微软大中华区开发工具及平台事业部 总经理 谢恩伟 作序

微软及中科院专家
靳文波、覃涛、任伟 联袂推荐

Visual C++ 2010 开发权威指南

尹成 颜成钢 编著

- 全面阐述Visual C++ 2010开发技术
- 全新体验Windows 7平台上炫彩开发技术
- 主流的开发案例

Windows 7的超级任务栏开发
Windows 7的传感器应用开发
多点触摸交互操作应用开发
语音识别与语音合成应用开发
笔迹文档、手写识别应用开发
信息安全加密、解密应用开发

全程30小时视频讲解

200多个范例源程序
300多页的CLR开发电子书

人民邮电出版社
POSTS & TELECOM PRESS

尹成

中科院硕士，Windows平台资深软件构架师，CSDN著名技术专家，CSDN学生大本营著名C/C++老师，CSDN知名IT作家协会会员。微软社区技术精英之一，微软C/C++ 特邀嘉宾，Intel软件技术专家。精通C/C++，对语音技术、信息安全、移动3G开发均有较深入的研究。博客交流地址：http://blog.csdn.net/yincheng01，MSN：yincheng01@163.com。

颜成钢

中科院博士，微软中科院大使，中科院微软技术俱乐部会长。参与多个国家级科研项目开发，具备丰富的C++编程经验，图形图像技术专家。2008年参加国际权威的视频检索评测TRECVID的ED比赛，在全球多个参赛队中，取得多项搜索任务第一名。联系邮箱： yanchenggang@ict.ac.cn。

分类建议：计算机／程序设计／Visual C++

人民邮电出版社网址：www.ptpress.com.cn

封面设计：任文杰

ISBN 978-7-115-23077-5

ISBN 978-7-115-23077-5

定价：99.00 元（附光盘）

Visual C++ 2010 开发权威指南

■ 尹成 颜成钢 编著

人民邮电出版社
北京

图书在版编目（CIP）数据

Visual C++2010开发权威指南 / 尹成，颜成钢编著
. -- 北京 : 人民邮电出版社，2010.8
ISBN 978-7-115-23077-5

Ⅰ. ①V… Ⅱ. ①尹… ②颜… Ⅲ. ①C语言－程序设计－指南 Ⅳ. ①TP312-62

中国版本图书馆CIP数据核字(2010)第088155号

内 容 提 要

微软公司新一代的开发工具 Visual C++ 2010 在 C++开发方面带来了很多革命性的变化。新的操作系统 Windows 7 也已经问世，Windows 7 的豪华界面与易用性、兼容性，势必会成为操作系统的主流。

本书全面阐述了 Visual C++ 2010 开发技术，通过实例让读者全新体验 Windows 7 平台上炫彩开发技术。书中包括许多主流的开发案例，如多点触控开发、语音技术开发、手写技术开发、信息安全技术等。

全书共 21 章，分为 5 部分。第一部分介绍了 Visual C++ 2010 开发环境与新特性；第二部分指引读者逐步学会在 Visual C++ 2010 下进行 MFC 开发，以及用实例引导读者学习 MFC 新特性的应用；第三部分介绍了 Visual C++ 2010 下 MFC 与 CLR 进行项目开发；第四部分发布 Visual C++ 2010 程序；第五部分讲解 Windows 7 新特性开发。

随书配套光盘中有 30 小时以上的开发视频，附带 200 多个范例源程序，以便帮助读者尽快学会用 Visual C++ 2010 开发。

Visual C++ 2010 开发权威指南

◆ 编　　著　尹　成　颜成钢
责任编辑　张　涛

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京隆昌伟业印刷有限公司印刷

◆ 开本：787×1092　1/16
印张：51.5
字数：2067 千字　　2010 年 8 月第 1 版
印数：1－3 000 册　　2010 年 8 月北京第 1 次印刷

ISBN 978-7-115-23077-5

定价：99.00 元（附光盘）

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154

拥抱旧时的美好，迎向崭新的机会

「微软会不会就此放弃 C++呢？」

在社区上或研讨会活动中，朋友们常常问我们这个问题。我被大家急切的心情所感动，这是 C++真正的老朋友，极度关心他们所钟爱的技术是否将被遗落在主流趋势之外。

我们认知到一个明显的情形：科技的不断进步刺激着现实世界的需求，现实世界的需求推动架构技术的演进，而架构技术的演进必然会使开发技术更趋复杂化，但人们不希望他们所熟悉的工具与技术有太过激烈的改变。大家渴望有新东西，但又不希望新东西的变化太过剧烈，因为旧东西一时之间无法丢弃。偏偏不再创新的东西，又会被其他的创新所超越，这就是 IT 行业的达尔文主义。

就全球大趋势来说，Managed Code 无疑是当前主流，.NET 与 Java 在此方面占据了一大部分的天下。但是，在中国，始终有约 20%~30%的开发者，继续坚持在 C++开发的岗位上，已经成为了独特的中国特色之一。这些 C++的开发者，特别是在游戏开发、高科技制造业，以及新兴的移动设备应用与嵌入式设备产业方面。尤其是后两者的蓬勃发展，在云计算应用越来越大行其道的当下，其重要性是显而易见的。另一方面，也随着 Windows 7 操作系统的发布，吸引了桌面应用软件开发者想要做出功能更强的新版本软件，这些都为所谓「C++的复兴」营造了良好的基础。

因此，微软的开发工具事业部在上海的研发团队，有一个 C++产品组，全力以赴开发更好的工具与平台，让中国的 C++开发者能够与时俱进，这些努力的成果都体现在 Microsoft Visual Studio 2010 上，我们非常荣幸地与您分享这些创新，更感谢尹成与颜成钢两位老师，倾全力在 Visual Studio 2010 发布的第一时间，就能够将中文版的 Visual C++ 2010 技术书籍付梓，让各位能够与全球技术同步零时差，这点是相当了不起的。这本书不但抢得先机，且绝对不是一本简单的书。本书的内容涵盖了 MFC 与.NET Framework CLR 两种平台的应用方式，无论您是注重程序执行的效能，还是程序开发的速度，这本书都能够满足您的不同需求。除了涵盖程序开发的接口端、通讯端、并行计算以及数据库应用，本书还涵盖了许多相当实用的部份，例如代码与程序的安全性、软件的国际化。总而言之，对于 C++的学习者而言，这本书当然有其价值，就算是资深的 C++开发者，本书也是一本好的参考书，能够全面地提升开发者的素质。

Visual C++ 2010 并不只是版本数字的往上增加，我们并不希望把 C++带入到一个更艰深、更高不可攀的方向。微软是擅长做工具的公司，面对新技术的挑战，微软的想法是，开发人员不需要去写越来越多行的程序代码，读越来越厚的文件，费力去摆弄越来越庞大的 AP Server，而是应该专注开发越来越好用的工具与平台，快速实现新技术并形成好的商业竞争力。这次 Visual C++ 2010 的创新，有相当多的部份体现在工具与语言支持上，希望能让您体会到微软的用心，获得更高的开发效率。

最后，感谢您购买这本书，国内的专业图书出版市场需要您的大力支持才会有更多的作者，像尹成与颜成钢两位老师一样，充满热忱地为各位引领技术的方向。祝愿各位读者有一个良好的学习旅程，并能快速掌握 Visual C++ 2010 的创新秘密，征服不可能的任务。

微软大中华区开发工具
及平台事业部总经理　谢恩伟

前　言

2010 年 4 月 12 日在全球 5 个城市：北京、拉斯维加斯、伦敦、班加罗尔和吉隆坡同时发布了 Visual Studio 2010。由于时差关系，北京成为了绝对时间上第一个发布此软件的城市，微软举办了千人发布会共同见证这一时刻。Visual Studio 2010 中有一大部分的技术实现是在上海微软中国研发中心完成的。发布当天张亚勤博士、微软总部的多位研发团队的资深主管、还有上海研发中心的一线产品开发人员为大家深度解密了 Visual Studio 2010 的诞生过程。

IT 界是一个创造奇迹的地方。微软的最新开发工具 Visual Studio 2010 已经诞生，伴随着最新的操作系统 Windows 7 的问世，Visual C++开发即将旋起新的浪潮。在当今.NET 平台已经流行的今天，Visual C++也能基于.NET 平台开发，MFC 被灌注新的特性，将会为程序员开发带来更加高效的代码生产力，带来更多新奇与便捷。

Visual C++ 2010 具备编译器多核优化等新特性。在 Visual Studio 2010 发布会上，就像微软大中华区产品市场经理胡德民先生所说："微软将给 C++开发者更多的关怀和支持。从 Visual Studio 2010 开始，不管是从 IDE 到编译器，C++开发者都能在这个工具中重新找回自己应该在的位置。"确实是这样的，微软新一代的开发工具 Visual Studio 2010 在 C++开发方面也带来了很多革命性的变化。新一代操作系统 Windows 7 的豪华界面与易用性、兼容性，势必成为操作系统的主流，Visual C++ 2010 也必将成为未来 Windows 7 平台下开发的主流语言之一。在 Visual Studio 2010 中，使得非托管 C++代码的开发更加容易和高效，例如，升级了 MFC 的库和头文件以全面支持 Windows 7 的界面元素，包括 Ribbon 界面、搜索功能、甚至多点触摸特性的支持。想让应用程序"Windows 7 Ready"，Visual Studio 2010 是首选。Visual Studio 2010 加上 Windows SDK for Windows 7 是开发 Windows 7 应用程序绝佳平台。

在当今时代，新技术不断兼容旧技术，取代旧技术，但是 C++作为系统平台级的语言，依旧是永不过时的。本书是第一本原创 Visual C++ 2010 的技术专业书，全方位地介绍了如何运用 Visual Studio 2010 开发 C++程序与 Windows 7 应用，本书用 200 多个原创的 Visual C++ 2010 精品范例，深入浅出地为读者展示了 Visual C++ 2010 的强大功能，指引读者学以致用地学习应用 Visual C++ 2010，力争帮助读者用最短的时间学会用 Visual C++ 2010 开发，以便高效地开发自己的产品。

全书共 21 章，分为 5 部分。

第一部分：介绍 Visual C++ 2010 开发环境与新特性。

第二部分：指引读者逐步学会在 Visual C++ 2010 下进行 MFC 开发，以及学习使用 MFC 提供的新特性。

第三部分：指引读者逐步学会在 Visual C++ 2010 下进行 CLR 和 MFC 开发。

第四部分：发布 Visual C++ 2010 的应用程序。

第五部分：Windows 7 平台下用 Visual C++ 2010 进行新特性开发。

感谢微软大中华区开发工具及平台事业部总经理谢恩伟为本书作序，感谢微软中国研发集团服务器及开发工具事业部总经理兼 Visual Studio Business Appliacations 总经理潘正磊、Visual C++团队项目经理 Raman Sharma 的帮助和支持，感谢 Visual C++ Roundtable 的诸位嘉宾（以下排名不分先后）焦桢、马骐、李争、高阳、陈良乔、衣明志、刘佳、荆明杰、蒋晟的指点，感谢 IT168 胡铭亚的鼎力支持。

感谢母校山东大学及培养我们的老师。

特别感谢中国科学院计算技术研究所李锦涛博士和张勇东研究员的支持与帮助。

感谢 CSDN 总裁蒋涛以及他率领的 CSDN 团队，CSDN 博客编辑陈娟、牛海彬、刘洪峰、张大志，程序员杂志编辑黄长著、张悦校、曾登高、陈洋彬、刘江、董世晓、常政、田民丽、高松、孟迎霞、郑柯、郭晓刚等提供的支持。

感谢微软社区精英计划工作人员廖浩志、班倩超、史义、朱清云的鼎力支持。

感谢安徽中科大讯飞信息科技股份有限公司嵌入式事业部经理黄海兵与副经理高军、研发总监谢信珍，感谢艾林威逊（北京）科技有限公司董事长李涛与总经理梁亚楠的鼎力支持，感谢英唐公司总经理胡庆周与英唐数码总经理秦玉香的鼎力支持。

感谢博客园（中国编程技术社区 www.cnblogs.com）站长 DUDU（杜勇），沪江网（专业外语网站 www.yeshj.com）站长阿诺（伏彩瑞）的支持。

感谢张晓波、互联网策划大师尹传喜、指纹安全专家邵宇，在百忙之中提供的帮助与支持。感谢北京亚略特诚信科技有限公司的总裁李强、研发总监王乐以及他带领的精英技术团队的鼎力支持。

本书 DVD 光盘配有 30 多个小时的开发视频，超过 200 个实用 Visual C++ 2010 范例。赠送 360 多页的 Visual C++ 2010 CLR 开发电子书。

本书由尹成、颜成钢主编，参与编写的还有陈荔城、陈兴、鹿晓文、刘武、崔雅超、逯刚刚、夏成、马广浩、胡琼、谢洪涛、张雷刚、陈南南、宋砚、俞力克、胡耀文、佟玲玲、张迎平、刘毅、钱厚荣、于海英、刘欣、王军昌。

尹成，中科院硕士，Windows 平台资深软件构架师，CSDN 著名技术专家，CSDN 学生大本营著名 C/C++老师，CSDN 知名 IT 作家协会会员。微软社区技术精英之一，微软 C/C++ 特邀嘉宾，Intel 软件技术专家。精通 C/C++，对语音技术、信息安全、移动 3G 开发均有深入的研究。

颜成钢，中科院博士，微软中科院大使，中科院微软技术俱乐部会长。参与多个国家级科研项目开发，具备丰富的 C++编程经验，图形图像技术专家。2008 年参加国际权威的视频检索评测 TRECVID 的 ED 比赛，在全球多个参赛队中，取得多项搜索任务第一名。联系邮箱为: yanchenggang@ict.ac.cn 。

由于时间仓促及水平有限，书中难免存有不妥之处，恳请广大读者批评指正，以便及时修正。 技术交流博客地址为 http://blog.csdn.net/yincheng01，MSN 为 yincheng01@163.com，欢迎进行技术交流。编辑联系邮箱为：zhangtao@ptpress.com.cn。

编者

目　录

第一部分　Visual C++ 2010 开发与新特性

第 1 章　Visual C++ 2010 开发环境简介 …… 1

1.1　Visual C++ 2010 简介 …… 1
1.2　Visual C++ 2010 下载安装指南 …… 1
1.3　Visual C++ 2010 主要特点与新特性 …… 3
1.4　Visual C++ 2010 开发环境操作指南 …… 6
1.4.1　创建 Visual C++应用程序 …… 6
1.4.2　Visual C++ 2010 菜单介绍 …… 9
1.5　Visual C++ 2010 MFC 简介 …… 12
1.6　Visual C++ 2010 CLR 简介 …… 13
1.7　Visual C++ 2010 64 位编程 …… 14
1.8　支持新的 C++语言标准 …… 14
1.8.1　支持新的 C++语言标准（C++ 0x） …… 14
1.8.2　Lambda 表达式 …… 15
1.8.3　静态断言 static_assert …… 17
1.8.4　auto 关键字 …… 18
1.8.5　右值引用 …… 19
1.8.6　安全数据类型 …… 22
1.8.7　移动构造 …… 23
1.9　支持开发并行程序 …… 26
1.9.1　运行库支持 native 代码 …… 26
1.9.2　调试和分析工具 …… 29
1.10　对 MFC 的增强 …… 31
1.10.1　任务对话框 CTaskDialog …… 31
1.10.2　重启管理器（Restart Manager）支持 …… 33
1.10.3　支持新的用户界面风格 …… 34

第 2 章　MFC 应用程序框架 …… 37

2.1　用 MFC 向导生成应用程序 …… 37
2.1.1　生成的程序框架 …… 37
2.1.2　生成的应用程序类型 …… 43
2.1.3　向导生成工程文件简介 …… 46
2.2　应用程序框架分析 …… 46
2.2.1　框架简介 …… 47
2.2.2　MFC 应用程序运行流程 …… 50
2.2.3　应用程序窗口 …… 51
2.2.4　消息机制与消息循环 …… 59
2.3　文档与视图 …… 65
2.3.1　文档类结构分析 …… 65
2.3.2　视图类结构分析 …… 65
2.3.3　文档与视图的关系 …… 67
2.3.4　SDI 应用程序分析 …… 69
2.3.5　MDI 应用程序分析 …… 69
2.4　工具栏与状态栏 …… 69
2.4.1　工具栏 …… 70
2.4.2　状态栏 …… 78
2.5　MFC 新功能——Office 2007 风格程序分析 …… 85
2.6　MFC 新功能——Visual Studio 风格程序分析 …… 90
2.7　MFC 新功能——Windows 资源管理器风格程序分析 …… 98
2.8　应用程序框架类对象分析 …… 100
2.8.1　视图类对文档类的调用 …… 100
2.8.2　在框架类中获得当前的文档类和视图类对象指针 …… 101
2.8.3　获得应用程序类对象的指针 …… 101
2.8.4　从应用类对象中获得主框架类对象的指针 …… 101
2.9　小结 …… 102

第 3 章　Visual C++ 2010 MFC 菜单编程 … 103

3.1　菜单编程 …… 103
3.1.1　创建菜单 …… 104
3.1.2　创建菜单热键 …… 115

3.1.3 标记菜单 …… 115
3.1.4 给菜单加入图标 …… 119
3.1.5 禁用菜单 …… 121
3.1.6 移除与加载菜单 …… 127
3.2 菜单消息的传输机制 …… 129
3.2.1 菜单消息的分类 …… 129
3.2.2 菜单消息的传输路由 …… 129
3.3 动态菜单操作 …… 131
3.3.1 添加菜单 …… 131
3.3.2 插入菜单 …… 133
3.3.3 删除菜单 …… 135
3.3.4 插入菜单的命令响应 …… 136
3.3.5 修改菜单 …… 137
3.4 小结 …… 140

第 4 章 Visual C++ 2010 MFC 对话框编程 …… 141

4.1 对话框简介 …… 141
4.1.1 对话框的控件简介 …… 141
4.1.2 对话框的种类简介 …… 149
4.1.3 设计对话框 …… 150
4.2 创建与销毁对话框 …… 153
4.2.1 模态对话框 …… 153
4.2.2 非模式对话框 …… 159
4.2.3 属性页对话框 …… 163
4.3 消息对话框 …… 173
4.4 通用对话框 …… 175
4.4.1 文件打开对话框 …… 176
4.4.2 文件保存对话框 …… 178
4.4.3 颜色对话框 …… 179
4.4.4 字体对话框 …… 181
4.4.5 查找对话框 …… 183
4.4.6 页面设置对话框 …… 185
4.4.7 打印对话框 …… 186
4.5 小结 …… 187

第 5 章 Visual C++ 2010 MFC 对话框控件 …… 188

5.1 Visual C++ 2010 Button 控件简介与开发 …… 191
5.2 Visual C++ 2010 List Box 控件简介与开发 …… 193
5.3 Visual C++ 2010 Com boBox 控件简介与开发 …… 195
5.3.1 创建扩展组合框控件 …… 196
5.3.2 在扩展组合框控件中使用图像列表 …… 197
5.3.3 设置各项的图像 …… 197
5.3.4 处理扩展组合框控件中的通知消息 …… 198
5.4 Visual C++ 2010 List 控件简介与开发 …… 198
5.4.1 列表控件和列表视图 …… 199
5.4.2 列表项和图像列表 …… 199
5.4.3 回调项和回调屏蔽 …… 200
5.4.4 创建列表控件 …… 200
5.4.5 创建图像列表 …… 201
5.4.6 向控件添加列（报表视图） …… 204
5.4.7 向控件添加项 …… 205
5.4.8 在列表控件中滚动、排列、排序和查找 …… 205
5.4.9 在列表控件中实现工作区 …… 205
5.4.10 处理列表控件中的通知消息 …… 206
5.4.11 更改列表控件样式 …… 206
5.4.12 虚拟列表控件 …… 207
5.4.13 列表控件的消息映射 …… 209
5.4.14 列表控件的风格选项及表头设置 …… 210
5.4.15 销毁列表控件 …… 210
5.5 Visual C++ 2010 Edit 控件简介与开发 …… 211
5.6 Visual C++ 2010 Rich Edit 控件简介与开发 …… 213
5.6.1 Rich Edit 控件中的字符格式 …… 215
5.6.2 Rich Edit 控件中的段落格式 …… 215
5.6.3 Rich Edit 控件中的当前选定内容 …… 215
5.6.4 Rich Edit 控件中的分词 …… 216

5.6.5 Rich Edit 控件中的剪贴板操作 …… 216
5.6.6 Rich Edit 控件中的流操作 …… 216
5.6.7 Rich Edit 控件中的打印操作 …… 216
5.6.8 无底的 Rich Edit 控件 …… 217
5.6.9 来自 Rich Edit 控件的通知 …… 217
5.7 Visual C++ 2010 Progress 控件简介与开发 …… 219
5.7.1 进度控件的样式 …… 219
5.7.2 进度控件的设置 …… 219
5.7.3 操作进度控件 …… 220
5.8 Visual C++ 2010 Tree 控件简介与开发 …… 220
5.8.1 树控件样式 …… 221
5.8.2 树控件父项和子项 …… 221
5.8.3 树控件项位置 …… 222
5.8.4 树控件项标签 …… 222
5.8.5 树控件标签编辑 …… 223
5.8.6 树控件项的状态 …… 223
5.8.7 树控件图像列表 …… 224
5.8.8 树控件项选择 …… 224
5.8.9 树控件拖放操作 …… 224
5.8.10 树控件项信息 …… 225
5.8.11 树控件通知消息 …… 225
5.9 Visual C++ 2010 DBgrid 控件简介与开发 …… 226
5.9.1 示例程序 1 …… 226
5.9.2 示例程序 2 …… 227
5.10 Visual C++ 2010 Rebar 控件简介与开发 …… 228
5.10.1 在 Rebar 控件中使用图像列表 …… 230
5.10.2 在 Rebar 控件中使用对话栏 …… 231
5.10.3 处理 Rebar 控件中的通知消息 …… 231
5.11 Visual C++ 2010 Timer 控件简介与开发 …… 232
5.12 Visual C++ 2010 Tab 控件简介与开发 …… 234
5.12.1 选项卡和选项卡控件属性 …… 235
5.12.2 选项卡控件的使用方法 …… 235
5.12.3 创建选项卡控件的方法 …… 235
5.12.4 处理选项卡控件通知消息 …… 236
5.12.5 CTabCtrl 类简介 …… 236
5.13 Visual C++ 2010 IP 控件简介与开发 …… 241
5.14 Visual C++ 2010 Picture 控件简介与开发 …… 241
5.15 Visual C++ 2010 Slider 控件简介与开发 …… 244
5.15.1 滑块控件样式 …… 244
5.15.2 滑块控件成员函数 …… 245
5.15.3 滑块控件通知消息 …… 246
5.16 Visual C++ 2010 Scroll Bar 控件简介与开发 …… 246
5.17 Visual C++ 2010 Hot Key 控件简介与开发 …… 248
5.17.1 使用热键控件 …… 248
5.17.2 设置热键 …… 249
5.18 Visual C++ 2010 Animation 控件简介与开发 …… 249
5.18.1 使用动画控件 …… 249
5.18.2 动画控件发送的通知 …… 250
5.19 Visual C++ 2010 Spin 控件简介与开发 …… 250
5.19.1 数值调节钮的样式 …… 250
5.19.2 数值调节钮成员函数 …… 251
5.20 Visual C++ 2010 GroupBox 控件简介与开发 …… 251
5.21 Visual C++ 2010 Data Time Picker 控件简介与开发 …… 252
5.21.1 创建日期和时间选择器控件 …… 253
5.21.2 访问嵌入的月历控件 …… 253
5.21.3 在日期和时间选择器控件中使用自定义格式字符串 …… 254

5.21.4 在日期和时间选择器控件中使用回调字段 …… 254
5.21.5 处理日期和时间选择器控件中的通知消息 …… 256
5.22 Visual C++ 2010 Month Canlendar 控件简介与开发 …… 256
5.22.1 创建月历控件 …… 257
5.22.2 处理月历控件中的通知消息 …… 257
5.22.3 设置月历控件的日状态 …… 257
5.23 Visual C++ 2010 Custom 控件简介与开发 …… 258
5.23.1 使用 MFC 方法定制控件必备的几个基本概念 …… 259
5.23.2 定制自定义控件的 3 种常见方法 …… 260
5.24 Visual C++ 2010 SysLink 控件简介与开发 …… 260
5.25 Visual C++ 2010 Split Button 控件简介与开发 …… 261
5.26 Visual C++ 2010 Network Address 控件简介与开发 …… 262
5.27 Visual C++ 2010 Check Box 控件简介与开发 …… 262
5.28 Visual C++ 2010 Radio Button 控件简介与开发 …… 264
5.28.1 为单选按钮控件分组 …… 264
5.28.2 获得被选中的单选按钮的文本 …… 264
5.29 Visual C++ 2010 MediaPlayer 控件简介与开发 …… 265
5.30 小结 …… 266

第二部分 Visual C++ 2010 下 MFC 开发

第 6 章 计算机测控系统概述 …… 267

6.1 Visual C++ 2010 SDI 开发简介 …… 267
6.1.1 建立应用程序基本框架 …… 267
6.1.2 处理视图 …… 267
6.1.3 处理文档 …… 271
6.1.4 串行化处理 …… 274
6.1.5 SDI 应用程序编程思路 …… 275
6.2 Visual C++ 2010 MDI 开发简介 …… 277
6.2.1 多文档接口 …… 277
6.2.2 生成程序 …… 278
6.2.3 程序类、文件和代码 …… 279
6.2.4 自定义资源 …… 281
6.3 Visual C++ 2010 View 开发 …… 282
6.3.1 生成源文件 …… 283
6.3.2 初始化视图类数据成员 …… 283
6.3.3 加入消息处理功能 …… 285
6.3.4 设计程序资源 …… 290
6.3.5 定制 MiniDraw 窗口 …… 292
6.3.6 程序清单 …… 293
6.4 Visual C++ 2010 EditView 开发 …… 298
6.4.1 生成 MiniEdit 程序 …… 299
6.4.2 修改程序菜单 …… 300
6.4.3 编辑加速键 …… 301
6.4.4 程序清单 …… 303
6.5 Visual C++ 2010 FormView 开发 …… 306
6.5.1 自定义 FormDemo 程序 …… 307
6.5.2 程序清单 …… 314
6.6 Visual C++ 2010 ScrollView 开发 …… 319
6.6.1 加入滚动功能 …… 319
6.6.2 坐标换算 …… 319
6.6.3 限制图形大小 …… 322
6.6.4 改变鼠标光标 …… 325
6.7 Visual C++ 2010 HtmlEditView 开发 …… 328
6.8 Visual C++ 2010 HtmlView 开发 …… 331
6.9 Visual C++ 2010 ListView 开发 …… 335
6.10 Visual C++ 2010 RichEditView 开发 …… 335
6.11 VisualC++ 2010 TreeView 开发 …… 336
6.12 Visual C++ 2010 Office 2007 风格文档视图开发框架 …… 337

6.13 Visual C++ 2010 Visual Studio 2008 风格文档视图开发框架 …… 342
6.14 Visual C++ 2010 Windows 资源管理器风格文档视图开发框架 …… 346
6.15 小结 …… 350

第 7 章 Visual C++ 2010 MFC 应用程序界面与美化 …… 351

7.1 应用程序窗口风格美化 …… 351
7.1.1 借助 ActiveSkin 美化窗口 …… 351
7.1.2 修改窗口外观 …… 352
7.2 应用程序窗口图标与背景修改 …… 359
7.2.1 修改窗口图标 …… 359
7.2.2 修改背景 …… 360
7.3 工具栏编程与美化 …… 365
7.3.1 创建工具栏 …… 365
7.3.2 在工具栏中添加、删除按钮 …… 366
7.3.3 从对话框创建工具栏 …… 368
7.4 状态栏编程与美化 …… 370
7.4.1 创建状态栏 …… 370
7.4.2 在状态栏中插入进度条 …… 370
7.5 鼠标光标编程 …… 371
7.5.1 鼠标光标编程步骤 …… 371
7.5.2 鼠标的消息处理机制 …… 373
7.5.3 示例 …… 374
7.6 创建启动界面 …… 376
7.7 创建特效窗口启动应用程序 …… 378
7.8 创建特效窗口关闭应用程序 …… 378
7.9 小结 …… 383

第 8 章 Visual C++ 2010 MFC 文本与字体 …… 384

8.1 CFont 字体类简介 …… 384
8.1.1 CFont 字体类成员介绍 …… 384
8.1.2 CFont 字体类初始化函数 …… 385
8.1.3 其他成员介绍 …… 390
8.2 创建文本插入符与图片插入符 …… 391
8.2.1 创建文本插入符 …… 391
8.2.2 创建图片插入符 …… 394
8.2.3 创建随鼠标移动的插入符 …… 396
8.3 输出文字与字体格式 …… 397
8.3.1 输出固定文字 …… 397
8.3.2 设定输出字体的格式 …… 398
8.3.3 字符输入 …… 399
8.4 输出彩色文字与变色文字 …… 404
8.4.1 DrawText()函数和字符串资源 …… 404
8.4.2 定时器和变色文字 …… 408
8.5 小结 …… 410

第 9 章 Visual C++ 2010 MFC 图形图像编程 …… 411

9.1 Windows 绘图简介 …… 411
9.1.1 设备描述表 …… 411
9.1.2 绘图属性 …… 412
9.1.3 元文件和路径 …… 412
9.1.4 颜色和调色板 …… 412
9.1.5 图形设备接口函数 …… 413
9.2 Windows 屏幕绘图简介 …… 414
9.2.1 窗口客户区 …… 414
9.2.2 映射模式 …… 414
9.2.3 图形刷新 …… 416
9.3 微软 GDI 绘图简介 …… 416
9.3.1 GDI 基础 …… 416
9.3.2 GDI 结构 …… 417
9.3.3 GDI 函数调用 …… 417
9.3.4 GDI 基本图形 …… 418
9.4 GDI 笔绘图 …… 419
9.4.1 Cpen 类简介 …… 419
9.4.2 使用 GDI 绘制线条 …… 419
9.4.3 使用 CPen 类绘制指定的线条 …… 422
9.4.4 绘制连续的线条 …… 424
9.5 GDI 画刷绘图 …… 425
9.5.1 CBrush 类介绍 …… 426
9.5.2 CBrush 类简单画刷的实现 …… 429
9.5.3 CBrush 类位图画刷的实现 …… 430
9.5.4 透明画刷的实现 …… 431
9.6 小结 …… 433

第 10 章 Visual C++ 2010 MFC 动态函数链接库 ····· 434

10.1 动态函数链接库简介 ····· 434
10.1.1 什么是动态函数链接库 ····· 434
10.1.2 动态函数链接库的优点 ····· 435
10.1.3 动态函数链接库的起源 ····· 436
10.1.4 动态函数链接库的原理 ····· 436
10.2 调用动态函数链接库 ····· 436
10.2.1 静态链接 ····· 436
10.2.2 动态链接 ····· 438
10.3 DLL 的框架简介 ····· 439
10.3.1 DllMain()函数简介 ····· 439
10.3.2 DLL 的导出函数 ····· 439
10.4 创建 MFC DLL 范例 ····· 440
10.4.1 建立 MFC DLL 工程 ····· 440
10.4.2 添加实现代码 ····· 442
10.4.3 编译并调用 ····· 443
10.5 创建 Win32 DLL 范例 ····· 443
10.5.1 建立 Win32 工程 ····· 443
10.5.2 添加动态链接库代码 ····· 444
10.5.3 编译工程 ····· 444
10.6 创建资源 DLL 范例 ····· 445
10.6.1 建立 MFC Application 工程 ····· 445
10.6.2 建立中文资源 DLL ····· 445
10.6.3 加载资源 DLL ····· 446
10.7 Hook 技术 ····· 446
10.7.1 Hook 函数类型 ····· 446
10.7.2 使用 Hook 函数 ····· 449
10.7.3 Hook 鼠标 ····· 449
10.7.4 Hook 键盘 ····· 450
10.8 小结 ····· 452

第 11 章 Visual C++ 2010 MFC ActiveX 控件 ····· 454

11.1 ActiveX 控件简介 ····· 454
11.2 ActiveX 控件测试与注册 ····· 455
11.2.1 ActiveX 控件的测试 ····· 455
11.2.2 ActiveX 控件的注册 ····· 457
11.3 MFC ActiveX 控件向导 ····· 458
11.4 ActiveX 控件属性开发 ····· 458
11.4.1 添加常用属性 ····· 459
11.4.2 添加自定义属性 ····· 460
11.4.3 高级属性实现 ····· 460
11.4.4 访问环境属性 ····· 461
11.5 ActiveX 控件事件开发 ····· 461
11.5.1 添加常用事件 ····· 462
11.5.2 添加自定义事件 ····· 463
11.6 ActiveX 控件方法开发 ····· 464
11.6.1 添加常用方法 ····· 465
11.6.2 添加自定义方法 ····· 465
11.6.3 从方法返回错误代码 ····· 466
11.7 完整 ActiveX 控件范例 ····· 467
11.7.1 创建工程 ····· 467
11.7.2 Clock 控件的实现 ····· 469
11.7.3 添加常用属性 ····· 470
11.7.4 添加自定义属性 ····· 473
11.7.5 添加方法 ····· 474
11.7.6 添加常用事件 ····· 475
11.7.7 添加自定义事件 ····· 476
11.8 调用 ActiveX 控件 ····· 477
11.9 小结 ····· 478

第 12 章 Visual C++ 2010 MFC 文件与注册表操作 ····· 479

12.1 文本操作串行化 ····· 479
12.1.1 文档类 Serialize()函数 ····· 479
12.1.2 CArchive 对文件进行读写 ····· 482
12.1.3 文档操作串行化代码分析 ····· 485
12.2 CFile 类 ····· 492
12.2.1 打开文件操作 ····· 493
12.2.2 读写文件操作 ····· 494
12.2.3 定位文件操作 ····· 496
12.2.4 关闭文件操作 ····· 497
12.2.5 异常操作 ····· 497
12.2.6 文件管理操作 ····· 498
12.3 .ini 文件读写操作 ····· 500
12.4 注册表读写操作 ····· 502
12.4.1 注册表简介 ····· 502

12.4.2 注册表 API …… 504
12.4.3 访问并修改注册表 …… 507
12.5 小结 …… 509

第 13 章 Visual C++ 2010 MFC 数据库开发 …… 510

13.1 数据库基本知识 …… 510
13.2 SQL 语言的基础知识 …… 511
13.3 ODBC 访问数据库 …… 512
13.3.1 注册 ODBC 数据库 …… 512
13.3.2 创建一个 MFC 的 ODBC 程序 …… 514
13.3.3 程序结构分析 …… 515
13.3.4 在视图上显示数据库查询结果 …… 520
13.3.5 对查询结果排序及设置查询条件 …… 524
13.3.6 动态设置查询条件并更新查询结果 …… 527
13.4 ODBC 更新数据库 …… 532
13.5 ODBC 访问 SQL Server …… 540
13.6 ADO 数据库访问 …… 543
13.6.1 ADO 数据库访问概述 …… 543
13.6.2 在 Visual C++中使用 ADO 编程 …… 546
13.7 ADO 访问 SQL Server 数据库 …… 556
13.8 小结 …… 556

第 14 章 Visual C++2010 MFC 多线程程序设计 …… 557

14.1 进程和多线程的概念 …… 557
14.2 线程的创建 …… 558
14.2.1 创建工作者线程 …… 558
14.2.2 创建用户界面线程 …… 559
14.3 线程的终止 …… 560
14.4 设置线程的优先级 …… 562
14.5 暂停及重新启动线程 …… 563
14.6 线程间的通信 …… 571
14.7 线程的同步 …… 572
14.7.1 临界区 …… 572
14.7.2 互斥量 …… 573
14.7.3 事件 …… 573
14.7.4 信号量 …… 574
14.8 小结 …… 579

第 15 章 Visual C++ 2010 MFC 网络程序设计 …… 580

15.1 计算机网络的基础知识 …… 580
15.1.1 TCP/IP 协议模型 …… 580
15.1.2 IP 地址 …… 582
15.1.3 端口 …… 582
15.1.4 数据封装 …… 582
15.2 Winsock 简介 …… 583
15.3 MFC 对 Windows Sockets 的支持 …… 583
15.3.1 Socket 的定义 …… 584
15.3.2 CAsyncSocket 类介绍 …… 584
15.3.3 CSocket 类介绍 …… 592
15.4 一个基于 UDP 的聊天室示例 …… 593
15.4.1 MFC 对 Windows Sockets 的初始化 …… 593
15.4.2 服务器端的实现 …… 595
15.4.3 客户端的实现 …… 599
15.5 一个基于 TCP 的聊天室示例 …… 605
15.5.1 服务器端的实现 …… 606
15.5.2 客户端的实现 …… 609
15.6 小结 …… 612

第 16 章 Visual C++ 2010 MFC 进程通信 …… 613

16.1 剪贴板通信 …… 613
16.1.1 OpenClipboard()函数 …… 613
16.1.2 CloseClipboard()函数 …… 614
16.1.3 EmptyClipboard()函数 …… 614
16.1.4 SetClipboardData()函数 …… 614
16.1.5 GlobalAlloc()函数 …… 615
16.1.6 GlobalLock()函数 …… 616
16.1.7 GlobalUnLock()函数 …… 616
16.1.8 GetClipboardData()函数 …… 616

16.1.9 一个利用剪贴板在不同进程之间交换数据的示例 …… 616
16.2 邮槽通信 …… 619
16.2.1 CreateMailslot()函数 …… 619
16.2.2 GetMailslotInfo()函数 …… 620
16.2.3 SetMailslotInfo()函数 …… 621
16.2.4 ReadFile()函数 …… 621
16.2.5 GetFileTime()函数和SetFileTime()函数 …… 621
16.2.6 CreateFile()函数 …… 622
16.2.7 WriteFile()函数 …… 623
16.2.8 CloseHandle()函数 …… 623
16.2.9 一个利用邮槽在不同进程间通信的示例 …… 623
16.3 匿名管道通信 …… 628
16.3.1 CreatePipe()函数 …… 628
16.3.2 CreateProcess()函数 …… 629
16.3.3 GetStdHandle()函数 …… 631
16.3.4 ReadFile()和 WriteFile()函数 …… 632
16.3.5 一个利用匿名管道在父子进程间通信的示例 …… 632
16.4 命名管道通信 …… 637
16.4.1 CreateNamedPipe()函数 …… 637
16.4.2 ConnectNamedPipe()函数 …… 639
16.4.3 DisconnectNamedPipe()函数 …… 640
16.4.4 WaitNamedPipe()函数 …… 640
16.4.5 利用命名管道通信的基本流程 …… 640
16.4.6 一个利用命名管道在不同进程间通信的示例 …… 641
16.5 共享内存通信 …… 647
16.5.1 CreateFileMapping()函数 …… 647
16.5.2 MapViewOfFile()函数 …… 648
16.5.3 UnmapViewOfFile()函数 …… 649
16.5.4 OpenFileMapping()函数 …… 649
16.5.5 利用共享内存通信的基本流程 …… 649
16.5.6 一个利用共享内存在不同进程间通信的示例 …… 650
16.6 小结 …… 655

第三部分 Visual C++ 2010 下 MFC 与 CLR 进行开发

第 17 章 Visual C++ 2010 CLR 开发基础 …… 656

17.1 什么是.NET …… 656
17.2 .NET 框架 …… 656
17.3 公共语言运行时（CLR） …… 657
17.3.1 托管代码 …… 659
17.3.2 代码验证 …… 659
17.3.3 代码访问验证 …… 659
17.3.4 垃圾回收 …… 659
17.3.5 语言的互操作性 …… 660
17.3.6 实时编译（JIT） …… 660
17.4 通用类型系统（CTS） …… 661
17.5 通用语言规范（CLS） …… 663
17.6 程序集 …… 664
17.6.1 元数据 …… 664
17.6.2 程序集版本管理 …… 665
17.6.3 微软中间语言（MSIL） …… 665
17.6.4 资源 …… 666
17.7 .NET 开发应用程序的范畴 …… 666
17.8 .NET 框架类库 …… 667
17.9 C++/CLR 开发语法简介 …… 668
17.10 小结 …… 670

第 18 章 Visual C++ 2010 CLR Windows 窗口编程 …… 671

18.1 创建 Windows 应用程序 …… 671
18.2 类层次结构 …… 677
18.3 Control 类 …… 677
18.3.1 大小与位置 …… 678
18.3.2 外观 …… 679
18.3.3 用户交互操作 …… 679

18.3.4 Windows 功能 …… 680
18.4 标准 Windows 控件使用指南 …… 681
18.4.1 Button 控件 …… 681
18.4.2 CheckBox 控件 …… 681
18.4.3 RadioButton 控件 …… 682
18.4.4 ComboBox 控件、ListBox 控件和 CheckedListBox 控件 …… 682
18.4.5 DateTimePicker 控件 …… 684
18.4.6 ErrorProvider 组件 …… 685
18.4.7 HelpProvider 组件 …… 686
18.4.8 ImageList 组件 …… 686
18.4.9 Label 控件 …… 686
18.4.10 ListView 控件 …… 687
18.4.11 PictureBox 控件 …… 688
18.4.12 ProgressBar 控件 …… 689
18.4.13 TextBox 控件、RichTextBox 控件与 MaskedTextBox 控件 …… 689
18.4.14 Panel 控件 …… 690
18.4.15 FlowLayoutPanel 控件和 TableLayoutPanel 控件 …… 690
18.4.16 SplitContainer 控件 …… 691
18.4.17 TabControl 控件和 TabPage 控件 …… 691
18.4.18 ToolStrip 控件 …… 692
18.4.19 MenuStrip 控件 …… 694
18.4.20 ContextMenuStrip 控件 …… 694
18.4.21 ToolStripMenuItem 控件 …… 694
18.4.22 ToolStripManager 类 …… 695
18.4.23 ToolStripContainer 控件 …… 695
18.5 窗体 WinForm …… 695
18.5.1 Form 类 …… 695
18.5.2 多文档界面 …… 700
18.5.3 定制控件 …… 700
18.6 小结 …… 707

第 19 章 Visual C++ 2010 MFC 与.NET 交互编程 …… 708

19.1 编写托管扩展应用程序 …… 708
19.2 编写访问.NET 的 MFC 程序 …… 709
19.3 混合模式编程问题 …… 711
19.4 运用.NET 类型 …… 713
19.4.1 定义和使用托管类型 …… 713
19.4.2 将非托管对象作为托管类的成员 …… 715
19.4.3 装箱和拆箱 …… 716
19.4.4 指针 …… 717
19.4.5 在非托管代码中使用托管数组 …… 719
19.5 小结 …… 720

第四部分 发布 Visual C++ 2010 程序

第 20 章 Visual C++ 2010 应用程序部署 …… 721

20.1 Windows Installer 介绍 …… 722
20.2 一个简单的 Windows 应用程序 …… 723
20.3 使用安装向导快速创建安装包 …… 726
20.4 手动创建安装程序 …… 728
20.5 使用安装编辑器 …… 731
20.5.1 File System（文件系统编辑器） …… 731
20.5.2 Registry 编辑器 …… 732
20.5.3 File Types 编辑器 …… 733
20.5.4 User Interface 编辑器 …… 734
20.5.5 Custom Actions 编辑器 …… 735
20.5.6 Launch Conditions 编辑器 …… 736
20.6 小结 …… 736

第五部分 基于 Windows 7 平台用 Visual C++ 2010 开发

第 21 章 Visual C++ 2010 基于 Windows 7 新特性开发 …… 737

21.1 实现 C++兼容开发 …… 737
21.1.1 实现 UAC 数据重定向 …… 737
21.1.2 实现高 DPI …… 741

21.1.3 实现安装程序检测 …… 742
21.1.4 会话 0 隔离 …… 743
21.1.5 用户界面特权隔离（UIPI） …… 746
21.1.6 版本检查 …… 748
21.2 Windows 7 系统专题 …… 750
21.2.1 实现超级任务栏 …… 750
21.2.2 实现 Shell 库 …… 759
21.2.3 实现后台服务 …… 762
21.2.4 开发基于 Windows 7 的设备与性能应用 …… 763
21.3 开发基于 Windows 7 的新特性 …… 769
21.3.1 实现多点触摸 …… 769
21.3.2 实现获取传感器与位置 …… 771
21.3.3 实现 Windows 7 Ribbon 界面开发 …… 774
21.3.4 基于 Visual C++ 2010 开发基于 Windows 7 的语音识别与语音合成 …… 776
21.3.5 基于 Visual C++ 2010 与 Windows SDK for Windows 7 开发 Windows 7 平台的 Tablet PC 应用 …… 787
21.3.6 开发 Windows 7 的安全体验 CryptoAPI 加密 …… 804

第一部分　Visual C++ 2010 开发与新特性

第 1 章　Visual C++ 2010 开发环境简介

1.1　Visual C++ 2010 简介

Microsoft Visual Studio（简称 VS）是微软公司推出的开发环境，是目前最流行的 Windows 应用程序开发平台。从 1992 年推出的 Visual Studio1.0，将用户带入图形化开发界面，到 2002 年.NET 口号的提出及通用语言框架机制（Common Language Runtime，CLR）的引入，再到 Visual Studio 2010 的发布，微软在开发平台上不懈努力，旨在给开发者提供友好的界面及高效稳健的运行环境。随着 Visual Studio 2010 的发布，很多新的功能和特性将帮助开发者更加清晰和便捷地打造应用程序，如灵活高效的 IDE、增强的并行计算等。Visual Studio 2010 的目标是成为一个经典的版本。图 1.1 是 Visual C++ 6.0 的经典界面，图 1.2 是 Visual Studio 2010 的清新改版。

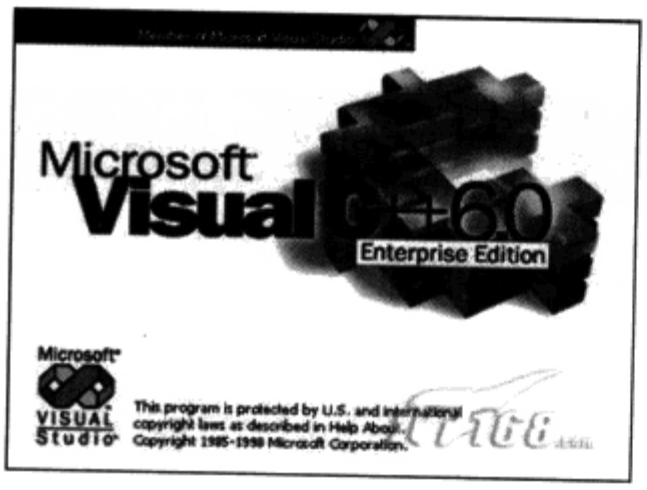


图 1.1　Visual C++ 6.0 启动界面

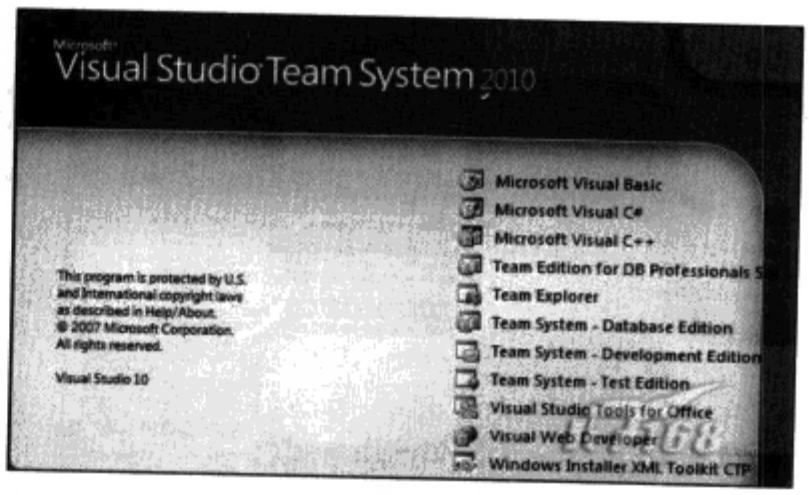


图 1.2　Visual Studio 2010 启动界面

虽然 Visual Studio 经历了诸多版本的改进升级，但是 Visual Studio 6.0 以后的 C++再没有多大的变化，包括上一个版本 Visual Studio 2008，都只是对 C++进行一些细小的改善。随着 C++新标准的公布，Visual Studio 2010 在 C++开发方面也带来了很多革命性的变化。本书将从 Visual Studio 2010 的安装讲起，到开发一个相对完整的应用程序结束，既有总体轮廓的描述又有具体到每一个类的讲解；既有旧知识的回顾，又有新知识的加载，无论是新手还是微软开发环境的忠实用户，都将有与众不同的体验和收获。

1.2　Visual C++ 2010 下载安装指南

安装 Visual C++ 2010 之前，必须安装 Visual Studio 2010 开发平台，这是因为 Visual C++ 2010 的安装是包含在 Visual Studio 2010 的安装过程中的。Visual C++ 2010 所需的软硬件配置与 Visual Studio 2010 相同，应满足以下要求（仅供参考）。

- 处理器最低 1.6 GB Hz，建议使用 2.2 GHz 或者更快的处理器；
- 支持的操作系统包括 Windows Server 2003、Windows Vista、Windows XP、Windows 7；

- 内存最低 512MB，建议使用 1GB 或者更大的内存；
- 硬盘最低 5400r/min，建议使用 7200r/min 或者更快的硬盘；
- CD-ROM 驱动器、键盘和鼠标；
- 显示器最低 1024 像素 × 768 像素的显示分辨率，建议使用 1280 像素 × 1024 像素的显示分辨率。

如果性能不满足，可以通过执行以下操作改进本产品在计算机上的总体性能。

- 关闭 devenv.exe 上的病毒检查；
- 安装本产品后，对硬盘驱动器进行碎片整理；
- 检查所安装版本的 RAM 和处理器建议，然后考虑是否升级内存或加快磁盘速度。

当打开集成开发环境（IDE）时：

- 关闭计算机之前，请关闭在启动时不使用的工具窗口，这可以加快下次启动时的启动速度；
- 在启动时不显示属性窗口，当打开解决方案时，属性窗口会自动显示。

当满足以上条件时，就可以下载安装 Visual C++ 2010 了。首先请登录微软官方网站 http://www.microsoft.com/visualstudio/zh-cn/products/2010-editions/ultimate。

看到下载字样，单击“下载”按钮即可。

下载的文件是.exe 格式，直接单击“运行”按钮。如图 1.3 所示，Visual Studio 2010 会提示选择安装目录，并且在右下方列出各个磁盘的大小、可用空间、需要空间，以及安装 Visual Studio 后的剩余空间，非常人性化。注意在窗口左侧选择要安装.NET 环境还是 C++环境。

单击“Install”按钮，进入安装过程，如图 1.4 所示。

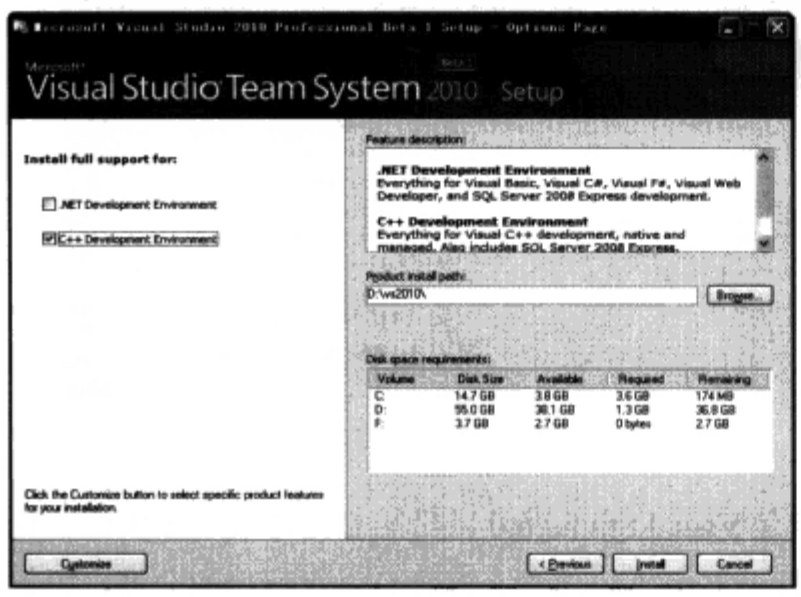


图 1.3 Visual Studio 2010 安装选择界面

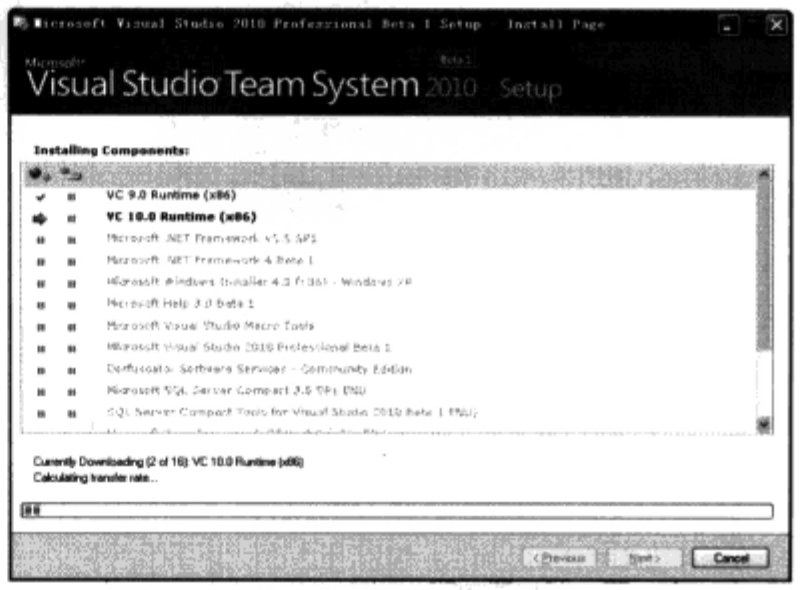


图 1.4 Visual Studio 2010 安装进度

图 1.4 上面提示框中显示的是要安装的组件列表，图中第一个组件已经安装完成，正在安装第二个组件，下面的进度提示显示正在安装的 16 个组件中的第二个的进度。这个安装过程需要的时间比较长，并且在组件安装完成后会提示重新启动计算机，按照提示重新启动即可。

安装程序成功后，窗口如图 1.5 所示，左侧显示安装有问题的组件，根据工程的实际需求及自己计算机的情况酌情处理有问题的组件。这时，Visual Studio 2010 已经成功装入计算机，在“开始”→程序”菜单中可以找到“Microsoft Visual Studio 2010”命令，单击它即可进入应用程序。注意：第一次安装程序，要选择默认环境设置。

图 1.6 所示是 Visual Studio 2010 的主界面，界面干净、简洁，颜色柔和，各种功能一目了然。下面介绍 Visual C++ 2010 的主要特点和新特性，以及如何操作开发环境。

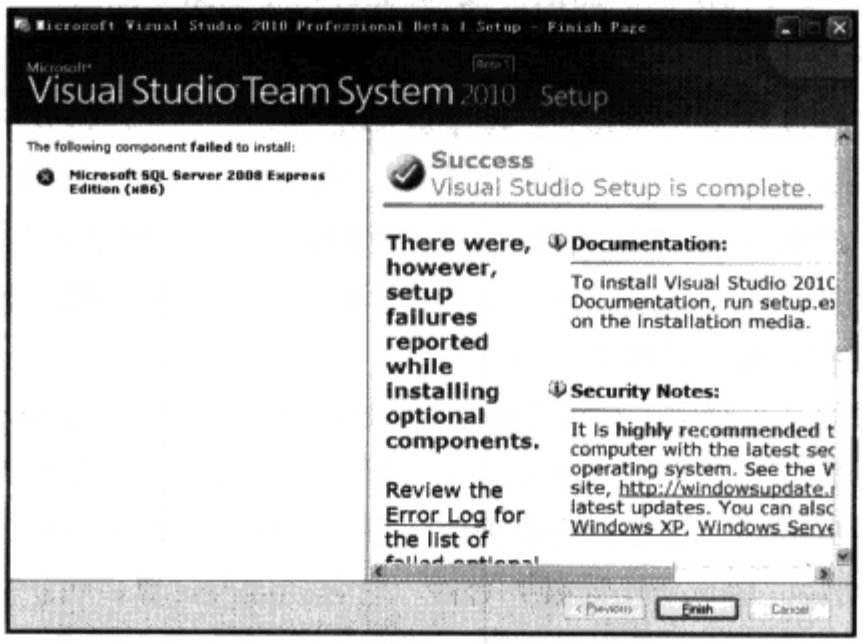


图 1.5　安装完成界面

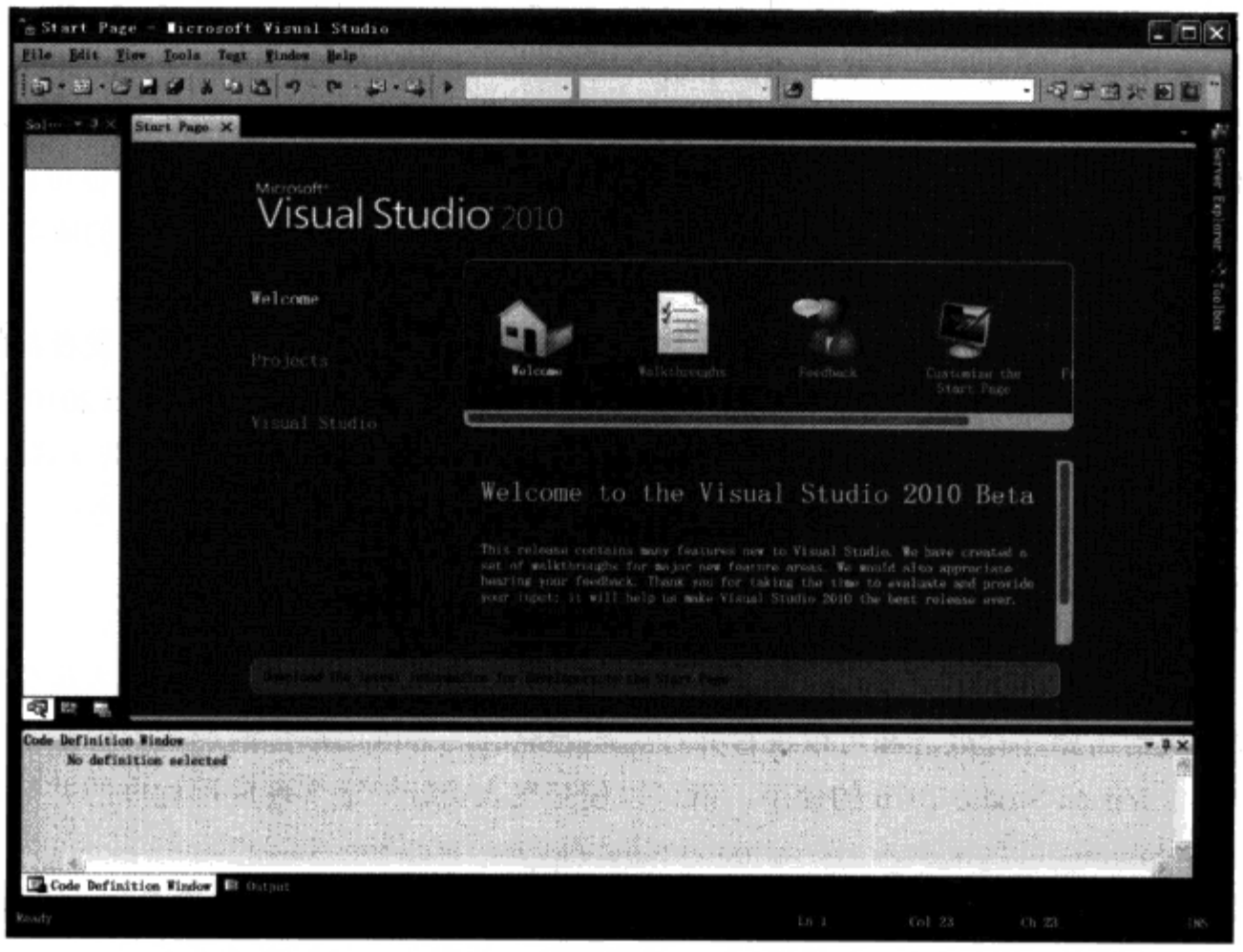


图 1.6　Visual Studio 2010 开始界面

1.3　Visual C++ 2010 主要特点与新特性

Visual Studio 2010 开发代号为“Rosario”（阿根廷港市罗萨里奥），相对于 Visual Studio 2008，从里到外都进行了全新设计，例如，将带来增强的用户界面体验、更有条理的应用程序生命周期管理、更高的创造性和开发效率、更深入的 Web 开发、强大的云计算开发（Azure）、更多的数据库支持、更简单的并行编程等。

自微软于 1998 年发布 Visual Studio 6 以来，Visual Studio 的 IDE 已经成为软件开发工具的标杆，

很多其他的开发工具，甚至是其他用途的应用程序，都在模仿 Visual Studio 的 IDE。但是从 Visual Studio 6 到 Visual Studio 2008，虽然 IDE 的功能越来越多，但是并没有什么革命性的变化，反倒因为功能太多带来了使用上的不便，导致开发效率低下。程序员都在期盼一个全新的 IDE 的出现。

程序员的梦想在 Visual Studio 2010 中成为了现实。在 Visual Studio 2010 中，微软用全新的 WPF 技术重新打造了编辑器，借助 WPF 的强大功能，新的编辑器可以实现很多 Visual Studio 2008 的 IDE 根本无法想象的功能，比如代码的无级缩放、多窗口即时更新、文档地图、代码的自动产生等，这些新的 IDE 特性都会极大地提高程序员的开发效率。

云计算的口号已经喊得越来越响，随着 Visual Studio 2010 CTP 和 Windows Azure 的发布，云计算终于成为事实。通过使计算分布在大量的分布式计算机上，而非本地计算机或远程服务器中，企业数据中心的运行将更与互联网相似，这使得企业能够将资源切换到需要的应用上，根据需求访问计算机和存储系统。Windows Azure 是一个托管服务套件，它包括虚拟计算、可扩展存储及自动化服务管理系统等，这些工具将用来为微软的服务提供支持，其中包括 MSN、Xbox Live，以及 Office Online 等，实现服务的网络化。

对于开发者而言，Windows Azure 已经建立起一个简单而快速的系统，最重要的是，它拥有标准的模型。开发者可以根据自己的需要选择使用第二层功能，比如数据库、业务工具，甚至是第三方软件提供的功能。使用 Windows Azure Tools for Visual Studio，可以为 Windows Azure 创建、调试和部署服务和应用程序。Visual Studio 2010 为 Windows Azure 提供了专门的项目模型，同时，也可以利用 Visual Studio 2010 对服务和应用程序进行调试。另外，利用 Visual Studio 2010 可以将创建的服务打包，然后通过 Windows Live Developer Portal 将其部署到 Windows Azure。

Visual Studio 2010 加大了对并行运算的支持，微软正在使尽量大的范围内的开发者都能高效地进行并行计算的开发，而不管使用的是非托管代码还是.NET Framework。在 Visual Studio 2010 中，将看到：

- Visual Studio IDE 对并行计算开发的大量支持，比如，Visual Studio 2010 的调试器知道代码的并行特性，并且能够在调试程序的不同执行单元的时候，表现应用程序的状态；
- 非托管的 C++库和编译器对并行计算的支持；
- .NET Framework 4.0 对并行计算的大量支持，包括 P-LINQ、并行语言语句等。

另外，Visual Studio 2010 还提供了一个“并行性能分析器”，它可以帮助开发人员分析应用程序的性能瓶颈，找到需要并行处理和可以进行并行处理的地方，并以图形化的形式表现出来。“并行性能分析器”配合 Visual Studio 2010 的使用，就可以使开发人员轻松地实现应用程序的并行化，再次吃上“免费的午餐”。

除了上文中阐述的种种新特性，可能最令各位开发人员感到兴奋的就是 C++开发的增强。自 Visual Studio 6 以来，Visual Studio 中的 C++再没有多大的变化，包括之前的 Visual Studio 2008，都只是对 C++进行一些细小的改善。但是随着 C++新标准——C++0x 的即将公布，Visual Studio 2010 在 C++开发方面也带来了很多革命性的变化。

首先是对 C++新标准——C++0x 的全面支持，不会再像 Visual C++ 6 那样对 C++标准支持不佳。在 IDE 方面，微软将 Visual C++的构建系统 VCBuild 整合到了 MSBuild 中，借助后台编译，Visual C++的 IntelliSense 更加智能，能够处理更多的文件、更加复杂的项目；在 MFC 方面，通过引入很多新的类，MFC 开始全面支持 Windows Vista、Windows 7 风格的 UI。这些特性都是 Visual Studio 2010 与 Visual Studio 2008 的重要差别，相信 C++程序员都会选择 Visual Studio 2010 而略过 Visual Studio 2008。

对于开发基于 WPF 的应用程序的开发者，Visual Studio 2010 同样提供了改进的工具，帮助开发者快速、高效地完成界面图像的设计、数据绑定等。

总之，若想让应用程序“Windows 7 Ready”，Visual Studio 2010 是首选。与 Visual Studio 2008 相比，Visual Studio 2010 作为微软着力打造的下一代开发工具平台，拥有无数诱人的特性，同时也寄托着无数人的期望。通过前面的一系列对比介绍，相信读者对 Visual Studio 2010 的成功还是抱有很大希望的。

微软将新版本的 Visual Studio 定位为提供一款基于其 WPF（Windows Presentation Foundation）技术的新编辑器，用于可视化应用软件的开发。除了 WPF 技术之外，Visual Studio 2010 还包含众多热门的技术元素：generate from usage 的功能强化代码管理、Silver light 2 支持、触摸操作、并行计算和云计算等。

微软开发工具产品的市场主管 Dave Mendel 表示，“我们正在改进 Visual Studio 的用户界面和用户体验，这是十多年以来的第一次”。Mendel 称：“我们要把 Visual Studio 应用本身变成一个（可视化的）WPF 应用”，开发环境将为开发者提供更具吸引力的用户体验，“我们正试图改进界面，让开发者感觉到工作环境很舒服”。微软称：这个基于 WPF 的编辑器将“史无前例”地深入具体应用，以“丰富并易于理解的方式”提供代码。

开发者现已能通过 Visual Studio 2008 创建 WPF 应用程序，而 Visual Studio 2010 本身就是一个 WPF 型应用。

另外，Visual Studio 2010 还新增了一项名为“generate from usage”的功能强化代码管理。由代码推理得出代码的开发者正试图做什么，并按照开发者的意图编写代码，此功能已被列入 Visual Studio 2010 预览版。

Visual Studio 2010 将全力支持 Silver light 2 丰富的互联网应用插件技术。Mendel 表示，“实际上这意味着，对想要通过 Visual Studio 来编辑 Silver light 2 应用程序的开发者来说，这个建立 Silver light 应用的环境拥有丰富的拖放[功能]”。

Visual Studio 2010 的其他功能规划还有：针对 Windows 7 用户操作系统的应用程序开发。开发者将能够利用 Ribbon 界面和图标显示文件内容的缩略图，还可以使用 Windows 7 的搜索功能来搜索文件操作系统、元数据和电子邮件。此外，Multitouch 功能也将可用。

Mendel 表示，“通过 Multitouch 功能，使用者可以在激活了触摸计算的计算机上进行触摸操作”。

无论是源代码开发人员编写 C++，还是托管代码开发者建设.NET 框架，Windows 7 的触摸功能和 Ribbon 界面均可支持。

微软表示，“Visual Studio 2010 提供了 C++全新的集成开发环境（IDE）来支持流行的应用，如并行计算、云计算和网络服务”。

开发者还可以使用 Visual Studio 2010 来开发和调试 SharePoint 协作应用。

Forrester Research 的资深分析师 Jeffrey Hammond 表示，“目前很多大型企业都能很顺利地安装使用 SharePoint，但是其开发过程（即使是对一位经验丰富的.NET 开发者来说）却是不甚顺利的。

在虚拟化方面，Visual Studio Team System 2010 提供了一个虚拟实验室管理组件，用于支持虚拟机进行应用程序运行环境测试，例如应用程序是如何运行于不同版本的 Windows 系统（如 Windows Vista 和 Windows XP）的。

微软的软件开发战略还涉及 ASP.NET 4.0 版，这是微软的一项网站开发技术。Hendlen 称，该技术将使“模型-视图-控制器（Model-View-Controller，MVC）”结构声名大噪。在 MVC 模式下，开发人员可以利用测试驱动开发来建立 MVC 应用程序，这些应用程序以分离成片的形式存在。ASP.NET 4.0 将成为 Visual Studio 2010 及.NET Framework 4.0 的一部分。

ASP.NET 4.0 的特点是，将在 Visual Studio 下采用 jQuery JavaScript 库开发 AJAX 应用程序。IntelliSense JavaScript 引擎也是一大特色，Hendlen 称“IntelliSense 的作用是（在开发人员写代码的时候）为开发人员提供一些建议”。

基于微软安全案发生命周期倡议，作为微软可信赖计算倡议（Trustworthy Computing Initiative）的一部分，微软将提供一些技术和程序，旨在使程序开发变得更安全，应用开发隐私度更高。

这些措施包括：

- SDL 优化模型，用于在微软之外的组织实施 SDL;
- SDL 威胁建模工具试用版（SDL Threat Modeling Tool Beta），用于应用程序中潜在的安全和隐私问题的早期分析及解决;
- SDL 专业网络（SDL Pro Network），用于服务供应商支持开发者实施 SDL。

一项名为“软件开发生命周期的专业网络”的合作伙伴计划即将问世，该计划支持者包括帮助执行微软的安全开发生命周期的安全专家。

1.4 Visual C++ 2010 开发环境操作指南

对开发人员而言，对一个工具的认识和熟悉，即了解 Visual C++ 2010 的开发环境是非常重要的。这种熟悉的过程也是培养工具使用感情的过程。新的 Visual C++开发平台和以往的平台有很大的不同，本节将对 Visual Studio 2010 集成开发环境的主要部分做简要的介绍。

1.4.1 创建 Visual C++应用程序

图 1.7 所示是 Visual Studio 2010 的主界面，界面最上端是标题栏，用于显示应用程序名和打开的文件名。标题栏下面是菜单栏和工具栏。工具栏下面的区域分成 3 个部分，启动时标签为“Start Page”的区域是一个多文档窗口，称之为主窗口，这个窗口嵌入了浏览器，代码编辑和网页浏览都在其中进行；左侧是工作区窗口 Solution Explorer；下面是代码定义窗口，这些窗口均允许用户用鼠标拖动大小，任意关闭和打开。下面进行详细的介绍。

图 1.7 Visual Studio 2010 开始界面

Visual Studio 2010 的主窗口包括“Welcome”、“Projects”、“Visual Studio”3 个选项卡，图 1-7

所示是“Welcome”选项卡，在这个选项卡中，右侧的图标显示其具体的功能，如单击“Walkthroughs”图标便进入微软 MSDN Visual Studio 开发中心，下方的文字是对当前功能的介绍。

在“Projects”选项卡中可以进行新工程的创建，打开已经存在的工程，以及进行代码编辑。单击“New Project”图标，弹出如图 1.8 所示的界面。

图 1.8　创建 C++工程

窗口左侧列出了可创建的工程模板，因为这里专注于 Visual Studio 平台上的 C++代码设计，所以图 1.8 中列出的是 Visual C++ 2010 的工程模板，单击 ⊞ 图标，“Other Project Types”、“Test Projects”将被展开，选择“Online Templates”命令，将在窗口右侧显示网上搜索模板结果。窗口中间部分是将左侧分类模板分别列出来的结果，在窗口上方可以选择.NET 的版本，以及根据名称升序或者降序排列。窗口右侧是对选中模板的说明。表 1.1 列出了可创建的工程模板。

表 1.1　　**可创建的工程模板**

C++工程模板	说　明
ATL Project	创建 ATL 项目，包括动态链接库、可执行程序和服务器应用程序
Class Library	创建在其他应用中需要的类
CLR Console Application	创建控制台应用
CLR Empty Project	创建本地应用的空工程
Windows Forms Application	创建 Windows 界面的用户应用
Windows Forms Control Library	创建在其他 Windows 程序中使用的 Windows Control
Windows Service	创建 Windows 的服务
Make file Project	创建 Make 文件项目
Custom Wizard	创建自定义的 Wizard
Empty Project	创建本地应用的空白工程
MFC Application	创建 MFC 应用程序
MFC DLL	创建 MFC 动态链接库
MFC ActiveX Control	创建 MFC ActiveX 控件程序
Test Project	创建测试程序
Win32 Project	创建 Win32 程序，包括控制台程序，Windows 程序，动态链接库或者静态库
Win32 Console Application	创建 Win32 控制台程序

选择“MFC”→“MFC Application”模板，并输入文件名，选择存储路径，单击“OK”按钮，会弹出如图 1-9 所示的“MFC Application Wizard”对话框，进入创建应用程序向导。

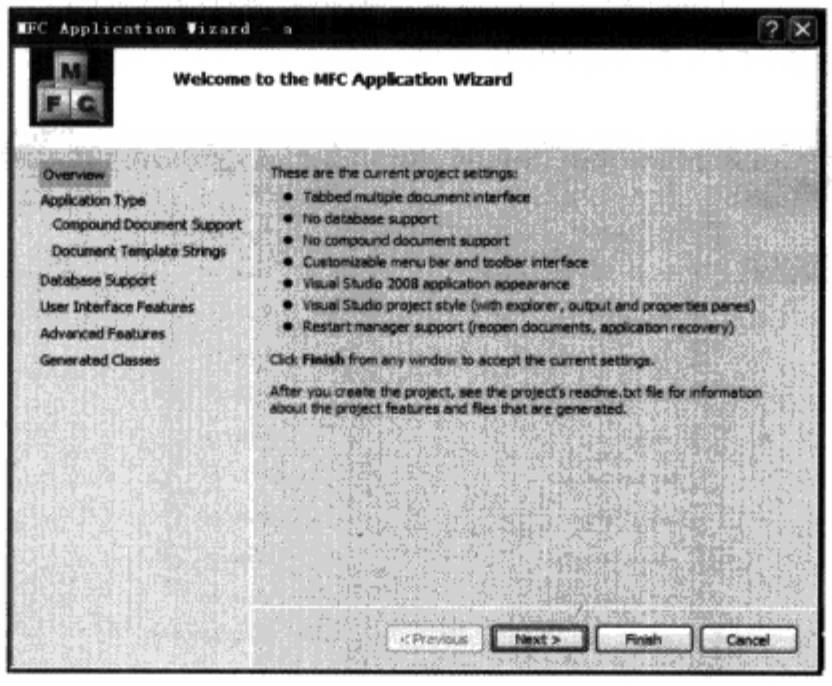


图 1.9　MFC 向导

此时窗口左侧高亮显示的是“Overview”选项，这个选项显示了工程的一些默认设置，如右侧显示的 Tabbed multiple document interface（多文档界面）、No database support（不支持数据库）、No compound document support（不支持复合文档）等。如果这时单击“Finish”按钮，向导将会按照默认设置创建一个新的工程；如果想要更改这些设置，单击“Next”按钮进入“Application Type”选项。

在“Application Type”对话框中，可以选择单文档、多文档或者基于对话框的应用程序，可以选择的工程风格是 Windows Explorer、MFC standard（MFC 标准风格）、Visual Studio 或者 Office，还可以选择语言、界面风格等。需要注意的是，“Application Type”选项下有两个子选项，即“Compound Document Support（用于指定支持何种类型的复合文档）”和“Document Template Strings（用于指定文档模板串）”，可以更近一步对应用程序类型进行设计，如图 1.10 所示。再单击“Next”按钮将进入“Database Support”选项，进入“Database Support”对话框，如图 1.11 所示，这里不再赘述。

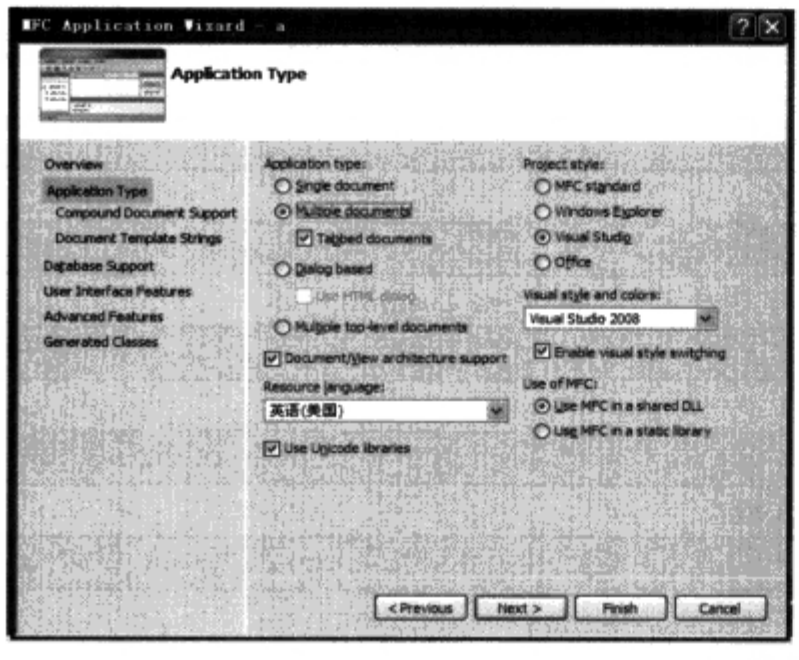


图 1.10　对话框选择项

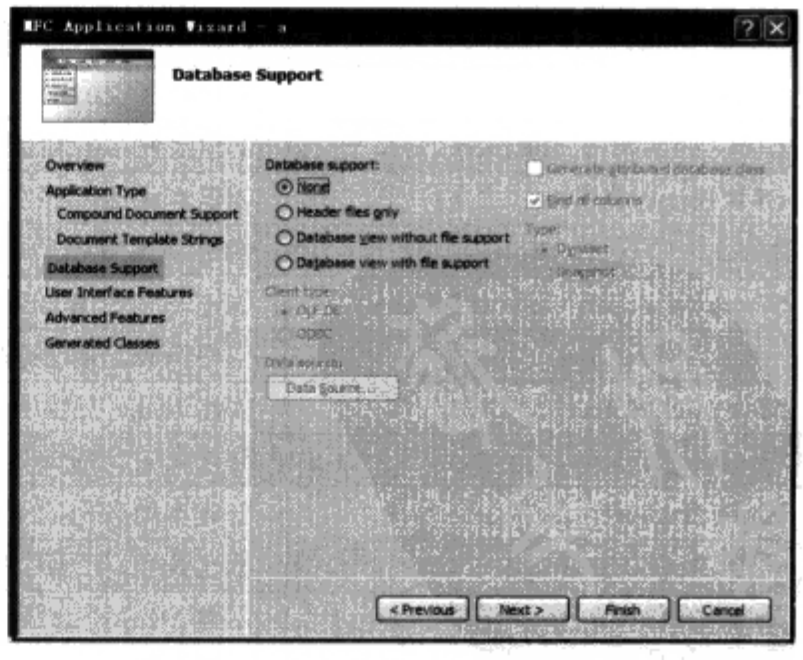


图 1.11　数据库支持

接着进入“User Interface Features”选项，进入“User Interface Features”对话框，如图 1.12 所示，在其中可以设置创建的应用程序外观。用户可以设置是否自动生成系统菜单和状态条，是否支持最大

化或者最小化等。

接着单击“Next”按钮，进入“Advanced Features”选项，如图 1.13 所示。

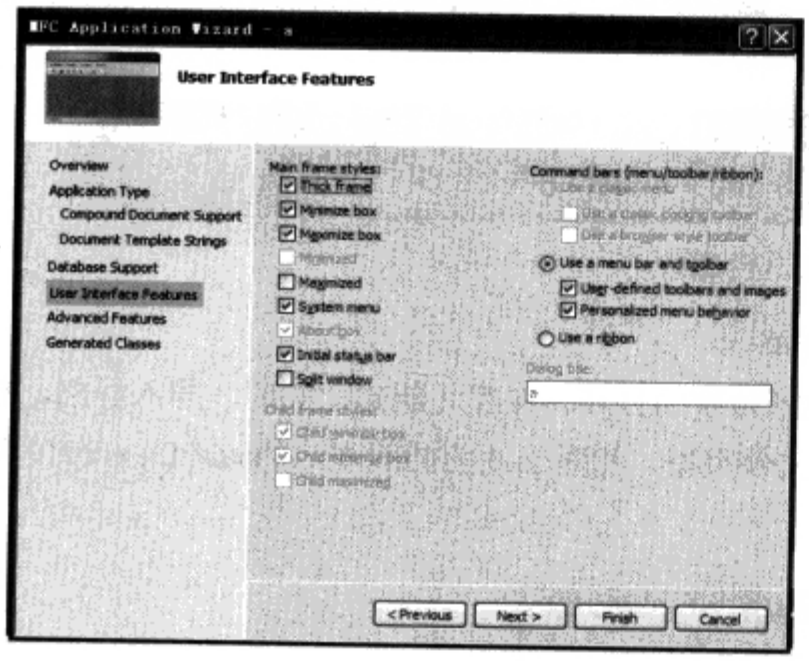


图 1.12　应用接口支持

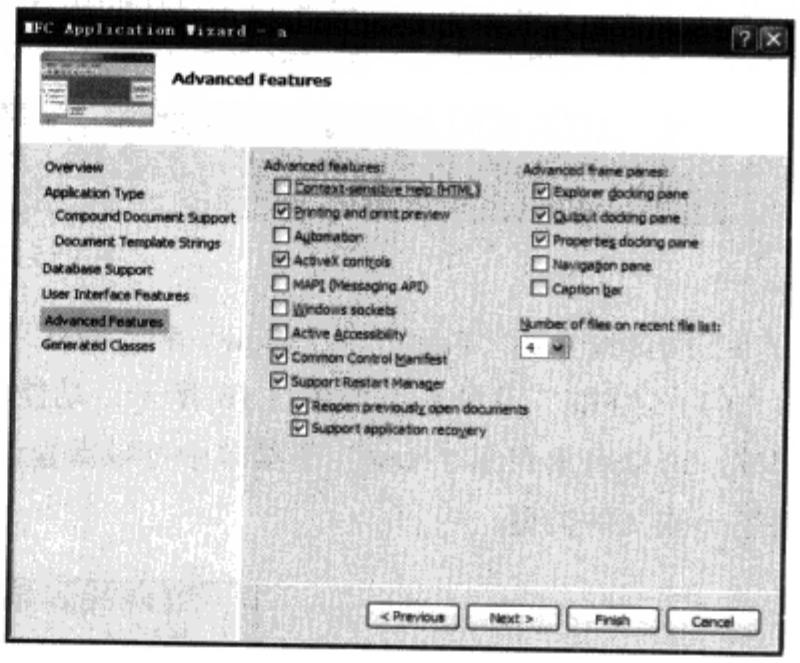


图 1.13　高级应用支持

在“Advanced Features”对话框中可以设置一些高级属性，包括 Printing and print preview（打印及打印预览）、ActiveX controls（ActiveX 控件）等。

最后进入“Generated Classes”选项，这个选项可以显示由向导生成的全部类，还可以为应用程序指定基类，让用户一目了然，如图 1.14 所示。如果有设置不当的地方，单击“Previous”按钮可以返回前面设置的任何一步。如果以上的设置都符合要求，单击“Finish”按钮，新的工程就生成了。

通过以上步骤，可以看到，Visual C++2010 的页面更加美观、简洁，用户可以随时查看并更改自己的选项，采用网页方式生成应用程序。相对于之前的版本，在创建工程时大的框架并没有改变，只是增加了一些子选项，让用户的选择余地更大。

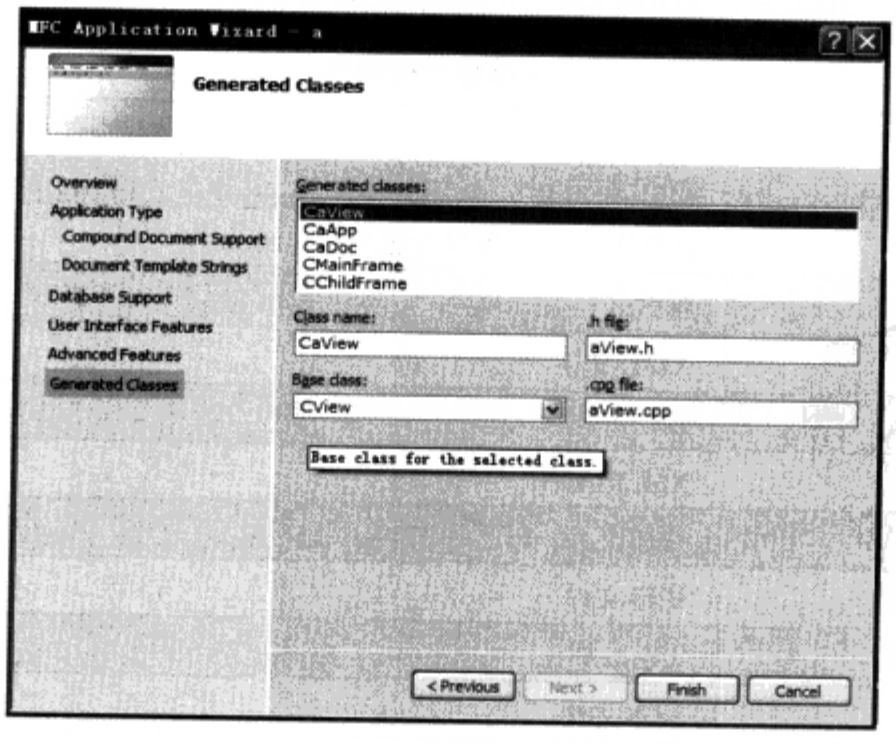


图 1.14　MFC 类选择

1.4.2　Visual C++ 2010 菜单介绍

创建新的工程之后，需要熟悉菜单，以便能方便、快捷地编写并调试程序。

如图 1.15 所示，创建工程之后菜单变成了 10 个：“File”、“Edit”、“View”、“Project”、“Build”、“Debug”、“Tools”、“Test”、“Window”、“Help”。相对于之前的版本，Visual C++ 2010 新增了“Test”菜单，这是为了方便测试程序，调试程序。这些菜单可以直接通过单击打开，也可以用“Alt+菜单首字母”组合键来打开。

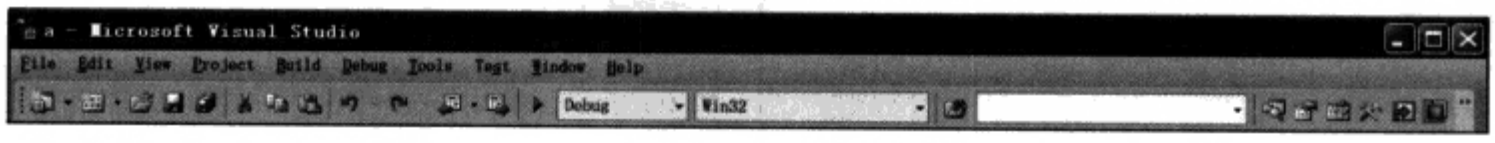


图 1.15　菜单栏截图

下面详细介绍“File”、“View”和“Test”菜单。

（1）“File”菜单命令如图 1.16 所示，功能包括传统的打开、保存、新建、关闭、导入导出、打印等，这里着重介绍“New”子菜单中可以新建的文件类型，相对于之前的版本，Visual C++ 2010 新增了一些文件类型。

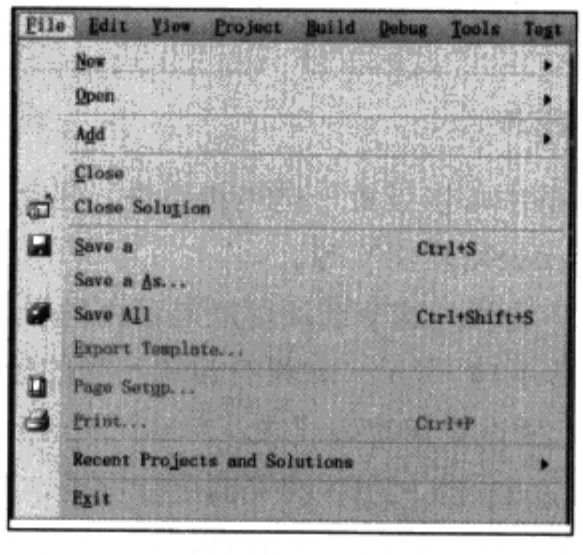


图 1.16　文件菜单截图

文件分为 3 大类型：General、Visual C++和 Script，具体如表 1.2 所示。

表 1.2　　3 大类型文件说明

General 文件类型	说　明
Text File	创建文本文件
HTML page	HTML 页面，可包含客户端代码
Style Sheet	样式表
XML File	XML 文件
XML Schema	XML 模式
XSLT File	XSLT 文件
Bitmap File	位图文件
Icon File	图标文件
Cursor File	光标文件
Native Resource Template	脚本文件

Visual C++文件类型	说　明
C++ File（.cpp）	C++源程序文件
Header File（.h）	C++头文件
Midl File（.idl）	界面解说语言文件

续表

Visual C++文件类型	说　明
Bitmap File（.bmp）	位图文件
Cursor File（.cur）	光标文件
Icon File（.ico）	图标文件
Resource Template File（.rct）	包含 Win32 模板的文件

Script 文件类型	说　明
JScript File	脚本文件
Windows Script Host	Windows 脚本宿主

除了表 1.2 中的文件类型以外，还可以选择“Online Templates”命令，这时主窗口自动进行网络搜索，并将得到的模板列到“Online Templates”下，以便于查询。可以看到，在 Visual C++ 2010 版本中，新增了一些文件类型，如 XSLT、XML Schema 等，这些文件的加入使得这个版本更加贴近实际应用，更加便捷。

（2）“View”菜单的命令如图 1.17 所示，可以看到，相对于之前的版本，Visual C++ 2010 重新整合了一些功能，也增加了一些功能。选择每个命令时可以用鼠标单击，也可以按照命令名右侧的组合键进行选择。

当选择“Bookmark Window”、“Call Hierarchy”、“Code Definition Window”、“Output”命令时，会在窗口下方显示相应的编辑窗口，并且按照标签的方式显示，使得界面整洁而不凌乱。当选择“Property Manager”、“Resource View”、“Solution Explorer”、“Object Browser”、“Class View”命令时，会在窗口左侧按照树状图显示当前的工程或者是需要的属性。如果想关闭某个标签，在选中这个标签时，单击“关闭”按钮即可。选择“Toolbox”命令会在窗口右侧显示工具箱，是通用控件和对话框控件，选择“Toolbars”命令可以选择当前的工程选择工具栏中的工具。

（3）“Test”菜单如图 1.18 所示，这个菜单是 Visual C++ 2010 版本中新添加的，是为了使测试程序和调试程序更加方便。

选择“New Test”命令新建一个测试程序，会弹出如图 1.19 所示的窗口，其中列出了可以新建测试程序的类型及注释，在右侧窗口中还可以选择左侧图标显示方式，即图片显示或者列表显示。

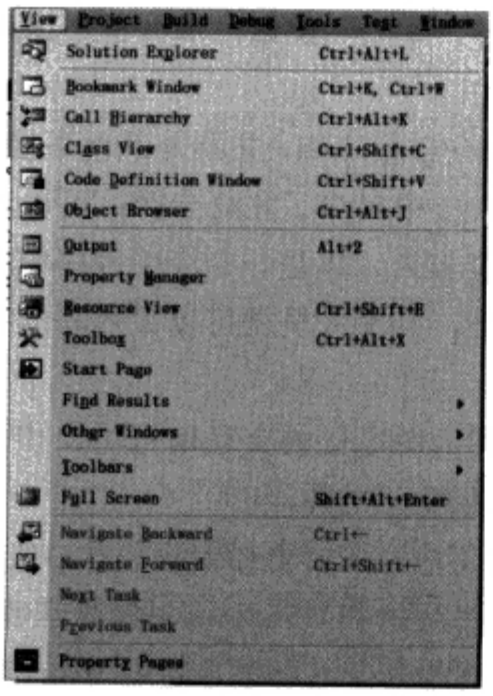


图 1.17　文件主菜单截图

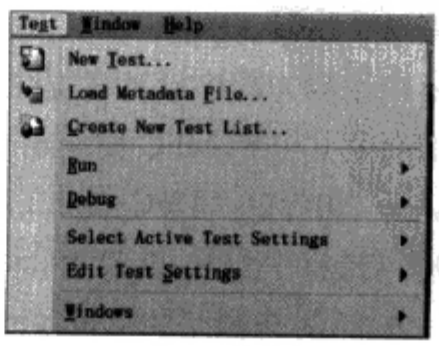


图 1.18　测试菜单截图

以上是对 Visual C++操作环境的介绍，包括主窗口和菜单介绍。熟悉以上功能之后，就能创建 C++应用程序及进行简单的调试。如果想了解更多的功能，请参见相关的帮助文档。

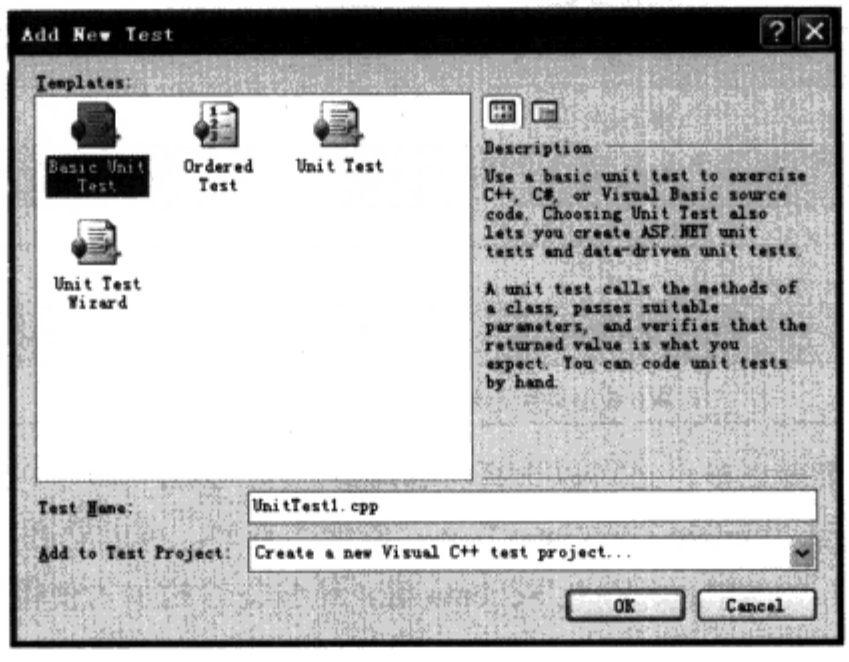


图 1.19　创建测试程序

1.5　Visual C++ 2010 MFC 简介

在了解 MFC 之前，首先要清楚 Windows 的编程模型。在过程化的编程方法中，每次程序调用从开始到结束的路径本身是可以预测的，如 C 程序从 main()函数的语句开始执行，到 main()函数返回值结束，在主程序中，main()函数会调用其他函数，并且允许嵌套调用，但是始终是程序而不是操作系统决定何时调用哪个函数。

Windows 程序则使用事件驱动编程模型，程序通过处理操作系统发送来的消息来响应事件，这些消息包括鼠标、窗口刷新等，编程模型如图 1.20 所示。

Windows 程序从 WinMain()函数进入，这个函数创建应用程序的窗口并进入消息循环。此时输入的各种消息，如图 1.20 所示的鼠标单击等都进入消息队列等待，窗口过程函数便调用其他函数来帮助处理收到的消息。可以调用应用程序自己的函数，也可以调用 Windows 提供的 API 函数。这时，如果用户从“文件”菜单中选择了“退出”命令，或者单击了“关闭”按钮，那么程序结束的消息便会进入消息队列，被检索出之后，应用程序结束。

MFC 就是微软提供的用于处理消息的 API 类库。在 MFC 程序中，并不是经常直接调用 API 函数，而是从 MFC 类创建对象并调用这些对象的成员函数。MFC 中封装了几百个类，可以直接使用，也可以将其作为基类来编写自己需要的类。MFC 不仅是一个类集合，也是一个应用程序的框架结构，可以为应用程序处理许多事情。有了这些封装成类的 API 函数，开发人员就可以很容易地利用面向对象的编程方法来编写 Windows 应用程序。

面向对象程序设计为程序设计者提供了革命性的便利，图像化的交互使得用户和计算机之间的关系更加紧密，这也是 C++和 MFC 如此重要、如此贴近人们生活的原因。

在 Visual C++ 2010 中，MFC 增加了新的功能，与时俱进，变得更加饱满和容易使用，主要表现在支持 Office 2007 风格程序设计，支持 Visual Studio 风格程序分析和 Windows 资源管理器风格程序设计。

支持 Office 2007 风格程序和支持 Visual Studio 风格程序分析主要都是通过 Visual Studio 2010 SP1 中 MFC 的 Ribbon 功能来实现的。

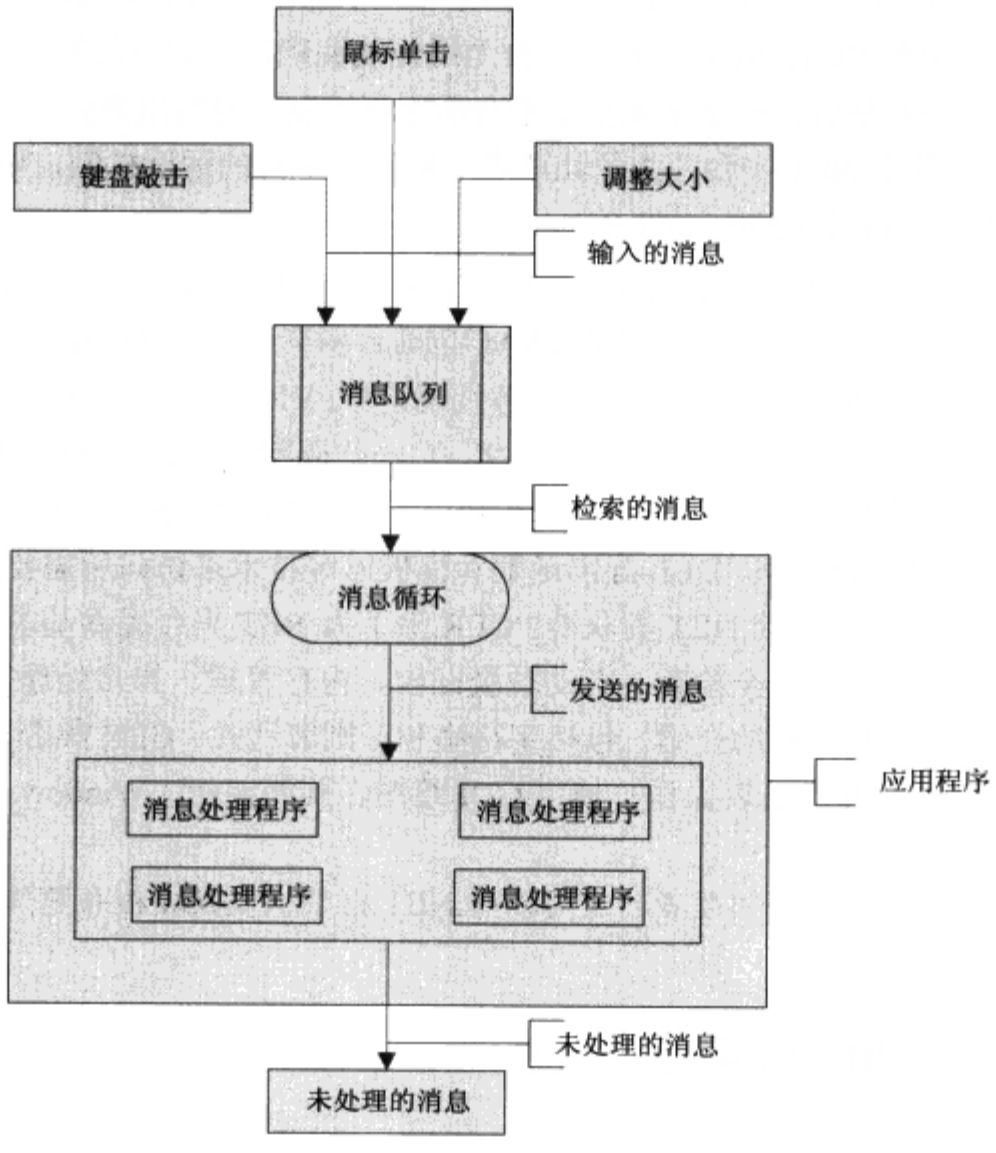


图 1.20　应用程序消息机制

比尔·盖茨说，“多年以来，在 Windows 软件中，采用‘文件’、‘编辑’、‘视图’的菜单模式，俨然已经成为标准。但是这样做会隐藏部分功能，下拉菜单很长，用户根本不知道菜单底部的命令。很多用户要求的功能，其实软件中早就已经有了。”Ribbon 界面就是把相似的功能集中在一起，形成不同的选项卡，方便用户浏览并且选择功能，这样的设计也更加适用于越来越流行的触屏操作。

除了这 3 个功能，Visual Studio 2010 的 MFC 还带来了任务对话框和重启管理器的特性。相比于消息框（Message Box），任务对话框不仅可以向用户显示消息，还可以显示自定义的按钮、命令行选项及更加丰富的辅助信息等，这改善了用户体验。重启管理器可以在程序意外崩溃的时候，保护用户未保存的文档数据，有效地降低了数据丢失的风险，极大地改善了程序的抗风险能力。

本书的第 2～16 章对 MFC 进行了全面并且系统的介绍。从 MFC 应用程序的框架开始分析，介绍了 MFC 菜单编程，对话框编程及对话框控件，文档视图编程，应用程序界面和美化，文本和字体，图形图像编程；除此以外，还围绕动态函数链接库，MFC ActiveX 控件，文件与注册表操作，数据库开发，多线程程序设计，网络程序设计，进程通信对 MFC 编程进行介绍。介绍得非常全面和细致，通过这些章节的学习，读者可以对 MFC 编程有系统性的认识，而且还非常实用。

1.6　Visual C++ 2010 CLR 简介

2002 年，微软提出了.NET 口号，Visual Studio.NET 可以被用来开发 Windows 应用程序、.NET 框架、Web 应用程序及移动设备。.NET 框架的目标是多语言运行平台，因为它采用中间语言机制，源程序被编译成中间语言，当执行的时候，中间语言被即时编译成适用于所运行平台的机器语言，这样就

建立起了多语言运行平台，在此平台上，包括 Visual C++，Visual Basic，Visual FoxPro 和 C#等开发工具。本书着重为大家介绍在 Visual Studio 2010 平台上用传统编程语言 C++编写程序的过程和体验。虽然只是介绍其中的一个开发平台，但是掌握整个平台的运行机制，对写出高效的程序很有帮助。

Visual Studio 2010 基于.NET 平台，为了让读者了解这个平台的运行机制，下面介绍.NET 通用语言运行库（Common Language Runtime，CLR）。

运行库的概念在微软技术中并不是刚刚出现的，读者应该有所了解，如 Visual Basic 运行库和 Java 虚拟机（JVM）。但是与 Visual Basic 运行库和 JVM 不同，.NET 运行库不局限于特定的语言，如上文介绍的微软和第三方公司的几种语言都可以编写源代码，如 Visual C++，通过对它进行扩展可以生成.NET 代码，这些扩展被称为 C++托管扩展。这样，C++代码就可以访问.NET 框架类库和用其他支持.NET 的语言编写的库，同样，其他语言可以使用由 C++编写的托管库。C++开发人员要创建功能全面的应用程序，不再需要使用 COM、DLL 导出函数和模板库等技术来访问所需要的库，所有必需的库代码都可以通过.NET 框架类库中的.NET 类获得。CLR 除了为.NET 平台提供基本功能外，还提供其他的多种功能，如代码验证、访问安全检查、自动垃圾回收、内存管理、异常处理等。

本书第 17～29 章都是对 CLR 的介绍，包括字符串和正则表达式、CLR 集合、CLR Windows 窗口编程、CLR 数据访问、文件和注册表操作、使用 GDI 绘图、网络编程、Windows 服务、线程和同步、还有安全性、国际化和事务处理。

通过这些具体到类的介绍，相信读者一定可以对.NET 框架机制及编程有整体并且深入的了解。

1.7 Visual C++ 2010 64 位编程

64 位计算技术从 2004 年推出至今，其产品线不断丰富，应用日趋广泛，如 Intel 公司的 EM64T 技术、IA-64 技术和 AMD 的 AMD64 技术，推出支持这些技术的 CPU 有使用 Nocona 核心的 Xeon 系列、使用 Prescott 2M 核心的 Pentium 4 6 系列和使用 Prescott 2M 核心的 Pentium 4 EE 系列，AMD 方面有 Athlon 64 系列、Athlon FX 系列和 Opteron 系列。

64 位技术是相对 32 位而言的，这个位数指的是通用寄存器的数据宽度为 64 位，64 位指令集就是运行 64 位数据的指令，也就是说处理器一次可以运行 64 位数据。从 32 位到 64 位，计算机寻址范围、最大内存容量、数据传输和处理速度、数值精度等指标成倍增加，其结果是 CPU 的处理能力大幅提升，尤其是对强烈依赖数值运算、存在大量数据吞吐和需要超大并发处理的应用提升效果非常明显，如科学计算、人工智能、平面设计、视频处理、3D 动画和游戏、数据库及各种网络服务器等。

从 Windows Vista 系统推出到 Windows7 的全面覆盖，64 位技术在操作系统上的全面应用使得应用软件的开发又有了一个新的平台。如何在 64 位技术上进行程序的高效编写，是从架构设计师到程序员都要面临的问题。

Visual C++2010 全面支持 64 位编程，从 32 位架构转换到 64 位架构，需要从数据模型的角度重新认识。通过对 MFC、CLR 的认识，读者就可以在 Visual C++ 2010 的平台上开发出属于自己的应用程序。第 30 和第 31 章介绍了如何将 MFC 和.NET 交互起来进行编程，并介绍了如何安装部署程序。

1.8 支持新的 C++语言标准

1.8.1 支持新的 C++语言标准（C++ 0x）

C++ 0x 将是下一代的 ISO C++标准，它将取代现行的 C++标准 ISO/IEC 14882（公布于 1998 年并

于 2003 年更新，通称 C++98 及 C++03）。新的标准将包含内核语言的新机能，而且会扩展 C++标准程序库，并入了大部分的 C++ Technical Report 1 程序库。

虽然目前 C++ 0x 标准还未正式发布，但是在 C++ 0x 标准中还是引入了众多的新特性，这些特性将对 C++语言的未来发展带来巨大的影响。作为下一代的开发工具，Visual Studio 2010 加入了对 C++ 0x 标准的支持。下面将分别讲解 Visual Studio 2010 对 C++0x 标准的几个主要特性的支持，这些特性包括 Lambda 表达式、静态断言 static_assert、auto 关键字、右值引用、安全数据类型和移动构造。

1.8.2　Lambda 表达式

许多编程语言都支持匿名函数的概念，所谓匿名函数指的是只有函数体而没有函数名的函数，匿名函数广泛运用于那些只被使用一次且函数体较小的情况。Lambda 表达式是一种与匿名函数密切相关的编程技术，实际上一个 Lambda 表达式隐式定义了一个函数类并构造了一个该函数类的函数对象。可以将 Lambda 表达式看作一个可以维持状态并且可以访问所在封闭作用域的变量的匿名函数。在作用上，Lambda 表达式类似于函数指针和函数对象，Lambda 表达式很好地兼顾了函数指针和函数对象的优点，却没有它们的缺点。相对于函数指针或是函数对象复杂的语法形式，Lambda 表达式使用非常简单的语法就可以实现同样的功能，降低了 Lambda 表达式的学习难度，避免了使用复杂的函数对象或是函数指针所带来的错误。下面为一个简单的使用 Lambda 表达式的例子：

```
#include <algorithm>
#include <iostream>
#include <ostream>
#include <vector>
using namespace std;
int main()
{
        vector<int> v;

         for (int i = 0; i < 10; ++i) {
                v.push_back(i);
          }

       for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });
       cout << endl;
}
```

这个程序的输出如下：

```
0 1 2 3 4 5 6 7 8 9
```

在语句

```
for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });
```

的后半段包含 Lambda 表达式，其中[]代表 Lambda 表达式引入符，这个符号告诉编译器一个 Lambda 表达式的开始。后面的(int n)为 Lambda 参数声明，告诉编译器这个匿名函数调用所使用的参数，后面的{}所包含的语句代表这个匿名函数的函数体，即告诉编译器这个匿名函数执行哪些操作。

与普通匿名函数不同的是，这里的 Lamda 表达式并没有指定（函数）返回值类型。实际上，Lambda 表达式默认的返回值类型为 void。为了对比，下面列出使用函数对象实现相同功能的代码：

```
#include <algorithm>
#include <iostream>
#include <ostream>
```

```
#include <vector>
using namespace std;

class LambdaFunctor {
public:
        void operator()(int n) const {
                cout << n << " ";
        }
};

int main() {
        vector<int> v;

        for (int i = 0; i < 10; ++i) {
                v.push_back(i);
        }

        for_each(v.begin(), v.end(), LambdaFunctor());
        cout << endl;
}
```

在这里，首先需要定义一个函数类 LambdaFunctor，定义()运算符的操作，这里明确指定返回值类型为 void，然后在 for_each 语句中构造这个类的一个函数对象实现调用。对比一下，就可以发现使用 Lambda 表达式简洁得多。

上面提到 Lambda 表达式可以操作所在作用域的变量，这需要通过被称为“捕获”的特殊语法来实现，就是通过在[]内列出将要捕获的“外部”变量列表，这样在函数体内就可以访问并操作这些变量，参考下面的代码：

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
        //创建一个包含 10 个元素的集合对象
        vector<int> v;
        for (int i = 0; i < 10; ++i)
        {
                v.push_back(i);
        }

        // 使用 for_each 语句和 Lambda 表达式来实现对偶元素的计数

        int evenCount = 0;
        for_each(v.begin(), v.end(), [&evenCount] (int n) {
                cout << n;

                if (n % 2 == 0)
                {
```

```
                cout << " is even " << endl;

                // Increment the counter.
                evenCount++;
            }
            else
            {
                cout << " is odd " << endl;
            }
        });

        // 将偶元素个数打印出来
        cout << "There are " << evenCount
             << " even numbers in the vector." << endl;
}
```

这个程序的输出是：

```
0 is even
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
There are 5 even numbers in the vector.
```

语句

```
for_each(v.begin(), v.end(), [&evenCount] (int n) {
```

通过[&evenCount]声明来捕获外部变量 evenCount，这里使用&表示为捕获引用，就是希望 Lambda 函数体可以返回 evenCount 的操作结果（默认为按值捕获，即捕获变量的一个副本，不会影响外部变量本身的值）。如果希望捕获多个变量，则使用逗号（,）来分隔这些变量，比如：

```
//x 为按值捕获，y,z 为捕获引用
for_each(v.begin(), v.end(), [x,&y,&z] (int n) {
```

1.8.3　静态断言 static_assert

断言表示为一些布尔表达式，表示程序员相信该表达式在程序中的某个特定点的值必须为真，否则就是程序设计或执行出错。程序员可以在任何时候启用和禁用断言验证，因此可以在测试时启用断言而在部署时禁用断言。使用断言可以创建更稳定、品质更好且易于出错的代码。

在之前的 C++ 03 标准中，可以使用两种断言：

- 使用预处理中的条件编译和#error 指令，可以在预处理阶段检查一些编译条件；
- 可以使用宏 assert 来进行运行时检查，以确保程序逻辑的正确性。

使用条件编译并不能对模板参数进行检查，这是因为模板实例化是在编译期进行的，而#error 方法是在预处理阶段进行的。宏 assert 则是在运行时进行检查。不难发现，这里缺少了可用于在编译期对模板进行检查的工具，于是静态断言应运而生。

静态断言的作用是提供一种编译阶段的断言机制，如果断言内指定的常量表达式为 0（false），那么将出现一个编译错误，同时编译器会将用户指定的错误信息显示出来，否则没有任何影响。

在新的 C++ 0x 标准中，加入了对静态断言的支持，引入了新的关键字 static_assert 来表示静态断言。静态断言的使用语法如下：

```
static_assert(
        constant-expression,        //执行判断的常量表达式
        string-literal              //指定输出的错误信息
);
```

其中，静态断言 static_assert 接受两个参数，第一个参数为可以转化为布尔类型的常量表达式，注意，必须为常量表达式，即在编译阶段就可以确定其值的表达式；第二个参数为一个字符串。下面的代码演示如何使用静态断言：

```
static_assert(sizeof(void *) == 4, "64-bit code generation is not supported.");
```

而下面的用法则是错误的：

```
int d   = rand() % 2;
static_assert( d == 0, "variable d is not even.");
```

由于必须在运行时才能确定 d 的值，所以 d==0 不是常量表达式，不能使用静态断言。下面举一个更复杂的例子来说明静态断言在处理模板参数方面的运用：

```
#include <type_traits>
#include <iosfwd>
namespace std {
    template <class CharT, class Traits = std::char_traits<CharT> >
class basic_string {
        static_assert(tr1::is_pod<CharT>::value,
"Template argument CharT must be a POD type in class template basic_string");
// ...
        };
}
struct NonPOD {
        NonPOD(const NonPOD &) {}
        virtual ~NonPOD() {}
};
int main()
{
      std::basic_string<char> bs;
}
```

在上面的代码中，静态断言在类作用域中。static_assert 用于验证模板类参数是否为 POD 类型。编译器在类定义时检查 static_assert 声明，但是直到在 main()函数中将模板实例化为 basic_string 类时才能确定静态表达式参数的值。

1.8.4 auto 关键字

auto 关键字是一种声明修饰符，是从 C++ 98 开始引入的，主要在声明（或定义）语句中使用。在 C++ 0x 标准之前，在声明（或定义）一个变量时加上 auto 修饰符，表示声明一个变量为自动存储类型，这意味着该变量只在定义所在块可见，具有局部生命期。实际上，对块作用域的变量而言，其默认都是自动存储类型，比如下面代码中包含的声明语句是等价的：

```
int foo()
{
    auto int i = 0;          //变量 i 被明确声明为 auto（自动存储类型）
    int j = 0;               //变量 j 被默认声明为 auto
}
```

正因为块作用域变量默认为 auto 类型，所以程序员很少在声明语句中使用 auto 关键字，这使 auto 关键字显得“无用”。

这里需要说明的一点是，不管是否使用 auto 修饰符，声明变量时都必须明确指定变量的类型，尽管有些时候一些变量的类型相当复杂。

但是在引入 C++ 0x 新特性的 Visual C++ 2010 中，auto 关键字的含义发生了变化。同样作用于声明语句，此时的 auto 关键字表示声明的变量类型将根据初始化表达式自动推导出来，而无须由程序员明确指定。auto 关键字的使用需注意两点：

（1）将变量类型替换为 auto；

（2）必须在定义处对变量进行初始化。

比如下面的语句是合法的：

```
auto k = 0;                  //变量 k 为 int 类型，因为 0 是一个整型数
auto d = 2 * 3.14159;        //变量 d 为 double 类型
```

而下面的用法则是错误的：

```
auto int k = 0;              //使用 auto 关键字后不要再指定类型
auto d;                      //必须在定义处对变量进行初始化
d = 2 * 3.14159;
```

还可以对 auto 关键字进一步使用 const、volatile、指针（*）、引用（&）等进行修饰，编译器负责分析初始化表达式，并根据分析得到的信息来推导出变量的正确类型：

```
auto x = 1, *y = &x, **z = &y;          //解析到 int
const auto a(2.01), *b (&a);            //解析到 double，使用 const 修饰
```

这是一个很有用的性质，尤其是在变量类型比较复杂时，可以大大简化程序员的工作。比如代码：

```
map<int,list<string>>::iterator i = m.begin();
auto i = m.begin();
```

上面的两个定义语句是完全等价的，在第二个语句中，编译器将根据 m 的类型及 begin()调用的返回值推导出变量 i 的类型为 map<int,list<string>>::iterator。第二种用法显然比第一种要简单得多。

还可以在 for 及 for each 循环语句中使用 auto 关键字，比如下面代码中的变量 it 及 elem：

```
#include <deque>
int main()
{
    deque<double> dq1(2,0.1);
    for (auto it = dq1.begin(); it != dq1.end(); ++it)
    { /* ... */ }
    for each(auto elem in dq1)
    { /* ... */ }
}
```

在 Visual C++ 2010 中， auto 关键字默认为新的含义，编译器选项为/Zc:auto。为了兼容旧有的 C++ 代码，可以使用/Zc:auto-编译器选项来明确告诉编译器采用 auto 关键字的原有定义。

1.8.5　右值引用

作为一项很重要的语言特性，右值引用（rvalue references）被引入 c++ 0x 中，通过操作符“&&”

来声明一个右值引用，而使用“&”声明的引用通常被称为左值引用：

```
A a;
A& a_ref1 = a;          //左值引用

A a;
A&& a_ref2 = a;         //右值引用
```

在进一步讲解如何使用右值引用之前，先来解释什么是左值（lvalue）和右值（rvalue）。

每个 C++表达式都有一个类型（比如 int、double 等），但是往往会忽视一个重要的事实，即，每个 C++表达式还具有一个表示其存储空间的属性：要么是左值，要么是右值。所谓左值就是指在表达式使用完之后仍存在的对象（存储空间），可以简单地认为左值就是具有名称的对象，所有的变量（包括 const 修饰的变量）都是左值。而右值指的是那些在表达式使用完之后就不再存在的存储空间，实际上所有的临时变量表达式都是右值。下面的代码解释了左值和右值的区别：

```
int x = 3 + 4;
double d = sin(x);
```

在上面的例子中，x 和 d 都是左值，这是因为在定义它们的表达式之外，其存储空间仍存在并可访问。而表达式 3+4 及函数调用 sin(x)都是右值，这是因为它们都将结果存储在临时对象中，在表达式之外，这些空间都是不可访问的。

右值引用允许程序员明确区分左值和右值。左值引用和右值引用的表现行为基本一致，但是它们也遵循一些不同的规则，一个明显的区别就是右值引用可以绑定到一个临时对象（右值）上，而左值引用不可以。右值引用至少可以用于解决以下两种问题：

- *移动语义*（move semantics）;
- *完美转发*（Perfect forwarding）。

下面分别介绍右值引用的这两个运用。

1．移动语义

移动语义允许程序员将资源（比如动态分配内存）从一个对象移动到另一个对象，这将大大提高程序的性能，注意这里是移动而非复制。右值引用可以支持移动语义，是因为它允许资源从临时对象移动出来，而除此之外是无法引用临时对象（右值）的。

当对右值进行操作时，右值本身往往没有必要保留，因此在有些情况下，右值资源可以直接移动给其他对象。通过右值引用，程序可以明确地区分出传入的参数是否为右值，从而避免不必要的复制，因而可以提高程序的效率。下面考虑 std 库内的实现数据交换的 swap()函数，假设 X 是一个已经实现备份构造函数及备份赋值运算符的类：

```
template<class T>
void swap(T& a, T& b)
{
    T tmp(a);
    a = b;
    b = tmp;
}

X a, b;
swap(a, b);
```

这里没有使用右值，所以 swap()函数内的 3 行代码都没有使用移动语义，每一行都需要数据备份。

在 C++ 0x 中的 std 库中将引入一个新的 std::move()函数，这个函数只是简单地将参数转化为右值。

所以在 C++ 0x 中，std 库内的 swap()函数将变成类似于下面的实现：

```
template<class T>
void swap(T& a, T& b)
{
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}

X a, b;
swap(a, b);
```

通过调用 std::move()函数引入右值引用，使得 swap()函数的 3 个语句具有移动语义，这样便消除了数据备份操作，只会将源对象移动到目标对象。

2．完美转发

完美转发（perfect forwarding）技术能够有效消除函数重载的需求并避免转发问题。当编写一个将引用作为它的参数的通用函数，并且这个参数将被传递（或转发）给另一个函数时，往往会出现转发问题（forwarding problem）。比如，如果一个通用函数要求的参数类型为 const T&，这意味着被调用函数不能修改该参数的值；如果通用函数要求参数类型为 T&，则不能使用右值（比如临时对象或一个整型常数）来调用。

一般情况下，为了解决这个问题，需要分别定义这个函数的两种不同重载形式，分别带 T&和 const T&参数，这样的后果是，随着通用函数所带参数的个数增加，需要的不同重载形式将成指数倍增加。而右值引用允许程序员只写一种版本的函数，就可以接受任意的参数，并将它们转发给其他的函数，就好像直接调用函数一样。

考虑下面例子中定义的 4 种类型 W、X、Y 和 Z，它们的构造函数分别需要 const 及非 const 的左值引用作为参数：

```
struct W
{
    W(int&, int&) {}
};

struct X
{
    X(const int&, int&) {}
};

struct Y
{
    Y(int&, const int&) {}
};

struct Z
{
    Z(const int&, const int&) {}
};
```

现在假设想写一个产生对象的通用函数，其中一种写法如下：

```
template <typename T, typename A1, typename A2>
T* factory(A1& a1, A2& a2)
{
    return new T(a1, a2);
}
```

下面的代码演示了对 factory()函数的合法调用方式：

```
int a = 4, b = 5;
W* pw = factory<W>(a, b);
```

但是下面的代码中对 factory 的调用是非法的，因为 factory()函数要求将左值引用作为参数，但却通过右值来调用：

```
Z* pz = factory<Z>(2, 2);
```

通常解决这个问题的方法是，创建分别处理 T&及 const T&的重载函数。右值引用允许程序员只写 factory()函数的一种版本，如下所示：

```
template <typename T, typename A1, typename A2>
T* factory(A1&& a1, A2&& a2)
{
    return new T(std::forward<A1>(a1), std::forward<A2>(a2));
}
```

这里使用右值引用作为 factory()函数的参数，而 std::forward()函数的目的就是将传递给 factory()函数的参数转发给模板类的构造器。

下面的代码进一步演示了可以使用不同的方式来调用 factory()函数分别产生不同类的对象，而这些调用参数总能被正确地转发给合适的类构造器（正如 perfect 所修饰）：

```
int main()
{
    int a = 4, b = 5;
    W* pw = factory<W>(a, b);
    X* px = factory<X>(2, b);
    Y* py = factory<Y>(a, 2);
    Z* pz = factory<Z>(2, 2);

    delete pw;
    delete px;
    delete py;
    delete pz;
}
```

1.8.6 安全数据类型

SafeInt 库是 Visual C++ 2010 新增加的一个库，这个库中包含的类和函数提供了一种对不同类型整数进行安全算术操作的机制，允许操作 8 位到 64 位的整数，SafeInt 库将自动检测算术运算是否溢出或者出现被 0 除错误。

SafeInt 库中最重要的就是 SafeInt 类，这个类还允许程序员对不同类型的整数进行比较大小操作，前提条件是它们都属于 SafeInt 对象。而对于一般的比较操作，需要先将两个数的类型转化为一致，并且保证转化没有数据丢失。

下面的代码演示了 SafeInt 类如何实现对两个 8 位整数的安全加法操作：

```
unsigned __int8 i1 = 250;              //8 位非负整数，最大值为 256
```

```
unsigned __int8 i2 = 150;                    //直接进行相加操作，将导致溢出
SafeInt<unsigned __int8> si1(i1);            //先转化为 SafeInt，以保证加法安全
SafeInt<unsigned __int8> si2(i2);
SafeInt<unsigned __int8> siResult = si1 + si2;
cout << (int)siResult << endl;
```

这里使用默认的 SafeIntException 类来处理操作异常，当检测到算术操作溢出或被 0 除错误时，直接终止应用程序。可以自己指定异常类处理：

```
unsigned __int8 i1 = 250;
unsigned __int8 i2 = 150;
SafeInt<unsigned __int8, CMySafeIntException> si1(i1);
SafeInt<unsigned __int8, CMySafeIntException> si2(i2);
SafeInt<unsigned __int8, CMySafeIntException> siResult = si1 + si2;
cout << (int)siResult << endl;
```

上面的代码将导致抛出一个 CMySafeIntException 异常，因为 250+150 的 8 位算术操作导致溢出。如果不使用 SafeInt 库将得到错误的结果 144，而程序运行不会给出任何错误或警告信息。下面列出自定义异常类的 CMySafeIntException 实现，这个类直接继承自 SafeIntException 类：

```
class CMySafeIntException : public SafeIntException
{
public:
    static void CMySafeIntException::SafeIntOnOverflow()
    {
        cout << "Caught a SafeInt Overflow exception!" << endl;
    }
    static void CMySafeIntException::SafeIntOnDivZero()
    {
        cout << "Caught a SafeInt Divide By Zero exception!" << endl;
    }
};
```

注意，这里的 CMySafeIntException 类并不像一般的 C++异常类那样被抛出（throw）和捕获（catch）。它不需要程序员显示对可能问题的代码段进行“监督”（catch），这个类包含两个特殊的函数，分别用于处理算术运算溢出及被 0 除错误。对于 SafeIntException 类，其默认的处理就是调用 abort()函数退出程序。而在这里，重载了这两个函数，只是在出现错误时打印出相关的信息，而不会退出函数（如果想退出，调用 exit()函数即可）。

1.8.7　移动构造

在 1.8.5 小节中，学习了利用一个对象的右值引用可以实现移动语义，移动语义的最大优点是消除了大量的临时对象的复制，而临时对象的复制是造成程序效率低下的主要原因之一。前面介绍的移动语义只针对系统的内置类型（比如 int、char），为了实现移动语义只需提供右值引用即可。

下面介绍为更复杂的对象来实现移动语义，比如自己设计的一个类（class），对于一个类而言，移动语义还是通过右值引用来实现，这里的关键是先为类提供一个被称为移动构造函数的特殊的构造函数，同样只需要操作右值引用，其他的交给编译器来完成（实现类对象移动）。

下面举个简单的例子来说明如何为一个类实现复制构造函数，首先列出没有实现移动语义的类的代码：

```
class CSomeObject
```

```
{
public:
    CSomeObject()
        : m_iBufferSize(0)
        , m_pBuffer(NULL)
        , m_iInstanceID(ms_iInstanceCounter++)
    {
    }

    // 正常构造函数
    CSomeObject(unsigned int iBufferSize)
        : m_iBufferSize(iBufferSize)
        , m_pBuffer(NULL)
        , m_iInstanceID(ms_iInstanceCounter++)
    {
        if (m_iBufferSize > 0)
        {
            m_pBuffer = new char[m_iBufferSize];
            memset(m_pBuffer, 0, m_iBufferSize);
        }
    }

    // 复制构造函数，注意这里使用左值引用，所以将导致对象的复制
    CSomeObject(const CSomeObject& objSource)
        : m_iInstanceID(ms_iInstanceCounter++)
    {
        m_iBufferSize = objSource.m_iBufferSize;
        if (m_iBufferSize > 0)
        {
            m_pBuffer = new char[m_iBufferSize];
            memset(m_pBuffer, 0, m_iBufferSize);
        }
    }
    ~CSomeObject()
    {
        if (m_pBuffer)
            delete [] m_pBuffer;
        m_pBuffer = NULL;
    }

protected:
    unsigned m_iBufferSize;
    char* m_pBuffer;
    int m_iInstanceID;

    static int ms_iInstanceCounter;
};
int CSomeObject::ms_iInstanceCounter = 0;
```

在上面的代码中，自定义了一个类 CSomeObject，这个类包含 3 个普通成员和一个静态成员

ms_iInstanceCounter（用于统计类对象的个数）；还定义了一个复制构造函数，注意这里的复制构造函数接受的参数是一个左值引用，这就意味着需要进行复制。如果传入的参数只是一个临时对象，就是说在复制构造完成后，这个对象就会被销毁，所以这将导致一次无谓的复制。更好的实现方式显然还是移动语义，将对象移动过去，避免复制，提高效率。

下面的代码演示如何在一个类中实现移动构造函数，以避免这种无谓的复制：

```
class CSomeObjectWithMoveConstructor : public CSomeObject
{
public:
    CSomeObjectWithMoveConstructor()
        : CSomeObject()
    {
    }

    // 正常构造函数
    CSomeObjectWithMoveConstructor(unsigned int iBufferSize)
        : CSomeObject(iBufferSize)
    {
    }

    // 复制构造函数
    CSomeObjectWithMoveConstructor(const CSomeObjectWithMoveConstructor& objSource)
        : CSomeObject(objSource)
    {
    }

    // 移动构造函数
    CSomeObjectWithMoveConstructor(CSomeObjectWithMoveConstructor&& objSource)
    {
        m_iBufferSize = objSource.m_iBufferSize;
        m_pBuffer = objSource.m_pBuffer;

        objSource.m_iBufferSize = 0;
        objSource.m_pBuffer = NULL;
    }
};
```

这里在前面代码的基础上，增加了一个移动构造函数，这个函数接受一个右值引用作为参数。首先将源对象的成员赋值给新对象，然后将源对象置为 NULL。通过这些操作，在移动构造函数内进行的操作实际上是将 m_pBuffer 成员分配的内存空间（资源）从源对象移动到新的对象。注意，移动完成后，必须将其设置为 NULL，是为了避免源对象的析构函数将这段空间释放掉，因为这段空间已经不再属于源对象。再次强调，移动构造函数操作的参数是一个右值引用（&&）。

在 Visual C++ 2010 的 STL 实现中，已经支持对集合中的类对象元素实现移动构造，比如 std::vector 对象在增加集合的大小（size）时，将自动实现对每个集合元素的移动构造。使用 Visual Studio 2010，对那些实现了移动复制的类对象集合进行操作时，可以享受到移动复制所带来的性能提升。

对于增加一个集合对象的大小，以前的实现步骤如下所示：

（1）新开辟一块更大空间的内存用于存放新的集合对象；

（2）将所有原集合中的元素复制到新的空间中；

（3）将所有原集合中的元素删除；

（4）释放原集合的存储空间。

分析这里的操作步骤，可知原集合内的每个元素实际上相当于临时对象，将对象复制到新空间后就被马上销毁，显然使用复制是完全没有必要的，反而会导致带来效率的低下。而使用移动则变为下面的步骤：

（1）为新集合新开辟一个更大的存储空间；

（2）将原集合内的所有元素移动到新的存储空间；

（3）将原集合中每个元素置为 NULL；

（4）释放原集合的存储空间。

这两种实现方式最大的差别在于步骤（2）中的移动和复制。下面通过一个实际的例子来测试这两种实现的性能差异：

```
static int siBufferSize = 1024*1024*10;
static int siObjectCount = 100;
//对没有实现移动构造的类对象集合进行操作
vector<CSomeObject> vec2;
for (int i=0; i<siObjectCount; ++i)
{
    vec2.push_back(CSomeObject(siBufferSize));
}

//对实现移动构造的类对象集合进行操作，将自动实现移动语义
vector<CSomeObjectWithMoveConstructor> vec3;
for (int i=0; i<siObjectCount; ++i)
{
    vec3.push_back(CSomeObjectWithMoveConstructor(siBufferSize));
}
```

测试这两段代码的执行时间，读者会发现上一个集合操作耗费的时间将明显多于下面集合操作的时间。

1.9 支持开发并行程序

1.9.1 运行库支持 native 代码

Visual Studio 2010 可以帮助 C++开发人员高效地写出并行计算的代码，这是因为新的 C++语言在并行方面引入了一系列类库和新功能的支持。其中对于 native 代码，提供了一个被称为并行运行时的并行支持，下面介绍如何利用并行运行时来开发并行程序。

并行运行时（concurrent runtime）是一个 C++并行编程框架，可以使并行编程更加简单。并行运行时为程序员提供更高的抽象，使得程序员不必再纠缠于烦琐的与并行相关的底层细节。并行运行时还允许程序员指定调度策略以满足具体运用的需求。

并行运行时为同时运行的运用及组件提供了统一性和可预见性，并行运行时带来合作任务调度及合作阻塞等优点。

并行运行时使用一种基于工作窃取（work-stealing）算法的合作任务调度，将工作有效地分配给计算资源。比如，考虑一个具有两个被同一运行时管理的线程的运用，如果一个线程完成了自己的任务，

它可以从另一个线程再分摊一些工作。这样的机制能够平衡应用分配给每个线程的负载。

并行运行时还提供同步原语，使用合作阻塞来实现对资源的访问的同步。比如，考虑一个任务必须取得对某一共享资源的独占访问，通过合作阻塞，运行时能够使用剩余量子使其他任务像第一个任务那样等待资源。这种机制有利于计算资源的最大化利用。

并行运行时被划分为 4 个模块：并行模式库（PPL）、异步代理库、任务调度器和资源管理器。这几个模块位于操作系统和应用程序之间，图 1.21 演示了并行运行时如何与操作系统及应用程序互动。

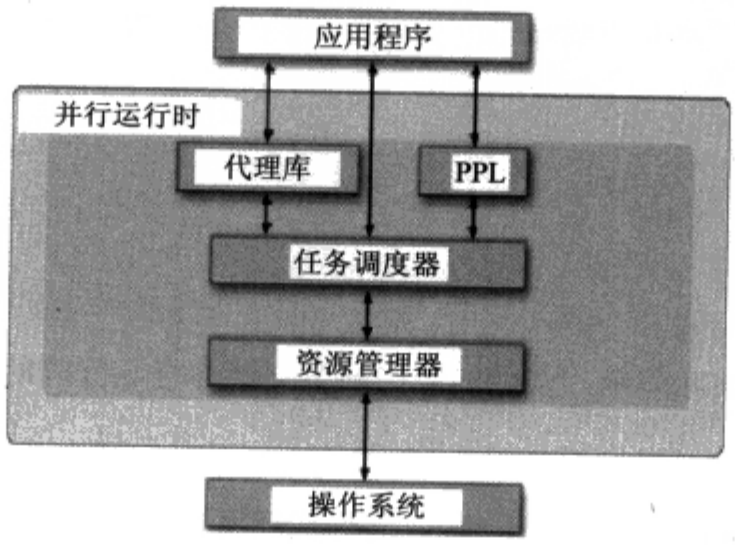


图 1.21　并行运行时体系结构

下面简要介绍各个模块的功能及何时使用它们。

1. PPL

并行模式库（PPL）为细粒度的并行提供了通用的容器和算法。PPL 通过提供将计算资源分布到集合或数据集的并行算法来达到数据并行（data parallelism），通过提供将多个独立的操作分布到计算资源的任务对象来支持任务并行。

任何可以从并行运行中受益的本地计算都可以考虑使用并行模式库。最简单的就是使用 parallel_for 算法来将程序中的 for 循环并行化。比如考虑两个矩阵相乘的例子，下面为串行实现：

```
void MatrixMult(int iSize, const vector<vector<int> >& matrix1, const vector<vector<int> >& matrix2, vector<vector<int> >& matrixResult)
{
    for (int i = 0; i < iSize; ++i)
    {
        for (int j = 0; j < iSize; ++j)
        {
            matrixResult[i][j] = 0;
            for (int k = 0; k < iSize; ++k)
            {
                matrixResult[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}
```

这里有 3 层循环，每一次循环都是访问独立的元素，循环之间没有任何数据依赖存在，所以完全可以将这个 for 循环并行，下面为使用 PPL 的 parallel_for 的实现：

```
void MatrixMultPar(int iSize, const vector<vector<int> >& matrix1, const vector<vector<int> >& matrix2, vector<vector<int> >& matrixResult)
```

```
{
    parallel_for(0, iSize, 1, [&](int i)
        {
            for (int j = 0; j < iSize; ++j)
            {
                matrixResult[i][j] = 0;
                for (int k = 0; k < iSize; ++k)
                {
                    matrixResult[i][j] += matrix1[i][k] * matrix2[k][j];
                }
            }
        }
    );
}
```

可以看到，实现并行化如此简单，其他的底层细节交给并行运行时处理。

2．异步代理库

异步代理库（Asynchronous Agents Library，简称代理库）同时为粗粒度的数据流及管道任务提供了基于行为的编程模型及消息传递接口，异步代理能够有效利用由于其他组件等待数据的延迟（在等待时为之分配额外的工作）。

当有多个需要异步通信的实体时，可以使用代理库。比如，可以创建一个实体用于从一个文件或网络连接上读取数据，然后使用消息传递接口将数据发送给其他代理。

3．任务调度器

任务调度器负责在运行时调度并协调任务的执行，任务调度器使用一种称为工作窃取的算法来实现对计算资源的最大利用。

并行运行库提供了一个默认的调度器，这样用户就不需要管理底层细节。但是当默认的调度器无法很好地满足特定应用时，并行运行时允许用户提供自己的调度策略以满足特殊需求。

4．资源管理器

资源管理器负责管理计算资源（如处理器、内存等）。资源管理器负责对运行时工作集的变化做出响应以动态分配资源，达到最高的效率。

资源管理器是计算资源上的一层抽象，主要与任务调度器互动。PPL、代理库及任务调度器使用资源管理器来随着工作集的变化而动态平衡资源的分配。

表 1.3 列出了并行运行时各个部件对应的头文件。

表 1.3

部　　件	头 文 件
并行模式库（PPL）	ppl.h concurrent_queue.h concurrent_vector.h
异步代理库	agents.h
任务调度器	concrt.h
资源管理器	concrtrm.h

1.9.2　调试和分析工具

并行程序的调试是一个典型的老大难问题，因为多线程带来的程序运行的不确定性，在 Visual Studio 以前的产品中，缺乏对并行调试和分析的工具，增加了程序员的调试难度。在 Visual Studio 2010 中加强了对并行调试的支持，为程序员提供了并行任务调试工具窗口和并行栈窗口，下面分别介绍如何使用这些工具进行并行调试和分析。

1．并行任务调试工具窗口（Parallel Tasks debugger toolwindow）

不管是托管（managed）还是本地（native）开发，Visual Studio 2010 都引入了一种新的被称为基于任务的编程模型，为此 Visual Studio 2010 提供了一种新的调试窗口：并行任务，专用于调试这类应用。在调试模式下，应用程序执行到某个断点（breakpoint）后就可以打开并行任务窗口，如图 1.22 所示。

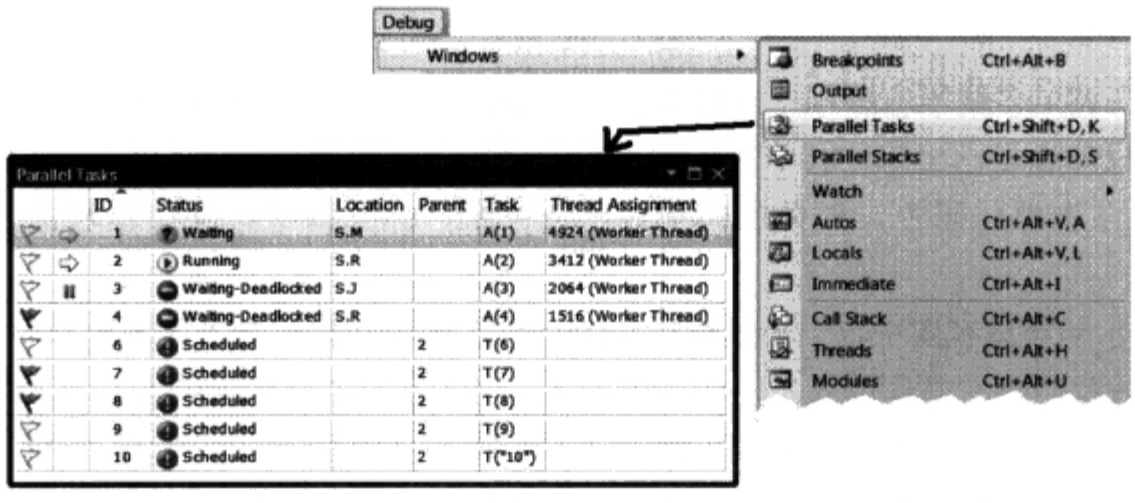


图 1.22　并行任务调试工具窗口

窗口中的各列之间可以互换顺序（通过将它们向左或向右拖动），而且同一列中的各个元素是可排序的（通过单击列名），列名框中的三角形显示排序的方式（如图 1.22 中 ID 列中的上三角形表示增序排列）。右击任意列，在弹出的下拉菜单中选中或不选某列表示在窗口中显示或隐藏对应列的内容。

如图 1.23 所示，还可以对任务按状态（Status）进行分组，通过分组可以对整组进行拖放操作，并显示出处于各种不同状态的任务个数，如图 1.24 所示。

下面分别介绍并行任务窗口中的几个主要列的作用。

- ID: 每个任务都有一个惟一的 ID，通过它可以指定将特定任务的输出导入到流中或者在不同调试窗口中交叉引用任务等。

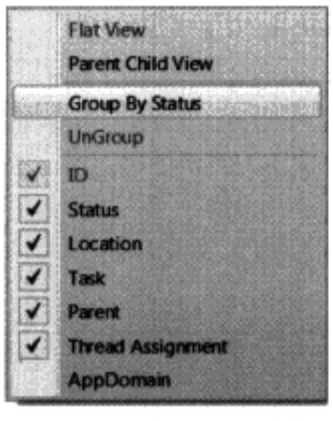


图 1.23　并行任务窗口的右键快捷菜单

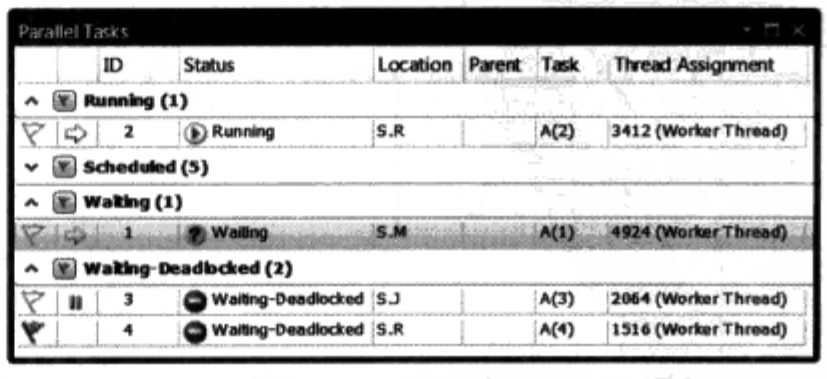


图 1.24　按状态分组的并行任务窗口

- Status: 每个任务都有状态，表明任务的调度情况，共有 4 种不同的状态：运行（Running）、被调度（Scheduled）、等待（Waiting）和死锁等待（Waiting-Deadlocked）。表 1.4 列出了这些

状态的含义。

表 1.4

状　态	含　义
运行	在调试器断点处，正在运行该任务的代码
被调度	相当于就绪，该任务被放在某个队列但是还未被线程执行过
等待	已经运行，但是由于等待某些资源而被阻塞
死锁等待	也是处于等待状态，而且被系统检测出存在等待环（可能出现死锁）

- Location：显示当前任务所处的方法名称。
- Task：显示任务的入口方法及调用参数。
- Parent：显示该任务的父任务 ID。
- Thread Assignment：运行该任务的线程 ID 及线程名。

2．并行栈窗口

并行栈窗口在调试多线程应用程序时特别有用，它的线程视图能够将应用程序中包含的所有线程的（函数）调用栈信息显示出来。其中在托管代码中，任务视图显示 System.Threading.Tasks.Task 对象的调用栈；而在本地代码中，任务视图则显示任务组、并行算法、异步代理及轻量级任务的调用栈。

在图 1.25 所示的线程视图中，一个线程从 main()函数进入函数 A 再进入函数 B，最后到达外部代码。而其他两个线程则拥有共同的入口，然后进入 A，但是其中之一继续进入 B 然后进入外部代码，将另一个线程则进入 C 然后执行一些匿名方法。

在图中当前线程的调用路径以蓝色高亮显示，而活跃的栈帧（stack frame）则用黄色箭头标记。

如果应用程序使用 System.Threading.Tasks.Task 对象（托管代码）或 task_handle 对象（本地代码）来实现并行，那么可以进入任务视图，任务视图显示任务的调用栈信息而不是线程。任务视图与线程视图的区别有：

- 那些不运行任务的线程的调用栈不会被显示；
- 在运行任务的线程调用栈的顶部和底部显示最相关的帧涉及的任务；
- 当多个任务位于同一个线程上时，这些任务的调用栈被分为分隔的的节点。

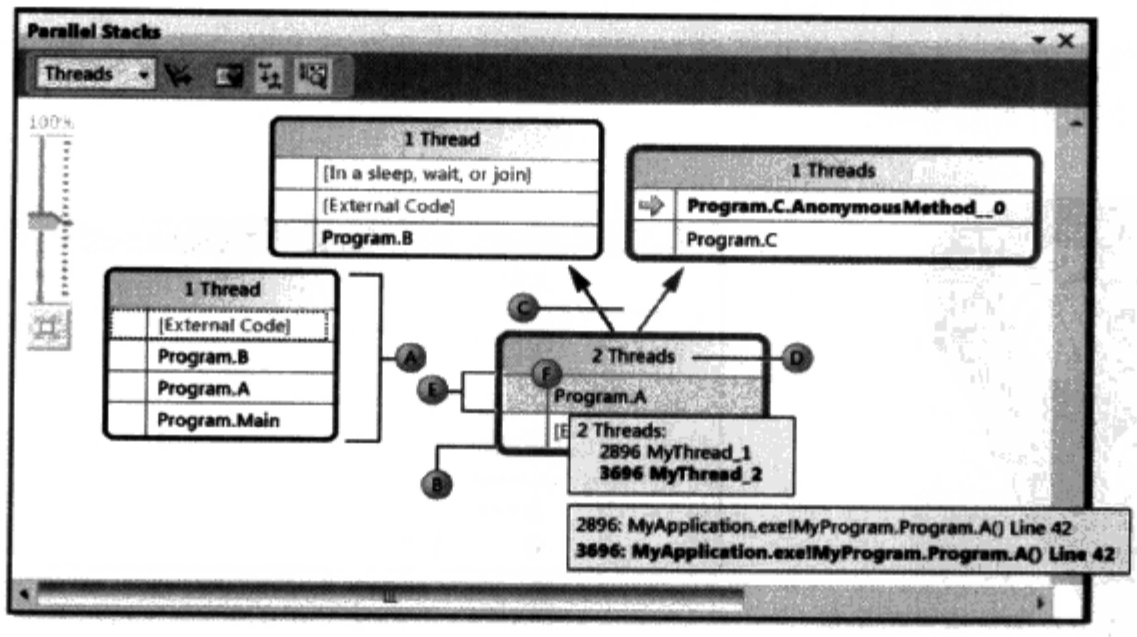


图 1.25　并行线程例子图

图 1.26 演示了并行栈窗口，右侧为任务栈视图，而左侧为对应的线程视图。

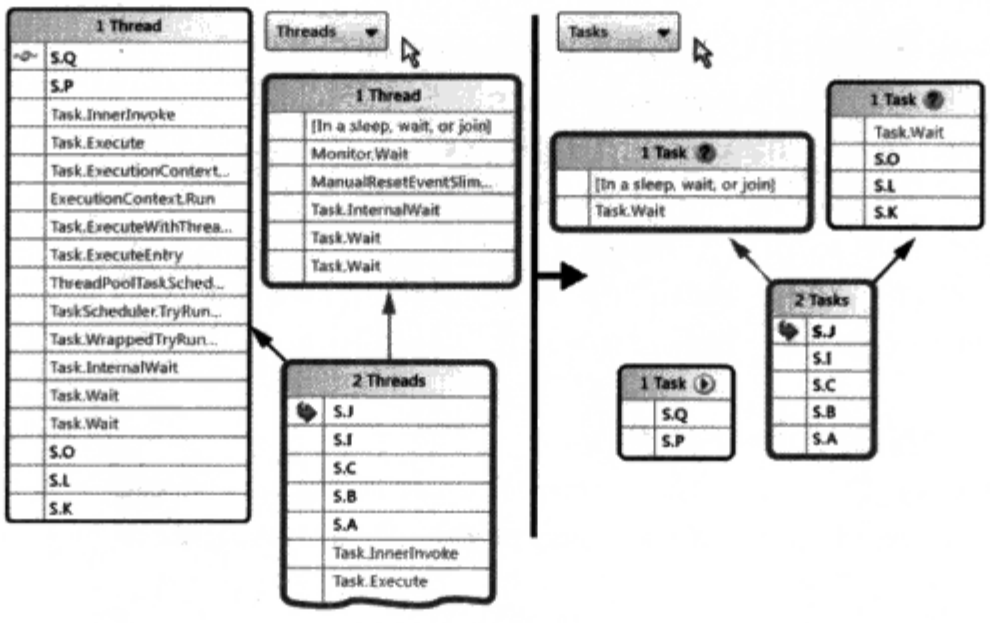


图 1.26　线程对应视图

1.10　对 MFC 的增强

1.10.1　任务对话框 CTaskDialog

任务对话框（CTaskDialog）是由 Vista 操作系统引进的，用于替换之前广泛使用的消息框（Message Box）。除了向用户显示基本的消息之外，任务对话框还允许设置下列元素：

- 窗口标题；
- 主图标；
- 主指令；
- 内容；
- 进度条；
- 单选框；
- 自定义按钮；
- 扩展按钮；
- 复选框；
- 页脚图标；
- 页脚文本；
- 一般按钮。

可以看到，任务对话框提供了很多可配置的选项，而最简单的配置看似与标准的消息框很像，在更高级的配置中可以支持进度条、自定义按钮及单选框等，设置可以支持在控件文本中定义一个 HTTP 链接或创建一个简单的向导。Visual Studio 2010 的 MFC 为开发人员提供了简单的机制来处理对话框事件。

CTaskDialog 类提供了一些简单的方法来创建并初始化控件，比如：

```
CTaskDialog taskDlg;
taskDlg.SetContent(_T("This is the main contents of the CTaskDialog.\n")
        _T("Of course it supports multiple lines and hyperlinks: <a href=\"calc.exe\">calc.exe</a>."));
taskDlg.SetMainInstruction(_T("This is the main instruction spread\nover two lines."));
taskDlg.SetWindowTitle(_T("The CTaskDialog Title"));
```

```
// 加入 3 个命令按钮
taskDlg.AddCommandControl(1, _T("Command 1 (ID=1)\nA second line."));
taskDlg.AddCommandControl(2, _T("Command 2 (ID=2)"));
taskDlg.AddCommandControl(3, _T("Command 3 (ID=3)\n(requires elevation)"), TRUE, TRUE);

// 加入 2 个单选框
taskDlg.AddRadioButton(4, _T("Radio button 1 (ID=4)"));
taskDlg.AddRadioButton(5, _T("Radio button 2 (ID=5)"));

// 加入一些可供用户选择显示或隐藏的解释消息
taskDlg.SetExpansionArea(_T("Some expanded explanation that can be shown/hidden with the expand/collapse button below.\n"), _T("Note also that hyperlinks are supported here as well: <a href=\"notepad.exe\">notepad.exe</a>."), _T("Expand this"), _T("Collapse this"));

// 配置页脚文本和页脚图标
taskDlg.SetFooterIcon(MAKEINTRESOURCE(IDI_INFORMATION));
taskDlg.SetFooterText(_T("This is some footer text.\nA second line of the footer.\nHyperlinks are also here supported: <a href=\"http://www.nuonsoft.com/\">www.nuonsoft.com</a>"));

// 设置任务对话框的主图标
taskDlg.SetMainIcon(m_hIcon);

// 向任务对话框添加一个进度条
taskDlg.SetProgressBarMarquee();
taskDlg.SetProgressBarRange(0, 100);
taskDlg.SetProgressBarPosition(0);

// 在任务对话框的底部添加一个复选框
taskDlg.SetVerificationCheckboxText(_T("Always show this dialog"));
taskDlg.SetVerificationCheckbox(TRUE);

int options = taskDlg.GetOptions();
options |= TDF_ENABLE_HYPERLINKS;
options |= TDF_CALLBACK_TIMER;
taskDlg.SetOptions(options);

// 显示任务对话框
INT_PTR iRes = taskDlg.DoModal();
```

上面的这段代码将创建一个如图 1.27 所示的任务对话框。

从 Windows Vista 操作系统才开始支持 CTaskDialog 类，更早版本的 Windows 系统将无法显示 CTaskDialog 对象。在使用任务对话框之前，需调用 CTaskDialog::IsSupported()方法来判断当前用户使用的系统是否支持任务对话框，当然标准 Windows 消息框在 Visual Studio 2010 中还是可用的。另外，CTaskDialog 类只对 Unicode 应用程序可用。

下面的代码演示了如何正确使用任务对话框：

```
if (CTaskDialog::IsSupported())
{
    ...                    //使用任务对话框
}
else MessageBox();         //使用标准消息框
```

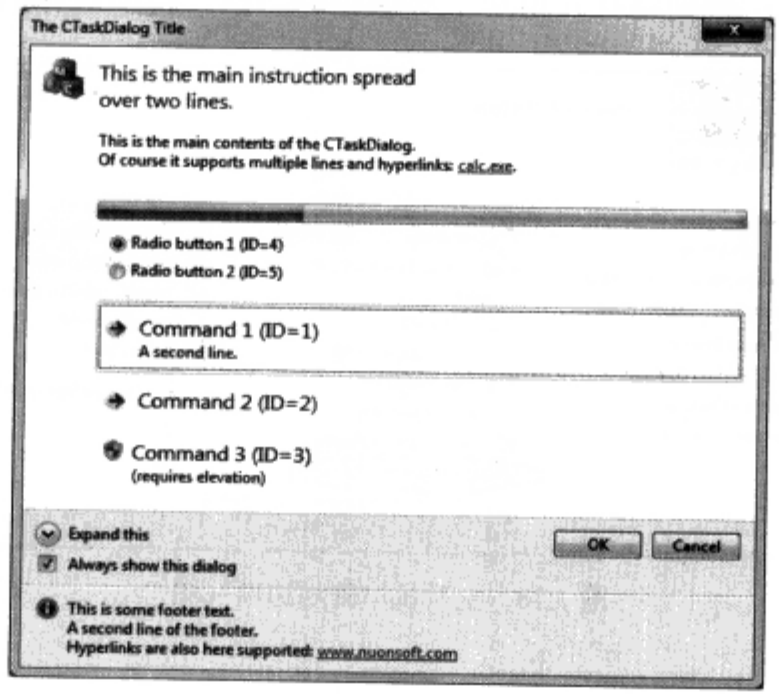


图 1.27　任务对话框视图

1.10.2　重启管理器（Restart Manager）支持

重启管理器是由 Microsoft 公司的 Windows Vista 操作系统引进的一项新技术，它可以帮助应用程序在下面几种情况下仍保持其数据（或设置）：需要将应用关闭的更新，出现不可预知的软硬件异常，程序崩溃等。不同于强制关闭应用，重启管理器使得应用程序能够在被强制终止之前执行保存操作（从而避免数据丢失），在重启应用程序时，可以将应用程序恢复到崩溃前的状态。

在 Visual Studio 2010 的 MFC 中，可以获取以下几方面的重启管理器的支持：

- 对于新建的 MFC 应用，可以从 MFC 应用向导免费获取应用程序的重启及自动修复特性；
- 所有重启管理器 API 中的可配置部分都通过可重载的虚拟成员提供给用户；
- 对已有的应用程序获取重启管理器支持，只需添加一行简单的代码即可。

在 Visual Studio 2010 中的 MFC 重启管理器中包含两种层次的支持。

（1）重启支持：在应用程序升级完毕或崩溃后自动重启（这个特性对所有类型的 MFC 应用都有效）。

（2）应用恢复支持：重新打开之前已经打开的文档；恢复自动保存的文档（这个特性只对文档/视图类型的 MFC 应用有效）。

对于新建的 MFC 应用程序和已存在的应用，获取重启管理器支持的方法是不同的。对于新的应用，只需要两步：① 使用 MFC 应用向导创建一个新的 MFC 应用程序；② 在高级特性设置页（Advance features page）中，选中“Support Restart Manager”复选框，如图 1.28 所示。

这里可以有 3 种不同的选项。

（1）只选中“Support Restart Manager”复选框：只提供对重启的支持，也就是说，在程序因升级关闭或崩溃之后能够自动重启，但是不会去重新打开之前已打开的文档或者恢复文档。

（2）同时选中“Support Restart Manager”和“Reopen previously open documents”复选框：除了支持自动重启外，还能够重新打开之前已打开的文档，但是不会恢复文档。

（3）选中（2）中的两个复选框，同时选中“Support application recovery”复选框：除支持自动重启和重新打开文档外，重启管理器还会试着将文档修复到自动保存的版本，通过弹出任务对话框（Unicode）或消息框（非 Unicode）的方式来提醒用户是否想恢复文档：如果单击“Yes”按钮，那么

将文档恢复到自动保存版本；否则将打开用户上次保存的文档，自动保存的变化将被丢弃。

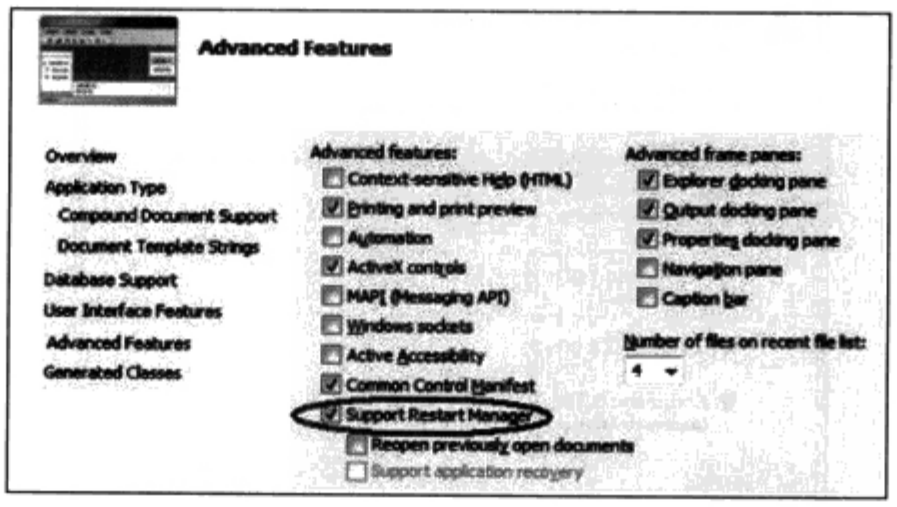
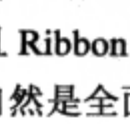


图 1.28　支持重启管理选项视图



（2）和（3）两项只对基于文档/视图类型的应用程序有效。

对于已存在的应用程序，只需要在应用程序类构造器内添加一行简单的代码（3 行代码分别对应上面 3 种不同的设置）：

```
（1）  m_dwRestartManagerSupportFlags = AFX_RESTART_MANAGER_SUPPORT_RESTART;
（2）  m_dwRestartManagerSupportFlags = AFX_RESTART_MANAGER_SUPPORT_RESTART_ASPECTS;
（3）  m_dwRestartManagerSupportFlags = AFX_RESTART_MANAGER_SUPPORT_ALL_ASPECTS;
```

比如下面的例子：

```
CMyApp::CMyApp()
{
    m_dwRestartManagerSupportFlags = AFX_RESTART_MANAGER_SUPPORT_RESTART;
    //其他初始化代码
}
```

1.10.3　支持新的用户界面风格

作为下一代用户界面，Ribbon 将为用户提供一种现代的方式，以便查找、理解和使用命令更加有效和直接，只需要简单的几个单击操作。

简单来讲，Ribbon 就是一个命令栏，将一个程序的特性组织成一系列的分栏放置在窗口的顶部。使用 Ribbon 为用户提供更直观的命令特性，使得用户更加容易学习程序，Ribbon 完全可以替代传统的菜单栏和工具栏。与传统的菜单式用户界面相比，Ribbon 界面的优势主要体现在如下几个方面：

- 所有功能有组织地集中存放，不再需要查找级联菜单、工具栏等；
- 更好地在每个应用程序中组织命令；
- 提供足够显示更多命令的空间；
- 丰富的命令布局可以帮助用户更容易地找到重要的、常用的功能；
- 可以显示图示，对命令的效果进行预览，例如改变文本的格式等；
- 更加适合触摸屏操作。

Ribbon 界面最早是在 Office 2007 中全面引入的，对于熟悉菜单式界面的用户而言，可能刚开始使用 Ribbon 会比较不适应，但是使用一段时间后，用户就能真切地体会到 Ribbon 带来的巨大便利。而且 Ribbon 界面势必成为未来的主流。Visual Studio 2010 作为面向下一代 Windows 平台的开发工具，自然是全面支持 Ribbon 界面。下面通过一个例子来演示如何在 MFC 应用程序中创建 Ribbon 界面应用。

首先创建一个MFC应用程序，在MFC向导中，允许用户选择不同的Ribbon风格。除了Visual Studio 2008已经支持的Office、Visual Studio及Windows Native风格外，在Visual Studio 2010中新加入了对Windows 7 Ribbon风格的支持，如图1.29所示。

图 1.29　Ribbon 风格视图列表

除了在MFC向导中可以设置Ribbon风格外，在应用程序运行后，还可以设置Ribbon UI的风格，通过在“Style”选项下重新选择一种即可。动态修改Ribbon风格只影响程序的外观，并不会对程序的行为产生任何影响。图1.30演示了在程序运行过程中选择新的Ribbon风格。

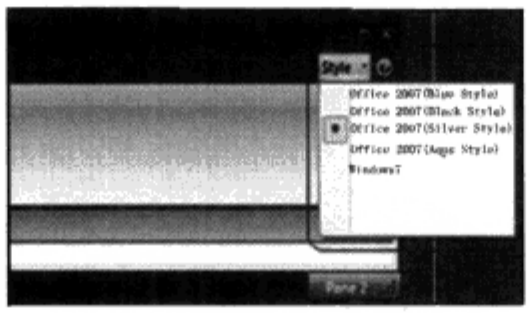


图 1.30　Ribbon 风格样式

在程序设计过程中，添加或删除 Ribbon 控件只需要简单地用鼠标拖动就可以实现，而为 Ribbon 工具添加行为（事件处理）也很方便。图1.31演示了所有可以在程序中使用的Ribbon控件。

在设计窗口的Toolbox中或者通过选择“View”→“Toolbox”命令进入工具栏设计，可以将Ribbon控件拖放到程序中需要的位置，还可以像添加其他资源（比如Dialog、Icon）那样通过“Add”“Resource”→“Ribbon”命令将Ribbon资源加入资源视图中。

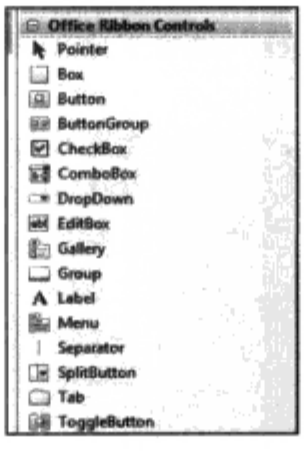


图 1.31　Ribbon 控件样式

可以在MFC应用程序中使用Ribbon资源，这在应用程序初始化过程中进行：

```
CMFCRibbonBar m_wndRibbonBar;                    //声明一个 Ribbon
if (!m_wndRibbonBar.Create (this))               //创建并初始化一个 Ribbon 控件
```

```
{
    return -1;
}
if (!m_wndRibbonBar.LoadFromResource(IDR_RIBBON))
{
    return -1;
}
```

可以在属性设计窗口中，改变 Ribbon 控件的属性，图 1.32 显示了如何设置一个 Ribbon 按钮的属性及对应生成的代码。

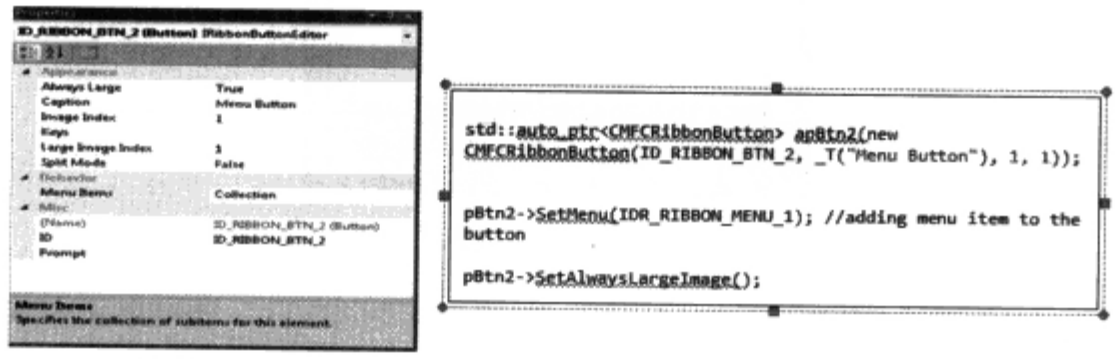


图 1.32　Ribbon 控件属性与代码生成

MFC 应用程序的 CMainFrame 类负责整个主框架界面的创建和管理，在 Ribbon 界面下，它同样负责 Ribbon 界面的创建和管理：

```
class CMainFrame : public CMDIFrameWndEx
{
…
protected:  // control bar embedded members
    CMFCRibbonBar m_wndRibbonBar;
    CMFCRibbonApplicationButton m_MainButton;
    CMFCToolBarImages m_PanelImages;
    CMFCRibbonStatusBar  m_wndStatusBar;
…
};
```

在 Ribbon 界面下，CMainFrame 类包含 3 个成员变量，即 m_wndRibbonBar、m_MainButton、m_wndStatusBar，分别用于控制 Ribbon 界面的命令面板、应用程序按钮和状态栏。

为了让 Ribbon 控件实现一定的功能，需要为 Ribbon 控件添加事件处理。与传统的菜单处理一样，右击 Ribbon 控件，选择“Add Event Handler”命令来添加事件处理。下面的代码简单演示了当用户单击 Ribbon 控件时，将弹出一个消息框：

```
// CRibbonUIView 事件处理
void CRibbonUIView::OnEditSayhello()
{
    // TODO: Add your command handler code here
    AfxMessageBox(_T("Hello RibbonUI!"));
}

void CRibbonUIView::OnEditSaygoodbye()
{
    // TODO: Add your command handler code here
    AfxMessageBox(_T("Goodbye RibbonUI!"));
}
```

第 2 章　MFC 应用程序框架

MFC 的英文全称是 Microsoft Foundation Classes，即微软的基本类库，它位于任何 Windows API 上，这个类库中包含 100 多个程序开发过程中最常用的对象。MFC 大量减少了在建立 Windows 程序时必须编写的代码，同时还具备所有一般 C++编程的优点，例如继承和封装。由于 MFC 编程方法充分利用了面向对象技术的优点，使得开发人员在编程时极少需要关心对象方法的实现细节，而只需要简单地调用已有对象的方法即可。即使类库中的那些对象的方法不能满足开发人员的需要，也可以利用面向对象技术中很重要的“继承”方法从类库中的已有对象派生出自己的对象，所派生出来的对象除了具有类库中的对象的特性和功能之外，还可以有开发人员根据需要加上的特性和方法，从而产生了一个更专门的、功能更强大的对象。这使得应用程序中程序员所需要编写的代码大为减少，有力地保证了程序的良好的可调试性。MFC 不仅给用户提供了 Windows 图形环境下应用程序的框架，而且还提供了创建应用程序的组件。

本章将详细介绍如何用 MFC 向导生成应用程序，以及生成的应用程序框架、框架对象的分析；另外，还将介绍一个 MFC 新功能的几种风格程序。

2.1　用 MFC 向导生成应用程序

可以用 MFC 和 Microsoft 向导编写 C++下的 Windows GUI 程序。MFC 和 Microsoft 向导不仅是 Visual C++中包括的最高级和最有趣的 Windows GUI 开发工具，而且也是最新的 Windows GUI 开发工具。

应用程序向导可以用来生成各种不同 GUI 程序的基本源文件，然后可以用 ClassWizard 工具生成派生类、定义处理消息和定制 MFC 性能的成员函数、管理对话框和完成其他任务的大部分过程代码。向导生成的代码充分利用了 MFC。注意，向导不仅可以生成简单程序框架，而且可以用于包含各种高级特性的程序，包括工具栏、状态窗口、上下文相关的联机帮助、OLE 支持、数据库访问及具有打开和保存文件、打印、打印预览和其他命令的完整菜单。一旦用向导生成了基本程序源代码，就可以立即将其加入自己的程序逻辑中。

用 MFC 向导生成应用程序，不仅会生成 MFC 预先编写的代码框架，而且能够利用生成的使用 MFC 和处理许多常规编程任务的源代码。MFC 和 Microsoft 向导解除了生成程序的直观接口所需的大部分工作，还保证这个接口符合 Microsoft 的准则。

2.1.1　生成的程序框架

什么是应用程序框架（application frame）？简单来讲，应用程序框架就是一个集合，而这个集合可以提供一般应用程序所需的全部面向对象软件组件。

一个应用程序框架是一个类库的超级集合。一般的库是设计集成到任何程序类的集合，但是一个应用程序框架不同于单纯的类的集合，它自己定义了程序的结构。

下面将通过使用 AppWizard 应用程序向导生成的一个最简单的程序，来使读者更好地理解什么是

应用程序框架。

为了用应用程序向导生成程序，要生成相应类型的新项目，然后指定应用程序向导显示的一系列对话框中所要的程序特性。首先要运行 Microsoft Visual Studio 2010，然后执行如下步骤。

（1）选择 Visual Studio 2010 的“File”→“New”→“Project”命令，或直接按【Ctrl+Shift+N】组合键，弹出“New Project”对话框，如图 2.1 所示。

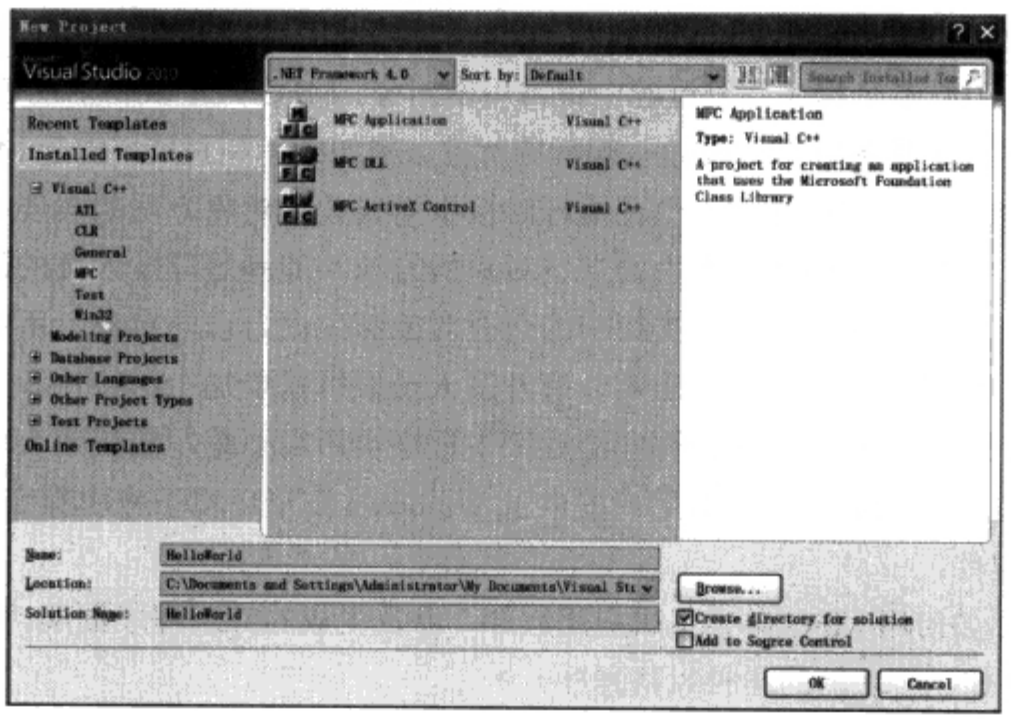


图 2.1　创建 MFC 程序

（2）在左侧的“Templates”窗格中，依次选择“Installed Templates”→“Visual C++”→“MFC”选项，此时中间窗格中指示将要创建的项目类型，从中选择“MFC Application”项目类型，按照应用程序向导提示输入详细信息，然后产生使用 MFC 的 Windows GUI 程序的基本 C++代码（如果要用应用程序向导生成动态链接库，则选择“MFC AppWizard(dll)”类型）。

（3）在“Name”文本框中为新建项目输入项目名，如“HelloWorld”，使 Visual C++将其赋予新项目（及该项目所在的项目工作区）。

（4）在“Location”下拉列表框中指定项目文件所在的文件夹（即项目文件夹）路径。一般而言，可以接受文本框中的默认文件夹（默认文件夹与项目工作区同名，为 HelloWorld）。单击“Browse”按钮，可以搜索其他位置。如果指定的项目文件夹不存在，则 Visual Studio 2010 会生成这个项目文件夹（并在项目文件夹中生成-Res 子文件夹，存放几个资源文件，并生成一个或几个输出子文件夹）。

（5）“Solution Name”默认与项目名相同。当取消选中“Create directory for solution”复选框时，不产生 Solution 文件夹。

（6）单击“OK”按钮，项目类型的向导随即打开。

（7）单击“Finish”按钮，不产生指定项目属性，或跳过这一步以使用 Visual Studio 的默认项目属性。若要自定义项目属性，则单击“Next”按钮，则弹出“Application Type”对话框，如图 2.2 所示。在需要选择的应用程序向导操作说明中，最初都选择了默认的选项。对于这些选项，只要确保不改变即可。

在“Application Type”对话框中，应用程序类型有：单文档（Single document）、多文档（Multiple documents）、基于对话框的应用程序（Dialog based）和顶级多文档（Multiple top-level documents）4 种。默认的是 Multiple documents，这里选择 Single document 应用程序类型，确保选中“Document/View architecture support”复选框（默认）并设置语言为英语（默认），其他的均按默认设置。需要用户了解的是，如果选择 Single document 应用程序类型，那么应用程序向导会产生一个单文档接口（SDI）

视图应用程序框架，一次只显示一个文档；如果选择的是 Multiple documents，那么应用程序会生成一个多文档-视图应用程序框架。其他应用程序类型将在 2.1.2 小节介绍。

Document/View architecture support 选项使应用程序向导关系到存放和浏览程序数据的类并提供读取或写入磁盘数据的代码。这个结构将在后面介绍（一般来说，除非想使程序长度最小化，否则均应选中“Document/View architecture support”复选框，以利用其支持的编程支持）。

最后，应用程序的程序菜单标题和标准消息都用所选语言显示。

若完成，则单击“Finish”按钮；否则单击“Next”按钮显示“Compound Document Support（复合文件支持）”对话框。

从这一步开始的任何应用程序向导对话框都可以单击“Previous”按钮返回上一步，浏览或修改前面的设置。也可以单击“Finish”按钮直接跳到最后，立即用剩余对话框的所有默认选项值产生程序源代码（本示例中暂不单击这个按钮）。单击右上角的问号（？）按钮，然后单击对话框中某个控件，可以取得关于有关选项的信息。单击“Next”按钮进入下一个应用程序向导对话框。

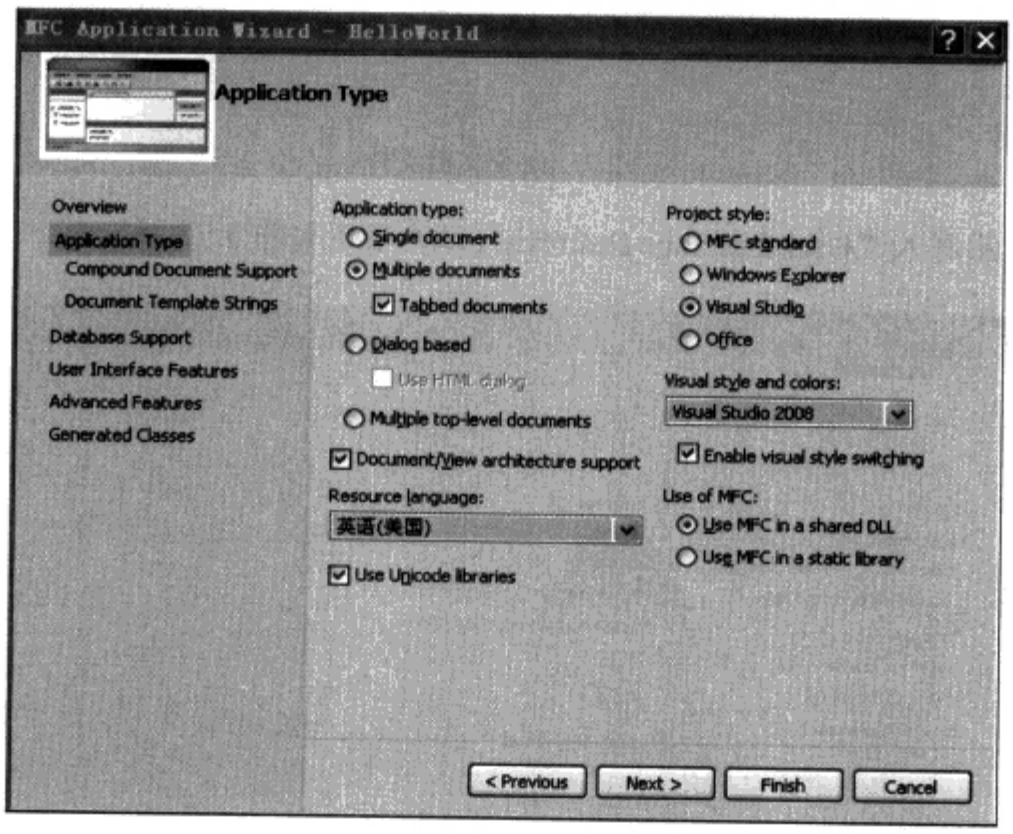


图 2.2　程序类型选项选择

（8）在“Compound Document Support”对话框中，设置用户应用程序支持何种复合文档：不支持复合文档（None）、容器（Container）、能够创建管理复合文档但 Mini-server 应用程序不能单独运行（Mini-server）、能够创建管理复合文档并且应用程序能单独运行（Full-server）、用户应用程序既是容器又是服务器（Container /Full server）。这里选择 None，其他设置保持默认值。选择“None”选项（默认）排除程序的复合文档支持，确保取消 Automation 选项以取消自动化支持。设置完成后，如图 2.3 所示。单击“Next”按钮进入“Document Template Strings（文件模板类）”对话框。

（9）在“Document Template Strings”对话框中，项目使用默认文档模板字符串的项目名称，每个对话框保留默认值，并单击“Next”按钮，进入“Database Support（数据库支持）”对话框。

（10）在“Database Support”对话框中，如图 2.4 所示，设置用户应用程序是否支持数据库：不支持数据库(None)、仅支持头文件(Header files only)、没有支持文件的数据库视图(Database view without file support)、带文件支持的数据库视图(Database view with file support)。这里选择 None。选择“None”选项（默认）排除程序中的数据库支持，关于用 Visual C++开发数据库应用程序的消息，可参考联机帮助。

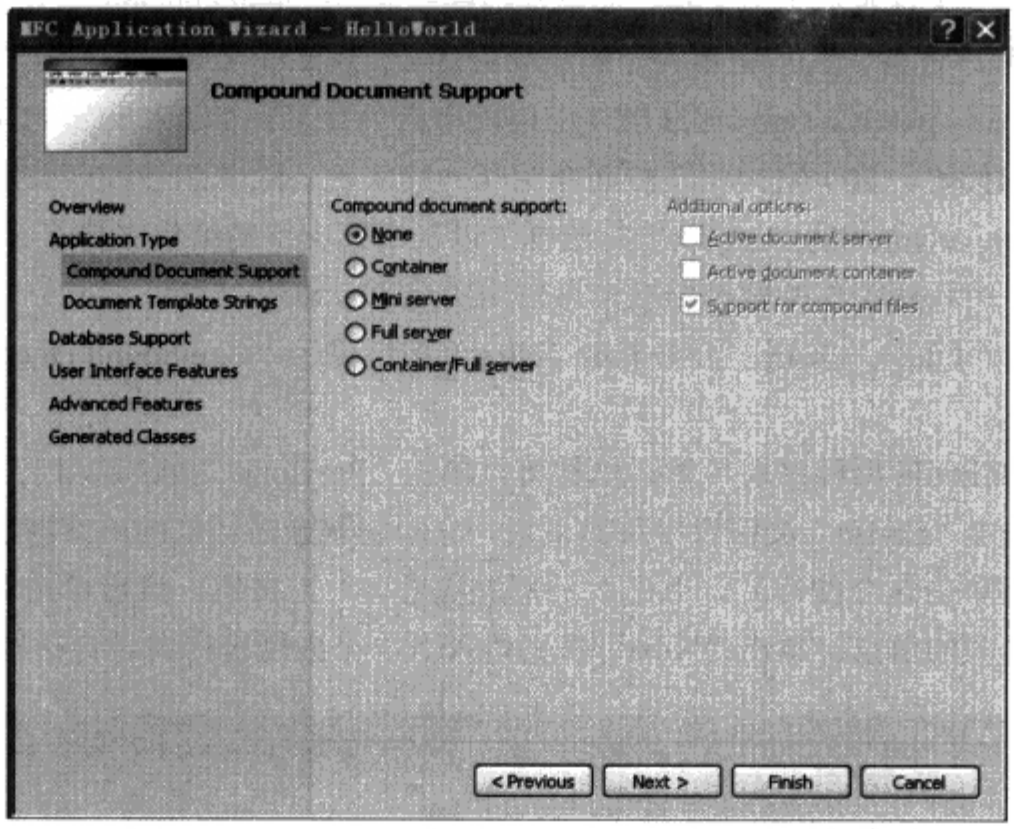


图 2.3　命令文档支持

单击“Next”按钮进入“User Interface Features（用户界面特征）”对话框。

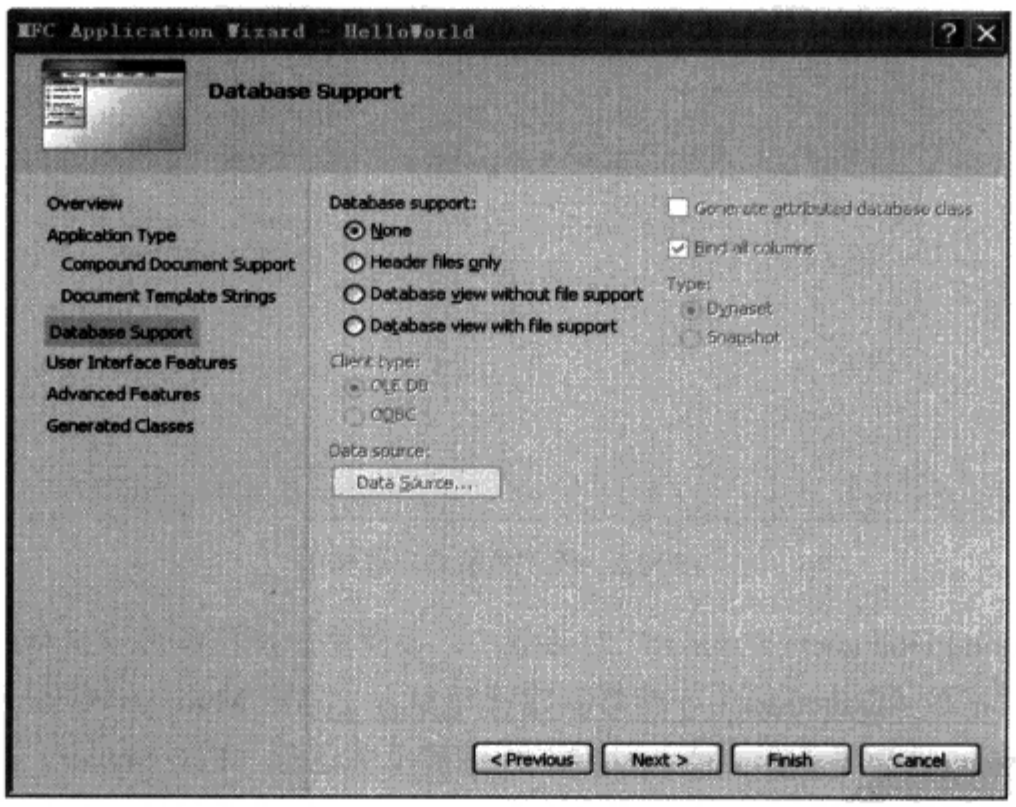


图 2.4　用户界面特征选择

（11）在“User Interface Features”对话框中，设置 MFC 标准项目样式选项（默认选项）产生程序的传统 MFC 用户接口。项目实现标准 Windows 用户界面功能，如系统菜单、状态栏、最大化和最小化框、“关于”框、标准停靠工具栏和子框架。

单击“Next”按钮进入“Advanced Features（高级特征）”对话框。

（12）在“Advanced Features”对话框中继续设置 MFC 标准项目样式选项（默认选项）：

项目支持打印和打印预览，项目支持 ActiveX 控件，项目不提供自动化、MAPI、Windows Sockets 或 Active Accessibility 支持。

单击“Next”按钮进入下一个应用程序向导对话框。

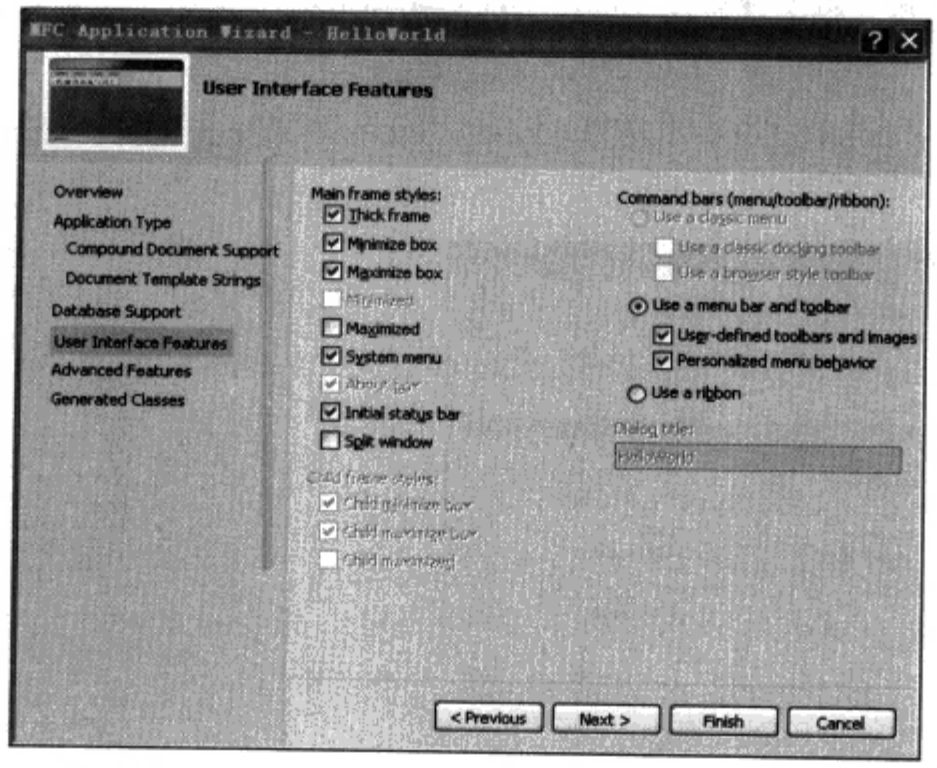


图 2.5　选择 ActiveX 支持

（13）在“Generated Classes”对话框中显示了应用程序向导为程序定义的 4 个基类的信息，如图 2.6 所示。对话框中的默认设置确定了类的名称及其所在文件的名称，用户可以改变名称，但本书建议使用向导自动生成的名称。在 Visual C++中，AppWizard 允许开发人员选用 MFC 类库中其他视图类作为应用程序视图类的基类，不一定非得是默认的 CView 类。这里，全部采用默认设置。不要改变这些设置，因为剩余的练习假定接受所有默认值。应用程序从下列 4 个基类派生。

CWinApp 是应用程序的“应用程序类”，负责初始化和运行应用程序。

CFrameWnd 是应用程序的“框架类”，负责显示和搜寻用户命令。

CDocument 是应用程序的“文档类”，负责装载和维护文档。文档可以是从文档到网络设备设置的任何内容。

CView 是应用程序的“视图类”，负责为文档提供一个或多个视图。

这是应用程序向导设置属性的最后一个对话框，这时应单击“Finish”按钮。

（14）单击“Finish”按钮后，应用程序向导会生成指定的项目文件夹，此时会关闭向导并在开发环境中打开新项目。生成程序源文件并打开新生成的项目 HelloWorld。创建项目后，可在解决方案资源管理器（Solution Explorer）中查看创建的文件。图 2.7 显示了项目工作区窗口 HelloWorld 项目的完全展开的 FileView 视图。

现在已经生成了一个使用 MFC 基本类库的应用程序基本框架（单文档-视图体系结构的应用程序框架），而且可以立即运行。运行之后，会发现这个应用程序框架具备 Windows 程序的所有特性：菜单、工具栏、状态栏、图标甚至还有关于对话框、打印预览……，这就是 AppWizard 通过 MFC 动态创建的一个应用程序框架。从这个例子可以看出，用 Visual C++/MFC 设计 Windows 程序是非常方便的。

为什么没有编写一句代码就生成了一个应用程序框架，而且没有 WinMain()函数和 WndProc()回调函数？其实，并不是没有 WinMain()函数和 WndProc()回调函数，也不是 Windows 程序结构发生了变化，这全都是 MFC 的功劳。这个应用程序框架的底层采用的仍然是 WinMain()函数和 WndProc()回调函数，只是它们被 MFC 封装在各个类中了，MFC 通过应用程序向导生成了这些类，开发人员所要做的就是往这个类中添加一些内容，如重载函数、消息处理等。应用程序向导生成的源文件足以建立功能性程序，也就是说，用应用程序向导生成的源文件可以立即建立并运行程序。但这样的框架程序通常只是一个基本框架，不能完成特定功能，因此通常要用不同的 Visual C++开发工具为特定应用程序

加入特殊的功能。MFC 程序框架就好比一座房子的架构和主梁，主梁和架构都有了，剩下要做的就是往里面添砖加瓦。

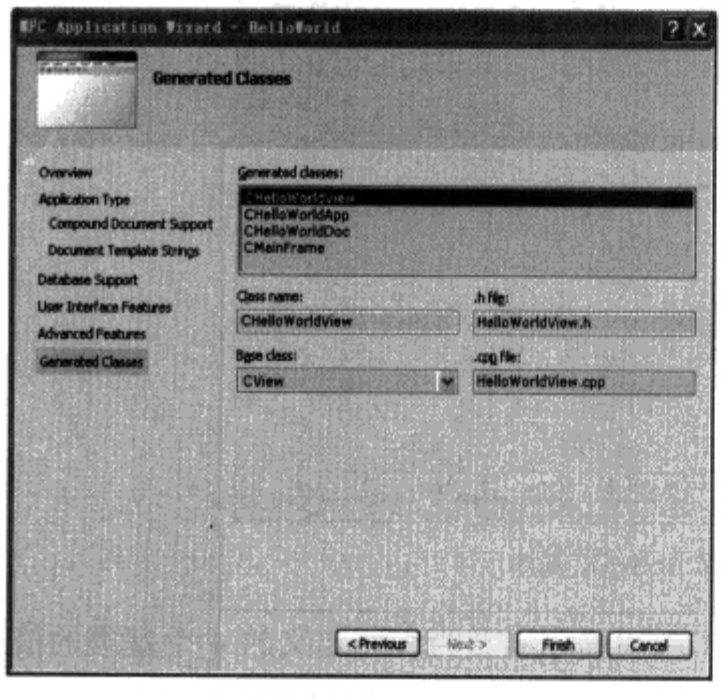


图 2.6　视图支持

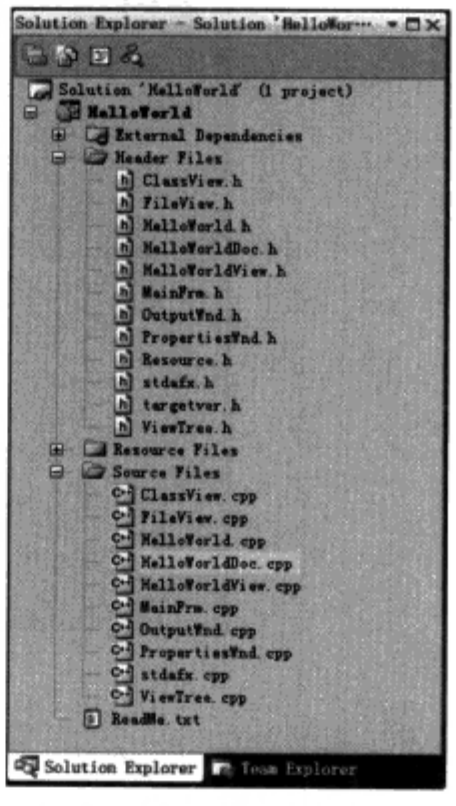


图 2.7　项目管理器视图

下面在生成的应用程序框架中输入一句代码，验证是否会在计算机屏幕上显示项目工作区窗口。在工作台中的 Solution Explorer 中选择 Source Files 中的 HelloWorldView.cpp 文件，并双击展开，在其中的 OnDraw()成员函数中输入如下一句代码：pDC->TextOut(0,0,_T("Hello World!"))；。OnDraw()成员函数如例 2.1 所示。

例 2.1　OnDraw()成员函数

```
void CHelloWorldView::OnDraw(CDC* pDC)
{
    pDC->TextOut(0,0,_T("Hello World!"));
    CHelloWorldDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;

    // TODO: add draw code for native data here
}
```

编译运行，结果如图 2.8 所示，显示了“Hello World”字样，这就是 MFC 应用程序框架的优点。

图 2.8　Hello World 应用程序运行结果

2.1.2　生成的应用程序类型

从整体来看，通过 Visual C++ 2010 可以生成 5 种类型的应用程序，分别为控制台应用程序、基于对话框的应用程序、单文档应用程序、多文档应用程序和基于 HTML 的应用程序。

1．控制台应用程序

适用于开发人员真正需要与已有系统保持某种兼容性或不需要为用户提供全功能操作界面的情况。

控制台应用程序具备 DOS 风格的窗口外观，而不是 Windows 风格的窗口外观。控制台应用程序使用等距字体，就像在 DOS 窗口中看到的那样，在控制台应用程序中可以使用标准 C 函数完成输入输出，如 printf 和 scanf()。然而，从内部来讲，控制台应用程序确实是一个 Windows 应用程序。更值得强调的是，使用控制台应用程序可以把某些功能型 DOS 代码转移到 Windows 系统之下。

为什么要创建这种应用程序呢？假设有一些工作得很好的老的代码，但它们是 DOS 下的应用程序。若要把这个程序迁移到 Windows 系统下，但又没有时间把所有代码都转换成 Windows 调用，那么控制台应用程序提供了一种折中的解决办法。在这类应用程序中，可以使用老的代码（至少是其中的一部分），但在 Windows 环境中执行这一程序。

从长远来看，创建控制台应用程序并不一定会节省时间和精力，这是因为它不能保证让开发人员使用所有现存代码。

使用控制台应用程序还有另外一个优点，当把应用程序从 DOS 搬到 Windows 中时，外观上就不一样，没有什么办法可以填平两者之间的鸿沟，从用户的观点来看，引入了一个完全不同的应用程序。另外，控制台应用程序外观上与 DOS 应用程序十分相似，因此并不需要花费多少精力进行配置，惟一需要做的就是习惯以不同方式运行应用程序。

2．基于对话框的应用程序

通常是实用程序的专利，也适用于极小型的不需要菜单系统的应用程序。用户界面只有一个对话框，没有框架类、文档类或视图类。对话框应用程序只使用应用程序类的派生类——CwinApp，对话框用 MFC 的 CDialog 类创建，如图 2.9 所示。

应用程序类	对话框类		
系统创建应用	应用程序类创建	对话框创建其他控件类，	没有创建框架、
程序类的实例	对话框类的实例	这些类创建屏幕上的窗口	文档或视图类

基于对话框的应用程序的程序规模较小，比较节省资源。此外，对话框版本的应用程序的加载速度也更快。基于对话框的应用程序比完成相同功能的基于窗口的应用程序更简单、有效。

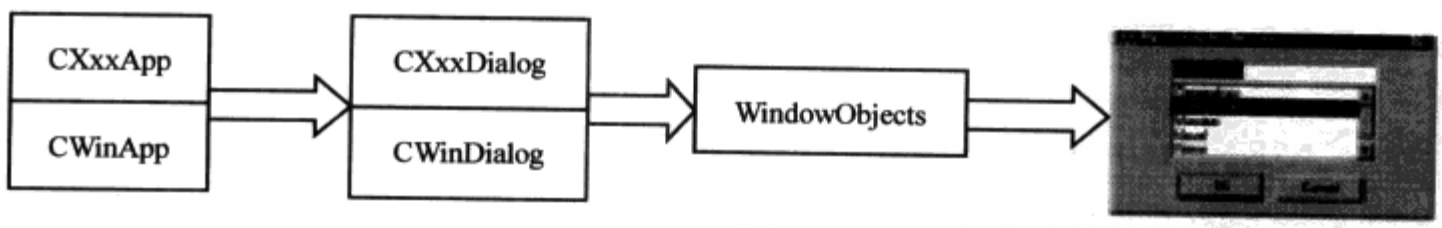


图 2.9　基于对话框应用程序结构

另外，创建基于对话框的应用程序的速度也很快。基于对话框的应用程序的特点就是规模小、效率高。当发现需要在这类应用程序中增加大量功能和特性时，或许说明一开始就选错了要创建的应用程序的类型。基于对话框的应用程序通常避免使用菜单和其他基于窗口应用程序为提供友好界面所需的部

件，该特性减少了程序员的开发和调试时间。显然，任何有利于提高程序员效率的方法都值得一试。

对话框应用程序惟一的缺点是某些程序员感到难以划分对话框。基于对话框的应用程序看上去会比同等功能的基于窗口的应用程序拥挤一些。解决办法是，就像 Windows 中用于配置的属性对话框那样，可以使用选项卡来降低基于对话框的应用程序界面的拥挤程度。

3．单文档应用程序

适用于操作自有数据的简单应用程序，比如记事本、画笔或小型数据库前端应用。这类应用程序也需要某种类型的菜单系统。能在某个时间内装入和编辑一个文档，使用了前面提及的所有 4 个基类。单文档应用程序包括以下类：从 CWinApp 派生的应用程序类、从 CFrameWnd 派生的框架类、从 CDocument 派生的文档类，以及每个文档的一个或多个视图类。这些视图类是从一些派生自 Cview 类的视图类派生的（如图 2.10 所示）。

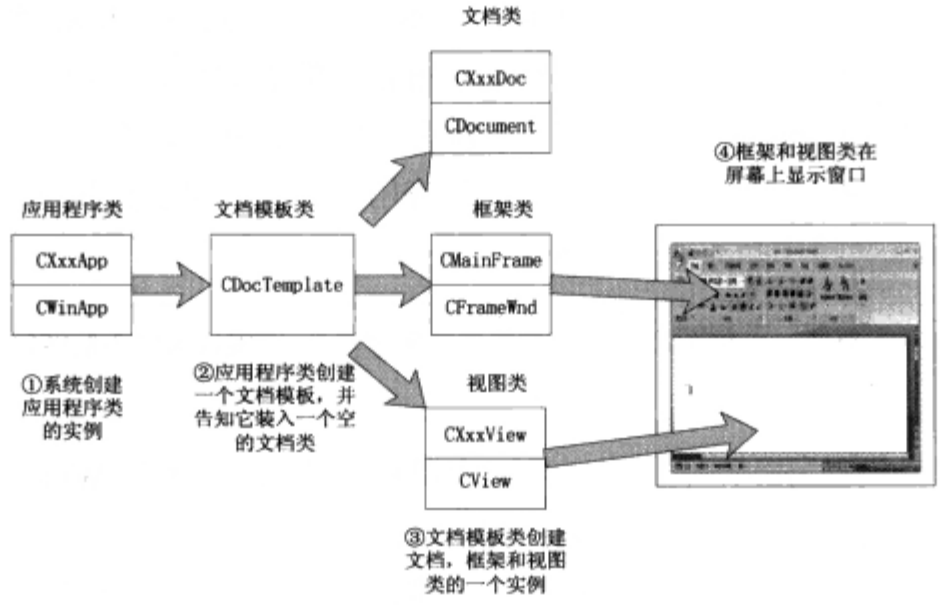


图 2.10　应用程序类型演示

单文档应用程序一次处理一个文档，降低了编程的复杂度并减少了运行程序时所需的资源。对某些小型应用程序（比如文本编辑器或小型图像编辑器），可以采用这种类型的窗口应用程序。单文档应用程序允许用户与其创建的文档进行全面的交互，但通常不如多文档应用程序耐用。另外，单文档应用程序至少要比多文档应用程序少一个菜单——就是用于选择要编辑文档的 Windows 菜单。

对只需要用户进行少量交互的小型应用程序来说，可以采用单文档、基于窗口的界面。

与基于对话框的应用程序相似，单文档应用程序也可以创建成 OLE 服务器。实际上，这类应用程序也可以作为 OLE 客户程序，尽管极少有程序员把这种能力添加到其应用程序中。

通过把应用程序的基视图类选择为 CHtmlView，可以把单文档应用程序转换成简单的 Web 浏览器。

单文档应用程序与基于对话框的应用程序的共同的缺点是：在同一时间，只能打开一个文档，若要查看一篇文档，需要在查看它之前先关闭当前打开的文档。

但是，当操作数据库管理系统时，单文档、基于窗口的应用程序工作得相当完美，其原因相当简单，极少有用户需要同时打开多个数据库。即使需要同时打开多个数据库，数据库本身的使用规则也减小了用户本身访问多个数据库的可能性。正常情况下，需要以可编程方式控制对各种数据库元素的访问，并把结果显示给用户。

4．多文档应用程序

多文档界面(MDI)应用程序包括以下类：派生于 CWinApp 的应用程序类、派生于 CMDIFrameWnd 的框架类、派生于 CMDIChildWnd 的一个或多个子框架类、派生于 CDocument 的每个子框架一个文档类和派生于 CView 的每个文档一个或多个视图类（如图 2.11 所示）。

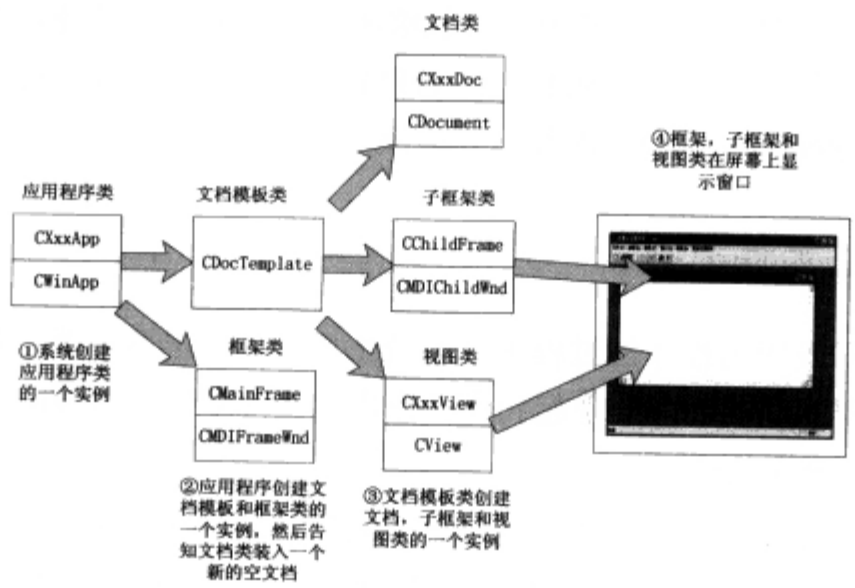


图 2.11　多文档应用程序结构

使用这种类型的基于窗口的应用程序可以创建功能完整的应用程序，如字处理程序或电子表格，例如，Microsoft Word 和 Microsoft Excel 都是多文档应用程序的示例。文本编辑器具备有限吸引力的原因正是其一次只能打开一个文档。人们需要在文档之间进行比较，这就是多文档应用程序占有一席之地的原因。

通过把应用程序的基视图类选择为 CHtmlView，可以把多文档应用程序转换成简单的 Web 浏览器。

多文档应用程序的同时处理多个文档的能力也意味着需要更多的额外编程工作。开发人员并不只是要跟踪所有打开的文档，也要提供 Windows 菜单来管理特殊的程序特性，比如要考虑屏幕划分问题。开发人员还需要决定是否允许用户同时显示多个文档。最小化其中一个文档，而最大化另一个文档这样的任务也需要额外的程序代码。总而言之，在开始编写多文档应用程序之前，就需要做大量的编程准备工作了。

由于多文档应用程序是 C++编程中十分复杂的一部分，因此，考虑建立这类应用程序时，应该在 Visual C++的灵活性与诸如 Visual Basic 之类的快速应用开发工具提供的速度之间进行权衡。多文档应用程序一次可以装载和编辑多个文档，不但使用了所有 4 个基类，而且还使用了 CFrameWnd 的两个派生类，即 CMDIFrameWnd 和 CMDIChildWnd。

5．基于 HTML 的应用程序

基于 HTML 的应用程序是 Visual C++新增的应用程序类型，它们操作某种类型的数据（像单文档或多文档应用程序那样），但与 Internet 结合在了一起。作为标准编辑器的替代物，用户会看到 Web 浏览器风格的前端应用。

创建基于 HTML 文档的应用程序正如创建自己定制的 Web 浏览器一样，可以把这个浏览器设置成自动浏览公司 Web 站点并限制用户访问 Web 上非商务站点。由于定制的浏览器不需要具备全功能浏览器所有的通用功能，因此，定制浏览器会减少内存需求和磁盘空间需求。换句话讲，可以建立一个提供浏览器所有功能、而又不存在它们所含问题的程序环境（至少在访问公司 Web 站点时是这样）。

这种新型应用程序的价值并不仅仅局限于此。例如，可以把 CHtmlView 类添加到现存应用程序中，让它能够访问基于 Web 服务器的帮助桌面。作为创建标准帮助文件并把它添加到应用程序的一种替代方法，开发人员可以创建十分专业的 Web 浏览器，并将其置入应用程序中。

基于 HTML 的帮助则既不需要编译器，也不需要特殊工具，只要有个文本编辑器即可（理论上说，在编写巨型帮助文件时，需要一个专门为操作 HTML 而设计的编辑器）。但是，HTML 帮助也有一些缺陷，一方面，难以在基于 HTML 的帮助中建立能够满足需要的查询功能，由于查询用户所需的信息与建立这些信息处于同等重要的位置，因此基于 HTML 的帮助并不适合初级用户；另一方面，基于 HTML 的帮助必须建立 Internet（至少为内部网）链接。

2.1.3 向导生成工程文件简介

在使用 Visual C++开发应用程序的过程中，系统生成了大量的各种类型的文件，本节将要详细介绍 Visual C++中这些不同类型的文件分别起到什么作用，在此基础上对 Visual C++如何管理应用程序所使用的各种文件有一个全面的认识。

首先要介绍的是扩展名为.dsw 的文件类型，这种类型的文件在 Visual C++中是级别最高的，称为 Workspace 文件。在 Visual C++中，应用程序是以 Project 的形式存在的，Project 文件以.dsp 为扩展名，在 Workspace 文件中可以包含多个项目，由 Workspace 文件对它们进行统一的协调和管理。

与.dsw 类型的 Workspace 文件相配合的一个重要的文件是以.opt 为扩展名的文件，这个文件中包含的是 Workspace 文件中要用到的本地计算机的有关配置信息，所以这个文件不能在其他计算机上共享。当打开一个 Workspace 文件时，如果系统找不到需要的.opt 类型文件，就会自动创建一个与之配合的包含本地计算机信息的.opt 文件。

上面提到 Project 文件的扩展名是.dsp，这个文件中存放的是一个特定的工程，也就是特定的应用程序的有关信息，每个工程都对应一个.dsp 类型的文件。

以.clw 为扩展名的文件是用来存放应用程序中使用的类和资源的信息的，这些信息是 Visual C++中的 ClassWizard 工具管理和使用类的信息来源。

每个应用程序都对应有一个 readme.txt 文件，这个文件中列出了应用程序中用到的所有文件的信息，打开并查看其中的内容就可以对应用程序的文件结构有一个基本的认识。

在应用程序中大量应用的是以.h 和.cpp 为扩展名的文件，以.h 为扩展名的文件称为头文件，以.cpp 为扩展名的文件称为实现文件，一般说来，以.h 为扩展名的文件与以.cpp 为扩展名的文件是一一对应配合使用的，以.h 为扩展名的文件中包含的主要是类的定义，而以.cpp 为扩展名的文件中包含的主要是类成员函数的实现代码。

在应用程序中经常要使用一些位图、菜单之类的资源，Visual C++中以.rc 为扩展名的文件称为资源文件，其中包含了应用程序中使用的所有的 Windows 资源，要指出的一点是，.rc 文件可以直接在 Visual C++集成环境中以可视化的方法进行编辑和修改。

最后要介绍的是以.rc2 为扩展名的文件，它也是资源文件，但是这个文件中的资源不能在 Visual C++的集成环境下直接进行编辑和修改，而是由开发人员自己根据需要手工地编辑这个文件。

以.ico，.bmp 等为扩展名的文件是具体的资源，产生这种资源的途径很多。使用.rc 资源文件的目的就是对程序中用到的大量的资源进行统一管理。

2.2 应用程序框架分析

MFC 库是开发 Windows 应用程序的 C++接口。MFC 提供了面向对象的框架，程序开发人员可以

基于这一框架开发 Windows 应用程序。MFC 采用面向对象设计，将大部分的 Windows API 封装到 C++类中，以类成员函数的形式提供给程序开发人员调用。

本节将对 AppWizard 自动生成的应用程序框架的内容进行详细的介绍，让读者对 MFC 方式的 Windows 应用程序的工作原理有全面、详细的认识，同时学会如何向程序框架中添加需要的代码。

2.2.1　框架简介

以 2.1 节中建立的应用程序框架为例，对应用程序框架进行介绍。

打开“Class View”工作台，可以看到系统在生成应用程序框架的过程中，自动生成了这些类：CHelloWorldApp、CMainFrame、CHelloWorldView、CHelloWorldDoc 等。这些类名称中的 HelloWorld 并不是固定的类名，而是为所建工程项目起的名称。例如，建立的工程名为 abc，那么系统生成的类名分别为 CabcApp、CMainFrame、CabctView、CabcDoc。如图 2.12 所示。

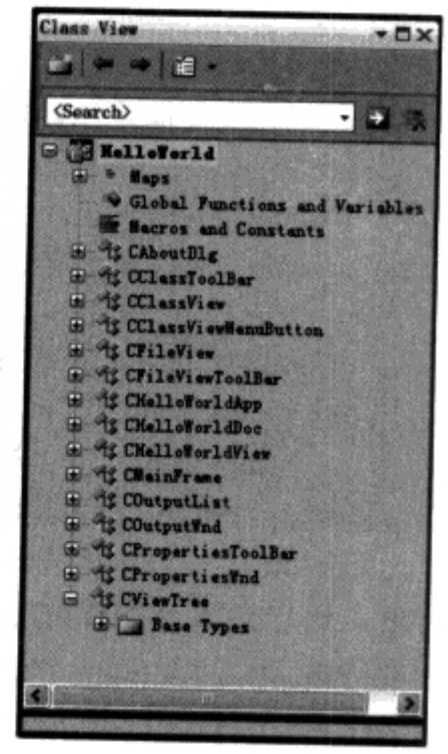


图 2.12　任务管理器

前面已经提到过，在应用程序中有一个重要的函数 WinMain()，这个函数是应用程序的基础，用户的操作所产生的消息经过这个函数的处理被派送到对应的对象中进行处理。在 MFC 方式的 Windows 应用程序中，用来处理消息的是系统自动生成的 MFC 中的类 CWinApp 的派生类 CHelloWorldAPP，下面就从这个类开始介绍应用程序的框架。

1．类 CHelloWorldApp（基类为 CWinApp）

若想创建 MFC 应用程序，首先要使用应用程序类 CWinApp，这是因为 CWinApp 类不仅代表了程序中运行的主线程，而且代表了应用程序本身，它是应用程序开始后创建的第一个对象，而且是在结束前执行的最后一个对象。启动时，应用程序类负责创建应用程序的其余对象：

- 对于一个对话框应用程序，CWinApp 类应用 Cdialog 类创建一个对话框；
- 对于一个 SDI 应用程序，CWinApp 类创建一个或多个文档模板，然后用模板打开一个空文档；
- 对于一个 MDI 应用程序，在主框架类中，应用程序类创建一个或多个文档模板，然后用模板打开一个空文档。

在任何 MFC 应用程序中只有一个 CWinApp 对象，它用于控制应用程序的初始化、应用程序的启动、主消息循环的运行、应用程序的终止。用 AppWizard 向导创建一个应用程序框架之后，应用程序框架自动生成了惟一的应用程序对象 theApp。从表面上看，它仅是某个类的一个实例，实际上，它内部隐藏了许多细节，如隐藏了程序最开始执行的 WinMain()函数。

CWinApp 类有 3 个重要的成员函数，分别为 InitInstance()、Run()、ExitInstance()，其中 InitInstance()成员函数是必须被重载的成员函数，AppWizard 向导生成的应用程序框架中的类 Chello WorldApp 便重载了该函数，它负责进行特定应用程序的初始化，包括装入 MRU 列表；Run()函数是用来处理消息循环的，不建议用户去重载它，该函数在后面讲述消息的章节中会有详细的介绍；ExitInstance()函数是做收尾工作的，应用程序通过调用该函数来结束运行，用户可以重载该函数以完成特殊的清除要求，如释放内存。

InitInstance()成员函数是必须被重载的成员函数，由于 CHelloWorldAPP 就是由 MFC 中的类 CWinAPP 派生来的，因此 CHelloWorldAPP 不仅是 AppWizard 向导自动生成的应用程序运行的基础，而且在这个类中除了具有一般类都有的构造函数之外，还有一个重要的成员函数，即 InitInstance()。众所周知，在 Windows 环境下可以运行同一程序的多个实例，函数 InitInstance()的作用就是在生成一个

新的实例的时候，完成一些初始化的工作。当生成一个 CTestApp 类的对象的时候，系统会主动调用 InitInstance()函数完成一些必要的初始化工作。

应用程序类的另一个主要的功能就是通过文档模板 CDocTemplate 将框架窗口类、文档类及视图类联系起来。应用程序打开一个文档时，文档模板定义创建什么样的框架、文档和视图。文档模板类是抽象类，因而不能直接使用，但可以使用它的两个派生类：针对单文档的 CSingleDocTemplate 类和针对多文档的 CMultiDocTemplate 类。要创建一个文档模板，要么为 SDI 应用程序创建一个 CSingleDocTemplate 类，要么为 MDI 应用程序创建一个 CMultiDocTemplate 类，并且用 3 个类的指针对类进行初始化。此功能也由应用程序类的 InitInstance()函数实现。

下面研究 InitInstance()函数所做的事情，注意例 2.2 的代码。

例 2.2　InitInstance()函数

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(CHelloWorldDoc),
    RUNTIME_CLASS(CMainFrame),          // main SDI frame window
    RUNTIME_CLASS(CHelloWorldView));
if (!pDocTemplate)
    return FALSE;
AddDocTemplate(pDocTemplate);
```

它定义了一个文档模板对象指针 PDocTemplate，通过 new 操作符，系统动态生成了这个文档模板对象，然后使用 AddDocTemplate()函数把这个文档模板对象加入应用程序所维护的文档模板链表中，文档模板 pDocTemplate 的作用就是把程序用到的框架窗口、CMainFrame、文档 CHello WorldDoc、视窗 CHelloWorldView 与应用对象 CHelloWorldApp 联系起来。其中，RUMTIME_CLASS 宏返回一个指向类的 CRuntimeClass 结构的指针，该结构用 DECLARE_DYNCREATE 和 IMPLEMENT_ DYNCREATE 宏填加到类中。文档模板通过用 CRuntimeClass::CreateObject()函数创建一个所有 3 个类的实例，来打开一个文档。

CHelloWorldApp 类的基类 CWinApp 自身是从 CWinThread 派生的。CWinThread 类封装了系统中用来创建和维护应用程序线程的 Windows API 函数。实际上，可以通过创建 CWinThread 类的另一个实例来实现应用程序多任务。在应用程序中，CWinApp 类代表主要的执行线程。此外，CHelloWorldApp 类提供了用户与 Windows 应用程序之间进行交流的界面。在生成这个类的对象后，这个对象自动在自身与 Windows 系统之间建立联系，用以接收 Windows 传送的消息，并交给程序中相应的对象去处理，这就免去了程序员的许多工作，使得开发 C++的 Windows 程序变得简单方便。

2．类 CMainFrame（基类为 CFrameWnd）

类 CMainFrame 是从 MFC 中的 CFrameWnd 基类派生来的，此框架类 CFrameWnd 是应用程序运行时创建的第二个对象，负责显示和监督用户对应用程序其余部分的操作，所以 CMainFrame 也是一个框架窗口。主框架窗口类负责标题栏、菜单栏、工具栏及状态栏的生成。事实上，用户是可以按照自己的意图自定义这些子窗口的，只要修改主框架窗口的实现文件即可。CMainFrame 是 CHelloWorldView 的基类，即视窗 View 显示在主框窗口 MainFrame 的客户区中。在类 CMainFrame 中，系统已经从类 CFrameWnd 那里继承了处理窗口的一般事件的 Windows 消息，比如改变窗口的大小、窗口最小化等的成员函数，因此编程的时候程序员不需要再关心此类消息的处理，从而减轻负担。当然，如果确实需要重新编写处理此类消息的成员函数，则需要对原有的成员函数进行重载，编写新的功能函数。

对于一个 SDI 应用程序，框架类是从 CFrameWnd 派生的，AppWizard 自动命名为 CMainFrame。

对于一个 MDI 应用程序，框架类派生于 CMDIFrameWnd 类，AppWizard 自动命名为 CMainFrame。在 MDI 应用程序中，每个打开的文档都有一个子框架类，每个子框架类是从 CMDIChildWnd 类派生的。AppWizard 自动指定子框架类名为 CChildFrm。

对一个对话框应用程序而言，没有框架类。在前面已提到过，一个对话框应用程序是由一个应用程序类和一个对话框类组成的。

图 2.13 是 MFC 类组织结构图的一部分，可以发现 CFrameWnd 是由 CWnd 派生的，另外，从 CWnd 派生的类还有 Cview，这说明这个程序中的 CMainFrame 类和 CHelloWorldView 类有一个共同的基类：CWnd 类。CWnd 类是 MFC 中的一个非常重要的类，它封闭了与窗口有关的操作。在 MFC 程序中，由于 CHelloWorldView 类是覆盖在 CMainFrame 类窗口之上的，因此开发人员并不需要经常对 CMainFrame 类进行操作，更多的是对视窗类进行操作，从而达到对程序中的数据进行编辑和修改的目的。

最后要指出的是，在 MFC 方式的程序中，当程序的一个实例被运行时，系统根据前面在 CHelloWorldApp 类中介绍的文档模板对象自动生成类 CMmainFrame、ChelloWorldView、CHelloWorldDoc 的对象，而不需要程序员主动去创建这些类的对象。另外，应用程序框架通过主框架窗口与视图对象、文档对象利用消息进行互操作，如主框架窗口访问视图对象的方法是调用 GetActiveView()成员函数，而访问文档对象的方法是调用 GetActiveDocument()成员函数。

3．CHelloWorldView 类（基类为 Cview）和 CHelloWorldDoc 类（基类为 CDocument）

把 CHelloWorldView 类和 CHelloWorldDoc 类一起介绍是因为这两个类是密切相关的。

视图是显示文档数据的界面，它主要占据主框架窗口的客户区，视图类不仅把程序文档数据显示出来，而且还能接受用户的输入、编辑。视图对象也通过消息与主框架、文档对象互操作。

文档对象是存储程序数据的地方。一般而言，把要存储的数据放到文档对象中，即把要存储的数据作为文档对象的成员变量，进而实现文档的串行化，并允许视图类编辑这些成员变量。一个文档包括从图形文件到可编程控制的任何内容。文档类派生于 CDocument 类，AppWizard 自动将其命名为 CXxxDoc，Xxx 是应用程序名。

图 2.14 可以说明文档与视窗的关系。

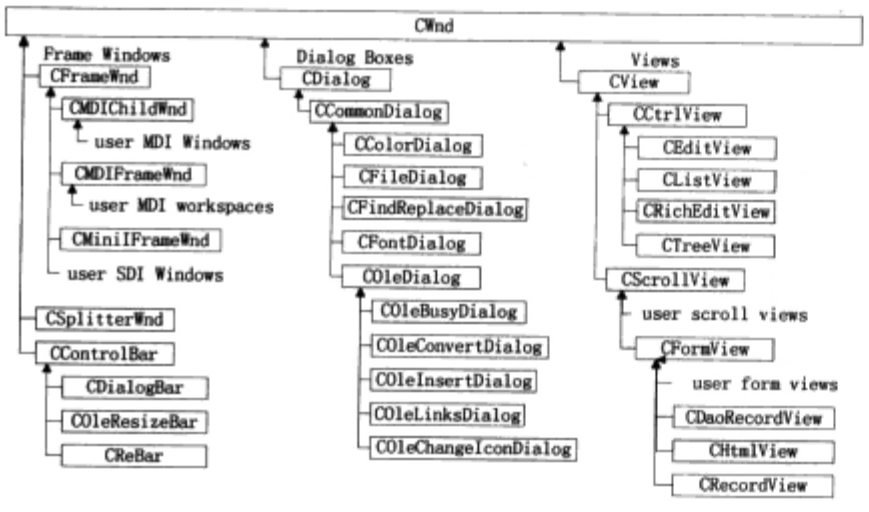


图 2.13　类视图

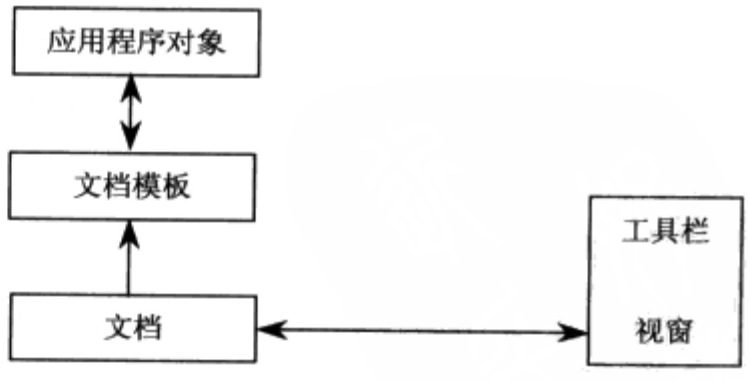


图 2.14　文档关系视图

在这个框图中，文档是由文档模板（Document Template）对象生成的，并由应用程序对象管理，而用户则是通过与文档相关联的视窗对象来存储、管理应用程序的数据的，用户与文档之间的交互是通过与文档相关联的视窗对象进行的。

生成一个新文档的同时，MFC 程序生成一个框架窗口，并且在框架窗口的客户区中生成一个视窗

对象作为框架窗口的子窗口，这个子窗口以可视化的方式表现文档中的内容。视窗的重要功能就是处理用户的鼠标、键盘等操作，通过对视窗对象的处理达到处理文档对象的目的。

总之，类 CHelloWorldView 的作用是显示数据，而数据的来源是类 CHelloWorldDoc，在 MFC 程序中，程序的数据是放在文档中的，而显示数据则是利用视窗方式，文档与视窗分离的一个优点就是一个文档可以同时有多个视窗，每个视窗可以只显示文档中的一部分数据，或者以特定的风格显示文档中的数据。文档与视窗分离的另一个优点就是在程序中可以处理多个文档，通过对不同的视窗的处理达到对不同的文档分别进行处理的目的。

要指出的一点是，Windows 应用程序框架通常分为单文档-视图应用程序框架（SDI）和多文档-视图应用程序框架（MDI）两种，在单文档界面中，文档窗口与主框架窗口是同一概念，而这时的视窗对象则显示在文档窗口的客户区中。之前生成的 HelloWorld 程序使用的就是单文档界面方式，此时文档窗口是主框架窗口，即类 CMainFrame 的对象。有关文档-视图体系结构的知识，在下一节将进行详细的介绍。

4．其他用户界面类

除框架和视图类外，MFC 还提供了许多其他支持用户界面的类。

（1）控件窗口类，封装了通用控件。

（2）菜单类，处理菜单，就像 CWnd 类处理窗口一样。

（3）对话框类，封装了对话框和通用对话（Common Dialog）框。

（4）控制条类，封装了控制条（工具栏、对话条和状态栏）。

（5）属性类，封装了属性表（Property Sheet）和属性页（Property Page）。

上面介绍了应用程序框架的结构及各类之间的关系，图 2.15 所示的结构可以帮助读者更进一步地理解本节的内容，掌握 MFC 应用程序框架是做好 MFC 编程的基础。

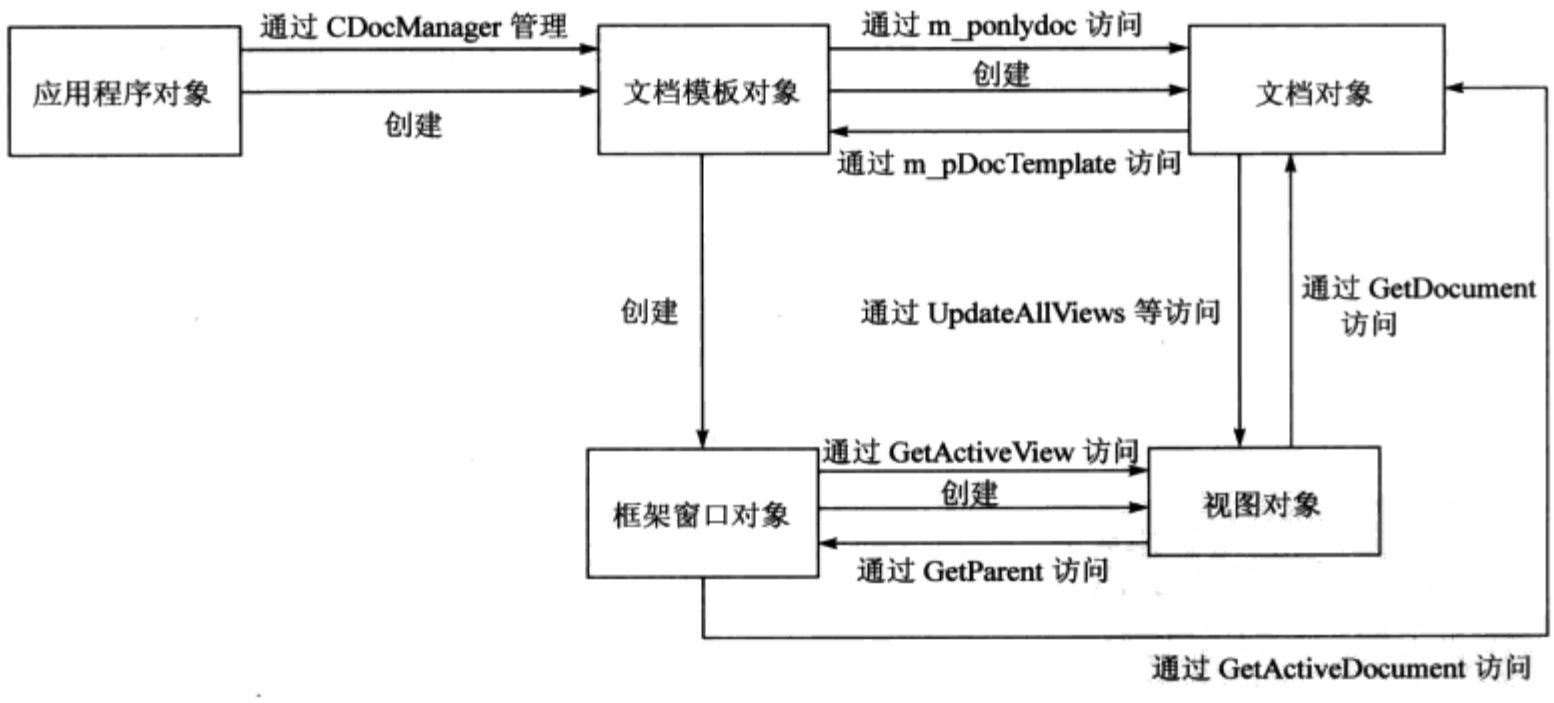


图 2.15　MFC 应用程序概要结构图

2.2.2　MFC 应用程序运行流程

为了更好地学习和掌握基于 MFC 的程序，有必要对 MFC 的运行机制及封装原理有所了解。

WinMain()函数是所有 Win32 程序的入口函数，类似于 DOS 下的 main()函数。MFC 应用程序也不例外，它也有一个 WinMain()函数，但该函数是在程序编译链接时，由链接器链接到 HelloWorld 应用

程序的。

当运行 MFC 用户应用程序时，程序中的应用程序框架首先获取控制权，然后依次执行以下功能。

（1）做部分初始化工作。

（2）构造应用程序的惟一应用类对象。

（3）调用 WinMain()函数。

（4）获取惟一的应用类对象指针。

（5）进行一些内部初始化，其中包括：

① 初始化应用程序，也就是调用应用类的 InitApplication()函数进行初始化。

② 进一步初始化应用程序，也就是调用应用类的 InitInstance()函数进行初始化。

（6）调用应用类的 Run()函数，运行消息，不断地获取消息和处理消息，直到获取 WM_QUIT 消息。

（7）调用应用类的 ExitInstance()函数。

（8）做退出应用程序之前的首尾工作。

（9）从 WinMain()函数返回后，删除惟一的应用程序对象。

（10）终止应用程序。

2.2.3　应用程序窗口

本小节将讨论 MFC 用户界面的基本要素：窗口。在此基础上，比较 API 窗口和 MFC 窗口的异同，描述如何创建一个窗口、销毁一个窗口和控制 Windows 操作系统与窗口的通信问题。

窗口是屏幕上的一个矩形区域，应用程序在该区域中显示数据并等待鼠标单击。Windows 应用系统的用户界面可以包含许多窗口，每个窗口都有不同于其他窗口的特点，但都是互相联系的。窗口的基本类型只有 3 种：重叠（Overlapped）窗口、弹出（Popup）窗口和子窗口（Child）。如图 2.16 所示，它们之间并没有太多内在的差异，但是使用不同的窗口风格，其外观是不同的。

重叠窗口通常用于建立应用程序主窗口，事实上，有时也叫做“主”窗口或者“框架”窗口。

弹出窗口通常以对话框或消息框的形式与用户对话。

子窗口通常用于视图（View）中，如文本编辑器中的文本显示；也用于控件中，如对话框中的“OK”按钮。而那些看似按钮或控件的子窗口，也称为“控件”窗口。

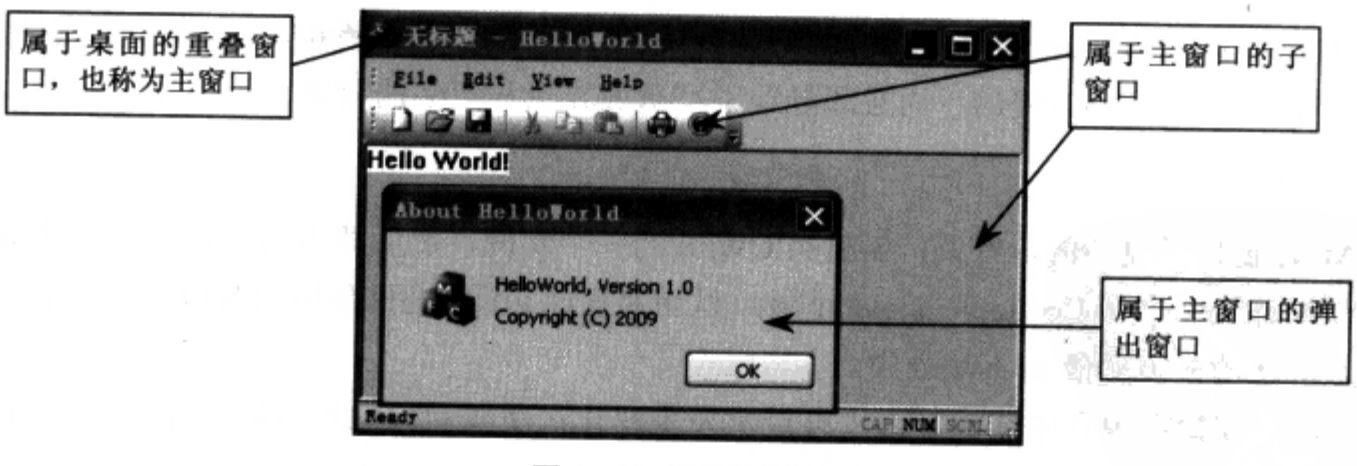


图 2.16　应用程序窗口

重叠窗口和弹出窗口的主要区别是弹出窗口显示时可以没有标题（也称为标题栏）。子窗口与重叠窗口或弹出窗口的主要区别是子窗口只能显示在另一个窗口中，并且子窗口的任何多余部分都被该窗口移去或剪切掉，另外，子窗口是惟一不能有菜单栏的窗口。

即使应用程序中有若干个窗口，每个窗口也只能是 3 种基本类型中的一种：重叠窗口、弹出窗口或子窗口。

每个窗口都包含由系统绘制的“非客户区”和由应用程序绘制的“客户区”。系统可以绘制图 2.17 所示的其中一个或者全部特征，当然也可以把所有的特征留给应用程序开发人员去绘制。

图 2.17　应用程序结构图

每个窗口代表内存中的一个窗口对象，并由该窗口对象告诉 Windows 操作系统将窗口绘制在何处，以及在对鼠标单击、键盘按键按下（假设该窗口拥有输入焦点）和时钟终止等事件做出响应时应调用什么应用程序。窗口对象自身是窗口类的一个实例，它不是 Visual C++的类，而是存在于 Visual C++之外，并早于 Visual C++的 Microsoft Windows 所属的类。然而，就像 C++的类一样，窗口类为每个基于它创建的窗口定义了若干特征，如背景色和往何处发送消息等。

Windows 操作系统提供了一个扩展的应用程序接口（API），可以用来创建和维护这些窗口对象。调用 CreateWindow()函数可以创建一个窗口对象；调用 SetWindowLong()函数可以改变由窗口类定义的特征；调用 MoveWindow()函数可以移动窗口；调用 DestroyWindow()函数可以退出窗口。

MFC 窗口是 C++和 Windows API 调用的综合。一般来讲，MFC 窗口提供了许多（但不是全部的）Windows API 的 C++封装，从而减轻了编写 Windows 应用程序时的一些乏味的工作，并提供了一些新的服务。

MFC 窗口不能对窗口对象进行直接控制，而在 API 环境中却是可以进行直接控制的。如果不能在 API 环境中实现某项功能，那么肯定也不能在 MFC 环境中实现，即 MFC 功能的实现是由 API 间接实现的。举例来说，MFC 库的 CWnd 类可以创建一个窗口，但它只是过去在 API 环境中使用的 API 调用的封装。

在 MFC 库中创建和维护一个窗口的逻辑其实已经是真正的 C++封装和控制，然而，这种方法虽使得 MFC 库万能，但却是不方便的，而且还导致了严重的冗余和更多的错误。

1．创建窗口

创建 MFC 窗口是复杂的，首先，创建类 CWnd 的一个实例，然后调用类 CWnd 的一个成员函数，该成员函数调用 API 中的 Create Window()函数。返回的窗口句柄（这只是指向窗口对象的非直接指针）保存在类 CWnd 的成员变量 m_hWnd 中。

注意，因为窗口在内存中创建，而内存经常发生变化，所以窗口地址可能是经常变化的。因此，窗口句柄并非直接指向窗口对象，而是指向跟踪窗口对象地址的另一个指针。

销毁窗口同样也是复杂的，必须确保销毁了该窗口对象，以及封装该窗口对象的 CWnd 实例。虽然 CWnd 对象知道窗口对象，但是窗口对象并不知道 CWnd 对象。

MFC 使用 CWnd 类创建一个窗口，窗口创建函数如下：

```
CWnd wnd;
BOOL b=wnd.CreateEx (ExStyle, ClassName, WindowName, Style, x, y, Width, Height, Parent, Menu,Param) ;
```

第一行代码创建了一个 CWnd 类的对象，第二行代码通过调用 Windows API 函数 Create WindowEx( )创建了真正的窗口：

```
HWND hwnd =::Create WindowEx (ExStyle, ClassName, WindowName, Style, x, y, Width, Height, Parent,Menu, Instance, Param);
```

由于 CWnd 类只是封装了用于创建窗口的 Windows API 函数（Create WindowEx()），因此，从本质上讲，创建窗口所必需的参数在 API 环境和 MFC 环境中是相同的：

- 参数 Style 和 ExStyle 决定窗口的“外观”和类型（重叠、弹出、子窗口）；
- 参数 ClassName 决定在创建窗口时使用的窗口类，即窗口标题名；
- 参数 WindowName 决定窗口标题内容（如果窗口标题有内容）；
- 参数 x、y、Width 和 Height 决定窗口的位置和大小；
- 参数 Parent 指向拥有该窗口的窗口指针，即父窗口（如果有这样的窗口）；
- 参数 Menu，即菜单句柄，指向内存中的一个对象，作为它的菜单使用，除非创建一个子窗口，如果是这样，那么该参数是帮助父窗口识别子窗口的 IDnumber；
- 参数 Instance，即实例句柄，识别该窗口属于哪个应用程序，以便发送到该窗口的消息能被发送到正确的应用程序的消息队列中，CWnd 类填入 Instance 参数；
- 参数 Param 是创建窗口时由窗口类使用的指针，该指针是指向附加信息的非强制性结构的指针。

返回的 hwnd 参数是指向创建的窗口对象的指针，但在未创建任何窗口时，该参数的返回值为 NULL。窗口句柄自动地保存在 CWnd 类的 m_hWnd 成员变量中。

在了解有关创建窗口的基本知识之后，下面进一步了解填写这些参数的规则。

（1）窗口名称参数。

该参数是一个零结尾的文本串，用该串指明在窗口标题栏中显示的内容。如果窗口没有标题栏，该参数可以为零（0）。

然而，某些通用控件也使用该参数。例如，按钮控件把该参数的内容放在按钮的表面。在创建窗口标题栏后，可以用类 CWnd 的成员函数 SetWindowText()来改变窗口标题栏的名称。

（2）风格和扩展风格参数。

这两个参数是 32 位的，用来指定创建什么类型的窗口。可以选择多种类型，如下所示：

```
WS_CHILD|WS_VISIBLE
```

下面简单介绍窗口风格参数的作用。

- 用于创建 3 种基本窗口类型的风格：用 WM_CHILD 创建一个子窗口，用 WM_POPUP 创建一个弹出窗口，用 WM_OVERLAPPED 创建一个重叠窗口。
- 如果不为窗口指定任何一种风格，那么窗口风格默认为 WM_OVERLAPPED。
- 用于增添窗口的非客户区特色的风格，例如，可以用 WS_VSCROLL 为窗口添加一个垂直滚动条。如图 2.18 所示为其他非客户区窗口风格。
- 由每个通用控件定义的风格，例如，BS_FLAT 按钮风格告诉按钮控件绘制一个二维按钮。
- 用于使窗口可视、可操作和/或初始最大化的风格。
- 用于标识控件组起始控件的风格，或者当用户按【Tab】键控制焦点在窗口中变化时，用于指示哪个控件有资格获得输入焦点的风格。

在创建窗口以后，可以用 CWnd 的成员函数 ModifyStyle()和 ModifyStyleEx()改变窗口风格。某些风格可能要求重新绘制窗口，这时，可以给 ModifyStyle()函数添加第三个参数，自动激发类 CWnd 的成员函数 SetWindowPos()做重新绘制窗口的工作。

```
CWnd wnd;
```

```
wnd.ModifyStyle (0,WS_BORDER,SWP_NOZORDER);
```

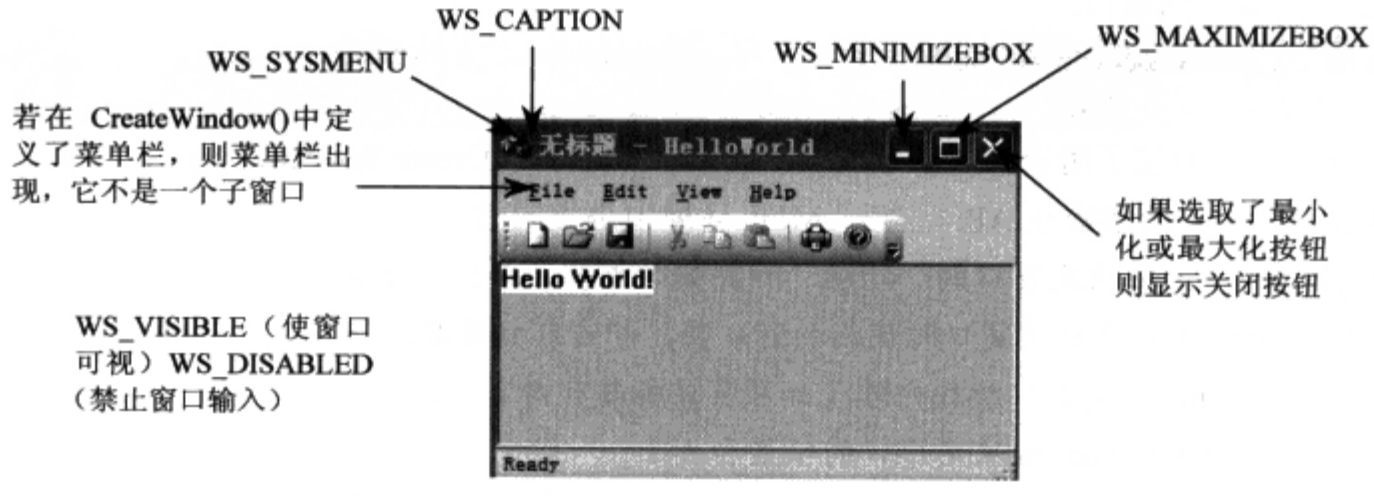


图 2.18　应用程序结构图

事实上，给 ModifyStyle()或 ModifyStyleEx()添加任何第三个参数，都需要添加下面的 SetWindowPos() 选项：SWP_NOZORDER、SWP_NOMOVE、SWP_NOACTIVATE 和 SWP_SHOW。

注意，有时一个重新绘制的窗口可能与新风格不相容，尤其对通用控件窗口来说更是这样。因为单个窗口决定重新绘制窗口时使用什么风格，在这种情况下，惟一的解决办法是：在事先存储已有窗口的风格和其他参数后，销毁和重建该窗口。

（3）X 和 Y 位置参数。

这是两个 32 位的参数，用于以像素为单位指定窗口的位置。创建重叠窗口和弹出窗口时，X 和 Y 是相对于桌面窗口左上角的位置；而创建子窗口时，X 和 Y 是相对于父窗口客户区左上角的位置。如果把 X 和 Y 参数都设置为 CW_USEDEFAULT，那么系统将自动为窗口选定一个位置，系统层叠排列这些新窗口（如图 2.19 所示）。

然而，如果 X 和 Y 参数都设置为 CW_USEDEFAULT，那么子窗口在创建的时候，总被创建在（0,0）的位置。

创建窗口以后，可以用类 CWnd 的成员函数 MoveWindow()移动窗口。

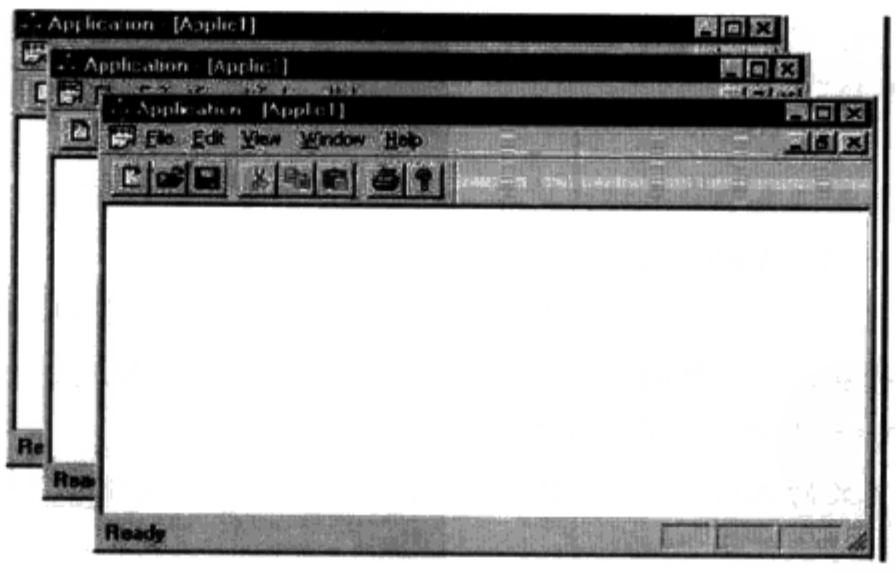


图 2.19　窗口位置图

（4）宽度和高度参数。

这是两个 32 位的参数，用于以像素为单位指定窗口的大小。如果将参数 Width 和 Height 都设置为 CW_USEDEFAULT，则系统将根据桌面窗口的大小，自动选定窗口的大小。然而，对一个子窗口而言，系统将创建一个长和宽均为 0 的窗口。如果窗口的风格是 WS_MINIMIZE 或 WS_MAXIMIZE，那么系统将忽视用户为 Width 和 Height 设置的任何值。

创建窗口以后，可以用类 CWnd 的成员函数 MoveWindow()重新设置窗口的大小。

（5）Z-Order。

当几个窗口占据屏幕上的同一区域时，Z-Order 决定哪个窗口显示在哪个窗口之上。Z-Order（Z 序）中的 Z 来源于坐标 *X-Y-Z* 轴的 *Z* 轴，其中 *Z* 轴垂直于屏幕，并由屏幕指向外面。当窗口最初被创建或选中时，窗口将显示在 Z-Order 的顶层，然而，该窗口永远不可能显示在一个最顶层窗口的上面，除非该窗口也是最顶层的窗口。“最顶层”窗口用 WS_EX_TOPMOST 窗口风格创建，并显示在所有非最顶层窗口的上面，而与该窗口是不是当前选中的窗口无关。

创建窗口后，可以用 CWnd 的成员函数 SetWindowPos()改变窗口的 Z 序。

（6）父窗口或物主窗口参数。

该参数是指向类 CWnd 对象的指针，根据创建的窗口类型标识是父窗口还是物主窗口（Owner）。

- 如果创建的是一个子窗口，那么用该参数来标识其父窗口（该子窗口所放置的并为之所截断的窗口）。该值不能为 NULL。子窗口只能显示在其父窗口中，当父窗口被销毁时，它们也被销毁，并且当父窗口被隐藏或最小化时，它们也被隐藏。
- 如果创建的是重叠窗口或弹出窗口，该参数用来标识物主窗口。如果该值为 NULL，则桌面窗口成为物主窗口。从属窗口总是显示在其主窗口上面，并且随着物主窗口的销毁而被销毁；物主窗口最小化时，从属窗口被隐藏；物主窗口被隐藏时，从属窗口并不被隐藏。

一个子窗口可能是另一个子窗口的父窗口，但绝不可能是一个物主窗口。如果试图使一个子窗口成为一个物主窗口，那么系统只能使那个子窗口的最顶层窗口作为物主窗口。

可以用 CWnd 的成员函数 SetOwner()改变已有窗口的物主窗口，用 CWnd 的成员函数 SetParent()改变父窗口。

（7）菜单或控件 ID 参数。

该参数用来标识菜单（Menu）句柄或控件（Control）ID，这要依赖于创建的窗口是子窗口、重叠窗口，还是弹出窗口。

- 如果创建的是一个子窗口，用该参数标识控件 ID，控件 ID 通常用来帮助父窗口识别子窗口。由于该参数寻求一个 HMENU 变量，因此需要用 HMENU 类型重载变量定义控件 ID。例如，如果子窗口的控件 ID 是 102，应该用如下方法定义它：

```
…, (HMENU) 102，…
```

- 如果创建的是一个重叠窗口或弹出窗口，用该参数定义窗口的菜单。若该值为 NULL，则菜单默认为在该窗口的窗口类中定义的任何菜单；如果窗口类也没有指定的菜单，则该窗口在创建时没有菜单。可以用下面的方法为该参数加载一个菜单：

```
HMENU hMenu=::LoadMenu (AfxGetInstanceHandle( ), MAKEINTRESOURCE (xx));
```

这里的 xx 是应用程序资源中的菜单 ID（Menu ID）。可以用类 CWnd 的成员函数 SetDlgCtrlID()改变已有子窗口的 ID，用类 CWnd 的成员函数 SetMenu()改变一个重叠窗口或弹出窗口的菜单。

（8）实例。

前面已提到，CWnd 类将填入该参数。CWnd 通过调用函数 AfxGetInstanceHandle()获得该实例（Instance）。一个应用程序的实例从本质上标识了哪个程序在内存中，fxGetInstanceHandle()是由 MFC 库提供的静态函数。

（9）参数。

这个 32 位参数（Parameter）是可选的，它通常是指向一个结构或者类对象的指针，而该结构或者类对象是创建某种类型的窗口时需要提供的。例如，当用 MDICLIENT 窗口类创建窗口时，需要该参数提供一个 CLIENTCREATESTRUCT 结构的指针。

（10）类名参数。

类名（Class Name）参数是一个零结尾字符串，创建一个窗口时，用来标识使用哪个窗口类。关于窗口类和窗口处理，将在本小节后面的内容中详细讨论。该参数不能为 NULL，在创建一个非常一般的 MFC 窗口时，使用 AfxRegisterWndClass(0)填入该参数。

2．销毁窗口

在创建完窗口后，还应该考虑的一点是如何销毁一个窗口。下面将详细介绍销毁一个 MFC 窗口的步骤。

如前所述，删除一个 MFC 窗口的操作可能有些烦琐，必须按下面的顺序删除两个对象：

```
pWnd -> Destroy Window( ); //destroys Window Object
deletepWnd; //destroys Cwnd Object
```

也可以只删除 CWnd 对象，这是因为 DestroyWindow()是从 CWnd 的析构函数中调用的，但不提倡使用这种方法。销毁一个窗口而不先调用 DestroyWindow()函数，将使某些析构消息不能被任何从 CWnd 派生的类处理。开发人员几乎不需要自己销毁一个窗口，用户或系统通常会做这项工作。

如果需要在销毁窗口的同时销毁 CWnd 对象，则应该在 CWnd 的派生类中添加下面的重载函数：

```
CYourWnd::PostNcDestroy()
{
    delete this;
}
```

PostNcDestroy()是销毁窗口前调用的最后一个成员函数，但是，几乎不需要对该函数进行重载，因为 CWnd 和派生类一般是嵌在另一个类中或建立在堆栈中的。

3．窗口类

前面已提及，一个窗口类不是一个 C++类，而是早于并存在于 C++之外的窗口专有的类。窗口类的作用就像一个模板，可以由此创建其他窗口，并可共享某些特征，包括下面所示的特征。

- 类风格：包括能给予窗口的一些微小的特征。
- 类图标：如果窗口有图标，用它来指定在窗口的左上角处所画的图标。
- 类菜单：如果窗口有菜单，用它来指定窗口中显示的菜单。
- 类光标：当光标通过窗口的客户区时，用它来指定显示哪种鼠标形状。
- 背景色：定义用什么颜色来擦除窗口的背景色。窗口的客户区将显示该颜色。
- 窗口进程：定义处理任何发送到该窗口的消息时应该调用的函数。定义窗口进程可能是窗口类最重要的惟一的前提。

窗口与环境的交互是通过发送和接收消息来实现的。如果系统要求窗口自己进行绘制，系统给它发送一个 WM_PAINT 消息；如果系统要求窗口销毁自己，则发送一个 WM_DESTROY 消息。这些消息都由窗口的窗口进程处理，该窗口进程的地址在窗口类中定义。对于发送到由相同的窗口类创建的窗口的消息，系统采用完全相同的窗口进程进行处理。相同的窗口进程是怎样跟踪所有的窗口的呢？它是怎么知道窗口 A 画在（10,34）位置，而窗口 B 画在（56,21）位置呢？所有这些工作的完成只需使用窗口的窗口对象。

例如，所有用按钮窗口类创建的窗口，都使用相同的按钮窗口进程。如果一个 WM_PAINT 消息发送到其中任何一个窗口，则按钮窗口进程根据每个窗口的窗口对象指定的大小、位置和风格画出确切的按钮。

4．创建窗口类

当用户创建一个窗口类时，实际上只是在 3 个操作系统列表之一中注册一个 WNDCLASS 结构。

系统为每个窗口类类型维护一个列表。

- 系统全局类（System Global Class）：在系统启动时注册，且必须注册，对所有应用程序都有效。
- 应用程序全局类（Application Global Class）：由应用程序注册，只对应用程序及其线程有效。
- 应用程序局部类（Application Local Class）：由应用程序注册，并只对注册它们的应用程序或 DLL 有效。

当系统搜索一个窗口类时，从应用程序局部类开始，然后搜索应用程序全局类，最后搜索系统全局类。

要创建一个窗口类，可以先创建 WNDCLASS 结构的实例，然后用 MFC 类库的 AfxRegisterClass() 函数注册它，也可以用 MFC 类库的 AfxRegisterWndClass() 函数来创建一个基于调用参数的 WNDCLASS 对象。

首先，使用 AfxRegisterWndClass()函数注册一个窗口类。

AfxRegisterWndClass()函数在使用上是非常自动化的，一些通常需要开发人员提供的参数，它都能自己输入，甚至连新的窗口类的名称也能自动产生：

```
LPCTSTR lpszClassName=AfxRegisterWndClass(UINT nClassStyle, HCURSOR hCursor=0,HBRUSH hbrBackground=0, HICON hIcon=0);
```

这些参数的使用规则如下。

（1）类名。根据传给该函数的参数，为新的窗口类产生名称。如果传输的参数完全相同，那么创建的窗口类也将完全相同。如果需要创建新的窗口类，可以使用 AfxRegisterClass()函数。

（2）风格。窗口类风格由表 2.1 所示的选项列表中一系列标记的或（OR）运算提供。

表 2.1　　窗口类风格

类　风　格	描　　述
CS_OWNDC	为该窗口类创建的每个窗口分配惟一的设备环境。有关设备环境的更详尽资料
CS_PARENTC	从系统高速缓存中检索设备环境，然后设置该设备环境的剪裁区，以组合到父窗口中，以便子窗口能画在父窗口上
CS_CLASSDC	分配一个设备环境给所有由该窗口类创建的窗口使用
CS_SAVEBITS	为由该类创建的任何窗口的视频存储区将被保存，以便窗口移动或关闭时，不需要重画基础窗口，这对快速机器来说作用不大
CS_BYTEALIGNCLIENT CS_BYTEALIGNWINDOW	当计算机的显卡和 CPU 速度较慢时，这两种风格有用。添加这些风格后，当窗口在字节边缘时，显卡更容易移动窗口
CS_GLOBALCLASS	如果设置该风格，则该类是应用程序的全局类，否则它是一个应用程序局部类
CS_VREDRAW	如果设置了垂直风格，并且窗口的垂直大小发生了变化，则整个窗口被重画
CS_HREDRAW	水平风格同垂直风格
CS_NOCLOSE	禁用系统菜单中的关闭命令
CS_DBLCLKS	如果未设置该参数，并且双击由该窗口类创建的窗口，则传送给应用程序的将不是双击事件，而是两个相继完成的单击事件

（3）图标（Icon）。该参数是显示在窗口左上角的图标的句柄，但只适用于使用 WS_SYSMENU 的窗口风格。

应用程序主窗口的图标也显示在任务栏上。如果将该参数设置为 NULL，并且设置了 WS_SYSMENU 风格，则系统将提供一个默认的图标。在 MFC 环境中，绝大部分图标已被处理过，用 CWnd 的 SetIcon()函数可改变已有的图标。

（4）光标（Cursor）。该参数是鼠标指针移经应用程序窗口的客户区时，将要显示的鼠标光标句柄。如果将该参数设置为 NULL，则显示的是箭头光标。可以用下面的语句装载一个光标：

```
HICON hIcon=AfxGetApp( )->LoadCursor (xx);
```

这里的 xx 是应用程序资源中光标的名称或 ID。

这里指定的光标意味着是该窗口的默认光标。如果想动态地改变光标形状，则应该处理该窗口的 WM_SETCURSOR 消息，并用 SetCursor()函数来改变光标形状。

（5）背景色。当系统创建窗口时，先在显示窗口的位置画一个矩形区域，以擦除该区域的背景色。该参数指定填充该矩形域时所用画刷的句柄（参见第 4 章有关画刷的详细内容）。为窗口类创建的画刷对象在该类退出注册时被自动释放。

在指定背景色时，也可以不创建画刷对象，而指定下面所列系统颜色之一：

```
COLOR_ACTIVEBORDER
COLOR_ACTIVECAPTION
COLOR_APPWORKSPACE
COLOR_BACKGROUND
COLOR_BTNFACE
COLOR_BTNSHADOW
COLOR_BTNTEXT
COLOR_CAPTIONTEXT
COLOR_GRAYTEXT
COLOR_HIGHLIGHT
COLOR_HIGHLIGHTTEXT
COLOR_INACTIVEBORDER
COLOR_INACTIVECAPTION
COLOR_MENU
COLOR_MENUTEXT
COLOR_SCROLLBAR
COLOR_WINDOW
COLOR_WINDOWFRAME
COLOR_WINDOWTEXT
```

然而，必须分配颜色到一个画刷句柄类型，并加 1。

```
(HBRUSH) (COLOR_WINDOW+1)
```

设置该参数为 NULL，则在画一个窗口之前，系统不对窗口进行擦除。在非客户区的绘制还是与平常一样，但客户区将保持窗口被画前原来屏幕所显示的外观。若该参数设为 NULL，应确认窗口是画全部客户区，还是处理 WM_ERASEBRGND 消息以擦除背景色。

其次，使用 AfxRegisterClass()函数创建一个窗口类。

如果想完全控制一个新窗口类的创建，如指定窗口类的名称，则应该用下面的语句：

```
BOOL AFXAPI AfxRegisterClass (WNDCLASS• lpWndClass);
```

这里的 WNDCLASS 结构定义如例 2.3 所示。

例 2.3

```
typedef struct _WNDCLASS {
UINT style; // style of the class
WNDPROC     lpfnWndProc ;          // function called by system when
// it has a message for a window
// created with this class
int                 cbClsExtra;          // extra bytes to add to the
// WNDCLASS structure when
// registering
int                 cbWndExtra;          // extra bytes to add to the Window
// Objects created with this class
```

```
HANDLE      hInstance;          // instance that owns this class
HICON       hIcon;              //icon to be used when window
// displays an icon
HCURSOR     hCursor;            // cursor to use when mouse is over
// a window created with this class
HRBUSH      hbrBackground;      // background color to use when
// erasing the background area under
// a window created with this class
LPCTSTR     lpszMenuName;       // menu name to be used when
// creating a menu for a window
// created with this class
LPCTSTR     lpszClassName;      // the class name that identifies
// this WNDCLASS for the system
} WNDCLASS;
```

有关填入类的“风格”、“图标”、“光标”和“背景色”参数的规则，请参看前面部分的介绍，其余参数的填入规则如下。

（1）类名。类名参数是用来标识新窗口类的零结尾字符串。可以任意命名窗口类，但是不要与已有的窗口类同名，除非想使该类无效。正如前面所提及的，系统从应用程序局部类开始，在 3 个列表中寻找类名匹配，如果在该列表中记录了一个与系统全局类同名的类，则应用程序将使用新记录的类。

（2）菜单名。该参数指向应用程序资源中的菜单名。在 MFC 环境中，大多数情况下，由系统载入菜单，但可以在此处指定菜单名。如果使用资源 ID，也可以采用下面的语句：

```
MAKEINTRESOURCE (IDR_MENU);
```

这将成为该窗口的默认菜单。若指定该值为 NULL，系统将使用在类 CWnd 的 CreateEx()函数中定义的菜单；若在此处没有指定的菜单，则该窗口将没有菜单。该参数对子窗口可以忽略，因为它们从来没有菜单。

（3）窗口进程和实例。这两个参数是窗口类中最重要的参数。有关窗口进程的知识在前面已经讨论过了，而实例只是用于标识哪个应用程序包含了该窗口进程。在 MFC 环境中，默认的窗口进程为 AfxWndProc()，可以使用下面的语句返回该进程，并为该参数使用：

```
AfxGetAfxWndProc( );
```

可以用 AfxGetInstanceHandle()函数填入 Instance 参数。

（4）Class Extra 和 Window Extra 参数。ClassExtra 和 WindowExtra 参数提供了一种存储数据的方法，允许应用程序自身将所属的数据存储到窗口对象或注册的窗口类中。注册类时，Class Extra 指定在类的末尾分配的额外字节数；创建窗口对象时，Window Extra 指定添加到该窗口对象尾部的额外字节数。

在这两种情况下，这些额外字节都被应用程序用来存储属于窗口或窗口类的消息。然而，由于 CWnd 对象与一个窗口关联，并且一个 CWnd 对象可以存储的信息量如果不比额外字节多，起码也与之相等；因此，这些额外字节参数实际上并没有多大用处，可以把它们设为 0。

5. 销毁窗口类

通过取消窗口类的注册来销毁窗口类。但是，如果用 AfxRegisterWnd()或 AfxRegisterWndClass()注册一个类，那么应用程序在结束的时候，类的注册会自动取消，即使应用程序不取消窗口类的注册，操作系统也会取消。

2.2.4　消息机制与消息循环

至此，注册窗口类、创建窗口、显示和更新窗口等初始化工作都已完成，WinMain()就应该进入消

息循环了：

```
MSG msg;
GetMdssage(
        LPMSG lpMsg;
        HWND hwnd;
        UINT     wMsgFilterMin;
        UINT     wMsgFilterMax;
)
```

在 Win32 应用程序中采用的是消息循环机制，用一系列的 switch-case 语句来处理消息，而在 MFC 中，对消息的处理采用的消息映射机制。

下面就将对 Win32 应用程序及 MFC 中采用的消息映射机制进行深入的讲解。首先介绍 Win32 应用程序的消息映射机制。

其中各个参数分别表示：lpMsg 指消息结构体指针；hwnd 指要获取哪个窗口消息。NULL 表示所有窗口；wMsgFilterMin 是设定消息最小值；wMsgFilterMax 是设定消息最大值：

```
While(GetMessage(&msg,...)){
        TranslateMessage(&msg);        //转换键盘消息
        DispatchMessage(&msg);         //分派消息
}
```

其中的循环是死循环，保证能够从消息队列中一直取消息，并且一次只能取一条消息；TranslateMessage(&msg)是为了将键盘消息化，即对取到的消息进行转换，转换为 WM_CHAR 消息，并放入消息队列中，如可以将按键的消息对 LBUTTON_DOWN 和 LBUTTON_UP 转换为 WM_CHAR 消息，并且不会影响原来的消息，只是产生新消息；DispatchMessage(&msg)会将消息传给窗口回调函数去处理，由操作系统调用窗口函数，即把消息路由给操作系统，操作系统再调用窗口回调函数进行处理。没有指定函数名称，却可以将消息传过去，是因为消息发生之时，操作系统已根据当时的状态，为它标明了所属窗口，而窗口所属的窗口类又已经明白标识了窗口函数（也就是 wc.lpfnWndProc 所指定的函数），所以 DispatchMessage()自有脉络可寻。DispatchMessage()经过 USER 模块的协助，才把消息传到窗口函数。

消息循环中的 GetMessage()是 Windows 3.x 非强制性（Non-preemptive）多任务的关键。应用程序借由此操作，提供了释放控制权的机会：如果消息队列上没有属于自己的消息，就把机会让给别人。通过程序之间彼此协调让步的方式，实现多任务功能。Windows 9X 和 Windows NT 具备强制性（Preemptive）多任务能力，不再非靠 GetMessage()释放 CPU 控制权不可，但程序写法依然不变，这是因为应用程序仍然需要靠消息推动，它还是需要抓消息！

消息循环中的 DispatchMessage()把消息分配到哪里呢？它通过 USER 模块的协助，送到该窗口的窗口函数中了。窗口函数通常利用 switch-case 方式判断消息种类，以决定处置方式。由于它是被 Windows 操作系统所调用的，并没有在应用程序的任何位置调用此函数，所以这是一种 call back 函数，意思是，在程序中被操作系统调用的函数。虽然这种函数由开发人员设计，但永远不会也不应该被开发人员调用，它们是为 Windows 操作系统准备的。

在程序进行过程中，消息由输入装置，经由消息循环的抓取，源源不断地传送给窗口并进而送到窗口函数中去。窗口函数的体积可能很庞大，也可能很精简，应该根据窗口感兴趣的消息数量多寡而定。至于窗口函数的形式，则相当一致，必然是：

```
LRESULT CALLBACK WndProc(HWND hWnd,
                         UINT     message,
                         WPARM    wParam,
```

```
LPARAM    lparam)
```

可以更改该窗口函数的名称，但参数类型必须保持不变。

该窗口函数的参数与消息结构体前 4 个参数一致，当调用 Dispatch()时，操作系统将调用该窗口回调函数，并且将消息结构体前 4 个参数作为实参传递给该窗口回调函数。

不论什么消息，都必须被处理，所以 switch/case 指令中的 default 处必须调用 DefWindowProc()，这是 Windows 内部默认的消息处理函数。

窗口函数的 wParam 和 lParam 的意义，因消息不同而不同。wParam 在 16 位环境中是 16 位，在 32 位环境中是 32 位，因此，参数内容（格式）在不同的操作环境中就有了变化。

那为什么 Windows Programming Modal 要把窗口函数设计为一个 call back 函数？为什么不让程序在抓到消息（Get message）之后直接调用它？原因是，除了开发人员需要调用它，有很多时候操作系统也要调用该窗口函数，例如，当某个消息产生或某个事件发生时，窗口函数被设计成 call back 形式，才能开放出一个接口给操作系统调用。

在 Win32 应用程序中采用的是消息循环机制，用一系列的 switch-case 语句来处理消息，而在 MFC 中，对消息的处理是采用消息映射机制。下面就将对消息及 MFC 的消息映射机制进行深入的讲解。

1．消息的种类

在 Windows 程序设计中，消息是一个极为重要的概念，用户通过窗口界面的各种操作最后都转化为发送到程序中的对象的各种消息，下面就介绍在 Windows 程序设计中最常用的一些消息。

（1）键盘消息。

```
WM_CHAR，该消息的处理函数 OnChar()
WM_KEYDOWN，用户按一个非系统键
WM_KEYUP，在非系统键被释放时产生消息
```

这 3 个消息用来处理用户的键盘数据，当用户在键盘上按某个键的时候，会产生 WM_ KEYDOWN 消息，释放按键的时候又会产生 WM_KEYUP 消息，所以 WM_KEYDOWN 与 WM_KEYUP 消息一般总是成对出现的，至于 WM_CHAR 消息，是在用户的键盘输入能产生有效的 ASCII 码时才会发生。值得提醒的是，前两个消息与 WM_CHAR 消息在使用上是有区别的。在前两个消息中，伴随消息传递的是按键的虚拟键码，所以这两个消息可以处理非打印字符，如方向键、功能键等；而伴随 WM_CHAR 消息的参数是所按的键的 ASCII 码。ASCII 码是可以区分字母的大小写的；而虚拟键码是不能区分大小写的。3 种消息原型分别如下：

```
afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
        afx_msg void OnKeyUp(UINT nChar, UINT nRepCnt, UINT nFlags);
```

（2）鼠标消息。

```
WM_MOUSEMOVE，用户将鼠标指针移进窗口或在窗口中移动
WM_LBUTTONDOWN，用户按下左键
 WM_LBUTTONUP，用户释放左键
 WM_LBUTTONDBCLICK，用户双击左键
 WM_RBUTTONDOWN，用户按下右键
 WM_RBUTTONUP，用户释放右键
 WM_RBUTTONDBCLICK，用户双击右键
```

这组消息是与鼠标输入有关的，WM_MOUSEMOVE 消息发生在移动鼠标的时候，剩余的 6 个消息则分别对应于鼠标左右键的按下、释放、双击事件。要指出的是，Windows 系统并不是在鼠标指针每移动一个像素时都产生 MOUSEMOVE 消息。消息原型分别如下：

```
afx_msg void OnMouseMove(UINT nFlags, CPoint point);
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
afx_msg void OnLButtonDblClk(UINT nFlags, CPoint point);
afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
```

（3）窗口消息。

```
WM_CREATE，窗口被创建
WM_DESTROY，窗口被销毁
WM_CLOSE，窗口被关闭
WM_MOVE，窗口发生移动
WM_SIZE，窗口发生改变
WM_PAINT，窗口发生重绘
```

创建一个窗口对象的时候，在创建过程中收到的就是 WM_CREATE 消息，对这个消息的处理过程一般用来设置一些显示窗口前的初始化工作，如设置窗口的大小、背景颜色等。WM_DESTROY 消息指示窗口即将要被撤销，在这个消息处理过程中，可以做窗口撤销前的一些工作。WM_CLOSE 消息发生在窗口将要被关闭之前，在收到这个消息后，一般的操作是回收所有分配给这个窗口的各种资源。在 Windows 系统中资源是很有限的，所以回收资源的工作是非常重要的。

窗口移动的时候产生 WM_MOVE 消息，窗口大小改变的时候产生 WM_SIZE 消息，而窗口工作区中的内容需要重画的时候就会产生 WM_PAINT 消息。消息原型分别如下：

```
afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
afx_msg void OnDestroy();
afx_msg void OnClose();
afx_msg void OnMove(int x, int y);
afx_msg void OnSize(UINT nType, int cx, int cy);
afx_msg void OnPaint();
```

（4）焦点消息。

```
WM_SETFOCUS，窗口得到焦点
WM_KILLFOCUS，窗口失去焦点
```

当一个窗口从非活动状态变为具有输入焦点的活动状态的时候，就会收到 WM_SETFOCUS 消息；而当一个窗口失去输入焦点变为非活动状态的时候，就会收到 WM_KILLFOCUS 消息。消息原型分别如下：

```
afx_msg void OnSetFocus(CWnd* pOldWnd);
afx_msg void OnKillFocus(CWnd* pNewWnd);
```

（5）定时器消息：WM_TIMER。

Windows 定时器是一种周期性消息产生装置，为一个窗口设置了定时器资源之后，系统就会按规定的时间间隔向窗口发送 WM_TIMER 消息，在这个消息中就可以处理一些需要定期处理的事情。定时器消息的响应函数是 OnTimer()，原型如下：

```
afx_msg void OnTimer(UINT nIDEvent);
```

对于定时器的操作，通常都会和函数 SetTimer()配合使用。在响应 OnTimer()函数之前，应先添加一个定时器对其进行触发，如果没有建立定时器，系统是不会自己去触发 OnTimer()函数的，SetTimer()函数的作用就是设置定时器，定义如下：

```
UINT SetTimer(UINT nIDEvent,UINT nElapse, void (CALLBACK EXPORT* lpfnTimer)(HWND, UINT, UINT, DWORD) );
```

其中，nIDEvent 代表定时器标志；nElapse 代表时间间隔，以毫秒为单位；lpfnTimer 指定定时器消息处理函数，通常为 NULL，表示以 OnTimer()为定时器消息处理函数，当然开发人员也可以重载该函数。

设置定时器之后，就可以在 OnTimer()函数中输入代码，当定时器时间触发时，系统就会调用 OnTimer()函数，处理用户希望发生的操作。

（6）命令消息：WM_COMMAND。

每当用户选择一个菜单命令、单击一个按钮或需要告诉系统应当执行什么操作的时候，则发送一条命令消息 WM_COMMAND。所有命令消息都包含一个共同的参数，那就是该命令消息需要操作的资源 ID 的值。例如，当选择“File”→“New”命令时，产生的命令消息将包含该菜单命令的资源 ID 值，如 ID_FILE_NEW，处理函数为 OnCommand()。

2．应用程序的 Run()函数

应用程序在初始化之后，被 MFC 封装在 CwinApp 类中的 WinMain()会开始调用 Run()函数来处理消息循环。Run()函数是应用程序的核心，要占用程序运行期的绝大部分时间。该函数是循环函数，负责从应用程序的消息队列中检索和处理各种消息，不断执行循环，检查消息队列中是否有消息。如果消息队列中有消息，并且不是 WM_QUIT 消息，那么 Run()就被调入 MFC 的消息处理代码中；如果是 WM_QUIT 消息，则循环结束。如果队列中没有消息，Run()就调用 OnIdle()函数，用户可以重载该函数，以使应用程序能够执行任何后台任务。

MFC 应用程序对消息的处理大致分为两个阶段，第一阶段，应用程序类的 Run()函数把消息从应用程序的消息队列中提取出来，并且发送到目标类对象——即主框架窗口类对象；第二阶段，所有消息（包括窗口消息、命令消息等）的最终目标并不一定是主窗口类，可能是其他类，如视图类等，主窗口对象在 MFC 消息机制的协助下继续寻找消息处理函数。一般情况下，应用程序类的 Run()函数继承 CWinApp 的虚函数 Run()，除非有特殊的要求，否则一般不去重载它。

3．消息映射表

当MFC应用程序类的Run()函数把消息交给主窗口后，主窗口函数如何处理这些消息呢？在Win32 程序中，处理消息的窗口回调函数 WndProc()利用 switch-case 结构来对消息进行判断并分类处理。但 MFC 应用程序的主窗口类对消息的处理并没有采用 Win32 的方法。有人可能会想到可不可以在 CWnd 类中为每个窗口消息都定义一个虚函数，这样，对指定消息的响应只需要在派生类重载该虚函数即可。这样做从原理上讲没问题，但是从编程角度讲，是不可取的。

为什么不可取？在应用程序中使用的每个相关的派生类，系统都要为它们分配一个 vtable 虚拟函数调度表，不管这些函数是否确实在派生类中被覆盖，每一个 vtable 对每一个虚函数需要一个 4 字节的输入项，而 CWnd 中通用函数是如此之多，CWnd 的派生类也有很多，这样做将使 CWnd 类及其派生类背着一个很大的虚拟函数表的包袱，是很占内存的，而且每次扫描虚拟函数表时很费时间，这样定义虚拟函数的做法显然是不合适的。那么 MFC 最终是如何实现消息处理的呢？

MFC 的做法是，在每个能接收和处理消息的类中，定义一个消息和消息函数静态对照表，该表叫做消息映射表。在消息映射表中，消息与对应的消息处理函数指针是成对出现的。某个类能处理的所有消息及其对应的消息处理函数的地址都列在这个类所对应的静态表中。当有消息需要处理时，程序只需搜索该消息映射表，查看表中是否含有该消息，即可知道该类能否处理此消息。如果能处理该消息，则同样依照映射表能很容易找到并调用对应的消息处理函数。

那么，类中是如何加入静态消息映射表，用户是怎么样添加消息映射的呢？MFC 是通过提供一对宏来实现的：DECLARE_MESSAGE_MAP()和 BEGIN_MESSAGE_MAP()、END_MESSAGE_MAP()。

这两对宏的说明如下。

- DECLARE_MESSAGE_MAP(): 在类的头文件（.h）中，在类声明的尾部，用于声明在源文件中存在的消息映射。
- BEGIN_MESSAGE_MAP()和 END_MESSAGE_MAP(): 在类的实现文件（.cpp）中使用，分别标识消息映射的开始和消息映射的结束。

4．如何手动添加用户自定义消息映射

一个 MFC 消息处理程序需要一个函数原型、一个函数体和一个在消息映射中的输入项（宏调用）。对于类本身提供的消息处理函数，可以通过 ClassWizard 或工作台添加消息映射。那么，如果要动态响应用户自定义的函数，又将怎样添加消息映射呢？

假设自定义发送一个 WM_HIDE 消息，其消息响应函数为 OnHide()，添加消息映射的过程如下。

（1）为自定义的消息起一个别名。

在头文件中或 Resource.h 文件中，为 WM_HIDE 消息定义一个 ID。如下：

```
#define   WM_HIDE   WM_USER+101
```

这里使用 WM_USER 是为了避免消息被重复定义，以及和 Windows 系统定义的消息发生冲突。WM_USE(0x0400)至 0x7FFF 是 Windows 操作系统专门为用户保留的用于标志用户自定义消息的。用户取值应该取这一段的值，如 WM_USER+1、WM_USER+2。注意，在同一个应用程序中，用户自定义的多个消息的 ID 值不能相同，都是惟一的。

（2）声明消息处理函数。

在类的头文件（.h）中，在类声明的部分，在宏 DECLARE_MESSAGE_MAP()之前添加消息响应函数。如下：

```
class CTestView : public CView
{
…
      protected:
  //{{AFX_MSG(CTest1View)
  //系统本身的消息处理函数
  //}}AFX_MSG
  afx_msg void OnHide();
  DECLARE_MESSAGE_MAP()
};
```

（3）建立消息映射。

在类的实现文件中，在宏 BEGIN_MESSAGE_MAP()和 END_MESSAGE_MAP()之间添加消息映射。如下：

```
IMPLEMENT_DYNCREATE(CTest1View, CView)
BEGIN_MESSAGE_MAP(CTest1View, CView)
  //{{AFX_MSG_MAP(CTest1View)
  //系统本身的消息映射
  //}}AFX_MSG_MAP
  // Standard printing commands
  ON_MESSAGE(WM_HIDE,OnHide)
END_MESSAGE_MAP()
```

（4）定义消息处理函数的函数体。

在类的实现文件中的末尾添加函数体，假设添加的是 View 类的自定义消息，则形式如下：

```
void CTest1View::OnHide()
```

```
{
    //添加代码完成特定的操作
}
```

2.3　文档与视图

文档-视图（Document-View）体系结构是 MFC 的一大特色，它将数据的管理和显示分离；文档对象是用来容纳数据的，即管理应用程序的数据；而视图对象则是用来显示数据并且对数据进行编辑的，即用于显示文档并管理与用户的交互作用。实际上，视图在用户与文档之间起桥梁作用。打开文档时，视图类的实现文件中应该有读取的文档数据，并将其显示于视图中。用户在视图区绘制的图像也应该通过视图类的实现文件添加到适当的文档类成员中，这些数据存入文档的操作由文档串行化来实现。

本节将介绍文档怎么样维护应用程序的数据，怎样向用户显示这些数据及在应用程序运行时文档和视图对象之间是怎样进行对话的。

2.3.1　文档类结构分析

CDocument 是应用程序的“文档类”，负责装载和维护文档。文档可以是从文档到网络设备设置的任何内容。

通常，文档类 CDocument(O/C)是应用程序打开一个新文档或一个已存在的文档时创建的对象。文档类负责将一个文档赋给它的成员变量，并允许视图类编辑这些成员变量。一个文档包括从图形文件到可编程控制的任何内容。文档类派生于 CDocument，AppWizard 自动将其命名为 CXxxDoc，Xxx 是应用程序的名称。

当利用 AppWizard 建立一个应用程序样本时，Visual C++会自动提供一个文档对象的边框结构并把实现此结构的代码放入文件 HelloWorldDoc.cpp 中，其中 HelloWorld 是具体项目的名称。文档数据的管理是通过 CDocument 派生的类对象来进行的，这样，可以利用 CDocument 类的固有能力来管理数据的串行化和其他任务。CDocument 的一个特征是它通常用在由 AppWizard 创建的程序中。当利用 AppWizard 产生一个应用程序边框并且创建一个文件 HelloWorldDoc.cpp 内的对象时，AppWizard 将自动把菜单命令 Save、Save as 和 Open 与文档对象联系起来。

2.3.2　视图类结构分析

CView 是应用程序的“视图类”，负责为文档提供一个或多个视图。视图以图形方式显示文档数据，接收用户输入并将其解释成对文档的操作。编程时，对视图类的处理主要包括以下几个方面：

- 处理视图类的 OnDraw()成员函数，该函数负责提供文档数据；
- 将 Windows 消息与用户界面对象及视图类中的消息处理函数连接起来；
- 实现消息处理函数，以便解释用户的输入；
- 根据需要，在派生类视图中重设 CView 的其他成员函数，如重设 OnInitialUpdate，以便进行必要的视图初始化工作等。

此外，MFC 类库还提供了几个专用的派生视图类，利用这些视图类可以增强应用程序视图的功能。

- CScrollView 类：提供自动滚动和缩放显示功能。
- CFormView 类：提供可滚动的视图，用于显示由对话框控件组成的表单。

- CRecordView 类和 CdaiRecordView 类：用于在控件中显示数据库表中字段的表单视图。
- CEditView 类：使视图具有可编辑文本控件的特性，可以使用 CEditView 对象实现简单的文件编辑器。

在创建文档类的一个实例之后，将创建视图类 CWiew(O/C/W)的一个实例。视图类负责描述文档类的内容，也允许用户编辑文档。窗口分区类，即 CSplitterWnd，允许文档有多视图；这些视图可以由相同视图类的一些实例创建，也可以由完全不同的视图类实例创建。

AppWizard 允许视图类从几个基类（包括 CTreeView、CEditView、CRichEditView、CListView 等）中的一个派生，每一个基类赋予应用程序不同的功能集。所有这些类都派生于 CView 类，不管选择哪个基类，AppWizard 自动命名派生类为 CXxxView，这里的 Xxx 是应用程序名。

1．在视图中绘制

所有应用程序的绘制工作都是在视图的 OnDraw()成员函数中实现的。在 MFC AppWizard 中，已定义了 OnDraw()函数的基本结构，如例 2.4 所示。

例 2.4

```
void CHelloWorldView::OnDraw(CDC* pDC)
{
    pDC->TextOut(0,0,_T("Hello World!"));
    CHelloWorldDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;

    // TODO: add draw code for native data here
}
```

函数 OnDraw()通过调用文档成员函数来获取文档数据，并通过调用传给 OnDraw()的设备文本对象的成员函数来显示数据。当更改文档数据时，视图必须重新绘制以反映数据的更改。

成员函数 OnUpdate()使视图的整个客户区无效时，可以重设 OnUpdate()函数以便反映文档数据的更改情况。

CDocument 类的成员函数 OnUpdateAllViews()和 CView 类的成员函数 OnUpdate()都包含用于描述文档被修改信息的参数。OnUpdateAllViews()函数的原型为：

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

OnUpdate()函数的原型为：

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

如果所有的视图都要更新，则参数 pSender 为 NULL，否则为指向视图的指针；lHint 包含与修改有关的信息，可以传递任何数据，pHint 为指向保存修改信息的对象，可以传递从 CObject 类派生的任何对象的指针。

2．与用户交互作用

在视图类中，通常要定义消息处理函数，以便处理标准 Windows 消息（由鼠标和键盘动作产生）和命令（来自于菜单、工具栏按钮和快捷键）。

消息处理函数将用户动作或命令解释为数据输入、选择或编辑，例如，要处理“Edit”菜单下的剪切、复制、粘贴或删除等与剪贴板有关的命令，消息处理函数就要调用 CWnd 类中与剪贴板有关的成员函数，以便将选中的数据传进或传出剪贴板。

2.3.3　文档与视图的关系

文档与视图之间存在以下关系：

- 由 MFC AppWizard 创建的文档类是从 CDocument 类派生的，视图类是从 CView 类派生的；
- CDocument 类为应用程序定义的文档类提供基本功能，而 CView 类为应用程序定义的视图类提供基本功能；
- 视图和文档连接在一起，在文档与用户间起桥梁作用。视图在屏幕上显示文档数据并把用户输入转换成对文档的操作；
- 类库中文档与视图数据的显示及用户对数据所做的操作分离。数据的所有更改都通过文档类管理，而视图则调用这个接口来访问和更新数据。

具体来说，在文档、视图和应用程序框架之间包含了一系列非常复杂的相互作用过程。为了了解这一过程，首先要熟悉有关文档类和视图类中的几个重要的成员函数。

1．CView::GetDocument()函数

一个视图对象有且只有一个与之相关联的文档对象，它所包含的 GetDocument()函数允许应用程序在视图中得到与之相联系的文档对象。假设一个视图对象得到一个消息，表明用户已经在编辑控制中输入了新的数据，那么视图必须告诉文档对象相应地更新它的内部数据。GetDocument()函数返回的是指向文档的指针，利用这个指针就可以访问文档类或其派生类的成员函数及公共数据成员。

当 AppWizard 生成一个派生的 CView 类的时候，它同时也创建了一个特殊的 typesafe（保护类型）的 GetDocument()函数，它返回的不是一个 CDocument 指针，而是一个指向其派生类对象的指针。该函数是一个内联函数，形式如例 2.5 所示。

例 2.5

```
CTestDoc* CTestView::GetDocument()     // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CTestDoc)));
    return (CTestDoc*)m_pDocument;
}
```

当编译器在视图类代码中看到一个 GetDocument()调用时，它用派生类版本来代替 GetDocument()版本，这样就不需要把返回的指针转向派生的文档类。这是因为 CView 的 GetDocument()函数不是一个虚函数，下面的语句：

```
pView->GetDocument();
```

调用基类 GetDocument()函数，这样返回的是一个指向 CDocument 对象的指针。

2．CDocument::GetNextView()函数

一个文档对象可以和多个视图对象相关联，CDocument 对象的 GetNextView()成员函数用来在文档中得到指向视图列表的指针，但是由于一个文档拥有不止一个视图，所以文档对象有必要通过这个指针遍历视图列表，对每个视图都调用一次这个成员函数，这时常将 CDocument 类的另一个成员函数 GetFirstViewPosition()与 GetNextView()函数配合使用。GetFirstViewPosition()函数返回文档对象中视图列表中的第一个视图的位置，即 POSITION 值，而 GetNextView()负责返回下一个视图，程序的样式通常如例 2.6 所示。

例 2.6

```
CTestDoc *pTestDoc;          //CTestDoc 为 CDocument 的派生类
```

```
CTestView *pTestView          //CTestView 为 CView 的派生类

pTestDoc=pTestView->GetDocument();
POSITON   pos=pTestDoc->GetFirstViewPosition();
While (pos!=NULL)
{
    pTestView=(CTestView*)pTestDoc->GetNextView();
    //对视图进行操作
}
```

3．CDocument::UpdateAllViews()函数

如果由于某种原因文档数据发生了变化，那么必须通知所有的视图以便它们能够对所显示的数据进行相应的更新，这时就要使用 UpdateAllViews()函数了。如果 UpdateAllViews()是在一个派生文档类的一个成员函数中调用，那么它的第一个参数 pSender 的值为 NULL；如果在视图派生类的成员函数中调用 UpdateAllViews()，那么应该把 pSender 参数的值设为当前视图，如下所示：

```
GetDocument()->UpdateAllViews(this);
```

其中，非空参数使得应用程序框架不再通知当前视图，这里是假定当前视图已经更新了自己。函数有可选提示参数，用来给出专门视图和应用程序决定的关于更新视图的哪部分的消息。这是该函数的一个高级用法，其基本形式如下：

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

其中 lHint 和 pHint 就是提示参数，可以利用它们给视图提供一些特殊的与应用程序有关的信息，以便视图能决定应该更新哪些部分。

4．CView::OnUpdate()函数

当 UpdateAllViews()被调用时，视图是怎样被通知的呢？就是利用 CView 类的 OnUpdate()函数。OnUpdate()函数是一个虚函数，当应用程序调用 UpdateAllViews()函数时，应用程序框架会相应地调用所有视图的 OnUpdate()函数。当然，也可以在派生的 CView 类中直接调用这个函数。通常情况下，派生视图类的 OnUpdate()函数访问文档，得到文档的数据，然后更新视图数据的成员反映这些变化。作为选择，OnUpdate()可以使视图的一部分无效，导致视图的 OnDraw()函数使用文档数据来在窗口中重画。OnUpdate()函数的基本形式如例 2.7 所示。

例 2.7

```
void CTestView::OnUpdate(CView* pSender,LPARAM lHint=0L,CObject* pHint =NULL)
{
    CTestDoc* pTestDoc=GetDocument();
    ASSERT_VALID(pTestDoc);
    //加入视图的初始化代码
}
```

其中 lHint 和 pHint 两个提示信息是 UpdateAllViews()函数直接传递过来的。默认的 OnUpdate()函数将使整个窗口无效，开发人员可以重写该函数，可以选择定义一个由提示信息具体说明的较小的无效矩形。

5．CView:: OnInitialUpdate()函数

当应用程序启动，或用户从“File”菜单中选择了“New”或“Open”命令时，CView 类的 OnInitialUpdate()函数会被调用，该函数是一个虚函数。基类 CView 类的 OnInitialUpdate()函数版本除了调用 OnUpdate()之外不做任何事情；如果在派生的视图类中重载了这个函数，要确保视图类调用了基类的 OnInitialUpdate()函数或派生类的 OnUpdate()函数。

可以利用派生类的 OnInitialUpdate()函数对视图对象进行初始化。当应用程序启动后，应用程序框架在调用 OnCreate()之后立即调用 OnInitialUpdate()（前提是在视图类中映射了 OnCreate()）。OnCreate()函数只能被调用一次，而 OnInitialUpdate()函数可以被调用多次。

6．CDocument::OnNewDocument()函数

在一个文档对象首次被构造之后并且用户从一个 SDI 应用程序的“File”菜单中选择了“New”命令时，框架调用这个虚拟函数，这是初始化文档对象的数据成员的好时机。AppWizard 在派生文档类中生成一个无效的 OnNewDocument()函数，别忘了保留对基类函数的调用。

7．CDocument::DeleteContents()函数

当需要一个函数来删除文档的内容时，一方面，开发人员可以自己编写一个私有成员函数进行删除；另一方面，应用程序框架为 CDocument 类定义了一个 DeleteContents()函数。当文档被关闭的时候，应用程序框架会调用用户重载的 DeleteContents()函数，进行文档的清理工作。

这里，简单地总结一下视图与文档的交互流程：当用户在视图中对文档数据进行编辑时，视图会调用它自己的成员函数 GetDocument()来通知文档对象更新其内部数据；当文档对象修改数据之后，它会调用文档类的 UpdateAllViews()成员函数通知所有与之联系的视图进行强制更新显示，实际上，当应用程序调用 CDocument::UpdateAllViews()时，应用程序框架会紧接着自动调用 CView 类的 OnUpdate()函数，这个 OnUpdate()函数会先访问文档的数据，然后对视图进行更新。在默认情况下，OnUpdate()函数使整个窗口矩形失效，从而触发视图的 OnDraw()函数，进而重绘窗口。

2.3.4　SDI 应用程序分析

一个单文档界面应用程序只有一个窗口。如果应用程序依赖于磁盘文件“文档”，那么一次只能加载一个文档，类似于 Windows 的“记事本”程序。

当创建的是单文档应用程序时，AppWizard 会创建 4 个类，即 CTestApp、CMainFrame、CtestView 和 CTestDoc，分别为 CWinApp、CFrameView、CDocument 和 CView 类的派生类。

2.3.5　MDI 应用程序分析

一个多文档应用程序拥有多个子窗口。每个窗口单独对应于一个文档，类似于 Windows 的“记事本”程序。

当创建的是多文档应用程序时，AppWizard 会创建 5 个类，即 CTestApp、CMainFrame、CChildFrame、CtestView 和 CTestDoc，分别为 CWinApp、CMDIFrameWnd、CMDIChildWnd、CDocument 和 CView 类的派生类。

2.4　工具栏与状态栏

工具栏和状态栏都属于控件栏。所有的控件栏的基类都是 CControlBar。工具栏（Toolbar）以一组按钮的形式提供了操作菜单的快捷方式；状态栏（Statusbar）以文本或进度条的形式显示应用程序目前的基本状态。尽管 Visual C++为创建一个具备各种控件栏的应用程序提供了大量的帮助，但开发人员仍应该熟悉各种控件栏的编程，这是因为每一个应用程序都有自己独特的命令集。下面分别介绍各类控件栏的概念及编程要领。

2.4.1 工具栏

工具栏是 Windows 应用程序中另一个非常重要的图形界面元素，它提供了一组顺序排列的带有位图图标的按钮。通常，用户通过位图就可以知道这个按钮所要实现的功能，因此，操作工具栏比操作菜单更方便、快捷。在应用程序中，可以将常用的功能封装在工具栏中，工具栏是一个特殊的窗口对象，不仅可以停靠在主框架窗口的一侧，还可以浮动在窗口中。

新建一个 MFC 框架的应用程序时，不管是单文档应用程序还是多文档应用程序，只要在“MFC AppWizard（Advanced Features）”对话框中，选中“Docking toolbar”复选框，就会在应用程序中生成一个工具栏，下面将要介绍 AppWizard 如何生成这个工具栏，以及如何在工具栏编辑器中修改、设计工具栏。在这之前，先熟悉一下 Visual C++创建工具栏的方法。

一种方法是使用资源编辑器创建，过程如下：

（1）创建工具栏资源；

（2）构造 CToolBar 对象；

（3）调用成员函数 Create()创建 Windows 工具栏并将其与 CToolBar 对象链接；

（4）调用成员函数 LoadToolBar()装载工具栏资源。

另一种方法是直接调用成员函数创建，过程如下：

（1）构造 CToolBar 对象；

（2）调用成员函数 Create()创建 Windows 工具栏并将其与 CToolBar 对象链接；

（3）调用成员函数 LoadBitMap()装载包含工具按钮图像的位图；

（4）调用成员函数 SetButtons()设置按钮风格并使每个按钮与位图图像相关。

其中，CToolBar 类是 MFC 类库中专门用来处理工具栏的类，CToolBar 类是从 CControlBar 类直接继承来的。

与工具栏相关的 CToolBar 类的成员函数如表 2.2 所示。

表 2.2　　CToolBar 类的成员函数

函　数	说　明
SetSizes()	设置按钮及位图的大小
SetHeight()	设置工具栏的高度
SetBitMap()	设置位图图像
LoadBitMap()	从位图资源中调入工具栏位图
CommandToIndex()	返回指定命令 ID 的按钮索引
GetItemID()	返回指定索引的按钮或分隔符的命令 ID
GetItemRect()	获取索引项的指定比例的显示区域
GetButtonStyle()	获取按钮风格
SetButtonStyle()	设置按钮风格
GetButtonInfo()	获取按钮的 ID、风格和图像号
SetButtonInfo()	设置按钮的 ID、风格和图像号
GetButtonText()	获取显示在按钮上的文本
SetButtonText()	设置显示在按钮上的文本
Create()	创建一个 Windows 工具栏，并将其链接到 CToolBar 对象上，将其设置为默认的高度
CreateEx()	创建一个带有扩展风格的工具栏，并将其链接到 CToolBar 对象上，将其设置为默认的高度
LoadToolBar()	直接加载工具栏资源

下面将介绍 CToolBar 中几个比较重要的成员函数。

（1）Create()函数。

```
BOOL Create( CWnd* pParentWnd, DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_TOP, UINT nID = AFX_IDW_TOOLBAR );
```

该函数用于创建一个 Windows 工具栏，并将其链接到 CToolBar 对象上，将其设置为默认的高度。

第一个参数 pParentWnd，表示工具栏所在的父窗口。

第二个参数 dwStyle，为工具栏的样式，该参数可取如下的值：

```
CBRS_TOP　工具栏停靠在框架窗口的顶部
CBRS_BOTTOM　工具栏停靠在框架窗口的底部
CBRS_NOALIGN　父窗口大小改变后，工具栏位置不变
CBRS_TOOLTIPS　工具栏显示提示条
CBRS_SIZE_DYNAMIC　工具栏是动态的
CBRS_SIZE_FIXED　工具栏是固定的
CBRS_FLOATING　工具栏是浮动的
CBRS_FLYBY　在状态栏中显示工具栏按钮的提示信息
CBRS_HIDE_INPLACE　工具栏对用户不可见
```

该参数被默认设置为 WS_CHILD | WS_VISIBLE | CBRS_TOP，作为子窗口显示出来并停靠在框架窗口的顶部。

第三个参数 nID，指定工具栏的 ID 值。

（2）CreateEx()函数。

```
BOOL CreateEx(CWnd* pParentWnd, DWORD dwCtrlStyle = TBSTYLE_FLAT, DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_ALIGN_TOP, CRect rcBorders = CRect(0, 0, 0, 0), UINT nID = AFX_IDW_TOOLBAR);
```

该函数用于创建一个带有扩展风格的工具栏，并将其链接到 CToolBar 对象上，将其设置为默认的高度。

第一个参数 pParentWnd，表示工具栏所在的父窗口。

第二个参数 dwCtrlStyle，设置内嵌在工具栏上的 CToolBarCtrl 类的对象的扩展风格。该参数默认设置为 TBSTYLE_FLAT。

第三个参数 dwStyle，为工具栏的样式，见 Create()函数的 dwStyle 参数。

第四个参数 rcBorders，定义工具栏窗口边框的宽度，默认设置为（0,0,0,0），即无边框。

第五个参数 nID，指定工具栏的 ID 值。

（3）LoadBitmap()函数。

```
BOOL LoadBitmap( UINT nIDResource );
BOOL LoadBitmap( LPCTSTR lpszResourceName );
```

该函数用于从位图资源中调入工具栏位图，其中参数 nIDResource 表示将要装载的位图资源的 ID 号，参数 lpszResourceName 表示装载的位图资源的名称。

（4）SetButtonInfo()函数。

```
void SetButtonInfo( int nIndex, UINT nID, UINT nStyle, int iImage );
```

该函数可以设置命令按钮的 ID、风格及图像索引号，如果是间隔区，则参数 iImage 为间隔区的高度。

第一个参数 nIndex，表示要设置信息的按钮或间隔区的索引。

第二个参数 nID，指向命令按钮的 ID。

第三个参数 nStyle，指定按钮的风格，可以为下列值：

```
TBBS_BUTTON　标准命令按钮，一般为默认设置
TBBS_SEPARATOR　间隔区
```

```
TBBS_CHECKBOX   自动确认框
TBBS_GROUP   标记为一组按钮的开始
TBBS_CHECKGROUP   标记为一组确认框的开始
```

第四个参数 iImage，指定位图在位图资源中的索引。

（5）LoadToolBar()函数。

```
BOOL LoadToolBar( LPCTSTR lpszResourceName );
BOOL LoadToolBar( UINT nIDResource );
```

该函数可以直接加载工具栏资源，其中参数 lpszResourceName 表示要加载的工具栏资源的名称，参数 nIDResource 表示要加载的工具栏资源的 ID。

熟悉 CToolBar 类及其成员函数以后，再来看看 AppWizard 是如何创建工具栏并在窗口中显示的。

上面已经了解了创建工具栏的基本步骤，工具栏是停靠或浮动在主框架窗口上的，它和菜单、状态栏一样都是框架窗口的子窗口，因此可以想象，工具栏的创建应该是在框架窗口类的 OnCreate()函数中，OnCreate()函数代码如例 2.8 所示。

例 2.8

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;
    //创建工具栏窗口及加载工具栏资源
    if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE | CBRS_TOP | CBRS_GRIPPER |
CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||!m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;      // fail to create
    }

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

    // TODO: Delete these three lines if you don't want the toolbar to
    //  be dockable
    //使工具栏窗口可停靠
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    //使主框架窗口可被停靠
    EnableDocking(CBRS_ALIGN_ANY);
    //停靠工具栏到主框架窗口
    DockControlBar(&m_wndToolBar);

    return 0;
}
```

下面将分步讲述 Appwizard 生成工具栏的过程。除了第一步是在框架类的头文件中声明一个对象以外，其他步骤的实现均是在 CMainFrame 的 OnCreate()函数中。

（1）在 CMainFrame 类的头文件中声明一个 CToolBar 类的对象。

打开工作台的 Class View 页面，在 CMainFrame 框架类的下面有一个保护型的成员变量 m_

wndToolBar，双击该变量就可以看到它在类的头文件中的声明，如下：

```
class CMainFrame : public CFrameWnd
{ …
protected:   // control bar embedded members
      CStatusBar   m_wndStatusBar;
      CToolBar     m_wndToolBar;
      …
      };
```

（2）在 CMainFrame 类的实现文件中，即 OnCreate()函数中创建工具栏窗口。

```
m_wndToolBar.CreateEx(this,TBSTYLE_FLAT,WS_CHILD|WS_VISIBLE|CBRS_TOP|CBRS_GRIPPER|CBRS_TOOLTIPS|CBRS_FLYBY|CBRS_SIZE_DYNAMIC)
```

这段代码就是创建工具栏窗口的代码，是用 CToolBar 类的成员函数 CreateEx()来实现的。

（3）加载工具栏资源。

```
m_wndToolBar.LoadToolBar(IDR_MAINFRAME)
```

这一句代码就是利用 CToolBar 类的成员函数 LoadToolBar()来实现的，其中的资源 ID 号 IDR_MAINFRAME 就是加载的工具栏资源的 ID。那么这个工具栏资源又是怎样产生的。

打开工作台的 Resource View 页面，展开倒数第二个文件夹“ToolBar”，在它下面有一个 ID 为 IDR_MAINFRAME 的工具栏资源，这个工具栏资源与以前讲的菜单资源一样，都是由 MFC 的 AppWizard 自动生成的。

（4）使工具栏窗口可停靠。

```
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
```

函数 EnableDocking()是 CControlBar 类的成员函数，它的作用是使一个控制条，此处即为工具栏，可以停靠。函数原型如下：

```
void EnableDocking( DWORD dwStyle );
```

参数 dwStyle 指示工具栏是否可停靠及停靠的位置，它可以有下列值：

```
0 也就是不指定任何值，表示工具栏不可以停靠
CBRS_ALIGN_TOP 允许工具栏停靠在客户区的顶部
CBRS_ALIGN_BOTTOM 允许工具栏停靠在客户区的底部
CBRS_ALIGN_LEFT 允许工具栏停靠在客户区的左侧
CBRS_ALIGN_RIGHT 允许工具栏停靠在客户区的右侧
CBRS_ALIGN_ANY 允许工具栏停靠在客户区的任意一侧
CBRS_FLOAT_MULTI 允许多个控制栏在一个单一的小框架窗口中浮动
```

（5）使主框架窗口可被停靠。

```
EnableDocking(CBRS_ALIGN_ANY);
```

设定工具栏窗口可停靠以后，还要设置主框架窗口可被停靠，这是因为工具栏是要停靠在主框架窗口上的，如果主框架窗口不可被停靠，工具栏即使想停靠，也靠不上去。

为实现主框架窗口可被停靠，使用了 EnableDocking()函数，看起来和上一步中使用的函数完全相同，但是它们并不是同一个，也不是一家的。这一步中使用的 EnableDocking()函数是 CFrameWnd 类的成员函数，原型如下：

```
void EnableDocking( DWORD dwDockStyle );
```

参数 dwDockStyle 指定框架窗口的哪一侧可以被停靠，可以有以下几个值：

```
CBRS_ALIGN_TOP 允许客户区的顶部被停靠
CBRS_ALIGN_BOTTOM 允许客户区的底部被停靠
CBRS_ALIGN_LEFT 允许客户区的左侧被停靠
```

```
CBRS_ALIGN_RIGHT 允许客户区的右侧被停靠
CBRS_ALIGN_ANY 允许客户区的任意一侧被停靠
```

（6）停靠工具栏到主框架窗口。

```
DockControlBar(&m_wndToolBar);
```

当设置好工具栏可以停靠及主框架窗口可以被停靠以后，就可以将工具栏停靠到主窗口上了。DockControlBar()是 CFrameWnd 类的成员函数，它的作用是将一个控制条停靠在框架窗口的某一侧，这里指的是工具栏，该函数原型如下：

```
void DockControlBar( CControlBar * pBar, UINT nDockBarID = 0, LPCRECT lpRect = NULL );
```

第一个参数 pBar，指向要停靠的控制条。

第二个参数 nDockBarID，决定控制条停靠在窗口的哪一侧，可以是下列值：

```
    0 可以将控制条停靠在框架窗口允许被停靠的任意一侧，该值为默认设置。
AFX_IDW_DOCKBAR_TOP 将控制条停靠在窗口的顶部
AFX_IDW_DOCKBAR_BOTTOM 将控制条停靠在窗口的底部
AFX_IDW_DOCKBAR_LEFT 将控制条停靠在窗口的左侧
AFX_IDW_DOCKBAR_RIGHT 将控制条停靠在窗口的右侧
```

第三个参数 lpRect，决定控制条可以停靠在目的框架窗口的非客户区的什么位置，默认为空。

以上介绍了 AppWizard 生成工具栏窗口的 6 个步骤，下面将介绍如何在工具栏编辑器中修改工具栏资源。

1．利用工具栏编辑器设计工具栏按钮

（1）展开工作台资源面板中的 Toolbar 文件夹，双击下面的 ID 号为 IDR_MAINFRAME 的工具栏资源，在窗口的右侧显示了工具栏编辑器窗口，如图 2.20 所示。

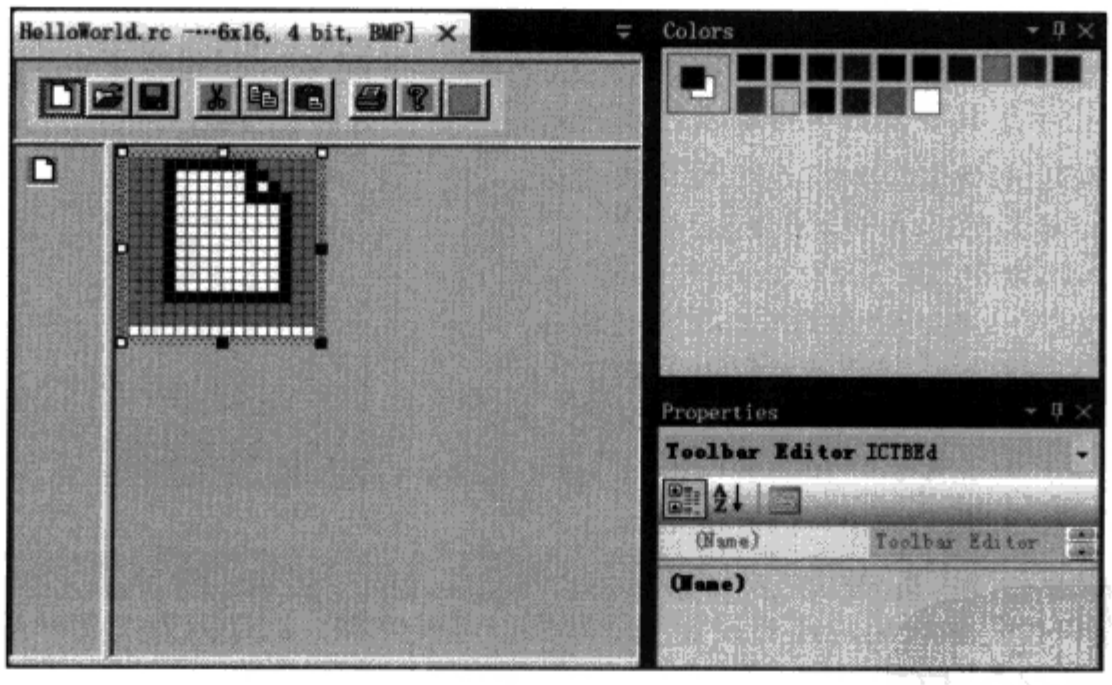


图 2.20　图标示图

在工具栏编辑器窗口的上面显示了 AppWizard 添加的工具栏，工具栏的下面是两个分隔的窗格，左侧窗格显示的是工具栏按钮上的位图的缩略图，右侧窗格显示的是工具栏按钮上的放大的位图，可以在这个窗格中修改位图。

（2）编辑位图按钮。

在工具栏编辑窗口中显示的工具栏的后面有一个空白的按钮，现在给这个按钮添加一个位图。可以在编辑器右侧的分隔窗格中编辑位图，例如在该位图上画一个红色的交叉线，如图 2.21 所示，当然，读者可以通过编辑窗口颜色栏和图形栏画出更丰富的位图。

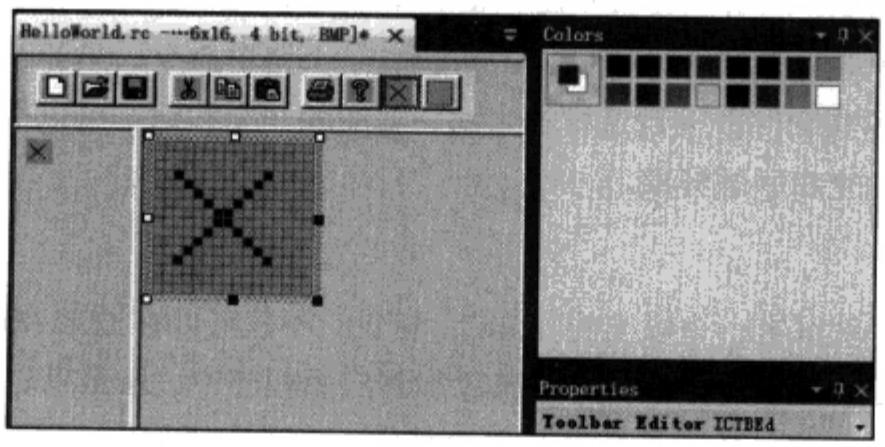


图 2.21　图标示图

（3）修改工具栏按钮的属性。

双击刚刚添加位图的工具栏按钮，就会在 Visual C++ 2010 右下方显示工具栏按钮属性面板，如图 2.22 所示。

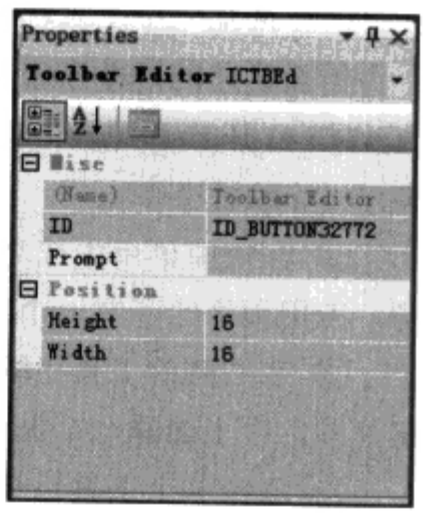


图 2.22　工具栏属性

在这个面板中可以设置工具栏按钮的 4 个属性。

- ID，表示工具栏按钮的 ID 号，它后面是一个下拉列表框，在该下拉列表框中应用程序为这个工具栏按钮自动分配了一个 ID 号，这个 ID 号看似一点意义也没有，一般不用这个默认的 ID。因为工具栏按钮都对应于菜单中的某个菜单项，为了和菜单项相关联，通常在下拉列表框中选择该工具栏按钮所对应的那个菜单项的 ID 作为其 ID 号。在这里，暂时不把它和菜单项关联，就使用默认值。
- Width，表示工具栏按钮的宽度，以像素为单位，默认宽度为 16 个像素。
- Height，表示工具栏按钮的高度，以像素为单位，默认高度为 16 个像素。

在修改工具栏的高度或宽度时，为达到整体美观的效果，还应该稍微调整一下其位图的大小。

- Prompt，表示工具栏按钮的提示信息。

添加该按钮后，运行程序，结果如图 2.23 所示。

可以看到，在工具栏的尾部多了一个添加的按钮，可是它现在是灰白的，这是因为在添加其 ID 属性的时候，用的是默认的 ID 号，而并没有给这个 ID 号添加消息处理，所以它现在不可用，假如把这个按钮的 ID 号改为菜单命令“工具栏”的 ID，即 ID_VIEW_TOOLS_BAR，就可以响应这个按钮了，如图 2.24 所示。

图 2.23　工具栏

图 2.24　工具栏

（4）调整顺序、增加分割线和删除按钮。

在工具栏编辑器的设计状态下，工具栏按钮之间有一个小小的间隔；在应用程序运行时，工具栏按钮之间的小间隔变成了一条竖线，这条竖线就是分割线。增加分割线的方法很简单，只需要在工具栏编辑器中，直接用鼠标拖动要分隔的按钮，往旁边移动一点点距离，然后释放即可。例如，要将刚刚添加的红色交叉线的按钮和带有问号位图的按钮分隔开，只需要用鼠标拖动红色交叉线的按钮往右侧移动一点然后释放即可，结果如图 2.25 所示。在按钮的旁边多了一个分割线。

图 2.25　工具栏

调整按钮顺序也很简单，只要用鼠标将按钮拖到用户希望的位置，然后释放鼠标即可。

删除按钮的方法和删除菜单的方法不同，若想删除某个菜单命令，只要选中该菜单命令，然后按【Delete】键即可，而删除按钮不能采用这样的方法。要删除一个按钮，可以用鼠标按住想要删除的按钮，然后将其拖到下面的空白区域释放。

2．新建一个工具栏

在一个应用程序中，可以有一个以上的工具栏，因此，不仅可以在原来 MFC 提供的工具栏的基础上增加按钮，还可以自己创建一个工具栏并将其添加到应用程序中。还是以这个上面 HelloWorld 程序为基础，给它添加一个工具栏，该工具栏中有 4 个按钮，分别为“加”、“减”、“乘”、“除”。下面简单介绍新建工具栏的步骤。

（1）增加一个工具栏资源。

前面讲述的是在原工具栏的基础上增加一个工具栏按钮，现在来增加一个新的工具栏资源。

① 打开 HelloWorld 工程，在工作台的资源面板（Resource View）中，右击“Toolbar”文件夹，弹出一个快捷菜单。

② 在快捷菜单中选择“Insert Toolbar”命令，在工作台的 Toolbar 的文件夹下多了一个工具栏资源，应用程序给了它一个默认的 ID 号 IDR_TOOLBAR1。

③ 在右侧的工具栏编辑器中显示了一个空白的工具栏按钮，单击这个工具栏按钮，设计下面的按钮位图，在上面画一个加号。

④ 双击这个工具栏按钮，弹出按钮属性面板，将这个按钮的 ID 定义为 ID_PLUS，并在“Prompt”一栏中写入提示信息“PLUS”，结果如图 2.26 所示。

⑤ 设计完该按钮之后，它后面又显示了一个空白的工具栏按钮，按照上面讲述的方法设计剩余的 3 个工具栏按钮，结果如图 2.27 所示。

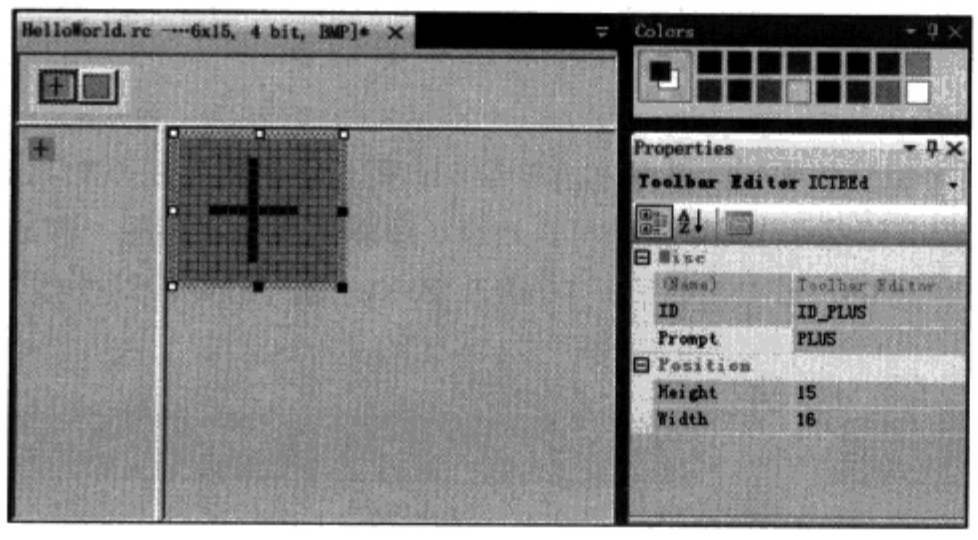


图 2.26　工具栏图标

图 2.27　工具栏视图

（2）在 CMainFrame 框架类的头文件中，声明一个 CToolBar 类的对象。

在工作台的“Class View”页面，右击 CMainFrame，在弹出的快捷菜单中选择“Add Variable”命令，进入“Add Member Variable Wizard”对话框，输入变量类型为“CtoolBar”，变量名为“m_newToobar”，访问类型设置为“Potected”保护型，然后单击“Finish”按钮完成。

或者直接找到框架类的头文件中声明变量的位置输入如下代码：

```
class CMainFrame : public CFrameWnd
{
…
protected:   // control bar embedded members
```

```
    CStatusBar   m_wndStatusBar;
    CToolBar     m_wndToolBar;
    CToolBar     m_newToolBar;
    …
    }
```

（3）在 CMainFrame 的 OnCreate()函数中完成工具栏窗口的创建和加载。

① 创建工具栏窗口并加载自定义的工具栏资源。代码如下：

```
    if (!m_newToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE | CBRS_RIGHT |  CBRS_GRIPPER  |
CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) || !m_newToolBar.LoadToolBar(IDR_TOOLBAR1))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;          // fail to create
    }
```

这次将自己创建的工具栏停靠在框架窗口的右侧。

其中 ID 号 IDR_TOOLBAR1 就是新创建的工具栏资源的 ID。

② 使 ID 号为 IDR_TOOLBAR1 的工具栏可停靠。代码如下：

```
    m_newToolBar.EnableDocking(CBRS_ALIGN_ANY);
```

③ 停靠工具栏在主框架窗口上。代码如下：

```
    DockControlBar(&m_newToolBar);
```

编辑完成后，编译运行。在运行窗口的框架右侧即可看到新创建的 4 个工具栏按钮。因为没有添加触发消息处理代码等的任何代码，所以单击 4 个按钮后，不会有任何操作进行。

3．显示/隐藏工具栏

在使用 Visual C++或 Word 等应用软件时，读者肯定都会注意到，如果右击工具栏或菜单栏的空白区域，就会弹出一个快捷菜单，这个快捷菜单中所显示的内容就是这个应用程序所包含的工具栏。旁边打钩的工具栏说明已经显示在窗口上了，而没有打钩的工具栏则还没有显示在窗口上。这样做的优点是节省屏幕的空间，试想，如果整个窗口都被工具栏占据，那么就没有编辑文字和编辑代码的位置了。

正是由于工具栏会占据窗口的部分位置，因此当一个应用程序中有多个工具栏时，应该提供一种可以使工具栏显示或隐藏的方法，以便在操作方便、快捷的前提下获取最大的客户区。

再次打开 HelloWorld 这个程序，会发现 AppWizard 自动生成的标准工具栏有隐藏/显示的功能，选择“查看”|“工具栏”命令就可以达到显示/隐藏工具栏的效果，而且当隐藏工具栏的时候，“工具栏”命令的左边的钩也被去掉了。但是在工程的文件中找不到显示/隐藏工具栏的代码，那是因为它们被封装在 MFC 应用程序框架中了。

一般来说，工具栏的初始显示状态是由创建工具栏时所设置的 WS_VISIBLE 属性所设置的，而这个属性是在调用 CToolBar 类的成员函数 CreateEx()或 Create()创建工具栏窗口时所设置的 dwStyle 参数中包含的。但是，这仅仅是工具栏的初始状态，用户还可以在应用程序中动态改变工具栏的显示和隐藏。

常用的显示和隐藏工具栏的方法有两种：一种是使用 CWnd::ShowWindow()函数来实现；另一种是使用 CFramWnd::ShowControlBar()函数来实现。

（1）使用 CWnd::ShowWindow()函数。

使用这种方法分为以下 3 步。

① 首先获取工具栏当前的显示状态，可以用 CWnd 类的成员函数 IsWindowVisible()，此函数没有

参数，如果窗口可见，则返回非 0 值；如果不可见，则返回 0。

② 根据当前状态来确定工具栏是否可见，可以使用 CWnd::ShowWindow()函数，此函数只有一个参数 nCmdShow，参数有两种取值：SW_SHOW 和 SW_HIDE。

③ 重新调整主框架窗口中控制栏的显示。

由于工具栏窗口被显示/隐藏之后，其他控制栏的停靠位置可能会有变动，因此需要调用 CMainFrame 类的 RecalcLayout()函数来重新调整。

（2）使用 CFramWnd::ShowControlBar()函数。

这种方法非常简单，因此是使用最广泛的。首先也是先确定工具栏当前的显示状态，然后直接调用 ShowControlBar()函数就可以了。该参数原型如下：

```
void ShowControlBar( CControlBar* pBar, BOOL bShow, BOOL bDelay );
```

第一个参数 pBar，指向被显示或被隐藏的工具栏的指针。

第二个参数 bShow，如果为 True，则工具栏显示；如果为 False，则不显示。

第三个参数 bDelay，如果为 True，则延迟工具栏的显示；如果为 False，则立刻显示。

2.4.2 状态栏

状态栏是 Windows 窗口上另一个很重要的界面元素。状态栏由输出面板或标识符组成。状态栏的主要作用是显示应用程序目前的基本状态，同时还会在选择菜单命令或单击工具栏按钮时显示提示信息。

创建状态栏一般遵循如下步骤：

（1）构造 CSatusBar 对象。

（2）调用成员函数 Create()创建状态栏窗口并将其与 CSatusBar 对象链接。

（3）调用成员函数 SetIndicators()设置标识符为数组中相应元素指定的值，装载每个标识符指定的字符串资源并将字符串设为标识符文本。

新建一个 MFC 框架的应用程序时，不管是单文档应用程序还是多文档应用程序，只要在“MFC Application Wizard”对话框中，选中“Initial Status Bar”复选框，就会在应用程序中生成一个状态栏，下面将要介绍 AppWizard 如何生成这个状态栏，以及开发人员如何修改状态栏。但在这之前，需要读者先熟悉 CStatusBar 类。

CStatusBar 类是 MFC 类库中专门用来处理状态栏的类，与工具栏类似，CStatusBar 类也是从 CControlBar 类直接继承来的。与状态栏相关的 CstatusBar 类的成员函数如表 2.3 所示。

表 2.3 CstatusBar 类的成员函数

函　数	说　明
CommandToIndex()	获取给定标识符 ID 的索引
GetItemID()	获取给定索引的标识符 ID
GetItemRect()	获取给定索引的显示区域
GetPaneInfo()	获取给定索引的 ID、风格和宽度
GetPaneStyle()	获取给定索引的标识符风格
GetPaneText()	获取给定索引的标识符文本
SetPaneInfo()	设置给定索引的 ID、风格和宽度
SetPaneText()	设置给定索引的标识符文本
Create()	创建一个状态栏窗口，并将其链接到 CstatusBar 对象上，同时设置初始字体和状态栏的高度
SetIndicatorsc()	设置指示器的 ID，并装入由每个 ID 指定的字符串资源，然后将指示器的文字设置为该字符串

下面简单介绍 CToolBar 类中的一些比较重要的成员函数。

（1）Create()函数。

```
BOOL Create( CWnd* pParentWnd, DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_BOTTOM, UINT nID = AFX_IDW_STATUS_BAR );
```

该函数用于创建一个状态栏窗口，并将其链接到 CStatusBar 对象上，同时设置初始字体和状态栏的高度。

第一个参数 pParentWnd，指向父窗口的指针。

第二个参数 dwStyle，指示窗口的风格，可以使用标准的 Windows 窗口风格，也可以使用下列风格：

CBRS_TOP 表示状态栏位于主框架窗口的顶部；

CBRS_BOTTOM 表示状态栏位于主框架窗口的底部；

CBRS_NOALIGN 当其父窗口的大小改变时，状态栏的位置不变。

第三个参数 nID，表示状态栏窗口的 ID 号。

（2）SetIndicators()函数。

```
BOOL SetIndicators( const UINT* lpIDArray, int nIDCount );
```

该函数用于设置指示器的 ID，并装入由每个 ID 指定的字符串资源，然后将指示器的文本设置为该字符串。该函数调用成功返回非 0 值，否则返回 0。

第一个参数 lpIDArray，表示指向一个 ID 数组的指针。

第二个参数 nIDCount，由 lpIDArray 指向的 ID 数组元素的个数。

（3）GetPaneInfo()和 SetPaneInfo()函数。

```
void GetPaneInfo( int nIndex, UINT& nID, UINT& nStyle, int& cxWidth ) const;
```

该函数用于返回指定指示器的 ID、风格和宽度。

第一个参数 nIndex，用来指示想要获取信息的指示器的序号。

第二个参数 nID，包含 ID 返回值 UINT 型引用。

第三个参数 nStyle，包含风格返回值 UINT 型引用。

第四个参数 cxWidth，包含宽度返回值的 INT 型引用。

```
void SetPaneInfo( int nIndex, UINT nID, UINT nStyle, int cxWidth );
```

该函数用于设置指定指示器的 ID、风格和宽度。

第一个参数 nIndex，用来指示想要设置的指示器的序号。

第二个参数 nID，指定状态指示器新的 ID 号。

第三个参数 nStyle，指定状态指示器新的风格，该参数支持下列值：

SBPS_NOBORDERS在窗格周围没有 3D 样式的边界；

SBPS_POPOUT 反转边界以使文字突出；

SBPS_DISABLED 使该状态指示器不可用；

SBPS_STRETCH 扩展指示器，填满不用的空间，一个状态栏中只能有一个指示器可以有这种风格；

SBPS_NORMAL 没有拉伸，没有边界，没有反转状态。

第四个参数 cxWidth，指定状态指示器新的宽度。

（4）GetPaneText()和 SetPaneText()函数。

```
CString GetPaneText( int nIndex ) const;
void GetPaneText( int nIndex, CString& rString ) const;
```

该函数用于得到状态栏中指示器中显示的文本内容。第一个函数形式返回一个包含状态指示器中显示的文本内容的 CString 类对象。第二个函数形式将返回的文本内容存放在参数 rString 中。

第一个参数 nIndex，表示要得到指示器文本的指示器的序号。

第二个参数 rString，是一个引用，包含返回的指示器的文本内容。

```
BOOL SetPaneText( int nIndex, LPCTSTR lpszNewText, BOOL bUpdate = TRUE );
```

该函数用于设置给定序号的指示器的文字内容。若函数调用成功，则返回一个非 0 值，否则返回 0。

第一个参数 nIndex，指定想要设置内容的状态指示器的序号。

第二个参数 lpszNewText，指定新的状态指示器的文本。

第三个参数 bUpdate，如果为 True，则该状态指示器在这些文本设置后将重画。

（5）GetItemRect()函数。

```
void GetItemRect( int nIndex, LPRECT lpRect ) const;
```

该函数可以获取状态栏窗格的矩形区域。该函数以像素为单位，复制由参数 nIndex 指定的状态指示器的坐标到参数 lpRect 中，坐标以状态栏左上角为原点。

第一个参数 nIndex，指定想要获取区域的状态指示器的序号。

第二个参数 lpRect，是一个 RECT 结构指针或 CRect 对象，将接受由 nIndex 指定的状态指示器的坐标。

熟悉 CStatusBar 类及其成员函数以后，再来看看 AppWizard 是如何创建并在窗口中显示状态栏的。

1．创建状态栏

与工具栏相同，状态栏也是主框架窗口的子窗口，因此它的主要创建步骤也是在框架窗口类的 OnCreate()函数中完成的，步骤如下：

（1）在 CMainFrame 类的头文件中声明 CStatusBar 类的对象。

```
class CMainFrame : public CFrameWnd
{
…
protected:     // control bar embedded members
               CStatusBar   m_wndStatusBar;
      CToolBar      m_wndToolBar;
…
      };
```

（2）在 CMainFrame 类的实现文件中，即 OnCreate()函数中创建状态栏窗口。

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

在 CMainFrame 类的 OnCreate()函数中，首先调用 CStatusBar 的 Create()函数创建状态栏窗口，即 m_ wndStatusBar.Create(this)；然后再调用 CStatusBar::SetIndicatiors()函数来分配窗格，即 m_wnd-StatusBar. SetIndicators(indicators, sizeof(indicators)/sizeof(UINT))；第一个参数 indicators 是一个指示器数组；第二个参数表明该状态栏的窗格总数。

一般情况下，一个应用程序只能有一个状态栏，因此如果要使应用程序显示开发人员自定义的提示信息，通常的做法是修改原有的状态栏，通过在上面添加窗格来实现。

2．修改状态栏

一个状态栏上有许多小窗格，这些小窗格用来显示各种提示信息。通常状态栏最左边的第一个窗格用作提示行，其长度是“可伸缩的”，即该窗格占据着其他窗格未使用的状态栏长度，剩余的窗格则是从右往左一个接一个地排列，如图 2.28 所示。

图 2.28　状态栏视图

可以看出，状态栏的窗格大体分为两类：一类是处于状态栏左侧的，用于显示菜单命令或工具栏命令提示信息或其他帮助信息的，称为提示行；另一类是在状态栏的右侧呈一格一格形式排列的，用于显示 SCROLL LOCK、NUMLOCK、CAPSLOCK 等开关状态或其他用户自添加状态等信息的，称为状态指示器。

（1）指示器数组。

在 MFC 应用程序中，增加或减少状态栏窗格数都是通过修改指示器数组来实现的。指示器数组 indicators 是用来说明状态栏窗格的数量及各个窗格的意义的。打开 CMainFrame 的实现文件，在.cpp 文件的开始部分有下面一段代码：

```
static UINT indicators[] =
{
    ID_SEPARATOR,           // status line indicator
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
```

不难看出，indicators 数组是一个静态的全局数组，它的第一项一般为 ID_SEPARATOR（显示一个提示行窗格），用来显示菜单命令或工具栏命令提示信息；后 3 项都是字符串 ID，用户可以在 String Table 字符串资源中找到这 3 个字符串，分别为“大写”、“数字”和“滚动”，而这 3 个字符串资源对应的正好是图 2.28 右侧的 3 个窗格，用来显示键盘状态。这些用于状态栏指示器的命令 ID 和字符串资源都是由系统预定义的。

（2）修改状态栏窗格。

熟悉指示器数组后，修改状态栏窗格就变得很容易了，只需要修改 indicators 数组就可以增加或减少窗格数。

首先，将 CMainFrame 类的实现文件中的 indicators 数组改成如下形式：

```
static UINT indicators[] =
{
    ID_SEPARATOR,           // status line indicator
    ID_SEPARATOR,
    ID_INDICATOR_STR,
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
```

由于，这里不仅加了一个提示行窗格 ID_SEPARATOR，还加了一个状态指示器窗格 ID_INDICATOR_STR，因此，应该在字符串资源表中再添加一个 ID 为 ID_INDICATOR_STR 的字符串资源，字符串内容为“添加一个窗格”。运行程序，则状态栏上多了两项内容，如图 2.29 所示。这部分

内容添加在 HelloWorld 工程代码中。

图 2.29　状态栏视图

3．在状态栏上显示鼠标指针的坐标、时钟和进度条

经过上面的学习，读者会觉得修改状态栏其实是一项很简单的工作，现在就来实际操作一下。将要新建一个 MFC 的单文档应用程序，该应用程序启动后，不仅在状态栏上显示进度条和当前的系统时间，而且在鼠标移动的过程中，还会在状态栏中显示鼠标指针的坐标。下面分别进行讲解。

（1）在状态栏上显示鼠标指针的坐标。

有很多应用软件都提供了在状态栏上显示鼠标指针的坐标的功能，如 HyperSnap-DX 抓图软件、CAD 制图软件等，这会给用户的操作会带来很大方便。

若想移动鼠标时，能够在状态栏中显示鼠标指针的坐标，那么至少要给应用程序添加鼠标指针移动的消息处理函数，并在该消息处理函数中改变状态栏相应窗格的显示。操作如下。

步骤 1：新建一个 MFC 单文档应用程序，工程名为 EX06_03 或由用户自定义。

步骤 2：添加一个新的状态栏窗格。

方法在前面讲过，在 CMainFrame 类的实现文件的开始部分，在 indicators 数组中添加一项 ID_SEPARATOR，结果如下：

```
static UINT indicators[] =
{
    ID_SEPARATOR,           // status line indicator
    ID_SEPARATOR,
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
```

步骤 3：在 CHelloWorldView 类中通过 ClassWizard 添加鼠标指针移动 WM_MOUSEMOVE 的消息处理函数 OnMouseMove()，并在 OnMouseMove()函数中添加代码完成鼠标指针坐标的显示。

完成鼠标指针坐标显示的方法有 3 种。

方法一：使用 SetWindowText()函数，在 OnMouseMove()函数中添加如下代码：

```
CString str;
str.Format("X=%d,Y=%d",point.x,point.y);
((CMainFrame*)GetParent())->m_wndStatusBar.SetWindowText(str);
```

此种方法需要将 CMainFrame 中的 m_wndStatusBar 改为 public 类型，而且还要在 CHelloWorldView 类的执行文件中加入 CMainFrame 类的头文件#include "MainFrm.h"。这样做虽然能达到效果，但破坏了 CMainFrame 的封装性，因此不建议使用这种方法。

其中函数 GetParent()用来取得该状态栏窗口的父窗口指针，即框架窗口指针，由于 m_wndStatusBar 声明在框架窗口类中，因此需要先取得框架窗口的指针。函数 SetWindowText()用来设置一个窗口显示的文本，在本句代码中是要设置 m_wndStatusBar 的显示文本。运行结果如图 2.30 所示。

图 2.30　状态栏视图

方法二：使用 SetMessageText()函数，在 OnMouseMove()函数中添加如下代码：

```
CString str;
str.Format("X=%d,Y=%d",point.x,point.y);
(CFrameWnd*)GetParent())->SetMessageText(str);
```

只需这几句代码即可，而且不需要将 m_wndStatusBar 改为 public 类型，也不需要在 CHelloWorldView 类的执行文件中加入#include "MainFrm.h"。运行结果同第一种方法。

其中函数 SetMessageText()是 CFrameWnd 类的成员函数，该函数可以用参数中提供的字符串文本来替换状态栏窗格中序号为 0 的窗格显示的内容，原型如下：

```
void SetMessageText( LPCTSTR lpszText );
```

其中参数 lpszText 就是想要替换的将要显示的文本。

方法三：使用 SetPaneText()函数，在 OnMouseMove()函数中添加如下代码：

```
CStatusBar* pStatus;
pStatus=(CStatusBar*)(AfxGetApp()->m_pMainWnd->GetDescendantWindow(AFX_IDW_STATUS_BAR));
pStatus->SetPaneText(1,str);
```

使用这种方法可以在状态栏中任意窗格上显示鼠标指针坐标，而不会像前两种方法，只能在第一个窗格中显示鼠标指针坐标。例如，这里就是在第二个窗格中显示，运行结果如图 2.31 所示。

图 2.31　状态栏视图

上面这段代码首先声明了一个 CStatusBar 类的指针变量，将该指针指向状态栏窗口，最后修改窗格显示的内容。其中全局函数 AfxGetApp()用来获取指向当前应用程序运行实例的指针，然后通过该指针获取该应用程序中的数据成员 m_pMainWnd(即主框架窗口)，最后在通过主框架窗口指针调用 CWnd 类的成员函数 GetDescendantWindow()来获取 ID 为 AFX_IDW_STATUS_BAR 的标准状态栏的指针。其实不管使用哪种方法，都应该先获取状态栏窗口的指针，然后再输出鼠标指针坐标。

如果读者细心一些就会发现，其实只有第三种方法使用了在 indicators 数组中添加的指示器 ID_SEPARATOR 一项，如图 2.31 所示（第三种方法）显示在第二个窗格中，而前两种方法并没有使用自己添加的 ID_SEPARATOR 一项，它是占用了原来的 indicators 数组中第一项 ID_SEPARATOR，如图 2.30 所示（第二种方法）显示在第一个窗格，而自定义的 ID_SEPARATOR 一项空闲在那儿。因此，若是采用前两种方法，甚至可以不在 indicators 数组中增加 ID_SEPARATOR。即使多增加了一项 ID_SEPARATOR，也不会使用，只会在鼠标指针坐标显示的后面多一个空白窗格，就像在图 2.30 中（第二种方法）所看到的。

本例中还是让鼠标指针坐标在第一个窗格中显示，这是因为后面还要在状态栏中显示进度条和当前机器的系统时间，若把这两种信息放在鼠标指针坐标的后面显示，看起来更和谐、美观一些，这里就要把之前在 indicators 数组中添加的 ID_SEPARATOR 暂时注释掉，让它去占用原有的 ID_SEPARATOR，以节省空间。

（2）在状态栏上显示时钟。

显示鼠标指针坐标之后，再来看看怎么在状态栏上显示系统时钟。要想在状态栏上显示的时间和机器的系统时间保持一致，那么至少应该在应用程序中添加一个计时器消息，并在该消息处理函数中改变状态栏相应窗格的显示。操作如下。

步骤 1：打开 HelloWorld 工程，添加一个新的状态栏指示器。

在 CMainFrame 类的实现文件的开始部分，修改 indicators 数组的内容，在其中添加一项 ID_INDICATOR_TIME，结果如下：

