Visual C++ 6.0 程序设计从入门到精通

求是科技 王正军 编著

人民邮电出版社

图书在版编目（**CIP**）数据

Visual C++6.0 程序设计从入门到精通／求是科技编著．—北京：人民邮电出版社，2006.1

ISBN 7-115-14137-1

Ⅰ．V… Ⅱ．求… Ⅲ．C 语言—程序设计 Ⅳ．TP312

中国版本图书馆 CIP 数据核字（2005）第 130000 号

内 容 提 要

本书首先从 Visual C++ 6.0 的基础知识入手，详细介绍了 Visual C++6.0 的操作环境和应用程序的基本框架，然后按照由易到难、由浅入深地顺序介绍了用 Visual C++6.0 开发 Windows 程序的方法，包括菜单和加速键、对话框、常用控件、文本和字体类、图形图像、文件操作、多线程、动态连接库、网络编程和数据库开发等内容。书中在每一个对应的知识点后面，给出了一个或多个示例。

本书适合想按部就班地学习 Visual C++ 6.0 编程并很快应用到实践中的初学者阅读。

**Visual C++ 6.0** 程序设计从入门到精通

♦编 著 求是科技 王正军责任编辑 张立科

♦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 北京鸿佳印刷厂印刷新华书店总店北京发行所经销

♦开本：787×1092 1/16

印张：23.75 字数：579 千字 2006 年 1 月第 1 版 印数：1 – 6 000 册 2006 年 1 月北京第 1 次印刷

ISBN 7-115-14137-1/TP· 5056

定价：39.00 元（附光盘）

读者服务热线：( **010**) **67132692** 印装质量热线：( **010**) **67129223**

前 言

随着计算机技术的发展，可视编程技术占有了越来越重要的作用。Visual C++ 6.0 是微软公司的重要产品—Visual Studio 工具集的组成部分。它用来在 Windows（包括 Windows 95、

Windows 98、Windows NT、Windows 2000 等）环境下开发应用程序，是一种功能强大、行之有效的可视化编程工具，因此成为广大软件开发人员的首选。

本书为希望学习 Visual C++的初学者而编写，按照由浅入深、循序渐进的顺序介绍了 Visual C++编程的相关方法和技巧。在介绍过程中，还在相应的知识点后给出了 1 个或多个相关的示例，以帮助读者尽快将所学知识应用到实践中。

全书的安排如下。

第 1 章 主要介绍了 Visual C++ 6.0 的集成开发环境（IDE），让读者对 Visual C++先有一个初步印象和了解。

第 2 章 主要介绍在 AppWizard 工具中创建应用程序框架的步骤，然后介绍生成的工程文件中各部分所包含的类及相互间的关系，并着重阐述应用程序框架及整个应用程序的运行机理。

第 3 章 主要介绍了 Windows 界面设计的两个基本元素—菜单和加速键的设计和使用，同时还介绍了 Windows 消息驱动机制的概念和遵循这一机制的程序设计方法。

第 4 章 主要介绍了对话框的一般创建流程，模态和非模态的对话框、属性页对话框、各种通用对话框以及消息对话框的使用方法。

第 5 章 主要介绍了 MFC 编程中常用控件的设计方法，包括传统的 Windows 控件和新型的 Win32 控件。

第 6 章 主要介绍了 Windows GDI 输出文本和字体的方法。

第 7 章 主要介绍了 Windows 基本的绘图函数和主要的 GDI 对象。

第 8 章 主要通过示例介绍在 MFC 中对文件进行操作的方法，包括串行化、CFile 文件操作和文件对话框 3 部分内容。

第 9 章 主要介绍 Windows 操作系统下多线程的基本概念、创建管理线程的方法，以及线程的同步问题。

第 10 章 主要介绍了在 Visual C++平台下进行 Win32 动态链接库的设计和应用。

第 11 章 主要介绍了网络编程的基本知识及基本步骤，并且详细介绍了 WinSock 类。

第 12 章 主要介绍了数据库的基本概念和数据库开发的几种主要技术、Access 数据库的创建和维护以及 SQL 语句的基本知识。

在本书的编写过程中，借鉴了许多现行教材的宝贵经验，在此，谨向这些作者表示诚挚的感谢。由于时间仓促，加之编者水平有限，书中难免有谬误或不足之处，敬请广大读者批评指正。

编者

#### 目 录

第 1 章 Visual C++ 6.0 开发环境介绍..........................................................................................1

1.1 Visual C++ 6.0 概述....................................................................................................... 1

1.2 Visual C++ 6.0 的主要特点........................................................................................... 1

1.3 Visual C++ 6.0 主框架窗口........................................................................................... 3

1.3.1 工作区和客户区................................................................................................. 3

1.3.2 菜单栏................................................................................................................. 4

1.3.3 工具栏................................................................................................................. 4

1.3.4 输出窗口...............................................................................................................4

1.4 资源编辑器......................................................................................................................4

1.5 本章小结..........................................................................................................................5 第 2 章 应用程序基本框架......................................................................................................... 6

2.1 利用 MFC AppWizard 自动生成应用程序................................................................... 6

2.1.1 关于工程及工作区的概念 ................................................................................. 6

2.1.2 利用向导生成应用程序框架 ...............................................................................6

2.1.3 Visual C++工程文件的组成...............................................................................12

2.2 程序的应用框架和运行过程........................................................................................14

2.2.1 应用框架.............................................................................................................14

2.2.2 程序的运行过程.................................................................................................15

2.3 应用程序类及其主要成员函数....................................................................................18

2.3.1 InitInstance()函数................................................................................................19

2.3.2 Run()函数............................................................................................................21

2.3.3 ExitInstance()函数 ..............................................................................................22

2.3.4 OnIdle()函数 .......................................................................................................22

2.4 文档类和视图类............................................................................................................23

2.4.1 文档类.................................................................................................................23

2.4.2 视图类.................................................................................................................25

2.4.3 文档类与视图类的相互关系 .............................................................................26

2.4.4 单文档（SDI）与多文档（MDI）应用程序...................................................28

2.5 窗口类及其控件栏........................................................................................................30

2.5.1 主窗口类.............................................................................................................30

2.5.2 子窗口类.............................................................................................................31

2.5.3 工具栏.................................................................................................................33

2.5.4 状态栏.................................................................................................................36

2.6 应用程序框架中各类对象间的关系 ............................................................................40

2.7 本章小结........................................................................................................................41

第 3 章 菜单与加速键.................................................................................................................42

3.1 消息与事件....................................................................................................................42

3.1.1 事件驱动机制.....................................................................................................42

3.1.2 消息和消息映射.................................................................................................42

3.1.3 消息处理过程.....................................................................................................46

3.2 菜单和加速键的创建....................................................................................................47

3.2.1 菜单的创建.........................................................................................................47

3.2.2 加速键的创建.....................................................................................................49

3.3 添加菜单所需处理函数................................................................................................51

3.3.1 添加普通菜单的处理函数 .................................................................................52

3.3.2 添加弹出式菜单的处理函数 .............................................................................57

3.4 创建加速键....................................................................................................................59

3.5 本章小结........................................................................................................................60 第 4 章 对话框.............................................................................................................................61

4.1 对话框的创建................................................................................................................61

4.1.1 对话框的创建流程.............................................................................................61

4.1.2 创建对话框资源.................................................................................................62

4.1.3 创建对话框类.....................................................................................................67

4.2 模态对话框....................................................................................................................73

4.3 非模态对话框................................................................................................................74

4.4 属性页对话框................................................................................................................77

4.4.1 主要相关类介绍.................................................................................................78

4.4.2 向导对话框的创建步骤.....................................................................................82

4.4.3 一般属性页对话框的创建步骤 .........................................................................88

4.5 通用对话框....................................................................................................................88

4.5.1 打印设置对话框.................................................................................................89

4.5.2 页面设置对话框.................................................................................................90

4.5.3 查找对话框.........................................................................................................91

4.5.4 文件对话框.........................................................................................................94

4.5.5 字体对话框.........................................................................................................95

4.5.6 颜色对话框.........................................................................................................96

4.6 消息对话框....................................................................................................................97

4.7 本章小结........................................................................................................................98 第 5 章 常用控件.........................................................................................................................99

5.1 静态文本（Static text）控件........................................................................................99

5.1.1 传统控件通知消息...........................................................................................100

5.1.2 静态控件的使用...............................................................................................101

5.1.3 Cstatic 类的主要成员函数...............................................................................101

5.2 文本编辑（Edit Box）控件........................................................................................103

目录

5.2.1 文本编辑控件的使用.......................................................................................103

5.2.2 Cedit 类的主要成员函数..................................................................................104

5.3 滚动条（Scroll Bar）控件..........................................................................................108

5.3.1 滚动条控件的使用...........................................................................................108

5.3.2 CscrollBar 类的主要成员函数.........................................................................109

5.4 按钮（Button、Radio Box、Check Box、Group Box）控件................................... 111

5.4.1 按钮控件的使用............................................................................................... 111

5.4.2 Cbutton 类的主要成员函数 ............................................................................. 112

5.5 列表框（List Box）控件 ............................................................................................ 117

5.5.1 列表框控件的使用........................................................................................... 117

5.5.2 ClistBox 类的主要成员函数............................................................................ 119

5.6 组合框（Combo box）控件 .......................................................................................122

5.6.1 组合框控件的使用...........................................................................................122

5.6.2 CcomboBox 类的主要成员函数......................................................................124

5.7 图片（Picture）控件...................................................................................................125

5.8 列表（List Ctrl）控件.................................................................................................128

5.8.1 Win32 新型控件概述 .......................................................................................128

5.8.2 列表控件的使用...............................................................................................129

5.8.3 ClistCtrl 类的主要成员函数 ............................................................................133

5.9 树型（Tree Ctrl）控件................................................................................................135

5.9.1 树型控件的使用...............................................................................................135

5.9.2 CtreeCtrl 类的主要成员函数 ...........................................................................139

5.9.3 树型控件的通知消息.......................................................................................141

5.10 选项卡（Tab Ctrl）控件...........................................................................................141

5.10.1 选项卡控件的使用.........................................................................................141

5.10.2 选项卡控件通知消息.....................................................................................142

5.11 本章小结....................................................................................................................145 第 6 章 文本和字体类...............................................................................................................146

6.1 基础知识介绍..............................................................................................................146

6.2 CFont 类.......................................................................................................................146

6.2.1 概述...................................................................................................................146

6.2.2 主要成员函数...................................................................................................147

6.3 文本输出过程..............................................................................................................151

6.4 字体和文本输出实例..................................................................................................153

6.5 本章小结......................................................................................................................161 第 7 章 图形图像.......................................................................................................................162

7.1 绘图的基础知识..........................................................................................................162

7.2 屏幕绘图的主要函数..................................................................................................163

7.3 图形设备接口（GDI）...............................................................................................165

· 3·

7.3.1 设备环境类（CDC）.......................................................................................165

7.3.2 GDI 对象...........................................................................................................166

7.4 画笔和画刷..................................................................................................................166

7.4.1 画笔...................................................................................................................167

7.4.2 画刷...................................................................................................................171

7.5 本章小结......................................................................................................................173 第 8 章 文件操作.......................................................................................................................174

8.1 串行化基础知识..........................................................................................................174

8.1.1 文档类...............................................................................................................174

8.1.2 Serialize()函数 ..................................................................................................177

8.1.3 CArchive 类.......................................................................................................180

8.1.4 加入串行化代码...............................................................................................182

8.2 文件 CFile 类...............................................................................................................186

8.2.1 打开操作...........................................................................................................188

8.2.2 读写操作...........................................................................................................190

8.2.3 定位操作...........................................................................................................192

8.2.4 关闭操作...........................................................................................................194

8.2.5 异常操作...........................................................................................................195

8.2.6 管理操作...........................................................................................................197

8.3 文件对话框..................................................................................................................203

8.3.1 多选文件对话框...............................................................................................203

8.3.2 重载文件对话框...............................................................................................204

8.4 本章小结......................................................................................................................206 第 9 章 多线程...........................................................................................................................207

9.1 多线程的基本概念......................................................................................................207

9.2 两种重要的线程..........................................................................................................207

9.2.1 用户界面线程...................................................................................................207

9.2.2 工作者线程.......................................................................................................208

9.3 线程的操作..................................................................................................................208

9.3.1 线程的创建.......................................................................................................208

9.3.2 线程的终止....................................................................................................... 211

9.3.3 线程的管理和操作...........................................................................................212

9.3.4 线程间的通信...................................................................................................229

9.3.5 线程的同步.......................................................................................................234

9.3.6 示例：串口通信...............................................................................................243

9.4 本章小结......................................................................................................................250 第 10 章 动态链接库.................................................................................................................251

10.1 动态链接库的基础知识............................................................................................251

目录

10.2 DLL 的出入口函数 ...................................................................................................252

10.2.1 DllMain()函数.................................................................................................253

10.2.2 DLL 的导出函数 ............................................................................................255

10.3 两种链接 DLL 的方式 ..............................................................................................257

10.3.1 隐式链接.........................................................................................................257

10.3.2 显式链接.........................................................................................................258

10.4 开发 DLL ...................................................................................................................259

10.4.1 创建 Non-MFC DLL 动态链接库..................................................................259

10.4.2 MFC AppWizard[dll]方式生成常规/扩展 DLL.............................................261

10.4.3 导出函数调用约定.........................................................................................261

10.4.4 模块定义文件（DEF 文件）.........................................................................264

10.5 动态链接库中的资源................................................................................................264

10.6 界面汉化示例............................................................................................................267

10.7 使用钩子（Hook）函数 ...........................................................................................271

10.7.1 钩子函数的类型.............................................................................................271

10.7.2 使用钩子函数.................................................................................................273

10.7.3 鼠标钩子应用实例.........................................................................................274

10.7.4 键盘钩子应用实例.........................................................................................278

10.8 本章小结....................................................................................................................279 第 11 章 网络编程.....................................................................................................................280

11.1 网络编程的基础知识 ................................................................................................280

11.1.1 Socket 简介 .....................................................................................................280

11.1.2 Windows 套接字规范简介.............................................................................280

11.2 网络应用程序............................................................................................................281

11.2.1 网络应用程序的基本模型 .............................................................................281

11.2.2 建立套接字对象 .............................................................................................282

11.2.3 绑定地址 .........................................................................................................283

11.2.4 建立连接 .........................................................................................................283

11.2.5 监听 socket......................................................................................................284

11.2.6 数据传输 .........................................................................................................284

11.3 WinSock 类 ................................................................................................................285

11.3.1 CAsyncSocket 类.............................................................................................285

11.3.2 CSocket 类.......................................................................................................286

11.4 网络编程的实现 ........................................................................................................288

11.4.1 程序设计的基本任务 .....................................................................................288

11.4.2 服务器端程序设计 .........................................................................................288

11.4.3 客户端程序设计 .............................................................................................309

11.5 利用 MSComm 控件进行串口编程........................................................................322 11.5.1 MSComm 控件介绍........................................................................................322

· 5·

11.5.2 利用 MSComm 控件编程的基本流程...........................................................323

11.5.3 利用 MSComm 进行串口通信.......................................................................324

11.6 本章小结....................................................................................................................328 第 12 章 数据库开发.................................................................................................................329

12.1 概述............................................................................................................................329

12.1.1 数据库的基本概念.........................................................................................329

12.1.2 数据库相关技术介绍.....................................................................................330

12.1.3 数据库应用程序的开发步骤 .........................................................................331

12.1.4 选择数据库管理系统的原则 .........................................................................332

12.2 设计创建和维护 Access 数据库...............................................................................333

12.2.1 创建 Access 数据库........................................................................................334

12.2.2 创建数据表.....................................................................................................334

12.2.3 设计和修改数据表的结构 .............................................................................335

12.2.4 设置表之间的关系.........................................................................................336

12.2.5 编辑管理数据库记录.....................................................................................338

12.3 标准的 SQL 语句.......................................................................................................340

12.3.1 SQL 的基本知识.............................................................................................340

12.3.2 基本的 SQL 语句............................................................................................341

12.3.3 关系数据库的联合查询技术 .........................................................................342

12.4 ADO 基础知识 ..........................................................................................................343

12.4.1 ADO 的技术特点 ...........................................................................................344

12.4.2 ADO 的结构 ...................................................................................................344

12.5 ADO 的对象和集合 ..................................................................................................345

12.5.1 连接对象.........................................................................................................345

12.5.2 命令对象.........................................................................................................346

12.5.3 记录集对象.....................................................................................................346

12.5.4 ADO 的其他对象 ...........................................................................................347

12.5.5 集合.................................................................................................................349

12.6 数据库操作................................................................................................................350

12.6.1 动态链接库的引入和 OLE/COM 库的初始化 .............................................351

12.6.2 ADO 与数据库的连接....................................................................................351

12.6.3 获得和遍历记录集.........................................................................................352

12.6.4 对记录的操作.................................................................................................354

12.6.5 关闭记录集.....................................................................................................355

12.7 利用 ADO 查询并操作数据库实例..........................................................................355

12.8 本章小结....................................................................................................................370

第 1 章 Visual C++ 6.0 开发环境介绍

## 1.1 Visual C++ 6.0 概述

Visual C++是微软公司的重要产品之一—Visual Studio 工具集的重要组成部分。它用来在 Windows（包括 Windows 95、Windows 98、Windows NT、Windows 2000 等）环境下开发应用程序，是一种功能强大、行之有效的可视化编程工具。Visual C++以可视化技术为基础，以 C++语言为蓝本，以众多的集成工具为骨架，在计算机领域的诸多方面都发挥着重要的作用。其以实用的开发环境和集成的工具集让用户高效率地开发应用程序。

Visual C++提供了 MFC 类库，使用户可以很方便地开发自己想实现的功能。与 Turbo C 等工具相比，Visual C++完成目标（尤其对于图形界面的程序）所花费的时间要少得多。

Visual C++ 6.0 是微软公司推出的 Visual C++开发工具的 6.0 版本，发行于 Microsoft 的 Visual Studio 6.0 套装软件中。该工具在速度和代码量要求较高的场合中是首选工具。Visual C++ 6.0 在以前版本的基础上又增加了许多特性。除了支持 Internet 特性外，ADO 数据绑定、

ATL 复合控件、Auto Completion、编辑和继续特性、OLE-DB 提供者模块、延迟加载移入以及新的调试特性等都给 Visual C++6.0 增色不少。开发者可以充分利用 Visual C++ 6.0 的这些特性，构建出自己的 Windows 应用程序。

## 1.2 Visual C++ 6.0 的主要特点

Visual C++ 6.0 中的 MFC 类库和以前的版本中相比没有做太大的改进，只是增加了对微软公司的 IntelliMouse（智能鼠标）的支持，但 Visual C++集成开发环境中做了一些改进，增加了一些新特性，使其更易于使用。这些新特性包括以下几个方面。

1．自动化和宏功能

自动化（Automation）功能用于实现一些重复性过程和工作。宏记录功能可以根据用户的操作自动生成宏操作序列。Visual Studio 及其组件都可以看作对象来处理，这意味着可以进行自动化和诸如打开、编辑、关闭文档和调整窗口等操作。

2．可定制的工具栏和菜单

可以灵活地定制菜单和工具栏，使其更适合工作需要。如可以创建新的工具条和菜单（增加、删除菜单命令和工具条按钮等）。

3．调试器

可以直接运行和调试程序，还可以使用宏语言来自动操作调试器。

4．支持 Internet 连接

可以直接在集成开发环境（IDE）中查看网页页面，可以使用全新的 InfoViewer 或注册的 Web 浏览器查看 Web 上的页面。该特性可以让 Visual Studio 用户了解最新信息、获取更新的文档以及完成产品的升级和修正工作。

5．项目工作区和文件

一个新的便捷的项目系统应该可以允许一个工作区内包含多个不同的项目类型。比如说，可以创建一个包含 Visual C++工程和 J++ Applet 的工作区。在 Visual C++ 6.0 中，工作区文件以 dsw 为后缀名（以前为 mdp），项目文件以 dsp（以前为 mak）为后缀名。Build 文件现在分为两种：内部文件（dsp 后缀）和外部文件（mak 后缀）。在 IDE 中创建一个新的工程或从以前版本的工程进行转换时创建，内部 Build 文件与 NMAKE 外部编译工具不兼容，可以通过选择“Project|Export Makefile”菜单命令创建一个与 NMAKE 兼容的外部 Build 文件。

6．在工作区内可以包含多个并列的工程文件

要在当前工作区中增加一个工程，可以打开该工作区，然后选择“Project|Insert Project into Workspace”菜单命令。通过选择“Project|Set Active Project”菜单命令，可以设置当前活动工程，也就是执行 Build 操作时编译的那一个工程，这可能是令许多程序员喜爱的一个特性，因为这一特性使得可以在不同工程之间复制代码和资源。在工作区中还支持工程中鼠标直接拖放对象，这样用户可以直接从一个工程的资源文件中拖动一个对话框资源，然后放到另一个工程的资源文件中，而在以前版本中要完成类似的操作，必需先打开一个工程，然后打开另一个工程的资源文件，再进行资源复制操作。

7．增强的资源编辑器

可以在对话框中使用 WizardBar 将程序同可视化元素联系起来。在使用加速键、对话框、菜单、字符串时，如果需要对多个项作同一修改，可以选择所有要改的项目，然后选择

“View|Properties”菜单命令，在 Properties 对话框中一次完成多个项的值的更改。

技巧：要选择多个项，可以先用鼠标选中某一项，然后按住 Ctrl 键，再用鼠标选中其他项。或用鼠标左键拉框选择一个区域的多个项。要向从多个项中去掉一个选择，可以按 Ctrl 键，单击不选的那一项，从而去掉那一项的选择。

8．文本编辑器

Find in Files 命令支持两个独立的输出窗格，可以保存上一次搜索结果。

这些新特性可以更好地利用 Visual C++开发工具进行 Windows 应用程序开发。下面将带领读者进入 Visual C++ 6.0 的集成开发环境。

2·

第 1 章 Visual C++ 6.0 开发环境介绍

## 1.3 Visual C++ 6.0 主框架窗口

Visual C++ 6.0 是一种集成开发环境（Integrated Development Environment，IDE），它拥有友好的可视化界面。在 Visual C++中能够进行多种操作，包括建立、打开、浏览、编辑、保存、编译等，下面将一一介绍各窗口、菜单和工具栏。由于 Visual C++是一个可视化的开发工具，在介绍 Visual C++ 6.0 的各个组成部分之前，首先了解一下可视化编程的概念。

可视化技术是当前发展迅速并引人注目的技术之一，它的特点是把原来抽象的数字、表格、功能逻辑等用直观的图形、图象的形式表现出来。可视化编程是它的重要应用之一。所谓可视化编程，就是指在软件开发过程中，用直观的具有一定含义的图标按钮、图形化的对象取代原来手工的抽象的编辑、运行和浏览操作，软件开发过程表现为鼠标点击按钮和拖放图形化的对象以及指定对象的属性、行为过程。这种可视化的编程方法易学易用，而且大大提高了工作效率。

从开始菜单中启动 Microsoft Visual C++ 6.0，将进入 IDE 的主窗口。默认的 IDE 主窗口由标题栏、菜单栏、工具栏、工作区窗口、代码编辑客户区、输出窗口和状态栏等几个部分组成，下面将分别加以介绍。

### 1.3.1 工作区和客户区

工作区是 Visual C++的一个最重要的组成部分。程序员的大部分工作都在 IDE 中完成， IDE 使用项目工作区来组织项目、元素以及项目信息在屏幕上出现的方式。在一个项目工作区中，可以处理一个工程和它所包含的文件、一个工程的子工程、多个相互独立的工程和多个相互依赖的工程。

一个项目工作区可包含由不同的开发工具包生成的工程，如 Visual C++和 Visual J++。在桌面上，项目工作区以窗口方式组织项目、文件和项目设置。项目工作区窗口一般位于屏幕左侧，如图 1-1 中“工作区窗口”所示。项目工作区窗口底部有一组标签，用于从不同的角度（视图）查看项目中包含的工程文件信息。

图 1-1 Visual C++主窗口

每个项目视图都有一个相应的文件夹，包含了关于该项目的各种元素。展开该文件夹可以显示该视图方式下工作区的详细信息。项目工作区包含 3 种视图：

* FileView（文件视图）：显示所创建的工程。展开文件夹可以察看工程中所包含的文件。
* ClassView（类视图）：显示项目中定义的 C++类，展开文件夹显示工程中所定义的所有类，展开类可查看类的数据成员和成员函数以及全局变量、函数和类型定义。
* ResourceView（资源视图）：显示项目中所包含的资源文件。展开文件夹可显示所有的资源类型。

项目工作区底部有 3 个标签项，分别为类视图（ClassView）、资源视图（Resource View）和文件视图（FileView）。单击某个具体的标签项可以切换到对应的项目工作区视图中。

客户区位于整个主窗口的中部，如图 1-1 中的“代码编辑客户区”所示。客户区是程序员进行代码开发的场所，能直接提供给程序员编写代码、改写代码和调试代码的空间。

### 1.3.2 菜单栏

菜单栏是 Visual C++集成开发环境中的重要组成部分，几乎包含了常用的所有操作功能。菜单所涉及到的操作主要包括文件控制、文本编辑、视图查看、工程设置、编译调试和工具选项等功能。

### 1.3.3 工具栏

工具栏也是 IDE 的重要组成部分，主要列出了在代码开发过程中经常用到的一些功能。显示和隐藏工具栏有下面两种方法：

* 选择“Tools|Customize|Toolbars”菜单命令，将显示所有工具栏名称，根据需要进行选择。
* 在工具栏上单击鼠标右键，弹出所有工具栏名称菜单，根据需要进行选择。

### 1.3.4 输出窗口

输出窗口位于整个主窗口的下方，主要用于显示代码调试和运行中的相关信息，包括下面几个方面。

* 编译（Compile）信息：列出代码和资源编译详细过程及编译过程中的警告（Warning）和错误（Error）信息。
* 连接（Link）信息：列出工程对目标模块（Obj）连接过程中的警告（Warning）和错误（Error）信息。
* 调试（Debug）信息：在调试（Debug）状态下输出相关的调试信息（如 TRACE 宏输出调试信息等）。

## 1.4 资源编辑器

资源编辑器是 Visual C++ 6.0 开发环境中的一个重要可视化开发工具，绝大多数 Windows

4·

第 1 章 Visual C++ 6.0 开发环境介绍

应用程序的可视化界面元素都可以利用资源编辑器生成。Windows 资源包括加速键、图标、光标、位图、对话框、工具栏等，通过资源编辑器，可以打开资源文件（res 文件），并能对资源文件里面的资源进行编辑。在 Visual C++ 6.0 中，源代码和资源编辑器为了应用资源和用户自定义的其他对象，引入了一种与资源一一对应的符号，从而使资源能够以字符串的方式被引用。

资源符号是映像到某个整数值上 的一字符串，即标识符。每当创建一个新的资源时，系统就会为其提供一个默认的资源符号名称并赋予一个整数值。

## 1.5 本章小结

本章介绍了 Visual C++ 6.0 的集成开发环境（IDE），旨在让读者对 Visual C++先有一个初步印象和了解，至于更详细的使用方法，将在后面章节随着开发技术的具体讲解进行仔细地介绍。

# 第 2 章 应用程序基本框架

在熟悉了 Visual C++6.0 的开发环境后，就可以在该开发环境下进行程序设计了。Visual C++6.0 提供了应用程序框架生成向导（AppWizard），以方便用户自动生成一个简单的应用程序。当然，利用 AppWizard 只是得到一个程序框架而已，说它是框架，是因为程序的执行结果仅仅是显示一个包括菜单、工具栏和状态栏的空窗口。

