETHODIMP_(DWORD) AddRef(void);
   STDMETHODIMP_(DWORD) Release(void);
   STDMETHODIMP QueryInterface(REFIID,
                               LPVOID FAR*);
   class PersistObj: public IPersist {
   public:
      CoMath* m_pParent;

      STDMETHODIMP_(DWORD) AddRef(void);
      STDMETHODIMP_(DWORD) Release(void);
      STDMETHODIMP QueryInterface(REFIID,
                                  LPVOID FAR*);
      STDMETHODIMP GetClassID(LPCLSID pclsid);
   } m_persistObj;

   class MathObj: public IMath {
   public:
      CoMath* m_pParent;

      STDMETHODIMP_(DWORD) AddRef(void);
      STDMETHODIMP_(DWORD) Release(void);
```

```
        STDMETHODIMP QueryInterface(REFIID,
                                    LPVOID FAR*);

        STDMETHODIMP Add(INT, INT, LPLONG);
        STDMETHODIMP Subtract(INT, INT, LPLONG);
    } m_mathObj;
};


#endif //_COMATHN_H
```

程序清单 11-13 使用嵌套类实现 CoMath

```
CoMath::CoMath() {
    m_dwRefCount = 0;
    m_persistObj.m_pParent = this;
    m_mathObj.m_pParent = this;
}

CoMath::~CoMath() {
}

DWORD CoMath::AddRef(void) {
  return ++m_dwRefCount;
}

DWORD CoMath::Release(void) {
  DWORD dwResult = --m_dwRefCount;
  if (!dwResult)
    delete this;
  return dwResult;
}
HRESULT CoMath::QueryInterface(REFIID iid, void** ppvObj) {
    if (iid == IID_IUnknown) {
        *ppvObj = (LPUNKNOWN)this;
        AddRef();
        return NOERROR;
    } else
    if (iid == IID_IPersist) {
        *ppvObj = (LPPERSIST)&m_persistObj;
        AddRef();
        return NOERROR;

    }
    else if (iid == IID_IMath) {
        *ppvObj = (IMath *)&m_mathObj;
        AddRef();
        return NOERROR;
    }
    return ResultFromScode(E_NOINTERFACE);
}


ULONG CoMath::PersistObj::AddRef()  {
    return m_pParent->AddRef();
```

```
}

ULONG CoMath::PersistObj::Release() {
    return m_pParent->Release();
}

HRESULT CoMath::PersistObj::QueryInterface(
    REFIID iid, void** ppvObj) {
    return m_pParent->QueryInterface(iid, ppvObj);
}


ULONG CoMath::MathObj::AddRef() {
    return m_pParent->AddRef();
}

ULONG CoMath::MathObj::Release() {
    return m_pParent->Release();
}

HRESULT CoMath::MathObj::QueryInterface(
    REFIID iid, void** ppvObj) {
    return m_pParent->QueryInterface(iid, ppvObj);
}

HRESULT CoMath::PersistObj::GetClassID(LPCLSID pclsid) {
  *pclsid = CLSID_CoMath;
  return NOERROR;
}

HRESULT CoMath::MathObj::Add(INT n1, INT n2, LPLONG lpResult) {
   *lpResult = n1 + n2;
   return NOERROR;
}

HRESULT CoMath::MathObj::Subtract(INT n1, INT n2, LPLONG lpResult) {

   *lpResult = n1 - n2;
   return NOERROR;
}
```

图 11-7 显示了 CoMath 和它的嵌套类的布局。

注意 IUnknown 是每一个接口的一部分，也就是说，每一个接口都必须包含 IUnknown 的 3 个函数：AddRef()、Release()以及 QueryInterface()。

让我们再来考虑一下 CoMath。我们需要 IUnknown 在整个类中都发挥作用。也就是说，我们要对整个类有一个引用计数，而且能够查询整个类以得到它的各种接口。在类的合成这种实现方式下，通过一个全局控制的类来实现这个功能，这个类拥有对象标识（CoMath），它实现了 IUnknown。而每一个嵌套类实现 IUnknown 是通过把 AddRef()、Release()和 QueryInterface()委托给 CoMath 实现。

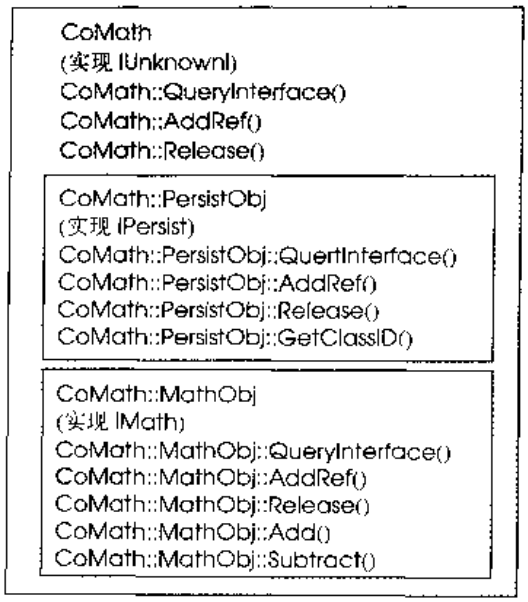


图 11-7 PersistObj 和 MathObj 嵌套在 CoMath 中

这个方法不是实现多接口的最简单的方法，但是它的实现很清晰而且工作得很好。而事实上，它跟 MFC 实现 COM 类的方式很类似。

CoMath 的类厂

由于 CoMath 是一个 COM 类，所以它当然需要一个类厂。类厂是另外一个 COM 对象，客户使用它来创建相对应的 COM 类的实例。只要一个类需要在服务器外面被创建，这个 COM 类就需要一个类厂对象。程序清单 11-14 和 11-15 是一个对应于 CoMath 的简单类厂。

程序清单 11-14 CoMath 类厂的头文件

```
class CoMathClassFactory : public IClassFactory {

public:
   CoMathClassFactory();

   STDMETHODIMP_(DWORD) AddRef();
   STDMETHODIMP_(DWORD) Release();
   STDMETHODIMP QueryInterface(REFIID riid, LPVOID FAR *ppv);

   STDMETHODIMP CreateInstance(IUnknown * pUnkOuter,
                               REFIID riid,
                               VOID ** ppvObject);
   STDMETHODIMP LockServer(BOOL fLock);

private:
   DWORD m_dwRefCount;
};
```

程序清单 11-15 CoMath 类厂的实现

```
// global reference count for the server
static DWORD dwServerCount = 0;
```

```
static CoMathClassFactory coMathCF;


CoMathClassFactory::CoMathClassFactory() {
   m_dwRefCount = 0;
}

CoMathClassFactory::~CoMathClassFactory() {
}

DWORD CoMathClassFactory::AddRef(void) {
  return ++m_dwRefCount;
}

DWORD CoMathClassFactory::Release(void) {
  return --m_dwRefCount;
}

HRESULT CoMathClassFactory::QueryInterface(REFIID riid, LPVOID FAR *ppv)
{
  *ppv = 0;

  // Test for each interface here...
  if (riid == IID_IClassFactory)
    *ppv = LPCLASSFACTORY(this);
  else if (riid == IID_IUnknown)
    *ppv = LPUNKNOWN(this);

  // If the Interface was available, AddRef on the way out
  if (*ppv) {
    LPUNKNOWN(*ppv)->AddRef();
    return NOERROR;
  }

  // Otherwise return the "No interface" error
  return ResultFromScode(E_NOINTERFACE);
}

HRESULT CoMathClassFactory::CreateInstance(IUnknown * pUnkOuter,
                                           REFIID riid,
                                           VOID ** ppvObject) {

   HRESULT hr = ResultFromScode(E_OUTOFMEMORY);

   CoMath *pCoMath = NULL;

   pCoMath = new CoMath;
   if(pCoMath) {

      hr = pCoMath->QueryInterface(riid, ppvObject);

      if(FAILED(hr))
         delete pCoMath;
   }
```

```
    return hr;
}

HRESULT CoMathClassFactory::LockServer(BOOL fLock) {
    if(fLock)
        dwServerCount++;
    else
        dwServerCount--;
};
```

这里需要注意的很重要的一点是：类厂代表了一个特定 COM 类的静态区域。对于每一个特定的 COM 对象，都有一个相对应的类厂实例。

让我们来看看类厂的 IUnknown 实现。由于服务器中保存了 CoMathClassFactory 的一个实例作为静态变量，所以 Release()并不删除这个指针。类厂存在于应用程序的整个生命周期中。而 QueryInterface()也仅仅检查请求的接口。只有当请求的接口是 IID_IUnknown 或 IID_IClassFactory 时，才返回相对应的接口指针。

现在让我们来看看类厂的 IClassFactory 中的函数。CoMathClassFactory::CreateInstance()只是为了创建 CoMath 对象而存在的。它的使命就是在有请求的时候分发一个新的 CoMath 对象。

CoMathClassFactory::LockServer()控制着服务器中的全局变量。当 LockServer(TRUE)被调用时，LockServer()将锁计数（lock count）加 1。当 LockServer(FALSE)被调用时，LockServer()将服务器的引用计数减 1。

前面的示例代码显示了如何使用多继承和嵌套类来实现 COM 类。MFC 采用嵌套类的方法。下面让我们来看看 MFC 是如何实现的。

MFC COM 类

在学习 OLE 的过程中，很大的一个障碍就是 OLE 在很多时候只是一个规范，它规定了需要强加到你的应用程序的各种特性。OLE 的很多特性（比如 OLE 文档、自动化和控件）都是通过 COM 来实现的。比如要实现 OLE 文档和自动化，我们只要写几个类，而这几个类实现了一些必要的接口就行。当然，“只要实现正确的接口”只是其中一个方面，真正去实现又是另外一方面的事。

举个例子，尽管要实现自动化只要创建一个支持 IDispatch 接口的类，但是并没有关于这个接口实现的细节。我们还必须实现所有的操作，比如动态函数发现、传输所有变量参数、保存局部信息等等。我们可以使用各种不同的语言和方法来实现 IDispatch。我们将在第 14 章讨论这些函数。

自动化在框架中工作得非常好。MFC 提供了一个 IDispatch 的很好的实现，而且很容易使用（特别是我们在用类向导的时候）。

另外一个例子是 OLE 文档服务器。一个 OLE 文档服务器必须实现一系列的接口以保证实现的是一个 OLE 文档服务器。OLE 文档服务器必须实现的接口有 IOleObject、

IPersistStorage、IDataObject、IOleInPlaceObject 和 IOleInPlaceActiveObject。查看 OLE 编程指南时，你就会发现这里面有些接口可能包含了超过 20 个的函数。你会为如此庞大的工作量而感到震惊。

也许，这又是类框架可以帮助我们做的事情。我们在编写一个 OLE 文档服务器的时候，需要花很大的精力来写一些同样的代码，而任何人在任何时间做这个工作，其做的事情几乎是一样的。也就是说，我们可以有很多的样本代码，这样就不必每次都做一些同样的工作了。

关键在于它的内部机制，MFC 只是实现了一些 COM 类，填写好类中的内容，所有的工作都是在 COM 类中实现的。MFC 的设计者只要选择一种方式实现多接口的 COM 类，然后在这些代码的基础上根据需求来真正地组装。

在 MFC 中，使用嵌套类的方法来实现 COM 类。在我们深入了解 MFC 实现 COM 的细节之前，先让我们了解一下为什么微软会使用嵌套类的方法来实现 COM 类。

为什么 MFC 使用嵌套类

在我们了解了实现 COM 类的两种主要的方法之后，也许你会有这样的感觉：使用嵌套类方法来实现 COM 类明显繁琐得多，而且随之而来的问题也更多。既然这样，那为什么微软会选择嵌套类来实现 COM 对象呢？而使用多继承来实现却又是那么的简洁明了。

实际上这是一个很普通的问题。原因如下所述：

1. 微软的立场是：人们在使用 MFC 的时候，并不一定要了解多继承机制。而作为一条通用的原则，MFC 中尽量避免使用多继承。

2. 在用多继承实现 COM 接口的时候，有时候确实也会产生一些很难解决的问题。

2a. 很难定义那些共享函数名的接口。如果一个接口中的函数和别的接口中的函数拥有同样的名字，则我们很难给这样的函数提供不同的实现。举个例子，这个就是为什么 IDataObject 中 advise 方法叫 DAdvise，而不是直接叫 Advise，因为 IOleObject 同样有一个 advise 方法。如果两个 advise 方法使用相同的名字，则在一个接口中同时实现 IOleObject 和 IDataObject 就会产生命名冲突了。

2b. 我们很难继承一个给定接口的实现。如果实现了 IUnkown 接口，比如在 CUnkown 中实现的，那么我们就要在所有的地方继承这个实现。比如类似于这种形式："COleClientItem：public CUnkown, IOleClientSite,IAdviseSink"。问题是这在 C++中是不允许的，你还是必须实现从 IUnkown 接口中继承下来（也许是间接的继承）的方法。这可以使用委托（delegation）很容易地解决。但是这样一来 MFC 在实现 COM 的时候就必须使用委托。多继承并不能解决这个重要的问题（也许，这个问题是 C++设计者当初欠考虑的一个问题）。因为不能立即解决这个问题，所以使用它的价值并不太大。

3. 多继承还带来了一个效率方面的问题。

3a. 微软的编译器（至少是 16 位的版本）产生了 vtable（也就是常说的虚表，每个接口一个。即使这个接口中的函数的实现在派生类中没有被重载，也会产生一个虚表），而这是完全没有必要的。

3b. 多继承中天生的效率损失（编译器产生的隐含代码）。在下面的例子中，你知道编译器产生了一个隐含的空指针检查吗？

```
*ppv = (IPersist*)this;
```

在编译后被扩展为：

```
if (this == NULL)
   *ppv = NULL;
else
```

```
*ppv = this+IPersist_offset;
```

这种隐含代码使代码变得没有必要的臃肿，但是这又是 C++定义中所必需的。它能够被避免，如果我们知道“this”永远都不会是 NULL 的话。也许这个看起来是个小问题，但是，在一个真正的程序中，这种额外的代码逐渐地积累也是很可怕的。

总而言之，微软认为使用多继承没有什么好处。不但如此，而且如上面所说的，还有很明显的坏处。由于很多的“蹩脚货”能够在 MFC 代码、MFC 宏或是 Wizard 中（在将来）隐藏，所以他们使用了嵌套类的方法来实现多接口。

使用 MFC 创建 CoMath

现在让我们来看看如何在 MFC 中实现 COM。为了让读者有更清晰的了解，我们将用 MFC 创建一个和前面我们所看到的一样的 CoMath 类。然后我们将使用这个 CoMath，把它放在一个简单的进程内服务器内。虽然 MFC 使用的方法和刚才描述的嵌套类方法基本一致，但是 MFC 在 CCmdTarget 类和接口映射表中隐藏了很多的实现细节。通过对这些细节的研究，我们就能够知道 MFC 是如何实现 COM 的，而且它们也没有最初看起来那么神秘了。

IUnknown 和 CCmdTarget

MFC 通过使用 CCmdTarget（加上一种称之为接口映射表的机制）来实现 COM 对象。在前面的嵌套类例子中，CoMath 是作为管理类实现了 IUnknown。让我们注意一下在这个例子里面，AddRef()、Release()和 QueryInterface()都是手工实现的。而在 MFC 中，CCmdTarget 作为 MFC 的与 OLE 相关的所有类的基类。它实现了 IUnknown 接口——一个所有的 COM 类都必须实现的接口。CCmdTarget 实现了两组 IUnknown 方法。一组实现 IUnknown 真正的功能（也就是真正的引用计数和接口查询），即 InternalAddRef()、InternalRelease()和 InternalQueryInterface()。另外一组是在 COM 聚合（aggregation）的时候使用的，即 ExternalAddRef()、ExternalRelease()和 InternalQueryInterface()。

你将会发现两组方法都在 CCmdTarget 中定义了：

```
class CCmdTarget: public CObject {

public:
   DWORD InternalQueryInterface(const void*, LPVOID* ppvObj);
   DWORD InternalAddRef();
   DWORD InternalRelease();

   DWORD ExternalQueryInterface(const void*, LPVOID* ppvObj);
   DWORD ExternalAddRef();
   DWORD ExternalRelease();
...
};
```

那么到底什么是 COM 的聚合呢？COM 的聚合是一种在二进制一级的基础上进行代码重用的方法。我们了解一下聚合是很有意义的，因为 MFC 内部已经提供了对它很好的支持。

COM 聚合

二进制重用

使用 COM 的一个根本目的就是将对象从源代码层转移到二进制层。COM 通过将对象解释成为客户可以理解一组接口的集合（这组接口可以通过各种语言来实现）来达到代码的重用。

COM 通过提供一种在二进制一级上共享代码的标准方法，从而达到软件的重用。但是 COM 并不支持通过继承来达到二进制重用。COM 使用的其中一种技术叫聚合（aggregation）。还有另外一种方法叫包容（containment），这是后话。

COM 聚合

COM 聚合是这样的一种方法：将一组 COM 对象组合在一起（每个 COM 对象都实现了一些接口），使得这些接口看起来好像是在一个对象内实现的。

从 COM 客户的观点来看，所有的 COM 对象都是一样的。它们都提供了一些接口。当我们要重用对象时，我们希望的是将以前写的这些对象结合在一起，使它们看起来就像是一个连续的整体（也就是平常的、标准的 COM 对象形式）。COM 聚合就是实现它的一种方法。

COM 聚合的实现

COM 聚合需要在 2 个或 2 个以上的 COM 对象中实现一种约定。第一个 COM 对象是客户看见的对象，称之为控制对象。另外还有一个或多个参与的对象。控制对象和参与对象都必须实现一些特殊的步骤，使聚合能运作起来。

假设你为一个重要的客户写了一个 COM 对象，名字叫 CoMath。它实现了 3 个接口：IUnknown（这个是必需的）、IaddSubstract 和 ImultiplyDivide。图 11-8 显示了 CoMath 对象。从 COM 客户的观点来看，只有一个对象实现了 3 个接口。

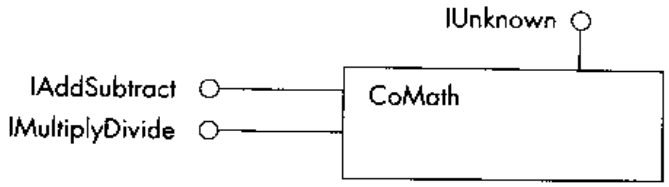


图 11-8 客户所看到的 COM 对象

现在假设 CoMath 对象很好地实现了 IAddSubstract，但是对于 IMultiplyDivide 的实现却不是那么完美，而你的客户要求你对它进行优化。不幸的是，你已经签下了下一份合同，也就是说你已经没有时间再去修改的你的旧代码去满足客户的要求了。幸运的是，你的一个朋友，也是一个软件开发者，他有一个 COM 对象，提供了很好的 IMultiplyDivide 实现（也许就是测试一下被 0 除永远都不会发生）。很自然的一个想法是你去使用他的算法，但是你的朋友不让你看他的源代码（什么样的朋友？）。在这个例子中，你将要做的任务就是得到你的朋友的实现了 IMultiplyDivide 接口的二进制版本的对象，然后把它集成到自己的对象中。这就是一个使用 COM 聚合的例子。

图 11-9 中显示了上面所说的两个 COM 对象：CoMath 和 CoBetterMath（就是你朋友实

现的那个)。

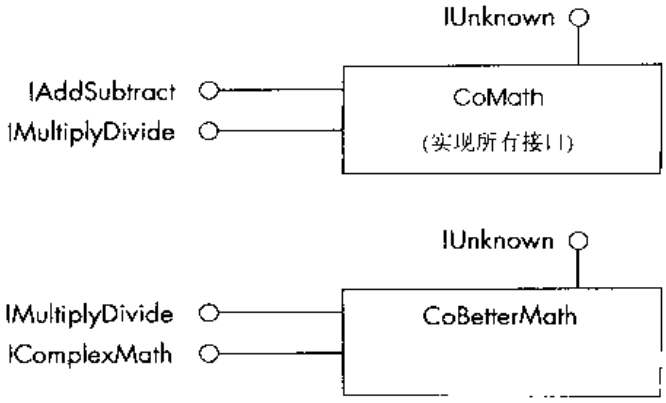


图 11-9 两个 COM 对象

CoBetterMath 有一个你想要使用的 IMultiplyDivide 的优化实现。然而，客户只想和 CoMath 打交道，也就是只调用 CoMath 中的函数。那么，我们该如何使用优化的 IMultiplyDivide 接口，而且在客户看来，就像是 CoMath 直接实现的？

聚合核心思想是：尽管在聚合实现中有很多个对象参与，但在客户的眼里，应该整个聚合就是一个整体。也就是说，IUnknown 接口（AddRef()、Release()和 QueryInterface()）将作用在整个聚合上。例如，调用 QueryInterface()来在你的朋友实现的 IMultiplyDivide 上查询 IAddSubtract 接口，就应该返回 IAddSubtract，而无论你朋友的 IMultiplyDivide 接口是否支持 IAddSubtract。

为了让聚合能很好地工作，两个对象必须在以下几个方面协同工作：

1. CoBetterMath（你朋友的对象）必须保存一个你的 IUnknown 接口指针的拷贝，叫做控制 IUnknown，这是实现 COM 聚合的关键。

2. 当创建 CoMath 实例的时候，也必须同时创建一个 CoBetterMath（实现了一个优化版本的 IMultiplyDivide）实例，这个也做为创建步骤的其中一部分。并且 CoMath 将它的 IUnknown 指针传给 CoBetterMath。

3. 当 CoBetterMath 得到 CoMath 的 IUnknown 指针时，将它保存下来。现在 CoBetterMath 就有了一个控制 IUnknown 指针。

4. 当客户向 CoMath 请求 IMultiplyDivide 接口，CoMath 只要将 CoBetterMath 的 IMultiplyDivide 接口指针返回即可。

现在看起来好像所有东西都运行得很好了。现在客户有了 IMultiplyDivide 的指针，可以用它来计算各种乘法和除法的问题。但是现在客户（拥有 IMultiplyDivide 指针）在需要一个 IAddSubtract 接口时遇到问题了：客户当然是调用 IMultiplyDivide 的 QueryInterface()来查询 IAddSubtract 接口，但是别忘了，现在的 IMultiplyDivide 接口是 CoBetterMath 的。那么，现在客户会得到什么呢？一个错误！让我们再看一下图 11-9，CoBetterMath 并没有实现 IaddSubtract 接口。所以，上面的安排还是有问题。

CoBetterMath 确实跟 IAddSubtract 有联系：CoBetterMath 有一份 CoMath 的 IUnknown 接口指针的拷贝。CoBetterMath 就是利用这个指针，将所有对接口 IUnknown 的调用，都转移到这个指针上，称之为委托。图 11-10 显示了这个关系。

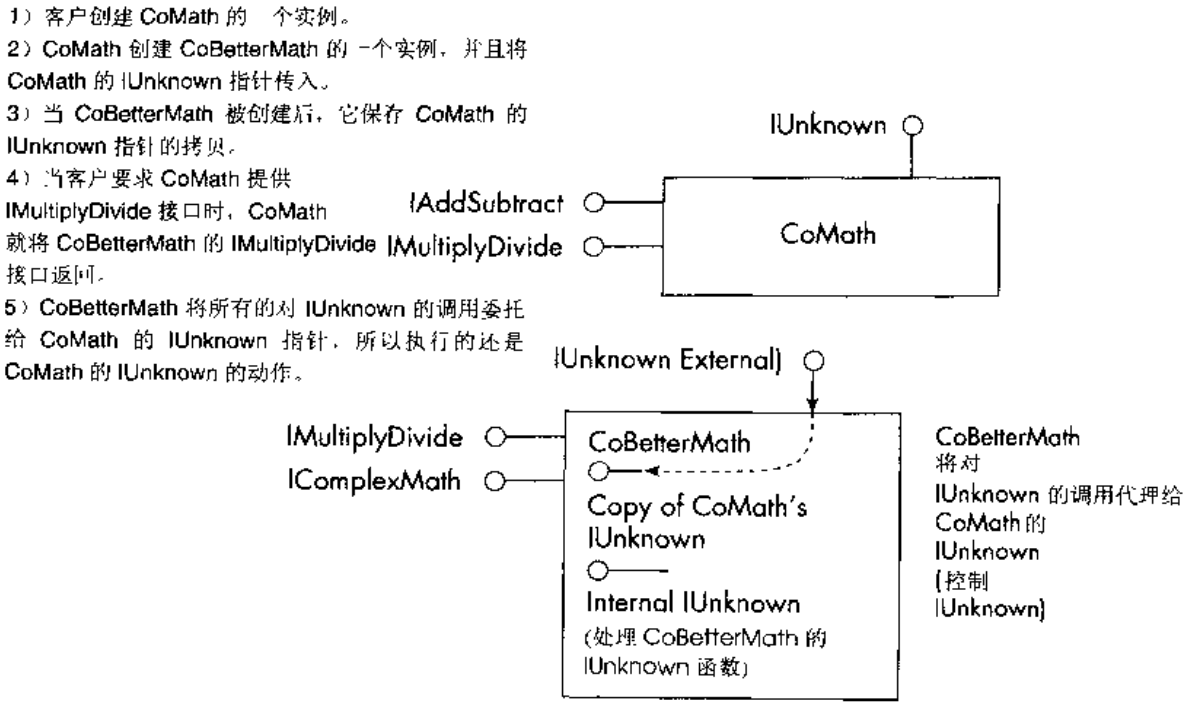


图 11-10 CoBetterMath 使用 COM 聚合

为了实现这个，CoBetterMath 实现了两套不同的 IUnknown 函数：一组用来将调用委托到控制 IUnknown 上去，还有一组实现 CoBetterMath 标准的 IUnknown 动作。这两个 IUnknown 指针通常分别被称为委托 IUnknown 和非委托 IUnknown。委托 IUnknown 将对 IUnknown 的调用传递给控制 IUnknown，而非委托 IUnknown 实现对象的标准的引用计数和 QueryInterface()。默认状态下，在聚合状态下的对象将所有的 IUnknown 调用都传递给控制 IUnknown。而是否处于聚合状态可以通过控制 IUnknown 的状态来判断（也就是判断它是否为 NULL）。

上面我们对 COM 聚合简略地描述了一下。我们可以看到，它是一个具有较高效率的二进制一级的代码重用，并且不需要在原来的基础上增加很多额外的代码。了解聚合是 COM 编程的基础，而且它将会变得越来越重要，特别是使用分布式（distributed）COM 的时候。所以任何实现 COM 的应用程序框架都必须对聚合提供支持。幸运的是，MFC 已经在它的 COM 实现中提供对了聚合的支持。

MFC 和 COM 聚合

当查看 MFC 的源代码时，你就会发现 IUnknown 的两个实现：一个内部版本和一个外部版本。这两个分别就是委托 IUnknown 和非委托 IUnknown。CCmdTarget 中包含了下面这些函数来实现 IUnknown：InternalQueryInterface()、InternalAddRef()、InternalRelease()、ExternalQueryInterface()、ExternalAddRef()以及 ExternalRelease()。

那么，在这些函数的不同版本中有什么区别呢？为了支持聚合，COM 对象中必须保存一个控制 IUnknown 的指针。如果这个指针是有效的（也就是非 NULL），则这个对象使用了聚合。在这种情况下，这个 COM 对象将对它的 IUnknown 接口的调用传递给控制 IUnknown。

这个就是 CCmdTarget 中 External...()这些 IUnknown 函数做的工作。CCmdTarget 中有一个数据成员叫 m_pOuterUnknown，当这个对象被作为聚合的一部分时，这个成员被初始化为控制 IUnknown。如果 m_pOuterUnknown 是有效的（非 NULL），则从 CCmdTarget 派生的类无条件地调用函数的 m_pOuterUnknown 版本。例如，如果客户调用一个 CCmdTarget 派生类的 Release()函数，并且这个类使用了聚合，则 CCmdTarget 的 ExternalRelease()将会调用 m_pOuterUnknown->Release()；否则，CCmdTarget 派生类仅仅对 CCmdTarget 的引用计数（m_dwRef）减 1。这是实现 COM 聚合的标准方法。

COM 聚合有两方面：（1）使得一个对象能够被聚合；（2）作为聚合的控制者。MFC 对两者都提供了支持。让我们先来看看怎么样使得一个 COM 类被聚合。

使 COM 类可被聚合

给一个 CCmdTarget 派生类增加对聚合的支持很容易。CCmdTarget 有一个名为 m_xInnerUnknown 的内部 IUnknown，它处理这个对象的标准的 IUnknown 操作。EnableAggregation()打开这个内部 Unknown 指针，用一个 IUnknown 的 vtable 对它初始化。

使用一个 COM 聚合

创建一个 COM 聚合可能稍微复杂一点，它需要下面几个步骤：

1. 声明一个成员变量（类型是 IUnknown*），用来保存聚合对象指针。

2. 在接口映射表中加入一个 INTERFACE_AGGREGATE 宏入口。它将通过名字指向上面那个成员变量（指向聚合对象的指针）。

3. 在 CCmdTarget::OnCreateAggregates()中初始化这个成员变量。

（注意，这个宏也许看起来有点奇怪，不过不用着急，我们将从下一节开始详细介绍 MFC 的接口映射表。）

举个例子，程序清单 11-16 显示了如何聚合你朋友的 CoBetterMath 对象，从而得到优化版本的 ImultiplyDivide。

程序清单 11-16 使用 MFC 的 COM 聚合

```
class CoMathAggr : public CCmdTarget {
public:
   CoMathAggr();

protected:
   IMultiplyDivide *m_lpMultiplyDivide;
   virtual BOOL OnCreateAggregates();

   DECLARE_INTERFACE_MAP()

   BEGIN_INTERFACE_PART(MathAggrObj, IAddSubtract)
     STDMETHOD(Add) (THIS_ INT, INT, LPLONG);
     STDMETHOD(Subtract) (THIS_ INT, INT, LPLONG);
   END_INTERFACE_PART(MathAggrObj)
};

CoMathAggr::CoMathAggr() {
   m_lpMultiplyDivide = NULL;
}
```

```
BOOL CoMathAggr::OnCreateAggregates() {
   CoCreateInstance(CLSID_CoBetterMath, GetConrollingUnknown(),
                    CLSCTX_ALL, IID_IMultiplyDivide,
                    (VOID FAR **)&m_lpMultiplyDivide);
   if (m_lpTypeInfo == NULL)
      return FALSE;
}

BEGIN_INTERFACE_MAP(CoMathAggr, CCmdTarget)
   INTERFACE_PART(CoMathAggr, IID_IMath, MathObj)
   INTERFACE_AGGREGATE(CoMathAggr, m_lpMultiplyDivide)
END_INTERFACE_MAP()
```

聚合的内部实现

INTERFACE_AGGREGATE 在接口映射表中插入一个 IUnknown 指针，这个指针代表了一个 COM 聚合。CCmdTarget 有一个函数叫 QueryAggregates()，在 CCmdTarget::InternalQueryInterface()中被调用。这使得被聚合的接口指针能够连同接口映射表中别的接口指针被一起检查。

在 COleObjectFactory 对 IClassFactory::CreateInstance()的实现中，在创建了控制对象后，调用 OnCreateAggregate()。这个地方也是创建聚合的另外成员的理想场所。

COM 聚合小结

OLE 首先是一项集成的技术，COM 的主要目标是使软件能够在二进制一级上重用(相对于源代码一级)。COM 聚合就是完成这个目标的一个途径。COM 聚合要求被聚合的对象保存控制对象的 IUnknown 指针，这样它就可以将 IUnknown 的调用委托给控制 IUnknown。

MFC 支持 COM，所以 CCmdTarget 就必须有两套的 IUnknown 实现：InternalQueryInterface()、InternalAddRef()、InternalRelease()；ExternalQueryInterface()、ExternalAddRef()、ExternalRelease()。

在 MFC 中实现聚合很直观。它需要使用两个成员函数（EnableAggregation()和 OnCreateAggregate()）和一个新的宏（INTERFACE_AGGREGATE）。也许一开始理解聚合有点困难，但是只要你掌握了它，它将是重用代码的很有效的一个方式。

内部 IUnknown 函数

为了实现 IUnknown 的引用计数函数（AddRef()和 Release()），CCmdTarget 保存了一个引用计数，叫 m_dwRef。你可以在 AFXOLE.INL 中找到 InternalAddRef()的实现，它只是对 m_dwRef 加 1。InternalRelease()稍微要复杂一点。

你可以在 OLEUNK.CPP 中找到 InternalRelease()的实现。InternalRelease()也是 IUnknown::Release()的一个标准实现。它对对象的引用计数减 1，并且如果已经到 0，(译者注)，则调用 CCmdTarget::OnFinalRelease()删除 this 指针。CCmdTarget::OnFinalRelease()不做别的工作，仅仅是执行“delete this;”，使 COM 对象在内存中将自己删除。

也许你现在会问：“那 QueryInterface()呢？”。CCmdTarget::InternalQueryInterface()通过使

用一个叫接口映射表的查询表来完成 QueryInterface()的功能。我们将在下面讨论它。

外部 IUnknown 函数

你可以在 OLEUNK.CPP 中找到 ExternalQueryInterface()、ExternalAddRef()和 ExternalRelease()的实现。如果它被实例化为聚合的一部分，则 CCmdTarget 保存一个指向控制 IUnknown 的指针（叫 m_pOuterUnknown）来支持 COM 聚合。所有的外部 IUnknown 函数都执行差不多相似的操作。首先检查一下 m_pOuterUnknown 是否有效，如果有效（也就是有控制 IUnknown 的接口指针），则 External...()函数将操作都委托给控制 unknown；否则 External...()函数将操作委托给 CCmdTarget::Internal...()函数。

AddRef()和 Release()都是使用 C++的普通功能进行 COM 编程，没有什么可多说的了，下面也就是一些细节的问题。在 MFC COM 类中更有意思的一个方面是它使用接口映射表对 IUnknown::QueryInterface()的实现。

通过嵌套类实现多接口

MFC 通过类的合成在一个类内部支持多接口。它在一个外部类里面嵌套了多个类，每个类都实现一个接口，通过这种方式使外部类支持多接口。前面我们提到的 CoMath 就支持 3 个接口：IUnknown、IMath 和 IPersist。

我们已经见过在 CCmdTarget 中如何实现 IUnknown 的对象生存期控制函数（AddRef()和 Release()），并且知道 CCmdTarget 以某种方式实现了 IUnknown::QueryInterface()。然而，我们到目前为止并不是很清楚 MFC 如何实现多接口和 QueryInterface()。在 CoMath 的例子中，IMath 和 IPersist 由两个嵌套类实现，这两个嵌套类分别是 MathObj 和 PersistObj。图 11-11 显示了在 CoMath 中包含的两个嵌套类。

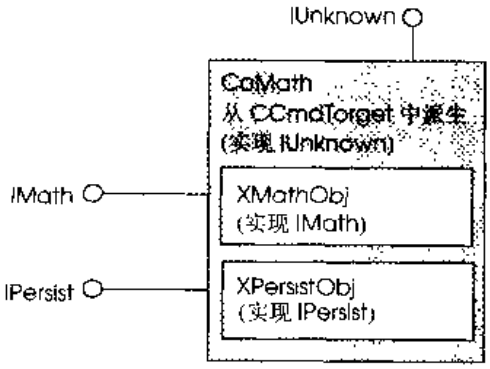


图 11-11 CoMath 嵌套两个独立的类

MFC 使用宏来实现嵌套类，并为 QueryInterface()提供查询表。下一个部分将介绍接口映射表以及产生它的宏。

MFC COM 和接口映射宏

所有的 COM 接口都必须支持 IUnknown。也就是说，我们必须可以在任何时候都能通过 QueryInterface()查询到一个对象实现了哪些别的接口。这就意味着在 CoMath 中每一个嵌套类都必须支持 IUnknown。由于对所有的 COM 类来说，实现 IUnknown 的代码看起来都几乎是一样的，所以 MFC 使用宏来产生这些冗余的代码。MFC 使用多个宏（和 CCmdTarget 一起工作）来实现它。

MFC 的 COM 宏把工作分为两个部分来完成。由于 MFC 对 COM 类的实现和前面的嵌

套类的例子很类似，所以宏的第一步工作就是创建嵌套类结构。接口映射宏的第二步工作就是创建 QureyInterface()要用的查询表。

声明嵌套类

MFC 定义了一对宏，BEGIN_INTERFACE_PART()和 END_INTERFACE_PART()，用它们来在 COM 类中声明嵌套类。这两个宏都可以在 AFXDISP.H 中找到。

下面是 BEGIN_INTERFACE_PART()宏的定义：

```
#define BEGIN_INTERFACE_PART(localClass, baseClass) \
   class X##localClass : public baseClass \
   { \
      public: \
         STDMETHOD_(ULONG, AddRef)(); \
         STDMETHOD_(ULONG, Release)(); \
         STDMETHOD(QueryInterface)(REFIID iid, LPVOID* ppvObj); \
```

这个宏有 2 个参数：实现接口的嵌套类和接口本身。例如，要在 CoMath 中实现 IMath，我们可以像下面这样使用宏：

```
BEGIN_INTERFACE_PART(MathObj, IMath)
```

使用这个宏会声明一个名字叫 XMathObj 的嵌套类，它从 IMath 派生。（MFC 在类名前面插入 X，以避免在命名空间中造成冲突。）BEGIN_INTERFACE_PART()也为 COM 类定义了 3 个 IUnknown 函数——AddRef()、Release()和 QueryInterface()。这里也是你定义接口的所有成员函数的地方。所以，这就和前面的嵌套类的例子几乎一样了。

定义嵌套类的第二个宏是 END_INTERFACE_PART()。这个宏完成从 BEGIN_INTERFACE_PART 开始的定义工作。下面是这个宏的定义：

```
#define END_INTERFACE_PART(localClass) \
        } m_x##localClass; \
        friend class X##localClass; \
```

这个宏所做的工作就是关闭嵌套类的声明，然后声明一个嵌套类类型的成员变量。注意一下，END_INTERFACE_PART()把这个成员声明为控制类的一个友元，这就使得嵌套类能够访问控制类的私有成员变量和成员函数。

现在我们还没有讨论一个问题，就是属于这个接口的其他函数的定义。例如，IPersist 另外还有一个成员函数（除了 IUnknown 中的成员）叫 GetClassID()。只有这些成员函数被声明了，嵌套类才会去实现这个接口。幸运的是，这个并不困难，我们所要做的只是插入每个函数原型即可。现在我们所做的全部工作就是为类声明一些成员函数。如果使用 MFC 来写 CoMath 类，下面就是在头文件中将会看到的 CoMath 声明：

```
// IMath members
BEGIN_INTERFACE_PART(MathObj, IMath)
  STDMETHOD(Add) (THIS_ INT, INT, LPLONG);
  STDMETHOD(Subtract) (THIS_ INT, INT, LPLONG);
END_INTERFACE_PART(MathObj)
```

```
// IPersist members
BEGIN_INTERFACE_PART(PersistObj, IPersist)
  STDMETHOD(GetClassID)(LPCLSID);
END_INTERFACE_PART(PersistObj)
```

以上就是使用宏来实现嵌套类的过程，下面让我们来看看用来实现 QueryInterface()的接口映射表是如何被创建的。

创建接口映射表

和别的映射表一样（例如消息映射表和分发映射表），接口映射表将一个标识符和一段可执行代码联系起来。例如，消息映射表得到一个窗口消息（比如 WM_COMMAND），将它映射到一个 C++类的成员函数（可以让这个类从 CCmdTarget 派生）。接口映射表也以类似的原理工作。但是，接口映射表不是将窗口消息映射到 C++类的成员函数，而是将一个接口标识符（一个 128 位的 GUID）映射到 COM 类中的一个嵌套类。这是 MFC COM 实现 QueryInterface()的方法。

接口映射宏和用来实现其他映射表（比如消息映射表）的宏十分类似。为了创建 MFC 查询表，MFC 通常在那个类的头文件提供一个 DECLARE...MAP()宏，并且在类的 C++文件中提供一对 BEGIN...MAP()/END...MAP()宏。（只要在省略的地方添上特定的映射，比如 BEGIN_MESSAGE_MAP()或 DECLARE_DISPATCH_MAP()）。为了与这种基于宏的技术保持一致，MFC 又为实现接口映射新定义了几个宏，分别是：DECLARE_INTERFACE_MAP()和 BEGIN_INTERFACE_MAP()/END_INTERFACE_MAP()。你可以在 AFXWIN.H 中找到 DECLARE_INTERFACE_MAP()的定义，在 AFXDISP.H 中找到 BEGIN_INTERFACE_MAP()和 END_INTERFACE_MAP()的定义。

DECLARE_INTERFACE_MAP()

下面是在 AFXWIN.H 中定义的 DECLARE_INTERFACE_MAP()宏：

```
#define DECLARE_INTERFACE_MAP() \
private: \
   static const AFX_INTERFACEMAP_ENTRY _interfaceEntries[]; \
protected: \
   static AFX_DATA const AFX_INTERFACEMAP interfaceMap; \
   virtual const AFX_INTERFACEMAP* GetInterfaceMap() const; \
```

这个宏定义了一种用于在 MFC COM 类中插入接口映射表的必要机制。首先，DECLARE_ INTERFACE_MAP()声明了一个 AFX_INTERFACEMAP_ENTRY 结构类型的不定长数组。在 AFXWIN.H 中，AFX_INTERFACEMAP_ENTRY 结构是这样定义的：

```
struct AFX_INTERFACEMAP_ENTRY
{
   const void* piid;
   size_t nOffset;   // offset of the interface vtable from m_unknown
};
```

其中第一个域是指向接口 ID 的指针。这个 ID 是用来标识一个特定接口的 GUID。第二个域是一个地址。我们将会在下面看到这个地址具体表示什么，以及它是如何被初始化的。

DECLARE_INTERFACE_MAP()还加入了一个类型为 AFX_INTERFACEMAP 的成员变量。这是一个结构，它包含了一个指向基类的接口映射表的指针和一个指向本类的接口映射表的指针。

```
struct AFX_INTERFACEMAP
{
   const AFX_INTERFACEMAP* pBaseMap;
   const AFX_INTERFACEMAP_ENTRY* pEntry; // map for this class
};
```

BEGIN_INTERFACE_MAP()/END_INTERFACE_MAP()

DECLARE_INTERFACE_MAP()声明了一个接口映射表结构和特定 COM 类的函数。当然，这个时候 MFC 已经在 CPP 文件中加入了一些代码，这就是 BEGIN_INTERFACE_MAP()和 END_INTERFACE_MAP()所做的工作。

BEGIN_INTERFACE_MAP()/END_INTERFACE_MAP()宏插在由 DECLARE_INTERFACE_MAP()定义的函数中。这两个宏在 AFXDISP.H 中定义。下面是 BEGIN_INTERFACE_MAP 的定义：

```
#define BEGIN_INTERFACE_MAP(theClass, theBase) \
   const AFX_INTERFACEMAP* theClass::GetInterfaceMap() const \
   { return &theClass::interfaceMap; } \
   const AFX_DATADEF AFX_INTERFACEMAP theClass::interfaceMap = \
   { &theBase::interfaceMap, &theClass::_interfaceEntries[0], }; \
   const AFX_DATADEF AFX_INTERFACEMAP_ENTRY theClass::_interfaceEntries[] = \
   { \
```

BEGIN_INTERFACE_MAP 有 2 个参数：（1）需要创建接口映射表的类；（2）这个类的直接祖先（也就是直接父类）。在源文件中插入 BEGIN_INTERFACE_MAP 这个宏，会产生 GetInterfaceMap()的实现代码，所有的代码就是返回这个类的接口映射表。这个宏还将类的 interfaceMap 成员变量初始化为指向表中第一个 AFX_INTERFACEMAP_ENTRY 的指针。接着，BEGIN_INTERFACE_MAP()使用 INTERFACE_PART()宏开始插入查询表表项。表项的类型为 AFX_MESSAGEMAP_ENTRY。

我们还记得，AFX_MESSAGEMAP_ENTRY 结构类型中保存一个接口 ID（一个 GUID）和一个 CCmdTarget 派生类的偏移量。接口 ID 是标识被实现的接口，地址是嵌套类在控制类中的偏移量。下面是在 AFXDISP.H 中定义的 INTERFACE_PART()：

```
#define INTERFACE_PART(theClass, iid, localClass) \
   { &iid, offsetof(theClass, m_x##localClass) }, \
```

MFC 在 COM 类中为实现的每一个接口都定义一个 INTERFACE_PART()入口。

最后，MFC 定义了 END_INTERFACE_MAP()来关闭接口映射表。这个宏也在 AFXDISP.H 中定义：

```
#define END_INTERFACE_MAP() \
      { NULL, (size_t)-1 } \
}; \
```

现在，我们已经了解了 MFC 的 COM 宏，下面让我们来看看如何利用这些宏来创建一个真正的 COM 类。

使用 MFC 的 CoMath 类

现在我们使用 MFC 来写 CoMath 类。我们知道，所有真正的 MFC COM 类都是从 CCmdTarget 继承下来的，因为 CCmdTarget 中实现了 IUnknown。程序清单 11-17 就是使用了 MFC 的接口映射表的 CoMath 的头文件：

程序清单 11-17 CoMath 的头文件（使用 MFC）

```
class CoMath : public CCmdTarget {

protected:

// IMath members
   BEGIN_INTERFACE_PART(MathObj, IMath)
     STDMETHOD(Add) (THIS_ INT, INT, LPLONG);
     STDMETHOD(Subtract) (THIS_ INT, INT, LPLONG);
   END_INTERFACE_PART(MathObj)
// IPersist members
   BEGIN_INTERFACE_PART(PersistObj, IPersist)
      STDMETHOD(GetClassID)(LPCLSID);
   END_INTERFACE_PART(PersistObj)

   DECLARE_INTERFACE_MAP()
   DECLARE_OLECREATE(CoMath)

public:
// constructor/destructor
   CoMath(void);
   virtual ~CoMath(void);

protected:
   DECLARE_DYNCREATE(CoMath)
};
```

现在我们先不用管 DECLARE_OLE_CREATE 宏（它给 CoMath 创建了一个类厂）。别的代码应该都是比较熟悉的了。

在 CoMath 的代码中，BEGIN_INTERFACE_PART()做两件事情：（1）声明实现 IMath 和 IPersist 的两个嵌套类；（2）为 CoMath 声明 IUnknown 接口中的函数。END_INTERFACE_PART() 结束嵌套类的声明，并且给嵌套类声明一个实例。在这两个宏中间的代码就是上面的接口函数，并且不包括 IUnknown 对每个接口的支持。而 DECLARE_INTERFACE_MAP()宏为类声明了一个接口映射表。

程序清单 11-18 给出了将宏展开后类的代码。

程序清单 11-18 预处理后的 CoMath 的头文件

```
class CoMath : public CCmdTarget {
protected:
   class XMathObj : public IMath {
   public:
```

```
        virtual ULONG __stdcall AddRef();
        virtual ULONG __stdcall Release();
        virtual HRESULT __stdcall QueryInterface(const IID & iid,
                                                 LPVOID* ppvObj);
        virtual HRESULT __stdcall Add ( INT, INT, LPLONG);
        virtual HRESULT __stdcall Subtract ( INT, INT, LPLONG);
      } m_xMathObj;
      friend class XMathObj;

      class XPersistObj : public IPersist {
      public:
        virtual ULONG __stdcall AddRef();
        virtual ULONG __stdcall Release();
        virtual HRESULT __stdcall QueryInterface(const IID & iid,
                                                 LPVOID* ppvObj);
        virtual HRESULT __stdcall GetClassID(LPCLSID);
      } m_xPersistObj;
      friend class XPersistObj;

      private:
        static const AFX_INTERFACEMAP_ENTRY _interfaceEntries[];
      protected:
        static  const AFX_INTERFACEMAP interfaceMap;
        virtual const AFX_INTERFACEMAP* GetInterfaceMap() const;
      protected:
        static  COleObjectFactory factory;
        static  const GUID __cdecl guid;

public:
    CoMath(void);
    virtual ~CoMath(void);

protected:
   public:
     static  CRuntimeClass classCoMath;
     virtual CRuntimeClass* GetRuntimeClass() const;
     static void __stdcall Construct(void* p);

};
```

CoMath 现在是完整的 COM 类，它支持 3 个接口：IUnknown、IPersist 和 IMath。下一步就是在 CPP 文件中实现 CoMath。这个需要两个步骤：(1)使用 BEGIN_INTERFACE_MAP() 和 END_INTERFACE_MAP()来创建 QueryInterface()的接口映射表；（2）实现每个接口。

接口映射表真正的实现是在 COMATH.CPP 文件中。除了 IUnknown，CoMath 还实现了两个接口。在 BEGIN_INTERFACE_MAP() 和 END_INTERFACE_MAP() 之间调用宏 INTERFACE_PART()——每个需要实现的接口都调用一次。

BEGIN_INTERFACE_MAP() 实现了 CoMath 的 GetInterfaceMap() 函数。每个 INTERFACE_PART()宏给 CoMath 的接口映射表中增加一个 AFX_INTERFACEMAP_ENTRY（包括接口 ID 和类偏移）。CoMath 有两个嵌套类，所以它包含两个接口映射宏——一个类对应一个。当框架需要访问类的接口时，通过 GetInterfaceMap()得到指向接口映射表的指针，

然后在元素类型为 AFX_INTERFACEMAP_ENTRY 的数组中查找相应的接口 ID。下面是预处理后的 BEGIN_INTERFACE_MAP()和 END_INTERFACE_MAP()产生的结果：

```
const AFX_INTERFACEMAP* CoMath::GetInterfaceMap() const {
   return &CoMath::interfaceMap;
}

const  AFX_INTERFACEMAP CoMath::interfaceMap = {
   &CCmdTarget::interfaceMap, &CoMath::_interfaceEntries[0],
};

const  AFX_INTERFACEMAP_ENTRY CoMath::_interfaceEntries[] = {
  { &IID_IMath, (size_t)&(((CoMath *)0)->m_xMathObj) },
  { &IID_IPersist, (size_t)&(((CoMath *)0)->m_xPersistObj) },
  { 0, (size_t)-1 }
};
```

在这个意义上，CoMath 看起来如图 11-12 所示。

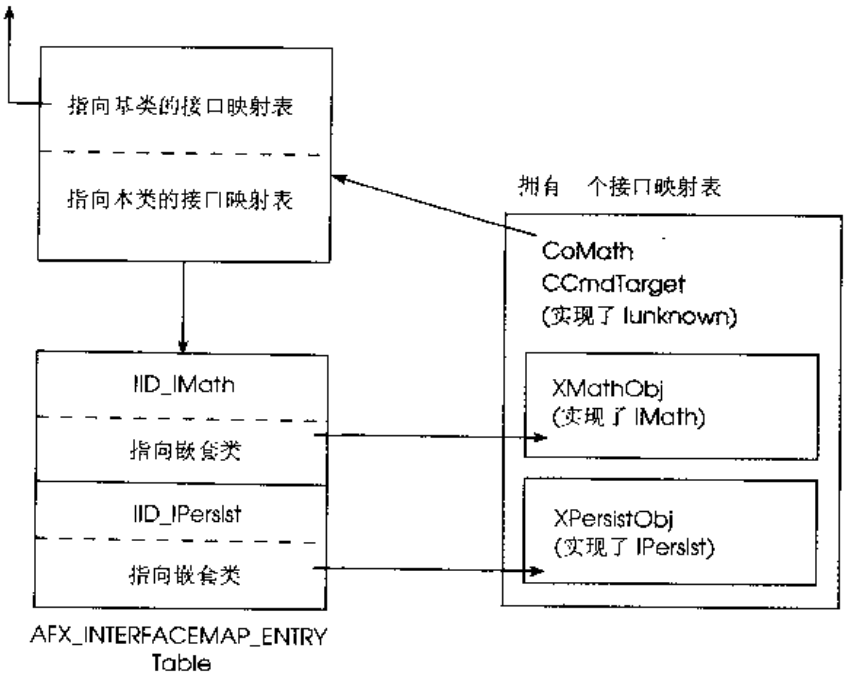


图 11-12 CoMath 的接口映射表

MFC COM 类和继承

CoMath 是一个从 CCmdTarget 派生的 MFC 类。它包含有两个嵌套类：XMathObj 和 XPersistObj，每一个嵌套类实现一个 COM 接口。CoMath 中还包含一个接口映射表，它将一个接口 ID 和实现对应接口的嵌套类指针关联起来。

图 11-13 展示了这个层次结构。假设有一个 MFC 类叫 CoMathEx，它实现了另外一个接口叫 IMath2。也许 IMath2 实现了两个其他的函数叫 Multiply()和 Divide()。你可以使用 MFC 使 CoMathEx 从 CoMath 派生，使 CoMathEx 包含 CoMath 的 Add()和 Subtract()函数的功能。

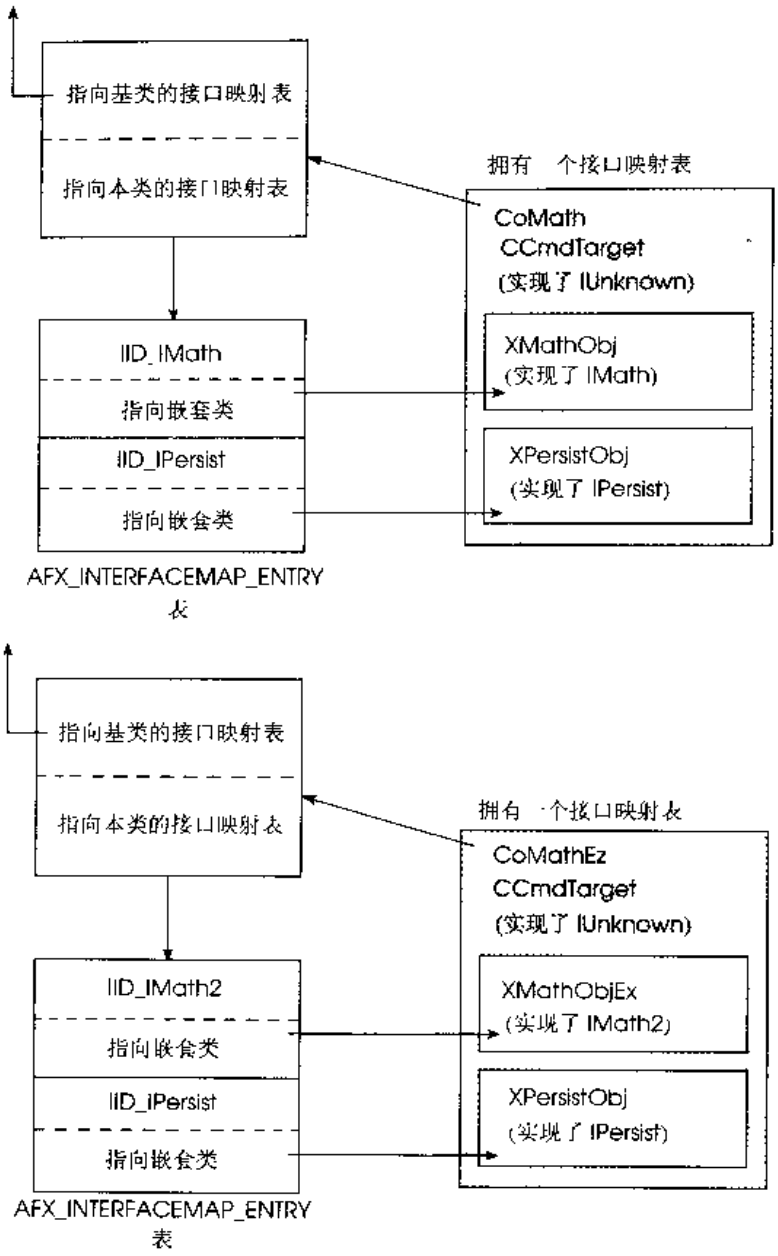


图 11-13 CoMathEx 继承 CoMath 的接口映射表

注意一下接口映射表是如何像消息映射表一样工作的。如果一个接口在一个特定的类中没有实现，MFC 的接口映射表机制会保证遍历所有的派生类，直到找到这个接口或是确定没有提供这种接口。

重新审视 InternalQueryInterface()

现在 CoMath 已经有了一个接口映射表，那么，接口映射表有什么作用呢？CCmdTarget::InternalQueryInterface() 使用接口映射表来实现它的功能。InternalQueryInterface() 调用 CCmdTarget::GetInterface()得到请求接口。

GetInterface()以如下的方式工作：COM 类会安装一个接口 hook。CCmdTarget 有一个虚

函数来取得这个接口 hook。如果 GetInterfaceHook()返回 TRUE，则 GetInterface()完成。否则，QueryInterface()还有一些工作要做。为了实现 QueryInterface()，GetInterface()只要遍历接口映射表，寻找请求接口的 ID。若找到接口，则返回在 AFX_MESSAGEMAP_ENTRY()中找到的接口指针（对应的嵌套类的地址），如程序清单 11-19 所示。

程序清单 11-19 CCmdTarget::GetInterface()的伪代码

```
LPUNKNOWN CCmdTarget::GetInterface(const void* iid) {
   Call CCmdTarget::GetInterfaceHook()
     to allow interface hook first chance
     If GetInterfaceHook returns a value,
     send that value back to the caller.

   Use CCmdTarget::GetInterfaceMap() to
    retrieve this object's interface map.