```
static UINT indicators[] =
```

```
{
    ID_SEPARATOR,           // status line indicator
    ID_INDICATOR_TIME,
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
```

增加一个字符串资源，ID 号为 ID_INDICATOR_TIME，字符串内容为“显示时间”。

打开工作台的资源面板，打开“String Table”文件夹，双击文件夹下的“String Table”字符串资源表，在右侧窗口中双击最后一项空白项，弹出“String Properties”对话框，在该对话框中的“ID”右侧输入“ID_INDICATOR_TIME”，在“Caption”右侧输入“显示时间”。

步骤 2：设置计时器并显示时间。

想要程序一启动就让状态栏显示时间，因此应该在 CMainFrame::OnCreate()函数中设置定时器并显示时间，代码如例 2.9 所示。

例 2.9

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{ …
    SetTimer(1,1000,NULL);
    CTime time=CTime::GetCurrentTime();
    CString str=time.Format("%H:%M:%S");
    CClientDC dc(this);
    CSize sz=dc.GetTextExtent(str);
    m_wndStatusBar.SetPaneInfo(1,ID_INDICATOR_TIME,SBPS_POPOUT,sz.cx);
    m_wndStatusBar.SetPaneText(1,str);
…
}
```

首先使用 SetTimer()函数设置计时器，第一个参数为计时器的标识，当一个应用程序中有多个计时器时，就用该参数来识别；第二个参数是时间值，指定相隔多少时间触发一次 OnTimer 事件；最后一个是处理 WM_TIME 消息的回调函数，一般为空。

在 MFC 类库出现之前，Windows 使用一个名为 SYSTEMTIME 的数据结构来表示时间，在这个结构中包括年、月、日、时、分、秒、毫秒、星期等成员；而在 MFC 类库中，由 CTime 类封装了这个结构及一些对时间的操作，与 CString 类相同，CTime 类也没有基类。GetCurrentTime()和 Format()都是 CTime 类的成员函数，分别用来获取当前系统时间和将时间按照给定格式写入一个字符串。

其中 sz=dc.GetTextExtent(str)这句代码是为了获取包含时间的字符串文本的宽度，因为这里并不知道要设置的窗格的宽度是多少，如果宽度设置大了，会影响空间和视觉，如果设置小了，则不能够完全显示文本，因此，采用这种方法来取得文本的宽度，进而再调用函数 SetPaneInfo()来设置要显示时间的窗格的宽度。在 indicators 数组中，序号是从 0 开始排列的，添加的 ID_INDICATOR_TIME 为第二项，因此其序号为 1。

最后，调用 SetPaneText()函数，将时间显示在序号为 1 的状态栏窗格中。

步骤 3：处理 WM_TIME 消息，并在 OnTimer 事件中不断获取当前时间并显示。

通过 ClassWizard，在 CMainFrame 类中添加 WM_TIME 消息。

在 WM_TIME 消息的消息处理函数 OnTimer()添加如下代码：

```
CTime time=CTime::GetCurrentTime();
CString str=time.Format("%H:%M:%S");
m_wndStatusBar.SetPaneText(1,str);
```

运行程序，在状态栏第二个窗格中显示出系统时间并且该时间随着系统时间的改变而改变，如图 2.32 所示。

图 2.32　状态栏视图

4．在状态栏上显示进度条

在前两项操作的基础上，再在状态栏上显示一个进度条。在状态栏上显示进度条就不会像前两项操作那么简单了，需要另外定义一个 CProgressCtrl 类的变量来产生进度条，具体操作如下。

步骤 1：在 CMainFrame 中定义成员变量，用于产生进度条。

在 CMainFrame 类的头文件中声明 CProgressCtrl 类的对象，如下：

```
CProgressCtrl   m_Progress;
```

步骤 2：添加一个新的状态栏指示器，其 ID 为 ID_INDICATOR_PROG，并在资源面板中再加入一个字符串资源，其 ID 同样为 ID_INDICATOR_PROG，字符串文本内容为“显示进度条”。

步骤 3：在 CMainFrame 中加入 WM__PAINT 的响应函数 OnPaint()，因为时间每变化一次或者进度条每前进一块，就会重画窗口，因此在 OnPaint()函数中显示进度条。代码如例 2.10 所示。

例 2.10

```
1    CRect rect;
2    m_wndStatusBar.GetItemRect(2,&rect);
3    if(m_Progress.m_hWnd)
4    {
5        m_Progress.SetWindowPos(NULL,rect.left,rect.top, rect.Width(),rect.Height(),SWP_NOZORDER);
6    }
7    else
8    {m_Progress.Create(WS_CHILD|WS_VISIBLE,rect,&m_wndStatusBar,111);}
```

第 2 行代码调用 GetItemRect()函数来取得显示进度条的窗格的大小，其中第一个参数设置为 2，是因为进度条指示器在 indicators 数组中的序号为 2；第 5 行代码调用函数 SetWindowPos()来调整进度条的位置和大小；第 9 行代码用的 Create()函数是 CProgressCtrl 类的成员函数，用来创建一个进度条。

步骤 4：在 OnTimer()函数中，使进度条向前走，代码如下：

```
m_Progress.StepIt();
```

其中 StepIt()是 CProgressCtrl 的成员函数，使进度条向前走一步并且重画进度条。

至此，关于状态栏的操作就完成了，运行结果如图 2.33 所示。

图 2.33　状态栏视图

2.5　MFC 新功能——Office 2007 风格程序分析

MFC 的 Office 2007 风格程序主要是通过 Microsoft Visual Studio 2010 SP1 中 MFC 的 Ribbon 功能来实现的。

与传统的菜单式用户界面相比较，Ribbon 界面的优势主要体现在如下几个方面：

- 所有功能有组织地集中存放，不再需要查找级联菜单、工具栏等；

- 更好地在每个应用程序中组织命令;
- 提供足够显示更多命令的空间;
- 丰富的命令布局可以帮助用户更容易地找到重要的、常用的功能;
- 可以显示图示，对命令的效果进行预览，例如改变文本的格式等;
- 更加适合触摸屏操作。

虽然从菜单式界面到 Ribbon 界面有一个漫长的熟悉的过程，但是一个不争的事实是，Ribbon 界面正在被越来越多的人接受，越来越多的软件开发商开始抛弃传统的菜单式界面，转而采用 Ribbon 界面。Visual Studio 2010 作为面向下一代 Windows 平台的开发工具，自然是对 Ribbon 界面全面支持。

下面看看如何利用 Visual Studio 2010 开发面向 Windows 7 的 Ribbon 界面。

1. 创建 MFC 应用程序项目

实际上，Ribbon 界面（Office 2007 风格的界面）的开发早在 2008 年就随着 Visual C++ Feature Pack 被引入到 Visual Studio 2008 中。在 Visual Studio 2010 中，Ribbon 界面的开发更是得到了全面的支持，使得 Ribbon 界面的开发更加简单、高效。

首先，启动 Visual Studio 2010，创建一个基于 MFC 的应用程序，选择“MFC Application”项目模版。

在接下来的 MFC 应用程序向导对话框中，就可以对项目的可视化风格进行选择和配置。Visual Studio 2010 支持 MFC 风格、标准的 Windows 风格，Visual Studio 2005 风格和 Office 2007 风格，而 Office 2007 风格就是要创建的 Ribbon 界面。

在接下来的向导对话框中，可以选择命令栏（工具栏、菜单栏）的样式，这里当然选择使用 Ribbon 了。当然，为了与旧有的系统保持兼容，Visual Studio 2010 也支持传统的命令式界面，如果用户比较保守，想继续使用传统的菜单式界面，那么也可以选择使用菜单栏和工具栏。

为了支持丰富的 Office 2007 界面风格，MFC 默认情况下为文档应用程序添加了类似于 Outlook 风格的导航面板（navigation pane）和标题栏（caption bar），由于这两者并不是我们关注的重点，为了更好地展示 Ribbon 界面，这里就去掉这两个多余的面板。

至此，针对新项目的设置就完成了，单击“Finish”按钮关闭应用程序向导，Visual Studio 2010 就会按照设置创建相应的 MFC 应用程序解决方案。编译运行这个解决方案，即可得到第一个具有 Ribbon 界面的应用程序。

2. Ribbon 界面的构成

为了支持新的界面风格的开发，MFC 提供了很多新的、经过扩展的应用程序类和界面类等。例如在实例程序中使用的 CribbonApp、CMainFrame 就是分别从 CWinAppEx、CMDIFrame WndEx 派生的。而 CWinAppEx 和 CMDIFrameWndEx 又是从传统的 CWinApp 和 CMDIFrameWnd 派生，在原有类的基础上，添加了对新的界面风格的支持。图 2.34 所示为支持新的界面风格的框架类。

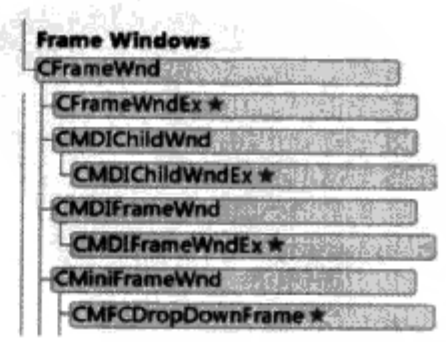


图 2.34　支持新的界面风格的框架类

熟悉 MFC 界面开发的读者应该都知道，应用程序的 CMainFrame 类负责整个主框架界面的创建和管理，在传统的菜单式界面下，它负责创建和管理菜单栏、状态栏、工具栏等。同样，在 Ribbon 界面下，它负责 Ribbon 界面的创建和管理。代码如例 2.11 所示。

例 2.11

```
class CMainFrame : public CMDIFrameWndEx
```

```
    {
        DECLARE_DYNAMIC(CMainFrame)
public:
        CMainFrame();

        // Attributes
public:

        // Operations
public:

        // Overrides
public:
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

        // Implementation
public:
        virtual ~CMainFrame();
        #ifdef _DEBUG
            virtual void AssertValid() const;
            virtual void Dump(CDumpContext& dc) const;
        #endif

protected:  // control bar embedded members
        CMFCRibbonBar     m_wndRibbonBar;
        CMFCRibbonApplicationButton m_MainButton;
        CMFCToolBarImages m_PanelImages;
        CMFCRibbonStatusBar  m_wndStatusBar;

        // Generated message map functions
protected:
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnWindowManager();
        afx_msg void OnApplicationLook(UINT id);
        afx_msg void OnUpdateApplicationLook(CCmdUI* pCmdUI);
        DECLARE_MESSAGE_MAP()

        void InitializeRibbon();
};
```

从上面的代码中可以看到，在 Ribbon 界面下，CMainFrame 类包含 3 个成员变量，即 m_wndRibbonBar、m_MainButton、m_wndStatusBar，分别用于控制 Ribbon 界面的命令面板、应用程序按钮和状态栏。清楚了各个变量对应的 Ribbon 界面元素后，就可以操作相应的成员变量，在 Ribbon 界面上添加所需的内容。

3．处理菜单资源和消息响应函数

Ribbon 界面和传统的菜单式界面的最大的差别就是用新的 Ribbon Bar 代替传统的菜单栏。可以看到，在默认情况下，Ribbon Bar 已经拥有了一个名为“Home”的命令分组（Category），其中又包含多个面板（Panel），每个面板中有一个或者多个命令按钮（Ribbon Button）。下面就来看看如何在 Ribbon Bar 上添加一个新的命令分组，然后在其中添加新的命令按钮以执行相应的功能。

按照传统的菜单式界面的开发流程，总是先编辑菜单资源，然后再创建相应的菜单命令消息响应函数，最后再实现这个消息响应函数，完成相应的任务。在新的 Ribbon 界面中，还是要先按照这样的流程，完成菜单资源的编辑，实现消息响应函数等，所以先打开 Visual Studio 2010 的资源编辑器，在 IDR_RibbonUITYPE 菜单资源中添加两个新的菜单命令，如图 2.35 所示。

然后，分别为这两个菜单命令添加相应的消息响应函数，如图 2.36 所示，代码如例 2.12 所示。

例 2.12

```
// CRibbonUIView message handlers
void CRibbonUIView::OnEditSayhello()
{
    // TODO: Add your command handler code here
    AfxMessageBox(_T("Hello RibbonUI!"));
}

void CRibbonUIView::OnEditSaygoodbye()
{
    // TODO: Add your command handler code here
    AfxMessageBox(_T("Goodbye RibbonUI!"));
}
```

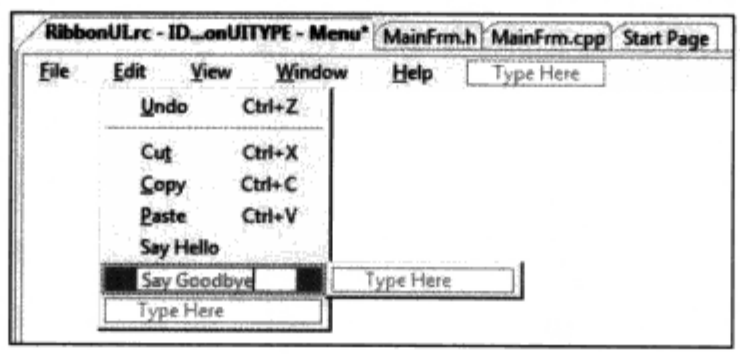


图 2.35　菜单视图

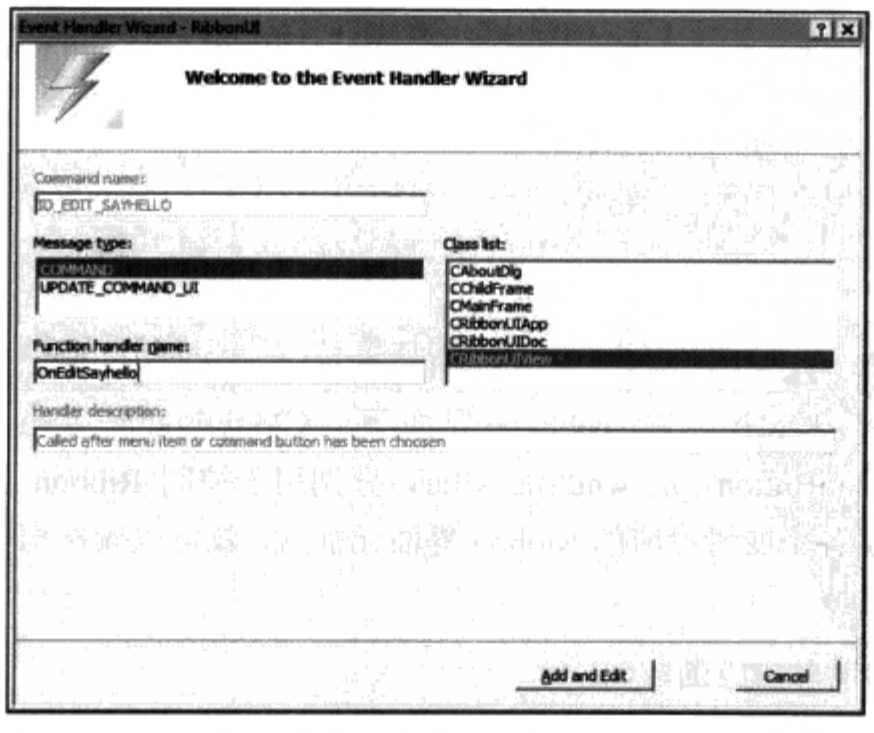


图 2.36　消息对应

4．添加 Ribbon 按钮

到现在为止，已经按照菜单式界面的开发流程，完成了资源的编辑和消息函数的实现。如果这时

编译运行解决方案，在程序界面上看不到任何变化，这是因为还没有把这些命令绑定到 Ribbon Bar 的命令按钮上。在 CMainFrame 类中，提供了一个专门的函数 CMainFrame::InitializeRibbon()来完成 Ribbon 界面的初始化工作，所以在这个函数中添加如例 2.13 所示的代码，创建新的命令分组，然后将命令按钮映射到刚刚创建的菜单命令。

例 2.13

```
void CMainFrame::InitializeRibbon()
{
    BOOL bNameValid;
        ///…
        ////////////////////////////////////////////////////
        // 添加一个命令分组(Category)"RibbonUI Category"
    CMFCRibbonCategory  *pRibbonUICategory  =m_wndRibbonBar.AddCategory  (_T("RibbonUI  Category"), IDB_
WRITESMALL, IDB_WRITELARGE);
        // 添加一个面板（Panel）
        CMFCRibbonPanel *pTestPanel = pRibbonUICategory->AddPanel(_T("RibbonUI Panel"), m_PanelImages.ExtractIcon(1));
    // 在面板上添加 Ribbon 命令按钮（CMFCRibbonButton）
        pTestPanel->Add(new CMFCRibbonButton(ID_EDIT_SAYHELLO, _T("Say Hello"), 0, 0));
        pTestPanel->Add(new CMFCRibbonButton(ID_EDIT_SAYHELLO, _T("Say Hello"), 1));
        pTestPanel->Add(new CMFCRibbonButton(ID_EDIT_SAYGOODBYE, _T("Say Goodbye"), 2));
        ////////////////////////////////////////////////////
…
}
```

在这段代码中，首先利用 Ribbon Bar 的 AddCategory()函数添加了一个新的命令分组（Category），也就是 Ribbon Bar 上的一个新的标签页；然后在这个新创建的标签页中，添加了一个新的面板（Panel）；最后创建了 3 个 Ribbon 命令按钮（CMFCRibbonButton），并将它们放置到面板中。通过指定与菜单资源中的菜单命令相同的资源 ID，将这些 Ribbon 命令按钮与之前创建的菜单命令一一对应起来。当单击某个 Ribbon 命令按钮时，就会执行相应的菜单命令响应函数，这样，就实现了菜单命令在 Ribbon Bar 上的绑定，同时在界面上也能看到刚刚添加的命令按钮。程序运行结果如图 2.37 所示。

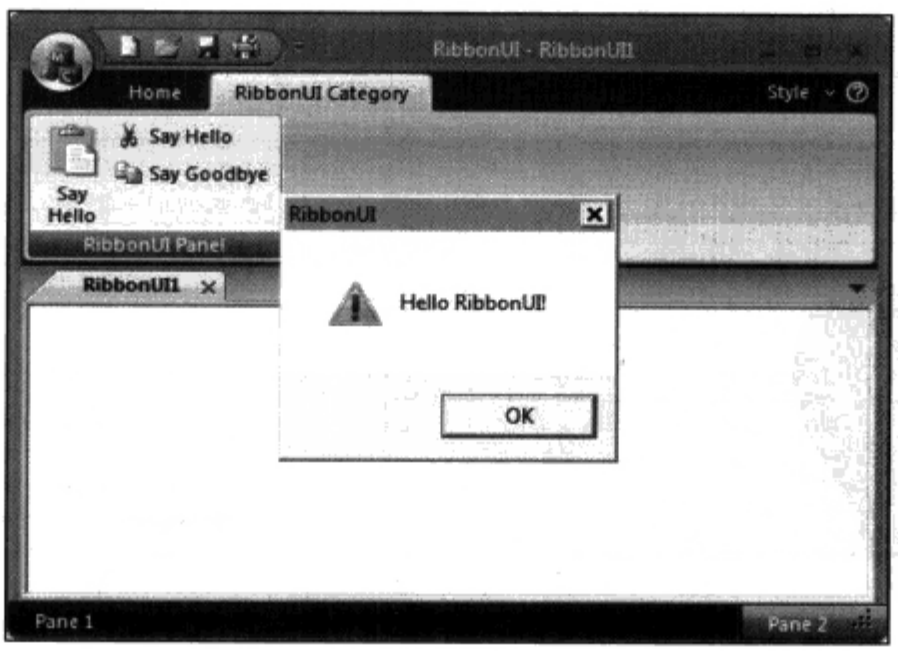


图 2.37　效果显示图

本节只是简单地向读者介绍了 Ribbon 界面的基本概念和大致的开发流程，要想开发与 Office 2007 一样专业的 Ribbon 界面，还有很长的一段路要走。

2.6 MFC 新功能——Visual Studio 风格程序分析

Visual Studio 风格的程序正如 Windows Office 2007 风格的程序，也是由 Ribbon 界面实现的。为了支持 Ribbon 界面，Visual Studio 2010 为开发人员提供了很多控件，除了上节介绍的命令按钮（CMFC-RibbonButton）之外，还有工具栏（CMFCRibbonGallery）、颜色按钮（CMFCRibbonColorButton），编辑框（CMFCRibbonEdit）、进度条（CMFCRibbonProgressBar）等。合理地使用这些控件，可以创建丰富的 Ribbon 界面，增强软件的可用性。下面就介绍一下如何使用这些控件，来创建更加丰富的 Ribbon 界面，完成更加复杂的交互任务。

为了更好地理解和创建 Ribbon 界面，在开始具体地介绍各种控件之前，先来了解一下 Ribbon 界面的结构层次。上节介绍了 Ribbon 界面主要由 Ribbon 面板（CMFCRibbonBar）构成，而 Ribbon 面板主要分为 3 个层次。

1. 分类（CMFCRibbonCategory）

很明显，“分类”就是作用相近的一类命令的组合，如图 2.38 所示。例如，在 Word 2007 的 Ribbon 界面中，微软将与插入元素相关的命令都放在“Insert”页面中，当用户想在 Word 文档中插入其他元素时，只要切换到这个页面就可以找到所需的命令。在形式上，“分类”表现为 Ribbon 面板上的一个 Tab 页面。开发人员可以使用函数 AddCategory()在 Ribbon 面板上添加一个新的“分类”，代码如例 2.14 所示。

例 2.14

```
// 添加一个命令分组(Category)"RibbonUI Category"
CMFCRibbonCategory *pRibbonUICategory = m_wndRibbonBar.AddCategory(_T
("RibbonUI Category"), IDB_WRITESMALL, IDB_WRITELARGE);
```

2. 面板（CMFCRibbonPanel）

“面板”是“分类”的下一个层次，是联系更加紧密的一组命令的组合，如图 2.39 所示。面板总是被放置在某个“分类”中，被“分类”包含。同时，“面板”又是一个容器，它包含着它的下一个层次“元素”。可以通过 AddPanel()函数在“分类”中添加新的“面板”：

例 2.15

```
// 添加一个面板（Panel）
CMFCRibbonPanel *pTestPanel = pRibbonUICategory->AddPanel(_T("RibbonUI Panel"), m_PanelImages.ExtractIcon(1));
```

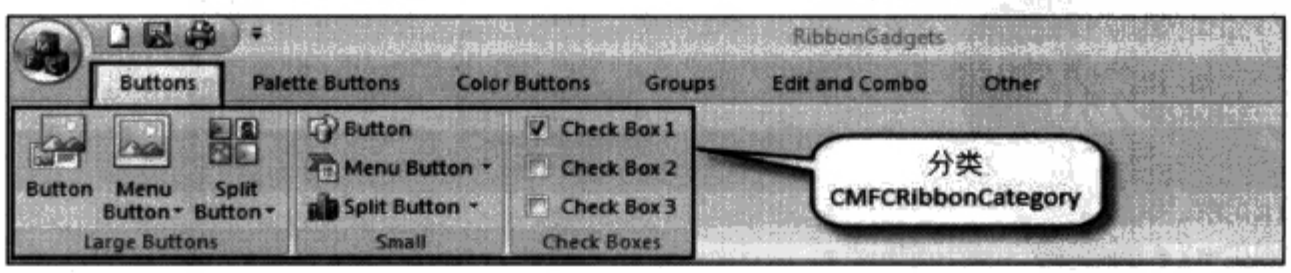


图 2.38　分类显示图

3. 元素（CMFCRibbonBaseElement）

“元素”就是通常意义上的控件，这些控件根据各自的功能，被分组放置在各个“面板”上，负责完成具体的交互任务，如图 2.40 所示。Visual Studio 2010 提供的 Ribbon 界面“元素”主要包括命令按

钮（CMFCRibbonButton）、工具栏（CMFCRibbonGallery）、颜色按钮（CMFCRibbonColorButton）、编辑框（CMFCRibbonEdit）、进度条（CMFCRibbonProgressBar）等。这些类都派生自类 CMFCRibbonBaseElement。

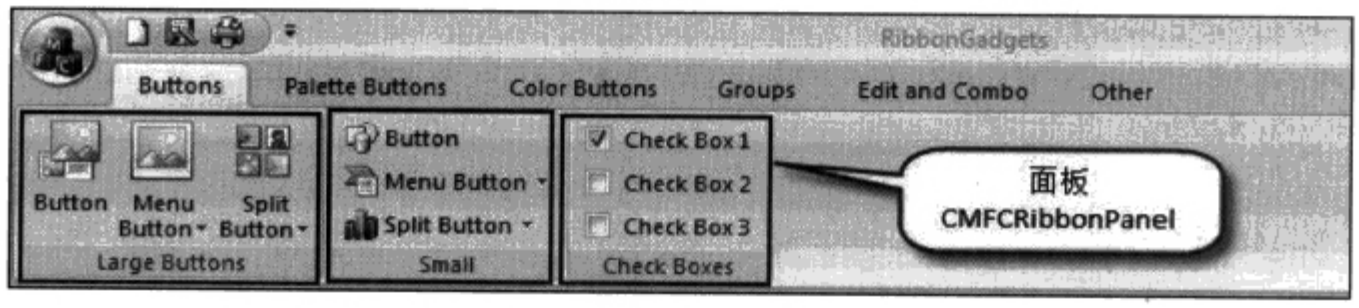


图 2.39　面板显示图

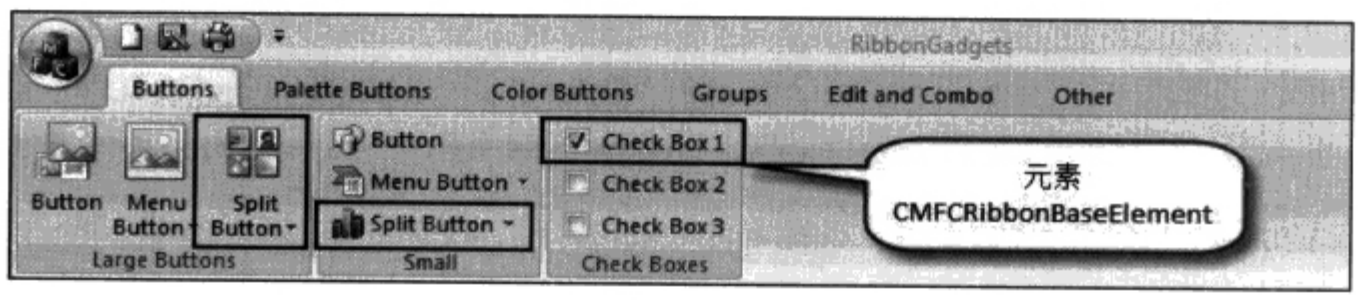


图 2.40　元素显示图

下面就来详细介绍各种 Ribbon 控件的使用，如图 2.41 所示。

（1）命令按钮。

命令按钮是最常用的 Ribbon 控件，用户通常都是通过命令按钮来发送某个命令，执行某个动作。它代替了过去的菜单命令，成为使用最频繁的 Ribbon 控件。在 Ribbon 界面中，主要有 3 种形式的命令按钮：大图标按钮，小图标按钮及表示选择的复选按钮（CheckBox）。

按照前面介绍的 Ribbon 界面开发流程，需要先准备菜单资源、图标资源、实现消息响应函数等，这里不再赘述这一过程，而把重点放在如何创建 Ribbon 界面上。例 2.16 分别演示了这 3 种按钮的创建过程。

例 2.16

```
// 创建一个新的面板，用于放置大图标按钮
CMFCRibbonPanel* pPanel1 = pCategory->AddPanel(_T("Large Buttons"));

// 创建按钮
CMFCRibbonButton* pBtn1 = new CMFCRibbonButton(ID_RIBBON_BTN_1, _T("Button"), 0, 0);
// 指定使用大图标
pBtn1->SetAlwaysLargeImage();

// 将按钮添加到面板中
pPanel1->Add(pBtn1);

CMFCRibbonButton* pBtn2 = new CMFCRibbonButton(ID_RIBBON_BTN_2, _T("Menu Button"), 1, 1);

// 可以通过 SetMenu()函数为按钮设置一个子菜单
pBtn2->SetMenu(IDR_RIBBON_MENU_1);
pBtn2->SetAlwaysLargeImage();
pPanel1->Add(pBtn2);
```

```
CMFCRibbonButton* pBtn3 = new CMFCRibbonButton(ID_RIBBON_BTN_3, _T("Split Button"), 2, 2);
pBtn3->SetMenu(IDR_RIBBON_MENU_1, TRUE);
pBtn3->SetAlwaysLargeImage();
// 可以通过 RemoveSubItem()和 AddSubItem()动态地改变按钮的子项目
pBtn3->RemoveSubItem(0);
pBtn3->AddSubItem(new CMFCRibbonButton(ID_RIBBON_MBTN_1, _T("Item 1"), 2), 0);
pPanel1->Add(pBtn3);

// 创建新的面板，用于放置小图标按钮
CMFCRibbonPanel* pPanel2 = pCategory->AddPanel(_T("Small"));

// 创建小图标按钮
CMFCRibbonButton* pBtn4 = new CMFCRibbonButton(ID_RIBBON_BTN_4, _T("Button"), 3);
pPanel2->Add(pBtn4);

CMFCRibbonButton* pBtn5 = new CMFCRibbonButton(ID_RIBBON_BTN_5, _T("Menu Button"), 4);
pBtn5->SetMenu(IDR_RIBBON_MENU_1);
pPanel2->Add(pBtn5);

CMFCRibbonButton* pBtn6 = new CMFCRibbonButton(ID_RIBBON_BTN_6, _T("Split Button"), 5);
pBtn6->SetMenu(IDR_RIBBON_MENU_1, TRUE);
pBtn6->SetAlwaysLargeImage();
pBtn6->RemoveSubItem(1);
pBtn6->AddSubItem(new CMFCRibbonButton(ID_RIBBON_MBTN_2, _T("Item 2"), 5), 1);
pPanel2->Add(pBtn6);

//创建新的面板，用于放置复选按钮
CMFCRibbonPanel* pPanel3 = pCategory->AddPanel(_T("Check Boxes"));

pPanel3->Add(new CMFCRibbonCheckBox(ID_RIBBON_BTN_7, _T("Check Box 1")));
pPanel3->Add(new CMFCRibbonCheckBox(ID_RIBBON_BTN_8, _T("Check Box 2")));
pPanel3->Add(new CMFCRibbonCheckBox(ID_RIBBON_BTN_9, _T("Check Box 3")));
```

代码 2.16 的运行结果如图 2.42 所示。

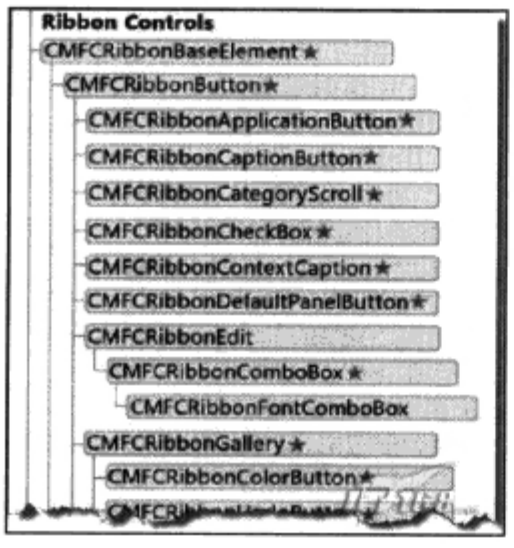


图 2.41　类视图

图 2.42　Ribbon 界面视图

（2）工具廊。

Ribbon 界面的一个重要革新，就是可以通过工具廊（CMFCRibbonGallery）控件，对命令的执行

效果进行直观的预览。例如，Word 2007 的段落格式设置，就是通过工具廊直观地展示了格式的外观，这很大程度上减少了用户通过不断尝试各种参数找到合适格式的过程，如图 2.43 所示。

下面就来看看如何创建工具廊这种新的交互工具。例 2.17 所示的代码演示了工具廊控件的创建过程，结果如图 2.43 所示。

例 2.17

```
CMFCRibbonPanel* pPanel1 = pCategory->AddPanel(_T("Standard"));

// 创建一个标准的工具廊控件，其中 IDB_RIBBON_PALETTE_1 指定了
// 工具廊中的各个按钮的图标，通过这些图标对命令效果进行预览
pPanel1->Add(new CMFCRibbonGallery(ID_RIBBON_PBTN_1, _T("Embedded"), 0, 0, IDB_RIBBON_PALETTE_1, 64));

// 按钮模式的工具廊控件
// 按钮模式的工具廊控件可以减少对面板空间的占用
CMFCRibbonGallery* pBtn2 = new CMFCRibbonGallery(ID_RIBBON_PBTN_2, _T("Button"), 1, 1, IDB_RIBBON_
PALETTE_1, 64);
// 设置面板按钮为按钮模式，默认情况下为画廊（Gallery）模式
pBtn2->SetButtonMode();
pBtn2->SetAlwaysLargeImage();
pPanel1->Add(pBtn2);


CMFCRibbonPanel* pPanel2 = pCategory->AddPanel(_T("Extended"));

// 对工具廊进行布局设置
CMFCRibbonGallery* pBtn3 = new CMFCRibbonGallery(ID_RIBBON_PBTN_3, _T("Resize Vertical"), 2, 2,
IDB_RIBBON_ PALETTE_1, 64);
pBtn3->SetButtonMode();

// 设置按钮模式下，下拉命令按钮容器（Gallery）的列数
pBtn3->SetIconsInRow(2);
pBtn3->EnableMenuResize(TRUE, TRUE);
pPanel2->Add(pBtn3);

CMFCRibbonGallery* pBtn4 = new CMFCRibbonGallery(ID_RIBBON_PBTN_4, _T("Resize Both"), 3, 3, IDB_
RIBBON_ PALETTE_1, 64);
pBtn4->SetButtonMode();
// 通过 SetIconInRow()和 EnableMenuResize()函数设置命令按钮的布局
pBtn4->SetIconsInRow04);
pBtn4->EnableMenuResize(TRUE);
pPanel2->Add(pBtn4);

CMFCRibbonGallery* pBtn5 = new CMFCRibbonGallery(ID_RIBBON_PBTN_5, _T("Groups && Subitems"), 4, 4);
// 通过 AddGroup()函数，对命令按钮进行分组
pBtn5->AddGroup(_T("Group 1"), IDB_RIBBON_PALETTE_1, 64);
pBtn5->AddGroup(_T("Group 2"), IDB_RIBBON_PALETTE_2, 64);
pBtn5->SetButtonMode();
```

```
pBtn5->SetIconsInRow(4);
pBtn5->EnableMenuResize(TRUE);

// 在按钮中添加子项目（按钮）
pBtn5->AddSubItem(new CMFCRibbonButton(ID_RIBBON_MENU_ITEM_1, _T("Item 1")));
pBtn5->AddSubItem(new CMFCRibbonButton(ID_RIBBON_MENU_ITEM_2, _T("Item 2")));
pBtn5->AddSubItem(new CMFCRibbonButton(ID_RIBBON_MENU_ITEM_3, _T("Item 3")));
pPanel2->Add(pBtn5);
```

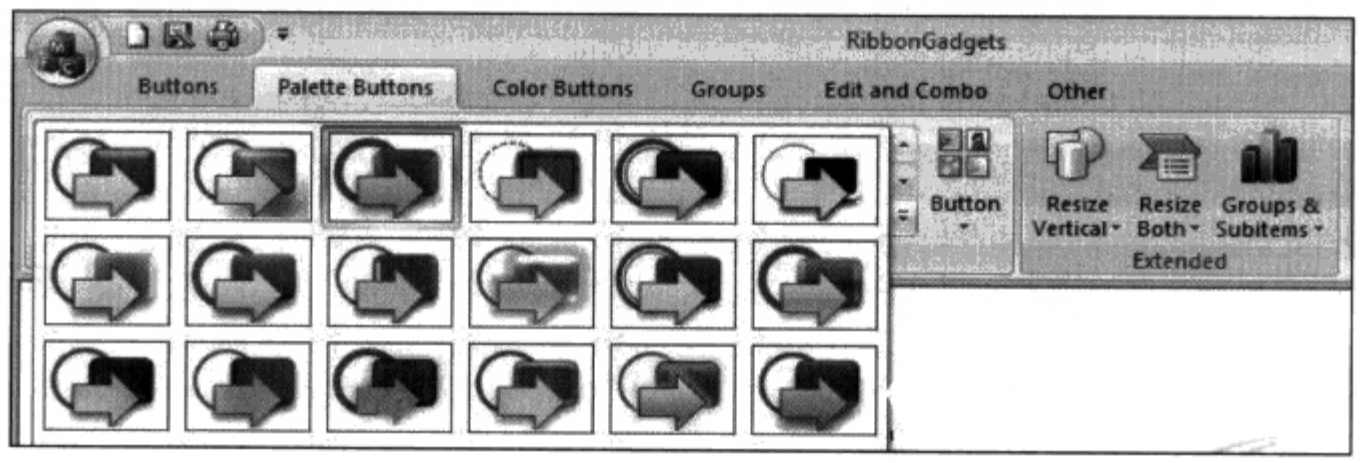


图 2.43　工具廊视图

（3）工具栏。

在传统的菜单式界面中，工具栏作为菜单的有益补充，被广泛使用。通过将一些常用命令放置到工具栏上，可以让用户直观而快速地访问到常用的功能，提高了效率。在 Ribbon 界面中，工具栏功能得到了进一步的加强，除了具备原来的工具栏功能外，因为使用命令按钮实现，还使得工具栏具备了下拉菜单等扩展功能。如图 2.44 所示。

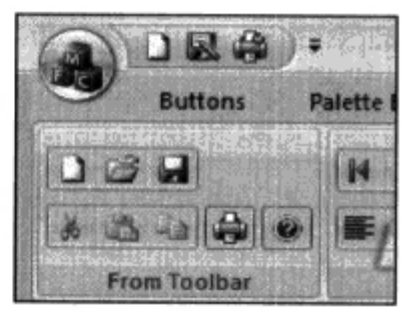


图 2.44　工具栏截图

例 2.18 所示的代码演示了如何创建 Ribbon 界面中的工具栏控件。

例 2.18

```
CMFCRibbonPanel* pPanel1 = pCategory->AddPanel(_T("From Toolbar"));

// 最简单的，通过 AddToolBar()函数，指定一个工具栏资源而创建工具栏
pPanel1->AddToolBar(IDR_TOOLBAR);

// 手动创建工具栏
CMFCRibbonPanel* pPanel2 = pCategory->AddPanel(_T("Manual"));

// 创建一个按钮组
CMFCRibbonButtonsGroup* pButtonsGroup1 = new CMFCRibbonButtonsGroup;

// 将新的按钮添加到按钮组中
pButtonsGroup1->AddButton(new CMFCRibbonButton(ID_RIBBON_GBTN_1, _T(""), 0));
pButtonsGroup1->AddButton(new CMFCRibbonButton(ID_RIBBON_GBTN_2, _T(""), 1));

// 创建一个编辑框控件
CMFCRibbonEdit* pEdit = new CMFCRibbonEdit(ID_RIBBON_GBTN_3, 65);
// 设置默认文本
pEdit->SetEditText(_T("Edit"));
```

```
pButtonsGroup1->AddButton(pEdit);

pButtonsGroup1->AddButton(new CMFCRibbonButton(ID_RIBBON_GBTN_4, _T(""), 2));
pButtonsGroup1->AddButton(new CMFCRibbonButton(ID_RIBBON_GBTN_5, _T(""), 3));

// 将按钮组添加到面板中
pPanel2->Add(pButtonsGroup1);

// 添加新的按钮组和按钮
CMFCRibbonButtonsGroup* pButtonsGroup2 = new CMFCRibbonButtonsGroup;
pButtonsGroup2->AddButton(new CMFCRibbonButton(ID_RIBBON_GBTN_6, _T(""), 4));
pButtonsGroup2->AddButton(new CMFCRibbonButton(ID_RIBBON_GBTN_7, _T(""), 5));
pButtonsGroup2->AddButton(new CMFCRibbonButton(ID_RIBBON_GBTN_8, _T(""), 6));
pButtonsGroup2->AddButton(new CMFCRibbonButton(ID_RIBBON_GBTN_9, _T(""), 7));

pPanel2->Add(pButtonsGroup2);

CMFCRibbonButtonsGroup* pButtonsGroup3 = new CMFCRibbonButtonsGroup;

CMFCRibbonButton* pBtn10 = new CMFCRibbonButton(ID_RIBBON_GBTN_10, _T(""), 8);
pBtn10->SetMenu(IDR_RIBBON_MENU_1);
pButtonsGroup3->AddButton(pBtn10);

CMFCRibbonButton* pBtn11 = new CMFCRibbonButton(ID_RIBBON_GBTN_11, _T(""), 9);
// 为按钮指定一个子菜单
pBtn11->SetMenu(IDR_RIBBON_MENU_1, TRUE);
pButtonsGroup3->AddButton(pBtn11);

pPanel2->Add(pButtonsGroup3);
```

（4）编辑框。

在传统的软件界面中，都是通过选择菜单命令或者单击工具栏中的按钮来简单执行某个命令，如图 2.45 所示。在新的 Ribbon 界面中，不仅可以单击按钮，还可以通过在编辑框中输入数据或者通过 ComboBox 快速地选择数据等，完成更加复杂的交互。

例 2.19 所示的代码演示了 Ribbon 编辑框的创建过程，结果如图 2.46 所示。

例 2.19

```
CMFCRibbonPanel* pPanel1 = pCategory->AddPanel(_T("Standard"));
// 创建简单编辑框
CMFCRibbonEdit* pBtn1 = new CMFCRibbonEdit(ID_RIBBON_EBTN_1, 90);
pBtn1->SetEditText(_T("Edit"));
pPanel1->Add(pBtn1);
// 创建可调编辑框
CMFCRibbonEdit* pBtn2 = new CMFCRibbonEdit(ID_RIBBON_EBTN_2, 90);
pBtn2->EnableSpinButtons(0, 99);
pBtn2->SetEditText(_T("0"));
pPanel1->Add(pBtn2);
// 创建 ComboBox
CMFCRibbonComboBox* pBtn3 = new CMFCRibbonComboBox(ID_RIBBON_EBTN_3, TRUE, 74);
```

```
    // 添加下拉选项
    pBtn3->AddItem(_T("Combo Box"));
    for (i = 0; i < 20; i++)
    {
        CString str;
        str.Format(_T("Item %d"), i + 1);
        pBtn3->AddItem(str);
    }
    pBtn3->SelectItem(0);
    pPanel1->Add(pBtn3);
    CMFCRibbonPanel* pPanel2 = pCategory->AddPanel(_T("With Icons and Labels"));
    // 创建带图标的编辑框
    CMFCRibbonEdit* pBtn4 = new CMFCRibbonEdit(ID_RIBBON_EBTN_4, 90, _T("Label:"), 0);
    pBtn4->SetEditText(_T("Edit"));
    pPanel2->Add(pBtn4);
    CMFCRibbonEdit* pBtn5 = new CMFCRibbonEdit(ID_RIBBON_EBTN_5, 90, _T("Label:"), 1);
    pBtn5->EnableSpinButtons(0, 99);
    pBtn5->SetEditText(_T("0"));
    pPanel2->Add(pBtn5);
    CMFCRibbonComboBox* pBtn6 = new CMFCRibbonComboBox(ID_RIBBON_EBTN_6, TRUE, 74, _T("Label:"), 2);
pBtn6->AddItem(_T("Combo Box"));
    for (i = 0; i < 20; i++)
    {
        CString str;
        str.Format(_T("Item %d"), i + 1);
        pBtn6->AddItem(str);
    }
        pBtn6->SelectItem(0);
    pPanel2->Add(pBtn6);
    CMFCRibbonPanel* pPanel3 = pCategory->AddPanel(_T("Font"));
    // 创建字体选择 ComboBox
    CMFCRibbonFontComboBox::m_bDrawUsingFont = TRUE;
    CMFCRibbonFontComboBox* pBtn7 = new CMFCRibbonFontComboBox(ID_RIBBON_EBTN_7);
    pBtn7->SelectItem(_T("Arial"));
    pPanel3->Add(pBtn7);
```

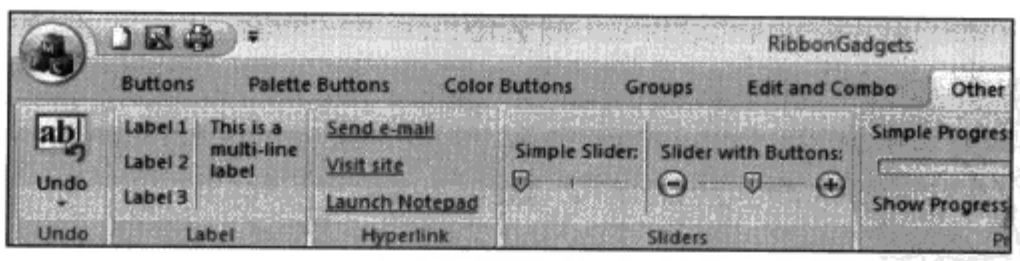


图 2.45　ribbon 视图

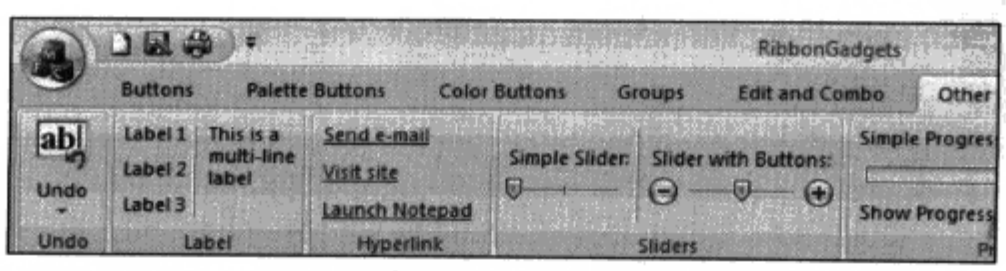


图 2.46　ribbon 视图

（5）其他控件。

除了前面介绍的按钮、工具栏、编辑框等基本控件外，为了支持现代软件对丰富的界面交互方式的要求，Visual Studio 2010 还提供了很多其他的辅助控件，例如通常会用到的“上一步”按钮、标签文本、超链接文本、滑动条、进度条等。这些辅助控件极大地丰富了 Ribbon 界面的表现力。

例 2.20 所示的代码演示了其他各种控件的创建过程。

例 2.20

```
// 添加“上一步”按钮
CMFCRibbonPanel* pPanel1 = pCategory->AddPanel(_T("Undo"));

CMFCRibbonUndoButton* pBtn1 = new CMFCRibbonUndoButton(ID_RIBBON_OBTN_1, _T("Undo"), 0, 0);
// 为返回按钮添加可以返回的动作
for (int i = 0; i < 10; i++)
{
        CString str;
    str.Format(_T("Action %d"), i + 1);
    pBtn1->AddUndoAction(str);
}
pPanel1->Add(pBtn1);

// 添加文本标签
CMFCRibbonPanel* pPanel2 = pCategory->AddPanel(_T("Label"));

pPanel2->Add(new CMFCRibbonLabel(_T("Label 1")));
pPanel2->Add(new CMFCRibbonLabel(_T("Label 2")));
pPanel2->Add(new CMFCRibbonLabel(_T("Label 3")));

// 在面板中添加一个分隔符
pPanel2->Add(new CMFCRibbonSeparator());

// 多行文本标签
pPanel2->Add(new CMFCRibbonLabel(_T("This is a multi-line label"), TRUE));

// 添加超链接标签
CMFCRibbonPanel* pPanel3 = pCategory->AddPanel(_T("Hyperlink"));

// 写 E-mail 给我啊
pPanel3->Add(new CMFCRibbonLinkCtrl(ID_RIBBON_OBTN_2, _T("Send e-mail"), _T("mailto:chenlq@live.com")));

// 欢迎访问我的 blog
pPanel3->Add(new CMFCRibbonLinkCtrl(ID_RIBBON_OBTN_3, _T("Visit site"), _T("http://space.itpub.net/
17237043/")));
pPanel3->Add(new CMFCRibbonLinkCtrl(ID_RIBBON_OBTN_4, _T("Launch Notepad"), _T("notepad")));

// 滑动条
CMFCRibbonPanel* pPanel4 = pCategory->AddPanel(_T("Sliders"));
pPanel4->SetCenterColumnVert();
```

```
pPanel4->Add(new CMFCRibbonLabel(_T("Simple Slider:")));
pPanel4->Add(new CMFCRibbonSlider(ID_RIBBON_OBTN_5, 70 /* Slider width */));

pPanel4->Add(new CMFCRibbonSeparator());

pPanel4->Add(new CMFCRibbonLabel(_T("Slider with Buttons:")));
CMFCRibbonSlider* pBtn6 = new CMFCRibbonSlider(ID_RIBBON_OBTN_6, 70 /* Slider width */);

// 设置缩放按钮
pBtn6->SetZoomButtons();
pBtn6->SetRange(0, 100);
pBtn6->SetPos(50);
pPanel4->Add(pBtn6);

// 进度条
CMFCRibbonPanel* pPanel5 = pCategory->AddPanel(_T("Progress Bars"));
pPanel5->SetCenterColumnVert();

// 简单进度条
pPanel5->Add(new CMFCRibbonLabel(_T("Simple Progress:")));
pPanel5->Add(new CMFCRibbonProgressBar(ID_RIBBON_OBTN_7, 100 /* Bar width */));
pPanel5->Add(new CMFCRibbonButton(ID_RIBBON_OBTN_8, _T("Show Progress 1")));

pPanel5->Add(new CMFCRibbonSeparator());

pPanel5->Add(new CMFCRibbonLabel(_T("Infinite Progress:")));
CMFCRibbonProgressBar* pBtn9 = new CMFCRibbonProgressBar(ID_RIBBON_OBTN_9, 100 /* Bar width */);

// 设置为无限模式，表示动作在进行，但是没有明确的进度
pBtn9->SetInfiniteMode();
pPanel5->Add(pBtn9);

pPanel5->Add(new CMFCRibbonButton(ID_RIBBON_OBTN_10, _T("Show Progress 2")));
```

至此，已经演练了 Ribbon 界面的大多数控件。通过这些实际代码的演练，相信读者对 MFC 的新功能——Office 2007 风格程序及 Visual Studio 风格程序的设计和实现也有了一定的了解。

2.7 MFC 新功能——Windows 资源管理器风格程序分析

许多 Windows 系统应用程序使用 Windows 资源管理器的用户界面（UI），例如，启动 Windows 资源管理器时，看到一个工作区由垂直拆分条分隔的应用程序。工作区的左侧提供导航和浏览功能，工作区的右侧显示与左窗格中的选定内容相关的详细信息。用户单击左窗格中的某项时，应用程序重新填充右窗格。在 MDI 应用程序中，可以使用“视图”菜单中的命令更改右窗格中显示的详细信息量。（在 SDI 或多顶级文档应用程序中，只能使用工具栏按钮更改详细信息。）

窗格的内容取决于应用程序。在文件系统浏览器中，左窗格显示目录或计算机（或计算机组）的分层视图，而右窗格显示文件夹、个别的文件或计算机及有关它们的详细信息。这些内容不一定必须是文件，它们可以是电子邮件、错误报告或是数据库中的其他项。向导为开发人员创建了以下类。

- CLeftView 类，定义工作区的左窗格，通常从CTreeView导出。
- CProjNameView 类，定义工作区的右窗格，默认情况下，它从CListView导出，但根据在向导生成的类页中的“基类”列表中指定的类，它可以是另一种视图类型。

生成的应用程序可以具有单文档界面（SDI）、多文档界面（MDI）或多顶级文档结构。应用程序创建的每个框架窗口都用CSplitterWnd垂直拆分。该应用程序的编码类似于使用拆分器的标准 MFC 应用程序的编码，不同的是，该应用程序类型的每个拆分器窗格中都有单独的控件视图。

如果在右窗格中使用默认的列表视图，向导将创建附加的菜单命令（仅在 MDI 应用程序中）和工具栏按钮，以在大图标、小图标、列表和详细信息模式间切换视图的样式。

（1）开始创建 Windows 资源管理器样式的 MFC 可执行文件。

（2）按照创建 MFC 应用程序的向导进行操作。

（3）在 MFC 应用程序向导的应用程序类型页中，选择“Windows 资源管理器”项目样式。

（4）在其他向导页中设置所需的任何其他选项。

（5）单击“完成”按钮生成主干应用程序。

例 2.21 和例 2.22 演示了各种控件的创建过程。

例 2.21

```
int CCustTreeView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
    return -1;

    // TODO: Add your specialized creation code here

    //创建风格
    DWORD dwStyle = WS_CHILD | WS_VISIBLE | WS_TABSTOP | TVS_HASLINES|TVS_HASBUTTONS;

    //此时不需要指定控件的真实大小，而在 OnSize()函数中指定
    BOOL bResult = m_tree.Create(dwStyle, CRect(0,0,0,0),
    this, IDC_TREE1);

    //展开根目录
    m_tree.Expand (m_tree.GetRootItem (), TVE_EXPAND);

    return (bResult ? 0 : -1);
}

void CCustTreeView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);

    // TODO: Add your message handler code here
    //当树形控件成功创建时，调整其大小为视图窗口的大小，这样控件就占据了整个视图窗口
    if (::IsWindow(m_tree.m_hWnd))
    {
        m_tree.MoveWindow(0, 0, cx, cy, TRUE);
    }
}
```

例 2.22

```
LRESULT CCustListView::OnChangeFolder(WPARAM,LPARAM)
{
    CString strPath;
    if (!m_list.GetCurrentFolder (strPath) &&
        !m_list.GetCurrentFolderName (strPath))
    {
        strPath = _T("????");
    }
    CMainFrame* pMainWnd=(CMainFrame*) AfxGetMainWnd ();

    //更新地址栏，记录新的地址，并且只记录最新的 6 个地址
    if (pMainWnd!= NULL)
    {
        //组合框
        pMainWnd->SetCurrFolder(strPath);
        AfxMessageBox("CCustListView::OnChangeFolder--strPath="+strPath);//debug
        //历史菜单
        CMenu* pSubMenu;
        //获取子菜单
        pSubMenu = pMainWnd->GetMenu()->GetSubMenu(2);
        //添加菜单命令
        int iCount=pSubMenu->GetMenuItemCount();
        pSubMenu->AppendMenu(MF_STRING | MF_ENABLED, 0xe00 + iCount,strPath);
    }

    return 0;
}

void CCustListView::OnInitialUpdate()
{
    CView::OnInitialUpdate();

    // TODO: Add your specialized code here and/or call the base class
    //更新主框架的地址栏和表单视图的内容
    OnChangeFolder (1, 1); }
```

2.8 应用程序框架类对象分析

在本章前几节中讨论了，可以在 MFC 多文档应用程序执行过程中，创建多于一个的文档类、视图类、子框架类对象和一个主框架类、应用类对象，这些对象是通过一定的方式联系在一起的。在应用程序设计中，时常需要通过这些对象之间的关系来实现文档、视图、框架类对象、应用类对象间的相互引用和调用。下面对这些对象间的调用方法进行简述。

2.8.1 视图类对文档类的调用

视图类对象的作用是在客户区显示文档类对象的内容，两者之间的调用函数最为常用。在视图类

对象中，通过 GetDocument()函数得到文档类对象的指针。在视图类 CDrawView 中可以找到一个 GetDocument()函数。

```
public:
    CDrawDoc* GetDocument();
```

在实现文件 drawview.cpp 中，有 GetDocument()函数的实现代码：

```
CDrawDoc* CDrawView::GetDocument()
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CDrawDoc)));
    Return (CDrawDoc*)m_pDocument;
}
```

GetDocument()函数的作用就是得到当前视图类对象所属的文档类对象的指针。在视图类 CDrawView 的成员函数中，可以通过以下代码调用 GetDocument()函数以得到当前文档类对象指针。

```
CDrawDoc* pDoc=GetDocument();
ASSERT_VALID(pDoc);
```

得到指向当前文档类对象的指针 pDoc 后，与使用普通 C++类的指针一样，可以调用文档类的成员变量和成员函数（只能调用公有型的）。

2.8.2　在框架类中获得当前的文档类和视图类对象指针

在主框架类 CMainFrame 和子框架类 CChildFrame 中，都可以利用基类的函数，得到指向当前活动的文档类对象和视图类对象的指针。

下面的代码可以获得指向当前活动的文档类对象和视图类对象的指针。

```
CDrawDoc* pDoc=(CDrawDoc *)GetActiveDocument();
CDrawView* pView=(CDrawView *)GetActiveView();
```

2.8.3　获得应用程序类对象的指针

在 MFC 应用程序执行过程中，自始至终存在一个应用类对象，可以在 MFC 派生类中或非 MFC 派生类中，通过全局 API 函数 AfxGetApp()得到应用类对象的指针。例如，可以在应用程序 Draw 中的任何位置，通过以下代码获得指向应用类对象的指针。

```
CDrawApp *pWinApp=(CDrawApp *)AfxGetApp();
```

2.8.4　从应用类对象中获得主框架类对象的指针

通过 MFC 应用程序的应用类对象，可以获得指向当前系统的主框架类对象的指针。在应用类 CWinApp 中有一个数据成员 m_pMainWnd，这个数据成员保存了指向应用程序主框架类对象的指针，通过它可以获得指向主框架类对象的指针。以下代码可以获得指向主框架类对象的指针。

```
CMainFrame* pFrame=(CMainFrame*)(AfxGetApp()->m_pMainWnd);
```

在视图类、文档类，以及非 MFC 派生类的成员函数（如全局函数）中，通过 API 函数 AfxGetApp()获得指向应用类对象的指针，再调用以上代码，得到指向主框架类对象的指针。

通过 API 函数 AfxGetApp()得到指向应用类对象的指针，再通过应用类对象得到指向主框架类对象的指针，通过主框架类对象又可以获得指向当前活动的文档类和视图类对象的指针。这种调用可以保证在各种情况下，都能获得指向 MFC 应用程序中各个主要框架类对象的指针，通过指针使用各框架类对象的成员（数据或函数）。

2.9 小结

本章主要介绍了 MFC 向导生成应用程序框架的过程，并对生成的程序和文件进行了分析。通过讲解 MFC 框架的运行机制，可以发现应用程序框架中的内容也有与 Win32 SDK 程序相应的操作，包括创建窗口，销毁窗口，设计窗口类，注册窗口类，创建、显示和更新窗口类等操作。完成这些基本操作之后，还应该通过消息循环及窗口处理过程来实现不断地接收消息、处理消息（只不过使用的是一个默认的窗口处理函数）。当然，MFC 最终的消息处理是利用消息映射来完成的，这将在后面的章节中介绍。

本章还介绍了文档类结构、视图类结构，以及两者之间的关系；另外，还介绍了工具栏与状态栏，以及这两种控件的创建、修改及其他操作；最后，还介绍了 MFC 新功能的 Office 2007 风格程序、Visual Studio 风格程序及 Windows 资源管理器程序等，总结分析了应用程序框架类对象的关系等。

通过本章的学习，读者应该：

（1）熟练使用 MFC 向导生成应用程序；

（2）知道使用 MFC 向导可以生成的应用程序类型；

（3）掌握 MFC 应用程序运行机制；

（4）掌握用 MFC 创建和销毁应用程序窗口和窗口类的过程；

（5）理解文档类和视图类的结构及二者之间的关系；

（6）掌握创建和修改工具栏和状态栏的基本原理和方法；

（7）熟练使用 MFC 创建各种不同风格的用户界面窗口的应用程序。

第 3 章　Visual C++ 2010 MFC 菜单编程

菜单、工具栏和状态栏是组成 Windows 图形界面的 3 个主要元素。其中菜单、工具栏提供了用户操作应用程序的命令界面；状态栏提供了一个输出区域，用来显示当前程序运行的状态和数据变化等。上一章已经详细地介绍了工具栏和状态栏及各自的相应操作，本章将详细介绍菜单的使用方法。

大多数的 Windows 应用程序都将菜单作为用户与应用程序之间传递命令的一个特殊的途径。使用菜单可以不必将大量的命令按钮摆放在窗口上，既节省空间又使用方便。本章将主要讲述菜单消息的传输机制、菜单的设计与创建、菜单编程，以及动态菜单的操作等。

3.1　菜单编程

首先介绍一下菜单编程中必不可少的工具——菜单编辑器。

使用菜单编辑器可以创建标准菜单和菜单命令，为菜单和菜单命令定义热键、加速键和状态栏提示，创建快捷菜单，等等。

菜单编辑器的功能有：

- 创建菜单及菜单命令；
- 创建菜单快捷键；
- 定义加速键及快捷菜单；
- 标记菜单；
- 在菜单中加入菜单；
- 启用及禁用菜单；
- 移除与加载菜单；
- 修改和控制系统菜单及用户自定义的菜单。

在向各位读者介绍完菜单编辑器及其功能后，下面了解一下菜单。

在众多的应用程序中，最常见的菜单有下拉式菜单、级联菜单和快捷菜单。

- 下拉式菜单：下拉式菜单一般显示在应用程序窗口的顶部，分类的类别排成一行，和某个类别相关的所有功能排在那个类别的下面，如果选中某个类别，其下就会弹出菜单，该菜单中有一系列具有相关功能的菜单命令可供用户选择。
- 级联菜单：级联菜单是下拉菜单的一个扩展。如果某个菜单命令的右侧有一个向右的黑三角符号，那么这个菜单命令其实就是一个级联菜单。级联菜单带有另一个子菜单，子菜单一般显示在所属菜单命令的右侧。这个子菜单和下拉式菜单相似，有一系列菜单命令可供用户选择。

要注意的是，请读者一定要区别弹出式菜单和菜单命令。一般来说，单击后会出现下拉子菜单的菜单称为弹出式菜单，而其他菜单叫菜单命令。级联菜单也是弹出式菜单，它右侧出现的子菜单中的各个小菜单也叫菜单命令。

- 快捷菜单：快捷菜单是可以通过在应用程序区域中右击调出的一种很方便的菜单，也可以称为弹出式菜单。由于该菜单的菜单命令依赖于被选中的对象或光标、鼠标光标在工作区域内所指

的位置，因此该种菜单也称为上下文菜单。

一般情况下，新建好一个 MFC 应用程序时，AppWizard 都会生成一个常用的菜单。单文档应用程序会自动生成标识符为 IDR_MAINFRAME 的菜单资源，而多文档应用程序会自动生成标识符为 IDR_MAINFRAME 和 IDR_xxxTYPE 的菜单资源，其中 xxx 为工程名。这些菜单资源在应用程序框架窗口创建时自动加载，开发人员可以修改这个菜单资源。

对一个单文档的工程来说，菜单是在 CxxxApp 的 Initinstance 中产生的（xxx 为工程名），产生代码如下：

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(CxxxDoc),
        RUNTIME_CLASS(CMainFrame),                    // main SDI frame window
        RUNTIME_CLASS(CxxxView));
AddDocTemplate(pDocTemplate);
```

其中 IDR_MAINFRAME 是菜单的 ID，在资源面板里可以看到，很多资源的 ID 都是 IDR_MAINFRAME，包括菜单、工具栏、加速键、图标和字符串表，由此可知，一个 ID 可以标识多个资源。

下面将详细介绍有关菜单编程的各种方法。

3.1.1 创建菜单

在应用程序中创建并使用一个菜单，一般都遵循下面几个步骤：

（1）在工程中添加一个自定义的菜单资源；

（2）用菜单编辑器设计菜单；

（3）将菜单加入应用程序；

（4）为每个菜单命令添加消息映射。

以上步骤是创建一个自定义菜单的常规步骤，读者也可以在菜单编辑器中或应用程序中修改、控制 AppWizard 生成的系统菜单及用户自定义菜单。

前面讲过如何添加一个字符串资源和位图资源，其实添加和修改菜单资源和添加前两个资源类似，所有的资源添加方法都是类似的。

1．添加一个菜单资源

首先，新建一个 MFC 单文档应用程序，命名为 Menu。

然后，编译运行这个新建的 Menu 程序，可以看到，对这个新建的程序而言，MFC 已经创建了一个默认的菜单，并且建立了一些基本的菜单功能。例如，选择“File”菜单下的“Open”命令时，弹出“打开文件”对话框；当选择“Save”命令时，弹出“另存为”对话框；而选择“Exit”命令时，退出当前应用。当用户选择某个菜单命令时，程序中就会调用相应菜单命令的命令响应函数，来完成这个菜单命令对应的功能。

在 Microsoft Visual Studio 2010 集成开发环境中，单击左侧窗格中的“Resource View”选项卡，并且依次选择“Menu”→“Menu.rc”→“Menu”选项，即可看到有一个名为 IDR_MAINFRAME 的菜单资源，它就是刚才在 Menu 应用程序界面中所看到的菜单，这是 MFC AppWizard 的 Menu 单文档应用程序自动创建的一个默认的主菜单。

双击这个菜单资源名称“IDR_MAINFRAME”，即可在 Visual Studio 2010 开发界面的右侧窗格中

打开菜单编辑器，如图 3.1 所示。

Visual Studio 2010 提供了一个所见即所得的资源编辑器。如果要为程序添加自己的菜单命令，可以在这个菜单中直接添加。程序运行时就能看到自己定义的菜单命令。例如，可以在“Help”菜单后添加一个主菜单命令，方法是单击图 3.1 中“Help”菜单后的虚线框中，则在虚线框中出现一个跳动的鼠标指针，说明此时可以输入此主菜单命令的名称了。把这个菜单命令的名称设为“Test”，并且在虚线框中输入。单击空白处后，在 IDR_MAINFRAME 菜单栏资源上，就能看到新添加的这个菜单命令，如图 3.2 所示。

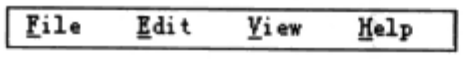


图 3.1　菜单编辑器

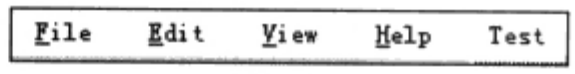


图 3.2　添加新菜单项结果

然后，右击“Test”菜单命令，并选择“Properties”命令，则在窗口的最右侧出现“Properties”面板，其中包括的属性有 Caption、Checked、Enabled、Grayed、Popup、Break、Right Justify、Right Order、Help、ID、Prompt、Separator。通过属性面板可以看到，对于刚才创建的“Test”菜单命令，其 ID 项、Prompt 和 Separator3 项属性都是灰色的。分别单击“File”、“Edit”、“View”、“Help”菜单，它们的这 3 项也都是灰色的。如果细心可以发现，当单击这些主菜单命令下的子菜单命令时，例如“File”主菜单命令下的“New”菜单项，会发现其 ID 项是可以输入和修改的。比较主菜单命令和子菜单命令的 Properties 选项，会发现，“File”、“Edit”、“View”、“Help”菜单这些主菜单命令的 Popup 选项的值都是 True，即选中状态；而子菜单命令“New”、“Open”、“Save”、“Exit”等子菜单命令的 popup 选项的值都是 False，即后者并未选中此选项。可以通过图 3.3 和图 3.4，比较 Popup 选项的值和 ID、Prompt、Separator 选项的状态。

MFC 中，把设置为 Popup 类型的菜单称为弹出式菜单，Visual Studio 2010 默认顶层菜单为弹出式菜单，这种菜单不能响应命令。那是不是顶层菜单只能是弹出式菜单呢？当然不是，只要将顶层菜单中的 Popup 选项的值由 True 改为 False，则该菜单就不再是弹出式菜单，而成为一个普通的菜单命令了。下面，把刚才新建的“Test”菜单的 Popup 选项设为 False，这时其 ID 项就能够被编辑了，可以输入一个 ID 号，例如 IDM_TEST，如图 3.5 所示。

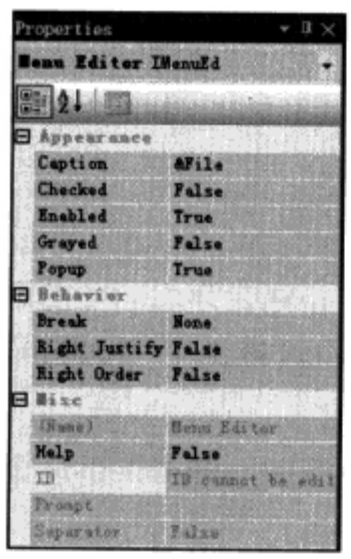


图 3.3　属性状态被禁用

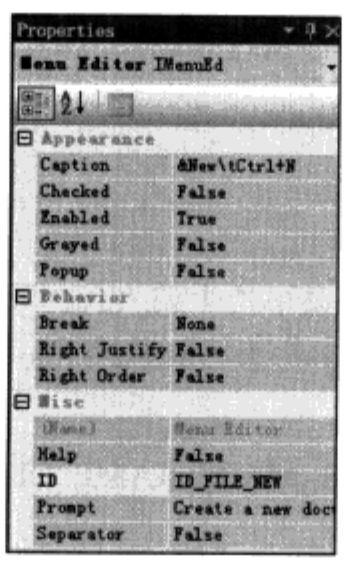


图 3.4　属性状态可用

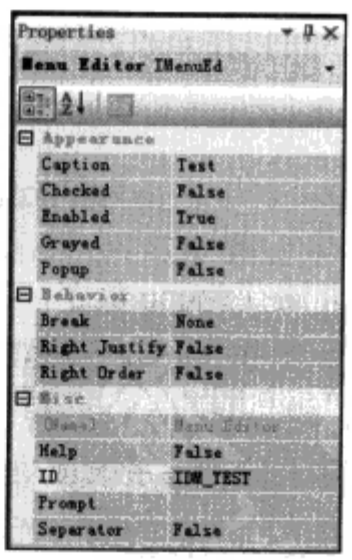


图 3.5　编辑菜单属性

一般在程序中会用到各种资源，在为资源确定 ID 名称时，为了明确区分资源类型，一般都遵循这样一个原则：在“ID”字符串后加上一个标识资源类型的字母。例如，在给 Test 菜单命令指定 ID 号时，就在“ID”字符串后加上一个字母“M”，这就表示它是 Menu 资源，即菜单资源。后面还会遇到光标（Cursor）资源，其 ID 号是在“ID”字符串后加上一个字母“C”；而图标资源（Icon）的 ID

号会在“ID”字符串后加上一个字母“I”。也就是说，菜单资源号以“IDM_”开始；光标资源 ID 号以“IDC_”开始；而图标资源 ID 号以“IDI_”开始。

下面为“Test”菜单添加命令响应，这里利用 Event Handler Wizard 工具来完成。

直接在“Test”这个菜单上右击，从弹出的快捷菜单中选择“Add Event Handler”命令，便可以打开如图 3.6 所示的“Event Handler Wizard”对话框。

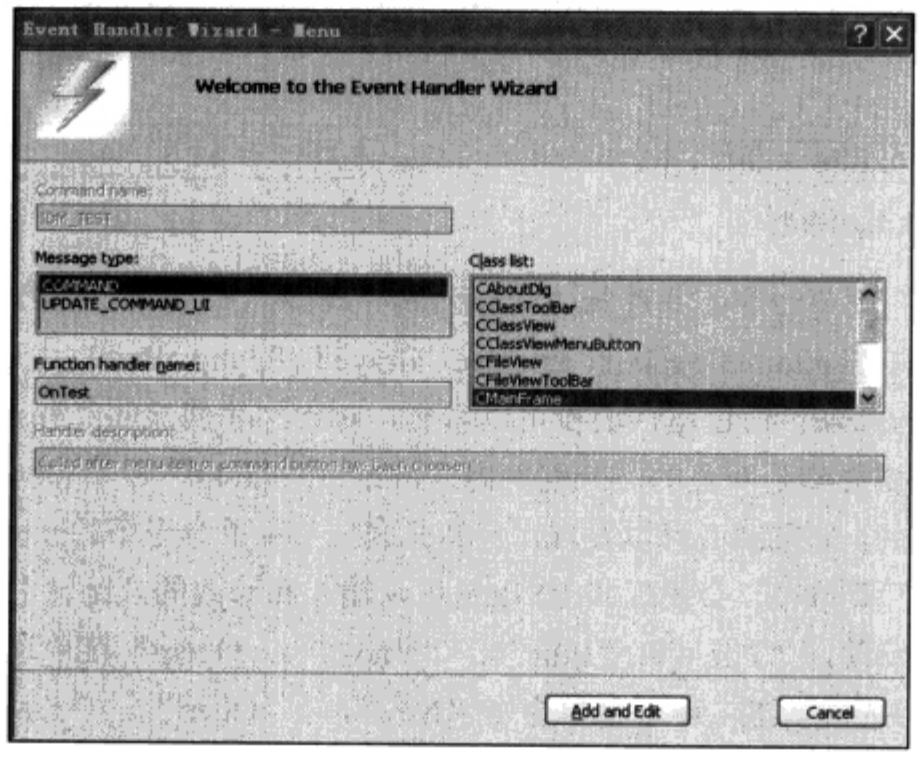


图 3.6　MFC 添加事件处理向导对话框

在框架窗口中添加响应这个菜单的命令。因此，在 MFC Event Handler Wizard 对话框中的对象 ID（Object IDs）列表框中选择“IDM_TEST”（默认），在类名称（Class list）列表框中选择“CMainFrame”选项，在消息（Message type）列表框中选择“COMMAND”项，然后在“Function handler name”列表框中增加一个响应函数，其中文本框中已经给出了“Test”菜单命令响应函数的默认名称“OnTest”，可以修改这个名称，本例保持该名称不变，单击“Add and Edit”按钮，即可打开 MainFrm.cpp 文件，并跳转到 OnTest()函数的定义处。

此时，可以在 OnTest()函数中添加下面这行代码，以弹出一个消息框表示该函数被调用了：

```
void CMainFrame::OnTest()
{
    MessageBox("¡ãMainFrame Clicked"¡À);// TODO: Add your command handler code here
}
```

编译并运行 Menu 程序，当单击程序菜单栏上的“Test”菜单命令时，就会弹出一个消息框，该消息框上显示了在 OnTest()函数中设置的字符串“MainFrame Clicked”，这就说明 OnTest()函数被调用了。

上面建立的是在系统自动生成的 IDR_MAINFRAME 的菜单资源中添加一个主菜单命令。下面将介绍如何建立一个新的菜单资源。

建立一个新的菜单资源，首先应该打开新建的 Menu 程序，然后从“View”菜单中选择“Resource View”菜单命令，这样在工作台右侧就会出现“Resource View”选项卡。选择工作台右侧的“Resource View”选项，使其显示资源面板页面。在资源浏览窗口中，右击 Menu，并且依次选择“Menu”→“Menu.rc”→“Menu”选项，然后右击“Menu”菜单，并且选择“Insert Menu”命令，会弹出一个快捷菜单，在该菜单中可以有两种选择来添加菜单资源，可以通过选择“Add Resource”命令来进行添加；也可以直接选择“Insert Menu”命令进行添加，这样可以省略调用“Insert Resource”对话框这一步，直接在工

作台“Menu”资源下产生一个新的菜单资源。

在执行上述操作之后，可以看到在工作台中多了一个 ID 号为 IDR_MENU1 的菜单资源。ID 号 IDR_MENU1 是系统自动生成的，可以保留这个 ID 号，也可以修改成自己想要的 ID 号。修改 ID 号的方法很简单，只要单击 ID 号 IDR_MENU1，就会在窗口右侧的 Menu Properties 选项卡中显示 IDR_MENU1 的属性值。如果想要修改 IDR_MENU1 这个菜单的 ID 号，在 Menu Properties 中的 ID 后的文本框中输入想要修改的 ID 值即可，如 IDR_MYMENU。

另外，MFC 都是采用大写字母来标识资源 ID 号的。为了与 MFC 保持一致，程序中也应该使用大写字母来标识 ID 号。

Menu Properties 面板如图 3.7 所示。

虽然添加了一个菜单资源，但它仅仅具有一个 ID 号，还没有任何内容，下面将使这个 ID 号具有真正的菜单。

2．利用菜单编辑器编辑菜单资源

双击工作台资源面板上 ID 号为 IDR_MYMENU 的新添加的菜单资源，在右侧的菜单编辑器中只有一个空白的菜单命令被高亮显示，如图 3.8 所示。

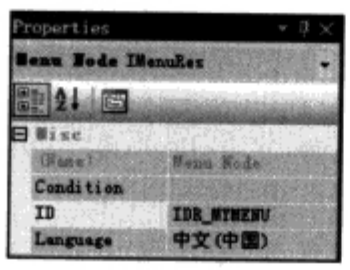


图 3.7　菜单项属性对话框

图 3.8　菜单编辑器

接下来，可以给这个菜单资源加入菜单命令了，假设要加一个弹出菜单“颜色”，它下面有两个菜单命令“红色”和“蓝色”，加入菜单命令的步骤如下。

（1）在菜单编辑器中，双击空白菜单命令，然后在文本框中会出现一个跳动的鼠标指针，说明可以在其中输入主菜单命令的名称，即 ID 号。在其中输入“颜色”，并按【Enter】键确认；或者用单击空白菜单命令，然后在窗体右侧出现的 Menu Properties 面板的“Caption”文本框中输入“颜色”，按【Enter】键确认，顶层菜单中就多了一个“颜色”下拉菜单。注意：在 Menu Properties 面板中，Popup 属性系统默认值为 True，这表示系统默认的顶层菜单为弹出式菜单，因此此时在这个弹出式菜单的下面又多了一个空白的菜单命令。

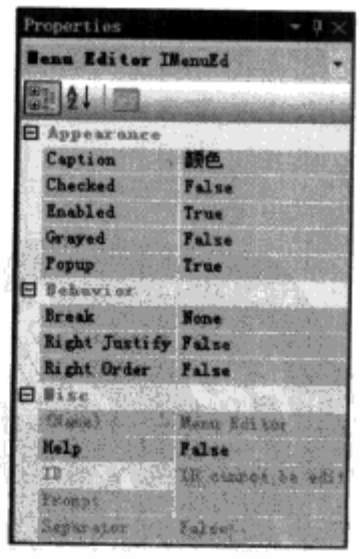


图 3.9　菜单项属性对话框

另外，应该注意的是，弹出式菜单的 ID 是不能够被编辑的。

Menu Properties 面板如图 3.9 所示。

在进行下一步操作之前，先来分析一下属性面板中的 5 组选项。

① ID（菜单命令的 ID）。菜单的 ID 后面有一个组合框，其中列出了所有使用的 ID 值，一般情况下，菜单的 ID 取值应遵循以下原则：

ID_顶层菜单名_下一级菜单名（_再下一级菜单名...）

这样做的优点是，当看到 ID 号时，就知道是哪个菜单命令了，并且当应用程序菜单中有多个菜单命令时，不容易造成 ID 号重复。需要注意的是，弹出式菜单不需要 ID 号，如“颜色”。

② Caption（菜单命令的名称）。菜单命令的 ID 代表其身份并且需要在程序中引用，将其显示在窗口上的时候，还需要有个名称。Caption 名既可以是英文，还可以是中文，还可以包含数字、符号等。

可以在 Caption 文本框中定义某个菜单命令的快捷键，给菜单加上快捷键的操作很简单，只需在 Caption 文本框中，在菜单标题中相应的字母前输入“&”即可。如果菜单是中文，就必须用小括号把快捷键的说明括起来。如图 3.10 所示的“New（新建）”菜单命令的快捷键使用“（&N）”定义。

Caption 文本框中还有一串字符“\tCtrl+N”，这是用来表明该菜单项的加速键，加速键可以在不打开菜单的情况下直接执行菜单命令。其中“\t”是制表符。

③ 一组复选框。

- “Separator”（分隔线）复选框：选中该复选框，则菜单命令变成一个分隔线，而原有菜单命令信息丢失。如果用户想在已添加的两个菜单命令中间加一条分隔线，可以先用鼠标拖动空白菜单到需要分隔线的位置，然后将这个空白菜单的“Separator”复选框选中。
- “Popup”（弹出式）复选框：选中此复选框，菜单命令将成为弹出式菜单，即单击此菜单会弹出下一级子菜单。弹出式菜单不需要指定 ID 号，Visual Studio 2010 默认顶层菜单为弹出式菜单，当然，也可以把它设为菜单命令。非顶层菜单也可以作为弹出式菜单（级联菜单），它的菜单命令的右侧显示一个三角符号。如果不选中这个复选框，表示该菜单为菜单命令。

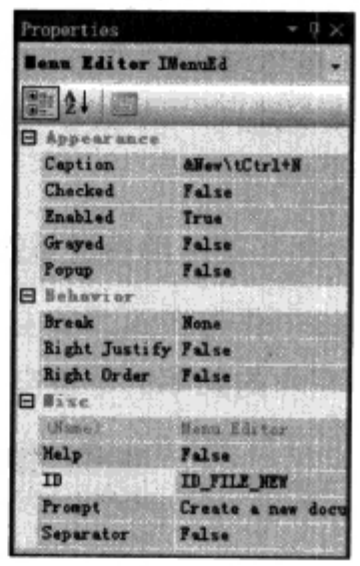


图 3.10 “New（新建）”菜单命令属性

- “Inactive”（不活动）复选框：选中此复选框表明菜单命令的初始状态为非活动状态。
- “Checked”（检查标记）复选框：此复选框用来给菜单一个初始的检查标记，选中该复选框，会在菜单命令的左侧打上一个钩。
- “Grayed”（变灰）复选框：选中此复选框，该菜单命令将以灰色显示，表明菜单的初始状态为不可访问。如果该复选框被选中，那么菜单一定是不活动的，所以此时“Inactive”复选框不可用。

④ Help（帮助形式）。如果把“Help”的值设置为 True，菜单会显示在菜单栏的右侧，如图 3.11 所示，将 AppWizard 生成的标准菜单中的“帮助”菜单选定为 Help 形式。

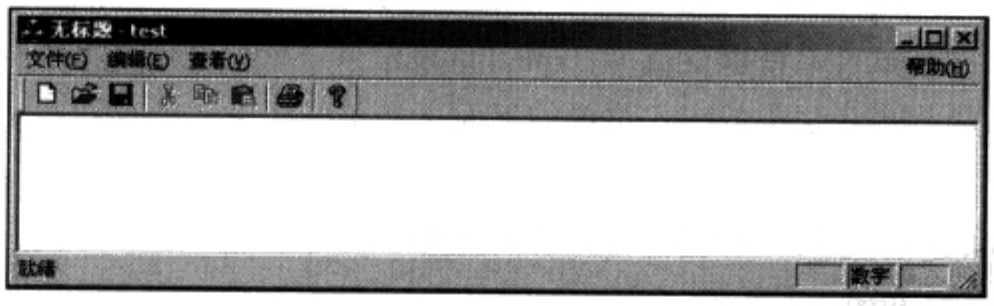


图 3.11 帮助形式菜单

⑤ Break（菜单命令的分隔）。

当一个菜单的菜单命令太多时，放在一列中会感觉很长，很拥挤，这时可以考虑把它分成两列来显示，Break 就起到了这样的作用。Break 选项对应一个下拉列表框，有 3 种选择方式：None、Column 和 Bar。

- None ： 系统默认选项，表示不分隔。
- Column: 列分隔，选中此项时，选中的菜单命令以下的菜单命令将另起一列。如图 3.12 所示为将“文件”菜单中的“打印”菜单命令选中此项的结果。
- Bar: 和 Column 类似，也是列分隔，只不过，与 Column 不同的是，两列之间有一条竖线将它

们隔开，如图 3.13 所示。

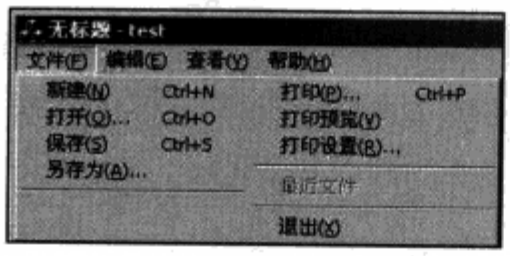


图 3.12　Column 分隔形式

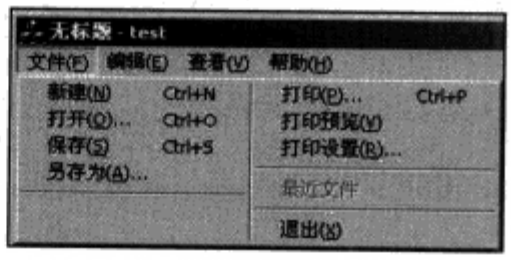


图 3.13　Bar 分隔形式

⑥ Prompt（菜单命令功能的提示）。在 Prompt 后面的文本框中可以输入文本，程序运行时，当鼠标指针移到菜单命令上时，在底部的状态栏将给出此菜单的功能提示。

在介绍完属性面板中的选项后，继续介绍如何在新建的主菜单命令下添加一个菜单命令。

（2）如图 3.14 所示，双击顶层菜单“颜色”下的空白的菜单命令，则会在窗口的最右侧再次弹出属性面板，在 Caption 后面的文本框中输入“红色”，或者直接在空白的菜单命令文本框中跳动的鼠标指针位置直接输入，然后设定其 ID 号为 ID_COLOR_RED，最终结果如图 3.14 所示。按【Enter】键确认，“红色”菜单命令就添加好了。

（3）再双击“红色”菜单命令下面的空白菜单命令，用相同的方法添加“蓝色”菜单命令，在 Caption 后面的文本框中输入“蓝色”，设定其 ID 号为 ID_COLOR_BLUE，结果如图 3.15 所示，按【Enter】键之后，这个菜单命令就添加好了。

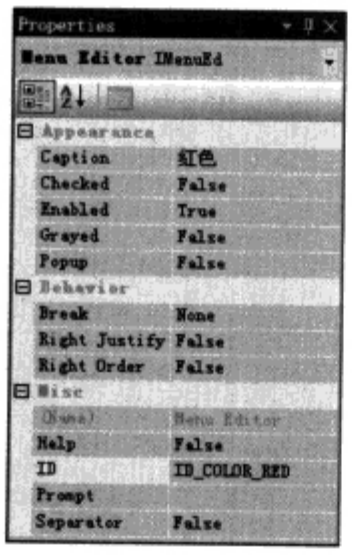


图 3.14　设置“红色”菜单项属性

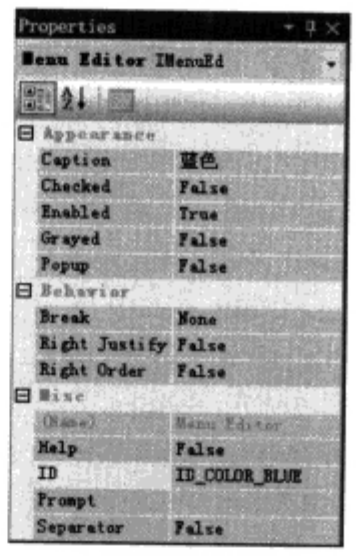


图 3.15　设置“蓝色”菜单项属性

经过上述 3 个步骤，带有“红色”和“蓝色”子菜单命令的弹出式菜单“颜色”菜单就创建好了，下面将讲述如何将它加入应用程序。

3．将菜单加入应用程序中

现在运行一下刚刚添加完“颜色”菜单的程序，发现窗口上显示的菜单依然是 AppWizard 添加的标准菜单，而刚刚创建的“颜色”菜单并没有显示出来，为什么呢？

一个应用程序只能有一个菜单，在前面已经讲过，单文档应用程序会自动生成标识符为 IDR_MAINFRAME 的菜单资源，因此在程序加载的时候，它只是加载了标识符为 IDR_MAINFRAME 的菜单，要想让开发人员添加的菜单显示出来，可以在框架类创建的时候，将自己的菜单加载进去。在 CmainFrame 类的 OnCreate()函数中编辑代码，如例 3.1 所示。

例 3.1

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
```

```
        if (CFrameWndEx::OnCreate(lpCreateStruct) == -1)
            return -1;

        BOOL bNameValid;
        // set the visual manager and style based on persisted value
        OnApplicationLook(theApp.m_nAppLook);

        if (!m_wndMenuBar.Create(this))
        {
            TRACE0("Failed to create menubar\n");
            return -1;          // fail to create
        }

        m_wndMenuBar.SetPaneStyle(m_wndMenuBar.GetPaneStyle() | CBRS_SIZE_DYNAMIC | CBRS_TOOLTIPS |
CBRS_FLYBY);

        // prevent the menu bar from taking the focus on activation
        CMFCPopupMenu::SetForceMenuFocus(FALSE);

        if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE | CBRS_TOP | CBRS_GRIPPER |
CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) || !m_wndToolBar.LoadToolBar(theApp.m_bHiColorIcons ?
IDR_MAINFRAME_256 : IDR_MAINFRAME))
        {
            TRACE0("Failed to create toolbar\n");
            return -1;          // fail to create
        }

        CString strToolBarName;
        bNameValid = strToolBarName.LoadString(IDS_TOOLBAR_STANDARD);
        ASSERT(bNameValid);
        m_wndToolBar.SetWindowText(strToolBarName);

        CString strCustomize;
        bNameValid = strCustomize.LoadString(IDS_TOOLBAR_CUSTOMIZE);
        ASSERT(bNameValid);
        m_wndToolBar.EnableCustomizeButton(TRUE, ID_VIEW_CUSTOMIZE, strCustomize);

        // Allow user-defined toolbars operations:
        InitUserToolbars(NULL, uiFirstUserToolBarId, uiLastUserToolBarId);

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

        // TODO: Delete these five lines if you don't want the toolbar and menubar to be dockable
        m_wndMenuBar.EnableDocking(CBRS_ALIGN_ANY);
```

```
        m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
        EnableDocking(CBRS_ALIGN_ANY);
        DockPane(&m_wndMenuBar);
        DockPane(&m_wndToolBar);


        // enable Visual Studio 2005 style docking window behavior
        CDockingManager::SetDockingMode(DT_SMART);
        // enable Visual Studio 2005 style docking window auto-hide behavior
        EnableAutoHidePanes(CBRS_ALIGN_ANY);

        // Load menu item image (not placed on any standard toolbars):
        CMFCToolBar::AddToolBarForImageCollection(IDR_MENU_IMAGES, theApp.m_bHiColorIcons ? IDB_MENU_
IMAGES_24 : 0);

        // create docking windows
        if (!CreateDockingWindows())
        {
            TRACE0("Failed to create docking windows\n");
            return -1;
        }

        m_wndFileView.EnableDocking(CBRS_ALIGN_ANY);
        m_wndClassView.EnableDocking(CBRS_ALIGN_ANY);
        DockPane(&m_wndFileView);
        CDockablePane* pTabbedBar = NULL;
        m_wndClassView.AttachToTabWnd(&m_wndFileView, DM_SHOW, TRUE, &pTabbedBar);
        m_wndOutput.EnableDocking(CBRS_ALIGN_ANY);
        DockPane(&m_wndOutput);
        m_wndProperties.EnableDocking(CBRS_ALIGN_ANY);
        DockPane(&m_wndProperties);

        // Enable toolbar and docking window menu replacement
        EnablePaneMenu(TRUE, ID_VIEW_CUSTOMIZE, strCustomize, ID_VIEW_TOOLBAR);

        // enable quick (Alt+drag) toolbar customization
        CMFCToolBar::EnableQuickCustomization();

        if (CMFCToolBar::GetUserImages() == NULL)
        {
            // load user-defined toolbar images
            if (m_UserImages.Load(_T(".\\UserImages.bmp")))
            {
                m_UserImages.SetImageSize(CSize(16, 16), FALSE);
                CMFCToolBar::SetUserImages(&m_UserImages);
            }
        }

        // enable menu personalization (most-recently used commands)
```

```
    // TODO: define your own basic commands, ensuring that each pulldown menu has at least one basic command.
    CList<UINT, UINT> lstBasicCommands;

    lstBasicCommands.AddTail(ID_FILE_NEW);
    lstBasicCommands.AddTail(ID_FILE_OPEN);
    lstBasicCommands.AddTail(ID_FILE_SAVE);
    lstBasicCommands.AddTail(ID_FILE_PRINT);
    lstBasicCommands.AddTail(ID_APP_EXIT);
    lstBasicCommands.AddTail(ID_EDIT_CUT);
    lstBasicCommands.AddTail(ID_EDIT_PASTE);
    lstBasicCommands.AddTail(ID_EDIT_UNDO);
    lstBasicCommands.AddTail(ID_APP_ABOUT);
    lstBasicCommands.AddTail(ID_VIEW_STATUS_BAR);
    lstBasicCommands.AddTail(ID_VIEW_TOOLBAR);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_OFF_2003);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_VS_2005);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_OFF_2007_BLUE);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_OFF_2007_SILVER);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_OFF_2007_BLACK);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_OFF_2007_AQUA);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_WIN7_SCENIC);
    lstBasicCommands.AddTail(ID_SORTING_SORTALPHABETIC);
    lstBasicCommands.AddTail(ID_SORTING_SORTBYTYPE);
    lstBasicCommands.AddTail(ID_SORTING_SORTBYACCESS);
    lstBasicCommands.AddTail(ID_SORTING_GROUPBYTYPE);

    CMFCToolBar::SetBasicCommands(lstBasicCommands);

    CMenu menu;
    menu.LoadMenu(IDR_MYMENU);
    this->SetMenu(&menu);

    return 0;
}
```

程序中前面的代码所实现的功能是创建并显示工具栏和状态栏，这部分代码是由应用程序自动生成的。

后面加粗部分的代码首先定义了一个 CMenu 类的变量，然后调用该类的成员函数 LoadMenu()将新添加的 ID 号为 IDR_MYMENU 的菜单载入该变量，即让变量 menu 指向新创建的菜单，最后的语句 **this->SetMenu(&menu);**用 SetMenu()函数替换现有菜单。函数 SetMenu()原型如下：

```
BOOL SetMenu( CMenu* pMenu );
```

参数 pMenu 为菜单指针，若该参数为空，则表示删除现有菜单；否则用参数所指的菜单来替换现有菜单。

其实，可以不用这么麻烦，完全可以通过修改原有的标准菜单来添加一个菜单。可以将原来的标准菜单添加一个或删掉一个弹出式菜单，也可以将某个弹出式菜单的某个菜单命令删除，或给它添加一个菜单命令，这些在菜单编辑器中都可以实现，其操作过程与前面所述也类似，在后面的实例中也会讲到。

运行结果如图 3.16 所示，但是，这个菜单还不能够进行任何操作，它的两个菜单命令都呈灰色状

态，这是因为还没有给它们添加消息响应函数。

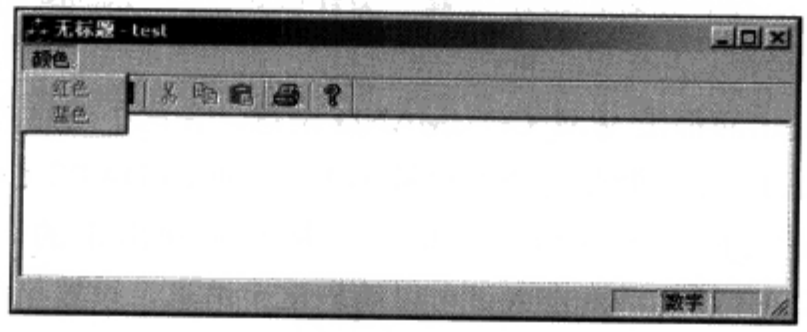


图 3.16　添加菜单项运行结果

4．给菜单命令添加 COMMAND 消息处理

现在，给“红色”和“蓝色”两个菜单命令添加消息响应。给菜单添加消息响应的过程如下所示。

（1）右击“红色”菜单命令，在弹出的快捷菜单中选择“Add Event Handler”命令，弹出“Event Handler Wizard”对话框。

（2）在“Class name”列表框中选择“CmainFrame”类，此时，“Message type”列表框中只有两个消息：COMMAND（命令消息）和 UPDATE_COMMAND_UI（更新命令用户界面消息）。如图 3.17 所示。

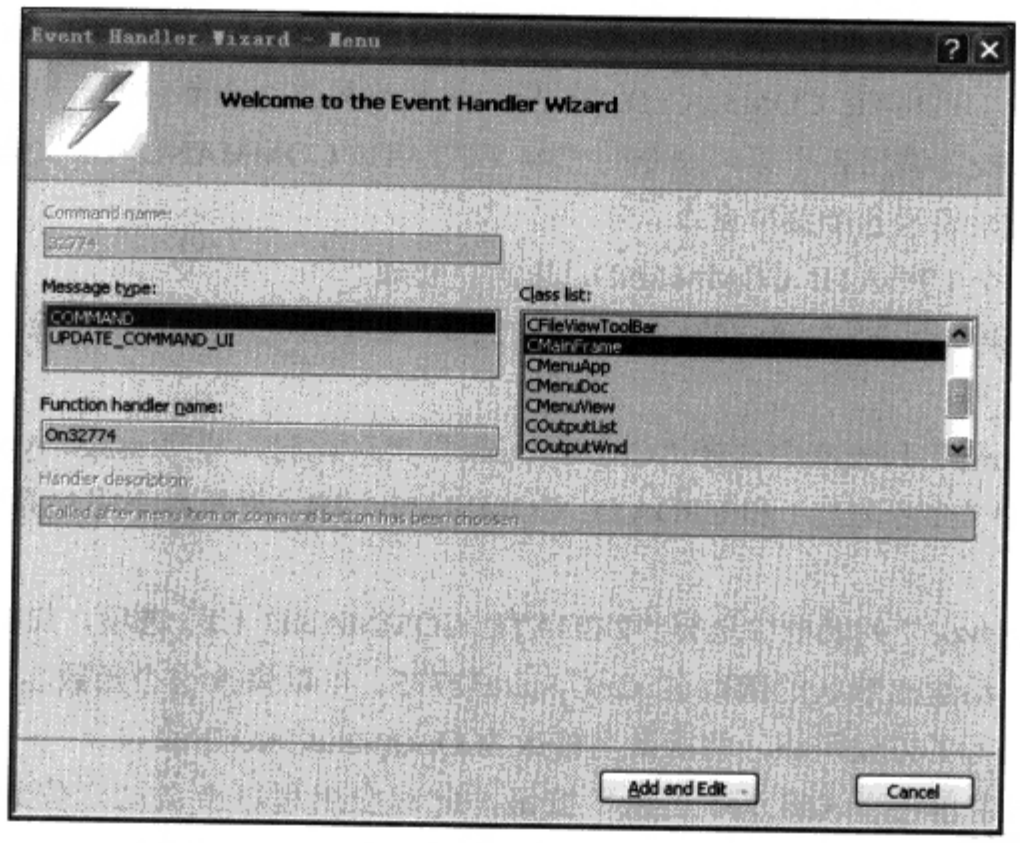


图 3.17　添加菜单项事件处理向导对话框

（3）在“Message type”列表框中选择“COMMAND”选项，如图 3.17 所示，在该对话框中，应用程序自动给出默认的函数名 On32774，开发人员可以修改这个函数名，但是作者建议使用默认的函数名，这里为了方便辨识，将函数名修改为 OnColorRed。

在单文档工程中，MFC AppWizard 生成的 4 个类都可以响应同一个菜单的 WM_COMMAND 消息。如果在这 4 个类里都加了同一个菜单的响应函数，4 个响应函数将只有一个被执行，它们之间有一个优先顺序：View→Document→MainFrame→App。具体响应函数应该放在哪个类中，就要看在这个响应函数中，实现的是什么功能，放在哪个类中方便。比如，如果单击一个菜单，想隐藏工具栏，它的响应函数就应放在 CmainFrame 中。

（4）在该对话框中“Add and Edit”按钮，此时应用程序会自动定位到 CMainFrame 类的 OnColorRed()

函数，就可以在这个函数中编写代码了。

在上面的第（3）步中，有的读者可能不理解，为什么添加 COMMAND 命令而不添加它下面的 UPDATE_COMMAND_UI 命令。

当单击菜单时，系统发出的都是 WM_COMMAND 消息，在这个消息的扩展参数 wParam 中，包含菜单的 ID，用户可以通过 ID 来判断是哪个菜单被单击了。而 UPDATE_COMMAND_UI 消息则是当菜单需要重新绘制或更新时由系统发送。菜单是一格一格弹出来的，每弹出一格，发出一个 ON_UPDATE_COMMAND_UI 消息，开发人员可以响应这个消息，设置菜单弹出的状态，比如打勾、打点、变灰等。注意：ON_UPDATE_COMMAND_UI 只能由拥有菜单的窗口发出！

5. 给菜单命令添加 UPDATE_COMMAND_UI 消息处理

在讲述如何添加 UPDATE_COMMAND_UI 消息处理之前，有必要先了解一下用户界面的更新原理。

（1）用户界面更新原理。

更新命令 UI 消息只适用于弹出式菜单的菜单命令，而对顶层菜单不起作用。在显示弹出式菜单时，框架按视图类→文档类→框架类→应用程序类的顺序发送弹出式菜单的所有菜单命令的 UPDATE_COMMAND_UI 消息。如果菜单命令有一个更新处理函数，则它被调用进行更新；否则框架检查 COMMAND 命令处理函数是否存在，如果不存在，则使菜单项变灰。因此，为了使一个菜单命令有效，必须为该菜单命令添加一个消息处理函数。

当用户展开一个菜单时，每个菜单命令都需要知道它应该显示为何种状态（如允许、禁止）。菜单命令就是通过 ON_UPDATE_COMMAND_UI 处理函数来提供这个消息的。当某个菜单被展开时，应用程序框架就会查找并调用各个菜单命令的 ON_UPDATE_COMMAND_UI 处理函数，然后根据处理函数的内容来正确地显示各个菜单命令。

（2）如何添加一个 UPDATE_COMMAND_UI 消息处理。

① 右击“红色”菜单命令，在弹出的快捷菜单中选择“Add Event Handler”命令，弹出“Event Handler Wizard”对话框。

② 在“Class name”列表框中选择“CMainFrame”类，此时，“Message type”列表框中只有两个消息：COMMAND（命令消息）和 UPDATE_COMMAND_UI（更新命令用户界面消息）。如图 3.17 所示。

③ 在“Message type”列表框中选择“UPDATE_COMMAND_UI”选项，如图 3.17 所示，在该对话框中，应用程序自动给出默认的函数名 OnUpdate32775，可以修改这个函数名，作者建议还是使用默认的函数名，这里为了方便辨识，将函数名修改为 OnUpdateColorBlue。

④ 在该对话框中单击“Add and Edit”按钮，此时应用程序会自动定位到 CMainFrame 类的 OnUpdateColorBlue()函数，就可以在这个函数中编写代码了。函数体如下：

```
void CMainFrame::OnUpdateColorBlue(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI-> SetCheck(TRUE);
}
```

可以看到，在调用 ON_UPDATE_COMMAND_UI 的响应函数时，系统为用户传来了 pCmdUI 指针。CCmdUI 是一个只被使用在 ON_UPDATE_COMMAND_UI 消息的响应函数中的类，CCmdUI 类没有基类，是一个独立的类，它可以控制正在被弹出的菜单命令的状态。这个类中的成员变量 m_nIndex 表示菜单命令在整个弹出菜单中的序号，另一个成员变量 m_nID 表示菜单的 ID。下面介绍一下其成员函数：

```
Enable():设置菜单命令是否有效；例 pCmdUI-> Enable(TRUE);
SetCheck():设置菜单命令是否打勾；例 pCmdUI-> SetCheck(TRUE);
SetRadio():设置菜单命令是否打点；例 pCmdUI-> SetRadio(TRUE);
SetText():设置菜单命令的文本；例 pCmdUI-> SetText (“蓝色”);
```

详细的成员函数说明可参照 MSDN。

3.1.2　创建菜单热键

打开菜单编辑器，在菜单的属性面板上，在“Caption”文本框中除了输入菜单命令的名称，还可以定义创建某个菜单命令的快捷键，即热键。给菜单加上快捷键的操作很简单，只需在“Caption”文本框中，在菜单标题中相应的字母前输入“&”即可。如果菜单是中文，就必须用小括号把快捷键的说明括起来。如图 3.18 所示的“新建”菜单命令的快捷键使用“(&N)”定义。

“Caption”文本框中还有一串字符“\tCtrl+N”，这是用来表明该菜单命令的加速键，加速键可以在不打开菜单的情况下直接执行菜单命令，其中“\t”是制表符。

3.1.3　标记菜单

新建一个名为 Menu 的 MFC Application 应用程序，运行前面新创建的 Menu 程序后，可以看到程序的“View”菜单下的菜单命令“Status Bar”前面带有一个对勾，如图 3.19 所示，称这种类型的菜单为标记菜单。

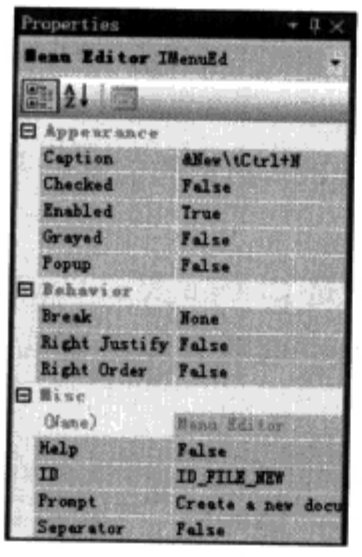


图 3.18　设置菜单项快捷键

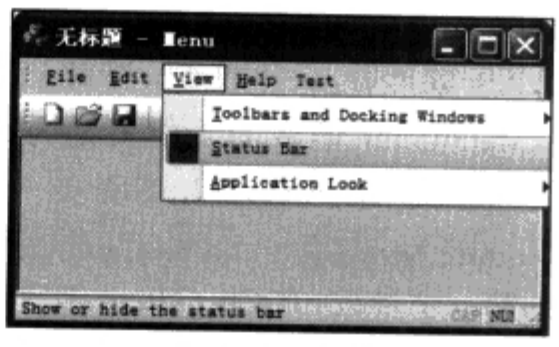


图 3.19　标记菜单

下面来实现这种标记菜单的功能。

（1）在“File”→“New”菜单命令上添加一个标记。

因为程序的主菜单属于框架窗口，所以需要在框架类窗口创建完成之后再去访问菜单对象。可以在框架类（CMainFrame 类）的 OnCreate()函数的最后（但一定要在 return 语句之前）添加实现这个功能的代码。

（2）为了获得“File”→“New”菜单命令，首先要获得程序的菜单栏，也就是要在框架窗口中获得指向菜单栏的指针。

这可以通过 CWnd 的成员函数 GetMenu()来实现，该函数具有如下形式的声明：

```
CMenu* GetMenu() const;
```

从其声明中可以知道，该函数返回一个指向 CMenu 类对象的指针。CMenu 类是一个 MFC 类，是 Windows 菜单 HMENU 的一个封装，提供了一些与菜单操作有关的成员函数，例如菜单的创建、更新

和销毁等，还可以获取一个菜单的子菜单，这是通过 GetSubMenu()成员函数实现的。该函数声明形式如下：

```
CMenu* GetSubMenu( int nPos) const;
```

通过声明可以知道，GetSubMenu()函数有一个参数，是整型的 nPos，该函数指定了子菜单的索引号。另外，GetSubMenu()函数也返回一个指向 CMenu 对象的指针，但是，这个函数的返回值所指向的对象与上面的 CWnd 类的 GetMenu()函数返回值所指向的对象是不同的。GetMenu()函数返回的是指向程序菜单栏对象的指针，而 CMenu 类的 GetSubMenu()成员函数返回的是由参数 nPos 指定的子菜单的指针。另一个重要的区别就是，二者所属的类是不同的，GetMenu()函数是 CWnd 类的成员函数，而 GetSubMenu()函数则是 CMenu 类的成员函数。

其实，系统菜单是有鼠标单击应用程序图标弹出的菜单。单击系统菜单时，发出 WM__SYSCOMMAND 消息，可添加此消息的响应函数来进行控制。下面是 GetMenu()、GetSubMenu()和 GetSystemMenu()3 个函数的具体声明形式及作用。

- GetMenu()函数。

```
CMenu* GetMenu( ) const;
```

返回窗口菜单指针，若无菜单，则返回 NULL。

- GetSystemMenu()函数。

```
CMenu* GetSystemMenu( BOOL bRevert ) const;
```

返回窗口系统菜单指针，可以用这个指针来修改系统菜单。

参数 bRevert 若为 False，则可得到并修改系统菜单；若为 True，则系统菜单恢复为默认状态。

- GetSubMenu()函数。

```
CMenu* GetSubMenu( int nPos ) const;
```

返回窗口菜单中的某个弹出式菜单。

参数 nPos 指定要取得的那个弹出式菜单在菜单中的位置，如果 nPos 指定的位置不存在，那么返回 NULL。

（3）在找到子菜单后，为了设置一个标记菜单，另外需要使用的就是 CMenu 类的 CheckMenuItem()函数。CheckMenuItem()函数的功能就是为菜单命令添加一个标记，或者移除菜单命令的标记。CheckMenuItem()函数的声明形式如下所示：

```
UINT CheckMenuItem( UINT nIDCheckItem, UINT nCheck );
```

该函数的第一个参数 nIDCheckItem 指定需要处理的菜单命令，它的值由第二个参数决定。第二个参数 nCheck 指定应该怎样设置菜单命令，以及如何定位相应菜单命令的位置，其值可以是 MF_CHECKED 或 MF_UNCHECKED 与 MF_BYPOSITION 或 MF_BYCOMMAND 的组合。这些标记的含义分别如下所述。

- MF_CHECKED：设置菜单命令的复选标记。
- MF_UNCHECKED：移走菜单命令的复选标记。
- MF_BYPOSITION：表明应该根据菜单命令的位置来访问菜单命令，即第一个参数指定的是菜单命令的索引号。
- MF_BYCOMMAND：表明应该根据菜单命令的命令来访问菜单命令，即第一个参数指定的是菜单命令的命令 ID。

此外，在 Menu 程序中，“File”子菜单及其“New”菜单命令的索引号均为 0。

下面可以开始实现为“File”子菜单中的“New”菜单命令添加标记这一功能了。例 3.2 所示的代码中加粗显示的那行代码就是具体的实现代码，只需要在 CMainFrame 类的 OnCreate()函数的 return 语句之前添加这条语句即可实现标记功能。

例 3.2

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWndEx::OnCreate(lpCreateStruct) == -1)
        return -1;
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWndEx::OnCreate(lpCreateStruct) == -1)
        return -1;

    BOOL bNameValid;
    // set the visual manager and style based on persisted value
    OnApplicationLook(theApp.m_nAppLook);

    if (!m_wndMenuBar.Create(this))
    {
        TRACE0("Failed to create menubar\n");
        return -1;          // fail to create
    }

    m_wndMenuBar.SetPaneStyle(m_wndMenuBar.GetPaneStyle()|CBRS_SIZE_DYNAMIC|CBRS_TOOLTIPS|CBRS_
FLYBY);
    // prevent the menu bar from taking the focus on activation
    CMFCPopupMenu::SetForceMenuFocus(FALSE);
    if   (!m_wndToolBar.CreateEx(this,   TBSTYLE_FLAT,   WS_CHILD|WS_VISIBLE|CBRS_TOP|CBRS_GRIPPER|
CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) || !m_wndToolBar.LoadToolBar(theApp.m_bHiColorIcons ?
IDR_MAINFRAME_256 : IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;          // fail to create
    }
    CString strToolBarName;
    bNameValid = strToolBarName.LoadString(IDS_TOOLBAR_STANDARD);
    ASSERT(bNameValid);
    m_wndToolBar.SetWindowText(strToolBarName);

    CString strCustomize;
    bNameValid = strCustomize.LoadString(IDS_TOOLBAR_CUSTOMIZE);
    ASSERT(bNameValid);
    m_wndToolBar.EnableCustomizeButton(TRUE, ID_VIEW_CUSTOMIZE, strCustomize);
    // Allow user-defined toolbars operations:
    InitUserToolbars(NULL, uiFirstUserToolBarId, uiLastUserToolBarId);

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

```
        // TODO: Delete these five lines if you don't want the toolbar and menubar to be dockable
        m_wndMenuBar.EnableDocking(CBRS_ALIGN_ANY);
        m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
        EnableDocking(CBRS_ALIGN_ANY);
        DockPane(&m_wndMenuBar);
        DockPane(&m_wndToolBar);

        // enable Visual Studio 2005 style docking window behavior
        CDockingManager::SetDockingMode(DT_SMART);
        // enable Visual Studio 2005 style docking window auto-hide behavior
        EnableAutoHidePanes(CBRS_ALIGN_ANY);

        // Load menu item image (not placed on any standard toolbars):
        CMFCToolBar::AddToolBarForImageCollection(IDR_MENU_IMAGES, theApp.m_bHiColorIcons ? IDB_MENU_
IMAGES_24 : 0);

        // create docking windows
        if (!CreateDockingWindows())
        {
            TRACE0("Failed to create docking windows\n");
            return -1;
        }

        m_wndFileView.EnableDocking(CBRS_ALIGN_ANY);
        m_wndClassView.EnableDocking(CBRS_ALIGN_ANY);
        DockPane(&m_wndFileView);
        CDockablePane* pTabbedBar = NULL;
        m_wndClassView.AttachToTabWnd(&m_wndFileView, DM_SHOW, TRUE, &pTabbedBar);
        m_wndOutput.EnableDocking(CBRS_ALIGN_ANY);
        DockPane(&m_wndOutput);
        m_wndProperties.EnableDocking(CBRS_ALIGN_ANY);
        DockPane(&m_wndProperties);
        // Enable toolbar and docking window menu replacement
        EnablePaneMenu(TRUE, ID_VIEW_CUSTOMIZE, strCustomize, ID_VIEW_TOOLBAR);

        // enable quick (Alt+drag) toolbar customization
        CMFCToolBar::EnableQuickCustomization();

        if (CMFCToolBar::GetUserImages() == NULL)
        {
            // load user-defined toolbar images
            if (m_UserImages.Load(_T(".\\UserImages.bmp")))
            {
                m_UserImages.SetImageSize(CSize(16, 16), FALSE);
                CMFCToolBar::SetUserImages(&m_UserImages);
            }
        }
        // enable menu personalization (most-recently used commands)
        // TODO: define your own basic commands, ensuring that each pulldown menu has at least one basic command.
        CList<UINT, UINT> lstBasicCommands;
```

```
    lstBasicCommands.AddTail(ID_FILE_NEW);
    lstBasicCommands.AddTail(ID_FILE_OPEN);
    lstBasicCommands.AddTail(ID_FILE_SAVE);
    lstBasicCommands.AddTail(ID_FILE_PRINT);
    lstBasicCommands.AddTail(ID_APP_EXIT);
    lstBasicCommands.AddTail(ID_EDIT_CUT);
    lstBasicCommands.AddTail(ID_EDIT_PASTE);
    lstBasicCommands.AddTail(ID_EDIT_UNDO);
    lstBasicCommands.AddTail(ID_APP_ABOUT);
    lstBasicCommands.AddTail(ID_VIEW_STATUS_BAR);
    lstBasicCommands.AddTail(ID_VIEW_TOOLBAR);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_OFF_2003);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_VS_2005);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_OFF_2007_BLUE);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_OFF_2007_SILVER);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_OFF_2007_BLACK);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_OFF_2007_AQUA);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_WIN7_SCENIC);
    lstBasicCommands.AddTail(ID_SORTING_SORTALPHABETIC);
    lstBasicCommands.AddTail(ID_SORTING_SORTBYTYPE);
    lstBasicCommands.AddTail(ID_SORTING_SORTBYACCESS);
    lstBasicCommands.AddTail(ID_SORTING_GROUPBYTYPE);
    CMFCToolBar::SetBasicCommands(lstBasicCommands);

GetMenu()->GetSubMenu(0)->CheckMenuItem(0,MF_BYPOSITION | MF_CHECKED );
    return 0;
}
```

编译并运行 Menu 程序，打开“File”子菜单，可以看到其下的“New”菜单命令左侧已经添加了一个复选标记。这时，如果希望取消这个标记，可以使用 MF_UNCHECKED 标记来调用 CheckMenuItem() 函数。

此外，还可以通过菜单命令标识来访问菜单命令。首先，打开菜单编辑器，利用菜单的属性面板，得到“New”菜单命令的标识：ID_FILE_NEW。然后，修改例 3.2 中加粗显示的那行代码，使用菜单命令标识来实现设置“New”菜单命令标记的功能，修改代码如例 3.3 所示。

例 3.3

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
…
    GetMenu()->GetSubMenu(0)->CheckMenuItem(ID_FILE_NEW, MF_BYCOMMAND | MF_CHECKED);
    return 0;
}
```

这时再次运行 Menu 应用程序，可以发现程序实现同样的效果，即为“File”子菜单下的“New”菜单命令添加了一个标记符号。

3.1.4　给菜单加入图标

在 Visual C++开发环境中，经常可以看到“Tools”子菜单下的很多菜单命令前面都带有图形，如图 3.20 所示。

要想实现这种在菜单命令前面添加图形的效果，可以使用 CMenu 类的 SetMenuItemBitmaps()函数，这个函数的作用是将指定的位图与菜单命令关联起来。该函数的声明如下：

```
BOOL SetMenuItemBitmaps(UINT nPostion,UINT nFlags, const CBitmap* pBmpUnchecked, const CBitmap*pBmp Checked);
```

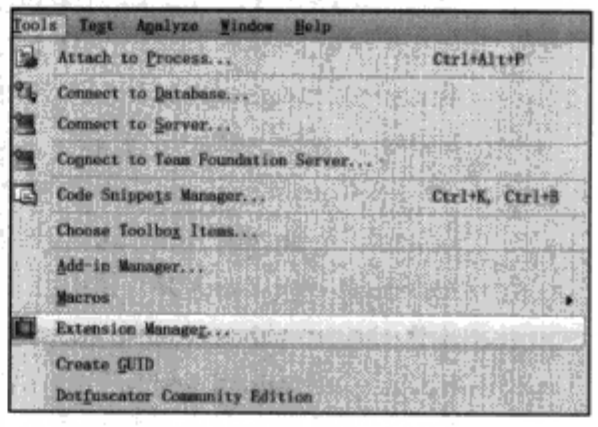


图 3.20　带有图标的菜单项

SetMenuItemBitmaps()函数有 4 个参数，同样，第一个参数（nPostion）的取值也是由第二个参数（nFlags）的值决定的。如果第二个参数的值是 MF_BYCOMMAND，则表明第一个参数的值是菜单命令标识；但如果第二个参数的值是 MF_BYPOSITION，则表明第一个参数的值是菜单命令位置索引。后两个参数都是 CBitmap 类型的指针，用来设置与菜单命令关联的两幅位图，其中，第三个参数（pBmpUnchecked）指定当取消菜单命令选中状态时的位图；第四个参数（pBmpChecked）指定选中菜单命令时显示的位图。这样，就可以让菜单命令显示这两幅位图来表明其选中和未选中状态。

以 Menu 程序中“查看”子菜单下的“工具栏”标记菜单命令为例，看看菜单命令选中与取消选中时的状态，以及产生的功能。当“工具栏”菜单处于选中状态时，即显示了该菜单命令的复选标记时，程序窗口中显示了工具栏。当再次选择“工具栏”菜单命令时，它前面的表示选中状态的标记消失，程序窗口中的工具栏也消失了。再次选择这个菜单命令时，它又处于选中状态，再次显示标记符号。同时，工具栏也在程序窗口中显示了。

为了要在菜单中加入图形标记，以显示图形标记菜单，首先应该准备的就是：图形。可以使用现成的图形，也可以新建一个位图资源，下面以新建一个新的位图资源为例进行讲解，即首先应该创建一个默认大小的白色背景的位图。

然后，可以在 CMainFrame 类的 OnCreate()函数中完成图形标记菜单的实现。这时因为要把位图作为图形标记菜单的内容来显示，这样就不能把位图作为局部变量来使用，否则当 OnCreate()函数调用结束时，这个位图对象的生命周期也就结束了，在程序中就看不到位图了。因此，要把位图对象设置为 CMainFrame 的成员变量，为 CMainFrame 类添加一个 CBitmap 类型的成员变量：m_bitmap。

在例 3.4 中加粗显示的代码是在 CMainFrame 类的 OnCreate()函数中添加的实现图形标记的菜单的具体代码。这段代码可以放在 OnCreate()函数的任意位置，但一定要在 return 语句之前。

例 3.4

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
…
    GetMenu()->GetSubMenu(0)->CheckMenuItem( 0, MF_BYCOMMAND | MF_CHECKED);
    GetMenu()->GetSubMenu(0)->SetDefaultItem( ID_FILE_OPEN,   FALSE);
    GetMenu()->GetSubMenu(0)-> SetDefaultItem ( 4, TRUE);

    m_bitmap.LoadBitmap(IDB_BITMAP1);
    GetMenu()->GetSubMenu(0) ->SetMenuItemBitmaps( 0, MF_BYPOSITION, &m_bitmap, &m_bitmap);

    return 0;
}
```

在例 3.4 所示的这段代码中，首先加载位图资源，初始化图形标记菜单将使用的位图对象，然后将程序“文件”子菜单下的“新建”菜单命令实现为图形标记菜单命令，并且将该菜单命令的选中和取消选中状态都设置为同一幅位图。

编译并运行 Menu 程序，打开“文件”子菜单，将发现“新建”菜单命令的前面并没有显示位图，这是因为在图形标记菜单上显示的位图的尺寸有固定的标准，这里创建的位图太大了，只显示了所创建的当前位图左上角的一小部分的内容，而这个位图资源左上角的这块范围没有内容，所以看到的结果就是在菜单命令前面的位图并没有明确显示出来。

通过 GetSystemMetrics()函数可以得到图形标记菜单上显示的位图的尺寸，这个函数的声明如下所示：

```
int   GetSystemMetrics ( int nIndex );
```

GetSystemMetrics()函数只有一个参数，即整型的 nIndex，它的作用是指定希望获取哪部分系统信息。当该函数的值为 SM_XMENUCHECK 或 SM_CYMENUCHECK 时，这个函数将获取标记菜单命令上标记图形的默认尺寸。前者是获得标记图形的宽度，后者将获得标记图形的高度。可以分别以这两个参数值调用 GetSystemMetrics()函数两次，从而得到标记菜单命令上位图的尺寸。于是，对例 3.4 所示的 Menu 程序中 CMainFrame 类的 OnCreate()函数进行改进，在加载位图的那行代码（即 LoadBitmap 调用）之前添加获取该位图尺寸的代码，结果如例 3.5 中加粗显示的代码所示。

例 3.5

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
…
    GetMenu()->GetSubMenu(0)->CheckMenuItem( 0, MF_BYCOMMAND | MF_CHECKED);
    GetMenu()->GetSubMenu(0)->SetDefaultItem( ID_FILE_OPEN,   FALSE);
    GetMenu()->GetSubMenu(0)-> SetDefaultItem ( 4, TRUE);

    CString str;
    Str.Format( *x=%d, y=%d", GetSystemMetrics(SM_CSMENUCHECK),GetSystemMetrics ( SM_CYMENUCHECK));
    MessageBox(str);

    m_bitmap.LoadBitmap(IDB_BITMAP1);
    GetMenu()->GetSubMenu(0) ->SetMenuItemBitmaps( 0, MF_BYPOSITION, &m_bitmap, &m_bitmap);

    return 0;
}
```

在上面的代码中，把获得的位图尺寸放到了字符串 str 中，然后利用 MessageBox()函数将其显示出来。

编译并运行 Menu 程序，这时会弹出一个消息框，它会表示作为图形标记菜单命令上显示的位图，其宽度和高度都是 13。

这样，就需要把程序中位图资源的尺寸调整到 13×13，但是如何知道位图资源当前的尺寸呢？当在位图资源编辑器中，用鼠标拖动位图资源外围的蓝色方框以改变位图资源大小时，在 Visual C++开发环境窗口的状态栏中有一格专门用来显示位图当前的尺寸信息，该信息会随着鼠标指针的移动而变化。在调整好位图资源的大小后，继续设置位图资源的内容。

然后，编译并运行 Menu 程序，此时打开运行程序的“文件”子菜单，可以看到在“新建”菜单命令的左侧就显示了刚刚创建的位图资源。

3.1.5　禁用菜单

运行 Menu 程序，并选择“文件”子菜单下的“打开”菜单命令，将会弹出“打开文件”对话框。

这是 MFC AppWizard 自动实现的特殊功能。如果要禁用某个菜单命令，可以使用下面的两种方法：一种是通过禁用屏幕文件打开的功能；另外一种是取消菜单命令的命令处理函数。

下面首先介绍第一种方法，要实现的是禁用这个菜单命令，以屏幕文件打开功能，这可以利用 CMenu 类的成员函数 EnableMenuItem()来完成。

EnableMenuItem()函数的作用是设置菜单命令的状态：启用（能够使用）、禁用或变灰显示。其声明形式如下所示：

```
UNIT   EnableMenuItem( UINT nIDEnableItem, UINT nEnable );
```

正如前面已经介绍过的 CMenu 类的其他成员函数一样，EnableMenuItem()函数的第一参数的含义也是由第二个参数决定的，后者可以是 MF_DISABLED、MF_ENABLED 或 MF_GRAYED 与 MF_BYCOMMAND 或 MF_BYPOSITION 的组合。

- MF_BYCOMMAND：指定第一个参数是菜单命令的标识 ID。
- MF_BYPOSITION：指定第一个参数是菜单命令的位置索引。
- MF_DISABLED：禁用菜单命令，用户不能选择该菜单命令，但该菜单命令并没有变成灰色。
- MF_ENABLED：使菜单命令可用，用户可以选择这个菜单命令。
- MF_GRAYED：禁用菜单命令，用户不能选择该菜单命令，并且该菜单命令变成灰色的显示形式。例如，运行 Menu 程序时，其“编辑”子菜单下的几个菜单命令都以灰色显示，即均不能单击也没有任何作用。同时，可以发现工具栏上与这几个菜单命令相对应的几个工具按钮也是不可用的。

Visual Basic 不能识别带有汉字的日期，如#2005 年 1 月 19 日#。

一般来说，如果某个菜单命令是变灰状态，是灰色显示的（即设置了 MF_GRAYED 标志）时，就知道这个菜单命令是不能用的，也是不能单击的。如果菜单命令是禁用状态（设置了 MF_DISABLED 标志），但菜单命令并没有变成灰色，用户就会认为这个菜单命令是可用的，但是，选择这个菜单命令时，程序并没有任何反应，这时用户一般就会认为这个程序有问题。因此，应该从用户的角度来考虑应用程序的编写，通常让菜单的禁用状态和变灰状态同时使用，即通常把 MF_GRAYED 和 MF_DISABLED 两个标志放在一起，用于菜单命令。虽然这种画龙点睛的做法并不是必需的，但可以让应用程序更加完美，也会让用户更加满意。

EnableMenuItem()函数可以禁用某个菜单命令，现在的问题是，应该在什么地方调用该函数，与上面的功能实现一样，在 CMainFrame 类的 OnCreate()函数中调用此函数可以实现禁用菜单命令的目标。下面将 EnableMenuItem()函数的代码添加到 CMainFrame 类的 OnCreate()函数中，希望可以通过这段代码禁用“文件”子菜单中的“打开”菜单命令（“打开”菜单命令在“文件”子菜单下的位置索引为 1），使其不再弹出打开对话框。程序代码如例 3.6 所示。

例 3.6

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CFrameWndEx::OnCreate(lpCreateStruct) == -1)
        return -1;

        BOOL bNameValid;
        // set the visual manager and style based on persisted value
        OnApplicationLook(theApp.m_nAppLook);
```

```
        if (!m_wndMenuBar.Create(this))

        TRACE0("Failed to create menubar\n");
        return -1;          // fail to create
    }

    m_wndMenuBar.SetPaneStyle(m_wndMenuBar.GetPaneStyle() | CBRS_SIZE_DYNAMIC | CBRS_TOOLTIPS |
CBRS_FLYBY);

    // prevent the menu bar from taking the focus on activation
    CMFCPopupMenu::SetForceMenuFocus(FALSE);

    if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE | CBRS_TOP | CBRS_GRIPPER |
CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) || !m_wndToolBar.LoadToolBar(theApp.m_bHiColorIcons ?
IDR_MAINFRAME_256 : IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;          // fail to create
    }

    CString strToolBarName;
    bNameValid = strToolBarName.LoadString(IDS_TOOLBAR_STANDARD);
    ASSERT(bNameValid);
    m_wndToolBar.SetWindowText(strToolBarName);

    CString strCustomize;
    bNameValid = strCustomize.LoadString(IDS_TOOLBAR_CUSTOMIZE);
    ASSERT(bNameValid);
    m_wndToolBar.EnableCustomizeButton(TRUE, ID_VIEW_CUSTOMIZE, strCustomize);

    // Allow user-defined toolbars operations:
    InitUserToolbars(NULL, uiFirstUserToolBarId, uiLastUserToolBarId);

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

    // TODO: Delete these five lines if you don't want the toolbar and menubar to be dockable
    m_wndMenuBar.EnableDocking(CBRS_ALIGN_ANY);
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockPane(&m_wndMenuBar);
    DockPane(&m_wndToolBar);
```

```
        // enable Visual Studio 2005 style docking window behavior
        CDockingManager::SetDockingMode(DT_SMART);
        // enable Visual Studio 2005 style docking window auto-hide behavior
        EnableAutoHidePanes(CBRS_ALIGN_ANY);

        // Load menu item image (not placed on any standard toolbars):
        CMFCToolBar::AddToolBarForImageCollection(IDR_MENU_IMAGES, theApp.m_bHiColorIcons ? IDB_MENU_
IMAGES_24 : 0);

        // create docking windows
        if (!CreateDockingWindows())
        {
            TRACE0("Failed to create docking windows\n");
            return -1;
        }

        m_wndFileView.EnableDocking(CBRS_ALIGN_ANY);
        m_wndClassView.EnableDocking(CBRS_ALIGN_ANY);
        ockPane(&m_wndFileView);
        CDockablePane* pTabbedBar = NULL;
        m_wndClassView.AttachToTabWnd(&m_wndFileView, DM_SHOW, TRUE, &pTabbedBar);
        m_wndOutput.EnableDocking(CBRS_ALIGN_ANY);
        DockPane(&m_wndOutput);
        m_wndProperties.EnableDocking(CBRS_ALIGN_ANY);
        DockPane(&m_wndProperties);


        // Enable toolbar and docking window menu replacement
        EnablePaneMenu(TRUE, ID_VIEW_CUSTOMIZE, strCustomize, ID_VIEW_TOOLBAR);

        // enable quick (Alt+drag) toolbar customization
        CMFCToolBar::EnableQuickCustomization();

        if (CMFCToolBar::GetUserImages() == NULL)
        {
            // load user-defined toolbar images
            if (m_UserImages.Load(_T(".\\UserImages.bmp")))
            {
                m_UserImages.SetImageSize(CSize(16, 16), FALSE);
                CMFCToolBar::SetUserImages(&m_UserImages);
            }
        }

        // enable menu personalization (most-recently used commands)
        // TODO: define your own basic commands, ensuring that each pulldown menu has at least one basic command.
        CList<UINT, UINT> lstBasicCommands;

        lstBasicCommands.AddTail(ID_FILE_NEW);
        lstBasicCommands.AddTail(ID_FILE_OPEN);
```

```
    lstBasicCommands.AddTail(ID_FILE_SAVE);
    lstBasicCommands.AddTail(ID_FILE_PRINT);
    lstBasicCommands.AddTail(ID_APP_EXIT);
    lstBasicCommands.AddTail(ID_EDIT_CUT);
    lstBasicCommands.AddTail(ID_EDIT_PASTE);
    lstBasicCommands.AddTail(ID_EDIT_UNDO);
    lstBasicCommands.AddTail(ID_APP_ABOUT);
    lstBasicCommands.AddTail(ID_VIEW_STATUS_BAR);
    lstBasicCommands.AddTail(ID_VIEW_TOOLBAR);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_OFF_2003);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_VS_2005);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_OFF_2007_BLUE);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_OFF_2007_SILVER);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_OFF_2007_BLACK);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_OFF_2007_AQUA);
    lstBasicCommands.AddTail(ID_VIEW_APPLOOK_WIN7_SCENIC);
    lstBasicCommands.AddTail(ID_SORTING_SORTALPHABETIC);
    lstBasicCommands.AddTail(ID_SORTING_SORTBYTYPE);
    lstBasicCommands.AddTail(ID_SORTING_SORTBYACCESS);
    lstBasicCommands.AddTail(ID_SORTING_GROUPBYTYPE);
    CMFCToolBar::SetBasicCommands(lstBasicCommands);

GetMenu()->GetSubMenu(0)->EnableMenuItem( 1, MF_BYPOSITION | MF_DISABLED);

    return 0;
}
```

编译并运行 Menu 应用程序，选择“文件”子菜单下的“打开”菜单命令，这时仍然会弹出“打开文件”对话框，这说明“打开”菜单命令并没有被禁用。这时，要找出程序出现问题以致出错的原因。首先，可能会想到是不是使用的菜单位置索引不对，但是“文件”子菜单的位置索引确实是 0，而“打开”菜单命令在其子菜单下的位置索引也确实是 1，因此，可以排除菜单命令索引出错的可能性；接下来，可能会想到，是不是 MF_DISABLED 和 MF_GRAYED 这两个标志必须一起使用？在上述代码中再加上 MF_GRAYED 标志，结果发现菜单命令还是没有被禁用，说明也不是这个原因；最后，求助于 MSDN，在 MSDN 中找到关于函数 EnableMenuItem()的声明，在该声明的最后提供的例子中，有一处注释提示信息，说明文字如下：

```
//NOTE:m_bAutoMenuEnable is set to FALSE in the constructor of
//CMainFrame so no ON_UPDATE_COMMAND_UI or ON_COMMAND handlers are
//needed, and CMenu::EnableMenuItem() will work as expected.
```

这个注释信息要说明的是，一旦在 CMainFrame 类的构造函数中把成员变量 m_bAutoMenuEnable 设置为 False 后，就不需要对 ON_UPDATE_COMMAND_UI 或 ON_COMMAND 消息进行响应处理了。CMenu 类的 EnableMenuItem()函数就能够正常工作。

实际上，MFC 为菜单提供了一种命令更新的机制，所以在程序运行时，根据此机制去判断哪个菜单可以使用，哪个菜单不能够使用，然后据此机制去显示其相应的状态。默认情况下，所有的菜单命令的更新都是由 MFC 的命令更新机制完成的。如果想自己更改菜单命令的状态，那就必须先把 m_bAutoMenuEnable 变量设置为 False，之后，对菜单命令的状态更新才能起作用。因此，在程序 Menu 的 CMainFrame 类构造函数中把 m_bAutoMenuEnable 这个变量初始化为 False，代码如例 3.7 所示。

例 3.7

```
CMainFrame::CMainFrame()
{
    //TODO:add member initialization code here
    m_bAutoMenuEnable = FALSE;
}
```

再次编译并运行 Menu 应用程序，此时如果选择“文件”子菜单下的“打开”菜单命令，将会发现程序没有弹出“打开文件”对话框，这就说明 EnableMenuItem()函数起作用了，“打开”菜单被禁用了。但是，这种菜单显示方式让人感觉有点别扭，这是因为在整个菜单显示的形式上看，感觉它好像应该是可以使用的，但是单击时却没有任何反应。

因此，通常将 MF_DISABLED 和 MF_GRAYED 这两个标志一起使用，修改例 3.6 所示的 OnCreate()函数中加粗显示的那行代码，在 EnableMenuItem()函数中加上 MF_GRAYED 标志，如例 3.8 所示。

例 3.8

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
...
    GetMenu()->GetSubMenu(0)->EnableMenuItem( 1, MF_BYPOSITION | MF_DISABLED | MF_GRAYED);

    return 0;
}
```

此时，如果再编译并运行 Menu 程序，选择“文件”菜单，可以发现“打开”菜单命令已经变灰了，选择这个菜单命令，它也不起作用了。

当然，也可以通过利用菜单命令标识调用 EnableMenuItem()函数来实现，方法是把这个函数的第一个参数换成“打开”菜单命令的标识 ID，并把 MF_BYPOSITION 标志换成 MF_BYCOMMAND 标志来实现。

另外，当在 CMainFrame 类的构造函数中把成员变量 m_bAutoMenuEnable 设置成 False 后，将会发现 Menu 程序的“编辑”子菜单下的几个菜单命令不再以灰色显示，这是因为当 m_bAutoMenuEnable 设置为 False 后，MFC 就不再利用它的菜单命令更新机制去判断哪个菜单可以使用，哪个菜单不能使用了，所以也就不能根据菜单命令的状态以不同的外观来显示。而此时菜单是否使用这些判断操作，就需要由开发人员自己去完成。

下面介绍禁用菜单命令的第二种方法。

它的策略就是：如果一个新的菜单命令没有命令处理函数，应用程序会自动禁用它。要启用这样的菜单命令，只要求用 Event Handler Appwizard 为该菜单命令添加一个命令处理函数。但是，如果想根据应用程序的某些条件，有选择地启用和禁用一个菜单命令，可以用 Event Handler Appwizard 添加一个界面消息处理函数，步骤如下。

（1）就像添加一个菜单命令到类中一样，但在选择命令函数时不是选择 COMMAND，而是选择 UPDATE_COMMAND_UI。然后，Event Handler Appwizard 会为类创建一个用户界面处理函数。

（2）添加下面的代码到这个新的处理函数中。在本例中，如果 m_bWzd 为 True，菜单将变得有效，如果是 False，则无效。

```
void CWzdView::OnUpdateWzdButton(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(m_bWzd);
}
```

- 当应用程序首次打开一个菜单时，它将查看在该菜单中每个菜单命令是否有一个 UPDATE_COMMAND_UI 消息处理函数；如果没有，应用程序就检查该菜单命令是否有某种 COMMAND 消息处理函数，如果也没有这样的处理函数，应用程序会自动禁用该菜单命令，并使它显示灰色。通过在 CMainFrame 的构造函数中把 m_bAutoMenuEnable 设为 False，可以禁用该自动功能。在软件开发中，该功能有助于提醒用户还有什么没做。
- 应用程序不仅可以启用和禁用菜单命令，也可以启用和禁用工具栏按钮、对话框控件和状态栏面板。

一般来说，作为一个用户，总是不喜欢禁用的菜单命令和工具栏按钮，因此，建议开发人员不要禁用一个菜单开发命令，而是显示一个消息框，用于指示一个菜单命令是不能用的，并告知怎样使它可用。

3.1.6　移除与加载菜单

在程序中，如果要移除一个菜单，可以通过 CWnd 类提供的 SetMenu()成员函数来实现。该函数的声明形式如下所示：

```
BOOL SetMenu（ CMenu* pMenu）;
```

SetMenu()函数只有一个参数，即 CMenu 类型指针的参数，它指向一个新菜单对象。如果这个参数值为 NULL，则当前菜单就被移除了，所以如果要移除 Menu 程序的菜单，可以在 CMainFrame 类的 OnCreate()函数的最后，但一定要在此 return 语句之前，添加 SetMenu()函数的调用，如例 3.9 所示。

例 3.9

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
...
    GetMenu()->GetSubMenu(0)->EnableMenuItem( 1, MF_BYPOSITION | MF_DISABLED |   MF_GRAYED);

    SetMenu(NULL);

    return 0;
}
```

此时，如果编译并运行 Menu 函数，将会发现 Menu 程序没有菜单了。

当然，也可以在程序中装载一个菜单资源并显示出来。例如，在例 3.9 中，将 Menu 应用程序的菜单移除了。这里，可以再把它显示出来，例 3.10 就是具体的实现代码，其中加粗显示的部分就是新添的代码。

例 3.10

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
...
    GetMenu()->GetSubMenu(0)->EnableMenuItem( 1, MF_BYPOSITION | MF_DISABLED |   MF_GRAYED);
    SetMenu(NULL);

    CMenu menu;
    menu.LoadMenu(IDR_MAINFRAME);
    SetMenu( &menu) ;

    return 0;
}
```

这段刚刚添加的代码首先定义了一个菜单对象——menu，其类型为 CMenu；然后与位图对象需要加载位图资源一样，也需要把菜单资源加载到这个菜单对象中，Menu 程序主菜单的资源标识是 IDR_MAINFRAME；最后调用 SetMenu()函数，把程序的菜单设置为刚刚加载的菜单对象。

编译并运行 Menu 程序，会发现 Menu 程序的菜单又显示了。

在程序中，除了使用 MFC 自动创建的 IDR_MAINFRAME 菜单以外，还可以创建自己的一个菜单资源并加载，然后调用 SetMenu()函数，从而使程序的菜单变成自定义的菜单。通过这种方式，可以实现动态更换程序菜单的功能。

如果上面的程序代码在 Windows NT 或者 Windows XP 下运行，其实存在着一个问题，即这里定义的 CMenu 对象——menu 其实是一个局部对象，但程序似乎并未出现异常。实际上，这段代码是有问题的，即运行过程中会显示如图 3.21 所示的异常信息对话框，这个异常的产生是因为这里的 CMenu 对象——menu 是一个局部对象。

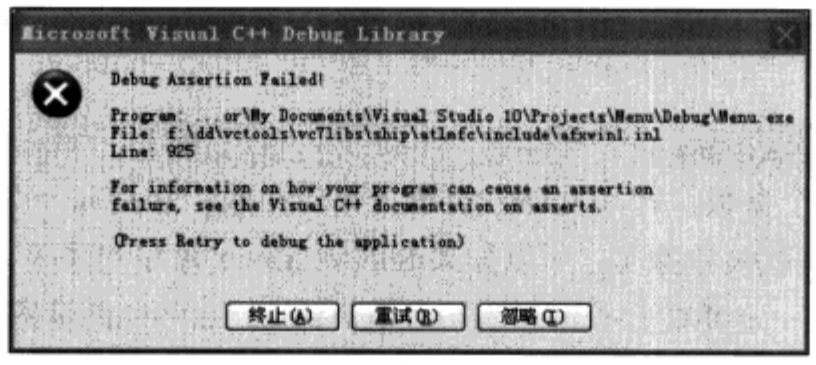


图 3.21　异常信息对话框

为了解决这个问题，可以把例 3.10 中的 CMenu 对象定义为 CMainFrame 类的一个成员变量。除此之外，还有一种解决方式，下面将重点介绍：仍把这个菜单对象定义为局部对象，但在调用 SetMenu()函数时，把此对象设置为窗口的菜单之后，立即调用 CMenu 类的另一个成员函数 Detach()，以便把菜单句柄与这个菜单对象分离。SetMenu()函数会把窗口的菜单设置成为其参数指定的新菜单，导致窗口重绘，以反映菜单的这种变化，同时也将该菜单对象的所有权交由该窗口对象。而随后的 Detach()函数会把菜单句柄与这个菜单对象分离，这样当这个局部菜单对象的生命周期结束时，它不会去销毁一个它不再具有拥有权的菜单。这个菜单在窗口销毁时会自动销毁。因此，在例 3.10 所示代码的最后，应该再添加对菜单对象的 Detach()函数的调用，代码如例 3.11 所示。

例 3.11

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
...
    GetMenu()->GetSubMenu(0)->EnableMenuItem( 1, MF_BYPOSITION | MF_DISABLED |   MF_GRAYED);
    SetMenu(NULL);

    CMenu menu;
    menu.LoadMenu(IDR_MAINFRAME);
    SetMenu( &menu) ;

    menu.Detach();

    return 0;
}
```

此时，再编译并运行 Menu 程序，就会发现程序一切正常，不会出现刚才的非法操作及异常消息

对话框。

3.2　菜单消息的传输机制

3.2.1　菜单消息的分类

菜单命令是一种消息，在 Windows 中，消息分为以下 3 类：

1．标准消息

除 WM_COMMAND 之外，所有以 WM_开头的消息都是标准消息。从 CWnd 派生的类，都可以接收到这类消息。

2．命令消息

命令消息是来自菜单、加速键或工具栏按钮的消息，这类消息都以 WM_COMMAND 形式呈现。在 MFC 中，通过菜单项的标识（ID）来区分不同的命令消息；在 SDK 中，通过消息的 wParam 参数识别。从 CCmdTarget 派生的类，都可以接收到这类消息。

3．通告消息

通告消息是由控件产生的消息，例如按钮的单击、列表框中选项的选择等都会产生这类消息，目的是向父窗口（通常是对话框）通知事件的发生。这类消息也是以 WM_COMMAND 形式呈现的。从 CCmdTarget 派生的类，都可以接收到这类消息。

通过 MSDN 提供的 MFC 类层次结构图，可以发现 CWnd 类实际上派生于 CCmdTarget 类。也就是说，凡是从 CWnd 派生的类，既可以接收标准消息，也可以接收命令消息和通告消息。而对于那些从 CCmdTarget 派生的类，则只能接收命令消息和通告消息，不能接收标准消息。

其中，文档类（CMenuDoc）和应用程序类（CWinApp）都派生于 CCmdTarget 类，所以它们可以接收菜单命令消息。但是因为它们不是从 CWnd 类派生的，所以不能接收标准消息。添加一个菜单后，并在视图类、文档类、框架类、应用程序类（即 C*App 类）添加该菜单的消息响应函数时，它的执行顺序是：视图类→文档类→框架类→应用程序类。当单击命令菜单时，首先接收消息的是框架类，然后框架类将其交给视图类，视图类根据消息映射机制看能否处理，不能则将其交给文档类。文档类处理过程与视图类相同。假如不能处理，则交给视图类，由视图类交给框架类，框架类根据消息映射机制看能否处理，不能则交给应用程序类处理。

3.2.2　菜单消息的传输路由

下面要讲解的是菜单消息的传输路由机制。

首先打开 Menu 应用程序，给 Menu 程序中的视图类添加“Test”菜单命令的命令响应函数。添加完成之后，可以发现，这时在应用程序的 3 个位置添加了与菜单命令消息响应函数相关的信息。

首先，在视图类的头文件中，在两个 AFX_MSG 注释宏之间添加了命令消息响应函数原型。

然后，在视图类的源文件中有两处消息，一处是在两个 AFX_MSG_MAP 注释宏之间添加了 ON_COMMAND 宏，将菜单 ID 号与命令响应函数关联起来；另一处是在视图类源文件中的命令消息响应函数的实现代码，如例 3.12 所示。

例 3.12

```
// MenuView.h : interface of the CMenuView class
//
/////////////////////////////////////////////////////////////////////////////
...
class CMenuView : public CView
{
        protected: //create from serialization only
        CMenuView();
        DECLARE_DYNCREATE(CMenuView)

...
    //Generated message map functions
    protected:
    //{{ AFX_MSG(CMenuView)
    afx_msg void OnTest();
    //}}AFX_MESSAGE_MAP()
};
//MenuView.cpp : implementation of the CMenuView class
//

#include "stdafx.h"
#include "Menu.h"

#include "MenuDoc.h"
#include "MenuView.h"
#include"resource.h"

...
/////////////////////////////////////////////////////////////////////////////
// CMenuView
IMLEMENT_DYNCREATE( CMenuView, CView)

BEGIN_MESSAGE_MAP(CMenuView, CView)
ON_WM_CONTEXTMENU()
//{{AFX_MSG_MAP(CMenuView)
ON_COMMAND(IDM_TEST, OnTest)
//}}AFX_MSG_MAP
// Standard printing commands
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()

...
void CMenuView::OnTest()
{
    //TODD:Add your command handler code here
}
```

可以发现，菜单命令消息响应函数的映射与标准消息的映射是相同的，只是命令消息使用的是 ON_COMMAND 宏。但是，命令消息与标准消息的传输路由过程还是有区别的，消息的传输路由过程如图 3.22 所示。

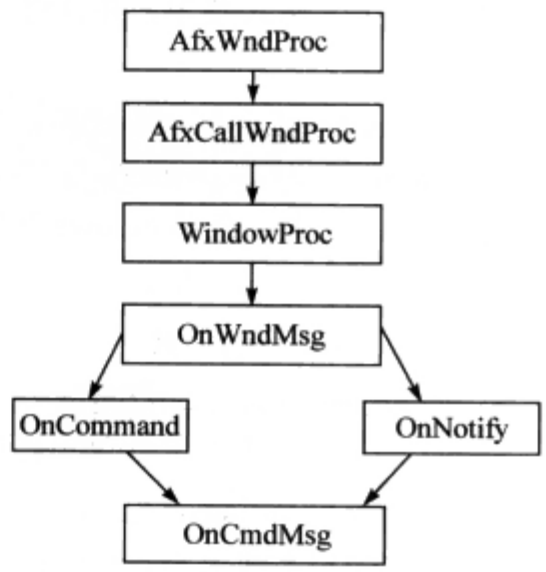


图 3.22　消息的传输路由过程

MFC 在后台把窗口过程函数换成了 AfxWndProc()函数，其中可以在 MFC 的源代码中查看一下这个函数，由这个函数对所有的消息进行处理。这个函数内部将调用 AfxCallWndProc()函数，后者又将调用 WindowProc()函数，这是 CWnd 类的一个成员函数，应用程序所有类型的消息都会进入这个函数中。WindowProc()函数又将调用 OnWndMsg()函数，这个函数会对接收到的消息进行类型判断。

如果是标准消息，就利用消息映射机制来查找哪个类响应了当前这个消息，并调用相应的消息映射函数，完成对消息的处理；如果是命令消息，它会交由 OnCommand()函数进行处理，该函数将完成命令消息的路由；而如果是通告消息，那么它将交由 OnNotify()函数进行处理，该函数将完成通告消息的路由。

命令消息和通告消息最后都会调用 OnCmdMsg()函数。

下面以 Menu 程序为例，来看看菜单命令消息传输的具体过程：当选择某个菜单命令时，最先接收到这个菜单命令消息的是框架类。

（1）框架类将把接收到的这个消息交给它的子窗口，即视图类，由视图类首先进行处理。

（2）视图类首先根据命令消息映射机制查找自身是否对此消息进行了响应，如果响应了，就调用相应的响应函数对这个消息进行处理，消息传输过程结束；如果视图类没有对此命令消息做出响应，就交由文档类。

（3）文档类同样查找自身是否对这个菜单命令进行了响应，如果响应了，就由文档类的命令消息响应函数进行处理，传输路由过程结束；如果文档类也未做出响应，就把这个命令消息交还给视图类，后者又把该消息交还给框架类。

（4）框架类查看自己是否对这个命令消息进行了响应，如果它也没有做出响应，就把这个菜单命令消息交给应用程序类，由后者来进行处理。

这就是菜单命令的传输机制。

3.3　动态菜单操作

前面所讲的创建菜单及修改菜单都是在应用程序设计阶段，通过菜单编辑器来操作的。一般来说，在程序运行过程中，仍然可以根据需要对程序的菜单进行添加、插入和删除操作。动态操作包括两种情况，一种是针对弹出式菜单的动态操作，一种是针对菜单命令的动态操作。

本节将介绍如何在应用程序运行时动态地修改和创建菜单。

3.3.1　添加菜单

首先，重新创建一个单文档类型的 MFC AppWizard(exe)工程，工程取名为 Menu2，在上面进行添加菜单项目（包括子菜单和菜单命令两种）的操作时，都是在资源编辑器中进行的。现在要通过代码来动态地添加菜单项目，这可以利用 CMenu 类提供的成员函数来完成。

- CreatePopupMenu()函数。

```
BOOL CreatePopupMenu( );
```

创建一个弹出式菜单，并且将该弹出式菜单和一个 CMenu 对象相关联，该菜单创建后初始时是空的。例如：

```
CMenu m_Menu;
m_Menu.CreatePopupMenu();
```

需要注意的是，在 Windows 2000 下，m_Menu 应定义为成员变量，否则会报错，而在 Windows NT 下则可以定义为局部变量。

- AppendMenu()函数。

```
BOOL AppendMenu( UINT nFlags, UINT nIDNewItem = 0, LPCTSTR lpszNewItem = NULL );
```

添加一个弹出式菜单到菜单栏或添加一个菜单命令到弹出式菜单。

第一个参数 nFlags，它指定新添加的菜单项目的状态信息，有多种这样的状态标志，它的值可以为下列值中的一个或多个：MF_CHECKED、MF_UNCHECKED、MF_DISABLED、MF_ENABLED、MF_GRAYED、MF_MENUBARBREAK、MF_MENUBREAK、MF_OWNERDRAW、MF_POPUP、MF_SEPARATOR、MF_STRING，如表 3.1 所示。当 nFlags 为 MF_POPUP 时，表示要添加一个已创建好的弹出式菜单到菜单栏中；当 nFlags 为 MF_STRING 时，表示要添加一个菜单命令到一个弹出式菜单中，剩余的其他选项基本是和菜单命令的状态有关的，不过下面几种值是不能够一起出现的：

```
MF_DISABLED, MF_ENABLED, and MF_GRAYED
MF_STRING, MF_OWNERDRAW, MF_SEPARATOR
MF_MENUBARBREAK and MF_MENUBREAK
MF_CHECKED and MF_UNCHECKED
```

表 3.1　　菜单标志

菜 单 标 志	说　　明
MF_CHECKED	在菜单命令前面添加一个对号标记
MF_UNCHECKED	移除菜单命令前面的对号标记
MF_ENABLED	使菜单命令可用
MF_DISABLED	禁用菜单命令
MF_CRAYED	禁用菜单命令并以灰色显示
MF_OWNERDRAW	表明菜单命令是一个 owner_draw 风格的菜单命令
MF_POPUP	表明是一个弹出式菜单命令
MF_SEPARATOR	表明是一个分隔栏项
MF_STRING	使用一个字符串表示该菜单命令
MF_MENUBREAK	更改菜单命令的显示方式。例如，如果是为一个弹出式菜单添加这样的菜单命令，那么将为这个菜单新添加一列以显示新添加的这个菜单命令，但在列之间并没有显示分隔线
MF_MENUBREAK	同上，但在菜单命令列之间显示分隔线

第二个参数 nIDNewItem 表示添加的菜单命令的 ID，该参数的值取决于第一个参数。当第一个参数 nFlags 为 MF_POPUP 时，第二个参数 nIDNewItem 的值为要添加的弹出式菜单的句柄值；当第一个参数 nFlags 为 MF_STRING 时，第二个参数 nIDNewItem 的值表示添加的菜单命令的 ID；当第一个参数 nFlags 为 MF_SEPARATOR 时，第二个参数 nIDNewItem 的值可以忽略。

第三个参数 lpszNewItem 表示添加的弹出式菜单或菜单命令的名称，其取值同样取决于第一个参数：当第一个参数 nFlags 为 MF_POPUP 或 MF_STRING 时，第三个参数 lpszNewItem 的值是一个字符串，用来表示添加的弹出式菜单或菜单命令的名称；如果第一个参数 nFlags 为 MF_SEPARATOR 时，第三个参数 lpszNewItem 的值可以忽略。

为了添加菜单，首先需要创建一个菜单对象，这可以利用 CMenu 类提供的另一个成员函数 CreatePopupMenu()来实现。这个函数的作用是创建一个弹出式菜单，然后将其与一个 CMenu 对象关联起来。同样，本例中，也在程序的 CMainFrame 类的 OnCreate()函数中实现动态添加菜单的这一功能。具体代码如例 3.13 所示。

例 3.13

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        ...
        CMenu m_Menu;
        m_Menu.CreatePopupMenu();
        GetMenu()->AppendMenu(MF_POPUP, (UINT)m_Menu.m_hMenu , "星期");
        m_Menu.AppendMenu(MF_STRING,111, " 星期六 ");
        m_Menu.AppendMenu(MF_STRING,112, " 星期日 ");
         return 0;
}
```

上面这段代码可以放在 CmainFrame::OnCreate()函数中，运行的结果是在菜单栏中“帮助”菜单的后面，即整个菜单栏的尾部添加了一个弹出式菜单“星期”，而这个弹出式菜单下面有两个菜单命令，分别为“星期六”和“星期日”，运行结果如图 3.23 所示。

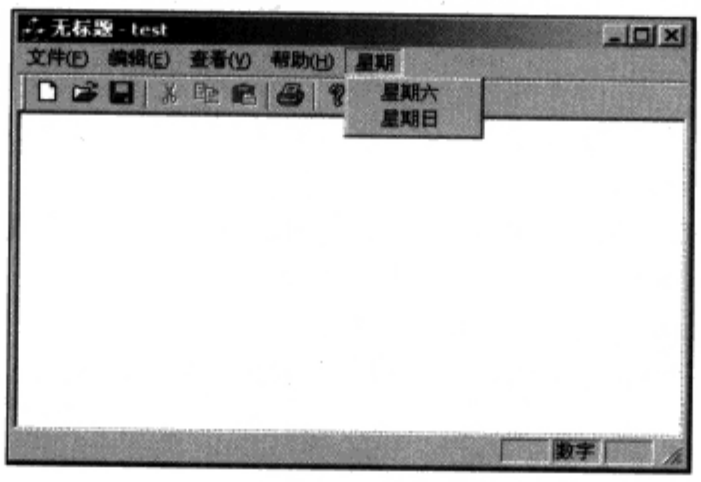


图 3.23　添加菜单项运行结果

例 3.13 所示的代码段首先定义了一个 CMenu 对象——m_menu，然后调用窗口的 GetMenu()函数得到程序框架窗口菜单栏的指针，并利用该指针调用 AppendMenu()函数，以添加一个新的弹出式菜单。这个新的弹出式菜单显示的文本为“星期”。因为 CMenu 对象的 m_hMenu 成员变量是菜单句柄，其类型为 HMENU，但这里函数要求的是 UINT 类型，所以必须进行强制转换。为创建好的子菜单添加菜单命令，这时利用 AppendMenu()函数实现，添加了两个菜单命令：星期六和星期日。

另外，现在添加的是一个弹出式菜单，相当于是程序框架窗口菜单的一个子菜单，所以在获得程序框架窗口的指针后，也不必再去调用 GetSubMenu()函数以得到某个子菜单了。

3.3.2　插入菜单

除了在已有的菜单后面添加新的菜单以外，也可以在已有的菜单之间插入一个新的菜单，这包括在两个子菜单之间插入一个子菜单，以及在两个菜单命令之间插入一个新的菜单命令。这可以利用 CMenu 类的 InsertMenu()成员函数来实现。这个函数的一种声明具有以下形式：

```
BOOL InsertMenu( UINT nPosition, UINT nFlags, UINT nIDNewItem = 0, LPCTSTR lpszNewItem = NULL );
```

在菜单栏中插入一个已创建的弹出式菜单或在一个弹出式菜单中插入一个菜单命令。与

AppendMenu()不同，AppendMenu ()是在尾部添加一个，而 InsertMenu()是在中间插入一个。

InsertMenu()函数有 4 个参数，第一个参数 nPosition 表示在 nPosition 指定的位置前插入新的菜单或菜单命令。nPosition 可以指定为菜单或菜单命令的 ID 号，也可以指定为菜单或菜单命令的位置索引，具体指定为 ID 还是位置索引，要看第二个参数 nFlags 设置为 MF_BYCOMMAND 还是 MF_BYPOSITION，当 nFlags 参数指定的是 MF_BYCOMMAND 标志时，第一个参数指定的是一个菜单命令标识，表示新菜单命令将在这个标识所表示的菜单命令之前插入；而如果 nFlags 参数指定的是 MF_BYPOSITION 标志，那么，新菜单命令将在第一个参数指定的位置所表示的菜单命令之前插入，也就是说，第一个参数这时所表示的就是这个新菜单命令插入后所在的位置。然而第二个参数 nFlags 不仅仅只有这两个值，它还包括许多指示菜单或菜单命令状态信息的值，具体用法和 AppendMenu()中的 nFlags 类似。

第三个参数和第四个参数的用法说明请参见 AppendMenu()中的 nIDNewItem 和 lpszNewItem。

在 Menu2 程序中，“编辑”子菜单的位置索引是 1，所以如果要在 Menu2 程序的“编辑”和“查看”这两个子菜单之间插入一个名为“月份”的菜单，插入位置应该是 2。应该把例 3.13 代码中调用 AppendMenu()函数的那行代码注释起来，并在其后添加上 InsertMenu()函数调用，详细代码如例 3.14 所示。

例 3.14

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
...
        CMenu m_Menu;
        m_Menu.CreatePopupMenu();
        // GetMenu()->AppendMenu(MF_POPUP, (UINT)m_Menu.m_hMenu ,"星期");
        // m_Menu.AppendMenu(MF_STRING,111," 星期六 ");
        // m_Menu.AppendMenu(MF_STRING,112," 星期日 ");
        GetMenu()->InsertMenu(2,MF_BYPOSITION|MF_POPUP,(UINT)m_Menu.m_hMenu ,"月份");
        return 0;
}
```

编译并运行 Menu2 程序，此时会发现在“编辑”和“查看”这两个菜单之间多了一个名为“月份”的菜单。

如果要在新插入的菜单“月份”中添加菜单命令“一月份”、“二月份”和“三月份”，同样可以使用 AppendMenu()函数来实现，可以在例 3.14 所示代码中新添加的 InsertMenu()函数之后添加如例 3.15 所示的代码中加粗显示的那几行代码。

例 3.15

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
......
        CMenu m_Menu;
        m_Menu.CreatePopupMenu();
        // GetMenu()->AppendMenu(MF_POPUP, (UINT)m_Menu.m_hMenu ,"星期");
        // m_Menu.AppendMenu(MF_STRING,111," 星期六 ");
        // m_Menu.AppendMenu(MF_STRING,112," 星期日 ");
        GetMenu()->InsertMenu(2,MF_BYPOSITION|MF_POPUP,(UINT)m_Menu.m_hMenu ,"月份");
        m_menu.AppendMenu(MF_STRING,111,"一月份");
        m_menu.AppendMenu(MF_STRING,112,"二月份");
```

```
        m_menu.AppendMenu(MF_STRING,113,"三月份");

        return 0;
    }
```

编译并运行 Menu2 程序，单击“月份”菜单，将发现它的下面多了 3 个菜单命令，就是上述代码创建的 3 个菜单命令。

如果想要在 Menu2 应用程序的“文件”菜单下添加一个菜单命令，那应该如何做呢？可以在 CMainFrame 类的 OnCreate()函数中添加下面这行代码：

```
GetMenu()->GetSubMenu(0)->AppendMenu(MF_STRING,114,“页面设置”);
```

此时，编译并运行 Menu2 应用程序，单击“文件”子菜单，将会发现它的下面多了一个菜单——页面设置。

而如果想要在 Menu2 程序的“文件”子菜单的“新建”和“打开”菜单命令之间插入一个新的菜单命令，可以在 CMainFrame 类的 OnCreate()函数中添加下面这行代码：

```
GetMenu()->GetSubMenu(0)->InsertMenu(ID_FILE_OPEN,MF_BYCOMMAND | MF_STRING, 115“发送”);
```

此时，编译并运行 Menu2 应用程序，单击“文件”菜单，将会发现在“新建”和“打开”菜单命令之间多了一个菜单命令“发送”。

3.3.3　删除菜单

CMenu 类还提供了两个成员函数，分别是 DeleteMenu()成员函数和 RemoveMenu()成员函数，这两个函数具有如下形式的声明：

```
BOOL RemoveMenu( UINT nPosition, UINT nFlags );
BOOL DeleteMenu( UINT nPosition, UINT nFlags );
```

RemoveMenu()函数是移除一个弹出式菜单或菜单命令，而 DeleteMenu()函数是删除一个菜单或菜单命令。两者的区别是，当用 DeleteMenu()函数删除一个弹出式菜单时，它不仅从菜单栏中删除这个菜单，而且将这个弹出式菜单的句柄销毁，释放空间。它可以删除一个菜单项目，包括子菜单，以及子菜单下的菜单命令，主要取决于调用这个函数的对象，如果该对象是程序的菜单栏对象，那么删除的就是指定的子菜单，如果该对象是一个子菜单对象，那么删除的就是该子菜单下的一个菜单命令；而用 RemoveMenu()函数移除一个弹出式菜单时，它仅仅是从菜单栏中删除这个菜单，而这个菜单的句柄仍保留，空间不释放。

二者的参数用法是相同的。

第一个参数 nPosition 表示将要删除的菜单或菜单命令的 ID 或位置索引。用 ID 号表示还是用位置索引表示取决于第二个参数。

第二个参数 nFlags 为 MF_BYCOMMAND 时，第一个参数用菜单或菜单命令的 ID 表示；为 MF_BYPOSITION 时，第一个参数用位置索引表示。

下面仍以 Menu2 程序为例，如果想要删除其“编辑”子菜单，那么就可以在 CMainFrame 类的 OnCreate()函数的最后，但一定要在 return 语句之前，添加下面这行代码：

```
GetMenu()->Delete(1,MF_BYYPOSITION);
```

编译并运行 Menu2 程序，将会发现“编辑”子菜单已经消失了。

如果要删除一个菜单命令，例如要删除“文件”子菜单下的“打开”菜单命令，则首先需要得到该菜单命令所在的子菜单，然后既可以按照菜单命令的标识，也可以根据其位置索引来删除它。实现代码如下所示：

```
GetMenu()->GetSubMenu(0)->DeleteMenu(2,MF_BYPOSITION);
```

将这行代码添加到 Menu2 程序中 CMainFrame 类的 OnCreate()函数的最后、return 语句之前。然后编译并运行 Menu2 应用程序，此时打开“文件”子菜单，将会发现没有“打开”菜单命令了。

例如，仍在前两段例子代码的后面追加如下代码：

```
GetMenu()->GetSubMenu(2)->RemoveMenu(3,MF_BYPOSITION);
GetMenu()->GetSubMenu(2)->RemoveMenu(2,MF_BYPOSITION);
```

运行结果是“星期”菜单和“月份”菜单的后两项均被移除了。

3.3.4 插入菜单的命令响应

前面，对菜单命令进行的命令响应都是针对静态创建的菜单命令，即在程序的菜单资源中已经存在的菜单命令标识，利用 ClassWizard 为某个类添加该菜单命令的响应函数；但对于动态添加的菜单命令，在程序运行之前，是不知道它们的标识的。在 ClassWizard 找不到它们的 ID，那么应该如何为它们添加命令响应函数？

下面以 Menu2 应用程序添加的“星期”子菜单下的“星期六”菜单命令为例，来看看如何为动态添加的菜单命令添加命令消息响应函数。首先应该为这个菜单命令创建一个菜单资源 ID。在 Menu2 工程的 FileView 选项卡上，打开“Header Files”目录，将会看到其中有一个 Resource.h 文件，这个文件中定义了程序当前使用的一些资源的 ID，可以手动在其中为“星期六”菜单命令添加一个新的 ID。例如：

```
#define IDM_SATURDAY 111
```

这样，Menu2 程序就有了一个名为 IDM_SATURDAY 的 ID，在程序中可以为“星期六”菜单命令使用这个 ID 了，即把以前程序中的代码中添加星期六菜单命令的那条 AppendMenu 语句，修改为下面这行代码：

```
Menu.AppendMenu(MF_STRING,IDM_SATURDAY, “星期六”);
```

下面为这个菜单命令添加命令响应函数了。这种情况下的命令响应一样要遵循 MFC 的消息映射机制，即需要添加 3 处代码来实现命令消息的响应。因此，可以按照以下 3 步来完成“星期六”菜单命令消息的响应。

（1）在响应这个菜单命令的程序类（本书中是 CMainFrame 类）的头文件中添加响应函数原型。添加的位置应该是在声明消息映射宏（DECLARE _MESSAGE_MAP）之前、两个 AFX_MSG 注释宏之后，也就是说，结果应该如例 3.16 所示代码中加粗显示的代码。

例 3.16

```
//{{AFX_MSG(CMainFrame)
afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
//NOTE – the ClassWizard will add and remove member functions here.
//    DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG
afx_msg void OnSaturday()
DECLARE_MESSAGE_MAP()
```

可以遵循 MFC 的命名习惯，在菜单命令名称的前面加上 On 来命名该菜单命令响应函数，因此，这里“星期六”菜单命令的响应函数名称可以命名为 OnSaturday。当然，这只是一种命名习惯，并不是规定函数一定要这样来命名，读者也可以根据自己的个人习惯来命令处理函数。

（2）在响应这个菜单命令的程序类的源文件中的消息映射表中添加消息映射。添加位置应该是在 BEGIN_MESSAGE_MAP 和 END_MESSAGE_MAP 宏之间，在两个 AFX_MSG_MAP 注释宏之后，并且菜单命令消息的映射宏是 ON_COMMAND。因此，添加之后的代码如例 3.17 所示。

例 3.17

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
//{{AFX_MSG_MAP(CMainFrame)
    //NOTE – the ClassWizard will add remove mapping macros here.
    //   DO NOT EDIT what you see in these blocks of generated code!
ON_WM_CREATE()
//}}AFX_MSG_MAP
ON_COMMAND(IDM_SATURDAY,OnSaturday)
END_MESSAGE_MAP()
```

当手动在消息映射表中添加消息映射时，一定要注意，在代码后不能加分号。

（3）实现菜单命令消息响应函数的定义体。可以在 CMainFrame 类的源文件中添加如例 3.18 所示的代码。

例 3.18

```
void CMainFrmae::OnSaturday()
{
    MessageBox("星期六");
}
```

在添加完代码后，编译并运行 Menu2 程序，打开“星期”菜单，将会发现“星期六”菜单命令现在可以使用了。选择“星期六”菜单命令，即可弹出一个消息提示对话框，其上面显示文字：星期六。

至此，就完成了为动态添加的菜单命令添加命令响应函数的功能。如果是初入门者，可能会觉得做这项工作有点困难。这里，笔者提供一个小技巧，可以利用 ClassWizard 对程序中某个已有的静态菜单命令添加命令消息响应，然后模仿 ClassWizard 在程序中为其添加的内容来完成为动态菜单添加命令响应函数这一任务。

3.3.5　修改菜单

1．CheckMenuItem()函数

```
UINT CheckMenuItem( UINT nIDCheckItem, UINT nCheck );
```

复选菜单命令，也就是将菜单命令打勾或去掉勾。

参数 nIDCheckItem 指定是哪个菜单命令，这个参数受制于第二个参数 nCheck，如果第二个参数 nCheck 为 MF_BYCOMMAND，第一个参数使用的是菜单命令的 ID 值，如 ID_FILE_NEW；如果第二个参数 nCheck 为 MF_BYPOSITION，第一个参数使用的是菜单命令在弹出式菜单中的位置，如 0、1 等。

第二个参数 nCheck 还有两个可选值：MF_CHECKED 表示复选参数 nIDCheckItem 指定的菜单命令；MF_UNCHECKED 表示不复选参数 nIDCheckItem 指定的菜单命令。

以 MFC 单文档应用程序中，AppWizard 生成的标准菜单为例，如果在 CMainFrame 类的 OnCreate() 函数中输入如例 3.19 所示的代码。

例 3.19

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    …
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_wndToolBar);
```

```
    GetMenu()->GetSubMenu(0)->CheckMenuItem(2,MF_BYPOSITION|MF_CHECKED);
    return 0;
}
```

那么会看到“文件”菜单中的“保存”菜单命令被复选了，如图 3.24 所示。

2. SetDefaultItem() 函数

```
BOOL SetDefaultItem( UINT uItem, BOOL fByPos = FALSE );
```

设置某个菜单命令为默认菜单。

参数 uItem 指定要设置为默认菜单的菜单命令的位置索引或 ID 号。若参数 fByPos 为 False，则第一个参数 uItem 指定菜单的 ID 号；若参数 fByPos 为 True，则第一个参数 uItem 指定菜单的位置。函数中默认第二个参数为 False。

还是以 MFC 单文档应用程序中，AppWizard 生成的标准菜单为例，如果在 CMainFrame 类的 OnCreate()函数中再输入如下代码：

```
GetMenu()->GetSubMenu(0)->SetDefaultItem(ID_FILE_NEW,false);
```

编译并运行程序，会发现“文件”菜单中的“新建”菜单命令被设置为默认菜单，呈加重黑体样式，如图 3.25 所示。

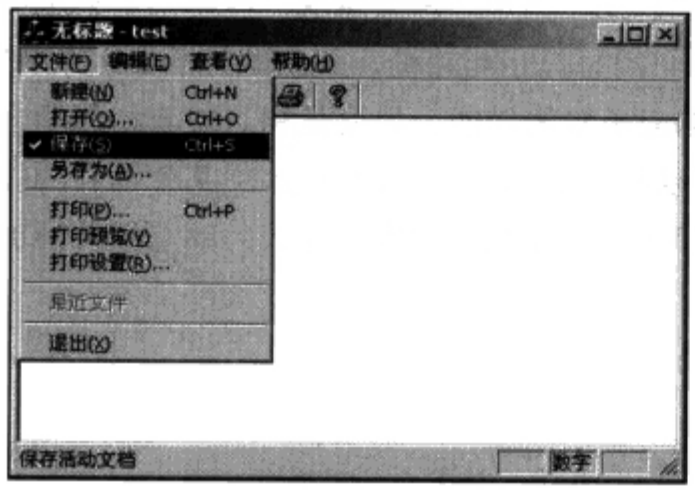


图 3.24 “保存”菜单命令被复选

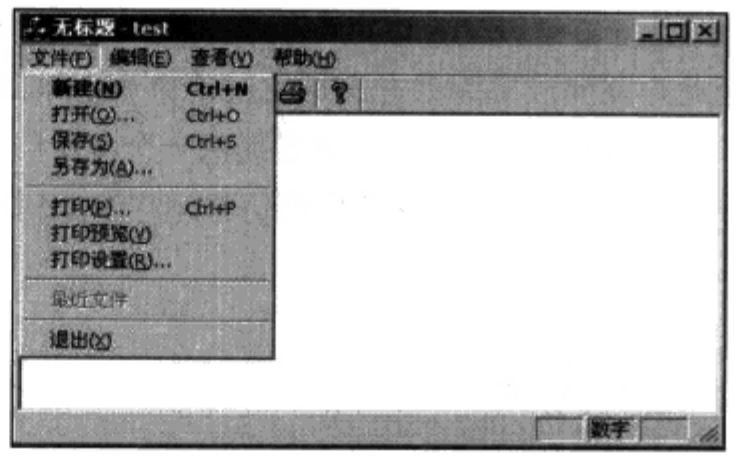


图 3.25 呈加重黑体样式的菜单项

3. EnableMenuItem()函数

```
UINT EnableMenuItem( UINT nIDEnableItem, UINT nEnable );
```

使某一菜单命令可用或不可用，或呈灰色状态。

参数 nIDEnableItem 指定是哪个菜单命令，这个参数受制于第二个参数 nEnable，如果第二个参数 nEnable 为 MF_BYCOMMAND，第一个参数使用的是菜单命令的 ID 值，如 ID_FILE_NEW；如果第二个参数 nEnable 为 MF_BYPOSITION，第一个参数使用的是菜单命令在弹出式菜单中的位置，如 0、1 等。

第二个参数 nEnable 还有 3 个可选值：MF_DISABLED 表示使参数 nIDEnableItem 指定的菜单命令不可用；MF_ENABLED 表示使参数 nIDCheckItem 指定的菜单命令可用；MF_GRAYED 表示使参数 nIDCheckItem 指定的菜单命令变灰色状态。

还是以 MFC 单文档应用程序中，AppWizard 生成的标准菜单为例，如果在 CMainFrame 类的 OnCreate()函数中输入如下代码：

```
GetMenu()->GetSubMenu(2)->EnableMenuItem(1,MF_DISABLED|MF_BYPOSITION|MF_GRAYED);
```

运行程序，会发现上面的代码没有生效。应该在 CMainFrame 的构造函数中将成员变量 m_bAutoMenuEnable 设为 False。当这个成员变量设置为 True 时，如果菜单命令没有 ON_UPDATE_

COMMAND_UI 或者 ON_COMMAND 的处理函数，用户单击弹出菜单时系统自动将菜单命令设为无效。如果菜单命令有 ON_COMMAND 的处理函数，菜单命令自动被设置为有效。

将 m_bAutoMenuEnable 成员变量在构造函数中设为 False 之后，再运行程序，那么“查看”菜单的“状态栏”菜单命令不可用了，如图 3.26 所示。

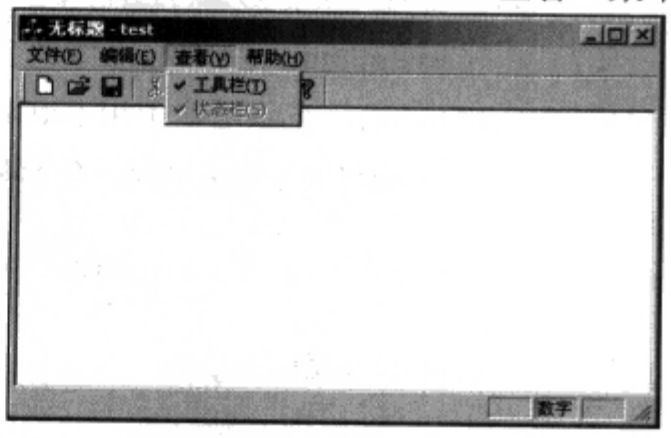


图 3.26　使菜单项不可用的运行结果

4．SetMenuItemBitmaps()函数

```
BOOL SetMenuItemBitmaps(UINT nPosition,UINT nFlags,
const CBitmap*pBmpUnchecked,const CBitmap* pBmp Checked );
```

设置图形复选菜单，也就是在某个菜单命令的左侧添加一个图标。

参数 nPosition 为选取的菜单命令的位置索引或 ID 号；参数 nFlags 为 MF_BYCOMMAND 时，第一个参数使用的是菜单命令的 ID 值，如 ID_FILE_NEW；参数 nFlags 为 MF_BYPOSITION 时，第一个参数使用的是菜单命令在弹出式菜单中的位置，如 0、1 等；第三个参数 pBmpUnchecked 表示该菜单命令没被选中时的位图信息；第四个参数 pBmpChecked 表示该菜单命令被选中时的位图信息。

实现图形复选菜单的步骤如下。

（1）在 CMainFrame 中增加两个成员变量，用于装入两个位图，一个是菜单被选中时的图标，一个是未被选中时的，分别为 CBitmap bmp1;和 CBitmap bmp2。

（2）在资源面板中增加两个位图资源，其 ID 为 IDB_BITMAP1、IDB_BITMAP2，也可以从其他文件夹中导入两个位图。

（3）在 CMainFrame 的 OnCreate()函数中加入如下代码：

```
bmp1.LoadBitmap(IDB_BITMAP1);
bmp2.LoadBitmap(IDB_BITMAP2);
GetMenu()->GetSubMenu(1)->SetMenuItemBitmaps(0,MF_BYPOSITION,&bmp1,&bmp2);
```

上面这段代码是设置“编辑”菜单的“撤销”菜单命令为图形复选菜单。

（4）此时位图有可能在菜单中不能完全显示，这是因为自己加的位图的大小不一定和菜单用的图标的大小吻合，需要用 GetSystemMetrics()函数来获取标准菜单图标的大小。下面的代码可以获取图标信息，得到图标大小信息后，可以将该段代码删除：

```
CString str;
str.Format("%d,%d",GetSystemMetrics(SM_CXMENUCHECK),
GetSystemMetrics(SM_CYMENUCHECK));
MessageBox(str);
```

按照取得的标准图标大小（13, 13）修改位图。

（5）在 CMainFrame 中再定义一个成员变量 int flag，该变量用来表示“撤销”菜单是否被选中，如果被选中，则 flag 变量为 1，否则为 0。因此，在 CmainFrame 的构造函数中需将该变量初始化为 0：

```
CMainFrame::CMainFrame()
{
    // TODO: add member initialization code here
    m_bAutoMenuEnable=FALSE;
    flag=0;
}
```

（6）在 CMainFrame 中添加“撤销”菜单命令的 COMMAND 型的消息处理函数 OnEditUndo()，当“撤销”菜单命令被选中时，调用该函数，因此在 OnEditUndo()函数中，将 flag 变量设为 1：

```
void CMainFrame::OnEditUndo()
```

```
{
    // TODO: Add your command handler code here
    flag=1;
}
```

（7）在 CMainFrame 中添加“撤销”菜单命令的 UPDATE_COMMAND_UI 型的消息处理函数 OnUpdateEditUndo()，在该函数中用 SetCheck()函数来实现改变菜单图标的操作。

```
void CMainFrame::OnUpdateEditUndo(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(flag==1)
      pCmdUI->SetCheck();
    else
      pCmdUI->SetCheck(0);
}
```

3.4 小结

本章主要介绍菜单编程方面的内容，可以分为两大部分：静态的菜单操作和动态的菜单操作。其中，静态的菜单操作主要包括：创建菜单及菜单命令，创建菜单热键，标记菜单，在菜单中加入图标，禁用菜单，移除和加载菜单等功能的实现原理及具体实现方法。动态菜单的操作主要包括：如何让程序在运行时产生新的子菜单和菜单命令，即添加菜单，插入菜单，删除菜单，修改菜单，以及如何手动为这些新产生的菜单命令添加消息响应处理函数。另外，本章还讲解了菜单消息的传输机制。在完成本章的学习后，读者应该：

（1）熟练掌握创建菜单及菜单命令、菜单热键的原理和方法；

（2）熟练掌握标记菜单及在菜单中加入图标的原理和方法；

（3）学会启用并禁用菜单；

（4）了解移除和加载菜单的原理和方法；

（5）掌握 Windows 中消息的分类，以及菜单消息的传输机制，具体的传输路由；

（6）熟练掌握动态菜单操作：包括添加菜单，插入菜单，删除菜单，插入菜单的命令响应及修改菜单。

第 4 章　Visual C++ 2010 MFC 对话框编程

Windows 应用程序工作的基本流程是从用户那里得到数据，经过相应的处理之后，再把计算结果输出到屏幕、打印机或者其他输出设备上。应用程序是如何从用户那里得到数据并且再将修改后的数据显示给用户呢？这就是通过 Windows 应用程序中一个很重要的界面对象——对话框来实现的。对话框（Dialog）实际上是一个窗口。在 MFC 中，对话框的功能被封装在了 CDialog 类中，CDialog 类是 CWnd 类的派生类。MFC 用来创建对话框的类除了 CDialog 类外，还有 CPropertySheet（属性单）和 CPropertyPage（属性页），后两者是创建属性页和向导对话框的基础，本章将对这几种对话框及其创建进行详细的介绍。

4.1　对话框简介

对话框是一个真正的窗口，它不仅可以接收消息，而且还可以被移动和关闭，甚至可以在其客户区中进行绘图。对话框也可以被看成一个大容器，在其中能够放置各种各样的标准控件及扩展控件，使程序支持用户输入的手段更加丰富。

4.1.1　对话框的控件简介

由于所有的控件类都是由 CWnd 类派生来的，因此控件实际上也是窗口。控件通常是作为对话框的子窗口而创建的，也可以出现在视图窗口、工具栏和状态栏中。控件是独立的小部件，在对话框与用户的交互过程中，扮演着重要角色。控件的种类很多，图 4.1 显示了对话框中的一些基本控件。

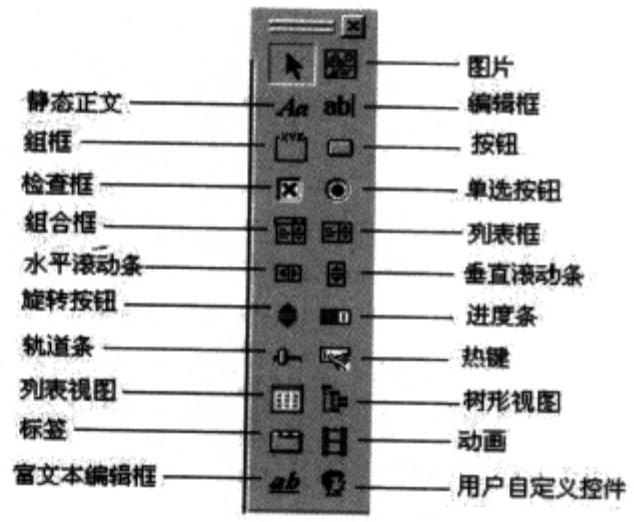


图 4.1　对话框中的基本控件

MFC 的控件类封装了控件的功能，表 4.1 列出了一些常用的控件及其对应的控件类。

表 4.1　常用控件及其对应的控件类

控　　件	功　　能	对应控件类
静态正文（Static Text）	显示正文，一般不能接受输入信息	CStatic
图片（Picture）	显式位图、图标、方框和图元文件，一般不能接受输入信息	CStatic
编辑框（Edit Box）	输入并编辑正文，支持单行和多行编辑	CEdit
命令按钮（PushButton）	响应用户的输入，触发相应的事件	CButton
复选框（Check Box）	用作选择标记，可以有选中、不选中和不确定 3 种状态	CButton
单选按钮（Radio Button）	用来从两个或多个选项中选中一项	CButton
组框（Group Box）	显示正文和方框，主要用来将相关的一些控件聚成一组	CButton
列表框（List Box）	显示一个列表，用户可以从该列表中选择一项或多项	CListBox
组合框（Combo Box）	是一个编辑框和一个列表框的组合，分为简易式、下拉式和下拉列表式	CComboBox
滚动条（Scroll Bar）	主要用来从一个预定义范围值中迅速而有效地选取一个整数值	CScrollBar

下面对这几种常用控件进行介绍。

1．静态正文控件

静态正文控件有时也称静态文本框，可用于显示文本（Text）、矩形（Rectangle）、图标（Icon）、光标（Cursor）、位图（Bitmap）及元文件（Metafile）等。静态正文控件是一种单向交互的控件，只能支持应用程序的输出，而不能接受用户的输入。

静态正文控件主要起说明和装饰作用。MFC 的 CStatic 类封装了静态正文控件。CStatic 类的成员函数 Create()负责创建静态正文控件，函数原型如下：

```
BOOL Create( LPCTSTR lpszText, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID = 0xffff );
```

该函数如果创建成功，返回 True，否则返回 False。

第一个参数 lpszText 指定了控件显示的正文。

第二个参数 dwStyle 指定了静态正文控件的风格，可以设为下列值的组合：

WS_CHILD	指定该控件是一个子窗口
WS_VISIBLE	指定该控件可显示
WS_DISABLED	指定该控件不可显示
SS_BLACKFRAME	指定一个具有与窗口边界同色的框（默认为黑色）
SS_BLACKRECT	指定一个具有与窗口边界同色的实矩形（默认为黑色）
SS_GRAYFRAME	指定一个具有与屏幕背景同色的边框
SS_GRAYRECT	指定一个具有与屏幕背景同色的实矩形
SS_LEFT	左对齐正文，正文能回绕
SS_CENTER	使显示的正文居中对齐，正文可以回绕
SS_LEFTNOWORDWRAP	左对齐正文，正文不能回绕
SS_NOTIFY	使控件能向父窗口发送鼠标事件消息
SS_RIGHT	右对齐正文，可以回绕
SS_WHITEFRAME	指定一个具有与窗口背景同色的框（默认为白色）
SS_WHITERECT	指定一个具有与窗口背景同色的实心矩形（默认为白色）

若要在静态文本框中显示光标、图标、位图或元文件，还可以设置为下列值：

SS_BITMAP	使控件显示一个在资源中定义的位图
SS_ICON	使控件显示一个在资源中定义的图标
SS_ENHMETAFILE	使控件显示一个放大的元文件
SS_CENTERIMAGE	使位图显示在静态正文控件的中间

第三个参数 rect 是一个对 RECT 或 CRect 结构的引用，用来说明控件的位置和尺寸。

第四个参数 pParentWnd 指向父窗口，不能为 NULL。

第五个参数 nID 则说明了控件的 ID。

例如：

```
CStatic myStatic;
myStatic.Create(_T("my static"), WS_CHILD|WS_VISIBLE|SS_CENTER,
  CRect(10,10,150,50),this);
```

CStatic 类常用的成员函数介绍如下。

- SetBitmap()：设置要在静态控件上显示的位图。
- SetIcon()：设置要在静态控件上显示的图标。
- SetCursor()：设置要在静态控件上显示的光标。
- SetEnhMetaFile()：设置要在静态控件上显示的元文件。

2．图片控件

图片控件也是一种静态控件，也由 CStatic 类管理。该控件提供一种方便的显示图形的方法，只要

在其属性面板中选择要显示的图形类型和要显示的图形就可以直接显示出来，而不需要编写任何代码。

3．编辑框控件

编辑框控件可以接受用户的输入和编辑。在编辑框中，用户可以输入文字、数字，并能用剪切、粘贴、复制、删除等操作来编辑输入的内容，编辑框既可以是单行的，也可以是多行的。当该控件获得焦点时，在其编辑框的内部会显示一个闪烁的光标，类似于编辑 Word 文档时显示的光标，但是它绝不能像 Word 一样可以允许用户插入表格和图片等，编辑框控件只接受纯文本形式的输入。

MFC 的 CEdit 类封装了编辑框控件。CEdit 类的成员函数 Create()负责创建编辑框控件，该函数原型如下：

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

该函数如果创建成功，返回 True，否则返回 False。

第一个参数 dwStyle 指定了编辑框控件的风格，dwStyle 可以为下列值的组合：

WS_CHILD	指定该控件是一个子窗口
WS_VISIBLE	指定该控件可显示
WS_DISABLED	指定该控件不可显示
WS_GROUP	指定该类控件作为一组
WS_TABSTOP	使控件具有 Tabstop 属性
WS_BORDER	使控件带边框
ES_AUTOHSCROLL	当用户在行尾输入一个字符时，正文将自动向右滚动 10 个字符，当用户按【Enter】键时，正文总是滚向左边
ES_AUTOVSCROLL	当用户在最后一个可见行按【Enter】键时，正文向上滚动一页
ES_CENTER	在多行编辑框中使正文居中
ES_LEFT	左对齐正文
ES_RIGHT	右对齐正文
ES_LOWERCASE	把用户输入的字母统一转换成小写字母
ES_UPPERCASE	把用户输入的字母统一转换成大写字母
ES_MULTILINE	指定一个多行编辑器
ES_PASSWORD	使所有输入的字符都用"*"来显示
ES_READONLY	将编辑框设置为只读的
ES_WANTRETURN	使多行编辑器接受【Enter】键输入并换行。如果不指定该风格，按【Enter】键会选择默认的命令按钮，这往往会导致对话框的关闭

下面给出常用的两种方案：如果想要创建一个普通的单行编辑框，应指定风格为 WS_CHILD|WS_VISIBLE|WS_TABSTOP |WS_BORDER|ES_LEFT|ES_AUTOHSCROLL，这将创建一个带边框、左对齐正文、可水平滚动的单行编辑器；如果要创建一个普通多行编辑框，还要附加 ES_MULTILINE|ES_WANTRETURN|ES_AUTOVSCROLL |WS_HSCROLL| WS_VSCROLL 风格，这将创建一个可水平和垂直滚动的、带有水平和垂直滚动条的多行编辑器。

第二个参数 rect 是一个对 RECT 或 CRect 结构的引用，指定了编辑框的位置和尺寸。

第三个参数 pParentWnd 指定了父窗口，不能为 NULL。

第四参数 nID 指定了编辑框的 ID。

例如：

```
CEdit* pEdit = new CEdit;
pEdit->Create(ES_MULTILINE | WS_CHILD | WS_VISIBLE | WS_TABSTOP | WS_BORDER, CRect(10, 10, 100, 100), this, 1);
```

CEdit 类常用的成员函数介绍如下。

- GetSel()：获得编辑框控件中当前选择的其始字符和终止字符的位置。

- SetSel(): 选择编辑框控件中的一个字符范围。
- GetLineCount(): 获得多行编辑框的行数。
- Undo()/Clear()/Copy()/Cut()/Paste(): 剪贴板操作，分别用于撤销上一次输入，清除编辑框中被选中的正文，把在编辑框中选择的正文复制到剪贴板中，清除编辑框中被选中的正文并把这些正文复制到剪贴板中，将剪贴板中的正文插入到编辑框的当前插入符处。

4. 按钮控件

通过表 4.1 中的内容就可以知道，在 Windows 系统中，命令按钮、复选框、单选按钮和组框都是由 MFC 类 CButton 管理的，可统称为按钮控件。按钮是指可以响应鼠标单击的小的矩形子窗口，它可以向父窗口发出两种控件通知消息：BN_CLICKED（在按钮上单击）和 BN_DOUBLECLICKED（在按钮上双击）。

按钮控件包括命令按钮（PushButton）、复选框（CheckBox）、单选按钮（RadioButton）、组框（GroupBox）和自绘式按钮（Owner-draw Button）。命令按钮的作用是对用户的鼠标单击做出响应并触发相应的事件，在按钮中既可以显示正文，也可以显示位图。复选框控件作为一种选择标记，可以有选中、不选中和不确定 3 种状态。单选按钮控件一般都是成组出现的，具有互斥的性质，即同组单选按钮中只能有一个是被选中的。组框用来将相关的一些控件聚成一组。自绘式按钮是指由程序而不是系统负责重绘的按钮。

如何在一个 CButton 类中区分这几种不同的按钮控件呢？这就要通过按钮风格（Button Style）。关于按钮风格，将在讲述该类的 Create()函数时进行介绍。

MFC 的 CButton 类封装了按钮控件。CButton 类的成员函数 Create()负责创建按钮控件，函数原型如下：

```
BOOL Create( LPCTSTR lpszCaption, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID );
```

该函数如果创建成功，返回 True，否则返回 False。

第一个参数 lpszCaption 指定了按钮显示的正文。

第二个参数 dwStyle 指定了按钮的风格。dwStyle 可以是下列值的组合：

WS_CHILD	指定该控件是一个子窗口
WS_VISIBLE	指定该控件可显示
WS_DISABLED	指定该控件不可显示
WS_GROUP	指定该类控件作为一组
WS_TABSTOP	使控件具有 Tabstop 属性
BS_CHECKBOX	指定该按钮控件是一个复选框控件
BS_PUSHBUTTON	指定该按钮控件是一个普通命令按钮控件
BS_RADIOBUTTON	指定该控件是一个单选按钮控件，在按钮的右侧显示正文
BS_GROUPBOX	指定该按钮控件是一个组框控件
BS_OWNERDRAW	指定该按钮控件是一个自绘式按钮控件
BS_AUTOCHECKBOX	同 BS_CHECKBOX，不过单击时按钮会自动反转
BS_3STATE	同 BS_CHECKBOX，不过控件有 3 种状态：选择、未选和变灰
BS_AUTO3STATE	同 BS_3STATE，不过单击按钮时会改变状态
BS_AUTORADIOBUTTON	同 BS_RADIOBUTTON，不过单击时按钮会自动反转
BS_DEFPUSHBUTTON	指定默认的命令按钮，这种按钮的周围有一个黑框，用户可以按【Enter】键来快速选择该按钮
BS_LEFTTEXT	使控件的标题显示在按钮的左侧

下面给出常用的几种方案：如果想要创建一个普通的命令按钮，应指定风格为 WS_CHILD|WS_VISIBLE|WS_TABSTOP；若要创建一个普通的复选框，应指定风格为 WS_CHILD|WS_VISIBLE|WS_

TABSTOP| BS_AUTOCHECKBOX；若要创建组中第一个单选按钮，应指定风格为 WS_CHILD | WS_VISIBLE | WS_TABSTOP | WS_GROUP| BS_AUTORADIOBUTTON，组中其他单选按钮指定风格，则不应该包括 WS_TABSTOP 和 WS_GROUP。

第三个参数 rect 是一个对 RECT 或 CRect 结构的引用，rect 说明了按钮的位置和尺寸。

第四个参数 pParentWnd 指向父窗口，该参数不能为 NULL。

第五个参数 nID 是按钮的 ID。

例如：

```
CButton myButton1, myButton2, myButton3;
// 创建一个普通命令按钮
myButton1.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
 CRect(10,10,100,30), this, 1);
// 创建一个单选按钮
myButton2.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_RADIOBUTTON,
CRect(10,40,100,70), this, 2);
// 创建一个复选框
myButton3.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_AUTOCHECKBOX,
CRect(10,100,100,130), this, 4);
```

MFC 还提供了 CButton 的派生类 CBitmapButton。利用该类可以创建一个拥有 4 幅位图的命令按钮，按钮在不同状态时会显示不同的位图，这样可以使界面显得生动、活泼。

CButton 类常用的成员函数介绍如下。

- GetState()：获得按钮控件的状态。
- SetState()：设置按钮控件的状态。
- SetCheck()/GetCheck()：设置/获得按钮控件的选择状态。
- SetButtonStyle()/ GetButtonStyle()：设置/获得按钮控件的风格。
- SetBitmap()/GetBitmap()：设置/获得与控件相关联的位图。

5．列表框控件

列表框控件是一个选项列表，它允许用户从所列出的选项中进行单项或多项选择，被选择的项呈高亮度显示。列表框具有边框，并且一般带有一个垂直滚动条。列表框分单选列表框和多重选择列表框两种。单选列表框一次只能选择一个列表项，而多重选择列表框可以进行多重选择。它可以向父窗口发送下面几种消息：

LBN_DBLCLK	用户双击了一个列表项。只有具有 LBS_NOTIFY 的列表框才能发送该消息
LBN_SELCHANGE	单击选择了一个列表项。只有具有 LBS_NOTIFY 的列表框才能发送该消息
LBN_SELCANCEL	当前的选择被取消。只有具有 LBS_NOTIFY 的列表框才能发送该消息
LBN_SETFOCUS	列表框获得输入焦点
WM_CHARTOITEM	当列表框收到 WM_CHAR 消息后，向父窗口发送该消息。只有具有 LBS_WANTKEYBOARDINPUT 风格的列表框才会发送该消息
WM_VKEYTOITEM	当列表框收到 WM_KEYDOWN 消息后，向父窗口发送该消息。只有具有 LBS_WANTKEYBOARDINPUT 风格的列表框才会发送该消息

MFC 的 CListBox 类封装了列表框。CListBox 类的 Create()成员函数负责列表框的创建，该函数原型如下：

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

如果创建成功，返回 True，否则返回 False。

第一个参数 dwStyle 指定了列表框控件的风格，dwStyle 可以是下列值的组合：

WS_CHILD	指定该控件是一个子窗口

WS_VISIBLE	指定该控件可显示
WS_DISABLED	指定该控件不可显示
WS_GROUP	指定该类控件作为一组
WS_TABSTOP	使控件具有 Tabstop 属性
WS_VSCROLL	使该控件具有水平滚动条
WS_HSCROLL	使该控件具有垂直滚动条
WS_BORDER	使控件带边框
LBS_EXTENDEDSEL	支持多重选择。在单击列表项时按住【Shift】键或【Ctrl】键即可选择多项
LBS_MULTICOLUMN	指定一个水平滚动的多列列表框，通过调用 CListBox::SetColumnWidth()函数来设置每列的宽度
LBS_MULTIPLESEL	支持多重选择。列表项的选择状态随着用户对该项单击或双击而翻转
LBS_NOREDRAW	当选择发生变化时防止列表框被更新，可发送 WM_SETREDRAW 来改变该风格
LBS_NOTIFY	当用户单击或双击时通知父窗口
LBS_SORT	使插入列表框中的选项按升序排列
LBS_STANDARD	相当于多重指定了 WS_BORDER|WS_VSCROLL|LBS_SORT |LBS_ NOTIFY
LBS_WANTKEYBOARDINPUT	允许列表框的父窗口接收 WM_VKEYTOITEM 和 WM_ CHARTOITEM 消息，以响应键盘输入

下面给出几种常用的方案：如果想要创建一个普通的单选择列表框，应指定的风格为 WS_CHILD|WS_VISIBLE|WS_TABSTOP|LBS_STANDARD；如果要创建一个多重选择列表框，应该在单选择列表框风格的基础上再加上 LBS_MULTIPLESEL 或 LBS_ EXTENDEDSEL；如果不希望列表框排序，就不能使用 LBS_STANDARD 风格。

第二个参数 rect 是一个对 RECT 或 CRect 结构的引用，说明了控件的位置和尺寸。

第三个参数 pParentWnd 指向父窗口，该参数不能为 NULL。

第四个参数 nID 说明了控件的 ID。

例如：

```
CListBox* pmyListBox;
pmyListBox->Create(WS_CHILD|WS_VISIBLE|LBS_STANDARD|WS_HSCROLL,
CRect(10,10,200,200), this, 1);
```

CListBox 类常用的成员函数介绍如下。

- AddString()：向列表框中增加一个字符串选项，增加的位置在最后。
- InsertString()：根据参数所给的位置，在列表框中插入一个字符串选项。
- ResetString()：删除列表框中所有的选项。
- Deletestring()：从列表框中删除一个选项。
- GetCount()：获得列表框中选项的个数。
- SetItemData()/GetItemData()：设置/获得与一个选项相关联的 32 位整数值。
- GetSel()：判断一个选项是否被选中。
- SetCurSel()：在单选列表框中选中一个选项。
- GetCurSel()：获得一个被选中选项的选项索引。
- GetSelCount()：在一个多选列表框中得到当前被选中的选项个数。
- GetText()：根据选项索引得到相应的选项字符串。

6. 组合框控件

组合框把一个编辑框和一个单选择列表框组合在了一起，用户既可以在编辑框中输入，也可以从列表框中选择一个列表项来完成输入。组合框有简易式（Simple）、下拉式（Dropdown）和下拉列表

式（Drop List）3 种风格。简易式组合框包含一个编辑框和一个总是显示的列表框；下拉式组合框与简易式组合框类似，二者的区别在于下拉式组合框仅当单击下拉按钮后列表框才会弹出；下拉列表式组合框也有一个下拉的列表框，但它的编辑框是只读的，不能输入字符。组合框可以向父窗口发送下面几种消息

CBN_CLOSEUP	组合框的列表框组件被关闭。简易式组合框不会发出该消息
CBN_DBLCLK	用户在某列表项上双击。只有简易式组合框才会发出该消息
CBN_DROPDOWN	组合框的列表框组件下拉。简易式组合框不会发出该消息
CBN_EDITCHANGE	编辑框的内容被用户改变了。与 CBN_EDITUPDATE 不同，该消息是在编辑框显示的正文被刷新后才发出的。下拉列表式组合框不会发出该消息
CBN_EDITUPDATE	在编辑框准备显示改变了的正文时发送该消息。下拉列表式组合框不会发出该消息
CBN_SELENDCANCEL	表明用户的选择应该取消。当用户在列表框中选择了一项，然后又在组合框控件外单击时就会导致该消息的发送
CBN_SELENDOK	用户选择了一项，然后按【Enter】键或单击了下拉按钮。该消息表明用户确认了自己所做的选择
CBN_KILLFOCUS	组合框失去了输入焦点
CBN_SETFOCUS	组合框获得了输入焦点
CBN_SELCHANGE	用户通过按箭头键改变了列表项的选择

MFC 的 CComboBox 类封装了组合框。需要指出的是，虽然组合框是编辑框和列表框的组合，但是 CComboBox 类并不是 CEdit 类和 CListBox 类的派生类，而是 CWnd 类的派生类。

CComboBox 的成员函数 Create()负责创建组合框，该函数原型如下：

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

如果创建成功，该函数返回 True，否则返回 False。

第一个参数 dwStyle 指定了组合框控件的风格，dwStyle 可以是下列值的组合。

WS_CHILD	指定该控件是一个子窗口
WS_VISIBLE	指定该控件可显示
WS_DISABLED	指定该控件不可显示
WS_GROUP	指定该类控件作为一组
WS_TABSTOP	使该控件具有 Tabstop 属性
WS_VSCROLL	使该控件具有水平滚动条
WS_HSCROLL	使该控件具有垂直滚动条
CBS_AUTOHSCROLL	使编辑框组件具有水平滚动的风格
CBS_DROPDOWN	指定一个下拉式组合框
CBS_DROPDOWNLIST	指定一个下拉列表式组合框
CBS_SIIMPLE	指定一个简易式组合框
CBS_SORT	自动对列表框组件中的选项进行排序
CBS_NOINTEGRALHEIGHT	组合框的尺寸由应用程序而不是 Windows 指定。通常，由 Windows 指定尺寸会使列表项的某些部分被隐藏起来

第二个参数 rect 指定了列表框组件下拉后组合框的位置和尺寸。

第三个参数 pParentWnd 指向父窗口，该参数不能为 NULL。

第四个参数 nID 则说明了控件的 ID。

例如：

```
CComboBox* pmyComboBox;
pmyComboBox->Create(WS_CHILD|WS_VISIBLE|WS_VSCROLL|CBS_DROPDOWNLIST,CRect(10,10,200,100), this, 1);
```

CComboBox 类常用的成员函数简介如下。

- AddString(): 在列表框组件的尾部添加一个字符串选项。

- InsertString(): 根据参数提供的位置，从列表框中插入一个字符串选项。
- ResetContent(): 删除列表框中所有的选项。
- DeleteString(): 从列表框中删除一个字符串选项。
- GetCount(): 获得组合框的列表框中选项的数目。
- SetCurSel(): 在列表框中选中一个选项。
- GetCurSel(): 获得被选中的选项的索引。
- SetEditSel(): 设置组合框编辑控件中当前选择的首尾字符的位置。
- GetEditSel(): 获得组合框编辑控件中当前选择的首尾字符的位置。
- Undo()/Clear()/Copy()/Cut()/Paste(): 组合框编辑框部分的剪贴板操作，分别用于撤销上一次输入，清除编辑框中被选择的正文，把在编辑框中选择的正文复制到剪贴板中，清除编辑框中被选择的正文并把这些正文复制到剪贴板中，将剪贴板中的正文插入到编辑框的当前插入符处。
- SetItemData()/GetItemData(): 设置/获得与组合框的一个列表框选项相关联的 32 位整数值。

7. 滚动条控件

滚动条（ScrollBar）的主要作用是从某一预定义值的范围内快速有效地选取一个整数值。按照走向，滚动条分为水平滚动条和垂直滚动条两种。在滚动条内有一个滚动框，用来表示当前的值。单击滚动条，可以使滚动框移动一页或一行，也可以直接拖动滚动框。滚动条既可以作为一个独立控件存在，也可以作为窗口、列表框和组合框的一部分。

需要指出的是，从性质上划分，滚动条可分为标准滚动条和滚动条控件两种。标准滚动条是由 WS_HSCROLL 或 WS_VSCROLL 风格指定的，不是一个实际的窗口，而是窗口的一个组成部分（例如列表框中的滚动条），只能位于窗口的右侧（垂直滚动条）或底端（水平滚动条）。标准滚动条是在窗口的非客户区中创建的。与之相反，滚动条控件并不是窗口的一个组成部分，而是一个实际的窗口，可以放置在窗口客户区的任意位置，它既可以独立存在，也可以与某一个窗口组合，行使滚动窗口的职能。由于滚动条控件是一个独立的窗口，因此可以拥有输入焦点，可以响应光标控制键，如 PgUp、PgDown、Home 和 End。

MFC 的 CScrollBar 类封装了滚动条控件。CScrollBar 类的 Create()成员函数负责创建控件，该函数的原型如下：

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

如果创建成功，该函数返回 True，否则返回 False。

第一个参数 dwStyle 指定了控件的风格。

WS_CHILD	指定该控件是一个子窗口
WS_VISIBLE	指定该控件可显示
WS_DISABLED	指定该控件不可显示
WS_GROUP	指定该类控件作为一组
BS_HORZ	指定该滚动条是水平方向滚动
BS_VERT	指定该滚动条是垂直方向滚动

下面给出几种方案：如果想要创建一个普通的水平滚动条，应指定风格为 WS_CHILD|WS_VISIBLE|BS_HORZ；如果要创建一个普通的垂直滚动条，应指定风格为 WS_CHILD|WS_VISIBLE|BS_VERT。

第二个参数 rect 说明了控件的位置和尺寸。

第三个参数 pParentWnd 指向父窗口，该参数不能为 NULL。

第四个参数 nID 则说明了控件的 ID。

例如：

```
CscrollBar m_ScrollBar
m_ScrollBar.Create(SBS_HORZ | SBS_TOPALIGN | WS_CHILD | WS_VISIBLE,
CRect(5, 5, 100, 30), this, 100));
```

CScrollBar 类常用的成员函数介绍如下。

- SetRange()/GetRange(): 设置/获得滚动条的最大位置和最小位置（即范围）。
- SetPos()/GetPos(): 设置/获得滚动条的当前的位置。
- SetPageSize()/GetPageSize(): 设置/获得每个滚动页的大小。

4.1.2　对话框的种类简介

从 MFC 编程的角度来看，一个对话框由两部分组成。

（1）对话框模板资源。对话框模板用于指定对话框的控件及其分布，Windows 根据对话框模板来创建并显示对话框。

（2）对话框类。对话框类用于实现对话框的功能，由于对话框的功能各不相同，因此一般需要从 CDialog 类、CpropertySheet 类或 CpropertyPage 类派生一个新类，以完成特定的功能。

一个对话框的从有到无是离不开这两部分的。在应用程序中，创建一个对话框的前提就是完成对话框模板的设计和对话框类的生成。

对话框分为模态（Model）对话框和非模态（Modeless）对话框两种。

（1）模态对话框是指这种对话框显示时，其父窗口将暂时失效，只有处理完对话框所要求的动作后，才会将控制权交还给父窗口。在用户想要进行对话框以外的应用程序操作时，必须首先对该对话框进行响应，如单击“确定”或“取消”按钮等将该对话框关闭。例如，通过选择“File”→“Open”命令打开一个文件对话框后，再用鼠标去选择其他菜单或进行该对话框以外的任何操作时，只会听到嘟嘟声，这是因为文件对话框是一个模态对话框。模态对话框垄断了用户的输入，此时若不输入文件名或关闭对话框，Notepad 就无法继续工作。当一个模态对话框打开时，用户只能与该对话框进行交互，而其他用户界面对象收不到输入信息。平时所见的大部分对话框都是模态对话框。

虽然可以赋予模态对话框任何窗口风格，但是 Windows 建议使用 DS_MODALFRAME、WS_CAPTION 及 WS_SYSMENU。DS_MODALFRAME 风格可使模态对话框具备它独有的厚边。

当屏幕上显示模态对话框时，它将拥有自己的消息循环，因此所有的输入窗口信息都不能指定为 WS_CHILD 风格，否则父窗口失效时，子窗口（对话框自己）也会随之失效。这样一来就无法继续执行了。

（2）非模态对话框（Modeless Dialog Boxes），又称为共存式对话框。不会像模态对话框那么专断，令父窗口失效，因此当打开非模态对话框时，使用者依然可以操作其他窗口，也可以继续执行父窗口的工作。典型的例子是 Windows 提供的记事本程序中的查找对话框，查找对话框不会垄断用户的输入，打开查找对话框后，用户仍可与其他用户界面对象进行交互，可以一边查找，一边修改文章，这样就大大方便了使用。Word 应用程序中，通过“Edit”菜单中的“Find”命令启动的对话框也为非模态对话框。

大部分的非模态对话框都指定 WS_POPUP、WS_CAPTION、WS_BORDER 及 WS_SYSTEMMENU 风格。典型的非模态对话框包括系统菜单和标题。

非模态对话框从主程序的消息循环中取得输入信息。如果非模态对话框中有控件存在，而且允许

用户使用键盘去选择，那么必须在消息循环中调用 CWnd 类（或其继承类）的成员函数 IsDialogMessage() 来判断输入信息是否要送给非模态对话框，如果是，就处理它，否则按百折不回的途径来处理。

非模态对话框的建立靠 CDialog 类的成员函数 Create()来完成，关闭则用 CWnd 类的成员函数 DestroyWindow()来实现。

4.1.3 设计对话框

要想在应用程序中添加一个自己的对话框并与应用程序之间进行数据操作，最重要的就是先设计好一个对话框资源，如果没有这个对话框资源，用户要对谁进行操作呢？设计对话框一般有如下几个步骤：

- 插入一个新的对话框资源；
- 添加控件到对话框上；
- 编辑控件属性。

1. 插入一个新的对话框资源

打开工作台的资源面板，打开“Dialog”文件夹，它下面有一个 ID 号为 IDD_ABOUTBOX 的对话框，这几乎是每个 MFC 应用程序都具备的。现在要添加一个自己的对话框。

右击“Dialog”文件夹，在弹出的快捷菜单中选择“Insert Dialog”命令，这时，不仅在资源面板中多了一个 ID 号 IDD_DIALOG1 的对话框资源，而且在右侧的编辑器中也显示了这个对话框模板的编辑窗口，如图 4.2 所示。

默认的对话框模板只有“OK”和“Cancel”两个按钮，现在就开始进行对话框模板资源的设计。首先，选中该对话框资源，按【Enter】键或双击对话框或者右击对话框并选择“Properties”命令，则在窗口的最右端弹出对话框的属性面板，如图 4.3 所示。

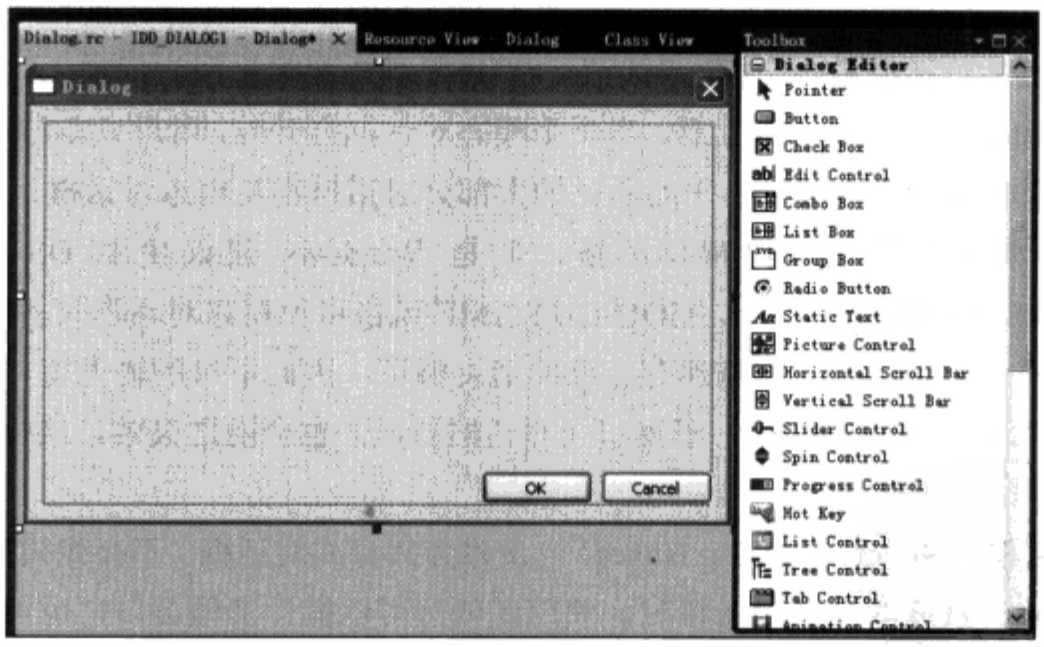


图 4.2 对话框模板编辑窗口

对话框一共有 5 个选项卡，分别为 Apperance、Behavior、Font、Misc 及 Position。其中，它们分别可以用来设置对话框的 ID 号、标题、菜单、初始位置及字体等属性，对话框其实也可以拥有自己的菜单。

第一个选项卡 Apperance，如图 4.4 所示，可以用来设置对话框窗口的风格，例如是否有标题、是否是系统菜单、最大化或最小化按钮，是否有水平或垂直滚动条等。其中有一个重要的属性 Style，它的下面提供了一个下拉列表框，其中提供了 3 个选项：Overlapped、Popup、Child。这 3 项正好对应于

3 种最基本的窗口风格——重叠窗口、弹出式窗口和子窗口，默认为弹出式窗口风格。还有一个属性 Border，也提供了一个下拉列表框，有 4 个选项，分别为无边框（没有标题栏）、细边框、粗边框可变大小、对话框边框，其中对话框边框为默认设置。

后 3 个选项卡可以给对话框编辑更详细的风格，但是，一般只给出默认的设置就足够了，若有特殊的风格要求，而又不了解那些属性设置的意义，在相应的选项卡上直接按【F1】键，在窗口右侧就会弹出每个属性设置的详细说明。

2. 添加控件到对话框上

前面说过，对话框可以看为一个容器，在其中可以放置各种各样的控件，如静态正文控件、编辑控件、按钮控件、列表框和组合框控件等，那么这些控件是如何添加到对话框上的呢？

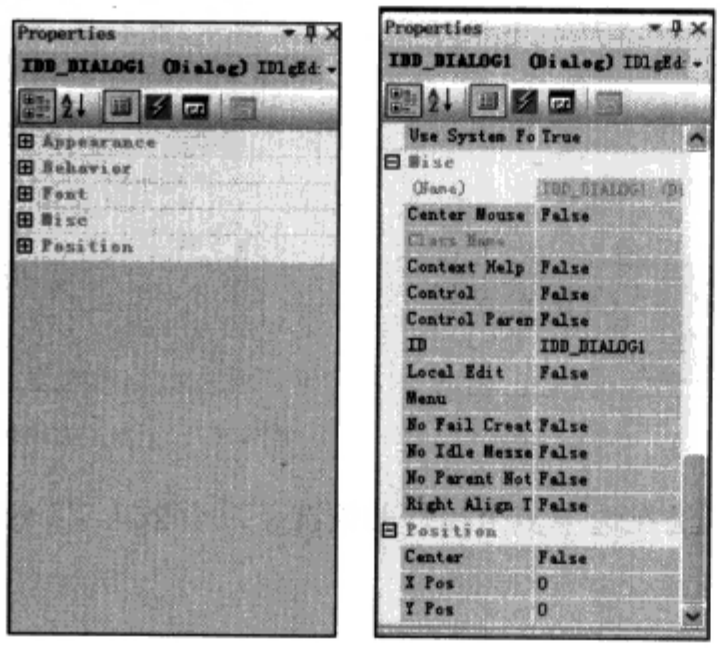


图 4.3　对话框的属性窗口

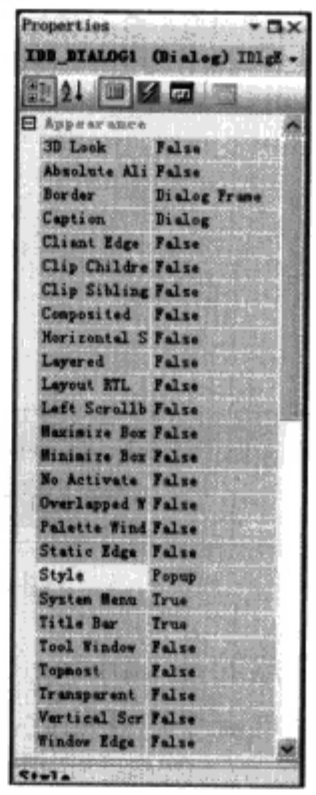


图 4.4　Apperance 选项卡

选择“View”→“ToolBox”命令，则会在对话框编辑窗口的旁边显示一个 ToolBox，如图 4.5 所示。在控件工具栏上选择一个控件，然后在对话框中单击并一直拖到自己满意的大小，然后释放鼠标，则相应的控件就被放置到了对话框模板中，或者也可以直接在控件工具栏中把一个控件拖到对话框上的指定位置然后释放鼠标。若想删除添加的控件，只需要用左键按住相应控件，并按【Delete】键即可，也可以直接在控件上右击，并选择“Delete”命令。

读者可以在对话框模板中随意添加几个控件试试。当选择对话框或控件时，在它们周围就会显示一个虚框，拖动虚框的边界就可以改变对话框或控件的大小，在屏幕下方的状态栏中会显示出所选对象的坐标和尺寸。可以拖动控件，也可以按箭头键来移动选中的控件。在拖动控件时若按住【Ctrl】键，则控件会被复制。

假设随便添加几个控件，如图 4.6 所示，为了将这几个控件堆砌，或者使它们的大小相同，位置排列整齐，可以使用 Dialog 工具栏中的相应按钮来完成，如图 4.7 所示，或通过选择菜单“Layout”中的前 7 个菜单命令来调整，注意“Layout”菜单只有在当前编辑窗口中为对话框编辑器时才会显示，如图 4.8 所示。这大大减轻了用户在排列多个控件时的苦恼，一般情况下，Dialog 工具栏显示在窗口的底部，如果窗口上没有，可以右击工具栏空白区，在弹出的快捷菜单中选择“Dialog”命令。

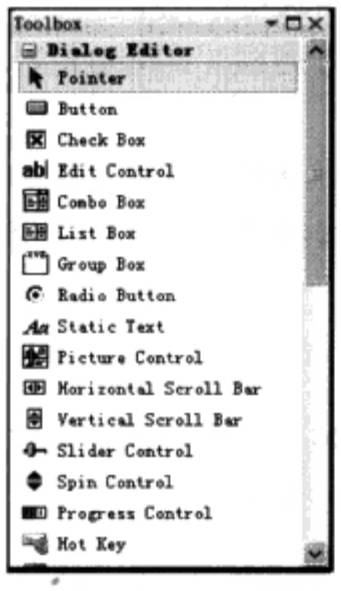


图 4.5　控件工具栏 ToolBox

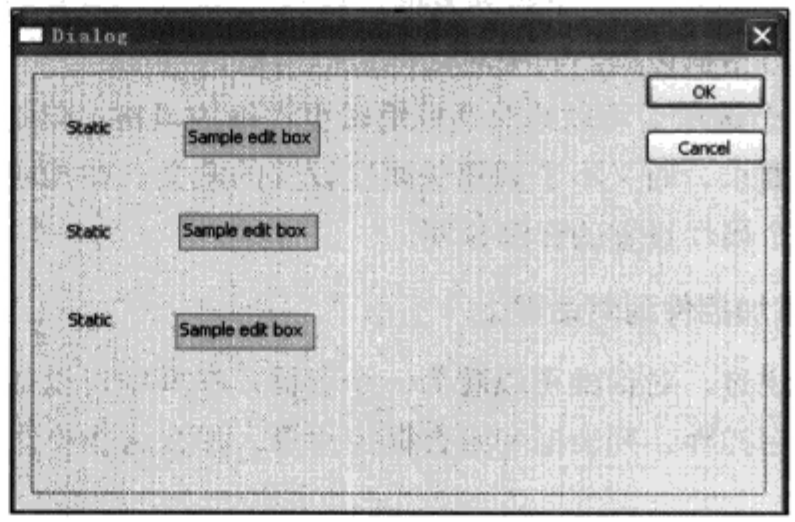


图 4.6　往对话框内添加控件

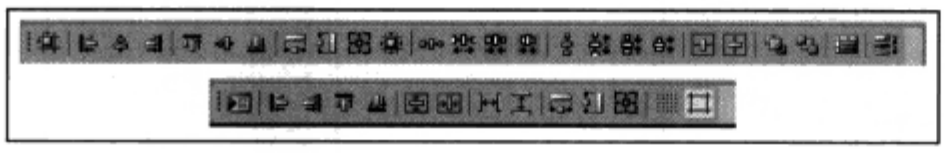


图 4.7　Dialog 工具栏按钮

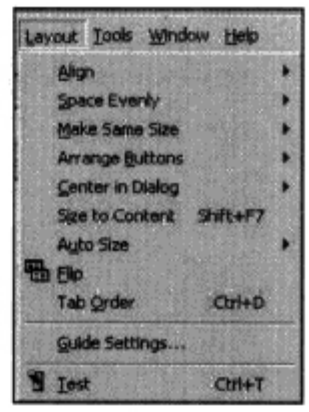


图 4.8　Layout 菜单项

需要注意的是，Dialog 工具栏中有许多按钮是在选中多个控件的情况下才有效。“Layout”菜单中的某些菜单命令也是如此。

3．编辑控件属性

选中控件或对话框后按【Enter】键，则会弹出一个属性面板，属性面板用来设置控件或对话框的各种属性。这些属性面板是标签式面板，第一页通过都是是常规属性，一个典型的控件属性面板如图 4.9 所示。如果不了解属性面板中的选项的意义，同样可以按【F1】键获得帮助。

在控件属性面板的常规属性中，有一些控件共同的属性。

- ID 属性：用于指定控件的标识符，Windows 依靠 ID 号来区分不同的控件。
- Caption（标题）属性：静态正文、组框、按钮、复选框、单选按钮等控件可以显示标题，用来对控件进行文字说明。控件标题中的字符&使紧跟其后的字符有下划线，按【Alt+下划线字符】组合键将启动该控件。若控件是一个单选按钮，则按【Alt+下划线字符】组合键将选择该按钮；若是复选框，则相当于对该复选框按空格键；若是按钮，则将激活按钮命令；若控件是一个静态正文，则将激活按 Tab 顺序紧随其后的下一个控件。
- Visible 属性：用来指定控件是否是可见的。
- Disable 属性：使控件允许或禁止，一个禁止的控件呈灰色显示，不能接收任何输入。
- Tabstop 属性：用户可以按【Tab】键移动到具有 Tabstop 属性的控件上。Tab 移动的顺序可以由用户指定。按【Ctrl+D】组合键或选择菜单“Layout”→“Tab Order”命令，则 Tab 顺序会显示出来，如图 4.10 所示，用户可以用鼠标重新指定 Tab 顺序。默认的 Tab 顺序是创建控件的先后次序。

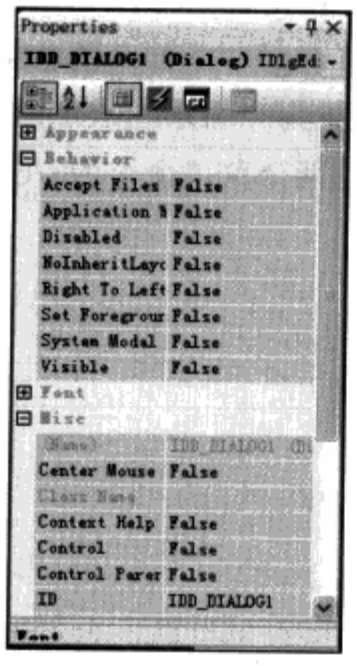


图 4.9　控件属性对话框

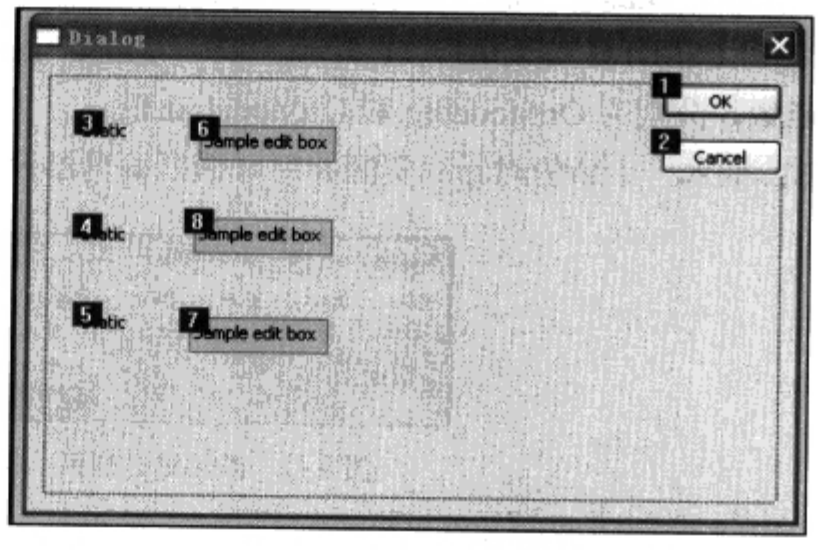


图 4.10　设定控件的 Tab 顺序

- Group 属性：用来指定几个控件为一组，用户可以通过按箭头键在该组控件内移动。同一组内的单选按钮具有互斥的特性，即在这些单选按钮中只能有一个是选中的。如果一个控件具有 Group 属性，则这个控件及按 Tab 顺序紧随其后的所有控件都属于一组，直到遇到另一个有 Group 属性的控件为止。Group 属性一般用于有多组单选按钮的情况，以便将单选按钮分组，否则默认情况下，一个对话框中所有的单选按钮都被放在一组中。

对话框的尺寸单位不是像素，而是与字体的大小有关。*X* 方向上的一个单位等于字符平均宽度的 1/4，*Y* 方向上的一个单位等于字符平均高度的 1/8。这样，随着字体的改变，对话框单位也会改变，对话框本身的总体比例保持不变。

4.2　创建与销毁对话框

不管是模态对话框还是非模态对话框，都需要拥有相应的对话框模板资源和新的对话框类，二者的创建都遵循以下步骤。

（1）添加一个新的对话框资源并用对话框编辑器设计对话框。

（2）创建新的对话框类，该类应从 CDialog 类派生。

（3）使用 ClassWizard 为对话框类加入与控件相关联的成员变量。

（4）在对话框类中添加需要的消息处理函数。

（5）创建并显示对话框。

实际上，不管是模态对话框还是非模态对话框，它们的前 4 步操作基本相同，只有最后一步稍有区别，下面将通过一个实例来讲解模态对话框和非模态对话框的创建。

4.2.1　模态对话框

建立一个 MFC 应用程序，为该应用程序新添加两个菜单命令，这两个菜单命令分别负责弹出模态对话框和非模态对话框，然而并不需要创建两个对话框，只要一个对话框就足够了。前面讲过，对于模态对话框和非模态对话框，只是在创建和显示时不同，因此只用一个对话框模板资源即可，稍后读者会明白的。

详细的创建步骤如下。

步骤 1：新建一个 MFC 单文档应用程序，工程名为 Dialog 或由用户自定义。

步骤 2：在 AppWizard 生成的标准菜单后面添加一个弹出式菜单“Dialog”，下面有两个菜单命令，分别为“Model Dialog”和“Modeless Dialog”，它们的 ID 分别为“IDM_MODEDLG”和“IDM_MODELESSDLG”，如图 4.11 所示。并在 CMainFrame 类中添加这两个菜单命令的 COMMAND 消息处理函数，分别为 OnModedlg 和 OnModelessdlg。

步骤 3：插入一个新的对话框资源并用对话框编辑器设计对话框。

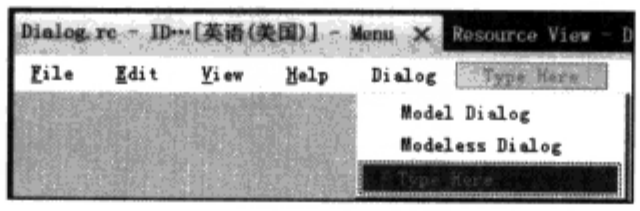


图 4.11　添加对话框菜单和菜单项

设置新插入的对话框的 ID 为 IDD_MYDLG，将对话框的标题改为“对话框实例（实现两数相加）”，对话框的属性设置如图 4.12 所示。

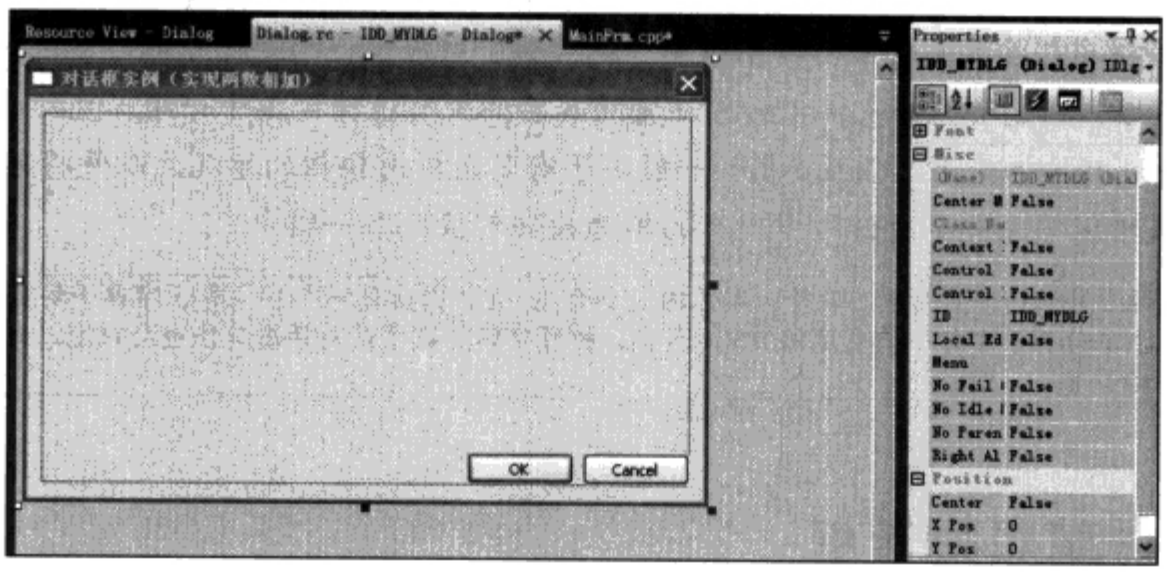


图 4.12　设置对话框的标题属性

分别在对话框上添加3个静态正文控件和3个编辑框控件。将3个静态正文控件属性面板中“Styles”选项卡中的 Align text 属性设置为 Right，意思是让静态正文框中的文本内容右对齐。然后，将 3 个静态正文控件的 Caption 属性分别设为“加数”、“被加数”和“两数之和”，不需要修改它们的 ID。

将 3 个编辑框控件的 ID 分别改为 IDC_EDIT_NUM1、IDC_EDIT_NUM2、IDC_EDIT_SUM。

将对话框上的“Cancel”按钮拖到下面，将其 Caption 属性改为“相加”。

对话框的样式如图 4.13 所示。

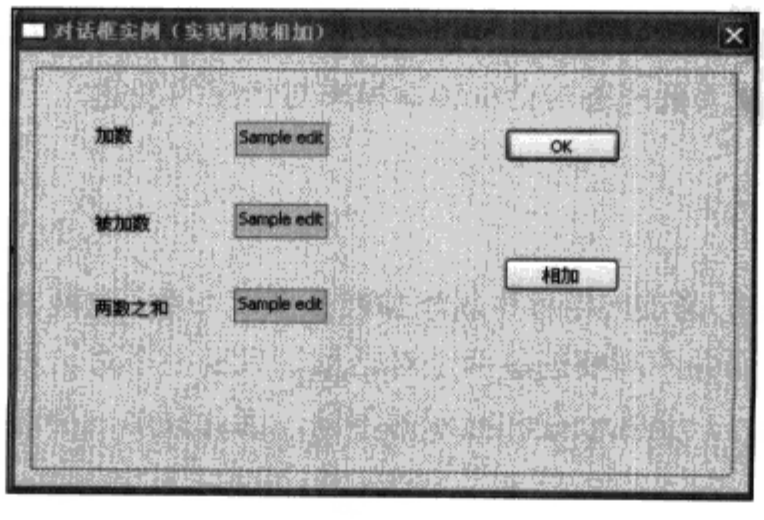


图 4.13　对话框设计界面

步骤 4：创建新的对话框类，该类应从 CDialog 类派生。

准备好对话框模板资源之后，就可以创建新的对话框类了。

利用 ClassWizard，用户可以十分方便地创建 MFC 窗口类的派生类，对话框类也不例外。请读者按以下几步进行操作：

选中 IDD_MYDLG 对话框模板，在 IDD_MYDLG 上右击，选择“Add Class”命令，就会进入 MFC Class Wizard 向导对话框，在“Name”文本框中输入 CMyDlg，在“Base class”下拉列表框中选择所要创建的类的基类，这里选择 CDialog（该类也是默认的），在“Dialog ID”下拉列表框中选择 IDD_MYDLG（一般情况下，ClassWizard 自动使类 CMyDlg 与 IDD_ MYDLG 模板联系起来），其他选项使用默认值，单击“Finish”按钮后，对话框类 CMyDlg 就被创建完成，如图 4.14 所示。

单击“Finish”按钮后，会自动关闭 MFC Class Wizard 向导对话框，回到工作台的 Class View 页面，在工程的类浏览器中多了一个新类 CMyDlg，如图 4.15 所示。此时，这个新类只有两个函数：一个构造函数和一个析构函数。

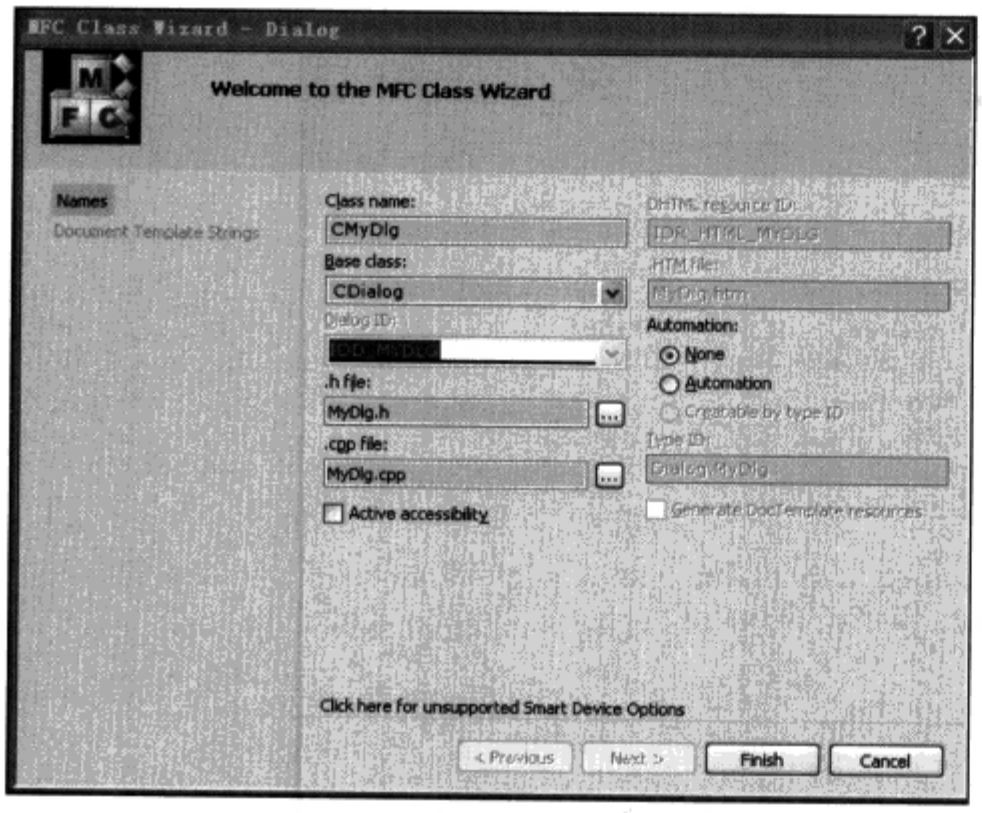


图 4.14　MFC 添加事件处理向导对话框

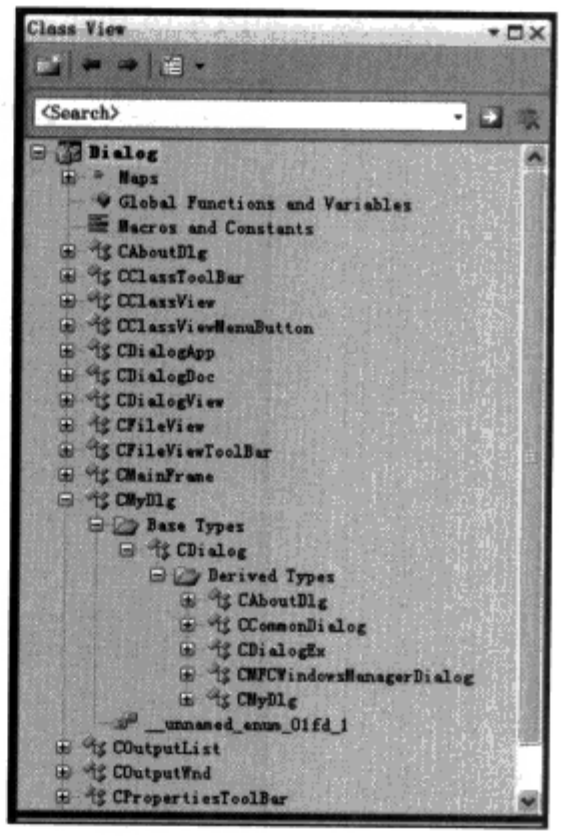


图 4.15　添加一个新类的结果

其实，完成这个步骤以后就可以将对话框显示出来了，但此时显示的对话框不能进行任何相加操作，这是因为还没有给这个对话框类添加与控件相关联的数据成员，还不能进行数据交换。

步骤 5：使用 ClassWizard 为对话框类加入与控件相关联的成员变量。

对话框的主要功能是进行数据交换，本例对话框的任务就是输入两个数据，然后显示两个数据之和。为实现这些功能，对话框需要有一组成员变量来存储数据。在对话框中，控件用来表示或输入数据，因此存储数据的成员变量应该与控件相对应。

与控件对应的成员变量既可以是一个数据，也可以是一个控件对象，这将由具体需要来确定。例如，可以为一个编辑框控件指定一个数据变量，这样就可以很方便地取得或设置编辑框控件所代表的数据，如果想对编辑框控件进行控制，则应该为编辑框指定一个 CEdit 对象，通过 CEdit 对象，用户可以控制控件的行为。需要指出的是，不同类的控件对应的数据变量的类型往往是不相同的，而且一个控件对应的数据变量的类型也可能有多种。表 4.2 说明了控件的数据变量的类型。

表 4.2　　控件数据变量的类型

控　　件	数据变量类型
编辑框	Cstring、int、UINT、long、DWORD、float、double、short、BOOL、COleDateTime、COleCurrency
普通检查框	BOOL（真表示被选中，假表示未选中）
三态检查框	int（0 表示未选中，1 表示选中，2 表示不确定状态）
单选按钮（组中的第一个按钮）	int（0 表示选择了组中第一个单选按钮，1 表示选择了第二个……-1 表示没有一个被选中）
不排序的列表框	CString（为空则表示没有一个列表项被选中）， int（0 表示选择了第一项，1 表示选了第二项，-1 表示没有一项被选中）
下拉式组合框	CString, int（含义同上）
其他列表框和组合框	CString（含义同上）

前面讲过，利用 ClassWizard 可以很方便地为对话框类加入成员变量，请读者按下列步骤进行操作。

（1）在 IDD_MYDLG 对话框上右击，并选择“Add Variable”命令，即可进入“Add Member Variable Wizard”对话框，如图 4.16 所示。

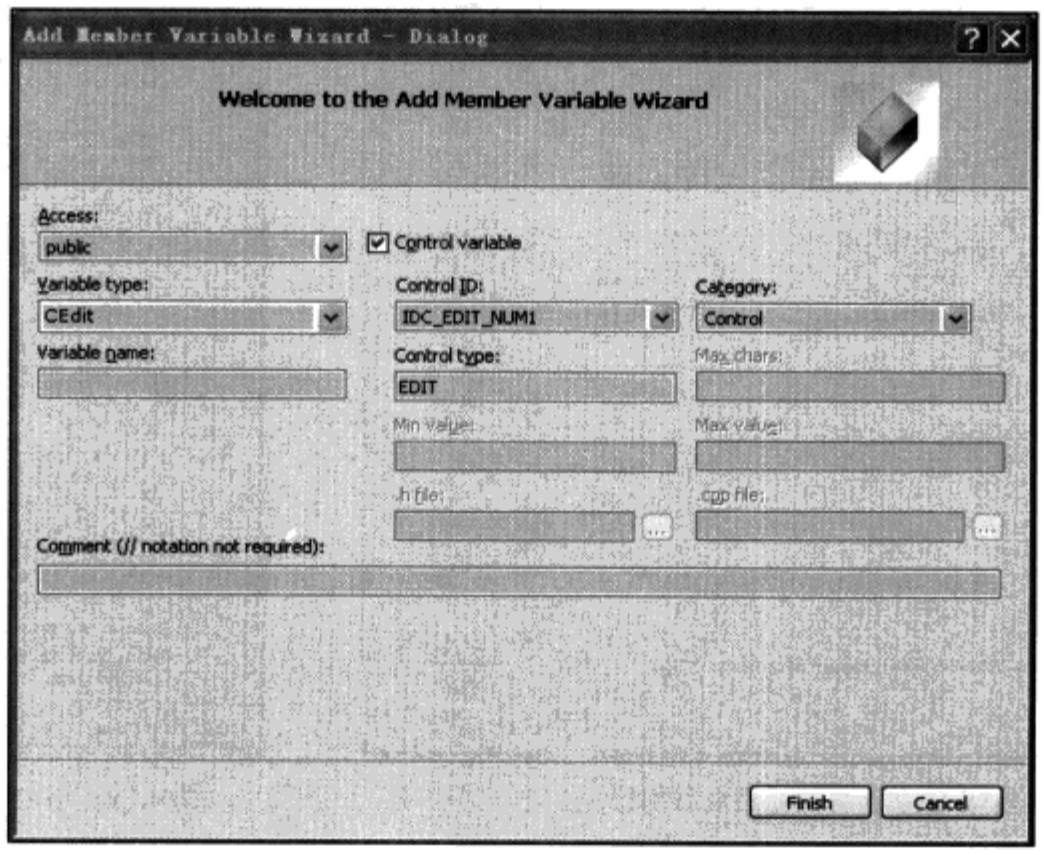


图 4.16　添加控件成员变量向导

（2）选择“Control variable”复选框，则表示此时添加的是控件成员变量。默认选择的类是 CMyDlg。这时，在下面的“Control ID”下拉列表框中会显示对话框控件的 ID，如图 4.17 所示。

（3）在“Control ID”下拉列表框中选中 IDC_EDIT_NUM1 控件，然后在窗口左侧的“Variable name”文本中输入变量名 m_nNum1，在窗口最右侧的“Category”下拉列表框中选择“Value”选项，在“Variable type”下拉列表框中选择“int”选项。其余的“Access”和“Control type”选项全部按照默认值设置。

（4）此时，给这个成员变量加一个限定值，将 m_ nNum1 的值限制在 0～200 之间。然后在“Add Member Variable Wizard”对话框中的“Min value”文本框中输入 0，在“Max value”文本框中输入 200，如图 4.17 所示。有了这个限制以后，对话框会对输入的值进行有效性检查，若输入的值不在限制范围内，则对话框会提示用户输入有效的值。设置完毕，单击“Finish”按钮后，数据变量 m_nNum1 就会被加入到变量列表中。

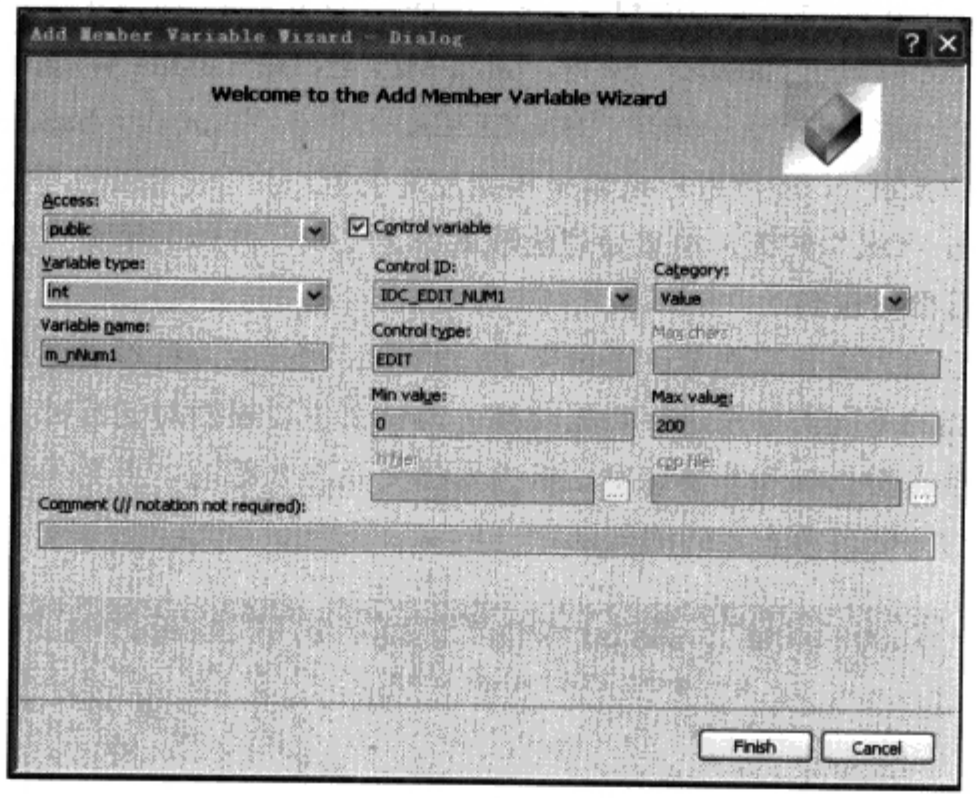


图 4.17　选择控件 ID

（5）按照上面的步骤，为剩余的 ID 号为 IDC_EDIT_NUM2 和 IDC_EDIT_SUM 的两个控件添加成员变量分别为 m_nNum2 和 m_nSum，添加过程和 m_nNum1 变量一样，如图 4.18 所示。变量 m_nNum2 与 m_nNum1 一样，设置限定值为 0～200，而变量 m_nSum 不需要设置限定值。

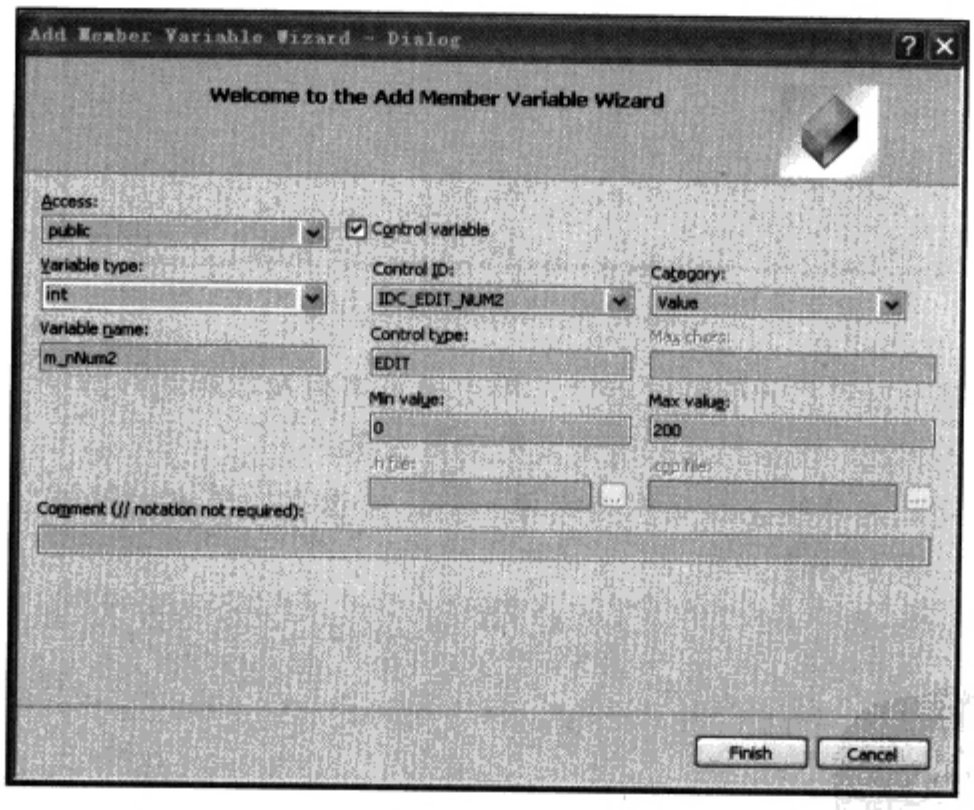


图 4.18　设置限定值

当对话框类中与控件相关联的成员变量添加完毕之后，就可以添加代码来进行数据交换了。如果对添加的成员变量不满意，可以在 ClassWizard 的 Member Variables 选项卡中选中不满意的变量，然后单击“Delete Variable”按钮将其删除或者从 ClassView 中找到 CMyDlg 类然后右击并选择“go to definition”命令，然后从 CMyDlg 类中找到并删除相应的成员变量。另外，还可以通过单击“Add Variable”按钮再次添加自己满意的正确的变量。

步骤 6：在对话框类中添加需要的消息处理函数。

由于程序最终的目的是实现两数相加，在对话框中修改的名为“相加”的按钮就是负责实现两数

相加，因此需要添加该按钮的消息处理函数。方法很简单，打开 IDD_MYDLG 对话框模板资源，右击“相加”按钮，选择“Add Event Handler”命令，弹出 MFC Event Handler Wizard 对话框，在这个对话框中，“Message type”给出了默认的选项“BN_CLICKED”，“Function handler name”也给出了默认的函数名选项“OnCancel”，应用程序通常是根据控件的 ID 号生成相应的函数名，这里就用这个默认的函数名，最后单击“OK”按钮，消息处理函数就添加好了。具体的函数内容将在下面创建模态对话框和创建非模态对话框时讲到。

至此，创建对话框的大部分工作都已经做完了，就差最后一步，将对话框窗口创建并显示出来，然后再根据对话框种类的不同而分别进行数据交换，为什么说要根据对话框种类的不同而分别进行数据交换呢？这是因为模态对话框和非模态对话框的数据交换不太相同。下面分别介绍如何创建并显示模态对话框和非模态对话框及两者之间的差别。

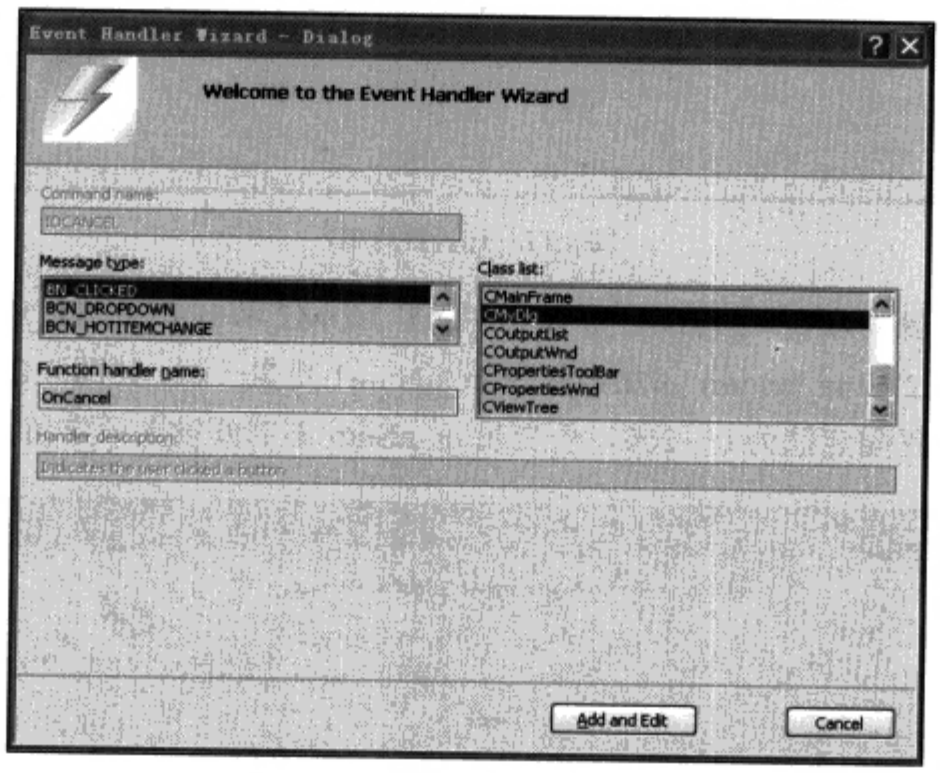


图 4.19　添加事件处理向导

模态对话框是主程序窗口打开的临时窗口，用于显示消息及取得用户数据，用户要关闭对话框才能恢复主窗口的工作，因此，模态对话框可定义为局部变量。

创建模态对话框比创建非模态对话框要简单一些，应该在“模态对话框”命令的消息处理函数 OnModedlg()中来实现，代码如下：

```
void CMainFrame::OnModedlg()
{
    // TODO: Add your command handler code here
    CMyDlg m_Mydlg;
    m_Mydlg.DoModal();
}
```

DoModal()负责创建和撤销模态对话框。在创建对话框时，DoModal()的任务包括载入对话框模板资源、调用 OnInitDialog()初始化对话框和将对话框显示出来。完成对话框的创建后，DoModal()启动一个消息循环，以响应用户的输入。由于该消息循环截获了几乎所有的输入消息，使主消息循环收不到对对话框的输入，致使用户只能与模态对话框进行交互，而其他用户界面对象收不到输入信息。

若用户在对话框中单击 ID 为 IDOK 的按钮（通常该按钮的标题是“确定”或“OK”），或按【Enter】键，则对话框类 CMyDlg 的父类 CDialog 的 OnOK()函数将被调用。OnOK()首先调用 UpdateData(True)将数据从控件传给对话框成员变量，然后调用 CDialog::EndDialog()关闭对话框。关闭对话框后，

DoModal()会返回值 IDOK。

若用户单击 ID 为 IDCANCEL 的按钮（通常其标题为“取消”或“Cancel”），或按【Esc】键，则会导致 CDialog::OnCancel()的调用。该函数只调用 CDialog::EndDialog()关闭对话框。关闭对话框后，DoModal()会返回值 IDCANCEL（该实例中，将 CDialog::OnCancel()的调用给注释了）。

程序根据 DoModal()的返回值是 IDOK 还是 IDCANCEL，就可以判断出用户是确认还是取消了对对话框的操作。

运行结果是，不退出模态对话框时，用户无法访问其他菜单命令或窗口，如图 4.20 所示。

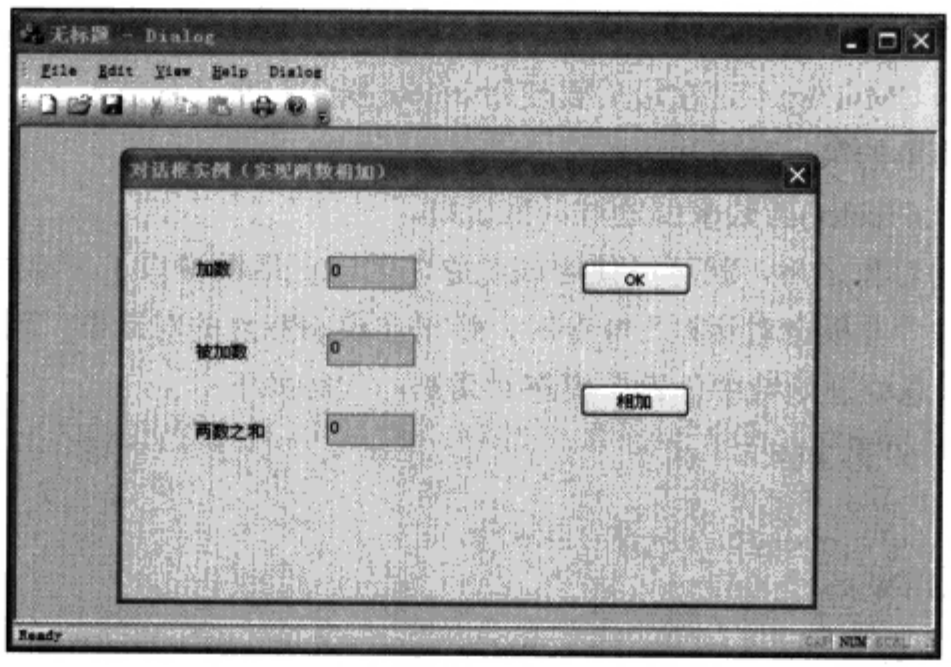


图 4.20　弹出模态对话框的运行结果

下面再看看模态对话框的数据交换。模态对话框中“相加”按钮的消息处理函数中的代码和非模态对话框中的代码是相同的，也就是不改变先前已经添加好的 OnCancel()中的代码。非模态对话框的数据交换代码详见 4.2.2 小节。

这时有的读者可能很不理解，在前面说过模态对话框和非模态对话框的数据交换不太相同，那么为什么在 OnCancel()中的代码是相同的呢？

以上讨论的数据交换都局限于控件和数据成员之间的交换，而真正应用程序中说到的数据交换包括对话框窗口、对话框类对象和视图类对象三者之间的数据交换，而不仅仅是控件和数据成员之间的传递。在非模态对话框中，对话框窗口、对话框类对象和视图类对象三者之间的数据传递都是要手动添加的；而在模态对话框中，对话框窗口和对话框类对象之间的数据交换是系统自动完成的，自动完成的过程就在 DoModal()函数中实现。

若要关闭模态对话框，只需要单击“OK”按钮或者“Cancel”按钮即可。CDialog 类提供有默认处理函数 OnOK()和 OnCancel()。默认处理函数将调用 CDialog 的成员函数 EndDialog()关闭对话框。

4.2.2　非模式对话框

1. 创建非模态对话框

由于非模态对话框是不垄断用户输入的，即使对话框不返回，也可以切换到其他窗口，所以非模态对话框必须定义为全局变量或用 new 操作符产生一个。要创建非模态对话框，一般需要 3 个步骤。

（1）构造一个对话框类的对象。

这里介绍两种构造一个对话框类的对象的方法。一种方法是在框架窗口类的头文件中声明一个对话框类的对象成员变量，如下：

```
class CMainFrame : public CFrameWnd
{
…
public:
CMyDlg m_Mydlg;
…
};
```

由于这里声明的是 CMyDlg 类的对象，因此还要在框架窗口类的头文件中加入语句#include "MyDlg.h"，将对话框类的头文件包含进来，否则编译时会报错。

另一种方法是在框架窗口类中声明一个对话框类的指针变量，然后使用 new 操作符动态创建，如 CMyDlg* pMyDlg=new CMyDlg;不过在最后要使用 delete 操作符将其释放。

这里，选用比较简单的第一种方法。

（2）调用 CDialog::Create()函数来创建对话框窗口。

构造完对话框类的对象之后，就可以使用该对象来创建对话框窗口了，前面添加“对话框”菜单的初衷是想当用户选择“非模态对话框”命令时，弹出非模态特点的对话框，因此在“非模态对话框”命令的 COMMAND 消息处理函数中创建非模态对话框，代码如下：

```
void CMainFrame::OnModelessdlg()
{
    // TODO: Add your command handler code here
    if (!m_Mydlg.m_hWnd)
    {
        m_Mydlg.Create(IDD_MYDLG);
    }
    m_Mydlg.ShowWindow(SW_SHOW);
}
```

（3）调用 CWnd::ShowWindow()函数来显示对话框窗口。

代码中的 if 语句是用来判断该对话框是否已经被创建，如果已经创建，则无须再创建，而只需将其显示出来。

使对话框窗口显示的代码同样是放在“非模态对话框”命令的 COMMAND 消息处理函数中，最后的运行结果如图 4.21 所示。

由图 4.21 不难看出，即使该对话框不关闭，用户也可以再次去选择其他菜单命令。但该对话框目前还不具备数据交换功能，在加上这一功能之前，有必要了解一下对话框的数据交换机制。

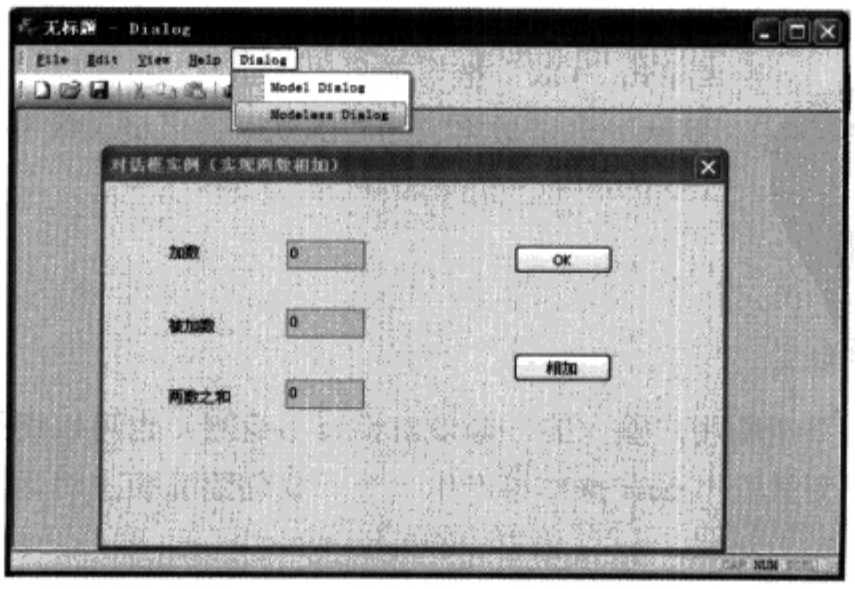


图 4.21　弹出非模态对话框的运行结果

2．对话框的数据交换机制

对话框的数据成员变量存储了与控件相对应的数据。数据变量需要和控件交换数据，以完成输入或输出功能。例如，一个编辑框既可以用来输入，也可以用来输出。当用作输入时，用户在其中输入数值之后，对应的数据成员应该更新以便与编辑框中的数值相同；当用作输出时，应及时刷新编辑框的内容以反映相应数据成员的变化。因此，对话框就需要一种机制来实现这种数据交换功能，这对对话框而言是至关重要的。

MFC 提供了类 CDataExchange 来实现对话框类与控件之间的数据交换（DDX），该类还提供了数据有效机制（DDV）。数据交换和数据有效机制适用于编辑框、复选框、单选按钮、列表框和组合框。

数据交换的工作由 CDialog::DoDataExchange()来完成。读者可以在工作台的 ClassView 类浏览器中找到对话框类 CMyDlg 类的 DoDataExchange()函数，代码如下：

```
void CMyDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(CMyDlg)
        DDX_Text(pDX, IDC_EDIT_NUM1, m_nNum1);
        DDV_MinMaxInt(pDX, m_nNum1, 0, 200);
        DDX_Text(pDX, IDC_EDIT_NUM2, m_nNum2);
        DDV_MinMaxInt(pDX, m_nNum2, 0, 200);
        DDX_Text(pDX, IDC_EDIT_SUM, m_nSum);
        //}}AFX_DATA_MAP
}
```

其中的斜体部分并不是真的在应用程序中用斜体字输入，而是为了告诉读者，这段代码在 VC 中的显示是浅灰色的，是由 ClassWizard 自动加入的。

可以看出，DoDataExchange()函数只有一个参数，即一个 CDataExchange 对象的指针 pDX。在该函数中调用了 DDX 函数来完成数据交换，DDV 函数来进行数据有效性检查。

当程序需要交换数据时，并不需要用户直接调用 DoDataExchange()函数，而应该调用 CWnd::UpdateData()函数，这是因为在 UpdataData()函数内部调用了 DoDataExchange()。UpdataData()函数原型如下：

```
BOOL UpdateData ( BOOL bSaveAndValidaet = TRUE);
```

该函数只有一个布尔型参数 bSaveAndValidate，它决定了数据传送的方向。若参数值为 Ture，即调用 UpdateData(True)，表示将数据从对话框的控件中传送到对应的数据成员中；若参数值为 False，即调用 UpdateData(False)，表示将数据从数据成员中传送给对应的控件。

对话框在弹出前会调用其成员函数 OnInitDialog()，默认的 CDialog::OnInitDialog()函数中调用了 UpdateData(False)，这样，在对话框创建时，数据成员的初始值就会反映到相应的控件上，因此，通常对话框上控件的初始化工作都放在对话框类的 OnInitDialog()函数中。若用户单击“OK”（确定）按钮退出对话框，则对话框认为输入有效，就会调用 UpdataData(True)将控件中的数据传给数据成员。

经过上面的分析，可知在非模态对话框的“相加”按钮的处理函数中就应该输入如下代码来实现数据交换：

```
void CMyDlg::OnCancel()
{
    // TODO: Add extra cleanup here
    UpdateData(TRUE);
    m_nSum=m_nNum1+m_nNum2;
```

```
        UpdateData(FALSE);
        //CDialog::OnCancel();
    }
```

首先将 UpdateData()函数参数置为 True，将用户输入到控件的数据传给数据成员，然后将两个数据成员的值相加并将所得结果赋值给另一个数据成员 m_nSum，最后，再调用 UpdateData()函数将参数置为 False，刷新屏幕上控件中显示的值。

在 CMyDlg::OnCancel()函数代码段下面，把 CDialog::OnCancel();注释掉的目的是不让它调用父类的 OnCancel ()函数，若调用了，对话框会关闭。

运行结果如图 4.22 所示。

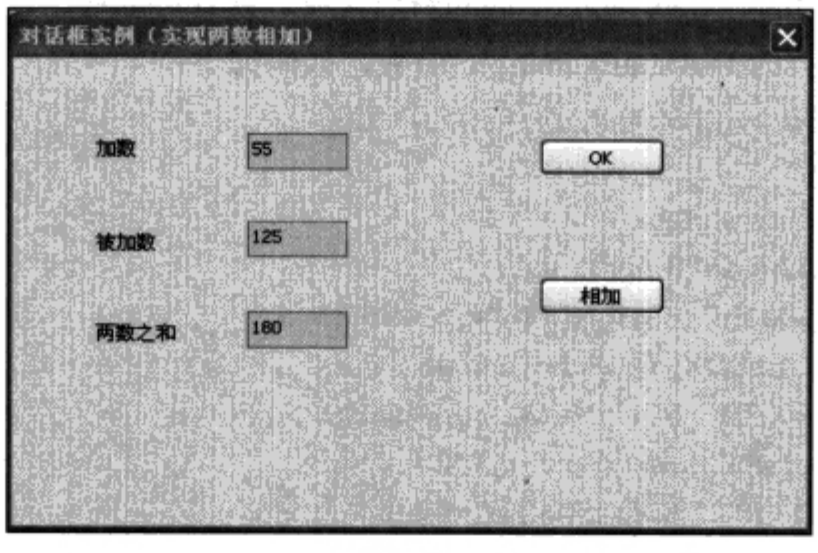


图 4.22　实现两数相加对话框的运行结果

对于非模态对话框，当其被关闭或者删除时，默认处理函数 OnClose()将调用 RestroyWindow()删除该对话框。

3．模态对话框和非模态对话框的区别

通过以上所讲述的有关模态对话框和非模态对话框创建的知识，读者不难分析二者之间的区别，基本如下。

- 非模态对话框的模板必须具有 Visible 风格，否则对话框将不可见，更保险的办法是调用 CWnd::ShowWindow(SW_SHOW)来显示对话框，而不管对话框是否具有 Visible 风格。而模态对话框则无须设置该项风格。
- 非模态对话框对象是用 new 操作符在堆中动态创建的或定义为全局变量形式，而不是以成员变量的形式嵌入到别的对象中或以局部变量的形式构建在堆栈上。通常应在对话框的拥有者窗口类内声明一个指向对话框类的指针成员变量，通过该指针可访问对话框对象。
- 通过调用 CDialog::Create()函数而不是 CDialog::DoModal()函数来启动对话框，这是模态对话框的关键所在，也是二者最大的区别。由于 Create()函数不会启动新的消息循环，对话框与应用程序共用同一个消息循环，这样对话框就不会垄断用户的输入。Create()在显示了对话框后就立即返回，而 DoModal()是在对话框被关闭后才返回的。众所周知，在 MFC 程序中，窗口对象的生存期应长于对应的窗口，也就是说，不能在未关闭屏幕上窗口的情况下先把对应的窗口对象删除。由于在 Create()函数返回后，不能确定对话框是否已关闭，这样也就无法确定对话框对象的生存期，因此只好在堆中构建对话框对象，而不能以局部变量的形式来构建之。
- 必须调用 CWnd::DestroyWindow()而不是 CDialog::EndDialog()来关闭非模态对话框。调用 CWnd::DestroyWindow()是直接删除窗口的一般方法。由于默认的 CDialog::OnOK()和 CDialog::OnCancel()函数均调用 EndDialog()函数，因此用户必须编写自己的 OnOK()和

OnCancel()函数并且在函数中调用 DestroyWindow()来关闭对话框。

- 如果是用 new 操作符构造非模态对话框对象，必须在对话框关闭后，用 delete 操作符删除对话框对象。在屏幕上一个窗口被删除后，框架类会调用 CWnd::PostNcDestroy()这个虚拟函数，程序可以在该函数中完成删除窗口对象的工作。
- 必须有一个标志表明非模态对话框是否已经打开。由于在非模态对话框不被关闭的情况下，用户很有可能再次打开这个非模态对话框，这样肯定是要报错的。因此，应该定义一个标志，程序根据标志来决定是打开一个新的对话框，还是只把原来打开的对话框激活。通常可以用拥有者窗口中的指向对话框对象的指针作为这种标志，当对话框关闭时，给该指针赋 NULL 值，以表明对话框对象已不存在了；也可以将所创建的类的对象的成员变量 m_hWnd 作为标志。本实例中就是用后者作为标志的。

4.2.3　属性页对话框

有过开发经验的读者，很可能都会遇到这种情况：当输入一份资料或对某一对象进行设置时需要用到大量的控件，以致于一个对话框中放不下，而这些控件描述的都是针对某一对象的类似的属性（如某一试题的篇、章、节、目、分数、时间、难度、信度、能力要求等，实际需要的属性远不止这些），这些属性不能分成多个对话框来显示，使用普通的对话框技术，很难解决这一问题。

MFC 提供了对属性页对话框的支持，可以很好地解决上述问题。属性页对话框实际上是一个包含了多个子对话框的对话框，这些子对话框通常被称为页（Page），每次只有一个页是可见的，在对话框的顶端有一行选项卡，用户通过单击这些选项卡可切换到不同的页。显然，属性页对话框可以容纳大量的控件来描述某一对象的诸多属性。

一个典型的属性页对话框如图 4.23 所示，这类对话框包括一个 CPropertySheet 和几个 CPropertyPage。可以这样看待这个对话框：其中，大的框架“页面设置”就是一个 CPropertySheet，而“页边距”、“纸张”、“版式”和“文档网络”每一项都是一个 CPropertyPage。

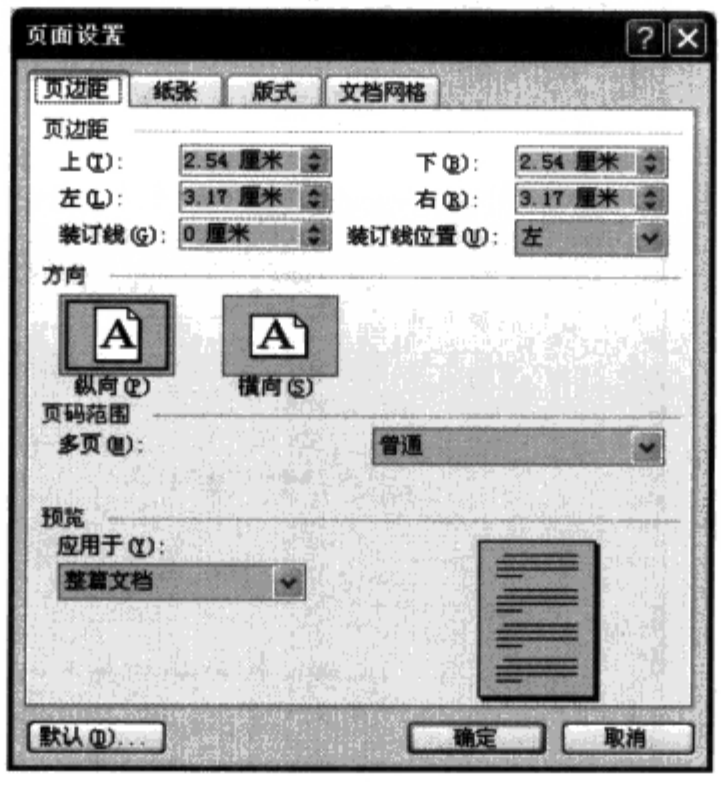


图 4.23　页面设置对话框

下面以一个实例来讲解如何创建一个属性页对话框。建立一个 MFC 应用程序，为该应用程序新添加一个菜单命令，该菜单命令负责弹出将要创建的属性页对话框，这个属性页对话框有 3 个页面，分

别表示个人信息、兴趣爱好及工作情况。

步骤 1：新建一个 MFC 单文档应用程序，工程名为 Dialog2 或由用户自定义。

步骤 2：分别为各个页创建对话框模板，并为每个页创建新类。

在资源面板中新插入 3 个对话框，Caption 属性分别改为“个人信息”、“兴趣爱好”、“工作情况”，为便于管理，将它们的 ID 分别改为 IDD_PAGE1、IDD_PAGE2 和 IDD_PAGE3。将 3 个对话框上的默认的“OK”和“Cancel”按钮去掉。

由于每一个对话框都对应于属性页对话框中的一个页面，因此对这 3 个对话框而言，它们的样式有特殊的要求：Style 属性只能是 Child， Boder 属性为 Thin，确保“Tilte Bar”和“Disable”为 True，这些要求可以在 MSDN 中查“CPropertyPage，styles”得到。其中属性面板的设置如图 4.24 所示。

在创建每个页的对话框模板时要注意，每页的模板最好具有相同的尺寸，如果尺寸不统一，则框架将根据最大的页来确定属性页对话框的大小。

接下来，根据各个页的模板，用 ClassWizard 分别为每个页创建 CPropertyPage 类的派生类。由于这一过程与创建普通对话框类的过程类似，不同的是在创建新类对话框中应在“Base class”下拉列表框中选择“CPropertyPage”而不是“CDialog”类，因此只给出其中一个页的新类的创建过程。

例如，为 ID 号为 IDD_PAGE1 的对话框创建新类，首先双击 ID 号为 IDD_PAGE1 的页，或者在 IDD_PAGE1 对话框上右击，并选择“Add Class”命令在弹出的“MFC Class Wizard”对话框中的“Class name”文本框中，输入 CPage1，在“Base class（基类）”下拉列表框中选择 CPropertyPage，系统会根据创建的类名而自动设置“Dialog ID”选项为 IDD_PAGE1（创建的类要和对话框的 ID 相关联），如图 4.25 所示，最后单击“Finish”按钮完成类的创建。另外两个对话框的类名分别为 CPage2 和 CPage3，同样都是从 CPropertyPage 类派生。

图 4.24　设置属性

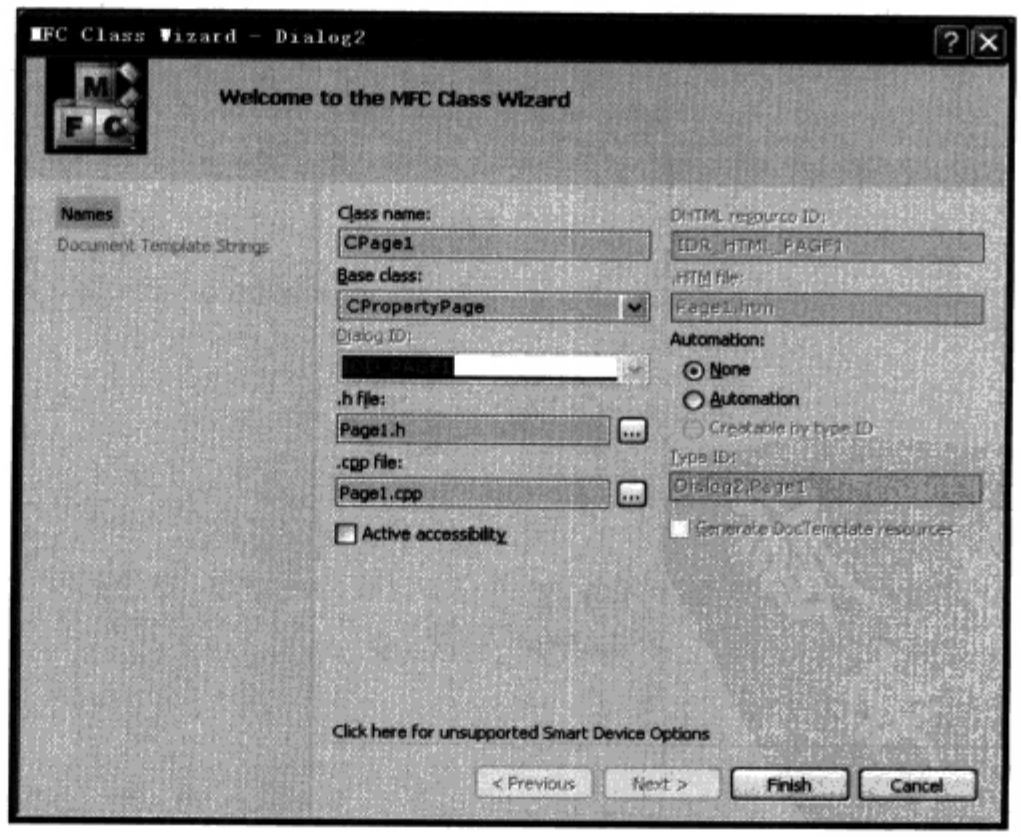


图 4.25　添加新的属性页类向导

步骤 3：为每页的对话框模板添加控件及其相对应的成员变量。

第一页对话框模板添加了 3 个静态正文控件、两个编辑框（ID 分别为 IDC_EDIT_NAME、

IDC_EDIT_AGE）和一个组合框（ID 为 IDC_COMBO_SEX），如图 4.26 所示。由于组合框的作用是让用户选择性别，而性别只有“男”和“女”，因此可以先给它赋初值，这一工作在组合框属性面板中的 Data 页面上完成，如图 4.27 所示。需要注意的是，在 Data 页面的列表框中，若想给组合框增加新的列表项，必须使用【Ctrl+Enter】组合键才能回车到下一行增加新的列。该组合框在程序运行的时候也不允许用户再向其中添加值，因此，将该组合框的 Type 属性设置为 Drop List，用户不能在组合框的文本框中输入数据，而且此时的组合框只能捆绑 int 类型的成员变量。Type 属性的设置在组合框属性面板的 Styles 页面上。

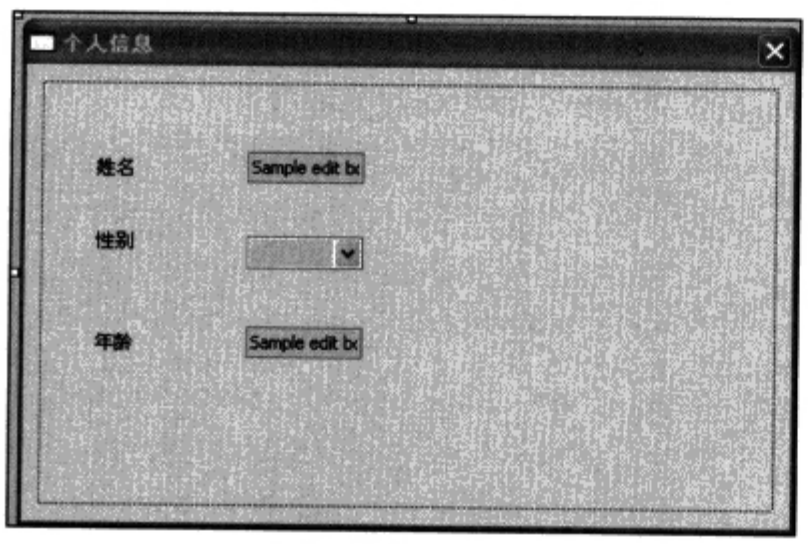


图 4.26　第 1 页对话框模板

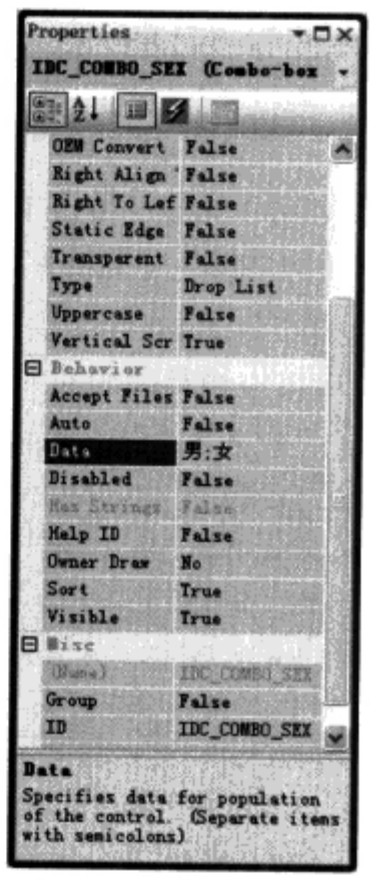


图 4.27　设置组合框控件属性

第二页对话框模板添加了 4 个复选框控件，设置其 ID 分别为 IDC_CHK_TBLTENNIS、IDC_CHK_BSKBALL、IDC_CHK_SWIM 和 IDC_CHK_TENNIS，如图 4.28 所示。

第三页对话框模板上有两个组框、3 个单选按钮（ID 分别为 IDC_RAD_PRO、IDC_RAD_SOFT、IDC_RAD_ANALY）和一个列表框，ID 改为 IDC_LIST_CITY，如图 4.29 所示。将第一个单选按钮的 Group 属性选中，直到下一个（其这里“下一个”是指 Tab Order 的顺序，而不是物理位置上的顺序）设置 Group 属性的控件之前的单选按钮设为一组，同一时间只能选一个，本实例中，要将 3 个单选按钮设为一组，因此只将第一个单选按钮的 Group 属性选中。一组单选按钮只能捆绑一个 int 型的成员变量，表示该组中选中的单选按钮的序号。列表框的属性面板中不同于组合框，它不能添加数据，需要在 OnInitDialog()中由用户手动添加。

设计好对话框及其中的控件之后，就应该为每页加入与控件对应的成员变量，这个过程与为普通对话框类加入成员变量类似。这里只演示第一个页面中，组合框控件的成员变量的添加方法。

打开 IDD_PAGE1 对话框，在上面右击，并选择“Add Variable”命令，即打开“Add Member Variable Wizard”对话框，选择“Member Variable”页面，然后选中“Control variable”复选框。在“Control ID”下拉列表框中双击“IDC_COMBO_SEX”选项，在“Variable name”文本框中输入变量名 m_sex，在“Category”下拉列表框中选择“Value”选项，在“Variable type”下拉列表框中选择“int”选项，最后单击“Finish”按钮完成设置。

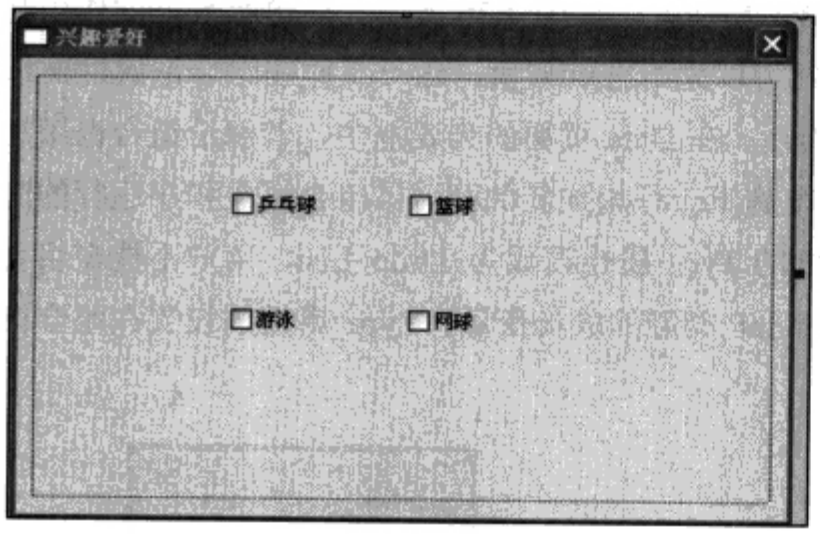


图 4.28　第二页对话框模板

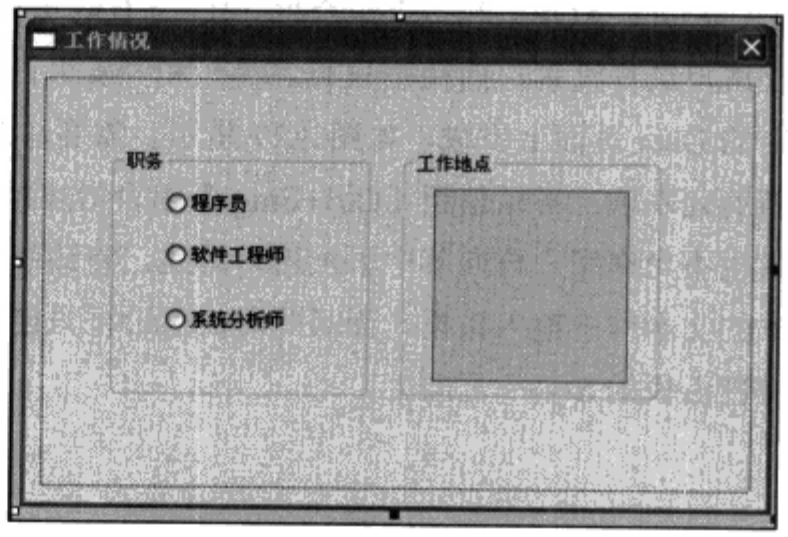


图 4.29　第三页对话框模板

但是 Visual Studio 2010 有时通过“Add Member Variable Wizard”向导对话框无法添加控件成员变量，此时可以直接在 Page1.h，Page1.cpp，Page2.h，Page2.cpp，Page3.h，Page3.cpp 这些函数中添加控件成员变量。由于 MFC 提供的 DDX（X 表示 eXchange），允许程序员事先设定控件与变量的关系，因此不但可以令控件的内容一旦发生改变就自动传送到变量去，也可以借 MFC 提供的 DDV（V 表示 Validation）设定字段的合理范围。例如，在 Page1.h 中添加公有控件成员变量：

```
Public:
    int m_sex;
    CString m_age;
    CString m_name;
```

然后再在 Page1.cpp 文件中的 DoDataExchange()函数中添加如下代码，也可以实现上述功能。

```
void CPage1::DoDataExchange(CDataExchange* pDX)
{
    CPropertyPage::DoDataExchange(pDX);
    DDX_CBIndex(pDX, IDC_COMBO_SEX, m_sex);
    DDX_Text(pDX, IDC_EDIT_AGE, m_age);
    DDX_Text(pDX, IDC_EDIT_NAME, m_name);
}
```

上面的代码表示将 Page1.h 中的 3 个成员变量映射到对话框中的 3 个控件——Combol、Edit、Edit 上。使用者单击“OK”按钮时，第一个字段的 Combol 字段的性别内容自动存储到 m_sex 变量中，第二个 Edit 字段的年龄内容自动存储到 m_age 中，第三个 Edit 字段的名称内容也会自动存储到 m_name 变量中。

相应的，在 Page2.h 中添加公有控件成员变量：

```
public:
    BOOL m_bskball;
    BOOL m_swim;
    BOOL m_tbltennis;
    BOOL m_tennis;
```

然后再在 Page2.cpp 文件中的 DoDataExchange()函数中添加如下代码：

```
void CPage2::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Check(pDX, IDC_CHK_BSKBALL, m_bskball);
    DDX_Check(pDX, IDC_CHK_SWIM, m_swim);
```

```
    DDX_Check(pDX, IDC_CHK_TBLTENNIS, m_tbltennis);
    DDX_Check(pDX, IDC_CHK_TENNIS, m_tennis);
}
```

在 Page3.h 中添加公有控件成员变量：

```
public:
    int m_job;
    CString m_city;
```

然后再在 Page3.cpp 文件中的 DoDataExchange()函数中添加如下代码：

```
void CPage3::DoDataExchange(CDataExchange* pDX)
{
    CPropertyPage::DoDataExchange(pDX);
    DDX_LBIndex(pDX, IDC_LIST_CITY, m_city);
    DDX_Radio(pDX, IDC_RAD_PRO, m_job);
}
```

每个页面上与控件相对应的成员变量如表 4.3 所示。

表 4.3　　每个页面上与控件相对应的成员变量

页　面	控 件 类 型	ID　号	成 员 变 量
第一个页面	编辑框（姓名）	IDC_EDIT_NAME	m_name (CString)
	组合框（性别）	IDC_COMBO_SEX	m_sex (int)
	编辑框（年龄）	IDC_EDIT_AGE	m_age (CString)
第二个页面	复选按钮（乒乓球）	IDC_CHK_TBLTENNIS	m_tbltennis (BOOL)
	复选按钮（蓝球）	IDC_CHK_BSKBALL	m_bskball (BOOL)
	复选按钮（游泳）	IDC_CHK_SWIM	m_swim (BOOL)
	复选按钮（网球）	IDC_CHK_TENNIS	m_tennis (BOOL)
第三个页面	单选按钮组（职务）	————	m_job (int)
	列表框（工作地点）	IDC_LIST_CITY	m_city (CString)

不难看出，本例中控件的成员变量的命名是和控件的 ID 号相关的，这样便于记忆，不容易混淆。而单选按钮组的成员变量的命名和 ID 无关，这是因为 3 个单选按钮设为一组，也就是说这 3 个单选按钮只有一个 int 型的成员变量，单选按钮组中表示的是职务，因此将其成员变量命名为 m_job。

步骤 4：创建并显示属性页对话框。

创建并显示对话框的代码通常放在显示对话框的命令处理函数中，因此在资源面板中，在原标准菜单的基础上新增加一个菜单“对话框”，在其下新建一个菜单命令“属性页对话框”，这个菜单命令的 ID 为 IDM_PROP，单击该菜单命令则弹出属性页对话框。

右击该菜单命令，并选择“Add Event Handler”命令，在 MFC“Event Handler Wizard”对话框中，选择 CDialog2View 类，在 CDialog2View 类中添加“属性页对话框”菜单命令的 COMMAND 消息处理函数 OnProp()，创建并显示属性页对话框的代码如下：

```
void CDialog2View::OnProp()
{
    // TODO: Add your command handler code here
    CPropertySheet propSheet;
    //定义 3 个属性页的页面对象
    CPage1 page1;
    CPage2 page2;
    CPage3 page3;
```

```
    //将页面加入到属性页对话框的框架对象中
    propSheet.AddPage(&page1);
    propSheet.AddPage(&page2);
    propSheet.AddPage(&page3);
    propSheet.DoModal();
}
```

其中 AddPage()函数是 CPropertySheet 类的成员函数，负责将一个页对象加入属性页对话框的框架对象中，原型如下：

```
void AddPage( CPropertyPage *pPage );
```

该函数只有一个参数 pPage，指向要加入的页面，该参数不能为空。

上面该段代码首先实例化一个 CPropertySheet 类的对象 propSheet，该对象就作为属性页对话框的框架；然后又分别实例化类 CPage1、CPage2、CPage3 的对象，它们是属性页对话框的 3 个页；再通过调用 CPropertySheet 类的成员函数 AddPage()将 3 个页面加入框架对象 propSheet 中；最后，调用 DoModal()函数创建并显示对话框窗口。

可以看出，属性页对话框框架的创建过程与普通对话框的创建过程是类似的，不同之处仅仅是还需要将页对象加入 CPropertySheet 对象中。如果要创建的是模态对话框，应调用 CPropertySheet::DoModal()，本例创建的是模态对话框；如果想创建非模态对话框，则应该调用 CPropertySheet::Create()。

在 CDialog2View 类的头文件中加入下面的语句来将新建的 3 个类的头文件包含进来，这是因为在 CDialog2View 类的菜单命令的处理函数 OnProp()中使用这 3 个类：

```
#include "Page2.h"
#include "Page1.h"
#include "Page3.h"
```

运行结果如图 4.30 所示，此时，这个对话框还不具有数据交换的能力，要给它添加这样一个功能：当用户将所有信息输入完毕之后，若单击“确定”按钮退出，则在屏幕上打印出用户的输入信息；如果单击“取消”按钮退出，则什么都不做。

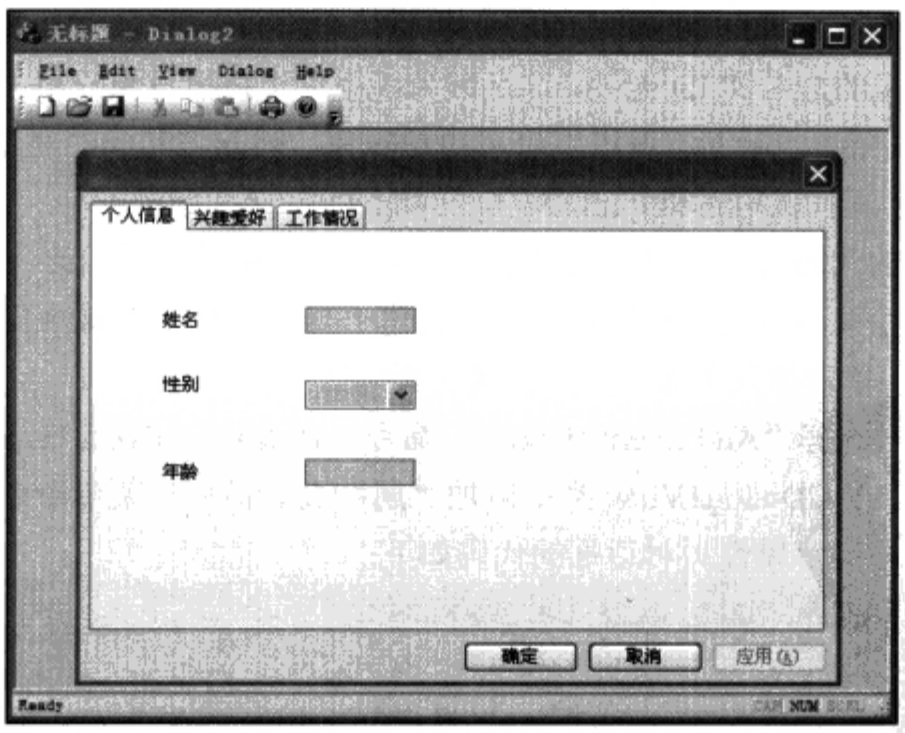


图 4.30　添加属性页运行结果

步骤 5：在各个页的 OnInitDialog()函数中初始化页。

读者可能已经注意到了，在图 4.30 显示的对话框上，第一页面上的组合框并没有显示内容，这是 Drop List 属性组合框的特点，只有当用户单击下拉按钮时，数据才会显示出来，要想使“性别”这个

选项在对话框一启动时就在组合框中显示出来，可以通过“Event Handler Wizard”对话框添加第一页面的 OnInitDialog()函数来实现。

打开 Resource View 面板，选中并展开“Dialog”选项，选择其中的 IDD_PAGE1，双击后会打开 IDD_PAGE1 的页面。在组合框上右击，并选择“Add Event Handler”命令弹出“Event Handler Wizard”对话框。在“Class list”列表框中选择 CPage1 类，然后在“Messages”列表框中找到 WM_INITDIALOG 消息，双击该消息，其消息处理函数 OnInitDialog()就被添加进来了，在该函数中输入如下代码：

```
BOOL CPage1::OnInitDialog()
{
    CPropertyPage::OnInitDialog();

    // TODO: Add extra initialization here
    ((CComboBox*)GetDlgItem(IDC_COMBO_SEX))->SetCurSel(0);
    return TRUE;   // return TRUE unless you set the focus to a control
                   // EXCEPTION: OCX Property Pages should return FALSE
}
```

其中 GetDlgItem()函数是 CWnd 类的成员函数，用于获取指定的控件，函数原型如下：

```
CWnd* GetDlgItem( int nID ) const;
```

该函数只有一个参数 nID，这个参数指定要获取的控件的 ID。如果指定的 ID 的控件不存在，则该函数返回 NULL。

函数 SetCurSel()是 CComboBox 类的成员函数，用于在组合框的列表框中选择一个选项显示在其编辑框中。函数原型如下：

```
int SetCurSel( int nSelect );
```

参数 nSelect 是要选择的选项的序号，该序号是从 0 开始索引的，即第一个项目的序号为 0，而不是 1。

由于 SetCurSel()函数是 CComboBox 类的成员函数，因此在取得 ID 为 IDC_COMBO_SEX 的控件之后，即使知道它就是一个组合框，也得用 CComboBox*进行强制转换。运行结果如图 4.31 所示，此时一启动该对话框，组合框中便显示了一个数据选项。

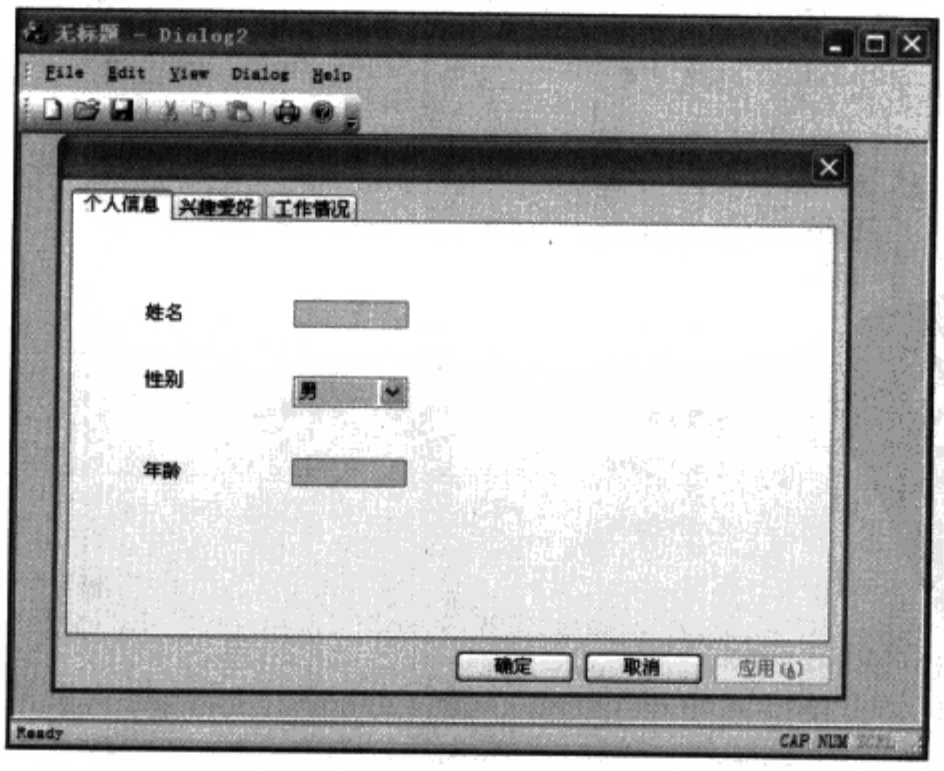


图 4.31　从组合框获取一条数据项

此时，初始化工作并没有结束，属性页对话框的最后一个页面中，“工作地点”列表框中还没有

内容，同样，要在 CPage3 类中添加其 WM_INITDIALOG 消息，在其消息处理函数 OnInitDialog()中添加如下代码：

```
BOOL CPage3::OnInitDialog()
{
    CPropertyPage::OnInitDialog();

    // TODO: Add extra initialization here
    //将城市信息添加到列表框
    ((CListBox*)GetDlgItem(IDC_LIST_CITY))->AddString("北京");
    ((CListBox*)GetDlgItem(IDC_LIST_CITY))->AddString("上海");
    ((CListBox*)GetDlgItem(IDC_LIST_CITY))->AddString("深圳");
    ((CListBox*)GetDlgItem(IDC_LIST_CITY))->AddString("天津");
    //默认第一条信息为高亮度显示
    ((CListBox*)GetDlgItem(IDC_LIST_CITY))->SetCurSel(0);
    ((CButton*)GetDlgItem(IDC_RAD_PRO))->SetCheck(1);
    return TRUE;  // return TRUE unless you set the focus to a control
                  // EXCEPTION: OCX Property Pages should return FALSE
}
```

其中 AddString()是 ClistBox 类的成员函数，负责在列表框或组合框的列表框中添加选项，原型如下：

```
int AddString( LPCTSTR lpszString );
```

该函数只有一个参数 lpszString，用于指定要添加的字符串。如果该函数调用成功，返回的值是添加的字符串在列表框中的序号。

函数 SetCheck()是 CButton 类的成员函数，用于设置按钮的状态，原型如下：

```
void SetCheck( int nCheck );
```

参数 nCheck 指定设置的按钮的状态，当参数为 0 时，表示该按钮不被选中；当参数为 1 时，表示该按钮呈选中状态。本例中，将单选按钮组中的“程序员”单选按钮设为选中状态。

至此，对话框的初始化工作基本完成了。属性页对话框的初始化包括框架对话框的初始化和页的初始化。以上进行的初始化工作是对页进行的初始化，页的初始化工作是在其类的 OnInitDialog()函数中完成的，而框架对话框的初始化应该在 OnCreate()函数中完成，运行结果如图 4.32 所示。

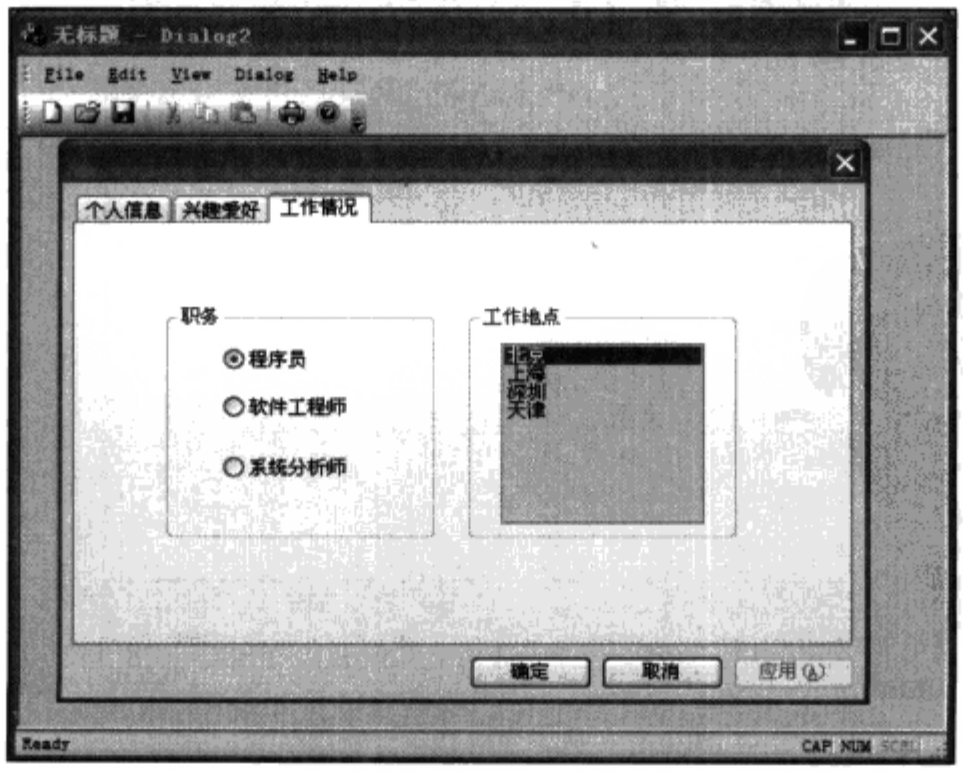


图 4.32　对属性页进行初始化

完成初始化工作之后，就应该实现数据的交换了。

步骤 6：将用户输入的数据显示在屏幕上。

为实现这一功能，至少要在 CDialog2View 类的头文件中定义一些相应的成员变量来保存用户的输入，如下：

```
class CDialog2View : public CView
{ …
 // Implementation
public:
    CString m_name;   // 对应于姓名编辑框的成员变量
    CString m_age;    //对应于年龄编辑框的成员变量
    CString m_city;   //对应于工作地点列表框的成员变量
    CString m_sex;    //对应于性别组合框的成员变量
 …
 };
```

读者应该注意到了，在 CDialog2View 类中定义的成员变量和为控件添加的成员变量同名，这样做是为了方便记忆。

用户输入的数据是否在屏幕上打印出来，取决于用户在关闭对话框时单击的是“确定”按钮还是“取消”按钮，因此这部分代码应该加在菜单命令的 OnProp()函数中通过一个判断语句 if（propSheet.DoModal()==IDOK）来判断用户是单击哪个按钮退出的，OnProp()函数中的代码改为例 4.1 所示的代码。

例 4.1

```
1   void CDialog2View::OnProp()
2   {
3       // TODO: Add your command handler code here
4       CPropertySheet propSheet;
5       CPage1 page1;
6       CPage2 page2;
7       CPage3 page3;
8       propSheet.AddPage(&page1);
9       propSheet.AddPage(&page2);
10      propSheet.AddPage(&page3);

11      if (propSheet.DoModal()==IDOK)  //判断用户是否单击“确定”按钮退出
12      {
13          /****获取并显示用户第一页输入的信息****/
14          m_age=page1.m_age;
15          m_name=page1.m_name;
16          if (page1.m_sex==0)
17              m_sex="男";
18          else
19              m_sex="女";
20          if (m_name.IsEmpty()) //判断用户是否输入姓名
21          {
22              MessageBox("您没输入姓名,不能显示");
23              return;
24          }
```

```
25          if (m_age.IsEmpty())   //判断用户是否输入年龄
26          {
27              MessageBox("您没输入年龄,不能显示");
28              return;
29          }
30          CClientDC dc(this);
31          CString strInput;
32          strInput="姓名: ";
33          strInput+=m_name;
34          strInput+=" , 性别: ";
35          strInput+=m_sex;
36          strInput+=" , 年龄: ";
37          strInput+=m_age;
38          dc.TextOut(0,0,strInput);

39          /****获取并显示用户第二页输入的信息****/
40          strInput="爱好： ";
41          if (page2.m_tbltennis)   //乒乓球复选框是否被选中
42              strInput+=" 乒乓球";
43          if (page2.m_bskball)
44              strInput+=" 篮球";
45          if (page2.m_swim)
46              strInput+=" 游泳";
47          if (page2.m_tennis)
48              strInput+=" 网球";
49          TEXTMETRIC tm;
50          dc.GetTextMetrics(&tm);   //获取当前字体的高度
51          dc.TextOut(0,tm.tmHeight,strInput);

52      /****获取并显示用户第三页输入的信息****/
53          strInput="职务： ";
54          if (page3.m_job==0)
55              strInput+="程序员";
56          if (page3.m_job==1)
57              strInput+="软件工程师";
58          if (page3.m_job==2)
59              strInput+="系统分析师";
60          m_city=page3.m_city;
61          strInput+=" , 工作地点:";
62          strInput+=m_city;
63          dc.TextOut(0,2*tm.tmHeight,strInput);
64      }
```

其中第 14 行代码 m_age=page1.m_age;是将控件的成员变量取出并赋值给 CDialog2View 类的成员变量 m_age，第 25 行代码通过 m_age.IsEmpty()条件判断用户是否输入了年龄，只要年龄和姓名有一项没有输入，就不在屏幕上显示。

第 16 行中 page1.m_sex 是第一页上组合框控件的成员变量，该变量是 int 型，返回的值就是用户在组合框的列表框中选择的选项的序号。在 CDialog2View 类中定义的表示性别的变量 m_sex 是 CString 型，通过判断 page1.m_sex 值为 0 或 1，为 m_sex 赋值男或女。

变量 strInput 是在 CDialog2View::OnProp()中定义的局部变量，用来存放用户输入的信息，显示第一页上的信息之后，获得字体高度，转到下一行再输入第二页上的信息，显示第二页上的信息之后再到下一行显示第三页上的信息，相应的变量 strInput 的值也不断重新赋值。

m_tbltennis、m_bskball、m_swim 和 m_tennis4 个变量为第二页上 4 个复选框控件的成员变量，都是布尔型的，为真时表示相应的爱好被选中，为假时表示没被选中。

第三页上的一组单选按钮控件的成员变量 m_job 为 int 型，表示被选中的单选按钮在单选按钮组中的序号，如果为 0，表示“程序员”单选按钮被选中，依此类推。

程序运行结果如图 4.33 所示。

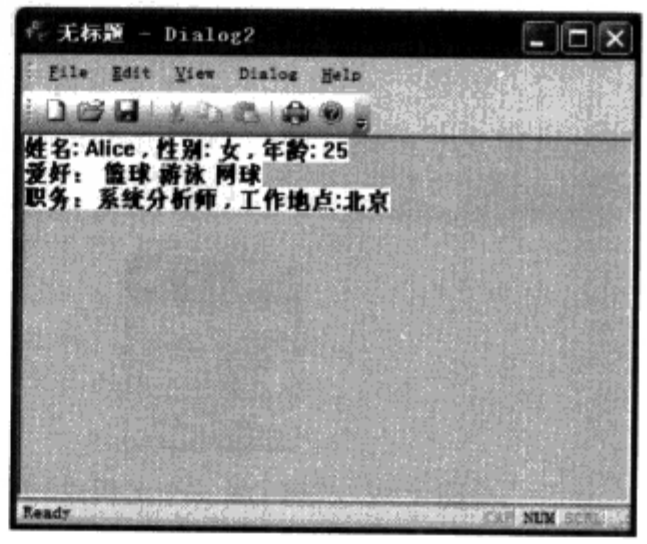


图 4.33　程序运行结果

4.3　消息对话框

消息对话框是一种简单的对话框，不需要开发人员自己创建就可以直接使用，在 Microsoft Visual Studio 2010 中提供了 AfxMessageBox()函数和 MessageBox()函数来弹出消息对话框。

AfxMessageBox()函数和 MessageBox()函数的语法格式为：

```
int AfxMessageBox( LPCTSTR lpszText, UINT nType = MB_OK, UINT nIDHelp = 0 );
int MessageBox( LPCTSTR lpszText, LPCTSTR lpszCaption = NULL, UINT nType = MB_OK );
```

AfxMessageBox()函数和 MessageBox()函数中的参数说明如表 4.4 所示。

表 4.4　　AfxMessageBox 函数和 MessageBox 函数中的参数说明

设　置　值	描　　述
lpszText	消息框中显示的文本，为 NULL 时，使用默认标题
nType	消息框中显示的按钮风格和图标风格的组合，可以使用“|”操作符来组合各种风格
nIDHelp	信息的上下文 ID
lpszCaption	消息框的标题

其按钮风格如表 4.5 所示。

表 4.5　　消息框按钮风格

风　　格	显示的按钮
MB_ABORTRETRYIGNORE	显示“终止”、“重试”、“忽略”按钮
MB_OK	显示“确定”按钮
MB_OKCANCEL	显示“确定”、“取消”按钮
MB_RETRYCANCEL	显示“重试”、“取消”按钮
MB_YESNO	显示“是”、“否”按钮
MB_YESNOCANCEL	显示“是”、“否”、“取消”按钮

其图标风格如表 4.6 所示。

表 4.6　　图标风格

显示的图标	风　　格
	MB_ICONHAND、MB_ICONSTOP、MB_ICONERROR
	MB_ICONQUESTION
	MB_ICONEXCLAMATION、MB_ICONWARNING
	MB_ICONASTERISK、MB_ICONINFORMATION

下面通过一个实例来具体讲解一下消息对话框的创建方法。

（1）首先，创建一个简单的消息框，代码为：

```
AfxMessageBox (_T("Hello"));
```

一个只有“OK”按钮的模态对话框被创建，它等待用户单击，如图 4.34 所示。

（2）创建一个使用应用程序资源文件中字符串表中字符串的简单消息对话框，如图 4.35 所示，即事先已定义了 IDS_HELLO，为#define IDS_HELLO _T("Hello")，用下面的语句也可以创建消息对话框：

```
AfxMessageBox (IDS_HELLO);
```

图 4.34　只有 OK 按钮的模式对话框

图 4.35　只有 OK 按钮的消息框

（3）要动态地编辑字符串表中的字符串，可用 AfxFormatString1 ()函数：

```
AfxFormatString1(msg, IDS_HELLO_1, "Goodbye");
AfxMessageBox(msg);
```

本例中，IDS_HELLO_1 代表字符串表中的“Hello %1”字符串， AfxFormatString1()用%1 替换“Goodbye”，这种方法允许用户保存绝大多数的消息字符串在字符串表中，这样可以使应用程序被轻松地转变为另一种语言，如西班牙语或法语，只需用另一个字符串表替换一个字符串表，即可把该字符串表放入资源库中。

（4）AfxFormatString2()函数替换两个字符串而不是一个，在下面的代码中，消息字符串是“Hello %1 %2”。

```
AfxFormatString2(msg, IDS_HELLO_2, "and", "Goodbye") ;
AfxMessageBox(msg);
```

（5）要询问 Yes 或 No 问题，应该添加一个风格标志到 AfxMessageBox()函数中。

```
if (AfxMessageBox(msg, MB_YESNO)==IDYES){}
```

其他可能的风格包括 MB_ABORTRETRYIGNORE、MB_OKCANCEL、MB_RETRYCANCEL、MB_YESNO 和 MB_YESNOCANCEL，其他可能的选择测试包括 IDNO、IDSABORT、IDCANCEL、IDSIGNORE、IDOK 和 IDRETRY。

（6）要使一个按钮（除了第一个按钮外）成为默认的按钮（当用户按【Enters】键时被单击），用 MB_DEFBUTTON2 或 MB_DEFBUTTON3 与风格标志进行“或”操作。

```
if (AfxMessageBox(msg,MB_YESNO|MB_DEFBUTTO N 2 ) = = IDYES ){
}
```

（7）要在消息对话框中显示除了感叹号以外的图标，可以用 MB_ICONSTOP、MB_ICONINFORMATION 或 MB_ICONQUESTION 与风格标志进行“或”操作。

```
AfxMessageBox (msg,MB_ICONSTO P );
```

用 AfxFormatString1 ()和 AfxFormatString2 ()使用的%1 和%2 标志可以以任意顺序出现，也可以按照每次用 CwinApp::DoMessageBox ()执行 AfxMessageBox ();来截获应用程序创建的所有消息对话框，只要用 ClassWizard 重载应用程序类的 DoMessageBox ()函数，在该函数中，可以给消息添加一个标准标题、时间，甚至把消息保存到日志中。

AfxMessageBox()是一种让最新输入的应用程序立即显示消息的非常好的方法。通常，在一个窗口应用程序中，必须拥有或指定一个窗口以创建另一个窗口，所有应用程序的窗口都属于应用程序主窗口，而它自身又属于桌面窗口，使用这种方法，所有窗口都可以被窗口管理器刷新。但是如果使用

AfxMessageBox()，则类成员函数或者调用该类成员函数的静态函数不需要拥有一个窗口，AfxMessageBox()会自动查找并使用应用程序的主窗口作为其物主窗口。

因此，即使一个普通的数据类也可以使用 AfxMessageBox ()显示一个消息（虽然数据类通常不显示消息），惟一的要求是把 Stdafx.h 包括在使用 AfxMessageBox ()的源文件中。如果那样有问题，可以用 Windows API 调用。

```
::MessageBox ( );
```

这时必须包括“Windows.h”文件。

下面在关闭应用程序时使用消息对话框进行确认。程序设计步骤如下。

（1）创建一个基于对话框的应用程序，将对话框的 Caption 属性修改为“应用程序”。

（2）删除对话框中自动生成的控件，为对话框处理 WM_CLOSE 消息，在该消息的响应函数中设置弹出消息对话框的功能。代码如下：

```
void CMessageDlg::OnClose() //WM_CLOSE 消息响应函数
{
    //判断是否单击“确定”按钮
    if(MessageBox(_T("确定要退出应用程序吗？"),LPCTSTR(_T("系统提示"), MB_OKCANCEL| MB_ICONQUESTION)!
=IDOK)
    return;  //用户单击“取消”按钮时不退出
    CDialog::OnClose(); //退出程序
}
```

运行结果如图 4.36 所示。

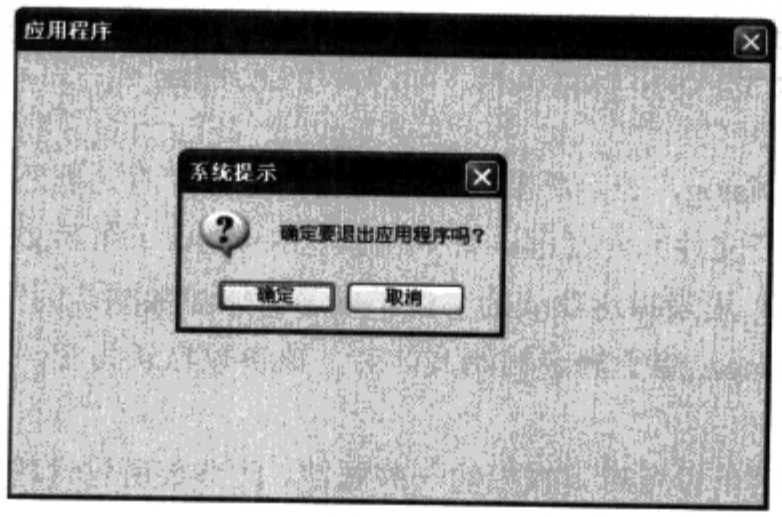


图 4.36　弹出消息框的运行结果

4.4　通用对话框

在绝大部分的 Windows 应用程序中，有一些对话框是常用的，如打开文件对话框、保存对话框、字体对话框、打印对话框、查找对话框和颜色对话框等，这些对话框在不同的应用程序中具有相同的功能及类似的界面，因此 Windows 系统本身提供了一组这样的通用对话框，使用户不必再设计这些对话框了。

对应于这 5 种对话框，MFC 提供了相应的公用对话框类，公用对话框类包括以下几种。

- CColorDialog 类：用于选择颜色对话框。
- CFileDialog 类：用于实现打开和保存文件对话框。
- CFindReplaceDialog 类：用于查找和替换文本。

- CFontDialog 类：用于选择字体对话框。
- COleDialog 类：该类及其派生类用于生成 OLE 应用程序对话框。
- CpageSetupDialog 类：OLE Page Setup（指定页面设置和打印边界）。
- CPrintDialog 类：用于打印和打印设置对话框。

这些公用对话框类都是由 CDialog 类的派生的，它们封装了公用对话框的功能。本节就将介绍其中的文件打开和文件保存对话框、颜色对话框、字体对话框、查找对话框、页面设置对话框及打印对话框。

4.4.1 文件打开对话框

CFileDialog 类用于实现文件选择对话框，以支持文件的打开和保存操作。只要用户要打开或保存文件，就会和文件选择对话框打交道，图 4.37 显示了一个标准的用于打开文件的文件选择对话框。用 MFC AppWizard 建立的应用程序中自动加入了文件选择对话框，在“File”菜单中选择“Open”或“Save As”命令就会启动它们。

文件保存和文件打开对话框的创建与一般对话框的创建过程很相似：首先是在堆栈上构建一个 CFileDialog 类的对象，然后再调用 CFileDialog::DoModal()函数创建并显示对话框。

对于将要创建的保存、打开对话框属性的设置，可以在构造 CFileDialog 类的对象时通过其构造函数实现，也可以通过 CFileDialog 类的数据成员 m_ofn 来实现。

CFileDialog 的构造函数如下：

```
CFileDialog( BOOL bOpenFileDialog, LPCTSTR lpszDefExt = NULL, LPCTSTR lpszFileName = NULL, DWORD dwFlags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT, LPCTSTR lpszFilter = NULL, CWnd* pParentWnd = NULL );
```

第一个参数 bOpenFileDialog，表示将要构造的对话框是打开对话框还是保存对话框。如果该参数为 True，将创建 Open（打开文件）对话框，否则就创建 Save As（保存文件）对话框。

第二个参数 lpszDefExt，用来指定默认的文件扩展名。该参数默认为空，表示不给出默认扩展名。

第三个参数 lpszFileName，用于规定初始文件名。该参数默认为空，表示在文件名编辑框中不给出初始文件名。

第四个参数 dwFlags，用于设置对话框中的一些属性。此参数所包含的属性值和 OPENFILENAME 结构中的 Flags 数据成员的值一样，请参见 MSDN。

第五个参数 lpszFilter，用于指向一个过滤字符串，如果用户只想选择某种或某几种类型的文件，就需要指定过滤字符串。

第六个参数 pParentWnd，是指向父窗口或拥有者窗口的指针。

OPENFILENAME 是一个很重要的结构，里面有很多数据成员，如 lpStrFilter、lpStrTitle、lpStrFile 等，这些成员包含着和文件选择对话框相关的信息。例如，当用户要创建一个打开或保存对话框时，可以通过 OPENFILENAME 结构来设置对话框的初始状态及默认的文件名等信息；当用户关闭对话框时，用户在对话框中选择的文件名等信息就会返回到这个结构中。CFileDialog 类的数据成员 m_ofn 就是一个 OPENFILENAME 结构的对象，下面通过一个实例来介绍这个数据成员及文件选择对话框的使用方法。

创建一个新的工程 Dialog4，以工程 Dialog4 为基础，在其菜单资源中再添加一个名为“文件对话框”的主菜单，然后给这个菜单添加两个菜单命令，分别为“打开文件对话框”和“保存文件对话框”，ID 分别为 ID_FILEIO_DLG_OPEN 和 ID_FILEIO_DLG_SAVE，结果如图 4.38 所示。

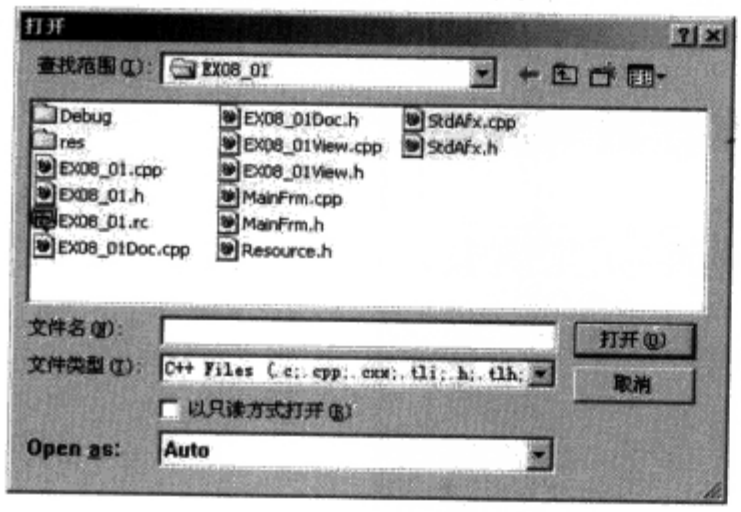


图 4.37　文件打开对话框

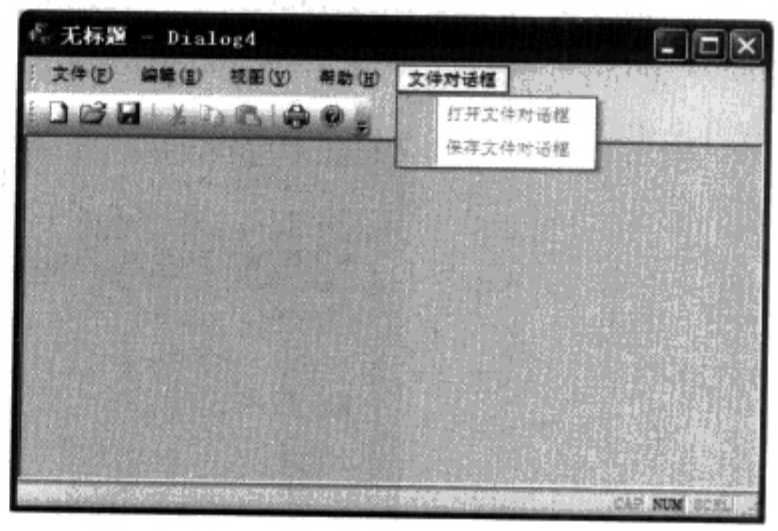


图 4.38　添加文件打开及保存菜单项

在 CDialog4View 类中再添加上述菜单命令“打开文件对话框”的 COMMAND 命令消息处理函数，为 OnFileioDlgOpen()，并添加如下代码：

```
void CDialog4View::OnFileioDlgOpen()
{
    // TODO: Add your command handler code here
    CFileDialog fileDlg(TRUE);
    fileDlg.m_ofn.lpstrFilter=LPCTSTR(_T("Text Files(*.txt)\0*.txt\0All Files(*.*)\0*.*\0\0"));
    //定义打开对话框的标题
    fileDlg.m_ofn.lpstrTitle=LPCTSTR(_T("打开文件"));
    char szchar[20];
    memset(szchar,0,20);
    if (fileDlg.DoModal()==IDOK)
    {
        CFile file(fileDlg.m_ofn.lpstrFile,CFile::modeRead);
        file.Read(szchar,20);
        file.Close();
        MessageBox(LPCTSTR(szchar));
    }
}
```

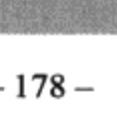
首先构建一个 CFileDialog 类的对象 fileDlg，设定构造函数中的第一个参数为 True，表示创建的是打开文件对话框，定义对话框的标题为“打开文件”。其中，lpstrFile 同样包含的是用户在单击“确定”按钮时实际打开的文件名。

消息处理函数的第二行代码，是在对话框显示之前设置过滤器。m_ofn 是 OPENFILENAME 结构类型的对象，lpstrFilter 是该结构的一个数据成员，用于指定包含一种或多种文件类型的过滤字符串，过滤字符串有特定的格式，它实际上由多个子串组成，每个子串由两部分组成，第一部分是过滤器的字面说明，如“Text files (*.txt)”，第二部分是用于过滤的匹配字符串，如“*.txt”，子串的两部分用“\0”或竖线字符“|”分隔开。各子串之间也要用“\0”或“|”分隔，且整个串的最后两个字符必须是两个连续的“\0”或竖线字符“||”。如果不设置该数据成员的值，或在构造函数中默认为 NULL，表示对话框不显示任何过滤器。

其中 lpstrTitle 和 lpstrFile 也是结构 OPENFILENAME 的数据成员，lpstrTitle 可以用于设置显示的对话框的标题，如本例中设置标题为“打开文件”；lpstrFile 包含文件名信息，在对话框启动前，如果设置 lpstrFile 的值，表示初始对话框上显示的默认的文件名；如果单击“确定”按钮关闭对话框，那么用户实际保存的文件名会返回到 lpstrFile 中。编译并运行程序，选择“文件对话框”→“打开文件

对话框”命令，结果如图 4.39 所示。可以看到，文件类型中所设置的两个过滤器：文本文件和所有文件。

图 4.39 设置打开文件类型过滤器的运行结果



虽然对话框上设置的默认过滤器为.txt，但是打开文件的时候，如果真的要打开扩展名为.txt 的文本文件，还是要自己在文件名后手动添加上扩展名.txt。

4.4.2 文件保存对话框

仍然以上一节创建的工程 Dialog4 为基础，在工程 Dialog4 的菜单资源中已经添加了主菜单“文件对话框”，并且已经加入了菜单命令“保存文件对话框”，ID 为 ID_FILEIO_DLG_SAVE，如图 4.40 所示。

图 4.40 文件保存对话框

在 CDialog4View 类中再添加“保存文件对话框”菜单命令的 COMMAND 命令消息处理函数，为 OnFileioDlgSave()，并添加如下代码：

```
void CEX08_01View::OnFileioDlgSave()
{
    // TODO: Add your command handler code here
    CFileDialog fileDlg(FALSE);
    //设置保存对话框的初始显示样式
    fileDlg.m_ofn.lpstrFilter=LPCTSTR(_T("Text Files(*.txt)\0*.txt\0All Files(*.*)\0*.*\0\0"));
    //定义保存对话框的标题
    fileDlg.m_ofn.lpstrTitle=LPCTSTR(_T("保存文件"));
    char szchar[20]="利用对话框";
    if (fileDlg.DoModal()==IDOK)
    {
        CFile file(fileDlg.m_ofn.lpstrFile,CFile::modeCreate|CFile::modeWrite);
        file.Write(szchar,strlen(szchar));
        file.Close();
        MessageBox(LPCTSTR("文件已保存"));
```

```
        }
    }
```

首先构建一个 CFileDialog 类的对象 fileDlg，其构造函数中的第一个参数为 False，表示创建的是保存对话框。

消息处理函数的第二行代码是在对话框显示之前设置过滤器。m_ofn 是 OPENFILENAME 结构类型的对象，lpstrFilter 是该结构的一个数据成员，用于指定包含一种或多种文件类型的过滤字符串，过滤字符串有特定的格式，它实际上由多个子串组成，每个子串由两部分组成，第一部分是过滤器的字面说明，如“Text files (*.txt)”，第二部分是用于过滤的匹配字符串，如“*.txt”，子串的两部分用“\0”或竖线字符“|”分隔开。各子串之间也要用“\0”或“|”分隔，且整个串的最后两个字符必须是两个连续的“\0”或竖线字符“||”。如果不设置该数据成员的值，或在构造函数中默认为 NULL，表示对话框不显示任何过滤器。

其中 lpstrTitle 和 lpstrFile 也是结构 OPENFILENAME 的数据成员，lpstrTitle 可以用于设置显示的对话框的标题，如本例中设置标题为“保存文件”；lpstrFile 包含文件名信息，在对话框启动前，如果设置 lpstrFile 的值，表示初始对话框上显示的默认的文件名；如果单击“确定”按钮关闭对话框，那么用户实际保存的文件名会返回到 lpstrFile 中。编译并运行程序，选择菜单中的“保存对话框”命令，结果如图 4.41 所示。可以看到，保存类型中所自己设置的两个过滤器：文本文件、所有文件。

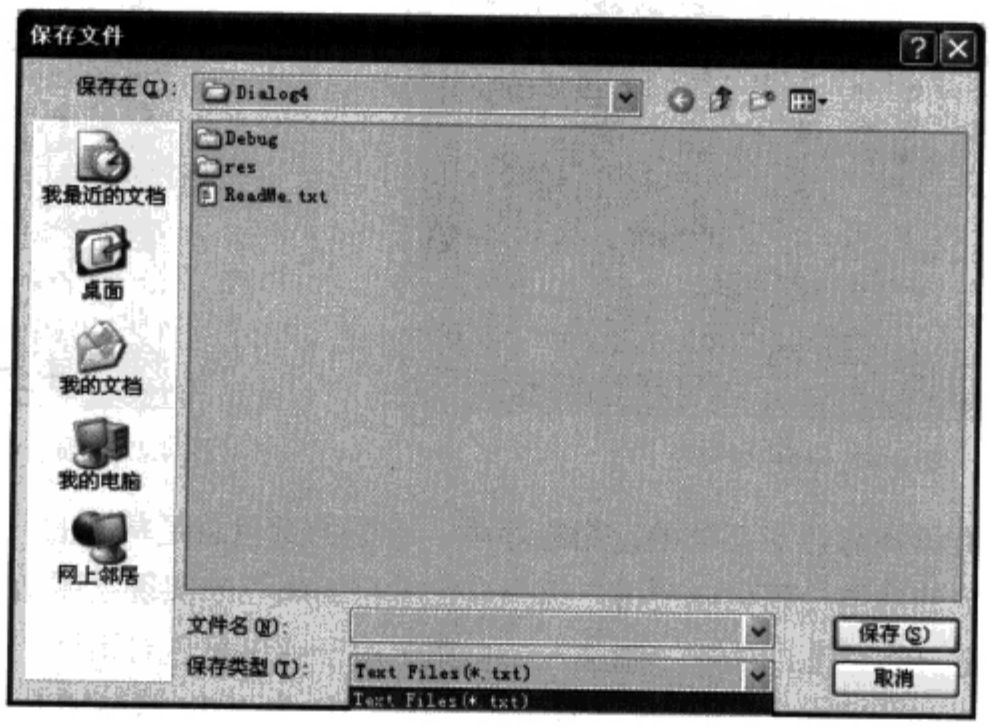


图 4.41　设置保存文件过滤器的运行结果

虽然对话框中设置的默认过滤器为.txt，但是保存文件的时候，如果真的要保存为文本文件，还是要自己在文件名后面手动添加上扩展名.txt。

4.4.3　颜色对话框

CColorDialog 类用于实现颜色选择的公用对话框。Windows 的画图程序中就有一个颜色对话框，一个标准的颜色对话框如图 4.42 所示。

颜色对话框的创建与一般对话框的创建过程很相似：首先是在堆栈上构建一个 CColorDialog 类的对象，然后再调用 CColorDialog::DoModal()来创建并显示对话框。

对于将要创建的颜色对话框属性的设置可以在构造 CColorDialog 类的对象时通过其构造函数给出，也可以通过 CColorDialog 类的数据成员 m_cc 来设置。

CColorDialog 的构造函数如下：