本章中，首先介绍在 AppWizard 工具中创建应用程序框架的步骤，然后介绍生成的工程文件中各部分所包含的类及相互间的关系。

本章着重阐述应用程序框架及整个应用程序的运行机理，这部分内容可能读者暂时不能完全明白，不过在后面各章的学习中回过头来阅读相关部分便能更为深刻的领悟。

## 2.1 利用 MFC AppWizard 自动生成应用程序

### 2.1.1 关于工程及工作区的概念

在启动 Visual C++6.0 之后，将会进入它的集成开发环境 IDE，用户可以在这个集成开发环境中进行程序设计。现在的应用程序，尤其是 Windows 应用程序，一般由很多相关联的文件共同组成，包括程序源文件、头文件、资源文件等，所以在利用 Visual C++进行 Windows 程序开发的时候，有必要引入工程的概念。讲一个应用程序作为一个工程来处理，通过工程化的管理，使组成应用程序的所有文件形成一个有机的整体。

在 Visual C++6.0 开发环境中，工程又是置于工作区的管理之下，所以工作区又称为工程工作区，一个工作区可以管理多个工程，甚至是不同类型的工程。同一个工作区中的工程之间相互独立，但共用一个工作区的设置环境。在 Visual C++6.0 的集成环境中专门设置了一个工作区窗口，用以显示当前工作区的内容。在新建一个工程时，可以选择是为该工程新建一个工作区还是加入当前工作区。

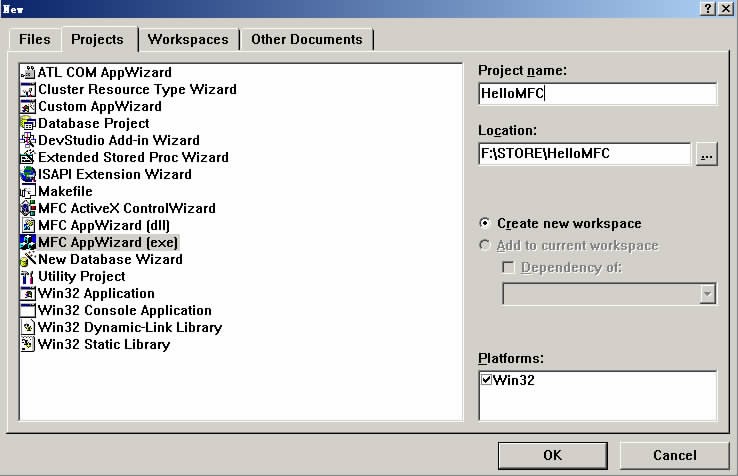
在利用应用程序向导（AppWizard）工具生成一个应用程序时，Visual C++会在用户选择的路径下以当前的工程名建立一个目录，然后将自动生成的文件置于该目录下。

### 2.1.2 利用向导生成应用程序框架

实例 2-1：利用向导生成的应用程序框架。源代码在光盘中“\02\实例 2-1\HelloMFC”目录下。

下面介绍使用应用程序向导工具生成 HelloMFC 应用程序的具体步骤。

1. 选择“File|New”菜单命令，将弹出“New”对话框，如图 2-1 所示。此时可以选择所要创建的文件类型，这里提供了 Files、Projects、Workspaces、Other Documents 4 种类型，每一个类型下面又包含许多具体的文件类型。为了生成应用程序框架，此时应选择 Projects 标签，标签下的列表框中列出的是各种不同的应用程序类型，这里选择“MFC Appwizard （exe）”选项，表示要创建的是一个使用 MFC 基本类的应用程序。然后在 Project name 编辑框中输入要建立工程的名称，这里把第一个程序命名为“HelloMFC”，在 Location 编辑框中显示了默认的工程存储路径，也可以进行修改，这里选用默认的路径。其他的设置都选用默认项，单击“OK”按钮。

 图 2-1 “New”对话框

1. 此时会自动弹出“MFC AppWizard-Step1”对话框，如图 2-2 所示。

在这个对话框中可以设置要建立应用程序的类型，有 3 种类型可以选择：

* Single document（单文档）；
* Multiple document（多文档）；
* Dialog based（基于对话框）。

这里选择 Single document 类型，生成一个单文档的应用程序框架，也就是说，应用程序运行时是一个单窗口的界面。

还可以在“What language would you like your resources in?”下拉列表框中选择界面语言，指定程序资源文件使用的语言类型，一般选用中文作为资源文件的语种。单击“Next”按钮进入下一步。

（3）此时会弹出“MFC AppWizard-Step 2 of 6”对话框，如图 2-3 所示。

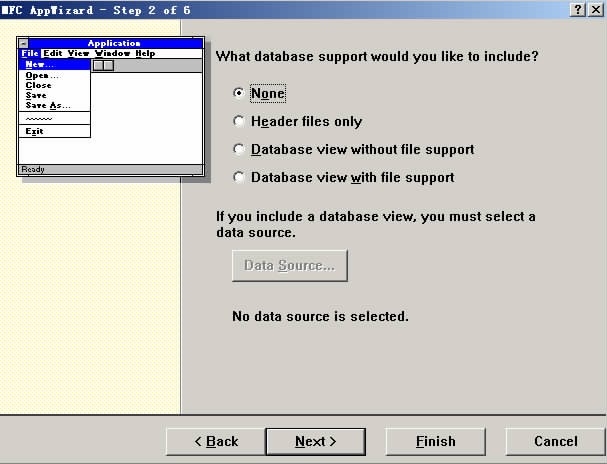


图 2-3 “MFC AppWizard-Step 2 of 6”对话框

该对话框用于指定数据库选项。MFC AppWizard 支持数据库并可以生成数据库应用程序所必需的代码。它可以是以下值之一。

* None：在应用程序中忽略所有的数据库支持。
* Header files only：包括定义基本数据库类的头文件，但不创建对应特定表的数据库类或视图类。
* Database view without file support：创建对应指定表的一个数据库类和一个视图类，不附加标准文件支持。
* Database view with file support：创建对应指定表的一个数据库类和一个视图类，并附加标准文件支持。

这里选择默认值 None，不使用数据库特性。单击 Next 按钮进入下一步。

1. 此时弹出“MFC AppWizard-Step 3 of 6”对话框，如图 2-4 所示。



图 2-4 “MFC AppWizard-Step 3 of 6”对话框

在这一步中可以向应用程序加入 OLE 支持，指定 OLE 选项的复合文档类型。因为本例中不用 OLE 特性，所以使用默认值 None。单击“Next”按钮，进入下一步。

1. 此时弹出“MFC AppWizard-Step 4 of 6”对话框，如图 2-5 所示。

在这一步中可以选择程序用户界面的功能，如工具栏、状态栏、打印和打印预览、上下文相关帮助等。此外在“How many files would you like on your recent file list?”框中还可以选择程序中文件菜单下面出现的最近打开过的文件个数。

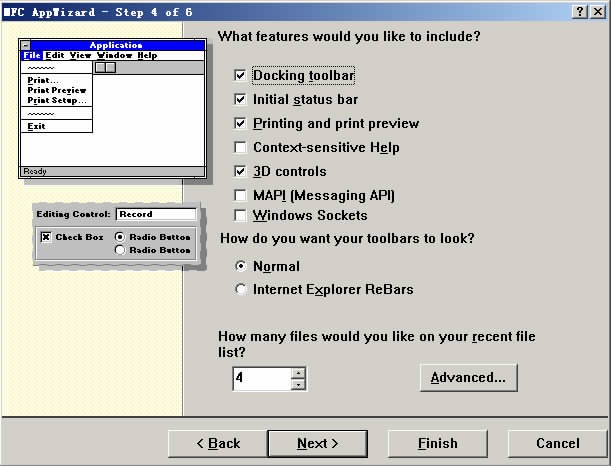


图 2-5 “MFC AppWizard-Step 4 of 6”对话框

单击“Advanced”按钮，将弹出“Advanced Options”对话框，其中有两个标签。在“Document Template Strings”标签下可以输入程序能处理的文件的扩展名以及更改程序窗口的标题等；在“Windows Styles”标签下可以调整程序窗口的风格，如有无系统菜单，有无最大化、最小化按钮等，如图 2-6、图 2-7 所示。

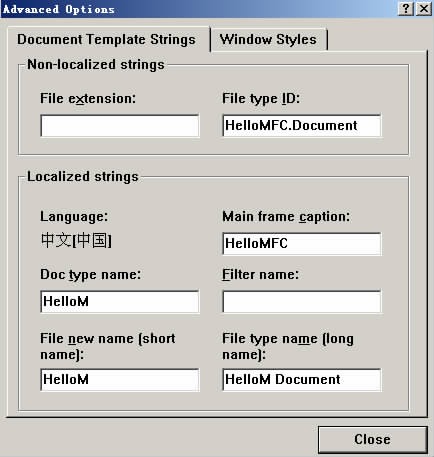


图 2-6 Advanced Options 对话框的 Document Template String 标签

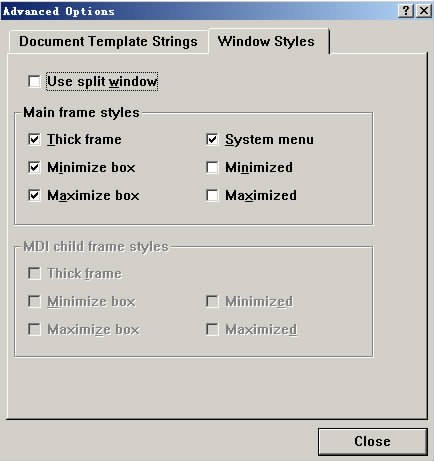


图 2-7 Advanced Options 对话框的 Window Styles 标签

本例中不用在“Advanced Options”对话框中输入内容，也不改变选项，单击“Close”

按钮返回到如图 2-5 所示的对话框。不改变默认选择，单击“Next”按钮，进入下一步。

（6）弹出“MFC AppWizard-Step 5 of 6”对话框，如图 2-8 所示。

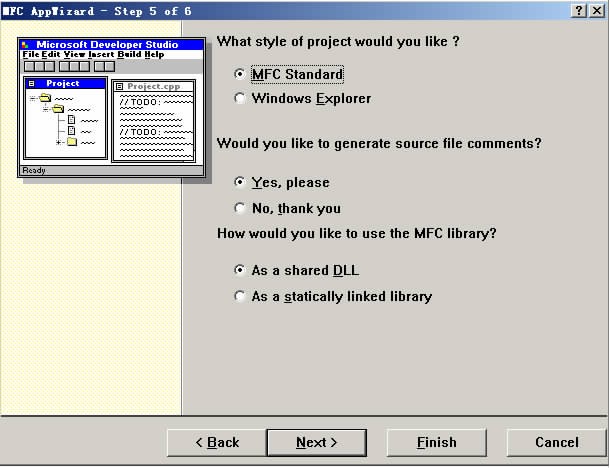


图 2-8 “MFC AppWizard-Step 5 of 6”对话框这里有 3 项内容需要设置。

“What style of project would you like? ”项用于设置工程的生成方式。

“Would you like to generate source file comments?”项用于设置是否为生成的源代码添加

必要的注释说明。由于注释说明有助于对应用程序源代码的学习和理解，因此一般选择“Yes，

Please”项。

“How would you like to use the MFC library?”项用于生成 MFC 类。选择“As a shared DLL”

项，所有 MFC 的类存放在动态链接库中（静态库方式），因此应用程序会小一些，但是发布该应用程序可执行文件时必需随同提供必要的动态链接库；选择“As a static linked library” 项时，应用程序所用到的 MFC 类都编译进了可执行文件之中，因此可执行文件比使用 DLL 方式的要大，但可以单独发行。一般的，对于小的应用程序可以采用静态库方式，对于大的

应用程序一般采用动态链接库方式。本例中使用默认设置，单击“Next”按钮，继续下一步。

1. 弹出 MFC AppWizard 最后一个对话框，如图 2-9 所示。

在对话框上部的列表框内，可以看到 MFC AppWizard 将要产生的 4 个类名称：一个视图类（CHelloMFCView）、一个应用类（CHelloMFCApp）、一个主框架窗口类（CMainFrame）和一个文档类（CHelloMFCDoc）。通过对话框下面的几个编辑框，可以修改默认的类名、类的头文件名和实现文件名。对于视图类，还可以修改其基类名称，默认的基类是 CView，其他可选的基类有 CEditView、CFormView、CHtmlView、CListView、CRichEditView、CScrollView 和 CTreeView。



图 2-9 “MFC AppWizard-Step 6 of 6”对话框

不改变这些名称，使用默认设置，单击“Finish”按钮。

1. 弹出“New Project Information”对话框，如图 2-10 所示。

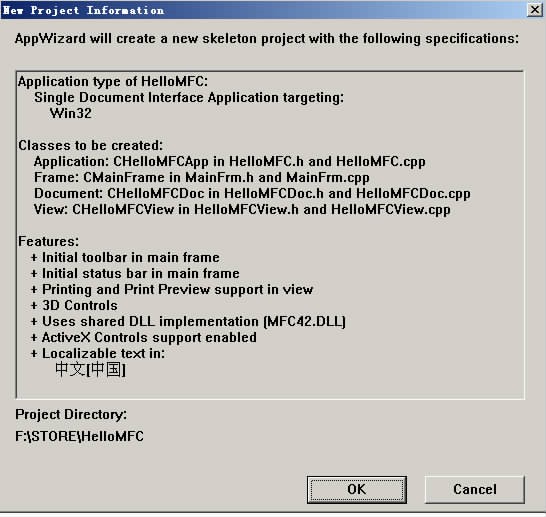


图 2-10 “New Project Information”对话框

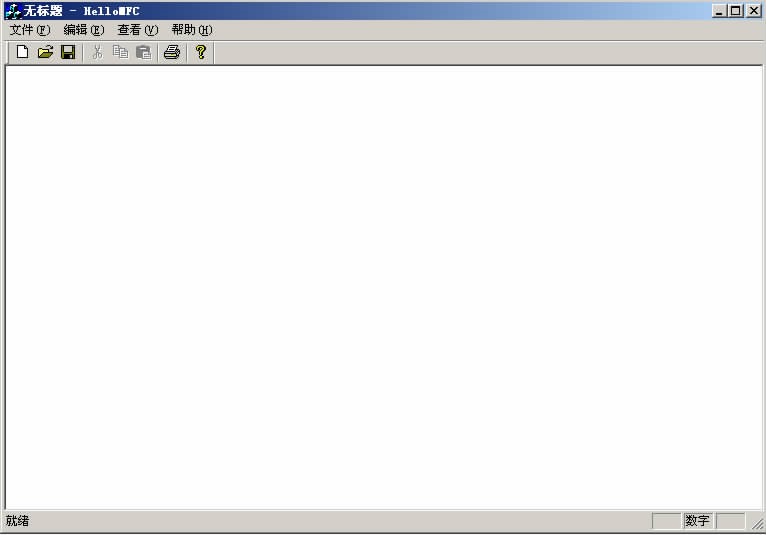
在对话框中列出了所建工程的一些基本信息，包括应用程序类型（单文档）、要创建的类及所在文件、应用程序的一些特性（是否支持工具条、状态栏）等。

确认信息后，单击“OK”按钮，则可创建一个工程，即一个应用程序的框架。

1. 生成程序后，在项目工作区中将自动打开 HelloMFC.dsw 项目工作区文件，窗口标题将显示为打开的项目名 HelloMFC。这样即生成了一个可执行的应用程序框架。

在菜单中依次选择“Build|Compile”菜单（或按快捷键 F7）编译该程序。编译完后选择

“Build|Execute HelloMFC.exe”菜单命令（或按快捷键 Ctrl+F5）运行该程序。也可以直接按组合键 Ctrl+F5，系统提示是否编译，选择“Yes”，IDE 将自动编译连接并运行 HelloMFC.exe 程序。结果如图 2-11 所示。

 图 2-11 MFC 自动生成的应用程序框架运行结果

### 2.1.3 Visual C++工程文件的组成

在用 AppWizard 创建的工程中包含了框架程序所有的类、全局变量的声明和定义。如果用 Windows 资源管理器查看 AppWizard 工具自动创建的工程目录，可以发现该目录下会生成十几个文件和两个目录，其中 res 目录用于存放工程的资源文件，Debug 目录用于存放编译、连接时产生的中间文件。

在 Visual C++中，一个工程文件可以产生两种不同版本的可执行程序：Debug 版本和 Release 版本。其中 Debug 版本中包含用于调试的信息和代码，而 Release 版本中由于不包含调试信息，所以不能进行调试，但其最终产生的文件更小。

根据选项设置的不同，AppWizard 所创建的文件会略有不同。标准的 AppWizard 文件包括下面 5 个部分：

* 工作区文件和项目文件；
* 应用程序源文件和头文件；
* 资源文件；
* 预编译头文件。

AppWizard 所创建的文件可以在指定的工程路径下找到，每一种文件都有自己明确的作用，下面将分别介绍这些文件的作用。

1．工作区文件和项目文件

工程区文件和项目文件主要是用于保存和更新工作区和项目信息，以及相关的类信息等，主要包括下面几个文件。

* HelloMFC.dsw：这是 MFC 自动生成的工作区文件，它包含当前工作区中的项目信息。
* HelloMFC.dsp：这是 MFC 生成的项目文件，它包含当前项目的设置、所包含的文件等信息。
* HelloMFC.clw：这个文件包含 ClassWizard 中用来编辑的现有类或增加新类的信息。

ClassWizard 还用这个文件来保存创建和编辑消息映射和对话框数据所需的信息，或是创建虚拟成员函数所需的信息。

2．应用程序源文件和头文件

根据应用程序的类型—单文档、多文档或基于对话框，AppWizard 将创建下述应用程序源文件和头文件中的某些文件，这些文件是工程的主体，负责主框架、文档、视图和相关操作的具体实现，主要包括下面几个方面。

* HelloMFC.h：这是应用程序的主头文件，它包含所有全局符号和用于包含其他头文件的#include 伪指令。
* HelloMFC.cpp：这个文件是应用程序的主源文件。它将创建 CHelloApp 类的一个对象

（从 CWinApp 派生），并覆盖 InitInstance 成员函数。

* MainFrm.cpp、MainFrm.h：这两个文件将从 CFrameWnd（SDI 应用程序）或

CMDIFrameWnd（MDI应用程序）派生CMainFrame类。如果在AppWizard的Application Options 页（AppWizard6 步中的第 4 步）中选择了对应的可选项的话，CMainFrame 类将处理工具条按钮和状态条的创建。MainFrame.cpp 文件中还包含 MFC 应用程序提供的默认工具条按钮的对象 ID—buttons 数组。

* HelloMFCDoc.cpp、HelloMFCDoc.h：这两个文件从 CDocument 类派生并实现名为

CHelloMFCDoc 的文档类，并含有用于初始化文档、串行化（保存和装入）文档和用于调试诊断的一些成员函数的框架。

* HelloMFCView.cpp、HelloMFCView.h：这些文件派生并实现名为 CHelloMFCView 的视类，用于显示和打印文档数据。CHelloMFCView 类是从 CView 或它的派生类派生出来的，含有绘制视图和用于调试诊断的一些成员函数框架。

3．资源文件预编译头文件

基于 MFC 的 Windows 应用程序一般都少不了应用程序资源的支持，AppWizard 会创建一些与资源相关的文件，同时，与 MFC 应用程序相关的预编译头文件也会被创建。

* HelloMFC.rc、Resource.h：这是项目的头文件及其资源文件。资源文件含有一般 MFC 应用程序的默认菜单定义和加速键表、字符串表，还指定了默认的 About 对话框和一个图标文件（res\HelloMFC.ico）。资源文件是标准的 MFC 类的资源，如果指定了支持工具条，它还将指定工具条位图文件（res\toolbar.bmp）。
* stdafx.cpp、stdafx.h：这两个文件用于建立一个预编译的头文件 HelloMFC.pch 和一个预定义的类型文件 stdafx.obj。由于 MFC 体系结构非常大，包含许多头文件，如果每次都编译的话比较费时，因此把常用的 MFC 头文件都放在 stdafx.h 中，如 afxwin.h、 afxext.h、afxdisp.h、afxcmn.h 等，然后让 stdafx.cpp 包含这个 stdafx.h 文件。这样，由于编译器可以识别哪些文件已经编译过，所以 stdafx.cpp 就只编译一次，并生成所谓的预编译头文件（因为它存放的是头文件编译后的信息）。如果以后在编程时不想让有些 MFC 头文件每次都被编译，也可以将它加入到 stdafx.h 中。采用预编译头文件可以加速编译过程。

## 2.2 程序的应用框架和运行过程

前面已经生成并运行了 HelloMFC 应用程序，并看到了运行结果，下面就开始介绍程序的运行过程。

### 2.2.1 应用框架

先请读者不要混淆应用框架和前面说到的应用程序框架。应用程序框架是用于生成一般的应用程序所必须的各种面向对象的软件组件的集合，简单地说就是 MFC 程序运行所必须的部分代码的集合。应用框架则是 MFC 程序基于 Windows 系统运行的底层机制，如从

WinMain()函数入口，进入窗口消息循环，结束窗口，退出 WinMain()主程序。

和任何一个 C 语言程序一样，对于一个 Windows 应用程序，其结构至少包括一个 WinMain()函数和一个窗口函数，WinMain()函数名不能更改，窗口函数可以自己取名。

WinMain()函数是主程序，是程序的入口点，而窗口函数则是 Windows 程序的工作核心。

有些读者喜欢自己定义两个初始化子函数 InitApplication()和 InitInstance()，然后在 WinMain()函数中调用。在 InitApplication()函数中进行注册窗口类的工作，而在 InitInstance() 函数中进行创建并显示程序主窗口的工作。

Windows 应用程序在本质上是面向对象的，程序提供给用户界面的可视对象在程序的内部一般也是一个对象，通过事件驱动模式触发对象的相应方法，从而实现用户对可视对象的操作。程序的运行过程就是用户的外部操作不断产生事件，这些事件又被相应的对象处理的过程。Windows 程序工作原理如图 2-12 所示。

在 Visual C++6.0 中当然可以只调用 Windows 的 API 函数进行 SDK 程序设计，不过这不在本书的讨论范围之内，本书主要讲述 MFC 编程。用 MFC 编程，虽然看起来生成的应用程序与 SDK 下的不同，但其运行机理是一致的。因为同是 Windows 应用程序，其运行机制也是一样的。和 SDK 程序不同的是，MFC 程序中很多工作都由框架来完成了。

为了对 MFC 方式的程序的框架有一个总体的认识，这里设计了一个表示程序中的主要类之间的关系的图表，如图 2-13 所示。

这个图表表示了使用 MFC 方式的应用程序的 4 个主要类之间的关系。从中可以看出， CHelloMFCApp 类主要的作用是用来处理消息的，它统一管理程序收到的所有的消息，然后把消息分配到相应的对象。CMainFrame 是 CHelloMFCView 的父窗口，也就是说视窗 View 显示在主框窗 MainFrame 的客户区中。类 CHelloMFCView 的作用是显示数据，而数据的来源是类 CHelloMFCDoc。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，文档与视窗分离带来的好处就是一个文档可以同时具有多个视窗，每个视窗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好处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处理的目的。关于文档和视图的具体关系，在 2.4 节中有详细介绍。

### 2.2.2 程序的运行过程

实例 2-2：介绍应用程序运行过程的实例。源代码在光盘中“\02\实例 2-2\hello”目录下。

为了将 MFC 应用程序的窗口概念阐述清楚，这里生成一个更为简单的 hello 例程，该例程只有一个 hello.cpp 文件。生成该工程的方法很简单，通过建立一个 Win32 Application 的空工程，然后加入 hello.cpp 即可。代码清单如下（添加行号是为了讨论方便，实际程序不能有行号）：

1 //hello.cpp

|  |
| --- |
| 1. #include <afxwin.h> 2. // Declare the application class 3. class CHelloApp : public CWinApp 4. { 5. public: 6. virtual BOOL InitInstance(); 7. }; 8. // Create an instance of the application class 9. CHelloApp HelloApp; 10. // Declare the main window class 11. class CHelloWindow : public CFrameWnd 12. { 13. CStatic\* cs; 15 public: 14. CHelloWindow(); 15. }; 16. // The InitInstance function is called each 19 // time the application first executes. 17. BOOL CHelloApp::InitInstance() 18. { 19. m\_pMainWnd = new CHelloWindow(); 20. m\_pMainWnd->ShowWindow(m\_nCmdShow); 21. m\_pMainWnd->UpdateWindow(); 22. return TRUE; 23. } 24. // The constructor for the window class 25. CHelloWindow::CHelloWindow() 26. { 27. // Create the window itself 28. Create(NULL, 29. "Hello World!", 30. WS\_OVERLAPPEDWINDOW, 31. CRect(0,0,200,200)); 32. // Create a static label 33. cs = new CStatic(); 34. cs->Create("hello world", 35. WS\_CHILD|WS\_VISIBLE|SS\_CENTER, 36. CRect(50,80,150,100), 37. this); 38. } |

该程序由 6 小部分组成，每一部分都起到很重要的作用，下面分别加以介绍。

* 首先，该程序第 2 行包含了头文件 afxwin.h，该头文件包含有 MFC 中所使用的所有的类型、类、函数和变量，也包含了其他头文件，如 Windows API 库等。
* 第 3 至 8 行从 MFC 说明的标准应用程序类 CWinApp 继承出了新的应用程序类

CHelloApp，该新类是为了要重载 CWinApp 中的 InitInstance()成员函数。InitInstance() 函数是一个应用程序开始执行时要调用的可重载函数。

* 在第 10 行中，类 CHelloApp 定义了一个单独的全局对象 HelloApp，这个对象代表了一个应用程序，它的大部分行为都被基类 CWinApp 决定了。当应用程序被装入内存并开始执行时，全局变量的建立会执行 CWinApp 类的默认构造函数。该构造函数会自动调用 18~26 行定义的 InitInstance()函数。
* 在第 11~17 行中，CHelloWindow 类是从 MFC 中的 CFrameWnd 类继承来的，类 Chello-

Window 是作为应用程序在屏幕上的窗口。建立新的类以便实现构造函数、析构函数和数据成员。

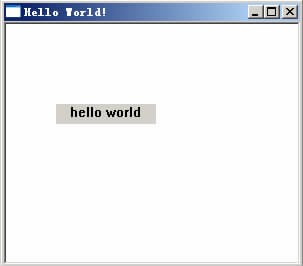
* 第 18~26 行实现了 InitInstance()函数，该函数产生一个 CHelloWindow 类的实例，因此会执行第 27~41 行中类的构造函数，它也会把新窗口放到屏幕上。
* 通过查看 MFC 帮助文件中的 CFrameWnd 定义部分发现，CFrameWnd 中并没有包含这两个成员函数（ShowWindow 和 UpdateWindow）。CFrameWnd 是从 CWnd 类继承来的，接下来可以查看 CWnd 类，会发现它包含有 200 多个不同的成员函数，其中包括

ShowWindow 和 UpdateWindow。第 22 行是初始化窗口，它为调用 new 函数分配内存。在调用 new 函数分配一个实例的时候，程序在执行时会调用 CHelloWindow 的构造函数，该构造函数在每次类的实例被分配时都要调用。在窗口构造函数的内部，窗口必

须通过调用 CFrameWnd 中的 Create()成员函数来建立。

* 第 27~41 行实现了窗口的构造函数，该构造函数实际是建立了窗口，然后在其中建立一个静态文本控制。

hello 例程的运行结果如图 2-14 所示。

 图 2-14 hello 例程的运行结果

要注意的是，在该程序中没有 main 或 WinMain 函数，也没有事件循环，所有这些操作都隐藏在主应用程序类 CWinApp 中，并且不必为它的事件处理而操心，它都是自动执行、在 MFC 中不可见的。关于 CWinApp 类的主要成员函数以及对 WinMain 和事件循环的处理，将在下一节作详细讨论。

从 hello 例程可以看出，每个 MFC 应用程序都包含如下 6 个部分。

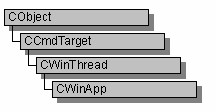
* 从 CWinApp 派生一个新类作为该应用程序的程序类，这个类应该重载 InitInstance() 函数。
* 将该应用程序类声明为全局变量，CWinApp 的构造函数将自动调用 InitInstance()函数。
* 从 CFrameWnd（或 CMDIFrameWnd 或 CMINIFrameWnd）派生一个类作为屏幕上的主窗口。
* 创建该窗口的 InitInstacne()函数。
* 创建该窗口的构造函数。
* 创建该窗口的析构函数。