   Start walking the interface map towards
     the base class.

   do
      while there are still interface map entries {
         Try to match the interface ID in the map
         If match found,
            return the interface pointer
         else
            Get the next interface map entry
      }
      Get the previous next interface map (towards CCmdTarget)
   until there are not more interface maps
}
```

另外，在 MFC 的类作用域外，可以利用 GetInterface()来使用 COM 接口。例如，COleServerItem 实现了一个叫 IDataObject 的接口。如果想使用 COleServerItem 的 IDataObject 接口，可以调用 GetInterface()来得到它。它使用一个存放 GUID 的地址作为参数，这个 GUID 用来标识一个接口（也就是 IID）。如果 MFC 实现了这个接口，则 GetInterface()返回 IUnknown 指针。当框架调用这个函数时，通常将 IUnknown 接口转换为确切的接口（也就是需要使用的接口）。这样做是安全的，因为 GetInterface()只有当请求接口被找到时才返回。下面是 CCmdTarget::GetInterface()的原型：

```
LPUNKNOWN GetInterface(const void*);
```

尽管 GetInterface()和 GetInterfaceHook()返回指向接口的指针，但是它们并没有使用 AddRef()作用于对它们的返回值。这在别的层次上做，比如在 InternalQueryInterface()中。

完成服务器的设计

定义了接口映射表后，为了完成服务器的设计，还得做下面一些工作：（1）真正实现接口（包括嵌套类的 IUnknown 函数）；（2）给 CoMath 类分配一个 GUID；（3）从 CWinApp 派生一个类，并且声明它的一个实例；（4）给服务器提供一个类厂；（5）实现和导出

DllCanUnloadNow()和 DllGetClassObject()两个函数。

实现接口

通过利用接口映射表，CoMath 使用嵌套类来实现 IPersist 和 IMath。尽管在每个接口中都声明了 IUnknown 函数，但是还必须为每一个子类实现 IUnknown。也就是说我们必须为 CoMath::XPersistObj 和 CoMath::XMathObj 分别实现 AddRef()、Release()和 QueryInterface()。

CoMath 会提供两个接口（除了 IUnknown）：IPersist 和 IMath，实现这两个接口的每个类也需要实现 IUnknown。然而，我们必须将 CoMath 看成一个整体。也就是说，在整个类中必须只提供一个 IUnknown 的实现。通过将每个嵌套类的 IUnknown 的实现委托给 CCmdTarget 的 IUnknown 就可以实现这一点。

我们在前面已经见过 CCmdTarget 的 IUnknown 函数：ExternalAddRef()、ExternalRelease()、ExternalQueryInterface()、InternalAddRef()、InternalRelease()和 InternalQueryInterface()。为了取得 COM 对象的指针（在这个例子中是 CoMath），必须取得指向 CoMath 的指针。使用 METHOD_PROLOGUE()宏，它需要两个参数：（1）外部类的名字；（2）实现特定接口的内部类的名字。METHOD_PROLOGUE()声明一个指向外部类的指针（CoMath），命名为 pThis，并且将它赋值为外部控制类的地址。METHOD_PROLOGUE()通过在 CoMath 中减去内部类的偏移量来实现这点。接着，这个指针（pThis）就可以用来访问控制类的成员了（包括 ExternalAddRef()、ExternalRelease()和 ExternalQueryInterface()）。

如果需要得到实现所需接口的 MFC COM 类的指针，我们就可以使用 METHOD_PROLOGUE()宏。比如，可以使用 METHOD_PROLOGUE()来实现每个在 CoMath 中通过嵌套类表示的接口。程序清单 11-20 展示了 CoMath 如何实现 IPersist 接口中的 IUnknown 函数。

程序清单 11-20　实现 IPersist 的 AddRef()、Release()和 QueryInterface()函数

```
STDMETHODIMP_(ULONG)
CoMath::XPersistObj::AddRef(void) {
   METHOD_PROLOGUE(CoMath, PersistObj)
   return pThis->ExternalAddRef();
}

STDMETHODIMP_(ULONG)
CoMath::XPersistObj::Release(void) {
   METHOD_PROLOGUE(CoMath, PersistObj)
   return pThis->ExternalRelease();
}

STDMETHODIMP
CoMath::XPersistObj::QueryInterface(REFIID riid, LPVOID FAR*ppv) {
   METHOD_PROLOGUE(CoMath, PersistObj)
   return pThis->ExternalQueryInterface(&riid, ppv);
}
```

这些函数在 XMathObj 中也必须实现。不过目前 MFC 还没有支持这个的接口向导，将来微软应该会提供一个。在 MFC 2.5 中，微软将这些工作全部都放到宏中来完成。将代码都

放入宏中的一个坏处是使调试变困难了。如果 QueryInterface()是通过一行代码产生，并且这行代码还顺序产生一些别的不相关的函数（比如 AddRef()/Release()）的话，那么要想在其中设断点就非常困难。最后，微软为了使调试变简单并且便于理解，去掉了一些更复杂的宏。现在使用 MFC 来写 COM 类还是比较困难的一件工作。然而，等到将来向导设计出来后，创建 COM 类会变得容易许多，而且调试也会变简单。

下面是在 AFXDISP.H 中定义的 METHOD_PROLOGUE()宏：

```
#define METHOD_PROLOGUE(theClass, localClass) \
theClass* pThis = \
      ((theClass*)((BYTE*)this - offsetof(theClass, m_x##localClass))); \
```

METHOD_PROLOGUE 有两个参数：一个 COM 类（从 CCmdTarget 派生）和一个嵌套类。METHOD_PROLOGUE 能够得到嵌套类在 COM 类中的偏移量。比如 METHOD_PROLOGUE 能够产生一段代码，通过 m_xPersistObj 取得 CoMath 的指针。

MFC 对类厂的支持

现在 CoMath 并没有真正完成。在我们对它进一步完善前，先让我们来看看 COM 的一个很重要的特性，那就是类厂。

OLE 类厂本质上来说是实现了 IClassFactory 接口的 COM 类。严格上来说，一个类厂所要做的全部工作就是实现 IClassFactory。但是在 MFC 中，类厂在实现 IClassFactory 的同时，还包括几个现成的注册函数。由于在 MFC 类厂的实现中包含了必要的 GUID 信息，所以在运行的时候注册类厂是很容易实现的。我们可以在 COleObjectFactory 中找到 MFC 对基本的类厂的支持。

COleObjectFactory

COleObjectFactory 是一个实现了 IClassFactory 接口的严格意义上的 COM 类。COleObjectFactory 使用 MFC 的方式（也就是使用接口映射表）来实现这个接口。如程序清单 11-21 所示，在 AFXDISP.H 中定义的 COleObjectFactory 中，可以看到 IClassFactory 的接口映射表。

程序清单 11-21 COleObjectFactory 的定义

```
class COleObjectFactory : public CCmdTarget
{
   DECLARE_DYNAMIC(COleObjectFactory)

// Construction
public:
   COleObjectFactory(REFCLSID clsid, CRuntimeClass* pRuntimeClass,
      BOOL bMultiInstance, LPCTSTR lpszProgID);

// Attributes
   BOOL IsRegistered() const;
```

```
    REFCLSID GetClassID() const;

// Operations
    BOOL Register();
    void Revoke();
    void UpdateRegistry(LPCTSTR lpszProgID = NULL);
       // default uses m_lpszProgID if not NULL
    BOOL IsLicenseValid();

    static BOOL PASCAL RegisterAll();
    static void PASCAL RevokeAll();
    static BOOL PASCAL UpdateRegistryAll(BOOL bRegister = TRUE);

// Overridables
protected:
    virtual CCmdTarget* OnCreateObject();
    virtual BOOL UpdateRegistry(BOOL bRegister);
    virtual BOOL VerifyUserLicense();
    virtual BOOL GetLicenseKey(DWORD dwReserved, BSTR* pbstrKey);
    virtual BOOL VerifyLicenseKey(BSTR bstrKey);

// Implementation
public:
    virtual ~COleObjectFactory();
#ifdef _DEBUG
    void AssertValid() const;
    void Dump(CDumpContext& dc) const;
#endif

public:
    COleObjectFactory* m_pNextFactory;  // list of factories maintained

protected:
    DWORD m_dwRegister;              // registry identifier
    CLSID m_clsid;                   // registered class ID
    CRuntimeClass* m_pRuntimeClass; // runtime class of CCmdTarget
                                     // derivative
    BOOL m_bMultiInstance;           // multiple instance?
    LPCTSTR m_lpszProgID;            // human readable class ID
    BYTE m_bLicenseChecked;
    BYTE m_bLicenseValid;
    BYTE m_bRegistered;           // is currently registered w/ system
    BYTE m_bReserved;             // reserved for future use
// Interface Maps
public:
    BEGIN_INTERFACE_PART(ClassFactory, IClassFactory2)
       INIT_INTERFACE_PART(COleObjectFactory, ClassFactory)
       STDMETHOD(CreateInstance)(LPUNKNOWN, REFIID, LPVOID*);
       STDMETHOD(LockServer)(BOOL);
       STDMETHOD(GetLicInfo)(LPLICINFO);
       STDMETHOD(RequestLicKey)(DWORD, BSTR*);
       STDMETHOD(CreateInstanceLic)(LPUNKNOWN, LPUNKNOWN, REFIID, BSTR,
          LPVOID*);
    END_INTERFACE_PART(ClassFactory)
```

```
    DECLARE_INTERFACE_MAP()

    friend SCODE AFXAPI AfxDllGetClassObject(REFCLSID, REFIID, LPVOID*);
    friend SCODE STDAPICALLTYPE DllGetClassObject(REFCLSID, REFIID, LPVOID*);
};
```

版本注释

从 MFC 4.0 版本开始，微软使用 COleObjectFactory 实现 IClassFactory2。而在此之前的版本中，COleObjectFactory 实现 IClassFactory，而 COleObjectFactoryEx 实现 IClassFactory2。IClassFactory2 包含 IClassFactory 中的所有函数，并且加上了对组件许可（component licensing）的支持。

COleObjectFactory 的构造函数有 4 个参数：（1）COM 类对应的 GUID；（2）与这个 COM 类关联的 CRuntimeClass 指针；（3）一个标志，用来表示服务器的一个实例能否生成多个 COM 类；（4）一个符合人们日常习惯的服务器名字。当一个 COleObjectFactory 被创建时，它在成员变量中保存类厂的 GUID、COM 类的 CRuntimeClass、多实例标志和程序 ID。它也会将 dwRegister 这个变量初始化为 0，表示类厂还没有被注册。

因为 COM 服务器可能包含多个类，所以它有时需要支持多个类厂，这是由于服务器中的每个 COM 类都分别对应一个类厂。MFC 保存一个类厂的链表，这个链表的第一个节点保存在 MFC 的状态信息中。每一个类厂被创建时，都会在类厂链表的最后加入自己的信息。

类厂和服务器的注册

COleObjectFactory 包含了几个函数，这些函数能够使用 OLE 注册类厂（对进程外服务器），并且能够更新系统注册表。下面是一个 COleObjectFactory 实例必须支持的 3 个函数：

```
BOOL Register();
void Revoke();
void UpdateRegistry(LPCTSTR lpszProgID = NULL);
```

COleObjectFactory::Register()会在注册表中注册类厂，这样客户就可以使用它提供的服务。这个函数其实也不是很复杂：它不过是对 CoRegisterClassObject()的一个包装。它会调用 CoRegisterClassObject()来注册类厂，并且给 m_dwRegister 赋个标记（token），以后 COleObjectFactory 可以使用这个标记来取消类厂的注册。

COleObjectFactory 有一个和 Register()对应的函数，叫 Revoke()。Revoke()可以取消一个类厂的注册。和 Register()类似，Revoke()也对 OLE API 函数 CoRevokeClassObject()进行包装，它使用 COleObjectFactory::m_dwRegister 中保存的标记来取消类厂的注册。

COleObjectFactory::UpdateRegistry()用来更新系统注册表。UpdateRegistry()只有一个参数：类注册的时候使用的 prog ID。COleObjectFactory::UpdateRegistry()调用了 AFX 辅助函数 AfxOleRegisterServerClass() 和 AfxOleRegisterHelper() 来实现真正的注册。AfxOleRegisterServerClass()和 AfxOleRegisterHelper()包装了几个 OLE 注册的 API 函数。

注册表是所有 COM 类的信息交换场所。注册表中包含了很多有关 COM 的不同的条目，所以，如果要为一个复杂的程序注册所有的条目，则会变得十分繁琐。

下面是注册表中跟 COM 相关的一些条目：

- CLSID——表示 COM 类的 GUID。
- InprocServer32——进程内服务器的路径。

- LocalServer32——本地服务器的路径。
- ProgID——用来表示 COM 类的字符串（便于使用者记忆）。
- Verb——OLE 文档服务器的操作。

COleObjectFactory 的 UpdateRegistry()函数会在系统注册表中加入相应的入口。

COleObjectFactory 的下面 3 个成员是 3 个静态成员函数。每个应用程序的 COleObjectFactory 类实例都包含它们：

```
static BOOL PASCAL RegisterAll();
static void PASCAL RevokeAll();
static void PASCAL UpdateRegistryAll();
```

这几个函数跟普通的非静态函数在使用上没有区别，除了它们可以作用于一个应用程序当中的所有类厂。每个函数都去遍历类厂链表，分别用于为每个类厂注册、注销或者更新注册表。

开发技巧：注册另外的信息

如果曾经试图往注册表里增加键值，那么就会体会到这是一件很麻烦的事情。例如，如果想给 OLE 文档服务器增加一个客户的操作，可以通过注册表编辑器手工加入，但是这是很危险的操作。我曾经见过有人不小心选错了菜单项，破坏了 Windows NT 的安装选项。（我们自己也曾破坏过）。我们也可以使用注册表的 API 函数来注册一个客户操作。通过程序来注册需要以下几个步骤：

1. 使用 RegOpenKey()打开在 HKEY_CLASSES_ROOT 下的 CLSID 键。
2. 使用 RegSetValue()设定值。
3. 使用 RegCloseKey()关闭键。

使用这种方法，必须检查错误代码和处理一些特殊情况。比如，对于某一个特殊的键，如果已经有值了该怎么办？

MFC 为应用程序在注册表中创建入口。我们可以使用 AfxOleRegisterHelper()来更新注册表，下面是 AfxOleRegisterHelper()的原型：

```
BOOL AFXAPI AfxOleRegisterHelper(LPCTSTR const* rglpszRegister,
        LPCTSTR const* rglpszSymbols, int nSymbols, BOOL bReplace,
        HKEY hKeyRoot = ((HKEY)0x80000000)); // HKEY_CLASSES_ROOT
```

AfxOleRegisterHelper()有 5 个参数：

1. rglpszRegister 是一个字符串数组，用来保存需要加入到注册表中的键和它们的值。
2. rglpszSymbols 也是一个字符串数组，用来保存替换的值。例如，想替换的 Class ID。
3. nSymbols 用来表示 lpszSymbols 数组中的数组元素。
4. bReplace 表示是否替换注册表中的条目。
5. bKeyRoot 表示所有这些条目的根键。由于是在注册表中注册 OLE 相关的组件，所以默认根键是 HKEY_CLASSES_ROOT。

那么，如何利用这些信息呢？如果想在 OLE 文档服务器中加入一个客户动作。比如开发一个复合文档服务器，使之能够播放一些媒体剪贴（比如微软的媒体播放器）。很自然的，我们想要加入一个“Play”的动作。MFC 注册表只有“Edit”和“Open”，我们该如何加入“play”动作呢？

程序清单 11-22 展示了实现这个功能代码。存放这些代码的最好的地方是在应用程序的初始化代码中，比如，CWinApp::InitInstance()函数就是实现这些代码的很好的场所。

程序清单 11-22 注册自定义的 “play” 操作

```
{
...
   //Add custom verb.
   // Setup the verb string
   static TCHAR szCustomVerb[] = _T("CLSID\\%1\\Verb\\2\0") _T("&Play,0,2");
   LPCTSTR lpszRegister[2];
   lpszRegister[0] = szCustomVerb;
   lpszRegister[1] = NULL;

   LPCTSTR lpszSymbols[2];
   LPTSTR lpszClassID;
   ::StringFromCLSID(clsid, &lpszClassID);
   lpszSymbols[0] = lpszClassID;
   lpszSymbols[1] = NULL;

   AfxOleRegisterHelper(lpszRegister, lpszSymbols, 1, TRUE);

   AfxFreeTaskMem(lpszClassID);
...
}
```

AfxOleRegisterHelper()的参数有两个数组，数组的元素是指向 TEXT 类型的字符串的指针。第一个数组保存注册表的键以及它们相对应的值，AfxOleRegisterHelper()通过 NULL 字符将这两者分开。在前面的例子中，szCustomVerb 将键存放在字符串的第一部分，把值存放在第二部分。lpszRegister 数组的第一个元素是 szCustomVerb 的地址，第二个元素是一个 NULL 指针。AfxOleRegisterHelper()通过这种机制来判别是否到达注册条目链表的最后。

第二个字符串数组是一个用来替换的符号的集合。在这个例子中只有一个符号：需要被注册的对象的 Class ID。同样地，这个数组也是以 NULL 指针结尾。注意一下在 szCustomVerb 中的“%1”，AfxOleRegisterHelper()使用这个符号来表示替换符号数组中的第一个元素。如果一个字符串包含“%2”，则表示 AfxOleRegisterHelper()将替换符号数组中的第二个元素。

符号数组的第一个元素存放的是用来表示对象的 Class ID 的字符串的地址。StringFromClassID()是一个 OLE 函数，它能够将任何一个没有任何规律的 GUID 转换为用平常语言表达的字符串。

第三个参数用来表示符号数组中是否只有一个符号。第四个参数表示 AfxOleRegisterHelper()是否要覆盖原来的键及其值（如果已经存在的话）。

通过调用上面定义的代码，AfxOleRegisterHelper()将会在注册表中给键“CLSID\{81634F00-DF4F-11CE-AEFF-A1970E160F92}\Verb\2”增加下面这个值：“&Play,0,2”。（当然了，你的 GUID 可能是不同的。）

COleObjectFactory 的核心：OnCreateObject()

由于 COleObjectFactory 支持 IClassFactory 接口，所以它必须实现 IClassFactory::

CreateInstance()。COleObjectFactory::OnCreateObject()就是实现这个函数的地方。还是回到我们的 CoMath 例子。在 CoMath 中，CoMath 的类厂使用 new 操作符来创建 CoMath 的实例。而我们还记得 COleObjectFactory 的构造函数的其中一个参数是指向类的运行时类信息的指针。只要一个类实现了动态创建机制（使用 DECLARE_DYNCREATE()和 IMPLEMENT_DYNCREATE()宏），这个类的运行时信息结构中就会包含一个 CreateObject()函数。该函数用来在外部创建这个类的一个实例。在 COleObjectFactory 中，正是使用 MFC 的动态创建机制，而不是用 new 操作符来创建 COM 类的实例。

严格来说，一个 COM 类使用 COleObjectFactory 作为它的类厂，也没有必要一定要用动态的创建方法。OnCreateObject()是一个虚函数，所以我们可以在需要的时候对它进行重载。如果由于某些原因，你需要使用别的机制来创建 COM 对象，就可以重载 COleObjectFactory::OnCreateObject()。可以使用 new 操作符或是别的一些机制来创建 COM 对象。一般没有很好的理由需要重载，但是起码编程者可以自由地重载 OnCreateObject()。

COleObjectFactory 和 IClassFactory::LockServer()

除了 CreateInstance()，IClassFactory 还有另外一个成员函数是 LockServer()。LockServer()用来控制服务器的引用计数。一个 MFC 模块（通常是 CWinApp）在它的 AFX_WIN_STATE 结构中包含一个引用计数，名字叫 m_nObjectCount。有两个全局的 MFC 函数分别用来给引用计数加 1 和减 1，这两个函数分别是 AfxOleLockApp()和 AfxOleUnlockApp()。IClassFactory::LockServer()就是使用 AfxOleLockApp()和 AfxOleUnlockApp()来管理应用程序的引用计数。

每次当应用程序调用 AfxOleUnlockApp()时，会将应用程序的对象计数减 1。如果对象计数的值到了 0，则 AfxOleUnlockApp()会调用 AfxOleOnReleaseAllObject()函数。AfxOleOnReleaseAllObject()控制着服务器的生存期。这个函数首先检查用户是否还控制着应用程序（比如这个应用程序是 MDI 的，而用户又新打开了一个文件），如果用户确实没有控制着应用程序，则 AfxOleOnReleaseAllObject()终止应用程序。如果应用程序有一个主窗口，则 AfxOleOnReleaseAllObject()会销毁这个窗口，否则，发送一个 WM_QUIT 的消息来结束这个应用程序。

在客户的应用程序中创建类厂

MFC 使得给自己的应用程序添加类厂非常容易。有两个宏用来在 MFC 中给某个特定的 COM 类添加类厂。这两个宏是 DECLARE_OLECREATE 和 IMPLEMENT_OLECREATE。跟别的相对应的宏（比如动态运行时信息、动态创建、序列化等）类似，DECLARE...宏在头文件中使用，而 IMPLEMENT...在源文件中使用。也就是说要为一个 COM 类创建类厂，在头文件中使用 DECLARE_OLECREATE，在 CPP 文件中使用 IMPLEMENT_OLECREATE。

下面是 DECLARE_OLECREATE()的定义：

```
#define DECLARE_OLECREATE(class_name) \
protected: \
        static AFX_DATA COleObjectFactory factory; \
        static AFX_DATA const GUID AFX_CDECL guid; \
```

DECLARE_OLECREATE()为特定的类声明一个 COleObjectFactory 的静态实例和一个与这个类相对应的静态 GUID。

下面是 IMPLEMENT_OLECREATE()的定义：

```
#define IMPLEMENT_OLECREATE(class_name, external_name, l, w1, w2, b1, \
        b2, b3, b4, b5, b6, b7, b8) \
    AFX_DATADEF COleObjectFactory class_name::factory(class_name::guid, \
        RUNTIME_CLASS(class_name), FALSE, _T(external_name)); \
    const AFX_DATADEF GUID AFX_CDECL class_name::guid = \
        { l, w1, w2, { b1, b2, b3, b4, b5, b6, b7, b8 } }; \
```

IMPLEMENT_OLECREATE()创建类厂，并且将 GUID 赋值为 DECLARE_OLECREATE()中声明的值。

```
protected:
static  COleObjectFactory factory; static  const GUID __cdecl guid;
```

下面是 IMPLEMENT_OLECREATE 宏在 CoMath 中展开后的代码：

```
COleObjectFactory CoMath::factory(CoMath::guid,
                                  (&CoMath::classCoMath), 0,
                                  "CoMath"); /
  const  GUID __cdecl CoMath::guid = { 0x06812840, 0x46e9, 0x11ce,
         { 0xae, 0xff, 0xe7, 0x98, 0xa8, 0x72, 0x14, 0x16 } };;
```

COleObjectFactory 和接口映射表

在我们结束类厂的讨论之前，让我们对它进行一个小结。类厂本质上来说也是一个 COM 类，而且微软确实是使用接口映射表来实现它。在 COleObjectFactory 的定义中我们可以看到 DECLARE_INTERFACE_MAP() 宏 和 BEGIN_INTERFACE_PART() /END_INTERFACE_PART()宏的使用，根据我们前面的讨论，当然知道这是在实现一个 COM 接口。COleObjectFactory 实现了 IClassFactory。我们可以看到在 COleObjectFactory 的定义中，在宏 BEGIN_INTERFACE_PART()中使用了 IClassFactory，并且 IClassFactory 的函数原型也被实现。

程序清单 11-23 展示了预处理后的 COleObjectFactory 的代码。注意一下接口映射表是如何声明 XClassFactory 嵌套类和这个嵌套类中的 IUnknown 和 IClassFactory 函数。它告诉我们 MFC 的接口映射表是如何运转的。

程序清单 11-23 预处理后的 COleObjectFactory

```
class COleObjectFactory : public CCmdTarget
{
public:
   static  CRuntimeClass classCOleObjectFactory;
   virtual CRuntimeClass* GetRuntimeClass() const;

public:
   COleObjectFactory(const CLSID & clsid,
                     CRuntimeClass* pRuntimeClass,
                     BOOL bMultiInstance,
                     LPCTSTR lpszProgID);


   BOOL IsRegistered() const;
   const CLSID & GetClassID() const;
```

```
    BOOL Register();
    void Revoke();
    void UpdateRegistry(LPCTSTR lpszProgID = 0);
    BOOL IsLicenseValid();

    static BOOL __stdcall RegisterAll();
    static void __stdcall RevokeAll();
    static BOOL __stdcall UpdateRegistryAll(BOOL bRegister = 1);


protected:
    virtual CCmdTarget* OnCreateObject();
    virtual BOOL UpdateRegistry(BOOL bRegister);
    virtual BOOL VerifyUserLicense();
    virtual BOOL GetLicenseKey(DWORD dwReserved, BSTR* pbstrKey);
    virtual BOOL VerifyLicenseKey(BSTR bstrKey);

public:
    virtual ~COleObjectFactory();

public:
    COleObjectFactory* m_pNextFactory;

protected:
    DWORD m_dwRegister;
    CLSID m_clsid;
    CRuntimeClass* m_pRuntimeClass;
    BOOL m_bMultiInstance;
    LPCTSTR m_lpszProgID;
    BYTE m_bLicenseChecked;
    BYTE m_bLicenseValid;
    BYTE m_bRegistered;
    BYTE m_bReserved;


public:
    class XClassFactory : public IClassFactory2 {
    public:
        virtual ULONG __export __stdcall AddRef();
        virtual ULONG __export __stdcall Release();
        virtual HRESULT __export __stdcall QueryInterface(const IID & iid,
                                                          LPVOID* ppvObj);

        virtual HRESULT __export __stdcall CreateInstance(LPUNKNOWN,
                                                          const IID &,
                                                          LPVOID*);
        virtual HRESULT __export __stdcall LockServer(BOOL);
        virtual HRESULT __export __stdcall GetLicInfo(LPLICINFO);
        virtual HRESULT __export __stdcall RequestLicKey(DWORD, BSTR*);
        virtual HRESULT __export __stdcall CreateInstanceLic(LPUNKNOWN,
                                                             LPUNKNOWN,
                                                             const IID &,
```

```
                                                    BSTR,
                                                    LPVOID*);
    } m_xClassFactory;
    friend class XClassFactory;

    private:
       static const AFX_INTERFACEMAP_ENTRY _interfaceEntries[];
    protected:
       static  const AFX_INTERFACEMAP interfaceMap;
       virtual const AFX_INTERFACEMAP* GetInterfaceMap() const;

    friend SCODE __stdcall AfxDllGetClassObject(const CLSID &,
                                                const IID &,
                                                LPVOID*);
    friend SCODE __export __stdcall DllGetClassObject(const CLSID &,
                                                      const IID &,
                                                      LPVOID*);
};
```

到现在为止，COM 只剩下一个要点没涉及了：MFC 如何实现 DllGetClassObject()和 DllCanUnloadNow()。

从一个 DLL 中提供类厂给客户

当客户调用 CoCreateInstance()或 CoGetClassObject()时，DllGetClassObject()会被调用。DllGetClassObject 就是用来提供类厂的，下面是它的原型：

```
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, IPVOID* ppv);
```

其中第一个参数是请求的 COM 类的 GUID；第二个参数是客户请求的接口的 GUID；第三个参数用来存放返回的接口指针的地址（返回的接口通常是 IClassFactory，因为这个函数通常是用来得到类厂的）。

MFC 在框架中对这个函数提供了支持。COleObjectFactory 有一个友元函数：AfxDllGetClassObject()。MFC 实现了 DllGetClassObject()这个用来提供类厂的函数，MFC 中这个函数的版本就是 AfxDllGetClassObject()，COleObjectFactory 将 AfxDllGetClassObject()声明为友元函数，是因为这个函数跟框架支持类厂的方式是紧密相关的。

MFC 在每一个 CWinApp 派生类中都会保存一个类厂链表。当一个 COleObjectFactory 实例被创建之后，MFC 都会将它加入到这个应用程序的类厂链表中去。AfxDllGetClassObject()遍历类厂链表，查询 rclsid 表示的 COM 类对应的类厂。如果 AfxDllGetClassObject()找到了对应的类厂，则对类厂调用一次 QueryInterface()，从而查询 riid 参数表示的接口。如果 QueryInterface()成功了，则 AfxDllGetClassObject()将接口接口指针放到 ppv 参数中；如果 AfxDllGetClassObject()不能找到类厂，则返回 CLASS_E_CLASSNOTAVAILABLE 这个错误码。

当客户程序调用 CoFreeUnusedLibraries()时，DllCanUnloadNow()就会被调用。这个函数返回一个布尔值，用来表示这个库能否被卸载。通常这个函数会检查整个服务器的引用计数。如果服务器能够被释放，则 DllCanUnloadNow()返回 TRUE，否则返回 FALSE。要在框架中使用这个函数，只要调用 AfxDllCanUnloadNow()即可。客户可以使用 AfxOleLockApp()来给服务器增加一把锁，而 AfxOleUnlockApp()可以取消这个锁。框架知道有关锁的信息。AfxDllCanUnloadNow()检查框架的锁的信息，如果锁的个数已经减到 0，则返回 TRUE，否则返回 FALSE.

下面是 AfxDllCanUnloadNow()的工作方式。AfxDllCanUnloadNow()调用另外一个全局函数 AfxOleCanExitApp()来检查锁计数。AfxOleCanExitApp()检查变量 m_nObjectCount 的值，这是服务器的一个状态信息。如果还有没有消耗的锁计数，则立刻返回 S_FALSE，否则 AfxDllCanUnloadNow()遍历类厂链表，寻找引用计数大于 1 的类厂对象。如果 AfxDllCanUnloadNow()发现类厂还在使用中，则返回 S_FALSE。如果 AfxDllCanUnloadNow()发现没有类厂还在被使用，则返回 S_OK。

结 语

总之，一句话，MFC 提供了对 COM 的支持。所有的 OLE 特性都是通过 COM 体现的。为了支持 OLE 特性，一个框架必须实现 COM 兼容类。有很多种方法可以实现 COM 兼容类，最常用的两种分别是：（1）使用一个或多个接口的时候采用多继承；（2）使用嵌套类这种类的合成方式。MFC 使用了类的合成方式（也就是嵌套类）来实现 COM。MFC 通过 CCmdTarget 来实现 IUnknown。凡是从 CCmdTarget 派生的类，都能够支持 COM 接口。MFC 还提供了两个宏来构造嵌套类：BEGIN_INTERFACE_PART 和 END_INTERFACE_PART。

为了支持 COM 聚合，CCmdTarget 实现了两套 IUnknown：一个用来将请求委托给控制 IUnknown，而另一个用来处理类本身的生存期控制和 negotiation。委托 IUnknown 的函数是 CCmdTarget::ExternalAddRef()、CCmdTarget::ExternalRelease()和 CCmdTarget::ExternalQueryInterface()。非委托 IUnknown 的函数是 CCmdTarget::InternalAddRef()、CCmdTarget::InternalRelease()和 CCmdTarget::InternalQueryInterface()。

CCmdTarget 在 InternalQueryInterface()中处理真正的 QueryInterface()查找。为了加快查找速度，MFC 实现了一个查找表，叫做接口映射表。每一个被 CCmdTarget 派生类实现的接口都有一个相应的嵌套类，用这个嵌套类实现这个接口的功能。MFC 的接口映射表将一个接口 ID（一个 128 位的 GUID）和实现这个接口的嵌套类的地址关联起来。CCmdTarget 通过接口映射表来实现 QueryInterface()，它会遍历整个接口映射表，直到找到匹配的接口 ID 或是遍历完所有条目而没有找到匹配的接口。

MFC 通过 COleObjectFactory 提供对类厂的支持。COleObjectFactory 通过 MFC 标准的嵌套类和接口映射表实现了 IClassFactory。MFC 还在 COleObjectFactory 中增加了一些新的功能，用来支持运行时类厂的注册以及注册表的更新功能。

如果能够好好研究 MFC 的源代码，我们会发现上面这些机制在所有的 MFC/OLE 类中都用到了。现在，我们已经知道了如何使用 MFC 和接口映射表来创建一个进程内服务器。不过在 Tech Note 38 中（在 Visual C++中），所提到的有关接口映射表的信息稍嫌粗略了一些，而且没有有关 AfxGetClassObject()和 AfxDllCanUnloadNow()的文档。所以当你需要创建一个类，而这个类需要实现一些 MFC 不支持的接口时，本章的内容就会比较有用了。当然，这种情况是不多见的，但是起码需要的时候你知道该如何去做。

有时候 MFC 调试的时候，会跟踪到 MFC 的源代码中去（比如 assertion 错误等等）。了解接口实现方式在你跟踪实现 OLE 接口的 MFC 源代码的时候就会比较有用了。否则，看到那些嵌套类的时候，真的会让人觉得无所适从。

现在我们已经知道 MFC 如何实现 COM，而另外的一些 OLE 特性，比如自动化（atuomation）、OLE 文档和 OLE 控件也是非常重要的。下面几章就分别介绍 OLE 的这些特性。

CHAPTER

12 统一数据传输和 MFC

这一章的内容是 OLE 的数据传输方式，即 UDT（Uniform Data Transfer，统一数据传输）。首先，我们会了解一下在传统的 Windows 系统中，数据传输方式

所面临的一些困难。然后我们会讲述 IDataObject 接口以及使用 MFC 如何实现 UDT 的剪贴板传输。最后我们会讨论一下 OLE 的拖放（drag-and-drop）以及 MFC 如何实现拖放。

历史回顾

Windows 对终端用户来说，是一个很丰富的操作系统，因为它提供了许多很好的特性。比如图形用户界面（GUI）就包括了可调整大小的框架、图标、对话框和另外一些 GUI 所应该具有的所有功能。除了所有的可视化的用户界面特性外，Windows 的其中一个最有用的特性是它的 IPC（Inter-Processd Communication，进程间通信）机制。而在这之前，基于 DOS 的系统中，数据从一个程序传输到另一个程序是很不方便的。用户不得不用一些现在看来是不可思议的方式，比如通过手工将数据拷贝到临时文件中（有时候还不得不改变格式），然后使用另外一种格式将它读取出来。幸运的是，在 Windows 下，我们再也不需要这么做了。

标准的 Windows 提供了两个基本的 IPC 机制：剪贴板（clipboard）和动态数据交换（dynamic data exchange）。虽然这些机制比 DOS 下那种数据传输是好了不少，但是还远远称不上完美。除了这些数据传输的标准方法外，还可以通过 DLL 进行数据共享。但是，所有这些方法都有其天生的不足。由于使用这些机制的困难性（我们将在下面讨论这个），微软提供了一种基于 OLE 特性的标准数据传输机制：UDT。

为了了解 UDT 的背景及环境，让我们先来看看在没有 OLE 之前 Windows 的数据传输。

以前的数据传输方法

在一个应用程序内部或是在两个应用程序之间传输数据在现代操作系统中是一个很重要的功能。没有单独一个程序能在各个方面都提供很好的性能。比如，很多字处理器在处理文本方面性能非常高，但是一旦要编辑一幅图片的时候，它却还有许多有待改进之处。在这个例子中，就必须能够将一幅图片从画图程序中拷贝出来，将它粘贴到你的文档中。

如果在 Windows 环境中，则有两种方式可以在一个应用程序中或两个应用程序之间传输

数据：通过使用 Windows 剪贴板和使用动态数据交换。

剪贴板是 Windows 最灵活的一项功能之一，也是最容易使用的功能。使用剪贴板，可以使在一个应用程序中或在不同的应用程序之间传输不同的数据类型变得非常简单。现在几乎每个 Windows 应用程序在主菜单中都至少会提供拷贝和粘贴的功能（一般是在 Edit 菜单下）。当你在菜单上选了拷贝，应用程序通常会将屏幕上指定的数据或文本拷贝到 Windows 剪贴板中。然后可以转到别的任意一个应用程序并选择粘贴选项，这时通常会取得剪贴板的信息并将它插入到当前的应用程序中。

Windows 剪贴板编程

加入 Windows 剪贴板支持是比较简单的事情。我们先很快地看一下如何对 Windows 剪贴板编程，因为它对于理解 OLE 剪贴板是很重要的。为了将数据拷贝到剪贴板，需要以下几个步骤：

1. 调用 OpenClipboard()告诉剪贴板数据在哪个窗口中。
2. 调用 EmptyClipboard()将剪贴板以前的数据清空。
3. 调用 SetClipboardData()将数据放到剪贴板上。在这个地方，可以有两种选择：可以真的将数据拷贝到剪贴板上或是只是通知剪贴板我有数据，只有在真正使用的时候才提供数据。SetClipboardData()有两个参数：一个 CLIPFORMAT 常量（可以是 CF_TEXT、CF_BITMAP 等等）和一个数据句柄。如果调用 SetClipboardData()的时候提供一个非 NULL 的句柄，则剪贴板保存这个数据，否则，如果传的参数是一个 NULL 的指针，那么就是客户通知剪贴板有数据，但是在需要数据的时候再提供。
4. 调用 CloseClipboard()使别的应用程序能够访问剪贴板。

如果剪贴板上的数据可用了，则别的应用程序就可以使用这些数据了。为了取得剪贴板上的数据，需要以下几个步骤：

1. 调用 OpenClipboard()访问剪贴板。
2. 调用 GetClipboardData()取得数据。
3. 调用 CloseClipboard()释放剪贴板。

如果 Windows 剪贴板确实存放着数据，则没有问题，剪贴板通过 GetClipboardData()将数据传输给使用者即可。如果数据产生者采用的是延迟供应（delayed rendering）（通过在调用 SetClipboard 的时候使用一个 NULL 指针），则 Windows 往数据提供者发送一个 WM_RENDERFORMAT 消息。数据提供者响应 WM_RENDERFORMAT 消息，并按照需要的格式产生数据。

动态数据交换

DDE（Dilapidated Data Exchange，动态数据交换）是 Windows 另外一种标准的数据传输方式。使用 DDE 的时候，两个应用程序之间必须建立一个会话，使数据交换能够进行。两个处于 DDE 会话中的应用程序能够交换原始数据，能够通知对方数据的改变，甚至还能够在另外一个上面执行命令。不幸的是，标准的 Windows 数据传输机制也有它的局限性。

Windows 剪贴板和 DDE 的局限性

剪贴板和 DDE 都有一个基本的问题，那就是它们都是依靠全局内存来传输数据。这个

会有问题，特别是在传输条目的时候（比如一个与设备无关的位图）。当使用全局内存进行数据传输的时候，Windows 必须要利用虚拟内存管理机制来完成很大内存的分配，也就是说 Windows 不得不将数据交换到硬盘上去，而在内存和硬盘之间交换数据需要很长的时间，这是不合适的。

使用 Windows 剪贴板进行数据传输的另外一个很明显的问题是只有一种方式来解释剪贴板上的数据：通过一个 16 位的整数。Windows 提供了几种标准的剪贴板格式（比如 CF_TEXT 或 CF_BITMAP），你也可以注册自己的格式。虽然可以描述数据的格式，但是并没有办法可以描述数据要以何种方式提供。

不幸的是，OLE 的第一个版本并没有多大的改进。由于 OLE1 的根本的数据传输还是通过 DDE 来进行，所有它必然也有 DDE 所具有的问题。现在 OLE 有了一种新的数据传输方式。OLE 的协议（UDT）解决了数据传输的问题。UDT 提供了一个统一的方式在应用程序中或在应用程序间传输数据。说它是统一的是因为所有的数据传输都通过 IDataObject 接口来进行（IDataObject 也是一个 COM 接口）。

OLE 和 UDT

使用 OLEUDT 原来的目的是为了支持 OLE 文档。除了在 OLE 文档中使用外，UDT 在拷贝粘贴剪贴板上数据以及进行 OLE 拖放都是很有用的。下面让我们来看看 IDataObject 这个 UDT 的主要接口。

重要的结构

使用 OLE 通过剪贴板或是拖放进行数据传输是比较简单的一件事情。然而，在我们了解通过剪贴板进行数据传输之前，让我们先来看看两个和 UDT 密切相关的很重要的结构：FORMATETC 和 STGMEDIUM。FORMATETC 结构描述了传输中的真正的数据。而 STGMEDIUM 结构描述了传输中使用的媒介。UDT 允许数据传输通过很多种媒介进行，而不仅仅是全局内存。

FORMATETC 结构

FORMATETC 结构描述数据的方法比传统的 Windows 剪贴板格式要丰富得多。除了描述数据格式外，这个结构还能描述数据以何种方式提供，这包括目标设备以及数据各个方面情况（比如是否将一幅位图或是一幅微型画作为数据提供）。FORMATETC 结构如下所示：

```
typedef struct tagFORMATETC {
    CLIPFORMAT          cfFormat;
    DVTARGETDEVICE *    ptd;
    DWORD               dwAspect;
    LONG                lindex;
    DWORD               tymed;
} FORMATETC;
```

第一个域是剪贴板格式（CLEPFORMAT），它的值可以是任何标准剪贴板格式，比如 CF_TEXT 和 CF_BITMAP，也可以是私有的应用程序格式（通过 RegisterClipboard()注册），而且还可以是 OLE 剪贴板格式，比如 CF_EMBEDSOURCE 或 CF_LINKSOURCE。注意一下 FORMATETC 留了空间来表示所有的老的剪贴板格式（在 cfFormat 域中）。但是 FORMATETC 除了表示传输的数据格式外，还有别的功能。

第二个域是一个指向结构的指针，这个结构描述了用来显示数据的设备的信息。通常它描述的是打印机，但是除了打印机，它还能描述任何设备。

第三个域 dwAspect 描述了数据的显示方式。dwAspect 中的值类型是 DVASPECT 枚举类型。如下所示：

```
typedef enum tagDVASPECT
{
    DVASPECT_CONTENT        = 1,
    DVASPECT_THUMBNAIL      = 2,
    DVASPECT_ICON           = 4,
    DVASPECT_DOCPRINT       = 8,
} DVASPECT;
```

虽然 DVASPECT 的各个值是不相关的标志位，但是任何时候只能一位为 1，也就是说，dwAspect 只能表示一个值。表 12-1 说明了这个枚举中每个值代表的意义。

表 12-1 描述复制图中应包含哪些细节的枚举

值	含义
DVASPECT_CONTENT	当嵌在容器内部时对象的一种表示。对于复合文档对象而言，这个值是最常见的。你可以用 DVASPECT_CONTENT 获得一个在屏幕上或打印上生面的嵌入对象的表示
DVASPECT_DOCPRINT	暗示对象应该按被打印那样显示
DVASPECT_THUMBNAIL	对象的一个微缩表示，这种表示通常是一个独立于设备的位图，它的大小是 120×120 个像素，有 16 色（推荐值），这个位图会包装在一个元文件（metafile）里。在浏览器里显示对象时，这一点十分有用。
DVASPECT_ICON	对象的一个图标表示
DVASPECT_DOCPRINT	表示像使用“文件”菜单里“打印”命令打印出的对象。描述它的数据表示一系列页

第四个域 lindex，它表示的意思和 dwAspect 的值相关。如果 dwAspect 的值是 DVASPECT_ICON 或是 DVASPECT_THUMBNAIL，则 lindex 被忽略。如果 dwAspect 的值是 DVASPECT_CONTENT 或 DVASPECT_DOCPRINT，则 lindex 表示感兴趣的那方面的信息。比如，它可以表示数据所需的页数。

最后一个域是 tymed，它指向存放数据的介质的类型。OLE 头文件定义了一个叫 TYMED 的枚举，用来表示数据传输中用到的各种介质类型。TYMED 有以下几个值：TYMED_HGLOBAL、TYMED_FILE、TYMED_ISTREAM、TYMED_ISTORAGE、TYMED_GDI、TYMED_MFPICT、TYMED_ENHMF 和 TYMED_NULL。

在数据传输中，数据处理（data producers）表示为数据分配内存，而数据消耗（data

了一个叫 ReleaseStgMedium()的包装函数，用它可以释放存放在 STGMEDIUM 中的数据。ReleaseStgMedium()使用一个指向 STGMEDIUM 的指针作为参数，它会调用恰当的方法来释放数据。

IDataObject 接口

UDT 的核心是一个叫 IDataObject 的接口，IDataObject 也是一个 COM 接口，并且你也可以选择去实现它（在某些情况下也可以不实现）。下面是 IDataObject 的定义：

```
interface IDataObject : IUnknown {
   virtual  HRESULT  GetData(FORMATETC* pformatetc, STGMEDIUM* pmedium)= 0;
   virtual  HRESULT  GetDataHere(FORMATETC* pformatetc, STGMEDIUM*
                                 pmedium) = 0;
   virtual  HRESULT  QueryGetData(FORMATETC* pformatetc) = 0;
   virtual  HRESULT  GetCanonicalFormatEtc(FORMATETC* pformatetcIn,
                                           FORMATETC* pformatetcOut) = 0;

   virtual  HRESULT  SetData(FORMATETC* pformatetc, STGMEDIUM* pmedium,
                             BOOL fRelease) = 0;
   virtual  HRESULT  EnumFormatEtc(DWORD wDirection, IEnumFORMATETC**
                                   penumformatetc) = 0;
   virtual  HRESULT  DAdvise(FORMATETC* pformatetc, DWORD grfAdvf,
                             IAdviseSink* pAdvSink, DWORD* pdwConnection) = 0;
   virtual  HRESULT  DUnadvise(DWORD dwConnection) = 0;
   virtual  HRESULT  EnumDAdvise(IEnumSTATDATA** ppenumAdvise) = 0;
};
```

跟所有的 COM 接口一样，IDataObject 也从 IUnknown 派生。除了 IUnknown 的 3 个函数外，IDataObject 还有 9 个用来实现数据传输的函数。下面让我们简略地浏览一下这些函数。

IDataObject::GetData()

很自然，一个支持数据传输的接口当然需要有一个获取数据的函数。GetData()从实现了 IDataObject 的对象中提取数据。让我们看一下 GetData()的参数：一个指向 FORMATETC 结构的指针和一个指向 STGMEDIUM 结构的指针。IDataObject::GetData()用来获取在 FORMATECT 结构中指定的数据（如果它有效的话），并且将数据传回到 STGMEDIUM 结构中。

IDataObject::GetDataHere()

IDataObject::GetDataHere()和 IDataObject::GetData()类似，惟一的区别是调用者提供真正的介质，而不是由被调用者来提供。

IDataObject::QueryGetData()

IDataObject::QueryGetData()可以使调用者来查询 FORMATETC 描述的数据和介质是否有效。

IDataObject::GetCanonicalFormatEtc()

在某些情况中，一个数据对象可能对多个 FORMATETC 结构返回同样的数据。

IDataObject::GetCanonicalFormatEtc()提供了这样一种机制，它使得数据对象与 FORMATETC 产生相同数据的调用者之间建立通信。

IDataObject::SetData()

IDataObject::SetData()是 IDataObject::GetData()的补充操作，它将数据放到数据对象中。

IDataObject::EnumFormatEtc()

调用者使用 IDataObject::EnumFormatEtc()来枚举数据对象通过 SetData()和 GetData()分别进行存放和取得的格式。

IDataObject::DAdvise()

客户程序调用 IDataObject::DAdvise()在客户程序的通报接收器（advisory sink）和数据对象之间建立一个通报连接（advisory connection）。当数据对象中的数据发生变化时，它就可以通过通报连接通知客户的通报接收器。

IDataObject::DUnadvise()

IDataObject::DUnadvise()将 IDataObject::DAdvise()所建立的通报连接断开。

IDataObject::EnumDAdvise()

IDataObject::EnumDAdvise()用来枚举数据对象当前所建立的通报连接。

以上所介绍的就是 OLE 的数据传输接口——IDataObject。下面让我们看看 OLE 剪贴板是如何使用它的。

OLE 剪贴板

OLE 剪贴板从标准的 Windows 剪贴板模型扩展而来。除了标准的 Windows 剪贴板 API，OLE 还给标准 Windows 剪贴板补充了 OLE 的数据传输机制。OLE 剪贴板对标准的 Windows 剪贴板向后兼容。而事实上，OLE 剪贴板是介于应用程序和标准 Windows 剪贴板之间的（见图 12-1）。

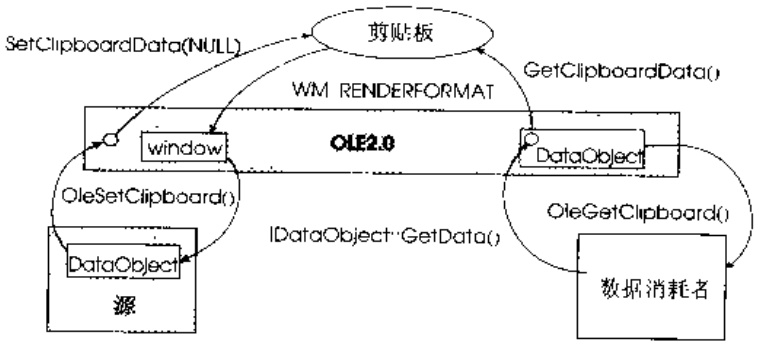


图 12-1 OLE 剪贴板

OLE 剪贴板通过 IDataObject 接口进行传输。有 4 个 API 函数支持 OLE 剪贴板：

- OleSetClipboard()——在剪贴板上放一个 IDataObject 接口指针。
- OleGetClipboard()——从剪贴板上取得一个 IDataObject 接口。
- OleFlushClipboard()——清空 OLE 剪贴板，将剪贴板上的 IDataObject 指针释放。
- OleIsCurrentClipboard()——判断指定的一个数据对象当前是否在剪贴板上。

下面是 OLE 剪贴板的工作步骤：一个应用程序将数据放到剪贴板上并实现 IDataObject，这个应用程序就是数据创造者。数据创造者调用 OleSetClipboard()，这个函数得到 IDataObject 指针的一个拷贝，并将它放到 OLE 剪贴板上。

一旦 IDataObject 指针放到了剪贴板上，OLE 就可以像普通的 Windows 程序一样使用 Windows 剪贴板。OLE 调用 OpenClipboard()来声明剪贴板的拥有者。OLE 剪贴板使用延迟供应的数据传输模式。OpenClipboard()需要一个窗口句柄，在调用 OleInitialize()的时候，OLE 剪贴板会创建一个窗口。这个是一个隐藏窗口，一般的客户在外面不能看见。（如果要使用 OLE 剪贴板，就必须调用 OleInitialize()——CoInitialize()并不能创建这个窗口。）

接着 OLE 枚举 IDataObject 的格式，同时对每一个在全局句柄中提供数据的格式调用 SetClipboard()。这是 OLE 和标准剪贴板不同的地方。标准剪贴板并不能支持文件和结构的传输，所以它只能将全局句柄放到剪贴板上。当 OLE 调用 SetClipboard()的时候，它会使用 NULL 指针，也就是告诉 Windows 剪贴板现在使用延迟供应方式提供数据。

现在，一个数据消耗者需要访问剪贴板。如果这个数据消耗者并不知道 OLE 的信息，那么它会利用标准的 Windows 方式也就是调用 GetClipboardData()来取得数据。由于 OLE 采用的是延迟供应的模式，所以数据提供者分几步提供数据。OLE 剪贴板首先查询 IDataObject 接口指针，然后调用 IDataObject 的 GetData()方法来取得数据。

如果数据消耗者也支持 OLE，则它可以调用 OleGetClipboard()从剪贴板中取得 IDataObject 的指针。然后它就可以调用 IDataObject::GetData()来取得数据了。OLE 剪贴板真正保存的是它自己的 IDataObject 接口指针。如果数据提供者将一个 IDataObject 指针放到剪贴板上，则剪贴板的 IDataObject 接口就直接调用数据提供者的 IDataObject。如果数据提供者没有在剪贴板上放 IDataObject 接口指针，则 OLE 剪贴板的 IDataObject 通过标准的 Windows 方式来访问剪贴板，它会给拥有剪贴板的窗口发送一个 WM_RENDERDATA 消息。

现在我们已经清楚了 OLE 剪贴板的工作方式，下面让我们来了解一下 MFC 是如何处理 IDataObject 和 OLE 剪贴板的传输。

MFC 的 IDataObject 类

MFC 中有两个类用来支持 IDataObject 接口：COleDataSource 和 COleDataObject。这两者的主要区别在于它们支持 IDataObject 的方式不同。COleDataSource 是一个完全的 COM 对象。如果去看 COleDataSource 的头文件和源代码，就可以发现它有一个接口映射表，也就是说 COleDataSource 是一个真正的 COM 对象。而 COleDataObject 却有一点不同：它封装了一个存在的 IDataObject 接口指针，使它成为一个对开发者很友好的 C++接口。

正如它的名字所示，COleDataSource 在数据传输中通常用在数据提供者一方，而 COleDataObject 通常在数据消耗者一方使用。在我们了解这两个类的细节之前，先让我们来看看在数据传输中如何使用这两个类。

通过剪贴板传输数据

在 MFC 程序中使用 OLE 剪贴板进行拷贝/粘贴数据是一件比较容易的事情。我们只要使

用两个面向 IDataObject 接口的 MFC 类即可，这两个类是 COleDataSource 和 COleDataObject。在产生数据的时候使用 COleDataSource，也就是在源程序的一方使用 COleDataSource。而在需要使用数据的时候就使用 COleDataObject，也就是在 MFC 的数据接收者一方使用 COleDataObject。使用 OLE 剪贴板进行拷贝和粘贴的步骤如下所述。

拷贝数据到剪贴板中

想要拷贝数据到剪贴板，首先必须得到指向数据的指针。例如，如果你想拷贝一段文本到剪贴板，则先要得到指向这段字符串的指针。接着是创建 COleDataSource 的一个实例，用这个对象来保存数据，直到数据消耗者访问它为止。

假设你想通过全局内存来传输数据，则调用 GlobalAlloc()来分配全局内存的句柄。然后使用 GlobalLock()得到一个有效的指向内存的指针，然后将数据拷贝到内存中。在将数据拷贝到全局内存中后，一定要调用 GlobalUnlock()。接着调用 COleDataSource 的 CacheGlobalData()函数将数据放入 COleDataSource 对象中。CacheGlobalData()有两个参数：数据的剪贴板格式（比如 CF_TEXT 或是 CF_BITMAP 等等）和存放数据的内存的句柄。最后，调用 SetClipboard()将数据对象放到剪贴板上。现在，另外一个应用程序就可以使用剪贴板上的数据了。下面的代码片断展示了如何将数据拷贝到剪贴板，从中也可以看出这不是一件很困难的事情。

```
{
    LPCSTR source = GetString();

    COleDataSource *pcods;
    pcods = new COleDataSource;
    HGLOBAL h =
       GlobalAlloc(GHND | GMEM_SHARE, strlen(source) + 1);
    strcpy(LPSTR(GlobalLock(h)), source);
    GlobalUnlock(h);
    pcods->CacheGlobalData(CF_TEXT, h);
    pcods->SetClipboard();
}
```

在这段代码中，你也许会看到一点让你觉得很危险的事情：这个函数创建了一个新的 COleObject 却没有将它释放！为什么会这样？在 MFC 的文档中，微软是这样解释的："在将数据放到 OLE 的情况下，比如调用 COleDataSource::SetClipboard()，并不用担心释放它的问题，因为 OLE 自己将会去释放它。"也就是说，在调用 COleDataSource::SetClipboard()后，你就已经将实现 IDataObject 的这个对象的所有权传递给了 OLE（在这个例子中，这个对象就是 COleDataSource 对象）。任何一个应用程序调用 OLE 的 API 函数::OleSetClipboard()时，OLE 都会清空剪贴板，并释放剪贴板上的 IDataObject 指针。当然，Release()将删除这个对象（由于不存在引用指针）。只要有一些数据消耗者可能需要使用它的数据，COleDataSource 对象就必须保留着不能被删除。

如果需要使用全局内存以外的介质，那也不是很困难。同样地，先创建 COleDataSource 类的一个实例。然后调用 COleDataSource::CacheData()来将数据放入数据源对象，而不是调用 COleDataSource::CacheGlobalData()。在使用上，COleDataSource::CacheData()比 COle-

DataSource::CacheGlobalData() 更灵活。COleDataSource::CacheData() 使用一个指向 FORMATETC 结构的指针和一个指向 STGMEDIUM 结构的指针作为参数，而不是一个剪贴板的格式和一个全局内存的句柄。FORMATETC 描述了数据的格式，而 STGMEDIUM 描述了传输数据的介质。一旦数据被拷贝到剪贴板，则别的应用程序就可以查询剪贴板，然后使用这些数据。从剪贴板中取得数据的过程叫做粘贴。

从剪贴板中粘贴数据

同使用 COleDataSource 将数据拷贝到剪贴板一样，从剪贴板获得数据也不困难。框架使用 COleDataObject 将数据传输到应用程序的数据消耗者这一端。

要从剪贴板中取得数据，首先要声明 COleDataObject 的一个实例，然后通过调用 COleDataObject::AttachClipboard()从剪贴板中取得数据。如果数据已经放到了 COleDataObject 中，则调用以下的两个方法中的一个来取得数据的格式：（1）调用 COleDataObject::IsDataAvailable()；（2）枚举各种格式。

发现有效的数据

COleDataObject::IsDataAvailable() 有两个参数：一个 CLIPFORMAT 和一个指向 FORMATETC 结构的指针。如果要向数据对象查询一个标准的剪贴板格式（比如 CF_TEXT 或 CF_BITMAP 等等），则将这个参数放到第一个参数中，并且将 FORMATETC 指针置为 NULL。如果在查询剪贴板的时候希望使用一些更详细的信息，那么填充 FORMATETC 结构，并将它传给 IsDataAvailable()。IsDataAvailable()返回一个布尔值来表示 COleDataObject 是否包含你所查询的格式的数据。

查询有效的数据格式的另外一个方法是枚举 COleDataObject 的所有剪贴板格式。COleDataObject 中提供了两个函数来做这件事情：BeginEnumFormats()和 GetNextFormat()。BeginEnumFormats()启动枚举的过程。在调用了 BeginEnumFormats()之后，你就可以调用 GetNextFormat()每次取得一个有效的格式。GetNextFormat()填充一个 FORMATETC 结构，用这个结构来描述数据的格式。如果 GetNextFormat()返回 TRUE，则这个数据是有效的。GetNextFormat()使用数据格式的信息来填充 FORMATETC 结构。

获取数据

在确定了数据有效性和数据的格式后，下一步要做的就是获取数据了。COleDataObject 提供了 3 个函数用来获取数据：GetData()、GetFileData()和 GetGlobalData()。

COleDataObject::GetData()提供了一种最常用的从数据对象中获取数据的方式。GetData() 有 3 个参数：一个 CLIPFORMAT，一个指向 STGMEDIUM 的指针和一个指向 FORMATETC 结构的指针。CLIPFORMAT 描述了返回的值的格式。GetData 将返回的值存放在 STGMEDIUM 中。FORMATETC 描述了在 CLIPFORMAT 中所没有描述的数据格式的信息。

如果你知道 COleDataObject 使用的存储介质是文件，则可以使用 COleDataObject::GetFileData()函数来获取数据。GetFileData()用两个参数来描述取得的数据格式：一个 CLIPFORMAT 标识符和一个指向 FORMATETC 结构的指针。GetFileData()返回一个指向 CFile 对象的指针，这个对象中存放着从 COleDataObject 取得的数据。注意一下返回的 CFile 可以是 COleStreamFile、CSharedFile 或是一个普通的 CFile，这取决于存放数据的介

质。还要注意的是，GetFileData()的调用者在使用完 CFile 后应该负责释放它。

如果你知道存储介质是全局内存句柄，则可以调用 COleDataObject::GetGlobalData()来获取数据。跟 GetFileData()一样，GetGlobalData()使用两个参数来描述获取的数据格式：一个 CLIPFORMAT 和一个指向 FORMATETC 结构的指针。GetGlobalData()返回一个全局句柄，你可以调用 GlobalLock()来锁定该句柄，从而获得一个指向全局内存的指针。下面的代码片断展示了如何使用 COleDataObject 从剪贴板中取得数据并且使用这些数据。

```
COleDataObject codo;

codo.AttachClipboard();

if (codo.IsDataAvailable(CF_TEXT)) {
   HANDLE h = codo.GetGlobalData(CF_TEXT);

   CEdit& rEditCtrl = GetEditCtrl();

   // Use data

   GlobalUnlock(h);
   GlobalFree(h);

   GetDocument()->UpdateAllViews(0);
   GetDocument()->SetModifiedFlag(TRUE);
}

codo.Release();
```

延迟供应

使用前面的方法进行剪贴板数据传输是一种立即供应（immediate rendering）的方式。也就是说，当数据产生者将数据放到剪贴板上后，数据消耗者只要一向剪贴板请求数据，则数据立即就可用。这是因为 COleDataSource 缓存功能使得在整个 COleDataSource 对象的生存期中，数据都在这个对象中真正保存着。

延迟供应是另外一种数据传输技术。在这种传输中，数据提供者只有在消耗者真正需要数据的时候才提供数据。在需要传输大量数据时，延迟供应是很有用的。因为在内存中长时间保留一大块的数据代价是很高的（比如在 COleDataSource 中保留这些数据）。一个更好的方法是告诉别人有数据，但是只有在真正用到的时候才提供数据。这个方法并不需要很大的系统资源来存放大块的数据。这个技术可以通过 COleDataSource 的 3 个成员函数来实现：DelayRenderData()、DelayRenderFileData()和 OnRenderData()。

要实现延迟供应，只要从 COleDataSource 中派生出一个类。你还需要派生出一个你自己的类，因为需要自己去重载 OnRenderData()函数。

当数据提供者调用 DelayRenderData()时，它就已经承诺提供数据，数据格式由一个 CLIPFORMAT 结构和一个 FORMATETC 结构来描述。这个函数在存储介质不是 CFile 对象的时候，需要延迟供应的时候被使用。

DelayRenderFileData()和 DelayRenderData()的工作方式类似，但是，它是为了使用 CFile 对象作为传输介质的数据延迟供应而专门设计的。它也使用 CLIPFORMAT 和一个 FORMATETC 结构作为它的参数。

那么在延迟供应中数据是如何到达数据消耗者的呢？COleDataSource 提供了 3 个成员函数在需要的时候的提供数据：OnRenderData()、OnRenderFileData()和 OnRenderGlobalData()。派生类必须重载这 3 个函数。

当数据消耗者通过 DelayRenderData()或者 DelayRenderFileData()请求数据时，MFC 通过 3 个 OnRender()函数（OnRenderData()、OnRenderFileData()或者 OnRenderGlobalData()）之一从数据提供者那里取得数据。这 3 个都是虚函数，你可以对它进行重载，然后将数据提供给数据消耗者。

框架调用 OnRenderData()获取数据。数据的格式在 CLIPFORMAT 和/或 FORMATETC 中说明，并且作为 OnRenderData()的参数传进去。OnRenderData()还有一个参数是指向 STGMEDIUM 结构的指针，它描述了在数据传输中使用的介质。要提供数据，则先检查 STGMEDIUM 的 tymed 域来确定数据消耗者需要什么样的存储介质，然后将数据拷贝到需要的介质中，并在 STGMEDIUM 中填充相应的域。

当数据通过 COleDataSource 的 DelayRenderFileData()成员函数放到剪贴板上后，如果数据消耗者需要取得这些数据，则框架会调用 OnRenderFileData()来实现。与 OnRenderData()类似，OnRenderFileData()通过一个 FORMATETC 类型的参数来描述数据格式信息。这个函数没有必要使用 STGMEDIUM 结构作为参数，因为它已经明确地指定使用文件作为传输介质。框架将会给一个 CFile 对象赋一个指针值，用这个 CFile 对象来存储数据。

最后一个 OnRender...()函数是 OnRenderGlobalData()。这个函数和 OnRenderFileData()很类似，除了框架调用这个函数是显式地使用全局内存句柄来传输数据。所以，在这个函数中不是使用一个 CFile 对象的指针（在 OnRenderFileData()中），而是将指针赋值给一个 HGLOBAL，用来将一个全局句柄传递给数据消耗者。

在传递 HGLOBAL*或 STGMEDIUM*参数进来的时候，需要注意一下。因为很有可能已经有一个非空的 HGLOBAL。在 GetData()和 GetDataHere()中都会调用 OnRender()函数。在 GetDataHere()中，调用者提供存放数据的空间，而在 GetData()中，STGMEDIUM 或 HGLOBAL 并没有被赋值。

例如，假设你通过调用 COleDataSource::DelayRenderData()通知客户你可以提供数据。然后客户会通过一个全局内存句柄来要求你的程序提供数据（通过调用 COleDataObject::GetGlobalData()来实现）。当客户试图从 COleDataObject 中提取数据时，框架就会调用你的 OnRenderGlobalData()函数。在 OnRenderGlobalData()函数中，将数据拷贝到一块全局内存中，并将参数 HGLOBAL 指向这块全局内存（如果 HGLOBAL 参数原来并没有被设置）。

有一点情况很有意思：提供数据的方式（立即的或是延迟的）对数据消耗者是透明的。当一个数据消耗者从剪贴板中获取数据时（如前面讨论的通过传统的方法粘贴数据），它并不知道这些数据是直接还是间接提供的（也就是说，它并不知道数据是立即供应，还是延迟供应）。

深入了解 MFC 的 IDataObject 类

总的来说，使用 MFC 实现剪贴板的数据传输还是比较容易的。或者将数据保存在一个 COleDataSource 的实例中，又或者调用其中一个延迟供应函数并且处理 OnRender...()函数。让我们了解一下内部细节。

COleDataSource

程序清单 12-1 显示了在 AFXOLE.H 中定义的 COleDataSource。

程序清单 12-1 COleDataSource 类

```
class COleDataSource : public CCmdTarget {
// Constructors
public:
   COleDataSource();

// Operations
   void Empty();

   // CacheData & DelayRenderData operations similar to ::SetClipboardData
   void CacheGlobalData(CLIPFORMAT cfFormat, HGLOBAL hGlobal,
      LPFORMATETC lpFormatEtc = NULL);    // for HGLOBAL based data
   void DelayRenderFileData(CLIPFORMAT cfFormat,
      LPFORMATETC lpFormatEtc = NULL);  // for CFile* based delayed render

   // Clipboard and Drag/Drop access
   DROPEFFECT DoDragDrop(
      DWORD dwEffects = DROPEFFECT_COPY|DROPEFFECT_MOVE|DROPEFFECT_LINK,
      LPCRECT lpRectStartDrag = NULL,
      COleDropSource* pDropSource = NULL);
   void SetClipboard();
   static void PASCAL FlushClipboard();
   static COleDataSource* PASCAL GetClipboardOwner();

   // Advanced: STGMEDIUM based cached data
   void CacheData(CLIPFORMAT cfFormat, LPSTGMEDIUM lpStgMedium,
      LPFORMATETC lpFormatEtc = NULL);    // for LPSTGMEDIUM based data
   // Advanced: STGMEDIUM or HGLOBAL based delayed render
   void DelayRenderData(CLIPFORMAT cfFormat, LPFORMATETC lpFormatEtc = NULL);

   // Advanced: support for SetData in COleServerItem
   //  (not generally useful for Clipboard or drag/drop operations)
   void DelaySetData(CLIPFORMAT cfFormat, LPFORMATETC lpFormatEtc = NULL);