```
CColorDialog( COLORREF clrInit = 0, DWORD dwFlags = 0, CWnd* pParentWnd = NULL );
```

第一个参数 clrInit 用来指定初始的颜色选择，如果该参数没有指定颜色，那么默认的颜色为 RGB(0,0,0)，即黑色。

第二个参数 dwFlags 用来设置对话框的功能及显示样式等属性，此参数所包含的属性值和 CHOOSECOLOR 结构中的 Flags 数据成员的值一样，最常用的属性为 CC_FULLOPEN，颜色对话框通常有两种显示样式，一种是如图 4.42 所示的包含完整的颜色设置的对话框样式；另一种是如图 4.43 所示的简易的只包含给定的标准颜色设置的对话框样式。而属性 CC_FULLOPEN 就是将颜色对话框设置为第一种的完整形式。

第三个参数 pParentWnd 用于指定对话框的父窗口或拥有者窗口，默认为 NULL。

CHOOSECOLOR 是一个很重要的结构，其中有很多成员，如 lStructSize、rgbResult、Flags 等，这些成员包含着和颜色对话框相关的信息。例如，当用户要创建一个颜色对话框时，可以通过 CHOOSECOLOR 结构来设置颜色对话框的初始状态；当用户关闭颜色对话框时，在对话框中选择的颜色等信息就会返回到这个结构中。CColorDialog 类的数据成员 m_cc 就是一个 CHOOSECOLOR 结构的对象，下面通过一个实例来介绍这个数据成员及颜色对话框的使用方法。

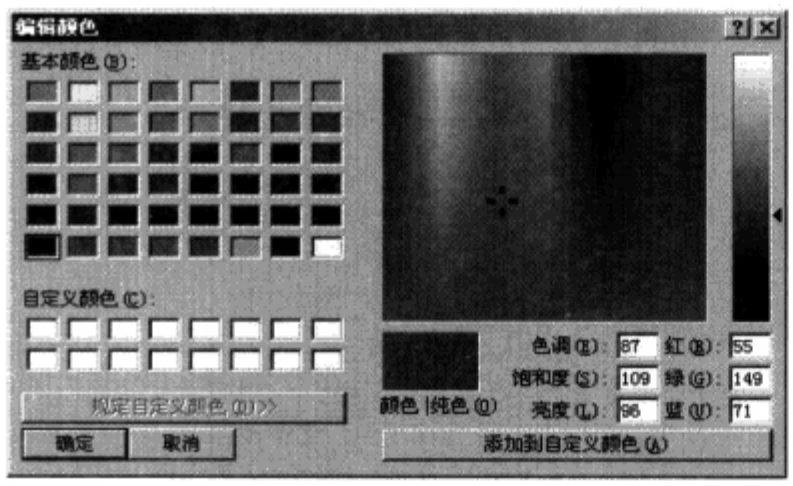


图 4.42　颜色对话框

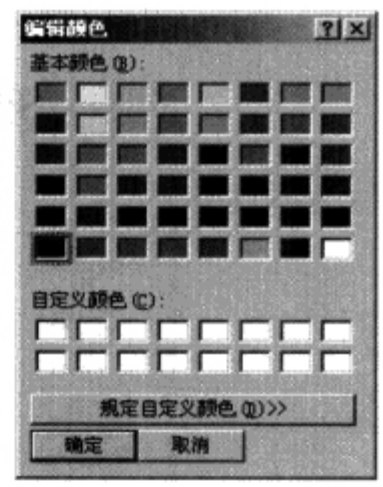


图 4.43　颜色对话框（简易显示）

一般而言，在没有添加颜色对话框的绘图程序中，绘图的颜色总是单一的，看起来很单调，如果给它添加一个颜色设置的功能，则可以使用户选择自己希望的颜色来实现绘图功能。具体操作如下。

步骤 1：首先新建一个工程 Dialog5，添加一个主菜单用于弹出颜色对话框。

打开工作台的 ResourceView 页面，在资源面板中修改 ID 为 IDR_MAINFRAME 的菜单资源，在主菜单给出的默认菜单的后面再添加一个主菜单“颜色设置”，其 ID 为 ID_GRAPH_CLRSET，在 CDialog5View 类中添加该菜单的 COMMAND 命令处理函数 OnGraphClrset()函数，在该函数中创建及显示颜色对话框，代码如下：

```
void CDialog5View::OnGraphClrset()
{
    // TODO: Add your command handler code here
    CColorDialog m_setClrDlg;
    m_setClrDlg.m_cc.Flags|=CC_FULLOPEN;
    m_setClrDlg.DoModal();
}
```

其中 m_setClrDlg.m_cc.Flags|=CC_FULLOPEN;这句代码就是设置弹出的颜色对话框的初始状态为完整显示，代码中的“=”前面有一个“或”符号“|”，这个符号必须加在等号前面，否则会出错。

运行程序，即使用户选择了颜色，在绘图时使用的颜色仍然只会是单一的，这是因为还没有将用户选择的颜色保存下来。

步骤 2：将用户设置的颜色保存下来。

首先，应该在 CDialog5View 类的头文件中声明一个 COLORREF 类型的变量 m_clr 来保存用户设置的颜色，其中 COLORREF 是一个 32 位的值，用来说明一个 RGB 颜色。

然后，在 CDialog5View 类的构造函数中对变量 m_clr 进行初始化，初始化为淡蓝色，如下：

```
m_clr=RGB(0,255,255);
```

最后，在关闭颜色对话框时将颜色保存下来，根据 DoModal()返回的是 IDOK 还是 IDCANCEL 可知道用户是否确认了对颜色的选择。因此，应该在菜单命令处理函数 OnGraphClrset()中修改代码如下：

```
void CDialog5View::OnGraphClrset()
{
    // TODO: Add your command handler code here
    CColorDialog m_setClrDlg;
    m_setClrDlg.m_cc.Flags|=CC_FULLOPEN;
    if (m_setClrDlg.DoModal()==IDOK)
    {
        m_clr=m_setClrDlg.m_cc.rgbResult;
    }
}
```

其中，rgbResult 中保存的就是用户在颜色对话框中选择的颜色值，将该值保存到 CDialog5View 类的 m_clr 变量中。

步骤 3：将颜色应用到绘图中。

将颜色保存下来之后，要用这个颜色来创建画笔。修改 CDialog5View::OnLButtonUp()函数，将其中生成画笔的代码改为如下形式：

```
CPen newpen(m_style,m_width,m_clr);
```

运行绘图程序，可以发现，用户选择什么颜色，绘出的图形就是什么颜色，看起来还不错，但还需要加一项功能，即在应用程序不退出的情况，再次打开颜色对话框时，对话框上显示的应该是用户上次设置的颜色信息。修改 OnGraphClrset()函数为如下代码：

```
void CDialog5View::OnGraphClrset()
{
    // TODO: Add your command handler code here
    CColorDialog m_setClrDlg;
    //设置颜色对话框的初始显示样式
    m_setClrDlg.m_cc.Flags|=CC_FULLOPEN|CC_RGBINIT;
        m_setClrDlg.m_cc.rgbResult=m_clr;
        if (m_setClrDlg.DoModal()==IDOK)
        {
            m_clr=m_setClrDlg.m_cc.rgbResult;
        }
}
```

在设置 Flags 时，多了一个 CC_RGBINIT 属性，只有将该属性添加进来，才可以将用户设置的颜色在下一次打开颜色对话框时显示出来作为初始颜色。在创建颜色对话框之前，可以把要设置的颜色对话框的初始颜色值存放到 rgbResult 中。可见 rgbResult 的用途有两种：在创建颜色对话框之前，存放对话框的初始颜色值；在关闭对话框后，保存用户设置的颜色值。

4.4.4　字体对话框

CFontDialog 类支持字体选择对话框，用于让用户选择字体。Microsoft Word 应用软件中就有字体

对话框，图 4.44 所示为一个字体对话框。

字体对话框的创建过程与颜色对话框类似：首先是在堆栈上构建一个 CFontDialog 对象，然后调用 CFontDialog::DoModal()来启动对话框。

CFontDialog 类的构造函数如下：

```
CFontDialog( LPLOGFONT lplfInitial = NULL, DWORD dwFlags = CF_EFFECTS | CF_SCREENFONTS, CDC* pdcPrinter = NULL, CWnd* pParentWnd = NULL );
```

第一个参数 lplfInitial 指向一个 LOGFONG 结构，用来初始化对话框中的字体设置。

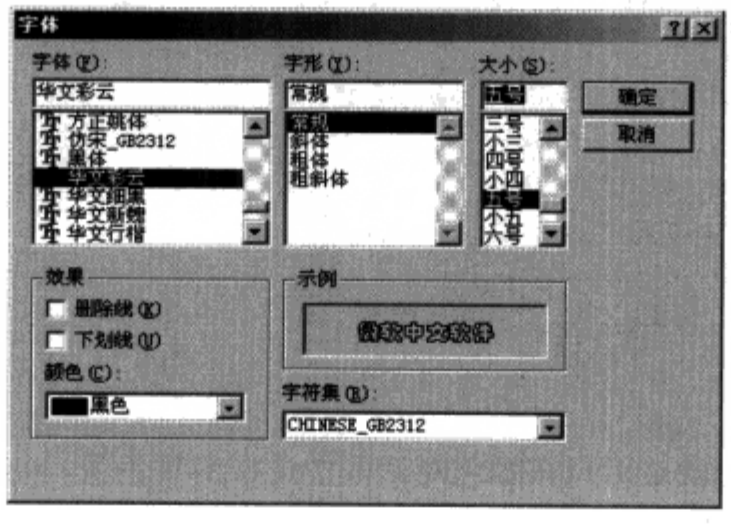


图 4.44　字体对话框

第二个参数 dwFlags 用于设置对话框，此参数所包含的属性值和 CHOOSEFONT 结构中的 Flags 数据成员的值相同。

第三个参数 pdcPrinter 用于指向一个代表打印机的 CDC 对象，若设置该参数，则选择的字体就为打印机所用。

第四个参数 pParentWnd 用于指定对话框的父窗口或拥有者窗口。

CHOOSEFONT 是一个很重要的结构，其中有很多成员，如 lpLogFont、lpszStyle、Flags 等，这些成员包含着和字体对话框相关的信息。在创建一个字体对话框前，用户可以通过 CHOOSEFONT 结构来设置字体对话框的初始状态；当关闭字体对话框时，用户在对话框中设置的字体等信息就会返回到这个结构中。CFontDialog 类的数据成员 m_cf 就是一个 CHOOSEFONT 结构的对象，下面通过一个实例来介绍这个数据成员及字体对话框的使用方法。继续在工程 Dialog5 的基础上添加一个设置字体的对话框，用户设置字体后，当关闭对话框时，设置的字体名称显示在窗口上。具体操作如下。

步骤 1：打开工程 Dialog5,在 CDialog5View 类中声明一个 CFont 类型的变量 m_font，用来保存用户在字体对话框中设置的字体。

步骤 2：在 CDialog5View 类中声明一个 CString 类型的变量 m_fontName，用来保存用户在字体对话框中设置的字体的名称。

步骤 3：在 CDialog5View 类添加一个菜单命令，用于弹出字体对话框并保存返回值。

打开工作台的 ResourceView 页面，在资源面板中修改 ID 为 IDR_MAINFRAME 的菜单资源，再添加一个菜单“字体设置”，其 ID 为 ID_GRAPH_FONTSET，在 CDialog5View 类中添加该菜单的 COMMAND 命令处理函数 OnGraphFontset()，在该函数中创建及显示字体对话框并保存返回的字体及字体名称，代码如下：

```
void CDialog5View::OnGraphFontset()
{
    // TODO: Add your command handler code here
```

```
    CFontDialog m_setfontDlg;
    //如果已经产生字体，那么为创建新的字体，将原来的字体删除
    if (m_font.m_hObject)
        m_font.DeleteObject();
        if (m_setfontDlg.DoModal()==IDOK)
        {
            m_font.CreateFontIndirect(m_setfontDlg.m_cf.lpLogFont);
            m_fontName=m_setfontDlg.m_cf.lpLogFont->lfFaceName;
            Invalidate();
        }
}
```

当用户单击“确定”按钮关闭字体对话框时，用户设置的字体就会保存在 lpLogFont 中，lpLogFont 是结构体 CHOOSEFONT 中的一个数据成员，而 lpLogFont 本身又是一个指向 LOGFONT 结构的指针，LOGFONT 结构中包含着返回的字体的相关信息，如字体名称 lfFaceName、是否具有下划线 lfUnderline、字体高度 lfHeight 等。

最后调用 Invalidate()重绘窗口。

步骤 4：在 CDialog5View::OnDraw()函数中将字体载入并显示其名称。

```
void CDialog5View::OnDraw(CDC* pDC)
{
    CEX06_00Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    CFont* poldFont=pDC->SelectObject(&m_font);
    pDC->TextOut(0,0,m_fontName);
    pDC->SelectObject(poldFont);
}
```

4.4.5　查找对话框

打开一个文本编辑器，按【Ctrl + F】组合键打开查找对话框，按【Ctrl + H】组合键打开查找和替换对话框，在 Windows 中定义了两个结构体和两个函数，使开发人员在编程的时候能够很方便地创建这两个对话框。

这两个对话框与系统中其他的对话框不同，这是非模态的对话框，允许用户在主窗口和这两个对话框之间来回切换。

CFindReplaceDialog 类支持创建查找对话框，Microsoft Word 应用软件中也有查找对话框，图 4.45 所示就是一个查找和替换对话框。

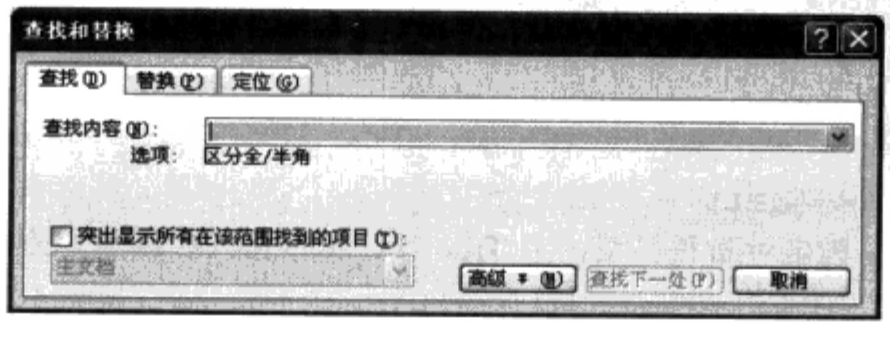


图 4.45　查找对话框

在创建对话框之前，必须先调用 RegisterWindowMessage()注册一个 FINDMSGSTRING 类型的消

息，获得消息的 ID 号。

FINDREPLACE 结构体：

```
typedef struct {
    DWORD lStructSize;
    HWND hwndOwner;
    HINSTANCE hInstance;
    DWORD Flags;
    LPTSTR lpstrFindWhat;
    LPTSTR lpstrReplaceWith;
    WORD wFindWhatLen;
    WORD wReplaceWithLen;
    LPARAM lCustData;
    LPFRHOOKPROC lpfnHook;
    LPCTSTR lpTemplateName;
} FINDREPLACE, *LPFINDREPLACE;
```

- lStructSize：指定结构体的大小，用 sizeof(FINDREPLACE)初始化。
- hwndOwner：指定创建查找和替换对话框的窗口的句柄，也是接收之前注册的消息的对话框的句柄。
- lpstrFindWhat：指定查找的字符串，使用动态分配的全局或者静态字符数组。
- lpstrReplaceWith：指定替换成的字符串。
- wFindWhatLen：指定查找字符串的长度。
- wReplaceWithLen：指定替换成的字符串的长度。
- hInstance、lCustData、lpfnHook、lpTemplateName：如果不使用复杂的钩子，则可以忽略这 4 个参数。

如果 Flags 设置了 FR_DOWN 位，则 DOWN 单选按钮被选中；

如果用户单击了对话框中的“查找下一个”按钮，在发送消息通知创建对话框窗口之前，会设置 Flags 位，在消息处理函数中可以通过 Flags 位来判断是否单击了“查找下一个”按钮。

配置了结构体以后，作为非模态对话框，必须用 new 操作符分配存储空间，再用 Create()函数进行初始化，最后用 ShowWindow()函数显示对话框。

下面的操作就可以创建一个查找对话框：先在菜单资源中用 ID 号 ID_EDIT_FIND 添加一个“查找”菜单命令，再用 MFC Event Handler Wizard 添加其响应函数 OnEditFind()，加入代码：

```
void CDialog6View::OnEditFind()
{
    CFindReplaceDialog* pDlg = new CFindReplaceDialog();
    pDlg->Create( true, NULL, NULL, FR_DOWN, this );   //创建查找对话框
    pDlg->ShowWindow( SW_SHOW );    //显示对话框
}
```

Create()函数的原型如下：

```
BOOL Create(BOOL bFindDialogOnly,LPCTSTR lpszFindWhat,LPCTSTR lpszReplaceWhat=NULL,DWORD dwFlag=
FR_DOWN，CWnd* pParentWnd=NULL);
```

- bFindDialogOnly：指定对话框类型，为 True 时，显示查找对话框；为 False 时，显示查找和替换对话框。
- lpszFindWhat：指定在查找框中显示的字符串。
- lpszReplaceWhat：指定在替换框中显示的字符串。

- dwFlag：指定标志位，用来定制对话框，它可以是一个或多个标志的组合，主要取值如下：

FR_DOWN 如果设置，对话框中的“向下”单选按钮被选中，否则“向上”单选按钮被选中；

FR_HIDEUPDOWN 不显示查找方向选项；

FR_HIDEMATCHCASE 不显示“区分大小写”复选框；

FR_HIDEWHOLEWORD 不显示“全字匹配”复选框；

FR_MATCHCASE 使“区分大小写”复选框处于选中状态；

FR_WHOLEWORD 使“全字匹配”复选框处于选中状态；

FR_NOMATCHCASE 使“区分大小写”复选框处于禁止（变灰）状态；

FR_NOUPDOWN 使查找方向选项处于禁止（变灰）状态；

FR_NOWHOLEWORD 使“全字匹配”复选框处于禁止（变灰）状态；

FR_SHOWHELP 在对话框中显示一个帮助按钮。

- pParentWnd：指向对话框的父窗口，如果为 NULL，则为主框架窗口，使用时需要让它指向接收查找和替换消息的窗口。

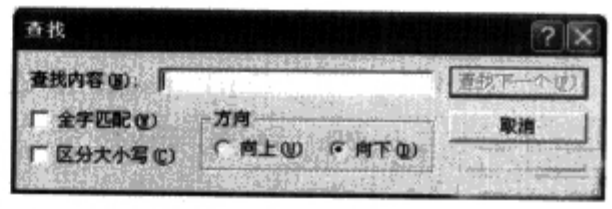


图 4.46　查找对话框运行结果

本例中定制的是一个查找对话框，如图 4.46 所示。

如果定制一个查找和替换对话框，可以添加 OnEditReplace()函数：

```
//替换菜单响应函数
void CDialog6View::OnEditReplace()
{
    CFindReplaceDialog* pDlg = new CFindReplaceDialog();
    pDlg->Create( false, m_FindString, m_ReplaceString, FR_DOWN, this );//创建查找和替换对话框
    pDlg->ShowWindow( SW_SHOW );    //显示对话框
}
```

4.4.6　页面设置对话框

“页面设置”对话框提供了一个可以方便地设置页面属性的用户界面，可以通过它来设置纸张、页边距、打印方向等，也可以用它来打开“打印设置”对话框。如图 4.47 所示为标准的“页面设置”对话框。

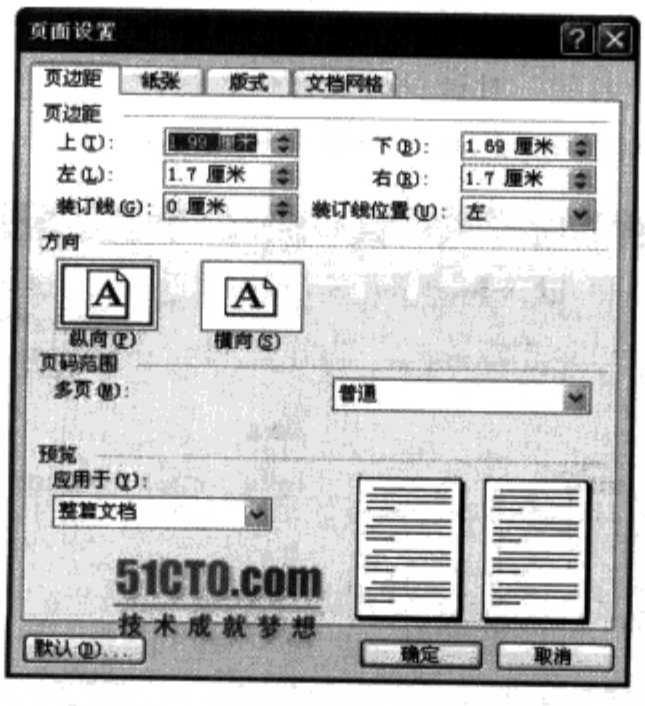


图 4.47　页面设置对话框

MFC 的 CPageSetupDialog 类对“页面设置”对话框进行了封装，在程序中可以使用带参数的构造函数进行“页面设置”对话框的创建，该函数原型如下：

```
CPageSetupDialog(DWORD dwFlags=PSD_MARGINS|PSD_INWININIINTLMEASURE, CWnd* pParentWnd= NULL);
```

其中 dwFlags 是用来定制对话框设置的标志，pParentWnd 是指向对话框的父窗口或主窗口的指针。

CPageSetupDialog 类提供了一个 PAGESETUPDIALOG 结构的成员变量 m_psd，用户可以通过它来更改打印机设置。

例如，创建一个名为 Dialog7 的工程，并在其中创建一个对话框，在对话框中添加一个按钮控件，将其 Caption 属性设置为“页面设置”，将其 ID 设置为 IDC_BUTPAGE，然后在该按钮上右击，并选择“Add Event Handler”命令，在弹出的对话框中将消息名命名为 OnButpage，类名为 CDialog7Dlg，单击“Finish”按钮完成消息创建，页面会自动进入 CDialog7Dlg 中，鼠标指针定位到 CDialog7Dlg::OnButpage()函数处。

在 CDialog7Dlg::OnButpage()函数中输入以下代码：

```
void CDialog7Dlg::OnButpage()
{
    CPageSetupDialog dlg;                    //利用默认参数构造页面设置对话框
    if(dlg.DoModal())
    {
        PAGESETUPDLG psd=dlg.m_psd;  //获取信息
        CString str;
        str.Format("页面大小为：%d",psd.ptPagerSize.x,psd.ptPagerSize.y);
        MessageBox(str, "页面设置");
    }
}
```

这段程序代码将创建一个“页面设置”对话框，并将设置的页面大小通过消息对话框显示出来。

4.4.7 打印对话框

使用 CPrintDialog()函数可以构造打印或打印设置对话框。语法如下：

```
CPrintDialog( BOOL bPrintSetupOnly,DWORD dwFlags=PD_ALLPAGES|PD_USEDEVMODECOPIES| PD_NOPAGENUMS
|PD_HIDEPRINTTOFILE | PD_NOSELECTION, CWnd* pParentWnd = NULL );
```

- bPrintSetupOnly：为 True，表示创建“打印设置”对话框；为 False，表示创建“打印”对话框。
- dwFlags：自定义对话框的一组标记。
- pParentWnd：表示指向“打印”对话框的父窗口的指针。

打印对话框使用 CPrintDialog()方法创建，如图 4.48 所示。

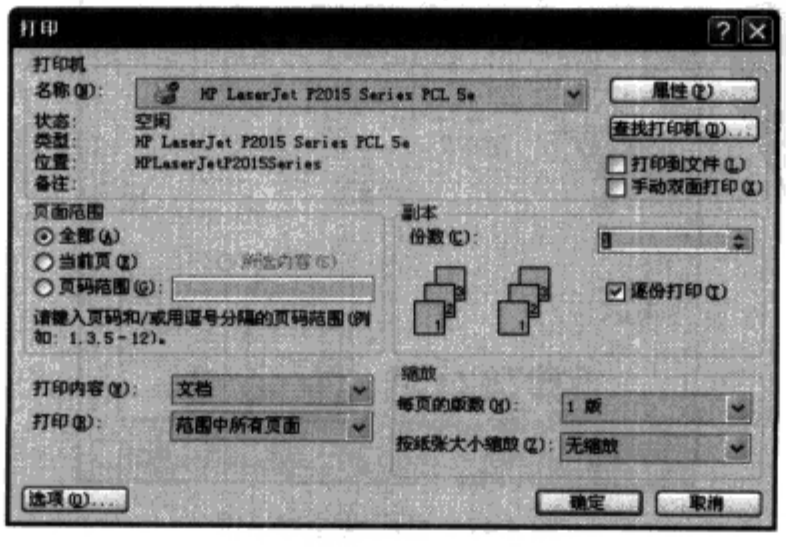


图 4.48 打印对话框

使用前一节创建的 Dialog7 工程，并在其中创建的对话框中再另外添加一个按钮控件，将其 Caption 属性设置为“打印”，将其 ID 设置为 IDC_BUTPRINT，然后在该按钮上右击，并选择“Add Event Handler”命令，在弹出的对话框中将消息名命名为 OnButprint，类名为 CDialog7Dlg，单击“Finish”按钮完成消息创建，页面会自动进入 CDialog7Dlg 中，鼠标指针定位到 CDialog7Dlg::OnButprint()函数处。

CDialog7Dlg::OnButprint()的程序代码如下：

```
void CDialog7Dlg::OnButprint()
{
    DWORD dwflags=PD_ALLPAGES | PD_NOPAGENUMS | PD_USEDEVMODECOPIES| PD_SELECTION | PD_
HIDEPRINTTOFILE;
    CPrintDialog dlg(FALSE,dwflags,NULL);
    if(dlg.DoModal() == IDOK)
    {
        CDC dc;
        dc.Attach(dlg.GetPrinterDC());
        DrawStr(&dc,TRUE);
    }
}
```

这段程序将创建一个“打印”对话框，并将获取的打印机通过消息对话框显示出来。

4.5　小结

本章主要介绍了对话框的控件、对话框的种类，以及如何设计对话框；然后在此基础上进一步讲解了对话框用户界面程序的编写，包括模态对话框与非模态对话框、属性页对话框及消息对话框的创建；还讲解了组合框（如何调整组合框的大小）、列表框、单选按钮、复选框等常用对话框控件的使用方法，以及获取并显示属性页上用户所选择的数据的方法；另外，还介绍了通用对话框的设计和实现方法，包括文件打开对话框、文件保存对话框、颜色对话框、字体对话框、查找对话框、页面设置对话框及打印对话框的创建方法。在完成本章的学习之后，读者应该：

（1）掌握模态对话框和非模态对话框的创建方法；

（2）掌握属性页对话框和消息对话框的创建方法；

（3）理解对话框控件的设计及使用方法；

（4）了解对话框的种类，以及设计对话框的方法；

（5）掌握各种通用对话框类，并学会设计并创建各种通用对话框。

第 5 章　Visual C++ 2010 MFC 对话框控件

由于所有的控件类都是由 CWnd 类派生来的，因此控件实际上也是窗口。控件通常是作为对话框的子窗口而创建的，也可以出现在视图窗口、工具条和状态条中。控件是独立的小部件，在对话框与用户的交互过程中，扮演着重要角色。控件的种类很多，图 5.1 显示了对话框中的一些基本的控件。

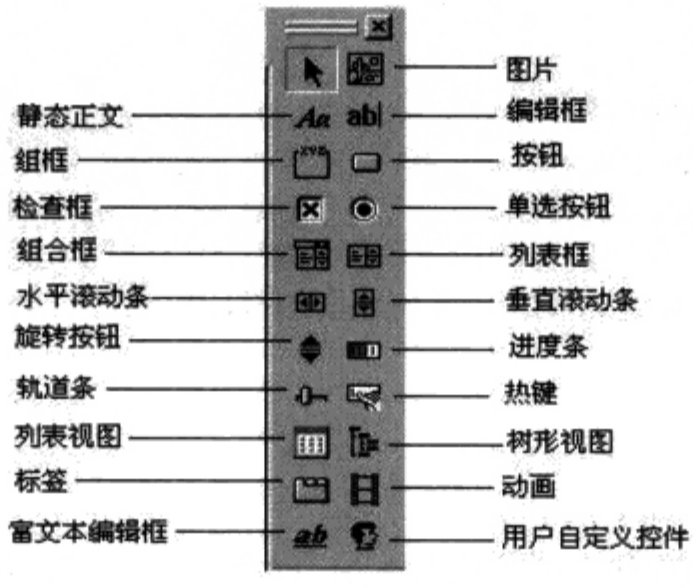


图 5.1　基本控件

MFC 的控件类封装了控件的功能，表 5.1 列出了一些常用的控件及其对应的控件类。

表 5.1　　**常用控件**

控　　件	功　　能	对应控件类
静态正文（Static Text）	显示正文，一般不能接受输入信息	CStatic
图片（Picture）	显式位图、图标、方框和图元文件，一般不能接受输入信息	CStatic
编辑框（Edit Box）	输入并编辑正文，支持单行和多行编辑	CEdit
命令按钮（PushButton）	响应用户的输入，触发相应的事件	CButton
复选框（Check Box）	用作选择标记，可以有选中、不选中和不确定 3 种状态	CButton
单选按钮（Radio Button）	用来从两个或多个选项中选中一项	CButton
组框（Group Box）	显示正文和方框，主要用来将相关的一些控件聚成一组	CButton
列表框（List Box）	显示一个列表，用户可以从该列表中选择一项或多项	CListBox
组合框（Combo Box）	是一个编辑框和一个列表框的组合，分为简易式、下拉式和下拉列表式 3 种。为用户显示一列选项，使用 Item 控件创建选项列表。每个 ComboBox 控件必须包括至少一个 Item 控件作为子项。当使用 ComboBox 控件时，用户也能够输入一个不在列表中显示的值（DropDown 控件需要用户选择列表中的某个选项）	CComboBox
滚动条（Scroll Bar）	主要用来从一个预定义范围值中迅速而有效地选取一个整数值	CScrollBar
按钮（Button）	提供基本的执行功能。单击按钮，会发生某项操作。上文已提及，Office 在多种上下文内容中使用按钮控件	CButton
按钮组（ButtonGroup）	将不同类型的按钮组合在一起。按钮显示在自然的框中，并且 Office 将它们放置在一起来表明其以某种方式联系。可以使用下列控件的组合：按钮、切换按钮、库、菜单、动态菜单和拆分按钮控件	CButton

续表

控　　件	功　　能	对应控件类
复选框（Check Box）	提供基本的选择功能。通过选中该控件，可以启用或禁用某选项。Office 为复选框控件提供了两种上下文环境，包括作为单独的控件或者是作为菜单的一部分	CCheckBox
下拉控件（DropDown）	为用户列出一列选项，使用 Item 或 Button 控件创建选项列表。列表至少应	CDropDown
拆分按钮（SplitButton）	创建具有默认操作和一列选择性选项的按钮。拆分按钮最好的示例之一是“开始”选项卡的“剪贴板”组中的“粘贴”按钮。必须包括 Button 或 ToggleButton 控件作为默认控件。在 Menu 控件中显示可选的操作，这里可以添加 Button 或 ToggleButton 控件	CButton
切换按钮（ToggleButton）	提供 CheckBox 和 Button 控件的组合。用户通过单击 ToggleButton 可以选择某状态和执行某操作	CButton

图 5.2 展示了各种不同类型的控件。

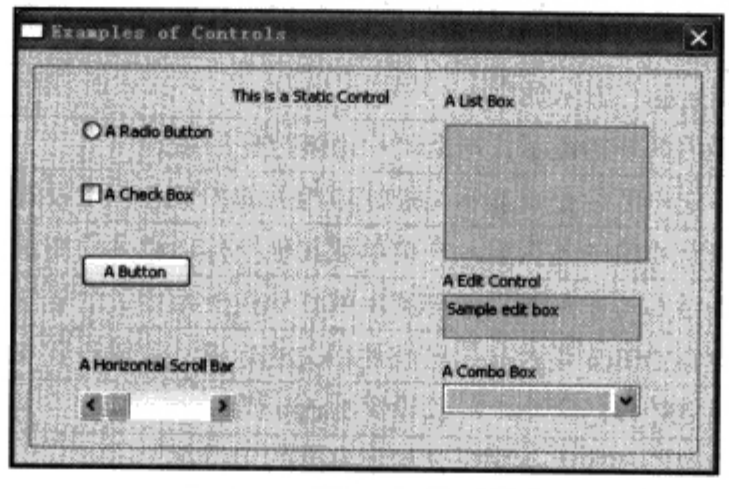


图 5.2　不同类型的控件

控件不仅有特定的作用，而且有大小和上下文内容。例如，在大多数编程环境中，按钮都是标准形式的按钮；然而在 RibbonX 中，可以创建大尺寸或者标准尺寸的按钮。此外，按钮的使用，以及在某些情况下的外观随上下文而不同：

- 功能区中的标准按钮；
- 对话框启动器的一部分；
- 下拉列表的一部分；
- 在快速访问工具栏中（QAT）；
- 拆分按钮的一部分；
- 带有标题的拆分按钮的一部分；
- 在控件组内；
- 菜单的一部分。

上下文使得控件在如何使用方面有非常大的不同。例如，在对话框启动器中必须包括一个按钮。Office 限制下拉列表中按钮的数量是 16，但不必包括任何内容；在拆分按钮中包括的按钮总是可见的，如果试图使用 Visible 属性，Office 就会提示错误。

表 5.2 提供了控件列表及其描述。

表 5.2　　控件属性

属　　性	描　　述
Description	当菜单的 Itemsize 属性设置为大时，指定 Office 显示的描述文本
Enabled	确定 Office 是否启用或禁用控件，可以选择 True 或 False 作为其值

续表

属　性	描　述
ID	为自定义控件指定标识符。不能与 IDMso 或 IDQ 属性一起使用
IDMso	为内置控件指定标识符。不能与 ID 或 IDQ 属性一起使用
IDQ	为限定的控件指定标识符（限定的控件取决于定义的自定义命名空间）。不能与 ID 或 IDMso 属性一起使用
Image	定义控件使用的自定义图像的名称
ImageMso	定义控件使用的内置图像的名称
InsertAfterMso	基于内置控件的位置，标识新控件的位置
InsertAfterQ	基于限定的控件的位置，标识新控件的位置
InsertBeforeMso	基于内置控件的位置，标识新控件的位置
InsertBeforeQ	基于限定的控件的位置，标识新控件的位置
ItemSize	确定菜单或其他控件项的大小。可以设置 Large 和 Normal
KeyTip	对控件添加指定的加速键组合。当用户按【Alt】键时显示键提示。可以指定任意键组合，使用 1～3 个字母
Label	指定控件显示的文本
ScreenTip	提供简要的提示，帮助用户理解控件的用途
ShowImage	确定 Office 是否显示与控件项相关的图像。可以设置 True 或 False
ShowItemImage	确定 Office 是否显示与菜单或其他控件项相关的图像。可以设置 True 或 False
ShowItemLabel	确定 Office 是否显示与菜单或其他控件项相关的标签。可以设置 True 或 False
ShowLabel	确定 Office 是否显示与控件相关的标签。可以设置 True 或 False
Size	确定控件的大小。可以设置 Large 或 Normal
SizeString	设置控件的大小来包含指定宽度的字符串
SuperTip	提供详细的提示，帮助用户理解控件的用途
Tag	包含用户定义的数据，可以在应用程序中用来与控件或其他元素交互
Title	为 MenuSeparator 控件指定显示的文本代替水平线
Visible	确定 Office 是否显示某控件或其他功能。可以设置为 True 或 False

表 5.3 介绍了创建应用程序时常用的回调。

表 5.3　　常用的控件回调

回　调	相关的控件	描　述
GetContent()	DynamicMenu	定义控件的内容
getDescription()	多个控件	获取控件的描述
getEnabled()	多个控件	让代码启用或禁用控件
getImage()	多个控件	获取为控件提供的自定义图像
getImageMso()	多个控件	获取为控件定义的标准图像
getItemCount()	ComboBox、DropDown、Gallery	获取控件的项目列表中的项目数
getItemHeight()	Gallery	以像素为单位确定项目显示在屏幕中的高度
getItemID()	ComboBox、DropDown、Gallery	确定当前项目的 ID
getItemImage()	ComboBox、DropDown、Gallery	获取与当前项目相关的图像
getItemLabel()	Combobox、DropDown、Gallery	获取与当前项目相关的标签
getItemScreenTip()	ComboBox、DropDown、Gallery	获取与当前项目相关的屏幕提示
getItemSuperTip()	ComboBox、DropDown、Gallery	获取与当前项目相关的超级提示
getItemWidth()	Gallery	以像素为单位确定项目显示在屏幕中的宽度
getKeyTip()	多个控件	获取控件的键提示（加速键）
getLabel()	多个控件	获取控件的标签

续表

回　调	相关的控件	描　述
getPressed()	CheckBox、ToggleButton	确定用户是否以手动的方式单击了控件以激活该控件
getScreenTip()	多个控件	获取控件的屏幕提示
getSelectedItemIndex()	DropDown、Gallery	确定用户从列表中选取的选项
getSelectItemID()	Gallery	获取用户已经在列表中选取的选项的 ID
getShowImage()	Button	确定控件是否显示图像（允许禁止图像，即使已经为控件定义了图像）
getShowLabel()	Button	确定控件是否显示标签（允许禁止标签，即使已经为控件定义了标签）
getSize()	多个控件	基于应用程序的输出，定义控件的大小
getSuperTip()	多个控件	获取控件的超级工具提示
getText()	ComboxBox、EditBox	获取与列表中当前所选择的项目相关的文本
getTitle()	MenuSeparator	为指定的控件提供标题。Office 显示文本来代替通常显示的水平线
getVisible()	Button	确定控件是否可见
loadImage()	CustomUI	作为整体装载与用户界面相关的图像
onAction()	多个控件	执行控件在应用程序代码中定义的操作
onChange()	ComboBox、EditBox	在用户选择或控件内容中发现变化
onLoad()	CustomUI	在装载过程中执行指定的操作

下面将对一些常用控件进行介绍，并讲解其开发过程。

5.1　Visual C++ 2010 Button 控件简介与开发

通过表 5.1 中的内容可以知道，在 Windows 系统中，命令按钮、复选框、单选按钮和组框都是由 MFC 类 CButton 来管理的，可统称为按钮控件。按钮是指可以响应鼠标单击的小矩形的子窗口，单击或者双击按钮可以完成某一任务，它可以向父窗口发出两种控件通知消息：单击时，将产生控件通知消息 BN_CLICKED；双击时，将产生控件通知消息 BN_DOUBLECLICKED。

按钮控件的基类——CButton 的成员函数及其作用如表 5.4 所示。

表 5.4　　CButton 类的常用成员函数

函　数	说　明
Create()	负责创建按钮控件
GetState()	获取单选按钮或复选框的状态
SetState()	设置按钮控件的高亮状态
SetCheck()	设置单选按钮或复选框的选中状态（参数为 0 时表示未选中；参数为 1 时表示选中；参数为 2 时表示不确定）
GetButtonStyle()	获取按钮对象的风格
SetButtonStyle()	更改按钮对象的风格
SetIcon()	指定按钮上显示的图标
GetIcon()	返回由 SetIcon()设置的图标句柄
SetBitmap()	指定按钮上显示的位图
SetCursor()	指定按钮上显示的光标
GetBitmap()	返回由 SetBitmap()设置的位图句柄
GetCursor()	返回由 SetCursor()设置的光标句柄

按钮控件包括命令按钮（PushButton）、复选框（Check Box）、单选按钮（Radio Button）、组框（Group Box）和自绘式按钮（Owner-draw Button）。命令按钮的作用是对用户的鼠标单击做出响应并触发相应的事件，在按钮中既可以显示正文，也可以显示位图。复选框控件作为一种选择标记，可以有选中、不选中和不确定 3 种状态。单选按钮控件一般都是成组出现的，具有互斥的性质，即同组单选按钮中只能有一个是被选中的。组框用来将相关的一些控件聚成一组。自绘式按钮是指由程序而不是由系统负责重绘的按钮。

如何在一个 CButton 类中区分这几种不同的按钮控件呢？这就要通过按钮风格（Button Style）。按钮风格的相关知识会在讲述该类的 Create()函数中有所介绍。

MFC 的 CButton 类封装了按钮控件。CButton 类的成员函数 Create()负责创建按钮控件，函数原型如下：

```
BOOL Create( LPCTSTR lpszCaption, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID );
```

该函数如果创建成功，返回 True，否则返回 False。

第一个参数 lpszCaption 指定了按钮显示的正文。

第二个参数 dwStyle 指定了按钮的风格。dwStyle 可以是下列值的组合：

WS_CHILD	指定该控件是一个子窗口
WS_VISIBLE	指定该控件可显示
WS_DISABLED	指定该控件不可显示
WS_GROUP	指定该类控件为一组
WS_TABSTOP	使控件具有 Tabstop 属性
BS_CHECKBOX	指定该按钮控件是一个复选框控件
BS_PUSHBUTTON	指定该按钮控件是一个普通命令按钮控件
BS_RADIOBUTTON	指定该控件是一个单选按钮控件，在按钮的右侧显示正文
BS_GROUPBOX	指定该按钮控件是一个组框控件
BS_OWNERDRAW	指定该按钮控件是一个自绘式按钮控件
BS_AUTOCHECKBOX	同 BS_CHECKBOX，但是单击时按钮会自动反转
BS_3STATE	同 BS_CHECKBOX，但是控件有 3 种状态：选择、未选和变灰
BS_AUTO3STATE	同 BS_3STATE，但是单击按钮时会改变状态
BS_AUTORADIOBUTTON	同 BS_RADIOBUTTON，但是单击时按钮会自动反转
BS_DEFPUSHBUTTON	指定默认的命令按钮，这种按钮的周围有一个黑框，用户可以按【Enter】键来快速选择该按钮
BS_LEFTTEXT	使控件的标题显示在按钮的左侧

下面给出常用的几种方案：如果想要创建一个普通的命令按钮，应指定的风格为 WS_CHILD|WS_VISIBLE|WS_TABSTOP；若要创建一个普通的复选框控件，应指定风格为 WS_CHILD|WS_VISIBLE|WS_TABSTOP| BS_AUTOCHECKBOX；若要创建组中第一个单选按钮，应指定风格 WS_CHILD | WS_VISIBLE | WS_TABSTOP | WS_GROUP| BS_AUTORADIOBUTTON，组中其他单选按钮指定风格则不应该包括 WS_TABSTOP 和 WS_GROUP。

第三个参数 rect 是一个对 RECT 或 CRect 结构的引用，rect 说明了按钮的位置和尺寸。

第四个参数 pParentWnd 指向父窗口，该参数不能为 NULL。

第五个参数 nID 是按钮的 ID。

例如：

```
CButton myButton1, myButton2, myButton3;
// 创建一个普通命令按钮
myButton1.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
 CRect(10,10,100,30), this, 1);
```

```
// 创建一个单选按钮
myButton2.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_RADIOBUTTON,
CRect(10,40,100,70), this, 2);
// 创建一个复选框
myButton3.Create(_T("My button"), WS_CHILD|WS_VISIBLE|BS_AUTOCHECKBOX,
CRect(10,100,100,130), this, 4);
```

MFC 还提供了 CButton 的派生类 CBitmapButton。利用该类可以创建一个拥有四幅位图的命令按钮，按钮在不同状态时会显示不同的位图，这样可以使界面显得生动活泼。

5.2　Visual C++ 2010 List Box 控件简介与开发

列表框控件是一个选项列表，用于显示项目列表，它允许用户从所列出的选项中进行单项或多项选择，被选择的项呈高亮度显示。列表框具有边框，并且一般带有一个垂直滚动条。列表框分单选列表框和多重选择列表框两种。单选列表框一次只能选择一个列表项，而多重选择列表框可以进行多重选择，即可以选择某一范围内的选项。

若要处理列表框控件窗口传送给父窗口的控件通知消息，则必须为每条消息添加消息映射选项和消息处理函数。它可以向父窗口发送如表 5.5 所示的几种消息。

表 5.5　　与列表框相关的常见消息

消　息	产生的原因
LBN_DBLCLK	用户双击了一个列表项。只有具有 LBS_NOTIFY 的列表框才能发送该消息
LBN_ERRSPACE	列表框不能分配足够的内存
LBN_SELCHANGE	单击选择一个列表项。只有具有 LBS_NOTIFY 的列表框才能发送该消息
LBN_SELCANCEL	当前的选择被取消。只有具有 LBS_NOTIFY 的列表框才能发送该消息
LBN_KILLFOCUS	列表框失去输入焦点
LBN_SETFOCUS	列表框获得输入焦点
WM_CHARTOITEM	当列表框收到 WM_CHAR 消息后，向父窗口发送该消息。只有具有 LBS_WANTKEYBOARDINPUT 风格的列表框才会发送该消息
WM_VKEYTOITEM	当列表框收到 WM_KEYDOWN 消息后，向父窗口发送该消息。只有具有 LBS_WANTKEYBOARDINPUT 风格的列表框才会发送该消息

用于操作列表框控件的 CListBox 类的成员函数及其作用如表 5.6 所示。

表 5.6　　CListBox 类的成员函数

函　数	说　明
GetCount()	返回列表框中选项的数目
GetHorizontalExtent()	返回列表框中可滚动范围的宽度
SetHorizontalExtent()	设置列表框的水平滚动宽度
GetTopIndex()	返回列表框中的第一个可见项的索引
SetTopIndex()	指定待定的列表项为可见的
GetItemRect()	返回列表框中当前显示列表项的矩形区域边界
SelectString()	搜索并选择列表框中的指定字符串
GetItemData()	返回并指定与列表项有关的 32 位值
SetItemDataPtr()	设置指向列表项的指针
ItemFromPoint()	获取与指定点最近的列表项
SetItemHeight()	设置列表项的高度

续表

函　数	说　明
GetItemHeight()	返回列表项的高度
GetSel()	返回指定列表项的选择状态
GetText()	复制列表项到缓冲区
GetTextLen()	返回列表项的字节数
SetColumnWidth()	设置多栏列表框的栏宽
SetTabStops()	设置列表框中制表符的位置
AddString()	添加字符串到列表框中
DeleteString()	从列表框中删除字符串
InsertString()	在列表框中的指定位置插入字符串
ResetContent()	从当前目录添加文件名到列表框
FindString()	搜索列表框中指定的字符串
FindStringExact()	搜索列表框中第一个与指定字符串匹配的字符

MFC 的 CListBox 类封装了列表框，CListBox 类的 Create()成员函数负责列表框的创建，该函数原型如下：

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

如果创建成功，返回 True，否则返回 False。

第一个参数 dwStyle 指定了列表框控件的风格，dwStyle 可以是下列值的组合：

```
WS_CHILD            指定该控件是一个子窗口
WS_VISIBLE          指定该控件可显示
WS_DISABLED         指定该控件不可显示
WS_GROUP            指定该类控件作为一组
WS_TABSTOP          使控件具有 Tabstop 属性
WS_VSCROLL          使该控件具有水平滚动条
WS_HSCROLL          使该控件具有垂直滚动条
WS_BORDER           使控件带边框
LBS_EXTENDEDSEL     支持多重选择。在单击列表项时按住【Shift】键或【Ctrl】键即可选择多项
LBS_MULTICOLUMN     指定一个水平滚动的多列列表框，通过调用 CListBox::SetColumnWidth 函数来设置每列的宽度
LBS_MULTIPLESEL     支持多重选择。列表项的选择状态随着用户对该项单击或双击而翻转
LBS_NOREDRAW        当选择发生变化时防止列表框被更新，可发送 WM_SETREDRAW 来改变该风格
LBS_NOTIFY          当用户单击或双击鼠标键时通知父窗口
LBS_SORT            使插入列表框中的项按升序排列
LBS_STANDARD        相当于多重指定了 WS_BORDER|WS_VSCROLL|LBS_SORT |LBS_NOTIFY
LBS_WANTKEYBOARDINPUT    允许列表框的父窗口接收 WM_VKEYTOITEM 和 WM_CHARTOITEM 消息，
                         以响应键盘输入
```

下面给出几种常用的方案：如果想要创建一个普通的单选择列表框，应指定的风格为 WS_CHILD|WS_VISIBLE|WS_TABSTOP|LBS_STANDARD；如果要创建一个多重选择列表框，应该在单选择列表框风格的基础上再加上 LBS_MULTIPLESEL 或 LBS_ EXTENDEDSEL；如果不希望列表框排序，就不能使用 LBS_STANDARD 风格。

第二个参数 rect 是一个对 RECT 或 CRect 结构的引用，说明了控件的位置和尺寸。

第三个参数 pParentWnd 指向父窗口，该参数不能为 NULL。

第四个参数 nID 则说明了控件的 ID。

例如：

```
CListBox* pmyListBox;
```

```
pmyListBox->Create(WS_CHILD|WS_VISIBLE|LBS_STANDARD|WS_HSCROLL,
CRect(10,10,200,200), this, 1);
```

5.3　Visual C++ 2010 Com boBox 控件简介与开发

组合框把一个编辑框和一个单选择列表框结合在了一起，用户既可以在编辑框中输入，也可以从列表框中选择一个列表项来完成输入。组合框有简易式（Simple）、下拉式（Dropdown）和下拉列表式（Drop List）3 种风格。简易式组合框包含一个编辑框和一个总是显示的列表框；下拉式组合框与简易式组合框类似，二者的区别在于下拉式组合框仅当用户单击下拉按钮后列表框才会弹出。下拉列表式组合框也有一个下拉的列表框，但它的编辑框是只读的，不能输入字符。

组合框可以向父窗口发送如表 5.7 所示的几种消息。

表 5.7　　与组合框有关的消息

消　　息	产生的原因
CBN_CLOSEUP	组合框的列表框组件被关闭。简易式组合框不会发出该消息
CBN_DBLCLK	用户在某列表项上双击。只有简易式组合框才会发出该消息
CBN_DROPDOWN	组合框的列表框组件下拉。简易式组合框不会发出该消息
CBN_EDITCHANGE	编辑框的内容被用户改变了。与 CBN_EDITUPDATE 不同，该消息是在编辑框显示的正文被刷新后才发出的。下拉列表式组合框不会发出该消息
CBN_EDITUPDATE	在编辑框准备显示改变了的正文时发送该消息。下拉列表式组合框不会发出该消息
CBN_SELENDCANCEL	表明用户的选择应该取消。当用户在列表框中选择了一项，然后又在组合框控件外单击时，就会导致该消息的发送
CBN_SELENDOK	用户选择了一项，然后按【Enter】键或单击了下拉按钮。该消息表明用户确认了自己所做的选择
CBN_KILLFOCUS	组合框失去了输入焦点
CBN_SETFOCUS	组合框获得了输入焦点
CBN_SELCHANGE	用户通过按箭头键改变了列表项的选择
CBN_ERRSPACE	组合框未分配到足够的空间

MFC 的 CComboBox 类封装了组合框。需要指出的是，虽然组合框是编辑框和列表框的组合，但是 CComboBox 类并不是 CEdit 类和 CListBox 类的派生类，而是 CWnd 类的派生类。

CComboBox 的成员函数 Create()负责创建组合框，该函数原型如下：

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

该函数如果创建成功，返回 True，否则返回 False。

第一个参数 dwStyle 指定了组合框控件的风格，dwStyle 可以是下列值的组合：

```
WS_CHILD          指定该控件是一个子窗口
WS_VISIBLE        指定该控件可显示
WS_DISABLED       指定该控件不可显示
WS_GROUP          指定该类控件作为一组
WS_TABSTOP        使该控件具有 Tabstop 属性
WS_VSCROLL        使该控件具有水平滚动条
WS_HSCROLL        使该控件具有垂直滚动条
CBS_AUTOHSCROLL   使编辑框组件具有水平滚动的风格
CBS_DROPDOWN      指定一个下拉式组合框
CBS_DROPDOWNLIST  指定一个下拉列表式组合框
CBS_SIIMPLE       指定一个简易式组合框
```

CBS_SORT	自动对列表框组件中的选项进行排序
CBS_NOINTEGRALHEIGHT	组合框的尺寸由应用程序而不是由 Windows 指定。通常，由 Windows 指定尺寸会使列表项的某些部分隐藏起来

第二个参数 rect 指定了列表框组件下拉后组合框的位置和尺寸。

第三个参数 pParentWnd 指向父窗口，该参数不能为 NULL。

第四个参数 nID 则说明了控件的 ID。

例如：

```
CComboBox* pmyComboBox;
pmyComboBox->Create(WS_CHILD|WS_VISIBLE|WS_VSCROLL|CBS_DROPDOWNLIST,CRect(10,10,200,100), this, 1);
```

CComboBox 类常用的成员函数介绍如下。

- AddString(): 在列表框的尾部添加一个字符串选项。
- InsertString(): 根据参数提供的位置，从列表框中插入一个字符串选项。
- ResetContent(): 删除列表框中所有的选项。
- DeleteString(): 从列表框中删除一个字符串选项。
- GetCount(): 获得组合框的列表框中选项的数目。
- SetCurSel(): 在列表框中选中一个选项。
- GetCurSel(): 获得被选中的选项的索引。
- SetEditSel(): 设置组合框编辑控件中当前选择的首尾字符的位置。
- GetEditSel(): 获得组合框中编辑框中当前选择的首尾字符的位置。
- Undo()/Clear()/Copy()/Cut()/Paste(): 组合框编辑框部分的剪贴板操作，分别用于撤销上一次输入，清除编辑框中被选择的正文，把在编辑框中选择的正文复制到剪贴板中，清除编辑框中被选择的正文并把这些正文复制到剪贴板中，将剪贴板中的正文插入编辑框的当前插入符处。
- SetItemData()/GetItemData(): 设置/获得与组合框的一个列表框选项相关联的 32 位整数值。

5.3.1 创建扩展组合框控件

扩展组合框（Extended Combo Box）控件是为项图像提供本机支持的标准组合框（Combo Box）控件的扩展。可以使用这些图像指示组合框中各项的状态，如当前选定项和取消选定项的状态。为了使项图像更易于访问，此控件提供了图像列表支持。

使用此控件提供组合框功能，开发人员而不必手动绘制项图形。

创建扩展组合框控件的方式取决于是在对话框窗口中使用该控件还是在非对话框窗口中使用此控件。

1. 直接在对话框窗口中使用 CComboBoxEx

（1）在对话框编辑器中，将 Extended Combo Box 控件添加到对话框模板资源，指定其控件 ID。

（2）使用扩展组合框控件的“属性”面板指定任何所需样式。

（3）使用添加成员变量向导添加带 Control 属性的 CComboBoxEx 类的成员变量，可以使用此成员调用 CcomboBoxEx 类的成员函数。

（4）对于任何需要处理的扩展组合框控件通知消息，使用“属性”面板在对话框类中映射处理函数。

（5）在 OnInitDialog()中为 CComboBoxEx 对象设置任何附加样式。

2. 在非对话框窗口中使用 CComboBoxEx

（1）在视图或窗口类中定义此控件。

（2）调用控件的 Create()成员函数，可能在 OnInitialUpdate()中，也可能与父窗口的 OnCreate()处理函数一样早设置此控件的样式。

5.3.2　在扩展组合框控件中使用图像列表

扩展组合框控件的主要功能是可以将图像列表中的图像与组合框（Combo Box）控件中的各项相关联。每项都可以显示 3 个不同的图像：一个是选定状态的图像，另一个是未选定状态的图像，第三个是覆盖图像的图像。

下面的过程将图像列表与扩展组合框控件相关联。

（1）通过使用 CImageList 构造函数并存储所得到的指针来构造新的图像列表（或者使用现有图像列表对象）。

（2）通过调用 CImageList::Create()函数初始化新的图像列表对象。以下代码是此调用的一个示例：

```
m_pImageList->Create(16, 16, ILC_COLOR, 2, 2);
```

为每个可能的状态添加可选图像：选定、未选定或覆盖。下面的代码可以添加 3 个预定义图像。

```
m_pImageList->Add(pApp->m_hIconSelected);
m_pImageList->Add(pApp->m_hIconNotSelected);
m_pImageList->Add(pApp->m_hIconOverlay);
```

（3）通过调用 CComboBoxEx::SetImageList()将图像列表与控件相关联。

将图像列表与控件相关联后，可以分别为每一项指定将用于 3 种可能状态的图像。

5.3.3　设置各项的图像

扩展组合框项所使用的不同类型的图像由 COMBOBOXEXITEM 结构的 iImage、iSelectedImage 和 iOverlay 成员中的值确定。每个值是关联控件图像列表中的图像索引。默认情况下，这些成员设置为 0，使控件不显示项的图像。如果要将图像用于特定项，可以在插入组合框项时相应地修改结构，或通过修改现有组合框项相应地修改结构。

1．设置新项的图像

如果正在插入新项，请使用正确的值初始化 iImage、iSelectedImage 和 iOverlay 结构成员，然后通过调用 CComboBoxEx::InsertItem()插入项。

下面的示例将新扩展组合框项（cbi）插入扩展组合框控件（m_comboEx），并为所有 3 个图像状态提供索引：

```
COMBOBOXEXITEM cbi;
CString str;
int nItem;
cbi.mask = CBEIF_IMAGE | CBEIF_INDENT | CBEIF_OVERLAY | CBEIF_SELECTEDIMAGE | CBEIF_TEXT;
cbi.iItem = i;
str.Format(_T("Item %02d"), i);
cbi.pszText = (LPTSTR)(LPCTSTR)str;
cbi.cchTextMax = str.GetLength();
cbi.iImage = 0;
cbi.iSelectedImage = 1;
cbi.iOverlay = 2;
cbi.iIndent = (i & 0x03);     //Set indentation according to item position
nItem = m_comboEx.InsertItem(&cbi);
ASSERT(nItem == i);
```

2. 设置现有项的图像

如果正在修改现有项，需要使用 COMBOBOXEXITEM 结构的 mask 成员。

修改现有项以使用图像：

- 声明 COMBOBOXEXITEM 结构并将 mask 数据成员设置为要修改的值；
- 使用此结构调用 CComboBoxEx::GetItem()；
- 使用适当的值修改刚返回的结构的 mask、iImage 和 iSelectedImage 成员；
- 传入已修改的结构来调用 CComboBoxEx::SetItem()。

下面的示例通过交换第三个扩展组合框项的选定和未选定图像来说明此过程：

```
COMBOBOXEXITEM cbi;
CString str;
int nItem;
cbi.mask = CBEIF_IMAGE | CBEIF_SELECTEDIMAGE;
cbi.iItem = 2;
m_comboEx.GetItem(&cbi);
cbi.iImage = 1;
cbi.iSelectedImage = 0;
nItem = m_comboEx.SetItem(&cbi);
ASSERT(nItem != 0);
```

5.3.4 处理扩展组合框控件中的通知消息

用户与扩展组合框交互时，控件（CComboBoxEx）向其父窗口发送通知消息，该窗口通常是视图或对话框对象。如果要进行某种响应，请处理这些消息，例如，用户激活下拉列表或单击控件的编辑框时，发送 CBEN_BEGINEDIT 通知消息。

使用“属性”面板为要实现的那些消息的父类添加通知处理程序。

下面介绍扩展组合框控件发送的各种通知。

- CBEN_BEGINEDIT：用户激活下拉列表或单击控件的编辑框时发送此通知。
- CBEN_DELETEITEM：删除项后发送此通知。
- CBEN_DRAGBEGIN：用户开始拖动控件的编辑框中显示的项的图像时发送此通知。
- CBEN_ENDEDIT：用户结束编辑框中的操作后或从控件的下拉列表中选定项后发送此通知。
- CBEN_GETDISPINFO：发送此通知以检索与回调项有关的显示信息。
- CBEN_INSERTITEM：将新项插入控件后发送此通知。

5.4 Visual C++ 2010 List 控件简介与开发

列表 List 控件可以看作功能增强的 ListBox，它提供了 4 种风格，而且可以同时显示一列的多种属性值。MFC 中使用 CListCtrl 类来封装列表控件的各种操作。通过调用 BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);创建一个窗口，dwStyle 中可以使用表 5.8 所示的列表控件的专用风格。

表 5.8　　列表控件风格

风　格	意　义
LVS_ICON ,LVS_SMALLICON ,LVS_LIST,LVS_REPORT	这 4 种风格决定控件的 4 种外观，同时只可以选择其中一种，分别对应：大图标显示、小图标显示、列表显示、报表显示

续表

风　格	意　义
LVS_EDITLABELS	结点的显示字符可以被编辑，对于报表风格来讲，可编辑的只有第一列
LVS_SHOWSELALWAYS	在失去焦点时也显示当前选中的结点
LVS_SINGLESEL	同时只能选中列表中的一项

使用列表控件显示带标签的图标的任何排列，像在 Windows 资源管理器或文本的纵栏式列表（带图标或不带图标）中那样。在列表控件中包括 4 种可能的视图（不要与 MFC 视图混淆），即大图标视图、小图标视图、列表视图和报表视图。

在一些视图中，用户可以将图标拖动到不同的位置或编辑图标标签。例如，Windows 资源管理器中的右窗格，该窗格在非对话框窗口中使用列表控件，可以实践一下资源管理器的“视图”菜单中的可用视图。

5.4.1　列表控件和列表视图

为方便起见，MFC 以两种方法封装列表控件，可以按下面的两种方式使用列表控件：

- 通过在对话框类中嵌入 CListCtrl 对象来直接使用。
- 通过使用类 CListView 来间接使用。

CListView 使得在 MFC 文档/视图结构中集成列表控件变得容易，封装此控件与 CEditView 封装编辑控件基本相同：控件填充 MFC 视图的整个表面区域（视图“是”转换为 CListView 的控件）。

CListView 对象从 CCtrlView 及其基类继承并添加一个检索基础列表控件的成员函数，像视图一样使用视图成员函数来处理视图。使用 GetListCtrl()成员函数获得对列表控件的成员函数的访问权限。使用这些成员可以：

- 添加、删除或操作列表中的“项”；
- 设置或获取列表控件的属性。

若要获得对作为 CListView 基础的 CListCtrl 的引用，请从列表视图类调用 GetListCtrl()函数：

```
CListCtrl& ctlList = GetListCtrl();
```

5.4.2　列表项和图像列表

列表控件（CListCtrl）中的“项”由图标、标签和可能的其他信息（在“子项”中）组成。

列表控件项的图标包含在图像列表中，第一个图像列表包含图标视图中使用的标准尺寸的图标；第二个可选图像列表包含控件的其他视图中所使用的相同图标的较小版本；第三个可选列表包含“状态”图像（如复选框），用于在某些视图的小图标前面显示；第四个可选列表包含列表控件的单个标头项中显示的图像。

如果列表视图控件是用 LVS_SHAREIMAGELISTS 样式创建的，则不再使用图像列表时，要负责销毁这些图像列表。如果将相同的图像列表分配给多个列表视图控件，请指定其样式；否则一个以上的控件可能会尝试销毁同一个图像列表。

若要创建列表控件，需要在将新项插入列表中时提供将要使用的图像列表。下面的代码将说明此过程：

```
// create, initialize, and hook up image list
m_pImageList = new CImageList();
ASSERT(m_pImageList != NULL);      // serious allocation failure checking
```

```
m_pImageList->Create(32, 32, TRUE,    4, 4);
m_pImageList->Add(pApp->LoadIcon(IDI_ICONLIST1));
m_pImageList->Add(pApp->LoadIcon(IDI_ICONLIST2));
m_listctrl.SetImageList(m_pImageList, LVSIL_NORMAL);
```

其中 m_pImagelist 是 CImageList 类型的指针，m_listctrl 是一个 CListCtrl 数据成员，关于 CImageList，将在后面介绍。

但是，如果不打算在列表视图或列表控件中显示图标，则不需要图像列表。

5.4.3 回调项和回调屏蔽

对于列表视图控件的每一项，列表视图控件通常存储标签文本、项的图标的图像列表索引和用于表示项状态的一组位标志。可以将单个项定义为回调项，这在应用程序已经存储某个项的一些信息时很有用。

可以通过为 LV_ITEM 结构（请参见 CListCtrl::GetItem()函数）的 pszText 和 iImage 成员指定适当的值来将一个项定义为回调项。如果应用程序维护项或子项的文本，则为 pszText 成员指定 LPSTR_TEXTCALLBACK 值。如果应用程序跟踪项的图标，则为 iImage 成员指定 I_IMAGECALLBACK 值。

除了定义回调项之外，还可以修改控件的回调屏蔽，该屏蔽是一组指定项状态的位标志，为这些项状态存储当前数据的是应用程序而不是控件。回调屏蔽适用于控件的所有项，这和回调项指定不同，后者适用于特定的项。默认情况下，回调屏蔽为零，意思是控件默认跟踪所有项状态，若要更改此默认行为，请将屏蔽初始化为下列值的任意组合。

- LVIS_CUT: 项被标记为进行剪贴操作。
- LVIS_DROPHILITED: 项突出显示为拖放目标。
- LVIS_FOCUSED: 项有焦点。
- LVIS_SELECTED: 项被选定。
- LVIS_OVERLAYMASK: 应用程序存储每一个项的当前覆盖图像的图像列表索引。
- LVIS_STATEIMAGEMASK: 应用程序存储每一个项的当前状态图像的图像列表索引。

5.4.4 创建列表控件

创建列表控件（CListCtrl）的方式取决于是直接使用此控件还是使用类 CListView。如果使用 CListView，框架将视图构造成其文档/视图创建序列的一部分。创建列表视图也就创建了列表控件（List Control）（二者相同）。控件在视图的 OnCreate()处理函数中创建，在这种情况下，通过调用 GetListCtr() 准备在控件中添加项。

1. 在对话框中直接使用 CListCtrl

（1）在对话框编辑器中，向对话框模板资源添加 List Control 控件，指定其控件 ID。

（2）使用添加成员变量向导添加带 Control 属性的 CListCtrl 类型的成员变量，可以使用该成员调用 CListCtrl 成员函数。

（3）对于需要处理的任何列表控件通知消息，使用“属性”面板来映射对话框类中的处理函数。

（4）在 OnInitDialog()中，设置 CListCtrl 的样式。虽然以后可以更改视图，但该操作可确定在控件中获得的“视图”类型。

2. 在非对话框窗口中使用 CListCtrl

（1）在视图或窗口类中定义此控件。

（2）调用控件的 Create()成员函数，可能在 OnInitialUpdate()中，也可能与父窗口的 OnCreate()处理函数一样早（如果创建此控件的子类）设置此控件的样式。

5.4.5　创建图像列表

不论是使用 CListView 还是使用 CListCtrl，创建图像列表是一样的。

只有在列表控件包括 LVS_ICON 样式时才需要图像列表。

使用 CImageList 类创建一个或多个图像列表（对于标准尺寸的图标、小图标和状态）。

为每一个图像列表调用 CListCtrl::SetImageList()函数，将指针传递给合适的 CImageList 对象。

下面介绍一下 CImageList 类及图像列表。

由 CImageList 类表示的图像列表是相同大小的图像的集合，每个图像可以通过其索引来引用。图像列表用于有效管理大的图标或位图集。图像列表不是窗口，本身并不是控件；但和几种不同类型的控件一起使用，包括列表控件（CListCtrl）、树控件（CTreeCtrl）和选项卡控件（CTabCtrl）。

图像列表中的所有图像都包含在屏幕设备格式的单个宽位图中。图像列表也可能包括单色位图，该位图包含用于透明地绘制图像（图标样式）的屏蔽。CImageList 提供的成员函数可以使开发人员绘制图像，创建和销毁图像列表，添加和移除图像，替换图像，合并图像，以及拖动图像。

1．图像列表的类型

有两种类型的图像列表（CImageList）：非屏蔽的和屏蔽的。非屏蔽的图像列表由包含一个或多个图像的色位图构成。屏蔽的图像列表由相同大小的两个位图构成，第一个是包含图像的色位图，第二个是包含一系列屏蔽的单色位图（一个屏蔽对应于第一个位图中的一个图像）。

Create()成员函数的一个重载采用一个标志来指示图像列表是否是屏蔽的（其他重载创建屏蔽的图像列表）。

绘制非屏蔽图像时，只是简单地将其复制到目标设备上下文，即在设备上下文的现有背景色上绘制图像；绘制屏蔽图像时，图像位和屏蔽位组合，一般在位图中产生显露目标设备上下文背景色的透明区域。绘制屏蔽图像时可以指定几种绘图样式，例如，可以指定图像进行抖色处理以指示选定的对象。

2．使用图像列表

图像列表的典型用法遵循下面的模式。

- 构造 CImageList 对象，调用其 Create()函数的一个重载来创建图像列表并将图像列表附加到 CImageList 对象上。
- 如果在创建图像列表时没有添加图像，则调用 Add()或 Read()成员函数向图像列表添加图像。
- 通过调用控件的适当的成员函数，将图像列表与此控件相关联，或者自己使用图像列表的 Draw()成员函数绘制图像列表中的图像。
- 允许用户使用图像列表的内置拖动支持图像拖动。

如果图像列表是用 new 操作符创建的，则在操作完 CImageList 对象后必须销毁它。

3．操作图像列表

Replace()成员函数用新的图像替换图像列表（CImageList）中的图像。如果需要动态增加图像列表

对象中的图像数，该函数也有效。SetImageCount()函数动态更改存储在图像列表中的图像数。如果增加图像列表的大小，请调用 Replace ()函数添加图像；如果减小图像列表的大小，则释放超出大小的图像。

Remove()成员函数用于移除图像列表中的图像。Copy()成员函数可以在图像列表内复制或交换图像，该函数使开发人员确定应将源图像复制到目标索引，还是应交换源图像和目标图像。

若要通过合并两个图像列表来创建新的图像列表，请使用 Create()成员函数的适当重载。该 Create()重载合并现有图像列表的第一个图像，并将所得到的图像存储在新的图像列表对象中。通过在第一个图像上透明地绘制第二个图像来创建新图像。新图像的屏蔽是对两个现有图像的屏蔽位执行逻辑 OR 运算的结果。

重复该操作一直到所有图像都合并并添加到新的图像列表中。

可以调用 Write()成员函数将图像信息写入并存档，并通过调用 Read()成员函数读回。

GetSafeHandle()、Attach()和 Detach()成员函数使开发人员得以操作附加到 CImageList 对象的图像列表句柄，而 DeleteImageList()成员函数删除图像列表但不销毁 CImageList 对象。

4．绘制图像列表中的图像

若要绘制图像，应该使用 CImageList::Draw()成员函数，需要指定指向设备上下文对象的指针、要绘制的图像的索引、设备上下文中绘制图像的位置，以及一组指示绘图样式的标志。

指定 ILD_TRANSPARENT 样式后，Draw()使用一个两步进程来绘制屏蔽的图像，首先，对图像位和屏蔽位执行逻辑 AND 运算；然后，对第一个运算的结果和目标设备上下文的背景位执行逻辑 XOR 运算。该进程在结果图像中创建透明区域；也就是说，屏蔽中的每一个空白位使结果图像中相应的位成为透明。

在纯色背景上绘制屏蔽图像之前，应使用 SetBkColor()成员函数将图像列表的背景色设置为和目标一样的颜色。设置此颜色可消除在图像中创建透明区域的必要，使得 Draw()可以只将图像复制到目标设备上下文中，从而极大地提高性能。若要绘制此图像，在调用 Draw()时指定 ILD_NORMAL 样式。

可以随时为屏蔽图像列表（CImageList）设置背景色，使其能在任何纯背景上进行正确绘制。默认情况下，背景色设置为 CLR_NONE，以便能透明地绘制图像。若要检索图像列表的背景色，应该使用 GetBkColor()成员函数。

ILD_BLEND25 和 ILD_BLEND50 样式用系统突出显示颜色对图像进行抖色处理。如果使用屏蔽图像表示用户可以选择的对象，则这些样式很有帮助。例如，可以使用 ILD_BLEND50 样式在用户选择它时绘制图像。

使用 SRCCOPY 光栅操作将非屏蔽图像复制到目标设备上下文。无论设备上下文的背景色是什么，该图像中的颜色都显示为原来的颜色。Draw()中指定的绘图样式对非屏蔽图像的外观也不起作用。

除了 Draw()成员函数，另一个函数 DrawIndirect()可以扩展呈现图像的能力。DrawIndirect()采用 IMAGELISTDRAWPARAMS 结构作为参数，该结构可用来自定义当前图像的呈现，包括光栅操作（ROP）代码的使用。

5．图像列表中的图像覆盖

每一个图像列表（CImageList）都包括一个用作覆盖屏蔽的图像列表。“覆盖屏蔽”指透明地在另一个图像上绘制出的图像。任何图像都可以用作覆盖屏蔽，最多可以为每个图像列表指定 4 个覆盖屏蔽。

可以使用 SetOverlayImage()成员函数、图像索引和覆盖屏蔽索引将图像索引添加到覆盖屏蔽列表。请注意，覆盖屏蔽索引从 1 开始，而不是从 0 开始。

使用单个 Draw()调用在图像上绘制覆盖屏蔽，参数包括要绘制的图像的索引和覆盖屏蔽的索引。

必须使用 INDEXTOOVERLAYMASK 宏来指定覆盖屏蔽的索引，也可以在调用 DrawIndirect()成员函数时指定覆盖图像。

6．拖动图像列表中的图像

CImageList 包括用于在屏幕上拖动图像的函数。拖动函数按颜色平滑地移动图像，并且没有任何光标闪烁。屏蔽图像和非屏蔽图像都可以被拖动。

BeginDrag()成员函数开始拖动操作。参数包括将要拖动的图像的索引和图像内的作用点位置，作用点是单个像素，拖动函数将其识别为图像的精确屏幕位置。应用程序一般将作用点设置为与鼠标光标的作用点重合。DragMove()成员函数将图像移动到新的位置。

DragEnter()成员函数在窗口内设置拖动图像的初始位置并在该位置绘制图像。参数包括指向绘制图像所在窗口的指针和指定窗口内初始位置坐标的点。坐标相对于窗口的左上角，而非工作区，这对所有采用坐标作为参数的图像拖动函数都一样，这意味着指定坐标时必须补充窗口元素（如边框、标题栏和菜单栏）的宽度。如果调用 DragEnter()时指定 NULL 窗口句柄，则拖动函数在与桌面窗口关联的设备上下文中绘制图像，并且坐标相对于屏幕的左上角。

在拖动操作过程中，DragEnter()锁定对给定窗口的所有其他更新。如果在拖动操作过程中需要进行任何绘制，如突出显示拖放操作的目标，则可以使用 DragLeave()成员函数临时隐藏拖动的图像，还可以使用 DragShowNoLock()成员函数。

在完成拖动图像后调用 EndDrag()函数。

SetDragCursorImage()成员函数通过将给定图像（一般为鼠标光标图像）和当前拖动图像合并来创建新的拖动图像。由于拖动函数在拖动操作过程中使用新图像，因此在调用 SetDragCursorImage()后应使用 Windows ShowCursor()函数来隐藏实际的鼠标光标；否则，在拖动操作期间系统可能显示两个鼠标光标。

应用程序调用 BeginDrag()后，系统创建一个临时的内部图像列表并将指定的拖动图像复制到内部列表。可以使用 GetDragImage()成员函数来检索指向临时拖动图像列表的指针，该函数还检索当前拖动位置和拖动图像相对于拖动位置的偏移量。

7．图像列表中的图像信息

CImageList 包括若干个检索图像列表信息的函数。GetImageInfo()成员函数用单一图像的有关信息填充 IMAGEINFO 结构，信息包括图像和屏蔽位图的句柄、颜色平面数和每像素的位数及图像位图内图像的边框。可以使用此信息直接操作图像的位图。

GetImageCount()成员函数检索图像列表中的图像数。

ExtractIcon()成员函数可以在图像列表中图像和屏蔽的基础上创建图标，该函数返回新图标的句柄。

下面通过一个具体的实例来讲解一下 ImageList。

首先需要设置列表控件所使用的 ImageList，如果使用大图标显示风格，就需要如下形式的调用：

```
CImageList* SetImageList( CImageList* pImageList, LVSIL_NORMAL);
```

如果使用其他 3 种风格显示而不想显示图标，可以不进行任何设置，否则需要如下形式的调用：

```
CImageList* SetImageList( CImageList* pImageList, LVSIL_SMALL);
int InsertItem( int nItem, LPCTSTR lpszItem );                    //插入行
```

其中，nItem 指明插入位置；lpszItem 为显示字符。

除 LVS_REPORT 风格外，其他 3 种风格都只需要直接调用 InsertItem()就可以，但如果使用报表风格，就必须先设置列表控件中的列信息。

InsertColumn()函数用于插入列：

```
int InsertColumn( int nCol, LPCTSTR lpszColumnHeading, int nFormat , int nWidth, int nSubItem);
```

其中，iCol 为列的位置，从 0 开始；lpszColumnHeading 为显示的列名；nFormat 为显示对齐方式；nWidth 为显示宽度；nSubItem 为分配给该列的列索引。

SetItemText()函数用于设置每列的显示字符：

```
BOOL SetItemText(int nItem, int nSubItem, LPTSTR lpszText);
```

其中，nItem 为行位置；nSubItem 为列位置；lpszText 为显示字符。

下面的代码演示了如何设置多列并插入数据：

```
m_list.SetImageList(&m_listSmall,LVSIL_SMALL);                    //设置 ImageList
m_list.InsertColumn(0,"Col 1",LVCFMT_LEFT,300,0);                 //插入列
m_list.InsertColumn(1,"Col 2",LVCFMT_LEFT,300,1);
m_list.InsertColumn(2,"Col 3",LVCFMT_LEFT,300,2);
m_list.InsertItem(0,"Item 1_1");                                  //插入行
m_list.SetItemText(0,1,"Item 1_2");                               //设置该行的不同列的显示字符
m_list.SetItemText(0,2,"Item 1_3")

COLORREF GetTextColor()/BOOL SetTextColor( COLORREF cr );
//用于得到/设置显示的字符颜色
COLORREF GetTextBkColor()/BOOL SetTextBkColor( COLORREF cr );
//用于得到/设置显示的背景颜色
void SetItemCount( int iCount );
//用于得到添加进列表中的项的数量
BOOL DeleteItem(int nItem);
//用于删除某一项
BOOL DeleteAllItems();
//将删除所有项
BOOL SetBkImage(HBITMAP hbm, BOOL fTile , int xOffsetPercent, int yOffsetPercent);
//用于设置背景位图
CString GetItemText( int nItem, int nSubItem );
//用于得到某项的显示字符
```

5.4.6 向控件添加列（报表视图）

列表控件在报表视图中时显示列，提供一种组织每一列表控件项的各种子项的方法，这种组织是通过列表控件中的列和列表控件项的相关子项之间的一一对应来实现的。Windows 95 和 Windows 98 资源管理器中的“详细信息”视图提供了报表视图中列表控件的示例。第一列列出文件夹、文件图标和标签，其他列列出文件大小、文件类型、上次修改日期等。

这些过程适用于 CListView 或 CListCtrl 对象。

虽然可以在任何时间向列表控件添加列，但是只有控件的 LVS_REPORT 样式为打开时列才可见。

每一列都有一个关联的标头项（CHeaderCtrl）对象，该对象标记列并允许用户调整列的大小。

如果列表控件支持报表视图，则需要为列表控件项中每一个可能的子项添加一列。要添加列，先准备 LV_COLUMN 结构，然后调用 InsertColumn()函数。添加必要的列（有时称标头项）后，可以使用属于嵌入式标头控件的成员函数和样式来对它们重新排序。

如果用 LVS_NOCOLUMNHEADER 样式创建列表控件，则将忽略任何插入列的尝试。

5.4.7　向控件添加项

若要向列表控件（CListCtrl）添加项，请根据所拥有的信息调用 InsertItem()成员函数的几种版本中的一种。一种版本采用准备的 LV_ITEM 结构。由于 LV_ITEM 结构包含大量成员，因此开发人员对列表控件项的属性具有更强大的控制能力。

LV_ITEM 结构的两个重要成员（相对于报表视图）是 iItem 和 iSubItem。iItem 成员是该结构正在引用的项的从 0 开始的索引，iSubItem 成员是子项的从 1 开始的索引（如果该结构包含有关某项的信息，则从 0 开始）。用这两个成员确定每项的子项信息的类型和值，此信息在列表控件位于报表视图中时显示。

其他成员指定项的文本、图标、状态和项数据。“项数据”是与列表视图项关联的应用程序定义值。

其他版本的 InsertItem()采用一个或多个与 LV_ITEM 结构中的成员相对应的独立值，使开发人员得以只初始化想要支持的成员。通常情况下，列表控件管理列表项的存储，但使用“回调项”可以将一些信息存储在应用程序中。

5.4.8　在列表控件中滚动、排列、排序和查找

默认情况下，列表控件（CListCtrl）是可滚动的。

可以调用 CListCtrl 的成员函数排列控件中的列表项，对项排序和查找特定的项。

5.4.9　在列表控件中实现工作区

默认情况下，列表控件按标准网格格式排列所有项，但也支持另一种方法：工作区。工作区将列表项排列到矩形组中。

工作区只有当列表控件处于图标或小图标模式时才可见。但是，如果视图切换到报表或列表模式，将维持任何当前工作区。

工作区可用来显示空边框（在项的左侧、顶部和/或右侧），或在通常不会有水平滚动条的时候显示水平滚动条。

另一个普通用法是创建多个工作区，并将项移动或放置到这些工作区，使用该方法可以在单一视图中创建具有不同意义的区域，然后将项放置到不同的区域以对它们进行分类。此类示例可以是一个文件系统的视图，它包含一个读/写文件区和一个只读文件区，如果将文件项移动到只读区，它将自动变为只读。将文件从只读区移动到读/写区会使文件成为可读/写的。

CListCtrl 为创建和管理列表控件工作区提供了几个成员函数：GetWorkAreas()和 SetWorkAreas () 检索并设置 CRect 对象（或 RECT 结构）数组，该数组存储当前实现的列表控件 Control）工作区；GetNumberOfWorkAreas()检索列表控件的当前工作区数目（默认值为 0）。

创建工作区后，工作区中的项成为其成员；同样，如果将一个项移动到某个工作区，则该项成为它所移动到的工作区的成员。如果某个项不在任何工作区中，则它自动成为第一个工作区（索引 0）的成员。如果想创建项并将其放置到一个特定的工作区，则需要创建该项，然后调用 SetItemPosition()

将项移动到所需的工作区。

下面的示例在列表控件（m_listctrl）中实现 4 个大小相等的工作区（rcWorkAreas），每个工作区都有 5 像素宽的边框。

代码如下：

```
CRect curRect;
CSize size;
size= m_listctrl.ApproximateViewRect();
size.cx+= 100;
size.cy+= 100;
CRect rcWorkAreas[4];
rcWorkAreas[0].SetRect(0, 0, (size.cx / 2) - 5, (size.cy / 2) - 5);
rcWorkAreas[1].SetRect((size.cx / 2) + 5, 0, size.cx, (size.cy / 2) - 5);
rcWorkAreas[2].SetRect(0, (size.cy / 2) + 5, (size.cx / 2) - 5, size.cy);
rcWorkAreas[3].SetRect((size.cx / 2) + 5, (size.cy / 2) + 5, size.cx, size.cy);
//set work areas
m_listctrl.SetWorkAreas(4, rcWorkAreas);
```

调用 ApproximateViewRect()来获取在一个区域显示所有项所需的总区域大小的估计值。之后，该估计值被分成 4 个区域并用 5 像素宽的边框填充。

下一个示例将现有列表项分配给每一个特定的工作区（rcWorkAreas），并刷新控件视图（m_listctrl）以实现此效果。

代码如下：

```
// set insertion points for each work area
CPoint rgptWork[4];
for (int i = 0; i < 4; i++)
{
        rgptWork[i].x = rcWorkAreas[i].left + 10;
        rgptWork[i].y = rcWorkAreas[i].top + 10;
}
// now move all the items to the different quadrants
for (i = 0; i < 20; i++)
        m_listctrl.SetItemPosition(i, rgptWork[i % 4]);

// force the control to rearrange the shuffled items
m_listctrl.Arrange(LVA_DEFAULT);
```

5.4.10 处理列表控件中的通知消息

用户单击列标题、拖动图标、编辑标签时，列表控件（CListCtrl）将通知消息发送给其父窗口。如果要进行某种响应，请处理这些消息，例如，用户单击列标题后，可能想基于单击的列的内容对项排序，如在 Microsoft Outlook 中所做的那样。

在视图类或对话框类中处理列表控件的 WM_NOTIFY 消息，基于正在处理的通知消息，用“属性”面板创建带 switch 语句的 OnChildNotify()处理函数。

5.4.11 更改列表控件样式

创建列表控件（CListCtrl）后，可以在任何时间更改其窗口样式。通过更改窗口样式，可以更改

控件使用的视图类型。例如，为了模拟“资源管理器”，可以提供菜单命令或工具栏按钮以在不同的视图（图标视图、列表视图等）之间切换控件。

例如，用户选择菜单命令后，可以调用 GetWindowLong()来检索控件的当前样式，然后调用 SetWindowLong()来重置样式。

可用样式在 Create()中列出。样式 LVS_ICON、LVS_SMALLICON、LVS_LIST 和 LVS_REPORT 用于指定 4 种列表控件视图。

5.4.12　虚拟列表控件

虚拟列表控件是指具有 LVS_OWNERDATA 样式的列表视图控件。该样式启用控件来支持的项数达到 DWORD（默认的项数只扩展到 int），该样式的最大便利是可以使内存中一次只有一个数据项子集，这使得虚拟列表视图（ListView）控件可以将自己借给大型信息数据库使用，而在这类数据库中已存在特定的数据访问方法。

MFC 除了在 CListCtrl 中提供虚拟列表功能外，还在 CListView 类中提供相同的功能。

在开发虚拟列表控件时应注意一些兼容性问题。

1. 处理 LVN_GETDISPINFO 通知

虚拟列表控件维护非常少的项信息，除了项选择和焦点信息，所有项信息都由控件的所有者管理。框架通过 LVN_GETDISPINFO 通知消息来请求信息。若要提供请求的信息，虚拟列表控件的所有者（或控件本身）必须处理该通知。使用“属性”面板可以很容易地完成此操作（请参见将消息映射到函数）。所得到的代码应类似于下面的示例（其中 CMyListCtrl 是虚拟列表控件对象，控件正在处理通知消息）。

```
BEGIN_MESSAGE_MAP(CMyListCtrl, CListCtrl)
    ON_NOTIFY_REFLECT(LVN_GETDISPINFO, OnGetdispinfo)
END_MESSAGE_MAP()
```

在 LVN_GETDISPINFO 通知消息的处理程序中，首先必须检查正在请求的信息的类型。

- LVIF_TEXT：必须填写 pszText 成员。
- LVIF_IMAGE：必须填写 iImage 成员。
- LVIF_INDENT：必须填写 iIndent 成员。
- LVIF_PARAM：必须填写 lParam 成员。
- LVIF_STATE：必须填写 state 成员。

然后应将所有请求的信息提供给框架。

下面的示例摘自列表控件对象的通知处理程序体，它通过为文本缓冲区和项的图像提供信息来说明一种可能的方法：

```
LV_DISPINFO* pDispInfo = (LV_DISPINFO*)pNMHDR;
LV_ITEM* pItem= &(pDispInfo)->item;

int iItemIndx= pItem->iItem;

if (pItem->mask & LVIF_TEXT) //valid text buffer
{
    switch(pItem->iSubItem){
        case 0: //fill in main text
```

```
                lstrcpy(pItem->pszText,
                    m_Items[iItemIndx].m_strItemText);
                break;
            case 1: //fill in sub item 1 text
                lstrcpy(pItem->pszText,
                    m_Items[iItemIndx].m_strSubItem1Text);
                break;
            case 2: //fill in sub item 2 text
                lstrcpy(pItem->pszText,
                    m_Items[iItemIndx].m_strSubItem2Text);
                break;
        }
    }

    if pItem->mask & LVIF_IMAGE) //valid image
            pItem->iImage=
                m_Items[iItemIndx].m_iImageIndex;
```

2. 缓存和虚拟列表控件

由于缓存和虚拟列表控件是提供给大的数据集的，因此建议缓存请求的项数据以提高检索性能。框架提供缓存提示机制，通过发送 LVN_ODCACHEHINT 通知消息来帮助优化缓存，但是，必须使用一种稍有不同的方法来处理该通知。使用“属性”面板，重写列表控件对象的 OnChildNotify()函数。在该示例的情况下，控件为 CMyListCtrl。

在处理程序体中检查 LVN_ODCACHEHINT 消息，如果找到，则准备缓存。

下面的示例（摘自 OnChildNotify()函数体）执行此检查并调用 CMyListCtrl 类的 PrepCache()成员函数：

```
NMLVCACHEHINT* pcachehint=NULL;

if (message == WM_NOTIFY)
    {
        NMHDR* phdr = (NMHDR*)lParam;

        switch(phdr->code)
        {
            case LVN_ODCACHEHINT:
                pcachehint= (NMLVCACHEHINT*) phdr;
// Load the cache with the recommended range.
                PrepCache(pcachehint->iFrom, pcachehint->iTo);
                break;
            default:
                return CListCtrl::OnChildNotify(message, wParam, lParam, pLResult);
        }
        return False;
    }
    else
        return CListCtrl::OnChildNotify(message, wParam, lParam, pLResult);
```

如果消息类型不是 LVN_ODCACHEHINT，则通知将传递给基类 (CListCtrl)。

3. 查找特定的项

当需要查找特定的列表控件项时，虚拟列表控件发送 LVN_ODFINDITEM 通知消息。列表视图控件接收快捷键访问或接收 LVM_FINDITEM 消息时发送该通知消息。搜索信息以 LVFINDINFO 结构的格式发送，该结构是 NMLVFINDITEM 结构的成员。通过重写列表控件对象的 OnChildNotify()函数来处理该消息，并在处理程序体中检查 LVN_ODFINDITEM 消息，如果找到此消息，则执行相应的操作。

应该准备好搜索与列表视图控件给定的信息匹配的项，如果搜索到匹配项，则应返回项的索引；否则返回 -1。

5.4.13　列表控件的消息映射

列表控件的消息映射同样使用 ON_NOTIFY 宏，形式如下：

```
N_NOTIFY( wNotifyCode, id, memberFxn )
```

其中的参数 wNotifyCode 为通知代码，id 为产生该消息的窗口 ID，memberFxn 为处理函数，函数的原型如同 void OnXXXList(NMHDR* pNMHDR，LRESULT* pResult)，其中 pNMHDR 为一个数据结构，在具体使用时需要转换成其他类型的结构。列表控件可能的取值和对应的数据结构如表 5.9 所示。

表 5.9　　列表控件可能取值和对应的数据结构

取　值	意　义	数 据 结 构
LVN_BEGINLABELEDIT	在开始某项编辑字符时发送	NMLVDISPINFO
LVN_ENDLABELEDIT	在结束某项编辑字符时发送	NMLVDISPINFO
LVN_GETDISPINFO	在需要得到某项信息时发送，如得到某项的显示字符	NMLVDISPINFO

关于动态提供结点所显示的字符。

首先需要指明 lpszItem 参数为 LPSTR_TEXTCALLBACK。在控件显示该结点时会通过发送 TVN_GETDISPINFO 来取得所需要的字符，在处理该消息时先将参数 pNMHDR 转换为 LPNMLVDISPINFO，然后填充其中的 item.pszText。通过 item 中的 iItem，iSubItem 可以知道当前显示的为哪一项。下面的代码演示了这种方法：

```
char szOut[8][3]={"No.1","No.2","No.3"};                    //添加结点
m_list.InsertItem(LPSTR_TEXTCALLBACK,...)
m_list.InsertItem(LPSTR_TEXTCALLBACK,...)                   //处理消息
void CParentWnd::OnGetDispInfoList(NMHDR* pNMHDR, LRESULT* pResult)
{
    LV_DISPINFO* pLVDI = (LV_DISPINFO*)pNMHDR;
    pLVDI->item.pszText=szOut[pTVDI->item.iItem];           //通过 iItem 得到需要显示的字符在数组中的位置
    *pResult = 0;
}
```

关于编辑某项的显示字符。

首先需要设置列表控件的 LVS_EDITLABELS 风格（在报表风格中只对第一列有效），在开始编辑时该控件将会发送 LVN_BEGINLABELEDIT，可以通过在处理函数中返回 True 来取消接下来的编辑；在编辑完成后会发送 LVN_ENDLABELEDIT，在处理该消息时需要将参数 pNMHDR 转换为 LPNMLVDISPINFO，然后通过其中的 item.pszText 得到编辑后的字符，并重置显示字符。如果编辑在中途中取消，该变量为 NULL。下面的代码说明如何处理这些消息：

```
//处理消息  LVN_BEGINLABELEDIT
void CParentWnd::OnBeginEditList(NMHDR* pNMHDR, LRESULT* pResult)
{
    LV_DISPINFO* pLVDI = (LV_DISPINFO*)pNMHDR;
    if(pLVDI->item.iItem==0);                          //判断是否取消该操作
        *pResult = 1;
    Else
        *pResult = 0;
}
//处理消息 LVN_BEGINLABELEDIT
void CParentWnd::OnBeginEditList(NMHDR* pNMHDR, LRESULT* pResult)
{
    LV_DISPINFO* pLVDI = (LV_DISPINFO*)pNMHDR;
    if(pLVDI->item.pszText==NULL);                     //判断是否已经取消编辑
        m_list.SetItemText(pLVDI->item.iItem,0,pLVDI->pszText);
        //重置显示字符
        *pResult = 0;
}
```

使用上面讲述的方法所进行的消息映射必须在父窗口中处理（同样，WM_NOTIFY 的所有消息都需要在父窗口中处理）。

得到当前选中项位置的方法：在列表控件中没有一个类似于 ListBox 中 GetCurSel()的函数，但是可以通过调用 GetNextItem(-1, LVNI_ALL | LVNI_SELECTED);得到选中项位置。

5.4.14 列表控件的风格选项及表头设置

List Control 中的风格选项设置：

```
m_list1.SetExtendedStyle( LVS_EX_FULLROWSELECT|LVS_EX_GRIDLINES|LVS_EX_CHECKBOXES );
```

其中，参数 LVS_EX_FULLROWSELECT 表示可以单击行中的任意一列来选中这条记录；LVS_EX_GRIDLINES 表示列之间有分隔符号；LVS_EX_CHECKBOXES 表示每一行的第一列都是 Check Box。

```
LVCOLUMN:listviewcolumn
```

下面通过这段代码讲解一下应该如何设置表头。

```
lvColumn.mask = LVCF_SUBITEM|LVCF_TEXT|LVCF_WIDTH|LVCF_FMT;    //设置表头风格
vColumn.fmt = LVCFMT_CENTER;                                   //设置表头对齐方式
lvColumn.iSubItem = i;                                         //表头列序
lvColumn.pszText = HeaderTxt;                                  //表头名称
lvColumn.cx = 90;                                              //表头宽度
m_list.InsertColumn(i,&lvColumn);                              //插入列
```

5.4.15 销毁列表控件

如果嵌入 CListCtrl 对象作为视图或对话框类的数据成员，则当销毁其所有者时该对象也被销毁；如果使用 CListView，则框架在销毁视图时销毁控件。

如果安排将一些列表数据存储在应用程序中而不是列表控件中，将需要安排对列表数据的解除分配。

另外，还要负责将所创建的与此列表控件对象关联的所有图像列表解除分配。

最后，通过一个完整的程序来讲解一下列表控件的完整的开发过程。

1．初始化 List

```
DWORD dwStyle = m_listData.GetExtendedStyle();
m_listData.SetExtendedStyle(dwStyle|LVS_EX_FULLROWSELECT|LVS_EX_GRIDLINES);
m_listData.InsertColumn(0, "用户名");
m_listData.InsertColumn(1, "密码");
m_listData.InsertColumn(2, "Key ID");
m_listData.InsertColumn(3, "Key  密码");
m_listData.SetColumnWidth(0, 80);
m_listData.SetColumnWidth(1, 120);
m_listData.SetColumnWidth(2, 80);
m_listData.SetColumnWidth(3, 120);
```

2．添加记录

```
m_list.InsertItem(nIndex, strName);
m_list.SetItem(nIndex, 1,LVIF_TEXT|LVIF_IMAGE, strSex, 0, 0L,0L, 0L );
m_list.SetItem(nIndex, 2,LVIF_TEXT|LVIF_IMAGE, strLisenceId, 0, 0L,0L, 0L );
m_list.SetItem(nIndex, 3,LVIF_TEXT|LVIF_IMAGE, strDept, 0, 0L,0L, 0L );
m_list.SetItem(nIndex, 4,LVIF_TEXT|LVIF_IMAGE, strJob, 0, 0L,0L, 0L );
m_list.SetItem(nIndex, 5,LVIF_TEXT|LVIF_IMAGE, strInitTime, 0, 0L,0L, 0L );
```

3．删除记录

```
m_list.DeleteAllItems();
```

4．单击列表，获得行号

```
NM_LISTVIEW* pNMListView = (NM_LISTVIEW*)pNMHDR;
int nMask = pNMListView->iItem;//nMask 为单击选择的行号
```

5．获取列表内容

```
char szValue[260] = {0};
m_list.GetItemText(nMask, 0, szValue, 260);//name
m_list.GetItemText(nMask, 1, szValue, 260);//sex
```

5.5　Visual C++ 2010 Edit 控件简介与开发

编辑框控件可以接受用户的输入和编辑。编辑框是一个矩形子窗口，用户可以在其中输入文本信息，如文字、数字，并能用剪切、粘贴、复制、删除等操作来编辑输入的内容。编辑框既可以是单行的，也可以是多行的。当该控件获得焦点时，在其编辑框的内部会显示一个闪烁的光标，类似于编辑 Word 文档时显示的光标，但是它绝不能像 Word 一样允许用户插入表格和图片等，编辑框控件只能接受纯文本形式的输入。

与编辑控件有关的消息及其产生的原因如表 5.10 所示。

表 5.10　　编辑控件有关的消息

消　息	产生的原因
EN_ERRSPACE	编辑框控件不能分配足够的内存
EN_CHANGE	改变编辑控件中的文本
EN_VSCROLL	单击编辑控件的垂直滚动条

续表

消　息	产生的原因
EN_KILLFOCUS	编辑控件失去输入焦点
EN_SETFOCUS	编辑控件获得输入焦点
EN_UPDATE	编辑控件即将显示被更改的文本
EN_MAXTEXT	当前插入的文本超过编辑控件指定的字符数并已被截断
ES_AUTOHSCROLL	要插入的字符数超过编辑控件的宽度
ES_AUTOVSCROLL	插入文本后导致总行数超过编辑控件的高度

编辑控件类 CEdit 的成员函数及其功能如表 5.11 所示。

表 5.11　　类 CEdit 的主要成员函数

函　数	说　明
Create()	负责创建编辑框控件
GetSel()	得到编辑框控件中当前选择的起始和结束字符位置
ReplaceSel()	用指定的内容替换编辑框控件的当前选择
SetSel()	设置编辑框控件中的字符范围
Clear()	清除编辑框控件中的当前选择
Copy()	以 CF_TEXT 格式复制编辑框控件中的当前选择到剪贴板中
Cut()	以 CF_TEXT 格式删除编辑框控件中的当前选择并将其复制到剪贴板中
Paste()	以 CF_TEXT 格式从剪贴板复制数据到编辑框控件的当前位置
Undo()	撤销最后的编辑框控件操作
CanUndo()	判断编辑框控件操作是否可以撤销
EmptyUndoButter()	重设或消除编辑框控件的撤销标记
GetModify()	判断编辑框控件的内容是否已修改
SetModify()	设置或消除编辑框控件的修改标记
SetReadOnly()	设置编辑框控件为只读状态
GetPassWordChar()	得到编辑框控件中显示的口令字符
SetPassWordChar()	设置编辑框控件中显示的口令字符
GetFirstVisibleLine()	判断编辑框控件中最顶端的可见行
LineLength()	得到编辑框控件的行长度
LineScroll()	滚动编辑框控件中指定行的文本
LineFromChars()	得到包含指定字符索引的行号
GetRect()	得到编辑框控件的格式化区域
LimitText()	限制可以输入到编辑框控件中的文本长度
GetLineCount()	得到多行编辑框控件的行数
GetLine()	从多行编辑框控件中得到指定的文本行

MFC 的 CEdit 类封装了编辑框控件。CEdit 类的成员函数 Create()负责创建编辑框控件，该函数原型如下：

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

该函数如果创建成功，返回 True，否则返回 False。

第一个参数 dwStyle 指定了编辑框控件的风格，dwStyle 可以为下列值的组合：

```
WS_CHILD        指定该控件是一个子窗口
WS_VISIBLE      指定该控件可显示
WS_DISABLED     指定该控件不可显示
```

WS_GROUP	指定该类控件作为一组
WS_TABSTOP	使控件具有 Tabstop 属性
WS_BORDER	使控件带边框
ES_AUTOHSCROLL	当用户在行尾输入一个字符时，正文将自动向右滚动 10 个字符，当用户按【Enter】键时，正文总是滚向左边
ES_AUTOVSCROLL	当用户在最后一个可见行按【Enter】键时，正文向上滚动一页
ES_CENTER	在多行编辑框中使正文居中
ES_LEFT	左对齐正文
ES_RIGHT	右对齐正文
ES_LOWERCASE	把用户输入的字母统一转换成小写字母
ES_UPPERCASE	把用户输入的字母统一转换成大写字母
ES_MULTILINE	指定一个多行编辑器
ES_PASSWORD	使所有输入的字符都用“*”来显示
ES_READONLY	将编辑框设置为只读的
ES_WANTRETURN	使多行编辑器接受【Enter】键输入并换行。如果不指定该风格，按【Enter】键会选择默认的命令按钮，这往往会导致对话框的关闭

下面给出常用的两种方案：如果想要创建一个普通的单行编辑框，应指定风格为 WS_CHILD|WS_VISIBLE|WS_TABSTOP |WS_BORDER|ES_LEFT|ES_AUTOHSCROLL，这将创建一个带边框、左对齐正文、可水平滚动的单行编辑器；如果要创建一个普通多行编辑框，还要附加 ES_MULTILINE|ES_WANTRETURN|ES_AUTOVSCROLL |WS_HSCROLL| WS_VSCROLL 风格，这将创建一个可水平和垂直滚动的、带有水平和垂直滚动条的多行编辑器。

第二个参数 rect 是一个对 RECT 或 CRect 结构的引用，指定了编辑框的位置和尺寸。

第三个参数 pParentWnd 指定了父窗口，不能为 NULL。

第四参数 nID 指定了编辑框的 ID。

```
例如：CEdit* pEdit = new CEdit;
    pEdit->Create(ES_MULTILINE | WS_CHILD | WS_VISIBLE | WS_TABSTOP | WS_BORDER, CRect(10, 10, 100, 100), this, 1);
```

5.6　Visual C++ 2010 Rich Edit 控件简介与开发

Rich Edit 控件可以看作一种功能扩展后的 Edit 控件。它提供了普通 Edit 控件所缺少的很多令人满意的功能，如可以使用多种字体和字体大小，多级 Undo/Redo，文本搜索，内嵌 OLE 对象，编辑的拖放支持等。

Rich Edit 控件（CRichEditCtrl）为格式化文本提供了一个编程接口。但是，应用程序必须实现任何使用户执行格式化操作所需的用户界面组件，也就是说，Rich Edit 控件支持更改选定文本的字符或段落属性。字符属性的示例包括粗体、斜体、字体系列和磅值；段落属性的示例包括对齐、页边距和制表位。但是，不论是工具栏按钮、菜单命令还是格式字符对话框，都由开发人员决定是否提供用户界面。也有向 Rich Edit 控件查询当前选定内容属性的函数，使用这些函数以显示属性的当前设置，例如，如果选定项有粗体字符格式属性，则在命令 UI 上设置复选标记。

Rich Edit 控件支持几乎所有用于多行编辑控件的操作和通知消息，因此，使用编辑控件的应用程序可以很容易更改为使用 Rich Edit 控件的程序。其他消息和通知使应用程序能够访问 Rich Edit 控件特有的功能。

与 Rich Edit 有关的类有 CRichEditView、CRichEditDoc 和 CRichEditCntrItem 类，它们在 MFC 的

文档/视图结构上下文中提供了 Rich Edit 控件（CRichEditCtrl）的功能。CRichEditView 维护文本和文本特有的格式特征，CRichEditDoc 维护此视图中 OLE 客户项的列表，CRichEditCntrItem 提供对 OLE 客户项容器端的访问。若要修改 CRichEditView 的内容，请使用 CRichEditView::GetRichEditCtrl()访问基础 Rich Edit 控件。

由于 Rich Edit 控件有这么多功能，因此它独存于一个单独的 DLL 文件中，这意味着如果要使用 Rich Edit 控件，就不能像使用其他普通控件那样只调用 InitCommonControls()函数，而必须先使用 LoadLibrary()来载入 Rich Edit 的 DLL 文件。

至今为止，Rich Edit 控件共有 3 个版本，版本 1.0、2.0 和 3.0。表 5.12 显示了每个版本对应的 DLL 文件名。

表 5.12　　DLL 文件名

DLL 文件名	Rich Edit 版本	RichEdit 类名
Riched32.dll	1.0	RICHEDIT
RichEd20.dll	2.0	RICHEDIT20A
RichEd20.dll	3.0	RICHEDIT20A

其中，Visual Studio 2010 中的 Rich Edit 控件采用的是版本 2.0。

版本 2.0 和 3.0 都使用了相同的 DLL 文件名，而且它们也使用了相同的类名！到目前为止，还没有一种正式的方法来区分版本 2.0 和版本 3.0。

当 Rich Edit DLL 文件被载入时，它会注册 RichEdit 窗口类，因此在创建 Rich Edit 控件之前必须先载入 DLL。Rich Edit 控件的类名是不同的，现在已经知道 Microsoft Visual Studio 2010 中的 Rich Edit 采用的是版本 2.0，但假若不知道，应该怎么样判断所使用的是哪个版本的 Rich Edit 控件呢？如果不需要使用那些特别的功能，使用最新的版本也不一定是最适宜的。表 5.13 列出了每个 Rich Edit 版本提供的功能。

表 5.13　　每个 RichEdit 版本提供的功能

功　能	版本 1.0	版本 2.0	版本 3.0
选择条	x	x	x
Unicode 编辑		x	x
Unicode 编辑		x	x
字符段落格式	x	x	x
文本搜索	向前搜索	向前/向后搜索	向前/向后搜索
内嵌 OLE	x	x	x
拖放编辑	x	x	x
Undo/Redo	单级	多级	多级
自动 URL 识别		x	x
加速键支持		x	x
非窗口操作		x	x
分行符	CRLF	CR	CR（可以模拟 1.0 版本）
Zoom			x
Paragraph numbering			x
simple table			x
normal and heading styles			x
underline coloring			x
hidden text			x
font binding			x

表 5.13 中列出的只是各个版本的重要的功能。

5.6.1　Rich Edit 控件中的字符格式

可以使用 Rich Edit 控件（CRichEditCtrl）的成员函数格式化字符并检索格式信息。对于字符，可以指定字体、大小、颜色和粗体、斜体、受保护等效果。

可以通过使用 SetSelectionCharFormat()和 SetWordCharFormat()成员函数来应用字符格式。若要确定选定文本的当前字符格式，应使用 GetSelectionCharFormat()成员函数。CHARFORMAT 结构与这些成员函数一起使用以指定字符属性。CHARFORMAT 的重要成员之一是 dwMask。在 SetSelectionCharFormat() 和 SetWordCharFormat() 中，dwMask 指定将由此函数调用设置的字符属性。GetSelectionCharFormat()报告选定内容中第一个字符的属性；dwMask 指定在整个选定内容中一致的属性。

也可以获取并设置“默认字符格式”，该格式应用于任何随后插入的字符。例如，如果应用程序将默认字符格式设置为粗体并且用户随后输入一个字符，那个字符就是加粗的。若要获取和设置默认字符格式，应该使用 GetDefaultCharFormat()和 SetDefaultCharFormat()成员函数。

“受保护的”字符属性不更改文本的外观。如果用户试图修改受保护的文本，Rich Edit 控件将向其父窗口发送一个 EN_PROTECTED 通知消息，使父窗口可以允许或禁止此项更改。若要接收此通知消息，必须使用 SetEventMask()成员函数启用此消息。

前景色是字符的属性，而背景色是 Rich Edit 控件的属性。若要设置背景色，应该使用 SetBackgroundColor()成员函数。

5.6.2　Rich Edit 控件中的段落格式

可以使用 Rich Edit 控件（CRichEditCtrl）的成员函数格式化段落并检索格式信息。段落格式属性包括对齐、制表符、缩进和编号。

可以通过使用 SetParaFormat()成员函数应用段落格式。若要确定选定文本的当前段落格式，应该使用 GetParaFormat()成员函数。PARAFORMAT 结构与这些成员函数一起使用以指定段落属性。PARAFORMAT 的重要成员之一是 dwMask。在 SetParaFormat()中，dwMask 指定将由此函数调用设置的段落属性。GetParaFormat()报告选定内容中第一个段落的属性；dwMask 指定在整个选定内容中一致的属性。

5.6.3　Rich Edit 控件中的当前选定内容

用户可以使用鼠标或者键盘选择 Rich Edit 控件（CRichEditCtrl）中的文本。当前选定内容就是选定字符的范围；如果未选定任何字符，则是插入点的位置。应用程序可以获取关于当前选定内容的信息，设置当前选定内容，确定当前选定内容何时更改，以及显示或隐藏选定内容的突出显示。

若要确定 Rich Edit 控件中的当前选定内容，请使用 GetSel()成员函数；若要设置当前选定内容，请使用 SetSel()成员函数。CHARRANGE 结构与这些函数一起使用以指定字符的范围。若要检索关于当前选定内容的信息，可使用 GetSelectionType()成员函数。

默认情况下，Rich Edit 控件在获得或失去焦点时显示或隐藏选定内容的突出显示。通过使用 HideSelection()成员函数，可以在任何时候显示或隐藏选定内容的突出显示。例如，应用程序可能会提供一个“搜索”对话框，用来查找 Rich Edit 控件中的文本；应用程序可能选择匹配文本但不关闭此对

话框，这种情况下必须使用 HideSelection()突出显示选定内容。

若要获取 Rich Edit 控件中的选定文本，应使用 GetSelText()成员函数。该文本被复制到指定的字符数组。必须确保该数组足够大，可以包含选定文本外加终止 NULL 字符。

通过使用 FindText()成员函数，可以搜索 Rich Edit 控件中的字符串。与该函数一起使用的 FINDTEXTEX 结构指定要搜索的文本范围和要搜索的字符串，也可以指定一些选项，如搜索是否区分大小写。

5.6.4 Rich Edit 控件中的分词

Rich Edit 控件（CRichEditCtrl）调用一个名为“word break procedure”的函数，以查找单词之间的断点并确定可以在何处分行。该控件在执行自动换行操作和处理【Ctrl+←】和【Ctrl+→】组合键时使用此信息。应用程序可以将消息发送到 Rich Edit 控件以替换默认的分词过程，检索分词信息和确定给定字符落在哪一行上。

5.6.5 Rich Edit 控件中的剪贴板操作

通过使用最常用的剪贴板格式或者特定的剪贴板格式，应用程序可以将剪贴板的内容粘贴到 Rich Edit 控件（CRichEditCtrl）中，开发人员也可以确定 Rich Edit 控件是否有能力粘贴剪贴板格式。

通过使用 Copy()或 Cut()成员函数，可以复制或剪切当前选定内容。同样，通过使用 Paste()成员函数，可以将剪贴板的内容粘贴到 Rich Edit 控件中。该控件粘贴所识别的第一个可用格式，该格式大概是最具描述性的格式。

若要粘贴特定的剪贴板格式，可以使用 PasteSpecial()成员函数，此函数对具有使用户能够选择剪贴板格式的“选择性粘贴”命令的应用程序很有用。可以使用 CanPast()成员函数确定给定格式是否可由此控件识别。

也可以使用 CanPaste()确定 Rich Edit 控件是否识别任何可用的剪贴板格式，此函数对 OnInitMenuPopup 处理程序很有用。应用程序可能启用“粘贴”命令或使其变灰，具体取决于该控件是否能够粘贴任何可用格式。

Rich Edit 控件注册两种剪贴板格式：RTF 格式和一个名为 RichEdit Text and Objects 的格式。通过使用 RegisterClipboardFormat()函数并指定 CF_RTF 和 CF_RETEXTOBJ 值，应用程序可以注册这些格式。

5.6.6 Rich Edit 控件中的流操作

可以使用流将数据传入或传出 Rich Edit 控件（CRichEditCtrl）。流通过 EDITSTREAM 结构来定义，该结构指定缓冲区和应用程序定义的回调函数。

若要将数据读入 Rich Edit 控件（即将数据流入），请使用 StreamIn()成员函数。该控件重复调用应用程序定义的回调函数，每次将这些数据的一部分传入缓冲区。

若要保存 Rich Edit 控件的内容（即将数据流出），应该使用 StreamOut()成员函数。该控件向缓冲区重复写入，然后调用应用程序定义的回调函数。对于每个调用，回调函数都保存缓冲区的内容。

5.6.7 Rich Edit 控件中的打印操作

可以通知 Rich Edit 控件（CRichEditCtrl）为指定设备（如打印机）呈现输出，也可以指定输出设

备，Rich Edit 控件将为该设备格式化其文本。

若要为特定设备格式化 Rich Edit 控件的部分内容，可以使用 FormatRange()成员函数。FormatRange 结构与此函数一起使用，可以指定格式化的文本范围和目标设备的设备上下文（DC）。

为输出设备格式化文本后，可以使用 DisplayBand()成员函数将输出发送到该设备。通过重复使用 FormatRange()和 DisplayBand()，打印 Rich Edit 控件内容的应用程序可以实现带区划分。

可以使用 SetTargetDevice()成员函数指定目标设备，Rich Edit 控件将为该设备格式化其文本。该函数对 WYSIWYG（所见即所得）格式很有用，这类格式的应用程序定位文本所使用的是默认打印机的字体标准，而不是屏幕的字体标准。

5.6.8　无底的 Rich Edit 控件

应用程序可以根据需要调整 Rich Edit 控件（CRichEditCtrl）的大小，以使其与其内容的大小总相同。无论何时，其内容的大小更改时，都向其父窗口发送一个 EN_REQUESTRESIZE 通知消息，Rich Edit 控件支持这种所谓的“无底的”功能。

处理 EN_REQUESTRESIZE 通知消息时，应用程序会将此控件的大小调整为指定 REQRESIZE 结构中的尺寸。应用程序也可能移动此控件附近的任何信息，以适应此控件在高度上的更改。若要调整此控件的大小，可以使用 CWnd 类的成员函数 SetWindowPos()。

通过使用 RequestResize()成员函数，可以强制无底的 Rich Edit 控件发送 EN_REQUESTRESIZE 通知消息。此消息在 OnSize()处理程序中可能有用。

若要接收 EN_REQUESTRESIZE 通知消息，必须使用 SetEventMask()成员函数启用此消息。

5.6.9　来自 Rich Edit 控件的通知

通知消息报告影响 Rich Edit 控件（CRichEditCtrl）的事件，它们可以被父窗口处理，或者通过使用消息反射，被 Rich Edit 控件自身处理。Rich Edit 控件支持所有这些和编辑控件（Edit Control）一起使用的通知消息及几个附加的通知消息。通过设置“事件屏蔽”，可以确定 Rich Edit 控件向其父窗口发送的通知消息。

若要为 Rich Edit 控件设置事件屏蔽，请使用 SetEventMask()成员函数。通过使用 GetEventMask () 成员函数，可以为 Rich Edit 控件检索当前事件屏蔽。

下面列出了几个特定的通知及其使用。

- EN_MSGFILTER: 处理 EN_MSGFILTER 通知以使某类（或者是 Rich Edit 控件，或者是其父窗口）可以筛选对此控件的所有键盘和鼠标输入。该处理程序可以阻止键盘或鼠标消息不被处理，或者可以通过修改指定的 MSGFILTER 结构来更改此消息。
- EN_PROTECTED: 处理 EN_PROTECTED 通知消息以检测用户何时试图修改受保护的文本。若要将某个文本范围标记为受保护的，可以设置受保护字符的效果。更多的相关信息，请参见 Rich Edit 控件中的字符格式。
- EN_DROPFILES: 通过处理 EN_DROPFILES 通知消息，可以使用户将文件放置在 Rich Edit 控件中。指定的 ENDROPFILES 结构包含关于正在放置的文件信息。
- EN_SELCHANGE: 通过处理 EN_SELCHANGE 通知消息，应用程序可以检测当前选定内容何时更改。通知消息指定 SELCHANGE 结构，该结构包含关于新选定内容的信息。

下面通过一个例子来具体地讲解一下 Rich Edit 控件的创建及释放的方法。

1．创建一个单文档的 MFC 应用程序，并在程序中添加一个对话框资源

在该对话框资源的编辑窗口中，通过拖拉方式从 ToolBox 控件面板中添加一个 Rich Edit 控件到所创建的对话框上。在对话框中右击并选择“Add Class”命令创建一个对话框类。在对话框构造函数或者对话框类中创建一个 InitInstance()函数，并向其中加入初始化 Rich Edit 控件相关的代码，如下所示：

```
AfxInitRichEdit();
m_hRichEdit = LoadLibrary("RICHED20.DLL");
```

其中 HMODULE m_hRichEdit;在对应的头文件中定义。

2．设置 Rich Edit 控件中文字的显示样式

下面是一段设置 Rich Edit 控件文字样式的代码：

```
CHARFORMAT cf;
ZeroMemory(&cf, sizeof(CHARFORMAT));
cf.cbSize = sizeof(CHARFORMAT);
cf.dwMask = CFM_BOLD | CFM_COLOR | CFM_FACE |  CFM_ITALIC | CFM_SIZE | CFM_UNDERLINE;
cf.dwEffects = CFE_UNDERLINE;
cf.yHeight = 12 * 12;                                    //文字高度
 cf.crTextColor = RGB(200, 100, 255);                    //设置文字颜色
strcpy(cf.szFaceName ,_T("隶书"));                       //设置字体
m_ctrlMyRichEdit.SetDefaultCharFormat(cf);
```

其中 CRichEditCtrl m_ctrlMyRichEdit;在对应对话框的头文件中定义，并且这个变量与对话框中的一个 Rich Edit 控件相关联。

3．显示文字

```
CString strText = "Hello, World!\r\nBy Colin";
m_ctrlMyRichEdit.SetWindowText(strText );
```

其中\r\n 表示换行，当然换行的前提是 Rich Edit 控件的 MultiLine 选项已经选中。

4．在 ExitInstance()中释放 Rich Edit 相关资源

```
FreeLibrary(m_hRichEdit);
```

在 MFC 中创建 Rich Edit 时，也有一些需要注意的地方：应该在对话框构造函数中用 AfxInitRichEdit()或者 AfxInitRichEdit2()。放在 OnInitDialog()中不起作用，否则程序会运行不起来，也没有任何编译错误信息。

这两个函数是用来加载 Riched20.dll 的，其中加载动态链接库的方法是需要在头文件 stdafx.h 或构造函数中用这句代码：

```
LoadLibrary(_T("RICHED20.DLL"));
```

如果要响应 Rich Edit 控件的 OnEnChange 事件，需要在 OnInitDialog()中添加：

```
//CRichEditCtrl().SetEventMask(ENM_CHANGE);
GetDlgItem(RichEditID)->SetEventMask(ENM_CHANGE);
```

或者

```
m_CtrlRichEdit.SetEventMask(ENM_CHANGE);
```

其中 m_CtrlRichEdit 是 Rich Edit 关联的控件变量，非 CString 或其他变量。

在 Edit 控件 OnEnChange 事件中不要随便用 UpdateData(False)，这是因为当输入字符超过 256 个时，输入焦点会自动跳转到首行第一格。

5.7　Visual C++ 2010 Progress 控件简介与开发

进度控件（Progress Control）用于指示一个长操作的进度。此控件是一个矩形的窗口，在操作进行过程中，该矩形由系统的突出显示颜色逐步填充。

在 MFC 中，进度控件用类 CProgressCtrl 表示。

最初创建进度控件时，需要指定其大小和位置、父窗口（通常是一个对话框）和 ID。通过使用 dwStyle 参数，也可以指定此控件的各种窗口样式及其填充样式。

5.7.1　进度控件的样式

最初创建进度控件（CProgressCtrl::Create()）时，使用 dwStyle 参数为进度控件指定想要的窗口样式。表 5.14 详细列出了适用的窗口样式，此控件忽略任何在此没有列出的窗口样式。应总是将此控件创建为对话框父级的子窗口。

表 5.14　　窗口样式

窗 口 样 式	效　　果
WS_BORDER	创建环绕窗口的边框
WS_CHILD	创建一个子窗口（应总是用于 CProgressCtrl）
WS_CLIPCHILDREN	排除在父窗口中拖动时子窗口所占的区域。当创建父窗口时使用
WS_CLIPSIBLINGS	相对于子窗口彼此之间的位置剪辑子窗口
WS_DISABLED	创建一个开始时禁用的窗口
WS_VISIBLE	创建一个开始时可见的窗口
WS_TABSTOP	指定当用户按【Tab】键移动到此控件时，它可以接收焦点

另外，可以指定只应用于进度控件的两种样式：PBS_VERTICAL 和 PBS_SMOOTH。

使用 PBS_VERTICAL 垂直（而不是水平）定向此控件；使用 PBS_SMOOTH 完全填充此控件，而不是显示增量填充此控件的画有边线的方形。

不使用 PBS_SMOOTH 样式时的进度条如图 5.3 所示。

图 5.3　不使用 PBS_SMOOTH 样式的进度条

使用 PBS_SMOOTH 和 PBS_VERTICAL 样式时的进度条如图 5.4 所示。

图 5.4

5.7.2　进度控件的设置

进度控件的基本设置（CProgressCtrl）是范围和当前位置，范围表示操作的整个持续时间，当前位置表示应用程序完成操作的进度。对范围或位置的任何更改都将导致进度控件重绘自身。

默认情况下，范围设置为 0～100，初始位置设置为 0。若要检索进度控件当前的范围设置，请使用 GetRange()成员函数；若要更改此范围，请使用 SetRange()成员函数。

若要设置位置，请使用 SetPos()；若要检索当前位置但不指定新值，请使用 GetPos()，例如，可能希望只查询当前操作的状态。

若要单步递增进度控件的当前位置，请使用 StepIt()；若要设置每一步的数量，请使用 SetStep()。

5.7.3 操作进度控件

有 3 种可以更改进度控件（CProgressCtrl）的当前位置的方法：

- 可以用预设增量更改此位置；
- 可以用任意数量更改此位置；
- 可以将此位置更改为一个特定值。

用预设数量更改此位置的操作如下。

（1）使用 SetStep()成员函数设置增量，默认情况下，该值为 10，该值通常设置为此控件的初始设置之一，可以为负。

（2）使用 StepIt()成员函数递增此位置，这将导致此控件重画其自身。

StepIt()将导致此位置换行。例如，给定一个 1～100 的范围，步长为 20，位置是 90，StepIt()会将此位置设置为 10.

用任意数量更改此位置的操作如下：

使用 OffsetPos()成员函数更改此位置，OffsetPos()接受负值。

与 StepIt()不同，OffsetPos()不使此位置换行。新位置将调整为保持在此范围内。

将此位置更改为特定值的操作如下。

使用 SetPos()成员函数将此位置设置为特定值。如果有必要，新位置将调整为在此范围内。

进度控件通常只用于输出。若要获取当前位置但不指定新值，请使用 GetPos()。

5.8 Visual C++ 2010 Tree 控件简介与开发

CTreeCtrl 类所表示的树控件（Tree Control）是一个窗口，该窗口显示了分层的项列表，如文档的标头、索引项或磁盘上的文件和目录。每一项都由标签和可选的位图化图像组成，并且可以有与之关联的子项列表。用户通过单击某项可以展开和折叠相关子项的列表。Windows 资源管理器左窗格中的目录树就属于树控件。

MFC 提供了两个封装树控件的类：CTreeCtrl 和 CTreeView。每个类应用于不同的情况。

当需要纯子窗口控件时（比如在对话框中），使用 CTreeCtrl 类。如果窗口中还有其他子控件，正如在典型的对话框中那样，这时尤其需要使用 CTreeCtrl 类。

如果希望树控件不仅作为树控件而且还作为文档/视图结构中的视图窗口存在，则使用 CTreeView 类。CTreeView 类占有框架窗口或拆分窗口的整个工作区，其大小随父窗口大小的调整而自动调整，并且可以处理来自菜单、快捷键和工具栏的命令消息。因为树控件包含显示树所需的数据，所以相应的文档对象不必太复杂，甚至可以在文档模板中使用 CDocument 作为文档类型。

树控件（CTreeCtrl）的典型使用遵循下列模式。

（1）创建控件。如果在对话框模板中指定了控件，或者正在使用 CTreeView 类，当创建对话框或视图时也自动创建控件。如果要将树控件创建为其他某个窗口的子窗口，请使用 Create()成员函数。

（2）如果树控件要使用图像，请调用 SetImageList()设置图像列表。还可以调用 SetIndent()来调整

缩进。调整缩进最好在 OnInitDialog()（对于对话框中的控件）或 OnInitialUpdate()（对于视图）中进行。

（3）通过为每个数据项调用一次 CtreeCtrl 类的 InsertItem()函数，向控件中填充数据。InsertItem()函数返回数据项的句柄，以后可以用该句柄引用数据项（比如添加子项时）。给数据项赋初始值最好是在 OnInitDialog()（对于对话框中的控件）或 OnInitialUpdate()（对于视图）中进行。

（4）用户和控件交互过程中，控件将发送各种通知消息。通过在控件窗口的消息映射中添加 ON_NOTIFY_REFLECT 宏，或者在父窗口的消息映射中添加 ON_NOTIFY 宏，可以指定一个函数来处理想要处理的每一条消息。有关的通知消息，请参见 5.8.11 小节介绍的内容。

（5）调用各种 Set 成员函数设置控件值。可以更改的设置包括缩进、文本、图像或与项关联的数据。

（6）使用各种 Get 函数检查控件内容。有些函数使开发人员得以检索指定项的父句柄、子句柄和同辈句柄。利用这些函数可以在树控件的内容中遍历。甚至可以对具体的某个结点的子级排序。

（7）使用完控件后，一定要正确销毁控件。如果树控件在对话框中或者是一个视图，它和 CTreeCtrl 对象将会自动销毁，否则就需要开发人员确保正确地销毁该控件和 CTreeCtrl 对象。

而对于与树控件的通信，创建对象的方法不同，在 CTreeCtrl 对象中调用成员函数的方法也不同。

- 如果树控件在对话框中，使用在对话框类中创建的 CTreeCtrl 类型的成员变量。
- 如果树控件是一个子窗口，使用过去常用来构造对象的 CTreeCtrl 对象（或指针）。
- 如果正在使用 CTreeView 对象，使用函数 CTreeView::GetTreeCtrl()获取对树控件的引用。可以利用该值初始化另外一个引用，或将引用的地址分配给 CTreeCtrl 指针。

5.8.1　树控件样式

树控件（CTreeCtrl）样式决定其外观特征。创建树控件时，就设置了树控件的初始样式；创建树控件后，可以检索和更改树控件的样式，方法是利用 GetWindowLong()和 SetWindowLong()函数为 nIndex 参数指定 GWL_STYLE。

TVS_HASLINES 样式通过在子项与其相应的父项之间画上一条连线，增强了树控件层次结构的图形化表示形式，这种样式不链接层次结构根位置的项。若要链接层次结构根位置的项，需要组合 TVS_HASLINES 和 TVS_LINESATROOT 样式。

双击父项，可以展开或折叠父项的子项列表。具有 TVS_SINGLEEXPAND 样式的树控件展开选定的项，而折叠未选定的项。如果单击选定的项并且该项是关闭的，则该项将展开。如果选定的项是打开的，则单击它时，它会折叠起来。

具有 TVS_HASBUTTONS 样式的树控件在每一父项的左侧添加一个按钮。单击该按钮可展开或折叠子项，这是除了用双击父项的方法展开或折叠子项以外的另一种方法。TVS_HASBUTTONS 样式不向在层次结构根位置的项添加按钮。若要向层次结构根位置的项添加按钮，必须组合 TVS_HASLINES、TVS_LINESATROOT 和 TVS_HASBUTTONS 样式。

TVS_EDITLABELS 样式使用户编辑树控件项的标签成为可能。

TVS_NOTOOLTIPS 样式禁用树视图控件的自动工具提示功能。当将鼠标指针移到某项上时，如果该项的整个标题当时是不可见的，则自动工具提示功能自动显示工具提示，其中就包含该项的标题。

5.8.2　树控件父项和子项

树控件（CTreeCtrl）中的任何项都可以有与之关联的名为子项的列表。有一个或多个子项的项称为父项。子项显示在其父项之下，并且是缩进的，以指示子项隶属于父项的关系。没有父项的项位于

层次结构中的最高层，称为根项。

在任何时候都可以展开或折叠父项的子项列表状态。处于展开状态时，子项显示在其父项之下；处于折叠状态时，则不显示子项。双击父项，或者在父项有 TVS_HASBUTTONS 样式时单击与父项关联的按钮，子项列表就自动在展开和折叠状态间切换。应用程序可以使用 Expand()成员函数展开或折叠子项。

通过调用 InsertItem()成员函数可以向树控件添加项，该函数返回 HTREEITEM 类型的句柄，该句柄惟一标识项。添加项时，必须指定新项的父项句柄。如果在 TVINSERTSTRUCT 结构或 hParent 参数中，指定的是 NULL 或 TVI_ROOT 值，而不是父项句柄，则项添加为根项。

即将展开或折叠父项的子项列表时，树控件发送 TVN_ITEMEXPANDING 通知消息。有些父项的属性依赖于子项列表的状态，而此通知使开发人员有机会禁止更改或者设置这些父项的属性。更改子项列表状态后，树控件发送 TVN_ITEMEXPANDED 通知消息。

展开子项列表时，子项列表相对于父项缩进。可以用 SetIndent()成员函数设置缩进量，或用 GetIndent()成员函数检索当前缩进量。

5.8.3 树控件项位置

使用 InsertItem()成员函数向树控件（CTreeCtrl）添加项时设置项的初始位置，该成员函数调用指定父项的句柄和要在其后添加新项的项的句柄，后一个句柄必须标识给定父项的子项，或者标识 TVI_FIRST、TVI_LAST 或 TVI_SORT 3 个值中的一个。

若指定的值是 TVI_FIRST 或 TVI_LAST，则树控件将新项放在给定父项的子项列表的开头或结尾；若指定的值是 TVI_SORT，树控件就按照项标签文本的字母顺序将新项插入子项列表中。

通过调用 SortChildren()成员函数，可以按照字母顺序对父项的子项列表排序，这个函数中有一个参数指定了给定父项下的各级子项是否都按字母顺序排序。

SortChildrenCB()成员函数使开发人员得以按照定义的准则对子项进行排序，调用该函数时，应该指定一个应用程序定义的回调函数。每当需要确定两个子项的相对顺序时，树控件就可以调用该回调函数，该回调函数接收正在比较的项的两个 32 位应用程序定义值，和开发人员在调用 SortChildrenCB () 时指定的一个 32 位值。

5.8.4 树控件项标签

通常在向树控件（CTreeCtrl）添加项时指定项标签文本，InsertItem()成员函数能传递 TVITEM 结构。该结构定义了项的属性，项属性中有一个包含标签文本的字符串。InsertItem()有几个可用各种参数组合调用的重载。

树控件分配存储各项的内存，其中大部分内存都被项标签文本占用。如果应用程序保存了树控件中字符串的副本，就可以减少控件所需的内存空间，方法是在 TV_ITEM 的 pszText 成员中或者在 lpszItem 参数中指定 LPSTR_TEXTCALLBACK 值，而不是将实际字符串传递给树控件。每当需要重绘某项时，LPSTR_TEXTCALLBACK 使树控件从应用程序中检索该项的标签文本。为了检索标签文本，树控件发送 TVN_GETDISPINFO 通知消息，该消息包括 NMTVDISPINFO 结构的地址，必须通过设置所含结构的适当成员来响应该通知消息。

树控件使用从创建它的进程堆分配的内存，它最多可以包含的项数取决于堆中可用的内存量，每个项占用 64 个字节。

5.8.5　树控件标签编辑

用户可以在具有 TVS_EDITLABELS 样式的树控件 CTreeCtrl）中直接编辑项标签，单击具有焦点的项标签即可开始编辑。应用程序使用 EditLabel()成员函数开始编辑，在编辑开始、被取消或完成时，树控件都发送通知。编辑完成后，若有需要，开发人员将负责更新项标签。

开始编辑标签时，树控件发送 TVN_BEGINLABELEDIT 通知消息。通过处理这条消息，可以允许对一些标签进行编辑和禁止对其他的标签进行编辑。返回 0，允许编辑；返回非 0 值，禁止编辑。

取消或完成标签编辑时，树控件发送 TVN_ENDLABELEDIT 通知消息。lParam 参数是 NMTVDISPINFO 结构的地址。item 成员是 TVITEM 结构，该结构标识项，并且包含编辑过的文本。有需要时，由开发人员负责更新项的标签，大多是在验证完编辑的字符串后进行更新。如果取消了编辑，TV_ITEM 的 pszText 成员为 0。

标签编辑过程中，通常在对 TVN_BEGINLABELEDIT 通知消息的响应过程中，可以使用 GetEditControl()成员函数获取指向用于标签编辑的编辑控件的指针。可以调用编辑控件的 SetLimitText() 成员函数，限制用户可输入的文本长度；或创建编辑控件的子类以截取和放弃无效字符。注意，只有发送 TVN_BEGINLABELEDIT 通知消息后才显示编辑控件。

5.8.6　树控件项的状态

树控件（CTreeCtrl）中的每项都有当前状态。例如，某项可以处在被选定状态、禁用状态、展开状态等。多数情况下，树控件自动设置项的状态以反映用户的操作，比如选择某项。但是，也可以使用 SetItemState()成员函数设置项的状态，并且可以利用 GetItemState()成员函数检索项的当前状态。

项的当前状态由 nState 参数指定。树控件可能更改项的状态以反映用户的操作，如选择某项或者为某项设置焦点。另外，应用程序也可能更改项的状态以禁用或隐藏某项，或者指定覆盖图像或状态图像。

指定或更改项的状态时，nStateMask 参数指定要设置的状态位，并且 nState 参数包含这些状态位的新值。例如，将 CTreeCtrl 对象（m_treeCtrl）中父项（由 hParentItem 指定）的当前状态改为 TVIS_EXPANDPARTIAL：

```
TVITEM curItem;
HTREEITEM hParentItem;

hParentItem= m_treeCtrl.GetSelectedItem();

//modify the parent item to keep the '+' sign
curItem.mask= TVIF_HANDLE;
curItem.hItem= hParentItem;
m_treeCtrl.GetItem(&curItem);

curItem.mask= TVIF_STATE;
curItem.state= TVIS_EXPANDPARTIAL;
curItem.stateMask= TVIS_EXPANDPARTIAL;
m_treeCtrl.SetItem(&curItem);
```

另一个更改状态的示例是设置项的覆盖图像。为此，nStateMask 必须包含 TVIS_OVERLAYMASK 值，而且 nState 必须包含覆盖图像从 1 开始的索引（已使用 INDEXTOOVERLAYMASK 宏将此覆盖图

像向左移动了 8 位）。索引可以是 0，这表示没有覆盖图像。覆盖图像必须已由 CImageList::SetOverlay-Image()函数的前一个调用添加到了树控件的覆盖图像列表中。该函数指定添加的图像从 1 开始的索引，这是 INDEXTOOVERLAYMASK 宏使用的索引。一个树控件最多可以有 4 个覆盖图像。

若要设置项的状态图像，nStateMask 必须包含 TVIS_STATEIMAGEMASK 值，而且 nState 必须包含状态图像从 1 开始的索引（已使用 INDEXTOSTATEIMAGEMASK 宏将该图像向左移动了 12 位）。索引可以是 0，这表示没有状态图像。

5.8.7 树控件图像列表

树控件（CTreeCtrl）中的每项都可以有一对与之关联的位图化图像，这些图像显示在项标签的左侧。如果选择了该项，就会显示一个图像；否则就会显示另一个图像。例如，某项可能在选定时显示一个打开的文件夹，而在未选定时显示关闭的文件夹。

若要使用项图像，必须通过构造 CImageList 对象和使用 CImageList::Create()函数创建相关图像列表。然后，将想要的位图添加到创建的列表中，并用 SetImageList()成员函数使列表与树控件相关联。默认情况下，所有的项在选定状态和未选定状态时，都显示图像列表中的第一个图像。使用 InsertItem()成员函数向树控件添加项时，可以指定选定图像和未选定图像的索引，以更改该项的默认行为。添加完项后，利用 SetItemImage()成员函数可以更改这两个索引。

树控件图像列表也可以包括覆盖图像，这些覆盖图像用来叠加在项图像上。树控件项状态的第 8 位到第 11 位中的非零值指定覆盖图像从 1 开始的索引（0 表示没有覆盖图像）。因为使用的从 1 开始的索引是 4 位的，所以覆盖图像必须在图像列表的前 15 位。

指定了状态图像后，树控件就在每项的图标左侧为状态图像保留空间。应用程序可以用状态图像（如选定和未选定的复选框）指示应用程序定义的项状态。项状态中第 12 位到第 15 位中的非零值指定状态图像从 1 开始的索引（0 表示没有状态图像）。

通过指定 I_IMAGECALLBACK 值而不是图像的索引，可以将对选定图像和未选定图像的指定延迟到要重绘项的时候。I_IMAGECALLBACK 通过发送 TVN_GETDISPINFO 通知消息指示树控件查询应用程序的索引。

GetImageList()成员函数检索树控件图像列表的句柄，如果需要向列表中添加更多的图像，该函数很有帮助。

5.8.8 树控件项选择

从选择某项转向选择另一项时，树控件（CTreeCtrl）发送 TVN_SELCHANGING 和 TVN_SELCHANGED 通知消息。

两个通知消息都有一个值指定是单击鼠标左键改变了选择，还是按键盘上的键改变了选择。这个条通知消息还包含有关正失去选择的项和正得到选择的项的信息。可以利用这些信息，设置那些依赖于项选择状态的项属性。作为对 TVN_SELCHANGING 通知消息的响应，返回 True 禁止选择另一项；返回 False 则允许选择另一项。

应用程序可以通过调用 SelectItem()成员函数来改变选择。

5.8.9 树控件拖放操作

树控件（CTreeCtrl）在用户开始拖动项时发送通知。当用户用鼠标左键开始拖动项时，树控件发

送 TVN_BEGINDRAG 通知消息；用鼠标右键开始拖动项时，则发送 TVN_BEGINRDRAG 通知消息。通过指定树控件 TVS_DISABLEDRAGDROP 样式，可以禁止树控件发送这些通知。

通过调用 CreateDragImage()成员函数获取在拖动操作过程中显示的图像。树控件根据正被拖动的项的标签，创建一个表现拖动操作的位图；然后创建图像列表，将该位图添加进去，并返回指向 CImageList 对象的指针。

必须提供实际执行拖动项操作的代码，这通常涉及使用图像列表中的拖动功能，并处理拖动操作开始后发送的 WM_MOUSEMOVE 和 WM_LBUTTONUP（或 WM_RBUTTONUP）消息。

若要对树控件中的某项执行拖放操作，则需要知道鼠标光标何时在该项上。为此可以调用 HitTest()成员函数来指定一个点和整数，或者指定包含鼠标光标当前坐标的 TVHITTESTINFO 结构的地址。调用的函数返回时，指定的整数或结构中就有一个指示鼠标光标相对于树控件的位置的标志。光标如果在树控件中的某项上，则结构中还包含该项的句柄。

通过调用 SetItem()成员函数将状态设置为 TVIS_DROPHILITED 值，可以指明作为拖放操作目标的项。按照用于指示拖放操作目标的样式，具有该状态的项将被拖动。

5.8.10　树控件项信息

树控件（CTreeCtrl）有许多检索项的信息的成员函数。GetItem()成员函数检索部分或全部与项关联的数据，这些数据可能包括项的文本、状态、图像、子项计数和应用程序定义的 32 位数据值。SetItem()函数，能够设置部分或全部与项关联的数据。

GetItemState()、GetItemText()、GetItemData()和 GetItemImage()成员函数检索项的每个属性，这些函数中的每一个都有一个对应的 Set 函数来设置项属性。

GetNextItem()成员函数检索与当前项有指定关系的树控件项，该函数能够检索项的父项、前一个或下一个可见项、第一个子项等。还有一些用于在树中遍历的函数：GetRootItem()、GetFirstVisibleItem()、GetNextVisibleItem()、GetPrevVisibleItem()、GetChildItem()、GetNextSiblingItem()、GetPrevSiblingItem()、GetParentItem()、GetSelectedItem()和 GetDropHilightItem()。

GetItemRect()成员函数检索树控件项的边框。GetCount()和 GetVisibleCount()成员函数分别检索树控件中的项计数和当前可见项的计数。可以调用 EnsureVisible()成员函数确保某项可见。

5.8.11　树控件通知消息

树控件（CTreeCtrl）发送如表 5.15 所示的通知消息作为 WM_NOTIFY 消息。

表 5.15　　树控件的通知消息

通 知 消 息	说　　明
TVN_BEGINDRAG	通知拖放操作开始
TVN_BEGINLABELEDIT	通知开始编辑标签
TVN_BEGINRDRAG	通知利用鼠标右键开始拖放操作
TVN_DELETEITEM	通知删除特定项
TVN_ENDLABELEDIT	通知标签编辑结束
TVN_GETDISPINFO	请求树控件显示某项所需的信息
TVN_ITEMEXPANDED	通知父项的子项列表已展开或已折叠
TVN_ITEMEXPANDING	通知即将展开或折叠父项的子项列表
TVN_KEYDOWN	通知键盘事件

续表

通 知 消 息	说　　明
TVN_SELCHANGED	通知已从选择某项转向选择另一项
TVN_SELCHANGING	通知即将从选择某项转向选择另一项
TVN_SETDISPINFO	为某项维护的信息更新通知

5.9　Visual C++ 2010 DBgrid 控件简介与开发

DBgrid 控件是企业级数据库前端应用程序开发中最强大的表格控件。DBGrid 是以二维表方式显示、编辑数据表的控件，是 Visual Studio 开发中最常用的数据编辑控件之一。DBGrid 属于扩展控件，需要通过在部件对话框中选中“Microsoft Data Bound Grid Control5.0（SP3）”复选框来添加到工具箱中。

其主要属性有 DataSource，用于选择数据源控件。

自定义属性介绍如下。

- General 页：设置 DBGrid 表格的标题等行属性。
- Keyboard 页：设置键盘属性。
- Columns 页：设置表格列属性。
- Layout 页：设置表格布局。
- Color 页：设置表格颜色。
- Font 页：设置表格字体。
- Split 页：设置表格分隔线。

在 Visual C++中使用 DBGrid 控件也需要一个类似于 Data 的控件，即 RemoteData 控件。

下面举两个示例程序来演示 DBGrid 控件的使用方法，这两个示例的项目名都为 DBGrid。

5.9.1　示例程序 1

按如下步骤创建示例程序 1。

（1）运行 AppWizard 生成项目 DBGrid。除选中“Single Document”单选按钮和将 CDBGridView 的基类改为 FormView 以外，其他均按默认设置处理。

（2）在项目 DBGrid 中添加 DBGrid 控件和 Microsoft RemoteData 控件。在 Developer Studio 中选择“Project”→“Add To Project”→“Componets And Controls”命令，选择 Registered ActiveX Controls，然后选择 DBGrid Control 和 Microsoft RemoteData Control。Developer Studio 将产生控件类 CMSDgridCtrl 和 CRdc，并把相应的文件添加到项目中。

（3）在 IDD_DBGRID_FORM 模板中加入控件。利用控件工具箱加入 RemoteData 和 DBGrid 控件。

对控件 RemoteData 进行编辑。选中该控件并右击，选择“Properties”命令，弹出 Microsoft RemoteData Control 2.0 Properties 对话框，选择“General”选项卡，修改资源 ID 为 IDC_RDCSTUDENT，Caption 为 Student。选择“Control”选项卡，设置 DataSource 为 Student（即前面 ODBC 数据源管理程序安装的数据源），在 SQL 中输入语句 SELECT ＊FROM student。

对控件 DBGrid 进行编辑。选择该控件并右击，选择“Properties”命令，弹出 DBGrid Control Properties 对话框。选择“General”选项卡，修改资源 ID 为 IDC_DBGSTUDENT，Caption 为 Student 或不设置。单击“All”选项卡，DataMode 设置为 0 －Bound，DataSource 设置为 IDC_RDCSTUDENT。

（4）编译并测试程序。DBGrid 在程序中正确地工作了。

示例程序 1 可能过于简单，它的目的只是演示 DBGrid 控件在 Visual Studio 2010 中使用的方法。下面对它略作改进。

（1）修改设计时 DBGrid 控件的列数。DBGrid 控件在设计时的列数默认为 2。选择 DBGrid 控件并右击，选择“DBGrid Control Object”→“Edit”命令，结果看上去没有什么变化，其实不然；再选择 DBGrid 控件并右击，可以发现此时的菜单不同了，选择“Insert”命令，此时 DBGrid 控件的列数由原来默认的 2 变成了 3。重复操作，直到 DBGrid 控件的列数满足要求为止。

（2）修改 DBGrid 控件的列标题。DBGrid 控件的列标题默认为数据集的字段名，可以改为想要的列标题。

步骤如下。

① 修改 DBGrid 控件的列标题。选择该控件并右击，选择“Properties”命令，弹出 DBGrid Control Properties 对话框。选择“Columns”标签，设置 Column 为 Column0，Caption 为“编号”，DataField 为 StudentID，在其他项中选择或输入所需的信息。重复本步骤，直至修改完所有的列标题为止。

② 关闭 DBGrid Control Properties 对话框，可以看到修改的列标题已经在 DBGrid 控件上显示了。

（3）在程序运行时改变 DBGrid 控件的尺寸。在视图尺寸大小改变时，DBGrid 控件的尺寸却没有改变，需要添加消息处理函数 OnSize()。

① 使用 Add Variable Wizard 为类 CDBGridView 加入成员变量。在 MFC Variable Wizard 中选择 IDC_DBGSTUDENT，添加变量 m_dbgstudent，类型为 Control。

② 使用 MFC Variable Wizard 映射视图类 CDBGridView 的 WM_SIZE 函数。编辑函数如下：

```
void CDBGridView::OnSize(UINT nType, int cx, int cy)
{
    CFormView::OnSize(nType, cx, cy);
    CRect rect;
    GetClientRect(rect);
    ::SetWindowPos(m_dbgstudent,HWND_TOP,0,0,rect.right－rect.left, rect.bottom－rect.top,SWP_SHOWWINDOW);
}
```

示例程序 1 演示了 DBGrid 控件在类 CFormView 的派生类中的使用方法，在对话框中使用 DBGrid 控件的方法与此完全相同。

本示例是 DBGrid 控件最简单的使用，因为它们都是把控件添加到控件容器中，然后再设定其属性，所以 RemoteData 和 DBGrid 控件可以像 Button、Edit 控件一样，单击几下就可以了。下面再介绍 DBGrid 控件在类 Cview 或其派生类等非控件容器中的使用方法。这时就不能用上面的方法向视图类中添加控件了，也就不能在设计时设定其属性，而必须在运行时使用 DBGrid 和 RemoteData 控件的 Create()函数创建对象，然后再设置属性，具体方法请看示例程序 2。

5.9.2　示例程序 2

按如下步骤创建示例程序 2。

（1）运行 AppWizard 生成项目 DBGrid。选中“Multiple Documents”复选框，CDBGridView 的基类仍为 CView，其他仍然按默认设置处理。

（2）在项目 DBGrid 中添加 DBGrid 控件和 Microsoft RemoteData 控件，与示例程序 1 相同。

（3）在 CDBGridView 类中加入两个数据成员，一个是类 CRdc 变量 m_rdcstudent，另一个是类 CMsDgridCtrl 变量 m_dbgstudent。

（4）加入这两个控件的子窗口 ID 常数。从 Developer Studio 中选择“View”→“Resource Sysbols”命令，然后加入 ID 符号 ID_RDCSTUDENT 和 ID_DBGSTUDENT。

（5）用 MFC Add Variable Wizard 映射视图的 WM_CREATE 和 WM_SIZE 函数。编辑 DBGidView.cpp 中的消息处理函数如下：

```
int CDbgridView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
            return -1;
    RECT r={0,0,100,30};
      //    create the RDC
    m_rdcstudent.Create(_T("Student"),        WS_CHILD,r,this,IDC_RDCSTUDENT);
    m_rdcstudent.SetDataSourceName(_T("Student"));
    m_rdcstudent.SetSql(_T("Select * from student,section"));
    m_rdcstudent.SetUserName(_T("admin"));
    m_rdcstudent.SetPassword(_T(""));
    m_rdcstudent.SetCursorDriver(2);

    // create the DBGrid control & connect it to IDC_RDCSTUDENT
    m_dbgstudent.Create(_T(""),WS_VISIBLE|WS_CHILD,r,this,IDC_DBGSTUDENT);
    LPUNKNOWN pCursor=m_rdcstudent.GetDSCCursor();
    ASSERT(pCursor!=NULL);
    m_dbgstudent.SetDataSource(pCursor); //bind the grid
    m_dbgstudent.BindProperty(0x9, &m_rdcstudent);
    return 0;
}

void CDbgridView::OnSize(UINT nType, int cx, int cy)
{
        CView::OnSize(nType, cx, cy);
        // TODO: Add your message handler code here
        CRect rect;
    GetClientRect(rect);
    ::SetWindowPos(m_dbgstudent,HWND_TOP,0,0,rect.right-rect.left, rect.bottom-rect.top,SWP_SHOWWINDOW);
}
```

（6）编译并测试程序。

5.10 Visual C++ 2010 Rebar 控件简介与开发

Rebar 控件用作子窗口的容器，这些子窗口（通常是其他控件）被分配给 Rebar 控件带区。Rebar 控件可以包含一个或多个带区，每个带区都是拖动条、位图、标签文本和子窗口的任意组合。但是，每个带区所包含的子窗口不能多于一个。

图 5.5 显示了有两个带区的 Rebar 控件，一个带区包含拖动条、文本标签（“地址”）和组合框子窗口；另一个带区包含拖动条、文本标签和平面工具栏（用子窗口实现）。

图 5.5 两个带区的 Rebar 控件

MFC 提供了两个用于创建 Rebar 的类：CReBar 和 CReBarCtrl（包装了 Windows 公共控件 API）。CReBar 提供 Rebar 公共控件的所有功能，并为开发人员处理许多必需的公共控件设置和结构。

CReBarCtrl 是一个 Win32 Rebar 控件的包装类，因此，如果不打算将 Rebar 集成到 MFC 结构中，实现起来可能更容易。如果计划使用 CReBarCtrl 并将 Rebar 集成到 MFC 结构中，必须特别注意使 Rebar 控件操作与 MFC 通信。此通信并不困难，但是当使用 CReBar 时，这是不必要的附加工作。

Visual C++提供两种使用 Rebar 公共控件的方法。

（1）使用 CReBar 创建 Rebar，然后调用 CReBar::GetReBarCtrl()获得对 CReBarCtrl()成员函数的访问。

CReBar::GetReBarCtrl()是一种转换此 Rebar 对象 this 指针的内联成员函数，这意味着在运行时，此函数调用无系统开销。

（2）使用 CReBarCtrl 的构造函数创建此 Rebar。

以上任何一种方法都使开发人员可以访问 Rebar 控件的成员函数。当调用 CReBar::GetReBarCtrl()时，它返回对 CReBarCtrl 对象的引用，以便开发人员可以使用两组成员函数中的任意一组。

CReBarCtrl 对象应在父对象可见之前创建，这使出现绘图问题的可能性减到最小。

例如，（在框架窗口对象中使用的）Rebar 控件通常用作工具栏（Tool Bar）控件的父窗口，因此 Rebar 控件的父级就是框架窗口对象。因为框架窗口对象是父级，所以（父级的）OnCreate()成员函数是创建 Rebar 控件的上佳位置。

若要使用 CReBarCtrl 对象，一般应遵循以下步骤：

（1）构造 CReBarCtrl 对象。

（2）调用 Create()创建 Windows Rebar 公共控件并将其连接到 CReBarCtrl 对象，并指定任何所需的样式。

（3）通过调用 CBitmap::LoadBitmap()，加载将用作 Rebar 控件对象背景的位图。

（4）创建并初始化 Rebar 控件对象将要包含的任何子窗口对象（工具栏、对话框控件等）。

（5）用将要插入的带区的必需信息初始化 REBARBANDINFO 结构。

（6）调用 InsertBand()以将现有子窗口（如 m_wndReToolBar）插入新的 Rebar 控件中。

Rebar 控件的主要用途是作为子窗口、通用对话框控件、菜单、工具栏等的容器。每个 Rebar 带区都可以包含拖动条、位图、文本标签和子窗口的任意组合。

CReBarCtrl 类有许多可用来检索和操作特定 Rebar 带区信息的成员函数。

- GetBandCount()　检索 Rebar 控件中的当前带区数。
- GetBandInfo()　用指定带区中的信息初始化 REBARBANDINFO 结构，有对应的 SetBandInfo 成员函数()。
- GetRect()　检索指定带区的边框。
- GetRowCount ()　检索 Rebar 控件中的带区个数。
- IDToIndex()　检索指定带区的索引。
- GetBandBorders()　检索带区的边框。

除了操作，还提供了几个使开发人员得以操作特定 Rebar 带区的成员函数。

InsertBand()和 DeleteBand()添加和移除 Rebar 带区。MinimizeBand()和 MaximizeBand()影响特定 Rebar 带区的当前大小。MoveBand()更改特定 Rebar 带区的索引。ShowBand()对用户显示或隐藏 Rebar 带区。

下例说明如何将工具栏带区（m_wndToolBar）添加到现有 Rebar 控件（m_wndReBar）中。通过

初始化 rbi 结构然后调用 InsertBand()成员函数对此带区进行了描述。

代码如下：

```
//load bitmap for toolbar background
m_bmap.LoadBitmap(IDB_BITMAP);
//create a toolbar band
m_wndToolBar.Create(this, TBSTYLE_TRANSPARENT | TBSTYLE_FLAT);
m_wndToolBar.LoadToolBar(IDR_MAINFRAME);

REBARBANDINFO rbi;
rbi.cbSize= sizeof(REBARBANDINFO);
rbi.fMask= RBBIM_BACKGROUND | RBBIM_CHILD | RBBIM_CHILDSIZE | RBBIM_STYLE | RBBIM_TEXT;
rbi.fStyle= RBBS_GRIPPERALWAYS;
rbi.cxMinChild= 300;
rbi.cyMinChild= 30;
rbi.lpText= "Band #1";
rbi.cch= 7;
rbi.cx= 300;
rbi.hbmBack= (HBITMAP)m_bmap;
rbi.hwndChild= (HWND)m_wndToolBar;
m_wndReBar.InsertBand(-1, &rbi);
```

5.10.1 在 Rebar 控件中使用图像列表

每个 Rebar 带区都可以在其他对象中包含来自相关图像列表的图像。以下过程详细描述了在 Rebar 带区中显示图像所需的步骤。

（1）通过调用 SetImageList()并传递指向现有图像列表的指针，将图像列表附加到 Rebar 控件对象。

（2）修改 REBARBANDINFO 结构，将图像分配到 Rebar 带区：

- 将 fMask 成员设置为 RBBIM_IMAGE，并根据需要使用按位“或”运算符包括其他标志。
- 将 iImage 成员设置为要显示的图像的图像列表索引。

（3）用必需的信息初始化任何剩余的数据成员，如包含的子窗口的大小、文本和句柄。

（4）通过调用 CReBarCtrl::InsertBand()插入新带区（包含此图像），并传递 REBARBANDINFO 结构。

下例假定一个包含两个图像的现有图像列表对象附加到了 Rebar 控件对象（m_wndReBar）。通过调用 InsertBand()添加一个新的 Rebar 带区（由 rbi 定义），此带区包含第一个图像。

代码如下：

```
REBARBANDINFO rbi;
rbi.cbSize= sizeof(REBARBANDINFO);
rbi.fMask= RBBIM_BACKGROUND|RBBIM_CHILD|RBBIM_IMAGE|RBBIM_CHILDSIZE|RBBIM_STYLE | RBBIM_
TEXT;
rbi.fStyle= RBBS_GRIPPERALWAYS;
rbi.cxMinChild= 200;
rbi.cyMinChild= 50;
rbi.lpText= "Band #1";
rbi.cch= 7;
rbi.cx= 300;
rbi.hbmBack= (HBITMAP)m_bmap3;
rbi.iImage= 0;
```

```
rbi.hwndChild= (HWND)m_wndToolBar;
BOOL bRes= m_wndReBar.InsertBand(-1, &rbi);
```

5.10.2　在 Rebar 控件中使用对话栏

正如在 Rebar 控件和带区中提到的，每个带区只能包含一个子窗口（或控件）。如果希望每个带区拥有的子窗口不止一个，可以创建一个包含多个控件的对话栏资源，然后将（包含此对话栏的）Rebar 带区添加到 Rebar 控件。

通常，如果希望对话栏带区显示为透明的，则为对话栏对象设置 WM_EX_TRANSPARENT 扩展样式。但是，由于 WM_EX_TRANSPARENT 在正确绘制对话栏背景方面存在一些问题，因此开发人员将需要做一点额外的工作来实现所需的效果。

以下过程详细描述了实现透明但不使用 WM_EX_TRANSPARENT 扩展样式所需的步骤。

（1）使用添加类对话框，添加实现对话栏对象的新类（例如，CMyDlgBar）。

（2）添加 WM_ERASEBKGND 消息的处理程序。

（3）在新处理程序中，修改现有代码以匹配下例。

代码如下：

```
BOOL CMyDlgBar::OnEraseBkgnd( CDC* pDC )
{
    CWnd* pParent = GetParent();
    ASSERT_VALID(pParent);
    CPoint pt(0, 0);
    MapWindowPoints(pParent, &pt, 1);
    pt = pDC->OffsetWindowOrg(pt.x, pt.y);
    LRESULT lResult = pParent->SendMessage(WM_ERASEBKGND, (WPARAM)pDC->m_hDC, 0L);
    pDC->SetWindowOrg(pt.x, pt.y);
    return lResult;
}
```

（4）添加 WM_MOVE 消息的处理程序。

（5）在新处理程序中，修改现有代码以匹配下例。

代码如下：

```
BOOL CMyDlgBar::OnMove( int cx, int cy )
{
    Invalidate();
}
```

新处理程序通过将 WM_ERASEBKGND 消息转发给父窗口，并在每次对话栏对象移动时都强制重画来模拟对话栏的透明。

5.10.3　处理 Rebar 控件中的通知消息

在 Rebar 控件的父类中，创建一个 OnChildNotify()处理函数，该函数对要处理的任何 Rebar 控件（CReBarCtrl）通知消息都有一个 switch 语句。当用户在 Rebar 控件上执行拖移对象，更改 Rebar 带区的布局，从 Rebar 控件删除带区等操作时，通知消息将被发送到父窗口。

下面介绍 Rebar 控件中的通知消息。

- RBN_AUTOSIZE: 当 Rebar 自动调整大小时，由（用 RBS_AUTOSIZE 样式创建的）Rebar 控件发送。

- RBN_BEGINDRAG: 当用户开始拖动带区时，由 Rebar 控件发送。
- RBN_CHILDSIZE: 当调整带区的子窗口大小时，由 Rebar 控件发送。
- RBN_DELETEDBAND: 在带区已被删除后，由 Rebar 控件发送。
- RBN_DELETINGBAND: 当带区即将被删除时，由 Rebar 控件发送。
- RBN_ENDDRAG: 当用户停止拖动带区时，由 Rebar 控件发送。

RBN_GETOBJECT：当对象被拖动到此控件中的带区上时，由（用 RBS_REGISTERDROP 样式创建的）Rebar 控件发送。

- RBN_HEIGHTCHANGE: 当其高度已更改时，由 Rebar 控件发送。
- RBN_LAYOUTCHANGED: 当用户更改此控件的带区布局时，由 Rebar 控件发送。

5.11 Visual C++ 2010 Timer 控件简介与开发

Timer 控件可用于完成给定时间间隔执行的任务，定时控件的主要属性是 Interval，用于确定定时器通知应用程序的频率。如果 Interval 设置为 1000，则定时器控件每秒发送一个 Timer 事件。

Timer 控件响应时间的流逝，它们独立于用户，编程后可用来在一定的时间间隔执行操作。此控件的一个一般用途是检查系统时钟，判断是否该执行某项任务。对于其他后台处理程序，Timer 控件也非常有用。

每个 Timer 控件都有 Interval 属性，指定定时器事件之间的毫秒数。除非禁止此属性，否则定时器在大致相等的时间间隔不断接受事件（称作定时器事件会更贴切）。

在为 Timer 控件编程时，应考虑对 Interval 属性的几条限制。

- 如果应用程序或其他应用程序正在进行对系统要求很高的操作，例如长循环、高强度的计算或者正在访问驱动器、网络或端口，则应用程序定时器事件的间隔可能比 Interval 属性指定的间隔长。
- 间隔的取值可在 0～64 767 之间（包括这两个数值），这说明即使是最长的间隔也不比一分钟长多少（大约 64.8s）。
- 间隔并不一定十分准确。要保证间隔准确，应在需要时才让定时器检查系统时钟，而不在内部追踪积聚的时间。
- 系统每秒生成 8 个时钟信号，所以即使用毫秒衡量 Interval 属性，间隔实际的精确度也不会超过 1/18s。

每个 Timer 控件必须与窗体关联，因此要创建定时器应用程序就必须至少创建一个窗体（如果不需要窗体完成其他操作，就不必使窗体可见）。

在对话框或者窗体上放置 Timer 控件的方法与绘制其他控件的方法相同：单击工具箱中的定时器控件并将其拖动到窗体或者对话框上。

Timer 控件只在设计时显示在对话框上，所以可以选定这个控件，查看其属性并编写事件过程。运行时，定时器不可见，所以其位置和大小也无关紧要。

初始化 Timer 控件时，应特别注意，Timer 控件有两个关键属性，如表 5.16 所示。

表 5.16　　Timer 控件有两个关键属性

属　性	设 置 值
Enabled	若希望窗体一旦加载定时器就开始工作，应将此属性设置为 True；否则，保持此属性为 False。有时可能选择由外部事件（例如单击命令按钮）启动定时器操作
Interval	定时器事件之间的毫秒数

（1）定时器的 Enabled 属性不同于其他对象的 Enabled 属性。对于大多数对象，Enabled 属性决定对象是否响应用户触发的事件；对于 Timer 控件，将 Enabled 设置为 False 时，就会暂停定时器操作。
（2）Timer 事件是周期性的。Interval 属性主要是决定“多少次”，而不是“多久”。间隔的长度取决于所需精确度。因为存在某些内部的错误可能性，所以应将间隔设置为精确度的一半。
（3）定时器事件生成越频繁，响应事件所使用的处理器事件就越多，这将降低系统综合性能。除非有必要，否则不要设置过小的间隔。

下面将通过一个具体的示例来讲解 Timer 控件的使用方法。

在 ClassWizard 中响应 ID 为~Dlg 中的 WM_TIMER 消息。

使用 SetTimer(nIDEvent,time,NULL)建立一个定时器，关闭定时器用 KillTimer(nIDEvent)函数。

可以响应 ON_WM_TIMER 消息来响应一个定时器完成一次计时后的程序。

响应方式如下：

```
void    CTimeDlg::OnTimer(UINT    nIDEvent)
{
      if(nIDEvent==1000)//间隔为 5 秒
      {
          //处理事件
      }
      elseif(nIDEvent==1001)//间隔为 10 秒
      {
          //处理事件
      }
      CDialog::OnTimer(nIDEvent);
}
```

以下是给出一个串口通信定时检查接收数据的部分代码：

```
void CMyDlg::OnOpenCom()
{
    // TODO: Add your control notification handler code here
    if( f_open_com==true )
    {
        f_open_com = false;
        GetDlgItem(IDC_OPEN_COM)->SetWindowText("打开通信端口");
        CloseHandle(hComm);
        KillTimer(1000);                                    // 关闭定时器
        return ;
    }
    SetTimer(1000, 1000, NULL);  ///nIDEvent==1000,time=5000ms
    const char *ComNo;
    DCB dcb;
    string temp("COM1");
    ComNo = temp.c_str();
    hComm = CreateFile( ComNo , GENERIC_READ|GENERIC_WRITE , 0 , NULL , OPEN_EXISTING , 0 , 0);
    if( hComm==INVALID_HANDLE_VALUE )          // 如果端口未打开
    {
        MessageBox("打开通信端口出错！ " , "Comm Error" , MB_OK);
        return ;
    }
    // 将 dcb 地址传入，以取得通信参数
   GetCommState(hComm,&dcb);                              // 得知目前通信状态
```

```
    dcb.BaudRate = CBR_9600;
    dcb.ByteSize = 8;                                    // 字节为 8
    dcb.Parity = NOPARITY;                               // Parity 为 None
    dcb.StopBits = ONESTOPBIT;                           // 1 个停止位
    if( !SetCommState( hComm , &dcb)){
        MessageBox("通信端口设置出错！ " , "Set Error" , MB_OK );
        CloseHandle(hComm);
        return;
    }
    GetDlgItem(IDC_OPEN_COM)->SetWindowText("关闭通信端口");
    f_open_com = true;
}

void CMyDlg::OnTimer(UINT nIDEvent)
{
    // TODO: Add your message handler code here and/or call default
    char inbuff[1024];
    DWORD nBytesRead , dwError;
    COMSTAT cs;
    //  取得状态
    ClearCommError( hComm , &dwError , &cs);
    //  数据是否大于所准备的缓冲区
    if( cs.cbInQue > sizeof(inbuff) )
    {
        PurgeComm(hComm , PURGE_RXCLEAR );                    // 清除通信端口数据
        return ;
    }
    ReadFile(hComm , inbuff , cs.cbInQue , &nBytesRead , NULL );          //接收通信端口的数据
    inbuff[cs.cbInQue] = '\0';
    MessageBox("打开通信端口出错！ " , "Comm Error" , MB_OK);
    m_Receive.Format("%s",inbuff);
    UpdateData(false);
    CDialog::OnTimer(nIDEvent);
}
```

5.12 Visual C++ 2010 Tab 控件简介与开发

选项卡控件（Tab Control）类似于笔记本中的分隔卡或文件柜中带标签的文件夹。用选项卡控件（由类 CTabCtrl 表示）以建议各页之间的对等或逻辑关系的格式，向用户显示包含多页的信息或控件，一次显示一页。一个选项卡控件上提供了多个选项卡，而每个选项卡又可以包含它们自己的控件集合。可以将选项卡控件用于应用程序客户区窗口。

可以将选项卡控件（CTabCtrl）的各个项定义为所有者描述的项。在创建选项卡控件（CTabCtrl）后，需要多少个选项卡，就可以添加多少。

添加选项卡项的具体方法如下：

（1）准备 TCITEM 结构；

（2）调用 CTabCtrl::InsertItem()以传递该结构；

（3）对其他选项卡项重复步骤（1）和步骤（2）。

5.12.1　选项卡和选项卡控件属性

对组成选项卡控件（CTabCtrl）的选项卡的外观和行为可以有相当多的控制，每个选项卡可以有与之关联的标签、图标、项状态和应用程序定义的 32 位数值。对每个选项卡，可以显示图标、标签或两者都显示。

另外，每个选项卡项可以有 3 种可能的状态：已按下、非按下或突出显示。该状态仅能通过修改现有选项卡项来设置，若要修改现有选项卡项，请调用 GetItem()检索它，修改 TCITEM 结构（特别是 dwState 和 dwStateMask 数据成员），然后通过调用 SetItem()返回修改过的 TCITEM 结构。若要清除 CTabCtrl 对象中所有选项卡项的项状态，请调用 DeselectAll()，该函数可以重新设置所有选项卡项或所有项的状态（当前选定的项除外）。

下面的代码清除所有选项卡项的状态，然后修改第三项的状态：

```
//modify the third item to be highlighted
TCITEM curItem;

m_tabCtrl.DeselectAll(FALSE); //reset all tab items
curItem.mask= TCIF_STATE;
m_tabCtrl.GetItem(2, &curItem);
curItem.mask= TCIF_STATE;
curItem.dwState= TCIS_HIGHLIGHTED;
curItem.dwStateMask= TCIS_HIGHLIGHTED;
m_tabCtrl.SetItem(2, &curItem);
```

5.12.2　选项卡控件的使用方法

使用选项卡控件（CTabCtrl）的最容易的方法是用对话框编辑器将其添加到对话框模板资源中，也可以单独使用选项卡控件。MFC 为开发人员提供了 InitCommonControls()函数，主要任务如下：

- 创建选项卡控件；
- 向选项卡控件添加选项卡；
- 处理选项卡控件通知消息。

如果在父级视图或对话框类中嵌入选项卡控件对象，当父级销毁时该控件也被销毁。

5.12.3　创建选项卡控件的方法

创建选项卡控件的方式取决于是在对话框窗口中使用该控件还是在非对话框窗口中使用它。

1．在对话框窗口中使用 CTabCtrl

（1）在对话框编辑器中，将选项卡控件添加到对话框模板资源中，指定其控件 ID。

（2）用添加成员变量向导添加具有 Control 属性且类型为 CTabCtrl 的成员变量，可以用该成员调用 CTabCtrl 成员函数。

（3）在对话框类中为需要处理的任何选项卡控件通知消息映射处理函数。

（4）在 OnInitDialog()中，设置 CTabCtrl 的样式。

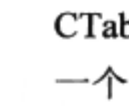
2. 在非对话框窗口中使用 CTabCtrl

（1）在视图类或窗口类中定义此控件。

（2）调用此控件的 Create()成员函数，可能在 OnInitialUpdate()中，也可能与父窗口的 OnCreate()处理函数一样早（如果正在创建此控件的子类）设置此控件的样式。

创建 CTabCtrl 对象后，可以设置或清除下列扩展样式。

- TCS_EX_FLATSEPARATORS: 选项卡控件将在选项卡项之间绘制分隔符。该扩展样式仅影响样式为 TCS_BUTTONS 和 TCS_FLATBUTTONS 的选项卡控件。默认情况下，用 TCS_FLATBUTTONS 样式创建选项卡控件时，将设置该扩展样式。
- TCS_EX_REGISTERDROP: 在将对象拖动到控件中的选项卡项上时，选项卡控件生成 TCN_GETOBJECT 通知消息以请求放置目标对象。



若要接收 TCN_GETOBJECT 通知，必须通过调用 AfxOleInit()初始化 OLE 库。

创建控件后，可以通过分别调用 GetExtendedStyle()和 SetExtendedStyle()成员函数来检索和设置这些样式。

例如，用下列代码行设置 TCS_EX_FLATSEPARATORS 样式：

```
DWORD dwExStyle= m_tabCtrl.GetExtendedStyle();
m_tabCtrl.SetExtendedStyle(dwExStyle | TCS_EX_FLATSEPARATORS);
```

用下列代码行从 CTabCtrl 对象中清除 TCS_EX_FLATSEPARATORS 样式：

```
DWORD dwExStyle= m_tabCtrl.GetExtendedStyle();
m_tabCtrl.SetExtendedStyle(dwExStyle & ~TCS_EX_FLATSEPARATORS);
```

这将移除显示在 CTabCtrl 对象之间的分隔符。

5.12.4 处理选项卡控件通知消息

当用户单击选项卡或按钮时，选项卡控件（CTabCtrl）向其父窗口发送通知消息。如果要进行某种响应，请处理这些消息，例如，当用户单击选项卡时，开发人员也许想在显示选项卡之前在页面上预设控件数据。

处理来自视图或对话框类中的选项卡控件的 WM_NOTIFY 消息。基于正在处理的通知消息，用“属性”面板创建带 switch 语句的 OnChildNotify()处理函数。有关选项卡控件可以发送到其父窗口的通知的列表。

5.12.5 CTabCtrl 类简介

通过前面的介绍，可以知道选项卡控件由类 CTabCtrl 表示，下面具体讲解 CTabCtrl 类的相关知识。

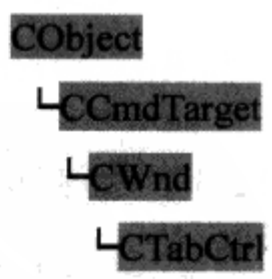


图 5.6 CTabCtrl 类的继承派生关系

CTabCtrl 类的继承派生关系如图 5.6 所示。

一个“标签控件”类似于一个笔记本中的分隔卡或一个文件柜中带标签的文件夹。通过使用标签控件，应用程序可以将一个窗口或对话框的相同区域定义为多个页面，每个页面包含一套信息或一组控件，当用户选择相应的标签时应用程序就会显示相应的信息或控件。一种特殊类型的标签控件把标签显示得像按钮一样。单击一个按钮将立即执行一条命令而不是显示一个页面。

CTabCtrl 类提供了 Windows 通用标签控件的性能，这个控件（也就是 CTabCtrl 类）只对运行在 Windows 95 和 Windows NT 3.51 或更新版本下的程序来说是可用的。

CTabCtrl 类成员构造函数如表 5.17 所示。

表 5.17　　CTabCtrl 类成员构造函数

构 造 函 数	功　能
CTabCtrl()	构造一个 CTabCtrl 对象
Create ()	创建一个标签控件并将它与一个 CTabCtrl 对象连接

表 5.18 给出了 CTabCtrl 类的属性成员函数及相应功能。

表 5.18　　CTabCtrl 类的属性类成员函数

属 性 函 数	功　能
GetImageList ()	获取与一个标签控件相关的图像列表
SetImageList ()	将一个图像列表分配给一个标签控件
GetItemCount ()	获取此标签控件中的标签的数目
GetItem ()	获取此标签控件中的某一个标签的信息
SetItemExtra ()	设置一个标签控件中的每一个标签为应用程序定义的数据所保留的字节数
GetItemRect ()	获取一个标签控件中的一个标签的边界矩形
GetCurSel ()	确定在一个标签控件中当前选择的标签
SetCurSel ()	在一个标签控件中选择一个标签
SetCurFocus ()	将焦点设置到一个标签控件中的指定标签上
SetItemSize ()	设置某个项的宽度和高度
SetPadding ()	设置一个标签控件中的每一个标签的图标和标签周围的空间（填料）
GetRowCount ()	获取一个标签控件中的标签的当前行数
GetToolTips ()	获取与一个标签控件相关联的工具提示控件的句柄
SetToolTips ()	将一个工具提示控件赋给一个标签控件
GetCurFocus ()	获取一个标签控件的具有当前焦点的标签
SetMinTabWidth ()	设置一个标签控件中的项的最小宽度
GetExtendedStyle ()	获取标签控件当前使用的扩展风格
SetExTendedStyle ()	设置一个标签控件的扩展风格
GetItemState ()	获取指定标签控件项的状态
SetItemState ()	设置指定标签控件项的状态

表 5.19 给出了 CTabCtrl 类的操作成员函数及功能。

表 5.19　　CTabCtrl 类的操作成员函数

操 作 函 数	功　能
InsertItem ()	在一个标签控件中插入一个新的标签
DeleteItem ()	从一个标签控件中删除一项
DeleteAllItems ()	从一个标签控件中删除所有的项
AdjustRect ()	根据一个给定的窗口矩形来估算一个标签控件的的显示区域，或根据一个给定的显示区域来估算与之对应的窗口矩形
RemoveImage ()	从一个标签控件的图像列表中删除一个图像
HitTest ()	确定哪一个标签（如果有）位于指定的屏幕位置
DeselectAll ()	重新设置一个标签控件中的项，清除任何被按下的项
HighlightItem ()	设置一个标签项的加亮状态

表 5.20 给出了 CTabCtrl 类的可重载函数及其功能。

表 5.20　　CTabCtrl 类的可重载函数

可重载函数	功　能
DrawItem ()	绘制一个标签控件的指定项

下面将具体讲解标签控件的创建方法。

1．标签控件的建立方法

```
CTabCtrl&  tabCtrl          建立标签控件对象结构
Create                      建立标签控件并绑定对象
```

标签控件 CTabCtrl::Create()的调用格式如下：

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

其中参数 dwStyle 用来确定标签控件的风格；rect 用来控制标签的大小和位置；pParentWnd 用来确定标签控件的父窗口句柄；nID 用来确定标签控件的标识符。

标签控件的风格可以是如表 5.21 所示的值的组合。

表 5.21　　标签控制的风格

风　格	意　义
TCS_BUTTONS	表示将标签外观定义成类似于按钮
TCS_FIXEDWIDTH	使所有标签具有相同的宽度
TCS_FOCUSNEVER	使特定标签永远不接收输入焦点
TCS_FOCUSONBUTTONDOWN	当标签被单击时接收输入焦点，其仅与 TCS_BUTTONS 合用
TCS_FORCEICONLEFT	强制图标在左侧，剩余部分使标签居中
TCS_FORCELABELLEFT	使图标和标签均左对齐
TCS_MULTILINE	允许标签控件显示多行标签
TCS_OWNERDRAWFIXED	允许父窗口自绘标签
TCS_RIGHTJUSTIFY	使标签右对齐
TCS_SHAREIMAGELISTS	当控件被撤销时标签控件的图像不被撤销
TCS_TOOLTIPS	允许标签控件存在工具提示控制
TCS_TABS	标签正常显示，为默认状态
TCS_SINGLELINE	将标签只显示在一行上，默认状态
TCS_RAGGEDRIGHT	不使标签自动填满控制区域，默认状态

同样，标签控件还可以使用窗口的一些控件风格，如表 5.22 所示。

表 5.22　　标签可使用的窗口的控制风格

窗口的控制风格	意　义
WS_CHILD	为标签控件创建子窗口，只能与 WS_POPUP 风格一起使用
WS_VISIBLE	建立一个初始可见的标签控件
WS_DISABLED	建立一个初始不可见的标签控件
WS_GROUP	建立标签控件群组的第一个控件
WS_TABSTOP	建立可用【Tab】键移动的标签控件

2．标签控件的属性函数

标签控件的属性函数包括取得与标签控件相关联的图像列表的 GetImageList()函数、设置标签控件的图像列表的 SetImageList()函数、取得标签控件中标签的总数的 GetItemCount()函数、取得标签控件中特定标签的相关信息的 GetItem()函数、设置标签的部分或全部属性的 SetItem()函数、检测当前被选中的标签的 GetCurSel()函数、将一个标签设置为选中状态的 SetCurSel()函数和取得具有当前输入焦点

的标签的 SetCurSel()函数等。

3．标签控件的操作方法

标签控件的操作方法包括在标签控件中插入一个标签的 InsertItem()函数、删除一个标签的 DeleteItem()函数、从标签控件中删除所有项目的 DeleteAllItems()函数、从标签控件中删除一个图像列表的 RemoveImage()函数和绘制标签控件中的特定一项的 DrawItem()函数等。

4．标签控件的数据结构

在使用标签控件时，必须使用的函数就是在标签控件中插入标签的 Insert Item()函数，其原型如下：

```
BOOL InsertItem(int nItem, TC_ITEM *   pTabCtrlItem);
```

该函数中的 TC_ITEM 为添加标签时所使用信息的数据结构，其数据成员的定义方法及含义如下：

```
typedef struct _TC_ITEM   {
UINT     mask;                  //确定结构成员的屏蔽或设置位
UINT     lpReserved1;           //保留未用
UINT     lpReserved2;           //保留未用
LPSTR     pszText;              //标签名称字符串
int     cchTextMax;             //标签名称字符串缓冲区大小
int     iImage;                 //标签控件的图像索引号
LPARAM     lParam;              //应用程序定义的相关的 32 位数据
}    TC_ITEM;
```

当单击标签控件中的标签时，标签控件就会向其父窗口发送相关的通知消息，通过处理这些通知消息，程序可以实现各种功能。

5．属性表和属性页的基本用法

在标签控件中，属性表对话框和属性页是必不可少的。

在 MFC 类库中，属性表对话框类 CPropertySheet 是由 CWnd 类派生而来的，而属性页类 CPropertyPage 是由 CDialog 类派生而来的，它们的用法基本相同。

（1）创建所有的属性页。创建属性页的方法与创建一般对话框资源的方法一样，利用对话框编辑器可以为每个属性页创建一个对话框模板，其区别在于，当利用类向导 ClassWizard 为属性页生成类时应选择属性页类 CPropertyPage 作为基类，而不是将一般的对话框类 CDialog 作为基类。

（2）创建属性表对话框，并将事先创建好的各属性页添加进去，两者的创建顺序可以互换，但在创建完之后将属性页添加到属性表对话框中这一步是必需的。

（3）显示属性表对话框。虽然属性表对话框类 CPropertySheet 不是由对话框类 CDialog 派生而来的，但两者的操作非常类似，调用 DoModal()函数就会显示一个模态属性表对话框，而调用 Create()函数就会显示一个非模态的属性表对话框。

（4）对数据交换的处理。与对话框类似，属性表对话框与对象之间的数据交换也是通过数据成员来实现的，只是属性表本身不带数据成员，而实际进行数据交换的是属性页中的数据成员。

（5）对向导对话框的处理。如果要显示一个向导对话框，在显示之前应首先调用 SetWizardMode()函数对向导对话框进行特殊处理，对于存在按钮的向导对话框，还应调用 SetWizardButtons()来对向导对话框的按钮功能进行定制，在用户操作结束时还应调用 SetFinishText()函数将“完成”按钮设置为有效状态。

下面通过一个具体的例子来详细地了解一个标签控件。

（1）首先创建一个 MFC 对话框框架，从工具箱中添加上一个标签控件到对话框资源上，根据需要修改一下属性，然后右击控件，为这个控件添加一个变量，将此控件与一个 CTabCtrl 类变量绑定在一

起，这里设为 m_tabctrl。

（2）创建两个新的对话框资源，对其属性进行如下修改。

```
Border：None          //边界为空，这样它就没了标题栏
Style：Child          //这样这个模板就可以当作另一个窗口的子窗口了
```

如果没有必要，其他属性选项就不必再修改了。

在其中添加控件的具体操作与普通对话框没有区别。完成后，从这两个对话框模板生成两个新的对话框类。

（3）在主对话框中为新添加的两个类增加两个变量：

```
CDialog1    m_mm1;
CDialog2    m_mm2;
```

（4）在主对话框的 OnInitDialog()函数中添加如下的代码：

```
TCITEM item;
item.mask = TCIF_TEXT;
item.pszText = "第一页";
m_tabctrl.InsertItem (0,&item);
item.pszText ="第二页";
m_tabctrl.InsertItem (1,&item);
m_mm1.Create (IDD_DIALOG1,&m_tabctrl);
m_mm2.Create (IDD_DIALOG2,&m_tabctrl);
m_mm1.SetWindowPos(NULL,10,30,400,100,SWP_SHOWWINDOW);
m_mm2.SetWindowPos(NULL,10,30,400,100,SWP_HIDEWINDOW);
```

其中，两个 InsertItem()函数的调用是为了给标签控件增加两个标签页面，文本是标题。

SetWindowPos()函数用于设置这两个对话框在应用程序窗口中的位置、显示或隐藏状态。

（5）在主对话框中为标签控件添加一个标签选择改变（TCN_SELCHANGE）的控件通知消息，以便在用户选择标签时通知主对话框。在主对话框的编辑界面右击标签控件，选择添加一个事件就可以完成这个操作。

在事件处理中添加如下代码：

```
void CtabdialogDlg::OnTcnSelchangeTab1(NMHDR *pNMHDR, LRESULT    *pResult) {
    CRect    r;
    m_tabctrl.GetClientRect (&r);
    switch(m_tabctrl.GetCurSel())
    {
        case 0:
        m_mm1.SetWindowPos(NULL,10,30,r.right-20,r.bottom-40,SWP_SHOWWINDOW);
        m_mm2.SetWindowPos(NULL,10,30,r.right-20,r.bottom-40,SWP_HIDEWINDOW);
        break;
        case 1:
        m_mm1.SetWindowPos(NULL,10,30,r.right-20,r.bottom-40,SWP_HIDEWINDOW);
        m_mm2.SetWindowPos(NULL,10,30,r.right-20,r.bottom-40,SWP_SHOWWINDOW );
        break;
    }
    *pResult=0;
}
```

要想知道用户选择哪个标签页，就要使用 m_tabctrl.GetCurSel()函数。为了不使显示的子对话框覆盖标签控件，要先获得标签控件的尺寸，然后再设置各页面的尺寸。

5.13　Visual C++ 2010 IP 控件简介与开发

IP 地址控件很像一个编辑框，它允许用户输入和操作一个 IP（Internet Protocol，网络层协议）地址。IP 地址控件由类 CIPAddressCtrl（继承于 CWnd）管理，其成员函数比较简单，包括 IsBlank()（确认是否控件的 R 个域全为空）、ClearAddress()（清除控件内容）、GetAddress()（得到输入的 IP 地址）、SetAddress()（设置控件 4 个域的内容）以及 SetFileRange()（设置在控件 4 个域中所输入的数的范围）等。

IP 控件只有一个关联事件——IPN_FIELDCHANGED，当控件的域被改变时触发。

下面通过一个具体的实例来讲解一下 IP 控件的使用方法。

```
#include <afxsock.h>

//把 CString 表示的 IP 地址写到 IP Address 中
CString strIP="192.168.0.17";
DWORD dwIP=ntohl(inet_addr(strIP));
m_IP.SetAddress(dwIP);

//读取 IP Address 的内容
DWORD dwIP;
IN_ADDR ia;
CString strIP;

m_IP.GetAddress(dwIP);
ia.S_un.S_addr=htonl(dwIP);
strIP=inet_ntoa(ia);
```

下面的代码的功能则是实现 IP 地址与字符串之间的转换：

```
in_addr addr;
addr.S_un.S_addr=inet_addr("192.168.1.1")
inet_ntoa(addr)
```

5.14　Visual C++ 2010 Picture 控件简介与开发

图片控件也是一种静态控件，也由 CStatic 类管理。该控件提供一种显示图形的方便的方法，只要在其属性面板中选择要显示的图形类型和要显示的图形就可以直接将其显示出来，而不需要编写任何代码。

Picture 控件的分类有两种：

（1）非动态显示图片（即图片先通过资源管理器载入，有一个固定 ID）；

（2）动态载入图片（即只需要在程序中指定图片的路径即可载入）。

为方便说明，这里已经建好一个基于对话框的工程，名为 Ttest。对话框类为 CTestDlg。

1．通过 Picture 控件非动态载入图片

方法 1：用 Picture 控件来实现：先在资源中导入一张图片，ID 为 IDB_BITMAP2；然后在对话框中添加一个 Picture 控件，右击打开属性面板，设置 Type 为 BITMAP，在显示的“Image”下拉列表框

中选择所要的图片。运行程序即可。

方法 2：用 Picture 控件实现，通过背景图。

先载入一张图片，ID 为 IDB_BITMAP2。

在 TestDlg.h 中，编写代码：

```
CBrush m_brBk;//在 public 中定义
```

在 TestDlg.cpp 中，在初始化函数 OnInitDialog()中加入：

```
BOOL CTestDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        CBitmap bmp;
        bmp.LoadBitmap(IDB_BITMAP2);
        m_brBk.CreatePatternBrush(&bmp);
        bmp.DeleteObject();
        return TRUE;   // return TRUE   unless you set the focus to a control
}
```

打开类向导，找到 WM_CTLCOLOR 消息，重载其对应的函数 OnCtlColor()，添加如下代码：

```
HBRUSH   CTestDlg::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
        HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor);
        if (pWnd == this)
        {
            return m_brBk;
        }
        return hbr;
}
```

2．通过 Picture 控件动态载入图片

方法 1：图像控件（本例用 KoDak 图像编辑控件）。

（1）首先应该保证系统中有这个控件。注意：它不能单独使用，必须和其他几个控件（特别是 Imgcmn.dll）一起使用。如果没有，从其他计算机中复制过来即可，这几个文件是 Imgadmin.ocx、Imgcmn.dll、Imgedit.ocx、Imgscan.ocx、Imgshl.dll、Imgthumb.ocx、Imgutil.dll，把它们复制到 WINDOWS\system 目录下，然后用 regsvr32.exe 分别注册它们。

（2）打开工程，进入资源管理器，在对话框上右击，选择“Insert ActiveX Control”命令，选择 KoDak 图像编辑控件，大小任意。

（3）在对话框中选中该控件，为其添加变量——m_ctrlPicture。

（4）在 BOOL CTestDlg::OnInitDialog()中添加如下代码：

```
BOOL CTestDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    m_ctrlPicture.SetImage("aa.jpg");                    //保证图像在工程目录下，也可以写绝对路径
    m_ctrlPicture.Display();
    return TRUE;   // return TRUE unless you set the focus to a control
                   // EXCEPTION: OCX Property Pages should return FALSE
}
```

编译并运行程序即可，此种方法的优点就是可以针对多种图像格式。

方法 2：Picture 控件。通过 Cbitmap 和 HBITMAP，直接用 OnPaint()绘制。

首先在 CTestDlg 类中声明一个变量——CBitmap m_bmp；

然后在对话框中加入一个 Picture 控件，名为 IDC_STATIC1，并编写代码如下：

```
BOOL CDisplayPic::OnInitDialog()
{
    CDialog::OnInitDialog();
    if( m_bmp.m_hObject != NULL )//判断
        m_bmp.DeleteObject();

    //载入图片
    HBITMAP hbmp=(HBITMAP)::LoadImage(AfxGetInstanceHandle(),"c:\\aaa.bmp", IMAGE_BITMAP, 0, 0, LR_
CREATEDIBSECTION|LR_LOADFROMFILE);
    if( hbmp == NULL )
    return FALSE;

    //该断程序用来取得加载的 BMP 图像的信息
    m_bmp.Attach( hbmp );
    DIBSECTION ds;
    BITMAPINFOHEADER &bminfo = ds.dsBmih;
    m_bmp.GetObject( sizeof(ds), &ds );
    int cx="bminfo".biWidth;                    //得到图像宽度
    int cy="bminfo".biHeight;                   //得到图像高度
    //得到图像的宽度和高度后，就可以对图像大小进行适应，即调整控件的大小，使其正好显示一张图片
    CRect rect;
    GetDlgItem(IDC_STATIC1)->GetWindowRect(&rect);
    ScreenToClient(&rect);
    GetDlgItem(IDC_STATIC1)->MoveWindow(rect.left,rect.top,cx,cy,true);
    //调整大小
    return TRUE;  // return TRUE unless you set the focus to a control
    // EXCEPTION: OCX Property Pages should return FALSE
}
```

图片加载成功，标签大小也适合后，下面就要绘制图像了，打开类向导，重载 WM_PAINT 消息：

```
void CDisplayPic::OnPaint()
{
    //以下 3 种情况，任选一种会是不同的效果（只能有一种存在）
      //CPaintDC dc(this);                    //若用此句，得到的是对话框的 DC，图片将被绘制在对话框上
    CPaintDC dc(GetDlgItem(IDC_STATIC1));   //若用此句，得到 Picture 控件的 DC，图像将被绘制在控件上
    //  CDC dc;
    //  dc.m_hDC=::GetDC(NULL);             //若用此两句，得到的是屏幕的 DC，图片将被绘制在屏幕上

    CRect rcclient;
    GetDlgItem(IDC_STATIC1)->GetClientRect(&rcclient);
    CDC memdc;
    memdc.CreateCompatibleDC(&dc);
    CBitmap bitmap;
    bitmap.CreateCompatibleBitmap(&dc, rcclient.Width(), rcclient.Height());
    memdc.SelectObject( &bitmap );
    CWnd::DefWindowProc(WM_PAINT, (WPARAM)memdc.m_hDC , 0);
```

```
        CDC maskdc;
        maskdc.CreateCompatibleDC(&dc);
        CBitmap maskbitmap;
        maskbitmap.CreateBitmap(rcclient.Width(), rcclient.Height(), 1, 1, NULL);
        maskdc.SelectObject( &maskbitmap );
        maskdc.BitBlt( 0, 0, rcclient.Width(), rcclient.Height(), &memdc, rcclient.left, rcclient.top, SRCCOPY);
        CBrush brush;
        brush.CreatePatternBrush(&m_bmp);
        dc.FillRect(rcclient, &brush);
        dc.BitBlt(rcclient.left, rcclient.top, rcclient.Width(), rcclient.Height(), &memdc, rcclient.left, rcclient.top,SRCP-
AINT);
        brush.DeleteObject();
        // Do not call CDialog::OnPaint() for painting messages
    }
```

5.15 Visual C++ 2010 Slider 控件简介与开发

CSliderCtrl 类表示滑块控件（Slider Control），也称为跟踪条控件。滑块控件是一个包含一个滑块和可选刻度线的窗口。当用户使用鼠标或者箭头键移动滑块时，滑块控件发送通知消息以指示此更改。

当希望用户选择一个不连续的值或者某个范围中的一组连续值时，滑块控件很有帮助。例如，通过将滑块移动到给定刻度线，使用滑块控件可以使用户设置键盘的重复速度。

滑块控件中的滑块以开发人员在创建它时所指定的增量移动。例如，如果指定滑块控件的范围为 5，则滑块只能占有 6 个位置：一个位置在滑块控件的左侧，而此范围中的每个增量各有一个位置。通常，这些位置中的每一个都用一个刻度线标识。

滑块控件的典型用法遵循以下模式。

（1）创建控件。如果在对话框模板中指定此控件，当创建对话框时会自动创建它（在对应滑块控件的对话框类中应该有一个 CSliderCtrl 成员。）；另一种创建方法是，使用 Create()成员函数将此控件创建为任何窗口的子窗口。

（2）调用各种 Set 成员函数设置控件值。可以进行的更改包括设置滑块的最小位置和最大位置，绘制刻度线，设置选择范围和重新定位滑块。对于对话框中的控件，应当在此对话框的 OnInitDialog()函数中进行这些更改。

（3）在用户和控件的交互过程中，控件将发送各种通知消息。通过调用 GetPos()成员函数，可以从此控件中提取滑块值。

（4）使用完此控件后，需要确保它被正确销毁。如果滑块控件在对话框中，它和 CSliderCtrl 对象将被自动销毁；否则开发人员需要确保此控件和 CSliderCtrl 对象都被正确销毁。

5.15.1 滑块控件样式

滑块控件（CSliderCtrl）可以为垂直方向或水平方向的，可以在任何一侧有、两侧都有或两侧都没有刻度线，也可以用来指定连续数值的范围。通过使用滑块控件样式（当创建滑块控件时指定），开发人员可以控制下面的属性。

TBS_HORZ 和 TBS_VERT：确定滑块控件的方向。如果没有指定方向，则控件为水平方向。

TBS_AUTOTICKS：创建数值范围中的每一个增量都有刻度线的滑块控件。当调用 SetRange()成员函数时自动添加这些刻度线。如果没有指定 TBS_AUTOTICKS，可以用成员函数（如 SetTic()和 SetTicFreq()）指定刻度线的位置。若要创建不显示刻度线的滑块控件，可以使用 TBS_NOTICKS 样式。

可以在滑块控件的任何一侧或两侧都显示刻度线。对水平滑块控件，可以指定 TBS_BOTTOM 或 TBS_TOP 样式；对垂直滑块控件，可以指定 TBS_RIGHT 或 TBS_LEFT 样式（TBS_BOTTOM 和 TBS_RIGHT 是默认设置）。对在任何方向上的滑块控件两侧的刻度线，指定 TBS_BOTH 样式。

只有在创建滑块控件时指定 TBS_ENABLESELRANGE 样式，滑块控件才显示选择范围。当滑块控件为该样式时，选择范围起始和结束位置的刻度线显示为三角形（而不是垂直短画线），并且选择范围被突出显示。例如，选择范围可能在简单调度应用程序中有用。用户可以对应于一天中的小时来选择刻度线的范围，以标识已安排的会议时间。

默认情况下，滑块控件的滑块长度随选择范围的更改而变化。如果滑块控件为 TBS_FIXEDLENGTH 样式，即使选择范围更改，滑块长度也保持不变。样式为 TBS_NOTHUMB 的滑块控件不包含滑块。

5.15.2　滑块控件成员函数

应用程序可以调用滑块控件的成员函数来检索有关它的信息并更改其特征。

若要检索滑块的位置（即用户已选择的值），请使用 GetPos()成员函数；若要设置滑块的位置，请使用 SetPos()成员函数。在任何时候都可以用 VerifyPos()成员函数来确保滑块位于最小值和最大值之间。

滑块控件的范围为其能够表示的连续数值集。第一次创建滑块控件后，大多数应用程序用 SetRange()成员函数设置其范围。在用 SetRangeMax()和 SetRangeMin()成员函数创建滑块控件后，应用程序可以动态改变其范围。允许动态更改范围的应用程序一般在用户处理完滑块控件后检索最后的范围设置。若要检索这些设置，请使用 GetRange()、GetRangeMax()和 GetRangeMin()成员函数。

应用程序可以用 TBS_AUTOTICKS 样式使滑块控件的刻度线自动显示，但是，如果应用程序需要控制刻度线的位置或频率，可以使用若干个成员函数。

若要设置刻度线的位置，应用程序可以使用 SetTic()成员函数。SetTicFreq()成员函数允许应用程序设置在滑块控件范围内以标准时间间隔显示的刻度线。例如，应用程序可以用该成员函数仅显示 1～100 范围内的 10 个刻度线。

若要检索对应于刻度线范围内的索引，请使用 GetTic()成员函数。GetTicArray()成员函数检索这些索引的数组。若要在工作区坐标中检索刻度线的位置，请使用 GetTicPos()成员函数。应用程序可以通过使用 GetNumTics()成员函数检索刻度线的数目。

ClearTics()成员函数用于移除所有滑块控件的刻度线。

滑块控件的行大小确定应用程序接收 TB_LINEDOWN 或 TB_LINEUP 通知消息时滑块移动的距离；同样，页面大小确定对 TB_PAGEDOWN 和 TB_PAGEUP 通知消息的响应。通过使用 GetLineSize()、SetLineSize()、GetPageSize()和 SetPageSize()成员函数，应用程序可以检索和设置行和页面的大小值。

应用程序可以用成员函数检索滑块控件的尺寸。GetThumbRect()成员函数检索滑块的边框。GetChannelRect()成员函数检索滑块控件通道的边框（通道为滑块在其上移动的区域并且包含选定范围时突出显示的部分）。

如果滑块控件为 TBS_ENABLESELRANGE 样式，用户可以从中选择连续的数值范围。若干成员函数允许动态调整选择范围。SetSelection()成员函数设置选择范围的起始和结束位置。当用户设置完选

择范围后，应用程序可以通过使用 GetSelection()成员函数来检索设置。若要清除用户的选择范围，则应使用 ClearSel 成员函数。

5.15.3 滑块控件通知消息

滑块控件通过向父窗口发送 WM_HSCROLL 或 WM_VSCROLL 消息（取决于滑块控件的方向），将用户操作通知其父窗口。若要处理这些消息，请将 WM_HSCROLL 和 WM_VSCROLL 消息的处理程序添加到父窗口，将向 OnHScroll()和 OnVScroll()成员函数传递通知代码、滑块的位置和指向 CSliderCtrl 对象的指针。注意，指针的类型为 CScrollBar 即使它指向 CSliderCtrl 对象。若要操作滑块控件，可能需要转换该指针的类型。

滑块控件不是使用滚动条通知代码，而是发送不同的通知代码集。当用户使用键盘与滑块控件交互时，滑块控件才发送 TB_BOTTOM、TB_LINEDOWN、TB_LINEUP 和 TB_TOP 通知消息；只有当用户使用鼠标与滑块控件交互时，才发送 TB_THUMBPOSITION 和 TB_THUMBTRACK 通知消息；在两种情况下，都发送 TB_ENDTRACK、TB_PAGEDOWN 和 TB_PAGEUP 通知代码。

表 5.23 列出了滑块控件通知消息和导致发送通知的事件（虚键控代码和鼠标事件）。

表 5.23　　滑块控件的通知消息

通 知 消 息	导致发送通知的事件
TB_BOTTOM	VK_END
TB_ENDTRACK	WM_KEYUP（用户释放了发送相关虚键控代码的键）
TB_LINEDOWN	VK_RIGHT 或 VK_DOWN
TB_LINEUP	VK_LEFT 或 VK_UP
TB_PAGEDOWN	VK_NEXT（用户单击了滑块下面或右侧的通道）
TB_PAGEUP	VK_PRIOR（用户单击了滑块上面或左侧的通道）
TB_THUMBPOSITION	WM_LBUTTONUP 在 TB_THUMBTRACK 通知消息之后
TB_THUMBTRACK	滑块移动（用户拖动了滑块）
TB_TOP	VK_HOME

5.16 Visual C++ 2010 Scroll Bar 控件简介与开发

滚动条（Scroll Bar）控件主要由滚动框和滚动框中的一个或两个滚动箭头组成。在 Windows 系列软件中，滚动条应用非常普遍。

滚动条的主要作用是从某一预定义值的范围内快速有效地选取一个整数值。按照走向，滚动条可以分为水平滚动条和垂直滚动条两种。在滚动条中有一个滚动框，用来表示当前的值。单击滚动条，可以使滚动框移动一页或一行，也可以直接拖动滚动框。滚动条既可以作为一个独立控件存在，也可以作为窗口、列表框和组合框的一部分。

需要指出的是，从性质上划分，滚动条可分为标准滚动条和滚动条控件两种。标准滚动条是由 WS_HSCROLL 或 WS_VSCROLL 风格指定的，它不是一个实际的窗口，而是窗口的一个组成部分（例如列表框中的滚动条），只能位于窗口的右侧（垂直滚动条）或底端（水平滚动条）。标准滚动条是在窗口的非客户区中创建的。与之相反，滚动条控件并不是窗口的一个零件，而是一个实际的窗口，可以放置在窗口客户区的任意位置，它既可以独立存在，也可以与某一个窗口组合来行使滚动窗口的职能。由于滚动条控件是一个独立窗口，因此它可以拥有输入焦点，可以响应光标控制键，如 PgUp、

PgDown、Home 和 End。

与滚动条控件有关的 CScrollBar 类的成员函数如表 5.24 所示。

表 5.24　　CScrollBar 类的成员函数

函　数	说　明
Create()	负责创建滚动条控件
GetSccrollPos()	获得滚动框的当前位置
SetScrollPos()	设置滚动框的当前位置
GetScrollRange()	获得指定滚动条的滚动范围
SetScrollRange()	设置指定滚动条的滚动范围
SetScrollInfo()	设置指定滚动条的有关信息
GetScrollInfo()	获得与指定滚动条有关的信息
ShowScrollBar()	显示或隐藏滚动条
GetScrollLimit()	获得滚动条的最大滚动位置
EnableScrollBar()	启用或禁用一个或两个滚动箭头

MFC 的 CScrollBar 类封装了滚动条控件。CScrollBar 类的 Create()成员函数负责创建控件，其原型如下：

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

如果创建成功，返回 True，否则返回 False。

第一个参数 dwStyle 指定了控件的风格。

```
WS_CHILD        指定该控件是一个子窗口
WS_VISIBLE      指定该控件可显示
WS_DISABLED     指定该控件不可显示
WS_GROUP        指定该类控件作为一组
BS_HORZ         指定该滚动条是水平方向滚动
BS_VERT         指定该滚动条是垂直方向滚动
```

下面给出几种方案：如果想要创建一个普通的水平滚动条，应指定风格为 WS_CHILD|WS_VISIBLE|BS_HORZ；如果要创建一个普通的垂直滚动条，应指定风格为 WS_CHILD|WS_VISIBLE|BS_VERT。

第二个参数 rect 说明了控件的位置和尺寸。

第三个参数 pParentWnd 指向父窗口，该参数不能为 NULL。

第四个参数 nID 则说明了控件的 ID。

例如：

```
CscrollBar m_ScrollBar
m_ScrollBar.Create(SBS_HORZ | SBS_TOPALIGN | WS_CHILD | WS_VISIBLE,
CRect(5, 5, 100, 30), this, 100));
```

CScrollBar 类的 GetScrollInfo()函数用于获得与指定滚动条有关的信息。

```
CScrollBar::GetScrollInfo
BOOL GetScrollInfo( LPSCROLLINFO lpScrollInfo, UINT nMask );
```

如果消息获取了任何值，则返回 True；否则返回 False。

第一个参数 lpScrollInfo 指向一个 SCROLLINFO 结构的指针。参见“Win32 程序员参考”，可以获得更多有关这个结构的信息。

第二个参数 nMask 指定要获取的滚动条参数，通常使用的是 SIF_ALL，表示是 SIF_PAGE、SIF_POS、SIF_TRACKPOS 和 SIF_RANGE 的组合。参见 SCROLLINFO，可以获得有关 nMask 值的更

多信息。

此成员函数用来获取由 SCROLLINFO 结构保存的关于一个滚动条的信息，包括滚动位置的最小和最大值、页面的大小和滚动框（拇指）的位置。GetScrollInfo()允许应用程序使用 32 位的滚动位置信息。

MFC Windows 消息句柄表示了滚动条的位置，CWnd::OnHScroll()和 CWnd::OnVScroll()只提供 16 位的位置数据，GetScrollInfo()和 SetScrollInfo()提供 32 位的滚动条位置数据。因此，一个应用程序在处理 GetScrollInfo()或 SetScrollInfo()时，可以调用 GetScrollInfo()来获取 32 位的滚动条位置数据。

下面是一个创建滚动条控件的例子。代码如下：

```
CScrollBar m_scrollbar;    //usuallyembeddedinparentclass
m_scrollbar.Create(WS_VISIBLE|WS_CHILD|dwStyle,rect,pParentWnd,id);
```

说明。

（1）与 Windows 在任何一个窗口的非客户区绘制滚动条不同，事实上，这些滚动条自身是一些完整的子窗口。一个父窗口可以有它自己的滚动条和任意数量的滚动条控件窗口；但是，非客户区滚动条和控件窗口滚动条都用 WM_HSCROLL 和 WM_VSCROLL 窗口消息回传到父窗口；要在父窗口中区别它们，需要在消息的 lParam 中查看滚动条句柄，当为该消息添加一个处理函数时，MFC 把该滚动条句柄封装到一个 CScrollBar 类中。

（2）当单独设置 SBS_HORZ 和 SBS_VERT 时，滚动条的宽度和高度可以是任意值；当单独设置 SBS_TOPALIGN 和 SBS_BOTTOMALIGN 时，滚动条要设置一个标准宽度。

（3）创建具有 SBS_SIZEGRIP 风格的滚动条控件，允许用户拖动控件所在的窗口来改变其大小，

5.17 Visual C++ 2010 Hot Key 控件简介与开发

由 CHotKeyCtrl 类表示的热键（Hot Key）控件是一个窗口，显示用户输入的组合键（如 Ctrl+Shift+Q）的文本表示形式，还维护此键的内部表示形式（虚键控代码的形式）和一组表示转换状态的标志。热键控件实际上不设置热键，此操作由程序执行。

使用热键控件获取将热键与窗口或线程相关联的用户输入。热键控件经常用于对话框，如要求用户分配热键时可能要显示的对话框。程序负责从热键控件中检索描述热键的值，并调用适当的函数将热键与窗口或线程相关联。

5.17.1 使用热键控件

热键控件的典型用法遵循下面的模式。

（1）创建控件。如果在对话框模板中指定此控件，当创建对话框时会自动创建它（对话框类中应该包含与热键控件相对应的 CHotKeyCtrl）；或者，可以使用 Create()成员函数创建控件作为任何窗口的子窗口。

（2）如果要为控件设置默认值，请调用 SetHotKey()成员函数；如果要禁止某些转换状态，请调用 SetRules()；对于对话框中的控件，最好在对话框的 OnInitDialog()函数中完成。

（3）当热键控件有焦点时，用户通过按热键与控件交互，然后以某种方式指示此任务已完成，多半是通过单击对话框中的按钮。

（4）当程序得到用户已选定了热键的通知消息时，应使用成员函数 GetHotKey()从热键控件中检索虚拟键和转换状态值。

（5）知道用户选定的键后，可以使用 5.17.2 小节中描述的一种方法设置热键。

（6）如果热键控件在对话框中，此控件和 CHotKeyCtrl 对象将自动销毁；如果不在对话框中，开发人员需要确保正确地将此控件和 CHotKeyCtrl 对象全都销毁。

5.17.2　设置热键

应用程序可以按下列两种方法之一使用由热键控件（CHotKeyCtrl）提供的信息：

- 通过向将要激活的窗口发送 WM_SETHOTKEY 消息，设置一个用于激活非子窗口的全局热键；
- 通过调用 Windows 函数 RegisterHotKey()来设置线程特定的热键。

1．全局热键

全局热键与特定的非子窗口关联，它使用户得以从系统的任何部分激活窗口。应用程序通过向特定的窗口发送 WM_SETHOTKEY 消息来设置该窗口的全局热键，例如，如果 m_HotKeyCtrl 是 CHotKeyCtrl 对象，pMainWnd 是指向按热键时要激活的窗口的指针，则可以使用下列代码将控件中指定的热键和 pMainWnd 指向的窗口关联起来。

```
WORD wKeyAndShift = m_HotKeyCtrl.GetHotKey( );
pMainWnd->SendMessage( WM_SETHOTKEY, wKeyAndShift );
```

每当用户按全局热键时，指定的窗口就会接收 WM_SYSCOMMAND 消息，该消息将 SC_HOTKEY 指定为命令类型，还将激活接收它的窗口。由于该消息不包括按下的确切键的任何信息，因此使用此方法不能区分可能附加到同一窗口的不同热键。在发送 WM_SETHOTKEY 的应用程序退出之前，热键一直保持有效。

2．线程特定的热键

应用程序使用 Windows RegisterHotKey()函数设置线程特定的热键（CHotKeyCtrl）。用户按线程特定的热键后，Windows 将一个 WM_HOTKEY 通知消息发布到特定线程的消息队列的开始位置。WM_HOTKEY 消息包含虚键控代码、转换状态及所按的特定热键的用户定义 ID。

请注意，用于调用 RegisterHotKey()的转换状态标志和 GetHotKey()成员函数返回的标志不同；在调用 RegisterHotKey()之前必须转换这些标志。

5.18　Visual C++ 2010 Animation 控件简介与开发

Animation 控件是一种复杂的控件，可以播放 AVI 格式的视频文件。

Animation 控件是由 CAnimateCtrl 类表示的，是显示影音交叉存取技术（Audio Video Interleaved，AVI）格式（标准 Windows 视频/音频格式）剪辑的窗口。AVI 剪辑是一系列位图帧，类似于电影。

由于显示 AVI 剪辑的同时，线程在继续执行，因此动画控件的一个常用用途是在长操作中指示系统活动。例如，Windows 当系统搜索文件时，窗口中显示一个移动的放大镜。

动画控件只能播放简单的 AVI 剪辑，不支持声音（有关完整的限制列表，请参见 CAnimateCtrl）。功能非常有限，并且受更改的限制，因此如果需要控件提供多媒体播放和/或录制功能，应该使用其他控件（如 MCIWnd 控件）。

5.18.1　使用动画控件

动画控件的典型用法遵循下面的模式。

（1）创建控件。如果在对话框模板中指定此控件，创建对话框时会自动创建它，对话框类中应该有与动画控件相对应的 CAnimateCtrl 成员；或者，也可以使用 Create()成员函数创建控件，作为任何窗口的子窗口。

（2）通过调用 Open()成员函数将 AVI 剪辑加载到动画控件中。如果动画控件在对话框中，最好在对话框类的 OnInitDialog()函数中完成加载。

（3）通过调用 Play()成员函数播放剪辑。如果动画控件在对话框中，最好在对话框类的 OnInitDialog()函数中完成播放。如果为动画控件设置了 ACS_AUTOPLAY 样式，则不必调用 Play()。

（4）如果要显示剪辑中的某些部分，或逐帧播放剪辑，请使用 Seek()成员函数；若要停止正在播放的剪辑，请使用 Stop()成员函数。

（5）如果不想立即销毁控件，请通过调用 Close()成员函数从内存中移除剪辑。

（6）如果动画控件在对话框中，此控件和 CAnimateCtrl 对象将自动销毁；如果不在对话框中，开发人员需要确保正确地将控件和 CAnimateCtrl 对象全都销毁。自动销毁控件将关闭 AVI 剪辑。

5.18.2 动画控件发送的通知

动画控件（CAnimateCtrl）发送两种不同类型的通知消息。通知以 WM_COMMAND 消息的形式发送。

动画控件开始播放剪辑时发送 ACN_START 消息，完成或停止播放剪辑时发送 ACN_STOP 消息。

5.19 Visual C++ 2010 Spin 控件简介与开发

数值调节钮控件（Spin Control）（也称为 up-down 控件）类似于一个按钮，提供一对箭头按钮，单击它们可以调整数值的大小，该值称为当前位置。该位置保持在数值调节钮范围内，当用户单击上箭头按钮时，该位置移向最大值；当用户单击下箭头按钮时，该位置移向最小值。

数值调节钮控件在 MFC 中由类 CSpinButtonCtrl 表示。

数值调节钮的默认范围为最大设置到零（0），最小设置到 100。由于最大值小于最小值，因此单击上箭头按钮时将减小位置，而单击下箭头按钮时将增大位置。用 CSpinButtonCtrl::SetRange()调整这些值。

一般情况下，当前位置显示在同伴控件中，同伴控件又称为“合作者窗口”。

5.19.1 数值调节钮的样式

数值调节钮（CSpinButtonCtrl）的许多设置由样式控制，可以用对话框编辑器中的“属性”面板设置下列样式。

- **方向**：“垂直”、“水平”。控制箭头按钮的方向。与 UDS_HORZ 样式相关联。
- **对齐**：“取消附加”、“左”和“右”中的一个。控制数值调节钮的位置。合作者窗口旁边的数值调节钮的“左”位置和“右”位置。合作者窗口的宽度将减小以适应数值调节钮。与 UDS_ALIGNLEFT 和 UDS_ALIGNRIGHT 样式相关联。
- **自动合作者**：自动以 Z 顺序选择上一窗口作为数值调节钮的合作者窗口。在对话框模板上，这是以【Tab】键顺序在数值调节钮之前的控件。与 UDS_AUTOBUDDY 样式相关联。

- **设置合作者整数**：导致数值调节钮控件随当前位置的更改而增大和减小合作者窗口的标题。与 UDS_SETBUDDYINT 样式相关联。
- **无千位分隔符**：不在合作者窗口标题的数值中插入千位分隔符。与 UDS_NOTHOUSANDS 样式相关联。
- **注意**：若要用对话框数据交换（DDX）从合作者控件中获取整数值，请设置该样式。DDX_Text 不接受嵌入的千位分隔符。
- **换行**：当增大或减小的数值超出控件范围时导致位置“换行”。与 UDS_WRAP 样式相关联。
- **箭头键**：当按上箭头键和下箭头键时导致数值调节钮增大或减小位置。与 UDS_ARROWKEYS 样式相关联。

5.19.2　数值调节钮成员函数

有几个成员函数可用于数值调节钮控件（CSpinButtonCtrl）。使用这些函数更改数值调节钮的下列属性。

- **加速**：可以调整用户单箭头按钮时位置更改的速率。若要使用加速，请使用 SetAccel()和 GetAccel()成员函数。
- **基数**：可以更改用于在合作者窗口标题中显示位置的基数（10 或 16）。若要使用基数，请使用 GetBase()和 SetBase()成员函数。
- **合作者窗口**：可以动态设置合作者窗口。若要查询或更改作为合作者窗口的控件，请使用 GetBuddy()和 SetBuddy()成员函数。
- **位置**：可以查询和更改位置。若要直接处理位置，请使用 GetPos()和 SetPos()成员函数。由于合作者控件的标题可能已经更改（例如，在合作者为编辑控件的情况下），所以使用 GetPos()检索当前标题并相应地调整位置。
- **范围**：可以更改数值调节钮的最大位置和最小位置。默认情况下，最大值设置为 0，最小值设置为 100。由于默认最大值小于默认最小值，所以箭头按钮的操作正好与直觉相反。一般情况下，用 SetRange()成员函数设置范围。若要查询范围，请使用 GetRange()。

5.20　Visual C++ 2010 GroupBox 控件简介与开发

Windows 窗体 GroupBox 控件用于为其他控件提供可识别的分组。通常，使用分组框按功能细分窗体，在分组框中对所有选项进行分组，为用户提供逻辑可视化线索，并且在设计时使所有控件可以方便地移动。移动单个 GroupBox 控件时，它包含的所有控件也会一起移动。

GroupBox 控件类似于 Panel 控件，但只有 GroupBox 控件显示标题，而且只有 Panel 控件可以有滚动条。

GroupBox 控件用于对其他控件分组。对控件分组的原因有 3 个：

- 对相关窗体元素进行可视化分组以构造一个清晰的用户界面；
- 创建编程分组（例如，单选按钮分组）；
- 设计时，将多个控件作为一个单元移动。

创建一组控件的方法如下。

（1）在窗体上绘制 GroupBox 控件。

（2）向分组框添加其他控件，在分组框中绘制各个控件。

如果要将现有控件放到分组框中，可以选定所有这些控件并将其剪切到剪贴板，选择 GroupBox 控件，再将选定的控件粘贴到分组框中；也可以将它们直接拖到分组框中。

（3）将分组框的 Text 属性设置为适当标题。

GroupBox 显示围绕一组控件的框架（带或不带标题），组框是可用于定义控件组的容器控件。

组框的典型用途是包含 RadioButton 控件的逻辑组。如果有两个分组框，每个分组框都包含多个选项按钮（也称为单选按钮），每组按钮都互相排斥，则为每个分组设置一个选项值。

通过使用控件属性的 Add()方法，可将控件添加到 GroupBox。

（1）只有 GroupBox 控件中包含的控件才可以被选中或接收焦点，整个 GroupBox 本身不能被选中或者接收焦点。

（2）GroupBox 不能显示滚动条。

下面的代码实例化并创建一个 GroupBox 和两个 RadioButton 控件。向组框中添加单选按钮，并将该组框添加到窗体中：

```
private:
    void InitializeMyGroupBox()
    {
        // Create and initialize a GroupBox and two RadioButton controls
        GroupBox^ groupBox1 = gcnew GroupBox;
        RadioButton^ radioButton1 = gcnew RadioButton;
        RadioButton^ radioButton2 = gcnew RadioButton;

        // Add the RadioButtons to the GroupBox
        groupBox1->Controls->Add( radioButton1 );
        groupBox1->Controls->Add( radioButton2 );

        // Add the GroupBox to the Form
        Controls->Add( groupBox1 );
    }
```

GroupBox 控件可以将其他的控件放在一个组中进行管理，即其他控件可以放置在 GroupBox 控件之上，并且这些控件属于同一组。GroupBox 控件用一条分界线来划分每一组，并且开发人员可以对每一组进行标记。通过单击或者拖动 ToolBox 窗口上的 GroupBox 控件可以在对话框中添加一个 GroupBox 控件，双击该控件，则在窗口右侧中显示“Properties”面板，开发人员可以修改 GroupBox 的 Caption、ID、Dock 等属性，如图 5.7 所示。

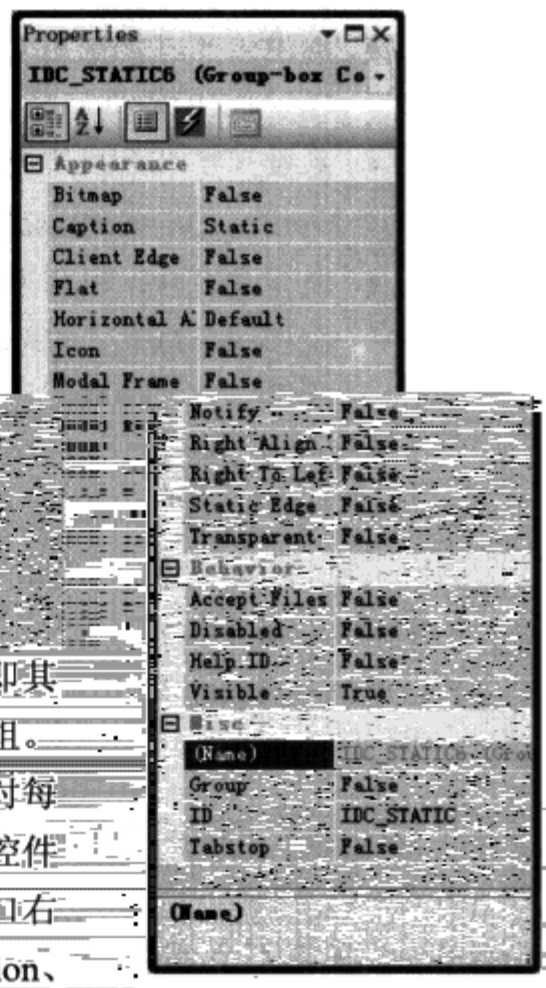


图 5.7 设置 GroupBox 控件的属性

5.21 Visual C++ 2010 Data Time Picker 控件简介与开发

日期和时间选择器控件（CDateTimeCtrl）实现了以一种直观且可识别的方法来输入或选择特定的日期。此控件的主接口在功能上类似于组合框，但是，如果用户展开此控件，会出现月历（Month Calendar）控件（默认情况下），允许用户指定特定日期，指定日期后，月历控件便自动消失。

若要在项目中同时使用 CDateTimePicker 类和 CMonthCalCtrl 类，则必须包含 AFXDTCTL.h（通常在项目的 STDAFX.h 文件中）。

5.21.1　创建日期和时间选择器控件

创建日期和时间选择器控件的方式取决于是在对话框窗口中使用此控件还是在非对话框窗口中使用此控件。

1．直接在对话框窗口中使用 CDateTimeCtrl

（1）在对话框编辑器中，将 Date Time Picker 控件添加到对话框模板资源中，并指定其控件 ID。

（2）使用日期和时间选择器控件的属性面板指定任何所需样式。

（3）使用添加成员变量向导添加带 Control 属性的 CDateTimeCtrl 类型的成员变量，可以使用此成员调用 CdateTimeCtrl 的成员函数。

（4）对于需要处理的任何 Date Time Picker 控件 通知消息，使用属性面板在对话框类中映射处理函数。

（5）在 OnInitDialog()中设置 CDateTimeCtrl 对象的任何附加样式。

2．在非对话框窗口中使用 CDateTimeCtrl

（1）在视图类或窗口类中声明该控件。

（2）调用此控件的 Create()成员函数，可能在 OnInitialUpdate()中，也可能与父窗口的 OnCreate()处理函数一样早（如果正在创建此控件的子类）设置此控件的样式。

5.21.2　访问嵌入的月历控件

通过调用 GetMonthCalCtrl()成员函数，可以从 CDateTimeCtrl 对象访问嵌入的月历控件对象。

只有当日期和时间选择器控件没有设置 DTS_UPDOWN 样式时，才使用嵌入的月历控件。

这对在显示该嵌入控件之前修改某些属性很有帮助。为此，可处理 DTN_DROPDOWN 通知，检索月历控件（使用 CDateTimeCtrl::GetMonthCalCtrl()），然后进行修改。遗憾的是，月历控件不能持久保持。

换句话说，只有当用户请求显示月历控件时，才创建新月历控件（DTN_DROPDOWN 通知之前）；当用户解除此控件时，它就被销毁（DTN_CLOSEUP 通知之后），这意味着显示该嵌入控件之前对属性所做的修改，在该嵌入控件被解除之后都将丢失。

下面的示例使用 DTN_DROPDOWN 通知的处理程序说明了此过程，该代码通过调用 SetMonthCalColor()将月历控件的背景色改为灰色。

```
void CMyDlg::OnDropdown(NMHDR* pNMHDR, LRESULT* pResult)
{
    //set the background color of the month to gray
    COLORREF clr= RGB(100, 100, 100);
    SetMonthCalColor(MCSC_MONTHBK, clr);
    *pResult = 0;
}
```

如前所述，当解除月历控件时，对该嵌入控件属性所做的所有修改都将丢失，但有两个例外，第一个例外是月历控件的颜色，这一点已经讨论过；第二个例外是月历控件所使用的字体。可以通过调用 CDateTimeCtrl::SetMonthCalFont()并传递现有字体的句柄来修改默认字体。下面的示例（其中

m_dtPicker 是日期和时间控件对象）将说明一种可能的方法：

```
CMonthCalCtrl* pMCCtrl= NULL;
pMCCtrl= m_dtPicker.GetMonthCalCtrl();
//create and initialize the font to be used
LOGFONT logFont;
logFont.lfHeight = -12;
logFont.lfWidth = 0;
logFont.lfWeight = FW_NORMAL;
logFont.lfItalic = FALSE;
logFont.lfUnderline = FALSE;
logFont.lfStrikeOut = FALSE;
logFont.lfEscapement = 0;
logFont.lfOrientation = 0;
lstrcpy(logFont.lfFaceName, _T("Verdana"));

m_MCFont.CreateFontIndirect(&logFont);
m_dtPicker.SetMonthCalFont((HFONT)m_MCFont);
```

通过调用 CDateTimeCtrl::SetMonthCalFont()更改字体后，将存储新字体并在下次显示月历控件时使用。

5.21.3 在日期和时间选择器控件中使用自定义格式字符串

默认情况下，日期和时间选择器控件提供 3 种格式来显示当前日期或时间（每种格式对应一种惟一的样式）。

- DTS_LONGDATEFORMAT: 以长格式显示日期，得出的输出形式为“Wednesday，January 3，2000”。
- DTS_SHORTDATEFORMAT: 以短格式显示日期，得出的输出形式为“1/3/00”。
- DTS_TIMEFORMAT: 以长格式显示时间，得出的输出形式为“5:31:42 PM”。

可以通过使用自定义格式字符串来自定义日期或时间的外观。此自定义字符串由现有格式字符或非格式字符组成，或由这两者共同组成。生成自定义字符串后，调用 CDateTimeCtrl::SetFormat()来传入自定义字符串，日期和时间选择器控件便可以使用自定义格式字符串显示当前值。

下面的示例代码（其中 m_dtPicker 为 CDateTimeCtrl 对象）将说明一种可能的自定义格式字符串的方法：

```
CString formatStr= _T("'Today is: 'yy'/'MM'/'dd");
m_dtPicker.SetFormat(formatStr);
```

除了自定义格式字符串，日期和时间选择器控件还支持回调字段。

5.21.4 在日期和时间选择器控件中使用回调字段

除了定义日期和时间选择器的标准格式字符，还可以通过将自定义格式字符串中的某些部分指定为回调字段来自定义输出。若要声明回调字段，请在格式字符串体中的任何位置包含一个或多个“X”字符（ASCII 码 88）。例如，字符串“'Today is: 'yy'/'MM'/'dd' (Day 'X') '”使日期和时间选择器控件将当前值的格式显示为年份、月份、日期及该日期是一年中的第几天。

回调字段中“X”的数目与将要显示的字符数不对应。

可以通过重复“X”字符来区分自定义字符串中的多个回调字段，因此，格式字符串“XXdddd MMMdd', 'yyyXXX”包含两个惟一的回调字段“XX”和“XXX”。

回调字段被视为有效字段，因此应用程序必须准备处理 DTN_WMKEYDOWN 通知消息。

日期和时间选择器控件中的回调字段实现由 3 部分组成：

（1）初始化自定义格式字符串；

（2）处理 DTN_FORMATQUERY 通知；

（3）处理 DTN_FORMAT 通知。

调用 CDateTimeCtrl::SetFormat()初始化自定义字符串，更多的相关信息，请参见 5.21.3 小节。通常在包含对话框类的 OnInitDialog()函数或包含视图类的 OnInitalUpdate()函数中设置自定义格式字符串。

其中，处理 DTN_FORMATQUERY 通知，如果控件在分析格式字符串的过程中遇到回调字段，应用程序将发送 DTN_FORMAT 和 DTN_FORMATQUERY 通知消息。通知中包含回调字段字符串，因此可以确定正查询的回调字段。

发送 DTN_FORMATQUERY 通知以检索当前回调字段中将要显示的字符串所允许的最大尺寸（以像素为单位）。若要正确计算此值，必须使用控件的显示字体计算要替换此字段的字符串的高度和宽度。调用 GetTextExtentPoint32()函数可以很容易地对字符串进行实际计算。确定大小后，将值传回应用程序并退出处理函数。

下面的示例是一种提供回调字段大小的方法：

```
NMDATETIMEFORMATQUERY* pDTFormatQuery= (NMDATETIMEFORMATQUERY *)pNMHDR;
CDC* pDC;
CFont* pFont;
CFont* pOrigFont;

//   Prepare the device context for the GetTextExtentPoint32 call.
pDC= GetDC();
pFont= GetFont();
if(!pFont)
        pFont->CreateStockObject(DEFAULT_GUI_FONT);

pOrigFont = pDC->SelectObject(pFont);

// Check to see if this is the callback segment desired. If so,
// use the longest text segment to determine the maximum
// width of the callback field, and then place the information into
// the NMDATETIMEFORMATQUERY structure.
if(!lstrcmp("X",pDTFormatQuery->pszFormat))
        ::GetTextExtentPoint32 (pDC->m_hDC, "366", 3, &pDTFormatQuery->szMax);

// Reset the font in the device context then release the context.
pDC->SelectObject(pOrigFont);
ReleaseDC(pDC);

*pResult = 0;
```

计算出当前回调字段的大小后，必须为该字段提供值，这在 DTN_FORMAT 通知的处理程序中完成。

处理 DTN_FORMAT 通知，应用程序使用 DTN_FORMAT 通知请求将被替换的字符串。下面的示例将说明一种可能的方法：

```
NMDATETIMEFORMAT* pDTFormat= (NMDATETIMEFORMAT *)pNMHDR;
COleDateTime oCurTime;
m_dtPicker.GetTime(oCurTime);
wsprintf(pDTFormat->pszDisplay,
    "%d",oCurTime.GetDayOfYear());
*pResult = 0;
```

通过将通知处理程序的第一个参数的类型转换为适当的类型，找到 NMDATETIMEFORMAT 结构的指针。

5.21.5 处理日期和时间选择器控件中的通知消息

用户与日期和时间选择器控件交互时，控件将通知消息发送到其父窗口，该窗口通常是视图或对话框对象。如果要进行某种响应，请处理这些消息。例如，当用户打开日期和时间选择器并显示嵌入的月历控件时，发送 DTN_DROPDOWN 通知。

使用属性面板为要实现的那些消息的父类添加通知处理程序。

下面描述日期和时间选择器控件发送的各种通知。

- DTN_DROPDOWN: 通知父级将要显示嵌入的月历控件。只有尚未设置 DTS_UPDOWN 样式时才发送此通知。有关此通知的更多信息，请参见 5.21.2 小节。
- DTN_CLOSEUP: 通知父级将要关闭嵌入的月历控件。只有尚未设置 DTS_UPDOWN 样式时才发送此通知。
- DTN_DATETIMECHANGE: 通知父级，控件已经发生了更改。
- DTN_FORMAT: 通知父级需要在回调字段中显示文本。有关此通知和回调字段的更多信息，请参见 5.21.4 小节。
- DTN_FORMATQUERY: 请求父级提供将在回调字段中显示的所允许的最大字符串。处理此通知使控件在任何时候都能正确地显示输出，并减少控件显示范围内的闪烁。有关此通知的更多信息，请参见 5.21.4 小节。
- DTN_USERSTRING: 通知父级用户已编辑完日期和时间选择器控件的内容。只有设置了 DTS_APPCANPARSE 样式时才发送此通知。
- DTN_WMKEYDOWN: 用户在回调字段中输入时通知父级。处理此通知以便为日期和时间选择器控件中所支持的非回调字段模拟相同的键盘响应。

5.22 Visual C++ 2010 Month Canlendar 控件简介与开发

月历（Month Calendar）控件（CMonthCalCtrl）可实现月历式样的用户界面，这给用户提供了非常直观和可识别的输入或选择日期的方法，此控件还为应用程序提供使用现有数据类型获取和设置控件中日期信息的方法。默认情况下，月历控件显示当前的日和月，但是用户可以滚动到前一个月和下一个月及选择特定的月和/或年。

若要在项目中使用 CMonthCalCtrl 类，则必须包括 AFXDTCTL.h（通常在 STDAFX.h 中）。

5.22.1　创建月历控件

创建月历控件的方式取决于是在对话框中使用该控件还是在非对话框窗口中使用此控件。

1．在对话框中直接使用 CMonthCalCtrl

（1）在对话框编辑器中，将 Month Calendar Control 添加到对话框模板资源，指定其控件 ID。

（2）使用月历控件的“属性”面板指定任何所需样式。

（3）使用添加成员变量向导添加带 Control 属性的 CMonthCalCtrl 类型的成员变量，可以使用该成员调用 CmonthCalCtrl 的成员函数。

（4）对于需要处理的任何月历控件通知消息，使用“属性”面板来映射对话框类中的处理函数。

（5）在 OnInitDialog()中，设置 CMonthCalCtrl 对象的所有其他样式。

2．在非对话框窗口中使用 CMonthCalCtrl

（1）在视图或窗口类中定义此控件。

（2）调用控件的 Create()成员函数，可能在 OnInitialUpdate ()中，也可能与父窗口的 OnCreate()处理函数一样早（如果创建此控件的子类）设置此控件的样式。

5.22.2　处理月历控件中的通知消息

用户与月历控件交互作用（选择日期和/或查看不同的月）时，控件（CMonthCalCtrl）将通知消息发送给其父窗口（通常为视图或对话框对象）。如果要进行某种响应，请处理这些消息。例如，用户选择新月进行查看时，可以提供一组应强调的日期。

使用“属性”面板为要实现的那些消息的父类添加通知处理程序。

下面描述月历控件发送的各种通知。

- MCN_GETDAYSTATE: 请求有关哪些天应以粗体显示的信息。有关处理该通知的信息，请参见 5.22.3 小节。
- MCN_SELCHANGE: 通知父级选定的日期或日期范围已经更改。
- MCN_SELECT: 通知父级已完成显式日期选定。

5.22.3　设置月历控件的日状态

月历控件中的一个属性是可以存储月中每一天的信息（称为控件的日状态），该信息用于强调当前显示的月的某些日期。

CMonthCalCtrl 对象必须具有 MCS_DAYSTATE 样式才能显示日状态信息。

日状态信息表示为 32 位数据类型，MONTHDAYSTATE。MONTHDAYSTATE 位域（1～31）中的每一位表示一月中一天的状态。如果某个位打开，相应的日期将用粗体显示；否则，日期将以无强调的方式显示。

有两种方法用于设置月历控件的日状态：通过调用 CMonthCalCtrl::SetDayState()显式地设置，或通过处理 MCN_GETDAYSTATE 通知消息来设置。

1．处理 MCN_GETDAYSTATE 通知消息

控件发送 MCN_GETDAYSTATE 消息来确定可见月内日应显示的方式。

由于控件缓存前面和后面的月（相对于可见月），因此每次选择新月都将收到该通知。

若要正确处理该消息，必须确定正在请求多少个月的日状态信息，用适当的值初始化 MONTHDAYSTATE 结构数组并用新信息初始化相关结构成员。

下面的过程（详细说明了必要步骤）假设有一个名为 m_monthcal 的 CMonthCalCtrl 对象和一个 MONTHDAYSTATE 对象数组 mdState。

2. 处理 MCN_GETDAYSTATE 通知消息

- 使用“属性”面板将 MCN_GETDAYSTATE 消息的通知处理程序添加到 m_monthcal 对象。
- 在处理程序体中添加下列代码:

```
NMDAYSTATE* pDayState= (NMDAYSTATE *)pNMHDR;

#define BOLDDAY(ds,iDay) if(iDay>0 && iDay<32)\
                                (ds)|=(0x00000001<<(iDay-1))
MONTHDAYSTATE mdState[3];
INT i, iMax;
```

该示例将 pNMHDR 指针转换为适当的类型，并定义一个有帮助的宏在 MONTHDAYSTATE 位域中设置特定的位。

- 添加了前面的代码示例后，添加下面的代码:

```
iMax=pDayState->cDayState;

for(i=0;i<iMax;i++)
{
    mdState[i] = (MONTHDAYSTATE)0;
    BOLDDAY(mdState[i],15);
 }
pDayState->prgDayState = mdState;
```

该示例首先确定正在请求多少个月的信息（pDayState->cDayState）。对于每一个月，当前位域（mdState）都初始化为 0，然后设置所需的日期（这种情况下，为每月的 15 日），最后设置 pDayState->prgDayState 成员，完成 MCN_DAYSTATE 处理程序。

5.23 Visual C++ 2010 Custom 控件简介与开发

对话框编辑器使开发人员可以在对话框模板中使用现有的“自定义”或“用户”控件。

Custom Control，即自定义控件，是指一种特殊格式的动态链接库（DLL）或者一个文件，它能够给基于 Windows 的应用程序用户接口添加特性和功能。自定义控件可以是现有的 Windows 对话框控件（例如，文本框适合与画笔和数字化输入板一起使用），也可以是全新类别的控件。

不能将这种意义上的自定义控件与 ActiveX 控件混淆。ActiveX 控件有时称为 OLE 自定义控件。此外，也不要将这些控件与 Windows 中的所有者描述控件混淆。

此功能旨在使开发人员可以使用 Windows 所提供控件以外的控件。在运行时，控件与窗口类（与 C++ 类不同）关联。完成该任务的一个更常用的方法是在对话框中安装任何控件，如静态控件；然后，

在运行时，在 OnInitDialog()函数中移除该控件并用自己的自定义控件替换它。

现在，建议在大多数情况下编写 ActiveX 控件或创建 Windows 公共控件的子类。

对于这些自定义控件，仅限于：

（1）设置对话框中的位置；

（2）输入标题；

（3）标识控件的 Windows 类的名称（应用程序代码必须使用该名称注册控件）；

（4）输入设置控件样式的 32 位十六进制值；

（5）设置扩展样式。

通过将工具箱中的“Custom Control”拖到对话框窗体，可以将自定义控件添加到对话框中。

MFC 中定制自定义控件的 3 种方法：拥有者绘制控件（Owner Draw Control and Menu）、自绘制控件（Self Draw Control and Menu）和子集化（Subclass）。

5.23.1　使用 MFC 方法定制控件必备的几个基本概念

控制定制风格的控件的几个标准 Windows 消息分别是：

```
WM_MEASUREITEM
WM_COMPAREITEM
WM_DELETEITEM
WM_DRAWITEM
```

1. WM_MEASUREITEM

当创建自绘制风格的控件（Owner-Drawn Button、Combo Box、List Box、List View Control，or Menu Item）时，控件的父窗体将收到这个消息用于定制控件的大小：

```
WM_MEASUREITE
MidCtl = (UINT) wParam;                          //控件的标识
lpmis   = (LPMEASUREITEMSTRUCT) lParam;          //控件的大小信息
//消息处理的返回值: 当函数处理了这个消息时，必须返回 True。
```

2. WM_COMPAREITEM

系统发送此消息用于设定一个需要排序的控件（如具有 CBS_SORT 风格的 Combo Box、有 LBS_SORT 风格的 List Box）新插入项的位置信息

```
WM_COMPAREITEM
idCtl = wParam;                                  // 控件标识
lpcis = (LPCOMPAREITEMSTRUCT) lParam;            // 两个进行比较的子项信息
```

消息处理的返回值介绍如下。

-1：子项 1 在子项 2 之前的位置

0：子项 1 和子项 2　具有相等的排序位置

1：子项 1 在子项 2 之后的位置

3. WM_DELETEITEM

当 List Box 或 Combo Box 被销毁或它们的某一子项将被除去时（如消息 LB_DELETESTRING TCONTENT、CB_DELETESTRING、CB_RESETCONTENT），系统会对应每一个被删除的控件子项发送消息给控件的父窗体。

```
WM_DELETEITEM
idCtl = wParam;                                  //控件标识
```

```
lpdis = (LPDELETEITEMSTRUCT) lParam;        //删除子项信息
```

消息返回值：当函数处理了这个消息时，必须返回 True。

4. WM_DRAWITEM

当自绘制风格的 Button、Combo Box、List Box、or Menu 的视觉方式需要改变时，将发送 WM_DRAWITEM 消息给所有者窗体。

```
WM_DRAWITEM
idCtl = (UINT) wParam;                      //控件标识
lpdis = (LPDRAWITEMSTRUCT) lParam;          //绘制控件的信息
```

函数返回值：当函数处理了这个消息时，必须返回 True。

5.23.2 定制自定义控件的 3 种常见方法

1. 拥有者绘制控件或菜单

Windows 支持拥有 WS_OWNERDRAW 风格的控件发送指定消息给控件的父窗体（控件或是菜单的拥有者），使得父窗体可以定制这些控件的视觉风格或行为。

MFC 在其消息路由中直接支持 4 种消息的处理：

```
CWnd::OnDrawItem()
CWnd::OnMeasureItem ()
CWnd::OnCompareItem ()
CWnd::OnDeleteItem ()
```

可以在 CWnd 的派生类（通常是 CMainFrame 或是 CDialog）中实现这些方法来定制控件。

这种实现方法有很大的弊端，代码重用率低，这样定制的控件如果在另一个地方重用，只能把代码从一个地方复制到另一个地方。

2. 自绘制的控件或是菜单

MFC 默认的实现自绘制标准消息的方法，将这些本来由父窗体实现的绘制的工作消息解码发送到指定控件，由这些控件来处理这些消息，这种优雅的处理方式使实现重用率很高的自定义风格的控件变得很容易。

在这些封装控件的 MFC 类（CMenu、CButton、CListBox、CListCtrl）中，只需要派生一个新类并重写对应的虚函数就可以轻松定制自定义风格的控件，如 CMenu 的 DrawItem()和 MesureItem()函数。

3. 动态子集化

子集化在 Windows 编程中是指用一个新的窗口过程取代旧的窗口过程，而是用旧的窗体过程作为默认的窗口处理，来使得窗体呈现新的特性。

superclass 和 subclass 在 Windows 编程中的概念可以用 C++中的基类和派生类的关系来理解。

在 Win32 中，这个过程可以用 API: SetWindowLong()来实现。

5.24 Visual C++ 2010 SysLink 控件简介与开发

若要添加 SysLink 控件，请添加一个自定义控件，然后将该控件的 Class 属性更改为 SysLink，这将导致属性刷新并显示 SysLink 控件属性。

SysLink 控件是由 CLinkCtrl 类实现的，CLinkCtrl 类提供了普通 SysLink 控件的功能。CLinkCtrl 类是由 CWnd 类继承而来的，如下：

```
class CLinkCtrl : public CWnd
```

SysLink 控件是一种 Windows XP 控件，不受早期版本的 Visual Studio 支持，如图 5.8 所示，它可以方便地在窗口中嵌入超文本链接。

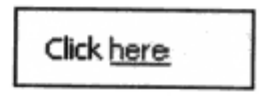


图 5.8　在窗口中嵌入超文本链接

本节将介绍 SysLink 控件的编程要素信息。

SysLink 控件为将超文本链接嵌入在窗体中提供了一种便捷方式，它的主要消息简介如下。

- LM_GETIDEALHEIGHT：捕捉到与控件的当前宽度相匹配的最合适的高度。
- LM_GETIDEALSIZE：捕捉到与控件的当前宽度相匹配的最合适的大小。
- LM_GETITEM：捕捉到一个项的状态和属性。
- LM_HITTEST：用于判定用户是否点击了特定的链接。
- LM_SETITEM：设置一个项的状态和属性。

SysLink 控件的结构如下。

- LHITTESTINFO：用于获取某一给定位置的链接的信息。
- LITEM：用于设置和恢复某一个链接项的信息。
- NMLINK：包含通知消息。发送这个结构消息时，一般都伴随了 NM_CLICK 和 NM_RETURN 消息。

5.25　Visual C++ 2010 Split Button 控件简介与开发

拆分按钮 Split Button 是一种很常用的功能控件，可以在 Button 的一侧显示下拉按钮，单击该下拉按钮后展示出分级的功能。它可以增强控件的表达能力，节约界面的空间，使界面更加人性化，是美工与功能结合的一样做法。在默认的情况下，系统不提供这功能，需要用户自定义。

Split Button 是创建具有默认操作和一列选择性选项的按钮。拆分按钮最好的示例之一是 Office2007 中“开始”选项卡的“剪贴板”组中的“粘贴”按钮。

在 Windows 的规范中，将这种按钮称为分割按钮。如果用鼠标指针指向某个按钮时，该按钮变为两个部分，则这个按钮是一个分割按钮。单击该按钮的主要部分会执行一个命令，而单击下拉按钮则会打开一个有更多选项的菜单。

Split Button 控件如图 5.9 所示。Split Button 是一种 Windows Vista 按钮，它有一个默认操作，但也可以通过一个巧妙的 UI 手势来访问其他可能的操作，该手势会显示一个具有更多操作的下拉菜单。Split Button 控件实质上是一种具有 BS_SPLITBUTTON 风格标记集的按钮控件。

图 5.9　拆分按钮控件

分割按钮的特性如下。

（1）有多个功能，但是某一个功能是最重要的或是最常用的。

（2）单击两个部分，相应操作不相同。

对于分割按钮在软件中的应用，Windows 的规范很好地定义了其使用原则，在 Web 中分割按钮用法有以下几点。

（1）默认状态：符合网站风格的默认状态，是按钮还是文本要看网站的风格定义。

（2）鼠标经过时要有不同表现：经过文本时文本应该发生变化，经过下拉按钮时下拉按钮也应该

发生变化。至于文本的变化是下划线还是背景色，要根据网站风格定义。

（3）文本和下拉按钮的不同响应：文本相应的是分割按钮的默认动作，而下拉按钮是选择其他动作，它们的响应本就不相同。

（4）提示信息：当鼠标指针在两者之上时，应该有不同的提示信息，告诉用户这个按钮引发的操作是什么。

5.26 Visual C++ 2010 Network Address 控件简介与开发

Network Address 控件，如图 5.10 所示，是一种简单的 Windows Vista 编辑框，负责验证输入，将输入限定为 IP v4 地址、IP v6 地址或友好的 DNS 名称。

图 5.10 网络地址控件

5.27 Visual C++ 2010 Check Box 控件简介与开发

Check Box 控件，即复选框控件，可同时选中多个复选框来表示用户可以选择的一个或几个选项。复选框控件的主要属性是 Value，关闭复选框时其值为 0，打开复选框时其值为 1。复选框是个触发/切换开关，每次单击都改变原来的状态（从选中变取消选中或者相反）。

使用应用程序时，经常要按预先给定的条件进行输入、统计、查询和显示等操作，如果条件组中的各个条件之间是可以组合的，可使用复选框。当程序界面上不止一个复选条件组时，尽管组与组之间不会产生什么影响，但从界面美观的角度出发，也应使用框架将组与组区分开来。

复选框在程序运行时，其外观有两种形式：一种是选中状态，另一种是未选中状态。如图 5.11 所示，“必修课程”复选框处于选中状态，“选修课程”复选框处于未选中状态。

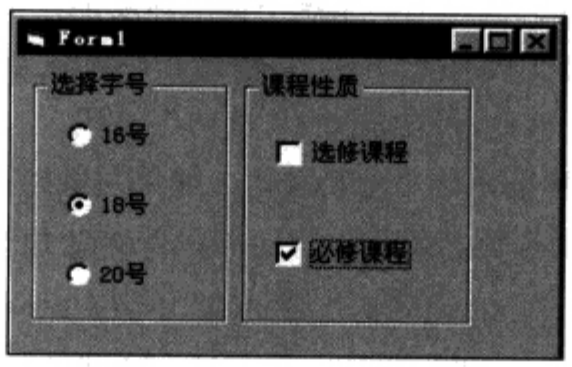


图 5.11 使用复选框的程序运行结果

（1）Caption 属性：用于设置复选框中的文字内容。图 5.11 中的“必修课程”和“选修课程”等文字内容就分别是通过两个复选框的 Caption 属性进行设置的，即分别是两个复选框的 Caption 属性值。

（2）Alignment 属性：用于设置复选框中文字的对齐方式。

（3）Style、DisabledPicture、DownPicture 和 Picture 属性：作用和使用方法与命令按钮相同。

（4）Value 属性：用于判断复选框是否处于被选中状态。

除上述属性以外，复选框还有许多其他属性，其作用和用法可参考其他控件的同名属性。

复选框的常用方法有 Move()、Refresh()和 Setfocus()，其调用方法可参考单选按钮对象的同名方法。

复选框最常用的事件为 Click 事件，程序运行时，单击复选框将引发该事件。单击复选框时，如果该复选框原来处于未选中状态，单击后将变成选中状态，同组的其他复选框仍保持原来的状态；如果该按钮原来处于选中状态，单击后将变成未选中状态，同组的其他复选框仍保持原来的状态。注意复选框和单选按钮的区别。

下面将讲解复选框的应用：使用复选框控件统计信息。

在应用程序中经常会使用复选框来统计信息，这是因为复选框操作简单，用户只需选中要选择的信息即可。首先调用 GetCheck()方法获得控件的选中状态。

语法格式如下：

```
int GetCheck( ) const;
```

然后调用 GetWindowText()函数获得复选框控件的显示信息。

步骤如下。

（1）创建一个基于对话框的应用程序，将对话框的 Caption 属性修改为“使用复选框控件统计信息”。

（2）向对话框中添加两个静态文本控件、两个编辑框控件、两个群组框控件、8 个复选框控件和一个按钮控件。

（3）在对话框的 OnInitDialog()函数中设置“语文”和“数学”两个复选框被选中。代码如下：

```
m_Chinese.EnableWindow(False);                    //设置“语文”复选框不可用
m_Chinese.SetCheck(1);                            //设置“语文”复选框选中
m_Arith.EnableWindow(False);                      //设置“数学”复选框不可用
m_Arith.SetCheck(1);                              //设置“数学”复选框选中
```

（4）处理“提交”按钮的单击事件，在该事件的处理函数中获得控件中的显示信息，并将获得的信息显示在消息框中。代码如下：

```
void CCountCheckDlg::OnButrefer()                 //“提交”按钮单击事件处理函数
{
    CString ID,Name;                              //声明字符串变量来保存编辑框文本
    GetDlgItem(IDC_EDIT1)->GetWindowText(ID);     //获得学号
    GetDlgItem(IDC_EDIT1)->GetWindowText(Name);   //获得姓名
    CString str,text;                             //声明字符串变量
    str = "学号：" + ID + "姓名：" + Name + "\r\n";  //设置字符串
    str += "必修科目：语文、数学\r\n 选修科目：";    //设置字符串
    for(int i=0;i<6;i++)                          //根据选修科目循环
    {
        CButton* but = (CButton*)GetDlgItem(IDC_CHECK3+i);  //设置指向复选框的指针
        if(but->GetCheck()==1)                    //判断复选框是否选中
        {
            but->GetWindowText(text);             //获得复选框的显示信息
            str += text + "、";                   //设置字符串
            str = str.Left(str.GetLength()-2);    //去掉字符串末尾的顿号
        }
    }
    MessageBox(str);                              //显示信息
}
```

运行结果如图 5.12 所示。

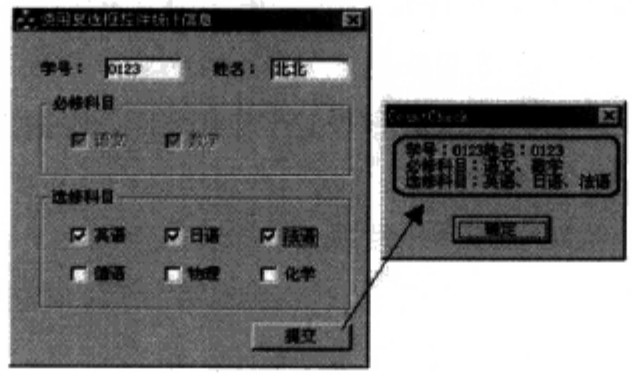


图 5.12　获取复选框输入的运行结果

5.28　Visual C++ 2010 Radio Button 控件简介与开发

单选按钮（Radio Button）控件是成组出现的，用户只能选择其中一个。单选按钮的主要属性是 Value，选定控件时其值为 True，否则为 False。单选按钮也属于触发/切换开关，每次单击时都改变状态（从选中变为取消选中或者相反）。

单选按钮控件也属于按钮的一种，可以分组使用。使用单选按钮控件同样可以简化用户的操作。本节将简单地介绍单选按钮控件的应用。

5.28.1　为单选按钮控件分组

在使用单选按钮时，有时会因为不同的需要把单选按钮分为几组，使每一组中只有一个处于选中状态。在默认情况下，所有的单选按钮都被视为一组。要为单选按钮分组，可以在属性面板中选择“General”选项卡中的 Group 属性，以【Tab】键顺序为基础。Group 属性是设置控件的群组关系的属性，为一个单选按钮选择了 Group 属性，以【Tab】键顺序为准，在这个单选按钮以后的没有选择该属性的单选按钮都划分为一组，而分为一组的单选按钮可以共用一个成员变量。

5.28.2　获得被选中的单选按钮的文本

要获得单选按钮中的文本，可以使用 GetWindowText()函数，只是在使用之前需要确定被选中的是哪个单选按钮，可以通过单选按钮的单击事件确定是哪个按钮被选中。

下面通过一个示例来讲解应该如何通过 GetWindowText()函数获得单选按钮中的数据。

步骤如下。

（1）创建一个基于对话框的应用程序，将对话框的 Caption 属性修改为“获得被选择的单选按钮的文本”。

（2）向对话框中添加 4 个单选按钮控件和一个按钮控件，为单选按钮分组，并关联一个整型变量 m_Radio。

（3）为 4 个单选按钮添加单击事件处理代码如下：

```
void CGetRadioDlg::OnRadio1()              //第一个单选按钮的单击事件处理函数
{
    m_Radio = 1;   //为变量赋值
}
void CGetRadioDlg::OnRadio2()              //第二个单选按钮的单击事件处理函数
```

```
{
    m_Radio = 2;    //为变量赋值
}
void CGetRadioDlg::OnRadio3()              //第三个单选按钮的单击事件处理函数
{
    m_Radio = 3;    //为变量赋值
}

void CGetRadioDlg::OnRadio4()              //第四个单选按钮的单击事件处理函数
{
    m_Radio = 4;    //为变量赋值
}
```

（4）处理“确定”按钮的单击事件，在该事件中获得当前选中的单选按钮的文本，并通过消息框将其显示出来。代码如下：

```
void CGetRadioDlg::OnButtonok()                    //“确定”按钮单击事件
{
    CString str;                                   //声明字符串变量
    CButton* Radiobutton = (CButton*)GetDlgItem
    (IDC_RADIO1+m_Radio-1);                        //计算选中的单选按钮
    Radiobutton->GetWindowText(str);               //获得单选按钮的文本
    MessageBox(str);                               //显示单选按钮中的文本
}
```

实例的运行结果如图 5.13 所示。

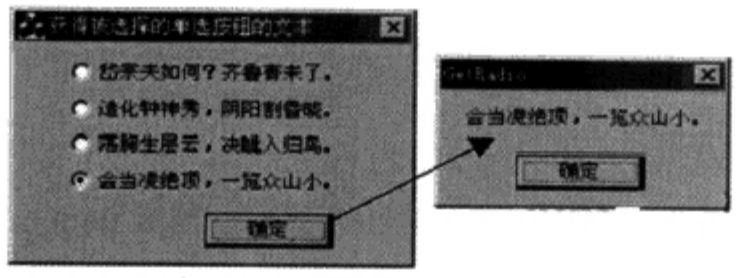


图 5.13　获取单选框输入的运行结果

5.29　Visual C++ 2010 MediaPlayer 控件简介与开发

MediaPlayer 控件是一种 ActiveX 控件。ActiveX 控件可以简单地理解为能够实现特定功能的控件，例如可以浏览网页的控件、可以播放音频与视频的控件、可以播放 Flash 动画的控件等。ActiveX 控件是一种可重用的软件组件，由编程语言开发，开发 ActiveX 控件可以使用各种编程语言，如 C、C++，当然也包括 Visual Studio.NET 环境的 VB.NET、VC.NET、C#等编程语言。ActiveX 控件目前仅适用于 Windows 平台。

ActiveX 控件一旦被开发出来，即和开发时所使用的编程语言无关。

ActiveX 控件通常保存在.ocx 或.dll 文件中。ActiveX 控件不能单独运行，必须依赖于某种应用程序，如 Windows 应用程序、Web 应用程序等，这些程序称为 ActiveX 控件的宿主程序。

默认状态下，工具箱并不包含 ActiveX 控件，要使用 ActiveX 控件，需要向工具箱中添加，也可以从工具箱中移除对不需要的 ActiveX 控件，添加与移除 ActiveX 控件需要通过“自定义工具箱”对话框实现。在工具箱中右击，选择“添加/移除项”命令，或选择“工具”→“添加/移除工具箱项”命

令，打开“自定义工具箱”对话框。

Windows Media Player 控件可以实现多种音频与视频格式文件的播放，可以通过设置控件的 URL 属性值来获取媒体播放文件。URL 属性是字符串类型，用于存储播放文件的名称与路径，控件从该属性中获取播放文件，假设有一个 Windows Media Player 的控件对象 axMedia，则设置 URL 的语句如下：

```
axMedia.URL="f:\\C#\\MP3_WMV\\贝多芬-致爱丽丝.mp3";
```

上述语句表示从本地硬盘获取播放文件。

Media Player 控件的具体创建方法如下。

（1）建立一个对话框 IDD_PLAYER_DIALOG。

（2）在对话框上右击，选择添加类，基类设置为 CDialog，类名为 CPlayer，单击确定按钮生成 CPlayer 类。

（3）在对话框上右击，选择插入 Activex 控件，选中 Windows Media Player 对应的复选框，单击确定按钮，控件被加入。这时可以在对话框中看到一个 Media Player 播放器。

（4）关闭资源视图。

（5）选择“Project”→“Add Class”命令，在模板中选择“在 ActiveX 控件中的 MFC 类”选项，在“可用的 ActiveX 控件”下拉列表框中选择“Windows Media Player”选项，在左下方的接口栏中显示可以选择的类的接口，选择需要的接口即可，这里选择 IWMPPlyaer4，单击完成按钮生成类 CWMPPlayer4。

（6）打开资源视图，在对话框的 Windows Media Player 控件上右击，添加变量，变量类型设置为 CWMPPlayer4，变量名 m_ctrlPlayer。

（7）调用 m_ctrlPlayer.put_url()可以播放媒体文件。

5.30 小结

本章主要讲解了在 MFC 开发中所应用的各种控件开发，包括 Button、List Box、Combo Box、List Control、Edit Control、Rich Edit Control、Progress Control、Tree Control、DBgrid Control、Message Control、Timer Control、Tab Control、IP Control、Picture Control、Slider Control、Scroll Control、Hot Key Control、Animation Control、Spin Control、GroupBox、Date Time Picker、Month Calendar、Custom Control、SysLink Control、Split Button、Network Address、Check Box、Radio Button、Media Player Control 等控件。在完成本章的学习之后，读者应该：

（1）掌握各种控件的功能、作用；

（2）掌握各种控件的创建方法；

（3）掌握各种控件可应用于什么应用程序，可以起到什么作用；

（4）能够熟练运用本章所学的控件开发应用程序。

第二部分　Visual C++ 2010 下 MFC 开发

第 6 章　计算机测控系统概述

6.1　Visual C++ 2010 SDI 开发简介

创建 SDI 应用程序时，应该首先利用 MFC AppWizard 生成一个简单的 SDI 应用程序框架，然后再利用 MFC Class Wizard 工具对其进行进一步丰富，使其可以在视图中绘制直线，并能够将所绘制的直线保存到文档中，以后可以由此应用程序打开这些文档并进行进一步的修改。

6.1.1　建立应用程序基本框架

利用 MFC AppWizard 建立应用程序基本框架的详细步骤可参见第 2 章中的介绍。MFC AppWizard 可以为应用程序生成开始文件和项目文件，但是 MFC AppWizard 生成 SDI 应用程序的基本边框时，只派生文档类、视图类、主边框窗口类和应用程序类。

以第 2 章生成的 HelloWorld 为例，HelloWorld 的文档类是 ChelloWorldDoc，头文件为 HelloWorld.h，它定义了该类的结构。成员函数的实现则在文件 HelloWorldDoc.cpp 中。要使文档能够处理更多的数据类型，可以修改类结构所在的.h 文件和类的成员函数所在的.cpp 文件，分别向这两个文件中添加更多的类定义及其实现。文档数据的更新操作（由串行化功能实现）的代码也位于此类的实现文件中。

HelloWorld 的视图类为 CHelloWorld，头文件为 HelloWorldView.h，实现文件为 HelloWorld-View.cpp。具体绘制图形的实现部分就在此类中实现。

HelloWorld 的主边框窗口类为 CMainFrame，是从 CFrameWnd 类派生的，头文件为 MainFrm.h，实现文件为 MainFrm.cpp。如果需要修改窗口风格，则修改此类的相关文件即可。

HelloWorld 的应用程序类为 CHelloWorldApp，头文件为 HelloWorld.h，实现文件为 Helloworld.cpp。

6.1.2　处理视图

1．在视图类中添加数据成员

在视图类中可以添加新的数据成员来管理在视图中绘制时的状态，如光标位置及状态，具体方法介绍如下。

（1）打开 HelloWorld.h 文件，将以下代码添加到 CHelloWorldView 类的定义中：

```
protected:
    int m_Draw;              //表示是否正在拖动鼠标
    HCURSOR m_HCursor;       //存放光标句柄
    CPoint m_pOld;           //存放光标的当前位置
    CPoint m_pOrigin;        //存放单击鼠标左键时光标所在的位置（画线的起点）
```

（2）打开文件 HelloWorldView.cpp，将以下代码添加到 CHelloWorldView 类的构造函数中，来初始化数据成员 m_Draw 和 m_HCursor。

```
m_Draw=0;     //初始时，未拖动鼠标
m_HCursor=AfxGetApp()->LoadStandardCursor(IDC_CROSS);
```

其中，AfxGetApp()返回一个指向应用程序对象的指针，通过该指针调用 CWinApp 的成员函数 LoadStandardCursor()来得到十字型光标。

2．用 MFC Class Wizard 映射鼠标消息

使用 MFC Class Wizard 来映射鼠标消息的方法如下。

（1）选择“Project”→“Add Class”命令，打开 MFC Add Class 对话框，双击其中的 MFC 选项，即可打开 MFC Class Wizard 对话框。

（2）在“Class name”文本框中输入要处理的类名 CHelloWorldView。

（3）在“Object ID”列表框中选择定义消息处理函数名 CHelloWorldView。

（4）在“Message type”文本框中设置要映射的鼠标消息为 WM_LBUTTONDOWN，然后单击“Add Function”按钮。

（5）重复步骤（4），映射鼠标消息为 WM_MOUSEMOVE 和 WM_LBUTTONUP。

生成消息处理函数时，MFC Class Wizard 在头文件 HelloWorldView .h 中把以下函数声明添加到类定义中：

```
//生成的消息映射函数
protected:
//APX_MSG(CHelloWorldView)
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnMouseMove(UINT nFlags, CPoint point);
afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
// AFX_MSG
DECLARE_MESSAGE_MAP()
```

在实现文件 HelloWorldView.cpp 中，MFC Class Wizard 把以下消息映射宏添加到 CHelloWorldView 类的消息映射中：

```
//AFX_MSG_MAP(CHelloWorldView)
ON_WM_LBUTTONDOWN()
ON_WM_MOUSEMOVE()
ON_WM_LBUTTONUP()
//AFX_MSG_MAP
```

此外，MFC Class Wizard 在实现文件 HelloWorldView.cpp 中插入消息处理函数的如下定义：

```
//鼠标左键按下的消息处理函数
void CHelloWorldView::OnLButtonDown(UINT nFlags, CPoint point)
{
    //TODO:Add your message handler code here and/or call default
    //TODO:在此处添加消息处理函数代码和/或调用默认代码
    CView::OnLButtonDown(nFlags, point);
}
//鼠标移动的消息处理函数
void CHelloWorldView::OnMouseMove(UINT nFlags, CPoint point)
{
    //TODO:Add your message handler code here and /or call default
    //TODO:在此处添加消息处理函数代码和/或调用默认代码
    CView::OnMouseMove(nFlags, point);
}
void CHelloWorldView::OnLButtonUp(UINT nFlags, CPoint point)
```

```
{
    //TODO:Add your message handler code here and/or call default
    //TODO:在此处添加消息处理函数代码和/或调用默认代码
    CView::OnLButtonUp(nFlags,point);
}
```

3．添加消息处理函数代码

映射鼠标消息后，接着要添加消息处理代码，以便能在视图窗口中画直线，具体方法如下。

（1）在消息处理函数 CHelloWorldView::OnLButtonDown()中添加如下代码：

```
m_pOld=point;           //存放鼠标指针的当前位置
m_pOrigin=point;        //存放画线的起始位置
SetCapture();           //捕获鼠标指针
m_Draw=1;               //表示正在拖动鼠标
RECT rect;
GetClientRect(&rect);   //获取视图窗口客户区的坐标
ClientToScreen(&rect);  //转换客户区坐标为屏幕坐标
ClipCursor(&rect);      //将光标限定在视图窗口客户区，接着调用 CWnd 的成员函数
ClipCursor(&rect);      //SetCapture()使随后的鼠标消息都被送往视图窗口
```

如果要释放鼠标并恢复正常输入，可调用 Windows API 函数 ReleaseCapture()。

其中，SetCapture()函数返回指向当前窗口对象的指针，函数的原型为：

```
CWnd * SetCapture();
```

然后调用 CWnd 类的成员函数 GetClientRect()获取窗口客户区的坐标，函数的原型为：

```
void GetClientRect(LPRECT lpRect);
```

RECT 结构定义如下：

```
Typedef struct tagRECT{
    LONG left;          //左上角 X 坐标
    LONG top;           //左上角 Y 坐标
    LONG right;         //右下角 X 坐标
    LONG bottom;        //右下角 Y 坐标
}RECT;
```

调用 CWnd 的成员函数 ClientToScreen()将窗口客户区坐标转换为屏幕坐标，函数的原型为：

```
void ClientToScreen(LPRECT lpRect);
```

调用 Windows API 函数 ClipCursor()将光标限定在窗口客户区内，函数原型为：

```
void ClipCursor(LPRECT lpRect);
```

（2）在消息处理函数 CHelloWorldView::OnMouseMove()中添加如下代码：

```
SetCursor(m_HCursor);               //设置自定义光标
if(m_Draw){                         //正在拖动鼠标
    CClientDC dc(this);             //创建设备文本对象
    dc.SetROP2(R2_NOT);             //设置绘图模式，以屏幕反转色为绘图颜色
    dc.MoveTo(m_pOrigin);           //删除前一 WM_MOUSEMOVE 画的直线
    dc.LineTo(m_pOld);
    dc.MoveTo(m_pOrigin);           //从起始位置到当前位置画一条新的直线
    dc.LineTo(point);
    m_pOld=point;                   //存放鼠标光标的当前位置
}
```

（3）在消息处理函数 CHelloWorldView::OnLButtonUp()中添加如下代码：

```
if(m_Draw){
    m_Draw=0;                    //表示鼠标已经释放
    ReleaseCapture();            //释放鼠标捕获并恢复正常输入
    ClipCursor(NULL);            //使光标可以移动到屏幕上的任意位置
}
```

4．定制视图窗口

要修改视图窗口的特性，可以在 ChelloWorldView 类的成员函数 PreCreateWindow()中添加如下代码：

```
cs.lpszClass=AfxRegisterWndClass(
    CS_HREDRAW|CS_VREDRAW,//类风格
    0,                                  //不用光标
    (HBRUSH)::GetStockObject(WHITE_BRUSH),//设置白色背景
    0);//没有图标
return CView::PreCreateWindow(cs);
```

5．修改窗口标题栏

默认情况下，标题栏中显示的标题名为“HelloWorld\n\nHelloWorld\n\n\nHelloWorld.Document\nHelloWorld.Document”。若要修改标题栏，可执行以下操作。

（1）在项目工作区窗口，切换到 Resource View 选项卡。

（2）选择 String Table（串表）资源。

（3）在串编辑器中找到 IDR_MAINFRAME 项，并在后面的标题名处双击，则可以修改 Caption 属性值。

（4）在 Caption 文本框中将原有的标题栏内容“HelloWorld\n\nHelloWorld\n\n\nHelloWorld.Document\nHelloWorld.Document”改成“单文档绘图程序示例”，如图 6.1 所示。

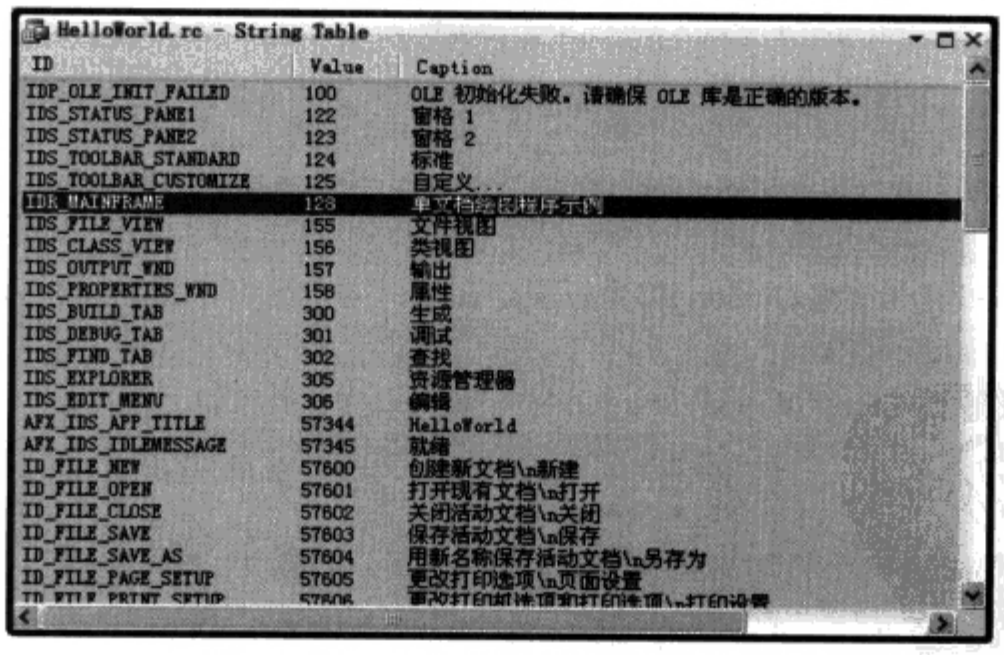


图 6.1　修改标题栏内容

（5）按【Esc】键退出。

6．建立并运行程序

完成以上修改后将项目保存，然后建立并运行应用程序，具体步骤如下：

（1）从“File”菜单中选择“Save all”命令将所有文件保存；

（2）从“Build”菜单中选择“Build HelloWorld”命令编译应用程序；

（3）从“Debug”菜单中选择“Start without Debugging”命令或者直接按【Ctrl+F5】组合键运行应用程序。

运行上面的程序后可以发现，已经能够在用户工作区绘制直线，但是还不能把所做的结果保存到文档中，这是因为这个应用程序还未对文档进行处理。要使此程序能够处理文档，必须再进一步添加新代码。下面介绍处理文档的实现方法。

6.1.3　处理文档

对于这个应用程序，由于只处理直线，而且这些直线也仅有起点和终点，而没有其他任何信息，所以要保存直线，只需保存直线的起点和终点坐标即可。当文档打开时，也只需要从文档中取得起点和终点信息，再在视图中把这些直线画出来即可实现文档数据的显示。下面介绍具体的实现过程。

1．在文档类中存放直线坐标

（1）这个程序的文档只包含直线，因此只需添加直线类，将直线类的声明添加在文档类头文件 HelloWorldDoc.h 中，具体代码如下：

```
class CMyLine:public CObject                    //声明直线类 CMyLine
{
    protected:
        int m_StartX;                           //直线起点的 X 坐标
        int m_StartY;                           //直线起点的 Y 坐标
        int m_EndX;                             //直线终点的 X 坐标
        int m_EndY;                             //直线终点的 Y 坐标
    public ;
        CMyLine(int StartX,int StartY,int EndX,int EndY) ;  //直线的构造函数
        void DrawLine(CDC *Pdc);                //直线的成员函数 DrawLine()用来在视图中绘制直线
}
```

（2）要让文档能够保存直线对象，还应该向文档类的声明中添加新的数据成员。此数据成员应该是刚刚声明的直线类对象，或指向直线类对象的指针，或指向直线类对象指针的数组。在此例中，声明一个数组类 CObArray 的对象，该对象可以用来保存指向直线类的指针，同时为文档类添加成员函数 AddLine()、GetLine()和 GetTotalLine()。这 3 个函数分别用于把一个直线对象添加到 CObArray 对象中，取得 CObArray 对象中当前位置的直线，以及取得 CObArray 对象中的直线数。在文档类头文件 HelloWorldDoc.h 中添加这些成员及成员函数，代码如下：

```
protected:
    CObArray m_ObArray;
public:
    void AddLine(int StartX,int StartY,int EndX,int EndY);
    CMyLine * GetLine(int index);
    Int GetTotalLine();
```

（3）在文档类实现文件 HelloWorldDoc.cpp 中编写新添加的成员函数的定义：

```
//直线类 CMyLine 的构造函数
CMyLine::CMyLine(int StartX,int StartY,int EndX,int EndY){
    m_StartX=StartX;
    m_StartY=StartY;
    m_EndX=EndX;
```

```
        m_EndY=EndY;
    }
```

构造函数 CMyLine()用于初始化直线的数据成员，即起点和终点坐标。

```
    //画直线函数
    void CMyLine::DrawLine(CDC *pDc){
        pDC->MoveTo(m_StartX,m_StartY);
        pDC->LineTo(m_EndX, m_EndY);
    }
```

成员函数 DrawLine()用于存入当前 CMyLine 对象中的用于画直线的坐标。

```
    //添加直线对象成员函数
    void CHelloWorldDoc::AddLine(int StartX,int StartY,int EndX,int EndY)
    {
        CMyLine * pMyLine=new CMyLine(StartX,StartY,EndX,EndY);
        MObArray.Add(pMyLine);
    }
```

成员函数 AddLine()用于创建新的 CMyLine 对象，并调用 CObArray 的成员函数 Add()向 mObArray 中添加 CMyLine 对象指针。

```
    //返回当前直线的成员函数
    CMyLine * CHelloWorldDoc::GetLine( int index)
    {
        if(index<0 || index>mObArray.GetUpperBound())
            return 0;
        else return (CMyLine * mObArray -> GetAt(index));
    }
```

成员函数 GetLine()用于返回指向 CMyLine 对象的指针。mObArray 中存放的对象指针可以通过索引值进行引用。

GetLine()首先判断参数所指明的索引值是否为有效，然后调用 CObArray 的成员函数 GetAt()得到参数指定的 CMyLine 对象并返回相应的指针。CObArray 的成员函数 GetUpperBound()返回最大的有效索引值。

```
    int CHelloWorldDoc::GetTotalLine()
    {
        return mObArray.GetSize();
    }
```

其中，成员函数 GetTotalLine()通过调用 CObArray 类的成员函数 GetSize()得到存放在 mObArray 中的直线数。

2. 画完直线后存放直线坐标

在消息处理函数 CHelloWorldView::OnLButtonUp()中添加如下代码：

```
    //释放鼠标左键后，画直线结束，应将所画直线添加到 CObArray 对象中
    CHelloWorldDoc * pDC=GetDocument();
    pDOC->AddLine(m_pOrigin.x, m_pOrigin.y, point.x, point.y);
```

3. 修改视图的 OnDraw()成员函数

向 OnDraw()成员函数中添加以下代码，以便修改视图：

```
    int index;
    index=pDOC -> GetTotalLine();
    while(index--)
```

```
pDOC->GetLine(index).DrawLine(pDOC);
```

以上代码调用 GetTotalLine()函数得到存放在文档对象中的直线数，然后依次画出各条直线。

4．删除文档数据

CDocument 类的所有派生类均应重载虚函数 DeleteContents()，以便新建文档或两次打开文档时清除当前文档对象的所有数据。本例在头文件 HelloWorldDoc.h 的文档类 CHelloWorldDoc 的声明中添加以下代码来实现从文档中删除已添加的全部直线。

```
virtual void DeleteContents();
```

在文档类的实现文件 HelloWorldDoc.cpp 中添加以下代码：

```
void CHelloWorldDoc::DeleteContents()
{
    int index;
    index=m_ObArray.GetSize();
    while(index)
        delete m_ObArray.GetAt(index--);
    m_ObArray.RemoveAll();
}
```

DeleteContents()首先调用 CObArray 的成员函数 GetSize()得到存放在 m_ObArray 中的直线数；然后调用 CObArray 的成员函数 GetAt()得到每个指针并用 delete 操作符删除相应的 CMyLine 对象；最后调用 CObArray 的成员函数 RemoveAll()删除所有的指针。

5．处理“Undo”命令

如果要使用“Edit”菜单的“Undo”命令撤销最后画的一条直线，首先应该将“Undo”命令映射到相应的消息处理函数。可以使用 ClassWizard 来映射“Undo”命令消息，方法如下：

（1）从“Project”菜单中选择“Add Class”命令，弹出 MFC Add Class 对话框，从中双击 MFC，则弹出 MFC Class Wizard 对话框；

（2）在“Class name”文本框中设置要处理的类名 ChelloWorldDoc；

（3）在“Object ID”列表框中选择标识符 ID_EDIT_UNDO；

（4）在“Message type”列表框中选择 COMMAND，单击“Add Function”按钮生成消息处理函数；

（5）重复步骤（4），生成消息 UPDATE_COMMAND_UI 的处理函数。

在生成消息处理函数时，MFC Class Wizard 在文档类的头文件 HelloWorldDoc.h 中把以下的函数原型声明添加到类定义中：

```
//生成的消息映射函数
protected:
//c(CHelloWorldDoc)
afx_msg void OnEditUndo();
afx_msg void OnUpdateEditUndo(CCmdUI * pCmdUI);
//{}AFX_MSG
DELCATE_MESSAGE_MAP()
```

在文档类的实现文件 HelloWorldDoc.cpp 中，MFC Class Wizard 把以下的消息映射宏添加到 HelloWorldDoc 类的消息映射中：

```
//{}AFX_MSG_MAP(CHelloWorldDoc)
ON_COMMAND(ID_EDIT_UNDO,OnEditUndo)
ON_UPDATE_COMMAND_UI(ID_EDIT_UNDO,OnUpdateEditUndo)
//{}AFX_MSG_MAP
```

此外，MFC Class Wizard 还在文档类的实现文件 HelloWorldDoc.cpp 中插入消息处理函数的基本定义：

```
void CHelloWorldDoc::OnEditUndo()
{
    //TODO:在此处添加命令处理代码
}

void CHelloWorldDoc::OnUpdateEditUndo(ComdUI * pCCmdUI)
{
    //TODO: 在此处添加命令处理代码

}
```

添加 CHelloWorldDoc::OnEditUndo()的代码如下：

```
int index;
index=m_ObArray.GetUpperBound();
if(index>=0)
{
    delete m_ObArray.GetAt(index);
    m_ObArray.Remove(index);
}
UpdateAllViews(index);
```

上述代码首先调用 CObArray 的成员函数 GetUpperBound()，得到添加到文档中的最后一条直线的索引值；然后调用 CObArray 的成员函数 GetAt()，得到指向 CMyLine 对象的指针，并用操作符 delete 删除相应的 CMyLine 对象；接下来调用 CObArray 的成员函数 RemoveAt()删除指针，并调用 UpdateAllViews()更新视图窗口；最后，在 CHelloWorldDoc::OnUpdateEditUndo()中添加以下代码：

```
pCCmdUI->Enable(m_ObArray.GetSize());
```

完成上述代码后，就可以使用“Build”菜单的“Build HelloWorld”命令和“Debug”菜单的“Start without Debugging”命令建立并运行应用程序，并在应用程序的视图窗口中画各种直线。

6.1.4 串行化处理

所谓文档的串行化是指在文档打开时能够自动把文档中的数据转换为文档类所支持的对象；在保存文档时能够自动将文档类对象转换为文档数据。实现文档的串行化一般要经过以下几步。

1. 添加串行化代码

重载文档基类 CDocument 的虚函数 Serialize()，以便使文档能够操纵由该类派生的文档类中的数据成员，在此例中就是向文档类实现文件中添加串行化代码：

```
void CHelloWorldDoc::Serialize(CArchive &ar)
{
    if(ar.IsStoring()){
        //TODO::add Storing Code here
        m_ObArray.Serialize(ar);
    }
    else
    {
        //TODO:add Loading Code here
        m_ObArray.Serialize(ar);
    }
```

```
}
```

2．编写 CMyLine 对象的串行化代码

由于此例的文档类 CHelloWorldDoc 只新增了一个数据成员，即由 CObArray 类的对象来表示文档所容纳的数据，而 CObArray 类的实际数据是 CMyLine 类的对象，所以最终应该重载 CMyLine 类的虚函数 Serialize()。如果 CMyLine 类声明中没有虚函数 Serialize()，则应添加该函数。总之，应编写 CMyLine 对象的串行化代码。

首先，在 CMyLine 类定义中添加两个宏 DECLARE_SERIAL 和 IMPLEMENT_SERIAL，并定义一个默认的类构造函数，以便 CObArray 的成员函数 Serialize()可以将 CMyLine 类信息存放在文件内并使用类信息动态创建 CMyLine 类的对象。在 CMyLine 类的声明中添加 DECLARE_SERIAL 宏和默认构造函数的代码如下：

```
CMyLine()
{}
DECLARE_SERIAL(CMyLine)
```

在此例中，这段代码位于文档类的头文件 CHelloWorldDoc.h 中。

在构造函数 CMyLine()之前添加 IMPLEMENT_SERIAL 宏的代码如下：

```
IMPLEMENT_SERIAL(CMyLine,CObject,1)
```

其中，CObArray 的成员函数 Serialize()将调用 CMyLine 类的 Serialize()函数进行实际的读写操作，可以分为以下两步。

（1）在 CMyLine 类中添加成员函数 Serialize()的声明：

```
virtual void Serialize(CArchive & ar);
```

（2）在文档类的实现文件 HelloWorldDoc.cpp 中添加成员函数 Serialize()的定义：

```
void CMyLine::Serialize(CArchive & ar)
{
    if( ar.IsStoring())
        ar<<(WORD)m_StartX<<(WORD)m_StartY<<(WORD)m_EndX<<(WORD)m_EndY;
    else
        ar>>(WORD)m_StartX>>(WORD)m_StartY>>(WORD)m_EndX>>(WORD)m_EndY;
}
```

（3）设置修改标志。

打开文档并读入数据时，CDocument 将修改标志设为 False。在文档数据被更改后，必须调用 CDocument 的成员函数 SetModifiedFlag()将修改标志设为 True。

在 HelloWorld 程序中，只有添加直线或执行“Undo”命令时才会更改文档数据，因此，需要分别在 AddLine()函数和 OnEditUndo()函数的最后添加以下代码：

```
SetModifiedFlag();
```

（4）建立并运行程序。

选择“Build”→“Build HelloWorld”命令和“Debug”→“Start without Debugging”命令可以建立并运行应用程序，然后可以在应用程序的视图窗口中画各种直线。选择“File”→“Save As”或“Save”命令可以保存所画的直线；选择“File”→“Open”命令可以打开文件。

6.1.5　SDI 应用程序编程思路

编写 SDI 应用程序时，首先应该正确理解本章前面几个小节所介绍的几个基本概念：主边框窗口类、视图类、文档类和文档的串行化。本小节结合前面的示例程序再对这几个概念进行进一步的阐述，

以帮助读者加深对基本概念的理解和应用。

1. 主边框窗口

在 SDI 应用程序中，主边框窗口同时也是文档的边框窗口，此窗口类 CMainFrame 的声明在 MainFrm.h 头文件中。MFC 类库中有其默认实现，如果要修改它，需要重载其虚函数 PreCreateWindow (CREATESTRUCT & cs)，此虚函数的实现代码在文件 MainFrm.cpp 中。

对于需要文档串行化支持的应用程序，应在类声明的构造函数之后添加宏 DECLARE_DYNCREATE(CMainFrame)，这是因为文档串行化需要动态建立相应的类对象。

边框窗口类的实现文件 MainFrm.cpp 中有一个消息映射，即 ON_WM_CEATE()，它处理函数 OnCreate(LPCREATESTRUCT lpCreateStruct)，该实现文件也定义了建立窗口的函数 PreCreateWindow (CREATESTRUCT & cs)。在定制主边框窗口风格及建立窗口时，执行的是默认操作，应分别修改这两个函数。在以上的例子中对这两个函数未做任何修改。

2. 视图

视图是应用程序的工作区，用户的各种输出（如文本的显示、图形的绘制等）都将由视图来表现，文档中的数据也要显示在视图中，因此定制视图是编写 SDI 应用程序的非常重要的环节。

由于视图也是窗口，因此它们也具有自己的风格。视图风格的设置通过重载虚函数 PreCreateWindow(CREATESTRUCT & cs)来实现。

打开文档后，文档数据的显示通过视图类成员函数 OnDraw()来实现，用户通过键盘、鼠标、菜单命令和工具等接口编辑文本和绘制图形的操作也由视图类来完成。文档的打印、打印设置及打印预览等操作可以看作是在打印机上显示视图，因此其功能的实现也应由视图类来完成。这些工作需要利用 MFC Class Wizard 来映射键盘、鼠标、菜单命令和工具消息，并实现相应的消息处理函数。在上面的例子中，对鼠标左键的按下与释放及鼠标的移动建立了消息映射并实现了相应的处理代码，而对其他消息的处理则使用 CView 类的相应成员函数作为默认消息处理函数。

如果要使应用程序具有动态建立视图类对象的功能，则应分别通过 DECLARE_DYNCREATE (CHelloWorldView)宏和 IMPLEMENT_DYNCREATE(ChelloWorldView, CView)宏来实现。

3. 文档

文档是基于文档的应用程序涉及的最重要的概念，也是应用程序代码编写工作量较大的一部分。文档是应用程序处理数据的集合，而文档中的数据要直观地显示给用户。例如，对于图形数据，应该将图形绘制在用户窗口中，而不是将表示图形几何特性的二进制数据显示给用户。同时，用户执行各种命令产生的各种应用数据也应能转换为文档中的二进制数据保存。此外，两者之间的转换应该是自动实现的，而不需要通过开发大量的代码来实现。MFC 类库提供了一个文档类 Cdocument，可帮助用户完成上述转换功能。

文档要包含的各种数据类型都应在文档类中声明新的数据成员，并由这些数据成员表示相应的应用数据。例如，在上面的例子中，对象数组类 CObArray 就是文档类的一个数据成员。对象数组类 CObArray 是数组类的一种，支持 CObject 对象指针，与数组非常相似，但是它可以随着所含指针元素的多少而动态地缩小或扩大，以适应元素数量的变化。

文档中所包含的数据及其直观显示由文档串行化功能来实现。文档串行化自动实现二进制数据与具体数据形式之间的转换，即打开文档时将二进制数据转换为相应类的数据成员，而保存文档时则将类的数据成员转换为二进制数据。要实现文档的串行化，需要重载 CDocument 类的虚函数 Serialize()，同时对于文档要处理的各种数据类都应该重载虚函数 Serialize()。

要实现文档的串行化，文档类及文档要包含的数据类的声明部分都应该有默认的构造函数和 DECLARE_SERIAL 宏，并在其实现部分的构造函数之前包括 IMPLEMENT_SERIAL 宏。

4．应用程序

Windows 环境下的每一个应用程序也都是一个类，一般称其为应用程序类。应用程序类继承自 CWapp 类，它们是应用程序在运行时的入口，其主要作用是维护全局性工作，如应用程序菜单的维护、应用程序进行的建立及命令行信息的分析等各方面的工作。

6.2　Visual C++ 2010 MDI 开发简介

在上一节中介绍了单文档接口（SDI）应用程序的开发步骤及相应程序的编写。SDI 应用程序一次只能查阅和编辑一个文档，本节会介绍如何编写多文档接口（MDI）程序。MDI 程序一次可以打开多个文档，每个文档在程序主窗口应用程序工作区包含的子窗口中显示或编辑。

尽管管理多个打开文档的接口看起来有些困难，但 Windows 和 MFC 提供的代码能够给予帮助。如果程序按照标准 MDI 程序模型的结构编写，则可以充分利用原有的代码，节省大量的编程工作。为了介绍编写 MDI 程序的技术，本节将介绍如何编写 HelloWorld 程序的 MDI 版本。

6.2.1　多文档接口

在创建 MDI 程序时，与 SDI 程序惟一的不同是，在“Application type”选项组中选中的不是“Single document”单选按钮，而是 Mutiple documents”单选按钮，而且选中了“Tabbed documents”复选框，如图 6.2 所示。其他的均与 SDI 应用程序的创建方法相同。

首次运行本节生成的 MDI 版的 HelloWorld 程序时，程序在子窗口打开新的空白文档（如图 6.3 所示）。如果再选择“File”→“New”或“Open”命令，则可以打开新文档或阅读已存在的文档资料。新打开的文档出现在另一个子窗口中，而不是像 SDI 程序中那样取代原来的文档。

可以用“Open”和“New”命令打开任意个文档。图 6.4 显示了打开几个文档后的程序。为了在特定文档上工作，要激活其对应的子窗口。可以通过单击激活窗口，也可以通过按【Ctrl+Tab】组合键或【Ctrl+F6】组合键（必要时可重复按）激活下一个子窗口（或按【Ctrl+Shift+Tab】组合键激活上一个子窗口），也可以通过在 Windows 菜单中单击标题栏激活特定子窗口。

可以选择“File”→“Close”命令，或者单击文档窗口右上角的关闭按钮，或者选择文档窗口的“Close”命令来关闭激活窗口的文档。“File”菜单下的“Save”、“Save as”和“Print”命令，以及“Edit”菜单下的所有命令都影响激活窗口的文档。

可以单击一个子窗口的最大化按钮，使该窗口充满程序主窗口的整个应用程序工作区；也可以单击子窗口的最小化按钮使窗口隐藏起来，变成由应用程序工作区的一个小条。可以选择 Windows 菜单的命令设置子窗口层叠（重叠）或平铺（不重叠），或设置最小化的子窗口的条（通过命令设置 Icons，最小化窗口在旧版本 Windows 中表示为图标）。

如果关闭所有的文档，则只显示“文件”、“视图”和“帮助”菜单。“文件”菜单只显示打开新的或现有的文档和退出程序的命令。

如果用“Open”命令打开已经在子窗口显示的文档，则程序只是激活现有的子窗口而不生成新窗口，这样，在 HelloWorld 中，一个文档一次只能在一个子窗口中查阅，后面将会具体介绍。也有 MDI 程序可以通过 Windows 菜单的 New Window 命令，让用户打开同一个文档的两个或多个子窗口。

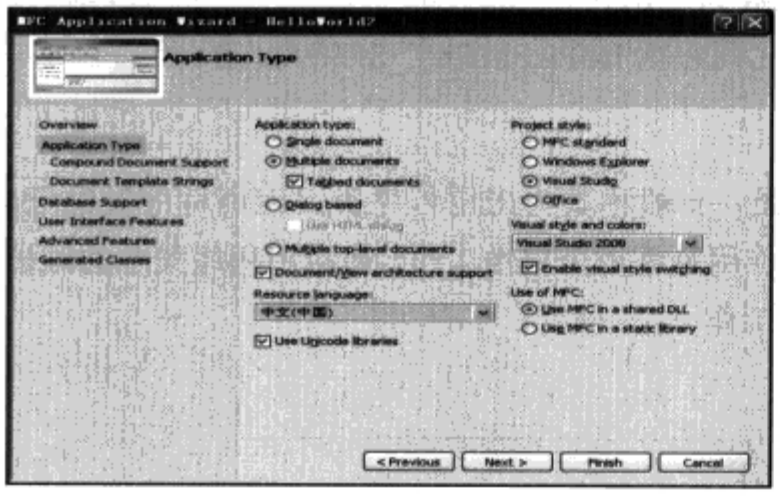


图 6.2　创建多文档应用程序

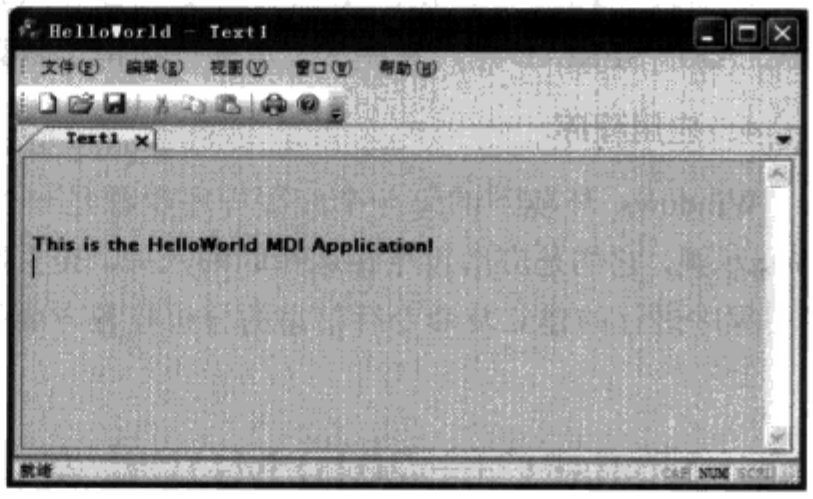


图 6.3　打开一个新文档

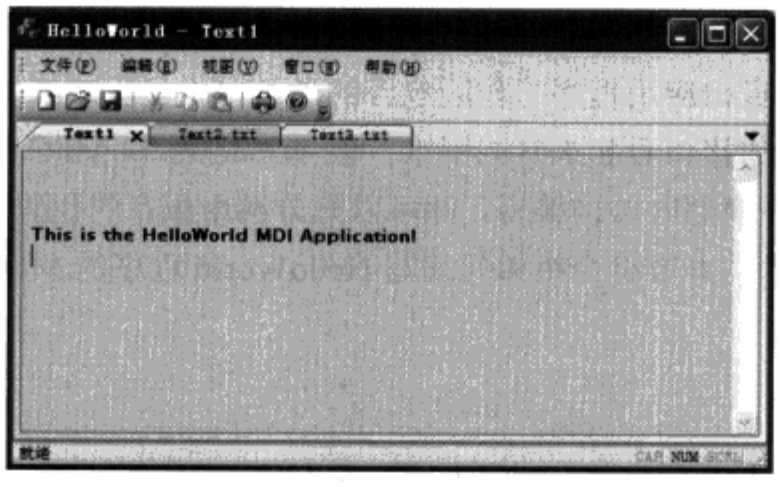


图 6.4　同时打开多个文档的结果

6.2.2　生成程序

为了生成 MDI 版的 HelloWorld 应用程序，可以按照 2 章中应用程序生成步骤中介绍的方法用应用程序向导生成新的源文件。在“New Project”对话框中，设置项目类型“MFC App Application”，在“Name”文本框中输入程序名 HelloWorld，在“Location”文本框中输入所要的项目文件夹路径；在应用程序向导各个步骤对话框中，按照 2.1.1 小节中程序选项进行设置，除了以下的选项。

（1）应用程序向导“Application Type”对话框中，选中“Multiple Documents”而不是“Single Document”单选按钮，完成的对话框如图 6.5 所示。这就是生成 MDI 而不是 SDI 应用程序所要做出的选择，它对应用程序所生成的类的代码有巨大的影响。

（2）在应用程序向导步骤 4 的对话框中，单击“Advanced”按钮，并在“Advanced Options”对话框中选择“Document Template Strings”选项卡，然后在“File extension”文本框中输入默认文件扩展名 txt；在“Doc type name”文本框中输入 Text；在“Filter name”文本框中输入 Text Files(*.txt)。所有设置如图 6.5 所示。

在应用程序向导步骤 6 的“Generated Classes”对话框中，设置“Generated classes”为 Chello WorldView；在“Base class”下拉列表框中，选择“CeditView”选项，这样，生成的 HelloWorld 程序就是从 CEidtView 而不是从 CView 派生的视图类，能在视图中自动提供完整的文本编辑功能，如图 6.6 所示。

HelloWorld 程序将“Document Template Strings”对话框中“Doc type name”文本框中输入的字符串（Text）作为新文档的默认名。第一个新文档名为 Text1，第二个为 Text2，依次类推。当然，程序用户存盘时可取不同的文件名。相应的，SDI 程序只命新文件为 Untitled（因此，生成 SDI 程序时指定文档类型名并不重要）。程序用“File extension”文本框的输入作为默认文件扩展名。

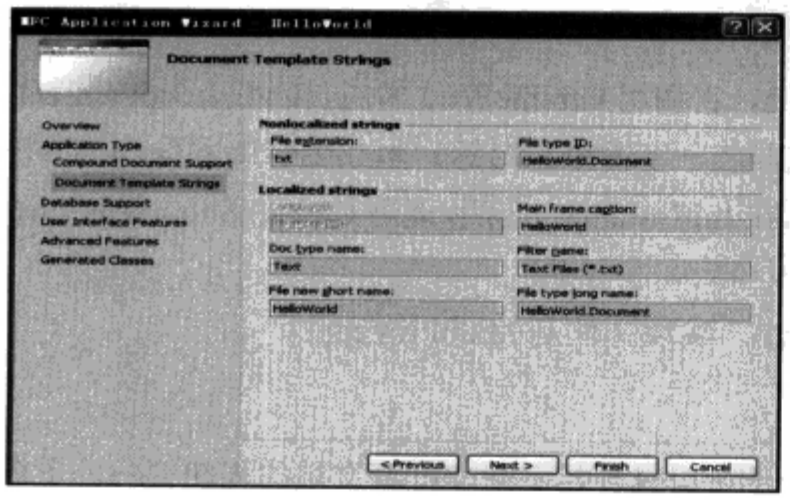


图 6.5　生成多文档程序的结果

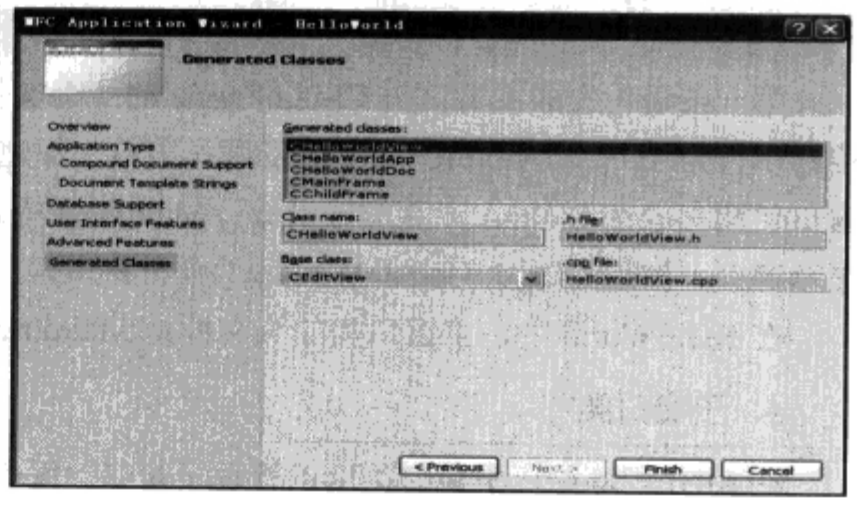


图 6.6　设置为 CEditView

6.2.3　程序类、文件和代码

应用程序向导为 MDI 程序生成的类和文件的方法与 SDI 程序的类和文件相似。与 SDI 程序相似，MDI 程序也包含应用程序类、文档类、主窗口类和视图类。但这些类的作用有所不同，MDI 程序还有另外一个类：子窗口类。

1．应用程序类

MDI 应用程序类和 SDI 程序相似，用于管理程序整体，有初始化程序的 InitInstance()成员函数。在 HelloWorld 程序中，应用程序类为 CHelloWorldApp，头文件为 HelloWorld.h，实现文件为 HelloWorld.cpp。

2．文档类

与 SDI 程序相似，MDI 应用程序的文档类存放文档数据并完成文件 I/O，但 MDI 程序对每个打开的文档生成这个类的一个实例，而不是重复使用同一个实例。在 HelloWorld 中，文档类名为 CHelloWorldDoc，头文件为 HelloWorldDoc.h，实现文件为 HelloWorldDoc.cpp。

3．主窗口类

与 SDI 程序相似，MDI 程序的主窗口类管理程序的主窗口，但它不是直接从 MFC 的 CFrameWnd 类派生，而是从它所派生的 CMDIFrameWnd 类派生。在 MDI 程序中，主窗口不是包含单视图窗口，而是用帧建立普通应用程序工作区。应用程序工作区中有每个打开文档的子窗口，每个子窗口包含一个视图窗口。

由于主窗口用帧建立普通应用程序工作区，而不是单个打开文档，它的类不包含在程序的文档模板中。由于文档首次打开时，主窗口不像 SDI 程序中那样自动生成，所以 InitInstance()函数要用下列代码明确生成并显示它：

```
//Create main MDI Frame Window

CMainFrame *pMainFrame = new CMainFrame;
if(!pMainFrame->LoadFrame(IDR_MAINFRAME))
    return FALSE;
m_pMainWnd = pMainFrame;
// …
//The main window has been initialized, so show and update it.
```

```
pMainFrame->ShowWindow(m_nCmdShow);
pMainFrame->UpdateWindow();
```

第一条语句生成主窗口类 CMainFrame 的一个实例，它调用 CframeWnd 的成员函数 LoadFrame()，用标识符为 IDR_MAINFRAME 的资源（菜单、加速器表、字符串和图标）生成主窗口本身。然后，窗口句柄存放在 CWinApp 数据成员 m_pMainWnd 中。在 InitInstance()之后，调用 CWnd::ShowWindow()显示窗口，调用 CWnd::UpdateWindow()画出窗口客户区。

在 HelloWorld 中，主窗口类的头文件是 MainFrm.h，实现文件是 MainFrm.cpp。

4．子窗口类

在 MDI 程序中，子窗口类用于管理子窗口。每个子窗口包含显示一个打开文档的视图窗口。子窗口类不用于 SDI 程序。图 6.7 显示了子窗口和 MDI 程序中生成的其他窗口。

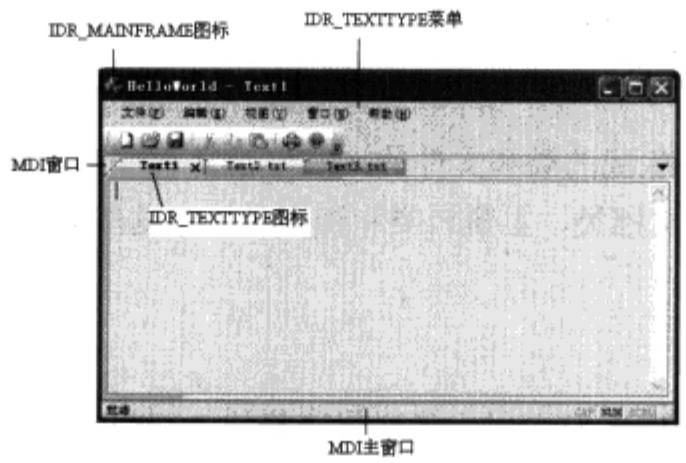


图 6.7　子窗口和 MDI 程序中生成的其他窗口

在 HelloWorld 中，子窗口类 CChildFrame 从 MFC 类 CMDIChildWnd 派生（它由 CFrameWnd 派生），其头文件为 ChildFrm.h，实现文件为 ChildFrm.cpp。

由于 CChildFrame 类用于生成并管理放置每个打开文档的子窗口，InitInstance()函数放在程序文档模板中（而不是像 SDI 程序一样包括主窗口类）：

```
//Register the application's document templates, Document
//templates serve as the connection between documents, frame
//windows and views.
CMultiDocTemplate * pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
      IDR_TEXTTYPE,
RUNTIME_CLASS(CHelloWorldDoc),
RUNTIME_CLASS(CChildFrame),   //custom MDI child frame
RUNTIME_CLASS(CHelloWorldView));
AddDocTemplate(pDocTemplate);
```

本模板属于 CMultiDocTemplate 类，它适用于 MDI 程序，而 CSingleDocTemplate 则用于 SDI 程序。模板赋予与文档相关联的资源（菜单、字符串和图标）标识符 IDR_TEXTTYPE。具体说来，打开一个或几个文档时，显示 IDR_TEXTTYPE 菜单，IDR_TEXTTYPE 字符串包含程序打开文档的默认文件扩展名和类型说明，放置文档的每个子窗口均显示 IDR_TEXTTYPE 图标。

5．视图类

MDI 程序的视图类用于生成并管理显示每个打开文档的视图窗口。每个文档的视图窗口占用文档子窗口的客户区。由于 HelloWorld 视图类从 CEditView 类派生，每个视图窗口均可作为文本编辑器。在 HelloWorld 中，视图类为 ChelloWorldView，头文件为 HelloWorldView.h，实现文件类为 HelloWorld-

View.cpp。

6．生成的代码

从 CEditView 派生视图类时，应用程序向导会生成读写文本文档的磁盘文件所需的代码，它把这个代码加入文档类的 Serialize()成员函数中。

同样，对于 HelloWorld 的多文档程序，也不必在文档类中加入 DeleteContents()成员函数。SDI 单文档程序中的 DeleteContents()函数用于响应“File”菜单的“New”命令，删除视图窗口的现有文档。对于 MDI 多文档程序，“New”命令生成新的视图窗口而不是使用现有的视图窗口，所以程序不需要删除文本。

此外，生成 MDI 多文档程序时，应用程序向导在应用程序类的成员函数 InitInstance()中自动调用 DragAcceptFiles()函数，可用于从 Windows 文件管理器或者资源管理器拖动文件并放到程序窗口来打开文件。应用程序向导还在 InitInstance()中自动加入 EnableShellOpen()和 RegisterShellFileTypes()函数的调用。通常，调用这些函数使用户可以双击具有程序默认文件扩展名的文件对象，从而打开文件，但 HelloWorld 的默认扩展名.txt 通常已经注册在其他程序（如 Windows Notepad.txt）程序中，因此，单击这个扩展名的文件可能不会运行 HelloWorld 程序。

6.2.4　自定义资源

本节要自定义 HelloWorld 资源，即修改菜单，加入加速键和提供自定义图标。首先，确保 HelloWorld 项目已经打开，单击项目工作区窗口的“Resource View”标签显示资源。

扩展 Resource View 的 Menu 分支，注意，由于 HelloWorld 现在是 MDI 版，Menu 分支出现两个菜单标识符 IDR_MAINFRAME 和 IDR_TEXTTYPE。IDR_MAINFRAME 是没有打开文档时显示的菜单标识符，该菜单不需要修改。IDR_TEXTTYPE 是至少打开一个文档时显示的菜单的标识符。单击 IDR_TEXTTYPE 标识符，可以打开这个菜单的菜单编辑器。

在菜单编辑器中，首先打开“文件”菜单，在现有的“另存为”命令下加一个分隔线和“打印”命令。表 6.1 列出了加入菜单命令的性质。图 6.8 显示了完成的菜单。

表 6.1　　**菜单项的性质**

ID	标　题	其 他 功 能
none	无	Separator
IDR_FILE_PRINT	打印	无

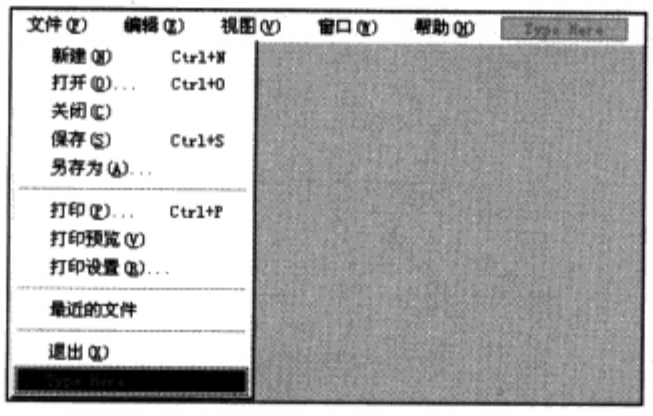


图 6.8　完成的菜单

然后，打开“编辑”菜单，在现有的“粘贴”命令下加入“全选”、分隔线、“查找”、“查找下一处”和“替换”命令。表 6.2 列出了这些菜单命令。图 6.9 显示了完成的“编辑”菜单。

表 6.2　　加入编辑菜单的新命令

ID	标　题	其 他 功 能
ID_EDIT_SELECT_ALL	全选	无
无	无	分隔线
ID_EDIT_FIND	查找	无
ID_EDIT_REPEAT	查找下一处	无
ID_EDIT_REPLACE	替换	无

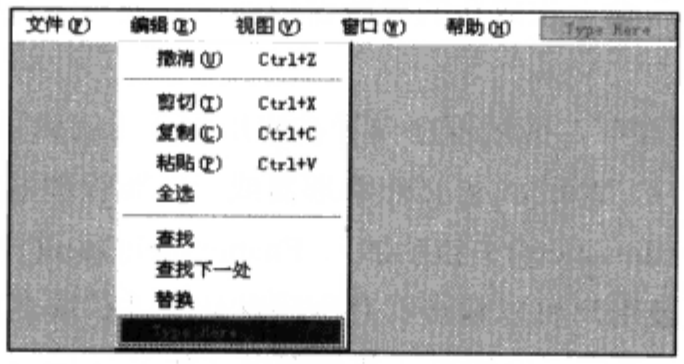


图 6.9　完成的编辑菜单

使用这种方法，可以自定义 MDI 应用程序中的其他资源。

6.3　Visual C++ 2010 View 开发

视图类是 Microsoft 基础类程序中用于管理显示窗口，处理用户输入和在视图窗口与其他设备上显示文档数据的部分。之前介绍过，在用应用程序向导生成程序时，它从 MFC CView 类派生了管理视图的类。应用程序向导派生的视图类本身不做任何事情，只是加入本程序的外壳。

本节会制作一个简单的绘图程序，用应用程序向导生成这个程序的基本外壳后，要在视图类中加入读取用户的鼠标输入和在视图窗口中画直线的代码。用 MFC Class Wizard 工具生成处理鼠标输入消息和定制视图窗口的函数，并用 Visual Studio 提供的资源管理器修改程序菜单和设计程序图标。

本节要生成的程序实现了程序视图类的基本特性，可以在视图窗口中绘制直线。为了画直线，应该将鼠标指针放在直线的一端，按下鼠标左键并拖动指针到线的另一端，再释放鼠标。要生成的程序如图 6.10 所示。

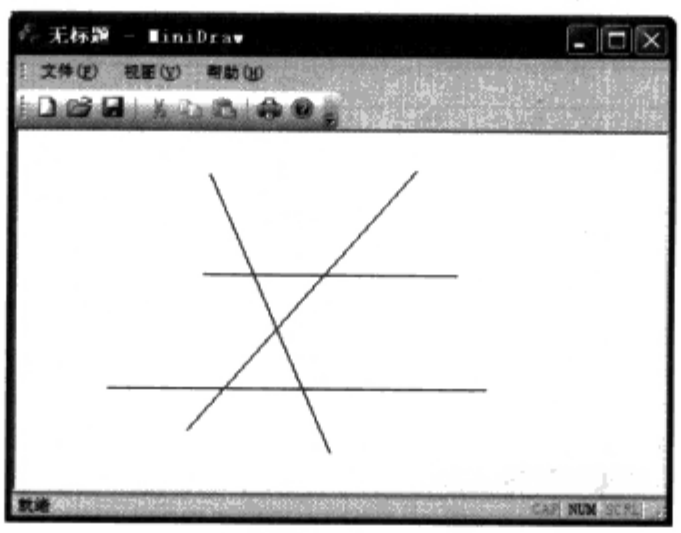


图 6.10　程序运行结果

在本程序中，窗口重绘或选择“File”→“New”命令时所有的线都要擦去，如改变窗口尺寸或删除重叠窗口时。本程序不实现重绘窗口的 OnDraw()函数，这是因为它没有存储重绘直线所要的数据。

6.3.1　生成源文件

为了生成程序源文件，与第 2 章中生成源代码部分一样，使用应用程序向导。但在“New Project”对话框的“Name”文本框中输入的是 MiniDraw，在“Locatoin”文本框中指定所要的项目文件夹路径，在此选择默认存储路径。图 6.11 显示了设置完参数的“New Project”对话框，在该图中，项目文件夹名为 MiniDraw，存放在 C:\Documents and Settings\Administrator\My Documents\Visual Studio 10 目录下。在应用程序向导对话框中，一定要设置和第 2 章生成应用程序 HelloWorld 时相同的选项。

使用应用程序向导生成源文件后，会在项目工作区窗口的“Solution Explorer”面板中显示这些文件的文件名，如图 6.12 所示（图已经完全展开）。当退出 Visual Studio 或者关闭 MiniDraw 项目后，可以随时选择“File”→“Open”→“Project/Solution”命令，并选择项目工作区文件 MiniDraw.sln，再次打开该项目文件。

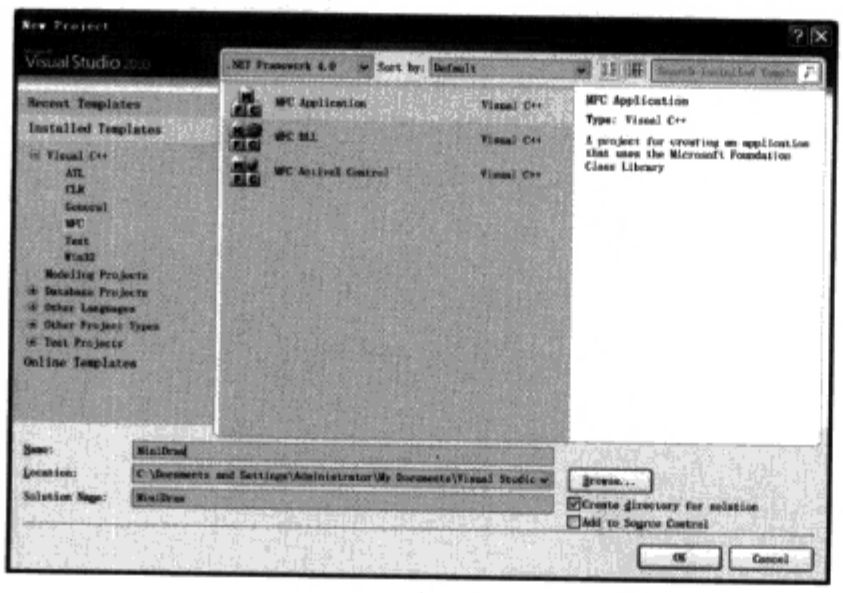


图 6.11　新建一个 MFC 应用程序

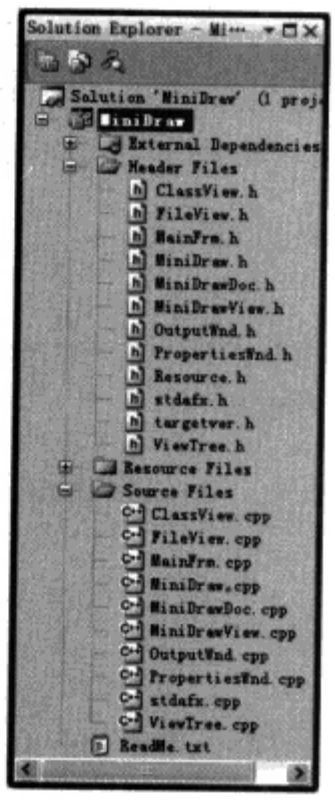


图 6.12　应用程序向导生成的源文件

6.3.2　初始化视图类数据成员

首先，在视图类中加入几个成员：m_ClassName、m_Dragging、m_HCross、m_PointOld 和 m_Point Origin。为此，打开文件 MiniDrawView.h，在 CMiniDrawView 类定义开头加入标为黑体的语句：

```
class CMiniDrawView : public CView
{

protected:
    CString m_ClassName;
    int m_Dragging;
    HCURSOR m_HCross;
    CPoint m_PointOld;
    CPoint m_PointOrigin;

protected: // 仅从序列化创建
```

```
    CMiniDrawView();
    DECLARE_DYNCREATE(CMiniDrawView)

// 属性
public:
    CMiniDrawDoc* GetDocument() const;

// 操作
public:

// 重写
public:
    virtual void OnDraw(CDC* pDC);  // 重写以绘制该视图
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);

// 实现
public:
    virtual ~CMiniDrawView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// 生成的消息映射函数
protected:
    afx_msg void OnFilePrintPreview();
    afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnContextMenu(CWnd* pWnd, CPoint point);
    DECLARE_MESSAGE_MAP()
};
```

其次，对 m_Dragging 和 m_HCross 的数据成员加入初始化代码（在 CMiniDrawView 类构造器中），此构造器在 MiniDrawView.cpp 文件中。

```
CMiniDrawView::CMiniDrawView()
{
    // TODO: 在此处添加构造代码
    m_Dragging=0;
    m_HCross=AfxGetApp()->LoadStandardCursor(IDC_CROSS);
}
```

m_HCross 存放在视图窗口中时程序显示的鼠标光标的句柄。调用 AfxGetApp()即取得程序的应用程序类对象（由 CWinApp 派生的 CMiniDrawApp 类的对象）的指针，这个指针用于调用 CwinApp 的成员函数 LoadStandardCursor()，它传送识别符 IDC_CROSS，返回标准十字形鼠标光标的句柄。下面会介绍 MiniDraw 程序如何在视图窗口显示该光标（在 OnMouse Move()函数部分）。表 6.3 列出了可

以传入 LoadStandardCursor()，以取得其他标准光标的句柄及其数值，表中列出了常用的光标用途，也可以将它们用于其他用途。

表 6.3　　可用的光标

数　值	光　标
IDC_ARROW	标准箭头光标
IDC_CROSS	十字形光标，用于选择
IDC_IBEAM	I 形光标，用于文本方式下
IDC_SIZEALL	带有四向箭头的方形光标，用于缩放对象
IDC_SIZENESW	双箭头光标，两端分别指向东北和西南
IDC_SIZENS	双箭头光标，两端分别指向北和南
IDC_SIZEWSE	双箭头光标，两端分别指向西北和东南
IDC_SIZEWE	双箭头光标，两端分别指向西和东
IDC_UPARROW	垂直箭头光标
IDC_WAIT	杯形光标，用于程序完成长时间任务时

AfxGetApp()是全局 MFC 函数，不是类的成员。MFC 中有许多全局 MFC 函数，函数名以 Afx 开头。

6.3.3　加入消息处理功能

为使用户用鼠标在显示窗口中画直线，程序必须响应窗口中产生的鼠标事件。为了处理鼠标输入，必须在视图类中加入处理发送到视图窗口的鼠标消息的成员函数，但在介绍加入消息处理函数之前，下面先简要地介绍一下 Windows 消息。

1．Windows 消息

Windows GUI 程序的每个窗口有一个相关的函数，称为窗口过程。与这个窗口相关的重大事件发生时，操作系统调用这个函数，传入发生事件的标识符及处理事件所需的相关数据，这个过程称为向窗口发送消息。

用 MFC 类生成和管理窗口时，MFC 提供窗口过程。MFC 提供的窗口过程完成可以发送的每类消息的最小默认处理。但如果要对特定消息类型完成定制处理，可以定义管理该窗口的类的成员消息处理函数。为了定义特定类的消息处理函数，可以用 MFC Class Wizard 工具。

例如，鼠标光标在显示窗口中时，用户单击鼠标左键，则视图窗口接到标识符为 WM_LBUTTONDOWN 的消息。为了自己处理这个消息，可以用 MFC Class Wizard 生成程序视图类的成员函数处理这个消息。

MFC 提供用户界面对象生成的消息的特殊处理。用户界面对象是 MFC 支持的标准接口元素，包括菜单命令、快捷键、工具栏中的按钮、状态栏指示图标和对话框控件（这里的对象不是 C++对象）。

用户界面对象生成的消息称为命令消息。用户界面对象或其中一个对象需要更新时，对象发送一个命令消息给主帧窗口，但 MFC 立即将消息重发到视图对象。如果视图对象没有提供处理器，则 MFC 重发消息到文档对象。如果文档对象不提供处理器，则重发消息到主帧窗口对象。如果主帧窗口对象未提供处理器，则重发消息到应用程序对象。最后，如果应用程序对象也不提供处理器，则消息接受最小默认处理。

这样，MFC 扩展基本的 Windows 消息机制，使命令消息不仅可以由管理窗口的对象处理，而且可以由任何其他的主程序对象处理（每个对象属于 MFC 类 CCmdTarget 直接或者间接派生类，它提供消

息机制的支持）。

MFC 命令消息路由机制的重要特性是，程序可以处理最适合的类中的特定命令消息。例如，在应用程序向导生成的程序中，“File”菜单的“Exit”命令是由应用程序类处理的，这是因为这个命令影响应用程序整体。与之对应，“File”菜单的“Save”命令则是由文档类处理的，这是因为这个类只负责存放和存储文档数据。

关于 MFC 消息发送路由和消息处理机制的进一步介绍，请查看 Visual C++联机帮助主题：Visual C++ Documentation, Using Visual C++, Visual C++ Programmer's Guide, Adding Program Functionality, Details, Message Handling and Mapping Topics。

2．添加 OnLButtonDown()函数

定义处理 WM_LBUTTONDOWN 消息的处理函数。鼠标光标在显示窗口中时，用户单击鼠标左键即发出这个消息。定义函数的步骤如下。

（1）确定 Visual Studio 2010 中打开了 MiniDraw 项目，选择“Project”→“Add Class”命令或按【Ctrl+W】组合键便会打开 Class Wizard 对话框（称为 MFC Class Wizard）。

（2）打开 MFC Class Wizard 对话框中定义成员函数的部分。

（3）设置“Class name”为 CMiniDrawView 类以便在视图类中加入消息处理函数。

（4）设置“Object ID”为 CminiDrawView，选择类名 CMiniDrawView 本身可以定义处理发送到显示窗口的普通通知消息的成员函数，还可以定义覆盖 CMiniDrawView 从 CView 和其他 MFC 基本类中继承的虚拟函数的成员函数。“Object ID”下拉列表框中的其他选项是特定用户界面对象的标识符，如菜单命令。选择一个选项，可以用于提供该对象产生的命令消息的处理器。

（5）设置 Message 为 WM_LBUTTONDOWN，这是要定义的函数处理的消息的标识符。“Message”下拉列表中可以包含发送到视图窗口的各类通知消息的标识符。消息标识符以 WM_开头，且全部大写。选择消息标识符，以定义处理该消息的函数。“Message”下拉列表中还包括属于 CView 类的每个虚拟函数名，选择其中一个以定义该函数的覆盖版本。注意：选择特定消息标识符或虚拟函数时，MFC Class Wizard 对话框询问分出现简单的说明了。

（6）单击“Add Function”按钮。MFC Class Wizard 生成消息处理函数的基本代码，名为 OnLButtonDown()。MFC Class Wizard 将函数声明加在文件 MiniDrawView.h 的 CMiniDrawView 类定义中。它在文件 MiniDrawView.cpp 中插入最小函数定义，并将函数加进类消息映射中。注意：这时的 Message 清单中该消息名显示黑体，表示该消息有消息处理器。注意 Member Functions 清单中加进了该函数和消息的名称（其中标有 W 的项目是 Windows 消息处理器，标有 V 的项目为覆盖虚拟函数）。

（7）单击“Finish”按钮，关闭 MFC Class Wizard 对话框。MFC Class Wizard 打开文件 MiniDrawView.cpp（如果尚未打开）并显示刚刚生成的函数 OnLButtonDown()。MFC Class Wizard 进入这个函数中，开发人员可以加入自己的编码。

（8）在 OnLButtonDown()中加入标黑体的语句：

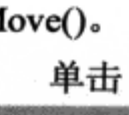
```
void CMiniDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    //TODO:Add your message handler code here and /or call
    //default
    m_PointOrigin = point;
    m_PointOld = point;
    SetCapture();
    m_Dragging =1;
```

```
        RECT Rect;
        GetClientRect ( & Rect);
        ClientToScreen(&Rect);
        ::ClipCursor(&Rect);

        CView::OnLButtonDown(nFlags, point);
    }
```

除了使用 MFC Class Wizard 向导生成 OnLButtonDown()函数之外，还可以自己在 MiniDrawView.h 和 MiniDrawView.cpp 中直接添加 OnLButtonDown()函数的声明和代码实现，这种方法比向导生成的方法更直接，更容易理解。

鼠标光标在视图窗口中，而用户单击鼠标左键时，OnLButtonDown()函数接到该事件，Point 参数包含光标的当前位置，所添加的代码将这个位置存放到 m_PointOrigin 和 m_PointOld 数据成员中。m_PointOrigin 存放左键按下的点坐标，将是所画直线的原点，m_PointOld 供其他鼠标消息处理器获得前一鼠标消息的鼠标位置。



MFC Class Wizard 向 OnLButtonDown()函数中加入了一行调用基本类中定义的 OnLButtonDown()版本的代码。这是通常要做的，便于基本类完成需要的默认消息处理。

调用 CWnd 成员函数 SetCapture()捕获鼠标，即所有的后续鼠标消息都发送到视图窗口（直到放弃捕获），这样，发生拖动操作时视图窗口充分控制鼠标，数据成员 m_Dragging 设置为 1，向其他鼠标消息处理器传送消息表示拖动操作正在运行。

注意：由于 CMiniDrawView 是直接从 CWnd MFC 类继承的，继承了所有的 CWnd 的众多成员函数（CWnd 类派生 CView 类，CView 类又派生 CminiDrawView 类）。为了找到特定类可用的所有的成员函数，参看关于类的文档中的类层次结构，参见下列联机帮助主题：Visual C++ Documentation Reference，Microsoft Foundation Class Library and Templates，Microsoft Foundation Class Library，Class Library Reference。然后从中看看各个基本类的文档中对成员函数的说明。本书提到特定成员函数时，通常表示出实际定义该函数的类，例如，SetCapture()函数或者说成 CWnd 成员函数 SetCapture()，写成 CWnd::SetCapture()（用作用域操作符表示包含函数定义的类）。

余下的代码行用于将鼠标指针限制在显示窗口内，使用户不会到窗口外去画直线。调用 CWnd::GetClient Rect()取得显示窗口的当前坐标，而调用 CWnd::ClientToScreen()将坐标换算为屏幕坐标，即对应于 Windows 屏幕左上角的坐标。最后，调用::ClipCursor()将鼠标光标限制在指定的屏幕范围内，从而保证光标在显示窗口中。

::ClipCursor()是 Win32 应用程序接口（API）提供的函数而不是 MFC 提供的。由于它是全局函数，名称前面使用作用域操作符（::）。除非全局函数被同名的成员函数掩盖，否则并不实际需要这个作用域操作符。但本书讲的 Win32 API 函数前用作用域操作符表示该函数不属于 MFC。

3．添加 OnMouseMove()函数

定义处理 WM_MOUSEMOVE 消息的函数。用户在显示窗口中移动鼠标光标时，这个窗口收到一系列的 WM_MOUSEMOVE 消息，用于报告光标的当前位置。为了定义这个消息的处理器，首先用 MFC Class Wizard 生成消息处理函数的外壳。按照上一小节的步骤进行，其中将 Message 设置为 WM_MOUSEMOVE，而不是 WM_LBUTTONDOWN。MFC Class Wizard 生成成员函数 OnMouse Move()。

单击“Finish”按钮后，将黑体行输入到 OnMouseMove()函数中：

```
void CMiniDrawView::OnMouseMove(UINT nFlags,CPoint point)
```

```
{
    //TODO:Add your message handler code here and /or call
    //default
    ::SetCursor(m_HCross);

    if(m_Dragging)
    {
        CClientDC ClientDC(this);
        ClientDC.SetROP2(R2_NOT);
        ClientDC.MoveTo(m_PointOrigin);
        ClientDC.LineTo(m_PointOld);
        ClientDC.MoveTo(m_PointOrigin);
        ClientDC.LineTo(point);
        m_PointOld=point;
    }
    CView::OnMouseMove(nFlags,point);
}
```

当然，也可以直接在 MiniDrawView.h 和 MiniDrawView.cpp 中添加 OnMouseMove()成员函数的声明和代码实现。这种方法比向导生成的方法更直接，更容易理解。

鼠标光标在视图窗口中移动时，OnMouseMove()函数以一定间隔频繁调用。加入的代码完成两项工作。

第一，调用 Win32 API 函数::SetCursor()显示十字形光标，而不是标准箭头光标（十字形光标的句柄是用类构造器取得的）。注意以后版本的 MiniDraw 程序根据激活的绘图工作，可以显示各种不同的光标。

第二，拖动操作进行时，即 m_Dragging 非 0 时，加入的代码完成下列操作。

（1）擦去上一个 WM_MOUSEMOVE 消息所画的线（如果有）。

（2）从直线原点（即单击鼠标左键的点，存在 m_PointOrigin 中）向光标当前位置画一条新直线。

（3）将光标当前位置存放到数据成员 m_PointOld 中。

为了在窗口中画线，OnMouseMove()首先生成与视图窗口相关的设备描述表，然后 OnMouseMove()调用 CDC::SetROP2()函数，生成逆转当前屏幕颜色来画线的绘图方式，这种方式下，特定位置首次画出的线是可见的，但同一位置再画一遍时，就不可见了。这样鼠标消息处理器很容易绘制和擦去一系列临时线。线是用函数 CDC::MoveTo()（指定直线的另一端的位置）和 CDC::LineTo()（指定直线另一端的位置）来绘制的。

OnMouseMove()函数的完整功能是，用户在视图窗口中拖动鼠标时，临时线总是连接原点与当前光标位置（这条线表示用户释放鼠标时固定直线所在的位置）。

4．添加 OnLButtonUp()函数

定义函数处理释放鼠标左键时发出的 WM_MOUSEMOVE 消息。用 MFC Class Wizard 像前面一样生成基本函数代码，但设置 Message 为 WM_MOUSEMOVE 标识符。函数生成后，加入 MiniDraw.cpp 中的 OnButtonUp()，定义如下：

```
void CMiniDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
    //TODO: Add your message handler code here and /or call default
    if(m_Dragging)
    {
```

```
            m_Dragging=0;
            ::ReleaseCapture();
            ::ClipCursor(NULL);
            CClientDC ClientDC(this);
            ClientDC.SetROP2(R2_NOT);
            ClientDC.MoveTo(m_PointOrigin);
            ClientDC.LineTo(m_PointOld);
            ClientDC.SetROP2(R2_COPYPEN);
            ClientDC.MoveTo(m_PointOrigin);
            ClientDC.LineTo(point);
        }

        CView::OnLButtonUp(nFlags,point);
    }
```

如果用户拖动鼠标指针（即 m_Dragging 为非 0），则加入的代码结束拖动操作并绘制固定直线，具体完成如下步骤。

（1）给 m_Dragging 赋值 0，告诉其他鼠标消息处理器，拖动操作不再进行。

（2）调用 Win32 API 函数::ReleaseCapture()结束鼠标捕获，鼠标消息再次发送到光标下面的窗口。

（3）向 Win32 API 函数::ClipCursor()传递 NULL，使用户可再次将鼠标指针移动到屏幕上的任何位置。

（4）擦去上一次 WM_MOUSEMOVE 消息处理器绘制的临时线。

（5）从原点到光标当前位置绘制固定线。

WM_LBUTTONDOWN 是 MiniDraw 程序中要处理的最后一个鼠标消息。表 6.4 和表 6.5 提供了鼠标的完整的通知消息。其他的 Windows 程序中可能要向处理器提供其中的其他消息。

表 6.4　　鼠标通知消息

消　息	鼠 标 事 件
WM_MOUSEMOVE	用户将鼠标移动到客户区中的新位置
WM_MOUSEWHEEL	用户滚动带轮鼠标的滚动轮
WM_LBUTTONDOWN	用户按下鼠标左键
WM_RBUTTONDOWN	用户按下鼠标右键
WM_LBUTTONUP	用户放开鼠标左键
WM_RBUTTONUP	用户放开鼠标右键
WM_LBUTTONDBCLK	用户双击鼠标左键
WM_RBUTTONDBCLK	用户双击鼠标右键

表 6.5　　鼠标的位标记

位　标　记	含　义
MK_CONTROL	按住【Ctrl】键时设置
MK_LBUTTON	按下鼠标左键时设置
MK_MKUTTON	按下鼠标中键时设置
MK_RBUTTON	按下鼠标右键时设置
MK_SHIFT	按住【Shift】键时设置

5．在 Message Map 中注册 3 个消息

打开 MiniDrawView.cpp，在 BEGIN_MESSAGE_MAP(CminiDrawView，CView)中添加 3 个消息

信息，如下所示。