MFC 程序的运行和启动过程和 hello 例程本质上是一致的，依次执行下面几个步骤，这些步骤在 MFC 程序运行的时候都是自动完成的，编程被屏蔽在外面，这里通过一个简单的 hello 例程看到了这些步骤，从而对整个启动和运行过程有一个透彻的理解。

* 应用程序对象产生（theApp），配置内存，设定初值。
* 调用 AfxWinMain()函数，执行 AfxWinInit()函数，后者又调用 AfxInitThread()函数，把消息队列尽量加大到 96。
* AfxWinMain()函数执行 InitApplication()函数。
* AfxWinMain()函数执行 InitInstance()函数，它是 CWinApp 的虚函数，必须重载并修改它。
* InitInstance()函数创建了一个 CMyFrameWnd 对象。
* CFrameWnd 构造函数调用 Create()函数，产生主窗口，在生成窗口前，Create()函数注册有关窗口类。
* 回到 InitInstance()函数中继续执行 ShowWindow()函数显示窗口。
* 执行 UpdateWindow()函数，发出 WM\_PAINT 消息。
* 回到 AfxWinMain()函数，执行 Run()函数，进入消息循环。

## 2.3 应用程序类及其主要成员函数

每个应用程序从类 CWinApp 派生出自己的应用程序类，并定义一个全局的对象。该应用程序类包含了 Windows 下 MFC 应用程序的初始化、运行和结束过程。基于框架建立的应用程序必须有一个（且只能有一个）从 CWinApp 派生的类的对象。在创建的 HelloMFC 程序中，从 CWinApp 中派生出了 CHelloMFCApp 类，定义了一个全局对象 theApp，CHelloMFCApp 类在 HelloMFC.cpp 文件中定义。CWinApp 类在 MFC 体系中的层次如图 2-15 所示。

 图 2-15 CWinApp 类层次表

CWinApp 所包含的数据成员主要用于存放和控制应用程序本身的信息，主要数据成员如下。

* m\_pszAppName：指定了应用程序名称。
* m\_hInstance：标识了应用程序的当前实例。
* m\_lpCmdLine：指示一个以 NULL 结尾的字符串，指向应用程序的命令行。
* m\_nCmdShow：指示最初如何显示窗口。
* m\_pszExeName：应用程序的模块名称。
* m\_pszHelpFilePath：应用程序的帮助文件路径。
* m\_pszProfileName：应用程序对应的.ini 文件名。

在应用程序中经常需要用到一些全局函数来得到应用程序及实例的一些信息，主要的和应用类相关的全局函数如下：

* AfxGetApp：返回指向应用程序的 CWinApp 对象的指针。
* AfxGetInstanceHandle：返回应用程序实例的句柄。
* AfxGetResourceHandle：返回应用程序资源的句柄。。
* AfxGetAppName：返回应用程序的名称。

以上全局函数可以在应用程序代码中的任何位置调用，如调用 AfxGetApp 函数可获得指向 CWinApp 类实体的指针。一旦拥有了这个指针，就可以轻松访问 CWinApp 类的任何公有成员变量。

CWinApp 的主要成员函数控制着应用程序的初始化、运行和中止，这在应用程序设计中起着关键的作用，下面就对 CWinApp 的主要成员函数加以介绍。

### 2.3.1 InitInstance()函数

InitInstance()函数用于初始化实例。Windows 下对同一个应用程序可以运行多个实例。例如，如果已经在运行资源管理器，用户还可以再一次或多次启动资源管理器，这样，系统中就有两个或多个资源管理器在运行，这些运行着的资源管理器就是资源管理器应用程序的多个运行实例。当每次启动某个应用程序的一个实例时，WinMain()函数都要调用 InitInstance() 函数。

InitInstance()函数主要完成设置注册数据库、载入标准设置（最近打开文件列表等）、注册文档模板等工作，其中注册文档模板过程中隐含地创建了主窗口。接着，处理命令行参数、显示窗口，然后返回、进入消息循环。在 CHelloMFCApp 应用程序类中重载了 CWinApp 的成员函数 InitInstance，HelloMFC 程序中调用 InitInstance()函数的代码如下：

BOOL CHelloMFCApp::InitInstance()

{

AfxEnableControlContainer();

// Standard initialization

// If you are not using these features and wish to reduce the size

// of your final executable, you should remove from the following

|  |
| --- |
| // the specific initialization routines you do not need.    #ifdef \_AFXDLL  Enable3dControls(); // Call this when using MFC in a shared DLL    #else    Enable3dControlsStatic(); // Call this when linking to MFC statically #endif    // Change the registry key under which our settings are stored.  // TODO: You should modify this string to be something appropriate // such as the name of your company or organization.  SetRegistryKey(\_T("Local AppWizard-Generated Applications"));    LoadStdProfileSettings(); // Load standard INI file options (including MRU)    // Register the application’s document templates. Document templates  // serve as the connection between documents, frame windows and views.    CSingleDocTemplate\* pDocTemplate; pDocTemplate = new CSingleDocTemplate(  IDR\_MAINFRAME,  RUNTIME\_CLASS(CHelloMFCDoc),  RUNTIME\_CLASS(CMainFrame), // main SDI frame window  RUNTIME\_CLASS(CHelloMFCView));  AddDocTemplate(pDocTemplate);  // Parse command line for standard shell commands, DDE, file open  CCommandLineInfo cmdInfo;  ParseCommandLine(cmdInfo);  // Dispatch commands specified on the command line if (!ProcessShellCommand(cmdInfo))  return FALSE;    // The one and only window has been initialized, so show and update it.  m\_pMainWnd->ShowWindow(SW\_SHOW); m\_pMainWnd->UpdateWindow(); return TRUE;  } |

从上面的代码可以看出，AppWizard 自动生成的 InitInstance()函数主要完成下述功能。

* 从.ini 文件或 Windows 注册表中获取一些标准的文件选项，包括取得最近使用的文件名称，以便在应用程序的文件菜单中列出，供用户快速打开最近编辑的文件。
* 该函数的中心工作是构造文档模板类（单文档或多文档模板类）的对象，指明了该文档模板对象对应的文档类、框架窗口类和视图类。这使得以后可以据此构造文档类、边框窗口类和视图类的对象及创建新窗口。
* 如果是 MDI 应用程序，则构造主边框窗口类的对象并创建主边框窗口。
* 另一重要工作是调用 ProcessShellCommand()函数进行程序窗口启动方式的分析处理，如果在其中程序的命令行中提供了文档文件名，则启动程序时，打开该文件；如果没有提供文档文件名，则新建一个空文件。

此外，InitInstance()函数还可执行其他一些功能，这里就不一一说明了。Windows 允许在同一时刻运行程序的几份拷贝。在概念上，应用程序的初始化可以被分为两个部分：一次性的应用程序初始化工作和示例的初始化工作，前者在应用程序第一次运行时完成，后者在每次运行程序时都会执行。框架中 WinMain()函数重载 InitInstance()函数以初始化在 Windows 下运行的应用程序的每个新实例。通常，重载 InitInstance 以构造主窗口对象并设置

CWinThread::m\_pMainWnd 数据成员，使其指向这个窗口。

技巧：快速显示当前函数相关的信息在编写代码或查看代码的时候，常常需要查看某个 MFC 类库中定义的函数的相关信息。可以通过快捷键 F1 来实现，具体做法是在 IDE 客户区将鼠标光标放在要查看函数名称处并停止鼠标光标移动，按 F1 键，这时将启动 MSDN Libray 并切换到要查看的函数信息页面上。

### 2.3.2 Run()函数

应用程序的大部分时间都是在应用程序类的 Run()成员函数中进行处理。WinMain()函数在初始化应用程序实例后，就调用 Run()函数来处理消息循环。

图 2-16 Run 成员函数的消息循环

Run()成员函数不断执行消息循环，检查消息队列中有没有消息。如果有消息，Run()函数将其派遣，交由框架去处理，然后返回继续消息循环。如果没有消息，Run()函数将调用 OnIdle()函数来执行用户或框架在空闲时要执行的工作，如用户接口更新消息处理等。如果既没有消息要处理，也没有空闲时的处理工作要做，则应用程序将一直等待，直到有事件发生。应用程序结束时，Run()函数将调用 ExitInstance()函数使应用程序退出。消息循环的流程图如图 2-16 所示。

Run()函数很少被重载，但是也可以重载它以提供特殊的功能，HelloMFC 中的默认 Run() 函数定义如下：

int CHelloMFCApp::Run()

{

return CWinApp::Run();

}

### 2.3.3 ExitInstance()函数

ExitInstance()函数是在用户退出应用程序的运行实例时由 Run()函数调用的。

框架在 Run()成员函数内部调用这个函数以退出应用程序的实例。此函数只能在 Run() 成员函数内部调用。这个函数的默认实现将框架的选项写入应用程序的.ini 文件。重载这个函数可以在应用程序退出的时候执行一些清除操作。

HelloMFC 中默认的 ExitInstance()函数定义如下：

int CHelloMFCApp::ExitInstance()

{

return CWinApp::ExitInstance();

}

返回值表示应用程序的退出码，0 表示没有错误，大于 0 的值表示有错误。这个值被用作 WinMain 的返回值。

### 2.3.4 OnIdle()函数

OnIdle()函数是在应用程序的消息队列中没有消息时由 Run()函数调用的。

如果要执行空闲时处理，则必须重载这个成员函数。当应用程序的消息队列为空时，

OnIdle 就在默认的消息循环中被调用，可以重载这个函数来调用后台空闲处理任务。

HelloMFC 中默认的 OnIdle()函数定义如下：

BOOL CHelloMFCApp::OnIdle(LONG lCount)

{

return CWinApp::OnIdle(lCount);

}

其中 lCount 参数是一个计数值，当应用程序的消息队列为空，OnIdle()函数被调用时，该计数值就增加 1。每当一条新消息被处理时，该计数值就被复位为 0，可以使用 lCount 参数来确定应用程序不处理消息时空闲时间的相对长度。

OnIdle()函数应返回 0 以表明不需要更多的空闲处理时间。当消息队列为空时，OnIdle() 每被调用一次 lCount 参数就增加 1，而每处理一条新消息 lCount 就被复位为 0，可以根据这个计数值调用不同的空闲处理例程。

空闲循环处理的过程如下。

* 如果 MFC 类库中的消息循环检查消息队列并发现没有未被处理的消息，就为应用程序对象调用 OnIdle()函数，并将 lCount 参数设为 0。
* OnIdle()函数执行一些处理，然后返回一个非零值，表示它还需要被调用，以进行进一步处理。
* 消息循环再次检查消息队列，如果没有未处理的消息，则再次调用 OnIdle()函数，增加 lCount 参数。
* OnIdle()函数结束所有的空闲任务并返回 0，这告诉消息循环停止调用 OnIdle()函数直到在消息队列中接收到下一条消息为止，在那时，空闲循环将重新启动，而参数被设为 0。

由于只有在 OnIdle 返回之后应用程序才能处理用户输入，因此在 OnIdle 中不应执行较长的任务。

注意：OnIdle 除可实现更新用户接口对象（如菜单项和工具条等）外，还实现了内部数据结构的清理。因此，如果重载了 OnIdle 函数，必须用重载版本中使用的 lCount 值来调用 CWinApp::OnIdle。首先调用所有基类的空闲处理（即直到基类的 OnIdle 返回 0），如果需要在基类处理完成之前进行一些工作，则应重复基类的实现以在工作期间选择一个合适的

lCount 值。

## 2.4 文档类和视图类

MFC AppWizard 自动生成的应用程序默认采用文档-视图结构，因为一般应用程序都要对某种文档进行处理（文档并不一定是文件，但通常可以理解为文件），而文档又通过视图与用户打交道（或称交互），文档的内容通过视图窗口显示给用户，用户在视图中对文档所作的修改由视图通知文档对象，视图实际上充当了一个中介者的角色。

下面将对文档视图结构中文档类和视图类及它们的相互关系作相应介绍。

### 2.4.1 文档类

文档类（CDocument）在 MFC 类库中的层次结构如图 2-17 所示。

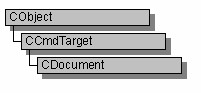


图 2-17 CDocument 在 MFC 类库中的位置

不管是 SDI 应用程序还是 MDI 应用程序，文档类都是从 CDocument 类派生出来的，

App Wizard 自动生成的文档类为 HelloMFCDoc，其定义在文件 HelloMFCDoc.h 中，实现在

HelloMFCDoc.cpp 中。若要使用 AppWizard 提供的文档类，必须执行下列操作。

* 为每个文档类型从 Cdocument 中派生一类。
* 添加成员变量以存储每个文档的数据。
* 在文档类中重写 CDocument 的 Serialize 成员函数，Serialize 用于从磁盘读取文档的数据和将文档数据写入磁盘。
* 可能还需要重写其他 CDocument 成员函数，如经常需要重写 OnNewDocument 和 OnOpenDocument 以初始化文档的数据成员、重写 DeleteContents 以动态销毁分配的数据。

在 HelloMFCDoc 类中，主要重载了基类的两个函数：OnNewDocument()和 Serialize()。其中 OnNewDocument()函数用于文档对象的初始化，Seralize()函数用于实现序列化。虽然定义了这两个函数，但只是提供了一个框架，具体的函数内容需要用户自己加入。处理文档序列化的 OnSerialize()函数的默认代码如下：

void CHelloMFCDoc::Serialize(CArchive& ar)

{

//存储

if (ar.IsStoring())

{

// TODO: add storing code here

} //读取

else

{

// TODO: add loading code here

}

}

技巧：当文档被修改时，在其标题上加上“\*”作标志。

文档类还可处理由菜单项、工具栏按钮或快捷键生成的某些命令。默认情况下，

CDocument 使用序列化方式处理“File|Save”和“File|Save as”菜单命令，文档可以有消息映射，但与视图不同，文档无法处理标准 Windows 消息，而只能处理 WM\_COMMAND 命令消息或命令。

实例 2-3：技巧演示程序。源代码在光盘中“\02\实例 2-3\EditApp”目录下。

如重载 CEditAppDoc 类的 SetModifiedFlag 函数，代码如下：

void CEditAppDoc::SetModifiedFlag(BOOL bModified /\* = TRUE \*/)

{

CString strTitle = GetTitle();

CString strDirtyFlag = " \*"; // note space before the ’\*’

|  |
| --- |
| // so we don’t break Save As dialog    if (!IsModified() && bModified)  {  SetTitle(strTitle + strDirtyFlag);  }  else if ( IsModified() && !bModified )  {  int nTitleLength = strTitle.GetLength(); int nDirtyLength = strDirtyFlag.GetLength();  SetTitle( strTitle.Left(nTitleLength - nDirtyLength) );  }  UpdateFrameCounts();  CDocument::SetModifiedFlag(bModified);    } |

同时映射 CEditAppView 的 EN\_CHANGE 消息如下：

|  |
| --- |
| void CEditAppView::OnChange()  {  // TODO: If this is a RICHEDIT control, the control will not  // send this notification unless you override the CEditView::OnInitDialog()  // function and call CRichEditCtrl().SetEventMask()  // with the ENM\_CHANGE flag ORed into the mask.  GetDocument()->SetModifiedFlag(TRUE);  // TODO: Add your control notification handler code here } |

运行结果如图 2-18 所示。

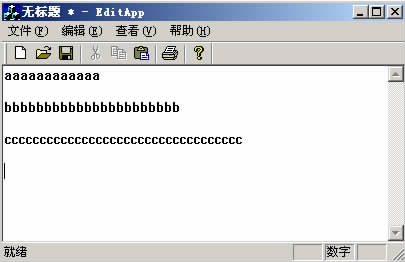


图 2-18 更新文档标题示例

### 2.4.2 视图类

AppWizard 产生的视图类名为 HelloMFCView，其定义在 HelloMFCView.h 中，实现在文

件 HelloMFCView.cpp 中。视图的职责是以图形方式将文档数据显示给用户，以及接受用户输入并将用户输入解释为对文档的操作。编写视图类的任务通常包括：

* 编写视图类的 OnDraw 成员函数来呈现文档数据。
* 将适当的 Windows 消息和用户界面对象（如菜单项）连接到视图类中的消息处理成员函数。用以实现处理程序解释用户输入。

另外，可能需要在派生的视图类中重写其他 CView 成员函数，具体说来，可能需要重写 OnInitialUpdate()函数为视图执行特殊的初始化，并重写 OnUpdate()函数以执行在视图重绘自身前所需的任何特殊处理。对于多页文档，可能还必须重写 OnPreparePrinting()函数，用要打印的页数和其他信息初始化打印对话框。

应用程序中几乎所有的绘图都在视图的 OnDraw 成员函数中发生，必须在视图类中重载

OnDraw()函数（鼠标绘图是个特例，必须处理鼠标动作消息）：

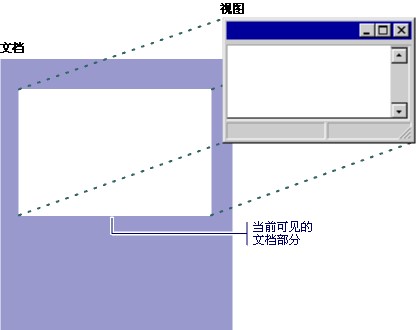
* 通过调用文档成员函数获取数据；
* 通过调用框架传递给 OnDraw()函数的设备上下文对象的成员函数来显示数据。

当文档的数据以某种方式更改后，必须重绘视图以反映该更改，一般情况下，文档数据的更改在用户通过视图对文档进行更改时发生。在此情况下，视图调用文档的 UpdateAllViews()成员函数，通知同一文档上的所有视图对自身进行更新，UpdateAllViews() 函数调用每个视图的 OnUpdate()成员函数，使视图的整个工作区无效。可以重写 OnUpdate，仅使工作区中映射到文档修改部分的那些区域无效。

### 2.4.3 文档类与视图类的相互关系

在文档-视图结构里，文档是一个应用程序数据基本元素的集合，它是构成应用程序所使用的数据单元；另外它还提供了管理和维护数据的手段。文档是一种数据源，数据源有很多种，最常见的是磁盘文件，也可以来自串行口、网络或摄像机输入信号等，文档对象负责来自所有数据源的数据的管理。

视图是数据的显示窗口，为用户提供了文档的可视的数据显示，它把文档的部分或全部内容在窗口中显示出来。视图还为用户提供了一个与文档中的数据交互的界面，它把用户的输入转化为对文档中数据的操作。一个文档可以有多个不同的视图。比如，在 Excel 电子表格中，可以将数据以表格方式显示，也可以将数据以图表方式显示。一个视图既可以输出到窗口中，也可以输出到打印机上。图 2-19 显示了文档-视图结构中的文档与视图的关系。

 图 2-19 文档与视图的关系

MFC 的文档视图结构机制把数据同它的显示以及用户对数据的操作分离开来，所有对数据的修改由文档对象来完成，视图调用这个对象的方法来访问和更新数据。

CDocument 类的 UpdateAllViews()成员函数和 CView 类的 OnUpdate()成员函数传递描述被修改的文档部分的信息。上述机制可限制视图必须重绘的区域。当视图变得无效时， Windows 将 WM\_PAINT 消息发送给它，该视图的 OnPaint()函数处理函数通过创建 CPaintDC 类的设备上下文对象来响应该消息并调用视图的 OnDraw()成员函数，通常不必编写重写的 OnPaint()处理成员函数。设备上下文是包含设备（例如显示器或打印机）绘图属性信息的 Windows 数据结构，所有的绘图调用都通过设备上下文对象进行。对于屏幕上的绘图， CPaintDC 对象传递给 OnDraw()函数。对于打印机上的绘图，为当前打印机设置的 CDC 对象传递给 OnDraw()函数。视图中的绘图代码首先检索指向文档的指针，然后通过设备上下文进行绘图调用，下面简单的 OnDraw()函数示例阐释了该过程：

|  |
| --- |
| void CMyView::OnDraw( CDC\* pDC )  {  CMyDoc\* pDoc = GetDocument();  CString s = pDoc->GetData(); // Returns a CString  CRect rect;  GetClientRect( &rect );  pDC->SetTextAlign( TA\_BASELINE | TA\_CENTER );  pDC->TextOut( rect.right / 2, rect.bottom / 2, s, s.GetLength() );  } |

在此例中，将 GetData()函数定义为派生文档类的成员，打印从文档获取的任何字符串，并使字符串在视图中居中。如果 OnDraw 调用用于屏幕绘图，则传递给 pDC 的 CDC 对象是一个 CPaintDC（它的构造函数已经调用 BeginPaint）对象，对绘图函数的调用通过设备上下文指针进行。

文档的大小经常比它的视图可以显示的大小要大，之所以会发生这种情况，是因为文档的数据增加了或用户缩小了为视图设置的窗口，在此情况下，视图必须支持滚动。任何视图都可以在其 OnHScroll 和 OnVScroll 成员函数中处理滚动条消息。可以通过自定义代码完成全部工作，在这些函数中实现滚动条消息处理；或者可以利用 CScrollView 类处理滚动。

CScrollView 在默认情况下执行下列操作：

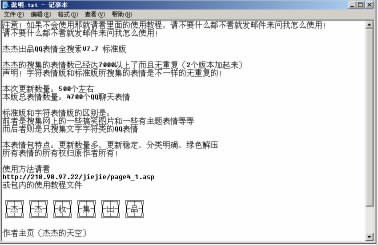
* 管理窗口和视图区大小以及映射模式；
* 响应滚动条消息进行自动滚动。

可以指定“页”的滚动量（当用户单击滚动轴时）和指定“行”的滚动量（当用户单击滚动箭头时），设计这些值以符合视图的自然特性。例如，可能需要以 1 像素的增量滚动图形视图，而在文本文档中则要使用基于行高的增量来滚动。

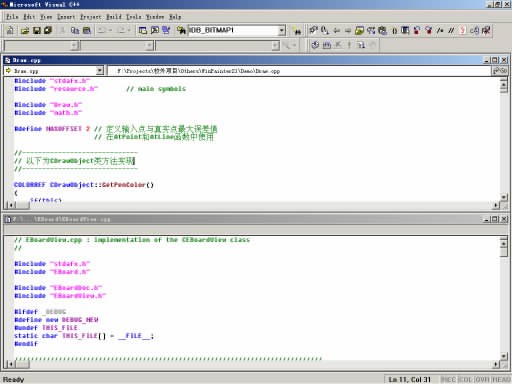
对单文档（SDI）界面来讲，一个应用程序中只用一个框架窗口，对多文档（MDI）界面来讲可能有多个 MDI 子窗口。每一个视图都是一个子窗口，在单文档界面中父窗口即是框架窗口，在多文档界面中父窗口为 MDI 子窗口。一个多文档应用中可以包含多个文档模板，一个模板定义了一个文档和一个或多个视图之间的对应关系。同一个文档可以属于多个模板，但一个模板中只允许定义一个文档，同样一个视图也可以属于多个文档模板。

### 2.4.4 单文档（SDI）与多文档（MDI）应用程序

在单文档界面程序中，同一时刻只能操作一个文档。如 Windows 2000 下的记事本（Notepad）程序就是这样的例子，如图 2-20 所示。在单文档界面中打开文档时，程序会自动关闭当前打开的活动文档，若文档修改后尚未保存，会提示是否保存所做的修改。由于一次只能打开一个窗口，因此不像 Visual C++集成开发环境那样需要一个窗口菜单。单文档应用程序一般都提供一个文件（File）菜单，在该菜单下有一组命令，用于新建文档（New）、打开已有文档（Open）、保存或重命名存盘文档等。这类程序相对比较简单，常见的应用程序为终端仿真程序或一些工具程序。

 图 2-20 单文档程序（Notepad）

在多文档界面应用程序中允许同时操作多个文档。Visual C++集成开发环境就是这样的例子，其界面如图 2-21 所示。在界面中可以同时打开多个文件（同时也就为每个文件打开一个窗口），并通过切换活动窗口激活相应的文档进行编辑。多文档应用程序也提供一个文件（File）菜单，用于新建、打开、保存文档。与单文档应用程序不同的是，它往往还提供一个关闭（Close）菜单项，用于关闭当前打开的文档；一个窗口菜单用于管理所有打开的子窗口，包括对子窗口的新建、关闭等，关闭一个窗口时，窗口内的文档也被自动关闭。

 图 2-21 多文档程序（Visual C++ IDE）

可以通过 AppWizard 很方便地生成 SDI 和 MDI 类型的应用程序，这两种结构在代码设计方面也有如下的区别。

* SDI 程序仅仅基于主框架窗口（CMainFrame），而 MDI 程序同时基于主框架窗口

（CMainFrame）和子框架窗口（CChildFrame）。

* 对于管理文档、框架和窗口的文档模板类，两种结构也有本质的差别。SDI 程序基于的文档模板类是 CSingleDocTemplate，只能管理单个文档类型；而 MDI 程序基于的文档模板类是 CMultiDocTemplate，能同时创建多种文档类型。
* 只能向 SDI 应用程序类中加入一种文档模板，而可以向 MDI 应用程序类中加入多个

CMultiDocTemplate 对象。

SDI 程序和 MDI 程序的文档视图结构都是由文档模板管理的，下面针对 MDI 应用程序的情况介绍文档模板在编程的应用。

每个应用程序类（CWinApp 的派生类）中都保留并维护了一份所有文档模板的指针列表，这是一个链表结构。应用程序为所要支持的每个文档类型动态分配一个 CMultiDocTemplate 对象，并在应用程序类的 CWinApp::InitInstance 成员函数中将每个 CMultiDocTemplate 对象传递给 CWinApp::AddDocTemplate，该函数将一个文档模板加入到应用程序可用文档模板的列表中。应用程序可以用 CWinApp::GetFirstDocTemplatePostion 获得应用程序注册的第一个文档模板的位置，利用该值来调用 CWinApp::GetNextDocTemplate 函数，获得第一个

CDocTemplate 对象指针。

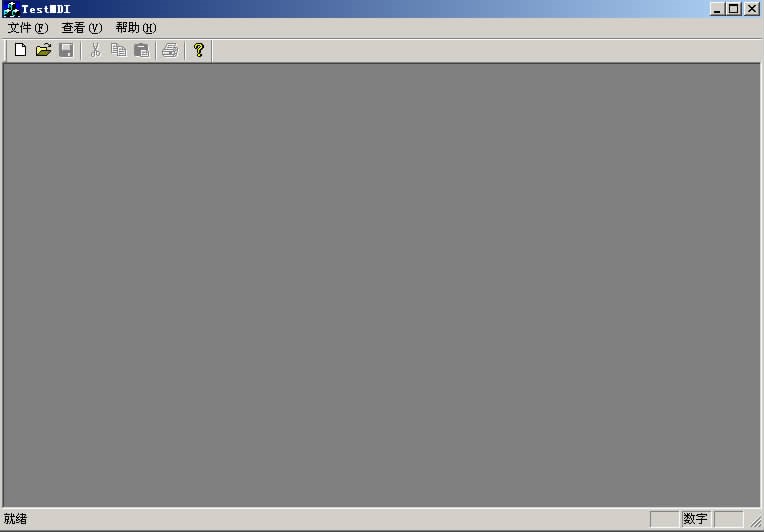
一个文档模板可以有多个文档，每个文档模板都保留并维护了一个所有对应文档的指针列表。应用程序可以用 CDocTemplate::GetFirstDocPosition 函数获得与文档模板相关的文档集合中第一个文档的位置，并用 POSITION 值作为 CDocTemplate::GetNextDoc 的参数来重复遍历与模板相关的文档列表。