// Overidables
   virtual BOOL OnRenderGlobalData(LPFORMATETC lpFormatEtc,
                                   HGLOBAL* phGlobal);
   virtual BOOL OnRenderFileData(LPFORMATETC lpFormatEtc, CFile* pFile);
   virtual BOOL OnRenderData(LPFORMATETC lpFormatEtc,
                             LPSTGMEDIUM lpStgMedium);
      // OnRenderFileData and OnRenderGlobalData are called by
      //  the default implementation of OnRenderData.
```

```
    virtual BOOL OnSetData(LPFORMATETC lpFormatEtc, LPSTGMEDIUM lpStgMedium,
        BOOL bRelease);
        // used only in COleServerItem implementation

// Implementation
public:
    virtual ~COleDataSource();

protected:
    AFX_DATACACHE_ENTRY* m_pDataCache;  // data cache itself
    UINT m_nMaxSize;  // current allocated size
    UINT m_nSize;     // current size of the cache
    UINT m_nGrowBy;   // number of cache elements to grow by for new allocs

    AFX_DATACACHE_ENTRY* Lookup(
        LPFORMATETC lpFormatEtc, DATADIR nDataDir) const;
    AFX_DATACACHE_ENTRY* GetCacheEntry(
        LPFORMATETC lpFormatEtc, DATADIR nDataDir);

// Interface Maps
public:
    BEGIN_INTERFACE_PART(DataObject, IDataObject)
        INIT_INTERFACE_PART(COleDataSource, DataObject)
        STDMETHOD(GetData)(LPFORMATETC, LPSTGMEDIUM);
        STDMETHOD(GetDataHere)(LPFORMATETC, LPSTGMEDIUM);
        STDMETHOD(QueryGetData)(LPFORMATETC);
        STDMETHOD(GetCanonicalFormatEtc)(LPFORMATETC, LPFORMATETC);
        STDMETHOD(SetData)(LPFORMATETC, LPSTGMEDIUM, BOOL);
        STDMETHOD(EnumFormatEtc)(DWORD, LPENUMFORMATETC*);
        STDMETHOD(DAdvise)(LPFORMATETC, DWORD, LPADVISESINK, LPDWORD);
        STDMETHOD(DUnadvise)(DWORD);
        STDMETHOD(EnumDAdvise)(LPENUMSTATDATA*);
    END_INTERFACE_PART(DataObject)

    DECLARE_INTERFACE_MAP()

    friend class COleServerItem;
};
```

正如你在头文件中看到的，COleDataSource 是专门为支持剪贴板和拖放传输设计的（注意一下“SetClipboard()”和“DoDragDrop()”函数）。我们先略过拖放，先来看看对剪贴板的支持。

让我们再来看一下 OLE 剪贴板的工作方式。当一个数据产生者需要将数据放到剪贴板上时，它只要实现 IDataObject 接口，并且调用 OleSetClipboard()将 IDataObject 接口指针放到剪贴板上。

COleDataSource 实现了 IDataObject，而且实现了接口映射宏。COleDataSource 还有一个成员函数叫 SetClipboard()。SetClipboard()调用全局 API 函数::OleSetClipboard()将 COleDataSource 的 IDataObject 接口指针放到剪贴板上。还有一个工作没完成，那就是将数据放入 COleDataSource 对象中。调用 COleDataSource 的成员函数 CacheData()和

CacheGlobalData()就可以完成这个工作。

缓存数据

缓存数据的原理是在 COleDataSource 中保存数据，从而在有客户请求数据的时候就可以直接提供了。这就要求 COleDataSource 使用某种数据结构来管理需要缓存的数据。事实上，COleDataSource 使用一个叫 AFX_DATACACHE_ENTRY 的结构来存储数据。下面是 AFX_DATACACHE_ENTRY 的定义：

```
struct AFX_DATACACHE_ENTRY
{
   FORMATETC m_formatEtc;
   STGMEDIUM m_stgMedium;
   DATADIR m_nDataDir;
};
```

上面的代码比较简单。这个结构包含了 3 个成员变量，一个是 FORMATETC 类型，一个是 STGMEDIUM 类型，还有一个是 DATADIR 类型。我们在前面已经见过了 FORMATETC 和 STGMEDIUM 结构类型，下面看一下 DATADIR 类型。DATADIR 是一个枚举类型，它描述了数据传输的方向。下面是在 OBJIDL.H 中定义的 DATADIR 类型：

```
enum tagDATADIR {
   DATADIR_GET     = 1,
   DATADIR_SET     = 2
} DATADIR;
```

DATADIR 在延迟供应的时候使用，我们将在下面讨论。

我们回过来再看一下 COleDataSource，就会发现在它里面包含了一个 AFX_DATACACHE_ENTRY 结构数组。成员变量 m_pDataCache 指向这个数组的第一个元素。使用 CacheData()和 CacheGlobalData()两个函数可以往这个数组添加元素。

CacheData()和 CacheGlobalData()函数内幕

COleDataSource::CacheData()和 COleDataSource::CacheGlobalData()是两个很类似的函数。这两个函数都是用来将数据缓存到一个 COleDataSource 对象中，使得有客户请求时可以立即提供数据。CacheData()将数据保存在一个 STGMEDIUM 结构中。而 CacheGlobalData()是一种特殊的处理，它将数据专门保存在一个 HGLOBAL 中。下面让我们来了解一下更为普遍的函数 CacheData()，从而搞清楚 COleDataSource 是如何缓存数据的。

如果一个应用程序需要使用 CacheData()来缓存数据，则它需要填充一个 STGMEDIUM 结构。另外，这个应用程序至少还应该提供剪贴板格式（CF_...等常量中的一个），用来表示数据的格式。如果应用程序还想用剪贴板格式以外的形式来表示数据，则还应该有一个 FORMATETC 结构，并且将这个结构作为 CacheData()的一个参数。

在 COleDataSource::CacheData()中包含了一个 FORMATETC 结构类型的参数。这个参数是可选的，如果调用者需要，可以只传递 CLIPFORMAT 参数。但是，AFX_D -ATACACHE_ENTRY 结构需要一个完整的 FORMATETC 结构，而不仅仅是一个 CLIPFORMAT。如果调用者没有传进去 FORMATETC 结构参数，则 CacheData()会调用_AfxFillFormatEtc()函数来填充这个结构。_AfxFillFormatEtc()使用以下的；默认值来填充 FORMATETC 结构：

```
lpFormatEtc->cfFormat = <the CF_ format provided by the caller>;
lpFormatEtc->ptd = NULL;
lpFormatEtc->dwAspect = DVASPECT_CONTENT;
lpFormatEtc->lindex = -1;
lpFormatEtc->tymed = (TYMED)-1
```

最后，CacheData()将 FORMATETC 结构的 tymed 域的值赋给调用者传递进来的 STGMEDIUM 结构的 tymed 域。

如果框架有一个有效的 FORMATETC 结构，则 CacheData()就会调用 COleDataSource::GetCacheEntry()在缓存中给数据开辟一块空间。GetCacheEntry()调用 COleDataSource 的 Lookup()遍历 AFX_DATACACHE_ENTRY 结构数组，寻找是否有条目的 FORMATETC 结构和调用者想要加入的相同。如果已经存在，则 Lookup()返回找到的 AFX_DATACACHE_ENTRY 地址。如果没有找到，则分配一个新的 AFX_DATACACHE_ENTRY，将它加入到数组中，并且返回一个指向新的 AFX_DATACACHE_ENTRY 的指针。通过这种方法，在 COleDataSource 的缓存条目中就不会有模糊了（可能有多个入口相同）。每一个在 FORMATETC 结构中描述的格式，在 COleDataSource 中至多只有一个条目与之对应。如果 GetCacheEntry()指向一个有效的 AFX_DATACACHE_ENTRY，则它会去填充 STGMEDIUM 结构。当然了，如果在结构中已经有数据了，则 GetCacheEntry()得先将它释放（调用 ReleaseStgMedium()）。如果数据已经存储到了 COleDataSource 中，则客户就可以通过下面的方式取得这些数据：先得到 IDataObject 的接口指针，然后调用 GetData()。（GetData()的工作方式将在下面讲述）

延迟供应内幕

当使用延迟供应方式时，数据源只有在客户真正请求数据时才提供数据。延迟供应和缓存数据的工作方式也基本一样。延迟供应也使用 AFX_DATACACHE_ENTRY，但是使用方式稍微有点不同。

跟 CacheData()一样，DelayRenderData()也使用 CLIPFORMAT 和 FORMATETC 两个参数。然而，DelayRenderData()没有使用 STGMEDIUM 参数。这是由于在延迟供应的机制下，数据源并不需要立即提供数据，只有当真正需要的时候才提供，所以 STGMEDIUM 结构就没有必要了。

跟 CacheData()一样，DelayRenderData()也调用_AfxFillFormatEtc()。跟以前一样，DelayRenderData()使用 GetCacheEntry()来重用一个缓存条目（如果这个 FORMATETC 结构已经存在），或者在需要的时候创建一个 AFX_DATACACHE_ENTRY 条目。DelayRenderData()不是将一个 STGMEDIUM 结构存到缓存中，而是将 AFX_DATACACHE_ENTRY 中的 STGMEDIUM 清空。这就是为什么 COleDataSource::GetData()知道是直接从缓存中取得数据，还是调用其中一个 COleDataSource::OnRenderData()函数。

正如你所见，何时提供数据是在开始就确定的。CacheData()和 DelayRenderData()都将一个 AFX_DATACACHE_ENTRY 结构放入 COleDataSource 中。但是，CacheData()存储 STGMEDIUM 结构和 FORMATETC 结构，而 DelayRenderData()只存储 FORMATETC 结构。

COleDataSource 通过上面的机制来支持 IDataObject::GetData()。

响应 GetData()函数

COleDataSource 是一个实现 IDataObject 接口的完全的 COM 类。当客户需要从剪贴板上

获取数据时，COleDataSource 的 IDataObject 接口必须实现 IDataObject::GetData()函数。

COleDataSource 的实现是很直观的。如果你寻找 MFC 源代码，就会发现有一个文件叫 OLEDOBJ2.CPP。在这个文件中，可以找到 COleDataSource 的 IDataObject::GetData()函数实现的源代码。当然，COleDataObject 实现这个接口的机制跟实现别的 MFC COM 对象的机制完全一样：它们都使用接口映射表来实现。真正实现的函数是 COleDataSource::XdataObject::GetData()。这个函数会遍历 COleDataSource 的数据缓存来寻找跟请求格式匹配的条目。如果数据缓存在 STGMEDIUM 域中有需要的数据，则 GetData()调用 MFC 辅助函数_AfxCopyStgMedium()来将数据返回给调用者。

_AfxCopyStgMedium()是一个很长的函数（大约 160 行）。它跟 API 函数 ReleaseStgMedium()很类似。在_AfxCopyStgMedium()中有一个很大的 switch 语句，这个 switch 语句作用于拷贝使用的介质类型。例如，如果拷贝使用的介质类型是 HGLOBAL，则_AfxCopyStgMedium()会使用 API 函数 CopyGlobalMemory()来拷贝数据。

如果缓存中的条目使用的是延迟供应，则数据缓存条目的 STGMEDIUM 结构就是空的。在这种情况下，GetData()会调用 COleDataSource::OnRenderData()，而数据源就必须开始提供数据。

IDataObject 接口的其他函数

GetDataHere()

IDataObject::GetDataHere()和 IDataObject::GetData()很类似，除了调用者提供的是介质而不是对象。COleSource 的实现使用 Lookup()函数在缓存中查找相应的条目。如果 Lookup()找到一个条目，则 GetDataHere()使用_AfxCopyStgMedium()将数据从缓存条目中拷贝到调用者提供的 STGMEDIUM 中。

QueryGetData()

QueryGetData()比较简单：它只是调用 COleDataSource::Lookup()来查询一下请求的格式是否在缓存中。

GetCanonicalFormatEtc()

MFC 的 COleDataSource 并没有一个规范的 FORMATETC。因为 MFC 要对服务器元文件格式提供目标设备（ptd）支持，所以 FORMATETC 中的每一个成员都是很重要的。

SetData()

SetData()在缓存中寻找与调用者提供的 FORMATETC 格式匹配的条目。如果能找到，则 SetData()调用虚函数 OnSetData()。MFC 对 OnSetData()的默认实现是不做任何事情。所以为了使 COleDataSource 的 SetData()函数有效，必须自己重载 OnSetData()。

EnumFormatEtc()

如果客户需要了解一个数据对象提供的所有格式，则可以调用 EnumFormatEtc()函数。为了支持这个函数，COleDataSource 在它的缓存中对所有条目提供了一个枚举器。

COM 定义了几个枚举器接口，包括 IEnumVARIANT、IEnumUnknown 和 IEnumFORMATETC。枚举一个集合的语法几乎都是一样的，除了它们包含的元素不一样以外。所以每个 COM 枚举器都使用同样的形式。也就是说，每个枚举器都包含 4 个函数：Next()、Skip()、Reset()和 Clone()。MFC 在 OLEIMPL2.H 中定义了一个辅助类，这个类定义了一个范化的枚举类叫 CEnumArray。CEnumArray 是一个 COM 对象，它实现了一个接口叫 IEnumVOID。

IEnumVOID 并不是标准的 COM 接口，它在 OLEIMPL.H 中定义。IEnumVOID 枚举了一个 void 指针集合。CEnumArray 中保存着任意一种类型的集合，并且提供枚举这个集合的所有必要的功能。

在 OLEDOBJ2.CPP 中，MFC 定义了一个叫 CEnumFormatEtc 的类，这个类从 CEnumArray 派生。CEnumFormatEtc 为 COleDataSource 的数据缓存实现了一个枚举器。COleDataSource 使用 CEnumFormatEtc 来实现 EnumFormatEtc()。EnumFormatEtc() 首先声明一个 CEnumFormatEtc 的实例，然后遍历数据缓存（也就是 AFX_DATACACHE_ENTRY 结构数组），然后将每一个 FORMATETC 结构加入到 CEnumFormatEtc 中保存的链表中去。最后，COleDataSource::EnumFormatEtc()将枚举器的接口指针返回给客户。

DAdvise()、DUnadvise()以及 EnumDAdvise()

这 3 个函数没有什么可说的。COleDataSource 是用来控制剪贴板和拖放传输的。每一个 advise-loop 函数都返回 OLE_E_ADVISENOTSUPPORTED。

COleDataObject

COleDataObject 通常用在数据传输的接收端。虽然 COleDataSource 类确实实现了 IDataObject 接口，但是它只不过是对一个 IDataObject 指针的包装而已。程序清单 12-2 是在 AFXOLE.H 中定义的 COleDataObject。

程序清单 12-2 *COleDataObject* 类

```
class COleDataObject {
// Constructors
public:
   COleDataObject();

// Operations
   void Attach(LPDATAOBJECT lpDataObject, BOOL bAutoRelease = TRUE);
   LPDATAOBJECT Detach();  // detach and get ownership of m_lpDataObject
   void Release(); // detach and Release ownership of m_lpDataObject
   BOOL AttachClipboard(); // attach to current Clipboard object

// Attributes
   void BeginEnumFormats();
   BOOL GetNextFormat(LPFORMATETC lpFormatEtc);
   CFile* GetFileData(CLIPFORMAT cfFormat, LPFORMATETC lpFormatEtc = NULL);
   HGLOBAL GetGlobalData(CLIPFORMAT cfFormat, LPFORMATETC lpFormatEtc =
      NULL);
   BOOL GetData(CLIPFORMAT cfFormat, LPSTGMEDIUM lpStgMedium,
      LPFORMATETC lpFormatEtc = NULL);
   BOOL IsDataAvailable(CLIPFORMAT cfFormat, LPFORMATETC lpFormatEtc = NULL);

// Implementation
public:
   LPDATAOBJECT m_lpDataObject;

   LPENUMFORMATETC m_lpEnumerator;
   ~COleDataObject();

   // advanced use and implementation
   LPDATAOBJECT GetIDataObject(BOOL bAddRef);
```

```
    void EnsureClipboardObject();
    BOOL m_bClipboard;      // TRUE if represents the Win32 Clipboard

protected:
    BOOL m_bAutoRelease;    // TRUE if destructor should call Release

private:
    // Disable the copy constructor and assignment by default so you will get
    //   compiler errors instead of unexpected behaviour if you pass objects
    //   by value or assign objects.
    COleDataObject(const COleDataObject&);  // no implementation
    void operator=(const COleDataObject&);  // no implementation
};
```

下面是在 COleDataObject 中最重要的函数：Attach()、AttachClipboard()、BeginEnumFormats/GetNextFormat()和 GetData...()。

Attach()和 Detach()函数

前面已经提到过，COleDataObject 只是一个已经存在的 IDataObject 接口指针的包装。在上面的代码中，我们可以看到，COleDataObject 中有一个成员叫 m_lpDataObject，它是 IDataObject 类型的。跟别的包装本地事物（比如句柄）的 MFC 类一样，COleDataObject 也专门有一个函数用来将一个存在的接口指针连接到它。这个函数叫 Attach()。Attach()有两个参数：一个指向 IDataObject 接口的指针和一个标志，这个标志用来表示在 COleDataObject 对象被释放时，这个接口指针是否需要释放。COleDataObject::Attach()先释放 m_lpDataObject（如果它有效并且 m_bAutoRelease 值为 TRUE），然后将 m_lpDataObject 和 m_bAutoRelease 的值设为参数所传进来的值。

跟 Attach()函数功能相反的还有一个函数叫 Detach()。这个函数将 m_lpDataObject 设成 NULL。COleDataObject::Release()函数会释放 IDataObject 接口指针（也就是 m_lpDataObject）——如果 m_bAutoRelease 是 TRUE 的话。

AttachClipboard()和 EnsureClipboardData()函数

微软提供的文档中要求，如果想从剪贴板中获取数据，就必须调用 COleDataObject::AttachClipboard()函数。AttachClipboard()函数很简单，它仅仅是将 COleDataObject 的 m_bClipboard 成员设为 TRUE。真正的操作都是在 COleDataObject::EnsureClipboardObject()中进行。

EnsureClipboardObject()是对 OleGetClipboard() API 函数的包装。如果 m_bClipboard 标志为 TRUE（在 AttachClipboard()中设定），则 OleGetClipboard()将 OLE 剪贴板的 IDataObject 接口指针复制一份，并且使用 COleDataObject 中的 MFC 类成员函数包装这个接口指针。

MFC 按照上面这种方式访问剪贴板，并将它作为最优解决方法。接下来你会看到为什么在那个时候检查 COleDataObject::IsDataAvailable()。

确定可用数据

COleDataObject 有一个成员函数叫 IsDataAvailable()。这个函数有两个参数，一个是 CLIPFORMAT 结构，还有一个是 FORMATETC 类型。IsDataAvailable()会去检查 m_bClipboard 标志。如果 m_bClipboard 为 FALSE，则 IsDataAvailable()调用 IDataObject 接口的 QueryGetData()

函数，查询在给定的格式中数据是否可用。

如果 m_bClipboard 为 TRUE，则在剪贴板上可能会有数据（也就是说，用户已经调用了 AttachClipboard()）。IsDataAvailable()只是调用标准的 Windows API 函数::IsClipboardFormatAvailable()，这个函数一般比调用 IDataObject::QueryGetData()要有效和可靠。

注意一下虽然 IsDataAvailable()使用了 CLIPFORMAT 和 FORMATETC 结构作为参数，但是，如果客户已经调用了 AttachClipboard()，则 IsDataAvailable()会调用一个 Win32 的 API 函数。这个函数只接受一个 CLIPFORMAT 结构类型作为参数。那么，如果客户需要使用 FORMATETC 结构来描述的数据，这个 API 函数该如何处理呢？但是，这个函数确实不知道如何处理 FORMATETC 结构。OLE API 除了 CLIPFORMAT 以外，忽略其它任何信息。也许当 Windows API 在 OLE 的顶层实现，而不是以别的方式的时候，这种情况可能会改变。

枚举格式

前面提到过 IDataObject 有一个成员函数叫做 EnumFormatEtc()。EnumFormatEtc()返回一个指向接口的指针，这个接口能够枚举剪贴板的格式。COleDataObject::BeginEnumFormats()通过从 IDataObject 接口中取得枚举接口指针开始进行枚举。要取得真正的格式，客户需要调用 COleDataObject::GetNextFormat()。GetNextFormat()使用 COleDataObject::BeginEnumFormats()返回的枚举器，并且调用枚举器的 Next()函数。

获取数据

GetData()会调用 EnsureClipboardData()来从剪贴板中取得 IDataObject 接口指针，然后通过这个指针调用 IDataObject 的 GetData()函数来获取数据。

COleDataObject 还有两个 GetData...()类型的函数，分别是 GetGlobalData()和 GetFileData()。这两个函数的工作方式和 GetData()类似，不过 GetGlobalData()是取得一个全局内存句柄，而 GetFileData()是取得一个文件句柄。

正如以上你所见的，MFC 通过使用 IDataObject 类（COleDataSource 和 COleDataObject）将剪贴板的数据传输包装得非常好。这两个类在 OLE 拖放中也很有用。而事实上，拖放操作与剪贴板传输其实非常类似，仅仅需要多做一些工作而已。

OLE 拖放

与 OLE 剪贴板类似，OLE 拖放操作也在数据源和客户之间使用了 IDataObject 接口来实现。在数据源这一端必须实现 IDataObject 接口，而在客户端就必须知道如何使用 IDataObject 接口指针。

除了 IDataObject，为了实现 OLE 拖放操作，MFC 还使用另外两个接口：IDropSource 和 IDropTarget。下面就是 IDropSource 接口。

IDropSource 接口

IDropSource 处理在拖放操作中的用户接口。数据产生者的工作就是实现这个接口。下面是这个接口的定义：

```
interface IDropSource : public IUnknown {
public:
  virtual HRESULT QueryContinueDrag(BOOL fEscapePressed,
                                    DWORD grfKeyState) = 0;
  virtual HRESULT GiveFeedback(DWORD dwEffect) = 0;
};
```

- QueryContinueDrag()用来查询是否继续或是取消一个拖放操作。
- GiveFeedback()根据释放标志来改变光标。标志可能是 DROPEFFECT_NONE、DROPEFFECT_MOVE、DROPEFFECT_COPY 或者别的值。

IDropTarget 接口

由它的名字就可以看出来，这个接口用来处理拖放操作的目标端。需要释放的 COM 对象需要实现 IDropTarget 接口。有一个叫做 RegisterDragDrop()的 OLE API 函数将一个窗口句柄和一个实现了 IDropTarget 的 COM 对象关联起来。如果一个放下的目标被注册了，则当鼠标拖动操作经过 RegisterDragDrop 描述的窗口时，会通过 IDropTarget 的 4 个成员函数的其中一个来通知这个窗口。IDropTarget 接口的定义如程序清单 12-3 所示。

程序清单 12-3 IDropTarget 接口

```
interface IDropTarget : public IUnknown {
public:
   virtual HRESULT DragEnter(IDataObject *pDataObj,
                             DWORD grfKeyState,
                             POINTL pt,
                             DWORD *pdwEffect) = 0;

   virtual HRESULT DragOver(DWORD grfKeyState,
                            POINTL pt,
                            DWORD *pdwEffect) = 0;

   virtual HRESULT DragLeave( void);

   virtual HRESULT Drop(IDataObject *pDataObj,
                        DWORD grfKeyState,
                        POINTL pt,
                        DWORD *pdwEffect) = 0;

};
```

- 当在拖放操作中一个鼠标的光标进入窗口时，OLE 会调用 IDropTarget::DragEnter()。
- 当一个表示拖放操作的鼠标光标在这个窗口上面移动多次时，OLE 会调用 IDropTarget::DragOver()。
- 当表示拖放操作的鼠标光标鼠标离开窗口时，OLE 会调用 IDropTarget::DragLeave()。
- 当鼠标的键在一个放置的目标窗口（drop target window）上释放时，OLE 会调用 IDropTarget::Drop()。

OLE 还有一个 API 函数叫 DoDragDrop()，这个函数被用来启动拖放操作。一旦被调用，DoDragDrop()就进入一个模式循环，监控鼠标和键盘的变化（特别是 ESCAPE 键，用来取消它），直到用户通过释放鼠标或是按下 Escape 键来完成拖放操作为止。

MFC 提供了一个比较方便的方式来实现拖放传输：只要使用 COleDataSource、COleDataObject、COleDropTarget 以及 COleDropSource 即可。

使用 MFC 实现拖放数据传输

使用拖放操作比剪贴板传输更加简单快捷。由于拖放数据传输是交互式的，所以它的要求和剪贴板数据传输是不同的。使用拖放的基本思想是：先按下鼠标左键选中你所需要的数据，然后按住鼠标，将鼠标光标移动（拖动数据）到另外一个窗口（这个窗口用来保存新的数据）。当鼠标光标移动到需要数据的窗口上时，释放鼠标左键，数据就被放到这个窗口中去了。

客户还可以通过使用一个或多个控制键来选择传输的类型，如表 12-2 所示。

表 12-2 传输类型与响应的键

按下的键	传输类型
没有控制键	移动数据
CONTROL	拷贝数据
CONTROL-SHIFT	建立快捷方式
ALT	移动数据

跟剪贴板数据传输一样，拖放传输也需要一个数据源和一个数据目的地——一个数据产生者和一个数据消耗者。在剪贴板和拖放中，数据源这一端都是使用 COleDataSource 类来发起数据传输，在客户端使用 COleDataObject 类来接收数据。实际上，从剪贴板上获取数据和从拖放操作中接收数据的代码通常是一样的。很多时候可以直接使用剪贴板的粘贴操作代码来实现从拖放操作中接收数据。然而，剪贴板数据传输和拖放数据传输还是有一些细微差别的。

发起一个拖放传输

拖放数据传输通常在使用者按下鼠标左键的时候开始。所以要初始化一个拖放传输，只要处理 WM_LBUTTONDOWN 消息即可。

数据产生端的操作和剪贴板也是很类似的：使用 CacheData()或 CacheGlobalData()将数据拷贝到一个 COleDataSource 对象中去。差别是，拖放操作不是调用 COleDataSource::SetClipboard()将数据放到剪贴板上，而是调用 COleDataSource::DoDragDrop()，这个函数会自动处理数据传输。下面的代码片断展示了如何初始化一个拖放操作：

```
COleDataSource cods;

LPCSTR source = GetString();
HGLOBAL h =
  GlobalAlloc(GHND | GMEM_SHARE, strlen(source) + 1);
strcpy(LPSTR(GlobalLock(h)), source);
```

共有 5 个不同的 DROPEFFECT...值，分别用来表示以下的拖放传输状态：

- DROPEFFECT_NONE——释放不允许。
- DROPEFFECT_COPY——正在做拷贝操作。
- DROPEFFECT_MOVE——正在做移动操作。
- DROPEFFECT_LINK——建立一个从释放的数据到原始数据的一个链接。
- DROPEFFECT_SCROLL—— 一个拖动/滚动操作将会发生或是正在目标端发生。

OnDragEnter()和 OnDragOver()可以使用 COleDataObject 中的信息、按键的屏蔽码以及 CPoint 参数来确定执行什么样的操作。例如，如果放置的目标只能接受 CF_BITMAP 格式的数据，而剪贴板中只有 CF_TEXT 格式的数据（可以通过查询 COleDataObject 得到），则视图不能接受这个数据。在这个例子中，OnDragEnter()和 OnDragOver()将返回 DROPEFFECT_NONE 来表示视图不能接受这个数据。通过返回恰当的 DROPEFFECT...的值，这些函数就可以通知框架需要显示什么样的光标。

另外，OnDragEnter()和 OnDragOver()函数能够查询按键的屏蔽码，从而能够发现什么控制键被按下了。需要改变光标的形状时，放置的目标只要返回相应的 DROPEFFECT...值，然后剩下的工作就由 MFC 来做了。

当鼠标光标离开视图时，这个视图的 OnDragLeave()函数就会被调用。这个函数没有参数，并且返回值的类型为 void。这个函数是放置拖放操作的清除代码的好地方。

最后一个函数是 OnDrop()。当客户在一个视图中释放鼠标左键，而这个视图是放置的目标的时候，就会调用这个函数。OnDrop()的参数和 OnDragEnter()以及 OnDragOver()一样：一个指向 COleDataObject 指针；一个 DWORD 类型，表示控制键的状态；一个 CPoint 对象，这个对象用来表示鼠标的位置。OnDrop()也返回一个 DROPEFFECT...值，用来表示实际发生的传输操作的类型。这个值会返回给 COleDataSource::DoDragDrop()。放置数据的代码和从剪贴板中粘贴数据的代码看起来是一样的。很多开发者对这两者只实现一份代码。

MFC 中实现拖放的类

总的来说，MFC 的拖放数据传输是剪贴板传输的一个扩充。与剪贴板传输一样，拖放传输也是从 COleDataSource 开始的。数据产生者可以缓存数据，也可以承诺以后提供数据（使用 DelayRenderData()）。然而，在拖放中不是调用 SetClipboard()而是调用 DoDragDrop()。在客户这一端，应用程序会注册一个带 COleDropTarget 的视图。然后这个视图接收拖放操作的消息。下面我们详细地讨论一下 MFC 中与拖放传输相关的类。

COleDataSource 类

除了包含一个数据缓存和对剪贴板提供支持外，COleDataSource 还有一个函数叫做 DoDragDrop()。这个 DoDragDrop()封装了 OLE 函数 DoDragDrop()。

```
DROPEFFECT DoDragDrop(
      DWORD dwEffects = DROPEFFECT_COPY|DROPEFFECT_MOVE|DROPEFFECT_LINK,
      LPCRECT lpRectStartDrag = NULL,
      COleDropSource* pDropSource = NULL);
```

COleDataSource::DoDragDrop()有 3 个参数：（1） 一个 DROPEFFECT 结构，用来表示什

么类型的拖放操作被允许；（2）一个矩形，用来表示拖放操作从何时开始；（3）一个指向 COleDropSource 的指针，用来处理拖放操作中用户界面方面的事务。如果最后一个参数为 NULL，则 DoDragDrop()使用由 MFC 提供的标准的 COleDropSource 类。

COleDropSource 类

COleDropSource 类实现了 IDropSource 接口。程序清单 12-4 显示了 COleDropSource 的定义。

程序清单 12-4 COleDropSource 类

```
class COleDropSource : public CCmdTarget {
// Constructors
public:
   COleDropSource();

// Overridables
   virtual SCODE QueryContinueDrag(BOOL bEscapePressed, DWORD dwKeyState);
   virtual SCODE GiveFeedback(DROPEFFECT dropEffect);
   virtual BOOL OnBeginDrag(CWnd* pWnd);

public:
   BEGIN_INTERFACE_PART(DropSource, IDropSource)
      INIT_INTERFACE_PART(COleDropSource, DropSource)
      STDMETHOD(QueryContinueDrag)(BOOL, DWORD);
      STDMETHOD(GiveFeedback)(DWORD);
   END_INTERFACE_PART(DropSource)

   DECLARE_INTERFACE_MAP()

   CRect m_rectStartDrag;
   BOOL m_bDragStarted;
   DWORD m_dwButtonCancel;
   DWORD m_dwButtonDrop;

   // metrics for drag start determination
   static AFX_DATA UINT nDragMinDist;
   static AFX_DATA UINT nDragDelay;

   friend class COleDataSource;
};
```

由上面的代码可知，COleDropSource 通过标准的 MFC 接口映射表来实现 IDropSource 接口。IDropSource::QueryContinueDrag() 和 IDropSource::GiveFeedback() 直接映射到 COleDropSource::QueryContinueDrag()和 COleDropSource::GiveFeedback()这两个虚函数。由于这两个函数是虚函数，所以尽可以放心大胆地去重载它。

在很多情况下，都需要重载 MFC 提供的 COleDropSource 的默认实现。例如，如果你不喜欢 MFC 提供的光标的形状，则可以从 COleDropSource 派生出一个子类来，然后重载 GiveFeedback()函数。另外 COleDropSource 还包含了一个用来标识取消拖放操作的变量（m_dwButtonCancel）和一个用来标识放置操作的鼠标按钮的变量（m_dwButtonDrop）。

COleDropSource 还有个静态变量来描述在拖放操作开始前的时间延迟（m_nDragDelay），

还有在拖放操作开始前鼠标必须移动的像素数。COleDropSource 的构造函数会初始化这些变量。拖动距离和拖动延迟变量从 WIN.INI 文件中读取。拖动距离的键是“DragMinDist”，而拖动延迟的键是“DragDelay”。如果这些条目在初始化文件中不存在，则拖动距离被初始化为 2，而拖动延迟被设置为 200 毫秒。

如果想为给定的拖放操作改变用户界面的特征，则可以从 COleDropSource 派生出一个自己的类，并且将这个类作为参数传递给 COleDataSource::DoDragDrop()函数。

COleDropTarget 类

COleDropTarget 实现了 IDropTarget 接口。COleDropTarget 也是一个 COM 类，它使用了 MFC 的接口映射表来实现接口，程序清单 12-5 是它的定义。

程序清单 12-5 COleDropTarget 类

```
class COleDropTarget : public CCmdTarget {
// Constructors
public:
   COleDropTarget();

// Operations
   BOOL Register(CWnd* pWnd);
   virtual void Revoke();  // virtual for implementation

// Overridables
   virtual DROPEFFECT OnDragEnter(CWnd* pWnd, COleDataObject* pDataObject,
      DWORD dwKeyState, CPoint point);
   virtual DROPEFFECT OnDragOver(CWnd* pWnd, COleDataObject* pDataObject,
      DWORD dwKeyState, CPoint point);
   virtual BOOL OnDrop(CWnd* pWnd, COleDataObject* pDataObject,
      DROPEFFECT dropEffect, CPoint point);
   virtual DROPEFFECT OnDropEx(CWnd* pWnd, COleDataObject* pDataObject,
      DROPEFFECT dropDefault, DROPEFFECT dropList, CPoint point);
   virtual void OnDragLeave(CWnd* pWnd);
   virtual DROPEFFECT OnDragScroll(CWnd* pWnd, DWORD dwKeyState,
      CPoint point);

// Implementation
public:
   virtual ~COleDropTarget();

protected:
   HWND m_hWnd;            // HWND this IDropTarget is attached to

 LPDATAOBJECT m_lpDataObject; // != NULL between OnDragEnter, OnDragLeave
 UINT m_nTimerID;         // != MAKEWORD(-1, -1) when in scroll area
 DWORD m_dwLastTick;      // only valid when m_nTimerID valid
 UINT m_nScrollDelay;     // time to next scroll

 // metrics for drag-scrolling
 static AFX_DATA int nScrollInset;
 static AFX_DATA UINT nScrollDelay;
 static AFX_DATA UINT nScrollInterval;
```

```
    // implementation helpers
    void SetupTimer(CView* pView, UINT nTimerID);
    void CancelTimer(CWnd* pWnd);

// Interface Maps
public:
    BEGIN_INTERFACE_PART(DropTarget, IDropTarget)
        INIT_INTERFACE_PART(COleDropTarget, DropTarget)
        STDMETHOD(DragEnter)(LPDATAOBJECT, DWORD, POINTL, LPDWORD);
        STDMETHOD(DragOver)(DWORD, POINTL, LPDWORD);
        STDMETHOD(DragLeave)();
        STDMETHOD(Drop)(LPDATAOBJECT, DWORD, POINTL pt, LPDWORD);
    END_INTERFACE_PART(DropTarget)

    DECLARE_INTERFACE_MAP()
};
```

COleDropTarget 中最重要的成员函数是 Register()、DragEnter()、DragOver()、DragLeave()以及 Drop()。

在 MFC 的拖放实现中另外一个很重要的类是 CView。COleDropTarget 能够和 CView 很好地协调工作，所以使得对 OLE 的拖放编程变得非常简单。

CView 类

也许你还记得 CView 也有成员函数叫 OnDragEnter()、OnDragLeave()、OnDragOver()和 OnDrop()。但是，这些函数一般是在 IDropTarget 中处理的，它们在 CView 中有何作用的呢？这些函数是如何被调用的呢？下面我们对 MFC 的拖放操作从头到尾看一遍，看看到底都发生了些什么。

MFC 拖放的工作过程

详细了解 MFC 如何实现 OLE 拖放的最好方法是分别检查两端：数据产生者和数据消耗者。在产生者这一端，有 COleDataSource 和 COleDropSource 两个类，而在数据消耗者这一端，也有 COleDataObject 和 COleDropTarget 两个类。让我们先来看看数据产生者这一端。

数据产生者

要成为一个拖动源，数据产生者或者缓存数据，或者使用 DelayRenderData()来建立延迟供应，然后调用 DoDragDrop()。

COleDataSource::DoDragDrop()在表面上看来是对 API 函数::DoDragDrop()的封装，但是，它却不仅仅是对::DoDragDrop()的简单封装。

当数据产生者调用 COleDataSource::DoDragDrop()时，MFC 会去检查调用传进来的 COleDropSource 参数。如果调用者提供了一个 COleDropSource，则 DoDragDrop()就使用这个 COleDropSource 来管理用户界面（即改变光标的形状和判断何时拖放操作可以继续，或是被取消）。如果调用者没有提供 COleDropSource，则 DoDragDrop()使用 MFC 提供的 COleDropSource 的默认实现。

如果调用者提供矩形参数，则 DoDragDrop()将这个矩形拷贝到自己的成员中（叫

m_rectStartDrag)。COleDataSource 使用这矩形来判断什么时候开始拖动，也就是说，真正的拖动直到光标离开这个矩形才开始发生。如果调用者没有提供矩形参数（即为 NULL），则 DoDragDrop()会在光标的当前位置周围创建一个它自己的空矩形。

在开始拖动操作之前，COleDataSource::DoDragDrop()会调用 OnBeingDrag()函数。这个函数是个虚函数，所以你在需要的时候可以去重载它。OnBeginDrag()的默认实现是：先捕捉到鼠标数据，然后建立一个循环，作为计时器，然后在时间耗尽后结束这个函数。

接着，COleDataSource::DoDragDrop()使用 GetInterface()来从传进来的 COleDropSource 对象中取得 IDataObject 和 IDropSource 接口。COleDataSource::DoDragDrop()使用这两个接口指针来调用 API 函数::DoDragDrop()。

MFC 以前的版本直接调用 DoDragDrop()，而现在，MFC 包含一个函数指针的队列，这些函数都具有 OLE 特性。例如，下面就是 MFC 用来调用 DoDragDrop()的代码片断：

```
#ifdef _AFXDLL
        _afxOLE.pfnDoDragDrop[0](lpDataObject, lpDropSource, dwEffects,
                                    &dwResultEffect);
#else
    ::DoDragDrop(lpDataObject, lpDropSource, dwEffects, &dwResultEffect);
#endif
```

在 MFC 4 中，微软将所有的分散的 MFC DLL 都合成到一个 DLL 中。如果一个应用程序直接链接到某个 DLL（比如 OLE32.DLL），就算是个很小的应用，这个应用程序也需要装载 OLE。所以，微软使用了一种很好的技术来动态加载所有不重要的 DLL。除了某些不可能的情况外，所有的这些实例都被宏隐藏了。MFC40.DLL 只是直接绑定到 KERNEL32.DLL、GDI32.DLL、USER32.DLL 和 MSVCRT20.DLL。另外的所有 DLL 都是当其第一个函数被调用时才在运行时动态加载进来。OLE32.DLL、OLEAUT32.DLL、SHELL32.DLL、ADVAPI32.DLL、COMCTL32.DLL、ODBC32.DLL 和 WINSPOOL.DLL 都是 DLL 的例子，这些 DLL 都是在运行时被动态加载的。这样 MFC 就维护了一个庞大的 DLL，但是对于那些并不使用全部 DLL 的应用程序而言，又不影响它们的性能。

数据消耗者

与剪贴板数据传输一样，拖放操作也是在 COleDataObject 中结束的。然而，在拖放操作中不是使用 OleGetClipboard()从剪贴板中取得一个 IDataObject 指针，而是当一个数据被放置到视图中时，MFC 会提供一个完整的 COleDataObject。

在 OLE 中注册

为了得到拖放操作的通知，视图必须先将自己注册为一个放置目标。也就是说，这个视图必须声明一个 COleDropTarget 的实例，然后调用 Register()函数。调用 Register()的标准位置是在视图的 OnCreate()函数中，因为这个函数是能对窗口做任何初始化的第一处地方。下面的代码片断显示了如何注册：

```
class CMyView: public CView {
...
   COleDropTarget m_dt;
...
};
```

```
CMyView::OnCreate() {
   m_dt.Register(this);
}
```

那么，Register()函数都做了哪些工作呢？在做别的工作之前，Register()会调用CoLockExternalObject()来给放置目标对象设置一个强锁（strong lock）。通过这个方法，放置目标对象就可以一直保留在内存中，直到客户调用的Release()次数等于AddRef()次数为止。如果CoLockExternalObject()没有被调用，则拖放操作只能工作一次，以后就每次都失败了。

接着，Register()调用OLE的API函数RegisterDragDrop()。这个API函数有两个参数：接收拖放消息的窗口的句柄；一个IDropTarget接口指针。在这些信息通过RegisterDragDrop()函数在OLE注册后，当鼠标光标移进或移过由窗口句柄指定的窗口时，OLE就可以调用IDropTarget的函数了（这些函数是OnDragEnter()、OnDragOver()、OnDragLeave()和OnDrop()）。在返回之前，COleDropTarget::Register()会保存传进来的CWnd对象和窗口句柄。COleDropTarget使用CWnd对象来实现OnDragEnter()、OnDragOver()、OnDragLeave()和OnDrop()。它还需要使用窗口句柄来实现它的Revoke()成员函数。

最后，CWnd会保留一个指向COleDropTarget的成员。除了保存CWnd对象和窗口句柄外，COleDropTarget::Register()将CWnd的COleDropTarget成员设置为COleDropTarget的this指针。在窗口被析构时，CWnd使用这个指针来取消拖放的注册。

取消拖放注册

COleDropTarget中有一个补充函数用来取消一个窗口的拖放注册。这个函数叫做Revoke()。COleDropTarget::Revoke()调用OLE API函数RevokeDragDrop()，并且释放加在COleDropTarget上的锁。开发者一般很少自己调用这个函数，如果CWnd有指向COleDropTarget的指针，则它会在OnNcDestroy()函数中调用COleDropTarget::Revoke()。

处理消息

如果一个窗口在OLE中注册，它就可以接收到拖放消息。鼠标光标一进入窗口的边界，这个窗口就可以接收到这些消息。在MFC中，拖放消息的处理在CView这个类中完成。如果一个窗口在OLE中被注册为一个放置目标，则就算是传进去一个空的CWnd，它的CView中也已经包含了对OnDragEnter()、OnDragOver()、OnDragLeave()和OnDrop()这4个函数的处理。如果不是使用CView进行注册，而是使用其它的窗口，则MFC的拖放操作不能进行。

当鼠标经过一个被注册为接收拖放消息的窗口时，OLE就发送消息给这个窗口。拖动中第一个被调用的消息处理函数是OnDragEnter()。由于COleDropTarget实现了IDropTarget，所以如果窗口和IDropTarget指针在OLE中注册，则OLE会调用IDropTarget::DragEnter()函数。当OLE调用IDropTarget的DragEnter()函数时，OLE传递一个完整的IDataObject（由数据产生者提供）和窗口句柄（光标拖动经过的窗口）给这个函数。COleDropTarget的DragEnter()所做的工作是：将一个导入的IDataObject连接到一个COleDataObject，然后调用CWnd::FromHandle()查询窗口句柄。COleDropTarget的DragEnter()仅仅是将CWnd和COleDataObject作为参数传递给OnDragEnter()成员函数。在COleDropTarget中，OnDragEnter()检查连接到的窗口是否是一个视图。如果连接的窗口不是一个视图，则OnDragEnter()返回一个DROPEFFECT_NONE并退出。否则，COleDropTarget::OnDragEnter()试图将这个CWnd指针转换为CView指针。如果转换成功，则COleDropTarget::OnDragEnter()将操作委托给

CView::OnDragEnter()。也就是说，目标视图能检查数据，判断数据是否有效，并且相应地更新用户界面。

另外的 IDropTarget 函数也以类似方式工作。当 OLE 调用由 COleDropTarget 表示的 IDropTarget 接口函数时，COleDropTarget 得到 IDataObject 指针，将它连接到一个 COleDataObject 对象，找到用来表示窗口的 CWnd 对象，将它转换为视图，然后调用相应的 CView 的函数（即 OnDragOver()、OnDragLeave()或 OnDrop()）。

从上面我们可以看出，实现拖放的大部分工作是安全可靠的。而且，MFC 为我们实现了绝大部分的代码。

结　语

OLE 的 UDT 比想像的要简单——特别是当使用 MFC 对 IDataObject 的包装时。使用 MFC 对数据传输进行编程，可以避免陷到 OLE 的泥沼中去（底层的 API）。这一章的内容可以帮助读者很好地理解如何使用 UDT 来实现剪贴板和拖放数据传输。在下一章中，我们将介绍 OLE 文档。想要了解更多的信息，请运行 www.infopower.com.cn 上的 DNEDIT.EXE 例子。DNEDIT 显示了如何在记事本之类的文本编辑器下使用 UDT。

CHAPTER

13 使用 MFC 实现 OLE 文档

这一章我们从 OLE（Object Linking and Embedding, 对象链接与嵌入）技术开始。

Windows 的老用户应该都还记得在 1991 年末和 1992 年初的时候，微软宣布了一种 OLE 技术，它是对动态数据交换（dynamic data exchange）技术的扩展，当时引起了很大的轰动。微软花了很大的力气来演示如何将一个 Microsoft Excel 的电子数据表以及 Microsoft Paint 的图片嵌入到 Word 中去。然后用户只要双击在 Word 中的图片，Paintbrush 就会在另外一个窗口中被激活。在 20 世纪 90 年代初期，这种技术是很令人振奋的。

微软将这种新技术命名为 OLE（对象链接与嵌入），是因为用户可以从各种不同的地方链接与嵌入不同的对象到一个文档中。现在已经有了基于 COM 的新版本的 OLE，OLE 的原始版本就叫 OLE1。OLE1 是将不同类型的数据集成到一个文档中的一种很好的方法。为了从不同的数据源中得到一份文档，OLE1 处理输入输出文件，对打印输出的纸进行裁剪，然后用订书机将它装订成文档。然而，由于 OLE1 也只是 DDE 的一个新的包装，所以它几乎拥有 DDE 的所有缺点：它是异步的，处理速度有时比较慢，而且由于它的数据传输介质只有全局内存句柄，所以不够灵活。

现在，基于 COM 的 OLE（真正的 OLE）比 OLE1（基于 DDE）要灵活多了。正如前面几章所讲述的，OLE 和 COM 支持很多的特性。复合文档特性（现在叫 OLE 文档）是 OLE 定义的能被扩展的技术之一。这章主要就是介绍 OLE 文档以及 MFC 如何实现它。在这一章中，我们将回答以下几个问题：

- MFC 如何管理复合文件（compound file）？如何将 OLE 的结构化存储模型（structured storage model）集成到 MFC 的序列化（serialization）模型中？
- 当用户双击一个内部的对象时，将会发生什么？菜单变换是如何完成的？
- 如何将 OLE 的接口函数映射到 MFC 类的成员函数？

OLE 文档 101

如果你还没看过 OLE 文档运转的话，那就试试。在 Word 文档中嵌入一个 Excel 的电子表格。双击 Word 文档中这个电子表格，你会突然发现已经在 Excel 中了。点击电子表格外面的地方，你又会回到 Word 中。

OLE 文档的核心思想就是将不同的数据放在同一个地方。也就是说，实现 OLE 文档模

型的应用程序必须能够将它本身的数据类型和别处来的数据混合。很经典的一个例子就是将电子表格混合到 Word 文档中，然后文字处理器就将电子表格的数据和自己本身的数据一起处理。

OLE 文档模型还要求在我们这个例子中，通过双击在 Word 中的电子表格，能够编辑电子表格数据的内容。

尽管能够做到上面的功能对用户来说是很好的一件事情，但是对开发者来说，确实是一件工作量很大的任务。很多的细节都需要考虑。例如，电子表格和 Word 是完全不同的两种类型数据，如何将这两个种数据混合到一个文件中去？另外，如何从一个应用程序转换到另外一个应用程序（当你双击电子表格的时候，出现 Excel 接口）？所以要实现这个功能，需要做很多的工作。

在我们深入了解 OLE 文档之前，先让我们学习一些术语和概念：链接（linking）、嵌入（embedding）、结构化存储模型（structured storage model）、定点激活（in-place activation）和可视化编辑（visual editing）。

链接与嵌入

在 OLE 中有两种方法可以将不同类型的数据集成到一个文档中，这两种技术就是：链接（linking）与嵌入（embedding）。链接的文档与嵌入的文档的区别在于数据存储的位置。

如果数据在 OLE 文档中是被链接的，则数据被存储在不同的文件中，OLE 文档包含指向数据的引用。而如果数据是嵌入到 OLE 文档中的，则 OLE 文档将嵌入的数据与它自身原有的数据保存到同一个文件中。

举个例子来说，你有一个 Excel 电子表格文件，现在将它插入到一个 Windows 的 Word 文档中，则你可以选择是采用链接还是嵌入方式来插入。如果选择链接方式，则数据还是保存在 Excel 文件中（在 Word 文档外面）。如果选择嵌入方式来插入这个 Excel 电子表格，则数据跟 Word 文档原有的数据一起被保存在 Word 文件中。

图 13-1 和图 13-2 显示了链接和嵌入之间的区别。

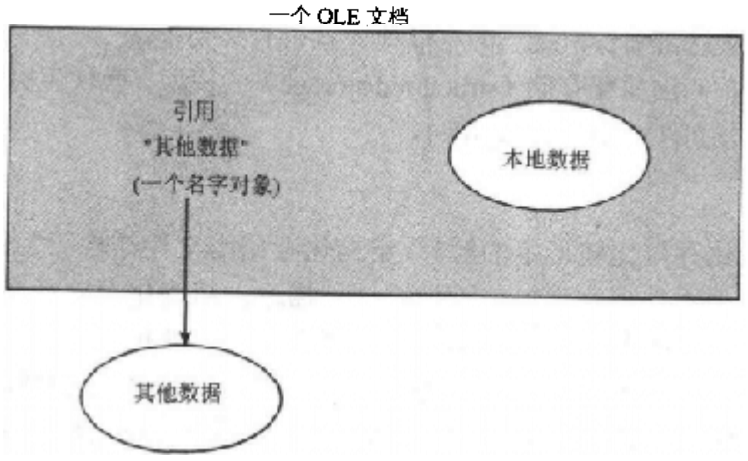


图 13-1 一个包含链接条目的 OLE 文档

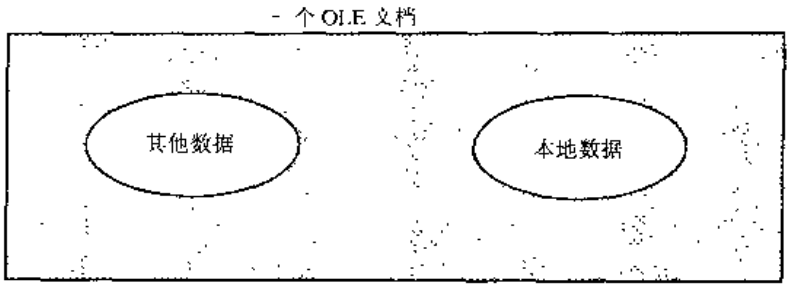


图 13-2 一个包含嵌入条目的 OLE 文档

链接与嵌入的比较

很明显，如果一个文档包含一个指向数据的链接（而不是一个单独的拷贝），则一般来说这个文档要比用嵌入的方式包含同样数据的文档要小。但是，链接数据的一个缺点是，如果移动链接所指向的源数据，则就会破坏这个链接。另外，如果想分发一个链接的 OLE 文档，则需要分发两部分：OLE 文档以及包含所链接数据的文件。如果是嵌入数据，则只需要分发 OLE 文档即可。因为嵌入的数据也是被存储到这个文件中，所以 OLE 文档的用户只需要一个文件就够了。

将许多不同类型的数据集成到一个单独的文档中又会产生另外一个问题，即如何管理存储在一个文件中的不同类型的数据？OLE 通过结构化存储模型来解决。

结构化存储、复合文件和永久对象

如果想在以前解决这个问题（在 OLE 之前），可能需要定义某种文件结构，用来包含头信息，这些信息指向包含真正数据的文件中不同的位置。而且以前确实是创建了类似的文件格式。

使用这种方式会有两个问题。首先，这种方式需要很繁重的编程工作。跟踪什么时候做了什么工作就需要很大的工作量，如果需要保持文件同步的话，那就更加麻烦了。其次，这种方式将文件格式与应用程序绑定了。其余任何应用程序需要访问这些数据的时候，都必须了解格式才可以。从这点来看，统一的存储模式就很有必要了。

OLE 提供了一种叫结构化存储（structured storage）的模型。微软还提供了一种结构化存储的实现，叫做复合文件（compound file）。

结构化存储

下面是对结构化存储比较形象的解释：结构化存储与文件结构的关系非常类似于操作系统与目录结构的关系。也许你以前听说过这种描述：结构化存储是文件中的文件系统。

与文件系统包含一个目录和文件的层次结构类似，一个 OLE 结构化存储文件也包含存储对象和流对象的类似的结构。在结构化存储模型中，存储对象跟目录类似，而流对象与文件类似。结构化存储将一个文件组织成各个存储对象，并且将真正的数据保存在流对象中。

图 13-3 是一个叫 DFVIEW.EXE 的程序运行后的截屏（DFVIEW 来自 Visual C++，是一个复合文件浏览器）。注意一下，根存储对象下挂了很多分开的子存储对象，而在子存储对象中又有流对象。这个图也许可以让你了解结构化存储是如何工作的。

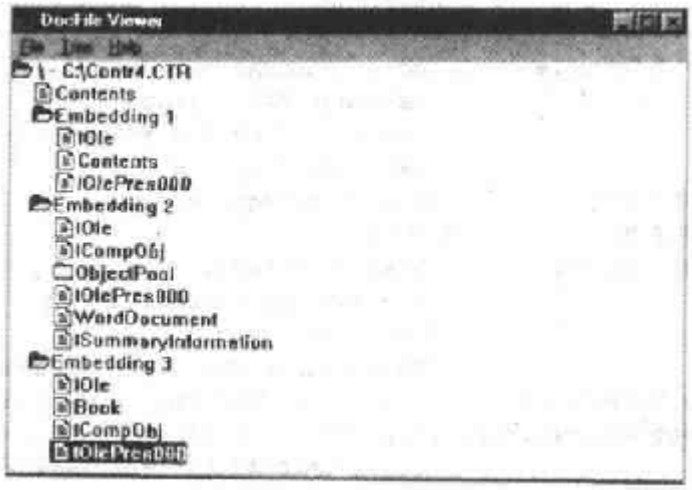


图 13-3 DFVIEW.EXE 展示了一个复合文档文件的结构

与别的 OLE 特性一样，结构化存储也使用了一些接口来实现。在结构化存储模型中，有 3 个特别的接口：IStorage、IStream 和 ILockBytes。IStorage 的函数用来管理存储组织，IStream 包含处理数据读写的函数，而 ILockBytes 用来处理抽象存储设备所必需的底层功能，使存储设备对客户来说看起来一个字节数组。

我们并不需要完全了解这些接口，也不需要知道 MFC 是如何实现 OLE 文档的。但是，知道每个接口的要点是很重要的。我们先来看看 IStorage 接口，如程序清单 13-1 所示。

程序清单 13-1 IStorage 接口

```
interface IStorage : public IUnknown {
public:
   virtual HRESULT CreateStream(const OLECHAR FAR *pwcsName,
                  DWORD grfMode,
                  DWORD reserved1,
                  DWORD reserved2,
                  IStream FAR *FAR *ppstm) = 0;
   virtual HRESULT OpenStream(const OLECHAR FAR *pwcsName,
                              void FAR *reserved1,
                              DWORD grfMode,
                              DWORD reserved2,
                              IStream FAR * FAR *ppstm) = 0;
   virtual HRESULT CreateStorage(const OLECHAR FAR *pwcsName,
                                 DWORD grfMode,
                                 DWORD dwStgFmt,
                                 DWORD reserved2,
                                 IStorage FAR *FAR *ppstg) = 0;
   virtual HRESULT OpenStorage(const OLECHAR FAR *pwcsName,
                               IStorage FAR *pstgPriority,
                               DWORD grfMode,
                               SNB snbExclude,
                               DWORD reserved,
                               IStorage FAR * FAR *ppstg) = 0;
   virtual HRESULT CopyTo(DWORD ciidExclude,
                          const IID FAR *rgiidExclude,
```

```
                                SNB snbExclude,
                                IStorage FAR *pstgDest) = 0;
    virtual HRESULT MoveElementTo(const OLECHAR FAR *pwcsName,
                                  IStorage FAR *pstgDest,
                                  const OLECHAR FAR *pwcsNewName,
                                  DWORD grfFlags) = 0;
    virtual HRESULT Commit(DWORD grfCommitFlags) = 0;
    virtual HRESULT Revert( void) = 0;
    virtual HRESULT EnumElements(DWORD reserved1,
                                 void FAR *reserved2,
                                 DWORD reserved3,
                                 IEnumSTATSTG FAR * FAR *ppenum) = 0;
    virtual HRESULT DestroyElement(const OLECHAR FAR *pwcsName) = 0;
    virtual HRESULT RenameElement(const OLECHAR FAR *pwcsOldName,
                                  const OLECHAR FAR *pwcsNewName) = 0;
    virtual HRESULT SetElementTimes(const OLECHAR FAR *pwcsName,
                                    const FILETIME FAR *pctime,
                                    const FILETIME FAR *patime,
                                    const FILETIME FAR *pmtime) = 0;
    virtual HRESULT SetClass(REFCLSID clsid) = 0;
    virtual HRESULT SetStateBits(DWORD grfStateBits,
                                 DWORD grfMask) = 0;
    virtual HRESULT Stat(STATSTG FAR *pstatstg,
                         DWORD grfStatFlag) = 0;
};
```

IStorage 和我们以前见过的接口很类似，它是一个纯虚基类，包含了一些需要实现的函数。在这里需要注意的是在这个接口中所包含的函数。所有的函数都是用来管理或组织存储的，这点在诸如 CreateStorage()和 CreateStream()等函数中体现的特别明显。

现在我们来看看 IStream 接口，如程序清单 13-2 所示。

程序清单 13-2 IStream 接口

```
interface IStream : public IUnknown {
public:
    virtual HRESULT Read(void FAR *pv,
                         ULONG cb,
                         ULONG FAR *pcbRead) = 0;
    virtual HRESULT Write(const void FAR *pv,
                          ULONG cb,
                          ULONG FAR *pcbWritten) = 0;
    virtual HRESULT  Seek(LARGE_INTEGER dlibMove,
                          DWORD dwOrigin,
                          ULARGE_INTEGER FAR *plibNewPosition) = 0;
    virtual HRESULT  SetSize(ULARGE_INTEGER libNewSize) = 0;
    virtual HRESULT  CopyTo(IStream FAR *pstm,
                            ULARGE_INTEGER cb,
                            ULARGE_INTEGER FAR *pcbRead,
                            ULARGE_INTEGER FAR *pcbWritten) = 0;
    virtual HRESULT Commit(DWORD grfCommitFlags) = 0;
    virtual HRESULT  Revert( void) = 0;
    virtual HRESULT  LockRegion(ULARGE_INTEGER libOffset,
                                ULARGE_INTEGER cb,
                                DWORD dwLockType) = 0;
```

```
    virtual HRESULT  UnlockRegion(ULARGE_INTEGER libOffset,
                                  ULARGE_INTEGER cb,
                                  DWORD dwLockType) = 0;
    virtual HRESULT  Stat(STATSTG FAR *pstatstg,
                          DWORD grfStatFlag) = 0;
    virtual HRESULT  Clone(IStream FAR *FAR *ppstm) = 0;
};
```

IStream 中的函数用来从设备中读取数据或是往设备中写数据。注意一下以下几个函数：Read()、Write()和 Seek()。IStream 中的函数与用来读写普通文件的运行时库中的函数很类似。

下面，让我们来看看 ILockBytes 接口，见程序清单 13-3。

程序清单 13-3 ILockBytes 接口