```
BEGIN_MESSAGE_MAP(CMiniDrawView, CView)
// 标准打印命令
ON_COMMAND(ID_FILE_PRINT, &CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_DIRECT, &CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, &CMiniDrawView::OnFilePrintPreview)
ON_WM_LBUTTONDOWN()
ON_WM_MOUSEMOVE()
ON_WM_LBUTTONUP()
END_MESSAGE_MAP()
```

6.3.4 设计程序资源

本小节将介绍如何用 Visual Studio 定制 MiniDraw 程序资源，即菜单和程序图标。为了编写 MiniDraw 程序资源，展开项目工作区窗口的“Resource View”下的树形目录，显示各类程序资源的名称（Accelerator、Dialog、Icon、Menu、StringTable 和 Version）。图 6.13 显示了不完全扩展时的 Resource View 视图。

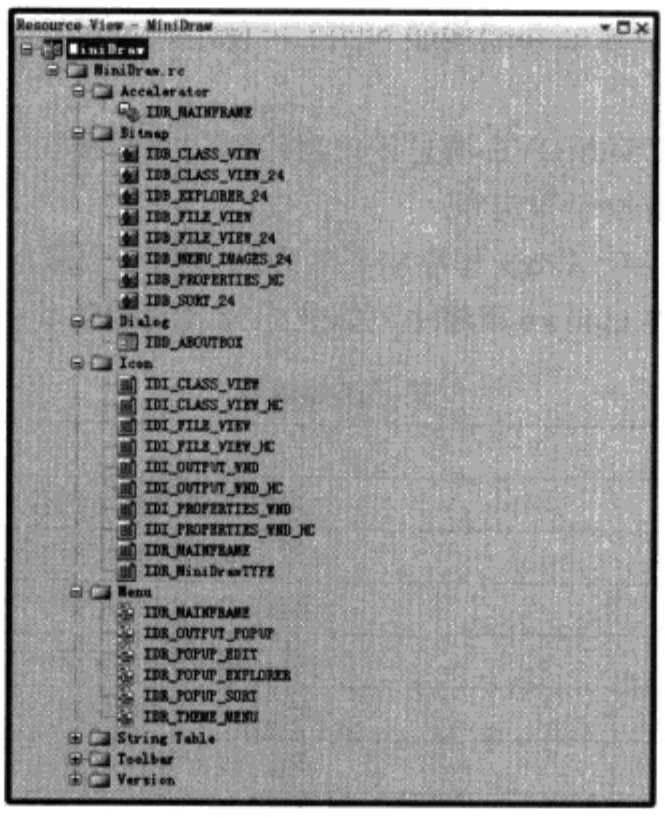


图 6.13　资源试图

定制 MiniDraw 菜单的具体步骤如下。

（1）双击“Resource View”下“Menu”分支下的 IDR_MAINFRAME 标识符。

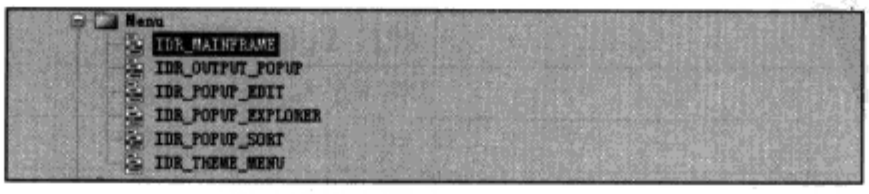


图 6.14　主框架菜单

同一标识符 IDR_MAINFRAME 也用于程序的加速表和主图标。Visual Studio 打开菜单编辑器，显示应用程序向导生成的 MiniDraw 程序菜单的直观表示，如图 6.15 所示。

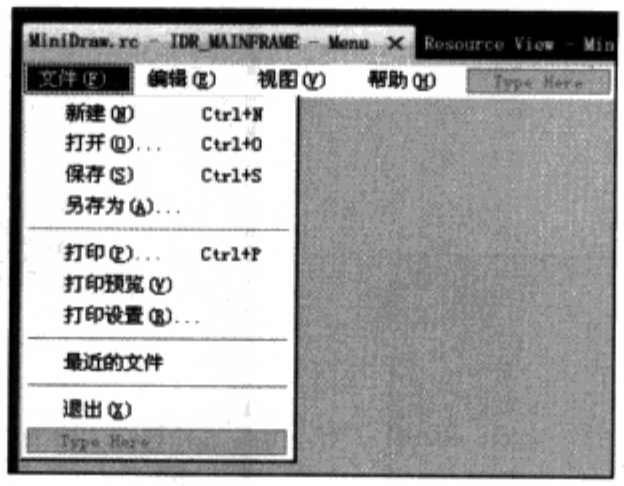


图 6.15　程序菜单

（2）在菜单编辑器中，单击“文件”菜单，删除菜单上除“新建”命令、“打开”命令和这两个命令间的分隔符以外的所有项目。要删除菜单命令和分隔符，只要单击该项目并按【Delete】键即可（菜单询问的窗框不必删除，这是加入新菜单项的位置，在最后的程序菜单中不会显示）。完成的文件菜单如图 6.16 所示。

（3）单击“编辑”菜单，并按【Delete】键删除整个“编辑”弹出菜单，Visual C++会询问是否要删除整个菜单，单击“是”按钮即可。

（4）现在删除了所有未用的菜单命令，最终的程序菜单如图 6.17 所示。双击系统菜单或单击“关闭”按钮关掉菜单编辑器。

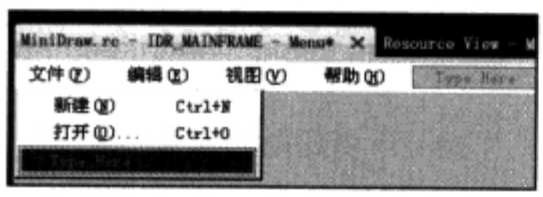


图 6.16　完成的文件菜单

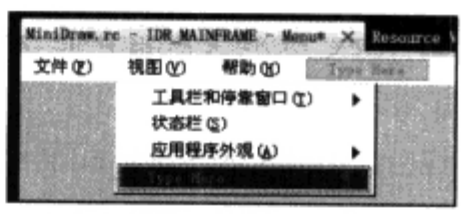


图 6.17　最终程序菜单

如果要设计 MiniDraw 程序的定制程序图标（取代标准的 MFC 图标），步骤如下。

- 双击项目工作区窗口“Resource View”中的“Icon”分支的 IDR_MAINFRAME，如图 6.18 所示。

Visual Studio 打开图形编辑器，显示当前程序图标。注意：图标文件包含个版本的图标图形：分别为 256×256 像素，48×48 像素，32×32 像素，24×24 像素，最小的图为 16×16 像素。如果用户在文件夹或 Explorer 的“View”菜单中选择“Small Icons”命令，则 Windows 文件夹或 Explorer 中显示小图表示程序对象。大图标则用于其他显示图标的位置（如程序的 About 对话框或 Windows Explorer 中，在用户选择“View”→“Large Icons”命令时）。

- 想要编辑哪个图标，只要在图形编辑器窗口顶端的“Device”中选择该图标。要删除当前图形以设计全新的图标，只要按【Delete】键即可。关于使用图形编辑器提供的命令和工具的信息，请看下列联机帮助主题：Visual C++ Documentation，Using Visual C++，Visual C++ User's Guide，Resource Editors，Graphics Editor。图 6.19 显示了 32×32 像素标准图标及其图形编辑器。
- 完成图标编辑后，双击系统菜单或单击关闭按钮，关闭图形编辑器窗口。

对程序资源做完所要的改变后，存储所有的修改。Visual C++将基本资源消息存放在 MiniDraw 资源定义文件 MiniDraw.rc 中，并将图标数据存放在 MiniDraw.ico 文件中（它是存放在项目文件夹\res 子文件夹中的二进制文件，资源定义文件中用 ICON 语句标识这个图标文件）。建立程序时，Microsoft 资源编译器程序（Rc.exe）处理这些文件的资源信息并将资源数据加入执行程序文件。

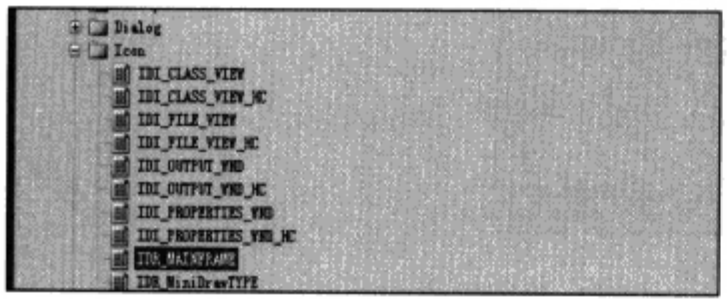


图 6.18　进入图标资源编辑

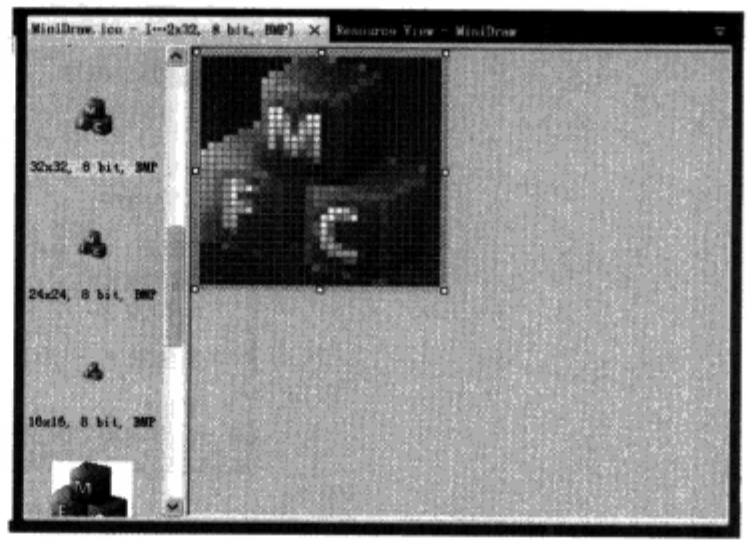


图 6.19　MFC 程序图标

6.3.5　定制 MiniDraw 窗口

当前的 MiniDraw 程序有两个问题。

- 尽管所写的 WM_MOUSEMOVE 消息处理器可以显示所需的十字形光标，但 Windows 也想显示标准箭头光标，这是因为 MFC 代码生成视图窗口时赋予它箭头光标。结果，用户在视图窗口中移动时，两种光标会来回闪烁。
- 如果用户用 Windows 控制面板选择黑色窗口颜色，则视图窗口中所画的直线不可见或难于看清，MFC 代码生成窗口时，将它设置成用控制面板上设置的当前窗口颜色画背景，而程序总是用黑色画直线。

可以在 CMiniDrawView 成员函数 PreCreateWindow()中加入若干条语句解决这两个问题。生成程序时，应用程序向导给 CMiniDrawView::PreCreateWindow()定义基本函数外壳。这个函数覆盖 CView 的虚拟成员函数 PreCreateWindow()，MFC 代码在生成视图窗口之前调用它。

由于 CView::PreCreateWindow()是重要的虚拟函数，应用程序向导在程序的视图类中自动生成函数的覆盖版本。

为了定制 MiniDraw 视图窗口，在 MiniDrawView.cpp 文件的 PreCreateWindow()中加入下面的黑体语句：

```
BOOL CMiniDrawView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: 在此处通过修改
    //  CREATESTRUCT cs 来修改窗口类或样式
    m_ClassName = AfxRegisterWndClass(CS_HREDRAW | CS_VREDRAW, 0, (HBRUSH)::GetStockObject(WHITE_BR-
USH), 0);
    cs.lpszClass=m_ClassName;
    return CView::PreCreateWindow(cs);
}
```

PreCreateWindow()接收 CREATESTRUCT 结构的调用，这个结构的字段存放生成窗口时 MFC 指定的窗口特性，如窗口坐标、窗口样式等。如果赋值这个结构的一个或几个字段，则 MFC 用所赋的值而不是默认值。

CREATESTRUCT 的 lpszClass 字段存放 Windows 窗口类名。Windows 窗口类不是 C++类，而是 Windows 系统维护的数据结构，存放窗口生成时使用的一组一般特性。刚刚加入的代码调用

AfxRegisterWndClass()生成新的 Windows 窗口类，然后将类名赋予 CREATESTRUCT 的 lpszClass 字段，使视图窗口用存放的这个 Windows 窗口类中定制生成（注意 AfxRegisterWndClass()是 MFC 提供的全局函数）。调用 AfxRegisterWndClass()函数指定下列特性。

- 第一个参数指定类样式 CS_HREDRAW 和 CS_VREDRAW，这两个样式使用户改变窗口大小时重绘窗口（MFC 视图窗口通常用这两种样式生成）。
- 第二个参数指定 Windows 在窗口内自动显示的鼠标光标，这个参数赋值为 0，使 Windows 不显示光标，因为光标由加入的 OnMouseMove()函数显式地显示，这样就消除了闪烁。
- 第三个参数提供标准魄画刷，用于绘制视图窗口背景。窗口背景总是魄的，黑线总是能够看见，而不受控制面板上选择的窗口颜色的影响。
- 最后一个参数指定窗口图标。由于视图窗口不显示图标，它赋值为 0（将程序图标赋予主帧窗口）。

MiniDraw 程序显示自定义光标，窗口类中不指定光标，OnMouseMove()消息处理函数中显示所要的图标，也可以将所要显示的光标的句柄作为第二个参数传递给 AfxRegisterWndClass()为窗口指定光标，但这个方法不便于程序运行时改变光标。

本小节介绍了 MiniDraw 程序所要做的最后改变，现在可以编译和运行程序了。选择“Build”→“Build MiniDraw”命令和“Debug”→“Start without Debugging”命令，或者直接按【Ctrl+F5】组合键，即可编译运行该程序。程序运行结果如图 6.20 所示。

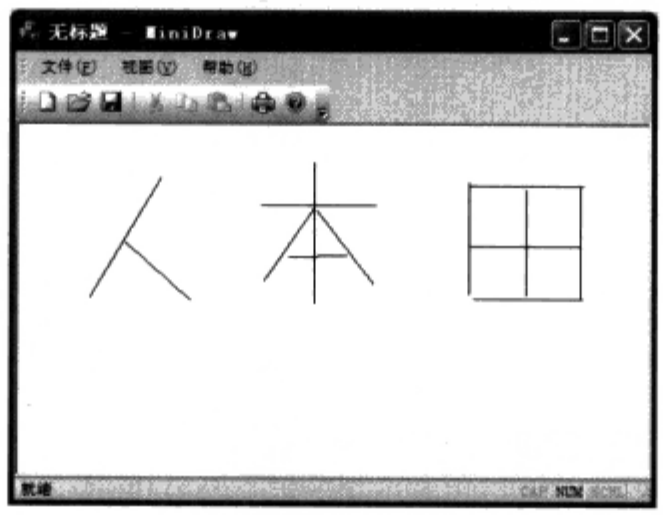


图 6.20　程序运行结果

6.3.6　程序清单

MiniDrawView.h

```
// MiniDrawView.h : CMiniDrawView 类的接口
//
#pragma once

class CMiniDrawView : public CView
{
protected:
	CString m_ClassName;
	int m_Dragging;
	HCURSOR m_HCross;
	CPoint m_PointOld;
```

```
        CPoint m_PointOrigin;

protected: // 仅从序列化创建
        CMiniDrawView();
        DECLARE_DYNCREATE(CMiniDrawView)

// 属性
public:
        CMiniDrawDoc* GetDocument() const;

// 操作
public:

// 重写
public:
        virtual void OnDraw(CDC* pDC);  // 重写以绘制该视图
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
        virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
        virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
        virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
        void OnLButtonDown(UINT nFlags, CPoint point);
        void OnMouseMove(UINT nFlags,CPoint point);
        void OnLButtonUp(UINT nFlags, CPoint point);

// 实现
public:
        virtual ~CMiniDrawView();
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// 生成的消息映射函数
protected:
        afx_msg void OnFilePrintPreview();
        afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
        afx_msg void OnContextMenu(CWnd* pWnd, CPoint point);
        DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // MiniDrawView.cpp 中的调试版本
inline CMiniDrawDoc* CMiniDrawView::GetDocument() const
   { return reinterpret_cast<CMiniDrawDoc*>(m_pDocument); }
#endif
```

Mini Draw View.cpp

```
// MiniDrawView.cpp : CMiniDrawView 类的实现
//
```

```
#include "stdafx.h"
#include "MiniDraw.h"

#include "MiniDrawDoc.h"
#include "MiniDrawView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// CMiniDrawView

IMPLEMENT_DYNCREATE(CMiniDrawView, CView)

BEGIN_MESSAGE_MAP(CMiniDrawView, CView)
    // 标准打印命令
    ON_COMMAND(ID_FILE_PRINT, &CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, &CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, &CMiniDrawView::OnFilePrintPreview)
    ON_WM_LBUTTONDOWN()
    ON_WM_MOUSEMOVE()
    ON_WM_LBUTTONUP()
END_MESSAGE_MAP()

// CMiniDrawView 构造/析构

CMiniDrawView::CMiniDrawView()
{
    // TODO: 在此处添加构造代码
    m_Dragging=0;
    m_HCross=AfxGetApp()->LoadStandardCursor(IDC_CROSS);
}

CMiniDrawView::~CMiniDrawView()
{
}

BOOL CMiniDrawView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: 在此处通过修改
    //  CREATESTRUCT cs 来修改窗口类或样式

    m_ClassName = AfxRegisterWndClass(CS_HREDRAW | CS_VREDRAW, 0, (HBRUSH)::GetStockOb-j ect(WHITE_
BRUSH), 0);
    cs.lpszClass=m_ClassName;

    return CView::PreCreateWindow(cs);
}
// CMiniDrawView 绘制
```

```
void CMiniDrawView::OnDraw(CDC* /*pDC*/)
{
    CMiniDrawDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;

    // TODO: 在此处为本机数据添加绘制代码
}


// CMiniDrawView 打印


void CMiniDrawView::OnFilePrintPreview()
{
    AFXPrintPreview(this);
}

BOOL CMiniDrawView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // 默认准备
    return DoPreparePrinting(pInfo);
}

void CMiniDrawView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: 添加额外的打印前进行的初始化过程
}

void CMiniDrawView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: 添加打印后进行的清理过程
}

void CMiniDrawView::OnRButtonUp(UINT /* nFlags */, CPoint point)
{
    ClientToScreen(&point);
    OnContextMenu(this, point);
}
void CMiniDrawView::OnContextMenu(CWnd* /* pWnd */, CPoint point)
{
    theApp.GetContextMenuManager()->ShowPopupMenu(IDR_POPUP_EDIT, point.x, point.y, this, TRUE);
}

// CMiniDrawView 诊断
#ifdef _DEBUG
void CMiniDrawView::AssertValid() const
```

```
{
    CView::AssertValid();
}
void CMiniDrawView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}
CMiniDrawDoc* CMiniDrawView::GetDocument() const // 非调试版本是内联的
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMiniDrawDoc)));
    return (CMiniDrawDoc*)m_pDocument;
}
#endif //_DEBUG
// CMiniDrawView 消息处理程序
void CMiniDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
//TODO:Add your message handler code here and /or call
//default
m_PointOrigin = point;
m_PointOld = point;
SetCapture();
m_Dragging =1;

RECT Rect;
GetClientRect ( & Rect);
ClientToScreen(&Rect);
::ClipCursor(&Rect);

CView::OnLButtonDown(nFlags, point);
}
void CMiniDrawView::OnMouseMove(UINT nFlags,CPoint point)
{
    //TODO:Add your message handler code here and /or call
//default
::SetCursor(m_HCross);
if(m_Dragging)
{
    CClientDC ClientDC(this);
    ClientDC.SetROP2(R2_NOT);
    ClientDC.MoveTo(m_PointOrigin);
    ClientDC.LineTo(m_PointOld);
    ClientDC.MoveTo(m_PointOrigin);
    ClientDC.LineTo(point);
    m_PointOld=point;
}
CView::OnMouseMove(nFlags,point);
}
void CMiniDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
```

```
        //TODO: Add your message handler code here and /or call default
        if(m_Dragging)
        {
            m_Dragging=0;
            ::ReleaseCapture();
            ::ClipCursor(NULL);
            CClientDC ClientDC(this);
            ClientDC.SetROP2(R2_NOT);
            ClientDC.MoveTo(m_PointOrigin);
            ClientDC.LineTo(m_PointOld);
            ClientDC.SetROP2(R2_COPYPEN);
            ClientDC.MoveTo(m_PointOrigin);
            ClientDC.LineTo(point);
        }

    CView::OnLButtonUp(nFlags,point);

}
```

6.4 Visual C++ 2010 EditView 开发

本节将制作一个简单的文字编辑程序，这个程序演示了如何从 MFC 类 CEditView 而不是从 CView 类派生程序的视图类，迅速生成完全功能化的文字编辑器。

从 MFC CEditView 类而不是 CView 类派生视图类可以方便地生成全功能的文字编辑器。从 CEditView 派生的视图类可以使程序用户在视图窗口中输入和编辑文本，包含相当完整的键盘和菜单编辑命令支持，不需要开发人员自己实现这些特性。

本节用应用程序向导生成 MiniEdit 程序。其中视图类要从 CEditView 而不是从 CView 中派生，因而生成视图窗口中方便的文字编辑器，然后用 Visual Studio 菜单编辑器加入菜单命令，使用户能够访问一些编辑器命令，还要用 Visual Studio 资源编辑器加入快捷键和设计程序图标。

MiniEdit 程序窗口如图 6.21 所示，程序可以用于在视图窗口中输入和编辑文本，并显示滚动条，可以在文本无法全部放入窗口时滚动文本，程序菜单包含打印文本，取消最后编辑动作，剪切、复制和粘贴文本，选择文本并完成查找与替换操作的命令。如果在视图窗口中右击，程序还会显示包含大多数 Edit 菜单命令的浮动菜单。

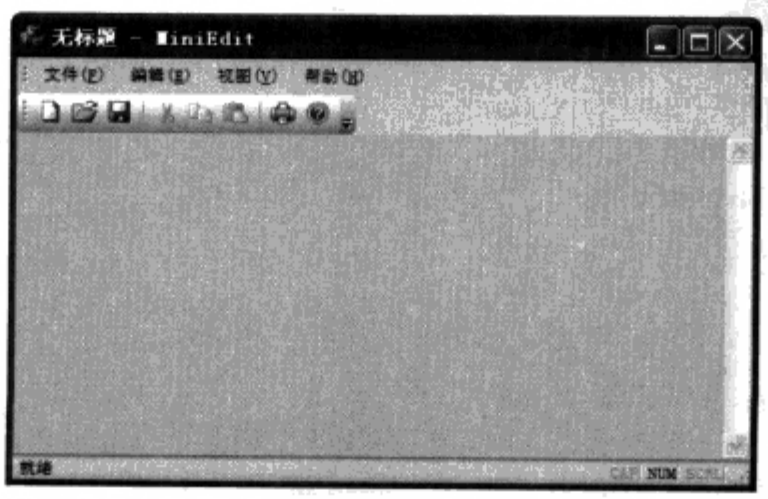


图 6.21 MiniEdit 程序窗口

6.4.1　生成 MiniEdit 程序

生成 MiniEdit 程序时，首先像 6.3 节的 MiniDraw 程序那样，用应用程序向导生成源代码文件。在“New Project”对话框中，在“Name”文本框中输入 MiniEdit，Location 文本框中输入项目文件夹路径，在此选择默认存储路径 C:\Documents and Settings\Administrator\My Documents\Visual Studio 10\Projects，如图 6.22 所示。

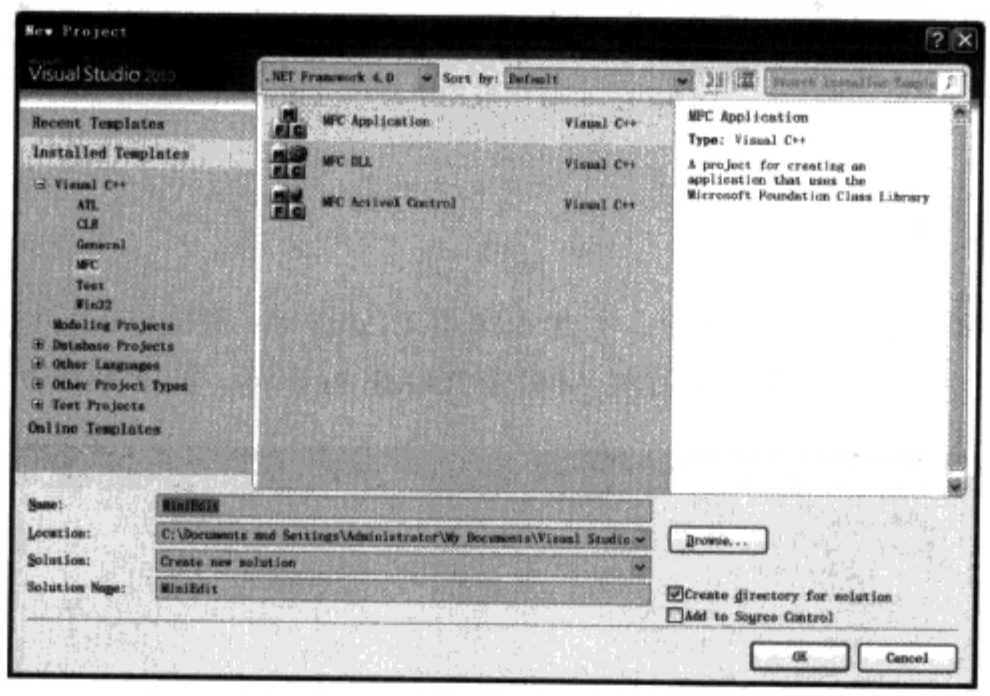


图 6.22　新建 MFC 应用程序 MiniEdit

在应用程序向导对话框中，选择与生成前面 MiniDraw 程序相同的选项，但在“Generated Classes”对话框中，不是接受所有的默认值，而是要按照如下步骤操作。

（1）设置“Generated classes”为 CMiniEditView。

（2）在“Base classes”下拉列表框中选择 CeditView 选项，完成的“Generated Classes”对话框如图 6.23 所示。

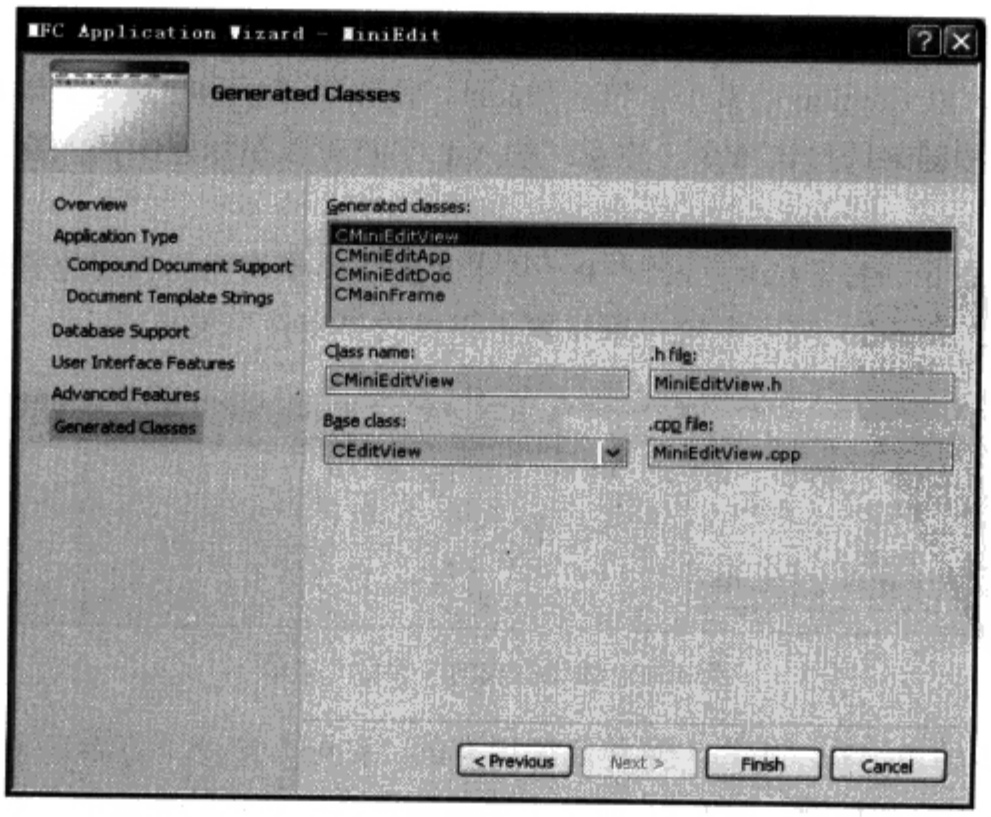


图 6.23　完成的 Generated Class 对话框

（3）单击“Finish”按钮，生成源代码。

生成源代码后，单击项目工作区窗口中的“Resource View”标签，显示程序资源，如图 6.24 所示。

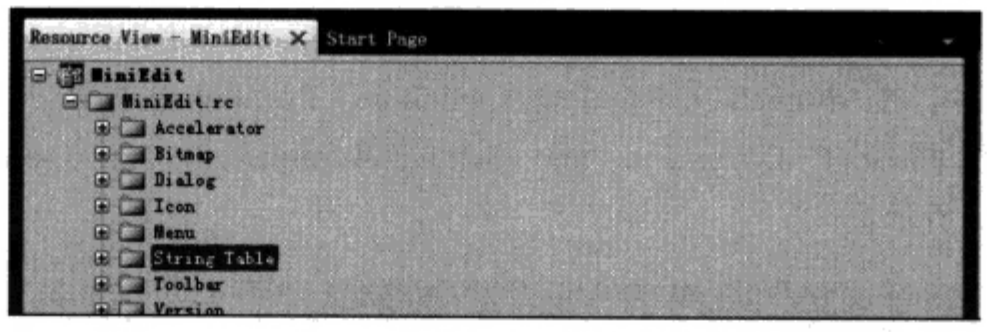


图 6.24　向导生成的字符串资源

6.4.2　修改程序菜单

首先修改程序菜单，删除不用的命令，并加入调用 CEditView 类提供的特性的新命令。为此，在 Resource View 中，双击 Menu 分支下的 IDR_MAINFRAME 标识符，如图 6.25 所示。

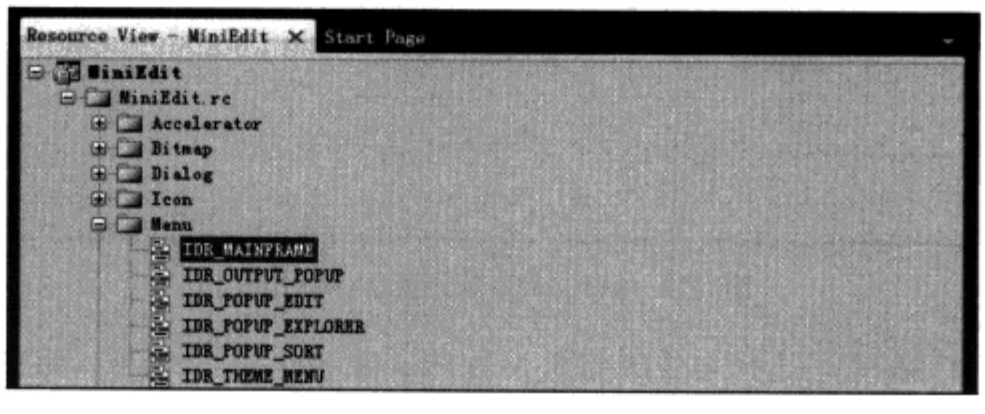


图 6.25　主框架菜单资源

打开菜单编辑器时，进行下列更改。

（1）删除“文件”菜单下除“退出”命令和其上的分隔线以外的所有项目。

（2）双击“文件”菜单询问的空框，加入新项目。Visual C++打开 Menu Item Properties 面板。在 ID 文本框中输入标识符 ID_FILE_PRINT，设置 Caption 为“打印(&P)...\tCtrl+P”，新命令即显示在“文件”菜单下（注意，可以 Caption 的某个字符前加&号，使该字符显示在菜单上时加下划线，用户可以按下划线字符对应的按键来选择该命令。注意：输入\t 字符可以在标题中插入制表符，使后面字符在菜单中右对齐）。

（3）用鼠标将新命令（打印(&P)...\tCtrl+P）从“文件”菜单询问的当前位置拖动到菜单命令顶部所留的分隔线的上方。图 6.26 显示了菜单编辑器中完成的“文件”菜单。

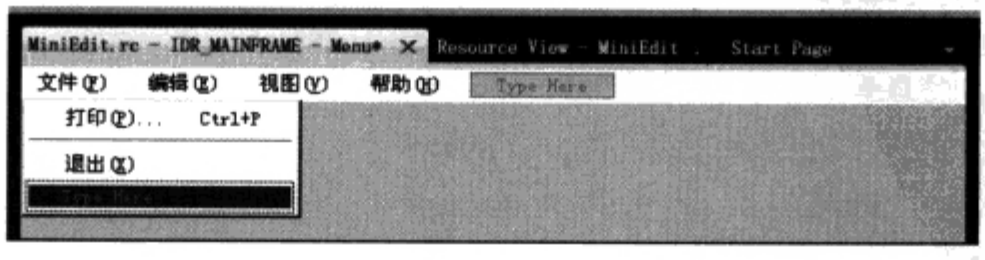


图 6.26　编辑完成的“文件”菜单

（4）用第（2）步的方法在“编辑”菜单底部现有的“粘贴”命令下方添加 4 个新命令。表 6.6 列出了这几个命令的标识符和标题。图 6.27 显示了菜单编辑器中显示的完成的“编辑”菜单，加入“全选”命令后，要插入一个分隔线。要插入分隔线，需要在 Menu Item Properties 面板中选择“分隔线”

选项而不是输入标识符和标题。

表 6.6　　菜单项标识符和标题

ID	标　题
ID_EDIT_SELECT_ALL	全选
ID_EDIT_FIND	查找
ID_EDIT_REPEAT	查找下一处\F3
ID_EDIT_REPLACE	替换

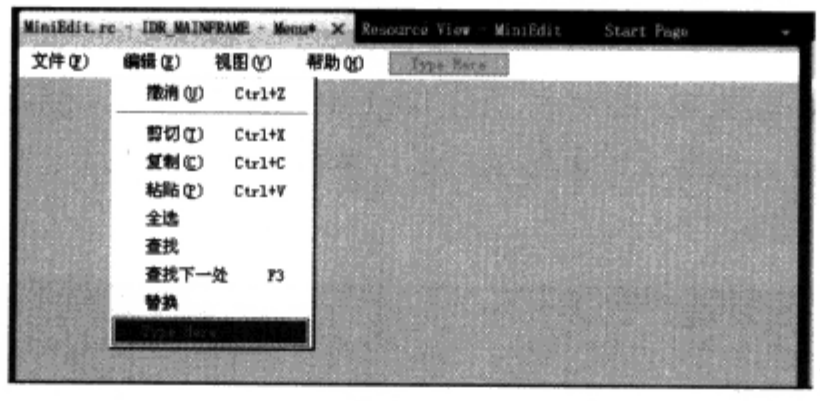


图 6.27　完成的“编辑”菜单

（5）双击系统菜单或单击关闭按钮退出菜单编辑器。

加入菜单命令时，要用说明中给出的准确菜单命令标识符，使 CEditView 中定义的消息映射能将每个命令消息发送到相应的消息处理函数(用于这些命令的是标准 MFC 标识符)。但是，对于加进“Edit”菜单的每个命令，Menu Item Properties 面板询问会显示一下预定义提示；而如果程序显示状态栏，则选择菜单命令时，MFC 代码把提示显示在状态栏中。

6.4.3　编辑加速键

为加入的两个命令的标题（“打印”命令和“查找下一处”命令）指定了执行命令的快捷键。这种快捷键称为键盘加速键，必须用 Visual Studio 的加速键编辑器定义。定义这两个键盘加速键的步骤如下。

（1）双击 Resource View 中 Accelerator 分支下的标识符 IDR_MAINFRAME，如图 6.28 所示。

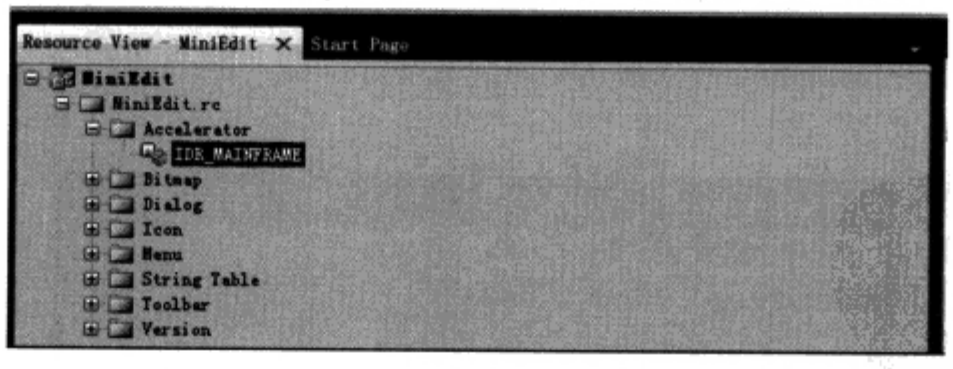


图 6.28　设置主框架菜单的加速键

打开加速键编辑器，如图 6.29 所示。

（2）双击加速键列表底部的空框并加入新键。在窗口最右侧打开 Accel Properties 面板。

（3）在 ID 文本框中输入标识符 ID_FILE_PRINT，这个标识符赋予“文件”菜单的“打印”命令。

（4）然后设置“Ctrl”为“True”，“Key”为“P”，这样，就将【Ctrl+P】组合键作为“打印”命令的加速键。

（5）再次双击加速键编辑器的列表底部的空框。

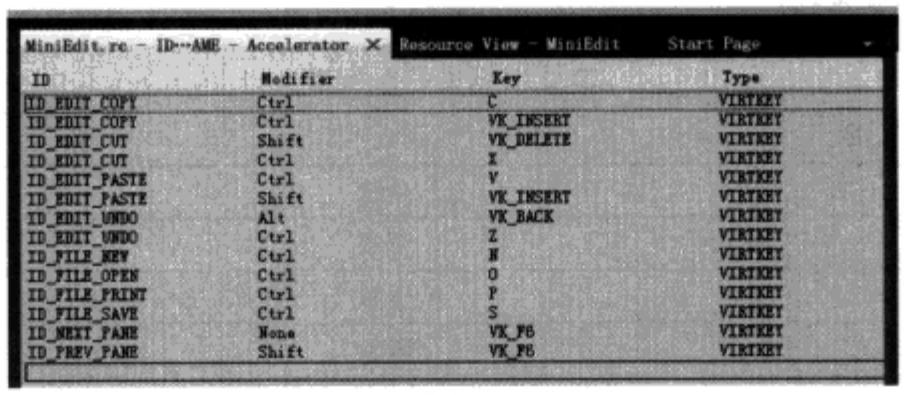


图 6.29　加速键编辑窗口

（6）在 Accel Properties 面板中，输入标识符 ID_EDIT_REPEAT，然后设置“Ctrl”为“False”，“Key”为“VK_F3”，这样，就将【F3】键变成了“编辑”→“查找下一处”命令的加速键。编辑后的加速键编辑器如图 6.30 所示。

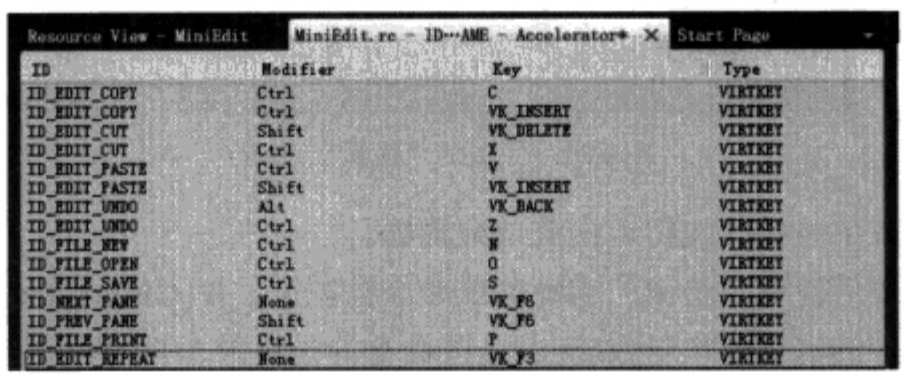


图 6.30　编辑后的加速键编辑器

（7）关闭加速键编辑器。

最后，可以用图形编辑器设计 MiniEdit 程序的定制图标。按照 MiniDraw 程序中设计程序资源部分的说明，来编辑 IDR_MAINFRAME 图标。

完成 MiniEdit 程序资源编辑器，选择“File”→“Save All”命令或单击标准工具栏中的“Save All”按钮存储所有修改。

现在，可以编译 MiniEdit 程序。选择“Build”→“Build MiniEdit”命令和“Debug”→“Start without Debugging”命令，编译并运行程序，就可以试试这些特性了，程序运行结果如图 6.31 所示。注意，所有的菜单命令都是全功能的，大多数命令由 CEditView 类提供的代码处理。MFC 提供了右击视图窗口时显示“Edit”菜单的代码。注意：如果输入文字到视图窗口后，退出 MiniEdit，程序还提供将文本存盘的机会。

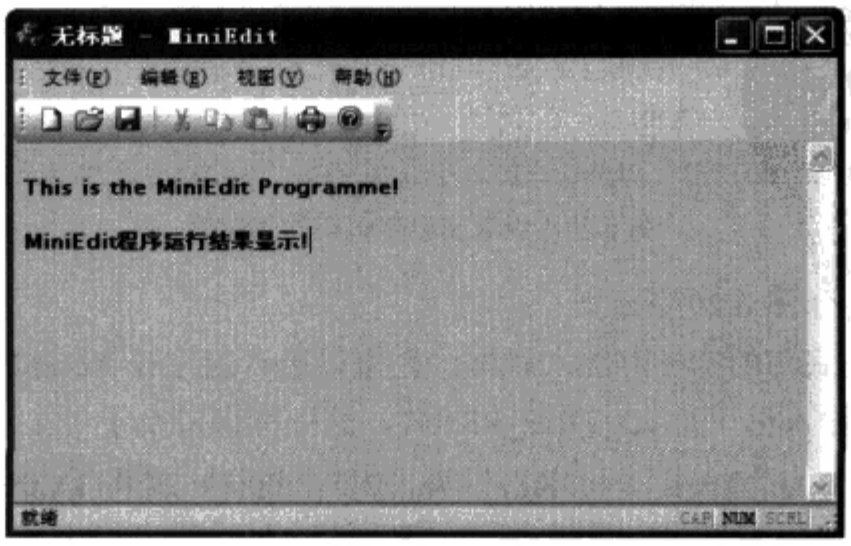


图 6.31　MiniEdit 运行结果

6.4.4　程序清单

MiniEditView.h

```
// MiniEditView.h : CMiniEditView 类的接口
//
#pragma once

class CMiniEditView : public CEditView
{
protected: // 仅从序列化创建
    CMiniEditView();
    DECLARE_DYNCREATE(CMiniEditView)

// 属性
public:
    CMiniEditDoc* GetDocument() const;

// 操作
public:

// 重写
public:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);

// 实现
public:
    virtual ~CMiniEditView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// 生成的消息映射函数
protected:
    afx_msg void OnFilePrintPreview();
    afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnContextMenu(CWnd* pWnd, CPoint point);
    DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // MiniEditView.cpp 中的调试版本
```

```
inline CMiniEditDoc* CMiniEditView::GetDocument() const
   { return reinterpret_cast<CMiniEditDoc*>(m_pDocument); }
#endif
```

Mini Edit View.cpp

```
// MiniEditView.cpp : CMiniEditView 类的实现
//
#include "stdafx.h"
#include "MiniEdit.h"
#include "MiniEditDoc.h"
#include "MiniEditView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// CMiniEditView

IMPLEMENT_DYNCREATE(CMiniEditView, CEditView)

BEGIN_MESSAGE_MAP(CMiniEditView, CEditView)
    // 标准打印命令
    ON_COMMAND(ID_FILE_PRINT, &CEditView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, &CEditView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, &CMiniEditView::OnFilePrintPreview)
END_MESSAGE_MAP()

// CMiniEditView 构造/析构

CMiniEditView::CMiniEditView()
{
    // TODO: 在此处添加构造代码

}

CMiniEditView::~CMiniEditView()
{
}

BOOL CMiniEditView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: 在此处通过修改
    //  CREATESTRUCT cs 来修改窗口类或样式

    BOOL bPreCreated = CEditView::PreCreateWindow(cs);
    cs.style &= ~(ES_AUTOHSCROLL|WS_HSCROLL);      // 启用换行

    return bPreCreated;
}
```

```
// CMiniEditView 打印


void CMiniEditView::OnFilePrintPreview()
{
    AFXPrintPreview(this);
}

BOOL CMiniEditView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // 默认 CEditView 准备
    return CEditView::OnPreparePrinting(pInfo);
}

void CMiniEditView::OnBeginPrinting(CDC* pDC, Cprint Info* pInfo)
{
    // 默认 CEditView 开始打印
    CEditView::OnBeginPrinting(pDC, pInfo);
}

void CMiniEditView::OnEndPrinting(CDC* pDC, CPrintInfo* pInfo)
{
    // 默认 CEditView 结束打印
    CEditView::OnEndPrinting(pDC, pInfo);
}

void CMiniEditView::OnRButtonUp(UINT /* nFlags */, CPoint point)
{
    ClientToScreen(&point);
    OnContextMenu(this, point);
}

void CMiniEditView::OnContextMenu(CWnd* /* pWnd */, CPoint point)
{
    theApp.GetContextMenuManager()->ShowPopupMenu(IDR_POPUP_EDIT, point.x, point.y, this, TRUE);
}


// CMiniEditView 诊断

#ifdef _DEBUG
void CMiniEditView::AssertValid() const
{
    CEditView::AssertValid();
}

void CMiniEditView::Dump(CDumpContext& dc) const
{
    CEditView::Dump(dc);
```

```
}

CMiniEditDoc* CMiniEditView::GetDocument() const // 非调试版本是内联的
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMiniEditDoc)));
    return (CMiniEditDoc*)m_pDocument;
}
#endif //_DEBUG


// CMiniEditView 消息处理程序
```

6.5 Visual C++ 2010 FormView 开发

用应用程序向导生成 FormDemo 程序基本代码。在“New Project”对话框中，设置项目类型 MFC AppWizard(exe)，在“Name”文本框中输入程序名称 FormDemo，“Location”文本框中输入所要的项目文件夹，在此选择默认存储路径 C:\Documents and Settings\Administrator\My Documents\Visual Studio 10\Projects。在应用程序向导“Application Type”对话框中，选中“Single document”单选按钮和“Document/View architecture support”复选框（窗体视图程序不选中“Dialog based”单选按钮），然后在“Resource language”下拉列表框中选择“中文（中国）”选项，如图 6.32 所示。

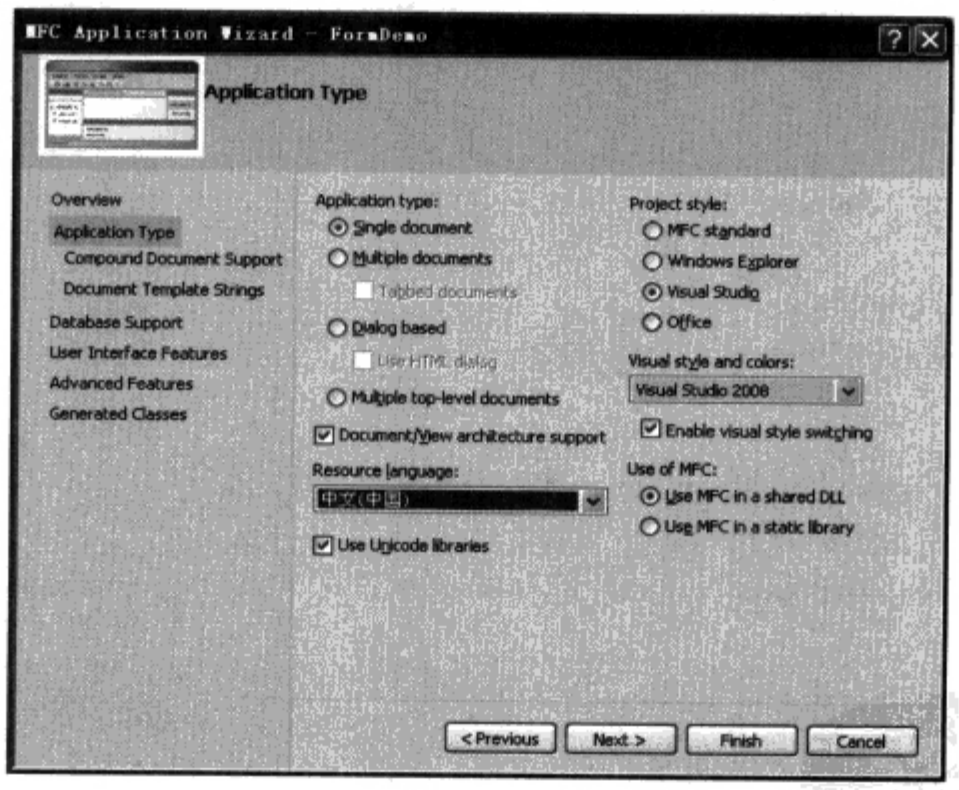


图 6.32 新建 FormDemo 程序向导设置

在应用程序向导对话框中设置的其他选项基本与 MiniEdit 程序相同，但在“Generated Classes”对话框中应该设置如下。

（1）选择对话框顶端列表框中的视图类 CFormDemoView。

（2）选择“Base class”下拉列表框中的 CFormView，使视图类从 CFormView 继承。完成的“Generated Classes”对话框如图 6.33 所示。

（3）单击“Finish”按钮，生成 FormDemo 程序清单。

应用程序向导生成源代码文件后，自动在 Visual Studio 对话框编辑器中打开程序对话框。

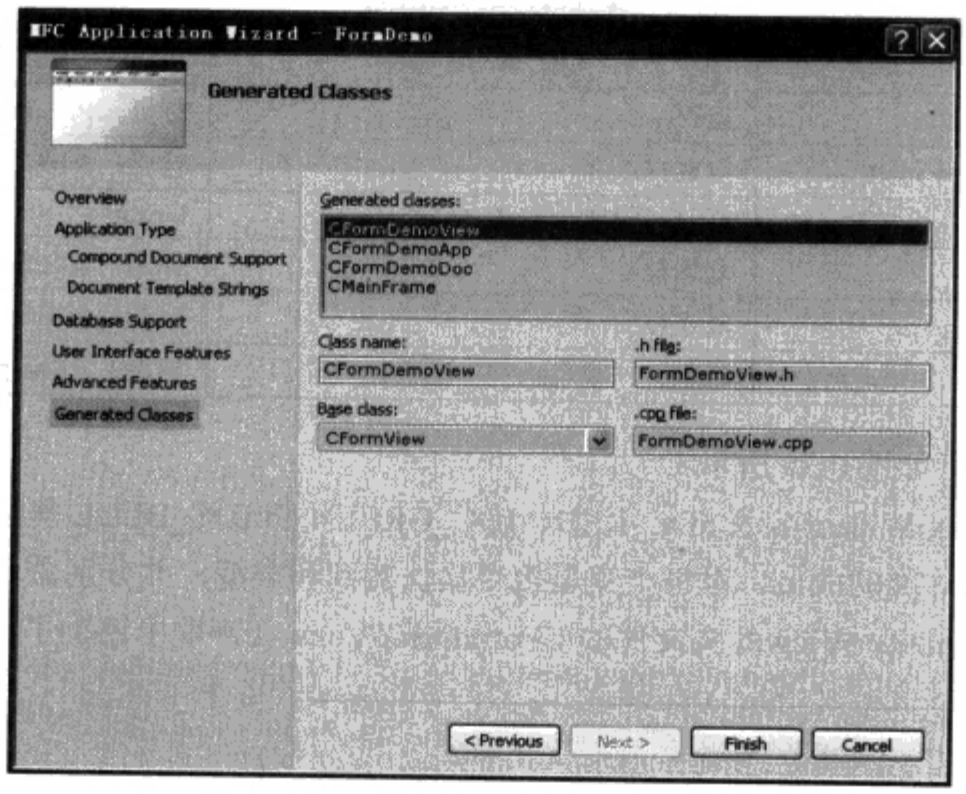


图 6.33　完成的 Generated Class 对话框

6.5.1　自定义 FormDemo 程序

完成源代码文件的生成后，继续编写 FormDemo 程序如下。

（1）此时，对话框编辑器已经打开程序对话框，否则单击项目工作区窗口的“Resource View”标签，双击 IDD_FORMDEMO_FORM 对话框标识符，打开对话框编辑器。IDD_FORMDEMO_FORM 对话框模板是由应用程序向导生成的，与视图类相联系，即视图窗口自动显示这个模板中包含的控件。注意：最初模板只包含 TODO 静态文本控件。不需要改变对话框的任何属性，这是因为应用程序向导最初赋予属性适合于表格视图窗口显示的对话框模板。对话框模板没有赋予边框、标题栏和其他可见元素，这是因为这些元素由分帧视图窗口的主帧窗口提供（打开 Dialog Properties 面板可以看到其属性）。

删除现有的静态文本控件，加入表 6.7 中和图 6.34 所示的控件。

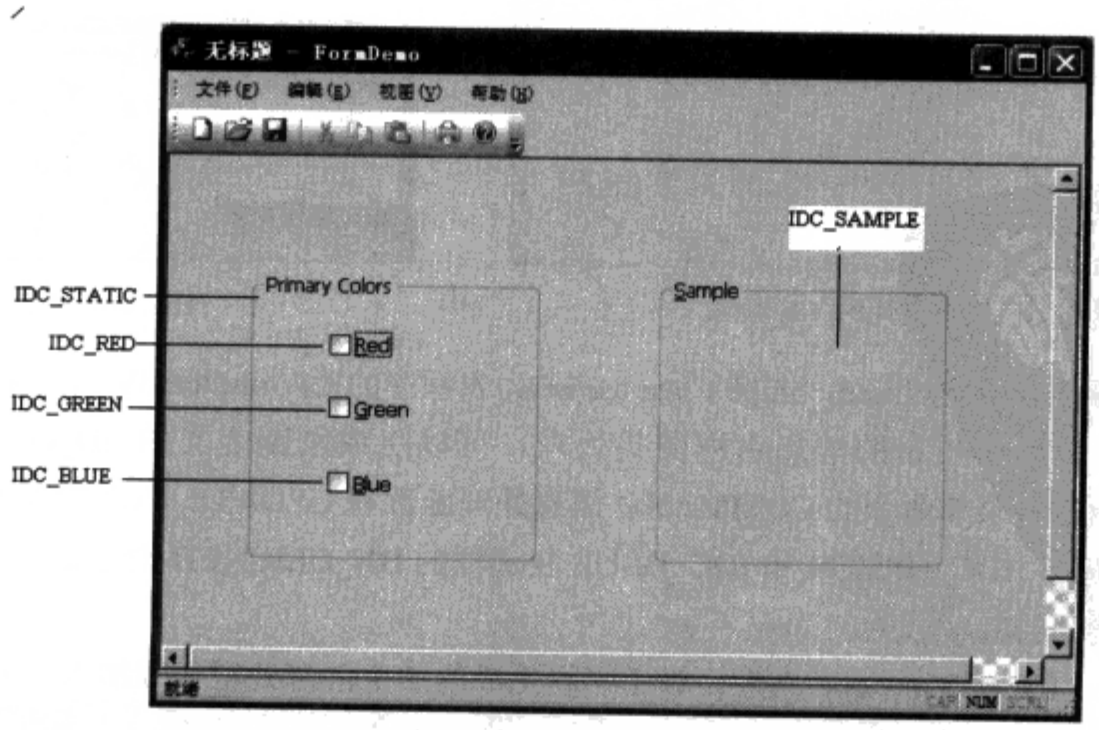


图 6.34　FormDemo 程序设计界面

表 6.7　　控件标识符和类型

ID	控 件 类 型	要设置的非默认属性
IDC_STATIC	组框	标题：Primary Colors
IDC_RED	复选框	标题：&Red
		Group
IDC_GREEN	复选框	标题：&Green
IDC_BLUE	复选框	标题：&Blue
IDC_SAMPLE	组框	标题：&Sample

（2）设置移动顺序如图 6.35 所示。

（3）用 MFC Class Wizard 定义 IDC_RED、IDC_GREEN 和 IDC_BLUE 复选框的成员变量，它们应为 CFormDemoView 类的成员。对每个变量接受默认类别和类型，并分别命名为 m_Red、m_Green 和 m_Blue。则在 FormDemoView.h 头文件的 CFormDemoView 类声明中添加了如下语句：

```
public:
CButton m_Red;
CButton m_Green;
CButton m_Blue;
```

（4）打开 IDR_MAINFRAME 菜单的菜单编辑器。删除“编辑”菜单，并将“文件”菜单标题变为“&选项”。删除“选项”菜单上除了“退出”命令及其上方分隔线以外的所有命令。最后在分隔线上面加入“浅色”和“深色”命令，如表 6.8 所示。完成的“选项”菜单如图 6.36 所示。

表 6.8　　颜色菜单项 ID 和标题

ID	标　　题
ID_OPTIONS_LIGHT	&浅色
ID_OPTIONS_DARK	&深色

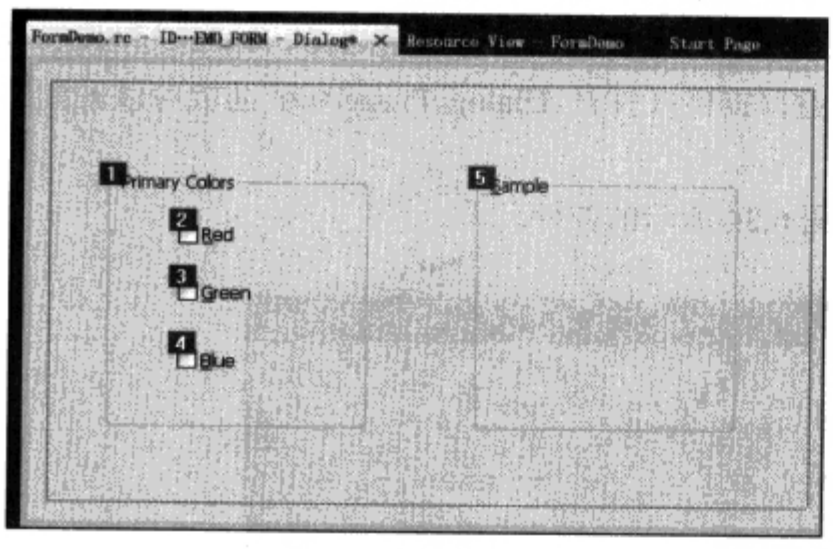


图 6.35　设置控件的 Tab 顺序

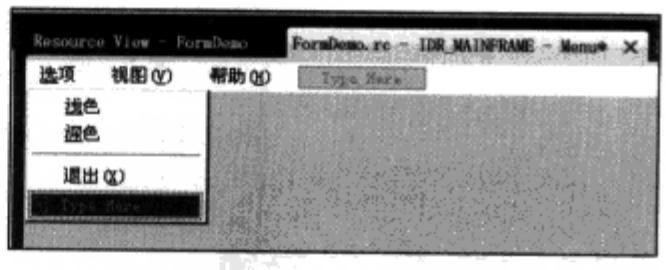


图 6.36　设置菜单项

（5）运行 MFC Class Wizard，选择 Class name 列表框中的 CFormDemoView。在 FormDemo 程序中，加入的菜单命令和按钮的消息由视图类处理，可以生成刚刚定义的 ID_OPTIONS_DARK 和 ID_OPTIONS_LIGHT 菜单命令的 COMMAND 消息处理函数和 UPDATE_COMMAND_UI 消息处理函数。定义 IDC_RED、IDC_GREEN 和 IDC_BLUE 复选框的 BN_CLICKED 消息处理函数。对于所有的函数，均接受默认名称。

（6）打开 FormDemoView.cpp 文件，对于刚刚生成的 5 个消息处理函数加入下列代码：

```
//CFormDemoView message Handlers
```

```
void CFormDemoView::OnOptionsDark()
{
    //TODO:Add your command handler code here
    m_Intensity=INT_DARK;
    CClientDC ClientDC(this);
    OnPrepareDC (&ClientDC);
    CRect Rect=m_RectSample;
    ClientDC.LPtoDP(&Rect);
    InvaliddataRect(&Rect);
    UpdateWindow();
}

void CFormDemoView::OnUpdateOptionsDark(CCmdUI* pCmdUI)
{
    //TODO:Add your command update UI handler code here
    pCmdUI->SetRadio (m_Intensity==INT_DARK);
}


void CFormDemoView::OnOptionsLight()
{
    //TODO:Add your command handler code here
    m_Intensity =INT_LIGHT;
    CClientDC ClientDC(this);
    OnPrepareDC(&ClientDC);
    CRect Rect=m_RectSample;
    ClientDC.LPtoDP(&Rect);
    InvalidateRect(&Rect);
    UpdateWindow();
}

void CFormDemoView::OnUpdateOptionsLight(CCmdUI* pCmdUI)
{
    //TODO:Add your command update UI handler code here
    pCmdUI->SetRadio(m_Intensity==INT_LIGHT);
}


void CFormDemoView::OnRed()
{
    //TODO:Add your control notification handler code here
    m_Red = IsDlgButtonChecked(IDC_RED);
    CClientDC ClientDC(this);
    OnPrepareDC(&ClientDC);
    CRect Rect=m_RectSample;
    ClientDC.LPtoDP(&Rect);
    InvalidateRect(&Rect);
    UpdateWindow();
}
```

```
void CFormDemoView::OnGreen()
{
    //TODO: Add your control notification handler code here
    m_Green = IsDlgButtonChecked( IDC_GREEN);
    CClientDC ClientDC(this);
    OnPrepareDC(&ClientDC);
    CRect Rect=m_RectSample;
    ClientDC.LPtoDP(&Rect);
    InvalidateRect(&Rect);
    UpdateWindow();
}

void CFormDemoView::OnBlue()
{
    //TODO: Add your control notification handler code here
    m_Blue = IsDlgButtonChecked(IDC_BLUE);
    CClientDC ClientDC(this);
    OnPrepareDC(&ClientDC);
    CRect Rect= m_RectSample;
    ClientDC.LPtoDP(&Rect);
    InvalidateRect(&Rect);
    UpdateWindow();
}
```

由于 Intensity 选项通过菜单命令而不是单选按钮设置，程序必须提供每个命令的更新处理程序（OnUpdateOptionsDark()和 OnUpdateOptionsLight()）。注意：更新处理程序调用 CCmdUI::SetRadio()而不是 CCmdUI::SetCheck()使所选菜单命令用项目条款而不是普通复选标记（项目符通常用于表示一组互相排斥菜单的项中的选择项）。

消息处理程序废除对话框中绘制色样的矩形区，但函数首先将样本区坐标从逻辑坐标换算为设备坐标，这一步是必要的。如果控件在视图窗口中放不下，用户可以滚动窗口显示内容，使存放在 m_RectSample 中的样本区逻辑坐标与设备坐标不同（InvalidateRect()要求传入设备坐标）。

（7）用 MFC Class Wizard 向视图类中加入 OnDraw()函数的更新版本（对于窗体视图程序，应用程序向导最初不包括 OnDraw()函数）。打开 MFC Class Wizard 对话框，设置 Class name 和 Object ID 为 CFormDemoView 类，设置 Message 为 OnDraw()函数，单击“Finish”按钮。然后加入语句如下：

```
void CFormDemoView::OnDraw(CDC *pDC)
{
    //TODO: Add your specialized code here and/or call the base class
    COLORREF Color=RGB(m_Red ? (m_Intensity==INT_DARK?128:255):0,
                       m_Green? (m_Intensity==INT_DARK?128:255):0,
                       m_Blue? (m_Intensity==INT_DARK?128:255):0)
    CBrush Brush(Color);
    pDC->FillRect(&m_RectSample,&Brush);
}
```

（8）在应用程序向导产生的 FormDemoView.cpp 文件中的 CFormView::OnInitialUpdate()函数中加入如下代码：

```
void CFormDemoView::OnInitialUpdate()
{
```

```
    CFormView::OnInitialUpdate();
    GetParentFrame()->RecalcLayout();
    ResizeParentToFit();

    GetDlgItem(IDC_SAMPLE)->GetWindowRect(&m_RectSample),
    ScreenToClient(&m_RectSample);
    int Border=(m_RectSample.right-m_RectSample.left)/8;
    m_RectSample.InflateRect(-Border,-Border);
}
```

视图类的 OnInitialUpdate()虚拟函数在新文档或现有文档首次显示在视图窗口之前调用。

调用 OnInitialUpdate()函数的基础类版本后，应用程序向导产生的代码调用主帧窗口对象的 RecalcLayout()函数，定位程序显示的控制条（但 FormDemo 中没有控制条）；然后应用程序向导产生的代码调用 CScrollView::ResizeParentToFit()函数，改变主帧窗口尺寸，以适合视图窗口内容。

调用 OnInitialUpdate()时，视图窗口还没有滚动，所以 m_RectSample 中存放的坐标既是样本区的设备坐标，也是其逻辑坐标。窗口滚动后，m_RectSample 中只保存逻辑坐标。

（9）在 FormDemoView.cpp 文件中，加入初始化数据成员 m_Intensity 的代码到视图类构造器中：

```
//CFormDemoView 构造/析构
CFormDemoView::CFormDemoView()
    : CFormView(CFormDemoView::IDD)
{
    // TODO: 在此处添加构造代码
    //AFX_DATA_INIT(CFormDemoView)
    m_Red=FALSE;
    m_Green=FALSE;
    m_Blue=FLASE;
    //AFX_DATA_INIT
    //TODO: add your construction code here
    m_Intensity=INT_LIGHT;
}
```

（10）在文件 FormDemoView.h 中，定义视图类的数据成员 m_DialogBrush、 m_Intensity 和 m_RectSample 并加入 INT_DARK 和 INT_LIGHT 枚举器的定义：

```
class CFormDemoView:public CFormView
{
public:
    CBrush m_DialogBrush;
    int m_Intensity;
    CRect m_RectSample;

    enum INT_DARK,INT_LIGHT;
}
```

（11）在 FormDemoView.cpp 文件中添加消息注册代码，如下所示：

```
BEGIN_MESSAGE_MAP(CFormDemoView, CFormView)
ON_COMMAND(ID_OPTIONS_DARK,OnOptionsDark)
ON_UPDATE_COMMAND_UI(ID_OPTIONS_DARK,OnUpdateOptionsDark)
ON_COMMAND(ID_OPTIONS_LIGHT,OnOptionsLight)
ON_UPDATE_COMMAND_UI(ID_OPTIONS_LIGHT,OnUpdateOptionsLight)
ON_BN_CLICKED(IDC_RED,OnRed)
```

```
ON_BN_CLICKED(IDC_GREEN,OnGreen)
ON_BN_CLICKED(IDC_BLUE,OnBlue)
END_MESSAGE_MAP()
```

（12）在 FormDemo.cpp 文件中，从函数 InitInstance()调用 SetWindowText()函数设置程序标题。

```
BOOL CFormDemoApp::InitInstance()
{
    // 如果一个运行在 Windows XP 上的应用程序清单指定要使用 ComCtl32.dll 版本或更高版本来启用可视化方式
    // 则需要 InitCommonControlsEx()；否则将无法创建窗口
    INITCOMMONCONTROLSEX InitCtrls;
    InitCtrls.dwSize = sizeof(InitCtrls);
    // 将它设置为包括所有要在应用程序中使用的公共控件类
    InitCtrls.dwICC = ICC_WIN95_CLASSES;
    InitCommonControlsEx(&InitCtrls);

    CWinAppEx::InitInstance();

    // 初始化 OLE 库
    if (!AfxOleInit())
    {
        AfxMessageBox(IDP_OLE_INIT_FAILED);
        return FALSE;
    }
    AfxEnableControlContainer();
    // 标准初始化
    // 如果未使用这些功能并希望减小最终可执行文件的大小，则应移除下列
    // 不需要的特定初始化例程
    // 更改用于存储设置的注册表项
    // TODO: 应适当修改该字符串，例如修改为公司或组织名
    SetRegistryKey(_T("应用程序向导生成的本地应用程序"));
    LoadStdProfileSettings(4);  //加载标准 INI 文件选项（包括 MRU）

    InitContextMenuManager();

    InitKeyboardManager();

    InitTooltipManager();
    CMFCToolTipInfo ttParams;
    ttParams.m_bVislManagerTheme = TRUE;
    theApp.GetTooltipManager()->SetTooltipParams(AFX_TOOLTIP_TYPE_ALL,RUNTIME_CLASS(CMFCToolTipC
trl), &ttParams);

    // 注册应用程序的文档模板
    CSingleDocTemplate* pDocTemplate;
    pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(CFormDemoDoc),
        RUNTIME_CLASS(CMainFrame),          // 主框架窗口
        RUNTIME_CLASS(CFormDemoView));
    if (!pDocTemplate)
        return FALSE;
    AddDocTemplate(pDocTemplate);
```

```
    // 分析标准外壳命令
    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);

    //调用命令行中指定的命令，如果用 RegServer、Register、Unregserver 或 Unregister 启用应用程序，则返回 False
    if (!ProcessShellCommand(cmdInfo))
        return FALSE;

    // 惟一的一个窗口已经初始化，因此显示它并对其进行更新
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();
    // 仅当具有后缀时才调用 DragAcceptFiles
    // 在 SDI 应用程序中，这应在 ProcessShellCommand 之后发生

    //The one and only window has been initialized, so show and update it.
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();
    m_pMainWnd->SetWindowText(_T("Form-View Demo"));
    return TRUE;
}
```

现在可以编译并运行 FormDemo 程序了。图 6.37 显示了程序首次运行时的 FormDemo 窗口；图 6.38 则显示了窗口缩小到小于对话框模板时程序将显示滚动条。

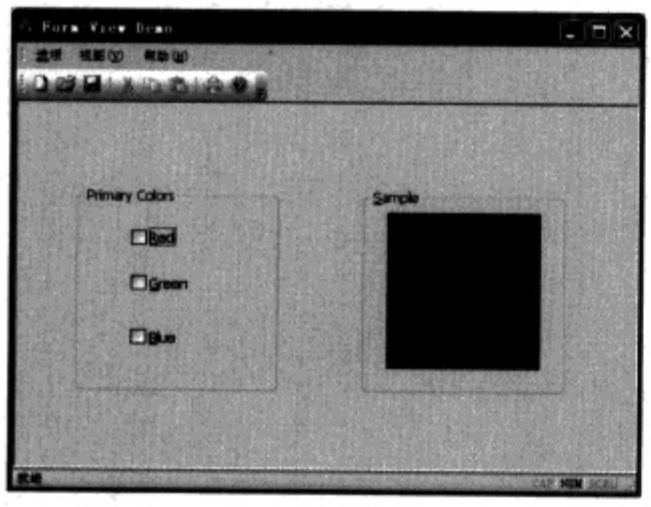


图 6.37　不带滚动条的运行结果

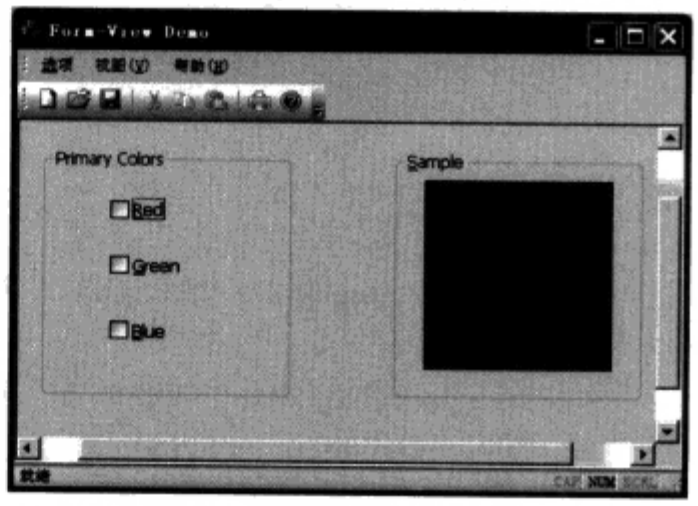


图 6.38　显示滚动条的运行结果

在程序运行结果中，随所选中的复选框的三基色的不同，Sample 中的颜色随之产生变化，如图 6.39 所示。

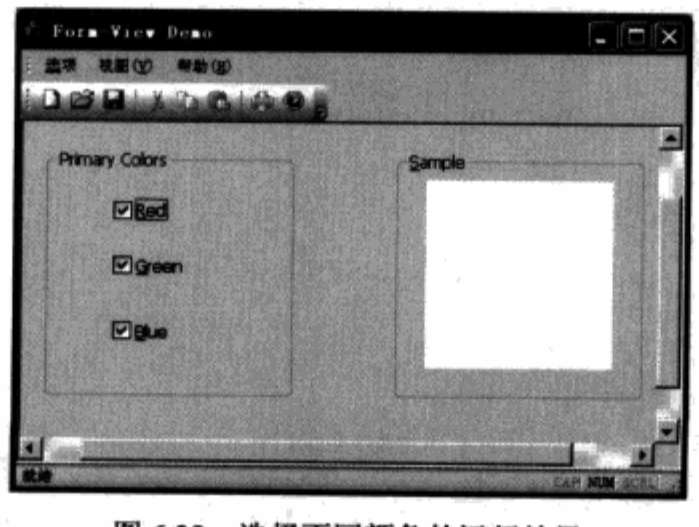


图 6.39　选择不同颜色的运行结果

6.5.2 程序清单

FormDemoView.h

```
// FormDemoView.h : CFormDemoView 类的接口
//
#pragma once
#include "afxwin.h"

class CFormDemoView : public CFormView
{
protected: //仅从序列化创建
    CFormDemoView();
    DECLARE_DYNCREATE(CFormDemoView)

public:
    enum{ IDD = IDD_FORMDEMO_FORM };

// 属性
public:
    CFormDemoDoc* GetDocument() const;

// 操作
public:

// 重写
public:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV 支持?
    virtual void OnInitialUpdate(); // 构造后第一次调用

// 实现
public:
    virtual ~CFormDemoView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// 生成的消息映射函数
protected:
    afx_msg void OnFilePrintPreview();
    afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnContextMenu(CWnd* pWnd, CPoint point);
    void OnOptionsDark();
```

```
        void OnUpdateOptionsDark(CCmdUI* pCmdUI);
        void OnOptionsLight();
        void OnUpdateOptionsLight(CCmdUI* pCmdUI);
        void OnRed();
        void OnGreen();
        void OnBlue();
        void OnDraw(CDC *pDC);
        DECLARE_MESSAGE_MAP()

    public:
        BOOL m_Red;
        BOOL m_Green;
        BOOL m_Blue;
        CBrush m_DialogBrush;
        int m_Intensity;
        CRect m_RectSample;

    enum {INT_DARK,INT_LIGHT};

    };

    #ifndef _DEBUG  // FormDemoView.cpp 中的调试版本
    inline CFormDemoDoc* CFormDemoView::GetDocument() const
       { return reinterpret_cast<CFormDemoDoc*>(m_pDocument); }
    #endif
```

FormDemoView.cpp

```
// FormDemoView.cpp : CFormDemoView 类的实现
//

#include "stdafx.h"
#include "FormDemo.h"

#include "FormDemoDoc.h"
#include "FormDemoView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CFormDemoView

IMPLEMENT_DYNCREATE(CFormDemoView, CFormView)

BEGIN_MESSAGE_MAP(CFormDemoView, CFormView)
ON_COMMAND(ID_OPTIONS_DARK,OnOptionsDark)
ON_UPDATE_COMMAND_UI(ID_OPTIONS_DARK,OnUpdateOptionsDark)
ON_COMMAND(ID_OPTIONS_LIGHT,OnOptionsLight)
```

```
ON_UPDATE_COMMAND_UI(ID_OPTIONS_LIGHT,OnUpdateOptionsLight)
ON_BN_CLICKED(IDC_RED,OnRed)
ON_BN_CLICKED(IDC_GREEN,OnGreen)
ON_BN_CLICKED(IDC_BLUE,OnBlue)

END_MESSAGE_MAP()

// CFormDemoView 构造/析构

CFormDemoView::CFormDemoView()
    : CFormView(CFormDemoView::IDD)
{
    // TODO: 在此处添加构造代码
    m_Red=FALSE;
    m_Green=FALSE;
    m_Blue=FALSE;
    //AFX_DATA_INIT
    //TODO: add your construction code here
    m_Intensity=INT_LIGHT;
}
CFormDemoView::~CFormDemoView()
{
}
void CFormDemoView::DoDataExchange(CDataExchange* pDX)
{
    CFormView::DoDataExchange(pDX);
    DDX_Check(pDX, IDC_RED, m_Red);
    DDX_Check(pDX, IDC_GREEN, m_Green);
    DDX_Check(pDX, IDC_BLUE, m_Blue);
}

BOOL CFormDemoView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: 在此处通过修改 CREATESTRUCT cs 来修改窗口类或样式

    return CFormView::PreCreateWindow(cs);
}
void CFormDemoView::OnInitialUpdate()
{
    CFormView::OnInitialUpdate();
    GetParentFrame()->RecalcLayout();
    ResizeParentToFit();
    CFormView::OnInitialUpdate();
    GetDlgItem(IDC_SAMPLE)->GetWindowRect(&m_RectSample),ScreenToClient(&m_RectSample);
    int Border=(m_RectSample.right-m_RectSample.left)/8;
    m_RectSample.InflateRect(-Border,-Border);
}

void CFormDemoView::OnRButtonUp(UINT /* nFlags */, CPoint point)
```

```
{
    ClientToScreen(&point);
    OnContextMenu(this, point);
}

void CFormDemoView::OnContextMenu(CWnd* /* pWnd */, CPoint point)
{
    theApp.GetContextMenuManager()->ShowPopupMenu(IDR_POPUP_EDIT, point.x, point.y, this, TRUE);
}
// CFormDemoView 诊断

#ifdef _DEBUG
void CFormDemoView::AssertValid() const
{
    CFormView::AssertValid();
}
void CFormDemoView::Dump(CDumpContext& dc) const
{
    CFormView::Dump(dc);
}

CFormDemoDoc* CFormDemoView::GetDocument() const // 非调试版本是内联的
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CFormDemoDoc)));
    return (CFormDemoDoc*)m_pDocument;
}
#endif //_DEBUG

// CFormDemoView 消息处理程序

//CFormDemoView message Handlers

void CFormDemoView::OnOptionsDark()
{
    //TODO:Add your command handler code here
    m_Intensity=INT_DARK;
    CClientDC ClientDC(this);
    OnPrepareDC (&ClientDC);
    CRect Rect=m_RectSample;
    ClientDC.LPtoDP(&Rect);
    InvalidateRect(&Rect);
    UpdateWindow();
}
void CFormDemoView::OnUpdateOptionsDark(CCmdUI* pCmdUI)
{
    //TODO:Add your command update UI handler code here
    pCmdUI->SetRadio (m_Intensity==INT_DARK);
}
void CFormDemoView::OnOptionsLight()
```

```
{
    //TODO:Add your command handler code here
    m_Intensity =INT_LIGHT;
    CClientDC ClientDC(this);
    OnPrepareDC(&ClientDC);
    CRect Rect=m_RectSample;
    ClientDC.LPtoDP(&Rect);
    InvalidateRect(&Rect);
    UpdateWindow();
}
void CFormDemoView::OnUpdateOptionsLight(CCmdUI* pCmdUI)
{
    //TODO:Add your command update UI handler code here
    pCmdUI->SetRadio(m_Intensity==INT_LIGHT);
}
void CFormDemoView::OnRed()
{
    //TODO:Add your control notification handler code here
    m_Red=IsDlgButtonChecked(IDC_RED);
    CClientDC ClientDC(this);
    OnPrepareDC(&ClientDC);
    CRect Rect=m_RectSample;
    ClientDC.LPtoDP(&Rect);
    InvalidateRect(&Rect);
    UpdateWindow();
}

void CFormDemoView::OnGreen()
{
    //TODO: Add your control notification handler code here
    m_Green = IsDlgButtonChecked( IDC_GREEN);
    CClientDC ClientDC(this);
    OnPrepareDC(&ClientDC);
    CRect Rect=m_RectSample;
    ClientDC.LPtoDP(&Rect);
    InvalidateRect(&Rect);
    UpdateWindow();
}
void CFormDemoView::OnBlue()
{
    //TODO: Add your control notification handler code here
    m_Blue = IsDlgButtonChecked(IDC_BLUE);
    CClientDC ClientDC(this);
    OnPrepareDC(&ClientDC);
    CRect Rect= m_RectSample;
    ClientDC.LPtoDP(&Rect);
    InvalidateRect(&Rect);
    UpdateWindow();
}
```

```
void CFormDemoView::OnDraw(CDC *pDC)
{
    //TODO: Add your specialized code here and/or call the base class
    COLORREF Color=RGB(m_Red ? (m_Intensity==INT_DARK?128:255):0,
                       m_Green? (m_Intensity==INT_DARK?128:255):0,
                       m_Blue? (m_Intensity==INT_DARK?128:255):0);
    CBrush Brush(Color);
    pDC->FillRect(&m_RectSample,&Brush);
}
```

6.6　Visual C++ 2010 ScrollView 开发

本节将介绍如何给 MFC 程序窗口加入滚动功能。滚动功能使用户可以阅读和编辑超出视图窗口文档的任何部分。完成本节的学习之后，可以发现这个特性加入程序中相对比较容易，几乎所有的工作都是用支持这些特性的专用 MFC 类完成的。

为了演示编程技巧，本节介绍如何向 6.3 节中的 MiniDraw 程序中加入滚动功能和分隔功能，这些特性应加到 6.3 节生成的 MiniDraw 源文件中。如果没有生成这些文件，可以从本书配套光盘的相应文件夹中复制。

6.6.1　加入滚动功能

在上一版 MiniDraw 程序中，如果图形大于视图窗口的当前尺寸，则用户只能查阅和编辑视图窗口内的图形部分。这里要向视图窗口加入垂直和水平滚动条及支持滚动的代码，使用户可以查阅和编辑超出窗口的部分。要加入的 MiniDraw 滚动条如图 6.40 所示。

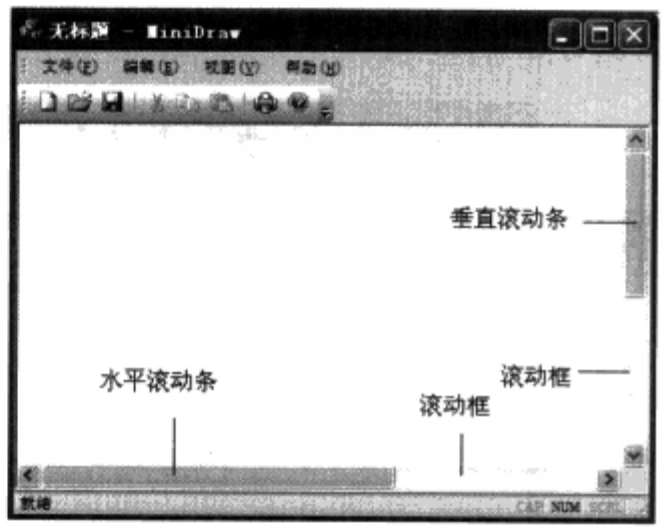


图 6.40　MiniDraw 的滚动条

MiniEdit 程序不需要加入滚动条或者滚动功能，这是因为派生程序类的 MFC CEditView 类能做这个工作。

打开 MiniDraw 项目后，首先要修改 MiniDraw 视图类，使它从 CScrollView 而不是从 CView 派生。和 MiniEdit 程序中用于生成编辑窗口的 CEditView 类一样，CScrollView 也是从 CView 类派生的专用视图类。从 CScrollView 派生视图类就自动将滚动条加入了视图窗口，并提供了支持滚动操作的大多数代码 ，但还要开发人员自己加入一些支持代码。

为了修改视图类，使其从 CScrollView 派生，只需要将 MiniDrawView.h 和 MiniDrawView.cpp 源文件中出现的所有 CView 类换成 CScrollView 类即可。如果文件符合书中的程序清单，则 CView 在 MiniDrawView.h 中出现了 1 次，在 MiniDrawView.cpp 中出现了 10 次。

本节介绍如何修改现有的程序，使其视图类从 CScrollView 派生。但生成新程序时，可让应用程序向导从 CScrollView 派生视图类，为此，在应用程序向导“Generated Classes”对话框中，从顶部的列表框中选择视图类名，然后在 Base class 下拉列表框中选择 CScrollView 类即可。

6.6.2　坐标换算

继续修改程序之前，要对 CScrollView 代码支持滚动的方式有一些了解。文档首次打开时，图形左

上角位于窗口左上角，就像前面的 MiniDraw 版本一样。但如果用户用滚动条滚动文档，则 MFC 调整一个称为视图原点的属性。

视图原点确定所画的图形或文本相对于窗口的位置。通常，如果在坐标（0, 0）上画点，则它显示在视图窗口的左上角；如果在坐标（50, 100）上画点，则它会显示在视图窗口中离左边沿 50 像素，离顶边 100 像素的位置。但如果用户用垂直滚动条将文档向下滚动 75 个像素，则 MFC 调整视图原点，使两个点都出现在相对于窗口的较高位置。这时已经看不到点（0, 0）（窗口外的图形被剪去，即放弃掉），而点（50, 100）显示在离窗口顶边 25 像素的位置。图 6.41 和图 6.42 分别显示了滚动前后的视图窗口。

MFC 响应滚动动作调整视图原点之后，MiniDraw 的 OnDraw()函数在视图窗口重绘直线。每条线的坐标保持原值，但由于视图原点变了，直线自动显示在滚动后的相应位置。这个系统的精彩之处在于，OnDraw()函数不需要改变就可以支持滚动，滚动逻辑是由 MFC 悄悄处理的。

绘制对象时，指定的坐标称为逻辑坐标，对象在窗口中的实际坐标称为设备坐标，如图 6.43 所示。传入 MFC 绘图函数（如 MoveTo()和 LineTo()）的坐标都是逻辑坐标。但 MFC 使用的某些其他坐标（如传入鼠标消息的光标位置）则是设备坐标。程序中加入滚动显示功能之前，逻辑坐标与设备坐标完全相等。两者之间的差别无关紧要。程序加入滚动显示功能之后，却必须对代码进行一些修改，以便于在两种坐标之间进行换算。

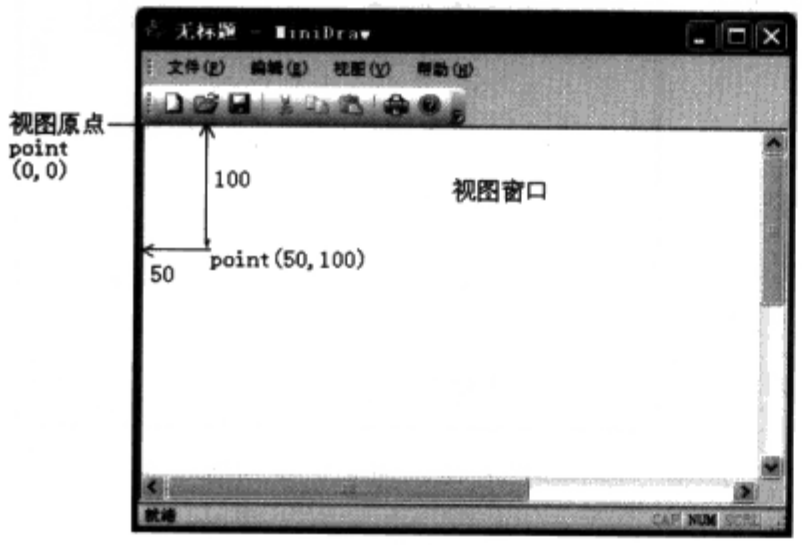


图 6.41　滚动前的视图窗口

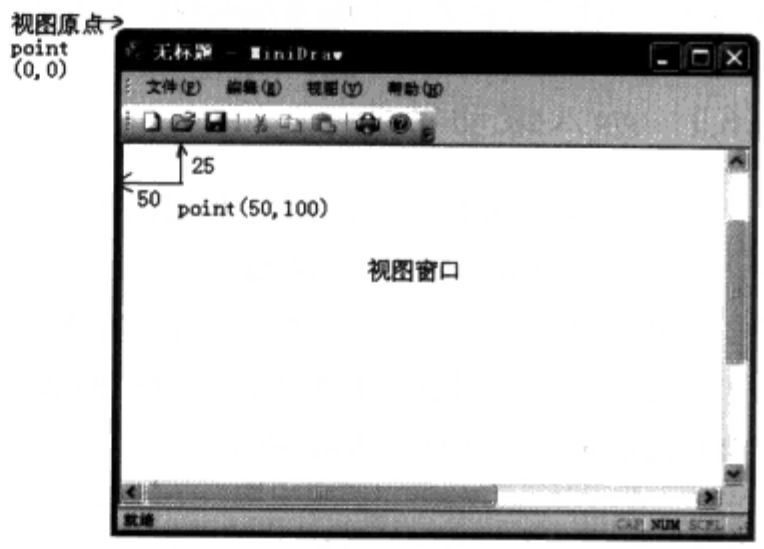


图 6.42　滚动后的视图窗口

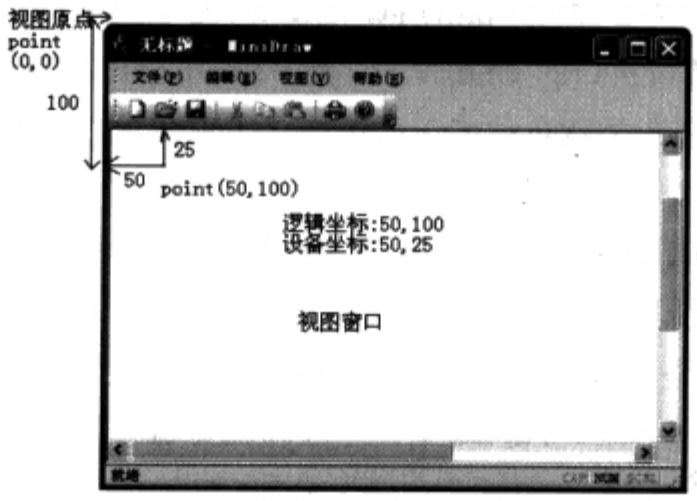


图 6.43　逻辑坐标与设备坐标（图中用户向下拖动了 75 个像素）

具体来说，需要传入鼠标消息处理函数的鼠标指针位置从设备坐标换算为逻辑坐标，使直线在图中相应的位置画出。为此，首先在 MiniDrawView.cpp 文件的 OnLButtonDown()函数中加入下列黑体代码：

```
void CMiniDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    //TODO:Add your message handler code here and /or call
    //default

    CClientDC ClientDC(this);
    OnPrepareDC(&ClientDC);
    ClientDC.DPtoLP(&point);

    m_PointOrigin = point;
    m_PointOld = point;
    SetCapture();
    m_Dragging =1;

    RECT Rect;
    GetClientRect ( & Rect);
    ClientToScreen(&Rect);
    ::ClipCursor(&Rect);

    CScrollView::OnLButtonDown(nFlags, point);
}
```

为了将特定设备的设备坐标换算为逻辑坐标，必须使用与设备相关的设备描述表。设备描述表管理向设备的输出，存放图形属性并提供显示文本或图形的成员函数。加入的第一条语句生成与视图窗口相关的设备描述表；第二条语句调用 CScrollView 成员函数 OnPrepareDC()，它根据图形的当前滚动位置调整设备描述表的视图原点（视图原点是设备描述表对象存放的一个图像属性）；第三条语句调用 CDC 成员函数 DPtoLP()，它将 point 中存放的光标位置用刚刚设置的新视图原点从设备坐标换算为设备描述表的逻辑坐标。一旦 point 换算成逻辑坐标，它所包含的坐标即可用于画直线。

传递到 OnDraw()函数的设备描述表的视图原点已经调整至图形的当前滚动位置，但生成自己的设备描述表时，必须将它传入函数 OnPrepareDC()，调整视图原点后再使用。

同样，需要修改 OnMouseMove()和 OnLButtonUp()鼠标处理函数，这两个函数已经生成设备描述表，所以只要调用 OnPrepareDC()和 DPtoLP()即可。这些语句在 OnMouseMove()中的使用如下：

```
void CMiniDrawView::OnMouseMove(UINT nFlags,CPoint point)
{
    //TODO:Add your message handler code here and /or call
    //default
    ::SetCursor(m_HCross);

    if(m_Dragging)
    {
        CClientDC ClientDC(this);
        OnPrepareDC(&ClientDC);
        ClientDC.DPtoLP(&point);
        ClientDC.SetROP2(R2_NOT);
        ClientDC.MoveTo(m_PointOrigin);
        ClientDC.LineTo(m_PointOld);
        ClientDC.MoveTo(m_PointOrigin);
```

```
            ClientDC.LineTo(point);
            m_PointOld=point;
        }
        CScrollView::OnMouseMove(nFlags,point);
}
```

语句在 OnLButtonUp()函数中的使用如下：

```
void CMiniDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
    //TODO: Add your message handler code here and /or call default
    if(m_Dragging)
    {
        m_Dragging=0;
        ::ReleaseCapture();
        ::ClipCursor(NULL);
        CClientDC ClientDC(this);
        OnPrepareDC(&ClientDC);
        ClientDC.DPtoLP(&point);
        ClientDC.SetROP2(R2_NOT);
        ClientDC.MoveTo(m_PointOrigin);
        ClientDC.LineTo(m_PointOld);
        ClientDC.SetROP2(R2_COPYPEN);
        ClientDC.MoveTo(m_PointOrigin);
        ClientDC.LineTo(point);
    }
    CScrollView::OnLButtonUp(nFlags,point);
}
```

6.6.3 限制图形大小

用户将滚动框移动到滚动条的特定位置时，CScrollView 中的 MFC 代码必须滚动到图形的对应位置。例如，如果用户拖动滚动框到垂直滚动条的底部，则 MFC 需滚动到图形的底部；同样，用户拖动滚动框到水平滚动条的右端时，MFC 也要滚动到图形的右端。因此，MFC 必须知道图形的总体尺寸。

本节要向 MiniDraw 加入向 MFC 报告图形尺寸的代码。首先在程序视图类中定义 OnInitialUpdate()虚拟函数的覆盖版本。在 MFC Class Wizard 对话框中，设置 Class name 和 Object ID 为 CMiniDrawView 类，设置 Message 为 OnInitialUpdate()。加入函数并单击“Edit Code”按钮后，在 MiniDrawView.cpp 的函数定义中加入下列代码：

```
void CMiniDrawView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();

    CSize sizeTotal;
                                        // TODO: 计算此视图的合计大小

    sizeTotal.cx =640 ;
    sizeTotal.cy = 480;
    SetScrollSizes(MM_TEXT, sizeTotal);
}
```

MFC 在新的或现有文档首次在视图窗口显示前调用视图类的 OnInitialUpdate()虚拟函数。

调用 CScrollView 成员函数 SetScrollSizes()，告诉 MFC 图形的尺寸。图形的水平尺寸和垂直尺寸赋予 SIZE 结构，作为第二个参数传递。在当前版本的 MiniDraw 程序中，图形设置成固定尺寸：640 像素宽和 480 像素高(这个随意选择的尺寸使得图形与标准 VGA 方式的整个 Windows 屏幕大小相同)。

完全特性化的绘图程序通常可以设置每个新图形的尺寸，这在程序要打印图形时特别重要，它能使用户生成符合打印页的尺寸的图形。用户指定图形大小后，尺寸存放在文档类中；图形保存到文件中时，该尺寸应该和其他数据一起保存。

另外，在 MiniDraw 程序中，文档是定长的，所以尺寸只需指定一次。如果用户输入或删除数据时应用程序（如字处理程序）改变文档长度，则每次改变长度时都应调用 SetScrollSizes()，这可以在视图类的 OnUpdate()成员函数中完成，这是因为每次改变文档数据时都要调用这个函数。

传入 SetScrollSizes()的第一个参数指示使用的映射方式。和视图原点一样，映射方式也是设备描述表维护的图形属性。映射方式指定用于绘制文本和图形的单位及坐标系统。MiniDraw 程序采用 MM_TEXT 映射方式。在 MM_TEXT 映射方式中，所有单位都用像素，水平坐标从左向右增大，垂直坐标从上向下增大（如图 6.44 所示）。

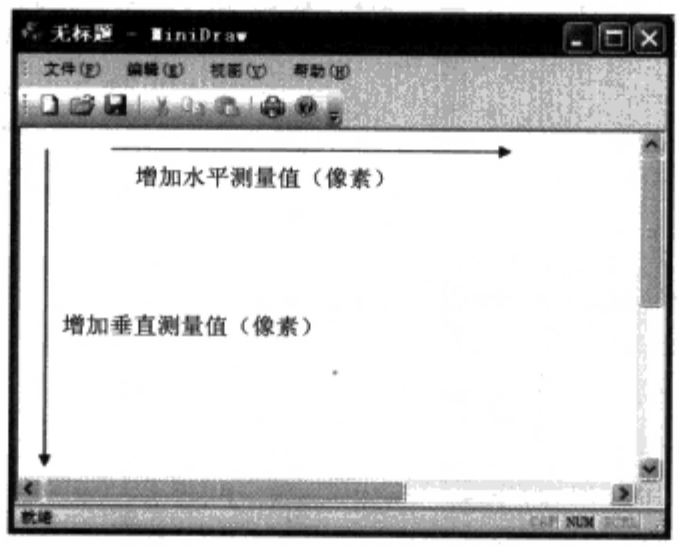


图 6.44　MM_TEXT 映象方式

如果用户单击按钮或者滚动块外的滚动条部分，则视图窗口滚动一页；默认情况下，一页等于文档尺寸的 1/10。如果用户单击滚动条上的滚动块，则视图窗口滚动一行；默认情况下，一行是一页的 1/10。如果要改变滚动的距离，可以将所需的页长或行长作为第三和第四个参数传递给 SetScrollSizes()，详见函数文档。

如果从 OnInitialUpdate()函数中调用 CScrollView::SetScaleToFitSize()而不是 SetScrollSizes()，则文本和图形缩放成适合整个文档当前视图窗口的大小。由于缩放不再需要滚动，滚动条消息详见函数文档。

由于 MiniDraw 图形有特定的尺寸，所以要在程序中加入特性，防止用户在图形区外画直线（根据当前所写的程序，如果视图窗口大于图形，则用户可以在图形区外画直线；滚动无法正确处理这些直线）。首先要在 OnDraw()函数中加入重绘视图窗口时绘制图形区右端和下端边框的代码，这些边框告诉程序用户图形的边界。为此，在 MiniDrawView.cpp 文件的 OnDraw()函数中加入下列代码中的黑体行：

```
//CMiniDrawView Drawing
void CMiniDrawView::OnDraw(CDC *pDC)
{

    CMiniDrawDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
```

```
        return;

    // TODO: 在此处为本机数据添加绘制代码
    CSize ScrollSize =GetTotalSize();
    pDC->MoveTo(ScrollSize.cx,0);
    pDC->LineTo(ScrollSize.cx, ScrollSize.cy);
    pDC->LineTo(0,ScrollSize.cy);
}
```

新代码首先调用 CScrollView 成员函数 GetTotalSize()，返回包含图形尺寸的 CSize 对象（即包含调用 SetScrollSizes()函数设置的图形尺寸）；然后，程序用 CSize 对象中包含的尺寸在图形右端和下端画直线。

接着，要在 OnLButtonDown()消息处理器中加入程序，防止用户将直线画在图形区外，为此，改写 MiniDrawView.cpp 文件中的 OnLButtonDown()为如下形式（新行和改写行标成黑体）：

```
// CMiniDrawView 消息处理程序
void CMiniDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    //TODO:Add your message handler code here and /or call
    //default

    CClientDC ClientDC(this);
    OnPrepareDC(&ClientDC);
    ClientDC.DPtoLP(&point);
    //test whether cursor is within drawing area of viewing window
    CSize ScrollSize=GetTotalSize();
    CRect ScrollRect(0,0,ScrollSize.cx,ScrollSize.cy);
    if(!ScrollRect.PtInRect(point))
        return;

    //save cursor position ,capture mouse and set dragging flag

    m_PointOrigin = point;
    m_PointOld = point;
    SetCapture();
    m_Dragging =1;

    //clip mouse cursor
    ClientDC.LPtoDP(&ScrollRect);

    CRect ViewRect;
    GetClientRect(&ViewRect);
    CRect IntRect;
    IntRect.IntersectRect(&ScrollRect,&ViewRect);
    ClientToScreen(&IntRect);
    ::ClipCursor(&IntRect);

    CScrollView::OnLButtonDown(nFlags, point);
}
```

由于视图类接受鼠标消息，鼠标光标必须在视图窗口中，但光标可能在绘图区外。为了测定光标是否在图形内，加入的代码定义 ScrollRect 对象，它是 MFC 类的 CRect 的实例，尺寸用图形的尺寸（调

用 GetTotalSize()取得）。然后，程序调用 ScrollRect 对象的 CRect::PtInRect()函数，传入光标的坐标。只有光标坐标在 ScrollRect 中存放的矩形区内时，PtInRect()才返回 True。如果光标在图形区域外，函数立即返回，所以用户无法在图形区外画图。

除了防止用户从图形区外开始画线外，OnLButtonDown()还需要防止用户将直线延伸出图形区外。因此，加入的代码将光标限制在本图形区内的视图窗口，即将光标限制在图形区和视图窗口的相交部分。为此，加入的代码首先调用 ClientDC 对象的 CDC::LPtoDP()函数，将存放在 ScrollRect 中的图形坐标从逻辑坐标转换成设备坐标。然后，定义 CRect 对象 ViewRect，赋予它视图窗口的设备坐标（调用 GetClientRect()取得）。接着定义另一个 CRect 对象 IntRect，并调用这个对象的 CRect::InterSectRect()赋予它图形区和视图窗口区相交部分的坐标。最后，调用 CWnd::ClientToScreen()函数以将 IntRect 换算为屏幕坐标，并将 IntRect 传递给::ClipCursor()，将光标限制在相交区，这些矩形区如图 6.45 和图 6.46 所示。

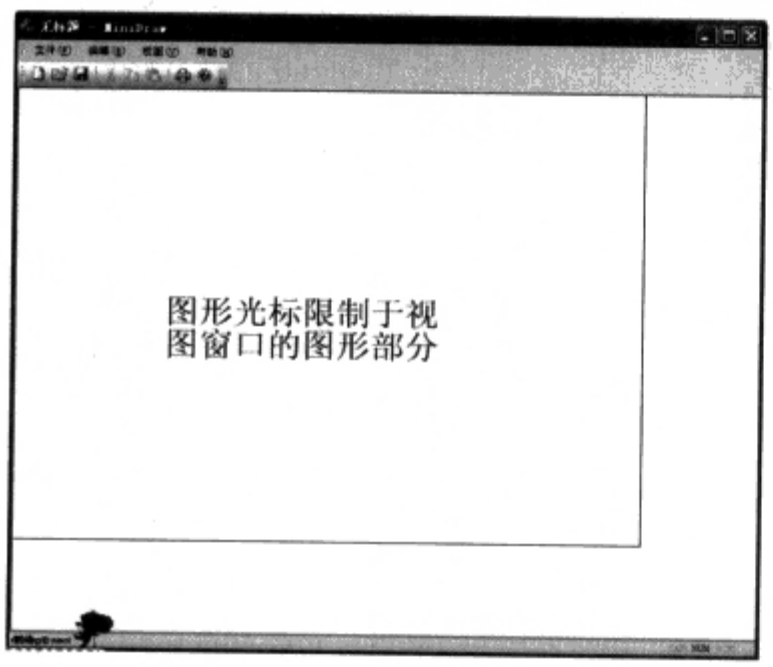


图 6.45　图形小于视图窗口

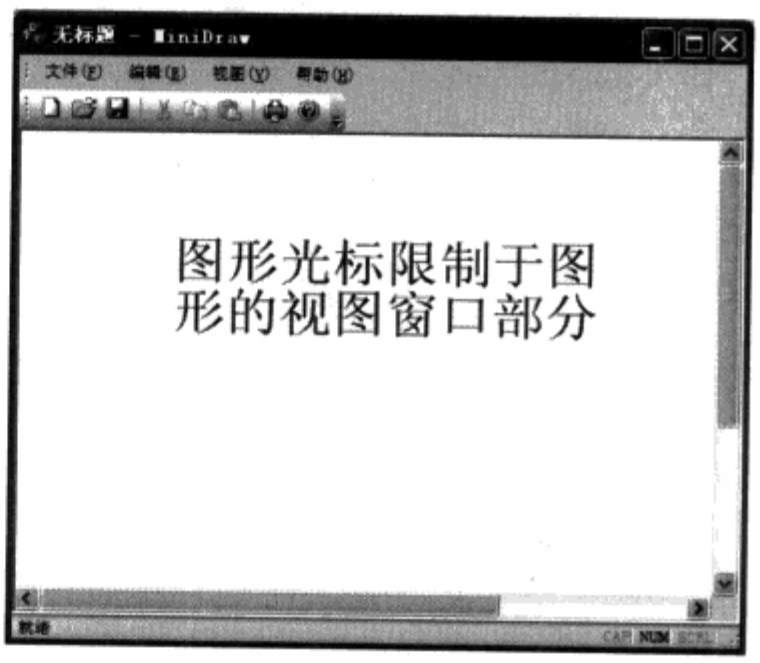


图 6.46　图形大于视图窗口

图形小于视图窗口时，光标限制于图形区内；而图形大于视图窗口时光标限制于视图窗口内。总之，光标限制于图形区和视图窗口相交的区域。

6.6.4　改变鼠标光标

修改程序使鼠标光标位于视图窗口的绘图区时呈现十字形（表示可以画直线），而位于绘图区外时呈现标准箭头形状（表示不能画直线）。为此，在 MiniDrawView.h 的 CMiniDrawView 类定义中加入数据成员 m_HArrow：

```
class CMiniDrawView : public CScrollView
{
protected:
    CString m_ClassName;
    int m_Dragging;
    CURSOR m_HArrow;
    CURSOR m_HCross;
    CPoint m_PointOld;
    CPoint m_PointOrigin;

protected:                              // 仅从序列化创建
    CMiniDrawView();
    DECLARE_DYNCREATE(CMiniDrawView)
```

```
                                        // 属性
public:
    CMiniDrawDoc* GetDocument() const;
                                        // 操作
public:
                                        // 重写
public:
    virtual void OnDraw(CDC* pDC);  // 重写以绘制视图
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
    void OnLButtonDown(UINT nFlags, CPoint point);
    void OnMouseMove(UINT nFlags,CPoint point);
    void OnLButtonUp(UINT nFlags, CPoint point);
    void OnInitialUpdate();

                                        // 实现
public:
    virtual ~CMiniDrawView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

                                        // 生成的消息映射函数
protected:
    afx_msg void OnFilePrintPreview();
    afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnContextMenu(CWnd* pWnd, CPoint point);
    DECLARE_MESSAGE_MAP()
};
```

并在 MiniDrawView.cpp 的 CMiniDrawView 构造器中初始化 m_HArrow 如下：

```
                                        // CMiniDrawView 构造/析构

CMiniDrawView::CMiniDrawView()
{
                                        // TODO: 在此处添加构造代码
    m_Dragging=0;
    m_HArrow=AfxGetApp()->LoadStandardCursor(IDC_ARROW);
    m_HCross=AfxGetApp()->LoadStandardCursor(IDC_CROSS);

}
```

初始化语句赋予 m_HArrow 标准箭头形状的鼠标光标的句柄。

接着，在 MiniDrawView.cpp 的 OnMouseMove()函数中，将 ClientDC()的声明和 OnPrepareDC()与 DPtoLP()的调用移到函数的开头，并加入黑体行，使函数定义如下：

```
void CMiniDrawView::OnMouseMove(UINT nFlags,CPoint point)
{
                                    //TODO:Add your message handler code here and /or call
                                                                        //default
::SetCursor(m_HCross);

if(m_Dragging)
{
    CClientDC ClientDC(this);
    OnPrepareDC(&ClientDC);
    ClientDC.DPtoLP(&point);

    CSize ScrollSize=GetTotalSize();
    CRect ScrollRect(0,0,ScrollSize.cx,ScrollSize.cy);
    if(ScrollRect.PtInRect(point))
        ::SetCursor(m_HCross);
    else
        ::SetCursor(m_HArrow);

    ClientDC.SetROP2(R2_NOT);
    ClientDC.MoveTo(m_PointOrigin);
    ClientDC.LineTo(m_PointOld);
    ClientDC.MoveTo(m_PointOrigin);
    ClientDC.LineTo(point);
    m_PointOld=point;
}
CScrollView::OnMouseMove(nFlags,point);
}
```

新版 OnMouseMove()不是总显示十字形光标，而是光标在绘图区内时显示十字形光标，在绘图区外时显示箭头形光标。

图 6.47 演示了绘图区小于视图窗口时的 MiniDraw 程序，显示了表示图形区边沿的直线，该图中，由于绘图区完全位于视图窗口内，MFC 代码覆盖了滚动条（窗口尺寸减少到不能完全放下图形时，滚动条会重新显示）。

编译并运行 MiniDraw 程序，来测试新加入的功能。

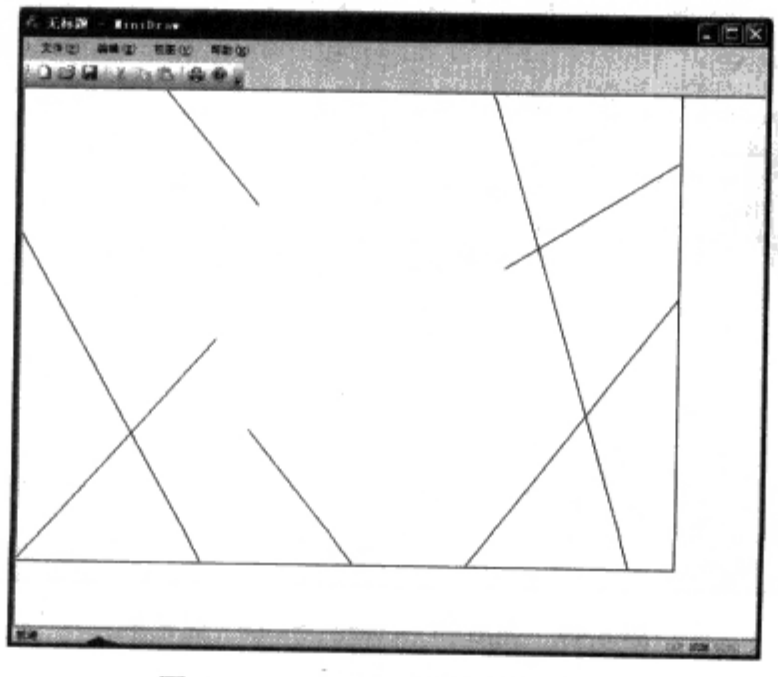


图 6.47　MiniDraw 程序运行结果

6.7 Visual C++ 2010 HtmlEditView 开发

为了允许对 HTML 文件进行编辑，微软在 Visual Studio 2010 中提供了 CHtmlEditCtrl 和 CHtmlEditView 两个类，这两个类都是从 CWnd 继承而来的。

本节用应用程序向导生成 MiniHtmlEdit 程序，其中视图类要从 CHtmlEditView 而不是从 CView 中派生，因而生成视图窗口内方便的网络编辑器，然后用 Visual Studio 菜单编辑器加入菜单命令，使用户能够访问一些编辑器命令。还要用 Visual Studio 资源编辑器加入快捷键和设计程序图标。

MiniHtmlEdit 程序窗口如图 6.48 所示，程序可以用于在视图窗口中输入网络地址，并打开和显示网页，有滚动条，可以在文本无法全部放入窗口时滚动文本。程序菜单包含打印文本，取消最后编辑动作，剪切、复制和粘贴文本，选择文本并完成查找与替换操作的命令。如果在视图窗口中右击，程序还会显示包含大多数“Edit”菜单命令的浮动菜单。

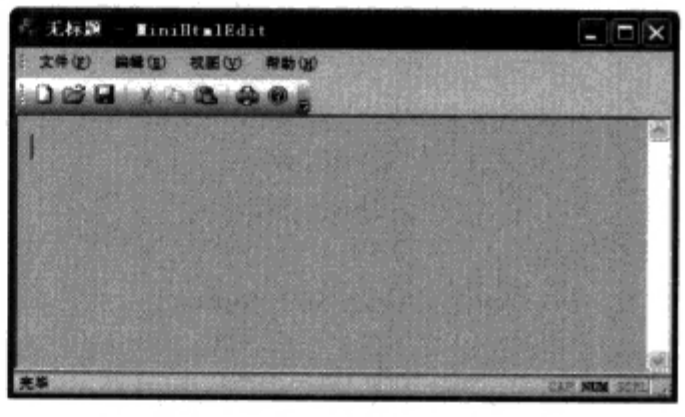


图 6.48 MiniHtmlEdit 程序窗口

生成 MiniHtmlEdit 程序时，首先像 6.3 节的 MiniDraw 程序那样，用应用程序向导生成源代码文件。在“New Project”对话框中，“Name”文本框中输入 MiniHtmlEdit，“Location”文本框中输入项目文件夹路径，在此选择默认存储路径 C:\Documents and Settings\Administrator\My Documents\Visual Studio 10\Projects。

在应用程序向导对话框中，设置与生成前面 MiniDraw 例子程序相同的选项，但在“Generated Classes”对话框中，不是接受所有的默认值，而是要按照如下的步骤进行设置。

（1）选择对话框顶部的列表框中的类名 CMiniHtmlEditView。

（2）在“Base class”下拉列表框中选择 CHtmlEditView，完成的“Generated Classes”对话框如图 6.49 所示。

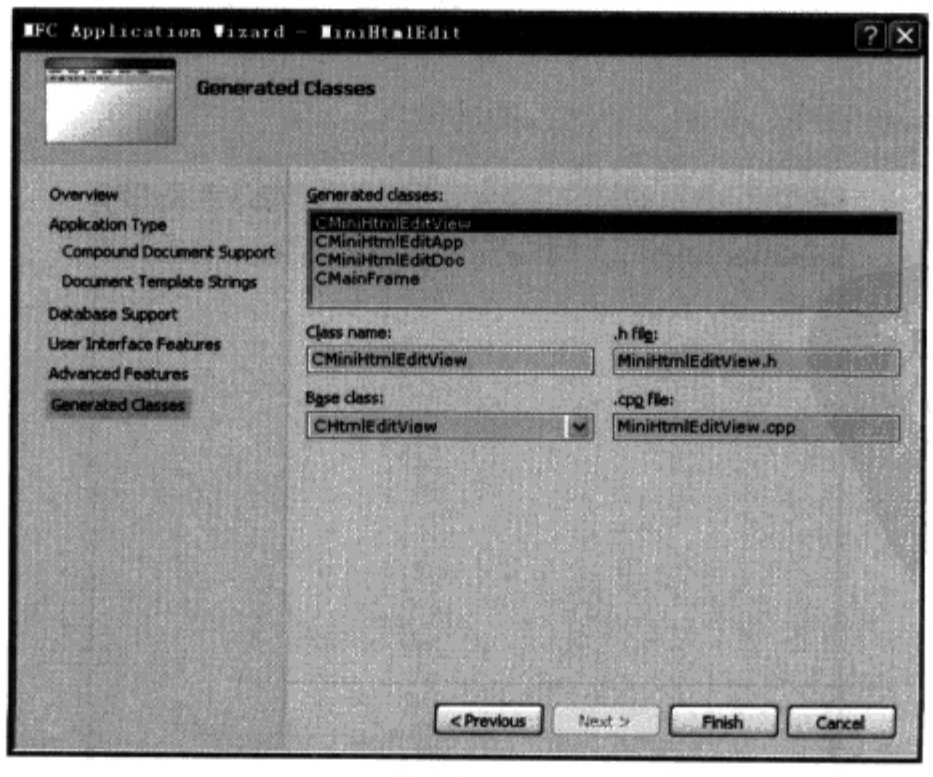


图 6.49 完成的 Generated Class 对话框

（3）最后，单击“Finish”按钮结束，生成源代码。

编译并运行生成的 MiniHtmlEdit 程序，运行结果如图 6.50 所示。

可以发现，运行后的 MiniHtmlEdit 窗口具有输入和编辑功能。在其中输入“Visual C++ 2010 MFC 文档视图程序开发”，如图 6.51 所示。

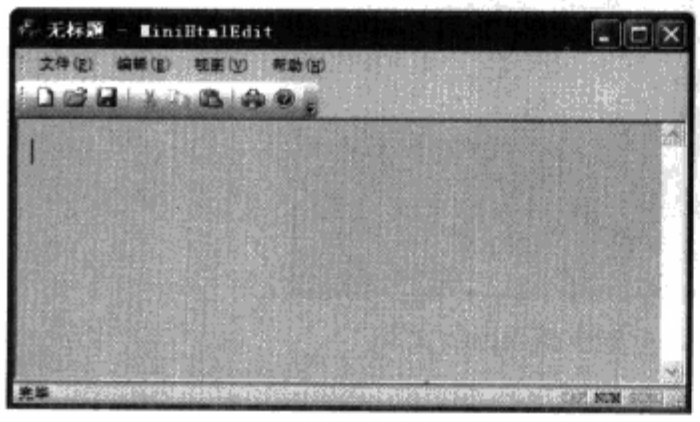


图 6.50　MiniHtmlEdit 程序运行结果

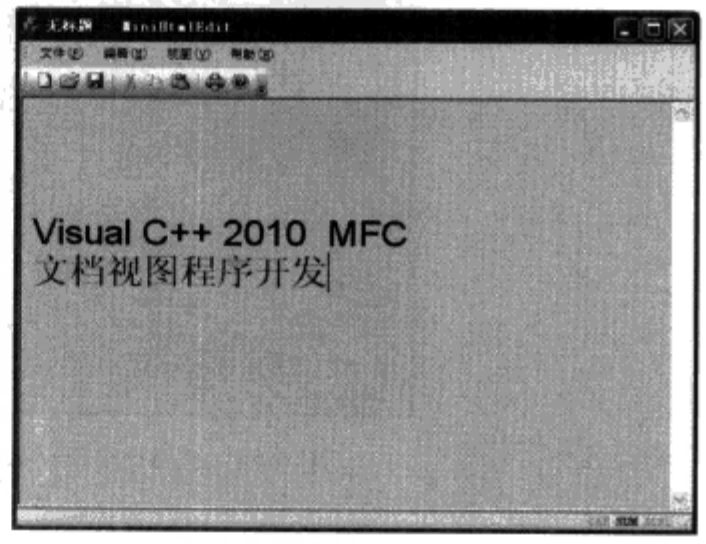


图 6.51　输入文字后的运行结果

在这些字上面右击，选择“查看源文件”命令，则弹出 about_blank[1]的文本框，内容如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD>
<META http-equiv=Content-Type content="text/html; charset=unicode">
<META content="MSHTML 6.00.2900.3314" name=GENERATOR></HEAD>
<BODY>
<P class=11 style="MARGIN: 6pt 121.5pt 6pt 0cm"><FONT size=6><FONT
face=Arial><SPAN lang=EN-US></SPAN></FONT></FONT> </P>
<P class=11 style="MARGIN: 6pt 121.5pt 6pt 0cm"><FONT size=6><FONT
face=Arial><SPAN lang=EN-US></SPAN></FONT></FONT> </P>
<P class=11 style="MARGIN: 6pt 121.5pt 6pt 0cm"><FONT size=6><FONT
face=Arial><SPAN lang=EN-US>Visual C++ 2010<SPAN
style="mso-spacerun: yes">  </SPAN>MFC<SPAN
style="mso-spacerun: yes">   </SPAN></SPAN><SPAN
style="FONT-FAMILY: 汉仪中黑简; mso-ascii-font-family: Arial">文档视图程序开发</SPAN></FONT></FONT>
</P></BODY></HTML>
```

这是一段 HTML 的脚本代码，可以通过修改该文本框中的代码来实现修改运行后的程序框中的文字的功能。例如，将

```
style="FONT-FAMILY: 汉仪中黑简; mso-ascii-font-family: Arial">文档视图程序开发</SPAN></FONT></FONT>
</P></BODY></HTML>
```

修改为：

```
style="FONT-FAMILY:华文彩云; mso-ascii-font-family: Arial">文档视图程序开发</SPAN></FONT></FONT>
</P></BODY></HTML>
```

此外，在空白处右击，可插入图标和编辑超链接的功能。例如，右击并单击插入图片选项，插入图片后，运行结果如图 6.52 所示。

另外，MiniHtmlEdit 运行文件中，还可以编辑和显示超链接。例如，在程序运行框中输入 www.baidu.com，然后在超链接上右击，选择“浏览视图”命令，此时鼠标指针再次停留在域名上时，就会变成一个小手状，表示此时可以打开此超链接。我们单击该地址，则打开网页，如图 6.53 所示。

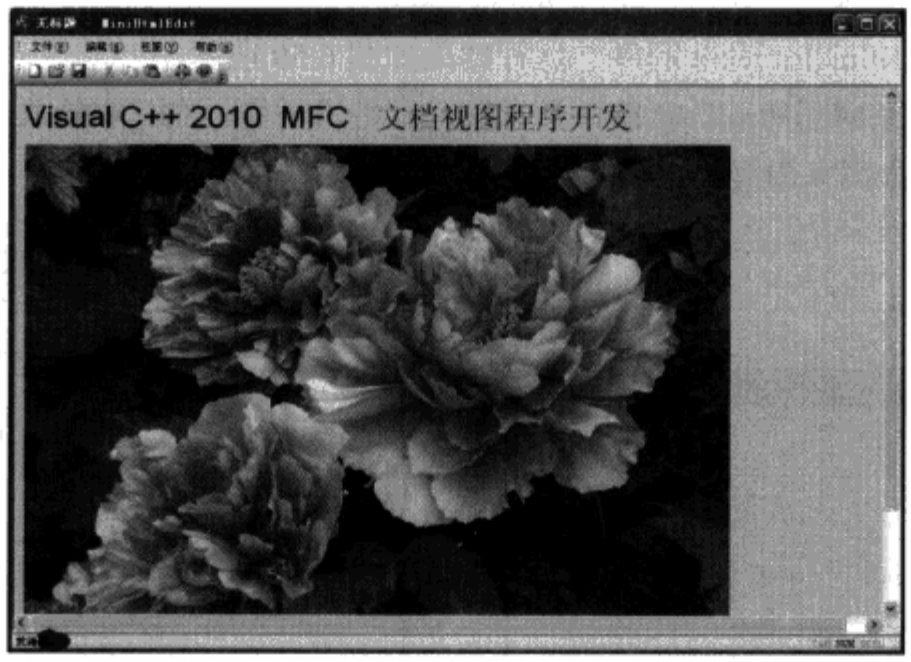


图 6.52　在 MiniHtmlEdit 程序运行框中插入图像文件

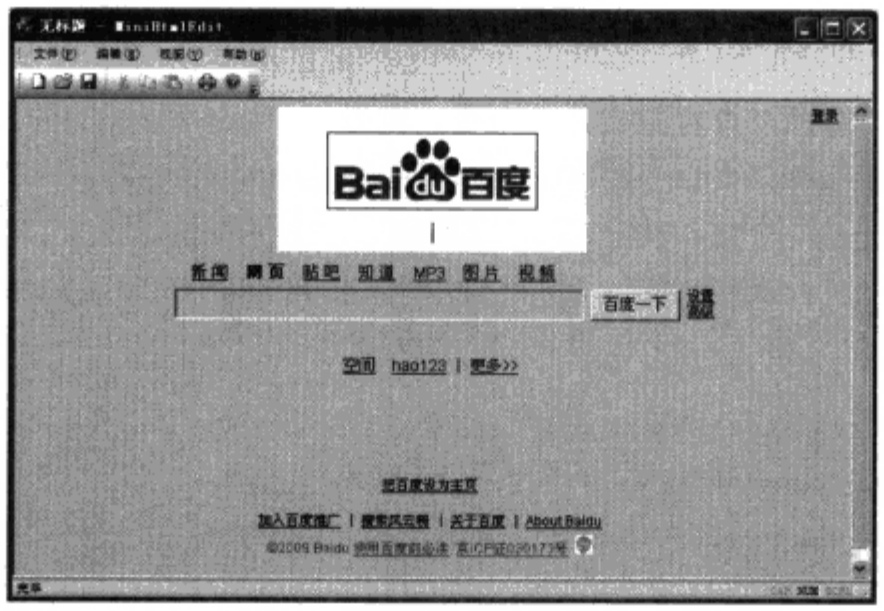


图 6.53　打开网页功能

此时，若在搜索框中输入“visual studio 2010”，仍然具有搜索功能，如图 6.54 所示。

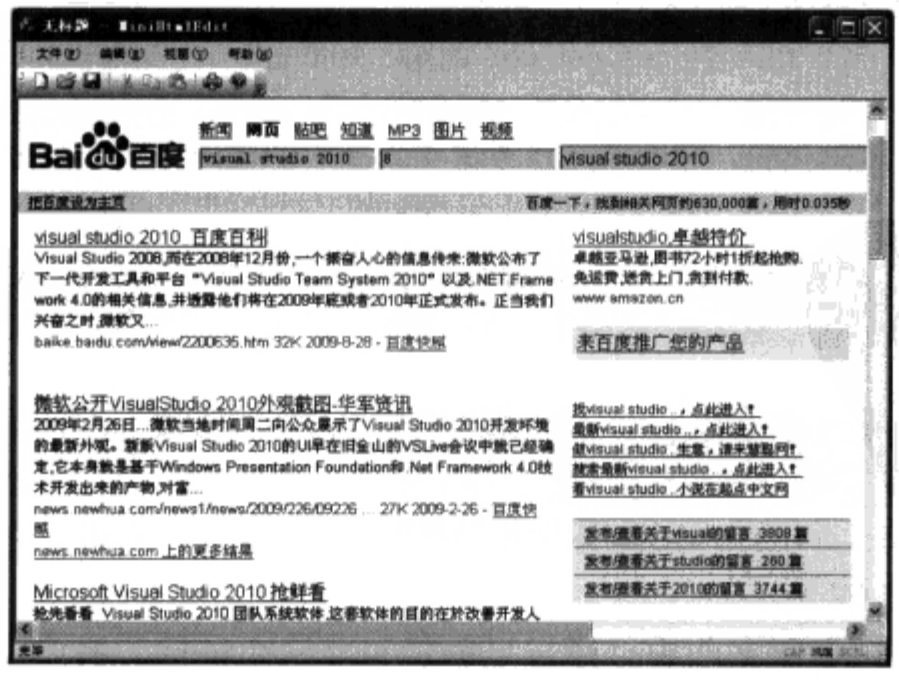


图 6.54　打开网页功能

此外，除了打开网页功能外，HtmlEditView 还具有编辑超链接的功能。

6.8　Visual C++ 2010 HtmlView 开发

在 Visual Studio 2010 中出现了一个新类 CHtmlView，利用这个类，可以实现在对话框的控件中显示 HTML 文件。要想使用 CHtmlView 类，对其定义和实现就必须有全面深入的理解。不妨将 CHtmlView 和 CListView 做比较，通过比较，会发现一些有趣的差别，首先，MFC 中 CListView 有一个对应的 CListCtrl 类，而 CHtmlView 却没有一个 CHtmlCtrl 类与之对应；其次，CListView 的使用依赖于 MFC 的文档/视图结构，而 CHtmlView 的实现是基于 COM 的。通过 IWebBrowser2 接口来实现，而且 IWebBrowser2 与 MFC 文档/视图结构之间没有任何关系。

首先介绍 CHtmlView 类，其继承派生关系如下所示：

```
CObject
└CCmdTarget
  └CWnd
      └CView
          └CScrollView
              └CFormView
                  └CHtmlView
```

CHtmlView 类在文档/视图结构的上下文中提供 WebBrowser 控件的功能。WebBrowser 控件是客户可浏览网址及本地文件系统和网络文件夹的窗口。WebBrowser 控件支持超级链接、统一资源定位符（URL）导航器并维护一张历史列表。

在标准框架应用中（基于 SDI 或 MDI），视图对象通常由指定的系列的类派生，这些类都由 CView 派生，提供高于 CView 提供的指定功能。

基于 CHtmlView 的应用视图类用 WebBrowser 控件提供视图，这使此应用成为一个网络浏览器。创建网络浏览器的更好方法是使用 MFC AppWizard，并将 CHtmlView 指定为视图类。CHtmlView 的功能是为访问网络（和/或 HTML 文件）的应用而设计的。下列 CHtmlView 成员函数只适用于 Internet Explorer 应用，这些函数可以替代 WebBrowser 控件，但它们无可见的效果：

```
·GetAddressBar ()
·GetFullName ()
·GetStatusBar ()
·SetAddressBar ()
·SetFullScreen ()
·SetMenuBar ()
·SetStatusBar ()
·SetToolBar ()
#include <afxhtml.h>
```

请参阅：联机文档“平台 SDK”中的 IWebBrowser2。

CHtmlView 类的成员函数介绍如表 6.9、表 6.10 和表 6.11 所示。

1．属性

表 6.9　CHtmlView 类的属性成员

属　性	功　能
GetType ()	获取文档对象的类型名

续表

属　性	功　能
GetLeft ()	获取 Internet Explorer 主窗口的左边缘的屏幕坐标
SetLeft ()	设置 Internet Explorer 主窗口的水平位置
GetTop ()	获取 Internet Explorer 主窗口的上边缘的屏幕坐标
SetTop ()	设置 Internet Explorer 主窗口的垂直位置
GetHeight ()	获取 Internet Explorer 主窗口的高度
SetHeight ()	设置 Internet Explorer 主窗口的高度
SetVisible ()	设置表示对象是可见还是隐藏的值
GetVisible ()	获取表示对象是可见还是隐藏的值
GetLocationName ()	获取 WebBrowser 当前显示的资源名
GetReadyState ()	获取 WebBrowser 的就绪状态
GetOffline ()	获取确定控件是否离线的值
SetOffline ()	设置确定控件是否离线的值
GetSilent ()	获取指示所有对话框是否能显示出来的值
SetSilent ()	设置确定控件是否显示在对话框的值
GetTopLevelContainer ()	获取指示当前对象是否是 WebBrowser 控件的顶级包容器的值
GetLocationURL ()	获取 WebBrowser 当前显示的资源的 URL
GetBusy ()	获取指示是否下载或其他活动仍在处理中的值
GetApplication ()	获取代替包含当前 Internet Explorer 应用实例的应用对象
GetParentBrowser ()	获取指向 Idispatch 界面的指针
GetContainer ()	获取 WebBrowser 控件的包容器
GetHtmlDocument ()	获取活动的 HTML 文档
GetFullName ()	获取显示在 WebBrowser 中（忽略 WebBrowser，只考虑 Internet Explorer）的资源的全名，包括路径
GetToolBar ()	获取确定工具栏是否可见的值
SetToolBar ()	设置确定工具栏是否可见的值（忽略 WebBrowser，只考虑 Internet Explorer）
GetMenuBar ()	获取确定菜单栏是否可见的值
SetMenuBar ()	设置确定菜单栏是否可见的值（忽略 WebBrowser，只考虑 Internet Explorer）
GetFullScreen ()	获取指示 WebBrowser 控件正操作在全屏模式还是普通窗口模式的值
SetFullScreen ()	设置指示 WebBrowser 控件正操作在全屏模式还是普通窗口模式的值（忽略 WebBrowser，只有 Internet Explorer）
QueryStatusWB ()	对正由 WebBrowser 控件执行的命令的状态的查询
GetRegisterAsBrowser ()	获取指示是否 WebBrowser 控件为目标名称的分解而登录为一个顶级浏览器的值
SetRegisterAsBrowser ()	设置指示是否 WebBrowser 控件为目标名字的分解而登录为一个顶级浏览器的值
GetRegisterAsDropTarget ()	获取指示是否 WebBrowser 控件为导航登录为一个落放目标的值
SetRegisterAsDropTarget ()	设置指示是否 WebBrowser 控件为导航登录为一个落放目标的值
GetTheaterMode ()	获取指示 WebBrowser 控件是否为影院模式的值
SetTheaterMode ()	设置指示 WebBrowser 控件是否为影院模式的值
GetAddressBar ()	确定 Internet Explorer 对象地址栏是否可见的值（忽略 WebBrowser，只考虑 Internet Explorer）
SetAddressBar ()	显示或隐藏 Internet Explorer 对象地址栏（忽略 WebBrowser，只考虑 Internet Explorer）
GetStatusBar ()	获取指示 Internet Explorer 对象地址条是否可见的值（忽略 WebBrowser，只考虑 Internet Explorer）
SetStatusBar ()	设置指示 Internet Explorer 对象地址条是否可见的值（忽略 WebBrowser，只考虑 Internet Explorer）

2．操作

表 6.10　　CHtmlView 类的操作成员

操　　作	功　　能
GoBack ()	导航到历史列表的上一项
GoForward ()	导航到历史列表的下一项
GoHome ()	导航到当前主页或起始页
GoSearch ()	导航到当前查找页
Navigate ()	导航到由 URL 标识的资源
Navigate2 ()	导航到由 URL 标识的资源，或由完整路径标识的文件
Refresh ()	重载当前文件
Refresh2 ()	重载当前文件并避免“pragma:nocache”标题被发送
Stop ()	停止打开文件
PutProperty ()	设置与指定对象有关的特性值
GetProperty ()	获取与指定对象有关的特性的当前值
ExecWB ()	执行一个命令
LoadFromResource ()	装载 WebBrowser 控件中的资源

3．可覆盖的函数

表 6.11　　CHtmlView 类的可覆盖函数成员

可覆盖的函数	功　　能
OnDraw ()	调用使一个图像屏幕显示打印或者打印先前值。需要实现
Create ()	创建 WebBrowser 控件
OnNavigateComplete2 ()	在到一个超级链接的导航完成后调用（窗口或框架元素）
OnBeforeNavigate2 ()	在导航发生在指定 WebBrowser 中之前调用（窗口或框架元素）
OnStatusTextChange ()	调用以通知一个应用：与 WebBrowser 控件有关的状态栏文本已改变
OnProgressChange ()	调用以通知一个应用：下载操作的过程被更新
OnCommandStateChange ()	调用以通知一个应用：WebBrowser 控件的有效状态被改变
OnDownloadBegin ()	调用以通知一个应用：导航操作开始了
OnDownloadComplete ()	当导航操作结束、中断或失败时调用
OnTitleChange ()	调用以通知一个应用：　WebBrowser 控件的文档标题是否有效或改变
OnPropertyChange ()	调用以通知一个应用：PutProperty()方法已经改变了特性的值
OnNewWindow2()	当新窗口被创建来显示资源时被调用
OnDocumentComplete ()	调用以通知一个应用：文档已达到 READYSTATE_COMPLETE 状态
OnQuit ()	调用以通知一个应用：Internet Explorer 应用准备退出（只适用于 Internet Explorer）
OnVisible ()	当 WebBrowser 控件窗口应被显示/隐藏时调用
OnToolBar ()	当 ToolBar 特性改变时被调用
OnMenuBar ()	当 MenuBar 特性改变时被调用
OnStatusBar ()	当 StatusBar 特性改变时被调用
OnFullScreen ()	当 FullScreen 特性改变时被调用
OnTheaterMode ()	当 TheaterMode 特性改变时被调用

为了实现在对话框的控件中显示 HTML 文件，也可以为 CHtmlView 创建一个对应的类 CHtmlCtrl。创建一个静态控件（也可以是其他控件），这个控件的 ID 及大小位置与界面上的控件相同。

```
BOOL CHtmlCtrl::CreateFromStatic(UINT nID, CWnd* pParent)
```

```
{
    CStatic wndStatic;
    if (!wndStatic.SubclassDlgItem(nID, pParent))
    return FALSE;
                                        // 获取静态控件的矩形区域并将其转换为父窗口的客户区坐标
    CRect rc;
    wndStatic.GetWindowRect(&rc);
    pParent->ScreenToClient(&rc);
    wndStatic.DestroyWindow();
                                        // 创建 HTML 控制 (CHtmlView)
    return
    Create(NULL,                        // 类名
        NULL,                           // 标题
        (WS_CHILD | WS_VISIBLE ),       // 风格
        rc,                             // 矩形区域
        pParent,                        // /父窗口
        nID,                            // 控制 ID
        NULL);                          ///框架/文档
}
```

为了避免主控程序将 CHtmlView 对象看作文档/视图框架，需要重载 CView::OnMouseActivate()和 CView::OnDestroy()。此外，当用户在控件中单击时，OnMouseActivate()要负责响应（WM_MOUSEACTIVATE）。

```
int CHtmlC:rl::OnMouseActivate(CWnd* pDesktopWnd, UINT nHitTest, UINT msg)
{
    //旁路 CView 文档/框架
    return CWnd::OnMouseActivate(pDesktopWnd, nHitTest, msg);
}
void CHtmlCtrl::OnDestroy()
{
    if (m_pBrowserApp)
    {
        m_pBrowserApp->Release();
        m_pBrowserApp = NULL;
    }
    CWnd::OnDestroy(); // 旁路 CView 文档/框架
}
```

通常，CHtmlView 是在 virtual void PostNcDestroy()中释放空间，但对话框中的控件常常是作为堆栈对象实现的，所以在 PostNcDestroy()中不必在做什么。

```
virtual void PostNcDestroy() {   }
```

为了实现“app:”伪协议，重载导航处理器 OnBeforeNavigate2()，传递“app:”链接到一个虚拟协议处理器。因为 app:是假协议，所以在浏览器中要取消这个导航。

```
void CHtmlCtrl::OnBeforeNavigate2( LPCTSTR lpszURL,
    DWORD nFlags,
    LPCTSTR lpszTargetFrameName,
    CByteArray& baPostedData,
    LPCTSTR lpszHeaders,
    BOOL* pbCancel )
{
    onst char APP_PROTOCOL[] = "app:";
```

```
        int len = _tcslen(APP_PROTOCOL);
        if (_tcsnicmp(lpszURL, APP_PROTOCOL, len)==0)
        {
                OnAppCmd(lpszURL + len);
                *pbCancel = TRUE;
        }
}
```

重载 OnAppCmd()，处理 app:命令，当浏览器准备导航到“app:foo”时，这个函数被调用，参数 lpszWhere 的值为“foo”。

```
void CHtmlCtrl::OnAppCmd(LPCTSTR lpszWhere){ // default: do nothing}
```

重载 OnMouseActivate()、OnDestroy()、和 PostNcDestroy()以后，CHtrmlCtrl 在对话框中就可以像个控件一样工作。

6.9　Visual C++ 2010 ListView 开发

生成 MiniList 程序时，首先像 6.3 节的 MiniDraw 程序那样，用应用程序向导生成源代码文件。在“New Project”对话框中，“Name”文本框中输入 MiniList，“Location”文本框中输入项目文件夹路径，在此选择默认存储路径 C:\Documents and Settings\Administrator\My Documents\Visual Studio 10\Projects。

在应用程序向导对话框中，设置与生成前面 MiniDraw 程序相同的选项，但在“Generated Classes”对话框中，不是接受所有的默认值，而是要按照如下的步骤进行设置。

（1）选择对话框顶部列表框中的类名 CMiniListView。

（2）在“Base class”下拉列表框中选择 CListView，完成的“Generated Classes”对话框如图 6.55 所示。后续开发可以在 MiniListView.h 和 MiniListView.cpp 文件中添加要实现的功能和函数的声明及实现。

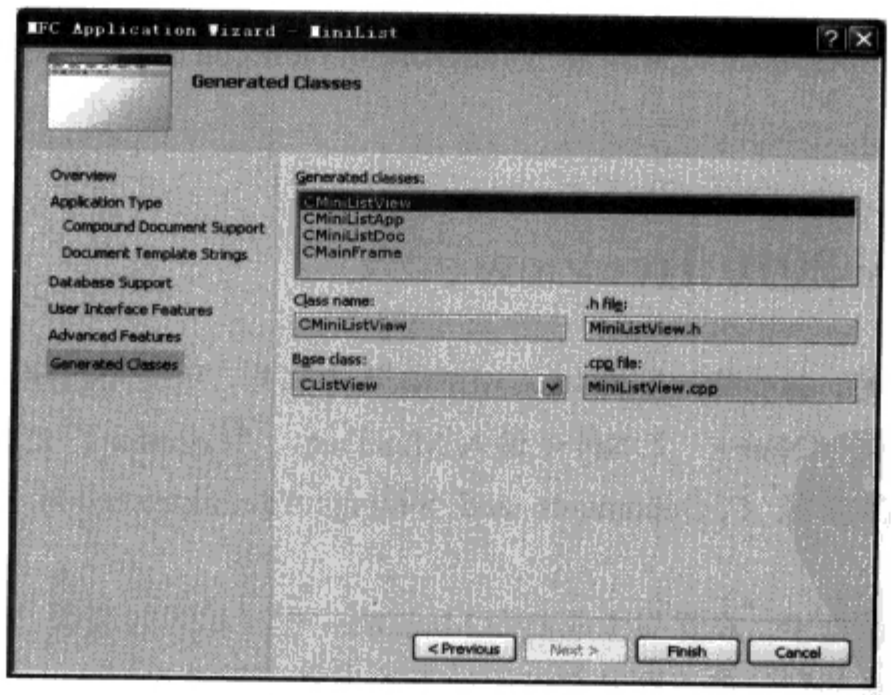


图 6.55　“Generated Classes”对话框

6.10　Visual C++ 2010 RichEditView 开发

生成 MiniRichEdit 程序时，首先像 6.3 节的 MiniDraw 程序那样，用应用程序向导生成源代码文件。

在“New Project”对话框中，“Name”文本框中输入 MiniRichEdit，“Location”文本框中输入项目文件夹路径，在此选择默认存储路径 C:\Documents and Settings\Administrator\My Documents\Visual Studio 10\Projects。

在应用程序向导对话框中，设置与生成前面 MiniDraw 程序相同的选项，但在“Generated Classes”对话框中，不是接受所有的默认值，而是要按照如下的步骤进行设置。

（1）选择对话框顶部列表框中的类名 CMiniRichEditView。

（2）在“Base class”下拉列表中选择 CRichEditView，完成的“Generated Classes”对话框如图 6.56 所示。

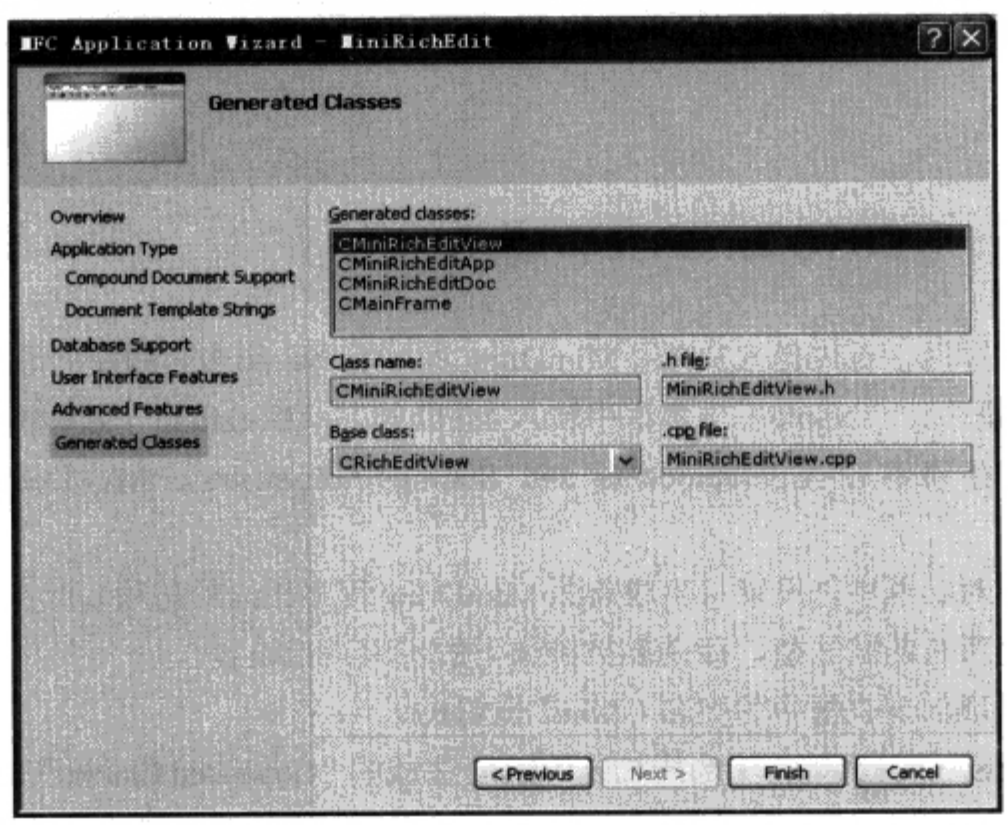


图 6.56 “Generated Classes”对话框

后续开发可以在 MiniRichEditView.h 和 MiniRichEditView.cpp 文件中添加要实现的功能和函数的声明及实现。

6.11 VisualC++ 2010 TreeView 开发

生成 MiniTree 程序时，首先像 6.3 节的 MiniDraw 程序那样，用应用程序向导生成源代码文件。在“New Project”对话框中，“Name”文本框中输入 MiniTree，“Location”文本框中输入项目文件夹路径，在此选择默认存储路径 C:\Documents and Settings\Administrator\My Documents\Visual Studio 10\Projects。

在应用程序向导对话框中，设置与生成前面 MiniDraw 程序相同的选项，但在“Generated Classes”对话框中，不是接受所有的默认值，而是要按照如下的步骤进行设置。

（1）选择对话框顶部列表框中的类名 CMiniTreeView。

（2）在“Base class”下拉列表框中选择 CListView，完成的“Generated Classes”对话框如图 6.57 所示。

后续开发可以在 MiniTreeView.h 和 MiniTreeView.cpp 文件中添加要实现的功能和函数的声明及实现。

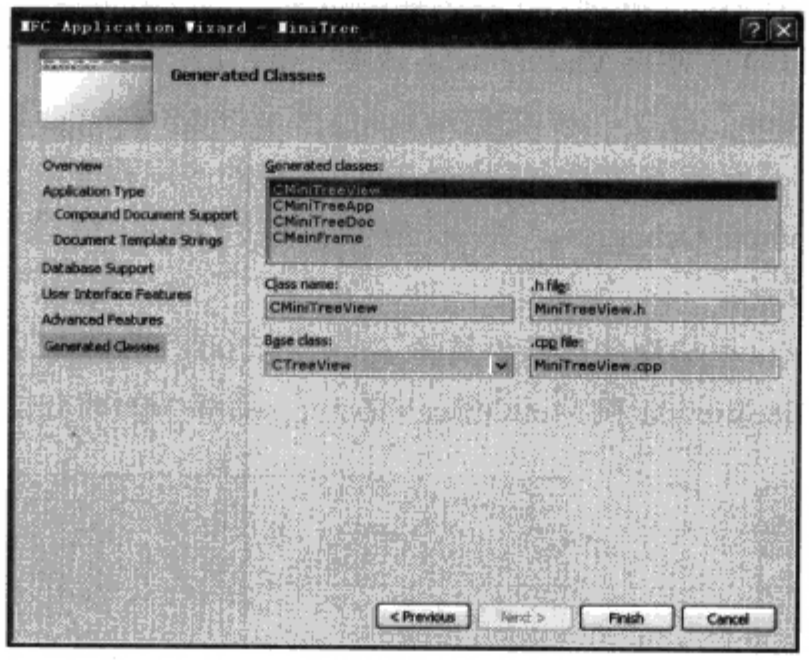


图 6.57 “Generated Classes”对话框

6.12　Visual C++ 2010 Office 2007 风格文档视图开发框架

为了生成程序源文件，和第 2 章中生成源代码部分一样，使用应用程序向导。但在“New Project”对话框的“Name”文本框中输入的是 MiniOffice，并在“Locatoin”文本框中指定所要的项目文件夹路径，在此选择默认存储路径。图 6.58 显示了完成设置的“New Project”对话框，在该图中，项目文件夹名为 MiniOffice，存放在 C:\Documents and Settings\Administrator\My documents\Visual Studio 10\Projects 目录下。然后单击“OK”按钮，进入向导对话框。

在应用程序向导对话框中，设置与生成前面 MiniDraw 例子程序相同的选项，但在“Application Type”对话框中，不是接受所有的默认值，而是要按照如下的步骤进行设置。

（1）设置“Application type”为“Single document”，即选择单文档类型。

（2）设置“Resource language”为默认的英语，也可以按照自己的习惯设置为中文。

（3）在“Project style”选项组中选中“office”单选按钮，“Visual style and colors”下拉列表框则选择相应的默认选项“office 2007（blue theme）”。当然，也可以根据个人习惯选择自己喜欢的风格。配置好的“Application Type”对话框如图 6.59 所示。

在其他的应用程序向导对话框中，一定要选择和第 2 章生成应用程序 HelloWorld 时相同的选项。

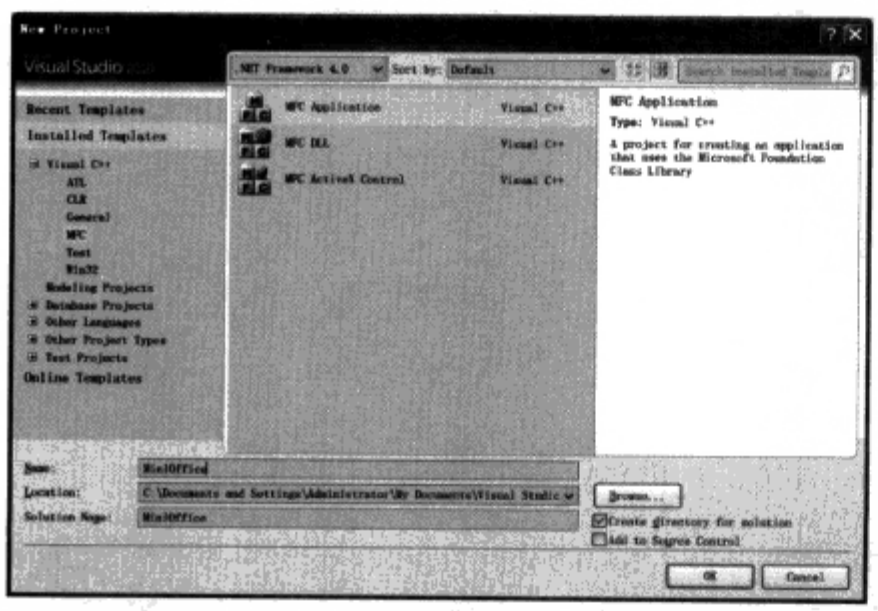


图 6.58 “New Project”对话框

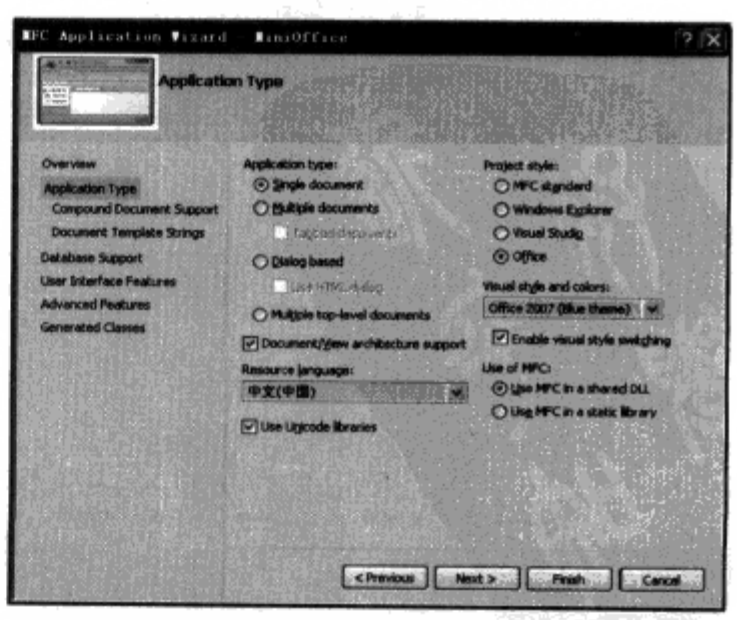


图 6.59　配置好的“Application Type”对话框

应用程序向导生成源文件后，会在项目工作区窗口的 Solution Explorer 中显示这些文件的文件名，如图 6.60 所示（图已经完全展开）。退出 Visual Studio 或者关闭 MiniOffice 项目后，可以随时选择“File”→“Open”→的“Project/Solution”命令，并选择项目工作区文件 MiniOffice.sln，再次打开该项目文件。

此时，已经建立了一个具有基本功能的 Office 2007 风格的应用程序。选择“Build”→“MiniOffice”命令和“Debug”→“Start without Debugging”命令，或者直接按【Ctrl+F5】组合键，编译并运行 MiniOffice 程序，运行结果如图 6.61 所示。

通过运行结果可以看出，框架程序已基本具备了 Office 2007 的风格，若想要修改或者添加自己想要的风格，可以在 MiniOfficeView.h 和 MiniOfficeView.cpp 文件中声明和定义相应功能的函数。

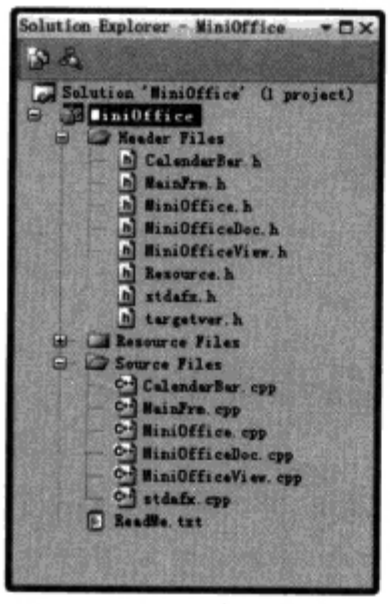


图 6.60　Solution Explorer 对话框

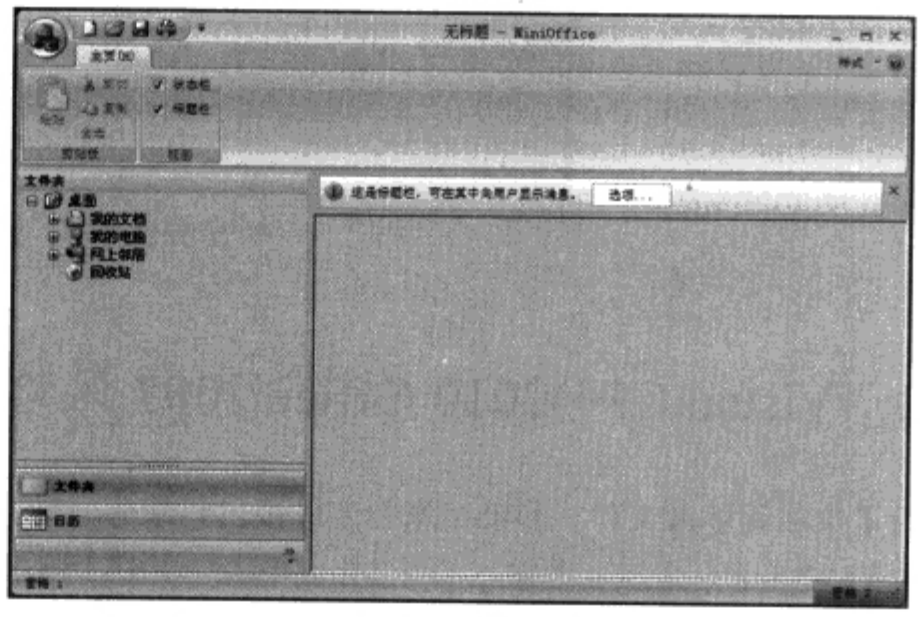


图 6.61　MiniOffice 程序的运行结果

程序的框架内容如下：

MiniOfficeView.h

```
// 这段 MFC 示例源代码演示如何使用 MFC Microsoft Office Fluent 用户界面
                    // ("Fluent UI")，该示例仅作为参考资料提供
                    // 用于补充 Microsoft 基础类参考和
                    // MFC C++ 库软件附的相关电子文档
                    // 复制使用或分发 Fluent UI 的许可条款是单独提供的
                    // 若要了解有关 Fluent UI 许可计划的详细信息，请访问
                    // http://msdn.microsoft.com/officeui
                    // 版权所有 D (C) Microsoft Corporation
                    // 保留所有权利

                    // MiniOfficeView.h : CMiniOfficeView 类的接口

#pragma once
class CMiniOfficeView : public CView
{
protected:          // 仅从序列化创建
    CMiniOfficeView();
    DECLARE_DYNCREATE(CMiniOfficeView)

                    // 属性
public:
    CMiniOfficeDoc* GetDocument() const;
```

```
                                // 操作
public:

                                // 重写
public:
    virtual void OnDraw(CDC* pDC);  //重写以绘制该视图
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);

                                // 实现
public:
    virtual ~CMiniOfficeView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

                                // 生成的消息映射函数
protected:
    afx_msg void OnFilePrintPreview();
    afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnContextMenu(CWnd* pWnd, CPoint point);
    DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG              // MiniOfficeView.cpp 中的调试版本
inline CMiniOfficeDoc* CMiniOfficeView::GetDocument() const
   { return reinterpret_cast<CMiniOfficeDoc*>(m_pDocument); }
#endif
```

MiniOfficeView.cpp

```
// MiniOfficeView.cpp : CMiniOfficeView 类的实现

#include "stdafx.h"
#include "MiniOffice.h"

#include "MiniOfficeDoc.h"
#include "MiniOfficeView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif
```

```
// CMiniOfficeView

IMPLEMENT_DYNCREATE(CMiniOfficeView, CView)

BEGIN_MESSAGE_MAP(CMiniOfficeView, CView)
                                                    // 标准打印命令
    ON_COMMAND(ID_FILE_PRINT, &CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, &CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, &CMiniOfficeView::OnFilePrintPreview)
END_MESSAGE_MAP()

                                                    // CMiniOfficeView 构造/析构

CMiniOfficeView::CMiniOfficeView()
{
                                                    // TODO: 在此处添加构造代码

}

CMiniOfficeView::~CMiniOfficeView()
{
}

BOOL CMiniOfficeView::PreCreateWindow(CREATESTRUCT& cs)
{
                                                    // TODO: 在此处通过修改
                                                    //   CREATESTRUCT cs 来修改窗口类或样式

    return CView::PreCreateWindow(cs);
}

                                                    // CMiniOfficeView 绘制

void CMiniOfficeView::OnDraw(CDC* /*pDC*/)
{
    CMiniOfficeDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;

                                                    // TODO: 在此处为本机数据添加绘制代码
}

                                                    // CMiniOfficeView 打印


void CMiniOfficeView::OnFilePrintPreview()
{
    AFXPrintPreview(this);
}
```

```
BOOL CMiniOfficeView::OnPreparePrinting(CPrintInfo* pInfo)
{
                                                    // 默认准备
    return DoPreparePrinting(pInfo);
}

void CMiniOfficeView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
                                                    // TODO: 添加额外的打印前进行的初始化过程
}

void CMiniOfficeView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
                                                    // TODO: 添加打印后进行的清理过程
}

void CMiniOfficeView::OnRButtonUp(UINT /* nFlags */, CPoint point)
{
    ClientToScreen(&point);
    OnContextMenu(this, point);
}

void CMiniOfficeView::OnContextMenu(CWnd* /* pWnd */, CPoint point)
{
    theApp.GetContextMenuManager()->ShowPopupMenu(IDR_POPUP_EDIT, point.x, point.y, this, TRUE);
}


                                                    // CMiniOfficeView 诊断

#ifdef _DEBUG
void CMiniOfficeView::AssertValid() const
{
    CView::AssertValid();
}

void CMiniOfficeView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

CMiniOfficeDoc* CMiniOfficeView::GetDocument() const // 非调试版本是内联的
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMiniOfficeDoc)));
    return (CMiniOfficeDoc*)m_pDocument;
}
#endif //_DEBUG


// CMiniOfficeView 消息处理程序
```

6.13 Visual C++ 2010 Visual Studio 2008 风格文档视图开发框架

为了生成程序源文件，和第 2 章中生成源代码部分一样，使用应用程序向导。但在“New Project”对话框的“Name”文本框中输入的是 MiniVisualStudio，在“Locatoin”文本框中指定所要的项目文件夹路径，在此选择默认存储路径。其中，项目文件夹名为 MiniVisual Studio，存放在 C:\Documents and Settings\Administrator\My documents\Visual Studio 10\Projects 目录下。单击“OK”按钮，进入向导对话框。

在应用程序向导对话框中，设置与生成前面 MiniDraw 程序相同的选项，但在“Application Type”对话框中，不是接受所有的默认值，而是要按照如下的步骤进行设置。

（1）设置 Application type 为 Single document，即选择单文档类型。

（2）设置 Resource language 为默认的英语选项，也可以按照自己的习惯设置中文（中国）选项。

（3）在“Project type”选项组中选中“Visual Studio”单选按钮，Visual style and colors 中选择相应的默认选项“Visual Studio 2008”。当然，开发人员也可以根据个人习惯选择自己喜欢的风格。配置好的“Application Type”对话框如图 6.62 所示。

在其他的应用程序向导对话框中，一定要选择和第 2 章生成应用程序 HelloWorld 时相同的选项。

应用程序向导生成源文件后，会在项目工作区窗口的 Solution Explorer 中显示这些文件的文件名，如图 6.63 所示（图已经完全展开）。当退出 Visual Studio 或者关闭 MiniVisualStudio 项目后，可以随时选择“File”→“Open”→“Project/Solution”命令，并选择项目工作区文件 MiniVisualStudio.sln，再次打开项目文件。

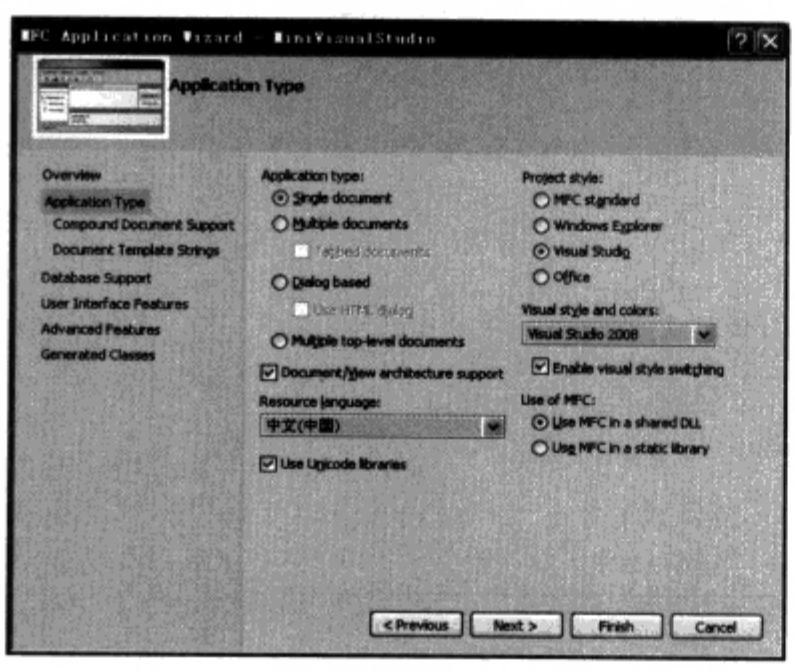


图 6.62 新建一个单文档程序

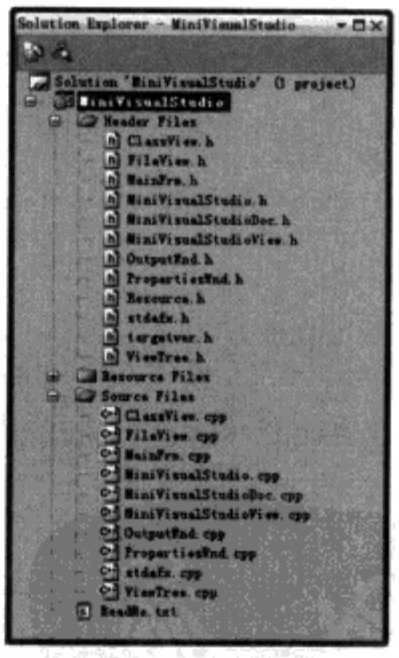


图 6.63 向导生成的文件

此时，已经建立了一个具有基本功能的 Visual Studio 2008 风格的应用程序。选择“Build”→“MiniVisual Studio”命令和“Debug”→“Start without Debugging”命令，或者直接按【Ctrl+F5】组合键，编译并运行 MiniVisualStudio 程序，运行结果如图 6.64 所示。

通过运行结果可以看出，框架程序已基本具备了 Visual Studio 2008 的风格。若想要修改或者添加自己想要的风格，则可以在 MiniVisualStudioView.h 和 MiniVisualStudioView.cpp 文件中声明和定义相应功能的函数。程序的框架内容如下：

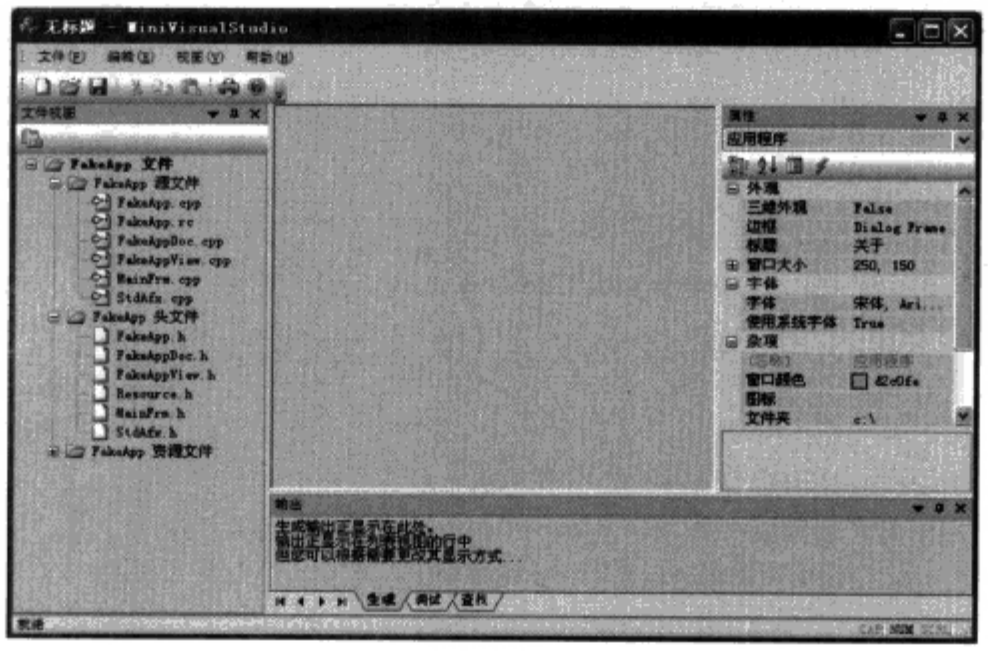


图 6.64　MiniVisualStudio 程序运行结果

MiniVisualStudioView.h

```
// MiniVisualStudioView.h : CMiniVisualStudioView 类的接口
//
#pragma once
class CMiniVisualStudioView : public CView
{
protected: // 仅从序列化创建
    CMiniVisualStudioView();
    DECLARE_DYNCREATE(CMiniVisualStudioView)

                                                    // 属性
public:
    CMiniVisualStudioDoc* GetDocument() const;

                                                    // 操作
public:

                                                    // 重写
public:
    virtual void OnDraw(CDC* pDC);                  // 重写以绘制该视图
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);

                                                    // 实现
public:
    virtual ~CMiniVisualStudioView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
```

```
protected:

                                                    // 生成的消息映射函数
protected:
    afx_msg void OnFilePrintPreview();
    afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnContextMenu(CWnd* pWnd, CPoint point);
    DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // MiniVisualStudioView.cpp 中的调试版本
inline CMiniVisualStudioDoc* CMiniVisualStudioView::GetDocument() const
   { return reinterpret_cast<CMiniVisualStudioDoc*>(m_pDocument); }
#endif
```

MiniVisualStudioView.cpp

```
// MiniVisualStudioView.cpp : CMiniVisualStudioView 类的实现
//
#include "stdafx.h"
#include "MiniVisualStudio.h"

#include "MiniVisualStudioDoc.h"
#include "MiniVisualStudioView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


                                                    // CMiniVisualStudioView

IMPLEMENT_DYNCREATE(CMiniVisualStudioView, CView)

BEGIN_MESSAGE_MAP(CMiniVisualStudioView, CView)
                                                    // 标准打印命令
    ON_COMMAND(ID_FILE_PRINT, &CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, &CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, &CMiniVisualStudioView::OnFilePrintPreview)
END_MESSAGE_MAP()

                                                    // CMiniVisualStudioView 构造/析构

CMiniVisualStudioView::CMiniVisualStudioView()
{
                                                    // TODO: 在此处添加构造代码

}

CMiniVisualStudioView::~CMiniVisualStudioView()
{
}
```

```
BOOL CMiniVisualStudioView::PreCreateWindow(CREATESTRUCT& cs)
{
                                        // TODO: 在此处通过修改 CREATESTRUCT cs 来修改窗口类或样式

    return CView::PreCreateWindow(cs);
}

                                        // CMiniVisualStudioView 绘制

void CMiniVisualStudioView::OnDraw(CDC* /*pDC*/)
{
    CMiniVisualStudioDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;

                                        // TODO: 在此处为本机数据添加绘制代码
}


                                        // CMiniVisualStudioView 打印


void CMiniVisualStudioView::OnFilePrintPreview()
{
    AFXPrintPreview(this);
}

BOOL CMiniVisualStudioView::OnPreparePrinting(CPrintInfo* pInfo)
{
                                        // 默认准备
    return DoPreparePrinting(pInfo);
}

void CMiniVisualStudioView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
                                        // TODO: 添加额外的打印前进行的初始化过程
}

void CMiniVisualStudioView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
                                        // TODO: 添加打印后进行的清理过程
}

void CMiniVisualStudioView::OnRButtonUp(UINT /* nFlags */, CPoint point)
{
    ClientToScreen(&point);
    OnContextMenu(this, point);
}

void CMiniVisualStudioView::OnContextMenu(CWnd* /* pWnd */, CPoint point)
```

```
{
    theApp.GetContextMenuManager()->ShowPopupMenu(IDR_POPUP_EDIT, point.x, point.y, this, TRUE);
}


                                        // CMiniVisualStudioView 诊断

#ifdef _DEBUG
void CMiniVisualStudioView::AssertValid() const
{
    CView::AssertValid();
}

void CMiniVisualStudioView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

CMiniVisualStudioDoc* CMiniVisualStudioView::GetDocument() const // 非调试版本是内联的
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMiniVisualStudioDoc)));
    return (CMiniVisualStudioDoc*)m_pDocument;
}
#endif //_DEBUG
                                        // CMiniVisualStudioView 消息处理程序
```

6.14 Visual C++ 2010 Windows 资源管理器风格文档视图开发框架

为了生成程序源文件，和第 2 章中生成源代码部分一样，使用应用程序向导。但在“New Project”对话框的“Name”文本框中输入的是 MiniWindows，在“Locatoin”文本框中指定所要的项目文件夹路径，在此选择默认存储路径。其中，项目文件夹名为 MiniVisual Studio，存放在 C:\Documents and Settings\Administrator\My documents\Visual Studio 10\Projects 目录下。然后单击“OK”按钮，进入向导对话框。

在应用程序向导对话框中，设置与生成前面 MiniDraw 程序相同的选项，但在“Application Type”对话框中，不是接受所有的默认值，而是要按照如下的步骤进行设置。

（1）设置 Application type 为 Single document，即选择单文档类型。

（2）在“Resource language”下拉列表框中，可以选择默认的英语选项，也可以按照自己的习惯选择中文（中国）选项。

（3）在“Project type”选项组中选中“Windows Explorer”单选按钮，则“Visual style and colors”下拉列表框中则选择相应的默认选项“Windows Native/Default”。当然，开发人员也可以根据个人习惯选择自己喜欢的风格。配置好的“Application Type”对话框如图 6.65 所示。

在其他的应用程序向导对话框中，一定要选择和第 2 章生成应用程序 HelloWorld 时相同的选项。

应用程序向导生成源文件后，会在项目工作区窗口的 Solution Explorer 中显示这些文件的文件名，如图 6.66 所示（图已经完全展开）。当退出 Visual Studio 或者关闭 MiniWindows 项目后，可以随时选择“File”→“Open”→“Project/Solution”命令，并选择项目工作区文件 MiniWindows.sln，再次打开该项目文件。

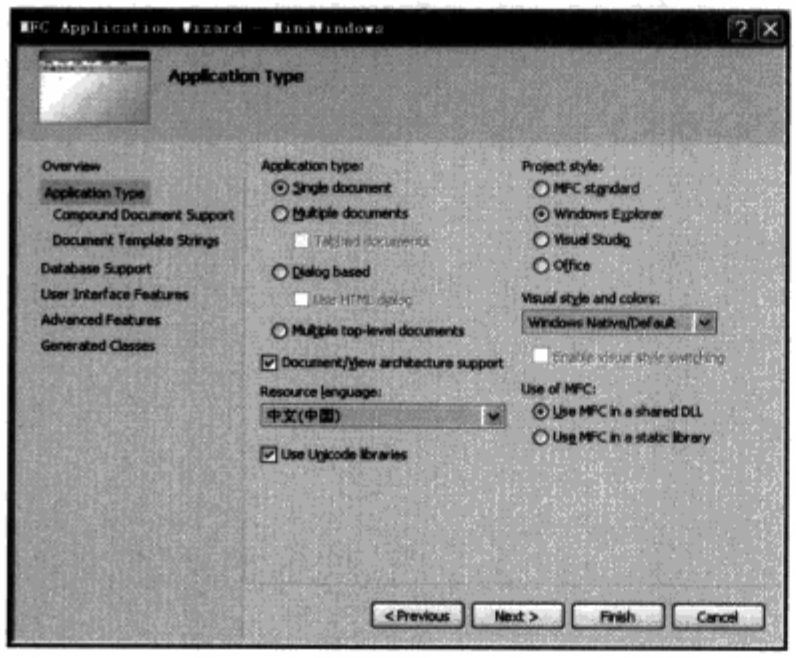


图 6.65　设置 MFC 向导

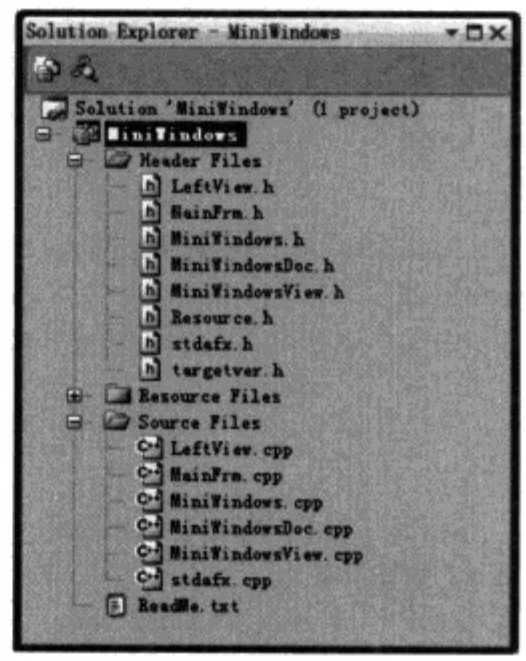


图 6.66　向导生成的文件

此时，已经建立了一个具有基本功能的 Windows Explorer 风格的应用程序。选择“Build”→“MiniWindows”命令和“Debug”→“Start without Debugging”命令，或者直接按【Ctrl+F5】组合键，编译并运行 MiniWindows 程序，运行结果如图 6.67 所示。

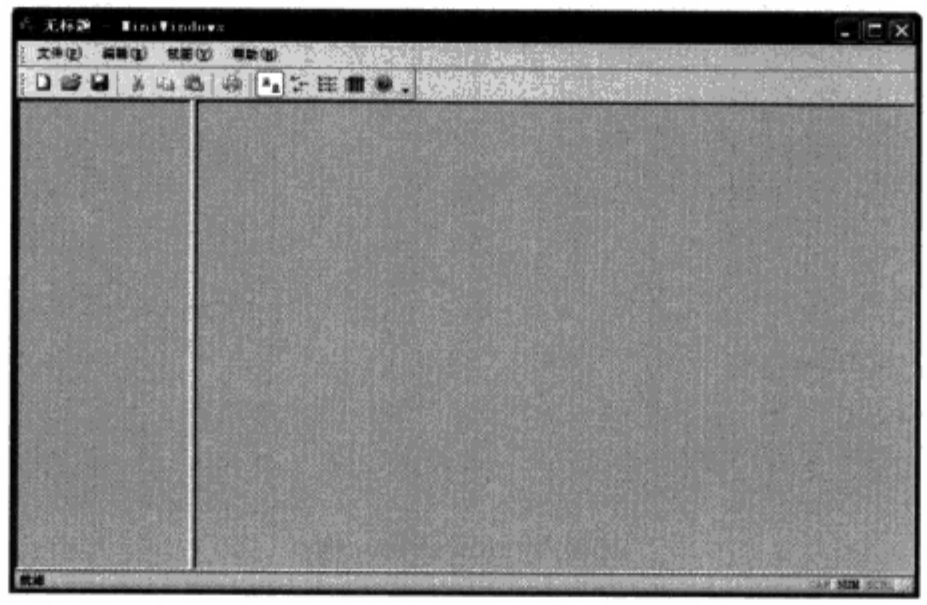


图 6.67　MiniWindows 程序运行结果

通过运行结果可以看出，框架程序已基本具备了 Windows 资源管理器的风格。若想要修改或者添加自己想要的风格，则可以在 MiniWindowsView.h 和 MiniWindowsView.cpp 文件中声明和定义相应功能的函数。程序的框架内容如下：

MiniWindowsView.h

```
// MiniWindowsView.h : CMiniWindowsView 类的接口
//
#pragma once
class CMiniWindowsView : public CListView
{
protected: // 仅从序列化创建
    CMiniWindowsView();
    DECLARE_DYNCREATE(CMiniWindowsView)
// 属性
public:
    CMiniWindowsDoc* GetDocument() const;
```

```
                                                    // 操作
public:

                                                    // 重写
public:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
    virtual void OnInitialUpdate(); // 构造后第一次调用

                                                    // 实现
public:
    virtual ~CMiniWindowsView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

                                                    //生成的消息映射函数
protected:
    afx_msg void OnStyleChanged(int nStyleType, LPSTYLESTRUCT lpStyleStruct);
    afx_msg void OnFilePrintPreview();
    afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnContextMenu(CWnd* pWnd, CPoint point);
    DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // MiniWindowsView.cpp 中的调试版本
inline CMiniWindowsDoc* CMiniWindowsView::GetDocument() const
   { return reinterpret_cast<CMiniWindowsDoc*>(m_pDocument); }
#endif
```

MiniWindowsView.cpp

```
// MiniWindowsView.cpp : CMiniWindowsView 类的实现
//
#include "stdafx.h"
#include "MiniWindows.h"

#include "MiniWindowsDoc.h"
#include "MiniWindowsView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif
// CMiniWindowsView

IMPLEMENT_DYNCREATE(CMiniWindowsView, CListView)

BEGIN_MESSAGE_MAP(CMiniWindowsView, CListView)
    ON_WM_STYLECHANGED()
END_MESSAGE_MAP()
```

```
                                        // CMiniWindowsView 构造/析构

CMiniWindowsView::CMiniWindowsView()
{
                                        // TODO: 在此处添加构造代码
}
CMiniWindowsView::~CMiniWindowsView()
{
}
BOOL CMiniWindowsView::PreCreateWindow(CREATESTRUCT& cs)
{
                                        // TODO: 在此处通过修改 CREATESTRUCT cs 来修改窗口类或样式
    return CListView::PreCreateWindow(cs);
}
void CMiniWindowsView::OnInitialUpdate()
{
    CListView::OnInitialUpdate();
                                        // TODO: 调用 GetListCtrl() 直接访问 ListView 的列表控件
                                        //   从而可以用项填充 ListView
}
void CMiniWindowsView::OnRButtonUp(UINT /* nFlags */, CPoint point)
{
    ClientToScreen(&point);
    OnContextMenu(this, point);
}
void CMiniWindowsView::OnContextMenu(CWnd* /* pWnd */, CPoint point)
{
    theApp.GetContextMenuManager()->ShowPopupMenu(IDR_POPUP_EDIT, point.x, point.y, this, TRUE);
}
                                        // CMiniWindowsView 诊断

#ifdef _DEBUG
void CMiniWindowsView::AssertValid() const
{
    CListView::AssertValid();
}
void CMiniWindowsView::Dump(CDumpContext& dc) const
{
    CListView::Dump(dc);
}
CMiniWindowsDoc* CMiniWindowsView::GetDocument() const // 非调试版本是内联的
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMiniWindowsDoc)));
    return (CMiniWindowsDoc*)m_pDocument;
}
#endif //_DEBUG
                                        // CMiniWindowsView 消息处理程序
void CMiniWindowsView::OnStyleChanged(int nStyleType, LPSTYLESTRUCT lpStyleStruct)
{
                                        //TODO: 添加代码以响应用户窗口视图样式的更改
    CListView::OnStyleChanged(nStyleType,lpStyleStruct);
}
```

6.15 小结

本章主要通过各种实例和程序介绍了如何实现 MFC 程序的视图类。在 MiniDraw 程序中，视图类从 CView 类派生，需要开发人员加入自己的代码接受用户输入并在显示窗口中输出。在 MiniEdit 程序中，显示类则从专用 CEdit View 类派生，实现全功能的文字编辑器，不必向视图类中加入自己的代码。在 FormDemo 程序中，显示类则是从专用的 CFormView 类派生；在增加滚动条功能的 MiniDraw 程序中，显示类则是从专用的 CScrollView 类派生，实现在显示的对话框中添加滚动条的功能。除此之外，还简要地介绍了 HtmlEditView、HtmlView、ListView、RichEditView、TreeView 及 Office 2007、Visual Studio 2008、Windows 资源管理器等风格的程序开发。完成本章的学习之后，读者应该：

（1）掌握单文档和多文档程序的开发；

（2）掌握各种 View 类的开发，包括 View、EditView、FormView、HtmlEditView、ScrollView、HtmlView、ListView、RichEditView、TreeView 的开发；

（3）掌握 MFC、Office 2007、Visual Studio 2008、Windows 资源管理器等风格的程序开发。

第 7 章　Visual C++ 2010 MFC 应用程序界面与美化

本章将讲述如何修改 MFC AppWizard 自动生成的应用程序的外观，在学习本章内容之前，读者有必要复习第 2 章中讲过的创建一个窗口的过程。

7.1　应用程序窗口风格美化

开发人员可以通过不同的方法来修改应用程序的背景、标题、光标、图标，以及框架是否具有最大、最小框等信息。由于视图类覆盖在框架窗口上，因此，通常在 MainFrame 中修改图标、标题及是否具有最大、最小框的属性；而背景及光标等信息就需要在 View 类中修改。

7.1.1　借助 ActiveSkin 美化窗口

ActiveSkin 是一款给软件更换皮肤的 ActiveX 控件，它很好地将软件界面设计工作从烦琐的程序代码编写中解放出来，使得功能设计者可以专心于功能代码的实现，而把软件界面交给美工人员处理，这样做提高了界面设计的工作效率，是一种很好的软件设计思想。

下面通过 3 个示例来介绍 ActiveSkin 控件的一般使用。

示例一：标准型皮肤 SkinForm 的对话框工程。

在 Visual C++环境下建立一个 MFC 基本对话框工程。在对话框的资源文件 Dlg.rc 中设计对话框界面时，删除已有的按钮 Buttons 和标签 Lables，并插入 ActiveSkin 的 ActiveX 控件，命名为 IDC_SKIN。右击 ActiveSkin 控件，选择“LoadSkin”命令，可预先加载一个 Skin 文件。

在 StdAfx.h 中，//{{AFX_INSERT_LOCATION}} 之前添加以下语句：

```
//目的是导入 ActiveSkin 的函数接口库
#include <atlbase.h>
#import "actskn43.ocx" no_implementation raw_interfaces_only raw_native_types
using namespace ACTIVESKINLib;
```

接着，在 Dlg.cpp 文件里添加语句：

```
CComQIPtr<ISkin> m_pSkin=GetDlgItem(IDC_SKIN)->GetControlUnknown();//m_pSkin 是应用程序全局量
m_pSkin->LoadSkin(L"皮肤文件 X 的路径");//如果 ActiveSkin 控件中已有皮肤，此句可省略。
m_pSkin->ApplySkin((int)m_hWnd);
```

编译运行程序，可以看到皮肤文件 X 成为了程序的运行界面。在程序运行时更换皮肤也是可以的。在对话框上添加一个 Button 控件，并添加 Click 事件：

```
void CMy1Dlg::OnBNewSkin()
{
    static char BASED_CODE szFilter[] = "ActiveSkin Files (*.skn)|*.skn||";        //文件过滤器设置
    CFileDialog dlg(FALSE, ".skn", NULL, OFN_HIDEREADONLY, szFilter);    //Skin 文件选择窗口
```

```
        if (dlg.DoModal() == IDOK)                                    //Skin 文件选择成功
        {
            USES_CONVERSION;
            m_pSkin->LoadSkin(T2W(dlg.GetPathName()));                //载入指定的 Skin 文件
            m_pSkin->ApplySkin((int)m_hWnd);                          //使当前 Skin 生效
        }
    }
```

可以发现，运行时主窗口已经换肤成功了，但“关于”对话框等非主窗口并没有变化。需要在 AboutDlg 的 OnInitDialog()事件中加入一句代码：

```
    m_pSkin->ApplySkin((int)m_hWnd);
```

再编译并运行程序就可以了。

示例二：标准皮肤 SkinForm 的多文档工程。

与示例一的做法类似，不同的只是子窗体在 ChildFrm.cpp 的 Create()事件中要做修改：

```
    BOOL bRes = CMDIChildWnd::Create(lpszClassName, lpszWindowName, dwStyle, rect, pParentWnd, pContext);
    m_pSkin->ApplySkin((long)m_hWnd);
    return bRes;
```

这样就可以了。

示例三：自由皮肤 SkinFreeForm 的事件操作。

和上述的 SkinForm 不同，SkinFreeFrom 是另一种类型的皮肤文件方案，因此其操作方法就有所不同了。可以参照 ActiveSkin 自带样例中的 DesktopColorizer。SkinForm 和 SkinFreeFrom 的主要区别在于 SkinApply()方法和事件的驱动方式：

```
    if(FAILED(m_pSkin->LoadSkin(Temp))||  //均使用 LoadSkin()装载某一 Skin，参数是 Skin 文件的路径名
      FAILED(m_pSkin->ApplySkinByName((long)m_hWnd, L"FreeForm")))

                                          //SkinForm 类型的皮肤使用 ApplySkin 使其生效，
                                          //这种方式会自动将某 Skin 中的所有 SkinObject 元素替代指定 Handle 的窗口
                                          //SkinFreeForm 类型的皮肤使用 ApplySkinByName()使其生效，
                                          //其中第一个参数是窗口句柄，第二个参数是 SkinBuilder 中一个 Object 的名称
    {
      MessageBox("Could not load or apply the skin.");
      PostMessage(WM_CLOSE);
      return FALSE;
    }
```

事件驱动的区别：SkinForm 换肤时不影响程序代码的原事件工作方式，只是界面上的换肤，因而调用 SkinForm::ApplySkin(Handle)就可以了；但是 SkinFreeForm 必须用一个消息泵，采用消息的方式使程序获得 Skin 上的事件，而且在其事件中采用 ISkinObject::GetName(BSTR *)获得相关的皮肤元素，用 GetSkinnedWindow(hWnd).FindObject("Screen")来获取相应的元素句柄并设置元素状态。简单的消息泵可以通过 Skin 控件的右击事件设置并编写。

在使用 Active Skin 控件前，请安装好 ActiveSkin4.3，可能还需要修改 Skin 控件初始的 LoadSkin()（Skin 文件）才可以。

7.1.2 修改窗口外观

新建一个 MFC 单文档应用程序，工程名为 ChangeFrm，或由用户自定义。

1．在 MainFrame 类中修改程序外观

（1）在 CMainFrame::PreCreateWindow()中修改。

在窗口产生之前，会调用其成员函数 PreCreateWindow()，此时窗口还没有产生，相当于一栋大楼修建之前，在其设计图纸上修改其式样，然后按照修改后的图纸去修建大楼。修改方法有以下几种。

- 直接修改 CREATESTRUCT 结构，代码如下：

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if( !CFrameWnd::PreCreateWindow(cs) )
        return FALSE;
                                    // TODO: Modify the Window class or styles here by modifying
                                    //  the CREATESTRUCT cs

    cs.style&=~WS_MAXIMIZEBOX;      //去掉窗口的最大化按钮
    cs.style&=~FWS_ADDTOTITLE;      //允许添加自己的标题
    cs.lpszName=_T("Visual C++教学");
    return TRUE;
}
```

其中 style 和 lpszName 是结构 CREATESTRUCT 的数据成员，分别表示框架窗口的样式和标题。需要注意的是，要想将窗口框架显示标题改为自己的标题，必须要将 style 的 FWS_ADDTOTITLE 属性去掉。

运行程序，会发现窗口的最大化按钮不见了，并且标题再不是以前的那种很枯燥的“无标题”，而是自己添加的“Visual C++教学”，结果如图 7.1 所示。

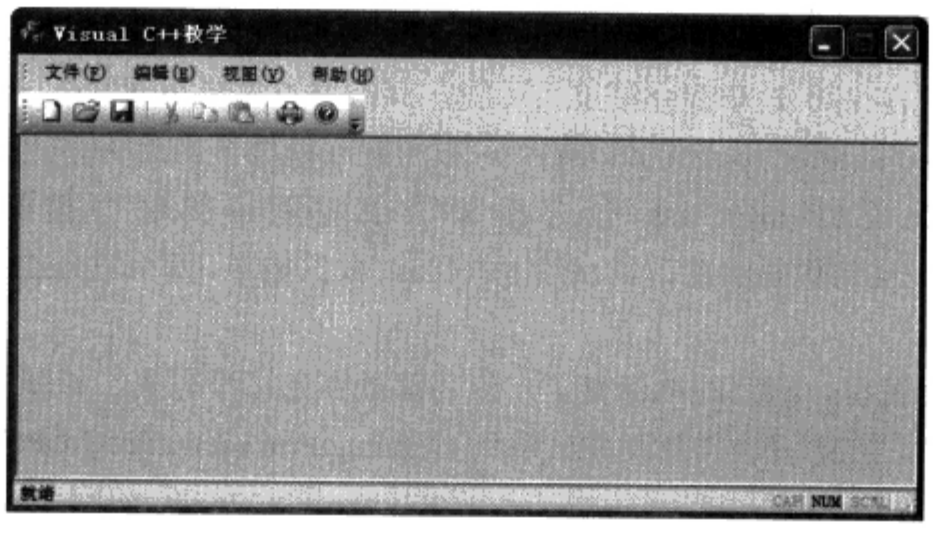


图 7.1　程序窗口视图

- 重新设计、填写 WNDCLASS，代码如例 7.1 所示。

例 7.1　CMainFrame::PreCreateWindow 函数代码

```
1   BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
2   {
3       if( !CFrameWnd::PreCreateWindow(cs) )
4           return FALSE;
5       // TODO: Modify the Window class or styles here by modifying
6       //  the CREATESTRUCT cs
7       WNDCLASS wndClass;
8       wndClass.style=CS_HREDRAW;
9       wndClass.lpfnWndProc=(WNDPROC)::DefWindowProc;
```

```
10        wndClass.cbClsExtra=0;
11        wndClass.cbWndExtra=0;
12        wndClass.hInstance=::AfxGetInstanceHandle();
13        wndClass.hIcon=LoadIcon(::AfxGetInstanceHandle(),
14                        MAKEINTRESOURCE(IDR_ChangeFrmTYPE));
15        wndClass.hCursor=LoadCursor(NULL,IDC_NO);
16        LOGBRUSH lgbr;
17        lgbr.lbStyle=BS_SOLID;
18        lgbr.lbColor=RGB(255,255,0);
19        lgbr.lbHatch=HS_CROSS;
20        wndClass.hbrBackground=CreateBrushIndirect(&lgbr);
21        wndClass.lpszMenuName=NULL;
22        wndClass.lpszClassName=_T("hello");
23        RegisterClass(&wndClass);
24        cs.lpszClass=_T("hello");
25        cs.style&=~WS_MAXIMIZEBOX;        //去掉窗口的最大化按钮
26        cs.style&=~FWS_ADDTOTITLE;        //允许添加自己的标题
27        cs.lpszName=_T("Visual C++教学");
28        return TRUE;
29    }
```

第 7～22 行代码定义了一个新的窗体类 wndClass 并设计该类，第 16～19 行代码中自定义了一个 LOGBRUSH 结构类型的画刷 lgbr，其中 lbStyle、lbColor 和 lbHatch 是该结构体的 3 个数据成员，分别表示所定义的画刷的样式等属性，设置该画刷的属性为实心的黄色，新设计的窗口类的背景就采用这个画刷设置。

其中，IDI_ICON1 是在资源面板中新添加的一个图标资源的 ID。

代码中的大部分内容在第 2 章都介绍过，可能；有的读者大概不理解 cs.lpszClass="hello";这句代码的用意。在 CMainFrame::PreCreateWindow()中，应用程序根据该函数传来的 cs 进行框架的显示，这个 cs 结构中的数据成员 lpszClass 包含框架生成时采用的窗口类的名称，如果不添加 cs.lpszClass=_T("hello");这句代码将窗口类赋给 lpszClass 数据成员，那么它还会采用原来系统默认的窗口类。

运行程序，读者会发现，框架的图标变了，但是背景和光标没有变，如图 7.2 所示，那是由于视图类覆盖在框架类上了，将此段代码复制并粘贴到 CChangeFrmView::PreCreateWindow()函数中，就可以看到效果了。

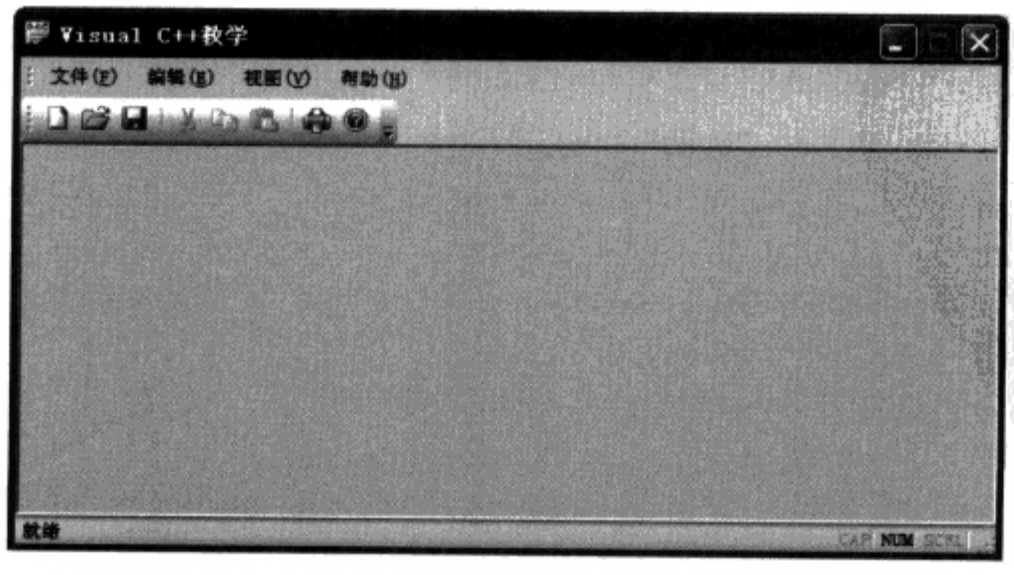


图 7.2　程序视图

- 利用 GetClassInfo 函数，代码如例 7.2 所示。

例 7.2　CMainFrame::PreCreateWindow()函数代码

```
1   BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
2   {
3       if( !CFrameWnd::PreCreateWindow(cs) )
4           return FALSE;
5       // TODO: Modify the Window class or styles here by modifying
6       //  the CREATESTRUCT cs
7       WNDCLASS wndclass;
8       ::GetClassInfo(AfxGetInstanceHandle(),cs.lpszClass,&wndclass);
9       wndclass.hIcon=::LoadIcon(::AfxGetInstanceHandle(),
10      MAKEINTRESOURCE(IDI_ICON1));
11      wndclass.lpszClassName="hello";
12      ::RegisterClass(&wndclass);
13      cs.lpszClass="hello";
14      return TRUE;
15  }
```

第 7 行首先定义一个窗口类对象 wndclass，然后在第 8 行调用 GetClassInfo()函数将取得的当前使用的窗口类信息赋给对象 wndclass，最后通过第 9~11 行修改 wndclass，再由第 12 行、第 13 行代码重新注册及加载修改后的窗口对象来实现，程序运行结果同图 7.2 一样。

函数 GetClassInfo()用于取得某一窗口类的信息，原型如下：

```
BOOL GetClassInfo( HINSTANCE hInstance, LPCTSTR lpClassName, LPWNDCLASS lpWndClass);
```

第一个参数 hInstance 表示创建窗口类的当前运行的应用程序实例。

第二个参数 lpClassName 表示要获取的窗口类的类名。

第三个参数 lpWndClass 指向 WNDCLASS 结构类型的对象，函数调用后获取的窗口类的信息存放到该参数中。

（2）在 CMainFrame:: OnCreate()中修改。

- 利用 SetWindowLong()函数，如下：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{ …
    SetWindowLong(m_hWnd,GWL_STYLE,
        ::GetWindowLong(m_hWnd,GWL_STYLE) & ~WS_MAXIMIZEBOX);
    return 0;
}
```

其中，GetWindowLong()函数用于取得指定窗口的信息，函数原型如下：

```
LONG GetWindowLong( HWND hWnd, int nIndex);
```

第一个参数 hWnd 表示要获取信息的窗口句柄，或窗口类所在的窗口。

第二个参数 nIndex 表示要获取的信息索引，此参数可取下列值：

```
GWL_EXSTYLE      获取窗口的扩展风格
GWL_STYLE        获取窗口的样式
GWL_WNDPROC      获取窗口过程的回调函数
```

函数 SetWindowLong()用于改变指定窗口的属性风格，该函数原型如下：

```
LONG SetWindowLong( HWND hWnd, int nIndex, LONG dwNewLong);
```

第一个参数 hWnd 表示要设置新属性的窗口句柄，或窗口类所在的窗口。

第二个参数 nIndex 表示要设置的属性的索引，此参数可取下列值：

GWL_EXSTYLE	设置新的窗口扩展风格
GWL_STYLE	设置新的窗口样式
GWL_ID	分配一个新的窗口标识

第三个参数 dwNewLong 表示要设置的新的属性值。

上段代码的实现过程是：首先调用 GetWindowLong()获取当前窗口的信息，然后去掉最大化按钮，最后调用 SetWindowLong()给窗口设置新的属性。

- 利用 SetClassLong()函数，如下：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{ …
    SetClassLong(m_hWnd,GCL_HICON,
        (long)AfxGetApp()->LoadIcon(IDI_ICON1));
    return 0;
}
```

其中，函数 SetClassLong()用于重置当前窗口所包含的窗口类的 WNDCLASS 结构中的指定数据成员的属性，函数原型如下：

```
DWORD SetClassLong( HWND hWnd, int nIndex, LONG dwNewLong);
```

第一个参数 hWnd 表示要设置新属性的窗口句柄，或窗口类所在的窗口。

第二个参数 nIndex 表示要设置的属性的索引，此参数可取下列值：

GCL_HBRBACKGROUND	设置新的背景画刷
GCL_HCURSOR	设置新的光标
GCL_HICON	设置新的图标
GCL_STYLE	设置新的窗口样式

第三个参数 dwNewLong 表示要设置的新的属性值。

2．在视图类中修改程序外观

（1）在 ChangeFrmView::PreCreateWindow()中修改。

- 重新设计、填写 WNDCLASS，代码如下：

```
BOOL CChangeFrmView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    //  the CREATESTRUCT cs
    //定义并设计新的窗口类 wndClass
    WNDCLASS wndClass;
    wndClass.style=CS_HREDRAW;
    wndClass.lpfnWndProc=(WNDPROC)::DefWindowProc;
    wndClass.cbClsExtra=0;
    wndClass.cbWndExtra=0;
    wndClass.hInstance=::AfxGetInstanceHandle();
    wndClass.hIcon=LoadIcon(::AfxGetInstanceHandle(),
            MAKEINTRESOURCE(IDR_ChangeFrmTYPE));
    wndClass.hCursor=LoadCursor(NULL,IDC_NO);
                                                //定义新的画刷来修改窗体类的背景色
    LOGBRUSH lgbr;
    lgbr.lbStyle=BS_SOLID;
    lgbr.lbColor=RGB(255,255,0);
    lgbr.lbHatch=HS_CROSS;
```

```
    wndClass.hbrBackground=CreateBrushIndirect(&lgbr);
    wndClass.lpszMenuName=NULL;
    wndClass.lpszClassName="hello";
    //注册窗口类
    RegisterClass(&wndClass);
    cs.lpszClass="hello";
    cs.style&=~WS_MAXIMIZEBOX;      //去掉窗口的最大化按钮
    cs.style&=~FWS_ADDTOTITLE;      //允许添加自己的标题
    cs.lpszName=_T("Visual C++教学");

    return CView::PreCreateWindow(cs);
}
```

该段代码放在框架窗口类中时，光标及背景并没有改变，将代码移到视图类中，就会看到如图 7.3 所示的效果。

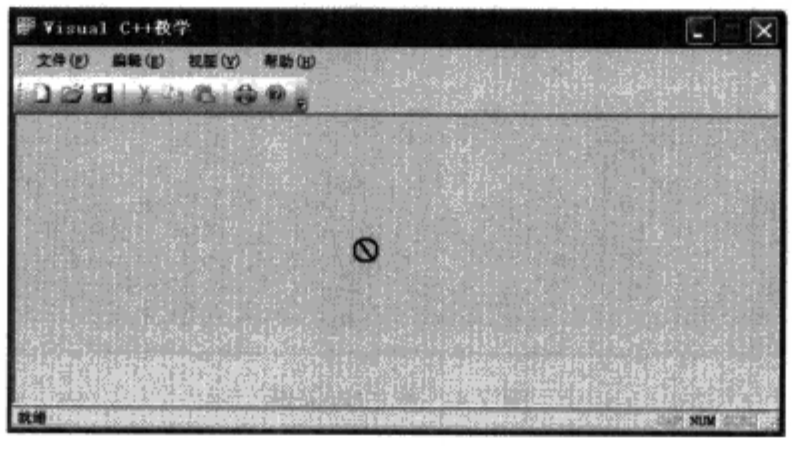


图 7.3　变色范例视图

- 利用 AfxRegisterWndClass()函数，代码如下：

```
BOOL CChangeFrmView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    //  the CREATESTRUCT cs
                                                    //定义并设计新的窗口类 wndClass
    WNDCLASS wndClass;
    wndClass.style=CS_HREDRAW;
    wndClass.lpfnWndProc=(WNDPROC)::DefWindowProc;
    wndClass.cbClsExtra=0;
    wndClass.cbWndExtra=0;
    wndClass.hInstance=::AfxGetInstanceHandle();
    wndClass.hIcon=LoadIcon(::AfxGetInstanceHandle(),
        MAKEINTRESOURCE(IDR_ChangeFrmTYPE));
    wndClass.hCursor=LoadCursor(NULL,IDC_NO);
                                                    //定义新的画刷来修改窗体类的背景色
    LOGBRUSH lgbr;
    lgbr.lbStyle=BS_SOLID;
    lgbr.lbColor=RGB(255,255,0);
    lgbr.lbHatch=HS_CROSS;
    wndClass.hbrBackground=CreateBrushIndirect(&lgbr);
    wndClass.lpszMenuName=NULL;
    wndClass.lpszClassName="hello";
```

```
        //注册窗口类
        RegisterClass(&wndClass);
        cs.lpszClass="hello";
        cs.style&=~WS_MAXIMIZEBOX;      //去掉窗口的最大化按钮
        cs.style&=~FWS_ADDTOTITLE;      //允许添加自己的标题
        cs.lpszName=_T("Visual C++教学");
        cs.lpszClass=AfxRegisterWndClass(CS_HREDRAW|CS_VREDRAW,
        LoadCursor(NULL,IDC_CROSS),(HBRUSH)GetStockObject(BLACK_BRUSH),0);

        return CView::PreCreateWindow(cs);
    }
```

运行程序结果如图 7.4 所示。

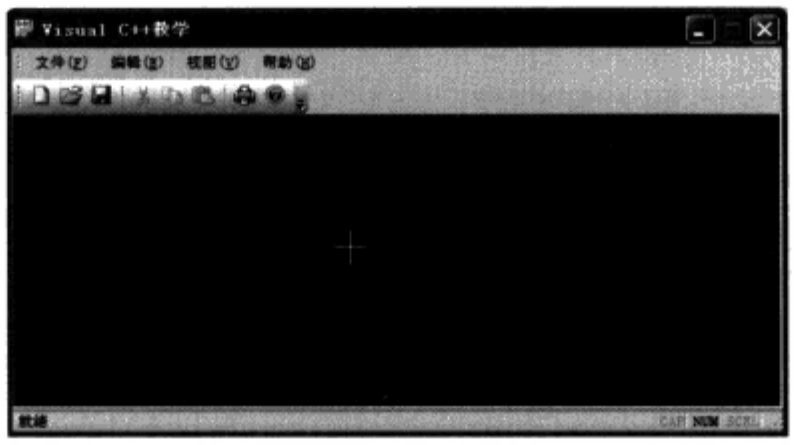


图 7.4　变色效果图

（2）在 CChangeFrmView::OnCreate()中修改。

- 利用 SetClassLong()函数，代码如下：

```
int CChangeFrmView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // TODO: Add your specialized creation code here
    SetClassLong(m_hWnd,GCL_HCURSOR,
    (LONG)LoadCursor(NULL,IDC_UPARROW));
    return 0;
}
```

运行结果如图 7.5 所示。

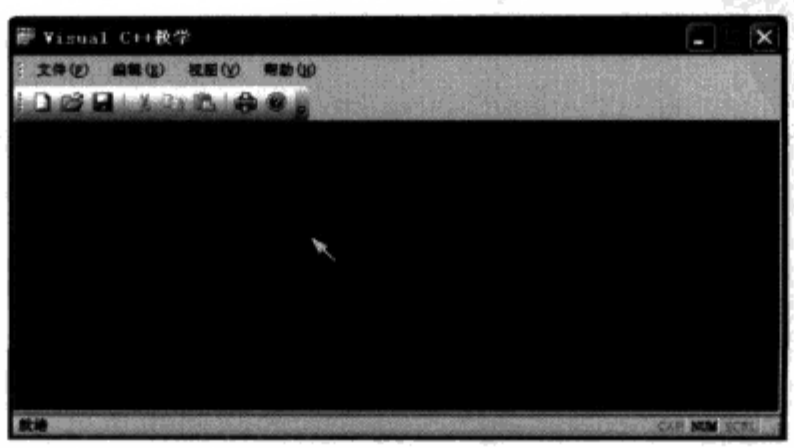


图 7.5　程序效果图

7.2　应用程序窗口图标与背景修改

7.2.1　修改窗口图标

本节将通过一个实例来讲述如何给自己的应用程序添加一组不断动态显示的图标。本实例将使应用程序框架上的图标不断变化，显示为一组脸部表情，当然，用到的图标是事先复制到文件夹 ChangeIcon\res 下的，在本工程的资源面板中只要导入这些图标就可以了，不需要开发人员自己再动手画一些图标。完整例程请参见光盘中 ChangeIcon 例子代码，具体操作步骤如下。

步骤 1：新建一个 MFC 单文档应用程序，工程名为 ChangeIcon 或由用户自定义。

步骤 2：在资源面板中导入 5 个图标。

5 个图标的名称分别为 ERROR.ICO、EXCLAMATION.ICO、INFO.ICO、QUESTION.ICO、CALC.ICO（包含在 ChangeIcon\res 文件夹下）。打开工作台的“Resource View”页面，右击 Icon 文件夹，在弹出的快捷菜单中选择“Add Resource”命令，在弹出的“Add Resource”对话框中，单击“Import”按钮，如图 7.6 所示，之后会弹出如图 7.7 所示的“Import”对话框，打开 ChangeIcon\res 文件夹，将该文件夹下的 5 个图标一一导入工程中。

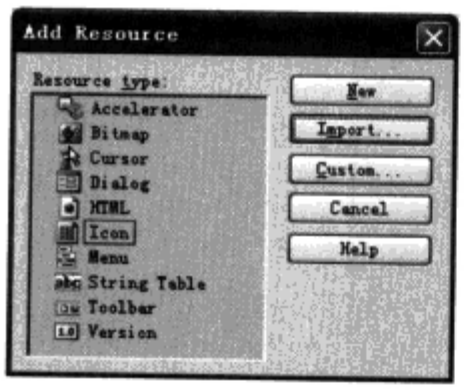


图 7.6　资源视图

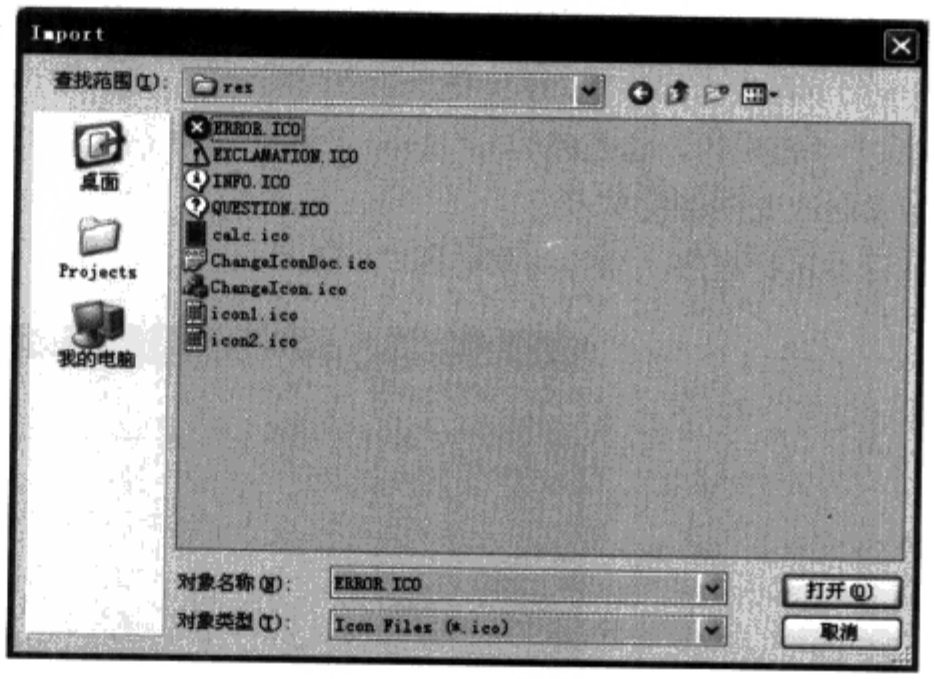


图 7.7　导入图标资源

5 个图标的 ID 均采用默认的 IDI_ICON1、IDI_ICON2、IDI_ICON3、IDI_ICON4、IDI_ICON5。

步骤 3：在框架窗口类中添加成员变量 m_hIcon[5]。

通常，改变图标的操作是在框架窗口类中，因此在 CMainFrame 类中添加 HICON 类型的成员变量 m_hIcon[5]，用来加载并保存图标资源：

```
HICON m_hIcon[5];
```

步骤 4：在框架窗口类的 OnCreate()函数中加载图标并设置计时器，代码如下：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{    …
     //加载图标资源
     m_hIcon[0]=AfxGetApp()->LoadIcon(IDI_ICON1);
     m_hIcon[1]=AfxGetApp()->LoadIcon(IDI_ICON2);
     m_hIcon[2]=AfxGetApp()->LoadIcon(IDI_ICON3);
```

```
    m_hIcon[3]=LoadIcon(AfxGetApp()->m_hInstance,MAKEINTRESOURCE(IDI_ICON4));
    m_hIcon[4]=LoadIcon(AfxGetInstanceHandle(),MAKEINTRESOURCE(IDI_ICON5));
    //改变框架的图标
    SetClassLong(m_hWnd,GCL_HICON,(LONG)m_hIcon[0]);
    SetTimer(1,500,NULL);
    return 0;
}
```

上段代码中，在加载 5 个图标时采用了 3 种方法，3 种方法的原理是相同的；然后调用 SetClassLong() 函数将框架的图标替换成第一个加载的图标；最后，设置计时器。接下来的工作就是在计时器事件中使图标不断地变化。

步骤 5：在 CMainFrame 类中添加 OnTimer()函数，代码如下：

```
void CMainFrame::OnTimer(UINT nIDEvent)
{
    // TODO: Add your message handler code here and/or call default
    static int iIndex=0;
    iIndex=++iIndex%5;
    SetClassLong(m_hWnd,GCL_HICON,(LONG)m_hIcon[iIndex]);
    CFrameWnd::OnTimer(nIDEvent);
}
```

其中，首先定义了一个静态整型变量 iIndex，用来标识图标数组变量的下标，静态变量在函数执行完毕时，不释放内存空间，并且变量的值仍然保留；第二句代码 iIndex=++iIndex%5;的作用是将 iIndex 控制在 0～4 范围内，这是因为 m_hIcon 数组变量的下标为 0～4。最后调用函数 SetClassLong()，在计时器触发时改变图标的显示。

编译并运行程序，运行结果如图 7.8 所示。

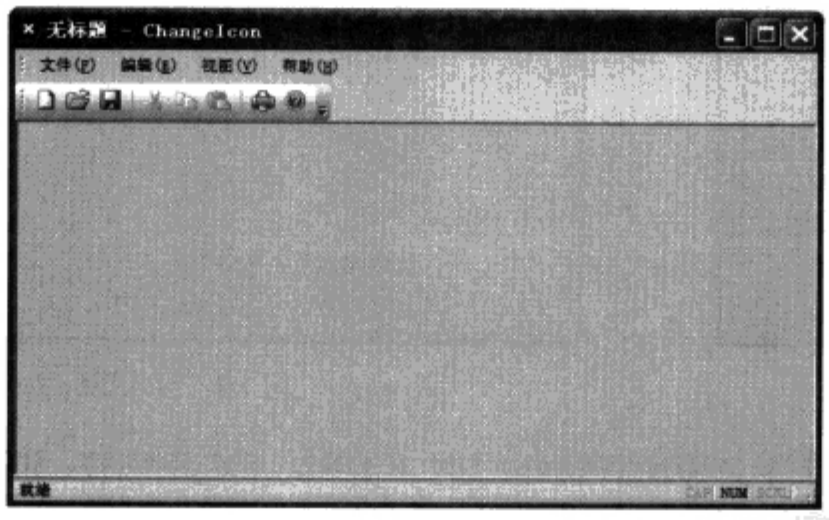


图 7.8　应用程序 窗口视图

随着时间的推进，5 个图标会交替循环显示。

7.2.2　修改背景

上一节介绍了如何修改 MFC AppWizard 生成的应用程序框架，包括窗口背景等属性，但是上一节中窗口背景只是改变了颜色，本节将向读者介绍如何将一幅图片作为窗口显示的背景。

通常情况下，将一幅位图选作窗口背景要经过以下几个步骤：

（1）创建一个兼容设备描述表；

（2）创建及装入位图资源；

（3）将位图选入兼容设备描述表；

（4）将兼容设备描述表上的内容复制到当前设备描述表上。

读者大概不能够很好地理解以上所说的几个步骤，为方便其理解，下面通过一个实例来解释以上几个步骤，完整例程请参见光盘中 ChangeBackground 例子代码，具体操作步骤如下。

步骤 1：新建一个 MFC 单文档应用程序，工程名为 ChangeBackground 或由用户自定义。

步骤 2：在资源面板中添加一幅位图资源。

可以找一幅自己喜欢的比较漂亮的位图，复制到本工程文件夹下的 res 文件夹下，命名为 bitmap.bmp，然后打开工作台的“Resource View”页面，右击并选择“Add Resource”命令，弹出如图 7.9 所示的“Add Resource”对话框，在该对话框中选择“Bitmap”选项，然后单击对话框中的“Import”按钮。

单击“Import”按钮后，弹出如图 7.10 所示的“Import”对话框，打开 ChangeBackground\res 文件夹，选择“bitmap.bmp”文件，单击“打开”按钮将该位图导入工程中。

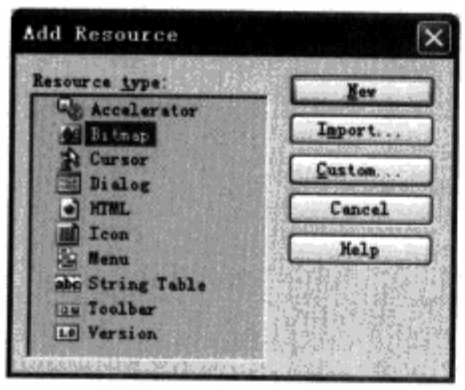


图 7.9　资源视图

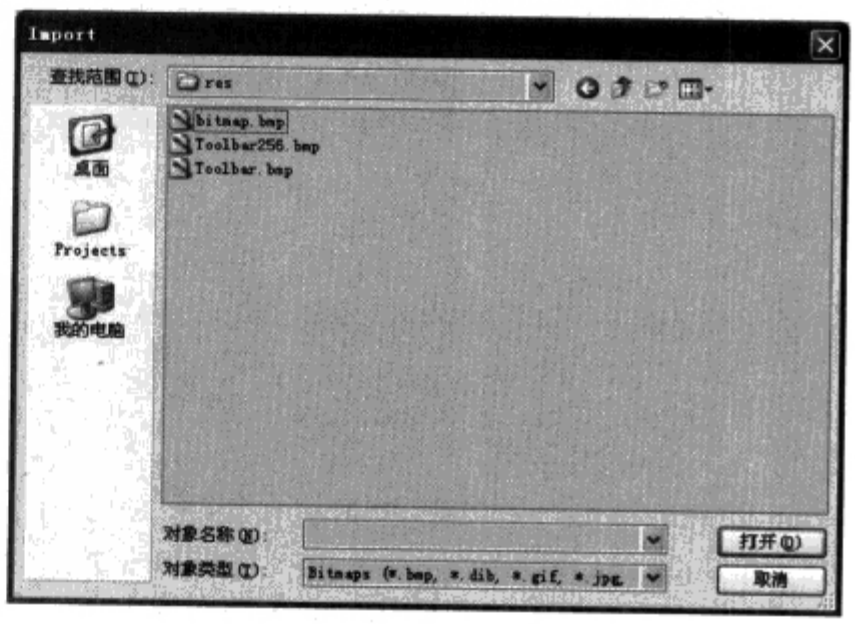


图 7.10　图标

将位图导入工程中时，会弹出如图 7.11 所示的消息对话框，该消息对话框提示用户位图已经正确地导入工程中了，但是由于位图的颜色大于 256 位色，因此不能加载到 Visual C++的位图编辑器中，并且不能编辑该位图，可以不必去理会该消息对话框，这是因为我们并不想修改这个位图。但是，如果读者在加载位图之后，希望能够修改这个位图，那么导入工程中的位图就应该是 256 位颜色以下的位图。

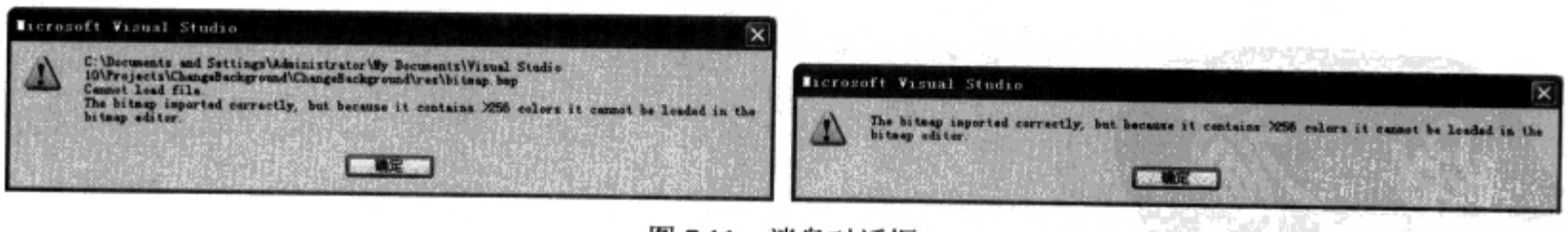


图 7.11　消息对话框

位图加载之后，就采用默认的 ID，即 IDB_BITMAP1。

步骤 3：在 CChangeBackgroundView::OnDraw()函数中贴图。

1．产生一个兼容 DC

```
void CChangeBackgroundView::OnDraw(CDC* pDC)
{
    CChangeBackgroundDoc* pDoc = GetDocument();
```

```
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;
    // TODO: 在此处为本机数据添加绘制代码
    CDC dcCompatible;
    dcCompatible.CreateCompatibleDC(pDC);
}
```

调用 CDC 类的成员函数 CreateCompatibleDC()创建兼容某一设备描述表的一个兼容 DC，函数原型如下：

```
BOOL CreateCompatibleDC( CDC* pDC );
```

该函数只有一个参数 pDC，指向要兼容的设备描述表，如果该参数为 NULL，那么创建的兼容 DC 是兼容系统显示屏幕的。本例中兼容的 DC 就是 CEX09_02View::OnDraw()函数中由系统传来的 pDC。

2．创建及装入位图资源

```
void CChangeBackgroundView::OnDraw(CDC* pDC)
{
    CChangeBackgroundDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;
    // TODO: 在此处为本机数据添加绘制代码
    CDC dcCompatible;
    dcCompatible.CreateCompatibleDC(pDC);
    CBitmap bmp;
    bmp.LoadBitmap(IDB_BITMAP1);
}
```

代码中首先创建一个 CBitmap 类的对象 bmp，然后调用 CBitmap 类的成员函数 LoadBitmap()将位图资源载入对象 bmp 中。

3．将位图选入兼容 DC

```
void CChangeBackgroundView::OnDraw(CDC* pDC)
{
    CChangeBackgroundDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;
                                        // TODO: 在此处为本机数据添加绘制代码
    CDC dcCompatible;
    dcCompatible.CreateCompatibleDC(pDC);
    CBitmap bmp;
    bmp.LoadBitmap(IDB_BITMAP1);
    dcCompatible.SelectObject(&bmp);
}
```

4．将兼容 DC 上的内容复制到当前 DC 上

```
void CChangeBackgroundView::OnDraw(CDC* pDC)
{
    CChangeBackgroundDoc* pDoc = GetDocument();
```

```
        ASSERT_VALID(pDoc);
        if (!pDoc)
            return;

                                                    // TODO: 在此处为本机数据添加绘制代码
        CDC dcCompatible;
        dcCompatible.CreateCompatibleDC(pDC);
        CBitmap bmp;
        bmp.LoadBitmap(IDB_BITMAP1);
        dcCompatible.SelectObject(&bmp);
        CRect rect;
        GetClientRect(&rect);
        pDC->BitBlt(0,0,rect.Width(),rect.Height(),&dcCompatible, 0,0,SRCCOPY);
    }
```

函数 GetClientRect()用于获取客户区的矩形区域大小，该函数只有一个参数，为 CRect 类或 RECT 结构的对象引用，调用该函数之后返回的矩形区域大小就存放在此参数中。

函数 BitBlt()用于将源 DC 上的位图复制到目的 DC，函数原型如下：

```
BOOL BitBlt( int x, int y, int nWidth, int nHeight, CDC* pSrcDC, int xSrc, int ySrc, DWORD dwRop );
```

第一个参数 x 表示目的 DC 的左上角的 x 坐标。

第二个参数 y 表示目的 DC 的左上角的 y 坐标。

第三个参数 nWidth 表示目的 DC 或源位图的宽度。

第四个参数 nHeight 表示目的 DC 或源位图的高度。

第五个参数 pSrcDC 指向包含位图的源 DC。

第六、第七个参数 xSrc、ySrc 分别表示源位图左上角的 x 和 y 坐标。

第八个参数 dwRop 表示贴图的模式，本例中选用 SRCCOPY 值，表示直接将源位图复制到目的 DC 中。

编译运行该程序，结果如图 7.12 所示，可以看出 BitBlt()函数是按 1:1 的比例将位图复制到 DC 中的，当视图窗口最大化时，可以看到整个位图；当视图窗口尺寸变小时，就只能看到位图的一部分，如果想在视图窗口上显示整个位图而不管视图窗口多大，可以用 StretchBlt()函数。

图 7.12　使用 BitBlt()函数将位图作为背景

步骤 4：修改 CChangeBackgroundView::OnDraw()函数代码，将位图拉伸显示，代码如下：

```
void ChangeBackgroundView::OnDraw(CDC* pDC)
{
    CEX09_02Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
```

```
        //产生一个兼容 DC
        CDC dcCompatible;
        dcCompatible.CreateCompatibleDC(pDC);
        //创建及装入位图资源
        CBitmap bmp;
        bmp.LoadBitmap(IDB_BITMAP1);
        //将位图选入兼容 DC
        dcCompatible.SelectObject(&bmp);
        //将兼容 DC 上的内容复制到当前 DC 上
        CRect rect;
        GetClientRect(&rect);
        //获取位图信息
        BITMAP bitmap;
        bmp.GetBitmap(&bitmap);
        pDC->StretchBlt(0,0,rect.Width(),rect.Height(),&dcCompatible,
            0,0,bitmap.bmWidth,bitmap.bmHeight,SRCCOPY);
    }
```

可以看出，上段代码中把调用 BitBlt()函数的语句去掉了，而且又添加了最后的 3 条语句。新加的代码中首先定义了一个 BITMAP 结构类型的对象 bitmap，BITMAP 结构包含很多数据成员，如 bmType、bmWidth、bmHeight 等，这些数据成员包含位图的相关信息。

代码 bmp.GetBitmap(&bitmap);是将位图 bmp 的信息及属性取出并放到先前定义的 bitmap 变量中。

最后一句代码调用函数 StretchBlt()将位图贴到视图窗口，并且调节位图大小来适应视图窗口尺寸的改变，函数原型如下：

```
BOOL StretchBlt( int x, int y, int nWidth, int nHeight, CDC* pSrcDC, int xSrc, int ySrc, int nSrcWidth, int nSrcHeight,
DWORD dwRop );
```

第一、第二个参数 x 和 y 分别表示目的矩形的左上角的 x 坐标和 y 坐标。

第三、第四个参数 nWidth 和 nHeight 分别表示目的矩形的宽度和高度。

第五个参数 pSrcDC 指向包含位图的源 DC。

第六、第七个参数 xSrc、ySrc 分别表示源矩形（即源 DC）左上角的 x 和 y 坐标。

第八、第九个参数 nSrcWidth、nSrcHeight 分别表示源矩形（即源 DC）的宽度和高度，本例中通过调用 bitmap.bmWidth 和 bitmap.bmHeight 取得位图的宽度和高度，这两个值也就是源 DC 的高度和宽度。其中 bmWidth 和 bmHeight 是 BITMAP 结构中用来表示位图宽度和高度的两个数据成员。

编译并运行程序，结果如图 7.13 所示，不管视图窗口怎样改变，都可以显示完整的位图图片，只是效果不太好。

图 7.13　使用 StretchBlt()函数将位图作为背景

系统在发出 WM_PAINT 消息之前，会发出 WM_EARSEBKGND 消息，在该消息处理函数 OnEraseBkgnd()中将执行真正的刷新背景的操作，因此也可以将 CChangeBackgroundView::OnDraw() 函数中的代码剪贴到 WM_EARSEBKGND 消息的响应函数 CChangeBackgroundView::OnEraseBkgnd() 中，然后将原来的 return CView::OnEraseBkgnd(pDC);改为 return true;。

7.3　工具栏编程与美化

工具栏是 Windows 应用程序中另一个非常重要的图形界面元素，它提供了一组顺序排列的带有位图图标的按钮。通常，用户通过位图就可以知道这个按钮所要实现的功能，因此操作工具栏比操作菜单更方便、快捷。在应用程序中，可以将常用的功能封装在工具栏中。工具栏是一个特殊的窗口对象，可以停靠在主框架窗口的一边，也可以浮动在窗口中。

7.3.1　创建工具栏

在一个应用程序中，可以有一个以上的工具栏，因此，开发人员不仅可以在 MFC 提供的工具栏的基础上增加按钮，还可以自己创建一个工具栏并将其添加到应用程序中。创建一个新的 MFC Appwizard 应用程序，命名为 AddToolBar，如图 7.14 所示；然后给它添加一个工具栏，该工具栏上有 4 个按钮，分别为“红色”、“绿色”、“蓝色”、“黄色”。下面将介绍一下新建工具栏的详细步骤。

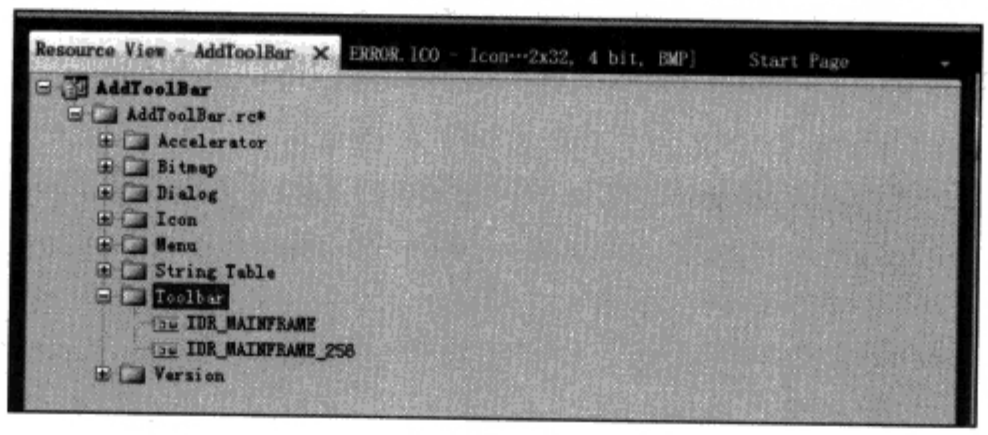


图 7.14　增加工具栏资源

1．增加一个工具栏资源

前面讲述的是在原工具栏的基础上增加一个工具栏按钮，现在增加一个新的工具栏资源。

（1）打开 AddToolBar 工程，在工作台的资源面板上，右击“Toolbar”文件夹，弹出一个快捷菜单。

（2）在快捷菜单中选择“Insert Toolbar”命令，在工作台的 Toolbar 的文件夹下面多了一个工具栏资源，应用程序给了它一个默认的 ID 号——IDR_TOOLBAR1。

（3）在右侧的工具栏编辑器中出现了一个空白的工具栏按钮，单击这个工具栏按钮，设计下面的按钮位图，在上面填充红色，用来表示“红色”按钮，如图 7.15 所示。

（4）双击这个工具栏按钮，弹出按钮属性面板，将这个按钮的 ID 设置为“绘图”菜单中“红色”菜单命令的 ID 号——ID_GRAPH_RED，并在“Prompt”栏中输入提示信息“红色按钮”，结果如图 7.16 所示。

（5）设计完该按钮之后，它后面又显示了一个空的工具栏按钮，按照上面讲的方法设计剩余的 3 个工具栏按钮：绿色、蓝色、黄色。结果如图 7.17 所示。

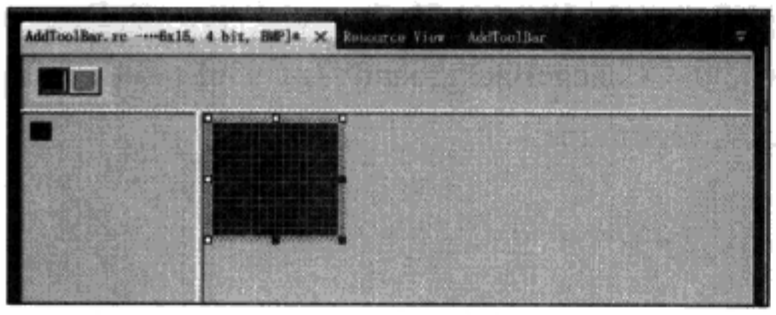


图 7.15 “红色”按钮

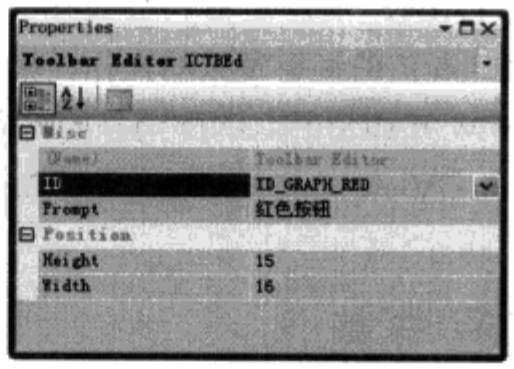


图 7.16 输入提示信息

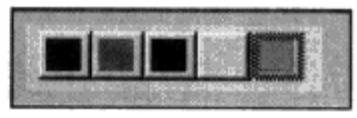


图 7.17 工具栏按钮

2．在 CMainFrame 框架类的头文件中，声明一个 CToolBar 类的对象

在工作台的 MFC ClassView 页面，右击 CMainFrame，在弹出的快捷菜单中选择“Add Member Variable”命令，设置变量类型为 CToolBar，变量名为 m_newToobar，访问类型为 potected。

或者直接找到框架类的头文件中声明变量的位置，输入如下代码：

```
class CMainFrame : public CFrameWnd
{…
protected:   // control bar embedded members
      CStatusBar   m_wndStatusBar;
      CToolBar     m_wndToolBar;
      CToolBar     m_newToolBar;
…
}
```

3．在 CMainFrame 的 OnCreate()函数中完成工具栏窗口的创建和加载

（1）创建工具栏窗口并加载自定义的工具栏资源。代码如下：

```
    if (!m_newToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE | CBRS_RIGHT | CBRS_GRIPPER |
CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) || !m_newToolBar.LoadToolBar (IDR_TOOLBAR1))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;          // fail to create
    }
```

这次将自己创建的工具栏停靠在框架窗口的右侧。其中 IDR_TOOLBAR1 就是新创建的工具栏资源的 ID。

（2）使 ID 号为 IDR_TOOLBAR1 的工具栏可停靠。代码如下：

```
    m_newToolBar.EnableDocking(CBRS_ALIGN_ANY);
```

（3）停靠工具栏在主框架窗口上。代码如下：

```
    DockPane(&m_newToolBar);
```

运行结果如图 7.18 所示。不难看出，新创建的工具栏上的 4 个按钮完全显示在程序运行结果界面中，而开发人员没有添加任何代码。

7.3.2 在工具栏中添加、删除按钮

展开工作台资源面板中的“Toolbar”文件夹，双击下面的 ID 号为 IDR_MAINFRAME 的工具栏资源，在窗口的右侧显示了工具栏编辑器窗口，如图 7.19 所示。

在工具栏编辑器窗口的上面显示了 AppWizard 添加的工具栏，工具栏的下面是两个分隔窗格，左侧窗格显示的是工具栏按钮上的位图的缩略图，右侧窗格显示的是工具栏按钮上的放大的位图，可以

在这个窗格中修改位图。

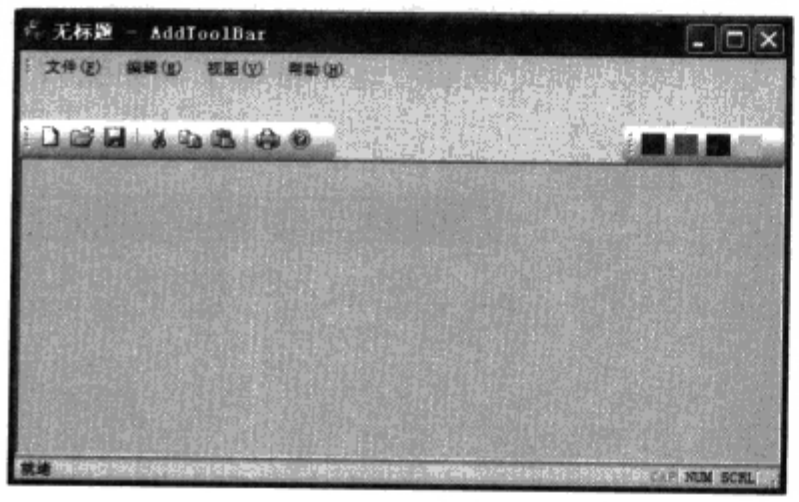


图 7.18　按钮视图

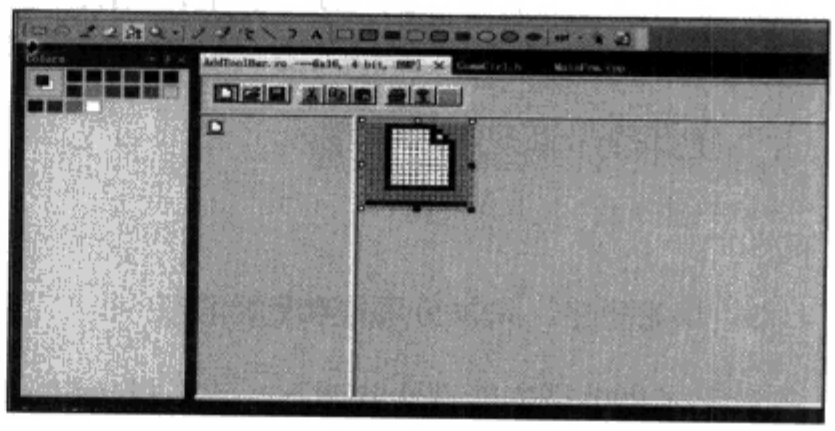


图 7.19　图标

1．编辑位图按钮

在工具栏编辑窗口中显示的工具栏的后面有一个空白的按钮，现在给这个按钮添加一个位图。可以在编辑器右侧的分隔窗格中编辑位图，例如在该位图上画一个红色的交叉线，如图 7.20 所示，当然，读者可以通过编辑窗口颜色栏和图形栏画出更丰富的位图。

2．修改工具栏按钮的属性

双击刚刚添加位图的工具栏按钮，就会打开工具栏按钮属性面板，如图 7.21 所示。

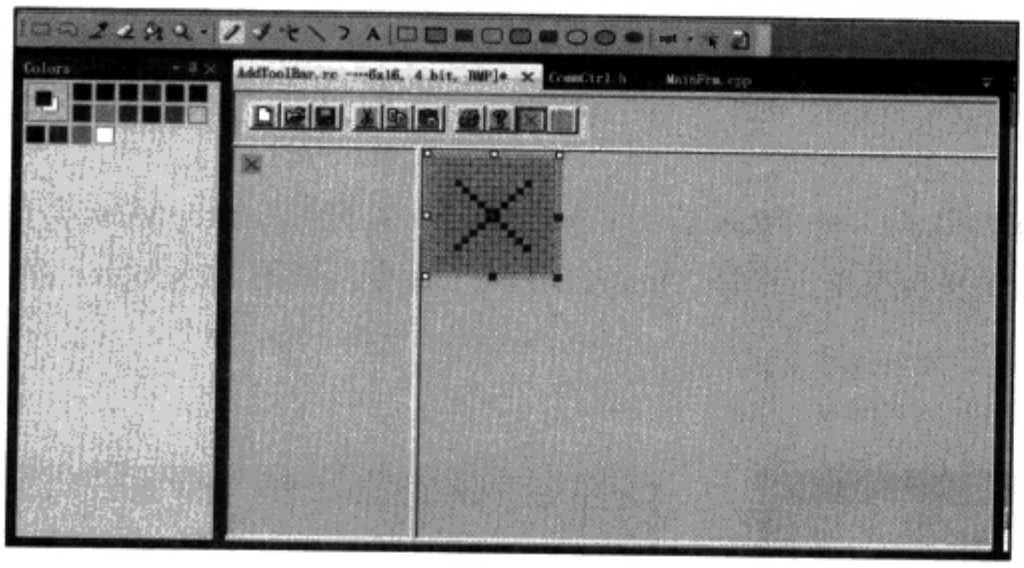


图 7.20　图标示图

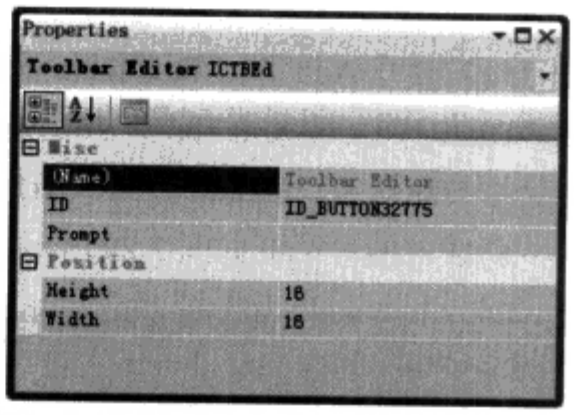


图 7.21　工具栏按钮属性

在这个对话框中可以设置工具栏按钮的 4 个属性。

- ID，表示工具栏按钮的 ID 号，它后面是一个下拉列表框，在该列表框中应用程序为这个工具栏按钮自动分配了一个 ID 号，这个 ID 号看上去一点意义也没有，一般不用这个默认的 ID。由于工具栏按钮都对应于菜单中的某个菜单命令，为了和菜单项相关联，因此，通常在下拉列表框中选择该工具栏按钮所对应的那个菜单命令的 ID 作为其 ID 号。这里，暂时不把它和菜单命令关联，就用默认的 ID。
- Width，表示工具栏按钮的宽度，以像素为单位，默认宽度为 16 个像素。
- Height，表示工具栏按钮的高度，以像素为单位，默认高度为 15 个像素。

在修改工具栏的高度或宽度时，为达到整体美观的效果，还应该稍微调整一下其位图的大小。

- Prompt，表示工具栏按钮的提示信息。

添加该按钮后，运行程序，结果如图 7.22 所示。

可以看到在工具栏的尾部多了一个添加的按钮，它现在是灰白的，并不可用这是因为在添加其 ID 属性时，用的是默认的 ID 号，而并没有给这个 ID 号添加消息处理函数。假如把这个按钮的 ID 改为菜单命令“工具栏”的 ID，即 ID_VIEW_TOOLS_BAR，就可以响应这个按钮了，如图 7.23 所示。

图 7.22　工具条视图

图 7.23　工具条视图

3．调整顺序、增加分隔线和删除按钮

在工具栏编辑器的设计状态下，工具栏按钮之间有一个小小的间隔；在应用程序运行时，工具栏按钮之间的小间隔变成了一条竖线，这条竖线就是分隔线。增加分隔线的方法很简单，只需要在工具栏编辑器中，直接用鼠标拖动要分隔的按钮，往旁边移动一点点距离，然后释放就可以了。例如，要将刚刚添加的红色交叉线的按钮和带有问号位图的按钮分隔开，只需要用鼠标拖动红色交叉线的按钮往右侧移动一点然后释放即可，结果如图 7.24 所示。在按钮的旁边多了一条分隔线。

调整按钮顺序也很简单，只要用鼠标拖动按钮到希望的位置，然后释放鼠标就可以了。

图 7.24　工具条视图

删除按钮的方法和删除菜单的方法不同，若想删除某个菜单命令，只要选中该菜单命令，然后按【Delete】键即可，而删除按钮不能采用这样的方法。要删除一个按钮，只需要用鼠标按住想要删除的按钮，然后将其拖到下面的空白区域释放即可。

7.3.3　从对话框创建工具栏

（1）打开 Add Tool Bar 项目，选择“View”→“Resource View”命令，打开 Resource View 页面，在“Dialog”文件夹上右击，并在弹出的快捷菜单上选择“Insert Dialog”命令，如图 7.25 所示。新建对话框的默认 ID 为 IDD_DIALOG1。

新插入的对话框如图 7.26 所示。

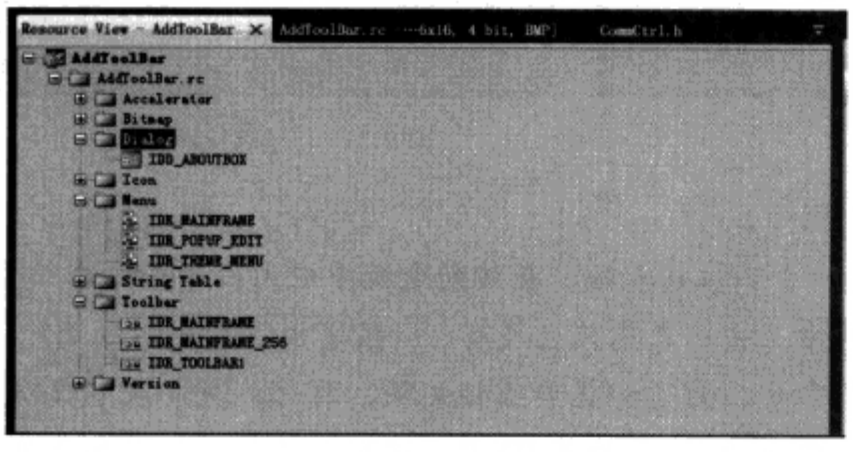


图 7.25　资源视图

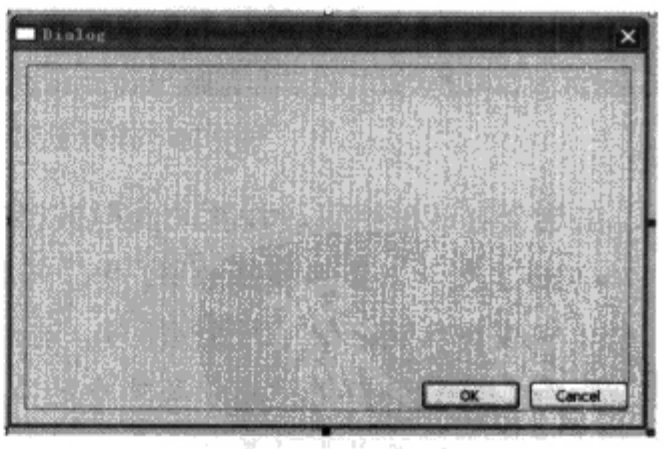


图 7.26　对话框视图

（2）在新插入的对话框资源上右击，并在弹出的快捷菜单中选择“Add Class”命令，在弹出的 MFC Class Wizard 对话框中，“Class name”文本框中输入 CDialogToolBar，并单击“Finish”按钮结束，如图 7.27 所示。之后会自动跳转到 DialogToolBar.h 文件中。

（3）在对话框的类定义 CDialogToolBar.h 中，加入如下代码：

```
CToolBar m_ToolBar;
```

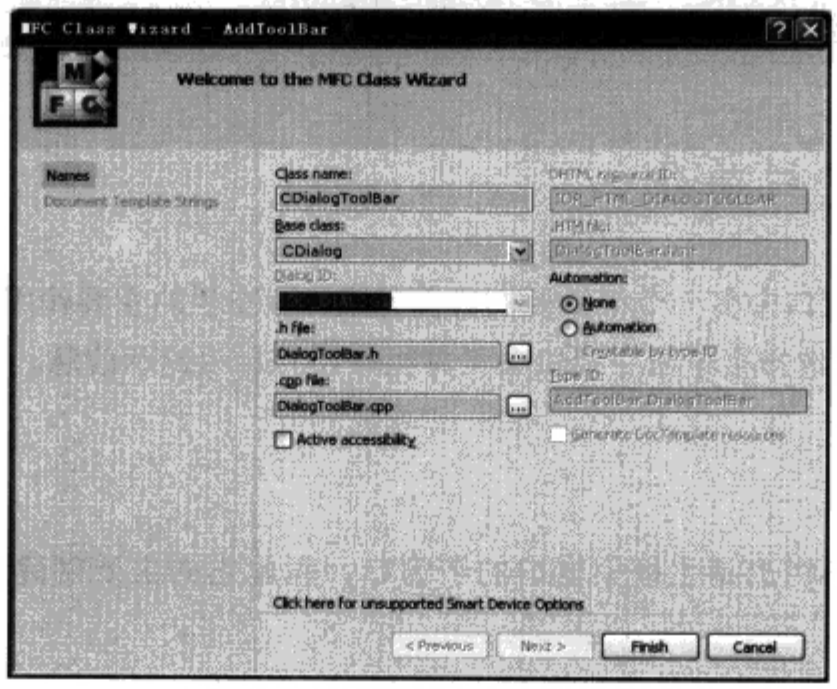


图 7.27　类向导视图

（4）在 OnInitDialog()中或其他合适的消息响应中加入如下代码（函数可查看 MSDN）：

```
m_ToolBar.Create(this);                          //创建工具栏
m_ToolBar.LoadToolBar(IDR_TOOLBAR);              //加载工具栏
                                                 //得出控件条大小.
CRect rect;
CRect rectNow;
GetClientRect(rect);
RepositionBars(AFX_IDW_CONTROLBAR_FIRST,AFX_IDW_CONTROLBAR_LAST,0,reposQuery,rectNow);
                                                 //设置控件条位置
CPoint ptOffset(rectNow.left-rect.left,rectNow.top-rect.top);
CRect rcChild;
CWnd* pwndChild=GetWindow(GW_CHILD);
while (pwndChild)
{
    pwndChild->GetWindowRect(rcChild);
    ScreenToClient(rcChild);
    rcChild.OffsetRect(ptOffset);
    pwndChild->MoveWindow(rcChild,FALSE);
    pwndChild=pwndChild->GetNextWindow();
}
                                                 //调整对话框尺寸
CRect rcWindow;
GetWindowRect(rcWindow);
rcWindow.right+=rect.Width()-rectNow.Width();
rcWindow.bottom+=rect.Height()-rectNow.Height();
MoveWindow(rcWindow, FALSE);
                                                 //控件条定位
RepositionBars(AFX_IDW_CONTROLBAR_FIRST,AFX_IDW_CONTROLBAR_LAST,0);

                                                 //窗口居中
CenterWindow();
```

（5）手工添加处理函数。

```
afx_msg void OnBtnXXX();                         //消息响应函数声明
```

```
ON_COMMAND(ID_BTN_XXX/*工具按钮 ID*/,OnBtnXXX/*函数名*/)   //消息映射
void CXXXDlg::OnBtnXXX(){}                                  //消息处理函数
```

7.4 状态栏编程与美化

状态栏是 Windows 窗口中另一个很重要的界面元素。状态栏的主要作用是显示应用程序目前的基本状态，同时还会在用户选择菜单命令或单击工具栏按钮时显示提示信息。

7.4.1 创建状态栏

与工具栏一样，状态栏也是主框架窗口的子窗口，因此它的主要创建步骤也是在框架窗口类的 OnCreate()函数中完成的。

（1）在 CMainFrame 类的头文件中声明 CStatusBar 类的对象。

```
class CMainFrame : public CFrameWnd
{ …
protected:   // control bar embedded members
      CStatusBar   m_wndStatusBar;
      CToolBar      m_wndToolBar;
      …
      };
```

（2）在 CMainFrame 类的实现文件中，即 OnCreate()函数中创建状态栏窗口。

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

在 CMainFrame 类的 OnCreate()函数中，首先调用 CStatusBar 的 Create()函数创建状态栏窗口：m_wndStatusBar.Create(this) 。 然 后 再 调 用 CStatusBar::SetIndicatiors() 函 数 来 分 配 窗 格 ：m_wndStatusBar.SetIndicators(indicators, sizeof(indicators)/sizeof(UINT))，其中第一个参数 indicators 是一个指示器数组；第二个参数表明该状态栏的窗格总数。

一般情况下，一个应用程序只能有一个状态栏，因此如果要使应用程序显示自定义的提示信息，通常的做法是修改原有的状态栏，在上面添加窗格。

7.4.2 在状态栏中插入进度条

在状态栏上显示一个进度条的操作就不会像前两项操作那么简单了，这里，需要再另外定义一个 CProgressCtrl 类的变量来产生进度条，具体操作如下。

步骤 1：在 CMainFrame 中定义成员变量，用于产生进度条。

在 CMainFrame 类的头文件中声明 CProgressCtrl 类的对象如下：

```
CProgressCtrl   m_Progress;
```

步骤 2：添加一个新的状态栏指示器，ID 为 ID_INDICATOR_PROG，并在资源面板中再加入一个字符串资源，其 ID 同样为 ID_INDICATOR_PROG，字符串文本内容为“显示进度条”。

步骤 3：在 CMainFrame 中加入 WM__PAINT 的响应函数 OnPaint()。由于时间每变化一次或者进度条每前进一块，就会重画窗口，因此在 OnPaint()中显示进度条。代码如例 7.3 所示。

例 7.3　CmainFrame::OnPaint 函数代码

```
1   CRect rect;
2   m_wndStatusBar.GetItemRect(2,&rect);
3   if(m_Progress.m_hWnd)
4   {
5       m_Progress.SetWindowPos(NULL,rect.left,rect.top,
6           rect.Width(),rect.Height(),SWP_NOZORDER);
7   }
8   else
9   {m_Progress.Create(WS_CHILD|WS_VISIBLE,rect,&m_wndStatusBar,111);}
```

第 2 行代码调用 GetItemRect()函数取得显示进度条的大小，其中第一个参数设置为 2，是因为进度条指示器在 indicators 数组中的序号为 2。第 5 行代码调用函数 SetWindowPos()调整进度条的位置和大小。第 9 行代码用的 Create()函数是 CProgressCtrl 类的成员函数，用来创建一个进度条。

步骤 4：在 OnTimer()函数中，使进度条向前走，代码如下：

```
m_Progress.StepIt();
```

其中 StepIt 是 CProgressCtrl 的成员函数，使进度条向前走一步并且重画进度条。

至此，关于状态栏的操作完成了，运行结果如图 7.28 所示。

图 7.28　状态栏视图

7.5　鼠标光标编程

7.5.1　鼠标光标编程步骤

用 Visual C++开发 Windows 应用程序时，可以分 4 步对窗口上的鼠标光标进行定制：手工编辑 Cursor，定义 Cursor 变量，加载 Cursor 变量，控制 Cursor 行为。每个步骤的详细说明如下。

（1）手工编辑 Cursor。在资源编辑器中增加一个 Cursor，Visual C++会自动给它分配一个 ID（比如 IDC_CURSOR1），然后手工编辑 Cursor 的形状，编辑好后保存。可以看到 Cursor 的属性面板中，“File name”项标明了该 Cursor 资源的文件路径（res\cursor1.cur）

（2）定义 Cursor 变量。定义一个 HCURSOR 类型的变量 myCursor，用于存放 Cursor 对象。

```
HCURSOR myCursor;
```

（3）加载 Cursor 变量。在需要改变鼠标光标的函数中添加 Cursor 的加载代码，将定义好的 Cursor 资源加载到前面的 Cursor 变量中，如果想程序一开始就自动显示个性化定制的鼠标光标，则在初始化函数（如对话框初始化函数 OnInitDialog()）中加载 Cursor。Visual C++加载 Cursor 的方法很多，可以用 CWinApp::LoadCursor()加载，也可以用 Windows API 中的 LoadCursor()加载。但这两种方法都只能改变 Cursor 的形状，不能改变其大小。如果需要改变大小就只能用 LoadImage()，代码如下：

```
m_cursor = AfxGetApp()->LoadCursor(IDC_CURSOR1);
```

也可以

```
m_cursor = LoadCursorFromFile(".\\res\\123.cur");
```

或者

```
myCursor=(HCURSOR)LoadImage(NULL,"res/cursor1.cur",IMAGE_CURSOR,72,32,LR_LOADFROMFILE );
```

LoadImage()这是一个比较复杂的 Windows API 函数，有 6 个参数，可以向内存中加载位图、ICON 和 Cursor。函数原型如下

```
HANDLE LoadImage(
HINSTANCE hinst,
| LPCTSTR lpszName,
UINT uType,
int cxDesired,
int cyDesired,
UINT fuLoad );
```

下面只介绍与加载 Cursor 相关的参数用法。

- hinst：指向需加载该资源的 module，一般可以将这个参数设为 NULL。
- lpszName：标识要加载的资源，如果 fuLoad 设为 LR_LOADFROMFILE，则该参数为资源的文件名，如位图文件、Icon 文件，当然也包括 Cursor 文件。
- uType：指明加载的资源类型，它的值可以是 IMAGE_BITMAP，加载位图；IMAGE_CURSOR，加载 Cursor；IMAGE_ICON，加载 Icon。
- cxDesired：光标的宽度，值是以像素为单位的整数。
- cyDesired：光标的长度，值是以像素为单位的整数。
- fuLoad：资源加载的方式，当值为 LR_LOADFROMFILE 时，指从文件加载。

（4）控制 Cursor 行为。最终控制 Cursor 的行为要通过 OnSetCursor()函数来实现。用 ClassWizard 为需要改变鼠标光标的窗口对象（CWin 的子类）添加 OnSetCursor()函数来处理窗口的 ON_WM_SETCURSOR 消息。程序运行时，只要鼠标进入该窗口的区域，Windows 就会给该窗口对象发送 ON_WM_SETCURSOR 消息，并触发 OnSetCursor()函数。所以可以在 OnSetCursor()函数中增加控制鼠标光标的逻辑，如可以简单地显示个性化光标 SetCursor(myCursor)；也可以加入一些复杂的逻辑，比如在一定的区域内显示个性化光标，在其他区域显示标准光标，当然这就因实际需要而定了。

重载函数格式如下：

```
OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT message),
```

显示光标函数格式如下：

```
::SetCursor(m_cursor);
```

至此，自定义的鼠标完成；当然也可以用.ani 格式的动态鼠标形状使程序更漂亮；

和载入.cur 一样：例如；

```
m_cursor = LoadCursorFromFile(".\\res\\123.ani");
```

此外，消息步骤还可一步完成，格式如下：

```
SetCursor(AfxGetApp()->LoadStandardCursor(IDC_CROSS));
```

或是：

```
HCURSOR   hc;
hc=LoadCursor(NULL,IDC_CROSS);
SetCursor(hc);
```

鼠标光标如表 7.1 所示。

表 7.1　　鼠标光标类型

ID	光　　标
IDC_APPSTARTING	带小沙漏的标准箭头
IDC_ARROW	标准箭头
IDC_CROSS	十字光标
IDC_HAND	Windows　2000：手型
IDC_HELP	带问号的箭头
IDC_IBEAM	I 型标
IDC_ICON	obsolete for applications marked version 4.0 or later.
IDC_NO	禁止符号
IDC_SIZE	obsolete for applications marked version 4.0 or later. use idc_sizeall.
IDC_SIZEALL	十字箭头
IDC_SIZENESW	指向东北和西南的双向箭头
IDC_SIZENS	指向南和北的双向箭头
IDC_SIZENWSE	指向西北和东南的双向箭头
IDC_SIZEWE	指向东西的双向箭头
IDC_UPARROW	上箭头
IDC_WAIT	沙漏

当然，也可以先把文件加载进来直接导入，步骤如下：

（1）先把.ani 文件导入；

（2）随便保存一个名称，然后在程序的.rc 文件下，把对应的代码修改成以下代码：

```
IDR_FLY          ANICURSOR DISCARDABLE   "fly.ani"
```

或者

```
IDR_FLY          CURSOR DISCARDABLE   "fly.ani"
```

不然有时可能出现错误，或者直接在.rc 文件下添加代码：

```
IDR_FLY          ANICURSOR DISCARDABLE   "fly.ani"
```

或者

```
IDR_FLY          CURSOR DISCARDABLE   "fly.ani"
```

就不用第（1）步骤了；然后

```
m_cursor = AfxGetApp()->LoadCursor("IDR_FLY");
```

这样就可以导入文件，.ani 格式可以通过由动画或.gif 格式转换过来。

7.5.2　鼠标的消息处理机制

Windows 编程中有两种改变指针的方法：一种是当应用程序的主窗口类注册时，为 wndclass 结构提供一个全程光标指针；另一种是通过在程序中处理 wm_setcursor 消息来设置鼠标光标。标准的 MFC 应用程序使用第一种方法自动在主窗口注册时将光标指针设置为一个箭头。如果要改变光标指针，则可以通过在主窗口或子窗口中重载消息 WM_SETCURSOR 的处理函数来重新设置鼠标指针：

```
// handle wm_setcursor in button class
bool cmybutton::onsetcursor(cwnd* pwnd, uint nhittest, uint msg)
{
    ::setcursor(m_hmycursor);
    return true;
}
```

当用户将鼠标指针移到按钮上时，鼠标不被捕获，Windows 发送一个 WM_SETCURSOR 消息到按钮。从上面 onsetcursor()的代码中可以看到，它传递的第一个参数是窗口句柄，即鼠标指针所指的窗口，这里指的是按钮本身；传递的第二个参数是 nhittest，这是一个鼠标单击测试代码，它以 htxxx 开头，用于 WM_NCHITTEST 消息；传递的第三个参数是触发事件的鼠标消息的消息 ID，例如，WM_MOUSEMOVE。WM_SETCURSOR 是专门用来设置鼠标指针的消息，当设置了鼠标指针以后，应该让它返回 True 以防止 Windows 再作默认处理。

WM_SETCURSOR 的处理机制是这样的：如果有父窗口，默认的窗口过程首先发送 WM_SETCURSOR 消息到父窗口，如果父窗口处理 WM_SETCURSOR 消息，则 Windows 不再做任何多余的事情，处理完消息便结束。如果父窗口不处理 WM_SETCURSOR 消息，Windows 让子窗口来处理 WM_SETCURSOR，如果子窗口也不做任何处理，Windows 使用全程光标指针，如果没有全程光标指针，则使用箭头指针。

如果在程序中要使用动态光标指针，必须决定是在子窗口中处理 WM_SETCURSOR 消息还是在父窗口中处理 WM_SETCURSOR 消息。两种方法各有利弊，应该根据具体情况而定。一般总是让对象决定它们自己的行为属性，也就是说最好在子窗口中做处理。本文中的子窗口为按钮，这就要派生一个新的按钮类，新的按钮类有自己的消息映射及其消息处理过程。可以使用类向导来产生新的按钮类，但那样要做的事情太多。如果已经有自己的按钮类，那当然是在自己的按钮类中处理 WM_SETCURSOR 消息。如果没有自己的按钮类而又想偷懒，那也可以在对话框中处理 WM_SETCURSOR 消息。

7.5.3 示例

以第 6 章中的 MiniDraw 程序为例讲解。

在 MiniDraw 程序中，可以修改程序，使鼠标光标位于视图窗口的绘图区时呈现十字形（表示可以画直线），而位于绘图区外时呈现标准箭头形状（表示不能画直线）。为此，在 MiniDrawView.h 的 CMiniDrawView 类定义中加入数据成员 m_HArrow：

```
class CMiniDrawView : public CScrollView
{
protected:
    CString m_ClassName;
    int m_Dragging;
    HCURSOR m_HArrow;
    CURSOR m_HCross;
    CPoint m_PointOld;
    CPoint m_PointOrigin;
protected: // 仅从序列化创建
    CMiniDrawView();
    DECLARE_DYNCREATE(CMiniDrawView)
// 属性
public:
    CMiniDrawDoc* GetDocument() const;
// 操作
public:
// 重写
public:
```

```
        virtual void OnDraw(CDC* pDC);   // 重写以绘制视图
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    protected:
        virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
        virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
        virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
        void OnLButtonDown(UINT nFlags, CPoint point);
        void OnMouseMove(UINT nFlags,CPoint point);
        void OnLButtonUp(UINT nFlags, CPoint point);
        void OnInitialUpdate();
    // 实现
    public:
        virtual ~CMiniDrawView();
    #ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
    #endif
    protected:
    // 生成的消息映射函数
    protected:
        afx_msg void OnFilePrintPreview();
        afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
        afx_msg void OnContextMenu(CWnd* pWnd, CPoint point);
        DECLARE_MESSAGE_MAP()
    };
```

并在 MiniDrawView.cpp 的 CMiniDrawView 构造器中初始化 m_HArrow 如下：

```
// CMiniDrawView 构造/析构
CMiniDrawView::CMiniDrawView()
{
    // TODO: 在此处添加构造代码
    m_Dragging=0;
    m_HArrow=AfxGetApp()->LoadStandardCursor(IDC_ARROW);
    m_HCross=AfxGetApp()->LoadStandardCursor(IDC_CROSS);
}
```

这个初始化语句赋予 m_HArrow 标准箭头形鼠标光标的句柄。

接着，在 MiniDrawView.cpp 的 OnMouseMove()函数中，将 ClientDC()的声明和 OnPrepareDC()与 DPtoLP()的调用移到函数的开头，并加入黑体行，使函数定义如下：

```
void CMiniDrawView::OnMouseMove(UINT nFlags,CPoint point)
{
    //TODO:Add your message handler code here and /or call
    //default
    ::SetCursor(m_HCross);
    if(m_Dragging)
    {
        CClientDC ClientDC(this);
        OnPrepareDC(&ClientDC);
        ClientDC.DPtoLP(&point);
        CSize ScrollSize=GetTotalSize();
```

```
            CRect ScrollRect(0,0,ScrollSize.cx,ScrollSize.cy);
            if(ScrollRect.PtInRect(point))
                ::SetCursor(m_HCross);
            else
                ::SetCursor(m_HArrow);
            ClientDC.SetROP2(R2_NOT);
            ClientDC.MoveTo(m_PointOrigin);
            ClientDC.LineTo(m_PointOld);
            ClientDC.MoveTo(m_PointOrigin);
            ClientDC.LineTo(point);
            m_PointOld=point;
        }
        CScrollView::OnMouseMove(nFlags,point);
    }
```

新版 OnMouseMove()不是总显示十字形光标，而是光标在绘图区内时显示十字形光标，在绘图区外时显示箭头形光标。

图 7.29 演示了绘图区小于视图窗口时的 MiniDraw 程序，显示了表示图形区边沿的直线，该图中，由于绘图区完全位于视图窗口内，MFC 代码覆盖了滚动条（窗口尺寸减小到不能完全放下图形时，滚动条会重新显示）。

编译并运行 MiniDraw 程序，测试新加入的功能。

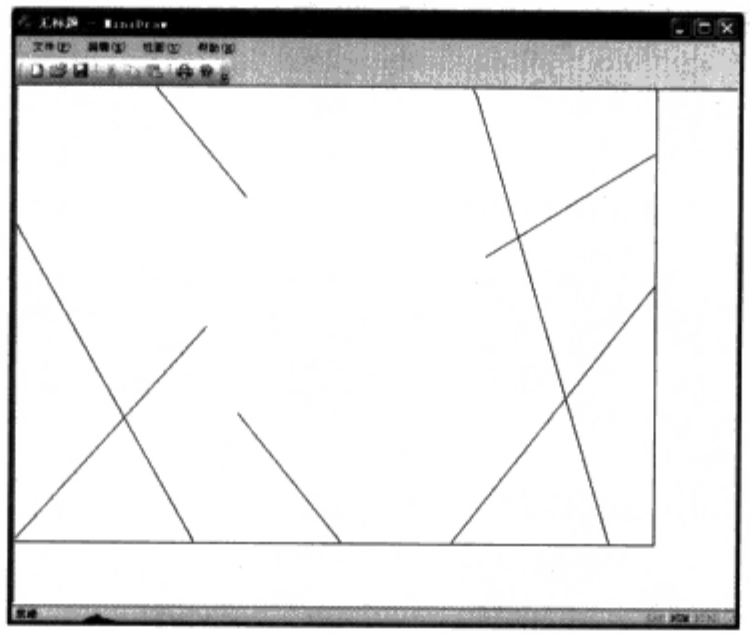


图 7.29　绘图区小于视图窗口的 MiniDraw 程序

7.6　创建启动界面

1．安装 Splash screen 组件

选择“Project”→“Add to project”→“Component and Control”命令，然后双击“Visual C++ Components”，选中 Splash screen 组件，接受默认的 ID 为 IDB_SPLASH。当然也可以自己重新设置 ID。

2．插入位图

用一幅准备好的真彩位图替换刚生成的位图即可，真彩位图在资源编辑器（它不能打开超过 256

色的图片）中是不可视的。或者，打开 Resource View 页面，在 Bitmap 上右击，选择“Add Resource”命令，并在弹出的“Add Resource”对话框中选择“Bitmap”选项，单击“Import”按钮，然后选择所要的真彩位图资源，将其 ID 设置为 IDB_SPLASH。

3．运行程序

编译并运行程序，则漂亮的真彩启动位图就显示出来了。

4．说明

（1）如果要改变启动画面的停留时间，就修改 SetTime()中的第二个参数，这里是 750ms。

```
int CSplashWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if(CWnd::OnCreate(lpCreateStruct)==-1)
        return -1;
    //Center the window;
    CenterWindow();
    //Set a timer to destroy the splash screen
    SetTimer(1,750,NULL);                    //此停留时间为 0.75 s，可通过修改第二个参数来改变闪屏时间
    return 0;
}
```

（2）基于对话框的程序不能插入闪屏，可如下设置：

首先创建一个对话框项目，将已经生成的 Splash.cpp 和 Splash.h 文件复制到工作文件夹下，并将其加入到基于对话框的项目中。

在 CDialogApp 派生类的 InitInstance()中加入下列代码：

```
#include "Splash.h"
BOOL CDialogApp::InitInstance()
{
    //CG: The following block was added by the Splash Screen component.
    {
        CCommandLineInfo cmdInfo;
        ParseCommandLine(cmdInfo);
        CSplashWnd::EnableSplashScreen(cmdInfo,m_bShowSplash);
    }
    ...
}
```

使用 Class Wizard 为在 CDialog 派生类添加 OnCreate()函数，并加入下列代码：

```
#include “Splash.h”
int CDialogDlg::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if(CDialog::OnCreate(lpCreateStruct)==-1)
        return -1;
    //CG: The following line was added by the Splash Screen component.
    CSplashWnd::ShowSplashScreen(this);
    return 0;
}
```

将 Splash.cpp 文件中的 CSplashWnd::Create()函数中的位图资源 ID 改为真彩位图的 ID 就可以了。

```
BOOL CSplashWnd::Create(CWnd* pParentWnd /*=NULL*/)
{
```

```
        if(!m_bitmap.LoadBitmap(IDB_BITMAP1))
            return FALSE;
        ...
    }
```

7.7 创建特效窗口启动应用程序

1. 利用无模式对话框显示启动画面

（1）用 AppWizard 建一个基于单文档的工程 Splash。

（2）导入用作启动画面的图片，更改 ID 为 IDB_SPLASH。

（3）新建一个对话框，在其中添加启动画面：

在资源中新建一个对话框，创建对话框类 CSplashDlg。在对话框中添加一个 Picture 控件，打开其属性面板，在“General”选项卡的“Type”下拉列表框中选择“Bitmap”选项，在“Image”下拉列表框中选择前面导入的位图资源的 ID 值——IDB_SPLASH。

（4）修改对话框的显示效果。

① 调整对话框大小，删除两个自动生成的按钮，并在属性面板的“Styles”页中取消对 Title bar 的选中；

② 选中图像，调整大小使之适应对话框的可编辑区，修改其属性面板的 Styles，使之居中。

（5）在 CMainFrame 类的 OnCreate()函数中添加创建、显示并销毁无模式对话框的代码。

```
#include "SplashDlg.h"                    //加到 MainFrm.cpp 文件的头文件调用部位
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
CSplashDlg *dlg = new CSplashDlg(this);
dlg->Create(CSplashDlg::IDD,this);    //创建对话框
dlg->ShowWindow(SW_SHOW);             //显示对话框
dlg->UpdateWindow();
Sleep(2000);                          //画面显示停留时间，单位为毫秒
...
dlg->DestroyWindow();                 //销毁对话框
return 0;
}
```

2. 通过发送消息显示和销毁启动画面

① 重复上面的添加启动界面的方法的步骤（1）～步骤（4）。

② 使用 Class Wizard 为 CMainFrame 类添加消息 WM_TIMER 的响应函数。

③ 修改代码，通过发送 WM_TIMER 消息启动和销毁启动画面。

7.8 创建特效窗口关闭应用程序

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

例如，用户单击主窗口的关闭按钮，或者选择选择系统菜单命令“关闭”，或者选择“文件”→“退出”命令，都会导致主窗口被关闭。

当用户选择“文件”→“退出”命令时，将发送 MFC 标准命令消息 ID_APP_EXIT。MFC 提供了函数 CWinApp::OnAppExit()来完成对该命令消息的默认处理。

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

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

1．边框窗口对 WM_CLOSE 的处理

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

该函数的原型如下，流程如图 7.30 所示。

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

从图 7.30 中可以看出，函数首先判断是否可以关闭窗口（m_lpfnCloseProc 是函数指针类型的成员变量，用于打印预览等情况），然后根据具体情况进行处理：

如果是主窗口被关闭，则关闭程序的所有文档，销毁所有窗口，退出程序。

如果不是主窗口被关闭，则是文档边框窗口被关闭，又可以分两种情况：若该窗口所显示的文档被且仅被该窗口显示，则关闭文档和文档窗口并销毁窗口；若该窗口显示的文档还被其他文档边框窗口所显示，则仅仅关闭和销毁文档窗口。

图 7.31 是处理 WM_CLOSE 消息时所涉及的一些函数。

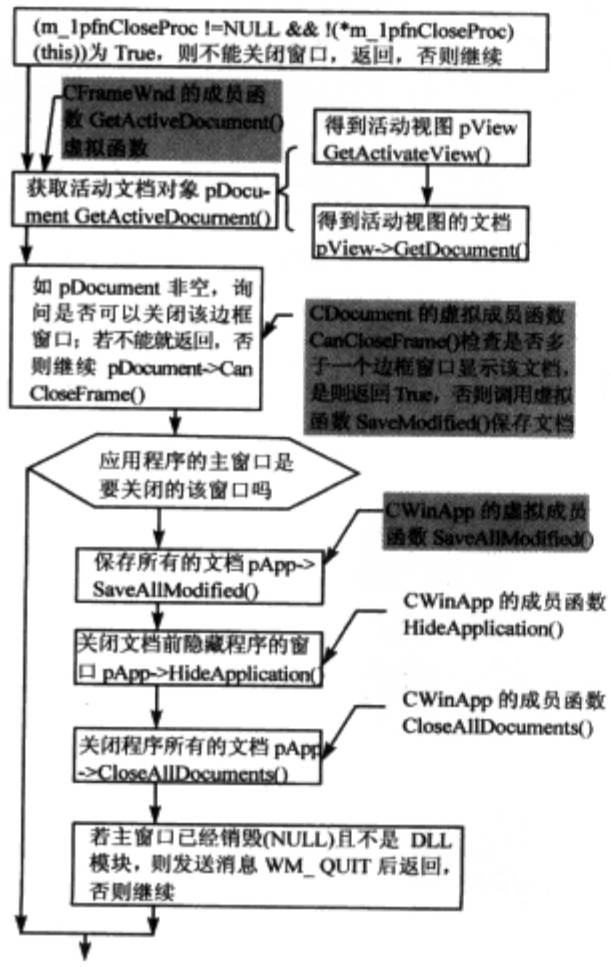


图 7.30　退出 MFC 应用程序主窗口流程图

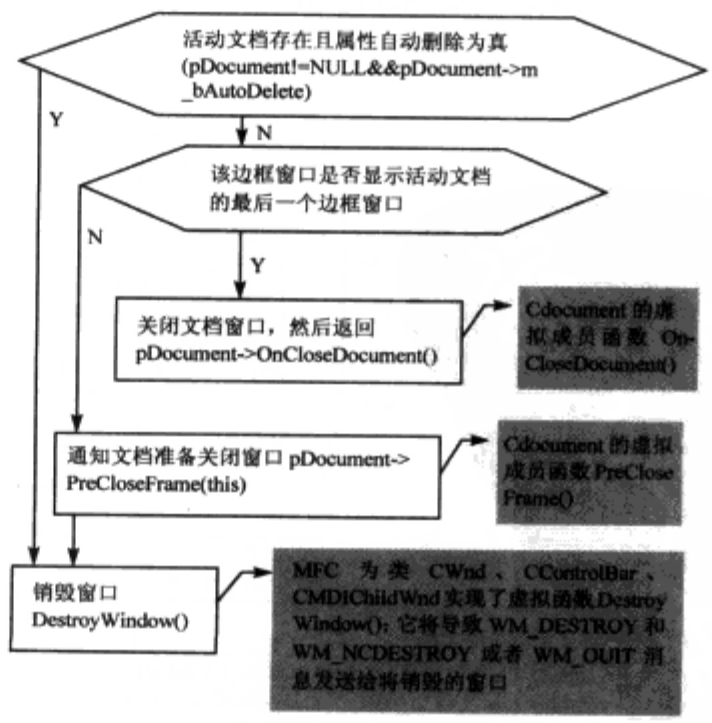


图 7.31　文档关系图

```
BOOL CDocument::SaveModified()
```

该虚拟函数的默认实现：首先调用 IsModifed()判断文档是否被修改，没有修改就返回，否则继续。

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

```
BOOL CDocument::OnSaveDocument(LPCTSTR lpszPathName)
```

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

根据参数 lpszPathName 打开文件 pFile；

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

设置鼠标为时间瓶形状；

使用 Serialize()函数完成序列化写；

完毕，恢复鼠标的形状。

```
CWinApp::SaveAllModified()
CWinApp::CloseAllDocuments(BOOL bEndSession)
```

这两个函数都遍历模板管理器列表，并分别对列表中的模板管理器对象逐个调用 CDocManager 的同名成员函数：

```
CDocManager::SaveAllModified()
CDocManager::CloseAllDocuments(BOOL bEndSession)
```

这两个函数都遍历其文档模板列表，并分别对列表中的模板对象逐个调用 CDocTemplate 的同名成员函数：

```
CDocTemplate::SaveAllModified()
CDocTemplate::CloseAllDocuments(BOOL bEndSession)
```

这两个函数都遍历其文档列表，并分别对列表中的文档对象逐个调用 CDocuemnt 的成员函数：

```
CDocument::SaveModified()
CDocument::OnCloseDocument()
CDocument::SaveModified()
CDocument::OnCloseDocument()
```

前面已对 CDocument::SaveModified()已作了解释。OnCloseDocument()是一个虚拟函数，其流程如图 7.32 所示。

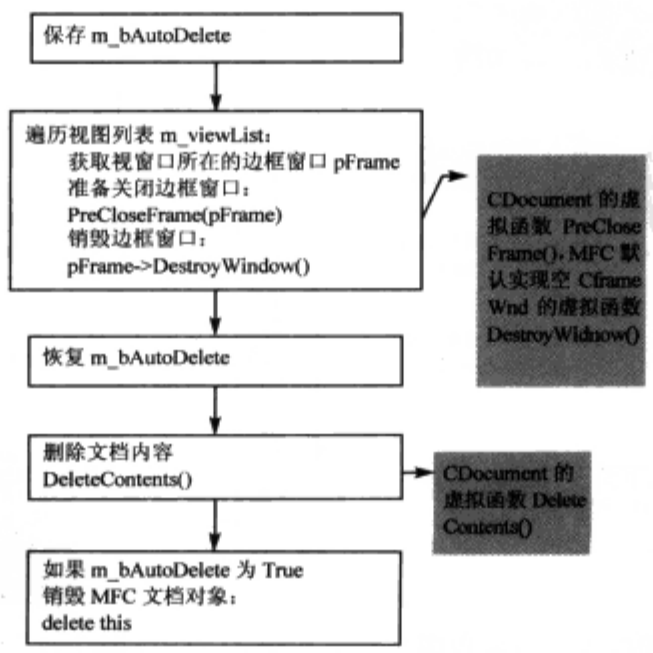


图 7.32　CDocument 的 OnCloseDocument()函数

通过文档对象所对应的视图，得到所有显示该文档的边框窗口的指针：在 SDI 程序关闭窗口时，获取的是主边框窗口；在 MDI 程序关闭窗口时，获取的是 MDI 子窗口。

然后，关闭并销毁对应的边框窗口。

如果文档对象的 m_bAutoDelete 为真，则销毁文档对象自身。

窗口的销毁所调用的函数为：

```
DestroyWindow()
```

从图 7.31 和图 7.32 中可以看出，销毁窗口是通过调用 DestroyWindow()来完成的。DestroyWindow()是 CWnd 类的一个虚拟函数。CWnd 实现了该函数，而 CMDIChildWnd 覆盖了该函数。

（1）CWnd::DestroyWindow()。

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

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

（2）CMDIChildWnd::DestroyWindow()。

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

2．处理 WM_DESTROY 消息

消息处理函数 OnDestroy()处理 WM_DESTROY 消息，CFrameWnd、CMDIChildWnd、CWnd、CView 及其派生类（如 CEditView 等）、CControlBar 等都提供了对该消息的处理函数。这里，主要讲解边框、文档边框、视图类的消息处理函数 OnDestroy()。

```
CWnd::OnDestroy()
```

调用默认处理函数 Default()。

```
CFrameWnd::OnDestroy()
```

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

```
CMDIFrameWnd::OnDestroy()
```

首先，调整客户窗口的边界类型；然后，调用基类 CFrameWnd 的 OnDestroy()。这时，MDI 子窗口的工具栏窗口列表为空，所以没有工具栏窗口可以被销毁。

```
CView::OnDestroy()
```

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

3．处理 WM_NCDESTROY 消息

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

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

```
CWnd::OnNcDestroy()
```

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

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

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

最后，调用虚拟函数 PostNcDestoy()。

```
PostNcDestoy
```

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

```
CWnd::PostNcDestroy()
```

MFC 默认实现空：

```
void CFrameWnd::PostNcDestroy()
```

调用 delete this 销毁自身这个 MFC 对象。

```
void CView::PostNcDestroy()
```

调用 delete this 销毁自身这个 MFC 对象。

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

```
CFrameWnd::~CFrameWnd()
```

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

如果有必要，删除 m_phWndDisable 数组。

```
CView::~CView()
```

在视图创建时，把自身加入到文档对象的视图列表中。现在，从列表中移走该视图对象。

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

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

参考图 7.31 分析 SDI 窗口、MDI 主、子窗口的关闭流程。

（1）SDI 窗口的关闭。

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

（2）MDI 主窗口的关闭。

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

（3）MDI 子窗口（文档边框窗口）的关闭。

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

7.9　小结

本章主要介绍了 MFC 应用程序界面与美化，包括应用程序窗口风格的美化，利用 ActiveSkin 美化窗口界面，修改窗口外观；应用程序窗口图标和背景的修改；工具栏的编程与美化；状态栏的编程与美化；鼠标光标编程；创建启动界面；创建特效窗口启动和关闭应用程序等内容。在完成本章的学习之后，读者应该：

（1）掌握 ActiveSkin 美化窗口界面的方法；

（2）掌握修改窗口外观的技术和方法；

（3）掌握修改窗口的图标和背景的步骤；

（4）掌握工具栏的创建，添加，删除按钮，以及如何从对话框创建工具栏的方法；

（5）掌握状态栏的创建，以及在状态栏中插入进度条的方法和步骤；

（6）掌握鼠标光标编程的步骤及相应的消息处理机制；

（7）理解启动界面的创建方法，以及学会创建特效窗口以启动和关闭应用程序。

第 8 章　Visual C++ 2010 MFC 文本与字体

本章主要介绍了如何使用 Visual C++ 2010 MFC 实现对文本与字体的编程。本章首先介绍了 Visual C++ 2010 MFC 提供的一个专门用来设置字体的类——CFont 类的结构与使用方法；接着介绍了如何创建文本插入符与图片插入符；然后又通过上面所介绍的知识实现了文字的输出与字体格式的改变；最后通过实现一个卡拉 OK 程序，说明了怎样输出彩色文字和变色文字，并对前面所学内容进行了总结。学习完本章之后，读者将对 Visual C++ 2010 MFC 文本与字体编程的方法有一定的了解，并且能够实现一个简单的文字处理程序。

本章主要内容有：

- CFont 字体类简介；
- 创建文本插入符与图片插入符；
- 输出文字与字体格式；
- 输出彩色文字与变色文字。

8.1　CFont 字体类简介

CFont 类是 Visual C++ 2010 MFC 提供的一个专门用来设置字体的类，这个类派生于 CGdiObject 类，它封装了一个 Windows 图形设备接口（GDI）的字体。

8.1.1　CFont 字体类成员介绍

使用 CFont 字体类时，首先要引入头文件 afxwin.h。如表 8.1 所示为 CFont 字体类的成员。

表 8.1　　CFont 字体类成员

构 造 函 数	
Cfont()	构造一个 CFont 类型的对象
初始化函数	
CreateFont()	使用指定的特征初始化一个 CFont 对象
CreateFontIndirect()	使用 LOGFONT 结构中指定的特征初始化一个 CFont 对象
CreatePointFont()	使用指定的高度和字体初始化一个 CFont 对象，高度的度量单位是 0.1 个像素点
CreatePointFontIndirect()	与 CreateFontIndirect()的功能一样，只是字体的高度是用 0.1 个像素点度量的
操作	
FromHandle()	当给定一个 Windows HFONT 时，返回一个指向 CFont 对象的指针
属性	
GetLogFont()	使用与 CFont 对象相关的逻辑字体的信息填充一个 LOGFONT 结构
operator HFONT()	返回附在 CFont 对象的字体上的 Windows GDI 句柄

在实际编程时，与其他 GDI 对象一样，当创建并初始化一个字体对象后，还必须将它选入设备描述表，之后这个新字体才能被正常使用，这可以利用 CDC 类的 SelectObject()函数来实现；同样，必须保存该函数所返回的先前的字体，并且在使用完新的字体后，还要把设备描述表中的字体恢复为先前的字体。关于设备描述表，会在第 9 章做详细介绍。

8.1.2　CFont 字体类初始化函数

在实际编程时，构造了一个 CFont 对象后，还必须利用 CFont 字体类提供的几个初始化函数之一来对 CFont 对象进行初始化。初始化函数的作用主要是将 CFont 对象与 C++字体关联起来，只有初始化之后才能使用该对象。CFont 字体类提供的初始化函数可以参见上一小节。下面将对它们进行详细介绍。

1．CFont::CreateFont()

该初始化函数的作用是使用用户指定的属性来初始化一个 CFont 字体对象。创建后的字体可被选作任何设备上下文的使用字体。使用 CreateFont()函数没有创建一个新 Windows GDI 字体，而是从 GDI 物理字体库中选择了最相近的匹配字体。当需要构造一个逻辑字体时，这些参数的大部分都可以使用默认设置，只有 nHeight 和 lpszFaceName 参数应该被指定。如果用户没有指定 nHeight 和 lpszFace Name，则该逻辑字体是与设备有关的。

该初始化函数的声明如下：

```
BOOL CreateFont(
    int nHeight,
    int nWidth,
    int nEscapenment,
    intnOrientation,
    int nWeight,
    BYTE bItalic,
    BYTE bUnderline,
    BYTE cStrikeOut,
    BYTE nCharset,
    BYTE nOutPrecision,
    BYTE nClipPrecision,
    BYTE nQuality,
    BYTE nPitchAndFamily,
    LPCTSTR lpszFacename
);
```

该函数返回值是 BOOL 类型的，如果初始化成功，则返回 True，否则返回 False。

参数说明如表 8.2 所示。

表 8.2　　CreatFont 函数参数说明

参　数	说　明
nHeight	使用逻辑单位方式指定字体的高度，字体高度可以为以下值之一： • 大于 0，此时高度被转化为设备单位，与可用字体的网格高度相比较； • 等于 0，此时使用合理的默认大小； • 小于 0，此时高度被转化为设备单位，而绝对值与可用字体的字符高度相比较。 nHeight 绝对值在转化后不可超过 16 384 设备单位。在所有的高度对比中，如果字体超过所要求的值，则字体映射器中大字体不会超过所需大小，小字体也不会超过

续表

参　数	说　明
NWidth	使用逻辑单位方式指定字体中字符平均宽度。如果为 0，那么系统会将设备方向比率与可用字体的数字方向比率作比较，找一个最近的值来匹配
nEscapement	指定偏离垂线与 *x* 轴在显示面上的夹角（用 0.1° 单位表示）。偏离垂线是一行中从开始字符到最后一个字符的线，此角从 *x* 轴逆时针方向开始度量
nOrientation	指定字符基线和 *x* 轴之间的夹角（用 0.1° 单位）
nWeight	指定字体磅数（单位是每 1000 点中墨点像素数）。尽管 nWeight 可设置为从 0 到 1000 中的任意整数值，但是一般常用值和常数如下：FW_DONTCARE0； FW_NORMAL 400；FW_REGULAR 400；FW_BOLD 700。以上各值是大约数，实际外观依赖字体大小。更多的常用值，请参见 LOGFONT 结构中的 IfWeight 成员
bItalic	指定字体是否为斜体
bUnderline	指定字体是否带下划线
bStrikeOut	指定是否字体字符突出。如果设置为非零，则字符是突出的
nCharSet	指定字体的字符集，以下是预定义的常数和值： ANSI_CHARSET 0 DEFAULT_CHARSET 1 SYMBOL_CHARSET 2 SHIFTJIS_CHARSET 128 OEM_CHARSET 255 OEM 字符集依赖于系统。具有其他字符集的字体可能存在于系统之中。字体映射器不能使用 DEFAULT_CHARSET 值，否则将使字体名和大小覆盖逻辑字体。如果指定的名称不存在，为避免不可预料的结果，应谨慎地使用 DEFAULT_CHARSET
nOutPrecision	指定所需的输出精度。输出精度定义输出与需要的字体高度、宽度、字符方向、走格、间距之间的接近程度，可以为以下值之一： OUT_CHARACTER_PRECIS OUT_STRING_PRECIS OUT_DEFAULT_PRECIS OUT_STROKE_PRECIS OUT_DEVICE_PRECIS OUT_TT_PRECIS OUT_RASTER_PRECIS
nClipPrecision	指定所需的剪贴精度。剪贴精度定义了如何剪贴部分超过剪贴范围的字符，可以为以下值之一： CLIP_CHARACTER_PRECIS CLIP_MASK CLIP_DEFAULT_PRECIS CLIP_STROKE_PRECIS CLIP_ENCAPSULATE CLIP_TT_ALWAYS CLIP_LH_ANGLES
nQuality	指示字体的输出质量，定义了 GDI 必须使逻辑字体特性和物理字体特性相匹配的程度。 可为以下值之一： DEFAULT_QUALITY 字体的外观无关紧要。 DRAFT_QUALITY 当 PROOF_QUALITY 使用时，字体的外观不太重要。对 GDI 光栅字体来说，允许缩放。在需要时可以使用黑体、斜体、下划线、突出和综合处理。 PROOF_QUALITY 字体的字符质量比精确的逻辑字体特性的匹配更重要。对 GDI 光栅字体，缩放无效，大小最接近的字体被选用。在需要时可以使用黑体、斜体、下划线、突出和综合处理
nPitchAndFamily	指定字体的间距和家族。两个低位指定字体的间距，可为以下值之一： DEFAULT_PITCH VARIABLE_PITCH FIXED_PITCH
lpszFacenameCString	指向一个以空终止字符串的指针，字符串指定字体字样的名称。此字符串的长度不能长于 30 个字符，WindowsEnumFontFamilies()函数可用于枚举所有当前可用字体。如果为 NULL，则 GDI 使用一个不依赖设备的字体

下面的代码展示了怎样使用 CreateFont()初始化一个 CFont 类对象。

```
CFont cfont;
//初始化 CFont
cfont.CreateFont(
    12,                          // nHeight
    0,                           // nWidth
    0,                           // nEscapement
    0,                           // nOrientation
    FW_NORMAL,                   // nWeight
    FALSE,                       // bItalic
    FALSE,                       // bUnderline
    0,                           // cStrikeOut
    ANSI_CHARSET,                // nCharSet
    OUT_DEFAULT_PRECIS,          // nOutPrecision
    CLIP_DEFAULT_PRECIS,         // nClipPrecision
    DEFAULT_QUALITY,             // nQuality
    DEFAULT_PITCH | FF_SWISS,    // nPitchAndFamily
    "Arial"));                   // lpszFacename
                                 // 使用初始化后的字体
CClientDC dc(this);
CFont* pold_font = dc.SelectObject(&cfont);
dc.TextOut(5, 5, _T("Hello , Visual C++ 2010"), 5);
dc.SelectObject(pold_font);
                                 // 删除初始化后的字体
cfont.DeleteObject();
```

其中阴影部分就是初始化字体所用的代码。后面展示了怎样将新建好的字体对象选入设备描述表并在屏幕上打印出来，以及最后删除新建的字体，这部分将在稍后再作解释。

2．CFont::CreateFontIndirect()

该初始化函数的声明如下：

```
BOOL CreateFontIndirect(const LOGFONT* lpLogFont);
```

该函数返回值是 BOOL 类型的，如果成功，则返回 True，否则为 False。

参数 lpLogFont 是一个指向一个定义逻辑字体特征的 LoGFont 结构的指针。所谓逻辑字体是指一个 GDI 对象，它的句柄存储在 HFONT 型态的变量中，逻辑字体是字体特征的描述。可以通过调用 CreateFont()或 CreateFontIndirect()函数来建立逻辑字体，它们是仅有的两个建立逻辑字体的函数。在使用逻辑字体绘制文字时，系统会采用一种特定的算法把逻辑字体映射为最匹配的物理字体（实际安装在操作系统中的字体）。

说明。

该函数使用 lpLogFont 指向的 LOGFONT 结构所定义的特征初始化一个 CFont 对象。

同样，要想使用 CreateFontIndirect()函数所初始化的 CFont 对象，必须先在设备上下文中选择字体，使用完成后还需要删除该 CFont 对象。因为该函数初始化的字体具有 LOGFONT 结构定义的特征，所以使用 CDC::SelectObject()函数选择该字体对象时，GDI 字体映射器将把逻辑字体和已存在的物理字体相匹配。如果找不到匹配的逻辑字体，则尽可能找与之特性匹配最多的字体替代。

下面的代码展示了怎样使用 CreateFontIndirect()函数初始化一个 CFont 类对象：

```
                                 //声明一个 CFont 对象
CFont cfont;
                                 //声明一个 LOGFONT 结构对象
```

```
LOGFONT clogfont;
                                        //设置 LOGFONT 结构的属性
memset(&clogfont, 0, sizeof(LOGFONT));
                                        //设置字体高度为 12 个像素点
clogfont.lfHeight = 12;
                                        //设置字体名字为“Arial”
_tcsncpy_s(clogfont.lfFaceName, LF_FACESIZE,
    _T("Arial"), 7);
                                        //使用上面得到的 LOGFONT 结构对象初始化 cfont
VERIFY(cfont.CreateFontIndirect(&clogfont));
                                        //使用初始化后的字体
CClientDC dc(this);
CFont* pold_font = dc.SelectObject(&cfont);
dc.TextOut(5, 5, _T("Hello , Visual C++ 2010"), 5);
dc.SelectObject(pold_font);
                                        //删除初始化后的字体
cfont.DeleteObject();
```

3. CFont::CreatePointFont()

该初始化函数的声明如下：

```
BOOL CreatePointFont(int nPointSize , LPCTSTR lpszFaceName , CDC* pDC = Null);
```

该函数的返回值是 BOOL 类型的，如果成功，则返回 True，否则返回 False。

参数 nPointSize 设置所需字体的高度，单位是 0.1 个像素点，例如该参数若设置为 120，则表示该字体对象的高度是 12 个像素点。

参数 lpszFaceName 参数是一个 CString 类型或指向一个以空终止符结尾的字符串，用来设置所需字体的名称。例如可以将该参数设置为“宋体”、“楷体”等。此字符串长度不应超过 30 个字符。使用 Windows 提供的 EnumFontFamilies()函数可以计算当前可用字体的总数。如果该参数设为 NULL，则 GDI 使用与设备无关的字体。

若想了解 Visual C++ 2010 开发环境所支持的字体名称，可以选择“Tools”→“Options”命令，如图 8.1 所示。

弹出“Options”对话框，选择左侧树形目录中的“Environment”→“Fonts and Colors”选项，在“Font”下拉列表框中就可以看到所有可用的字体名称，如图 8.2 所示。

参数 pDC 指向 CDC 对象的指针，将 nPointSize 中指定的高度值转化为逻辑单位。如果为 NULL，则通过屏幕设备描述表来完成转换。

说明：

此函数提供了一种创建用户指定字体和点大小的字体对象的方法，它自动将 nPointSize 中的高度转化为逻辑单位（通过 pDC 指向的 CDC 对象实现）。此函数是常用的初始化字体对象的方法。同样，当完成用 CreatePointFont()函数创建 CFont()对象后，从设备上下文中选择字体，使用完成后删除 CFont 对象。

下面的代码展示了怎样使用 CreatePointFont()函数初始化一个 CFont 类对象：

```
CClientDC dc(this);
                                        //声明一个 CFont 对象
CFont cfont;
                                        //使用新宋体初始化 cfont，并设置高度为 12 个像素点
VERIFY(cfont.CreatePointFont(120, _T("新宋体"), &dc));
```

```
                                                    //使用初始化后的字体
CClientDC dc(this);
CFont* pold_font = dc.SelectObject(&cfont);
dc.TextOut(5, 5, _T("Hello , Visual C++ 2010"), 5);
dc.SelectObject(pold_font);
                                                    //删除初始化后的字体
cfont.DeleteObject();
```

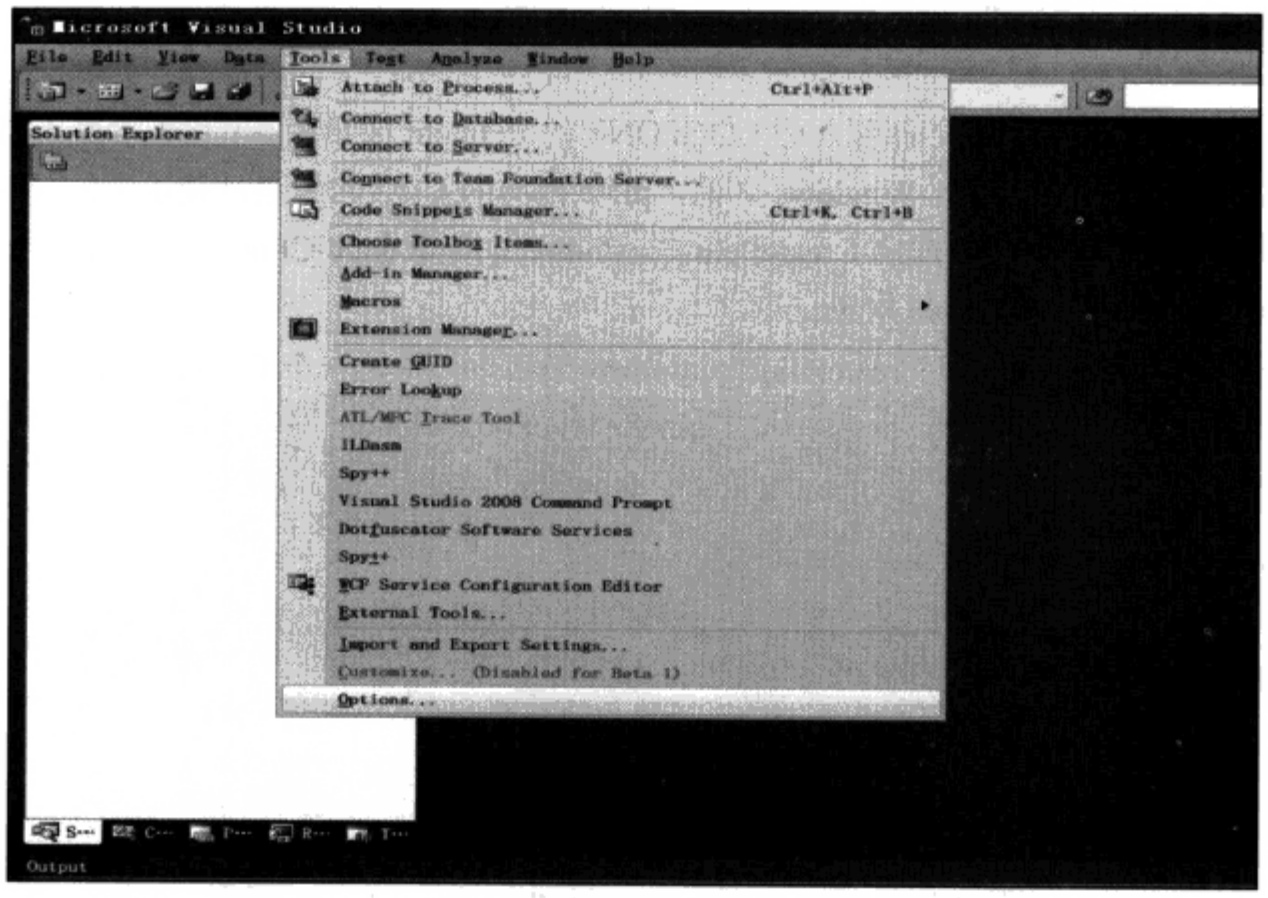


图 8.1　菜单显示图

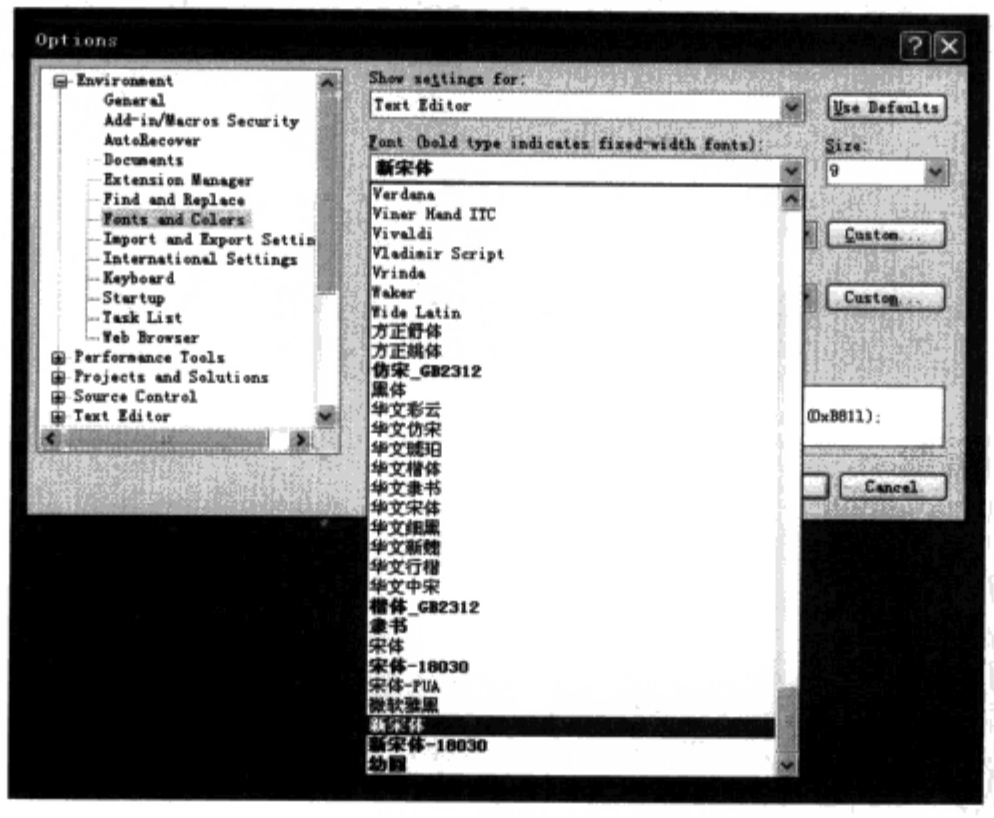


图 8.2　“Fonts and Colors”下拉列表框

4．CFont::CreatePointFontIndirect()

该初始化函数的声明如下：

```
BOOL CreatePointFontIndirect(const LOGFONT* lpLogFont ,CDC* pDC = NULL);
```

该函数返回值是 BOOL 类型的，如果成功，则返回 True，否则返回 False。

参数 lpLogFont 指向 LOGFONT 结构，用于定义逻辑字体特征。关于该参数的介绍，可见上面 CreateFontIndirect()的参数介绍。但是注意，该函数中 LOGFONT 的 lfHeight 成员用 0.1 个像素点计算而不用逻辑单位计算（如果设置 lfHeight 为 120，则要求 12 个像素点字体）。

pDC 指向 CDC 对象的指针，将 nPointSize 中指定的高度值转化为逻辑单位。如果为 NULL，则通过屏幕设备描述表来完成转换。

说明：

此函数与 CreateFontIndirect()相似，但 LOGFONT 的 lfHeight 成员由 0.1 点表示而不是用逻辑单位表示。在传递 LOGFONT 结构到 Windows 之前，pDC 指向的 CDC 对象自动使函数的 lfHeight 中的高度转换为逻辑单位。同样，当完成用 CreatePointFontIndirect()函数创建 CFont 对象后，从设备上下文中选择字体，使用完成后删除 CFont 对象。

下面的代码展示了怎样使用 CreatePointFontIndirect()函数初始化一个 CFont 类对象：

```
                                                    //声明一个 CFont 对象
CFont cfont;
                                                    //声明一个 LOGFONT 结构对象
LOGFONT clogfont;
                                                    //设置 LOGFONT 结构的属性
memset(&clogfont, 0, sizeof(LOGFONT));
                                                    //设置字体高度为 12 个像素点
clogfont.lfHeight = 12;
                                                    //设置字体名为“Arial”
_tcsncpy_s(clogfont.lfFaceName, LF_FACESIZE,
    _T("Arial"), 7);
                                                    //使用上面得到的 LOGFONT 结构对象初始化 cfont
CClientDC dc(this);
VERIFY(cfont.CreatePointFontIndirect(&clogfont,&dc));
                                                    //使用初始化后的字体
CFont* pold_font = dc.SelectObject(&cfont);
dc.TextOut(5, 5, _T("Hello , Visual C++ 2010"), 5);
dc.SelectObject(pold_font);
                                                    //删除初始化后的字体
cfont.DeleteObject();
```

8.1.3 其他成员介绍

1. CFont::FromHandle()

该函数的声明如下：

```
static CFont* PASCAL FromHandle (HFONT hFont);
```

返回值：如果成功，则返回指向 CFont 对象的指针，否则为 NULL。

参数 hFontWindows 指向字体中的 HFONT 句柄。

说明：

当给定一个 Windows GDI 字体对象的 HFONT 句柄时，返回一个指向该 CFont 对象的指针。如果没有 CFont 对象附加给句柄，一个临时 CFont 对象被创建并附加给句柄，该临时 CFont 对象仅当应用

在事件循环中有空闲时才可用，此时临时图形对象被删除，另一种说法是临时对象仅在一个窗口消息进程中才可用。

2．CFont:: GetLogFont()

该函数的声明如下：

```
int GetLogFont(LOGFONT* pLogFont);
```

返回值：如果成功，则返回非 0 值，否则为 0。

该函数的参数 pLogFont 指向 LOGFONT 结构，以接收字体信息。

说明：

调用此函数获取一个 CFont 对象中的 LOGFONT 结构的备份。

3．CFont::operator HFONT()

该函数的声明：

```
operator  HFONT( )  const;
```

返回值：如果成功，则返回附加给 CFont 的 Windows GDI 字体对象句柄，否则为 NULL。

说明：

用此操作获取附加给 CFont 的 Windows GDI 字体对象句柄。由于操作自动将 CFont 转换为字体和文本，因此可以将 CFont 对象传递到除 HFONT 以外的函数。

有关 CFont 字体类的内容就介绍到这里，更多内容大家可以参考 MSDN 帮助手册。下面将开始介绍怎样利用这些创建好的字体对象实现更多的功能。

8.2　创建文本插入符与图片插入符

大家在使用常用的文本处理程序（例如记事本、写字板、Word）时，在编辑窗口中都会有一条闪烁的竖线，可以称其为插入符。插入符的作用就是提示用户输入的文字信息或者图片信息将在屏幕中的哪个位置显示出来。本节将介绍插入符的使用方法和相关知识。

8.2.1　创建文本插入符

如何在程序中创建文本插入符呢？Visual C++ 2010 MFC 提供了 CWnd 类（该类提供了微软基础类库中所有窗口类的基本功能）的 CreateSolidCaret()函数来实现这个功能。

CreateSolidCaret()函数的声明如下：

```
void CreatSolidCaret ( int nWidth , int nHeight );
```

参数 nWidth 指定了文本插入符的宽度（逻辑单位）。如果这个参数为 0，则宽度被设为系统定义的窗口边框宽度。

nHeight 指定了文本插入符的高度（逻辑单位）。如果这个参数为 0，则高度被设为系统定义的窗口边框高度。

说明：

这个函数为系统文本插入符创建一个实心矩形，并声明对文本插入符的所有权。文本插入符的形状可以是线条，也可以是方块。参数 nWidth 和 nHeight 指定了文本插入符的宽度和高度（逻辑单位）；实际的宽度和高度（以像素为单位）依赖于实际应用环境中的映射模式。

为了保证文本插入符在高分辨率显示中依然能够显示清楚，可以参考系统的窗口边框的宽度和高

度（调用 Windows 的 GetSystemMetrics()函数可以获得，具体参见 MSDN 帮助文档）。

CreateSolidCaret()函数会自动销毁原来的文本插入符形状特征。在被创建之后，文本插入符是自动隐藏的，要想显示文本插入符，必须调用 ShowCaret()函数。

下面通过一个简单的例子来实现文本插入符的创建。

首先，使用 Visual C++ 2010 MFC 创建一个单文档类型的 MFC Application 工程，命名为 Text。

单文档类型的 MFC 工程有两个窗口——框架类窗口和视类窗口，根据第 2 章所介绍的内容，视图类窗口始终位于框架类窗口之上，对窗口客户区的鼠标和键盘操作实际上都是在视图类窗口上进行的，所以插入符要创建在视图类窗口中。

文本插入符要在哪个消息的响应函数中创建呢？因为文本插入符应该在窗口创建之后进行，所以它可以在 WM_CREATE 消息的响应函数 OnCreate()中创建窗口的代码之后创建。下面添加该消息的响应函数，并在该函数中创建一个宽度为 10，高度为 50 的插入符，代码如下：

```
// CTextView 消息处理程序
int CTextView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    CreateSolidCaret(10,50);    //创建文本插入符
    ShowCaret();                //显示文本插入符
    // TODO:  Add your specialized creation code here
    return 0;
}
```

注意，一定要使用 ShowCaret()函数，否则插入符是隐藏的，无法显示。运行程序，结果如图 8.3 所示。

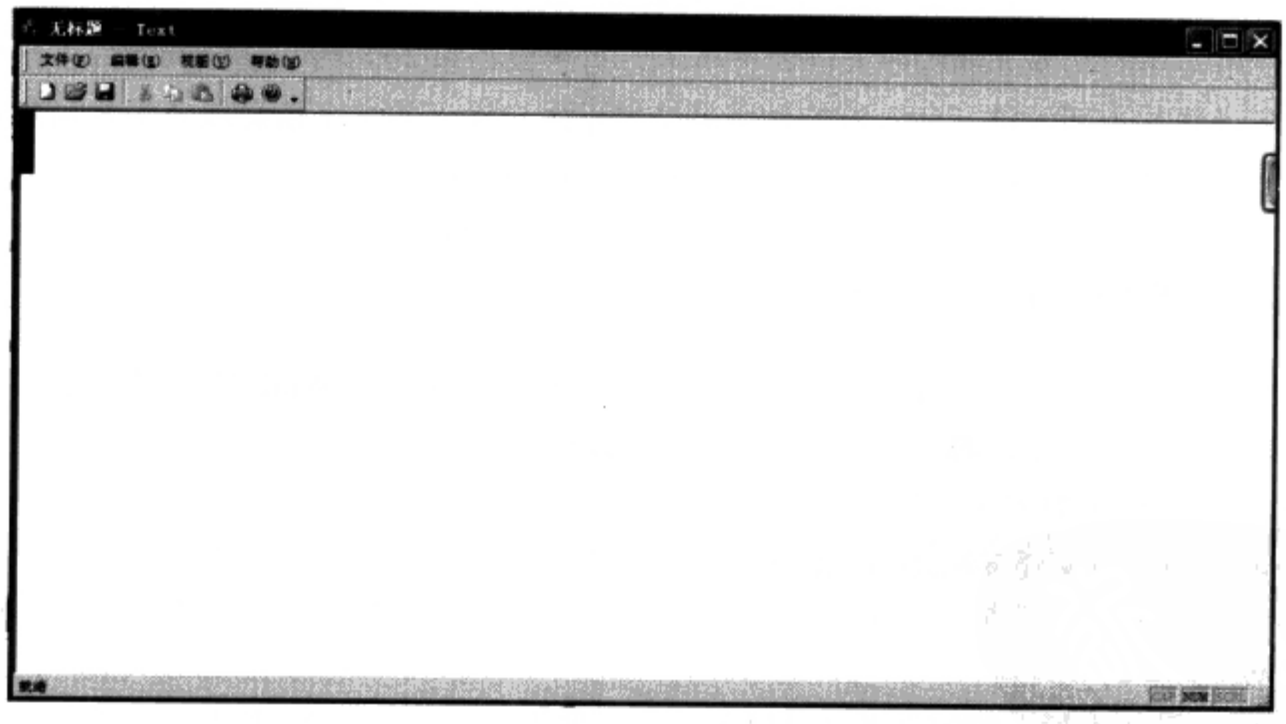


图 8.3　程序运行效果图

由图 8.3 可知，文本插入符被成功创建了，但是这个插入符的大小是固定的，而平常使用的文本处理程序（例如记事本、写字板、Word）等，它们的文本插入符都会根据当前插入字体的大小自动转换大小，怎样才能实现类似功能呢？

首先，需要得到设备描述表中当前字体的信息，然后根据字体的大小信息来调整插入符的大小。可以调用 CDC 类的 GetTextMetrics()成员函数来得到设备描述表中当前字体的度量信息。该函数的声

明如下：

```
BOOL   GetTextMetrics (LPTEXTMETRIC   lpMetric)   const;
```

该函数的参数是一个 TEXTMETRIC 结构体的指针，当调用 GetTextMetrics()函数时，它会用设备描述表中当前字体的信息来填充这个结构体。TEXTMETRIC 结构体包含了物理字体的基本信息，其定义如下：

```
typedef struct tagTEXTMETRIC {
    int   tmHeight;                  //字体的高度
    int   tmAscent;
    int   tmDescent;
    int   tmInternalLeading;
    int   tmExternalLeading;
    int   tmAveCharWidth;            //字体的平均宽度
    int   tmMaxCharWidth;            //字体的最大宽度
    int   tmWeight;
    BYTE tmItalic;
    BYTE tmUnderlined;
    BYTE tmStruckOut;
    BYTE tmFirstChar;
    BYTE tmLastChar;
    BYTE tmDefaultChar;
    BYTE tmBreakChar;
    BYTE tmPitchAndFamily;
    BYTE tmCharSet;
    int   tmOverhang;
    int   tmDigitizedAspectX;
    int   tmDigitizedAspectY;
} TEXTMETRIC;
```

该结构体有众多数据成员，常用的数据成员包括 tmHeight（字体的高度）、tmAveCharWidth（字体中字符的平均宽度）、tmMaxCharWidth（字体中最宽字符的宽度）。也许有读者会问，为什么在这个结构体中要定义平均字符宽度和最大字符宽度，而不定义字符宽度呢？先来简单看两个字母：w 和 i。如果单纯地说取字体的宽度，那么是取 w 的宽度还是 i 的宽度呢？所以该结构只能定义平均字符宽度和最大字符宽度。关于 TEXTMETRIC 结构体的其他一些数据成员的详细说明，请参照 MSDN 帮助文档。

回到文本插入符的创建，可以定义一个 TEXTMETRIC 结构体类型的变量，将该变量的地址传递给 GetTextMetrics()函数，就可以得到当前字体的信息；然后再使用字体的高度和平均宽度来计算插入符的高度和宽度。具体的实现代码如下：

```
// CTextView 消息处理程序
int CTextView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    CClientDC cdc(this);                                    //创建设备描述表
    TEXTMETRIC ltm;                                         //定义 TEXTMETRIC 结构体变量
    cdc.GetTextMetrics(&ltm);                               //获得设备描述表中的文本信息
    CreateSolidCaret(ltm.tmAveCharWidth/8,ltm.tmHeight);    //根据当前字体的大小创建文本插入符
    ShowCaret();                                            //显示插入符
```

```
        // TODO:    Add your specialized creation code here
        return 0;
    }
```

这里将字体宽度除以 8 是一个经验值，读者也可以试用其他数值，看结果是否符合自己的需要。

运行编写好的程序，其结果如图 8.4 所示，在应用程序窗口的左上角成功创建了一个符合当前字体大小的文本插入符。

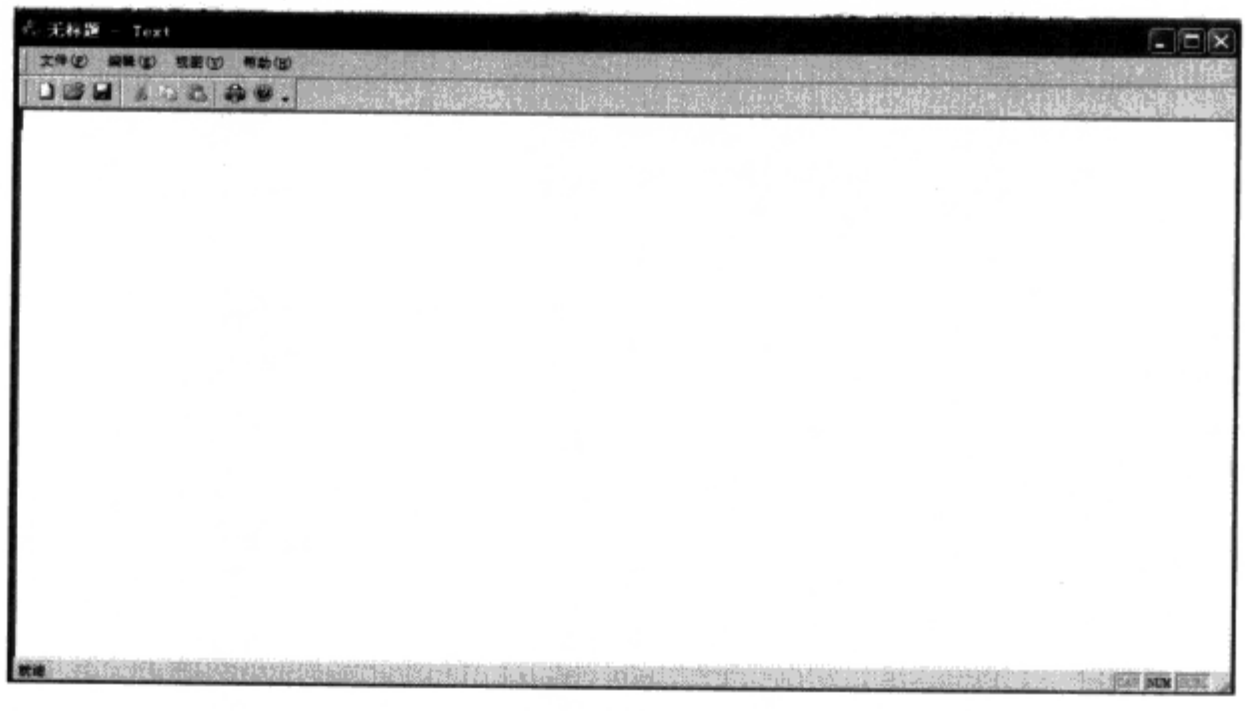


图 8.4　程序效果图

8.2.2　创建图片插入符

刚刚介绍完如何创建文本插入符，再来学习一下另一种插入符——图片插入符的创建。想要创建图片插入符，可以使用 CWnd 类提供的另一个函数——CreateCaret()函数来实现。

CreateCaret()函数的声明如下：

```
void CreatCaret(CBitmap*    pBitmap);
```

参数 pBitmap 标识了一个位图，定义了插入符的形状，这个位图必须是先前用 CreateBitmap()函数、Windows 的 CreateDIBitmap()函数或 LoadBitmap()函数创建的。

该函数用来创建图片插入符，并声明对插入符的所有权。与 CreateSolidCaret()函数一样，CreateCaret()函数会自动销毁原来的文本插入符形状特征。在被创建之后，图片插入符是自动隐藏的，要想将其显示，必须调用 ShowCaret()函数。

下面通过一个简单的例子来实现图片插入符的创建。

因为 CreateCaret()函数要使用一个位图来当作参数，所以首先要获得一个位图对象。开发人员既可以创建一个位图资源，也可以导入一个已有的位图资源。下面学习一下如何导入一个已有的位图资源。

首先，使用 Visual C++ 2010 MFC 创建一个单文档类型的 MFC Application 工程。然后选择该工程 Solution Explorer 中的 Resource View，右击并选择“Add Resource”命令，如图 8.5 所示。

然后，在弹出的对话框中选择“Bitmap”选项，后单击“Import”按钮，如图 8.6 所示。

接着，在弹出的“Import”对话框中选择需要的位图图片，这里选择 CBitmap.bmp，然后单击“打开”按钮，即可实现已有位图的导入，如图 8.7 所示。

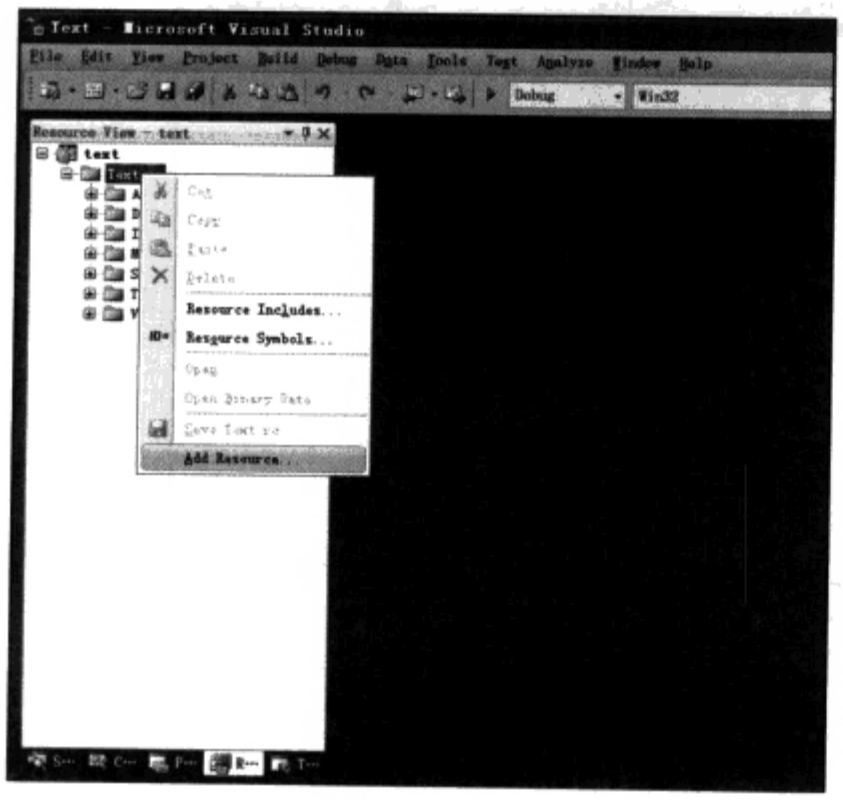


图 8.5　资源创建

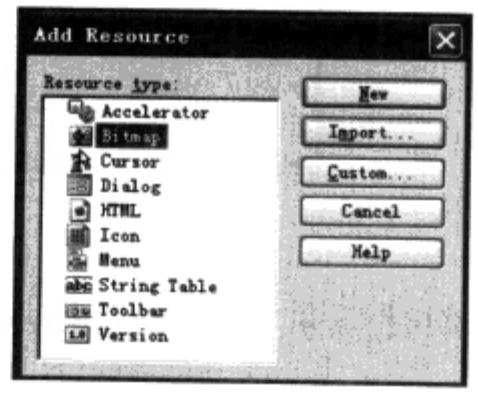


图 8.6　选择类型

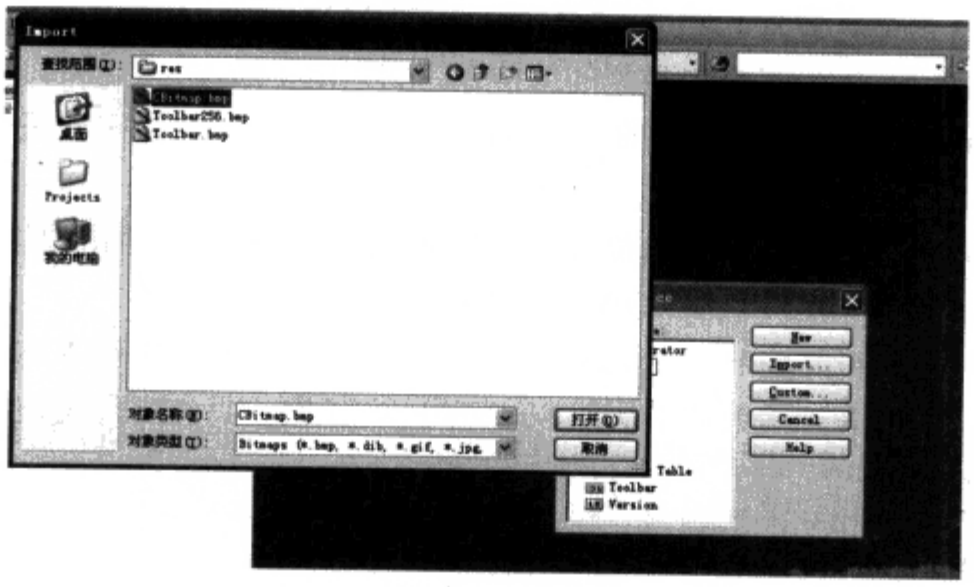


图 8.7　导入资源

创建图片插入符的具体代码如下：

```
// CTextView 消息处理程序
int CTextView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    bitmap.LoadBitmap(IDB_BITMAP1);   //装载位图资源
    CreateCaret(&bitmap);             //创建一个图片插入符
    ShowCaret();                      //显示插入符
    // TODO:  Add your specialized creation code here
    return 0;
}
```

运行程序后，并不能看到图片插入符，这是因为所使用的位图对象是在 OnCreate()函数中创建的，当该函数执行完毕之后，里边的所有对象都要被析构，因此需要将位图对象设置为全局变量。将位图对象设置为全局变量后，其运行结果如图 8.8 所示。

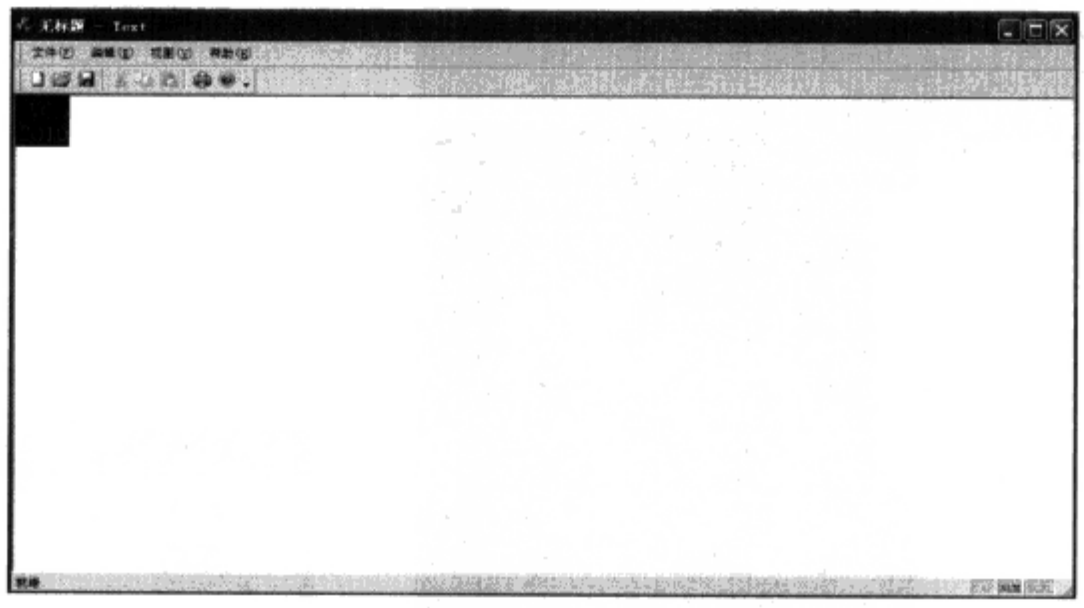


图 8.8　程序运行效果图

8.2.3　创建随鼠标移动的插入符

之前创建的插入符，都只能显示在程序窗口的左上角。怎样实现用鼠标单击窗口的某个位置，插入符就移动到这个位置呢？Cwnd 类提供的 SetCaretPos()函数可以实现这个功能。该函数的声明如下：

```
static void PASCAL SetCaretPos ( POINT point);
```

该函数的参数 point 是一个 POINT 类型的结构体，它指定了插入符的新的 x 坐标和 y 坐标（客户坐标），该函数仅当插入符属于当前任务的一个窗口时才移动它。不论插字符是否隐藏，SetCaretPos()都移动插入符的位置。

下一步就需要知道插入符插入的新的位置。由于是移动到鼠标左键单击的位置，所以可以通过捕获鼠标左键按下消息（WM_LBUTTONDOWN）来获得这个位置。

下面，在上节图片插入符的基础上实现这个功能。首先添加 WM_LBUTTONDOWN 的消息响应函数 OnLButtonDown()，然后在该函数体中加入 SetCaretPos()函数，具体实现代码如下：

```
void CTextView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    SetCaretPos(point);
    CView::OnLButtonDown(nFlags, point);
}
```

运行程序，然后使用鼠标左键单击窗口任意位置，将发现图片插入符会随着鼠标左键单击的位置而移动，程序运行结果如图 8.9 所示。

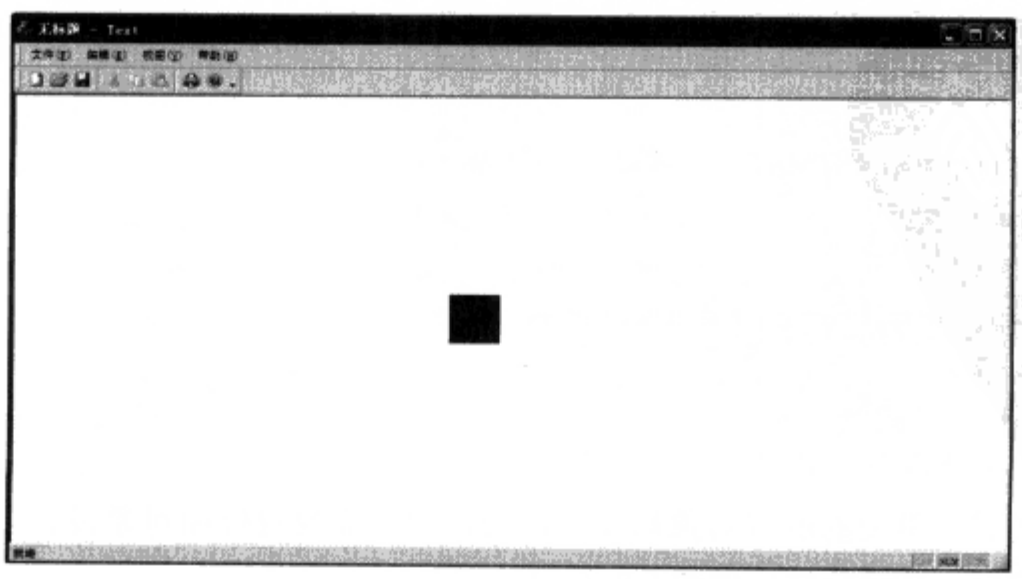


图 8.9　程序效果图

8.3　输出文字与字体格式

8.3.1　输出固定文字

怎样将文字输出到窗口中呢？在介绍 CFont 字体类的时候，了解了 C D C 类提供的 TextOut()函数，这个函数即可用来实现文字在窗口上的输出。

TextOut()函数的声明如下：

```
BOOL TextOut ( int x , int y ,const CString& str );
```

该函数的参数 x 用来指定文本起点的 X 逻辑坐标，y 用来指定文本起点的 Y 逻辑坐标，CString 对象 str 存储需要显示的文字内容。

该函数的作用就是将参数 str 中的内容使用设备描述表中指定的字体在窗口中坐标为（x,y）的位置显示出来。下面简单介绍如何使用这个函数来实现文字输出功能。

首先，还是要确定在哪个消息的响应函数中实现文字输出。这里，先简单介绍一下窗口重绘的概念。在 Windows 程序运行时，在程序的窗口大小发生变化或者被其他窗口覆盖又重新显示后，窗口就会发生重绘，窗口中已经输入的内容就会被擦掉。那么如何使窗口发生重绘后窗口中的内容不发生改变呢？一种方法是在响应 WM_PAINT 消息的函数中将内容再次输出，另一种方法是使用 MFC 中提供的一个类似于 WM_PAINT 消息响应函数的 OnDraw()函数，当窗口发生重绘时，应用程序就会调用该函数。因此，若想使输出的字体在窗口重绘后依然显示在窗口中，可以将文字输出放在 OnDraw()函数中实现。

使用 Visual C++ 2010 MFC 创建一个单文档类型的 MFC Application 工程，在该工程的视图类窗口类中的 OnDraw()函数中添加如下代码：

```
void CtextOutView::OnDraw(CDC* pDC)
{
    CtextOutDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;
    CString str;                                    //创建一个 CString 类型的字符串
    str="Visual C++ 2010   MFC   文本与字体";  //初始化 str 字符串
    pDC->TextOut(100,100,str);                      //输出该字符串
    pDC->SelectObject(pold_font);                   //还原设备描述表中的字体对象
    cfont.DeleteObject();                           // 删除字体
                                                    // TODO: 在此处为本机数据添加绘制代码
}
```

从这个 OnDraw()函数的定义可知，当它被调用时，应用程序框架会构造一个 CDC 类对象的指针并将其传递给这个函数。这样，就可以直接使用传递进来的 CDC 指针去调用 TextOut()函数。接下来，声明一个 CString 类型的字符串对象，并用想要输出在窗口中的文字初始化该对象；然后调用 TextOut()函数，即可完成固定文字的输出。

该程序的运行结果如图 8.10 所示。

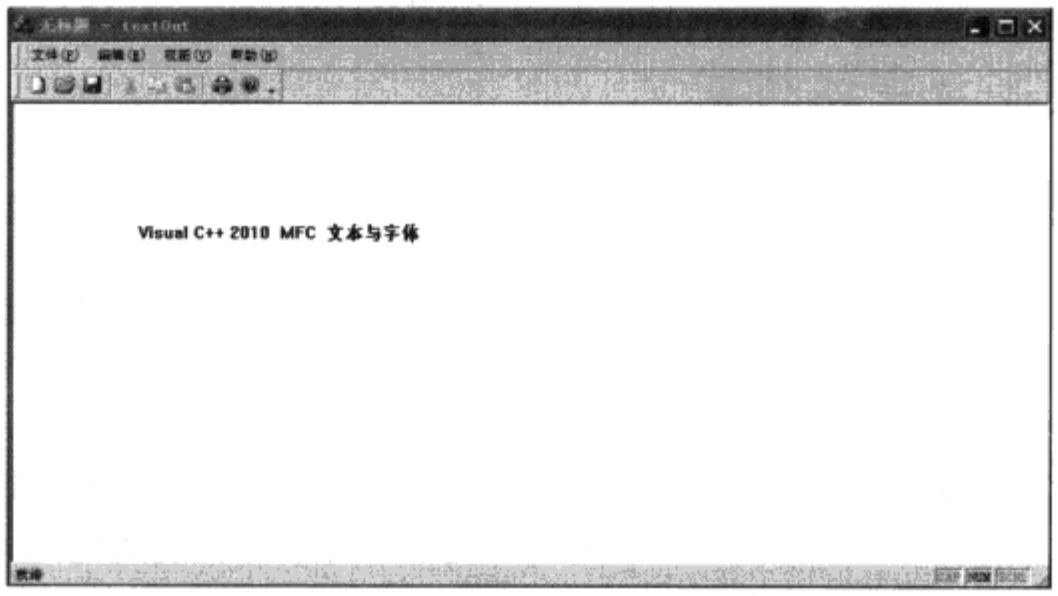


图 8.10　程序效果图

8.3.2　设定输出字体的格式

上节实现了文字的输出，但是输出的字体格式是设备描述表指定的格式，那么怎样来实现输出用户指定格式的字体呢？这里就要用到 9.1 节所介绍的 CFont 字体类的相关知识了。

首先，创建一个 Font 类的对象，这里使用 CreatPointFont()函数初始化该对象。假设想要输出一个华文彩云格式的文字，字体的大小为 16 个点。具体的初始化方法请参考 9.1.2 小节。

由于应用程序输出文字到当前窗口使用的是设备描述表中默认的字体格式（关于设备描述表的概念，将在下一章详细介绍），因此要想用自己指定的格式来输出文字，必须用初始化后的字体对象来替换当前设备描述表中的字体对象。具体的替换通过使用 SelectObject()函数来实现，该函数的返回值是指向先前被选对象的指针，其目的是完成当前字体输出操作之后，还原设备描述表。在完成绘图之后，利用 SelectObject()函数把先前的字体对象选入设备描述表使其恢复到原始的状态。最后，还要删除开发人员自己创建的字体对象。具体实现代码如下：

```
void CtextOutView::OnDraw(CDC* pDC)
{
    CtextOutDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;
                                                    //声明一个 CFont 对象
    CFont cfont;
                                                    //使用华文彩云初始化 cfont,并设高度为 16 个像素点
    cfont.CreatePointFont(160,_T("华文彩云"), NULL);
                                                    //使用初始化后的字体
    CFont* pold_font = pDC->SelectObject(&cfont);
    CString str;                                    //创建一个 CString 类型的字符串
    str="Visual C++ 2010   MFC   文本与字体"; //初始化 str 字符串
    pDC->TextOut(100,100,str);                      //输出该字符串
    pDC->SelectObject(pold_font);                   //还原设备描述表中的字体对象
    cfont.DeleteObject();                           //删除字体
                                                    // TODO: 在此处为本机数据添加绘制代码
}
```

程序的运行结果如图 8.11 所示。

对于其他的字体类初始化函数，其使用过程都是相同的。读者可以自己依次尝试一下，看能否输出自己想要的字体类型。

在这里，只说明了如何改变字体的格式信息，为了实现其他复杂的字体处理功能，MFC 还提供了 CEditView 和 CRichEditView 两个类。

CEditView 类是一个视图类，它包含 Windows 标准编辑控件（CEdit）的视图。编辑控件支持文字编辑、查找、替换和滚动等功能，可以用来执行简单文本编辑操作等。因为 CEditView 类是 CView 类的派生，CEditView 的对象可以和文档及文档模板同时使用。每个 CEditView 控件的文本存放到其全局存储对象，应用程序能拥有任意多个 CEditView 对象。

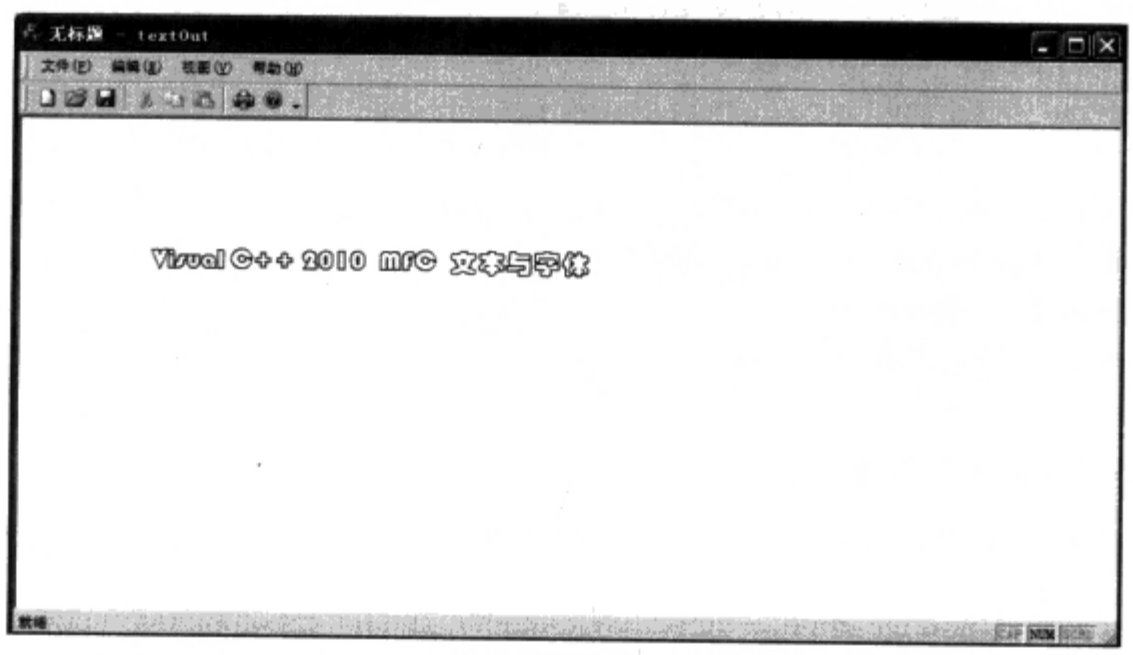


图 8.11　程序效果图

CRichEditView 类与 CEditView 类的区别是，CRichEditView 类包含 Windows 丰富编辑控件（CRichEditctrl）的视图。除了编辑控件的功能外，丰富编辑控件还支持字体、颜色、段落格式和嵌入的 OLE 对像的管理等功能。