在文档中可以调用 CDocument::GetDocTemplate 获得指向该文档模板的指针。一个文档可以有多个视图，每一个文档都保留并维护一个所有相关视图的列表。CDocument::AddView 将一个视图连接到文档上，将该视图加入到文档相联系的视图的列表中，并将视图的文档指针指向该文档。当有选择菜单命令“File|New”、“File|Open”、“Windows|New”或 “Window|Split”菜单命令而将一个新创建的视图的对象连接到文档上时，MFC 会自动调用该函数，框架通过文档视图结构将文档和视联系起来。当然，也可以根据特定需要手动调用该函数。

|  |
| --- |
| 技巧：使 MDI 启动时不建立一个新文件使用 AppWizard 工具生成的 MDI 程序启动时会默认创建一个空白文件，这有时候会带来麻烦（也许在程序启动时不允许创建新文件）。为了使 MDI 启动时不建立一个新文件，需要修改 InitInstance()函数如下：  CCommandLineInfo cmdInfo;  if（cmdInfo.m\_nShellCommand == CCommandLineInfo::FileNew）  {  cmdInfo.m\_nShellCommand = CCommandLineInfo::FileNothing;  }  ParseCommandLine（cmdInfo）; |

此时程序启动后显示如图 2-22 所示。

实例 2-4：技巧演示实例。源代码在光盘中“\02\实例 2-4\TestMDI”目录下。

 图 2-22 MDI 启动时不生成新文件示例

## 2.5 窗口类及其控件栏

框架窗口为应用程序界面提供结构框架，它是应用程序的主窗口，负责管理其包容的窗口，一个应用程序的最顶层的框架窗口为应用程序启动时创建的第一个窗口。MFC 提供了 CFrameWnd、CMDIFrameWnd、CMDIChildWnd 和 Cdialog 类，分别用于支持单文档窗口、多文档窗口和对话框。同时，在基于文档视图的应用框架中也经常用到相关的控件栏，如工具栏、状态栏等，这些也是应用程序不可缺少的部分。下面将对这些方面进行分别介绍。

### 2.5.1 主窗口类

主窗口类 CMainFrame 可以应用在 SDI 或 MDI 结构中。在 SDI 框架结构中，CMainFrame 派生于类 CFrameWnd，形成单个文档及其视图的边框。框架窗口既是应用程序的主框架窗口，也是当前文档对应的视图的边框。在 MDI 框架结构中， CMainFrame 派生于类 CMDIFrameWnd，它是所有 MDI 文档窗口的容器，并与它们共享菜单条，MDI 框架窗口是出现在桌面中的顶层窗口。

窗口的创建分为两步：第一步是用 new 运算符创建一个窗口对象，但是此时只是初始化窗口的数据成员，并没有真正创建窗口（这一点与一般的对象有所不同）。

//第一步：创建一个 C++对象，其中 CMainFrame 是从 CFrameWnd 派生的对象

CMainFrame\* pMyFrame=new CMainFrame();//用 new 操作符创建窗口对象

或

CMainFrame MyFrame;//定义一个窗口对象，自动调用其构造函数

第二步是创建窗口。使 CFrameWnd 的 Create 成员函数创建窗口，并将其句柄保存在 C

公共数据成员 m\_hWnd 中。

//第二步：创建窗口

pMyFrame->Create(NULL,“My Frame Window”);

或

MyFrame.Create(NULL,“My Frame Window”);

### 2.5.2 子窗口类

子窗口类 CChildFrame 主要用在多文档（MDI）结构中，在单文档中只存在主窗口

（CMainFrame），不存在子窗口的概念。CChildFrame 派生于类 CMDIChildWnd，用于在 MDI 主框架窗口中显示打开的各个文档，每个文档及其视图都有一个 MDI 子框架窗口，子框架窗口包含在 MDI 主框架窗口中。子框架窗口看起来类似一般的框架边框窗口，但它是包含在主框架窗口中，而不是位于桌面上，并且被主窗口所裁剪。而且 MDI 子窗口没有自己的菜单，它与主 MDI 框架窗口共享菜单。框架自动更改 MDI 框架的菜单，用于代表当前活动的 MDI 子窗口。为了为应用创建有用的 MDI 子窗口，需要从 CMDIChildWnd 派生类，为派生类增添成员变量，从而为应用存储指定的数据。实现消息处理的成员函数，并在派生类中实现消息映射，以便于指出当消息传递给窗口后，将按照什么方式处理消息。构造 MDI 子窗口主要有下面 3 种途径：

* 使用 Create()函数直接构造；
* 使用 LoadFrame()函数直接构造；
* 通过文档模板直接构造。

|  |
| --- |
| 技巧：如何改变 MDI 窗口框架的颜色?  MDI 框架的客户区被另一个窗口的框架所覆盖，为了改变客户区的背景色，需要重画这个客户窗口。为了做到这点，要处理消息 WM\_ERASEBKND 产生一个新类，如从 CWnd 继承一个新类，称为 CMDIClient，具体可见下面的代码片断。 |

实例 2-5：改变 MDI 窗口框架的颜色。源代码在光盘中“\02\实例 2-5\TestMDI”目录下。

#include "MDIClient.h"

class CMainFrame : public CMDIFrameWnd

{

...

protected:

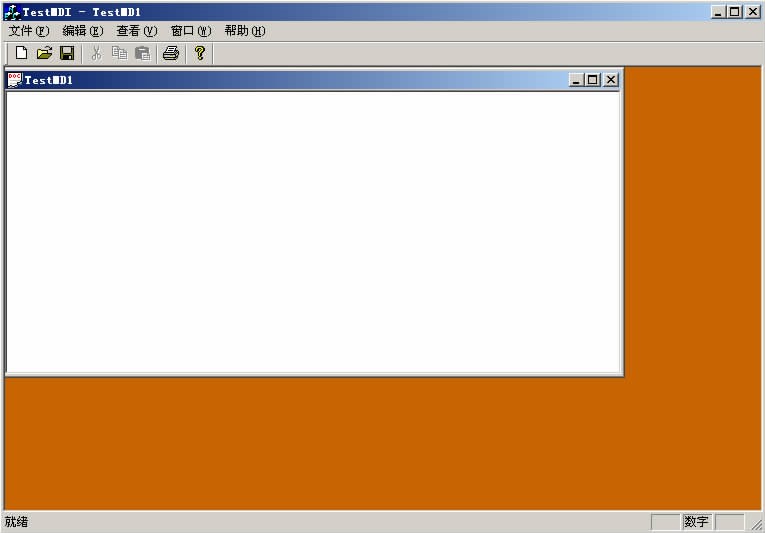
CMDIClient m\_wndMDIClient;

}

在 CMainFrame 中重载 CMDIFrameWnd::OnCreateClient

|  |
| --- |
| BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext\* pContext)  {  if ( CMDIFrameWnd::OnCreateClient(lpcs, pContext) )  {  m\_wndMDIClient.SubclassWindow(m\_hWndMDIClient); return TRUE;  } else  return FALSE;  }  //然后就可以加入对消息 WM\_ERASEBKGND 的处理函数。  BOOL CMDIClient::OnEraseBkgnd(CDC\* pDC)  {  CBrush brush(RGB(200,100,0));  CRect rect;  GetClientRect(&rect); pDC->FillRect(rect,&brush);  return TRUE;  } |

程序运行结果如图 2-23 所示，其中框架颜色变成了橘红色，有关该技巧的源代码请参见附带光盘。

 图 2-23 改变 MDI 窗口的框架颜色

### 2.5.3 工具栏

工具栏由一系列的按钮和间隔（Seperator）组成。一般功能相关的按钮组成一组，各组按钮之间以一定的间隔分开。通常，工具栏作为主框架窗口中的一个子窗口存在。MFC 中使用 CToolBar 类来封装工具条控件的各种操作，可以在资源编辑器的 ToolBar 上单击鼠标右键，选择 Insert ToolBar 项，选中一个工具栏后，鼠标左键双击该工具栏就可以编辑，如图 2-24 所示。

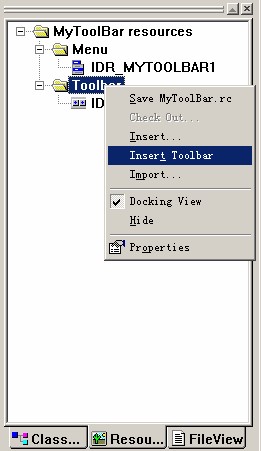


图 2-24 添加一个新的 ToolBar 资源

工具栏上所有按钮的图形在资源中实际上是以一个位图文件存在的。在利用 AppWizard 创建的工程中，自动创建的工具栏位图文件名为 Res\ToolBar.bmp。在位图中每个按钮占用一个宽为 16 像素、高为 15 像素的区域。可以用图形工具条及颜色盒绘制按钮的外表，它的属性包括 ID、长、宽及鼠标指向它时的说明。一般在 CMainFrame 中定义 CToolBar，其实现在

CMainFrame 的 OnCreate 函数中完成，在该函数中用于创建工具栏的具体特征。

在工具栏中嵌入编辑框等控件，可以通过下面的方法：

* 定义一个编辑控件对象，不能是局部变量，否则会被释放掉；
* 调用 Create，惟一注意的是要在工具条中调用。

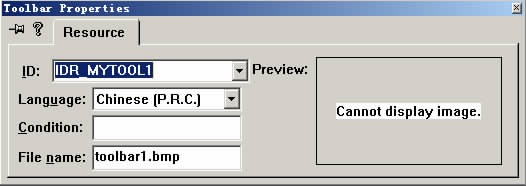
CRect rect;

m\_wndToolBar.GetItemRect(1,&rect);

edit.Create(WS\_CHILD|WS\_VISIBLE,rect,&m\_wndToolBar,ID\_EDIT);

实例 2-6：添加工具栏程序。源代码在光盘中“\02\实例 2-6\MyToolBar”目录下。

下面是一个 MyToolbar 的示例程序，集中演示了 ToolBar 在框架窗口中的应用。新建一个空的 Win32 Application 工程，然后加入下面的 MyToolBar.cpp 源文件，按照前面给出的步骤加入新的工具栏资源，这里工具栏的 ID 设置为 IDR\_MYTOOL1，其属性如图 2-25 所示。

 图 2-25 工具栏属性对话框

|  |
| --- |
| #include <afxwin.h>  #include <afxext.h> #include "resource.h" class CMyApp:public CWinApp  { public:  virtual BOOL InitInstance();  };  class CMyWnd:public CFrameWnd  { public:  ~CMyWnd();  CToolBar MyToolBar;  afx\_msg int OnCreate(LPCREATESTRUCT lpCreateStruct); afx\_msg void OnButton10(); afx\_msg void OnButton11(); afx\_msg void OnButton12(); afx\_msg void OnButton13();  DECLARE\_MESSAGE\_MAP()  };  CMyApp MyApp;  BEGIN\_MESSAGE\_MAP(CMyWnd,CFrameWnd)  ON\_WM\_CREATE()  ON\_COMMAND(ID\_BUTTON40010,OnButton10) ON\_COMMAND(ID\_BUTTON40011,OnButton11)  ON\_COMMAND(ID\_BUTTON40012,OnButton12) ON\_COMMAND(ID\_BUTTON40013,OnButton13)  END\_MESSAGE\_MAP()  BOOL CMyApp::InitInstance()  {  CMyWnd\* pCWindow=new CMyWnd;  pCWindow->LoadFrame(IDR\_MYTOOLBAR1);//创建主窗口，加载菜单 m\_pMainWnd = pCWindow; |

|  |
| --- |
| pCWindow->ShowWindow(m\_nCmdShow); pCWindow->UpdateWindow(); return TRUE;  }  int CMyWnd::OnCreate(LPCREATESTRUCT lpCreateStruct) {  CFrameWnd::OnCreate(lpCreateStruct);  SetWindowText("ToolBar");  MyToolBar.CreateEx(this, TBSTYLE\_FLAT,  WS\_CHILD | WS\_VISIBLE | CBRS\_TOP| CBRS\_GRIPPER);  MyToolBar.LoadToolBar(IDR\_MYTOOL1);  MyToolBar.EnableDocking(CBRS\_ALIGN\_ANY);    EnableDocking(CBRS\_ALIGN\_ANY); DockControlBar(&MyToolBar); return 0;  }  CMyWnd::~CMyWnd()  { }  void CMyWnd::OnButton10()  {  AfxMessageBox("Click button 10");  }  void CMyWnd::OnButton11()  {  AfxMessageBox("Click button 11");  }  void CMyWnd::OnButton12()  {  AfxMessageBox("Click button 12");  }  void CMyWnd::OnButton13()  {  AfxMessageBox("Click button 13");  } |

程序运行结果如图 2-26 所示。

### 2.5.4 状态栏

利用 AppWizard 自动创建的工程文件中，一般都自动添加了状态栏，在编程过程中，只要在原有状态栏的基础上进行修改就可以了。

状态栏实际上是个窗口，一般分为几个窗格，每个窗格显示不同的信息。AppWizard 会为应用程序自动创建一个默认的状态栏，该状态栏包括几个窗格，分别用来显示状态栏提示、大小写锁定键（CAPS LOCK）、数字锁定键（NUM LOCK）和滚动锁定键（SCROLL LOCK）的状态。在 MFC 中，状态栏的功能由 CStatusBar 类实现，创建一个状态栏需要以下几个步骤：

* 构建一个 CStatusBar 对象；
* 调用 CStatusBar::Create 创建状态栏窗口；
* 调用 CStatusBar::SetIndicators 函数分配窗格，并将状态栏的每一个窗格与一个字符串

ID 相联系。

|  |
| --- |
| 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  } |

实例 2-7：电子时钟。源代码在光盘中“\02\实例 2-7\EClock”目录下。

下面将通过一个电子时钟（在状态栏中嵌入一个电子时钟）的例程来向读者详细讲述工具栏的使用。

首先在 indicators 数组的 ID\_SEPARATOR 项之后插入一个名为 ID\_INDICATOR\_CLOCK 的 ID，然后找到并双击字符串资源，打开字符串资源编辑窗口，接着在编辑窗口内按 Insert 键以插入一个新的字符串，指定字符串的 ID 为 ID\_INDICATOR\_CLOCK，内容为 00:00:00。由于状态栏将根据字符串的长度来确定相应窗格的默认宽度，所以指定为 00:00:00 就为时间的显示预留了空间。添加该字符串资源的结果如图 2-27 所示。时间窗格显示的时间必须每隔一秒钟更新一次。更新时间窗格的正文可调用 CStatusBar:: SetPaneText 函数，要定时更新，则应利用定时器消息（WM\_TIMER）。在 Windows 中用户可以安装一个或多个计时器，计时器每隔一定的时间间隔就会自动发出一个 WM\_TIMER 消息，而这个时间间隔可由用户指定。MFC 的 Window 类提供了 WM\_TIMER 消息处理函数

OnTimer，可以在该函数内进行更新时间窗格的工作。

 图 2-27 添加 ID\_INDICATOR\_CLOCK 字符串资源的结果

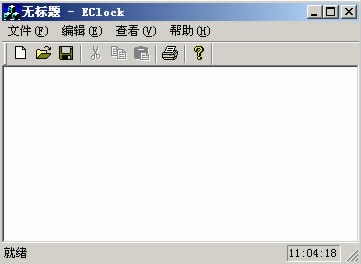
可以利用 ClassWizard 工具给 CMainFrame 类加入 WM\_TIMER 的消息处理函数 OnTimer 和 WM\_CLOSE 消息的处理函数 OnClose。具体方法是在 Class name 栏中选择 CMainFrame，在 Object IDs 栏中选择 CMainFrame，在 Messages 栏中找到 WM\_TIMER 和 WM\_CLOSE 项，分别双击之然后单击“OK”按钮退出 ClassWizard。CMainFrame::OnClose 函数是在关闭主框架窗口是被调用的，程序可以在该函数中做一些清除工作。部分核心代码如下：

|  |
| --- |
| int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)  {  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)  return -1;  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  } |

|  |
| --- |
| m\_wndToolBar.EnableDocking(CBRS\_ALIGN\_ANY);  EnableDocking(CBRS\_ALIGN\_ANY);  DockControlBar(&m\_wndToolBar); SetTimer(1,1000,NULL); //设置时钟定时器 return 0;  }  void CMainFrame::OnTimer(UINT nIDEvent)  {  // TODO: Add your message handler code here and/or call default  CTime time;  time=CTime::GetCurrentTime(); //得到当前时间  CString s=time.Format("%H:%M:%S");//格式化时间显示格式 m\_wndStatusBar.SetPaneText(m\_wndStatusBar.CommandToIndex(ID\_INDICATOR\_CLOCK),s); CFrameWnd::OnTimer(nIDEvent);  }  void CMainFrame::OnClose()  {  // TODO: Add your message handler code here and/or call default  KillTimer(1); //删除定时器  CFrameWnd::OnClose();  } |

在 CMainFrame::OnCreate 函数内调用了 CWnd::SetTimer 以设置一个计时器，SetTimer 的第一个参数指定计时器 ID 为 1，第二个参数则规定了计时器的时间间隔为 1000 毫秒即 1 秒，这样，每隔 1 秒 OnTimer 函数就会被调用一次。在 OnTimer 函数中，首先构建了一个 CTime 对象，接着调用 CTime 的静态成员函数 GetCurrentTime 以获得当前的系统时间，然后利用 CTime::Format 函数返回一个按“时：分：秒”的格式表示的字符串，最后调用 CStatusBar::SetPaneText 来更新时间窗格显示的正文。SetPaneText 的第一个参数是窗格的索引，对于某一个窗格 ID，可调用 CStatusBar::CommandToIndex 来获得索引。在撤销主框架窗口时应关闭计时器，因此在 CMainFrame::OnClose 函数内调用了 KillTimer 函数。

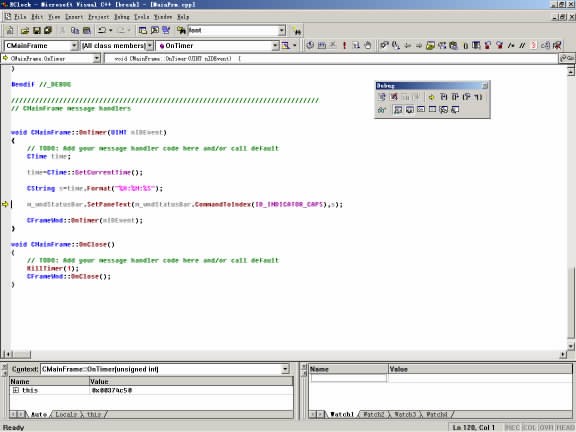
电子时钟运行结果如图 2-28 所示，在框架底部的状态栏的右方可以看到当前时间。

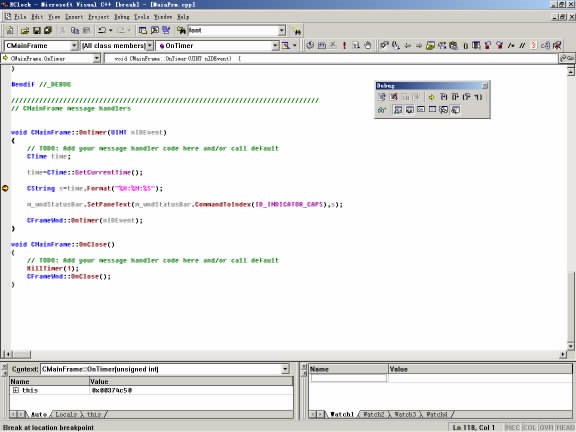
 图 2-28 电子时钟示例程序运行结果

在调试程序的时候，可能常会碰到程序存在 bug 或运行错误的情况，下面将介绍处理错误信息的一般方法。在电子时钟程序中，运行程序时可能提示如图 2-29 的错误信息。

 图 2-29 电子时钟 Debug 错误信息

为了查明导致 Debug Assertion Failed 的错误代码，选择“Build|Debug|Go”菜单命令进入调试状态，这时通过运行时的异常可以判断出代码的大概错误位置，在 Debug 模式下的界面如图 2-30 所示。也可以在对程序出错代码的大概估计的基础上在“可疑”代码附近设置断点到进行调试，界面如图 2-31 所示。

 图 2-30 电子时钟调试信息

 图 2-31 电子时钟断点调试信息

在断点附近处通过按键 F10（Step Over）或 F11（Step Into）键可以找到存在 bug 的代码。通过观察和分析知道，在下面的这行代码中，由于 ID\_INDICATOR\_NUM 没有在 indicators 数组中被定义，才会导致错误。通过更改 ID\_INDICATOR\_NUM 为 ID\_INDICATOR\_CLOCK，电子时钟运行便正常。

m\_wndStatusBar.SetPaneText(m\_wndStatusBar.CommandToIndex(ID\_INDICATOR\_NUM),s);

## 2.6 应用程序框架中各类对象间的关系

通过从 2.1 节到 2.5 节的学习，相信读者对于利用 AppWizard 生成的应用程序框架有了一定的了解。下面以单文档（SDI）应用程序 HelloMFC 为例对应用程序框架中各主要对象的构造以及各对象之间的访问情况等方面进行介绍。

在 HelloMFC 应用程序中，AppWizard 生成了 4 个主要的类，这些类都是 MFC 类的派生类，分别包含在对应的头文件以及实现文件中，这些类的对象构成了框架的核心。

1．框架窗口类及其相关的文件

框架窗口类对应应用程序的主窗口，明白这一点后就可以建立一个对这个类的感性认识了，其定义是在头文件 MainFrm.h 中，而实现则是在 MainFrm.cpp 文件中。所有与框架窗口向光的功能都是在这里定义和实现的。

2．文档类及其相关的文件

文档类在应用程序中没有直观的对应关系，但是，应该明白的一点就是 MFC 框架的一个特点就是文档视图结构。这里可以抽象一点地来理解，比如说在 Word 中打开了一个文件，其实，这个文件就是文档，而看到的只不过是这个文件一个视图，所以说文档提供了应用程序显示的支持，但是真正看到的应用程序显示的则是一个视图，文档类的定义是在

“HelloMFCDoc.h”中，而其实现则是在“HelloMFCDoc.cpp”文件中。

3．视图类及其相关的文件

视图类是用来显示文档对象内容的，在 Word 程序中所看到的界面就是一个视图，要修改、画图，首先操作的对象就是视图，所以，视图类就如其名称所指出的那样，提供了从用户角度能看到的东西。视图类的定义在“HelloMFCView.h”中，实现是在文件“Hello

MFCView.h”中。

4．应用程序类及其相关的文件

MFC 应用程序的初始化，启动运行和结束都是由应用程序对象完成的。他对应的文件是”

HelloMFC.cpp”和”HelloMFC.h”。

从上面的陈述中可以大致看出 MFC 应用程序对象之间的关系了。应用程序一开始生成应用程序对象，然后，在 InitInstance 中将会创建文档模板对象（通过 CSingleDocTemplate 管理），这样一来，应用程序就可以通过建立的模板对象来管理文档、视图、和框架窗口。

在基于文档视图结构的 Windows 程序在不同的调用环境常常需要得到各种对象的指针，以实现对象间数据的交互。实现各对象之间的相互访问有如下几种方法：

* 全局函数 AfxGetApp 可以得到 CWinApp 应用类指针；
* AfxGetApp()->m\_pMainWnd 为框架窗口指针；
* 在框架窗口中 CFrameWnd::GetActiveDocument 得到当前活动文档指针；
* 在框架窗口中 CFrameWnd::GetActiveView 得到当前活动视指针；
* 在视图中 CView::GetDocument 得到对应的文档指针；
* 在文档中 CDocument::GetFirstViewPosition，CDocument::GetNextView 用来遍历所有和文档关联的视；
* 在文档中 CDocument::GetDocTemplate 得到文档模板指针；
* 在多文档界面中：CMDIFrameWnd::MDIGetActive 得到当前活动的 MDI 子窗口。

## 2.7 本章小结

本章主要对通过 MFC 应用程序向导（AppWizard）创建的应用程序框架作了介绍。由于读者是可能第一次接触到 MFC 编程，因此首先对工程及工作区的概念及创建方法进行了介绍。在此基础上通过 AppWizard 工具一步步地创建出的基于单文档（SDI）的 HelloMFC 应用程序，并对其工程文件的组成及作用进行了说明。针对 HelloMFC 应用程序，介绍了类向导（ClassWizard）工具的基本用法，接着对程序的运行过程以及应用类（CWinApp）的几个主要的成员函数进行了解释。由于 HelloMFC 是基于文档视图结构的，接下来对该结构的文档类和视图类的各自作用及相互之间的关系分别进行了说明，同时对经常用到的单文档

（SDI）和多文档（MDI）结构进行了对比。针对框架结构中不可缺少主窗口类（CMainFrame）和子窗口类（CChildFrame），本章也对它们的功能和使用方法进行了介绍，同时介绍了在框架窗口使用频率很高的工具栏（Toolbar）和状态栏（StatusBar）也介绍了创建过程和使用方法，电子时钟（在状态栏中创建一个电子时钟）例程演示了自定义状态栏的使用方法。最后，对应用程序框架中的主要对象的关系及相互访问情况进行了总结。

本章阐述的应用程序框架是对话框、控件、文件操作及 Visual C++高级编程的基础，希望读者能够仔细阅读推敲，并在后面章节的学习过程中能够反复翻阅体会，以便达到对 MFC 应用程序框架的深刻理解。

# 第 3 章 菜单与加速键

菜单是 Windows 程序界面的重要元素，程序的各种操作和功能都是通过单击菜单项完成的。一个设计合理、功能完善的菜单能极大地方便用户的使用。加速键是菜单的辅助功能，程序使用者可以通过快捷键直接访问常用的菜单选项。本章就介绍在程序中创建菜单和加速键的方法。

## 3.1 消息与事件

Windows应用程序是通过消息驱动的，因此消息和事件是Windows程序设计的核心概念。

本节将简要介绍消息和事件的处理过程，为以后的编程实践打下基础。

### 3.1.1 事件驱动机制

在生活中，人们都会对外界发生的各种各样的和自身相关的刺激做出一定的反应，这就是所谓的应激性。Windows 系统中也有类似的应激性，也就是本小节要介绍的概念—事件驱动机制。

Windows 作为一个以图形界面为主的操作系统，主要面向与用户的交互。这个特点决定了在 Windows 上面运行的程序大部分都要以等待用户输入，并根据用户输入做出相应的操作为主要运行方式。因此，用户输入是 Windows 系统中消息和事件的一个主要来源。根据用户输入做出相应的操作，这个过程称为消息处理的过程。同时 Windows 系统也将系统和程序，程序和程序之间的相互作用，抽象为另外一类消息和事件。这样程序在运行中与系统环境的相互作用就体现为接收消息，然后对消息进行处理。这就是事件驱动机制的主要内涵。

Windows 下的应用程序的主要工作是进行消息的循环处理，通过循环等待消息的到来和事件的发生，然后对不同的消息和事件运行相关的代码，完成相应的操作。

### 3.1.2 消息和消息映射

Windows 系统中的程序大部分都是通过消息和事件驱动的。为了完成各种功能，Windows 提供了大量的消息。这些消息主要有 3 种类型.

* 标准 Windows 消息：消息名以 WM\_为前缀的消息（也有例外，如控件通知消息

WM\_COMMAND），例如 WM\_SIZE（改变窗口大小消息）、WM\_MOVE（移动窗口消息）等。

* 控件通知消息：由列表框、按钮等控件或子窗口发给其父窗口的通知消息，以

WM\_COMMAND 为消息名，消息中包含有控件通知代码，用以区分不同的控件通知

消息。

* 命令消息：命令消息也以 WM\_COMMAND 为消息名，在消息中包含有命令的标识符，用以区分不同的命令。命令消息主要来自菜单、工具栏和加速键。其中菜单和加速键消息将是本章要学习的内容。

上面的这 3 类消息在 MFC 中有统一的处理框架，即消息映射机制。消息映射机制将消息及其处理函数表示成一一对应的表，并对这张表进行分析处理。CCmdTarget 类的派生类用于存放消息映射机制，使用消息映射必须在源程序中加入以下两方面内容：

1. 在类的头文件中加入如下语句：

DECLARE\_MESSAGE\_MAP()

例如在类 CTestApp 的头文件 TestApp.h 中有如下代码段：

|  |
| --- |
| class CTestApp : public CWinApp  { public:  CTestApp();    // Overrides  // ClassWizard generated virtual function overrides  //{{AFX\_VIRTUAL(CTestApp) public:  virtual BOOL InitInstance();  //}}AFX\_VIRTUAL    // Implementation  //{{AFX\_MSG(CTestApp) afx\_msg void OnAppAbout();  // NOTE - the ClassWizard will add and remove member functions here.  // DO NOT EDIT what you see in these blocks of generated code !  //}}AFX\_MSG  DECLARE\_MESSAGE\_MAP()  }; |

DECLARE\_MESSAGE\_MAP()就添加在 CTestApp 类的定义程序块的末尾。

1. 在类的实现文件中加入消息映射表：

|  |
| --- |
| BEGIN\_MESSAGE\_MAP(类名, 父类名)  ⋯ ⋯ 消息映射表项 ⋯ ⋯  END\_MESSAGE\_MAP() |

在 CTestApp 类的实现文件 TestApp.cpp 的开头、常量定义的后面，就有如下消息映射表的代码：

/////////////////////////////////////////////////////////////////////////////

// CTestApp

BEGIN\_MESSAGE\_MAP(CTestApp, CWinApp)

//{{AFX\_MSG\_MAP(CTestApp)

ON\_COMMAND(ID\_APP\_ABOUT, OnAppAbout)

// NOTE - the ClassWizard will add and remove mapping macros here.