```
interface ILockBytes : public IUnknown {
public:
    virtual HRESULT  ReadAt(ULARGE_INTEGER ulOffset,
                            void FAR *pv,
                            ULONG cb,
                            ULONG FAR *pcbRead) = 0;
    virtual HRESULT  WriteAt(ULARGE_INTEGER ulOffset,
                             const void FAR *pv,
                             ULONG cb,
                             ULONG FAR *pcbWritten) = 0;
    virtual HRESULT  Flush( void) = 0;
    virtual HRESULT  SetSize(ULARGE_INTEGER cb) = 0;
    virtual HRESULT  LockRegion(ULARGE_INTEGER libOffset,
                                ULARGE_INTEGER cb,
                                DWORD dwLockType) = 0;
    virtual HRESULT  UnlockRegion(ULARGE_INTEGER libOffset,
                                  ULARGE_INTEGER cb,
                                  DWORD dwLockType) = 0;
    virtual HRESULT  Stat(STATSTG FAR *pstatstg,
                          DWORD grfStatFlag) = 0;
};
```

实现一个系统（比如结构化存储）就是提供一组接口，让它们能够运行在各种操作系统之上。这个就是为什么要引入 ILockBytes 的原因。接口 ILockBytes 中包含将存储设备看成一个字节数组所必需的函数。客户使用 ILockBytes 可以将存储介质看成是一个字节数组。下面让我们来考虑一下 DOS 的文件系统：文件分配表（File Allocation Table）系统。如果调用一个 C 运行时函数来操作数据，则我们并不需要了解控制磁盘的扇区（sector）和簇（cluster），操作系统会为我们做这个。同样的道理，实现 ILockBytes 接口的 OLE 对象控制了同样复杂的存储介质，所以 ILockBytes 客户可以认为是在一个字节数组上处理。

正如你所见，OLE 文档有 3 个很重要的接口。现在，最重要的一件事情是你知道存在这些接口，并且了解如何使用它们。IStorage 接口的函数用来管理存储，IStream 中的函数用来处理在流对象中的读写操作，而 ILockBytes 接口就类似于某种“设备驱动”，将存储设备抽象为一个字节数组。

复合文件

作为一个程序员，也许你并不需要实现这些接口，因为结构化存储模型毕竟是属于操作

系统的领域（如果有可能，我们尽量不去实现这些接口）。幸运的是，微软给我们提供了一个结构化存储的实现：复合文件（compound file）。

让我们来回忆一下在一个普通的程序中如何使用文件 I/O。如果需要创建一个文件，通常会调用运行时库的一个与文件操作相关的函数，比如_create()或_open()。这些操纵文件的函数使用文件句柄来表示文件。操作系统使用文件句柄来管理磁盘和文件 I/O。例如，在成功地调用了_create()或_open()后，会返回一个有效的文件句柄，后面的文件操作就可以利用这个句柄了。如果有了文件句柄，就可以对这个文件执行_lseek()、_read()和_write()等操作来访问这个文件的内容。

复合文件（微软对结构化存储的实现）跟上面的普通文件的操作方式有点不同。它不是调用运行时库的函数来创建或打开一个文件，而是调用一个全局的 API 函数。创建一个复合文件存储对象的最常用的函数是 StgCreateDocFile()。但是，这个函数并不产生一个文件句柄，而是返回一个接口指针。在这个例子中，返回的是一个 IStorage 接口。如果操作系统返回一个 IStorage 接口，就可以利用这个接口来管理存储对象和流对象了。

例如，现在有一个 IStorage 指针，则可以通过调用 IStorage::CreateStorage()来创建一个新的子存储对象。而调用 IStorage::CreateStream()可以在这个存储对象下创建一个流对象。也许，你已经注意到了 IStorage 接口并没有 Close()函数，这是因为 IStorage 也是一个 OLE 接口，它会调用 Release()来关闭存储对象。

调用 IStorage::CreateStream()的时候，可以返回一个 IStream 指针。我们可以利用这个指针来读写一个流对象。也就是说，可以将流对象当做普通的文件一样来操作。

当然，并没有说一定要限制在复合文件这个微软的结构化存储模型内。IStorage、IStream 和 ILockBytes 只是几个等待实现的 OLE 接口。但是，微软的复合文件确实可以运行得非常好，所以我们并没有什么必要重新自己建立一个结构化存储模型。而事实上，MFC 也使用复合文件存储 OLE 文档。

IStorage、IStream、ILockBytes 接口组成了存储的一方面。COM 对象得到 IStorage 和 IStream 接口，然后向某种永久介质写入数据。那么，如何创建一个永久对象？COM 对象自己能够保存自己是对象存储技术的另一个方面，这个也非常很重要。OLE 定义了另外一组接口来专门实现这个功能。

永久对象（Persistent Object）

永久对象能够将自己保存到某种介质（比如磁盘）中。在 OLE 中，一个永久对象会提供下面的几个永久接口的其中一个：IPersistStorage、IPersistStream 和 IPersistFile。

通过提供 IPersistStorage 接口给外面，COM 对象告诉外面它能够将自己保存到一个 OLE 结构化存储对象内。IPersistStorage 包含了以下几个函数：Load()、Save()和 IsDirty()。客户可以通过使用这个接口中的函数让 COM 对象使用 OLE 结构化存储对象来保存自己。COM 对象通过提供 IPersistStream 接口来通知客户自己可以保存到一个 OLE 结构化存储流对象中。而 IPersistFile 表示这个 COM 对象能够将自己保存到一个普通的磁盘文件中（与 OLE 存储对象和流对象相对应）。

定点激活（in-place activation）和可视化编辑（visual editing）

一个 OLE 文档中会有另一个 OLE 文档作为它的内容对象（content object）。在前面的例

子中，电子表格就是内容对象。当内容对象作为 OLE 文档一部分时，OLE 称它们为定点条目。在 OLE 文档中，这些条目经常定点被激活和编辑。换句话说，也就是用户不用切换到另外一个应用程序来编辑这个对象。一个应用程序能够得到另外一个应用程序的用户界面，并且能够在不离开本应用程序的前提下编辑内部对象的这种能力就称之为定点激活（in-place activation）。

定点激活也就是告诉文档的一个条目通过其动作（verb）来完成一些操作。链接和嵌入对象可以支持很多不同的动作，最常用的动作是 Open（打开）和 Edit（编辑）。当执行 Open 动作时，一个 OLE 文档的定点条目通常会在另一个窗口中打开一个服务器应用程序。当执行 Edit 动作时，一个 OLE 文档的定点对象会将外面应用程序的主菜单替换为一个复合菜单。复合菜单的内容从外部应用程序的菜单条和服务器应用程序的菜单条中取得。

另外，一个 OLE 文档对象还有可能包含别的动作。例如，微软的媒体播放器包含一个 Play（播放）动作，这个动作通知媒体播放器播放一个 AVI 文件。

当然，和所有的 client/server 结构一样，OLE 文档也分为两部分：容器（container）和服务器（server）。容器中存放着内部对象，这些对象由 OLE 文档服务器提供。

OLE 文档容器

容器中存放着 OLE 文档，也就是说它们包含 OLE 文档内容对象。在前面的例子中，文字处理应用程序就是一个容器，而电子表格应用程序就是一个服务器。一个容器需要做以下这些事情：

- 能够存储整个文档。
- 能够截获内容对象的改变通知。
- 对鼠标双击和执行菜单命令能够响应。

要实现以上这些功能，容器至少要实现 IOleClientSite、IAdviseSink 和 IOleInplaceSite 接口。

容器通过实现 IOleClientSite 接口可以与内容对象进行通信，并且给内容对象提供服务。我们也可以认为 IOleClientSite 是内容对象在 OLE 文档中的一个固定点。容器给每一个内容对象提供一个 IOleClientSite 接口。我们将在下面详细介绍这个接口。

容器也必须实现 IAdivseSink 接口。OLE 文档的内容对象通过容器的 IAdviseSink 接口来接收数据变化、视图变化以及复合文档的变化消息。例如，容器需要通过使用这些消息来使得它们的链接和嵌入对象能够及时更新。OLE 文档容器还在内容对象的 IDataObject 接口中插入了一个 IAdviseSink 接口的实例（这个就是 IDataObject::DAdvise()所做的工作）。如果内容对象有了 IAdviseSink 指针，则它就可以调用这个接口的成员函数来通知容器各种变化。我们也会在下面详细介绍 IAdviseSink 接口。

当服务器进入并接管容器的菜单时，IOleInPlaceSite 接口用来管理 OLE 文档容器的用户界面方面的操作。

下面我们介绍 OLE 文档服务器。

OLE 文档服务器

OLE 文档容器中包含的内容对象都是由 OLE 文档服务器提供的。OLE 文档服务器有以

下一些功能：

- 在容器提供的存储对象中存储相应的数据。
- 将变化通知容器（通过容器的 IAdivseSink 接口）。
- 如果内容对象能够被定点编辑，则提供定点资源（比如菜单和工具条）。
- 提供菜单/用户界面协商

OLE 文档服务器至少要实现 3 个接口：IOleObject、IPersistStorage 和 IDataObject。

让我们先来看看 IOleObject 接口。这个接口也算的上是一个大接口了，有 21 个函数。容器将这个接口作为和内容对象的主要通信手段。例如，IOleObject 包含 DoVerb()和 GetExtent()这样的函数。当一个容器需要通知一个内容对象做什么工作时，这个容器就会调用内容对象的 IOleObject::DoVerb()函数。如果容器需要知道内容对象的大小，则它就可以调用内容对象的 IOleObject::GetExtent()函数。MFC 在 COleSeverItem 和 COleServerDoc 类中实现了这个接口。

在 UDT 中，我们已经讨论过 IDataObject 接口。OLE 文档对象通过提供这个接口使得容器能够得到内容对象的信息。

OLE 文档内容对象实现了 IPersistStorage 接口，容器通过这个接口可以通知内容对象持续保存。IPersistStorage 接口中有函数叫 Load()和 Save()。OLE 文档容器从内容对象中取得这个接口，然后调用这两个函数来导入和保存 OLE 文档内容对象。

上面这些是实现 OLE 文档的最重要的接口。但是它们并不是全部的接口，我们在下面介绍 MFC 的 OLE 文档支持的时候还会介绍用来支持定点激活的接口。

OLE 文档协议

OLE 文档也只是 OLE 特性中的一种。OLE 文档的说明阐述了 OLE 文档容器必须实现的功能以及服务器需要做的工作。也就是说，它规定了在 OLE 文档的两边分别必须实现的接口以及每个接口必须在什么环境下使用。

OLE 文档协议并没有什么特殊的地方。它只是描述了如何使一些 COM 接口与高层协议协同工作。MFC 实现了一个 OLE 文档，下面让我们来看看。

MFC 对 OLE 文档的支持

AppWizard 包含了几个创建 OLE 文档容器和服务器应用程序的按钮。这些按钮到底是拿来干什么的呢？下面让我们来看看。

MFC 对 OLE 文档的支持深深地扎根于它的文档/视图结构。MFC 的文档/视图结构提供了一种一致的模型来管理和提供数据。那么，除了更多（尽管不同类型）的数据外，复合文档的元素是什么呢？

MFC 通过改进文档/视图结构来实现 OLE 文档。基本的思想是 MFC OLE 文档类管理一个对象的集合，这些对象从 CDocItem 派生。在容器这一端，文档类管理 COleClientItem 对象，而在服务器这端，文档类管理 COleServerItems 对象。我们下面先简单地来了解一下基类的信息。

基类：CDocItem 和 COleDocument

CDocItem 是保存在基于 MFC 的 OLE 文档应用程序中的条目的基类。MFC 的 OLE 文档类 COleDocument 管理着一个 CDocItem 链表。CDocItem 类与文档的关系跟视图与文档的关系类似。也就是说，可以循环访问 CDocItem 对象的链表。同样给定一个 CDocItem，也可以访问文档。

CDocItem 除了管理与文档的连接，所做的工作并不很多。CDocItem 并没有实现什么接口，大部分的工作都留给了它的派生类 COleClientItem 和 COleServerItem 来做。

COleDocument 类也没有实现任何接口。它在生存期内的所有工作就是管理一个 CDocItem 对象的链表。COleDocument 从 CDocument 派生，所以 COleDocument 支持所有标准的文档/视图结构的操作，而且它包含一个 CDocItem 对象的链表。

COleDocument 有几个很有意思的成员函数，并且有访问文档条目链表和处理结构化存储的特性。

COleDocument 管理着一个 CDocItem 对象的链表，叫做 m_docItemList。COleDocument 使用下面的几个函数来管理这个链表：GetStartPosition()、GetNextItem()、GetNextClientItem()、GetNextServerItem()、AddItem()和 RemoveItem()。

GetStartPosition()和 GetNextItem()是循环访问这个链表用的，它们是类型安全的。AddItem()和 RemoveItem()使用标准的 MFC 链表管理函数从这个链表中增加或是删除一个文档条目。

COleDocument 只包含一个链表。这个链表中可以存放 COleClientItem 对象，也可以存放 COleServerItem 对象。如果一个应用程序既是一个容器，又是一个服务器，则它可以创建一个混合链表，将 COleClientItem 和 COleServerItem 混合存放在链表中。然而，在很多时候应用程序需要一次访问同一种类型的所有条目。例如，为了实现显示功能，一个容器的视图需要循环访问所有的 COleClientItem 对象，但是略过所有的 COleServerItem 对象。这就是为什么要引入 GetNextClientItem()和 GetNextServerItem()的原因。这两个函数使得应用程序能够访问某种类型的全部条目，这通过调用 COleDocument 中的 GetNextItemOfKind()函数来实现。GetNextItemOfKind()使用 CRuntimeClass 作为参数，扩展了 GetNextItem()的功能。GetNextItemOfKind()循环访问链表，然后返回与 CRuntimeClass 类型匹配的条目。

在结束讨论 CDocItem 链表之前，注意一下它可以保存各种从 CDocItem 中派生出来的对象。也就是说，用户可以从 CDocItem 中派生出自己的类，然后和 COleClientItem 与 COleServerItem 一起存放。

为了管理结构化存储，COleDocument 有一个函数叫 EnableCompoundFile()。这个函数没有什么可说的。调用 EnableCompoundFile()会将 COleDocument 的布尔类型标志 m_bCompoundFile 置为 TRUE，从而打开文档的结构化存储。当文档需要确定是使用标准的文件还是使用 OLE 结构化存储对象时，它就会不停地去检查这个变量。

最后，COleDocument 还包含一个根存储对象（也就是一个 IStorage 指针）。在这个存储对象中保存本地数据以及链接和嵌入的对象。这个变量叫做 m_lpRootStorage。当我们看 OLE 文档容器和 OLE 文档服务器之间的交互时，就会知道如何使用它了。因为根存储对象也只是一个 IStorage 接口，所以在它下面可以很方便地自己创建流对象和存储对象（除了在向文档

中加入 OLE 条目时由 MFC 生成的那些)。

用 MFC 实现的 OLE 文档容器

实际上，MFC 对 OLE 文档容器的支持比较简单。核心就是两个类：COleClientItem 与 COleDocument。

COleClientItem 类

前面说过，COleDocument 保存了一个 CDocItem 链表，COleClientItem 从 CDocItem 派生，所以 COleDocument 可以在它的链表中包含它。在 COleClientItem 中可以找到对 OLE 文档容器的大部分的支持。COleClientItem 有两个功能：(1) 可以管理与服务器应用程序的通信；(2) 可以作为文档中每一个嵌入的或者链接的条目的锚点 (anchor)。

在一个基于 MFC 的 OLE 文档容器中，对每一个嵌入的或是链接的条目，文档中都保存着一个 COleClientItem 与之对应。例如，如果一个 (基于 MFC 的) OLE 文档包含一个 Excel 电子表格、一个用画笔做的图和一个视频剪辑 (video clip)，则这个 OLE 文档需要控制 3 个 COleClientItem。

COleClientItem 通过实现以下几个接口来实现 OLE 文档的协议标准：IOleClientSite、IAdviseSink、IOleWindow 和 IOleInPlaceSite。

IOleClientSite 接口

通过提供一个 IOleClientSite 接口，OLE 文档容器就可以提供一个通道 (channel)，从而 OLE 文档内容对象就可以通过这个通道与容器进行通信。下面是这个接口的定义：

```
interface IOleClientSite : public IUnknown {
public:
   virtual HRESULT SaveObject( void) = 0;
   virtual HRESULT GetMoniker( DWORD dwAssign,
                               DWORD dwWhichMoniker,
                               IMoniker FAR * FAR *ppmk) = 0;
   virtual HRESULT GetContainer(IOleContainer FAR *FAR *ppContainer) = 0;
   virtual HRESULT ShowObject( void) = 0;
   virtual HRESULT  OnShowWindow(BOOL fShow) = 0;
   virtual HRESULT  RequestNewObjectLayout( void) = 0;
};
```

在这里列出这个接口仅供参考，我们将在讨论 MFC OLE 文档容器和服务器的工作时再来具体看这个接口中的函数。

COleClientItem 也实现了 IAdviseSink 接口。

IAdviseSink 接口

COleClientSite 实现了 IAdviseSink 接口。通过这个接口，一个 OLE 文档的内容对象在变化的时候就可以通知 OLE 文档容器。也许你还记得在讨论 UDT 的时候 IDataObject::DAdvise() 就使用 IAdviseSink 作为一个参数。下面是 IAdviseSink 接口的定义：

```
interface IAdviseSink : public IUnknown {
public:
   virtual void OnDataChange(FORMATETC FAR *pFormatetc,
                             STGMEDIUM FAR *pStgmed) = 0;
```

```
    virtual void OnViewChange(DWORD dwAspect, LONG lindex) = 0;
    virtual void OnRename(IMoniker FAR *pmk) = 0;
    virtual void OnSave( void) = 0;
    virtual void OnClose( void) = 0;
};
```

如果这个接口被插入到内容对象的 IDataObject 接口中（使用 IDataObject::DAdvise()），则这个内容对象就可以将它的变化通知容器。例如，如果服务器端的数据发生了改变，则内容对象所要做的工作就是调用 IAdviseSink::OnDataChange()来通知客户数据发生了改变（与之相对应，也许容器会执行一个操作，比如重画它的界面）。

IOleWindow 接口

IOleWindow 使得 OLE 文档容器和内容对象在定点激活的时候能够共享被激活的窗口的句柄。IOleWindow 还能在 OLE 复合文档的应用程序之间管理上下文相关的帮助。下面是 IOleWindow 接口的定义：

```
interface IOleWindow : public IUnknown {
    virtual HRESULT GetWindow(HWND FAR *phwnd) = 0;
    virtual HRESULT ContextSensitiveHelp(BOOL fEnterMode) = 0;
};
```

有几个与定点激活相关的接口从 IOleWindow 接口中继承，包括 IOleInPlaceSite。这个 COleClientItem 实现的最后一个接口。

IOlePlaceSite 接口

COleClientItem 实现了 IOleInPlaceSite 接口，从而协调容器与对象的定点客户端之间的交互操作。这个接口中的函数用来管理定点对象。例如，如果一个用户想要激活一个内容对象，则内容对象使用 IOleInPlaceSite 来（1）确定是否有对象能够被激活；（2）管理激活；（3）管理解除激活的操作。当容器中的某个对象被激活时，内容对象也使用 IOleInPlaceSite 接口来通知容器，并且通知容器准备菜单协商（也就是将服务器的高层菜单传递给容器）。IOleInPlaceSite 还包括一些方法，可以为定点对象提供窗口对象的层次结构以及对象在定点激活窗口中的位置（在父窗口中）。另外，IOleInPlaceSite 还管理容器对对象的滚动，管理对象的取消状态以及当它的边界改变时通知这个对象。这是一个相当大的接口，幸运的是，MFC 已经帮我们实现了它。程序清单 13-4 列出了 IOleInPlaceSite 接口的定义。

程序清单 13-4 IOleInPlaceSite 接口

```
interface IOleInPlaceSite : public IOleWindow {
public:
    virtual HRESULT CanInPlaceActivate( void) = 0;
    virtual HRESULT OnInPlaceActivate( void) = 0;
    virtual HRESULT OnUIActivate( void) = 0;
    virtual HRESULT GetWindowContext(IOleInPlaceFrame FAR *FAR *ppFrame,
                                     IOleInPlaceUIWindow FAR *FAR *ppDoc,
                                     LPRECT lprcPosRect,
                                     LPRECT lprcClipRect,
                                     LPOLEINPLACEFRAMEINFO lpFrameInfo) = 0;
    virtual HRESULT Scroll(SIZE scrollExtant) = 0;
    virtual HRESULT OnUIDeactivate(BOOL fUndoable) = 0;
    virtual HRESULT OnInPlaceDeactivate( void) = 0;
```

```
    virtual HRESULT DiscardUndoState( void) = 0;
    virtual HRESULT DeactivateAndUndo( void) = 0;
    virtual HRESULT  OnPosRectChange(LPCRECT lprcPosRect) = 0;
};
```

在 MFC 中实现一个嵌入和链接容器只要实现 COleDocument 和 COleClientItem 类即可。一个简单的嵌入式容器只包含嵌入对象和简单的链接。然而，别的应用程序可能给你的容器中嵌入的条目或是别的容器嵌入的条目创建链接。

例如，假设你有一个文字处理器文档，这个文档中嵌入了一个电子表格。为了支持嵌入对象的链接，一个应用程序能够给（在字处理器文档中）电子表格建立一个链接。这样，这个应用程序就可以使用电子表格中的数据而不用知道字处理器是从哪里得到它的。这是一种对数据的新的用法。

MFC 还提供了另外一个文档类来支持这个特性，这个类是 COleLinkingDoc。

COleLinkingDoc 类

MFC 的 OLE 文档层次中下一个要介绍的类是 COleLinkingDoc。COleLinkingDoc 从 COleDocument 派生，所以它和 COleDocument 一样，管理着一个 COleClientItem 链表。然而 COleLinkingDoc 还实现另外几个 COM 接口，这几个接口用来链接嵌入在别的地方的条目。这些接口是 IPersistFile、IParseDisplayName、IOleContainer 和 IOleItemContainer。

IPersistFile 接口

一个对象通过实现 IPersistFile 接口来通知它的客户，它能够把自己装载或是保存到一个磁盘文件（而不是一个存储对象）中。实现 IPersist 接口的对象会负责打开它自己的磁盘文件，因为打开一个文件的操作可能从一个应用程序转到另一个应用程序去。COleLinkingDocument 通过实现这个接口，使得别的 OLE 文档能够链接到已经包含嵌入对象的 MFC OLE 文档容器上。换句话说，也就是包含嵌入对象的 OLE 文档容器自己能够被链接到别的一些 OLE 文档中。

```
interface IPersistFile : public IPersist {
   virtual HRESULT IsDirty( void) = 0;
   virtual HRESULT Load( LPCOLESTR pszFileName,
                         DWORD dwMode) = 0;
   virtual HRESULT Save(LPCOLESTR pszFileName,
                        BOOL fRemember) = 0;
   virtual HRESULT SaveCompleted(LPCOLESTR pszFileName) = 0;
   virtual HRESULT GetCurFile(LPOLESTR FAR *ppszFileName) = 0;
};
```

IPersistFile 通常在这样的对象中实现：这个对象还包含支持名字对象（moniker）的接口。

名字对象（moniker）是一个新的概念。那么，什么是名字对象呢？当 OLE 文档的内容对象被链接时，实际的数据是保存在别的地方。一个名字对象管理着一些必要的信息，使得 OLE 文档容器能够根据这些信息来定位链接的数据然后绑定到它。一个名字对象需要实现一个 IMoniker 接口。

MFC OLE 文档容器需要处理两种类型的名字对象：文件名字对象（file moniker）和条目名字对象（item moniker）。文件名字对象标志了存储在文件中的数据，它包括到达链接的数

据的一条路径。条目名字对象能够管理一个更合理的粒度：它标志在文件中的数据项。

那么，在 MFC 和 OLE 文档中该如何做呢？当一个 OLE 文档容器创建了到另外一个文件的简单链接时，这个文档使用文件名字对象进行处理。MFC 的实现也没有什么好说的。当使用一个链接的容器时（这个容器包含了别处对嵌入对象的链接），MFC 会使用条目名字对象。使用条目名字对象的一个很好的例子是当在一个文档中需要命名一系列的电子表格单元时，就可以使用条目名字对象来进行处理。我们并不需要链接到整个电子表格，而只要链接到这些单元即可。这一系列单元可以由一个条目名字对象来标志。

COleLinkingDoc 还实现了另外 3 个接口来实现一个 OLE 链接文档。这 3 个接口是 IParseDisplayName、IOleContainer 和 IOleItemContainer。

IParseDisplayName 接口

IParseDisplayName 接口将一个用自然语言书写的名字字符串转换为一个名字对象。这个接口只包含一个函数：

```
interface IParseDisplayName : public IUnknown {
   virtual HRESULT ParseDisplayName(IBindCtx FAR *pbc,
                                    LPOLESTR pszDisplayName,
                                    ULONG FAR *pchEaten,
                                    IMoniker FAR *FAR *ppmkOut) = 0;
};
```

IOleContainer 接口

通过实现 IOleContainer 接口，支持链接到嵌入对象的应用程序能够提供对象枚举、名字解析以及链接源数据的自动更新功能。也就是说，没有链接的容器没有必要实现这个接口，所以这个接口在 COleLinkingDoc 中实现（而不是在 COleDocument 中）。

下面是这个接口的定义：

```
interface IOleContainer : public IParseDisplayName {
   virtual HRESULT EnumObjects(DWORD grfFlags,
                               IEnumUnknown FAR * FAR *ppenum) = 0;
   virtual HRESULT LockContainer(BOOL fLock) = 0;
};
```

IOleItemContainer 接口

当条目名字对象被绑定到它的对象时，需要用到 IOleItemContainer 接口。

```
interface IOleItemContainer : public IOleContainer {
   virtual HRESULT GetObject(LPOLESTR pszItem,
                             DWORD dwSpeedNeeded,
                             IBindCtx FAR *pbc,
                             REFIID riid,
                             void FAR * FAR *ppvObject) = 0;
   virtual HRESULT GetObjectStorage(LPOLESTR pszItem,
                                    IBindCtx FAR *pbc,
                                    REFIID riid,
                                    void FAR * FAR *ppvStorage) = 0;
   virtual HRESULT IsRunning(LPOLESTR pszItem) = 0;
};
```

名字对象的提供者（比如一个支持链接的 OLE 文档服务器）必须实现 IOleItemContainer 接口。一个名字对象能够提供标识对象的名字对象，并且使名字对象的客户能够访问这些对象（也就是 OLE 文档容器）。

支持链接的 OLE 文档服务器定义它们自己的名字体系来标识容器内的对象。这个命名体系由服务器来决定。例如，一个文档中的嵌入对象可以命名为“Embedding1”、“Embedding2”等等（而事实上，MFC 容器确实是这么命名嵌入对象的）。另一方面，电子表格单元格链表可能被命名为“A1:B4”等等。（一列电子表格单元格是一种伪对象的例子。它们并没有自己的永久存储，只不过是表示容器的内部状态。）一个应用程序使用 CreateItemMoniker()这个 API 函数来创建一个条目名字对象来表示一个对象的名字，并且将它传递给名字对象的客户。当条目名字对象被绑定时，容器对 IOleItemContainer 的实现就必须能够根据名字返回相应的对象。

上面是 MFC 对 OLE 文档容器的支持，下面我们来看 MFC 对 OLE 文档服务器的支持。

使用 MFC 实现 OLE 文档服务器

MFC 的 OLE 文档服务器结构与 MFC 对容器的实现结构非常类似。一个 MFC OLE 文档服务器从 COleServerDoc（而不是 COleDocument 或 COleLinkingDoc）中派生出它的文档，在文档的 CDocItem 链表中包含 COleServerItem 对象。

COleServerDoc 类

COleServerDoc 也是 CDocument 继承体系下面的一个子类，所以它拥有保存 CDocItem 链表所需的所有机制（与由 COleLinkingDoc 提供的链接容器的支持一样）。并且，COleServerDoc 也能在链表中支持 COleServerItem。

一个嵌入的 OLE 文档内容对象至少要支持以下几个接口：IOleObject、IPersistStorage 和 IDataObject。在 COleServerDoc 的接口映射表中可以找到 COleServerDoc 对这几个接口的实现。通过实现 IOleInPlaceObject 和 IOleInPlaceActiveObject 这两个接口，COleServerDoc 也能够支持定点激活（in-place activation）。

IPersistStorage 接口

COleServerDoc 类通过实现 IPersistStorage 接口来通知 OLE 文档容器它支持 OLE 的结构化存储。下面是 IPersistStorage 接口的定义，我们稍微看一眼它的成员函数：

```
interface IPersistStorage : public IPersist {
public:
   virtual HRESULT IsDirty( void) = 0;
   virtual HRESULT InitNew(IStorage FAR *pStg) = 0;
   virtual HRESULT Load(IStorage FAR *pStg) = 0;
   virtual HRESULT Save(IStorage FAR *pStgSave,
                        BOOL fSameAsLoad) = 0;
   virtual HRESULT SaveCompleted(IStorage FAR *pStgNew) = 0;
   virtual HRESULT HandsOffStorage(void) = 0;
};
```

OLE 文档容器得到内容对象的 IPersistStorage 接口，然后使用这个接口来存储 OLE 文档内容对象。

IOleObject 接口

OLE 文档内容对象所必须实现的第二个接口是 IOleObject。这个接口负责处理内容对象的通信功能。程序清单 13-5 就是 IOleObject 接口的定义。

程序清单 13-5 IOleObject 接口

```
interface IOleObject : public IUnknown {
public:
   virtual HRESULT  SetClientSite(IOleClientSite FAR *pClientSite) = 0;
   virtual HRESULT  GetClientSite(IOleClientSite FAR *FAR
                                  *ppClientSite) = 0;
   virtual HRESULT  SetHostNames(LPCOLESTR szContainerApp,
                                 LPCOLESTR szContainerObj) = 0;
   virtual HRESULT  Close(DWORD dwSaveOption) = 0;
   virtual HRESULT  SetMoniker(DWORD dwWhichMoniker,
                               IMoniker FAR *pmk) = 0;
   virtual HRESULT  GetMoniker(DWORD dwAssign,
                               DWORD dwWhichMoniker,
                               IMoniker FAR *FAR *ppmk) = 0;
   virtual HRESULT  InitFromData(IDataObject FAR *pDataObject,
                                 BOOL fCreation,
                                 DWORD dwReserved) = 0;
   virtual HRESULT  GetClipboardData(DWORD dwReserved,
                                     IDataObject FAR *FAR
                                     *ppDataObject) = 0;
   virtual HRESULT  DoVerb(LONG iVerb,
                           LPMSG lpmsg,
                           IOleClientSite FAR *pActiveSite,
                           LONG lindex,
                           HWND hwndParent,
                           LPCRECT lprcPosRect) = 0;
   virtual HRESULT  EnumVerbs(IEnumOLEVERB FAR *FAR *ppEnumOleVerb) = 0;
   virtual HRESULT  Update( void) = 0;
   virtual HRESULT  IsUpToDate( void) = 0;
   virtual HRESULT  GetUserClassID(CLSID FAR *pClsid) = 0;
   virtual HRESULT  GetUserType(DWORD dwFormOfType,
                                LPOLESTR FAR *pszUserType) = 0;
   virtual HRESULT  SetExtent(DWORD dwDrawAspect,
                              SIZEL FAR *psizel) = 0;
   virtual HRESULT  GetExtent(DWORD dwDrawAspect,
                              SIZEL FAR *psizel) = 0;
   virtual HRESULT  Advise(IAdviseSink FAR *pAdvSink,
                           DWORD FAR *pdwConnection) = 0;
   virtual HRESULT  Unadvise(DWORD dwConnection) = 0;
   virtual HRESULT  EnumAdvise(IEnumSTATDATA FAR *FAR *ppenumAdvise) = 0;
   virtual HRESULT  GetMiscStatus(DWORD dwAspect,
                                  DWORD FAR *pdwStatus) = 0;
   virtual HRESULT  SetColorScheme(LOGPALETTE FAR *pLogpal) = 0;
};
```

这也是一个很大的接口。很幸运的是 MFC 已经帮我们实现了这个接口。IOleObject 总共有 21 个函数（不包括 IUnknown 接口的几个函数）。MFC 将这个接口分到 COleServerItem 类和 COleServerDoc 类中来实现。我们将在讨论 MFC OLE 文档容器和服务器之间的交互时再来看 MFC 如何实现这个接口。

IDataObject 接口

在第 12 章我们已经见过了 IDataObject 接口。尽管可以使用 IDataObject 接口很方便地来实现剪贴板和拖放操作，但是 IDataObject 本来设计的目的就是在 OLE 文档服务器中使用。程序清单 13-6 就是这个接口的定义。

程序清单 13-6 IDataObject 接口

```
interface IDataObject : IUnknown {
   virtual   HRESULT  GetData(FORMATETC* pformatetc,
                              STGMEDIUM* pmedium) = 0;
   virtual   HRESULT  GetDataHere(FORMATETC* pformatetc,
                                  STGMEDIUM* pmedium) = 0;
   virtual   HRESULT  QueryGetData(FORMATETC* pformatetc) = 0;
   virtual   HRESULT  GetCanonicalFormatEtc(FORMATETC* pformatetcIn,
                                            FORMATETC* pformatetcOut) = 0;
   virtual   HRESULT  SetData(FORMATETC *pformatetc,
                              STGMEDIUM* pmedium,
                              BOOL fRelease) = 0;
   virtual   HRESULT  EnumFormatEtc(DWORD dwDirection,
                                    FORMATETC* pformatetc) = 0;
   virtual   HRESULT  DAdvise(FORMATETC* pformatetc,
                              DWORD grfAdvf,
                              IAdviseSink *pAdvSink,
                              DWORD pdwConnection) = 0;
   virtual   HRESULT  DUnadvise(DWORD dwConnection) = 0;
   virtual   HRESULT  EnumDAdvise(IEnumSTATDATA *ppenumAdvise) = 0;
}
```

除了这 3 个必要的接口外，MFC OLE 文档服务器类还需要另外几个接口来支持定点激活：IOleInPlaceObject 和 IOleInPlaceActiveObject 接口。

IOleInPlaceObject 接口

容器使用 IOleInPlaceObject 接口来激活定点对象或使定点对象无效，还利用这个接口来管理可见定点对象的数目。下面是 IOleInPlaceObject 接口的定义，它从 IOleWindow 派生：

```
interface IOleInPlaceObject : public IOleWindow
   virtual HRESULT InPlaceDeactivate( void) = 0;
   virtual HRESULT UIDeactivate( void) = 0;
   virtual HRESULT SetObjectRects(LPCRECT lprcPosRect,
                                  LPCRECT lprcClipRect) = 0;
   virtual HRESULT ReactivateAndUndo( void) = 0;

};
```

COleServerDoc 最后一个需要实现的接口是 IOleInPlaceActiveObject。

IOleInPlaceActiveObject 接口

OLE 文档内容对象通过实现 IOleInPlaceActiveObject 接口在下面 3 者之间提供一个直接通道：（1）一个定点对象；（2）对象服务器的应用程序的最外层的框架窗口；（3）容器应用程序中的文档窗口。这个接口所做的工作是翻译消息、管理框架窗口的状态（激活或使无效）。当需要改变内容对象边界的大小时，IOleInPlaceActiveObject 也会通知内容对象。它还管理无模式对话框。下面是这个接口的定义：

```
interface IOleInPlaceActiveObject : public IOleWindow {
   virtual HRESULT TranslateAccelerator(LPMSG lpmsg) = 0;
   virtual HRESULT OnFrameWindowActivate(BOOL fActivate) = 0;
   virtual HRESULT OnDocWindowActivate(BOOL fActivate) = 0;
   virtual HRESULT ResizeBorder(LPCRECT prcBorder,
                                IOleInPlaceUIWindow FAR *pUIWindow,
                                BOOL fFrameWindow) = 0;

   virtual HRESULT EnableModeless(BOOL fEnable) = 0;
};
```

IOleInPlaceActiveObject 也从 IOleWindow 类派生，所以它也有 GetWindow()和 ContextSensitiveHelp()成员函数。

COleServerItem 类

MFC 对 OLE 文档服务器支持的另外一部分在 COleServerItem 类中实现。COleServerItem 实现了两个接口：IOleObject 和 IDataObject。MFC 将这两个接口分开到 COleServerDoc 和 COleServerItem 两个类中实现。有关 COleServerItem 类的内容在这里就不多说了，因为在执行过程中看这些类更有意思。下面让我们来看 MFC OLE 文档的容器和服务器如何工作。

容器/服务器的协调工作

单独讲述 MFC 的 OLE 文档实现的任何一个部分都是很枯燥的。相比较而言，当 OLE 文档容器和服务器运行时，观察它们之间的交互操作相对有趣一些。我们循着嵌入的 OLE 文档容器和服务器之间的操作来讨论，在这个过程中，将会讨论下面一些问题：

- 一个容器文档如何创建一个内容对象。
- 容器和服务器如何处理结构化存储。
- 容器和服务器之间如何通信。

容器：创建一个新文件

在容器这边，当应用程序视图创建一个新文件的时候就开始执行操作。在下面两种情况下会产生操作：（1）应用程序开始启动；（2）用户在 File 菜单上选了 New。MFC 通过调用 CWinApp 的 OnFileNew()函数来响应。现在我们假设用户选了 FileNew。框架会去调用 CWinApp::OnFileNews()函数来响应 FileNew 菜单操作。CWinApp::OnFileNew()通过几个步骤来得到一个新的文档，然后运行它。这几个步骤是：

- CWinApp::OnFileNew()调用文档管理器的 OnFileNew()函数。

- 文档管理器得到一个文档模板，然后调用模板的 OpenDocumentFile()函数来打开文档文件。
- 文档模板的 OpenDocumentFile()调用文档模板的 CreateNewDocument()函数。
- 最后调用文档的 OnNewDocument()函数。

上面的步骤是标准的 MFC 文档/视图结构的操作，在第 7 章和第 8 章中有讨论。

如果使用 AppWizard 来产生 OLE 文档服务器代码，则 AppWizard 会在文档的构造函数中插入一个 EnableCompoundFile()函数的调用。这个函数会将 COleDocument 的 m_bEmbedded 标志设为 TRUE。

AppWizard 还会在文档的 OnNewDocument()函数中插入对 COleDocument::OnNewDocument()函数的调用。如果这个文档中复合文件能够被使用，那么 COleDocument::OnNewDocument()释放已有的 m_lpRootStorage 指针，然后使用 OLE API 函数 StgCreateDocFile()创建一个新的指针。在结构化存储被启动然后开始运行后，CMultiDocTemplate::OpenDocumentFile()调用 InitialUpdataFrame()来更新所有的视图。

当容器的结构化存储被打开后，它就可以接受一个新的 OLE 文档的条目了。

加入容器条目

MFC 的 OLE 文档实现中管理着一个 COleClientItem 对象链表。那么这些对象是从何而来的呢？通常这些对象从 3 个地方来：(1) 用户从菜单上选择“Insert New Object”之类的选项；(2) 用户在 Edit 菜单上选择 Paste 或 Paste Special 选项；(3) 用户从 Edit 菜单上选择 Paste Link 选项。这些方法都会构造一个 COleClientItem 对象。下面是在 MFC 中如何实现它。

构造 COleClientItem 对象

在文档中加入一个 COleClientItem 之前，必须先构造一个。COleClientItem 的构造函数只有一个参数——一个指向 COleDocument 的指针。下面是它的函数原型：

```
COleClientItem::COleClientItem(COleDocument* pContainerDoc);
```

COleClientItem 的构造函数将成员函数初始化为默认的值，基本上是 NULL、void 或是 FALSE。然后这个构造函数使用 COleDocument 的 AddItem()函数将 COleClientItem 加入到文档中。

这个确实非常简单。但是，如果一个 COleClientItem 已经被创建了，那该如何做呢？我们从头开始来看看如何使用 MFC 的 COleInsertDlg 类创建一个新的 COleClientItem。

容器：插入一个新的条目

大部分的容器都有一个菜单选项用来插入一个新的 OLE 文档对象。比如 Word 中就有一个“InsertIObject”菜单项。使用 AppWizard 来产生应用程序代码会在 Edit 菜单下面产生一个“Insert New Object”选项。处理“Insert New Object”菜单项最简单的做法是激活 COleInsertDlg 对话框，然后使用它来创建一个 OLE 文档内容对象。

从开发者的观点来看，使用 COleInsertDlg 类是相当简单的，只要声明它的一个实例，然后调用 DoModal()函数来激活它即可。COleInsertDlg 是对 OLE API 函数::OleUIInsertObject()的一个包装。::OleUIInsertObject()会激活标准的 Insert Object 对话框。通过这种方法，用户可

以选择一个对象源和类名，还可以选择是显示对象的全部内容还是只显示图标。

图 13-4 就是 COleInsertDlg 生成的 Insert Object 对话框。

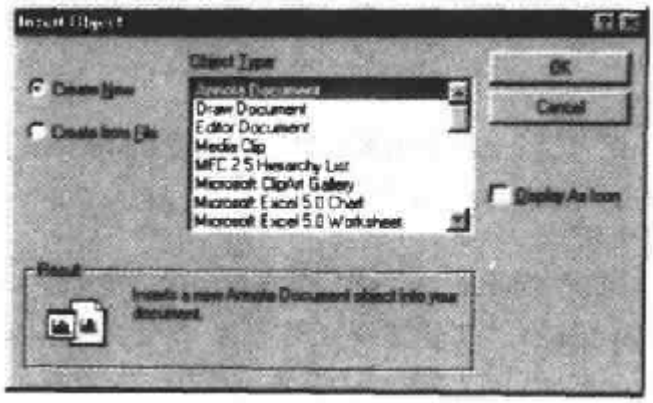


图 13-4 COleInsertDlg 对话框

在调用完 COleInsertDlg::DoModal()后，就可以调用 COleInsertDlg 的 CreateItem()函数来创建一个客户条目：

```
BOOL CreateItem(COleClientItem* pItem);
```

现在只是创建了一个 COleClientItem 对象，下面要做的是将新建的 COleClientItem 传递给 COleInsertDlg 的 CreateItem()函数。CreateItem()能够处理在创建 COleClientItem 的新子存储对象（substorage）过程中的所有细节，并且会激活服务器以及建立 OLE 文档协议所需要的一些连接。

程序清单 13-7 展示了做这些工作的过程，这些代码都是 AppWizard 生成的。

程序清单 13-7 在一个容器中插入一个 OLE 文档内容对象的代码

```
void OnEditInsertNewObject() {
   COleInsertDialog dlg;
   if (dlg.DoModal() != IDOK)
      return;

   CContainerItem* pItem = NULL;
   TRY
   {
      // Create new item connected to this document.
      CContainerDoc* pDoc = GetDocument();
      pItem = new CContainerItem(pDoc);

      // Initialize the item from the dialog data.
      if (!dlg.CreateItem(pItem))
         AfxThrowMemoryException();  // any exception will do
      ASSERT_VALID(pItem);
   }

   CATCH(CException, e) {
      if (pItem != NULL) {
         pItem->Delete();
```

```
        }
        AfxMessageBox("Couldn't create OLE Document content object");
    }
    END_CATCH

    EndWaitCursor();
};
```

MFC 是不是帮我们做完了全部创建内容对象的工作呢？MFC 做了很多的工作，下面让我们来看看 COleInsertDlg::CreateItem()的内部实现。

CreateItem()的内部实现

如果新的客户条目只是简单嵌入，则 COleInsertDlg::CreateItem()就将工作委托给 COleClientItem::CreateNewItem()。其他情况还包括从一个文件中创建一个条目（使用 COleClientItem::CreateFromFile()）或者从一个文件中链接一个条目（使用 COleClientItem::CreateLinkFromFile()）。

OLE 文档容器的一个主要功能就是管理和组织一个文件中的各种内容对象。COleClientItem::CreateNewItem()所做的第一件事情就是给内容对象创建一个子存储对象（substorage）。每一个子存储对象都必须有一个惟一的名字，所以 CreateNewItem()需要产生这个名字。COleDocument 保存着这文档中条目的数目，每次当一个新的客户条目被加入到文档中，文档会将这个计数加 1。新的条目就和下一个数字关联起来。客户条目使用这个数字来惟一标识客户条目的子存储对象。每一个新的客户条目都使用类似于“EmbeddingX”的名字来存储，其中“X”是条目关联的数字。

下一步，COleClientItem::CreateNewItem()通过调用 COleClientItem::GetItemStorage()来为条目创建一个新的存储空间。GetItemStorage()会去检查文档的 m_bCompoundFile 标志（这个标志通过调用 COleDocument::EnableCompoundFile()函数来设定）。如果 COleDocument 使用的是一个复合文件，则 GetItemStorage()会调用 GetItemStorageCompound()，否则，就调用 GetItemStorageFlat()来创建存储对象。

GetItemStorageCompound()使用文档的 m_lpStorage 变量来创建一个新的存储对象。m_lpStorage 是一个指向 IStorage 接口的指针。IStorage 有一个成员函数叫 CreateStorage()，它会创建一个新的子存储对象。然后 GetItemStorageCompound()会将 COleClientItem 的 m_lpStorage 成员变量指向新创建的存储对象。这个 IStorage 就是内容对象用来存储它的数据的地方。

如果应用程序没有使用复合文件，则客户条目使用 GetItemStorageFlat()函数来创建一个新的存储对象。首先，GetItemStorageFlat()使用 OLE API 函数 CreateILockBytesOnHGlobal()来创建一个对象，这个对象实现了 ILockBytes 的接口。然后，GetItemStorageFlat()将 ILockBytes 接口指针传递给 OLE API 函数 StgCreateDocfileOnILockBytes()。这个函数返回一个 IStorage 接口指针，这个指针所指的地方存放着内容对象的真正数据。

上面的两种方法都会使用一个 IStorage 接口指针。那么在 CreateNewItem()中是如何使用这个存储对象的呢？CreateNewItem()会使用它来调用 OleCreate()函数。下面是 OleCreate()的定义：

```
WINOLEAPI  OleCreate(REFCLSID rclsid, REFIID riid, DWORD renderopt,
                     LPFORMATETC pFormatEtc, LPOLECLIENTSITE pClientSite,
                     LPSTORAGE pStg, LPVOID FAR* ppvObj);
```

OleCreate()是 CoCreateInstance()的一个升级版本。这个函数有 7 个参数：

1. 一个 GUID，表示需要的 COM 类。
2. 一个 GUID，表示请求的接口。
3. 数据的供应方式，表示对象如何被缓存。
4. 一个指向 FORMATETC 结构的指针，用来辅助说明对象被缓存的方式。
5. 一个指向客户位置（一个 IOleClientSite 接口）的指针。
6. 一个指向存储对象（一个 IStorage 接口）的指针。
7. 一个位置，用来存放请求的接口指针。

下面演示 COleInsertDlg::CreateItem()如何调用 OleCreate()：

```
OleCreate(clsid, IID_IUnknown, render,
          lpFormatEtc, lpClientSite, m_lpStorage,
          (LPLP)&m_lpObject);
```

其中的 clsid 是通过 COleInsertDlg::CreateItem()传递给 CreateNewItem()。Class ID 是从用户选择的 program ID 中转换过来的（在列表框中选择高亮的部分）。CreateNewItem()还请求 IUnknown 接口。为了描述显示缓存的方式，CreateNewItem()提供的默认设置是 OLERENDER_DRAW，表示将在屏幕上画数据。在 CreateNewItem()的参数链表中，FORMATETC 指针的默认设置是 NULL。CreateNewItem()使用 GetInterface()从 COleClientItem 中取得 IOleClientSite 接口。IStorage 指针是在容器文档的根存储对象下创建的一个子存储对象。

OleCreate()分以下几步执行：

1. OleCreate()使用 CoCreateInstance()创建一个 COM 对象。
2. OleCreate()调用 QueryInterface()得到内容对象的 IPersistStorage 接口指针。
3. OleCreate()调用 IPersistStorage::InitNew()传递给内容对象一个 IStorage 指针。内容对象就使用这个子存储对象在容器的文档中存储自己的数据。
4. OleCreate()接着传递给内容对象一个 IOleClientSite 的接口指针，然后开始初始化缓存。

如果顺着执行一遍，你就可以发现容器如何调用 OleCreate()来启动服务器应用程序，得到内容对象的类厂，以及如何创建内容对象。如果服务器是一个 MFC 应用程序，则通常会创建文档。

当文档被创建后，容器将会试图得到内容对象的 IPersistStorage 接口，然后调用 IPersistStorage::InitNew()来传递新创建的子存储对象。如果服务器是一个 MFC 应用程序，则这个调用在 COleServerDoc::XPersistStorage::InitNew()中结束。

在服务器这一端，COleServerDoc 通过调用 COleServerDoc::OnNewEmbedding()来处理 IPersistStorage::InitNew()。COleServerDoc::OnNewEmbedding()会清空文档。这个清空操作包括调用 COleLinkingDoc::DeleteContents()来删除以前的链接。COleLinkingDoc:: DeleteContent()会调用 COleDocument::DeleteContents()来删除所有的 COleClientItem 对象。然后文档会删除所有的 COleServerItem 对象。最后，COleServerDoc 会使用一个成员变量 m_lpRootStorage 来

保存客户的 IStorage 接口。COleServerDoc 需要利用这个接口来访问容器的存储对象（读写数据）。然后 OnNewEmbedding()调用 OnNewDocument()来初始化一个新的文档。

从上面来看，服务器会竭尽所能来初始化一个特定的文档。例如，一个 MFC 的 Scribble 指南会从文档中删除所有的直线。

在返回之前，OnNewEmbedding()会设置“文档被改变”的标志，然后调用 SendInitialUpdate()来更新所有的视图。OnNewEmbedding()返回到 COleServerDoc::XPersistStorage::InitNew()中，该函数会将控制返回到容器中（控制还在 OleCreate()中）。

接着，OleCreate()会调用内容对象的 IOleObject::SetClientSite()，使得内容对象能够保存容器的客户地址。如果服务器是基于 MFC 的应用程序，则在 COleServerDoc::XOleObject::SetClientSite()这个过程中结束。COleServerDoc 将客户地址保存到 m_lpClientSite 这个成员变量中。然后控制转移回容器对象，结束对 OleCreate()的调用。

现在回到容器这一端（在调用了 OleCreate()后），COleClientItem 现在需要连接到内容对象。COleClientItem 有一个名为 FinishCreate()的函数就用来完成这个工作。FinishCreate()使用一个 IUnknown 指针作为参数（这个指针是从 OleCreate()返回的），并调用 QueryInterface()来取得内容对象的 IOleObject 接口。FinishCreate()在 COleClientItem 的一个成员变量中保存 IOleObject 接口指针。客户条目在它的生存期内会经常使用这个接口。FinishCreate()还会取得内容对象的 IViewObject2 接口指针。最后，FinishCreate()将客户条目的 IAdviseSink 插入到内容对象的 IDataObject 接口中（使用 IDataObject::DAdvise()函数），然后再插入到内容对象的 IViewObject2 接口中（使用 IViewObject2::SetAdvise()）。现在容器就可以和内容对象很方便地通信了，而且内容对象也能够及时地将变化通知容器。

在容器创建了 COleClientItem 后，它就可以使用这个类了。通常容器做的第一件事情就是调用内容对象的 IOleObject 接口中的 DoVerb()来激活对象，并使用 OLEIVERB_PRIMARY 常数。这个过程比较繁琐，但是还是比较有趣的。

DoVerb()函数和定点激活

每当一个容器调用内容对象的 IOleObject::DoVerb()函数时，就会发生定点激活。这个地方也是所有菜单迷惑人的地方，所以值得研究一下。

如前所述，容器和内容对象之间可以有很多种方式进行对话，而在 COleClientItem 中也保存了内容对象的 IOleObject 接口指针。客户主要使用 IOleObject 接口跟内容对象通信。在 IOleObject 接口中会发现 DoVerb()函数。

COleClientItem 中有一个函数叫 DoVerb()，用来在服务器端执行一个动作。COleClientItem::DoVerb()会去调用 COleClientItem::Activate()。Activate()有 3 个参数：表示动作的数字、一个指向 CView 的指针以及一个指向触发动作的消息的指针。

IOleObject::DoVerb()有几个参数，COleClientItem::Activate()必须对这些参数进行包装：

```
HRESULT DoVerb(LONG iVerb, // verb to execute
               LPMSG lpmsg, // message that invoked the verb
               IOleClientSite FAR *pActiveSite, // container's client site
               LONG lindex, // reserved
               HWND hwndParent, // parent window
               LPCRECT lprcPosRect); // position of inplace item
```

Activate()首先调用 COleClient::OnGetItemPosition()得到条目在屏幕上的位置。(这个函数是“CONTAIN”指南中说明的需要重载的函数之一，现在起码你应该知道何时使用它。)

然后，Activate()会取得 COleClientItem 的 IOleClientSite 接口指针以及视图的 HWND。由于 COleClientItem 保存着一个内容对象的 IOleObject 接口指针，所以 COleClientItem 就可以直接调用这个接口的函数：

```
m_lpObject->DoVerb(nVerb, lpMsg,
                   lpClientSite, -1,
                   hWnd, lpPosRect);
```

注意：如果这时候服务器还没有被装载，服务器就会导入默认的处理函数，然后调用内容对象的 IPersistStorage::Load()函数。在一个基于 MFC 的 OLE 文档服务器中，这个调用在 COleServerDoc::XPersistStorage::Load()后结束。

现在我们来看看内容对象。COleClientItem 对 IOleObject::DoVerb()的调用在 COleServerDoc::XOleObject::DoVerb()中结束。稍微看一下 COleServerDoc::XOleObject::DoVerb()的实现就知道，它忽略了除了动作的其他所有参数。

```
COleServerDoc::XOleObject::DoVerb(LONG iVerb,
                                  LPMSG /*lpmsg*/,
                                  LPOLECLIENTSITE /*pActiveSite*/,

                                  LONG /*lindex*/,
                                  HWND /*hwndParent*/,
                                  LPCRECT /*lpPosRect*/)
```

COleServerDoc::XOleObject::DoVerb()通过调用 COleServerDoc::GetEmbeddedItem()来取得文档的 COleServerItem，然后将工作委托给嵌入条目的 OnDoVerb()成员函数。

COleServerItem::OnDoVerb()基本上是一个 switch 语句，它对动作（verb）进行 switch 判断。OnDoVerb()处理标准的动作 Open、Show 和 Hide。如果请求的动作不是标准动作之一而且是个负数，则 OnDoVerb()会返回 E_NOTIMPL。如果请求的动作不是标准动作之一而且它个正数，则 OnDoVerb()默认地执行基本的动作。

COleServerItem::OnDoVerb()是一个虚函数。所以如果需要在 OLE 文档服务器中增加一个自定义的动作，可以在这个函数里面添加。只要重载 COleServerItem::OnDoVerb()并且处理自定义的动作，同时将别的动作交给框架进行处理即可。

我们现在需要看看容器在激活条目时会发生什么事，所以动作是 OLEIVERB_PRIMARY。OnDoVerb()会调用 COleServerItem::OnShow()函数。这个函数会回过头来取得文档，然后调用 COleServerItem::ActivateInPlace()。

在定点编辑（in-place editing）中 COleServerItem::ActivateInPlace()会去激活条目。这个函数会执行定点激活中所需要的全部操作，其中包括（1）创建一个定点框架窗口；（2）定位定点窗口；（3）将容器和服务器窗口相互关联，使得菜单命令能够被正确地执行；（4）建立共享菜单控件；（5）将条目放入视图中，然后将焦点放到定点框架窗口上。下面我们来看看 MFC 如何分别实现这些函数。

创建定点框架窗口

COleServerItem::ActivateInPlace()首先会根据应用程序的名字以及文件类型创建一个标

题。比如 Scribble 的嵌入标题就叫做“Scribble－SCRIB”。

下一步，ActivateInPlace()通过对容器的 IOleClientSite 接口调用 QueryInterface()函数来取得 IOleInPlaceSite 接口。如果容器的 IOleInPlaceSite 接口是有效的，则内容对象会使用 IOleInPlaceSite::CanInPlaceActivate()来查询容器是否能够被“定点激活”。

在客户端，在调用 COleClientItem::XOleIPSite::OnInPlaceActivate()后结束，其中该函数会去调用 COleClientItem::OnActivate()。OnActivate()调用 OleLockRunning()将内容对象从导入状态转换到运行状态。在返回之前，OnActivate()会调用客户条目的 OnChange()函数（这样容器就可以执行诸如更新屏幕内容之类的操作了）。

现在控制返回到 COleServerDoc 中（还在 ActivateInPlace()中）。下一步，ActivateInPlace()会调用 IOleInPlaceSite 的 GetWindow()函数。

在容器这一端，在调用完 COleClientItem::XOleIPSite::GetWindow()后就结束了。它只是把视图的窗口句柄返回给内容对象。

在服务器这一端，COleServerDoc::ActivateInPlace()会创建一个 CWnd 对象，然后 ActivateInPlace()利用这个 CWnd 对象创建一个定点框架（这样容器的视图就成为了定点框架的父亲）。

COleServerDoc 使用一个成员函数 CreateInPlaceFrame()来创建定点框架。COleServerDoc::CreateInPlaceFrame()取得文档模板（由于现在控制还是在文档对象中，所以可以得到文档模板）。CDocTemplate 中的函数 CreateOleFrame()会创建一个 COleIPFrameWnd 然后装载相应的服务器资源。CreateInPlaceFrame()使用为文档创建的第一个视图。这个视图会暂时和原来的框架分离，然后和新创建的定点框架关联起来。完成这个功能的两个函数 CreateInPlaceFrame()和 DestroyInPlaceFrame()都是虚函数。如果这两个函数默认的实现不符合用户的应用程序，可以对它们进行重载。如果你对视图的排列是非标准的或者是半标准的，重载就显得很有必要了。例如，如果想在定点会话中使用裁剪，则必须提供这两个函数的一个新的实现，也就是重载它们。

下一步，ActivateInPlace()会调用客户的 IOleInPlaceSite::OnUIActivate()函数。

定位定点窗口、连接服务器与容器窗口

现在控制转移到了 COleClientItem::XOleIPSite::OnUIActivate()中。这个函数会去调用 COleClientItem::OnActivateUI()。OnActivateUI()所做的工作很简单，就是检查一下容器视图的 WS_CLIPCHILDREN 位是否设置了。这一步很有必要，因为对象的定点窗口中的内容必须限制在定点窗口中。接着 COleClientItem::XOleInPlaceSite::OnUIActivate()会取得服务器的 IOleInPlaceObject 接口指针，然后使用这个接口得到服务器的定点框架的 HWND，然后将它保存到 COleClientItem::m_hWndServer 中。

在服务器中，内容对象会调用容器的 IOleInPlaceSite::GetWindowContext()函数。内容对象使用这个函数得到两个信息：（1）对象的定点激活窗口在父窗口中的位置；（2）形成窗口对象层次结构的窗口接口（这些接口包括 OleInPlaceFrame 和 OleInPlaceUIWindow 接口）。OLE 使用术语框架窗口（frame window）和文档窗口（document window）来描述这个层次结构。框架窗口是外部的那个窗口（带菜单的），而文档窗口是真正的表示数据的窗口。

现在控制转移到了 COleClientItem::XOleIPSite::GetWindowContext()中。首先，客户条目会先建立一个位置矩形和一个裁剪矩形，这样内容对象就可以知道在哪里放它的定点窗口了。

COleClientItem::XOleIPSite::GetWindowContext()使用 COleClientItem::OnGetItemPosition()和COleClientItem::OnGetClipRect()这两个虚函数来确定这些位置。

除了位置信息外，客户条目还需要告诉内容对象如何将它的定点窗口与容器窗口关联起来。内容对象为完成这个功能，需要一个 OLEINPLACEFRAMEINFO 类型的结构以及 IOleInPlaceFrame 和 IOleInPlaceUIWindow 这两个接口。下面让我们来看看如何提供这些信息给内容对象。

在这个过程中需要使用 COleClientItem 的另外一个成员函数 OnGetWindowContext()，这个函数也是个虚函数。这个函数功能是：（1）给容器提供快捷键和窗口句柄的信息；（2）建立窗口层次结构，使得 COleClientItem::XOleIPSite::GetWindowContext()能够返回正确的 IOleInPlaceFrame 和 IOleInPlaceUIWindow 指针，使它们能对应正确的窗口。

为了提供框架窗口的额外信息，OLE 定义了一个叫 OLEINPLACEFRAMEINFO 的结构：

```
typedef struct tagOIFI
{
    UINT cb;
    BOOL fMDIApp;
    HWND hwndFrame;
    HACCEL haccel;
    UINT cAccelEntries;
} OLEINPLACEFRAMEINFO, *LPOLEINPLACEFRAMEINFO;
```

在 COleClientItem::OnGetWindowContext()中，MFC 使用了文档模板从上面的信息结构中获取快捷键的信息，并将容器的主框架窗口的句柄值设置为 hwndFrame。

接着，OnGetWindowContext()会建立窗口层次结构。如果容器是一个 MDI（多文档）应用程序，则 OnGetWindowContext()会创建一个指向最顶层的框架的指针（CMDIFrameWnd 对象），并将这个作为主框架。然后会创建一个指向活动的 CMDIChildWnd 对象的指针，并将其作为文档窗口。如果容器是一个 SDI（单文档）应用程序，则 OnGetWindowContext()会创建一个指向顶层的框架的指针（CFrameWnd 对象），将它作为主框架窗口，然后将文档窗口指针赋值为 NULL。

在调用完 OnGetWindowContext()后，COleClientItem 接下来需要做的事情是提供 IOleInPlaceFrame 和 IOleInPlaceUIWindow 这两个接口。MFC 使用一个叫 COleFrameHook 的函数来实现这个功能，这个函数实现和封装了 IOleInPlaceFrame 和 IOleInPlaceUIWindow 这两个接口。在创建了 OLEINPLACEFRAMEINFO 结构后，MFC 要做的就是将 IOleInPlaceFrame 接口与容器的框架窗口挂钩，将 IOleInPlaceUIWindow 与容器的 MDI 子窗口挂钩（如果是 MDI 应用程序的话）。

现在再回来看看 COleClientItem::OnGetWindowContext()函数。框架已经创建了指向容器的主框架和 MDI 子窗口的指针（如果是 MDI 应用程序），现在应该使用这两个接口了。为了实现 IOleInPlaceFrame 和 IOleInPlaceUIWindow，COleClientItem::XOleIPSite::GetWindowContext()创建了两个 COleFrameHook 的实例：一个基于外部的框架，一个基于 MDI 子窗口（如果应用程序是 MDI 的话）。COleFrameHook 的构造函数有两个参数：一个指向 CFrameWnd 类型的指针和一个指向 COleClientItem 的指针。COleFrameHook 还有几个很重要的成员变量：一个指向 COleClientItem 的指针（m_pActiveItem）、一个指向框架窗口的指针（m_pFrameWnd）

和一个窗口句柄（m_hwnd）。

在构造外部框架与 IOleInPlaceFrame 的关联的时候，COleFrameHook 的构造函数将 m_pActiveItem 赋值为活动的 COleClientItem，将 m_pFrameWnd 赋值为外部框架，将 m_hwnd 赋值为外部框架的窗口句柄。在 MFC 的 CFrameWnd 类中，有一个 COleFrameHook 的成员叫 m_pNotifyHook。当框架（framework）构造外部框架（frame）时，它将 m_pNotifyHook 赋值为新创建的 COleFrameHook 对象。

在构造 MDI 子窗口与 IOleInPlaceUIWindow 的关联的时候，COleFrameHook 的构造函数将 m_pActiveItem 赋值为活动的 COleClientItem 对象，将 m_pFrameWnd 赋值为 MDI 子窗口，然后将 m_hwnd 赋值为 MDI 子窗口的句柄。它也将 m_pNotifyHook 赋值为新创建的 COleFrameHook 对象。

容器和内容对象之间是通过 OLE 接口进行通信的。内容对象希望得到两个接口指针，而不是两个 MFC 对象。也就是说，内容对象希望通过 IOleInPlaceFrame 接口看见基于外部框架的 COleFrameHook 对象，以及通过 IOleInPlaceUIWindow 接口看到基于 MDI 框架的 COleFameHook 对象。COleClientItem::XOleIPSite::GetWindowContext()使用 GetInterface()来取得每个新创建的 COleFrameHook 对象的相应的接口（GetInterface()这个快捷的方法在所有从 CCmdTarget 派生的实现了接口映射表的类中都存在）。得到这几个接口指针后，客户就可以将这些接口指针返回给内容对象。

如上所述，在容器和内容对象之间建立窗口的上下文环境确实需要做很多的工作。

现在控制转移到了内容对象中。我们现在是在执行一个动作。为了搞清楚这一点，我们先看看现在的调用栈：