如果读者以后要实现字处理程序，让程序的视图类直接派生于这两个类中的一个即可，这样，就不用自己再去实现一些简单的编辑功能。对于这两个类的详细使用说明，请大家参考 MSDN 帮助文档。

8.3.3　字符输入

上面两个小节实现的都是固定的文字信息的输出，而一个真正的文字处理程序，必然要输出用户输入的任意可输出字符，并对其他字符做出响应。那么怎样才能捕获用户的输入并实现其在屏幕上的显示呢？本小节将逐步实现这一功能。

1．输入文字的捕获

首先，要实现的就是捕获用户在键盘上按下的某个字符按键，这里将通过使用前面介绍的 WM_CHAR 消息来实现。可以先捕获这个消息，然后在该消息的响应函数中完成字符输出功能。

但是，这里有一个问题需要注意。利用 TextOut()函数输出字符时，需要提供字符显示位置的 x 和 y 坐标。例如，要输出单词“with”，那么将在（0,0）位置输出第一个字符 w，但是怎样才能确定第二个字符“i”或者第三个字符“t”的位置呢？正如在 8.2.1 小节介绍的那样，因为每个英文字符的宽度是不一样的，所以 C++只提供了获得当前字体平均字符宽度和最大字符宽度的方法，而不能得到每个字符的宽度，这样就无法获得“with”中“w”之后字符的输出位置了。但是，可以使用一个简单的方法绕过这个问题，即可以将每次输入的字符都存储到一个字符串中。例如，输入字符“w”和“i”后，

就将这两个字符组成一个字符串“wi”。如果之后用户再输入 t，则再将其与“wi”组成一个新的字符串“wit”。每当用户输入新的字符时，都在窗口插入符的位置把这个字符重新输出一次。利用人眼所具有的视觉残留的特点，用户就感觉好像每次只是输入了一个字符，而其他都没有改变。

按照如上的思路，首先需要在 CTextView 类中定义一个 CString 类型的成员变量 strLine，并在该类的初始化函数中将其初始化为空；然后再在该类中插入一个捕获 WM_CHAR 消息的函数；最后，在该函数中，使用 strLine 来存储用户输入的字符串。

定义 CString 类型的成员变量 strLine：

```
private:
    CString strLine;//创建保存用户输入字符的字符串
```

使用 strLine 来存储用户输入的字符串。

```
void CtextOutView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default
        strLine=strLine.Left(strLine.GetLength()-1);
        strLine=strLine+(char)nChar;
    CView::OnChar(nChar, nRepCnt, nFlags);
}
```

2．实现字符串在插入符位置的输出

首先，要使用 8.2.3 小节中介绍的内容加入一个可以随鼠标移动的文字插入符，用来确定文字输出的位置。其实现代码如下：

```
int CtextOutView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    // TODO:  Add your specialized creation code here
    CClientDC cdc(this);                                    //创建设备描述表
    TEXTMETRIC ltm;                                         //定义 TEXTMETRIC 结构体变量
    cdc.GetTextMetrics(&ltm);                               //获得设备描述表中的文本信息
    CreateSolidCaret(ltm.tmAveCharWidth/8,ltm.tmHeight);    //根据当前字体的大小创建文本插入符
    ShowCaret();                                            //显示插入符
    return 0;
}
void CtextOutView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    SetCaretPos(point);
    strLine.Empty();
    cpoint=point;
    CView::OnLButtonDown(nFlags, point);
}
```

注意代码中的阴影部分，该代码的作用是在每次鼠标改变插入符的位置时清空 strLine 的内容，这是因为每次改变插入符的位置时，就需要重新开始输入文字，而不应该把在其他位置输入的文字再带到这里来。

接着，实现文字的输入，这里还是使用 8.3.1 小节介绍过的 TextOut()函数来输出文字。因为 TextOut()

函数需要指定文字输出的坐标，所以需要给 CTextView 类添加一个 CPoint 类型的成员变量 point，用来存储当前插入符的位置，也就是文字输出的位置；然后在处理鼠标按下位置的函数中给该变量赋值。具体代码如下：

```
void CtextOutView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    SetCaretPos(point);
    strLine.Empty();
    cpoint=point;
    CView::OnLButtonDown(nFlags, point);
}
```

这样，就可以在 OnChar()函数中实现文字的输出了。其实现代码如下：

```
void CtextOutView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default
    strLine=strLine+(char)nChar;
    CClientDC cdc(this);
    cdc.TextOut(cpoint.x,cpoint.y,strLine);
    CView::OnChar(nChar, nRepCnt, nFlags);
}
```

运行程序，其结果如图 8.12 所示。

细心的读者可以发现，运行结果中文字插入符并没有如通常人们所使用的文字处理程序一样随着文字的输入向后移动，而是一直处于最初单击鼠标左键的位置。那么如何来实现这个功能呢？在 8.2.3 小节学习了如何使用 SetCaretPos()函数来设置插入符的位置，但是如何确定到底应该将插入符设置在哪里呢？因为这里是用一个字符串保存了整个要输出的文字，所以只要能获得字符串的宽度，就可以知道插入符到底要向后移动多少了。而字符串的宽度可以简单通过前面介绍的 GetTextExtent()函数得到。修改后的代码如下所示：

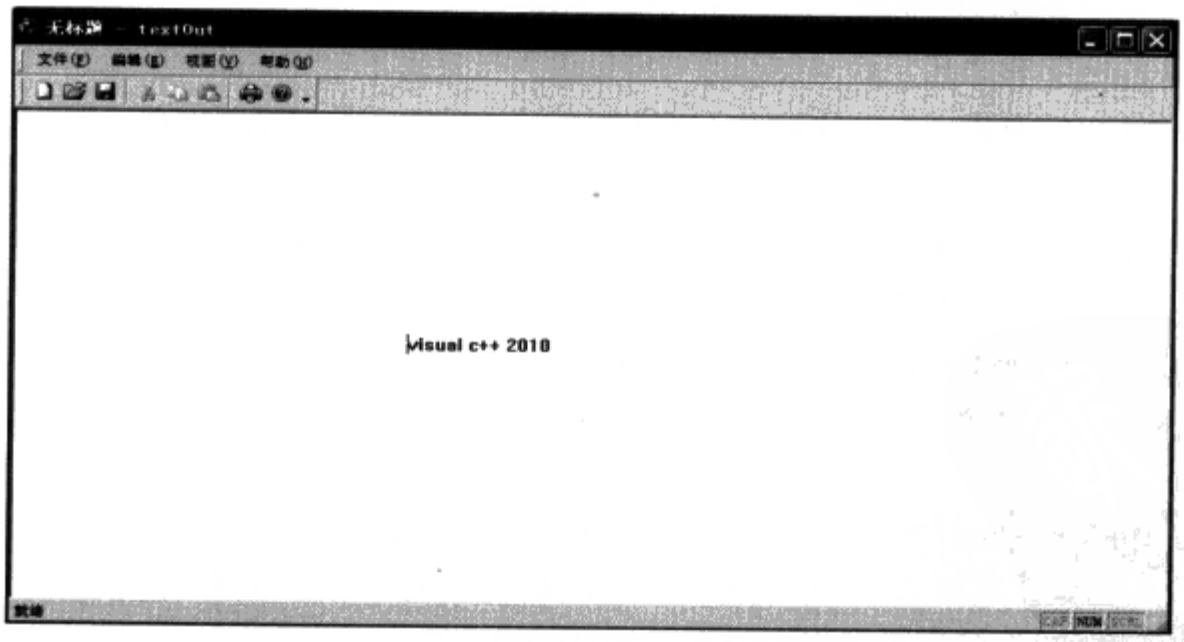


图 8.12　程序效果图

```
void CtextOutView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default
strLine=strLine+(char)nChar;
```

```
    CClientDC cdc(this);
        CSize size=cdc.GetTextExtent(strLine);              //获得字符串的宽度
        CPoint carect_point;
        carect_point.x=cpoint.x+size.cx;                    //将插入符插入位置的 x 坐标右移字符串宽度的距离
        carect_point.y=cpoint.y;                            //插入符插入位置的 y 坐标不变
        SetCaretPos(carect_point);
        cdc.TextOut(cpoint.x,cpoint.y,strLine);
        CView::OnChar(nChar, nRepCnt, nFlags);
    }
```

再次运行程序，输入字符，发现可以成功地使插入符随着文字的输入而向后移动了。程序运行结果如图 8.13 所示。

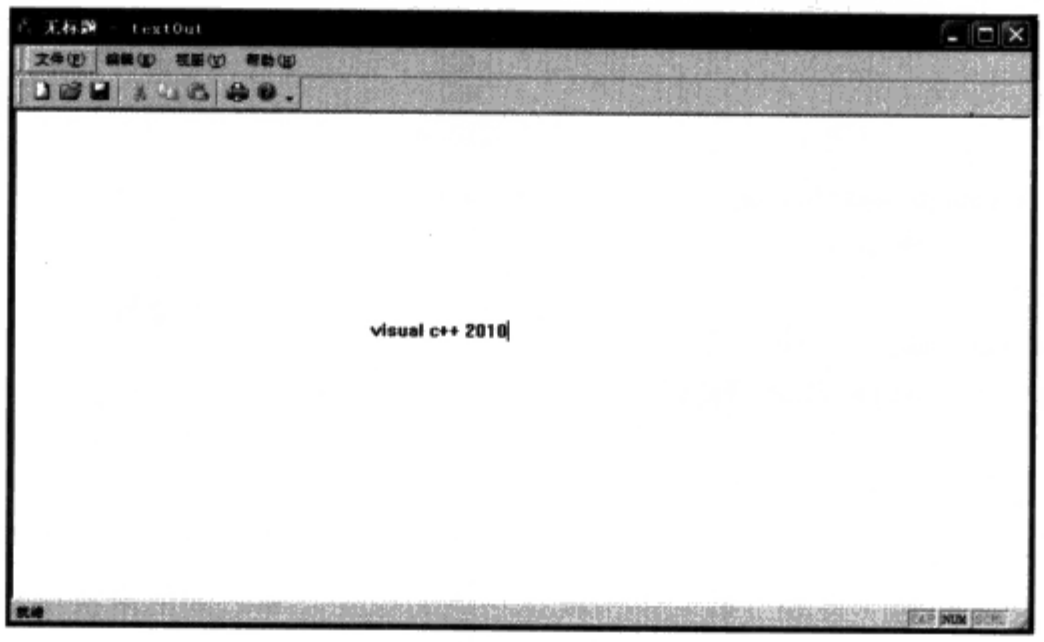


图 8.13　程序效果图

3．回车键和退格键的实现

在实现文字输入的时候，有一些字符需要特别考虑，比如回车键（Enter）和退格键（Backspace）。下面就介绍一下如何处理这些字符。

当用户按【Enter】键之后，希望得到的响应是插入符换到下一行，然后新输入的文字应该在新的插入符位置开始输出的屏幕上。也就是说，捕获回车键之后，需要实现下面两种操作：一是使插入符的横坐标变为 0，纵坐标变为下一行坐标，即利用已经保存的当前插入点的纵坐标加上当前字体的高度就可以得到回车后的新的插入符的纵坐标；二是要把上一行保存的字符清空，即把字符串 strLine 中的内容清空。明白了这两个需求，就可以处理回车键了。回车键 ASCII 码的十六进制值是 0x0d。插入符的移动可以使用 SetCaretPos()函数和 GetTextMetrics()函数（获得当前设备描述表中的字体高度信息，具体介绍请见 8.3.1 小节）配合来实现。而字符串的清空可以使用 CString 类提供的 Empty()函数实现，其具体实现代码如下：

```
void CtextOutView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default
    CClientDC cdc(this);
    TEXTMETRIC tm;
    cdc.GetTextMetrics(&tm);
    if(0x0d==nChar)                          //处理回车键
    {
```

```
        strLine.Empty();                    //清空字符串
        cpoint.y=cpoint.y+tm.tmHeight;      //移动插入符
    }
    else
    {
        strLine=strLine+(char)nChar;
    }
    CSize size=cdc.GetTextExtent(strLine); //获得字符串的宽度
    CPoint carect_point;
    carect_point.x=cpoint.x+size.cx;        //将插入符插入位置的 x 坐标右移字符串宽度的距离
    carect_point.y=cpoint.y;                //插入符插入位置的 y 坐标不变
    SetCaretPos(carect_point);
    cdc.TextOut(cpoint.x,cpoint.y,strLine);
    CView::OnChar(nChar, nRepCnt, nFlags);
}
```

用户按退格键后，又应该怎么处理呢？当用户按退格键之后，希望得到的响应是删除屏幕上位于插入符前面的那个字符，同时，插入符应该回退一个字符的位置。但是这样实现起来很难，所以可以换一种方法，将原有的字符串以屏幕背景色输出，将原来屏幕上的一行字覆盖，由于覆盖在上面的字符串颜色和屏幕的颜色相同，因此看起来整行字不见了，然后将字符串的最后一个字符删除，再将原来设备描述表中的字体颜色取回来，使用这个颜色输出字符串，由于人们的视觉暂留特点，感觉就好像是删掉了一个字符。

下面就根据以上的原理来实现退格键的功能，其代码如下：

```
void CtextOutView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default
    CClientDC cdc(this);
    TEXTMETRIC tm;
    cdc.GetTextMetrics(&tm);
    if(0x0d==nChar)                         //处理回车键
    {
        strLine.Empty();                    //清空字符串
        cpoint.y=cpoint.y+tm.tmHeight;      //移动插入符
    }
    else if(0x08==nChar)
    {
                                            //将文字颜色设置为背景色，并将原来的颜色保留到 oldClr 中
        COLORREF oldClr;
        oldClr=cdc.SetTextColor(cdc.GetBkColor());
                                            //用背景色将字符串再输出一遍，相当于将字符串擦去
        cdc.TextOut(cpoint.x,cpoint.y,strLine);
                                            //将文字颜色设置为正常颜色
        cdc.SetTextColor(oldClr);
                                            //将字符串的最后一个字符去掉
        strLine=strLine.Left(strLine.GetLength()-1);
    }
    else
    {
```

```
            strLine=strLine+(char)nChar;
        }
        CSize size=cdc.GetTextExtent(strLine);          //获得字符串的宽度
        CPoint carect_point;
        carect_point.x=cpoint.x+size.cx;                //将插入符插入位置的 x 坐标右移字符串宽度的距离
        carect_point.y=cpoint.y;                        //插入符插入位置的 y 坐标不变
        SetCaretPos(carect_point);
        cdc.TextOut(cpoint.x,cpoint.y,strLine);
        CView::OnChar(nChar, nRepCnt, nFlags);
    }
```

退格键 ASCII 码的十六进制值是 0x08。代码中的 COLORREF 类型用于保存 RGB 颜色值，函数 GetBkColor()用于获取屏幕的背景色，SetTextColor()函数用于设置输出文本的颜色值，该函数原型如下：

```
virtual COLORREF SetTextColor( COLORREF crColor );
```

该函数只有一个参数——crColor，即用户希望文本输出的颜色，该函数执行成功之后会返回一个 COLORREF 类型的颜色值，返回的颜色值就是先前文本的正常颜色，因此，之前定义了一个 COLORREF oldClr 来保存当设备描述表选取新的文本颜色时返回的原来的文本颜色。

然后，用背景色将字符串输出一遍，相当于将字符串擦去，输出之后，再将文字颜色设置为先前的正常色。

最后，将字符串的最后一个字符去掉，这里使用了 CString 的 Left()函数来实现，再在屏幕上重新输出更改后的文字。

读者可以自行测试编写好的程序，看是否完成了对回车键和退格键的响应。至此，一个基本的文字输出程序就完成了，读者可以在这个基础再实现一些其他的功能，使这个文字输出程序更加完善。

8.4 输出彩色文字与变色文字

在 8.3.3 小节的退格键的实现中，介绍了一种改变输出文字颜色的方法，即使用 SetTextColor()函数来改变文字的颜色。本节将介绍另外一种改变文字颜色的方法。

相信读者对“卡拉 OK”肯定不会陌生，随着音乐开始，电视屏幕上的文字就会根据音乐的旋律和节奏逐渐地将字的颜色改变成另一种颜色，以便演唱者掌握节奏。下面将通过介绍如何实现该功能，来介绍怎样输出彩色文字和变色文字。

8.4.1 DrawText()函数和字符串资源

1. DrawText()函数

为了实现对屏幕上的一段文字逐个进行颜色改变，首先介绍另外一个文字输出函数——DrawText()函数。与 TextOut()函数不同的是，该函数用于在一个矩形范围内输出文本，如果文本长度大于矩形宽度，超出的部分将被裁剪掉。

DrawText()函数的声明如下：

```
int DrawText ( const CString& str , LPRECT lpRect , UNIT nFormat);
```

该函数如果调用成功，则其返回值是文本高度。

参数 str 含有要被绘制的文本的 CString 对象。

参数 lpRect 指向 RECT 结构或 CRect 对象的指针，结构（或对象）中包含矩形（逻辑单位表示），

该矩形即是文字显示的范围。

参数 nFormat 指定文字的输出格式，其可取值请参见 MSDN 帮助文档。

下面介绍一下如何使用该函数，代码如下：

```
void CTextOutView::OnDraw(CDC* pDC)
{
    CTextOutDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
                                        // TODO: 在此处为本机数据添加绘制代码
                                        //定义并初始化字符串
    CString str;
    Str="Visual C++ 2010 程序设计入门到精通";
                                        //定义一个 rect 类型的变量
    CRect rect;
                                        //设置 rect 的左上角坐标
    rect.top=100;
    rect.left=0;
                                        //设置 rect 的右下角坐标
    CSize sz=pDC->GetTextExtent(str);
    rect.bottom=100+sz.cy;
    rect.right=100;
                                        //输出字体
    COLORREF oldClr;
    oldClr=pDC->SetTextColor(RGB(255,0,0));
    pDC->DrawText(str,&rect,DT_LEFT);
    Pdc->SetTextColor(oldClr);
}
```

这段代码首先定义了一个 CString 类型的字符串，然后定义了一个 CRect 对象 rect，设置其 top 和 left 分别为 50 和 0（即左上角坐标为（0,50）），右下角 bottom 设置为文本输出的左上角的 y 值加上文本的高度，right 设置为 100；最后使用 DrawText()输出文字，设置其文字输出格式为左对齐（DT_LEFT），其运行结果如图 8.14 所示。

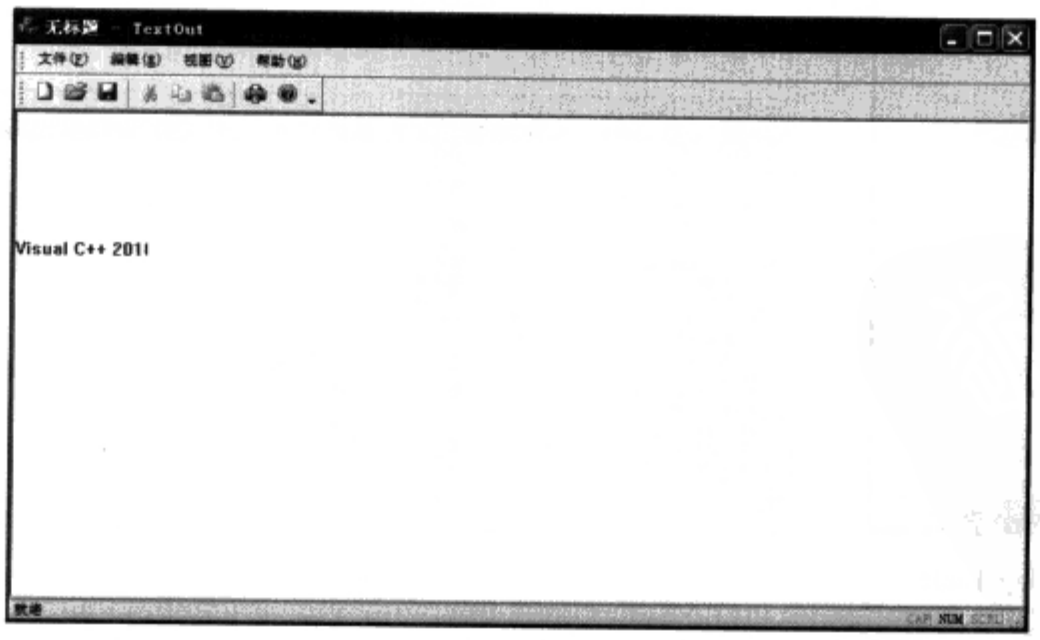


图 8.14　程序效果图

由运行结果可见，定义的字符串“Visual C++ 2010 程序设计入门到精通”并没有完全显示出来，这是因为设置的矩形范围的长度要比输出字体的长度小，超出的文字部分被自动裁剪掉了。如果把 rect

对象的 right 设置为 200，则字符串就完全输出了，如图 8.15 所示。

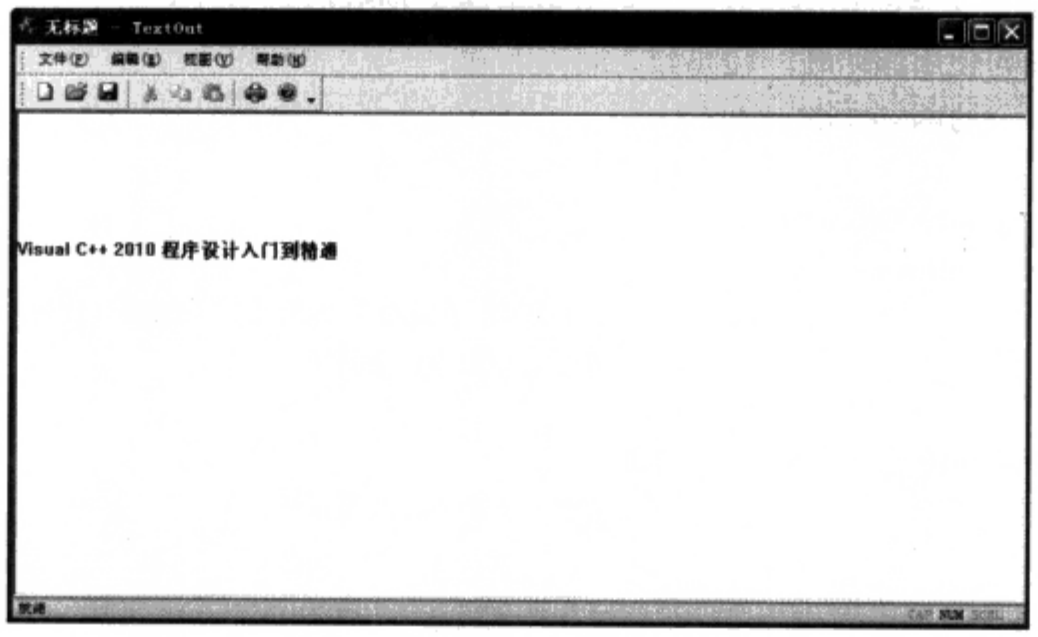


图 8.15　程序效果图

2．字符串资源

下面再来介绍一下怎样在工程中添加字符串资源。在 CString 类中，提供了一个成员函数 LoadString()，该函数可以装载一个由 nID 标识的字符串资源。LoadString()函数声明如下：

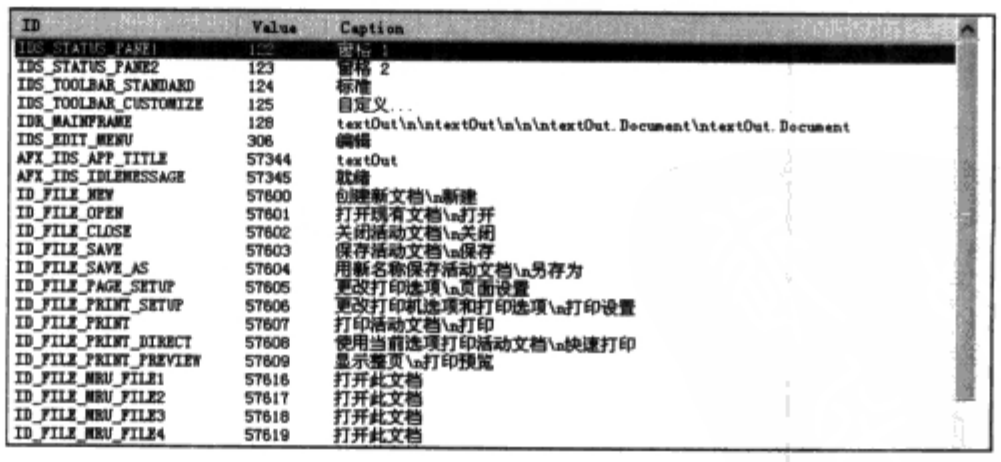
```
BOOL LoadString( UINT nID );
```

参数 nID 是在资源面板中的 String Table 中定义的字符串资源的 ID。使用字符串资源的优点是，可以不修改代码而改变字符的内容。

首先，还是新建一个单文档视图结构的 MFC 应用程序，工程名为 TextOut；然后，单击工作台的“Resource View”标签，使工作台窗口显示到资源面板页面。展开最顶层的 textOut.rc 选项之后，会看到一个“String Table”（字符串表）文件夹。展开“String Table”文件夹会看到该文件夹下有一个和文件夹同名的 String Table 表，如图 8.16 所示。

双击图 8.16 所示的 String Table 表，在右侧的编辑窗口中会显示一个字符串资源表，如图 8.17 所示。

图 8.16　String Table



ID	Value	Caption
IDS_STATUS_PANE1	122	窗格 1
IDS_STATUS_PANE2	123	窗格 2
IDS_TOOLBAR_STANDARD	124	标准
IDS_TOOLBAR_CUSTOMIZE	125	自定义...
IDR_MAINFRAME	128	textOut\n\ntextOut\n\n\ntextOut.Document\ntextOut.Document
IDS_EDIT_MENU	306	编辑
AFX_IDS_APP_TITLE	57344	textOut
AFX_IDS_IDLEMESSAGE	57345	就绪
ID_FILE_NEW	57600	创建新文档\n新建
ID_FILE_OPEN	57601	打开现有文档\n打开
ID_FILE_CLOSE	57602	关闭活动文档\n关闭
ID_FILE_SAVE	57603	保存活动文档\n保存
ID_FILE_SAVE_AS	57604	用新名称保存活动文档\n另存为
ID_FILE_PAGE_SETUP	57605	更改打印选项\n页面设置
ID_FILE_PRINT_SETUP	57606	更改打印机选项和打印选项\n打印设置
ID_FILE_PRINT	57607	打印活动文档\n打印
ID_FILE_PRINT_DIRECT	57608	使用当前选项打印活动文档\n快速打印
ID_FILE_PRINT_PREVIEW	57609	显示整页\n打印预览
ID_FILE_MRU_FILE1	57616	打开此文档
ID_FILE_MRU_FILE2	57617	打开此文档
ID_FILE_MRU_FILE3	57618	打开此文档
ID_FILE_MRU_FILE4	57619	打开此文档

图 8.17　字符串资源表

将该表格的滚动条拉到最底端，发现有一个空白表项，单击该表项，即可在右上角添加一个字符串资源，可以给添加的字符串资源定义一个 ID 号并输入窗口上 ID 右侧的文本框中，如 IDS_NEWSTRING；在 Caption 右侧的文本框中输入该字符串的文本，如“Visual C++ 2010 程序设计入门到精通”，如图 8.18 所示。

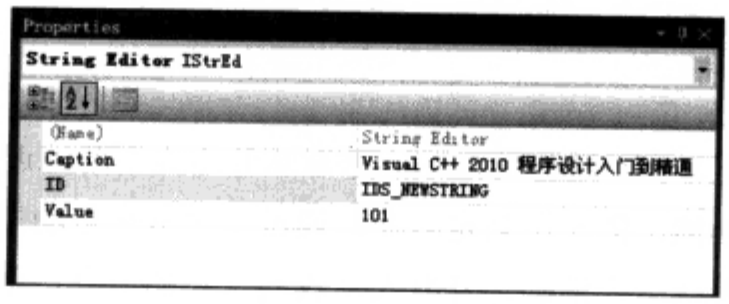


图 8.18　字符属性对话框

这样，就新建好了一个字符串资源，现在可以在字符串资源表中看到它。改变上面 DrawText()函数的代码，看能否使用该字符串资源。代码如下：

```
void CTextOutView::OnDraw(CDC* pDC)
{
    CTextOutDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
                                          // TODO: 在此处为本机数据添加绘制代码
                                          //定义并初始化字符串
    CString str;
    str.LoadString(IDS_NEWSTRING);        //定义一个 rect 类型的变量
    CRect rect;
                                          //设置 rect 的左上角坐标
    rect.top=100;
    rect.left=0;
                                          //设置 rect 的右下角坐标
    CSize sz=pDC->GetTextExtent(str);
    rect.bottom=100+sz.cy;
    rect.right=100;
                                          //输出字体
    COLORREF oldClr;
    oldClr=pDC->SetTextColor(RGB(255,0,0));
    pDC->DrawText(str,&rect,DT_LEFT);
    Pdc->SetTextColor(oldClr);
}
```

该程序的运行结果如图 8.19 所示。

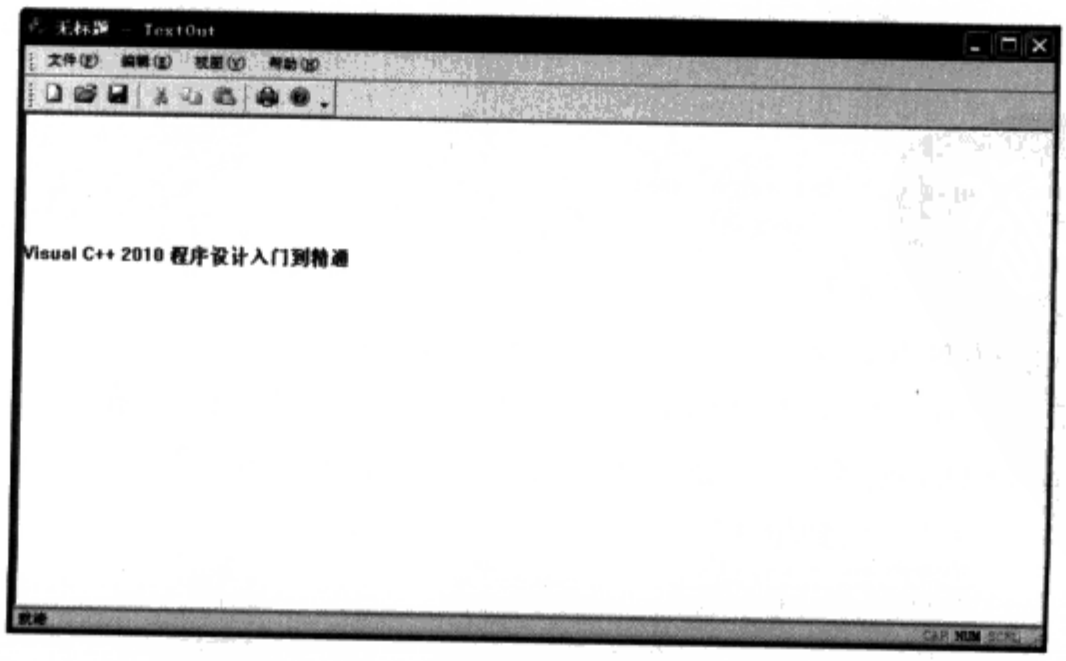


图 8.19　输出字符串资源

8.4.2 定时器和变色文字

文字变色是一个不断变化、自动进行的过程，这意味着需要不断地调用 DrawText()函数，同时增大矩形的宽度。要实现这个功能，需要用到定时器，通过定时器来自动控制文字变色的过程。定时器的功能是每隔一定时间，就发送一个消息，开发人员收到这样的消息，就知道时间到了。利用定时器不断发送消息的特点，可以在响应定时器消息的响应函数中，不断增加显示文字的矩形宽度，从而实现平滑文字的变色效果。

使用 CWnd 类的 SetTimer()成员函数可以设置定时器，该函数声明如下：

```
UNIT SetTimer (UINT nIDEvent , UINT nElapse , void (CALLBACK EXPORT*
                lpfnTimer) (HWND , UINT , UINT , DWORD ));
```

如果函数调用成功，则返回新定时器的标识符。应用程序可以将这个值传递给 KillTimer()成员函数以销毁定时器。

参数介绍如表 8.3 所示。

表 8.3 SetTimer 函数参数介绍

参　数	说　明
nIDEvent	指定了不为 0 的定时器标识符
nElapse	指定了定时值，以毫秒为单位
LpfnTimer	指定了应用程序提供的 TimerProc()回调函数的地址，该函数被用于处理 WM_TIMER 消息。如果这个参数为 NULL，则 WM_TIMER 消息被放入应用程序的消息队列并由 CWnd 对象来处理
SetTimer	函数设置一个系统定时器，指定了一个定时值，一旦发生超时，系统就向设置定时器的应用程序的消息队列发送一个 WM_TIMER 消息，或者将消息传递给应用程序定义的 TimerProc()回调函数

在工程 TextOut 的 CTextOutView 类中添加成员变量 int width，用于记录矩形的边界。将该变量在构造函数中初始化为 0。

下面就来实现在窗口刚刚产生的时候，生成一个定时器。

（1）在工程 TextOut 的 CTextOutView 类中加入 WM_CREATE 消息的处理函数 OnCreate()。

（2）在 OnCreate()函数中加入以下代码：

```
int CTextOutView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    SetTimer(1,100,NULL);
    // TODO:  Add your specialized creation code here
    return 0;
}
```

代码 SetTimer(1,100,NULL);表示设置了一个每隔 0.1ms 就发送一次定时器消息的定时器。接着，就通过定时器来实现将红色的文本“Visual C++ 2010 程序设计入门到精通”逐渐变成蓝色。在 CTextOutView 类的 OnDraw()函数中添加以下代码，以实现在应用程序开始时屏幕上输出红色的文本“Visual C++ 2010 程序设计入门到精通”。

```
void CTextOutView::OnDraw(CDC* pDC)
{
    CTextOutDoc* pDoc = GetDocument();
```

```
    ASSERT_VALID(pDoc);
                                        // TODO: 在此处为本机数据添加绘制代码
                                        //定义并初始化字符串
    CString str;
    str.LoadString(IDS_NEWSTRING);
    pDC->SetTextColor(RGB(255,0, 0));
    pDC->TextOut(0,50,str);
}
```

. 打开 ClassWizard，在工程的 CTextOutView 类中加入 WM_TIMER 消息的处理函数 OnTimer()，在该函数中加入以下代码以实现文字颜色渐变的效果。

```
void CTextOutView::OnTimer(UINT_PTR nIDEvent)
{
    // TODO: Add your message handler code here and/or call default
    CString str;
    str.LoadString(IDS_NEWSTRING);
    CClientDC cdc(this);
    cdc.SetTextColor(RGB(0,0,255));
    CRect rect;
    rect.top=50;
    rect.left=0;
    CSize sz=cdc.GetTextExtent(str);
    rect.bottom=50+sz.cy;
    width=width+5;
    rect.right=width;
    cdc.DrawText(str,&rect,DT_LEFT);
    if(width>sz.cx)
    {
        width =0;
        cdc.SetTextColor(RGB(0,255,0));
        cdc.TextOut(0,50,str);
    }
    CView::OnTimer(nIDEvent);
}
```

开始还是要加载字符串资源，然后定义 rect 的 top 和 left 分别为 50 和 0（即左上角坐标为（0,50）），因为在开始的 OnDraw()函数中输出文本时，是在（0,50）的位置输出的，所以为了看到颜色渐变的效果，也要将变色的文本在同一位置上输出。矩形 rect 的右下角（right）是随着 Timer 事件不断变化的，但有一点可以肯定，其 bottom 始终不变，始终是文本输出的左上角的 y 值加上文本的高度。

最后，使用 DrawText()函数实现卡拉 OK 逐步变色的效果。在 OnDraw()中输出了一次 IDS_NEWSTRING 标识的字符串，在 OnTimer()中，将文字颜色设置为另一种颜色，然后用 DrawText()函数在原来的位置再输出一次字符串，开始在一个小矩形上输出，每调用一次 OnTimer()，矩形的宽度就变大一点，所得到的效果就是文字在一点一点地变色，模拟了“卡拉 OK”的逐渐变色的效果。

程序的最终运行结果如图 8.20 所示。

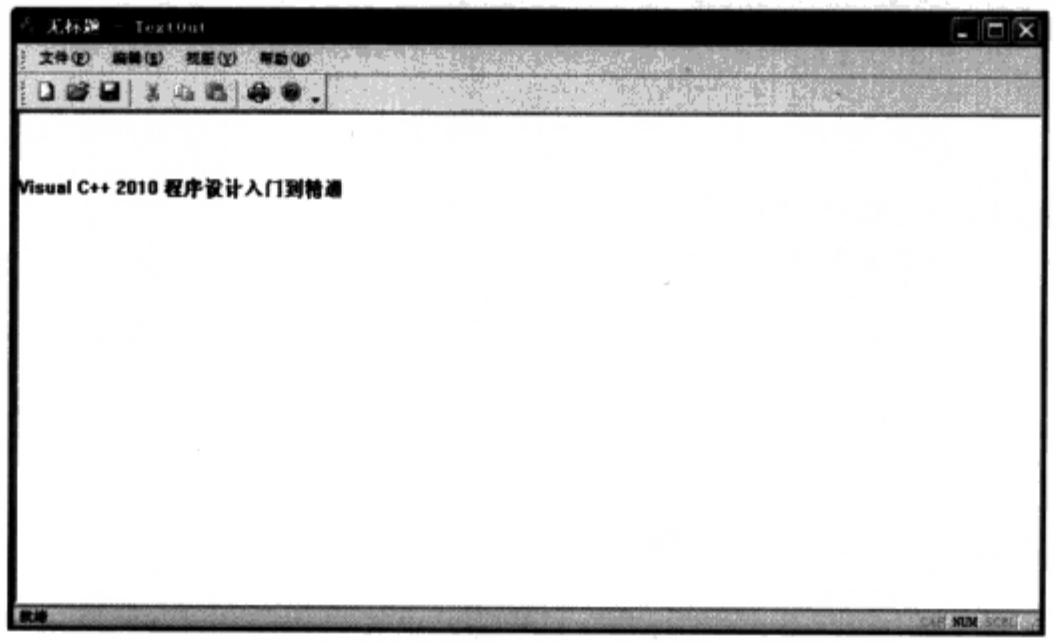


图 8.20　文字的平滑变色结果

8.5　小结

本章详细介绍了 Visual C++ 2010 MFC 文本与字体编程的方法，以及如何使用这些方法实现简单的文字处理程序。本章的重点是：

- CFont 类是 Visual C++ 2010 MFC 提供的一个专门用来设置字体的类，使用该类构造一个 CFont 对象后，还必须使用其提供的 4 个初始化函数中的一个来对 CFont 对象进行初始化；
- 创建一个通用的文本插入符和图像插入符的常用方法；
- 使用 TextOut()函数输出指定的文字，并通过 CFont 类提供的方法实现对输出字体格式的改变；
- 捕获用户输入的文字并实现在应用程序窗口中的输出，注意回车键(Enter)和退格键(Backspace)等特殊字符应该怎样处理；
- 字符串资源的创建和使用方法；
- 定时器的创建和使用方法；
- 使用 DrawText()函数实现文字颜色的渐变的方法。

第 9 章　Visual C++ 2010 MFC 图形图像编程

本章主要介绍了如何使用 Visual C++ 2010 MFC 实现对图形图像的编程。首先介绍了 Windows 绘图及 Windows 屏幕绘图的一些原理，接着介绍了 Windows 绘图中的一个重要元素——图形设备接口（GDI）；然后又通过使用 MFC 中的 GDI 提供的笔和画刷对象实现简单的绘画功能介绍了 Windows 绘图的具体实现。完成本章的学习之后，读者将对 Visual C++ 2010 MFC 图形图像编程原理有一定的了解，并且能够自己绘制各种不同的图形。

本章主要内容有：

- Windows 绘图简介;
- Windows 屏幕绘图简介;
- 微软 GDI 绘图简介;
- GDI 笔绘图;
- GDI 画刷绘图。

9.1　Windows 绘图简介

Windows 绘图不仅是指常见的屏幕绘图，还包括在所有输出设备上的绘图。使用应用程序可以直接向一个物理设备输出，比如显示器或打印机；也可以向一个“逻辑”设备输出，比如内存设备或元文件。本节将从总体上把握一下 Windows 绘图过程中所涉及的一些原理，包括设备描述表、绘图属性、元文件和调色板等基本概念。

9.1.1　设备描述表

绘制一条直线或者填充一个区域时，必须首先确定这个绘图过程发生时的设备描述表。设备描述表是 Windows 定义的一种数据结构，它包含的信息允许 Windows 将输出请求转换为对正在使用的特定物理输出设备的动作，输出请求采用设备无关性 GDI 函数调用的形式。将图像绘制在任何显示器或者任何打印机上，在这个过程中，不需要考虑显卡或打印机的类型。

Windows 使用设备描述表（Device Context）对象的方法——该对象的创建没有使用一个 C++类，原因显而易见，Windows 早于 C++存在。设备描述表对象用一些默认值创建，例如应用 Windows 绘制一条直线，所需的是一个设备描述表对象的指针和直线坐标，设备描述表对象的指针叫做设备描述表句柄（HDC），可以通过调用 Windows API 函数获得设备描述表句柄。

获得设备描述表句柄的方法主要有两种。

一种是调用 BeginPaint()函数。

应用程序响应 WM_PAINT 消息进行图形刷新时，主要通过调用 BeginPaint()函数获取设备环境：

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

ps 为 PAINTSTRUCT 类型结构，定义方式为 PAINTSTRUCT ps。

由 BeginPaint()函数获取的设备环境要用 EndPaint()函数释放：

```
void EndPaint(HWND hwnd, PAINTSTRUCT &ps);
```

另一种是调用 GetDC()函数。

如果绘图工作并非由 WM_PAINT 消息驱动，则调用 GetDC()函数获取设备环境。

```
hdc=GetDC(hwnd);
```

由 GetDC()函数获取的设备环境必须用 ReleaseDC()函数释放。

```
void   ReleaseDC(HWND hwnd);
```

以上这两种方法的区别是：

使用 BeginPaint()函数的方法只用于图形刷新时获取设备环境，获取设备环境后，操作区域为无效区域，最后由 EndPaint()函数释放。而 GetDC()函数使用较广泛，获取设备环境后，操作区域为整个客户区，最后由 ReleaseDC()函数释放。

因为设备描述表在内存中创建，而内存的地址是经常变化的，所以一个设备描述表句柄不是直接指向设备描述表对象，而是指向另外一个跟踪设备描述表地址的指针。

设备的无关性是指操作系统屏蔽了硬件设备的差异，因而设备无关性能使用户编程时无须考虑特殊的硬件设置，它允许开发人员设置一个设备描述表，而不管输出设备是屏幕还是打印机。设备描述表创建完成之后，可以被分配给任意一台打印机、屏幕或者文件。

关于设备描述表的详细信息，将在 9.3 节中做详细介绍。

9.1.2 绘图属性

前面介绍了使用设备描述表对象来保存所有绘图的一些属性信息，由于所有的绘图函数共同使用一个设备描述表，因此单个函数可能只使用存储在设备描述表中的一小部分属性。

一个设备描述表既可以自身包含一种特定的绘图属性，也可以指向另一个包含某种特定属性的对象。例如，画一条直线的属性可以存储在一个单独的 Pen 对象中，而由设备描述表指向该对象。这些属性对象的设置及怎样将设备描述表对象指向该对象，将在 9.3 节中再做详细介绍。

可以通过设备描述表，设置画线属性、填充属性、文本属性、映射模式、调色板属性、混合属性、剪裁属性和位图属性等。

9.1.3 元文件和路径

元文件和路径分别是 Windows 和 MFC 提供的保存一幅图像供以后使用的方法：

元文件是一个结构的集合，这些结构是以与设备无关的格式来存储图像，实质上是捕获任何绘图到一个设备描述表中，以便以后能重放或者存储到磁盘上。位图一旦装载到内存中，就和设备无关，因此可以认为元文件在内存中是设备无关的位图。一旦应用程序将输出结果记录到了一个元文件中，那么该元文件就可以被使用任意多次，并且输出结果可以被发送到任意多个物理设备上。但是，元文件有一个缺点：它通常比位图的绘图速度慢。因此，如果一个应用程序要求有较快的绘图速度，而不需要具有设备无关性，则应该用位图代替元文件。

路径，像元文件一样捕获绘图函数，但路径能绘制修改版形状，也可以填充一个路径图像或者把它变成一个剪裁区域。

9.1.4 颜色和调色板

在 Windows 中的大部分程序，我们使用 RGB()宏定义绘图颜色，RGB()宏的语法如下：

```
RGB (BYTE nRed, BYTE nGreen, BYTE nBlue)
```

nRed、nGreen和nBlue代表颜色强度，它们的范围为0～255，例如要指定红色，可以使用RGB(255，0，0)，白色是RGB(255，255，255)，黑色是(0，0，0)。

用RGB宏创建的颜色实际上是抖动色，抖动色是由模拟预期颜色的不同颜色点的图案产生的。例如，如果仔细地看用RGB(255，127，127)表示的粉红色，会察觉到它实际上是由散布的红白像素构成的，远看就成了粉红色。使用RGB( )宏时，标准色是单色的，如红色和白色，但是任何非标准色都是抖动色。抖动处理使得用户能够绘制任何可能的颜色，而不必考虑系统调色板和视频内存是否支持。然而，任何非标准色看起来都会有点模糊。

屏幕上的每个像素都能实际显示从黑到白的整个颜色范围，这意味着可以通过直接设置一个像素颜色来显示任何可能的颜色，而不经抖动。那为什么还要进行抖动呢？所有像素都需要一个32位的RGB值来表示它是什么颜色的，这样，假设要填充一个800像素×600像素分辨率的屏幕，将需要大约2MB的显示内存，这样将会大大增加对显卡等设备的性能负担。

系统调色板可以得到任何色调的纯色而不需要大容量的显存，它的创建是基于如下思想的：虽然在画面中可能有成千上万个像素，但它们可能只包含几百种纯色，这样就不必重复存储相同的颜色值，而是将这些纯色存储到系统的调色板中，从而每个像素点只指向那个调色板。这样，每个像素点可能只要求4或8位，以指向该调色板中的一种颜色，这样对显存的要求就骤然降低。

9.1.5　图形设备接口函数

图形设备接口函数包括以下几类。

位图函数（Bltmap）：位图是一个图形对象，该函数可将图像作为文件进行创建、处理（比例缩放、滚动、旋转和绘制）和存储。位图函数提供了一系列处理位图的方法。

笔刷函数（Brush）：笔刷是一种绘图工具，Win32 应用程序可使用它绘制多边形、椭圆形和路径的内部。

剪裁函数（Clipping）：剪裁是一种处理过程，它将输出到某个区域或路径中的内容限制在应用程序窗口的显示区内。剪裁函数提供了一系列处理剪裁区域的方法。

颜色函数（Color）：颜色是组成 Win32 应用程序所生成的图片和图像的一个重要元素。Win32 API 提供了一系列管理和使用画笔、笔刷、文本和位图的颜色的函数。

坐标空间及映射函数（Coordinate Space and Transformation）：Win32 应用程序使用坐标空间和映射函数对输出的图形进行比例缩放、旋转、转换、剪裁和反射。坐标空间是基于笛卡尔坐标系统的一个平面空间。该坐标系统要求有两个垂直相交的、长度相等的坐标轴。共有 4 种坐标空间：现实坐标、页面坐标、设备坐标、物理设备坐标（显示区，或桌面，或打印纸的页面）。映射方式就是改变（“映射”）对象的大小、方向和形状的一种算法。

设备环境函数（Device Context）：设备环境函数用于对设备环境进行创建、删除或获取信息。

填充图形函数（Filled Shape）：填充图形是一些几何图形，其轮廓由当前的画笔绘制，内部由当前的笔刷填充。共有 5 种填充图形：椭圆、弦图、饼图、多边形、矩形。填充图形函数用于对填充图形进行操作。

字体和文本函数（Font and Text）：字体用于在视频显示器或其他输出设备上绘制文本。Win32 API 提供了一系列用于安装、选择和查询各种字体的字体和文本函数。

9.2 Windows 屏幕绘图简介

在下一节开始使用 MFC 进行绘图之前，本节先来了解一下 Windows 屏幕绘图。与 Windows 中的其他操作类似，对屏幕上的窗口进行的绘图操作都是通过使用 Windows API 函数完成的。从直接的意义上来说，Windows 的最后一个限制是实际上没有把数据写入屏幕。所有到屏幕的输出都是图形，而不管它是直线、圆还是文本，Windows 使用 GDI（图形设备界面）定义这种输出。

对一个窗口完成一次绘图操作后，必须还要对一些重绘操作进行管理，例如窗口大小的改变、窗口的移动等。通常，Windows 会自动完成这些操作，当需要重新绘制整个或部分窗口时，Windows 将向应用程序发送 WM_PAINT 消息，接收到该消息的应用程序中的某个函数会截获这个事件，然后进行相应的处理。

9.2.1 窗口客户区

由于 Windows 屏幕上的窗口是可以随时移动的，并且可以根据用户需求随时改变窗口的大小，因此，屏幕上的窗口并没有一个固定的位置，那么 Windows 是如何知道应该在什么位置绘图的呢？

在这里，Windows 使用了窗口客户区的概念。一个 Windows 应用程序的界面分为视图类窗口和框架窗口。如图 9.1 所示，整个程序窗口是框架窗口，工具栏以下白色区域部分是视图类窗口。框架窗口既有客户区（菜单栏以下部分）又有非客户区（程序运行界面中的标题栏和菜单栏）；视图类窗口只有客户区（即视图类窗口本身）。绘图操作一般都是在窗口的客户区进行的，Windows 为一个窗口的客户区提供了一种坐标系统，它对于这个窗口来说是本地的，而且始终把客户区的左上角作为其参考点，客户区中的所有点都是对应于这个点定义的。

不管这个窗口在屏幕上的什么位置，也不管其大小如何，某个点对于客户区左上角的水平和垂直距离始终不变。当然，Windows 需要跟踪这个窗口在什么位置，在客户区中的一个点上进行绘图操作时，Windows 还需要知道这个点在屏幕上的实际位置。

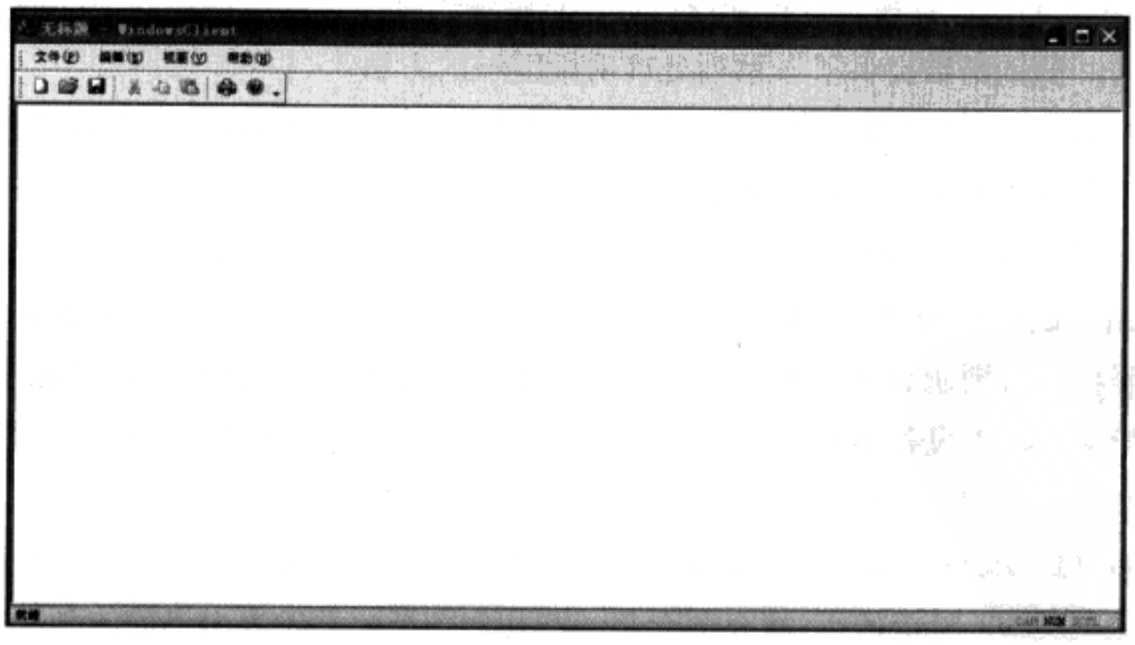


图 9.1 框架窗口

9.2.2 映射模式

设备环境提供了一种称为映射模式的可选坐标系统，它将被自动转换为客户坐标。设备环境中的

每种映射模式都有一个 ID 标识，每个标识都有一个前缀 MM_，表明它将定义映射模式。Windows 提供的映射模式如表 9.1 所示。

表 9.1　　Windows 提供的映射模式

映 射 方 式	逻 辑 单 位	x 轴 增 加	y 轴 增 加	毫　　米
MM_TEXT	像 素 点	右	下	与设备有关
MM_LOMETRIC	0. 1 mm	右	上	0.1
MM_HIMETRIC	0. 01 mm	右	上	0.01
MM_LOENGLISH	0. 254 mm	右	上	0.254
MM_HIENGLISH	0. 025 4 mm	右	上	0.025 4
MM_TWIPS	0.017 6 mm	右	上	0.017 6
MM_ISOTROPIC	任 意（x=y）	可 选	可 选	可 设
MM_ANISOTROPIC	任 意（x!=y）	可 选	可 选	可 设

MM_TEXT 是设备环境的默认映射模式，如图 9.2 所示，它的原点坐标在左上角，x 轴向右逐渐增加，y 轴向下逐渐增加。如果需要使用一种不同的映射模式，就必须设法修改它。

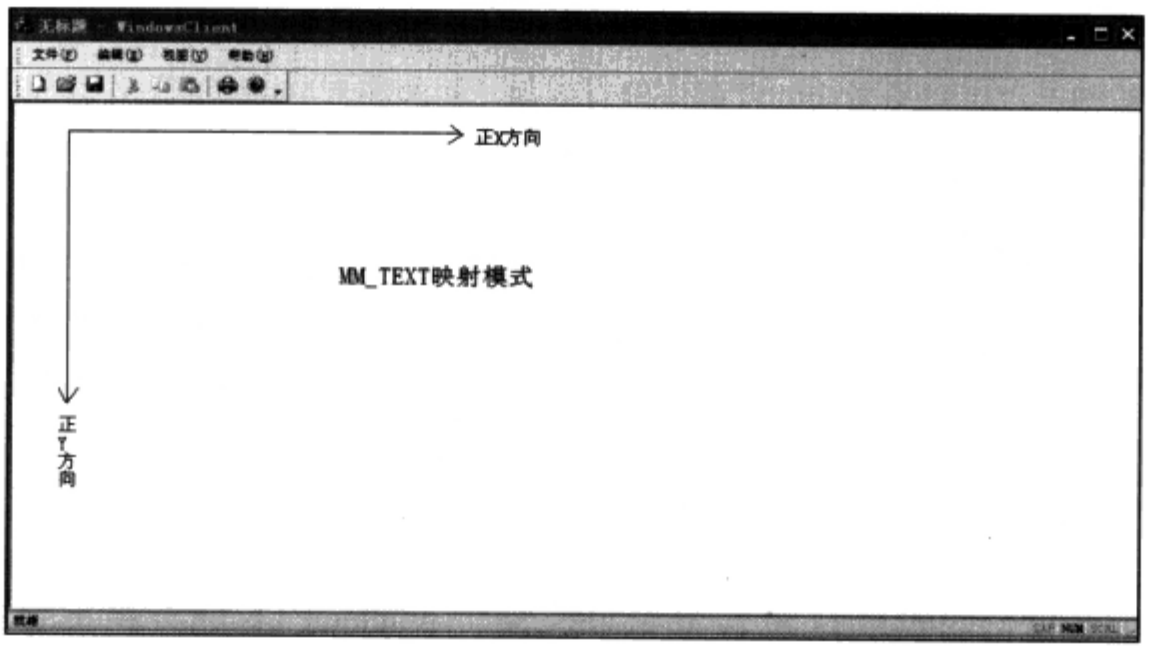


图 9.2　设备环境的默认映射模式

默认情况下，在每种映射模式中，位于客户区左上角的点的坐标都是（0,0），但是如果有需要，也可以把原点移动到客户区左上角以外的位置。当原点在 MM_TEXT 模式中位于左上角时，距左边框 320 个像素、客户区顶部 240 个像素的点的坐标是（320,240）。这里需要注意的是，当屏幕的分辨率改变时，同样的坐标点在屏幕上的位置会发生改变，比如（320,240），在分辨率为 1280 像素×768 像素的屏幕中的位置要比在分辨率为 1280 像素×1024 像素的屏幕中更靠右。同理，不同的分辨率也会影响屏幕显示图形的外观。

在 MM_LOENGLISH 模式和其余所有映射模式中，x 和 y 轴的方向都一样，但是它与 MM_TEXT 不相同。MM_LOENGLISH 模式的坐标轴如图 9.3 所示。

该映射模式的 y 轴由下向上增加，而原点位于客户区的左上角，因此对于可视客户区内的点来说，它的 y 值始终为负。该模式 x 轴上的单位是 0.01 英寸，所以点（320,–240）距左边框 3.2 英寸，距客户区顶部 2.4 英寸。因此，该映射模式的位置不受屏幕分辨率影响。

如果坐标点设置在客户区之外，例如 MM_LOENGLISH 模式下，y 坐标为负，那么将无法看到该点。这时，可以调用 Windows API 函数 SetViewportOrg()或者 MFC 类 CDC 的 SetViewportOrg()来移动参考点的位置。另外，使用坐标空间及映射函数也可以对窗口的坐标空间进行相应的修改。

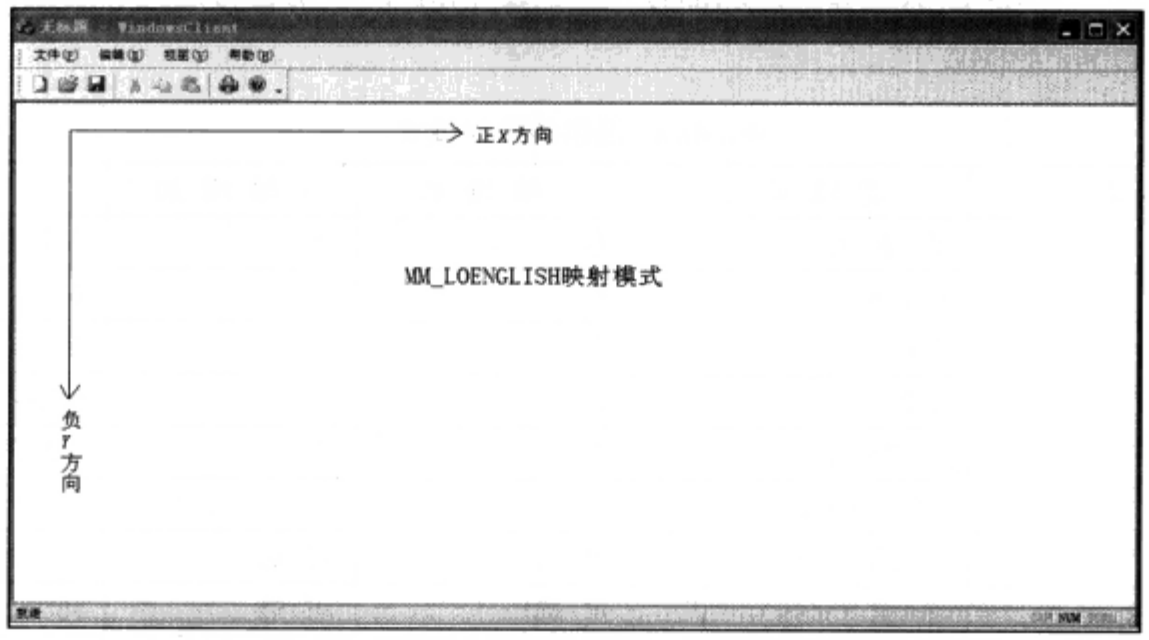


图 9.3　MM_LOENGLISH 模式的坐标轴

9.2.3　图形刷新

图形刷新是绘图过程中必须考虑的重要问题。例如，在绘图程序中绘制一个矩形，选择颜色列表时覆盖了矩形的一部分，则关闭颜色选框后，应用程序需要恢复被覆盖部分的颜色和形状。图形刷新主要包括刷新请求、对刷新请求的响应、刷新方法。

刷新请求主要有 3 种：一是窗口移动后的刷新，如用户窗口移动或显示、用户窗口大小改变、程序通过滚动条滚动窗口等；二是被覆盖区域的刷新，如上面提到的窗口被颜色选框覆盖及下拉式菜单的关闭等；三是对象穿越后的刷新，如光标穿过用户区，图标拖过用户区等，这种刷新会由系统自动完成。

系统对刷新请求的响应是指当用户区的内容需要刷新时，系统向应用程序消息队列发送 WM_PAINT 消息，系统在应用程序的消息队列中加入该消息，以通知窗口函数执行刷新处理。

窗口被另一个窗口覆盖的区域称为无效区域。Windows 系统为每个窗口建立了一个 PAINTSTRUCT 结构，该结构中包含了包围无效区域的一个最小矩形的结构 RECT，应用程序可以根据这个无效矩形执行刷新操作。

Windows 应用程序刷新窗口的方法有 3 种，它们分别是：记录事件，即刷新时重新执行这个曾经发生的事件；保存副本，即刷新时将副本复制到相应的窗口中；重新绘制，即将图形绘制处理程序放在消息 WM_PAINT 响应模块中，刷新时重绘图形。

9.3　微软 GDI 绘图简介

9.3.1　GDI 基础

Windows 图形设备接口（GDI）是为与设备无关的图形设计的，它接受 Windows 应用程序的绘图请求（表现为 GDI 函数调用），并将它们传给相应的设备驱动程序，完成到特定硬件的输出，比如打印机输出和屏幕输出。

GDI 负责 Windows 的所有图形输出，包括在屏幕上输出像素，在打印机上输出硬备份，以及绘制 Windows 用户界面。应用程序可以使用 GDI 创建 3 种类型的图形输出：矢量输出、光栅图形输出和文

本输出。

9.3.2　GDI 结构

为了支持 GDI 绘图，MFC 提供了两种重要的类：设备上下文类，用于设置绘图属性和绘制图形；绘图对象类，封装了各种 GDI 绘图对象，包括画笔、刷子、字体、位图、调色板和区域。

1．设备上下文类

设备上下文类包括 CDC 及其派生类 CClientDC、CPaintDC、CWindowDC、CMetaFileDC。

CDC 是设备上下文类的基类，除了一般的窗口显示外，还用于基于桌面的全屏幕绘制和非屏幕显示的打印机输出。CDC 类封装了所有图形输出函数，包括矢量、光栅和文本输出。

CClientDC 用于客户区的输出，它在构造函数中封装了 GetDC()函数，在析构函数中封装了 ReleaseDC()函数。一般在响应非窗口重画消息（如键盘输入时绘制文本、鼠标绘图）时要使用它。

CPaintDC 用于响应窗口重绘消息（WM_PAINT）。CPaintDC 在构造函数中调用 BeginPaint()取得设备描述表，在析构函数中调用 EndPaint()释放设备描述表。EndPaint()除了释放设备描述表外，还负责从消息队列中清除 WM_PAINT 消息，因此，在处理窗口重画时，必须使用 CPaintDC，否则 WM_PAINT 消息无法从消息队列中清除，将引起不断的窗口重画。CPaintDC 也只能在 WM_PAINT 消息处理之中使用。

CWindowDC 用于窗口客户区和非客户区（包括窗口边框、标题栏、控制按钮等）的绘制。除非用户要自己绘制窗口边框和按钮（如一些 CD 播放程序等），一般情况下不会使用这个类。

CMetaFileDC 专门用于图元文件的绘制。图元文件记录一组 GDI 命令，可以通过这组 GDI 命令重建图形输出。使用 CMetaFileDC 时，所有的图形输出命令会自动记录到一个与 CMetaFileDC 相关的图元文件中。

2．图形对象类

图形对象类包括 CGdiObject、画笔、画刷、字体、位图、调色板、区域等。CGdiObject 是图形对象类的基类，但该类不能直接为应用程序所使用。要使用 GDI 对象，必须使用其派生类：画笔、刷子、字体、位图、区域等。

使用图形对象时，要注意两点。

- 与其他 MFC 对象一样，GDI 对象的创建也要分为两步：第一步，定义一个 GDI 绘图对象类的实例；第二步，调用该对象的创建方法真正创建对象。
- 创建对象：使用该对象，首先要调用 CDC::SelectObject()将其选入设备描述表中，同时保存原来的设置到一个 GDI 对象指针，比如说 pOldObject 中。在使用完后，再用 SelectObject（pOldObject）恢复原来的设置。但是，如果该设备上下文是用户自己创建的，则不必恢复原来的设置，因为框架会在该设备描述表生存期结束时删除它，同时也就删除了原来存放于该设备描述表中的绘图对象设置。

9.3.3　GDI 函数调用

组成 GDI 的几百个函数调用可以分为几大类：

1．获取和释放设备描述表的函数

在绘图时需要设备内容代号，GetDC()和 RealseDC()函数可以让开发人员在非 WM_PAINT 的消息

处理期间来做到这一点，而 BeginPaint()和 EndPaint()函数可以在进行绘图的 WM_PAINT 消息处理期间使用。

2．获得设备描述表信息的函数

这类函数最接近硬件，为了使设计的程序有更好的适用性和可移植性，最好能在编写程序时了解硬件设备信息，使编写出的程序受硬件的限制和影响尽可能小。例如，可以使用 GetTextMetrics()函数来取得有关设备内容中目前所选字体的尺寸信息。

3．设定和获取设备描述表属性的函数

设备内容的“属性”决定有关绘图函数工作的细节，并且涉及程序绘图的各个方面。例如，可以用 SetTextColor()来指定 TextOut（或者其他文字输出函数）所绘制的文字色彩。设备内容的所有属性都有默认值，取得设备内容时这些默认值就设定好了。对于所有的 Set 函数，都有相应的 Get 函数，以允许开发人员取得目前设备内容属性。

4．使用 GDI 对象的函数

Windows GDI 有 6 种可创建的 GDI 对象，它们是画笔、画刷、位图、字体、区域和调色板。有许多用于创建这些对象的函数和结构，例如，函数 CreatePen()用于创建逻辑画笔，函数 CreateBitmap()用于创建 GDI 位图。

5．绘图函数

这些函数是程序执行绘图工作的真正部分，前面介绍的各类 GDI 函数和属性的使用都是在为使用这类函数做准备。例如，使用 TextOut()函数在窗口的显示区域显示一些文字，使用其他的绘图函数还可以画线、画矩形等。

9.3.4 GDI 基本图形

在屏幕或打印机上显示的图形形态本身可以被分为几类，通常被称为“基本图形”，有：

1．直线和曲线

线条是所有向量图形绘制系统的基础。GDI 支持直线、矩形、椭圆、弧及贝塞尔曲线等。所有更复杂的曲线可由折线（polyline）代替，折线通过一组非常短的直线来定义一条曲线。线条用设备内容中目前选中的画笔绘制。

2．画刷

当一系列直线或者曲线封闭一个区域时，该区域可以使用目前 GDI 画刷对象进行填充。这个画刷可以是实心色彩、图案或者是在区域内垂直或者水平重复的位图图像。

3．位图

位图是位的矩形数组，这些位对应于显示设备上的像素，它们是位映像图形的基础工具。位图通常用于在视频显示器或者打印机上显示复杂（一般都是真实的）图像。位图还可以用于显示必须快速绘制的小图像（诸如图标、鼠标光标及在应用工具栏中出现的按钮等）。

4．文字

用于定义 GDI 字体对象和取得字体信息的数据结构是 Windows 中最庞大的部分之一，目前 GDI 支持多种字体的绘制和显示。

9.4　GDI 笔绘图

Visual C++ 2010 中的 MFC 提供了一个 CPen 类，该类封装了 Windows 图形设备接口（GDI）中的画笔工具，它可以帮助开发人员绘制直线或曲线等。开发人员可以使用 MFC 提供的几种画笔进行绘图，也可以根据需要定制自己的画笔工具。

9.4.1　Cpen 类简介

MFC 中的画笔工具，是通过 CPen 类提供的。使用 CPen 类，必须引入头文件 afxwin.h。下面先了解一下 CPen 类的成员，如表 9.2 所示。

表 9.2　　CPen 类的成员

构 造 函 数	
CPen()	构造一个 CPen 对象
初始化函数	
CreatePen()	使用指定的样式、宽度和画笔属性创建一个画笔，并把它加载给 CPen 对象
CreatePenIndirect()	使用指定的样式、宽度和画笔属性创建一个 LOGPEN 结构的画笔，并把它加载给 CPen 对象
操　　作	
FromHandle()	给定一个 Windows HPEN 对象句柄时，返回一个指向 CPen 对象的指针
属　　性	
GetExtLogPen()	得到一个 ExtLogPen 的底层结构
GetLogPen()	得到一个 LogPen 的底层结构
operator HPEN()	返回一个 CPen 对象上的 Windows 句柄

9.4.2　使用 GDI 绘制线条

GDI 与 CPen 相结合，能实现很多复杂的绘图功能，但是这里只是实现一个绘制线条的操作，而其他复杂的绘图都是在这个操作的基础上实现的。

首先，打开 Microsoft Visual Studio 2010，创建一个名为 UsePen 的 MFC 单文档应用程序。

因为两点决定一条直线，所以要绘制一个直线，必须将线条的始点和终点传递给画线函数。在这里，假设鼠标左键按下时开始画线，松开时画线结束，因此，在 CUsePenView 类中添加一个全局变量 m_point，来存放画线时鼠标按下的位置。变量 m_point 的类型设置为 CPoint，访问权限设置为 private。代码如下：

```
private:
                                                    //记录鼠标按下的位置
    CPoint m_point;
```

在 CUsePenView()构造函数中初始化变量为 0，代码如下：

```
CUsePenView::CUsePenView()
{
    m_point=0;
                                                    // TODO: 在此处添加构造代码
}
```

接着，为了获取鼠标按下和松开的位置，在 CUsePenView 类中添加 WM_ LBUTTONDOWN 和 WM_LBUTTONUP 这两个消息的消息响应函数 OnLButtonDown()和 OnLButtonUp()。然后在 OnLButtonDown()函数定义中保存鼠标按下的消息，即将鼠标按下时的坐标点传给 m_point。

在 Visual Studio 2010 自动生成的两个消息响应函数中，都有一个 CPoint 参数，该参数即保存了鼠标此时的位置。代码如下：

```
void CUsePenView::OnLButtonDown(UINT nFlags, CPoint point)
{
                                                    // TODO: Add your message handler code here and/or call default
                                                    //记录鼠标按下的位置
    m_point=point;
    CView::OnLButtonDown(nFlags, point);
}
void CUsePenView::OnLButtonUp(UINT nFlags, CPoint point)
{
                                                    // TODO: Add your message handler code here and/or call default
    CView::OnLButtonUp ( nFlags , point );
}
```

得到鼠标按下和松开的坐标位置后，就可以使用 CDC 类来实现画图了。

下面就实现如图 9.4 所示的画线功能。

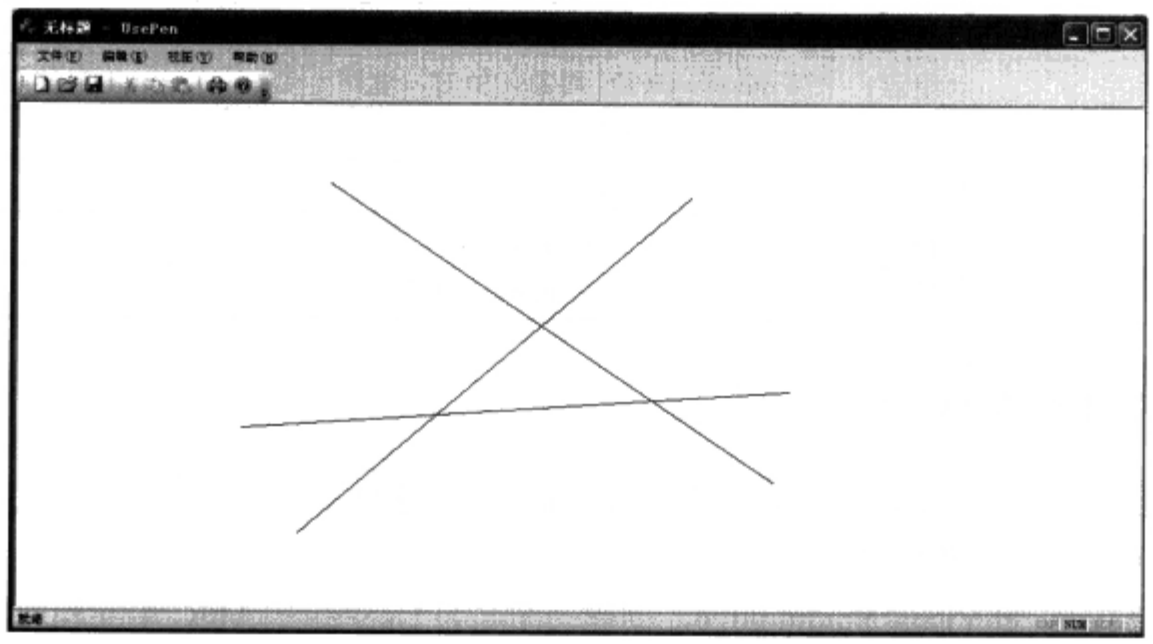


图 9.4　画线功能

1．使用 MFC 的 CDC 类实现画线功能

首先使用基类 CDC 来实现画线的操作。CDC 类封装了所有与绘图相关的操作，该类有一个数据成员 m_hDC，用来存放与 CDC 类相关的 DC 句柄。

使用该类实现画线功能，首先要定义一个 CDC 类型的指针，并利用 CWnd 类的成员函数 GetDC()获得当前窗口的设备描述表对象的指针，然后调用 CDC 类的 MoveTo()函数回到画线的起点，接着调用 LineTo()函数完成画线。最后，还要调用 CWnd 类的成员函数 ReleaseDC()来释放设备描述表资源。

具体的代码如下：

```
void CUsePenView::OnLButtonUp(UINT nFlags, CPoint point)
{   // TODO: Add your message handler code here and/or call default
                                                    //定义一个 CDC 类型的指针并指向当前窗口的设备描述表
    CDC* pcdc=GetDC();
```

```
                                        //完成画线操作
    pcdc->MoveTo(m_point);
    pcdc->LineTo(point);
                                        //释放当前窗口的设备描述表
    ReleaseDC(pcdc);
    CView::OnLButtonUp(nFlags, point);
}
```

2．使用 MFC 的 CClientDC 类实现画线功能

下面使用 CClientDC 类来实现画线功能。CClientDC 类用于客户区的输出，与 CDC 类的区别是它在内部构造函数中封装了 GetDC()，在析构函数中封装了 ReleaseDC()函数。也就是说，在使用 CClientDC 类时不用显式地调用 GetDC()和 ReleaseDC()函数了。下面介绍如何使用 CClientDC 类来实现画线。

首先，必须声明一个 CClientDC 类的对象，该对象需要用一个 CWnd 类型的指针来初始化。因为绘图操作发生在 CUsePenView，因此可以直接将一个 this 指针作为参数传递给 CClientDC 的构造函数。生成 CClientDC 类的对象 ccdc 后，无须调用 GetDC()函数。

随后，直接调用 CClientDC 类的 MoveTo()和 LineTo()函数即可实现画线，而且，画线结束后，无须再调用 ReleaseDC()函数。

具体的代码如下：

```
void CUsePenView::OnLButtonUp(UINT nFlags, CPoint point)
{   // TODO: Add your message handler code here and/or call default
                                        //生成一个 CClientDC 类的对象，使用 UsePenView 对象的指针进行初始化
    CClientDC ccdc(this);
                                        //实现画线操作
    ccdc.MoveTo(m_point);
    ccdc.LineTo(point);
    CView::OnLButtonUp(nFlags, point);
}
```

3．使用 MFC 的 CWindowDC 类实现画线功能

下面再介绍使用 CWindowDC 类实现画线这个操作。CWindowDC 类与前面的两个类的不同是，CWindowDC 类也是在构造时调用 GetWindoDC()函数来获得相应的设备描述表的对象，在析构时调用 ReleaseDC()函数来释放设备描述表所占用的资源。CWindow DC 与 CClientDC 类函数的不同是，由该类生成的对象可以访问整个窗口区域，包括框架窗口的客户区和非客户区。

下面介绍这个类的实现。CWindowDC 类的操作与 CClientDC 类的操作是一样的，具体实现如下：

```
void CUsePenView::OnLButtonUp(UINT nFlags, CPoint point)
{   // TODO: Add your message handler code here and/or call default
                                        //生成一个 CClientDC 类的对象，使用 UsePenView 对象的指针进行初始化
    CWindowDC cwdc(this);
    cwdc.MoveTo(m_point);
    cwdc.LineTo(point);
    CView::OnLButtonUp(nFlags, point);
}
```

运行刚才写好的程序，读者会发现该程序与先前 CClientDC 类生成的程序并没有任何区别，这是因为当前绘图实现的视图类窗口并没有区分客户区与非客户区。可以在 CMainFrame 类框架窗口中再次实现该类，来查看区别。在框架窗口中绘图，需要在构造 CWindowDC 的对象时获得框架窗口的指针作为参数，这个指针可以调用 GetParent()函数来实现。将代码修改如下：

```
void CUsePenView::OnLButtonUp(UINT nFlags, CPoint point)
{   // TODO: Add your message handler code here and/or call default
                                                    //生成一个 CClientDC 类的对象，使用 UsePenView 对象的指针进行初始化
    CWindowDC cwdc(GetParent());//获得父窗口的指针
    cwdc.MoveTo(m_point);
    cwdc.LineTo(point);
    CView::OnLButtonUp(nFlags, point);
}
```

编写完程序后，运行该程序，结果如图 9.5 所示。

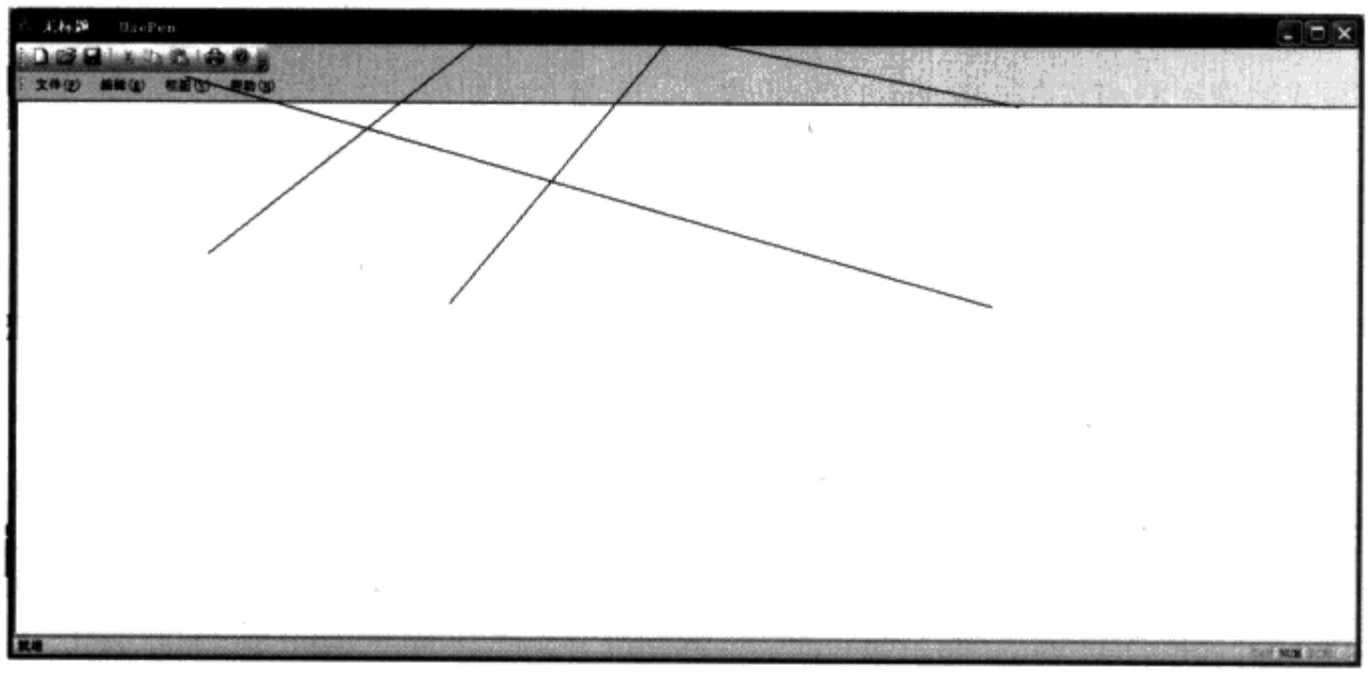


图 9.5　使用 MFC 的 CWindowDC 类实现画线

9.4.3　使用 CPen 类绘制指定的线条

本小节学习如何使用 CPen 类来绘制不同类型的线条。这里将使用设备描述表来设置画笔的属性，关于设备描述表的概念，请参考 9.1.1 小节设备描述表中的内容。

Windows 窗口一旦创建，就自动地产生了与之相对应的设备描述表数据结构，用户可以运用该结构，实现对窗口显示区域的 GDI 操作，如画线、写文本、绘制位图、填充等，并且所有这些操作均要通过设备描述表句柄进行。

在程序中，构造一个 GDI 对象后，该对象并不会立即生效，必须选入设备描述表，才会在以后的绘制操作中生效。SelectObject()函数便是 GDI 提供的把 GDI 对象选入设备描述表的函数。该函数的返回值是指向先前被选对象的指针，其目的是当完成当前绘制操作之后，还原设备描述表。在完成绘图之后利用 SelectObject()函数把先前的 GDI 对象选入设备描述表使其恢复到先前的状态。

下面开始学习如何构造一个 CPen 类的对象。CPen 类提供了两个构造函数：

```
CPen (int nPenStyle, int nWidth, COLORREF crColor);
CPen (int nPenStyle, int nWidth, const LOGVRUSH* pLogBrush, int nStyleCount=0, const DWORD* lpStyle=NULL);
```

参数 nPenStyle 表示了一个画笔的类型；nWidth 表示画笔的线宽；crColor 表示所创建画笔的颜色，可以使用 RGB 宏来构建颜色的值；pLogBrush 指向一个 LOGBRUSH 结构，该结构定义了一个画刷的风格、颜色和阴影线种类；参数 nStyleCount 指定了 lpStyle 数组在双字节下的长度，如果 nPenStyle 不是 PS_USERSTYLE，该参数为 0；参数 lpStyle 指向了一个双字节数组，如果 nPenStyle 不是 PS_USERSTYLE，该参数为 0。关于这些参数的具体取值，可以参考 CSDN 帮助文档。

一般情况下，使用第一个函数初始化一个 CPen 对象。下面来看一些不同类型的 CPen 的简单实现。

首先，可以创建一个红色的宽度为 10 的实线线条。代码实现如下：

```
void CUsePenView::OnLButtonUp(UINT nFlags, CPoint point)
{   // TODO: Add your message handler code here and/or call default
                                        //创建一个红色的宽度为 10 的实线画笔
    CPen pen(PS_SOLID,10,RGB(255,0,0));
                                        //生成一个 CClientDC 类的对象，使用 UsePenView 对象的指针进行初始化
    CClientDC ccdc(this);
                                        //将新的画笔对象选入设备描述表，并保存设备描述表中原始的画笔对象
    CPen *pOPen=ccdc.SelectObject(&pen);
                                        //实现画线操作
    ccdc.MoveTo(m_point);
    ccdc.LineTo(point);
                                        //还原设备描述表
    ccdc.SelectObject(pOPen);
    CView::OnLButtonUp(nFlags, point);
}
```

运行结果如图 9.6 所示。

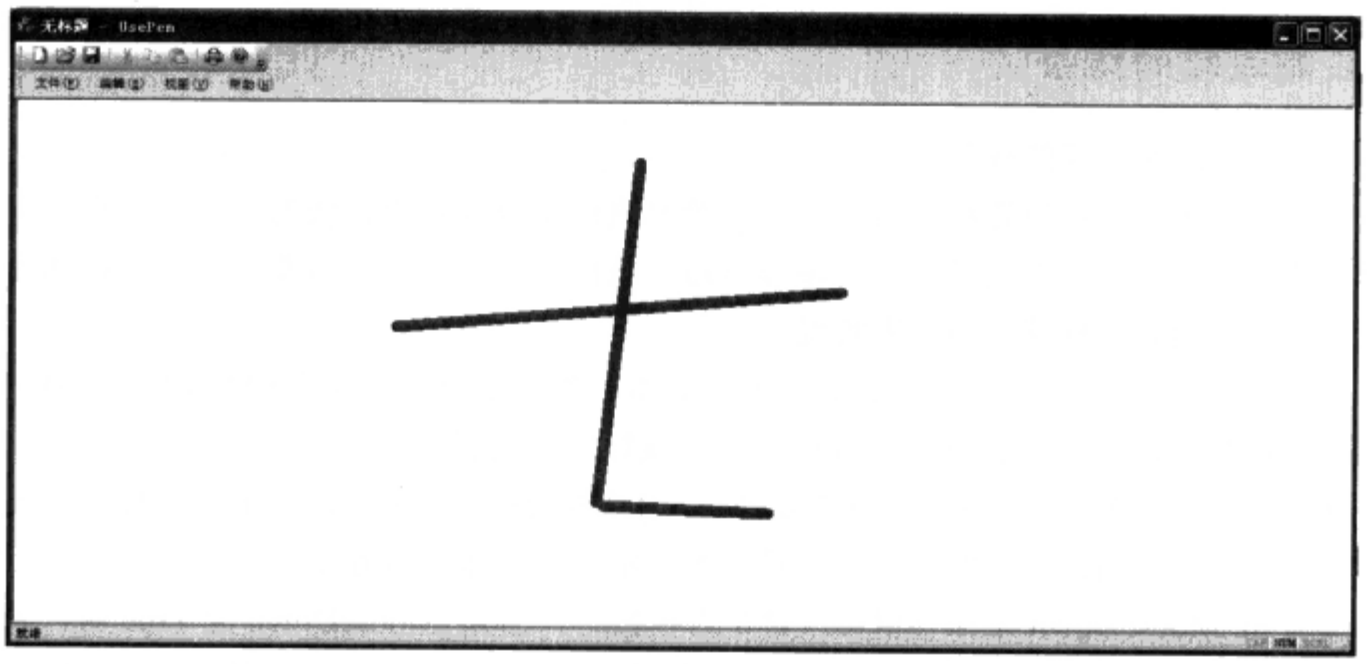


图 9.6　使用 CPen 类绘制指定的线条

接着，生成一个蓝色的虚线画笔。注意：虚线线性只对线宽为 1 的画笔有效。其实现如下：

```
void CUsePenView::OnLButtonUp(UINT nFlags, CPoint point)
{   // TODO: Add your message handler code here and/or call default
                                        //创建一个蓝色的宽度为 1 的虚线画笔
    CPen pen(PS_DASH,1,RGB(0,0,255));
                                        //生成一个 CClientDC 类的对象，使用 UsePenView 对象的指针进行初始化
    CClientDC ccdc(this);
                                        //将新的画笔对象选入设备描述表，并保存设备描述表中原始的画笔对象
    CPen *pOPen=ccdc.SelectObject(&pen);
                                        //实现画线操作
    ccdc.MoveTo(m_point);
    ccdc.LineTo(point);
                                        //还原设备描述表
    ccdc.SelectObject(pOPen);
    CView::OnLButtonUp(nFlags, point);
```

```
}
```

运行结果如图 9.7 所示。

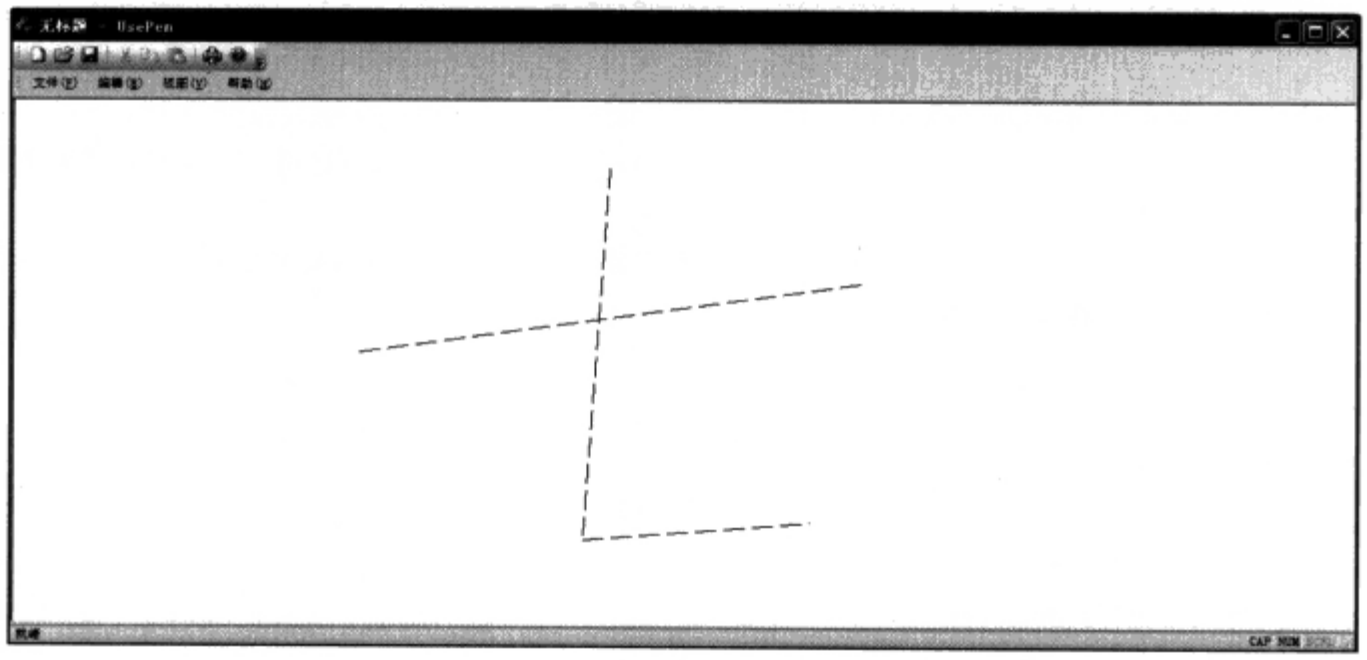


图 9.7　使用 Cpen 类绘制虚线

9.4.4　绘制连续的线条

本小节将实现绘制连续的线条。

首先，需要考虑如何绘制连续的线条。之前绘制直线，只需获得直线的起点和终点即可。利用同样的原理，可以把一个连续的线条看作许多非常短的线段的集合，因此需要捕获鼠标移动经过的每一个点，这可以通过捕获鼠标移动消息来实现。

根据上面的思路，首先在 CUsePenView 类中增加一个鼠标移动消息（WM_MOUSEMOVE）的响应函数 OnMouseMove()，然后在该方法体中添加直线绘制的实现。

但是这里还存在一个问题，因为 OnMouseMove()函数会捕获任何情况下的鼠标移动情况，而这里只希望其鼠标左键按下后，即开始画图的时候，才开始捕获鼠标移动情况，所以我们设置一个 bool 类型的全局变量 m_MouseDown 来保存鼠标左键是否被按下，当 m_MouseDown 为 True 时，表示鼠标左键按下了，相反为 False。然后在 OnMouseMove()中通过判断 m_MouseDown 的状态，来选择是否捕获鼠标移动并绘图。

在 CUsePenView 类添加一个 bool 型全局变量，并且在构造函数中将其初始化为 False：

```
private:
                                        // 判断鼠标左键是否按下的全局变量
    bool m_MouseDown;
```

保存鼠标左键是否按下的代码：

```
void CUsePenView::OnLButtonDown(UINT nFlags, CPoint point)
{
                                        // TODO: Add your message handler code here and/or call default
                                        //记录鼠标按下的位置
    m_point=point;
                                        //表示鼠标左键按下
    m_MouseDown=true;
    CView::OnLButtonDown(nFlags, point);
}
```

```
void CUsePenView::OnLButtonUp(UINT nFlags, CPoint point)
{
                                                    //表示鼠标左键没有被按下
    m_MouseDown=false;
    CView::OnLButtonUp(nFlags, point);
}
```

使用 OnMouseMove()进行绘图的代码：

```
void CUsePenView::OnMouseMove(UINT nFlags, CPoint point)
{
                                                    // TODO: Add your message handler code here and/or call default
    CClientDC ccdc(this);
                                                    //判断鼠标左键是否被按下
    if(m_MouseDown)
    {   //开始绘图
        ccdc.MoveTo(m_point);
        ccdc.LineTo(point);
                                                        //更新线段的起点
        m_point=point;
    }
    CView::OnMouseMove(nFlags, point);
}
```

程序运行的结果如图 9.8 所示。

在窗口中，用户可以使用画笔自由地画图了。当然，如果将上一节画笔的使用添加进来，就可以绘制不同颜色、不同格式的线条了，这里读者可以自己实现一下。

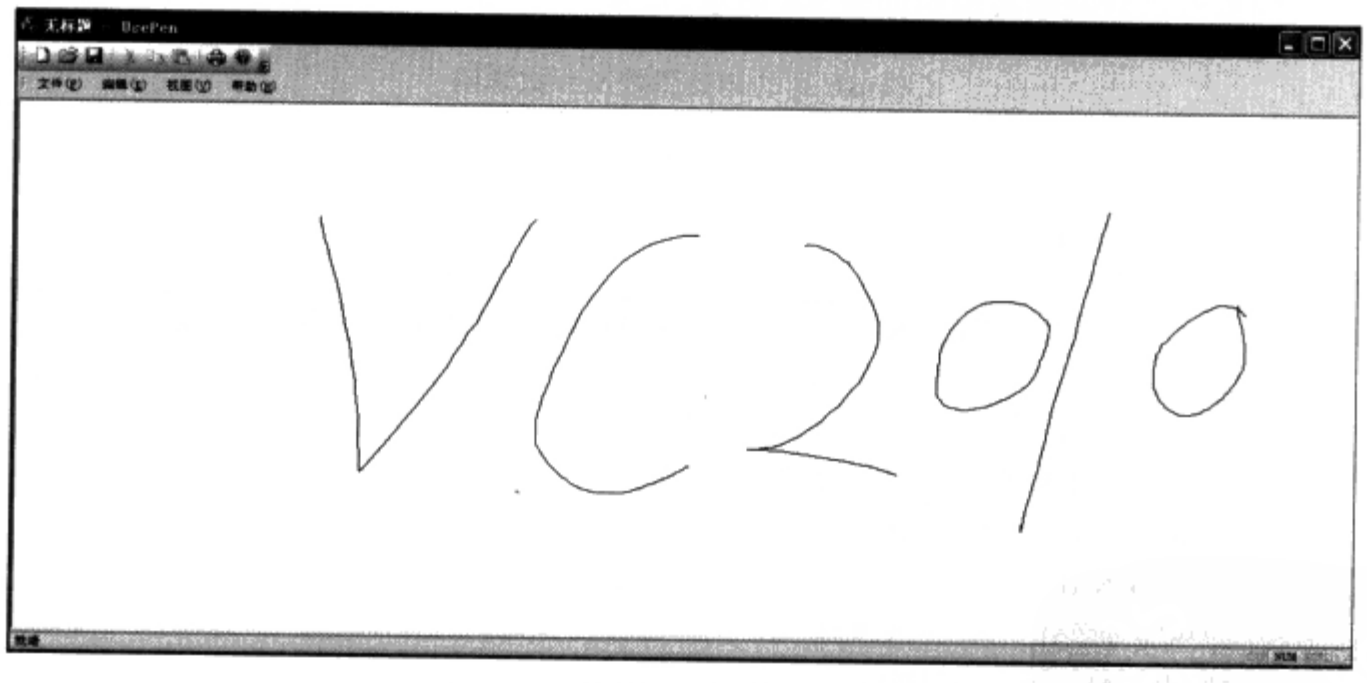


图 9.8　不同颜色不同格式的线条

9.5　GDI 画刷绘图

所谓画刷，是指用于与 Graphics 对象一起创建实心形状或呈现文本的对象的工具，通常用来填充一块区域。与画笔等对象相同，画刷的内部数据结构是由 GDI 来管理的。 Visual C++2010 中的 MFC 提供了一个 CBrush 类，封装了 Windows 图形设备接口（GDI）中的画刷。

9.5.1 CBrush 类介绍

使用 CBrush 类，必须先包含头文件 afxwin.h。下面先了解 CBrush 类的成员，如表 9.3 所示。

表 9.3　　CBrush 类的成员

构 造 函 数	
CBrush()	构造一个 CBrush 对象
初始化函数	
CreateSolidBrush()	用指定的实线初始化画刷
CreateHatchBrush()	用指定的阴影线初始化画刷
CreateBrushIndirect()	用结构 LOGBRUSH 中指定的风格、颜色和模式初始化画刷
CreatePatternBrush()	用位图指定的模式初始化画刷
CreateDIBPatternBrush()	用独立于设备的位图（DIB）初始化画刷
CreateSysColorBrush()	创建一个使用系统默认颜色的画刷
操　　作	
FromHandle()	给出一个 Windows HBRUSH 对象句柄时，返回一个指向 CBrush 对象的指针
属　　性	
GetLogBrush()	取得一个 LOGBRUSH 结构
operator HBRUSH()	返回 CBrush 对象上的 Windows 句柄

下面是这些成员的详细说明。

1．CBrush 类的构造函数

CBrush 类的构造函数及参数说明如表 9.4 所示。

表 9.4　　CBrush 类的构造函数及参数说明

CBrush( ); CBrush(COLORREF crColor); CBrush(int nIndex,COLORREF crColor); explicit CBrush(CBitmap* pBitmap);	
参　数	crColor 指定画刷的前景色（RGB 方式）。如果画刷是阴影线型的，则指定阴影线的颜色。 nIndex 指定画刷阴影线采用的风格，取值如下： HS_BDIAGONAL　45° 的向下阴影线（从左到右） HS_CROSS 水平和垂直方向以网格线做出阴影 HS_DIAGCROSS　45° 的网格线阴影 HS_FDIAGONAL　45° 的向上阴影线（从左到右） HS_HORIZONTAL　水平的阴影线 HS_VERTICAL　垂直的阴影线 pBitmap 指向 CBitmap 对象的指针，该对象指定了画刷要绘制的位图
说　明	类 CBrush 一共有以上 4 个构造函数。第一个构造函数不带任何参数，构造一个未初始化的 CBrush 对象，在使用该对象之前需要另外初始化。如果使用了这个构造函数，则必须用 CreateSolidBrush()、CreateHatchBrush()、CreateBrushIndirect()、CreatePatternBrush()或 CreateDIBPatternBrush()来初始化返回的 CBrush 对象。如果使用了带参数的构造函数，则不再需要初始化 CBrush 对象。带参数的构造函数在出错时会产生一个异常，而不带参数的构造函数总是返回构造成功。 只带有一个参数 COLORREF 的构造函数是用指定的颜色构造一个实线型的画刷。颜色是一个 RGB 值，可以用 WINDOWS.H 中的宏 RGB()构造出来。 带两个参数的构造函数构造一个阴影线型的画刷，参数 nIndex 指定了阴影线模式的指数（index），参数 crColor 指定了画刷的颜色。

续表

说　明	带有一个 CBitmap 型参数的构造函数构造一个模式化的画刷。参数指定一个位图。该位图应该是用 CBitmap::CreateBitmap()、CBitmap::CreateBitmapIndirect()、CBitmap::LoadBitmap()或 CBitmap::CreateCompatiableBitmap()建立或加载的位图。填充模式下的位图的最小尺寸为 8 像素×8 像素

2. 初始化函数

BOOL CreateBrushIndirect(const LOGBRUSH* lpLogBrush) ;

函数的返回值、参数及说明如表 9.5 所示。

表 9.5　　CreateBrushIndirect 函数的返回值、参数及说明

返 回 值	调用成功时返回非 0 值，否则为 0
参　数	lpLogBrush 指向 LOGBRUSH 结构的指针，结构中记录了画刷的相关信息
说　明	本函数用结构 LOGBRUSH 中指定的风格、颜色和模式初始化画刷。初始化以后，该画刷可以被选为任何设备描述表的当前画刷。采用黑白位图创建的画刷使用当前的文本颜色和背景色。位值置为 0 的像素点将用当前的文本颜色绘制，位值置为 1 的像素点将用当前的背景色绘制

BOOL CreateDIBPatternBrush(HGLOBAL hPackedDIB, UINT nUsage);
BOOL CreateDIBPatternBrush(const void* lpPackedDIB, UINT nUsage);

函数的返回值、参数及说明如表 9.6 所示。

表 9.6　　CreateDIBPatternBrush 函数的返回值、参数及说明

返回值	调用成功时返回非 0 值，否则为 0
参　数	hPackedDIB 指定一个全局内存对象，其中包含了一个压缩的独立于设备的位图（DIB）。 nUsage 指明 BITMAPINFO 数据结构（压缩的 DIB 中的一部分）的 bmiColors[]成员是否包含明确的 RGB 值或指向当前逻辑调色板的索引值。参数必须取以下值之一：DIB_PAL_COLORS 颜色表由一个 16 位的索引数组组成；DIB_RGB_COLORS 颜色表中包含字母式的 RGB 颜色值。DIB_PAL_INDICES 只在第二个函数中有效未提供颜色表。位图中包含了指向画刷选中的设备上下文的逻辑调色板的索引。 lpPackedDIB 指向一个包括了 BITMAPINFO 结构的压缩的 DIB。BITMAPINFO 结构后紧跟着的是定义位图像素的字节数组
说　明	本函数用 DIB 指定的模式初始化画刷。画刷随后就可以被任何支持光栅操作的设备上下文选中。 CreateDIBPatternBrush()函数的两个版本在处理 DIB 时有些不同：在前一个函数中，取得 DIB 的句柄需要调用 Windows 函数 GlobalAlloc()来分配内存，然后用 DIB 填充这块内存；在后一个函数中，不再需要调用函数 GlobalAlloc()为压缩的 DIB 分配内存。 一个压缩的 DIB 由一个 BITMAPINFO 结构和紧跟着的一个字节数组（该数组定义了位图的像素）构成。用作填充模式的位图应该是 8 像素×8 像素的。如果位图太大，Windows 将用该位图的左上角的 8 行 8 列的像素来创建填充模式。当应用选择一个两种颜色的 DIB 模式到一个黑白的设备上下文中时，Windows 将忽略 DIB 中指定的颜色，而用该设备上下文中当前文本的颜色和背景色显示模式画刷。DIB 中对应于第一种颜色的像素（在 DIB 颜色表偏移为 0 处）将采用文本的颜色显示；对应于第二种颜色的像素（在 DIB 颜色表偏移为 1 处）将采用背景色显示。要了解有关以下 Windows 函数的更详细的信息，请参阅 SDK 联机文档。 ::CreateDIBPatternBrush()（本函数是为了兼容 Windows 3.0 之前的应用而提供的） ::CreateDIBPatternBrushPt()（本函数只用于基于 Win32 的应用）

BOOL CreateHatchBrush(int nIndex, COLORREF crColor);

函数的返回值、参数及说明如表 9.7 所示。

表 9.7　　Create Hatch Brush()函数的返回值、参数及说明

返回值	调用成功时返回非 0 值，否则为 0
参　数	nIndex 指定画刷的阴影线风格。可取的值如下： HS_BDIAGONAL　45° 的向下阴影线（从左到右） HS_CROSS　水平和垂直方向以网格线做出阴影 HS_DIAGCROSS　45° 的网格线阴影

续表

参　数	HS_FDIAGONAL　45° 的向上阴影线（从左到右） HS_HORIZONTAL 水平的阴影线 HS_VERTICAL 垂直的阴影线 crColor 指定画刷的前景色（RGB 形式的值），就是阴影的颜色。要了解更详细的信息，请参阅 SDK 联机文档中关于 COLORREF 的描述
说　明	本函数用指定的阴影模式和颜色初始化画刷。此画刷随后该画刷就可以被选为任何设备上下文的当前画刷了

BOOL CreatePatternBrush(CBitmap* pBitmap);

函数返回值、参数及说明如表 9.8 所示。

表 9.8　　Create Pattern Brush()函数返回值、参数及说明

返回值	调用成功时返回非 0 值，否则为 0
参　数	pBitmap 指定一个位图。说明本函数用位图指定的模式初始化画刷。此画刷随后就可用于任何支持光栅操作的设备上下文。由 PBitmap 指定的位图一般用以下的函数初始化： CBitmap:: CreateBitmap() CBitmap::CreateBitmapIndirect() CBitmap::LoadBitmap() Cbitmap:: CreateCompatibleBitmap()
说　明	用作填充模式的位图应该是 8 像素×8 像素的。如果位图太大，Windows 将用该位图的左上角的 8 行 8 列的像素来创建填充模式。删除模式画刷时不会影响相关的位图，这意味着可以用一个位图来创建任意个模式画刷。采用黑白位图创建的画刷使用当前的文本颜色和背景色。位值置为 0 的像素点将用当前的文本颜色绘制，位值置为 1 的像素点将用当前的背景色绘制。要了解有关使用 Windows 函数::CreatePatternBrush()的更详细的信息，请参阅 SDK 联机文档

BOOL CreateSolidBrush(COLORREF crColor);

函数返回值、参数及说明如表 9.9 所示。

表 9.9　　Create Solid Brush()函数返回值、参数及说明

返回值	调用成功时返回非 0 值，否则为 0
参　数	crColor 指定画刷颜色的一个 COLORREF 结构。该颜色指定了一个 RGB 颜色值，可以用头文件 WINDOWS.h 中的宏 RGB 生成
说　明	本函数用指定的颜色初始化画刷。随后该画刷就可用作任何设备上下文的当前画刷。当应用使用完 CreateSolidBrush()创建的画刷之后，应该从设备上下文中移出该画刷

BOOL CreateSysColorBrush(int nIndex);

函数返回值、参数及说明如表 9.10 所示。

表 9.10　　Create Sys Color Brush()函数返回值、参数及说明

返回值	调用成功时返回非 0 值，否则为 0
参　数	nIndex 指定阴影线画刷的风格。可能的取值如下： HS_BDIAGONAL　45° 的向下阴影线（从左到右） HS_CROSS 水平和垂直方向以网格线做出阴影 HS_DIAGCROSS　45° 的网格线阴影 HS_FDIAGONAL　45° 的向上阴影线（从左到右） HS_HORIZONTAL 水平的阴影线 HS_VERTICAL 垂直的阴影线
说　明	本函数初始化画刷的颜色。随后该画刷就可用作任何设备上下文的当前画刷。当应用使用完 CreateSysColorBrush()创建的画刷之后，应该从设备上下文中移出该画刷

3. 操作

static CBrush* PASCAL FromHandle(HBRUSH hBrush);

函数返回值、参数及说明如表 9.11 所示。

表 9.11　　FromHandle()函数返回值、参数及说明

返回值	调用成功时返回一个指向 CBrush 对象的指针，否则返回 NULL
参　数	hBrush 指向一个 Windows GDI 画刷的句柄
说　明	本函数返回一个指向 CBrush 对象的指针，所需参数为一个指向一个 Windows GDI 画刷的句柄。如果该 CBrush 对象没有应用到句柄上，则创建一个临时的 CBrush 对象并应用它。该临时 CBrush 对象保持有效，直到应用在它的事件循环中出现空闲时间，此时 Windows 会删除所有的临时图形对象。换句话说，临时对象仅在一个 Windows 消息的处理过程中有效。要了解有关使用图形对象的更详细的信息，请参阅 SDK 联机文档

4. 属性

int GetLogBrush(LOGBRUSH* pLogBrush);

函数返回值、参数及说明如表 9.12 所示。

表 9.12　　GetLogBrush()函数返回值、参数及说明

返回值	调用成功且参数 pLogBrush 有效时，返回存到缓冲区中的字节数；调用成功但参数 pLogBrush 为 NULL 时，返回为了保存数据到缓冲区所需的字节数；否则返回 0
参　数	pLogBrush 指向一个 LOGBRUSH 结构的指针，结构中包含了画刷的有关信息
说　明	本函数用于检索 LOGBRUSH 结构中的信息。该结构定义了画刷的风格、颜色和模式。例如，调用本函数来取得某个位图的颜色和模式

operator HBRUSH( ) const;

函数返回值、参数及说明如表 9.13 所示。

表 9.13　　HBRUSH()函数返回值、参数及说明

返回值	调用成功时返回一个指向 Windows GDI 对象的句柄，该句柄表示一个 CBrush 对象，否则返回 NULL
参　数	无
说　明	本操作符用于取得某个 CBrush 对象上的 Windows GDI 对象句柄。这是一个校验性操作符，可直接参考 HBRUSH 对象。要了解有关使用图形对象的更详细的信息，请参阅 SDK 联机文档

9.5.2　CBrush 类简单画刷的实现

现在开始设计一个利用黄色画刷填充鼠标绘制的矩形区域的小程序。通过这个简单的小程序，看一下画刷的作用和实现方法。

首先，需要初始化一个画刷对象，这里使用 Crush 类的构造函数 CBrush(COLORREF crColor)构造一个画刷对象。

对于矩形的绘制，可以使用 CDC 提供的填充矩形的函数 CDC::FillRect(LPCRECT lpRect, CBrush* pBrush)来实现。参数 lpRect 可以使用一个 CRect 结构，该结构的一个构造函数是包含一个矩形的开始点和结束点。开始点的位置同样通过设置一个全局变量 m_point 来保存鼠标左键按下的位置来实现。矩形的结束点可以通过捕获释放鼠标松开的消息来实现。具体的实现过程可以参见 9.4.2 小节，这里不再赘述。pBrush 即为所创建的画刷对象。

该程序的具体代码如下：

```
void CUseCBrushView::OnLButtonUp(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CBitmap bitmap;
```

```
        CBrush brush(RGB(255,255,0));
        //生成一个 CClientDC 类的对象，使用 CUseCBrushView 对象的指针进行初始化
        CClientDC ccdc(this);
        //利用黄色画刷填充鼠标拖动而成的矩形区域
        ccdc.FillRect(CRect(m_point,point),&brush);
        CView::OnLButtonUp(nFlags, point);
    }
```

程序运行效果如图 9.9 所示。

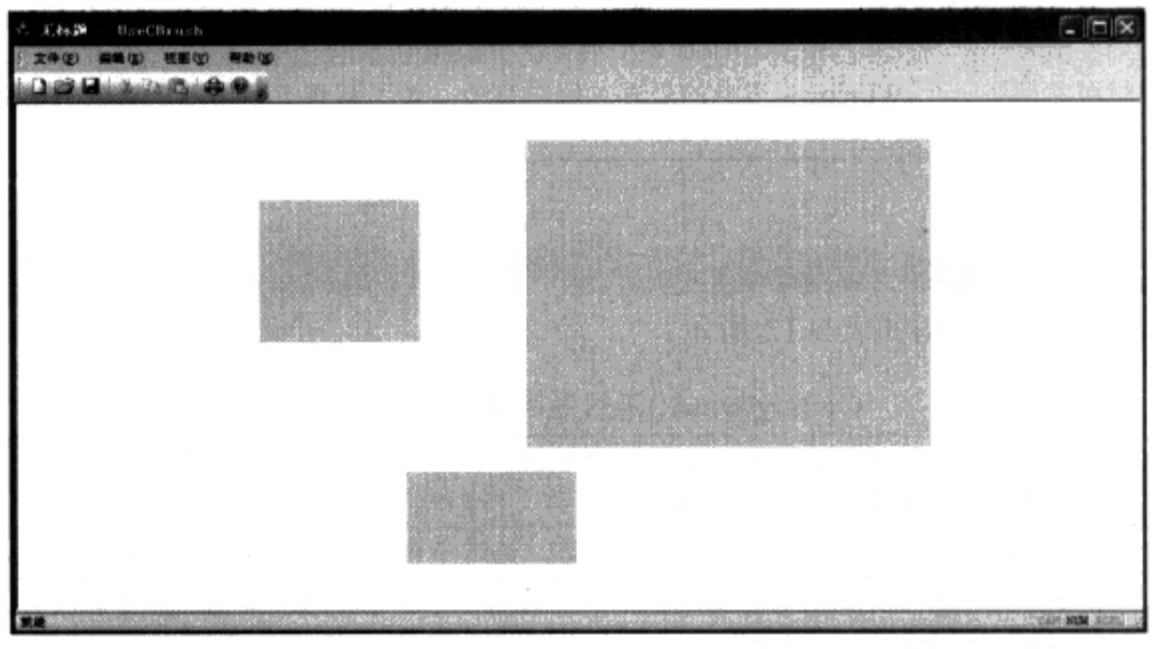


图 9.9　简单画刷功能的实现

9.5.3　CBrush 类位图画刷的实现

使用画刷，不仅可以用指定的颜色绘制填充一块区域，还可以使用一幅位图来填充指定的区域。实现这一功能，只需要使用 CBrush 类的构造函数 CBrush(CBitmap* pBitmap)。

该构造函数的参数是一个位图对象的指针。这里，介绍一个通过 MFC 提供的加载资源的方法来获得一个位图对象的指针。

首先单击 Visual C++ 2010 主窗口左侧 Solution Explorer 页面的“Resource Files”文件夹，在弹出的文件夹内容中双击 UseCBrush.rc，即可进入资源菜单；然后右击资源菜单中的任意一个文件夹，选择“Add Resource”命令，会弹出如图 9.10 所示的对话框。

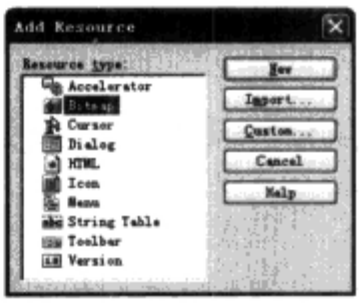


图 9.10　资源对话框

选择“Bitmap”选项，然后单击“New”按钮，即可生成一个默认名为 bitmap1.bmp 的位图资源，并进入位图编辑界面，如图 9.11 所示。现在，用户就可以使用左上角的 Visual C++ 2010 提供的位图编辑工具编辑位图了。

保存好编辑后的位图资源，并且记录好该位图资源的资源 ID。创建的位图资源的 ID 为默认的 IDB_BITMAP1，该资源 ID 可以让用户自由更改。

有了位图文件之后，使用位图类 CBitmap 的 LoadBitmap()函数来加载位图，这样，就可以获得一个位图对象的指针来初始化画刷对象了。

具体的实现过程如下：

```
void CUseCBrushView::OnLButtonUp(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CBitmap bitmap;
```

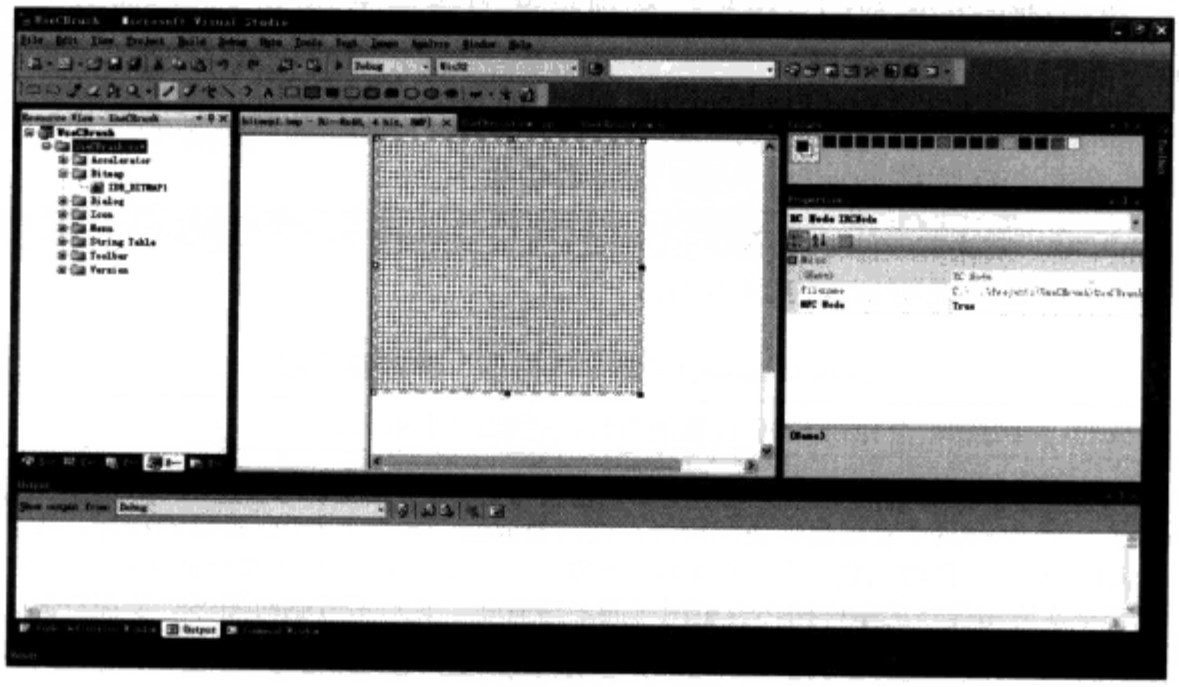


图 9.11　Visual Studio 2010 位图编辑工具编辑位图

```
    bitmap.LoadBitmap(IDB_BITMAP1);
    CBrush brush(&bitmap);
                                    //生成一个 CClientDC 类的对象，使用 UseCBrushView 对象的指针进行初始化
    CClientDC ccdc(this);
                                    //利用位图画刷填充鼠标拖动而成的矩形区域
    ccdc.FillRect(CRect(m_point,point),&brush);
    CView::OnLButtonUp(nFlags, point);
}
```

程序的运行结果如图 9.12 所示。

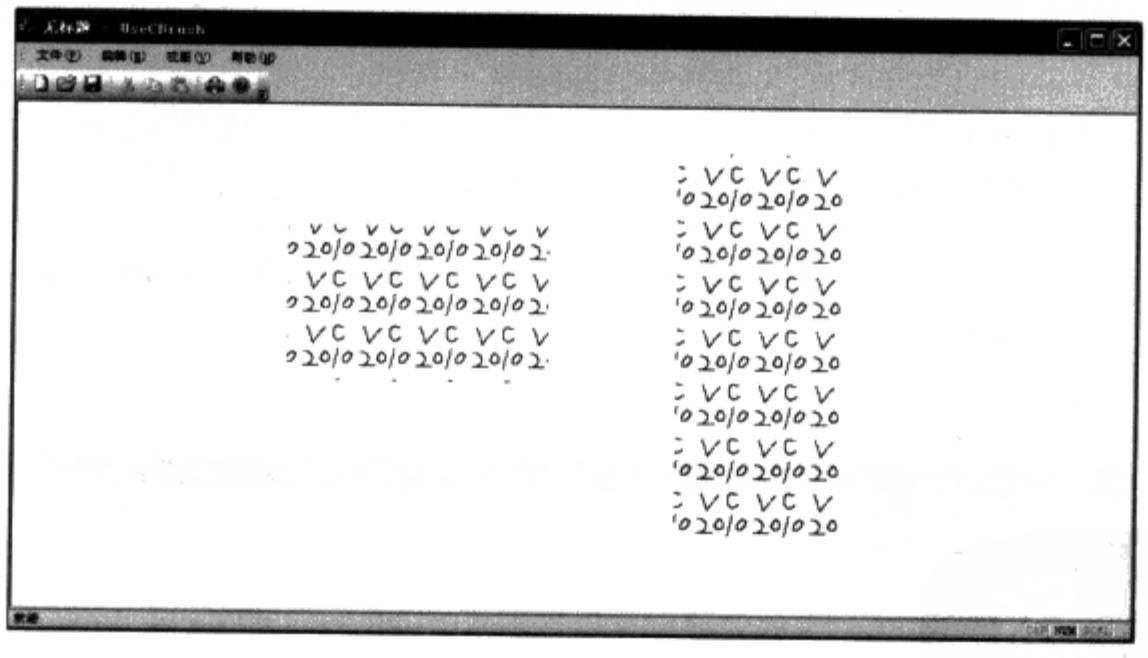


图 9.12　位图画刷功能的实现

9.5.4　透明画刷的实现

前面实现的两种画刷都是有填充效果的。如果想要绘制一些内部不填充任何内容，而是透明的，可以做到吗？

答案是肯定的，下面就来实现透明画刷。

由于 CBrush 类中没有提供创建透明画刷的方法，这里通过 GetStockObject()函数来获得一个透明

画刷的句柄，GetStockObject()函数的作用是检索预定义的备用笔、刷子、字体或者调色板的句柄。将该函数的参数取为 NULL_BRUSH，就可以获得一个空画刷，即所需要的透明画刷。在这里，使用GetStockObject()函数获取的是一个画刷句柄，而进行绘制操作时需要一个画刷对象，因此，还必须通过 FromHandle()函数来实现这样的功能，该函数的声明如下：

```
static CDC* PASCAL FromHandle(HDC hDC );
```

函数的返回值、参数及说明如表 9.14 所示。

表 9.14　　From Handle()函数的返回值、参数及说明

返回值	如果成功，则返回一个指向 CDC 对象的指针；否则返回 NULL
参　数	Windows 设备描述表句柄
说　明	当给予一个 Windows 设备描述表句柄时，此函数返回指向一个 CDC 对象的指针。如果此句柄没有连接一个 CDC 对象，则将创建一个临时 CDC 对象，并将此 CDC 对象连接到此句柄上。这个临时 CDC 对象只能在下次应用程序的事件循环有空闲时间之前有效，之后所有临时图形对象都将被删除。也就是说，临时对象仅在处理一条 Windows 消息的期间有效。

具体实现代码如下：

```
void CUseNullCBrushView::OnLButtonUp(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
                                        //创建并获得设备描述表
    CClientDC dc(this);
                                        //创建一个透明画刷
    CBrush *pBrush = CBrush::FromHandle((HBRUSH)GetStockObject(NULL_BRUSH));
                                        //将透明画刷选入设备描述表
    CBrush *pOldBrush =dc.SelectObject(pBrush);
                                        //使用透明画刷绘制矩形
    dc.Rectangle(CRect(m_point,point));
                                        //恢复先前的画刷
    dc.SelectObject(pOldBrush);
    CView::OnLButtonUp(nFlags, point);
}
```

注意，这里绘制矩形使用的是 Rectangle()函数，在使用前需要将使用的画刷选入设备描述表中，然后再调用该函数绘制矩形。

程序的运行结果如图 9.13 所示。

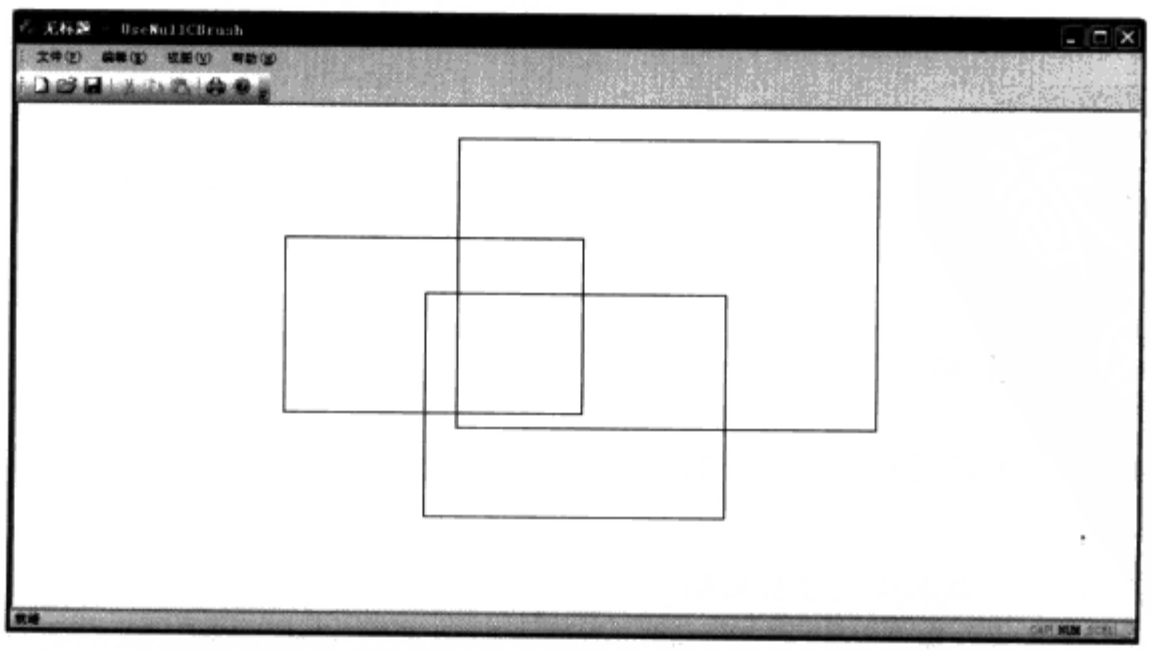


图 9.13　实现效果

9.6　小结

本章详细介绍了 Visual C++ 2010 MFC 图形图像编程的原理，以及如何利用画笔和画刷绘制简单的图形。本章的重点是。

- 设备描述表是 Windows 定义的一种数据结构，它包含的信息允许 Windows 将输出请求转换为对正在使用的特定物理输出设备的动作。输出请求采用设备无关性 GDI 函数调用的形式。
- 用户绘图前必须通过 Windows API 提供的相应方法获得设备描述表句柄（HDC）。
- 绘图操作一般都是在窗口的客户区进行的。Windows 为一个窗口的客户区提供了一种坐标系统，它对于这个窗口来说是本地的。
- 设备环境提供了一种称为映射模式的可选坐标系统，它将被自动转换为客户坐标。
- Windows 图形设备接口（GDI）是为与设备无关的图形设计的。它接受 Windows 应用程序的绘图请求（表现为 GDI 函数调用），并将它们传给相应的设备驱动程序，完成到特定硬件的输出。
- MFC 中的画笔工具，是通过 CPen 类提供的。GDI 与 CPen 相结合，能实现很多复杂的绘图功能。
- MFC 中的画刷工具，是通过 CBrush 类提供的。GDI 与 CPen 相结合，能实现很多复杂的绘图功能。

第 10 章　Visual C++ 2010 MFC 动态函数链接库

本章主要介绍了如何使用 Visual C++ 2010 MFC 实现对动态函数链接库的操作，以及 HOOK 技术的相关知识。在动态链接库部分中，本章首先介绍了动态链接库的概念、优势和原理等；接着说明了调用动态函数链接库的方法；然后又说明了动态链接库的框架；最后使用一些例子详细介绍了如何创建各种类型的动态链接库。在 HOOK 技术部分，本章首先介绍了什么是 HOOK；然后说明了创建钩子的方法；最后分别介绍了简单的鼠标钩子和键盘钩子的创建和使用方法。完成本章的学习之后，读者将对 Visual C++ 2010 MFC 动态链接库操作有一定的了解，并且能够在编程中使用自己创建的动态链接库实现不同的功能；同时还可以实现带有自己设定的钩子的程序。

本章主要内容有：

- 动态函数链接库简介；
- 调用动态函数链接库；
- DLL 的框架简介；
- 创建 MFC DLL 范例；
- 创建 Win32 DLL 范例；
- 创建资源 DLL 范例；
- HOOK 技术。

10.1　动态函数链接库简介

10.1.1　什么是动态函数链接库

动态函数链接库，通常称为 DLL，是 Dynamic Linked Library 的缩写。

DLL 和 EXE 文件一样，其中包含的也是程序的二进制执行代码和程序所需的资源（比如图标、对话框、字符串等），可是为什么要把代码放在 DLL 中，而不是做成 EXE 文件呢？其实 DLL 中的代码是以 API 函数形式出现的，通俗地讲，DLL 中包含的程序代码都被做成了一个个小模块，应用程序通过按下所需 DLL 中特定的按钮，来调用 DLL 中这个按钮所代表的功能。在使用记事本等程序时，如果要保存文件或打开文件，就会弹出通用文件对话框，选择文件位置，这就调用了系统底层 DLL 中的通用对话框界面。

Windows 系统中有 3 个非常重要的底层 DLL：Kernel32.dll、User32.dll、GDI32.dll。其中 Kernel32.dll 含有内核相关的功能，主要包含用于管理内存、进程和线程的函数；而 User32.dll 中包含的则是用于执行用户界面任务的函数，比如把用户的鼠标单击操作传递给窗口，以便窗口根据用户的单击来执行预定的事件；GDI32.dll 的名称使用了英文缩写，全称是 Graphical Device Interface（图形设备接口），包含用于画图和显示文本的函数，比如要显示一个程序窗口，就调用其中的函数来画这个窗口。

下面简单介绍 Windows 操作系统中的一些作为 DLL 实现的文件。

- DLL（.dll）文件：这是最常见的动态函数链接库的文件格式。
- ActiveX 控件（.ocx）文件：ActiveX 控件的一个示例是日历控件，它使用户可以从日历中选择日期。
- 控制面板（.cpl）文件：控制面板文件的一个示例是位于控制面板中的项，每个项都是一个专用 DLL。
- 设备驱动程序（.drv）文件：设备驱动程序的一个示例是控制打印到打印机的打印机驱动程序。

10.1.2　动态函数链接库的优点

前面只介绍了 DLL 将程序代码封装成函数的原理和 Windows 操作系统中 DLL 的存在形式。为什么封装成函数，就能成为系统大量使用 DLL 的理由呢？这是因为使用 DLL 有以下优点。

1．扩展应用程序

由于 DLL 能被应用程序动态载入内存，所以，应用程序可以在需要时才将 DLL 载入内存中；同时这让程序的可维护性变得很高。例如 QQ 的视频功能需要升级，那么负责编写 QQ 的程序员不必将 QQ 所有代码都重写，而只需将视频功能相关的 DLL 文件重写即可。

2．便于程序员合作

这个和最终用户关系不大，但是却和项目开发人员紧密相关。大家都知道编程工具有很多，如 Visual Basic、Visual C++、Delphi 等，如果多个人合作编写一个大的程序，那么可能有的人用 Visual Basic，有的人用 Visual C++，每人负责的部分所使用的编程语言都不同，究竟放在哪个编译器中进行编译呢？这就好比一群来自各个国家的人在共同写一篇文章，如果他们所使用的语言都不同，写出来的文章怎么可能凑到一起呢？而有了 DLL 后，可以让 Visual C++程序员写一个 DLL，然后 Visual Basic 程序员在程序中调用，无须为怎么将它们都编译为一个单独的 EXE 文件而发愁了。

3．节省内存

如果多个应用程序调用的是同一个动态链接库，那么这个 DLL 文件不会被重复多次装入内存中，而是由这些应用程序共享同一个已载入内存的 DLL。就好比一个办公室中，很少会为每一个员工都配置一台饮水机，而是在一个公共位置放上一台饮水机，所有需要喝水的职员都可以共用这台饮水机，既降低了成本，又节约了空间。

4．共享程序资源

包括前面提到的通用文件对话框在内，DLL 文件提供了应用程序间共享资源的可能。资源可以是程序对话框、字符串、图标或者声音文件等。

5．解决应用程序本地化问题

在下载某个程序的汉化包后，打开汉化说明，经常可以看到用下载包中的 DLL 文件覆盖掉程序原来的 DLL，汉化就完成了。这些程序都是将执行代码和应用程序界面分开编写，所以汉化者只需简单地将其中和程序界面相关的 DLL 汉化并发布即可。

6．解决代码的保密性问题

如果有些核心代码有很高的保密性要求，可以把它封装为一个动态函数链接库，同时提供函数接口给程序员使用，这样就可以解决代码的保密性问题。

10.1.3 动态函数链接库的起源

动态链接库（DLL）是从 C 语言函数库和 Pascal 库单元的概念发展而来的。所有的 C 语言标准库函数都存放在某一函数库中，同时用户也可以用 LIB 程序创建自己的函数库。在链接应用程序的过程中，链接器从库文件中复制程序调用的函数代码，并把这些函数代码添加到可执行文件中。

但是，随着 Windows 这样的多任务环境的出现，函数库的方法显得过于累赘。如果为了完成屏幕输出、消息处理、内存管理、对话框等操作，每个程序都不得不拥有自己的函数，那么 Windows 程序将变得非常庞大。Windows 的发展要求允许同时运行的几个程序共享一组函数的单一备份。动态链接库就是在这种情况下出现的。动态链接库不必重复编译或链接，一旦装入内存，DLL 函数就可以被系统中的任何正在运行的应用程序软件所使用，而不必再将 DLL 函数的另一个备份装入内存。

10.1.4 动态函数链接库的原理

“动态链接”这几个字指明了 DLL 是如何工作的。对于常规的函数库，链接器从中复制它需要的所有库函数，并把确切的函数地址传送给调用这些函数的程序。而对于 DLL，函数存储在一个独立的动态链接库文件中。在创建 Windows 程序时，链接过程并不把 DLL 文件链接到程序上。直到程序运行并调用一个 DLL 中的函数时，该程序才要求这个函数的地址。此时 Windows 才在 DLL 中寻找被调用函数，并将其地址传送给调用程序。采用这种方法，DLL 达到了复用代码的极限。

动态链接库的另一个方便之处是对动态链接库中函数的修改可以自动传播到所有调用它的程序中，而不必对程序作任何改动或处理。DLL 不仅提供了函数重用的机制，而且提供了数据共享的机制。任何应用程序都可以共享由装入内存的 DLL 管理的内存资源块。只包含共享数据的 DLL 称为资源文件，如 Windows 的字体文件等。

10.2 调用动态函数链接库

前面讲了DLL的基本知识，但是如何调用动态函数链接库呢？有两种调用动态函数链接库的方式，分别是静态链接和动态链接，一般也称为隐式调用和显式调用。

假设已经编写好了一个动态函数链接库，该 DLL 的功能是计算参数的阶乘，关于如何创建这个 DLL，请参见 10.5 节的内容。

下面分别展示一下如何使用静态链接和动态链接来调用这个 DLL。

10.2.1 静态链接

（1）新建一个 Dialog based MFC 应用程序，命名为 StaticLinkDemo，如图 10.1 所示。

（2）在对话框中添加一个文本框和两个按钮，如图 10.2 所示。

（3）为文本框添加 int 类型的全局变量 m_num。

（4）双击第一个按钮，添加如下事件代码：

```
#include "../factorial/factorial.h"
#pragma comment(lib, "factorial.lib")
void CStaticLinkDemoDlg::OnBnClickedButton1()
{
```

```
    // TODO: Add your control notification handler code here
    UpdateData();
    CString s;
    int f;
    f=factorial(m_num);
    s.Format(L"%d",f);
    MessageBox(s);
}
```

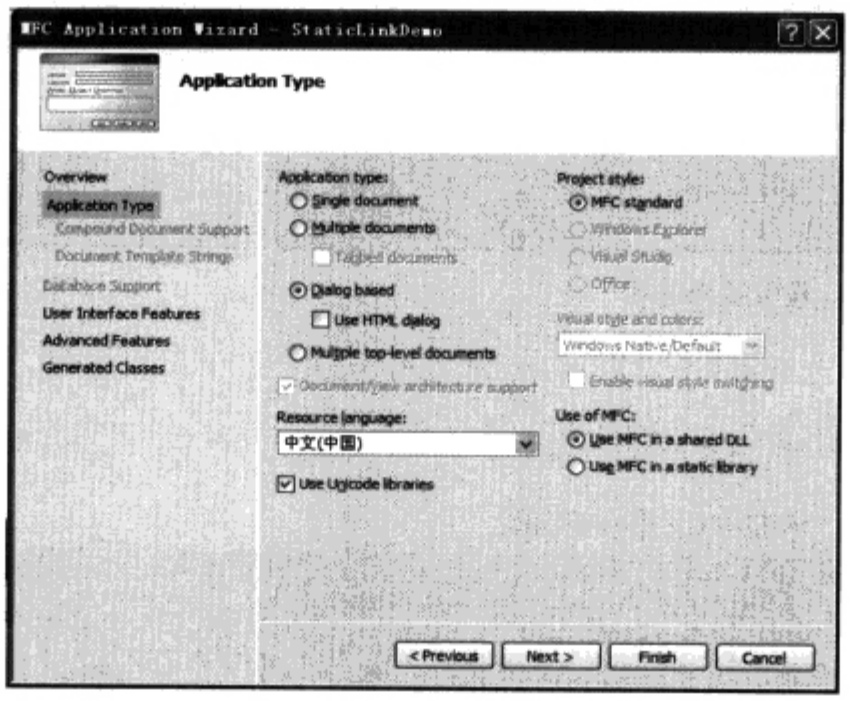


图 10.1　新建 Dialog Based MFC 应用程序对话框

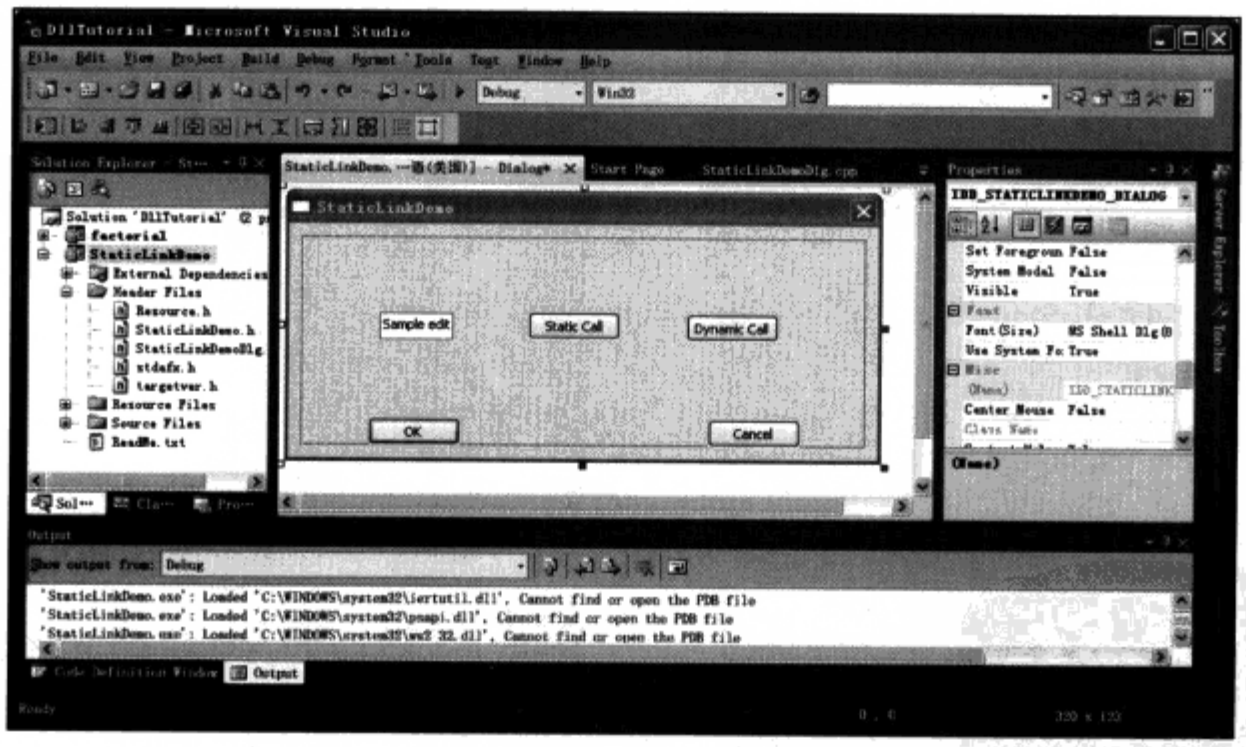


图 10.2　添加一个文本框和两个按钮

（5）设置编译选项，添加额外的 Library 目录，如图 10.3 所示。

（6）编译并运行程序，会显示如图 10.4 所示的结果。

静态链接也叫做加载时动态链接。在加载时动态链接中，应用程序像调用本地函数一样对导出的 DLL 函数进行显式调用。使用加载时动态链接，需要在编译和链接应用程序时提供头文件（.h）和导入库文件（.lib）。这样做时，链接器将向系统提供加载 DLL 所需的信息，并在加载时解析导出的 DLL 函数的位置。

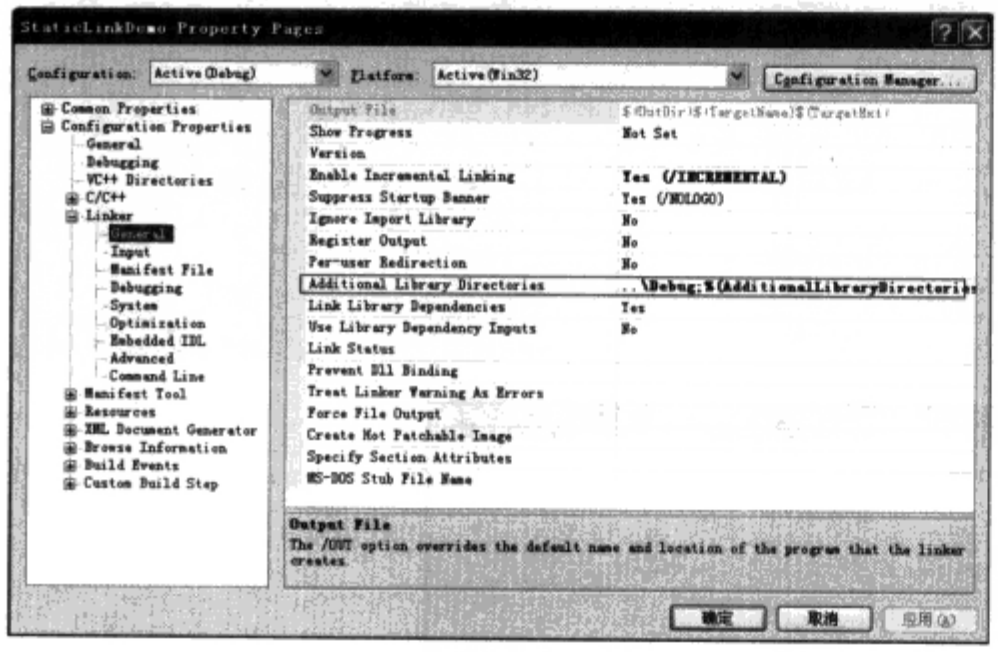


图 10.3　设置编译选项

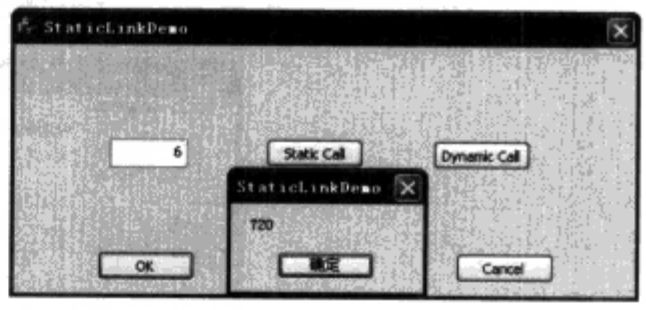


图 10.4　编译运行的结果

10.2.2　动态链接

（1）双击第二个按钮，添加如下事件代码：

```
typedef int (*FactorialFunc)(int);
void CStaticLinkDemoDlg::OnBnClickedButton2()
{
    // TODO: Add your control notification handler code here
    UpdateData();
    FactorialFunc _FactorialFunc;
    HINSTANCE hInstLibrary = LoadLibrary(L"factorial.dll");
    if (hInstLibrary == NULL)
    {
        FreeLibrary(hInstLibrary);
    }
    _FactorialFunc = (FactorialFunc)GetProcAddress(hInstLibrary, "factorial");
    if (_FactorialFunc == NULL)
    {
        FreeLibrary(hInstLibrary);
    }
    CString s;
    int f;
    f=_FactorialFunc(m_num);
    s.Format(L"%d",f);
    MessageBox(s);
    FreeLibrary(hInstLibrary);
}
```

（2）编译并运行程序，将显示如图 10.5 所示的结果。

在运行时动态链接中，应用程序调用 LoadLibrary()函数或 LoadLibraryEx()函数实现在运行时加载 DLL。成功加载 DLL 后，可以使用 GetProcAddress()函数获得要调用的导出的 DLL 函数的地址。在使用运行时动态链接时，无须使用导入库文件。

该如何决定是使用加载时链接还是运行时链接呢？可以从以下几个方面考虑。

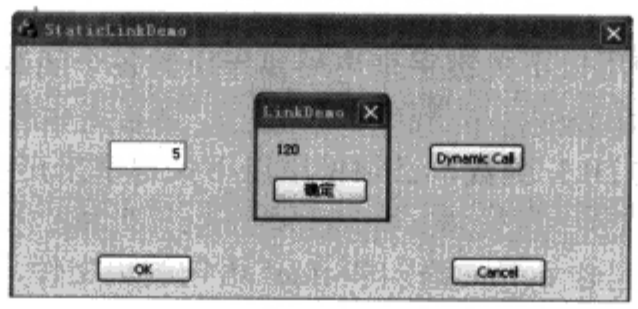


图 10.5　动态链接的编译运行结果

- 启动性能：如果应用程序的初始启动性能很重要，则应使用运行时动态链接。
- 易用性：在加载时动态链接中，导出的 DLL 函数类似于本地函数，这使开发人员可以方便地调用这些函数。
- 应用程序逻辑：在运行时动态链接中，应用程序可以分支，以便开发人员按照需要加载不同的模块。在开发多语言版本时，这一点很重要。

10.3　DLL 的框架简介

10.3.1　DllMain()函数简介

在创建 DLL 时，可以有选择地指定入口点函数。当进程或线程将它们自身附加到 DLL 或者将它们自身从 DLL 分离时，将调用入口点函数。开发人员可以使用入口点函数根据 DLL 的需要来初始化数据结构或者销毁数据结构；此外，如果应用程序是多线程的，则可以在入口点函数中使用线程本地存储（TLS）来分配各个线程专用的内存。下面的代码是一个 DLL 入口点函数的示例：

```
#include "stdafx.h"
BOOL APIENTRY DllMain(HMODULE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved)
{
    switch(ul_reason_for_call)
    {
        case DLL_PROCESS_DETACH;
        case DLL_THREAD_ATTACH;
        case DLL_THREAD_DETACH;
        case DLL_PROCESS_ATTACH;
        break;
    }
    return True;
}
```

当入口点函数返回 False 时，如果使用的是加载时动态链接，则应用程序不启动；如果使用的是运行时动态链接，则只有个别 DLL 不会加载。

入口点函数只应执行简单的初始化任务，不应调用任何其他 DLL 加载函数或终止函数。例如，在入口点函数中，不应直接或间接调用 LoadLibrary()函数或 LoadLibraryEx()函数。此外，入口点函数不应在进程终止时调用 FreeLibrary()函数。

10.3.2　DLL 的导出函数

要导出 DLL 函数，可以向导出的 DLL 函数中添加函数关键字，也可以创建模块定义文件（.def）

以列出导出的 DLL 函数。

要使用函数关键字，必须使用以下关键字来声明要导出的各个函数：

```
__declspec ( dllexport )
```

要在应用程序中使用导出的 DLL 函数，必须使用以下关键字来声明要导入的各个函数：

```
__declspec ( dllimport )
```

通常情况下，最好使用一个包含 define 语句和 ifdef 语句的头文件，以便分隔导出语句和导入语句。

开发人员还可以使用模块定义文件来声明导出的 DLL 函数。当使用模块定义文件时，不必向导出的 DLL 函数中添加函数关键字。在模块定义文件中，开发人员可以声明 DLL 的 LIBRARY 语句和 EXPORTS 语句。下面的代码是一个定义文件的示例：

```
//FactorialDLL.def
LIBRARY "FactorialDLL"
EXPORT
   factorial @1
```

10.4 创建 MFC DLL 范例

使用 MFC 可以生成两种类型的 DLL：MFC 扩展 DLL 和常规 DLL。常规 DLL 又可以分为动态链接和静态链接。

1. MFC 扩展 DLL

每个 DLL 都有某种类型的接口：变量、指针、函数、客户程序访问的类。它们的作用是让客户程序使用 DLL，MFC 扩展 DLL 可以有 C++的接口，也就是说，它可以导出 C++类给客户端。导出的函数可以使用 C++/MFC 数据类型作为参数或返回值，导出一个类时，客户端能创建类对象或者派生这个类。同时，在 DLL 中也可以使用 DLL 和 MFC。

Visual C++使用的MFC类库也保存在一个DLL中，MFC扩展DLL动态链接到MFC代码库的DLL，客户程序也必须要动态链接到 MFC 代码库的 DLL（这里谈到的两个 DLL，一个是开发人员自己编写的 DLL，一个是装载 MFC 类库的 DLL）。现在 MFC 代码库的 DLL 也存在多个版本，客户程序和扩展 DLL 都必须使用相同版本的 MFC 代码 DLL。为了让 MFC 扩展 DLL 能很好地工作，扩展 DLL 和客户程序都必须动态链接到 MFC 代码 DLL，而这个 DLL 必须在客户程序运行的计算机上存在。

2. 常规 DLL

使用 MFC 扩展 DLL 的一个问题就是 DLL 仅能和 MFC 客户程序一起工作，如果需要一个使用更广泛的 DLL，最好采用常规 DLL，因为它不受 MFC 的某些限制。常规 DLL 也有缺点：不能与客户程序发送指针或 MFC 派生类和对象的引用，即，常规 DLL 和客户程序的接口不能使用 MFC，但在 DLL 和客户程序的内部还是可以使用 MFC。

当在常规 DLL 的内部使用 MFC 代码库的 DLL 时，可以是动态链接/静态链接。如果是动态链接，也就是常规 DLL 需要的 MFC 代码没有构建到 DLL 中，这种情况有点和扩展 DLL 类似，在 DLL 运行的计算机上必须要有 MFC 代码库的 DLL。如果是静态链接，常规 DLL 中已经包含了需要的 MFC 代码，这样 DLL 的体积将比较大，但它可以在没有 MFC 代码 DLL 的计算机上正常运行。

下面以动态链接到 MFC 库的常规 DLL 来展示如何使用 MFC DLL。

10.4.1 建立 MFC DLL 工程

（1）启动 Visual Studio 2010，选择“File”→“New”→“Project”命令，弹出“New Project”对

话框，选择“MFC DLL”模板，输入项目名称和路径，单击“OK”按钮，如图 10.6 所示。

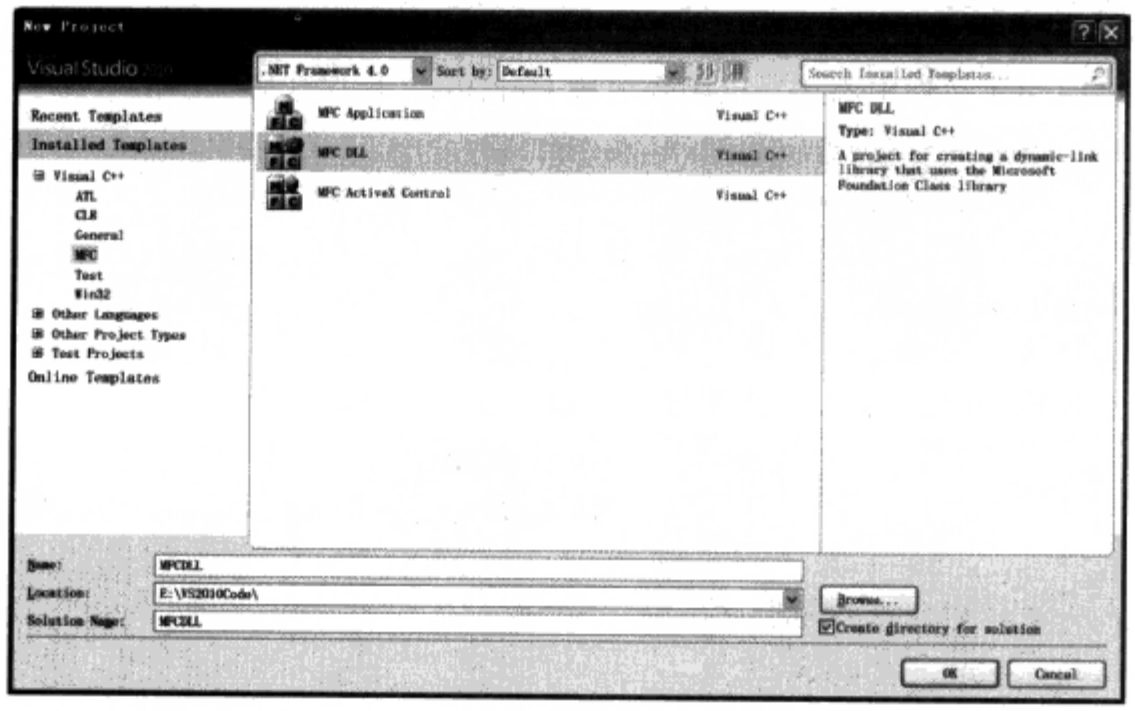


图 10.6　MFC DLL（1）

（2）打开如图 10.7 所示的对话框，单击“Next”按钮。

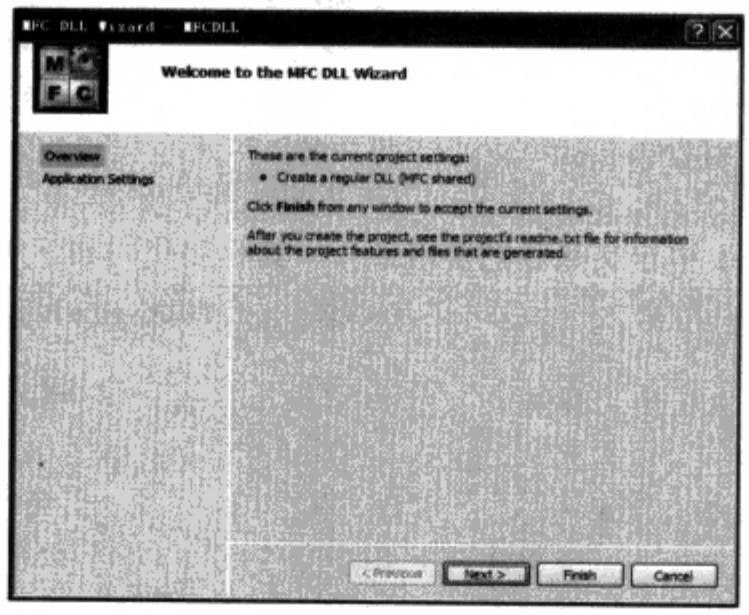


图 10.7　MFC DLL（2）

（3）在“DLL type”选项组中选中“Regular DLL using shared MFC DLL”单选按钮，然后单击“Finish”按钮，如图 10.8 所示。

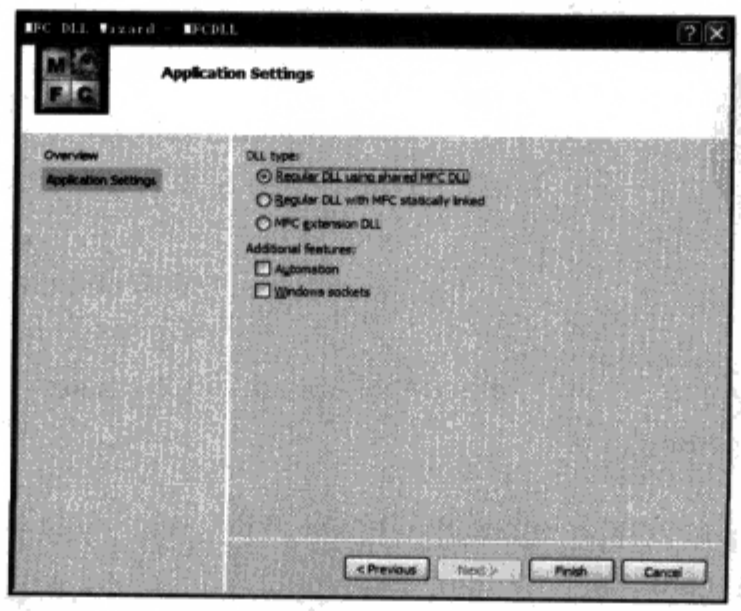


图 10.8　MFC DLL（3）

10.4.2 添加实现代码

（1）在 MFCDLL.def 文件中添加导出函数名称 HintBox，程序代码如下所示：

```
; MFCDLL.def : Declares the module parameters for the DLL.
LIBRARY        "MFCDLL"
EXPORTS
    ; Explicit exports can go here
    HintBox
```

（2）在头文件 MFCDLL.h 文件中声明导出函数，程序代码如下所示：

```
                                        // CMFCDLLApp
                                        // See MFCDLL.cpp for the implementation of this class
                                        //
extern "C" void PASCAL EXPORT HintBox();
class CMFCDLLApp : public CWinApp
{
public:
    CMFCDLLApp();
                                        // Overrides
public:
    virtual BOOL InitInstance();
    DECLARE_MESSAGE_MAP()
};
```

（3）在文件 MFCDLL.cpp 中定义函数的实现部分，其程序代码如下所示：

```
                                        // MFCDLL.cpp : Defines the initialization routines for the DLL.
#include "stdafx.h"
#include "MFCDLL.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#endif
                                        // CMFCDLLApp
BEGIN_MESSAGE_MAP(CMFCDLLApp, CWinApp)
END_MESSAGE_MAP()
                                        // CMFCDLLApp construction
CMFCDLLApp::CMFCDLLApp()
{
                                        // TODO: add construction code here,
                                        // Place all significant initialization in InitInstance
}
                                        // The one and only CMFCDLLApp object
CMFCDLLApp theApp;
                                        // CMFCDLLApp initialization
BOOL CMFCDLLApp::InitInstance()
{
    CWinApp::InitInstance();
    return TRUE;
}
```

```
extern "C" void PASCAL EXPORT HintBox()
{
    AfxMessageBox(L"您的工作时间太长了，请休息片刻下！^_^");
}
```

10.4.3　编译并调用

选择“Build ”→“Build MFC DLL”命令，等待编译完成。开发人员可以参照 10.2 节的调用方法编写程序调用该动态链接库，运行结果如图 10.9 所示。

图 10.9　调用动态链接库

10.5　创建 Win32 DLL 范例

Win32 DLL 也叫做 Non-MFC DLL（非 MFC 动态链接库），这种动态链接库没有采用 MFC 的类库结构，而是直接用 C/C++语言编写的 DLL，它的导出函数是标准的 C/C++接口，能被 MFC 和非 MFC 编写的应用程序调用。相对于采用 MFC 类库编写的 DLL，它占用很少的磁盘和内存空间。下面动手创建一个 Win32 DLL。

10.5.1　建立 Win32 工程

（1）启动 Visual Studio 2010 开发环境，选择“File”→“New”→“Project”命令，在弹出的对话框中选择“Win32 Project”模板，然后输入工程名称和工程路径，如图 10.10 所示，单击“OK”按钮。

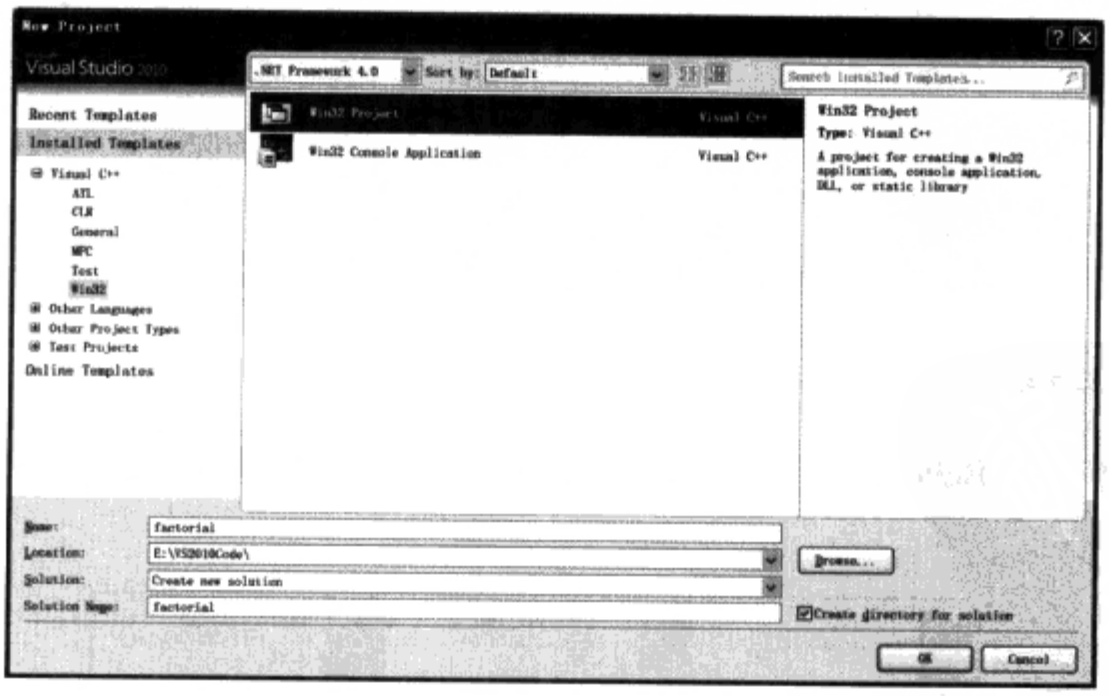


图 10.10　Win32 Project（1）

（2）打开如图 10.11 的对话框，单击“Next”按钮。

（3）在“Application type”选项组中选中“DLL”单选按钮，然后单击“Finish”按钮，如图 10.12 所示。

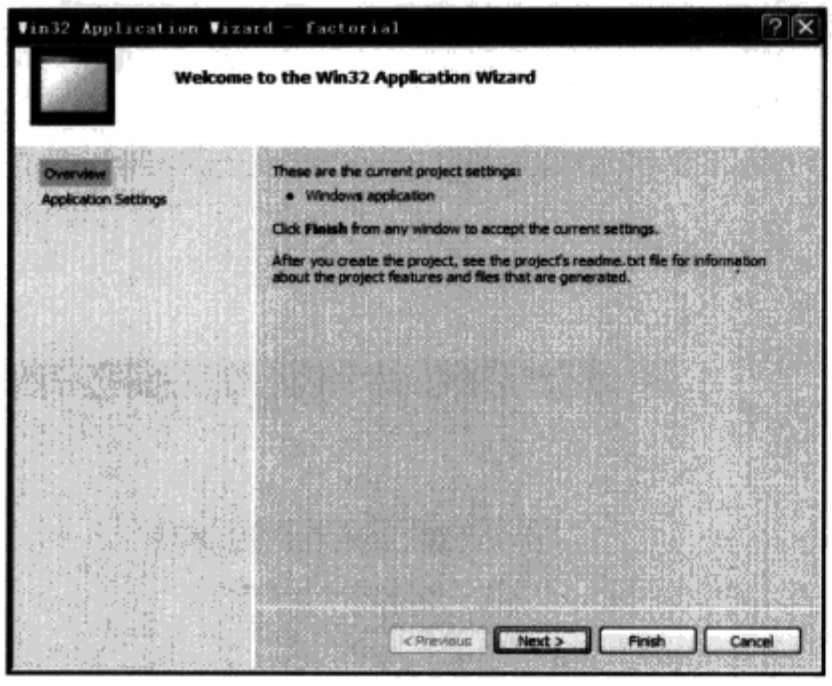


图 10.11　Win32 Project（2）

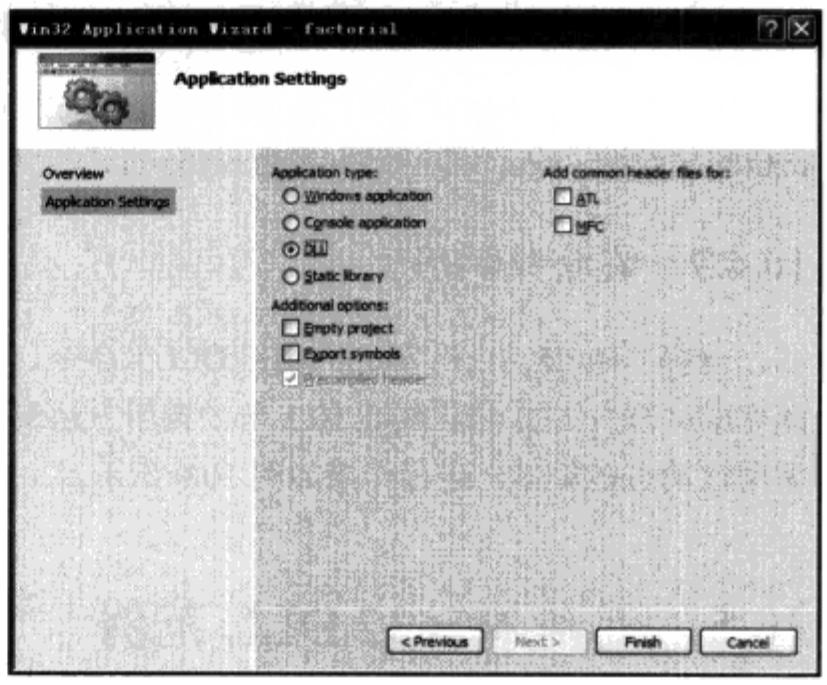


图 10.12　Win32 Project（3）

10.5.2　添加动态链接库代码

（1）首先添加头文件 factorial.h，并输入如下所示的代码：

```
// factorial.h : interface for the DLL application.
#ifndef   FACTORIAL_H
#define   FACTORIAL_H
extern "C" int __declspec(dllexport) factorial(int n);
#endif
```

（2）在文件 factorial.cpp 中添加如下所示的代码：

```
#include "stdafx.h"
#include "factorial.h"
int factorial(int n)
{
    int i,r=1;
    for(i=2;i<=n;i++)
        r*=i;
    return r;
}
```

10.5.3　编译工程

选择“Build”→“Build factorial”命令，等待编译完成，会在 Debug 文件夹下显示如图 10.13 所示的文件。

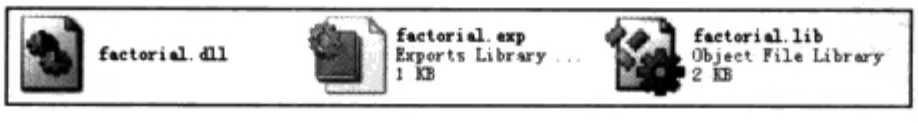


图 10.13　Win32 Project 的结果文件

其中 factorial.dll 文件是动态链接库，包含编译过的运行时二进制代码；factorial.exp 文件是指导出库文件的文件，简称导出库文件，其中包含导出函数和数据项的信息；factorial.lib 文件是静态链接库文件，包含导出函数的名称、参数、地址等信息，用于静态链接。

10.6　创建资源 DLL 范例

纯资源 DLL 是仅包含资源（如图标、位图、字符串和对话框）的 DLL，使用纯资源 DLL 是在多个程序之间共享同一组资源的好方法，同时也是针对多种语言进行本地化的好方法。

下面演示如何使用资源 DLL 来解决国际化问题，并实现一个可以切换中英文界面的软件。

10.6.1　建立 MFC Application 工程

（1）启动 Visual Studio 2010，选择“File”→“New”→“Project”命令，选择“MFC Application”选项，输入项目名称 DemoApp 和保存路径，单击“OK”按钮建立工程。

（2）在“Application type”选项组中选中“Single document”单选按钮，在“Project style”选项组中选中“MFC standard”单选按钮，单击“Finish”按钮。

（3）查看该工程的菜单资源，如图 10.14 所示。

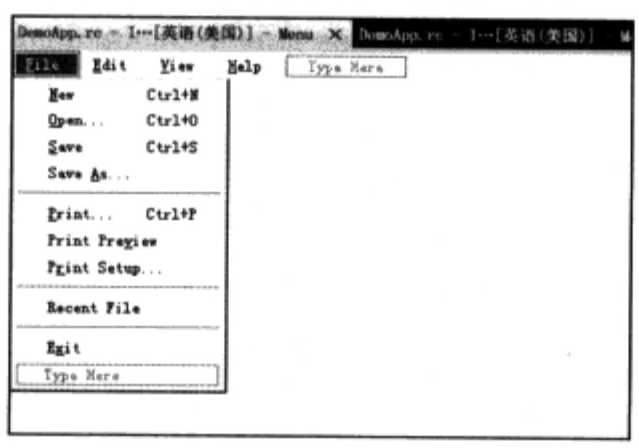


图 10.14　MFC Application 工程

10.6.2　建立中文资源 DLL

（1）新建 MFC DLL 项目，输入项目名称 ResDll 和路径，单击“OK”按钮，如图 10.15 所示。

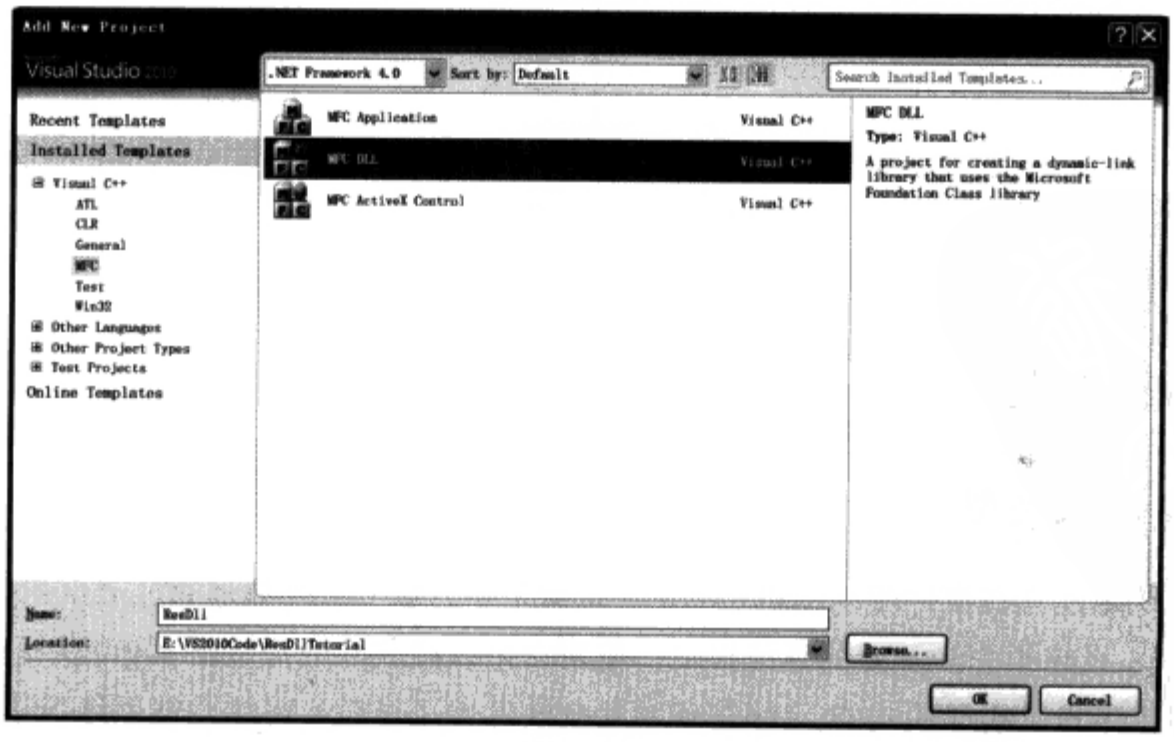


图 10.15　中文资源 DLL

（2）在打开的对话框中保持默认设置，直接单击“Finish”按钮。

（3）移除原来的资源文件 ResDll.rc、ResDll.rc2 和 Resource.h，将 DemoApp 工程的响应资源文件复制到相应目录下，并添加到该资源 DLL 工程中。将菜单修改为中文菜单，如图 10.16 所示。

（4）编译程序，生成 ResDll.dll 文件。

图 10.16　将英文菜单修改为中文菜单

10.6.3　加载资源 DLL

（1）修改 DemoApp 工程的初始化函数 CDemoAppApp::InitInstance()，添加如下所示的代码：

```
AfxEnableControlContainer();
//#define  _CHINESE   //若显示中文菜单，则去掉该语句前面的注释
#ifdef       _CHINESE
HINSTANCE m_hLangDLL;
m_hLangDLL=::LoadLibrary(L"ResDll.dll");
AfxSetResourceHandle(m_hLangDLL);
#endif
```

在应用程序中显式地调用这个 DLL，使用 LoadLibrary()函数装入资源 DLL，使用 FindResource()和 LoadResource()装入各种资源，或者使用下面特定的资源装入函数：

```
FormatMessage
LoadAccelerators
LoadBitmap
LoadCursor
LoadIcon
LoadMenu
LoadString
```

当资源使用结束时，应用程序必须调用 FreeLibrary()函数来释放资源。

（2）分别选择不同的语言，会有如图 10.17 所示的运行结果。

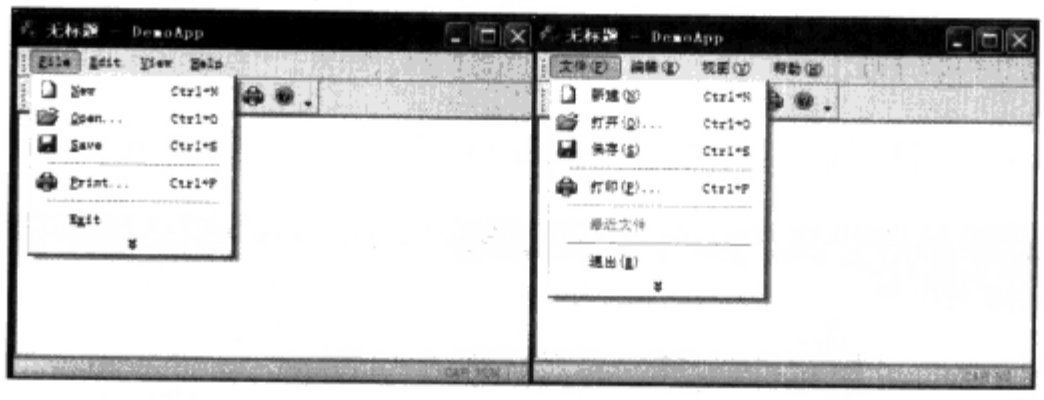


图 10.17　不同语言菜单的比较

10.7　Hook 技术

10.7.1　Hook 函数类型

Windows 系统是建立在事件驱动的机制上的，也就是说，整个系统都是通过消息的传递来实现的。而钩子是 Windows 系统中非常重要的系统接口，使用它可以截获并处理送给其他应用程序的消息，从而完成普通应用程序难以实现的功能。Hook 函数（也叫做钩子函数）可以监视系统或进程中的各种事件消息，截获发往目标窗口的消息并进行处理。这样，就可以在系统中安装自定义的钩子，监视系统

中特定事件的发生，完成特定的功能，比如截获键盘、鼠标输入，屏幕取词，日志监视等。可见，利用钩子可以实现许多特殊而有用的功能，因此，对于高级编程人员来说，掌握钩子的编程方法是很有必要的。

Hook 函数按事件分类，有如下的几种常用类型：

- 键盘钩子和低级键盘钩子可以监视各种键盘消息；
- 鼠标钩子和低级鼠标钩子可以监视各种鼠标消息；
- 外壳钩子可以监视各种 Shell 事件消息，比如启动和关闭应用程序；
- 日志钩子可以记录从系统消息队列中取出的各种事件消息；
- 窗口过程钩子监视所有从系统消息队列发往目标窗口的消息。

此外，还有一些特定事件的钩子提供给开发人员使用，这里不再一一列举。

下面描述常用的 Hook 类型。

1. WH_CALLWNDPROC 和 WH_CALLWNDPROCRET Hooks

WH_CALLWNDPROC 和 WH_CALLWNDPROCRET Hooks 使开发人员可以监视发送到窗口过程的消息。系统在消息发送到接收窗口过程之前调用 WH_CALLWNDPROC Hook 子程，并且在窗口过程处理完消息之后调用 WH_CALLWNDPROCRET Hook 子程。WH_CALLWNDPROCRET Hook 传递指针到 CWPRETSTRUCT 结构，再传递到 Hook 子程。CWPRETSTRUCT 结构包含来自处理消息的窗口过程的返回值，同样也包含与这个消息关联的消息参数。

2. WH_CBT Hook

在以下事件之前，系统都会调用 WH_CBT Hook 子程，这些事件包括：

- 激活、建立、销毁、最小化、最大化、移动、改变尺寸等窗口事件；
- 完成系统指令；
- 来自系统消息队列中的移动鼠标、键盘事件；
- 设置输入焦点事件；
- 同步系统消息队列事件。

Hook 子程的返回值确定系统是否允许或者防止这些操作中的一个发生。

3. WH_DEBUG Hook

在系统调用系统中与其他 Hook 关联的 Hook 子程之前，系统会先调用 WH_DEBUG Hook 子程。可以使用这个 Hook 来决定是否允许系统调用与其他 Hook 关联的 Hook 子程。

4. WH_FOREGROUNDIDLE Hook

当应用程序的前台线程处于空闲状态时，可以使用 WH_FOREGROUNDIDLE Hook 执行低优先级的任务。当应用程序的前台线程大概要变成空闲状态时，系统就会调用 WH_FOREGROUNDIDLE Hook 子程。

5. WH_GETMESSAGE Hook

应用程序使用 WH_GETMESSAGE Hook 来监视从 GetMessage()或 PeekMessage()函数返回的消息。可以使用 WH_GETMESSAGE Hook 去监视鼠标和键盘输入，以及其他发送到消息队列中的消息。

6. WH_JOURNALPLAYBACK Hook

WH_JOURNALPLAYBACK Hook 使应用程序可以插入消息到系统消息队列。可以使用这个 Hook 回放通过使用 WH_JOURNALRECORD Hook 记录下来的连续的鼠标和键盘事件。只要 WH_

JOURNALPLAYBACK Hook 已经安装，正常的鼠标和键盘事件就是无效的。WH_JOURNALPLAYBACK Hook 是全局 Hook，它不能像线程特定 Hook 一样使用。WH_JOURNALPLAYBACK Hook 返回超时值，这个值告诉系统在处理来自回放 Hook 当前消息之前需要等待多长时间（毫秒），这就使 Hook 可以控制实时事件的回放（按键精灵应该是用该 Hook 做的）。

7. WH_JOURNALRECORD Hook

WH_JOURNALRECORD Hook 用来监视和记录输入事件。可以使用这个 Hook 记录连续的鼠标和键盘事件，然后通过使用 WH_JOURNALPLAYBACK Hook 来回放。WH_JOURNALRECORD Hook 是全局 Hook，它不能像线程特定 Hook 一样使用。WH_JOURNALRECORD 是 System-wide Local Hooks。

8. WH_KEYBOARD Hook

在应用程序中，WH_KEYBOARD Hook 用来监视 WM_KEYDOWN 和 WM_KEYUP 消息，这些消息通过 GetMessage()函数或 PeekMessage function()返回。可以使用这个 Hook 来监视输入消息队列中的键盘消息。

9. WH_KEYBOARD_LL Hook

WH_KEYBOARD_LL Hook 监视输入线程消息队列中的键盘消息。

10. WH_MOUSE Hook

WH_MOUSE Hook 监视从 GetMessage()或者 PeekMessage()函数返回的鼠标消息。使用这个 Hook 可以监视输入消息队列中的鼠标消息。

11. WH_MOUSE_LL Hook

WH_MOUSE_LL Hook 监视输入线程消息队列中的鼠标消息。

12. WH_MSGFILTER 和 WH_SYSMSGFILTER Hooks

WH_MSGFILTER 和 WH_SYSMSGFILTER Hooks 可以监视菜单、滚动条、消息框、对话框消息并且发现用户使用【Alt+Tab】或【Alt+Esc】组合键切换窗口。WH_MSGFILTER Hook 只能监视传递到菜单、滚动条、消息框的消息，以及传递到通过安装了 Hook 子程的应用程序建立的对话框的消息。WH_SYSMSGFILTER Hook 监视所有应用程序消息。WH_MSGFILTER 和 WH_SYSMSGFILTER Hooks 可以在模式循环期间过滤消息，这等价于在主消息循环中过滤消息。通过调用 CallMsgFilter()可以直接调用 WH_MSGFILTER Hook。通过使用这个函数，应用程序能够在模式循环期间使用相同的代码去过滤消息，如同在主消息循环中一样。

13. WH_SHELL Hook

外壳应用程序可以使用 WH_SHELL Hook 去接收重要的通知。当外壳应用程序是激活的并且当顶层窗口建立或者销毁时，系统调用 WH_SHELL Hook 子程。

按照惯例，外壳应用程序都不接收 WH_SHELL 消息，所以在应用程序能够接收 WH_SHELL 消息之前，应用程序必须调用 SystemParametersInfo()注册它自己。

以上是 13 种常用的 Hook 类型。

按使用范围分类，主要有线程钩子和系统钩子。

（1）线程钩子监视指定线程的事件消息。

（2）系统钩子监视系统中的所有线程的事件消息。因为系统钩子会影响系统中所有的应用程序，所以钩子函数必须放在独立的动态链接库（DLL）中，这是系统钩子和线程钩子最大的不同。

这里有几点需要说明。

（1）如果对于同一个事件（如鼠标消息）既安装了线程钩子又安装了系统钩子，那么系统会自动先调用线程钩子，然后调用系统钩子。

（2）对同一个事件消息可安装多个钩子处理过程，这些钩子处理过程形成了钩子链。当前钩子处理结束后应把钩子信息传递给下一个钩子函数。而且最新安装的钩子放在链的开始，而最早安装的钩子放在链的最后，也就是后加入的钩子先获得控制权。

（3）钩子特别是系统钩子会消耗消息处理时间，降低系统性能。只有在必要的时候才安装钩子，在使用完毕后要及时卸载它。

10.7.2　使用 Hook 函数

编写钩子程序的过程分为 3 步：定义钩子函数、安装钩子和卸载钩子。

1．定义钩子函数

钩子函数是一种特殊的回调函数。钩子监视的特定事件发生后，系统会调用钩子函数进行处理。不同事件的钩子函数的形式是各不相同的。下面以鼠标钩子函数为例说明钩子函数的原型：

```
LRESULT CALLBACK HookProc(int nCode ,WPARAM wParam,LPARAM lParam)
```

参数 wParam 和 lParam 包含所钩消息的信息，比如鼠标位置、状态，键盘按键等。nCode 包含有关消息本身的信息，比如是否从消息队列中移出。

先在钩子函数中实现自定义的功能，然后调用函数 CallNextHookEx()把钩子信息传递给钩子链的下一个钩子函数。CallNextHookEx()的原型如下：

```
LRESULT CallNextHookEx( HHOOK hhk, int nCode, WPARAM wParam, LPARAM lParam )
```

参数 hhk 是钩子句柄，nCode、wParam 和 lParam 是钩子函数，当然也可以通过直接返回 True 来丢弃该消息，从而阻止该消息的传递。

2．安装钩子

在程序初始化的时候，调用函数 SetWindowsHookEx()安装钩子。其函数原型为：

```
HHOOK SetWindowsHookEx( int idHook,HOOKPROC lpfn, INSTANCE hMod,DWORD dwThreadId )
```

参数 idHook 表示钩子类型，它是和钩子函数类型一一对应的，比如，WH_KEYBOARD 表示安装的是键盘钩子，WH_MOUSE 表示是鼠标钩子等；lpfn 是钩子函数的地址；hMod 是钩子函数所在的实例的句柄，对于线程钩子，该参数为 NULL，对于系统钩子，该参数为钩子函数所在的 DLL 句柄；dwThreadId 指定钩子所监视的线程的线程号，对于全局钩子，该参数为 NULL。该函数返回所安装的钩子句柄。

3．卸载钩子

当不再使用钩子时，必须及时将其卸载。简单地调用函数：

```
BOOL UnhookWindowsHookEx( HHOOK hhk)
```

值得注意的是，线程钩子和系统钩子的钩子函数的位置有很大的差别。线程钩子一般在当前线程或者当前线程派生的线程内，而系统钩子必须放在独立的动态链接库中，实现起来要麻烦一些。

10.7.3　Hook 鼠标

下面实现一个进程内的钩子程序，来监视并截获鼠标消息。新建一个基于对话框的 MFC Application，取名为 hook。

然后按照上一节对使用 Hook 函数过程的介绍，首先定义钩子函数，代码如下：

```
LRESULT CALLBACK MouseProc(int nCode ,WPARAM wParam ,LPARAM lParam )
{
    return 1;
}
```

将该段代码放在 ChookDlg 类的 OnInitDialog()函数定义之前，这里让函数返回 1 是因为如果钩子函数返回非 0 值，表示已对当前消息进行了处理，这样，系统就不会再将这个消息传递给目标窗口过程。因此，为了不让其他消息处理函数再对鼠标消息进行响应，应返回一个非 0 值；否则，还可以调用 CallNextHookEx()函数并返回该函数的返回值，以便其他安装了 WH_HOUSE 类型钩子过程的应用程序获得相应的钩子通知。

因为已经在上一步返回了非 0 值，所以无须再调用 CallNextHookEx()函数，而是进入安装钩子的过程。在调用 SetWindowsHookEx()函数之前，首先要定义一个 HHook 类型的全局变量并将其初始化为 NULL，来保存该函数返回的鼠标钩子过程句柄，代码如下：

```
HHook hMouse = NULL ;
```

然后，就可以在 CHookDlg 类的 OnInitDialog()函数中调用 SetWindowsHookEx()函数了。具体代码如下：

```
                // TODO: 在此添加额外的初始化代码
hMouse = SetWindowsHookEx ( WH_MOUSE , MouseProc , NULL , GetCurrentThreadId() ) ;
return TRUE;        // 除非将焦点设置到控件，否则返回 True
```

第一个参数设定为 WH_HOUSE，表示安装的钩子过程对鼠标消息进行监视；第二个参数是刚刚定义的鼠标钩子过程名称；第三个参数设为 NULL，是因为这里创建的钩子过程和当前进程中的主线程相关；第四个参数使用 GetCurrentThreadId()函数获得当前进程的 ID。

编译并运行该程序，会发现在应用程序窗口中无论怎样单击鼠标，都没有响应，只能使用【Enter】键退出程序。运行结果如图 10.18 所示。

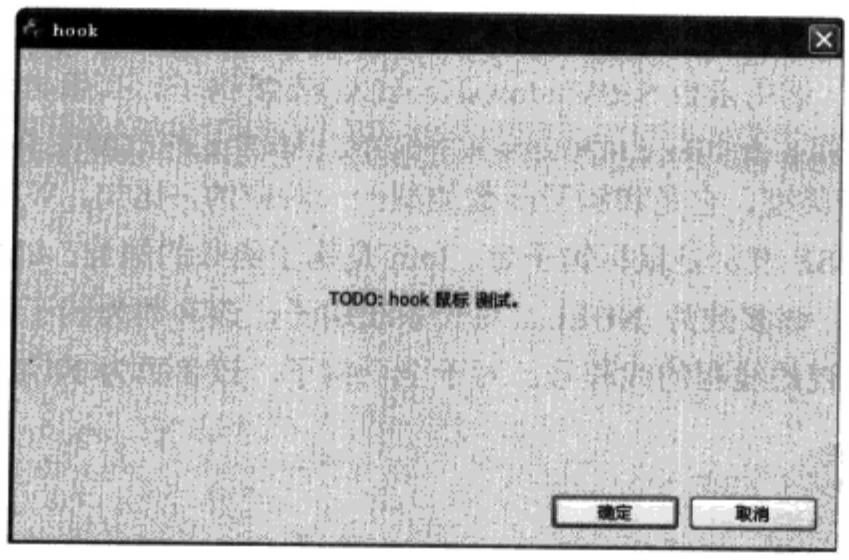


图 10.18　hook 鼠标运行结果

10.7.4　Hook 键盘

下面，再用同样的方法创建一个键盘钩子，使得刚才创建的工程不仅屏蔽鼠标消息，还可以屏蔽键盘消息。首先还是要定义一个键盘钩子过程，并让该过程返回非 0 值，表示该过程已经处理了键盘消息。代码如下：

```
LRESULT CALLBACK KeyboardProc(int code ,WPARAM wParam ,LPARAM lParam )
```

```
{
    return 1;
}
```

接着，定义一个 HHOOK 类型的全局变量并将其初始化为 NULL，来保存 SetWindowsHookEx()函数返回的键盘钩子过程句柄，代码如下：

```
HHOOK hKeyboard = NULL;
```

然后，在 CHookDlg 类的 OnInitDialog()函数中调用 CallNextHookEx()函数，具体代码如下：

```
                    // TODO: 在此添加额外的初始化代码
hMouse = SetWindowsHookEx ( WH_MOUSE , MouseProc , NULL , GetCurrentThreadId() );
hKeyboard = SetWindowsHookEx ( WH_KEYBOARD , KeyboardProc , NULL , GetCurrentThreadId() );
return TRUE;        // 除非将焦点设置到控件，否则返回 True
```

编译并运行该程序，会发现该应用程序窗口不仅对鼠标没有反应，而且也无法使用键盘来操作了。这样就成功实现了两个钩子来截获键盘和鼠标消息。最后，只能使用任务管理器来结束该程序了，运行结果如图 10.19 所示。

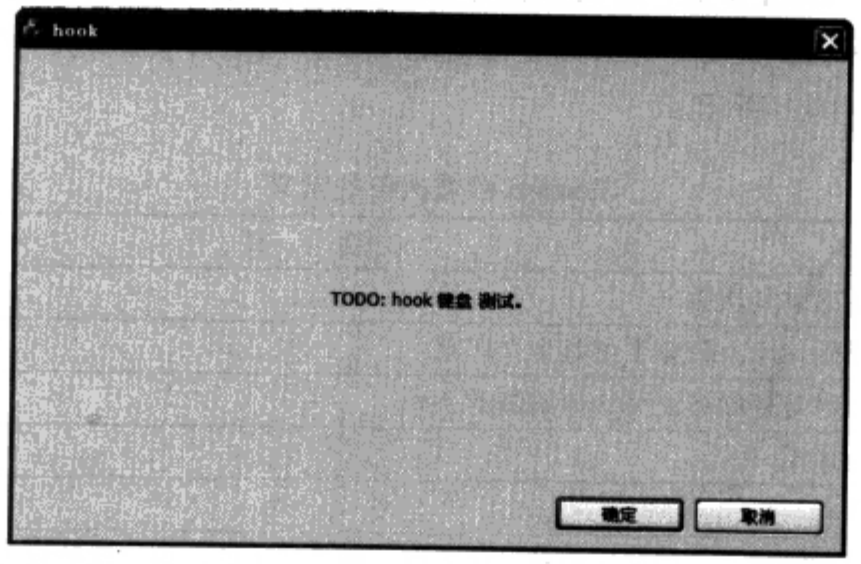


图 10.19　hook 键盘运行结果

使用任务管理器来退出程序的操作是不好的，应该预留一个“后门”，比如用户按【F4】键，即可退出程序，而按其他键则仍然无效，这应该如何实现呢？

首先要做的，当然是捕获用户按【F4】键这个消息。读者可以看到，在定义的键盘钩子过程函数中有 3 个参数，根据之前介绍的，参数 wParam 和 lParam 包含所钩消息的信息，比如鼠标位置、状态，键盘按键等。那么，就可以在钩子过程中通过 wParam 参数获得当前按下的是哪个键。这里，wParam 参数的值是当前按键消息的键盘按键的虚拟键代码。其中，【F4】键的虚拟键码是 VK_F4（关于键盘所有按键的虚拟键码，读者可以在 MSDN 联机帮助文档中搜索到）。

捕获到用户按下的【F4】键消息之后，发送一个 WM_CLOSE 消息，让应用程序从主窗口退出，这里可以使用 SendMessage()函数来发送消息。但是需要注意的是，这里所定义的键盘钩子过程是一个全局函数，因此在它的内部只能调用全局的 SendMessage()函数，该函数的第一个参数是目标窗口的句柄，CHookDlg 类的成员变量 m_hWnd 保存了这个窗口的句柄。因此需要定义一个全局窗口变量 hwnd 并将 m_hWnd 的值赋给它。定义代码如下：

```
HHOOK hMouse = NULL;
HHOOK hKeyboard = NULL;
HWND hWnd=NULL;
```

然后在 OnInitDialog()函数中为其赋值：

```
hWnd = m_hWnd;
```

这样，就可以定义键盘钩子过程函数了，代码如下：

```
LRESULT CALLBACK KeyboardProc(int code ,WPARAM wParam ,LPARAM lParam )
{
    if(VK_F4==wParam)
    {
        ::SendMessage(hWnd,WM_CLOSE,0,0);
        UnhookWindowsHookEx(hKeyboard);
        UnhookWindowsHookEx(hMouse);
    }
    return 1;
}
```

SendMessage()函数下面两行是前面提到过的卸载钩子的函数，使用该函数，在程序退出后将安装的钩子卸载。

编译并运行程序，按【F4】键，程序就成功关闭了。

上面的程序使用了 wParam 参数，那么参数 lParam 又有什么作用呢？它是一个 32 位的整数，其中的每一位或某几位表示特定的含义，用来指定按键的重复的次数、扫描码、扩展键标记、上下文代码等标记，其具体内容如表 10.1 所示。

表 10.1　　lParam 参数的位的含义

位	含　义
0～15 位	表示按键的重复次数
16～23 位	表示扫描码，此值依赖于键盘生产厂家
24 位	表示该按键是扩展按键（比如右侧的 Alt）
25～28 位	系统保留未用
29 位	上下文代码
30 位	表示先前的键状态，如果在消息发送之前，键是按下的，该位就被设为 1，否则为 0
31 位	表示变化状态，如果此键正在被按下，其值为 0

使用这个参数，可以做什么呢？举个简单的例子，通常退出应用程序都是使用【Alt+F4】组合键，那么下面的代码就可以实现使用【Alt+F4】组合键来代替【F4】键，关闭应用程序。代码如下：

```
LRESULT CALLBACK KeyboardProc(int code ,WPARAM wParam ,LPARAM lParam )
{
    if(VK_F4==wParam&&(1==(lParam>>29&1)))
    {
        ::SendMessage(hWnd,WM_CLOSE,0,0);
        UnhookWindowsHookEx(hKeyboard);
        UnhookWindowsHookEx(hMouse);
    }
    return 1;
}
```

代码中阴影部分就是配合使用参数 lParam、wParam 实现了组合键的输入，读者可以自行实现一下。

10.8　小结

本章详细地介绍了 Visual C++ 2010 MFC 中动态函数链接库的相关知识，包括动态链接库的基本

概念、原理和优点，调用动态函数链接库的方法，动态链接库的框架，以及如何创建各种类型的动态链接库，等等，最后还有 Hook 的一些相关知识。本章的重点是：

- 动态链接库的作用和优点，为什么要使用动态链接库；
- 两种调用动态链接库的方法——静态链接和动态链接及具体实现；
- 创建不同类型动态链接库的方法，包括 MFC DLL、WIN32 DLL、资源 DLL 及其各自的作用；
- 什么是 Hook，创建一个 Hook 的过程；
- 如何创建一个鼠标钩子和键盘钩子。

第 11 章　Visual C++ 2010 MFC ActiveX 控件

本章主要介绍了 Visual C++ 2010 MFC ActiveX 控件的基本知识与开发。首先对 ActiveX 控件进行了简单的介绍，接着介绍了 ActiveX 控件的测试、注册，然后对 ActiveX 控件的属性、事件和方法开发进行了详细的说明。完成本章的学习之后，读者将对 Visual C++ 2010 ActiveX 控件有一定的了解，并且能够自己进行 ActiveX 控件方面的开发。

本章主要内容有：

- ActiveX 控件简介；
- ActiveX 控件测试与注册；
- MFC ActiveX 控件向导；
- ActiveX 控件属性开发；
- ActiveX 控件事件开发；
- ActiveX 控件方法开发；
- 完整的 ActiveX 控件范例；
- 调用 ActiveX 控件。

11.1　ActiveX 控件简介

ActiveX 控件是基于组件对象模型（COM）的可重用软件组件，它支持广泛的 OLE 功能并可自定义以满足多种软件的需要。ActiveX 控件技术涉及了几乎所有 COM 和 OLE 的技术精华，如可链接对象、统一数据传输、OLE 文档、属性页、永久存储及 OLE 自动化等。ActiveX 控件旨在用于普通的 ActiveX 控件容器和 Internet 上的网页。既可以用此处介绍的 MFC，也可以用活动模板库（ATL）来创建 ActiveX 控件。

ActiveX 控件可以在自己的窗口中对自身进行描述，对事件（如单击鼠标）做出响应，并可以通过包括属性和方法的接口进行管理，这些属性和方法与自动化对象中的属性和方法相似。可以为许多用途（如数据库访问、数据监视或图形绘制）开发这些控件。除可移植性外，ActiveX 控件还支持之前所不具备的功能，如与现有 OLE 容器的兼容性和将其菜单与 OLE 容器的菜单集成在一起的功能。另外，ActiveX 控件完全支持自动化，使控件得以公开读/写属性和一组可由控件用户调用的方法。可以创建无窗口的 ActiveX 控件和只有在活动时才创建窗口的控件。无窗口控件可以加速应用程序的显示，并可包含透明控件和非矩形控件，也可以异步加载 ActiveX 控件的属性。

ActiveX 控件作为进程内服务器（通常是一个小型对象）的实现，而进程内服务器可用于任何 OLE 容器。请注意，只有在识别 ActiveX 控件的 OLE 容器内使用时，ActiveX 控件的全部功能才可用。这种类型的容器（以后称为“控件容器”）可以通过使用 ActiveX 控件的属性和方法来操作该控件，并可以从 ActiveX 控件接收事件形式的通知。事件的触发通常是通过控件包容器提供的 IDispatch 接口来调用自动化对象的方法来实现的。在设计 ActiveX 控件时就应当考虑控件可能会发生哪些事件，以及包容器程序将会对其中的哪些事件感兴趣并将这些事件包含进来。与自动化服务不同，ActiveX 控件的

方法、属性和事件均有自定义（custom）和常用（stock）两种不同的类型。自定义的方法和属性也就是普通的自动化方法和属性，自定义事件则是自己选取名称和 Dispatch ID 的事件；而所谓的常用方法、属性和事件则是使用了 ActiveX 控件规定了名称和 Dispatch ID 的“标准”方法、属性和事件。

ActiveX 控件可以使 COM 组件从外观和使用上与普通的窗口控件一样，而且还提供了类似于设置 Windows 标准控件属性的属性页，使其能够在包容器程序的设计阶段对 ActiveX 控件的属性进行可视化设置。ActiveX 控件提供的这些功能使得对其的使用将是非常方便的。

当在控件容器内使用控件时，该控件使用两种机制进行通信：一种是公开属性和方法，另一种是引发事件。

ActiveX 控件使用 3 种编程元素与 ActiveX 控件容器应用程序之间进行交互，它们分别是：

1．基类 COleControl 类

要开发的 ActiveX 控件都是从 COleControl 类中派生出来的，COleControl 类中不仅封装了常用事件、属性和方法（这方面内容后面会进行详细介绍），而且还具有其他强大的功能，比如支持 ActiveX 文件对象和 ActiveX 自动化。此外，COleControl 是从 CWnd 类派生的，因此它具有所有窗口类的性质和功能。COleControl 类是连接 ActiveX 控件和 ActiveX 控件容器的纽带。

2．事件激发函数

ActiveX 控件具有触发事件的能力，当某个条件得到满足时，它发送事件给 ActiveX 控件容器，另外还可以在发送消息的时候附带上参数，这样开发人员就可以向 ActiveX 容器发送有关控件的扩充消息。事件激发函数是 ActiveX 控件通知 ActiveX 控件容器如何进行操作的方法。

3．派遣映射表

派遣映射表的作用是公开属性和方法给 ActiveX 控件容器，通过派遣映射表，ActiveX 控件容器可以采用不同的方法来操作 ActiveX 控件，它可以改变控件的外观，改变某个属性值，或是对控件发出一个请求，等等。派遣映射表是 ActiveX 控件容器控制和操纵 ActiveX 容器的手段。

11.2　ActiveX 控件测试与注册

11.2.1　ActiveX 控件的测试

可以使用两种方法对 ActiveX 控件进行测试，一种是使用 ActiveX 控件测试容器，另一种是使用 Visual Basic 进行测试。下面主要对使用 ActiveX 控件测试容器的方法进行介绍。

但是 Visual Studio 2010 中没有 TSTCON32.exe 文件，而是微软把这个文件放到了例子程序中，需要用户自己编译。具体方法为：比如将 Visual Studio 2010 装在了 F:\盘下，那么请用户进入“F:\ Microsoft Visual Studio 10.0\Samples\1033\”目录，在该文件夹下找到 AllVCLanguageSamples.zip 文件并将其解压缩，在生成的文件夹下可以找到 TSTCON32.exe 的源代码“C++\MFC\ole\TstCon”，然后编译该文件，就可以生成所需的 TSTCON32.exe 文件。

如果在 Visual Studio 2010 开发环境中运行编写的 ActiveX 程序，将打开 ActiveX 控件测试容器应用程序，如图 11.1 所示。

然后就可以加载特定的 ActiveX 控件，方法是选择“编辑”→“插入新控件”命令，这时将打开如图 11.2 所示的对话框，在该对话框左侧的列表框中任意选中一个控件，比如快速连续地按【M】和【Y】键，就可以直接定位到刚刚生成的 MyActiveX 控件。

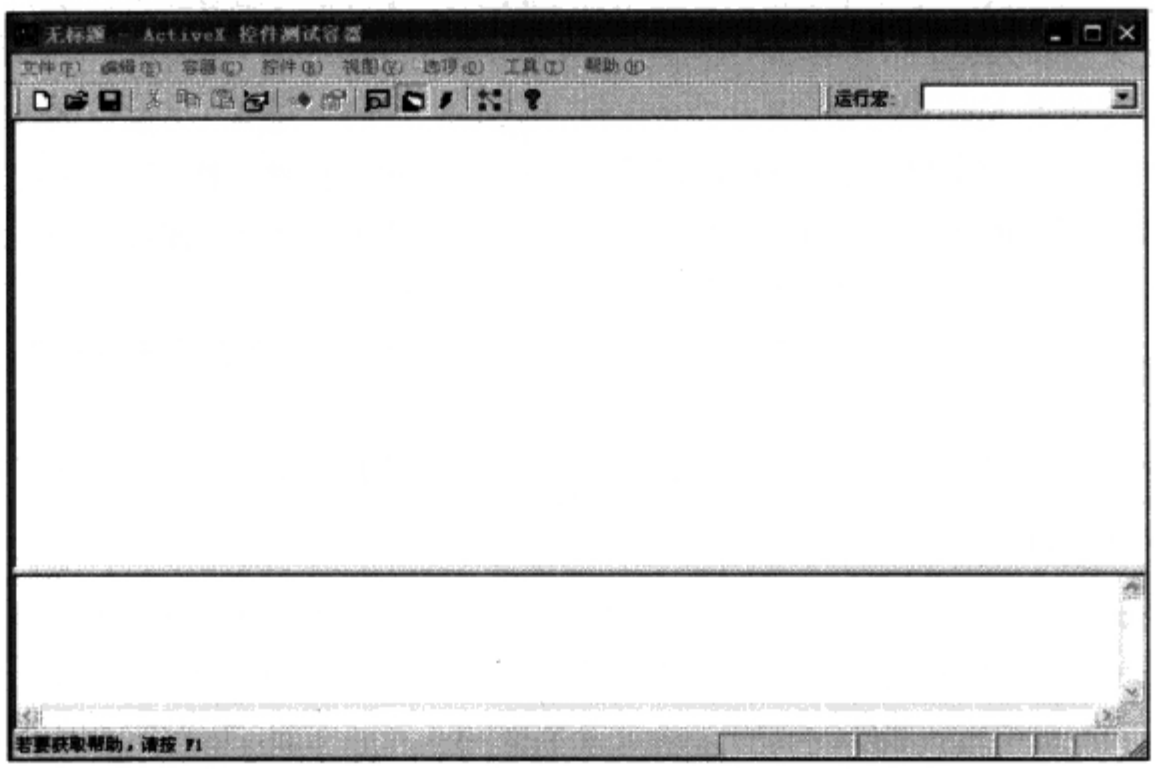


图 11.1　ActiveX Control Test Container 应用程序初始界面

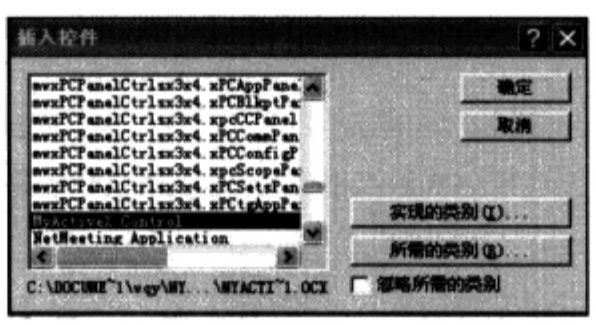


图 11.2　插入 My ActiveX 控件对话框

然后单击对话框中的“确定”按钮关闭该对话框，这时在 ActiveX 控件测试容器应用程序中就加载了 MyActiveX 控件，这个 ActiveX 控件当前的功能就是绘制一个椭圆，如图 11.3 所示。当然，这时该控件的代码都是 MFC 自动生成的，开发人员并没有编写一行代码。读者可以看到，利用 MFC ActiveX 控件，不需要开发人员编写一行代码就可以生成一个绘制椭圆的 ActiveX 控件。

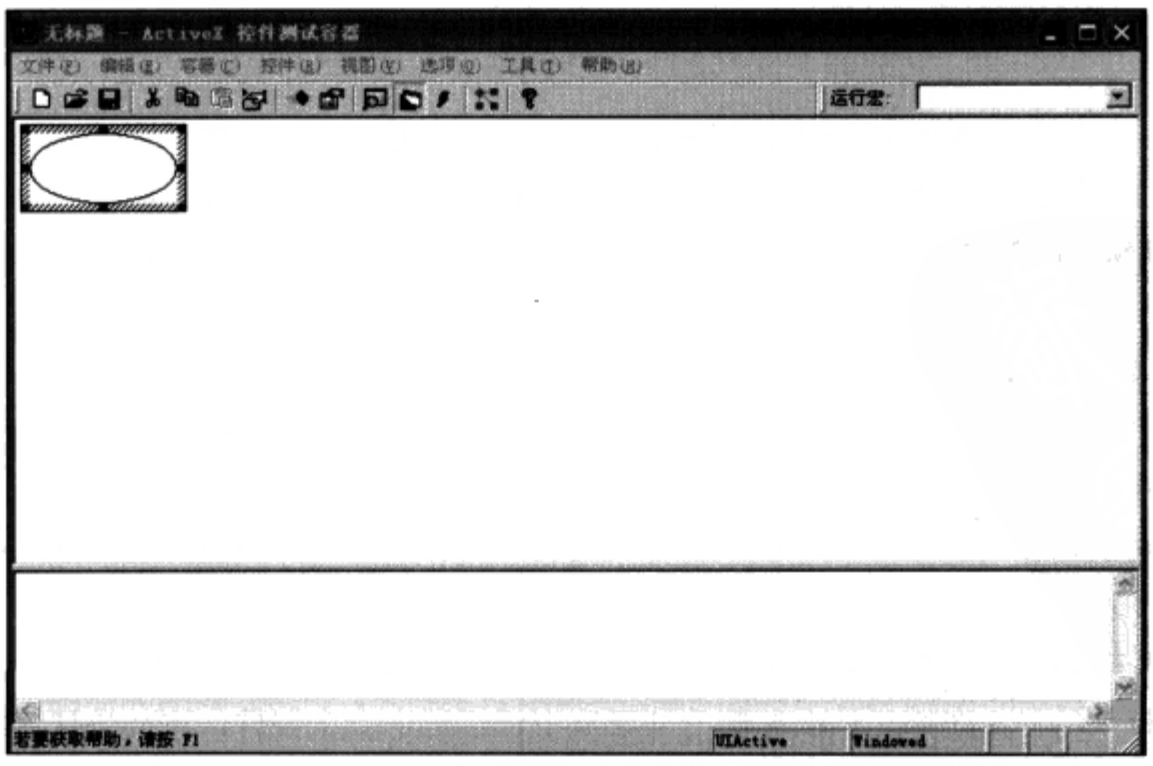


图 11.3　ActiveX 控件绘制的一个椭圆

11.2.2　ActiveX 控件的注册

实际上，当在 Visual Studio 2010 开发环境中生成 MyActiveX 控件程序时，就已经通过 regsvr32 程序注册了 AcitveX 控件；也就是说，在生成控件时，Visual Studio 2010 编译器会调用 regsvr32 程序注册 MyActiveX 控件，将该控件的所有信息，包括其所在路径都写入注册表中。当在 ActiveX 控件测试容器或 Visual Basic 这些测试容器中加载 ActiveX 控件时，它们都会从注册表中搜寻所有的 ActiveX 控件的相关信息，并在它们的控件对话框中列出这些控件，这就是能够在测试容器中找到 ActiveX 控件相关信息的原因。所有的 ActiveX 控件必须在注册之后才能使用。

如果想要删除 ActiveX 控件的注册信息，可以利用 regsvr 程序的/u 选项来实现。通常，可以选择系统的“开始”→“运行”命令，然后在“运行”对话框中的下拉列表框中输入“regsvr32 /u”，再在其后输入想要删除的 ActiveX 控件的完整路径，如图 11.4 所示。

单击“确定”按钮，这时会弹出如图 11.5 所示的信息对话框。

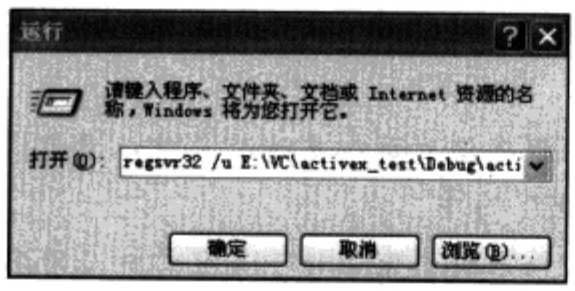


图 11.4　删除的 ActiveX 控件的完整路径

图 11.5　删除的 ActiveX 控件的完整路径

该信息对话框中提示“E:\VC\active_test\Debug\activex_test.ocx 中的 DllUnregisterServer 成功”，这里的 DllUnregisterServer()是一个函数，并且是 ActiveX 控件提供的一个函数。执行“regsvr32 /u”命令时实际上调用的是指定控件的 DllUnregisterServer()函数来删除该控件的注册信息，因为对于 regsvr 这个程序来说，并不知道需要删除哪些信息，所以它只是调用控件的 DllUnregisterServer()函数，由后者来删除该控件在注册表中的注册信息。

当删除了这个控件在注册表中的信息之后，如果在 ActiveX 控件测试容器或 Visual Basic 程序中再想加载这个控件，在控件列表中就找不到它了。

如果想再次注册这个控件，仍可以选择 regsvr32 程序，但不需要使用/u 选项，其他同上，这时将显示如图 11.6 所示的对话框。

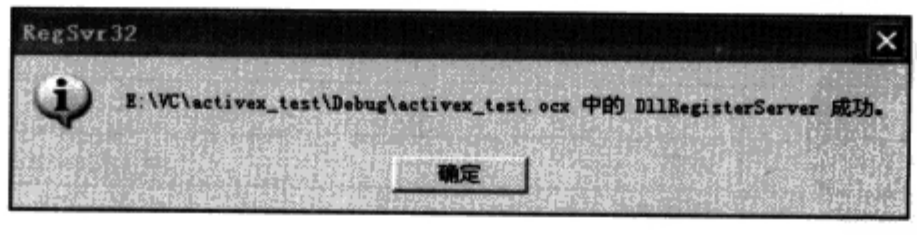


图 11.6　再次注册控件

在该信息对话框中提示“E:\VC\active_test\Debug\activex_test.ocx 中的 DllRegisterServer 成功”。同样，DllRegisterServer()也是 ActiveX 控件提供的一个函数。当执行 regsvr32 命令时，实际上是调用指定控件的 DllRegisterServer()函数，将该控件的信息写入注册表。因此，ActiveX 控件的注册和取消注册都是利用控件自身提供的两个函数来完成的，regsvr32 程序只是调用这些函数而已。当注册完成之后，在 ActiveX 控件测试容器或 Visual Basic 程序的控件列表中就可以找到这个控件了。

11.3 MFC ActiveX 控件向导

ActiveX 控件是特定的自动化服务器类型，它是可重用的组件。承载 ActiveX 控件的应用程序是该控件的自动化客户端，如果目标是创建这种可重用的组件，则使用该向导创建控件。或者，可以使用 MFC 应用程序向导创建自动化服务器 MFC 应用程序。

用该向导创建的 ActiveX 控件可以有用户界面或是不可见的，可以在向导的控件设置页中指示此选项。定时器（Timer）控件是希望其不可见的 ActiveX 控件的示例。ActiveX 控件可以有复杂的用户界面，一些控件可以像封装窗体：一个控件包含许多字段，而每个字段本身就是一个 Windows 控件。例如，作为 MFC ActiveX 控件实现的自动部件对象可能呈现类似于窗体的用户界面，通过该界面用户可以读取和编辑部件编号、部件名称和其他信息。

MFC 起始程序包括 C++ 源文件（.cpp）、资源文件（.rc）和一个项目文件（.vcproj），这些起始文件中生成的代码基于 MFC。

此向导描述正在创建的 MFC ActiveX 控件项目的当前应用程序设置。默认情况下，该向导按如下方式创建项目。

- 默认项目不生成运行时许可证和帮助文件。可以在应用程序设置页中更改这些默认设置。只有在该 ActiveX 控件向导页中所做的选择才反映在“概述”页中。
- 项目包括基于项目名称的控件类和属性页类，可以在控件名称页中编辑项目名称和文件名。
- 控件不基于现有的 Windows 控件，当变为可见时激活，有用户界面并且包括“关于”对话框。可以在控件设置页中更改这些默认设置。

11.4 ActiveX 控件属性开发

ActiveX 控件通过引发事件与其控件容器进行通信，而容器使用方法和属性与控件进行通信，方法和属性分别与 C++类的成员函数和成员变量的用法和用途相似。属性是 ActiveX 控件中向所有容器公开的数据成员，为包含 ActiveX 控件的应用程序（如自动化客户端和 ActiveX 控件容器）提供接口。

ActiveX 控件的属性分为常用属性和自定义属性，常用属性由 COleControl 类提供，而自定义属性由 MFC 所提供的派遣映射表支持。

常用属性如表 11.1 所示。

表 11.1　　ActiveX 控件的常用属性

属　性	说　明	调　用
Appearance	外观	m_sAppearance 变量
BackColor	背景色	GetBackColor()函数
BorderStyle	边界样式	m_sBorderStyle 变量
Caption	标题	InternalGetText()函数
Enable	启用标志	m_bEnabled 变量
Font	字体	GetFont()和 SetFont()函数
ForeColor	前景颜色	GetForeColor()函数
hWnd	窗口句柄	m_hWnd 变量
Text	文字	InternalGetText()函数

在自定义属性中，还可以进一步分为 4 种实现方式：成员变量、具有通知特性的成员变量、Get/Set 函数和参数化，下面分别进行介绍。

- 成员变量：这种实现方式是在控件类中作为成员变量的属性状态，如果不关心属性值的改变，就可以使用成员变量来实现。在上述 4 种实现方式中，这种实现方式所创建的代码最少，成员变量实现的派遣映射宏为 DISP_PROPERTY。
- 具有通知特性的属性：这种实现是由一个成员变量和一个类向导所创建的通知函数组成的，当属性值发生改变时，通知函数自动被调用，这种实现方式的派遣映射表项宏为 DISP_PROPERTY_NOTIFY。
- Get/Set 函数：这种实现是由一个成员函数组成的，当控件的用户需要获取属性的值时调用 Get 函数，当需要设置控件的值时调用 Set 函数。在运行时需要计算属性的值，或是改变实际属性之前要对传入的值进行预处理及需要实现一个只读/只写属性时，需要使用这种实现方法。这种实现方法的派遣映射表项宏为 DISP_PROPERTY_EX。
- 参数化：参数化实现是由类向导支持的，参数化属性可以用来通过控件的单一属性访问一组值，这种实现的派遣映射表项宏为 DISP_PROPERTY_PARAM。

11.4.1　添加常用属性

常用属性与自定义属性的区别在于，常用属性已由 COleControl 类实现。COleControl 类包含支持控件的常用属性的预定义成员函数，常用属性包括控件的标题及前景色和背景色。常用属性的调度映射项总是以 DISP_STOCKPROP 开头。

1．使用添加属性向导添加常用属性

添加常用属性所需的代码比添加自定义属性要少，这是因为对属性的支持由 COleControl 自动处理。

2．使用添加属性向导添加常用 Caption 属性

下列过程说明如何将常用 Caption 属性添加到 ActiveX 控件框架。此过程也可用于添加其他常用属性，用选定的常用属性名替换 Caption 即可。

- 加载控件的项目；
- 在类视图中展开控件的库节点；
- 右击控件的接口节点（库节点的第二个节点）打开快捷菜单；
- 在快捷菜单中选择“Add”→“Add Property”命令，此操作将打开添加属性向导；
- 在“Property name”列表框中，选择“Caption”选项；
- 单击“Finish”按钮。

3．添加属性向导对常用属性的更改

由于 COleControl 支持常用属性，添加属性向导不以任何方式更改类声明；它将属性添加到调度映射。添加属性向导将下列代码行添加到控件的调度映射，此调度映射位于实现（.cpp）文件中：

```
DISP_STOCKPROP_CAPTION()
```

下列代码行被添加到控件的接口说明（.idl）文件：

```
[id ( DISPID_CAPTION ) , bindable , requestedit] BSTR Caption;
```

该行代码为 Caption 属性分配一个特定的 ID。注意，该属性是可绑定的，它在修改值之前从数据库请求权限，这使控件的用户可以使用 Caption 属性。

11.4.2 添加自定义属性

自定义属性有 4 种实现类型：成员变量、带通知的成员变量、Get/Set 方法和参数化。

1. “成员变量”实现

这种实现将属性的状态表示为控件类中的成员变量。对于控件而言，当知道属性值何时更改并不重要时，使用“成员变量”实现。在这 4 种类型中，这种实现为属性创建的支持代码量最少。“成员变量”实现的调度映射项宏为 DISP_PROPERTY。

2. “带通知的成员变量”实现

这种实现由一个成员变量和一个由添加属性向导创建的通知函数组成。属性值更改后，框架将自动调用这个通知函数。当需要在属性值更改后得到通知时，使用“带通知的成员变量”实现。由于这种实现需要调用一个函数，因此需要较长的时间。“带通知的成员变量”实现的调度映射项宏为 DISP_PROPERTY_NOTIFY。

3. “Get/Set 方法”实现

这种实现由控件类中的一对成员函数组成。当控件用户请求属性的当前值时，“Get/Set 方法”实现自动调用 Get 成员函数；而当控件用户请求更改属性值时，自动调用 Set 成员函数。当需要在运行时计算属性值、在更改实际属性之前验证控件用户传递的值或实现只读或只写属性类型时，使用这种实现。“Get/Set 方法”实现的调度映射项宏为 DISP_PROPERTY_EX。

4. “参数化”实现

添加属性向导支持参数化实现。参数化属性（有时称为属性数组）可用于通过控件的单个属性访问一组值。“参数化”实现的调度映射项宏为 DISP_PROPERTY_PARAM。

11.4.3 高级属性实现

添加属性向导为控件提供了一种实现只读或只写属性的便捷方法。

1. 实现只读或只写属性

- 加载控件的项目；
- 在类视图中展开控件的库节点；
- 右击控件的接口节点（库节点的第二个节点）打开快捷菜单；
- 在快捷菜单中选择“Add”→“Add Property”命令，此操作将打开添加属性向导；
- 在“Property name”下拉列表框中，输入属性名；
- 设置“Implementation type”为“Get/Set methods”；
- 在“Property type”下拉列表框中，选择正确的属性类型；
- 如果需要只读属性，则清除 Set 函数名，如果需要只写属性，则清除 Get 函数名。
- 单击“Finish”按钮。

执行此操作时，添加属性向导在调度映射项中插入 SetNotSupported()或 GetNotSupported()函数以取代常规的 Set 或 Get 函数。如果要将现有属性更改为只读或只写，可以手动编辑调度映射，并将不必要的 Set 或 Get 函数从控件类中移除。如果希望属性根据条件成为只读或只写（例如，仅当控件以特定模式操作时），可以像正常情况那样提供 Set 或 Get 函数，并在适当的位置调用 SetNotSupported()或 GetNotSupported()函数。例如：

```
void CSampleCtrl::SetMyProperty( short propVal)
{
    if ( m_bReadOnlyMode )
        SetNotSupported();
    else
        m_ipropVal = propVal;
}
```

如果 m_bReadOnlyMode 数据成员为 True，则此代码示例调用 SetNotSupported()；如果为 False，则将此属性设置为新值。

2．从属性返回错误代码

若要指出在尝试获取或设置属性时发生了错误，请使用 COleControl::ThrowError()函数，它采用 SCODE（状态代码）作为参数。可以使用预定义 SCODE 或定义自己的 SCODE。大多数常用的预定义 SCODE 都有 Helper()函数，例如：

- COleControl::SetNotSupported()函数;
- COleControl::GetNotSupported()函数;
- COleControl::SetNotPermitted()函数。

11.4.4　访问环境属性

控件可以通过访问容器的环境属性来获得其容器的有关信息。这些属性公开可视特性和可操作特性；其中可视特性指容器的背景色、容器所使用的当前字体等特性，可操作特性指容器当前为用户模式还是设计器模式等特性。控件可以使用环境属性调整其外观和行为，以适应控件所嵌入的特定容器；但是，控件永远不应假定其容器将支持任何特定的环境属性。实际上，一些容器可能根本不支持任何环境属性。没有环境属性时，控件应假定合理的默认值。若要访问环境属性，调用 COleControl::GetAmbientProperty()函数。此函数需要环境属性的调度 ID 作为其第一个参数（OLECTL.h 文件定义标准环境属性集的调度 ID）。

GetAmbientProperty()函数的参数为调度 ID、指示所需属性类型的变量标记和指向返回值所在的内存位置的指针。该指针引用的数据类型因变量标记而异。如果容器支持属性，则此函数返回 True，否则返回 False。

11.5　ActiveX 控件事件开发

ActiveX 控件使用事件来通知 ActiveX 控件容器有关控件所发生的事情，一些常用的事件有：鼠标单击控件、用键盘输入数据和改变控件的状态等。这些行为发生时，控件将激发一个事件来通知 ActiveX 控件容器。

MFC 支持两种类型的事件：常用事件和自定义事件。常用事件是指那些由 COleControl 类自动处理的事件；自定义事件允许控件对一些特有的行为触发事件，比如控件的内部状态改变或者接收到某种类型的窗口消息等。为使控件正确地引发事件，控件类必须将控件的每个事件分别映射到一个成员函数，当相关事件发生时将调用相应的成员函数，此映射机制（称为事件映射）集中了有关事件的信息，并使 Visual Studio 得以轻松访问和操作控件的事件。

为了正确激发 ActiveX 控件的事件，控件类必须把每一个事件映射为一个成员函数，相关事件发生时，该成员函数被调用，这种映射机制称为“事件映射表”，它集中处理有关事件的信息，并允许

类向导访问和操纵事件。

此事件映射由下面的宏声明，该宏位于控件类声明的头文件（.h）中：

```
DECLARE_EVENTSINK_MAP()
```

声明了事件映射后，必须在控件的实现文件（.cpp）中定义该映射。下面的代码行定义事件映射，使控件得以引发特定的事件：

```
BEGIN_EVENT_MAP( CSampleCtrl,COleControl)
//{{AFX_EVENT_MAP(CSampleCtrl)
...
//}}AFX_EVENT_MAP
END_EVENT_MAP()
```

如果使用 MFC ActiveX 控件向导创建项目，它将自动添加这些代码行；否则必须手动添加这些代码行。使用类视图可以添加受 COleControl 类支持的常用事件或用户定义的自定义事件。对于每个新事件，类视图自动将适当的项添加到控件的事件映射和控件的.idl 文件中。

11.5.1 添加常用事件

ActiveX 控件所支持的常用事件如表 11.2 所示。

表 11.2　　ActiveX 控件所支持的常用事件

事　件	引 发 函 数	说　明
Click	void FireClick( )	使用鼠标中的任意一个键单击控件时触发这个事件，要注意的是，MouseDown 和 MouseUp 事件发生在 Click 事件之前。 事件映射项：EVENT_STOCK_CLICK( )
DblClick	void FireDblClick( )	使用鼠标中的任意一个键双击控件时触发这个事件，与 Click 事件相同的是，MouseDown 和 MouseUp 事件发生在 Click 事件之前。 事件映射项：EVENT_STOCK_DBLCLICK( )
Error	void FireError(SCODE scode、LPCSTR lpszDescription、UINT nHelpID = 0)	当 ActiveX 控件在方法和属性访问操作之外发生错误时触发这个事件。 事件映射项：EVENT_STOCK_ERROREVENT( )
KeyDown	void FireKeyDown(short nChar、short nShiftState)	当 ActiveX 控件接收到 WM_SYSKEYDOWN 或 WM_KEYDOWN 消息时触发这个事件。 事件映射项：EVENT_STOCK_KEYDOWN( )
KeyUp	void FireKeyPress(short* pnChar)	当 ActiveX 控件接收到 WM_SYSKEYUP 或 WM_KEYUP 消息时触发这个事件。 事件映射项：EVENT_STOCK_KEYPRESS( )
KeyPress	void FireKeyUp(short nChar、short nShiftState)	当 ActiveX 控件接收到一个 WM_CHAR 消息时触发这个事件。 事件映射项：EVENT_STOCK_KEYUP( )
MouseDown	void FireMouseDown (short nButton、short nShiftState、float x、float y)	当鼠标的任意一个键在 ActiveX 控件中按下时触发这个事件。 事件映射项：EVENT_STOCK_MOUSEDOWN( )
MouseMove	void FireMouseMove (short nButton、short nShiftState、float x、float y)	当鼠标在 ActiveX 控件上移动时触发这个事件。 事件映射项：EVENT_STOCK_MOUSEMOVE( )
MouseUp	void FireMouseUp (short nButton、short nShiftState、float x、float y)	当鼠标的任意一个键在 ActiveX 控件中抬起时触发这个事件。 事件映射项：EVENT_STOCK_MOUSEUP( )
ReadyStateChange	void FireReadyStateChange( )	控件因接收到的数据量而转换到下一个就绪状态时引发该事件。 事件映射项：EVENT_STOCK_READYSTATECHANGE( )

常用事件与自定义事件的不同之处在于，常用事件由 COleControl 类自动引发。COleControl 包含预定义成员函数，它们引发常见操作所导致的事件。一些由 ColeControl 类实现的常见操作包括单击和双击控件、键盘事件和鼠标按钮状态发生更改。常用事件的事件映射项总是以 EVENT_STOCK 作为前缀。

1．使用添加事件向导添加常用事件

- 加载控件的项目；
- 在类视图中，右击 ActiveX 控件类以打开快捷菜单；
- 在快捷菜单中，选择“Add”→“Add Event”命令；此操作将打开添加事件向导；
- 在“Event name”下拉列表框中，选择“KeyPress”选项；
- 单击“Finish”按钮。

此过程可用于添加其他常用事件，用选定的常用事件名替换 KeyPress 即可。

2．添加事件向导对常用事件的更改

由于常用事件由控件的基类处理，添加事件向导不以任何方式更改类声明。添加事件向导将事件添加到控件的事件映射，并在其.idl 文件中生成项。下列代码行被添加到控件的事件映射中，此事件映射位于控件类的实现文件（.cpp）中：

```
EVENT_STOCK_KEYPRESS()
```

添加了此代码后，当接收到 WM_CHAR 消息且控件活动时将引发 KeyPress 事件。通过从控件代码中调用 KeyPress 事件的引发函数（如 FireKeyPress()），可以在其他时间引发此事件。添加事件向导将下列代码行添加到控件的.idl 文件中：

```
[id(DISPID_KEYPRESS)] void KeyPress(short* KeyAscii);
```

该行将 KeyPress 事件与其标准调度 ID 相关联，并使容器得以预测 KeyPress 事件。

11.5.2　添加自定义事件

1．使用“添加事件向导”添加自定义事件

- 加载控件的项目；
- 在类视图中，右击 ActiveX 控件类以打开快捷菜单；
- 在快捷菜单中，选择“Add”→“Add Event”命令，此操作将打开添加事件向导；
- 在“Event name”下拉列表框中输入“ClickIn”；
- 在“Internal name”文本框中，输入事件的引发函数名。在本例中，使用添加事件向导提供的默认值（FireClickIn）；
- 使用 Parameter name 和 Parameter type 控件，添加称为 xCoord（OLE_XPOS_PIXELS 类型）的参数；
- 添加另一个称为 yCoord（OLE_YPOS_PIXELS 类型）的参数；
- 单击“Finish”按钮创建事件；
- 可以使用此过程添加其他自定义事件。用自定义事件名及其参数替换 ClickIn 事件名和参数。

2．添加事件向导对自定义事件的更改

当添加自定义事件时，添加事件向导会更改控件类的.h、.cpp 和.idl 文件。下列代码示例专用于 ClickIn 事件，被添加到控件类的头文件（.h）：

```
void FireClickIn(OLE_XPOS_PIXELS xCoord, OLE_YPOS_PIXELS yCoord)
{
```

```
        FireEvent(
        eventidClickIn ,
        EVENT_PARAM( VTS_XPOS_PIXELS VTS_YPOS_PIXELS) ,
           xCoord ,  yCoord
        );
    }
```

此代码声明了一个称为 FireClickIn()的内联函数，该函数用 ClickIn 事件和使用添加事件向导定义的参数调用 COleControl::FireEvent()。另外，下列代码行被添加到控件的事件映射，此事件映射位于控件类的实现（.cpp）文件中：

```
EVENT_CUSTOM("ClickIn", FireClickIn, VTS_XPOS_PIXELS   VTS_YPOS_PIXELS)
```

此代码将 ClickIn 事件映射到内联函数 FireClickIn()，并传递使用添加事件向导定义的参数。最后，下列代码被添加到控件的 .idl 文件：

```
[id(1)] void ClickIn(OLE_XPOS_PIXELS xCoord, OLE_YPOS_PIXELS yCoord);
```

这行代码根据 ClickIn 事件在添加事件向导事件列表中的位置，为该事件分配一个特定的 ID 号。事件列表中的项使容器得以预测该事件，例如，它可能提供在引发事件时执行的处理程序代码。

3. 使用添加事件向导添加消息处理程序

- 加载控件的项目；
- 在类视图中，选择 ActiveX 控件类；
- 在属性面板中，单击消息按钮。属性面板显示可由 ActiveX 控件处理的消息列表，任何以粗体显示的消息都有一个分配的处理函数；
- 在属性面板中，选择要处理的消息，在本例中选择“WM_LBUTTONDOWN”选项；
- 从右侧的下拉列表框中，选择“<Add> OnLButtonDown”选项；
- 在类视图中双击新的处理函数，跳转到 ActiveX 控件的实现文件（.cpp）中的消息处理程序代码。

11.6 ActiveX 控件方法开发

ActiveX 控件激发事件是为了使其能够和 ActiveX 控件容器之间进行通信，与此相对，ActiveX 控件容器也可以通过属性和方法来和 ActiveX 控件进行通信。属性和方法是一种公开出来的输出接口，通过它可以对 ActiveX 控件进行控制。

与事件相似，ActiveX 方法也包括常用方法和自定义方法两种，常用方法是 COleControl 类所提供的方法，而自定义方法则是由用户自己开发的方法。

MFC 为了能够实现控件的常用方法和自定义方法，采用了两种机制作为支持，一种是 COleControl 类机制，它的成员函数是由 CWnd 派生的，支持所有的 ActiveX 控件的常用方法；另一种就是派遣映射表机制，派遣映射表与消息映射表相似，但派遣映射表使用的不是 Windows 消息 ID，而是虚拟成员函数的映射派遣 ID。派遣映射表的主要目的是建立被外部调用者（例如 ActiveX 控件容器）所使用的方法与实现该方法的控件类的成员函数之间的关系。

为使控件正确支持不同的方法，该控件的类必须声明调度映射，这由位于控件类头文件（.h）中的下列代码行实现：

```
DECLARE_DISPATCH_MAP()
```

调度映射的主要用途是在以下两者之间建立关系：外部调用者（如容器）所使用的方法名和实现

这些方法的控件类的成员函数。声明调度映射后，需要在控件的实现文件（.cpp）中定义此调度映射。下列代码行定义调度映射：

```
BEGIN_DISPATCH_MAP(CSampleCtrl, COleControl)
...
END_DISPATCH_MAP()
```

如果使用 MFC ActiveX 控件向导创建项目，则将自动添加这些代码行；否则必须手动添加这些代码行。

11.6.1　添加常用方法

COleControl 支持两个常用方法：DoClick()和 Refresh()。Refresh()由控件的用户调用，用于立即更新控件的外观；而 DoClick()是用于引发控件的 Click 事件。

常用方法如表 11.3 所示。

表 11.3　　COleControl 支持两个常用方法

方　法	说　明
DoClick()	激发一个 Click 事件
Refresh()	立即更新控件的外观

1．使用添加方法向导添加 Refresh 方法

- 加载控件的项目；
- 在类视图中展开控件的库节点；
- 右击控件的接口节点（库节点的第二个节点）打开快捷菜单；
- 在快捷菜单中选择“Add”→“Add Method”命令，此操作将打开添加方法向导；
- 在“Method name”下拉列表框中，选择“Refresh”选项；
- 单击“Finish”按钮。

2．添加方法向导对常用方法的更改

由于控件的基类支持常用 Refresh()方法，添加方法向导不以任何方式更改控件的类声明。添加方法向导将方法的项添加到控件的调度映射及其.idl 文件。下面的代码行被添加到控件的调度映射中，调度映射位于控件的实现文件（.cpp）中：

```
DISP_STOCKFUNC_REFRESH( )
```

这使控件的用户可以使用 Refresh()方法。

下面的代码行被添加到控件的.idl 文件：

```
[id(DISPID_REFRESH)] void Refresh(void);
```

这行代码为 Refresh()方法分配了一个特定的 ID 号。

11.6.2　添加自定义方法

ActiveX 控件用户可以随时调用自定义方法来执行控件特定的操作，自定义方法的调度映射项为 DISP_FUNCTION 窗体。

1．使用添加方法向导添加自定义方法

下面的过程说明如何将自定义方法 PtInCircle()添加到 ActiveX 控件的主干代码。PtInCircle()确定传递给控件的坐标是在圆的内部还是外部，此过程也可用于添加其他自定义方法。用自定义方法名及其

参数替换 PtInCircle()方法名和参数即可：

- 加载控件的项目；
- 在类视图中展开控件的库节点；
- 右击控件的接口节点（库节点的第二个节点）打开快捷菜单；
- 在快捷菜单中选择“Add”→“Add Method”命令，此操作将打开添加方法向导；
- 在“Method name”下拉列表框中输入“PtInCircle”；
- 在“Internal name”文本框中输入方法的内部函数名或使用默认值（此例中为 PtInCircle）；
- 在“Return type”下拉列表框中设置方法的返回类型为“VARIANT_BOOL”；
- 使用“Parameter type”和“Parameter name”控件，添加一个称为 xCoord 的参数（OLE_XPOS_PIXELS 类型）；
- 使用“Parameter type”和“Parameter name”控件，添加一个称为 yCoord 的参数（OLE_YPOS_PIXELS 类型）；
- 单击“Finish”按钮。

2. 添加方法向导对自定义方法的更改

添加自定义方法时，添加方法向导对控件类头文件（.h）和实现文件（.cpp）进行一些更改。下面的代码行被添加到控件类头文件（.h）中的调度映射：

```
VARIANT_BOOL PtInCircle(OLE_XPOS_PIXELS xCoord, OLE_YPOS_PIXELS yCoord);
```

此代码声明一个称为 PtInCircle()的调度方法处理程序，控件用户可以使用外部名称 PtInCircle 调用此函数。

下面的代码行被添加到控件的 .idl 文件：

```
[id(1), helpstring("method PtInCircle")] VARIANT_BOOL
PtInCircle(OLE_XPOS_PIXELS xCoord, OLE_YPOS_PIXELS yCoord);
```

此行为PtInCircle()方法分配一个特定的ID号，即该方法在添加方法向导方法和属性列表中的位置。由于自定义方法是用添加方法向导添加的，此方法的项自动添加到项目的.idl 文件。

另外，下面的代码行（位于控件类的实现文件（.cpp）中）被添加到控件的调度映射：

```
DISP_FUNCTION(CSampleCtrl, "PtInCircle", PtInCircle, VARIANT_BOOL,
VTS_XPOS_PIXELS , VTS_YPOS_PIXELS)
```

DISP_FUNCTION 宏将 PtInCircle()方法映射到控件的处理函数 PtInCircle()，将返回类型声明为VARIANT_BOOL 并声明传递给PtInCircle()的VTS_XPOS_PIXELS 和VTS_YPOSPIXELS 类型的两个参数。

最后，添加方法向导将存根函数 CSampleCtrl::PtInCircle()添加到控件的实现（.cpp）文件的末尾。为使 PtInCircle()按前面介绍的方式运行，必须对其做如下修改：

```
VARIANT_BOOL CSampleCtrl::PtInCircle(OLE_XPOS_PIXELS xCoord, OLE_YPOS_PIXELS yCoord)
{
    return InCircle(CPoint(xCoord, yCoord));
}
```

11.6.3 从方法返回错误代码

若要指出方法中发生了错误，应使用 COleControl::ThrowError()成员函数，它采用 SCODE（状态代码）作为参数，可以使用预定义 SCODE 或定义自己的 SCODE。ThrowError()只是作为从属性的 Get 或 Set 函数内或自动化方法返回错误的一种手段。只有在这些时候，堆栈上才会存在适当的异常处理程序。

多数常用的预定义 SCODE 都有 Helper 函数，如 COleControl::SetNotSupported()、COleControl::GetNotSupported()和 COleControl::SetNotPermitted()。

11.7　完整 ActiveX 控件范例

下面对一个 Clock 控件的完整实现进行介绍。

11.7.1　创建工程

打开 Visual Studio 2010 后，选择“File”→“New”→“Project”命令，然后按图 11.7 对话框所示内容进行设置，并输入工程名 Clock，单击“OK”按钮。

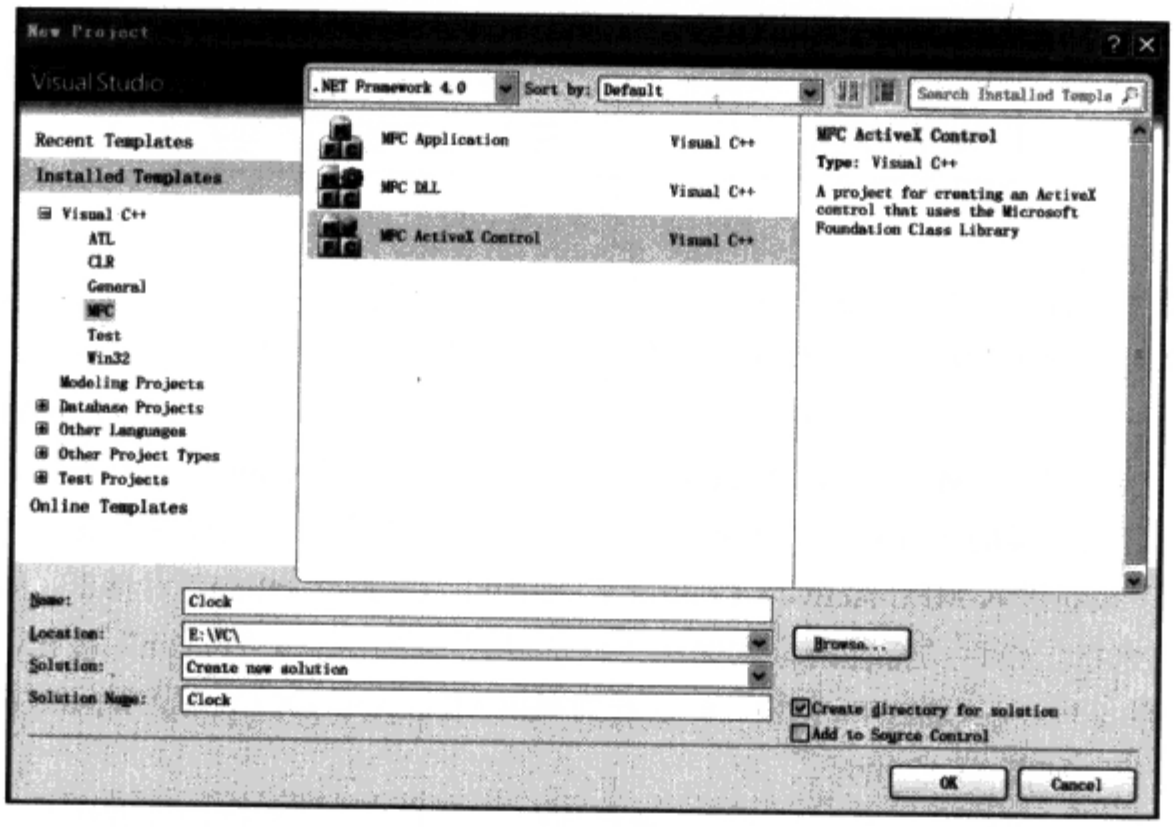


图 11.7　创建工程（1）

现在进入 MFC ActiveX Control Wizard 对话框，如图 11.8 所示。

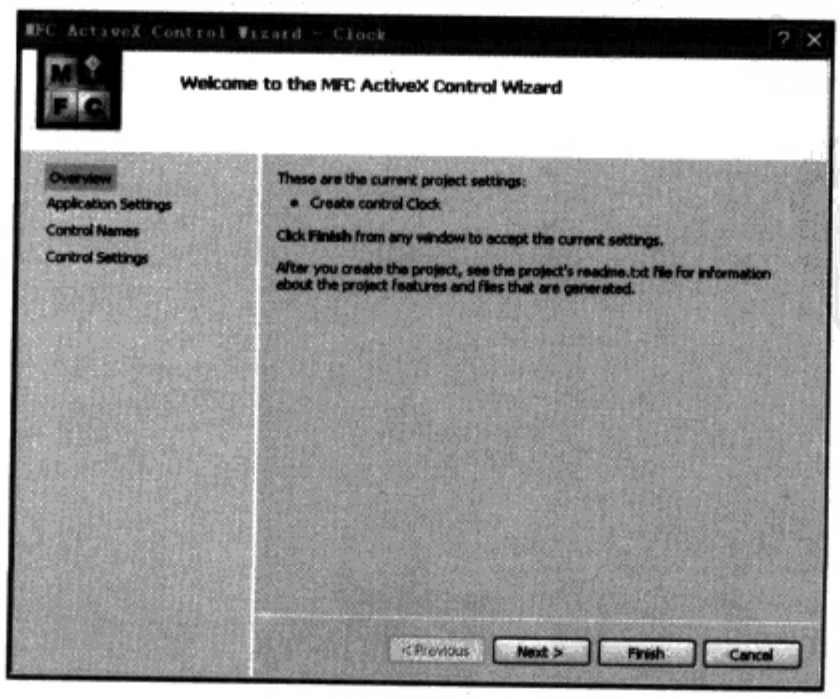


图 11.8　创建工程（2）

单击“Next”按钮进入 MFC ActiveX Control Wizard 的“Application Settings”对话框，如图 11.9 所示。

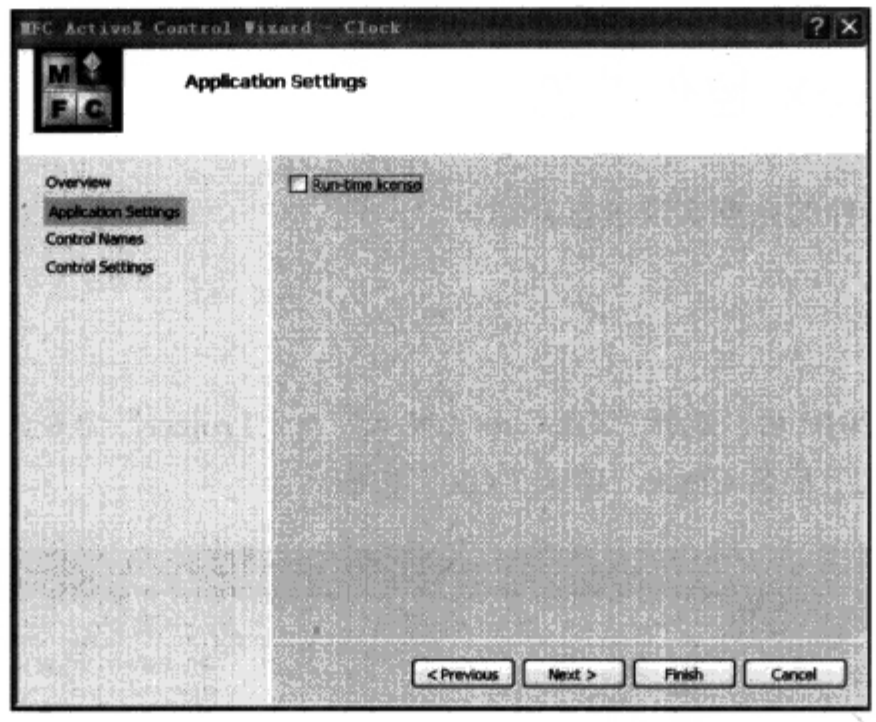


图 11.9　创建工程（3）

在该对话框中，询问用户该工程的控件中是否想要指定运行时许可，开发人员花费大量的时间和精力开发了一个功能强大的控件，当然不希望其他人免费使用这个控件，那么在编写这个控件之前，就可以选择生成一个运行时许可，这样即使其他人拿到这个控件，如果没有许可，也将无法使用。

单击“Next”按钮进入 MFC ActiveX Control Wizard 的“Control Names”对话框，如图 11.10 所示。在该对话框中可以实现类的名称和文件的名称的修改，本例中都保持默认设置不变。

单击“Next”按钮进入 MFC ActiveX Control Wizard 的“Control setting”对话框，如图 11.11 所示，这些复选框是用来指定控件的特性的，本例保持默认设置不变。单击“Finish”按钮，就创建了一个 MFCActiveX 控件工程。在 Class View 中，可以看到利用 MFC ActiveX Control Wizard 向导创建的工程自动生成了 3 个类，如图 11.12 所示。

创建完成之后，可以将工程编译并运行，然后使用上文中介绍的测试控件方法对 Clock 控件进行测试，测试结果与图 11.3 相同。

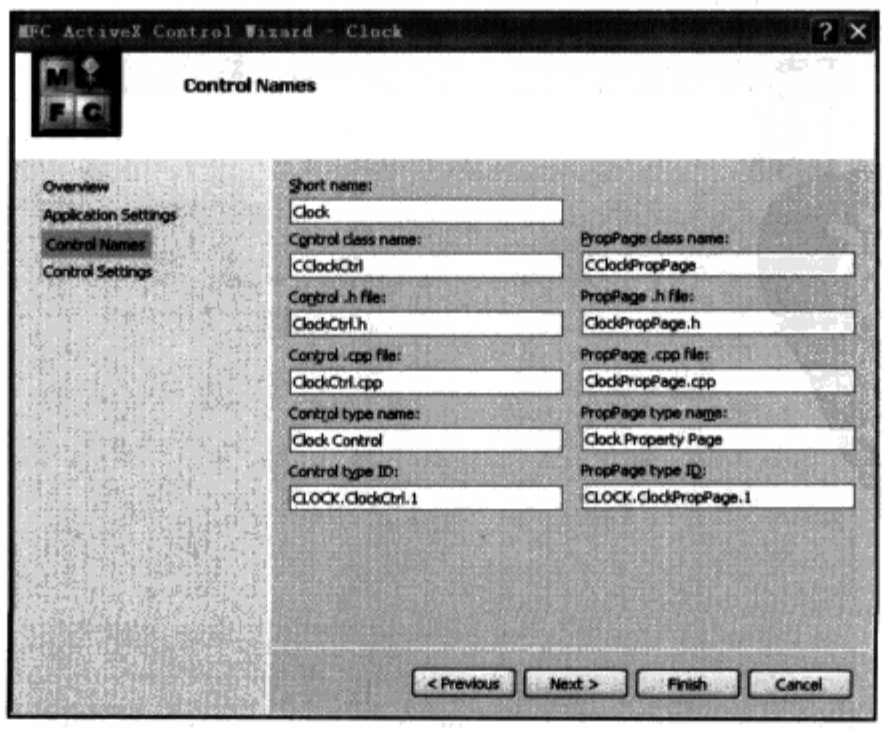


图 11.10　创建工程（4）

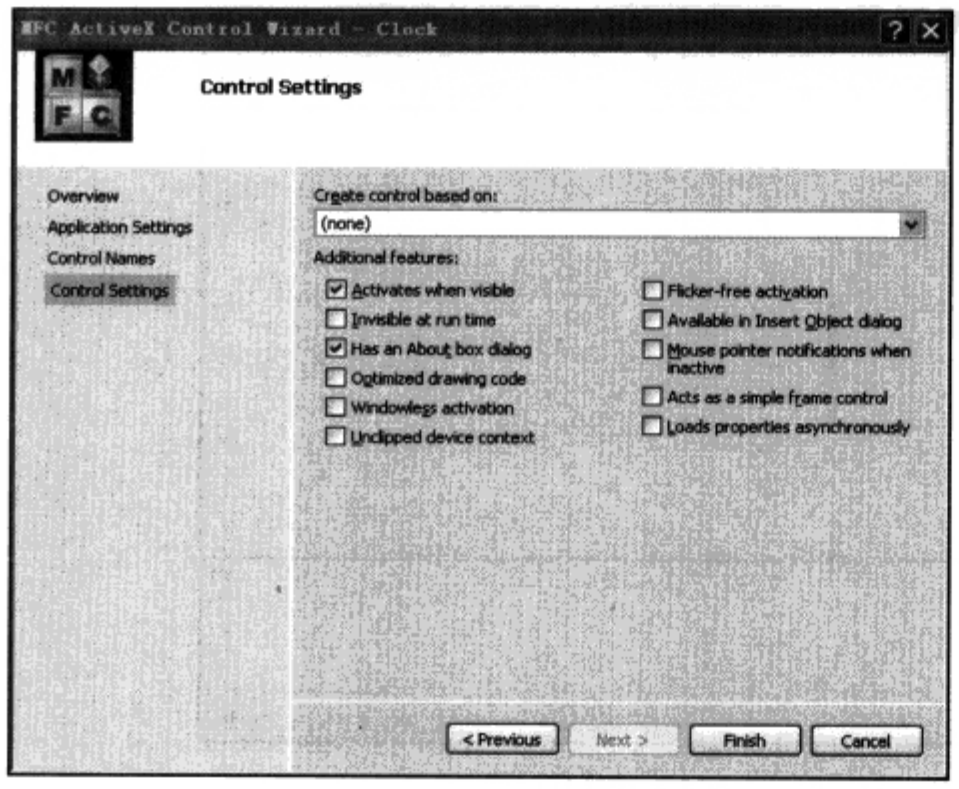


图 11.11　创建工程（5）

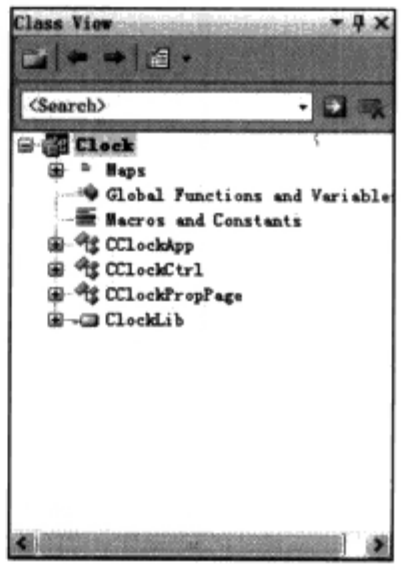


图 11.12　MFC ActiveX Control Wizard 向导创建的工程自动生成了 3 个类

11.7.2　Clock 控件的实现

下面继续完成 Clock 控件的实现，让该控件显示系统当前时间，这可以在 CClockCtrl 类的 OnDraw() 函数中完成。这时，该函数中已经自动生成了两行代码，分别用来填充控件的背景和绘制椭圆，先将这两行代码注释起来，然后添加下面的代码：

```
void CClockCtrl::OnDraw( CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    if (!pdc)
        return;
    CTime time=CTime::GetCurrentTime();
    CString str=time.Format("%H:%M:%S");
    pdc->TextOut(0,0,str);
}
```

如果想要获得当前系统时间，可以使用 CTime 类的静态方法——GetCurrentTime()，该函数将返回表示系统当前时间的 CTime 对象；之后就可以利用 CTime 对象的 Format()方法对得到的 CTime 类的事件进行格式化，返回一个 CString 对象；然后将表示时间的字符串显示在控件的窗口中。当控件需要刷新时，就会调用 OnDraw()函数，并传递一个 CDC*类型的指针变量，在此函数中就可以利用该指针完成窗口的绘制工作。本例利用该指针在控件窗口（0,0）位置显示格式化后得到的时间字符串。

按照上文中介绍的控件测试方法，可得到如图 11.13 所示的测试结果，这时可以看到 Clock 控件的窗口上显示了系统的当前时间。

但是，这时控件上显示的时间是静止的，为了让该时间“动起来”，需要设置一个定时器，使其每隔 1s 发送一个 WM_TIMER 消息，在响应该定时器的消息处理函数中，让控件刷新，重新输出当前系统时间。为此，需要 CClockCtrl 类增加 WM_CREATE 消息的处理函数，然后在此函数中，在控件窗口创建完成之后，调用 SetTimer()函数创建定时器。因此，为了使控件响应 WM_CREATE 和 WM_TIMER 事件，要在 CTimeCtrl 类中添加 WM_CREATE 和 WM_TIMER 消息响应函数 OnCreate() 和 OnTimer()。具体代码如下：

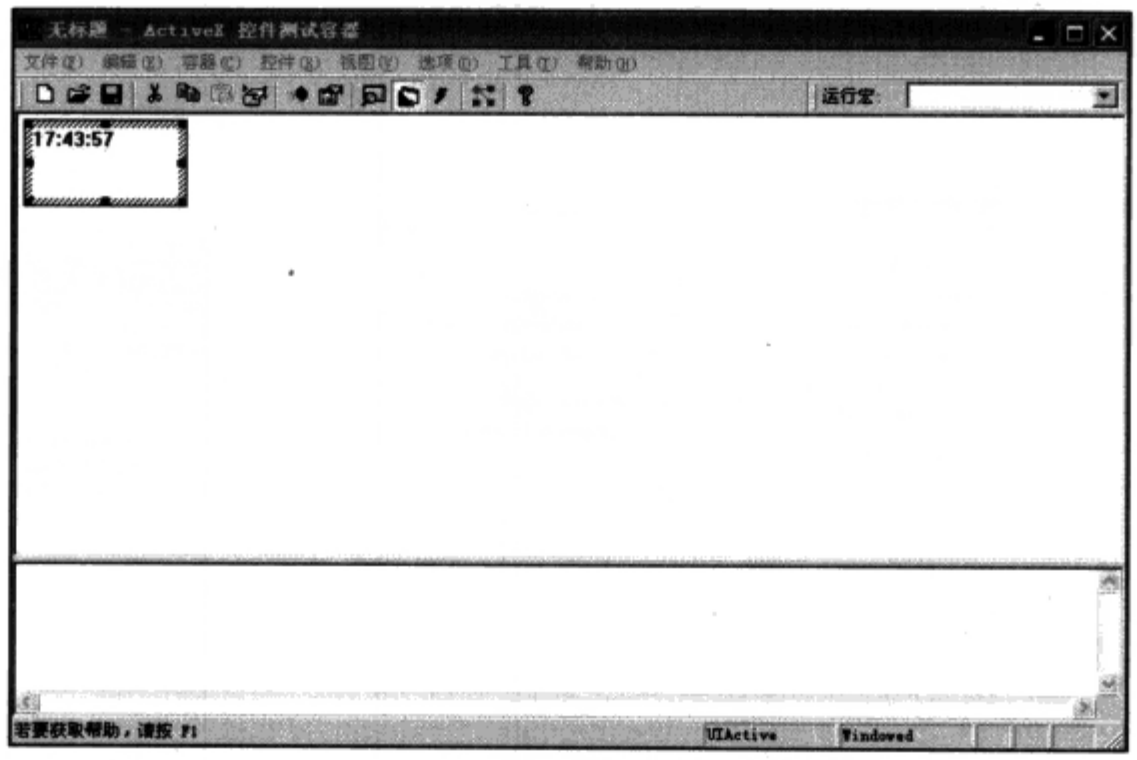


图 11.13　控件窗口上显示系统当前时间

```
int CClockCtrl::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (COleControl::OnCreate(lpCreateStruct) == -1)
        return -1;
    // TODO:  Add your specialized creation code here
    SetTimer(1,1000,NULL); //设置计时器
    return 0;
}
void CClockCtrl::OnTimer(UINT_PTR nIDEvent)
{
    // TODO: Add your message handler code here and/or call default
    InvalidateControl();  //实际上是调用 CTimeCtrl::OnDraw()函数
    COleControl::OnTimer(nIDEvent);
}
```

编译并运行 Clock 控件，在随后打开的 ActiveX 控件测试容器应用程序中插入该控件，将会看到这时这个 Clock 控件显示的时间随系统当前时间而变化了。

11.7.3　添加常用属性

可以通过上文中介绍的方法为控件添加常用属性，下面对添加背景色和前景色两个常用属性进行详细介绍。

首先，打开类试图，展开以 Lib 结尾的选项，然后选中以_D 开头的选项，如图 11.14 所示。右击并选择“Add”→“Add Property”命令，然后添加常用属性，如图 11.15 所示。

在图 11.15 中选择了 BackColor 常用属性，其他保持默认设置，然后单击“Next”按钮，进入添加属性向导“IDL Attributes”对话框，如图 11.16 所示。

在这一步中，同样采取默认设置，并单击“Finish”按钮，完成属性添加。添加前景色 ForeColor 的方法与添加背景色的方法相同。如果这样重新对控件进行编译并使用 ActiveX 控件测试容器对控件进行测试。在插入的控件上右击，选择“Properties”命令弹出属性对话框，但对话框中只有一个 General 选项卡，找不到刚添加的两个常用属性。现在就要给控件的属性对话框上添加一个“颜色”选项卡，

要实现此功能，就要修改类 CClockCtrl 的.cpp 文件中的属性页映射宏，在宏 BEGIN_PROPPAGEIDS (CTimeCtrl, 1)和宏 END_PROPPAGEIDS(CtimeCtrl)之间再添加一个属性页 CLSID_CColorPropPage，相应的 BEGIN_PROPPAGEIDS 宏的参数中的计数也要加 1，结果如下：

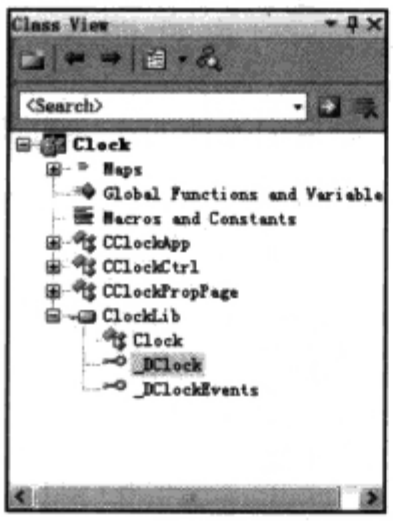


图 11.14　类试图

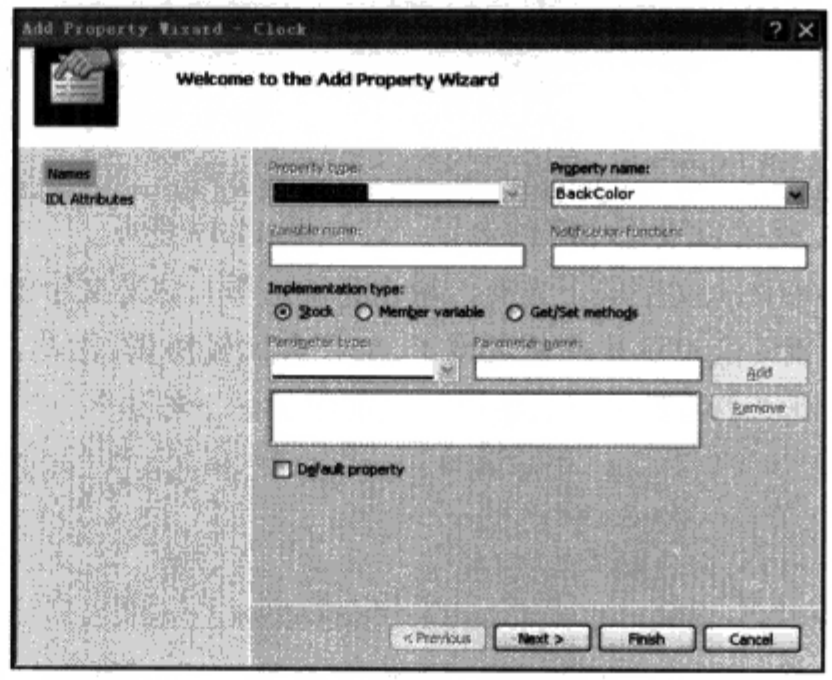


图 11.15　添加常用属性

```
// TODO: Add more property pages as needed.  Remember to increase the count!
BEGIN_PROPPAGEIDS(CClockCtrl, 2)
    PROPPAGEID(CClockPropPage::guid)
    PROPPAGEID(CLSID_CColorPropPage)
END_PROPPAGEIDS(CClockCtrl)
```

现在再运行，右击控件选择“Properties”命令，在控件属性对话框的第二个页面中可以看到前面加的颜色属性，如图 11.17 所示。

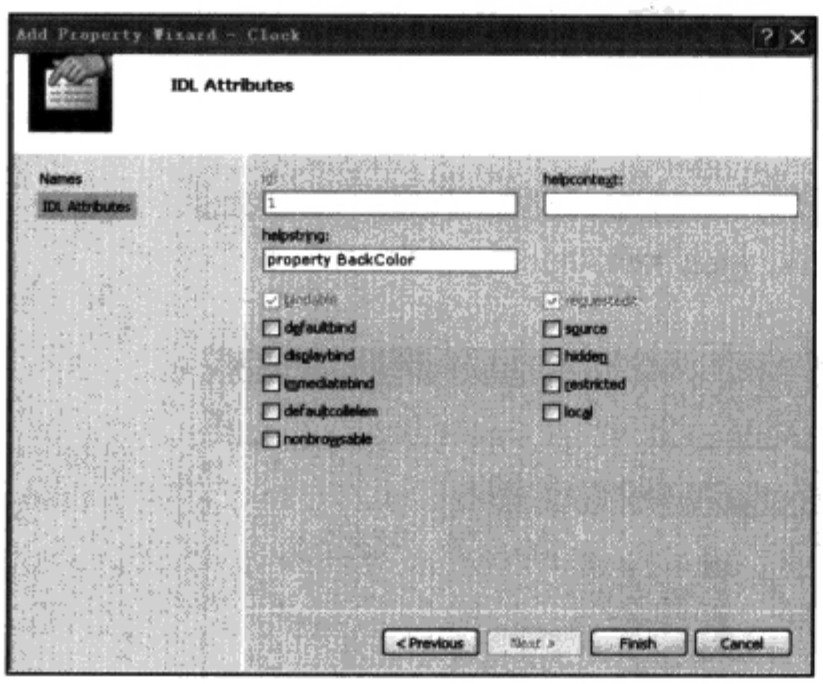


图 11.16　BackColor 常用属性

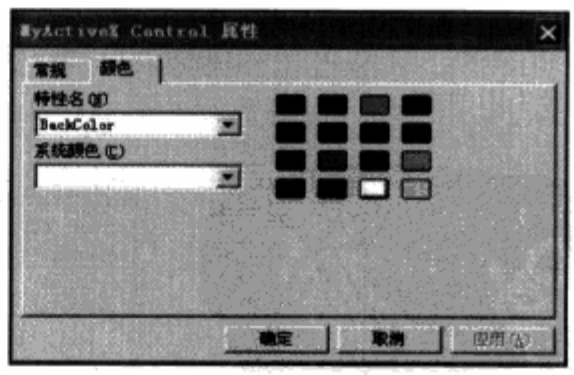


图 11.17　控件选择 Properties

但是，当设置两个颜色的属性时，发现窗体中 Clock 控件并没有发生相应的改变，为了实现这一功能，还需要在 CClockCtrl 类中的 OnDraw()函数中编写一些代码，如下所示：

```
void CClockCtrl::OnDraw(
            CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    if (!pdc)
```

```
        return;
    CBrush brush(TranslateColor(GetBackColor()));
    pdc->FillRect(rcBounds,&brush);
    pdc->SetTextColor(TranslateColor(GetForeColor()));
    CTime time=CTime::GetCurrentTime();
    CString str=time.Format("%H:%M:%S");
    pdc->TextOut(0,0,str);
}
```

这时，就可以进行相应的设置了，如图 11.18 所示。

Clock 控件上文字的下方有一块白色的区域，这是文字的背景色。如果把控件的文字的背景设置为透明模式，就不会有这块白色区域了，效果会比较协调。因此，关闭测试容器，回到工程中，在 CClockCtrl 类中的 OnDraw()函数中第五条语句之前添加如下的语句，将文字的背景设置为透明模式。

```
CBrush brush(TranslateColor(GetBackColor()));
pdc->FillRect(rcBounds,&brush);
pdc->SetBkMode(TRANSPARENT);
pdc->SetTextColor(TranslateColor(GetForeColor()));
```

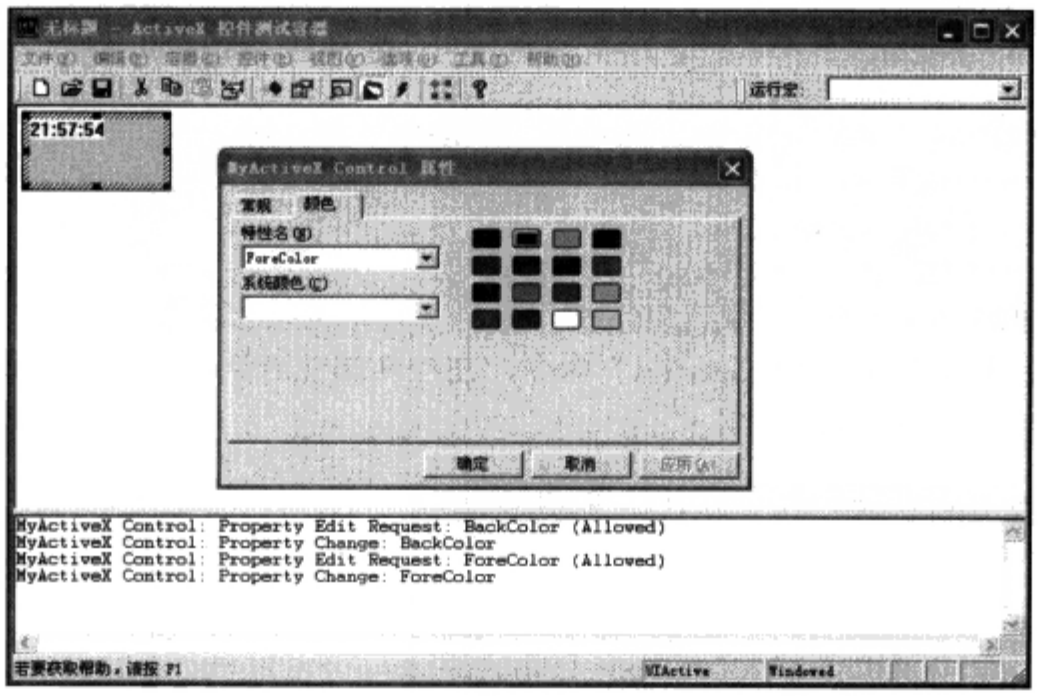


图 11.18　Clock 控件（1）

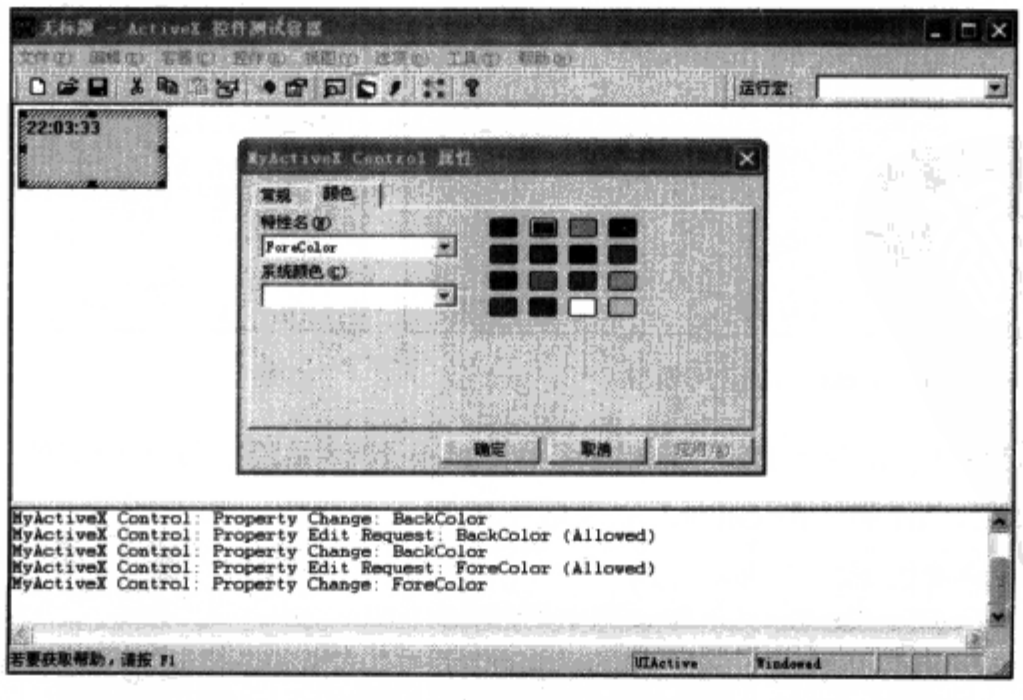


图 11.19　Clock 控件（2）

11.7.4　添加自定义属性

下面将添加一个新的自定义属性：时间间隔（UpdateInterval）。用户设置该属性的值以后，Clock 就会按照用户指定的时间间隔值来更新显示的时间。

与添加常用属性相同，弹出添加属性向导后，在“Property name”下拉列表框中输入“UpdateInterval”，“Property type”设置为 SHORT 类型。添加属性向导自动添加了一个通知函数 OnUpdateInterval Changed()，如图 11.20 所示。

单击“Finish”按钮即可完成自定义属性 UpdateInterval 的添加，这时在类视图选项卡上，可以看到_DClock 接口中又增加了一个属性——UpdateInterval，并且在 CclockCtrl 类中增加了一个函数——OnUpdateIntervalChanged()。当 UpdateInterval 这一外部属性被修改时，就会调用这个 OnUpdateInterval Changed()函数。该函数的默认实现代码如下所示：

```
void CClockCtrl::OnUpdateIntervalChanged(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    // TODO: Add your property handler code here
    SetModifiedFlag();
}
```

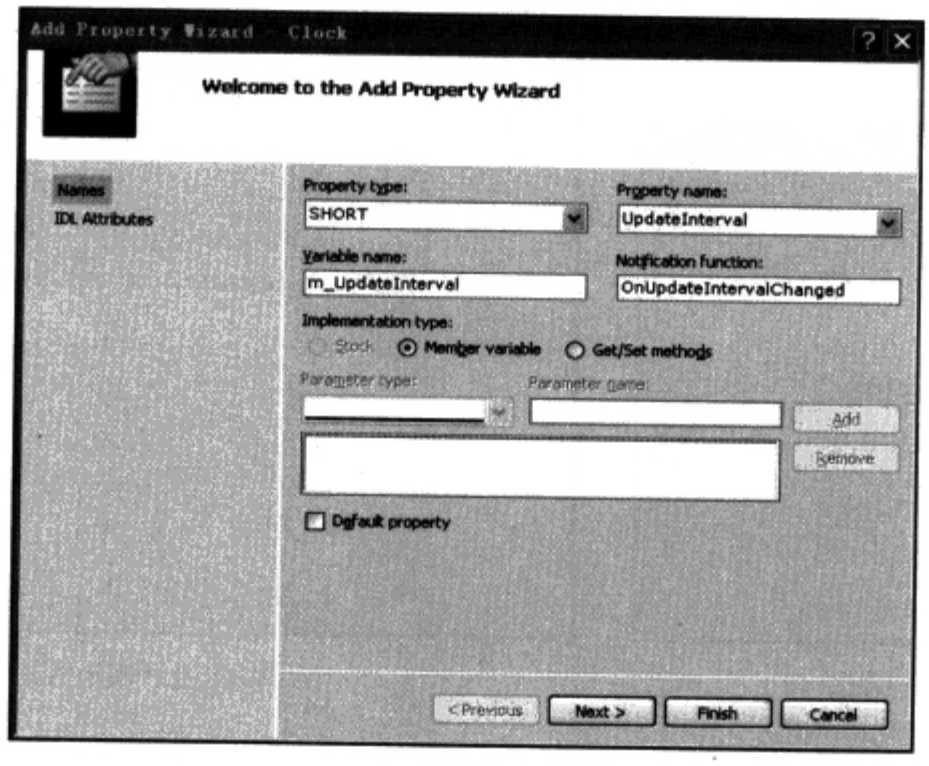


图 11.20　添加自定义属性

另外，可以发现 CClockCtrl 类提供了一个成员变量：m_UpdateInterval。接下来，就在 OnUpdate IntervalChanged()函数中根据用户输入的时间间隔值控制 Clock 控件的显示更新。具体代码如下：

```
void CClockCtrl::OnUpdateIntervalChanged(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    // TODO: Add your property handler code here
    //解决输入值不合要求的问题
    if(m_UpdateInterval<1000||m_UpdateInterval>5000)
        m_UpdateInterval=1000;
    //解决输入值不是整数的问题
    m_UpdateInterval=m_UpdateInterval/1000*1000;
    KillTimer(1);
```

```
        SetTimer(1,m_UpdateInterval,NULL)
        SetModifiedFlag();
    }
```

编译生成最新的 Clock 控件，然后利用 ActiveX 控件测试容器测试该控件。在选择“Edit”→“Insert New Control”命令插入该控件后，为了测试控件的属性，需要选中该控件，然后选择“Control”→“Invoke Methods”命令，这时将显示如图 11.21 所示的对话框。

如果想要得到某个属性值，应该选择 PropGet 类型的方法；如果想要设置某个属性的值，则应该选择 PropPut 类型的方法。这里，想要设置 Clock 控件的 UpdateInterval 属性的值，因此选择 UpdateInterval(PropPut)项，并在随后出现的 Parameter 编辑框中输入数值 2000，单击“Set Value”按钮，就把 UpdateInterval 属性的值设为 2000 了，但是这个属性值此时仍未生效，需要单击【Invoke】按钮才可以。之后就会发现 Clock 控件显示的时间每隔 2s 跳动一次，说明设置生效了。

11.7.5 添加方法

下面为 Clock 控件添加一个自定义方法。与添加属性使用同样的方法，在类试图的_Dclock 选项上右击，选择“Add”→“Add Method”命令，这时将打开如图 11.22 所示的对话框。

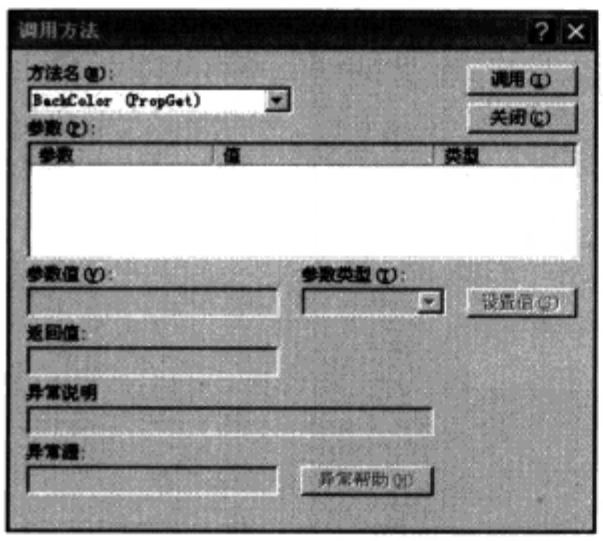


图 11.21 添加方法（1）

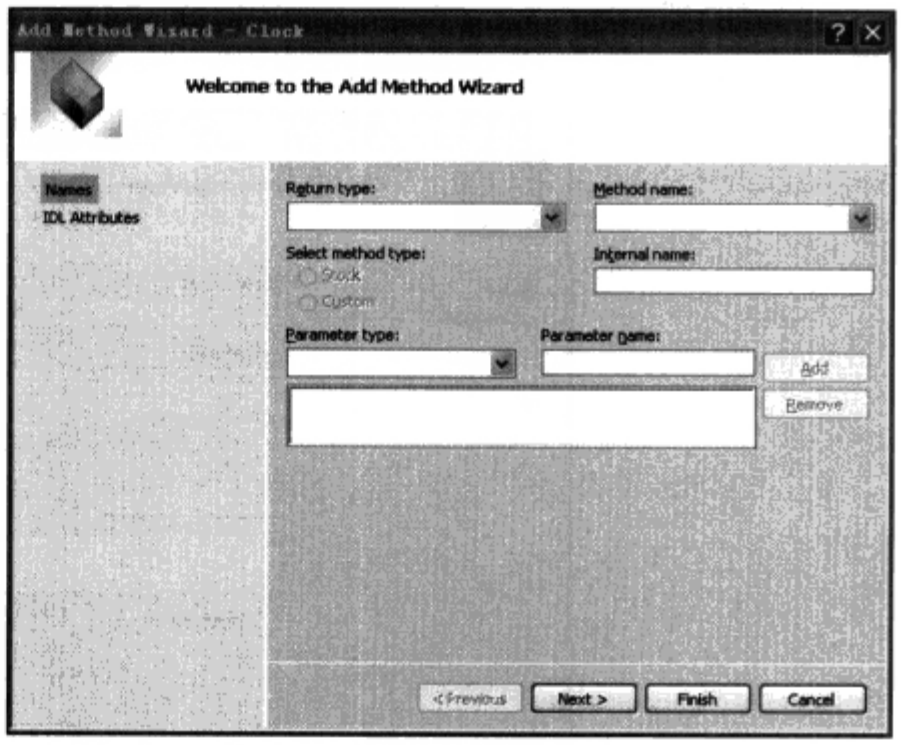


图 11.22 添加方法（2）

该对话框中提供了几个选项，其中 Method name 是给外部程序使用控件的方法时使用的，这里将其设置为 Hello。可以看到，系统自动为该方法提供了一个内部名称（Internal name）“Hello”，这个内部名称是在控件内部使用的方法名称，它可以与外部名称不一样。然后将返回类型（Return type）设置为 void，不必给这个方法设置参数，结果如图 11.23 所示。

单击“Finish”按钮完成添加方法的操作，这时在类视图中，可以看到在_DClock 接口下增加了一个方法——Hello()，同时在 CClockCtrl 类中提供了该方法的实现。在此方法中，可以使用 MessageBox()函数显示一个消息对话框，具体代码如下：

```
void CClockCtrl::Hello(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    // TODO: Add your dispatch handler code here
    MessageBox(_T("Hello World!"));
}
```

编译程序，生成最新的 Clock 控件，再次利用 ActiveX 控件测试容器测试该控件。在该容器中调用控件方法的步骤是：选中 Clock 控件，选择“Control”→“Invoke Methods”命令，这时将打开“Invoke Methods”对话框，在此对话框的“Method name”下拉列表框中选择“Hello”选项，然后单击“Invoke”按钮，就会调用 Clock 控件的 Hello()方法，显示如图 11.24 所示的消息对话框。

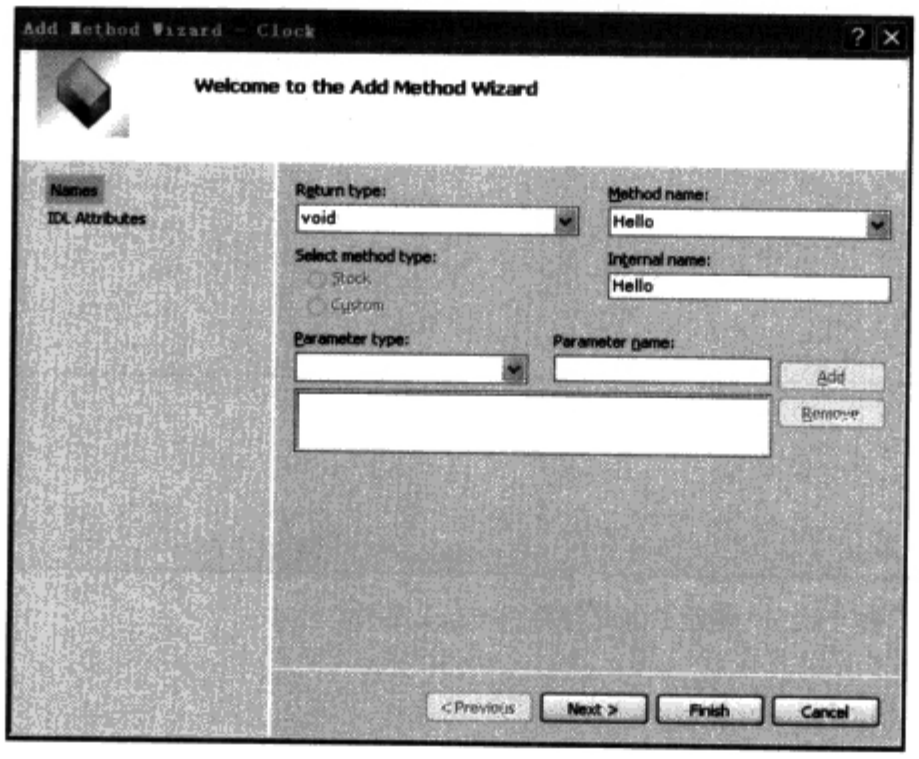


图 11.23　添加方法（3）

图 11.24　消息框

11.7.6　添加常用事件

在类视图中，右击 Cclock 选项，然后选择“Add”→“Add Event”命令，将显示添加事件向导对话框，如图 11.25 所示。在“Event name”下拉列表框中选择“Click”选项，保持其他不变，单击“Finish”按钮关闭添加事件向导对话框。这时，在类视图中可以看到，在_DClockEvents 接口下面增加了一个方法 Click，该方法就是刚刚添加的 Click 事件。再次利用 ActiveX 控件测试容器测试该控件。插入 Clock 控件后，在此控件上单击，这时在该容器下面的窗格中可以看到这样一句话“Clock Control: Click”，即触发了 Clock 控件的 Click 事件，如图 11.26 所示。

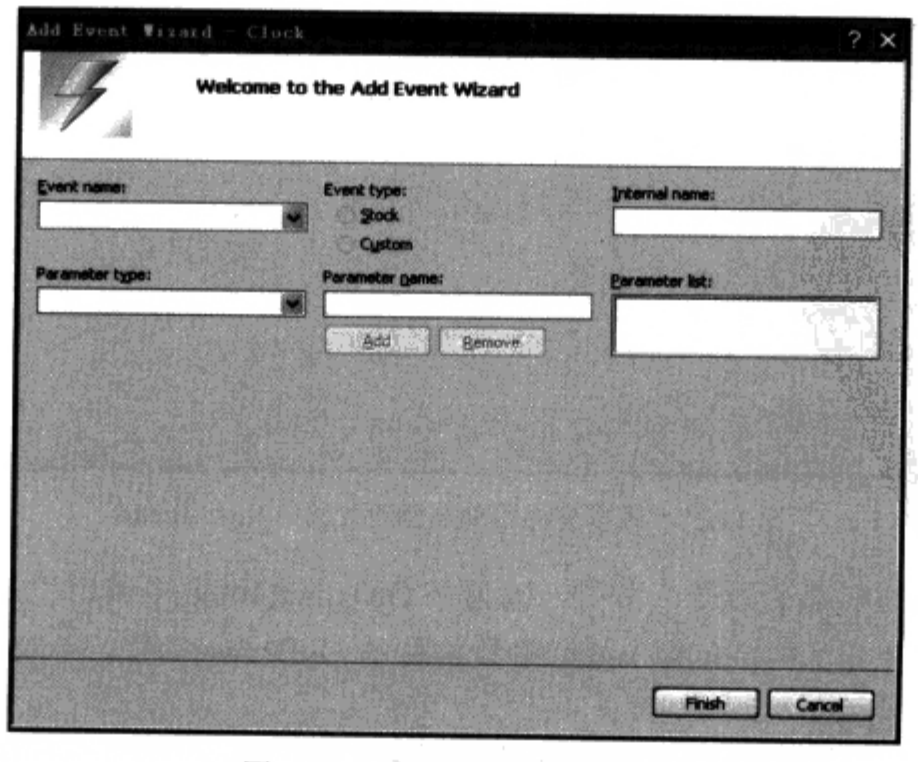


图 11.25　显示添加事件对话框

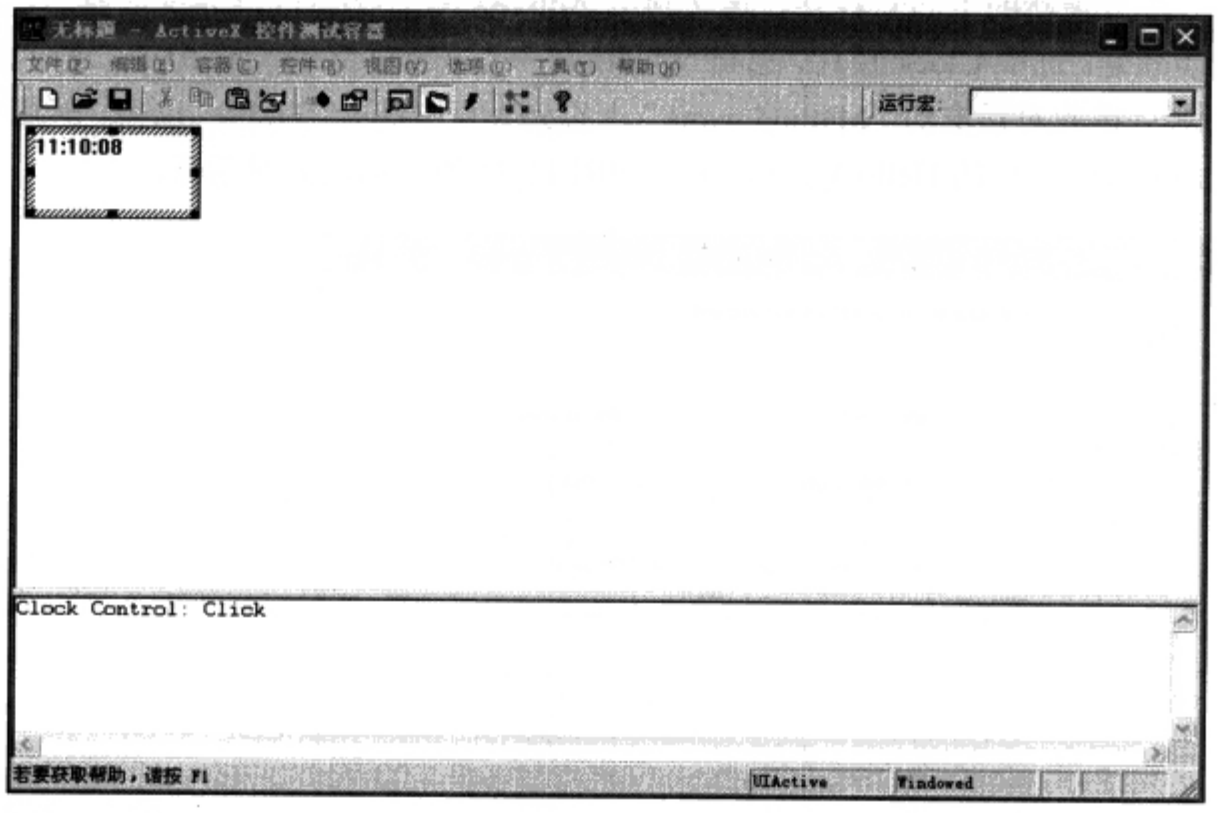


图 11.26　触发 Clock 控件的 Click 事件

11.7.7　添加自定义事件

给这个时钟添加一个 NewMinute 事件，与添加常用方法一样，打开添加事件向导对话框，并输入事件的名称（Event name）“NewMinute”，系统自动将改事件的内部名称设置为“NewMinute”，结果如图 11.27 所示。

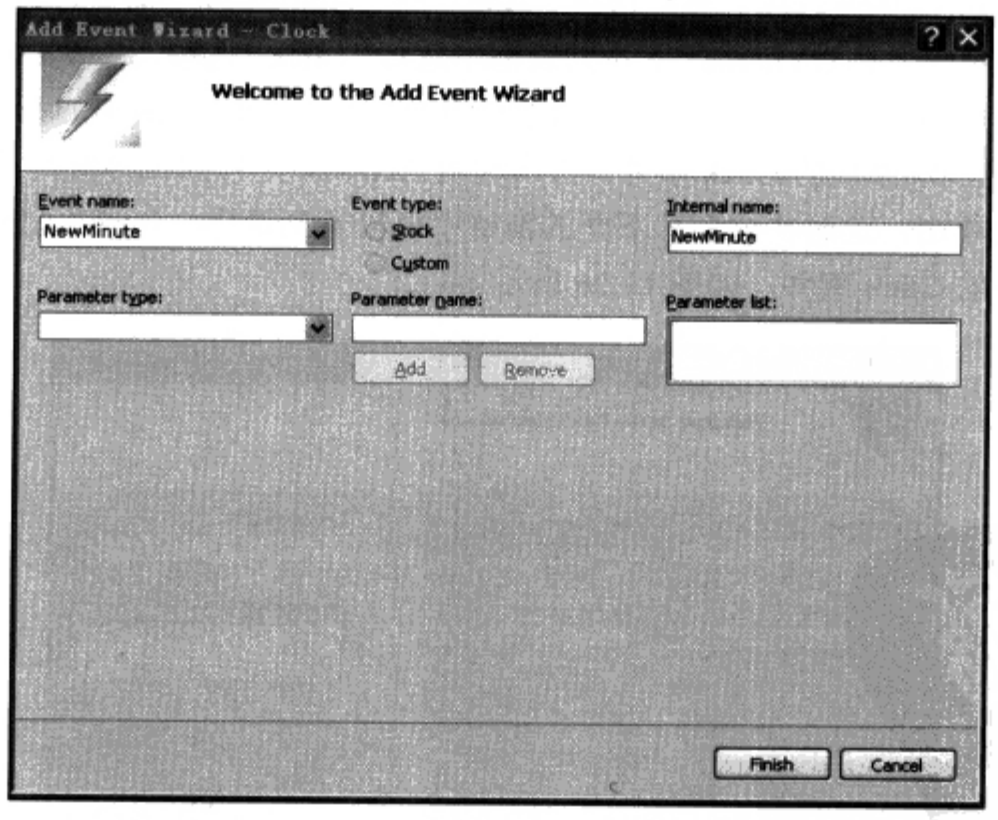


图 11.27　改事件的内部名称设置为：NewMinute

接下来，就要判断时钟是否过了一分钟。因为在 OnTimer()函数中调用了 InvalidateControl();语句，即每过一秒（或用户修改后的时间间隔）就会触发控件的 OnDraw()函数，所以可以在 OnDraw()函数中判断是否过了一分钟，如下：

```
void CClockCtrl::OnDraw(
```

```
                CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    if (!pdc)
        return;
    CBrush brush(TranslateColor(GetBackColor()));
    pdc->FillRect(rcBounds,&brush);
    pdc->SetBkMode(TRANSPARENT);
    pdc->SetTextColor(TranslateColor(GetForeColor()));
    CTime time=CTime::GetCurrentTime();
                                                    //判断是否过了一分钟
    if(time.GetSecond()==0)
    {
        NewMinute();                                //触发 NewMinute 事件
    }
    CString str=time.Format("%H:%M:%S");
    pdc->TextOut(0,0,str);
}
```

然后利用 ActiveX 控件测试容器测试该控件。插入该控件后，可以看到一旦该控件显示的时间到达新的一分钟，该容器下面的窗格中就会显示这样一句话“Clock Control: NewMinute”，即触发了一个 NewMinute 事件，如图 11.28 所示。

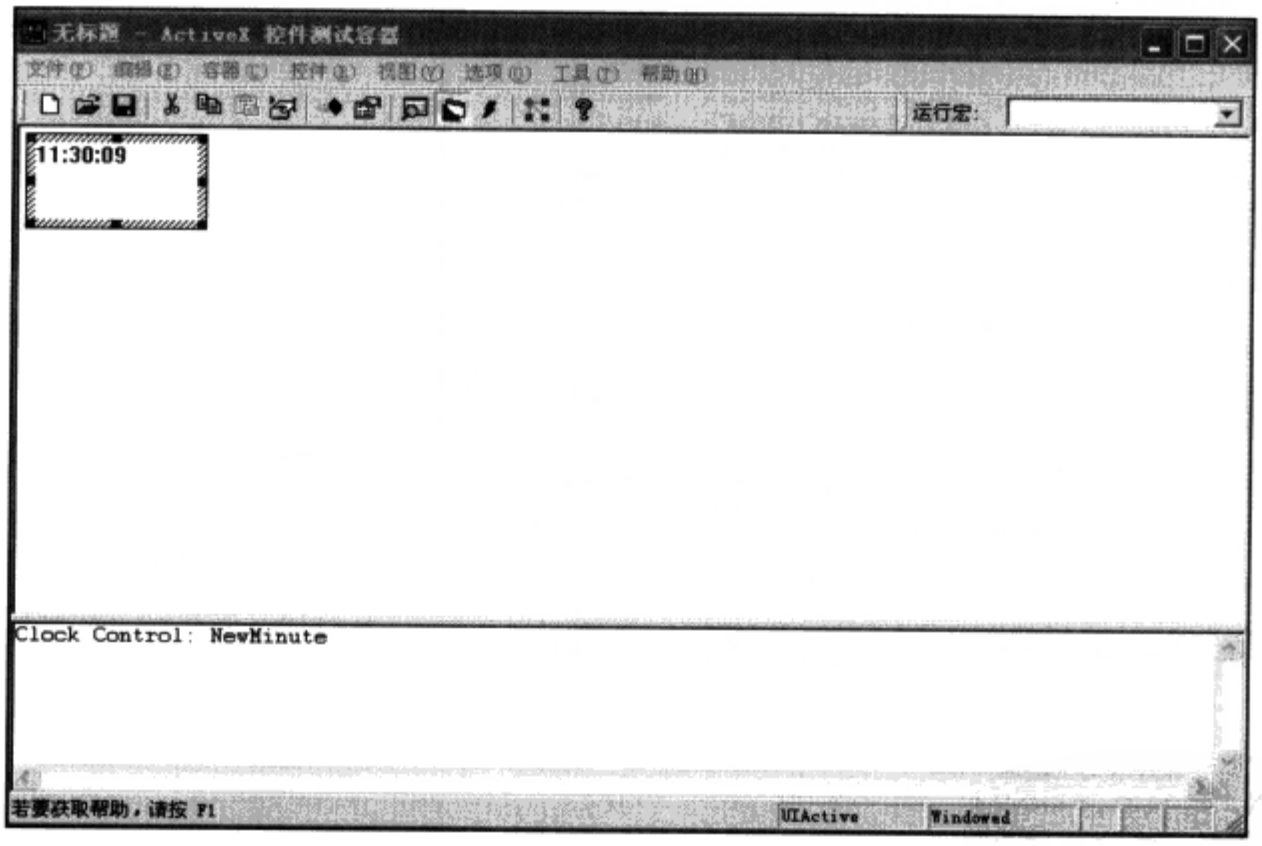


图 11.28　添加一个 NewMinute 事件

11.8　调用 ActiveX 控件

下面使用 Visual Studio 2010 编写一个 ActiveX 控件的测试程序。新建一个基于对话框的工程，命名工程为“ClockTest”，并删除自动生成的对话框资源上的静态文本控件。在 Visual Studio 2010 开发环境中，在对话框资源上添加一个 ActiveX 控件的方法是：在对话框资源上右击，从弹出的快捷菜单中选择“Insert ActiveX Control”命令，这时将显示如图 11.29 所示的对话框，在此对话框中找到 Clock

控件并选中，单击“OK”按钮关闭该对话框即可。

这时，在对话框资源上就插入了 Clock 控件，可以通过属性面板设置其属性，如图 11.30 所示。运行程序即可查看结果。

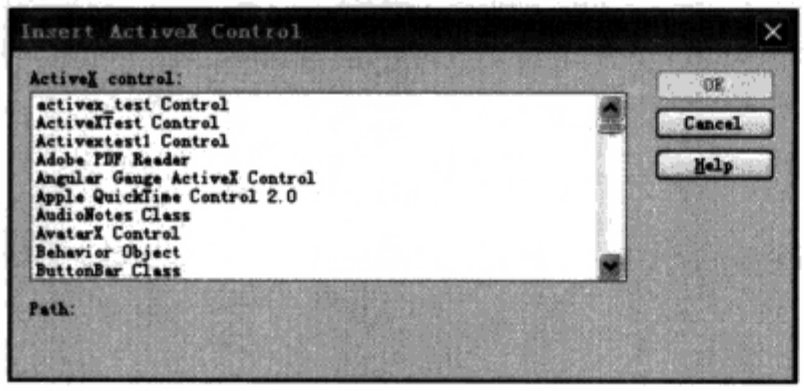


图 11.29　调用 ActiveX 控件

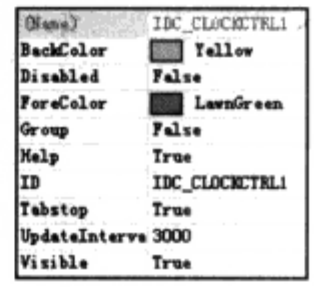


图 11.30　插入 Clock 控件

在 Visual Studio 2010 中，插入 ActiveX 控件的另一种方法是，选择“Tools”→“Choose ToolBox Items”命令，然后选择“COM Components”选项卡，选择要插入的控件，单击“OK”按钮即可，如图 11.31 所示。

这样，在工具箱中就增加了 Clock 控件的图标，如图 11.32 所示。

图 11.31　插入 ActiveX 控件

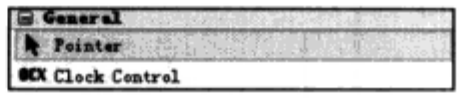


图 11.32　插入 ActiveX 控件

上面介绍的两种方法都可以完成对控件的插入及使用。

11.9　小结

本章主要分两部分详细介绍了 Visual C++ 2010 MFC ActiveX 控件方面的知识，以及实现 ActiveX 控件的方法。

在第一部分中介绍了有关 ActiveX 控件技术的一些基本知识，并对 ActiveX 控件的 3 种特性——属性、方法和事件进行了详细的介绍。这一部分是开发 ActiveX 控件的理论依据。

在第二部分中，主要通过一个简单的例子介绍了 ActiveX 控件的开发方法，这个例子虽然简单，但是覆盖了 ActiveX 控件开发的大部分技术特点，起到了抛砖引玉的作用。

如果读者希望了解关于 ActiveX 控件的更多知识，可以查看 MSDN 中相关的内容。

第 12 章　Visual C++ 2010 MFC 文件与注册表操作

本章主要介绍了如何使用 Visual C++ 2010 MFC 实现对文件与注册表的操作。首先介绍了如何通过 Visual C++ 2010 MFC 提供的 Serialize()函数和 CArchive 类来实现串行化操作；接着介绍了 MFC 提供的支持文件操作的基类 CFile 类的组成成员和使用方法；然后又说明了 Windows 的系统配置文件.ini 文件的读写方法；最后又向读者介绍了 Windows 操作系统的一个重要组成部分——注册表，以及如何使用 Visual C++ 2010 MFC 实现对注册表的读写操作。完成本章的学习之后，读者将对 Visual C++ 2010 MFC 文件与注册表操作有一定的了解，并且能够编写应用程序来实现对 Windows 文件和注册表的读写。

本章主要内容有：

- 文本操作串行化;
- CFile 类介绍;
- .ini 文件读写操作;
- 注册表读写操作。

12.1　文本操作串行化

使用前面所学的内容，可以创建一个完善的视图文档，但是由于这些视图文档对象只是存活于内存中，其被销毁后，信息就转瞬即逝，因此必须将这些文档存储在硬盘上或者打印出来。在基于 MFC 的程序中，文档并非一个简单的实体，它可以是非常复杂的类结构，通常包含各种对象，而且这些对象又有可能包含其他对象，这个结构可能包含很多层次。

我们希望能把这些文档保存在文件中，但是将类对象写入文档必然会存在很多问题，这是因为类对象不同于整数或字符这样的基本数据项，可以连续存入文件并根据已知字节大小读取这些数据。即使将一个对象的所有数据成员连续写入一个文件，想要恢复原始对象也是非常困难的。必须写入包含完整的类结构的数据信息，还必须设定读取规则，以便根据文件中的数据能够完整恢复原始对象。

为了解决这个问题，MFC 支持一种称为串行化的机制，串行化是指将对象存储到介质（如文件、内在缓冲区等）中或是以二进制方式通过网络传输。之后，可以通过反串行化从这些连续的字节数据重新构建一个与原始对象状态相同的对象。简单来讲，让对象数据持久化的过程就称为串行化（或称为序列化）。

串行化的基本思想是任何可串行化的类都必须负责存储和检索自己，即要使类成为可串行化的，它们就必须能够将自己写入文件，并且再从文件中读出，这也意味着要使一个类成为可串行化的，用于声明该类数据成员的所有类的类型也必须是可串行化的。

12.1.1　文档类 Serialize()函数

虽然文档串行化是一个困难的过程，但是串行化文档的基本功能已经完全由 Application Wizard 内

置到应用程序中。所生成的 MFC 应用程序的“保存“和“打开”菜单命令都假定文档已经实现了串行化，并且已经包含了支持串行化的代码。下面新建一个 MFC 单文档工程文件，然后来看一下 CFileDoc 类的代码：

```
class CFileDoc : public CDocument
{
protected:                          // 仅从序列化创建
    CFileDoc();
    DECLARE_DYNCREATE(CFileDoc)
                                    // 属性
public:
                                    // 操作
public:
                                    // 重写
public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
                                    // 实现
public:
    virtual ~CFileDoc();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
protected:
                                    // 生成的消息映射函数
protected:
    DECLARE_MESSAGE_MAP()
};
```

其中阴影标出来的 3 个部分是与串行化文档对象有关的内容：

首先是默认的构造函数；然后是 DECLARE_DYNCREATE 宏，用来在串行化输入过程中，支持应用程序架构动态地创建 CFileDoc 类对象。在类实现过程中，还有一个互补宏 IMPLEMENT_DYNCREATE()与其配合使用，这些宏只应用于 CObject 派生的类。对于所有要串行化的类来说，CObject 都必须是直接或间接基类，这是因为它将添加支持串行化操作的功能。几乎所有的 MFC 类都派生于 CObject 类，因此它们都是可串行化的。

最后，需要重点关注的是虚函数 Serialize()的声明。每个可串行化的类都必须包括这个函数，它被调用时将对 CFileDoc 类的数据成员执行输入和输出串行化操作。下面研究这个函数的定义：

```
                                    // CFileDoc 序列化
void CFileDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
                                    // TODO: 在此添加存储代码
    }
    else
    {
                                    // TODO: 在此添加加载代码
    }
}
```

首先，该函数有一个 CArchive 引用类型的参数。关于该类型的详细说明，将在下一节提到，这里读者只需知道该参数传递给这个函数的 CArchive 类型的对象确定将要发生的操作是输入还是输出。

在此函数中，首先判断 CArchive 对象 ar 的当前状态，如果是存储状态，则在其后增加相应的存储代码；否则在 else 子块中增加加载数据的代码。注意，这里必须对每个想要实现串行化的类重写 Serialize()函数，这样，Application Wizard 才能知道文档中包含什么数据，怎样对这些数据进行存储和读取操作。读写这些信息的过程全部依赖于开发人员。

下面，看一段重写 Serialize()函数的代码：

```
                                    // CFileDoc 序列化
void CFileDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
                                    // TODO: 在此添加存储代码
        int i=10;
        char ch='c';
        double d=10.0;
        CString str("Visual C++ 2010 程序设计入门到精通");
        ar<<i<<ch<<d<<str;//保存数据
    }
    else
    {
                                    // TODO: 在此添加加载代码
        int i=0;
        char ch=' ';
        double d=0.0;
        CString str("");
        ar>>i>>ch>>d>>str;//读取数据
        CString result;
                                    //保存对加载后得到的数据进行格式化后的字符串
        result.Format(_T("%d,%c,%f,%s"),i,ch,d,str);
        AfxMessageBox(result);//弹出消息框
    }
}
```

上面的代码其实就是在设定存储操作的区域定义了一个整型变量、一个字符变量、一个双字节浮点数、一个字符串，并为它们赋值，然后利用 CArchive 类重载的插入操作符保存这些数据。接着在加载区域中使用 CArchive 类重载的加载操作符重新加载这些变量，并将其保存到一个新的字符串变量中，使用 AfxMessageBox()函数弹出消息对话框来显示加载的数据。

运行程序，单击“保存”按钮，将文档保存为 file（如图 12.1 所示）。此时，所定义的 4 个变量应该被保存到 file 文件中了，然后立即使用打开按钮，打开刚才保存的 file 文件。这时，程序并没有如所设定的那样弹出显示加载结果的消息对话框，这是因为在程序运行过程中，当保存数据时，先前创建的文档类的对象就与该数据关联在一起了。再次打开刚保存的文件时，在 MFC 框架内部，它判断出这是同一份数据，并且和这个数据相关联的文档类对象的指针已经存在了，就不会再去调用文档类的 Serialize()函数了，所以 Serialize()函数中的加载代码就没有运行，自然也就不会弹出消息对话框。这样做的目的是避免重复打开同一份文档，提高程序运行效率。

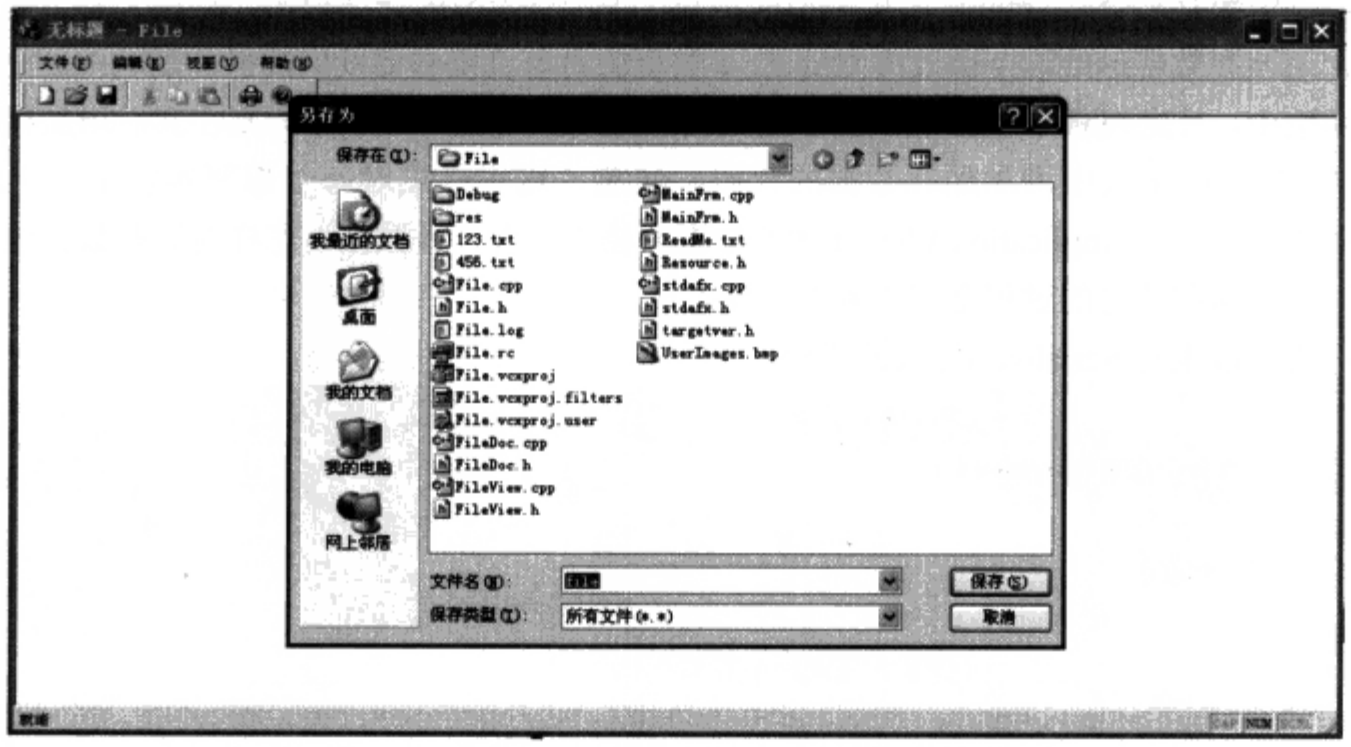


图 12.1　文档类 Serialize 函数（1）

这样，单击“新建”按钮，新建一个文件，然后再选择打开刚才保存的 file 文件，这时程序弹出消息对话框，并且显示所保存的数据，如图 12.2 所示。

图 12.2　文档类 Serialize 函数（2）

12.1.2　CArchive 对文件进行读写

上一小节数次提到 CArchive 类，下面详细介绍该类。CArchive 类是发动串行化机制的引擎。C++ 中的流操作在控制台程序中从键盘读取数据，然后写入屏幕，CArchive 类则提供了基于 MFC 的流操作。读者可以认为 CArchive 对象就是一个二进制流，像一个输入/输出流一样，一个 CArchive 对象与一个文件相关，并允许带缓冲机制的数据写入与读取。与输入/输出流不同的是，后者处理的是 ASCII 码字符序列，而一个 CArchive 对象则以一种有效的、非冗余的格式处理二进制对象数据。

下面介绍 CArchive 类的成员，如表 12.1 所示。

表 12.1　CArchive 类的组成成员

数 据 成 员	
m_pDocument	指向被串行化的 CDocument 对象
构造函数	
Carchive()	创建一个 CArhcive 对象
Abort()	在不异常的情况下，关闭归档文件
Close()	冲掉未写入数据并且释放与 CFile 的连接
基础输入/输出	
Flush()	从归档文件缓冲区中冲掉未写入数据
operator >>	装载对象和归档文件的主要类型
operator <<	存储对象和归档文件的主要类型
Read()	读入原始类型
Write()	写入原始类型
WriteString()	写一行文本
ReadString()	读一行文本
状　　态	
GetFile()	获得此归档文件的 CFile 对象指针
GetObjectSchema()	由 Serialize()函数调用来确定被非串行化的对象的版本
SetObjectSchema()	在归档文件中存储对象概要
IsLoading()	确定归档文件是否被装载
IsStoring()	确定归档文件是否被存储
IsBufferEmpty()	确定在一个 Windows Socket 接收过程中缓冲区是否被清空
对象输入/输出	
ReadObject()	调用一个用于装载的 Serialize()函数
WriteObect()	调用一个用于装载的 Serialize()函数
MapObject()	在没有对文件串行化的映射中放置对象，但是此映射对参考的子对象有效
SetStoreParams()	设置 Hash 表的大小和映射的块的大小，在串行化的过程中识别惟一的对象
LoadParams()	设置装载数组扩展的大小。必须在被装载对象之前或调用 MapObject()或 ReadObject()之前
ReadClass()	读入一个原先存储在 WriteClass 中的类的参考
WriteClass()	把对 CRuntime 的参考写入 CArchive
SerializeClass()	根据 CArchive 方向，读取或写入对 CArchive 对象的类的参考

关于 CArchive 类的每个成员函数的详细使用方法，将在具体使用时再做说明，如果读者有兴趣，可以参考 MSDN 帮助文档。

在使用 CArchive 对象时，需要注意几点情况：首先是 CArchive 对象有一个与其相关联的 CFile 对象，因此，在创建 CArchive 对象之前必须先创建一个 CFile 类或其派生类对象，并且因为存档对象既可以用来加载数据，也可以用来保存数据，所以必须确保这个 CFile 类对象的打开方式与该存档（CArchive）对象的加载/保存状态相一致。一个文件（CFile 对象）只能与一个活动的存档对象相关联。这里的 CFile 对象将在下一小节做详细介绍。

下面将学习如何使用 CArchive 类实现对文件进行读写操作。

在上一节所建的 MFC 工程 File 中的主菜单上添加一个新的菜单“File”，然后为其添加两个新的菜单命令“read”和“write”。为这两个菜单命令在 CFileView 类中添加两个命令响应函数 OnFileRead()和 OnFileWrite()。然后在响应函数 OnFileWrite()中添加如下代码：

```
void CFileView::OnFileWrite()
{
```

```
    // TODO: Add your command handler code here
                                                    //构造 CFile 文件对象
    CFile file(_T("file.txt"),CFile::modeCreate|CFile::modeWrite);
                                                    //构造存档对象
    CArchive ar(&file,CArchive::store);
    int i=10;
    char ch='c';
    double d=10.0;
    CString str("Visual C++ 2010 程序设计入门到精通");
    ar<<i<<ch<<d<<str;                              //保存数据
}
```

下面详细分析一下上面的代码。第一行代码构造了一个 CFile 文件对象来与 CArchive 对象关联，关于 CFile 类的使用方法将在下节说明，这里只需要知道该方法创建了名为 file.txt 的文件。

接着，构造了一个 CArchive 对象。CArchive 类的构造函数声明如下：

```
CArchive(CFile* pFile,UINT nMode,int nBufSize=4096,void* lpBuf=NULL);
```

该函数的参数说明如表 12.2 所示。

表 12.2　　CArchive 的参数说明

参　数	说　明
pFile	CFile 对象的指针。CFile 对象是持久数据的最终的源或目标
nMode	标识，它指定了对象是否从存档对象中装载或存储到文件中。nMode 参数必须是下列值之一： • CArchive::load()从存档对象装载数据，CFile 只读。 • CArchive::store()把数据保存到存档对象中，允许 CFile 写操作。 • CArchive::bNoFlushOnDelete()当存档对象析构程序被调用时，防止存档对象自动调用 Flush()。如果设定了此标识，则在析构程序被调用之前必须负责调用 Close；否数据就会崩溃
nBufSize	指定内部文件缓冲区大小的整数，以字节计算。注意，默认的缓冲区大小为 4096 字节。如果例程存档大的对象，使用大一些的缓冲区，即多个文件缓冲区，将会提高例程的执行效率
lpBuf	指向 nBufSize 大小的缓冲区的指针。如果不指定这个参数，存档对象从本地堆为存档对象分配一个缓冲区并且当对象被毁弃时，释放缓冲区。存档对象不能释放一个提供的缓冲区

该函数构造 CArchive 对象并且指定它将被用于装载对象还是存储对象。在创建存档对象之后，不能改变这个指定内容，不能使用 CFile 操作来改变文件的状态直到已经关闭存档对象。任何这样的操作都将会毁坏存档对象的完整性。

这里将构造函数的第一个参数设为刚刚创建的 CFile 对象，第二个参数使用 CArchive::store 标识，最后两个参数使用默认值。

然后，参考上一节的内容，定义一个整型变量、一个字符变量、一个双字节浮点数、一个字符串并为它们赋值。

最后，使用 CArchive 类重载的插入操作符（<<）将上面创建的数据保存到文件中。CArchive 类重载的提取和插入操作符（<<和>>）对派生于 CObject 的类的对象及大量基本数据类型分别进行输入和输出操作。对于对象中的基本数据类型，可以使用提取和插入运算符串行化数据。注意，这里使用提取和插入操作符必须确定输入和输出的顺序一致。

接着再在响应函数 OnFileRead()中添加如下所示的代码：

```
void CFileView::OnFileRead()
{
    // TODO: Add your command handler code here
    CFile file(_T("file.txt"),CFile::modeRead);
                                    //构造存档对象
```

```
        CArchive ar(&file,CArchive::load);
        int i=0;
        char ch=' ';
        double d=0.0;
        CString str;
        ar>>i>>ch>>d>>str;          //读取数据
        CString result;
                                    //保存对加载后得到的数据进行格式化后的字符串
        result.Format(_T("%d,%c,%f,%s"),i,ch,d,str);
        AfxMessageBox(result);//弹出消息对话框
    }
```

这里还是先创建一个 CFile 对象来打开刚刚修改的 file.txt 文件，只不过因为是要读取文件，所以使用 modeRead 模式；然后再构造一个 CArchive 对象，并使用 load 标识来读取第一个参数所指向的文件指针；接着定义一个整型变量、一个字符变量、一个双字节浮点数、一个字符串来接收从文件中读取的数据；接下来使用 CArchive 类重载的提取操作符（>>）来提取文件中被串行化的数据。注意，这里的读取顺序一定要和保存顺序一致。最后，使用弹出消息对话框来显示提取的内容。

编译并运行程序，当显示应用程序界面后，首先选择“File”→“write”命令，如图 12.3 所示，这时，会发现该程序所在的文件夹中多了一个“file.txt”文件，这就是将数据串行化后所写入的文件。再选择“File”→“read”命令，程序就会从文件“file.txt”中读取数据然后通过弹出消息对话框来显示读取的内容，如图 12.4 所示。

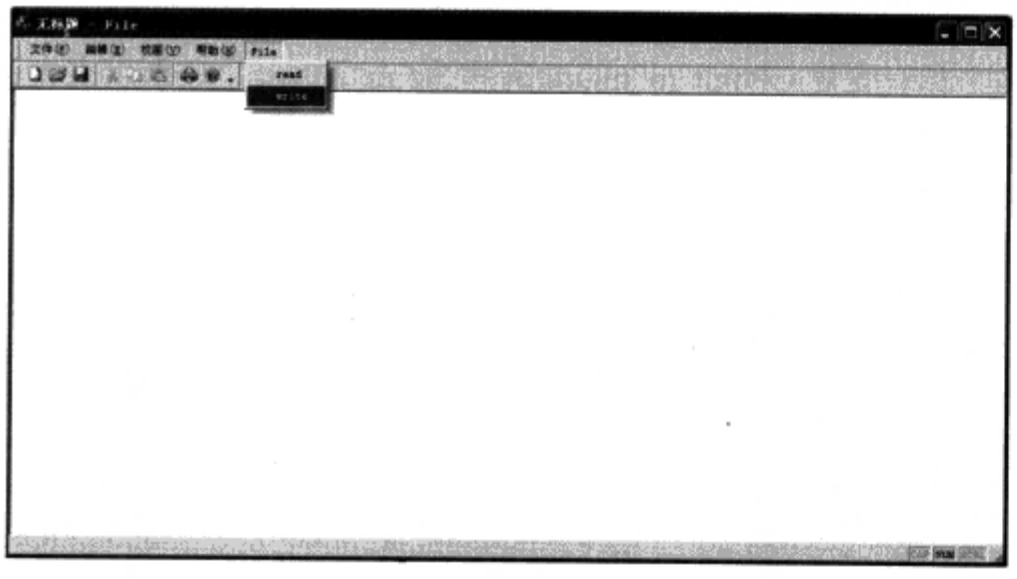


图 12.3　单击 File 菜单中的 write 子菜单（1）

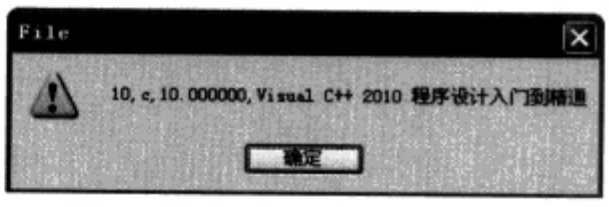


图 12.4　File 菜单中的 write 子菜单（2）

12.1.3　文档操作串行化代码分析

了解串行化机制中的两个重要部分——Serialize()函数和 CArchive 类，并且利用它们将 CString 类等 MFC 提供的类串行化并保存后，下面再来学习一下如何利用它们实现对自己定义的类和创作的文档进行串行化。

实现一个类的串行化通常需要以下几个步骤。

（1）从 CObject 或其派生类派生出用户的类。

（2）在类声明文件中，加入 DECLARE_SERIAL 宏，编译时，编译器将扩充该宏，这是串行化对象所必需的。

DECLARE_SERIAL 宏声明：

```
DECLARE_SERIAL( class_name)
```

其中，参数 class_name 是想要成为可串行化类的类名。

（3）重载 Serialize()成员函数，加入必要的代码，用于保存对象的数据成员到 CArchive 对象，以及从 CArchive 对象载入对象的数据成员状态。

（4）定义一个不带参数的构造函数。

（5）在实现文件中加入 IMPLEMENT_SERIAL 宏：

```
IMPLEMENT_SERIAL( class_name, base_class_name, wSchema )
```

其中，参数 class_name 是想要成为可串行化类的类名，base_class_name 是其基类的名称，wSchema 是版本号。

下面通过一个实例来详细介绍如何将自己定义的类串行化。在第 9 章中，学习了如何使用 MFC 实现绘图程序，但是并没有介绍如何将用户绘制的图像保存下来，下面将实现这个过程。

首先，创建一个单文档 MFC 应用程序，命名为 GraphFile，在该应用程序的菜单栏中新加入一个菜单“绘图”，然后添加 4 个菜单命令“点”、“线”、“矩形”、“椭圆”，分别为其设置 ID 号和消息响应函数如表 12.3 所示。

表 12.3　　点、线、矩形、椭圆的 ID 号和消息响应函数

菜单命令名称	菜单命令 ID	菜单命令响应函数
点	ID_GRAPH_DOT	OnGraphDot
线	ID_GRAPH_LINE	OnGraphLine
矩形	ID_GRAPH_RECTANGLE	OnGraphRectangle
椭圆	ID_GRAPH_ELLIPSE	OnGraphEllipse

在 CGraphFileView 类中添加以下两个私有变量 CPoint m_ptOrigin 和 int m_nType，分别用来表示绘图的起点和绘图的类型，并在构造函数中将它们初始化为 0。

然后在相应的消息响应函数里添加如下代码：

```
void CGraphFileView::OnGraphDot()
{
    // TODO: Add your command handler code here
    m_nType=1;
}
void CGraphFileView::OnGraphLine()
{
    // TODO: Add your command handler code here
    m_nType=2;
}
void CGraphFileView ::OnGraphRectangle()
{
    // TODO: Add your command handler code here
    m_nType=3;
}
void CGraphFileView::OnGraphEllipse()
{
    // TODO: Add your command handler code here
    m_nType=4;
}
```

这段代码将 m_nType 设置为不同的值，以告诉程序用户想要绘制什么样的图形。下面再在 CGraphFileView 类中添加鼠标按下和松开的两个消息响应函数，并在其中添加用于实现图像的绘制的

代码如下：

```
void CGraphFileView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    m_ptOrigin=point;                                    //记录鼠标按下的点
    CView::OnLButtonDown(nFlags, point);
}
void CGraphFileView::OnLButtonUp(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CClientDC dc(this);
    switch(m_nType)
    {
    case 1:
        dc.SetPixel(point,RGB(255,0,0));                 //画点
        break;
    case 2:
        dc.MoveTo(m_ptOrigin);                           //画线
        dc.LineTo(point);
        break;
    case 3:
        dc.Rectangle(CRect(m_ptOrigin,point));           //画矩形
        break;
    case 4:
        dc.Ellipse(CRect(m_ptOrigin,point));             //画椭圆
        break;
    default:
        break;
    }
    CView::OnLButtonUp(nFlags, point);
}
```

至此，就完成了应用程序的绘图功能，并为应用程序添加了一个可序列化的类 CGraph。注意，一定要选择该类的基类是 CObject，否则无法对其进行串行化。如图 12.5 所示。

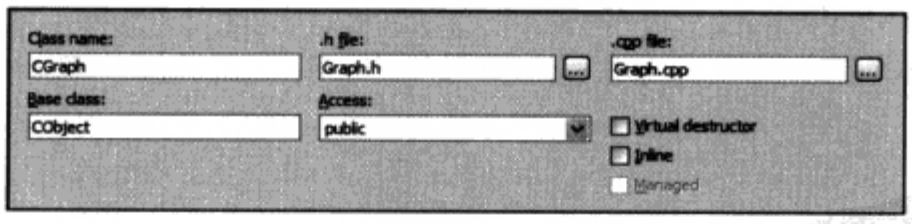


图 12.5　添加类 CGraph

然后进入第（2）步，在该类声明文件中，加入 DECLARE_SERIAL 宏，代码如下：

```
class CGraph :
    public CObject
{
public:
    CGraph(void);
public:
    ~CGraph(void);
    DECLARE_SERIAL( CGraph )
};
```

第（3）步，为 CGraph 类添加函数 Serialize()，如下所示：

```
public:
    void Serialize(CArchive& ar);
```

第（4）步，定义一个不带参数的构造函数，在 CGraph 类生成的时候，系统已经自动添加了一个 CGraph 类的不带参数的构造函数。下面直接进入第(5)步，在 CGraph.cpp 中加入 IMPLEMENT_SERIAL 宏，代码如下：

```
IMPLEMENT_SERIAL(CGraph,CObject,1)
CGraph::CGraph(void)
{
}
```

使 CGraph 类可以串行化的工作已经完成，下面具体实现该类的串行化。首先，在 CGraph 类中添加 3 个变量——m_ptOrigin、m_ptEnd、m_nType，分别用来保存用户所画图像的类型、起点和终点。代码如下：

```
public:
    CPoint m_ptOrigin;          //记录起始点
    CPoint m_ptEnd;             //记录终点
    int m_nType;                //记录画图类型
```

为该类添加一个带有 3 个参数的构造函数，用来初始化刚刚创建的 3 个变量，代码如下：

```
CGraph::CGraph(int m_Type, CPoint m_ptStart, CPoint m_ptTo)
{
    m_nType=m_Type;
    m_ptOrigin=m_ptStart;
    m_ptEnd=m_ptTo;
}
```

在该构造函数中，用传递到构造函数的画图类型、起点和终点 3 个参数来初始化 Cgraph 类中相应意义的 3 个变量。

有了画图的类型、起点和终点，下面要实现为该类添加一个图形绘制函数 Draw()，从而将图形数据和图形绘制封装在一个类中。这样，当视图类需要重绘图形时，只需要向该类发送一条消息，告诉它要绘制自己即可。Draw()函数的实现代码如下所示：

```
void CGraph::Draw(CDC *pDC)
{
    switch(m_nType)
    {
    case 1:
        pDC->SetPixel(m_ptEnd,RGB(255,0,0));        //画点
        break;
    case 2:
        pDC->MoveTo(m_ptOrigin);                    //画线
        pDC->LineTo(m_ptEnd);
        break;
    case 3:
        pDC->Rectangle(CRect(m_ptOrigin,m_ptEnd)) ;//画矩形
        break;
    case 4:
        pDC->Ellipse(CRect(m_ptOrigin,m_ptEnd));    //画椭圆
        break;
```

```
        default:
            break;
        }
    }
```

至此，CGraph 类已经构建完毕，下面使用可串行化的类的 Serialize()函数来实现保存和加载对象。

首先，需要在文档类 CGraphFileDoc 中定义一个成员变量，用于保存每一个图形对象。

现在已经有了一个可以用来表示用户所绘图形的类 CGraph，接下来就要将这些对象保存下来。用户每画一个图形，就产生一个这样的图形对象，因此，这个对象是动态的、不断增长的。MFC 中的 CObArray 类是一个对象数组类，它可以动态调整自己的大小以适应放在其中的元素的个数，可以存放从 CObject 类派生出的任何对象（如前面的 CGraph 对象），其大小只受系统的内存空间的限制。MFC 中其他动态数组类包括 CStringArray、CByteArray、CWordArray、CDWordArray、CptrArray，它们的不同之处在于存放的对象的类型的不同，具体介绍请大家参考 MSDN 帮助文档。

下面在 CGraphFileDoc 类中添加一个 CObArray 类型的变量 obArray。若要将图形对象保存到对象数组 obArray 中，首先要得到对象的绘图类型和起点、终点，然后创建一个新的图形对象，并将其加入对象数组 obArray 中。这个过程可以在图像生成的时候完成，即当用户按下鼠标左键，拖动出一个图形，最后松开鼠标的时候实现，因此将 CGraphFileView 类的 OnLButtonUp()函数中代码做如下修改：

```
void CGraphFileView::OnLButtonUp(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CClientDC dc(this);
    switch(m_nType)
    {
    case 1:
        dc.SetPixel(point,RGB(255,0,0));                    //画点
        break;
     case 2:
        dc.MoveTo(m_ptOrigin);                              //画线
        dc.LineTo(point);
        break;
    case 3:
        dc.Rectangle(CRect(m_ptOrigin,point));              //画矩形
        break;
    case 4:
        dc.Ellipse(CRect(m_ptOrigin,point));                //画椭圆
        break;
    default:
        break;
    }
    CGraph *pGraph=new CGraph(m_nType,m_ptOrigin, point);
    GetDocument()->obArray.Add(pGraph);
    CView::OnLButtonUp(nFlags, point);
}
```

在阴影部分的代码中，首先声明一个 CGraph 类的指针对象，并且用该类的带参数的构造函数来构建这个对象，构造函数中的参数就是 OnLButtonUp()函数中用于画图的类型和起点、终点。注意，这里使用 CGraph 类，必须引入其头文件 CGraph.h。然后将该对象通过 obArray.Add()保存到对象数组中。由于对象数组是在文档类中定义的，在视图类中不能直接引用，因此前面需要调用 GetDocument()函数

来取得访问文档类的权利。

将用户绘制的所有图像对象保存后，还需要将它们依次绘制出来，因此再在 CGraphFileView 类的 OnDraw()函数中添入如下代码：

```
void CGraphFileView::OnDraw(CDC* pDC)
{
    CGraphFileDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;
    // TODO: 在此处为本机数据添加绘制代码
        if (pDoc->obArray.GetSize())
    {
        for(int i=0;i<pDoc->obArray.GetSize();i++)
        {
            ((CGraph*)pDoc->obArray.GetAt(i))->Draw(pDC);
        }
    }
}
```

在阴影部分的代码中，首先判断数组对象 obArray 中是否含有元素，即是否有图形需要绘制，这通过 CObject 类的 GetSize()函数来实现。如果没有，就什么都不做，否则就通过 for 循环从文档类的数组对象中依次取出图形对象（包括 3 个重要绘图参数），使用 CObject 类的 GetAt()函数来实现。然后调用图形对象，也就是 Graph 的 Draw()函数来绘制图像对象中的内容。

至此，可以进入保存和再显示图形功能的最后实现阶段，也就是对 Serialize()函数的编写。

首先，修改 CGraphFileDoc 类的 Serialize()函数中的代码如下：

```
void CGraphFileDoc::Serialize(CArchive& ar)
{
    obArray.Serialize(ar);
}
```

上面的代码将 Serialize()函数原来的代码删除，然后使用 CObject 对象调用 CGraph 类的 Serialize()函数，这样，就可以将剩余的操作放在 CGraph 类的 Serialize()函数中完成。将 CGraph 类的 Serialize()函数修改如下：

```
void CGraph::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
        ar<<m_nType<<m_ptOrigin<<m_ptEnd;
    else
        ar>>m_nType>>m_ptOrigin>>m_ptEnd;
}
```

由于 m_ptOrigin、m_ptEnd 和 m_nType 都是可串行化的基本数据类型，因此直接使用 Carchive 类的提取和插入操作符对它们进行串行化操作，将它们保存到文件中或者按照同样的顺序从文件中提取出来。

最后，还要完成一个删除操作，即当用户新建或者打开一个文档对象时，需要销毁先前文档中保存的数据，也就是释放先前占用的内存。当用户选择“打开”和“新建”命令时，应用程序会自动调用 DeleteContents()虚函数，因此可以把释放内存的工作放在这里完成。

DeleteContents()虚函数主要由框架调用，用来删除文档的数据，同时并不销毁 CDocument 对象本

身。该函数在文档将要被销毁之前调用，也在该文档重复使用之前被调用，以确保文档是空的。对单文档应用程序来说，这一点特别重要，因为单文档应用程序仅用一个文档，无论用户是创建还是打开另一个文档，该文档对象都是被重复使用的，所以在文档对象被重复使用之前，应释放已分配的内存。

右击 CGraphFileDoc 类，选择“Properties”命令，然后选择属性面板中的 Overrides 项，选择 DeleteContents 来增加 DeleteContents()函数，如图 12.6 所示。修改该函数的代码如下：

```
void CGraphFileDoc::DeleteContents()
{
    // TODO: Add your specialized code here and/or call the base class
    int index=obArray.GetSize();
    while(index--)
    {
        delete (CGraph*)obArray.GetAt(index);
    }
    obArray.RemoveAll();
    CDocument::DeleteContents();
}
```

至此，完成了对 GraphFile 工程添加串行化操作的任务。编译并运行程序，来检验一下成果。

首先，使用“绘图”菜单中的相应命令绘制一些图像，如图 12.7 所示。

图 12.6　Overrides 项

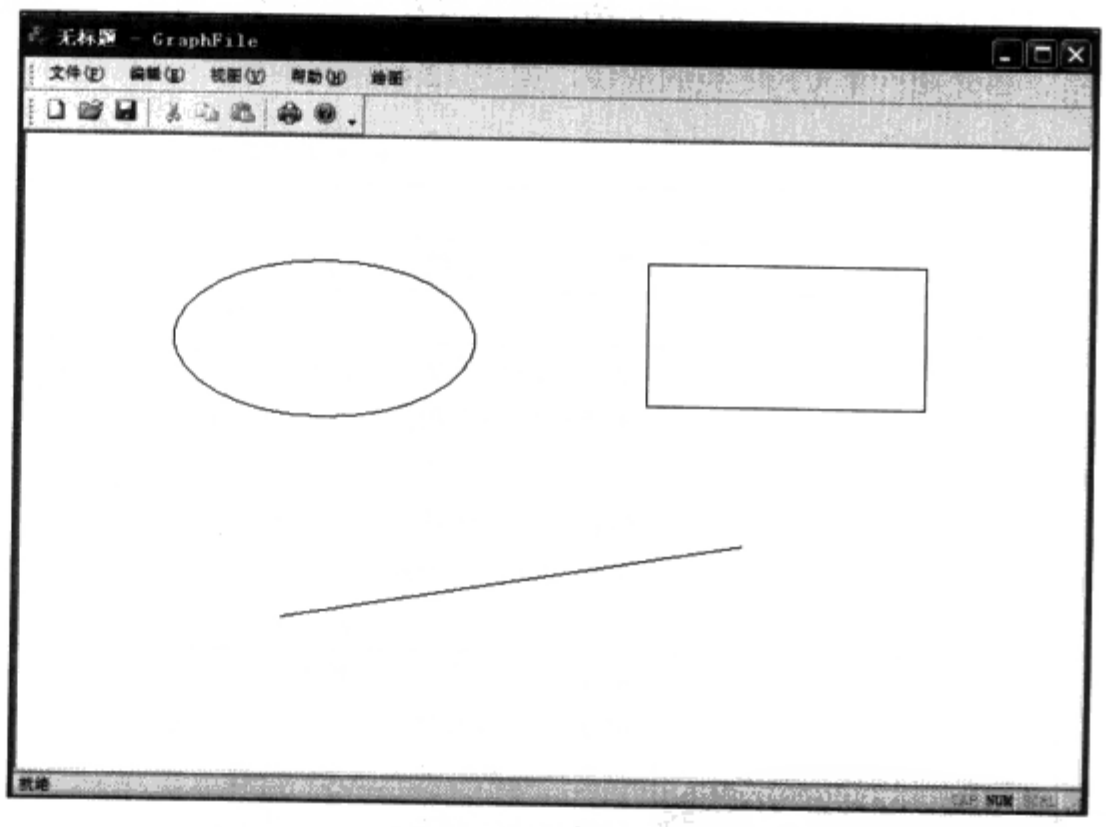


图 12.7　绘制一些图像

然后，选择“文件”菜单中的“保存”命令，保存文件为“graph”，如图 12.8 所示。

保存完毕后，选择“新建”命令，则窗口中的图形就会消失。重新选择“打开”命令，打开刚才保存的“graph”文件，则图形重新被绘制在应用程序窗口中。

序列化简化了对象的保存和载入操作，为对象提供了持久性，但是，序列化本身仍具有一定的局限性。由于序列化一次从文件中载入所有对象，因此它不适用于大文件编辑器和数据库。对于数据库和大文件编辑器，它们每次只是从文件中读入一部分，此时，就不应该采用文档的序列化机制来直接读取和保存文件。另外，使用外部文件格式（预先定义的文件格式而不是本应用程序定义的文件格式）的程序一般也不使用文档的序列化。下一节将介绍另一种打开和保存文件的方法——CFile 类。