// DO NOT EDIT what you see in these blocks of generated code!

//}}AFX\_MSG\_MAP

// Standard file based document commands

ON\_COMMAND(ID\_FILE\_NEW, CWinApp::OnFileNew)

ON\_COMMAND(ID\_FILE\_OPEN, CWinApp::OnFileOpen)

// Standard print setup command

ON\_COMMAND(ID\_FILE\_PRINT\_SETUP, CWinApp::OnFilePrintSetup)

END\_MESSAGE\_MAP()

MFC AppWizard 生成的应用程都已经自动包含消息映射的代码，但用户也可以在源程序中手动添加。同时 Visual C++另一个强大的工具 ClassWizard 也提供了处理消息映射表的自动化机制。当然，ClassWizard 只能够生成类和消息处理的框架，包括创建消息处理函数的声明，而具体的实现代码还需要用户去补充。

在类中，消息处理函数都是类的成员函数。当程序要对一个消息进行响应时，消息映射机制就会自动调用该函数。用 ClassWizard 添加消息处理函数时，会自动添加以下 3 部分内容：

* 在类的定义中添加该消息处理成员函数的函数声明；
* 在类的消息映射表中加入相应的消息映射表项； • 在类的实现中加入该消息处理成员函数的函数实现。

注意 ClassWizard 添加的函数实现中，只是一个不能完成任何操作的空函数框架，还需

要加入具体的实现代码。但当用户手动添加消息处理函数时，也必须添加上述 3 部分内容。

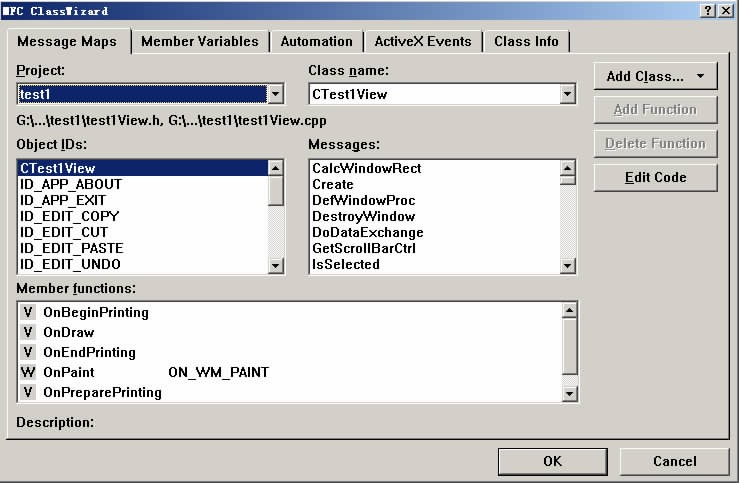
对于标准 Windows 消息，在 CWnd 类中已经预定义了默认的处理函数，有确定的函数名，

一般以 On 开头，以消息名为函数名的后半部分，例如 WM\_PAINT 消息的处理函数为

OnPaint()，WM\_CREATE 消息处理函数是 OnCreate()。这些预定义的消息处理函数根据消息的不同有不同的参数和返回值。

对于控件通知消息和命令消息，一般没有默认的消息处理函数，可以由用户自定义。MFC 建议这类函数命名应遵循一些约定，一般以 On 开头，函数名中应包含该消息的简要说明，方便阅读代码。ClassWizard 在创建这类消息处理函数时会提供一个建议的函数名，一般无需修改。例如 OnFileOpen()函数表明处理命令标识符 ID\_FILE\_OPEN 的命令消息。

本章和以后的章节都将主要利用 ClassWizard 为程序添加消息处理函数。因此下面简要介绍一下 ClassWizard 对话框，如图 3-1 所示。在 IDE 主菜单栏中选择“View|ClassWizard” 菜单命令，或者在开发环境的界面中直接按快捷键 Ctrl+W，即可打开“MFC ClassWizard” 对话框。选择“Message Maps”的选项卡，即可进行编辑消息映射的设置。

 图 3-1 消息映射选项卡

ClassWizard 对话框的“Message Maps”（消息映射）选项卡中的各项功能介绍如下：

* Project（工程）下拉列表框：用于选择需要编辑的工程名称。
* Class name（类名）下拉列表框：用于选择需要编辑消息映射的类的名称。选择好类后，在该下拉列表框 和“Project”下拉列表框的下一行将显示所选类的头文件和源文件路径。
* Object IDs（对象 ID）列表框：以列表形式显示可以产生消息的对象的 ID，包括菜单项、按钮等的 ID。该列表框的第一项总是显示所选择的类。
* Messages（消息）列表框：列举在“Object IDs”列表框中选定的对象所能产生的各种消息。
* Member functions（成员函数）列表框：以列表形式显示选定的类中的成员函数。该列表框中只列出与消息处理函数以及 MFC 中与消息处理有关的虚函数。前者标记为

“W”，并在函数名后显示对应的消息映射宏；后者标记为“V”。

* Add Class（添加类）按钮：单击该按钮，添加新的类到工程中。该按钮在“MFC

ClassWizard”对话框中的所有选项卡中都存在。

* Add Function（添加函数）按钮：单击该按钮添加选定消息的新的消息处理函数，包括在消息映射定义中添加对应的宏指令，在选定的类中添加该消息处理函数的声明和完成基本功能的代码实现。如果选定的消息已经有消息处理函数，则该按钮无效。对于对象产生的消息，单击该按钮将弹出如图 3-2 所示的对话框，要求用户输入消息处理函数的名称，在“Member function name”（成员函数名称）输入框中输入函数名，确定后 ClassWizard 就会自动添加该消息处理函数。对于由该选定的类产生的消息， MFC 已经定义好了消息处理函数名称，单击“Add Function”按钮将直接添加该消息处理函数，不会提示用户输入自定义的函数名。注意，只有单击“MFC ClassWizard” 的“OK”按钮或者“Edit Code”按钮退出“MFC ClassWizard”对话框后，ClassWizard 才会真正修改源文件。

 图 3-2 添加消息处理函数的对话框

* Delete Function（删除函数）按钮：单击该按钮删除选定消息的消息处理函数，包括函数的声明和定义该消息映射的宏语句。如果选定的消息没有消息处理函数，则该按钮无效。注意，当该函数已经存在于源文件中时，单击该按钮会提示如图 3-3 所示的对话框，告诉用户 ClassWizard 只能删除该函数的声明和定义该消息映射的宏语句，而无法删除该函数的实现代码，用户需要手动删除。

 图 3-3 提示用户手动删除消息处理函数的对话框

* Edit Code（编辑代码）按钮：单击该按钮退出“MFC ClassWizard”对话框，同时 IDE 的代码编辑窗口直接转到选定消息的处理函数进行编辑。该按钮也使得在“MFC

ClassWizard”中所做的更改生效。如果选定的消息没有消息处理函数，则该按钮无效。

* OK（确定）按钮：单击该按钮使得在“MFC ClassWizard”中的编辑生效，并退出“MFC

ClassWizard”对话框。

* Cancel（取消）按钮：单击该按钮取消所有“MFC ClassWizard”对话框中的设置，并退出“MFC ClassWizard”对话框。

### 3.1.3 消息处理过程

在 Windows 程序设计中，消息的处理是通过消息循环进行接收与分发的。MFC 运用消息映射机制封装了这一处理过程。下面介绍 MFC 中消息的处理过程。

1．标准 Windows 消息和控件通知消息

标准 Windows 消息和控件通知消息的处理过程比较简单。这类消息必须由 CWnd 或其派生类的对象处理，消息传递到封装对应的窗口或子窗口的类的对象中。也就是说，当关闭程序的时候，会产生一个 WM\_CLOSE 消息并通知主框架类，主框架类调用相应的处理函数进行程序关闭前的清理操作，如保存文件和数据等。

2．命令消息

命令消息与其他两个类型的消息不同，它可以由应用程序其他类的对象（例如应用类对象、文档对象和文档模板对象）处理。这就涉及一个消息传递路径的问题，具体来说就是允许一个还是多个对象处理该消息、允许哪些对象处理等。MFC 中采取了简单的方法，只允许其中一个对象处理该消息，当沿着消息传递路径找到第一个包含该消息的处理函数的对象时，则处理该消息，然后中断传递过程，不再搜索后续的对象。

由于命令消息可以被多种类的对象处理，因此命令消息的传递路径比较复杂，但总体思想是由子窗口到父窗口，由窗口到文档。下面举一个简单的例子。

假设是一个多文档 MDI 应用程序，用户触发了“编辑”菜单下的“粘贴”菜单项的命令消息，此时消息依照以下顺序传递。

（1）MDI 主框架窗口首先收到此命令消息。

1. 主框架将处理命令的机会先让给其当前活动的 MDI 子框架窗口。
2. 子框架窗口又将处理该命令的机会让给其活动的视图。
3. 视图先搜索自己的消息映射，寻找对应的处理函数，如果没找到，则将消息传递给与其关联的文档，找到则调用并中止传递过程。
4. 文档搜索自己的消息映射，寻找对应的处理函数。一般情况下文档中包含“粘贴”消息的处理函数，调用该处理函数完成粘贴功能。如果没有找到，文档将消息传递给文档模板。
5. 文档模板进行相同的过程，如果没找到，将消息传递回子框架窗口。
6. 子框架窗口搜索对应的处理函数，如果没找到，则将消息传回 MDI 主框架窗口。
7. 如果主框架窗口中也没找到，则消息最后停留在应用对象，不再进行进一步的传递。

从上面的叙述可以看出，命令消息的传递过程很复杂。然而一般情况下不需要了解这么详细的消息传递路径，只需根据完成功能的要求，在相应的类中添加实现代码，就不会产生冲突，比如例子中提到的粘贴功能，一般就是由文档完成的，因此只需在文档类中添加这个命令消息的处理函数，而不必在别的类中添加与之冲突的处理函数。

## 3.2 菜单和加速键的创建

在程序中使用菜单和加速键，首先需要创建相应的资源。菜单和加速键资源告诉 Windows 系统自动为程序绘制菜单，并将与菜单和加速键相关的用户操作作为命令消息发送给程序的窗体。资源的存在大大简化了程序的处理，是 Windows 系统程序设计的一个重要部分。

下面分别介绍菜单和加速键资源的创建过程。

### 3.2.1 菜单的创建

实例 3-1：创建菜单实例。源代码在光盘中“\03\实例 3-1\test1”目录下。

1．创建工程

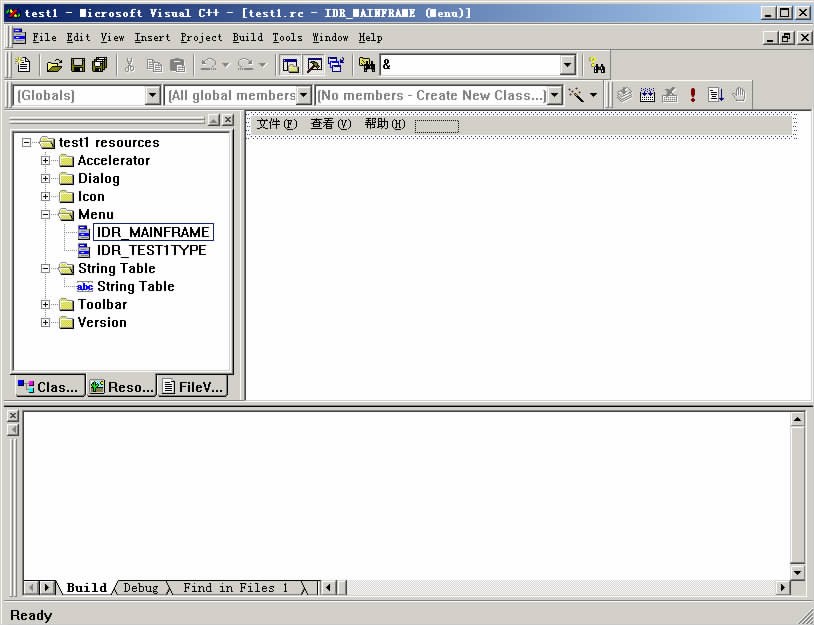
用 AppWizard 创建一个 MDI 工程，工程名为 test1。

创建工程后，可以发现自动生成的应用程序框架中已包含了默认的菜单，根据用户在创建时的不同选项，默认菜单中的菜单项有所不同。对于 SDI 应用程序，只生成一个菜单，其资源 ID 为 IDR\_MAINFRAME（资源 ID 用于标识资源，程序装入资源需要指定该 ID 来确定需要装入哪个资源），这是整个应用程序共用的菜单。对于 MDI 应用程序，要生成两个菜单，其资源 ID 分别为 IDR\_MAINFRAME 和 IDR\_工程名 TYPE，当应用程序尚未打开文档时显示第一个菜单，打开文档后显示第二个菜单。

2．添假菜单资源

添加菜单资源。演示如何为资源 ID 为 IDR\_MAINFRAME 的菜单资源添加一个包含“月光宝盒”和“仙履奇缘”两个子菜单项的“大话西游”菜单项。其步骤如下：

1. 在集成开发环境（IDE）中单击左侧工作区窗口中的“ResourceView”选项卡，在树状列表中单击展开“Menu”节点，找到需要编辑的菜单资源的 ID，双击就可以打开该菜单资源的编辑界面，如图 3-4 所示。

 图 3-4 菜单资源编辑界面

1. 双击 “帮助”菜单项右侧的虚线框，弹出新建顶层菜单项的属性编辑对话框，如图 3-5 所示。注意，这个虚线框可以用鼠标拖动到顶层菜单的任何位置。在“Caption”文本框中输入菜单项的标题“大话西游（&D）”。标题中“&D”的作用是在程序运行时用户可以按 Alt+D 组合键直接打开该菜单项，在显示时，“&D”会自动转换为添加了下划线的“D”，如图 3-6 所示。“Pop-up”复选框默认被选中，表明这是一个弹出菜单项，允许有子菜单。

 图 3-5 添加“大话西游”菜单项

图 3-6 “大话西游”菜单项

1. 关闭如图 3-5 所示的对话框，单击资源编辑器中“大话西游”的菜单项，会显示它的下层菜单，如图 3-6 所示。因为还没有添加任何子菜单项，所以“大话西游”的下层菜单是空的，显示为一个虚线框。在图 3-6 中单击这个虚线框，弹出类似于图 3-5 的属性编辑对话框，如图 3-7 所示。



图 3-7 添加“月光宝盒”菜单项

其中，在“ID”文本框中输入“ID\_MOVIE1”，“Caption”文本框中输入“月光宝盒(&Y)”， “Prompt”文本框中输入“打开电影《大话西游之月光宝盒》”。标题中的“&Y”表示打开“大话西游”菜单项后，在键盘上按“Y”键可以直接选择“月光宝盒”菜单项。当鼠标光标移动到“月光宝盒”这个菜单项上时，会在程序的状态栏中显示“Prompt”文本框中的内容，即“打开电影《大话西游之月光宝盒》”。回车关闭属性编辑对话框，保存修改。

1. 按照步骤（3）的方法加入菜单项“仙履奇缘”，其属性对话框如图 3-8 所示。

 图 3-8 添加“月光宝盒”菜单项

1. 完成两个子菜单项的添加后，在资源编辑器中得到如图 3-9 所示的效果。



图 3-9 添加完成后的“大话西游”菜单项

### 3.2.2 加速键的创建

键盘加速键是应用程序中定义的某些组合键，按下组合键可以快捷地选择相应的菜单项和执行某些相应的操作。加速键可以和菜单项相关联，也可以定义某些菜单上没有提供的命令。例如 Visual C++ IDE 中复制和粘贴的快捷键分别是 Ctrl+C 和 Ctrl+V，这两个快捷键就是直接关联到 IDE“编辑”菜单下的“复制”和“粘贴”菜单项。与菜单项的关联只需要将加速键的 ID 设置为对应菜单项的 ID，比如 Ctrl+C 的 ID 设置为 ID\_EDIT\_COPY 就可以关联到

ID 同样是 ID\_EDIT\_COPY 的“编辑”菜单下的“复制”菜单项。

下面以在 3.2.1 节中创建的工程中添加加速键为例介绍添加加速键的方法。

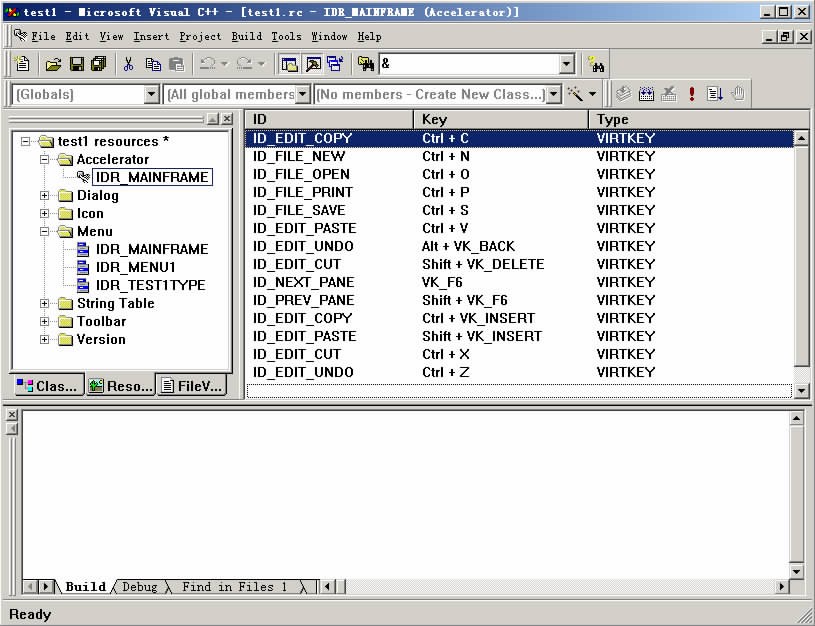
1. 在集成开发环境（IDE）左侧单击“ResourceView”选项卡，在树状列表中寻找 “Accelerator”节点下的“IDR\_MAINFRAME”子节点，单击该节点就会在右侧客户窗口中打开加速键资源 IDR\_MAINFRAME 的加速键列表，如图 3-10 所示。IDR\_MAINFRAME 加速键资源是 AppWizard 在创建应用程序时默认添加的，一般无需另外创建新的加速键资源。
2. 双击编辑界面右侧客户区最下方的虚线框，打开一个加速键属性编辑对话框，如图

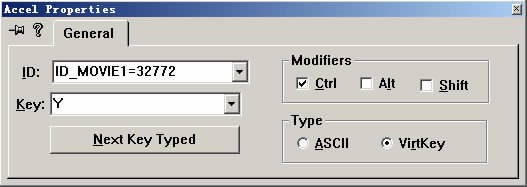
3-11 所示，在该对话框中即可添加新的加速键。

在属性编辑对话框的“ID”文本框中输入“ID\_MOVIE1”，对应“大话西游”菜单项下的“月光宝盒”菜单项的 ID，输入完毕后“ID”文本框会自动显示“ID\_MOVIE1=数字”，这里数字表示 ID\_MOVIE1 的数值，一般由 Visual C++自动给出，用户无需更改。

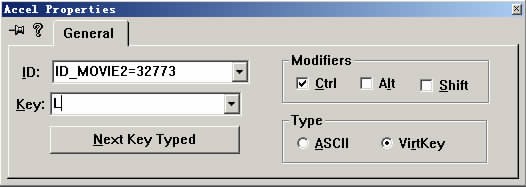
在“Key”文本框中输入“Y”，选中“Modifiers”区域中的“Ctrl”复选框，取消其他复选框的的选取，在“Type”区域中选中“VirtKey”单选框，表示使用虚拟键（Virtual Key）。

这样就建立起加速键 Ctrl+Y 和“月光宝盒”菜单项的对应。程序运行后，只需按 Ctrl+Y 键就可以选择“月光宝盒”菜单项，触发相应的命令消息了。

 图 3-10 加速键资源编辑界面

 图 3-11 “月光宝盒”加速键资源编辑界面

1. 按照步骤（2）中的方法，加入加速键 Ctrl+L 与“仙履奇缘”菜单项的对应，相应的属性对话框如图 3-12 所示。

 图 3-12 “仙履奇缘”加速键资源编辑界面

1. 完成加速键的添加后，为了完善用户界面，需要在菜单项标题中添加加速键的说明。在 IDE 左侧的“ResourceView”选项卡中，展开“Menu”节点，双击 IDR\_MAINFRAME 进入 IDR\_MAINFRAME 菜单资源编辑界面。然后在 IDE 右侧客户区中单击“大话西游”菜单项，弹出子菜单，在“月光宝盒”菜单项上双击鼠标左键，弹出属性编辑对话框，将“Caption” 文本框中的标题修改为“月光宝盒(&Y)\tCtrl+Y”，如图 3-13 所示。其中“\t”表示“Ctrl+Y” 的显示与前面“月光宝盒(&Y)”之间隔一个制表位（Tab）。

 图 3-13 添加“月光宝盒”的加速键说明

1. 参照前一步，在“仙履奇缘”菜单标题中加上 Ctrl+L 的加速键说明。

## 3.3 添加菜单所需处理函数

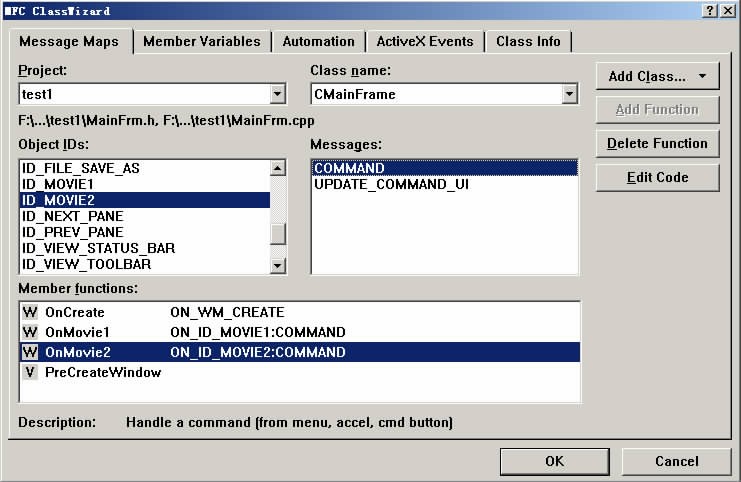
前一节介绍了创建和添加菜单资源的方法，但添加菜单资源只是创建菜单的第一步。用户编译执行前一节的例子就会发现，运行时关闭所有文档，程序显示的 IDR\_MAINFRAME 菜单中“大话西游”下的两个菜单项都是禁用的。这是因为缺少相应的用户处理函数，主框架类自动禁用了这两个菜单项。因此还需要为菜单消息添加消息处理函数。同时为了完善菜单的各种功能，还需要在运行中动态地创建和修改菜单，并添加代码。本节将详细讲解这些内容。

### 3.3.1 添加普通菜单的处理函数

1．添加菜单命令处理函数

由 AppWizard 创建的应用程序会在程序运行时自动将菜单资源装入内存。用户只需要添加相应消息处理函数即可完成菜单的基本功能。下面以 3.2 节中创建的菜单为例，分步骤讲解如何为应用程序添加菜单命令消息处理函数。

1. 在 IDE 主菜单栏中依次选择“View|ClassWizard”菜单命令或者在 IDE 界面中直接使用快捷键 Ctrl+W 打开“ClassWizard”对话框，单击“Message Maps”选项卡。
2. 添加“月光宝盒”菜单项的消息处理函数。在“Class name”下拉列表框中选择需要编辑的类—主框架类 CMainFrame，在“Object IDs”列表框中选择需要添加处理函数的命令消息 ID—ID\_MOVIE1“月光宝盒”，双击“Messages”列表框中的 COMMAND 或者在其上单击鼠标左键，然后单击“Add Function”按钮，为 ID\_MOVIE1 添加命令消息处理函数。在弹出的如图 3-2 所示的对话框中直接单击“OK”按钮（对话框中函数名为默认的 OnMovie1），返回 ClassWizard 对话框。可以看见在“Member functions”列表中已经添加了一个名为 OnMovie1 的新函数。
3. 按照步骤（2）中的方法，为 ID 为 ID\_MOVIE2“仙履奇缘”的菜单项添加消息处理函数 OnMovie2()。添加后的 ClassWizard 对话框内容如图 3-14 所示。

 图 3-14 添加了两个命令消息处理函数后的 ClassWizard 对话框

1. 在“Member functions”列表框中选择函数 OnMovie1，单击“Edit Code”按钮， 关闭 ClassWizard 对话框，自动打开 MainFrm.cpp 文件，并将光标定位在 OnMovie1()函数体中，在这里添加相应代码。这里为 OnMovie1()函数添加如下代码：

void CMainFrame::OnMovie1()

{

// TODO: Add your command handler code here

MessageBox("悟空你也太调皮了，我跟你说过叫你不要乱扔东西，你怎么又⋯ ",

"月光宝盒");

}

1. 在 MainFrm.cpp 文件中找到 CMainFrame::OnMovie2()函数，添加如下代码：

void CMainFrame::OnMovie2()

{

// TODO: Add your command handler code here

MessageBox("曾经有一份真诚的爱情摆在我的面前，但是我没有珍惜⋯ ",

"仙履奇缘");

}

1. 保存文件，编译并运行程序，关闭所有文档，依次选择“大话西游|仙履奇缘”可以得到如图 3-15 所示的运行结果。

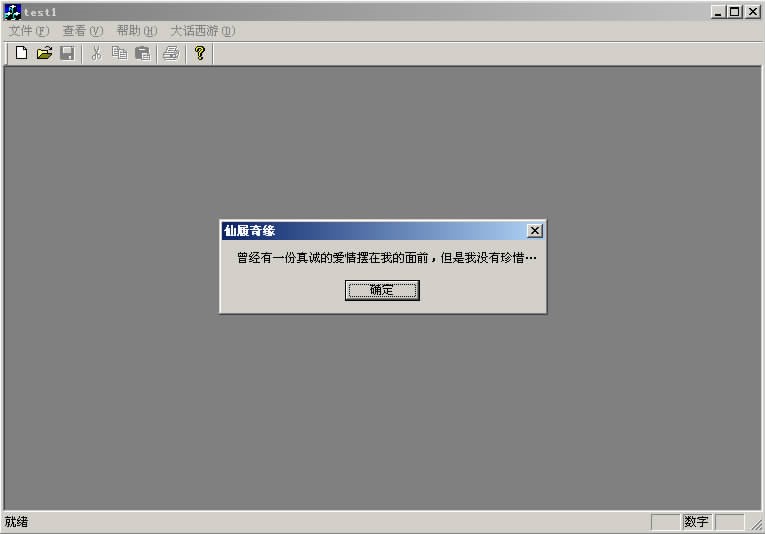


图 3-15 菜单应用程序的运行结果

2．添加动态修改菜单状态的处理函数

每一个菜单项都有很多属性，通过动态改变菜单项的属性可以及时反映菜单的状态，为用户提供相关的信息。但在介绍菜单资源编辑器的时候也涉及到了一些跟菜单属性有关的内容。在资源编辑器中只能设置菜单的初始状态，并不能在程序运行时动态地改变菜单属性，这就需要通过程序代码来实现这些功能。