```
COleServerItem::;OnDoVerb()
    which begat COleServerItem::OnShow()
        which begat COleServerDoc::ActivateInPlace()
```

当窗口都关联起来后，内容对象接下来需要做的就是将共享菜单拼凑起来。前面说过 ActivateInPlace()创建了一个 COleIPFrameWnd 对象。ActivateInPlace()调用了定点框架窗口的 BuildSharedMenu()函数。此处就是将菜单关联的地方。BuildSharedMenu()取得活动文档的文档模板，然后根据这个模板，BuildSharedMenu()从中取得内容对象的定点菜单资源。接着 BuildSharedMenu()会调用 Windows API 函数 ::CreateMenu()来创建一个菜单句柄。

还记得 COleFrameHook 类吗？现在我们来看看如何使用这个类。现在内容对象已经有了一个指向容器的 IOleInPlaceUIWindow 接口的指针。IOleInPlaceUIWindow 有一个成员函数叫 InsertMenu()，用来创建一个共享菜单。（IOleInPlaceUIWindow 由 COleFrameHook 实现。）

建立菜单（menu）和工具条（toolbar）

内容对象调用 IOleInPlaceUIWindow::InsertMenus()，该函数在容器的 COleFrameHook::XOleInPlaceFrame::InsertMenus()中结束。真正的菜单协商是在 COleClient 条目中发生的。现在我们看看 COleFrameHook 对象。我们是如何得到 COleClientItem 对象的呢？COleFrameHook 的其中一个成员是活动的 COleClientItem，它在 COleFrameHook 对象被构造的时候被赋值。COleFrameHook::XOleInPlaceFrame::InsertMenus()使用活动的 COleClientItem，并且调用 COleClientItem::InsertMenus()往空的菜单里面插入容器应用程序的菜单。

MFC 对 InsertMenus()的实现是插入定点容器菜单。也就是说，这个函数用来插入 File、

Container 和 Window 等菜单集合。这个菜单资源使用 CDocTemplate::SetContainerInfo()与容器的文档模板建立关联。

在共享菜单框架构造起来之后，BuildSharedMenu()调用 AfxMergeMenus()来合并容器和服务器菜单。

在容器创建了菜单后，服务器就可以往里面插入自己的选项了。COleIPFrameWnd::BuildSharedMenu()使用 AfxMergeMenus()将服务器的菜单加入到共享菜单中。

在返回之前，COleIPFrameWnd::BuildSharedMenu()会调用 OLE 的 API 函数 OleCreateMenuDescriptor()来创建一个 OLE 菜单描述符。这个是一个 OLE 提供的数据结构，用来描述菜单。OLE 在分发菜单消息和命令的时候需要用到这个信息。COleIPFrameWnd 在它的 m_hOleMenu 成员中保存这个结构。

接着，内容对象会将它的工具条放入容器中，并且调用 OnSetItemRects()在容器的框架窗口中定位定点编辑框架窗口。

然后内容对象会调用容器的 IOleInPlaceFrame::SetActiveObject()，将它的 IOleInPlaceActiveObject 接口传给客户。内容对象的 IOleInPlaceActiveObject 接口为下面几者之间提供了一条相互通信的通道：（1）内容对象；（2）服务器应用程序的最外层框架窗口；（3）容器的文档窗口。这个接口能处理下面一些用户接口的事务：翻译消息；设置框架窗口的状态（激活或是使无效）；设置文档窗口的状态（激活或是使无效）；在窗口需要调整大小的时候通知它；管理无模式对话框。

内容对象会显示它的工具条和所有的无模式对话框，并且调用 COleServerDoc::ResizeBorder()。这个函数会根据窗口的大小调整工具条等用户界面元素的大小与位置。

下一步，内容对象会调用容器的 IOleInPlaceFrame::SetFrame()函数在容器中设置定点菜单。

在让内容对象可见之前，服务器会调用容器的 IOleClientSite::ShowObject()函数。在容器这端，COleClientItem::XOleClientSite::ShowObject()会调用 COleClientSite::ShowObject()。这个函数会使定点条目在容器窗口中被显示出来。

COleServerDoc::ActivateInPlace()执行的最后一个操作是调用框架的 ShowWindow()函数显示定点框架。

定点激活小结

上面是 MFC 实现定点激活的过程。正如我们所见的，MFC 已经帮我们做了几乎全部的工作。如果这些代码都要自己写的话，那可是一份工作量很大的活。

在结束 OLE 文档的讨论之前，我们再来看看下面两个主题：（1）当使一个条目无效时会发生什么？（2）如何将结构化存储集成到 MFC 的序列化模型中？

使条目无效

在一个定点激活条目的边界外部点击鼠标，通常会使它变得无效（不处于激活状态）。MFC 容器通过调用活动条目的 Close()函数来使定点激活条目无效。这里值得我们研究一下，因为此处 OLE 的结构化存储和 MFC 的序列化机制协同工作。

MFC 对 OLE 文档服务器提供支持的一个很有用的功能就是如果你使用了 MFC 的序列化模型，那么存储数据到容器的文件中就是很方便的事情了。也就是说，如果你的文档使用了 MFC 的标准序列化机制，则要实现文档的 COleClientItem 对象的存储就不需要自己写任何代码了。

回忆一下 MFC 的序列化机制的运行方式。框架调用文档的 Serialize()函数，用简洁的话来说就是："现在要保存数据了，你希望做些什么？" MFC 给 CArchive 对象传递一个指针。MFC 根据不同的模式替换这个 archive。例如，如果服务器是运行于单独模式，则 archive 就基于应用程序创建的或是打开的文件；如果服务器嵌入到别的容器中运行，则 archive 就基于一个由容器提供的 OLE 结构化存储对象。应用程序本身并不需要考虑它是运行在单独模式下还是嵌入式下，MFC 会根据运行模式传递一个基于恰当的存储对象的 archive。下面我们来看一下如何使用 COleClientItem::Close()来使一个活动的条目变无效。

COleClientItem::Close()的内部机制

COleClientItem::Close()将一个 OLE 条目的状态从运行态转到装载完毕的状态。也就是说，这个条目的处理方法还是在内存中，但是服务器没有运行。一个 OLE 容器会用以下 3 种方式之一来调用 Close()：

- OLECLOSE_SAVEIFDIRTY——保存 OLE 条目。
- OLECLOSE_NOSAVE——不保存 OLE 条目。
- OLECLOSE_PROMPTSAVE——提示用户是否要保存 OLE 条目。

整个过程的第一步就是通过内容对象的 IOleObject 接口调用 Close()。这个调用完毕之后，在服务器这端调用 COleServerDoc::XOleObject::Close()后就执行完毕了。COleServerDoc::XOleObject::Close()仅仅是将操作委托给文档的 OnClose()函数。如果嵌入的数据是脏的（也就是被修改过），则 OnClose()会调用 COleServerDoc::SaveEmbedding()。

COleServerDoc::SaveEmbedding()函数

COleServerDoc::SaveEmbedding()所做的全部工作就是在文档中保存嵌入的数据。服务器调用容器的 IOleClientSite::SaveObject()函数来保存数据。客户通过它的 IOleObject 接口调用 QueryInterface()来查询内容对象的 IPersistStorage 接口。然后客户通过 IPersistStorage::IsDirty()来查询对象是否被修改过。在服务器这端，内容对象只要检查 document-dirty 这个标志（这个标志用函数 CDocument::SetModifiedFlag()来设置）。

如果内容对象是脏的（被修改过），则客户会调用 OLE API 函数 OleSave 来将内容对象保存到容器的存储对象中。下面是这个函数的原型：

```
HRESULT OleSave(
   IPersistStorage * pPS,
   IStorage * pStg,
   BOOL fSameAsLoad
);
```

OleSave()函数有 3 个参数：（1）一个属于存储对象的 IPersistStorage 接口指针；（2）一个 IStorage 指针，用来存放数据；（3）一个用来标识对象是否是从 IStorage 接口中装载的标志。COleServerDoc::SaveEmbedding()会传递内容对象的 IPersistStorage 接口指针、用来保存内容对象数据的存储对象和一个标志，该标志用来表示是否将它保存到装载它的同一个存储对象。

OleSave()会执行以下几个步骤：

1. 调用 IPersistStorage::GetClassID()取得 CLSID，
2. 使用 WriteClassStg()将 CLSID 写入到存储对象。
3. 调用 IPersistStorage::Save()保存对象。
4. 如果存储过程中没有错误，则调用 IPersistStorage::Commit()来提交这个改变。

在容器调用 OleSave()后，第一步是调用 COleServerDoc::XPersistStorage::GetClassID()。GetClassId()用于从类厂中取得内容对象的类 ID。

下一步是调用 COleServerDoc::XPersistStorage::Save()。这个函数调用 COleServerDoc::OnSaveEmbedding()。OnSaveEmbedding()然后调用 COleServerDoc::OnSaveDocument()，该函数会接着调用 COleLinkingDoc::OnSaveDocument()，COleLinkingDoc::OnSaveDocument()接着调用 COleDocument::OnSaveDocument()。如果文档是保存在一个存储对象中（也就是说如果文档的 m_lpRootStorage 为非 NULL），则 COleDocument::OnSaveDocument() 调用 SaveToStorage()。这个调用从 COleLinkingDco::SaveToStorage()开始。该函数保存对象的类 ID，并使用 OLE API 函数 WriteClassStg()将这个类 ID 写入到存储对象中。COleLinkingDoc::SaveToStorage()调用基类的实现，即 COleDocument::SaveToStorage()。

下面是 MFC 使用 OLE 结构化存储接口来创建一个 archive。COleDocument::SaveToStorage()在栈上声明一个 COleStreamFile 对象。COleStreamFile 封装了一个 OLE IStream 接口指针。COleStreamFile 对象可以与 CFile 对象采用相似的操作。（也就是说可以使用它们创建一个 CArchive 对象。）在 SaveToStorage()声明了一个 COleStreamFile 对象后，它使用 COleStreamFile::CreateStream()来打开一个流文件。在 COleStreamFile 创建了一个流对象后，就能够使用它来构造一个 CArchive 对象了。

下面是 SaveToStorage()函数所做的工作：它使用内容对象的子存储对象的流对象来创建一个 CArchive 对象。接着文档进行它平常的序列化过程。文档并不需要关心 archive 的类型，它只要按照平常的做法进行即可。

这就是内容对象如何保存它的数据。下面我们看看容器的文档。

保存容器的文档

内容对象在每次关闭的时候（通常通过定点取消激活（in-placed eactivation）），都将它的数据保存到子存储对象中。不过，还有一个问题是保存容器的文档。

当 OLE 文档内容对象在存储它们自己的数据时，OLE 文档容器也要存储有关它内部的内容对象的信息。下面让我们循着序列化的过程来看看这个存储过程。

OLE 文档容器也采用和普通的 MFC 文档一样的序列化机制。如果用户通知应用程序保存文档（一般是从 File 菜单下选择 Save），这样就会去调用文档的 OnSaveDocument()函数。

与内容对象一样，OnSaveDocument()在文档的根存储对象下面创建一个流对象，然后基于这个流对象构造一个 CArchive 对象。接着 OnSaveDocument()调用 SaveToStorage()函数。COleDocument::SaveToStorage()被调用，然后保存文档的内容。

除了保存文档自己本身的内容外（比如，Scribble 保存了一个 stroke 对象链表），文档还需要保存有关它内部的 COleClientItem 对象的信息。MFC 对 COleDocument::Serialize()的默认实

现有两个功能：（1）在 CArchive 对象中存储 COleClientItem 对象（实际上是 OLE 结构化存储对象）的个数；（2）遍历对象链表，分别通知每个对象执行序列化。所以，如果在容器中的对象中包含了一些特殊的信息（比如一个矩形的位置），则下次导入这个文件的时候，这些信息还能够恢复。除了一些特殊的信息，每个 COleClientItem 还有一些数据需要保存：（1）条目的个数；（2）显示方面的属性；（3）一个标志，表示是否在导入的时候创建一个名字对象（moniker）；（4）当前默认的绘制性质。当这些对象被写入存储对象后，COleClientItem::Serialize()调用 COleClientItem::WriteItem()函数。这个函数会顺序提交存储对象中的条目。

装载 OLE 文档

装载 OLE 文档的过程和保存的过程基本类似，只不过是做相反的操作。在基于 MFC 的应用程序中，装载 OLE 文档也被集成到序列化模型中来完成。

当选择了 File 菜单中的 Open 选项时，MFC 会给 CWinApp::OpenDocumentFile()发送一条消息。该函数最后会调用文档的 OnOpenDocument()成员。因为文档是从 COleDocument 中派生的，所以框架会调用 COleDocument::OnOpenDocument()。COleDocument:: OnOpenDocument()使用 OLE 的 API 函数 StgOpenStorage()来打开文档。与 StgCreateDocFile()一样，StgOpenStorage()也会给调用者返回一个 IStorage 接口指针。OnOpenDocument()将根存储成员（m_lpRootStorage）设置为新打开的存储对象。现在文档就有了一个它能使用的活动的 IStorage 接口指针。

COleDocument::OnOpenDocument()首先使用 LoadFromStorage()函数从文档中读取条目。LoadFromStorage()执行与 SaveToStorage()相反的操作。与 SaveToStorage()一样，LoadFromStorage()也需要基于文档的根存储对象创建一个 COleStreamFile 对象。然后就可以利用这个 COleStreamFile 对象创建 CArchive 对象。这个对象是 MFC 的序列化模型需要使用的。

接着 OnOpenDocument()通过使用 CArchive 对象（实际上是一个 IStream 接口指针）调用文档的 Serialize()函数来装载文档。在装载完所有的应用程序所需要的数据后（例如，如果使用一个字处理器来支持 OLE 文档，则这个文档就必须装载所有字处理的数据），文档的 Serialize()函数调用 COleDocument::Serialize()函数来装载客户条目。

当 MFC 序列化一个 COleDocument 的时候，都会将文档中的客户条目个数保存。现在，如果 COleDocument 需要客户条目的个数，以便能够装载所有的客户条目，则它就可以使用保存的这个数字了。COleDocument::Serialize()会去序列化每一个条目：

```
// read number of items in the file
DWORD dwCount;
ar >> dwCount;

// read all of them into the list
while (dwCount--) {
   CDocItem* pDocItem;
   ar >> pDocItem;      // as they are serialized, they are added!
}
```

当然，在序列化的时候，MFC 会去调用每一个条目的 Serialize()函数。客户条目首先调用基类的 Serialize()函数：

```
COleClientItem::Serialize(ar)
```

COleClientItem::Serialize()会调用 CDocItem::Serialize()函数将条目放到文档中。另外，条目还必须将它以前保存的变量读取到它的成员变量中：条目的个数、缓存的显示属性、名字对象标志和默认的显示属性。接着 COleClientItem::Serialize()调用 ReadItem()来装载 OLE 对象。ReadItem()打开客户条目的子存储对象（这个存储对象在条目被插入的时候创建），然后将 IStorage 指针赋给 COleClientItem::m_lpStorage 成员。然后 ReadItem()调用 OleLoad()使得内容对象进入导入的状态。OleLoad()执行以下这些操作：

1. 自动转换对象（如果这个对象在注册表中注册过的话）。
2. 调用 IStorage::Stat()从一个打开的存储对象中取得 CLSID。
3. 调用 CoCreateInstance()创建一个管理程序的实例。默认状态下，MFC 使用默认的管理程序——OLE32.DLL。
4. 调用 IOleObject::SetClientSite()将客户地址指针传递给对象。
5. 调用 QueryInterface()函数查询对象的 IPersistStorage 接口。如果成功，则 OleLoad()调用 IPersistStorage::Load()函数。
6. 对请求的接口调用 QueryInterface()并且返回这个接口。

COleClientItem::ReadItem()会请求对象的 IUnknown 接口，然后利用这个接口去查询内容对象的 IOleObjectPointer 接口。如果成功，ReadItem()会在 COleClientItem::m_lpObject 成员中保存这个接口指针。

COleClientItem::Serialize()调用 COleClientItem::FinishCreate()完成连接客户条目的功能。

最后，如果条目是嵌入的而不是链接的，则 COleClientItem::Serialize()会取得对象的 IMoniker 接口。

文档使用上面的方法装载每一个客户条目，直到整个文档装载完毕（并且每一个内容对象都导入状态）。当用户在条目上双击或是激发别的动作时，容器就可以调用对象的 IOleObject 接口来响应这些命令。

结　语

从某种意义上来说，OLE 就是复合文档，以前老的技术是基于动态数据交换的，这种方法有很大的效率以及灵活性的问题。现在，OLE 文档的基础是一组 COM 接口，基本上这些问题都被解决了。

与别的 OLE 技术类似，OLE 文档很大程度上也是作为一种标准存在的。MFC 只不过是一种实现它的方法。然而，如果你想完全自己去实现 OLE 文档接口，那将会做大量重复性的工作，因为这些工作微软都已经帮我们做好了。

OLE 文档以及 MFC 对它的实现实际上可以写整整一本书。现在我们只是接触最重要的部分，也是最有意思的部分。我们还有很多的内容没有讨论，例如，对于链接和名字对象，我们就没有做深入的探讨。想要知道更多的信息，请读者去看看下面这些 CPP 文件：OLESVR1.CPP、OLESVR2.CPP、OLECLI1.CPP、OLECLI2.CPP、OLECLI3.CPP、OLEDOC1.CPP、OLEDOC2.CPP。在这些文件当中也许你会发现很多很有意思的东西。

CHAPTER

14 MFC 与自动化

现在，随着软件开发者越来越意识到自动化（automation）的便利，这项 OLE 技术便受到了越来越多的关注。也许现在你可

能还认为用不着自动化这项技术，但是将来一定离不开它。在这一章中，我们将分两个部分来讲述 MFC 和自动化：第一部分介绍 OLE 自动化的原理——也就是在 COM 的层次上了解 IDispatch 这个接口的工作原理；第二部分介绍 MFC 对自动化的实现。

在 COM 这个层次上理解自动化很重要，因为这可以帮助我们理解 IDispatch 接口（处理自动化的主要接口）。这个接口从本质上来说也只是一个 COM 接口，通过实现这个接口，就可以在我们的项目中增加自动化的特性。

自动化的历史

自动化的发展由于有以下的需求：（1）自动化应用程序需要使用该程序之外已有的工具；（2）在高层使用通用的工具（比如 Visual Basic）将几个应用程序集成到一起。在没有 OLE 之前，很多应用程序都能以支持自动化（不是 OLE 自动化）的方式打开，并且通过以下两种方法之一的方式集成：（1）宏语言；（2）动态数据交换（DDE）。

首先，很多应用程序提供它们自己的宏语言。任何一个有经验的老手都应该知道每种主要的电子表格应用程序（Lotus 1-2-3、Borland Quattro 和 Microsoft Excel）都有自己的宏语言。这种语言在自己的应用程序内部非常有用，但是一出应用程序就没什么用处了。尽管这些宏语言使得这些应用程序支持自动化，但是它们通常都被限制在本应用程序中，并不能同时驱动多个应用程序。

也许你还记得有一种比较古老的数据交换技术叫动态数据交换，可以用来在应用程序之间共享数据，并且能调用另外一个应用程序中的函数。尽管现在 DDE 技术还存在，但是它速度太慢，而且很笨重，实现起来也很困难。

自动化的功能

自动化也是一种 OLE 特性，使用它可以用来解决宏语言的扩散（proliferating）（不同的应用程序必须使用自己的宏语言）和像 DDE 等数据交换协议的繁重工作等问题。但是，自

动化技术并不仅仅只是用来使应用程序自动化，它在别的领域也是一种很重要的技术。例如，自动化是 OLE 控件使用的核心技术。自动化也许是这样的一种技术：刚开始它看起来很复杂而且没什么用，但是使用它之后，你就会叹服它的方便，在任何能够使用的地方都想使用它了。

支持自动化的应用程序会将属性和方法提供给能够使用自动化的客户（比如 Visual Basic）。例如，如果你正在编一个 Visual Basic 的应用程序，而这个应用程序是一个自动化的客户，则它可以很方便地访问提供自动化的应用程序的自动化属性和方法。假设别人为你写了一个自动化的服务器，叫做“StockMaster”，它能够从纽约股票交易所得到某种股票的当前价格。这个软件有几个方法可以用来启动和停止操作，可能还有一些成员用来表示股票的当前价格。在支持自动化的应用程序中，你可以说这个对象提供了两个方法（启动和停止取得价格的操作）和一个属性（股票的当前价格）。

使用 Visual Basic 可以很方便地实现这样的一个对象。在 VB 程序的[general]部分，你可以这样声明这个对象：

```
dim StockInfo as object
```

为了创建这个自动化对象，你只要在 Visual Basic 程序的“Form”部分中插入下面代码：

```
StockInfo = CreateObject("StockMaster")
```

然后在初始化数据的地方加入下面这行代码：

```
StockInfo.StartAcquisition
```

接着在停止数据收集的地方加入下面的代码：

```
StockInfo.CurrentPrice
```

最后，在任何想要取得当前股票的价格的地方，只要取得代表当前价格的属性即可：

```
StockInfo.StopAcquisition
```

想像一下上面的方法是多么的方便。如果不是这样，要实现像 StockMaster 这样的功能，很多的底层工作需要我们去实现：为了取得数据我们必须遵守某些通信协议，这样就必须建立一个同纽约股票交易所连接的 RS-232 的通信程序。在这个程序的顶层，还必须实现一些代码，用来管理连接以及监控传输的数据。自动化就是所有这些底层细节的一个抽象，只要实现一个可编程的接口，这个接口可以用类似于 Visual Basic 的脚本语言实现。

自动化使得你可以将整个程序作为一个对象来处理。自动化为客户提供了一个很好的可编程的接口。如果需要调用 StockMaster 的函数，Visual Basic 的客户只要使用一个点来访问整个函数（StockInfo.StartAcquisition）。一个 Visual Basic 程序也可以用同样的方法访问它的属性：使用点来进行反引用（dereferencing）属性（StockInfo.CurrentPrice）。

下面是一种比较 COM 接口和分发接口的很好的方法。使用 COM 接口是用一种限制严格的方法来使用 COM 对象，而使用自动化是用一种相对放松的方法使用 COM 对象。像 Visual Basic 之类的语言对于像类型检查和参数传递等操作来说就没有那么严格。一个 Visual Basic 程序可以使用一些参数（无论它们是什么类型的），然后将它们传递给一个自动化对象。当调用传递到自动化对象后，这个对象会去将参数包解开，然后得到参数的类型，确定它们是否符合这个函数调用的上下文环境，最后将它们放到一个限制严格的、面向 C++的栈中，为函

数调用准备。

当你看到自动化技术在运转的时候，也许会突然意识到这是一种集成多个应用程序的一种方法。也许会突然想到可以为广大的软件学习者写一个软件模型（VB 的使用者远远比 VC 使用者要多）。如果你想使你的程序能够被大部分的程序开发者使用，则最好能够支持自动化。

使用 MFC 实现自动化应用程序

如果你决定使用 MFC，则自动化是一个很容易添加的特性——特别是当你从创建一个应用程序开始，只要在使用 AppWizard 的时候按下“Automation－Yes，please”按钮即可。（微软最后改变了“No”标签，现在变为“Automation－No，thank you”。）就算不是从创建一个应用程序开始，在一个已经存在的应用程序中添加自动化属性也不是一件很困难的事情。如果需要了解更详细的信息，请参见 Mary Kirtland 在《VC++ Professional》（1995 年 1 月）上的有关 OLE 自动化的文章。在 VC 中，利用 ClassWizard 可以将你所需要的自动化的属性和方法添加到应用程序当中去。

自动化的工作机制

自动化运行起来后我们现在还不是很了解，有时候甚至会觉得它很神奇，是什么使得两个应用程序能够如此轻松地进行通信呢？

与别的 OLE 特性一样，自动化也是通过组件对象模型（COM）来实现的。COM 可以说是代表了一种编程的方式。自动化处于 COM 的最顶层，它实现了一个特殊的 COM 接口叫 IDispatch。

COM 和 OLE 让人容易产生糊涂的其中一个方面是 COM 和 OLE 总的来说只是一个标准，它描述了如何在你的应用程序当中实现某一些特性。OLE 定义的功能大约有 15%的已经默认地由微软实现了，例如，你可以通过调用一些微软提供的全局函数直接得到结构化存储（Structured Storage）和内存分配（memory-allocation）的服务（例如调用 StgCreateDocFile() 实现结构化存储以及调用 CoGetMalloc()来管理 OLE 内存）。

然而，微软定义的另外 85%的服务并没有默认地实现。这些服务需要软件开发者根据需要自己来实现。没有默认实现的其中一个服务就是自动化。这是因为自动化这项特性是存在于应用程序这个域里面的（相对于操作系统）。所以，如果想要在应用程序中加入自动化的特性，则需要自己实现它（也可以让别人来实现它，比如一个框架）。

COM 接口与自动化

在深入讨论自动化技术之前，先让我们来对比一下 COM 接口与自动化的区别，因为这两者目的都是让多个应用程序之间可以共享对方提供的功能。

复习 COM 接口

在 COM 中，所有的功能都是通过 COM 接口来提供的。COM 接口就是定义了某种服务的一组函数。正如 11 所述，OLE 通过一个 COM 接口来管理内存。CoGetMalloc()可以从任务分配器中取得一个指针。在取得 IMalloc 指针后，就可以通过这个接口使用 9 个函数。这些函数包括 QueryInterface()、AddRef()、Release()、Alloc()、Free()、Realloc()、GetSize()、DidAlloc()和 HeapMinimize()。根据 COM 的调用-使用-释放机制，先调用 CoGetMalloc()取得接口指针，然后使用接口函数，最后释放这个接口。下面的代码片断展示了这个过程。函数 AllocMem()使用 OLE 函数 CoGetMalloc()取得一个 IMalloc 指针，接着使用 IMalloc 接口分配了一块内存，然后再释放它：

```
HRESULT AllocMem() {
     LPMALLOC pMalloc;
     LPSTR lpsz;

     ::CoGetMalloc(MEMCTX_TASK, &pMalloc);
     if (lpsz = LPSTR(pMalloc->Alloc(256)))
        result = NOERROR;
     else
        result = ResultFromScode(E_OUTOFMEMORY);

     if(lpsz)
        pMalloc->Free(lpsz);

     pMalloc->Release();

     return result;
}
```

所以一个接口仅仅是一个 OLE 客户能够访问的函数表。在第 11 章中讲过 IMath 接口：IMath 是一个客户接口，它包括 2 个函数(除了 IUnknown 的 3 个函数)——Add()和 Subtract()。这是一个相当简单的接口，但是这个接口已经能足够清楚地体现了 COM 接口与自动化之间的差异。下面是 IMath 的接口定义：

```
interface IMath
{
    // *** IUnknown methods ***
    virtual HRESULT QueryInterface(REFIID riid, LPVOID FAR* ppvObj) = 0;
    virtual ULONG AddRef() = 0;
    virtual ULONG Release() = 0;

    // *** IMath methods ***
    virtual HRESULT Add(INT, INT, LPLONG) = 0;
    virtual HRESULT Subtract(INT, INT, LPLONG) = 0;
};
```

任何 COM 对象都可以实现这个接口（还可以和别的接口一起实现）。在第 11 章中还描述了如何使用一个叫 CoMath 的 COM 类来实现这个接口。

如果需要使用这个接口，一个 COM 客户只要创建一个对象的实例然后调用接口的函数

即可。下面的代码实现这个功能：

```
{
...
   IMath *pMath;
   HRESULT hr;

   hr = ::CoCreateInstance(CLSID_CoMath, NULL,
                           CLSCTX_INPROC_SERVER,
                           IID_IMath,
                           (VOID FAR **)&pMath);

    LONG lSum, lDifference;

    hr = pMath->Add(10, 50, &lSum);
    hr = pMath->Subtract(150, 50, &lDifference);
...
}
```

接口在一个头文件中定义，而这个头文件在编译客户程序的时候会被包含到源文件中，所以客户程序能够在编译的时刻知道接口的结构。如果创建了一个实现 IMath 接口的对象的实例，就可以调用 IMath 接口中的函数了。当编译器碰到任何有关 IMath 接口函数的调用时，编译器根据接口 vtable（虚表）中的偏移地址可以知道如何调用这个函数。图 14-1 展示了这个结构。

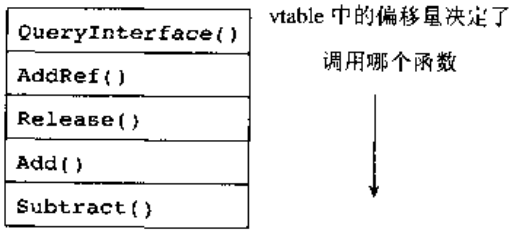


图 14-1 IMath 接口

这里的关键就是 COM 接口使用早绑定（early binding）。也就是说，编译器事先知道（在运行之前）接口的 vtable，也知道接口中每个函数的位置。这些信息已经足够使得编译语言使用早绑定技术。但是，还有很多用户使用像 Visual Basic 这样的解释型语言（interpreted language）。由于解释型语言需要在运行时进行函数绑定（迟绑定），所以这个机制就会失败。这就需要引入自动化和一个新的接口叫 IDispatch。

IDispatch 接口：自动化技术的关键

自动化需要某种迟绑定的机制，因为使用自动化的客户不能使用标准的 COM 接口使用的早绑定机制。使用普通 COM 接口的客户会进行严格的类型检查，事先知道所有的参数，在函数调用之前就已经建立了堆栈的框架。而自动化的运行机制跟这个有所区别。

假设某个自动化的客户需要从某个自动化服务器中调用 Foo()函数。由于解释型语言一般比 C++有更加灵活的编程环境，所以自动化也就使用一种更加松散的方法在应用程序之间共享功能。在自动化中，也许客户会这样说："OK，我觉得在另外一端有一个函数叫 Foo()，并且我还有一些参数。对象先生，你能帮我做些什么吗？"现在，接下来的工作就转给了对象，它要做（1）判断是否确实有个函数叫 Foo()；（2）计算参数；（3）判断参数是否有效；（4）进行函数调用。在 OLE 中，使用 IDispatch 接口来完成这些功能。IDispatch 接口包含 4 个函数（将会很快在下面看到）。在这 4 个函数中，Invoke()是用来进行迟绑定的函数。

现在假设一个叫 DispMath 的 COM 对象实现了 IDispatch 接口，所以像 Visual Basic 之类

的客户就可以在 Add()和 Subtract()这两个函数上进行迟绑定。图 14-2 表示了这个关系。

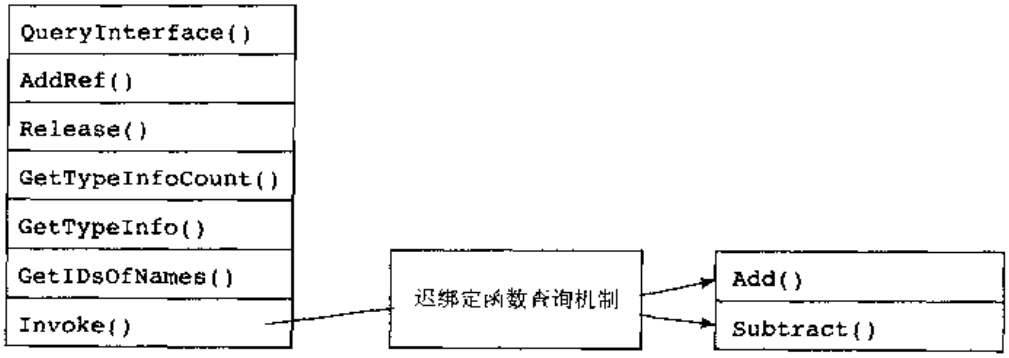


图 14-2 IDispatch（通过 DispMath 实现）

从上图可以看出，IDispatch 也只是一个 COM 接口。它的最上面放的是 3 个标准的 IUnknown 接口函数，后面跟着 IDispatch 本身的函数。但是，DispMath 接口里面不是直接通过客户 vtable 来实现 Add()和 Subtract()，而是使用另外一种机制来管理参数传递以及解决函数指针的问题。

在自动化的环境中，客户可以向 DispMath 请求 IDispacth 接口指针。由于 DispMath 实现了 IDispatch 接口，所以它会返回一个 IDispatch 接口指针给请求的客户。客户得到了 IDispatch 接口后，就可以通过调用 IDispatch::Invoke()来进行函数调用。

在讨论 Invoke()机制前，让我们先来看看 IDispatch 接口。下面是 IDispatch 接口的定义：

```
interface IDispatch : IUnknown {
   virtual HRESULT GetTypeInfoCount(unsigned int *pcTInfo) = 0;
   virtual HRESULT GetTypeInfo(unsigned int iTInfo, LCID lcid,
                               ITypeInfo *ppTInfo) = 0;
   virtual HRESULT GetIDsOfNames(REFIID riid, OLECHAR **rgszNames,
                                 unsigned int cNames, LCID lcid,
                                 DISPID *rgdispid) = 0;
   virtual HRESULT Invoke(DISPID dispID, REFIID riid, LCID lcid,
                          unsigned short wFlags,
                          DISPPARAMS *pDispParams,
                          VARIANT *pVarResult, EXCEPINFO *pExcepInfo,
                          unsigned int *puArgErr) = 0;
};
```

注意一下 IDispatch 包括 4 个函数：GetTypeInfoCount()、GetTypeInfo()、GetIDsOfNames()和 Invoke()。在 MFC 的实现中，GetIDsOfNames()和 Invoke()是最重要的两个函数。（实际上，到目前为止，MFC 对 GetTypeInfo()和 GetTypeInfoCount()还没有实现。）我们先从 IDispatch::Invoke()以及 IDispatch::GetIDsOfNames()开始，然后再讨论类型信息。

IDispatch::Invoke()函数

现在我们来看一下 Invoke()函数，这个函数是实现自动化技术的关键所在。Invoke()是比较复杂的，它有 8 个参数。第一个参数是 dispID，用来标识调用的自动化方法或者属性。DISPID 是 32 位的整数，自动化服务器用它来标识它的属性和方法。DISPID 类似于一个系统句柄，作为开发人员并不需要关心这个值到底是多少，只要得到这个变量，然后利用它和系统进行

通信即可。下面我们将更加详细地讨论 DISPID。

第二个参数是指向一个接口 ID 的引用，这个参数是保留的，值永远都是 IID_NULL。第三个参数是 lcid，它的表示解释参数所使用的当前上下文环境，不支持多语言的应用程序可以忽略这个参数。

因为 Invoke()既可以调用函数，又可以访问属性，所以它需要使用某种方法来区分调用的不同情况。它的第四个参数 wFlags 就是用来标识调用的属性。它可以是下面 4 个值中的一个：DISPATCH_METHOD、DISPATCH_PROPERTYGET、DISPATCH_PROPERTYPUT 以及 DISPATCH_PROPERTYPUTREF。DISPATCH_METHOD 表示客户需要进行一个函数调用。DISPATCH_PROPERTYGET 或 DISPATCH_PROPERTYPUT 表示客户想要获取或设置某个自动化属性。而 DISPATCH_PROPERTYPUTREF 表示客户想要通过一个引用而不是一个值来改变某个属性。

无论函数调用是迟绑定还是早绑定，都需要能将将参数正确地传递到函数。Invoke()的第五个参数就是用来完成这个功能的。DISPPARAMS 结构描述了一个参数链表，这些参数的类型由另外一个叫 VARIANT 的结构来描述。下面是 DISPPARAMS 结构的定义：

```
typedef struct FARSTRUCT tagDISPPARAMS{
    VARIANTARG FAR* rgvarg;            // Array of arguments
    DISPID FAR* rgdispidNamedArgs; // Dispatch IDs of named arguments
    unsigned int cArgs;              // Number of arguments
    unsigned int cNamedArgs;         // Number of named arguments
} DISPPARAMS;
```

自动化需要在运行时绑定函数，而不是在编译时。这也就意味着数据不可能提前知道。因此，IDispatch 使用了一种自描述的（self-describing）数据类型叫 VARIANT。VARIANT 结构的第一个域是一个无符号短整型，表示在这个结构中保存的数据的类型。一个 VARIANT 结构可以表示 25 种数据格式的其中任何一种。程序清单 14-1 是 VARIANT 的定义。

程序清单 14-1 VARIANT 数据类型

```
typedef struct tagVARIANT  {
   VARTYPE vt;
   unsigned short wReserved1;
   unsigned short wReserved2;
   unsigned short wReserved3;
   union {
      short         iVal;             /* VT_I2                */
      long          lVal;             /* VT_I4                */
      float         fltVal;           /* VT_R4                */
      double        dblVal;           /* VT_R8                */
      VARIANT_BOOL  bool;             /* VT_BOOL              */
      SCODE         scode;            /* VT_ERROR             */

      CY            cyVal;            /* VT_CY                */
      DATE          date;             /* VT_DATE              */
      BSTR          bstrVal;          /* VT_BSTR              */
      IUnknown      FAR* punkVal;     /* VT_UNKNOWN           */
      IDispatch     FAR* pdispVal;    /* VT_DISPATCH          */
      SAFEARRAY     FAR* parray;      /* VT_ARRAY|*           */
      short         FAR* piVal;       /* VT_BYREF|VT_I2       */
```

```
        long          FAR* plVal;       /* VT_BYREF|VT_I4       */
        float         FAR* pfltVal;     /* VT_BYREF|VT_R4       */

        double        FAR* pdblVal;     /* VT_BYREF|VT_R8       */
        VARIANT_BOOL  FAR* pbool;       /* VT_BYREF|VT_BOOL     */
        SCODE         FAR* pscode;      /* VT_BYREF|VT_ERROR    */
        CY            FAR* pcyVal;      /* VT_BYREF|VT_CY       */
        DATE          FAR* pdate;       /* VT_BYREF|VT_DATE     */
        BSTR          FAR* pbstrVal;    /* VT_BYREF|VT_BSTR     */
        IUnknown FAR* FAR* ppunkVal;    /* VT_BYREF|VT_UNKNOWN  */
        IDispatch FAR* FAR* ppdispVal;  /* VT_BYREF|VT_DISPATCH */
        SAFEARRAY  FAR* FAR* parray;    /* VT_ARRAY|*           */
        VARIANT       FAR* pvarVal;     /* VT_BYREF|VT_VARIANT  */

        void          FAR* byref;       /* Generic ByRef        */
    };
} ;
```

使用 VARIANT 结构来传递数据会变得非常简单，因为所有的参数都是类似的。但是，这也意味着在自动化的客户和服务器两端都必须添加一些额外的工作：客户必须将参数打包到 VARIANT 中，而服务器必须将参数从这个包里面解开来；而服务器也必须能够将参数打包成 VARIANT 结构，并且客户也能够解开这个包。幸运的是，有一系列的 OLE 函数可以帮助我们来做这些工作。

第六个参数是存放函数的结果也就是返回值，它也是一个 VARIANT 类型。

最后两个参数用来处理错误情况以及错误代码。OLE 定义了一个处理异常的结构，Invoke()可以在出现异常的情况下（比如客户使用了一个无效的 DISPID）填充这个结构。

现在对于 IDispatch 只剩下一个问题了。Invoke()只有通过 DISPID 才能知道需要访问哪个方法或是属性。在 Visual Basic 中，是使用一个对客户很友好的名字来访问函数，但是根据这个名字怎么得到 DISPID 呢？IDispatch 接口专门有一个函数叫 GetIDsOfName()来完成这个功能。

IDispatch::GetIDsOfNames()函数

自动化客户可以使用 IDispatch::GetIDsOfNames()函数将一个用户友好的名字转换为一个 DISPID。GetIDsOfNames()将一个分发成员（dispatch member）（一个属性或是方法）和任意一组参数的名字映射为相对应的一组 DISPID。也就是说，客户传递给 GetIDsOfNames()函数一个可读性很好的名字（也就是自然语言的名字）和一个空的 DISPID 数组，GetIDsOfNames()会使用相对应的值填充这个 DISPID 数组。接着客户就可以使用得到的 DISPID 来调用 IDispatch::Invoke()。

手工实现 IDispatch 接口

IDispatch 中需要实现的两个很重要的函数是 Invoke()和 GetIDsOfNames()。要实现这两个函数，需要经过以下这些步骤：

1. 为你的方法和属性分别建立相对应的 DISPID。
2. 在代码中使用某种机制将 DISPID 映射到方法和属性上。

3. 使用映射机制实现 Invoke()函数，使得客户能够调用方法和访问属性。

4. 实现 GetIDsOfNames()，使得客户能够根据名字来查找。

选择什么样的 DISPID 是由客户来决定的，并且要遵循一定的限制。DISPID 只不过是客户用来调用函数以及访问属性的一个标记而已。微软把所有的负数的 DISPID 的值都保留了，以备将来使用。它定义了一些标准的 DISPID，如表 14-1 所示。

表 14-1 标准的 DISPID

符号	值	含义
DISPID_VALUE	0	分发接口的默认成员
DISPID_UNKNOWN	-1	由 GetIDsOfNames()返回，表示找不到一个分发 ID
DISPID_PROPERTYPUT	-3	表示一个参数，这个参数接受设置 OLE 自动化属性的赋值语句的值
DISPID_NEWENUM	-4	集合使用
DISPID_EVALUATE	-5	控制器使用，计算括号内表达式的值
DISPID_CONSTRUCTOR	-6	标识一个类似对象的构造函数的方法。保留
DISPID_DESTRUCTOR	-7	标识一个类似对象的析构函数的方法。保留

除了表里面的这 7 个值外，微软还保留了从-500~-999 的 DISPID 值，这些值供 OLE 控件使用。除了这些，客户可以自由地使用从 1 到 20 亿之间的任何一个数，也就是说可供选择的 DISPID 的值范围已经够广的了。

实现 Invoke()函数

尽管实现 Invoke()不是那么困难，但是也相当烦琐。首先，必须为每一个创建的分发方法建立实现功能的 C++代码。然后对于每一个创建的分发属性，必须在 C++代码中增加一个变量来存放这个属性的值。这个也许不是很重要的事情，但又是不得不做的。

接着，必须设计一个系统，将分发 ID 映射到 C++代码中。这个工作也比较简单，相当于就是写一个 switch 语句将每一个分发 ID 转换过来，或者是建立一个优化的散链表来实现。无论使用哪一种方法，目的都是要把传进来的 DISPID 映射到程序中某段实际的代码上去。

Invoke()函数总的来说还是相当复杂的。为了实现 Invoke()，除了 DISPID，它还包括了很多别的信息。比如，如果是使用本地服务器，就得使用 lcid 这个参数。如果客户想要调用一个函数或者访问一个属性，则必须使用 wFlags 这个参数。然而，最复杂的是用来传递参数以及结果的参数。

VARIANT 结构

处理 pDispParams 和 pVarResult 参数是很烦琐的事情。IDispatch 使用 VARIANT 结构来传递参数和返回结果。Invoke()并不使用在编译时就已经知道的一组参数链表。所有的参数都是在运行时才确定下来的。所以，我们就必须遍历 VARIANT 的参数链表，解开每一个参数，把它们转换为原始的表示，然后用 VARIANT 结构返回结果。微软实现一些函数和宏用来处理 VARIANT 结构，这些函数有 VariantInit()、VariantClear()和 VariantChangeType()。

VariantInit()函数用来初始化一个 VARIANT 结构。VariantClear()用来清空一个 VARIANT 结构，并且在 VARIANT 出了作用域的时候必须调用这个函数。VariantChangeType()可以将一个 VARIANT 从一种类型转换成另外一种类型。这个函数在判断传进来的参数是不是正确类型的时候非常有用。

微软还提供了一组如下形式的宏：

```
V_<some type> =  <native data>
```

例如，如果想要在一个 VARIANT 的实例 var 中插入一个长整型的值，叫做“lSomeValue”，则使用下面的宏：

```
VT(&var) = VT_I4;
V_I4(&var) = lSomeValue;
```

如果想从同样的一个 VARIANT 结构中取出一个长整型的值，则可以使用同一个宏：

```
lSomeValue = V_I4(var);
```

OLE 提供了一个函数叫 DispGetParam()，它可以从一个 DISPPARAMS 的数组中取得一个特殊的 VARIANT 参数。它使用 DISPPARAMS 数组和一个索引做参数，然后根据特定的参数填充另外一个 VARIANT 结构。

完成 Invoke()的实现

Invoke()的最后两个参数是用来处理异常和错误代码的。如果客户对异常的信息有兴趣，则会传进来一个 EXCEPINFO 结构，然后 Invoke()在这个结构中填入相应的信息。客户还会传进来一个指向无符号整型的指针，然后 Invoke()返回一个错误代码。

总而言之，微软提供一些很好的工具来帮助实现 IDispatch::Invoke()函数。但是，自己实现这个函数也不是没有危险。为 DispMath 对象实现 Invoke()函数的代码如程序清单 14-2 所示。

程序清单 14-2　IDispatch::Invoke()的实现

```
STDMETHODIMP
DispMath::XDispObj::Invoke(DISPID dispidMember, REFIID riid,
                           LCID lcid, WORD wFlags,
                           DISPPARAMS FAR *pdispparams,
                           VARIANT FAR *pvarResult,
                           EXCEPINFO FAR *pexcepinfo,
                           UINT FAR *puArgErr) {

   unsigned int uArgErr;
   VARIANTARG varg1;
   VARIANTARG varg2;
   VARIANT varResultDummy;


   // Make sure the wFlags are legal
   if(wFlags & ~(DISPATCH_METHOD | DISPATCH_PROPERTYGET |
      DISPATCH_PROPERTYPUT | DISPATCH_PROPERTYPUTREF))
      return ResultFromScode(E_INVALIDARG);

   // This object only exposes a "default" interface.
   if(!IsEqualIID(riid, IID_NULL))
```

```
        return ResultFromScode(DISP_E_UNKNOWNINTERFACE);

      // It simplifies the following code if the caller
      // ignores the return value.
      if(puArgErr == NULL)
         puArgErr = &uArgErr;
      if(pvarResult == NULL)
         pvarResult = &varResultDummy;

      VariantInit(&varg1);
      VariantInit(&varg2);

      // Assume the return type is void, unless we find otherwise.
      VariantInit(pvarResult);


      unsigned long ulFirstNo, ulSecondNo;
      long lResult;
      unsigned int uError;
     // Unpack the parameters
     DispGetParam(pdispparams, 0, VT_I4,&varg1, &uError);

     DispGetParam(pdispparams, 1, VT_I4, &varg2, &uError);

     ulFirstNo = V_I4(&varg1);
     ulSecondNo = V_I4(&varg2);

     switch(dispidMember){

       case DISPID_ADD:
          lResult = ulFirstNo + ulSecondNo;
          V_VT(pvarResult) = VT_I4;
          V_I4(pvarResult) = lResult;
          break;

       case DISPID_SUBTRACT:
          lResult = ulFirstNo - ulSecondNo;
          V_VT(pvarResult) = VT_I4;
          V_I4(pvarResult) = lResult;
          break;

        default:
           return ResultFromScode(DISP_E_MEMBERNOTFOUND);
     }
     return NOERROR;
}
```

注意上面的代码，大部分的工作都是在做错误检查以及将参数解包成为一个有用的格式。保证所有参数都合法，以及返回的结果是正确的类型是服务器的工作。

实现 GetIDsOfNames()函数

GetIDsOfNames()相对来说更容易处理了。我们所要做的全部工作就是检查传进来的分发函数和属性名字的链表，然后填充 DISPID 的数组。程序清单 14-3 就是在 CoMath 中的

GetIDsOfNames()一个简化的实现。

程序清单 14-3 IDispatch::GetIDsOfNames()的实现

```
STDMETHODIMP
DispMath::XDispObj::GetIDsOfNames(REFIID riid, LPOLESTR *rgszNames,
                                  UINT cNames, LCID lcid,
                                  DISPID FAR *rgdispid) {
   SCODE sc = S_OK;
   // check arguments
   if (riid != IID_NULL)
      return DISP_E_UNKNOWNINTERFACE;

   // fill in the member name
   if (lstrcmpi(rgszNames[0], "Add") == 0) {
      rgdispid[0] = DISPID_ADD;
   } else if (lstrcmpi(rgszNames[0], "Subtract") == 0) {
      rgdispid[0] = DISPID_SUBTRACT;
   }  else {
      rgdispid[0] = DISPID_UNKNOWN;
      sc = DISP_E_UNKNOWNNAME;
   }


   // make the argument names DISPID_UNKNOWN for this implementation
   for (UINT nIndex = 1; nIndex < cNames; nIndex++)
      rgdispid[nIndex] = DISPID_UNKNOWN;

   return sc;
}
```

也许你注意到了在前面的例子中有两个 IDispatch 函数都没有完整的实现，这个两个函数是 GetTypeInfo()和 GetTypeInfoCount()。另外，也许你会觉得这个例子太复杂了，它在对参数进行打包和解包以及查找函数并调用函数这些方面花的工作量太大了。这个例子只是展示了一个实现 IDispatch 接口的方法（其实这也是最困难的一种方法）。也许它不是一种实现 IDispatch 的理想方法。实际上，确实有一种更加简单的方法可以实现——通过使用一种叫类型信息的技术。

George 的有关自动化的轶事

我挣钱的一个方式就是四处宣传，讲述如何使用 MFC 来开发 OLE 应用程序。在做了几次之后，我发现一个很有意思的现象。假设我去讲课的公司有 10 个开发者正在做 OLE 的最近版本的开发。但是只有 9 个人会来听课。当我问："为什么 Jane/Joe 没有来上课？"通常回答都是："Oh，他/她几年前就已经实现了我们的自动化接口了，他/她不需要上这个课了，因为 Jane/Joe 已经使用了最困难的方法实现它了。"没来上课的人一般都是这样的人，他们在没有使用 MFC 的情况下实现自动化而赢得了开发小组中成员的尊敬。这种情况出现了很多次！这也表示了自动化在 OLE 体系中的重要性。

很容易忘记有很多种方法可以用来实现自动化，而使用 MFC 的 ClassWizard 只是其中的一种方法而已。但是使用 ClassWizard 还是用来实现自动化接口最快的和最方便方法。

实现自动化的另外一种方法：使用类型信息

类型信息已经成为了自动化中最重要的方面之一。特别是对 OLE 控件来说，它更是重要。下面我们先简单地看一下类型信息，当碰到 OLE 控件的时候将会更详细地讲述。

类型信息

类型信息是一种将对象（包括接口——分发接口或其他接口）的信息通知客户的一种方法。例如，客户可以使用类型信息来知道对象的接口信息而不用调用 QueryInterface()。了解类型信息的最好一种方式是将它看做是一个二进制的头文件。例如，当你在一个模块中写了一个函数或者类，而想在另外一个模块中使用它，那么该怎么做呢？你可以在一个头文件中发布这些信息。预处理器知道这个头文件的语法，然后在背后做一些工作，比如建立一个符号表使得编译器能够使用这些信息创建可执行代码。类型信息也采用同样的思想，只不过是在 COM 接口的环境下而已。

在基于浏览思想的开发工具上下文环境中，类型信息就显得特别重要了。如果有一种方法能够事先得到一个对象的属性、方法、接口等信息，就有可能创建一个开发浏览器，它能够预先查询对象的信息，并将这些对象的信息提供给开发者，使得他们能够使用某种浏览器将一些可编程的对象链接到一起。

但是，从哪里取得类型信息呢？这是由另外一种语言来描述的，这种语言叫对象描述语言（Object Description Language）。

对象描述语言（Object Description Language）

为 COM 对象产生类型信息的最简单的方法是用类型描述语言（ODL）编写脚本，然后使用一个叫 MKTYPLIB.EXE 的工具来编译它。（还有别的方法可以创建类型信息，但是在这里它们都没有必要了解。使用 MKTYPLIB 让微软来帮我们编译是最好的方法了。）对象描述语言首先声明它的属性（比如 GUID、版本和位置信息等），接着描述真正的类型信息。

例如，Math 对象的 ODL 脚本如程序清单 14-4 所示。

程序清单 14-4 用 ODL 脚本描述 CoMath

```
01 [
02   uuid(06812841-46e9-11ce-aeff-e798a8721416)
03   helpstring("MFC Internals CoMath 1.0 Type Library"),
04   lcid(0x9),
05   version(1.0)
06 ]
07 library Math08
08 {
09
10    importlib("stdole.tlb");
11
12     [
13       uuid(06812842-46e9-11ce-aeff-e798a8721416)
```

```
14        helpstring("IMath Dispatch Interface"),
15        odl
16      ]
17      interface IMathDisp : IUnknown
18      {
19        [propput,  helpstring("Set the current value"), id(0)]
20        void CurrentValue([in] long val);
21
22        [propget, helpstring("Get the current value"), id(0)]
23        long CurrentValue(void);
24
25        long Add([in] long val1, [in] long val2);
26        long Subtract([in] long val1, [in] long val2);
27      }
28
29
30      [
31        uuid(06812843-46e9-11ce-aeff-e798a8721416)
32        helpstring("DMathDisp Dispatch Interface")
33      ]
34      dispinterface DMathDisp
35      {
36        interface IMathDisp;
37      }
38
39      [
40        uuid(06812840-46e9-11ce-aeff-e798a8721416)
41        helpstring("CoMath: A COM Math Class")
42      ]
43      coclass CoMath
44      {
45        dispinterface DMathDisp;
46        interface IMathDisp;
47      }
48  }
```

脚本的最前面 6 行描述了整个库的属性。类型库有一个 GUID 与之相对应，类型库还包括一个版本号和一些帮助信息。这些信息被一个中括号给括起来。

接下去的脚本（7~48 行）描述了类型库的数据。类型库有几个类型信息块组成，它包括一组特殊接口（IMathDisp 和 DMathDisp）的类型信息和实现这些接口的 COM 类的类型信息。

当COM对象的信息被放入到类型库中后，就可以使用这个类型库来实现IDispatch接口了。

使用类型信息实现 IDispatch 接口

如果是从类型信息中得到 COM 对象和分发接口的定义，则可以有几种方法使用 MFC 来实现 IDispatch（而不是什么都自己手工完成）。COM 使用类型信息来实现 IDispatch 有两种标准的方法。但是，要实现这个功能，首先要装载类型库，然后取得必要的接口指针：ITypeLib 和 ITypeInfo。（注意我们现在是在 COM 中，所以所有东西都是通过接口来完成的。）

装载一个类型库

OLE 中有好几个 API 函数可以用来装载类型库。从自动化对象的观点来看，最常用的两

个函数是 LoadRegTypeLibrary()和 LoadTypeLibrary()。LoadRegTypeLibrary()从注册表中导入类型库，然后返回一个 ITypeLib 接口指针。LoadTypeLibrary()将类型库所在的路径作为第一个参数，然后返回一个 ITypeLib 指针给调用者。LoadTypeLibrary()可以在装载类型库的同时，在注册表内建立一个入口（如果这个类型库原来没有在注册表中的话）。

也许在 ODL 文件中你注意到了整个类型库是由一个 GUID 来标识的。另外，在类型库中每一段类型信息也是使用 GUID 来标识的。ITypeLib 中有一个成员函数叫 GetTypeInfoOfGuid()，只要你知道某一个特定类型信息块的 GUID，就可以调用 GetTypeInfoOfGuid()得到一个表示那个块的 ITypeInfo 指针。

现在你已经有了一个 ITypeInfo 的指针，接下来该做些什么呢？

利用类型信息

ITypeInfo 是最复杂的 COM 接口之一，它有 19 个函数（不包括 IUnknown 的 3 个函数）。很多的函数是用来获取类型信息的各种属性，比如函数和参数的信息。你没有必要把这些函数都记住（当然也没有反对你全记住）。ITypeInfo 有两个很有意思的成员函数，它们是 Invoke()与 GetIDsOfNames()。仔细检查你会发现，这两个函数与 IDispatch::Invoke()和 IDispatch::GetIDsOfNames()很相似：

```
HRESULT ITypeInfo::Invoke(void *lpvInstance
                DISPID dispidMember,
                WORD wFlags,
                DISPPARAMS FAR* pdispparams,
                VARIANT FAR* pvarResult,
                EXCEPINFO FAR* pexcepinfo,
                UINT FAR* puArgErr);

HRESULT ITypeInfo::GetIDsOfNames(char FAR* FAR* rgszNames,
                      UINT cNames,
                      DISPID FAR* rgdispid);
```

最主要的区别就是 Invoke()的第一个参数。这个参数表示了实现了 IDispatch 接口的 COM 类。但是这个类是从何而来的呢？

开发者必须自己写这个类，但是 MKTYPLIB 会给开发者一个很好的基础。使用 MKTYPLIB 编译类型信息后，会产生一个标准 C 的头文件。这个头文件定义了一个接口，该接口与 ODL 文件中描述的类型信息相对应。要使用类型信息，只要写一个 COM 类，让这个类从 IDispatch 和这个头文件中定义的接口中派生。在你的 COM 类的构造函数中别忘了装载类型库（使用 LoadTypeLibrary()）和取得类型信息（使用 GetTypeInfoOfGuid()）。然后当一个客户调用你的 COM 对象的 Invoke()或 GetIDsOfNames()函数时，只要分别将它们委托给 ITypeInfo 的函数即可。然后，OLE 会做其余的工作，比如将函数打包和将 DISPID 解释为真正的函数调用。

IDisPatch::GetTypeInfo()和 IDispatch::GetTypeInfoCount()函数

现在你已经了解了类型信息的概念。IDispatch::GetTypeInfo()和 IDispatch::GetTypeInfoCount()这两个函数也许能进一步加深你的理解。

在得到了描述自动化接口的类型库后，使自动化客户能够使用这些信息就不是很难的事

情了。只要导入类型库（使用 LoadTypeLibrary()），取得类型信息（使用 GetTypeInfoOfGuid()）然后返回 ITypeInfo 接口指针给客户即可。

实现 GetTypeInfoCount()函数也很简单。只要在类型信息可用时返回 1，在类型信息不可用时返回 0。

另外一个实现方法：CreateStdDispatch()

还有另外的一种方法可以实现 IDispatch 接口，就是使用另外一个 COM API 函数，叫做 CreateStdDispatch()。下面是这个函数的原型：

```
STDAPI CreateStdDispatch(
    IUnknown FAR* punkOuter,
    void FAR* pvThis,
    ITypeInfo FAR* ptinfo,
    IUnknown FAR* FAR* ppunkStdDisp);
```

当你将类型信息传递给 CreateStdDispatch()时，这个函数会根据 ITypeInfo 指针中的类型信息来创建一个实现了 IDispatch 接口的 COM 对象。它的第一个参数是一个指向控制 IUnknown 的指针。这个就是我们在第 11 章所学的聚合模型的用法。CreateStdDispatch()实际上会在你的对象里面创建一个聚合对象。CreateStdDispatch()创建一个 COM 对象，然后别的成员都隐藏在控制 IUnknown 后面。CreateStdDispatch()会返回一个 IDispatch 接口，在客户用 QueryInterface()请求的时候就返回这个接口。

了解自动化的本来面目

DispMath 对 IDispatch 的第一个实现版本是不完全的，因为它没有类型信息。但是，这个版本作为一个简单的自动化对象也能运行得很好，而且 Visual Basic（或是另外一个自动化客户，他知道如何使用 GetIDsOfNames()）能够使用它 www.infopower.com.cn 上有一个程序叫 MATHSRVR。这是一个标准的 MFC 应用程序，它包含了 DispMath 对象。使用 TESTMSTH.MAK 这个 Visual Basic 程序可以测试它。你可以使用 debugger 来运行 MATHSRVR，从而查看执行的具体细节。

实现 IDispatch 的第二种和第三种方法都需要产生一个类型库，然后使用类型信息。

为什么绕了一大圈来介绍自动化呢？也许一开始你就会说："我需要使用的是 MFC！"。完全理解 COM（和 OLE）的其中一个障碍是它们只是一个标准。也许会有很多的方法可以用来实现同一个接口。IDispatch 是其中的一个典型代表。如果你了解了如何完全自己实现自动化，MFC 的实现就更显得有意义了。下面一个部分是讲述 MFC 如何实现 IDispatch 接口。和大多数通过 MFC 实现的 OLE 特性一样，自动化也是从 CCmdTarget 这个类开始的。我们将会讲述 CCmdTarget 的扩展以及分发映射宏，MFC 使用这些宏来实现自动化。

MFC 与自动化

现在你已经知道了在 COM 级别上自动化的运行方式，下面让我们来看看 MFC 是如何实现自动化的。我们已经知道，自动化是通过一个叫 IDispatch 的接口实现的，所以要实现自动

化的 COM 类就必须实现这个接口。IDispatch 接口包括 4 个函数，分别是 Invoke()、GetIDsOfNames()、GetTypeInfo()和 GetTypeInfoCount()。Invoke()使用一个叫 DISPID 的 32 位标识符来调用自动化的方法和访问自动化的属性。自动化的客户可以使用 GetIDsOfNames()方法将一个可读性很好的名字转换为一个 DISPID。另外两个函数，GetTypeInfo()和 GetTypeInfoCount()，是用来访问与自动化对象相关的类型信息的。

程序 MATHSRVR 中实现了名为 DISPMATH 的对象，这个对象支持自动化。DISPMATH 提供了对 IDispatch 接口的最小实现，它只实现了 Invoke()和 GetIDsOfNames()函数而没有实现 GetTypeInfo()和 GetTypeInfoCount()。DISPMATH 的 Invoke()函数实现了在自动化中使用 VARIANT 来利用传递参数。

另外还可以使用类型信息来实现 IDispatch 接口。

但是这是一种比较困难的方法。Visual C++的其中很重要的一个特性就是它对自动化提供了支持。AppWizard 使得产生支持自动化的应用程序变得非常容易，而 ClassWizard 可以很容易地给自动化对象添加方法和属性。那么 MFC 是怎么做的呢？当你按下这些按钮时，MFC 内部都做了些什么呢？这一部分内容就是讲述 MFC 如何实现自动化。我们将会去了解 CCmdTarget 扩展——一个叫做分发映射表的机制，MFC 使用它来进行动态调用（比如自动化）。

CCmdTarget 的扩展

跟 MFC 实现的其他 GLE 特性一样，自动化也是通过 CCmdTarget 来实现的。这是因为 CCmdTarget 实现了 IUnknown 和 IDispatch。这带来的一个很大的好处就是可以在任何从 CCmdTarget 派生的类中添加自动化特性。CCmdTarget 是实现自动化的关键。

IDispatch 没有接口映射表入口

MFC 使用嵌套类/合成方法来实现 COM。它使用一组宏来给每个必要的接口生成相对应的嵌套类。例如，MFC 中有一个类叫 COleDataSource，它实现了 IDataObject 接口。在 AFXOLE.H 中可以看见 COleDataSource 的定义，它使用了宏来定义一个嵌套类，用它来实现 IDataObject 接口，还使用了接口映射表（CCmdTarget 使用它来实现 QueryInterface()）。由于 CCmdTarget 实现了 IDispatch 接口，所以可以预见必然有 COM 宏来实现 IDispatch 接口。

但是，在 CCmdTarget 的源代码中，我们却找不到任何实现 IDispatch 接口的 MFC 宏。那么 CCmdTarget 是如何实现 IDispatch 接口的呢？

在 MFC 的源代码目录下有一个文件叫 OLEIMPL2.H，它定义了一个叫 COleDispatchImpl 的类。这个类中包含了 IDispatch 接口的 7 个函数。其中有 3 个 IUnknown 接口的函数：AddRef()、Release()、QueryInterface()以及 4 个 IDispatch 定义的函数：GetTypeInfoCount()、GetTypeInfo()、GetIDsOfName()和 Invoke()。所以 COleDispatchImpl 定义了一个 C++类，它实现了 IDispatch 接口。看起来 MFC 是利用这个类实现了 IDispatch，但是，如果 CCmdTarget 实现了 IDispatch，那么 COleDispatchImpl 是如何与 CCmdTarget 挂钩的呢？答案是使用了 CCmdTarget 中的 EnableAutomation()函数。

CCmdTarget::EnableAutomation()函数

MFC 文档上说，必须要调用 CCmdTarget 的 EnableAutomation()函数才能激活自动化。

EnableAutomation()是内部函数之一，在 MFC 领域内的文档不会解释这个函数（它会说："别管那么多，只管用就行了！"）。EnableAutomation()函数将 CCmdTarget 连接到 COleDispatchImpl 类上。

让我们回去再看看 CCmdTarget 的定义，CCmdTarget 将自己声明为 COleDispatchImpl 的友元。（COleDispatchImpl 提供了 IDispatch 接口的标准实现）CCmdTarget 还定义了一个叫 XDispatch 的嵌套类，这个类中只有一个 DWORD 类型的变量，它表示 IDipatch 的 vtable：