图 12.8　保存菜单项

12.2　CFile 类

上一节多次使用了 CFile 类，CFile 类是 MFC 中提供的支持文件操作的基类，该类提供了没有缓存的二进制格式的硬盘文件输入输出功能，并直接通过派生类支持文本文件和内存文件。上一节也提到了，将 CFile 与 CArchive 类共同使用，可以支持 MFC 对象的串行化。CFile 类与其派生类的层次关系还可以让程序通过多个 CFile 接口操作所有文件对象。例如，一个内存文件相当于一个磁盘文件。

下面介绍一下 CFile 类的成员，如表 12.4 所示。

表 12.4　　CFile 类的成员

数 据 成 员	
m_hFile	常包含操作系统文件句柄
构 造 函 数	
Cfile()	从一个路径或文件句柄构造一个 CFile 对象
Abort()	不管任何警告和错误，关闭一个文件
Duplicate()	构造一个基于该文件的副本
Open()	带错误检验选项，安全打开一个文件
Close()	关闭文件，删除对象
输入/输出	
Read()	从文件的当前位置读数据
ReadHuge()	可从文件的当前位置读取大于 64KB 的数据，在 32 位编程中废弃了
Write()	将文件数据写入当前文件位置
WriteHuge()	可将文件的大于 64KB 的数据写到当前文件位置，在 32 位编程中废弃了
Flush()	溢去未被写入的任何数据
定　　位	
Seek()	定位于当前文件指针
SeekToBegin()	定位当前文件指针到文件开头
SeekToEnd()	定位当前文件指针到文件尾
GetLength()	获取文件长度
SetLength()	改变文件长度
封　　锁	
LockRange()	在文件中封锁一定范围内的字节
UnlockRange()	解锁一定范围内的字节

续表

状　　态	
GetPosition()	获取当前文件指针
GetStatus()	获取打开文件的状态
GetFileName()	获取被选文件的文件名
GetFileTitle()	获取被选文件的标题
GetFilePath()	获取被选文件的完整路径
SetFilePath()	设置被选文件的完整路径
静　　态	
Rename()	重命名指定文件（静态函数）
Remove()	删除指定文件（静态函数）
GetStatus()	获取指定文件的状态（静态，虚函数）
SetStatus()	设置指定文件的状态（静态，虚函数）

下面详细介绍如何使用 CFile 类提供的函数实现对文件的操作。

12.2.1　打开文件操作

如要使用 CFile 类打开文件，首先要声明一个 CFile 类的对象。这里，先来介绍一下 CFile 类的构造函数，该类有 3 种形式的构造函数，其声明分别如下：

```
CFile( );
CFile(int hFile);
CFile(LPCTSTR lpszFileName,UINT nOpenFlags);
  throw(CFileException);
```

第一个不带任何参数的构造函数不打开文件，而是将 m_hFile 设置为 CFile::hFileNull()。当构建一个 file 对象后，可以使用成员函数 Open()打开文件并捕获异常。Open()函数的声明如下：

```
virtual BOOL Open(LPCTSTR lpszFileName,UINT nOpenFlags,CFileException *pError = NULL);
```

此函数是 CFile 类提供的文件的打开操作函数，如果成功打开文件，则返回一个非零的整数，否则返回 0。只有打开文件失败时，参数 pError 才有效。

参数 lpszFileName 表示待打开文件的路径，路径可以为绝对路径、相对路径或网络名。参数 nOpenFlags 指定文件共享和访问的方式，指定当打开文件时进行的动作，表 12.5 说明了该参数可取的值。可以用 OR(|)操作符将这些取值连接起来，至少应有一个访问权限和一个共享选项，modeCreate()和 modeNoInherit()是可选的。参数 pError 指向一个存在的文件异常对象，用来获取失败操作的状态。

表 12.5　　nOpenFlags 参数取值及说明

取　　值	说　　明
CFile::modeCreate()	指示构造函数创建一个新文件。如果该文件已经存在，那么将其长度截断为 0
CFile::modeNoTruncate()	与 Cfile:modeCreate()组合使用。如果正创建的文件已经存在，那么其长度不会被截断为 0
CFile::modeRead()	打开文件，该文件仅用于读取操作
CFile::modeReadWrite()	打开文件，该文件可读可写
CFile::modeWrite()	打开文件，该文件仅用于写入操作
CFile::modeNoInherit()	禁止子进程继承该文件
CFile::shareDenyNone()	打开文件，同时并不拒绝其他进程对该文件的读取或写入访问。如果该文件已经被其他进程以兼容模式打开，那么文件创建失败
CFile::shareDenyRead()	打开文件，同时拒绝其他进程对该文件的读取访问。如果该文件已经被其他进程以兼容模式，或者为读取访问打开，那么文件创建失败

续表

取　值	说　明
CFile::shareDenyWrite()	打开文件，同时拒绝其他进程对该文件的写入访问。如果该文件已经被其他进程以兼容模式，或者为写入访问打开，那么文件创建失败
CFile::shareExclusive()	以排它独占模式打开文件，拒绝其他进程对该文件的读取和写入访问。如果该文件已经以任何其他模式被读取或写入访问，即使是由当前进程打开的，构造函数失败
CFile::shareCompat()	此标志在 32 位 MFC 中不可用。当在 Cfile::Open()函数中使用时，此标记将映射为 Cfile::shareExcluseve()
CFile::typeText()	设置文本模式，带有一对回车换行字符（仅在 CFile 类的派生类中使用）
CFile::typeBinary()	设置二进制模式（仅在 CFile 类的派生类中使用）

下面的代码演示了如何使用该方法打开一个文件：

```
CFile file;
CFileException e;
char* pFileName = "file.txt";
if(!file.Open(pFileName, CFile::modeCreate | CFile::modeWrite,&e))
{
    #ifdef _DEBUG
    afxDump<<"无法打开文件"<<e.m_cause<<"\n";
    #endif
}
```

第二个带一个参数的构造函数构造一个 CFile 对象，对应于由 hFile 标识的操作系统文件。对访问模式或文件类型不进行检查。当 CFile 对象被析构时，操作系统文件不会关闭，必须由用户手动关闭它。

最常用的构造函数是第三个带两个参数的构造函数，该函数构造一个 CFile 对象，并打开与指定路径相对应的操作系统文件。该构造函数将第一个构造函数和 Open()成员函数组合起来。如果打开文件出错，则会产生一个异常，通常这说明该错误不可解决，用户将被警告。

第三种构造形式有两个参数，读者可以发现它们与 Open()函数的前两个参数是类似的。其中参数 lposzFileName 也是指向所需文件路径的字符串，此路径可以是相对路径、绝对路径或网络名。nOpenFlags 参数同样指定文件共享和访问的方式，指定当打开文件时进行的动作，具体取值也可以参考表 12.5。

直接使用该构造函数打开文件，代码如下：

```
char* pFileName = "file.txt";
TRY
{
    CFile file(pFileName, CFile::modeCreate | CFile::modeWrite);
}
CATCH(CFileException,e)
{
    #ifdef _DEBUG
        afxDump<<"无法打开文件"<<e.m_cause<<"\n";
    #endif
}
END_CATCH
```

12.2.2　读写文件操作

下面介绍如何使用 CFile 类读取和写入文件。

CFile 类提供了 Write()函数来将数据从缓冲区写入与 CFile 对象相关联的文件。该函数声明如下：

```
virtual void Write(const void* lpBuf,UINT nCount);
throw (CFileException);
```

该函数的参数 lpBuf 指向用户提供的缓冲区，包含将写入文件中的数据，可以是一个字符串或是其他数据。参数 nCount 表示从缓冲区内传输的字节数，对于文本模式的文件，回车换行作为一个字符。

下面就来实现写入一个文件。创建一个单文档 MFC 工程 File，在该应用程序主菜单上添加一个新的菜单“File”，然后为其添加两个新的菜单命令“read”和“write”。为这两个命令在 CFileView 类中添加两个命令响应函数 OnFileRead()和 OnFileWrite()。然后在响应函数 OnFileWrite()中添加如下代码：

```
void CFileView::OnFileWrite()
{
    // TODO: Add your command handler code here
                                                    //构造 CFile 文件对象
    CFile file(_T("file.txt"),CFile::modeCreate|CFile::modeWrite);
                                                    //写入文件
    file.Write("Visual C++ 2010 程序设计入门到精通",strlen("Visual C++ 2010 程序设计入门到精通"));
                                                    //关闭文件
    file.Close();
}
```

在代码中首先构建了一个 CFile 对象，该对象的文件名是“file.txt”，使用 modeCreate 或者 modeWrite 模式打开，然后在该文件中写入“Visual C++ 2010 程序设计入门到精通”，最后关闭该文件。

运行程序，选择“File”→“write”命令，可以发现程序所在文件夹中多了一个“file.txt”文件，打开文件，其中正是所写入的“Visual C++ 2010 程序设计入门到精通”文字。

然后，再使用 CFile 类提供的 Read()函数来读取文件。该函数声明如下：

```
    virtual UINT Read (void* lpBuf,UINT nCount);
throw(CFileException);
```

此函数返回传输到缓冲区的字节数。注意，对所有 CFile 类，如果到达文件尾，则返回值可能比 nCount 小。参数 lpBuf 指向用户提供的缓冲区以接收从文件中读取的数据。参数 nCount 表示可以从文件中读出的字节数的最大值。对于文本模式的文件，回车换行作为一个字符。

下面在响应函数 OnFileRead()中实现对文件的读取：

```
void CFileView::OnFileRead()
{
    // TODO: Add your command handler code here
                                                    //构造 CFile 文件对象
    CFile file(_T("file.txt"),CFile::modeRead);
    char * pBuf;
                                                    //获得文件长度
    DWORD fileLen;
    fileLen=file.GetLength();
    pBuf=new char[fileLen+1];
    pBuf[fileLen]=0;
                                                    //读取文件
    file.Read(pBuf,fileLen);
    file.Close();
    CString result(pBuf);
    MessageBox(result);
}
```

在代码中首先使用 modeRead 模式打开刚刚创建的文件“file.txt”；然后使用 CFile 类提供的 GetLength()函数来获取文件的当前逻辑长度，以字节表示；接着创建一个 char 类型的数组来保存从文件中读取的内容；最后，使用消息对话框将读取到的内容显示出来。

编译并运行程序，选择“File”→“read”命令，则消息对话框成功显示了读取到的内容“Visual C++ 2010 程序设计入门到精通”，运行结果如图 12.9 所示。

图 12.9 读写文件操作

12.2.3 定位文件操作

CFile 类提供了一个 Seek()函数，可以使用该函数在以前打开的文件中重新定位一个指针。Seek()函数使用户可以随机访问一个文件的内容，这是通过指针移动一定量来完成的，指针的移动可以为绝对或相对。该函数的声明如下：

```
virtual LONG Seek(LONG lOff,UINT nFrom);
throw(CFileException);
```

该函数的返回值是 LONG 类型的，如果要求的位置合法，则 Seek()返回从文件开始起的字节偏移量；否则该值没有定义并产生 CFileException 异常。

该函数的第一个参数 lOff 记录了指针移动的字节数；而参数 nFrom 规定了指针移动的模式，它可以取以下值之一：

- CFile::begin 从文件开始，把指针向后移动 lOff 字节；
- CFile::current 从当前位置开始，把指针向后移动 lOff 字节；
- CFile::end 从文件尾开始，把指针向前移动 lOff 字节，注意，必须移动到存在的文件中，因此 lOff 应为负值；否则超出文件尾。

下面看一个实际使用 Seek()函数的例子。修改 OnFileRead()的代码如下：

```
void CFileView::OnFileRead()
{
    // TODO: Add your command handler code here
                                                    //构造 CFile 文件对象
    CFile file(_T("file.txt"),CFile::modeRead);
                                                    //重新定位文件指针
    LONG lOffset = 7;
    LONG lActual = file.Seek(lOffset, CFile::begin);
    char * pBuf;
                                                    //获得文件长度
    DWORD fileLen;
    fileLen=file.GetLength();
    pBuf=new char[fileLen+1-lOffset];
    pBuf[fileLen-lOffset]=0;
                                                    //读取文件
    file.Read(pBuf,fileLen-lOffset);
    file.Close();
    CString result(pBuf);
    MessageBox(result);
}
```

在代码中，设定了从文件开始，把指针向后移动 7 字节；相应的，读取文章的长度也减少了 7 字节。编译并运行程序，选择“File”→“read”命令，其结果如图 12.10 所示。可以看到，消息对话框中显示的读取到的内容是“C++ 2010 程序设计入门到精通”，少了前面“Visual ”7 字节的内容。

为了方便操作，CFile 类还提供了 SeekToBegin()方法和 SeekToEnd()方法，前者是将文件指针指向文件开始位置，等价于 Seek(0L, CFile::begin)；后者是将文件指针指向文件逻辑尾部，等价于 CFile::Seek(0L, CFile::End)。读者如果有兴趣，可以试验一下这两个函数。

图 12.10　定位文件操作

12.2.4　关闭文件操作

完成对一个文件的读写操作之后，需要关闭该文件。CFile 类提供了两种关闭文件的方法，除了前面已经使用的 Close()函数，还有 Abort()函数。这两个函数的声明分别如下：

```
virtual void Close( );
throw(CFileException);

virtual void Abort( ):
```

由这两个函数的声明可以看出，这两个函数的不同之处在于处理异常时，首先，Abort()函数不会因失败而产生异常值，这是因为 Abort()忽略失败；其次，如果文件未被打开或已经关闭，则 Abort()不会插入断言。

这两个函数的共同点在于，其作用都是关闭与调用该方法的对象有关的文件并使之不可读写。当析构一个对象时未关闭文件，则构造函数关闭它；如果用 new 操作符放置 CFile 对象到堆顶，在关闭文件后必须删除它。二者都设置文件句柄 m_hFile 为 CFile::hFileNull。

12.2.5　异常操作

读者一定注意到了，在 CFile 类的许多函数声明中，都会抛出一个 CFileException 类型的异常等待用户编程时捕获。在使用 CFile 及其派生类的对象的时候，如果产生异常则会创建和抛出 CFileException 对象，这个 CFileException 对象代表了与文件相关的异常。CFileException 类包括可移植的异常原因代码和操作系统指定错误值等公共数据成员，此类也提供产生文件异常的静态成员函数，返回操作系统和 C++运行中的错误原因代码。

CFileException 对象可以在 CFile 成员函数中被创建和产生，也可以在派生类成员函数中创建或产生。可以在 CATCH 表达式范围内访问这些对象。因为该类包括有可移植的异常原因代码和操作系统指定错误误值等公共数据成员，所以可以仅使用错误原因代码获取出现异常的原因，可以仅用出错代码获取出现异常的原因。

CFileException 类的成员如表 12.6 所示。

表 12.6　　CFileException 类的成员

数 据 成 员	
m_cause	包含异常原因相对应的可移植代码
m_IosError	包含相关操作系统错误代码，可参阅操作系统技术手册，得到错误代码列表
m_strFileName	包含异常的文件名
构 造 函 数	
CfileException()	构造一个 CFileException 对象
代 码 转 换	
OsErrorToException()	返回与操作系统错误值相对应的出错原因代码
ErrnoToException()	返回与运行错误值相对应的出错原因代码
帮 助 函 数	
ThrowOsError()	返回一个基于操作系统错误值的出错原因代码
ThrowErrno()	产生一个基于运行错误值的文件异常

数据成员中的 m_cause 包含异常原因相对应的可移植代码，是由 CFileException 枚举类型定义的值。此数据成员是一个公共的 int 类型变量，枚举符与含义如表 12.7 所示。

表 12.7　　CFileException 枚举类型定义的值

枚 举 符	说　明
CFileException::none	无错误发生
CFileException::generic	发生未指定错误
CFileException::FileNotFound	不可定位的文件
CFileException::badPath	全部或部分路径无效
CFileException::tooManyOpenFiles	超过允许打开的文件数
CFileException::accessDenied	文件不可访问
CFileException::invalidFile	想使用一个无效的文件句柄
CFileException::removeCurrentDir	当前工作目录不能被移走
CFileException::directoryFull	无目录可访问
CFileException::badSeek	设置文件指针出错
CFileException::hardIO	硬件错误
CFileException::sharingViolation	未加载 SHARE.EXE 或共享区被锁
CFileException::lockViolation	试图封锁一个已封锁的区域
CFileException::diskFull	磁盘满
CFileException::endOfFile	已达文件末尾

但是请注意，这些原因枚举符与 CArchiveException 原因枚举符不相同。

读者可以使用以下方式，获得并处理 CFileException 类抛出的异常：

```
char* pFileName = "file.txt";
TRY
{
    CFile file(pFileName, CFile::modeCreate | CFile::modeWrite);
}
CATCH(CFileException,e)
{
    #ifdef _DEBUG
      afxDump<<"无法打开文件"<<e.m_cause<<"\n";
    #endif
}
END_CATCH
```

12.2.6　文件管理操作

常用的文件管理操作包括复制、删除、重命名、获取文件的位置、获取文件的路径等。下面将逐一对它们进行介绍。

（1）文件的刷新，CFile::Flush()函数，具体声明如下：

```
virtual void Flush( );
```

此函数强制滞留在缓冲区的数据写入文件。使用 Flush()函数不能保证 CArchive 缓冲区被清空。如果正在使用一个档案，可先调用 CArchive::Flush()。

（2）文件的备份，CFile::Duplicate()函数，具体声明如下：

```
virtual CFile* Duplicate( ) const;
```

此函数为给定文件构造一个备份 CFile 对象，它的返回值指向备份 CFile 对象。

（3）获取文件的名称，CFile::GetFileName()，具体声明如下：

```
virtual CString GetFileName( ) const;
```

此函数的返回值是一个文件名，调用此成员函数获取一个指定文件的文件名。例如，调用 GetFileName()来产生一个关于文件“C:\Visual C++ 2010\file.txt”的消息给用户时，文件名“file.txt”被返回。

（4）获取文件的路径，CFile::GetFilePath()，具体声明如下：

```
virtual CString GetFilePath( ) const;
```

此函数的返回值是指定文件的完整路径，调用此成员函数获取指定文件的全路径。例如，调用 GetFilePath()来产生一个关于文件“C:\Visual C++ 2010\file.txt”的消息给用户时，文件路径“C:\Visual C++ 2010\file.txt”被返回。

（5）获取文件的标题，CFile::GetFileTitle()，具体声明如下：

```
virtual CString GetFileTitle( ) const;
```

此函数的返回值是指定的文件的标题，调用此成员函数获取指定文件的标题，例如，调用 GetFileTitle()产生一个关于文件“C:\Visual C++ 2010\file.txt”的消息给用户时，文件标题 file 被返回。

（6）获取文件的长度，CFile::GetLength()，具体声明如下：

```
virtual DWORD GetLength( ) const;
```

此函数的返回值是指定文件的长度，调用此函数获取文件的当前逻辑长度，以字节表示。前面已经使用过这个函数。

（7）获取文件指针的当前值，CFile::GetPosition()，具体声明如下：

```
virtual DWORD GetPosition ( )const;
```

此函数的返回值是指定文件的 32 位双字文件指针，调用此函数获取文件指针的当前值，可用于以后的 Seek()调用，例如：

```
CFile file;
file.Open(_T("file.txt"), CFile::modeCreate |CFile::modeReadWrite);
LONGLONG lOffset = 1000;
ULONGLONG lActual;
lActual = file.Seek(lOffset, CFile::begin);
ASSERT(cfile.GetPosition() == lActual);
```

（8）删除指定的文件，CFile::Remove()函数，具体声明如下：

```
static void PASCAL Remove(LPCTSTR lpszFileName);
```

参数 lpszFileName 表示所要删除文件的路径字符串。路径可以为相对或绝对路径，但不可以包含网络名。此静态函数删除由路径指定的文件，但不可移去一个目录。如果有相关联的文件被打开或文件不可移去，则函数产生一个异常。下面是该函数的使用示例：

```
//CFile::Remove 使用示例
char* pFileName ="file.txt";
TRY
{
    CFile::Remove(pFileName);
}
CATCH(CFileException,e)
{
  #ifdef _DEBUG
      afxDump <<"文件"<<pFileName<<"不能被删除\n";
```

```
    #endif
}
END_CATCH
```

（9）文件的重命名，CFile::Rename()函数，具体声明如下：

```
static void PASCAL Rename(LPCTSTR lpszOldName,LPCTSTR lpszNewName);
```

参数 lpszOldName 表示原路径，lpszNewName 表示新路径，此函数更改一个指定文件的名称。注意：目录不可更改名称。

```
// CFile::Rename 使用示例
char* pOldName="file.txt";
char* pNewName="newfile.txt";
TRY
{
    CFile::Rename(pOldName,pNewName);
}
CATCH(CFileException,e)
{
    #ifdef _DEBUG
    afxDump << "文件" << pOldName << "不能被重命名，原因是" << e->m_cause <<"\n";
    #endif
}
END_CATCH
```

（10）指定文件的路径，CFile::SetFilePath()函数，具体声明如下：

```
virtual void SetFilePath(LPCTSTR lpszNewName);
```

参数 lPszNewName 指向指定新文件的路径字符串，调用此成员函数可以指定文件路径。例如，当构造一个 CFile 对象而文件的路径无效时，可调用此成员函数提供路径。但是，SetFilePath()不打开或创建文件，只将 CFile 对象和路径名联系起来，即可使用。

由于本书篇幅限制，关于文件管理和操作的内容就先介绍到此，读者可以自行实践一下这些管理操作。如果读者需要这些操作的详细信息或者详细使用方法，可以参考 MSDN 帮助文档。

12.3 .ini 文件读写操作

.ini 文件是 Windows 的系统配置文件，统管 Windows 的各项配置。一般用户使用 Windows 提供的各项图形化管理界面就可实现相同的配置，但在某些情况下，还是要直接编辑.ini 才方便，一般只有很熟悉 Windows 操作系统的用户才能去直接编辑。.ini 文件开始时用于 Win3.1 下，自从 Windows 95 的推出，在 Windows 系统中引入了注册表的概念，.ini 文件在 Windows 系统的地位就开始不断下滑，这是因为注册表的独特优点使应用程序和系统都把许多参数和初始化信息放入了注册表中。但在某些场合，.ini 文件还拥有其不可替代的地位。

除了 Windows，很多 Windows 下的应用软件也有.ini 文件，用来配置应用软件以实现不同用户的要求。在开发人员写的程序中，总有一些配置信息需要保存下来，以便完成程序的功能，最简单的办法就是将这些信息写入.ini 文件中，程序初始化时再读入。

下面介绍一下.ini 文件的结构。

```
;注释   [小节名]   关键字=值   ...
```

.ini 文件允许有多个小节，每个小节又允许有多个关键字，“=”后面是该关键字的值。值的类型有 3 种：字符串、整型数值和布尔值。其中字符串存储在.ini 文件中时没有引号，布尔真值用 1 表示，

布尔假值用 0 表示。注释以分号“;”开头。图 12.11 是笔者计算机的 Win.ini 文件的内容，第一行为注释，下面是小节名【fonts】，该小节没有关键字；在【Mail】小节下的第一行是关键字 MAPI 及其值 1。

```
; for 16-bit app support
[fonts]
[extensions]
[mci extensions]
[files]
[Mail]
MAPI=1
CMCDLLNAME32=mapi32.dll
CMC=1
MAPIX=1
MAPIXVER=1.0.0.1
OLEMessaging=1
```

图 12.11　Win.ini 的内容

在 Windows 操作系统中，System.ini 和 Win.ini 这两个文件包含了操作系统所有的控制功能和应用程序的信息，System.ini 管理计算机硬件，而 Win.ini 管理桌面和应用程序。所有驱动、字体、设置和参数都会保存在.ini 文件中，任何新程序都会被记录在.ini 文件中。这些记录会在程序代码中被引用，下面介绍一下如何读写 Win.ini 文件，实现应用程序的初始化。Win32 API 提供了 WriteProfileString()函数和 GetProfileString()函数来满足对 Win.ini 文件的读写需求。下面将分别介绍这两个函数。

WriteProfileString()函数的原型声明如下：

```
BOOL WriteProfileString（LPCTSTR lpAppName,LPCTSTR lpKeyName, LPCTSTR
 lpString);
```

该函数的 3 个参数的作用分别如表 12.8 所示。

表 12.8　　WriteProfileString()函数 3 个参数的作用

lpszSection	指定要在其中写入新串的小节。若该小节尚不存在，会创建这个小节。这个字符串不区分大小写
lpszKeyName	要设置的项名或条目名。这个字符串不区分大小写，用 vbNullString 可删除这个小节的所有设置项
lpszString	指定为这个项写入的字符串值。用 vbNullString 表示删除这个项现有的字符串

了解 WriteProfileString() 函数的定义，就来应用一下这个函数，将上节创建的 MFC 工程 File 的初始信息写入 Win.ini 文件中。在 CFileApp 类的 InitInstance()函数中写入以下代码：

```
//将初始信息写入 Win.ini 文件
::WriteProfileString (_T("Visual C++ 2010") , _T("file") , _T("ini") );
```

该代码向 Win.ini 文件中写入了一个新的小节【Visual C++ 2010】，在其上创建一个新键，键名为 file，其值为 ini。重新编译并运行程序，然后打开 Win.ini 文件，发现文件中多了新加入的这个小节，其结果如图 12.12 所示。

```
wma=MPEGVideo
wmv=MPEGVideo
wmx=MPEGVideo
wpl=MPEGVideo
wvx=MPEGVideo
[SciCalc]
layout=0
[Visual C++ 2010]
file=ini
```

图 12.12　修改后的 Win ini 文件

GetProfileString()函数的原型声明如下：

```
BOOL GetProfileString（LPCTSTR lpAppName,LPCTSTR lpKeyName,
    LPCTSTR lpDefault,LPCTSTR lpReturnedString,DWORD nSize);
```

该函数的参数的作用分别如表 12.9 所示。

表 12.9　　Get Profile String 函数说明

参　数	类型及说明
lpAppName	要在其中查找条目的小节名。这个字符串不区分大小写。如果为 vbNullString，则在 lpReturnedString 缓冲区中装载这个.ini 文件的所有小节
lpKeyName	要获取的项名或条目名。这个字符串不区分大小写。如果为 vbNullString，则在 lpReturnedString 缓冲区装载指定小节内的所有项
lpDefault	指定条目未找到时返回的默认值，可设为空（""）
lpReturnedString	指定一个预先初始化好的字符串缓冲区，长度至少为 nSize 个字符
nSize	装载到 lpReturnedString 缓冲区的最大字符数

在 File 工程的菜单中添加一个新的菜单命令 getIni，为该命令在类 CFileView 中添加一个新的消息响应函数 OnGetini()，在该函数中添加如下代码：

```
void CFileView::OnGetini()
{
```

```
    // TODO: Add your command handler code here
    CString str;
    ::GetProfileString(_T("Visual C++ 2010"), _T("file"), _T(""),
    str.GetBuffer(50), 50 );
    AfxMessageBox(str);
}
```

利用代码中阴影所标注的代码读取刚刚在 Win.ini 文件中写入的小节内容，编译并运行程序，选择“getIni”命令，程序的运行结果如图 12.13 所示，显示成功读取了小节【Visual C++ 2010】中键 file 的值 ini。

图 12.13 小节“Visual C++ 2010”中键 file 的值

当用户运行完该程序后，不要忘记删除在 Win.ini 文件中新加入的小节。在现在的 Windows 操作系统中，Win.ini 文件的功能越来越被注册表替代，下面学习一下注册表的相关知识。

12.4 注册表读写操作

注册表是 Windows 程序员构造的一个复杂的信息数据库，是多层次式的。在不同系统上注册表的基本结构相同，其中的复杂数据会在不同方式上结合，从而产生出一个绝对惟一的注册表。本节将简单地介绍一下注册表及其读写操作方法。

12.4.1 注册表简介

1. 注册表的作用

注册表是 Windows 的命根，其中存储了大量的系统信息，说白了就是一个庞大的数据库。注册表中的所有信息平时都是由 Windows 操作系统自主管理的，用户也可以通过软件或手工修改它。注册表中有很多系统的重要信息，包括外设、驱动程序、软件、用户记录等，在很大程度上“指挥”计算机怎样工作。注册表功能非常强大，是 Windows 的核心。

注册表主要包括以下内容。

- 软、硬件的有关配置的初始状态信息。注册表中保存着应用程序和资源管理器外壳的初始条件、首选项和卸载数据。
- 整个计算机系统的设置，该系统获得的各种许可协议，应用程序与文件扩展名的关联、硬件部件的描述、状态和属性。
- 性能记录、其他底层的系统状态信息，以及其他数据。

如果注册表受到了破坏，轻者使 Windows 的启动过程出现异常，重者将导致整个系统全部瘫痪。因此正确地认识和使用注册表，特别是及时备份，以及在有问题时恢复注册表，对 Windows 用户来说显得十分重要。在这里提醒一下读者，如果要修改注册表却没有完全把握修改正确，在修改之前，一定要备份注册表，以便在修改错误后还可以还原。

2. 注册表的结构

在注册表中，所有的数据都通过一种树状结构以主键和子键的方式组织起来，类似于目录结构（如图 12.14 所示）。

每个键都包含一组特定的信息，每个键的键名都是和它所包含的信息相关的。如果这个键包含子键，则在注册表编辑器窗口中代表这个键的文件夹的左侧将有“＋”符号，以表示在这个文件夹中有

更多的内容。如果这个文件夹被用户打开了，那么“＋”符号就会变成“－”符号。

下面首先介绍一下注册表都包含哪些主键，如图 12.15 所示。

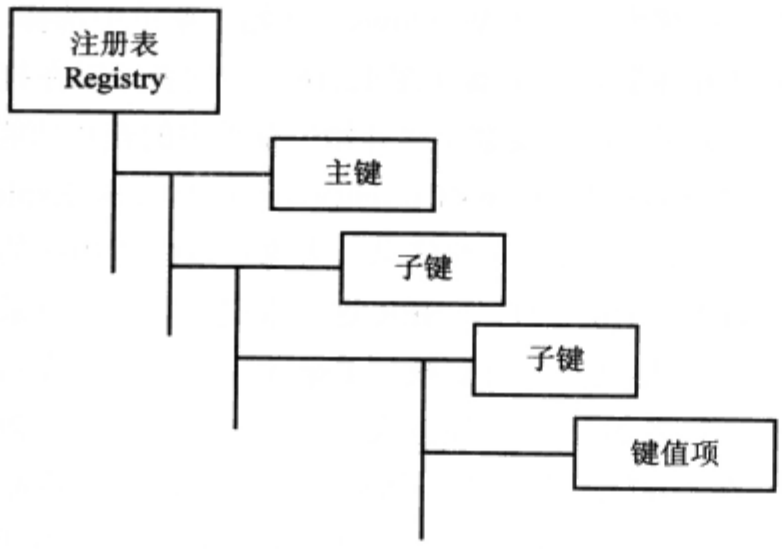


图 12.14　注册表中的数据结构

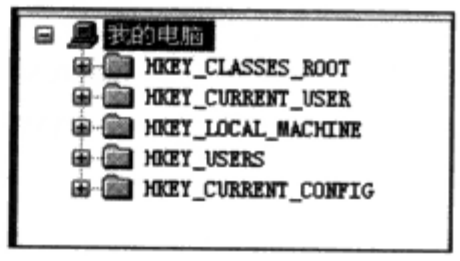


图 12.15　注册表的主键

- HKEY_USERS：该根键保存了存放在本地计算机口令列表中的用户标识和密码列表。每个用户的预配置信息都存储在 HKEY_USERS 根键中。HKEY_USERS 是远程计算机访问的根键之一。
- HKEY_CURRENT_USER：该根键包含本地工作站中存放的当前登录的用户信息，包括用户登录用户名和暂存的密码。用户登录操作系统时，其信息从 HKEY_USERS 中相应的项复制到 HKEY_CURRENT_USER 中。
- HKEY_CURRENT_CONFIG：该根键保存了定义当前用户桌面配置（如显示器等）的数据、最后使用的文档列表（MRU）和其他有关当前用户的信息。
- HKEY_CLASSES_ROOT：包含注册的所有 OLE 信息和文档类型，是从 HKEY_LOCAL_MACHINE\SOFTWARE \Classes 复制的。根据 Windows 98 中文版中安装的应用程序的扩展名，该根键指明其文件类型的名称。
- HKEY_LOCAL_MACHINE：该根键保存本地计算机硬件数据，此根键下的子关键字包括在 SYSTEM.DAT 中，用来提供 HKEY_LOCAL_MACHINE 所需的信息，或者在远程计算机中可访问的一组键中。该根键中的许多子键与 System.ini 文件中设置项类似。
- HKEY_DYN_DATA：该根键保存了系统在运行时动态数据，此数据在每次显示时都是变化的，因此此根键下的信息没有放在注册表中。

注册表通过键和子键来管理各种信息，但是，注册表中的所有信息是以各种形式的键值项数据保存下来的。在注册表编辑器的右窗格中，保存的都是键值项数据（如图 12.16 所示）。这些键值项数据可分为如下 3 种类型。

- 字符串值：在注册表中，字符串值一般用来表示文件的描述、硬件的标识等，通常由字母和数字组成，最大长度不能超过 255 个字符。
- 二进制值：在注册表中，二进制值是没有长度限制的，可以是任意字节长。在注册表编辑器中，二进制以十六进制的方式显示出来。
- DWORD 值：DWORD 值是一个 32 位（4 字节，即双字）长度的数值。在注册表编辑器中，系统会以十六进制的方式显示 DWORD 值。

上面简单地介绍了关于注册表的一些内容，当然，注册表的内容远不止这些，感兴趣的读者可以阅览专门介绍注册表的一些书籍。

3. 注册表的修改

修改注册表内容的方法有很多，这里先介绍一种使用 Windows 的注册表编辑器——regedit.exe 来修改注册表的方法。这里以 Windows XP SP3 操作系统为例，在 Windows“开始”菜单中选择“运行”命令，在对话框输入“regedit”命令运行，即可打开注册表编辑器（图 12.16）。读者可以在打开的编辑器中查找需要修改的注册表修改项。这里举一个例子，比如要禁止“开始”菜单中的查找功能，需要修改的内容为 HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\ Explorer，在右边的窗格中新建一个 DWORD 值“NoFind”，并设值为 1（详细信息，大家可以参考相应的注册表修改手册）。打开注册表编辑器，可以看到有 HKEY_CURRENT_USER 这个根键，展开该目录并找到 Software 项，再展开找到 Microsoft 项，这样一层一层地展开，直到找到 Explorer 项。然后在右侧的窗格中右击，选择“新建”→“DWORD 值”命令，输入 NoFind（注意大小写），可以发现右侧窗格中多了一个名为 NoFind 的键名，然后给它赋值，在 NoFind 上双击，然后在弹出的窗口中将原来默认的 0 改成 1，如图 12.17 所示。最后，重新启动计算机，会发现开始菜单里的“查找”命令消失了。如果想把它改回来，那么按上面的方法，把 1 改成 0，然后重新启动计算机就可以了。其他的修改方法也类似，关键是要找到修改的内容，这里可以参考一些注册表修改手册。

另外，还有其他一些修改注册表的方法，比如用专门的修改软件，手工写.reg 注册表文件等。当然，使用 Visual C++编写的应用程序也可以修改注册表，具体方法将在后面介绍。

12.4.2 注册表 API

注册表在 Windows 操作系统中是以二进制形式存储的，Win32 API 提供了大量的函数，以便应用程序访问注册表，下面介绍一些常用的 API 函数。

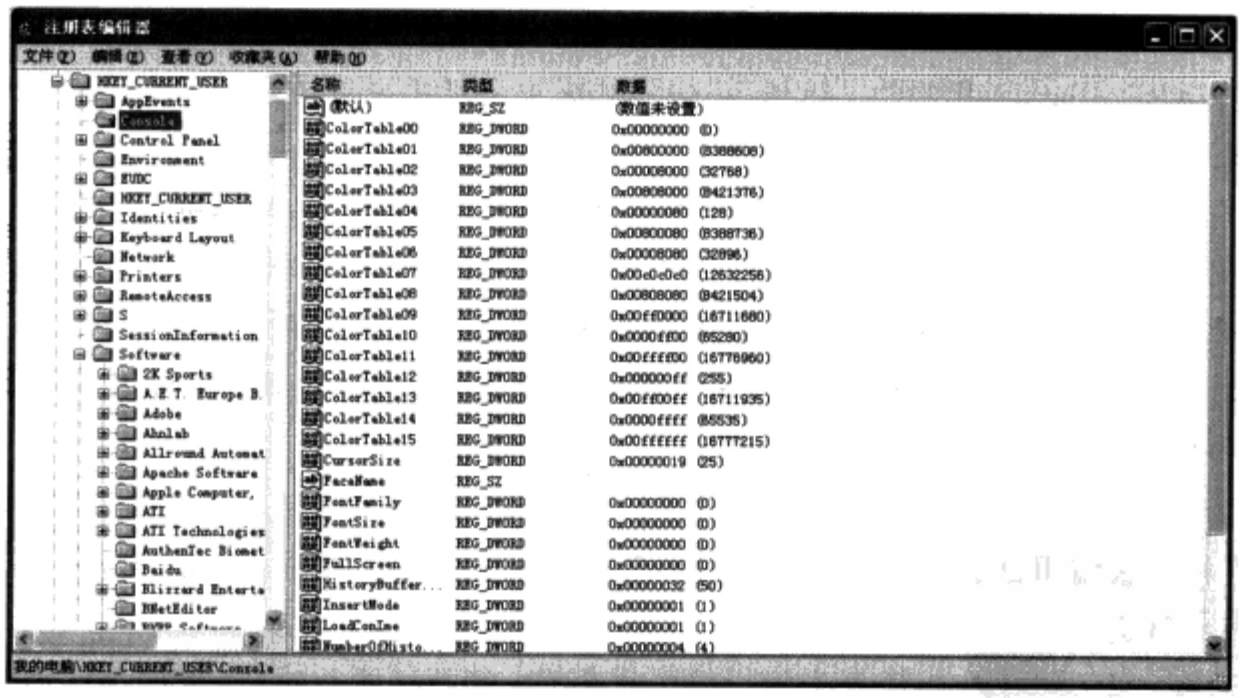


图 12.16 注册表编辑器

1. 创建键

首先介绍 RegCreateKey()函数，该函数的作用是创建指定的注册表项，如果这个表项已经存在，则打开这个表项。该函数的声明如下：

```
LONG RegCreateKey(HKEY hKey ,LPCTSTR lpSubKey ,PHKEY phkResult);
```

该函数的返回值是 LONG 类型的，0（ERROR_SUCCESS）表示成功，其他任何值都代表一个错误代码。

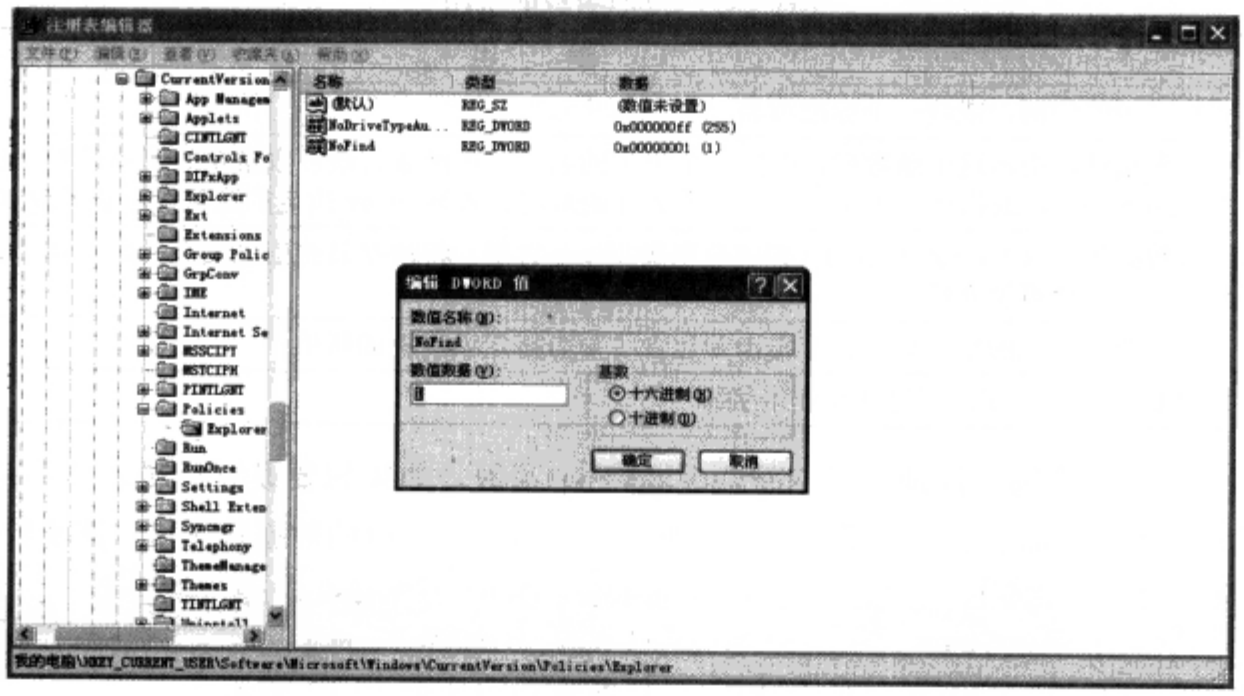


图 12.17　修改 NoFind 键的值

该函数的参数说明如表 12.10 所示。

表 12.10　　RegCreateKey()函数的说明

参　数	说　明
hKey	要打开项的句柄，或者一个标准项名。标准项名实际上就是注册表中的几个主键： • HKEY_USERS • HKEY_CURRENT_USER • HKEY_CURRENT_CONFIG • HKEY_CLASSES_ROOT • HKEY_LOCAL_MACHINE
lpSubKey	字符串指针，指示这个函数欲创建或打开的新子项名。这里可同时创建多个项，只需用反斜杠将它们分隔开即可。例如 level1\level2\newkey
phkResult	指定一个变量，用于装载新子项的句柄，当不再需要此返回的注册表项句柄时，调用 RegCloseKey()函数关闭这个句柄

2．打开键

如果想要处理注册表，需要打开注册表中的一个键，可以使用 RegOpenKey()函数来实现。该函数用来打开一个现有的注册表项，其声明如下：

```
LONG RegOpenKey (HKEY hKey ,LPCTSTR lpSubKey ,PHKEY phkResult);
```

该函数的返回值是 LONG 类型的，0（ERROR_SUCCESS）表示成功，其他任何值都代表一个错误代码。

该函数的参数和上面介绍的 RegCreateKey()函数的参数都是相同的，这里不再赘述。

3．写入注册表

前面曾介绍如何使用 Windows 的注册表编辑器来修改注册表，那么如何使用 API 函数来修改注册表呢？这里，Windows 提供了 RegSetValue()函数来完成这个功能，其声明如下：

```
LONG RegSetValue (HKEY hKey ,LPCTSTR lpSubKey ,DWORD dwType ,LPCTSTR lpData ,DWORD cbData);
```

该函数的返回值是 LONG 类型的，0（ERROR_SUCCESS）表示成功，其他任何值都代表一个错误代码。

接着，再来学习一下该函数的参数。通过前面的学习，可以知道在注册表中，对值来说，包含 3 个部分：值名称、值类型和值本身。该函数的参数，即对应这 3 个值的修改，如表 12.11 所示。

表 12.11　　RegSetValue()函数的说明

参　数	说　明
hKey	要打开项的句柄，或者一个标准项名
lpSubKey	字符串指针，指示这个函数欲修改的一个子项的名称。该函数为该参数指定的子项设置默认值。如果此参数为 NULL，或指向一个空字符串，那么此函数将为参数 hKey 指定的注册表项设置默认值
dwType	该参数必须是 REG_SZ 类型的（即字符串类型），它指示将被存储的信息类型。如果要存储其他类型的数据，下面会再做介绍
lpData	指向一个空终止的字符串，该字符串中包含了要为指定项设置的新值
cbData	指定 lpData 的长度，不包括空中止字符

下面介绍一下另一个写入注册表的 RegSetValueEx()函数。如果想要设置注册表项不是默认值，即有名称的数据和类型时，需要调用该函数，在注册表项下设置指定值的数据和类型。该函数的声明如下：

```
LONG RegSetValueEx (HKEY hKey ,LPCTSTR lpValueName ,DWORD Reserved,
                    DWORD dwType ,CONST BYTE *lpData ,DWORD cbData);
```

该函数的返回值与 RegSetValue()函数相同，其参数介绍如表 12.12 所示。

表 12.12　　RegSetValueEx()函数说明

参　数	说　明
hKey	一个已打开项的句柄，或指定一个标准项名
lpValueName	指向字符串的指针，包含了要设置值的名称。若拥有该名称的值并不在指定的注册表项中，则此函数将其加入该项中
Reserved	暂时未用，设为 0
DwType	指定要被存储的值的数据类型，可取值参考 MSDN 帮助手册
lpData	包含数据的缓冲区中的第一字节，其中的数据是指定名称所要更新的值
cbData	指定 lpData 的长度，不包括空中止字符

4．从注册表中读数据

Windows API 提供了 RegQueryValue()函数来获得与指定注册表默认的值或未命名的值相关的数据，该函数的声明如下：

```
LONG RegQueryValue (HKEY hKey ,LPCTSTR lpSubKey ,LPTSTR lpValue .PLONG
                    lpcbValue);
```

该函数的返回值是 LONG 类型的，0（ERROR_SUCCESS）表示成功，其他任何值都代表一个错误代码。

该函数的参数如表 12.13 所示。

表 12.13　　RegQueryValue()函数说明

参　数	说　明
hKey	一个已打开项的句柄，或者指定一个标准项名
lpSubKey	字符串指针，指示这个函数欲获取的子项名。可设为 vbNullString，表示获取 hKey 的值
lpValue	用于容纳指定项值的一个字符串
lpcbValue	指定一个变量，用于装载 lpValue 缓冲区的长度。一旦返回，它会设为实际载入缓冲区的字节数量

同样，如果想要读取注册表项不是默认值，即有名称值的数据和类型时，可以调用 RegQueryValueEx()函数，该函数将读取注册表项下指定值的数据和类型。该函数的声明如下：

```
LONGRegQueryValueEx(HKEYhKey,LPCTSTRlpValueName,LPDWORDlpValueName,LPDWORDlpType,LPBYTElp
Data,LPDWORD lpcbData);
```

该函数的返回值与 RegQueryValue()函数相同，其参数介绍如表 12.14 所示。

表 12.14　　RegQueryValueEx()函数说明

参　数	说　明
hKey	一个已打开项的句柄，或者指定一个标准项名
lpValueName	指向字符串的指针，包含要查找值的名称。如果此参数为空，则 RegQueryValueEx()函数将获得该注册表项的未命名值或默认值的类型及数据，前提是这些值和数据存在
lpReserved	暂时未用，设为 0
lpType	指定要被查询的值的数据类型
lpData	一个指向缓冲区的指针，该缓冲区接收该值的数据
lpcbData	指定 lpData 的长度，不包括空中止字符

12.4.3　访问并修改注册表

在单文档 MFC 工程 File 的菜单栏中添加一个新的菜单“操作注册表”，然后给这个菜单添加两个菜单命令，分别为“写入注册表”和“读取注册表”，它们的 ID 分别为 ID_REG_REGEDIT_WRITE 和 ID_REG_REGEDIT_READ。结果如图 12.18 所示。

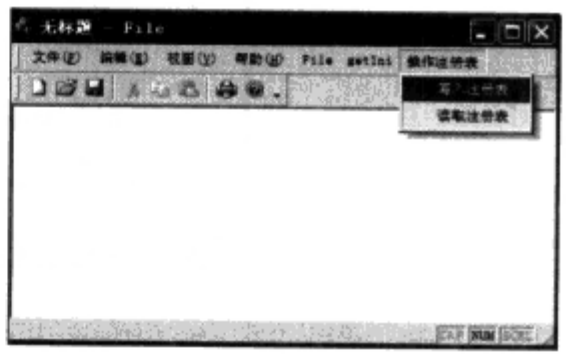


图 12.18　为 File 工程添加新的菜单

在 CFileView 类中，为上述两个菜单命令添加两个消息处理函数，分别为 OnRegRegeditRead()和 OnRegRegeditWrite()，然后在 OnRegRegeditWrite()函数中添加如下代码来写入注册表：

```
void CFileView::OnRegRegeditWrite()
{
    // TODO: Add your command handler code here
    HKEY m_hKey;
                                        //创建子键
    RegCreateKey(HKEY_LOCAL_MACHINE,_T("software\\VisualC++2010 " ),&m_hKey);
                                        //设置键值
    RegSetValue(m_hKey,_T("第 12 章"),REG_SZ,_T("Visual C++ 2010 MFC 文件与注册表操作"),
        strlen("Visual C++ 2010 MFc 文件与注册表操作"));
                                        //关闭注册表项句柄
    RegCloseKey(m_hKey);
    MessageBox(_T("已写入注册表"));
}
```

在上面的代码中，首先创建了一个注册表项句柄变量——hKey；然后调用 RegCreateKey()函数创建一个注册表项，即在主键 HKEY_LOCAL_MACHINE 下的 SOFTWARE 键中建立了一个子键“Visual C++ 2010”；接着调用 RegSetValue()函数设置注册表项的值，这里在“VC 教学”子键下添加一个名为“第 12 章”的键，设置其键值为“Visual C++ 2010 MFC”。最后调用 RegCloseKey()函数关闭开始创建的注册表项句柄。

编译并运行该程序，选择“操作注册表”→“写入注册表”命令，会弹出已写入注册表的消息对话框。打开注册表，就可以看到新添加的键和键值，如图 12.19 所示。

运行结束后，再在消息处理函数 OnRegRegeditRead()中实现注册表的读取，其代码如下：

```
void CFileView::OnRegRegeditRead()
{
    // TODO: Add your command handler code here
    LONG lRead;
    RegQueryValue(HKEY_LOCAL_MACHINE,_T("software\\Visual C++ 2010\\
```

```
                第 12 章") , NULL , &lRead);
    wchar_t * pStr=new wchar_t[lRead];
    RegQueryValue(HKEY_LOCAL_MACHINE,_T("software\\Visual C++ 2010\\
                第 12 章") ,pStr,&lRead);
    MessageBox(pStr);
}
```

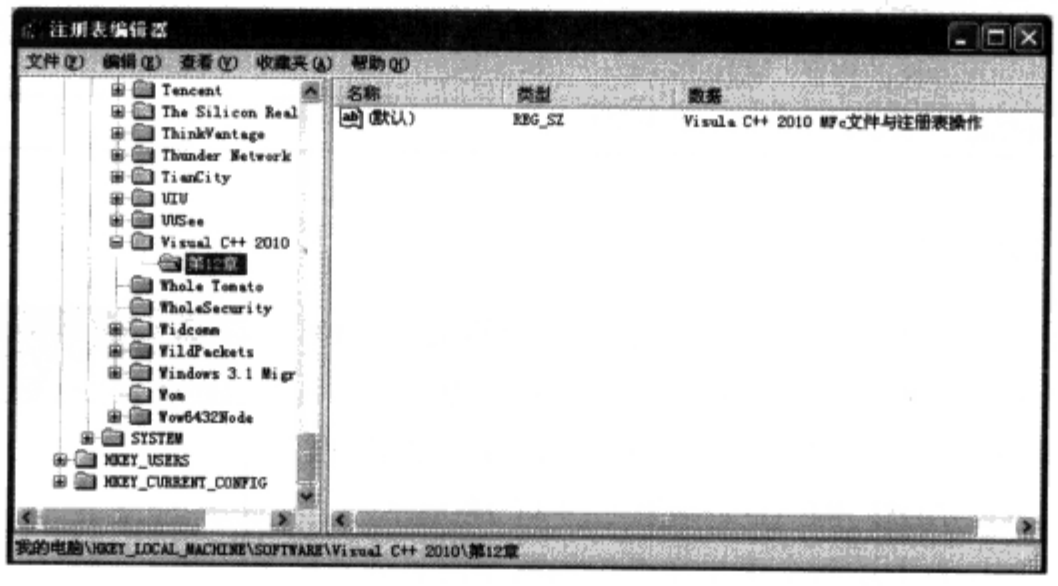


图 12.19　修改后的注册表

在消息处理函数 OnRegRegeditRead()中两次调用 RegQueryValue()函数，第一次调用时将第三个参数置为 NULL，目的是让系统自动获得返回的键值的长度，并将该长度存放到事先声明的变量 lRead 中；然后又声明了一个变量 pStr，并为其分配由 lRead 指定大小的内存空间；最后，第二次调用 RegQueryValue()函数时，将变量 pStr 带入第三个参数用于获取键值，而第四个参数就是已经得到的键值长度 lRead；最后使用 MessageBox()函数将读取的数据显示出来。

编译并运行程序，选择“操作注册表”→“写入注册表”命令，会弹出如图 12.20 所示的消息对话框，可以看到程序成功读到了注册表中的内容。

以上就是通过 Win32 API 提供的函数对注册表进行的操作，所介绍的只是其中的一部分函数，其他读者可以通过查找相关资料再做了解。另外，操作注册表还有另一种方法，就是使用类 CWinApp 的成员函数 SetRegistryKey()、GetProfileInt()、WriteProfileInt()，只是这些较古老的注册表操作函数存在很大的局限性，比如由这些旧函数写入注册表的数据只能写入 HKEY_CURRENT_USER\Software 键下，因此这种方法用得不多了，读者简单地了解即可，感兴趣的读者可参看 MSDN 帮助文档。

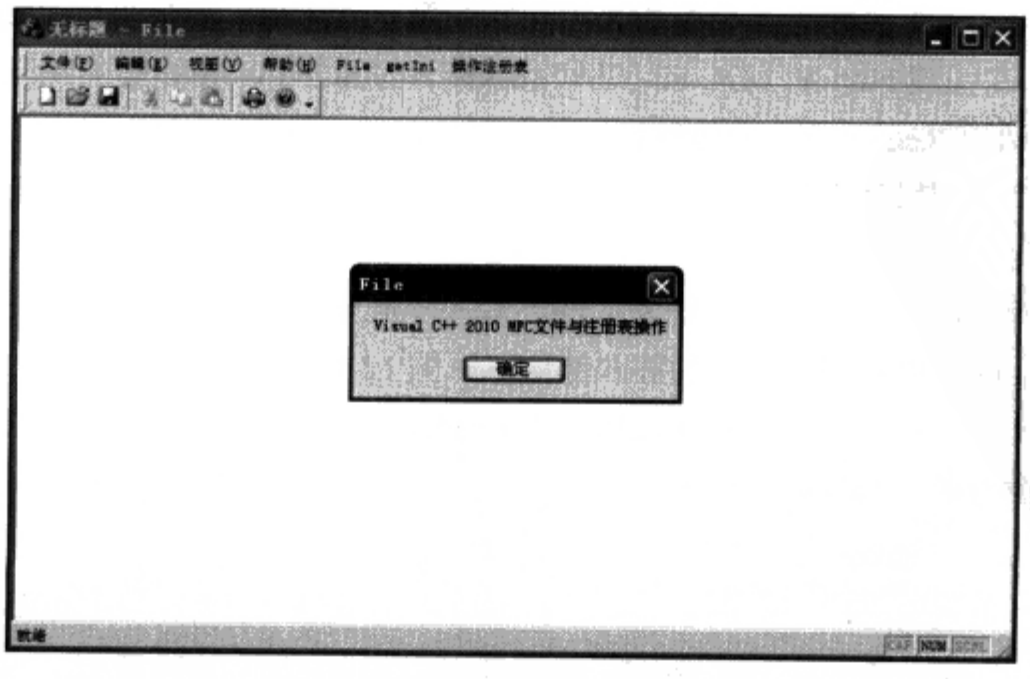


图 12.20　读取注册表的运行结果

12.5　小结

本章详细介绍了 Visual C++ 2010 MFC 文件与注册表编程的方法，包括如何使用 MFC 支持的文件串行化过程，CFile 类的使用方法，.ini 文件和注册表的读写操作等。本章的重点是。

- 了解 Serialize()函数和 CArchive 类的定义和使用方法。串行化过程使用 CArchive 对象执行输入和输出，要使用传递到 Serialize()函数的 CArchive 对象对类的数据成员进行串行化。
- 实现对自定义文档和类的串行化需要经过的步骤。
- CFile 类的定义和使用方法，使用该类实现文件的打开、读写、定位、关闭和其他管理操作，以及对异常的捕获和处理。
- .ini 文件的作用、结构，使用 WriteProfileString()函数和 GetProfileString()函数读写 Win.ini 文件的方法。
- 注册表的作用、结构和修改方法，包括使用 regedit.exe 来访问、修改注册表，以及使用 Win32 API 函数来访问、修改注册表。

第 13 章　Visual C++ 2010 MFC 数据库开发

数据库变得越来越重要，如何编写程序来高效地访问数据库就变得十分重要。本章介绍了如何使用 Visual C++和 MFC 连接数据库，以及如何访问、更新数据库中的数据。本章只是介绍对数据库的基本操作方法，这是因为详尽地讲解如何使用 Visual C++进行数据库应用开发足以写成一本书。本章首先介绍了一些数据库的基本概念。

13.1　数据库基本知识

数据库（DataBase，简称 DB）可以被视为能够进行自动查询和修改的数据集，是人们有效地进行数据存储、管理和共享的工具。数据库有多种类型，从最简单的存储各种数据的表格到能够进行海量数据存储的大型数据库系统都在各个领域得到了广泛的应用。目前，最常用的数据库类型是关系型数据库（Relational DataBase），在关系型数据库中，数据被组织成一个或多个表（Table）。表含有行（Row）和列（Column），每一行代表一个记录（Record），这是数据组织的基本单位；每一列含有一个列名，代表一个域（Field），同一列表示相同的域的不同取值。一个行或记录包含一个或多个域，可以通过列名来索引记录中的各个域的值。每个表都应该至少有一个特殊的列——主键（Primary Key），各个记录在这一列的取值是惟一的。

下面通过一个例子——Northwind 数据库来讲解这些概念，这个数据库中包含多个表，如图 13.1 所示。

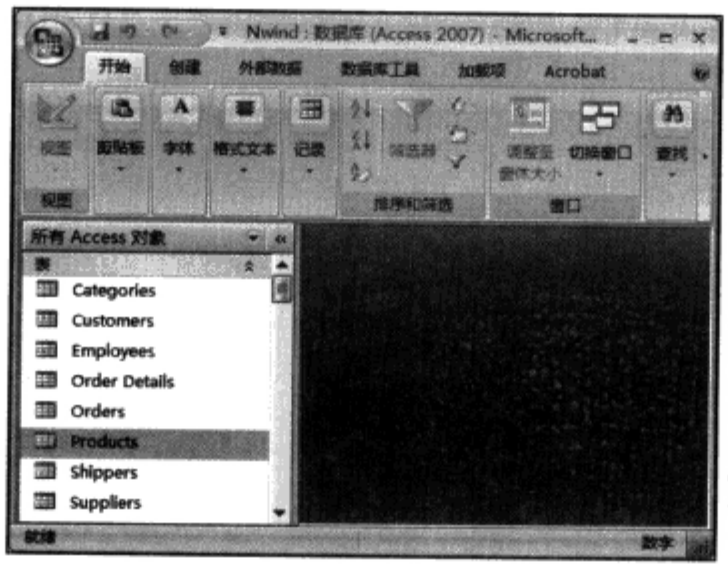


图 13.1　Northwind 数据库

查看其中的 Products 表，这是一个存放产品相关信息的表，一共有 10 列，这些列的列名依次是 Product ID、Product Name、Supplier、Category、Quantity Per Unit、Unit Price、Units In Stock、Units In Order、Reorder Level、Discontinued。

截取其中的一部分数据，如图 13.2 所示。

Products 表的主键是 Product ID，不同行中的 Product ID 域取值都不相同，每一行（记录）含有 10 个域，这就是一个表的直观表示。

图 13.2　Products 表的部分数据

13.2　SQL 语言的基础知识

结构化查询语言 SQL（Structured Query Language），是一种专用于查询或修改关系型数据库的查询语言，最早由 IBM 公司开发出来，现在已经成为 ISO 国际标准，被广泛应用于数据库管理系统。SQL 是一种简单、易懂的查询语言，主要包括 6 种句式，如表 13.1 所示。

表 13.1　　SQL 语言的 6 种句式

句　式	说　明
SELECT	用于查询数据库，可以一次查询一个或多个表，选择需要的列，并设置查询条件，形成特定的查询结果集，这是最常用的 SQL 命令
CREATE	主要用于在数据库中创建新表
DROP	删除数据库中的表
INSERT	向数据库中的表插入记录
DELETE	删除数据库中表中的满足特定条件的记录
UPDATE	修改数据库中表中的记录的值

对数据库的查询，还可以设置查询条件，这通过 SQL 从句实现，最常用的 SQL 从句在表 13.2 中列出。

表 13.2　　SQL 语言的最常用从句

从　句	说　明
FROM 从句	用于指定要查询的表的名称，可以有多个表，各个表之间使用逗号（,）隔开
WHERE 从句	指定查询条件，只有那些满足条件的记录才包含在结果中
ORDER BY 从句	指定查询结果集按照指定的列的顺序排列，可以升序（ASC），也可以降序（DESC）

如果在从句中需要多个查询条件，需要逻辑运算符来组合这些查询条件。SQL 的逻辑运算符主要有 3 种。

- AND——逻辑“与”，即仅当两个条件都满足时才为真。
- OR——逻辑“或”，即仅当两个条件都不满足时才为假。
- NOT——逻辑“非”，即取反操作。

对于数值类型的域，还可以通过比较运算来设置查询条件，这些比较运算符与 C 语言内的相似，常用的有<（小于）、>（大于）、<=（不大于）、>=（不小于）、=（等于）、BETWEEN（指定数据范围）。

13.3 ODBC 访问数据库

ODBC（Open Database Connectivity，开放数据库互连）是 20 世纪 80 年代末、90 年代初出现的技术，它为编写关系型数据库的客户软件提供了一种标准的 API 方法来访问数据库管理系统（DBMS）。这些 API 利用 SQL 来完成其大部分任务，ODBC 本身也提供对 SQL 语言的支持，用户可以直接将 SQL 语句传给 ODBC 处理。ODBC 具有独立性和开放性的特点：与具体的编程语言无关，与具体的数据库系统无关，与具体的操作系统无关。使用 ODBC 可以访问多种不同类型的数据库，只要提供相应的 ODBC 驱动。ODBC 驱动充当用户应用程序与 DBMS 之间的桥梁，用户程序只与 ODBC 驱动通信，发送各种请求。ODBC 驱动负责将用户程序请求（函数调用）转换为特定 DBMS 的命令，从而实现对数据库的访问。

ODBC API 是比较低层的数据库接口，在较低的层次上使客户应用程序配置并操作数据库。直接使用 ODBC 的 API 可以实现对程序的最大控制，但是由于 ODBC API 过于冗杂，直接使用将比较困难。在 Visual C++中，MFC 提供了一些类对 ODBC 进行了封装，提供了对象支持以简化 ODBC API 的调用，大大降低了 ODBC 编程的复杂性 MFC 提供的对 ODBC 进行封装的类主要有 5 个，如表 13.3 所示。

表 13.3　　MFC 提供的对 ODBC 进行封装的类

MFC 提供的类	说　明
CDatabase	代表与数据库的一个连接，在对数据库进行任何操作之前，必须先建立一个与它的连接
CRecordset	代表对数据库执行查询操作（SQL 的 SELECT 语句）返回的结果集合。这是一个记录的集合，每次只能查看一个记录，通过该类提供的成员方法可以向前或向后移动当前记录。用户程序需要从这个类继承以满足特定的应用
CRecordView	数据库视图类，用于显示对应的记录集的当前记录信息。这个视图本质上是一个对话框，通过对话框数据交换（DDX）机制实现记录在控件上的显示
CFieldExchange	实现在记录结果集与数据库之间交换数据，这个过程将由 MFC 向导自动完成
CDBException	代表使用 ODBC 对数据库操作时出现的异常，通过捕获操作过程中的异常，可以获得操作失败的有用信息

13.3.1 注册 ODBC 数据库

在通过 ODBC 访问数据库之前，必须先向操作系统注册这个数据库。依次选择“控制面板”→“管理工具”→“数据源（ODBC）”命令，进入“ODBC 数据源管理器”对话框（“控制面板”命令在“开始”菜单中），在该对话框中，即可注册数据库，如图 13.3 所示。

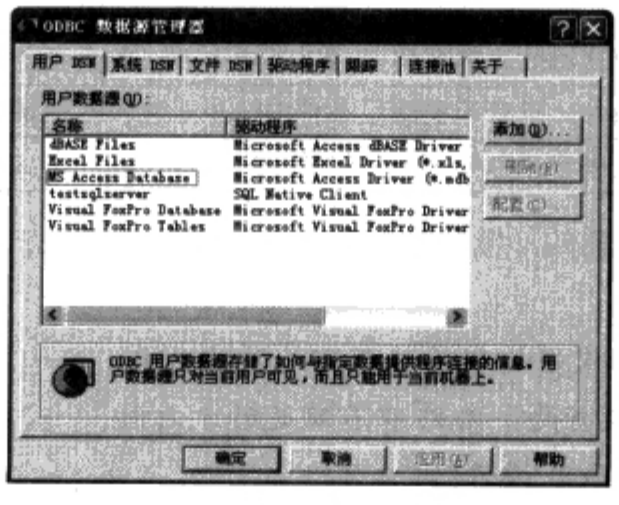


图 13.3　ODBC 数据源管理器

“ODBC 数据源管理器”对话框中列举出了所有注册过的数据库，“用户 DSN”表示注册的数据

库只能供用户自己使用，而“系统 DSN”则表示对这台计算机上的所有用户都可用，“文件 DSN”则允许通过网络访问数据库，这里使用“用户 DSN”。

单击“添加”按钮，打开“创建新数据源”对话框，如图 13.4 所示，其中列举出了所有的 ODBC 驱动程序，选择一种合适的驱动，如此处使用的是 Access 数据库，所以选择“Microsoft Access Driver (*.mdb)”驱动，选好后单击“完成”按钮。

接着，打开“ODBC Microsoft Access 安装”对话框，如图 13.5 所示。首先输入数据源名及其描述，然后单击“选择”按钮，去寻找已经存在的数据库文件。如果想新建一个数据库文件，就单击“新建”按钮。

图 13.4　选择数据源的驱动程序

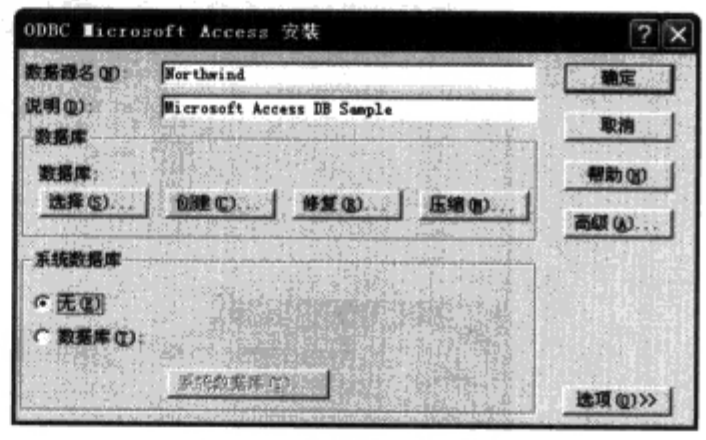


图 13.5　ODBC Microsoft Access 安装

进入“选择数据库”对话框，如图 13.6 所示。在中间的“目录”列表框中选择数据库文件所在的目录，在左侧会显示目录下的所有数据库文件，选择好一个数据库文件，这里选择“Nwind.mdb”，单击“确定”按钮。

现在回到了“ODBC 安装”对话框，其中显示了设置的数据库文件的目录，如图 13.7 所示，确认数据库文件无误后，单击“确定”按钮就完成了数据库的注册工作。

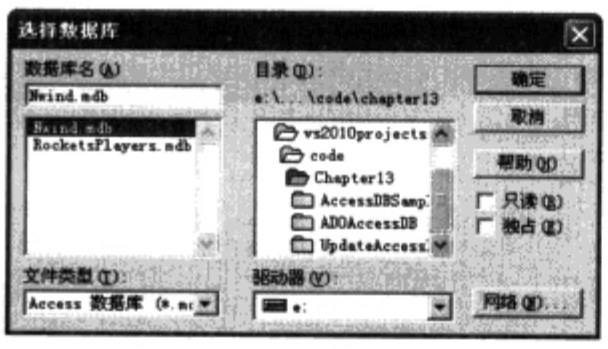


图 13.6　选择数据库

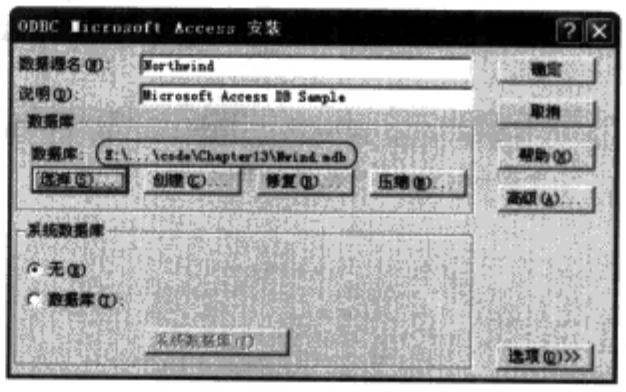


图 13.7　ODBC Microsoft Access 安装

这时，回到“ODBC 数据源管理器”对话框，就能看到刚刚注册的数据库文件信息，如图 13.8 所示。如果还想继续添加就单击“添加”按钮，否则就单击“确定”按钮完成。

图 13.8　ODBC 数据源管理器

13.3.2 创建一个 MFC 的 ODBC 程序

打开 Visual Studio2010 程序，选择“File”→“New”→“Project”命令或按【Ctrl+Shift+N】组合键，进入“New Project”对话框。利用 MFC 向导来自动生成访问数据库的代码可以减少很多烦琐的工作，在“Installed Templates ”下选择“MFC”→“MFCApplication”类型，在对话框下方输入工程名，并设置工程的保存路径后，单击“OK”按钮，如图 13.9 所示。

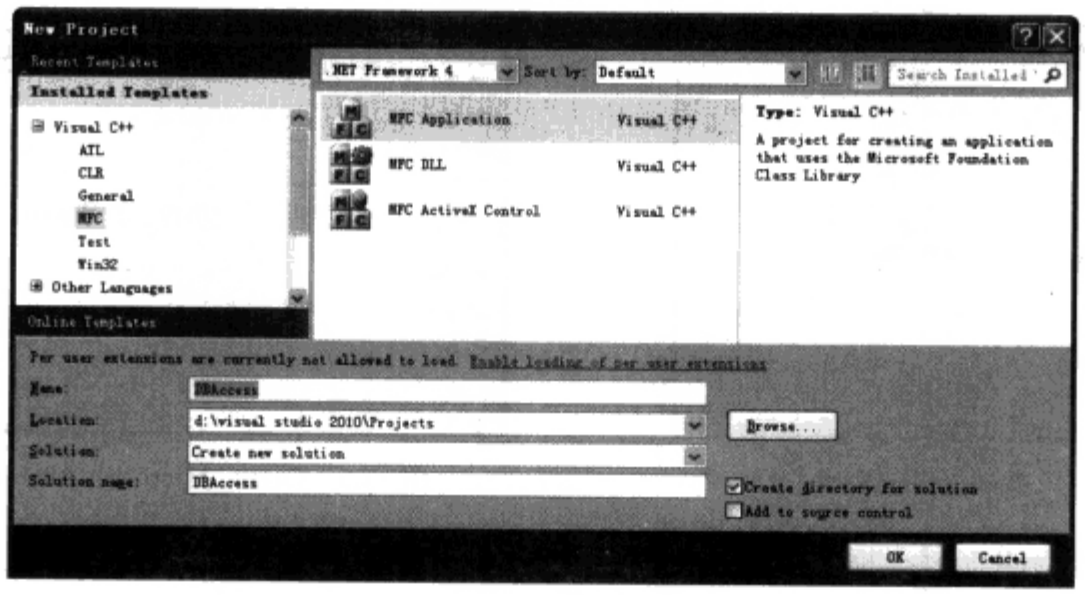


图 13.9　新建一个 MFC 应用程序工程

进入 MFC 向导对话框，这里一共有 6 步，第一步不需要进行什么设置，直接单击“Next”按钮，如图 13.10 所示。

在第二步选择 MFC 应用类型及其他一些设置，选中“Single document”单选按钮，其他使用默认设置，如图 13.11 所示，单击“Next”按钮进入第三步。

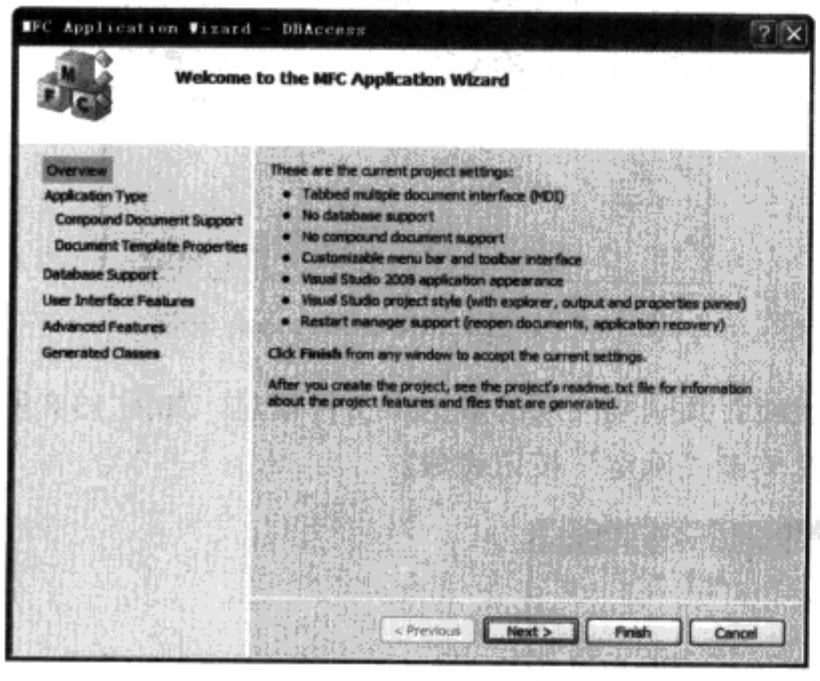


图 13.10　MFC 向导第 1 步

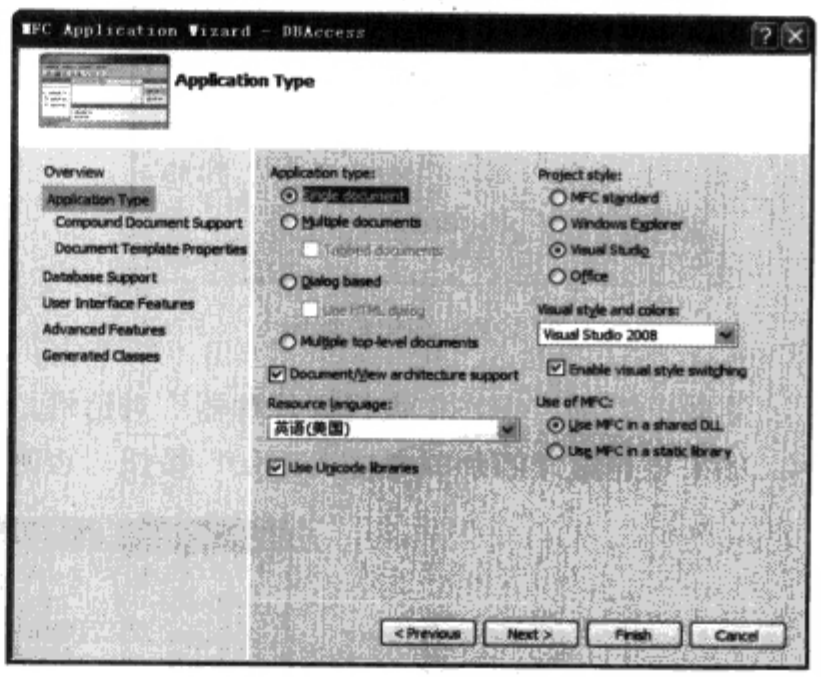


图 13.11　选择单文档类型

在“Database support”选项组中选中“Database view without file support”单选按钮，在“Client type”选项组中选中“ODBC”单选按钮，在右侧的“Type”选项组中选中“Snapshot”单选选钮，如图 13.12 所示。

然后单击“Data Source”按钮，进入“选择数据源”对话框。在“机器数据源”选项卡中选择将访问的数据库，这里选择“Northwind”数据库，如图 13.13 所示。单击“确定”按钮后，弹出一个“登

录”对话框。

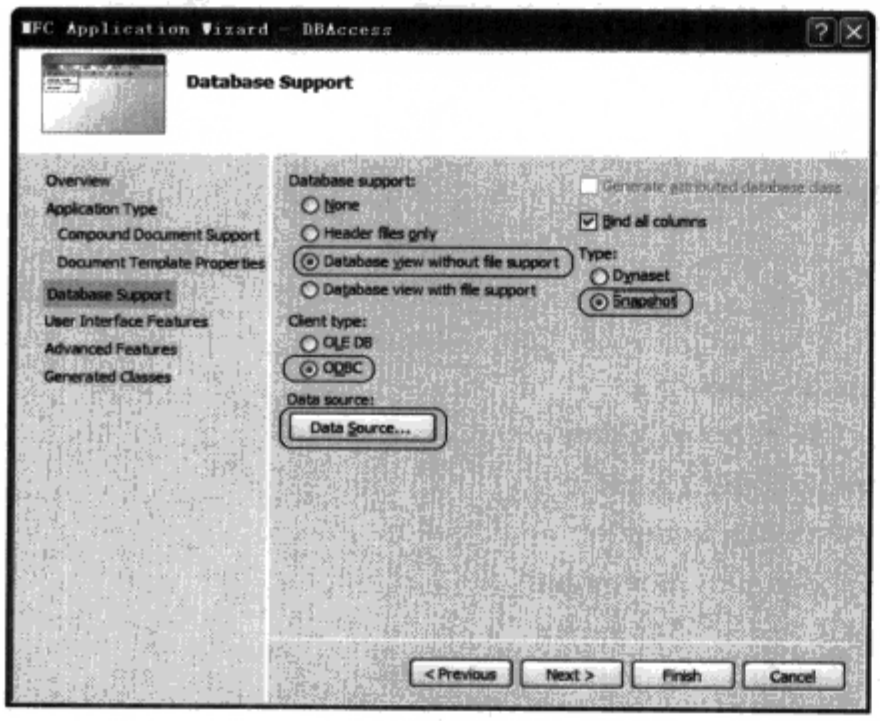


图 13.12　MFC 向导设置对数据库的支持

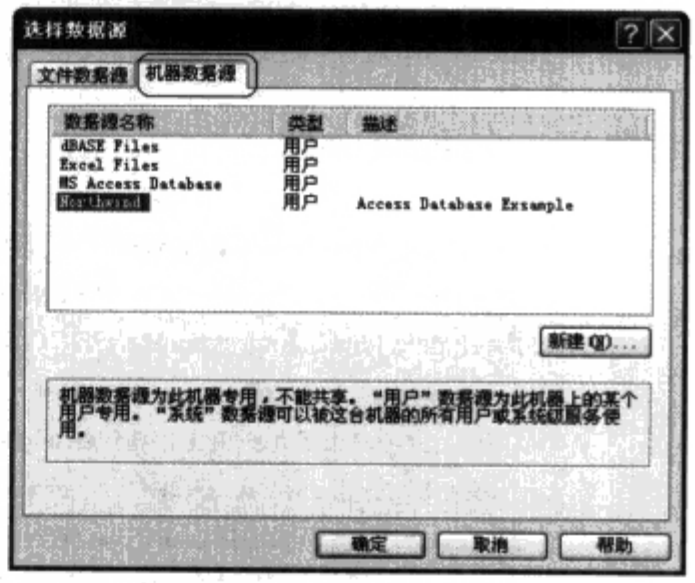


图 13.13　选择数据源

其中“登录名称”和“密码”就是进入本地机器的账户和密码，填写好后单击“确定”按钮，如图 13.14 所示。

弹出“Select Database Object”对话框，在其中选择将在程序中访问的表，这里只访问 Products 表，如图 13.15 所示。设置完成之后，单击“OK”按钮。

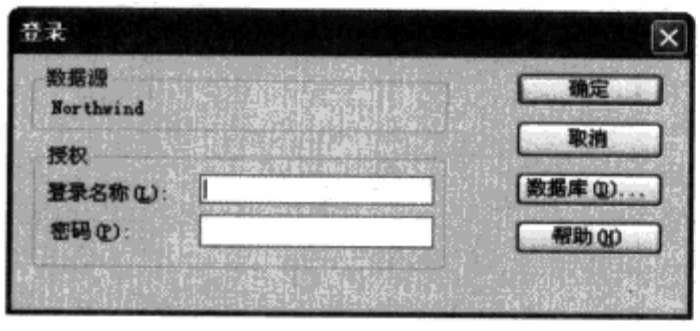


图 13.14　登录对话框

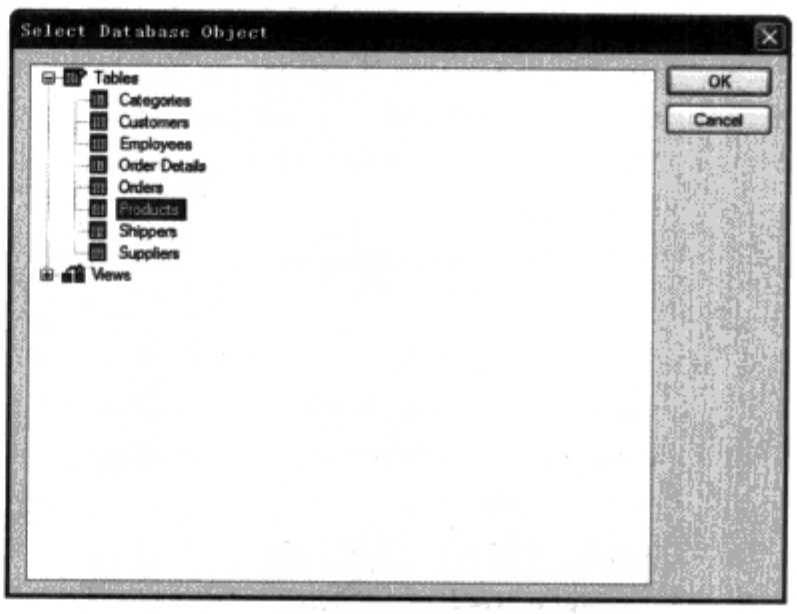


图 13.15　选择数据库对象

MFC 向导后面的几步都保持默认设置，单击“Finish”按钮完成向导。

13.3.3　程序结构分析

利用 MFC 向导生成的程序结构与普通的 MFC 单文档程序差不多，都包括一个应用程序类 CAccessDBSampleApp、一个主窗体类 CmainFrame、一个文档类 CAccessDBSampleDoc、一个视图类 CAccessDBSampleView，以及一个 CAccessDBSampleSet 类。

1．CAccessDBSampleSet 类

CAccessDBSampleSet 类继承自 CRecordset 类，主要用于存放数据库查询的结果，CaccessDB SampleSet 类的内部定义和实现如下：

```
//从 CRecordset 继承
class CChap13_ODBC_DatabaseSet : public CRecordset
{
public:
    CChap13_ODBC_DatabaseSet(CDatabase* pDatabase = NULL);
    DECLARE_DYNAMIC(CChap13_ODBC_DatabaseSet)

    // Field/Param Data

    // The string types below (if present) reflect the actual data type of the
    // database field - CStringA for ANSI datatypes and CStringW for Unicode
    // datatypes. This is to prevent the ODBC driver from performing potentially
    // unnecessary conversions.  If you wish, you may change these members to
    // CString types and the ODBC driver will perform all necessary conversions.
    // (Note: You must use an ODBC driver version that is version 3.5 or greater
    // to support both Unicode and these conversions).
    //这里向导根据访问的数据库自动生成这些成员，用于存放一个记录内的各列数据
    long m_ProductID;   //Number automatically assigned to new product.
    CStringW     m_ProductName;
    long m_SupplierID;  //Same entry as in Suppliers table.
    long m_CategoryID; //Same entry as in Categories table.
    CStringW     m_QuantityPerUnit;        //(e.g., 24-count case, 1-liter bottle).
    double   m_UnitPrice;
    int   m_UnitsInStock;
    int   m_UnitsOnOrder;
    int   m_ReorderLevel;       //Minimum units to maintain in stock.
    BOOL   m_Discontinued;  //Yes means item is no longer available.
    // Overrides
    // Wizard generated virtual function overrides
public:
    virtual CString GetDefaultConnect();     // Default connection string
    virtual CString GetDefaultSQL();         // default SQL for Recordset
    virtual void DoFieldExchange(CFieldExchange* pFX);       // RFX support
    // Implementation
    #ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
    #endif
};
```

可以看到，构造函数要求传入一个 pDatabase 指针，默认值为 NULL。如果传入 NULL，则 CRecordset 将根据 GetDefaultConnet()函数返回的数据库连接信息自动建立一个数据库连接，并存储在成员 m_pDatabase 内。中间的那些成员变量是向导根据要访问的数据库表的列名和数据类型自动生成的，在本例中要求访问所有的列，所以可以看到 Products 表的所有列名（前面加了前缀 m_）。

GetDefaultSQL()函数的实现如下：

```
CString CChap13_ODBC_DatabaseSet::GetDefaultSQL()
{
    return _T("[Products]");
}
```

GetDefaultSQL()返回的 SQL 语句将被作为 CRecordset 的 Open()函数的参数来查询数据库，并将结果存放在 CRecordset 中。

本例中只是简单地返回一个表名，实际上这相当于下面的 SQL 语句：

```
SELECT * FROM [Products];
```

完全可以用这个完整的 SQL 语句来替换。另外，还可以加入 WHERE 从句或 ORDER 从句来查询。比如可以将这个函数改为：

```
return _T("SELECT * FROM [Products] WHERE [UnitPrice] >50 ORDER BY\[UnitPrice]");
```

但是，在后面将看到可以使用更简单的方法来设置这些查询条件。

CAccessDBSampleSet 的构造函数如下：

```
CChap13_ODBC_DatabaseSet::CChap13_ODBC_DatabaseSet(CDatabase* pdb)
    : CRecordset(pdb)
{
    m_ProductID = 0;
    m_ProductName = L"";
    m_SupplierID = 0;
    m_CategoryID = 0;
    m_QuantityPerUnit = L"";
    m_UnitPrice = 0.0;
    m_UnitsInStock = 0;
    m_UnitsOnOrder = 0;
    m_ReorderLevel = 0;
    m_Discontinued = FALSE;
    m_nFields = 10;
    m_nDefaultType = snapshot;
}
```

构造函数除了设置那些存储表的各列的成员变量外，还初始化 m_nFields 变量，m_nFields 代表要访问的列的个数，本例中为 10。如果用户只需要访问其中的某几列，则 m_nFields 的值应该随之变化。最后设置的数据库类型 m_nDefaultType 为 snapshot。

CAccessDBSampleSet 对象在文档类内声明，由于没有传入一个打开的 CDatabase 指针，在初始化之后，CAccessDBSampleSet 对象将根据 GetDefaultConnect()返回的连接信息，自动构造一个数据库连接。GetDefaultConnect()函数实现如下：

```
//#error Security Issue: The connection string may contain a password
// The connection string below may contain plain text passwords and/or
// other sensitive information. Please remove the #error after reviewing
// the connection string for any security related issues. You may want to
// store the password in some other form or use a different user authentication.
CString CChap13_ODBC_DatabaseSet::GetDefaultConnect()
{
    return _T(
        "DSN=Northwind;\
        DBQ=E:\\GUCAS\\VS2010PROJECTS\\Nwind.mdb;\
        DriverId=281;\
        FIL=MS Access;
        MaxBufferSize=2048;
        PageTimeout=5;
        UID=admin;"
```

```
        );
    }
```

由于在访问数据库之前，要求输入用户 ID 及密码，这在向导中已经输入过了，这就导致在连接信息中可能会明文显示出用户的 ID 及密码，这是不安全的，所以向导在 GetDefaultConnect()函数的上方有一个#error 的提示信息：

```
//#error Security Issue: The connection string may contain a password
```

希望用户确认没有将密码显示出来，这条语句是无法正常编译通过的，只需简单将这一句注释掉即可 。

CDatabase 在调用 Open()函数连接数据库时，需要传入一个包含连接信息的字符串，GetDefaultConnect()就用于提供这个连接信息，这是由向导根据用户设置的数据源信息自动生成的，其中每个参数的含义如表 13.4 所示。

表 13.4　　GetDefaultConnect()函数的各个参数

连 接 参 数	说　　明
DSN	建立连接的数据源，这是在前面注册时提供的
DBQ	数据源文件的路径信息
DriverID	数据库对应的 ODBC 驱动的 ID 号
FIL	数据库文件的类型，本例中使用 MS Access
MaxBufferSize	用于数据传输的最大缓冲区的大小
PageTimeout	等待数据库连接的最长时间，单位为秒。这个参数对于远程连接访问的数据库很重要
UID	访问数据库的用户 ID

其实还需要提供访问密码，但向导已经把密码隐藏起来。如果用户希望在程序运行时再指定数据源，则可以将这个函数改为

```
return _T("ODBC");
```

这样在程序运行时就会弹出一个对话框，要求用户选择数据源。

下面介绍如何在数据源和 CRecordset 之间交换数据。前面看到向导自动为每一列生成对应的成员变量来存放一个记录的每个域。通过 DoFieldExchange()函数来实现在数据源和 Recordset 对象间的数据传递数据：

```
void CChap13_ODBC_DatabaseSet::DoFieldExchange(CFieldExchange* pFX)
{
    pFX->SetFieldType(CFieldExchange::outputColumn);
    // Macros such as RFX_Text() and RFX_Int() are dependent on the
    // type of the member variable, not the type of the field in the database.
    // ODBC will try to automatically convert the column value to the requested
    //type
    RFX_Long(pFX, _T("[ProductID]"), m_ProductID);
    RFX_Text(pFX, _T("[ProductName]"), m_ProductName);
    RFX_Long(pFX, _T("[SupplierID]"), m_SupplierID);
    RFX_Long(pFX, _T("[CategoryID]"), m_CategoryID);
    RFX_Text(pFX, _T("[QuantityPerUnit]"), m_QuantityPerUnit);
    RFX_Double(pFX, _T("[UnitPrice]"), m_UnitPrice);
    RFX_Int(pFX, _T("[UnitsInStock]"), m_UnitsInStock);
    RFX_Int(pFX, _T("[UnitsOnOrder]"), m_UnitsOnOrder);
    RFX_Int(pFX, _T("[ReorderLevel]"), m_ReorderLevel);
    RFX_Bool(pFX, _T("[Discontinued]"), m_Discontinued);

}
```

向导根据变量类型自动选择对应的 RFX_XXX 函数来进行数据交换。首先第一条语句：

```
pFX->SetFieldType(CFieldExchange::outputColumn)
```

设置为表的列数据交换，这是双向的。

下面的 RFX_XXX 则指定将列名对应到域成员变量内。

2．CAssessDBSampleView 类

CAssessDBSampleView 类继承自 CRecordView 类，这是一个对话框类，对应的对话框 ID 在类中由 IDD 指定：

```
class CChap13_ODBC_DatabaseView : public CRecordView
{
protected: // create from serialization only
    CChap13_ODBC_DatabaseView();
    DECLARE_DYNCREATE(CChap13_ODBC_DatabaseView)
public:
    enum{ IDD = IDD_CHAP13_ODBC_DATABASE_FORM };
    CChap13_ODBC_DatabaseSet* m_pSet;
// Attributes
public:
    CChap13_ODBC_DatabaseDoc* GetDocument() const;
// Operations
public:
// Overrides
public:
    virtual CRecordset* OnGetRecordset();
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
        // DDX/DDV support
    virtual void DoDataExchange(CDataExchange* pDX);
    virtual void OnInitialUpdate(); // called first time after construct
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
// Implementation
public:
    virtual ~CChap13_ODBC_DatabaseView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
protected:
// Generated message map functions
protected:
    afx_msg void OnFilePrintPreview();
    afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnContextMenu(CWnd* pWnd, CPoint point);
    DECLARE_MESSAGE_MAP()
};
#ifndef _DEBUG  // debug version in Chap13_ODBC_DatabaseView.cpp
```

```
    inline CChap13_ODBC_DatabaseDoc* CChap13_ODBC_DatabaseView::GetDocument() const
    { return reinterpret_cast<CChap13_ODBC_DatabaseDoc*>(m_pDocument); }
#endif
```

CAssessDBSampleView 类中含有一个成员 m_pSet，其中存放关联的要显示的 CRecordset 对象的指针。另一个比较重要的函数是 DoDataExchange()，这个函数负责在 CRecordset 对象和控件之间传输数据，后面具体讨论这个函数的实现。

这个类的构造函数如下：

```
CChap13_ODBC_DatabaseView::CChap13_ODBC_DatabaseView()
    : CRecordView(CChap13_ODBC_DatabaseView::IDD)
{
    m_pSet = NULL;
    // TODO: add construction code here
}
```

成员 m_pSet 在构造函数内被设置为 NULL。实际上，在调用构造函数之后将立即自动调用 OnInitialUpdate()函数，在这里 m_pSet 才被设置成指向结果集的指针。

```
void CChap13_ODBC_DatabaseView::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_Chap13_ODBC_DatabaseSet;
    CRecordView::OnInitialUpdate();
}
```

在设置好 m_pSet 的值后，调用 CRecordView::OnInitialUpdate()函数，这个函数将负责重组 CRecordset 对象的 SQL 语句，调用 Open()函数来查询数据库并获得结果。

13.3.4 在视图上显示数据库查询结果

为了将查询结果在视图上显示出来，需要首先向与视图相关联的对话框（IDD_ACCESSDB SAMPLE_FORM）中添加控件，然后通过 DoDataExchange()函数来将结果集中的数据与控件关联。

首先向对话框中添加控件。打开 Resource View，如果没有则选择“View”→“Resource View ”命令，或者按【Ctrl+Shift+E】组合键将其打开，在 Dialog 文件夹下找到 IDD 为 IDD_ACCESSDB SAMPLE_FORM 的对话框，双击进入对话框编辑器，这时显示为空白。选择“View”→“Toolbox”命令，或按【Ctrl+Shift+X】组合键打开工具箱，其中列出了可以加入其中对话框中的所有控件，暂时只需要编辑控件来显示域值，静态文本控件来显示列名。

在工具箱中选中要用到的控件后，将其放到适当的位置，右击控件，在弹出的快捷菜单中选择“Properties”命令（属性），打开属性面板。

在 Caption 中输入对应列名，如图 13.16 所示，并设置其他的属性。对编辑控件，因为暂时只是查询显示数据库的内容，不需要输入任何内容，所以还要将 Read Only 属性设定为 True。另外为了便于将后面的数据与控件关联，强烈建议将控件 ID 设置为有意义的宏，比如对应显示 ProductID 域的值，则将 ID 设置为 IDC_EDIT_PRODUCTID。

实际上这些控件 ID 都在 resource.h 文件中定义：

```
#define IDC_EDIT_PRODUCTID              1001
#define IDC_EDIT_PRODUCTNAME            1002
#define IDC_EDIT_QUERYPERUNIT           1003
#define IDC_EDIT_SUPPLIERID             1004
#define IDC_EDIT_CATEGORYID             1005
```

```
#define ID_EDIT_UNITPRICE              1006
#define IDC_EDIT_UNITSINSTOCK          1007
#define ID_EDIT_UNITPRICE2             1008
```

这是由向导自动完成的，用户只需设置相应的属性值即可。添加控件后的结果如图 13.17 所示。

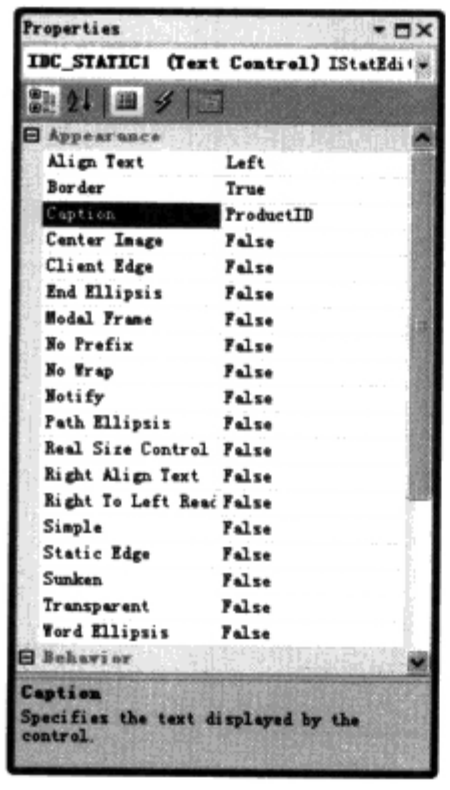


图 13.16　设置控件属性

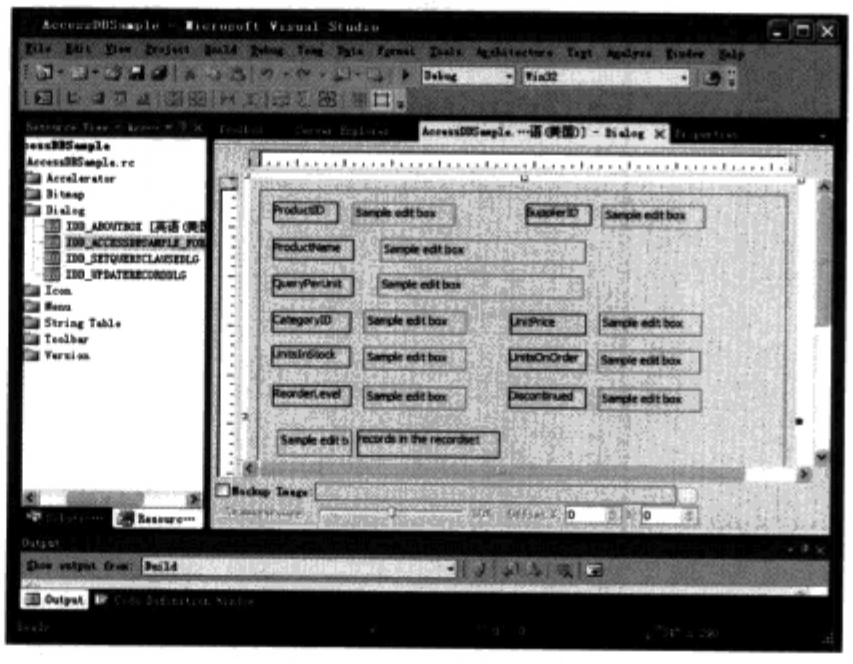


图 13.17　程序设计界面

除了显示所有域的值外，这里还在最后一行设置一个 Editr 控件来显示查询到的记录个数，右击控件，在菜单中选择“Add Variable”命令，如图 13.18 所示。

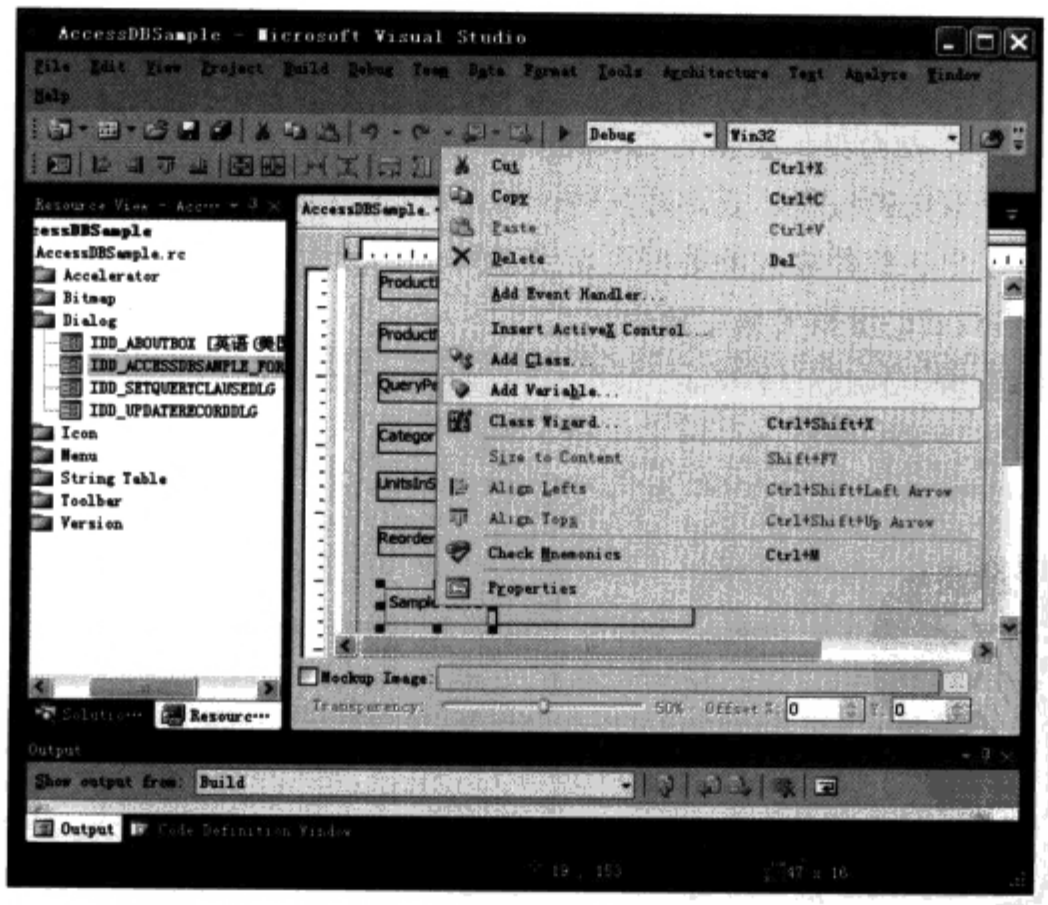


图 13.18　为控件添加关联变量

在添加成员变量向导对话框中输入如下内容：先在“Category”下拉列表框中选择“Value”选项，再在“Variable type”下拉列表框中选择“int”类型，最后在“Variable name”文本框中输入成员变量名 m_iRecordCout，表示查询结果内的记录个数，如图 13.19 所示。这时可以看到在 CAccessDBSample

View 类中加了一个 int 型成员变量，并在 DoDataExchange()函数中自动加入了控件数据交换：

```
DDX_Text(pDX, IDC_EDIT_RECORDCOUNT, m_iRecordCount);
```

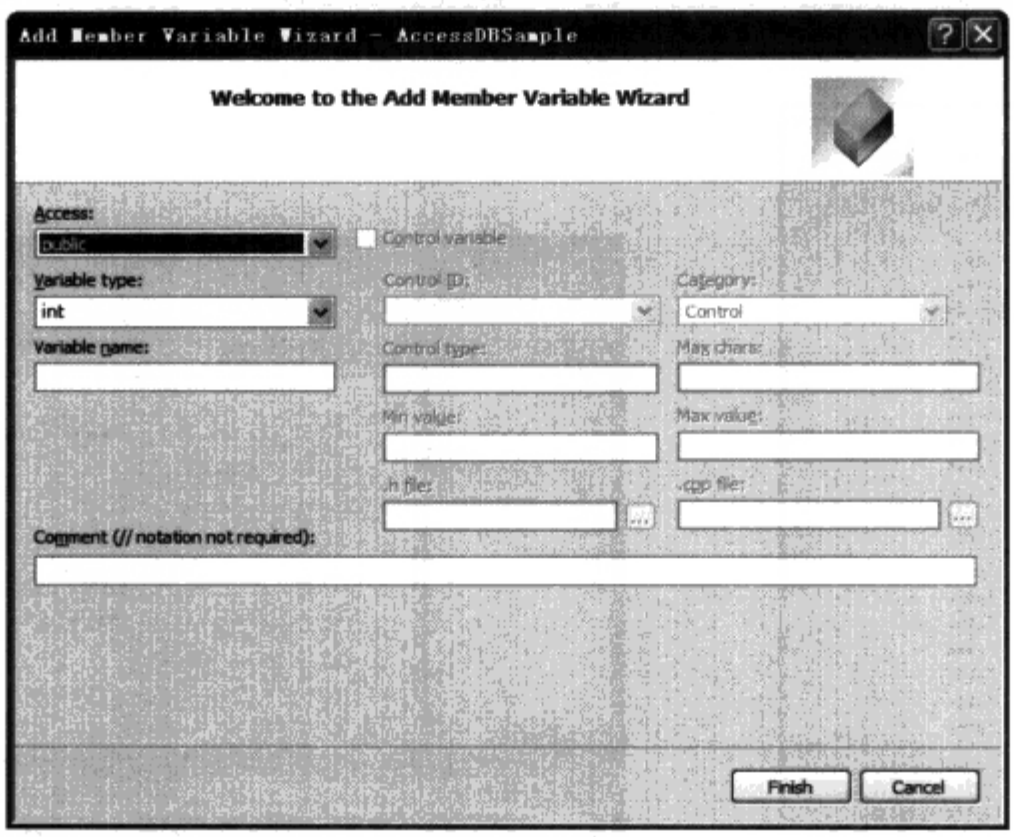


图 13.19　添加成员变量向导

通过 CAccessDBSampleView::DoDataExchange()函数可以将 m_pSet 内的域成员变量和控件关联起来，并通过控件将值显示出来，这通过调用 DDX_FieldText()函数实现，这个函数需要传入 4 个参数：

- 第一个参数是一个指向 CDataExchange 对象的指针，这里直接使用 DoDataExchange()传入的参数；
- 第二个参数是对应控件的 ID；
- 第三个参数是一个指向 CRecordset 对象内的域成员变量的引用；
- 第四个参数是一个指向提供数据的 CRecordset 对象的指针。

实际上，这是一个双向的过程，查询数据时，将数据从 CRecordset 内传给控件显示，而在更新数据时，仍可以这样将用户从控件输入的数据更新到 CRecordset 内，进而更新数据库。

下面是 DDX_FieldText()函数的典型调用形式：

```
DDX_FieldText(pDX, IDC_MYEDITBOX, m_pSet->m_szColumn1, m_pSet);
```

完整的 DoDataExchange()函数实现如下，这个函数需要开发人员手工输入：

```
void CAccessDBSampleView::DoDataExchange(CDataExchange* pDX)
{
    CRecordView::DoDataExchange(pDX);
    // you can insert DDX_Field* functions here to 'connect' your controls to the database fields, ex.
    // DDX_FieldText(pDX, IDC_MYEDITBOX, m_pSet->m_szColumn1, m_pSet);
    // DDX_FieldCheck(pDX, IDC_MYCHECKBOX, m_pSet->m_bColumn2, m_pSet);
    // See MSDN and ODBC samples for more information
    DDX_FieldText(pDX,IDC_EDIT_PRODUCTID,m_pSet->m_ProductID,m_pSet);
    DDX_FieldText(pDX,IDC_EDIT_PRODUCTNAME,m_pSet->m_ProductName,m_pSet);
    DDX_FieldText(pDX,IDC_EDIT_SUPPLIERID,m_pSet->m_SupplierID,m_pSet);
    DDX_FieldText(pDX,IDC_EDIT_CATEGORYID,
        m_pSet->m_CategoryID,m_pSet);
    DDX_FieldText(pDX,IDC_EDIT_QUERYPERUNIT,
        m_pSet->m_QuantityPerUnit,m_pSet);
```

```
    DDX_FieldText(pDX,ID_EDIT_UNITPRICE,m_pSet->m_UnitPrice,m_pSet);
    DDX_FieldText(pDX,IDC_EDIT_UNITSINSTOCK,
        m_pSet->m_UnitsInStock,m_pSet);
    DDX_FieldText(pDX,IDC_EDIT_REORDERLEVEL,
        m_pSet->m_ReorderLevel,m_pSet);
    DDX_FieldText(pDX,ID_EDIT_DISCONTINUED,
        m_pSet->m_Discontinued,m_pSet);
    DDX_FieldText(pDX,ID_EDIT_UNITSONORDER,
        m_pSet->m_UnitsOnOrder,m_pSet);
    //m_iRecordCount = m_pSet->GetRecordCount();
    //pDX->m_bSaveAndValidate = TRUE;
    DDX_Text(pDX, IDC_EDIT_RECORDCOUNT, m_iRecordCount);
}
```

下面讲解如何获得查询结果的记录个数，实际上 CRecordset 类提供了一个成员函数：

```
long GetRecordCount( ) const;
```

该函数返回当前包含的记录个数，但是为了提高效率，这个返回值实际上是当前访问到的最高位置的记录的序号，就是说即使查询完毕，如果不访问任何记录，这个函数返回的结果也将是 0，而不是真正的总的记录个数。必须往后遍历访问每个记录，才能获得总的记录个数。

另外，为了将这个值赋给成员变量 m_iRecordCout，可以在 CAccessDBSampleView：OnInitialUpdate() 内实现，在调用母类的 OnInitialUpdate()后，已经进行了查询。通过逐个往后遍历，到达最后，就可以获得记录个数：

```
void CAccessDBSampleView::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_AccessDBSampleSet;
    CRecordView::OnInitialUpdate();
    if(m_pSet->IsOpen()){
                                    //逐个往下遍历，直到最后
        while( !m_pSet->IsEOF() ){
            m_pSet->MoveNext();
        }
        m_iRecordCount = m_pSet->GetRecordCount();
                                    //将当前记录移回第一个
        this->OnMove(ID_RECORD_FIRST);
                                    //设置窗口的标题
        CString strCaption = _T("Table Name:");
        CString strTable = m_pSet->GetTableName();
        if(!strTable.IsEmpty()){
            strCaption += strTable;
        }
        GetDocument()->SetTitle(strCaption);
    }
}
```

这里补充 CRecordset 类提供的几个成员函数的说明。

- IsOpen() ： 查询 CRecordset 对象是否已经调用 Open()函数对数据库进行了查询，如果是返回 True，否则返回 False。
- IsEOF()：查看当前记录指针是否已经到达末尾。
- MoveNext():将当前记录指针往下移一个位置。

最后调用 CRecordView 的 OnMove()函数将记录指针重新指向第一个记录。这个函数的原型如下：

```
virtual BOOL OnMove(
    UINT nIDMoveCommand
);
```

参数 nIDMoveCommand 表示移动的方向，可以取 4 个不同值。

- ID_RECORD_FIRST：移到记录集的第一个记录位置。
- ID_RECORD_LAST：移到最后一个记录。
- ID_RECORD_NEXT：移到下一个记录。
- ID_RECORD_PREV：移到上一个记录。

最后的几行代码是设置窗口标题，通过调用 CRecordset::GetTableName()函数返回访问的表的名称，通过 GetDocument()->SetTitle()函数设置窗口标题。

程序的运行结果如图 13.20 所示。

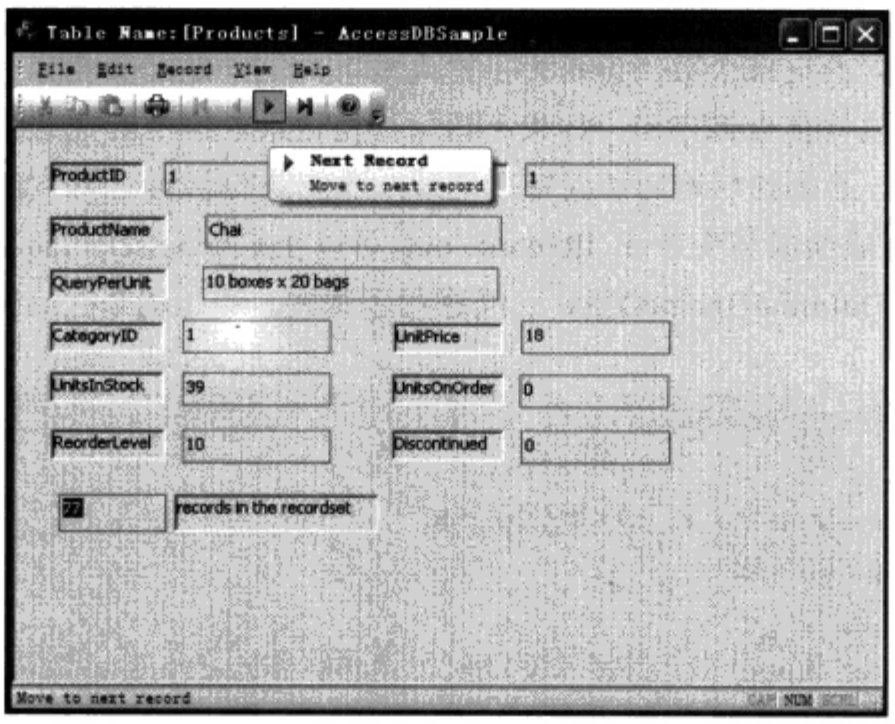


图 13.20　程序运行界面

可以看到，程序自动生成了移到下一记录、移到上一记录、移到最后一个记录、移到第一个记录的工具栏按钮，单击这些工具栏按钮就可以移动查看不同的记录，最后一行的 Edit 控件正确显示出包含的记录个数。

13.3.5　对查询结果排序及设置查询条件

除了查询整张表的所有内容外，还可以通过 WHERE 从句设置查询条件，通过 ORDER BY 从句对查询结果进行排序，这通过修改 CRecordset::GetDefaultSQL()函数实现，如改为下面代码：

```
CString CAccessDBSampleSet::GetDefaultSQL()
{
    //return _T("[Products]");
    return _T("SELECT * FROM [Products] WHERE [UnitPrice] > 50 \
        ORDER BY [UnitPrice];");
}
```

这里表示只选择那些 UnitPrice 值大于 50 的记录并对记录按 UnitPrice 值进行升序排列。

则运行的结果如图 13.21 所示。

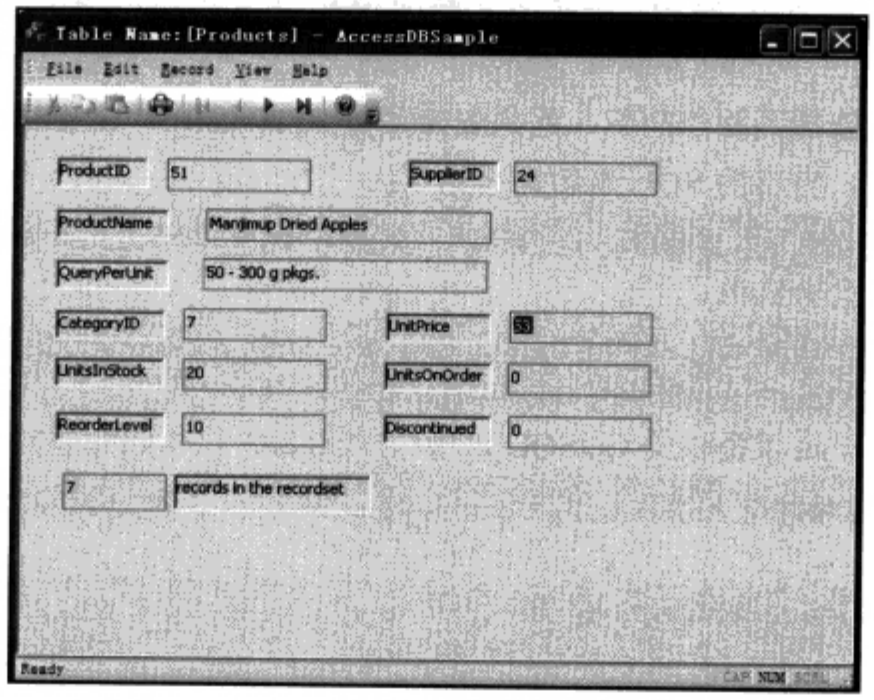


图 13.21　设置查询条件后的运行结果

还有一种方法来设置 WHERE 从句和 ORDER BY 从句，就是直接对 CRecordset 的 m_strFilter 成员及 m_strSort 成员赋值，这里的赋值必须在 CRecordset 调用 Open()函数查询数据库之前在 CRecordView 的 OnInitialUpdate()函数中进行，为了实现与上面相同的功能，只需添加如下代码：

```
void CAccessDBSampleView::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_AccessDBSampleSet;
                                                    //必须在 CRecordView::OnInitialUpdate()函数调用之前赋值
    m_pSet->m_strSort = _T("[UnitPrice]");
    m_pSet->m_strFilter = _T("[UnitPrice] > 50");
    CRecordView::OnInitialUpdate();
    if(m_pSet->IsOpen()){
                                                    //逐个往下遍历，直到最后
        while( !m_pSet->IsEOF() ){
            m_pSet->MoveNext();
        }
        m_iRecordCount = m_pSet->GetRecordCount();
                                                    //将当前记录移回第一个
        this->OnMove(ID_RECORD_FIRST);
                                                    //设置窗口的标题
        CString strCaption = _T("Table Name:");
        CString strTable = m_pSet->GetTableName();
        if(!strTable.IsEmpty()){
            strCaption += strTable;
        }
        GetDocument()->SetTitle(strCaption);
    }
}
```

注意，不要忘记将 GetDefaultSQL()函数改回原来的只返回表名的格式，否则运行时会出错，提示为不合法的 SQL 语句：

```
return _T("[Products]");
```

这是因为在执行查询之前，向导会根据 GetDefaultSQL()返回的字符串及 m_strFilter 成员和

m_strSort 成员来构造完整的 SQL 语句，当提供完整的 SQL 语句再对这两个成员设置值时，就会连接成非法的 SQL 语句。

而对这两个成员的赋值规则相当简单，就是去掉关键字(WHERE，ORDER BY)后的子句，允许按多个条件及对多个域排序。完整的 SQL 语句构造如下：

```
SELECT * FROM [GetDefaultSQL()返回的表名] WHERE [m_strFilter] ORDER BY [m_strSort]
```

下面的赋值语句是完全合法的：

```
m_pSet->m_strFilter = _T("[UnitPrice] > 20 AND [UnitPrice] < 80 OR ProductID > 70 ");
m_pSet->m_strSort = _T("[UnitPrice] DESC,[ProductID] DESC");
```

可以通过调用 CRecordset 类提供的 GetSQL()函数来查看最终执行的完整的 SQL 语句，函数原型是：

```
const CString& GetSQL( ) const;
```

修改代码如下：

```
void CAccessDBSampleView::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_AccessDBSampleSet;
                                                    //必须在 CRecordView::OnInitialUpdate()函数调用之前赋值
    //m_pSet->m_strSort = _T("[UnitPrice]");
    //m_pSet->m_strFilter = _T("[UnitPrice] > 50");
    m_pSet->m_strFilter = _T("[UnitPrice] > 20 AND [UnitPrice] < 80 OR ProductID > 70 ");
    m_pSet->m_strSort = _T("[UnitPrice] DESC,[ProductID] DESC");
    CRecordView::OnInitialUpdate();
    if(m_pSet->IsOpen()){
        CString strRecordsetSQL = m_pSet->GetSQL();
        AfxMessageBox(strRecordsetSQL);
                                                    //逐个往下遍历，直到最后
        while( !m_pSet->IsEOF() ){
            m_pSet->MoveNext();
        }
        m_iRecordCount = m_pSet->GetRecordCount();
                                                    //将当前记录移回第一个
        this->OnMove(ID_RECORD_FIRST);
        //设置窗口的标题
        CString strCaption = _T("Table Name:");
        CString strTable = m_pSet->GetTableName();
        if(!strTable.IsEmpty()){
            strCaption += strTable;
        }
        GetDocument()->SetTitle(strCaption);
    }
}
```

通过一个消息对话框来显示最终的 SQL 语句，结果如图 13.22 所示。

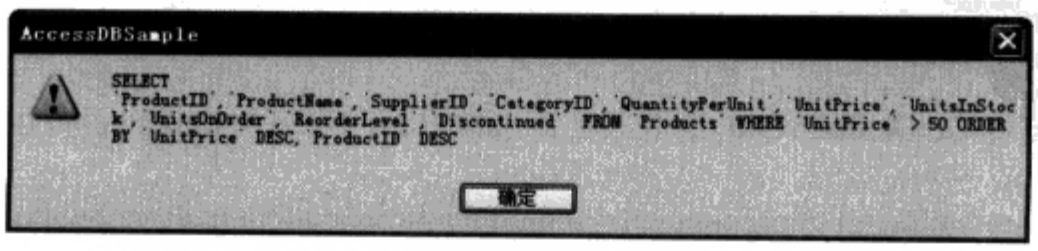


图 13.22　程序输出的最终 SQL 语句

与前面分析稍有区别的是，这里返回的 SQL 语句将所有列名一一列举出来，而不是使用“SELECT *”，而后面的 WHERE 从句和 ORDER BY 从句则与前面分析完全相同。

13.3.6　动态设置查询条件并更新查询结果

接下来，希望自己能够动态设置查询的 WHERE 从句和 ORDER BY 从句，然后重新查询数据库，这可以通过 CRecordset 的 Requery()函数实现，函数原型是：

```
virtual BOOL Requery( );
```

这个函数的功能是根据 GetDefaultSQL()返回的串及成员 m_strFilter 和成员 m_strSort 重新组织 SQL 语句，并重新查询以获得新的查询结果。如果查询成功，返回 True，否则返回 False。

设置两个 Edit 控件来显示当前的 WHERE 从句和 ORDER BY 从句，并设置一个按钮用来要求重新设置查询条件，并执行 Requery 语句。如图 13.23 所示。

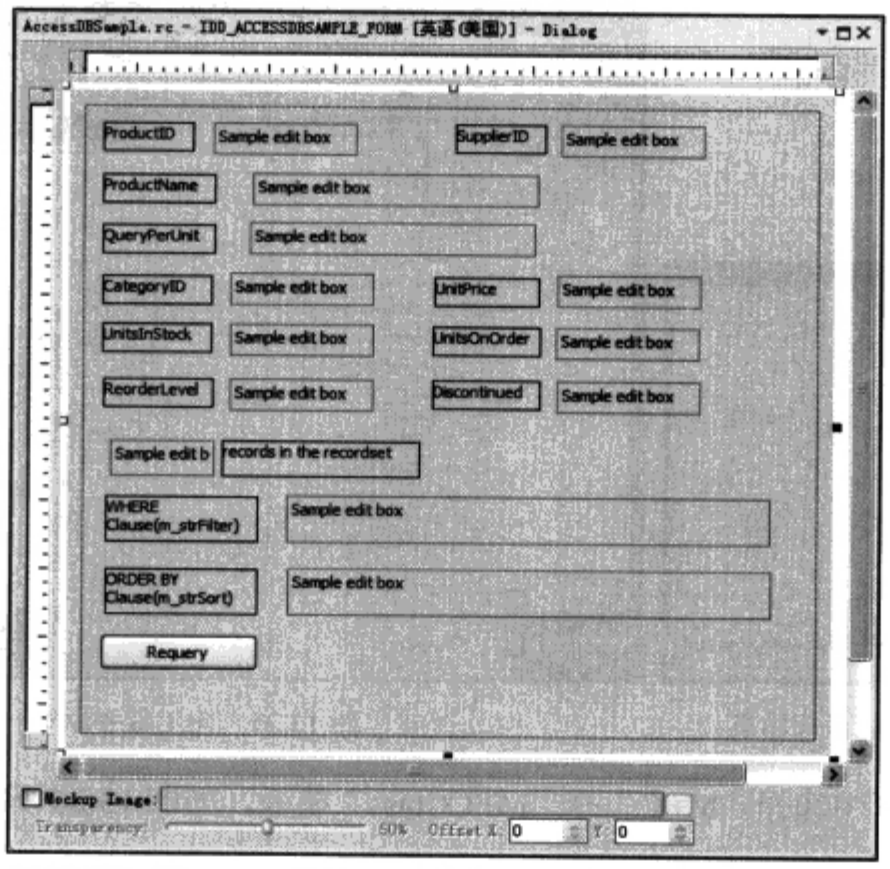


图 13.23　添加查询条件相关控件后的界面

图 13.23 中下面两个 Edit 控件分别显示当前的 m_strFilter 及 m_strSort 的值，也就是对应的 WHERE 从句和 ORDER BY 从句。通过手工将这两个变量的信息关联到控件上显示出来，首先在 CAccessDBSampleView 类内定义两个 public 成员，如下：

```
CStringW m_strSetFilter;
CStringW m_strSetOrder;
```

在 CAccessDBSampleView 构造函数中将这两个成员初始化为空串，这里使用 CStringW 类型是为了适应程序可能使用 Unicode 字符串。

下面通过 DoDataExchange()函数来将成员传递给控件显示：

```
DDX_Text(pDX, IDC_EDIT_SETFILTER, m_strSetFilter);
DDX_Text(pDX, IDC_EDIT_SETSORT, m_strSetOrder);
```

这里使用的函数是 DDX_Text()，这个函数用于在对话框控件和视图的成员变量之间传输数据，允许传输的数据类型有 int、UINT、long、DWORD、Cstring、float 及 double。函数原型是：

```
void AFXAPI DDX_Text(
    CDataExchange* pDX,
```

```
        int nIDC,
        T& value
    );
```

T 代表上面列出的某个数据类型。

与上面显示记录个数的方式相似，在 CAccessDBSampleView 的 OnInitialUpdate()函数内取得查询从句的值：

```
    m_strSetFilter = m_pSet->m_strFilter;
    m_strSetOrder = m_pSet->m_strSort;
```

先不对按钮（Button）控件做任何处理，运行程序，可以看到当前的查询条件显示如图 13.24 所示。

下面增加允许用户动态输入查询子句的功能，这需要增加一个对话框类。选择“View”→“Class View”命令，进入类视图，右击“AccessDBSample”选项，选择“Add”→“Class”命令，如图 13.25 所示

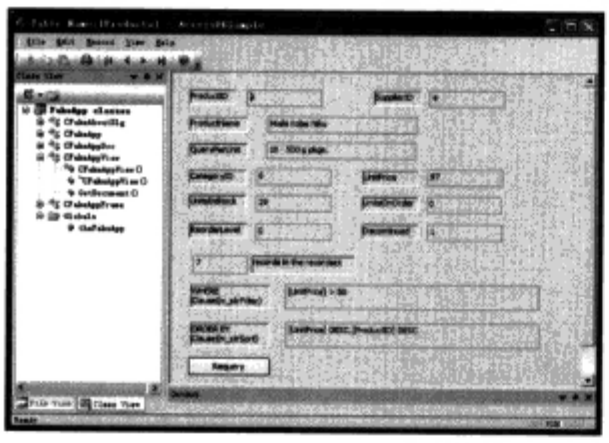


图 13.24　显示查询条件的运行结果

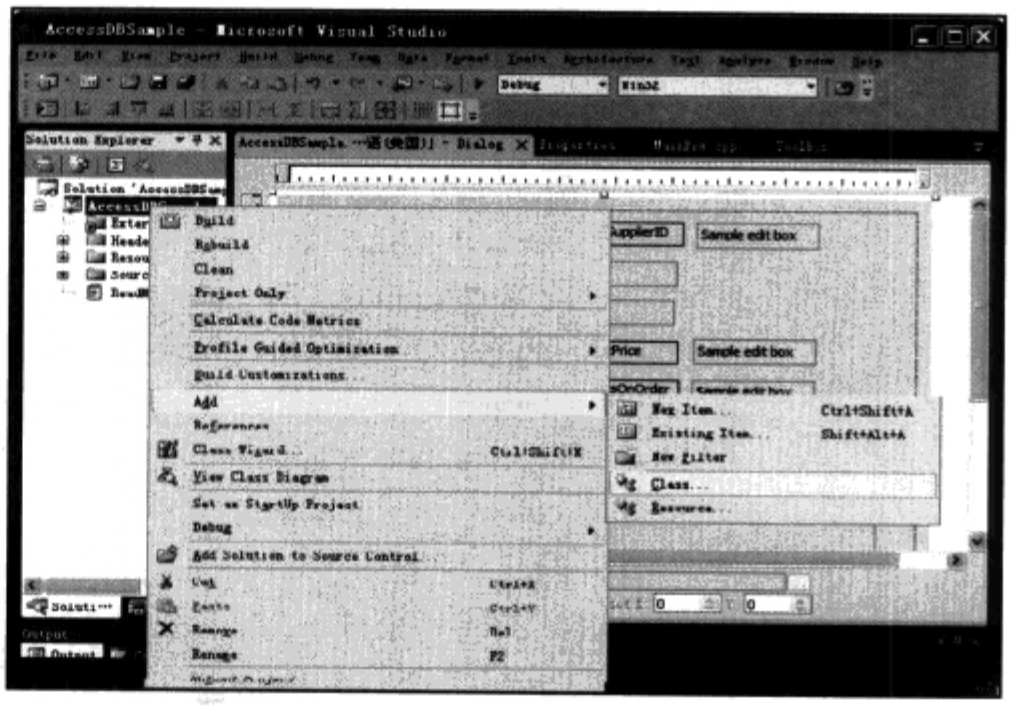


图 13.25　向工程中添加一个新类

在添加类向导对话框中选择“MFC”→“MFCClass”模板，如图 13.26 所示。

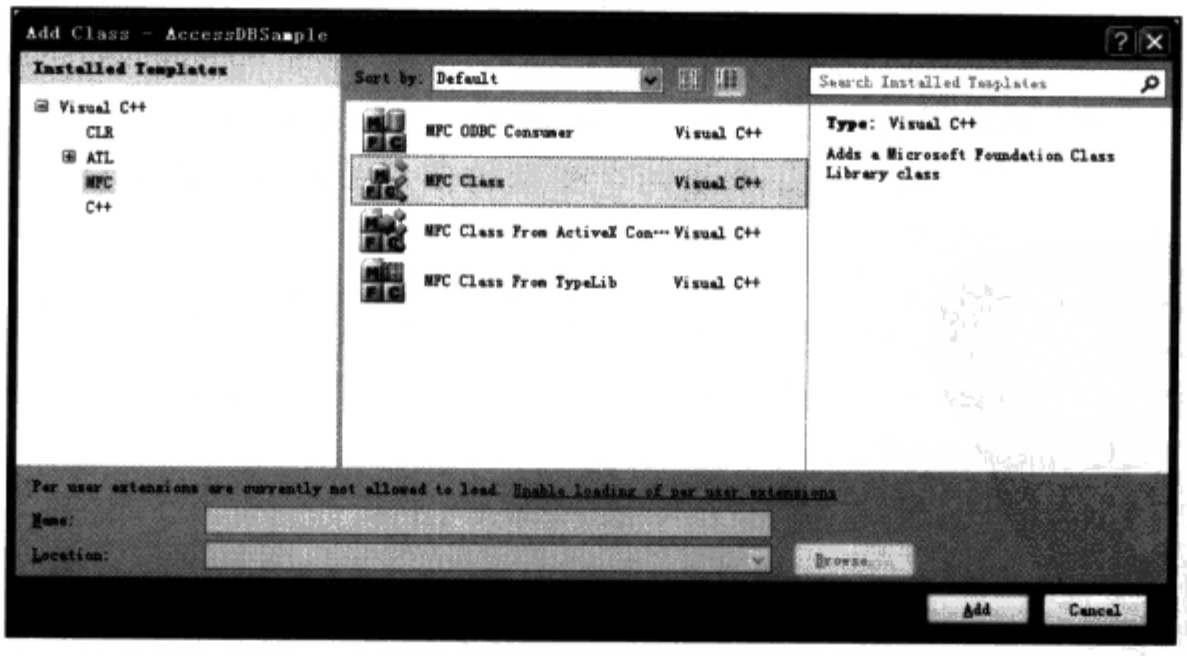


图 13.26　添加新类向导

单击“Add”按钮，输入类名，并设置“Base class”为“Cdialog”，这样向导会自动根据类名创建一个对话框并设置对话框的 ID，如图 13.27 所示。

打开 Resource View，在“Dialog”文件夹下可以看到新加入一个 IDD=IDD_SETQUERYCLAUSEDLG 的对话框资源，打开对话框编辑器，加入如图 13.28 所示的控件。

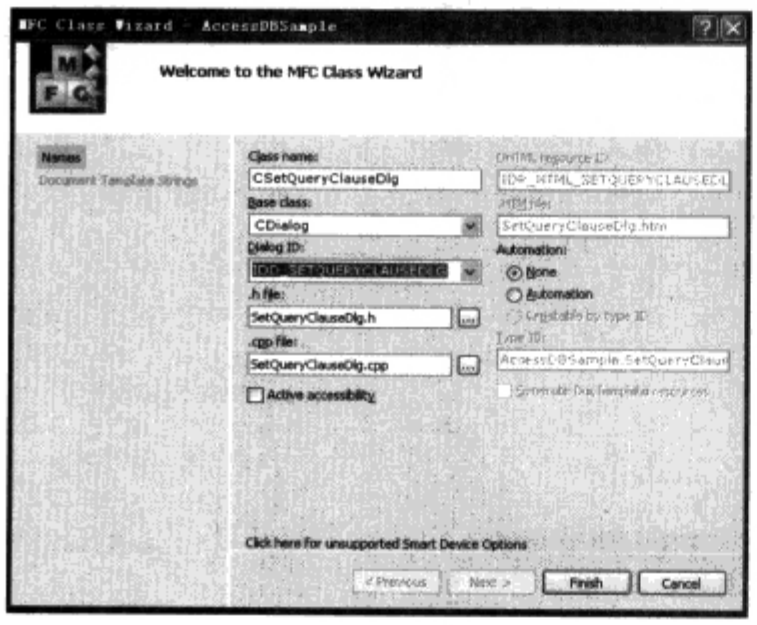


图 13.27　添加 MFC 类向导

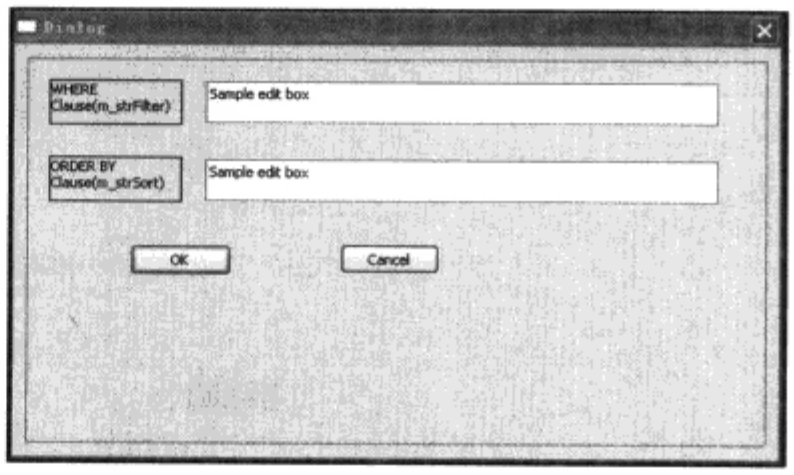


图 13.28　对话框界面

设置好控件属性后，在 CSetQueryClauseDlg 类中加入两个成员变量来与控件获得的数据进行传输：

```
// CSetQueryClauseDlg dialog
class CSetQueryClauseDlg : public CDialog
{
    DECLARE_DYNAMIC(CSetQueryClauseDlg)
public:
    CSetQueryClauseDlg(CWnd* pParent = NULL);          // standard constructor
    virtual ~CSetQueryClauseDlg();
// Dialog Data
    enum { IDD = IDD_SETQUERYCLAUSEDLG };
protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    DECLARE_MESSAGE_MAP()
public:
    CStringW m_strSetFilterDlg;
    CStringW m_strSetOrderDlg;
};
```

在构造函数中初始化变量为空串，并通过 DoDataExchange()函数实现数据传输：

```
void CSetQueryClauseDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Text(pDX,IDC_EDIT_SETFILTERDLG,m_strSetFilterDlg);
    DDX_Text(pDX,IDC_EDIT_SETSORTDLG,m_strSetOrderDlg);
}
```

完成这些步骤之后，就可以对按钮控件加入事件处理函数了。在 Resource View 下，打开 IDD_ACCESSDBSAMPLE_FORM 对话框，右击按钮，在弹出的快捷菜单中选择“Add Event Handler”命令，如图 13.29 所示。

在下面向导中列出按钮控件的所有可能事件及负责捕捉事件的类，默认值为单击事件及由视图类来处理事件，保持这个默认设置，单击“Add and Edit”按钮，进入事件处理函数。

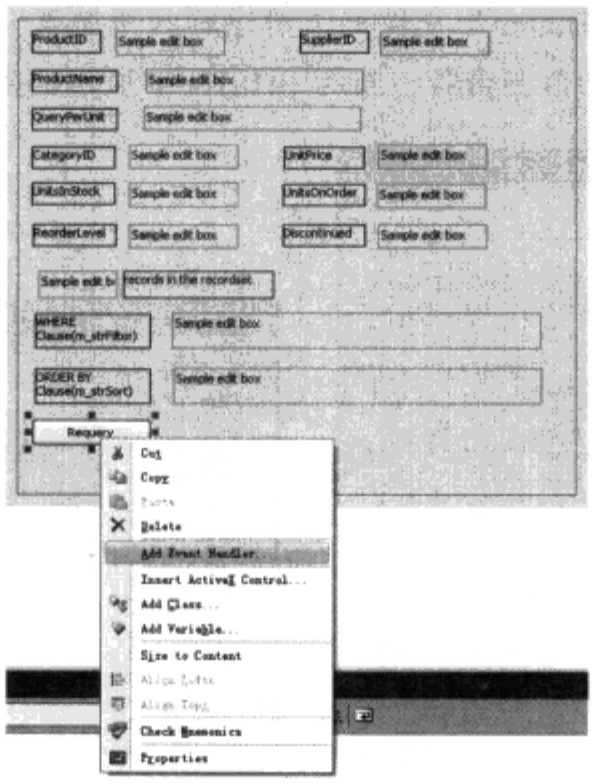


图 13.29　加入事件处理函数

事件处理函数的主要功能就是调用对话框获取用户输入的新的查询子句。如果子句发生变化，则调用 m_pSet 的 Requery()函数重新查询，并重新获得记录的个数和显示新的查询子句。在运行之前，还要先将新加入的对话框类的头文件包含进来，在 AccessDBSampleView.cpp 文件内加入下面一句：

```
#include "SetQueryClauseDlg.h"
```

Requery 按钮的单击事件处理函数如下：

```
void CAccessDBSampleView::OnBnClickedButtonRequery()
{
    // TODO: Add your control notification handler code here
    CSetQueryClauseDlg setQueryDlg;
                        //显示原来的查询子句
    setQueryDlg.m_strSetFilterDlg = m_pSet->m_strFilter;
    setQueryDlg.m_strSetOrderDlg = m_pSet->m_strSort;
                        //弹出对话框，输入新的查询子句
    if(setQueryDlg.DoModal() == IDOK ){
                        //如果输入了新的子句，则调用 Requery()函数重新查询，并重新设置记录个数和子句内容
        bool bSameQuery = true;
        AfxMessageBox(setQueryDlg.m_strSetFilterDlg);
        if(setQueryDlg.m_strSetFilterDlg.Compare(m_pSet->m_strFilter) != 0){
            m_pSet->m_strFilter = setQueryDlg.m_strSetFilterDlg;
            bSameQuery = false;
        }
        if(setQueryDlg.m_strSetOrderDlg.Compare(m_pSet->m_strSort) != 0){
            m_pSet->m_strSort = setQueryDlg.m_strSetOrderDlg;
            bSameQuery = false;
        }
        if(!bSameQuery){
            m_pSet->Requery();
            if(m_pSet->IsOpen()){
                CString strRecordsetSQL = m_pSet->GetSQL();
                AfxMessageBox(strRecordsetSQL);
```

```
                    //逐个往下遍历，直到最后
                    while( !m_pSet->IsEOF() ){
                        m_pSet->MoveNext();
                    }
                    m_iRecordCount = m_pSet->GetRecordCount();
                    m_strSetFilter = m_pSet->m_strFilter;
                    m_strSetOrder = m_pSet->m_strSort;
                    //将当前记录移至第一个
                    this->OnMove(ID_RECORD_FIRST);
                }
            }
        }
    }
```

编译并运行程序，结果如图 13.30 所示。

单击“Requery”按钮就会弹出设置查询条件对话框，输入新的查询子句，单击“OK”按钮，如图 13.31 所示。

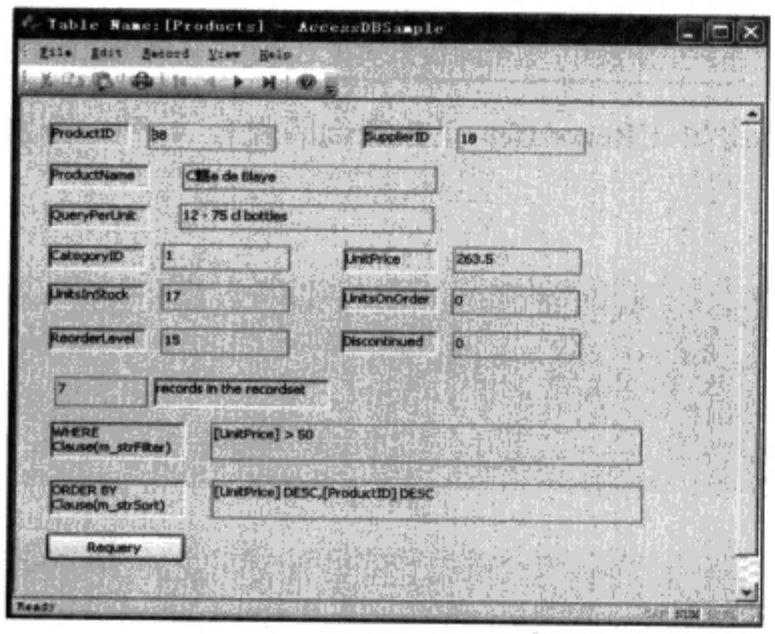


图 13.30　程序运行结果

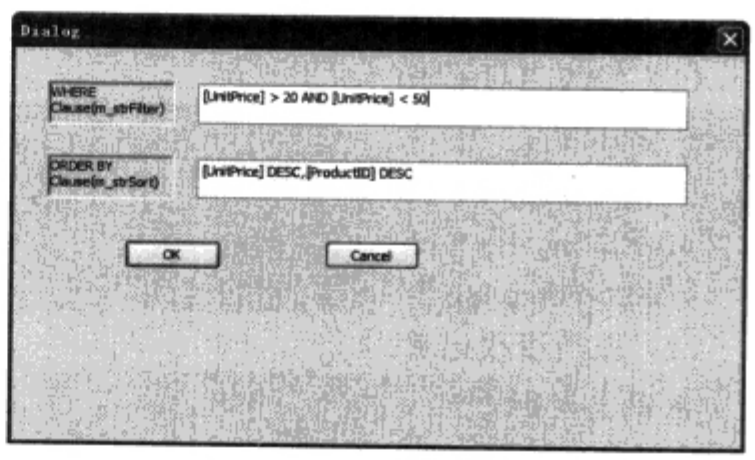


图 13.31　设置查询条件

可以看到程序获得了新的查询结果，运行结果如图 13.32 所示。

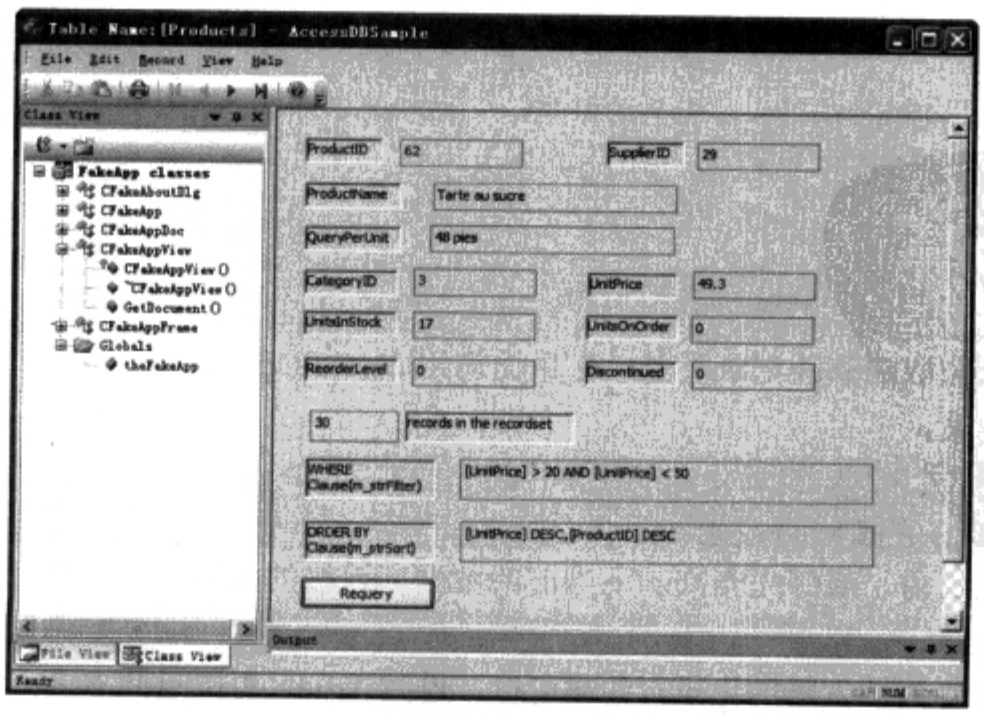


图 13.32　设置新的查询条件后的运行结果

13.4 ODBC 更新数据库

前面介绍了数据库查询操作的相关知识，下面介绍数据库的更新操作，主要是使用 CRecordset 类的几个函数来实现更新，如表 13.5 所示。

表 13.5　　CRecordset 类提供的实现更新操作的成员函数

更 新 操 作	说　　明
virtual void AddNew();	向数据库新插入一条记录，用法是调用这个函数，然后设置 CRecordset 类内的各个域成员变量的值，调用 Update()函数才将记录插入数据库，需要调用 Requery()来查看新的结果
virtual void Edit();	修改当前记录的属性值，先调用这个函数，再将要改变的域成员值设置为新值，调用 Update()函数向数据库更新记录
virtual void Delete();	删除当前访问到的记录，同时将当前的域成员变量置为 NULL，在删除当前记录后，要手动向前或向后移动记录。注意，这个函数后不需要再调用 Update()函数
virtual BOOL Update();	将 AddNew()和 Edit()对记录的更新操作向数据库提交，实现数据库更新
void CancelUpdate();	取消 AddNew()或 Edit()函数对数据库的更新操作，这个函数必须在 Update()被调用之前进行

下面举个简单例子来说明这些函数的基本用法，比如 CRecordset 正在查询的表有 4 列，分别为 Name、Sex、Age、Height。

（1）修改当前记录：

```
m_pSet->Edit();
m_pSet->m_Sex = "Male";
m_pSet->m_Age = 30;
m_pSet->Update();
```

（2）取消修改记录：

```
m_pSet->Edit();
m_pSet->m_Sex = "Male";
m_pSet->m_Age = 30;
m_pSet->CancelUpdate();
```

（3）插入新记录：

```
m_pSet->AddNew();
m_pSet->m_Name="ZhangSan";
m_pSet->m_Sex = "Male";
m_pSet->m_Age = 28;
m_pSet->Height = 1.75;
m_pSet->Update();
```

取消插入新记录与取消修改记录类似，在后面调用 CancelUpdate()即可。

（4）删除当前记录：

```
m_pSet->Delete();
m_pSet->MoveNext();//移到下一个记录
```

在调用这些函数之前，最好先查询一下当前的数据库连接的权限，盲目执行操作会抛出异常。

```
BOOL CanAppend( ) const;
```

查询能否向数据库内插入新的记录，如果可以，返回 True，否则返回 False。调用 AddNew()之前，进行此查询操作。

```
BOOL CanUpdate( ) const;
```

查询是否能更新数据库的记录。如果在连接时，数据库属性被指定为 CRecordset::readOnly，则只能查询，不能执行任何更新操作，并返回 False，否则返回 True。

下面介绍一个通过 ODBC 更新数据库的例子。

下面通过一个简单的例子来说明这些更新操作的执行。由于 Nwind 数据库内各个表之间有关联，更新数据库受到诸多限制，所以决定自己创建一个数据库，这是包含 NBA 火箭队的主要球员的一张表 Rockets。表的属性有 PlayerID、Name、Number、Age、State、Height、Weight。首先，向 ODBC 数据源注册这个数据库；接着，利用 MFC 向导选择这个数据源，并使用 Rockets 表，如图 13.33 所示，其他设置与前面相同。

与前面相似，设置对应的对话框，布局如图 13.34 所示。

设置好各个控件的 ID 等属性，在 CUpdateAccessDBRocketsView::DoDataExchange()函数中添加如下代码，实现 m_pSet 内的域成员变量与控件的关联。代码中的控件 ID 可以根据自己的设置进行相应修改：

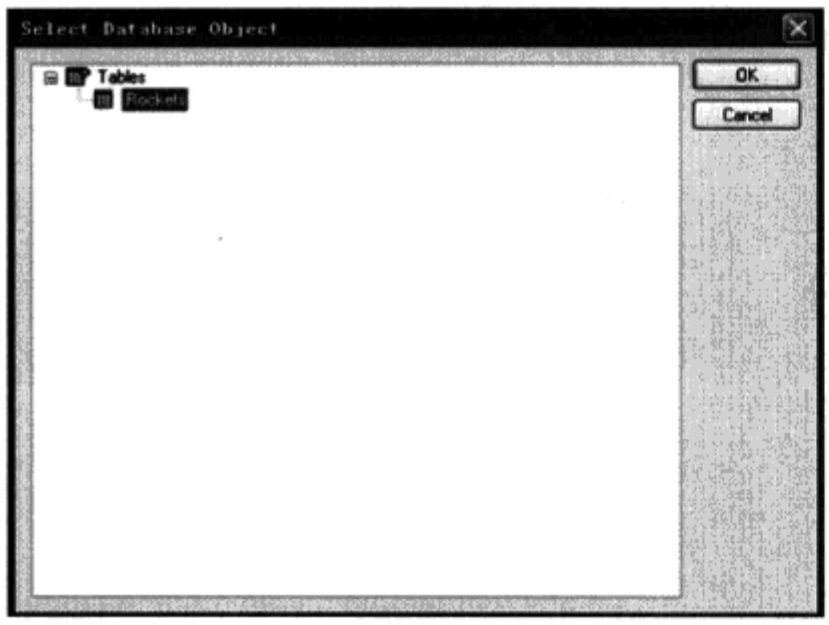


图 13.33　选择数据库对象

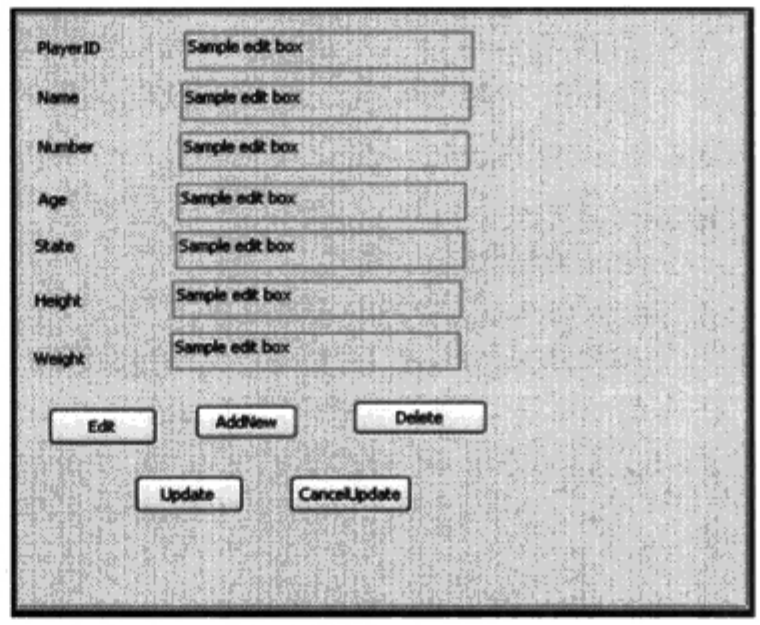


图 13.34　对话框设计界面

```
void CUpdateAccessDBRocketsView::DoDataExchange(CDataExchange* pDX)
{
    CRecordView::DoDataExchange(pDX);
    // you can insert DDX_Field* functions here to 'connect' your controls to
    //the database fields, ex.
    // DDX_FieldText(pDX, IDC_MYEDITBOX, m_pSet->m_szColumn1, m_pSet);
    // DDX_FieldCheck(pDX, IDC_MYCHECKBOX, m_pSet->m_bColumn2,
        //m_pSet);
    // See MSDN and ODBC samples for more information
    DDX_FieldText(pDX,IDC_EDIT_PLAYERID,m_pSet->m_PlayerID,m_pSet);
    DDX_FieldText(pDX,IDC_EDIT_NAME,m_pSet->m_Name,m_pSet);
    DDX_FieldText(pDX,IDC_EDIT_NUMBER,m_pSet->m_Number,m_pSet);
    DDX_FieldText(pDX,IDC_EDIT_AGE,m_pSet->m_Age,m_pSet);
    DDX_FieldText(pDX,IDC_EDIT_STATE,m_pSet->m_State,m_pSet);
    DDX_FieldText(pDX,IDC_EDIT_HEIGHT,m_pSet->m_Height,m_pSet);
    DDX_FieldText(pDX,IDC_EDIT_WEIGHT,m_pSet->m_Weight,m_pSet);
}
```

关于记录的更新，实际上只要将对应的 Edit 控件的 Read Only 属性设置为 False，在运行时直接往

控件内输入新的值，再移动记录位置就可以实现，这是因为 DDX_FieldText()函数的数据传输是双向的，在修改 m_pSet 中域成员变量的值后，移动记录位置，程序会自动把新的值更新到数据库，以后查询时就是这个新值。

但是这样直接修改记录很不安全，如果控件对用户一直可写，很容易输入一些错误的更新值。可以加入更多的控制，查看记录时控件是只读的；当移动到指定的想更新的记录时，再设定控件可写；更新完成后，再回到只读。设定这几个按钮的目的就是模拟 CRecordset 提供的那几个更新函数的功能。

通过 CUpdateAccessDBRocketsView 类的：

```
virtual CWnd* GetDlgItem(
    int nID
) const;
```

获得控件窗口句柄，再设置窗口的属性。

要设置 Edit 控件的只读属性，需要先将返回值转换为 CEdit*类型，再调用 SetReadOnly()函数实现，如下：

```
((CEdit*)(this->GetDlgItem(IDC_EDIT_NAME)))->SetReadOnly(TRUE);
```

参数 True 设定控件为只读，而 False 则设定控件为可写。

先介绍 Edit 功能的实现，这里涉及记录集的两种模式，一种是只读模式，另一种是更新模式。在 CUpdateAccessDBRocketsView 类内声明一个枚举类型：

```
protected:
    //当前对记录的访问模式
    enum DBMode{QUERY,UPDATE};
    DBMode m_Mode;
```

在构造函数中，将其初始化为 QUERY 模式（即查询只读模式）：

```
CUpdateAccessDBRocketsView::CUpdateAccessDBRocketsView()
    : CRecordView(CUpdateAccessDBRocketsView::IDD)
{
    m_pSet = NULL;
    // TODO: add construction code here
    m_Mode = QUERY;
    m_iRecordsetCount = 0;
}
```

由于只有在调用 Edit()函数之后才能调用 Update()或 CancelUpdate()函数，所以程序运行时，设置“Update”及“CancelUpdate”按钮为不可用。当用户单击“Edit”按钮，程序进入 UPDATE 模式，将设置控件为可写，同时“Update”及“CancelUpdate”按钮可用，而“Edit”按钮不可用，等待用户输入新值；输入完毕后，单击“Update”按钮向数据库更新，单击“CancelUpdate”按钮取消更新，程序又回到只读模式，这可以通过 CWnd 的 EnableWindow()函数实现。首先，初始化设置控件属性，这在 CUpdateAccessDBRocketsView:: OnInitialUpdate()函数内实现：

```
void CUpdateAccessDBRocketsView::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_UpdateAccessDBRocketsSet;
    CRecordView::OnInitialUpdate();
    m_iRecordsetCount = 0;
    if( m_pSet->IsOpen() ){
                        //获取记录集内所含的记录个数
        while( !m_pSet->IsEOF() ){
            m_pSet->MoveNext();
```

```
            m_iRecordsetCount++;
        }
        m_pSet->MoveFirst();
                    //设置窗口标题
        CStringW title = _T("Table Name");
        CStringW tablename = m_pSet->GetTableName();
        if( !tablename.IsEmpty() ){
            title += _T(":") + tablename;
        }
        GetDocument()->SetTitle(title);
                    //设置按钮的可用属性，“Edit”可用，而“Update”及“CancelUpdate”不可用
        this->GetDlgItem(IDC_BUTTON_EDIT)->EnableWindow(TRUE);
        this->GetDlgItem(IDC_BUTTON_ADDNEW)->EnableWindow(TRUE);
        this->GetDlgItem(IDC_BUTTON_DELETE)->EnableWindow(TRUE);
        this->GetDlgItem(IDC_BUTTON_UPDATE)->EnableWindow(False);
        this->GetDlgItem(
            IDC_BUTTON_CANCELUPDATE)->EnableWindow(False);
    }
}
```

右击“Edit”按钮，添加“Edit”按钮的单击事件处理函数：

```
void CUpdateAccessDBRocketsView::OnBnClickedButtonEdit()
{
    // TODO: Add your control notification handler code here
    //保存原记录的各个域的值
    m_pSet->m_temp_Age = m_pSet->m_Age;
    m_pSet->m_temp_Height = m_pSet->m_Height;
    m_pSet->m_temp_Name = m_pSet->m_Name;
    m_pSet->m_temp_Number = m_pSet->m_Number;
    m_pSet->m_temp_PlayerID = m_pSet->m_PlayerID;
    m_pSet->m_temp_State = m_pSet->m_State;
    m_pSet->m_temp_Weight = m_pSet->m_Weight;
    //设置对应域成员的 Edit 控件为可写
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_NAME)))->SetReadOnly(False);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_NUMBER)))->SetReadOnly(False);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_AGE)))->SetReadOnly(False);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_STATE)))->SetReadOnly(False);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_HEIGHT)))->SetReadOnly(False);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_WEIGHT)))->SetReadOnly(False);
    //设置按钮属性，此时进入 UPDATE 模式，等待用户输入并确定是否进行 Update
    this->GetDlgItem(IDC_BUTTON_EDIT)->EnableWindow(False);
    this->GetDlgItem(IDC_BUTTON_ADDNEW)->EnableWindow(False);
    this->GetDlgItem(IDC_BUTTON_DELETE)->EnableWindow(False);
    this->GetDlgItem(IDC_BUTTON_UPDATE)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_CANCELUPDATE)->EnableWindow(TRUE);
    m_Mode = UPDATE;
    m_pSet->Edit();
}
```

这个函数的主要功能是在设置完控件属性后，进入 UPDATE 模式，并调用 m_pSet->Edit()函数，准备对记录值进行更新。这里需要注意的是，在输入新值的时候不希望用户移动记录位置，这可以通过禁止 Record 菜单下的向前移动、向后移动、移到第一个记录及移到最后一个记录来实现。

在 Recourse View 下的 Menu 文件夹内，找到 Record 菜单下的 4 个子菜单 ，右击子菜单，选择“Add Event Handler”→“UPDATE_COMMAND_UI”命令，如图 13.35 所示，在“Classlist”列表框中选择“CUpdateAccessDBRocketsView”选项，单击“Add And Edit”按钮进入事件处理函数。

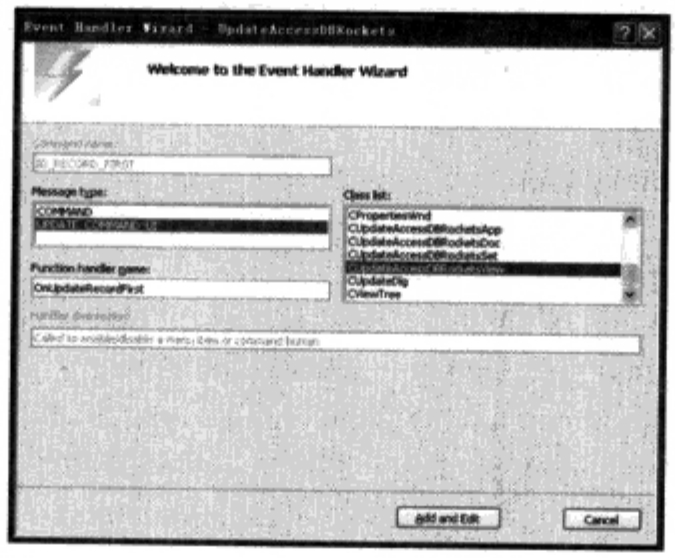


图 13.35　添加事件处理向导

编辑 4 个更新菜单的处理函数如下：

```
void CUpdateAccessDBRocketsView::OnUpdateRecordFirst(CCmdUI *pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI->Enable( m_Mode==QUERY && !this->IsOnFirstRecord());
}
void CUpdateAccessDBRocketsView::OnUpdateRecordPrev(CCmdUI *pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI->Enable( m_Mode==QUERY && !this->IsOnFirstRecord());
}
void CUpdateAccessDBRocketsView::OnUpdateRecordNext(CCmdUI *pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI->Enable( m_Mode==QUERY && !this->IsOnLastRecord());
}
void CUpdateAccessDBRocketsView::OnUpdateRecordLast(CCmdUI *pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI->Enable( m_Mode==QUERY && !this->IsOnLastRecord());
}
```

pCmdUI->Enable()函数设置菜单命令是否可用，仅当在查询模式下，并且对于“移到第一个记录”及“向前移动一个记录”命令，只有当前不是第一个记录时才可用；而对“移到最后一个记录”及“向后移动一个记录”命令，只有当前不是最后一个记录时才可用。

明白这些之后，就可以编写“Update”及“CancelUpdate”按钮的处理函数了。右击“Update”按钮，加入单击事件处理函数，编写代码：

```
void CUpdateAccessDBRocketsView::OnBnClickedButtonUpdate()
{
    // TODO: Add your control notification handler code here
    //调用 m_pSet->Update()函数向数据库更新记录
```

```
    m_pSet->Update();
    //更新之后，回到查询模式，设置控件为只读
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_NAME)))->SetReadOnly(TRUE);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_NUMBER)))->SetReadOnly(TRUE);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_AGE)))->SetReadOnly(TRUE);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_STATE)))->SetReadOnly(TRUE);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_HEIGHT)))->SetReadOnly(TRUE);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_WEIGHT)))->SetReadOnly(TRUE);

    this->GetDlgItem(IDC_BUTTON_EDIT)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_ADDNEW)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_DELETE)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_UPDATE)->EnableWindow(False);
    this->GetDlgItem(
        IDC_BUTTON_CANCELUPDATE)->EnableWindow(False);
    m_Mode = QUERY;
}
```

如果不想更新，简单地调用 m_pSet->CancelUpdate()函数只是暂时不把当前的新值写回数据库，但是此时由于已经把控件写入了新的值，前面介绍过，接着移动记录将把控件上的新值写回数据库。为了避免这种情况，需要在对控件输入新值之前将原记录的值保存下来，如果后面不想更新，就将记录原值恢复给控件即可，这需要在 CUpdateAccessDBRocketsSet 类内加入几个成员，用于在更新数据前保存原值：

```
long m_temp_PlayerID;
CStringW    m_temp_Name;    //The players' name
int   m_temp_Number;
int   m_temp_Age;
CStringW    m_temp_State;
CstringW    m_temp_Height;
CStringW    m_temp_Weight;
```

在构造函数内，模仿域成员变量对其初始化。

在“CancelUpdate”按钮的事件处理函数中，需要加入恢复记录原值的操作：

```
void CUpdateAccessDBRocketsView::OnBnClickedButtonCancelupdate()
{
    // TODO: Add your control notification handler code here
    //撤销更新操作
    m_pSet->CancelUpdate();
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_NAME)))->SetReadOnly(TRUE);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_NUMBER)))->SetReadOnly(TRUE);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_AGE)))->SetReadOnly(TRUE);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_STATE)))->SetReadOnly(TRUE);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_HEIGHT)))->SetReadOnly(TRUE);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_WEIGHT)))->SetReadOnly(TRUE);
    //恢复记录原值
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_NAME)))->SetWindowTextW(m_pSet->m_temp_Name);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_WEIGHT)))->SetWindowTextW(m_pSet->m_temp_Weight);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_HEIGHT)))->SetWindowTextW(m_pSet->m_temp_Height);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_STATE)))->SetWindowTextW(m_pSet->m_temp_State);
```

```
        CStringW orinVal;
        orinVal.Format(_T("%d"),m_pSet->m_temp_Age);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_AGE)))->SetWindowTextW(orinVal);
        orinVal.Format(_T("%d"),m_pSet->m_temp_Number);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_NUMBER)))->SetWindowTextW(orinVal);
        this->GetDlgItem(IDC_BUTTON_EDIT)->EnableWindow(TRUE);
        this->GetDlgItem(IDC_BUTTON_ADDNEW)->EnableWindow(TRUE);
        this->GetDlgItem(IDC_BUTTON_DELETE)->EnableWindow(TRUE);
        this->GetDlgItem(IDC_BUTTON_UPDATE)->EnableWindow(False);
        this->GetDlgItem(
            IDC_BUTTON_CANCELUPDATE)->EnableWindow(False);
        m_Mode = QUERY;
    }
```

这里通过 CEdit 类的 SetWindowTextW()函数来设置控件的值，这个函数需要输入一个 StringW 类型的串作为控件的值，由于一些域成员变量不是 StringW 类型，所以需要先将其转换为 StringW 类型，再进行设置。

至此，就完成了更新及取消更新的操作。下面介绍 AddNew 操作，即向数据库中插入一条新的记录，这需要加入一个对话框来接收用户输入的新记录，通过类向导加入一个基类为 CDialog 的类，编辑对应的对话框如图 13.36 所示。

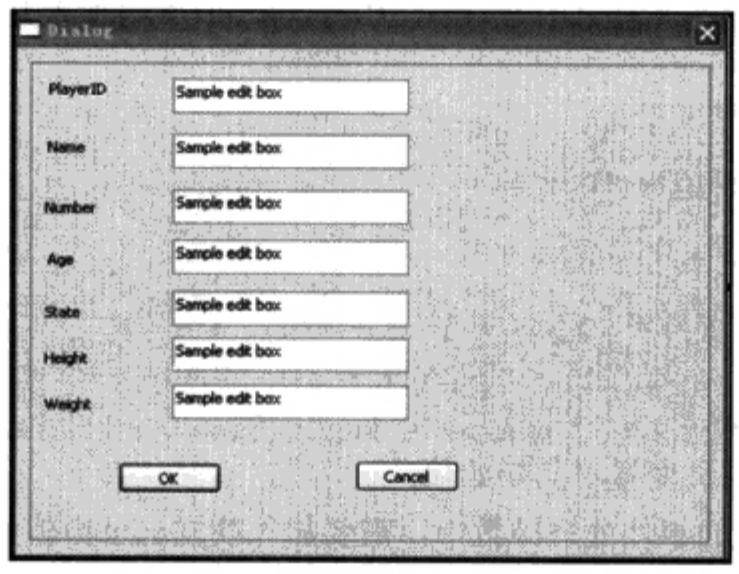


图 13.36　添加记录对话框

对话框中的每个 Edit 控件都是可写的。基本工作流程是，用户单击“AddNew”按钮，将弹出图 13.36 所示的对话框，等到用户输入完毕并单击“OK”按钮后，将向数据库内插入一个新记录，单击“Cancel”按钮则取消插入操作。

先在这个新的对话框类中加入成员来接收用户的输入：

```
// CUpdateDlg dialog
class CUpdateDlg : public CDialog
{
    DECLARE_DYNAMIC(CUpdateDlg)
public:
    CUpdateDlg(CWnd* pParent = NULL);                    // standard constructor
    virtual ~CUpdateDlg();
// Dialog Data
    enum { IDD = IDD_UPDATEDLG };
protected:
```

```
    virtual void DoDataExchange(CDataExchange* pDX);      // DDX/DDV support
    DECLARE_MESSAGE_MAP()
public:
    long m_temp_PlayerID;
    CStringW    m_temp_Name;                              //The players' name
    int   m_temp_Number;
    int   m_temp_Age;
    CStringW    m_temp_State;
    CStringW    m_temp_Height;
    CStringW    m_temp_Weight;
};
```

初始化成员变量与域成员变量的初始化一致。

通过 DoDataExchange()成员函数实现数据传输：

```
void CUpdateDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Text(pDX,IDC_EDIT_UPDATEDLG_PLAYERID,m_temp_PlayerID);
    DDX_Text(pDX,IDC_EDIT_UPDATEDLG_NAME,m_temp_Name);
    DDX_Text(pDX,IDC_EDIT_UPDATEDLG_NUMBER,m_temp_Number);
    DDX_Text(pDX,IDC_EDIT_UPDATEDLG_AGE,m_temp_Age);
    DDX_Text(pDX,IDC_EDIT_UPDATEDLG_STATE,m_temp_State);
    DDX_Text(pDX,IDC_EDIT_UPDATEDLG_HEIGHT,m_temp_Height);
    DDX_Text(pDX,IDC_EDIT_UPDATEDLG_WEIGHT,m_temp_Weight);
}
```

将这个新的对话框类的头文件包含到 UpdateAccessDBRocketsView.cpp 文件中：

```
#include "UpdateDlg.h"
```

“AddNew”按钮的事件处理函数如下：

```
void CUpdateAccessDBRocketsView::OnBnClickedButtonAddnew()
{
    CUpdateDlg updateDlg;
                                                  //如果用户单击“OK”按钮则将新记录插入数据库
    if( updateDlg.DoModal() == IDOK ){
        m_pSet->AddNew();
        m_pSet->m_Age = updateDlg.m_temp_Age;
        m_pSet->m_Height = updateDlg.m_temp_Height;
        m_pSet->m_Name = updateDlg.m_temp_Name;
        m_pSet->m_Number = updateDlg.m_temp_Number;
        m_pSet->m_PlayerID = updateDlg.m_temp_PlayerID;
        m_pSet->m_State = updateDlg.m_temp_State;
        m_pSet->m_Weight = updateDlg.m_temp_Weight;
        if( m_pSet->Update() ){
            m_pSet->Requery();
            if( !this->IsOnLastRecord() ){
                this->OnMove(ID_RECORD_LAST);
            }
        }
    }
}
```

为了查看新插入的记录，需要调用 Requery()函数；再移到最后一个记录的位置，是为了能直接看到新插入的记录。

“Delete”按钮的事件处理函数比较简单，首先弹出一个对话框要求用户确认是否真的要删除当前记录。Delete()函数只删除当前记录，不会自动移到新的记录，需要手工移动，通过调用CUpdateAccessDBRocketsView类的OnMove()函数实现移动到新的位置，代码如下：

```
void CUpdateAccessDBRocketsView::OnBnClickedButtonDelete()
{
    // TODO: Add your control notification handler code here
    if ( AfxMessageBox(_T("Confirm to delete this record"),MB_YESNO)
            == IDYES ){
        m_pSet->Delete();
        if( ! (this->IsOnLastRecord()) ){
            this->OnMove(ID_RECORD_NEXT);
        }
        else{
            this->OnMove(ID_RECORD_LAST );
        }
    }
}
```

移动记录的策略是，如果当前记录不是最后一个记录，那么向后移动一个记录；否则移到新的最后一个记录的位置（也就是原来的倒数第二个记录的位置）。

13.5 ODBC 访问 SQL Server

前面已经介绍过，ODBC是与数据库独立的，也就是说，只要存在对应的ODBC驱动，那么就可以通过统一的ODBC API来访问。但是，在系统注册SQL Server数据库的方式与前面有所不同，下面简单介绍一下。

为了在ODBC数据源中注册SQL Serser数据库，首先还是依次选择“控制面板”→“管理工具”→“数据源（ODBC）”命令，进入“ODBC数据源管理器”对话框，如图13.37所示。

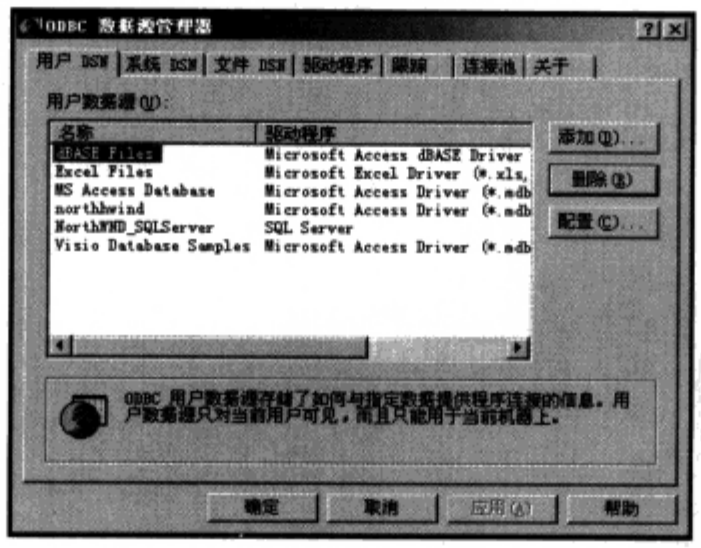


图 13.37 ODBC 数据源管理器

单击“添加”按钮，在弹出的“创建新数据源”对话框中，对于SQL Server 2000，选择“SQL Server”驱动程序；而对于SQL Server 2005，则选择“SQL Nqtive Client”驱动程序，这里选择“SQL Nqtive Client”驱动程序，如图13.38所示。

单击“完成”按钮后，就进入为SQL Server创建一个新的数据源的对话框，在“Name”文本框中输入数据源名称，在“Server”下拉列表框中输入“(local)\SQLEXPRESS”，而若选择“（local）”项

可能会导致连接失败；还可以在“Description”文本框中输入附加描述信息，如图 13.39 所示。

图 13.38　为 SQL Server2005 选择驱动程序

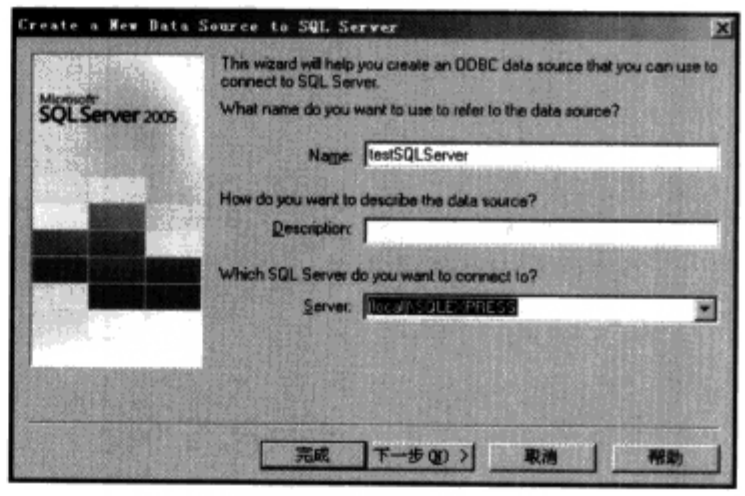


图 13.39　为 SQL Server 添加一个新的数据源

单击“下一步”按钮，在这里选择一个数据库，默认使用“master”数据库，如图 13.40 所示。如果想自己指定一个数据库，则应选中“Change the default database to”复选框，并在下面的下拉列表框中选择数据库，如图 13.41 所示。

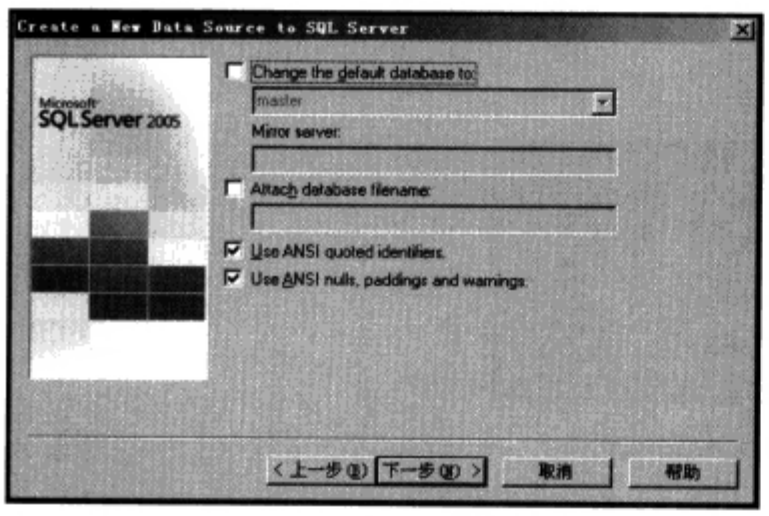


图 13.40　选择数据库

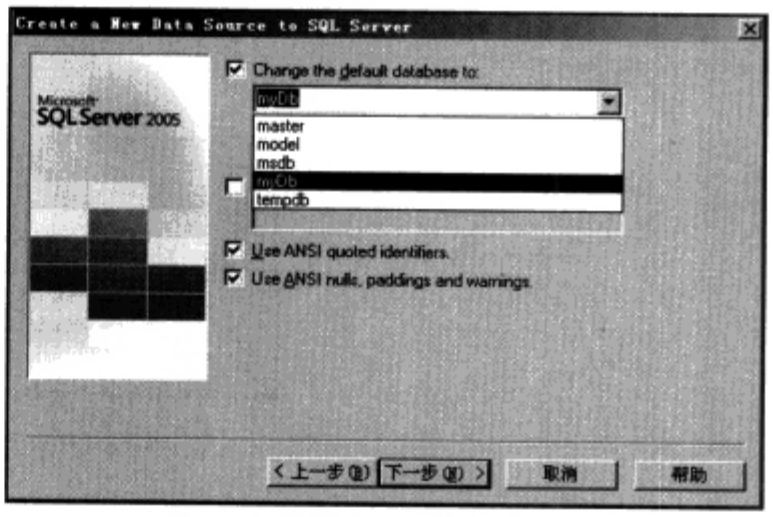


图 13.41　改变默认的数据库

单击“下一步”按钮，使用这些默认选项，单击“完成”按钮，如图 13.42 所示。

显示 SQL Server 数据源的安装信息，单击“Test Data Source”按钮测试数据源连接，如图 13.43 所示。

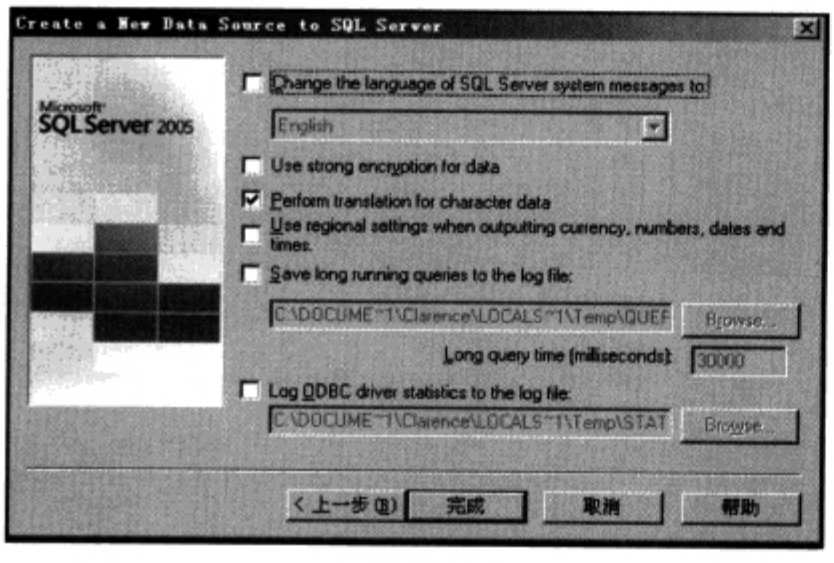


图 13.42　使用默认设置

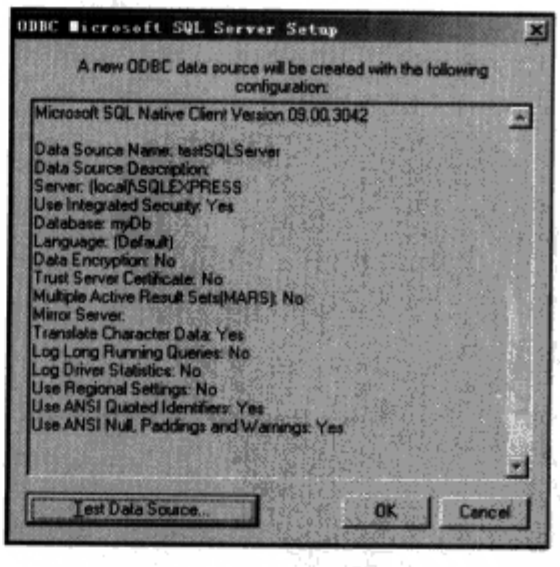


图 13.43　测试数据源连接

测试成功，说明数据源创建成功，单击“OK”按钮，如图 13.44 所示。

这样，就可以在“机器数据源”选项卡中看见自己新加入的 SQL Server 数据源了，如图 13.45 所示。

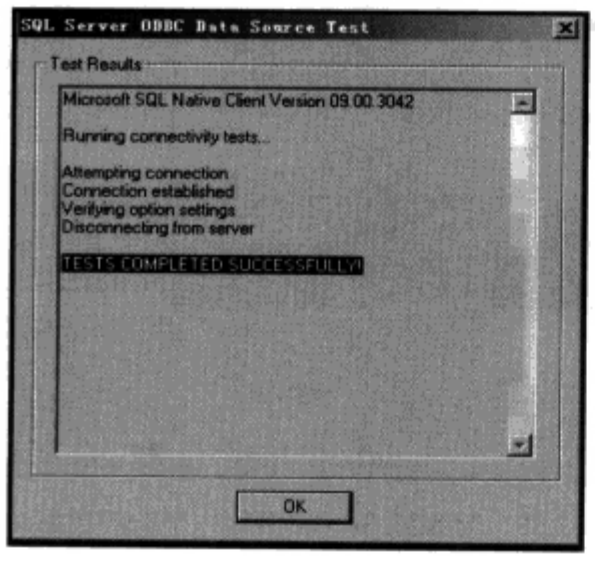


图 13.44　测试成功

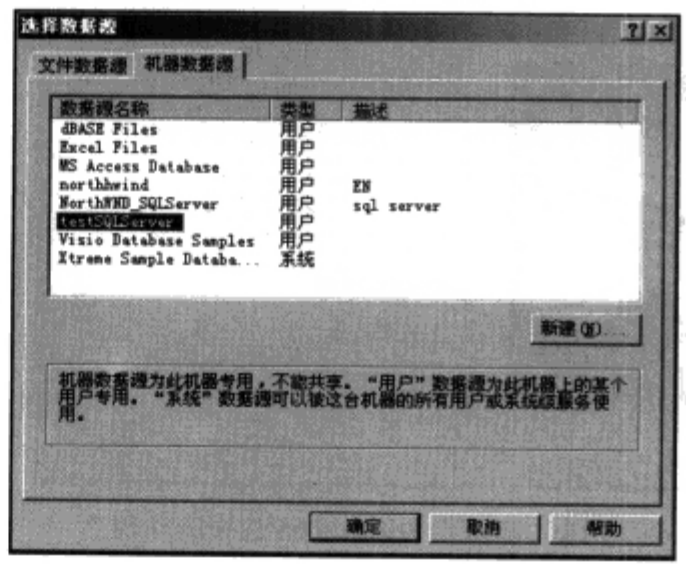


图 13.45　加入 SQL Server 数据源后的结果

下面介绍利用 MFC 向导通过 ODBC 访问 SQL Server 数据源的程序结构。创建程序的过程与前面介绍的基本相同，选择单文档类型，设置 ODBC 支持，在选择数据源后会弹出“SQL Server Login”对话框，直接按默认设置，使用可信任连接，单击“OK”按钮，如图 13.46 所示，在后面的对话框中选择对应的表即可。

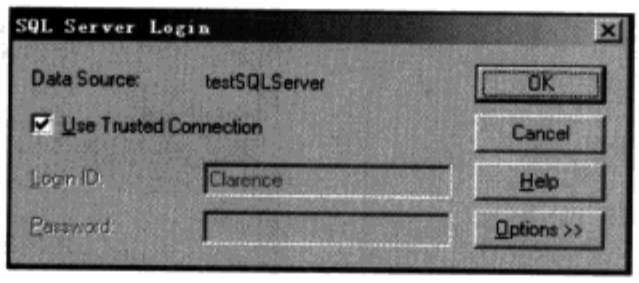


图 13.46　登录到 SQL Server

可以看到，生成的程序结构与前面例子基本一致，有一个继承自 CRecordset 类的记录集类存放查询结果，有一个继承自 CRecordView 类的视图类显示记录。这里需要说明的是，记录集类中的 GetDefaultConnect()函数返回的包含连接信息的字符串与前面不同：

```
//#error Security Issue: The connection string may contain a password
// The connection string below may contain plain text passwords and/or
// other sensitive information. Please remove the #error after reviewing
// the connection string for any security related issues. You may want to
// store the password in some other form or use a different user authentication.
CString CsqlserverdbSet::GetDefaultConnect()
{
    return _T(
        "DSN=testsqlserver; \
        Trusted_Connection=Yes;\
        APP=Microsoft\x00ae Visual Studio\x00ae 2010;
        WSID=ICT-D3E52427D67; \
        DATABASE=myDb"
    );
}
```

这些参数的具体含义如表 13.6 所示。

表 13.6　　连接字符串的参数

连 接 参 数	说　　明
DSN	数据源名
Trusted_Connection	是否为可信任的连接，在向导内选择使用可信任连接，所以这里值为 True
APP	数据源访问程序的类型，这里使用 Visual Studio 2010 程序
WSID	就是访问数据源的用户 ID
DATABASE	要访问的数据库名，这里使用自己创建的数据库，默认为访问 master 数据库

同样，这里需要将函数上方的#error 语句注释掉才能编译成功。其他的，如在记录集内移动、修改、添加、删除记录等的操作都与前面相同，这里不再赘述。

13.6　ADO 数据库访问

13.6.1　ADO 数据库访问概述

ADO（ActiveX Data Object，动态数据对象）提供了一种数据库编程的对象模型，它允许程序在 Visual C++、Visual Basic、VBScript、Java 等编程语言中访问、ADO 为程序员提供了访问，更新数据库的逻辑对象模型。

ADO 的主要对象在表 13.7 中列出。

表 13.7　　ADO 的主要对象

ADO 对象	说　　明
Connection 对象	代表一个与数据源的惟一会话。在基于客户端/服务器的数据库系统中，这可能等价于与服务器的一个网络连接。不同的数据提供者提供的该对象的集合、方法和属性不同
Command 对象	代表对数据源执行的特定的命令，用户通过这个对象可以查询数据库，并返回一个 Recordset 对象
Recordset 对象	代表数据库查询的结果，这是一个记录集合，通过这个对象，用户可以对数据源执行更新、插入、删除等操作，本章的主要操作集中在这个类
Record 对象	代表一个 Recordset 内的一个记录，一次只能访问一条记录，通过 Recordset 提供的方法可以向前或向后移动记录
Fields 集合	代表 Recordset 对象内的所有列的集合，通过列名可以访问到特定的列
Error 对象	代表 ADO 数据操作时发生的错误信息，不同的数据源提供不同的 Error 对象

下面主要介绍 Connection 对象及 Recordset 对象的一些常用方法。

1．Connection 对象

（1）Open()函数。

在 Visual C++中的 Open()函数原型是：

```
HRESULT Open( _bstr_t ConnectionString, _bstr_t UserID, _bstr_t Password, long Options);
```

参数 ConnectionString 代表连接信息的字符串，这里既可以通过 ODBC 提供的驱动连接，也可以直接连接 SQL Server。

参数 UserID 代表建立连接时使用的用户名的字符串。

参数 Password 代表连接时的密码的字符串。

参数 Options 决定 Open()函数是同步执行还是异步执行。这是一个 ConnectOptionEnum 类型的参数，有两个不同的值，即 adAsyncConnect 和 adConnectUnspecified，如表 13.8 所示。

表 13.8	ConnectOptionEnum 类型的参数
adAsyncConnect	以异步方式打开连接，函数不等连接建立完就返回，值为 16
adConnectUnspecified	以同步方式打开连接，函数必须在连接建立完后才能返回，值为-1，这是默认值

（2）Close()函数。

Visual C++内的 Close()函数原型是：

```
HRESULT Close();
```

关闭连接，在对数据库操作完毕之后执行这个操作。

另外，Connection 对象还提供了 Execute()方法直接执行 SQL 命令，BeginTrans()、CommitTrans()及 RollbackTrans()来开始一个新事务、提交事务及取消事务。

2．Recordset 对象

Recordset 对象保存数据库的查询结果，通过 Recordset 对象提供的成员函数，不仅可以向前或向后移动记录指针，访问不同的记录；而且还可以对数据库进行更新、插入、删除等操作。

（1）Open()函数。

在 Visual C++中的 Open()函数原型是：

```
HRESULTOpen(const_variant_t&Source,const_variant_t&ActiveConnection,CursorTypeEnumCursorType, LockTypeEnum LockType, long Options );
```

参数 Source 代表访问的数据源，可以使是一个合法的 Command 对象、一条完整的 SQL 查询语句、一个欲访问的表名、一个过程调用、一个 URL 或一个文件路径等。

参数 ActiveConnection 可以是一个已经打开的 Connection 对象；也可以是包含连接信息的字符串，这样由 Recordset 对象自己打开到数据库的连接。

参数 CursorType 指定 Recordset 对象访问数据源的光标类型，默认值为 adOpenForwardOnly，下面介绍。

参数 LockType 指定 Recordset 对象访问记录时的锁定类型，默认值为 adLockReadOnly，下面介绍。

参数 Options 指示数据源应该怎样解读 Sources 参数，这需要根据前面提供的不同的 Source 参数类型来决定。比如，如果 Source 为一个表名，则 Options 的值就应该对应为 adCmdTable。

ADO 定义了 4 种不同的光标类型来代表不同的访问数据库的方式，如表 13.9 所示。

表 13.9	ADO 内的光标类型
光 标 类 型	说　明
Dynamic cursor（动态光标）	在访问数据库期间，其他用户对数据库的添加、修改及删除操作都可见。允许在不依赖于书签的记录集中任意移动；如果数据源支持，允许使用书签
Keyset cursor（关键集光标）	类似于一个 Dynamic cursor，但是其他用户对数据库新插入的记录不可见，同时禁止访问已经被其他用户删除的记录。其他用户对数据库的修改仍可见。总是支持书签，允许在记录集内任意移动
Static cursor（静态光标）	提供一个对查询结果的静态备份，总是支持书签，支持在记录集内的任意移动。其他用户对数据库的修改、插入、删除操作都不可见
Forward-only cursor（只能向前光标）	只允许在记录集内向前移动，其他用户对数据库的修改、插入、删除操作都不可见

对应的枚举类型 CursorTypeEnum 分别代表这 4 种光标类型，如表 13.10 所示。

表 13.10	CursorTypeEnum 枚举类型的可取值
CursorTypeEnum 枚举值	说　明
adOpenDynamic	使用 Dynamic cursor 访问数据库，值为 2
adOpenKeyset	使用 Keyset cursor 访问数据库，值为 1

续表

CursorTypeEnum 枚举值	说　明
adOpenStatic	使用 Static cursor 访问数据库，值为 3
adOpenForwardOnly	使用 Forward-only cursor 访问数据库，值为 0。这是访问数据库的默认方式

ADO 定义了 4 种不同的锁定类型，用于指定访问记录的锁定方式，这通过枚举类型 LockType 定义，如表 13.11 所示。

表 13.11　　锁定类型

锁 定 类 型	说　明
adLockReadOnly	记录是只读的，不能更新数据，值为 1
adLockPessimistic	消极锁定，在用户修改数据之后马上锁定所有记录，以保证对记录的成功修改，值为 2
adLockOptimistic	积极锁定，只有在用户调用 Update()，确认更新数据库时，才锁定所有的记录，值为 3
adLockBatchOptimistic	积极批量更新，在批量更新模式下使用，值为 4

（2）Close()函数。

Visual C++中的 Close()函数原型是：

```
HRESULT Close();
```

该函数用来关闭记录集。

另外，Recordset 对象还提供了 MoveFirst()、MoveLast()、MoveNext()、MovePrevious()方法来移到第一个记录、移到最后一个记录、移动到前一个记录和移到后一个记录；Requery()方法用来重新查询数据库获得结果；Update()、AddNew()、Delete()、CancelUpdate()方法分别用来修改记录、插入一个新记录、删除记录及取消对记录的更新。这些方法基本上与 MFC 提供的 CRecordset 类内的方法相对应，而且调用方式一致。

这里需要注意的是，Recordset 对象对记录的更新操作与 CRecordset 类有几点不同。

- Recordset 对象没有 Edit()方法，直接对记录的域赋值然后调用 Update()函数即可。
- AddNew()方法提供了多种方式，既可以采用与 CRecordset 一致的方式；也可以传入两个数组作为参数，这两个数组分别存放列名和对应的新值。
- Recordset 对象的 Delete()方法的原型变为：

```
HRESULT Delete(AffectEnum AffectRecords)
```

这里需要一个 AffectEnum 类型的参数，代表这个删除操作所影响的列的范围，有 4 个不同值，如表 13.12 所示。

表 13.12　　删除类型

删 除 类 型	说　明
adAffectAll	如果 Recordset 对象查询时没有设置 Filter，就影响所有记录；否则只影响那些满足 Filter 查询条件的记录，值为 3
adAffectAllChapters	影响 Recordset 对象内的所有记录，包括那些因为设置了 Filter 而不可见的记录，值为 4
adAffectCurrent	只影响当前的记录，这是最常用到的参数，值为 1
adAffectGroup	只影响满足 Filter 条件的记录，但是必须将 Filter 设置成 FilterGroupEnum 内的值或书签数组，值为 2

最后，简要说明一下使用 ADO 访问数据的一些基本步骤：

（1）创建 Connection（连接）对象，设置连接信息，调用 Open()函数连接数据库；

（2）创建 Recordset（记录集）对象，调用 Open()函数取得查询结果；

（3）移动、查看各个记录，并对记录进行修改、插入、删除操作；

（4）调用 Recordset 对象的 Close()函数关闭记录集；

（5）调用 Connection 对象的 Close()函数关闭连接。

13.6.2 在 Visual C++中使用 ADO 编程

Visual C++提供了对 ADO 编程的支持，在每个 Windows 系统的系统盘下一般都有目录“Program Files\Common Files\System\ado”，在这个目录下会包含一个 msado**.dll 文件，根据不同的 Windows 版本，可以提供 msado15.dll、msado2.dll。利用#import 宏可以将这个动态库文件加载到工程中，这个动态库提供了对 ADO 对象的封装，主要的类有：

- _ConnectionPtr——指向某个 Connection 对象的指针;
- _RecordsetPtr——指向某个 Recordset 对象的指针;
- _CommandPtr——指向某个 Command 对象的指针。

这里需要说明的是，在使用#import 宏后，不能自己创建某个 ADO 类的对象，而必须通过这些封装类的 CreateInstance()函数来创建一个对应的对象。如：

```
ConnectionPtr   m_pAdoConnect = NULL;
//创建一个 Connection 对象
m_pAdoConnect.CreateInstance(__uuidof(Connection));
```

下面通过一个例子来说明上面各个类的具体用法。首先通过 MFC 的向导新建一个基于对话框的工程，在 MFC 向导的第 2 步选中“Dialog based”单选按钮，其他都是用默认选项。

在对话框类的头文件的所有#include 宏之后及类声明之前加入下面的#import 宏：

```
#import "D:\Program Files\Common Files\System\ado\msado15.dll"
no_namespace rename("EOF", "EndOfFile")
```

这个#import 宏应根据不同系统的 ado 所在目录及 msado**.dll 而相应改动。

打开 Resource View，在对话框内设置如图 13.47 所示的控件及其布局。

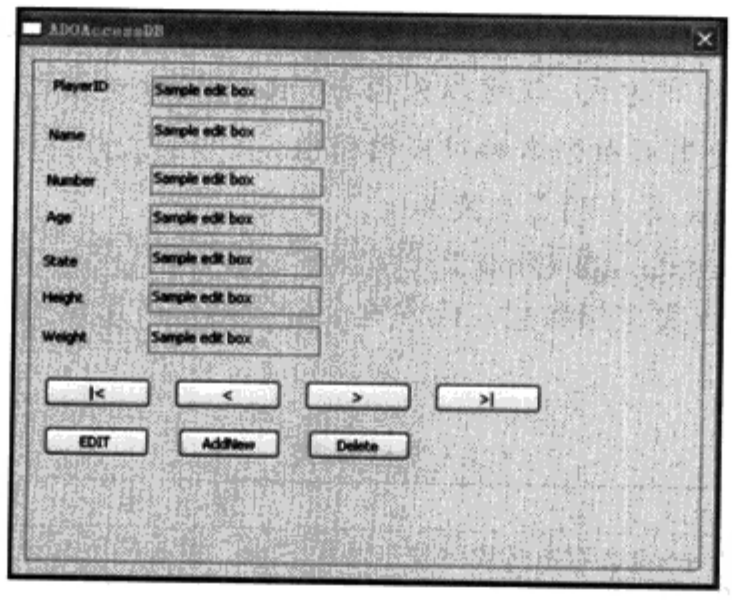


图 13.47　ADO 访问数据库程序的布局

中间的 4 个按钮分别表示移到第一个记录、移到前一个记录、移到后一个记录及移到最后一个记录；下面的 3 个按钮分别表示修改记录、插入一个新记录及删除当前记录的操作；上面的 7 个 Edit 控件分别关联表的 7 个域，为此在对话框类内定义变量与控件关联，定义如下：

```
public:
long m_lPlayerID;
CStringW m_strName;
int m_iNumber;
int m_iAge;
```

```
CStringW m_strState;
CStringW m_strHeight;
CStringW m_strWeight;
```

初始化及与控件数据交换如下：

```
CADOAccessDBDlg::CADOAccessDBDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CADOAccessDBDlg::IDD, pParent)
,m_lPlayerID(0),m_strName(_T("")),m_iNumber(0),m_iAge(0),
m_strState(_T("")),m_strHeight(_T("")),m_strWeight(_T(""))
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
void CADOAccessDBDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Text(pDX, IDC_EDIT_PLAYERID, m_lPlayerID);
    DDX_Text(pDX,IDC_EDIT_NAME,m_strName);
    DDX_Text(pDX,IDC_EDIT_NUMBER,m_iNumber);
    DDX_Text(pDX,IDC_EDIT_AGE,m_iAge);
    DDX_Text(pDX,IDC_EDIT_STATE,m_strState);
    DDX_Text(pDX,IDC_EDIT_HEIGHT,m_strHeight);
    DDX_Text(pDX,IDC_EDIT_WEIGHT,m_strWeight);
}
```

下面定义_ConnectionPtr 及_RecordsetPtr 对象来实现 ADO 数据库连接和操作：

```
public:
    _ConnectionPtr m_pAdoConnect;
    _RecordsetPtr m_pAdoRecordset;
```

在构造函数中，这两个成员都先初始化为 NULL，真正的赋值在对话框类的 OnInitDialog()函数内进行：

```
//
    if(FAILED(::CoInitialize(NULL))){
        ::AfxMessageBox(_T("fail to CoInitialize(NULL)"));
        PostQuitMessage(-8);
    }
    //实例化一个 Connection 对象
    HRESULT hr =    m_pAdoConnect.CreateInstance(__uuidof(Connection));
    if( FAILED(hr) ){
        ::AfxMessageBox(_T("fail to create instance for _ConnectPtr"));
        PostQuitMessage(-8);
    }
                                    //这里的连接使用 ODBC 驱动
    bstr_t strConnect = "DSN=RocketsPlayers;\
                                    DBQ=E:\\gucas\\vs2010projects\\RocketsPlayers.mdb;\
                                    DriverId=25;FIL=MS Access;MaxBufferSize=2048;PageTimeout=5;";
    try{
                                    //建立与数据库的连接
        m_pAdoConnect->Open(strConnect,"clc"," ",NULL);
    }
    catch (_com_error &e)
     {
      ::AfxMessageBox(e.Description());
```

```
        PostQuitMessage(-8);
    }
    m_pAdoRecordset = NULL;
    hr = m_pAdoRecordset.CreateInstance(__uuidof( Recordset ));
    if( FAILED(hr)){
        ::AfxMessageBox(_T("fail to create instance for _RecordsetPtr"));
        PostQuitMessage(-8);
    }
                        //利用打开的连接，对指定的表进行查询，取回查询结果
    m_pAdoRecordset->Open(_variant_t("Rockets"),
            _variant_t((IDispatch *)m_pAdoConnect,true), adOpenKeyset,
            adLockOptimistic, adCmdTable);
    ExchangeDatasWithControls();
```

这里对 Rockets 的整个表进行查询，相当于执行下面的 SQL 语句：

```
SELECT * FROM Rockets;
```

使用结果集及连接完毕，要调用 Close()函数将它们关闭，这可以在对话框类的析构函数中执行：

```
CADOAccessDBDlg::~CADOAccessDBDlg(){
    if( m_pAdoRecordset ){
        if( m_pAdoRecordset->State == adStateOpen ){
            m_pAdoRecordset->Close();
        }
    }
    if( m_pAdoConnect ){
        if( m_pAdoConnect->State == adStateOpen ){
            m_pAdoConnect->Close();
        }
    }
    ::CoUninitialize();
}
```

调用 Recordset 对象的 Open()函数取得查询结果后，就要在控件上将其显示出来，这通过下面将要定义的 ExchangeDatasWithControls()函数来完成。在对话框类中定义这个函数为 public，如下：

```
void ExchangeDatasWithControls(bool output=true);
```

函数实现如下：

```
void CADOAccessDBDlg::ExchangeDatasWithControls(bool output/*=true*/){
    if(output){
                        //将记录集内的当前记录的各个域的值赋给与控件相关的变量
        m_lPlayerID = (long)(m_pAdoRecordset->Fields->
            Item[_variant_t("PlayerID")]->Value);
        m_strName = (CStringW)(m_pAdoRecordset->Fields->
            Item[_variant_t("Name")]->Value);
        m_iNumber = (int)(m_pAdoRecordset->Fields->
            Item[_variant_t("Number")]->Value);
        m_iAge = (int)(m_pAdoRecordset->Fields->
            Item[_variant_t("Age")]->Value);
        m_strState = (CStringW)(m_pAdoRecordset->Fields->
            Item[_variant_t("State")]->Value);
        m_strHeight = (CStringW)(m_pAdoRecordset->Fields->
            Item[_variant_t("Height")]->Value);
        m_strWeight = (CStringW)(m_pAdoRecordset->Fields->
            Item[_variant_t("Weight")]->Value);
```

```
    }
    //将变量传送到控件显示
    UpdateData(False);
}
```

需要提到的是，通过 Recordset 对象内的 Fields 集合访问当前记录的指定域的值的方法是：

```
pRecordset->Fields->Item["列名"]->Value 或
pRecordset->Fields-> GetItem("列名")->Value
```

其中 pRecordset 为指向 Recordset 对象的指针。

通过这种方法，还可以将值写入记录的域：

```
pRecordset->Fields-> GetItem("列名")->Value = someValue;
```

最后的 UpdateDate()函数更新控件到成员变量的数据传输，参数指定传输方向：如果传入 False，则将数据从成员变量传输到控件；如果为 True，则将控件接收到的新值传输到成员变量。

完成这些后，就可以定义关于记录移动的那 4 个按钮的单击事件处理函数了。显然只有当前不是第一个记录时才允许移到第一个记录及向前移动一个记录可用；同样只有当前不是最后一个记录时，才允许移到最后一个记录及向后移动一个记录可用。而初始时，记录指针指向第一个记录，所以移到第一个记录及向前移动一个记录这两个按钮不可用。另外，在用户修改记录的值及插入一个新记录期间不希望这 4 个按钮可用，用户更新完毕后再恢复这 4 按钮的可用状态，这需要引入 4 个 BOOL 成员变量来分别记录当前 4 个按钮的可用状态。与前面类似，设置访问状态，有只读（查询）、修改及插入 3 个状态。在后两个状态下，这 4 个移动记录的按钮将不可用。在对话框类头文件中定义如下：

```
protected:
    //代表 3 个访问状态：只读、修改、插入
    enum AccessMode{QUERY,UPDATE,ADDNEW};
    AccessMode m_accMode;
    BOOL m_bEnableMvFirstButton;
    BOOL m_bEnableMvLastButton;
    BOOL m_bEnableMvPrevButton;
    BOOL m_bEnableMvNextButton;
```

在构造函数中的初始化如下：

```
m_accMode = QUERY;
m_bEnableMvFirstButton = False;
m_bEnableMvLastButton = False;
m_bEnableMvPrevButton = False;
m_bEnableMvNextButton = False;
```

真正有用的初始化是在 OnInitDialog()函数内，在这个函数的最后、return 语句之前，初始化这 4 个按钮的可用状态，代码如下：

```
this->GetDlgItem(IDC_BUTTON_FIRSTRECORD)->EnableWindow(False);
this->GetDlgItem(IDC_BUTTON_PREVRECORD)->EnableWindow(False);

this->GetDlgItem(IDC_BUTTON_LASTRECORD)->EnableWindow(TRUE);
this->GetDlgItem(IDC_BUTTON_NEXTRECORD)->EnableWindow(TRUE);
m_bEnableMvFirstButton = FALSE;
m_bEnableMvPrevButton = FALSE;
m_bEnableMvLastButton = TRUE;
m_bEnableMvNextButton = TRUE;
```

右击移动记录的按钮，在“Add Event Handler”命令下选择单击事件处理函数，分别编辑它们的单击事件处理函数。

移到第一个记录的按钮的事件处理函数为：

```
void CADOAccessDBDlg::OnBnClickedButtonFirstrecord()
{
    // TODO: Add your control notification handler code here
    //调用 Recordset 对象的 MoveFirst()方法，使当前记录为第一个记录
    m_pAdoRecordset->MoveFirst();
    //将当前记录的各个域的值赋给对应的成员变量，进而传输给控件显示
    ExchangeDatasWithControls();
    //设置按钮可用状态
    this->GetDlgItem(IDC_BUTTON_FIRSTRECORD)->EnableWindow(FALSE);
    this->GetDlgItem(IDC_BUTTON_PREVRECORD)->EnableWindow(FALSE);
    this->GetDlgItem(IDC_BUTTON_LASTRECORD)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_NEXTRECORD)->EnableWindow(TRUE);
    m_bEnableMvFirstButton = FALSE;
     m_bEnableMvLastButton = TRUE;
     m_bEnableMvPrevButton = FALSE;
     m_bEnableMvNextButton = TRUE;

}
```

向前移动一个记录的按钮的事件处理函数为：

```
void CADOAccessDBDlg::OnBnClickedButtonPrevrecord()
{
    // TODO: Add your control notification handler code here
    //先试探性地向前移动一个记录，接着查询是否过界，如果是则移动到第一个记录
    m_pAdoRecordset->MovePrevious();
    if(m_pAdoRecordset->BOF){
        this->OnBnClickedButtonFirstrecord();
        return;
    }
    ExchangeDatasWithControls();
    this->GetDlgItem(IDC_BUTTON_LASTRECORD)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_NEXTRECORD)->EnableWindow(TRUE);
    m_bEnableMvLastButton = TRUE;
    m_bEnableMvNextButton = TRUE;

}
```

这里查看了 Recordset 对象的 BOF 属性，来判断是否已经到达第一个记录之前。

移到最后一个记录的按钮的事件处理函数为：

```
void CADOAccessDBDlg::OnBnClickedButtonLastrecord()
{
    // TODO: Add your control notification handler code here
    m_pAdoRecordset->MoveLast();
    ExchangeDatasWithControls();
    this->GetDlgItem(IDC_BUTTON_FIRSTRECORD)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_PREVRECORD)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_LASTRECORD)->EnableWindow(FALSE);
    this->GetDlgItem(IDC_BUTTON_NEXTRECORD)->EnableWindow(FALSE);
    m_bEnableMvFirstButton = TRUE;
     m_bEnableMvLastButton = FALSE;
     m_bEnableMvPrevButton = TRUE;
     m_bEnableMvNextButton = FALSE;
```

```
}
```

向后移动一个记录的按钮的事件处理函数为：

```
void CADOAccessDBDlg::OnBnClickedButtonNextrecord()
{
    // TODO: Add your control notification handler code here
    m_pAdoRecordset->MoveNext();
    if(m_pAdoRecordset->EndOfFile ){
        this->OnBnClickedButtonLastrecord();
        return;
    }
    ExchangeDatasWithControls();
    this->GetDlgItem(IDC_BUTTON_FIRSTRECORD)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_PREVRECORD)->EnableWindow(TRUE);
    m_bEnableMvFirstButton = TRUE;
    m_bEnableMvPrevButton = TRUE;
}
```

这里查看 Recordset 对象的 EndOfFile 属性，判断是否到达最后一个记录之后，如果是，则移到最后一个记录。

至此关于查询及移动记录的处理就完成了，下面介绍关于更新数据源的最下面的 3 个按钮的事件处理函数。

修改记录的“EDIT”按钮的事件处理函数如下：

```
void CADOAccessDBDlg::OnBnClickedButtonEditUpdate()
{
        // TODO: Add your control notification handler code here
        //查询 Recordset 对象的 adUpdate 属性，判断是否允许对记录进行修改(更新)
    if( !m_pAdoRecordset->Supports( adUpdate ) ){
        AfxMessageBox(_T("this DB recordset does not support update"));
        return;
    }
    if( m_accMode == QUERY ){
            //如果原来是 QUERY 状态，则进入 UPDATE 状态，设置按钮显示为 Update，其余按钮不可用
        this->GetDlgItem(IDC_BUTTON_EDIT_UPDATE)->
            SetWindowTextW(_T("Update"));
        this->GetDlgItem(IDC_BUTTON_ADDNEW)->EnableWindow(FALSE);
        this->GetDlgItem(IDC_BUTTON_DELETE)->EnableWindow(FALSE);
        this->GetDlgItem(IDC_BUTTON_FIRSTRECORD)->
            EnableWindow(FALSE);
        this->GetDlgItem(IDC_BUTTON_PREVRECORD)->
            EnableWindow(FALSE);
        this->GetDlgItem(IDC_BUTTON_LASTRECORD)->
            EnableWindow(FALSE);
        this->GetDlgItem(IDC_BUTTON_NEXTRECORD)->
            EnableWindow(FALSE);
        m_accMode = UPDATE;
            //设置控件可写，等待用户输入新的值
        ((CEdit*)this->GetDlgItem(IDC_EDIT_NAME))->SetReadOnly(FALSE);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_NUMBER)))->
            SetReadOnly(FALSE);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_AGE)))->SetReadOnly(FALSE);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_STATE)))->SetReadOnly(FALSE);
```

```
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_HEIGHT)))->SetReadOnly(FALSE);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_WEIGHT)))->
            SetReadOnly(FALSE);
    }
    else if(m_accMode == UPDATE){
                    //如果原来为 UPDATE 状态，则要求确认修改后，对记录进行修改，然后回到 QUERY 状态
        if( AfxMessageBox(_T("Confirm to update this record"),
            MB_YESNO) == IDYES ){
            try{
                UpdateData(TRUE);
                m_pAdoRecordset->Fields->GetItem(_variant_t("Age"))->
                    Value = _variant_t(m_iAge);
                m_pAdoRecordset->Fields->Item[_variant_t("Name")]->
                    Value = _variant_t(m_strName);
                m_pAdoRecordset->Fields->Item[_variant_t("Number")]->
                    Value = _variant_t(m_iNumber);
                m_pAdoRecordset->Fields->Item[_variant_t("State")]->
                    Value = _variant_t(m_strState);
                m_pAdoRecordset->Fields->Item[_variant_t("Height")]->
                    Value = _variant_t(m_strHeight);
                m_pAdoRecordset->Fields->Item[_variant_t("Weight")]->
                    Value = _variant_t(m_strWeight);
                m_pAdoRecordset->Update();
            }
            catch(_com_error &e){
                AfxMessageBox(_T("Error in Update a record:")
                    +e.Description() );
                PostQuitMessage(-8);
            }
    }
            //按钮显示为 EDIT,3 个更新操作的按钮可用
    this->GetDlgItem(IDC_BUTTON_EDIT_UPDATE)->
        SetWindowTextW(_T("Edit"));
    this->GetDlgItem(IDC_BUTTON_ADDNEW)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_DELETE)->EnableWindow(TRUE);
    m_accMode = QUERY;
            //回到只读模式，控件设置为只读
    ((CEdit*)this->GetDlgItem(IDC_EDIT_NAME))->SetReadOnly(TRUE);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_NUMBER)))->
        SetReadOnly(TRUE);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_AGE)))->SetReadOnly(TRUE);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_STATE)))->SetReadOnly(TRUE);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_HEIGHT)))->
        SetReadOnly(TRUE);
    ((CEdit*)(this->GetDlgItem(IDC_EDIT_WEIGHT)))->
        SetReadOnly(TRUE);
            //恢复移动记录的 4 个按钮的可用状态
    this->GetDlgItem(IDC_BUTTON_FIRSTRECORD)->
        EnableWindow(m_bEnableMvFirstButton);
    this->GetDlgItem(IDC_BUTTON_PREVRECORD)->
        EnableWindow(m_bEnableMvPrevButton);
```

```
            this->GetDlgItem(IDC_BUTTON_LASTRECORD)->
                EnableWindow(m_bEnableMvLastButton);
            this->GetDlgItem(IDC_BUTTON_NEXTRECORD)->
                EnableWindow(m_bEnableMvNextButton);
        }
}
```

插入新记录的“AddNew”按钮的事件处理函数：

```
void CADOAccessDBDlg::OnBnClickedButtonAddnew()
{
                    //查询 adAddNew 属性，判断是否可以插入新记录
    if( !m_pAdoRecordset->Supports( adAddNew ) ){
        AfxMessageBox(_T("this DB recordset does not\
            support AddNew update"));
        return;
    }
    if( m_accMode = = QUERY ){
        this->GetDlgItem(IDC_BUTTON_ADDNEW)->
            SetWindowTextW(_T("Update"));
        this->GetDlgItem(IDC_BUTTON_EDIT_UPDATE)->
            EnableWindow(FALSE);
        this->GetDlgItem(IDC_BUTTON_DELETE)->EnableWindow(FALSE);
        m_accMode = UPDATE;
                    //移到最后一个记录，调用 AddNew()函数，为插入一个新记录做准备
        m_pAdoRecordset->MoveLast();
        m_pAdoRecordset->AddNew();
        m_lPlayerID = m_pAdoRecordset->Fields->
            Item[_variant_t("PlayerID")]->Value;
        m_strName = _T("");
        m_iNumber = 0;
        m_iAge = 0;
        m_strState = _T("");
        m_strHeight = _T("");
        m_strWeight = _T("");
        UpdateData(FALSE);
        ((CEdit*)this->GetDlgItem(IDC_EDIT_NAME))->SetReadOnly(FALSE);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_NUMBER)))->
            SetReadOnly(FALSE);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_AGE)))->SetReadOnly(FALSE);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_STATE)))->SetReadOnly(FALSE);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_HEIGHT)))->
            SetReadOnly(FALSE);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_WEIGHT)))->
            SetReadOnly(FALSE);
        this->GetDlgItem(IDC_BUTTON_FIRSTRECORD)->
            EnableWindow(FALSE);
        this->GetDlgItem(IDC_BUTTON_PREVRECORD)->
            EnableWindow(FALSE);
        this->GetDlgItem(IDC_BUTTON_LASTRECORD)->
            EnableWindow(FALSE);
        this->GetDlgItem(IDC_BUTTON_NEXTRECORD)->
            EnableWindow(FALSE);
```

```
    }
        else if( m_accMode == UPDATE ){
        UpdateData(TRUE);
        m_pAdoRecordset->Fields->GetItem(_variant_t("Age"))->Value
            = _variant_t(m_iAge);
        m_pAdoRecordset->Fields->Item[_variant_t("Name")]->Value =
            _variant_t(m_strName);
        m_pAdoRecordset->Fields->Item[_variant_t("Number")]->Value =
            _variant_t(m_iNumber);
        m_pAdoRecordset->Fields->Item[_variant_t("State")]->Value =
            _variant_t(m_strState);
        m_pAdoRecordset->Fields->Item[_variant_t("Height")]->Value =
            _variant_t(m_strHeight);
        m_pAdoRecordset->Fields->Item[_variant_t("Weight")]->Value =
            _variant_t(m_strWeight);
        if( AfxMessageBox(_T("Confirm to addnew(insert) this record"),
            MB_YESNO) == IDYES ){
            try{
                m_pAdoRecordset->Update();
            }
            catch(_com_error &e){
                AfxMessageBox(_T("Error in Update AddNew:")+
                    e.Description() );
                PostQuitMessage(-8);
            }
        }
        else{
            try{
                m_pAdoRecordset->CancelUpdate();
            }
            catch(_com_error &e){
                AfxMessageBox(_T("Error in CancelUpdate AddNew:")
                    +e.Description() );
                PostQuitMessage(-8);
            }
        }
        this->GetDlgItem(IDC_BUTTON_ADDNEW)->
            SetWindowTextW(_T("AddNew"));
        this->GetDlgItem(IDC_BUTTON_EDIT_UPDATE)
            ->EnableWindow(TRUE);
        this->GetDlgItem(IDC_BUTTON_DELETE)->EnableWindow(TRUE);
        m_accMode = QUERY;
        ((CEdit*)this->GetDlgItem(IDC_EDIT_NAME))->SetReadOnly(TRUE);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_NUMBER)))->
            SetReadOnly(TRUE);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_AGE)))->SetReadOnly(TRUE);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_STATE)))->SetReadOnly(TRUE);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_HEIGHT)))->
            SetReadOnly(TRUE);
        ((CEdit*)(this->GetDlgItem(IDC_EDIT_WEIGHT)))->
            SetReadOnly(TRUE);
```

```
            this->GetDlgItem(IDC_BUTTON_FIRSTRECORD)->
                EnableWindow(m_bEnableMvFirstButton);
            this->GetDlgItem(IDC_BUTTON_PREVRECORD)->
                EnableWindow(m_bEnableMvPrevButton);
            this->GetDlgItem(IDC_BUTTON_LASTRECORD)->
                EnableWindow(m_bEnableMvLastButton);
            this->GetDlgItem(IDC_BUTTON_NEXTRECORD)->
                EnableWindow(m_bEnableMvNextButton);
            this->OnBnClickedButtonLastrecord();
        }
    }
```

“Delete”按钮的事件处理函数：

```
void CADOAccessDBDlg::OnBnClickedButtonDelete()
{
                // TODO: Add your control notification handler code here
                //查询 Recordset 对象的 adDelete 属性，判断是否支持删除记录操作
    if( !m_pAdoRecordset->Supports( adDelete    ) ){
        AfxMessageBox(_T("this DB recordset does not support\
            Delete update"));
        return;
    }
    if( AfxMessageBox(_T("Confirm to delete this record"),
        MB_YESNO) == IDYES ){
        try{
                //选参数 adAffectCurrent，只影响本记录，然后移到下一个记录位置
            m_pAdoRecordset->Delete(adAffectCurrent);
            this->OnBnClickedButtonNextrecord();
        }
        catch(_com_error &e){
                AfxMessageBox(_T("Error in Delete this record:")
                    +e.Description() );
                PostQuitMessage(-8);
            }
    }
}
```

程序最后的运行结果如图 13.48 所示。

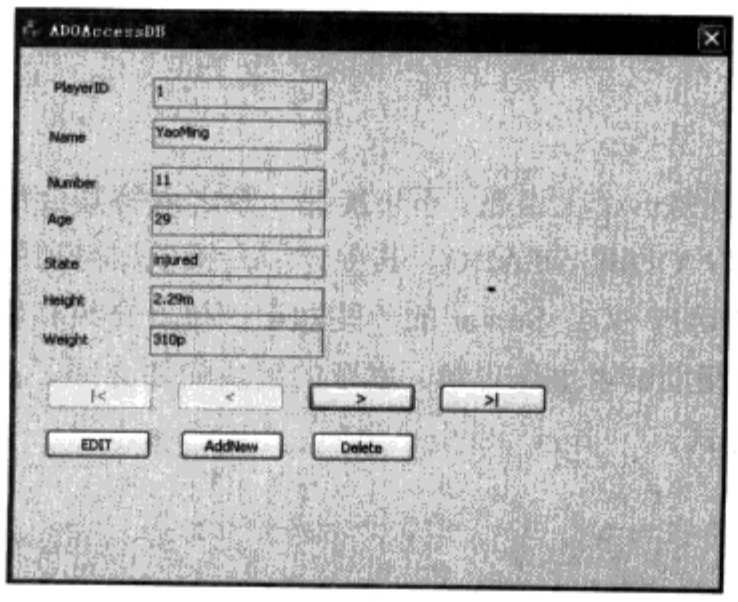


图 13.48　ADO 访问数据库的运行结果

13.7 ADO 访问 SQL Server 数据库

下面介绍两种访问 SQL Server 数据库的方式。

（1）直接向 SQL Server 建立连接，这样在 Connection 对象建立连接时就需要使用 SQLOLEDB 作为数据源提供者，具体需要这些信息。

- Provider: 数据源提供者，一般使用 SQLOLEDB。
- Server: 服务器名。
- Database: 要访问的数据库名。

这 3 项是必需的，建立的连接分为两种。

一种是建立可靠连接，此时不需要提供账户和密码，只需设置：

```
Trusted_Connection=yes;
```

另一种就是标准安全方式，需要提供登录账户和密码进行验证：

```
Uid=myUserID;Pwd=myPassword。
```

一个可能的连接字符串为

```
_bstr_t strConnect="Provider=SQLOLEDB; Server=server_name;"
  "Database=database_name; Uid=user_name; Pwd=password;";
```

使用这个字符串再调用 Connection 对象的 Open()函数就能建立连接。

（2）通过 ODBC 驱动再向 SQL Server 建立连接，这里需要提供 ODBC 驱动程序的名称，才能建立连接，这通过设置 Driver 属性的值来实现。

对于 SQL Server 2000，使用：

```
Driver={SQL Server}
```

而对于 SQL Server 2005，使用：

```
Driver={SQL Native Client}
```

其他还要设置 Server、Database 等。

与第（1）种方式相似，一个可能的连接字符串为：

```
_bstr_t strConnect="Provider=SQLOLEDB; Server=server_name;"
 "Database=database_name; Trusted_Connection=yes;";
```

建立好连接之后，就可以通过 Recordset 对象访问记录，修改、添加、删除记录，这与前面介绍的方法是一致的，不再赘述。

13.8 小结

利用 MFC 进行数据库访问是一个大课题，不可能在一章之内将其全部讲明白。本章力图介绍目前最常用的两种数据库访问技术：ODBC 和 ADO。其中详细介绍了如何利用 ODBC 技术来连接、访问、更新数据库；然后介绍了关于访问 SQL Server 的一些知识；接着介绍利用 ADO 进行数据库访问的技术，并介绍了一个实例，希望帮助读者加深理解。本章的最后简要介绍了在使用 ADO 访问 SQL Server 数据库时的两种不同的连接方式。

第 14 章　Visual C++2010 MFC 多线程程序设计

14.1　进程和多线程的概念

进程就是应用程序的一个运行实例，它具有独立的内存空间、代码段、数据段、文件句柄等系统资源，进程是系统分配资源的基本单位。现在的 32 位 Windows 操作系统都支持多任务处理，即可以同时运行多个进程，在每个进程内还支持多线程处理。

线程是进程内的一个执行路径，是操作系统调度处理器时间的基本单位。在 Win32 操作系统中，采用抢占式多线程调度策略。系统为每个线程都分配一个优先级，根据这个动态变化的优先级，系统来决定选择哪个线程来独占 CPU，即运行这个线程，系统会根据当前所处的状态来决定分配一定的 CPU 时间片给线程，一旦线程使用完这段时间，就会被暂停，并进入新一轮的线程调度。实际上，系统总是选择当前优先级最高的线程来运行，在某一时刻，如果出现某一个优先级更高的线程，那么当前运行的线程即使还没有用完分配给它的 CPU 时间片，也会被替换出去，这就是抢占的含义。如何选择线程看起来相当“简单”，难的是如何动态调整各个线程的优先级，使各个线程的运行达到平衡，以避免某些高优先级的线程长时间占用 CPU，而其他线程却长时间得不到执行，为此，Windows 操作系统设计了相当复杂的调度策略。由于系统在线程调度，切换线程运行的时间很短，让人感觉这些程序，更严格地说是这些线程好像是同时在运行。但是实际上在任何时刻，只有一个线程在占用 CPU，在真正运行，一般称之为活动线程，其他的线程都在系统队列中等待被调度。

每个进程在启动时都会自动创建并运行一个线程，这个线程称为该进程的主线程，一旦主线程终止，那么整个进程就终止了。Windows 系统支持在一个进程内创建多个线程，就是说除了主线程之外，还可以再创建多个线程，同一进程内的所有线程共享进程的地址空间、全局对象等资源，因此线程也常被称为“轻量级进程”。

在 Windows 提供的 API 中，完全可以处理诸如线程的创建、同步等操作；但是由于这是比较底层的操作，所以对初学者来说相当繁杂。MFC 对这些 API 提供了面向对象的封装，使得多线程的程序设计更加简单易懂，所以本章只介绍如何利用 MFC 来进行多线程的程序设计。

在 MFC 中，类 CWinThread 封装了可执行的线程，所有想使用 MFCAPI 的线程都必须通过 CWinThread 类的对象创建。通过 Windows API 提供的 CreateThread()函数也可以创建线程，但是在这个线程内就无法使用 MFCAPI，所以在 MFC 程序内，最好使用 CWinThread 类来实现多线程。另外，CWinThread 类还是线程安全的，所有线程的局部数据都由 CWinThread 类的对象管理。

对于 Windows 的程序来说，所有的线程都是一样的。但是在 MFC 中，CWinThread 类却区分支持两种不同类型的线程：工作者线程（Worker Threads）和用户界面线程（User-interface Threads）。这两类线程的区别是工作者线程没有消息循环而用户界面线程有。工作者线程主要用来执行后台的计算，不需要用户交互，工作者线程很适合执行一些可放到后台执行并可从程序的其他部分分离出来的独立任务。例如，播放背景音乐就是一个很好的工作者线程的应用，这个线程的工作就是反复播放一个音频（不需要用户交互），直到主线程通知结束为止。而用户界面线程可以创建窗口，建立消息循环，与用户交互，处理用户发送给线程对应窗口的消息。MFC 程序的主线程为继承自 CWinApp 类的对象，

而 CWinApp 类继承自 CWinThread 类，所以 MFC 程序的主线程就是一个 CWinThread 对象，而且是用户界面线程。通过在 MFC 程序中创建附加的 CWinThread 对象，就可以实现多线程运用。

14.2 线程的创建

尽管用户可以通过创建一个 CWinThread 对象，并调用 CWinThread 类的 CreateThread()方法来创建一个新的线程，但是用户一般都不这样做，而是通过 MFC 提供的一个更加方便的 AfxBeginThread()函数来创建线程，这个函数既可用于创建工作者线程，也可用于创建用户界面线程。但是，这两种线程的创建方式很不相同，为此 MFC 提供了这个函数的两种不同的重载形式。当然，AfxBeginThread()并不仅仅是对 Windows API 提供的 CreateThread()函数的简单封装，在创建线程的同时还负责初始化主结构的状态，并执行线程安全检查。

14.2.1 创建工作者线程

创建工作者线程的过程相当简单，只需指定一个控制函数，传递函数参数，并设置线程的相应属性即可。对于工作者线程的创建，AfxBeginThread()函数的原型是：

```
CWinThread* AfxBeginThread(
    AFX_THREADPROC pfnThreadProc,
    LPVOID pParam,
    int nPriority = THREAD_PRIORITY_NORMAL,
    UINT nStackSize = 0,
    DWORD dwCreateFlags = 0,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL
);
```

这个函数创建一个新的 CWinThread 对象，并创建一个依附于这个 CWinThread 对象的工作者线程。该函数返回一个指向这个 CWinThread 对象的指针，通过这个指针可以控制进程的执行、暂停及设置线程优先级等操作。

这个函数接受 6 个参数，各个参数的含义如下。

- 参数 pfnThreadProc 表示一个指向绑定到工作者线程的控制函数的指针。工作者线程启动后将“进入”这个控制函数，通过指定不同的控制函数可以使工作者线程执行不同的功能。这个参数不能为 NULL，必须指定。

工作者线程对应的控制函数的原型必须是：

```
UINT MyControllingFunction( LPVOID pParam );
```

这是一个回调函数，必须是静态成员函数或全局函数。参数 pParam 用于向控制函数传递参数：如果控制函数不需要参数，则传入 NULL；如果控制函数需要一个或多个不同类型的参数，则一般作法就是将所有这些参数定义到一个结构体内，然后将 pParam 指定为指向该参数结构体的指针，在控制函数中就可以通过这个指针来访问参数了，后面会通过例子来说明怎样向控制函数传递参数。

- 参数 pParam 就是需要传递给控制函数的那个 pParam 参数。
- 参数 nPriority 指定新线程的优先级。默认值为 THREAD_PRIORITY_NORMAL，表示新创建的线程与它所附属的进程具有相同的优先级类型，这是最常用的。通过指定不同的值，可以调节线程的优先级。
- 参数 nStackSize 指定新线程所用的栈的大小，单位为字节。默认值为 0，表示新线程将使用与

父线程相同大小的栈。每一个线程在运行时都有属于自己的栈，这样即使不同线程使用相同的控制函数，但只要不访问静态数据或全局数据就不会有任何问题。

- 参数 dwCreateFlags 控制线程的创建方式。有两种不同的方式：如果值为 0，表示一旦线程创建成功，就马上启动线程，进入运行状态，这是默认值；而如果值为 CREATE_SUSPENDED，表示创建线程对象后，使线程进入暂停状态，直到调用线程对应的 CWinThread 对象的 ResumeThread()函数，这个线程才会开始运行。使用这个参数是希望对线程有更多的控制，在启动线程之前，用户可能需要设置一些特殊的属性。
- 参数 lpSecurityAttrs 指定线程的安全属性，默认值 NULL 表示使用与父线程相同的安全属性。

对于创建一般的工作者线程，只需设置前两个参数，后面都使用默认值即可，这也是最常用的方式。

这里有一个问题需要注意：如果创建多个工作者线程，而这些工作者线程都绑定到同一个控制函数，那么就可能会出现重入问题。如果控制函数操作的所有对象都是在栈上分配的局部变量，那么由于每个线程都维护自己的栈，就不会有任何问题。而如果控制函数访问静态对象或全局对象，特别是写访问，如果控制不当，就会导致冲突，程序行为将不可预知。

下面举一个简单的例子来说明工作者线程的创建，首先定义控制函数：

```
UINT ThreadFunc( LPVOID pParam){
    int n = *((int*)pParam);
    int i;
    //这里执行一段时间的空操作
    for(i=0;i<n;i++);
    return 0;
}
```

这个控制函数根据传入的参数，执行一段时间的空操作，函数返回值 0，代表正常结束。

接着，就可以创建工作者线程了：

```
// m_iCount 为某个成员
CWinThread* pThread = AfxBeginThread(ThreadFunc,&m_iCount);
```

这里就指定前面的两个参数，后面都使用默认参数。如果希望线程开始为暂停状态，则可以将 dwCreateFlags 参数设置为 CREATE_SUSPENDED：

```
//这里线程创建后将进入暂停状态
CWinThread* pThread = AfxBeginThread(ThreadFunc,&m_iCount,
THREAD_PRIORITY_NORMAL,0, CREATE_SUSPENDED,NULL);
…
//直到调用 ResumeThread()函数，线程才会开始运行
pThread->ResumeThread();
```

14.2.2　创建用户界面线程

创建用户界面线程的过程与创建工作者线程的过程截然不同，工作者线程的行为是由绑定的控制函数定义的，而用户界面线程的行为是由从 CWinThread 类派生出来的可动态创建类（提供的消息循环）来控制。一般来说，用户界面线程会创建一个独立的窗口，建立消息循环，定义消息映射和处理函数，从而达到与用户交互的目的，这与一般的 MFC 程序很相似。创建用户界面线程的 AfxBeginThread()函数的原型是：

```
CWinThread* AfxBeginThread(
    CRuntimeClass* pThreadClass,
    int nPriority = THREAD_PRIORITY_NORMAL,
```

```
    UINT nStackSize = 0,
    DWORD dwCreateFlags = 0,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL
);
```

参数 pThreadClass 为一个指向 CWinThread 的一个派生类的运行时类对象的指针，这可以通过 RUNTIME_CLASS 宏来获得，其他参数的含义与上面相同，此处不再赘述。

创建用户界面线程的过程一般可遵循以下的 3 个步骤：

（1）针对特定的应用，从 CWinThread 类中派生新类；

（2）必须重载基类 CWinThread 的 InitInstance()函数，这个函数主要用来创建用户自定义的窗口，显示窗口并更新等操作；

（3）调用 AfxBeginThread()函数创建线程对象。

下面给出一个建立用户界面的例子，首先派生一个窗口类：

```
class CMyUIThread: public CWinThread {
                                                    //声明为可动态创建类
    DECLARE_DYNCREATE(CMyUIThread) ;
public:
                                                    //这个函数必须重载，初始化线程对应的窗口等操作
    virtual BOOL InitInstance();
};
IMPLEMENT_DYNCREATE(CMyUIThread, CWinThread);
BOOL InitInstance(){
                                                    //创建线程对应的主窗口
                                                    //这里假设 CMyMainWindow 为一个主窗口类
    m_pMainWnd = new CMyMainWindow;
                                                    //显示窗口并更新
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}
```

然后就可以通过 AfxBeginThread()函数来创建用户界面线程：

```
CWinThread*pMyUIThread =
    AfxBeginThread( RUNTIME_CLASS( CMyUIThread ) );
```

用户界面线程一旦启动，就会进入消息循环，与创建它的线程异步执行。

14.3 线程的终止

对于工作者线程，如果控制函数执行到返回语句 return、退出语句 exit 或者在控制函数内的某个位置调用函数 AfxEndThread()，那么工作者线程就会终止；而对于用户界面线程，如果执行到退出语句或接收到用户或其他线程发出的退出消息 WM_QUIT，或者在用户界面线程内调用了 AfxEndThread()函数，用户界面线程就会终止。

AfxEndThread()函数的原型是：

```
void AFXAPI AfxEndThread(
    UINT nExitCode,
    BOOL bDelete = TRUE
);
```

- 参数 nExitCode 指定线程的退出码，一般 0 表示正常退出。
- 参数 bDelete 指定是否马上将线程对应的 CWinThread 对象从内存中清除。默认值为 True，表示一旦线程终止，就将 CWinThread 对象自动删除，用户不用操心这些对象的空间回收问题。

在线程终止时，都会返回一个退出码来表示线程终止的原因。如果其他线程想查询这个退出码，进而确定线程的退出原因，可以通过 GetExitCode()函数来实现，这个函数的原型如下：

```
BOOL GetExitCodeThread(
    HANDLE hThread,
    LPDWORD lpExitCode
);
```

- 参数 hThread 表示想查询的线程的句柄。
- 参数 lpExitCode 是一个返回值，用来存放（结果）退出码。

如果指定线程仍处于生命期还没有终止，那么调用这个函数得到的结果值将是 STILL_ACTIVE。如果查询成功，返回 True（非 0 值，注意上面的线程没有终止时的查询也是成功的）；如果执行出错，返回 False（0 值）。通过调用 GetLastError()函数可以查看函数执行失败的原因。

下面的例子说明如何查询一个线程的退出码：

```
DWORD dwExitCode;
::GetExitCodeThread(pThread->m_hThread,&dwExitCode);
                                        //下面就可以检测 dwExitCode，进而分析线程退出原因，作相应处理
    …
```

上面的代码有一个问题：在默认情况下，MFC 的线程所对应的 CWinThread 对象在线程终止时就会自动被删除。如果想在线程结束后调用 GetExitCodeThread()函数来查询线程的退出码，这个函数要求传入线程的句柄，但是在线程结束后，这个线程句柄已经被自动清除，就是一个无效指针，这样上面的代码就会出问题。因此，为了查询线程的退出码，必须改变线程的这个自动删除属性。

在 CWinThread 类中提供了一个 BOOL 型的成员 m_bAutoDelete，指定是否希望在线程结束后自动删除线程对象，默认值为 True。可以在创建线程对象之后，线程开始运行之前，将 m_bAutoDelete 的值设置为 False，那么在线程结束之后就要求用户自己显式地删除线程对象的空间。当然，这里在创建线程时就要求 dwCreateFlags 参数为 CREATE_SUSPENDED，下面的代码演示了这个过程：

```
CWinThread *pThread = AfxBeginThread(ThreadFunc,pParams,
THREAD_PRIORITY_NORMAL,0, CREATE_SUSPENDED);
                                        //设置不要自动删除线程对象
pThread->m_bAutoDelete = FALSE;
                                        //开始启动线程
pThread->ResumeThread();
                                        //执行其他操作
…
                                        //这里查询线程的退出码，假设函数执行成功，
                                         对失败处理暂不考虑
DWORD dwExitCode;
::GetExitCodeThread(pThread->m_hThread,&dwExitCode);
                                        //如果线程还没有终止
if(dwExitCode == STILL_ACTIVE){
                                        //执行其他操作，等待线程终止后再查询
}
                                        //线程已经终止，需要显式删除线程对象
else{
                                        //删除线程对象
```

```
        delete pThread;
                                                    //对退出码进行处理
        ...
    }
```

14.4 设置线程的优先级

前面简单介绍过 Windows 系统根据线程优先级进行抢占式的过程。线程的优先级对系统的调度是相当重要的，可以通过设置线程的优先级来影响这个调度过程，从而满足特定的需求。比如，一个线程需要在短时间内接收一批大数据，那么通过将这个线程的优先级提高，就可以使线程能及时响应并接收数据。

除了在线程创建的时候，可以设置其优先级。在线程的生命期内，也可以动态地调整线程的优先级。通过 CWinThread 类提供的 GetThreadPriority()函数可以获取线程的当前优先级，而通过 SetThreadPriority()函数可以动态地改变线程的优先级。

这里所说的线程优先级实际上都是一个相对优先级，即相对于线程所附属的进程的优先级。在 Windows 系统中，进程的优先级可以取表 14.1 内列出的这些值。

表 14.1　　进程优先级

进程优先级	说　明
ABOVE_NORMAL_PRIORITY_CLASS0 x00008000	进程优先级在 NORMAL_PRIORITY_CLASS 之上，但又在 HIGH_PRIORITY_CLASS 之下。 Windows NT 和 Windows Me/98/95：不支持这个值
BELOW_NORMAL_PRIORITY_CLASS0 x00004000	进程优先级在 IDLE_PRIORITY_CLASS 之上，NORMAL_PRIORITY_CLASS 之下 Windows NT 和 Windows Me/98/95：不支持这个值
HIGH_PRIORITY_CLASS 0x00000080	对于实时性要求较高的任务使用这个优先级。属于这个优先级的进程的线程将优先抢占属于 normal 和 idle 进程的线程
IDLE_PRIORITY_CLASS 0x00000040	只有在系统处于空闲（idle）状态时，属于这个优先级进程的线程才能运行。这些线程的时间片将被具有更高优先级的进程的线程抢占
NORMAL_PRIORITY_CLASS 0x00000020	默认的进程优先级，不需要系统任何特殊的调度
REALTIME_PRIORITY_CLASS 0x00000100	这是进程的最高优先级，抢占所有优先级比它低的进程的线程的时间片

一般进程的优先级都是 NORMAL_PRIORITY_CLASS，但是在进程的生命期内可以通过函数::SetPriorityClass()来动态设置其优先级。

Windows 系统为每个线程都分配一个 0～31 的优先级，这是线程参与调度的最终优先级，值越大，优先级越高，越可能被系统调度分配到 CPU。这个线程的最终优先级就是由线程所属的进程优先级类型和线程本身的相对优先级共同决定的。通过设置线程的相对优先级，可以影响线程的最终优先级，进而影响线程的调度。

在线程的生命期内，可以通过 GetThreadPriority()及 SetThreadPriority()函数来获取和设置线程的相对优先级。这两个函数的原型分别如下：

```
int GetThreadPriority( );
BOOL SetThreadPriority(
    int nPriority
```

```
);
```

线程的相对优先级是一个 int 型，而且只能取表 14.2 中列出的 7 个不同的值，这里的优先级是从高到低排列的。

表 14.2　　线程的相对优先级

线程优先级	说　明
THREAD_PRIORITY_TIME_CRITICAL 15	如果进程优先级为 REALTIME_PRIORITY_CLASS，则线程最终优先级为 31；对其他优先级进程，线程最终优先级为 15
THREAD_PRIORITY_HIGHEST 2	线程的最终优先级等于进程优先级类型加 2
THREAD_PRIORITY_ABOVE_NORMAL 1	线程的最终优先级等于进程的优先级类型加 1
THREAD_PRIORITY_NORMAL 0	这是默认值，线程的最终优先级等于进程的优先级类型
THREAD_PRIORITY_BELOW_NORMAL -1	线程的最终优先级等于进程的优先级类型减 1
THREAD_PRIORITY_LOWEST -2	线程的最终优先级等于进程的优先级类型减 2
THREAD_PRIORITY_IDLE -15	如果进程优先级为 REALTIME_PRIORITY_CLASS，则线程最终优先级为 16；对其他优先级进程，线程最终优先级为 1

附　这里所说的线程的最终优先级指的是线程参与调度的“基点”优先级，并不是说线程的优先级就保持这个值不变，因为 Windows 系统的调度策略本身就会根据系统状态动态地改变线程的优先级，使这些线程都有机会被调度。

对于一般的程序，线程的优先级只要使用默认值 THREAD_PRIORITY_NORMAL 就足够了，除非是一些对实时性要求比较高的线程，才设置高的优先级；如果盲目地提高线程的优先级，将导致线程过分地占用 CPU，这样本线程的速度是提高了，但会导致其他线程的速度和响应能力大大降低，就会得不偿失。

14.5　暂停及重新启动线程

一个线程在生命期内，可能处于运行状态，此时可以参与操作系统调度以分配到 CPU 时间片而运行；也可能处于暂停状态，此时线程将被放入暂停队列，不参与系统调度，自然就不可能执行。

为了控制线程的状态，系统为每个线程都维护一个暂停计数器，只有当这个暂停计数器值为 0 时，线程才处于运行状态；而当这个值大于 0 时，线程将处于暂停状态。通过 CWindThread 类提供的 SuspendThread()函数及 ResumeThread()函数可以分别对暂停计数器执行加 1 及减 1 操作，从而可以控制线程的运行状态。

在前面介绍线程的创建时，如果提供 CREATE_SUSPENDED 参数，就会使线程的暂停计数器值初始化为 1，线程将处于暂停状态；而如果提供默认值 0，暂停计数器值初始化为 0，线程就处于运行状态。

SuspendThread()函数的原型是：

```
DWORD SuspendThread( );
```

将对应线程的暂停计数器值加 1。如果操作成功，返回原来的暂停计数器的值；否则返回负值 0xFFFFFFFF。

ResumeThread()函数的原型是：

```
DWORD ResumeThread( );
```

- 如果暂停计数器原来的值为 0，则线程原来就处于运行状态，那么暂停计数器值还是 0 不变，此时返回值为 0。
- 如果暂停计数器原来的值大于 0，则线程原来处于暂停状态，那么将暂停计数器的值减 1，并返回原来暂停计数器的值；进一步，如果此时暂停计数器的值变为 0，则线程将进入运行状态，

否则仍然处于暂停状态。

- 如果操作失败，将返回 0xFFFFFFFF。

这里有几点需要说明：

（1）暂停计数器的值不可能为负值，如果原来的值为 0，再调用 ResumeThread()函数，它的值还是 0，还是处于运行状态。

（2）如果一个线程被调用了多次 SuspendThread()函数，那么只有当至少调用了相同次数的 ResumeThread()函数时，才能使线程回到运行状态。

（3）线程可以自己调用 SuspendThread()，也可以是别的线程为它调用 SuspendThread()；但是一个处于暂停状态的线程不能自己调用 ResumeThread()以使自己恢复运行，而必须由其他线程为其调用 ResumeThread()。

（4）如果线程想自己睡眠一段时间，而后自动恢复运行状态，可以通过调用 Sleep()函数来实现。Sleep 函数的原型是：

```
void Sleep(DWORD dwMilliseconds);
```

使调用这个函数的线程至少睡眠一段指定的时间，参数 dwMilliseconds 设置睡眠时间，单位为毫秒（ms）。比如下面的调用：

```
::Sleep(1000);
```

使线程睡眠至少 1s 的时间，这段时间内线程将不能参与系统调度。具体睡眠多长时间，则根据系统的状态决定，也就是说，睡眠时间是不确定的，但是至少是 1s。

另外，还有一个比较有意思的地方就是设置睡眠时间为 0：

```
::Sleep(0);
```

这条语句的作用是，如果系统中存在比自己优先级更高或相等的线程，当前线程将暂停执行，并将自己的剩余时间片让给这些线程；而如果不存在这样的线程，则函数马上返回，当前线程继续往下执行直到用完自己的剩余时间片。

下面通过一个例子来说明如何在 MFC 程序中使用多线程，这是一个基于对话框的 MFC 程序，程序的运行界面如图 14.1 所示。

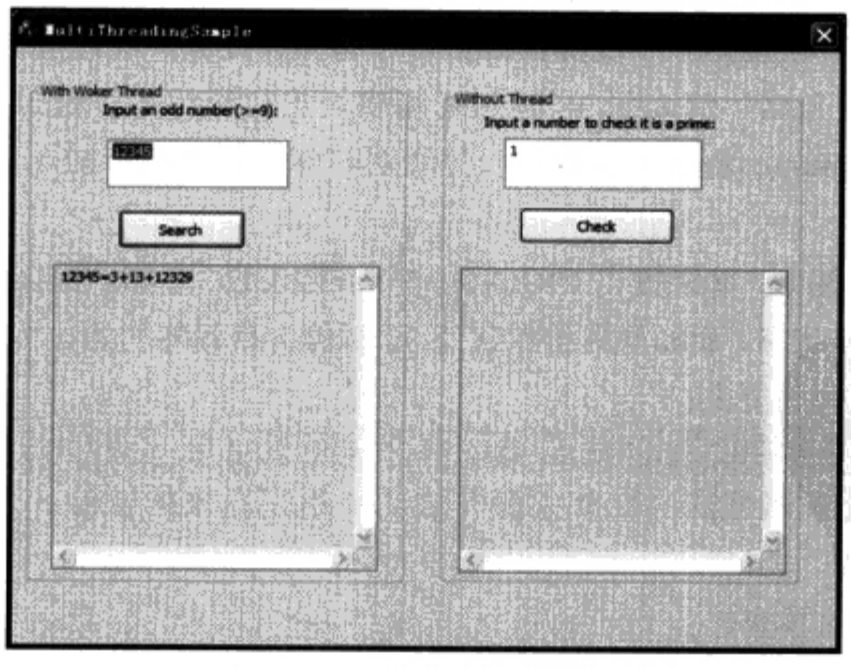


图 14.1　使用多线程程序的运行结果

这里将对话框分为两个部分，左半部分用来搜索哥德巴赫猜想之一：任意一个不小于 9 的奇数，都可以表示成 3 个奇素数之和。左半部分的上面的文本框接受用户输入的一个不小于 9 的奇数，然后搜索到 3 个奇素数，它们的和就是用户输入的值。比如用户输入 12345，则程序会找到结果：

12345=3+13+12329。这是一个计算密集型的过程，很适合放到后台去执行，所以这里将创建一个工作者线程来负责这个计算过程，找到结果后，再将结果通过消息的形式发送给主窗口显示。右半部分只是判断用户输入的一个整数是否为素数，操作比较简单，所以在主线程中直接计算。将它们放到一起是为了显示多线程的优点：可以在左边工作者线程搜索结果的时候，继续进行右边的操作，而不必长时间等待计算结果。

首先，建立一个基于对话框的工程，在 Resource View 下设置如图 14.1 所示的布局。下面简单介绍这些控件的作用。

左半部分的上面的那个文本框（Edit Control）接受用户输入一个整数值，先在对话框头文件内定义一个 public 的 unsigned 型的成员变量 m_iOddNum 负责接受用户的输入，再在 DoDataExchange()函数内实现数据传输。

右半部分上面的那个文本框以同样的方式加入一个变量 m_iCheckNum。

下面的两个编辑框只用于显示操作的结果，设置其 Read Only 属性为 True。另外，把新的结果拼接到原来结果的后面，所以输出文本会有很多行，需要设置其滚动条属性为 True，这样程序将会自动在内容超出时，使用滚动条来查看前后结果，如图 14.2 所示。

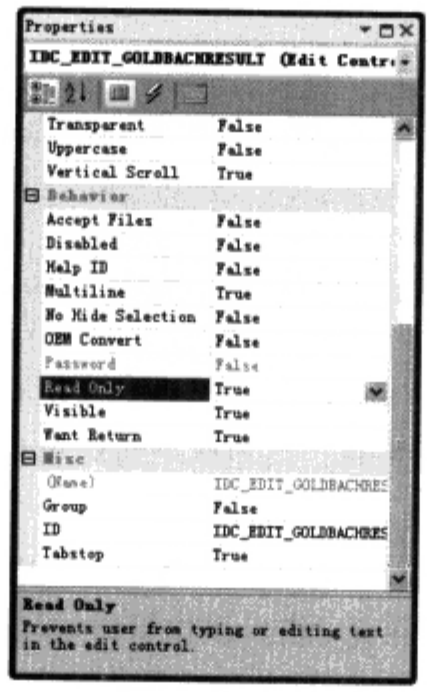


图 14.2　设置文本框控件为只读

对话框类的主要成员定义如下：

```
public:
    unsigned m_iOddNum;
    afx_msg void OnBnClickedButtonCalgoldbach();
    CEdit m_GoldbatchResultCtrl;
    unsigned m_iCheckNum;
    CEdit m_CheckPrimeCtrl;
    afx_msg void OnBnClickedButtonCheckisprime();
    CEdit m_CheckPrimeResultCtrl;
```

其中成员 m_iOddNum 存放大于 9 的奇数，而成员 m_iCheckNum 代表待检测的素数，其他几个为对应的文本框变量。

构造函数及对话框 DoDataExchange()函数的代码如下：

```
CMultiThreadingSampleDlg::CMultiThreadingSampleDlg(CWnd* pParent /*=NULL*/)
    :m_iOddNum(9), m_iCheckNum(1), m_iWokerNum(0), CDialog(CMultiThreadingSampleDlg::IDD, pParent)
{
```

```
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
void CMultiThreadingSampleDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Text(pDX,IDC_EDIT_ODDNUM,m_iOddNum);
    DDX_Text(pDX,IDC_EDIT_CHECKISPRIME,m_iCheckNum);
    DDX_Control(pDX, IDC_EDIT_GOLDBACHRESULT, m_GoldbatchResultCtrl);
    DDX_Control(pDX, IDC_EDIT_CHECKISPRIME, m_CheckPrimeCtrl);
    DDX_Control(pDX, IDC_EDIT_CHECKPRIMERESULT, m_CheckPrimeResultCtrl);
    DDX_Text(pDX,IDC_EDIT_WOKERNUM,m_iWokerNum);
}
```

消息映射关系如下：

```
BEGIN_MESSAGE_MAP(CMultiThreadingSampleDlg, CDialog)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    //}}AFX_MSG_MAP
    ON_BN_CLICKED(IDC_BUTTON_CALGOLDBACH,&CMultiThreadingSampleDlg::OnBnClickedButtonCalgoldbach)
    //
    ON_MESSAGE (WM_USER_THREAD_FINISHED, &CMultiThreadingSampleDlg::OnThreadFinished)
    ON_BN_CLICKED(IDC_BUTTON_CHECKISPRIME,&CMultiThreadingSampleDlg::OnBnClickedButtonCheckisprime)
END_MESSAGE_MAP()
```

其中，ON_MESSAGE 为开发人员自定义的一个消息 WM_USER_THREAD_FINISHED，用于接收工作者线程发回的搜索结果，将在后面解释；其他两个为对应两个按钮“Search”和“Check”的单击事件处理函数。

为按钮添加事件处理函数的过程如下：首先右击按钮，比如“Search”按钮，在弹出的菜单中选择“Add Event Handler”命令，如图 14.3 所示，进入添加事件处理向导。

这里选择向导提供的默认 BN__CLICKED 单击事件处理，单击“Add and Edit”按钮，如图 14.4 所示，就完成了添加事件处理的过程，将进入这个事件处理函数，后面将在这里添加代码。

下面定义工作者线程的控制函数，先定义几个辅助结构体和函数，在对话框类的头文件中定义如下：

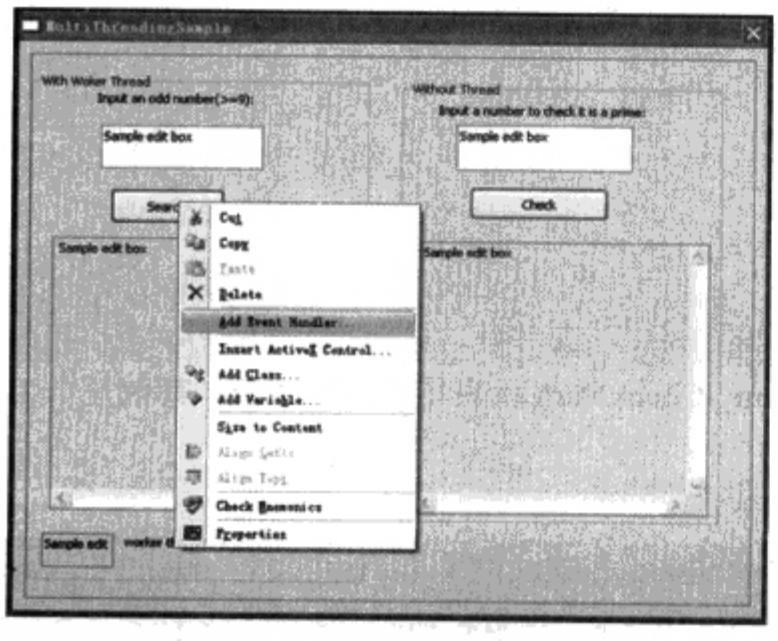


图 14.3　为按钮控件添加事件处理函数

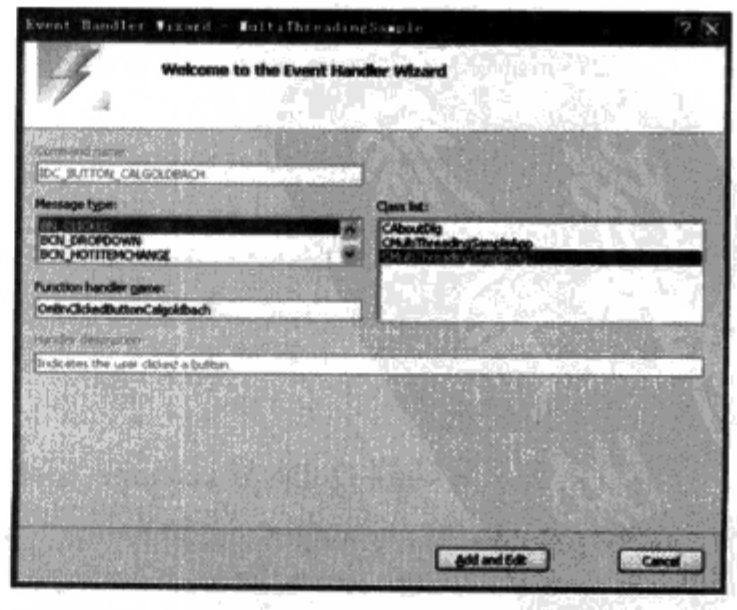


图 14.4　添加事件处理函数向导

```
        //代表哥德巴赫猜想的找到的结果，oddNum 为输入值，p1、p2、p3 代表找到的结果，为 3 个奇素数
struct Goldbach_Result{
    unsigned oddNum;
    unsigned   p1;
    unsigned   p2;
    unsigned   p3;
};
        //这个函数判断输入的整数是否为素数
BOOL IsPrime(unsigned   n);
        //搜索奇素数组合，并将结果存放到 pResult 内
void GetGoldbachResult(unsigned oddNum,Goldbach_Result*pResult);

        //由于最后要将结果发送给主窗口，所以需要将窗口句柄传入控制函数
struct ThreadParam{
    unsigned oddNum;
    HWND hWnd;//主窗口句柄
};
        //工作者线程的控制函数
UINT ThreadFunc (LPVOID pParam);
```

IsPrime()函数用于判断输入的整数是否为素数，算法比较简单：首先判断是否为 1、2 这样的特殊数，如果是则作特殊处理；然后判断是否为偶数，如果是偶数，就不可能是素数；接着试探从 3 到 $\sqrt{n}+1$ 之间的奇数，如果都不能被这些数整除，就是素数，否则就不是素数。代码实现如例 14.1 所示。

例 14.1

```
BOOL IsPrime(unsigned   n){
    if( n == 1 ){
      return FALSE;
    }
    if( n == 2 ){
      return TRUE;
    }
                    //如果被 2 整除，肯定不是素数
    if( n % 2 == 0 ){
      return FALSE;
    }
    unsigned   num_test_end = (unsigned )sqrt((double)(n+1));
    unsigned   i;
    for( i = 3; i <= num_test_end; i += 2 ){
      if( n % i ==0 ){
          return FALSE;
      }
    }
    return TRUE;
}
```

而 GetGoldbachResult()函数首先找到所有小于输入值的奇素数，并将它们分成 3 个子类：小于输入值的 1/3、在 1/3 和 2/3 之间、大于 2/3。这样可以缩小搜索范围，比如不可能有超过两个的素数在大于 2/3 的集合内，至少有一个值在小于 1/3 的集合内等。对于一种特殊情况：如果输入值被 3 整除，而且它的 1/3 正好是素数，则找到结果，返回即可。代码如例 14.2 所示。

例 14.2

```
void GetGoldbachResult(unsigned oddNum,Goldbach_Result*pResult){
    pResult->oddNum = oddNum;
                    //首先找到所有比它小的素数，并分成 3 个子类：小于它的 1/3、在 1/3 和 2/3 之间，大于 2/3
    CArray<unsigned,unsigned&>littlePrime;
    CArray<unsigned,unsigned&>middlePrime;
    CArray<unsigned,unsigned&>largePrime;
    unsigned testNum = 3;
    while( testNum <= oddNum/3 ){
      if( IsPrime(testNum) ){
            littlePrime.Add(testNum);
      }
      testNum += 2;
    }
    while( testNum <= 2*oddNum/3 ){
      if( IsPrime(testNum) ){
            middlePrime.Add(testNum);
      }
      testNum += 2;
    }
    while( testNum <=   oddNum ){
      if( IsPrime(testNum) ){
            largePrime.Add(testNum);
      }
      testNum += 2;
    }
                    //如果这个数被 3 整除，且它的 1/3 为一个素数，则找到结果
if( oddNum % 3 == 0 && IsPrime(oddNum/3) ){
    pResult->p1 = oddNum/3;
    pResult->p2 = oddNum/3;
    pResult->p3 = oddNum/3;
    return;
}
                    //在 3 个子集合内适当查找，一旦找到一个结果就返回
int i,j,k;
for( i=0;i<littlePrime.GetSize();i++ ){
   for( j=0; j<littlePrime.GetSize();j++){
          for(k=0;k<middlePrime.GetSize();k++){
                if( oddNum == littlePrime.GetAt(i)+littlePrime.GetAt(j)+
                              middlePrime.GetAt(k) ){
                      pResult->p1 = littlePrime.GetAt(i);
                      pResult->p2 = littlePrime.GetAt(j);
                      pResult->p3 = middlePrime.GetAt(k);
                      return;
                }
          }
          for(k=0;k<largePrime.GetSize();k++){
                if( oddNum == littlePrime.GetAt(i)+littlePrime.GetAt(j)
                              +largePrime.GetAt(k) ){
                      pResult->p1 = littlePrime.GetAt(i);
```

```
                    pResult->p2 = littlePrime.GetAt(j);
                    pResult->p3 = largePrime.GetAt(k);
                    return;
                }
            }
        }
        for(j=0;j<middlePrime.GetSize();j++){
            for(k=0;k<middlePrime.GetSize();k++){
                if( oddNum == littlePrime.GetAt(i)+middlePrime.GetAt(j)
                        +middlePrime.GetAt(k) ){
                    pResult->p1 = littlePrime.GetAt(i);
                    pResult->p2 = middlePrime.GetAt(j);
                    pResult->p3 = middlePrime.GetAt(k);
                    return;
                }
            }
            for(k=0;k<largePrime.GetSize();k++){
                if( oddNum == littlePrime.GetAt(i)+middlePrime.GetAt(j)
                        +largePrime.GetAt(k) ){
                    pResult->p1 = littlePrime.GetAt(i);
                    pResult->p2 = middlePrime.GetAt(j);
                    pResult->p3 = largePrime.GetAt(k);
                    return;
                }
            }
        }
    }
}
```

工作者线程的控制函数 ThreadFunc()的代码如例 14.3 所示。

例 14.3

```
UINT ThreadFunc (LPVOID pParam){
                        //传入结果体 ThreadParam：包含待搜索的值 oddNum 和主窗口句柄，用于后面发回结果
    ThreadParam*ptp = (ThreadParam*)pParam;
    unsigned oddNum = ptp->oddNum;
    HWND hWnd = ptp->hWnd;
    delete ptp;
    Goldbach_Result* presult = new Goldbach_Result;
                        //调用搜索过程获得结果后，通过消息发回给主窗口
    GetGoldbachResult(oddNum,presult);
    ::PostMessageW(hWnd,WM_USER_THREAD_FINISHED,(WPARAM)presult,
        NULL);
    return 0;
}
```

这里通过调用::PostMessageW()函数向主窗口以消息的形式发送回结果，其中消息 WM_USER_THREAD_FINISHED 是自定义消息，代表工作者线程计算完毕，在对话框头文件中定义如下：

```
#define WM_USER_THREAD_FINISHED WM_USER+0x400
```

定义对话框的消息处理函数为 OnThreadFinished()，注意消息处理函数的返回值必须是 LRESULT 类型，并且传入两个参数 wParam 和 lParam，这个消息处理函数定义如下：

```
afx_msg LRESULT    OnThreadFinished(WPARAM wParam, LPARAM lParam);
```

通过 ON_MESSAGE()宏来建立消息映射：

```
ON_MESSAGE (WM_USER_THREAD_FINISHED,
        &CMultiThreadingSampleDlg::OnThreadFinished)
```

消息处理函数 OnThreadFinished()的代码如例 14.4 所示。

例 14.4

```
LRESULT    CMultiThreadingSampleDlg::OnThreadFinished(WPARAM wParam,
        LPARAM lParam){
                                        //获得工作者线程发送过来的结果
    Goldbach_Result*pResult = (Goldbach_Result*)wParam;
                                        //获得消息对话框的原来输出，将新的结果拼接到后面
    CStringW outputText;
    m_GoldbatchResultCtrl.GetWindowTextW(outputText);
    CStringW gotresult;
    gotresult.Format(_T("%u=%u+%u+%u\r\n"),pResult->oddNum,
            pResult->p1,pResult->p2,pResult->p3 );
    outputText += gotresult;
                                        //显示新的结果
    m_GoldbatchResultCtrl.SetWindowTextW(outputText);
    delete pResult;
    this->GetDlgItem(IDC_BUTTON_CALGOLDBACH)->EnableWindow(TRUE);
    return 0;
}
```

这里的主要操作就是从发送过来的消息参数中获得计算结果，并拼接到 Edit 控件的输出文本上显示。

用户在左侧的文本框（Edit 控件）中输入一个整数后，单击“Search”按钮就可以进行搜索计算。“Search”按钮的单击事件处理函数代码如例 14.5 所示：

例 14.5

```
void CMultiThreadingSampleDlg::OnBnClickedButtonCalgoldbach()
{
                                        // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    CStringW outputText;
    m_GoldbatchResultCtrl.GetWindowTextW(outputText);
                                        //输入值不合法
    if( m_iOddNum % 2 == 0 || m_iOddNum < 9 ){
        outputText = outputText + _T("Invalid input number: it must be odd and >=9") + _T("\r\n");
        m_GoldbatchResultCtrl.SetWindowTextW(outputText);
        return;
    }
    this->GetDlgItem(IDC_BUTTON_CALGOLDBACH)->EnableWindow(FALSE);
                                        //创建控制函数的参数，包括输入值和主窗口句柄
    ThreadParam*ptp = new ThreadParam;
    ptp->oddNum = m_iOddNum;
    ptp->hWnd = this->m_hWnd;
                                        //创建工作者线程搜索结果
    AfxBeginThread(ThreadFunc,ptp);
}
```

这里，首先从控件获得输入值，并检测输入值是否合法（不小于 9 的奇数），如果不合法，则输出提示信息，马上返回；否则就会创建一个工作者线程进行搜索计算。

对于右半部分的判断一个数是否为素数的“Check”按钮，由于计算过程比较简单，直接在事件处理函数中计算并立即显示结果。“Check”按钮的单击事件处理函数代码如例 14.6 所示。

例 14.6

```
void CMultiThreadingSampleDlg::OnBnClickedButtonCheckisprime()
{
                                                    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    CStringW outputCheckPrime;
    m_CheckPrimeResultCtrl.GetWindowTextW(outputCheckPrime);
                                                    //直接调用 IsPrime()函数判断
    if( IsPrime(m_iCheckNum) ){
        CStringW appended;
        appended.Format(_T("%u is a prime number\r\n"),m_iCheckNum);
        outputCheckPrime += appended;
        m_CheckPrimeResultCtrl.SetWindowTextW(outputCheckPrime);
    }
    else{
        CStringW appended;
        appended.Format(_T("%u is not a prime number\r\n"),m_iCheckNum);
        outputCheckPrime += appended;
        m_CheckPrimeResultCtrl.SetWindowTextW(outputCheckPrime);
    }
}
```

至此，这个程序的设计过程就介绍完了，下面编译并运行程序。如果在左半部分输入一个很大的奇数（比如 8 位数），单击“Search”按钮，将进行比较长时间的计算。因为这里是创建一个工作者线程放到后台去计算，所以并不会影响右边判断是否为素数的计算。而如果没有使用多线程，那么程序只能停留在密集的计算中而无法再去响应用户的其他操作，这就是使用多线程的优点。

程序的运行结果如图 14.5 所示。

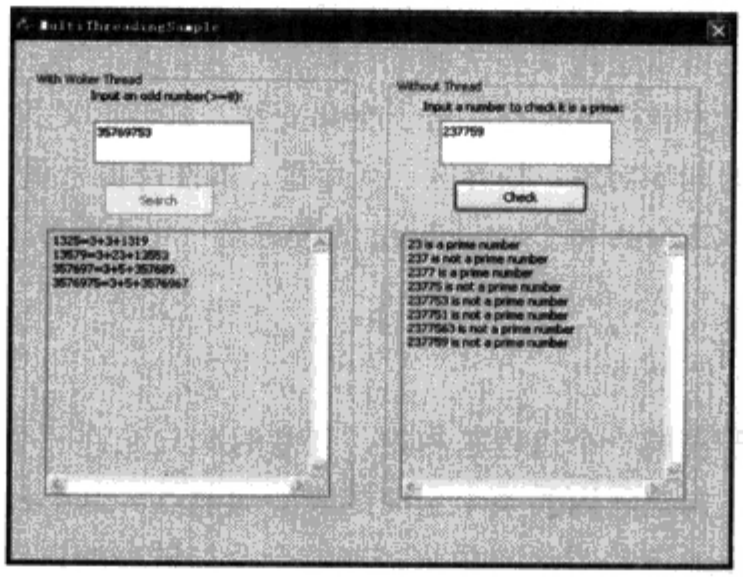


图 14.5　程序运行结果

14.6　线程间的通信

实际上，在上面的例子中已经涉及了线程的通信：在创建一个新的工作者线程时，父线程通过传

递控制函数的参数就实现了线程的通信。但是这个太被动了，只能限于在父线程创建子线程时向子线程传递数据。

另外，在上面的例子中，为了向主窗口线程传回计算结果，使用了发送消息的方式，主线程再在消息处理函数内处理工作者线程发回的结果。由于主线程本质上是一个用户界面线程，所以向一个运行中的用户界面线程传递数据也可以通过发送消息的方式实现：在用户界面线程内自定义消息，用于接收其他线程发送的不同消息；只要其他线程能够获得这个用户界面线程的句柄，就可以通过::PostMessageW()函数向它传递数据。

和一个运行中的工作者线程通信的一个最简单的办法就是使用全局变量或静态变量。因为一个进程内的所有线程共享同一块地址空间，所以进程内的全局变量对所有线程都是可见的，通过修改这些全局变量的值可以影响运行中的工作者线程的行为。比如，一个线程想在适当的时候终止另一个工作者线程，那么可以定义一个 BOOL 型的全局变量 bActive，在工作者线程启动前将其设置为 True。并在工作者线程内监测这个值，一旦这个值为 False，工作者线程就会终止，控制函数代码如下：

```
UNIT ThreadFunc(LPVOID pParam){
    while(bActive){
                                    //仍然活动，继续工作
    }
                                    //一旦全局标志被设置为 False，线程就终止
    return 0;
}
```

这样在适当的时候，只要其他线程将这个全局标志设置为 False，就可以终止那个工作者线程。由于进程内的所有线程都可以访问全局变量，所以要注意对数据的同步访问。例如，有些数据在任意时刻只允许一个线程进行写操作，而其他线程必须等待，这就得通过线程的同步来实现。

14.7 线程的同步

要进行多线程程序的设计，就要涉及令人头疼的同步问题。为了完成一个任务，线程的执行不能随意进行，而必须加以同步控制，否则很容易出错。同步可以保证在同一时间内只有一个或有限个线程可以访问系统资源，同步还可以使得某些关键的代码按一定的顺序执行。比如，只有在某个线程计算出中间结果后，另一个线程才能往下执行；对于全局变量，任意时刻只允许一个线程写等。

Windows 支持 4 种类型的同步对象：临界区、互斥量、事件及信号量。这些对象分别被 MFC 的名为 CcriticalSection、Cmutex、CEvent 和 Csemaphore 的类封装。下面分别介绍这些同步对象。

14.7.1 临界区

临界区的主要用途就在于在同一个进程范围内，任意时刻只允许一个线程访问一个资源、修改数据或进入某块代码段。比如，向进程内的某个全局链表添加节点，这个过程每次只能允许一个线程进行。通过创建一个 CCriticalSection 临界区对象来控制这个全局链表的访问，可以有效保证任意时刻至多一个线程访问链表。

临界区对象主要通过在访问之前锁定资源来实现线程的独占访问，而在访问结束之后释放锁，以允许其他线程访问。如果一个线程试图锁定一个已经被锁住的资源，那么这个线程就会被阻塞，直到该临界区对象空闲为止。

CCriticalSection 类提供了 Lock()成员函数来锁定临界区对象，UnLock()成员函数来释放对临界区

资源的锁定。其中 Lock()函数提供了两种不同的重载形式：

```
BOOL Lock( );
BOOL Lock(
    DWORD dwTimeout
);
```

参数 dwTimeout 用于指定超时时间。

在以前的 MFC 版本中，如果线程被阻塞的时间超过超时时间，而临界区还没有空闲，那么 Lock()将返回，并说明锁定出错。但是在最新的 MFC 版本中，这个参数已经被忽略，也就是说，这两种形式的 Lock()调用都将使线程一直阻塞直到临界区空闲，以后都使用第一种重载形式。

UnLock()函数的原型是：

```
BOOL Unlock( );
```

如果调用成功返回 True，否则返回 False。

下面的一段代码简单地演示了如何使用临界区对象：

```
                                        //首先在进程内定义一个全局临界区对象
CCriticalSection   critSect;
…
                                        //在希望独占访问某个资源之前调用 Lock()函数
critSect.Lock();
                                        //对资源进行操作
…
critSect.UnLock();//访问结束，释放锁
```

14.7.2　互斥量

互斥量也是用来获得对共享资源的独占性访问，但是互斥量除了能同步同一个进程内的线程外，也能同步不同进程中的线程；而临界区只能用于同一进程内线程的同步。临界区的速度优于互斥量，当访问速度很关键且资源访问不需要跨越进程边界时（在同一个进程内同步），优先使用临界区。

同样 CMutex 类也提供了 Lock()函数来锁定，UnLock()函数来解锁。

Lock()函数的原型是：

```
virtual BOOL Lock(
        DWORD dwTimeout = INFINITE
);
```

参数 dwTimeout 设置超时时间是有效的，如果线程阻塞时间超过超时时间，Lock()函数就会返回，并返回失败状态。默认值 INFINITE 表示线程将一直阻塞，直到互斥量对象空闲为止。

互斥量的使用方法与临界区类似，这里就不再赘述。

14.7.3　事件

事件对象主要用于一个线程通知另一个线程某个特定事件已经发生，事件在通知线程何时执行特定的任务很有用处。比如，一个线程需要从某个数据源获取数据，那么在新数据到达时，应该通过事件来通知这个线程准备接收数据。

MFC 的 CEvent 类封装事件，每个 CEvent 对象只能处于两种不同的状态：活动（或信号）状态和非活动（或非信号）状态。通过调用 CEvent 类的 SetEvent()函数来激发事件并将释放所有等待此事件的线程，而 ResetEvent()函数则将事件重新设置为非活动状态。这两个函数原型如下：

```
BOOL SetEvent( );
BOOL ResetEvent( );
```

一个线程调用 CEvent 对象的 Lock()函数而在事件对象上阻塞，等待事件变为活动状态时才会返回。而其他线程如果调用该事件对象的 SetEvent()函数，就会触发该事件，那么所有在此事件上阻塞的线程都能被唤醒，因此事件也常被称为“线程触发器”。

CEvent 类支持两种不同的事件类型：人工重置事件和自动重置事件。这两者的区别是：人工重置事件必须通过调用 SetEvent()函数和 ResetEvent()函数来显式地设置事件的状态；而自动重置事件在事件被触发后，如果至少有一个线程因此而被唤醒，那么该事件对象将自动回到非信号状态。

CEvnet 类的构造函数接受 4 个参数，如下：

```
CEvent(
        BOOL bInitiallyOwn = FALSE,
        BOOL bManualReset = FALSE,
        LPCTSTR lpszName = NULL,
        LPSECURITY_ATTRIBUTES lpsaAttribute = NULL
);
```

- 第一个参数 bInitiallyOwn 指定事件对象初始化为信号状态（True）还是非信号状态（False）。
- 第二个参数 bManualReset 指定事件类型为人工重置（True）还是自动重置（False）。
- 第三个参数 lpszName 指定事件对象的名称，对于不同进程间的线程同步，必须给事件指定一个名称；如果为同一进程内的线程同步，则使用匿名事件（NULL）。
- 第四个参数 lpsaAttribute 描述事件对象的安全属性，默认值 NULL 适用于大部分运用。

下面的例子演示了怎样使用事件对象来同步线程：一个事件表明缓冲区有数据到达，那么线程 1 就通过设置事件为活动状态来通知线程 2 准备接收数据。

```
//定义全局的事件对象
CEvent gEvent;//采用默认值，自动重置，非信号状态
        …
//线程 1
//在缓冲区数据到达之后，设置激活事件，唤醒线程 2
gEvent.SetEvent();
…
//线程 2
gEvent.Lock();//阻塞，等待事件被激活
//下面就可以处理缓冲区内的数据
```

在上面的例子中，使用自动重置事件，这样如果线程 2 被唤醒，那么事件将自动回到非信号状态，线程 2 在处理完数据后，又将进入等待状态，直到下一批数据到达。

14.7.4 信号量

信号量对象对线程的同步提供了更大的灵活性，信号量对于控制共享资源只能支持有限个线程同时访问时特别有用。信号量对象内部维护共享资源的当前可允许访问的线程个数。当这个数为 0 时，以后所有对这个共享资源的访问请求都将被阻塞，放到系统的等待队列内，直到正在访问的线程超时或者可用数大于 0 为止。锁定信号量将减少可用数，释放信号量将增加可用数。

CSemaphore 类的构造函数如下：

```
CSemaphore(
    LONG lInitialCount = 1,
    LONG lMaxCount = 1,
```

```
    LPCTSTR pstrName = NULL,
    LPSECURITY_ATTRIBUTES lpsaAttributes = NULL
);
```

- 第一个参数 lInitialCount 指定初始化的可用资源数，必须不小于 0 且不大于 lMaxCount，一般设置为 lMaxCount 值。
- 第二个参数 lMaxCount 指定信号的最大可用数，必须大于 0。
- 第三个参数 pstrName 指定信号量的名称，对于不同进程间的线程同步，必须给事件指定一个名称；如果为同一进程内的线程同步，则使用匿名事件（NULL）。
- 第四个参数 lpsaAttribute 描述事件对象的安全属性，默认值 NULL 适用于大部分运用。

信号量的使用相当简单：初始化时指定可用资源数，以后各线程在访问之前都调用 Lock()函数，如果当前可用数大于 0，则进入访问；如果这个数不大于 0，则阻塞。访问完毕后，调用 UnLock()释放即可，下面的简单代码演示了信号量的使用方法：

```
                                    //信号量的当前可用数和最大可用数都是 4
CSemaphore gSemaphore(4,4);
                                    //某个线程欲访问信号量
gSemaphore.Lock();//如果当前可用数大于 0，则进入；否则阻塞
…
gSemaphore.UnLock();
```

除了直接使用这些同步对象的 Lock()及 UnLock()函数来实现线程的同步外，MFC 还提供了两个类来实现更加安全的同步：CSingleLock 类和 CMultiLock 类。当只需要等待一个同步对象时，使用 CSingleLock 对象；如果需要同时等待多个同步对象，则使用 CMultiLock 对象。

这两个类通过封装同步对象，来避免对同步对象的直接锁定和解锁操作，从而提高代码的安全性；因为如果直接操作同步对象，而在中间发生异常，那么由于解锁函数没有得到执行，将导致这些同步对象被一直锁定，得不到释放。而通过这两个对象的锁定和解锁就可以避免这种情况：如果异常发生，可以有效控制同步对象的状态，并自动释放这些同步对象。

CSingleLock 类的构造函数为：

```
explicit CSingleLock(
        CSyncObject* pObject,
        BOOL bInitialLock = FALSE
);
```

- 参数 pObject 指定封装的同步对象的指针。
- 参数 bInitialLock 指定初始化同步对象为锁定（True）状态还是非锁定（False）状态。

创建完 CSingleLock 对象后，就可以调用 Lock()函数锁定，UnLock()函数解锁。下面的例子演示了如何使用 CSingleLock 对象来进行安全的锁定和解锁：

```
                                    //初始化为非锁定状态
CSingleLock singleLock(&m_CritSection);
                                    //锁定同步对象
singleLock.Lock();
if (singleLock.IsLocked())   // 检测是否锁定成功
{
                                    //执行操作
                                    //访问结束后，对同步对象解锁
    singleLock.Unlock();
}
```

最后，扩展前面介绍过的例子，加入同步对象控制来允许启动多个工作者线程分别对多个不同的

奇数并行进行查询。在对话框中添加一个控件来显示当前的工作者线程的数目，这里设置最大的数目为 8 个。程序运行界面如图 14.6 所示。

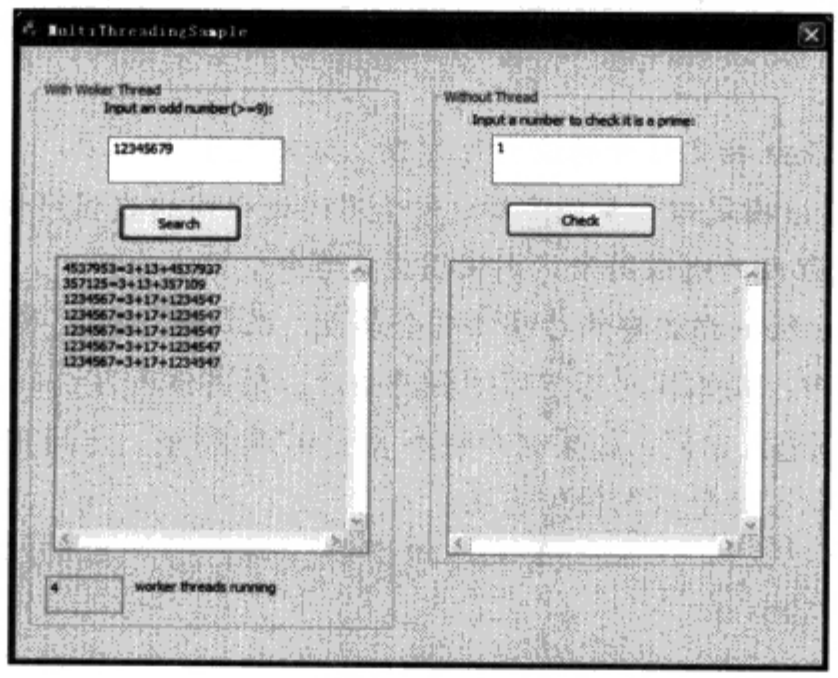


图 14.6　加入线程同步后程序的运行结果

将代表工作者线程的数目的变量作为一个全局变量，这样在主线程启动一个新的工作者线程时，将这个值加 1；而当工作者线程计算完毕，在线程终止前将这个值减 1。这个全局变量需要同步访问管理，为此加入一个临界区对象来同步线程对这个变量的访问。在对话框的头文件内定义如下全局变量：

```
extern int g_iWorkerNum;
extern CCriticalSection g_critSect;
```

这两个全局变量的定义和初始化放在对话框类的实现文件中：

```
//当前的工作者线程数目
int g_iWorkerNum = 0;
//临界区对象，同步线程对上面变量的访问
CCriticalSection g_critSect;
```

在类中定义一个成员变量来存放工作者线程的个数，并将个数传输到控件中显示：

```
public:
    int m_iWokerNum;
```

在 DoDataExchange()函数内实现数据传输：

```
DDX_Text(pDX,IDC_EDIT_WOKERNUM,m_iWokerNum);
```

在 OnInitDialog()函数内初始化控件显示的值，代码如下：

```
CSingleLock singleLock(&g_critSect);
    singleLock.Lock();
    if( singleLock.IsLocked() ){
        m_iWokerNum = g_iWorkerNum;
        singleLock.Unlock();
}
```

这里使用 CSingleLock 对象来封装临界区对象，在访问之前加锁，访问完毕后解锁。

对“Search”按钮的单击事件处理函数需要做相应改变，结果如例 14.7 所示。

例 14.7

```
void CMultiThreadingSampleDlg::OnBnClickedButtonCalgoldbach()
{
    // TODO: Add your control notification handler code here
```

```
    UpdateData(TRUE);
    CStringW outputText;
    m_GoldbatchResultCtrl.GetWindowTextW(outputText);
                                                        //输入值不合法
    if( m_iOddNum % 2 == 0 || m_iOddNum < 9 ){
        outputText = outputText + _T("Invalid input number: it must be odd and >=9") + _T("\r\n");
        m_GoldbatchResultCtrl.SetWindowTextW(outputText);
        return;
    }
    CSingleLock singleLock(&g_critSect);
    singleLock.Lock();
    if( singleLock.IsLocked() ){
                                                        //更新工作者线程数目
        g_iWorkerNum++;
        m_iWokerNum = g_iWorkerNum;
                                                        //最多允许 8 个工作者线程
        if(g_iWorkerNum>=8){
            this->GetDlgItem(IDC_BUTTON_CALGOLDBACH)->
                EnableWindow(FALSE);
        }
        singleLock.Unlock();
    }
    UpdateData(FALSE);
                                                        //创建控制函数的参数，包括输入值和主窗口句柄
    ThreadParam*ptp = new ThreadParam;
    ptp->oddNum = m_iOddNum;
    ptp->hWnd = this->m_hWnd;
                                                        //创建工作者线程搜索结果
    AfxBeginThread(ThreadFunc,ptp);
}
```

这里主要就是加入同步控制，在启动一个新的工作者线程之前，将数目加 1。如果当前工作者线程数目达到 8 个，将禁止再响应，然后将新的数目传输到控件上显示。

工作者线程的控制函数除了负责计算结果以外，还要在自己终止之前将表示工作者线程数目的全局变量减 1，这需要加入临界区同步控制，控制函数的代码如例 14.8 所示。

例 14.8

```
UINT ThreadFunc (LPVOID pParam){
                                    //传入结果体 ThreadParam：包含待搜索的值 oddNum 和主窗口句柄。用于后面发回结果
    ThreadParam*ptp = (ThreadParam*)pParam;
    unsigned oddNum = ptp->oddNum;
    HWND hWnd = ptp->hWnd;
    delete ptp;
    Goldbach_Result* presult = new Goldbach_Result;
                                    //调用搜索过程获得结果后，通过消息发回给主窗口
    GetGoldbachResult(oddNum,presult);
    CSingleLock singleLock(&g_critSect);
    singleLock.Lock();
    if( singleLock.IsLocked() ){
                                    //工作完毕，工作者线程个数减 1
```

```
            g_iWorkerNum--;
            singleLock.Unlock();
        }
        ::PostMessageW(hWnd,WM_USER_THREAD_FINISHED,(WPARAM)presult,
            NULL);
        return 0;
}
```

对应的 WM_USER_THREAD_FINISHED 消息的处理函数如例 14.9 所示。

例 14.9

```
LRESULT   CMultiThreadingSampleDlg::OnThreadFinished(WPARAM wParam,
        LPARAM lParam){
                                        //获得工作者线程发送过来的结果
    Goldbach_Result*pResult = (Goldbach_Result*)wParam;
                                        //获得消息对话框的原来输出，将新的结果拼接到后面
    CStringW outputText;
    m_GoldbatchResultCtrl.GetWindowTextW(outputText);
    CStringW gotresult;
    gotresult.Format(_T("%u=%u+%u+%u\r\n"),pResult->oddNum,
        pResult->p1,pResult->p2,pResult->p3 );
    outputText += gotresult;
                                        //显示新的结果
    m_GoldbatchResultCtrl.SetWindowTextW(outputText);
    delete pResult;
    CSingleLock singleLock(&g_critSect);
    singleLock.Lock();
    if( singleLock.IsLocked() ){
        if(!this->GetDlgItem(IDC_BUTTON_CALGOLDBACH)->IsWindowEnabled() &&
        g_iWorkerNum < 8){
            this->GetDlgItem(IDC_BUTTON_CALGOLDBACH)->
                EnableWindow(TRUE);
        }
        m_iWokerNum = g_iWorkerNum;
        singleLock.Unlock();
    }
    UpdateData(FALSE);

    return 0;
}
```

这里首先查询工作者线程数目的新值，如果“Search”按钮被禁止，且当前工作者线程数目小于 8，则激活“Search”按钮，然后将新的工作者线程数目更新到控件上显示。

在这个实例程序中，只加入了一个很简单的同步控制，但是在线程数目逐渐增多时，还是会明显感觉到主窗口的响应速度降低，部分原因是线程多了，分配到的 CPU 时间片减少；更主要的原因还是在主线程与工作者线程之间的同步操作，主线程可能会因为资源被锁住而阻塞，导致性能下降。

在实际的多线程程序设计中，要特别谨慎地设计线程间的同步操作，否则很容易导致性能下降，更严重的后果是出现死锁。死锁问题是一个相当复杂的课题，有兴趣的读者可以进一步查阅相关资料进行学习。

14.8　小结

本章首先介绍了如何在 MFC 程序中创建新线程，包括工作者线程和用户界面线程，以及如何终止线程，在线程终止后如何查看线程的退出码；接着介绍了如何查询线程的优先级，设置线程的优先级，以及线程优先级的含义；然后介绍如何暂停一个线程，以及如何使线程重新运行，还简单地介绍了一下如何实现线程之间的通信；最后介绍了令人头痛的线程同步问题，MFC 提供了 4 种不同的同步对象——临界区、互斥量、事件和信号量，本章分别介绍了它们的基本用法，还简单介绍了利用 CSingleLock 及 CMultiLock 对象来封装以实现更安全的线程同步。

第 15 章　Visual C++ 2010 MFC 网络程序设计

近年来，因特网发展十分迅速，越来越多的应用程序能够通过网络（包括因特网）与其他应用程序进行通信。网络程序的实现可以有多种方式，如 Windows API 提供的 Socket、MFC 提供的对 Windows Sockets 的面向对象封装等。可以说，Socket 是连接应用程序和网络的桥梁。通过 Socket 提供的接口应用程序进行通信根本不必考虑底层的网络结构，只需要简单地将数据交给 Socket 或从 Socket 获取数据，Socket 使网络编程变得更加直观容易。

本章将介绍关于网络的一些基本知识、TCP/IP 协议、Winsock 等概念，以及如何通过 Winsock 接口在两个或多个计算机之间进行网络通信。

15.1　计算机网络的基础知识

计算机网络是利用通信设备和线路将地理位置不同的、功能独立的多个计算机系统连接起来，以功能完善的网络软件实现网络的硬件、软件及资源共享和信息传递的系统。简而言之，计算机网络就是自主计算机的互联集合，计算机网络的核心就是通信和资源共享。

要想同连在网络上的计算机进行通信，首先得知道它在网络上的标识，对 TCP/IP 协议而言，就是 IP 地址。有了 IP 地址，路由器才能寻址定位到目标机器，并将信息通过线路传输过去。如今的操作系统都支持多进程，即使消息到达目标机器，还得指定发送给哪个应用程序（进程），这就需要指定一个端口号。通信时还需要一个大家都必须遵守的规则，这就是通信协议，目前最常用的就是 TCP/IP 协议。

15.1.1　TCP/IP 协议模型

如果读者学习过有关计算机网络的课程，一定对于“学院派”的 OSI（Open System Interconnection）国际标准的七层模型比较熟悉，但是该标准太过繁琐复杂，在实际运用中并没有被推广，因此这里并准备介绍这一经典模型。而 TCP/IP 至今仍是国际上通用的工业标准，因此我们把注意力放到实用的 TCP/IP 协议上。

TCP/IP 协议可以说是对 OSI 模型的简化，它只有 4 层，其结构如图 15.1 所示。其中每一层的功能如下。

- 网络接口层：一般包括操作系统中的设备驱动程序和计算机中对应的网络接口卡，它们一起处理与通信电路的物理接口细节，是数据包从一个设备的网络层传输到另外一个设备的网络层的方法。发送的一端将完成如添加包头准备发送，通过物理媒介实际发送这样一些数据链路功能；另一端将完成数据帧的接收并将包头去掉，然后将接收到的包向上传输到网络层。该层包含的协议有：ARP（地址转换协议）和 RARP（反向地址转换协议）。

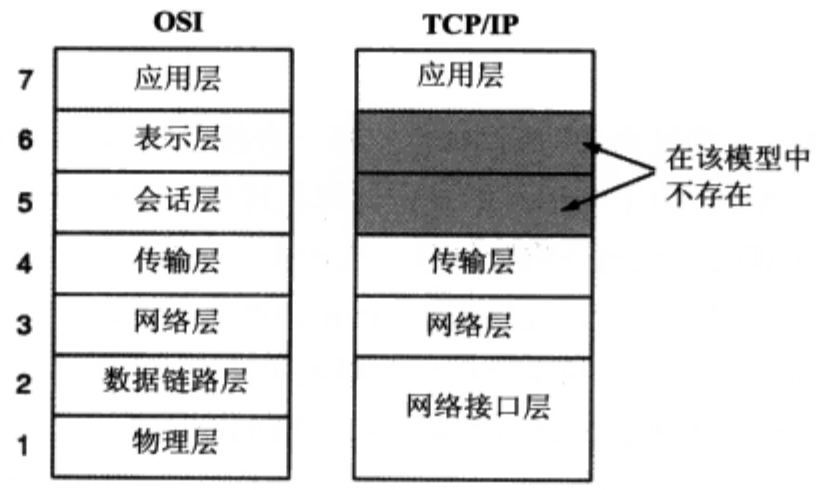


图 15.1　OSI 模型与 TCP/IP 模型各层的关系

- 网络层：也称为网络互联层，负责数据在网络中的活动，解决在一个单一的网络上传输数据包的问题，包括 IP（网际协议）、ICMP（互联网控制报文协议）以及 IGMP（Internet 组管理协议）等协议。其中 IP 完成数据从源发送到目的地的基本任务，它能够承载多种不同的高层协议的数据，这些高层协议使用一个惟一的 IP 协议号进行标识。ICMP 和 IGMP 分别是 1 和 2。
- 传输层：传输层是一个软件抽象，主要为两台主机上的应用程序提供端到端的数据通信，能够解决诸如可靠性和保证数据按照正确的顺序到达这样的问题。在 TCP/IP 协议族中，传输协议也包括所给数据应该发送给哪个应用程序（进程），而不仅仅只是指定发送给哪台主机。这一个层提供了面向连接的 TCP（传输控制协议）和无连接的 UDP（用户数据报协议）。

TCP（IP 协议号为 6）提供一个可靠的面向连接的传输机制，它提供一种可靠的字节流来保证数据完整、无重复，并且严格按照发送的顺序到达，是端到端质量保证的数据传输。该层负责数据的分组、质量控制和超时重发等，对于应用层就不必顾虑这些网络是否不可靠，它会向网络层提供一个可靠传输的抽象。

UDP（IP 协议号为 17）是一个无连接的数据报协议，它只是简单的把数据报从一端发送到另一端，没有任何质量保证。对于数据的超时、丢失、重复、乱序到达等问题都得由应用层来处理，是不可靠传输，但是其好处就是比 TCP 要快，所以在一些对时间要求较高而对准确度要求不那么严格的情况下，如网络流媒体应用，使用 UDP 更适合。

- 应用层：该层负责处理实际的应用程序细节，包括所有和应用程序协同工作，利用网络基础交换应用程序专用数据的协议。应用层是大多数与网络相关的普通程序为了通过网络与其他程序通信所使用的层。这个层的处理过程是应用特有的：数据从与网络相关的程序以这种应用内部使用的格式进行传送，然后被编码成标准协议的格式。该层包含的协议有 HTTP(The World Wide Web）、FTP（数据传输协议）、SMTP（简单邮件传输协议）、SSH（安全远程登录协议）和 DNS（域名解析）等著名协议。

这些层以及它们所包含的典型协议如图 15.2 所示。

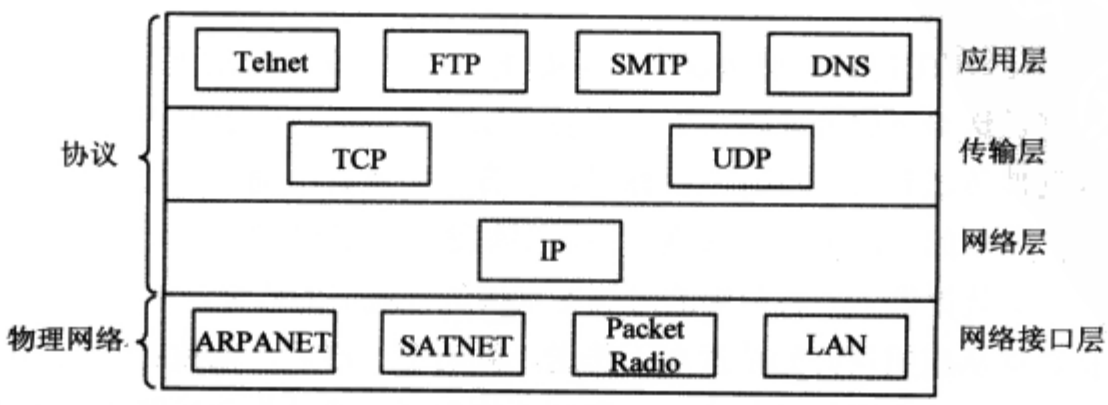


图 15.2　各层包含的典型协议

15.1.2 IP 地址

在 TCP/IP 协议栈中，编址由 IP 协议规定。每一台接入网络的计算机都必须分配一个惟一的“身份标识”，这就是 IP 地址。为了与远程的某台计算机通信，必须指定目标机器的 IP 地址，这样路由器就能根据复杂的路由算法逐级向外将数据包发送出去，直至到达目标机器。目前 IPv6 大有替代 IPv4 之势，并且在一些网络应用上已经开发出基于 IPv6 的产品，可以说 IPv6 是未来的大势所趋，但是这仍需要一个较漫长的推广过程。目前主流用户使用的仍是 IPv4 地址格式，这是一个 32 位的二进制格式，4 字节，一般采用点分十进制的格式表示，每字节的取值范围为 0～255，如 192.168.1.100。

过去 IANA（Internet Assigned Numbers Authority，Internet 号码分配局）按照不同的国家和地区将 IP 地址分为 A、B、C、D 四类，并将 32 位的 IP 地址分为两个部分：网络（外网）地址和局域网（内网）地址。例如，在 C 类网络中，前 24 位为网络地址，后 8 位为局域网地址，可提供 254 个设备地址（其中 255 代表广播地址，0 代表网络本身，这两个值不能为网络设备所用）。它使用网络掩码的方式限制了网络的范围，网络掩码使用 1 代表网络部分，使用 0 代表设备地址部分，如 C 类地址最常用的网络掩码是 255.255.255.0。

IPv6 的地址是一个 128 位的二进制数，一般表示成 32 个十六进制数，写作 8 组，每组 4 个十六进制数，如 2001:0db8:85a3:08d3:1319:8a2e:0370:7344。IPv6 的地址由两个逻辑部分组成：一个 64 位的网络前缀和一个 64 位的主机地址，主机地址通常根据物理地址自动生成，叫做 EUI-64（64-位扩展惟一标识）。

15.1.3 端口

端口是一个提供服务的进程的标识，引入端口后，允许同一个 IP 地址的机器同时提供多种不同的服务。在客户端/服务器模型中，客户端一般无须自己指定端口号，而由系统自动分配一个可用的端口号，而且每次客户端都可能被分配一个不同的端口号，因此客户端口号又称做临时端口号。对于固定的服务器，需要用户指定一个固定的端口号，这样有利于客户端顺利连接服务器，与服务器通信。对 TCP/IP 实现来说，一些基本的网络服务都分配了固定的端口号，如 FTP 服务器为 21，Telnet 服务器为 23 等。

服务器的端口号是一个 16 位的非负整数，范围为 0～65535。0～1023 这段范围被分配给任何 TCP/IP 实现所提供的服务，大多数的 TCP/IP 实现给临时端口分配 1024～5000 之间的端口号，大于 5000 的端口号为其他并不常用的服务器保留。在编写网络应用程序时，要为程序（特别是服务器端）指定一个 1024 以上的端口号。

15.1.4 数据封装

在前面介绍的 OSI 和 TCP/IP 分层协议中，其实每一层都只跟自己的下一层通信，下层为上层提供服务。在上层将数据发送给下层处理时，下层都会在原始数据的基础上加上一些特殊的包头，然后再往下发送。而当数据被传送到目标层时，目标层将检查包头是否正确，去掉包头后再将数据传送给上层，也就是说实际上两端的上层根本不知道也不需要知道数据曾经被封装过，传输两端的对等层接收到的数据是一致的，这就提供了一种两端的对等层直接进行通信的假象。

发送端进程将数据从应用层开始依次向下传送，到达下一层后，就被封装上相应的包头，作为新的数据到达再下一层，如此不断封装包头，直到到达物理层。通过物理线路传输到达目的端，此时数

据是从下往上传送，每到一层，就会将对应的包头去除，最后到达的就是发送进程发送的原始数据。TCP/IP 协议的 4 层数据传输过程与此类似，也是一个发送端从上往下传输，封装包头；接收端从下往上传输，去除包头的过程。

15.2　Winsock 简介

Winsock 是一套开放的、支持多种协议的 Windows 网络编程接口，是 Windows 网络编程事实上的标准。应用程序并不是直接通过网络进行通信，而是通过 Socket（套接字）接口进行的，所以说 Socket（套接字）是网络通信的基石。

套接字最初是在美国加州大学伯克利分校在 UNIX 系统下开发出来的一种应用程序访问通信协议的操作系统调用套接字，使用套接字可以很方便地访问 TCP/IP，从而开发出网络应用程序。套接字存在于通信域中，通信域是为了处理一般的线程通过套接字通信而引进的一种抽象概念。套接字通常只能和同一个域中的套接字进行通信。

根据通信性质可以将套接字分为两类：流套接字和数据报套接字。流套接字提供双向的、有序的、无重复的数据流服务，提供面向连接的可靠服务。数据报套接字也支持双向的数据流，但是不保证数据传输的可靠性、有序性和无重复，是面向无连接的。

使用 Windows Sockets 可以保证应用程序可以在任何支持 Windows Sockets API 的网络内正常通信，在 Win32 内，Windows Sockets 还提供了线程安全性。

使用 Socket 进行网络编程最常用的模型就是客户端/服务器模型，服务器提供一定的服务，客户端向服务器发送服务请求，服务器接受请求并作相应的处理，再将结果发送给客户端。该通信过程如图 15.3 所示。

一个服务在一个特定的地址和端口监听客户端的请求，此时服务器一直处于休眠状态，直到客户端的请求到达，将触发服务器对请求做出响应。基于连接的服务是设计客户端/服务器模型应用程序的标准，但这也可以通过非连接的数据报套接字来提供。

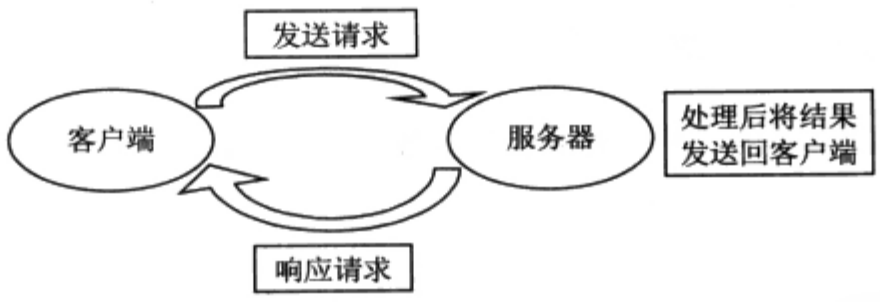


图 15.3　客户端/服务器通信模型

Windows Sockets 是 Windows 网络设计的接口，是从伯克利的 Socket 扩展而来的，继承了它的主要特征。Windows Sockets 以动态连接库的形式提供编程 API，这个 API 提供了大量的函数用于建立网络程序。但是直接使用这些 API 比较麻烦，幸好 MFC 对这些 API 进行了面向对象的封装，简化了程序员的编程工作，所以下面只介绍 MFC 对 Windows Sockets 的支持。

15.3　MFC 对 Windows Sockets 的支持

MFC 内提供了两个类——CAsyncSocket 和 CSocket 来支持封装 Windows Sockets API 编程，其中

CAsyncSocket 是对 Windows Sockets API 较低层次的封装，而 CSocket 类是直接从 CAsyncSocket 派生出来的，提供较高层次的封装，通过对 Socket 对象的串行化读/写封装来传输数据（后面将介绍），简化程序员的编程工作。这两个类为将 Windows Sockets 结合 MFC 的应用程序提供了面向对象的支持。

15.3.1 Socket 的定义

一个 Socket 是一个通信终端。通过 Socket 对象来实现基于 Windows Sockets 的应用程序间的网络通信（发送或接收数据），就是说应用程序（或进程）不是直接通过网络通信，而必须以 Socket 对象为通信的接口：进程首先将数据交给 Socket 对象，Socket 对象通过网络将数据发送到接收端的 Socket 对象，接收端的 Socket 对象再将数据交给接收进程处理，如图 15.4 所示。

图 15.4 利用 Socket 对象实现进程间通信

一个 Socket 对象必须附属到一个运行中的进程，目前 Socket 对象只能同使用相同通信域的其他 Socket 对象通信。可以使用两种不同类型的 Socket 对象进行通信：流式 Socket 和数据报 Socket。这两种类型 Socket 的区别如下。

- 流式 Socket 提供面向连接、可靠的数据传输服务，基于 TCP 实现，以数据流的方式保证数据严格按发送顺序接收，且数据无差错和无重复地发送。
- 数据报 Socket 提供无连接的数据传输服务，不能保证数据按发送的顺序接收，数据可能丢失或重复，基于 UDP 实现。

这两种类型的 Socket 都支持数据的双向传输，即允许一个 Socket 对象发送和接收数据。

每个 MFC 的 Socket 对象内都含有一个指向 Windows Sockets 对象的句柄，该句柄的具体类型是 SOCKET。Socket 对象可以调用 Create()成员函数自己创建一个 SOCKET 句柄，然后依附到 Socket 对象，也可以调用 Attach()函数指定将一个已经存在的 SOCKET 句柄依附进来。

15.3.2 CAsyncSocket 类介绍

下面介绍 CAsyncSocket 类的一些主要的成员函数，这些函数同样也是 CSocket 类的主要操作。

1．构造函数

```
CAsyncSocket( );
```

该函数不接收任何参数，创建一个“空”的 Socket 对象。在构造一个 Socket 对象之后，必须调用 Create()函数来创建附属的 SOCKET 句柄并绑定 IP 地址，之后 Socket 对象才能用于通信。

2．Create()成员函数

```
BOOL Create(
    UINT nSocketPort = 0,
    int nSocketType = SOCK_STREAM,
    long lEvent = FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT| FD_CONNECT | FD_CLOSE,
    LPCTSTR lpszSocketAddress = NULL
);
```

该函数首先根据参数创建一个 SOCKET 句柄并附属到 Socket 对象内。如果调用成功则返回非 0（True），否则返回 0（False），可调用 GetLastError()函数来查看失败的原因。该函数接收 4 个参数，

各个参数的含义如下。

- 第一个参数 nSocketPort 指定 Socket 对象的端口号，默认值 0 表示由 Socket 自动选择一个端口号。对于服务器端的 Socket 对象，必须指定一个明确的端口号；而对于客户端的 Socket 对象，使用默认值 0 即可。
- 第二个参数 nSocketType 指定创建的 Socket 对象的类型，如上所述，其有两种不同类型：默认值 SOCK_STREAM 代表流式 Socket；SOCK_DGRAM 代表数据报 Socket。
- 第三个参数 lEvent 代表应用程序感兴趣的网络事件的掩码，各个值之间使用“或”（|）连接表示同时“关注”多个事件，可取得事件类型值如表 15.1 所示。

表 15.1　　网络事件类型

事 件 类 型	说　　明
FD_READ	在准备好读数据时得到通知
FD_WRITE	在准备好写数据时得到通知
FD_OOB	在包含边界的数据到达时得到通知
FD_CONNECT	在连接请求到达时得到通知
FD_CONNECT	在连接建立完成后得到通知
FD_CLOSE	在 Socket 对象关闭时得到通知

- 第四个参数 lpszSocketAddress 指定 Socket 对象绑定的 IP 地址，采用点分的格式，如“128.56.22.18”，一般使用默认值 NULL，由系统自动关联本机的 IP 地址。

3. GetSockName()成员函数

该函数用于返回 Socket 对象的本地名称。其有两种不同的重载形式，第一种原型为：

```
BOOL GetSockName(
    CString& rSocketAddress,
    UINT& rSocketPort
);
```

其用于取回 Scoket 对象绑定的 IP 地址（或机器域名）和端口号。

- 参数 rSocketAddress 接收点分十进制格式的 IP 地址或机器域名。
- 参数 rSocketPort 接收端口号。

第二种重载原型为：

```
BOOL GetSockName(
    SOCKADDR* lpSockAddr,
    int* lpSockAddrLen
);
```

其用于取回 Socket 对象的本地地址。

- 参数 lpSockAddr 为指向一个 SOCKADDR 结构体的指针，用于存放结果。
- 参数 lpSockAddrLen 返回 SOCKADDR 结构体内的地址长度，单位为字节。

而 SOCKADDR 结构体的定义如下：

```
struct sockaddr {
    unsigned short sa_family;
    char sa_data[14];
};
```

该结构类型有两个数据成员，第一个数据成员 sa_family 表示套接字使用的地址族；第二个数据成员 sa_data 表示所有不同的套接字地址结构的最大字节数。

为了支持 IPv6 格式地址的运用，该类还提供了一个 GetSockNameEx()函数，函数原型如下：

```
BOOL GetSockNameEx(
    CString& rSocketAddress,
    UINT& rSocketPort
);
```

参数含义和上面的一致。

4. GetPeerName()成员函数

该函数用于返回该 Socket 对象连接的对等 Socket 对象的信息，参数含义与上面的相同：

```
BOOL GetPeerName(
    CString& rPeerAddress,
    UINT& rPeerPort
);
BOOL GetPeerName(
    SOCKADDR* lpSockAddr,
    int* lpSockAddrLen
);
```

同样，GetPeerNameEx()函数支持 IPv6 格式的地址。其原型如下：

```
BOOL GetPeerNameEx(
    CString& rPeerAddress,
    UINT& rPeerPort
);
```

5. Listen()成员函数

该函数用于监听客户端发送过来的连接请求，一般只有流式（面向连接）服务器端 Socket 对象才需要调用该函数。其原型如下：

```
BOOL Listen(
    int nConnectionBacklog = 5
);
```

参数 nConnectionBacklog 指定等待连接队列的最大长度，合法值只能取[1,5]。

该函数只由那些可以同时支持多个连接请求的流式服务器端 Socket 对象调用，当一个连接请求在等待连接队列已满的情况下到达时，那么对应的客户端 Socket 对象会接收到一个标识为 WSAECONNREFUSED 的错误信息。注意：这里所指的等待连接队列长度并不是指服务器端 Socket 对象所能支持的连接个数。如果将该值设置为 1，那么表示一次只能处理一个连接请求，只有在当前没有其他连接请求时，此连接请求才能被服务器接受（调用 Accept()函数），但是服务器却是支持多个客户端连接的。

6. Accept()成员函数

当等待连接队列中存在连接请求时，服务器端的 Socket 对象调用 Accept()函数来接受该连接请求。服务器端将创建一个新的 Socket 对象来负责与对应的客户端 Socket 对象通信，而服务器端的 Socket 对象继续回到监听连接请求的状态。也就是说，实际上真正与客户端 Socket 对象通信的并不是服务器端的 Socket 对象，而是这个新创建的“临时”Socket 对象。Accept()函数的原型如下：

```
virtual BOOL Accept(
    CAsyncSocket& rConnectedSocket,
    SOCKADDR* lpSockAddr = NULL,
    int* lpSockAddrLen = NULL
);
```

- 参数 rConnectedSocket 表示负责与客户端通信的 Socket 对象。在调用 Accept()函数之前，要先创建一个“空白”的 Socket 对象，就是说此时不需要调用 Create()函数来创建附属的 SOCKET 句柄，因为 Accept()函数会自动为这个新的 Socket 对象创建附属的 SOCKET 句柄。
- 参数 lpSockAddr 是一个返回值，保存客户端 Socket 对象的地址信息。这是一个指向 SOCKADDR 结构（参见 GetSockName()函数内的说明）的指针，如果没有任何信息，该值将为 NULL。
- 参数 lpSockAddrLen 返回 SOCKADDR 结构体内的地址长度，单位为字节。

前面介绍过，一个服务器端的 Socket 对象可以同时处理多个连接请求。当等待连接队列内存在多个连接请求时，它将处理挂在队列上的第一个连接请求，并创建一个与这个 Socket 对象属性完全一致的 SOCKET 句柄附属到这个新的 Socket 对象内。这个函数只能由流式（面向连接）的服务器端 Socket 对象调用。

7．Connect()函数

该函数用于向未连接的流式或数据报 Socket 对象发送连接请求，并建立连接。连接的目标可以是流式的服务器端 Socket 对象，此时将建立一个真正的连接，一旦函数调用成功，就可以进行数据传输；也可以是数据报 Socket 对象，此时将建立一个默认的目标（可认为是一个虚拟连接）。这样做的好处是后面发送数据时可以不必每次都指定目标 Socket 对象的地址信息，但是它实质上还是数据报，是不可靠的服务，不能保证数据接收严格按发送的顺序，不能避免数据的丢失和重复。

该函数有两种不同的重载形式，一种是通过目标的网络地址和端口号来建立连接：

```
BOOL Connect(
    LPCTSTR lpszHostAddress,
    UINT nHostPort
);
```

- 参数 lpszHostAddress 表示目标 Socket 对象的地址，可以是目标机器的域名或 IP 地址。
- 参数 nHostPort 指定目标 Socket 对象的端口号。

另一种是通过 SOCKADDR 结构表示的地址来建立连接：

```
BOOL Connect(
    const SOCKADDR* lpSockAddr,
    int nSockAddrLen
);
```

- 参数 lpSockAddr 为指向目标 Socket 地址信息的 SOCKADDR 结构的指针。
- 参数 nSockAddrLen 指定地址的长度，单位为字节。

8．Send()成员函数

Send 函数既用于在面向连接的 Socket 对象之间发送数据，也可用于在前面调用过 Connect()函数建立虚拟连接的数据报 Socket 对象间发送数据。如果发送数据成功，则返回此次发送的数据的字节数。函数原型为：

```
virtual int Send(
    const void* lpBuf,
    int nBufLen,
    int nFlags = 0
);
```

- 参数 lpBuf 表示一个指向待发送数据缓冲区的指针。
- 参数 nBufLen 指定发送缓冲内的数据长度，单位为字节。
- 参数 nFlags 影响该函数调用的方式，函数的行为由 Socket 对象本身的选项（Options）和这个

标志共同决定。可以取下面两个不同的值。

（1）MSG_DONTROUTE——指定数据不提交给路由选择。这个参数可能会被忽略。

（2）MSG_OOB——发送带外数据，只对流式 Socket 有意义。

这里参数 nBufLen 指定发送缓冲区的长度，但实际传输的数据长度并不会与这个值相等，要通过检测返回值，才能确定这次发送的数据量。

对于数据报 Socket 对象而言，即使 Send()函数调用成功，也不能保证这个数据会被成功接收。对于流式 Socket 对象而言，可以写入的数据长度由两端的缓冲区大小共同决定，避免出现发送的数据长度超过接收端的接收缓冲区的长度。

9. SendTo()成员函数

该函数用于向一个特定的目标发送数据，由于有两种不同的指定目标的方式：地址+端口号和 SOCKADDR，所以这个函数提供了两种不同的重载形式：

```
int SendTo(
    const void* lpBuf,
    int nBufLen,
    UINT nHostPort,
    LPCTSTR lpszHostAddress = NULL,
    int nFlags = 0
);
int SendTo(
    const void* lpBuf,
    int nBufLen,
    const SOCKADDR* lpSockAddr,
    int nSockAddrLen,
    int nFlags = 0
);
```

如果函数调用成功，则返回发送。数据的字节数，否则返回 SOCKET_ERROR。

- 参数 lpBuf 为指向发送缓冲区的指针。
- 参数 nBufLen 指定此次发送数据的大小，单位为字节。
- 参数 nFlags 指定函数的调用方式，参见 Send()函数的说明。
- 另外两个参数用来指定目标机器的地址，参见 GetSockName()函数的说明。

这个函数既可用于数据报（非连接）Socket 对象，当然这也是最主要的用途；也可用于流式（面向连接）Socket 对象，这很少见，但也可以用，此时将提供不可靠的服务，即使发送成功也不能保证数据能够被成功接收。对于数据报的发送，一定要注意每次发送的数据量不能超过系统支持的最大长度，否则将出错。

10. SendToEx()成员函数

该函数主要用于支持在使用 IPv6 地址的机器（Socket 对象）之间发送数据，当然也兼容 IPv4 版本。这个函数的原型如下：

```
int SendToEx(
    const void* lpBuf,
    int nBufLen,
    UINT nHostPort,
    LPCTSTR lpszHostAddress = NULL,
    int nFlags = 0
);
```

对于 IPv6，只能通过主机地址和端口号访问。其参数含义与返回值与 SendTo()函数一致。如果希望编写支持 IPv6 的应用程序，可使用该函数。

11. Receive()成员函数

该函数与 Send()对应，用于接收对方 Socket 对象发送过来的数据，既可用于面向连接的流式 Socket 对象，此时服务是可靠的；也可用于调用过 Connect()函数建立一个虚拟连接的数据报 Socket 对象，此时服务是不可靠的。函数原型如下：

```
virtual int Receive(
    void* lpBuf,
    int nBufLen,
    int nFlags = 0
);
```

- 参数 lpBuf 为指向接收输入数据的缓冲区的指针。
- 参数 nBufLen 为接收缓冲区的大小，单位为字节，注意一定要保证发送数据的大小不超过接收缓冲区的大小。
- 参数 nFlags 指定函数调用的方式，参见 Send()函数的参数说明。

如果函数调用成功（即成功接收数据），返回值为此次接收到的数据的字节数；如果连接已经被关闭，则返回值为 0；否则返回值为 SOCKET_ERROR，表示接收出错，可调用 GetLastError()函数来查看失败原因。

12. ReceiveFrom()成员函数

该函数与 SendTo()对应，用于接收非连接的 Socket 对象发送过来的数据，并能获知发送方的地址信息。由于有两种表示地址的方式，所以这个函数对应有两种重载形式：

```
int ReceiveFrom(
    void* lpBuf,
    int nBufLen,
    CString& rSocketAddress,
    UINT& rSocketPort,
    int nFlags = 0
);
int ReceiveFrom(
    void* lpBuf,
    int nBufLen,
    SOCKADDR* lpSockAddr,
    int* lpSockAddrLen,
    int nFlags = 0
);
```

- 参数 lpBuf 为指向接收缓冲区的指针。
- 参数 nBufLen 为接收缓冲区的大小（字节数）。
- 参数 nFlags 指定函数调用的方式。
- 另外两个参数用于存储发送方的地址：rSocketAddress 为地址和端口号，rSocketPort 将地址信息存储在 SOCKADDR 结构体内。

如果接收数据成功，则返回值为接收的数据的字节数；如果连接已经关闭，则返回值为 0；如果出错，则返回 SOCKET_ERROR。

13. ReceiveFromEx()函数

与 SendToEx()函数对应，支持 IPv6 地址格式 Socket 对象的数据接收。其原型如下：

```
int ReceiveFromEx(
      void* lpBuf,
      int nBufLen,
      CString& rSocketAddress,
      UINT& rSocketPort,
      int nFlags = 0
);
```

各个参数含义与第一个 ReceiveFrom()函数相同。

同样，接收成功的返回值为接收到的数据字节数；如果连接已关闭，则返回值为 0；否则返回 SOCKET_ERROR，表示接收出错。

14. ShutDown()成员函数

该函数用来关闭 Socket 对象的发送、接收数据的功能或将两者全部关闭。函数原型如下：

```
BOOL ShutDown(
      int nHow = sends
);
```

参数 nHow 指定关闭的功能，可取 3 个不同的值，如表 15.2 所示。

表 15.2　　可关闭的功能

功　能	说　明
Receives=0	禁止 Socket 对象执行接收数据操作
sends=1	禁止 Socket 对象执行发送数据操作
both=2	这两个操作都被禁止

如果调用成功，则返回 True（非 0）；否则返回 False（0）。

这里调用 ShutDown()函数只是禁止了 Socket 对象的功能，而并没有将 Socket 关闭，该 Socket 对象占用的资源并不会被释放，必须调用 Close()函数才能关闭 Socket 对象并将分配给它的资源释放。

一旦对一个 Socket 对象调用 ShutDown()函数禁止某些功能，那么后面将无法再重用这些功能，也就是说一旦禁止就不能再用了。此时，对这些 Socket 对象建立连接可能不会被支持，也就是说这些受限的 Socket 对象将可能无法建立连接。

15. Close()成员函数

其功能是关闭一个 Socket 对象，并释放资源。

```
virtual void Close( );
```

CAsyncSocket 类的析构函数将自动调用这个函数，所以用户不必显式调用；如果用户想复用同一个 Socket 对象则要先调用 Close()函数，再调用 Create()函数创建新的附属 SOCKET 句柄。

16. 消息重载函数

还有几个比较特殊的函数，称为消息重载函数，这些函数包括：OnAccept()、OnClose()、OnConnect()、OnReceive()、OnSend()和 OnOutOfBandData()。这些函数主要用于在 Socket 对象检测的事件发生时，由主框架向应用程序发出通知，这样用户就可以对这些事件进行处理。如果用户希望在事件发生时进行一些特殊的操作，可以派生出一个新的针对特定应用的子类，然后重载这些函数，就可以改变它们的行为。

（1）OnAccept()成员函数

当连接请求达到时，主框架将通知调用 Listen()函数对连接请求进行侦听的 Socket 对象可以对这个请求进行处理。函数原型为：

```
virtual void OnAccept(
    int nErrorCode
);
```

参数 nErrorCode 返回 Socket 对象最近的错误，只有两个可能的值：如果值为 0 表示这个函数成功执行；如果值为 WSAENETDOWN，表示检测到网络子系统发生错误。

（2）OnClose()函数

主框架调用这个函数通知 Socket 对象，与它连接的对等 Socket 对象已经被它所属的进程关闭。函数原型为：

```
virtual void OnClose(
    int nErrorCode
);
```

这里参数 nErrorCode 返回 Socket 对象最近的错误，可以取 4 个不同的值，如表 15.3 所示。

表 15.3　　可能返回的错误码

错　误　码	说　　明
0	表示这个函数成功执行
WSAENETDOWN	检测到网络子系统出现错误
WSAECONNRESET	这个连接已经被远端重置
WSAECONNABORTED	这个连接由于超时或其他原因已经被取消

（3）OnConnect()成员函数

主框架通知 Socket 对象，本次连接请求的尝试已经完毕，结果可能成功也可能失败。函数原型是：

```
virtual void OnConnect(
    int nErrorCode
);
```

参数 nErrorCode 同样返回最近的错误，0 表示函数成功执行，其他值代表出现错误，请参考 MSDN 查询详细情况。

（4）OnOutOfBandData()成员函数

主框架调用这个函数通知接收数据的 Socket 对象，发送端的带外数据将被发送过来。函数原型是：

```
virtual void OnOutOfBandData(
    int nErrorCode
);
```

参数 nErrorCode 代表最近的错误，只取两个值：0 表示这个函数成功执行；WSAENETDOWN 表示检测到网络系统发生错误。

（5）OnReceive()成员函数

主框架调用这个函数通知 Socket 对象，接收缓冲区内已经有数据到达，可以调用 Receive()函数接收数据。函数原型是：

```
virtual void OnReceive(
    int nErrorCode
);
```

参数 nErrorCode 代表最近的错误，只取两个值：0 表示这个函数成功执行；WSAENETDOWN 表示检测到网络系统发生错误。

（6）OnSend()成员函数

主框架通知 Socket 对象可以调用 Send()函数发送数据。函数原型是：

```
virtual void OnSend(
    int nErrorCode
);
```

参数 nErrorCode 代表最近的错误，只取两个值：0 表示这个函数成功执行；WSAENETDOWN 表示检测到网络系统发生错误。

15.3.3 CSocket 类介绍

下面介绍 CSocket 类的一些重要的成员函数。CSocket 类直接派生自 CAsyncSocket 类，所以上面的成员函数在 CSocket 类中都是可用的。CSocket 类是对 Windows Sockets API 的高层包装，主要用于支持利用文件 CSocketFile 方式和串行化 CArchive 支持来方便地管理数据的发送和接收。

还有一个区别就是 CSocket 对象提供阻塞支持，用于同步 CArchive 串行化对象的操作，而 CAsyncSocket 对象提供的操作默认都是非阻塞的。阻塞函数 Receive()、Send()、ReceiveFrom()、SendTo()、和 Accept()（这些都是继承自 CAsyncSocket 类）不会返回 WSAEWOULDBLOCK 错误，而是会一直等待直到操作完成。另外，通过调用 CancelBlockingCall()函数可以取消正在等待的函数，并返回错误 WSAEINTR，提供了更大的灵活性。

1. 构造函数 CSocket()

该函数用于创建一个空白的 CSocket 对象。函数原型为：

```
CSocket( );
```

在创建完一个 CSocket 对象后，必须调用 Create()函数来创建附属的 SOCKET 句柄，设置 CSocket 对象的类型、地址等属性。

2. Create()成员函数

CSocket 类对这个函数进行了重载，函数原型变为：

```
BOOL Create(
    UINT nSocketPort = 0,
    int nSocketType = SOCK_STREAM,
    LPCTSTR lpszSocketAddress = NULL
);
```

- 参数 nSocketPort 指定 Socket 对象绑定的端口号。
- 参数 nSocketType 指定 Socket 对象的类型，默认值 SOCK_STREAM 代表流式 Socket，SOCK_DGRAM 代表数据报 Socket。
- 参数 lpszSocketAddress 指定绑定的地址，一般使用默认值 NULL，由系统自动指定合适的地址。

3. IsBlocking()成员函数

该函数用于查询 Socket 对象是否正在处理执行阻塞调用。函数原型如下：

```
BOOL IsBlocking( );
```

如果 Socket 对象正在被阻塞，返回 True，否则返回 False。

4. CancelBlockingCall()成员函数

该函数用于取消 Socket 对象正在执行的阻塞调用。函数原型为：

```
void CancelBlockingCall( );
```

执行完该函数后，原来的阻塞调用会立即被终止，并返回错误码 WSAEINTR（可通过 GetLastError()函数查看）。如果阻塞是由 Connect()引起，虽然会立即终止阻塞调用，但是除非连接完成或超时，否则该 Socket 对象的资源可能无法被释放。

15.4　一个基于 UDP 的聊天室示例

下面利用 CSocket 类实现一个基于 UDP 无连接的聊天室程序。服务器维护一个当前在聊天室内的所有客户的地址列表，在有消息到达时，服务器根据该列表将消息依次发送给这些客户端。在用户进入聊天室时，客户端会先向服务器发送一个特殊的消息，这里就采用字符串“enter”，这样服务器就将这个客户端的 IP 地址和端口号添加进列表；在客户退出聊天室时，也发送一个特殊的消息，这里使用“byebye”，这样服务器会将该客户的地址从通信列表内删除。因为是基于数据报的 Socket 对象进行通信，所以用到的发送和接收函数分别是 SendTo()和 ReceiveFrom()。操作的基本流程如下。

1．客户端操作的基本流程

（1）创建一个空白的 CSocket 类或 CSocket 派生类的对象。

（2）调用 Socket 对象的 Create()函数创建附属的 SOCKET 句柄，这里使用 SOCK_DGRAM 表示使用数据报类型。

（3）调用 SendTo()函数发送一个特殊的消息，通知服务器该客户端进入聊天室。

（4）调用 SendTo()、ReceiveFrom()函数向服务器发送数据及从服务器接收数据。

（5）在客户端退出之前，发送一个特殊消息，通知服务器该客户端退出聊天室。

2．服务器端操作的基本流程

（1）创建一个空白的 CSocket 类或 CSocket 派生类的对象。

（2）调用 Create()函数创建附属的 SOCKET 句柄，这里使用 SOCK_DGRAM 表示使用数据报类型。

（3）调用 ReceiveFrom()函数接收客户端发送的数据。

（4）调用 SendTo()函数将数据广播给所有客户端。

15.4.1　MFC 对 Windows Sockets 的初始化

这里使用 MFC 的对话框应用程序，在 MFC 创建对话框应用程序向导的第 4 步 Advanced Features 中启用 Windows Sockets 支持，这样向导会自动将支持 Windows Sockets 编程的必要头文件和库包含进来，如图 15.5 所示。

向导自动生成的 stdafx.h 文件内包含了 MFC Socket 支持的头文件 afxsock.h：

```
#include <afxsock.h>                    // MFC socket extensions
```

在应用程序类的 InitInstance()函数内，向导自动调用了 AfxScoketInit()函数对 Windows Sockets 进行初始化。在使用 MFC 提供的 Socket 类之前，必须调用该函数进行初始化。代码如下：

```
if (!AfxSocketInit())
    {
        AfxMessageBox(IDP_SOCKETS_INIT_FAILED);
        return FALSE;
    }
```

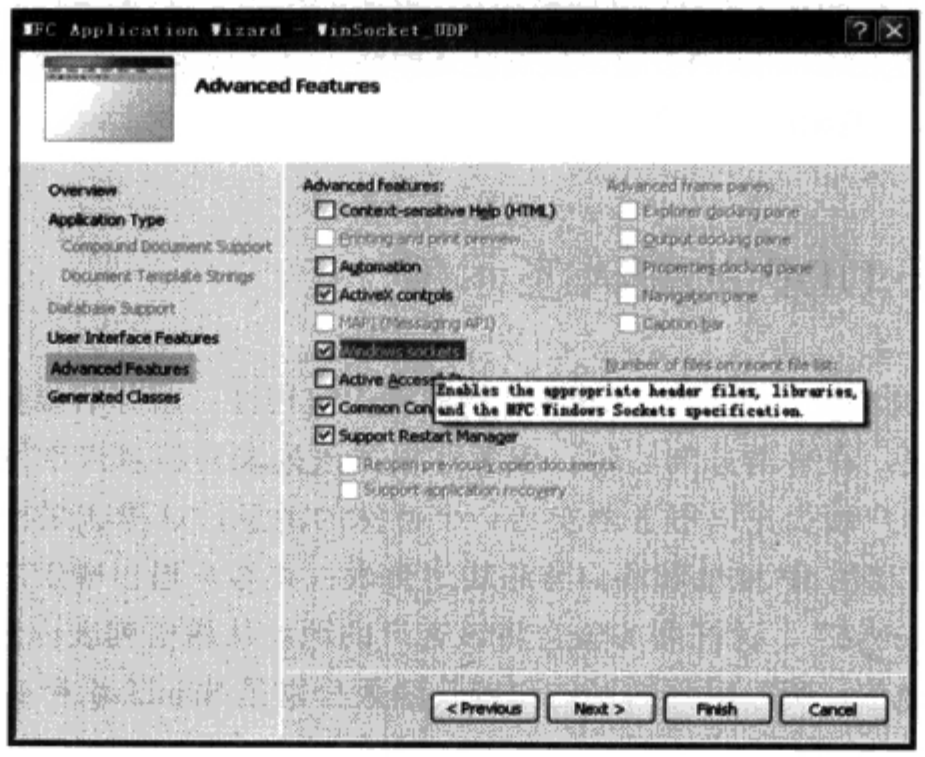


图 15.5　在 MFC 向导中选中 Windows Sockets 支持

AfxScoketInit()函数的原型是：

```
BOOL AfxSocketInit(
    WSADATA* lpwsaData = NULL
);
```

参数 lpwsaData 是一个返回值，为一个指向 WSADATA 的指针。如果输入值不是 NULL，AfxSocketInit()将进一步调用 WSAStartup()函数将其加载的库版本的有关信息填充到这个结构体内；同时，在程序退出前，将自动调用 WSACleanup()函数清除这个对象的空间。WSADATA 结构体的定义如下：

```
struct WSAData {
    WORD wVersion;
    WORD wHighVersion;
    char szDescription[WSADESCRIPTION_LEN+1];
    char szSystemStatus[WSASYSSTATUS_LEN+1];
    unsigned short iMaxSockets;
    unsigned short iMaxUdpDg;
    char FAR * lpVendorInfo;
};
```

该结构主要用于存储 Windows Sockets 的初始化初始化信息，一共有 7 个成员，这些成员的说明如表 15.4 所示。

表 15.4　　WSAData 结构的成员

成　　员	说　　明
WVersion	存放 Windows Sockets DLL 希望调用者使用的 Windows Sockets 的版本
WHighVersion	当前 DLL 支持的 Windows Sockets 的最高版本，一般与 wVersion 值相同
szDescription	存放 DLL 复制的关于 Windows Sockets 的描述信息，主要在显示状态信息时用到这些信息
szSystemStatus	存放 DDL 复制相对状态或配置信息
iMaxSockets	指定一个应用程序所能打开的 Socket 对象的最大数目。该值将由 DLL 及网络软件共同配置。用户在应用程序启动时，可通过这个值查询 Windows Sockets 在程序中是否可用
iMaxUdpDg	指定每次可以发送或接收的数据报的最大长度。如果值为 0，表示没有任何限制
lpVendorInfo	返回一个指向特定 Vendor 数据结构的指针

15.4.2　服务器端的实现

下面先看服务器端的实现。由于是无连接，服务器端的工作很简单：监听是否有数据到达，如果有，就接收数据，然后解读数据。如果是一般消息，就依次发送给所有客户端；如果是一个代表新客户端进入的信息，那么将这个客户端的地址加入列表，然后广播一个说明有新客户端加入的系统信息；如果是代表一个客户端退出的信息，就将这个客户端的地址从列表中删除，然后广播一个客户离开的系统信息。

该程序为 MFC 的对话框应用程序，服务器端的界面如图 15.6 所示。上面的文本框（Edit Control）显示用户的进入、退出信息及所有用户发送的数据。这里直接用客户端的 IP 地址和端口号来代表客户端。下面的文本框显示当前聊天室内的客户端个数。

为了实现在数据到达时，服务器将信息广播给所有客户端，将从 CSocket 类继承一个新类，该类主要是重载 OnReceive()函数，当数据到达时进行处理。首先利用向导在工程内添加一个基类为 CSocket 的新类 CChatServerSocket，定义如例 15.1 所示。

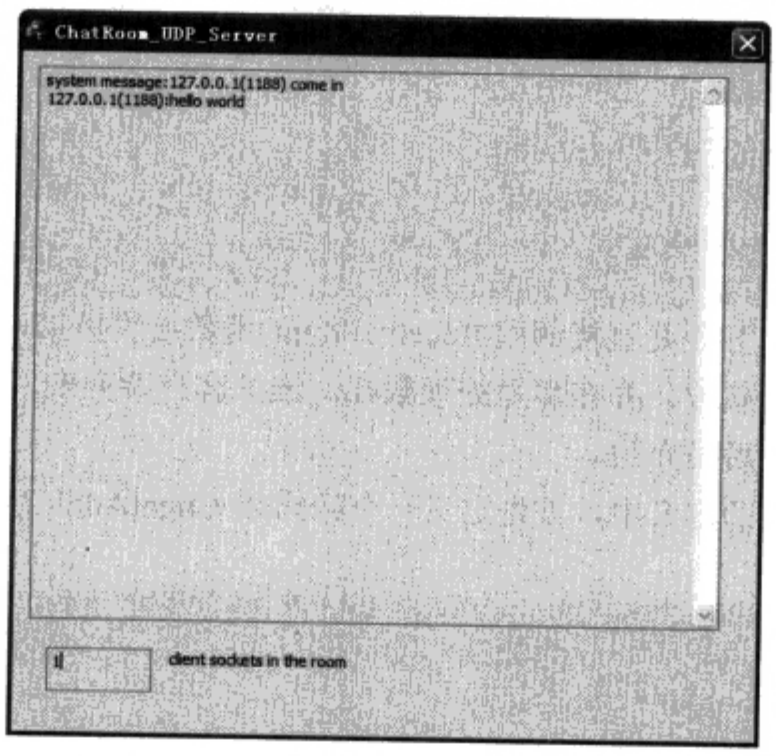


图 15.6　基于 UDP 聊天室的服务器端运行界面

例 15.1

```
#pragma once
class CChartRoom_UDP_ServerDlg;
// CChatServerSocket command target
class CChatServerSocket : public CSocket
{
public:
    CChatServerSocket(CChartRoom_UDP_ServerDlg*pdlg);
    virtual ~CChatServerSocket();
private:
    CChartRoom_UDP_ServerDlg* m_pdlg;
protected:
    virtual void OnReceive(int nErrorCode);
};
```