在前一部分中使用 ClassWizard 对话框添加消息处理函数时可以看到如图 3-14 所示的菜单 ID 对应的“Messages”列表框中有两项，一项是已经用到的“COMMAND”，另一项 “UPDATE\_COMMAND\_UI”则是本部分将要介绍的菜单项的状态更新消息。该消息在 MFC 要更新整个程序界面时产生，提示用户设置该菜单项当前的状态。

下面为前一节的示例添加“仙履奇缘”菜单项的状态更新处理，要求必须在选择过“月

光宝盒”菜单项后才能选择“仙履奇缘”，否则“仙履奇缘”菜单项处于禁用状态。

1. 为主框架类添加一个成员变量，用以标记“月光宝盒”菜单项是否被选择过。要添加成员变量，首先应在 IDE 左侧单击“ClassView”选项卡，展开“Classes”节点，在 CMainFrame 节点上单击鼠标右键，在弹出菜单中选择“Add Member Variable”菜单项，打开“Add Member Variable”对话框，如图 3-16 所示。在其中的“Variable Type”变量类型文本框中输入“BOOL”，在“Variable Name”变量名文本框中输入“m\_bHaveSeenMovie1”，并选中“Access”区域中的“Public”单选框。确认后即为 CMainFrame 添加了一个公共成员变量 m\_bHaveSeenMovie1。该变量的值将决定是否禁用“仙履奇缘”菜单项，当它的值为 TRUE 时，菜单项允许，为 FALSE 时则被禁用。



图 3-16 为 CMainFrame 添加的成员变量

1. 初始化新创建的成员变量，初始值为 FALSE，即“仙履奇缘”菜单项默认是被禁用的。在 IDE 工作区的 ClassView 选项卡下，展开 CMainFrame，双击类的构造函数

CMainFrame()，客户区自动打开 MainFrame.cpp 文件并定位在 CMainFrame::CMainFrame()函数上。在该函数中添加如下代码：

CMainFrame::CMainFrame()

{

// TODO: add member initialization code here m\_bHaveSeenMovie1 = FALSE;

}

1. 在 CMainFrame 中添加“仙履奇缘”菜单项的更新用户接口对象状态命令消息处理函数。

在 IDE 主菜单中依次选择“View|ClassWizard”菜单命令或者在开发界面中直接使用快

捷键 Ctrl+W 打开“ClassWizard”对话框，单击“Message Maps”选项卡，在“Class name” 下拉列表框中选择 CMainFrame，在“Object IDs”列表框中选择 ID\_MOVIE2“仙履奇缘” 项，双击“Messages”列表框中的 UPDATE\_COMMAND\_UI 或者在其上单击鼠标左键，然后单击“Add Function”按钮，为 ID\_MOVIE2 添加更新用户接口对象状态命令消息处理函数。在弹出的如图 3-2 所示的对话框中直接单击“OK”按钮（对话框中函数名为默认 OnUpdateMovie2），返回“ClassWizard”对话框。可以看见在“Member functions”列表中添加了一个名为 OnUpdateMovie2 的新函数。单击“Edit Code”按钮，则退出“ClassWizard” 对话框，并打开 MainFrm.cpp 文件，在刚添加的 CMainFrame::OnUpdateMovie2()中添加如下代码：

void CMainFrame::OnUpdateMovie2(CCmdUI\* pCmdUI)

{

// TODO: Add your command update UI handler code here pCmdUI->Enable(m\_bHaveSeenMovie1);

}

1. 接着根据是否选择过“月光宝盒”菜单项设置“仙履奇缘”是否禁用。当选择“月光宝盒”菜单项后，使其消息处理函数将 m\_bHaveSeenMovie1 设置为 TRUE 就可以激活“仙履奇缘”菜单项。在工作区 ClassView 选项卡中展开 CMainFrame，双击 OnMovie1()节点，IDE 自动打开 MainFrm.cpp 并定位 OnMovie1()函数。修改该函数代码如下：

void CMainFrame::OnMovie1()

{

// TODO: Add your command handler code here

MessageBox("悟空你也太调皮了，我跟你说过叫你不要乱扔东西，你怎么又⋯ ",

"月光宝盒");

m\_bHaveSeenMovie1=TRUE;

}

1. 编译并运行该程序，就可以实现既定功能。

在 CMainFrame::OnUpdateMovie2()函数中，参数 pCmdUI 指向当前要更新的菜单项的对象，这里即为“仙履奇缘”菜单项。通过 pCmdUI 的成员方法就可以改变菜单项的属性。下面分别介绍 CCmdUI 中改变菜单属性主要的成员方法：

• Enable()

Enable()函数用于设置菜单项是否有效。其原型为：

virtual void Enable(BOOL bOn = TRUE);

bOn 为 TRUE 时菜单项有效，反之菜单项无效，显示为灰色，不响应鼠标单击或者其他有同样效果的操作。bOn 的默认值是 TRUE。下面的代码根据标志 m\_bMenuItemEnable 变量设置某个菜单项是否有效：

void CMainFrame::OnUpdateMenuitem (CCmdUI\* pCmdUI)

{

pCmdUI->Enalbem\_bMenuItemEnable);

}

无效的子菜单如图 3-17 所示。

图 3-17 无效的菜单项 Menu

* SetCheck()

SetCheck()函数用于设置菜单项的复选状态。其原型为：

virtual void SetCheck(int nCheck = 1 );

nCheck 等于 0 时菜单项没有被选中，nCheck 等于 1 时菜单项被选中，在菜单项标题前有一个小勾。nCheck 的默认值为 1。下面的代码根据标志 m\_bMenuItemChecked 变量是否为真设置某个菜单项是否被复选选中。

|  |
| --- |
| void CMainFrame::OnUpdateMenuitem (CCmdUI\* pCmdUI)  {  if (m\_bMenuItemChecked) pCmdUI->SetCheck(1); else  pCmdUI->SetCheck(0);  } |

菜单项被复选选中的效果如图 3-18 所示。



图 3-18 复选选中的菜单项 Menu1 和 Menu2

* SetRadio()

SetRadio()函数用于设置菜单项的单选状态。其原型为：

virtual void SetRadio(BOOL bOn = TRUE );

bOn 等于 FALSE 时菜单项没有被选中，bOn 等于 FALSE 时菜单项被选中，在菜单项标题前有一个小圆点。bOn 的默认值为 TRUE。下面的代码根据标志 m\_nMenuItemChosen 变量设置某个菜单项是否被单选选中。

void CMainFrame::OnUpdateMenuitem (CCmdUI\* pCmdUI)

{

pCmdUI->SetRadio(m\_nMenuItemChosen==2);

}

菜单项被单选选中的效果如图 3-19 所示。



图 3-19 单选选中的菜单项 Menu1

注意，SetRadio()函数和 SetCheck()函数的惟一区别是标题前面画的是圆点，而不是小勾，系统并没有如同对话框的单选框控件一样提供内部的处理程序使得在同一列菜单中只有惟一的菜单项被选中，需要程序中自行处理。

• SetText()

SetText 函数用于设置菜单项的标题。其原型为：

virtual void SetText(LPCTSTR lpszText);

参数 lpszText 用于设置菜单项的新标题。下面的代码根据标志 m\_nColor 设置某个菜单项的标题：

|  |
| --- |
| void CMainFrame::OnUpdateMenuitem (CCmdUI\* pCmdUI)  {  if (m\_nColor=1)  pCmdUI->SetText("Red"); else  pCmdUI->SetText("Blue");  } |

### 3.3.2 添加弹出式菜单的处理函数

弹出式菜单是指单击鼠标右键在鼠标光标旁出现的活动菜单，如图 3-20 所示。弹出式菜单给与程序使用者一个快捷的访问菜单的方式，在 Windows 程序界面设计中十分常见。本节就介绍如何创建这种类型的菜单。

1．动态创建菜单对象

弹出式菜单由于不像普通菜单一样可以在程序运行前预先装入，因此都是动态创建的。弹出式菜单一般都是由单击鼠标右键产生的，故需要在鼠标右键按下的消息中进行弹出式菜单的处理。

利用 ClassWizard 为一个应用程序视图类（这里命名为 CSampleView）加入鼠标右键按下的消息处理函数的过程如下：

1. IDE 主菜单栏中依次选择“View|ClassWizard”菜单命令或者在开发环境的界面中直接使用快捷键 Ctrl+W 打开 ClassWizard 对话框。
2. 单击 Message Maps 选项卡，在“Class name”下拉列表框中选择“CSampleView”，在“Object IDs”列表框中选择“CSampleView”列出该视图可以接收的标准 Windows 消息和控件通知消息，双击“Message”列表框中的“WM\_RBUTTONDOWN”或者在其上单击左键，然后单击“Add Function”按钮，为 CSampleView 添加 WM\_RBUTTONDOWN 鼠标右键按下的消息处理函数，函数名默认为 OnRButtonDown。
3. 可以看见在“Member functions”列表中添加了一个名为 OnUpdateMovie2 的新函数。单击“Edit Code”按钮，则退出 ClassWizard 并打开 MainFrm.cpp 文件，创建一个

WM\_RBUTTONDOWN 的消息处理函数，得到类似于下面的函数：

void CSampleView::OnRButtonDown(UINT nFlags, CPoint point)

{

// TODO: Add your message handler code here and/or call default

CView::OnRButtonDown(nFlags, point);

}

在函数内的注释语句后添加弹出式菜单的实现代码。首先创建一个动态菜单对象，有以

下两种方法：

1. 创建一个新的菜单对象以下代码创建一个空的弹出式菜单对象：

CMenu menu; menu.CreatePopupMenu();

CreatePopupMenu()函数为菜单对象 menu 创建一个空的弹出式菜单对象。然后用

AppendMenu()函数为这个新的菜单对象添加菜单项，例如：

menu.AppendMenu(0,ID\_TEST1,"Test1"); menu.AppendMenu(0,ID\_TEST2,"Test2");

1. 从资源中装入一个菜单对象创建动态菜单对象，还可以直接从资源中装入，而无须在代码中手动添加菜单项。首先依照 3.2 节中的内容为应用程序工程新创建一个菜单资源，在 IDE 主菜单栏中依次选择

“Insert|Resource”菜单命令，或者使用快捷键 Ctrl+R，打开“Insert Resource”对话框，在列表框中选择 Menu，单击“New”按钮即可创建一个有默认资源 ID 为 IDR\_MENU1 的新的菜单资源，并自动进入该菜单的编辑界面，然后添加菜单项。图 3-21 给出了一个已经修改好的弹出菜单资源的例子。

然后编辑视图类的源文件，在已经添加的 WM\_RBUTTONDOWN 消息处理函数

OnRButtonDown()中添加如下代码：

CMenu menu;

menu.LoadMenu(IDR\_MENU1);

CMenu \*psubMenu=menu.GetSubMenu(0);

LoadMenu()函数为 CMenu 对象装入菜单资源，其参数为需要装入菜单资源的 ID。于是

psubMenu 就指向了菜单资源 IDR\_MENU1 里 test1 菜单对象，包含 test1 菜单中的所有菜单项。

2．显示并使用弹出式菜单

在视图类的 OnRButtonDown()函数中完成菜单的创建后，就可以根据鼠标光标的位置显示弹出式菜单，代码如下：

CPoint pt;

GetCursorPos(&pt); //获得鼠标位置

menu.TrackPopupMenu(TPM\_RIGHTBUTTON,pt.x,pt.y,this);

变量 menu 是前一小节的第一部分例子中创建的新的菜单对象。如果使用前一小节第二部分从菜单资源中装入的菜单对象 psubMenu，也可以用类似的代码显示弹出式菜单。

GetCursorPos()用于获取鼠标光标位置。TrackPopupMenu()函数在指定位置显示一个浮动的弹出式菜单，并跟踪对其菜单项的选择。

3．销毁弹出式菜单

弹出式菜单使用完毕后，需要在视图类的 OnRButtonDown 中用 DestroyMenu()销毁创建的菜单，例如：

menu.DestroyMenu();

这样就完成了整个弹出式菜单的生命周期。

编译运行程序后，在视图中单击鼠标右键，就可以打开如图 3-20 所示的弹出式菜单。

## 3.4 创建加速键

与菜单的创建相比，加速键的创建相对简单。这是因为加速键已经自动绑定到对应的菜单项命令消息中，当已经完成菜单项命令消息处理函数的编写时，则按下加速键后会自动产生相应的命令消息，并通过相应的消息处理函数完成指定的操作。因此用户只需了解加速键资源如何装入内存即可。

与菜单资源一样，加速键资源也必须在使用前先装入内存。这个过程是由 CFrameWnd 的成员函数 LoadAccelTable()完成的。这个函数为框架窗口动态装入加速键资源。

LoadAccelTable()的原型为：

BOOL LoadAccelTable(

LPCTSTR lpszResourceName

);

参数 lpszResourceName 指向加速键资源的名称，也可以指向通过 MAKEINTRESOURCE 宏转换的加速键资源 ID。

用户有时候可能需要用到两套不同的加速键，根据当前程序运行的状态选择不同的加速键。比如一个既能进行文本编辑，又能进行图像编辑的程序，当进行文本编辑时程序使用一套加速键，而当进行图像编辑时，程序却使用另外一套加速键，相同的加速键在不同的加速键资源中可能对应不同的命令消息，因此需要动态加载加速键，这时候就可能用到

LoadAccelTable()函数。

创建新的加速键资源的过程与创建新的菜单资源类似。例如要创建并使用一个 ID 为

IDR\_MYACCEL 的加速键资源，可以通过以下步骤：

1. 在 IDE 主菜单栏中依次选择“Insert|Resource”菜单命令，或者使用快捷键 Ctrl+R，打开“Insert Resource”对话框，在列表框中选择 Accelerator，单击“New”按钮即可创建一个有默认资源 ID 为 IDR\_ACCELERATOR1 的新的菜单资源（如图 3-7 所示，不过此时在

“Resource type”树状列表框中选择的是 Accelerator），并自动进入该菜单的编辑界面。

1. 找到 IDE 左侧的“ResourceView”选项卡中 IDR\_ACCELERATOR1 的节点，打开资源属性编辑对话框，在 ID 输入框中将资源 ID 修改为 IDR\_MYACCEL。然后参照 3.2 节中的方法添加加速键，完成加速键资源的创建。
2. 创建新的加速键资源后，就需要在程序中要求切换加速键资源的地方添加如下代码：

((CMainFrame \*)AfxGetMainWnd())->LoadAccelTable(

MAKEINTRESOURCE(IDR\_ACCEL));

这句代码装入 ID 为 IDR\_ACCEL 的加速键资源。

用户可以自己创建和使用加速键资源，但特别是由 AppWizard 生成的程序代码中，MFC 已经封装了上述过程。同普通菜单资源的创建一样，MFC 的文档视图结构在装入文档模板的时候，自动将 ID 为 IDR\_MAINFRAME 的加速键资源装入内存，这个过程是和其他资源包括菜单资源一起装入内存的。用户无需深究其中的细节，只需要编辑 IDR\_MAINFRAME 的加速键资源就可以很方便地为程序创建加速键。

## 3.5 本章小结

本章介绍了 Windows 界面设计的两个基本元素—菜单和加速键的设计和使用，同时还介绍了 Windows 消息驱动机制的概念和遵循这一机制的程序设计方法。读者在本章应当掌握如下几个内容：

* Windows 消息驱动机制以及如何用 ClassWizard 添加消息处理函数；
* 菜单和加速键资源的编辑；
* 在程序中使用菜单和加速键。

Windows 消息驱动机制和 ClassWizard 添加消息处理函数的功能是 Windows 程序设计的重点，在以后章节中也反复出现，读者需要仔细理解和掌握这一内容。

# 第 4 章 对话框

对话框是程序中一个常见的用户界面元素。一个具有良好的人机交互功能的软件中，对话框要操作简便，功能完备，信息明了，整体统一。本章将详细介绍对话框的一般创建流程，模态和非模态的对话框、属性页对话框、各种通用对话框以及消息对话框的使用方法。为了使读者能深入了解对话框的编程方法和使用方法，本章中给出了大量的示例，并提供了相关类和函数的参考说明。

## 4.1 对话框的创建

与第 3 章菜单、加速键等 Windows 界面元素一样，对话框也是 Windows 程序的一种资源，称作对话框模板。本节介绍对话框模板的创建和编辑方法。

### 4.1.1 对话框的创建流程

对话框的创建流程如图 4-1 所示，主要分为两个大步骤。

创建对话框资源包括添加对话框模板、修改对话框的属性并向对话框中添加各种控件两步。

创建对话框类包括添加控件变量和添加控件消息处理函数这两步。

实例 4-1：创建对话框实例。源代码在光盘中“\04\实例 4-1\Sample4\_1”目录下。

下面的 4.1.2 和 4.1.3 小节将通过实例 Sample4\_1 介绍对话框创建的各个步骤和方法。

为应用程序 Sample4\_1 添加一个登录对话框。该对话框在应用程序运行的开始时弹出，提示用户输入能运行此程序的用户名和密码。如果输入正确，就可以打开程序的主窗口进入程序，如果输入错误，则直接退出程序。

首先创建工程 Sample4\_1。

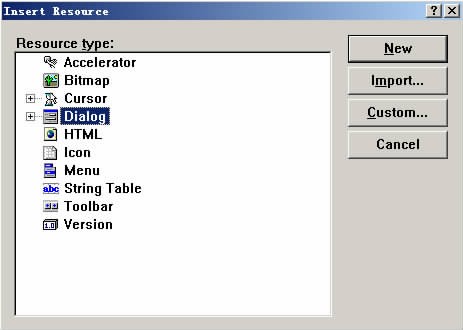
1. 在 IDE 中依次选择菜单“File|New”，或直接按快捷键 Ctrl+N，打开“New”对话框。
2. 在“New”对话框中单击“Projects”选项卡，在列表框中选择“MFC AppWizard(exe)” 项，在“Location”文本框中输入工程所在的目录，在“Project name”文本框中输入“Sample4\_1”，其他使用默认值，单击“OK”按钮，弹出“MFC AppWizard Step-1”对话框。
3. 在“MFC AppWizard Step-1”对话框中，选中“Multiple Document”单选框，其他使用默认值，然后单击“Finish”按钮，在弹出的“New Project Information”对话框中单击“OK” 按钮，就可以完成工程的建立。

### 4.1.2 创建对话框资源

创建对话框首先要创建对应的资源。以下通过实例 Sample4\_1，具体地讲解如何运用对话框编辑器创建对话框资源的方法。

1．为应用程序添加对话框

打开 Sample4\_1 工程后，在 IDE 中依次选择“Insert|Resource”菜单，打开“Insert Resource” 对话框，在“Resource type”属性列表框中选择 Dialog 选项，如图 4-2 所示。然后单击“New” 按钮，为程序添加一个新的对话框，并进入对话框编辑器，如图 4-3 所示。

 图 4-2 添加资源对话框

进入如图 4-3 所示对话框编辑器后，可以看到 IDE 左侧工作区窗口的“ResourceView” 选项卡中列出的工程资源文件中添加了一个新的对话框 ID。

|  |
| --- |
| 技巧：在 IDE 左侧工作区窗口的“ResourceView”选项卡中，可以选定某个资源，进行复制、粘贴操作，可以添加与原来资源一样的新资源。例如在图 4-3 中，在 IDE 左侧工作区的 “ResourceView”选项卡中，展开 Dialog 节点，选中 IDD\_DIALOG1，依次选择菜单  “Edit|Copy”或直接按 Ctrl+C 复制资源，然后依次选择菜单“Edit|Paste”或直接按 Ctrl+V 就可以在工程资源中粘贴一个内容与 IDD\_DIALOG1 一样的资源。 |

2．修改对话框属性

为 Sample4\_1 添加新的对话框后，该对话框的属性都是由 IDE 设置的默认值，因此第二步应当修改对话框的属性。将鼠标光标移动到对话框编辑器中需要编辑的对话框上，然后单击鼠标右键打开弹出式菜单，选择“Properties”属性，打开“Dialog Properties”对话框，如图 4-4 所示。

 图 4-4 对话框属性编辑对话框

对如图 4-3 所示对话框的属性进行如下修改：

1. 修改对话框 ID，在“ID”输入框中输入“IDD\_LOGINDIALOG”。
2. 修改对话框标题，在“Caption”输入框中输入“登录对话框”。
3. 为了在中文系统下的美观，修改对话框字体。单击“Font”按钮，弹出“Select Dialog Font”对话框，在“Font”列表框中选择字体为“宋体”，在“Size”列表框中设置字号为 10，单击“OK”确认退出“Select Dialog Font”对话框。
4. 修改完成后，回车确认修改并退出“Dialog Properties”对话框。 3．在对话框中添加控件

依据实例要求，确认和取消按钮已经默认添加好了，只需要添加用户名和密码两个输入框，并添加相应的说明。具体步骤如下：

1. 在对话框中添加一个静态文本框（Static Text）。在如图 4-3 所示的客户区右侧浮动的 Control 工具栏中的“Static Text”控件按钮上单击，然后在对话框资源中按下鼠标左键拖动，出现一个矩形框，松开鼠标左键，则在对话框资源中出现了一个“Static Text”控件。

另外，添加控件还可以利用鼠标将 Control 工具栏中的控件直接拖到对话框资源中。单击对话框资源中的控件后，控件周围会出现一个虚线框，表示当前选中该控件。在控件中央按下鼠标左键不放可以将控件拖动到指定位置，在虚线框上按下鼠标左键不放并拖动，可以调整控件的大小。按 Delete 键可以删除当前选中的控件。通过这些操作将添加的“Static Text” 控件的位置和大小调整到如图 4-5 中标题为“用户名：”的静态文本框控件所示位置和大小。

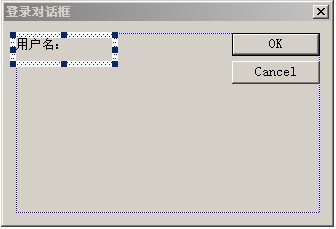


图 4-5 在登录对话框中添加一个静态文本框

此时添加的“Static Text”控件，显示“Static”的字样。接着将鼠标移动到该控件上，单击鼠标右键，在弹出菜单中选择“Properties”，打开控件的属性编辑对话框，修改标题（在

Caption 输入框中）为“用户名：”，如图 4-6 所示。

 图 4-6 修改静态文本框的属性

1. 在对话框中添加输入框（Edit Box），添加过程与步骤（1）相同，但应当在 Control 工具栏中选择“Edit Box”控件的图标，将该控件调整到与如图 4-7 所示虚线框选中的控件一致。

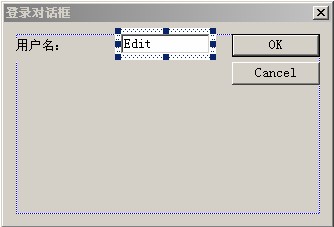


图 4-7 在登录对话框中添加一个输入框

打开控件属性编辑对话框，在 ID 输入框中将输入框的 ID 改为 IDC\_LOGINNAME，如图 4-8 所示。



图 4-8 修改输入框的属性

1. 用步骤（1）的方法添加一个标题为“密码：”的静态文本框。
2. 用步骤（2）的方法添加一个 ID 为 IDC\_PASSWORD 的输入框作为密码输入框，注意修改输入框的属性为密码框。

|  |
| --- |
| 技巧：（1）在对话框编辑界面下，可以选定一个或通过按住 Ctrl 键不放用鼠标左键单击选定多个控件，在对话框中复制并粘贴，可以添加与原来控件外观一样的新控件。  （2）输入框用作输入密码时，需要保证输入的字符不被回显在输入框中以防止密码被窃取，这需要设置输入框属性为密码输入框。将鼠标移动到输入框控件上，单击鼠标左键，打开属性编辑对话框，单击“Styles”选项卡，选中“Password”复选框（如图 4-9 所示）。选中此复选框后，输入框会自动将输入的字符回显为“\*”。 |

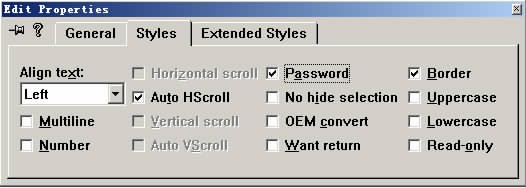


图 4-9 设置输入框的密码属性

1. 打开“OK”和“Cancel”按钮的属性编辑对话框，在“Caption”输入框中将标题分别改为“确认”和“取消”，并调整两个按钮的位置和对话框的大小，得到如图 4-10 所示的登录对话框。



图 4-10 设置好界面的登录对话框

1. 设置对话框中各种控件的响应顺序，即运行时按下 Tab 键，输入焦点从一个控件切换到另一个控件的顺序。

在对话框编辑界面中，可以通过选择主菜单“Layout|Tab Order”或者按快捷键 Ctrl+D，进入 tab 顺序的设置界面，此时对话框资源编辑器的客户区如图 4-11 所示。

如图 4-11 所示，每个控件左上角的数字就是它的响应顺序。标明数字 1 的“确认”按钮控件表示对话框打开时输入焦点最初的位置。然后按下 Tab 键，输入焦点转移到标明数字 2 的“取消”按钮。再按下 Tab 键，输入焦点转移到标明数字 3 的“用户名：”静态文本框，由于静态文本框不接受输入，因此输入焦点自动转移到标明数字 4 的输入框。其他数字以此类推。

设置 Tab 顺序是通过按所需的顺序依次单击各个控件完成，如进入如图 4-11 所示的界面后，依次单击“用户名”输入框，“用户名”静态文本框，“密码”输入框，“密码”静态文本框，“确认”按钮和“取消”按钮，就可设置成如图 4-12 所示的程序。

最后按 ESC 键，就可以确认修改并退出设置界面。

图 4-12 比较合理的登录对话框的 tab 顺序

|  |
| --- |
| 技巧：调用 CWinApp::SetDialogBkColor()可以改变对话框的背景颜色和文本颜色。该函数一般在应用程序类的 InitInstance 函数中调用，设置应用程序中对话框与消息对话框的默认背景颜色和文本颜色。例如：  BOOL CMyApp::InitInstance()  {  // Standard initialization  ……  //将对话框背景颜色设置为黄色（第一个参数） //将对话框文本颜色设置为蓝色（第二个参数） |

SetDialogBkColor(RGB(255,255,0),RGB(0,0,255));

//装入标准的 INI 文件设置（包括 MRU）

LoadStdProfileSettings();

……

}

### 4.1.3 创建对话框类

创建对话框的第二步是创建对话框类，即创建一个 CDialog 类的派生类与新建的对话框

资源关联。对话框类 CDialog 提供了访问控件属性，以及响应控件和对话框自身消息的功能。

1．创建对话框类

创建与关联的对话框类的派生类的过程如下：

1. 在 IDE 主菜单栏中依次选择菜单“View|ClassWizard”，或者在开发环境的界面中直接使用快捷键 Ctrl+W，打开“ClassWizard”对话框。同时系统会检测到添加了新的对话框资源，并自动打开如图 4-13 所示的对话框，提示用户是否为新建的对话框添加一个对话框类。
2. 在如图 4-13 所示的对话框中，选中“Create a new class”单选项，为对话框资源创建一个新的类，单击“OK”按钮确认后打开如图 4-14 所示的“New Class”对话框。

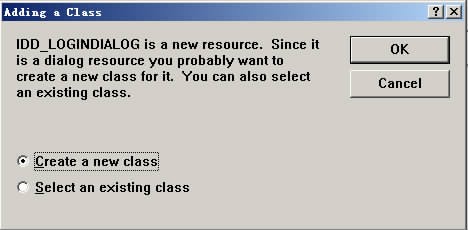
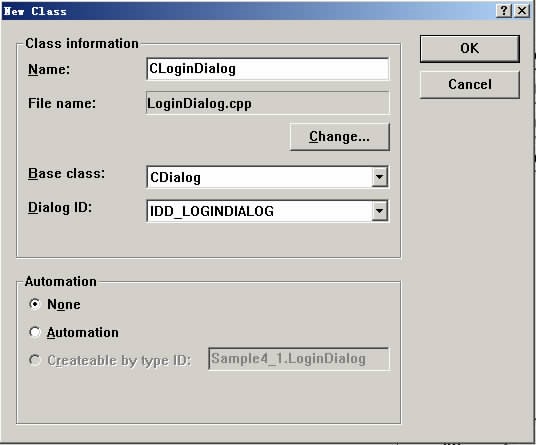
 