```
struct XDispatch
{
   DWORD m_vtbl;    // place-holder for IDispatch vtable
} m_xDispatch;
```

这个类的目的就是得到一个实现 IDispatch 接口的标准的 vtable，并将它加到 CCmdTarget 中，下面的就是 EnableAutomation()函数所做的工作：它将 COleDispatchImpl 的 vtable 的地址拷贝到一个 DWORD 中，这个 DWORD 就是 CCmdTarget 需要的 IDispatch 接口的 vtable。在普通的模式下（即没有调用 EnableAutomation()的情况下使用 CCmdTarget），CCmdTarget 不支持 IDispatch，XDispatch::m_vtbl 被初始化为 0。调用 EnableAutomation()后会将 XDispatch::m_vtbl 的值设为 COleDispatchImpl 的 vtable。CCmdTarget 总是会拥有一个 IDispatch 接口的 vtable，这个值总是为 NULL，只有在调用了 EnableAutomation()的时候才改变。CCmdTarget::EnableAutomation()就是将一个 IDispatch 的 vtable 插入到 CCmdTarget 中。

现在，CCmdTarget 有了一个 IDispatch 的 vtable。那么接口映射表在哪里呢？在调用 QueryInterface()的时候，取得 IDispatch 接口的过程并不是那么显而易见。根据 MFC 对自动化的实现机制，在 CCmdTarget 的接口映射表中应该有 IID_IDispatch 的入口。在调试状态下单步跟踪 MFC 如何使用 InternalQueryInterface()进行接口映射表查询的时候，我们就可以看到 IID_IDispatch 的 GUID 在 CCmdTarget 的接口映射表中总是第一个入口。这是为什么呢？让我们来看看 CMDTARG.CPP 文件就知道了。MFC 在这个文件中为 CCmdTarget 的接口映射表声明的第一个入口的 IID 就叫 IID_IDispatch，它对应的 vtable 偏移是 CCmdTarget 的 m_xDispatch 成员。

```
const AFX_INTERFACEMAP_ENTRY CCmdTarget::_interfaceEntries[] =
{
   INTERFACE_PART(CCmdTarget, _afx_IID_IDispatch, Dispatch)
   { NULL, (size_t)-1 }    // end of entries
};
```

通过这种方法，当客户对 CCmdTarget 的派生类调用 QueryInterface()函数时，这个类总会有一个 IID_Dispatch 的入口。vtable 指针的值可能是 NULL 或者是 COleDispatchImpl 的 vtable 的值，这取决于是否调用了 EnableAutomation()函数。

自动化需要使用的 CCmdTarget 的其他函数

为了实现 IDispatch 接口，CCmdTarget 还有一对很有用的成员函数：GetIDispatch()和 FromIDispatch()。GetIDispatch()从 CCmdTarget 中取得一个 IDispatch 接口指针。如果你总是使用有效的 IDispatch 指针，这是很容易得到的。FromIDispatch()返回一个 CCmdTarget 类，它关联着一个特定的 IDispatch 接口指针。

在 CCmdTarget 的扩展中，MFC 使用了分发映射表通过 DISPID 来遍历和访问分发的函

数。下面是它们的工作方式。

分发映射表（dispatch map）

MFC 实现 IDispatch 的第二个结构是一种叫分发映射表（dispatch map）的技术。MFC 在很多地方都使用了映射表，为什么自动化必须跟另外的映射表有所区别呢？MFC 使用分发映射表来实现 IDispatch::GetIDsOfNames()和 IDispatch::Invoke()。在 DISPMATH 对象中，对 DISPID 使用 switch 语句来实现 Invoke()。相对应地，分发映射表提供了表驱动（table-driven）来代替 switch 语句。

MFC 使用宏来实现分发映射表。MFC 的分发映射表宏所遵循的基本规则与所有其它的映射表完全相同。使用一个宏来声明分发映射表所需要的成员变量和成员函数。一般地，这个宏叫 DECLARE_DISPATCH_MAP。另外有两个宏用来实现真正的成员变量和成员函数，这两个宏是 BEGIN_DISPATCH_MAP 与 END_DISPATCH_MAP。最后，还有一组宏用来往分发映射表里面填充数据。MFC 提供了 4 个不同的宏来产生分发映射表。其中一个宏用来定义自动化的函数，另外 3 个用来定义自动化的属性，这 4 个宏是 DISP_FUNCTION、DISP_PROPERTY、DISP_PROPERTY_NOTIFY、DISP_PROPERTY_EX。

如果使用 AppWizard 来生成应用程序，并且打开“Automation—yes，please”按钮来生成一个支持自动化的文档。也就是说，这个文档的构造函数会去调用 EnableAutomation()，并且这个文档会得到一个空的分发映射表，可以往里面添加表项。当一个 CCmdTarget 的派生类拥有了一个分发映射表后，就可以使用 ClassWizard 来添加和删除属性和方法了。下面我们来看看 MFC 的分发映射表的内部构造。

声明分发映射表

通过在头文件里使用 DECLARE_DISPATCH_MAP 宏，可以建立实现 IDispatch 查询所必需的结构。下面是 DECLARE_DISPATCH_MAP 的定义：

```
#define DECLARE_DISPATCH_MAP() \
private: \
   static const AFX_DISPMAP_ENTRY _dispatchEntries[]; \
   static UINT _dispatchEntryCount; \
protected: \
   static AFX_DATA const AFX_DISPMAP dispatchMap; \
   virtual const AFX_DISPMAP* GetDispatchMap() const; \
```

DECLARE_DISPATCH_MAP 定义了一个 AFX_DISPMAP_ENTRY 结构的静态数组，名字叫_dispatchEntries。它还声明了一个变量，用来保存 IDispatch 查询表的大小。然后 DECLARE_DISPATCH_MAP 加入一个 AFX_DISPMAP 结构和一个虚函数，用来获取分发映射表，这个虚函数叫 GetDispatchMap()。

下面是 AFX_DISPATCH 结构的定义：

```
struct AFX_DISPMAP
   const AFX_DISPMAP* pBaseMap;

   const AFX_DISPMAP_ENTRY* lpEntries;
   UINT* lpEntryCount;
};
```

AFX_DISPMAP 保存着一个指向基（base）分发映射表的指针、一个指向当前类的分发映射表第一个元素的指针以及分发映射表中条目的个数。

在分发映射表中存放的数据就是各种分发方法以及属性。它们都由 AFX_DISPMAP_ENTRY 结构来表示。这个结构的定义如下：

```
struct AFX_DISPMAP_ENTRY
{
   LPCTSTR lpszName;         // member/property name
   long lDispID;             // DISPID (may be DISPID_UNKNOWN)
   LPCSTR lpszParams;        // member parameter description
   WORD vt;                  // return value type / or type of property
   AFX_PMSG pfn;             // normal member On<membercall> or, OnGet<proper
   AFX_PMSG pfnSet;          // special member for OnSet<property>
   size_t nPropOffset;       // property offset
   AFX_DISPMAP_FLAGS flags; // flags (e.g. stock/custom)
};
```

下面我们将会看到如何利用 MFC 提供的 5 个宏来填充分发映射表条目。但是，首先让我们来看看 BEGIN_DISPATCH_MAP 和 END_DISPATCH_MAP 这两个宏。

实现分发映射表

在声明了分发映射表之后，CCmdTarget 派生类还必须使用 BEGIN_DISPATCH_MAP/END_DISPATCH_MAP 来做另外的一些工作。让我们先来看看 BEGIN_DISPATCH_MAP 宏。

与 MFC 另外的查询机制（比如消息映射表和接口映射表）中的 BEGIN...宏一样，BEGIN_DISPATCH_MAP 通过实现 GetDispatchMap()，填充 dispatchMap 结构以及往分发映射表中添加条目，从而开始运转。下面是 BEGIN_DISPATCH_MAP 的定义：

```
#define BEGIN_DISPATCH_MAP(theClass, baseClass) \
        const AFX_DISPMAP* theClass::GetDispatchMap() const \
              { return &theClass::dispatchMap; } \
        const AFX_DISPMAP theClass::dispatchMap = \
              { &baseClass::dispatchMap, &theClass::_dispatchEntries[0], \
                        &theClass::_dispatchEntryCount }; \
        UINT theClass::_dispatchEntryCount = (UINT)-1; \
        const AFX_DISPMAP_ENTRY theClass::_dispatchEntries[] = \
        { \
```

分发映射表使用 END_DISPATCH_MAP 宏来结束。END_DISPATCH_MAP 在分发映射表的查询表的最后添加一个 NULL 条目并且插入封闭的大括号和分号。下面是 END_DISPATCH_MAP 的定义：

```
#define END_DISPATCH_MAP() \
        { VTS_NONE, DISPID_UNKNOWN, VTS_NONE, VT_VOID, \
          (AFX_PMSG)NULL, (AFX_PMSG)NULL, (size_t)-1, afxDispCustom } }; \
```

当然，如果分发映射表中没有数据，那么它就是没用的。MFC 提供了 5 个宏，用来在分发映射表中插入自动化方法和属性的查询信息。这 5 个宏分别是（1）DISP_FUNCTION；（2）DISP_PROPERTY；（3）DISP_PROPERTY_NOTIFY；（4）DISP_PROPERTY_EX；（5）DISP_PROPERTY_PARM。

DISP_FUNCTION 宏

当使用 ClassWizard 插入一个自动化方法时，ClassWizard 会自动地将这个宏放入到分发映射表中。DISP_FUNCTION 通过在 AFX_DISPMAP_ENTRY 中填充必要的信息来将一个自动化方法添加到分发映射表中。下面是这个宏的定义：

```
#define DISP_FUNCTION(theClass, szExternalName, pfnMember, vtRetVal,
vtsParams) \
   { _T(szExternalName), DISPID_UNKNOWN, vtsParams, vtRetVal, \
      (AFX_PMSG)(void (theClass::*)(void))pfnMember, (AFX_PMSG)0, 0, \
      afxDispCustom }, \
```

DISP_FUNCTION 宏有 5 个参数，它们分别是创建这个分发映射表的类的名字、外面（也就是自动化客户）所能看见的这个方法的名字、C++成员函数、一个 VARIANT 标志（它表示这个函数的返回值）以及一个 VARIANT 标志链表，它表示这个方法的参数。

这个宏会往 AFX_DISPMAP_ENTRY 结构中添加分发方法的名字、DISPID_UNKNOWN 的值、参数描述符、一个描述返回值类型的 VARIANT 标志以及一个指向 C++类的成员函数的指针。使用 ClassWizard 来添加自动化方法，它会在 C++类中加入一个成员函数，并且使用 DISP_FUNCTION 宏创建相对应的分发映射表的条目。

DISP_PROPERTY 宏

如果使用 ClassWizard 来添加一个无格式的属性，即这个属性没有 get/set 方法，也没有通知方法。ClassWizard 会使用 DISP_PROPERTY 在分发映射表中插入一个自动化属性。下面是这个宏的定义：

```
#define DISP_PROPERTY(theClass, szExternalName, memberName, vtPropType) \
   { _T(szExternalName), DISPID_UNKNOWN, NULL, vtPropType, (AFX_PMSG)0, \
    (AFX_PMSG)0, offsetof(theClass, memberName), afxDispCustom }, \
```

DISP_PROPERTY 有 4 个参数：（1）这个分发映射表所关联的类的名字；（2）属性的外部名字；（3）CCmdTarget 派生类的成员变量的名字；（4）一个表示属性类型的 VARIANT 标志。

这个宏会在 AFX_DISPMAP_ENTRY 中添加分发属性的名字、DISPID_UNKNOWN 的值，一个 NULL 字符串（表示没有参数）、一个表示属性类型的 VARIANT 变量以及 C++类的成员变量的地址。ClassWizard 会在 C++类中加入这个成员变量并且将这个宏与分发映射表关联起来。

DISP_PROPERTY_NOTIFY 宏

当自动化客户改变了属性的时候，分发映射表必须能够及时地知道。通过使用 ClassWizard，并提供属性的通知方法的名字，它就可以使用 DISP_PROPERTY_NOTIFY 将一个自动化属性插入到分发映射表，并且同时提供一个通知函数。这个函数在属性被自动化客户改变的时候就会被调用。DISP_PROPERTY_NOTIFY 宏的定义如下：

```
#define DISP_PROPERTY_NOTIFY(theClass, szExternalName, memberName,
pfnAfterSet, vtPropType) \
   { _T(szExternalName), DISPID_UNKNOWN, NULL, vtPropType, (AFX_PMSG)0, \
      (AFX_PMSG)(void (theClass::*)(void))pfnAfterSet, \
      offsetof(theClass, memberName), afxDispCustom }, \
```

这个宏与 DISP_PROPERTY 很类似，除了它多加了一个通知函数的指针。它将这个函数指针赋值给 AFX_DISPMAP_ENTRY 的 pfnSet 域。ClassWizard 会将成员变量和通知函数都加入到 C++类中。

DISP_PROPERTY_EX 宏

除了给属性添加一个通知函数外，也可以通过一对 Get 和 Set 方法将属性提供出去。DISP_PROPERTY_EX 在分发映射表中插入一个自动化属性并且提供一对访问方法。下面是 DISP_PROPERTY_EX 的定义：

```
#define DISP_PROPERTY_EX(theClass, szExternalName, pfnGet, pfnSet,
vtPropType) \
   { _T(szExternalName), DISPID_UNKNOWN, NULL, vtPropType, \
      (AFX_PMSG)(void (theClass::*)(void))pfnGet, \
      (AFX_PMSG)(void (theClass::*)(void))pfnSet, 0, afxDispCustom }, \
```

DISP_PROPERTY_EX 和 DISP_PROPERTY_NOTIFY 很类似，除了它不是提供一个通知函数，而是使用一个 Get 和一个 Set 方法来访问属性。使用 ClassWizard 加入 Get/Set 这两个自动化方法会在 C++类中添加 Get...()和 Set...()两个方法。这个宏会将 Get...()和 Set...()方法赋值给 AFX_DISPMAP_ENTRY 的 pfnGet 和 pfnSet 域。

DISP_PROPERTY_PARAM 宏

有时也许 Get...()和 Set...()函数为了实现特殊的功能需要额外的参数。为了支持这个，MFC 提供了 DISP_PROPERTY_PARAM 宏。这个宏和 DISP_PROPERTY_EX 很类似——它定义了一个自动化属性，包括 Get/Set 访问方法。只不过这里的 Get/Set 的参数可以是另外的参数。下面就是 DISP_PROPERTY_PARAM 的定义：

```
#define DISP_PROPERTY_PARAM(theClass, szExternalName, pfnGet, pfnSet,
vtPropType, vtsParams) \
   { _T(szExternalName), DISPID_UNKNOWN, vtsParams, vtPropType, \
      (AFX_PMSG)(void (theClass::*)(void))pfnGet, \
      (AFX_PMSG)(void (theClass::*)(void))pfnSet, 0 }, \
```

使用 ClassWizard 添加这种类型的属性，会在 C++类中添加 Get/Set 函数。这两个函数的特征反映了有额外的参数。

宏的展开

下面是一个例子，说明了这些分发映射宏是如何工作的。分发映射表是从一个叫 AUTOTEST 的简单程序中取得的。这是一个标准的 MFC 程序，它的文档（CAutotestDoc）支持自动化，分发映射表中有每一种类型的条目：一个分发函数、一个没有通知的分发属性、一个有通知的分发属性、一个有 Get/Set 方法的分发属性以及一个分发参数（它的 Get/Set 访问方法有额外的参数）。

程序清单 14-5 展示了 ClassWizard 产生的代码。

程序清单 14-5 ClassWizard 产生的分发映射表

```
BEGIN_DISPATCH_MAP(CAutotestDoc, CDocument)
   //{{AFX_DISPATCH_MAP(CAutotestDoc)
   DISP_FUNCTION(CAutotestDoc, "SomeMethod", SomeMethod, VT_I2,
      VTS_I4 VTS_I2)
```

```
    DISP_PROPERTY(CAutotestDoc, "PlainProperty", m_plainProperty, VT_I4)
    DISP_PROPERTY_NOTIFY(CAutotestDoc, "NotifyProperty",
       m_notifyProperty, OnNotifyPropertyChanged, VT_R8)
    DISP_PROPERTY_EX(CAutotestDoc, "GetSetProperty",
       GetGetSetProperty, SetGetSetProperty, VT_R4)
    DISP_PROPERTY_PARAM(CAutotestDoc, "GetSetProperty2",
       GetGetSetProperty2, SetGetSetProperty2, VT_I4, VTS_I2 VTS_I4)
    //}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()
```

当编译完上面这段代码后，就可以得到如程序清单 14-6 所示的完整的分发映射表。两个条目之间的空行是为了看起来清晰而加上去的。

程序清单 14-6 预编译器生成的分发映射表

```
const AFX_DISPMAP* CAutotestDoc::GetDispatchMap() const {
return &CAutotestDoc::dispatchMap; }

const AFX_DISPMAP CAutotestDoc::dispatchMap = {
   &CDocument::dispatchMap,
   &CAutotestDoc::_dispatchEntries[0],
   &CAutotestDoc::_dispatchEntryCount

};

UINT CAutotestDoc::_dispatchEntryCount = (UINT)-1;
const AFX_DISPMAP_ENTRY CAutotestDoc::_dispatchEntries[] = {
   { "SomeMethod", ( -1 ), "\x03" "\x02", VT_I2,
     (AFX_PMSG)(void (CAutotestDoc::*)(void))SomeMethod, (AFX_PMSG)0, 0 },

   { "PlainProperty", ( -1 ), 0, VT_I4,
     (AFX_PMSG)0, (AFX_PMSG)0,
     (size_t)&(((CAutotestDoc *)0)->m_plainProperty) },

   { "NotifyProperty", ( -1 ), 0, VT_R8,
     (AFX_PMSG)0, (AFX_PMSG)
     (void (CAutotestDoc::*)(void))OnNotifyPropertyChanged,
     (size_t)&(((CAutotestDoc *)0)->m_notifyProperty) },

   { "GetSetProperty", ( -1 ), 0, VT_R4,
     (AFX_PMSG)(void (CAutotestDoc::*)(void))GetGetSetProperty,
     (AFX_PMSG)(void (CAutotestDoc::*)(void))SetGetSetProperty, 0 },

   { "GetSetProperty2", ( -1 ), "\x02" "\x03", VT_I4,
     (AFX_PMSG)(void (CAutotestDoc::*)(void))GetGetSetProperty2,
     (AFX_PMSG)(void (CAutotestDoc::*)(void))SetGetSetProperty2, 0 },

   { 0, ( -1 ), 0, VT_VOID, (AFX_PMSG)0, (AFX_PMSG)0, (size_t)-1 }
};
```

MFC 和 DISPID

我们仔细看一下分发映射表就可以发现所有条目的分发 ID 都是-1（它对应的是 DISPID_UNKNOWN）。这是怎么一回事呢？DISPID 不是固定的，MFC 在运行中再得到具体的分发 ID。下面让我们来看看当自动化客户需要得到成员函数或属性的分发 ID 时 MFC 会做

些什么。

CCmdTarget 和 GetIDsOfNames()函数

GetIDsOfNames()通过人可阅读的名字得到一个分发属性的 DISPID。在 MFC 中，GetIDsOfNames()在 COleDispatchImpl 类中实现。GetIDsOfNames()有 5 个参数，最重要的参数是一个字符串数组，它表示分发成员的名字，还有一个重要的参数就是分发 ID 的空的数组。下面是 GetIDsOfNames()的函数原型：

```
HRESULT GetIDsOfNames(REFIID riid,
                      LPTSTR* rgszNames,
                      UINT cNames,
                      LCID,
                      DISPID* rgdispid)
```

没有类型信息的 GetIDsOfNames()函数

与别的映射表类似，MFC 也在一个链表中保存类层次结构的分发映射表信息。在这个链表中每一个支持自动化的类都有自己的分发映射表。每一个分发映射表都是一个 AFX_DISPMAP_ENTRY 结构的数组。为了说明这个，假设有两个支持自动化的类（当然都是从 CCmdTarget 类中派生），分别叫做 CMath 和 CMathEx。CMath 实现了两个虚函数：Add()和 Subtract()，而 CMathEx 除了继承 Add()和 Subtract()外，还实现了 Multiply()和 Divide()两个函数。CMath 和 CMathEx 都有自己的分发映射表。每个分发映射表都是向后链接在一起的，并从最后派生的类开始，最后链接的是基类（CCmdTarget）。图 14-3 表示了这个关系。

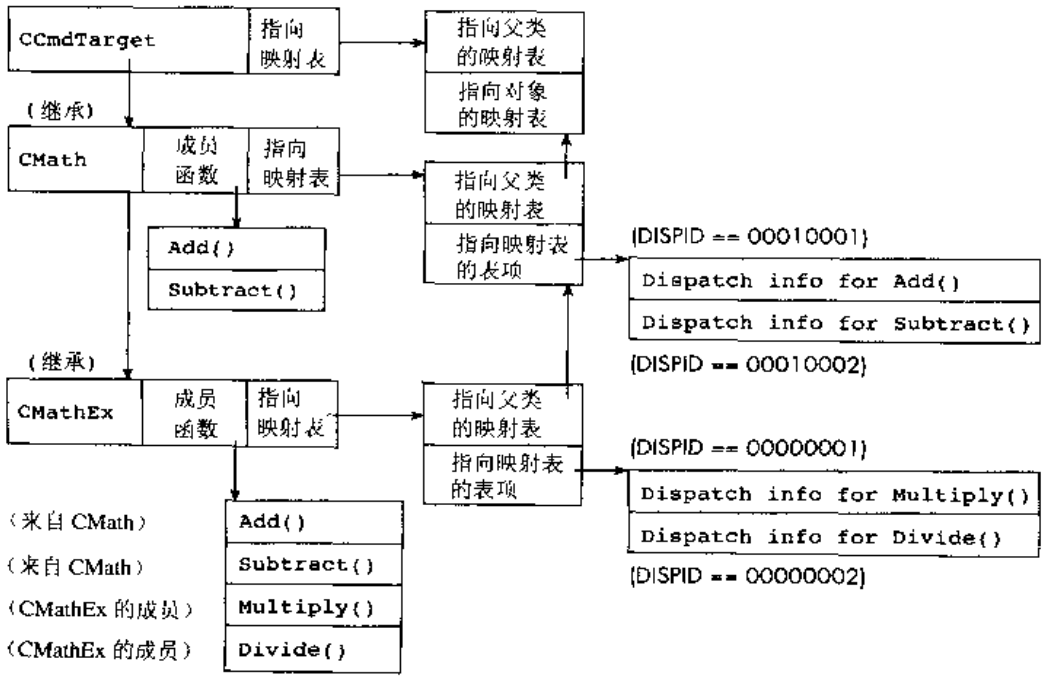


图 14-3 一个分发映射表的图形表示

如果你只有一个 CMath 对象，则 Add()和 Subtract()的 DISPID 就必须改变了，因为它没有基类。尽管 CMath 和 CMathEx 的 DISPID 是相关的，但是 MFC 并不允许以多态方式访问

CMath 和 CMathEx。但是，这种“面向对象”的特性并不是自动化的特性。你可以使用继承来实现对象，但就客户端而言，这两种对象是完全不相关的两种实现。

MFC 使用这种机制动态地创建 DISPID。MFC 将分发 ID（类型是 DWORD）分为两部分。高位的 WORD 是分发方法到当前类（也就是最后一个派生类）的距离。低位的 WORD 表示在分发映射表中的索引。所以在前面的例子中，CMath 的 Add()方法的 DISPID 是 00010001，而 CMath 的 Subtract()方法的 DISPID 是 00010002。高位 WORD 为 1 是因为 Add()和 Subtract()这两个方法属于 CMath，而 CMath 到最后一个派生类（CMathEx）的距离为 1。CMathEx 的 Multiply()函数的 DISPID 是 00000001，CMathEx 的 Divide()函数的 DISPID 是 00000002。高位的 WORD 为 0 是因为 Multiply()和 Divide()这两个方法属于 CMathEx，它是最后一个派生类（也就是说到最后一个派生类的距离为 0）。低位的 WORD 表示在 CMath 分发映射表中的位置。

MFC 通过遍历类的分发映射表动态地产生 DISPID。为了产生 DISPID，CCmdTarget 派生类首先使用 GetDispatchMap()函数（这个函数由宏提供）得到指向分发映射表的指针。然后 GetIDsOfNames()调用 CCmdTarget::MemberIDFromName()，传进去对象的分发映射表以及调用者提供的分发 ID 数组的第一个元素的指针，希望从人可阅读的名字中得到一个 DISPID。

分发映射表中的每个条目都包含一个字符串，用来标识分发成员。CCmdTarget::MemberIDFromName()从最后一个派生类的分发映射表开始搜索，寻找和分发成员匹配的名字。如果不能从最后一个派生类的分发映射表中找到匹配，则转到下一个分发映射表中，它保留了在整个层次中的位置。当这个函数找到一个匹配，它将分发映射表的索引和分发成员的索引融合得到一个分发 ID。GetIDsOfNames()将这个值返回给自动化客户，自动化客户就可以使用这个分发 ID 来调用方法或访问属性。

也许产生 DISPID 的这种方法看起来有点古怪，但是，你很快就会看到 MFC 正是通过这种方法优化了 Invoke()。

CCmdTarget 类与 Invoke()函数

前面我们提到过，CCmdTarget 并没有真正地实现 IDispatch::Invoke()函数，它使用的是 COleDispatchImpl 这个辅助类的实现。COleDispatchImpl::Invoke()拥有标准的 Invoke()的全部特性，它的原型如下：

```
STDMETHODIMP COleDispatchImpl::Invoke(
        DISPID dispid, REFIID riid, LCID lcid,
        WORD wFlags, DISPPARAMS* pDispParams,
        LPVARIANT pvarResult,
        LPEXCEPINFO pexcepinfo, UINT* puArgErr);
```

COleDispatchImpl::Invoke()调用 CCmdTarget::GetDispEntry()取得分发映射表条目。GetDispEntry()使用 DISPID 的高位 WORD 来找到正确的分发映射表，然后利用 DISPID 的低位 WORD 在这个分发映射表中直接得到对应的条目。这种查询方法速度是很快的，因为不需要遍历整个分发映射表（在 GetIDsOfNames()中已经遍历过一遍了）。

Invoke()的大部分工作都是在做错误检查——检查参数的个数以及返回值类型是否正确。如果分发成员是一个方法或是一个 Get/Set 属性，则 Invoke()使用 CCmdTarget::

CallMemberFunction()来调用这个方法或访问这个属性。如果分发成员是一个标准的属性（或是一个有通知方法的属性），则 Invoke()根据客户是获取还是设置这个属性，分别调用 CCmdTarget::GetStandardProperty()和 CCmdTarget::SetStandardProperty()函数。

CallMemberFunction()函数

CallMemberFunction()会断开参数链表，然后建立一个调用堆栈，接着类的成员函数就可以使用这些参数了。当堆栈的框架建立起来后，CallMemberFunction()使用_AfxDispatchCall()来建立真正的堆栈。_AfxDispatchCall()使用汇编语言使得真正的运行时栈指向 CallMemberFunction()建立的栈。接着_AfxDispatchCall()调用分发映射表条目的 pfn 成员所指向的函数。如果分发成员是一个 Get/Set 属性，则分发映射表条目的 pfn 域指向相对应的 Get...()或 Set...()函数。

SetStandardProperty()和 GetStandardProperty()函数

如果分发成员是一个属性，则 CCmdTarget 使用两个函数来获取或设置这些普通的属性，这两个函数是 CCmdTarget::SetStandardProperty()和 CCmdTarget::GetStandardProperty()。

当加入一个标准的分发属性或是加入一个带通知方法的分发属性时，ClassWizard 会在 C++类中添加一个成员变量。与此同时，ClassWizard 还会在这个类的分发映射表中插入一个 AFX_DISPMAP_ENTRY 结构。AFX_DISPMAP_ENTRY 结构的最后一个域是这个成员变量的偏移量，叫做 nPropOffset。当客户需要设置这个属性时，MFC 需要利用这个地址来访问成员变量。在 SetStandardProperty()函数中，MFC 将成员变量所在内存设为客户提供的值。如果这个属性是带通知函数的，则 AFX_DISPMAP_ENTRY 的 pfnSet 域指向这个类的通知函数。通过这种方法，任何时候只要客户改变了分发属性，SetStandardProperty()都会去调用通知函数。

GetStandardProperty()使用同样的机制工作，只不过方向相反。也就是说，GetStandardProperty()取得 AFX_DISPMAP_ENTRY 结构的 nPropOffset 域所指向的位置的值，然后将这个值填充到一个 VARIANT 中作为结果返回。Invoke()会将这个值返回给自动化客户。

正如我们看见的，MFC 已经将基本的自动化功能做得非常好了。但是作为 OLE 特性中越来越重要的类型信息，Visual C++和 MFC 又是如何对其提供支持的呢？

MFC 和类型信息

直到提供了类型库，一个自动化服务器才算真正完整了。正如前面所描述的，类型库包含了 COM 类的信息。类型库现在已经成为自动化成败的关键——特别是对于 OLE 控件而言。OLE 控件容器在很大程度上依赖于类型信息来实现 OLE 控件的事件技术。另外，使用 Visual C++的开发者可以使用类型库很快地给自动化对象写一个 C++客户端。Visual C++ 2.0 或者更高的版本的 AppWizard 会产生一个 ODL 文件，这个文件能够被编译成一个类型库。ClassWizard 甚至可以随着你加入属性和方法来及时更新 ODL 文件。

直到 OLE 控件的出现，才标志着 Visual C++和 MFC 对类型库的全面支持。在这之前的很长一段时间内，MFC 都不对 GetTypeInfo()和 GetTypeInfoCount()这两个处理类型信息的 IDispatch 接口成员函数提供支持。在 MFC 以前对 IDispatch 的实现中（在 COleDispatchImpl 类中实现），GetTypeInfo()和 GetTypeInfoCount()只是返回 E_NOTIMPL 这个错误码。

现在在 MFC 4.0 版本中，提供了对类型信息更好的支持。我们将在下面讨论它对类型库和类型信息的支持，当我们在第 15 章讨论 OLE 控件的时候将进一步深入探讨，因为类型库使用的最多的地方就是 OLE 控件。

MFC 提供了两个宏来支持类型库，这两个宏是 DECLARE_OLETYPELIB 和 IMPLEMENT_OLETYPELIB。MFC 还通过 CCmdTarget 的一些成员函数来处理类型信息。

支持类型库的宏

和 MFC 对类厂的支持一样，MFC 也提供了一些宏来支持类型库。MFC 中对类型库提供支持的宏是 DECLARE_OLETYPELIB 和 IMPLEMENT_OLETYPELIB。

如果要在自动化对象中添加类型库支持，就在对象的头文件中插入 DECLARE_OLETYPELIB 宏。下面是这个宏的定义：

```
#define DECLARE_OLETYPELIB(class_name) \
   protected: \
     virtual UINT GetTypeInfoCount(); \
     virtual HRESULT GetTypeLib(LCID, LPTYPELIB*); \
     virtual CTypeLibCache* GetTypeLibCache(); \
```

DECLARE_OLETYPELIB 会在自动化类中加入下面 3 个虚函数：GetTypeInfoCount()、GetTypeLib()和 GetTypeLibCache()。而 IMPLEMENT_OLETYPELIB 宏实现了这 3 个虚函数。下面是 IMPLEMENT_OLETYPELIB 的定义：

```
#define IMPLEMENT_OLETYPELIB(class_name, tlid, wVerMajor, wVerMinor) \
   UINT class_name::GetTypeInfoCount() \

      { return 1; } \
   HRESULT class_name::GetTypeLib(LCID lcid, LPTYPELIB* ppTypeLib) \
      { return ::LoadRegTypeLib(tlid, wVerMajor, wVerMinor, lcid, \
        ppTypeLib); } \
   CTypeLibCache* class_name::GetTypeLibCache() \
      { return AfxGetTypeLibCache(&tlid); } \
```

IMPLEMENT_OLETYPELIB 有 4 个参数：（1）类型库想要包含的信息所在的类；（2）类型库的 GUID；（3）类型库的主版本；（4）类型库的辅助版本。IMPLEMENT_OLETYPELIB 实现了 DECLARE_OLETYPELIB 声明的 3 个虚函数。GetTypeInfo()是最简单的：它返回 1，表示这个对象有类型信息。GetTypeLib()只是 LoadRegTypeLib()这个 API 函数的包装。而 GetTypeLibCache()返回一个指向这个模块的类型库缓存的指针。那么类型库缓存是从哪里来的呢？

类型库缓存

MFC 为支持类型信息的自动化应用程序保存了类型库指针。可以在 AFX_OLE_STATE 结构中找到它。AFX_OLE_STATE 结构包含了一个 CMapPtrToPtr 类型的成员，叫做 m_mapExtraData：

```
CMapPtrToPtr m_mapExtraData;
```

m_mapExtraData 的名字将来也许会改变。微软一直在调整 MFC 的性能，它已经改变了类型库的实现，使得每个模块一个类型库（最普遍的方式）。这种方式是一种优化的方式。当

MFC 看见多于一个的类型库被保存，则它就利用这个映射表来解决。也许微软当初引入这个映射表的时候，除了保存类型库外，还在别的地方使用了它。

m_mapExtraData 保存了一个 CTypeLibCache 结构的字典。这个字典将一个类型库的 GUID 映射为一个 CTypeLibCache 的实例。CTypeLibCache 包装了一个 OLE 类型库，它提供了从类型库中提取类型信息的函数。CTypeLibCache 类的定义如程序清单 14-7 所示。

程序清单 14-7 MFC 的 CTypeLibCache 类定义

```
class CTypeLibCache
{
public:
    CTypeLibCache(const GUID* ptlid);
    ~CTypeLibCache();

    void Lock();
    virtual void Unlock();
    BOOL Lookup(LCID lcid, LPTYPELIB* pptlib);
    void Cache(LCID lcid, LPTYPELIB ptlib);
    BOOL LookupTypeInfo(LCID lcid, REFGUID guid, LPTYPEINFO* pptinfo);
    void CacheTypeInfo(LCID lcid, REFGUID guid, LPTYPEINFO ptinfo);

protected:
    const GUID* m_ptlid;
    LCID m_lcid;
    LPTYPELIB m_ptlib;
    GUID m_guidInfo;
    LPTYPEINFO m_ptinfo;

    long m_cRef;
};
```

MFC 有一个全局函数叫 AfxGetTypeLibCache()。这个函数只有一个参数：一个表示类型库的 GUID。AfxGetTypeLibCache()使用这个 GUID 来查询匹配的 CTypeLibCache 类。

CCmdTarget 和 CTypeLibCache 互相协作，提供了对类型库很好的支持。

CCmdTarget 对类型库的支持

CCmdTarget 中有 6 个成员函数用来支持类型库：

1. CCmdTarget::EnableTypeLib()取得应用程序的类型库缓存并且放入一个引用计数(使用 CTypeLibCache::Lock())。

2. CCmdTarget::GetTypeInfoOfGuid()从 CCmdTarget 的类型库中取得相应的类型信息。

3. CCmdTarget::GetDispatchIID()是一个虚函数，它需要在派生类中实现。CCmdTarget 的 IDispatch 实现在调用 IDispatch::GetTypeInfo()时会使用这个函数。GetDispatchIID()会返回一个表示 CCmdTarget 的类型信息的 GUID。

4. CCmdTarget::GetTypeInfoCount()也是虚函数。它在使用类型库宏的时候自动被重载。

5. CCmdTarget::GetTypeLibCache()也是一个虚函数，它也在类型库宏中被重载。

6. CCmdTarget::GetTypeLib()也被类型库宏重载，它包装了 OLE API 函数 LoadRegTypeLib()。

Visual C++的 ControlWizard 对类型库提供了直接的支持。也就是说，无论何时创建一个 OLE 控件，ControlWizard 都会自动加入宏并且重载 GetDispatchIID 函数。在 AppWizard 对标

准自动化对象提供了这个级别的支持之前，我们都还必须自己手动地实现对类型库的支持。

结语：使用“MFC 方式”的结果

总之，MFC 提供了一个相当好的 IDispatch 实现。MFC 会在别的地方使用分发映射表的技术，因为它跟消息映射表等技术是很相似的。这点使得微软可以很容易地将 ClassWizard 集成为一个工具集合。ClassWizard 已经提供了对消息映射表的支持，那么使用分发映射表在 ClassWizard 中提供对自动化的支持是一个很自然的扩展。

MFC 的 IDispatch 实现有很高的效率。分发成员的查询是很具有弹性的，因为可以很容易地使用派生类的分发属性和方法。除此之外，MFC 对 IDispatch 的实现还很灵活，惟一跟效率相关的就是真正的分发 ID 查询。我们假设客户足够聪明，每个分发成员只调用一次这个函数并且将结果保存起来。当执行完查询后，MFC 就知道了如何去分发映射表中找到必要的信息来访问属性或是调用方法。

MFC 对 IDispatch 的实现也有缺陷，它不能很容易地支持双接口（至少目前不能）。如果想使用双接口，就必须去研究 AppWizard 产生的 ODL 文件（打断 ClassWizard），使用 Typing Wizard 来加入 COM 接口支持（所有的与接口映射表有关的内容我们在 11 章已经讨论过），然后委托 MFC 实现 IDispatch 接口。如果很多客户已经在使用 IDispatch 接口，则这点非常重要；MFC 的 IDispatch 实现要比 IDispatch 的通过类型信息驱动实现快很多。

种 Visual Basic 控件（也叫做 VBX）。VBX 主要是给程序员提供了一种定制控件的方法，以此来增强 Visual Basic 程序。另外，这种定制控件并不只限于在 Visual Basic 程序中使用：它们也可以应用于用 C++甚至 Turbo Pascal 编写的 Windows 程序。最后，这场运动催生了一个新的行业，许多公司开始提供自己设计的 VBX，例如电子数据表模块和通信模块。

然而，VBX 标准有一些缺陷。首先，VBX API 并没经过深思熟虑，结果出现了运行于不同平台上的 3 种 VBX 版本。其次，VBX API 本质上是 16 位的，这是一个严重的问题，例如：VBX API 在许多地方使用了基指针（based pointer）。最后，最大的问题是微软公司已经宣布 VBX 标准不会被带入 32 位的系统，它将推出一个替代品：OLE 控件。

OLE 控件

现在我们就进入 OLE 控件的世界。就像你在上文中所看到的那样，它几乎包含了所有现有的 OLE 技术。这意味着你可以将 OLE 控件看作一个定点对象（In-place object），可以看作一个自动化对象（automation object）（因为 OLE 控件实现了 IDispatch），或者看作一个简单的数据源（因为它实现了 IDataObject）。

OLE 控件除了可以作为定点对象和支持规则自动化（regular automation）（也就是一个接收输入的自动化接口）外，还可以支持另一种方式的 OLE 自动化（也就是一个可以输出数据的自动化接口）。换句话说，OLE 控件的容器（container）实现了 IDispatch，以使该控件可以调用它。OLE 控件中输出数据的自动化接口是做为事件实现的。OLE 控件实现了事件，使得当一些事情发生时它可以通知它的客户。例如，你可以写一个股票交易监视器，该监视器可以监视从纽约股票交易所传来的实况信息。使用 OLE 控件事件技术，当某些股票价格发生变动时 OLE 可以通知上级程序，而这种要求是无法用规则自动化来实现的，因为规则自动化只支持输入自动化接口。你也可以使用 notification 属性来通知上级程序。一个 OLE 控件的 notification 属性是一个规则属性（你可以使用 ClassWizard 加入该属性），该属性可以在属性值变化时通知上级程序。

你可以想像，实现所有的模板代码并不像在公园里散步那么容易。实际上，为了实现一个 OLE 控件，必须做大量的工作。但是，MFC 使这一切变得容易了，让我们来看看如何编写一个 OLE 控件吧。

写一个 OLE 控件

MFC 使得编写一个 OLE 控件变得十分容易。Visuall C++提供了一个说明如何建立一个简单 OLE 控件的手册，另外还有很多说明这一过程的好文章。基本上，编写一个 OLE 控件只要建立一个 OLE 控件的工程，如此就建立了一个简单的 OLE 控件，但该控件仅仅在其绘制区画一个圆。

当然，在控件编译和链接后，你需要在系统注册表中注册该控件。这并不是一个大问题，因为微软公司已经定义了两个著名的函数给控件使用：DllRegisterServer() 和

DllUnregisterServer()。DllRegisterServer()实现了在系统注册表中注册控件所需的各个步骤；DllUnregisterServer()实现了相反的操作。Visual C++也提供了一个注册工具：REGSVR32.EXE，该工具载入一个 OLE 控件的动态链接库并调用 DllRegisterServer()函数来注册该控件。顾名思义，DllUnregisterServer()取消注册并在注册数据库中清除控件的注册信息。Visual C++在它的工具条上包含了 REGSVR32.EXE 工具，如果要注册你新编译的控件，可以在 Tools 菜单中选择 Register Control，这将启动 REGSVR32 程序以注册该控件。

为了让你的控件做一些有用的工作，你需要决定你的控件应该实现哪种类型的用户接口，同时也要确定该控件的外形。另外，你还需要确定该控件应该发布哪些事件。例如：前面例子中的股票市场监视器，当某种股票价格变动时，它应该发布一个价格变动事件。

除了发布事件以外，OLE 控件也要支持规则自动化。在这种自动化中，OLE 控件的上级程序可以设置控件属性和调用控件中的函数。在股票监视器的例子中，你可能想要让上级程序设置监视的上市公司的名字。

当然，如果没有地方放置 OLE 控件，你设计的控件就毫无意义。让我们来看看如何在一个应用程序中使用 OLE 控件。

在工程里使用 OLE 控件

MFC 支持在两种情况下使用 OLE 控件：在一个对话框中或者在一个窗口中。Visual C++可以很好地支持在对话框中使用 OLE 控件，但是在一个非对话框的窗口中使用 OLE 控件却需要做一些工作。虽然可以很容易地在一个非对话框的窗口中放置 OLE 控件，但是你也需要自己实现一些事件响应机制。不过别担心，当你阅读完本章，你就会了解如何去实现这些机制。现在先让我们看看如何在一个对话框中放置一个控件。

在对话框中使用 OLE 控件

使用 Visual C++新提供的工具“Component Gallery”，你将发现在对话框中添加一个 OLE 控件是一件非常简单的事情。Component Gallery 可以很方便地将一个预先写好的软件模块加入你的应用程序中，OLE 控件当然也是如此。

Component Gallery 是一个有数个标签式页面（tabbed page）的对话框。其中一页中画满了图标，每个图标代表已经注册的一个 OLE 控件。若要将一个 OLE 控件加入到你的工程，只需高亮相应的图标并按下“Insert”按钮即可。Component Gallery 将产生一个基于该 OLE 控件的 C++类。当下一次使用对话框编辑器时，你将会在控件面板上看到这个 OLE 控件。这时若要把该控件加入对话框，只需在控件面板上选择该控件并将其放到对话框上。图 15-1 显示了一个例子。

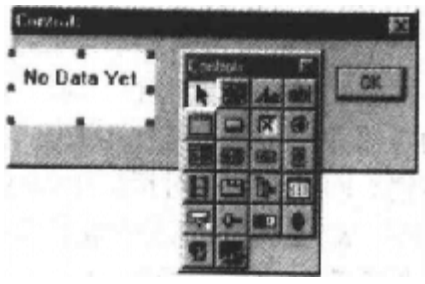


图 15-1 将股票监视控件加入到一个对话框中以及控件面板上的图标

虽然在对话框中加入一个 OLE 控件可以利用 ClassWizard 的巨大支持，但在一个非对话

框窗口中使用 OLE 控件需要做一些工作，但这并不是不可完成的任务。下面将介绍如何在一个视图（view）中加入一个 OLE 控件。

在视图里使用 OLE 控件

为了在视图里使用 OLE 控件，需要进行下面几个步骤：（1）将该控件加入工程中；（2）在你的视图里加入一个该控件的实例；（3）创建该控件。

第一步是将控件加入到你的工程中，使用 Component Gallery 来完成这一步。在主菜单中选择 Insert\Component，然后在 OLE 控件页中选择你想使用的 OLE 控件，将它加入你的工程。

在将控件加入你的工程后，就需要在你的视图里声明一个该控件的实例，可以将其作为视图类的一个成员变量。例如：如果你想把股票监视控件加入你的视图，就需要声明一个成员变量，变量的类型就是这个控件。

接下来，要为你的控件定义一个数字。当你创建该控件时，需要给它赋值一个数字（就像通用 Windows 控件都有一个控件 ID）。

将一个控件放入视图里并不困难。由于 OLE 控件在容器中可以看成一个 CWnd 类，所以你就可以像加入其它 Windows 控件一样将其加入视图。创建这个控件最好的地方是在视图的 WM_CREATE 事件处理函数里。创建这个控件需要提供标题、窗口风格标志（style flag）、位置矩形、指向父窗口的指针以及控件 ID 号：

```
m_stocks.Create(NULL, WS_VISIBLE,
                CRect(150,150,250,250),
                this, ID_STOCKSCTL);
```

做完这些后，你的屏幕上将出现这个控件，它会绘制自身并响应 Windows 消息，但是它的容器（container）却接收不到它传出的消息。为了使容器接收事件消息，容器必须实现一个事件接收器。在你将 OLE 控件加入对话框时，ClassWizard 会自动为你完成这个工作。但是为了将控件加入一个非对话框的窗口，你需要自己输入一些 MFC 的事件接收宏。我们将在介绍事件时说明这一工作（请参考本章的编程技巧一节）。

这些看起来似乎不可思议，但 OLE 控件就是如此实现了一个高级的 OLE 协议。

它是如何工作的

你在上文中已经看到了，利用 MFC 和 Visual C++，写出一个 OLE 控件或者一个 OLE 控件的容器并不困难。而使用 ClassWizard 加入一个自动化成员（函数和方法）也是一件轻松的事情。同样地，在对话框中响应 OLE 控件消息也是相当容易。但这一切并不是用魔法来实现的，MFC 为实现这些做了大量的工作。首先，我们将看一下 MFC 中支持 OLE 控件的类；然后我们将看一看从 OLE 控件创建到结束的整个过程里 OLE 控件及其容器是如何交互作用的。介绍这些的主要是以下 4 部分：（1）MFC 如何管理 OLE 控件的容器；（2）MFC 的控件容器如何创建控件；（3）MFC 如何实现 OLE 连接；（4）MFC 如何处理 OLE 控件的事件。

MFC 对 OLE 控件的支持可以分为 OLE 控件和 OLE 控件容器类。下面是对这些类的一个概述。

MFC 的 OLE 控件类

我们再看一下 OLE 控件所能做的事情：定点嵌入（In-place embedded）对象支持、自动化、属性页以及连接和事件技术。这对实现者的要求实在是高，这也许就是为什么 MFC 为 OLE 控件专门写了 25 个源代码文件的原因。

为了实现 OLE 控件，MFC 引入了几个新类，这些新类如下：

- COleControlModule。
- COleControl。
- COlePropertyPage。
- COleConnectionPoint。
- CPictureHolder。
- CFontHolder。
- CPropExchange。

下面让我们浏览一下这些类。

COleControlModule

COleControlModule 代表了一个 OLE 控件的服务端，它从 CWinApp 派生，因此它拥有 CWinApp 的所有函数。实际上这个类很简单。因为控制模块需要初始化控件并在结束后做一些清除工作，所以它使用了 CWinApp::InitInstance()和 CWinApp::ExitInstance()两个函数来达到这个目的。

这样得到的 COleControlModule 类的定义就十分简单：

```
class COleControlModule : public CWinApp
{
    DECLARE_DYNAMIC(COleControlModule)
public:
    virtual BOOL InitInstance();
    virtual int ExitInstance();
};
```

OLE 控件真正的动作是在 COleControl 类中实现的。

COleControl

这是 OLE 控件最主要的类，它封装了一个简单的控件，并处理了一大批控件函数。该类由 CWnd 类继承而来，因此它继承了 CWnd 类的所有函数。此外，COleControl 类也支持对事件的处理。因为 COleControl 类本质上是从 CCmdTarget 类继承而来，所以它同时也处理了 IDispatch（这样 OLE 控件就可以支持自动化属性和方法了）。

因为 COleControl 类是 OLE 控件类中最主要的类，所以你可以想像到它肯定支持了一些 OLE 接口。实际上，COleControl 类实现了许多接口。仔细看一看 CTLCORE.CPP 文件，你就可以发现一个接口映射表，这个映射表包含了所有以下接口：

- IOleObject——利用这个接口，定点对象可以与容器交换信息。它有 DoVerb()和 Close()

两个函数。

- IConnectionPointContainer——这个接口允许客户端询问控件是否支持连接点（Connection Point）。
- IOleControl——这个接口给 OLE 控件提供了一些函数，扩展了定点对象的功能。
- IPersist——这个接口返回一个 COM 对象的 GUID。
- IPersistMemory——这是为控件设计的一个新接口（它在 OLECTL.H 文件中定义）。它为快速持续性（persistence）提供了一个接口。该接口主要用来代替原有的 IPersistStreamInit 接口，以提供更好的性能。原有的接口是将持续性处理成一个数据流，而现在你必须访问一个缓冲区，IPersistMemory 接口给了控件访问容器缓冲区的权限。
- IPersistStreamInit——控件实现此接口主要是为了使容器可以告诉控件保持持续性。
- IOleInPlaceObject——这个接口给了容器一个激活和停止定点对象的方法，同时也管理着有多少定点对象应该是可见的。
- IOleInPlaceActiveObject——OLE 文档内容对象实现这个接口是为了在下述对象间建立一个直接的渠道（1）一个定点对象；（2）对象服务端程序的最外面的窗口；（3）容器程序中的文档窗口（document window）。这个接口用于传送消息、管理窗口状态（激活或者禁用）以及管理文档窗口的状态（激活或者禁用）。这个接口也可以在要改变其边界大小时通知内容对象（content object），并且管理无模式（modeless）对话框。
- IDispatch——用来提供一个控件的自动化属性和方法。
- IOleCache——这个接口给控件提供了隐藏在对象内部的数据。即使服务端程序不在运行或无效，这些隐藏的数据对对象的容器而言也是可见的。
- IViewObject——这个接口使控件可以直接显示自身，而不用传送数据给调用者。另外，这个接口可以建立和管理一个拥有接收器的连接，凭借这个可以在显示对象变化时通知调用者。
- IViewObject2——这个接口是 IViewObject 接口的扩展，可以返回一个对象绘图区的大小。只要对象没在运行，你就可以调用这个方法来阻止其运行，而不用调用 IOleObject::GetExtent。
- IDataObject——这个接口被用来通知容器发生的变化。
- IPersistPropertyBag——这是 OLE 控件的一个新接口，它给控件提供了一个保持其属性持续性的新方法。控件可以实现这个接口，以代替原来的 IPersistPropertySet。这个接口基本上是一个完成基于文本的持续性的快速方法（就像你在 Visual Basic 的.FRM 文件中看到的那种持续性）。
- ISpecifyPropertyPages——容器可以使用这个接口来查询一个 OLE 控件支持哪些属性页。
- IPerPropertyBrowsing——这个接口支持一次浏览所有属性页（与一次浏览一个属性页相比）。它被用来实现一个属性修改的用户界面，就像 Visual Basic 的属性格（property grid）或者 Visual C++的“All Properties”属性页。
- IProvideClassInfo——这个接口由控件实现，容器可以使用它来获得控件的事件和通知属性的类型信息。

- IPersistStorage——这个接口由控件实现，容器可以使用它来告诉控件在 IStorage 中保持持续性。

因为 COleControl 类是 MFC 支持 OLE 控件的最主要的一个类，所以本章将花费大部分篇幅介绍这个类。

COlePropertyPage

OLE 控件是一个自动化对象，并且通过 IDispatch 向外提供属性。大多数的 OLE 控件允许用户通过属性页来访问属性。这些属性页通常被合在一起并在一个带标签的对话框中显示出来。COlePropertyPage 类主要就是处理如何在一个对话框中显示 OLE 控件的属性。

COleProperty 类由 CDialog 继承而来，因此它拥有所有标准的对象。正是这些对象使得我们可以如此容易地使用一个 MFC 对话框。例如：COlePropertyPage 完全支持 DDX/DDV。另外，COlePropertyPage 类也扩展了数据交换机制，以使属性可以在属性页和控件之间交换。

COleConnectionPoint

一个区分 OLE 控件和一般无格式的自动化的方法是，OLE 控件可以完成以下功能：（1）向上级程序发布事件；（2）当属性变化时通知上级程序。OLE 控件通过连接技术来实现这些功能。而 COleConnectionPoint 就代表了 OLE 控件输出数据的接口。别担心，我们将在介绍 OLE 控件事件时介绍这个类的细节。

CPictureHolder

CPictureHolder 类主要是为了实现 OLE 控件的图像属性。使用这个类，你可以定义一个位图、图标或者媒体文件，并用同样的方法显示它们。

CFontHolder

CFontHolder 类和 CPictureHolder 类有些相似，它封装了 Windows 字体对象和 IFont 接口的功能。它被用来实现 stock 字体属性。可以用这个类为你的控件实现自己的字体属性。

CPropExchange

在给出了 OLE 控件的属性后，你可能会希望这些属性可以保持持续性，而 CPropExchange 类（和它的子类）支持 OLE 控件属性的持续性。

在另一方面，MFC 包含了实现 OLE 控件容器的一些类。让我们在深入 OLE 控件前再来看一看这些类。

MFC 和 OLE 控件的容器

一直到 1995 年底，我们在 OLE 控件上惟一能做的工作就仅仅是创建它们。微软公司提供了一个方便的 OLE 控件测试容器，但是它从未显示源代码，这使得我们没有一个简便的方法去观察程序的运行。而 MFC 4.0 的 OLE 控件容器最终完全支持了这些，但是尽管它使你从头实现一个控件容器变为可能，但它依然不是一个吸引人的好方法，因为我们仍然必须手动编写几个 OLE 接口。现在，MFC 和 Visual C++终于可以让我们非常容易地在基于 MFC 的

程序中使用 OLE 控件了——尤其是在对话框中。事实上，微软公司扩展了资源编辑器和 ClassWizard，使得 OLE 控件就像其它通用控件一样。现在我们可以在控件面板上看到 OLE 控件，而且可以在消息映射表里看到控件事件了。另外，OLE 控件不再只限制在对话框中使用了：你可以在任何继承自 CWnd 的对象中使用 OLE 控件。

MFC 对 OLE 控件容器的支持绝大部分依赖于 3 个类：COleControlContainer、COleControlSite 以及 COccManager。

COleControlContainer

COleControlContainer 类实现了 IOleInPlaceFrame 和 IOleContainer，这些对 OLE 控件实现它们的定点对象是必需的。在 OLE 控件容器的设计里，一个继承 CWnd 的对象如果想包含 OLE 控件，它就必须有一个 COleControlContainer 类的成员变量，这个类实际上管理着一个或数个 COleControlSite 对象。

COleControlSite

COleControlSite 类只在文档中提过一次，就是在 CWnd::OnAmbientProperty()的参数中出现过一次。除此之外，该类没有其他记录。但是，这个类可以说是 MFC 对 OLE 控件支持的核心，它处理了 OLE 控件及其容器之间的信息交流。在一个容器内部，每个 OLE 控件都有一个该类对象。此外，该类还负责在一个容器内实际创建每一个独立控件。

COleControlSite 是一个真正的 COM 对象，它实现了下列几个接口：

- IOleClientSite——规则 OLE 文档支持。
- IOleInPlaceSite——规则定点激活支持。
- IOleControlSite——这个接口由容器实现，可以让控件使用。控件使用它主要是为了通知容器一些变化信息，或者从容器处得到一些变化信息。
- IDispatch（基于事件）——这个接口由容器实现，控件可以通过它把事件传达给容器。
- IDispatch（基于环境变量）——这个接口由容器实现，控件可以通过它访问容器的环境变量。
- IPropertyNotifySink——这个接口由容器实现，当控件属性变化或者要求编辑变化的属性时，控件可以通这个接口通知容器。

当我们介绍 OLE 控件代码如何工作时，我们将对 COleControlSite 作出更多的说明。现在让我们来看一看另一个类：COccManager。

COccManager

这个类提供了两种服务：（1）管理对话框中的 OLE 控件；（2）管理事件的走向，引导事件到达正确的响应目标。MFC 包含了一个该类型的全局对象，对象名是 afxOccManager。

上述这些是 OLE 控件容器类的一个纲要性介绍，现在让我们看看 OLE 控件是如何与其容器交互作用的。当容器创建了 OLE 控件时，它们将通过接口交流，并且处理事件。

OLE 控件的生存周期

请记住 OLE 控件实际上是一个 COM 对象，它们实现了一些接口，以使得客户端可以与

它们交流信息。回忆一下第 11 章的内容，在该章中，COM 对象具体化成为客户端调用 CoCreateInstance()的结果。再回忆一下 CoCreateInstance()是通过检查注册表来找到实现 COM 对象的服务器的路径的。一旦 CoCreateInstance()找到相应的服务器，它就会载入该服务器。在 OLE 控件中，服务器就是一个 DLL，这样服务控制管理器（Service Control Manager）就会载入那个 DLL。一旦 DLL 被载入，操作系统就会提供一个入口，该入口就是通往著名的 DllGetClassObject()函数的。然后我们将获得 COM 对象库并且返回指向客户端的 IclassFactory 接口指针。这个客户端调用 IClassFactory::CreateInstance()以创建一个 COM 对象的实例，并返回要求的接口。

到了这里，客户端使用新获得的接口来储存更深的接口，并且打开一个适应该控件的对话框。让我们看看这一过程——这个过程确实十分有趣。

控件的创建

控件总是来自于某个地方，在 MFC 中有两种产生 OLE 控件的方法：（1）把控件加到一个对话框模板中；（2）调用外部类的 Create()函数。让我们分别看看每一种方法，先看看对话框中的 OLE 控件。

在对话框中创建 OLE 控件

既然对话框能够包含一种新的控件（就是 OLE 控件），你可以想像微软公司不得不修改他们的对话框模板代码，以使得对话框可以包含 OLE 控件。如果你看过对话框中创建 OLE 控件的代码，你将会发现对话框模板信息确实改变了。

OLE 控件是在对话框的 WM_INITDIALOG 消息处理函数中创建的。CDialog::HandleInitDialog()就是处理这个消息的函数，该函数得到外框架的 COccManager（即 afxOccManager），然后使用对话框模板调用 afxOccManager 的 CreateDlgControls()函数。

COccManager::CreateDlgControls()函数用 LoadResource()函数取来对话框模板，接着 COccManager::CreateDlgControls()循环创建对话框中的每个控件（用一个 DIALOGITEMTEMPLATE 结构）。对于对话框中的每个 OLE 控件，COccManager::CreateDlgControls()调用 COccManager::CreateDlgControl()函数来创建控件。

如果 DIALOGITEMTEMPLATE 结构表示了一个 OLE 控件，它将包括一个叫做 GUIDs 的字符串。为了从对话框模板中创建一个控件，CreateDlgControl()函数使用一个 OLE API 函数 CLSIDFromString()来把类的 ID 串传送进 GUID。CreateDlgControl()同时也获得许可密钥（license key）（如果需要它创建控件）。

CreateDlgControl()函数的第一个参数是父窗口（在本例中，就是对话框）。回忆一下，OLE 控件总是存在于某个窗口中（在本例中，这个窗口就是对话框），这个窗口保存了一个 COleControlContainer 对象，该对象管理着一个或多个 COleControlSite 对象。在 CWnd 类中有一个成员变量 m_pCtrlCont，该变量类型就是 COleControlContainer。另外 CWnd 还有一个成员函数 InitControlContainer()，该函数创建一个新的 COleControlContainer 对象，并把这个对象赋值给 CWnd::m_pCtrlCont。CreateDlgControl()函数就是调用 InitOleContainer()函数来为对话框创建一个 COleControlContainer，以用它来存放控件。

COleControlContainer 继续通过它的一个成员函数 CreateControl()来创建控件。结果是，

COleControlContainer::CreateControl() 创建一个新的 COleControlSite 对象，并调用 COleControlSite 的 CreateControl()函数来创建实际的控件。

如果需要，COleControlSite::CreateControl()将初始化 OLE，然后调用 COleControlSite::CreateOrLoad()。CreateOrLoad()使用一个没有文档说明的函数 CoCreateInstanceLic()（你可以在 OCCSITE.CPP 文件中找到它）来创建控件。下面是 CoCreateInstanceLic()函数的声明：