这里添加了一个成员变量 m_pdlg 存放主对话框类对象的指针，这样在数据到达时，该 Socket 对象就会响应到达事件，调用对话框的成员函数对到达的数据进行处理。该类的实现如例 15.2 所示。

例 15.2

```
#include "stdafx.h"
#include "ChartRoom_UDP_Server.h"
#include "ChatServerSocket.h"
#include "ChartRoom_UDP_ServerDlg.h"
// CChatServerSocket
CChatServerSocket::CChatServerSocket(CChartRoom_UDP_ServerDlg*pdlg)
{
    m_pdlg = pdlg;
}
CChatServerSocket::~CChatServerSocket()
{
}
// CChatServerSocket member functions
void CChatServerSocket::OnReceive(int nErrorCode){
    //首先调用基类处理函数
    CSocket::OnReceive(nErrorCode);
    //在对话框对象中处理消息
    m_pdlg->ProcessPendingRead();
}
```

在 OnReceive()函数内，并不是直接调用 ReceiveFrom()函数接收数据，而是放到对话框的成员函数 ProcessPendingRead()内进行接收，并在那里处理数据，维护客户端列表，将数据广播给所有客户端。该函数是通信的核心，将在后面介绍。

在服务器端主对话框类的定义中，首先定义一个结构体 ClientAddr 存放客户端的地址信息，定义如下：

```
struct ClientAddr{
    //存放 IP 地址
    CStringW strIPAddr;
    //存放端口号
    UINT uiPort;
};
```

该结构只有两个成员，分别存放客户端的 IP 地址和端口号。

主对话框类的主要成员函数和成员变量如下：

```
public:
    UINT m_uiClientNum;
public:
    void ProcessPendingRead();
protected:
    CChatServerSocket* m_pServerSocket;
    CArray<ClientAddr,ClientAddr&>m_ClientAddrList;
```

成员 m_uiClientNum 存放当前的客户端数目，并显示到控件上。成员 m_pServerSocket 为一个指向服务器端的 Socket 对象的指针，通过它来接收客户端发送过来的数据，并将数据发送给每个客户端。成员 m_ClientAddrList 为一个 CArray 类型，代表当前所有客户端的通信地址信息，只有通过该地址列表信息，服务器端才能将消息广播给每个客户端。成员函数 ProcessPendingRead()就是数据到达时真正

的处理函数：首先调用 Socket 对象的 ReceiveFrom()函数接收数据，然后分析数据，并作相应处理。

这些成员的初始化及析构函数如下：

```
CChartRoom_UDP_ServerDlg::CChartRoom_UDP_ServerDlg(CWnd* pParent
        /*=NULL*/)
    : m_uiClientNum(0),CDialog(CChartRoom_UDP_ServerDlg::IDD, pParent)
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    m_pServerSocket = NULL;
}
void CChartRoom_UDP_ServerDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Text(pDX,IDC_EDIT_CLIENTSOCNUM,m_uiClientNum);
}
CChartRoom_UDP_ServerDlg::~CChartRoom_UDP_ServerDlg(){
    if(m_pServerSocket!=NULL){
        delete m_pServerSocket;
        m_pServerSocket = NULL;
    }
}
```

在构造函数内，这个指向 Socket 对象的指针被赋值为 NULL，通过 DoDataExchange()函数将客户端的数目在控件上显示出来。在析构函数中，要将 Socket 对象释放掉，在 Socket 对象被释放时会自动调用 Close()函数，所以这里不用显式调用 Close()函数。

真正的 Socket 对象的初始化在 OnInitDialog()函数内进行。代码如下：

```
m_pServerSocket = new CChatServerSocket(this);
//创建数据报 Socket
m_pServerSocket->Create(8080,SOCK_DGRAM);
```

先构造出一个空白的 Socket 对象 ，再调用 Create()函数创建附属的 SOCKET 句柄。这里指定服务器端口号为 8080，类型为 SOCK_DGRAM 数据报，IP 地址由系统自动指定。为了使服务器能获得一个稳定的端口号，最好自己指定一个端口号，而不要依赖于系统自动分配，因为每次运行时系统状态会发生变化，每一次都可能分配到不同的空闲端口号。

最后就是最主要的处理数据到达的函数 ProcessPendingRead()。该函数首先调用 Socket 对象的 ReceiveFrom()函数接收数据，然后分析数据。如果是代表一个新的客户端进入，就将这个新的客户端地址加入到列表内，然后广播新用户进入的消息；如果是代表一个客户离开，则先将这个客户的地址信息从列表内删除，接着广播客户离开的消息；如果是客户发送的普通消息，则首先在信息前面加上客户的地址信息，然后将这个“封装”后的消息广播出去。在服务器的文本框内，将这个消息添加到后面，在控件上显示消息。该函数的代码如例 15.3 所示。

例 15.3

```
void CChartRoom_UDP_ServerDlg::ProcessPendingRead(){
    TCHAR buff[4096];
    int nRead;
    ClientAddr sendClientAddr;
    //首先接收数据
    nRead = m_pServerSocket->ReceiveFrom(buff, 4096,sendClientAddr.strIPAddr,sendClientAddr.uiPort);
    if( nRead==SOCKET_ERROR ){
```

```
        return;
    }
    buff[nRead] = _T('\0');
    CString strTemp(buff);
    int i;
    CString strServerMsg;
    //然后分析数据
    if( strTemp.CompareNoCase(_T("enter")) == 0 ){
        //一个新的客户端进来
        m_ClientAddrList.Add(sendClientAddr);
        CStringW strEnterMsg;
        strEnterMsg.Format(_T("system message:%s(%d) come in"),sendClientAddr.strIPAddr,sendClientAddr.uiPort);
        for(i=0;i<m_ClientAddrList.GetSize();i++){
            ClientAddr& tempClient = m_ClientAddrList.ElementAt(i);
            m_pServerSocket->SendTo(
                    (LPCTSTR)strEnterMsg,
                    strEnterMsg.GetLength()*8,
                    tempClient.uiPort,
                    tempClient.strIPAddr
            );
        }
        strServerMsg = strEnterMsg;
        m_uiClientNum = m_ClientAddrList.GetSize();
        UpdateData(FALSE);
    }
    else if( strTemp.CompareNoCase(_T("byebye")) == 0 ){
        //一个客户离开房间
        for(i=0;i<m_ClientAddrList.GetSize();i++){
            ClientAddr& tempClient = m_ClientAddrList.ElementAt(i);
            if(tempClient.uiPort == sendClientAddr.uiPort
                    && tempClient.strIPAddr.Compare(sendClientAddr.strIPAddr) == 0 ){
                    break;
            }
        }
        if(i<m_ClientAddrList.GetSize()){
            m_ClientAddrList.RemoveAt(i);
        }
        CStringW strLeaveMsg;
        strLeaveMsg.Format(_T("system message:%s(%d) leave"),sendClientAddr.strIPAddr,sendClientAddr.uiPort);
        for(i=0;i<m_ClientAddrList.GetSize();i++){
            ClientAddr& tempClient = m_ClientAddrList.ElementAt(i);
            m_pServerSocket->SendTo(
                    (LPCTSTR)strLeaveMsg,
                    strLeaveMsg.GetLength()*8,
                    tempClient.uiPort,
                    tempClient.strIPAddr
            );
        }
        strServerMsg = strLeaveMsg;
```

```
            m_uiClientNum = m_ClientAddrList.GetSize();
            UpdateData(FALSE);
        }
        else{
                                        //普通的通话信息
            CStringW strMsg;
            strMsg.Format(_T("%s(%d):%s"),sendClientAddr.strIPAddr,sendClientAddr.uiPort,strTemp);
            for(i=0;i<m_ClientAddrList.GetSize();i++){
                ClientAddr& tempClient = m_ClientAddrList.ElementAt(i);
                m_pServerSocket->SendTo(
                        (LPCTSTR)strMsg,
                        strMsg.GetLength()*8,
                        tempClient.uiPort,
                        tempClient.strIPAddr
                    );
            }
            strServerMsg = strMsg;
        }
        CStringW allMsg;
        this->GetDlgItemTextW(IDC_EDIT_ALLCHATMSG,allMsg);
        allMsg += strServerMsg + _T("\r\n");
        this->SetDlgItemTextW(IDC_EDIT_ALLCHATMSG,allMsg);
}
```

15.4.3　客户端的实现

下面讲解客户端的实现过程，客户端的运行界面如图 15.7 所示。

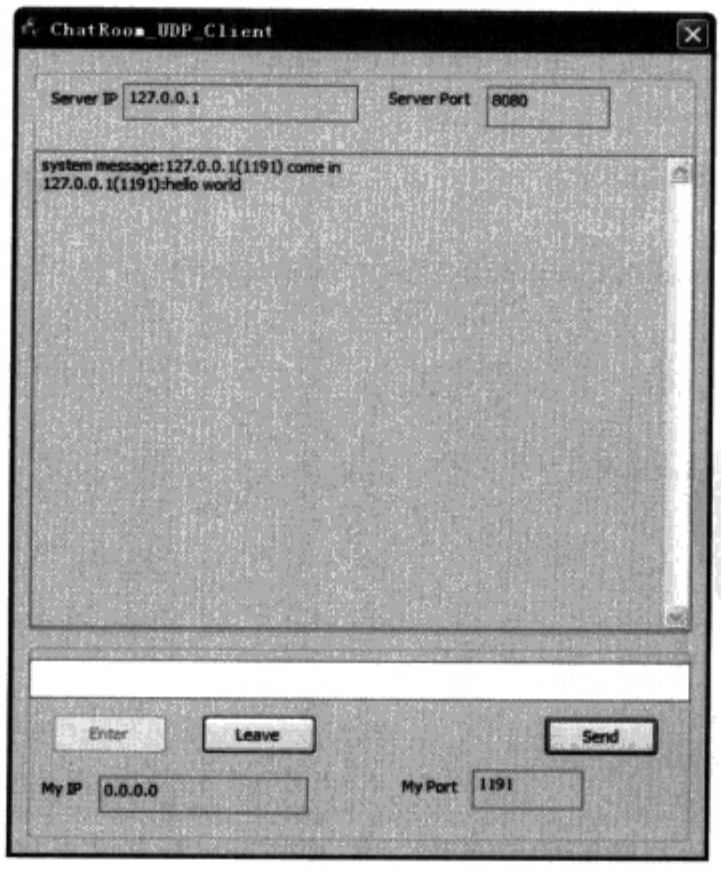


图 15.7　基于 UDP 聊天室的客户端运行界面

最上面的一行分别显示服务器端的 IP 地址和端口号，最下面一行显示自己的 IP 地址和端口号，

这里是使用系统自动分配的值（可以通过调用 GetSockName()函数查询）。中间的大文本框内显示服务器发送过来的所有消息，包括客户端进入、退出及其他客户端发送的普通消息。其下面的小文本框用于接收用户的输入信息，单击 Send 按钮就表示将当前的输入发送给服务器端，再通过服务器端广播给所有客户端。在通信之前需要先单击 Enter 按钮进入聊天室，其实只是发送一条特殊的消息而已。单击 Leave 按钮表示客户端退出聊天室，也只是发送一条特殊的消息而已。

同样，需要先从 CSocket 类派生出一个代表客户端 Socket 的类，该类也是主要重载 OnReceive()函数，在数据到达事件发生时进行处理，实际上就是处理服务器发送过来的数据，因为此时都是只跟服务器通信。

该类定义如例 15.4 所示。

例 15.4

```
#pragma once
class CChartRoom_UDP_ClientDlg;
// ChatClientSocket command target
class ChatClientSocket : public CSocket
{
public:
    ChatClientSocket(CChartRoom_UDP_ClientDlg*pdlg);
    virtual ~ChatClientSocket();
protected:
    virtual void OnReceive(int nErrorCode);
private:
    CChartRoom_UDP_ClientDlg*m_pdlg;
};
```

该类的实现如例 15.5 所示。

例 15.5

```
#include "stdafx.h"
#include "ChartRoom_UDP_Client.h"
#include "hatClientSocket.h"
#include "ChartRoom_UDP_ClientDlg.h"
// ChatClientSocket
ChatClientSocket::ChatClientSocket(CChartRoom_UDP_ClientDlg*pdlg)
{
    m_pdlg = pdlg;
}
ChatClientSocket::~ChatClientSocket()
{
}
// ChatClientSocket member functions
void ChatClientSocket::OnReceive(int nErrorCode){
    CSocket::OnReceive(nErrorCode);
    m_pdlg->ProcessPendingRead();
}
```

可以看到 OnReceive()函数与前面的服务器端基本类似，不在这里接收数据，而是放到对话框的成员函数 ProcessPendingRead()内集中处理。

主对话框类的主要成员如下：

```
protected:
    ChatClientSocket* m_pClientSock;
    virtual BOOL DestroyWindow();
    BOOL m_bEnterRoom;
public:
    SocketAddr m_ServerAddr;
    afx_msg void OnBnClickedButtonEnterroom();
public:
    void ProcessPendingRead();
```

构造函数、析构函数及数据传输代码如下：

```
CChartRoom_UDP_ClientDlg::CChartRoom_UDP_ClientDlg(CWnd* pParent
        /*=NULL*/)
    : CDialog(CChartRoom_UDP_ClientDlg::IDD, pParent)
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    //初始值 127.0.0.1 代表本地 IP
    m_ServerAddr.strIPAddr = _T("127.0.0.1");
    m_ServerAddr.uiPort = 8080;
    m_pClientSock = NULL;
    m_bEnterRoom = FALSE;
}
CChartRoom_UDP_ClientDlg::~CChartRoom_UDP_ClientDlg(){
    if(m_pClientSock!=NULL){
        delete m_pClientSock;
        m_pClientSock = NULL;
    }
}
void CChartRoom_UDP_ClientDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Text(pDX,IDC_EDIT_SERVERIP,m_ServerAddr.strIPAddr);
    DDX_Text(pDX,IDC_EDIT_SERVERPORT,m_ServerAddr.uiPort);
}
```

Socket 对象的真正初始化在 OnInitDialog 函数内进行，主要代码如下：

```
m_pClientSock = new ChatClientSocket(this);
//使用数据报 Socket 类型
m_pClientSock->Create(0,SOCK_DGRAM);
SocketAddr myAddr;
//取回地址信息
m_pClientSock->GetSockName(myAddr.strIPAddr,myAddr.uiPort);
this->SetDlgItemTextW(IDC_EDIT_MYIP,myAddr.strIPAddr);
CStringW strMyPort;
strMyPort.Format(_T("%d"),myAddr.uiPort);
this->SetDlgItemTextW(IDC_EDIT_MYPORT,strMyPort);
this->GetDlgItem(IDC_BUTTON_ENTERROOM)->EnableWindow(TRUE);
this->GetDlgItem(IDC_BUTTON_LEAVEROOM)->EnableWindow(FALSE);
this->GetDlgItem(IDC_BUTTON_SENDMSG)->EnableWindow(FALSE);
((CEdit*)this->GetDlgItem(IDC_EDIT_SENDMSG))->SetReadOnly(TRUE);
```

```
((CEdit*)this->GetDlgItem(IDC_EDIT_SERVERIP))->SetReadOnly(FALSE);
((CEdit*)this->GetDlgItem(IDC_EDIT_SERVERPORT))->SetReadOnly(FALSE);
```

这里除了创建 Socket 对象外，还控制各控件的初始属性：初始化时，需要先进入聊天室，所以只有 Enter 按钮可用，Leave 和 Send 按钮不可用，而且用户输入框也设置为只读，无法输入信息。但是服务器端的 IP 及端口号控件设置为可写，允许用户自己指定服务器的地址和端口号。初始化时各个控件的状态如图 15.8 所示。

接着介绍 Enter 按钮的事件处理。这里只是简单的发送一个消息过去，由于是数据报，因此即使发送成功也不能保证是否会被服务器接收到。为了简化，这里没有进行出错处理，而是假定发送成功就可以被服务器接收到。Enter 按钮的单击事件处理函数如例 15.8 所示。

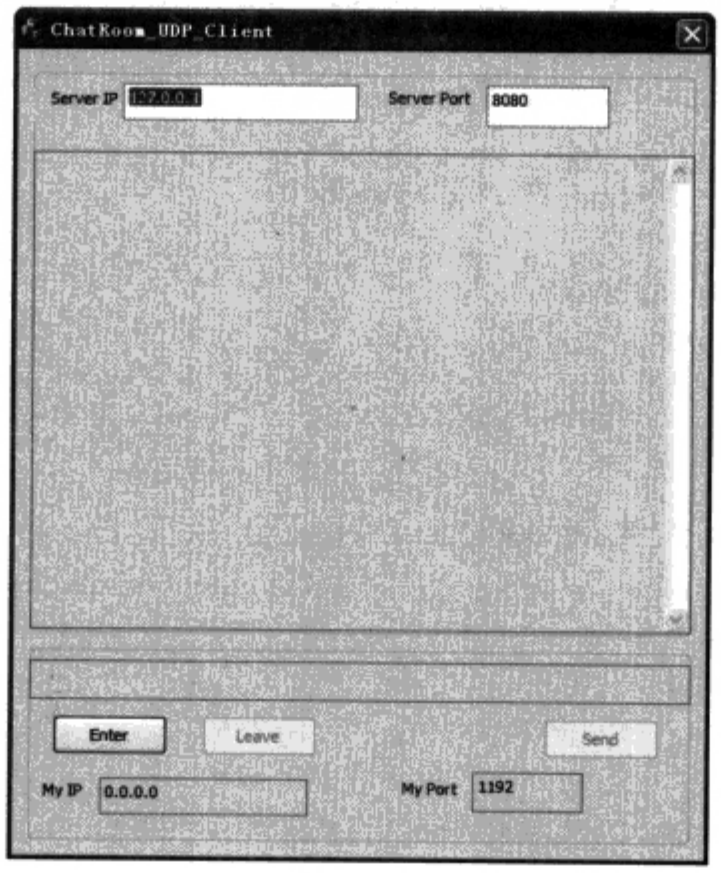


图 15.8　客户端各个控件的初始化状态

例 15.6

```
void CChartRoom_UDP_ClientDlg::OnBnClickedButtonEnterroom()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    CString strEnterMsg("enter");
    int nSend;
    nSend = m_pClientSock->SendTo(
        (LPCTSTR)strEnterMsg,
        strEnterMsg.GetLength()*8,
        m_ServerAddr.uiPort,
        m_ServerAddr.strIPAddr
    );
    if(nSend==SOCKET_ERROR ){
        AfxMessageBox(_T("进入聊天室失败，可能服务器端还没运行"));
        return;
    }
    this->GetDlgItem(IDC_BUTTON_ENTERROOM)->EnableWindow(FALSE);
```

```
    this->GetDlgItem(IDC_BUTTON_LEAVEROOM)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_SENDMSG)->EnableWindow(TRUE);
    ((CEdit*)this->GetDlgItem(IDC_EDIT_SENDMSG))->SetReadOnly(FALSE);
    ((CEdit*)this->GetDlgItem(IDC_EDIT_SERVERIP))->SetReadOnly(TRUE);
    ((CEdit*)this->GetDlgItem(IDC_EDIT_SERVERPORT))->SetReadOnly(TRUE);
    m_bEnterRoom = TRUE;
}
```

这里先发送一个特殊信息通知服务器该客户端已经进入聊天室，然后设置相应控件的属性：用户可以输入消息，Send 按钮可用，Leave 按钮可用，Enter 按钮不可用，显示服务器端 IP 地址和端口号的控件变为只读，不允许用户修改。由于客户端的 IP 地址和端口号由系统指定，不用修改，所以一直为只读，如图 15.9 所示。

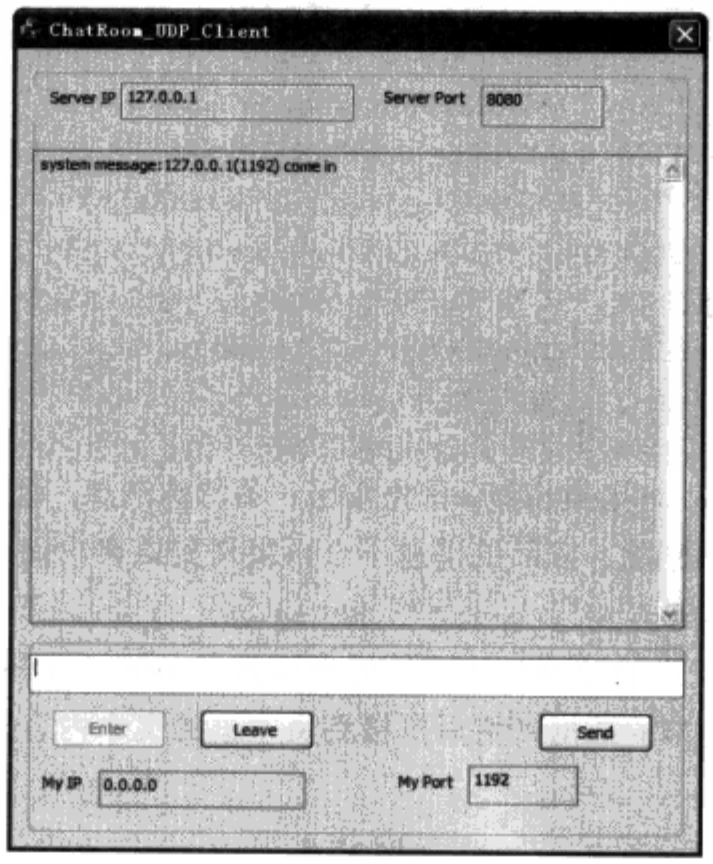


图 15.9　进入聊天室后客户端各个控件的状态

发送用户输入信息的 Send 按钮单击事件的处理函数如例 15.7 所示。

例 15.7

```
void CChartRoom_UDP_ClientDlg::OnBnClickedButtonSendmsg()
{
    // TODO: Add your control notification handler code here
    CStringW strSendMsg;
    this->GetDlgItemTextW(IDC_EDIT_SENDMSG,strSendMsg);
    m_pClientSock->SendTo(
        (LPCTSTR)strSendMsg,
        strSendMsg.GetLength()*8,
        m_ServerAddr.uiPort,
        m_ServerAddr.strIPAddr
    );
    this->SetDlgItemTextW(IDC_EDIT_SENDMSG,_T(""));
}
```

这里的操作相当简单，首先获取用户在文本框内输入的内容，然后将其发送到服务器，服务器加

上发送客户端的地址信息后，广播到所有客户端（包括自己）。这里的操作对控件属性没有影响。

Leave 按钮的单击事件处理函数如例 15.8 所示。

例 15.8

```
void CChartRoom_UDP_ClientDlg::OnBnClickedButtonLeaveroom()
{
    // TODO: Add your control notification handler code here
    CString strEnterMsg("byebye");
    m_pClientSock->SendTo(
        (LPCTSTR)strEnterMsg,
        strEnterMsg.GetLength()+*8,
        m_ServerAddr.uiPort,
        m_ServerAddr.strIPAddr
    );
    this->GetDlgItem(IDC_BUTTON_ENTERROOM)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_LEAVEROOM)->EnableWindow(FALSE);
    this->GetDlgItem(IDC_BUTTON_SENDMSG)->EnableWindow(FALSE);
    ((CEdit*)this->GetDlgItem(IDC_EDIT_SENDMSG))->SetReadOnly(TRUE);
    ((CEdit*)this->GetDlgItem(IDC_EDIT_SERVERIP))->SetReadOnly(FALSE);
    ((CEdit*)this->GetDlgItem(IDC_EDIT_SERVERPORT))-> SetReadOnly(FALSE);
    m_bEnterRoom = FALSE;
}
```

其与 Enter 按钮单击事件处理函数相似，首先发送一条特殊消息，通知服务器该客户端退出了聊天室，将客户端的地址从列表中删除，以后的消息不会再发送到该客户端，然后改变控件的属性，回到程序初始化（未进入）时的状态。

下面是客户端在服务器发送的数据到达时的处理函数 ProcessPendingRead()，首先调用 ReceiveFrom()函数接收服务器发送过来的数据，然后拼接到文本框的文本内容后面，再显示到控件上。该函数的代码如例 15.9 所示。

例 15.9

```
void CChartRoom_UDP_ClientDlg::ProcessPendingRead(){
    TCHAR buff[4096];
    int nRead;
    SocketAddr sendClientAddr;
    //从服务器接收数据
    nRead = m_pClientSock->ReceiveFrom(buff, 4096, sendClientAddr.strIPAddr,sendClientAddr.uiPort);
    if( nRead==SOCKET_ERROR ){
        return;
    }
    //将新的接收数据拼接显示到控件
    buff[nRead] = _T('\0');
    CStringW strTemp(buff);
    CStringW chartMsg;
    this->GetDlgItemTextW(IDC_EDIT_CHARTMSG,chartMsg);
    chartMsg += strTemp + _T("\r\n");
    this->SetDlgItemTextW(IDC_EDIT_CHARTMSG,chartMsg);
}
```

最后有一个细节上的处理，就是客户端在退出程序前，可能不进行 Leave 操作，导致服务器维护的客户端数据出错。为了保证在程序退出前必须先退出聊天室，重载对话框类的 DestroyWindow()函数。

对于在窗口销毁之前而没有执行过 Leave 操作的客户端，就先调用 Leave 按钮的处理函数，退出聊天室后，再退出程序。

```
BOOL CChartRoom_UDP_ClientDlg::DestroyWindow(){
    if(m_bEnterRoom){
        OnBnClickedButtonLeaveroom();
    }
    return CDialog::DestroyWindow();
}
```

在运行程序时，设置了服务器的默认端口号是 8080，先运行服务器端的程序，然后运行客户端程序，设置好服务器的 IP 地址和端口号（使用默认值即可）后，就可以执行 Enter 操作，然后就可以发送消息了。下面运行 4 个客户端，都执行了 Enter 操作，并发送了一些消息。服务器端的运行界面如图 15.10 所示，4 个客户端运行界面如图 15.11 所示。

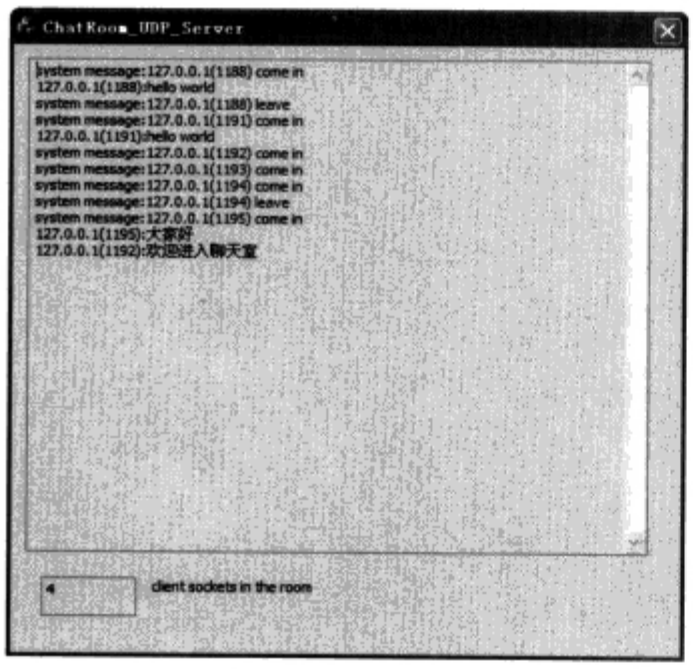


图 15.10　服务器端最终运行结果

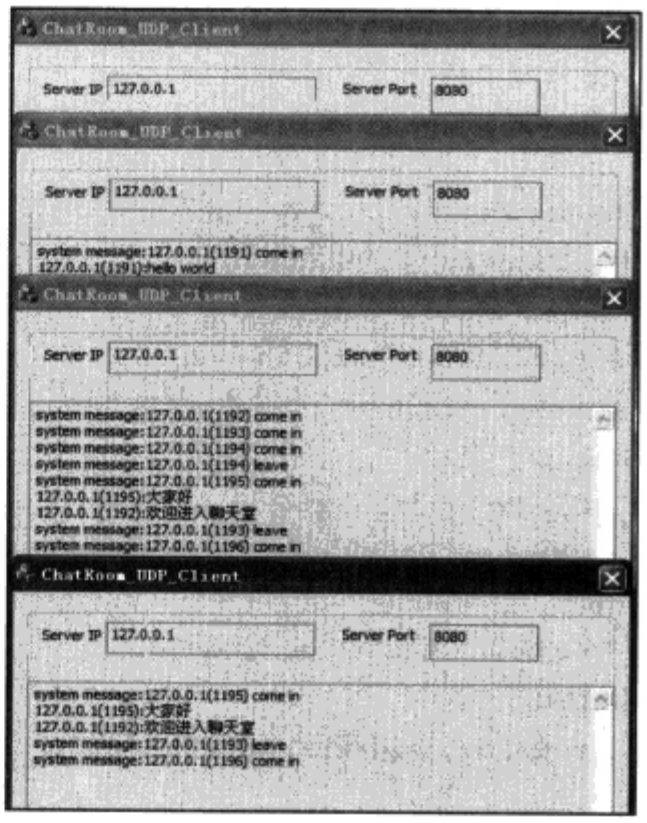


图 15.11　4 个客户端的运行结果

15.5　一个基于 TCP 的聊天室示例

在前面基于 UDP 的聊天室的基础上，使用流式 Socket 来实现一个基于 TCP 的聊天室程序，处理流程如下。

1．客户端的处理流程

（1）创建一个空白的 CSocket 类或 CSocket 派生类的对象。
（2）调用 Create()函数创建流式 Socket。
（3）调用 Connect()函数建立一个到服务器的连接。
（4）调用 Send()、Receive()函数向服务器发送消息及从服务器接收消息。
（5）发送一个特殊的消息，通知服务器客户端退出聊天室。
（6）调用 Close()函数关闭 Socket 对象，回收空间。

2．服务器端的处理流程

（1）创建一个空白的 CSocket 类或 CSocket 派生类的对象。

（2）调用 Create()函数创建流式 Socket，指定一个端口号。

（3）调用 Listen()函数监听来自客户端的连接请求。

（4）调用 Send()、Receive()函数将消息依次发送给各个客户端，接收客户端发送过来的数据。

（5）调用 Close()函数关闭 Socket 对象，回收空间。

15.5.1 服务器端的实现

前面已经介绍过，服务器端的主 Socket 对象只负责监听来自客户端的连接请求，如果有请求，就新建一个 Socket 对象来负责与客户端通信，而自己再回到监听的状态。为此需要再加入一个 CSockt 类的派生类，这个类重载 OnAccept()函数，专门用于处理连接请求到达事件。该类的定义如例 15.10 所示。

例 15.10

```
#pragma once
class CChartRoom_UDP_ServerDlg;
// CListenSocket command target
class CListenSocket : public CSocket
{
public:
    CListenSocket(CChartRoom_UDP_ServerDlg* pdlg);
    virtual ~CListenSocket();
private:
    CChartRoom_UDP_ServerDlg* m_pdlg;
protected:
    virtual void OnAccept(int nErrorCode);
};
```

主要的 OnAccept()函数实现如下：

```
void CListenSocket::OnAccept(int nErrorCode)
{
    CSocket::OnAccept(nErrorCode);
    m_pdlg->ProcessPendingAccept();
}
```

同样将实际的处理放到主对话框类内进行。

还要从 CSocket 类派生一个类专门用于与客户端的通信，该类主要是重载了 OnReceive()函数，具体实现如下：

```
void CChatServerSocket::OnReceive(int nErrorCode){
    //首先调用基类处理函数
    CSocket::OnReceive(nErrorCode);
    //在对话框对象中处理消息
    m_pdlg->ProcessPendingRead(this);
}
```

这里因为实际接收数据的是该 Socket 对象，而不是主 Socket 对象，但是在对话框对象内无法区分哪个 Socket 对象有数据到达，所以需要将该 Socket 对象的指针传入，通过此指针接收客户端的数据。

服务器端主对话框类的主要成员如下：

```
public:
    UINT m_uiClientNum;
public:
    void ProcessPendingRead(CChatServerSocket*pSocket);
    void ProcessPendingAccept();
protected:
    //服务器端的主 Socket
    CListenSocket* m_pListenSocket;
    //维护用于与客户端通信的 Socket 对象列表
    CArray<CChatServerSocket*,CChatServerSocket*&>m_ServerSocketList;
```

这里的 m_pListenSocket 成员为服务器的主 Socket 对象，只负责监听客户端的连接请求，维护一个实际与客户端通信的 Socket 对象列表：当有新的连接请求到达时，先创建一个空白的用于通信的 Socket 对象，然后主 Socket 对象调用 Accept()函数接受请求，并将该 Socket 对象加入列表；如果客户端发送一个退出聊天室的信息，就将负责与这个客户端通信的 Socket 对象从列表中删除；如果是普通消息，就按列表广播给每个客户端。

主 Socket 对象的初始化在 OnInitDialog 中，代码如下：

```
m_pListenSocket = new CListenSocket(this);
    CServerPortDlg setServerDlg;
    setServerDlg.DoModal();
    //创建流式 Socket
    m_pListenSocket->Create(setServerDlg.m_uiServerPort);
    //调用 Listen()函数监听客户端的连接请求
    if( !m_pListenSocket->Listen() ){
        AfxMessageBox(_T("server socket fail in listen"));
        return FALSE;
    }
```

这里通过指定的端口号创建一个流式 Socket，然后调用 Listen()函数来监听来自客户端的连接请求。一旦有连接请求到达，主框架就会调用主 Socket 对象的 OnAccept()函数，该函数再调用对话框的 ProcessPendingAccept()函数来真正处理这个连接请求。对话框类的 ProcessPending Accept()函数实现如例 15.11 所示。

例 15.11

```
void CChartRoom_UDP_ServerDlg::ProcessPendingAccept(){
    CChatServerSocket*pSevSoc = new CChatServerSocket(this);
    if( m_pListenSocket->Accept(*pSevSoc) ){
        m_ServerSocketList.Add(pSevSoc);
    }
    else{
        delete pSevSoc;
    }
}
```

这里的处理相当简单，主 Socket 对象调用 Accept()函数，接受连接请求，并将以后的通信指派给新建的通信 Socket 对象，将该通信 Socket 对象加入到列表中。

最后是客户端发送的数据到达时的处理函数 ProcessPendingRead()的实现，如例 15.12 所示。

例 15.12

```
void CChartRoom_UDP_ServerDlg::ProcessPendingRead(
CChatServerSocket*pSevSoc){
    TCHAR buff[4096];
    int nRead;
    ClientAddr sendClientAddr;
    //首先接收数据
    nRead = pSevSoc->Receive(buff, 4096);
    if( nRead==SOCKET_ERROR ){
        return;
    }
    //获取发送数据的客户端的地址信息
    pSevSoc->GetPeerName(sendClientAddr.strIPAddr,sendClientAddr.uiPort);
    buff[nRead] = _T('\0');
    CString strTemp(buff);
    int i;
    CString strServerMsg;
    //然后分析数据
    if( strTemp.CompareNoCase(_T("enter")) == 0 ){
        //一个新的客户端进来
        CStringW strEnterMsg;
        strEnterMsg.Format(_T("system message:%s(%d) come in"),sendClientAddr.strIPAddr,sendClientAddr.uiPort);
        for(i=0;i<m_ServerSocketList.GetSize();i++){
            CChatServerSocket*&    tempClient = m_ServerSocketList.ElementAt(i);
            tempClient->Send( (LPCTSTR)strEnterMsg,strEnterMsg.GetLength()+100);
        }
        strServerMsg = strEnterMsg;
        m_uiClientNum = m_ServerSocketList.GetSize();
        UpdateData(FALSE);
    }
    else if( strTemp.CompareNoCase(_T("byebye")) == 0 ){
        //一个客户离开房间
        for(i=0;i<m_ServerSocketList.GetSize();i++){
            CChatServerSocket*& tempClient = m_ServerSocketList.ElementAt(i);
            ClientAddr tempClientAddr;
            tempClient->GetPeerName(tempClientAddr.strIPAddr, tempClientAddr.uiPort);
            if(tempClientAddr.uiPort == sendClientAddr.uiPort && tempClientAddr.strIPAddr.Compare(
                        sendClientAddr.strIPAddr) == 0 ){
                break;
            }
        }
        if(i<m_ServerSocketList.GetSize()){
            //将负责与这个离开的客户端通信的 Socket 对象从列表中删除
            m_ServerSocketList.RemoveAt(i);
        }
        CStringW strLeaveMsg;
        strLeaveMsg.Format(_T("system message:%s(%d) leave"),
            sendClientAddr.strIPAddr,sendClientAddr.uiPort);
        for(i=0;i<m_ServerSocketList.GetSize();i++){
            CChatServerSocket*& tempClient = m_ServerSocketList.ElementAt(i);
```

```
                tempClient->Send((LPCTSTR)strLeaveMsg,strLeaveMsg.GetLength()+100);
            }
            strServerMsg = strLeaveMsg;
            m_uiClientNum = m_ServerSocketList.GetSize();
            UpdateData(FALSE);
        }
        else{
            //普通的通话信息
            CStringW strMsg;
            strMsg.Format(_T("%s(%d):%s"),sendClientAddr.strIPAddr, sendClientAddr.uiPort,strTemp);
            for(i=0;i<m_ServerSocketList.GetSize();i++){
                CChatServerSocket*&    tempClient = m_ServerSocketList.ElementAt(i);
                tempClient->Send((LPCTSTR)strMsg, strMsg.GetLength()+100);
            }
            strServerMsg = strMsg;
        }
        CStringW allMsg;
        this->GetDlgItemTextW(IDC_EDIT_ALLCHATMSG,allMsg);
        allMsg += strServerMsg + _T("\r\n");
        this->SetDlgItemTextW(IDC_EDIT_ALLCHATMSG,allMsg);
}
```

这里首先根据传入的通信 Socket 对象的指针调用 Accept()函数来接收数据，然后调用 GetPeerName()函数来获得对应发送数据的客户端的地址信息（就是负责接收数据的通信 Sokcet 对象的对等端），接着分析数据，执行相应操作。

15.5.2　客户端的实现

客户端的处理主要是先要建立一个到服务器的连接，然后向服务器发送数据或接收来自服务器的消息，执行完毕就断开与服务器的连接，退出程序。

客户端的运行界面如图 15.12 所示。

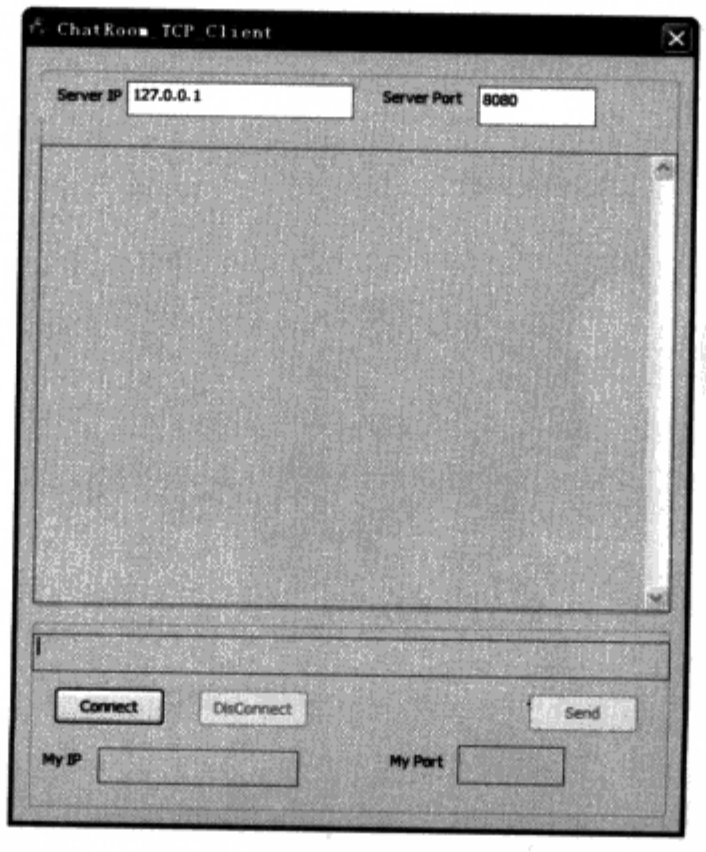


图 15.12　基于 TCP 聊天室的客户端运行界面

Connect 按钮用于建立一个到服务器的连接，Send 按钮用于将用户的输入发送到服务器端，服务器端再将消息广播给每个客户端（包括自己），DisConnect 用于按钮断开连接。

同样需要从 CSocket 派生一个类，重载 OnReceive()函数，对数据到达事件进行处理。该类的主要代码如下：

```
void ChatClientSocket::OnReceive(int nErrorCode){
    CSocket::OnReceive(nErrorCode);
    m_pdlg->ProcessPendingRead();
}
```

数据接收操作放到主对话框的 ProcessPendingRead()函数内进行，主对话框的主要成员如下。

```
protected:
    ChatClientSocket* m_pClientSock;
    virtual BOOL DestroyWindow();
    BOOL m_bEnterRoom;
public:
    SocketAddr m_ServerAddr;
    afx_msg void OnBnClickedButtonEnterroom();
public:
    void ProcessPendingRead();
    afx_msg void OnBnClickedButtonLeaveroom();
    afx_msg void OnBnClickedButtonSendmsg();
```

成员 m_pClientSock 就是客户端的 Sokcet，成员 m_bEnterRoom 标识当前客户端是否与服务器建立了一个连接。

Connect 按钮的单击事件处理函数如例 15.13 所示。

例 15.13

```
void CChartRoom_UDP_ClientDlg::OnBnClickedButtonEnterroom()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    if(m_pClientSock != NULL){
        delete m_pClientSock;
        m_pClientSock = NULL;
    }
    m_pClientSock = new ChatClientSocket(this);
    //使用流式 Socket
    m_pClientSock->Create();
    SocketAddr myAddr;
    //取回地址信息
    m_pClientSock->GetSockName(myAddr.strIPAddr,myAddr.uiPort);
    this->SetDlgItemTextW(IDC_EDIT_MYIP,myAddr.strIPAddr);
    CStringW strMyPort;
    strMyPort.Format(_T("%d"),myAddr.uiPort);
    this->SetDlgItemTextW(IDC_EDIT_MYPORT,strMyPort);
    //调用 Connect()函数建立与服务器端的连接
    if( !m_pClientSock->Connect(m_ServerAddr.strIPAddr,
        m_ServerAddr.uiPort) ){
        AfxMessageBox(_T("fail to connect to the server,try it again"));
        return;
```

```
    }
    CString strEnterMsg("enter");
    m_pClientSock->Send((LPCTSTR)strEnterMsg, strEnterMsg.GetLength()+100);
    this->GetDlgItem(IDC_BUTTON_ENTERROOM)->EnableWindow(FALSE);
    this->GetDlgItem(IDC_BUTTON_LEAVEROOM)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_SENDMSG)->EnableWindow(TRUE);
    ((CEdit*)this->GetDlgItem(IDC_EDIT_SENDMSG))->SetReadOnly(FALSE);
    ((CEdit*)this->GetDlgItem(IDC_EDIT_SERVERIP))->SetReadOnly(TRUE);
    ((CEdit*)this->GetDlgItem(IDC_EDIT_SERVERPORT))->SetReadOnly(TRUE);
    m_bEnterRoom = TRUE;
}
```

这里首先创建一个流式 Socket，然后调用 Connect()函数建立一个到服务器的连接，如果连接成功，则立即调用 Send()函数发送一个特殊的消息，通知服务器自己已经进入聊天室，然后设置相应的控件属性。

DisConnect 按钮的单击事件处理函数如例 15.14 所示。

例 15.14

```
void CChartRoom_UDP_ClientDlg::OnBnClickedButtonLeaveroom()
{
    // TODO: Add your control notification handler code here
    CString strEnterMsg("byebye");
    m_pClientSock->Send( (LPCTSTR)strEnterMsg, strEnterMsg.GetLength()+100);
    this->GetDlgItem(IDC_BUTTON_ENTERROOM)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_LEAVEROOM)->EnableWindow(FALSE);
    this->GetDlgItem(IDC_BUTTON_SENDMSG)->EnableWindow(FALSE);
    ((CEdit*)this->GetDlgItem(IDC_EDIT_SENDMSG))->SetReadOnly(TRUE);
    ((CEdit*)this->GetDlgItem(IDC_EDIT_SERVERIP))->SetReadOnly(FALSE);
    ((CEdit*)this->GetDlgItem(IDC_EDIT_SERVERPORT))->SetReadOnly(FALSE);
    m_bEnterRoom = FALSE;
}
```

这里就是调用 Send()函数发送一个特殊的消息，通知服务器自己已经断开连接，离开聊天室。

Send 按钮的单击事件处理函数如例 15.15 所示。

例 15.15

```
void CChartRoom_UDP_ClientDlg::OnBnClickedButtonSendmsg()
{
    // TODO: Add your control notification handler code here
    CStringW strSendMsg;
    this->GetDlgItemTextW(IDC_EDIT_SENDMSG,strSendMsg);
    m_pClientSock->Send((LPCTSTR)strSendMsg, strSendMsg.GetLength()+1000);
    this->SetDlgItemTextW(IDC_EDIT_SENDMSG,_T(""));
}
```

首先获得用户输入的数据，然后调用 Send()函数将这些数据发送到服务器端。

最后介绍对话框类内处理数据到达事件的函数 ProcessPendingRead()，如例 15.16 所示。

例 15.16

```
void CChartRoom_UDP_ClientDlg::ProcessPendingRead(){
    TCHAR buff[4096];
```

```
        int nRead;
        SocketAddr sendClientAddr;
        //从服务器接收数据
        nRead = m_pClientSock->Receive(buff, 4096);
        if(nRead==SOCKET_ERROR){
            return;
        }
        //将接收的新数据拼接显示到控件
        buff[nRead] = _T('\0');
        CStringW strTemp(buff);
        CStringW chartMsg;
        this->GetDlgItemTextW(IDC_EDIT_CHARTMSG,chartMsg);
        chartMsg += strTemp + _T("\r\n");
        this->SetDlgItemTextW(IDC_EDIT_CHARTMSG,chartMsg);

    }
```

这里首先调用 Receive()函数从服务器端接收数据，然后拼接显示到控件上。

15.6 小结

本章主要讲述了利用 MFC 提供的对 WinSocket 的封装类 CSocket 编写网络应用程序相关的知识。本章首先介绍了计算机网络的一些基本知识，详细介绍了 TCP/IP 协议模型、IP 地址、端口、数据封装这些概念的涵义；接着简要介绍了 Winsock 的概念，以及利用 Winsock 进行网络编程的模型；然后详细介绍了 MFC 对 Winsock 提供封装的两个类 CAsyncSocket 和 CSocket，其中 CAsyncSocket 提供了对 Socket 的低层封装，而 CSocket 类是对 Windows Sockets API 的高层封装，主要用于支持利用文件 CSocketFile 方式和串行化 CArchive 支持来方便地管理数据的发送和接收。

本章的最后利用 CSocket 类分别编写了一个基于 UDP 的聊天室程序以及一个基于 TCP 的聊天室程序，并详细介绍了程序的创建过程，希望对读者理解 MFC 网络编程有一定的指导意义。

第 16 章　Visual C++ 2010 MFC 进程通信

进程是一个可执行的程序，由私有虚拟地址空间、代码、数据和其他操作系统资源（如进程创建的文件、管道、同步对象等）组成。当启动一个进程时，操作系统会为此进程建立一个 4GB 的地址空间。一个应用程序可以有一个或多个进程，一个进程可以有一个或多个线程，其中一个是主线程。位于同一进程中的多个线程共享同一个地址空间，因此线程之间的通信比较简单。但是为了安全，每个进程的空间都是私有的，任何试图直接访问其他进程空间的操作都会被视为段错误而导致进程被终止，但是可以通过其他途径来实现进程间的通信。本章主要介绍 MFC 提供的 5 种进程间通信方式，具体如下：

- 剪贴板通信；
- 邮槽通信；
- 匿名管道通信；
- 命名管到通信；
- 共享内存通信。

16.1　剪贴板通信

Windows 系统的用户对于剪贴板是再熟悉不过了。例如，在某个打开的 Word 文件内选中一些文字或图片，按【Ctrl+C】组合键或单击“复制”命令，然后在一个打开的 PowerPoint 文件内按【Ctrl+V】组合键或单击“粘贴”命令，就可以把前面的数据复制进来。这实际上就是一个 Word 进程利用剪贴板与一个 PowerPoint 进程之间的通信。在 Windows 系统下，为了安全，各个进程的地址空间都是独立的、相互隔离的，如果一个进程试图直接访问另一个进程的地址空间将导致段错误。而剪贴板为开发者在不同的进程间交换数据提供了一种简单快捷的方法。实际上，剪贴板就是由系统维护的一个全局公共内存区域，任何进程都可以访问这块区域：既可以将自己的数据写入这块区域（称为复制，copy），也可以从这块区域内读取数据（称为粘贴，paste）。下面介绍操作剪贴板的相关函数。

16.1.1　OpenClipboard()函数

应用程序可调用这个函数来打开剪贴板，任何进程在访问剪贴板之前都必须先打开剪贴板。该函数的原型是：

```
BOOL OpenClipboard(
    HWND hWndNewOwner
);
```

如果函数成功执行，则返回非 0（True）；如果打开失败，则返回 0（False）。如果剪贴板已经被其他进程打开，那么此时调用该函数将失败。

参数 hWndNewOwner 指定关联到打开的剪贴板的窗口句柄，如果传入的参数为 NULL，就表示打开的剪贴板将被关联到当前任务。

一旦一个进程打开了剪贴板，那么在这个进程关闭剪贴板之前，其他进程都将无法再打开剪贴板，更无从修改剪贴板内的数据，这是因为剪贴板每次至多只能允许一个进程打开并访问。一个应用程序每成功打开一次剪贴板，都应该对应地调用一次 CloseClipboard()函数来将剪贴板关闭，否则将导致其他进程无法访问剪贴板。

注意这里只是成功地打开剪贴板，并关联到指定窗口，还是不允许向剪贴板写入数据。要想向剪贴板写入数据，首先得成为剪贴板的所有者，而只有在调用 EmptyClipboard()函数将剪贴板内原来的数据清空完毕后，这个窗口才能成为剪贴板的所有者，才能写入数据。

为了方便操作，MFC 的 CWnd 类内提供了一个 OpenClipboard()成员函数：

```
BOOL OpenClipboard( );
```

这里就是将 CWnd 对象的窗口关联到打开的剪贴板上。

16.1.2 CloseClipboard()函数

该函数用于关闭剪贴板，允许其他进程访问剪贴板。函数原型如下：

```
BOOL CloseClipboard(
    void
);
```

该函数没有参数。如果关闭剪贴板成功，则返回非 0（True）；否则返回 0（False）。

当一个进程成功关闭剪贴板后，是不允许再往剪贴板内写入数据的，因为此时它已经不再是剪贴板的所有者。在进程对剪贴板操作完毕后，必须调用该函数将剪贴板关闭，否则其他进程将无法访问剪贴板。

16.1.3 EmptyClipboard()函数

该函数首先将剪贴板内的数据清空，释放数据资源，然后指定当前打开剪贴板的窗口为剪贴板的所有者。函数原型如下：

```
BOOL EmptyClipboard (
    void
);
```

该函数没有参数。如果调用成功，则返回非 0（True）；否则返回 0（False）。

再次强调，在进程往剪贴板内写入数据之前，必须调用该函数，使自己成为剪贴板的所有者。如果只是从剪贴板内读出数据，就不用成为剪贴板的所有者。

16.1.4 SetClipboardData()函数

该函数将数据按特定的格式放入剪贴板，在向剪贴板写入数据之前，当前窗口必须成为剪贴板的所有者。函数原型如下：

```
HANDLE SetClipboardData(
    UINT uFormat,
    HANDLE hMem
);
```

如果函数成功执行，返回值将是数据的句柄；如果写入数据失败，则返回值为 NULL。

- 参数 uFormat 指定一种写入数据的剪贴板格式，既可以是自己注册的格式，也可以是系统内置的标准格式。标准的剪贴板格式有很多种，表 16.1 列出了几种常用的格式。

表 16.1　剪贴板的几种常用格式

数 据 格 式	说　明
CF_BITMAP	位图句柄格式（HBITMAP）
CF_TEXT	文本格式，每行数据以回车换行这一组合字符结束，并以一个空字符来表示数据的结束。ANSI 文本采用这种格式
CF_UNICODETEXT	对于 Windows NT/2000/XP 系统，表示 Unicode 文本格式，以回车换行表示一行数据结束，以一个空字符代表数据结束
CF_PALETTE	调色板句柄。如果应用程序将依赖于调色板的数据放入剪贴板，那么必须将调色板数据也放入剪贴板
CF_OEMTEXT	包含 OEM 字符串的文本，每行数据以回车换行这一组合字符结束，并以一个空字符来表示数据的结束

- 参数 hMem 代表具有特定格式的数据的句柄。这个参数可以指定为 NULL，表示窗口将采用延迟提交技术：只有在对剪贴板的数据请求真正到达时，调用窗口才会为剪贴板提供数据，这时窗口必须处理 WM_RENDERFORMAT 和 WM_RENDERALLFORMATS 消息。

这里要求写入剪贴板的内存对象句柄必须是 GMEM_MOVEABLE 属性，见下面 GlobalAlloc()函数的说明。

一旦 SetClipboardData()函数调用成功完成，那么系统将拥有这些提交的数据。一旦系统取得了被提交数据的所有权，那么应用程序将不能再写入数据或将这些数据释放，但是在关闭剪贴板之前，这些数据仍可以被应用程序锁定并读取（在关闭剪贴板之前，应该对数据解除锁定）。如果 hMem 参数被指定为一个内存对象，那么这个对象必须在被创建时使用 GMEM_MOVEABLE 标志，一般都使用 GlobalAlloc()函数来创建这些将被放入剪贴板的数据对象。

16.1.5　GlobalAlloc()函数

该函数在堆上动态分配指定大小（单位为字节）的内存区域。函数原型如下：

```
HGLOBAL GlobalAlloc(
    UINT uFlags,
    SIZE_T dwBytes
);
```

如果函数成功执行，则返回指向动态分配的内存对象的句柄；如果分配失败，则返回 NULL。

- 参数 uFlags 指定内存分配的属性，表 16.2 列出了可取的内存属性。

表 16.2　内存属性

内 存 属 性	说　明
GMEM_FIXED 0x0000	分配一块固定的内存区域，不允许系统移动，这时的返回值是一个指针（指向固定的地址）
GMEM_MOVEABLE 0x0002	分配一块可移动的内存区域。实际上内存块在物理内存中是不可能移动的，这里的可移动指的是在应用程序的默认逻辑堆内可以移动。返回值是内存对象的句柄。可以通过调用 GlobalLock()函数将一个句柄转化为一个指针。这个标志不能和 GMEM_FIXED 同时使用
GMEM_ZEROINIT 0x0040	初始化内存对象为全 0。如果不用这个标志，内存对象将为不确定的内容
GHND 0x0042	GMEM_MOVEABLE 和 GMEM_ZEROINIT 标志联合使用，即可移动同时初始化为 0
GPTR 0x0040	GMEM_FIXED 和 GMEM_ZEROINIT 标志联合使用，即不可移动同时初始化为 0

如果该参数指定为 0，将使用默认的标志 GMEM_FIXED。

- 参数 dwBytes 指定内存对象的字节数。

如果指定大小为 0，而且使用 GMEM_MOVEABLE 标志请求分配内存，那么函数返回值将是一个指向标记为被舍弃的内存对象的句柄。如果函数成功执行，将分配至少指定大小的内存。调用 GlobalSize()函数可查询分配的内存真正大小。如果堆空间不足，将返回 NULL，表示分配失败。

16.1.6 GlobalLock()函数

该函数锁定一个内存对象，并将这个内存对象的锁定计数器值加 1。函数原型如下：

```
LPVOID GlobalLock(
    HGLOBAL hMem
);
```

参数 hMem 为一个指向全局内存对象的句柄，这个句柄必须通过GlobalAlloc()或GlobalReAlloc()函数创建。

如果函数成功执行，将返回一个指向内存对象起始地址的指针，通过这个指针，应用程序就可以对其进行访问；否则返回 NULL，表示锁定失败。

系统为每个全局内存对象都维护一个锁定技术器，初始化为 0。调用 GlobalLock()函数将使计数器加 1，调用 GlobalUnLock()函数将使计数器减 1。一旦锁定计数器值大于 0，那么这块内存区域将不允许被移动或删除，只有当值回到 0 时，才解除对这块内存对象的锁定。所以一旦对内存对象访问完毕，一定要配套调用 GlobalUnLock()函数。

如果在分配内存对象时指定为 GMEM_FIXED 属性，那么计数器值将一直保持为 0。

16.1.7 GlobalUnLock()函数

与 GlobalLock 函数对应，将属性为 GMEM_MOVEABLE 的内存对象的锁定计数器值减 1。对 GMEM_FIXED 属性的内存对象没有任何影响（因为计数器值一直为 0 ）。函数原型如下：

```
BOOL GlobalUnlock(
    HGLOBAL hMem
);
```

如果函数成功执行，则返回非 0（True）；否则返回 0（False）。

参数 hMem 为一个指向全局内存对象的句柄，这个句柄必须通过GlobalAlloc()或GlobalReAlloc()函数创建。

16.1.8 GetClipboardData()函数

该函数用于从剪贴板读取（或接收）特定格式的数据，在读取之前，必须先打开剪贴板，但不要求成为剪贴板的所有者。函数原型如下：

```
HANDLE GetClipboardData(
    UINT uFormat
);
```

如果函数成功执行，则返回一个指向剪贴板内特定对象的句柄；否则返回 NULL 表示出错。

参数 uFormat 指定所要读取的数据的格式。详见 SetClipboardData()函数的说明。

16.1.9 一个利用剪贴板在不同进程之间交换数据的示例

下面通过一个简单的示例来讲解怎样将数据写入剪贴板，以及怎样从剪贴板内读取数据。这是一

个基于 MFC 对话框的程序，程序运行界面如图 16.1 所示。

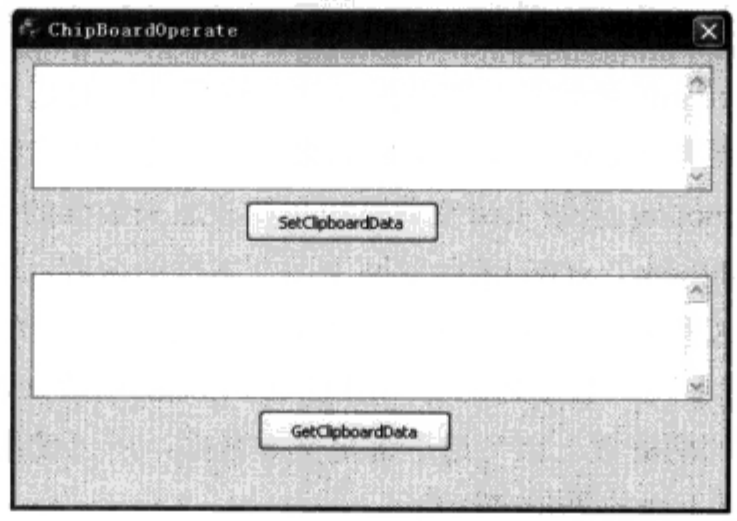


图 16.1　剪贴板操作的程序设计界面

在上面的文本框中输入数据，然后单击 SetClipboardData 按钮打开剪贴板，将该文本框中的数据写入剪贴板，最后关闭剪贴板。而单击下面的 GetClipboardData 按钮将打开剪贴板，从剪贴板内读取数据，显示到下面的文本框中，最后关闭剪贴板。这个程序相当简单，先来看 SetClipboardData 按钮的单击事件处理函数，如例 16.1 所示。

例 16.1

```
void CChipBoardOperateDlg::OnBnClickedButtonSetclipboarddata()
{
    // TODO: Add your control notification handler code here
    if (!OpenClipboard())
    {
        AfxMessageBox(_T("Cannot open the Clipboard"));
        return;
    }
    // 将剪贴板上的内容清空
    if(!EmptyClipboard())
    {
        AfxMessageBox(_T("Cannot empty the Clipboard"));
        return;
    }
    // 取回当前控件的数据，hData 开辟全局内存区域，存放数据
    CStringW str;
    this->GetDlgItemTextW(IDC_EDIT_COPYCB,str);
    size_t cbStr = (str.GetLength() + 1) * sizeof(TCHAR);
    HGLOBAL hData = GlobalAlloc(GMEM_MOVEABLE, cbStr);
    memcpy_s(GlobalLock(hData), cbStr, str.LockBuffer(), cbStr);
    GlobalUnlock(hData);
    str.UnlockBuffer();
    // 选择合适的格式
    UINT uiFormat = (sizeof(TCHAR) == sizeof(WCHAR)) ? CF_UNICODETEXT : CF_TEXT;
    if (::SetClipboardData(uiFormat, hData) == NULL)
    {
        AfxMessageBox(_T("Unable to set Clipboard data"));
        CloseClipboard();
        return;
```

```
    }
    CloseClipboard();
}
```

这里首先调用 OpenClipboard()函数打开剪贴板（注意这里使用的是 CWnd 的成员函数 OpenClipboard()，因为对话框类就是继承自 CWnd 类，所以如果打开成功，则将剪贴板关联到本窗口），如果打开成功，则调用 EmptyClipboard()函数来清空剪贴板（释放原来数据的空间），从而使自己成为剪贴板的所有者。接着调用 GetDlgItemTextW()函数取回上面文本框的内容，然后调用 GlobalAlloc()函数创建一个全局内存对象，注意这里必须使用 GMEM_MOVEABLE 参数。然后调用 memcpy_s()函数将数据复制到全局内存对象，当然在写入数据之前要调用 GlobalLock()函数来锁定这个句柄，操作完成后再调用 GlobalUnLock()函数将句柄解锁。因为这里都是文本数据，所以使用文本格式，但是要判断一下是否使用 Unicode 文本，通过下面的语句来判断：

```
UINT uiFormat = (sizeof(TCHAR) == sizeof(WCHAR)) ? CF_UNICODETEXT : CF_TEXT;
```

最后调用 SetClipboardData()函数将数据写入剪贴板，操作完成之后调用 CloseClipboard()函数来关闭剪贴板。

下面介绍 GetClipboardData 按钮的单击事件处理函数，该按钮用于从剪贴板获取数据，并显示到下面的文本框中，如例 16.2 所示。

例 16.2

```
void CChipBoardOperateDlg::OnBnClickedButtonGetclipboarddata2()
{
    // TODO: Add your control notification handler code here
    if (!OpenClipboard())
    {
        AfxMessageBox(_T("Cannot open the Clipboard"));
        return;
    }
     // 选择合适的格式
    UINT uiFormat = (sizeof(TCHAR) = = sizeof(WCHAR)) ? CF_UNICODETEXT : CF_TEXT;
    HANDLE hData = ::GetClipboardData(uiFormat);
    if( hData == NULL ){
        AfxMessageBox(_T("Unable to get Clipboard data"));
        CloseClipboard();
        return;
    }
    CStringW str;
    LPCTSTR lpdata = (LPCTSTR)GlobalLock(hData);
    this->SetDlgItemTextW(IDC_EDIT_PASTECB,lpdata);
    GlobalUnlock(hData);
    CloseClipboard();
}
```

这里首先打开剪贴板，然后设置合适的格式，调用 GetClipboardData()函数取回剪贴板上的数据的句柄，将数据显示到下面的控件中，最后将剪贴板关闭。

编译并运行程序，在上面的文本框中输入一些数据，如“这是一个操作剪贴板的例子”，为了显示操作结果，这里打开 Office 剪贴板。然后单击 SetClipBoardData 按钮，可以看到数据被写入剪贴板。然后单击 GetClipboardData 按钮，就会将刚才放入剪贴板的数据读出，并显示在下面的文本框中。该程序的运行结果如图 16.2 所示。

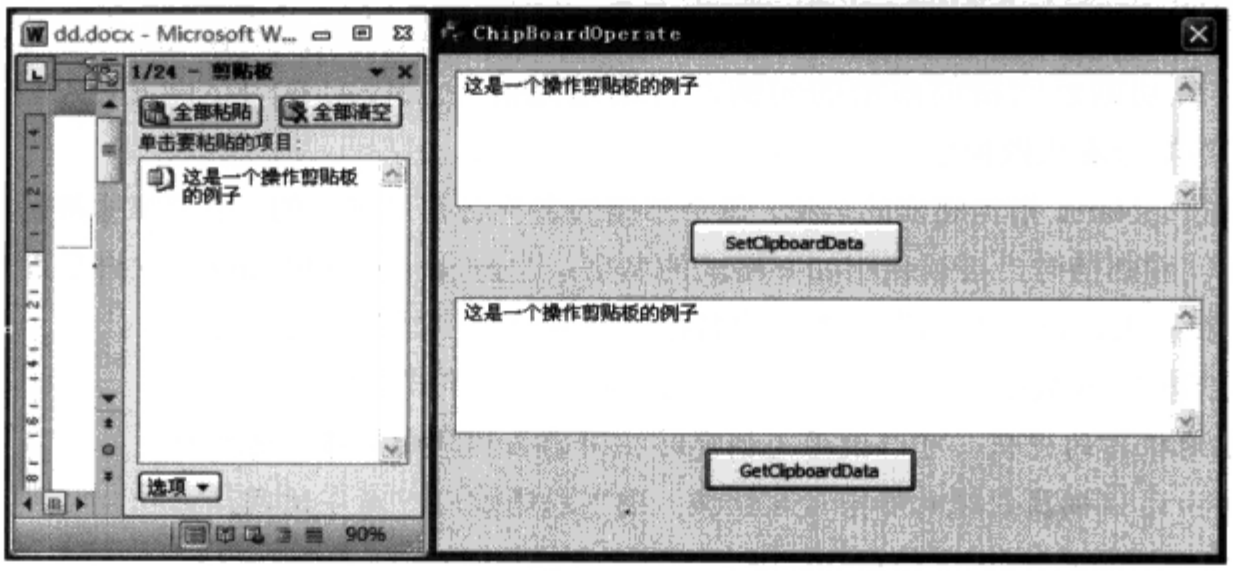


图 16.2　剪贴板操作的运行结果

16.2　邮槽通信

邮槽是一个内置在内存中的伪文件，可以使用标准的文件函数来访问（读出或写入数据）。邮槽既可用于同一台机器上不同进程间的通信，也可用于通过网络在不同的机器之间传递数据。放入邮槽的数据可以是任意形式，但是如果数据是在不同的计算机之间传输，那么长度不能超过 424 字节。不像那些真正存放在磁盘上的文件，邮槽只是临时的，一旦所有指向邮槽的句柄都被关闭（引用计数变为 0），那么该邮槽以及存放在该邮槽内的所有数据都将被系统释放。

邮槽提供了一种在进程间单向通信的机制。这里涉及服务器和客户端的概念。

- 邮槽的服务器就是创建邮槽从而成为邮槽所有者的进程，所有的邮槽都只是局部存在于创建它的进程空间内，进程不能创建远程邮槽。一旦一个进程创建了一个邮槽，就将获取该邮槽的句柄，只有通过该句柄才能从邮槽内接收数据。只有邮槽的创建进程或通过其他机制获取到该邮槽句柄的进程才能从邮槽内读出数据。也就是说，只有邮槽的服务器能从邮槽内接收数据。
- 所有将数据写入一个邮槽的进程都可以称做这个邮槽的客户端，只要获知邮槽的名称，进程就可以向其存放数据。新的数据将被放到邮槽内原来数据的后面，而不会将原来的数据覆盖掉。

使用邮槽的最大好处就是可以在一个域（domain）内广播消息。如果处于同一个域内的进程都使用相同的名称创建邮槽，那么每个写入具有该名称的邮槽的数据都将被发送到该域内的所有参与进程。一个进程既可以成为一个邮槽的服务器，也可以成为一个邮槽的客户端，所以在同一个域内利用邮槽实现一个消息广播设备将会非常简单。注意在不同的机器间发送的数据长度不能超过 424 字节，如果想发送更多的数据，可使用命名管道或 Windows Sockets 来实现。

下面介绍一些与邮槽通信密切相关的函数。

16.2.1　CreateMailslot()函数

该函数用于创建一个具有指定定名称的邮槽，并返回一个指向该邮槽的句柄。如果已经存在一个具有相同名称的邮槽，则创建失败。函数原型如下：

```
HANDLE CreateMailslot(
    LPCTSTR lpName,
    DWORD nMaxMessageSize,
    DWORD lReadTimeout,
```

```
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

如果创建成功则返回指向邮槽的句柄，否则返回值为 INVALID_HANDLE_VALUE，可通过 GetLastError()函数查看失败原因。

第一个参数 lpName 指定邮槽的名称，每个邮槽的名称必须是惟一的。由于邮槽都是局部于本地计算机的，因此创建邮槽时，必须使用如下格式的名字：\\.\mailslot\[path]name，允许后面包含多级的伪目录，如“\\.\mailslot\abc\def\ghi”。邮槽的名称是不区分大小写的，这点要注意。

如果想向邮槽写入数据，则不同情况应使用不同的名字格式访问。

- 对于本地机器的进程，可以使用与创建时相同的名称来向邮槽发送数据。
- 如果要向远程机器创建的邮槽写入数据，则可以通过以下名字访问。

  \\ComputerName\mailslot\[path\]name;
- 如果想向处于同一个域内的所有拥有相同名称的邮槽广播写入数据，则使用如下名字访问。

  \\DomainName\mailslot\[path\]name
- 如果想往系统中所有具有相同名字的邮槽都写入数据（不用考虑它们所处的域），则使用如下名字。

  *\mailslot\[path\]name

第二个参数 nMaxMessageSize 指定每次向邮槽写入数据的最大长度，单位为字节。如果设置为 0，则表示对消息的长度没有限制。

第三个参数 lReadTimeout 指定读操作最多能等待的将消息写入邮槽的超时时间，单位为毫秒（ms）。如果值为 0，则表示如果邮槽内没有数据到达将马上返回；如果值为 MAILSLOT_WAIT_FOREVER(−1)，则表示读操作将一直（阻塞）等待直到数据到达。

第四个参数 lpSecurityAttributes 为一个指向 SECURITY_ATTRIBUTES 结构体的指针，代表创建邮槽的安全属性。如果值为 NULL，则表示该邮槽句柄不能被子进程继承。

16.2.2 GetMailslotInfo()函数

该函数用于返回指定邮槽的信息，这些信息包括：单条消息的最大长度，邮槽内下一条消息的长度，邮槽内消息的个数，以及读操作等待消息的超时时间。函数原型如下：

```
BOOL GetMailslotInfo(
    HANDLE hMailslot,
    LPDWORD lpMaxMessageSize,
    LPDWORD lpNextSize,
    LPDWORD lpMessageCount,
    LPDWORD lpReadTimeout
);
```

如果函数成功执行，则返回非 0（True），如果失败则返回 0（False）。

- 第一个参数 hMailslot 指定邮槽的句柄。
- 第二个参数 lpMaxMessageSize 是返回值，存放该邮槽所能允许的每条消息的最大长度，注意这里取回的值可能会超过在创建邮槽时指定的值。该参数可以为 NULL。
- 第三个参数 lpNextSize 返回下一条消息的长度（单位为字节），如果值为特殊的 MAIL SLOT_NO_MESSAGE(−1)，则表示邮槽内还没有下一条消息。该参数可以为 NULL。
- 第四个参数 lpMessageCount 返回邮槽内等待被读取的消息个数。该参数可以为 NULL。
- 第五个参数 lpReadTimeout 表示读操作可以等待一条消息写入邮槽的超时时间。该参数可以为

NULL。

后面 4 个参数都是返回值，如果对某个返回值不感兴趣，就对其传入 NULL 值。例如，只想查询下一条消息的长度和总的消息个数，可以采用如下调用：

```
GetMailslotInfo(m_hMailslot,NULL,&m_iNextSize,&m_iMsgCount,NULL);
```

16.2.3　SetMailslotInfo()函数

该函数用于设置指定邮槽的读操作等待消息写入邮槽的超时时间。函数原型如下：

```
BOOL GetMailslotInfo(
    HANDLE hMailslot,
    DWORD lReadTimeout,
);
```

如果设置成功，则返回 True，否则返回 False。

- 参数 hMailslot 指定邮槽的句柄。
- 参数 lReadTimeout 指定新的读等待超时时间。

16.2.4　ReadFile()函数

在这里，该函数用于服务器从邮槽内以文件的形式读取数据。函数原型如下：

```
BOOL ReadFile(
    HANDLE hFile,
    LPVOID lpBuffer,
    DWORD nNumberOfBytesToRead,
    LPDWORD lpNumberOfBytesRead,
    LPOVERLAPPED lpOverlapped
);
```

如果读取成功，则返回非 0（True）；否则返回 0（False）。

- 第一个参数 hFile 为数据源，这里就是邮槽的句柄。
- 第二个参数 lpBuffer 指定接收数据的缓冲区。
- 第三个参数 nNumberOfBytesToRead 指定本次读操作从邮槽中读出的最大字节数。
- 第四个参数 lpNumberOfBytesRead 是返回值，返回本次从邮槽中读出的实际字节数。
- 第五个参数 lpOverlapped 为一个指向 OVERLAPPED 结构的指针，表示采用重叠 I/O 机制，允许通过异步方式读取数据。

16.2.5　GetFileTime()函数和 SetFileTime()函数

GetFileTime()函数用于查询邮槽伪文件的创建，上次被访问及上次被修改时间。该函数的原型如下：

```
BOOL GetFileTime(
    HANDLE hFile,
    LPFILETIME lpCreationTime,
    LPFILETIME lpLastAccessTime,
    LPFILETIME lpLastWriteTime
);
```

如果查询成功则返回非 0（True），否则返回 0（False）。

- 第一个参数 hFile 为一个指向文件的句柄，这里指的是邮槽句柄。

- 后面 3 个参数都是返回值，都是指向 FILETIME 结构体的指针，分别返回邮槽的创建时间、上次被访问时间及上次被写入时间。如果用户对某些信息不感兴趣，可将对应参数设置为 NULL。

SetFileTime()函数用于重新设置邮槽伪文件的创建、上次被访问及上次被修改时间。该函数的原型如下：

```
BOOL SetFileTime(
    HANDLE hFile,
    const FILETIME*lpCreationTime,
    const FILETIME*lpLastAccessTime,
    const FILETIME*lpLastWriteTime
);
```

其参数含义与上面相同，如果用户不想修改其中的某个值，只需设置为 NULL 即可。

16.2.6 CreateFile()函数

该函数用于打开或创建一个文件、文件流、目录、物理磁盘、缓冲区、邮槽、命名管道等对象。这里指的是打开一个邮槽，准备向邮槽写入数据。该函数只用于客户端，函数原型如下：

```
HANDLE CreateFile(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDispostion ,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);
```

如果函数成功执行，将返回一个用于向邮槽写入数据的句柄；如果失败，则返回值为 INVALID_HANDLE_VALUE。

- 第一个参数 lpFileName 指定要访问的邮槽的名称。
- 第二个参数 dwDesiredAccess 指定访问的方式：只读、只写或读写。在这里只能设置为 GENERIC_WRITE，表示只写，因为客户端只能向邮槽写入数据。
- 第三个参数 dwShareMode 指定打开对象的共享模式：共享只读、只写、两者均禁止或两者均可。在邮槽操作中，该参数必须指定为 FILE_SHARE_READ，表示共享只读，这样允许服务器从邮槽内读出数据。
- 第四个参数 lpSecurityAttributes 设置访问的安全属性，对邮槽操作没有影响，设置为 NULL。
- 第五个参数 dwCreationDispostion 指定在文件存在或不存在时的动作，表 16.3 列出了可取的值。在邮槽的操作中，对于客户端一般选用 OPEN_EXISTING 参数，打开已存在的邮槽，并将写入的数据放到邮槽内原来数据的后面。

表 16.3　　创建文件的动作

动　作	说　明
CREATE_ALWAYS	总是创建一个新的文件，如果文件已经存在，将清空并覆盖原文件的所有内容
CREATE_NEW	如果文件不存在，则创建一个新的文件；如果已经存在，则调用失败
OPEN_ALWAYS	总是将文件打开；如果文件不存在，则新建一个文件并打开
OPEN_EXISTING	仅当文件存在时才打开，否则调用失败
TRUNCATE_EXISTING	打开一个已存在的文件，并将文件截断，这样文件大小将变为 0。如果文件不存在，则调用失败

- 第六个参数 dwFlagsAndAttributes 指定文件的属性和标志。对于邮槽，使用一般属性 FILE_ATTRIBUTE_NORMAL。
- 第七个参数 hTemplateFile 为一个指向临时文件的句柄，在邮槽操作中没有意义，设置为 NULL。

16.2.7　WriteFile()函数

该函数用于在客户端将数据写入邮槽，注意如果是向其他机器的邮槽写入数据，长度不能超过 424 字节。函数原型如下：

```
BOOL WriteFile(
    HANDLE hFile,
    LPCVOID lpBuffer,
    DWORD nNumberOfBytesToWrite,
    LPDWORD lpNumberOfBytesWritten,
    LPOVERLAPPED lpOverlapped
);
```

如果写入成功，则返回非 0（True）；写入失败返回 0（False）。

- 第一个参数 hFile 必须是调用 CreateFile()函数成功返回的代表邮槽的文件句柄。
- 第二个参数 lpBuffer 指定存放写入数据的缓冲区。
- 第三个参数 nNumberOfBytesToWrite 指定本次写入数据的最大长度。
- 第四个参数 lpNumberOfBytesWritten 为返回值，返回实际写入的数据的字节数。
- 第五个参数 lpOverlapped 为一个指向 OVERLAPPED 结构的指针，表示采用重叠 I/O 机制，允许通过异步方式向邮槽写入数据。

16.2.8　CloseHandle()函数

该函数用于在客户端将数据写入邮槽完毕后，关闭对应的文件句柄（由 CreateFile()函数返回）。函数原型如下：

```
BOOL CloseHandle(
    HANDLE hObject
);
```

如果关闭句柄成功，则返回非 0（True）；否则返回 0（False）。

参数 hObject 为一个指向打开对象的合法句柄，这里指的是指向打开的邮槽的文件句柄。

16.2.9　一个利用邮槽在不同进程间通信的示例

下面通过一个简单的示例讲讲怎样利用邮槽在进程之间传递数据，确切地说是客户端将数据写入邮槽，而服务器端从邮槽中将数据读出。

先看服务器端的实现，服务器端的运行界面如图 16.3 所示。

这是一个基于对话框的应用程序，在对话框初始化时会创建一个邮槽，之后单击下面的 Read Mailslot 按钮，将从邮槽中读出所有消息，然后显示到上面的控件中。

服务器端的主要成员如下：

```
protected:
    CStringW m_strSlotName;
    HANDLE m_hSlot;
```

```
public:
    afx_msg void OnBnClickedButtonRecvfrommailslot();
```

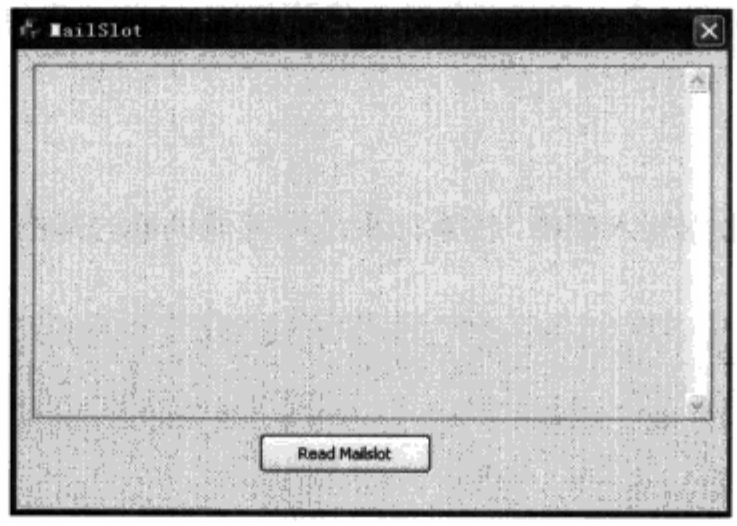


图 16.3　邮槽服务器端程序界面

成员 m_strSlotName 存放邮槽的名称，成员 m_hSlot 为指向该邮槽的句柄。它们在构造函数内的初始代码如下：

```
CMailSlotDlg::CMailSlotDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CMailSlotDlg::IDD, pParent)
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    m_strSlotName = _T("\\\\.\\mailslot\\my_mailslot");
    m_hSlot = NULL;
}
```

这里指定邮槽的名称为“\\\\.\\mailslot\\my_mailslot”，因为在 C++中字符“\”是特殊字符，所以需要用转义字符“\\”来表示字符“\”。

真正创建邮槽的操作放到 OnInitDialog()成员函数内进行，主要代码如下：

```
//创建邮槽
    m_hSlot = ::CreateMailslotW(m_strSlotName,0,
        MAILSLOT_WAIT_FOREVER,NULL);
    if( m_hSlot == INVALID_HANDLE_VALUE ){
        CStringW errMsg;
        errMsg.Format(_T("CreateMailslot failed with %d"),::GetLastError());
        AfxMessageBox(errMsg);
        ::PostQuitMessage(-8);
        return FALSE;
    }
```

这里首先调用 CreateMailslotW()函数（因为程序默认支持 Unicode，所以函数名对应变为 CreateMailslotW，与 CreateMailslot()函数作用相同）创建指定名称的邮槽，并将新建的邮槽的句柄存放到 m_hSlot 中，以后从邮槽读取数据都将通过该句柄进行。如果创建失败，首先弹出一个消息框显示创建邮槽失败的错误码，从 GetLastError()函数可以获知，然后调用 PostQuitMessage()函数向程序发送退出的消息，这样程序就会在创建失败后退出，参数-8 表示程序的退出码。

接下来就是从邮槽内读取数据的 Read Mailslot 按钮的单击事件处理函数，如例 16.3 所示。

例 16.3

```
void CMailSlotDlg::OnBnClickedButtonRecvfrommailslot()
{
```

```
// TODO: Add your control notification handler code here
DWORD cbMessage, cMessage, cbRead;
BOOL fResult;
LPTSTR lpszBuffer;
TCHAR achID[80];
DWORD cAllMessages;
cbMessage = cMessage = cbRead = 0;
fResult = GetMailslotInfo( m_hSlot,              // 邮槽句柄
    (LPDWORD) NULL,                              // 不关注消息最大长度
    &cbMessage,                                  // 下一条消息的长度
    &cMessage,                                   // 邮槽内消息个数
    (LPDWORD) NULL);                             // 不关注读超时
if (!fResult)
{
    //printf("GetMailslotInfo failed with %d.\n", GetLastError());
    CStringW errMsg;
    errMsg.Format(_T("GetMailslotInfo failed with %d"), ::GetLastError());
    AfxMessageBox(errMsg);
    return;
}
if (cbMessage == MAILSLOT_NO_MESSAGE)
{
    //当前邮槽内没有数据
    CStringW errMsg;
    errMsg.Format(_T("当前邮槽内没有数据可读"));
    AfxMessageBox(errMsg);
    return;
}
cAllMessages = cMessage;
while (cMessage != 0)                            // 每次读回一条消息，直到读完
{
    //分配接收缓冲区的空间，就是下一条消息的长度
    lpszBuffer = (LPTSTR) GlobalAlloc(GPTR,cbMessage);
    if( NULL == lpszBuffer )
        return ;
    lpszBuffer[0] = '\0';
    //从邮槽内读回下一条消息
    fResult = ReadFile(m_hSlot,
        lpszBuffer,
        cbMessage,
        &cbRead,
        NULL);
    if (!fResult)
    {
        CStringW errMsg;
        errMsg.Format(_T("ReadFile failed with %d"), ::GetLastError());
        AfxMessageBox(errMsg);
        GlobalFree((HGLOBAL) lpszBuffer);
        return;
```

```
        }
        //将数据显示到控件
        CStringW mailslotMsg(lpszBuffer);
        CStringW wholeMsg;
        this->GetDlgItemTextW(IDC_EDIT_RECVDATA,wholeMsg);
        wholeMsg += mailslotMsg + _T("\r\n");
        this->SetDlgItemTextW(IDC_EDIT_RECVDATA,wholeMsg);
        GlobalFree((HGLOBAL) lpszBuffer);
        fResult = GetMailslotInfo( m_hSlot,              // 邮槽句柄
              (LPDWORD) NULL,                            // 不关注消息最大长度
              &cbMessage,                                // 下一条消息的长度
              &cMessage,                                 // 邮槽内消息个数
              (LPDWORD) NULL);                           // 不关注读超时
        if (!fResult)
        {
              CStringW errMsg;
              errMsg.Format(_T("GetMailslotInfo failed with %d"), ::GetLastError());
              AfxMessageBox(errMsg);
              return;
        }
    }
    CStringW wholeMsg;
        this->GetDlgItemTextW(IDC_EDIT_RECVDATA,wholeMsg);
        wholeMsg += _T("-----------------------------------------------\r\n");
        this->SetDlgItemTextW(IDC_EDIT_RECVDATA,wholeMsg);
}
```

这里首先调用 GetMailslotInfo()函数来获取当前邮槽内下一条消息的长度和总的消息个数，如果没有任何消息就直接返回，否则将调用 ReadFile()函数每次从邮槽内取回一条消息，并拼接到文本框输出，同时继续查询下一条消息的长度，然后读下一条消息，直到邮槽内没有任何消息为止。前面介绍过，客户端向邮槽内写入数据时会以一条消息的形式接到原数据的最后，而且邮槽会自动维护消息列表及消息个数。读数据时就可以每次读一条消息，这样可以保证读出来的数据的正确性。

下面介绍邮槽客户端的实现，客户端运行界面如图 16.4 所示。

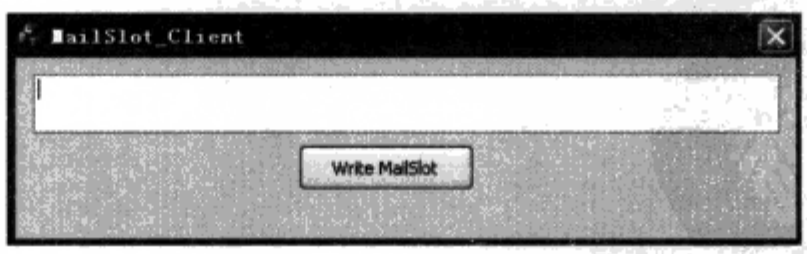


图 16.4　邮槽客户端程序界面

上面的文本框接收用户的输入，单击 Write MailSlot 按钮即可将这些数据写入邮槽，服务器端就可以读出这些数据了。这里的主要操作都在 Write MailSlot 按钮的单击事件处理函数内，如例 16.4 所示。

例 16.4

```
void CMailSlot_ClientDlg::OnBnClickedButton1Writemailslot()
{
    // TODO: Add your control notification handler code here
    CStringW SlotName(_T("\\\\.\\mailslot\\my_mailslot"));
    HANDLE hFile;
```

```
    //打开邮槽，准备写入数据
    hFile = ::CreateFileW(SlotName,
            GENERIC_WRITE,
            FILE_SHARE_READ,
            (LPSECURITY_ATTRIBUTES) NULL,
            OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL,
            (HANDLE) NULL
    );
    if (hFile == INVALID_HANDLE_VALUE)
    {
        //printf("CreateFile failed with %d.\n", GetLastError());
        CStringW errMsg;
        errMsg.Format(_T("CreateFile failed with %d"),::GetLastError());
        AfxMessageBox(errMsg);
        return ;
    }
    //得到用户输入的数据
    CStringW inputText;
    this->GetDlgItemTextW(IDC_EDIT_SENDDATA,inputText);
    BOOL fResult;
    DWORD cbWritten;
     //将数据写入邮槽
    fResult = ::WriteFile(hFile,
            (LPCTSTR)inputText,
            (DWORD) (inputText.GetLength()+1)*sizeof(TCHAR),
            &cbWritten,
            (LPOVERLAPPED) NULL);
        if (!fResult)
        {
            CStringW errMsg;
            errMsg.Format(_T("WriteFile failed with %d"),::GetLastError());
            AfxMessageBox(errMsg);
            CloseHandle(hFile);
            return ;
        }
         this->SetDlgItemTextW(IDC_EDIT_SENDDATA,_T(""));
         //关闭文件句柄
         CloseHandle(hFile);
    }
```

这里首先调用 CreateFileW()函数打开邮槽，并取得访问的文件句柄。注意这里在打开邮槽时指定的邮槽名必须与服务器端创建的邮槽名相同，这样才能进行通信。在得到用户输入数据后就调用 WriteFile()函数将数据写入邮槽。操作完成后一定要调用 CloseHandle()函数将文件句柄关闭。

客户端和服务器端都运行后，在客户端程序内写入一些数据，单击 Write MailSlot 按钮将数据写入邮槽，再在服务器端单击 Read MailSlot 按钮就可以获取这个数据。如果多次从客户端向邮槽写入数据，那么服务器端可以将每个消息区分开来。客户端及服务器的运行结果如图 16.5 所示。可以看到，在客户端多次往邮槽内写入数据，然后在服务器读取数据一次，每一条消息都被正确接收，各个消息以一

行显示。一次读取所有消息完毕，以虚横线分割。

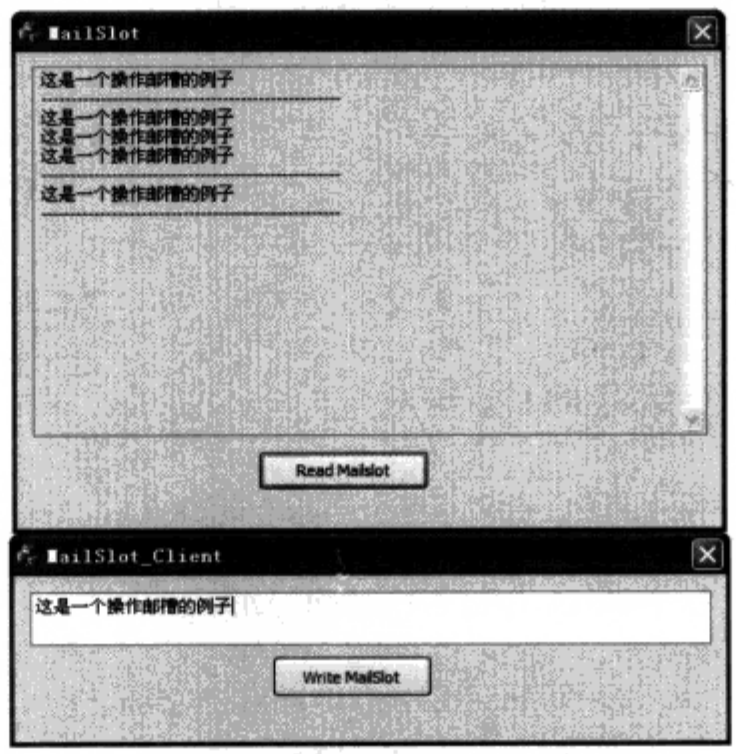


图 16.5　利用邮槽在服务器和客户端之间通信的运行结果

16.3　匿名管道通信

管道是用于在进程间通信的一块共享内存。创建管道的进程称为管道服务器，而连接上管道的进程称为管道客户端。不同于邮槽，管道提供了一种支持双向的进程间通信的机制，既可以支持单向的数据传输，也可以支持双向传输。管道分为匿名管道和命名管道。匿名管道只支持在父子进程之间传输数据，是无名字的、单向的，而且只局限于本地，不能用于通过网络（在不同的机器间）进行通信。而命名管道在创建时必须指定一个特殊格式的名字，管道的服务器和客户机之间既可以单向传输，也可以双向传输。

利用管道在进程间通信的基本过程是一个进程将数据写入管道，而另一个进程从管道内读出数据。同邮槽一样，管道也是驻留在内存中的伪文件，可以通过访问文件的 ReadFile()及 WriteFile()函数来读取管道内数据及将数据写入管道。这一节先介绍匿名管道的通信，下一节介绍命名管道。首先介绍与匿名管道通信相关的几个函数。

16.3.1　CreatePipe()函数

该函数用于创建一个匿名管道，并返回管道的写入（发送）端和读出（接收）端。函数原型如下：

```
BOOL CreatePipe(
    PHANDLE hReadPipe,
    PHANDLE hWritePipe,
    LPSECURITY_ATTRIBUTES lpPipeAttributes,
    DWORD nSize
);
```

如果函数成功执行，则返回非 0（True），如果失败则返回 0（False），通过 GetLastError()函数来查看失败原因。

- 第一个参数 hReadPipe 返回管道的读出端句柄，进程通过这个句柄调用 ReadFile()函数从管道内读取数据。

- 第二个参数 hWritePipe 返回管道的写入端句柄，进程通过这个句柄调用 WriteFile()函数向管道内写入数据。
- 第三个参数 lpPipeAttributes 是一个指向 SECURITY_ATTRIBUTES 结构体的指针，设置安全属性，这里主要关注返回的管道句柄是否可以被子进程继承。如果该值被指定为 NULL，表示返回的句柄不能被子进程继承。

结构体 SECURITY_ATTRIBUTES 主要用于对新创建的句柄设置是否可以在子进程内继承的属性。该结构体有 3 个分量，定义如下：

```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD nLength;
    LPVOID lpSecurityDescriptor;
    BOOL bInheritHandle;
}SECURITY_ATTRIBUTES,
*PSECURITY_ATTRIBUTES,
*LPSECURITY_ATTRIBUTES;
```

其中第一个分量nLength指定结构体的大小，单位为字节；第二个分量lpSecurityDescriptor是一个指向安全描述符的指针，该安全描述符控制共享属性，对于管道不用关注该分量，将其设置为 NULL，表示使用系统默认的安全描述符；第三个参数bInheritHandle是一个 BOOL 值，表示当一个新的进程被创建时，该句柄是否可以被子进程继承，为了利用管道句柄在父子进程之间通信，必须将该分量设置为 True，表示子进程可以继承管道句柄。

- 第四个参数 nSize 设置管道的缓冲区的大小，单位为字节。注意这只是参考值，系统根据该值计算出一种合适的缓冲机制。如果该参数指定为 0，表示系统将使用默认的缓冲大小。

16.3.2　CreateProcess()函数

该函数用于创建一个新的进程及其主线程，这个新的进程（子进程）将在调用进程（父进程）的安全上下文下运行。函数原型如下：

```
BOOL WINAPI CreateProcess(
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

如果创建子进程成功，则返回值为非 0（True），创建失败则返回 0（False）。

- 第一个参数 lpApplicationName 指定将由子进程绑定运行的程序的名称。该字符串既可以是可执行程序的完整路径和文件名，也可以使用部分路径。如果使用部分路径，那么将以父进程所处的当前目录作为起始点搜索可执行文件名，而不会使用搜索路径进行搜索。如果没有指定文件的扩展名，那么系统默认将使用.exe 扩展名。所以如果一个文件的扩展名是.com，那么这个参数一定要包括扩展名，否则会出错。当然对于 EXE 文件，可以不指定扩展名。

这个参数可以指定为 NULL，这时的可执行程序名字将是lpCommandLine传递的第一个字符串，每个字符串之间用空格分隔（与传递命令行参数的格式一致）。如果使用了包含空格的长文件名，那么应使用双引号将文件名包含起来，以指示文件名的结束和参数的开始，否则可能出错。如字符串“c:\program files\sub dir\program name”，中间多个空格，如果不使用双引号包含起来，系统将按下面的次序得到多种可能的解析：

```
c:\program.exe files\sub dir\program name
c:\program files\sub.exe dir\program name
c:\program files\sub dir\program.exe name
c:\program files\sub dir\program name.exe
```

如果想执行批处理文件，必须启动命令解析器；将参数lpApplicationName指定为 cmd.exe 并将参数 lpCommandLine 指定为批处理文件的名称。

- 第二个参数lpCommandLine指定传入可执行程序的命令行参数，该参数的最大长度是 32 KB 个字符。如果lpApplicationName参数为 NULL，那么lpCommandLine包含的第一个字符串就是可执行程序名，这部分的长度被限制在 MAX_PATH 字节以内。

该参数可以指定为 NULL，那么将使用lpApplicationName参数的除可执行文件名外的其余部分作为命令行参数。如果lpApplicationName参数和lpCommandLine参数都不为 NULL，那么将使用 lpApplicationName 参数内以 null 终止的字符串作为可执行程序名，而将lpCommandLine参数内以 null 终止的字符串作为命令行参数，可以包含多个参数，各个参数以空格隔开。在新的进程内可以调用 GetCommandLine()函数来获得整个命令行参数。对于 C 语言的控制台进程，可以通过访问 main()函数的 argc 和 argv 参数来获得命令行参数。

- 第三个参数lpProcessAttributes是一个指向 SECURITY_ATTRIBUTES 结构体的指针，以决定是否将返回的句柄继承给将来父进程创建的其他子进程。如果指定为 NULL，句柄将不能被其他子进程继承。
- 第四个参数lpThreadAttributes也是一个指向 SECURITY_ATTRIBUTES 结构体的指针，含义与上面相同，不过这是针对子进程的主线程属性，每个进程都必须包含一个主线程才能运行。
- 第五个参数bInheritHandles为一个 BOOL 值，如果值为 True，那么每个调用进程（父进程）内的可被继承的句柄都将被新的进程（子进程）继承；如果为 False，所有句柄都将不能被继承。注意被继承的句柄与原句柄具有相同的值和相同的访问权限。在这里将该参数设置为 True，使子进程能访问到管道句柄。
- 第六个参数dwCreationFlags指定进程的优先级类型和创建的附加标志。这里都将该参数设置为 0，表示 NORMAL_PRIORITY_CLASS，这样子进程与父进程具有相同的优先级类型。
- 第七个参数lpEnvironment为一个指向环境块的指针，用于设置新进程的环境变量。如果指定为 NULL，那么子进程将使用与父进程相同的环境。
- 第八个参数lpCurrentDirectory指定子进程的当前目录，并须使用完整路径。如果指定为 NULL，表示将使用父进程的当前目录作为新进程的当前目录。
- 第九个参数lpStartupInfo为一个指向 STARTUPINFO 结构体的指针，指定子进程在创建时的窗口状态、桌面、标准句柄及主窗口的显示。STARTUPINFO 结构体的定义如下：

```
typedef struct _STARTUPINFO {
    DWORD cb;
    LPTSTR lpReserved;
    LPTSTR lpDesktop;
    LPTSTR lpTitle;
    DWORD dwX;
```

```
    DWORD dwY;
    DWORD dwXSize;
    DWORD dwYSize;
    DWORD dwXCountChars;
    DWORD dwYCountChars;
    DWORD dwFillAttribute;
    DWORD dwFlags;
    WORD wShowWindow;
    WORD cbReserved2;
    LPBYTE lpReserved2;
    HANDLE hStdInput;
    HANDLE hStdOutput;
    HANDLE hStdError;
} STARTUPINFO,
*LPSTARTUPINFO;
```

该结构体的成员很多，这里只讲解与应用有关的几个成员，其他成员都可以设置为 0。成员cb指定本结构的大小，单位为字节；成员dwFlags指定本结构体的哪些成员在进程创建窗口时是可用的，这里将该参数设置为 STARTF_USESTDHANDLES，表示标准输入、标准输出及标准错误句柄这 3 个成员可用；成员 hStdInput、hStdOutput、hStdError 分别代表新的进程标准输出、标准输入及标准错误句柄。

- 第十个参数 lpProcessInformation 是返回值，这是一个指向 PROCESS_INFORMATION 结构体的指针，返回新进程的标识信息。结构体 PROCESS_INFORMATION 的定义如下：

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION,
*LPPROCESS_INFORMATION;
```

成员hProcess和hThread分别代表新建进程的句柄及其主线程的句柄，通过这两个句柄可以分别对进程对象和主线程对象进行操作。后两个成员dwProcessId和dwThreadId分别是全局进程标识符和全局线程标识符。这两个标识符只在进程的生命期内有效，当进程终止时，可能会被系统重用而指派给其他进程和线程使用。

16.3.3　GetStdHandle()函数

该函数用于获取进程的标准输出、标准输入或标准错误的句柄。函数原型如下：

```
HANDLE GetStdHandle(
    DWORD nStdHandle
);
```

参数 nStdHandle 指定获取哪个标准设备的句柄，可以取 3 个不同的值，如表 16.4 所示。

表 16.4　　标准设备句柄

标 准 句 柄	说　明
STD_INPUT_HANDLE (DWORD)-10	标准输入设备，初始化为控制台的输入缓冲 CONIN$
STD_OUTPUT_HANDLE (DWORD)-11	标准输出设备，初始化为活动的控制台屏幕缓冲 CONOUT$

续表

标 准 句 柄	说　　明
STD_ERROR_HANDLE (DWORD)-12	标准错误设备，初始化为活动的控制台屏幕缓冲 CONERR$

16.3.4　ReadFile()和 WriteFile()函数

这两个函数的介绍参见 16.2 节邮槽的操作。前面说过管道也是一个驻留在内存中的伪文件，可以通过这两个函数来读/写。这里要说明的一点是当调用 ReadFile()函数时，如果管道内没有数据，该函数就会阻塞等待，直到有其他进程将数据写入管道才返回。另外，如果所有管道的写入句柄都已被关闭或读操作期间出现错误，ReadFile()函数也会返回。

16.3.5　一个利用匿名管道在父子进程间通信的示例

下面看一个基于匿名管道在父子进程之间互相传递数据的示例。父进程先创建一个匿名管道，获得管道的写入句柄和读出句柄；然后创建一个子进程，并将管道句柄继承给子进程，这样它们就可以利用管道交换数据。

1．服务器端实现

先看服务器端的实现，这是基于 MFC 对话框的程序，运行界面如图 16.6 所示。

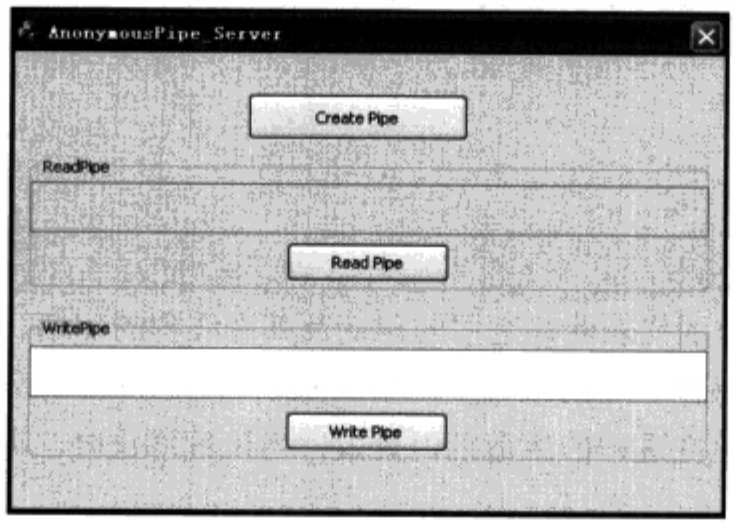


图 16.6　匿名管道服务器端的程序界面

最上面的 Create Pipe 按钮用于创建一个匿名管道，接着创建一个子进程，并将管道句柄继承给子进程；中间的 Read Pipe 按钮用于利用管道的读出句柄从管道内读出数据，并将读到的数据显示到中间的文本框内；下面的 Write Pipe 按钮用于将用户在下面文本框内输入的内容通过管道的写入句柄写入管道。

对话框类的主要成员如下：

```
protected:
    HANDLE hWrite;
    HANDLE hRead;
```

这里 hWrite 存放管道的写入句柄，用于向管道写入数据；而 hRead 存放管道的读出句柄，用于从管道内读出数据。在构造函数中将这两个成员都初始化为 NULL。代码如下：

```
CAnonymousPipe_ClientDlg::CAnonymousPipe_ClientDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CAnonymousPipe_ClientDlg::IDD, pParent)
{
```

```
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    hRead = NULL;
    hWrite = NULL;
}
```

在程序退出时需要将这两个句柄关闭，这在析构函数内进行。代码如下：

```
CAnonymousPipe_ClientDlg::~CAnonymousPipe_ClientDlg()
{
    if( hRead != NULL ){
        ::CloseHandle(hRead);
    }
    if( hWrite != NULL ){
        ::CloseHandle(hWrite);
    }
}
```

创建管道和子进程的“Create Pipe”按钮的单击事件处理函数如例 16.5 所示：

例 16.5

```
void CAnonymousPipe_ClientDlg::OnBnClickedButtonCreatepipe()
{
    // TODO: Add your control notification handler code here
    SECURITY_ATTRIBUTES sa;
    sa.bInheritHandle = TRUE;
    sa.nLength = sizeof(SECURITY_ATTRIBUTES);
    sa.lpSecurityDescriptor = NULL;
    //创建匿名管道，得到管道句柄
    if( ! ::CreatePipe( &hRead, &hWrite, &sa, 0 ) ){
        AfxMessageBox( _T("创建匿名管道失败") );
        return;
    }
    STARTUPINFO sui;
    ZeroMemory(&sui,sizeof(sui));
    sui.cb=sizeof(STARTUPINFO);
    sui.dwFlags=STARTF_USESTDHANDLES;
    sui.hStdInput=hRead;
    sui.hStdOutput=hWrite;
    sui.hStdError=GetStdHandle(STD_ERROR_HANDLE);
    PROCESS_INFORMATION pi;
    //创建子进程
    if( !CreateProcessW (_T("..\\..\\AnonymousPipeClient\\Debug\\AnonymousPipeClient.exe"),
        NULL,NULL,NULL,TRUE,0,NULL,NULL,&sui,&pi))
    {
        MessageBox(_T("create child process failed!"));
        CloseHandle(hRead);
        CloseHandle(hWrite);
        return;
    }
    //关闭进程句柄和主线程句柄
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

这里首先调用 CreatePipe()函数创建一个可继承的匿名管道，并将返回的管道的读出句柄和写入句柄分别存放到成员 hRead 和 hWrite 内，然后调用 CreateProcessW()函数创建子进程，这里将管道的读出句柄和写入句柄设置为子进程的标准输入和标准输出，这样就可以在子进程内获取这两个管道句柄。

下面介绍利用管道进行数据发送和接收操作，发送数据由 Write Pipe 按钮完成，该按钮的单击事件处理函数如例 16.6 所示。

例 16.6

```
void CAnonymousPipe_ClientDlg::OnBnClickedButtonWritepipe()
{
    // TODO: Add your control notification handler code here
    //获取用户输入
    CStringW strWriteMsg;
    this->GetDlgItemTextW(IDC_EDIT_WRITEPIPE,strWriteMsg);
    BOOL fResult;
    DWORD cbWritten;
    //将数据写入管道
    fResult = ::WriteFile(hWrite,
          (LPCTSTR)strWriteMsg ,
          (DWORD) (strWriteMsg.GetLength()+1)*sizeof(TCHAR),
          &cbWritten,
          (LPOVERLAPPED) NULL);
    if (!fResult)
    {
        CStringW errMsg;
        errMsg.Format(_T("WriteFile failed with %d"),::GetLastError());
        AfxMessageBox(errMsg);
        //CloseHandle(hFile);
        return;
    }
    this->SetDlgItemTextW(IDC_EDIT_WRITEPIPE,_T(""));
}
```

这里的操作相当简单，首先获取用户的输入，然后调用 WriteFile()函数通过管道的写入句柄将数据写入管道。如果写入出错，弹出一个消息框提醒出错原因。

负责读出数据的 Read Pipe 按钮的单击事件处理函数如例 16.7 所示。

例 16.7

```
void CAnonymousPipe_ClientDlg::OnBnClickedButtonReadpipe()
{
    // TODO: Add your control notification handler code here
    TCHAR buff[4096];
    DWORD nRead;
    if( !::ReadFile(hRead,buff,4096,&nRead,NULL) ){
        CStringW errMsg;
        errMsg.Format(_T("ReadFile failed with %d"),::GetLastError());
        AfxMessageBox(errMsg);
        return;
    }
    buff[nRead] = _T('\0');
```

```
    CStringW strTemp(buff);
    this->SetDlgItemTextW(IDC_EDIT_READPIPE,strTemp);
}
```

这里就是调用 ReadFile()函数通过管道的读出句柄将管道内数据读出，并显示到控件。注意如果管道内没有数据，那么 ReadFile()函数将被一直阻塞，直到有数据被写入管道后才会返回。

2．客户端实现

客户端也是一个 MFC 对话框程序，运行界面如图 16.7 所示。

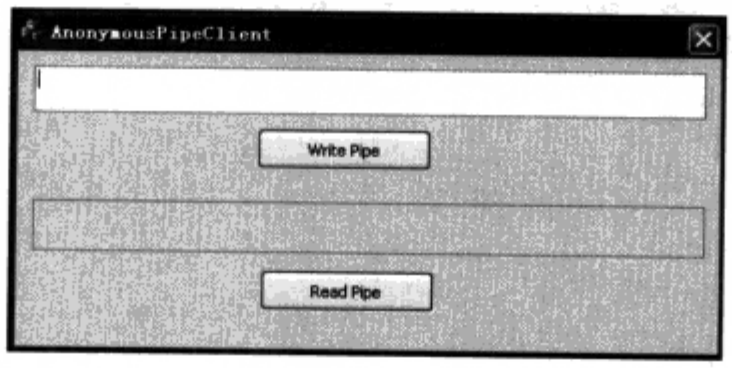


图 16.7　匿名管道客户端的程序界面

上面的 Write Pipe 按钮用于将用户输入的信息写入管道，下面的 Read Pipe 按钮用于从管道内读出数据并显示到下面的文本框。该对话框类的主要成员如下：

```
protected:
    HANDLE hWrite;
    HANDLE hRead;
```

同样，成员 hWrite 存放管道的写入句柄，成员 hRead 存放管道的读出句柄。

与服务器端相似，在构造函数内将句柄初始化为 NULL，在析构函数内将句柄关闭。该对话框类的构造函数和析构函数如下：

```
CAnonymousPipeClientDlg::CAnonymousPipeClientDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CAnonymousPipeClientDlg::IDD, pParent)
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    hRead = NULL;
    hWrite = NULL;
}
CAnonymousPipeClientDlg::~CAnonymousPipeClientDlg()
{
    if( hRead != NULL ){
        ::CloseHandle(hRead);
    }
    if( hWrite != NULL ){
        ::CloseHandle(hWrite);
    }
}
```

这两个句柄的真正初始化在 OnInitDialog 函数内进行：

```
hRead=GetStdHandle(STD_INPUT_HANDLE);
hWrite=GetStdHandle(STD_OUTPUT_HANDLE);
```

[说明]

在父进程（服务器端）创建子进程时，将标准输入和标准输出设置为管道的读出句柄和写入句柄，

那么在这里调用 GetStdHandle()函数获取的标准输入及标准输出句柄实际上就是管道的读出和写入句柄。下面就通过这两个句柄向父进程发送数据或从父进程接收数据。

客户端从管道读出数据和写入数据的方式与服务器端相似，写入数据的 Write Pipe 按钮的单击事件处理函数如例 16.8 所示。

例 16.8

```
void CAnonymousPipeClientDlg::OnBnClickedButtonWritepipe()
{
    // TODO: Add your control notification handler code here
    CStringW strWriteMsg;
    this->GetDlgItemTextW(IDC_EDIT_WRITEPIPE,strWriteMsg);
    BOOL fResult;
    DWORD cbWritten;
    //将数据写入管道
    fResult = ::WriteFile(hWrite,
        (LPCTSTR)strWriteMsg,
        (DWORD) (strWriteMsg.GetLength()+1)*sizeof(TCHAR),
        &cbWritten,
        (LPOVERLAPPED) NULL);
    if (!fResult)
    {
        CStringW errMsg;
        errMsg.Format(_T("WriteFile failed with %d"),::GetLastError());
        AfxMessageBox(errMsg);
        //CloseHandle(hFile);
        return;
    }
    this->SetDlgItemTextW(IDC_EDIT_WRITEPIPE,_T(""));
}
```

读出数据的 Read Pipe 按钮的单击事件处理函数如例 16.9 所示。

例 16.9

```
void CAnonymousPipeClientDlg::OnBnClickedButtonReadpipe()
{
    // TODO: Add your control notification handler code here
    TCHAR buff[4096];
    DWORD nRead;
    if( !::ReadFile(hRead,buff,4096,&nRead,NULL) ){
        CStringW errMsg;
        errMsg.Format(_T("ReadFile failed with %d"),::GetLastError());
        AfxMessageBox(errMsg);
        return;
    }
    buff[nRead] = _T('\0');
    CStringW strTemp(buff);
    this->SetDlgItemTextW(IDC_EDIT_READPIPE,strTemp);
}
```

整个程序介绍完毕，运行时首先运行服务器端，然后单击 Create Pipe 按钮创建匿名管道并启动客户端程序，接着就可以通过管道在父子进程之间交换数据，运行界面如图 16.8 所示。当然，父进程也

可以自己向管道写入数据，再自己从管道内读出数据，就是自己跟自己通信，如图 16.9 所示。

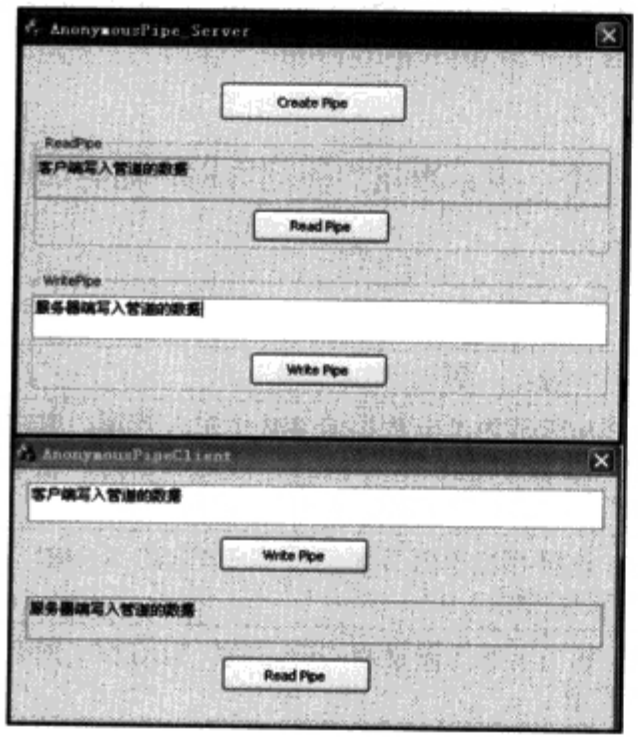


图 16.8　使用匿名管道在服务器端和客户端通信的运行结果

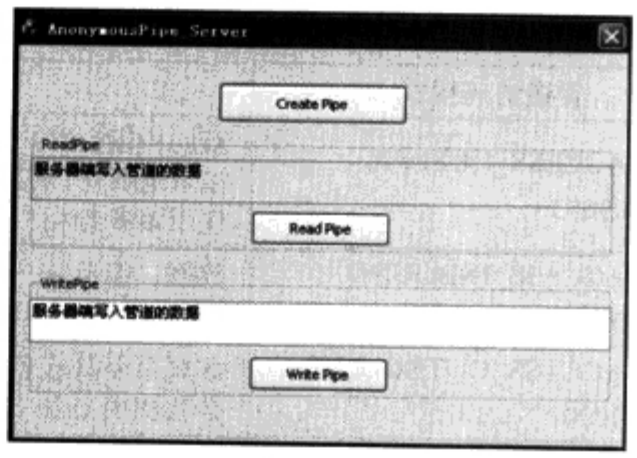


图 16.9　服务器端自己读写匿名管道的运行结果

16.4　命名管道通信

命名管道可以实现跨网络的进程间通信，它屏蔽了底层的网络协议细节。同样，命名管道也建立了一个客户端/服务器模型，并在其中可靠地传输数据。创建管道的进程称为服务器，而连接管道的进程称为客户端，任何进程都可以通过管道的名称连接管道，而不必局限于父子进程之间。每一个命名管道的实例都共享同一个管道的名称，但是每个实例都有自己的缓冲和句柄，这样在服务器和每个客户端之间都提供了一个独立的通道，互不干扰，也就允许多个管道客户端同时使用相同名称的管道与服务器通信。下面介绍关于命名管道操作的函数。

16.4.1　CreateNamedPipe()函数

该函数用于创建一个命名管道的实例并返回一个用于后面管道操作的句柄。一个命名管道的服务器进程调用这个函数或者创建具有特定名称的管道的第一个实例，并设置该命名管道的基本属性或者创建一个已经存在的命名管道的新的实例。函数原型如下：

```
HANDLE CreateNamedPipe(
    LPCTSTR lpName,
    DWORD dwOpenMode,
    DWORD dwPipeMode,
    DWORD nMaxInstances,
    DWORD nOutBufferSize,
    DWORD nInBufferSize,
    DWORD nDefaultTimeOut,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

如果函数成功执行，将返回一个指向服务器端命名管道实例的句柄；如果创建失败，则返回 INVALID_HANDLE_VALUE。

- 第一个参数 lpName 指定管道的名称，必须使用下面格式的名称：\\.\pipe\pipename。客户端如果要连接远程机器上的管道服务器，则相应指定服务器的名称：\\ServerName\pipe\PipeName。注意管道名称不区分大小写。
- 第二个参数 dwOpenMode 指定管道的打开模式。

该参数必须使用表 16.5 列出的 3 个管道访问模式中的一个，表示数据的流动方向，而且所有的管道实例都必须使用相同的模式。

表 16.5　　管道访问模式

管道访问模式	说　明
PIPE_ACCESS_DUPLEX 0x00000003	双向传输；客户端和服务器都可以从管道内读出数据或将数据写入管道。客户端连接管道时可以使用 GENERIC_READ 或 GENERIC_WRITE 或两者组合，服务器端使用该参数相当于使用 GENERIC_READ|GENERIC_WRITE 访问管道
PIPE_ACCESS_INBOUND 0x00000001	数据只能从客户端向服务器传输，这样相当于服务器使用 GENERIC_READ，而客户端必须使用 GENERIC_WRITE 连接管道
PIPE_ACCESS_OUTBOUND 0x00000002	数据只能从服务器向客户端传输，相当于服务器使用 GENERIC_WRITE，而客户端必须使用 GENERIC_READ 连接管道

参数还可以使用表 16.6 中列出的一个或多个（用“|”连接），用来指定使能写直通和重叠模式，而且允许不同的管道实例使用不同的参数。

表 16.6　　管道写通和重叠模式

访 问 模 式	说　明
FILE_FLAG_WRITE_THROUGH 0x80000000	使能写直通模式。该模式只对字节模式管道的写操作有影响，并且只在客户端和服务器端位于不同的机器时才有效。如果该模式有效，那么向一个管道写入数据的操作在数据通过网络传输到远程计算机的管道缓冲区之前是不会返回的。如果该模式关闭，那么将由系统通过缓冲数据来提高网络的效率，直到累积的字节数达到最小值或超过了最大时间
FILE_FLAG_OVERLAPPED 0x40000000	使能重叠模式。如果采用该模式，那么读、写和连接可能需要花费一定时间进行操作并立即返回。该模式允许线程启动这些操作，然后将这次操作放到后台执行，从而可以执行其他操作。如果采用重叠模式，那么读、写和连接操作在执行完成之前是不会返回的

表 16.7 列出的 3 个参数用于指定管道的安全访问模式，可以选择一个或多个任意组合，不同的管道实例可以使用不同的模式。

表 16.7　　管道的安全访问模式

安全访问模式	说　明
WRITE_DAC 0x00040000L	应用程序可对命名管道的授权访问控制列表（DACL）进行写操作
WRITE_OWNER 0x00080000L	用程序可对命名管道的所有者进行写操作
ACCESS_SYSTEM_SECURITY 0x01000000L	应用程序可对命名管道的系统访问控制列表（SACL）进行写操作

- 第三个参数 dwPipeMode 指定命名管道的类型。

该参数只能取表 16.8 中列出的值之一，指定管道的类型，每一个管道的实例都必须使用相同的参数。

表 16.8　　命名管道的类型

管 道 类 型	说　明
PIPE_TYPE_BYTE 0x00000000	数据将以字节流的方式写入管道，数据之间没有界定符。该模式不能在 PIPE_READMODE_MESSAGE 下使用

续表

管 道 类 型	说　明
PIPE_TYPE_MESSAGE 0x00000004	数据以消息流的方式写入管道，各个消息之间含有界定符，必须以消息为单位对数据进行读写。该模式在 PIPE_READMODE_MESSAGE 和 PIPE_READMODE_BYTE 下都可用

表 16.9 列出了管道的读模式，取表中的值之一，不同的管道实例可以使用不同的参数。

表 16.9　　管道的读模式

读　模　式	说　明
PIPE_READMODE_BYTE 0x00000000	数据以字节流的方式从管道中读出。该模式在 PIPE_READMODE_MESSAGE 和 PIPE_READMODE_BYTE 下都可用
PIPE_READMODE_MESSAGE 0x00000002	数据以消息流的方式从管道中读出。该模式只在 PIPE_READMODE_ MESSAGE 下可用

表 16.10 列出了管道的等待模式，取表中的值之一，不同的管道实例可以使用不同的等待模式。

表 16.10　　管道的等待模式

等 待 模 式	说　明
PIPE_WAIT 0x00000000	使能阻塞模式，当管道调用特定的 ReadFile()、WriteFile()或 ConnectNamePipe()函数时，直到有数据被读取，所有数据都写入或一个客户段连接完成，才能返回
PIPE_NOWAIT 0x00000001	使能非阻塞模式，上面的操作将立即返回

- 第四个参数 nMaxInstances 指定管道所能允许的实例的最大数量。只有管道的第一个实例可以指定这个值，后面创建的实例必须指定相同的值，可用的值的范围是 1 ~ PIPE_UNLIMITED_INSTANCES（255）。如果该值超过 PIPE_UNLIMITED_INSTANCES，该函数将出错，返回 ERROR_INVALID_PARAMETER。
- 第五个参数 nOutBufferSize 指定为管道的输出缓冲保留的空间大小，单位为字节。
- 第六个参数 nInBufferSize 指定为管道的输入缓冲保留的空间大小，单位为字节。
- 第七个参数 nDefaultTimeOut 指定管道的默认超时时间，每个管道的实例都必须使用相同的值。该值在 WaitNamePipe()函数指定 NMPWAIT_USE_DEFAULT_WAIT 标志时有效。
- 第八个参数 lpSecurityAttributes 为一个指向 SECURITY_ATTRIBUTES 结构体的指针，指定返回的句柄能否被子进程继承。如果值为 NULL，则表示该句柄不能被子进程继承。

16.4.2　ConnectNamedPipe()函数

管道服务器进程调用该函数等待来自客户端进程对命名管道的一个实例的连接，客户端可以通过 CreateFile()或 CallNamePipe()函数来连接。函数原型如下。

```
BOOL ConnectNamedPipe(
    HANDLE hNamedPipe,
    LPOVERLAPPED lpOverlapped
);
```

如果函数成功执行，则返回非 0（True）；失败则返回 0（False）。

- 第一个参数 hNamedPipe 为一个指向服务器端命名管道实例的句柄，该句柄必须是CreateNamedPipe的合法返回值。
- 第二个参数 lpOverlapped 为一个指向 OVERLAPPED 结构的指针，指定当调用 CreateNamedPipe()

函数创建管道使用 FILE_FLAG_OVERLAPPED（重叠式 I/O）标记时，该函数将以异步方式工作。若将该参数设为 NULL，该函数将阻塞直到客户机成功建立同服务器的连接为止。

16.4.3 DisconnectNamedPipe()函数

管道服务器进程调用该函数来断开同客户端的连接。函数原型如下。

```
BOOL DisconnectNamedPipe(
    HANDLE hNamedPipe
);
```

如果函数成功执行，则返回 True；失败则返回 False。

参数 hNamedPipe 为一个指向服务器端命名管道实例的句柄，该句柄必须是CreateNamedPipe()函数的合法返回值。

16.4.4 WaitNamedPipe()函数

客户端调用该函数等待服务器端的命名管道可用于连接或者等待时间超过设置的超时时间。函数原型如下。

```
BOOL WaitNamedPipe(
    LPCTSTR lpNamedPipeName,
    DWORD nTimeOut
);
```

如果函数成功执行，则返回 True；失败则返回 False。

- 参数 lpNamedPipeName 指定要等待的管道的名称，如果是远程机器上的管道，需要指定服务器名。
- 参数 nTimeOut 指定等待一个命名管道实例可用的时间，单位为毫秒（ms）。可以使用表 16.11 列出的两个特殊值。

表 16.11　　管道的超时时间

超 时 时 间	说　明
NMPWAIT_USE_DEFAULT_WAIT 0x00000000	超时时间将被指定为服务器进程在调用 GreateNamePipe()函数（对应 nDefaultTimeOut 参数）创建管道时设置的默认值
NMPWAIT_WAIT_FOREVER 0xffffffff	该函数将一直阻塞等待直到一个命名管道的实例可用，才返回。

如果根本就没有该命名管道的任何实例，该函数将马上返回，而忽略超时时间的值。这里的等待时间指的是等待一个管道实例可用的时间，前提是该实例已经创建，但是正被其他连接占用。

如果函数成功执行，进程需要使用 CreateFile()函数来打开对应命名管道的句柄。即使该函数成功执行，表示当前至少有一个管道实例可用，但是后续的调用 CreateFile()函数也可能失败，因为该管道实例可能会被服务器关闭或者被其他客户端先打开。注意一个管道实例一次只能处理一个客户端的连接，如果希望同时与多个客户端连接，服务器就需要创建多个管道实例。这样的模式与前面介绍的流式 Socket 的通信方式十分相似，也可以认为命名管道是面向连接的服务，在通信之前需要建立连接。

16.4.5 利用命名管道通信的基本流程

下面简单介绍不同进程利用命名管道进行通信的基本流程，这分为服务器端和客户端。

服务器端的主要工作：

（1）调用 CreateNamePipe()函数创建一个特定名字的管道的一个或多个实例。

（2）调用 ConnectNamedpipe()函数监听来自客户端对该管道实例的连接请求。

（3）根据管道的类型，调用 ReadFile()函数从管道内读取数据；调用 WriteFile()函数向管道写入数据。

（4）调用 DisconnectNamedPipe()函数关闭与一个客户端的连接。

（5）调用 CloseHandle()函数关闭服务器端的管道实例，释放相应的资源。

客户端的主要工作：

（1）调用 WaitNamePipe()函数查询是否有可用的服务器端管道实例可供客户端连接。

（2）调用 CreateFile()函数建立一个与服务器端管道实例的连接，并获取用于通信的管道句柄。

（3）调用 ReadFile()函数从管道读取数据；调用 WriteFile()函数向管道内写入数据。

（4）调用 CloseHandle()函数关闭连接，关闭管道句柄并释放资源。

下面通过一个实例来讲解管道通信的大概过程。

16.4.6　一个利用命名管道在不同进程间通信的示例

1．服务器端的实现

先介绍服务器端的实现。这是一个基于 MFC 对话框的程序，服务器端的运行界面如图 16.10 所示。

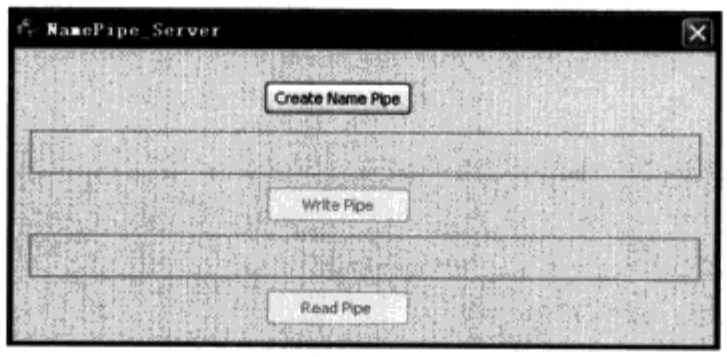


图 16.10　命名管道服务器端的程序界面

上面的 Create Name Pipe 按钮用于创建一个命名管道的实例，然后调用 ConnectNamedPipe()函数等待客户端的连接，这时整个窗口将阻塞，直到有客户端建立连接成功。中间的文本框在客户端连接成功后将变为可写，接收用户输入的数据，然后单击下面的 Write Pipe 按钮将这些数据写入管道。单击最下面的 Read Pipe 按钮可从管道读出数据，并显示到按钮上方的文本框中。

该对话框类的主要成员如下：

```
protected:
    HANDLE m_hPipe;
    CStringW m_strPipeName;
```

成员 m_hPipe 存放一个指向命名管道实例的句柄，通过该句柄对管道进行读/写操作。成员 m_strPipeName 存放管道的名称，注意名称必须符合前面介绍的格式，而且不区分大小写。它们的初始化代码如下：

```
CNamePipe_ServerDlg::CNamePipe_ServerDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CNamePipe_ServerDlg::IDD, pParent)
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    m_strPipeName = _T("\\\\.\\Pipe\\MyNamePipe");
```

```
        m_hPipe = NULL;
    }
```

在服务器端程序退出时，需要先断开连接并关闭管道句柄释放资源，这在析构函数内进行（需要用户自己添加析构函数），代码如下：

```
CNamePipe_ServerDlg::~CNamePipe_ServerDlg()
{
    if(m_hPipe != NULL)
    {
        ::DisconnectNamedPipe(m_hPipe);
        ::CloseHandle(m_hPipe);
        m_hPipe = NULL;
    }
}
```

这里首先调用 DisconnectNamedPipe()函数断开连接，然后调用 CloseHandle()函数关闭句柄并释放资源。

在服务器端创建命名管道实例并取得一个客户端连接之前，上面的发送数据、接收数据的按钮都应该是不可用的，这在 OnInitDialog()函数内设置，代码如下。

```
this->GetDlgItem(IDC_BUTTON_WRITEPIPE)->EnableWindow(FALSE);
this->GetDlgItem(IDC_BUTTON_READPIPE)->EnableWindow(FALSE);
((CEdit*)this->GetDlgItem(IDC_EDIT_WRITEPIPE))->SetReadOnly(TRUE);
```

创建命名管道及在服务器端监听连接的操作在 Create Name Pipe 按钮的单击事件处理函数内进行，如例 16.10 所示。

例 16.10

```
void CNamePipe_ServerDlg::OnBnClickedButtonCreatepipe()
{
    // TODO: Add your control notification handler code here
    //创建一个命名管道的实例，返回管道句柄
    m_hPipe = CreateNamedPipeW(
        m_strPipeName,                  // 管道名称
        PIPE_ACCESS_DUPLEX,             // 管道的读/写模式
        PIPE_TYPE_MESSAGE |             // 使用消息流类型的管道
        PIPE_READMODE_MESSAGE |         // 消息流读模式
        PIPE_WAIT,                      // 阻塞模式
        PIPE_UNLIMITED_INSTANCES,       // 允许最大管道实例个数
        BUFSIZE,                        // 输出缓冲大小
        BUFSIZE,                        // 输入缓冲大小
        0,                              // 客户端超时时间
        NULL);                          // 使用默认安全属性
    if (m_hPipe == INVALID_HANDLE_VALUE){
        //创建管道失败
        AfxMessageBox(_T("CreateNamePipe failed"));
        ::PostQuitMessage(-8);
        return;
    }
    this->SetDlgItemTextW(IDC_BUTTON_CREATEPIPE,_T("Waiting Connect..."));
    //阻塞等待客户端连接
    BOOL fConnected = ConnectNamedPipe(m_hPipe, NULL) ?
```

```
            TRUE : (GetLastError() == ERROR_PIPE_CONNECTED);
        this->SetDlgItemTextW(IDC_BUTTON_CREATEPIPE,_T("Create Name Pipe"));
        this->GetDlgItem(IDC_BUTTON_CREATEPIPE)->EnableWindow(FALSE);
        this->GetDlgItem(IDC_BUTTON_WRITEPIPE)->EnableWindow(TRUE);
        this->GetDlgItem(IDC_BUTTON_READPIPE)->EnableWindow(TRUE);
        ((CEdit*)this->GetDlgItem(IDC_EDIT_WRITEPIPE))->SetReadOnly(FALSE);
        if(fConnected)
        {
            AfxMessageBox(_T("ConnectNamePipe Successfully"));
        }
        else{
            ::CloseHandle(m_hPipe);
            m_hPipe = NULL;
            AfxMessageBox(_T("ConnectNamePipe Failed"));
            ::PostQuitMessage(-8);
            return;
        }
    }
```

这里首先调用 CreateNamePipe()函数创建一个指定名称的管道实例，如果创建失败就退出程序，否则调用 ConnectNamePipe()函数监听客户端的连接，该函数将一直阻塞直到客户端的连接建立成功为止。一旦连接成功，就允许用户对管道进行写入和读出操作。这里指定缓冲区大小为 BUFSIZE，这在对话框头文件内定义。

```
#define BUFSIZE 4096
```

向管道写入数据的操作通过 WriteFile()函数实现，这在 Write File 按钮的单击事件处理函数内进行，如例 16.11 所示。

例 16.11

```
void CNamePipe_ServerDlg::OnBnClickedButtonWritepipe()
{
    // TODO: Add your control notification handler code here
    CStringW strWriteMsg;
    this->GetDlgItemTextW(IDC_EDIT_WRITEPIPE,strWriteMsg);
    BOOL fResult;
    DWORD cbWritten;
    //将数据写入管道
    fResult = ::WriteFile(m_hPipe,
        (LPCTSTR)strWriteMsg ,
        (DWORD) (strWriteMsg.GetLength()+1)*sizeof(TCHAR),
        &cbWritten,
        (LPOVERLAPPED) NULL);
   if (!fResult)
   {
        CStringW errMsg;
        errMsg.Format(_T("WriteFile failed with %d"),::GetLastError());
        AfxMessageBox(errMsg);
        return ;
   }
    this->SetDlgItemTextW(IDC_EDIT_WRITEPIPE,_T(""));
}
```

从管道内读出数据通过调用 ReadFile()函数完成，这在 Read File 按钮的单击事件处理函数内进行，如例 16.12 所示。

例 16.12

```
void CNamePipe_ServerDlg::OnBnClickedButtonReadpipe()
{
    // TODO: Add your control notification handler code here
    TCHAR buff[BUFSIZE];
    DWORD nRead;
    if( !::ReadFile(m_hPipe,buff,BUFSIZE,&nRead,NULL) ){
        CStringW errMsg;
        errMsg.Format(_T("ReadFile failed with %d"),::GetLastError());
        AfxMessageBox(errMsg);
        return ;
    }
    buff[nRead] = _T('\0');
    CStringW strTemp(buff);
    this->SetDlgItemTextW(IDC_EDIT_READPIPE,strTemp);
}
```

2. 客户端的实现

下面介绍客户端的实现，客户端的运行界面如图 16.11 所示。

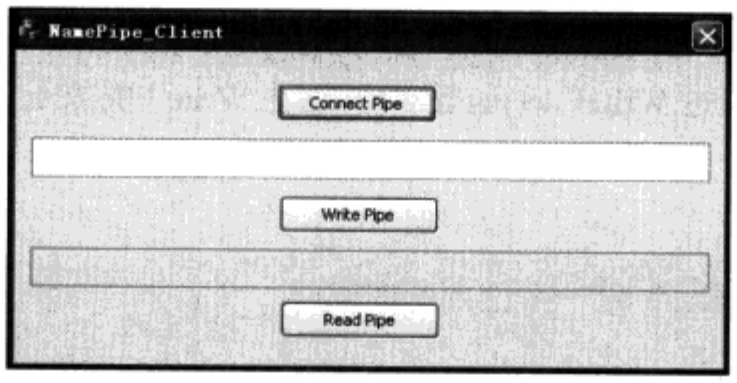


图 16.11　命名管道客户端的程序界面

最上面的 Connect Pipe 按钮用于打开之前服务器端创建的命名管道，并建立与服务器的连接，下面的两个按钮分别用于从管道内读出和写入数据。

对话框类的主要成员如下：

```
protected:
    HANDLE m_hPipe;
    CStringW m_strPipeName;
```

成员 m_hPipe 存放管道句柄，成员 m_strPipeName 存放管道名称，该管道名称必须与服务器创建的管道名称一致。初始化代码如下：

```
CNamePipe_ClientDlg::CNamePipe_ClientDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CNamePipe_ClientDlg::IDD, pParent)
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    m_strPipeName = _T("\\\\.\\Pipe\\MyNamePipe");
    m_hPipe = NULL;
}
```

在退出程序之前需要关闭句柄并释放资源，这在析构函数内进行，代码如下：

```
CNamePipe_ClientDlg::~CNamePipe_ClientDlg()
{
    if(m_hPipe != NULL){
        ::CloseHandle(m_hPipe);
        m_hPipe = NULL;
    }
}
```

Connect Pipe 按钮主要用于打开管道并建立一个到服务器的连接，该按钮的单击事件处理函数如例 16.13 所示。

例 16.13

```
void CNamePipe_ClientDlg::OnBnClickedButtonConnectpipe()
{
    // TODO: Add your control notification handler code here
    //查询是否有管道实例可用
    if (!WaitNamedPipeW(m_strPipeName, NMPWAIT_USE_DEFAULT_WAIT))
    {
        AfxMessageBox(_T("Could not open pipe"));
        return;
    }
    m_hPipe = CreateFileW(
        m_strPipeName,          // 管道名称
        GENERIC_READ |          // 以可读写模式打开管道
        GENERIC_WRITE,
        0,                      // 不支持共享
        NULL,                   // 默认安全属性
        OPEN_EXISTING,          // 只打开已存在的管道
        0,                      // 默认属性
        NULL);                  // 不需要默认文件
    if (m_hPipe != INVALID_HANDLE_VALUE){
        this->GetDlgItem(IDC_BUTTON_CONNECTPIPE)->EnableWindow(FALSE);
        this->GetDlgItem(IDC_BUTTON_WRITEPIPE)->EnableWindow(TRUE);
        this->GetDlgItem(IDC_BUTTON_READPIPE)->EnableWindow(TRUE);
        ((CEdit*)this->GetDlgItem(IDC_EDIT_WRITEPIPE))->SetReadOnly(FALSE);
        return;
    }
    // Exit if an error other than ERROR_PIPE_BUSY occurs
    if (GetLastError() != ERROR_PIPE_BUSY)
    {
        AfxMessageBox(_T("Could not open pipe"));
        return;
    }
}
```

这里先调用 WaitNamePipeW()函数查询是否存在可用的管道实例，如果存在就调用 CreateFileW()函数打开管道并建立与服务器的连接，该函数将一直阻塞直到一个连接建立完成。

将数据写入管道的 Write Pipe 按钮的单击事件处理函数如例 16.14 所示。

例 16.14

```
void CNamePipe_ClientDlg::OnBnClickedButtonWritepipe()
```

```
{
    // TODO: Add your control notification handler code here
    CStringW strWriteMsg;
    this->GetDlgItemTextW(IDC_EDIT_WRITEPIPE,strWriteMsg);
    BOOL fResult;
    DWORD cbWritten;
    //将数据写入管道
    fResult = ::WriteFile(m_hPipe,
    (LPCTSTR)strWriteMsg ,
    (DWORD) (strWriteMsg.GetLength()+1)*sizeof(TCHAR),
     &cbWritten, (LPOVERLAPPED) NULL);
    if (!fResult)
    {
        CStringW errMsg;
        errMsg.Format(_T("WriteFile failed with %d"),::GetLastError());
        AfxMessageBox(errMsg);
        return ;
    }
    this->SetDlgItemTextW(IDC_EDIT_WRITEPIPE,_T(""));
}
```

而从管道读取数据的 Read File 按钮的单击事件处理函数如例 16.15 所示。

例 16.15

```
void CNamePipe_ClientDlg::OnBnClickedButtonReadpipe()
{
    // TODO: Add your control notification handler code here
    TCHAR buff[BUFSIZE];
    DWORD nRead;
    if( !::ReadFile(m_hPipe,buff,BUFSIZE,&nRead,NULL) ){
        CStringW errMsg;
        errMsg.Format(_T("ReadFile failed with %d"),::GetLastError());
        AfxMessageBox(errMsg);
        return ;
    }
    buff[nRead] = _T('\0');
    CStringW strTemp(buff);
    this->SetDlgItemTextW(IDC_EDIT_READPIPE,strTemp);
}
```

关于该程序的设计就介绍到此，下面就可以编译并运行该应用程序。首先运行服务器端，创建管道并等待客户端连接；然后运行客户端，打开管道并与服务器连接。该程序总的运行结果如图 16.12 所示。在客户端与服务器端的管道实例建立连接之后，服务器端写入管道的数据就可以被客户端从管道读出，而客户端写入管道的数据也能被服务器端接收。当管道内没有数据时，读取管道将导致进程阻塞直到其他进程将数据写入管道为止。

图 16.12　利用命名管道在服务器端和客户端之间进行通信的运行结果

16.5　共享内存通信

实际上前面介绍的剪贴板、邮槽、命名管道都是利用共享内存在进程间交换数据的。除了利用这些经过封装的方式，还可以直接对共享内存进行操作。下面介绍共享内存的相关概念。

创建一个共享内存对象的进程称为服务器，而访问该共享内存的进程称为客户端。这样通过共享内存进行通信就非常简单明了：某个进程将数据写入特定的共享内存，而另一个进程就从该共享内存中读出数据。下面介绍与共享内存操作有关的函数。

16.5.1　CreateFileMapping()函数

该函数用于对特定的文件创建或打开一个命名或匿名的文件映射对象，如果不指定文件句柄将创建共享内存。函数原型如下：

```
HANDLE CreateFileMapping(
    HANDLE hFile,
    LPSECURITY_ATTRIBUTES lpAttributes,
    DWORD flProtect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
     LPCTSTR lpName
);
```

如果函数成功执行，则返回指向文件映射对象的句柄；如果该对象已经存在，则返回指向该对象的句柄，同时 GetLastError()函数返回 ERROR_ALREADY_EXISTS；如果失败，则返回 NULL。

- 第一个参数 hFile 为一个指向将创建文件映射对象的文件句柄，如果值设置为 INVALID_HANDLE_VALUE，那么将创建一个共享内存。
- 第二个参数lpAttributes为一个指向 SECURITY_ATTRIBUTES 结构体的指针，指定句柄的安全属性，主要是设置该返回句柄是否可以被子进程继承。如果值设置为 NULL，表示该句柄不能被继承。
- 第三个参数flProtect指定文件被映射的对应文件视图的保护属性，即这块映射内存区域的访问属性。该参数可以取如表 16.12 所示的值之一。

表 16.12　　文件视图的保护属性

保 护 属 性	说　　明
PAGE_READONLY	设置只读访问。任意尝试对该文件视图的写操作都将导致失败。对应的文件句柄 hFile 在创建时必须使用 GENERIC_READ 访问属性
PAGE_READWRITE	设置读写访问，对应的文件句柄 hFile 在创建时必须使用 GENERIC_READ 和 GENERIC_WRITE 属性
PAGE_WRITECOPY	设置写复制访问。对应的文件句柄 hFile 在创建时必须使用 GENERIC_READ 访问属性

该参数还可以使用如表 16.13 所示的值中的一个或多个组合（用“|”组合），用于指定页面的分配属性。

表 16.13　　页面的分配属性

分 配 属 性	说　　明
SEC_COMMIT	为所有的页面分配物理存储或页文件，这是默认设置

续表

分 配 属 性	说　　明
SEC_IMAGE	表示该文件被指定为可执行文件
SEC_NOCACHE	所有页面都不能放到 Cache。该标志不常用，除非设备这样要求
SEC_RESERVE	所有的页面都保留，不为其分配物理存储

- 第四个参数 dwMaximumSizeHigh 指定文件映射对象的最大空间大小的高端 DWORD 部分。
- 第五个参数 dwMaximumSizeLow 指定文件映射对象的最大空间大小的低端 DWORD 部分。如果这两个参数都指定为 0，那么文件映射对象的最大空间大小将为文件句柄 hFile 指向的文件大小。
- 第六个参数lpName指定文件映射对象的名称，如果该参数为 NULL，则表示创建一个匿名的对象。通过该名称，其他进程就可以访问该共享内存。

16.5.2 MapViewOfFile()函数

该函数用于将一个文件映射对象映射一个视图到调用进程的地址空间内，之后调用进程就可以访问该文件映射对象，在这里就是共享内存。函数原型如下：

```
LPVOID MapViewOfFile(
    HANDLE hFileMappingObject,
    DWORD dwDesiredAccess,
    DWORD dwFileOffsetHigh,
    DWORD dwFileOffsetLow,
    SIZE_T dwNumberOfBytesToMap
);
```

如果函数成功执行，将返回一个指向映射视图的起始地址的指针；如果失败，则返回 NULL。

- 第一个参数 hFileMappingObject 指定一个文件映射对象的句柄，该句柄必须由 CreateFilMapping() 或 OpenFileMapping()函数返回。
- 第二个参数dwDesiredAccess指定对该文件映射对象的访问类型，必须与该对象创建时的页面保护属性对应。该参数可以取如表 16.14 所示的值之一。

表 16.14　　文件映射对象的访问类型

访 问 类 型	说　　明
FILE_MAP_ALL_ACCESS	相当于指定 FILE_MAP_WRITE|FILE_MAP_READ。该对象在创建时必须使用 PAGE_READWRITE 选项
FILE_MAP_COPY	映射文件的一个写复制视图，对象在创建时必须使用 PAGE_WRITECOPY 选项。这表示只有当真正要将数据写入页面时，系统才会为页面分配物理存储，但是对该页面的任何修改都不会被写回到原文件内
FILE_MAP_READ	映射文件的一个只读视图。该对象在创建时必须使用 PAGE_READWRITE 或 PAGE_READONLY 选项
FILE_MAP_WRITE	映射文件的一个读写视图。该对象在创建时必须使用 PAGE_READWRITE 选项

- 第三个参数 dwFileOffsetHigh 和第四个参数 dwFileOffsetLow 联合指定映射对应视图起始处的文件偏移量。参数 dwFileOffsetHigh 指定该偏移量的高端 DWORD 部分，而参数 dwFileOffsetLow 指定该偏移量的低端 DWORD 部分。
- 第五个参数 dwNumberOfBytesToMap 指定映射到视图内的文件大小，单位为字节。后面这 3 个参数表示可以指定映射从某个地方开始的一段指定大小的区域，而不用从头开始全部映射进来。

16.5.3　UnmapViewOfFile()函数

该函数用于关闭一个在进程地址空间内的映射视图。函数原型如下：

```
BOOL UnmapViewOfFile(
    LPCVOID lpBaseAddress
);
```

如果函数成功执行，则返回非 0（True），所有“脏”（被修改过的）页面都将采用“懒惰”写回磁盘；如果失败，则返回 0（False）。

参数 lpBaseAddress 为一个指向映射视图基地址的指针，该指针必须由 MapViewOfFile()或 MapViewOfFileEx()函数返回。

在对映射进来的视图操作完毕后，必须调用该函数来关闭视图；最后还要调用 CloseHandle()函数来关闭句柄，释放资源。

16.5.4　OpenFileMapping()函数

该函数用于打开一个命名文件映射对象，客户端对共享内存进行访问之前必须调用该函数。函数原型如下：

```
HANDLE OpenFileMapping(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    LPCTSTR lpName
);
```

如果函数成功执行，将返回一个指向打开的文件映射对象的句柄；如果失败则返回 NULL。

- 第一个参数 dwDesiredAccess 指定对文件映射对象的访问类型。
- 第二个参数 bInheritHandle 指定该句柄是否可以被子进程继承，如果值为 True 则表示可以被子进程继承。
- 第三个参数 lpName 指定打开的文件映射对象的名称。

16.5.5　利用共享内存通信的基本流程

1．服务器端操作流程

（1）调用 CreateFileMapping()函数创建一个指定名称的共享内存的文件映射对象。
（2）调用 MapViewOfFile()函数将文件映射对象映射到服务器进程的地址空间上的一个视图。
（3）调用 CopyMemory()函数从共享内存读取数据或将数据写入共享内存。
（4）调用 UnmapViewOfFile()函数关闭文件映射对象在服务器进程的地址空间上的视图。
（5）调用 CloseHandle()关闭指向文件映射对象的句柄并释放资源。

2．客户端操作流程

（1）调用 OpenFileMapping()函数打开一个已经创建的指定名称的共享内存的文件映射对象。
（2）调用 MapViewOfFile()函数将文件映射对象映射到服务器进程的地址空间上的一个视图。
（3）调用 CopyMemory()函数从共享内存读取数据或将数据写入共享内存。
（4）调用 UnmapViewOfFile()函数关闭文件映射对象在服务器进程的地址空间上的视图。

（5）调用 CloseHandle()关闭指向文件映射对象的句柄并释放资源。

16.5.6 一个利用共享内存在不同进程间通信的示例

这是一个基于 MFC 对话框的程序，服务器端主要就是创建一个共享内存，并允许多个客户端程序打开该共享内存，然后就可以对共享内存进行读/写，从而实现在服务器端进程与多个客户端进程之间交换数据。

1．服务器端的实现

首先介绍服务器端的实现，服务器端的运行界面如图 16.13 所示。

上面的 CreateFileMapping 按钮用于创建一个共享内存，并在本进程空间上映射一个视图。下面的文本框和按钮用于读/写共享内存。该对话框类的主要成员如下：

```
protected:
    HANDLE m_hMapFile;
    CStringW m_strMapFileName;
    LPCTSTR m_pBuf;
```

成员 m_hMapFile 存放指向共享内存的文件映射对象的句柄，成员 m_strMapFileName 存放共享内存的名称，成员 m_pBuf 为指向将文件映射对象映射到服务器进程地址空间的视图的起始地址的指针。

这几个成员的初始化代码如下：

```
CSharedMem_ServerDlg::CSharedMem_ServerDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CSharedMem_ServerDlg::IDD, pParent)
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    m_hMapFile = NULL;
    m_strMapFileName = _T("Global\\MyFileMappingObject");
    m_pBuf = NULL;
}
```

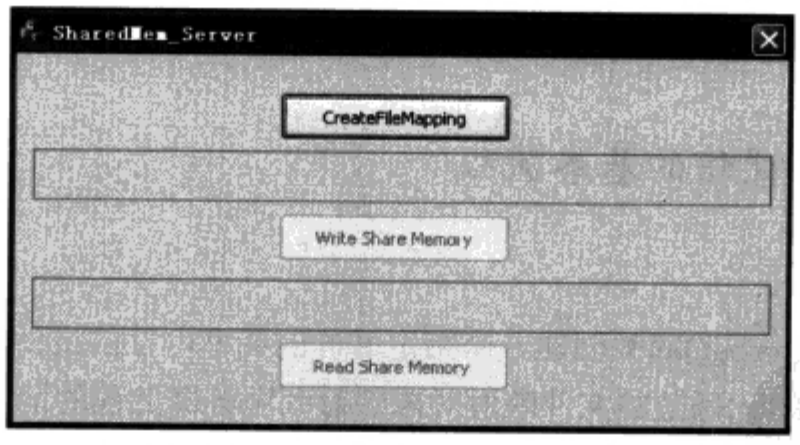


图 16.13 共享内存服务器端的程序界面

在这里，共享内存的名称没有任何限制，只要满足一般文件的命名规则即可，另外还允许包含目录。

在程序退出前需要关闭视图，关闭句柄，这在析构函数内进行，代码如下：

```
CSharedMem_ServerDlg::~CSharedMem_ServerDlg()
{
    if(m_pBuf != NULL){
        ::UnmapViewOfFile(m_pBuf);
        m_pBuf = NULL;
    }
```

```
    if(m_hMapFile != NULL){
        ::CloseHandle(m_hMapFile);
        m_hMapFile = NULL;
    }
}
```

创建共享内存的 CreateFileMapping 按钮的单击事件处理函数如例 16.16 所示。

例 16.16

```
void CSharedMem_ServerDlg::OnBnClickedButtonCreatefilemapping()
{
    // TODO: Add your control notification handler code here
    m_hMapFile = ::CreateFileMappingW(
        INVALID_HANDLE_VALUE,      // 创建共享内存
        NULL,                      // 默认安全属性
        PAGE_READWRITE,            // 可读写访问
        0,                         // 最大对象空间
        BUF_SIZE,                  // 缓冲大小
        m_strMapFileName);         // 映射对象名
    if (m_hMapFile == NULL)
    {
        AfxMessageBox(_T("Could not create file mapping object (%d)"),GetLastError());
        return;
    }
    //将共享内存映射到服务器地址空间的视图
    m_pBuf = (LPTSTR) ::MapViewOfFile(m_hMapFile,          // 映射对象句柄
        FILE_MAP_ALL_ACCESS,       // 可读/写访问
        0,
        0,
        BUF_SIZE);
    if (m_pBuf == NULL)
    {
        AfxMessageBox(_T("Could not map view of file (%d)"), GetLastError());
        return;
    }
    this->GetDlgItem(IDC_BUTTON_CREATEFILEMAPPING)->EnableWindow(FALSE);
    this->GetDlgItem(IDC_BUTTON_WRITESHAREMEM)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_READSHAREMEM)->EnableWindow(TRUE);
    ((CEdit*)this->GetDlgItem(IDC_EDIT_WRITESHAREMEM))->SetReadOnly(FALSE);
}
```

这里的主要操作就是先调用 CreateFileMappingW()函数创建一个共享内存，这里设置为可读写模式，然后调用 MapViewOfFile()函数将共享内存映射到服务器进程的地址空间内的一个视图，并返回一个指向视图起始地址的指针，通过该指针就可以对共享内存进行读/写操作。

读/写共享内存的操作都通过 CopyMemory()函数来完成，就是直接对某一块内存进行读/写。向共享内存写入数据的 Write Share Memory 按钮的单击事件处理函数如例 16.17 所示。

例 16.17

```
void CSharedMem_ServerDlg::OnBnClickedButtonWritesharemem()
{
    // TODO: Add your control notification handler code here
```

```
    CStringW strWriteMsg;
    this->GetDlgItemTextW(IDC_EDIT_WRITESHAREMEM,strWriteMsg);
    ::RtlCopyMemory((PVOID)m_pBuf, (LPCTSTR)strWriteMsg, (strWriteMsg.GetLength()+1)*sizeof(TCHAR));
    this->SetDlgItemTextW(IDC_EDIT_WRITESHAREMEM,_T(""));
}
```

由于本程序选择支持 Unicode 编码，所以函数对应变为 RtlCopyMemory()。

从共享内存读取数据的 Read Share Memory 按钮的单击事件处理函数如例 16.18 所示。

例 16.18

```
void CSharedMem_ServerDlg::OnBnClickedButtonReadsharemem()
{
    // TODO: Add your control notification handler code here
    TCHAR buff[BUF_SIZE];
    ::RtlCopyMemory((PVOID)buff, m_pBuf, BUF_SIZE);
    CStringW strTemp(buff);
    this->SetDlgItemTextW(IDC_EDIT_READSHAREMEM,strTemp);
}
```

至此，服务器端的设计介绍完毕，下面学习客户端的设计。

2. 客户端的设计

客户端的运行界面如图 16.14 所示。

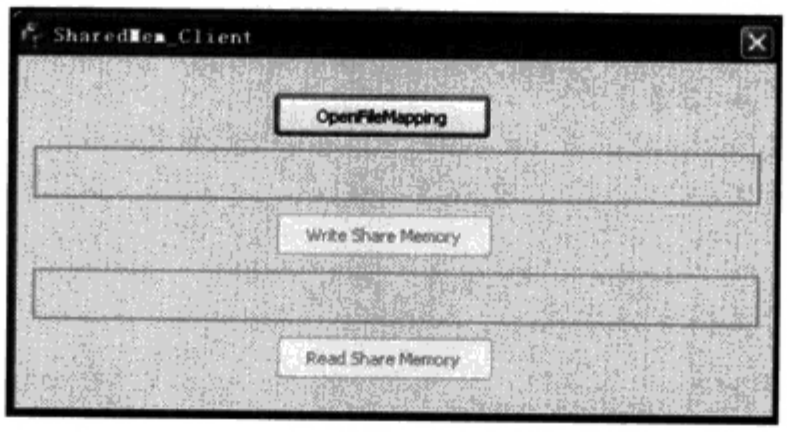


图 16.14　共享内存客户端的程序界面

上面的 OpenFileMapping 按钮主要功能是打开一个指定名称的共享内存，下面的文本框和按钮用于接收显示数据。

对话框的主要成员和服务器端一致，具体如下：

```
protected:
    HANDLE m_hMapFile;
    CStringW m_strMapFileName;
    LPCTSTR m_pBuf;
```

初始化代码如下：

```
CSharedMem_ClientDlg::CSharedMem_ClientDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CSharedMem_ClientDlg::IDD, pParent)
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    m_hMapFile = NULL;
    m_strMapFileName = _T("Global\\MyFileMappingObject");
    m_pBuf = NULL;
}
```

在析构函数内关闭句柄，释放资源。

```
CSharedMem_ClientDlg::~CSharedMem_ClientDlg()
{
    if(m_pBuf != NULL){
        ::UnmapViewOfFile(m_pBuf);
        m_pBuf = NULL;
    }
    if(m_hMapFile != NULL){
        ::CloseHandle(m_hMapFile);
        m_hMapFile = NULL;
    }
}
```

打开共享内存的 OpenFileMapping 按钮的单击事件处理函数如例 16.19 所示。

例 16.19

```
void CSharedMem_ClientDlg::OnBnClickedButtonOpenfilemapping()
{
    // TODO: Add your control notification handler code here
    m_hMapFile = ::OpenFileMappingW(
        FILE_MAP_ALL_ACCESS,          //可读/写访问
        FALSE,                        //不继承共享内存名称
        m_strMapFileName);
  if (m_hMapFile == NULL)
  {
        CString errMsg;
        errMsg.Format(_T("Could not Open file mapping object(%d)"),::GetLastError());
        AfxMessageBox(errMsg);
        return;
    }
    m_pBuf = (LPTSTR) ::MapViewOfFile(m_hMapFile,          // 映射对象句柄
        FILE_MAP_ALL_ACCESS,                               // 可读/写访问
        0,0,BUF_SIZE);
    if (m_pBuf == NULL)
    {
        CString errMsg;
        errMsg.Format(_T("Could not map view of file (%d)"),::GetLastError());
        AfxMessageBox(errMsg);
        return;
    }
    this->GetDlgItem(IDC_BUTTON_OPENFILEMAPPING)->EnableWindow(FALSE);
    this->GetDlgItem(IDC_BUTTON_WRITESHAREMEM)->EnableWindow(TRUE);
    this->GetDlgItem(IDC_BUTTON_READSHAREMEM)->EnableWindow(TRUE);
    ((CEdit*)this->GetDlgItem(IDC_EDIT_WRITESHAREMEM))->SetReadOnly(FALSE);
}
```

这里首先调用 OpenFileMappingW()函数打开一个指定名称（必须与服务器端的名称相同）的共享内存，然后调用 MapViewOfFile()将该共享内存映射到客户端进程的地址空间内的一个视图，返回指向视图起始地址的指针，通过该指针来对共享内存进行读/写操作。

读/写操作与服务器端相似，都是通过 CopyMemory()函数完成，写操作的 Write Share Memory 按钮的单击事件处理函数如例 16.20 所示。

例 16.20

```
void CSharedMem_ClientDlg::OnBnClickedButtonWritesharemem()
{
    // TODO: Add your control notification handler code here
    CStringW strWriteMsg;
    this->GetDlgItemTextW(IDC_EDIT_WRITESHAREMEM,strWriteMsg);
    ::RtlCopyMemory((PVOID)m_pBuf, (LPCTSTR)strWriteMsg, (strWriteMsg.GetLength()+1)*sizeof(TCHAR));
    this->SetDlgItemTextW(IDC_EDIT_WRITESHAREMEM,_T(""));
}
```

读共享内存的 Read Share Memory 按钮的单击事件处理函数如例 16.21 所示。

例 16.21

```
void CSharedMem_ClientDlg::OnBnClickedButtonReadsharemem()
{
    // TODO: Add your control notification handler code here
    TCHAR buff[BUF_SIZE];
    ::RtlCopyMemory((PVOID)buff, m_pBuf, BUF_SIZE);
    CStringW strTemp(buff);
    this->SetDlgItemTextW(IDC_EDIT_READSHAREMEM,strTemp);
}
```

至此，整个程序的设计介绍完毕，编译并运行程序。在运行程序时，首先运行服务器端程序，创建一个共享内存后，再运行客户端程序，打开共享内存，然后就可以对共享内存进行读/写实现数据交换。整个程序的运行界面如图 16.15 所示。

可以看到，服务器端写入共享内存的数据可以被客户端读取，而客户端写入共享内存的数据也可以被服务器端读取。实际上任何一个打开共享内存的客户端进程都可以对共享内存进行读/写操作，从而实现在不同进程间传递数据。

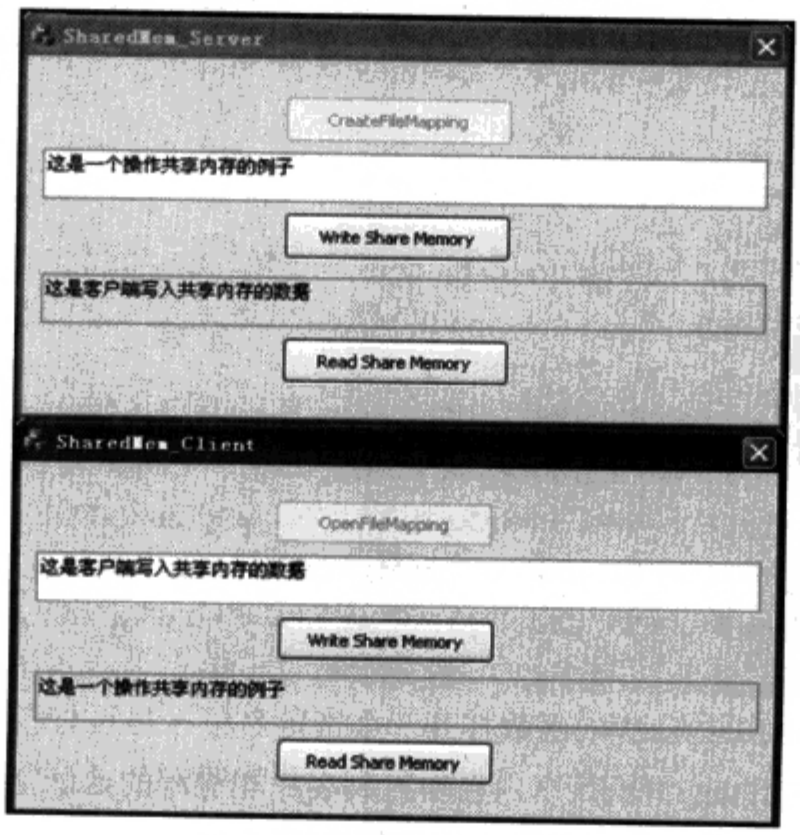


图 16.15　利用共享内存在服务器端和客户端之间通信的运行结果

16.6　小结

本章介绍了利用 MFC 提供的全局函数实现进程间通信的 5 种方式。其中剪贴板和匿名管道只能实现同一台机器上的不同进程间的通信，而无法实现远程机器的进程间通信；而命名管道和邮槽则可以实现跨网络的进程间通信。另外，邮槽还可实现一对多通信，但邮槽的缺点是每次能发送的数据量很小，受到严格的限制（通常在 424 字节以下）。实际上，剪贴板、邮槽、命名管道都是利用共享内存在进程间交换数据，只是进行了不同的封装和完善以适应不同的需求。最后介绍了如何直接操作（低层的）共享内存实现进程间的通信。

第三部分　Visual C++ 2010 下 MFC 与 CLR 进行开发

第 17 章　Visual C++ 2010 CLR 开发基础

微软的.NET 框架引入了许多新的概念、技术和术语。本章的目的是让读者对.NET 框架的体系结构有一个总体的认识，并对.NET 框架中出现的一些新技术和术语有一个基本的了解。.NET 框架包括编程语言和执行平台，以及提供内置功能的强类库。本章将向读者展示将源代码生成为应用程序的全过程，同时对它们的工作原理进行详细解释。

17.1　什么是.NET

.NET 框架（.NET Framework）是构建并运行应用程序的平台，支持多种不同的编程语言和库联合工作，以创建基于 Windows 的应用程序，使得创建、管理、开发以及与其他网络系统的集成更加容易。.NET 的核心思想是在不同的计算机系统之间更容易地通信。使用.NET 技术可以使应用程序在整个 Internet 上都可用。.NET 使用基于 XML 的被称为 Web service 的技术，这使得.NET 可以和不是基于.NET 架构的系统相互通信，前提是系统能够支持 XML。所以基于.NET 的应用程序可以和 UNIX、Linux、Mac OS 等可以支持 XML 的操作系统通信。

Microsoft .NET 主要是关于 XML Web 服务的，但是.NET 框架同样支持其他的编程模型。除了 Web 服务外，还可以编写控制台应用程序、GUI（Windows 窗体）应用程序、Web 窗体应用程序、Web 服务客户端等。当然，用.NET 编写的应用程序不一定必须使用 Web 服务。

17.2　.NET 框架

.NET 框架是用于简化在高度分布式 Internet 环境下进行应用程序开发的一个计算平台。Framework 涵盖了在操作系统上进行软件开发的所有层次，并对不同层次间的集成提供了功能丰富的级别，如表示技术、组件技术和数据库技术等。Framework 实际上是操作系统之上的一个包装器，它将开发人员与 OS 中的细节问题隔绝开来。在不同的平台上开发.NET 语言编译器，可以移植到不同的平台和硬件环境中。

.NET 框架主要由如下 3 个层次组成。

- 应用开发技术，如 ASP.NET、Windows Forms、ADO.NET、Windows Presentation Foundation、Windows Communication Foundation、Windows Workflow Foundation、Windows CardSpace 及 LINQ。
- .NET 框架类库。
- CLR（Common Language Runtime，公共语言运行时）。

如图 17.1 所示，每一层都依赖于其下方的一层（提供的服务）。最下面一层是 CLR，直接架构在操作系统之上，并将操作系统的细节向上层屏蔽掉，相当于对上层建立了一个虚拟运行环境（正如 Java 语言上的虚拟机 JVM），CLR 负责加载、验证并执行 Web 服务、ASP.NET 应用、控制台应用等，所

有的.NET 应用程序都只能在 CLR 下运行。

.NET 基类库包含了大量的由多个命名空间分隔的提供了访问系统功能的类，并设计作为建立.NET 应用程序、组件和控件的基础。这些类提供了处理诸如文件 IO、数据库访问、安全管理、线程、图形化用户接口（GUI）等功能。作为一个 C++/CLI 的开发人员，应该花时间学习这些类的用法。

应用开发技术提供了比基类库更高层次的抽象，C++/CLI 开发人员可以使用这些技术开发基本的多种应用，当然也可以跳过这一层而直接使用基类库进行开发，实现更多的控制。

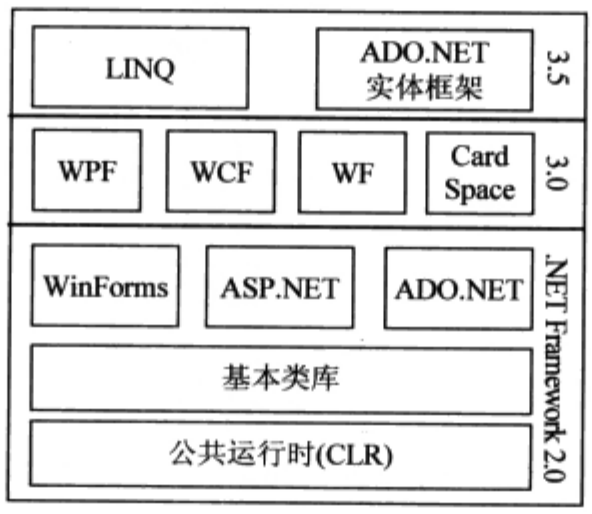


图 17.1　.NET 框架的组成

.NET 框架使用诸如 SOAP 和 XML 技术作为底层通信机制，使得.NET 的应用可以在 Internet 上通用。而且.NET 支持多种编程语言，其关键在于.NET 编译器，对于不同的编程语言使用不同的.NET 编译器，但是所有的.NET 编译器都将程序编译成一种统一的 Microsoft 中间语言，也简写为 MSIL 或 IL。开发人员可以使用诸如 C++/CLI、C#、J#、Visual Basic、COBOL、Dephi、Perl 等多种编程语言开发应用程序，这些.NET 编译器支持的语言都可以完全访问.NET 基类库。

17.3　公共语言运行时（CLR）

CLR 位于操作系统之上，抽象操作系统服务，并为宿主托管应用程序提供虚拟环境。CLR 是.NET 框架的基础，负责在程序运行时加载并管理代码，提供如内存管理、线程管理和远程化等核心服务。代码管理是运行时的基本原理。

运行时并不是一个新概念，早在 Visual Basic 6.0 中就有 msvbvm60.dll，Java 中有 Java 虚拟机（JVM）。而公共语言运行时（CLR）则是.NET 的运行系统。正如其名称所示，CLR 能够支持多种不同的编程语言及平台的运行时，当然前提是得有对应的编译器（先将它转化为 MSIL）。

运行时至少扮演两种角色：运行代码以及为大多数的应用添加通用调用。CLR 除了为.NET 平台提供这两种几本功能外，还提供其他多种服务，如代码验证、访问安全性检查、自动垃圾回收、异常处理、多语言支持以及将 MSIL 实时编译成本地 CPU 的可执行机器代码。CLR 的工作流程如图 17.2 所示。

在.NET 框架内启动一个应用程序相当简单：首先 CLR 加载应用程序集，以及任何引用的开发程序集和基类库程序集；接着应用程序类有选择地进行类型验证和合法访问安全检查；然后依据程序集内包含的元数据信息加载 MSIL 代码，并调用 JIT 编译器将其实时编译成本地机器代码；最后就可以运行应用程序了。

对.NET 应用程序源代码进行编译得到的目标代码并不能直接在处理器上运行，而是被转换为 Microsoft 中间语言（Microsoft Intermediate Language，MSIL 或 IL），这是一种与 CPU 无关的指令集。MSIL 包含了对对象的加载、存储、初始化和调用等方法的指令，同时也包含用于算术和逻辑操作、流

程控制、直接内存访问、异常处理等操作。

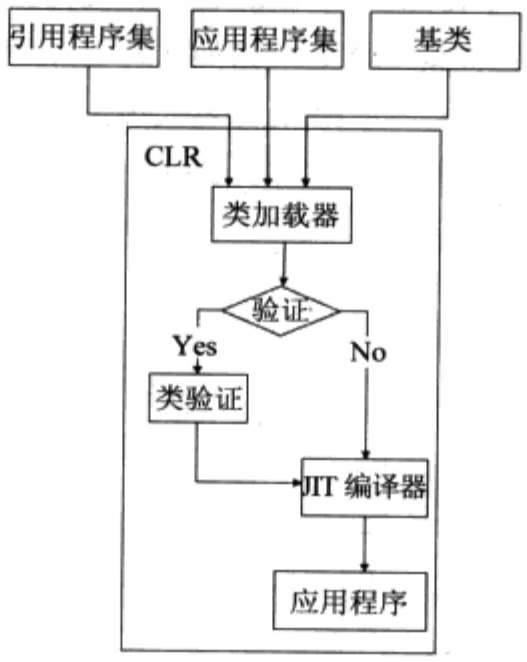


图 17.2　CLR 的工作流程

在运行代码之前，CLR 还必须通过一个即时（Just In Time，JIT）编译器将 MSIL 转换成特定 CPU 的机器码。只要对不同体系结构都提供一种或多种 JIT 编译器，那么同一个 MSIL 集被对应的 JIT 转换后完全可以在不同的体系结构上运行。在大部分情况下，给定方法只进行一次实时编译，即当它第一次被调用时，其后便被缓存在内存中以便下次执行时没有延迟。未调用的代码决不会被实时编译，这与解释型的代码是不同的。

在一个编译器产生 MSIL 时，它还将同时产生元数据（meta data）。元数据描述了代码的类型，包括每种类型的定义、每种类型成员的签名、代码引用的成员以及在执行代码时使用的其他数据。MSIL 与文件中元数据为代码提供了自我描述，这就意味着不再需要类型库或接口定义语言。CLR 会在执行代码期间根据需要从文件中定位和提取元数据。

元数据具有如下优点。

- 元数据省去了源代码编译时对头文件和库文件的需求，这是因为含有实现类型和成员的 IL 代码文件中已经包含了所有被引用的类型和成员的信息，编译器可直接从托管模块中读取元数据。
- Visual Studio 2010 就是利用元数据来辅助编写代码。其智能感知特性就是通过分析元数据来告诉开发者某种类型使用了哪些方法，以及某个方法需要哪些类型的参数。
- CLR 代码验证过程利用元数据来确保代码的安全。
- 元数据允许一个对象的字段被序列化到一个内存块中，并远程传送到另一台机器，然后在远程机器上反序列化，重新建立对象及其状态。
- 元数据允许垃圾回收器追踪对象的生命期。对于任何对象，垃圾回收器都能够根据元数据确定该对象的类型，以及该对象的哪些字段引用了其他的哪些对象。

可以使用任何支持 CLR 的编程语言来创建源代码文件，然后用相应的编译器来做语法检查和源代码分析。不管使用何种编译器，最终产生的都是一个托管模块（managed module）。托管模块是需要 CLR 才能执行的标准 Microsoft Windows 可移植执行（portable executable）文件。托管模块的组成如表 17.1 所示。

表 17.1　　托管模块的组成

部　分	描　述
PE 表头	标准 Windows PE 文件表头，类似于通用对象文件格式（COFF）表头。该表头指出文件的类型：GUI、CUI 或 DLL。另外，还有一个时间标记来表明该文件的创建时间。对于仅包含 IL 代码的模块，PE 表头的大多数信息将被忽略；对于包含本地 CPU 代码的模块，该表头将会包含有关本地 CPU 代码的信息

续表

部　分	描　述
CLR 表头	包含标识托管模块的信息。该表头包含需要的 CLR 的版本、一些标记、托管模块入口点方法（Main()方法）的 MethodDef 元数据标记，以及有关该模块的元数据、资源、强命名、标记等信息
元数据	每个托管模块内都包含元数据表。有两种主要类型的表：一种用于描述源代码中定义的类型和成员；另一种用于描述源代码中引用的类型和成员
中间语言（IL）代码	编译器在编译源代码时产生的 IL 指令，随后 CLR 会将这些 IL 指令变成本地 CPU 的机器指令

17.3.1　托管代码

针对.NET 框架的 CLR 代码称为托管代码（managed code），提供内存管理、代码访问安全以及多语言集成功能。对应的本地机器代码被称为非托管代码，这不是针对 CLR 的。本地代码在 CLR 以外运行，这就意味着将失去 CLR 提供的自动垃圾回收和代码安全检查等优点。

托管代码受益于以下一些特性：跨语言集成、跨语言异常处理、增强的安全性、版本管理和配置支持、组件交互的简化模型、调试以及特性描述服务等。为了使用这些服务，编译器必须生成元数据。元数据描述了类型、成员以及对代码的引用情况等，元数据将与编译产生的 MSIL 代码一起存储，任何可加载的公共语言运行时映像都将包含元数据。运行时将使用元数据对类进行定位和加载，在内存中分配实例，解决方法调用，生成本机代码，强制安全性以及设置运行时的上下文边界等。运行时自动处理对象布局以及管理对对象的引用，并在它们不再被使用时将其释放。以这种方式管理生存期的对象称为托管数据（managed data）。

除了加载和运行程序集以外，CLR 还提供其他服务。代码验证和代码访问验证为在程序集被加载进来之前的可选服务，而自动垃圾回收在程序集运行时总是活动的。

17.3.2　代码验证

代码验证在应用程序实际运行之前进行，其目标是遍历检查整个代码，确保代码是安全可执行的。它主要检查代码以确保以下几个方面。

- 代码没有被替换。
- 代码创建好后没有被篡改。
- 没有被同样代码的其他版本重写。

17.3.3　代码访问验证

代码访问验证同样需要遍历整个代码，检查所有代码是否都有运行许可。该服务的目标是尽量阻止对用户主机的恶意攻击。CLR 内包含了一个所有许可的行为列表，而程序集内则包含了所有运行代码所要求的许可列表。如果 CLR 可以允许程序集内的所有许可，那么程序集可以毫无问题地运行，但是如果 CLR 不允许所有的许可，那么在代码运行过程中一旦执行了没有被许可的行为，都将产生异常。

17.3.4　垃圾回收

垃圾回收使得运行时可以动态检测并从托管堆（managed heap）上释放那些不会再被应用程序访问的托管对象。当垃圾回收器完成释放对象的工作后，它会通过压缩堆的方式整理垃圾，然后用连续

的内存空间满足新的分配请求，这样的方式会更加高效。

垃圾回收为 C++/CLI 程序员免除了一个极大的麻烦。C++程序员经常由于内存泄露而感到苦恼，泄露是由于没能及时将不用的资源返还给系统造成的。如果太急于返还资源，也可能使情况变得更糟，因为可能会访问到无效的内存区域。CLR 的内存回收机制使得程序员在处理动态内存时不用再为在何处调用 delete 命令而苦恼，只要在创建对象时使用 gcnew 命令，那么 CLR 将托管所有的资源回收问题。

17.3.5 语言的互操作性

为了与.NET 兼容，并确保各种应用程序之间的互操作性，所有.NET 语言都必须遵从微软的技术规范要求。.NET 语言必须是面向对象的，必须使用由“公共类型系统”（CTS）定义的一套标准数据类型。具体要求.NET 框架具有如下特性。

- 通用数据类型由所有语言共享，也就是说.NET 框架支持的所有编程语言都应该能够访问 CTS。
- 在任何一种语言中的对象句柄或引用都必须可以作为方法的参数进行传递。
- 允许调用在其他语言中开发的类的方法。
- 允许在类中包含由其他语言开发的类的实例。
- 允许继承由其他语言开发的类。
- 开发和调试环境对所有的编程语言都是一致的。

从 CLR 的角度来看，所有语言都是平等的，只要有一个能生成 MSIL 代码的编译器即可。同样，一旦应用程序被编译为 MSIL 代码，它就与原代码的语言无关了，这样就可以轻松地调用其他语言编写的库函数，调用过程中传递的参数和返回值也是兼容的。前面介绍过，任何一种.NET 兼容的语言都必须能够被编译成 MSIL 代码和元数据并被关联到程序集内，其中 MSIL 代码告诉 CLR 执行什么操作，而元数据则提示如何执行（使用哪个接口）。图 17.3 演示了源代码文件的编译过程。

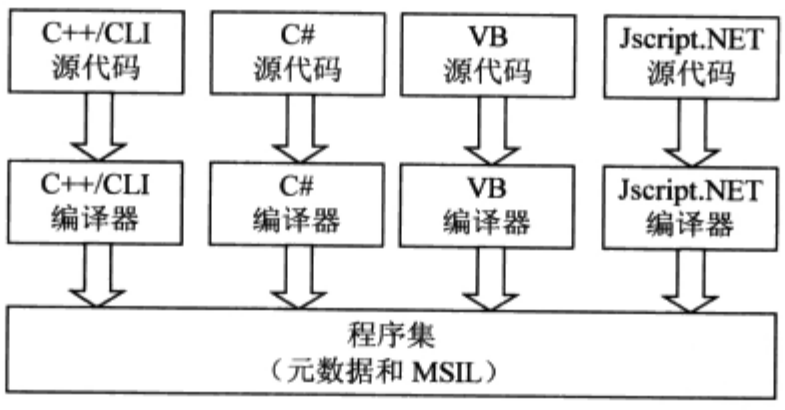


图 17.3 不同语言源代码文件的编译过程

17.3.6 实时编译（JIT）

尽管.NET 应用程序都要被编译成一种统一的中间语言（MSIL），但是.NET 应用程序绝不是解释型的，相反它们最终还是会被进一步编译为本地的可执行机器代码。这部分工作就是由 JIT 编译器负责，在程序集内元数据的帮助下将 MSIL 代码转换成机器代码。

当一个应用程序被启动运行时，JIT 编译器将被 CLR 调用以将 MSIL 代码转换成机器代码。为了避免在启动时就将整个应用程序编译导致开销过大，JIT 编译器只对那些确实被应用程序调用的代码部分进行转换。转换完成的代码将被放到缓冲内存中然后就可以运行。在整个运行过程中，这些代码都被放在缓存中，这样以后的运行就直接从这里开始，而不用每次调用时都重新编译再运行。另外，只有那些真正被调用的代码才会被 JIT 转换，没有被调用的代码不会被 JIT 转换，这样保证了执行效率。

JIT 在转换代码时会针对不同的处理器类型进行优化，所以即使每次运行代码时都会因为 JIT 带来一些开销，但是由于 JIT 的优化编译，使得.NET 程序的运行速度几乎可以和直接运行机器代码的速度相当，甚至可以超过非托管代码的运行速度，所以用户不必担心这些开销。

17.4　通用类型系统（CTS）

通用类型系统（CTS）定义了所有的类型在.NET 框架（准确地说 CLR）上的定义、使用及管理，同时 CTS 也是 CLR 多语言支持的关键部件。CTS 主要用来支持以下特性。

- 为所有.NET 框架支持的编程语言都提供一种面向对象的数据模型，CTS 内定义的所有类型都可被这些语言使用。
- 为所有.NET 支持的编程语言提供一套使用这些数据类型的限制，使得这些语言编写的应用程序可以同其他语言编写的程序通信。
- 为所有.NET 支持的编程语言提供整合和数据类型安全性。

CTS 内定义了两种不同的数据类型：值类型和引用类型。值类型如 int、float 等，其值直接存放在变量所代表的空间内（即放在栈上）；而引用类型如句柄、类、数组等，其值被存放在托管堆空间内，而变量所代表的空间本身则存放指向这些值的引用。CTS 内常用的数据类型分类如图 17.4 所示。

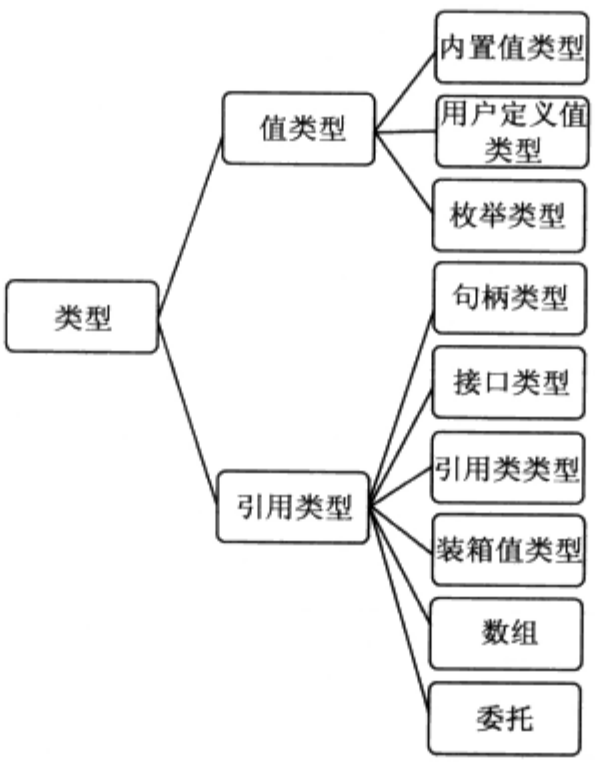


图 17.4　CTS 内常用的数据类型

下面简单介绍这些类型。

- 数组：为一个一维或多维的可索引的类型集合。
- 装箱值类型：一个临时的指向值类型的引用，这样其值就可以被放到托管堆上。
- 内置值类型：为主要的值类型，代表整数、实数、布尔值及字符等。
- 引用类类型：用户自定义的类及方法，将在托管堆上创建对象。
- 委托：这是一种引用类型，其指向的对象是一个方法。
- 枚举类型：类似于 C++的枚举类型，代表相关的整数常量的列表，但是比 C++中的枚举类型更加复杂。
- 接口类型：为一个类类型，该类内的所有方法都是抽象的。
- 句柄类型：为一个引用，指向一种类型的引用。

- 用户定义值类型：用户自定义的对标准的主要值类型的扩展。

需要指出的是 CTS 内定义了所有.NET 支持的语言的所有数据类型，但是 CTS 内定义的类型可能不会被某一种语言支持。表 17.2 列举出了 CTS 内常用的类型及其对编程语言内的支持类型。

CTS 规范规定一个类型可以包含 0 个或多个成员，这些成员可以为以下类型。

- 字段：字段是一个属于对象状态部分的数据成员，字段由它们的名称和类型标识。
- 方法：方法是一个在对象上执行某种操作的函数，通常会改变对象的状态。方法有一个名称、签名和修饰符。方法的签名指定了方法的调用约定、参数个数、类型及返回值类型等。
- 属性：对于调用者，属性看起来类似于字段；但是对于类型的实现者，属性看起来更像是一个方法。属性允许实现者在访问数据之前验证输入参数和对象状态的有效性，或者仅在需要的情况下进行求值运算。属性使得类型的用户可以使用简化的语法来表达他们的意图。另外，可以创建只读、只写或读写属性。
- 事件：事件允许在一个对象和其他相关联的对象之间建立一个通知机制。例如，一个按钮可以提供一个事件，当按钮被按下时，其他对象将得到通知。

表 17.2　　CTS 内常用的类型及在不同编程语言内的对应类型

基本类	Visual Basic	C#	C++/CLI
System::Byte	Byte	byte	unsigned char
System::Sbyte	Sbyte	sbyte	char
System::Int16	Short	short	short 或__int16
System::Int32	Integer	int	int, long 或__int32
System::Int64	Long	long	long long 或__int64
System::UInt16	Ushort	ushort	unsigned short 或 unsigned __int16
System::UInt32	Uinteger	uint	unsigned int, unsigned long 或 unsigned __int32
System::UInt64	Ulong	ulong	unsigned long long 或 unsigned __int64
System::Single	Single	float	float
System::Double	Double	double	double
System::Object	Object	object	Object^
System::Char	Char	char	__wchar_t
System::String	String	string	String^
System::Decimal	Decimal	decimal	Decimal
System::IntPtr	IntPtr	IntPtr	IntPtr
System::UIntPtr	UIntPtr	UIntPtr	UIntPtr
System::Boolean	Boolean	bool	bool

CTS 还定义了类型的可见性和访问类型成员的规则，由于存在程序集的概念，因此 CTS 的访问比 C++更复杂。CTS 建立了以程序集作为类型可见性边界的规则，而 CLR 实现了这种可见性规则。下面列出对于方法或字段进行访问控制的选项。

- Public：方法或字段对于任何程序集内的任何代码都可见（或可访问）。
- Assembly：方法或字段只对同一个程序集内的代码可见。
- Family：方法或字段对所有派生类都可见，而不管派生类是否属于同一个程序集；而对于同一个程序集内的其他类，如果不是其派生类，是不可见的。这类似于 C++内的 protected。
- Family 或 Assembly：方法或字段对同一个程序集内的代码可见或对所有派生类可见。
- Family 与 Assembly：方法或字段只对同一个程序集内的派生类型可见。
- Private：方法或字段只在同一个类型内可见，类似于 C++的 private。

CTS 还定义了诸多规则来管理类型继承、虚函数及对象生存周期等。但是我们大可不必学习这些

规则，因为我们所选择的语言语法和类型规则会在生成托管模块时自动映射为 CLR 语言的规则。

另外，在 CTS 中所有的类型都直接或间接地继承自系统预定义类型 System.Object。Object 类是定义在 System 命名空间中的一个类型的名称，Object 类作为所有其他类型的根类保证了每个类型实例都有一个公共行为的最小集合。这些行为包括：判断两个实例是否相等，获取实例的散列值，查询实例的类型，执行实例的浅拷贝，获取当前实例状态的字符串表示。

17.5　通用语言规范（CLS）

前面介绍过 CTS 的所有特性并不是每一种.NET 编程语言都全部支持。为了在不同的语言之间保持兼容，需要定义一种所有语言都支持的最小 CTS 子集，这就是通用语言规范（Common Language Specification, CLS）的作用。CLS 为所有编译厂商详细描述了面向 CLR 所必须支持的一个最小特性集合，使得在 CLR 所集成的所有编程语言内，允许一种语言创建的对象在另一种不同语言编写的代码中被看做同等的成员。

CLR/CTS 支持的特性比 CLS 定义的子集丰富得多。如果不关心语言之间的互操作，则可以开发出所选语言的特性集合支持的非常丰富的类型。具体而言，CLS 定义了可以被任何 CLS 兼容的编程语言访问到的、外部可见的类型和方法所必须遵循的规则。图 17.5 说明了 CLS 与 CLR/CTS 的关系。

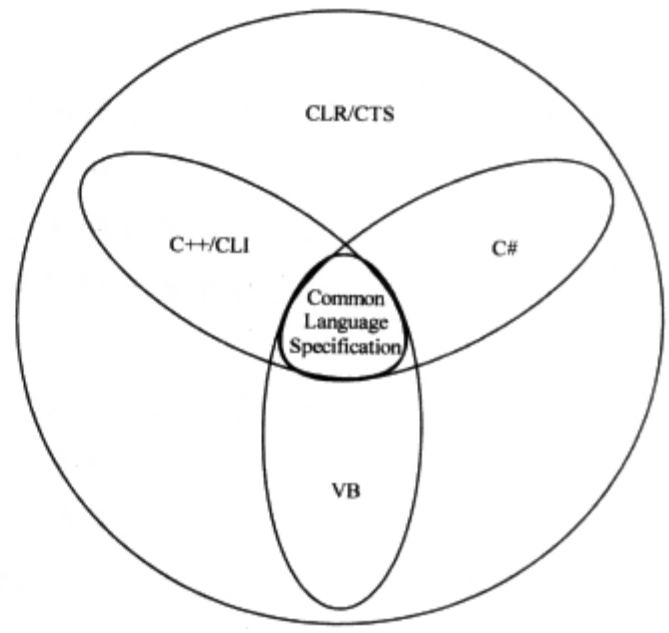


图 17.5　CLS 与 CLR/CTS 关系图

CLR/CTS 提供了一组特性，而一些语言会提供这些特性的一个较大的子集。例如，使用 IL 汇编语言的开发人员可以使用 CLR/CTS 提供的所有特性。大多数其他语言（如 Visual Basic、C++/CLI、C#等）提供给开发人员的将是 CLR/CTS 特性的一个子集。而 CLS 定义了所有.NET 语言必须支持的最小特性集合。

为了保证语言之间的互操作性，只有 CLS 定义的子集才能使用。如果希望采用某种语言设计的类型能够被其他语言使用，那么就不应该使用 CLS 之外的任何特性，那样做将意味着使用其他编程语言的开发人员可能访问不到该类型的成员。

对于 CLS 规范，必须注意以下几点。

- 全局变量和方法是不允许的。
- CLS 内定义的类型名是不区分大小写的。
- CLS 支持的简单数据类型只有 byte、int16、int32、int64、single、double、boolean、char、decimal、intptr 和 string。
- 变长的参数列表是不允许的，使用固定长度的数组替代。
- 指针是不允许的。
- 类类型必须从 CLS 包含的类继承，当然 System::Object 是所有类的基类。如果使用超出 CLS 规范的类或使用从这些类继承的子类，都将不再与 CLS 兼容。
- 数组的元素类型必须是 CLS 兼容的类型。

在 CLR 中，类型的每一个成员或者是一个字段，或者是一个方法。这意味着每一种编程语言都必须能够访问字段和调用方法。这些字段和方法以普通或特殊的应用方式展现出来。为了简化程序设计，

一些编程语言通常会提供某些额外的抽象，如一些语言提供了枚举、数组、属性、索引器、委托、事件、构造器、析构器、重载运算符、转换运算符等。当编译器在源代码中遇见这些结构时，必须将它们翻译成字段和方法，只有这样 CLR 和其他的编程语言才能访问它们。

17.6 程序集

程序集就是建立在一起形成一个逻辑功能单元的一组类型和资源，它是.NET 框架应用程序组织的基本单位。程序集是一种自描述的以中间语言形式存储的实现功能操作及运行应用程序时需要用到的资源。如图 17.6 所示，程序集主要由以下 4 个部分组成：

- 程序集元数据；
- 类型元数据；
- 微软中间语言（MSIL）代码；
- 资源。

程序集的一个重要特性就是它是自描述的，所有关于如何使用程序集的信息都包含在程序集内，更确切地说是由程序集内的元数据描述。程序集内包含两种不同类型的元数据：程序集元数据和类型元数据。除了程序集元数据外，其他部分都是可选的。

程序集分为私有程序集和共享程序集两种。

- 私有程序集：将被用于一个应用程序。像调用应用程序一样，私有程序集必须被置于应用程序自身所在的同一个文件夹下或者其子文件夹中，这种程序集对于文件系统下的其他应用程序是不可见的。如果一个应用程序使用了许多私有程序集，则所有程序集必须和该应用程序一起发布。创建一个私有程序集无须顾及命名冲突问题，惟一需要注意的是保证在应用程序中命名的惟一性。
- 共享程序集：将被存储在全局程序集缓存（Global Assembly Cache，GAC）内。GAC 实际上就是一个目录结构，里面存放所有对计算机全局可用的程序集。一个共享程序集可以被多个应用程序使用，其名称在整个系统中必须是惟一的。在系统盘的 WINDOWS\assembly 目录下可以找到所有的共享程序集，如图 17.7 所示。

图 17.6 程序集的结构

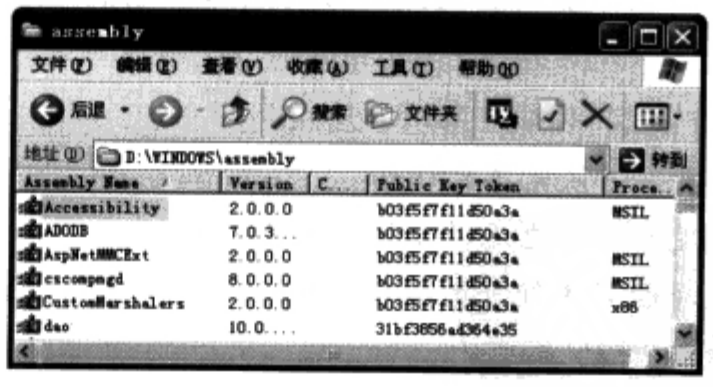


图 17.7 共享程序集

17.6.1 元数据

程序集元数据也被称为程序集清单，主要用来描述程序集自身的基本信息，如程序集的名称、版本号，程序集使用的语言文化、公钥和数字签名（用于惟一标识程序集的创建者 ID），程序集内包含的所有文件的列表，被引用程序集的列表，以及类型引用信息。

而类型元数据则描述程序集内部创建的类型信息。对于一个方法，元数据将包含方法名、返回值类型、参数列表的个数及各个参数的类型、代码访问的安全等级；对于一个属性，则涉及 get()和 set()两个方法的描述。

17.6.2　程序集版本管理

每个程序集的版本号都由 4 个部分组成（如 9.0.0.0），作为其标识的一部分，这是 CLR 实现版本无关性的关键。版本号的格式如下：

```
<major version>.<minor version>.<build number>.<revision>
```

任何对主版本号或次版本号的修改都被认为是非兼容修改。例如，如果从程序集中删除了一些类型和方法，至少需要修改次版本号；而调试 Bug 以及类似的操作则被认为是兼容修改，需修改构建和修订号。

每一个程序集都有一个版本号，如果应用程序集引用在 GAC 内的强命名程序集，那么将对程序集的版本号敏感：在运行的时候，CLR 会自动对这些应用程序集的版本号进行验证。如果版本号不匹配，程序集将被认为是不兼容的，在默认情况下将不被加载，CLR 将选取具有最高构建和修订号的程序集。

17.6.3　微软中间语言（MSIL）

前面介绍过，在.NET 框架中，源代码首先被编译成中间语言 MSIL，然后再由 JIT 实时编译成本地机器代码，才能运行。MSIL 是一种与 CPU 无关的类似于汇编语言的一系列指令的集合，例如它也包含算术及逻辑运算指令、控制流指令。但是不同于普通的汇编指令，它还包含“高层”的指令，如加载、存储、初始化及调用类对象的指令。下面是一个 MSIL 代码的例子，该程序就是最简单的 Hello World 程序生成的 MSIL 代码，如例 17.1 所示。

例 17.1

```
.method assembly static int32   main(string[] args) cil managed
{
  // Code size          26 (0x1a)
  .maxstack  2
  .locals ([0] int32 V_0,
           [1] int32 b,
           [2] int32 a,
           [3] int32 c)
  IL_0000:  ldc.i4.0
  IL_0001:  stloc.0
  IL_0002:  ldc.i4.s   10
  IL_0004:  stloc.2
  IL_0005:  ldc.i4.s   20
  IL_0007:  stloc.1
  IL_0008:  ldloc.2
  IL_0009:  ldloc.1
  IL_000a:  add
  IL_000b:  stloc.3
  IL_000c:  ldstr      "Hello World"
  IL_0011:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0016:  ldc.i4.0
```

```
    IL_0017:  stloc.0
    IL_0018:  ldloc.0
    IL_0019:  ret
  } // end of method 'Global Functions'::main
```

可以看到，MSIL 很像一般的汇编语言指令。值得一提的是，这是一种基于堆栈的语言，没有使用任何寄存器，任何要运算的操作数都必须先放到堆栈里，运算后再将结果从堆栈中取出（当然操作数在运算时会被自动弹出栈）。例如，ldloc.2 指令就是将操作数 2 放到栈顶，然后 ldloc.1 将操作数 1 放到栈顶，接着 add 指令将当前栈顶的两个操作数相加，这两个操作数被自动弹出栈，并将结果放到栈顶，下面的 stloc.3 指令将结果取回。

MSIL 的一个关键特性就是这是一种面向对象的“汇编语言”，只是加了限制只允许单继承。所有类型（值类型或引用类型）的使用都必须符合 CTS 规范，错误处理使用异常机制。同时，MSIL 还是实现.NET 框架语言无关性的关键。任何语言编写的代码都要先被编译成统一的 MSIL，在运行时 CLR 只能看到 MSIL，而根本不知道也不用知道是使用哪种语言编写的。

17.6.4 资源

在.NET 框架中，诸如字符串表、图像、图标等资源可以存放在两个不同的位置：在扩展名为.resources 的文件中或者在程序集的目录下。.NET 框架的基类库提供了 3 个类来方便地访问资源，这 3 个类都在 System::Resources 命名空间中。

- ResourceManager 类：用于访问程序集内的资源。
- ResourceWriter 类：用于向扩展名为.resources 的文件写入资源。
- ResourceReader 类：用于从扩展名为.resources 的文件内读取资源。

17.7 .NET 开发应用程序的范畴

可以使用.NET 开发的应用程序主要包括 6 种。

- 控制台应用程序：这是最简单的应用程序，对于较简单的用户界面需求，控制台程序提供了一种快速而轻便的创建方式。典型的编译器、实用程序和工具程序都被实现为控制台应用程序。对于所有的控制台应用程序，最关键的元素就是main()函数和System::Console::WriteLine()函数。
- Windows 窗体应用程序：即 Windows Forms，Windows 图形用户界面（GUI）应用程序。对于 C++/CLI 程序员而言，这几乎是最大的改变。C++/CLI 不再直接支持 MFC 类库，相应的，.NET 框架提供了更好的解决方案，那就是 Windows Forms。可以说 Windows Forms 为程序员创建窗体程序提供了像创建 Visual Basic 程序一样简单却能实现像 C++/CLI 一样强大的功能。

创建 Windows Forms 窗体程序时，应该使用繁大的 System::Windows::Forms 命名空间。对于 Windows 窗体应用程序，.NET 框架还提供了一种新的改进的图形设备接口（GDI），被称为 GDI+。通过 GDI+可以很方便地在应用程序中操作字体，改变颜色，绘制图形。而所有这些操作都集成在一个类 System::Drawing::Image 内实现，该类还允许加载多种图像文件格式，包括 GIF、JPEG 和 BMP 到内存中，进而对其进行操作并显示到屏幕上。

- Web 窗体：即 Web Forms。这是一种基于 HTML 的应用程序。Web 窗体应用程序通常会做一些数据库查询和 Web 服务调用，组合并筛选返回的信息，然后通过一个基于 HTML 的用户界面将信息显示在浏览器中。不同于传统的利用 ASP.NET 开发的 Web 窗体应用程序利用解释型

脚本运行，.NET 支持编译运行的应用程序，而且这些应用程序通常使用 C#和 Visual Basic 语言来开发。不幸的是，C++/CLI 并不直接支持通过 Visual Studio 来开发 Web 窗体应用程序。

- Web 服务：又称为 XML Web 服务。该服务使得程序员可以通过使用一个简单的 HTTP 请求就能跨越互联网来调用远程机器上的方法。该服务基于 XML 技术，更确切地说基于 XML 驱动的简单对象访问协议（Simple Object Access Protocol，SOAP）。SOAP 被用来实现在无中心（服务器）的分布式环境中使用 HTTP 来交换信息。开发 Web 服务应用程序，用到的主要是 System::Web::Services 命名空间。
- Windows 服务：一个 Windows 服务实际上就是一个特殊的 Windows 窗体应用程序，它是在操作系统启动时自动运行，当然也可以手动启动 Windows 服务。这是由 Windows 服务控制管理器（Service Control Manager，SCM）控制的服务应用程序。
- 组件库：.NET 框架允许创建单独的组件类型，它们可以应用于上面提到的各种应用程序。.NET 框架类库包含了大量的类型，这些类型被部署到不同的命名空间中。例如，System 命名空间包含了 Object 基类型，所有其他类型都必须直接或间接从该类继承。另外，System 还包括了整数、字符、字符串、异常处理、控制台输入输出等实用类型，它们可以用来安全地转换数据类型、格式化数据类型等操作，几乎所有的应用程序都要用到 System 命名空间中的类型。

17.8　.NET 框架类库

.NET 框架中包含了一组.NET 框架类库（Framework Class Library，FCL）程序集，FCL 包含了数以千计的类和结构，每个类都提供了某些特定的功能，这些类通过不同的命名空间组织分开。对于一般的程序只要直接使用这些类，而对于特殊的需求则可以从这些类继承。.NET 框架类库包含了两种主要的命名空间层次：一种是平台无关的 System 命名空间，另一种就是针对微软平台的 Microsoft 命名空间。表 17.3 所示为一些最常用的命名空间，并简要描述了用于命名空间的类型。

表 17.3　　NET 框架中常用的命名空间

命 名 空 间	描　　述
Microsoft::Win32	包含一些处理由操作系统触发的事件以及操作系统文档的类
System	包含处理基本类型、数学运算的类
System::Collections	包含定义对象的集合的类，如数组、队列、列表、哈希表、字典等
System::Configuration	包含访问 app.config 和 web.config 文件中配置数据的类
System::Data	包含处理数据库访问的类，该命名空间中的类型合在一起被称为 ADO.NET
System::Data::OleDb	包含处理 OLE DB 数据库访问的类
System::Data::OracleClient	包含处理 Oracle 数据库访问的类
System::Data::SqlClient	包含处理 Microsoft SQL Server 数据库访问的类
System::Diagnostics	包含帮助调试应用程序及追踪应用程序执行的类
System::Drawing	包含处理 GDI+图形操作的类
System::Drawing::Drawing2D	包含处理高级二维及向量图形操作的类
System::Drawing::Imaging	包含处理高级 GDI+图像显示等操作的类
System::Drawing::Printing	包含处理定制用户打印设置的类
System::Globalization	包含支持多国语言的类，如字符串比较、格式化及日历功能
System::IO	包含处理读写数据流及文件操作的类
System::Media	包含用于播放.wav 格式的音频文件或系统音频的类
System::Net	包含处理网络通信中的多种协议和服务的类

续表

命 名 空 间	描　　述
System::Reflection	包含用于查看元数据与延迟绑定类型及其成员的类
System::Resource	包含用于创建、存储及管理多种特定资源的类
System::Runtime::InteropServices	包含访问非托管的 COM 对象及本地系统 API 的类
System::Runtime::Remoting	包含用于创建和配置分布式应用程序的类，主要用于从远程机器上访问类型
System::Text	包含代表多种字符编码方式的类，用于将字节块编码成字符串或从字符串解码成字节块
System::Security	包含管理 CLR 安全系统的类，用于保护数据和资源
System::Threading	包含处理多线程编程的类
System::Timers	包含用于在特定时间间隔定时触发事件的类
System::Web	包含处理在浏览器和服务器间通信的类
System::Web::Mail	包含通过使用 SMTP 邮件服务创建及发送 E-mail 的类
System::Web::Security	包含处理在 Web 窗体应用程序中的安全访问的类
System::Web::Services	包含用于创建及使用 Web 服务的类
System::Web::UI	包含用于在 Web 窗体应用程序中创建控件及页面的类
System::Windows::Forms	包含用于创建基于 Windows 的应用程序的类
System::XML	包含用于处理 XML 模式和数据的类

17.9　C++/CLR 开发语法简介

在介绍 C++/CLR 语法之前，先看看经典的 Hello World 程序在.NET 框架上的实现。使用 Visual Studio 2010 创建一个 CLR 控制台程序相当简单，首先新建一个工程，在左侧选择 CLR，并选择 CLR Console Application，表示新建一个控制台程序，并输入工程名称，然后单击 OK 按钮，如图 17.8 所示。这里使用的版本是.NET Framework 4.0。

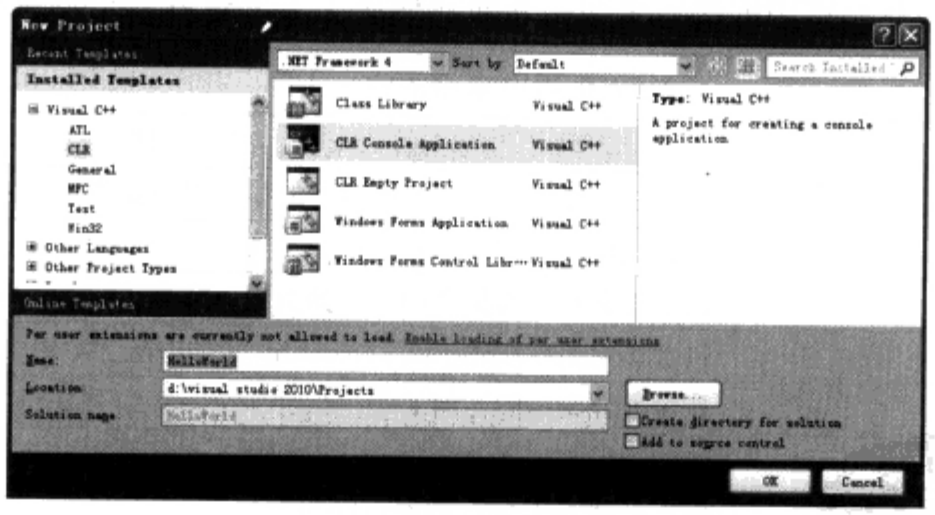


图 17.8　新建一个 CLR 命令行应用程序

这样向导将自动生成输出 Hello World 的程序，主要代码如例 17.2 所示。

例 17.2

```
#include "stdafx.h"
using namespace System;
int main(array<System::String ^> ^args)
{
    Console::WriteLine(L"Hello World");
    return 0;
}
```

这里首先使用与 C++一样语法的#include 预处理来包含需要的头文件，下面的语句使用在.NET 类库内定义的命名空间 System：

```
using namespace System;
```

前面介绍过.NET 框架类库本身就被组织为一系列的命名空间，所有最常用的类都很可能在 System 命名空间内。如果不使用 using 指令，将必须对所有的引用进行限定，在类名前面添加相应的命名空间，如下所示：

```
System::Console::WriteLine(L"Hello World");
```

但如果使用 using 指令，就可以只指定类名来简化引用。在这里这种简化的作用并不明显，但当编写的程序要包含成百上千个 FCL 类引用时，将无须一再输入命名空间名。那么，到底什么是命名空间呢？

实际上，命名空间是一个非常简单的概念。在程序代码及在.NET 类库的代码中，名称将被用于大量的数据：数据类型、变量、方法等都需要指定名称。问题是如果指定的名称已经在库中使用，就会带来混乱，系统不知道应该使用哪一个。而命名空间就提供了解决这类问题的一种方法。通过将名称限定在一个命名空间中，所有对名称的引用都必须明确带上命名空间作为前缀。这样就允许在不同的命名空间内使用相同的名称，这些名称互不干扰。例如，在 System 命名空间内定义了 String 类，而如果在自己的一个命名空间也定义了 String 类，那么当指定引用 System::String 时表示使用的将是 System 命名空间内的类型。

接下来就是我们熟悉的 main()函数，这是程序的入口点。函数原型变为：

```
int main(array<System::String ^> ^args)
```

这里的 main()函数只需要一个参数 args 为.NET 框架内的 String 数组，用于接收用户输入的命令行参数。这里都是使用句柄，通过“^”表示，这样这些对象都将被创建在托管堆上，并使用引用计数，由 CLR 负责在引用计数变为 0 时自动进行垃圾回收。下面的语句

```
Console::WriteLine(L"Hello World");
```

用于向控制台输出 Hello World 语句，这里的 WriteLine()表示在输出语句之后自动换行，而字符串前面的 L 前缀表示使用的是宽字符的串，每个字符占用两字节。

正如前面所述，当该 HelloWorld 程序运行时，首先将由编译器发出一个 x86 占位程序将控制权交给 CLR。作为回应，CLR 定位并调用程序的入口点 main()方法，该方法已经被编译成 MSIL 代码存放在程序集内，然后 CLR 将调用 JIT 编译其将 MSIL 代码实时转换成本地机器代码并运行。通过系统提供的 ILDASM 工具（该工具默认位于系统盘的 Microsoft SDKs 目录下，如 D:\Program Files\Microsoft SDKs\Windows\v7.0A\bin\）可以查看该程序生成的 MSIL 代码以及 main()函数的 MSIL 代码，如图 17.9 及 17.10 所示。

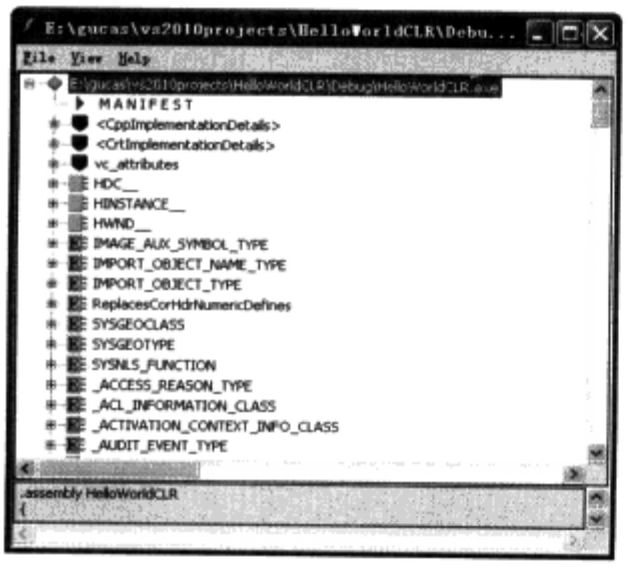


图 17.9　程序的 MSIL 代码

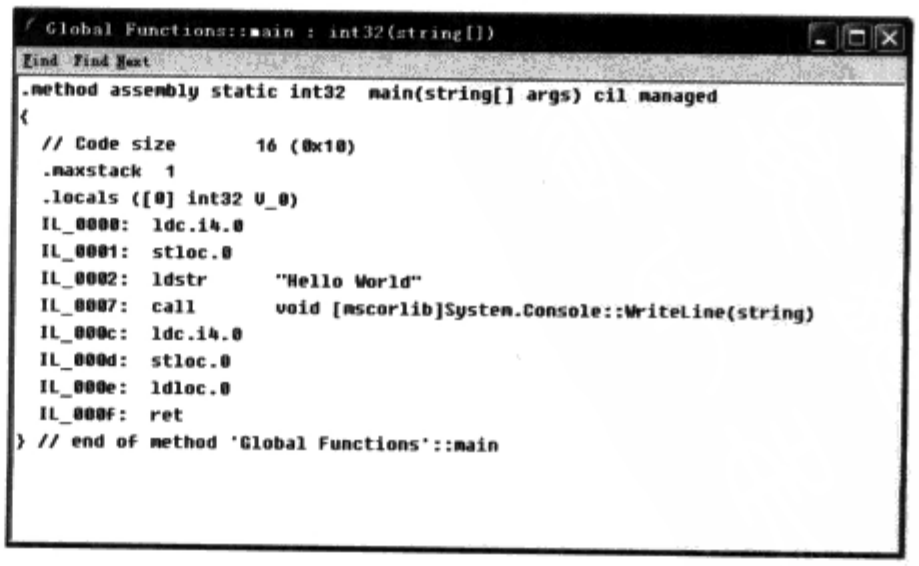


图 17.10　main 函数的 MSIL 代码

17.10 小结

本章介绍了.NET 框架涉及的一些新概念，概述了组成.NET 框架的各个组件，包括公共语言进行时（CLR）、公共语言规范（CLS）、公共类型系统（CTS）。接着介绍了.NET 框架下程序集的概念，主要讨论了程序集内部包含的主要内容，有元数据、程序集版本、微软中间语言和资源。然后介绍了.NET 开发应用程序的范畴，从这里可以了解到 CLR 可以做些什么，又有哪些做不了。然后对.NET 框架中的类库进行介绍，简要描述了类库中各个命名空间的主要功能，方便读者查找所需的类库。最后介绍了经典的 Hello World 程序在.NET 框架下的实现，使读者对.NET 应用程序有一个初步的了解。

第 18 章　Visual C++ 2010 CLR Windows 窗口编程

Windows 窗体已经对 Windows 开发产生了影响。当应用程序处于初始设计阶段时，是建立基于 Web 的应用程序还是建立客户应用程序已经很难抉择了。Windows 客户应用程序开发起来非常高速和高效，它们可以为用户提供丰富的体验。

Visual Basic 开发人员对 Windows 窗体应该比较熟悉。创建新窗体（也称为窗口或对话框）也采用把控件从工具箱拖放到窗体设计器上的方式。但是，如果在创建消息泵和监视消息时使用的是 C 样式的传统 Windows 编程，或者您是一位 MFC 程序员，就会发现如今可以获得需要的低级内部功能了。

本章将主要介绍 Windows 窗体的以下几个方面。

- Form 类；
- Windows 窗体的类层次结构；
- System::Windows::Forms 命名空间中的控件和组件；
- 菜单和工具栏；
- 创建控件；
- 创建用户控件。

18.1　创建 Windows 应用程序

首先需要创建一个 Windows 窗体应用程序。下面的示例创建了一个空白窗体，并把它显示在屏幕上。该示例没有使用 Visual Studio 2010，而是在文本编辑器中输入代码，使用命令行编译器进行编译。例 18.1 是代码清单。

例 18.1

```
#pragma once
/*
*这个程序用于演示如何手工输入一个创建空白 Form 的程序
*在编译之前，先设置 Project-->Properties(Alt+F7)
*在 Framework and References 中单击"Add New Reference..."按钮
*手工添加两个引用 System 和 System::Windows::Forms
*/
using namespace System;
using namespace System::Windows::Forms;

namespace NotepadForms
{
        public ref class MyForm : public Form
    {
        public:
```

```
            MyForm()
            {
            }

        };
}

[STAThreadAttribute]
int main(array<System::String ^> ^args)
{
        // 这个语句创建一个 Form 窗体，不过是作为控制台程序的子窗体，
        //一旦关闭主程序，这个 Form 窗体就会被关闭
        Application::Run(gcnew NotepadForms::MyForm());
        return 0;
}
```

在编译和运行该示例时，会得到一个没有标题的小空白窗体。该窗体没有什么实际功能，但它却是一个 Windows 窗体。

代码中有两个地方需要注意。第一个是使用继承功能来创建 MyForm 类。下面的代码声明 MyForm 派生于 System::Windows::Forms。

```
public ref class MyForm : public Form
```

Form 类是 System::Windows::Forms 命名空间的一个主要类。代码的其他部分如下：

```
[STAThreadAttribute]
int main(array<System::String ^> ^args)
{
        // 这个语句创建一个 Form 窗体，不过是作为控制台程序的子窗体，
        //一旦关闭主程序，这个 Form 窗体就会被关闭
        Application::Run(gcnew NotepadForms::MyForm());
        return 0;
}
```

main()是 C++客户应用程序的默认入口，负责完成需要的启动处理。在本例中，在项目属性对话框中设置启动的类名。注意属性[STAThreadAttribute]，它把 COM 线程模型设置为单线程单元（Single-Threaded Apartment，STA）。COM 交互操作需要 STA 线程模型，默认为添加到 Windows 窗体项目中。

Application::Run()方法负责启动标准的应用程序消息循环。它有 3 个重载版本：第一个重载版本不带参数，第二个重载版本把 ApplicationContext 对象作为参数，在本例中第三个重载版本把窗体对象作为参数。在本示例中，MyForm 对象是应用程序的主窗体，这表示在关闭该窗体时，应用程序就结束了。使用 ApplicationContext 类，可以对主消息循环何时结束和应用程序何时退出有更多的控制权。

Application 类包含一些非常有用的功能。它提供了一些静态方法和属性，用于控制应用程序的启动和停止过程，访问由应用程序处理的 Windows 消息。表 18.1 列出了其中的一些比较有用的方法和属性。

表 18.1　　常用的方法/属性

方法/属性	说　明
CommonAppDataPath	对应用程序的所有用户都通用的数据路径。一般是 BasePath\CompanyName\ProductName\Version，其中 BasePath 是 C:\Documents and Settings\username\ApplicationData。如果该路径不存在，就创建一个
ExecutablePath	启动应用程序的可执行文件的路径和文件名

续表

方法/属性	说　明
LocalUserAppDataPath	类似于 CommonAppDataPath，但该属性支持漫游
MessageLoop	如果当前线程上存在消息循环，就返回 True，否则返回 False
StartupPath	类似于 ExecutablePath，但不返回文件名
AddMessageFilter	用于处理前的消息，在基于 IMessagerFilter 的对象上执行，消息可以从消息循环中过滤出来，或者在消息发送到循环中之前进行特殊的处理
DoEvents	类似于 Visual Basic 的 DoEvents 语句，允许处理队列中的消息
EnableVisualStyles	允许对应用程序的各种可视化元素使用 XP 可视化形式。它有两个重载版本，接收各种信息。一个重载版本的参数是信息流，另一个重载版本的参数是信息所有的完整名称和路径
Exit 和 ExitThread	Exit 结束所有当前运行的消息循环，并退出应用程序。ExitThread 只结束消息循环，关闭当前线程上的所有窗口

在 Visual Studio 2010 中生成本示例时，创建窗体的代码在 Form1.cpp 文件中，这个文件是：

```
/*
*这个程序主要是简单地演示 Visual Studio 2010 自动生成的代码
*这里我们再主窗体 Form1 中加入了一个 Button 和 TextBox 控件
*在 InitializeComponent()函数内可以看到自动生成的代码
*/

namespace VisualStudioForm {

        using namespace System;
        using namespace System::ComponentModel;
        using namespace System::Collections;
        using namespace System::Windows::Forms;
        using namespace System::Data;
        using namespace System::Drawing;

        /// <summary>
        /// Summary for Form1
        ///
        /// WARNING: If you change the name of this class, you will need to change the
        ///          'Resource File Name' property for the managed resource compiler tool
        ///          associated with all .resx files this class depends on.  Otherwise,
        ///          the designers will not be able to interact properly with localized
        ///          resources associated with this form.
        /// </summary>
        public ref class Form1 : public System::Windows::Forms::Form
        {
        public:
                Form1(void)
                {
                        InitializeComponent();
                        //
                        //TODO: Add the constructor code here
                        //
                }
        protected:
```

```
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        ~Form1()
        {
            if (components)
            {
                delete components;
            }
        }
    private: System::Windows::Forms::Button^  button1;
    protected:
    private: System::Windows::Forms::TextBox^  textBox1;

    private:
        /// <summary>
        /// Required designer variable.
        /// </summary>
        System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        void InitializeComponent(void)
        {
            this->button1 = (gcnew System::Windows::Forms::Button());
            this->textBox1 = (gcnew System::Windows::Forms::TextBox());
            this->SuspendLayout();
            //
            // button1
            //
            this->button1->Location = System::Drawing::Point(100, 48);
            this->button1->Name = L"button1";
            this->button1->Size = System::Drawing::Size(110, 26);
            this->button1->TabIndex = 0;
            this->button1->Text = L"button1";
            this->button1->UseVisualStyleBackColor = true;
            //
            // textBox1
            //
            this->textBox1->Location = System::Drawing::Point(70, 101);
            this->textBox1->Name = L"textBox1";
            this->textBox1->Size = System::Drawing::Size(176, 21);
            this->textBox1->TabIndex = 1;
            //
            // Form1
            //
```

```
            this->AutoScaleDimensions = System::Drawing::SizeF(6, 12);
            this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
            this->ClientSize = System::Drawing::Size(292, 266);
            this->Controls->Add(this->textBox1);
            this->Controls->Add(this->button1);
            this->Name = L"Form1";
            this->Text = L"Form1";
            this->ResumeLayout(false);
            this->PerformLayout();

        }
#pragma endregion
    };
}
```

首先，本示例应用程序的代码比简单的命令行示例长。类的开头有好多个 using 语句，在本例中大多数是不必要的，但保留它们并无大碍。类 Form1 派生于 System::Windows::Forms，Form 1. Designer 文件的第一行代码如下。

```
System::ComponentModel::Container ^components;
```

在本示例中，这一行代码并没有做什么工作。当给窗体添加组件时，也就把这行代码添加到了组件对象，该代码是一个容器。添加这个容器与窗体的释放有关。窗体类支持 IDisposable 接口，因为它是在 Component 类中执行的。在组件中被添加到组件窗口中时，容器将确保组件被正确地跟踪，并在释放窗体时释放它。查看代码中的析构方法，就可以看到它，如下所示：

```
protected:
~Form1()
{
    if (components)
    {
        delete components;
    }
}
```

可以看到，在调用 Dispose()方法时，也会调用组件对象的 Dispose 方法，因为组件对象包含其他组件，所以它们也会被释放。

Form1 类的构造函数在 Form1.cs 中，如下所示：

```
public:
Form1(void)
{
    InitializeComponent();
    //
    //TODO: Add the constructor code here
    //
}
```

这是对 InitializeComponent()方法的调用。InitializeComponent()在 Form1.Designer.cs 中，顾名思义，InitializeComponent()方法初始化了添加到窗体上的所有控件，还初始化了窗体的属性。在本示例中，InitializeComponent()的代码如下所示：

```
private : void InitializeComponent()
{
```

```
        this->component =gc new System::ComponentModel::Container();
        this->Text="Form1";
    }
```

这是基本的初始化代码。该方法与 Visual Studio 的设计器相关联。使用设计器修改窗体时，这些改动会在 InitializeComponent()中反映出来。如果在 InitializeComponent()中修改了任意类型的代码，下次在设计器中进行修改时，这些改动就会丢失。每次在设计器中进行修改后，InitializeComponent()都会重新生成。如果需要为窗体或窗体上的控件和组件添加其他初始化代码，就应在调用 InitializeComponent()后添加。InializeComponent()还负责实例化控件，这样在 InializeComponent()之前所有引用控件的调用都会失败，并生成一个空引用异常。

要在窗体上添加控件或组件，可以按【Ctrl+Alt+X】组合键或者在 Visual Studio 2010 的 View 菜单中选择 ToolBox 命令。此时，Form1 应处于设计模式。在 ToolBox 内，选择 Button 控件，将其拖放到设计器的窗体上，也可以双击该控件，将其添加到窗体上。对 TextBox 控件进行相同的操作。

在窗体上添加 TextBox 控件和 Button 控件后，InitializeComponent()就会扩展，包含如下代码：

```
void InitializeComponent(void)
{
        this->button1 = (gcnew System::Windows::Forms::Button());
        this->textBox1 = (gcnew System::Windows::Forms::TextBox());
        this->SuspendLayout();
        //
        // button1
        //
        this->button1->Location = System::Drawing::Point(100, 48);
        this->button1->Name = L"button1";
        this->button1->Size = System::Drawing::Size(110, 26);
        this->button1->TabIndex = 0;
        this->button1->Text = L"button1";
        this->button1->UseVisualStyleBackColor = true;
        //
        // textBox1
        //
        this->textBox1->Location = System::Drawing::Point(70, 101);
        this->textBox1->Name = L"textBox1";
        this->textBox1->Size = System::Drawing::Size(176, 21);
        this->textBox1->TabIndex = 1;
        //
        // Form1
        //
        this->AutoScaleDimensions = System::Drawing::SizeF(6, 12);
        this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
        this->ClientSize = System::Drawing::Size(292, 266);
        this->Controls->Add(this->textBox1);
        this->Controls->Add(this->button1);
        this->Name = L"Form1";
        this->Text = L"Form1";
        this->ResumeLayout(false);
        this->PerformLayout();

}
```

查看该方法中的前 3 行代码，会看到 TextBox 控件和 Button 控件被实例化了。注意给控件指定的名称 textBox1 和 button1。默认情况下，设计器会使用控件的名称，并在该名称的最后添加一个整数值。在添加另一个按钮时，设计器会使用名称 button2，依次类推。下一行代码是 SuspendLayout()和 ResumeLayout()对的部分。SuspendLayout()临时挂起控件第一次初始化时发生的布局事件。在该方法的最后，将调用 ResumeLayout()方法，把事件重置为正常状态。在包含许多控件的复杂窗体上，InitializeComponent()方法会非常长。

要修改控件的复杂值，可以按【F4】键，或从 View 菜单中选择 Properties Window 命令。该窗口允许修改控件或组件的大多数属性。在 Properties 窗口中进行修改后，InitializeComponent()方法就会重新编写，以反映新属性值。例如，如果在 Properties 窗口中将 Text 属性改为 My Button，Initialize Component()就会包含如下代码：

```
//
// button1
//
this->button1->Location = System::Drawing::Point(100, 48);
this->button1->Name = L"button1";
this->button1->Size = System::Drawing::Size(110, 26);
this->button1->TabIndex = 0;
this->button1->Text = L"My Button";
this->button1->UseVisualStyleBackColor = true;
```

如果使用的不是 Visual Studio .NET 编辑器，就需要在设计中包含 InitializeComponent()类型函数。把所有这些初始化代码放在同一个位置，有助于使构造函数更简洁明了。如果有多个构造函数，还需要确保能从每个构造函数中调用初始化代码。

18.2　类层次结构

在设计和构建定制控件时，理解层次结构是非常重要的。如果定制控件派生于已有的控件，如文本框，并添加了属性和方法，就应使定制控件派生于文本框控件，再重写、添加属性和方法，以满足要求。但是，如果创建的控件与.NET Framework 中已有的控件不匹配，就必须从 3 个基类中派生：如果需要自动滚动功能，就从 Control 或 ScrollableControl 中派生；如果控件应是其他控件的容器，就应从 ContainerControl 类中派生。图 18 1 显示了 System::Windows::Forms 命名空间中类的关系。

下面将介绍窗体程序中的类，以及它们如何协同工作，如何使用它们建立具有专业化外观的客户应用程序。

18.3　Control 类

System::Windows::Forms 命名空间中有一个特殊的类，它是每个控件和窗体的基类，这个类就是 System::Windows::Forms::Control。Control 类执行核心功能，创建用户所见的界面。Control 类派生于 System::ComponentModel::Component 类。Component 类为 Control 类提供了必要的基础结构，在把控件拖放到设计界面上以及包含在另一个对象中时需要它。Control 类为派生于它的类提供了一个很长的功能列表。这个列表太长，不能在这里全部列出，所以仅介绍 Control 类提供的比较重要的功能。本章的后面在介绍基于 Control 类的特定控件时，将在一些示例代码中论述属性和方法。下面几小节将按照功能组合方法和属性，把相关的功能放在一起进行讨论。

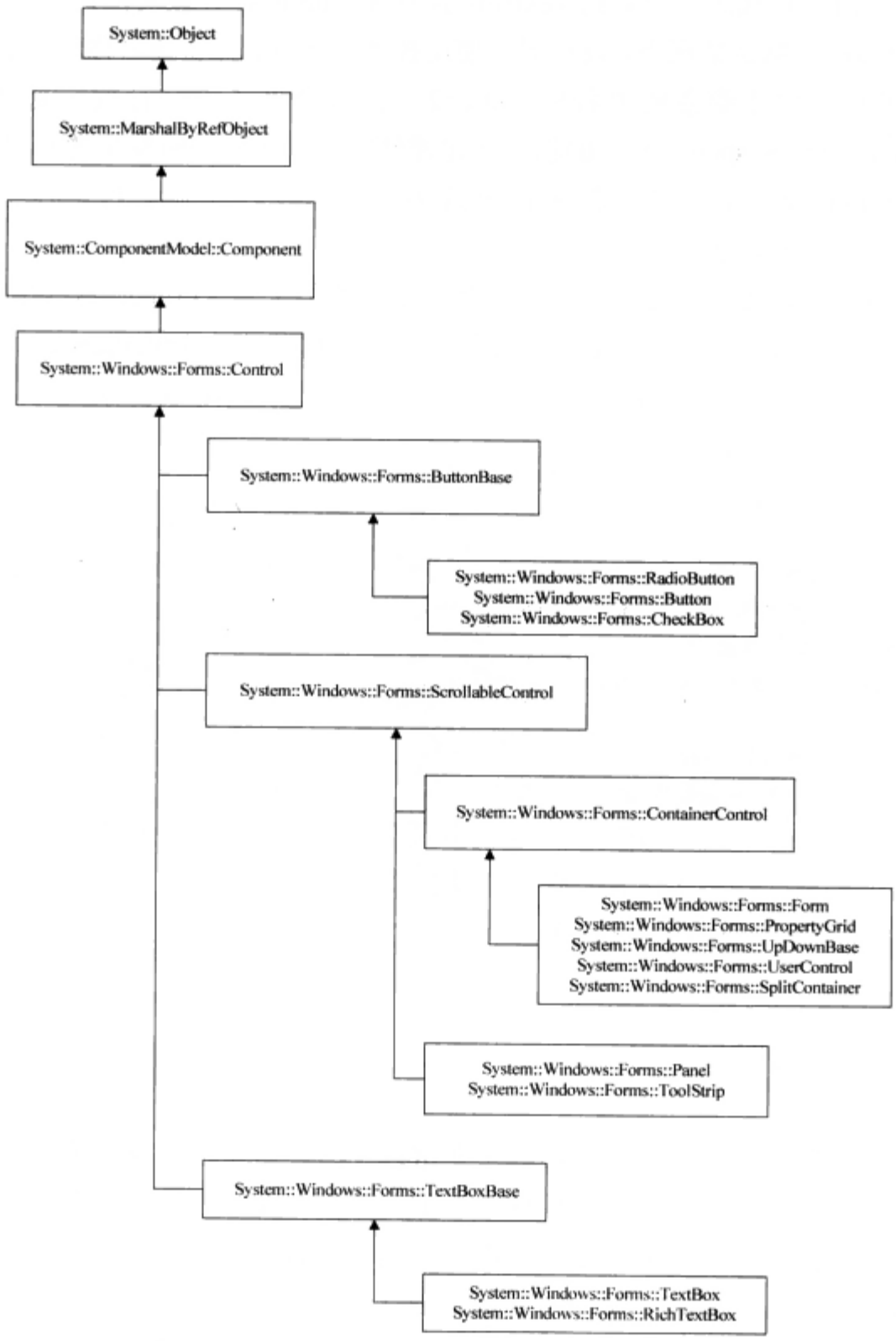


图 18.1　System::Windows::Forms 命名空间中类的关系

18.3.1　大小与位置

控件的大小和位置由属性 Height、Width、Top、Bottom、Left、Right 以及辅助属性 Size 和 Location 确定。区别是 Height、Width、Top、Bottom、Left、Right 属性值都是一个整数，而 Size 的值使用一个 Size 结构来表示，Location 的值使用一个 Point 结构来表示。Size 结构和 Point 结构都包含 X/Y 坐标。Point 结构一般相对于一个位置，而 Size 结构是对象的高和宽。Size 和 Point 都位于 System::Drawing 命名空间。它们非常类似，因为它们都提供了 X/Y 坐标对，还拥有用于简单地比较和转换的重写运算符。例如，可以对两个 Size 结构执行相加操作。对于 Point 结构，加法运算符已进行了重写，可以把 Size 结构加到 Point 结构上，得到一个新的 Point。其结果是给某个位置加上某个距离值，得到一个新位置。如果动态创建窗体或控件，这是非常方便的。

Bounds 属性返回一个 Rectangle 对象，它表示一个控件区域。该区域包含滚动条和标题栏。Rectangle

也位于 System::Drawing 命名空间。ClientSize 属性是一个 Size 结构，表示控件的客户区域，不包含滚动条和标题栏。

PointToClient()和 PointToScreen()方法是方便的转换方法，它们的参数是 Point 结构，返回一个 Point 结构。PointToClient()的 Point 参数表示屏幕坐标，该方法把屏幕坐标转换为基于当前客户对象的坐标。这非常便于进行拖放操作。PointToScreen()正好与之相反，它提取客户对象的坐标，把它们转换为屏幕坐标。还有 RectangleToScreen()和 ScreenToRectangle()方法，它们具有相同的功能，只是用 Rectangle 结构代替 Point 结构。

Dock 属性确定子控件停放在父控件的哪条边上。DockStyle 枚举值用做其属性值。该值可以是 Top、Bottom、Right、Left、Fill 和 None。Fill 值会使控件的大小正好匹配父控件的客户区域。

Anchor 属性把子控件的一条边与父控件的一条边对齐。这与停靠不同，因为它不设置父控件的一条边，而是把到该边界的当前距离设置为常量。例如，如果把子控件的右边界与父控件的右边界对齐，并重新设置父控件的大小，子控件右边界到父控件右边界的距离将保持不变。Anchor 属性采用 AnchorStyles 枚举的值，其值可以是 Top、Bottom、Right、Left 和 None。通过设置该属性值，可以在重新设置父控件的大小时，动态地设置子控件的大小。这样，当用户重新设置窗体的大小时，按钮和文本框就不会被剪切或隐藏。

Dock 和 Anchor 属性与 Flow 和 Table 布局控件(详见本章后面的内容)一起使用时，可以创建非常复杂的用户窗口。对于包含许多控件的复杂窗体来说，窗体大小的重新设置比较困难，这些工具有助于完成这个任务。

18.3.2　外观

与控件外观相关的属性有 BackColor 和 ForeColor，它们把 System::Drawing.Color 对象作为其值。BackGroundImage 属性把基于 Image 的对象作为其值。System::Drawing.Image 是一个抽象类，用做 Bitmap 和 Metafile 类的基类。BackColorImageLayout 属性使用 ImageLayout 枚举设置图像在控件上的显示方式，其有效值是 Center、Tile、Stretch、Zoom 和 None。

Font 和 Text 属性处理文字的显示。要修改 Font 属性，需要创建一个 Font 对象。在创建 Font 对象时，要指定字体名称、字号和样式。

18.3.3　用户交互操作

用户交互操作最好描述为控件创建和响应的各种事件。一些比较常见的事件有 Click、DoubleClick、KeyDown、KeyPress、Validating 和 Paint。

鼠标事件 Click、DoubleClick、MouseDown、MouseUp、MouseEnter、MouseLeave 和 MouseHover 处理鼠标和控件的交互操作。如果处理 Click 和 DoubleClick 事件，每次捕获一个 DoubleClick 事件时，也会引发 Click 事件。如果处理不正确，就会出现不希望的结果。Click 和 DoubleClick 事件都把 EventArgs 作为其参数，而 MouseDown 和 MouseUp 事件把 MouseEventArgs 作为其参数。MouseEventArgs 包含几个有用的信息，如单击的按钮、按钮被单击的次数、鼠标轮制动器(鼠标轮上的凹槽)的数目和鼠标当前的 X/Y 坐标。如果可以访问这些信息，就必须处理 MouseDown 或 MouseUp 事件，而不是 Click 或 DoubleClick 事件。

键盘事件的工作方式与此类似：需要一些信息来确定处理什么事件。对于简单的情况，KeyPress

事件接收一个 KeyPressEventArgs，它包含表示被按键的字符值 KeyChar。Handled 属性用于确定事件是否已被处理。把 Handled 属性设置为 True，事件就不会由操作系统进行默认处理。如果需要被按的键的更多信息，则处理 KeyDown 或 KeyUp 事件比较合适，它们都接收 KeyEventArgs。KeyEventArgs 中的属性包括【Ctrl】、【Alt】或【Shift】键是否被按下。KeyCode 属性返回一个 Keys 枚举值，表示被按下的键。与 KeyPressEventArgs.KeyChar 不同，KeyCode 属性指定键盘上的每个键，而不仅仅是字母数字键。KeyData 属性返回一个 Key 值，还设置修饰符。修饰符与值进行 OR 运算，指定是否同时按下了【Shift】或【Ctrl】键。KeyValue 属性是 Keys 枚举的整数值。Modifiers 属性包含一个 Keys 值，它表示被按下的修饰符键。如果选择了多个修饰符键，这些值就进行 OR 运算。键盘事件以下述顺序来触发：

（1）KeyDown；

（2）KeyPress；

（3）KeyUp。

Validating、Validated、Enter、Leave、GotFocus 和 LostFocus 事件都处理获得焦点(或被激活)和失去焦点的控件。在用户用【Tab】键选择一个控件或用鼠标选择该控件时，该控件就获得了焦点。Enter、Leave、GotFocus 和 LostFocus 事件的功能非常类似。GotFocus 和 LostFocus 事件是低级事件，与 Windows 消息 WM_SETFOCUS 和 WM_KILLFOCUS 相关。一般应尽可能使用 Enter 和 Leave 事件。Validating 和 Validated 事件在验证控件时发生。这些事件接收一个 CancelEventArgs，利用该参数，把 Cancel 属性设置为 True，就可以取消以后的事件。如果定制了验证代码，而且验证失败，就可以把 Cancel 属性设置为 True，且控件也不会失去焦点。Validating 事件在验证过程中发生，Validated 事件在验证过程后发生。这些事件的触发顺序如下：

（1）Enter；

（2）GotFocus；

（3）Leave；

（4）Validating；

（5）Validated；

（6）LostFocus。

理解这些事件的触发顺序是很重要的，可以避免不小心创建递归事件。例如，在控件的 LostFocus 事件中设置控件的焦点，就会创建一个消息死锁，且应用程序会停止响应。

18.3.4 Windows 功能

System::Windows::Forms 命名空间是依赖 Windows 功能的少数几个命名空间之一。Control 类就是一个很好的示例。如果对 System::Windows::Forms.dll 进行反编译，就会看到 UnsafeNativeMethods 类的引用列表。.NET Framework 使用该类封装所有的标准 Win32 API 调用。使用与 Win32 API 的交互操作，标准 Windows 应用程序的外观和操作方式就可以通过 System::Windows:: Forms 命名空间获得。

支持与 Windows 交互操作的功能包括 Handle 和 IsHandleCreated 属性。Handle 属性返回一个包含控件 HWND(Windows 句柄)的 IntPtr。窗口句柄是惟一标识窗口的 HWND。可以将控件看做是一个窗口，所以它有相应的 HWND。可以使用 Handle 属性进行任意数量的 Win32 API 调用。

为了访问 Windows 消息，可以重写 WndProc 方法。该方法把一个 Message 对象作为其参数。Message 对象是 Windows 消息的一个简单封装器。它包含 HWnd、LParam、WParam、Msg 和 Result 属性。如

果希望由系统处理消息，就必须确保把消息传送给 base.WndProc(msg)方法。如果希望自己处理消息，就不需要传送消息。

18.4　标准 Windows 控件使用指南

18.4.1　Button 控件

按钮控件在 MFC 中用 CButton 表示。CButton 包含了 3 种样式的按钮：Push Button、Check Box 和 Radio Box。所以在利用 CButton 对象生成按钮窗口时需要指明按钮的风格。

创建按钮：

```
BOOL CButton::Create( LPCTSTR lpszCaption, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID );
```

其中 lpszCaption 是按钮上显示的文字，dwStyle 为按钮风格，除了 Windows 风格可以使用外（如 WS_CHILD|WS_VISUBLE|WS_BORDER），还有按钮专用的一些风格，如表 18.2 所示。

表 18.2　　按钮风格

风　格	说　明
BS_AUTOCHECKBOX	检查框，按钮的状态会自动改变
BS_AUTORADIOBUTTON	圆形选择按钮，按钮的状态会自动改变
BS_AUTO3STATE	允许按钮有 3 种状态，即选中、未选中、未定
BS_CHECKBOX	检查框
BS_DEFPUSHBUTTON	默认普通按钮
BS_LEFTTEXT	左对齐文字
BS_OWNERDRAW	自绘按钮
BS_PUSHBUTTON	普通按钮
BS_RADIOBUTTON	圆形选择按钮
BS_3STATE	允许按钮有 3 种状态，即选中、未选中、未定

rect 参数为窗口所占据的矩形区域，pParentWnd 参数为父窗口指针，nID 参数为该窗口的 ID 值。

获取/改变按钮状态：检查按钮和圆形按钮可能有两种状态，即选中和未选中，如果设置了 BS_3STATE 或 BS_AUTO3STATE 风格，就可能出现第三种状态——未定，这时按钮显示灰色。通过调用 int CButton::GetCheck();可得知当前是否被选中，返回 0 表示未选中，1 表示选中，2 表示未定。调用 void CButton::SetCheck(int nCheck);可设置当前选中状态。

处理按钮消息：要处理按钮消息，需要在父窗口中进行消息映射，映射宏为 ON_BN_CLICKED(id, memberFxn)。id 为按钮的 ID 值，就是创建时指定的 nID 值。处理函数原型为 afx_msg void memberFxn( );。

18.4.2　CheckBox 控件

CheckBox（复选框）控件也派生于 ButtonBase 类，用于接收来自用户的二状态或三状态响应。如果把 ThreeState 属性设置为 True，复选框的 CheckState 属性就可以是以下 3 个 CheckState 枚举值之一。

- Checked：复选框有一个选中标记。
- Unchecked：复选框没有选中标记。
- Indeterminate：复选框为灰色显示。

Indeterminate 值只能在代码中设置，不能由用户设置。如果需要告诉用户选项还未设置，就可以

使用这个值。如果希望使用布尔值，还可以使用 Checked 属性。

CheckedChanged 和 CheckStateChanged 事件在 CheckState 或 Checked 属性改变时发生。捕获的这些事件可以根据复选框的新状态设置其他值。在 frmControls 窗体类中，几个复选框的 CheckedChanged 事件由下面的方法处理：

```
private:void checkBoxChanged(object^ sender, EventArgs^ e)
{
    CheckBox^ checkbox=(CheckBox^)sender;
    MessageBox::Show(checkbox->Name+L"new value is"+checkbox->Checked.ToString());
```

在每个复选框的 Checked 属性改变时，都会显示一个消息框，其中包含了改变的复选框名称和新值。

18.4.3 RadioButton 控件

最后一个派生于 ButtonBase 类的控件是 RadioButton（单选按钮）。单选按钮一般用做一个钮，有时称为选项按钮。单选按钮允许用户从几个选项中选择一个，当同一个窗口中有多个 RadioButton 控件时，一次只能选择一个按钮。所以如果有 3 个选项，如 Red、Green 和 Blue，如果 Red 选项被选中，而用户单击 Blue，则 Red 会自动取消选择。

Appearance 属性使用 Appearance 枚举值，即 Button 或 Normal。当选择 Normal 时，单选按钮看起来像一个小圆圈，其旁边有一个标签。选择按钮会填充圆圈，选择另一个按钮会取消选择当前选中按钮，使圆圈为空。当选择 Button 时，RadioButton 控件看起来像一个标准按钮，但工作方式类似于开关，选中是指焦点在位置中，取消选择是指正常状态或焦点在位置外。

CheckedAlign 属性确定圆圈与标签文本的相对位置，它可以在标签的顶部、左右两侧或下方。

只要 Checked 属性的值改变，就会引发 CheckedChanged 事件。这样就可以根据控件的新值执行其他动作。

18.4.4 ComboBox 控件、ListBox 控件和 CheckedListBox 控件

ComboBox、ListBox 和 CheckedListBox 都派生于 ListControl 类，该类提供了一些基本的列表管理功能。使用列表控件最重要的事情是给列表添加数据和选择数据。使用哪个列表一般取决于列表的用法和列表中数据的类型。如果需要选择多个选项，或用户需要在随时查看列表中的几项，最好使用 ListBox 和 CheckedListBox。如果一次只选择一个选项，就可以使用 ComboBox。

在使用列表框之前，必须先添加数据。为此，应给 ListBox.ObjectCollection 添加对象。该集合可以使用列表的 Items 属性访问。由于该集合存储了对象，因此可以把任意有效的.NET 类型添加到列表中。要标识对象，需要设置两个重要的属性。第一个是 DisplayMember 属性，告诉列表控件，在列表中显示对象的哪个属性。另一个是 ValueMember 属性，它是要返回值的对象属性。如果在列表中添加了字符串，这两个属性就默认使用字符串值。例 18.2 所示的应用程序中的 frmLists 窗体显示了如何把对象和字符串（也是对象）加载到列表框中。该示例使用 Vendor 对象作为列表数据。Vendor 对象只包含两个属性——Name 和 PhoneNo，DisplayMember 属性设置为 Name，这告诉列表控件，把列表中的 Name 属性值显示给用户。

访问列表控件中的数据有两种方式，如例 18.2 所示。列表中加载了 Vendor 对象，设置了 DisplayMember 和 ValueMember 属性。这段代码在示例应用程序的 frmLists 窗体中。

首先是 LoadList 方法，它给列表加载了 Vendor 对象或一个包含供应商姓名的简单字符串。选中一个单选按钮，看看在列表中加载了哪些值。

例 18.2

```
private: void LoadList(Control^ ctrlToLoad)
{
    ListBox^ tmpCtrl=(ListBox^)ctrlToLoad;
    tmpCtrl->Items->Clear();
    tmpCtrl->DataSource=nullptr;

    if(radioButton1->Checked)
    {
        //加载对象
        tmpCtrl->Items->Add(gcnew Vendor(L"XYZ Company",
                        L"555-555-1234"));
        tmpCtrl->Items->Add(gcnew Vendor(L"ABC Company",
                        L"555-555-2345"));
        tmpCtrl->Items->Add(gcnew Vendor(L"Other Company",
                        L"555-555-3456"));
        tmpCtrl->Items->Add(gcnew Vendor(L"Other Company",
                        L"555-555-4567"));
            tmpCtrl->Items->Add(gcnew Vendor(L"Other Company",
                        L"555-555-6789"));
tmpCtrl->Items->Add(gcnew Vendor(L"Other Company",
            L"555-555-7890"));
        tmpCtrl->DisplayMember=L"Name";
        }

    else
    {
        tmpCtrl->Items->Clear();
        tmpCtrl->Items->Add (L"XYZ Company");
        tmpCtrl->Items->Add (L"ABC Company");
        tmpCtrl->Items->Add (L"Other Company");
        tmpCtrl->Items->Add (L"Another Company");
        tmpCtrl->Items->Add (L"More Company");
        tmpCtrl->Items->Add (L"Last Company");
    }
}
```

在列表中加载数据后，就可以使用 SelectedItem 和 SelectedIndex 属性获取数据了。SelectedItem 返回当前选中的对象。如果列表设置为允许选择多个选项，就不能保证返回选中的选项。此时，应使用 SelectObject 集合，它包含列表中当前选中的所有选项。

如果需要特定索引的选项，可以使用 Items 属性访问 ListBox.ObjectCollection。这是一个标准的.NET 集合类，该集合中的项可以用与其他集合类相同的方式访问。

如果使用 DataBinding 填充列表，SelectedValue 属性就会返回选中对象内设置为 ValueMember 属性的属性值。如果 Phone 设置为 ValueMember，SelectedValue 就从选中的项中返回 Phone 值。要使用 ValueMember 和 SelectValue，列表必须通过 DataSource 属性来加载。必须先使用对象加载 ArrayList 或其他基于 IList 的集合，再给列表赋予 DataSource 属性。下面的例 18.3 演示了该方法。

例 18.3

```
listBox1->DataSource=nullptr;
System::Collection::ArrayList^ lst = gcnew System::Collection::ArrayList();
lst->Add(gcnew Vendor(L"XYZ Company", L"555-555-1234"));
lst->Add(gcnew Vendor(L"XYZ Company", L"555-555-2345"));
lst->Add(gcnew Vendor(L"XYZ Company", L"555-555-3456"));
lst->Add(gcnew Vendor(L"XYZ Company", L"555-555-4567"));
lst->Add(gcnew Vendor(L"XYZ Company", L"555-555-6789"));
lst->Add(gcnew Vendor(L"XYZ Company", L"555-555-7890"));
listBox1->Items->Clear();
listBox1->DataSource=lst;
listBox1->DisplayMember=L"Name";
listBox1->ValueMember=L"Phone";
```

使用 SelectedValue，但未使用 DataBinding，会导致 NullException 异常。下面的代码演示了访问列表中数据的语法：

```
//将选中的 Verdor 对象赋值给 obj
obj = listBox1->SelectedItem;

//obj is set to the Vendor object with index of 3(4th object)
//将 obj 赋值成数组中下标为 3 的 Vendor 对象(第 4 个对象)
obj=listBox->Items[3];

listBox1->ValuesMember=L"Phone";
obj=listBox1->SelectValue;
```

这些方法都把对象返回为类型。要使用对象的值，必须把它转换为正确的数据类型。

ComboBox 的 Items 属性返回 ComboBox.ObjectCollection。ComboBox 组合了编辑控件和列表框。把一个 DropDownStyle 枚举值传送给 DropDownStyle 属性，就可以设置 ComboBox 的样式。表 18.3 列出了 DropDownStyle 的各个值。

表 18.3　　DropDownStyle 的各个值

值	说　明
DropDown	组合框的文本部分是可以编辑的，用户可以输入值。用户必须单击箭头按钮，才能显示下拉列表
DropDownList	文本部分不能编辑，用户必须从列表中选择
Simple	类似于 DropDown，但列表总是可见的

如果列表中的值比较宽，就可以使用 DropDownWidth 属性改变控件下拉列表的宽度。MaxDrop DownItems 属性设置在显示下拉列表时的最大项目数。

FindString()和 FindStringExact()方法是列表控件的另外两个有用的方法。FindString()在列表中查找以传入字符串开头的第一个字符串。FindStringExact()查找与传入字符串匹配的第一个字符串。它们都返回找到的值的索引，如果没有找到，就返回-1。它们还可以将要搜索的起始索引整数作为参数。

18.4.5　DateTimePicker 控件

DateTimePicker 控件允许用户在许多不同格式中选择一个日期或时间值（或两者），可以以任何

标准时间/日期格式显示 DateTime 的值。Format 属性使用 DateTimePickerFormat 枚举，它可以把格式设置为 Long、Short、Time 或 Custom。如果 Format 属性设置为 DateTimePickerFormat.Custom，就可以把 CustomFormat 属性设置为表示格式的字符串。

DateTimePicker 还包含 Text 属性和 Value 属性。Text 属性返回 DateTime 的文本表示，Value 属性返回 DateTime 对象。还可以用 MinDate 和 Maxdate 属性设置日期所允许的最大值和最小值。

在用户单击向下箭头时，会显示一个日历，允许用户选择日历中的一个日期。DateTimePicker 还包含一些属性，可以设置标题、月份背景色和前景色，改变日期的外观。

ShowUpDown 属性确定控件上是否显示 UpDown 箭头，单击向上或向下箭头就可以改变向前突出显示的值。

18.4.6　ErrorProvider 组件

ErrorProvider 实际上并不是一个控件，而是一个组件。当把该组件拖放到设计器上时，它会显示在设计器下方的组件栏中。当存在一个错误条件或验证失败时，ErrorProvider 可以在控件的旁边显示一个图标。假定有一个 TextBox 控件用于输入年龄，业务规则是年龄不能大于 65，如果用户试图输入大于 65 的年龄，就必须通知用户该年龄大于所允许的值，需要改变输入的值。有效值的检查在文本框的 Validated 事件中进行。如果验证失败，就调用 SetError()方法，传送引起错误的控件和一个字符串，将该错误告知用户。然后，一个图标开始闪烁，表示出现了一个错误，将鼠标指针放在该图标上时，会显示错误文本。

可以为窗体上产生错误的每个控件创建一个 ErrorProvider，但如果窗体上有很多控件，就会很麻烦。另一种选择是使用一个 ErrorProvider，在验证事件中调用被验证控件的 IconLocation 方法和一个 ErrorIconAlignment 枚举值。该值设置图标与哪个控件对齐。如果没有错误，就调用 SetError()方法，并把空字符串作为错误字符串，清除 ErrorProvider。下面的例 18.4 说明了其工作原理。

例 18.4

```
private: System::Void textAge_Validating(System::Object^   sender,
                System::ComponentModel::CancelEventArgs^   e) {
if( textAge->Text->Length > 0 && Convert::ToInt32(textAge->Text) > 65 ){
        errMain->SetIconAlignment((Control^)sender,ErrorIconAlignment::MiddleRight);
        errMain->SetError((Control^)sender,L"Value must be less than 65.");
        e->Cancel = true;
}
else{
        errMain->SetError((Control^)sender,L"");
}
}

private: System::Void textZipCode_Validating(System::Object^   sender,
                System::ComponentModel::CancelEventArgs^   e) {
        if( textZipCode->Text->Length > 0 && textZipCode->Text->Length != 5 ){
                errMain->SetIconAlignment((Control^)sender,
                ErrorIconAlignment::MiddleRight);
                errMain->SetError((Control^)sender,L"Must be 5 characters.");
                e->Cancel = true;
        }
        else
```

```
        {
            errMain->SetError((Control^)sender,L"");
        }
    }
```

如果验证失败（如 txtAge 中的年龄超过 65），就调用 ErrorProvider 组件 errMain 的 SetIcon()方法。该方法把图标放在验证失败的控件旁边，接着设置错误。这样，在用户将鼠标指针放在该图标上时，就会显示一条消息，告知用户哪个控件导致了验证失败。

18.4.7 HelpProvider 组件

HelpProvider 类似于 ErrorProvider，也是一个组件，而不是控件。HelpProvider 允许挂起控件，显示帮助主题。要把控件与 HelpProvider 关联起来，应调用 SetShowHelp()方法，给该方法传送控件和一个确定是否显示帮助的布尔值。HelpNamespace 属性允许设置帮助文件。在设置 HelpNamespace 属性时，只要按【F1】键，就会显示帮助文件，用 HelpProvider 注册的控件还会获得焦点。可以用 SetHelpKeyword()方法为帮助文件设置一个关键字。SetHelpNavigator()带一个 HelpNavigator 枚举值，用于确定显示帮助文件中的哪个元素。可以把它设置为特定的主题、索引、目录表或搜索页面。SetHelpString()把与帮助相关的文本字符串值关联到控件上。如果没有设置 HelpNamespace 属性，按【F1】键就会在弹出窗口中显示该文本。下面在上一个示例中添加一个 HelpProvider：

```
helpProvider1.SetHelpString(txtAge, "Enter an age that is less than 65");
helpProvider1.SetHelpString(txtZipCode, "Enter a 5 digit zip code");
```

18.4.8 ImageList 组件

ImageList 组件就是一个图像列表。一般情况下，该组件用于存储一个图像集合，这些图像用做工具栏图标。许多控件都包含 ImageList 属性。该属性一般和 ImageIndex 属性一起使用。ImageList 属性设置为 ImageList 组件的一个实例，ImageIndex 属性设置为 ImageList 中应在控件中显示的图像的索引。使用 ImageIndex.Images 属性的 Add()方法可以把图像添加到 ImageList 组件中。Images 属性返回一个 ImageCollection。

两个最常用的属性是 ImageSize 和 ColorDepth。ImageSize 使用 Size 结构作为其值。其默认值是 16×16，但可以取 1~256 之间的任意值。ColorDepth 使用 ColorDepth 枚举作为其值。颜色深度值可以是 4～32 位。在.NET Framework 1.1 中，默认为 ColorDepth.Depth8Bit。

18.4.9 Label 控件

Label 控件一般用于给用户提供描述文本。文本可以与其他控件或当前系统状态相关。通常标签和文本框一起使用。标签为用户提供了在文本框中输入的数据类型的描述。标签控件总是只读的，用户不能修改 Text 属性的字符串值。但是，可以在代码中修改 Text 属性。UseMnemonic 属性允许用户启用访问键功能。在 Text 属性中，在一个字符前面加上宏符号&，标签控件中的该字母就会加上下划线。按【Alt】键和带有下划线的字母键就会把焦点移动到 Tab 顺序的下一个控件上。如果 Text 属性的文本包含一个宏符号，就应添加第二个宏符号，其后的字母将不带下划线。例如，如果标签文本是 Nuts & Bolts，就应该把属性设置为 Nuts && Bolts。由于标签控件是只读的，所以不能获得焦点。这就是焦点会移动到下一个控件上的原因。因此，要记住，如果启用 Mnemonic，就必须正确设置窗体上的 Tab 顺序。

AutoSize 属性是一个布尔值，指定标签是否根据标签的内容自动设置大小。在多语言应用程序中，Text 属性的长度会根据当前语言的不同而发生变化，此时就可以使用这个属性。

18.4.10　ListView 控件

ListView 控件允许以 4 种不同的方式显示条目。可以显示文本和可选的大图标，显示文本和可选的小图标，在垂直列表中显示文本和小图标，以及在详细视图中显示条目文本，并在列中显示文本和子条目。资源管理器的右边就用这种方式显示文件夹的内容。ListView 包含一个 ListViewItems 集合。ListViewItems 允许设置一个用于显示的 Text 属性，它的另一个属性 SubItems 包含在详细视图中显示的文本。

例 18.5 演示了如何使用 ListView。本示例包含国家的短列表。每个 CountryList 对象都包含国名、国家简称和货币属性。

例 18.5

```
using namespace System;

namespace FormSample
{
    public ref class CountryItem:public System::Windows::Forms::ListViewItem
    {
        String^ _cntryName=L"";
        String^ _cntryAbbrev=L"";

        public: CountryItem(String^ countryName,String^ countryAbbreviation, String^ currency)
        {
            _cntryName=countryName;
            _cntryAbbrev=countryAbbreviation;
            base->Text=_cntryName;
            base->SubItems->Add(currency);
        }
        public: property String^ CountryName
        {
            get { return _cntryName;}
        }
        public : property String^ CountryAbbreviation
        {
            get { return _cntryAbbrev;}
        }
    }
}
```

CountryList 类派生于 ListViewItem。这是因为只能给 ListView 控件添加基于 ListViewItem 的对象。在构造函数中，给 base->Text 属性传送国家名，给 base->SubItems 属性添加货币值。这会在列表中显示国家名，在 Details 视图的另一个单独的列中显示货币。

接着，需要在窗体的代码中给 ListView 控件添加几个 CountryItem 对象：

```
lvCountries->Items->Add(gcnew CountryItem(L"United States", L"US",
    L"Dollar"));
lvCountries->Items[0]->ImageIndex=0;
lvCountries->Items->Add(gcnew CountryItem(L"Great Britain", L"GB",
```

```
        L"Pound"));
lvCountries->Items[1]->ImageIndex=1;
lvCountries->Items->Add(gcnew CountryItem(L"Canada", L"CA", L"Dollar"));
lvCountries->Items[2]->ImageIndex=2;
lvCountries->Items->Add(gcnew CountryItem(L"Japan", L"JP", L"Yen"));
lvCountries->Items[3]->ImageIndex=3;
lvCountries->Items->Add(gcnew CountryItem(L"Germany", L"GM",
        L"Deutch Mark"));
lvCountries->Items[4]->ImageIndex=4;
```

这里给 ListView 控件 lvCountries 的 Items 集合添加了一个新的 CountryItem。注意在把 CountryItem 添加到控件中后，才设置 CountryItem 的 ImageIndex 属性。有两个 ImageIndex 属性，一个用于大图标，一个用于小图标（SmallImageList 和 LargeImageList 属性）。要给两个 ImageList 指定不同的图像大小，应确保以相同的顺序给 ImageList 添加条目。这样，每个 ImageList 的索引就表示尺寸不同的相同图像。在本例中，ImageList 包含添加的每个国家的标志图标。

窗体的顶部是一个组合框 cbView，它列出了 4 个不同的 View 枚举值。将这些条目添加到 cbView 中，如下所示：

```
cbView->Items->Add(View::LargeIcon);
cbView->Items->Add(View::SmallIcon);
cbView->Items->Add(View::List);
cbView->Items->Add(View::Details);
cbView->SelectedIndex=0;
```

在 cbView 的 SelectedIndexChanged 事件中，添加下面这行代码。

```
lvCountries->View=(View^)cbView->SelectedItem;
```

这行代码把 lvCountries 的 View 属性设置为在 ComboBox 控件中选中的新值。注意需要将其转换 View 类型，因为从 cbView 的 SelectedItem 属性中返回的是 object 类型。

最后，必须给 Column 集合添加列。这些列在 Details 视图中显示。在本例中添加了两列：Country Name 和 Currency。列的顺序是 ListViewItem 的 Text，然后是 ListViewItem->SubItem 集合中的每一项，按照它们在集合中的顺序显示。添加列时，需要创建一个 ColumnHeader 对象，并设置 Text 属性，还可以设置 Width 和 Alignment 属性。在创建 ColumnHeader 对象后，就可以把它添加到 Column 属性中。添加列的另一种方法是使用 Columns->Add()方法的重写版本，它允许传送 Text、Width 和 Alignment 值。下面是一个示例：

```
lvCountries->Column->Add(L"Country", 100, HorizontalAlignmentLeft);
lvCountries->Column->Add (L"Currency", 100, HorizontalAlignmentLeft);
```

如果把 AllowColumnReorder 属性设置为 True，用户就可以拖动列标题，重新安排列的顺序。

ListView 上的 CheckBoxes 属性在 ListView 的条目旁边显示复选框，允许用户在 ListView 控件中选择多个条目。使用 CheckItems 集合可以检查哪些项目被选中。

Alignment 属性设置 Large 和 Small 图标视图中图标的对齐方式。该值可以是 ListViewAlignment 枚举中的任意值，即 Default、Left、Top、SnapToGrid。Default 值允许用户把图标放在任意位置。在选择 Left 或 Top 时，条目应与 ListView 控件的左边或顶边对齐。在选择 SnapToGrid 时，条目会捕捉到 ListView 控件上不可见的栅格。AutoArrange 属性可以设置为布尔值，它会根据 Alignment 属性自动对齐图标。

18.4.11 PictureBox 控件

PicktureBox 控件用于显示图像。图像可以是 BMP、JPEG、GIF、PNG、元文件或图标。SizeMode

属性使用 PictureBoxSizeMode 枚举确定图像在控件中的大小和位置。SizeMode 属性可以是 AutoSize、CenterImage、Normal 和 StretchImage。

设置 ClientSize 属性，可以改变 PictureBox 的显示区域大小。要加载 PictureBox，首先创建一个基于 Image 的对象。例如，要把 JPEG 文件加载到 PictureBox 中，需要编写如下代码：

```
Bitmap^ myJpeg=gcnew Bitmap(L"mypic.jpg");
pictureBox1->Image=(Image^)myJpeg;
```

注意需要转换回 Image 类型，因为这是 Image 属性所要求的。

18.4.12　ProgressBar 控件

ProgressBar 控件是较长操作状态的可视化表示。它指示用户正在进行某个操作，用户应等待。ProgressBar 控件工作时要设置 Minimum 和 Maximum 属性。这些属性对应于进度指示器的最左端（Minimum）和最右端(Maximum)。设置 Step 属性，以确定每次调用 PerformStep()方法数值的增量。还可以使用 Increment()方法，递增在方法调用中传入的值。Value 属性返回 ProgressBar 的当前值。

可以使用 Text 属性通知用户已完成了操作的百分数或尚未处理的条目数，还有一个 BackgroundImage 属性可以定制进度条的外观。

18.4.13　TextBox 控件、RichTextBox 控件与 MaskedTextBox 控件

TextBox 控件是工具箱中最常用的控件之一。TextBox、RichTextBox 和 MaskedTextBox 控件都派生于 TextBoxBase 类。TextBoxBase 类提供了 MultiLine 和 Lines 属性。MultiLine 属性是一个布尔值，允许 TextBox 控件显示多行文本。文本框中的每一行都是字符串数组的一部分，该数组通过 Lines 属性来访问。Text 属性把整个文本框内容返回为一个字符串。TextLength 是返回的文本字符串的总长。MaxLength 属性把文本的长度限制为指定的数字。

SelectedText、SelectionLength 和 SelectionStart 都处理文本框中当前选中的文本。选中的文本是控件获得焦点时突出显示的文本。

TextBox 控件增加了几个有意义的属性。AcceptsReturn 属性是一个布尔值，允许 TextBox 把【Enter】键接收为一个新行，或者激活窗体上的默认按钮。该属性设置为 True 时，按下【Enter】键会在文本框中创建一个新行。CharacterCasing 确定文本框中文本的大小写。CharactorCasing 枚举包含 3 个值：Lower、Normal 和 Upper。Lower 会使所有的文本小写，Upper 则把所有的文本转变为大写，Normal 把文本显示为输入时的形式。PasswordChar 属性用一个字符表示用户在文本框中输入文本时要显示给用户的内容，这通常用于输入密码和 PIN。Text 属性返回输入的文本，只有显示的内容会受该属性的影响。

RichTextBox 是一个文本编辑控件，它可以处理特殊格式的文本。顾名思义，RichTextBox 控件使用 Rich Text Format(RTF)处理特殊的格式。使用 Selection 属性 SelectionFont、SelectionColor、SelectionBullet 可以修改格式，使用 SelectionIndent、SelectionRightIndent、SelectionHangingIndent 可以修改段落格式。所有 Selection 属性的工作方式都相同。如果有一个选中的文本段，对 Selection 属性的修改就会影响选中的文本。如果没有选中文本，这些修改就对当前插入点后面的文本起作用。

控件的文本可以使用 Text 属性或 Rtf 属性提取。Text 属性只返回控件的文本，而 Rft 属性返回带格式的文本。

LoadFile()方法可以用两种方式从文件中加载文本，它可以使用一个表示文件名和路径的字符串，也可以使用一个流对象，还可以指定 RichTextBoxStreamType。表 18.4 列出了 RichTextBoxStream 的值。

表 18.4　　RichTextBoxStream 的值

值	说　明
PlainText	没有格式信息，包含 OLE 对象，允许使用空格
RichNoOleObjs	Rich 文本格式，但不包含 OLE 对象已经包含的空格
RichText	格式化的 RTF，且包含 OLE 对象
TextTextOleObjs	无格式文本，用文本转换 OLE 对象
UnicodePlainText	与 PlainText 相同，但编码为 Unicode

SaveFile()方法使用相同的参数，把控件中的数据存储在指定的文件中。如果文件已经存在，就覆盖它。

MaskedTextBox 可以限制用户在控件中输入的内容，它还可以自动格式化输入的数据。使用几个属性可以验证或格式化用户的输入。Mask 属性包含覆盖字符串，覆盖字符串类似于格式字符串，使用 Mask 字符串可以设置允许的字符数、允许字符的数据类型和数据的格式。基于 MaskedTextProvider 的类也提供了需要的格式化和验证信息。MaskedTextProvider 只能在它的构造函数中设置。

有 3 个不同的属性返回 MaskedTextControl 的文本。Text 属性返回控件的当前文本，它可以根据控件是否获得焦点而不同，而控件是否获得焦点取决于 HidePromptOnLeave 属性的值。该属性是一个字符串，告诉用户应输入什么内容。InputText 属性总是只返回用户输入的文本。OutputText 属性返回根据 IncludeLiterals 和 IncludePrompt 属性格式化的文本。例如，如果对电话号码进行覆盖，Mask 字符串就应包含括号和几个短横线。这些都是字面字符。如果 IncludeLiteral 属性设置为 True，括号和短横线都应包含在 OutputText 属性中。

MaskedTextBox 控件还有几个额外的事件，OutputTextChanged 和 InputTextChanged 在 InputText 或 OutputText 改变时触发。

18.4.14　Panel 控件

Panel 控件就是包含其他控件的控件。它把控件组合在一起，放在一个面板上，更易于管理这些控件。例如，可以禁用面板，从而禁用该面板上的所有控件。Panel 控件派生于 ScrollableControl 类，所以还可以使用 AutoScroll 属性。如果可用区域上有过多的控件要显示，就可以把它们放在一个面板上，并把 AutoScroll 属性设置为 True，这样就可以滚动所有的控件了。

面板在默认情况下不显示边框，但把 BorderStyle 属性设置为不是 none 的其他值，就可以使用面板可视化地组合相关的控件，这会使用户界面更友好。

面板是 FlowLayoutPanel、TableLayoutPanel、TabPage 和 SplitterPanel 的基类。使用这些控件，可以创建非常复杂或专业化的窗体或窗口。FlowLayoutPanel 和 TableLayoutPanel 对创建正确大小的窗体尤其有帮助。

18.4.15　FlowLayoutPanel 控件和 TableLayoutPanel 控件

FlowLayoutPanel 和 TableLayoutPanel 是.NET Framework 新增的控件。顾名思义，面板可以采用 Web 窗体的方式进行 Windows 窗体布局。FlowLayoutPanel 是一个窗口，允许以垂直或水平的方式旋转包含的控件。除了旋转控件外，还可以剪辑控件。放置方向使用 FlowDirection 属性的 FlowDirection 枚举来设置。WrapContents 属性确定在重新设置窗体的大小时，控件是放在下一行、下一列，还是剪辑控件。

TableLayoutPanel 使用栅格结构控制控件的布局。所有的 Windows 窗体控件都是 TableLayoutPanel

的子控件，包括另一个 TableLayoutPanel，所有窗口的布局可以非常灵活，并可以动态设置。把一个控件添加到 TableLayoutPanel 上时，会给属性页面的 Layout 类别添加 4 个属性：Column、ColumnSpan、Row 和 RowSpan。与 Web 页面上的 HTML 表一样，可以给每个控件设置行和列间距。控件默认放置在表的单元格中心，但这可以使用 Anchor 和 Dock 属性改变。

行和列的默认样式可以使用 RowStyles 和 ColumnStyles 集合改变。这两个集合分别包含 RowStyle 和 ColumnStyle 对象。Style 对象有一个公共属性 SizeType。SizeType 使用 SizeType 枚举来确定列宽或行高。Values 包含 AutoSize、AbSolute 和 Percent。AutoSize 与其他同等控件共享该空间。Absolute 允许使用一组像素值来设置大小。Percent 要求控件把列或宽度设置为父控件的一个百分数。

行、列和子控件都可以在运行期间添加和删除。GrowStyle 属性使用 TableLayoutGrowStyle 枚举值来设置在已填满的表中添加一个新控件时，是给表添加列、行，还是使表保持固定的大小。如果其值是 FixedSized，则在试图添加另一个控件时，就抛出一个 AurgumentException 异常。如果表中的单元格为空，控件就放在空单元格中。该属性仅在表格已满，但要添加控件时起作用。

示例应用程序中的 frmPanel 窗体有 FlowLayoutPanels 和 TableLayoutPanels，并在其中放置了各种控件。试验这些控件，尤其要试用布局面板中控件的 Dock 和 Anchor 属性，这是理解其工作方式的最佳途径。

18.4.16　SplitContainer 控件

SplitContainer 控件把 3 个控件组合在一起，其中有两个面板控件，它们之间有一个分隔栏。用户可以移动分隔栏，重新设置面板的大小。在重新设置面板的大小时，面板上的控件也会重新设置大小。SplitContainer 的最佳示例是资源管理器。左面板包含文件的树形视图，右面板包含文件内容的列表视图。用户在分隔栏上移动鼠标指针时，鼠标指针会改变，此时可以拖动分隔栏。SplitContainer 可以包含任意控件，包括布局面板和其他 SplitContainer，因此，可以创建非常复杂、专业化程度很高的窗体。

分隔栏的移动和定位可以用 SplitterDistance 和 SplitterIncrement 属性控制。SplitterDistance 属性确定分隔栏与控件左边界或顶边的距离。SplitterIncement 确定在拖动时分隔栏移动的像素值。面板可以使用 Panel1MinSize 和 Panel2MinSize 属性设置其最小尺寸，这些属性的单位是像素。

Splitter 控件会引发与移动相关的两个事件 SplitterMoving 和 SplitterMoved。SplitterMoving 事件在移动过程中触发，SplitterMoved 在移动结束后触发，它们都接收一个 SplitterEventArgs。

SplitterEventArgs 的 SplitX 和 SplitY 属性表示 Splitter 左上角的 X 和 Y 坐标，X 和 Y 属性表示鼠标指针的 X 和 Y 坐标。

18.4.17　TabControl 控件和 TabPage 控件

TabControl 允许把相关的组件组合到一系列选项卡页面上。TabControl 管理 TabPage 集合。有几个属性可以控制 TabControl 的外观。Appearance 属性使用 TabAppearance 枚举确定选项卡的外观。其值是 FlagButtons、Buttons 或 Normal。MutiLine 属性的值是一个布尔值，确定是否显示多行选项卡。如果 MultiLine 属性设置为 False，当有多个选项卡不能一次显示出来时，就提供一组箭头，允许用户滚动查看剩余的选项卡。

TabPage 的 Text 属性是在选项卡上显示的内容。Text 属性也在重写的构造函数中用做参数。

一旦创建了 TabPage 控件，它基本上就是一个容器控件，用于放置其他控件。Visual Studio 2010 中的设计器使用集合编辑器，很容易给 TabControl 控件添加 TabPage 控件。在添加每个页面时都可以设置各种属性。接着把其他子控件拖放到每个 TabPage 控件上。

通过查看 SelectedTab 属性可以确定当前的选项卡。每次选择新选项卡时，都会引发 SelectedIndex 事件。通过监听 SelectedIndex 属性，再用 SelectedTab 属性确认当前选项卡，就可以根据每个选项卡进行特定的处理。

18.4.18 ToolStrip 控件

ToolStrip 控件是一个用于创建工具栏、菜单结构和状态栏的容器控件。ToolStrip 直接用于工具栏，还可以用于 MenuStrip 和 StatusStrip 控件的基类。

ToolStrip 控件用于工具栏时，使用一组基于抽象类 ToolStripItem 的控件。ToolStripItem 可以添加公共显示和布局功能，并管理控件使用的大多数事件。ToolStripItem 派生于 System::Component-Model::Component 类，而不是 Control 类。基于 ToolStripItem 的类必须包含在基于 ToolStrip 的容器中。

Image 和 Text 是要设置的最常见属性。Image 可以用 Image 属性设置，也可以使用 ImageList 控件，把它设置为 ToolStrip 控件的 ImageList 属性后，就可以设置各个控件的 ImageList 属性了。

ToolStripItem 上文本的格式化用 Font、TextAlign 和 TextDirection 属性来处理。TextAlign 设置文本于控件的对齐方式，它可以是 ControlAlignment 枚举中的任意值，默认为 MiddleRight。TextDirection 属性设置文本的方向，其值可以是 ToolStripTextDirection 枚举中的任意值，包括 Horizontal、Inherit、Vertical270 和 Vertical90。Vertical270 把文本旋转 270°，Vertical90 把文本旋转 90°。

DisplayStyle 属性控制在控件上是显示文本、图像、文本和图像，还是什么都不显示。在 AutoSize 设置为 True 时，ToolStripItem 会重新设置其大小，以确保使用最少量的空间。

直接派生于 ToolStripItem 的控件如表 18.5 所示。

表 18.5　　直接派生于 ToolStripItem 的控件

ToolStripItems	说　明
ToolStripButton	表示用于可以选择的按钮
ToolStripLabel	显示不能选择的文本或图像，ToolStripLabel 还可以显示一个或多个超链接
ToolStripSeparator	用于分解和组合其他 ToolStripItems，选项根据功能来组合
ToolStripDropDownItem	显示下拉选项，是 ToolStripDropDownButton、ToolstripMenuItem 和 ToolStripSplitButton 的基类
ToolStripControlHost	在 ToolStrip 上存放其他非 ToolStripItem 的派生控件，是 ToolStripComboBox、ToolStripProgressBar 和 ToolStripTextBox 的基类

表 18.5 中的前两项 ToolStripDropDownItem 和 ToolStripControlHost 需要详细探讨。ToolStripDrop DownItem 是 ToolStripMenuItems 的基类，用于建立菜单结构。ToolStripMenuItems 添加到 MenuStrip 控件上。如前所述，MenuStrips 派生于 ToolSrip 控件，在处理和扩展菜单项时，这是很重要的。因为工具栏和菜单派生于同一个类，所以很容易创建管理并执行命令的框架。

ToolStripControlHost 可以包含其他不派生于 ToolStripItem 的控件，可以直接放在 ToolStrip 中的控件是派生于 ToolStripItem 的控件，例 18.6 演示了如何在 ToolStrip 中放置 DateTimePicker 控件。

例 18.6

```
public : midParent()
{
    InitializeComponent();

    ToolStripControlHost^ _dateTimeCtl;
    _dateTimeCtl=gcnew ToolStripControlHost(gcnew DateTimerPicker());
```

```
    ((DateTimePicker^)_dateTimeCtl.Control)->ValueChanged += delegate{
        toolStripLabel1->Text= ((DateTimePicker^)_dateTimeCtl.Control)->Value.Subsract(Date- Time::Now).ToString();
    };
    _dateTimeCtl->Width=200;
    _dateTimeCtl->DisplayStyle=ToolStripItemDisplayStyle->Text;
    toolStrip->Items->Add(_dateTimeCtl);
}
```

这是例 18.2 代码中 frmMain 窗体的构造函数。首先声明并实例化一个 ToolStripControlHost。注意在实例化该控件时，应把窗体要包含的控件传送给构造函数。下一行代码启动 DateTimePicker 控件的 ValueChanged 事件。该控件可以通过 ToolStripHostControl 的 Control 属性访问，并返回一个 Control 对象，因此需要将其类型转换为正确的类型。之后，就可以使用窗体包含的控件的属性和方法了。

提供更好封装性能的另一种方法是创建一个派生于 ToolStripControlHost 的新类。下面的代码是 DateTimePicker 工具栏控件 ToolStripDateTimePicker 的另一个版本：

```
namespace FormsSample.SampleControls
{
    public ref class ToolStripDateTimePicker: public
            System::Windows::Forms::ToolStripControlHost
    {
        //need to declare the event that will be exposed
        public :event EventHandler ValueChanged;

        public ToolStripDateTimePicker() : base(gcnew DateTimePicker())
        {
        }
        //创建强类型控件属性
        public : new DateTimePicker Control
        {
            get { return (DateTimePicker^) base->Control;}
        }
        //创建强类型值属性
        public : DateTimeValue
        {
            get { return Control->Value;}
        }
        protected : override void OnSubscribeControlEvents(Control^ control)
        {
            base->OnSubscribeControlEvents(control);
            ((DateTimePicker^)control)->ValueChanged +=gcnew EventHandler(Handle ValueChanged;
        }
        protected : override void OnUnsubscribeControlEvents(Control^ control)
        {
            base->OnSubscribeControlEvents(control);
            ((DateTimePicker^)control)->ValueChanged +=gcnew EventHandler(Handler ValueChanged);
        }
        private: void HandleValueChanged(object^ sender, EventArgs^ e)
        {
            if(ValueChanged != nullptr)
```

```
                ValueChanged(this, e);
            }
        }
    }
```

该类主要的工作是提供 DateTimePicker 选中的属性、方法和事件。这样，主机应用程序就不必维护底层控件的引用了。提供事件的过程有点复杂。OnSubscribcontrolEvents 方法用于为基于 ToolStripControlHost 的类(本例是 ToolStripDateTimePicker)同步被包含控件(在这里是 DateTimePicker)的事件。在本例中，ValueChanged 事件传送给 ToolStripDateTimePicker，其作用是允许控件的用户在主机应用程序中建立事件，就好像 ToolStripDateTimePicker 派生于 DateTimePicker，而不是 ToolStripControlHost 一样。下面的代码演示了这一点，它使用了 ToolStripDateTimePicker：

```
public mdiParent()
{
    ToolStripDateTimePicker^ otherDateTimePicker= gcnew ToolStripDateTimePicker();
    otherDateTimePicker->Width=200;
    otherDateTimePicker->ValueChanged +=gcnew EventHandler(otherDateTimePicker _ValueChanged);
    toolStrip1->Items->Add(otherDateTimePicker);
}
```

注意在建立 ValueChanged 事件处理程序时，要引用 ToolStripDateTimePicker 类，而不是像前面的示例那样引用 DateTimePicker 控件。本示例的代码与第一个例子相比简单了很多。不仅如此，由于 DateTimePicker 封装在另一个类中，因此封装性能大大提高了。ToolStripDateTimePicker 在应用程序的其他部分中的使用也简单了许多。

18.4.19 MenuStrip 控件

MenuStrip 控件是应用程序菜单结构的容器。如前所述，MenuStrip 派生于 ToolStrip 类。在建立菜单系统时，要给 MenuStrip 添加 ToolStripMenu 对象。这可以在代码中完成，也可以在 Visual Studio 的设计器中进行。把一个 MenuStrip 控件拖放到设计器的一个窗体中，MenuStrip 就允许直接在菜单项上输入菜单文本。

MenuStrip 控件只有两个额外的属性。GripStyle 使用 ToolStripGripStyle 枚举把栅格设置为可见或隐藏。MdiWindowsListItem 属性提取或返回 ToolStripMenuItem。该 ToolSripMenuItem 是在 MDI 应用程序中显示所有已打开窗口的菜单。

18.4.20 ContextMenuStrip 控件

要显示弹出菜单，或在用户右击鼠标时显示一个菜单，就应使用 ContextMenuStrip 类。与 MenuStrip 一样，ContextMenuStrip 也是 ToolStripMenuItems 对象的容器，但它派生于 ToolStripDropDownMenu 类。ContextMenu 的创建与 MenuStrip 相同，也是添加 ToolStripMenuItems，定义每一项 Click 事件，执行某个任务，弹出菜单赋予特定的控件，为此，要设置控件的 ContextMenuStrip 属性。在用户右击该控件时，就显示该菜单。

18.4.21 ToolStripMenuItem 控件

ToolStripMenuItem 是建立菜单结构的类。每个 ToolStripMenuItem 对象都表示菜单系统上的一个

菜单选项。每个 ToolStripMenuItem 都有一个包含子菜单的 ToolStripItemCollection。该功能继承自 ToolStripDropDownItem。

由于 ToolStripMenuItem 派生于 ToolStripItem 类，因此可以使用所有的格式化属性。图像在菜单文本的右边显示为小图标。菜单项的旁边可以有复选框标记，可用 Checked 和 CheckState 属性设置该标记。

还可以给每个菜单项指定快捷键。快捷键一般包含两个按键，如【Ctrl+C】（Copy 的快捷键）。在知道快捷键时，将 ShowShortCutKey 属性设置为 True，还可以在菜单上显示该快捷键。

在用户单击菜单项或使用定义好的快捷键时，应执行某个任务。为此，最常见的方式是处理 Click 事件。如果使用了 Checked 属性，还可以使用 CheckstateChanged 和 CheckedChanged 事件确定选中状态的变化。

18.4.22　ToolStripManager 类

菜单和工具栏的结构可以很复杂，这样就难以管理。ToolStripManager 类可以创建较小、易于管理的菜单或工具栏结构，并在需要时合并它们。例如，如果窗体上有几个不同的控件，每个控件都必须显示一个弹出菜单。所有的菜单都要使用某些菜单项。但每个控件还有几个独特的菜单项。可以在一个 ContextMenuStrip 属性上定义共有菜单项，独特的菜单项可以预先定义，也可以在运行期间创建。对于需要弹出菜单的控件，应复制共有菜单项，再使用 ToolStripManager.Merge()方法将独特的菜单项与共有菜单选项合并起来，最后将得到的菜单赋予控件的 ContextMenuStrip 属性。

18.4.23　ToolStripContainer 控件

ToolStripContainer 控件用于停放基于 ToolStrip 的控件。添加一个 ToolStripContainer，将 Docked 属性设置为 Fill，就在窗体的两侧边添加了 ToolStripPanel，在窗体的中间添加了 ToolStripContainerPanel，在 ToolStripPanels 中可以添加任意 ToolStrip（ToolStrip、MenuStrip 或 StatusStrip）。用户可以选择 ToolStrips，将其拖放到窗体的两边或底部。将 ToolStripPanels 的 Visible 属性设置为 False，就不能把 ToolStrip 放在面板上了。窗体中心的 ToolStripContainerPanel 可用于放置窗体需要的其他控件。

18.5　窗体 WinForm

前面讨论了如何创建简单的 Windows 应用程序。该示例包含一个派生于 System::Windows::Forms.Form 的类。根据.NET Framework 说明文档，“窗体是应用程序中窗口的表示方式。”如果已具有 Visual Basic 基础，就会很熟悉术语“窗体”。如果是使用 MFC 的 C++程序员，就可能习惯把窗体称为窗口、对话框或框架。

窗体是与用户交互的基本方式。前面已经介绍了 Control 类中一些较常见的属性、方法和事件，而且 Form 类派生于 Control 类，所以 Form 类中也存在这些属性、方法和事件。Form 类还在 Control 类的基础上添加了大量的功能。本节将介绍这些功能。

18.5.1　Form 类

Windows 客户应用程序可以包含一个窗体或上百个窗体。它们可以是基于 SDI（Single Document Interface，单文档界面）或 MDI（Multiple Document Interface，多文档界面）的应用程序。但无论怎样，

System::Windows::Forms.Form 类都是 Windows 客户应用程序的核心。Form 类派生于 ContainerControl 类，ContainerControl 类又派生于 ScrollableControl 类，ScrollaleControl 类则派生于 Control 类，因此可以假定，窗体可以是其他控件的容器，当其包含的控件在客户区域中显示不下时，可以滚动显示。窗体可以拥有与其他控件相同的属性、方法和事件。所以，Form 类相当复杂。

1. 窗体的实例化和释放

理解创建窗体的过程是很重要的。要完成的工作取决于编写初始化代码的位置。对于实例，事件按如下顺序发生：

（1）构造函数；

（2）Load；

（3）Activated；

（4）Closing；

（5）Closed；

（6）Deactivate。

前 3 个事件在初始化的过程中发生。根据初始化的类型，可以确定要关联哪个事件。该类的构造函数在对象的实例化过程中执行。Load 事件在对象实例化后、窗体可见之前发生，它与构造函数的区别是窗体的可见性。在触发 Load 事件时，窗体已存在，但不可见。在构造函数执行过程中，窗体还不存在，处在实例化过程中。Activated 事件在窗体处于可见状态并处于当前状态时发生。

有一种情形会略微改动一下事件执行顺序。如果在窗体构造函数执行的过程中，Visible 属性设置为 True，或调用 Show 方法（它把 Visible 属性设置为 True），就会立即触发 Load 事件。这也会使窗体可见，并处于当前状态，所以还会触发 Activate 事件。如果在设置 Visible 属性后还有代码，就执行这些代码。所以启动事件的执行顺序如下所示：

（1）构造函数，执行到 Visible=true 为止；

（2）Load；

（3）Activate；

（4）构造函数，执行 Visible=true 之后的代码。

这会产生一些无法预料的结果。尽可能在构造函数中进行初始化。

关闭窗体时会发生什么情况？Closing 事件可以取消处理，它把 CancelEventArgs 作为一个参数，如果把 Cancel 属性设置为 True，就会取消该事件，窗体仍处于打开状态。Closing 事件在窗体关闭时发生，而 Closed 实际在窗体关闭后发生。这两个事件都允许执行必要的清理工作。注意 Deactivate 事件在窗体关闭后发生，这是另一个可能产生难以查找的错误的来源。确保不再 Deactive 事件中执行防止窗体被正常垃圾回收的操作。例如，设置对另一个对象的引用会使窗体仍未被释放。

如果调用 Application.Exit()方法，且当前有一个或多个窗体处于打开状态，就不会触发 Closing 和 Closed 事件。如果打开了正要清理的文件或数据库连接，这就是一个需要考虑的问题。此时应调用 Dispose()方法，所以另一种更好的方法是把大多数清理代码放在 Dispose()方法中。

与窗体启动相关的一些属性有 StartPosition、ShowInTaskBar 和 TopMost，StartPosition 可以是 FormStartPosition 枚举中的一个值。

- CenterParent: 窗体位于父窗体的客户区域中心。
- CenterScreen: 窗体位于当前屏幕的中心。
- Manual: 窗体的位置根据 Location 属性的值来确定。
- WindowsDefaultBounds: 窗体位于默认位置，使用默认的大小。

- WindowsDefaultLocation：窗体位于默认位置，但其大小根据 Size 属性来定。

ShowInTaskbar 属性确定窗体是否应在任务栏上可见。当窗体是一个子窗体，且只希望父窗体显示在任务栏上时，才使用该属性。TopMost 属性指定窗体在应用程序启动时位于最上面，即使窗体没有立即获得焦点，也位于最上面。

为了让用户与应用程序交互，用户必须能看到窗体。利用 Show()和 ShowDialog()方法就可以实现这一点。Show()方法仅使窗体对用户可见。下面的代码演示了如何创建一个窗体，并把它显示给用户，假定要显示的窗体为 MyFormClass：

```
MyFormClass myForm=new MyFormClass();
myForm.Show();
```

这非常简单，但其缺点是没有向调用代码发出任何通知，说明 MyForm 已经处理完毕，并退出。这并不重要，但 Show()方法工作得很好。

ShowDialog()方法是提供某种通知比较好的选择。

在调用 Show()方法后，Show()方法后面的代码会立即执行。在调用 ShowDialog()方法后，调用代码被暂停执行，等到调用 ShowDialog()方法的窗体关闭后再继续执行。不仅调用代码被暂停执行，窗体还可以返回一个 DialogResult 值。DialogResult 枚举是一组标识符，它们描述了对话框关闭的原因，包括 OK、Cancel、Yes、No 和其他几个标识符。为了让窗体返回一个 DialogResult 值，必须设置窗体的 DialogResult 属性，或者在窗体的一个按钮上设置 DialogResult 属性。

例如，假定应用程序的一部分要求提供客户的电话号码，窗体中包含一个输入电话号码的文本框和两个按钮——OK 和 Cancel。如果把 OK 按钮的 DialogResult 属性设置为 DialogResult.OK，把 Cancel 按钮的 DialogResult 属性设置为 DialogResult.Cancel，则在选择其中一个按钮时，窗体就会不可见，并向正在调用的窗体返回相应的 DialogResult 值。此时注意窗体没有释放，只是把 Visible 属性设置为 False，这是因为必须从窗体中获取值。在本示例中，需要电话号码，在窗体上为电话号码创建一个属性，这样父窗体就可以获取值，并调用窗体上的 Close()方法了。下面就是子窗体的代码：

```
namespace FormsSample::DialogSample
{
    partial ref class Phone:public Form
    {
        public Phone()
        {
            InitializeComponent();
            btnOK->DialogResult=DialogResult::OK;
            btnCancel->DialogResult=DialogResult::Cancel;
        }
        public : property String PhoneNumber
        {
            String get() { return textBox1->Text;}
            void set(String^value) { textBox1->Text=value;}
        }
    }
}
```

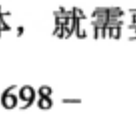
首先要注意此处不包含处理按钮单击事件的代码。因为设置了每个按钮的 DialogResult 属性，所以在单击 OK 或 Cancel 按钮后，窗体就消失了。添加的惟一属性是 PhoneNumber。下面的代码显示了父窗体中调用 Phone 对话框的方法：

```
Phone^ frm=gcnew Phone();
```

```
Frm->ShowDialog();
if(frm->DialogResult==DialogResult::OK)
{
    label1->Text=L"Phone number is" + frm->PhoneNumber;
}
else if (frm->DialogResult==DialogResult::Cancel)
{
    label1->Text=L"Form was Canceld"
}
Frm->Close();
```

创建新的 Phone 对象 frm，在调用 frm.ShowDialog()方法时，该方法中的代码会停止执行，等待 Phone 窗体返回。接着检查 Phone 窗体的 DialogResult 属性。由于窗体还未释放，是不可见的，因此仍可以访问公共属性，其中一个公共属性就是 PhoneNumber。一旦获取了需要的数据，就可以调用窗体的 Close()方法。

如果返回的电话号码格式不正确，而其他一切正常，应该怎么办呢？如果把 ShowDialog()放在循环中，就可以再次调用它，让用户重新输入值，这样就可以得到正确的值。



如果用户单击了 Cancel 按钮，还必须处理 DialogResult.Cancel。

```
Phone^frm=gcnew Phone();
while(true)
{
    frm->ShowDialog();
    if(frm->DialogResult==DialogResult::OK)
    {
        lable1->Text=L"Phone number is" + frm->PhoneNumber;
        if(frm->PhoneNumber->Length==8
                || frm->PhoneNumber->Length ==12)
        {
            break;
        }
        else
        {
            MessageBox::Show(L"Phone number was not formatted correctly.
                    Please correct entry.");
        }
    }
    else if(frm->DialogResult==DialogResult::Cancel)
    {
        label1->Text=L"Form was canceled.";
        break;
    }
}
frm->Close();
```

如果电话号码的长度没有通过简单的测试，Phone 窗体就会显示出来，让用户更正错误。ShowDialog 没有创建窗体的新实例，在窗体上输入的文本仍在该窗体上，所以如果必须重新设置窗体，就需要程序员自己完成。

2．外观

用户首先看到的是应用程序的窗体。这是应用程序中第一个也是最重要的功能。如果应用程序不解决业务问题，则其外观就无关紧要了。这并不是说，窗体和应用程序的整体 GUI 设计不应美观，像颜色组合、字体大小和窗口大小等的设计都可以使应用程序更容易让用户操作和接受。

有时不希望用户访问系统菜单。在单击窗口左上角的图标时，该菜单就会显示出来。一般情况下，它包含还原、最小化、最大化和关闭等命令。ControlBox 属性允许设置系统菜单的可见性，还可以用 MaximizeBox 和 MinimizeBox 属性设置最大化和最小化按钮的可见性。如果删除了所有的按钮，再把 Text 属性设置为空字符串（""），标题栏就会完全消失。

如果设置了窗体的 Icon 属性，但没有将 ContrlBox 属性设置为 False，图标就会显示在窗体的左上角。通常应把 Icon 属性设置为 app.ico，这会使每个窗体的图标都与应用程序的图标相同。

FormBorderStyle 属性用于设置显示在窗体周围的边框类型。它使用 FormBorderStyle 枚举，其值是：

- Fixed3D;
- FixedDialog;
- FixedSingle;
- FixedToolWindow;
- None;
- Sizable;
- SiazbleToolWindow。

大多数值的意义都一目了然，只有两个工具窗口边框除外。无论怎样设置 ShowInTaskBar 属性，工具窗口都不显示在任务栏上。当用户按【Alt+Tab】组合键时，工具窗口也不会显示在窗口列表中。默认设置是 Sizable。

除非有明确要求，否则大多数 GUI 元素的颜色都应设置为系统颜色，而不是特定的颜色。这样，如果一些用户喜欢把所有的按钮都设置为紫字绿底，应用程序就会采用这种颜色设置。为了把控件设置为使用特定的系统颜色，必须调用 System::Drawing.Color 类的 FromKnownColor()方法。FromKnownColor()方法将一个 KnownColor 枚举值作为其参数。该枚举中定义了许多颜色和各种 GUI 元素颜色，如 Control、ActiveBorder 和 Desktop。例如，如果窗体的背景色应总是匹配 Desktop 颜色，就应使用下面的代码。

```
myForm->BackColor=Color::FromKnownColor(KnownColor::Desktop);
```

如果用户改变了桌面的颜色，窗体的背景色也会随之改变，这将增强应用程序的友好性，用户可以为桌面选择某种颜色组合，这由他们的偏好决定。

Windows XP 引入了一个叫做可视化样式的特性。当鼠标指针悬停在按钮、文本框、菜单和其他控件上或单击它们时，这些控件的外观会改变，并做出响应。调用 Application::EnableVisualStyles()方法，可以激活应用程序的可视化样式。该方法必须在实例化任何类型的 GUI 之前调用。所以，它一般在 Main()方法中调用，如下面的示例所示：

```
[STAThreadAttribute]
int main(array<System::String ^> ^args)
 {
     Application::EnableVisualStyles();
     Application::Run(gcnew Form1());
}
```

这段代码允许支持可视化样式的各种控件采用可视化样式。由于 EnableVisualStyles()方法存在一个问题，所以必须在调用 EnableVisualStyles()之后立即添加 Application.DoEvents()方法。这能够解决工具栏上的图标在运行期间消失的问题。EnableVisualStyles()方法也可以在.NET Framework 2.0 中使用。

对于控件，还有一个必须完成的任务。大多数控件都包含 FaltStyle 属性，它把 FaltStyle 枚举作为其值。该属性可以使用如下 4 个值。

- Flat: 当鼠标指针悬停在控件上时，控件显示为平面模式。
- Flat3D: 类似于 Flat，但当鼠标指针悬停在控件上时，控件显示为 3D 模式。
- Standard: 控件显示为 3D 模式。
- System: 控件的外观由操作系统控制。

为了在控件上使用可视化样式，FaltStyle 属性应设置为 FaltStyle.System。应用程序如今大多采用 Windows XP 的外观和操作方式。

18.5.2 多文档界面

当应用程序可以显示同一类型窗体的多个实例，或以某种方式包含不同的窗体时，就应使用 MDI 类型的应用程序。同一类型窗体多个实例的一个示例就是文本编辑器，它可以同时显示多个编辑窗口。第二种应用程序的示例就是 Microsoft Access，可以在 Access 中同时打开查询窗口、设计窗口和表窗口，这些窗口都不会超出主 Access 应用程序的边界。

包含本章示例的项目就是一个 MDI 应用程序，项目中的窗体 midParent 就是 MDI 父窗体。将 IsMdiContainer 设置为 True，就会把该窗体设置为 MDI 父窗体。如果在设计器中创建窗体，注意其背景会变成暗灰色，说明这是一个 MDI 父窗体。仍可以给该窗体添加控件，但最好不要这样做。

为了使子窗体成为 MDI 子窗体，子窗体需要知道其父窗体是哪个窗体。为此，应把子窗体的 MdiParent 属性设置为父窗体。在这个例子中，所有的子窗体都是用 ShowMdiChild()方法创建的，它的参数是要显示的子窗体的引用。把 MdiParent 属性设置为 this（表示引用 mdiParent 窗体）后，就显示该窗体。下面是 ShowMdiParent 方法的代码：

```
private : void ShowMdiChild(Form^ childForm)
{
    childForm->MdiParent=this;
    childForm->Show();
}
```

MDI 应用程序的一个问题是，在任意时刻都可以打开几个子窗体。对当前活动的子窗体的引用可以使用父窗体的 ActivateMdiChild 属性来提取。这就是 Windows 菜单中的 Current Active 菜单项的作用。单击该菜单项，会显示一个包含窗体名称和文本值的消息框。

子窗体可以调用 LayoutMdi()方法来安排。LayoutMdi()方法把 MdiLayout 枚举值作为参数，其值可以是 Cascade、TileHorizontal 和 TileVertical。

18.5.3 定制控件

使用控件和组件是使窗体软件包（如 Windows 窗体）的开发非常高效的一个主要因素。创建自己的控件、组件和用户控件会使开发更加高效，因为创建控件可以把功能封闭在能够多次使用的软件包中。

创建控件有许多方式，可以从头开始创建，从 Control 类、ScrollableControl 类或 ContainerContrl 类中派生自己的类。除了为给控件添加需要的功能以外，还必须重写 Paint 事件，完成绘制工作。

如果控件是当前属性的一个改进版本，就必须从该控件中派生出要改进的控件。例如，如果需要一个 TextBox 控件，但要改变其背景色，设置其 ReadOnly 属性，则创建一个全新的 TextBox 控件就会浪费时间。其实只需从 TextBox 控件中派生，再重写 ReadOnly 属性即可。由于 TextBox 控件的 ReadOnly 属性没有标记为重写，所以必须使用 new 语句。下面的代码展示了新的 ReadOnly 属性：

```
public : property bool ReadOnly
{
    bool get() { return base->ReadOnly; }
    void set(bool value) {
        if(value)
            this->BackgroundColor = Color::Red;
        else
            this->BackgroundColor=Color::FromKnowColor(KnowColor::Window);

    base->ReadOnly=value;
}
```

对于 get 属性，返回为基本对象设置的内容。把文本框设置为只读的属性在这里并不重要，应把这个功能传递给基本对象。在属性集中检查传递的值是 True 还是 False。如果是 True，就要将颜色改为只读颜色（在本例中是 Red）；如果是 False，就把 BackGroundColor 设置为默认值。最后，把值传送给基本对象，让文本框成为只读文本框。可以看出，重写一个简单的属性，就可以给控件添加一个新功能。

1．控件属性

可以给定制控件添加属性，改进设计期间控件的功能。表 18.6 列出了一些比较有用的属性。

表 18.6　有用的属性

属 性 名 称	描　　述
BindableAttribute	在设计期间用于确定属性是否支持双向数据绑定
BrowsableAttribute	确定属性是否显示在可视化设计器中
CategoryAttribute	确定在属性窗口中，属性显示在哪个类别中。可使用预定义的类别或创建新的类别。默认值是 Miisc
DefaultEventAttrbute	指定类的默认属性
DefaultPropertyAttribute	指定类的默认属性
DefaultValueAttribute	指定属性的默认值，一般是初始值
DecriptionAttribute	在选中属性时，这是显示在设计器窗口底部的文本
DesignOnlyAttribute	把属性标记为只能在设计模式下编辑

另外，还有其他属性（与在设计期间使用的编辑器相关）以及一些在设计期间使用的高级功能。此外还应添加 Category 和 Description 属性，它们能帮助使用该控件的其他开发人员更好地理解属性的作用。为了添加 Intelligence 支持，应给每个属性、方法和事件都添加 XML 注释。在使用/doc 选项编译控件时，所生成的注释 XML 文件将为控件提供 Intelligence 支持。

2．基于 TreeView 的定制控件

下面将介绍如何根据 TreeView 控件开发定制控件。该控件显示驱动器中的文件结构。这里将为其添加属性，设置基本文件夹或根文件夹，确定显示哪些文件和文件夹，并使用上一节讨论的各种属性。

与任何新项目一样，也必须为控件定义需求。下面是必须完成的基本需求列表。

- 读取文件夹和文件，并显示给用户。
- 在树形层次结构视图中显示文件夹结构。
- 可以隐藏文件。
- 定义哪个文件夹是基本文件夹或根文件夹。
- 返回当前选中的文件夹。
- 提供延迟加载文件结构的功能。

把 TreeView 控件作为新控件的基本控件，就可以满足上述的一个需求。

TreeView 控件以层次结构的形式显示数据。它显示的数据描述了列表中的对象，并可以带有图标。单击对象或使用箭头键，就可以展开和折叠这个列表。

在 Visual Studio 中创建一个新的 Windows 控件库项目，命名为 FolderTree，删除类 UserControl1，添加一个新类，命名为 FolderTree。因为 FolderTree 类派生于 TreeView 类，所以类声明由：

```
public ref class FolderTree
```

改为：

```
public ref class FolderTree :public System::Windows::Forms::TreeView
```

此时就有了一个功能全面且能工作的 FolderTree 控件了。该控件可以完成 TreeView 能够完成的所有任务。

TreeView 控件包含一个 TreeNode 对象集合。我们不能直接把文件和文件夹加载到控件中，但有两种方式可以把加载的 TreeNode 映射到 TreeView 的 Node 集合和它表示的文件与文件夹中。

例如，在处理每个文件夹时，都会创建一个新的 TreeNode 对象，Text 属性设置为文件或文件夹的名称。如果在某一刻需要文件夹的其他信息，就必须再次进入磁盘收集该信息，或把与文件或文件夹相关的其他数据存储在 Tag 属性中。

另一个方法是创建一个派生于 TreeNode 的新类，可以添加新属性和方法，且仍能使用 TreeNode 的基本功能。本例就使用该方法，其设计更为灵活。如果需要新属性，就可以添加，而无须中断已有的代码。

有两类对象必须加载到控件中：文件和文件夹。每类对象都有自己的特性。例如，文件夹有一个 DirectoryInfo 对象，它包含了额外的信息，而文件有一个 FileInfo 对象。由于有这些区别，所以使用两个类来加载 TreeView 控件。FileNode 类的代码如下：

```
namespace FormsSample::SampleControls
{
    public ref class FileNode :public System::Windows::Forms::TreeNode
    {
        String^ _fileName=L"";
        FileInfo^ _info;
        public FileNode(string^ filename)
        {
            _fileName=filename;
            _info=gcnew FileInfo(_fileName);
            base->Text=_info->Name;
            if(_info->Extension->ToLower()==L".exe")
                this->ForeColor=System::Drawing::Color::Red;
        }
        public :property String^ FileName
        {
            String^ get() { return _fileName;}
```

```
            void set(String^value) { _fileName=value;}
        }
        public : property FileInfo^ FileNodeInfo
        {
            FileInfo^ get() { return _info;}
        }
    }
}
```

正在处理的文件名被传送给 FileNode 的构造函数。在构造函数中为文件构建 FileInfo 对象，并把它设置为成员变量_info。base.Text 属性设置为文件名。因为 FileNode 派生于 TreeNode，所以设置 TreeNode 的 Text 属性，这是显示在 TreeView 控件中的文本。

再添加两个属性来获取数据。FileName 返回文件名，FileNodeInfo 返回文件的 FileInfo 对象。

下面是 FolderNode 类的代码。其结构非常类似于 FileNode 类，区别是用 DirectoryInfo 属性代替了 FileInfo 属性，用 FolderPath 代替了 FileName：

```
namespace FormsSample::SampleControls
{
    public ref class FolderNode :public System::Windows::Forms::TreeNode
    {
        String^ _folderPath=L"";
        DirectoryInfo ^_info;
        public: FolderNode(string^ folderPath)
        {
            _folderPath=folderPath
            _info=gcnew DirectoryInfo(folderPath);
            this->Text=_info->Name;
        }

        public: property String^ FolderPath
        {
            String^ get(){ return _folderPath;}
            void set(String^value) { folderPath=value;}
        }
        public:property DirectoryInfo^ FolderNodeInfo
        {
            DirectoryInfo^ get(){ return _info;}
        }
    }
}
```

下面构建 FolderTree 控件。根据要求，需要一个属性来读取和设置 RootFolder，还需要 ShowFiles 属性来确定文件是否显示在树中。SelectedFolder 属性返回树中当前突出显示的文件夹。下面是 FolderTree 控件的代码：

```
using System;
using System.Windows.Forms;
using System.IO;
using System.ComponentModel;

namespace FolderTree
```

```
{
    ///<summary>
    /// Summary description for FolderTreeCtrl.
    ///</summary>
    public class FolderTree : System.Windows.Forms.TreeView
    {
        string _rootFolder="";
        bool _showFiles=true;
        bool _inInit=false;
        public FolderTree()
        { }
        [Category("Behavior"),
          Description("Gets or sets the base or root folder of the tree"),
          DefaultValue("C:\\")]
        public string RootFolder
        {
            get{ return _rootFolder;}
            set
            {
                _rootFolder=value;
                if(!=_inInit)
                    InitializeTree();
            }
        }
        [Category("Behavior"),
            Description("Indicates whether files will seen in the list."),
            DefaultValue(true)]
        public bool showFiles
        {
            get { return _showFiles;}
            set { _showFiles=value;}
        }
        [Browsable(false)]
        public string SelectedFolder
        {
            get
            {
                if(this.SelectedNode is FolderNode)
                    return(FolderNode)this.SelectedNode).FolderPath;
                return "";
            }
        }
    }
}
```

这里添加了 3 个属性：ShowFiles、SelectedFolder 和 RootFolder。为 ShowFiles 和 RootFolder 设置 Category、Description 和 DefaultValues。这两个属性显示在设计模式下的属性浏览器中。SelectedFolder

在设计期间没有什么意义，所以选择 Browsable=false 属性。SelectedFolder 不在属性浏览器中显示，但由于它是一个公共属性，因此会在 Intelligence 中出现，并可以通过代码访问它。

接着，初始化文件系统的加载操作。初始化控件有些困难，在设计期间和运行期间进行初始化都必须仔细考虑。控件位于设计器中时，实际是在运行。如果在构造函数中调用了一个数据库，该调用就是在设计器中拖放控件时进行的。在拖放 FolderTree 控件时，就存在一个问题。下面看看加载文件的方法：

```
private void LoadTree(FolderNode folder)
{
    string [] dirs=Directory.GetDirecoties(folder.FolderPath);
    foreach(string dir in dirs)
    {
        folderNode tmpfolder = new FolderNode(dir);
        folder.Nodes.Add(tmpfolder);
        LoadTree(tmpfolder);
    }
    if(_showFiles)
    {
        string [] files=Directory.GetFiles(folder.FolderPath);
        foreach(string file in files)
        {
            FileNode fnode=new FileNode(file);
            folder.Nodes.Add(fnode);
        }
    }
}
```

showFiles 是一个布尔成员变量，在 ShowFiles 属性中设置。如果设置为 True，就在树中显示文件。现在惟一的问题是何时调用 LoadTree()方法。有几种选择，可以在设置 RootFolder 属性时调用，这在一些情况下是很理想的，但在设计期间并不合适。控件在设计器中是激活的，所以在设置 RootNode 属性时，控件就会试图加载文件系统。

要解决这个问题，应查看 DesignMode 属性，如果控件在设计器中，它就返回 True。于是可以编写代码，初始化控件了：

```
private : void InitializeTree()
{
    if(!this->DesignMode&& _rootFolder!=L"")
    {
        FolderNode^ rootNode=gcnew FolderNode(_rootFolder);
        LoadTree(rootNode);
        this->Nodes->Clear();
        this->Nodes->Add(rootNode);
    }
}
```

如果控件不处于设计模式，且_rootFolder 也不是空字符串，应开始加载树。先创建 Root 节点，再将其传送给 LoadTree()方法。

另一个选择是执行公共方法 Init()，在 Init()方法中调用 LoadTree()方法。这个选择存在的问题是使用控件的开发人员需要调用 Init()。根据情况的不同，这或许是一个可以接受的解决方法。

对于添加的灵活性，可以实现 ISupportInitialize 接口。ISupportInitialize 有两个方法：BeginInit()和

EndInit()。在控件实现 ISupportInitialize 接口时，就会在 InitializeComponent 生成的代码中自动调用 BeginInit()和 EndInit()方法。这样就会延缓初始化过程，直到设置完所有的属性为止。ISupportInitialize 还允许父窗体中的代码延缓初始化过程。如果在代码中设置 RootNode，先调用 BeginInit()就可以在控件加载文件系统之前设置 RootNode 属性和其他属性，或执行动作。在调用 EndInit()时，初始化控件。BeginInit()和 EndInit()方法如下所示：

```
#region ISupportInitialize Members
void ISupportInitialize::BeginInit()
{
    _inInit=true;
}
void ISupportInitialize::EndInit()
{
    if(_rootFolder!="")
    {
        InitializeTree();
    }
    _inInit=false;
}
#endregion
```

在 BeginInit()方法中，只把成员变量_inInit 设置为 True。该标志用于确定控件是否处于初始化过程，并在 RootFolder 属性中使用。如果在 InitializeComponent 类的外部设置 RootFolder 属性，就需要重新初始化树。在 RootFolder 属性中，检查_inInit 是 True 还是 False。如果是 True，就不需要查看初始化过程。如果 inInit 是 False，就调用 InitializeTree()。还可以用一个公共方法 Init()完成这个任务。

在 EndInit()方法中，检查控件是否处于设计模式，_rootFolder 是否被赋予了有效的路径，然后仅调用 InitializeTree()。

为了添加专业化的外观，需要添加一个位图图像。当把控件添加到项目中时，该图像将显示在工具箱中，位图图像的尺寸应是 16 像素×16 像素、16 色。可以用任意图形编辑器创建该图像文件，甚至可以在 Visual Studio 2010 中创建这个文件，只要正确设置大小和颜色深度即可。

要测试控件，可在同一个解决方案中创建 TestHarness 项目。TestHarness 是一个简单的 Windows 窗体应用程序，只包含一个窗体。在引用部分添加对 FolderTreeCtl 项目的引用，在 Toolbox 窗口中添加对 FolderTreeCtl.DLL 的引用。FolderTreeCtl 现在应显示在工具箱中，且位图已添加为图标。单击该图标，将其拖放到 TestHarness 窗体上，把 RootFolder 设置为一个可用的文件夹，并运行解决方案。

要成为一个完整的软件，还必须做一些工作以改进控件，使之成为功能全面、可用于生产成品的控件。例如，可以添加：

- 异常；
- 后台加载；
- 颜色代码；
- 图标。

3．用户控件

用户控件是 Windows 窗体中功能比较强大的一个特性。它们允许把用户界面设计封闭到可重用的软件包中，以便于以后把软件包插入其他项目。公司有若干个常用用户控件库是最常见的，不仅用户界面功能可以包含在用户控件中，而且常见的数据验证也可以包含在用户控件中，如格式化电话号码

或 id 号。用户控件中可以包含一个预定义的项目列表，以便快速加载列表框或组合框。状态代码或国家代码也可以划在这个类别中。在用户控件中尽可能组合多种与当前应用程序不相关的功能，会使控件在公司中的用途更加强大。

18.6　小结

本章介绍了建立基于 Windows 的客户应用程序的基础知识，解释了每个基本控件，讨论了 Windows::Forms 命名空间的层次结构，论述了控件的各种属性和方法。

此外，还讨论了如何创建基本定制控件和基本用户控件。创建自己的控件是非常灵活的。通过创建自己的定制控件工具箱，基于 Windows 的客户应用程序将更容易开发和测试，因为可以重用测试过的同一组件。

第 19 章　Visual C++ 2010 MFC 与.NET 交互编程

从 Visual C++ .NET 开始，微软为 C++编译器添加了托管扩展功能，使本地 C++可用于开发在.NET 框架的公共语言运行时（CLR）顶层运行的托管应用程序。托管扩展是一种允许综合运用托管语法和非托管代码或本地代码的语言结构。例如，可以在 MFC 应用程序中使用.NET 类库提供的众多实用类。

19.1　编写托管扩展应用程序

在学习混合模式编程之前，首先了解.NET 应用程序的一些基本概念。对于已经学过.NET 编程的读者，完全可以跳过这一节。从一个实例着手，首先创建一个简单的托管扩展控制台应用程序。

（1）选择 File|New Project 命令，启动新建工程向导。

（2）弹出 New Project 对话框后，选择 Visual C++项目类型，在右边的列表中选择 CLR Console Application。

（3）在下方的文本框中输入一个工程名。

（4）单击 OK 按钮，向导将生成该项目及所要的文件。

解决方案资源管理器（Solution Explorer）中包含当前解决方案中定义的所有项目列表和每个项目的文件列表。这里面包含一个很重要的 AssemblyInfo.cpp 文件，该文件包含与当前模块的程序集相关的信息。下面简单介绍程序集的.NET 定义以及它是如何与该文件关联的。

所有使用.NET CLR 的.NET 二进制文件均包含一个程序集或一个程序集集合。因此创建这个 CLR 控制台应用程序时，实际上就是创建一个.NET 程序集，而不是创建标准的 Windows 可移植可执行文件。从不同的角度考虑，程序集具有不同的含义：从客户的角度来看，程序集就是已命名的具有版本信息和自描述的模块集合或导出的类型或资源；而从程序集创建者的角度来看，程序集是打包相关模块、类型及资源的方式，而且只导出客户应该使用的部分。而程序集清单正是程序集的实现细节与程序集客户之间的非直接层次，下面列出程序集清单常见的若干元素。

- 程序集名字：程序集的文本化名字。
- 版本信息：程序集的版本信息，由 4 部分组成，包括主版本号、次版本号、修订号和构建号。
- 共享名字和签名的程序集散列：与程序集的部署有关。
- 文件：程序集包含的所有文件的列表。
- 引用的程序集：说明本程序集引用的所有外部程序集列表。
- 类型：程序集中定义的所有类型的列表。
- 安全权限：程序集显式拒绝的安全性权限列表。
- 产品信息：包含程序集开发者有关的信息。

下面来看向导生成的 AssemblyInfo.cpp 文件中的内容，如例 19.1 所示。

例 19.1

```
#include "stdafx.h"
using namespace System;
using namespace System::Reflection;
using namespace System::Runtime::CompilerServices;
using namespace System::Runtime::InteropServices;
using namespace System::Security::Permissions;
//
//通过修改下面的属性来控制程序集的信息
//这些属性依次是
//
[assembly:AssemblyTitleAttribute("Firstapp")];                          //程序集名称
[assembly:AssemblyDescriptionAttribute("")];                            //程序集描述信息
[assembly:AssemblyConfigurationAttribute("")];                          //程序集配置信息
[assembly:AssemblyCompanyAttribute("ict")];                             //公司名称
[assembly:AssemblyProductAttribute("Firstapp")];                        //产品名称
[assembly:AssemblyCopyrightAttribute("Copyright (c) ict 2009")];        //版权信息
[assembly:AssemblyTrademarkAttribute("")];
[assembly:AssemblyCultureAttribute("")];                                //文化信息
//程序集的版本信息
[assembly:AssemblyVersionAttribute("1.0.*")];
[assembly:ComVisible(false)];
[assembly:CLSCompliantAttribute(true)];
[assembly:SecurityPermission(SecurityAction::RequestMinimum, UnmanagedCode = true)];
```

由此可知，文件中包含大量属性，这些属性用方括号内的文本标识，大部分默认属性都与版本控制和程序集信息有关。Visual Studio .NET 自动在 AssemblyInfo.cpp 文件中插入这些值，并在默认时取值为空，可以根据特定的应用程序修改这些值。

19.2　编写访问.NET 的 MFC 程序

本地代码与托管代码的接口有两种方式：COM Interop 和结合使用托管扩展与 IJW（It Just Works）。在 MFC 中，使用.NET 框架提供的众多 BCL 类可以大大提高编程效率。对于 VS2010 编译器，其默认设置是不支持 MFC 和.NET 混合编程的，下面通过创建一个实例来具体讲解如何修改编译器设置，使其支持混合编程。

（1）新建一个基于对话框的 MFC 应用程序，保持向导提供的其他默认设置。

（2）选择 Project|Properties 命令，在工程属性对话框中，将 Common Language Runtime Support 选项设置为 Common Language Runtime Support (/clr)，表示希望在 MFC 应用程序中获取 CLR 支持，如图 19.1 所示。这样就可以在程序中使用.NET 提供的众多类库。

（3）这时如果直接编译程序会报错，这是因为默认工程属性的有些设置与 CLR 支持冲突。下面分别指出发生冲突的部分，并将它们改为与 CLR 兼容的设置。首先在 C/C++下的 General 项中，将 Debug Information Format 选项改为 Program Database (/Zi)，如图 19.2 所示。然后在 C/C++下的 Code Generation 项中，将 Enable Minimal Rebuild 选项设置为 No(/Gm-)。将 Basic Runtime Check 选项设置为 Default，如图 19.3 所示。

（4）这些属性设置好后，就可以在 MFC 程序中使用.NET 框架提供的类库了。与.NET 编程一样，

在使用类之前需要使用 using 语句将该类所在的命名空间包含进来，最常用的就是 System 命名空间。首先在对话框类的头文件的开头加入如下语句。

```
using namespace System;
```

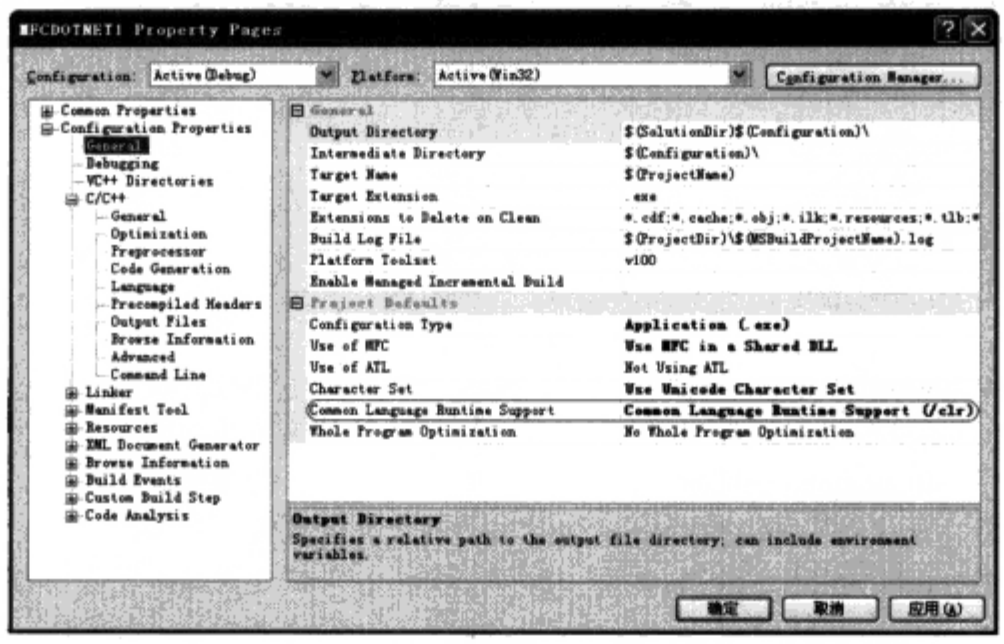


图 19.1　设置 CLR 支持

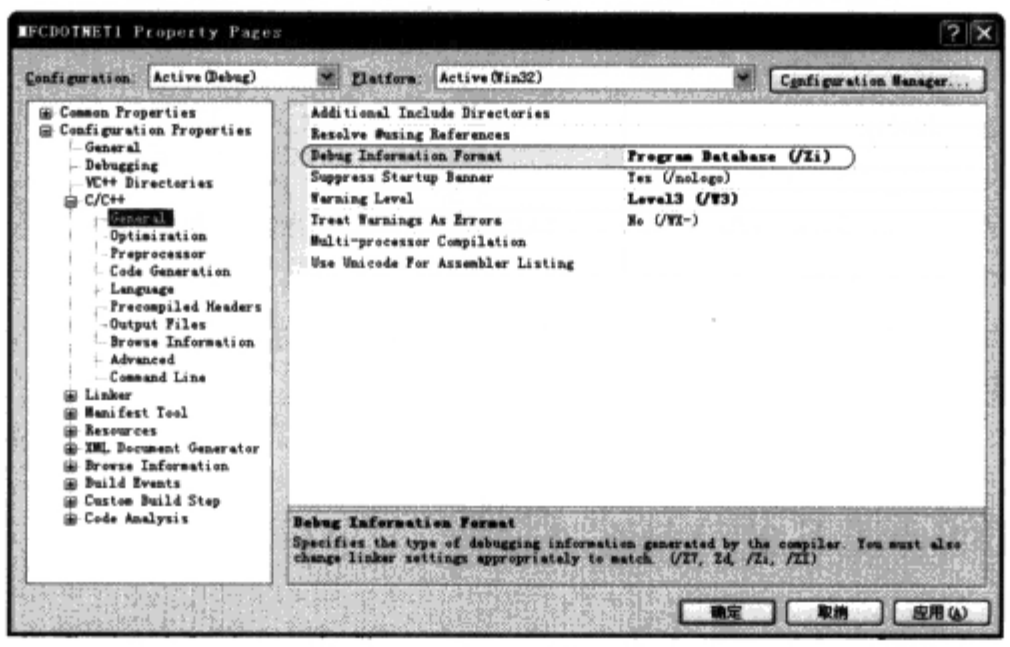


图 19.2　设置调试信息格式

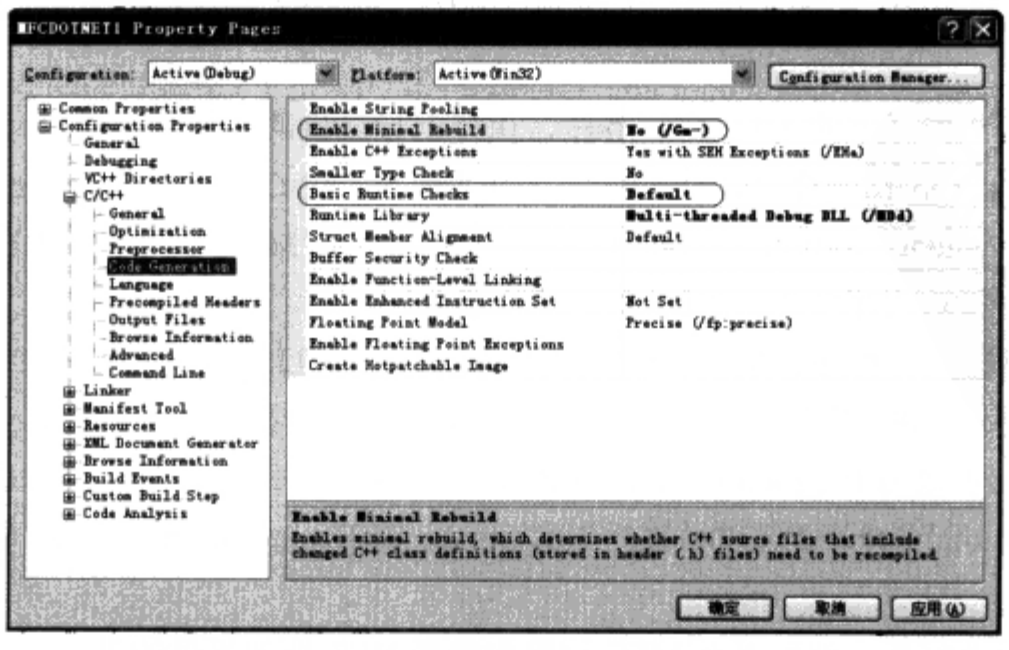


图 19.3　设置运行时检测

然后在 Resource View 下设置如图 19.4 所示的对话框布局。

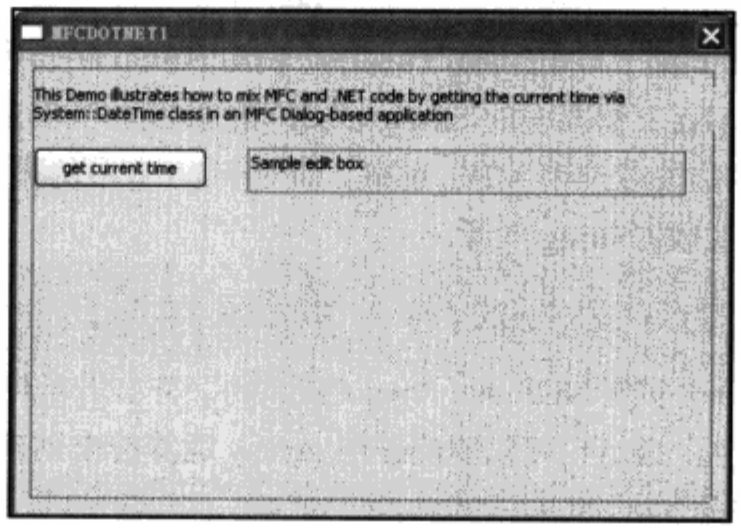


图 19.4　程序对话框布局

下面为 get current time 按钮添加单击事件处理函数，如例 19.2 所示。

例 19.2

```
void CMFCDOTNET1Dlg::OnBnCLRckedButtonDate()
{
    // TODO: Add your control notification handler code here
    this->SetDlgItemTextW(IDC_EDIT_DATE,(CString)(DateTime::Now.ToString()));
}
```

这里使用 System 命名空间内的 DateTime 类中的 Now 属性来获取当前的系统时间，然后调用 ToString()方法将其转化为.NET 框架下的 String 类，最后强制转化为 MFC 中的 CString 类，并显示在文本框中。

这个程序的运行结果如图 19.5 所示。

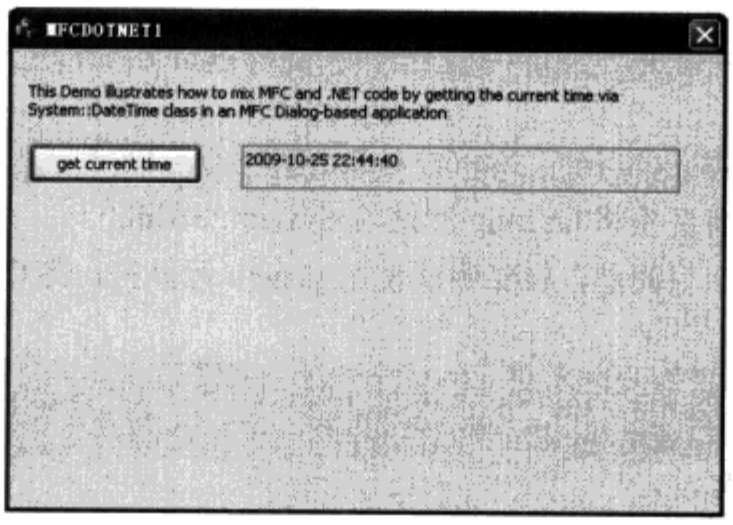


图 19.5　程序运行结果

19.3　混合模式编程问题

在 MFC 中混合使用托管代码并不是这么直接，因为很多时候可能会出现一些以前很少考虑甚至从未遇到过的错误，最常见就是在 MFC 的字符串 CString 和.NET 下使用的 String 类之间的转换问题。下面对字符串间的转换进行讨论。

在不同的开发语言间进行字符串转换是一个困难的过程，尤其是当在一种平台中使用 ASCII 字符

串而在另一种平台中使用 Unicode 字符串，更是为字符串的转换增加麻烦。

在 VS2010 的 MFC 程序中，默认都是使用 Unicode 字符组成的宽字符串，当然也可以选择使用 ASCII 字符。本节使用 Unicode 宽字符串的 MFC 程序，使用 ASCII 字符串的处理方法与之类似。实际上通过 IJW，MFC CString 和.NET 框架中的 String 类之间的转换相当容易。

这里约定对于 MFC 中的类变量使用 MFC_前缀，而对于.NET 中的类变量使用 NET_前缀。下面的代码演示了如何将 String^转换成 CString 对象：

```
String^ NET_str = L"hello";
CString MFC_str = NET_str;
```

这里首先初始化一个 String^实例 NET_str，"hello"前面的字符 L 表示使用宽字符串。这里的 NET_str 是托管对象，分配在托管堆上，当然也可以明确使用 gcnew 运算符来创建一个字符串。

```
String^ NET_str = gcnew String(L"hello");
```

初始化好 NET_str 对象之后，可以直接将其赋值给一个 CString 变量，而不需要任何附加操作，这是因为在现在的 MFC 版本中，类 CString 的构造函数含有一个接收 System::String 对象的重载，正是这个重载函数使得字符串的转换变得相当容易。

注意在这里变量 NET_str 是分配在托管堆上，而变量 MFC_str 是分配在栈上，但是在转换时不必考虑这些细节，这也体现了 IJW（It Just Works，它就是可行的）的涵义。

但是在将 CString 变量转换成 String^时，如果使用直接赋值的方式进行转换，编译器就会报错：

```
CString MFC_str = _T("hello");
String^ NET_str = MFC_str;
```

编译时将出现如下错误信息：

```
error C2440: 'initializing' : cannot convert from 'CString' to 'System::String^'
        No user-defined-conversion operator available, or
        No user-defined-conversion operator available that can perform this
         conversion, or the operator cannot be called
```

而使用强制类型转换：

```
String^ NET_str =(String^) MFC_str;
```

也报出类似的错误：

```
error C2440: 'type cast' : cannot convert from 'CString' to 'System::String^'
```

这里的报错信息表明不能直接将'CString'转换为 System::String^。

其实解决这个问题的方法很简单，只要明确使用 gcnew 运算符，将 CString 对象作为参数即可，例如下面的代码：

```
CString MFC_str = _T("hello");
String^ NET_str = gcnew String(MFC_str);
```

就不会有任何问题。

更进一步，在.NET 框架中，String 类是托管类，只能使用句柄访问，并将对象创建在托管堆上。但是在 MFC 中，有两种不同的创建 CString 对象的方式：直接在栈上创建或在堆上创建，并用一个 CString 指针指向它。那么 String^和 CString 指针能不能直接相互转换呢？答案是否定的，下面的代码在编译时都将报错：

```
//将 String^直接装换为 CString*
String^ NET_str = gcnew String(L"hello");
CString* MFC_p_str = NET_str;
```

报错信息为：

```
error C2440: 'initializing' : cannot convert from 'System::String ^' to 'CString *'
//将 CString 指针直接转换为 String^
```

```
CString* MFC_p_str = new CStirng(_T("hello"));
String^ NET_str = MFC_p_str;
```

报错信息为：

```
error C2440: 'initializing' : cannot convert from 'CString *' to 'System::String^'
//将 CString 指针作为参数创建 Stirng^
CString* MFC_p_str = new CString(_T("hello"));
String^ NET_str = gcnew String(MFC_p_str);
```

报错信息为：

```
error C2664: 'System::String::String(const wchar_t *)' : cannot convert parameter 1 from 'CString *' to 'const wchar_t *'
```

解决方法很简单，如果想将 CString 指针转换为 String^，只需先使用“*”运算符取回字符串内容，再作为 String 构造方法的参数即可：

```
CString* MFC_p_str = new CString(_T("hello"));
String^ NET_str = gcnew String(*MFC_p_str);
```

而如果想将 String^转换为 CString 指针，只需要使用 new 运算符，将 String^作为参数，明确在堆上分配空间即可：

```
String^ NET_str = gcnew String(L"hello");
CString* MFC_p_str = new CString(NET_str);
```

19.4　运用.NET 类型

上面介绍了在混合模式编程中经常遇到的字符串转换问题，下面讲解使用.NET 类型的基本知识，讲解如何定义和使用引用类型与值类型，装箱与拆箱的概念，使用托管和非托管指针，处理.NET 数组，.NET 对象的正确析构。

19.4.1　定义和使用托管类型

从前面的介绍中可知，通过设置编译器的托管扩展部分能够编写声明和使用 BCL 类的 MFC 应用程序。因此托管扩展要支持两种不同的托管类型：引用类型和值类型。为了使 MFC 应用程序与.NET 类型进行交互，必须熟悉它们之间的区别，尤其是定义和使用上的区别。

1．在 MFC 中定义引用类型

引用类型分配在托管堆上，只能通过 gcnew 运算符创建，它们的生命期由.NET 框架的垃圾回收器（GC）控制，用户不必显式使用 delete 运算符来删除对象。

与.NET 中的 C++/CLR 编程一样，要定义引用类型，需要在类名的前面加上关键字 ref 作为前缀，而且所有定义的引用类型都将派生于 BCL 中的 System::Object 类（即使是在 MFC 程序中定义也是这样），而且不支持多重继承，但是允许同时实现多个接口。

下面的代码演示了如何在 MFC 程序中定义一个引用类型，如例 19.3 所示。

例 19.3

```
ref class Employe{
protected:
    int eid;
    String name;
    CString* p_str;
public:
```

```
        //通过属性来访问
        property int EmlpoyeID{
            int get(){
                return eid;
            }
            void set(int value){
                eid = value;
            }
        };
        property String^ EmployeName{
            String^ get(){
                return name;
            }
            void set(String value){
                name = value;
            }
        };
public:
    Employe(){
        eid = 0;
        name = nullptr;
    }
};
```

这里定义了一个表示雇员的引用类 Employe，注意需在类名前加上 ref 前缀。在 MFC 程序中使用这个类与.NET 提供的其他类一样，都是通过 gcnew 运算符进行，而且只能使用句柄来引用：

```
Employe^ em = gcnew Employe;
em->EmlpoyeID = 7;
em->EmployeName = L"Bill";
```

从上面的代码可以看出，Employe 引用类型必须使用托管扩展的 gcnew 运算符来创建对象。使用托管扩展时，编译器负责识别被分配类型是本地还是托管类型，并执行对应的对象构造。因为将对象分配在托管堆上，所以不会发生内存泄露，CLR 将负责清除对象，一旦不再有代码引用它就排队等待释放。

2．在 MFC 中定义值类型

值类型通常在类名的前面加上 value 关键字作为前缀。值类型一般用于保存初级数据类型（如 int、char、bool 等）。不同于引用类型，值类型可以在栈上创建，也可以在托管堆上创建。

定义值类型的最大优点是：代码不会带来垃圾回收器在性能和资源上的开销，因此值类型比引用类型在处理简单数据方面更加高效。另外，值类型是密封的，不能从现有的值类型进一步派生自己的类型，这一点需要注意。

例 19.4 演示了如何在 MFC 程序中定义值类型。

例 19.4