图 4-13 提示用户为新添加的对话框创建一个对话框类 图 4-14 创建新类的对话框

1. 在“New Class”对话框中，为 4.1.2 节中创建的实例 Sample4\_1 的登录对话框创建新的类。

* 在“Name”输入框中输入“CLoginDialog”，表明新建类的名称为 CLoginDialog。
* 此时“File name”输入框中的内容自动设置为“LoginDialog.cpp”，表明类的源文件为

LoginDialog.cpp。

* 在“Base class”下拉列表框中选择“CDialog”，表明 CLoginDialog 类的基类为 CDialog。
* 在“Dialog ID”下拉列表框中选择 IDD\_LOGINDIALOG，表明 CLoginDialog 关联的对话框资源为 IDD\_LOGINDIALOG。
* 单击“OK”按钮后，关闭 New Class 对话框，返回 ClassWizard 对话框，同时 ClassWizard 为登录对话框创建了一个新的类 CLoginDialog。

2．为对话框类添加控件成员变量

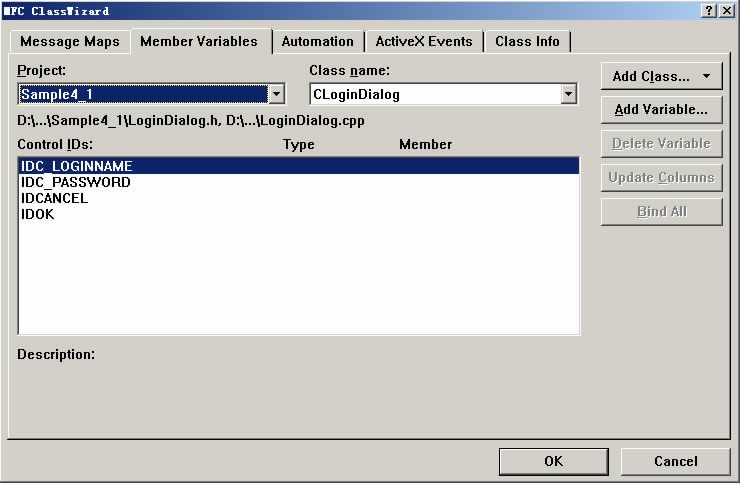
在返回的 ClassWizard 对话框中单击“Member Variables”选项卡进入如图 4-15 所示的编辑对话框控件成员变量的界面。这个界面用于设置为控件关联变量，使得在程序中通过这些变量可以访问控件的属性。

介绍该界面中的主要内容如下。

* Project 下拉列表框：选择需要编辑的工程名。
* Class name 下拉列表框：选择需要编辑的类名。
* Control IDs 列表框：列出对话框的控件及其对应的控件成员变量信息。其中有 3 列，

“Control IDs”列给出对话框的控件 ID，“Type”列给出控件变量的类型，“Member” 列给出控件变量的名称。选定某个控件，还可以编辑该控件对应的变量。

* Add Class 按钮：添加新类。
* Add Variable 按钮：为选定的控件添加变量。
* Delete Variable 按钮：为选定的控件删除原来的变量。
* Update Columns 和 Bind All 按钮：跟数据库有关，这里不作介绍。

 图 4-15 ClassWizard 编辑对话框控件成员变量的界面

在如图 4-15 所示的“Project”下拉列表框中已经默认选择 Sample4\_1，在“Class name” 下拉列表框中选择 CLoginDialog，从 Control IDs 列表框中可以看到登录对话框中有 4 个控件，

IDC\_LOGINNAME、IDC\_PASSWORD、IDCANCEL 和 IDOK。需要获得用户输入的用户名和密码，因此需要为 IDC\_LOGINNAME 和 IDC\_PASSWORD 添加两个变量。步骤如下：

1. 在如图 4-15 所示的对话框的“Control IDs”列表框中选择 IDC\_LOGINNAME，然后单击“Add Variable”按钮，打开如图 4-16 所示的添加成员变量对话框。
2. 按如下方法设置 IDC\_LOGINNAME 的成员变量。

* 在“Member variable name”输入框中输入 m\_szLoginName，即将变量命名为 m\_szLoginName。



图 4-16 添加成员变量对话框

* 在“Category”下拉列表框中选择“Value”，表明创建的变量是值，而不是控件。
* 在“Variable type”下拉列表框中选择 CString，表明变量类型为 CString。

单击“OK”按钮，确认并返回 ClassWizard 对话框。

（3）添加一个 m\_szPassword 变量，对话框设置如下。

* 在“ Member variable name ”输入框中输入 m\_szPassword ，即将变量命名为 m\_szPassword；
* 在“Category”下拉列表框中选择“Value”，表明创建的变量是值，而不是控件；
* 在“Variable type”下拉列表框中选择“CString”，表明变量类型为 CString。

单击“OK”按钮，确认并返回 ClassWizard 对话框，如图 4-17 所示。

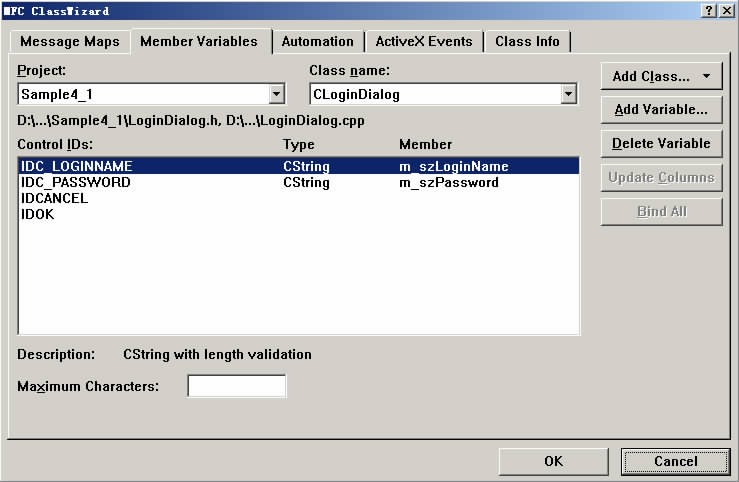


图 4-17 完成变量添加的编辑成员变量的界面

1. 此时可以看到，在“Control IDs”列表框中选择“IDC\_LOGINNAME”项后，在对话框的下方将出现该变量的属性介绍和说明。对于 CString 变量，还有一个“Maximum Characters”输入框，用于设置 C\_LOGINNAME 输入框中的输入字符的个数。在本实例中可以不作这个限定，故此输入框设置为空。
2. 单击“ClassWizard”对话框中的“OK”按钮后，回到 IDE 主界面，完成变量的添加。

此时在 IDE 左侧单击“ClassView”选项卡，可以看到新建了一个类 CLoginDialog，展开

CLoginDialog 节点，可以看到该类中添加了 m\_szLoginName 和 m\_szPassword 两个控件变量。

3．数据交换和检验

程序运行时，用户更改控件属性，如在输入框中输入文字或者改变复选框的选中状况后，

MFC 会修改对应控件变量的值。这个过程是通过 MFC 为对话框类自动添加的成员函数

DoDataExchange()实现的，称为对话框数据交换和检验机制。该函数在 LoginDialog.cpp 中的代码如下：

void CLoginDialog::DoDataExchange(CDataExchange\* pDX)

{

CDialog::DoDataExchange(pDX); //处理 MFC 默认的数据交换

//下面的代码处理控件与控件变量之间的数据交换，一般由 MFC 自动添加

//{{AFX\_DATA\_MAP(CLoginDialog)

//处理控件 IDC\_LOGINNAME 和变量 m\_szLoginName 之间的数据交换

//DDX\_Text 表明是进行文本数据交换

DDX\_Text(pDX, IDC\_LOGINNAME, m\_szLoginName);

//处理控件 IDC\_PASSWORD 和变量 m\_szPassword 之间的数据交换

DDX\_Text(pDX, IDC\_PASSWORD, m\_szPassword);

//}}AFX\_DATA\_MAP

}

MFC 提供了处理控件与控件变量之间的数据交换的机制，因此在代码中访问控件的属性，就可以直接通过访问控件变量进行。如实例 Sample4\_1 运行时，打开登录对话框，在“用户名”输入框中输入字符串，通过 CLoginDialog::DoDataExchange()函数，m\_szLoginName 就自动保存所有输入的字符串值，反之如果程序运行时修改了 m\_szLoginName，则通过

CLoginDialog::DoDataExchange()函数，“用户名”输入框就会显示 m\_szLoginName 的值。

这里需要注意，控件和控件变量之间的数据交换并非自动完成的，即 DoDataExchange() 函数不是自动被调用的。例如在“用户名”输入框中输入字符串时，m\_szLoginName 的值并不随之改变，需要调用 CDialog::UpdateData()函数才能做到从控件到控件变量或者从控件变量到控件的数据传送。

CDialog::UpdateData()函数的原型为：

BOOL UpdateData(

BOOL bSaveAndValidate = TRUE

);

bSaveAndValidate 参数用于指示数据传输的方向，TRUE 表示从控件传给变量，FALSE

表示从变量传给数据。

CDialog::UpdateData()函数的返回值用于确定操作是否成功，为 FALSE 表示不成功。在实例 Sample4\_1 中，如果要通过控件变量访问“用户名”和“密码”两个输入框的输入值，可以用如下代码：

|  |  |  |
| --- | --- | --- |
| UpdateData();  //验证用户名和密码 |  | //参数默认值为 TRUE，用于保存对话框的控件值 |

if (m\_szLoginName == "user" && m\_szPassword == "test1234")

{

//验证通过，并提示用户

⋯ ⋯

}

如果要设置“用户名”和“密码”两个输入框的值，则可以用如下代码：

|  |
| --- |
| //清除对话框中的输入  m\_szLoginName = ""; m\_szPassword = "";  //根据控件变量设置对话框的控件的值  UpdateData(FALSE); |

4．为对话框类添加控件通知消息的处理函数

MFC 为对话框和控件定义了许多消息，可以通过“ClassWizard”对话框来查看、新建和删除相应的消息响应函数。下面在实例 Sample4\_1 的登录对话框中添加单击“确认”和“取消”两个按钮的消息处理函数，步骤如下：

1. 在 IDE 主菜单栏中依次选择菜单“View|ClassWizard”，或者在开发环境的界面下直接使用快捷键 Ctrl+W，打开“ClassWizard”对话框，选择“Message Maps”选项卡，在“Project” 下拉列表框中选择 Sample4\_1，在“Class name”下拉列表框中选择“CLoginDialog”。

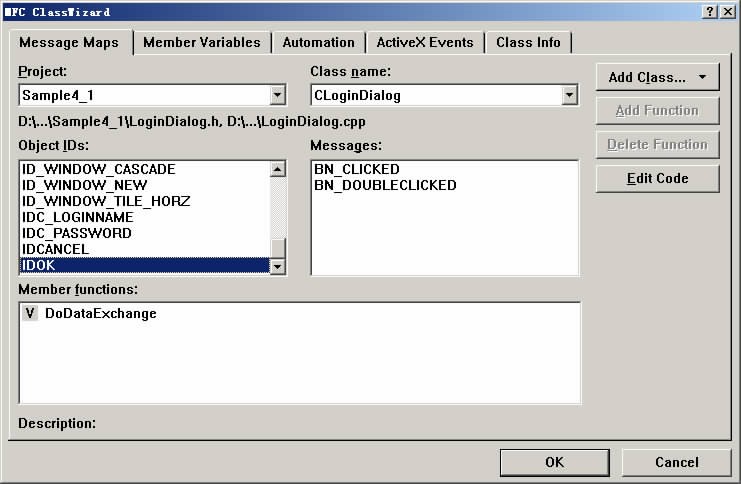


图 4-18 控件 IDOK 的通知消息

1. 在“Object IDs”列表框中选择 IDOK，对应的控件是“确认”按钮，按钮一般有两种通知消息，分别是 BN\_CLICKED（按钮被单击）和 BN\_DOUBLECLICKED（按钮被双击）。（3）要为对话框添加用户单击“确认”按钮后的验证过程，需要添加一个处理 IDOK 的

BN\_CLICKED 通知消息成员函数。在“Messages”列表框中选择 BN\_CLICKED，单击“Add

Function”按钮，或直接双击“Messages”列表框中的 BN\_CLICKED，弹出 Add member function

对话框，单击“OK”按钮就可以创建一个名称为 OnOK 的消息处理函数。单击“Edit Code” 按钮，退出 ClassWizard，并自动打开 LoginDialog.cpp 文件，且定位在 OnOK()函数上。为 OnOK()函数添加如下代码：

void CLoginDialog::OnOK()

|  |  |
| --- | --- |
| {                } | // TODO: Add extra validation here  //以下添加用户名和密码的验证过程  //注意，在对话框的 OnOK 函数中自动调用 CDialog::UpdateData(TRUE)，  //即自动更新控件变量的值  //本例为简单起见，合法的用户名只能是 user，对应密码为 test1234  if (m\_szLoginName == "user" && m\_szPassword == "test1234")  {  //验证通过，并提示用户  //MessageBox 是 Windows 系统的消息对话框，用于提示简单的消息，  //具体内容可以参考本章的最后一节  MessageBox("用户验证通过", "登录提示",MB\_OK);  } else  {  //验证不通过  //提示用户  MessageBox("用户验证失败", "登录提示",MB\_OK);  //退出主程序  AfxGetMainWnd()->PostMessage(WM\_QUIT);  //返回登录失败  //EndDialog 退出对话框，参数表示对话框的返回值，本例  //用"取消"按钮的 ID 作为验证失败返回值  EndDialog(IDCANCEL);  }  //默认的消息处理函数，用于退出对话框，返回值为 IDOK  CDialog::OnOK(); |

|  |
| --- |
| 技巧：一般要退出程序，只需要销毁该程序的主窗口，即向该程序的主窗口发送 WM\_QUIT 消息，  代码为 AfxGetMainWnd()->PostMessage(WM\_QUIT);。 |

（4）按前面所述方法，打开“ClassWizard”对话框，为 CLoginDialog 的 IDCANCEL 按钮控件的 BN\_CLICKED 消息添加消息处理函数 OnCancel()。在程序中单击“取消”按钮与验证失败的处理方法相同，因此修改 CLoginDialog::OnCancel()函数，代码如下：

void CLoginDialog::OnCancel()

{

// TODO: Add extra cleanup here

//验证不通过

//提示用户

|  |  |
| --- | --- |
| } | MessageBox("用户验证失败", "登录提示",MB\_OK);  //退出主程序  AfxGetMainWnd()->PostMessage(WM\_QUIT);  //默认的消息处理函数，用于退出对话框，返回值为 IDCANCEL，  //同本例的 OnOK 验证失败的返回值  CDialog::OnCancel(); |

添加控件通知消息处理函数后，对话框类的创建任务基本完成，对话框类能基本实现对话框所需的功能。

## 4.2 模态对话框

Windows 系统中对话框大致可以分为以下两种：

* 模态对话框。这类对话框弹出后，其他程序会被挂起，只有当前对话框响应用户的输入。模态对话框一般要求用户做出某种选择。
* 非模态对话框。打开非模态对话框后，程序的其他窗口仍能响应消息，程序仍可以继续运行。非模态对话框一般用来显示信息，或者实时地进行一些设置。

实例 4-2：创建模态对话框实例。源代码在光盘中“\04\实例 4-2\Sample4\_1”目录下。

本节还以实例 Sample4\_1 为例，具体讲解模态对话框的创建步骤。

1．创建对话框类

按照 4.1 节所述的方法创建对话框类和对话框资源，Sample4\_1 的对话框资源和对话框类在前面已经创建，这里不再详述。

2．创建对话框类的实例并初始化

首先创建对话框类的对象。在 Sample4\_1 中，为了在程序运行最开始完成登录功能，可以在主窗口创建时显示登录对话框。创建 CLoginDialog 实例的过程如下：

1. 在 IDE 左侧工作区，单击“FileView”选项卡，展开 Sample4\_1 files 节点，然后展开 Source Files 节点，双击其中的 MainFrm.cpp 文件，进入 MainFrm.cpp 编辑界面，在文件头部其他#include 语句加入下面的包含语句：

#include "LoginDialog.h"

1. 在 IDE 左侧工作区，单击“ClassView”选项卡，展开 CMainFrame 节点，双击其中的 OnCreate()函数，就可以在右侧客户区编辑 CMainFrame::OnCreate()函数。OnCreate()函数是 CWnd 中窗口创建消息 WM\_CREATE 的处理函数，在使用 AppWizard 生成的 SDI 和 MDI 程序中，CMainFrame 类已经默认添加了 OnCreate()函数。在该函数中添加模态登录对话框的使用代码。
2. 创建 CLoginDialog 对话框类的对象实例。在 CMainFrame::OnCreate()函数最后的 return 语句前加入如下代码：

|  |  |  |
| --- | --- | --- |
| CLoginDialog dlg; |  | //创建 CLoginDialog 的对象 |

其次在创建对象后，可以加入其他代码，对对象的一些成员变量进行设置。

3．运行模态对话框并获得返回值

创建对话框类的实例后就可以通过 CDialog::DoModal()函数运行模态对话框，并得到对话框的返回值。在上一步中创建对象的代码后继续添加如下代码：

|  |
| --- |
| //运行模态对话框并获得返回值 //如果返回值为 IDCANCEL 则退出程序  if (dlg.DoModal()==IDCANCEL)  return -1; //返回值为-1，告诉 MFC 销毁窗口 |

CDialog::DoModal()函数的原形如下：

virtual INT\_PTR DoModal( );

该函数根据对话框类对应的模板打开一个对话框，并挂起非该对话框的用户输入权，直到该对话框关闭。返回值是通过 CDialog::EndDialog()函数设置的，该函数用于关闭对话框，并将惟一的参数作为 DoModa()函数的返回值供调用窗口访问。

在 4.1.3 节第 4 部分的消息处理函数代码中，CLoginDialog::OnOK()验证用户名和密码失败时，就调用了 EndDialog()函数，并设置参数为 IDCANCEL。因此如果验证失败，DoModal 函数就会得到 IDCANCEL 的返回值，之后程序退出。到此，整个实例的功能已经实现。

运行程序，首先看到如图 4-19 所示的登录对话框，提示程序用户输入用户名和密码，如果输入错误，则退出程序，如果输入正确，则打开主窗口。



图 4-19 登录对话框的运行效果

此外，对话框中的控件变量，在程序中可以直接访问。DoModal()结束后，控件变量的值就是对话框结束前的最终结果。例如在实例 Sample4\_1 的 CMainFrame::OnCreate()函数中，访问 dlg.m\_szLoginName 就可以获取用户的登录名。

## 4.3 非模态对话框

本节将介绍另一类型的对话框，即非模态对话框的使用。这类对话框在没有关闭前，也能允许其他应用程序继续运行，并可以在应用程序间互相切换窗口。因此此类对话框的生存周期较长，也就是说非模态对话框的对象生存周期需要存在于父窗口类对象的整个生存周期。

因此需要在父窗口类添加指向非模态对话框类对象的指针成员变量。

本节介绍如何为实例 Sample4\_1 添加一个非模态对话框。

实例 4-3：创建非模态对话框实例。源代码在光盘中“\04\实例 4-3\Sample4\_1”目录下。

1．创建对话框类

在创建对话框资源和对话框类时，没有模态和非模态的区别。因此创建非模态对话框类

与创建模态对话框的方法相同，具体步骤如下：

1. 在资源编辑器中为 Sample4\_1 创建一个新的对话框资源，将标志号 ID 设置为

IDD\_MODALLESS，并添加控件。

1. 用 ClassWizard 创建与其相关联的对话框类，类名为 CDialogModalless，头文件和源文件分别为 DialogModalless.h 和 DialogModalless.cpp。
2. 在 ClassWizard 中为 CDialogModalless 添加控件成员变量和消息处理函数。完成对话框类的创建。 2．创建对话框类的实例并初始化

在实例 Sample4\_1 中，通过选择菜单项打开 IDD\_MODALLESS 非模态对话框，因此需要在视图类 CSample4\_1View 中添加一个 CDialogModalless 类的指针成员变量，步骤如下：

1. 单击选中 ClassView 选项卡，在 CSample4\_1View 节点上单击鼠标右键，选择 Add

Member Variable 菜单项。

1. 在弹出的对话框中向“Variable Type”输入框输入“CDialogModalless \*”，在“Variable Declaration”输入框中输入 m\_pModallessDlg，单击“OK”按钮。
2. 展开 ClassView 选项卡中 CSample4\_View 节点，可以看见其中包含 m\_pModallessDlg 项，说明添加成员变量成功。
3. 双击 ClassView 选项卡中的 CSample4\_1View 节点，打开 CSample4\_1View.h 文件，在其头部添加包含语句，代码如下：

⋯ ⋯

#if \_MSC\_VER > 1000

#pragma once

#endif // \_MSC\_VER > 1000

#include "DialogModalless.h"

class CSample4\_1View : public CView

{

⋯ ⋯

1. 在 IDE 左侧 ClassView 选项卡中展开 CSample4\_1View，双击其中的 CSample4\_1View ()，打开 CSample4\_1View.cpp 文件，并定位到构造函数 CSample4\_1View:: CSample4\_1View()，在构造函数中添加变量 m\_pModallessDlg 的初始化代码，具体如下：

CSample4\_1View::CSample4\_1View()

{

// TODO: add construction code here

m\_pModallessDlg = NULL; //初始化指针

}

3．创建和显示对话框

实例 Sample4\_1 中的非模态对话框通过选择菜单项打开，下面讲解如何实现此功能。（1）首先创建打开非模态对话框的菜单项。

单击 IDE 左侧工作区的“ResourceView”标签，展开 Menu 节点，双击 IDR\_SAMPLETYPE，

打开 IDR\_SAMPLETYPE 菜单资源编辑界面。在 IDE 右侧客户区展开顶层“查看”菜单，双击虚线框，打开添加一个新的菜单项的对话框。在“ID”输入框中输入 ID\_MODALLESS，在“Caption”输入框中输入“打开非模态对话框”，回车确认修改。

（2）为新建的菜单项添加消息处理函数。

在 IDE 主菜单栏中依次选择菜单“View|ClassWizard”或者在开发环境的界面中直接使

用快捷键 Ctrl+W，打开“ClassWizard”对话框。单击“Message Maps”选项卡，在“Class name” 下拉列表框中选择“CSample4\_1View”，在“Object IDs”列表框中选择 ID\_MODALLESS，在“Messages”列表框中选择 COMMAND，单击“Add Function”按钮，在弹出的对话框中单击“OK”按钮，就创建了名为 OnModalless 的菜单项消息处理函数。

（ 3 ）单击“ Edit Function ”按钮，关闭“ ClassWizard ”对话框， IDE 定位到

CSample4\_1View::OnModalless()函数，编辑 CSample4\_1View::OnModalless()函数，代码如下：

void CSample4\_1View::OnModalless()

{

// TODO: Add your command handler code here

//若指针为 NULL，则对话框未被创建，需要动态创建

if (m\_pModallessDlg==NULL)

{

//创建非模态对话对话框

m\_pModallessDlg=new CDialogModalless(); m\_pModallessDlg->Create(IDD\_MODALLESS);

}

//显示非模态对话框

m\_pModallessDlg->ShowWindow(TRUE);

}

4．退出和销毁对话框

当程序退出时需要销毁非模态对话框，在视图类 CSample4\_1View 的析构函数中删除动

态创建的非模态对话框对象即可完成非模态对话框的销毁。

在 IDE 左侧工作区单击“ClassView”选项卡，展开 CSample4\_1View 节点，双击

~CSample4\_1View()，即可定位到析构函数 CSample4\_1View::~CSample4\_1View()，修改该函数，代码如下：

void CSample4\_1View::~CSample4\_1View()

{

if (m\_pModallessDlg!=NULL)

{

|  |  |  |
| --- | --- | --- |
|  |  | //删除对象  delete m\_pModallessDlg; |
| } | } |  |

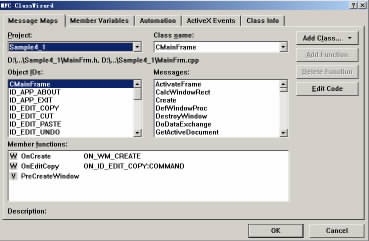
编译运行 Sample4\_1，依次选择菜单“查看|打开非模态对话框”，就可以打开非模态对话框。

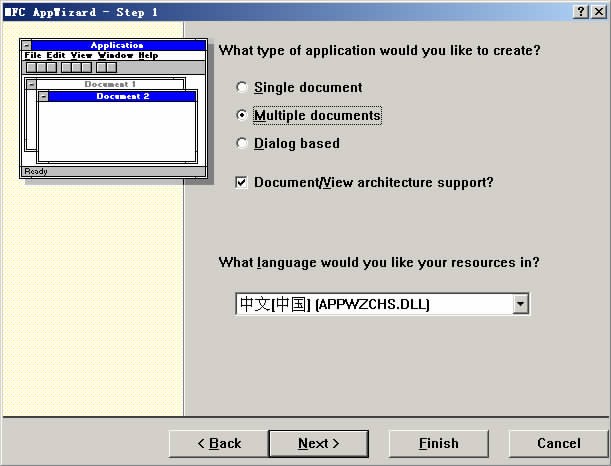
技巧：可以删除工程目录中的 .clw 文件，然后打开“ClassWizard”对话框，选择重建类向导数据库，就可以为工程手动添加类到 ClassWizard 中。

## 4.4 属性页对话框

属性页对话框是一种特殊的对话框，它将多个对话框集中起来，通过标签或按钮来激活各个页面。属性页对话框主要分为一般属性页对话框和向导对话框。在一般属性页对话框中，页面的切换通过单击不同的选项卡实现。在向导对话框中，页面的选择是通过单击“上一页”

（Back）、“下一页”（Next）按钮等按钮实现的。图 4-20 和图 4-21 分别给出了这两种对话框的界面。

 图 4-20 MFC ClassWizard 一般属性页对话框

 图 4-21 MFC AppWizard 向导对话框

### 4.4.1 主要相关类介绍

与属性页对话框相关的类主要有 CPropertyPage 类和 CPropertyPage 类。本小节分别讲述 CPropertyPage 和 CPropertySheet 类的使用方法和参数。

1．CPropertyPage 类

CPropertyPage 类用于单个属性页的处理。使用属性页对话框，必须为每个属性页创建一个继承自 CPropertyPage 类的属性页类。CPropertyPage 类是 CDialog 类的一个派生类，其主要成员如下：

（1）构造函数构造一个新的属性页类的对象，原型如下：

|  |
| --- |
| CPropertyPage( ); explicit CPropertyPage( UINT nIDTemplate,  UINT nIDCaption = 0,  DWORD dwSize = sizeof(PROPSHEETPAGE)  );  explicit CPropertyPage(  LPCTSTR lpszTemplateName,  UINT nIDCaption = 0,  DWORD dwSize = sizeof(PROPSHEETPAGE)  ); |

* nIDTemplate 参数：给出属性页对话框资源 ID。
* nIDCaption 参数：给出属性页选项卡的标题字符串的 ID，如果为 0，则默认为指定对话框的标题。
* lpszTemplate 参数：给出属性页对话框资源的名称字符串。

（2）Construct()函数

该函数用于创建一个 CPropertyPage 对象，原型如下：

void Construct(

UINT nIDTemplate,

UINT nIDCaption = 0

);

void Construct(

LPCTSTR lpszTemplateName,

UINT nIDCaption = 0

);

* nIDTemplate 参数：给出属性页对话框资源 ID。
* nIDCaption 参数：给出属性页选项卡的标题字符串的 ID，如果为 0，则默认为指定对话框的标题。
* lpszTemplate 参数：给出属性页对话框资源的名称字符串。