```
HRESULT CoCreateInstanceLic(REFCLSID clsid, LPUNKNOWN pUnkOuter,
                            DWORD dwClsCtx, REFIID iid, LPVOID* ppv,
                            BSTR bstrLicKey)
```

除了有一个 license key 参数外，这个函数非常像规则的 API 函数 CoCreateInstance()。如果没有许可信息，CoCreateInstanceLic()使用 CoGetClassObject()函数获得控件类库，同时要求传入一个 IClassFactory 接口。然后 CoCreateInstanceLic()函数就可以简单地调用 IClassFactory::CreateInstance()来创建控件的一个实例。如果控件需要许可信息，那么 CoCreateInstanceLic()也将用 CoGetClassObject()函数来获得控件类库，但要求的是 IClassFactory2 接口（它是 IClassFactory 接口的第二次迭代，微软公司认为这是处理许可的最好地方）。然后 CoCreateInstanceLic()将调用 IClassFactory2::CreateInstanceLic()，同时传送许可密钥。这样控件就可以检查是否被使用在适当的地方。

在这里，执行线程就将进入控件模块。回忆一下第 11 章，调用一个 COM 对象库的 CreateInstance()函数将创建该对象。后面我们将看看另一方面将发生什么，但是现在让我们首先看看一个独立的（stand-alone）控件是如何创建的。

创建一个独立的控件

当你在对话框以外创建 OLE 控件时，你肯定是在一个规则的从 CWnd 派生的对象中创建控件。在本章开头提到的股票监视器的例子中，你为股票监视控件创建了一个封装类（wrapper class），并使用 Component Gallery 将控件加入你的工程中。这个封装类就是从 CWnd 派生的，并且包含了使用控件自动化属性和方法的一些成员函数。另外，由 Component Gallery 产生的这个封装类重载了 CWnd::Create()函数（这是一个虚函数）。这样如果你想把这个股票监视控件加入你的视图里，只需要声明该封装类的一个实例作为视图类的成员变量，并且通过调用封装类的 Create()函数来处理 WM_CREATE 即可，如下所示：

```
m_stocks.Create(NULL, WS_VISIBLE,
                CRect(150,150,250,250),
                this, ID_STOCKSCTL);
```

这个封装类的 Create()函数调用 CWnd::CreateControl()函数，以把继承自 CWnd 的对象转变成一个 OLE 控件端，这样就使得它可以装入 OLE 控件（记住，实际控件存在于某个 DLL 中）。

```
BOOL CWnd::CreateControl(REFCLSID clsid, LPCTSTR lpszWindowName,
                         DWORD dwStyle, const RECT& rect,
                         CWnd* pParentWnd, UINT nID, CFile* pPersist,
                         BOOL bStorage, BSTR bstrLicKey)
```

CWnd::CreateControl()将调用父窗口（本例中父窗口就是视图）的 InitControlContainer 函数。该函数将为视图创建一个 COleControlContainer 对象，并且把这个对象赋值给视图的

m_pCtrlCont 变量。然后 CWnd::CreateControl()函数将调用视图类的 m_pCtrlCont->CreateControl()。

在这以后，控件的创建就与在对话框中的创建类似了。控件端得到控件的类库（使用CoGetObject()函数），然后用类库产生一个控件实例。控件容器将要求返回控件的 IOleObject 接口，容器将用此接口与控件交流信息。

控件的内部

一旦 DLL 被载入内存，容器将会调用 DLL 的那个著名函数——DllGetClassObject()，这个函数将调用 MFC 的 AfxDllGetClassObject()函数。回忆一下第 11 章的内容，AfxDllGetClassObject()函数会浏览类库的 DLL 链表以找到正确的类库。如果 AfxDllGetClassObject()函数找到了正确的类库，它将返回 IClassFactory 指针，这样容器就可以创建控件了。容器还会使用类库的 CreateInstance()函数，该函数在类库中结束。而控件类库实际上使用 MFC 的动态创建机制来创建控件对象。

构造控件

很自然，执行线程是在控件的构造函数中结束的。MFC 给 COleControl 提供的默认构造函数可以初始化所有成员变量，它将做如下几项工作：

- 调用 EnableAggregation()函数使控件可聚集。
- 调用 EnableAutomation()函数为规则自动化启动 IDispatch vtable。
- 调用 EnableConnection()函数，使得控件及其容器可以交流连接（例如，事件和属性通知）。顺便说一下，EnableConnection()的实现很像 EnableAutomation()函数，它把一个 IconnectionPointContainer vtable 插入到 OLE 控件机制中去。
- 调用 AfxOleLockApp()函数，使程序使用控件时，DLL 都保留在内存里。

在 COleControl::COleControl()函数结束后，就将执行子类的构造函数。ControlWizard 会在控件的构造函数中插入一个对函数 InitializeIIDs()的调用。COleControl 类保存了 GUID 以表示控件的基本分发接口和事件接口。InitializeIIDs()函数也将调用 EnableTypeLib()函数来启动对控件的类型库支持。

作为对安全性的一个特别测试，InitializeIIDs()函数验证了类型库中是否包含了正确的信息。如果在 CPP 文件中的 GUID 和 ODL 文件中的不一样，MFC 将在这里发出一些断言（assertion）。

到了这里，控件已经被构造好了，执行线程将返回到容器一方。

结束控件的创建

在控件创建后，容器仍然需要做一些工作。记住，OLE 控件也是一个定点对象，因此容器需要建立通信部分，同时也要建立必要的连接（就像事件接收器和属性通知器）。

在 COleControlSite::CreateOrLoad()函数内部，通过调用控件类库的 CreateInstance()函数，容器就将形成了。

到了这里，控件就可以通过它的状态位来立刻表明它想要拥有容器的 COleControlSite 了。如果控件想要做一些事，例如访问容器的环境变量，这一点就变得很重要了。容器可以使用

控件的 IOleObject::GetMiscStatus()函数来获得控件的各个状态位，可以检查 OLEMISC_SETCLIENTSITEFIRST 位是否被设置了，如果该位被设置了，就意味着控件想要立即同客户端对话。这些使得控件可以和容器的环境变量保持同步。

到了这里，容器需要让控件序列化它的属性。首先，容器试图通过 IPersistMemory 初始化控件，容器将向控件请求它的 IPersistMemory 接口（通过控件的 IOleObject 指针调用 QueryInterface()函数）。容器将调用控件的 IPersistMemory::Load()函数，这中间将使用一个临时的 CMemFile 对象，该对象在容器的 COccManager::CreateDlgControl()函数中创建。在容器的临时 CMemFile 对象外，控件创建了一个 CArchive 对象，并使用它调用控件的 Serialize()函数。COleControl::Serialize()函数将调用控件的 DoPropExchange()函数来在控件及其容器间交换持续性属性。

如果控件没有实现 IPersistMemory 接口，控件容器将执行一个控件的 QueryInterface()函数来得到控件的 IPersistStreamInit()接口，这样容器就可以使用一个流（stream）来存取控件属性了。这时 OLE 控件就只被看作一个 CArchive 对象——它可能基于一个 CMemFile 类或者一个 OLE 流。

作为 CreateOrLoad()函数的最后一步（在控件属性被序列化后），容器把它的 IClientSite 接口交给控件（这是作为一个定点嵌入对象的要求）。

如果 COleControlSite::CreateOrLoad()函数执行成功，控件就诞生了，并且准备好进行它的工作了。然而，容器仍然需要连接控件，以使控件可以（1）将事件传达给容器；（2）当属性变化时通知容器。这些将把我们带到 OLE 控件非常有趣的一个方面：OLE 连接。

OLE 连接

OLE 是通过连接技术实现事件的。OLE 连接包含了一个特殊类型的接口“连接点”（Connection Point）。通用的 OLE 接口实现和提供 OLE 对象的功能，这些通用接口叫做入接口（incoming interface）。而一个连接点实现的接口走的是另一条路：出接口（outgoing interface）。

一个 OLE 接口包括两部分：源对象（调用连接接口的对象）和接收对象（实现连接接口的对象）。在使用 OLE 控件中，OLE 控件就是源对象而容器就是接收对象。通过一个连接点，OLE 控件允许控件容器和控件建立一个连接。使用连接点机制，一个 OLE 控件可以获得一套由容器实现的函数的指针。

在默认情况下，一个继承自 COleControl 的类实现了两个连接点：一个是给事件，另一个给属性变化通知函数。事件连接用来传达事件，而属性变化通知连接用来在一个属性值变化时通知容器。当然，一个 OLE 控件可以随意生成自己的接口。

OLE 连接接口

为了支持连接，OLE 定义了两个接口：IConnectionPoint 和 IConnectionPointContainer。IConnectionPoint 接口仅仅在 OLE 控件和容器间定义了一个单独的连接，而 IConnectionPointContainer 保持了这个连接点。

下面是 IConnectionPointContainer 接口的定义：

```
interface IConnectionPointContainer {
public:
   virtual HRESULT EnumConnectionPoints(LPENUMCONNECTIONPOINTS* ppEnum) = 0;
   virtual HRESULT FindConnectionPoint(REFIID iid,
                                       LPCONNECTIONPOINT* ppCP) = 0;
};
```

在默认情况下，OLE 控件保持了两个连接点：事件和属性通知接口。每一个连接点都是由一个实现了 IConnectionPoint 接口的对象单独表示的。

另一个连接接口是 IConnectionPoint，此接口仅仅在两个 OLE 对象间定义了一个单独的连接。下面是 IConnectionPoint 接口的定义：

```
interface IConnectionPoint {
public:
  virtual HRESULT GetConnectionInterface(IID* pIID) = 0;
  virtual HRESULT GetConnectionPointContainer(
                  IConnectionPointContainer** ppCPC) = 0;
  virtual HRESULT Advise(LPUNKNOWN pUnkSink,
                         DWORD* pdwCookie) = 0;
  virtual HRESULT Unadvise(DWORD dwCookie) = 0;
  virtual HRESULT EnumConnections(LPENUMCONNECTIONS* ppEnum) = 0;
};
```

上述两个接口都拥有实现一个可连接对象的所有必要函数。现在让我们来看一看连接是如何建立的。

建立一个连接

OLE 控件容器可以调用 QueryInterface()函数来检查控件的 IConnectionPointContainer 接口，以便了解控件是否支持连接。如果控件返回了一个有效的 IConnectionPointContainer 指针，则控件就能使用 IConnectionPointContainer::FindConnectionPoint()函数查询控件的实际连接，同时使用 GUID 请求接口。例如，如果容器想要和控件的事件接口建立一个连接，容器将调用 FindConnectionPoint()函数，同时使用 GUID 表示控件的事件接口。

记住，容器已经实现了控件希望实现的事件接口。一旦容器找到了正确的连接，容器就可以调用 IConnection::Advise()把它的接口指针插入控件的事件集合中。IConnectionPoint::Advise()函数保存了接口，以提供控件以后使用。例如：如果容器连接上了控件，并且把它的 IDispatch 接口插入到 IConnectionPoint::Advise()函数中，控件就可以调用容器的 IDispatch::Invoke()函数了。

MFC 和连接

因为 OLE 控件主要是依靠 MFC，所以 MFC 必须实现 IConnectionPoint 和 IConnectionPointContainer 这两个接口。MFC 是在一个没有文档说明的 COleConnPtContainer 类中实现 IConnectionPointContainer 接口，你可以在 OLECONN.CPP 文件中找到该类。而 IConnectionPoint 接口是在 CConnectionPoint 类中实现的，你也可以在 OLECONN.CPP 文件中找到这个类。让我们回到 OLE 控件的创建过程去看一看 MFC OLE 控件是如何与容器建立连接的。

我们停在 COleControlSite::CreateControl()函数中。COleControlSite::CreateOrLoad()刚刚完成了它的工作，现在控件和容器需要建立连接。为了建立事件连接，COleControlSite 类使用 IprovideClassInfo2::GetGUID()来向控件请求它的事件接口 GUID，同时请求表示控件基本事件集的 GUID。这中间容器会使用一个叫做 GUIDKIND_DEFAULT_SOURCE_DISP_IID 的常数，该常数在标准头文件中定义。当控件看到这些时，它就知道应该传回事件接口的 GUID。

现在容器需要用某种方式来实现 IDispatch，以使得控件可以使用它将事件通知容器。这个特别的 IDispatch 接口通常被称为事件接收器（event sink）。在 MFC 中，事件接收器是在 COleControlSite 中实现的。COleControlSite 的事件接收器是一个嵌套的成员类，叫做 m_xEventSink。这个类是一个规则 IDispatch 接口（使用 GetTypeInfo()、GetTypeInfoCount()、GetIDsOfNames()和 Invoke()）。然而就像我们很快要看到的那样，它的实现和标准的 MFC dispatch 接口不大一样。COleControlSite::CreateControl()函数使用另一个成员函数 ConnectSink()来把容器的事件 IDispatch 和控件相连。ConnectSink()函数将把控件端的 dispatch 接口传给事件接收器，即成员类 m_xEventSink。

ConnectSink()函数可以获得控件的 IConnectionPointContainer 接口（当然是使用 QueryInterfance()接口）。如果容器能获得这个接口，容器就能知道控件支持连接。接下来，ConnectSink()函数使用控件的 IConnectionPointContainer::FindConnection()函数来为事件集找到连接点。这个将带给我们又一个 MFC 映射表：连接映射表。

连接映射表

MFC 的连接映射表非常像我们在第 11 章中看到的接口映射表。记住，接口映射表是 MFC 在一个单独的 C++类中实现多接口的方法，主要使用了嵌套类和表驱动（table-driven）机制。而 MFC 对 COM 的支持可以明确地划分为两套宏。

第一套宏（BEGIN_INTERFACE_PART/END_INTERFACE_PART）是在一个单独的从 CCmdTarget 派生的类中定义嵌套类。可以用这对宏命令加入每一个用 COM 类实现的接口。在加入这对嵌套类的宏后，你可以声明接口函数的原型。通过预处理程序处理这些声明，将会通过嵌套类产生实现多接口的从 CCmdTarget 派生的类。

第二套宏（DECLARE_INTERFACE_MAP、BEGIN_INTERFACE_MAP/END_INTERFACE_MAP 以及 INTERFACE_PART）通过 QueryInterface()定义了一个查询表。MFC 的接口映射表使一个 GUID 和嵌套类的 vtable（客户端希望把这个看作一个接口）相连。每次当你使用 INTERFACE_PART 时，你就把一个 AFX_INTERFACEMAP_ENTRY 结构加入了消息映射表中。CCmdTarget 的 QueryInterface()函数将在接口映射表中搜索接口的 GUID，如果它找到接口，就会返回接口的 vtable。

MFC 的连接映射表基本上以同样的方式工作，你可以使用宏把 OLE 连接加入任一个继承自 CCmdTarget 的类中。

MFC 有两个宏（BEGIN_CONNECTION_PART 和 END_CONNECTION_PART），它们定义了一个嵌套类来表示连接。下面就是这两个宏，请注意它们与 BEGIN_IINTERFACE_PART 和 END_INTERFACE_PART 两个宏是多么的相像：

```
#define BEGIN_CONNECTION_PART(theClass, localClass) \
class X##localClass : public CConnectionPoint \
```

```
{ \
public: \
   X##localClass() \
   { m_nOffset = offsetof(theClass, m_x##localClass); }
#define END_CONNECTION_PART(localClass) \
} m_x##localClass; \
friend class X##localClass;
```

MFC 对嵌套类的支持需要你声明接口函数的原型。连接的部分工作与之相似：你要在你的连接里声明你想要的的函数原型。如果你想重载任何 CConnectionPoint 类的成员函数或者加入你自己的成员函数，需要在上述两个宏中声明它们。例如：MFC 在 COleControl 类中为控件事件定义了一个连接点：

```
// Connection point for events
BEGIN_CONNECTION_PART(COleControl, EventConnPt)
   virtual void OnAdvise(BOOL bAdvise);
   virtual REFIID GetIID();
END_CONNECTION_PART(EventConnPt)
```

把这些宏加入到 COleControl 类中，你就定义了一个代表事件连接点的嵌套类，并且带有两个函数：OnAdvise()和 GetIID()。

MFC 同时也在 COleControl 类中为属性通知（notification）定义了一个相似的连接点：

```
// Connection point for property notifications
BEGIN_CONNECTION_PART(COleControl, PropConnPt)
   CONNECTION_IID(IID_IPropertyNotifySink)
END_CONNECTION_PART(PropConnPt)
```

上面的 CONNECTION_ID 宏可以展开成如下声明：

```
REFIID GetIID() { return iid; }
```

使用宏建立连接点只能说是完成了一半的工作，MFC 还需实现一个查询机制。也就是另外一套映射宏：DECLARE_CONNECTION_MAP、BEGIN_CONNECTION_PART/END_CONNECTION_MAP 以及 CONNECTION_PART。

鉴于接口映射表建立一个 AFX_INTERFACEMAP_ENTRY 结构的表，连接映射表也构建了 AFX_CONNECTION_ENTRY 结构的表。

```
struct AFX_CONNECTIONMAP_ENTRY
{
   const void* piid;
   size_t nOffset;
};
```

这个 AFX_CONNECTIONMAP_ENTRY 结构把连接接口 ID 映射到连接接口 vtable 的地址。MFC 使用这个表来查询 OLE 控件的连接点。

这样，MFC 就通过连接映射表实现了连接点。让我们回头继续看一下控件和容器是如何建立连接的。

建立一个连接（续）

我们上面说到要在 OLE 控件和容器间建立一个连接点，容器要把它的事件接收器插入到

控件的连接中，而控件也要尝试着使用控件的 IConnectionPointContainer::FindConnectionPoint()函数来找到事件集的连接点。

控件的 FindConnectionPoint()函数将搜寻适当的连接。如果容器请求事件连接，那么 FindConnectionPoint()函数将迅速地返回事件的连接点。否则，该函数将查找连接映射表，以获得一个相应的接口 GUID。

如果容器找到了正确的连接，它将调用连接点的 Advise()函数，把它的事件接收器插入到控件中。对控件来说，控件的连接点将把接口（例如事件的 IDispatch 或者属性通知器）插入到它保持的接口链表中。因为连接点保持了多个接口，所以在某种意义上控件能把事件传送给多个控件端。

一旦容器连接了它的事件 IDispatch，容器将把它的属性通知器和控件连接起来。

接下来，COleControlSite::CreateControl()调用 ColeControSite::DoVerb()函数来定点激活控件。而 DoVerb()函数只是调用控件的 IOleObject::DoVerb()函数。

到了这里，控件已经完全运转起来，它可以和容器交流信息，而且已经准备好把事件发送给容器了。现在让我们看看 OLE 控件的事件是如何工作的。

OLE 控件的事件

究竟什么是 OLE 控件事件？我们需要先定义这个术语。一个 OLE 控件事件就是当控件中发生了一件我们感兴趣的事情时，控件的上级程序收到的通知。例如：想像一下你有一个从一个实时数据源（例如上文中提到的纽约股票交易）收集数据的 OLE 控件。你可以在程序中使用这个控件来监视市场上的价格变化。控件可以用一个事件来向你的程序通知关于股票价格的变化情况。

使用 ClassWizard 往 OLE 控件中加入一个事件是十分容易的。只需要按下“Add Event”按钮，输入一个事件名，并且加入一些必要的参数，ClassWizard 就会在你的控件里加入相应的函数。无论何时你想要将事件通知容器，只需要调用控件里相应的函数即可。

例如，再看看上文中的股票监视控件，当某种股票价格变化时，一个你所感兴趣的事件就发生了。为了把这个事件加入你的股票监视控件，你首先应该打开 ClassWizard，并按下“Add Event”按钮，接着在事件名一栏内输入“CurrentPriceChange”，这样 ClassWizard 就将在你的控件里加入 FireCurrentPriceChanged()函数。此外，你可能想要把旧的价格和新的价格都告诉容器，因此你还可以给事件函数加一些参数。无论何时你想告诉容器此事件发生了，只需要调用 FireCurrentPriceChanged()函数即可，该函数自然会把事件通知容器。

OLE 控件的事件机制确实是非常有趣的，因此让我们来仔细看一看。

如果你已经理解了 Windows 的回调函数的工作方式，你将很容易了解 OLE 控件事件的工作。先想一想 Windows 的事件工作方式，Windows 只是一直坐在那等待某些事件发生——比如鼠标的移动和键盘的敲击。无论何时，一旦某个感兴趣的事件发生了，Windows 就会告知应用程序处理该事件。为了实现这个机制，每一个在 Windows 下运行的应用程序都会为它的每一个窗口注册一个回调（callback）函数——一个 Windows 过程（有时叫做消息处理函数）。如果一个事件属于某个窗口（例如：鼠标在窗口内的单击事件），Windows 就会

产生一个消息，并且调用消息处理函数。这里主要的思想就是每一个应用程序都提供了一个函数并在 Windows 内注册它，这样 Windows 就可以用这个函数来通知应用程序每个发生的事件。

OLE 控件事件背后的基本思想和 Windows 的回调函数的思想一样，OLE 控件的事件机制只是更加灵活和强大一些。

这里有一个了解事件的更好方式。OLE 控件容器实现了处理事件的函数。为了激发一个事件，OLE 控件调用由容器实现的事件处理函数。可以从规则的 Windows 结构类推一下，消息处理函数实际上是一个很大的事件接收器，该接收器不断地被事件源（这里就是 Windows 本身）调用。

OLE 控件容器实现了一些类似于回调函数的东西，然而它并不是一个函数，而是一个接口。控件容器实现了一个自动化接口并且使控件可以使用该接口。这样控件就可以通过这个接口把事件通知给容器。

让我们看一看 MFC 是怎么做到这些的。

激发事件

每次当你加入一个事件，ClassWizard 都会在你的代码里加入一个函数。在股票监视器的例子里，ClassWizard 加入了一个 FireCurrentPriceChanged()函数：

```
void FireCurrentPriceChanged(double dOldPrice, double dNewPrice);
```

这个函数实际上与 COleControl::FireEvent()函数一样。FireEvent()函数使用一个分发 ID 和一个指向表示事件参数的字节数组指针。它用 va_start()函数建立一个变量参数链表，然后调用变量参数的启用链表函数来激发事件，即 FireEventV()函数。FireEventV()函数可以从控件得到一个代表控件事件的 dispatch 接口。

控件之所以知道容器的事件 dispatch 接口，是因为连接是在控件创建时建立的。让我们来看一看 MFC 是如何处理 OLE 控件事件的。

MFC OLE 控件事件

OLE 控件通过调用一个接口内的函数来激发事件，而 FireEventV()函数必须获得容器的接口。下面介绍 OLE 连接是从哪儿进来的。

MFC 的 COleControl 类有一个类型为 CConnectionPoint 的成员类 m_xEventConnPt。为了调用事件，FireEventV()函数首先需要从容器处得到事件处理函数。它将使用 m_xEventConnPt 的 GetConnection()函数得到连接。CConnectionPoint 类保存了一个接口指针的数组，同时也保存了连接接口数组。当容器创建控件时，容器会向控件请求表示事件接口的连接点，然后使用 IConnection::Advise()函数插入事件接口。

在 MFC 里，标准的事件连接也是一个 dispatch 接口（或 IDispatch 指针）。为了执行容器事件，FireEventV()在栈里声明了一个 COleDispatchDriver 对象。COleDispatchDriver 类是一个方便的 MFC 类，该类封装了一个 IDispatch 指针，使得我们访问 IDispatch 的方法和属性变得更加容易。

接下来，对每一个由控件的 m_xEventConnPt 对象代表的连接，FireEventV()将从

m_xEventConnPt 处获得容器的 IDispatch 指针。这些连接是用一个 IDispatch 指针的数组表示的。然后 FireEventV() 使用 COleDispatchDriver::Attach() 函数把 IDispatch 接口和 COleDispatchDriver 对象连接起来。一旦这个连接建立起来，FireEventV()只需要传入事件的 dispatch ID 和参数链表，就能容易地使用 COleDispatchDriver::InvokeHelper()函数调用容器的 IDispatch。在控件调用完容器的方法后，FireEventV()将断开 COleDispatchDriver 对象和容器的 IDispatch 指针。

以上说明了 OLE 控件是如何激发事件的。在控件和容器之间实际上是一个共生关系，控件只能在容器有了处理事件的函数后才可以传送事件。显然，容器也将承担一些责任，下面就让我们看看都是些什么责任。

MFC 如何处理事件

回忆一下，控件事件的分发接口实际上是在 COleControlSite 类中，这个类把给事件的 IDispatch 接口称为事件接收器，而事件接收器仅仅是 IDispatch 接口以 MFC 方式实现的一个版本。也就是说，COleControlSite 使用了一个嵌套的类/接口映射表来实现，这意味着 COleControlSite 在嵌套类中保持了一个 IDispatch 的 vtable，该嵌套类叫做 COleControlSite::XEventSink，而且 COleControlSite 有一个事件接收器的成员变量 m_xEventSink。

当一个 OLE 控件容器创建了一个 OLE 控件，容器将向控件请求它的 IConnectionPointContainer 接口。如果 OLE 控件容器得到了有效的 IConnectionPointContainer 接口，容器就知道控件支持连接。接下来，容器向控件请求它的基本事件集的 GUID（使用控件实现的另一个接口 IProvideClassInfo2）。一旦容器有了事件接口的 GUID，控件就能得到代表事件的 IConnectionPoint 接口。而一旦控件有了这个接口，容器就能使用 IConnectionPoint::Advise()注册它的事件处理函数（事件接收器——容器事件的分发接口——COleControlSite::m_xEventSink）。

记住，事件接收器事实上仅仅是 IDispatch 的另一个版本。MFC 实际上通过一些宏来实现事件接收器，这些宏产生了一个事件接收映射表（难以置信吧？……越来越多的映射表和宏）。

任何继承自 CCmdTarget 的类都可以有一个事件接收器——因为每个 CCmdTarget 都有一个消息映射表。一个 CCmdTarget 的事件接收映射表把一个事件集的分发成员和一个继承自 CCmdTarget 的类的成员函数联系了起来。

现在让我们通过容器的事件接收器来跟踪事件。

无论何时，一旦一个 OLE 控件想调用一个事件，该控件就会调用 IDispatch::Invoke()函数（该函数由容器实现）。如果容器是用 MFC 写的，这个调用将在 COleControlSite::m_xEventSink::Invoke()函数中结束。MFC 的事件接收器 m_xEventSink::Invoke()将调用 COleControlSite::OnEvent()，同时也将调用控件父窗口的 OnCmdMsg()，而 OnCmdMsg()调用了 COccManager::OnEvent()（这是一个管理 OLE 控件的没有文档说明的 MFC 类）。最后将在 CCmdTarget::OnEvent()函数中结束。

MFC 实现了一个查询机制，可以查询和 OLE 控件事件相对应的处理该事件的 C++函数。一个 CCmdTarget 的事件接收映射表像 MFC 中其它的查询机制一样，包含了一套宏（DECLARE_EVENTSINK_MAP 和 BEGIN_EVENTSINK_MAP/END_EVENTSINK_MAP）来建立映射表。这个事件接收映射表可以概括为一个事件接收器入口结构的数组。

```
struct AFX_EVENTSINKMAP_ENTRY
{
   AFX_DISPMAP_ENTRY dispEntry;
   UINT nCtrlIDFirst;
   UINT nCtrlIDLast;
};
```

请注意，AFX_EVENTSINKMAP_ENTRY 包含了一个 AFX_DISPMAP_ENTRY，这是一个线索，它说明了 MFC 实际上把事件看做了另一个自动化方法。而 AFX_DISPMAP_ENTRY 代表了激发事件后调用的方法。

CCmdTarget::OnEvent()函数使用了 CCmdTarget::GetEventSinkEntry()函数来查看对象的事件接收映射表，以找到需要的事件映射表入口。如果 OnEvent()函数找到了一个事件接收器入口，OnEvent()就将参数打包（记住，这些参数依然是一个 VARIANT 的数组），并且使用 CCmdTarget::CallMemberFunction()来调用事件。

以上这些都是相当精心的设计，它无疑工作得非常好。

技巧：在一个视图中加入一个事件接收器

在一个非对话框的窗口内加入你自己的事件接收器并不是非常难——你只需要做好那些 AppWizard 和 ClassWizard 做的工作。为了响应 OLE 控件的事件，你必须做两件事：（1）写一个事件响应函数；（2）建立一个事件接收器，这两个任务都不难——你只需要做一些额外的输入工作。

加入一个事件处理函数

记住 OLE 控件事件是如何工作的：一旦控件被创建，控件就要求控件容器实现处理事件的函数。毕竟，OLE 控件事件就是一个调用主机内函数的 OLE 控件。因此首先需要控件容器实现该函数，所以在本例中，我们需要在视里实现该函数。如果你知道了事件处理函数如何声明，这个工作就变得相当容易了。

了解函数签名

当你使用对话框编辑器在你的对话框中加入一个 OLE 控件并且写出事件处理函数后，对话框编辑器会自动查找控件事件的类型信息。ClassWizard 会用正确的签名（signature）插入一个函数。但是如果你不用 ClassWizard，就需要自己指出函数签名。

有两种方法来指出事件处理函数的签名。比较快的方法就是建立一个对话框，并且在对话框中加入控件。然后使用 ClassWizard 加入事件处理函数。ClassWizard 会使用正确的签名在你的对话框中加入成员函数。你可以通过查看 ClassWizard 产生的函数来找到添加到视图里的函数的签名。当然，这种方法在你的程序里加入了一个（可能是无用的）对话框，使代

码和资源都变得更庞大了。

如果你不喜欢在你的程序里加入一个无用的对话框，则可以使用一个可以解析件类型库的浏览器。OLE2VW32.EXE（由 Charlie Kindel 编写）就是一个非常好的浏览器。Visual C++ 中带有该程序。

OLE2VW32.EXE 在注册表里查找系统中所有的 OLE 对象。到目前为止，很多这些对象都是 OLE 控件。你只需要高亮一个 OLE 对象，该程序就会在窗口右下角显示这个对象的所有接口。

双击链表中的 IDispatch 接口，将弹出一个显示对象分发接口的新对话框。OLE 控件事件就是通过分发接口产生的。

接下来，在你的头文件中声明函数。例如：如果你想在视图里处理 CurrentPriceChanged 事件，只需要在视图里加入下面的函数：

```
class CTestctlsView : public CView
{
...
   afx_msg void OnCurrentPriceChanged(double dOldPrice,
                                      double dNewPrice);
...
};
```

然后在你的视图里实现该事件处理函数：

```
void CTestctlsView::OnCurrentPriceChanged(double dOldPrice,
                                          double dNewPrice) {
   // Respond to the current price changing...
}
```

当这些都做好后，你需要为你的视图创建一个事件接收器，然后把事件处理函数加入到事件接收器中。

建立事件接收映射表

MFC 使用事件接收映射表（event sink map）来实现事件接收。像其他的 MFC 查询机制一样，事件接收映射表也是使用宏来建立的，即 DECLARE_EVENTSINK_MAP、BEGIN_EVENTSINK_MAP 和 END_EVENTSINK_MAP。

可以在你的视图的头文件中使用 DECLARE_EVENTSINK_MAP 宏。例如：如果你的视图的名字是 CTestctlsView，则可以在类里做如下定义：

```
class CTestctlsView : public CView {
...
// Some private methods an attributes
...
public:
...
// Some public methods an attributes
...
protected:
   DECLARE_EVENTSINK_MAP()
};
```

接下来在你的视图的CPP文件里使用BEGIN_EVENTSINK_MAP和END_EVENTSINK_MAP宏。

例如：你可以在你的视图的CPP文件里加入以下内容：

```
BEGIN_EVENTSINK_MAP(CTestctlsView, CView)
// The event sink entries will go here.
END_EVENTSINK_MAP()
```

最后，你需要在你的事件映射表里加入事件处理函数。当然，也有相应的宏来完成这项工作，即ON_EVENT()宏。这个宏有5个参数：（1）事件接收映射表所在的类；（2）控件的ID号；（3）事件的分发ID；（4）真正的事件处理函数；（5）事件处理函数的签名。

那么这些参数都是从哪里得到的呢？前两个参数很容易。你已经知道了视图的名字和你需要的控件的ID。如果你生成一个额外的对话框（如前文所述），你就能根据ClassWizard自动加入的ON_EVENT宏得到分发ID。当然你也可以用OLE2VW32得到它。它就在程序右下角的窗口中显示（FUNCDESC标题下的那个），只要看看memid域就可以知道了。接下来的事件处理函数就是你要定义的函数。最后，签名是一系列的VTS_常量，这些常量表示了各种自动化数据类型。你可以在MFC的AFXDISP.H文件中找到这些常量的定义。

例如：为了把OnCurrentPriceChanged()函数加入到你的事件接收器中，应该在你的事件接收映射表里加入以下内容：

```
ON_EVENT(CTestctlsView, ID_STOCKSCTL, 1,
        OnCurrentPriceChanged, VTS_R8 VTS_R8)
```

这个事件属于CTestctlsView类，ID是ID_STOCKSCTL所表示的数，分发ID为1，使用OnCurrentPriceChanged()函数作为事件处理函数，并且参数表中包含了两个double类型的数。

只要控件发出了current-price-changed事件，视图就可以通过它的事件接收映射表来获取事件，并且调用CTestctlsView::OnCurrentPriceChanged()函数来处理。你可以实现该函数以处理新获得的数据，例如你可能想重新生成一张股票价格图。

在结束介绍OLE控件前，让我们再来看一看OLE控件另一个有趣的方面：属性页。

OLE控件的属性页

因为OLE控件是一个自动化对象，所有它可以通过IDispatch展示其属性。这意味着它的客户可以很容易地访问控件的属性。除了用这个通用方式展示属性外，OLE控件还实现了一些带有标签的对话框，用户可以通过这些对话框来改变它的属性。设计一个OLE控件的属性页是很容易的——尤其是当使用Visual C++的ClassWizard时。然而，如果你只是跟着微软的指示走，你将错过许多OLE控件的精彩之处，也会错过目睹OLE控件综合技术的机会。

让我们仔细看一看OLE控件的属性页，看看在这后面究竟发生了什么。

属性页综述

OLE控件实现了属性页，这使得客户可以轻松地修改控件的属性，而不需要自己实现任

何用户接口代码。例如：如果没有属性页，任何人想要修改你的控件的属性，都不得不花费大量的时间来写一个对话框来实现对控件属性的修改。这主要是由于 OLE 控件是一个自包含的软件包。为什么一个 OLE 控件的客户不得不自己实现许多用户接口代码来操纵控件属性？因为控件自己知道它所有的属性，所以它提供了一些可以操作属性的用户接口是很有意义的。OLE 控件是通过一个 verb 来提供这些用户接口的。

因为 OLE 控件是一个 OLE 文档内容对象，所以它们能够实现 verb。而且 OLE 控件也实现了几个 verb，其中一个是属性 verb。OLE 控件对属性 verb 的反应就是显示一个可以操纵控件各种属性的属性对话框。图 15-2 是一个微软的 grid 控件（由 Visual C++提供，插入在测试容器里）。这张图显示了微软的 grid 控件如何通过显示带有标签的对话框来响应属性 verb。

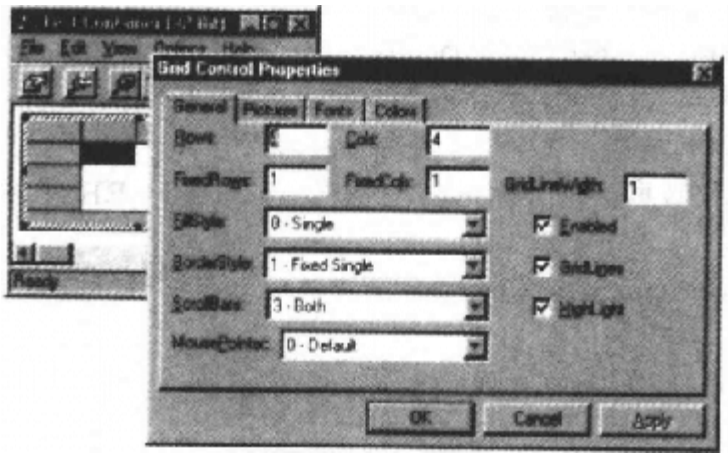


图 15-2 微软的 grid 控件的属性

在分析属性前，让我们先看一看图 15-2，该图总共显示了 6 个 OLE 对象。首先，OLE 控件容器（即测试容器）实现了一些 OLE 接口，所以它是一个 OLE 对象。其次，grid 控件本身是 OLE 对象。最后，请注意对话框中的 4 个标签，每一个都是一个单独的 OLE 对象。这是 OLE 综合技术的一个完美体现。

注意一下测试容器、grid 控件和属性对话框是如何结合成一个独立单元的。如果你以前没有接触过，你可能会认为它们都在同一个 EXE 文件中（正如在出现 OLE 以前的年代里那样）。但是实际情况并不是这样！测试容器是一个单独的 EXE 文件，而 grid 控件当然是在一个 DLL 文件中。此外对话框的标签（例如“General”标签）也是在一个控件 DLL 文件中。最后，在其它 3 个标签（即“Picture”、“Font”和“Color”）后面的那个对象是来源于 MFC40.DLL，这个文件是由 Visual C++提供的。光滑地、无缝地、标准地综合了这 3 个独立的二进制文件，这正是 OLE 技术的主要目标之一。现在再让我们看看它是如何工作的。

设计属性页

先让我们看看 Visual C++是如何建立这种机制的。在 Visual C++中使用 OLE 控件属性页是非常容易的，只需要 3 个步骤：（1）定义 OLE 控件属性；（2）为属性页建立一个对话框模板；（3）在对话框和控件之间链接属性。下面分别介绍这 3 个步骤。

定义 OLE 控件属性

第一步是在 OLE 控件内定义属性。如果你正在使用 Visual C++，这一步没有任何问题，只需要用 ClassWizard 加入自动化属性。首先进入“OLE Automation”标签对话框，然后按下“Properties”按钮加入属性。记住属性的 External 名是很重要的，因为这对于实现属性页很重要。例如：如果你正在写一个监控实时数据源的 OLE 控件，你可能想在数据变化时让控件发出声音。进一步，你可以想要提供一个可以打开和关闭声音的方法。而实现这个最直接的方法就是在控件里增加一个叫做“SOUND”的属性。

创建对话框模板

第二步是为属性页创建对话框模板，每一个属性页实际上都是一个对话框。事实上，ControlWizard 总是在你创建控件时生成一个空的默认属性页。你可以直接使用这个属性页或者自己建立一个新的，只需要给对话框加入一个可以操纵 OLE 控件属性的控件。例如上文中的“SOUND”属性，你可以在对话框中加入一个复选框（checkbox），这就很好了。

在对话框控件和 OLE 控件属性间建立链接

现在在对话框中已经有一个复选框了，你需要将复选框的状态（选中或未选中）和 OLE 控件属性连接起来。ClassWizard 再一次使这个工作变得很容易。先用 ClassWizard 在对话框中加入一个代表复选框的成员变量。你可以单击“Member Variables”标签并且按下“Add Variable”按钮来完成这些。然后 ClassWizard 会弹出一个对话框来询问你关于这个变量的一些信息，如图 15-3 所示。

这样我们就加入了一个代表你的属性页对话框中复选框的成员变量。另外，ClassWizard 还建立了 MFC 的动态数据交换机制（DDX/DDV）。该机制将复选框的状态（选中或未选中）和成员变量（m_bSound）联系起来。

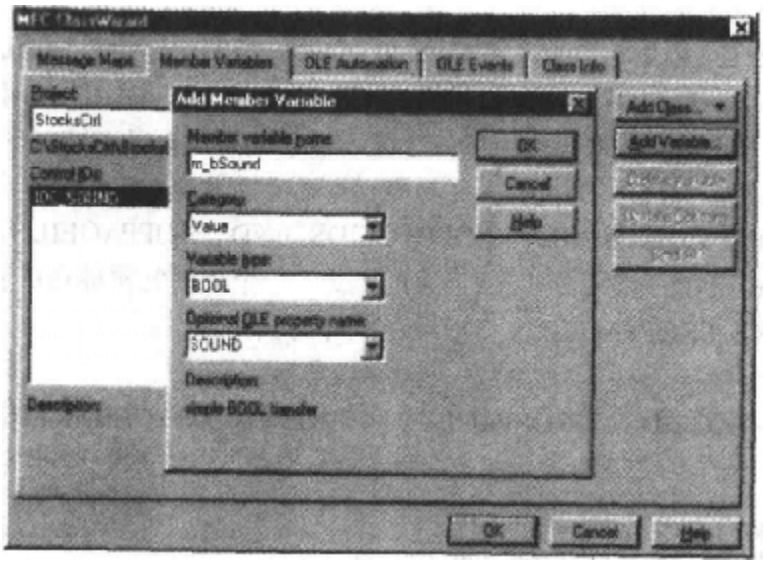


图 15-3 给属性页对话框添加一个成员变量

请注意上图中的“Optional OLE property name”域，该域显示了一个“SOUND”字符

串。你应该还记得 OLE 控件的声音属性（一个规则的自动化属性）的名字就是“SOUND”。如果你认为 OLE 控件的声音属性和属性页中的复选框状态是通过自动化链接在一起的，那么你猜对了。要记住控件声音属性的 External 名就是“SOUND”。在实际效果上，属性页变成了控件的一个自动化客户，它继承了控件的 IDispatch 接口。下面介绍它是如何工作的。

属性 verb

当一个 OLE 控件被嵌入某种容器内后，容器通常拥有一些调用控件的 verb 的方法。例如：如果你在微软公司的测试容器内加了一个控件，你可以把你的鼠标移到控件的边界上并单击鼠标右键以弹出 verb 菜单。如果控件支持属性页，你将在菜单里看见属性 verb。选择这个属性 verb 可以告诉定点对象激活它的属性 verb。（OLE 控件实际上是一个展示 IOleObject 接口的嵌入对象，容器可以使用这个接口来调用 verb，即调用 IOleObject::DoVerb()。）

OLE 控件是一个 OLE 文档内容对象，因此它实现了 IOleObject 接口，客户可以使用这个接口来调用 verb。也就是说，客户可以调用控件的 IOleObject::DoVerb()函数。在控件内部，框架将容器的 IOleObject::DoVerb()函数调用路由到 COleControl::OnDoVerb()函数，然后在 COleControl::OnProperties()函数中结束。

COleControl::OnProperties()函数按以下方式工作：控件属性是通过 IDispatch 链接的，每一个属性页都是一个独立的 OLE 对象，并且可以通过控件的主 IDispatch 指针与 OLE 控件交流信息。这意味着控件必须把它的 IDispatch 指针交给属性页，使得属性页可以通过它操纵控件属性。COleControl::OnProperties()可以获得 OLE 控件的主 IDispatch 指针（在本例中，可以通过这个指针来访问控件的声音属性）。因为 COleControl 最终是从 CCmdTarget 继承而来的，所以控件可以用 CCmdTarget::GetIDispatch()函数来交出它的 IDispatch 指针。而 OLE 控件也将用这个分发指针建立属性页。

因为每一个属性页都是一个独立的 OLE 对象，所以它们可以通过普通的 OLE 方法创建，即调用 CoCreateInstance()函数。这个函数使用一个 OLE 对象的 GUID 去创建 OLE 对象。当然，这意味着一个 OLE 控件需要知道每一个需要显示的属性页的 GUID。

为了保持传统，MFC 仍然使用了宏来管理属性页的 GUID 链表。这些宏就是 DECLARE_PROPPAGEIDS、BEGIN_PROPPAGEIDS、END_PROPPAGEIDS 和 PROPPAGEID，你可以在一些 OLE 控件的源代码里看到它们。例如：一个控件的属性页链表可能会类似于如下形式。DECLARE_PROPPAGEIDS 将出现在控件的头文件中：

```
DECLARE_PROPPAGEIDS(CStocksCtrl)
```

BEGIN_PROPPAGEIDS、PROPPAGEID 和 END_PROPPAGEIDS 将出现在控件的 CPP 文件中：

```
BEGIN_PROPPAGEIDS(CStocksCtrl, 1)
   PROPPAGEID(CStocksCtrlPropPage::guid)
END_PROPPAGEIDS(CStocksCtrl)
```

这些宏仅仅是简单地扩展产生一个 GUID 静态数组。DECLARE_PROPPAGEIDS 声明了一个 GUID 数组（代表属性页）和一个成员函数，这样程序可以在需要的时候获得 GUID 链

表。BEGIN_PROPPAGEIDS、PROPPAGEID 和 END_PROPPAGEIDS 扩展后实现了这个 GUID 数组。

在获得 OLE 控件的 IDispatch 数组后，COleControl::OnProperties()调用 OLE API 函数 OleCreatePropertyFrame()来创建保存属性页的对话框。COleControl::OnProperties()函数还使用了 OLE 控件的 IDispatch 指针和 GUID 链表来创建对话框框架。而 OleCreatePropertyFrame()函数则创建了一个模态对话框来存放带有标签的属性页。如果你调用它，OLE 就会接收并创建属性页。基本上 OleCreatePropertyFrame()函数仅仅是为链表中的每一个 GUID 调用 CoCreateInstance()函数，按顺序创建每一个属性页。

为了介绍属性页余下的内容，让我们从另一方面看一看 MFC 是如何实现 OLE 属性页的。这个类就是 COlePropertyPage。

COlePropertyPage

在 MFC 中，COlePropertyPage 类是用来处理 OLE 属性页的类。它从 CDialog 派生，因此它可以完成 CDialog 所能做的所有工作。除了标准的对话框支持外，COlePropertyPage 类还实现了一个叫做 IPropertyPage2 的接口。当你显示 OLE 控件的属性页时，这些带有标签的对话框都是放在同一个对话框框架内的。这个对话框框架使用每一个属性页的 IPropertyPage2 接口和属性页交流信息。

在属性页的客户（在本例中，就是 OLE 控件）调用了 OleCreatePropertyPage 后，OLE 将开始为每一个控件管理的属性页调用 CoCreateInstance()函数。在这个函数成功执行后，控制将移交给 COlePropertyPage 的构造函数。在构造函数执行完毕后，OleCreatePropertyFrame()函数将调用 QueryInterface()来获得属性页的 IPropertyPage2 接口。一旦 OLE 有了这个接口，OLE 将使用它来建立属性页。

记住，OLE 控件实现了 IDispatch（这样它可以展示它的属性），因此任何想访问 OLE 控件属性的客户都必须拥有 OLE 控件的 IDispatch 属性的拷贝。换句话说，OLE 属性页需要 OLE 控件的 IDispatch 指针。下面介绍一下属性页是在哪里得到控件的 IDispatch 指针的：首先 IPropertyPage2 接口有一个 SetObjects()函数，这个函数接受一个 IUnknown 指针的数组。另外在一个基于 MFC 的标准 OLE 控件中，OleCreatePropertyFrame()函数将控件的主 IDispatch 指针传送给了属性页，属性页把 IDispatch 指针保存起来以待后用（例如：当属性页想要访问 OLE 控件属性的时候）。

注意到属性框架和 IPropertyPage2::SetObjects()函数都处理 IUnknown 指针，因此这里有许多可扩展性。虽然 IDispatch 是控件处理属性的最合乎逻辑的地方，但是对你使用 IDispatch 并没有任何限制，那些 IUnknown 指针可以是你自己定义的任何接口。虽然 MFC 使用 IDispatch，可是并没有任何理由让你不能写出自己的属性页类。然而，必须注意到控件和属性页必须在使用何种类型的接口上达成一致。

访问属性

一旦 OLE 属性页有了 OLE 控件的 IDispatch 指针，属性页就能很容易地访问控件属性。因为属性页拥有 OLE 控件主 IDispatch 指针的一个拷贝，所以实现这一点就没有任何问题。MFC 还扩展了 DDX 机制来处理 OLE 控件属性页。

使用前面的例子，默认的属性页是在 DoDataExchange()函数中实现这些的。下面是向导

自动加入的代码：

```
void CStocksCtrlPropPage::DoDataExchange(CDataExchange* pDX)
{
   //{{AFX_DATA_MAP(CStocksCtrlPropPage)
   DDP_Check(pDX, IDC_SOUND, m_bSound, _T("Sound") );
   DDX_Check(pDX, IDC_SOUND, m_bSound);
   //}}AFX_DATA_MAP
   DDP_PostProcessing(pDX);
}
```

在注释之间的两行代码提供了数据交换：（1）在对话框间；（2）在控件和对话框的成员变量间。MFC 提供了许多属性交换函数，表 15-1 列出了 MFC 所有的属性页交换函数：

表 15-1 MFC 的属性交换函数

函数名	将这种类型的对话框控件链接到一个属性
DDP_CBIndex:	在组合框中选择的字符串的索引
DDP_CBString:	在组合框中选择的字符串（非精确匹配）
DDP_CBStringExact:	在组合框中选择的字符串（精确匹配）
DDP_Check:	一个复选框
DDP_LBIndex:	在列表框中选择的字符串的索引
DDP_LBString:	在列表框中选择的字符串（非精确匹配）
DDP_LBStringExact:	在链表框中选择的字符串（精确匹配）
DDP_Radio:	单选按钮
DDP_Text:	文本

COlePropertyPage 实现了几个成员函数以完成一些实际工作：GetPropText()/SetPropText()、GetPropCheck()/SetPropCheck()、GetPropRadio()/SetPropRadion()和 GetPropIndex()/SetPropIndex()。除了它们交换的数据类型不同外，这些函数多少都是以相同方式工作的。作为一个例子，接下来看一看一个 COlePropertyPage 类是如何在一个复选框和自动化属性间交换数据的。

GetPropCheck()函数

在从 OLE 控件的属性页中获得数据后，GetPropCheck()函数仅仅是使用了控件的主 IDispatch 指针来设置控件的属性。COlePropertyPage 保存了一个 IDispatch 指针的数组，这个数组是在属性页创建时（也就是在 OLE 控件调用 OleCreatePropertyFrame()函数，而该函数调用 IPropertyPage2::SetObject()时）被赋值的。首先，GetPropCheck()函数在栈里声明了一个 COleDispatchDriver 对象。这个对象是 MFC 对 IDispatch 的封装。

接着 GetPropCheck()函数检查数组中每一个想要访问自动化属性的 IDispatch 指针，它是通过调用 IDispatch::GetIDsOfNames()函数来获得属性的 DISPID。一旦获得了 DISPID，它就会把 IDispatch 指针和栈里的 COleDispatchDriver 对象联系起来，并且调用 COleDispatchDriver::GetProperty()函数来获得控件的自动化属性。

SetPropCheck()函数几乎是以相同的方式工作的，只不过函数是设置 OLE 控件的属性而不是读属性。

属性页总结

属性页是用户访问 OLE 对象各种属性的快捷方法，它是 OLE 控件的一个标准，并且在你用 ControlWizard 和 MFC 创建控件时自动生成。

OLE 控件通过 IDispatch 展示它的属性。无论何时一个 OLE 控件的客户想要使用控件的属性页，它只需调用属性 verb。作为一个结果，OLE 控件使用 OLE API 函数 OleCreatePropertyPage()来创建一个对话框，这个对话框保存了一系列代表不同控件属性的标签式对话框。OLE 控件在调用 OleCreatePropertyPage()函数时传送它的主 IDispatch 指针，而这个函数把 IDispatch 指针传送给属性页。每一个属性页保存了 OLE 控件的 IDispatch 指针并且使用这个指针访问 OLE 控件的属性。

属性页无疑是 MFC 无缝连接 EXE 和 DLL 模块的一个精彩例子。

结　语

现在你已经知道了如何使用 OLE 控件了。这种控件可能是这些日子里 OLE 最激动人心的领域。使用 OLE 控件无疑是你定义自己的接口的最快捷的方式之一。另外，微软公司正在努力把 OLE 控件结合到他们的 Internet 领域中去。

OLE 控件几乎实现了所有成熟的 OLE 技术：

- 自动化技术。
- 定点激活技术。
- 持续性技术。
- 连接技术。
- 属性页技术。

通过实现上述这些技术，OLE 控件几乎能够应用在各种地方。你能在你的 Visual Basic 程序中使用它们，也可以自己创建自己的 OLE 控件容器。

虽然大多数的特性都要求完成大量的工作来使控件运转，可是无论谁要写一个 OLE 控件，大量的工作已经在模板文件里完成了。在 OLE 控件领域，MFC 无疑是极其有用的，它实现了模板文件并引导你完成你自己的工作。

如果留心，你会看见 OLE 控件将被应用在大量的程序里，这项技术正在一步步地走向成熟。

APPENDIX

A MFC 源代码导读

有时看来，研究 MFC 的 150000 行代码是一件不可思议的事情。为了帮助你穿越这片 MFC 的丛林，我们写了这个代码导读。导读会教会你如何探索 MFC，而且当你看到成堆的 MFC 代码放在面前而不知所措时，它会做一个参考。

代码导读从回顾一些通用的 MFC 编码风格开始。关于编码风格方面的知识可以帮助你理解独立模块的功能，而不再需要查看其他文件的声明。这些编码风格也可以作为你写 MFC 程序的规范。然后，我们会给出一些提示，以便你能更容易地理解 MFC 代码。我们还会给出一些可以利用的工具。

在说明了 MFC 的编码风格和一些提示之后，我们会对 MFC 源代码的以下几个方面进一步地讨论：

- MFC 目录结构。
- MFC 头文件。
- MFC 内嵌文件（inline file）。
- MFC 资源（resource）。
- MFC 源文件。

MFC 编码技术

在这部分，我们会回顾一些微软在写 MFC 代码时所使用的编码技术。微软编码风格方面的知识可以帮助你加快学习 MFC 内幕的速度，还可以帮助你写出更好的 MFC 类（如果你打算利用我们推荐的微软的风格）。

类的声明部分

不用看很多头文件，你就会意识到 MFC 小组遵循了一个很严谨的风格。类声明围绕着关键字注释块组织在一起。MFC 小组用这些注释将类声明分成若干个部分，顺序如下：

- //Constructors。
- //Attributes。
- //Operations。
- //Overrideables。
- //Implementation。

下面是每个子部分的详细解释，以及你可能在相应的子部分中找到的内容的例子：

- //Constructors——显然，C++的构造函数（Constructor）在这个子部分里，但还不是十分明显，MFC 有时还会将一些其他负责初始化的成员函数放在这个子部分，例如 Create()。
- //Attributes——这里记录了成员数据，以及一些对成员数据进行操作（设置和读取操作）的成员函数。这些成员函数有时称为访问函数（accessor function）。
- //Operations——记录了 MFC 的使用者（或 MFC 类）可以直接调用的方法。有些情况下，操作很多，微软会将这个部分再分成若干子部分（例如，//Operations——Grid functions，//Operations——Graphing functions 等等）。
- //Overrideables——派生类要覆盖的函数都在这个部分（又叫做虚函数）。
- //Implementation——我们最感兴趣的部分。在 MFC 中，这一部分的所有内容都是实现细节，也就是一些没有文档说明的东西。如果你想知道一个类如何运作，就要从这部分下手。

变量命名——匈牙利命名

如果你从前学习 Windows 编程时读过 Petzold 的书，你应该对匈牙利命名很熟悉了。简而言之，它是这样一种命名规则：在变量名的前面加上变量的类型。

在前缀后面，下一个字符要大写，任何字的首字符也都大写。表 A-1 给出了对于基本的 C++类型变量的前缀应该是什么，还有一些例子。

表 A-1 匈牙利命名

类型	前缀	例子	注释
char	c	cDirSeparator	
BOOL	b	blsSending	
int	n	nVariableCount	
UINT	n	nMyUnsignedInt	
WORD	w	wListID	
LONG	l	lAxisRatio	
DWORD	dw	dwPackedmessage	
*(pointer)	p	pWnd	
FAR*	lp	lpWnd	
LPSTR	lpsz	lpszFileName	z 表示以 NULL 结尾
handle	h	hWnd	
callback	lpfn	lpfnHookProc	指向一个函数

注意：因为一些开发人员严格地遵守匈牙利命名规则，所以如果你想快速解读 MFC 内幕，就应该能熟练地识别出匈牙利命名。

MFC 在匈牙利命名规则的基础上又增加了一些内容。首先，微软在每个类前面增加了前

缀“C”，在每个成员变量前面增加了“m_”，例如 CObject 和 m_hWnd。而且，对于一些更通用的 MFC 来说，还有一些标准的前缀，如表 A-2 所示。

表 A-2 MFC 对匈牙利命名规则的扩展

类	前缀	例子
CRect	rect	rectScroll
CPoint	pt	ptMouseClick
CSize	sz	szRectangle
CString	str	strFind
CWnd	Wnd	WndControl
CWnd*	pWnd	pWndDialog

贯穿本书，你可以看到很多使用匈牙利命名规则命名的变量。当你在 MFC 源代码中看到一个变量以“m_”开头时，你应该立刻反应出这是一个类成员变量，而不是一个局部变量或函数的参数。

符号命名规则

关于资源符号的命名，MFC 有一系列严格的规定。知道它们使用的前缀和覆盖的范围可以帮助你解码不同的符号。表 A-3 给出了符号命名规则。

Tech Note 35 给出了关于资源范围更多的细节，如果你想知道关于符号范围的细节，最好先阅读一下它。

表 A-3 MFC 符号命名规则

类型	前缀	例子	范围
Shared by multiple resources	IDR_	IDR_MAINFRAME	1-0x6FFF
Dialog resource	IDD_	IDD_ABOUT	1-0x6FFF
Dialog resource help context ID (for context-sensitive help	HIDD_	HIDD_HELP_ABOUT	0x20001-0x26FF
Bitmap resource	IDB_	IDB_SMILEY	1-0x6FFF
Cursor resource Icon resource	IDC_	IDC_HAND	1-0x6FFF
Menu or toolbar command	ID_	ID_CIRCLE_TOOL	0x8000-0xDFFF
Command help context	HID_	HID_CIRCLE_TOOL	0x1800-0x1DFFF
Message box prompt	IDP_	IDP_FATALERROR	8-0xDFFF
Message box help context	HIDP_	HIDP_FATALERROR	0x30008-0x3DFFF
String resource	IDS_	IDS_ERROR12	1-0x7FFF
Control in dialog template	IDC_	IDC_COMBO1	8-0xDFFF

简单的验证

通过理解微软对 MFC 类声明、变量命名和符号命名的规则，就可以在阅读大量的代码时，在不用研究很深的情况下迅速指出一些关键点。为了说明这一点，我们从 MFC 的源代码中摘抄出了一小段代码：

```
if (bIdleStatus && m_bStatusSet) {
    m_bStatusSet = FALSE;
    GetOwner()->SendMessage(WM_SETMESSAGESTRING, AFX_IDS_IDLEMESSAGE);
}
if (bResetTimer)
    KillTimer(ID_TIMER); m_bDelayDone = FALSE;
if (m_pToolTip != NULL)
    m_pToolTip->DestroyWindow(); delete m_pToolTip; m_pToolTip = NULL;
```

看一下这段代码，你会得出下面一些结论：

- bIdleStatus——或者是一个局部变量，或者是一个参数，是 BOOL 类型的。
- m_bStatusSet——一个类成员数据，是 BOOL 类型的。
- AFX_IDS_IDLEMESS——一个字符串资源。
- bResetTimer——或者是一个局部变量，或者是一个参数，是 BOOL 类型的。
- m_bDelayDone——一个类成员数据，是 BOOL 类型的。
- m_pToolTip——一个类成员数据，是指针类型的（可能指向一个工具提示类）。

很令人惊讶，不是吗？匈牙利命名的知识和 MFC 的一些其他规则可以告诉你代码中的信息，而不需要去看类的声明或变量声明。当你浏览一块代码时，看看你解读这些变量的速度。在读过几千行 MFC 源代码之后，这些反应就变成本能了。

通用的 MFC #defines

知道一些 MFC 代码中使用的通用#defines 可以帮助你理解代码块的功能。

下面是关于通用符号的一些信息：

- _MAC——当你查看 MFC 的源代码时，你会记起微软在 Macintosh 上也支持 MFC（通过 Visual C++的跨平台的针对 Macintosh 的版本来实现）。在两个平台上只有一份 MFC 源代码，所以很多针对 Macintosh 的代码都用_MAC ifdef 标注。
- _AFX_NO_OCC_SUPPORT——Macintosh 上不支持 OLE 控件，所以用_AFX_NO_OCC_SUPPORT 标志来关闭它们。如果你要探索 MFC 中的 OLE 控件，就会在 OLE 控件的代码中发现很多这个标志。
- _DEBUG——这个标志在调试版本（debug build）中才会设置，在发布版本（release build）中会关闭。
- _AFXDLL——用来区分 DLL 代码和 LIB 代码（见第 10 章）。
- _UNICODE——表示在 MFC 编译时是否增加了对 Unicode 的支持。

粒度和交换调整

在一些 MFC 源代码文件中，你会发现 IMPLEMENT_DYNAMIC 调用和文件的内容不匹

配。一个例子就是在 VIEWCORE.CPP 中，VIEWCORE.CPP 包含了 CView 的大部分实现代码，但是在文件的最后你会发现下面这样的代码：

```
// IMPLEMENT_DYNAMIC for CView is in wincore.cpp for .OBJ granularity
//   reasons
IMPLEMENT_DYNAMIC(CSplitterWnd, CWnd)   // for swap tuning