```
value class Personality{
protected:
    double height;
    int age;
    bool married;
    String^ name;
```

```
    //CString hello;
public:
    property double Height{
        double get(){
            return height;
        }
        void set(double val){
            if(val>0){
                height = val;
            }
        }
    }
    property int Age{
        int get(){
            return age;
        }
        void set(int val){
            if( val>= 0 ){
                age = val;
            }
        }
    }
    property bool Married{
        bool get(){
            return married;
        }
        void set(bool val){
            married = val;
        }
    }
};
```

可见，从代码定义的角度来看，引用类型和值类型没有多大的区别，只是前缀的关键字不同而已，主要区别还是在于它们在运行时如何分配及如何被运行时管理。对于内部不含引用类型成员的值类型而言，创建对象可以有两种方式：

```
//在栈上创建值类型对象
Personality p1;
p1.Height = 175;
p1.Age = 20;
p1.Married = false;
//在托管堆上创建值类型对象
Personality^ pp = gcnew Personality;
pp->Height = 175;
pp->Age = 20;
pp->Married = false;
```

对于在栈上分配的对象，当它们超出作用域时将自动销毁，这与 MFC 中的一般栈式管理是一样的。

19.4.2　将非托管对象作为托管类的成员

在上面讨论的类中，不管是值类型还是引用类型，使用的成员要么是基本类型，要么是托管类。那么能否将其他的非托管类的对象作为托管类的成员呢？实际上只有指向非托管类的对象指针可以作

为托管对象的成员，而非托管类的对象本身是不能直接作为托管类的成员。例如下面的代码没有问题：

```
//将这个类改为值类型也是正确的
ref class vc{
protected:
//将非托管类型的指针作为成员
    CString* p_str;
};
```

而下面的代码就会报错：

```
ref class vc{
protected:
//出错信息：error C4368: cannot define 'str' as a member of managed 'vc':
//mixed types are not supported
    CString str;
};
```

如果试图将托管类型作为非托管类的成员，那是根本不可行的。例如下面的代码将出错：

```
class native_class{
protected:
//出错信息：error C3265: cannot declare a managed 'str' in an unmanaged
//'native_class'
    String^ str;
};
```

19.4.3 装箱和拆箱

作为所有值类型的基类，Syatem::Object 类能够处理对基本类型（如 int、double、bool 等）值的装箱（Boxing）和拆箱（Unboxing）。装箱是指将分配在栈上的实例转换为放在托管堆上的一个引用，而拆箱是相反的过程。具有装箱和拆箱的功能意味着基本类型的值将可以像托管对象那样使用，而且可以直接利用算术运算符对其进行运算，而节省了作为对象的开销。基本值类型在通常情况下分配在栈上，只有当需要向对象一样操作时（如调用 Console::WriteLine()）才会被装箱成 System::Object^，此时运行时会在托管堆上新建一个对象，包含原来的值类型信息。装箱的过程由编译器自动进行，用户不必考虑。

当然，用户也可以进行明确的装箱操作，很简单，只要直接将基本类型的变量赋值给一个 Object^ 即可，例如：

```
double PI = 3.1415926;
Object^ boxedValue = PI;
```

第二个语句将一个基本值类型变量装箱，并通过句柄 boxedValue 引用。

对于基本类型，也可以明确使用 gcnew 运算符，在托管堆上创建对象，例如：

```
double^ PI = gcnew   double(3.14159);
```

这个语句明确将值 3.14159 装箱，并可通过句柄 PI 引用。

对于已经被装箱的基本值类型，可以通过“*”运算符对其拆箱：

```
double PIPI = (*PI)*(*PI);
//而直接对装箱的句柄进行运算将出错：
// error C2676: binary '*' : 'System::Double ^' does not define this operator or a
// conversion to a type acceptable to the predefined operator
double PIPI = PI*PI;
```

还可以通过 safe_cast<>运算符来进行拆箱：

```
double native_PI = safe_cast<double>(PI);
```

19.4.4　指针

非托管 C++数据类型的对象可以存放在两个地方：栈或堆上。当在非托管代码中使用指针时，意味着将处理那些在堆上的数据。但是在 C++中，指针可以指向任何地方，在混合模式下编程时，指针也可能会指向托管堆中的某个数据，或者指向 Windows 操作系统内核空间的某个地方。当然，这些情况导致的后果就是运行出错。

有一点需要明确，在非托管 C++ 中，指针内存放的实质上就是一个内存地址，如果不去改变它，在它的生命期内，这个地址不会发生变化。

在非托管 C++中，一个指针可以通过多种方式创建。

- 直接将某个地址放到指针变量中（需要用户来保证地址有意义）。
- 通过对其他指针的算术运算得到目标地址（如对数组元素的访问）。
- 通过“&”运算符获取某个已存在对象的地址。
- 使用 new 运算符动态创建对象，并返回它的地址存入指针变量。

指针最让人诟病的就是它的安全问题，如果使用不当，将非常容易出错。所以在托管代码中，指针的概念已经被抛弃，取而代之的是引用或句柄。但是指针还是非常高效的。实际上 C++/CLR 并没有将指针完全丢弃，而是做了折中：既希望获取它的高效，又尽可能地减少安全隐患。为此 C++/CLR 提供了一种称为内部指针（Interior Pointer）的概念，内部指针是专用于指向托管堆上对象的指针。

有一点需要说明，托管堆上的对象在其生命期内是可以移动的（例如垃圾回收器为了回收空间，会将一些碎片集中成连续区域并回收，这期间就需要移动托管堆上对象），这也就意味着托管堆上的对象没有固定的地址，所以是绝对不能用一个普通的指针去指向它的。客户不可能去跟踪对象的移动，并相应调整指针所指向的地址。这个工作只有交给运行时来完成，这也就是内部指针之所以能正确工作的原因：一旦将一个托管对象的“地址”赋值给内部指针，那么当垃圾回收器移动对象时，运行时将自动更新内部指针的值使得它总是指向正确的地方。可以看出内部指针的概念有点像 Windows 编程中的句柄的概念，但是不像句柄每次访问对象之前都需要锁定对象，访问结束还要将对象解锁，可以直接对内部指针进行操作，不用操心地址变换等问题，就像在使用一般的指针一样方便。既然涉及地址的变化，对内部指针的操作还是有些限制的，不能改变内部指针的值或者通过其他内部指针的算术运算来得到内部指针的值。

声明一个内部指针的语法如下：

```
interior_ptr<托管类型引用> 指针名 = &托管对象;
```

例如，现在已经新创建了一个 String 对象 str，那么定义一个指向该对象的内部指针可用如下代码：

```
Interior_ptr<String^>ip = &str;
```

而要访问内部指针所指向的对象，则使用*运算符，例如：

```
(*ip) = L"Hello World";
```

例 19.5 演示了内部指针的用法。

例 19.5

```
using namespace System;
ref class Point{
public:
    int x;
    int y;
};
int main(array<System::String^ > ^args)
```

```
{
    Point^ p = gcnew Point;
    p->x = 0;
    p->y = 0;
    interior_ptr<Point^>ip1 = &p;
    (*ip1)->x = 8;
    p->y = 20;
    Console::WriteLine(L"(&ip1)={0:X};p->x={1},p->y={2};(*ip1)->x={3},(*ip1)->y={4}",
        (int)&ip1,p->x,p->y,(*ip1)->x,(*ip1)->y);
    interior_ptr<int>ip2 = &p->x;
    (*ip2) = 12;
    Console::WriteLine(L"(&ip2)={0:X};p->x={1},p->y={2};(*ip1)->x={3},(*ip1)->y={4}",
        (int)&ip2,p->x,p->y,(*ip1)->x,(*ip1)->y);
Console::ReadLine();
return 0;
}
```

该程序的运行结果如图 19.6 所示。

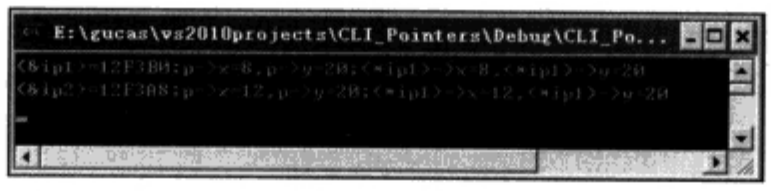


图 19.6　使用内部指针的运行结果

前面介绍过，托管堆中的对象被移动，将导致地址变化。在混合编程模式下，如果想将托管对象的句柄作为参数传递给一个非托管的函数处理，将导致出问题，因为该句柄可能在函数操作期间被移动，所以在对其处理之前，要先把它固定住（类似于 Windows 编程下的锁定句柄）。为此，.NET 框架提供了另一种类型的指针，称为固定指针（Pinning Pointer），如果将一个托管对象的地址赋给固定指针，就能阻止运行时移动对象，直到出了这个固定指针的作用域或将它赋值为 nullptr。

定义固定指针使用 pin_ptr<>关键字，其用法与内部指针基本相同。

例 19.6 所示的代码演示了内部指针和固定指针的区别。

例 19.6

```
ref class vcTest{
public:
    int i;
    vcTest(){
        i = 0;
    }
};
void modify(int * i){
    (*i) += 8;
}
//下面的代码将报错：error C2664: 'modify' : cannot convert parameter 1 from
//'CLR::interior_ptr<Type>' to 'int *'
vcTest^ test = gcnew vcTest;
interior_ptr<int>ip = &test->i;
modify(ip);
//使用固定指针没有问题
pin_ptr<int>pp = &test->i;
modify(pp);
```

这里首先定义一个引用类型 vcTest，其中包含一个 int 型成员，然后在 MFC 非托管代码中定义一个函数 modify()，这个函数接收一个 int 型指针作为参数，我们希望利用这个非托管函数来修改托管对象 vcTest 中的 i 成员的值。那么在这个函数的操作过程中，应该禁止运行时移动 vcTest 对象，如果使用内部指针由于满足不了这个要求，因此报错。而使用固定指针，确保不会移动对象，所以可以正确的调用 modify()函数。

19.4.5　在非托管代码中使用托管数组

.NET 框架在 System 命名空间中实现了一个功能强大的 Array 范型类，它提供了使用大型的任意对象类型组的简单、灵活方式。Array 对象支持单维、多维。下面分别介绍在 MFC 应用程序中如何使用托管数组。

1．单维数组

单维数组可以保存引用或值类型，在不同的场合下语法稍有区别。下面的代码演示了如何定义和使用单维数组，该数组包含 String^。要注意的是这里数组的定义以及数组内的方法和属性的使用，这在非托管代码的数组中是没有的：

```
array<String^> ^NET_arr = gcnew array<String^>{
    L"hello",L"world",L"hello",L"vs2010"};
for each(String^ item in NET_arr){
    Console::WriteLine(item);
}
for(int i=0;i<NET_arr->Length;i++){
    Console::WriteLine(NET_arr[i]);
}
NET_arr[0] = L"good";
```

对于托管数组，可以使用 foreach 语句来遍历数组内的所有元素，也可以通过访问 Length 属性来获悉数组的长度，然后访问数组元素。通过[下标]来访问数组元素，既可以获取它的值，也可以设置一个新值。

2．多维数组

在托管代码中，可以创建二维或更多维的数组，数组的最大维数是 32。在尖括号中类型的后面可以指定数组的维数，在它们的中间使用逗号分开。下面的代码演示了如何在非托管代码中创建并使用二维数组：

```
array<int,2> ^NET_multi_arr = gcnew array<int,2>(4,5);
for(int i=0;i<4;i++){
    for(int j=0;j<5;j++){
        NET_multi_arr[i,j] = i+j;
    }
}
```

这里访问元素的方式与非托管代码有些不同：数组名[下标 1，下标 2]。

3．给非托管函数传递托管数组

因为托管数组实质上是 Array 类的对象，所以可以把它们作为非托管函数的参数。下面的示例函数带有一个二维数组作为参数：

```
for ( int i = twoDArray->GetLowerBound( 0 ); i <= twoDArray->GetUpperBound( 0 ); i++ )
{
```

```
        for ( int j = twoDArray->GetLowerBound( 1 ); j <= twoDArray->GetUpperBound( 1 ); j++ )
        {
            twoDArray[i,j] = (i*j).ToString();
        }
    }
```

这个函数是类型安全的，因为它只接收二维数组，而且通过 GetLowerBound()和 GetUpperBound()方法可获取指定维数组的下限下标和上限下标。而在非托管代码中，只能通过指针来传递数组，而且需要额外的传递数组的下标作为参数：

4．从非托管函数返回托管数组

在非托管 C++中，数组通过数组的首元素指针返回给客户代码。因此不能保证类型安全，正确处理返回数组由客户负责（如确定数组的维数、每个维数的大小这些信息都需要客户自己维护）。而通过托管扩展，准确的数组类型和大小在函数原型声明中指定，因此客户端犯错的机会就更少了。下面的代码演示了如何返回一个托管数组对象：

```
array<int,2> ^getArray(){
    array<int,2>^twoDArray = gcnew array<int,2>(8,8);
    for ( int i = twoDArray->GetLowerBound( 0 ); i <= twoDArray->GetUpperBound( 0 ); i++ )
    {
        for ( int j = twoDArray->GetLowerBound( 1 ); j <= twoDArray->GetUpperBound( 1 ); j++ )
        {
            twoDArray[i,j] = i*j;
        }
    }
    return twoDArray;
}
```

可以看到返回一个数组相当直观，上面的函数返回一个二维的 int 托管数组，如果试图返回其他类型或不同维数的数组都将导致编译错误。有些读者可能会有疑问，在函数的内部创建一个局部变量 twoDArray，这是一个二维数组的引用，但是它的作用域只是在函数的内部，而现在我们将它作为返回值，就是说在函数退出后这个对象会被销毁，那么这个函数返回的数组不就是一个悬空的区域？其实不会这样，因为在托管代码中，是在托管堆上分配空间，并分配一个引用计数，只有当引用计数变为 0 后，空间才会被垃圾回收器自动回收。现在我们将它作为函数返回值，这就使这个数组区域的引用计数又加上 1，在局部变量 twoDArray 销毁后，由于引用计数还是 1，因此这个数组区域是会保留在托管堆中的。

19.5 小结

这一章主要介绍了足够的托管扩展和.NET 基础知识。为了理解托管扩展应用程序与 MFC 应用程序之间的关键区别，首先介绍一个由 VS2010 向导生成的“Hello World”CLR 控制台程序，本章还介绍了程序集清单、引用等.NET 概念。在之后还介绍了一个基本的演示程序，说明如何使用 MFC 和.NET 混合编程。

接着着重介绍了如何在 MFC 的字符串和.NET 框架下的字符串类 String 之间进行转换的问题，最后讨论 NET 类型的使用，着重介绍托管类型的创建、装箱与拆箱，如何在托管代码中使用指针以及在非托管代码中使用托管数组。

第四部分　发布 Visual C++ 2010 程序

第 20 章　Visual C++ 2010 应用程序部署

编译源代码并完成测试后，并不意味着开发过程的结束，还需要将应用程序提供给用户，也就是说应用程序必须部署到目标环境中。部署就是指把测试完毕的应用程序或者网站安装到目标系统上，并设置相应的环境使应用程序可以正常运行的过程。比较常见的是安装程序，例如安装 Visual C++应用程序、安装 Office 软件的过程都是典型的部署过程。而.NET 框架使得部署工作比以前容易得多，因为不需要注册 COM 组件、编写注册表项。

Visual Studio 2010 提供了多种不同的应用程序安装部署的项目类型。在 VS2010 中的新建工程，选择工程类型模板为 Other Project Types，在展开的子项中选择 Setup and Deployment，右边的模板列表中列出了 VS2010 提供的可实现的安装和部署类型，如图 20.1 所示。

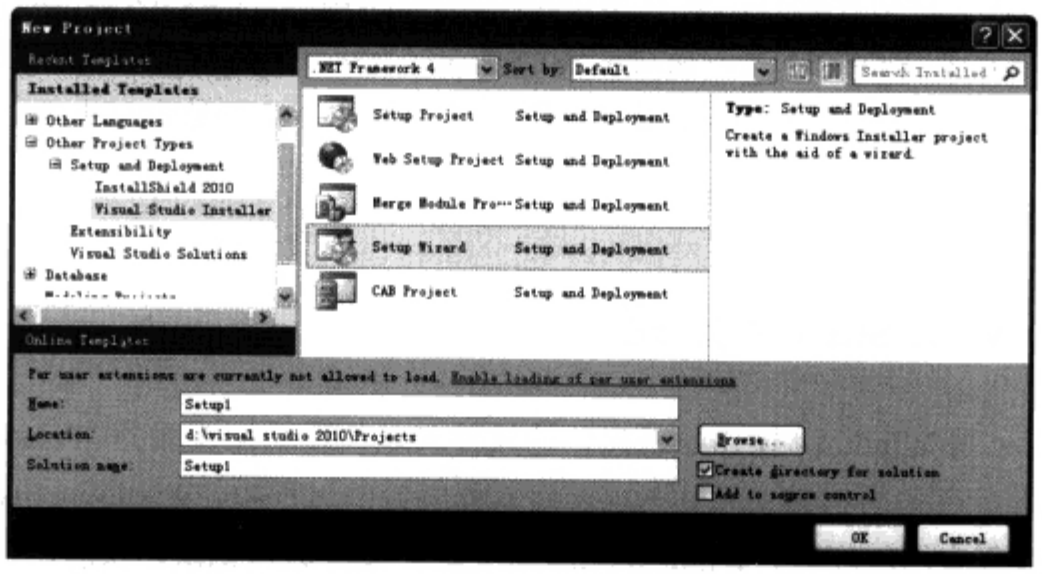


图 20.1　新建一个应用程序部署工程

可以看到，VS2010 一共提供了 5 种安装项目类型，表 20.1 列出了这些类型和说明。

表 20.1　　**安装项目类型**

项 目 类 型	说　　明
Setup Project	用于安装客户应用程序、中间层应用程序以及运行为 Windows 服务的应用程序
Web Setup Project	用于安装基于 Web 的应用程序
Merge Module Project	用于创建合并模块，这些模块可以和其他基于 Windows Installer 的安装应用程序一起使用
Setup Wizard	借助于向导的帮助来创建部署项目
CAB Project	创建 cab 文件，通过旧式的部署技术进行发布

除了这些安装项目类型外，.NET 还提供了其他几种不同的部署选项，以适用于不同的应用需求。

1．Xcopy 实用工具

在.NET 之前，安装组件需要首先将组件复制到相应目录中，然后创建适当的 Registry 项。不过，.NET 框架的主要目标就是实现 Xcopy 部署来简化部署。换句话说，要安装组件，要进行的所有操作就是将

程序集复制到客户应用程序的 bin 目录中，然后就可以使用它启动应用程序，这是因为程序集是自我包含的，元数据描述了包含在程序集中的内容，所以不需要在注册表中注册。每个程序集都要跟踪它需要执行的其他程序集。默认情况下，程序集会在当前的应用程序文件夹中查找从属文件。由于.NET 框架中的编译器将标识符或元数据嵌入到编译后的模块中，所以 CLR 可以使用该信息载入适当的程序集版本。Xcopy 部署又称为无影响安装（zero-impact install），也可以使用 gacutil 实用程序将组件安装到 Global Assembly Cache（GAC，全局程序集缓存）中。

无影响安装也可以采用不影响系统的方式卸载组件。完成该卸载组件的操作只需在多个特定目录中删除特定的文件即可。

2．Copy Web 工具

如果开发的是 Web 项目，使用 Web 站点菜单中的 Copy Web 选项就会把运行应用程序所需要的组件复制到服务器上。

3．发布 Web 站点

在发布 Web 站点时，会编译整个站点，然后复制到指定的位置。在预编译时，所有的源代码都会从最终的输出中删除，找出和处理所有的编译错误。

4．ClrckOnce 技术

ClrckOnce 可以建立自动升级的、基于 Windows 的应用程序。ClrckOnce 允许把应用程序发布到 Web 站点、文件共享、刻录到 CD 上。在对应用程序进行升级，重新生成后，开发小组可以把它们发布到相同的位置或站点上。最终用户在使用应用程序时，程序会检查是否有更新版本，如果有，就进行更新。

20.1 Windows Installer 介绍

Windows Installer 是 Windows 操作系统提供的软件安装程序和配置服务。Windows Installer Service 维护与安装的应用程序有关的信息记录。在执行部署程序包的过程中，Windows Installer 运行时会检查这些记录。在试图卸载应用程序时，Windows Installer 就会检查这些记录，以确定在删除它们之前没有其他应用程序会依赖其组件。如果找到使用组件的其他应用程序，它就不会卸载这些组件。

Windows Installer 将应用程序划分成下列 3 个层次。

（1）产品：就是用户可以安装的程序。如 Microsoft Word 就是用户可以安装的产品。

（2）功能：一项产品可包含多个功能，功能是产品功能性的最小单位。如创建 PDF 功能可以看做是 Word 的一项功能。

（3）组件：组件是多个功能之间共享的最小单位。Windows Installer 组件可以是单一文件或按照逻辑组织在一起的多个文件。它可以是 EXE、DLL 或简单文本文件，可以组建组件集合来提供功能，也可以在多项功能之间共享一个组件。组件对安装在计算机上的所有产品都是全局性的。例如，Word 的拼写检查器组件可以为所有希望执行拼写检查功能的产品全局共享。

安装数据库（installation database）是扩展名为.msi 的文件，这个文件包含了产品功能和组件的信息，还包含产品安装过程中显示的用户界面序列。通过打开.msi 文件，安装程序会从安装数据库文件中读取产品信息，确定是否已经安装该产品。如果还没有安装，就会启动产品的安装序列；如果已经安装过，就会调用如添加或删除功能、修复安装或卸载产品等功能。正因为 Windows Installer 在数据库中跟踪应用程序的安装，那么在卸载应用程序时，Windows Installer 就可以很容易地跟踪和删除已添

加的注册表设置、复制到磁盘上的文件，以及已添加到桌面和“开始”菜单中的图标。如果某个文件仍被其他应用程序所引用，卸载过程就会把它保留在磁盘上，以免导致其他应用程序中断。数据库还可以修复应用程序的安装。例如，某些注册表设置或应用程序相关的 DLL 被用户不小心删除了，那么在修复过程中通过读取上一次安装的数据库，复制该安装过程就可以很容易地恢复应用程序，而不用再从头开始重新安装整个程序，那将相当费事费时。

.NET 框架除了可以完全操控 Windows Installer 的创建过程外，还提供了以下额外的部署功能。

- 注册表（Registry）表项的读/写。
- 允许在 Windows 文件系统中创建额外目录。
- 提供自动注册 COM 组件和.NET 组件（存放到 GAC 中）的机制。
- 安装过程中从用户收集反馈信息。
- 允许设置安装程序的启动条件，如搜索文件、注册表项、Windows Installer 文件、已安装的应用程序、.NET CLR 的存在信息等。
- 安装完成后，可以运行自定义程序或脚本。

20.2　一个简单的 Windows 应用程序

在后面的内容中，将通过一个实际的示例来具体讲解怎样进行应用程序的部署。首先新建一个 Windows Forms 工程，这个程序的界面布局如图 20.2 所示。

图 20.2　程序运行界面

将 TextBox 的 ReadOnly（只读）属性设置为 True，用于输出通过调用 DLL 内一个类的静态方法生成的随机数。在当前项目下新建一个工程，用于创建自己的类库：在 Solution Explorer 视图下，右击项目名（Solution），在弹出的快捷菜单中选择 Add|New Project 命令，工程类型设置为 CLR 下的 Class Library，表示生成自己的类库，如图 20.3 所示。输入工程名后，就可以编写类库，这里输入的工程名为 AppSupport。这里只是简单地创建一个类，这个类包含一个静态方法，用于返回一个随机整数，其代码如例 20.1 所示。

例 20.1

```
using namespace System;
namespace AppSupport {
    public ref class Support
    {
        // TODO: Add your methods for this class here.
    private:
        Support(){}
    public:
        static int GetOneRandInt(){
            //获取当前系统时间作为声称随机数的种子
            Random^ rand = gcnew Random(
                System::DateTime::Now.Millisecond);
            return rand->Next();
        }
    };
}
```

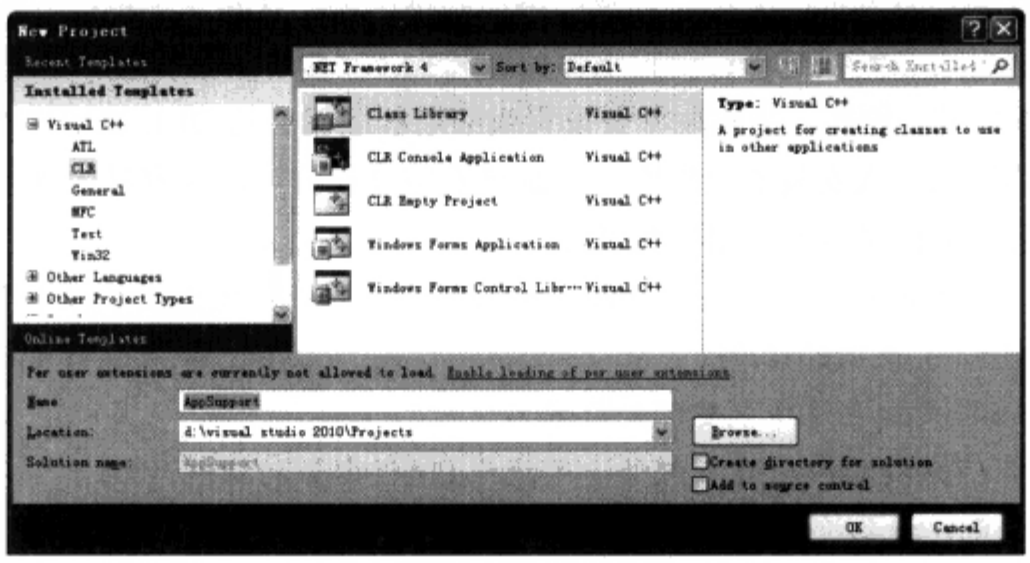


图 20.3　新建一个类库

编译这个新的工程，将生成一个类库，可以在这个工程目录的 Debug 目录下看到生成的 AppSupport.dll.intermediate.manifest 文件。在创建好类库之后，下面在之前一个工程中应用这个类库。右击工程名，在弹出的快捷菜单中选择 Reference 命令，在 Property Page 对话框的左边部分中选择 Framework and References 项，中间的部分会显示本工程应用的命名空间，如图 20.4 所示。

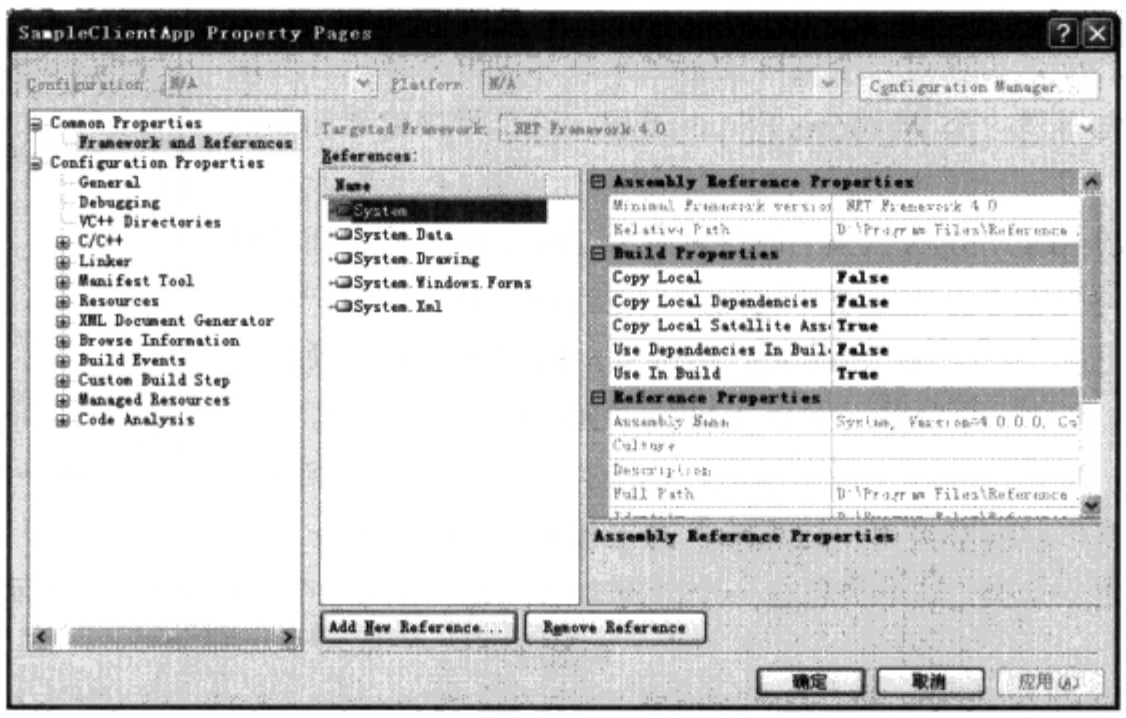


图 20.4　设置工程的引用

单击 Add New Reference 按钮，在弹出的对话框中选择 Project 选项卡，在这里选中创建的类库工程，这样就可以引用这个类库了，如图 20.5 所示。

然后为工程界面内的 CLRck Me 按钮添加单击事件处理方法：

```
private: System::Void button1_CLRck(System::Object^ sender,
                System::EventArgs^ e) {
    this->textBox1->Text =
        AppSupport::Support::GetOneRandInt().ToString();
}
```

这里使用了类库里 AppSupport 命名空间中的 Support 类提供的 GetOneRandInt()静态方法，这个静态方法返回一个随机整数，然后将这个随机整数显示到文本框中。

在编译这个工程之前，需要设置这个项目内两个工程的依赖关系。右击工程名，在弹出的快捷菜单中选择 Project Dependencies 项，在下面的 Depends on 项中，选中类库工程（这里为 AppSupport 工

程），因为这里引用了这个类库中的方法，如图 20.6 所示。

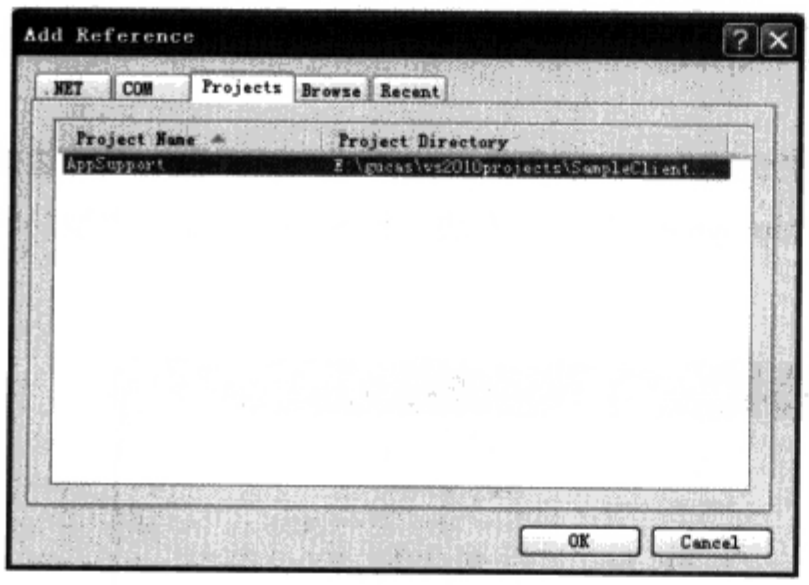


图 20.5　添加引用

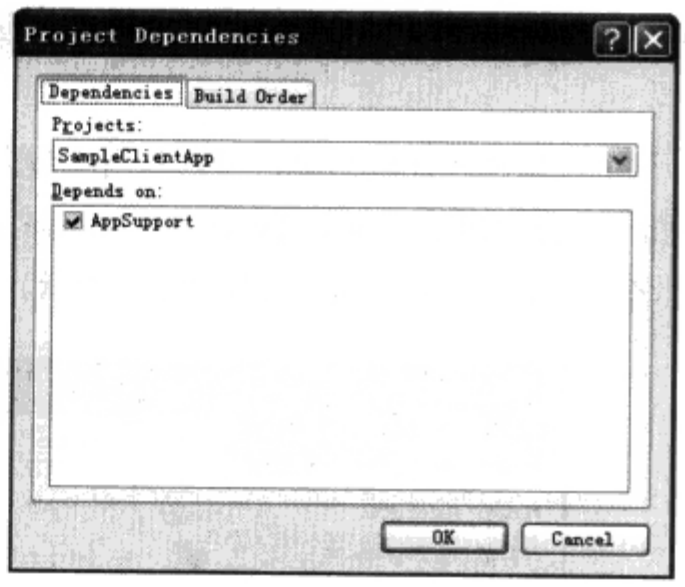


图 20.6　设置工程的依赖关系

设置完毕后，就可以编译并运行这个项目，图 20.7 所示为运行结果。

这个项目调试完毕，下面就可以对这个项目进行部署。在部署项目之前，需要生成 Release 版本，可以选择 Project |Properties 命令，或按【Alt+F7】组合键，在弹出的对话框中，单击右上角的 Configuration Manager 按钮，如图 20.8 所示。

图 20.7　最终程序运行结果

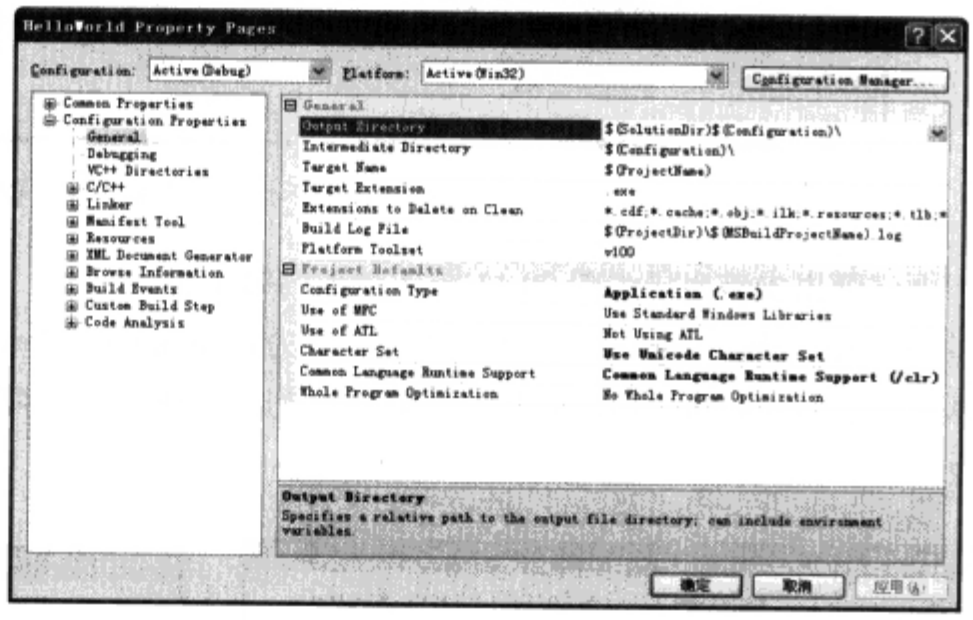


图 20.8　工程配置管理器

将两个工程的 Configuration 都设置成 Release，如图 20.9 所示。再重新编译这个项目，这样就生成了工程的 Release 版本。

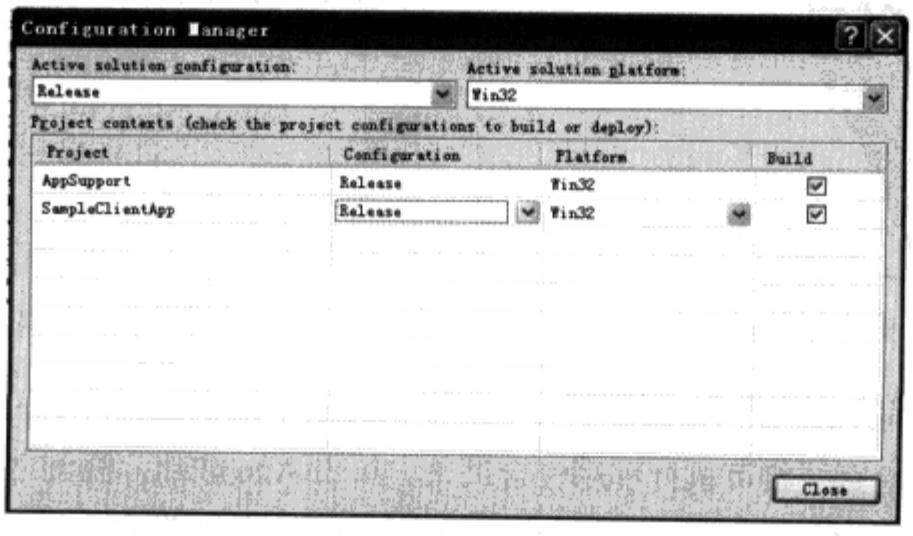


图 20.9　配置工程生成 Release 版本

20.3 使用安装向导快速创建安装包

这个应用程序比较简单，可以利用安装向导来快速生成安装包。首先新建一个工程，在左边的工程模板中选择 Other Project Types 下的 Setup and Deployment 项，在右边选择 Setup Wizard 类型，输入工程名后，单击 OK 按钮，如图 20.10 所示。

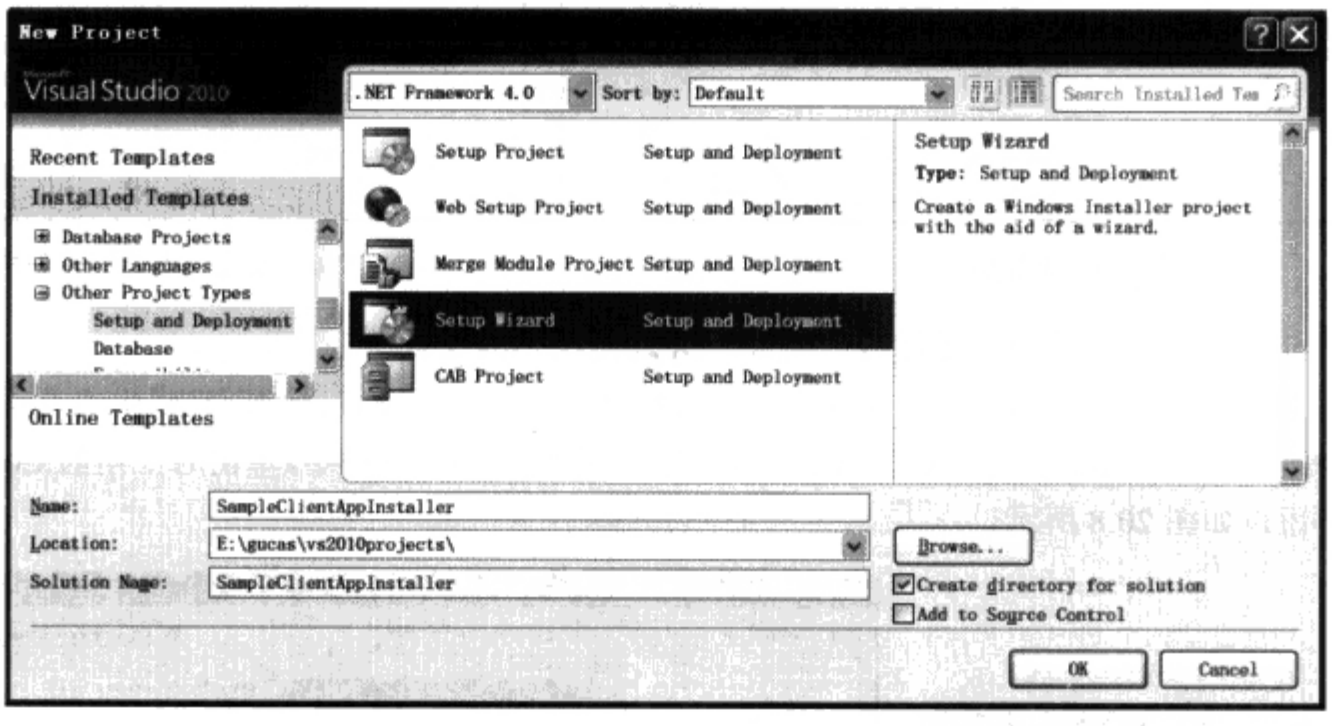


图 20.10　新建一个安装向导工程

下面将弹出安装向导，图 20.11 是 4 步中的第 1 步。

单击 Next 按钮，选择安装文件的类型，因为是 Windows Forms 应用程序，所以选择 Create a setup for a Windows application 项，而其他选项分别表示 Web 应用程序、Merge Module Project 或 CAB file，如图 20.12 所示。

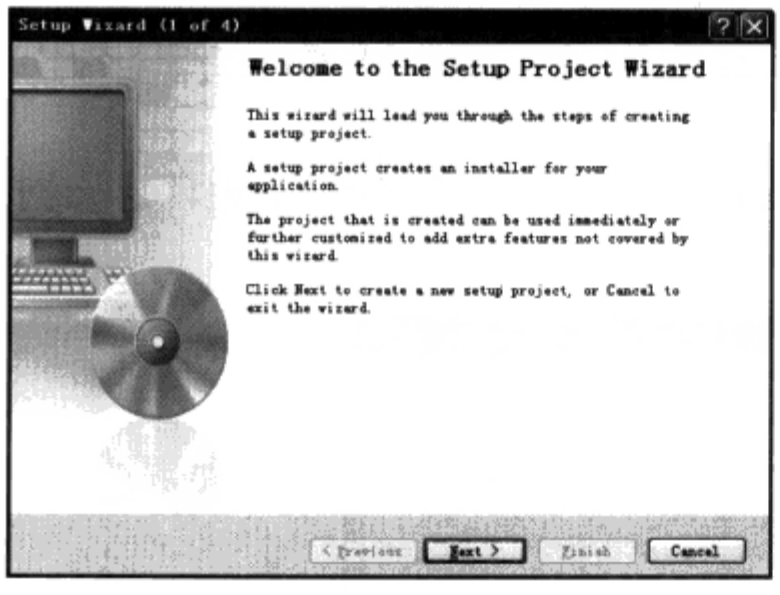


图 20.11　安装向导第 1 步

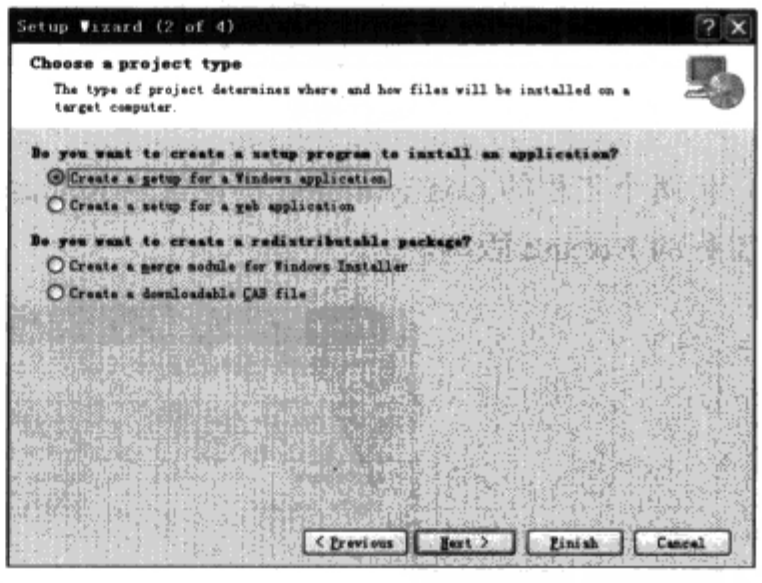


图 20.12　安装向导第 2 步

向导的第 3 步要求输入是否包含额外的文件，如 ReadMe 或应用程序的说明文档等，由于这个程序比较简单，可以在这里直接将可执行程序包含进来。单击 Add 按钮，找到之前生成的 Release 版本，只需将可执行文件加进来即可，如图 20.13 所示。程序的运行需要类库，但是并不需要将它显式地加

进来，这是因为程序集是自包含的，在程序集的元数据内包含了本程序集所依赖（或引用）的其他程序集，而.NET 框架的向导正是利用这个信息将该程序集所依赖的所有程序集都自动加进来，这可以在最后生成的工程视图中看到。

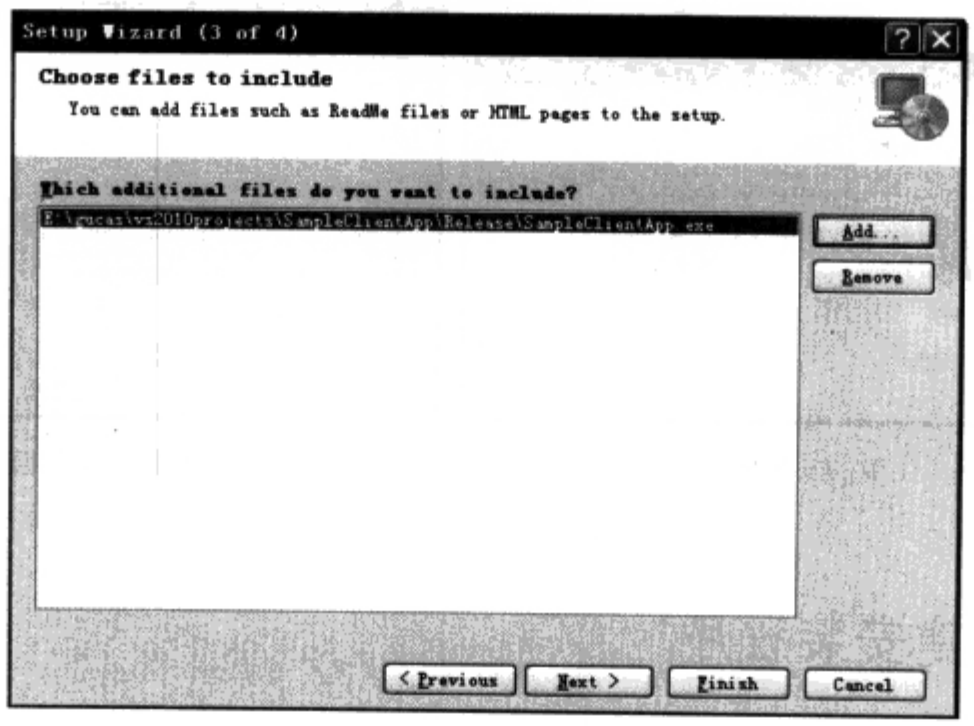


图 20.13　安装向导第 3 步

向导的第 4 步将列出前面的设置和包含进来的文件信息，如图 20.14 所示。

单击 Finish 按钮，就完成了向导，这时在工程的 Solution Explorer 视图下，可以看到向导在 Detected Dependencies 目录下自动将 AppSupport.DLL 加进来，如图 20.15 所示。

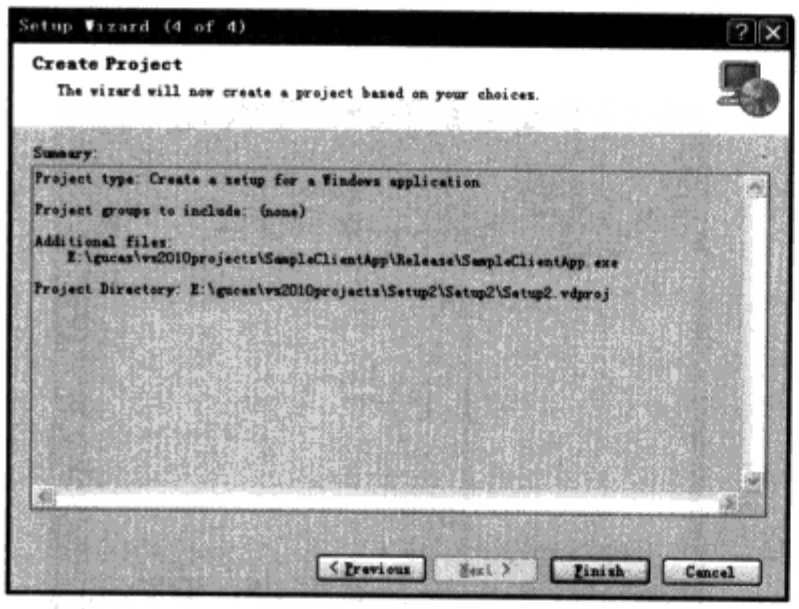


图 20.14　安装向导第 4 步

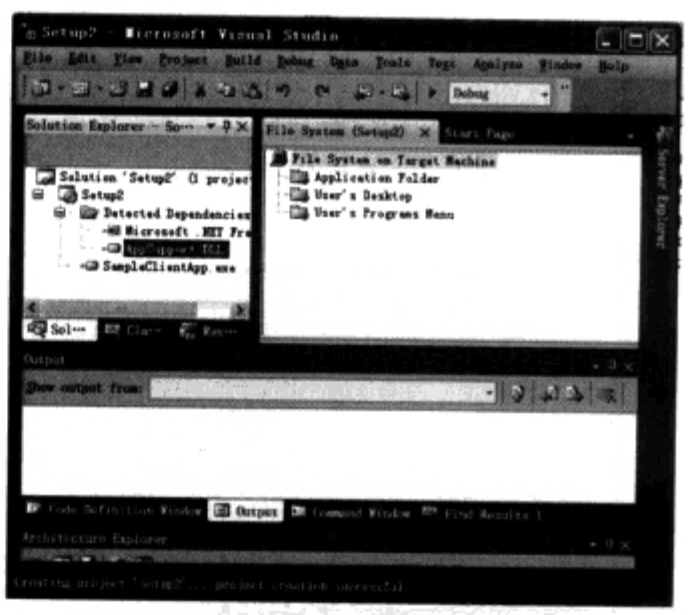


图 20.15　工程的 Solution Explorer 视图

向导完成之后，就可以直接编译这个工程了。如果要发布 Release 版本，同样利用 Project|Properties 命令下设置 Configuration 为 Release。这样在工程的 Release 文件夹下，就会生成两个文件：SampleCLRentAppInstaller.msi 文件和 setup 文件，如图 20.16 所示。

双击这两个文件中的任何一个都可以启动安装程序，开始运行安装向导，一共有 4 步，可以设置安装路径及可用的用户（只对自己可用或对所有用户可用）。安装向导的运行界面如图 20.17 所示。

WARNING 处提示在本地机器上还没有安装过该应用程序，并提示安装过程。而如果已经在本机上安装过这个应用程序，那么 Installer 将会通过查找数据库信息，这时的运行结果就是提示修复或卸载应用程序，如图 20.18 所示。

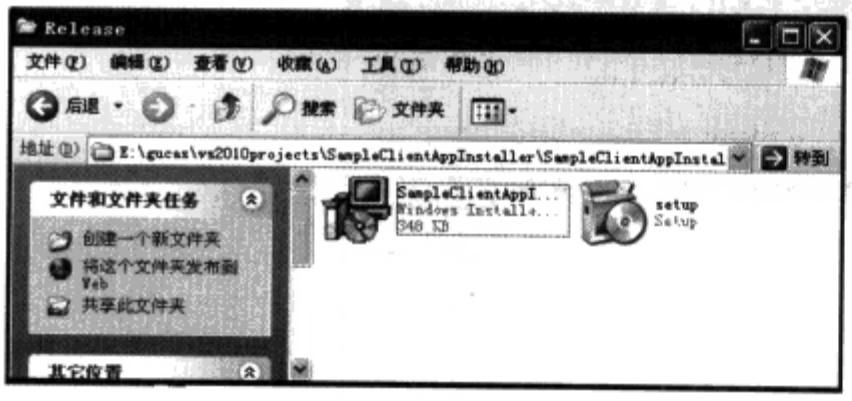


图 20.16　生成的安装文件

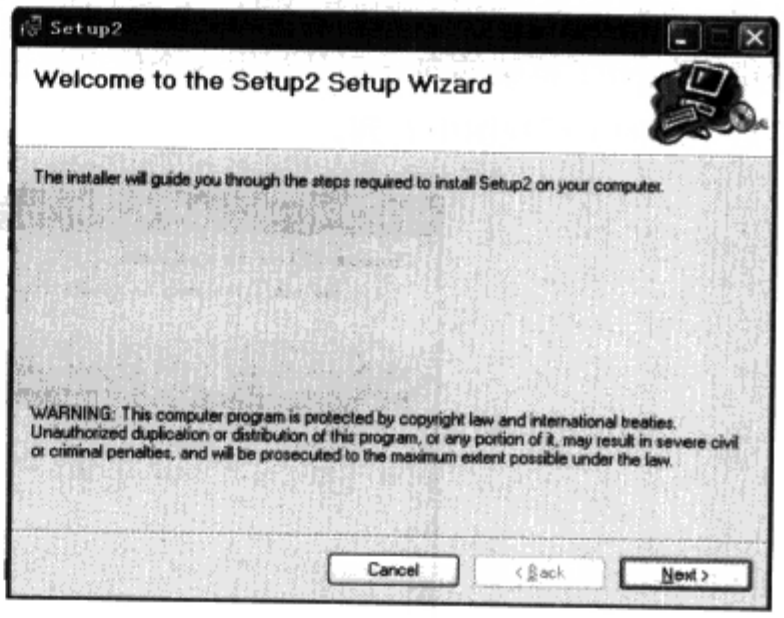


图 20.17　安装向导运行界面

20.4　手动创建安装程序

使用向导可以很方便地创建简单的应用程序安装包，但是手动创建安装程序则具有更大的灵活性，也可以先使用向导进行基本的设置，再设置完成后继续手工进行编程。

新建一个工程，在工程类型中选择 Setup Project 类型，输入工程名后，在 Solution Exploer 视图下选中工程名，在 Properties 窗口中就可以看到一组属性，如图 20.19 所示。这些属性将在应用程序的安装过程中显示，其中一些属性还会显示在“添加/删除程序”窗口中。

表 20.2 列出了安装工程的主要属性和说明。

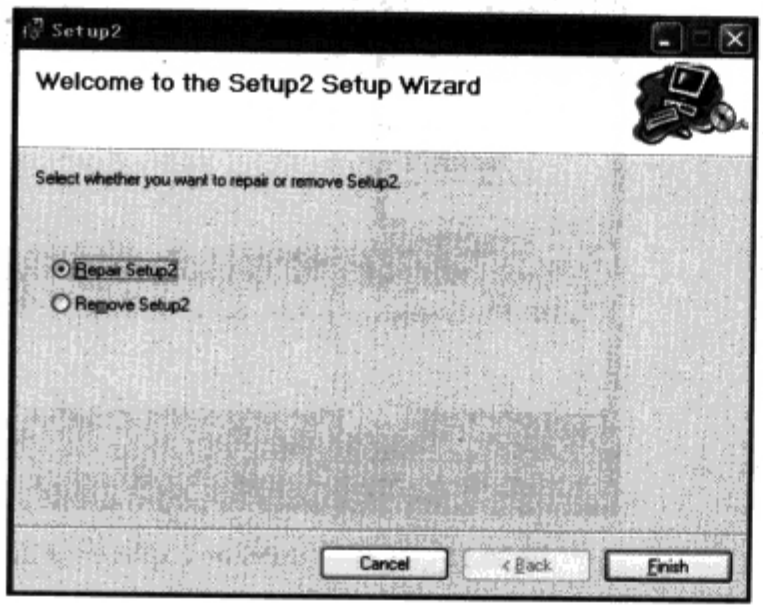


图 20.18　程序已安装的运行结果

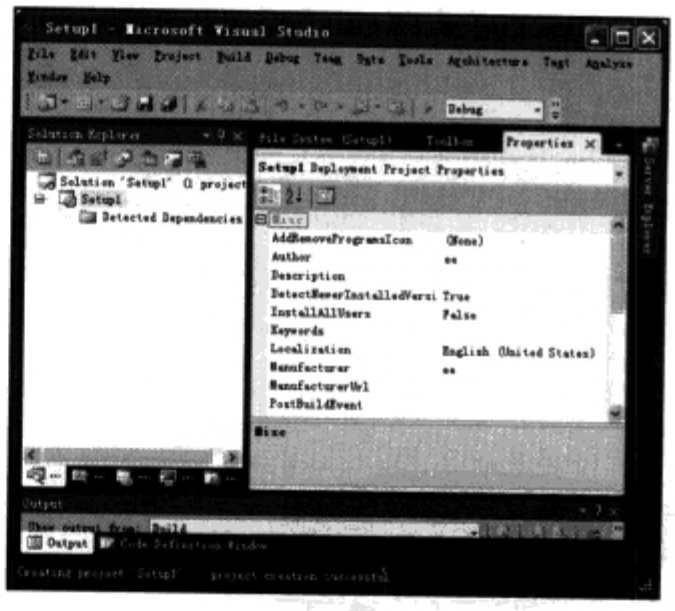


图 20.19　安装工程及其属性

表 20.2　　安装工程的主要属性

项 目 属 性	说　　明
AddRemoveProgramsInco	在目标计算机的“添加/删除程序”窗口中指定要显示的图标
Author	允许指定应用程序作者。这个属性通常与 Manufacturer 相同，它显示在 MSI 软件包的 Properties 对话框中的 Summary 选项卡中，以及“添加/删除程序”窗口的 SupportInfo 页面上的 Contact 字段中
Description	允许指定安装过程中显示的说明，这是一个形式自由的文本字段，描述了要安装的应用程序或组件

续表

项 目 属 性	说　明
DetectNewerInstalledVersion	这是一个布尔值，指定在安装过程中是否检查更新的应用程序版本。如果设置为 True，则检测到更新版本时，将停止安装过程
InstallAllUsers	这是一个布尔值，指定是否为目标计算机上的所有用户安装应用程序。如果设置为 False，就只有当前用户能访问应用程序
Keywords	指定可以用于在目标计算机上搜索安装程序的关键字，可用于在目标计算机上搜索 MSI 文件
Localization	用于字符串资源和注册设置的地域
Manufacturer	开发应用程序或组件的公司名称，一般与 Author 属性相同
ManufacturerURL	一个 Web 站点的 URL，该站点与要安装的应用程序或组件有关
PostBuildEvent	在构建结束后执行的命令
PreBuildEvent	在构建开始前执行的命令
ProductCode	应用程序或组件的惟一的标识符 GUID。Windows Installer 使用该属性标识应用程序的升级或安装
ProductName	指定应用程序的名称，在“添加/删除程序”窗口中用做应用程序的描述，还用做默认安装路径的一部分：C:\Program Files\ Manufacturer \ProductName
RemovePreviousVersions	一个布尔值，如果设置为 True，就检查计算机上是否安装了应用程序的以前版本。如果是，就调用以前版本的卸载功能，之后继续安装。这个属性使用 ProductCode 和 UpgradeCode 确定是否卸载。UpgradeCode 应相同，而 ProductCode 应不同
RunPostBuildEvent	运行 PostBuildEvent 的时间，其选项有 On successful build 或 Always
SearchPath	一个字符串，表示从属程序集、文件或合并模块的搜索路径。在开发机器上建立安装软件包时使用该属性
Subject	与应用程序相关的其他信息
SupportPhone	为应用程序或组件提供支持的电话号码
SupportURL	为应用程序或组件提供支持的 Web URL
TargetPlatform	支持 Windows 的 32 或 64 位版本
Title	安装程序的标题
UpgradeCode	一个字符串 GUID，表示同一个应用程序的不同版本共享的标识符
Version	安装程序、CAB 文件或合并模块的版本号。注意它不是要安装的应用程序的版本号

在了解了这些属性后，下面介绍如何将前面的同一个应用程序加载进来，并进行相关的属性设置，以改变一些安装选项。右击工程名，在弹出的快捷菜单中选择 Add 命令，这时会出现 4 个命令，这里选择 Assembly 命令，如图 20.20 所示。4 个命令的具体含义如下。

- Project Output ： 用于向安装包添加工程输出组。
- File: 用于向安装项目添加应用程序的 Readme 文件、帮助文档等并非编译必需的文件。
- Merge Module: 单独创建一个合并模块（merge module）。
- Assembly: 选择应用程序所在的程序集，这是安装包的主体。

在弹出的对话框中，选中应用程序的 Release 版本，同样.NET 框架会自动将依赖的类库 AppSupport.DLL 加载进来。

将应用程序等加载进来以后，就可以设置工程属性了。除了工程的基本属性外，选择 Project|Properties 命令，在弹出的对话框中设置生成的安装包的一些属性，如图 20.21 所示。这些属性的说明如下。

- Output file name: 指定工程编译后生成的安装包的名字，为 MSI 或 MSM 类型的文件。
- Package flies: 指定工程内包含的所有文件的打包方式，提供的选项如下。

- As loose uncompressed files，以松散的非压缩形式，这样所有的部署文件将被存放在与 MSI 文件的相同目录下。
- In setup file: 所有的部署文件将被打包到 MSI 文件中，这是默认设置。
- In cabinet file（s）：部署文件将被放入一个或多个 CAB 文件中，放在相同目录下。如果选中该项，下面的 CAB Size 就会有效。

• Compression: 设置部署文件的压缩类型，可选的类型如下。

- Optimized for speed，侧重压缩解压速度，安装时速度更快但是文件较大，这是默认设置。
- Optimized for size，更小，但是安装时速度较慢。

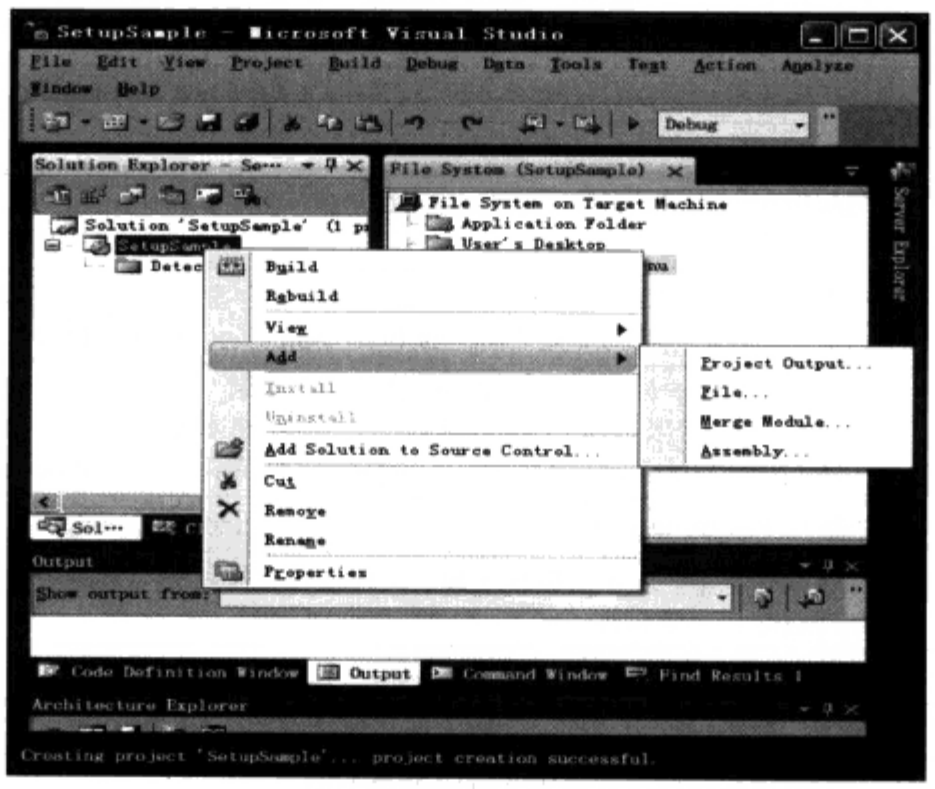


图 20.20　加载应用程序

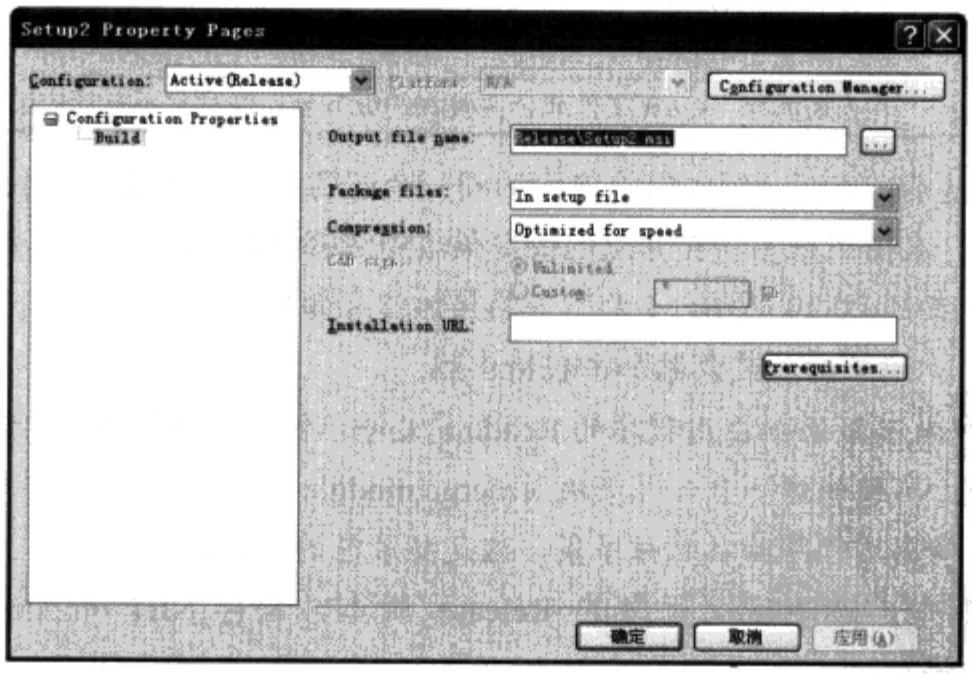


图 20.21　设置安装包属性

- None，不进行压缩。

• CAB size: 如果选择 Unlimited，将只创建一个 cabinet 文件，而选择 Custom 则由用户指定各个 CAB 文件的最大尺寸。

设置好这些属性之后就可以编译工程，同样也可以生成 Release 版本。

20.5　使用安装编辑器

由于 Visual Studio 2010 中的部署项目非常灵活，因此可以很方便地指定在何处如何部署解决方案。安装程序要完成的大量工作就是将文件复制到正确的位置。但是很多应用程序可能在安装时都需要包含 Registry 配置选项、检查安装条件等。能够定制安装程序的用户界面对于需要这些功能用户来说是很有用的，为此 Visual Studio 提供了如下安装编辑器。

- File System 编辑器：常用于将文件和快捷方式（如“开始”菜单内的菜单项）添加到安装包中。
- Registry 编辑器：允许对安装应用程序的目标计算机上的注册表项进行操作。
- File Types 编辑器：允许将特定的文件扩展名与应用程序关联起来。这在应用程序使用自定义文件扩展名以及要将特定的应用程序与该文件扩展名关联起来的情况下使用它是很有用的。
- User Interface 编辑器：允许配置安装过程中显示的对话框。
- Custom Actions 编辑器：允许在安装和卸载应用程序的过程中启动外部应用程序。
- Launch Conditions 编辑器：允许指定要安装在目标计算机上的应用程序启动要求。

在 View|Editor 子菜单下可以访问上述的所有编辑器，如图 20.22 所示。下面我们将详细介绍这些编辑器。

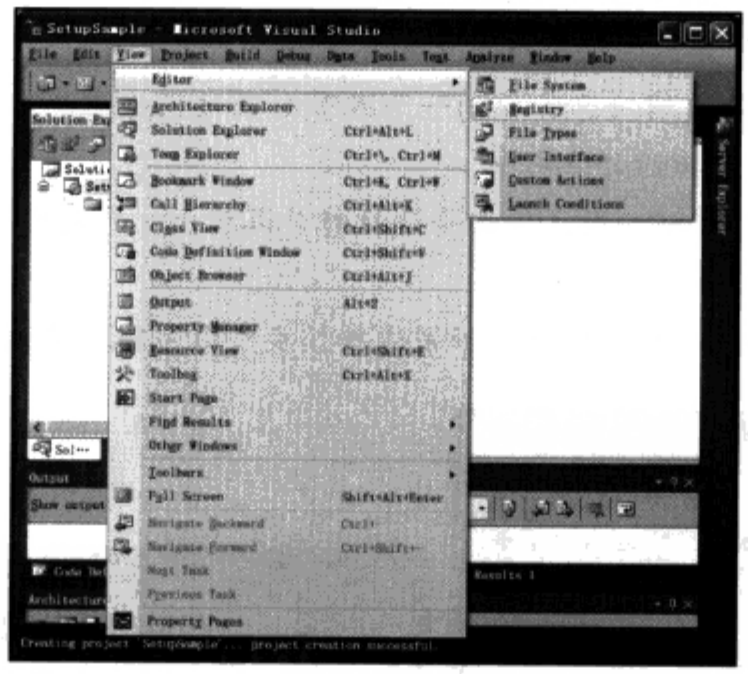


图 20.22　安装编辑器

20.5.1　File System（文件系统编辑器）

文件系统编辑器允序将项目输出文件、程序集及其他文件添加到部署项目中，也可用于指定组成应用程序的各种文件和程序集部署到目标机器的位置。

选择 View|Editor|File System 命令，就可以打开文件系统编辑器，该编辑器的界面如图 20.23 所示。

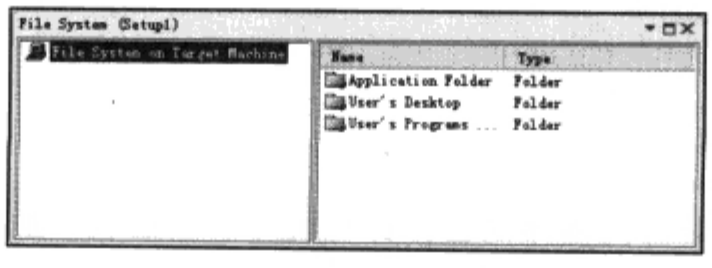


图 20.23　文件系统编辑器

右击 File System on Target Machine 项，在弹出的快捷菜单中选择 Add Special Folder|Global Assembly Cache Folder 命令，可添加该全局程序集缓存目录（默认情况下不会显示这个目录），如图 20.24 所示。

使用上述任何一个预定义文件夹，都可以在目标计算机上选择最终文件夹，甚至不必知道该文件夹的实际路径，安装程序会在虚拟路径获取该文件夹的实际路径。这些基本文件夹的作用如下。

- Application Folder: 指定应用程序的默认的安装根目录，它的常用构造路径为[ProgramFilesFolder][Manufacturer][ProductName]，默认的完整路径为：C:\Program Files\Manufacturer\ProductName。在设置项目的 Manufacturer 及 ProductName 属性时，就对应设置了应用程序的默认安装目录。在安装应用程序时，用户可以自己选择安装目录，而不必使用默认安装目录。
- Global Assembly Cache Folder: 指定应用程序中包含的需要作为共享程序集安装在目标计算上的路径。
- User's Desktop: 该文件夹充当显示在用户桌面上的文件和文件夹的占位符。该文件夹的默认路径为 C:\Documents and Settings\UserName\Desktop。
- User's Programs Menu: 该文件夹充当应该出现在用户程序集上的条目的占位符。该文件夹的默认路径为 C:\Documents and Settings\UserName\Start Menu\Programs。

使用文件系统编辑器可以将项目添加到这些特殊的文件中。选中一个文件夹。右击，在弹出的快捷菜单中选择 Add 命令，这里将列出 4 个类型，如图 20.25 所示。

- Folder: 允许在目标计算机的指定目录中创建文件夹。
- Project Output: 允许指定解决方案中的一个或多个项目的输出在目标计算机上的部署位置，也可以将所有依存关系添加到文件夹中。
- File: 使用该类型，可以将松散文件部署到目标计算机上，它对于部署帮助文件是很有用的，如 Readme、项目文档等。
- Assembly: 指定需要添加的程序集。如果选择该类型，系统会自动添加引用的所有程序集。

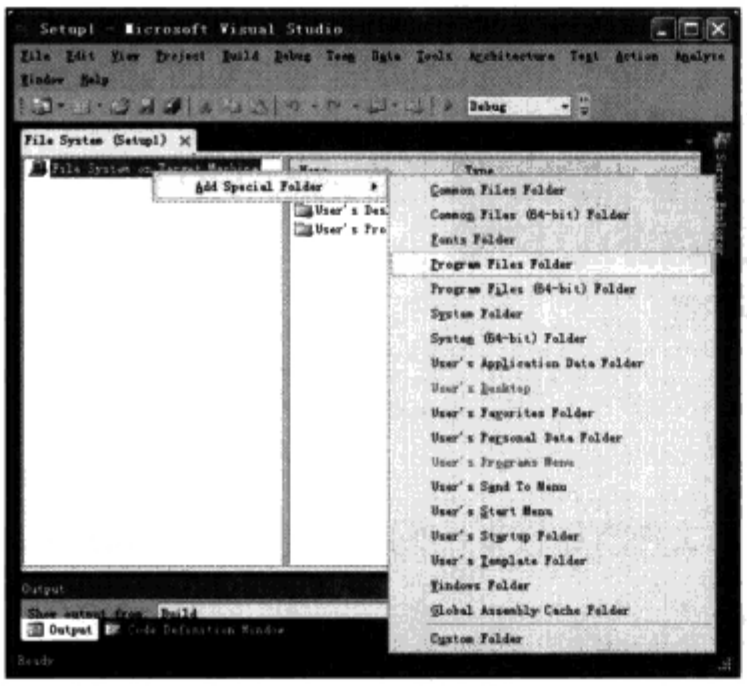


图 20.24 添加特殊文件夹

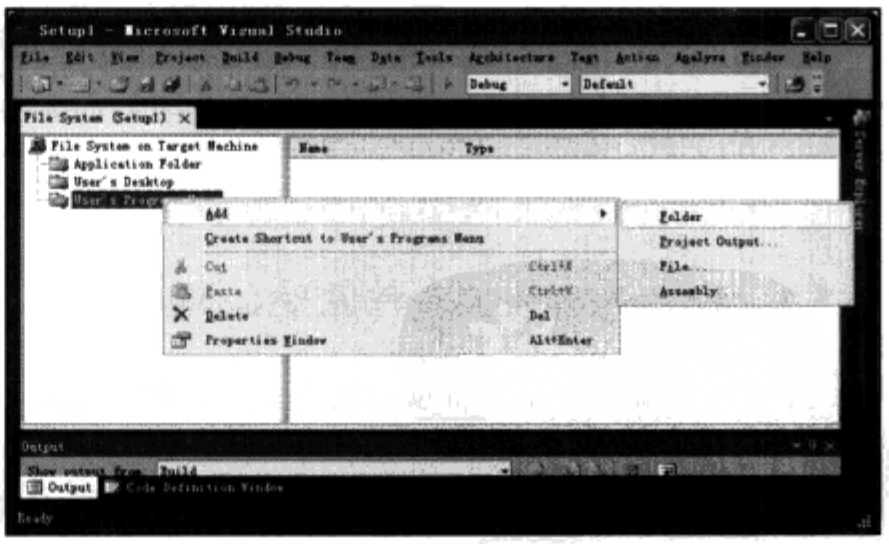


图 20.25 添加文件夹

20.5.2 Registry 编辑器

注册表编辑器允许对安装应用程序的目标计算机上的注册表添加键和数据。选择 View|Editor|Registry 命令就可以打开注册表编辑器，如图 20.26 所示。

可以看到，这个编辑器会显示标准的 Windows Registry 项，这些主键有：HKEY_CLASSES_ROOT、HKEY_CURRENT_USER、HKEY_LOCAL_MACHINE、HKEY_USERS 和 User/Machine Hive。默认情况下，HKEY_CLASSES_ROOT 和 HKEY_LOCAL_MACHINE 在 Software/[Manufacturer]键中包含了额外的条目，其中[Manufacturer]为在部署项目指定的 Manufacturer 属性的值。使用注册表编辑器可以将自定义的 Registry 项添加到上述任何项的下方。

如果要为应用程序添加额外的键和值，应选中编辑器左边的某个主键，右击，在弹出的快捷菜单中选择 New 命令，选择要添加的键或值类型，可选的值类型有 String Value、Environment String Value、Binary Value、DWORD Value。如果希望从外部导入已定义好的 REG 文件，则选中 Registry on Target Machine 项，右击，在弹出的快捷菜单中选择 Import 命令，在选择文件对话框中指定 REG 文件即可。

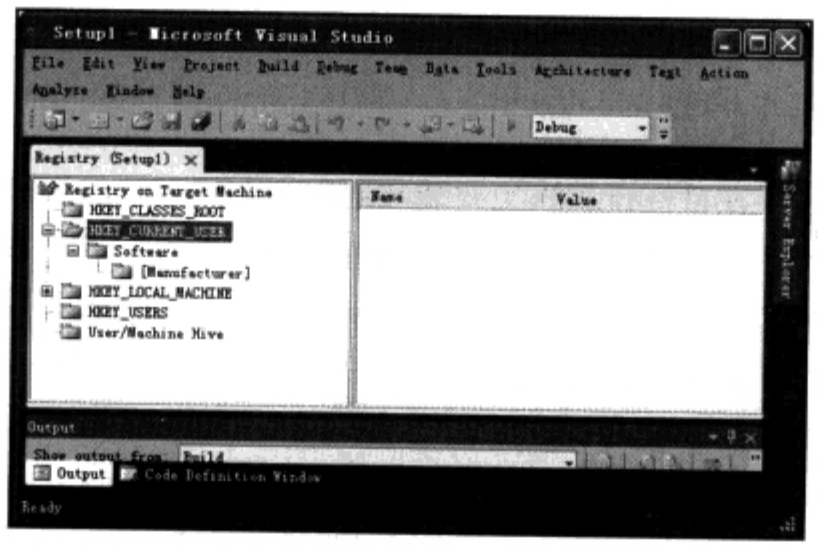


图 20.26　注册表编辑器

而如果想为某个键创建默认值，可先指定值类型，创建好一个名/值对后，选中这个值，在属性窗口中设置好 Value 属性（对应为值）后，将“（Name）”属性设置为空，这样就会显示为（Default），表明为该键的默认值。

这里需说明一点，维护应用程序设置的首选方法是使用基于 XML 的配置文件。与注册表相比，XML 文件提供了更大的灵活性，也更容易恢复和备份。

20.5.3　File Types 编辑器

文件类型（File Types）编辑器用于在目标计算机上建立文件和应用程序之间的关联。例如，在双击一个扩展名为.cpp 的文件时，Windows 就会启动 Visual Studio 打开这个 C++源文件。实际上文件类型编辑器就是为将部署的应用程序设置这种关联。如果将应用程序指定为某个文件扩展名，那么在双击文件时就可以启动正确的应用程序。一旦完成初始关联操作，Windows 资源管理器的文件类型列表中就会显示扩展名和文件类型说明。

为了添加文件和应用程序的关联，选中 File Type on Target Machine 项，右击，在弹出的快捷菜单中选择 Add File Type 项，这时出现一个新项，同时在属性窗口中显示这个关联的属性。表 20.3 列出了这些属性。

表 20.3　关系属性

属　性	说　明
（Name）	文件类型编辑器中使用的名称，以确定特定的文件类型
Command	指定用户使用该类型打开文件时应该启动的可执行文件，单击“…”按钮，就可以选择可执行的应用程序

续表

属　性	说　明
Description	对文件类型的说明信息
Extensions	指定可执行应用程序应该注册的文件扩展名。注意设置该属性的值时，不要输入点号，可以用分号隔开多个扩展名，如 exe1;exe2
Icon	指定该类型文件（指具有该扩展名的文件）在目标机器上显示的文件类型图标
MIME	指定与该文件类型相关联的一个或多个 MIME 类型

注意可以为某个可执行的应用程序同时指定多个不同的文件扩展名，但是任何一个扩展名都只应与一个应用程序相关。在指定自己的应用程序关联的文件类型时，不要使用那些常用的扩展名，如.doc、ppt 等。

20.5.4　User Interface 编辑器

用户界面编辑器用于设置在目标计算机上安装应用程序期间显示的用户界面对话框序列，可以为一组预定义的对话框指定属性。选择 View|Editor|User Interface 命令，就可以打开用户界面编辑器，如图 20.27 所示。可以看到该编辑器主要分为两个主要部分：Install 和 Administrative Install，其中前者用于标准安装，而后者针对管理员安装。这两部分又进一步划分为下列 3 个子部分。

- Start：这类对话框在安装开始时显示。
- Progress：在安装期间，向用户提供安装进度的反馈信息。
- End：在安装成功完成后显示的对话框，也可以用于查看 Readme 文件或立即启动应用程序。

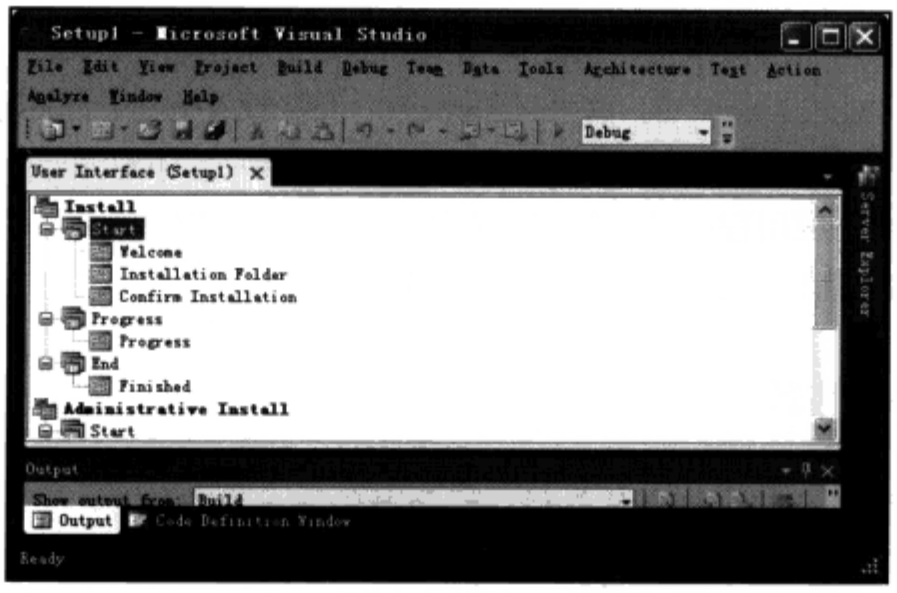


图 20.27　用户界面编辑器

选中这 3 个子部分中的一个，右击，在弹出的快捷菜单中选择 Add Dialog 命令，弹出的对话框如图 20.28 所示，将列出可以添加到项目中的对话框类型，如表 20.4 所示。

表 20.4　　对话框类型

对　话　框	说　明
Confirm Installation	该对话框用于在安装之前确认各个设置
RadioButtons（2～4 buttons）	包含 2～4 个单选按钮，每个按钮都具有 Label 和 Value 属性
Checkboxes（A～C）	至多包含 4 个复选框，每个复选框都具有 Label、Property、Value 和 Visible 属性
Customer Information	包含序列号模板、组织名称，以及是否显示序列号的编辑字段
Textboxes（A～C）	至多包含 4 个文本框，每个文本框都具有 Label、Property、Value 和 Visible 属性
Finished	安装完成后显示的对话框

续表

对　话　框	说　　明
Installation Folder	用于在客户安装应用程序时，显示一个让客户指定安装文件夹的对话框
License Agreement	显示应用程序许可协议，其中的 LicenseFile 属性指定协议所在的文件
Progress	在安装过程中，用于显示安装进度
Read Me	显示 readme 信息，其中的 ReadmeFile 属性指定指定这些信息所在的文件
Register User	指导用户注册操作的应用程序，这个应用程序必须在安装项目中提供
Splash	显示一个指定的位图图像
Welcome	显示应用程序的欢迎信息和版权警告信息，包含两个属性：CopyrightWarning 和 WelcomeText，都是字符串属性

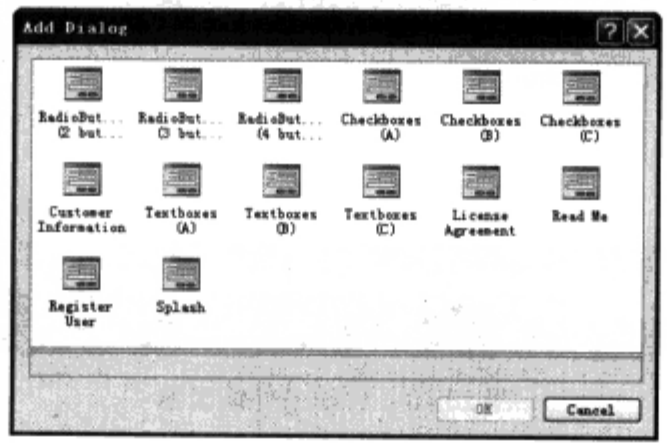


图 20.28　添加对话框

20.5.5　Custom Actions 编辑器

客户行为编辑器用于定义在应用程序安装的某些阶段进行的定制动作。要创建自定义操作，首先需要创建执行自定义操作的.dll、.exe、脚本或 Installer 类，然后将其添加到部署项目。选择 View|Editor|Custom Actions 命令，就可以打开客户行为编辑器，如图 20.29 所示。

这里包含 4 种不同的阶段用于定制客户动作。

- Install: 定制动作在安装阶段的最后执行。
- Commit: 定制动作在安装完成后执行。
- Rollback: 定制动作在回退阶段文成后执行。
- Uninstall: 定制动作在卸载完成后执行。

要对某个阶段添加动作，可选中该项，然后右击，在弹出的快捷菜单中选择 Add Custom Action 命令，打开 Select Item in Project 对话框，选择定制动作。注意，这些包含定制动作的组件必须是部署项目的一部分。

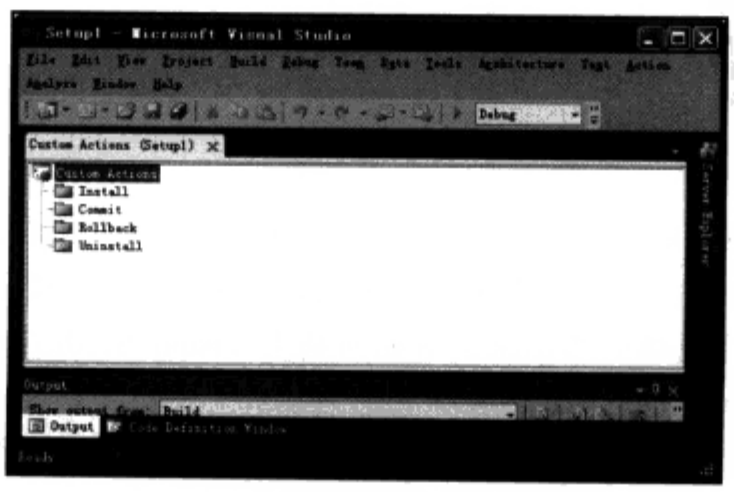


图 20.29　客户行为编辑器

20.5.6 Launch Conditions 编辑器

启动条件编辑器用于指定运行安装程序必须满足的条件。如果用户试图将应用程序安装到不满足启动条件的计算机上，安装程序就不会运行。设置启动条件时，也可以指定在目标计算上执行的搜索操作，以确定特定文件、Registry 项或组件等存在。选择 View|Editor|Launch Conditions 命令，就可以打开启动条件编辑器，如图 20.30 所示。

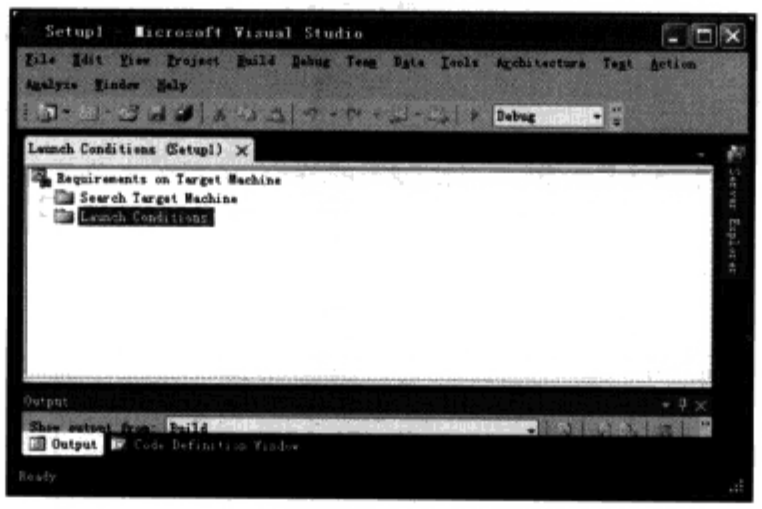


图 20.30 启动条件编辑器

可以看到，Requirements on Target Machine 项下包含两个组：Search Target Machine 和 Launch Conditions。前者用于指定在目标计算机上需要执行的搜索类型，有 File Search、Registry Search 和 Windows Installer Search 3 种类型；而后者用于定义启动应用程序安装程序必须满足的条件。一般需要进行搜索设置，根据该搜索的成功与否来执行条件。这可以通过设置搜索的 Property 属性实现，可以在安装过程中访问 Property 属性；还可以在编辑器中添加 Launch Condition，并将条件的 Condition 属性设置为对应搜索的 Property 属性值；也可以在条件中指定 InstallUrl 属性，用于下载所搜索的文件、注册表或安装组件。

File Search 搜索文件或文件类型，通过设置相关的属性来决定如何搜索文件，这些属性有：Depth（搜索深度）、FileName（文件名）、Folder（文件夹位置，默认为[SystemFolder]）、MaxData、MinData（文件日期范围）、MaxSize,MinSize（文件大小范围）、MaxVersion,MinVersion（版本信息范围）及指定一个 Property 值。

Registry Search 通过设置 Root 属性指定搜索的根键名，通过设置 RegKey 和 Value 属性设置搜索的键和值。

Windows Installer Search 搜索指定的安装组件，通过指定 ComponentId 属性来搜索，这个属性实际上对应为 GUID。

20.6 小结

本章简单讨论了 Visual Studio2010 应用程序的安装和部署，并介绍在.NET 框架下进行应用程序部署的常用工具，有 XCOPY 工具、Copy Web 工具及 CLRckOnce 等。接下来介绍了 Windows Installer 的一些基本概念，并通过一个具体的实例来讲解如何利用 Visual Studio2010 提供的安装向导功能，只用了 5 步，就完成了一个安装程序。

接着介绍了如何手动进行安装程序的设计，这里详细介绍了安转程序的各种属性及其含义，最后详细讨论了 Visual Studio2010 提供的 6 种安装项目编辑器。

第五部分　基于 Windows 7 平台用 Visual C++ 2010 开发

第 21 章　Visual C++ 2010 基于 Windows 7 新特性开发

21.1　实现 C++兼容开发

如果应用程序与 Windows Vista 兼容，我们期望在 Windows Vista 中运行的大部分应用程序都能在 Windows 7 上运行。显然，这不是绝对的，必须检查应用程序是否确实与 Windows 7 兼容。但是，如果应用程序是专为 Windows XP（或更早版本）设计的，不确定它与 Windows Vista 的兼容性，那么应该特别注意一些方面（如 UAC）。本节将讨论开发与 Windows 7 兼容的应用程序需注意的几个问题，包括 UAC 数据重定向、高 DPI、安装程序检测、会话 0 隔离、用户界面特权隔离和版本检查。

21.1.1　实现 UAC 数据重定向

很多应用程序都设计为将文件写入 Program Files、Windows 目录或者系统根（一般是 C 盘）文件夹。一些应用程序被设计为更新 Microsoft Windows 注册表的值，特别是 HKLM/Software 中的值。但是这样会存在一个问题：文件或注册表的值并没有被创建或更新。用户可能遇到过下面的一种或多种情况。

- 应用程序写入 Program Files、Windows 目录或者系统根（一般是 C 盘）文件夹，但是这些地方并没有相应文件。
- 应用程序写入 Windows 注册表，特别是 HKLM/Software 中，但是没有看到注册表进行了更新。
- 切换到另一个账户时，应用程序无法找到已写入 Program Files、Windows 目录或者系统根（一般是 C 盘）文件夹的文件，或者找到了这些文件的老版本。
- 在启用或禁用用户账户控制（UAC）后，应用程序无法找到 Program Files 或 Windows 目录中的文件。

如果应用程序出现了这些情况，都是由于 UAC 虚拟化造成的。

在 Windows Vista 中，一般由管理员来运行应用程序。这样，应用程序可以自由地读/写系统文件和注册表的值。如果标准用户运行这些应用程序，可能会由于没有足够的权限而导致失败。Windows Vista 通过重定向写入（以及后续文件或者注册表操作）对应用户的配置文件（profile）位置，来对标准用户运行应用程序的能力进行提升。

例如，如果一个应用程序试图写入 C:\Program Files\Contoso\Settings.ini，并且用户没有权限来对该文件夹进行写入操作，写入操作将会被重定向到 C:\Users\Username\AppData\Local\VirtualStore\ Program Files\Contoso\settings.ini。如果应用程序试图在注册表中写入 HKEY_LOCAL_MACHINE\ Software\ Contoso\，这将会被自动重定向到 HKEY_CURRENT_USER\Software\Classes\VirtualStore\MA CHINE\ Software\Contoso 或 HKEY_USERS\UserSID_Classes\VirtualStore\Machine\Software\Contoso。

图 21.1 显示了 Windows 虚拟化进程的两个组件：文件虚拟化和注册表虚拟化。

虚拟化是为了帮助现存的应用程序解决兼容性问题而开发的。为 Windows 7 设计的新的应用程序不应该对敏感的系统区域进行写入操作，不应该依靠虚拟化来对不正确的应用程序行为进行修正。开发应用程序时，应该始终考虑使其运行在标准用户的权限下，而不是在管理员权限下运行。测试应用程序时，也要用标准用户权限而不是管理员权限。

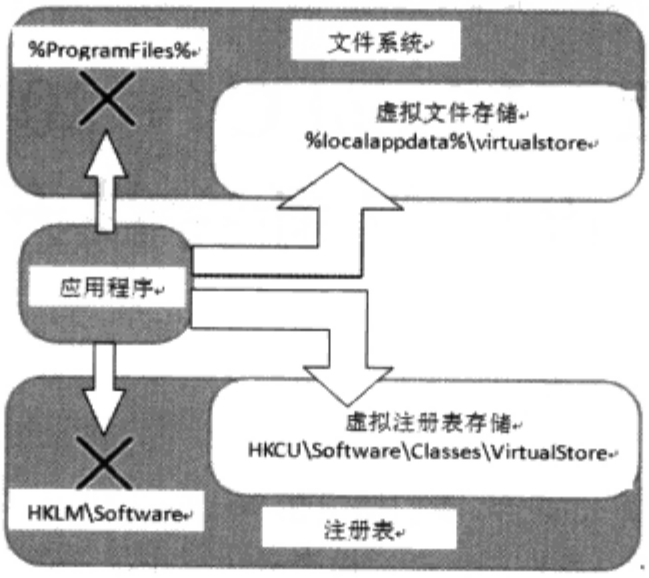


图 21.1　文件虚拟化和注册表虚拟化

如果正在使用 UAC 虚拟化来为 Windows 7 开发应用程序，那么请您应用程序重新设计为将文件写入合适位置。当更新现有的代码以在 Windows 7 上运行时，应该：

- 确定在运行时，应用程序存储数据只是在对应用户的位置或者在对访问控制列表（ACL）进行了设置的%alluserprofile%系统位置。
- 使用已知的文件夹来写入数据文件。所有用户可以使用的一般数据应该被写入一个对所有人共享的公共位置。所有其他数据都应该被写入对应用户的位置。

一般数据文件可以包含日志文件、配置文件（INI/XML）、状态存储应用程序，如保存的游戏等。用户文件则不同，它们应该保存到 Documents 文件夹（或者用户指定的位置）。

- 确保没有在代码中指定路径。建议使用下面的模型和 APIs 来获取特定的 Windows 的正确路径。

C/C++ Native 应用程序：使用 SHGetKnownFolderPath 方法来获取已知文件夹的整个路径，它们可以用文件夹的 KNOWNFOLDERID 来识别。它是一个标识想要获取的已知位置的 GUID 参数：

FOLDERID_ProgramData，向所有人共享项目数据的目录；

FOLDERID_LocalAppData，对应用户项目数据的目录（不可移动）；

FOLDERID_RoamingAppData，对应用户项目数据的目录（可移动）。

托管代码：使用 System.Environment.GetFolderPath()方法。GetFolderPath 包含一个标识想要获取的已知位置的参数：

Environment.SpecialFolder.CommonApplicationData，向所有人共享项目数据的目录；

Environment.SpecialFolder.LocalApplicationData，对应用户项目数据的目录（不可移动）；

Environment.SpecialFolder.ApplicationData，对应用户项目数据的目录（可移动）。

- 如果上面提到的方法都不起作用，则使用环境变量：

%ALLUSERSPROFILE%，向所有人共享项目数据的目录；

%LOCALAPPDATA%，对应用户项目数据的目录（不可移动），Windows Vista 或更高版本可用；

%APPDATA%，对应用户项目数据的目录（可移动），Windows Vista 或更高版本可用。

以上讲解了与 UAC 虚拟化相关的情况，解释了为什么会发生重定向，并给出了建议的解决方案。下面给出一些试验和操作，以便一点一点地定位问题，并使用相应的方法来解决它。

试验 1

- 使用标准用户权限运行应用程序。
- 运行写入受保护的文件夹（如 Program Files 或者系统根（C:\）目录）的操作。
- 期待的结果：应用程序成功将文件写入受保护的文件夹；但是，无法在预期的位置找到。
- 结论：这说明 UAC 数据虚拟化把文件重定向到了另一个位置。

试验 2

- 使用 Windows 资源管理器在 VirtualStore 文件夹中搜索文件。
  - VirtualStore 文件夹是在配置文件（profile）下的用来存储重定向文件的文件夹。
  - VirtualStore 文件夹的名字和位置会在 Windows 以后的版本中改变。
- 在%localappdata%\VirtualStore 下查找。
- 如果没有找到，则尝试使用命令：dir %userprofile%\yourfile.dat /s /a。
- 期待的结果：应该可以在 VirtualStore 文件夹中找到文件。
- 结论：这可以证明 UAC 虚拟化确实发生了。

试验 3

- 以管理员的身份登录并以管理员身份运行应用程序。
  - 要想以管理员身份运行应用程序，可右击可执行文件，并选择“以管理员身份运行”命令。
  - 当应用程序以管理员身份运行时，虚拟化处于禁用状态，且应用程序有权限向受保护的文件夹进行写入操作。
- 不要始终使应用程序获取管理员权限来绕过虚拟化，这样会在系统中留下很多容易被攻击的地方，也会使应用程序不能使用受限或是标准用户权限运行，并且匿名用户在使用时也无须弹出 UAC 窗口。
- 期待的结果：应用程序成功向受保护的文件夹进行了写入操作，并且可以在期望的位置找到该文件。
- 结论：使用管理员身份运行应用程序会禁用虚拟化，并具有可以向受保护的系统位置进行写入的权限。

试验 4

- 运行 Process Monitor（ProcMon）。
- 配置过滤，使其只显示文件操作和执行这些操作的进程。

当运行 Process Monitor 时，会弹出 Process Monitor Filter 对话框。

- 将下面的条目添加到过滤器中：Column=Process Name, Relation=is, Value=YourApp.exe, Action=Include，如图 21.2 所示。

再次调用 Process Monitor Filter 对话框，在工具栏中单击 按钮。如图 21.3 所示。

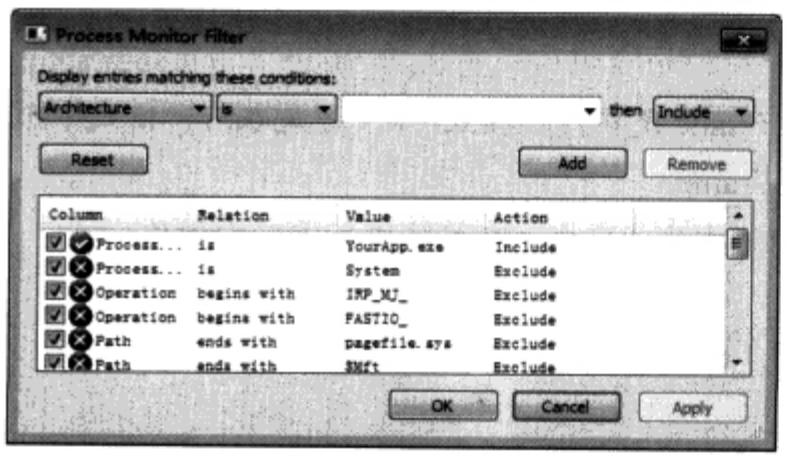


图 21.2　进程监视器过滤器

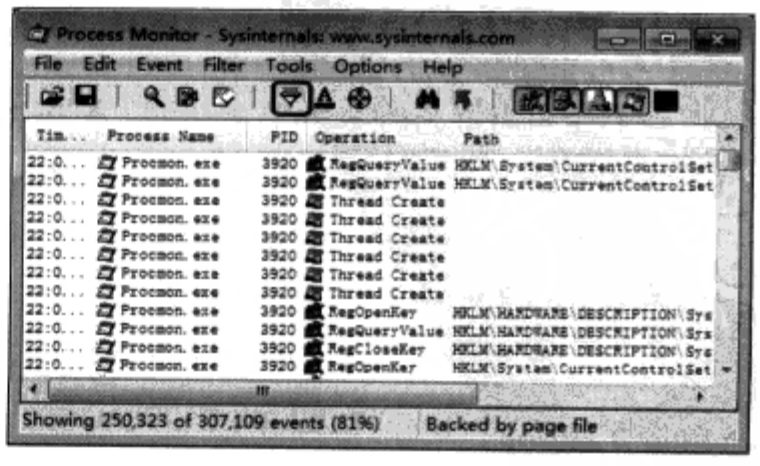


图 21.3　重新调用进程监视器过滤器

- 单击 OK 按钮，通过启用如图 21.3 所示的工具栏按钮来配置 Process Monitor 只记录文件事件： ，如图 21.4 所示。
- 按【Ctrl+X】组合键来清除日志，按【Ctrl+E】组合键来开启或关闭 logging。
- 期待的结果：将看到结果为 REPARSE 的文件操作，下面一行（结果是 SUCCESS）通常是重定向操作，如图 21.5 所示。

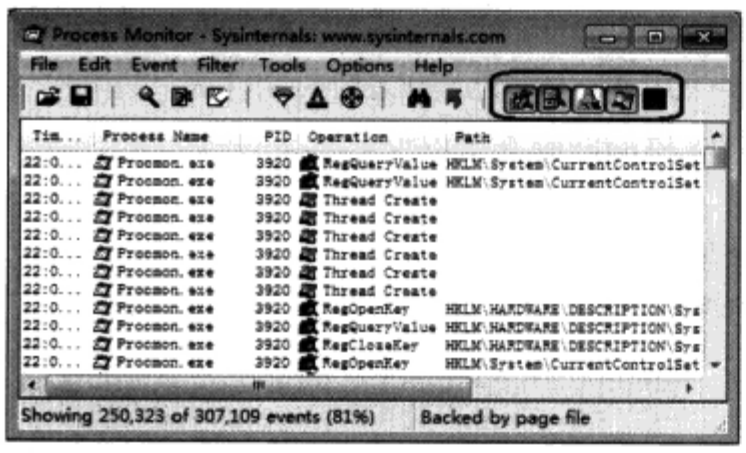


图 21.4　配置记录文件事件

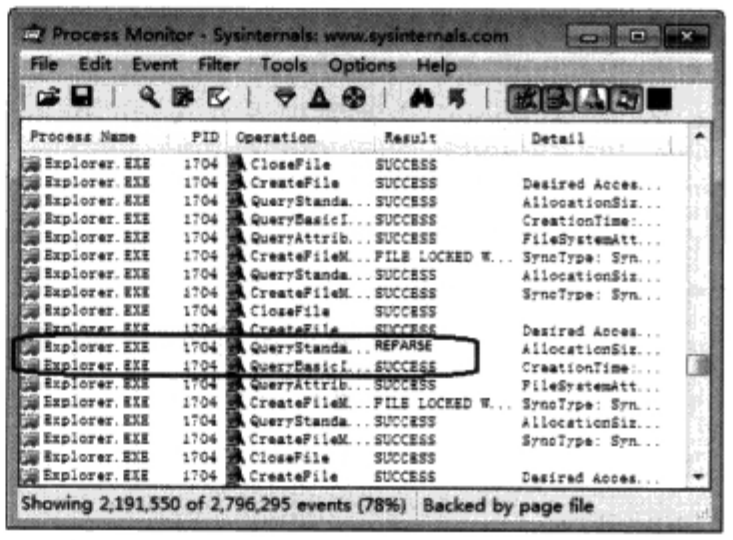


图 21.5　进程监视器内查看的重定向结果

- 双击显示 REPARSE 结果的行，并单击 Stack 选项卡来查看操作时所调用的堆栈，如图 21.6 所示。

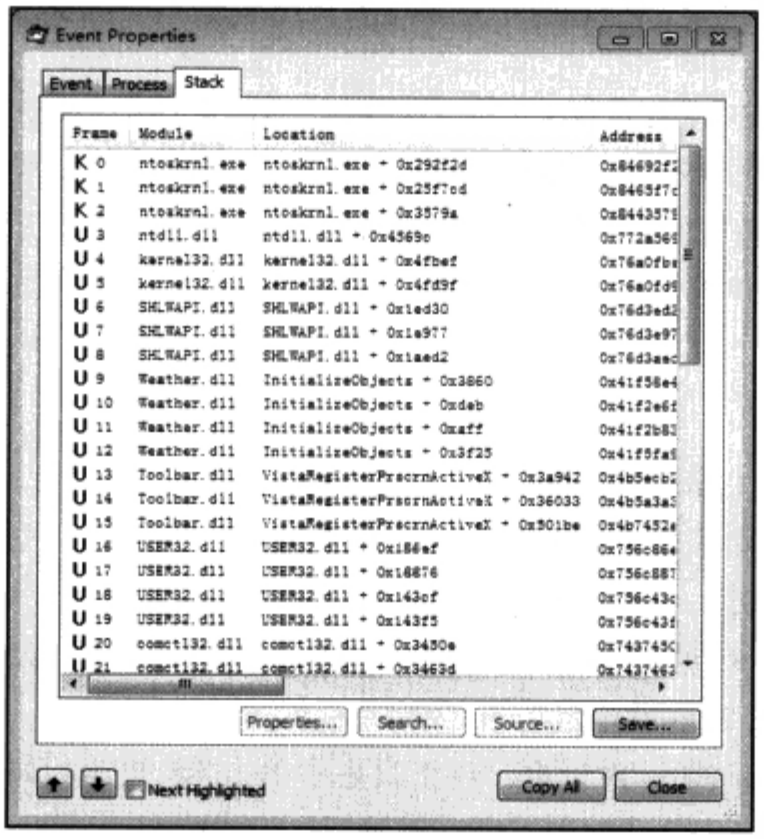


图 21.6　操作所调用的堆栈

- 左侧显示的粉色的 K 和蓝色的 U，表示 Stack Frame 是在内核模式（K）还是在用户模式（U）。在这种情况下，只关注在用户模式下的 Stack Frame。
- 本例中，在 BrokenAppNative.exe 中的 SaveFile()方法（Frame 21）将进行重定向操作。
- 为更有意义的界面进行标志配置。
- 结论：该测试证明了 UAC 虚拟化确实发生了，还显示了应用程序哪里需要修改。

试验 5

- 为包含 UAC 指令的应用程序添加一个清单文件。

  - 这将把应用程序标记为支持 UAC 的应用程序，并且将会关闭 UAC 虚拟化。
  - 清单文件是一个程序嵌入 DLL 或 EXE 文件中的作为资源的 XML 文档，但它可以是一个独立的名为 YourApp.exe.manifest 或 YourDLL.dll.manifest 的文件。
  - 清单文件可以包含很多通常关于应用程序兼容性的信息，如 Visual C++运行时的确切版本、公用控件库的版本，以及 UAC 配置。
- 期待的结果：该应用程序无法向受保护的文件夹进行写入操作，并返回了“拒绝访问”错误。
  - 这是一个很好的结果，因为 UAC 数据虚拟化并没有发挥作用。
  - 作为开发者，应该认识到这一点（在清单文件中把应用程序标记为支持 UAC），并且避免对受保护的位置进行写入操作。
- 结论：Windows 开启虚拟化是因为应用程序没有被标记为支持 UAC。使应用程序支持 UAC，以关闭虚拟化。

21.1.2　实现高 DPI

调查表明，只有约 55% 的用户设置的是高分辨率。这表明，几乎一半的用户没有充分享用高分辨率的优势。

大部分用户使用低分辨率，是因为他们发现高分辨率下的文字比较小。不过，可以完全减少高分辨率所带来的负面影响，包括下面的各项。

- ClearType 不正确地呈现非本地的分辨率。
- 用户无法看到全保真高清内容。
- 高分辨率的照片不能呈现保真度最高的状态。

为了增大文字，鼓励用户增加 DPI 设置显示分辨率高的优势，而不是降低分辨率。虽然这解决了与高保真流失的问题，但是应用程序在没有 DPI 感知的情况下经常呈现一些不合适的效果，如裁剪文本、Windows 窗体布局问题、图片大小问题、界面模糊、字体大小不匹配等。

随着 Windows 7 的发布，将更广泛地采用高 DPI，理由如下。

- 更多的 OEM 厂商预计将开始销售高 DPI 设置配置的笔记本电脑。
- Windows 7 的简洁配置已预先配置为高 DPI。
- 在用户界面的控制面板中可以很容易地设置高 DPI

使应用程序支持高 DPI 比以往更重要。首先，保证计算机在高 DPI 模式，可以通过控制面板中的显示设置来完成此操作，如图 21.7 所示。

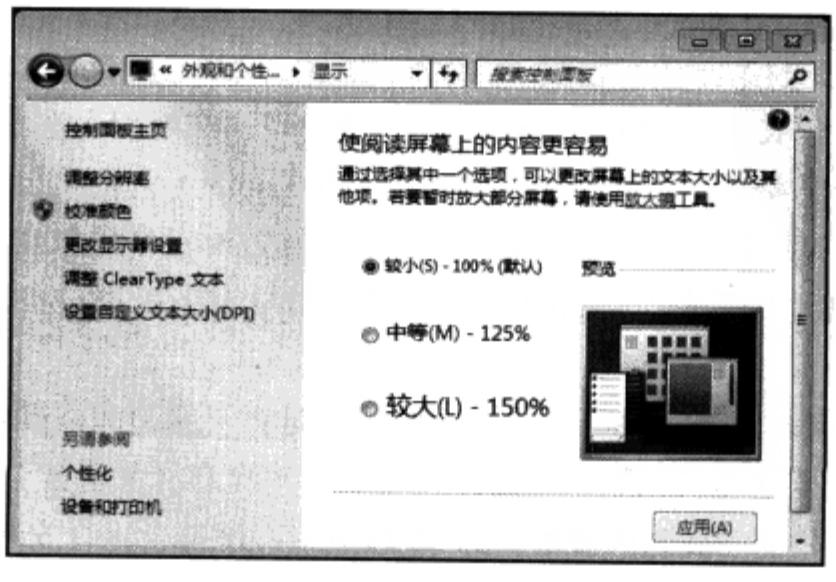


图 21.7　控制面板中的显示设置

注意，Windows 7 使用缩放百分比来代替原始的 DPI 值，其中 100%=96 DPI，125%=120 DPI，150%=144 DPI。

表 21.1 列出了在不同 DPI 设置下系统参数的变化。

表 21.1　　不同 DPI 设置下的系统参数

系 统 参 数	96 DPI	120 DPI	144 DPI
GetDeviceCaps(hDC, LOGPIXELSX)	96	120	144
DEFAULT_GUI_FONT height	13	16	20
SM_CXHTHUMB (scroll bar control)	17	21	26
SM_XICON (icon width)	32	40	48

可以看到，随着 DPI 值的增加，UI 使用更多的像素来绘制图表、UI 控件和文本。

测试高 DPI 环境中的应用程序。WinForm 应用程序通常结构良好，只需要很少的 UI 控件调整。有时，需要改变“自动缩放模式”。本应用程序可能会遇到上述很多问题。首先，加入下列代码，作为 DPI 应用感知：

```
<application  xmlns="urn:schemas-microsoft-com:asm.v3">
<windowsSettings>
<dpiAware  xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">true</dpiAware>
</windowsSettings>
</application>
<!-- See http://msdn2.microsoft.com/en-us/library/ms633543.aspx -->
```

将应用程序声明为 DPI 应用感知后，就需要将应用中的所有 UI 文本、点及尺寸使用 GetDeviceCaps API 调用返回的值进行调整，如下面的代码所示：

```
//获取系统 DPI 值
HDC hdc = GetDC(NULL);
intdpiX = GetDeviceCaps(hdc, LOGPIXELSX);
intdpiY = GetDeviceCaps(hdc, LOGPIXELSY);
ReleaseDC(NULL, hdc);
…
//使用 DPI 值进行调整
SIZE buttonSize = {60, 26};
buttonSize.cx = buttonSize.cx * dpiX / 96;
buttonSize.cy = buttonSize.cy * dpiY / 96;
CreateWindow("Button",…, buttonSize.cx, buttonSize.cy,…);
```

21.1.3　实现安装程序检测

随着用户账户控制（UAC）在 Windows Vista 中的出现，应用程序在默认情况下都是以标准用户的权限来运行的。这对于早期版本的应用程序会是一个问题，因为它们将以管理员权限来运行。然而，一些项目需要以管理员的权限来运行，多数安装程序显然需要对特殊权限的位置进行写入操作，如 Program Files 文件夹或者是注册表中的 HKEY_LOCAL_MACHINE。以标准用户访问这些项目都将失败，并收到拒绝访问错误。

在 Windows Vista 中要对此进行修复，操作系统将会进行一些检测以确定它是不是一个安装程序。如果是，系统会为其提升至管理员权限。但是，有时候这些检测自身也会引发问题。

所有没有清单文件（外部的或是内置的）的 32 位应用程序都会进行安装检测。操作系统假设这些应用程序都是“早期版本”，它们都将以管理员权限来运行。

检测方法包含查看文件名并搜索文件名中是否包含关键字 setup、install 和 update，如果找到了，一个盾牌似的图标会覆盖在原有图标上，就像如图 21.8 所示的 Windows 资源管理器图标一样，表明这个应用程序需要以管理员权限来运行。

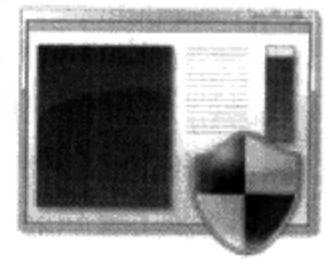


图 21.8　资源管理器图标

虽然该检测可以使早期版本的安装程序在 Windows Vista 或者以后版本的操作系统中正确运行，但它也会引起对一些非安装应用程序进行不必要的权限提升（如 StockUpdater.exe）。这可能给用户带来安全隐患和麻烦，并且由于害怕它们可能给系统带来损害，用户可能会放弃使用该应用程序。

反过来也仍然会有问题。一个不使用 Microsoft Windows Installer（MSI）并且没有按照检测规则命名的自定义安装程序，可能确实需要以管理员权限来运行，但是由于检测没有把它识别为一个安装程序，没有添加类似盾牌的图标或者没有提升权限，应用程序的安装可能会失败。

最直接的解决方案是添加代码，可以添加外部或内置代码。如果应用程序源不可用，那么只能在包含可执行文件的文件夹中添加一个外部代码清单文件。该清单文件的名称应该使用与可执行文件相同的名称，并在其后面加上.manifest（例如，StockUpdater.exe.manifest）。该清单文件应该定义了应用程序何时需要管理员权限，或者使用标准用户权限就可以成功运行。

下面给出一个示例：

```
<?xmlversion="1.0"encoding="utf-8"?>
<asmv1:assemblymanifestVersion="1.0"xmlns="urn:schemas-microsoft-com:asm.v1"xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<assemblyIdentityversion="1.0.0.0"name="MyApplication.app"/>
<trustInfoxmlns="urn:schemas-microsoft-com:asm.v2">
<security>
    <requestedPrivilegesxmlns="urn:schemas-microsoft-com:asm.v3">
    <!-- UAC Manifest Options
    If you want to change the Windows User Account Control level replace
 the requestedExecutionLevel node with one of the following.
      <requestedExecutionLevel  level="asInvoker" uiAccess="false" />
     <requestedExecutionLevel  level="requireAdministrator" uiAccess="false" />
     <requestedExecutionLevel  level="highestAvailable" uiAccess="false" />
     -->
       <requestedExecutionLevellevel="asInvoker"uiAccess="false" />
   </requestedPrivileges>
  </security>
  </trustInfo>
</asmv1:assembly>
```

安装检测通过确定早期版本的项目为安装程序来允许这些程序正常运行。但是，这些检测可能会导致不必要的结果。系统只会对没有清单文件的 32 位可执行文件进行检测，因此可以通过添加一个清单文件来去除不必要的检测，并指定所期望的行为。

21.1.4　Session 0 隔离

服务是整合在 Microsoft Windows 操作系统中的结构。服务与用户应用程序不同，因为用户可以对服务进行配置，不需要用户登录，就可以使这些服务在系统启动时运行，直到系统关闭。Windows 中的服务负责所有种类的后台活动，但不包括从远程过程调用（RPC）服务到网络位置服务的用户。

一些服务可能会试图显示一些用户界面对话框，或者与用户的应用程序进行通信，那么这些服务将会遇到与 Windows 7 的兼容性问题。例如，可能有一个服务试图显示一个对话框，但是仅仅在任务栏中显示一个闪烁的图标。服务还可能会遇到下面的一些现象。

- 服务正在运行，但是没有做它们本希望去做的事情。
- 服务正在运行，但是其他的一些进程却不能和这个服务进行通信。
- 服务试图通过窗体消息与用户的应用程序进行通信，但是窗体消息并没有被目标所接收到。
- 服务在任务栏中显示一个闪烁的图标，说明此服务希望与桌面信息进行交互。

这些是 Windows 7 服务的 Session 0 隔离机制的一些症状。它们可以大致分为两大类：服务显示用户界面失败，或者仅仅显示了一个简化的用户界面。当一个服务试图展示任何有关用户界面的元素（即使这个服务被允许与桌面信息进行交互）时，一个叫做 Interactive services dialog detection 的简单对话框也会弹出，用于提示用户。用户可以进入 Session 0 的安全桌面中查看服务的用户界面，但是，由于工作流的干扰，使得这变成了一个严重的应用程序兼容性问题。

当一个对象由服务创建出来，并且允许标准应用程序访问（以标准用户权限运行）时，那么这个对象将不能在全局命名空间中找到（它将被 Session 0 私有）。此外，安全变更也将保证即使对象是可见的，但它也是不能被访问的。这些将有可能影响其他与服务进行交互的进程（如普通应用程序）。

在 Windows XP、Windows Server 2003 或者更早期的 Windows 操作系统中，所有的服务和应用程序都是运行在第一个登录到控制台的用户的会话中。这个会话叫做 Session 0。在 Session 0 中，服务和用户应用程序一起运行。由于服务是以高权限运行的，因此会造成一些安全风险。一些恶意代码往往会利用这一点，来寻找提升它们自身的权限途径。

在 Windows Vista 中，服务在一个叫做 Session 0 的特殊会话中加载。由于应用程序运行在用户登录到系统后所创建的 Session 0 之后的会话中，所以应用程序和服务也就隔离开来：第一个登录的用户在 Session 1 中，第二个在 Session 2 中，依此类推。事实上，运行在不同的会话中，如果没有特别将其放入全局命名空间并且进行相应的访问控制配置，是不能互相传递窗体消息、共享 UI 元素或共享 kernel 对象的，如图 21.9 所示。

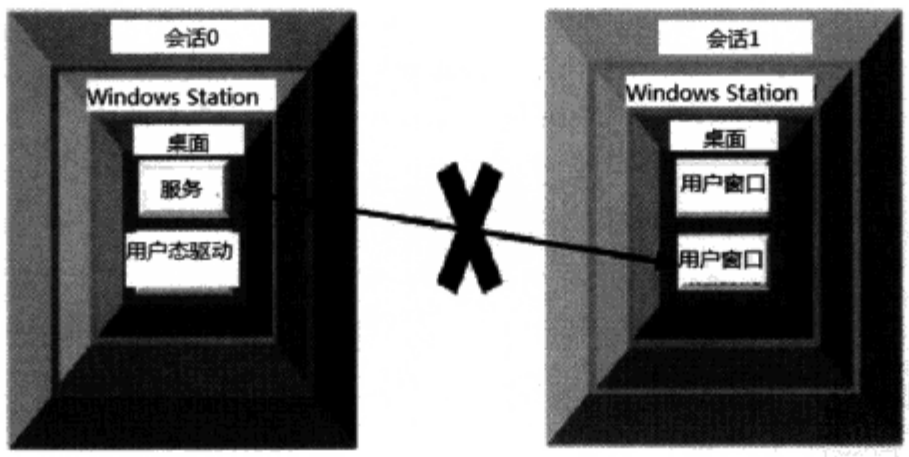


图 21.9　不同会话之间无法传递窗体信息

解决此问题有以下几种方案。

- 如果一个服务需要向用户传递一条消息，可以使用 WTSSendMessage()方法。这个方法和 MessageBox 基本相同，它将可以给不需要复杂 UI 界面的服务一个简单而有效的解决方案。同时，由于所显示的消息对话框不能用于控制底层的服务，因此这个解决方案也是安全的。
- 如果需要使用一个复杂的 UI 界面，可以使用 CreateProcessAsUser()方法，在提出请求的用户桌面中创建一个进程。
- 如果这两种交互方式都需要，则可以使用诸如 Windows Communication Foundation

（WCF）、.NET 远程处理、命名管道或者其他的进程间通信（IPC）结构（除窗体消息之外）来跨会话通信。

- 对于安全通信和其他共享对象（如命名管道、文件地图），可以使用自由访问控制列表（DACL）来严格控制用户的权限设置，使用系统访问控制列表（SACL）来确保中低权限的进程依然可以访问由系统或高权限服务所创建的结构。
- 确保跨会话访问的 kernel 对象是以 Global\字符串为首字母进行命名的，这意味着它们是属于全局会话命名空间中的。

以上了解了所有的 Windows 服务 Session 0 隔离机制的情况，解释了什么是服务隔离，它是如何影响服务和应用程序的，并且提出了一些解决方案。下面进行一些测试和操作，以找到问题并解决。

试验 1

（1）打开进程浏览器（可到 Microsoft Tech Net 网站下载）。

（2）使进程浏览器显示所有的进程：（1）单击 File，（2）选择 Show Processes From All Users 命令。

（3）定位到可疑的服务，并且查看其属性。右击进程，选择 Properties 命令。切换到 Security 选项卡，注意服务运行在哪个 Session 中（通常在 Session 0 中），及其全部级别。

图 21.10 展示使用系统权限运行的进程属性。

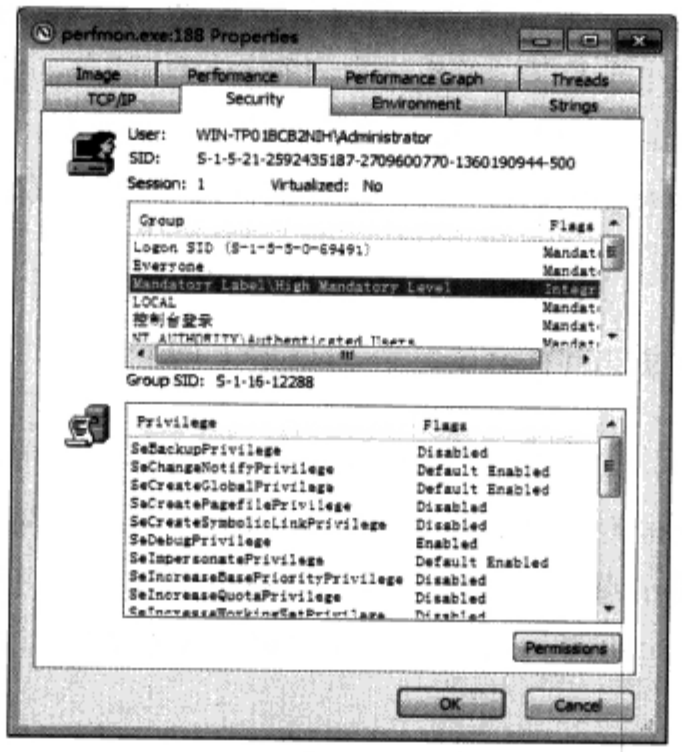


图 21.10　以系统权限运行的进程

如果服务以高权限运行在会话 0 中，那么它将不能直接显示 UI。不过即使这样，当与服务共享 kernel 对象或者文件时，可能还会遇到一些问题。

试验 2

（1）打开进程浏览器。

（2）确保进程浏览器显示了所有的进程。

（3）定位到可疑的服务。

（4）如果服务包含已知的与用户应用程序共享的对象，可在下面的 Handles 窗格中检查它们的句柄（按【Ctrl+H】组合键查看，或者从 View 菜单中选择）。

① 右击每一个可疑的句柄，选择 Properties 命令。

② 切换到“安全”选项卡，查看当前用户和组是否允许使用当前句柄来访问对象。

图 21.11 显示了一个不管是不是在会话 0 中运行的系统服务，所有人都可以访问共享对象（“同步”权限）。

图 21.12 显示了一个只允许管理员和 SYSTEM 组访问的共享对象。

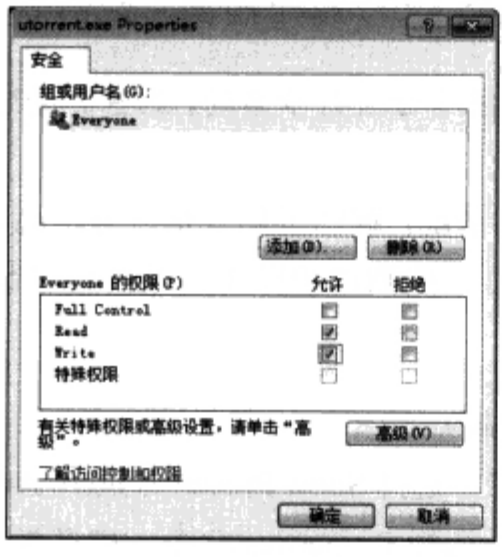


图 21.11 所有人都可以访问的共享对象

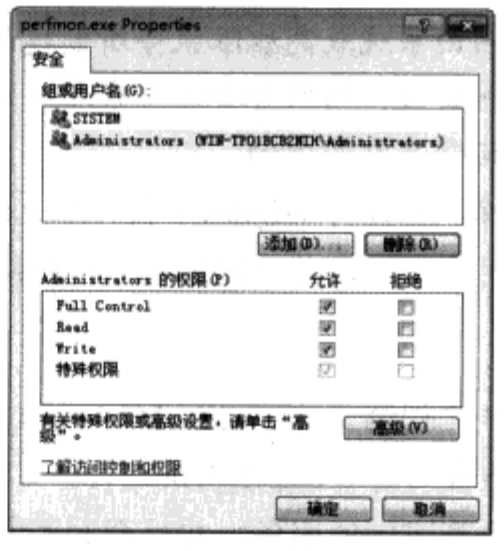


图 21.12 只允许管理员和系统组才能访问的共享对象

（5）如果服务需要创建一个用户应用程序可以访问的文件，可按照以下步骤操作。

① 打开命令行窗口。单击“开始”|“所有程序”|“附件”|“命令提示符”命令。

② 运行 icacls 工具查看文件或者目录的集合权限和 DACL 信息，并且进行修改。

③ 运行 icacls MyFile 来显示名为 MyFile 的文件的访问控制列表。

④ 运行 icacls MyFile /setintegritylevel Medium 将 MyFile 的集合权限级别修改为中级，这将使它可以访问大部分用户应用程序。

21.1.5 用户界面特权隔离（UIPI）

在早期版本的 Windows 中，同一个用户账号下运行的进程共享相同的安全权限。例如，一个线程可以发送窗口信息到另一个线程的窗口。从 Windows Vista 开始，权限级别被附加到进程上（也可以是其他特性），进程不可以将消息窗口发送到具有更高权限级别的进程，尽管它们已运行于同一个用户账号（如当前登录用户）。UIPI 可防止来自其他更高权限的线程创建的 Windows 消息进行恶意代码攻击。

此问题在 UAC（用户账户控制）启用时显而易见。在 UAC 中，一个管理员登录时获取两个令牌对象：第一个是管理员令牌，具有最高权限（类似于 Windows Vista 以前版本的系统账号）；第二个令牌是过滤后的版本，带有标准用户权限。

默认情况下，进程用完整性级别为 Normal（UIPI 级别分为 Low、Normal、High 和 System）的标准用户启动。作为对比，进程使用管理员权限（例如，用户利用“以管理员身份运行”或用“打开方式”命令调用 SheellExecute）获得 High 的权限级别。

这样，系统可以使用单一权限级别运行两种类型的进程（从技术上讲是使用同一用户账号）。UIPI 支持低完整性级别（integrity level）的进程和更高级别的进程进行通信。

使用 Process Explorer 查看权限级别，注意要使用管理员身份来启动，如图 21.13 所示。

选择“Integrity Level”列，如图 21.14 所示。

这里有基于同一图片的两个进程，其中一个用标准用户权限启动，另一个使用管理员权限启动。注意它们是不同的完整性级别，如图 21.15 所示。

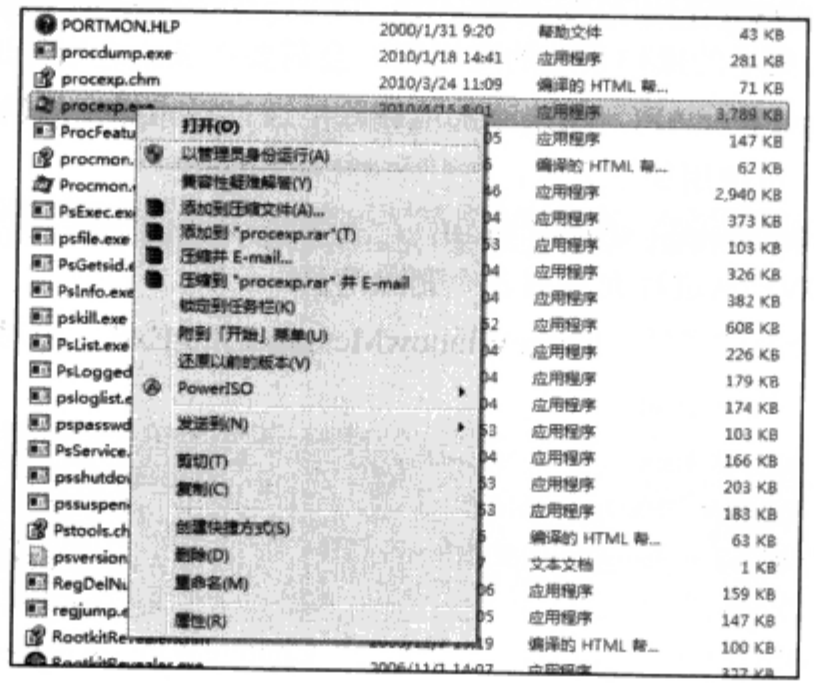


图 21.13　使用管理员特权启动进程浏览器

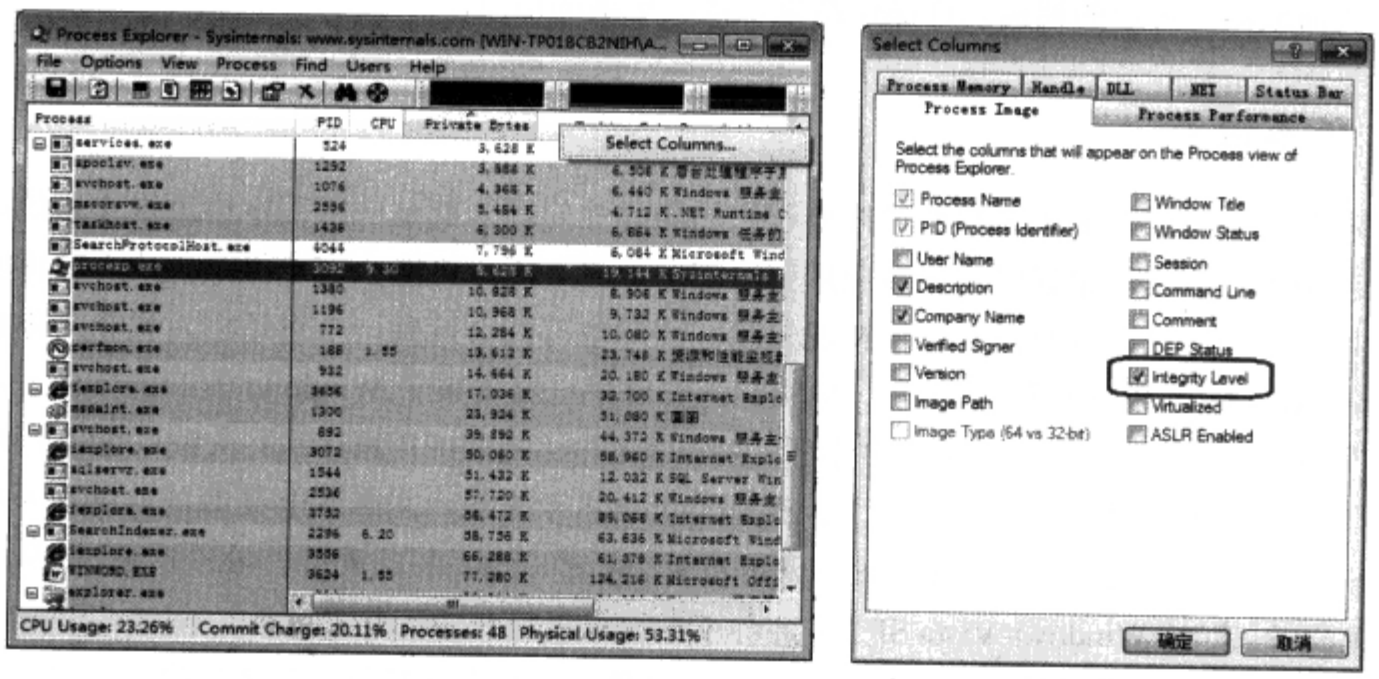


图 21.14　选择 Integrity Level 列

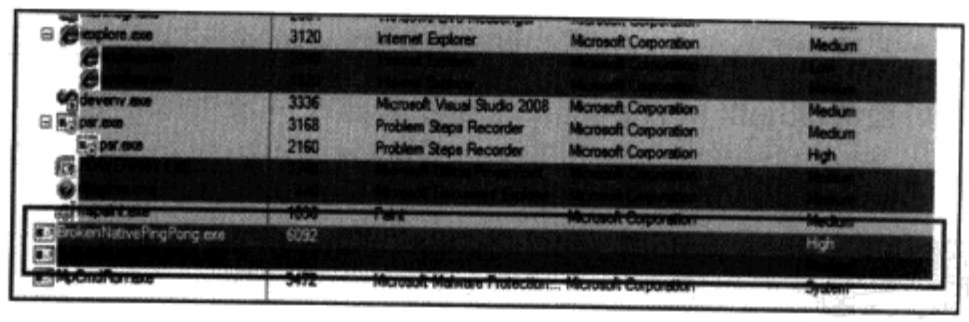


图 21.15　使用不同完整性级别的两个进程

较低权限的应用程序由于受到 UIPI 施加的限制，不能进行如下操作。

- 验证更高权限进程创建的窗口，调用 SendMessage 或 PostMessage 到更高权限进程创建的窗口。
- 使用线程钩子来附加一个更高权限的进程。
- 使用日志钩子（SetWindowsHookEx）来监视更高权限的进程。
- DLL 注入到更高权限的进程。

下列 Windows 消息永远是允许状态：

0x000 - WM_NULL，0x003 - WM_MOVE，0x005 - WM_SIZE，0x00D - WM_GETTEXT，0x00E - WM_GETTEXTLENGTH，0x033 - WM_GETHOTKEY，0x07F - WM_GETICON，0x305 - WM_RENDERFORMAT，0x308 -

```
WM_DRAWCLIPBOARD，0x30D - WM_CHANGECBCHAIN，0x31A - WM_THEMECHANGED，0x313, 0x31B
```

基于比 Windows Vista 更早的操作系统的应用程序会需要在彼此之间传递消息。Windows Vista 引入了 ChangeWindowMessageFilter API，用来添加或删除能通过隔离级别的消息。需要允许消息传递到更高权限的进程的窗口时，可使用如下方法：

```
ChangeWindowMessageFilter(message, MSGFLT_ADD);
```

标记 MSGFLT_REMOVE 从进程允许列表中删除消息。

Windows 7 引入了一个新的函数 ChangeWindowMessageFilterEx()，相对于控制整个进程，它可以在特定窗口控制允许消息。其原型如下：

```
BOOL ChangeWindowMessageFilterEx(
    HWND hWnd, UINT message, DWORD action,
    PCHANGEFILTERSTRUCT pChangeFilterStruct
);
```

事件的参数可以是 MSGFLT_ALLOW （类似于 ChangeWindowMessageFilter 中的 MSGFLT_ADD ）、MSGFLT_DISALLOW （类似于 MSGFLT_REMOVE）和 MSGFLT_RESET，将窗口重新设置为默认筛选器，可选的架构允许操作结果以接收更多的信息。

21.1.6 版本检查

用户和开发人员面对的大多数普通应用程序的兼容性问题，就是应用程序检测操作系统版本后运行失败。当版本检测出现错误时，很多应用程序都会出错。用户可能会有“静默失败”的体验，也就是一个应用程序加载失败并且没有任何情况发生。用户也可能会看到一个对话框，说明“你必须运行在 Microsoft Windows XP 或更高版本”，但实际上计算机已经安装了 Windows 7 。很多由于不严谨的版本检测导致的结果，会给用户带来很多不便。由于版本检测造成应用程序运行失败，通常是两个原因。

- 检测版本的代码存在缺陷（错误）。例如，次版本号降低，主版本号提高（如版本号从 5.1 变为 6.0），或者一些 Service Pack（SP）没有安装，或者已经安装了更新的操作系统（如从 Windows XP SP 2 变为 Windows Vista SP 1），都可能造成应用程序运行失败。
- 故意的阻断。应用程序的开发人员会阻止应用程序运行在没有经过测试的操作系统版本上（建议不要阻止应用程序运行在未来版本的操作系统中）。

当一个应用程序以“不兼容”（由于不严谨的版本检测）的 Windows 版本运行时，它通常会提示一个错误信息，但也有可能静默退出或者执行其他行为。通常，使用版本检测可以让应用程序能够很好地运行，最终用户和 IT 专业人员就能使用一个简单的修复，让应用程序运行在一个早期的 Windows 版本上。

当遇到版本问题时，一个最简单的解决方法就是使用兼容性模式。启用兼容性模式，可以修复一系列的兼容性问题，并为早期 Windows 版本所开发的应用程序提供一个更加兼容的运行时环境。在这些修复的问题中，其中一个就是“版本欺骗”，就是使函数返回一个用户在属性对话框中的“兼容性”选项卡中选中的操作系统版本，而不是计算机的实际 Windows 版本。启用兼容性模式只需下面几步。

（1）右击可执行文件或者可执行文件的快捷方式图标。

（2）选择“属性”命令。

（3）选择“兼容性”选项卡。

（4）选中“以兼容模式运行这个程序”复选框，并选择希望应用程序运行的操作系统，如图 21.16 所示。有些应用程序包含多个可执行文件，可能需要为每一个文件都执行这样的操作。

（5）单击“确定”按钮关闭对话框。

（6）运行应用程序。

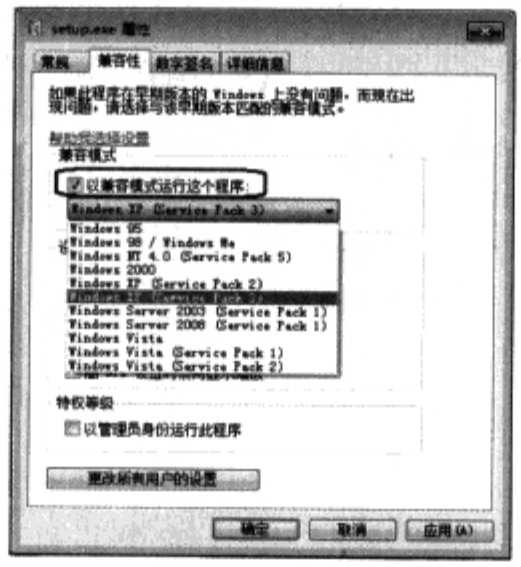


图 21.16　设置程序属性为以兼容模式运行

通过识别当前操作系统版本来决定这个操作系统所拥有的功能，并不是最佳的方法。但是，如果不能将应用程序设计成能够检测指定的功能，那么惟一的确定兼容性的方式就是通过版本检测。那么，可参考下面的方法。

对于本地应用程序，可能需要确保应用程序逻辑能够在更新的 Windows 版本上运行，当版本变更时不要阻止应用程序运行。下面的示例演示了如何在应用中进行正确的版本检查：

```
OSVERSIONINFOEX osvi;
DWORDLONG dwlConditionMask = 0; int op=VER_GREATER_EQUAL;
ZeroMemory(&osvi, sizeof(OSVERSIONINFOEX));
osvi.dwOSVersionInfoSize = sizeof(OSVERSIONINFOEX);
osvi.dwMajorVersion = 5;
osvi.dwMinorVersion = 1; // Windows XP
// 初始化条件掩码
VER_SET_CONDITION( dwlConditionMask, VER_MAJORVERSION, op );
VER_SET_CONDITION( dwlConditionMask, VER_MINORVERSION, op );
if (!VerifyVersionInfo(&osvi, VER_MAJORVERSION | VER_MINORVERSION | VER_SERVICEPACKMAJOR |
VER_SERVICEPACKMINOR, dwlConditionMask)
{ // 显示错误 }
```

检测操作系统的版本来确定操作系统所能提供的功能不是最佳方法，这是因为操作系统可能会在可再发行 DLL 中增加新的功能。与其使用 GetVersionEx 来决定操作系统的平台或者版本号，不如检测操作系统本身所提供的功能更为有效。例如，准备使用 Direct2D 和 DirectWrite 这些 API 和 Windows Vista 所提供的 Ribbon 的 API，那么在使用这些 API 时不需要阻止应用程序。

如果可能，应用程序应该在没有这些功能的情况下依然能够正常运行，只不过是减少一些功能或者降低性能。

对于 Win32，可使用下面的技术。

- 使用 LoadLibrary()来加载一个还没有加载进应用程序的库。如果对已经加载的 DLL 文件（如 kernel32.dll）中的新功能感兴趣，那么可以调用 GetModuleHandle()来获取模块的句柄。如果这些函数都返回空值，将引起一个错误。
- 使用 GetProcAddress()来获取一个函数指针。如果 GetProcAddress()返回一个空值，那么这个函数可能会退出。需要将函数的指针设置成适当的原型。有一些函数，尽管已退出，但也有可能返回一个“未完成”的错误。我们同样需要检查这些错误。

下面的代码示例就展示了这一技术：

```
//定义函数指针类型
typedef BOOL (WINAPI *SetWaitableTimerExProc)(
    __in   HANDLE hTimer,
    __in   const LARGE_INTEGER *lpDueTime,
    __in   LONG lPeriod,
    __in   PTIMERAPCROUTINE   pfnCompletionRoutine,
    __in   LPVOID   lpArgToCompletionRoutine,
    __in   PREASON_CONTEXT WakeContext,
    __in   ULONG TolerableDelay
);
//获取模块句柄
HMODULE hKernel32Module = GetModuleHandle(_T("kernel32.dll"));
//获取函数地址
SetWaitableTimerExProcpFn = (SetWaitableTimerExProc)GetProcAddress(hKernel32Module, "SetWaitableTimerEx");
if (pFn != NULL)
    pFn(hTimer, &liDueTime,1000, NULL, NULL,&reasonContext, 1000);
```

21.2 Windows 7 系统专题

Windows 7 系统除了提供更炫的用户界面、更友好的操作接口外，还对原来系统的多个部分进行优化，这些优化主要有任务栏、Shell 库、后台服务、设备和性能。下面分别进行介绍。

21.2.1 实现超级任务栏

新的 Windows 7 任务栏体现了 Windows 界面的不断改进。新的任务栏集成了很多最终用户的使用环境。

- 运行应用程序。
- 实现了正在运行的应用程序和运行在同一个应用程序中的窗口之间的切换。
- 对用户最近使用的/频繁使用的应用程序进行管理。
- 访问公用应用程序任务。
- 通过任务栏按钮报告进度和状态。

在 Windows 7 平台上，新的任务栏是最终用户开始管理活动的主要地方。因此，在时尚的 Windows 7 应用程序中对新任务栏特性进行集成就成了一项非常重要的工作。

将锁定功能集成到新的任务栏中为用户提供了一个更加清晰的 Windows 用户界面，方便用户查找关键特性、任务和目标，以适应大多数最终用户的应用环境。

下面简单介绍如何通过编程实现对任务栏的操作，包括覆盖图标、使用进度条、使用缩略图工具栏、使用任务栏快捷菜单。

1. 覆盖图标

使用由 ITaskbarList3 接口提供的 SetOverlayIcon()函数实现在应用程序的任务栏按钮上覆盖一个指定的图标，用于指示应用程序的状态或向用户醒目显示。这个函数的原型如下：

```
HRESULT SetOverlayIcon(
    [in]   HWND hwnd,
```

```
    [in]  HICON hIcon,
    [in]  LPCWSTR pszDescription
);
```

- 参数 hwnd 指定将被图标覆盖的任务栏按钮关联的窗口句柄，这个句柄必须属于调用进程的关联按钮应用，而且必须是合法的 HWND，否则这个调用将被忽略（不起任何作用）。
- 参数 hIcon 指定用于覆盖任务栏按钮的图标，必须是在 96 DPI 设置下、大小为 16 像素×16 像素的小图标。如果任务栏已经被某个图标覆盖，那么函数调用将使用当前图标来覆盖。

这个参数可以指定为 NULL，如何处理 NULL 取决于当前任务栏按钮代表单一窗口还是代表一组窗口。当为单一窗口时，NULL 表示移除任务栏上的覆盖图标（如果之前被覆盖）；当为一组窗口时，则只移除当前窗口的覆盖图标，而以前的覆盖图标仍可用，并成为任务栏的当前覆盖图标。

当 hIcon 对应的图标资源不再使用后，应该由调用该函数的应用负责释放。一般在调用完 SetOverlayIcon()函数后就可以释放图标资源，因为任务栏会复制图标，并使用这个备份（而非原图标）。

- 参数 pszDescription 为一个字符串指针，显示说明信息。

如果函数调用成功，将返回 S_OK，否则返回一个 HRESULT 类型的错误码。

下面的代码演示了如何实现在任务栏上覆盖图标：

```
ICON hIcon = NULL; // for IDM_OVERLAY_CLEAR
Icon = LoadIcon(g_hInstance, MAKEINTRESOURCE(IDI_OVERLAY1));
//设置窗口的覆盖图标，可以为 NULL
g_pTaskbarList->SetOverlayIcon(hWnd, hIcon, NULL);
if (hIcon) {
    //不再使用时，释放图标
    DestroyIcon(hIcon);
}
```

在设置图标之前，需要先设置 ITaskbarList3 *g_pTaskbarList = NULL，然后调用 CoCreateInstance()方法创建：

```
CoCreateInstance(
    CLSID_TaskbarList,
    NULL,
    CLSCTX_INPROC_SERVER,
    IID_PPV_ARGS(&g_pTaskbarList)
);
```

运行程序，可以看到如图 21.17 所示的效果，左边为没有覆盖图标时的应用程序，而右边为任务栏内的应用程序被覆盖了一个红色图标。

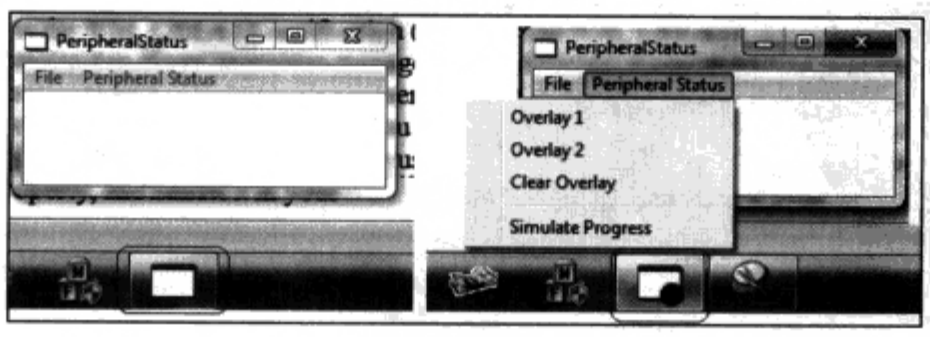


图 21.17　任务栏内覆盖图标运行结果

而对于.NET 框架内的托管代码，通过设置 Taskbar 类的 OverlayImage 属性可以很方便地实现覆盖图标：

```
Taskbar.OverlayImage = new OverlayImage(TaskbarDemo.Properties.Resources.Red, "Red");
```

2．使用进度条

在 Windows 7 中，可以在应用程序的图标中编程控制进度条的行为。对于本地代码，通过 ITaskbarList3 接口提供的 SetProgressState()和 SetProgressValue()函数可以实现进度条控制，前者用于设置进度条的状态（如不确定或出错），后者用于设置进度值。这两个函数的原型如下：

```
HRESULT SetProgressState(
    [in]  HWND hwnd,
    [in]  TBPFLAG tbpFlags
);
```

- 参数 hwnd 指定将显示进度条的任务栏按钮关联的窗口句柄。
- 参数 tbpFlags 指定进度条的状态标志，只能取表 21.2 中列出的值之一。

表 21.2　　进度条的状态标志

状 态 标 志	含　义
TBPF_NOPROGRESS（0x00000000）	停止显示进度，并返回到 Normal 状态
TBPF_INDETERMINATE（0x00000001）	不确定状态，进度条仍活跃，但是无法确定进度何时结束
TBPF_NORMAL（0x00000002）	进度条正常显示，根据指定的结束时刻确定从左到右依次增长特定的长度
TBPF_ERROR（0x00000004）	出错状态，进度条将变为红色，这也是确定状态
TBPF_PAUSED（0x00000008）	暂停状态，进度条变为黄色。可以由用户指定继续进行，也是一个确定状态

```
HRESULT SetProgressValue(
    [in]  HWND hwnd,
    [in]  ULONGLONG ullCompleted,
    [in]  ULONGLONG ullTotal
);
```

- 参数 hwnd 指定任务栏关联的窗口句柄。
- 参数 ullCompleted 指定已经完成的进度部分所对应的进度条的值。
- 参数 ullTotal 指定操作完成时进度条所处的值。

下面的代码演示了如何在任务栏中使用进度条：

```
case WM_TIMER:
g_nProgress++;
if (g_nProgress == 1)
{
    //第一次处理，设置进度条处于不确定状态，用于模拟一个后台计算无法确定
    //计算量的大小
    g_pTaskbarList->SetProgressState(hWnd, TBPF_INDETERMINATE);
}
 else if (g_nProgress == MAX_PROGRESS_IND)
{
     //设置进度条为正常状态，表示将开始显示正常的进度
     g_pTaskbarList->SetProgressValue(hWnd, 0, MAX_PROGRESS_NORMAL);
     g_pTaskbarList->SetProgressState(hWnd, TBPF_NORMAL);
}
else if (g_nProgress > MAX_PROGRESS_IND)
{
    if (g_nProgress - MAX_PROGRESS_IND <= MAX_PROGRESS_NORMAL)
    {
        //模拟后台计算的过程并显示相应的进度
```

```
            g_pTaskbarList->SetProgressValue(
                hWnd,
                g_nProgress - MAX_PROGRESS_IND,
                MAX_PROGRESS_NORMAL);
        }
        else
        {
            //进度完成，停止进度并回到正常状态
            KillTimer(hWnd, g_nTimerId);
            g_nTimerId = 0;
            g_pTaskbarList->SetProgressState(hWnd, TBPF_NOPROGRESS);
            MessageBox(hWnd, L"Done!", L"Progress Complete", MB_OK);
        }
    }
    break;
```

对于托管代码，使用 Taskbar 类提供的 ProgressBar 属性可以设置当前进度的值、进度最大值、进度状态等，例如：

```
Taskbar.ProgressBar.State =
    (TaskbarButtonProgressState)Enum.Parse(
            typeof(TaskbarButtonProgressState),
            (string)comboBoxProgressBarStates.SelectedItem);
if (Taskbar.ProgressBar.State != TaskbarButtonProgressState.Indeterminate)
    Taskbar.ProgressBar.CurrentValue = progressBar1.Value;
```

3. 使用缩略图工具栏

首先简单介绍缩略图的概念。缩略图是一个很有用的特性，它允许用户在应用程序不处于焦点甚至不可见时仍可以访问应用程序的功能，而不用先选中那个应用（使之成为活动窗口）。图 21.18 演示了 Windows 多媒体播放器的缩略图工具栏。

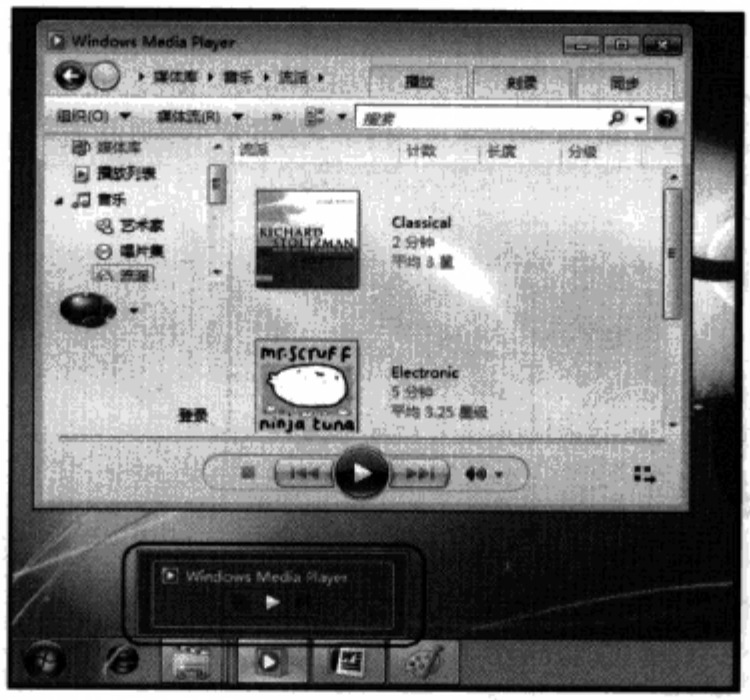


图 21.18　Windows 7 下的多媒体播放器的缩略图工具栏

从中可以看到，将鼠标指针移动到多媒体播放器图标上时，将在上方显示 3 个按钮：播放/暂停，下一首歌，上一首歌。单击这 3 个按钮就可以实现对播放器的操作（而不必先选中应用窗口）。

使用缩略图工具栏需要满足以下几个条件。

（1）缩略工具栏内至多显示 7 个按钮，而且大小必须为 16 像素×16 像素（对于高 DPI 设置，限制为 24 像素×24 像素）

（2）可以为应用的每个活动窗口设置一个缩略图工具栏（即多个窗口可以分别设置不同的缩略图工具栏）。

（3）向缩略图工具栏内添加或删除按钮必须在构造任务栏之前进行，一旦任务栏构造完毕就不允许添加或删除按钮，但是可以将按钮隐藏。

对于本地代码应用，可以使用 ITaskbarList4 接口提供的方法来对任务栏动态执行添加、删除元素操作。对于缩略图工具栏进行操作，需要使用下面的几个方法。

- ThumbBarAddButtons(): 向缩略图工具栏添加指定的一组按钮。
- ThumbBarSetImageList(): 指定缩略图工具栏的图像列表。
- ThumbBarUpdateButtons(): 根据 Windows 当前的状态，对缩略图工具栏内的按钮执行显示、隐藏、使能或禁止操作。

在执行操作之前，首先需要确定任务栏可用。为了确认任务栏是否可用，需要使用 RegisterWindowMessage()方法注册 TaskbarCreatedMessage 消息，然后等待这个消息到达 Win32 的 WndProc()消息处理函数，一旦应用接收到这个消息，就可以创建缩略图工具栏的按钮了。而你缩略图工具栏内创建的按钮的处理消息（如单击）将会以 WM_COMMAND 消息的形式到达 WndProc()函数，这时 wParam 参数的低字（WORD）代表发送消息的按钮 ID，指示哪个按钮被单击，而高字则直接被设置为 THBN_CLICKED。下面的代码演示了这个处理过程：

```
LRESULT CALLBACK WndProc(
                        HWND hWnd, UINT message,
                        WPARAM wParam, LPARAM lParam)
{
    static UINT s_uTBBC = WM_NULL;
    if (s_uTBBC == WM_NULL)
    {
        //注册 TaskbarButtonCreated 消息
        s_uTBBC = RegisterWindowMessage(L"TaskbarButtonCreated");
        //允许创建任务栏按钮，允许向按钮发送命令
        ChangeWindowMessageFilter(s_uTBBC, MSGFLT_ADD);
        ChangeWindowMessageFilter(WM_COMMAND, MSGFLT_ADD);
    }
    if (message == s_uTBBC) //wmTaskbarButtonCreated)
    {
        //一旦获取 TaskbarButtonCreated 消息，就可以创建缩略图工具栏
        CreateThumbnailToolbar(hWnd);
    }
    else switch (message)
    {
        case WM_COMMAND:
        {
            int const wmId = LOWORD(wParam);
            switch (wmId)
            {
                case IDTB_BUTTON1:
                case IDTB_BUTTON2:
```

```
                case IDTB_BUTTON3:
                {
                    WCHAR szMsg[100];
                    StringCchPrintf(
                        szMsg, ARRAYSIZE(szMsg),
                        L"Thumbnail toolbar button clicked,
                        ID=%d", wmId);
                    MessageBox(hWnd, szMsg, L"Application", MB_OK);
                    break;
                }
                case IDM_EXIT:
                    DestroyWindow(hWnd);
                    break;
                default:
                    return DefWindowProc(hWnd, message, wParam, lParam);
            }
            break;
        }
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

以上代码只是很简单地处理 WM_COMMAND 消息，获取被单击按钮的 ID（通过参数 wParam 的低字），然后以消息框显示，并没有执行任何有用的操作，这只是一个演示例子。

下面来看如何创建指定缩略图工具栏。在上面的代码中，通过调用 CreateThumbnail Toolbar()函数实现。下面是这个函数的实现代码：

```
HRESULT CreateThumbnailToolbar(HWND hWnd)
{
    ITaskbarList4 *pTaskbarList;
    HRESULT hr = CoCreateInstance(
        CLSID_TaskbarList,
        NULL,
        CLSCTX_INPROC_SERVER,
        IID_PPV_ARGS(&pTaskbarList));
    if (SUCCEEDED(hr))
    {
        hr = pTaskbarList->HrInit();
        if (SUCCEEDED(hr))
        {
            //指定缩略图工具栏使用的按钮位图，这里使用系统小图标宽度，这样对 DPI 是友好的
            struct
            {
                PCWSTR pbmp;
                int cx;
```

```
    }
    const bitmaps[3] =
    {
        { MAKEINTRESOURCE(IDB_BUTTONIMAGES_96), 16 },
        { MAKEINTRESOURCE(IDB_BUTTONIMAGES_120), 20 },
        { MAKEINTRESOURCE(IDB_BUTTONIMAGES_144), 24 }
    };
    int const cxButton = GetSystemMetrics(SM_CXSMICON);
    int iButtons = 0;
    for (int i = 0; i < ARRAYSIZE(bitmaps); i++)
    {
        if (bitmaps[i].cx <= cxButton)
        {
            iButtons = i;
        }
    }
    HIMAGELIST himl = ImageList_LoadImage(
        g_hInstance,
        bitmaps[iButtons].pbmp,
        bitmaps[iButtons].cx,
        0,
        RGB(255,0,255),
        IMAGE_BITMAP,
        LR_CREATEDIBSECTION);
    if (himl)
    {
        hr = pTaskbarList->ThumbBarSetImageList(hWnd, himl);
        if (SUCCEEDED(hr))
        {
            THUMBBUTTON buttons[3] = {};
            //第一个按钮
            buttons[0].dwMask = THB_BITMAP | THB_TOOLTIP | THB_FLAGS;
            buttons[0].dwFlags = THBF_ENABLED | THBF_DISMISSONCLICK;
            buttons[0].iId = IDTB_BUTTON1;
            buttons[0].iBitmap = 0;
            StringCchCopy(buttons[0].szTip,ARRAYSIZE(buttons[0].szTip), L"Button 1");
            //第二个按钮
            buttons[1].dwMask = THB_BITMAP | THB_TOOLTIP | THB_FLAGS;
            buttons[1].dwFlags = THBF_ENABLED | THBF_DISMISSONCLICK;
            buttons[1].iId = IDTB_BUTTON2;
            buttons[1].iBitmap = 1;
            StringCchCopy(buttons[1].szTip,ARRAYSIZE(buttons[1].szTip), L"Button 2");
            // 第三个按钮
            buttons[2].dwMask = THB_BITMAP | THB_TOOLTIP | THB_FLAGS;
            buttons[2].dwFlags = THBF_ENABLED | THBF_DISMISSONCLICK;
            buttons[2].iId = IDTB_BUTTON3;
            buttons[2].iBitmap = 2;
            StringCchCopy(buttons[2].szTip,ARRAYSIZE(buttons[2].szTip), L"Button 3");
            //为缩略图工具栏加入按钮列表
```

```
                    hr = pTaskbarList->ThumbBarAddButtons(
                        hWnd,
                        ARRAYSIZE(buttons),
                        buttons);
                }
                ImageList_Destroy(himl);
            }
        }
        //释放 ITaskbarList3
        //缩略图工具栏仍会保持
        pTaskbarList->Release();
    }
    return hr;
}
```

首先需要创建一个 ITaskbarList4 COM 接口，接着调用 HrInit()来对任务栏列表对象进行初始化。在对 ITaskList 对象进行操作之前必须先进行初始化。然后设置按钮相关的位图，必须确认位图的大小为一个小图标尺寸。然后调用 ThumbBarSetImageList()函数来设置按钮的图标。接着创建包含 3 个 THUMBBUTTON 的数组，对每个按钮设置位图、标志，还为每个按钮设置不同的 ID 来区分到底哪个按钮被单击，这个 ID 与 WM_COMMAND 消息对应。最后调用 ThumbBarAddButtons()函数来向缩略图工具栏添加按钮。这个程序的运行结果如图 21.19 所示。

4．使用任务栏快捷菜单

作为开发人员，可以认为快捷菜单就是应用程序的“迷你”版开始菜单，在快捷菜单中可以放置应用程序经常使用的目标或执行的任务，这为用户提供了一种快速执行任务及快速启动应用的途径。图 21.20 为一个快捷菜单，从这里可以加载一个新的应用或关闭应用。

除了可以直接使用 Windows 默认的快捷菜单外，还可通过 ICustomDestinationList 接口创建自定义的快捷菜单。这个接口主要提供以下函数。

- AppendCategory()：定义一个自定义类别以及包含在快捷菜单中的项目。
- AppendKnownCategory()：指定经常或最近打开的类别将被包含到快捷菜单中。
- BeginList()：为快捷菜单初始化一个创建会话。
- CommitList()：表明由 BeginList 初始化的快捷菜单已经完成，可以显示。

图 21.19　向缩略图工具栏添加按钮的运行结果

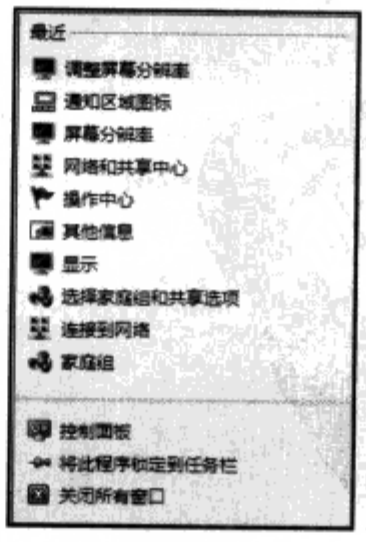


图 21.20　弹出列表

下面的代码演示了如何创建一个自定义快捷菜单，并在上面添加元素：

```
void CreateJumpList()
```

```
{
    ICustomDestinationList *pcdl;
    HRESULT hr = CoCreateInstance(
        CLSID_DestinationList,
        NULL,
        CLSCTX_INPROC_SERVER,
        IID_PPV_ARGS(&pcdl));
    if (SUCCEEDED(hr))
    {
        //必须先为快捷菜单设置 App ID
        hr = pcdl->SetAppID(c_szAppID);
        if (SUCCEEDED(hr))
        {
            UINT uMaxSlots;
            IObjectArray *poaRemoved;
            hr = pcdl->BeginList(
                &uMaxSlots,
                IID_PPV_ARGS(&poaRemoved));
            if (SUCCEEDED(hr))
            {
                hr = _AddCategoryToList(pcdl, poaRemoved);
                if (SUCCEEDED(hr))
                {
                    pcdl->CommitList();
                }
                poaRemoved->Release();
            }
        }
    }
}
```

首先要为快捷菜单指定 App ID，然后调用 BeginList()函数为其初始化创建会话。这个函数返回允许添加元素的最大个数，默认值为 10。接着调用_ AddCategoryToList()函数将元素添加到自定义类别中。下面为_ AddCategoryToList()函数的实现代码：

```
HRESULT _AddCategoryToList(ICustomDestinationList *pcdl, ObjectArray *poaRemoved)
{
    IObjectCollection *poc;
    HRESULT hr = CoCreateInstance
        (CLSID_EnumerableObjectCollection,
        NULL,
        CLSCTX_INPROC_SERVER,
        IID_PPV_ARGS(&poc));
    if (SUCCEEDED(hr))
    {
        for (UINT i = 0; i < ARRAYSIZE(c_rgpszFiles); i++)
        {
            IShellItem *psi;
            if (SUCCEEDED(SHCreateItemInKnownFolder(
                FOLDERID_Documents,
```

```
                    KF_FLAG_DEFAULT,
                    c_rgpszFiles[i],
                    IID_PPV_ARGS(&psi))))
                {
                    if(!_IsItemInArray(psi, poaRemoved))
                    {
                        poc->AddObject(psi);
                    }
                    psi->Release();
                }
            }
            IObjectArray *poa;
            hr = poc->QueryInterface(IID_PPV_ARGS(&poa));
            if (SUCCEEDED(hr))
            {
                pcdl->AppendCategory(L"Custom category", poa);
                poa->Release();
            }
            poc->Release();
        }
        return hr;
}
```

这里调用 pcdl->AppendCategory(L"Custom category", poa);向快捷菜单添加一个名为 Custom category 的类别，最后调用 CommitList()函数来结束由 BeginList()函数开始的会话。这个程序的运行结果如图 21.21 所示。

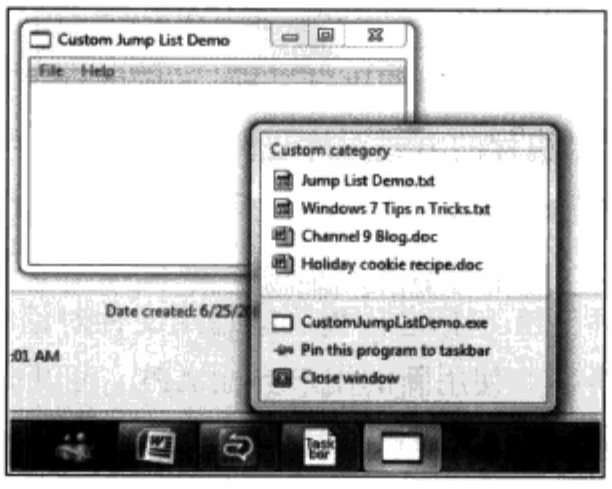


图 21.21　用户自定义弹出列表运行结果

21.2.2　实现 Shell 库

Windows 7 引入了一种叫做库的新概念，是用户数据的新的入口点。用户可以轻松地以集合数据的方式找到并且管理自己的数据，而且这些数据可能位于计算机中的多个不同的位置。库代替了早期 Windows 版本中的固有文件夹（如我的文档/图片/音乐）。Shell 库的 API 提供给应用程序一种简单的方法来对该库进行交互操作。应用程序可以创建、交互并且像及环境中的一级元素一样支持该库。

在 Windows 7 中，Shell 库的概念就是，通过允许用户对其文档库文件夹结构进行完全控制权限，试图解决用户在 PC 上到处存放数据的问题。换而言之，在 Windows 7 中，用户可以在文档库中来定

义哪个文件夹用来存放哪类文件。也可以说，Shell 库是一种用户自定义的、合理的、代表用户数据的文件夹集合。库中所包含的文件夹其实就是告诉 Windows，用户的重要数据在哪里存放。系统将为这些文件夹建立索引，以更快地查询检索，在基于文件属性和元数据的 Windows 资源管理器中具有更丰富的视图展示能力。

在早期的 Windows 版本中，每一个应用程序都有属于自己的属性库。例如，Windows Media Player 与 iTunes 相比，拥有不同的作用域集合，并且两者都不是与 Music 文件夹一致的。使用 Shell 库的 API，应用程序就可以定义并且共同使用用户自定义的那个区域。

Shell 库中同样也可以包含网络文件夹，用户无论是在家还是在单位，这一功能都会给用户带来便利。用户无论何时打开一个文件对话框，都能获得所有可用的库指向的集合视图，如图 21.22 所示。

图 21.22　Windows 7 操作系统下的库

在 Windows 7操作系统中，可以手动对库进行管理，如进行库的创建、文件夹的添加和删除等。但是作为程序员，更加关心如何以编程的方式对库进行操作。为了帮助我们在应用程序中使用库这种新的文件管理方式，Windows 7 提供了一组 API 用于库的编程开发。

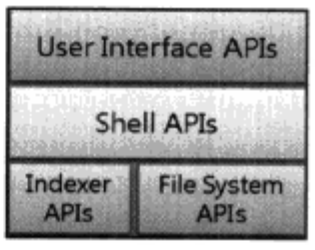


图 21.23

其中，位于最顶层的用户界面 API 包括之前提到的通用文件对话框 CFD、导航栏树形控件等，如图 21.23 所示。使用新的用户界面 API，可以调用支持库的新版通用文件对话框，从而让应用程序在打开或者保存文件时，可以支持库这一新的特性。

除了用户界面 API，我们重点关注的是位于中间的 Shell API。利用 Shell API，可以对库进行管理。这组 Shell API 包装了多个 COM 对象，利用这些对象可以对库进行操作。其中几个比较常用的对象如下。

- IShellLink 对象代表到文件、文件夹或可执行文件的一个链接。
- IShellFolder 对象实际上代表一个文件夹对象，可以通过遍历 IShellFolder 对象来访问这个文件夹下的所有内容，检索文件夹中的项目显示名称，分析文件夹的显示名称，获取文件夹下的项目 ID 列表等。
- IShellLibrary 是 Windows 7 新引入的一个对象，它与一个库相对应。通过这个对象，可以对库进行各种操作。

另外，Shell API 也提供了多个辅助函数用于对库进行操作。例如，创建库，打开一个已经存在的库，添加文件夹到库中或者删除一个库中的文件夹，获得一个库的文件夹列表，获得或者设置库的各

种选项，获得或者设置库的图标等。

Shell API 是基于 COM 的，为了使用库内的对象、接口及函数，需要包含以下两个头文件：

```
#include <shobjidl.h>
#include <objbase.h> // IID_PPV_ARGS macro
```

要创建一个新库，可调用 SHCreateLibrary()函数实现：

```
HRESULT SHCreateLibrary(REFIID riid,void **ppv );
```

使用宏创建库的示例：

```
IShellLibrary *pIShellLib;
SHCreateLibrary(IID_PPV_ARGS(&pIShellLib));
```

表 21.3 列出了常用的对库进行操作的函数。

表 21.3　　常用的对库进行操作的函数

函 数 名	说　明
SHAddFolderPathToLibrary ()	向库添加一个文件夹
SHCreateLibrary ()	创建一个 IShellLibrary 对象，对应新建一个库
SHLoadLibraryFromItem()	创建并从指定的库定义文件加载 IShellLibrary 对象
SHLoadLibraryFromKnownFolder ()	创建并从指定的文件夹加载一个 IShellLibrary 对象
SHLoadLibraryFromParsingName()	创建并从指定的路径加载一个 IShellLibrary 对象
SHRemoveFolderPathFromLibrary ()	从库中删除一个文件夹
SHResolveFolderPathInLibrary ()	恢复库中被移动过或被改名的指定目标
SHSaveLibraryInFolderPath ()	将一个 IShellLibrary 对象保存到磁盘

其中 IShellLibrary 接口是对 Shell 库进行操作的最重要的接口，主要包含下面几种操作。

- 提交（Commit）：将库的更新提交到文件中。
- Set/Get 操作：设置或读取属性信息。
- SetFolderType：设置文件夹模板（如 Pictures 或 Misic）。

下面的代码演示了如何在本地代码中创建一个新库并向库添加文件夹：

```
IShellLibrary *pIShelLibrary;
HRESULT hr = SHCreateLibrary(IID_PPV_ARGS(&pIShelLibrary));
if (SUCCEEDED(hr))
{
    IShellItem *pIShellItem;
    SHAddFolderPathToLibrary(pIShelLibrary,L"C:\\Users\\Public\\Documents");
    hr = pIShelLibrary->SaveInKnownFolder(FOLDERID_Libraries,
            L"My New Library",
            LSF_MAKEUNIQUENAME,
            &pIShellItem);
    pIShellItem->Release();
    pIShelLibrary->Release();
}
```

对于.NET 框架上的托管代码，则是通过从 ShellContainer 继承的类 ShellLibrary 来实现：

```
public sealed class ShellLibrary : ShellContainer,
IList<FileSystemFolder>
{
    public ShellLibrary(…);
    public static ShellLibrary Copy(…);
    public static ShellLibrary Load(…);
```

```
        public static IKnownFolderLibrariesKnownFolder { get; }
        public override string Name {get;}
        public IconReferenceIconResourceId {get; set;}
        public LibraryFolderTypeLibraryType {get; set;}
        public GuidLibraryTypeId {get; set;}
        public string DefaultSaveFolder {set; get;}
        public boolIsPinnedToNavigationPane {get; set;}
        public boolShowManageLibraryUI(…);
        public void Add(…)
        public void Close();
        …
    }
    using (ShellLibrary library =new ShellLibrary(name, true))
    {
        library.Add(folderPath);
    }
```

这里首先定义 ShellLibrary 类，然后使用 using 语句创建一个新库，并调用 library.Add 函数来向库添加一个文件夹。在 using 语句内还可以实现更多的设置，例如下面的代码：

```
using (ShellLibrary lib = ShellLibrary.Load("Documents", false))
{
    lib.Add(@"C:\");
    lib.IsPinnedToNavigationPane = true;
    lib.DefaultSaveFolder = @"C:\";
}
```

21.2.3 实现后台服务

与一般的用户应用程序不同，服务是整合到操作系统中的一种运行机制，服务的启动与用户的上下文无关。可以配置一个服务在系统启动时自动开启，并直到关机后才退出，所以即使没有用户登录，这些服务仍在运行。

随着 Windows 越来越复杂，越来越多的后台服务被加入到操作系统中。实际上，很多服务并不是用户所需要的，它们只是在做一些无用的操作，但是这些服务却占用系统资源，影响用户的正常使用。

Windows 7 和 Windows Vista 在后台处理方面有了很多改进。如今实现有效后台处理的挑战包括。

- 性能——启动延迟，登录延迟，关闭延迟；后台处理干扰前台处理。
- 耗电。
- 安全——攻击面增多。

Windows 7 后台服务和计划任务采用各种机制最大程度地减小耗电量，减少系统攻击面，并提高应用程序和系统性能。这些机制包括：

- 服务请求的安全权限；
- 服务 SID；
- 延迟自动启动服务；
- 触发器启动服务；
- 计划任务条件和设置。

由于如今的 Windows 中存在大量服务，后台处理必须满足安全性、性能和耗电要求。

解决上述问题的方法是将改变服务的这种永远运行的状态，变为其他类型的后台活动，如被调度

任务或触发器启动服务。下面重点介绍 Windows7 中引入的触发器启动服务。

所谓触发器启动服务是指一个常规的服务可以被配置成仅当被某事件触发后才运行或停止运行。也就是说，只有当满足用户定义的特定条件（如当第一个网络 IP 地址可用）时才会启动或停止服务。只有当真正需要时，Windows 7 才会触发服务；而一旦服务处于空闲状态，则将服务关闭。

下面列出可用于启动给定服务的触发器：

- 设备接口到达或离开；
- 加入或离开一个域；
- 打开或关闭一个防火墙端口；
- 群策略变化；
- 第一个 IP 地址可用/最后 IP 地址离开；
- 自定义事件——ETW。

一个触发器包括以下几部分：触发器事件类型、触发器事件子类型、触发行为、一个或多个触发器特定的数据对象。

在 Windows 7 中，可以使用 ChangeServiceConfig2()函数来配置触发器服务的信息，使用 QueryServiceConfig2()函数来查询服务状态。

ChangeServiceConfig2()函数的原型如下：

```
BOOL WINAPI ChangeServiceConfig2(
  __in SC_HANDLE hService,
  __in DWORD dwInfoLevel,
  __in_opt   LPVOID lpInfo
);
```

- 参数 hService 指定服务句柄。
- 参数 dwInfoLevel 指定被修改的配置信息。
- 参数 lpInfo 指向将被设置的配置信息的新值，如果值为 NULL，那么服务对应的信息值不变。

下面的代码演示了如何配置一个触发器服务：

```
SERVICE_TRIGGER trigger = {0};
trigger.dwTriggerType = SERVICE_TRIGGER_TYPE_IP_ADDRESS_AVAILABILITY;
trigger.dwAction = SERVICE_TRIGGER_ACTION_SERVICE_START;
trigger.pTriggerSubtype = (GUID*)&NETWORK_MANAGER_FIRST_IP_ADDRESS_ARRIVAL_GUID;
SERVICE_TRIGGER_INFO info = {0};
info.cTriggers = 1;
info.pTriggers = &trigger;
ChangeServiceConfig2 (hService, SERVICE_CONFIG_TRIGGER_INFO, &info);
```

21.2.4　开发基于 Windows 7 的设备与性能应用

在程序开发过程中，经常会出现一些难以理解的现象，例如程序突然性能很差或者占用很大的内存空间，而通过常规的调试工具不容易发现问题所在。Windows 7 系统提供了性能监视和事件日志用于对程序执行过程中的行为进行记录，程序员通过分析这些记录，就可以很快发现问题所在。下面分别介绍这两种检测性能的机制。

1．性能监视

性能监视是 Windows NT 引入的一种系统功能。从 Windows NT 以后，Windows 操作系统总是集成各种性能监视工具，提供有关操作系统当前运行状况的信息，针对各种性能对象提供了数百个性能

计数器。性能对象，就是被性能计数器监视的对象，通常比较关心的监视对象主要有 Processor、Process、Memory、TCP/UDP/IP/ICMP、Physical Disk 等。性能计数器通常提供操作系统、应用程序、服务、驱动程序等的性能相关信息，以此来分析系统瓶颈和对系统及应用程序性能进行诊断和调优。除了针对操作系统本身，通过性能计数器机制，也可以在应用程序或者是操作系统组件中向性能监视器（Performance Monitor）报告一些与性能有关的统计信息，以此来查看软件的性能信息，对其进行诊断和调优。

在 Windows 7 中，微软对性能计数器做了进一步的改进和优化。例如，采用了新的 2.0 版本的核心模块 API，采用 XML 定义，提供更加强大的性能、更高的可扩展性和健壮性，增加了多个系统计时器等。同时，为了方便系统管理员进行管理，还增加了 PowerShell 对计时器日志文件的处理功能等，使得性能计数器的功能大大增强。

通过性能监视器，可以直观地获得系统所提供的各种性能监视的数据，了解系统的运行状态。但是，如果想在应用程序中以编程的方式获得这些性能计数器的信息，该怎样处理呢？实际上，在 Visual Studio 2010 中，微软提供了很多性能计时器供我们在应用程序中使用。在 Visual Studio 2010 的 Server Explorer（服务器浏览器）中，可以看到系统提供的很多性能计数器。可以在应用程序中简单地以控件的形式使用这些性能计数器，以获取系统的各种消息。

在托管代码中，位于 System.Diagnostics 命名空间中的 PerformanceCounter 类代表一个 Windows NT 中的性能计数器组件。

计数器提供了收集应用程序或系统性能数据的机制，在注册表中保存了所有可用的计数器，常见的计数器有处理器利用率、内存使用率、从网络接口发送或接收的数据量等。

每个计数器都由它的名字和所在路径惟一标识（这与文件系统中文件标识类似）。计数器标识（信息）包括 4 个组成部分：计算机名字、类别（category）、类别实例和计数器名字。其中计数器信息必须包括类别或者称为性能对象，用于表明计数器记录的内容（CPU 利用率或内存使用率）。

下面的代码演示了如何使用普通的计数器来获取性能数据，这里的计数器同时收集应用程序和处理器的信息：

```
using namespace System;
using namespace System::Diagnostics;
using namespace System::Threading;
using namespace System::Collections;
using namespace System::IO;
 int   main(System::array<System::String^>^   args)
  {
     PerformanceCounter^   privBytes = gcnew PerformanceCounter("Process", "Private Bytes", "PerfCounters.vshost");
     PerformanceCounter^   ioRead = gcnew PerformanceCounter("Process",
         "IO Write Operations/sec", "PerfCounters.vshost");
     PerformanceCounter^   availBytes =gc new PerformanceCounter("Memory", "Available Bytes", "");
     PerformanceCounter^   diskQueue = gcnew PerformanceCounter("PhysicalDisk",
         "Avg. Disk Queue Length", "_Total");
     StreamWriter^ log = File::CreateText("..\\Debug\\perfLog.txt");
     Console::WriteLine("Private Bytes\tIO Write Operations/sec\tAvailable Bytes\tAvg. Disk Queue Length");
     // 循环采样
     for (int j = 0; j < 100; j++)
     {
         Console::WriteLine(privBytes->NextValue().ToString() + "\t" + ioRead->NextValue().ToString() + "\t\t\t"
             + availBytes->NextValue().ToString() + "\t" + diskQueue->NextValue().ToString());
```

```
            //为性能计数器产生一些文件 I/O 操作
            log->WriteLine(privBytes->NextValue().ToString() + "\t" + ioRead->NextValue().ToString() + "\t\t\t"
                + availBytes->NextValue().ToString() + "\t" + diskQueue->NextValue().ToString());
            log->Flush();
            System::Threading::Thread::Sleep(500);
        }
        Console::ReadLine();
    }
```

2. Event Tracing for Windows

Event Tracing for Windows（ETW，事件日志）是由操作系统提供的一种通用的、系统开销较低（与性能日志和警报相比）的事件追踪手段，用于监控具有负载的系统的性能。ETW 主要用于必须频繁记录事件、错误、警告或审核的服务器应用程序。ETW 提供用户模式的应用程序和内核模式的设备驱动程序所触发的事件追踪机制。此外，ETW 还能够动态地启用或者禁用日志记录，以便于进行详细的追踪，而无须重新启动操作系统或者应用程序。

ETW 最先在 Windows 2000 中被引入。从此，各种 Windows 操作系统核心和服务组件都通过 ETW 记录其活动，它如今是 Windows 平台上的关键系统仪表技术之一。在 Windows 7 中，ETW 得到了进一步的增强。首先是更强的功能、可伸缩性和健壮性，增加了多个新的 Windows 事件和事件提供者。在 ETW 的开发方面，增加了设计时支持、从 XML 自动生成代码等。正是基于 ETW 的优秀性能和强大功能，越来越多的第三方应用程序开始放弃自己的日志系统，逐渐开始使用 ETW 来追踪和记录其状态和活动。这里，就利用 ETW 来记录我们自己开发的应用程序的各种事件活动，从而进行性能调优或者应用程序的日常维护。图 21.24 演示了 ETW 的体系结构。

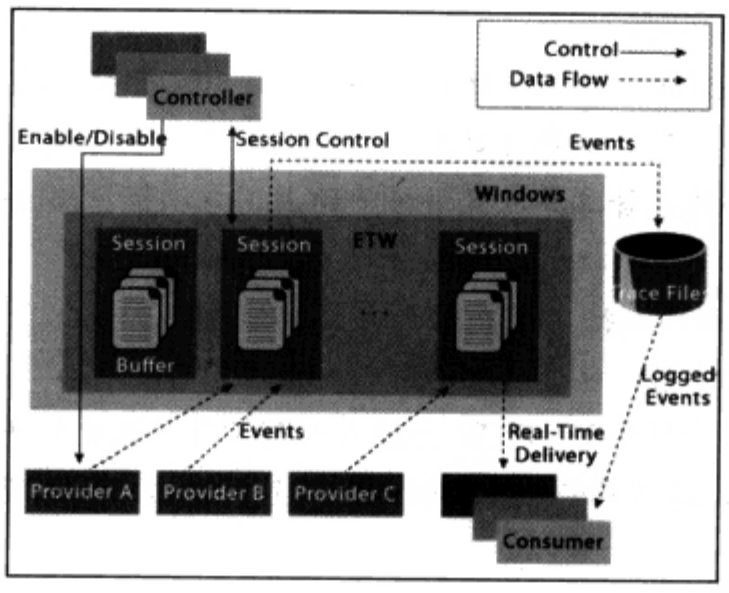


图 21.24　ETW 的体系结构

ETW 主要由 4 个部分组成：事件提供者、控制者、消费者和事件跟踪会话。缓存和日志记录在事件跟踪会话中进行，在这里接收事件并创建一个跟踪文件。对于 ETW 会话，可以选择多种不同的日志模式，例如，可以将一个会话配置成将事件直接发送给消费者应用，或者当日志到达一定大小后将文件中的旧事件覆盖。每个会话都会创建一个独立的写线程来将它们写入到文件或到实时消费者应用。为了达到较高的性能，每个处理器都各自使用独立的缓存来避免使用锁。

事件提供者是向 ETW 会话写入事件的逻辑入口。任何可记录的重要活动都可以称为事件，并且每个都可以用写入到 ETW 的事件表示。事件提供者可以是一个用户态应用、托管应用、驱动或任何软件体，惟一的要求是事件提供者必须通过注册 API 向 ETW 注册一个提供者 ID。注册完成后，就可以从代码中的多个点通过调用 ETW 日志 API 来向 ETW 写入事件。ETW 控制者应用能够动态唤醒事件提

供者，调用日志 API 向由控制者指定的事件跟踪会话发送事件。每个被发送的事件包括一个固定格式的头、事件元数据以及用户相关的上下文数据。下面给出代表事件描述的数据结构定义：

```
typedefstruct _EVENT_DESCRIPTOR {
    USHORT Id;
    UCHAR Version;
    UCHAR Channel;
    UCHAR Level;
    UCHAR Opcode;
    USHORT Task;
    ULONGLONG Keyword;
} EVENT_DESCRIPTOR, *PEVENT_DESCRIPTOR;
```

每个成员的含义如表 21.4 所示。

表 21.4　　事件描述的成员

成　员	说　明
Id	事件的惟一 ID
Version	事件版本，可能在将来实现中改变
Channel	事件所处的事件组标号
Level	过滤设置
Keywords	代表提供者的子模块
Opcode	指定对事件的特定操作
Task	指定一个被监督的普通逻辑模块或任务

例如，Windows 内核提供者将所有的文件 I/O 操作事件成组到一个 FileIO 任务，Opcode 指定进行什么操作，如创建、打开、读和写。

而 Channel 成员定义了一组面向同一目标的事件，一共有 4 种类型的 Channel。

- Admin: 目标为管理员，发送到事件日志。
- Operational: 面向高层的监督工具，提供更多的上下文细节，比 Admin 事件出现更频繁，进入事件日志。
- Analytic: 面向专家级别的支持，提供诊断细节和检测工具。
- Debug: 用于产生调试消息，包含将被送到开发者的事件。

当一个事件被记录到一个会话中时，ETW 会加入一些额外的数据到事件中，包括时间戳、进程号、线程号和 CPU 使用的数据等，这些额外数据将随事件一起被送到事件消费者，很多消费者都需要用到这些数据。

事件控制者负责启动和停止 ETW 会话以及唤醒会话关联的提供者。在一些情况下，如调试，需要启动控制者工具来进行深度跟踪。控制者必须获取 Windows 系统的许可才能控制会话，这在默认情况下只能由特权用户提供。

事件消费者负责从会话读取事件文件，或者监听实时事件并处理这些事件。事件消费基于回调机制。消费者注册一个事件回调函数，这将被 ETW 在事件到达时调用，事件按照严格的到达时间顺序被发送给 ETW 消费者。可以使用通用的事件消费者工具将事件导出为不同的格式。

为了便于在应用程序中使用ETW，Visual Studio 2010 在 System.Diagnosis 命名空间提供了 EventLog 类来进行简单的事件输出，利用这个类来记录简单的应用程序日志，以代替应用程序的日志系统。

下面的示例演示了如何使用 EventLog 类：

```
using namespace System;
using namespace System::Diagnostics;
```

```
using namespace System::Threading;
int main()
{
    //如果源不存在，先创建指定名字的源
    if (!EventLog::SourceExists("MySource"))
    {
        //创建事件日志源后不能马上使用，必须等待一段时间，应用程序才能使用这个源
        //退出程序后，再次运行这段代码使得可以访问源
        EventLog::CreateEventSource("MySource", "MyNewLog");
        Console::WriteLine( "CreatingEventSource" );
        //创建源完毕，退出应用使得源可以被注册
        return 0;
    }

    //创建一个 EventLog 实例，并指定源
    EventLog^ myLog = gcnew EventLog;
    myLog->Source = "MySource";
    //向事件日志写入信息
    myLog->WriteEntry( "Writing to event log." );
}
```

除了使用默认的事件，Visual Studio 2010 还允许用户自定义事件，主要包括 5 步，如图 21.25 所示。

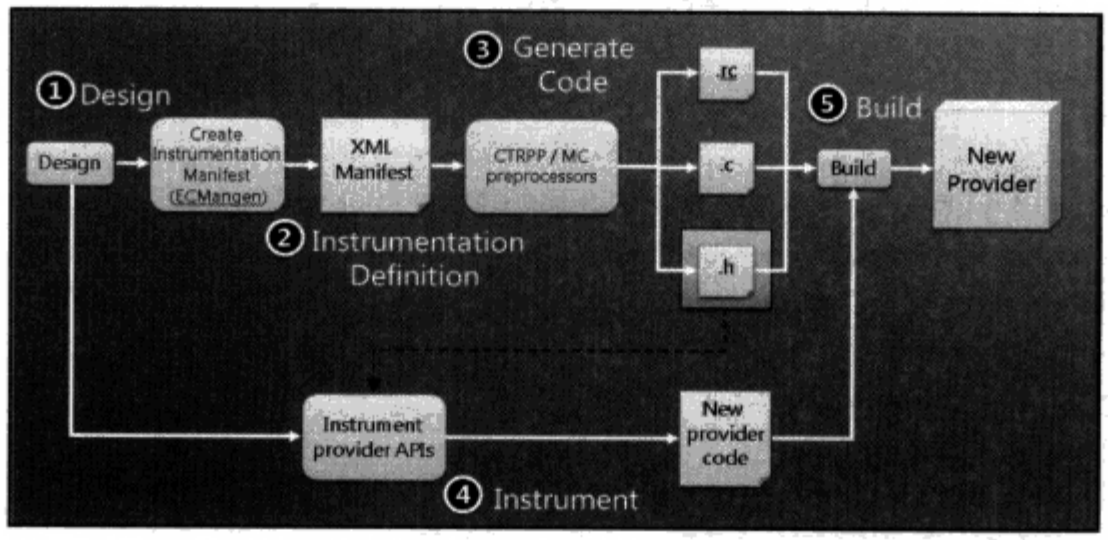


图 21.25　用户自定义事件的 5 个步骤

（1）Design：定义在事件中输出的信息，可以是事件的类型、等级、事件模板等。

（2）Instrumentation Definition：使用 XML 文件来描述和定义一个事件，需要包括事件提供者的 ID、事件类型、事件模板等属性。可以直接使用文本编辑，更简单的是使用 Windows SDK 的 Manifest Generator 工具来直接创建事件的 XML 文件。下面给出一个事件 manifest 的 XML 文件的示例：

```
<provider name="Microsoft-Windows-Kernel-Registry"
    guid="{70eb4f03-c1de-4f73-a051-33d13d5413bd}"
    symbol="RegistryProvGuid"
    resourceFileName="%SystemRoot%\System32\advapi32.dll"
    messageFileName="%SystemRoot%\System32\advapi32.dll">
  <channels>
    <channel name="Microsoft-Windows-Kernel-Registry/Analytic"
        chid="RegistryEvents" symbol="REG_Events" type="Analytic"
        isolation="System">This channel contains registry
        events.</channel>
```

```
    </channels>
    <opcodes>
      <opcode value="32" name="CreateKey" symbol="" />
      ...
    </opcodes>
    <keywords>
      <keyword name="CreateKey" symbol="" mask="0x1000" />
      ...
    </keywords>
    <templates>
      <template tid="tid_RegOpenCreate">
        <data name="BaseObject" inType="win:Pointer"
              outType="win:HexInt64" />
        <data name="KeyObject" inType="win:Pointer"
              outType="win:HexInt64" />
        <data name="Status" inType="win:UInt32"
              outType="win:HexInt32" />
        <data name="Disposition" inType="win:UInt32" />
        <data name="BaseName" inType="win:UnicodeString"
              outType="xs:string" />
        <data name="RelativeName" inType="win:UnicodeString"
              outType="xs:string" />
      </template>
      ...
    </templates>
    <events>
      <event value="1" symbol="ETW_REGISTRY_EVENT_CREATE_KEY"
             template="tid_RegOpenCreate" opcode="CreateKey"
             channel="RegistryEvents" level="win:Informational"
             keywords="CreateKey"
             message="$(string.Registry.RegOpenCreate)"/>
             ...
        </events>
      </provider>
        ...
    <localization>
      <resources culture="en-US">
        <stringTable>
          <string id="Registry.RegOpenCreate"
                  value="Registry key %6 was created with status %3." />
        </stringTable>
      </resources>
    </localization>
```

（3）Generate Code：产生 manifest 的 XML 文件后，并不能直接用于创建事件提供者，还需要利用 Windows SDK 提供的 Windows Message Compiler（mc.exe）编译器来对 XML 文件进行编译，以产生相应的事件提供者代码。

（4）Instrument：创建事件提供者（类）完毕，就可以在自己的应用程序代码中创建事件提供者实例，并将对应的事件输出到日志中。

（5）Build：代码编写完毕，就可以编译整个项目，生成一个新的事件提供者。使用时需要将其部

署到目标机器上。

21.3　开发基于 Windows 7 的新特性

21.3.1　实现多点触摸

Windows 7 使用户无须使用中间设备，通过手指触摸方式就能够管理应用程序。这扩展了笔记本电脑基于触摸板的功能。与其他定点设备不同，这种新功能支持在不同定点位置上同时发生多个输入事件，支持复杂的场景，例如通过手指或由多个并行用户管理应用程序。然而，要成功实现此功能，必须调整应用程序的用户界面和行为，以支持这种新的输入模式。

为了在 Windows 7 系统上实现多点触摸，首先需要一台支持多点触摸的硬件设备。目前市场上只有极少量的产商提供这种设备，如 HP 公司的 TouchSmart All-in-One PC、Dell 公司的 Letitude XT。当然随着多点触摸技术的流行，未来市场上必定会出现越来越多的支持多点触摸的硬件产品。

Windows 7 提供了对多点触摸的系统支持。为了支持多种不同的应用需求，Windows 触摸平台提供了 3 种不同级别的支持：“好”、“出色”或“最佳”方法。

“好”方法是这些方法中最简单的。设计应用程序用户界面时应该将触摸能力考虑在内。可以使用大量基于 Win32 的简单工具构建一种自然的界面，以提供更出色的用户体验。滚动等触摸能力来自于 Win32 控件，无须额外的工作。例如，尝试使用手指滚动正在阅读的文档，这就是“好”方法。

“出色”方法让系统接收各种低级触摸事件，并获取系统对这些事件的探索结果。例如，如果用户在屏幕上做出一个旋转动作，系统将发出一个带有旋转角度的旋转手势。尽管“出色”方法易于使用，但它具有自身的局限性。使用手势无法同时实现旋转、平移和缩放，也无法同时处理多个基于不同触摸方式的操作。例如，两个用户操作窗口的不同区域。

“最佳”方法是读取低级触摸事件，将其作为应用程序的输入。Piano 等应用程序或可供用户同时操作多个滑块等复杂控件都是不错的例子。运行 MS Paint，从工具箱中选择一个绘图工具，然后使用手指进行绘制（如果硬件支持）。

无论何时，用户触摸一个 Windows 7 支持的触摸感知设备，Windows 7 多点触摸平台在默认情况下都会向应用程序发送手势消息，消息号为 WM_GESTURE。这里的手势是指一根或两根手指的动作作为输入，并被系统转化为一些预先定义好的动作（这就是手势识别）。一旦操作系统正确识别出用户的手势，就会向应用发送消息。这个消息包含了所有需要编码的信息并使应用做出正确的操作。Windows 7 支持下面列出的几种手势：

- 缩放（Zoom）；
- 单根手指和两根手指的平移（Pan）；
- 旋转；
- 两根手指轻拍（tap）；
- 按下和轻拍（Press and tap）。

为了使应用程序能够通过手势进行操作，需要处理 OS 发送给应用程序的 WM_GESTURE 消息。如果使用 Win32 编程，可以直接在应用程序的 WndProc() 函数内判断接收到的消息是否为 WM_GESTURE。

WM_GESTURE 是对上面介绍的 5 种手势的通用消息，为了进一步判断用户确切的手势，需要对手势消息进行解码。有关手势的信息可以在 lParam 参数内获取，需要通过一个特殊的 GetGestureInfo() 函数来实现对消息的解码。下面的代码演示了如何获取手势的确切信息：

```
GESTUREINFO gi;
```

```
ZeroMemory(&gi, sizeof(GESTUREINFO));
gi.cbSize = sizeof(gi)4
BOOL bResult = GetGestureInfo((HGESTUREINFO)lParam, &gi);
```

这里需要使用一个 GESTUREINFO 结构体来存放手势的信息，通过访问这个结构体的 dwID 成员来标识接收到的手势。另外，GESTUREINFO 结构体内还包含其他几个重要的成员，简要介绍如下。

- cbSize：存放这个结构体的大小，单位为字节。
- ptsLocation：这是一个 POINT 结构体集合，存放手势相关的坐标（动作发生位置的坐标）。
- dwFlags：标识手势的状态，如开始、惯性和结束。
- ullArguments：这是一个 64 字节的非负整数，包含手势的参数，这个成员对于不同的手势类型代表不同的信息。

获取了手势消息后，就可以使用一个完全的 switch…case 方法来分别列于举出不同的手势类型，并分别进行不同的处理，如下面的代码：

```
void CMTTestDlg::DecodeGesture(WPARAM wParam, LPARAM lParam)
{
    GESTUREINFO gi;
    ZeroMemory(&gi, sizeof(GESTUREINFO));
    GetGestureInfo((HGESTUREINFO)lParam, &gi);      //获取手势信息
    switch (gi.dwID){                                //判断手势类型
        case GID_ZOOM:
            //处理缩放操作
            break;
        case GID_PAN:
            break;
        case GID_ROTATE:
            break;
        case GID_TWOFINGERTAP:
            break;
        case GID_PRESSANDTAP:
            break;
        default:
            //这是系统无法识别的手势
            break;
    CloseGestureInfoHandle((HGESTUREINFO)lParam);
}
```

这里需要注意的是，在处理的最后一定要调用 CloseGestureInfoHandle()函数，这个函数的作用是将手势信息句柄相关的资源关闭。如果处理了 WM_GESTURE 消息，那么就需要负责关闭手势信息的句柄，否则就可能导致内存泄露。

通过上面的讨论可知，处理手势消息一般都遵循固定的流程，包括配置、对手势消息解码、对不同类型的手势分别处理，最后关闭句柄。下面演示一个对缩放手势进行具体处理的示例。缩放手势是通过两根手指的相对运动实现的：如果两根手指靠近运动则代表希望进行缩小操作；如果两根手指做远离运动，则代表放大。缩放手势一般用于控制对象的缩放显示，如图 21.26 所示。

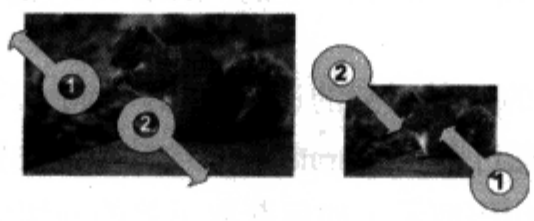


图 21.26　缩放手势演示：左边为放大手势，右边为缩小手势

下面介绍如何处理 GID_ZOOM 手势来使用户的应用程序实现缩放效果。GESTUREINFO 结构体中的 dwFlags 成员用于确定手势状态，可能是下面列出的几种不同值。

- GF_BEGIN：表示手势开始，这是第一个接收到的 WM_GESTURE 消息。
- GF_INERTIA：表示手势处于运动惯性。
- GF_END：表示手势动作完成。
- 其他值：表示手势消息的剩余部分，经常用做 delta。

这里将使用 GF_BEGIN 标识来保存手势开始的坐标，而在手势动作过程中，不断获取手势的当前位置，利用这两个值来决定缩放的比例。如下面的代码所示：

```
case GID_ZOOM:                                          //处理缩放手势
switch(gi.dwFlags)
{
case GF_BEGIN:                                          //手势开始
    _dwArguments = LODWORD(gi.ullArguments);
    _ptFirst.x = gi.ptsLocation.x;                      //获取手势开始位置
    _ptFirst.y = gi.ptsLocation.y;
    ScreenToClient(hWnd,&_ptFirst);                     //将坐标从屏幕坐标系转化为客户坐标系
    break;
default:
    //获取手势的第二点坐标
    _ptSecond.x = gi.ptsLocation.x;
    _ptSecond.y = gi.ptsLocation.y;
    ScreenToClient(hWnd,&_ptSecond);
    //计算算放中间点坐标
    ptZoomCenter.x = (_ptFirst.x + _ptSecond.x)/2;
    ptZoomCenter.y = (_ptFirst.y + _ptSecond.y)/2;
    //缩放的比例由起始点和下一点的距离确定
    k = (double)(LODWORD(gi.ullArguments))/(double)(_dwArguments);
    //根据缩放比例将对象进行缩放操作
    ProcessZoom(k,ptZoomCenter.x,ptZoomCenter.y);
    InvalidateRect(hWnd,NULL,TRUE);
    //将新的坐标点作为下次缩放操作的起始点
    _ptFirst = _ptSecond;
    _dwArguments = LODWORD(gi.ullArguments);
    break;
}
break;
```

处理 Windows 7 识别的其他类型手势的流程与上面举例说明的缩放手势处理是相似的，这里不再赘述。

21.3.2　实现获取传感器与位置

Windows 7 操作系统提供了对传感器设备的支持。这包括对位置传感器的支持，如 GPS 设备。作为这种支持的一部分，Windows 传感器和位置平台提供了一种设备制造的标准方法，以使软件开发商和客户能够正确感知传感器设备。同时，该平台为开发人员提供了一个标准化的 API 和设备驱动程序接口（DDI）与传感器和传感器的数据协同工作。

传感器用于获取多种配置，从这个角度来看，几乎所有的东西，只要能提供有关的物理现象的数

据都可以被称为一个传感器。逻辑式传感器也可以提供软件或固件传感器的功能，提供仿真的信息。此外，单一的硬件设备可以包含多个传感器。

在 Windows 平台中，传感器和地点组织成类。而代表传感器设备的两大类是传感器和类型。例如，在视频游戏控制器，检测的位置和球员的手部动作（也许是视频保龄球游戏），传感器将被归类为导向传感器，但其类型为三维加速度计。在代码方面，Windows 通过使用全局惟一标识符（GUID）来识别不同的传感器，其中有许多预定义的类别和类型。当它需要新的类别和类型时，设备制造商可以通过确定和发布新的 GUzID。

定位设备组成一个特别有趣的类别。现在，大多数人都对全球定位系统（GPS）有所了解。在 Windows 中，全球定位系统传感器作为位置类别的一部分，位置类还包括其他的传感器类型。这些传感器有些是基于软件的，如 IP 解析器——提供以位置信息为基础的互联网地址，移动电话塔 triangulator——决定在附近的塔或者一个 Wi - Fi 提供的网络位置读取位置信息的定位无线网络连接的枢纽。

在 Windows 传感器和定位平台由以下开发者和用户组成。

- 直通内线使 Windows 以提供一个传感器设备的标准方法连接到电脑，并提供数据，其他子系统。
- 在 Windows API 提供了传感器的方法，属性设置和活动，与相连的传感器和传感器数据。
- 在 Windows API 的位置，它是基于 Windows API 的内置传感器，提供了一套包括编程对象脚本对象，与位置信息工作。
- 地点和其他传感器控制面板，使计算机管理员设置传感器，位置传感器，包括为每个用户。

图 21.27 显示了组件之间的关系。

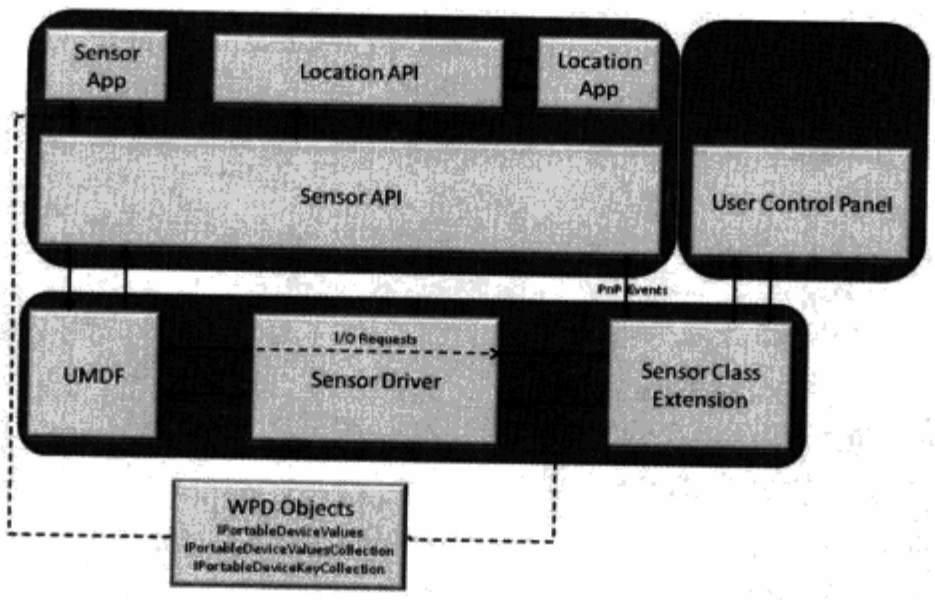


图 21.27 Windows 传感器和定位平台各个组件之间的关系

1. 设备驱动程序接口

传感器制造商可以通过开发设备驱动将传感器连接到 Windows 7 系统上。传感器驱动的实现基于 Windows 便携式设备（WPD）驱动模型，这个驱动模型是基于 Windows 用户模式驱动框架(UMDF)建立的。由于这些技术的推出，对于经验丰富的设备驱动程序员将会发现编写一个传感器驱动程序只不过是一个业已熟悉的任务，因为这些驱动都是基于同一个模型编写出来的。传感器 DDI 使用特定的 UMDF 及 WPD 数据类型和接口，接着只需定义传感器所需要的特定的 WPD 命令和参数即可。

2. 传感器 API

这个 Windows 传感器 API 允许 C++开发人员使用一系列 COM 接口来开发基于传感器的程序。该 API 定义了执行通用传感器编程任务的接口，包括传感器，与传感器堆协同工作，与独立传感器工作

以及处理数据。

传感器 API 主要包含在两个头文件 Sersorsapi.h 和 Sensors.h 中，主要包括 3 个主要接口。

- ISensorManager：包括 Sersor 类型列举、依附的事件和请求许可。
- ISensor：代表传感器本身，可以获取属性、事件、报告等。
- ISensorDataReport：获取传感器的报告数据。

下面的代码演示了如何将系统支持的所有传感器类型列举出来：

```
#include <sensorsapi.h>
#include <sensors.h>
HRESULT hr;
CComPtr<ISensorManager> pSensorManager;
pSensorManager.CoCreateInstance(CLSID_SensorManager);
CComPtr<ISensorCollection> pALSCollection;
CComPtr<ISensor> pALSSensor;
//获取系统中所有的 ATL 传感器
pSensorManager->GetSensorsByType(SENSOR_TYPE_AMBIENT_LIGHT, &pALSCollection);
hr = pSensorManager->RequestPermissions(
    0,                  // 所有者窗口
    pALSCollection,     // 传感器集合访问许可
    TRUE);              // 模块标识
if(SUCCEEDED(hr))
{
    pALSCollection->GetAt(0, &pALSSensor);
}
```

下面的代码演示了如何获取传感器的数据，这里是获取灯的亮度信息。

```
STDMETHODIMP CALSEventSink::OnDataUpdated(
ISensor* pSensor, ISensorDataReport* pNewData)
{
    PROPVARIANT lightLevel;
    PropVariantInit(&lightLevel);
    //从 ISensorDataReport 对象获取传感器的数据
    pNewData->GetSensorValue(SENSOR_DATA_TYPE_LIGHT_LEVEL_LUX, &lightLevel);
    //获取数值信息
    float luxValue = V_FLOAT(lightLevel);
    //对数据进行归一化处理
    double lightNormalized = ::pow(luxValue, 0.4) / 100.0;
    UpdateUI(lightNormalized);
    PropVariantClear(&lightLevel);
    return S_OK;
}
```

3．位置 API

建立在传感器平台之上，位置 API 提供了一种简单的方法来检索有关地理位置的数据，同时保护用户隐私。定位 API 通过一系列 COM 接口所代表的对象提供功能，这些对象既可以在 C++编程语言中使用，也可以在脚本语言中使用，比如 Jscript 脚本支持使得访问工程的位置数据更加容易。

位置 API 使用的默认 COM 接口是 Ilocation，用于代表一个位置对象。通过这个对象，可以注册或注销报告位置数据的事件，指定报告事件的频率，获取某种特殊的位置类型数据，以及为用户访问位置数据打开一个系统对话框。位置使得程序可以在当前位置发生变化或某种特殊位置类型状态发生变化时收到事件通知。

4. 位置和其他传感器的控制面板

Windows 7 操作系统包含位置和传感器控制面板，便于管理员或所有用户能够开启或关闭安装在系统上的传感器。由于一些传感器可能暴露敏感数据，这个用户界面严格控制每个用户可以访问到的传感器。用户还可以查看传感器的属性信息并修改出现在用户界面中的传感器描述信息。

控制面板还提供了一个默认位置，使用户能够提供它们的位置。当没有可用传感器时，该平台将使用用户提供的这个位置。用户可以提供的位置信息，包括街道地址、城市、州或省，以及国家或地区。

21.3.3 实现 Windows 7 Ribbon 界面开发

Ribbon 界面是随着 Office 2007 一同发布的。在过去的几年中，微软不断吸取用户对 Ribbon 界面的反馈，不断对其进行改进。到了 Windows 7，微软正式加入了名为 Scenic Ribbon 的第二代 Ribbon 界面，即取代第一代 Ribbon 的升级版本。随着 Scenic Ribbon 成为 Windows 7 的一部分以及微软所提供的 Scenic Ribbon API，这意味着今后第三方程序开发者能在自己开发的应用程序中免费使用这种界面形式。Scenic Ribbon 作为第二代 Ribbon 界面，适用于所有的 Windows 应用程序，和上一代有所不同是理所当然的。例如，由于最初 Ribbon 界面是 Office 专用，因此 Office 2007 程序窗口左上角有一个比较显眼的 Office 图标按钮。当应用到其他程序之后，这个图标显然不能继续存在，因此在 Scenic Ribbon 中，这个位置就被相应应用程序自身的图标所取代。另外，为了满足各种应用程序的需要，Scenic Ribbon 还添加了部分 Ribbon 控件，使得控件更加丰富，能够应对各种应用程序的需要。值得一提的是，Office 作为 Ribbon 界面的开山鼻祖 Office 2010 也全面使用了 Scenic Ribbon，以取代第一代 Ribbon 界面。

同时，微软在 Visual C++ 2010 中提供了 Ribbon 资源设计器，使 Ribbon 开发更加简单快捷。

Scenic Ribbon API 是一套基于 COM 的非托管 API。通过这套 API，可以为所有基于 Windows 的非托管应用程序创建 Ribbon 界面面板，添加 Ribbon 控件，处理 Ribbon 控件的各种动作，执行某种业务逻辑等。而至于 Ribbon 界面的布局、各个控件的位置和属性等，则是通过基于 XAML 的 XML 文件来进行定义和描述的。编译器会将这个 XML 文件编译成相应的资源文件、头文件和二进制的 BML 文件。而 Scenic Ribbon API 最终会读取这些描述控件信息的文件并创建 Ribbon 界面。在业务逻辑层，Scenic Ribbon API 提供了相应的 Execute()函数来处理所有控件的消息，在其中可以完成相应的业务逻辑。

在 Windows 7 中，微软还提供了用于为非托管代码（Native）应用程序创建 Ribbon 界面的 Scenic Ribbon API 的解决方案。另外，针对基于 WPF 的托管代码应用程序，微软还将提供另外的基于 WPF 的 Ribbon 界面解决方案。

下面通过一个示例说明如何创建 Ribbon 界面的应用程序。首先新建一个 MFC 应用程序，选择 Windows 7 风格，如图 21.28 所示。

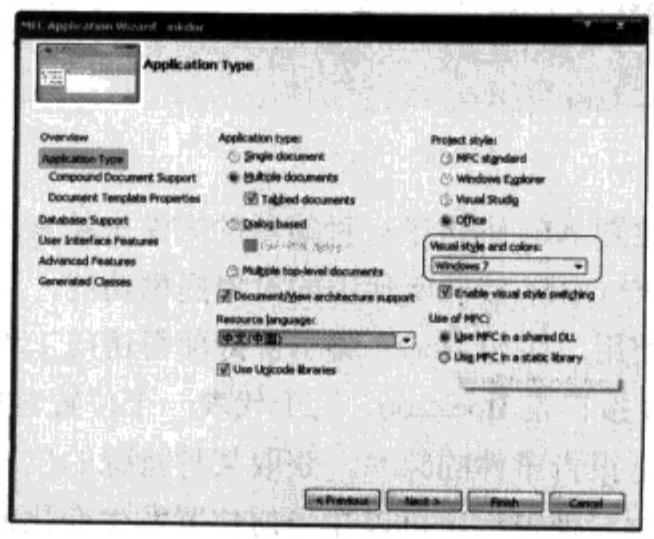


图 21.28 新建一个 Windows 7 风格的 MFC 应用程序

在 MFC 向导的第 6 步，设置使用 Ribbon 支持，其他几步使用默认值，如图 21.29 所示。

在向导第 7 步设置高级选项，选择支持重启管理器，如图 21.30 所示。

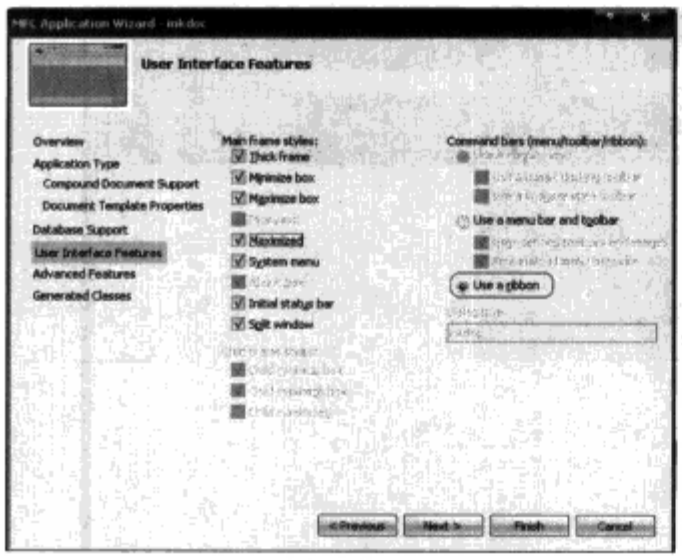


图 21.29　设置使用 Ribbon 支持

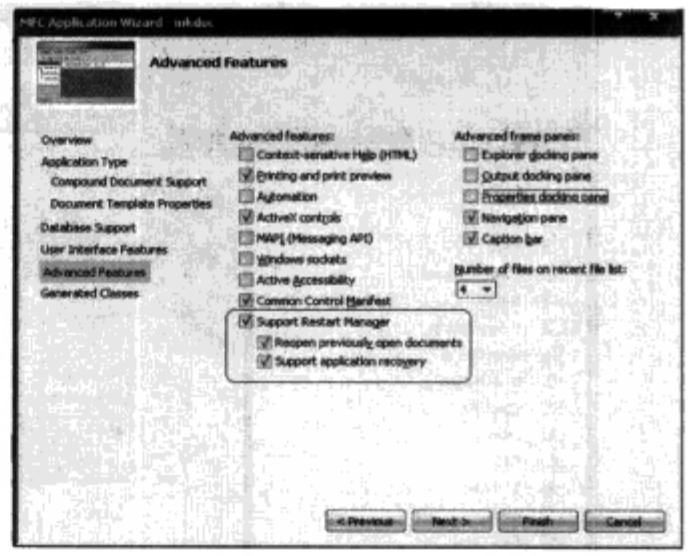


图 21.30　选择重启管理器支持

在下一步中，选择视图类为 CFormView，如图 21.31 所示。

向导结束后，打开 ToolBox，可以看到其中包含 Ribbon 工具，列出可以使用的 Ribbon 控件，如图 21.32 所示。

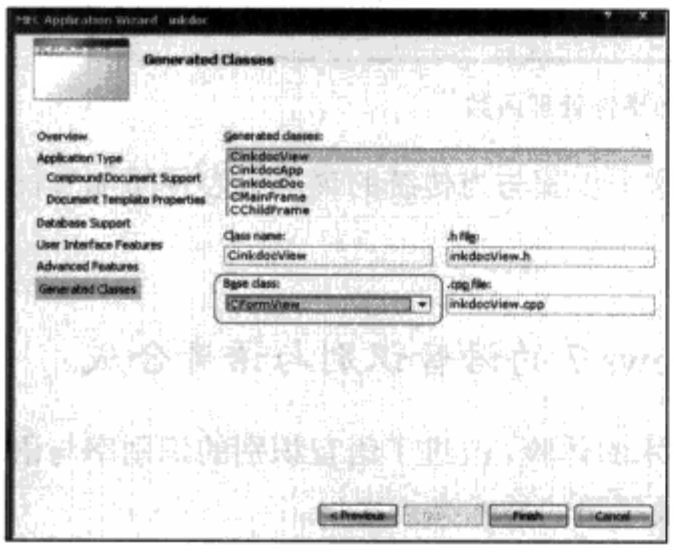


图 21.31　选择视图类为 CFormView

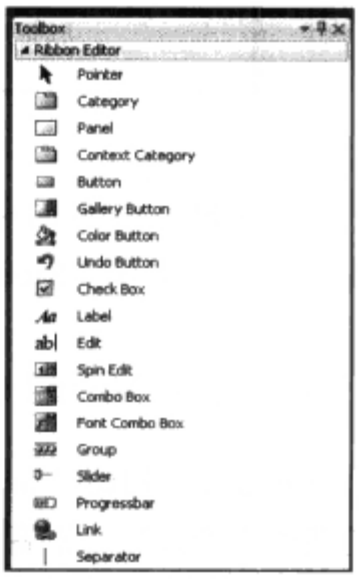


图 21.32　Ribbon 工具栏

像普通控件一样，向窗口内添加 Ribbon 控件，并通过属性窗口设置属性，如图 21.33 所示。

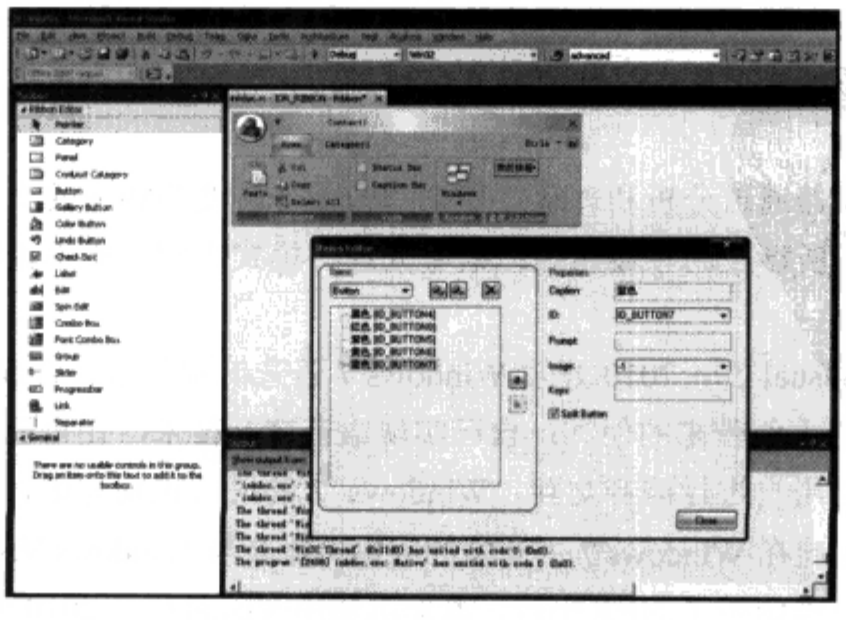


图 21.33　设置属性

为了使 Ribbon 控件实现一定的功能，需要为其添加事件处理函数。右击 Ribbon 控件，在弹出的快捷菜单中选择 Add Event Handler 命令，如图 21.34 所示。

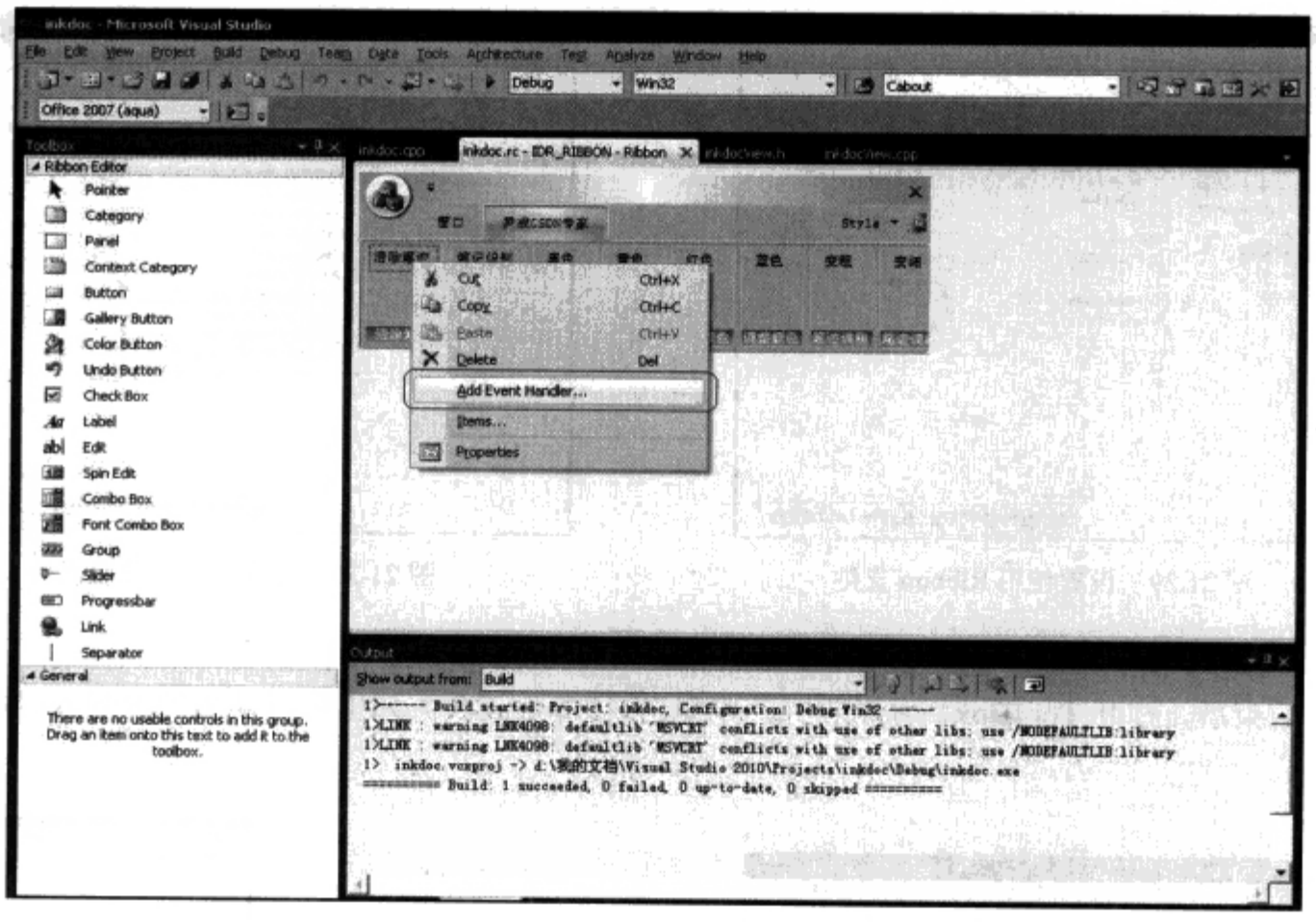


图 21.34　为 Ribbon 控件添加事件处理函数

进入 Ribbon 控件的事件处理函数，添加处理代码。这个步骤与为传统的菜单或控件添加事件处理函数是一样，只不过 Ribbon 界面更加直观方便而已。

21.3.4　基于 Visual C++ 2010 开发基于 Windows 7 的语音识别与语音合成

Windows 7 相比 Windows Vista 进一步改善了语音技术的体验，改进了语音识别的识别率与语音合成的朗读声音效果。语音技术与多点触摸是 Windows 7 最耀眼的技术亮点之一。

语音合成和语音识别技术是实现人机语音通信，建立一个有听和讲能力的口语系统所必需的两项关键技术，使电脑具有类似于人一样的说话能力，是当今时代信息产业的重要竞争市场。

语音合成，又称文语转换（Text to Speech）技术，能将任意文字信息实时转化为标准流畅的自然语音并朗读出来。它涉及声学、语言学、数字信号处理、计算机科学等多个学科，是中文信息处理领域的一项前沿技术，解决的主要问题是如何将文字信息转化为可听的声音信息，即让机器像人一样开口说话。

语音识别，就是让机器通过分析和理解过程把语音信号转变为相应的文本或命令的尖端技术。语音识别是一门交叉学科，所涉及的领域包括：信号处理、模式识别、概率论和信息论、发声机理和听觉机理、人工智能等。

让我们来体验下用 Visual C++ 2010 开发 Windows 7 语音识别与语音合成技术的快感，欣赏微软最新的 Windows 7 新技术给人们带来的便利：自己实现语音交互技术，让自己开发的软件（准备运行在 Windows 7 上的）可以与用户进行语音交互。Windows 7 的语音识别与语音合成与 Windows Vista 与 Windows XP 有所不同，因此在 Windows 7 下开发的应用可能会与 Windows Vista/XP 遇到 API 兼容问题。

因为 Windows 7 都已经附带了.NET 平台，本文讲述用 Visual C++ 2010 开发基于 CLR 的语音技术

应用程序，并同时提供了基于 MFC 程序兼容调用 CLR 的方法。无论是 Visual C++基于 CLR 还是 MFC 的程序，如果想升级到 Windows 7，都可以很快添加语音识别功能、语音朗读功能。

（1）创建一个 CLR 的 Windows Forms Application 项目，命名为 CSDN-speech，如图 21.35 所示。

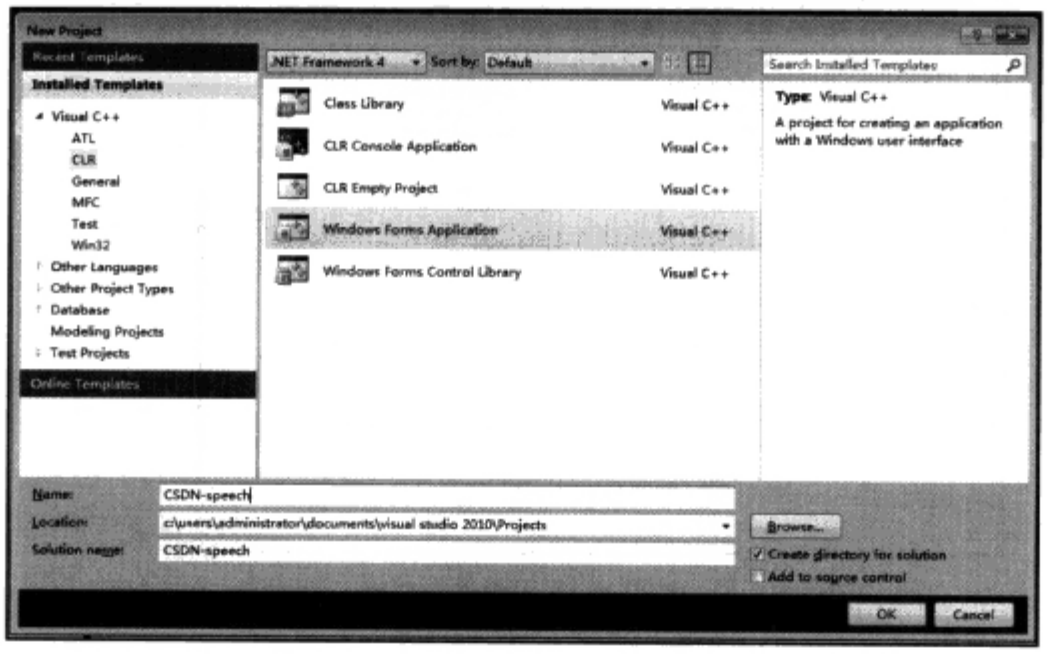


图 21.35　新建一个 CLR Windows Forms 应用程序

（2）插入控件，如图 21.36 所示。

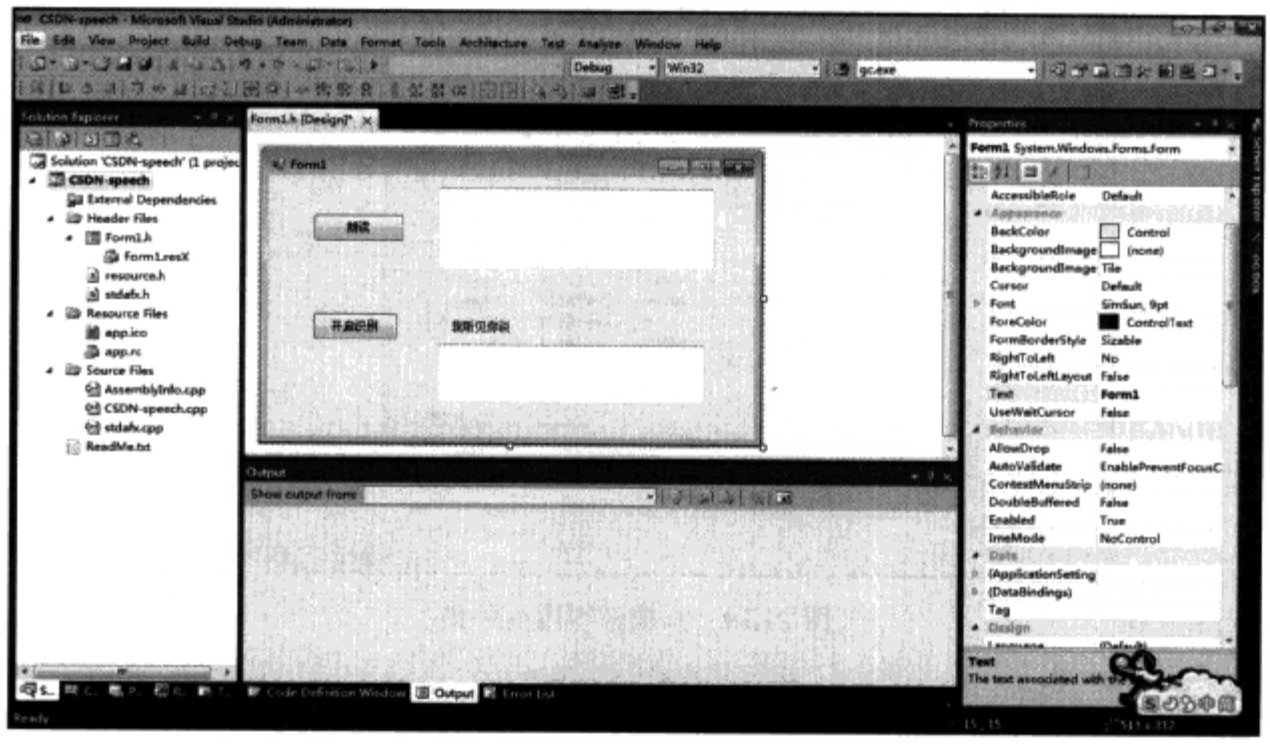


图 21.36　程序设计界面

（3）单击 Project 菜单，选择项目属性命令，如图 21.37 所示。

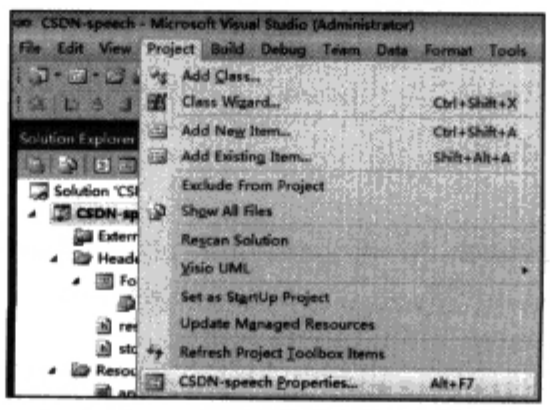


图 21.37　设置项目属性

（4）在项目属性对话框中将看到如图 21.38 所示的属性。

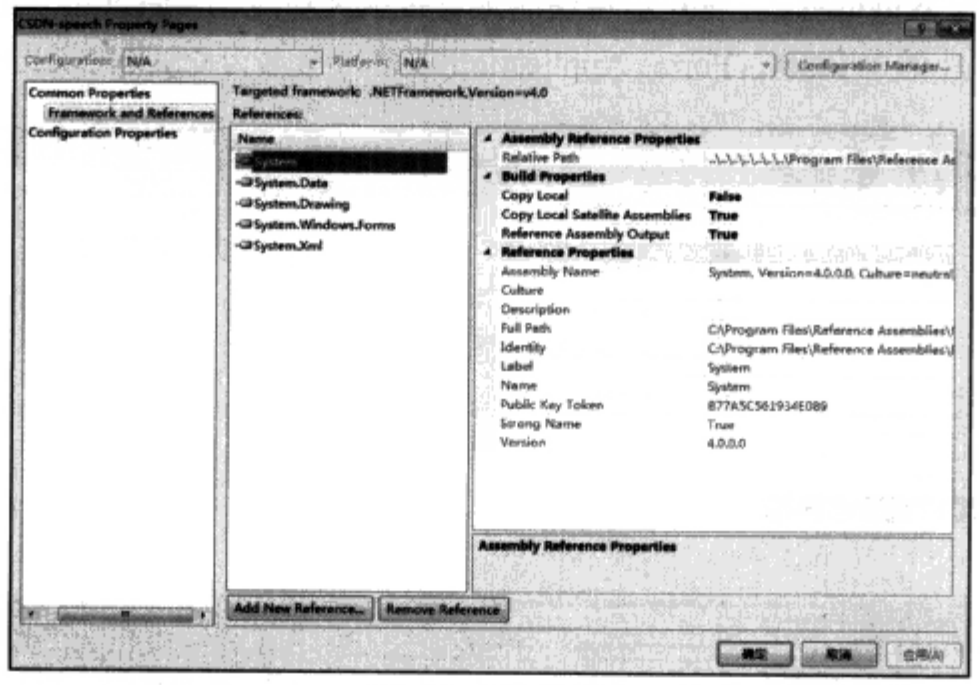


图 21.38　查看项目属性

（5）单击 Add New Reference 按钮，添加 System.Speech（版本为 4.0）的调用，如图 21.39 所示。

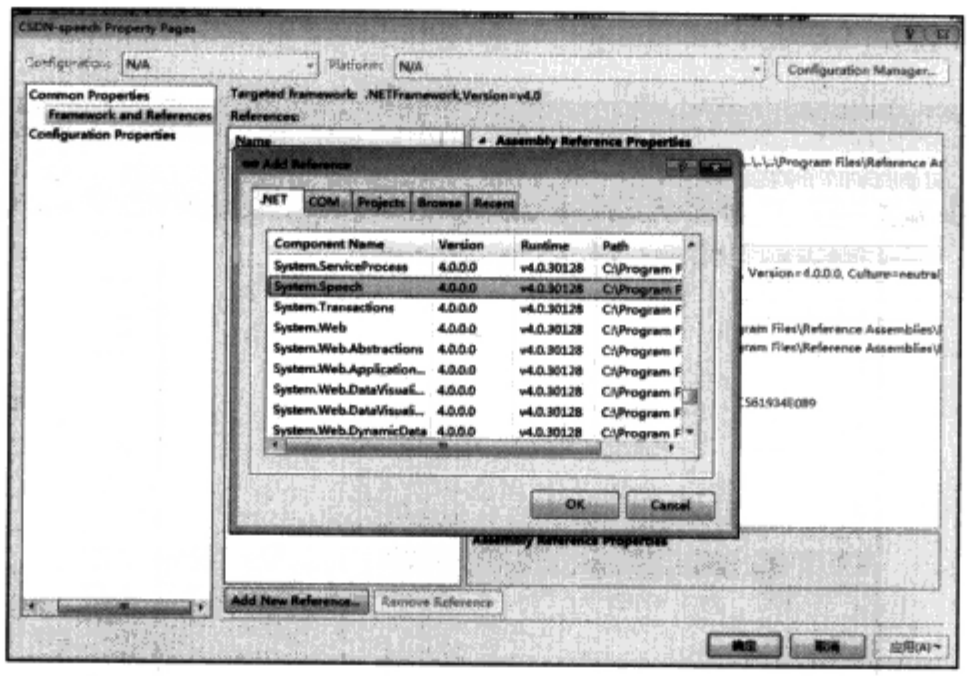


图 21.39　添加新引用程序集

（6）选中“朗读”按钮，创建“朗读”按钮的触发程序，如图 21.40 所示。

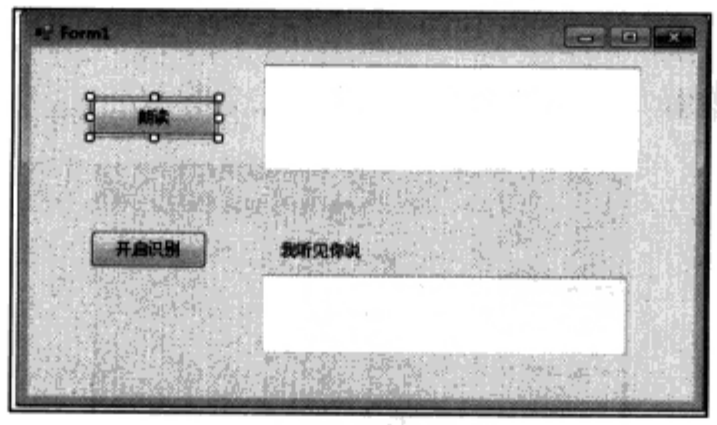


图 21.40　创建朗读按钮触发程序

（7）在程序初始化部分添加下列代码，以添加对于语音技术及语言文化类的引用：

```
using namespace System::Speech;
```

```
using namespace System::Speech::Recognition;
using namespace System::Speech::Recognition::SrgsGrammar;
using namespace System::Speech::Synthesis;
using namespace System::Globalization;
```

（8）为“朗读”按钮添加下列代码：

```
SpeechSynthesizer^ synth;
    synth =gcnew SpeechSynthesizer();
    synth->Speak(this->textBox1->Text);
```

（9）实验语音朗读，试听来自 Windows 7 的语音朗读效果，如图 21.41 所示。

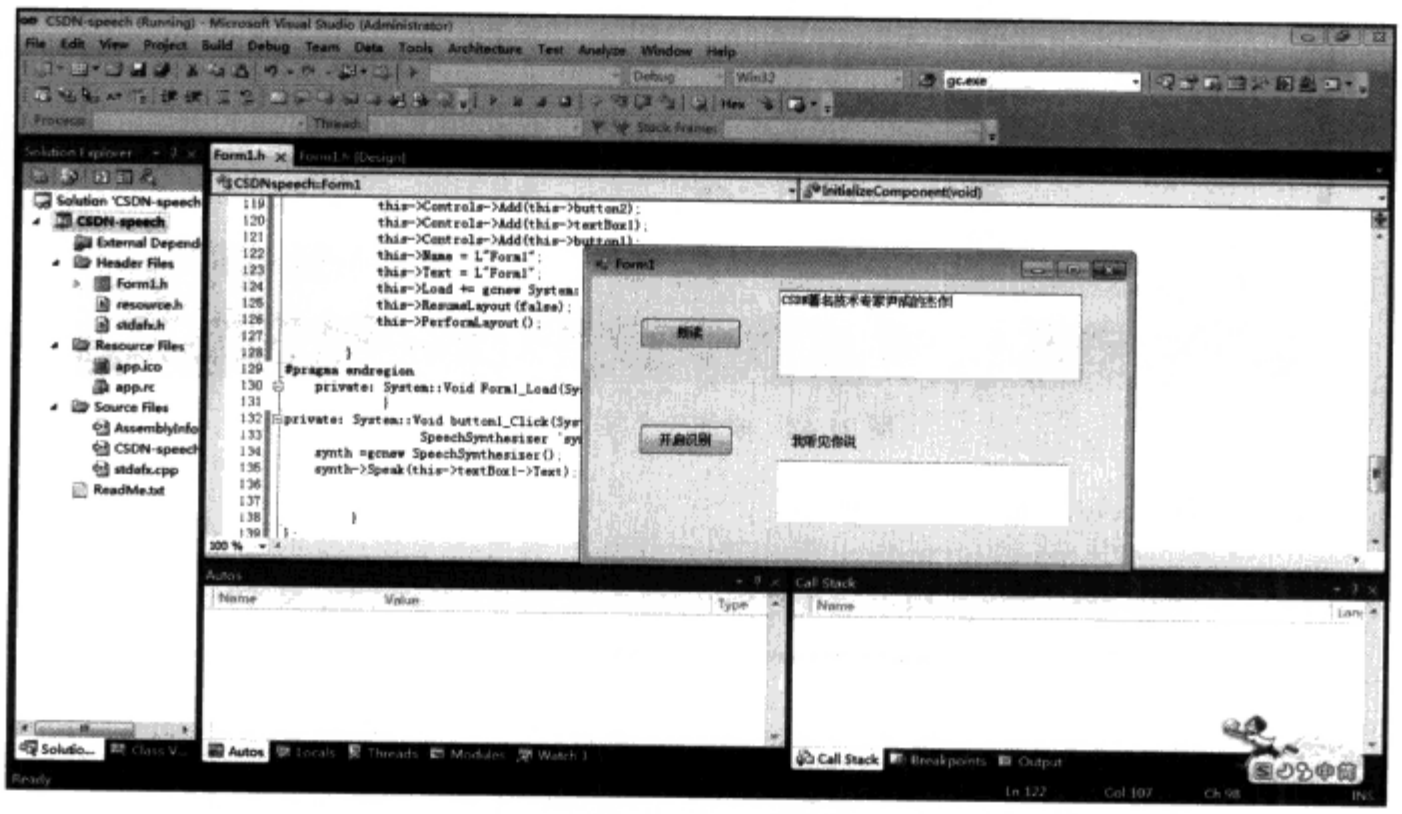


图 21.41　语音朗读程序的运行结果

（10）右击项目资源管理器中的 CSDN-speech 节点，在弹出的快捷菜单中选择 Add|Class 命令，添加一个类，如图 21.42 所示。

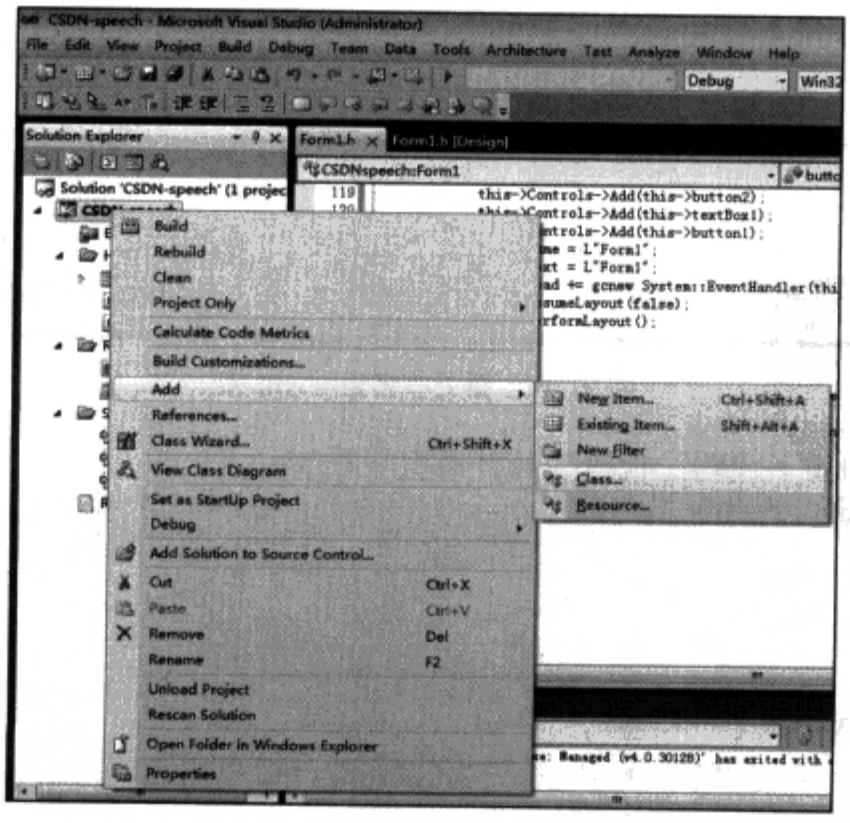


图 21.42　向项目中添加一个新类

（11）添加一个纯 C++类，如图 21.43 所示。单击 Add 按钮。

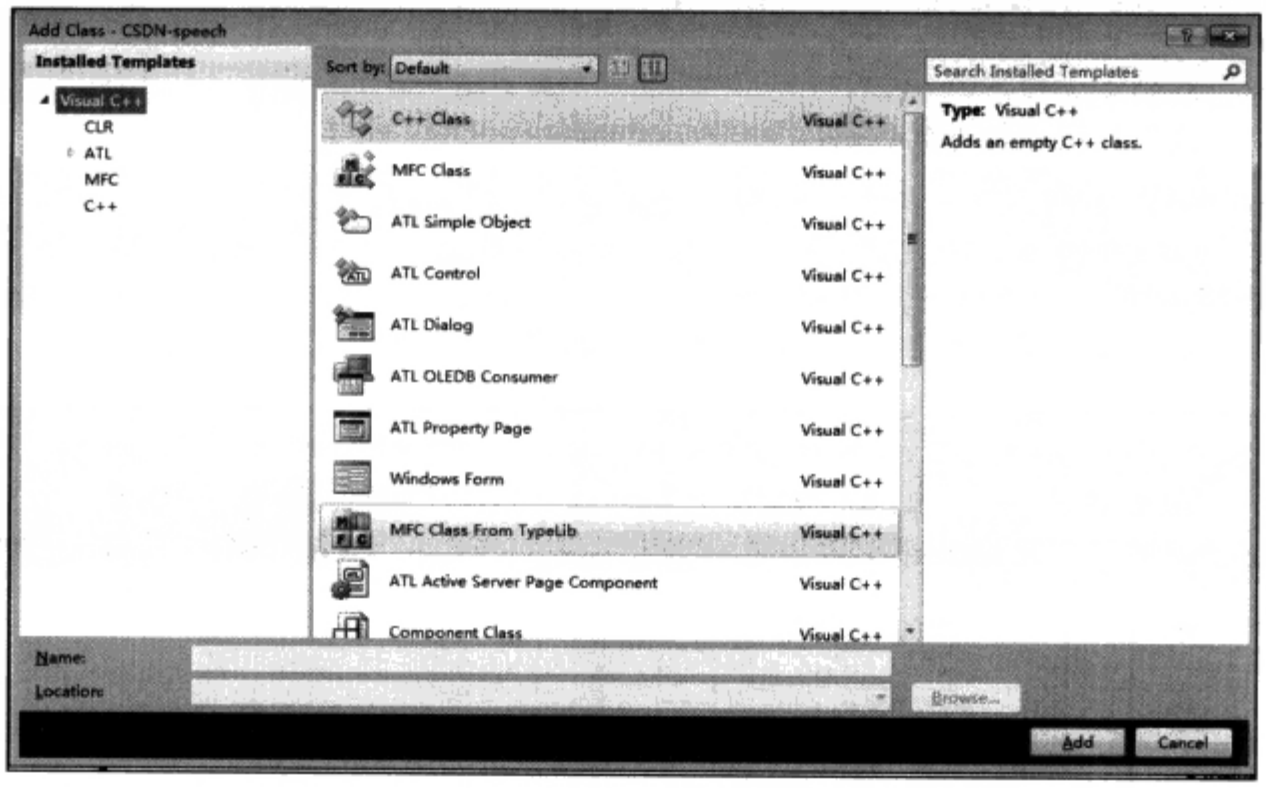


图 21.43　添加一个纯 c++类

（12）将类命名为 yuyin，如图 21.44 所示。

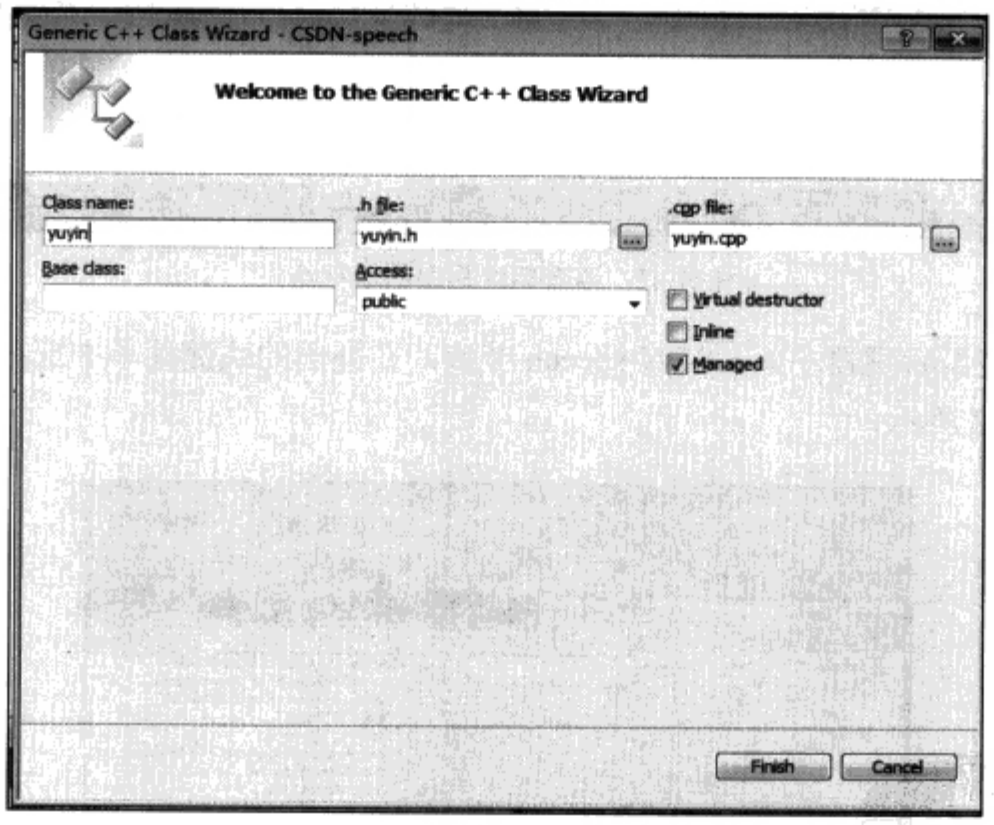


图 21.44　新类命名为 yuyin 类

（13）为 yuyin.h 添加如下代码，代码的具体意义见注释：

```
#pragma once
using namespace System::Linq;
using namespace System::Text;
using namespace System::Speech;
using namespace System::Speech::Synthesis;
using namespace System::Speech::Recognition;
using namespace System::Globalization;
using namespace System::Windows::Forms;
```

```
namespace CSDNspeech {
        public ref class yuyin
        {
       public:
            SpeechRecognitionEngine ^recognizer;                                    //定义语音识别器
            DictationGrammar ^dictationGrammar;                                     //定义语法
            System::Windows::Forms::Control ^cDisplay;                              //定义一个关联文本控件
            yuyin(array<System::String^> ^fg);                                      //创建关键词语列表
       private:
            void InitializeSpeechRecognitionEngine(array<System::String^> ^fg);
       public:
            void BeginRec(Control ^tbResult);                                       //开始语音识别，关联窗口控件
            void over();                                                            //结束语音识别
            virtual Grammar ^CreateCustomGrammar(array<System::String^> ^fg);
       private:
            void TurnSpeechRecognitionOn();                                         //开启语音识别
            void TurnSpeechRecognitionOff();                                        //关闭语音识别
            void recognizer_SpeechHypothesized(System::Object ^sender, SpeechHypothesizedEventArgs ^e);
            void recognizer_SpeechRecognized(System::Object ^sender, SpeechRecognizedEventArgs ^e); //语音识别器
            void TurnDictationOn();                                                 //打开词典
            void TurnDictationOff();                                                //关闭词典
       };
}
```

（14）将 yuyin.cpp 文件的代码更新为下列代码，代码的具体含义见注释：

```
#include "StdAfx.h"
#include "yuyin.h"
using namespace System;
using namespace System::Collections::Generic;
using namespace System::Linq;
using namespace System::Text;
using namespace System::Speech;
using namespace System::Speech::Synthesis;
using namespace System::Speech::Recognition;
using namespace System::Globalization;
using namespace System::Windows::Forms;
namespace CSDNspeech {
      yuyin::yuyin(array<System::String^> ^fg)                                  //fg 为语音识别关键词语列表
      {
          CultureInfo ^myCIintl = gcnew CultureInfo("zh-CN", false);                        //创建简体中文标示
          foreach (RecognizerInfo ^config in SpeechRecognitionEngine::InstalledRecognizers()) //获取所有语音引擎
          {
              if (config->Culture->Equals(myCIintl) && config->Id == "MS-2052-80-DESK")  //选择中文语音识别引擎
              {
                    recognizer = gcnew SpeechRecognitionEngine(config);
                    break;
              }
          }
          if (recognizer != nullptr)
          {
```

```
            InitializeSpeechRecognitionEngine(fg);                          //选择默认语音引擎
            dictationGrammar = gcnew DictationGrammar();
        }
        else
        {
            MessageBox::Show("创建语音识别失败");
        }
    }
    void yuyin::InitializeSpeechRecognitionEngine(array<System::String^> ^fg)    //语音识别初始化
    {
        recognizer->SetInputToDefaultAudioDevice();                         //选择默认语音设备
        Grammar ^customGrammar = CreateCustomGrammar(fg);                   //定义语法
        recognizer->UnloadAllGrammars();
        recognizer->LoadGrammar(customGrammar);
        recognizer->SpeechRecognized += gcnew EventHandler<SpeechRecognizedEventArgs^>(this, &yuyin::
recognizer_SpeechRecognized);
        recognizer->SpeechHypothesized += gcnew EventHandler<SpeechHypothesizedEventArgs^>(this, &yuyin::
recognizer_SpeechHypothesized);
    }
    void yuyin::BeginRec(Control ^tbResult)
    {
        TurnSpeechRecognitionOn();
        TurnDictationOn();
        cDisplay = tbResult;
    }
    void yuyin::over()
    {
        TurnSpeechRecognitionOff();
    }
    Grammar ^yuyin::CreateCustomGrammar(array<System::String^> ^fg)          //创建语法
    {
        GrammarBuilder ^grammarBuilder = gcnew GrammarBuilder();
        grammarBuilder->Append(gcnew Choices(fg));
        return gcnew Grammar(grammarBuilder);
    }
    void yuyin::TurnSpeechRecognitionOn()
    {
        if (recognizer != nullptr)
        {
            recognizer->RecognizeAsync(RecognizeMode::Multiple);
        }
        else
        {
            MessageBox::Show("创建语音识别失败");
        }
    }
    void yuyin::TurnSpeechRecognitionOff()
    {
        if (recognizer != nullptr)
```

```
        {
            recognizer->RecognizeAsyncStop();
            TurnDictationOff();
        }
        else
        {
            MessageBox::Show("创建语音识别失败");
        }
    }
    //语音识别状态
    void yuyin::recognizer_SpeechHypothesized(System::Object ^sender, SpeechHypothesizedEventArgs ^e)
    {
    }
    //显示语音识别结果
    void yuyin::recognizer_SpeechRecognized(System::Object ^sender, SpeechRecognizedEventArgs ^e)
    {
        System::String ^text = e->Result->Text;
        cDisplay->Text = text;
    }
    void yuyin::TurnDictationOn()
    {
        if (recognizer != nullptr)
        {
            recognizer->LoadGrammar(dictationGrammar);
        }
        else
        {
            MessageBox::Show("创建语音识别失败");
        }
    }
    void yuyin::TurnDictationOff()
    {
        if (dictationGrammar != nullptr)
        {
            recognizer->UnloadGrammar(dictationGrammar);
        }
        else
        {
            MessageBox::Show("创建语音识别失败");
        }
    }
}
```

（15）在 Form1.h 中添加头文件支持：

```
#include "yuyin.h"
```

（16）选中“开启识别”按钮，为其添加下列代码：

```
array<System::String^> ^fg ;
fg = gcnew array<System::String ^> {"尹成","中国","专家","技术"};
//定义一个优先识别的词组
```

```
yuyin^   sdk;
sdk =gcnew yuyin(fg);
sdk->BeginRec(this->textBox1);
```

（17）按【F5】键，打开麦克风试听说话效果，如图 21.45 所示。

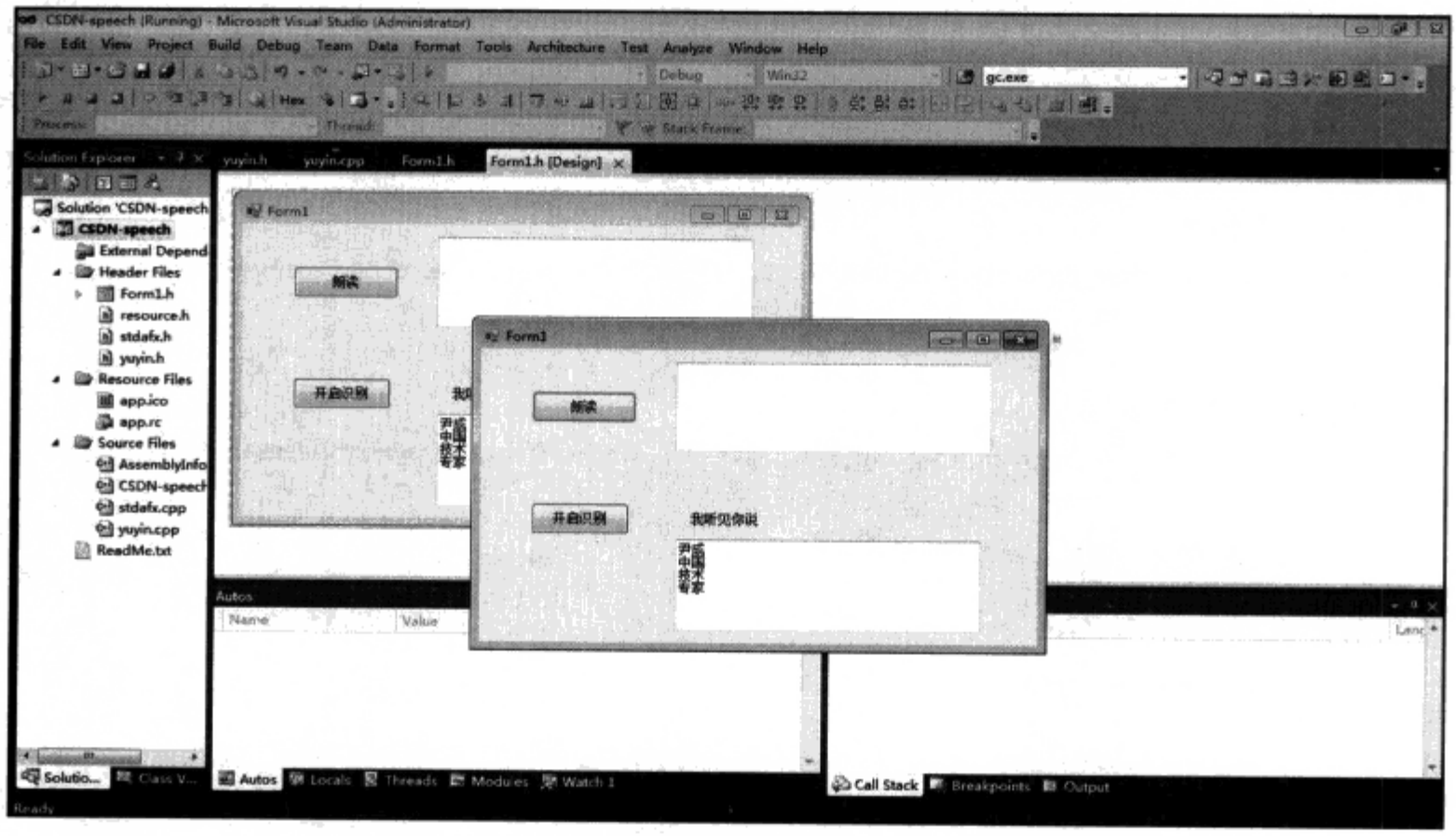


图 21.45　语音识别程序运行结果

Visual C++ 2010 中 MFC 程序兼容 CLR 代码（MFC 与.NET 4.0 混合编程），可以在 MFC 中调用 CLR 代码（不局限于语音朗读与语音合成，可以为旧的 MFC 程序添加.NET 4.0 功能）。

（1）打开 Visual Studio 2010，创建一个名为 csdndialog 的 MFC 对话框程序，采用默认设置，如图 21.46 和图 21.47 所示。

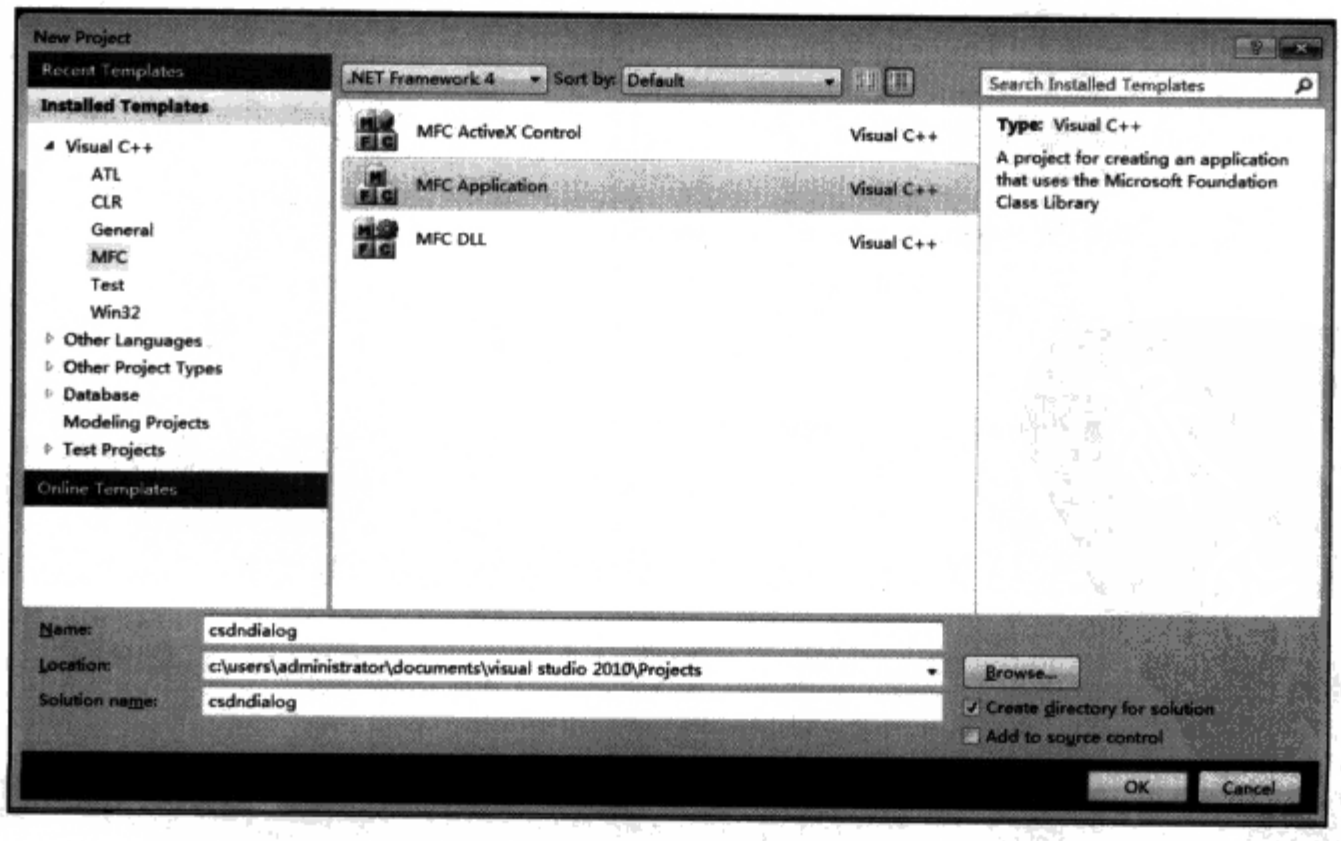


图 21.46　新建一个 MFC 运用程序

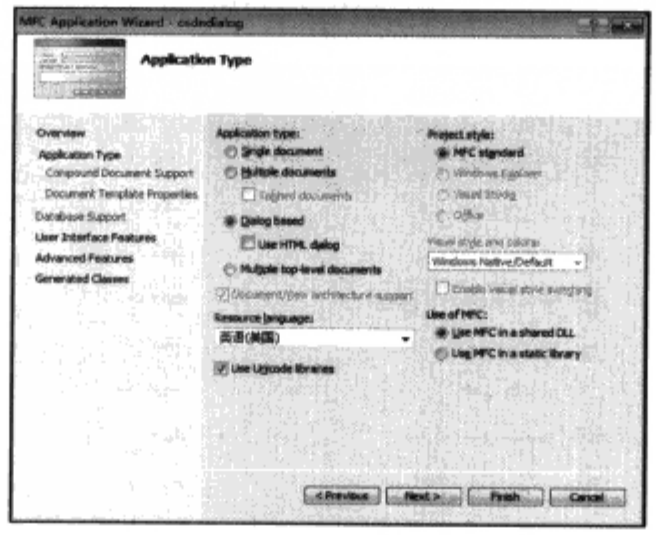


图 21.47　选择对话框类型

（2）在 MFC 对话框中添加一个按钮，将其名称改为“MFC 里面玩 CLR”，如图 21.48 所示。

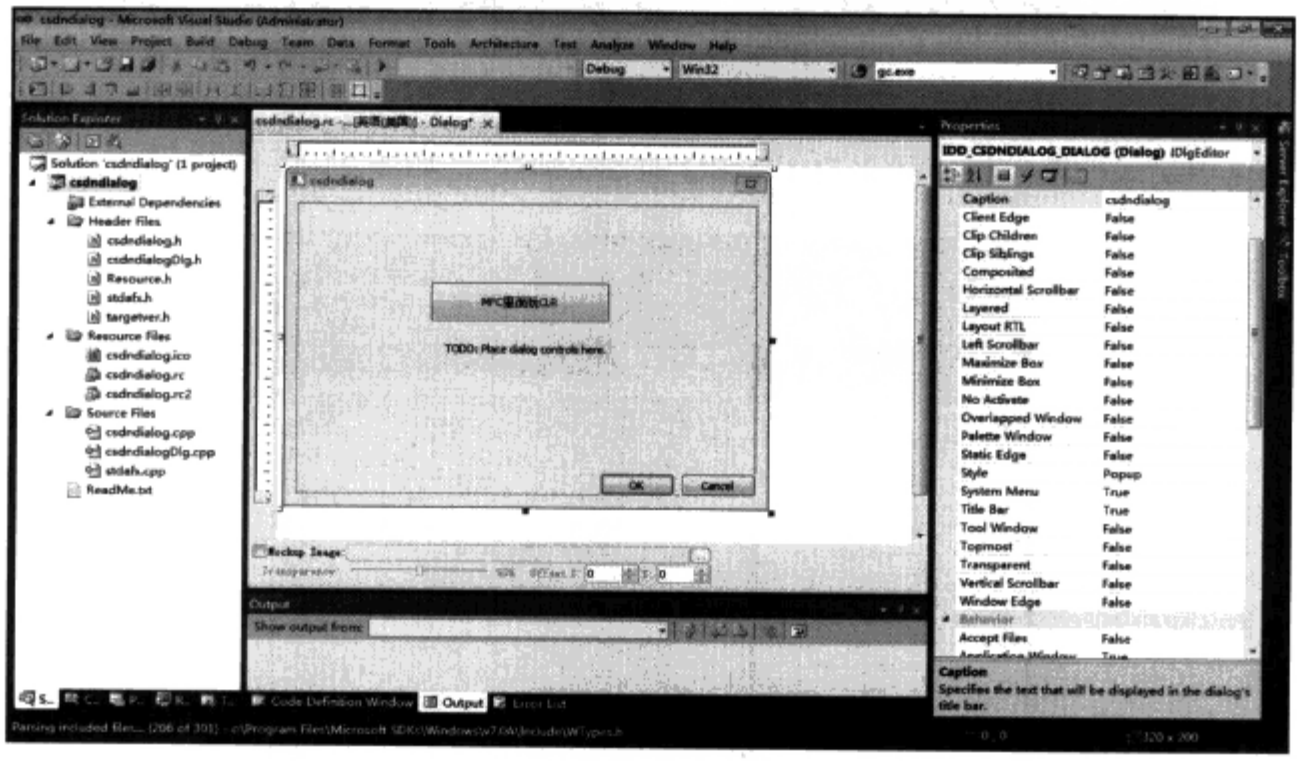


图 21.48　程序设计界面

（3）在配置属性中选择 General 项，然后选择 Common Language Runtime Support（/clr）选项，如图 21.49 所示。

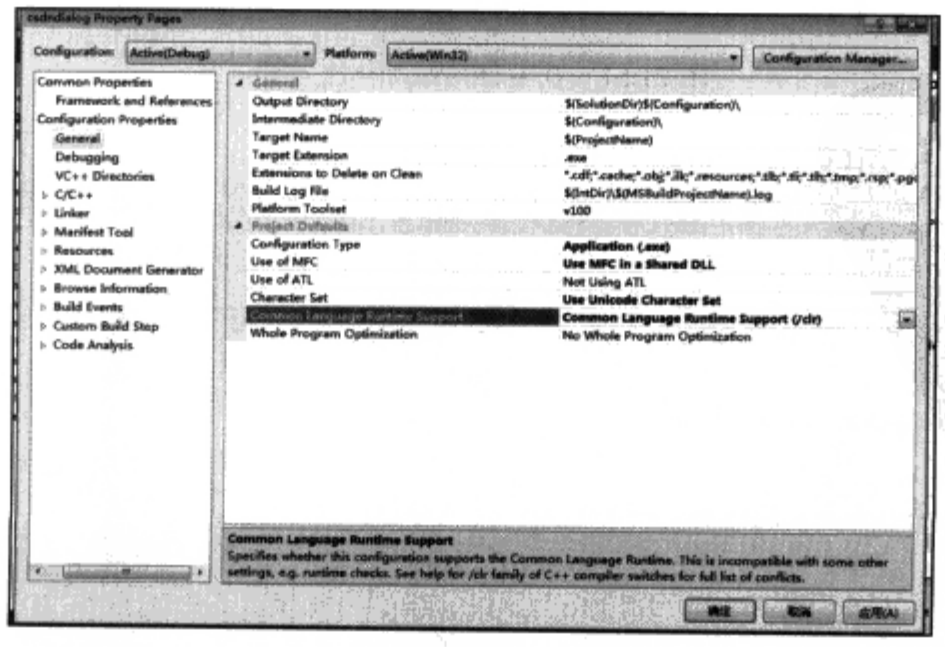


图 21.49　选择 CLR 支持

（4）为项目添加一个项，选择创建一个 Windows Form，命名为 csdn，如图 21.50 所示。

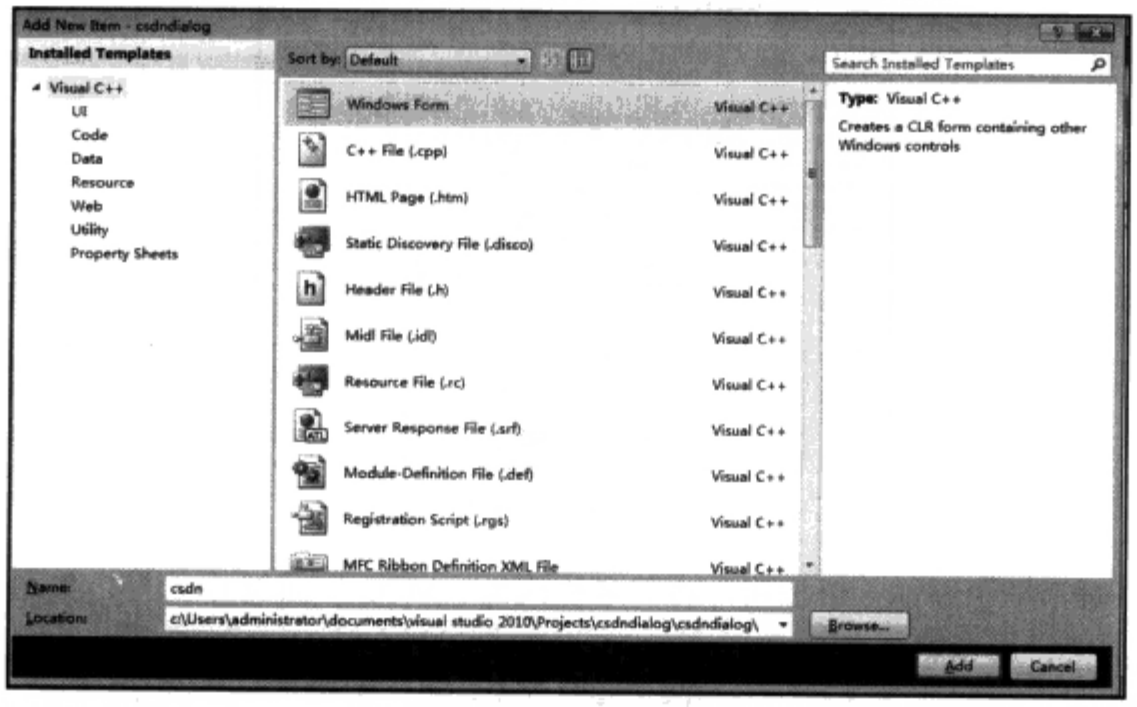


图 21.50　添加一个新的 Form 窗体

（5）在窗体中插入一个按钮“我来自于 MFC 的 diaolog”，如图 21.51 所示。

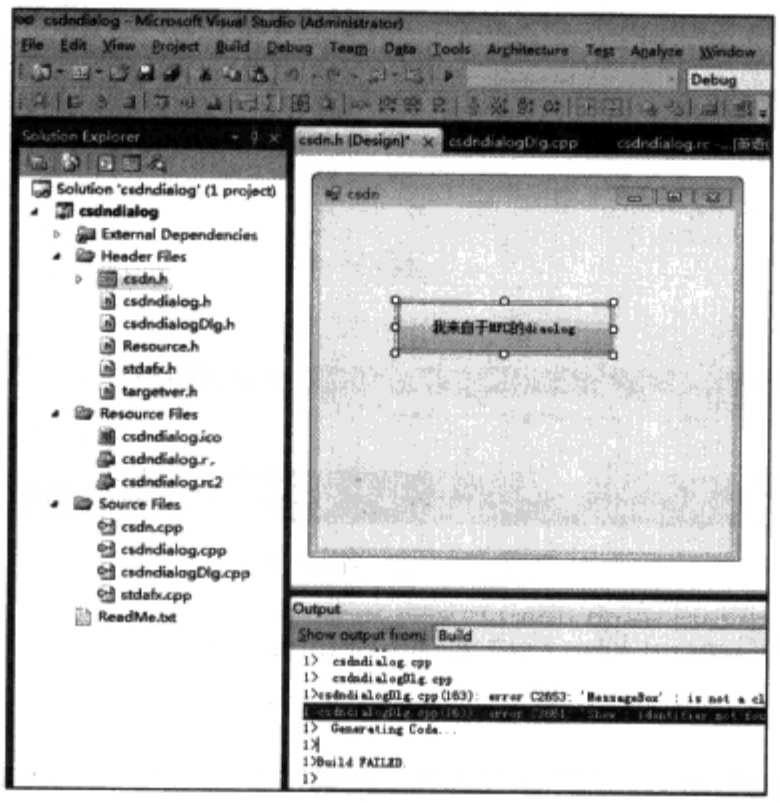


图 21.51　往窗体内添加一个按钮

（6）双击“我来自于 MFC 的 diaolog”按钮，添加代码：

```
MessageBox::Show(L"我是 CSDN 著名技术尹成的杰作");
```

（7）双击“MFC 里面玩 CLR”按钮，在头文件支持中添加如下代码：

```
#include"csdn.h"
using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
using namespace csdndialog;
```

```
csdn ^ ff=gcnew csdn();
ff->Show();
```

（8）按【F5】键，编译执行，分别单击按钮查看效果，如图 21.52 所示。

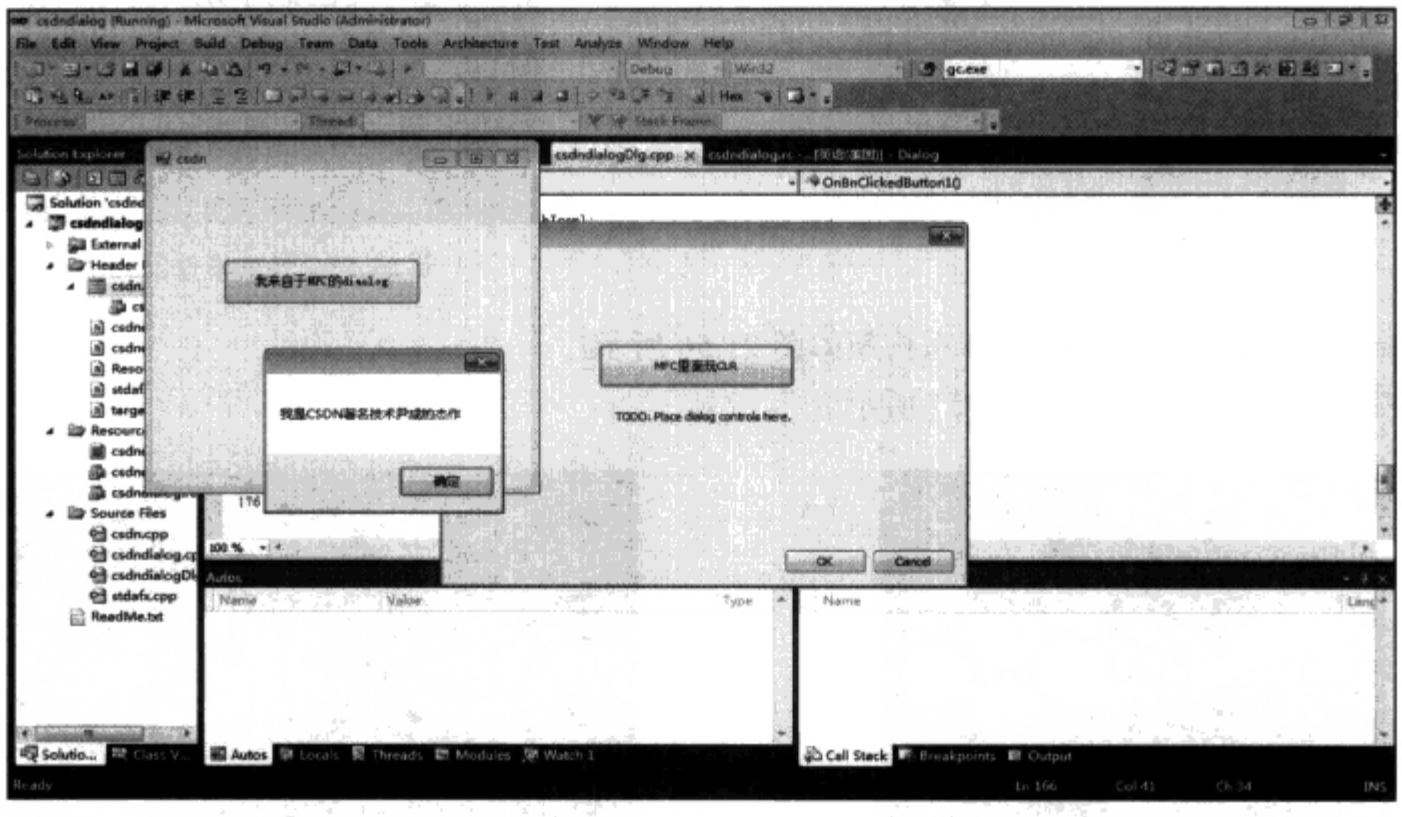


图 21.52　程序运行结果

读者只要看懂了整个示例，就知道如何在 MFC 中调用 CLR 实现语音识别、语音朗读功能了。同样，.NET 4.0 的所有功能都可以由 MFC 来调用，同时 MFC 的老代码可以借助.NET 恢复活力。在未来，Windows 7 必然占据操作系统的主流。MFC 与.NET 都是重要的开发平台，融合是必然趋势。

21.3.5　基于 Visual C++ 2010 与 Windows SDK for Windows 7 开发 Windows 7 平台的 Tablet PC 应用

图 21.53 所示为微软在操作系统中增加的 Tablet PC 功能，其中一个很重要的功能就是数字墨水。

图 21.53　Windows 7 平台上的 tabletpc 应用

Windows 7 在便携化、人性化方面有了很大改进，如图 21.54 和图 21.55 所示。其中数字墨水就是一个很重要的功能。

图 21.54　开发支持

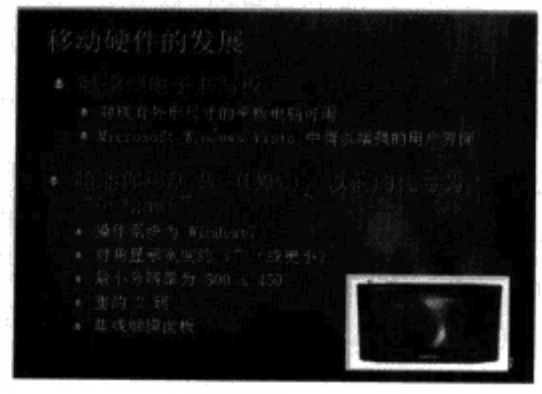


图 21.55　移动硬件发展

Windows 7 的 Tablet PC API 体系构架如图 21.56 所示。

“数字墨水”对象简介如图 21.57 所示。

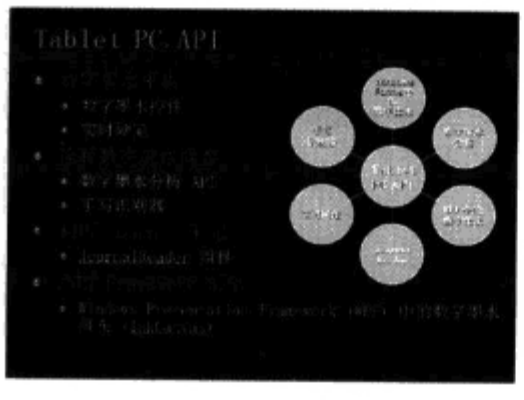


图 21.56　Tablet PC 对象介绍

图 21.57　数字墨水对象

“笔划”对象简介如图 21.58 所示。

“笔画集”对象介绍如图 21.59 所示。

图 21.58　笔画对象

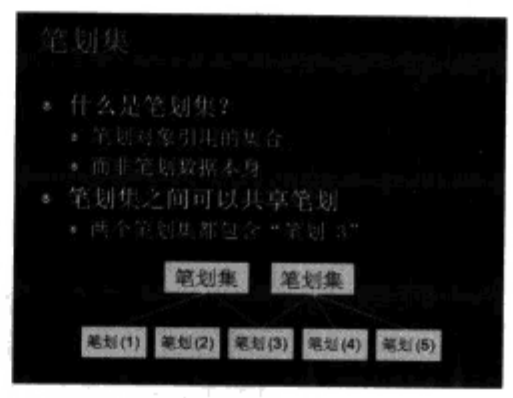


图 21.59　笔画集合

数字墨水分析如图 21.60 所示。

数字墨水附加功能如图 21.61 所示。

图 21.60　数字墨水分析

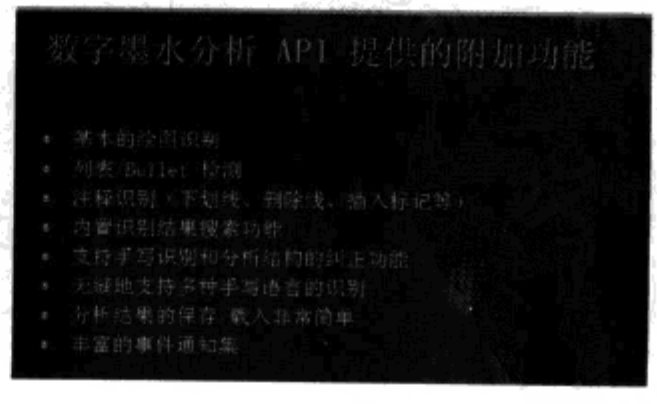


图 21.61　数字墨水 API

下面基于 Visual C++ 2010 与 Windows SDK for Windows 7 开发一个应用，以演示 Tablet PC 所有

的功能。

先看看程序效果。

（1）主窗口如图 21.62 所示。

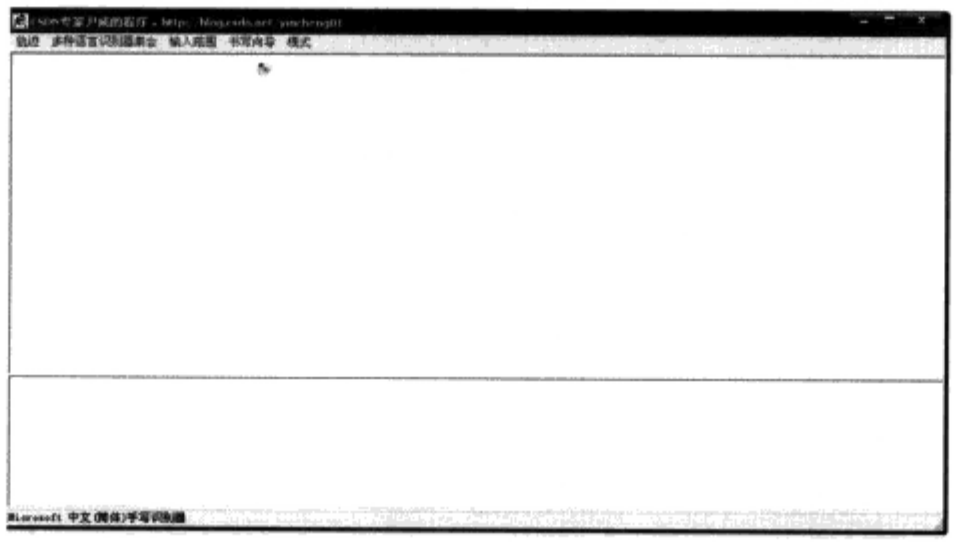


图 21.62　程序视图

（2）12 个手写语言识别器如图 21.63 所示。

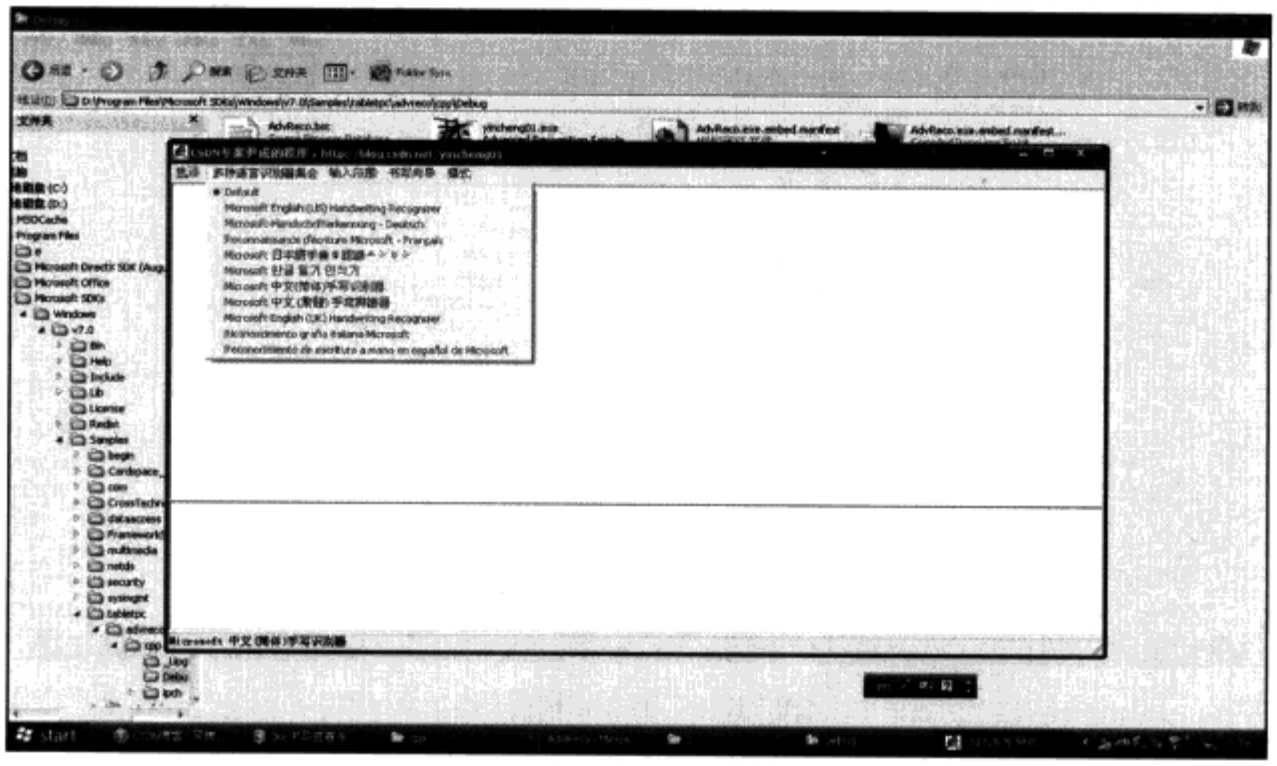


图 21.63　菜单视图

（3）英文识别范例如图 21.64 所示。

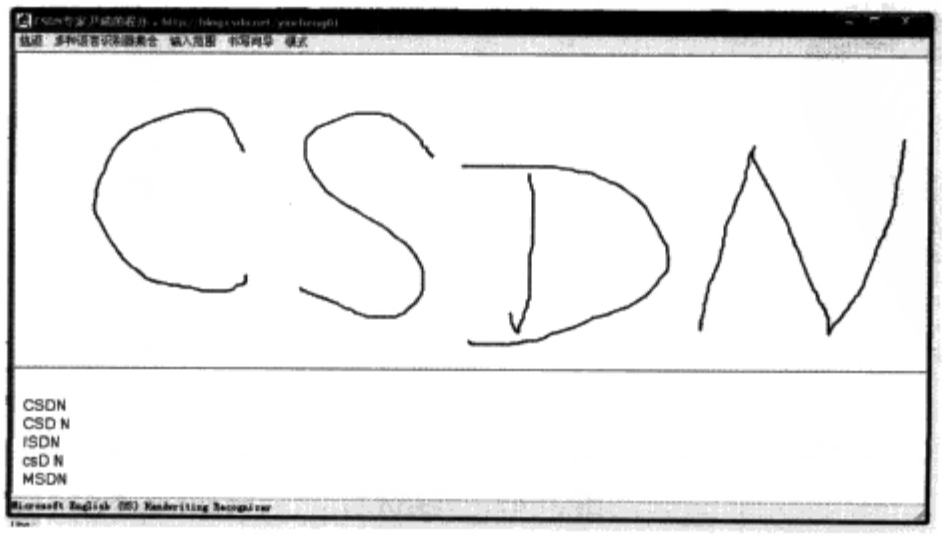


图 21.64　英文识别范例

（4）韩文识别范例如图 21.65 所示。

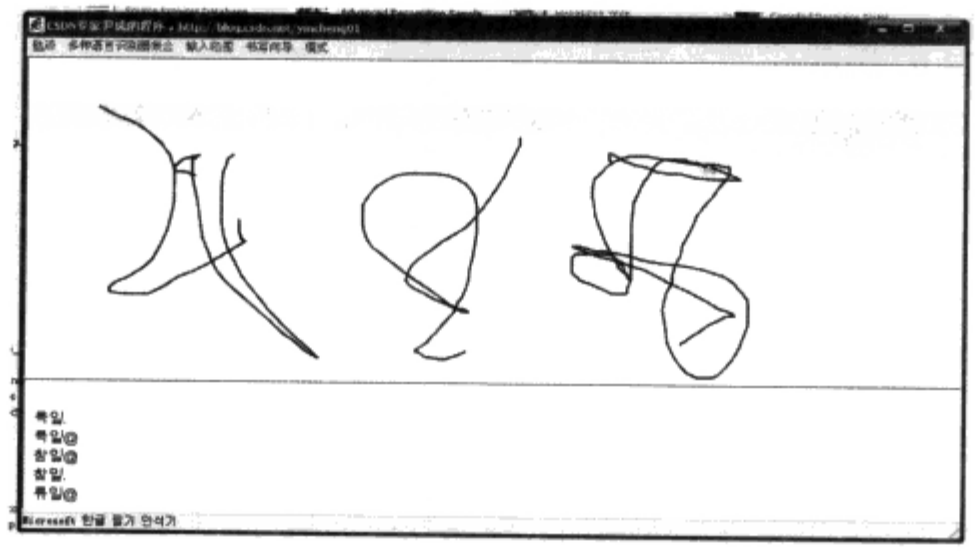


图 21.65　韩文识别范例

（5）中文识别范例如图 21.66 所示。

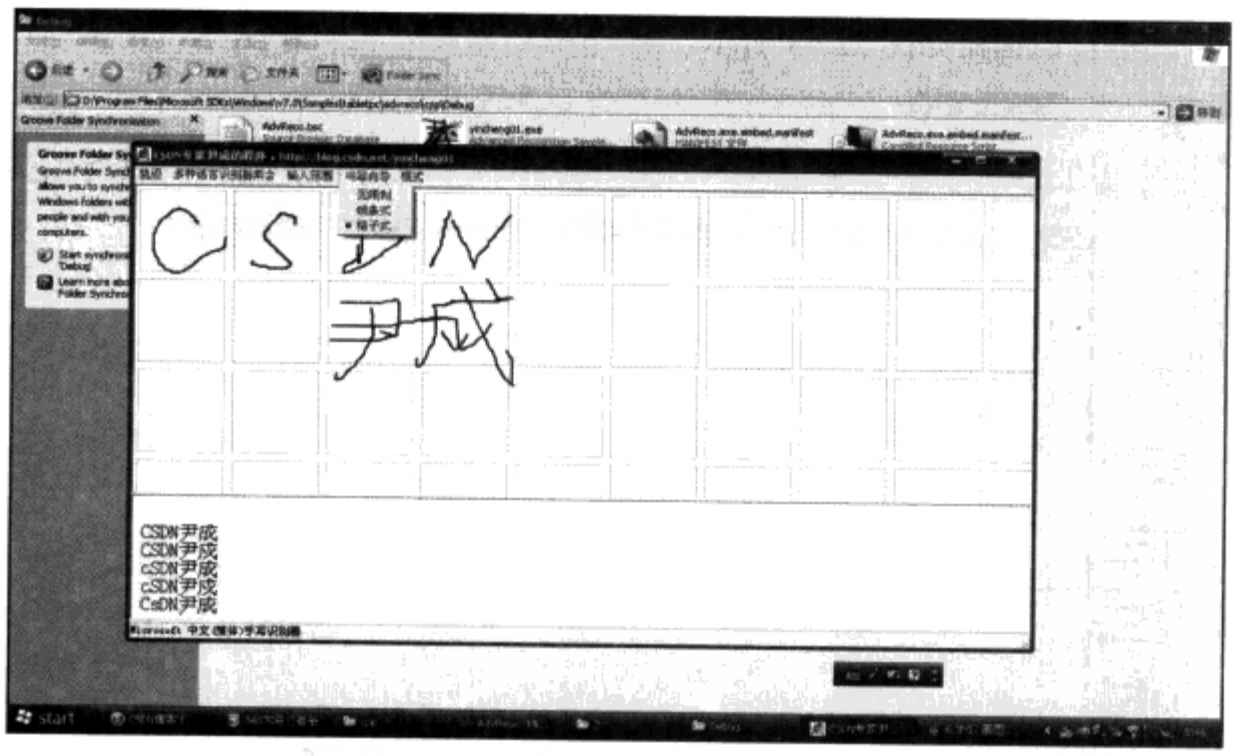


图 21.66　中文识别范例

（6）笔势分析如图 21.67 所示。

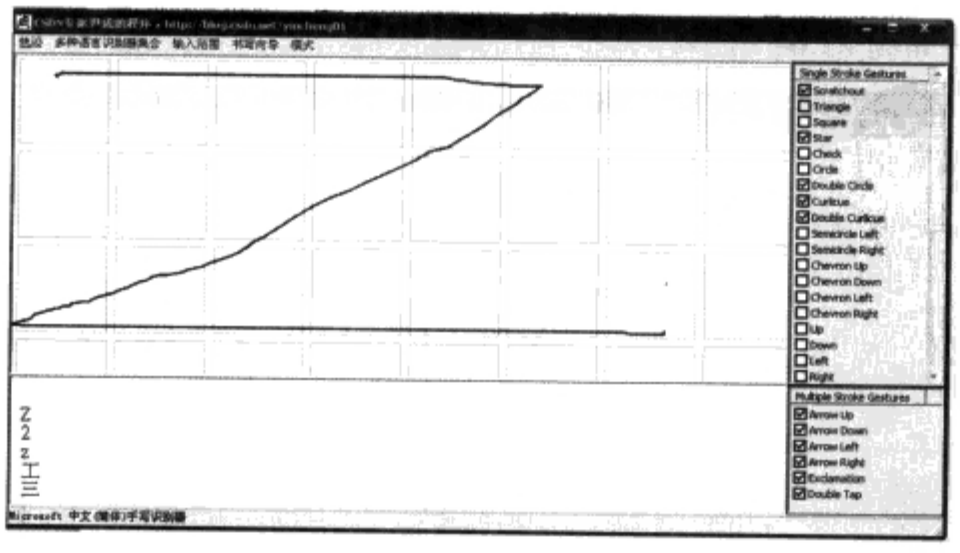


图 21.67　笔势分析

核心源码及分析（详细注释）如下：

```
enum {
    //子指数
    mc_iSubmenuRecognizers = 1,
    mc_iSubmenuInputScopes = 2,
    mc_iSubmenuGuides = 3,
    mc_iSubmenuModes = 4,
    // child windows IDs
    mc_iInputWndId = 1,
    mc_iOutputWndId = 2,
    mc_iStatusWndId = 3,
    mc_iSSGestLVId = 4,
    mc_iMSGestLVId = 5,
    //识别指南框数据
    mc_iNumRowsCols = 100,
    mc_iGuideColWidth = 100,
    mc_iGuideRowHeight = 100,
    mc_cxBoxMargin = 4,
    mc_cyBoxMargin = 4,
    //t 宽度
    mc_cxGestLVWidth = 160,
    //格子数目
    mc_cNumSSGestures = 36,                    // single stroke gestures
    mc_cNumMSGestures =6,                      // multi-stroke gestures
    //笔的笔迹宽度
    mc_iPenWidth = 5
};
// COM 指针，手写引擎的对象指针
CComPtr<IInkCollector>m_spIInkCollector;
CComPtr<IInkRenderer> m_spIInkRenderer;
CComPtr<IInkDisp> m_spIInkDisp;
CComPtr<IInkStrokes>m_spIInkStrokes;
CComPtr<IInkRecognizerContext>m_spIInkRecoContext;
CComPtr<IInkRecognizers>m_spIInkRecognizers;
CComPtr<IInkRecognizerGuide>m_spIInkRecoGuide;
//子窗口
CInkInputWndm_wndInput;
CRecoOutputWndm_wndResults;
HWNDm_hwndStatusBar;
HWNDm_hwndSSGestLV;
HWNDm_hwndMSGestLV;
//定义数据变量
UINTm_nCmdRecognizer;
UINTm_nCmdInputScope;
UINTm_nCmdGuide;
UINTm_nCmdMode;
CComBSTRm_bstrCurRecoName;
boolm_bCoerceInputScope;
SIZEm_szGuideBox;
boolm_bAllSSGestures;
```

```
boolm_bAllMSGestures;
//消息函数
LRESULT OnCreate(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);
LRESULT OnDestroy(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);
LRESULT OnSetFocus(UINT, WPARAM, LPARAM, BOOL& bHandled);
LRESULT OnSize(UINT, WPARAM, LPARAM, BOOL& bHandled);
LRESULT OnLVColumnClick(int idCtrl, LPNMHDR pnmh, BOOL& bHandled);
LRESULT OnLVItemChanging(int idCtrl, LPNMHDR pnmh, BOOL& bHandled);
//命令句柄
LRESULT OnRecognizer(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);
LRESULT OnInputScopeCoerce(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);
LRESULT OnInputScope(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);
LRESULT OnGuide(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);
LRESULT OnMode(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);
LRESULT OnRecognize(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);
LRESULT OnClear(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);
LRESULT OnExit(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);
//笔迹句柄
HRESULT OnStroke(IInkCursor* pIInkCursor, IInkStrokeDisp* pIInkStroke,
 VARIANT_BOOL* pbCancel);
HRESULT OnGesture(IInkCursor* pIInkCursor, IInkStrokes* pIInkStrokes,
VARIANT vGestures, VARIANT_BOOL* pbCancel);
//识别器句柄
HRESULT OnRecognitionWithAlternates(IInkRecognitionResult* pIInkRecoResult,
VARIANT vCustomParam,
InkRecognitionStatus RecognitionStatus);
include <windows.h>
#include <commctrl.h>                    //COM 指针所需头文件
#ifndef ListView_SetCheckState
#define ListView_SetCheckState(hwndLV, i, fCheck) \
ListView_SetItemState(hwndLV, i, INDEXTOSTATEIMAGEMASK((fCheck ? 2 : 1)), LVIS_STATEIMAGEMASK)
#endif
#ifndef ListView_GetCheckState
#define ListView_GetCheckState(hwndLV, i) \
((((UINT)(SNDMSG((hwndLV), LVM_GETITEMSTATE, (WPARAM)(i), LVIS_STATEIMAGEMASK))) >> 12) -1)
#endif
#ifndef MIIM_STRING
#define MIIM_STRING0x00000040
#endif
#ifndef MIIM_FTYPE
#define MIIM_FTYPE 0x00000100
#endif
//定义数组容纳笔势
#define countof(array)(sizeof(array)/sizeof(array[0]))
#include <atlbase.h>/
CComModule _Module;
#include <atlwin.h>
#include <atlcom.h>
// Tablet PC 头文件
```

```
#include <msinkaut.h>
#include <msinkaut_i.c>
#include <tpcerror.h>
#include "resource.h"
#include "EventSinks.h"
#include "ChildWnds.h"
#include "AdvReco.h"
//定义初始化变量
const LPOLESTR gc_pwsInputScopes[] = {
    L"(!IS_DEFAULT)",
    L"(!IS_URL)",
    L"(!IS_FILE_FULLFILEPATH)",
    L"(!IS_FILE_FILENAME)",
    L"(!IS_EMAIL_USERNAME)",
    L"(!IS_EMAIL_SMTPEMAILADDRESS)",
    L"(!IS_LOGINNAME)",
    L"(!IS_PERSONALNAME_FULLNAME)",
    L"(!IS_PERSONALNAME_PREFIX)",
    L"(!IS_PERSONALNAME_GIVENNAME)",
    L"(!IS_PERSONALNAME_MIDDLENAME)",
    L"(!IS_PERSONALNAME_SURNAME)",
    L"(!IS_PERSONALNAME_SUFFIX)",
    L"(!IS_ADDRESS_FULLPOSTALADDRESS)",
    L"(!IS_ADDRESS_POSTALCODE)",
    L"(!IS_ADDRESS_STREET)",
    L"(!IS_ADDRESS_STATEORPROVINCE)",
    L"(!IS_ADDRESS_CITY)",
    L"(!IS_ADDRESS_COUNTRYNAME)",
    L"(!IS_ADDRESS_COUNTRYSHORTNAME)",
    L"(!IS_CURRENCY_AMOUNTANDSYMBOL)",
    L"(!IS_CURRENCY_AMOUNT)",
    L"(!IS_DATE_FULLDATE)",
    L"(!IS_DATE_MONTH)",
    L"(!IS_DATE_DAY)",
    L"(!IS_DATE_YEAR)",
    L"(!IS_DATE_MONTHNAME)",
    L"(!IS_DATE_DAYNAME)",
    L"(!IS_DIGITS)",
    L"(!IS_NUMBER)",
    L"(!IS_ONECHAR)",
    L"(!IS_TELEPHONE_FULLTELEPHONENUMBER)",
    L"(!IS_TELEPHONE_COUNTRYCODE)",
    L"(!IS_TELEPHONE_AREACODE)",
    L"(!IS_TELEPHONE_LOCALNUMBER)",
    L"(!IS_TIME_FULLTIME)",
    L"(!IS_TIME_HOUR)",
    L"(!IS_TIME_MINORSEC)",
    L"((0|1|2|3|4|5|6|7|8|9) ?(0|1|2|3|4|5|6|7|8|9) ?(0|1|2|3|4|5|6|7|8|9) ?-? ?)?(0|1|2|3|4|5|6|7|8|9) ?(0|1|2|3|4|5|6|7|8|9) ?(0|1|2|3|
4|5|6|7|8|9) ?-? ?(0|1|2|3|4|5|6|7|8|9) ?(0|1|2|3|4|5|6|7|8|9) ?(0|1|2|3|4|5|6|7|8|9) ?(0|1|2|3|4|5|6|7|8|9)",
```

```
        L"(!IS_PERSONALNAME_FULLNAME)((!IS_PERSONALNAME_PREFIX)? +(!IS_PERSONALNAME_GIVENNAME)
+ +(!IS_PERSONALNAME_MIDDLENAME)* +(!IS_PERSONALNAME_SURNAME)+)",
        L"MN(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)(A|B)(!IS_DIGITS)(X|Y)((0|1)*)"
    };
    const LONG gc_lMaxInputScopeMenuItemLength = 40;
    //定义笔势常量
    const InkApplicationGesture gc_igtSingleStrokeGestures[] = {
        IAG_Scratchout, IAG_Triangle, IAG_Square, IAG_Star, IAG_Check,
        IAG_Circle, IAG_DoubleCircle, IAG_Curlicue, IAG_DoubleCurlicue,
        IAG_SemiCircleLeft, IAG_SemiCircleRight,
        IAG_ChevronUp, IAG_ChevronDown, IAG_ChevronLeft,
        IAG_ChevronRight, IAG_Up, IAG_Down, IAG_Left, IAG_Right, IAG_UpDown, IAG_DownUp,
        IAG_LeftRight, IAG_RightLeft, IAG_UpLeftLong, IAG_UpRightLong, IAG_DownLeftLong,
        IAG_DownRightLong, IAG_UpLeft, IAG_UpRight, IAG_DownLeft, IAG_DownRight, IAG_LeftUp,
        IAG_LeftDown, IAG_RightUp, IAG_RightDown, IAG_Tap
    };
    const UINT gc_nRecommendedForMixedMode[] = {
        0 /*Scratchout*/, 3/*Star*/, 6/*Double Circle*/,
        7 /*Curlicue*/, 8 /*Double Curlicue*/, 25 /*Down-Left Long*/ };
        const InkApplicationGesture gc_igtMultiStrokeGestures[] = {
        IAG_ArrowUp, IAG_ArrowDown, IAG_ArrowLeft,
        IAG_ArrowRight, IAG_Exclamation, IAG_DoubleTap
    };
    const _ATL_FUNC_INFO IInkRecognitionEventsImpl<CAdvRecoApp>::mc_AtlFuncInfo =
        {CC_STDCALL, VT_EMPTY, 3, {VT_UNKNOWN, VT_VARIANT, VT_I4}};
    const _ATL_FUNC_INFO IInkCollectorEventsImpl<CAdvRecoApp>::mc_AtlFuncInfo[2] = {
        {CC_STDCALL, VT_EMPTY, 3, {VT_UNKNOWN, VT_UNKNOWN, VT_BOOL|VT_BYREF}},
        {CC_STDCALL, VT_EMPTY, 4, {VT_UNKNOWN, VT_UNKNOWN, VT_VARIANT, VT_BOOL|VT_BYREF}}
};
const TCHAR gc_szAppName[] = TEXT("CSDN 专家尹成的程序，http://blog.csdn.net/yincheng01");
//程序初始化窗口
int APIENTRY WinMain(
    HINSTANCE hInstance,
    HINSTANCE /*hPrevInstance*/, // not used here
    LPSTR /*lpCmdLine*/, // not used here
    int nCmdShow
    )
{
    int iRet = 0;
    //初始化 COM
    if (S_OK == ::CoInitializeEx(NULL, COINIT_APARTMENTTHREADED))
    {
        _Module.Init(NULL, hInstance);
        //注册空间窗口
        INITCOMMONCONTROLSEX icc;
        icc.dwSize = sizeof(icc);
        icc.dwICC = ICC_LISTVIEW_CLASSES | ICC_BAR_CLASSES;
        if (TRUE == ::InitCommonControlsEx(&icc))
        {
```

```
            iRet = CAdvRecoApp::Run(nCmdShow);
        }
        else
        {
            ::MessageBox(NULL, TEXT("Error."),
             gc_szAppName, MB_ICONERROR | MB_OK);
        }
        _Module.Term();
        ::CoUninitialize();
    }
    return iRet;
}
//程序启动窗口
int CAdvRecoApp::Run(
    int nCmdShow
    )
{
    CAdvRecoApp theApp;
    HMENU hMenu = theApp.LoadMenu();
    if (NULL == hMenu)
        return 0;
    int iRet;
    WNDCLASSEX& wc = CAdvRecoApp::GetWndClassInfo().m_wc;
        wc.hIcon = wc.hIconSm = ::LoadIcon(_Module.GetResourceInstance(),
        MAKEINTRESOURCE(IDR_APPICON));
    if (theApp.Create(NULL, CWindow::rcDefault, gc_szAppName,
        WS_OVERLAPPEDWINDOW, 0, (UINT)hMenu) != NULL)
{
    theApp.SendMessage(WM_COMMAND, ID_GUIDE_NONE);
    //创建程序初始化消息
    theApp.SendMessage(WM_COMMAND, ID_INPUTSCOPE_FIRST);
    //创建默认识别器
    theApp.SendMessage(WM_COMMAND, ID_RECOGNIZER_DEFAULT);
    //设置收集数据识别器
    theApp.SendMessage(WM_COMMAND, ID_MODE_INK);
    theApp.ShowWindow(nCmdShow);
    theApp.UpdateWindow();
    //开始消息循环
    MSG msg;
    while (::GetMessage(&msg, NULL, 0, 0) > 0)
    {
        ::TranslateMessage(&msg);
        ::DispatchMessage(&msg);
    }
    iRet = msg.wParam;
}
else
{
    ::MessageBox(NULL, TEXT("Error "), gc_szAppName, MB_ICONERROR | MB_OK);
```

```
            ::DestroyMenu(hMenu);
            iRet = 0;
        }
        return iRet;
}
// Window message 传递
        LRESULT CAdvRecoApp::OnCreate(
        UINT /*uMsg*/,
        WPARAM /*wParam*/,
        LPARAM /*lParam*/,
        BOOL& /*bHandled*/
        )
{
        if (false == CreateChildWindows())
            return -1;
        HRESULT hr;
        //初始化 COM 指针 CLSID_InkCollector
        hr = m_spIInkCollector.CoCreateInstance(CLSID_InkCollector);
        if (FAILED(hr))
            return -1;
        //初始化 COM 指针 m_spIInkDisp
        hr = m_spIInkCollector->get_Ink(&m_spIInkDisp);
        if (FAILED(hr))
            return -1;
        //初始化 com 指针 m_spIInkStrokes
        hr = m_spIInkDisp->get_Strokes(&m_spIInkStrokes);
        if (FAILED(hr))
            return -1;
        //挂接笔迹事件
        hr = IInkCollectorEventsImpl<CAdvRecoApp>::DispEventAdvise(m_spIInkCollector);
        if (FAILED(hr))
            return -1;
        //设置笔势
        PresetGestures();
        //启动书写
        hr = m_spIInkCollector->put_hWnd((long)m_wndInput.m_hWnd);
        if (FAILED(hr))
            return -1;
        hr = m_spIInkCollector->put_Enabled(VARIANT_TRUE);
        if (FAILED(hr))
            return -1;
        //创建识别器向导
        if (SUCCEEDED(m_spIInkRecoGuide.CoCreateInstance(CLSID_InkRecognizerGuide)))
        {
            _InkRecoGuide irg;
            irg.midline = -1;                                    //不用格子
            irg.cRows = irg.cColumns = 0;                        //不用划线
            ::SetRect(&irg.rectWritingBox, 0, 0, mc_iGuideColWidth, mc_iGuideRowHeight);
            //M 创建书写区域
```

```
        irg.rectDrawnBox = irg.rectWritingBox;
        ::InflateRect(&irg.rectDrawnBox, -mc_cxBoxMargin, -mc_cyBoxMargin);
        m_wndInput.SetGuide(irg);
        //收集笔迹区域
        hr = m_spIInkCollector->get_Renderer(&m_spIInkRenderer);
        if (SUCCEEDED(hr))
        {
            HDC hdc = m_wndInput.GetDC();
            if (NULL != hdc)
            {
                //转换笔迹
                m_spIInkRenderer->PixelToInkSpace((long)hdc, &irg.rectWritingBox.left, &irg.rectWritingBox.top);
                m_spIInkRenderer->PixelToInkSpace((long)hdc, &irg.rectWritingBox.right, &irg.rectWritingBox.bottom);
                m_spIInkRenderer->PixelToInkSpace((long)hdc, &irg.rectDrawnBox.left, &irg.rectDrawnBox.top);
                m_spIInkRenderer->PixelToInkSpace((long)hdc, &irg.rectDrawnBox.right, &irg.rectDrawnBox.bottom);
                hr = m_spIInkRecoGuide->put_GuideData(irg);
                ReleaseDC(hdc);
            }
            else
            {
                hr = E_FAIL;
            }
        }
        if (FAILED(hr))
        {
            m_spIInkRecoGuide.Release();
        }
    }
    return 0;
}
//窗口销毁，删除所有对象
LRESULT CAdvRecoApp::OnDestroy(
    UINT /*uMsg*/,
    WPARAM /*wParam*/,
    LPARAM /*lParam*/,
    BOOL& /*bHandled*/
    )
{
    if (m_spIInkCollector != NULL)
    {
        IInkCollectorEventsImpl<CAdvRecoApp>::DispEventUnadvise(m_spIInkCollector);
        m_spIInkCollector->put_Enabled(VARIANT_FALSE);
        m_spIInkCollector.Release();
    }
    if (m_spIInkRecoContext != NULL)
    {
        m_spIInkRecoContext->EndInkInput();
        IInkRecognitionEventsImpl<CAdvRecoApp>::DispEventUnadvise(m_spIInkRecoContext);
        m_spIInkRecoContext.Release();
```

```
        }
        m_spIInkRecoGuide.Release();
        m_spIInkStrokes.Release();
        m_spIInkRecognizers.Release();
        ::PostQuitMessage(0);
        return 0;
    }
    //大小改变
    LRESULT CAdvRecoApp::OnSize(
        UINT /*uMsg*/,
        WPARAM wParam,
        LPARAM /*lParam*/,
        BOOL& /*bHandled*/
        )
    {
        if (wParam != SIZE_MINIMIZED)
        {
            UpdateLayout();
        }
        return 0;
    }
    // InkCollector 句柄指针事件
    HRESULT CAdvRecoApp::OnStroke(
        IInkCursor* /*pIInkCursor*/,
        IInkStrokeDisp* pIInkStroke,
        VARIANT_BOOL* /* pbCancel */
        )
    {
        if (NULL == pIInkStroke)
            return E_INVALIDARG;
        if (m_spIInkStrokes == NULL)
            return S_OK;
        HRESULT hr = m_spIInkStrokes->Add(pIInkStroke);
        if (SUCCEEDED(hr) && m_spIInkRecoContext != NULL)
        {
            m_spIInkRecoContext->StopBackgroundRecognition();
            CComVariant vCustomData;                    //更新识别结果
            m_spIInkRecoContext->BackgroundRecognizeWithAlternates(vCustomData);
        }
        return hr;
    }
    //笔势事件
    HRESULT CAdvRecoApp::OnGesture(
        IInkCursor* /*pIInkCursor*/,
        IInkStrokes* pInkStrokes,
        VARIANT vGestures,
        VARIANT_BOOL* pbCancel
        )
    {
```

```
    if (((VT_ARRAY | VT_DISPATCH) != vGestures.vt) || (NULL == vGestures.parray))
        return E_INVALIDARG;
      if (0 == vGestures.parray->rgsabound->cElements)
        return E_INVALIDARG;
//识别结果获取
LRESULT CAdvRecoApp::OnRecognizer(
    WORD /*wNotifyCode*/,
    WORD wID,
    HWND /*hWndCtl*/,
    BOOL& /*bHandled*/
    )
{
    if (m_spIInkRecognizers == NULL || wID == m_nCmdRecognizer)
        return 0;
    HRESULT hr;
    CComPtr<IInkRecognizer> spIInkRecognizer;
    if (ID_RECOGNIZER_DEFAULT == wID)
    {
        hr = m_spIInkRecognizers->GetDefaultRecognizer(0, &spIInkRecognizer);        //获取默认识别器设置之
    }
    else
    {
        hr = m_spIInkRecognizers->Item(wID - ID_RECOGNIZER_FIRST, &spIInkRecognizer);//否则获取第一个识别器
                                                                                     //设置之
    }
    if (SUCCEEDED(hr) && UseRecognizer(spIInkRecognizer))
    {
        UpdateMenuRadioItems(mc_iSubmenuRecognizers, wID, m_nCmdRecognizer);
        m_bstrCurRecoName.Empty();
        spIInkRecognizer->get_Name(&m_bstrCurRecoName);
        UpdateStatusBar();
        m_nCmdRecognizer = wID;
    }
    return 0;
}
//输入获取
LRESULT CAdvRecoApp::OnInputScopeCoerce(
    WORD /*wNotifyCode*/,
    WORD wID,
    HWND /*hWndCtl*/,
    BOOL& /*bHandled*/
    )
{
    if (m_spIInkRecoContext == NULL )
        return 0;
    if (m_spIInkStrokes != NULL)
    {
        m_spIInkRecoContext->putref_Strokes(NULL);
    }
```

```
if (FAILED(m_spIInkRecoContext->put_RecognitionFlags (m_bCoerceInputScope?IRM_None:IRM_Coerce)))
{
    MessageBox(TEXT("获取结果失败!"), gc_szAppName, MB_ICONERROR | MB_OK);
    // Re-attach the stroke collection to the context
    if (m_spIInkStrokes != NULL)
    {
        m_spIInkRecoContext->putref_Strokes(m_spIInkStrokes);
    }
    return 0;
}
m_bCoerceInputScope = !m_bCoerceInputScope;
//收集笔迹
if (m_spIInkStrokes != NULL)
{
    m_spIInkRecoContext->putref_Strokes(m_spIInkStrokes);
}
//更新识别结果
CComVariant vCustomData;// no custom data
m_spIInkRecoContext->BackgroundRecognizeWithAlternates(vCustomData);
//更新菜单
HMENU hMenu = GetMenu();
if (NULL != hMenu)
{
    HMENU hSubMenu = ::GetSubMenu(hMenu, mc_iSubmenuInputScopes);
    if (NULL != hSubMenu)
    {
        ::CheckMenuItem(hSubMenu, wID,
        MF_BYCOMMAND | (m_bCoerceInputScope ? MF_CHECKED : MF_UNCHECKED));
    }
}
return 0;
}
LRESULT CAdvRecoApp::OnInputScope(
    WORD /*wNotifyCode*/,
    WORD wID,
    HWND /*hWndCtl*/,
    BOOL& /*bHandled*/
    )
{
    UINT iInputScope = wID - ID_INPUTSCOPE_FIRST;
    if (wID == m_nCmdInputScope || iInputScope >= countof(gc_pwsInputScopes))
        return 0;
    if (m_spIInkRecoContext != NULL)
    {
        if (m_spIInkStrokes != NULL)
        {
            m_spIInkRecoContext->putref_Strokes(NULL);
        }
        CComBSTR bstrInputScope(gc_pwsInputScopes[iInputScope]);
```

```
            HRESULT hr = m_spIInkRecoContext->put_Factoid(bstrInputScope);
            if (m_spIInkStrokes != NULL)
            {
                m_spIInkRecoContext->putref_Strokes(m_spIInkStrokes);
            }
            if (SUCCEEDED(hr))
            {
                CComVariant vCustomData;
                m_spIInkRecoContext->BackgroundRecognizeWithAlternates(vCustomData);
            }
            else if (TPC_E_INVALID_PROPERTY == hr)
            {
                return 0;
            }
            else
            {
                return 0;
            }
        }
        UpdateMenuRadioItems(mc_iSubmenuInputScopes, wID, m_nCmdInputScope);
        m_nCmdInputScope = wID;
        UpdateStatusBar();
        return 0;
}
//识别向导
LRESULT CAdvRecoApp::OnGuide(
        WORD /*wNotifyCode*/,
        WORD wID,
        HWND /*hWndCtl*/,
        BOOL& /*bHandled*/
        )
{
        if (m_spIInkRecoGuide == NULL || wID == m_nCmdGuide)
            return 0;
        int cRows = 0, cColumns = 0;
        if (ID_GUIDE_LINES == wID || ID_GUIDE_BOXES == wID)
        {
            cRows = mc_iNumRowsCols;
            if (ID_GUIDE_BOXES == wID)
            {
                cColumns = mc_iNumRowsCols;
            }
        }
        if (SUCCEEDED(m_spIInkRecoGuide->put_Rows(cRows))
            && SUCCEEDED(m_spIInkRecoGuide->put_Columns(cColumns)))
        {
            HRESULT hr = S_OK;
            if (m_spIInkRecoContext != NULL)
            {
```

```
            if (m_spIInkStrokes != NULL)
            {
                m_spIInkRecoContext->putref_Strokes(NULL);
            }
            hr = m_spIInkRecoContext->putref_Guide(m_spIInkRecoGuide);
            if (m_spIInkStrokes != NULL)
            {
                m_spIInkRecoContext->putref_Strokes(m_spIInkStrokes);
            }
            CComVariant vCustomData;
            m_spIInkRecoContext->BackgroundRecognizeWithAlternates(vCustomData);
        }
        if (SUCCEEDED(hr))
        {
            m_wndInput.SetRowsCols(cRows, cColumns);
            UpdateMenuRadioItems(mc_iSubmenuGuides, wID, m_nCmdGuide);
            m_nCmdGuide = wID;
        }
    }
    return 0;
}
//变换模式
LRESULT CAdvRecoApp::OnMode(
    WORD /*wNotifyCode*/,
    WORD wID,
    HWND /*hWndCtl*/,
    BOOL& /*bHandled*/
    )
{
    if (wID == m_nCmdMode)
        return 0;
    InkCollectionMode icm;
    switch (wID)
    {
        default:
            return 0;
        case ID_MODE_INK:
            icm = ICM_InkOnly;
            break;
        case ID_MODE_INK_AND_GESTURES:
            icm = ICM_InkAndGesture;
            break;
        case ID_MODE_GESTURES:
            icm = ICM_GestureOnly;
            break;
    }
    if (m_spIInkCollector != NULL && SUCCEEDED(m_spIInkCollector->put_Enabled(VARIANT_FALSE)))
    {
        if (SUCCEEDED(m_spIInkCollector->put_CollectionMode(icm)))
```

```
            {
                UpdateMenuRadioItems(mc_iSubmenuModes, wID, m_nCmdMode);
                m_nCmdMode = wID;
                UpdateLayout();
            }
            if (FAILED(m_spIInkCollector->put_Enabled(VARIANT_TRUE)))
            {
                MessageBox(TEXT("模式设置失败!"),
                 gc_szAppName, MB_ICONERROR | MB_OK);
            }
        }
        return 0;
    }
    //识别
        WORD /*wNotifyCode*/,
        WORD /*wID*/,
        HWND /*hWndCtl*/,
        BOOL& /*bHandled*/
        )
    {
        if (m_spIInkRecoContext != NULL)
        {
            m_spIInkRecoContext->EndInkInput();
            CComPtr<IInkRecognitionResult> spIInkRecoResult;
            InkRecognitionStatus ink_reco_status = IRS_NoError;
            if (SUCCEEDED(m_spIInkRecoContext->Recognize(&ink_reco_status, &spIInkRecoResult)))
            {
                CComVariant vCustomData;
                OnRecognitionWithAlternates(spIInkRecoResult, vCustomData, ink_reco_status);
            }
            else
             {
                MessageBox(TEXT("识别失败."),
                gc_szAppName, MB_ICONERROR | MB_OK);
            }
            if (m_spIInkStrokes != NULL)
            {
                if (FAILED(m_spIInkRecoContext->putref_Strokes(m_spIInkStrokes)))
                {
                    MessageBox(TEXT("Failed to attach the stroke collection to the recognition context!"),
                    gc_szAppName, MB_ICONERROR | MB_OK);
                }
            }
        }
        return 0;
    }
```

因为代码量很大，所以这仅介绍核心源码。相关细节源码请见附属光盘的源代码，配有详细注释。

21.3.6 开发 Windows 7 的安全体验 CryptoAPI 加密

Cryptographic API（CryptoAPI）是微软在 Windows 操作系统中添加的密码编译机能。作为数据加密与解密功能的重要基础，CryptoAPI 支持同步、异步的私钥加密处理，以及操作系统中数字凭证的管理工作。

下面举例说明如何在 Windows 7 中利用 CryptoAPI 来对数据进行加密。对数据进行加密主要包括下面几步操作。

（1）打开一个支持加密的算法提供者，如 BCRYPT_DES_ALGORITHM。

（2）创建一个用于对数据进行加密的密钥。可以使用下面的函数来创建一个密钥。

- 对于非对称加密，使用 BCryptGenerateKeyPair 或 BCryptImportKeyPair()。
- 对于对称加密，使用 BCryptGenerateSymmetricKey 或 BCryptImportKey()。

（3）获取加密数据的大小。

（4）直接在数据所在空间进行加密或者将数据复制到一个独立的缓存中进行加密。

（5）调用 BcryptEncrypt()函数对数据进行加密，该函数将把加密后的数据写入到 pbOutput 参数内。

（6）根据需要保留加密后的数据。

（7）重复第（5）、（6）步，直至将所有数据都加密完毕：

```
#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0400
#endif
#include <stdio.h>
#include <windows.h>
#include <wincrypt.h>
#define MY_TYPE  (PKCS_7_ASN_ENCODING | X509_ASN_ENCODING)

//-------------------------------------------------------------------
// 签名证书名称，此证书必须为签名证书，并带有私钥
// 它必须具有 CERT_KEY_PROV_INFO_PROP_ID 或 CERT_KEY_CONTEXT_PROP_ID 属性
#define SIGNER_NAME  L"DUMMY_SIGNER_NAME"

//-------------------------------------------------------------------
// 证书库名称
#define CERT_STORE_NAME  L"MY"

//-------------------------------------------------------------------
//  函数申明
void HandleError(char *s);

void main(void)
{
  //-------------------------------------------------------------------
  // 变量申明与初始化
  HCERTSTORE hStoreHandle;                                          //系统证书库
  BYTE* pbMessage=(BYTE*)"CryptoAPI is a good way to handle security";   //预签名消息
  DWORD cbMessage = strlen((char*) pbMessage)+1;                    //预签名消息长度
  PCCERT_CONTEXT pSignerCert;                                       //签名证书
  CRYPT_SIGN_MESSAGE_PARA  SigParams;                               //签名参数结构
```

```
DWORD cbSignedMessageBlob;                                    //签名数据块长度
BYTE  *pbSignedMessageBlob;                                   //签名数据块
DWORD cbDecodedMessageBlob;                                   //解码数据块长度
BYTE  *pbDecodedMessageBlob;                                  //解码数据块
CRYPT_VERIFY_MESSAGE_PARA VerifyParams;                       //验证参数结构

//--------------------------------------------------------------------
// 创建消息队列及消息长度队列
const BYTE* MessageArray[] = {pbMessage};
DWORD MessageSizeArray[1];
MessageSizeArray[0] = cbMessage;
printf("开始处理. \n");
printf(" 将要被签名的消息是\n-> %s.\n",pbMessage);

//--------------------------------------------------------------------
// 打开证书库

if ( !( hStoreHandle = CertOpenStore(
  CERT_STORE_PROV_SYSTEM,
  0,
  NULL,
  CERT_SYSTEM_STORE_CURRENT_USER,CERT_STORE_NAME)))
{
   HandleError("MY 证书库不能被打开.");
}

//--------------------------------------------------------------------
//获取签名证书指针，此证书必须拥有签名私钥
if(pSignerCert = CertFindCertificateInStore(
  hStoreHandle,
  MY_TYPE,
  0,
  CERT_FIND_SUBJECT_STR,
  SIGNER_NAME,
  NULL))
{
  printf("签名者证书被找到.\n");
}
else
{
   HandleError( "签名证书未找到.");
}

//--------------------------------------------------------------------
// 初始化签名数据结构

SigParams.cbSize = sizeof(CRYPT_SIGN_MESSAGE_PARA);
SigParams.dwMsgEncodingType = MY_TYPE;
SigParams.pSigningCert = pSignerCert;
SigParams.HashAlgorithm.pszObjId = szOID_RSA_MD5;
SigParams.HashAlgorithm.Parameters.cbData = NULL;
```

```
SigParams.cMsgCert = 1;
SigParams.rgpMsgCert = &pSignerCert;
SigParams.cAuthAttr = 0;
SigParams.dwInnerContentType = 0;
SigParams.cMsgCrl = 0;
SigParams.cUnauthAttr = 0;
SigParams.dwFlags = 0;
SigParams.pvHashAuxInfo = NULL;
SigParams.rgAuthAttr = NULL;

//--------------------------------------------------------------------
// 两次调用 CryptSignMessage()获取签名消息
// 第一次获取签名数据 大小
if(CryptSignMessage(
   &SigParams,                  // 签名参数结构
   FALSE,                       // 是否拆开
   1,                           // 消息队列数目
   MessageArray,                // 消息队列
   MessageSizeArray,            // 消息大小队列
   NULL,                        // 签名后的数据
   &cbSignedMessageBlob))  // 签名数据长度
{
   printf("签名数据块的大小是 %d.\n",cbSignedMessageBlob);
}
else
{
   HandleError("获取签名数据块长度失败.");
}

//--------------------------------------------------------------------
// 分配内存

if(!(pbSignedMessageBlob =
  (BYTE*)malloc(cbSignedMessageBlob)))
{
   HandleError("签名时内存分配出错.");
}

//--------------------------------------------------------------------
// 签名操作，pbSignedMessageBlob 指向签名数据块
if(CryptSignMessage(
    &SigParams,                 // 签名参数结构
    FALSE,                      // 是否拆开
    1,                          // 消息队列数目
    MessageArray,               // 消息队列
    MessageSizeArray,           // 消息大小队列
    pbSignedMessageBlob,    // 签名后的数据
    &cbSignedMessageBlob))// 签名数据长度
{
   printf("此消息被签名成功. \n");
}
```

```
else
{
   HandleError("获取签名数据块出错.");
}



//--------------------------------------------------------------------
// 第二阶段：验证数字签名
// 一般的，这段程序由另一个用户在另一个应用程序中完成

//--------------------------------------------------------------------
// 初始化验证消息结构

VerifyParams.cbSize = sizeof(CRYPT_VERIFY_MESSAGE_PARA);
VerifyParams.dwMsgAndCertEncodingType = MY_TYPE;
VerifyParams.hCryptProv = 0;
VerifyParams.pfnGetSignerCertificate = NULL;
VerifyParams.pvGetArg = NULL;

//--------------------------------------------------------------------
// 两次调用 CryptVerifyMessageSignature()，验证、解码签名消息
// 第一次，获取解码消息长度

if(CryptVerifyMessageSignature(
   &VerifyParams,              // 验证参数结构
   0,                          // 签名序号
   pbSignedMessageBlob,        // 签名数据
   cbSignedMessageBlob,        // 签名数据长度
   NULL,                       // 解码后数据
   &cbDecodedMessageBlob, // 解码后数据长度
   NULL))                      // 签名证书指针
{
   printf(" %d 字节的空间分配给了缓冲区.\n",cbDecodedMessageBlob);
}
else
{
   printf("验证消息失败. \n");
}

//--------------------------------------------------------------------
// 分配空间

if(!(pbDecodedMessageBlob =
  (BYTE*)malloc(cbDecodedMessageBlob)))
{
   HandleError("为解码数据块分配空间出错.");
}

//--------------------------------------------------------------------
// 验证签名，获取解码消息
```

```
if(CryptVerifyMessageSignature(
    &VerifyParams,                    // 验证参数结构
    0,                                // 签名序号
    pbSignedMessageBlob,              // 签名数据
    cbSignedMessageBlob,              // 签名数据长度
    pbDecodedMessageBlob,             // 解码后数据
    &cbDecodedMessageBlob,            // 解码后数据长度
    NULL))                            // 签名证书指针
{
  printf("被验证的消息是 \n-> %s \n",
      pbDecodedMessageBlob);
}
else
{
  HandleError("验证消息失败. \n");
}

//--------------------------------------------------------------
// 释放空间、资源

if(pbSignedMessageBlob)
    free(pbSignedMessageBlob);
if(pbDecodedMessageBlob)
    free(pbDecodedMessageBlob);
if(pSignerCert)
    CertFreeCertificateContext(pSignerCert);
if(CertCloseStore(
     hStoreHandle,
     CERT_CLOSE_STORE_CHECK_FLAG))
{
  printf("证书库已关闭并且所有证书已经被释放. \n");
}
else
{
  printf("签名之后证书库关闭 -- \n"
      "但不是所有证书,证书撤销列表或证书信任列表都被释放了.");
}
}

// HandleError：错误处理函数，打印错误信息，并退出程序
void HandleError(char *s)
{
  printf("程序执行发生错误!\n");
  printf("%s\n",s);
  printf("错误代码为: %x.\n",GetLastError());
  printf("程序终止执行!\n");
  exit(1);
}
```