IMPLEMENT_DYNAMIC(CCtrlView, CView)
```

在查看了头文件之后，我们询问 MFC 内幕的权威人士——DeanMcCrory 院士，这段代码为什么是这样的。下面是他给出的答案：

> 这种情况下的交换调整（swap tuning）术语有些用词不当。我们用微软的一个内部工具来交换调整，这个工具在 DLL 中组织页面（内存中的页面），从而使 MFC 的工作集（working set）更小（这些是在 DLL 链接完成之后）。
>
> IMPLEMENT_宏出现在看似随机的文件中的原因是“库粒度”（library granularity）。也就是在注释中的.OBJ 粒度。在静态链接时，我们需要 MFC 成为一个你所希望（pay as you go）的系统。你用的 MFC 特性越多，链接进来的 MFC 代码就越多。你用的越少，链接进来的 MFC 代码就越少。
>
> 大多数情况下，链接器和编译器合作完成这些。我们用/Gy 参数编译，再用/opt:noref 链接。/Gy 参数告诉编译器将每个函数放到自己的 comdat 中（可以看作是将每个函数放入到对链接器来说是独立的.OBJ 文件的方法）。/opt:noref 告诉编译器削减那些未引用的 comdat。这样做通常都会达到 pay as you go 这个目标——只有引用的函数才会最终出现可执行文件中。当你有指向函数的 non-combat 部分时，就会有问题。这样的 non-combat 部分包括类似消息表（包含了消息处理函数的地址）的数据和类似 CRuntimeClass 对象的数据（它包含 CreateObject 函数的地址。CreateObject 函数调用了构造函数，构造函数会指向一个 vtable。这个 vtable 就是指向每个虚函数的数据，通过这些数据就可能会找到一些没有用到的函数）。后面的问题是唯一对 DECLARE_SERIAL 和 DECLARE_DYNCREATE 有影响的。你可能发现你给出的例子中是 DECLARE_DYNAMIC。那么我们要解决的问题是什么呢？
>
> 有可能我们引用一个特定对象的 CRuntimeClass——即使只是 DECLARE_DYNAMIC 的 CRuntimeClass 对象。如果这样的 CRuntimeClass 对象的定义和这个类的其他代码放入了同一个文件，你可能也在相同的.OBJ 文件中放入了相同的数据。如果数据（可能是消息映射表）指向了这个类中的函数，你最终就将所有这样的函数都放入了.OBJ 文件，即使这些函数没有被引用。它们只会被那些不被引用的数据引用。但是因为它们不在一个 comdat 中，而且 non-comdat 部分不能削减，所以链接器就会阻塞。结果就是产生了很多你不需要的代码。
>
> 以 CView 为例，IMPLEMENT_DYNAMIC(CView)在文件 WINFRM.CPP 中，这是因为 CFrameWnd 需要引用 RUNTIME_CLASS(CView)。因此所有使用了 CFrameWnd 的内容都需要 CRuntimeClass 对象。如果我们把 CView 的 CRuntimeClass 类的定义放在 VIEWCORE.CPP 中（你可能希望在这里找到 CRuntimeClass），那么使用 CFrameWnd 的应用程序就会变得比需要的大。这样就使得小的应用程序和 MFC 1.0 风格的应用程序变得更大（例如，samples\mfc\hello\hello.exe 在静态链接情况下会变得更大）。
>
> 我们确定定到哪里该做什么的方法就是取定几个场景（例如，helloapp 例子和 AppWizard 生成的缺省应用程序），在链接后检查 MAP 文件确定是否有不必要的内容加入进来。你无法做到 100%的满意，但是做这样小的改变对粒度的改进也是惊人的。

探索 MFC 的工具

即使你对匈牙利命名规则很熟悉，你可能有时也需要查看 MFC 源代码。下面是我们已经发现的一些工具，这些工具可以帮助你探索 MFC。

Visual C++ Browser

Visual C++ Browser 在帮助你探索 MFC 源代码方面做的很好。这个浏览器的大部分内容在文件 MFC.BSC 中，这个文件在你的 Visual C++光盘的\msdev\mfc\src 目录下。

Visual C++ Find in Files

有时，你必须查找某个特定的符号。Visual C++ Find in Files 就是完成这一工作的主要工具。它会在 MFC 源代码中指出符号所在位置。

Visual C++ Syntax-Coloring Tip

你可以通过定制 Visual C++ syntax-coloring 来自动地将 MFC 的每个关键字都加亮。这对你快速地将自己的类和 MFC 的类区分的最方便的工具。关于详细内容请查看 www.infopower.com.cn 中的 USERTYPE.DAT 文件

商业产品

ACI ObjectViewer 被认为是一个优秀的通用 C++浏览器（虽然它有 Macintosh-ish 的接口）。

还有一个共享的工具 IXFWIN（Interactive Cross Referencer for Windows），它是一个很好的“interactive grep”（交互查找）工具（比前面推荐的 Find in Files 更好）。

关于更多信息，请访问 WWW 站点 http://www.kudonet.com/~ixfwin 与这个工具的作者联系，或者向 ixfwin@kudonet.com 发送 E-mail。

Visual Extensions 是一个工具集，它可以插入到 Visual C++中（包括更好的 searching/grepping）。你可以通过联系 ray@newton.apple.com 获得更多的信息。

MFC 源代码指南

现在，让我们看一下 MFC 源代码的布局、结构和内容。当你要深入研究 MFC 源代码而需要一个快速指南时，这部分是一个很方便的指引。

MFC 目录结构

图 A-1 给出了 MFC 的结构树。

注意：你的安装可能略有不同，这依赖于你所选择的安装配置。如果你想知道所有的目录，可以在 Visual C++ 光盘中找到这些目录。

下面是对每个目录的简短解释：

- Mfc\——根目录。这里什么都没有，只是一些子目录。

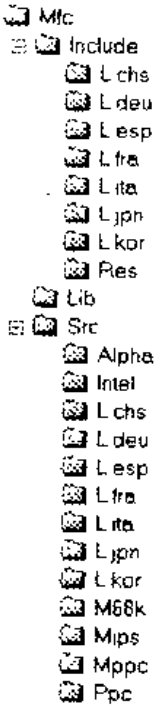


图 A-1 MFC 目录树

- Mfc\Include——包含了 MFC 的所有头文件（.H）、资源文件（.RC）、内嵌文件（.INL）、Macintosh 资源（.R）和 MFC Help ID（.HM）。
- Mfc\Include\L.chs——瑞典语资源。
- Mfc\Include\L.deu——德语资源。
- Mfc\Include\L.esp——西班牙语资源。
- Mfc\Include\L.fra——法语资源。
- Mfc\Include\L.ita——意大利语资源。
- Mfc\Include\L.jpn——日语资源。
- Mfc\Include\L.kor——韩语资源。
- Mfc\Include\Res——MFC 光标、位图以及在资源（.RC）脚本中包含的 Macintosh 光标和位图。
- Mfc\Lib——为了调试库，包含了预编译的 MFC 库（.LIB）和程序的数据库文件（.PDB）。
- Mfc\Src——存储了 MFC 的所有实现，包括 MFC 源文件（.CPP）、隐藏的头文件（.H）、资源脚本（.RC）和 makefile（.MAK）。
- Mfc\Src\Alpha——针对 Alpha 平台的 DLL .DEF/.PRF 文件（管理是不同的）。
- MFC\Src\Intel——针对 Intel 平台的 DLL .DEF/.PRF 文件。
- Mfc\Src\L.chs——瑞典语资源。
- Mfc\Src\L.deu——德语资源。
- Mfc\Src\L.esp——西班牙语资源。
- Mfc\Src\L.fra——法语资源。
- Mfc\Src\L.ita——意大利语资源。
- Mfc\Src\L.jpn——日语资源。
- Mfc\Src\L.kor——韩语资源。
- Mfc\Src\M68k——Macintosh Motorola 68K OLE 存根（stub）。
- Mfc\Src\Mips——针对 MIPS 平台的 DLL .DEF/.PRF 文件。
- Mfc\Src\Mppc——针对 Macintosh PowerPC 和该平台上的 OLE 存根的 DLL.DEF/.PRF 文件。
- Mfc\Src\Ppc——针对 NT 的 PowerPC 版本的 DLL .DEF/.PRF 文件。

.PRF 文件是什么？

我们关于这个问题也请教了院士，他的回答是这样的：

.PRF 文件就是我们摆脱工作集调节器（working set tuner）的一种方法。它包含了我们在调节工作集的过程中所遇到的函数链表。组织这些函数的目的是为了减少调节过程中涉及的页面的个数（一个示例过程就是导入一个默认的 appwiz mfc 应用程序，或导入写字板或在对话框应用程序中使用 OLE 控件等等）。如果你查看这些函数，你会看到文件前面的内容是经常遇到的，后面的内容是很少用到的。.PRF 文件是提供给链接器使用的，这样链接器就可以按照.PRF 文件中函数的顺序组织函数。.PRF 的名字来自单词“profiling”，但是在这种情况下我们做了一种不同的 profiling（工作集 profiling）。

MFC 头文件

在这一部分，我们会给出两种查看 MFC 头文件的方法。在第一部分，我们会看每个头文件（.H），并列出每个头文件所在的类。在第二部分是每个 MFC 类和相应的 MFC 头文件的链表。

头文件命名规则

MFC 头文件有一个非常有趣的命名规则。以下划线结尾的头文件（例如 afxplex_.h）仅由其他 MFC 头文件引用（下划线是微软标记特定的头文件的方式，它将这种头文件标记为“implementation”，和头文件自身的//Implementation 部分很类似）。应用程序应该避免直接引用这些以“_”结尾的头文件，因为在未来的 MFC 版本种它们可能会有很大改动，甚至消失。当然，这并不会阻止我们。

MFC 头文件到声明它的类的映射表

AFX.H—Non-GUI Classes

CRuntimeClass	CObject
CException	CArchiveException
CFileException	CSimpleException
CMemoryException	CNotSupportedException
CFile	CStdioFile
CMemFile	CString
CTimeSpan	CTime
CFileStatus	CMemoryState
CArchive	CDumpContext

AFXCMN.H—Windows Common Control Classes

CToolInfo	CImageList
CDragListBox	CListCtrl
CTreeCtrl	CSpinButtonCtrl
CHeaderCtrl	CSliderCtrl
CProgressCtrl	CHotKeyCtrl
CToolTipCtrl	CTabCtrl
CAnimateCtrl	CToolBarCtrl
CStatusBarCtrl	CRichEditCtrl

AFXCOLL.H—MFC Collections

CByteArray	CWordArray
CDWordArray	CUIntArray
CPtrArray	CObArray
CPtrList	CObList
CMapWordToOb	CMapWordToPtr
CMapPtrToWord	CMapPtrToPtr
CStringArray	CStringList
CMapStringToPtr	CMapStringToOb
CMapStringToString	

AFXCTL.H—OLE Control Classes

COleControlModule	CFontHolder

CPictureHolder
COleControl
COlePropertyPage
CPropExchange
CControlFrameWnd

AFXCVIEW.H—Control View Classes

CListView
CTreeView

AFXDAO.H—DAO Classes

CDaoException
CDaoRecordView
CDaoWorkspace
CDaoDatabase
CDaoRecordset
CDaoTableDef
CDaoQueryDef
CDaoFieldExchange
CDaoFieldCache
CDaoErrorInfo
CDaoWorkspaceInfo
CDaoDatabaseInfo
CDaoTableDefInfo
CDaoFieldInfo
CDaoIndexInfo
CDaoRelationInfo
CDaoQueryDefInfo
CDaoParameterInfo

AFXDB.H—ODBC Classes

CDBException
CFieldExchange
CDatabase
CRecordset
CRecordView
CRecordsetStatus
CFieldInfo

AFXDB_.H—CLongBinary Only

CLongBinary

AFXDD_.H—DDX/DDV Macro Declarations

AFXDISP.H—IDispatch and Class Factory Support

CConnectionPoint
COleException
COleDispatchException
COleObjectFactory
COleTemplateServer
COleDispatchDriver
COleVariant
COleCurrency
COleDateTime
COleDateTimeSpan

AFXDLGS.H—Common Dialogs

CCommonDialog
CFindReplaceDialog
CFileDialog
CFontDialog
CColorDialog
CPrintDialog
CPageSetupDialog
CPropertySheet
CPropertyPage

AFXDLL_.H—MFC Extension DLL Declarations

CDynLinkLibrary

AFXDLLX.H—MFC DLL Helpers

AFXEXT.H—Enhanced UI Classes

CBitmapButton
CControlBar
CStatusBar
CToolBar
CDialogBar
CSplitterWnd

CFormView CEditView
CMetaFileDC CRectTracker
CPrintInfo CPrintPreviewState
CCreateContext CControlBarInfo

AFXMSG_.H—Age Map Signature Declarations

AFXMT.H—Multithreaded Class Declarations

CSyncObject CSemaphore
CMutex CEvent
CCriticalSection CSingleLock
CMultiLock

AFXODLGS.H—OLE Common Dialogs

COleUILinkInfo COleDialog
COleInsertDialog COleConvertDialog
COleChangeIconDialog COlePasteSpecialDialog
COleLinksDialog COleUpdateDialog
COleBusyDialog COlePropertiesDialog
COleChangeSourceDialog

AFXOLE.H—MFC OLE Support

COleDocument COleLinkingDoc
COleServerDoc CDocItem
COleClientItem COleServerItem
COleDataSource COleDropSource
COleDropTarget COleMessageFilter
COleIPFrameWnd COleResizeBar
COleStreamFile COleDataObject

AFXPLEX_.H—MFC Collection Memory Helpers

CPlex

AFXPRIV.H—MFC Private Classes: Undocumented

CSharedFile CPreviewDC
CPreviewView COleCntrFrameWnd
CMiniDockFrameWnd CRecentFileList
CDockState CDockContext
CArchiveStream CDialogTemplate
CDockBar COleControlLock

AFXRES.H—MFC Resource Symbol Definitions

AFXRICH.H—MFC Rich Text View Classes

CRichEditView CRichEditDoc
CRichEditCntrItem

AFXSOCK.H—WinSock Classes

CAsynchSocket CSocket
CSocketFile CSocketWnd

AFXSTAT_.H—Various MFC STATE Structure Definitions (AFX_STATE, AFX_THREAD_STATE,

and so on)

AFXTEMPL.H—MFC Template Collection Classes

AFXTLS_.H—MFC Thread Local Storage Information

CSimpleList	CTypedSimpleList
CThreadLocal	CProcessLocal

AFXV_CFG.H—Defines _AFX_PORTABLE for Portability across Compilers

AFXV_CPU.H—Declares/Defines CPU-Specific Stuff

AFXV_DLL.H—Special Header for MFC Extension DLLs

AFXV_MAC.H—Macintosh Definitions

AFXV_W32.H—Target/Version Control for Win32

AFXVER_.H—MFC Version Information

AFXWIN.H—MFC "Window" Classes

CSize	CPoint
CRect	CResourceException
CUserException	CGdiObject
CPen	CBrush
CFont	CBitmap
CPalette	CRgn
CDC	CClientDC
CWindowDC	CPaintDC
CMenu	CCmdTarget
CWnd	CDialog
CStatic	CButton
CListBox	CCheckListBox
CComboBox	CEdit
CScrollBar	CFrameWnd
CMDIFrameWnd	CMDIChildWnd
CMiniFrameWnd	CView
CScrollView	CWinThread
CWinApp	CDocTemplate
CSingleDocTemplate	CMultiDocTemplate
CDocument	CCmdUI
CDataExchange	CCommandLineInfo
CDocManager	CCtrlView

WINRES.H—MFC Specific Windows Resource Declarations

MFC window messages, bitmap IDs, extended styles, virtual key codes, and so on.

MFC 类到头文件的映射表

CAnimateCtrl	AFXCMN.H	CArchive	AFX.H
CArchiveException	AFX.H	CArchiveStream	AFXPRIV.H
CAsynchSocket	AFXSOCK.H	CBitmap	AFXWIN.H
CBitmapButton	AFXEXT.H	CBrush	AFXWIN.H

CButton	AFXWIN.H
CCheckListBox	AFXWIN.H
CCmdTarget	AFXWIN.H
CColorDialog	AFXDLGS.H
CCommandLineInfo	AFXWIN.H
CConnectionPoint	AFXDISP.H
CControlBarInfo	AFXEXT.H
CCreateContext	AFXEXT.H
CCtrlView	AFXWIN.H
CDC	AFXWIN.H
CDaoDatabase	AFXDAO.H
CDaoErrorInfo	AFXDAO.H
CDaoFieldCache	AFXDAO.H
CDaoFieldInfo	AFXDAO.H
CDaoParameterInfo	AFXDAO.H
CDaoQueryDefInfo	AFXDAO.H
CDaoRecordset	AFXDAO.H
CDaoTableDef	AFXDAO.H
CDaoWorkspace	AFXDAO.H
CDataExchange	AFXWIN.H
CDialog	AFXWIN.H
CDialogTemplate	AFXPRIV.H
CDocManager	AFXWIN.H
CDockBar	AFXPRIV.H
CDockState	AFXPRIV.H
CDragListBox	AFXCMN.H
CDynLinkLibrary	AFXDLL_.H
CEditView	AFXEXT.H
CException	AFX.H
CFieldInfo	AFXDB.H
CFileDialog	AFXDLGS.H
CFileStatus	AFX.H
CFont	AFXWIN.H
CFontHolder	AFXCTL.H
CFrameWnd	AFXWIN.H
CHeaderCtrl	AFXCMN.H
CImageList	AFXCMN.H
CListCtrl	AFXCMN.H
CLongBinary	AFXDB_.H
CMDIFrameWnd	AFXWIN.H
CMapPtrToWord	AFXCOLL.H
CMapStringToPtr	AFXCOLL.H
CMapWordToOb	AFXCOLL.H
CMemFile	AFX.H
CMemoryState	AFX.H
CByteArray	AFXCOLL.H
CClientDC	AFXWIN.H
CCmdUI	AFXWIN.H
CComboBox	AFXWIN.H
CCommonDialog	AFXDLGS.H
CControlBar	AFXEXT.H
CControlFrameWnd	AFXCTL.H
CCriticalSection	AFXMT.H
CDBException	AFXDB.H
CDWordArray	AFXCOLL.H
CDaoDatabaseInfo	AFXDAO.H
CDaoException	AFXDAO.H
CDaoFieldExchange	AFXDAO.H
CDaoIndexInfo	AFXDAO.H
CDaoQueryDef	AFXDAO.H
CDaoRecordView	AFXDAO.H
CDaoRelationInfo	AFXDAO.H
CDaoTableDefInfo	AFXDAO.H
CDaoWorkspaceInfo	AFXDAO.H
CDatabase	AFXDB.H
CDialogBar	AFXEXT.H
CDocItem	AFXOLE.H
CDocTemplate	AFXWIN.H
CDockContext	AFXPRIV.H
CDocument	AFXWIN.H
CDumpContext	AFX.H
CEdit	AFXWIN.H
CEvent	AFXMT.H
CFieldExchange	AFXDB.H
CFile	AFX.H
CFileException	AFX.H
CFindReplaceDialog	AFXDLGS.H
CFontDialog	AFXDLGS.H
CFormView	AFXEXT.H
CGdiObject	AFXWIN.H
CHotKeyCtrl	AFXCMN.H
CListBox	AFXWIN.H
CListView	AFXCVIEW.H
CMDIChildWnd	AFXWIN.H
CMapPtrToPtr	AFXCOLL.H
CMapStringToOb	AFXCOLL.H
CMapStringToString	AFXCOLL.H
CMapWordToPtr	AFXCOLL.H
CMemoryException	AFX.H
CMenu	AFXWIN.H

CMetaFileDC	AFXEXT.H	CMiniDockFrameWnd	AFXPRIV.H
CMiniFrameWnd	AFXWIN.H	CMultiDocTemplate	AFXWIN.H
CMultiLock	AFXMT.H	CMutex	AFXMT.H
CNotSupportedException	AFX.H	CObArray	AFXCOLL.H
CObList	AFXCOLL.H	CObject	AFX.H
COleBusyDialog	AFXODLGS.H	COleChangeIconDialog	AFXODLGS.H
COleChangeSourceDialog	AFXODLGS.H	COleClientItem	AFXOLE.H
COleCntrFrameWnd	AFXPRIV.H	COleControl	AFXCTL.H
COleControlLock	AFXPRIV.H	COleControlModule	AFXCTL.H
COleConvertDialog	AFXODLGS.H	COleCurrency	AFXDISP.H
COleDataObject	AFXOLE.H	COleDataSource	AFXOLE.H
COleDateTime	AFXDISP.H	COleDateTimeSpan	AFXDISP.H
COleDialog	AFXODLGS.H	COleDispatchDriver	AXDISP.H
COleDispatchException	AFXDISP.H	COleDocument	AFXOLE.H
COleDropSource	AFXOLE.H	COleDropTarget	AFXOLE.H
COleException	AFXDISP.H	COleIPFrameWnd	AFXOLE.H
COleInsertDialog	AFXODLGS.H	COleLinkingDoc	AFXOLE.H
COleLinksDialog	AFXODLGS.H	COleMessageFilter	AFXOLE.H
COleObjectFactory	AFXDISP.H	COlePasteSpecialDialog	AFXODLGS.H
COlePropertiesDialog	AFXODLGS.H	COlePropertyPage	AFXCTL.H
COleResizeBar	AFXOLE.H	COleServerDoc	AFXOLE.H
COleServerItem	AFXOLE.H	COleStreamFile	AFXOLE.H
COleTemplateServer	AFXDISP.H	COleUILinkInfo	AFXODLGS.H
COleUpdateDialog	AFXODLGS.H	COleVariant	AFXDISP.H
CPageSetupDialog	AFXDLGS.H	CPaintDC	AFXWIN.H
CPalette	AFXWIN.H	CPen	AFXWIN.H
CPictureHolder	AFXCTL.H	CPlex	AFXPLEX_.H
CPoint	AFXWIN.H	CPreviewDC	AFXPRIV.H
CPreviewView	AFXPRIV.H	CPrintDialog	AFXDLGS.H
CPrintInfo	AFXEXT.H	CPrintPreviewState	AFXEXT.H
CProcessLocal	AFXTLS_.H	CProgressCtrl	AFXCMN.H
CPropertyPage	AFXDLGS.H	CPropertySheet	AFXDLGS.H
CPropExchange	AFXCTL.H	CPtrArray	AFXCOLL.H
CPtrList	AFXCOLL.H	CRecentFileList	AFXPRIV.H
CRecordView	AFXDB.H	CRecordset	AFXDB.H
CRecordsetStatus	AFXDB.H	CRect	AFXWIN.H
CRectTracker	AFXEXT.H	CResourceException	AFXWIN.H
CRgn	AFXWIN.H	CRichEditCntrItem	AFXRICH.H
CRichEditCtrl	AFXCMN.H	CRichEditDoc	AFXRICH.H
CRichEditView	AFXRICH.H	CRuntimeClass	AFX.H
CScrollBar	AFXWIN.H	CScrollView	AFXWIN.H
CSemaphore	AFXMT.H	CSharedFile	AFXPRIV.H
CSimpleException	AFX.H	CSimpleList	AFXTLS_.H
CSingleDocTemplate	AFXWIN.H	CSingleLock	AFXMT.H

CSize	AFXWIN.H	CSliderCtrl	AFXCMN.H
CSocket	AFXSOCK.H	CSocketFile	AFXSOCK.H
CSocketWnd	AFXSOCK.H	CSpinButtonCtrl	AFXCMN.H
CSplitterWnd	AFXEXT.H	CStatic	AFXWIN.H
CStatusBar	AFXEXT.H	CStatusBarCtrl	AFXCMN.H
CStdioFile	AFX.H	CString	AFX.H
CStringArray	AFXCOLL.H	CStringList	AFXCOLL.H
CSyncObject	AFXMT.H	CTabCtrl	AFXCMN.H
CThreadLocal	AFXTLS_.H	CTime	AFX.H
CTimeSpan	AFX.H	CToolBar	AFXEXT.H
CToolBarCtrl	AFXCMN.H	CToolInfo	AFXCMN.H
CToolTipCtrl	AFXCMN.H	CTreeCtrl	AFXCMN.H
CTreeView	AFXCVIEW.H	CTypedSimpleList	AFXTLS_.H
CUIntArray	AFXCOLL.H	CUserException	AFXWIN.H
CView	AFXWIN.H	CWinApp	AFXWIN.H
CWinThread	AFXWIN.H	CWindowDC	AFXWIN.H
CWnd	AFXWIN.H	CWordArray	AFXCOLL.H

MFC 内嵌文件

在看什么内容在哪个文件之前，我们先回答一个重要问题：内嵌（inline）文件是什么？MFC 在调试版本创建时关闭 C++的内嵌代码，在发布版本创建时启用 C++的内嵌代码。MFC 这样做是因为内嵌的函数很难调试，因为编译器已经做过代码替换了。

为了完成内嵌的关闭及启用，MFC 就使用了内嵌文件，内嵌文件的扩展名是 INL。每个 INL 文件对应一个头文件（例如 AFX.H、AFX.INL），文件中有很多 INLINE 函数。MFC 没有再使用 C++关键字“inline”，而是使用了_AFX_INLINE。

当 MFC 的_DEBUG 启用时，也就是当内嵌被关闭时，就发生了一件有趣的事。在这种情况下，_AFX_INLINE 就没有定义。如果定义了_DEBUG，CPP 文件就引入（#include）这些 INL 文件。然后在编译时，CPP 文件被认为包含了非内嵌的源代码。

在发布版本创建的情况下（_DEBUG 没有定义），没有引入 CPP 文件。这时，_AFX_INLINE 被定义为关键字“inline”，INL 文件被引入（#include）到相应的头文件中，在头文件中编译器将它还原成内嵌的。

因为开始的时候内嵌文件有点让人迷惑，所以我们先看个例子：AFX.INL。相应的头文件是 AFX.H，相应的 CPP 文件是 AFXINL1.CPP。

下面二行代码来自文件 AFXVER_.H：

```
#ifndef _DEBUG
    #define _AFX_ENABLE_INLINES
#endif
```

在 AFX.H 文件中，我们就有下面这些代码：

```
#ifdef _AFX_ENABLE_INLINES
    #define _AFX_INLINE inline
    #include <afx.inl>
#endif
```

在 AFXINL1.CPP 中，我们就有下面这些代码：

```
#ifndef _AFX_ENABLE_INLINES
    #define _AFX_INLINE
    #include "afx.inl"
#endif
```

当 MFC 在_DEBUG 模式下创建时，AFX_ENABLE_INLINES 没有定义，所以 AFX.H 没有引入 AFX.INL。但是，AFXINL1.CPP 中引入了 AFX.INL，位置在#define_AFX_INLINE 之后。

当 MFC 在发布模式（!_DEBUG_）下创建时，AFX_ENABLE_INLINES 有定义，所以 AFX.H 中引入了 AFX.INL，位置在将_AFX_INLINE 定义为“inline”之后。在这种情况下，AFXINL1.CPP 没有做任何事。

MFC 内嵌文件和相应的头文件通常都是一一对应的，除了 AFXWIN1.INL 和 AFXWIN2.INL。这两个内嵌函数都对应着 AFXWIN.H 头文件。微软将它们分开是因为内容太多，太长。

你会在 MFC 源文件部分看到 CPP 到 INL 的映射关系。

MFC 资源

MFC 的源代码中有很多 RC 文件，这些文件在“include”目录下和“src”目录下。你可以用 Visual C++打开这些文件看看。知道里面是什么对你很有帮助，所以下面看一下每个 RC 文件中都有些什么：

- include\AFXCTL.RC——OLE 控件资源。
- include\AFXDB.RC——数据库资源。
- include\AFXOLECL.RC——OLE 客户端资源。
- include\AFXOLESV.RC——OLE 服务器端资源。
- include\AFXPRINT.RC——打印资源。
- include\AFXRES.RC——默认资源。
- src\INDICATE.RC——AFX 状态栏的提示字符串。
- src\MFCDB.RC——为 MFC DB DLL 定义了资源。
- src\MFCDLL.RC——为 MFC DLL 定义了资源，包含了 include*.RC 文件。
- src\MFCINTL.RC——针对语言的 DLL 的资源。
- src\MFCNET.RC——MFC NET DLL 的资源。
- src\MFCOLE.RC——MFC OLE DLL 的资源。
- src\PROMPTS.RC——包含了所有标准的 MFC 菜单提示的字符串表。

依赖于语言的 ID，这些 RC 文件包含了针对语言的子目录字符串（例如 L.fra、L.deu 等等）。

如果你有空闲时间，可以打开 MFCDLL.RC，这里面有很多很酷的资源。看你是否能指出这些资源都在什么地方使用了，以及为什么使用。

一个神秘的资源

图 A-2 显示了一个很奇怪的资源，它在文件 MFCDLL.RC 中。

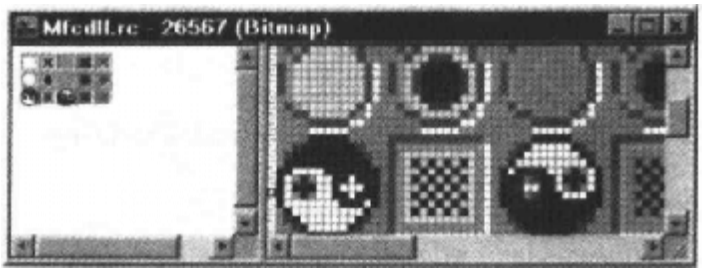


图 A-2 神秘的乱画的资源

这个资源是一个位图，它的 ID 是 26567。它包含了所有三状态复选框（three-state checkbox）的所有可能复选框图像。但是仔细看一下最后一行，这里有两个太极图。

你知道为什么这里有两个太极图码？可能 MFC 小组想要做秘密的太极 MFC 控件类吧。

实际上，这些状态对于一个三状态的复选框而言是永远不会用到的，所以看起来画这张图的人在开玩笑。这个就是 MFC 在乱画。想想看，Windows 95 的每个拷贝在 MFC DLL 中都带有这样的位图（你的应用程序中也有）。

MFC 源文件

在这部分，我们会揭示每个 MFC 源文件中都引入了什么文件。和引入文件部分一样，我们首先看从文件到类的映射关系，然后看从类到文件的映射关系，这样方便你查找。

有些 MFC 类只在一个源文件中实现。大一些的 MFC 类（例如 CWnd、CFrameWnd、CWinApp）都跨越了几个文件。这个指南会告诉你哪个文件包含了类实现的主要部分。你可能需要用一个查找工具来找到指定的成员函数。而且，别忘了，还有一些类是完全实现在内嵌文件中的，所以在执行查找时还要引入 INL 文件。

MFC 源文件到文件内容的映射

AFXABORT.CPP	AfxAbort() helper
AFXASERT.CPP	ASSERT helper: AfxAssertFailedLine()
AFXCRIT.CPP	MFC internal thread helpers: for marking critical sections inside MFC
AFXDBCS.CPP	Static _afxDBCS symbol lives and is initialized here
AFXINL1.CPP	Inline file for AFX.INL, AFXCOLL.INL, AFXDLGS.INL, AFXEXT.INL
AFXINL2.CPP	Inline file for AFXWIN1.INL
AFXINL3.CPP	Inline file for AFXWIN2.INL
AFXMEM.CPP	MFC memory helpers: operator new, debug memory allocators and checkers, CMemoryState
AFXSTATE.CPP	MFC state helpers, constructors, and destructors (_AFX_THREAD_STATE and AFX_MODULE_STATE)
AFXTLS.CPP	CSimpleList, CThreadLocal
AFXTRACE.CPP	MFC TRACE helpers, including a map of all messages to strings
APP3D.CPP	_AFX_CTL3D_STATE helpers, constructor, destructor
APP3DS.CPP	CWinApp::Enable3dControlsStatic()
APPCORE.CPP	CWinApp "core," CCommandLineInfo
APPDLG.CPP	CWinApp "dialog"
APPGRAY.CPP	CWinApp gray background color

APPHELP.CPP	Context-sensitive help helpers
APPHELPX.CPP	CWinApp help
APPINIT.CPP	CWinApp initialization and helpers
APPMODUL.CPP	_AFX_TERM_APP_STATE constructor, and other things
APPPRNT.CPP	CWinApp printing
APPTERM.CPP	AfxWinTerm() helper; frees everything
APPUI.CPP	CWinApp user interface
APPUI2.CPP	More CWinApp UI
APPUI3.CPP	Still more CWinApp UI
ARCCORE.CPP	CArchive "core," CRuntimeClass
ARCEX.CPP	CArchiveException
ARCOBJ.CPP	CArchive/CObject interface
ARRAY_B.CPP	CByteArray
ARRAY_D.CPP	CDWordArray
ARRAY_O.CPP	CObArray
ARRAY_P.CPP	CPtrArray
ARRAY_S.CPP	CStringArray
ARRAY_U.CPP	CUIntArray
ARRAY_W.CPP	CWordArray
AUXDATA.CPP	AUX_DATA() constructor and destructor
BARCORE.CPP	CControlBar "core"
BARDLG.CPP	CDialogBar
BARDOCK.CPP	CDockBar, CMiniDockFrameWnd
BARSTAT.CPP	CStatusBar
BARTOOL.CPP	CToolBar
CCDATA.CPP	Common control class helper routines
CMDTARG.CPP	CCmdTarget, CCmdUI
CTLCACHE.CPP	COleControl cache
CTLCONN.CPP	COleControl connection
CTLCORE.CPP	COleControl "core"
CTLDATA.CPP	COleControl "data"
CTLEVENT.CPP	COleControl "event"
CTLFONT.CPP	CFontHolder
CTLFRAME.CPP	CControlFrameWnd
CTLINL.CPP	OLE control inline file (AFXCTL.INL)
CTLINPLC.CPP	COleControl "in-place activation"
CTLINTL.CPP	_WEP for a resource-only DLL
CTLLIC.CPP	AfxVerifyLicFile() helper
CTLMODUL.CPP	COleControlModule
CTLOBJ.CPP	COleControl::XOleObject()
CTLPBAG.CPP	CBlobProperty, COleControl::XPeristPropertyBag
CTLPICT.CPP	CPictureHolder
CTLPPG.CPP	COlePropertyPage
CTLPROP.CPP	COleControl "property"
CTLPROPX.CPP	CPropExchange
CTLPSET.CPP	COleControl "property set," CPropsetPropExchange

CTLPSTG.CPP	COleControl::XPersistStorage
CTLPSTM.CPP	COleControl::XPersistStreamInit
CTLREFL.CPP	CReflectorWnd, CParkingWnd
CTLREG.CPP	OLE registration helpers
CTLTRACK.CPP	COleControl "tracker"
CTLVIEW.CPP	COleControl::XViewObject
DAOCORE.CPP	CDaoDatabase, CDaoErrorInfo, CDaoException, CDaoFieldInfo, CDaoIndexInfo, CDaoParameterInfo, CDaoQueryDef, CDaoQueryDefInfo, CDaoRecordset, CDaoRelationInfo, CDaoTableDef, CDaoTableDefInfo, CDaoWorkspace, and CDaoWorkspaceInfo
DAODFX.CPP	CDaoFieldCache, CDaoFieldExchange
DAOVIEW.CPP	CDaoRecordView
DBCORE.CPP	CDBException, CDatabase, CRecordset
DBFLT.CPP	Record exchange helpers (RFX_*)
DBLONG.CPP	CLongBinary
DBRFX.CPP	CFieldExchange
DBVIEW.CPP	CRecordView
DCMETA.CPP	CMetaFileDC
DCPREV.CPP	CPreviewDC
DLGCLR.CPP	CColorDialog
DLGCOMM.CPP	CCommonDialog and helpers
DLGCORE.CPP	CDialog "core"
DLGDATA.CPP	CDataExchange and DDX/DDV helpers
DLGFILE.CPP	CFileDialog
DLGFLOAT.CPP	Floating point DDX/DDV helpers
DLGFNT.CPP	CFontDialog
DLGFR.CPP	CFindReplaceDialog
DLGPRNT.CPP	CPageSetupDialog, CPrintDialog
DLGPROP.CPP	CPropertyPage, CPropertySheet
DLGTEMPL.CPP	CDialogTemplate
DLLDB.CPP	Special DLL code for ODBC/DAO
DLLINIT.CPP	CDynLinkLibrary, extension DLL helpers
DLLMODUL.CPP	DLL module
DLLNET.CPP	Initializes MFC extension DLL
DLLOLE.CPP	Special OLE code for DLLs
DOCCORE.CPP	CDocument, CMirrorFile
DOCKCONT.CPP	CDockContext
DOCKSTAT.CPP	CControlBarInfo, CDockState, CDockBar, and CControlBar
DOCMAPI.CPP	CDocument MAPI support
DOCMGR.CPP	CDocManager
DOCMULTI.CPP	CMultiDocTemplate
DOCSINGL.CPP	CSingleDocTemplate
DOCTEMPL.CPP	CDocTemplate
DUMPCONT.CPP	CDumpContext
DUMPFLT.CPP	CDumpContext for floats
DUMPINIT.CPP	MFC dumping initialization

DUMPOUT.CPP	MFC dump output helpers.
EXCEPT.CPP	CException, CSimpleException, CMemoryException, CNotSupportedException, and CUserException
FILECORE.CPP	CFile
FILELIST.CPP	CRecentFileList
FILEMEM.CPP	CMemFile
FILESHRD.CPP	CSharedFile
FILEST.CPP	CFileStatus
FILETXT.CPP	CStdioFile
FILEX.CPP	CFileException
LIST_O.CPP	CObList
LIST_P.CPP	CPtrList
LIST_S.CPP	CStringList
MAP_PP.CPP	CMapPtrToPtr
MAP_PW.CPP	CMapPtrToWord
MAP_SO.CPP	CMapStringToOb
MAP_SP.CPP	CMapStringToPtr
MAP_SS.CPP	CMapStringToString
MAP_WO.CPP	CMapWordToOb
MAP_WP.CPP	CMapWordToPtr
MTCORE.CPP	CSyncObject
MTEX.CPP	CSemaphore, CMutex, CEvent, CSingleLock, and CMultiLock
NOLIB.CPP	#pragmas that run off libraries
OBJCORE.CPP	CObject
OCCCONT.CPP	CWnd OLE control containment
OCCDDX.CPP	DDX routines for OLE controls
OCCDDXF.CPP	DDX float routines for OLE controls
OCCDLG.CPP	COccManager
OCCEVENT.CPP	CCmdTarget OLE control code
OCCLOCK.CPP	COleControlLock
OCCMGR.CPP	COccManager "core"
OCCSITE.CPP	COleControlSite
OLEBAR.CPP	COleResizeBar
OLECALL.CPP	_AfxDispatchCall assembly routine for IDispatch (yucky!)
OLECLI1.CPP	COleClientItem
OLECLI2.CPP	COleFrameHook, COleClientItem
OLECLI3.CPP	COleClientItem
OLECONN.CPP	CConnectionPoint, COleConnPtContainer, and CEnumConnPoints
OLEDATA.CPP	OLE helpers
OLEDISP1.CPP	COleDispatchException
OLEDISP2.CPP	COleDispatchDriver
OLEDLGS1.CPP	COleChangeIconDialog, COleConvertDialog, COleDialog, COleInsertDialog, COleLinksDialog, COlePasteSpecialDialog, COleUpdateDialog, and COleUILinkInfo
OLEDLGS2.CPP	COleBusyDialog, COleDialog
OLEDLGS3.CPP	COleChangeSourceDialog, COleDialog, and COlePropertiesDialog

OLEDLL.CPP	OLE DLL helpers
OLEDOBJ1.CPP	COleDataObject
OLEDOBJ2.CPP	COleDataSource
OLEDOC1.CPP	CDocItem, COleDocument
OLEDOC2.CPP	COleDocument
OLEDROP1.CPP	COleDropSource
OLEDROP2.CPP	COleDropTarget
OLEENUM.CPP	CEnumArray
OLEEXP.CPP	OLE DLL helpers
OLEFACT.CPP	COleObjectFactory
OLEINIT.CPP	OLE_DATA and OLE initialization helpers
OLEIPFRM.CPP	COleCntrFrameWnd, COleIPFrameWnd
OLELINK.CPP	COleLinkingDoc
OLELOCK.CPP	OLE app exit helpers
OLEMISC.CPP	COleException
OLEMSGF.CPP	COleMessageFilter
OLEPRO32.CPP	OLE property helpers
OLEPSET.CPP	CProperty, CPropertySet
OLEREG.CPP	OLE registration helpers
OLESTRM.CPP	COleStreamFile
OLESVR1.CPP	COleServerDoc
OLESVR2.CPP	COleServerItem
OLETSVR.CPP	COleTemplateServer
OLETYPLB.CPP	CCmdTarget IDispatch routines, CTypeLibCache
OLEUI1.CPP	COleClientItem, COleDocument
OLEUI2.CPP	COleDocument
OLEUNK.CPP	MFC IUnknown helpers
OLEVAR.CPP	COleCurrency, COleDateTime, COleTimeSpan, and COleVariant
OLEVERB.CPP	CEnumOleVerb
PLEX.CPP	CPlex
PPGCOLOR.CPP	CColorButton, CColorPropPage
PPGFONT.CPP	CFontPropPage
PPGPICT.CPP	CPicturePropPage
PPGSTOCK.CPP	CStockPropPage
SOCKCORE.CPP	CAsyncSocket, CSocket, CSocketWnd, CSocketFile
STDAFX.CPP	Used for precompiled headers
STRCORE.CPP	CString
STREX.CPP	CString advanced operators
THRDCORE.CPP	CWinThread
TIMECORE.CPP	CTime, CTimeSpan
TOOLTIP.CPP	CToolTipCtrl, CWnd tooltip code, and CToolInfo
TRCKRECT.CPP	CRectTracker
VALIDADD.CPP	AFX memory helpers: AfxIsValidAddress()
VIEWCMN.CPP	CListView, CTreeView
VIEWCORE.CPP	CView, CCtrlView

VIEWFORM.CPP	CFormView
VIEWPREV.CPP	CPreviewView, CPrintPreviewState, CView print preview
VIEWPRNT.CPP	CPrintingDialog, CPrintInfo, CView printing
VIEWRICH.CPP	CRichEditView, CRichEditDoc, CRichEditCntrItem
VIEWSCRL.CPP	CScrollView
WINBTN.CPP	CBitmapButton
WINCORE.CPP	CWnd "core"
WINCTRL1.CPP	CStatic, CButton, CListBox, CComboBox, CEdit, and CScrollBar
WINCTRL2.CPP	CDragListBox, CSplitButtonCtrl, CSliderCtrl, CProgressCtrl, CHeaderCtrl, CHotKeyCtrl, CAnimateCtrl, CTabCtrl, CTreeCtrl, CListCtrl, CToolBarCtrl, CStatusBarCtrl, CImageList, and CArchiveStream
WINCTRL3.CPP	CCheckListBox
WINCTRL4.CPP	CRichEditCtrl
WINFRM.CPP	CFrameWnd core
WINFRM2.CPP	CFrameWnd docking
WINFRMX.CPP	CWnd/CFrameWnd help support
WINGDI.CPP	CDC, CClientDC, CWindowDC, CPaintDC, CPen, CBrush, CFont, CBitmap, CPalette, CRgn, and CGdiObject
WINGDIX.CPP	CDC utility members, some GDI helpers
WINHAND.CPP	Internal MFC CHandleMap
WINMAIN.CPP	AfxWinMain()
WINMDI.CPP	CMDIFrameWnd, CMDIChildWnd
WINMENU.CPP	CMenu
WINMINI.CPP	CMiniFrameWnd
WINOCC.CPP	CWnd OLE control members
WINSPLIT.CPP	CSplitterWnd
WINSTR.CPP	String helpers
WINUTIL.CPP	Window helpers

MFC 类到源文件的映射

CArchive	ARCCORE.CPP, ARCOBJ.CPP
CArchiveException	ARCEX.CPP
CArchiveStream	WINCTRL2.CPP
CAsynchSocket	SOCKCORE.CPP
CBlobProperty	CTLPBAG.CPP
CBitmap	WINGDI.CPP
CBitmapButton	WINBTN.CPP
CBrush	WINGDI.CPP
CButton	WINCTRL1.CPP
CByteArray	ARRAY_B.CPP
CCheckListBox	WINCTRL3.CPP
CClientDC	WINGDI.CPP
CCmdTarget	CMDTARG.CPP, OCCEVENT.CPP, OLETYPLB.CPP
CCmdUI	CMDTARG.CPP
CColorButton	PPGCOLOR.CPP

CColorPropPage	PPGCOLOR.CPP
CColorDialog	DLGCLR.CPP
CComboBox	WINCTRL1.CPP
CCommandLineInfo	APPCORE.CPP
CCommonDialog	DLGCOMM.CPP
CConnectionPoint	OLECONN.CPP
CControlBar	BARCORE.CPP, DOCKSTAT.CPP
CControlBarInfo	DOCKSTATE.CPP
ControlFrameWnd	CTLFRMAE.CPP
CCreateContext	100% inline
CCriticalSection	100% inline
CCtrlView	VIEWCORE.CPP
CDBException	DBCORE.CPP
CDC	WINGDI.CPP, WINGDIX.CPP
CDWordArray	ARRAY_D.CPP
CDaoDatabase	DAOCORE.CPP
CDaoDatabaseInfo	DAOCORE.CPP
CDaoErrorInfo	DAOCORE.CPP
CDaoException	DAOCORE.CPP
CDaoFieldCache	DAODFX.CPP
CDaoFieldExchange	DAODFX.CPP
CDaoFieldInfo	DAOCORE.CPP
CDaoIndexInfo	DAOCORE.CPP
CDaoParameterInfo	DAOCORE.CPP
CDaoQueryDef	DAOCORE.CPP
CDaoQueryDefInfo	DAOCORE.CPP
CDaoRecordView	DAOVIEW.CPP
CDaoRecordset	DAOCORE.CPP
CDaoRelationInfo	DAOCORE.CPP
CDaoTableDef	DAOCORE.CPP
CDaoTableDefInfo	DAOCORE.CPP
CDaoWorkspace	DAOCORE.CPP
CDaoWorkspaceInfo	DAOCORE.CPP
CDataExchange	DLGDATA.CPP
CDatabase	DBCORE.CPP
CDialog	DLGCORE.CPP
CDialogBar	BARDLG.CPP
CDialogTemplate	DLGTMPL.CPP
CDocItem	OLEDOC1.CPP
CDocManager	DOCMGR.CPP
CDocTemplate	DOCTEMPL.CPP
CDockBar	BARDOCK.CPP, DOCKSTAT.CPP
CDackContext	DOCKCONT.CPP
CDockState	DOCKSTAT.CPP
CDocument	DOCCORE.CPP, DOCMAPI.CPP

CDragListBox	WINCTRL2.CPP
CDumpContext	DUMPCONT.CPP, DUMPFLT.CPP
CDynLinkLibrary	DLLINIT.CPP
CEdit	WINCTRL1.CPP
CEditView	VIEWEDIT.CPP
CEnumArray	OLEENUM.CPP
CEnumConnPoints	OLECONN.CPP
CEnumOleVerb	OLEVERB.CPP
CEvent	MTEX.CPP
CException	EXCEPT.CPP
CFieldExchange	DBRFX.CPP
CFieldInfo	100% inline
CFile	FILECORE.CPP
CFileDialog	DLGFILE.CPP
CFileException	FILEX.CPP
CFileStatus	FILEST.CPP
CFindReplaceDialog	DLGFR.CPP
CFont	WINGDI.CPP
CFontDialog	DLGFNT.CPP
CFontHolder	CTLFONT.CPP
CFontPropPage	PPGFONT.CPP
CFormView	VIEWFORM.CPP
CFrameWnd	WINFRM.CPP, WINFRM2.CPP, WINFRMX.CPP
CGdiObject	WINGDI.CPP
CHeaderCtrl	WINCTRL2.CPP
CHotKeyCtrl	WINCTRL2.CPP
CImageList	WINCTRL2.CPP
CListBox	WINCTRL1.CPP
CListCtrl	WINCTRL2.CPP
CListView	VIEWCMN.CPP
CLongBinary	DBLONG.CPP
CMDIChildWnd	WINMDI.CPP
CMDIFrameWnd	WINMDI.CPP
CMapPtrToPtr	MAP_PP.CPP
CMapPtrToWord	MAP_PW.CPP
CMapStringToOb	MAP_SO.CPP
CMapStringToPtr	MAP_SP.CPP
CMapStringToString	MAP_SS.CPP
CMapWordToOb	MAP_WO.CPP
CMapWordToPtr	MAP_WP.CPP
CMemFile	FILEMEM.CPP
CMemoryException	EXCEPT.CPP
CMemoryState	AFXMEM.CPP
CMenu	WINMENU.CPP
CMetaFileDC	DCMETA.CPP

CMiniDockFrameWnd	BARDOCK.CPP
CMiniFrameWnd	WINMINI.CPP
CMirrorFile	DOCCORE.CPP
CMultiDocTemplate	DOCMULTI.CPP
CMultiLock	MTEX.CPP
CMutex	MTEX.CPP
CNotSupportedException	EXCEPT.CPP
CObArray	ARRAY_O.CPP
CObList	LIST_O.CPP
CObject	OBJCORE.CPP
COccManager	OCCDLG.CPP, OCCMGR.CPP
COleBusyDialog	OLEDLGS2.CPP
COleChangeIconDialog	OLEDLGS1.CPP
COleChangeSourceDialog	OLEDLGS3.CPP
COleClientItem	OLECLI1.CPP, OLECLI2.CPP, OLECLI3.CPP, OLEUI1.CPP
COleCntrFrameWnd	OLEIPFRM.CPP
COleConnPtContainer	OLECONN.CPP
COleControl	CTLCACHE.CPP, CTLCONN.CPP, CTLCORE.CPP, CTLDATA.CPP, CTLEVENT.CPP, CTLFONT.CPP, CTLFRAME.CPP, CTLINPLC.CPPP, CTLINTL.CPP, CTLLIC.CPP CTLOBJ.CPP, CTLPBAG.CPP, CTLPICT.CPP, CTLPPG.CPP, CTLPROP.CPP, CTLPROPX.CPP, CTLPSET.CPP, CTLPSTG.CPP, CTLPSTM.CPP, CTLREFL.CPP, CTLREG.CPP, CTLTRACK.CPP, CTLVIEW.CPP
COleControlLock	OCCLOCK.CPP
COleControlModule	CTLMODUL.CPP
COleControlSite	OCCSITE.CPP
COleConvertDialog	OLEDLGS1.CPP
COleCurrency	OLEVAR.CPP
COleDataObject	OLEDOBJ1.CPP
COleDataSource	OLEDOBJ2.CPP
COleDateTime	OLEVAR.CPP
COleDateTimeSpan	OLEVAR.CPP
COleDialog	OLEDLGS1.CPP, OLEDLGS2.CPP, OLEDLGS3.CPP
COleDispatchDriver	OLEDISP2.CPP
COleDispatchException	OLEDISP1.CPP
COleDocument	OLEDOC1.CPP, OLEUI2.CPP, OLEDOC2.CPP, OLEUI1.CPP
COleDropSource	OLEDROP1.CPP
COleDropTarget	OLEDROP1.CPP
COleException	OLEMISC.CPP
COleFrameHook	OLECLI2.CPP
COleIPFrameWnd	OLEIPFRM.CPP
COleInsertDialog	OLEDLGS1.CPP
COleLinkingDoc	OLELINK.CPP
COleLinksDialog	OLEDLGS1.CPP
COleMessageFilter	OLEMSGF.CPP
COleObjectFactory	OLEFACT.CPP

COlePasteSpecialDialog	OLEDLGS1.CPP
COlePropertiesDialog	OLEDLGS3.CPP
COlePropertyPage	CTLPPG.CPP
COleResizeBar	OLEBAR.CPP
COleServerDoc	OLESVR1.CPP
COleServerItem	OLESVR2.CPP
COleStreamFile	OLESTRM.CPP
COleTemplateServer	OLETSVR.CPP
COleUILinkInfo	OLEDLGS1.CPP
COleUpdateDialog	OLEDLGS1.CPP
COleVariant	OLEVAR.CPP
CPageSetupDialog	DLGPRNT.CPP
CParkingWnd	CTLREFL.CPP
CPaintDC	WINGDI.CPP
CPalette	WINGDI.CPP
CPen	WINGDI.CPP
CPictureHolder	CTLPICT.CPP
CPicturePropPage	PPGPICT.CPP
CPlex	PLEX.CPP
CPoint	100% inline
CPreviewDC	DCPREV.CPP
CPreviewView	VIEWPREV.CPP
CPrintDialog	DLGPRNT.CPP
CPrintInfo	VIEWPRNT.CPP
CPrintPreviewState	VIEWPREV.CPP
CProcessLocal	100% inline
CProgressCtrl	WINCTRL2.CPP
CProperty	OLEPSET.CPP
CPropertyPage	DLGPROP.CPP
CPropertySheet	DLGPROP.CPP
CPropertySet	OLEPSET.CPP
CPropExchange	CTLPROPX.CPP
CPropsetPropExchange	CTLPSET.CPP
CPtrArray	ARRAY_P.CPP
CPtrList	LIST_P.CPP
CRecentFileList	FILELIST.CPP
CRecordView	DBVIEW.CPP
CRecordset	DBCORE.CPP
CRecordsetStatus	100% inline
CRect	100% inline
CRectTracker	TRCKRECT.CPP
CReflectorWnd	CTLREFL.CPP
CResourceException	100% inline
CRgn	WINGDI.CPP
CRichEditCntrItem	VIEWRICH.CPP
CRichEditCtrl	WINCTRL4.CPP

CRichEditDoc	VIEWRICH.CPP
CRichEditView	VIEWRICH.CPP
CRuntimeClass	ARCCORE.CPP
CScrollBar	WINCTRL1.CPP
CScrollView	VIEWSCRL.CPP
CSemaphore	MTEX.CPP
CSharedFile	FILESHRD.CPP
CSimpleException	EXCEPT.CPP
CSimpleList	AFXTLS.CPP
CSingleDocTemplate	DOCSINGL.CPP
CSingleLock	MTEX.CPP
CSize	100% inline
CSliderCtrl	WINCTRL2.CPP
CSocket	SOCKCORE.CPP
CSocketFile	SOCKCORE.CPP
CSocketWnd	SOCKCORE.CPP
CSpinButtonCtrl	WINCTRL2.CPP
CSplitterWnd	WINSPLIT.CPP
CStatic	WINCTRL1.CPP
CStatusBar	BARSTAT.CPP
CStatusBarCtrl	WINCTRL2.CPP
CStdioFile	FILETXT.CPP
CStockPropPage	PPGSTOCK.CPP
CString	STRCORE.CPP, STREX.CPP
CStringArray	ARRAY_S.CPP
CStringList	LIST_S.CPP
CSyncObject	MTCORE.CPP
CTabCtrl	WINCTRL2.CPP
CThreadLocal	AFXTLS.CPP
CTime	TIMECORE.CPP
CTimeSpan	TIMECORE.CPP
CToolBar	BARTOOL.CPP
CToolBarCtrl	WINCTRL2.CPP
CToolInfo	TOOLTIP.CPP
CToolTipCtrl	TOOLTIP.CPP
CTreeCtrl	WINCTRL2.CPP
CTreeView	VIEWCMN.CPP
CTypeLibCache	OLETYPLB.CPP
CUIntArray	ARRAY_U.CPP
CUserException	EXCEPT.CPP
CView	VIEWCORE.CPP, VIEWPRNT.CPP
CWinApp	APPCORE.CPP, APPDLG.CPP, APPGRAY.CPP, APPHELPX.CPP, APPINIT.CPP, APPPRNT.CPP, APPUI.CPP, APPUI2.CPP, APPUI3.CPP
CWinThread	THRDCORE.CPP
CWindowDC	WINGDI.CPP
CWnd	WINCORE.CPP, TOOLTIP.CPP, WINFRMX.CPP, WINOCC.SPP,

	OCCCONT.CPP
CWordArray	ARRAY_W.CPP

MFC中隐藏了实现的头文件

在Mfc\Src目录下，有很多头文件，这些头文件都隐藏了一些实现细节。这些文件是：

- AFXIMPL.H——AUX_DATA，Macintosh字节交换辅助程序。
- BUILD_.H——MFC创建号（微软会使用）。
- COMMIMPL.H——通用控件实现辅助程序。
- CTLIMPL.H——包含了一些未记录的OLE控件类和一些辅助类，包括CControlFrameWnd、CReflectorWnd、CParkingWnd、CPropertySection、CPropertySet、CArchivePropExchange、CPropsetPropExchange、CStockPropPage、CColorButton、CColorPropPage和CPicturePropPage。
- DAOIMPL.H AFX_DAO_STATE。
- DBIMPL.H——AFX_ODBC_CALL、AFX_DB_STATE。
- ELEMENTS.H——CString的针对集合类和CPlex的辅助类。
- OCCIMPL.H——COleControlContainer、COleControlSite、CoccManager。
- OLEIMPL.H——OLE辅助类和状态对象。
- OLEIMPL2.H——COleFrameHook、CTypedLibCache、CEnumArray、COleDispatchImpl、OLE_DATA、_AFX_OLE_STATE
- SOCKIMPL.H——_AFX_SOCK_STATE。
- STDAFX.H——Master MFC include文件，这些文件根据预处理器的设置引入了所有其他的MFC includes。
- WINHAND_.H——CHandleMap的声明。

愉快的旅途

既然我们已经给你显示了MFC源代码，你应该开始准备自己探索。当然，你会发现很多本书中介绍的类，但是如果你不是自己探索，可能希望有一个方便的MFC源代码导读，以免在MFC丛林中迷路。

祝你在MFC探索的旅途上愉快！

APPENDIX

B 本书的示例代码

进入 www.infopower.com.cn ——→计算机专业技术与教材——→下载中心 ——→《深入解析 MFC》源代码并下载后，有一个 README.TXT 文件，这个文件详细地描述了所下载的内容。这些内容包括：

- 第 6 章的例子 Objfun 在 CHAP6 目录下。
- 第 11 章的例子 CoMath 在 CHAP11 目录下。这个例子以三种方式写出：使用多继承、使用嵌入类以及使用 MFC。
- 第 12 章的例子 DNDEDIT 在 CHAP12 目录下，这个例子是一个可以拖放的文本编辑器。
- 第 14 章的例子 MATHSRVR 在目录 CHAP14 下，这个例子用 MFC 实现了自动化。
- 你自己的 MFC FAQ 的拷贝（有 Word 格式的和 HTML 格式的）。
- 一些我们认为你可能会感兴趣的演示和产品说明。

你还可以到下面的 URL 那里查找本书中文本的更新版本和其他一些消息：
http://www.stingsoft.com/mfc_internal。

术 语 表

abstraction	抽象
accelerator	快捷键
aggregation	聚合
animation	动画
array	数组
assertion	断言
association	关系
automation	自动化
base class	基类
based point	基指针
bind	绑定
bitmap	位图
brush	画刷
call back	回调
class	类
class factory	类厂
clip-board	剪贴板
cluster	簇
collection	集合
collision	冲突
command-routing	命令路由选择
common control	通用控件
compiler	编译器
compound document	复合文档
Component Object Model	组件对象模型
connection	连接
connection point	连接点
constructor	构造函数
container	容器
containment	包容
content object	内容对象
context	环境
context-sensitive	上下文相关
control	控件
Control Developer Kit	控件开发包
C Run-time library	C 运行时库
Data Access Object	数据访问对象
data member	数据成员
deactivate	使无效
delegation	委托
dereference	反引用
derive	派生
derived class	派生类
destroy	析构
device context	设备环境
device driver	设备驱动
diagnostic	诊断
dialog	对话框
Dialog Data Exchange	对话框数据交换
Dialog Data Validation	对话框数据验证
disable	禁用
dispatch	分发
dispatch map	分发映射表
drag-and-drop	拖放
dockable	可停放的
document	文档
Document Template	文档模板
Double Byte CharacterSet	双字节字符集
dual interface	双接口
Dynamic Data Exchange	动态数据交换
Dynamic Link Library	动态链接库
dump	转储
early binding	早绑定
edit view	编辑视图

encapsulation	封装
entry	入口
enumeration	枚举
event sink	事件接收器
exception	异常
extraction operator	提取操作符（>>）
friend	友元
form view	窗体视图
general-purpose	通用
Graphics Device Interface	图形设备接口
handle	句柄
incoming interface	输入接口
inheritance	继承
inline	内嵌
In-place	定点
In-place activation	定点激活
insertion operator	插入操作符（<<）
interface	接口
internationalization	国际化
Inter-process Communication （IPC）	进程间通信
key	键
library	库
license key	许可密钥
list	链表
macro	宏
map	映射表
marshaling	列集
member function	成员函数
member variable	成员变量
memory management	内存管理
message-dispatch	消息分发
message handler	消息处理函数
message loop	消息循环
message Map	消息映射表
message pump	消息泵
metafile	元文件
MFC exception-handling macro	MFC 异常处理宏
Microsoft Foundation Class	Microsoft 基础类
modularity	模块化
moniker	名字对象
Microsoft Software Development Kit	Microsoft 软件开发工具包
Multiple Document Interface	多文档接口
multiple inheritance	多继承
negotiating	协商
notification	通知
object factory	对象厂
Object Linking and Embedding	对象链接与嵌入
object orientedness	面向对象
Object-Oriented Programming	面向对象编程
Open Database Connectivity	开放式数据库互连
operation system	操作系统
operator	操作符
override	覆盖
outgoing interface	输出接口
paintbrush	画笔
palette	调色板
pane	窗格
persistent storage	持续存储
pen	画笔
placeholder	占位符
private	私有的
property sheet	属性对话框
property page	属性页
protocol	协议
proxy	代理
pointer	指针
polymorphism	多态性
reference	引用
regular automation	规则自动化
Remote Procedure Call	远程过程调用
Run-Time Checking	运行时检查
Run-time class information	运行时类信息
script	脚本
scrolling view	滚动视图
scope	作用域
sector	扇区
serialization	序列化

server	服务器
Self-describing	自解释
Single Document Interface	单文档接口
snapshot	快照
socket	套接字
split	分割
splitter box	分割框
splitter bar	分割条
splitter intersection	分割交点
stack unwinding	堆栈展开
stream	流
strong typing	强类型
struct	结构
structured storage	结构化存储
storage	存储对象
stub	存根
substorage	子存储对象
Synchronization Object Class	同步对象类
timer	定时器
Uniform Data Transfer	统一数据传输
Uniform Nameing Convention	统一命名标准
union	联合
user-interface thread	UI 线程
utility	实用类
value	值
video clip	视频剪辑
view	视图
visual editing	可视化编辑
virtual function	虚函数
virtual function table	虚函数表
virtual member function	虚成员函数
virtual memory	虚拟内存
Windows Application Programming Interface	Windows 应用程序编程接口
Window class	窗口类
Window procedure	窗口过程
worker thread	辅助线程
wizard	向导