1. CancelToClose()函数

在模态对话框中，当一个不可恢复的过程完毕后，调用该函数将“确定”（OK）按钮改变为“关闭”（Close）按钮，同时禁用“取消”（Cancel）按钮。其原型如下：

void CancelToClose( );

1. SetModified()函数

该函数允许或禁用“应用”（Apply）按钮。其原型如下：

void SetModified(

BOOL bChanged = TRUE

);

• bChanged 参数：值为 TRUE 时“应用”按钮可用，为 FALSE 时禁用该按钮。

（5）可重载的消息处理函数

为了响应属性页对话框的按钮消息和页面切换消息，CPropertyPage 类中增加了一系列消息处理函数。重载这些函数就可以对属性页对话框中用户的操作进行处理。这些消息处理函数如表 4-1 所示。

表 4-1 属性页类的消息处理函数说明

|  |  |
| --- | --- |
| 消息处理函数 | 函数说明 |
| OnApply | 响应属性表的“应用”按钮被单击的消息 |
| OnCancel | 响应属性表的“取消”按钮被单击的消息 |
| OnKillActive | 响应该属性页从当前活动页被切换出去的消息，用于数据验证 |
| OnQueryCancel | 响应属性表的“取消”按钮被单击前发出的消息 |
| OnReset | 响应属性表的“重置”按钮被单击的消息 |
| OnSetActive | 响应该属性页切换为当前活动页的消息 |
| OnWizardBack | 响应属性表的“上一步”按钮被单击的消息，仅在向导对话框中有效 |
| OnWizardFinish | 响应属性表的“完成”按钮被单击的消息，仅在向导对话框中有效 |
| OnWizardNext | 响应属性表的“下一步”按钮被单击的消息，仅在向导对话框中有效 |

2．CPropertySheet 类

CPropertySheet 类负责对属性表进行控制。该类完成的功能有装入或删除属性页、打开属性页对话框，以及属性页对话框运行时切换属性页等功能。CPropertySheet 类是 CWnd 的派生类，但也有类似于对话框的两种运行方式，即模态和非模态。下面介绍 CPropertySheet 类中与属性表操作相关的函数。

（1）构造函数构造一个新的属性表类的对象，原型如下：

CPropertySheet( );

CPropertySheet(

UINT nIDCaption,

CWnd\* pParentWnd = NULL,

UINT iSelectPage = 0

);

CPropertySheet(

LPCTSTR pszCaption,

CWnd\* pParentWnd = NULL,

UINT iSelectPage = 0

);

* nIDCaption 参数：给出标题的字符串的 ID。
* pParentWnd 参数：给出对话框的父窗口，为 NULL 时，父窗口默认为程序主窗口。
* iSelectPage 参数：给出对话框打开时初始属性页的序号。
* pszCaption 参数：给出标题的字符串。

1. GetActiveIndex()函数

该函数返回当前活动的属性页的序号，原型如下：

int GetActiveIndex( ) const;

返回值就是当前活动的属性页的序号。

1. GetActivePage()函数

该函数返回当前活动的属性页对象，原型如下：

CPropertyPage\* GetActivePage( ) const;

返回值为当前活动的属性页对象的指针。可以通过强制类型转换，将指针类型由

CPropertyPage 变为程序中自定义的属性页类。

1. GetPage()函数

该函数返回某个属性页对象，原型如下：

CPropertyPage\* GetPage( int nPage

) const;

* nPage 参数：表示所要获取的属性页的序号，从零开始。
* 返回值：属性页对象的指针。

1. GetPageCount()函数

该函数返回属性页的总数，原型如下：

int GetPageCount( ) const;

返回值为属性页的总数。

1. GetPageIndex()函数

该函数获取属性页在对话框中的序号，原型如下：

int GetPageIndex(

CPropertyPage\* pPage

);

* pPage 参数：要获取序号的属性页对象的指针。
* 返回值：属性页对象对应的序号。

（7）SetActivePage()函数

该函数设置某个属性页为当前活动的属性页，原型如下：

BOOL SetActivePage( int nPage

);

BOOL SetActivePage(

CPropertyPage\* pPage

);

* nPage 参数：要设置为活动的属性页的序号。
* pPage 参数：要设置为活动的属性页的对象指针。
* 返回值：表示操作是否成功。

（8）SetTitle()函数

该函数设置属性表的标题，原型如下：

void SetTitle(

LPCTSTR lpszText,

UINT nStyle = 0

);

* lpszText 参数：新的标题字符串指针。
* nStyle 参数：设置标题的属性，只能为 0 或者 PSH\_PROPTITLE。取值为 0 时没有附加属性；取值为 PSH\_PROPTITLE 会在新的标题后面添加“属性”两个字，比如 lpszText 为“Text”，则当 nStyle 为 PSH\_PROPTITLE 时，真正的标题显示为“Text 属性”。

1. AddPage()函数

该函数为属性表添加新的属性页，原型如下：

void AddPage(

CPropertyPage \*pPage

);

pPage 参数为要添加的属性页对象的指针标题字符串。

1. PressButton()函数

该函数模拟单击属性表的某个按钮，原型如下：

void PressButton( int nButton

);

nButton 参数给出要模拟单击的按钮，可以取下列值中的一个。

* PSBTN\_BACK：选择“上一步”按钮。
* PSBTN\_NEXT：选择“下一步”按钮。
* PSBTN\_FINISH：选择“完成”按钮。
* PSBTN\_OK：选择“确认”按钮。
* PSBTN\_APPLYNOW：选择“应用”按钮。
* PSBTN\_CANCEL：选择“取消”按钮。
* PSBTN\_HELP：选择“帮助”按钮。

（11）RemovePage()函数

该函数为属性表删除已有的属性页，原型如下：

|  |
| --- |
| void RemovePage(  CPropertyPage \*pPage  );  void RemovePage( int nPage  ); |

* pPage 参数：要删除的属性页的对象指针。
* nPage 参数：要删除的属性页的序号。

（12）SetWizardButtons()函数

该函数进行设置向导对话框的典型按钮（比如前进、后退、完成等）。注意该函数必须在调用 DoModal()时才能使用，因此一般在属性页中的 CPropertySheet::OnSetActive()函数中调用。函数其原型为：

void SetWizardButtons( DWORD dwFlags

);

dwFlags 参数设置向导按钮的属性，可以为以下值的组合：

* PSWIZB\_BACK：显示“上一步”按钮，如果没有包含该值，“上一步”按钮被禁用。
* PSWIZB\_NEXT：显示“下一步”按钮，如果没有包含该值，“下一步”按钮被禁用。
* PSWIZB \_FINISH：显示“完成”按钮。
* PSWIZB \_DISABLEFINISH：显示被禁用的“完成”按钮。

1. SetWizardMode ()函数

该函数设置属性页对话框为向导对话框模式，应当在调用 DoModal()函数之前调用该函数。其原型为：

void SetWizardMode( );

1. DoModal ()函数

该函数显示一个模态属性页对话框，其原型为：

virtual INT\_PTR DoModal( );

返回值：返回对话框关闭的状态，对于一般属性页，返回值一般为 IDOK、IDCANCEL 或者 0；对于向导对话框返回值为 ID\_WIZFINISH 或 IDCANCEL。

### 4.4.2 向导对话框的创建步骤

向导对话框用于完成一系列设置步骤，每一步通过“上一步”和“下一步”按钮进行切换。本节将在实例 Sample4\_1 的基础上创建一个有 3 个步骤（包含 3 个属性页）的向导对话框。

实例 4-4：创建向导对话框实例。源代码在光盘中“\04\实例 4-4\Sample4\_1”目录下。

1．创建属性页模板

创建向导对话框首先要创建属性页模板，步骤如下：

1. 打开实例 Sample4\_1 的工程后，在 IDE 中依次选择菜单“Insert|Resource”，打开“Insert Resource”对话框。在“Resource type”属性列表框中选择 Dialog 选项，单击“New”按钮，创建一个默认 ID 为 IDD\_DIALOG1 的新的对话框资源。
2. 在 IDE 左侧工作区中单击“ResourceView”选项卡，展开“Sample4\_1 resources|Dialog”，找到 IDD\_DIALOG1，在其上单击鼠标右键，选择菜单“Properties”，打开对话框资源属性设置对话框。在“ID 输入框”中输入 IDD\_TESTPAGE1，即设置对话框资源 ID 为

IDD\_TESTDPAGE1，回车关闭属性设置对话框并确认修改。

1. 在 IDE 左侧“ResourceView”选项卡中，双击刚刚修改过的 IDD\_TESTPAGE1，打开 IDD\_TESTPAGE1 的对话框资源编辑器，在 IDE 右侧客户区的对话框上单击鼠标右键，选择菜单项“Properties”，打开该对话框的属性设置对话框。
2. 在对话框的属性设置对话框中设置对话框的标题。单击“General”选项卡，在“Caption” 输入框中输入“测试页 1”。
3. 仍在步骤 4 中对话框的属性设置对话框中，设置对话框的样式为属性页对话框的样式。单击“Styles”选项卡，如图 4-22 所示。在“Style”下拉列表框中选择 Child，在“Border” 下拉列表框中选择“Thin”，选中“Title bar”复选框，清除其他复选框的选项，如图 4-22 所示。回车确认设置并关闭此对话框。
4. 按照 4.1.2 节第 3 部分的讲解，删除 IDD\_TESTPAGE1 中的“确认”（OK）按钮和

“取消”（Cancel）按钮，并为 IDD\_TESTPAGE1 添加合适的控件。完成控件的添加后，

IDD\_TESTPAGE1 就是要创建的属性页对话框的第一个属性页模板。

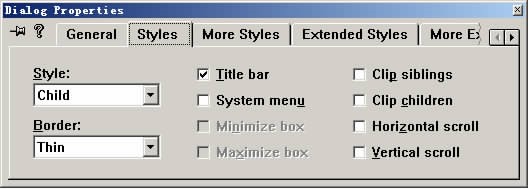


图 4-22 属性页模板的风格设置

2．为属性页添加属性页类

创建完属性页模板后，需要用 ClassWizard 为每个属性页添加 CPropertyPage 的派生类，并添加相应的控件变量和消息处理函数，以及添加属性表按钮被单击的消息处理函数。其步骤如下：

1. 在 IDE 主菜单栏中依次选择菜单“View|ClassWizard”，或者在开发环境的界面下直接使用快捷键 Ctrl+W，打开“ClassWizard”对话框。同时它会检测到添加了新的对话框资源，并打开“Adding a Class”对话框，提示用户是否为新建的对话框添加一个对话框类。
2. 在“Adding a Class”对话框中，选中“Create a new class”单选框，意思是要为对话框资源创建一个新的类，单击“OK”按钮确认后打开“New Class”对话框。
3. 在“New Class”对话框中，ClassWizard 要为新添加的 IDD\_TESTPAGE1 模板创建一个新的类。在“Name”输入框中输入 CTestPage1，在“Base class”下拉列表框中选择 CPropertyPage，如图 4-23 所示。单击“OK”按钮关闭“New Class”对话框，返回“ClassWizard” 对话框，就可以看见“ClassWizard”对话框中的“Class name”下拉列表框中已经自动选择了新创建的类 CTestPage1。

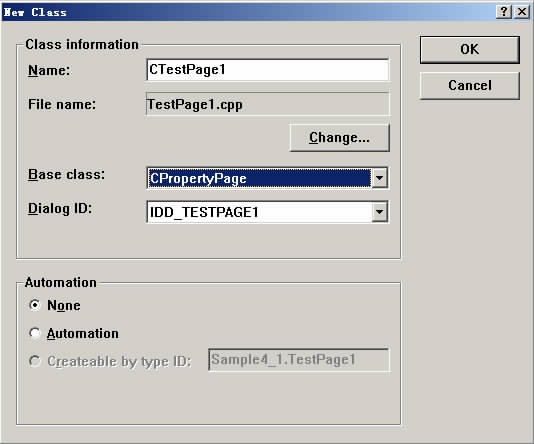


图 4-23 创建属性页类的对话框

1. 依照 4.1.3 第 2 到第 4 部分的讲解，为 CTestPage1 添加合适的控件变量和控件消息处理函数。
2. 为 CTestPage1 添加合适的属性页消息处理函数。在已经打开的“ClassWizard”对话框中，单击“Message Maps”选项卡，在“Class name”下拉列表框中选择 CTestPage1，在 “Object IDs”列表框中选择 CTestPage1，在“Messages”列表框中选择表 4-1 中出现的合适的消息处理函数，单击“Add Function”按钮，就可以添加对应的消息处理函数。

在实例 Sample4\_1 中，由于 CTestPage1 作为向导对话框的第一个属性页，因此显示测试页 1 时向导对话框中只有“下一个”（Next）按钮。这就需要为 CTestPage1 添加 OnSetActive() 函数。在“Messages”列表框中选择 OnSetActive，单击“Add Function”按钮添加该函数。再单击“ Edit Function ”按钮，关闭“ ClassWizard ”对话框， IDE 自动定位到

CTestPage1::OnSetActive()函数，修改函数代码如下：

BOOL CTestPage1::OnSetActive()

{

// TODO: Add your specialized code here and/or call the base class

// 获得父窗口，即属性表 CPropertySheet 类

CPropertySheet\* psheet = (CPropertySheet\*) GetParent();

// 设置属性表只有“下一步”按钮

psheet->SetWizardButtons(PSWIZB\_NEXT);

return CPropertyPage::OnSetActive();

}

经过上述步骤，就为属性页 IDD\_TESTPAGE1 添加了对应的属性页类 CTestPage1。 3．创建其余的属性页和对应的属性页类

为实例 Sample4\_1 创建其余的属性页，包括第二页 IDD\_TESTPAGE2 和最后一页

IDD\_TESTPAGE3，对应的属性页类为 CTestPage2 和 CTestPage3。步骤如下：

1. 参照第 1 部分，为实例 Sample4\_1 创建第二个属性页模板 IDD\_TESTPAGE2。
2. 参照第 2 部分，为 IDD\_TESTPAGE2 创建属性页类 CTestPage2，为 CTestPage2 添加合适的控件变量和控件消息处理函数，注意在 CTestPage2 中不用添加 OnSetActive()消息处理函数。如果用户想为向导对话框添加更多的中间步骤，可以重复步骤 1 和 2。
3. 参照第 1 部分，为实例 Sample4\_1 创建最后一个属性页模板 IDD\_TESTPAGE3。
4. 参照第 2 部分，为 IDD\_TESTPAGE3 创建属性页类 CTestPage3，为 CTestPage3 添加合适的控件变量和控件消息处理函数。注意在 CTestPage3 中需要添加 OnSetActive()和

OnWizardFinish()消息处理函数。

添加 CTestPage3::OnSetActive()函数，使得向导对话框进行到最后一页时只有“完成”

（Finish）按钮。在已经打开的 ClassWizard 对话框中，单击“Message Maps”选项卡，在“Class name”下拉列表框中选择 CTestPage1，在“Object IDs”列表框中选择 CTestPage3，在“Messages” 列表框中选择 OnSetActive，单击“Add Function”按钮添加该函数。再单击“Edit Function” 按钮，关闭“ClassWizard”对话框，IDE 自动定位到 CTestPage3::OnSetActive()函数，修改函数代码如下：

BOOL CTestPage3::OnSetActive()

{

// TODO: Add your specialized code here and/or call the base class

CPropertySheet\* psheet = (CPropertySheet\*) GetParent();

//设置属性表只有“完成”按钮

psheet->SetFinishText("完成");

return CPropertyPage::OnSetActive();

}

重 复 前 面 的 过 程 ， 在 “ Messages ” 列 表 框 中 选 择 OnWizardFinish ， 添 加

CTestPage3::OnWizardFinish()函数，使得完成向导对话框时提示用户，添加的函数代码如下：

BOOL CTestPage3::OnWizardFinish()

{

// TODO: Add your specialized code here and/or call the base class

MessageBox("您已经完成向导！");

return CPropertyPage::OnWizardFinish();

}

这样就完成了向导对话框的所有属性页模板和相应属性页类的创建。

4．创建属性表类创建完向导对话框的所有属性页类后，就可以为实例Sample4\_1添加属性表CPropertySheet 类的派生类，用于操作向导对话框。其步骤如下：

1. 在 IDE 的主菜单中选择“View| ClassWizard”菜单命令，或按快捷键 Ctrl+W 打开

ClassWizard 对话框。

1. 在“ClassWizard”对话框中，单击“Add Class”按钮，在弹出的下拉菜单中选择“New” 菜单项，打开“New Class”对话框。
2. 在“NewClass”对话框中创建名为 CTestSheet 的属性表派生类。在“Class type”下拉列表框中选择 MFC Class，在“Name”输入框中输入 CTestSheet，在“Base class”下拉列表框中选择 CPropertySheet，如图 4-24 所示。单击“OK”按钮退出“New Class”对话框，返回“ClassWizard”对话框，可以看到“Class name”下拉列表框中已经选择了新创建的

“CTestSheet”类。

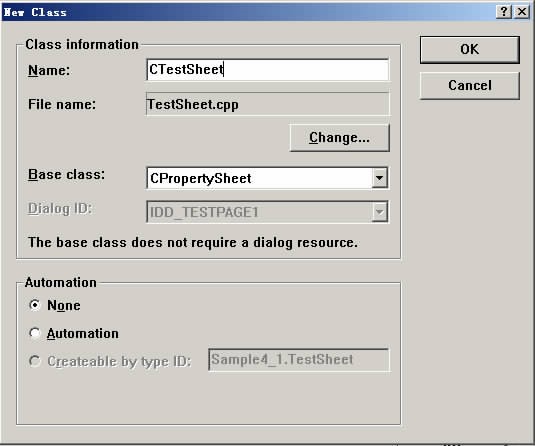


图 4-24 添加 CProperySheet 类的派生类

1. 为 CTestSheet 类添加每个属性页类的成员变量。关闭“ClassWizard”对话框，在 IDE 左侧工作区中单击“ClassView”选项卡，展开 Sample4\_1 classes 节点，在 CTestSheet 上单击鼠标右键，选择“Add Member Variable”菜单项，打开 Add Member Variable 对话框。在“Variable Type”输入框中输入 CTestPage1，在“Variable Name”输入框中输入 m\_page1，单击“OK” 按钮就可以为 CTestSheet 类添加第一个属性页类 CTestPage1 的 m\_page1 成员变量。重复这一过程，为 CTestSheet 添加成员变量 CTestPage2 类的 m\_page2 和 CTestPage3 类的 m\_page3。
2. 最后在属性表类 CTestSheet 的构造函数中将所有属性页添加到属性表中。在 IDE 左侧工作区中单击“ClassView”选项卡，展开 Sample4\_1 classes|CTestSheet 节点，双击

“CTestSheet(LPCTSTR pszCaption, CWnd\* pParentWnd = NULL, UINT iSelectPage = 0)”，即可编辑 CTestSheet 的构造函数，修改函数代码如下：

|  |
| --- |
| CTestSheet::CTestSheet(LPCTSTR pszCaption, CWnd\* pParentWnd, UINT iSelectPage)  :CPropertySheet(pszCaption, pParentWnd, iSelectPage)  {  AddPage(&m\_page1); AddPage(&m\_page2); AddPage(&m\_page3);  } |

5．打开向导对话框

在实例 Sample4\_1 中添加一个菜单项打开前面创建的向导对话框，步骤如下：（1）首先创建打开向导对话框的菜单项。

单击 IDE 左侧工作区的“ResourceView”标签，展开 Menu 节点，双击 IDR\_SAMPLETYPE，打开“IDR\_SAMPLETYPE”菜单资源编辑界面。在 IDE 右侧客户区展开顶层“查看”菜单，双击虚线框，打开添加一个新的菜单项的对话框。在“ID”输入框中输入 ID\_WIZARD，在 “Caption”输入框中输入“打开向导对话框”，回车确认修改。

（2）为新建的菜单项添加消息处理函数。

在 IDE 主菜单栏中依次选择菜单“View|ClassWizard”，或者在开发环境的界面中直接使

用快捷键 Ctrl+W，打开“ClassWizard”对话框。单击“Message Maps”选项卡，在“Class name” 下拉列表框中选择“CSample4\_1View”，在“Object IDs”列表框中选择 ID\_WIZARD，在 “Messages”列表框中选择 COMMAND，单击“Add Function”按钮，在弹出的对话框中单击 “OK”按钮，就创建了名为 OnWizard 的菜单项消息处理函数。

（ 3 ）单击“ Edit Function ”按钮，关闭“ ClassWizard ”对话框， IDE 定位到

CSample4\_1View::OnWizard()函数，编辑 CSample4\_1View::OnModalless()函数，代码如下：

void CSample4\_1View::OnWizard()

{

// TODO: Add your command handler code here

// 创建属性表对象

CTestSheet sheet("");

// 设置属性对话框为向导对话框

sheet.SetWizardMode();

// 打开模态向导对话框

sheet.DoModal();

}

至此，创建向导对话框的所有步骤已经完成，编译运行 Sample4\_1，选择菜单“查看|打开向导对话框”，就可以打开创建的向导对话框，图 4-25 所示为在向导对话框最后一步单击 “完成”按钮后的运行结果。

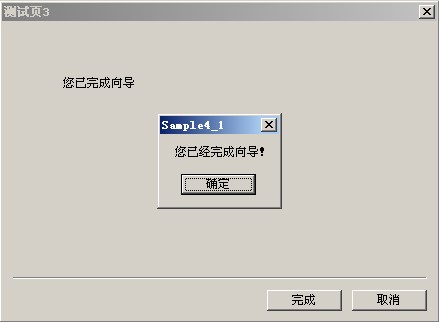


图 4-25 向导对话框的最后一步

### 4.4.3 一般属性页对话框的创建步骤

一般属性页对话框提供了选项卡的功能，为属性进行合理的分类。本节将介绍这类属性对话框的创建步骤。

1. 创建属性页模板。
2. 为各个属性页添加属性页类。
3. 创建属性表类。

以上 3 个步骤都与向导对话框的创建步骤相同。惟一的区别是不用添加 4.4.2 节中提到

的 OnSetActive()和 OnWizardFinish()函数。

1. 打开属性页对话框。

打开一般属性对话框与打开向导对话框的方法类似，只需要把打开前的 SetWizardMode() 函数去掉即可。例如在 Sample4\_1 中打开一个为 CTestSheet 类的一般属性页对话框，首先根据 4.4.2 节的第 4 部分，为 Sample4\_1 创建 ID 为 ID\_PROPERTIES 的“查看|打开一般属性页对话框”菜单项，并为之在 CSample4\_1View 类中添加消息处理函数 OnProperties()。添加如下代码就可以直接打开一般属性页对话框：

void CSample4\_1View::OnProperties()

{

// TODO: Add your command handler code here

// 创建属性表对象

CTestSheet sheet("");

// 打开模态一般属性页对话框

sheet.DoModal();

}

|  |
| --- |
| 技巧：开发大型程序时有可能会创建许多类。这种情况下可以创建目录优化工作区。例如在  Sample4\_1 中可以将 CTestSheet 及其包含的属性页类 CTestPage1、CTestPage2 和 CTestPage3  归在同一个目录下。具体操作为如下：   1. 单击在 IDE 左侧工作区窗口的“ClassView”选项卡。 2. 在“ClassView”中的 Sample4\_1 classes 节点上单击鼠标右键，选择“New Folder”菜单项，打开“New Folder”对话框。 3. 在“New Folder”对话框中的“Name of the new folder”输入框中输入 TestSheet，单击 “OK”按钮退出对话框，就可以看到“ClassView”中的 Sample4\_1 classes 节点多了一个   TestSheet 的目录。   1. 在“ClassView”中将 CtestSheet、CTestPage1、CTestPage2 和 CTestPage3 拖至 TestSheet 目录就可以完成归类。 |

## 4.5 通用对话框

在 Windows 系统中提供了一些通用对话框，如打印设置对话框、页面设置对话框、查找对话框、文件对话框、字体对话框和颜色对话框等。这些是 Windows 系统中出现频率很大的对话框，将它们定义为通用对话框可以极大地减轻程序的复杂度。同时，MFC 已经用类封装了这些通用对话框。一般来讲，要使用通用对话框，不需要派生新的类，因为 MFC 提供的基类已经提供了常用的功能。而且在对话框结束后，可以通过成员函数得到用户在对话框中的选择。

本节用一个包含各种通用对话框的例程来分别介绍 Windows 系统提供的通用对话框的编程方法。

实例 4-5：创建通用对话框实例。源代码在光盘中“\04\实例 4-5\GridDemo”目录下。

GridDemo，是一个表格编辑器。该表格编辑器提供一个固定大小的自绘表格，用户可以新建、打开和保存表格。同时该程序还提供了一些基本的编辑功能，如查找表格中出现的字符串、设置表格中文本的字体和颜色。最后该表格还可以调整打印效果，并在打印机中输出。这些功能都是用通用对话框进行设置的。

首先介绍一下 GridDemo 的基本框架。GridDemo 是一个单文档应用程序，类和主要成员变量的关系如图 4-26 所示。

* CMainFrame：GridDemo 的框架类。
* CGridDemoDocument：GridDemo 的文档类，用于存取文档并更新相应的视图。
* m\_CellTextArray：CGridDemoDocument 的成员变量，用于保存表格中的文本，是一个一维的字符串数组，通过行数 m\_nRow 和列数 m\_nCol 映射到二维的表格中。
* CGridDemoView：GridDemo 的视图类，用于显示和修改表格。表格的每个单元格在

CGridDemoView 中以输入框的控件形式出现。这个表格由一个输入框控件数组 m\_CellArray 组成，并且由这个控件数组进行绘制。

* m\_CellArray：CGridDemoView 的成员变量，用于显示和修改表格，是一个一维的

CMyEdit 类型数组，数组下标与文档类中的 m\_CellTextArray 一致。

* CMyEdit：重载的 CEdit 类，用于更改输入框文字颜色和字体。

### 4.5.1 打印设置对话框

打印设置对话框如图 4-27 所示，这个对话框提供了 Windows 标准打印设置，包括选择打印机、打印范围和打印份数等。这些选项一般在打印输出前进行设置。

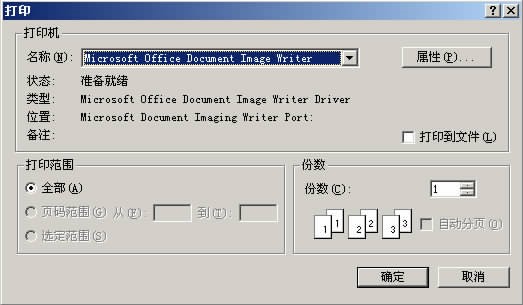


图 4-27 打印设置对话框

CPrintDialog 封装了打印设置对话框的操作。在 GridDemo 的 GridDemoView.cpp 文件中，

CGridDemoView::OnFilePrint()函数用于创建打印设置对话框，步骤如下：

1. 创建 CGridDemoView::OnFilePrint()函数。

CGridDemoView::OnFilePrint()是 GridDemo 例程视图类“文件|打印”菜单项的消息处理函数。在用 IDE 的 AppWizard 创建 SDI 应用程序的过程中，默认创建了“文件|打印”菜单项，用户只需要依照第 3 章的讲解，在 CGridDemoView 类中为该菜单项 ON\_FILE\_PRINT 添加消息处理函数，就可以创建 CGridDemoView::OnFilePrint()函数。

1. 创建 CPrintDialog 对象。CGridDemoView::OnFilePrint()函数创建 CPrintDialog 对象

dlg，代码如下：

//打开一个打印设置对话框

CPrintDialog dlg(FALSE,PD\_NOSELECTION|PD\_NOPAGENUMS,this);

1. 打开 CPrintDialog 对话框。这通过调用 CPrintDialog::DoModal()函数实现，

CGridDemoView::OnFilePrint()函数中的代码如下：

if(dlg.DoModal() == IDOK)

上面的 if 语句打开打印设置对话框，并判断如果用户是否单击“确定”按钮，如果是，

DoModal()函数返回 IDOK，进行 if 语句块中的打印操作，反之则不进行打印。

1. 将从打印设置对话框获得的打印设备上下文附加到打印输出时使用的设备上下文对

象。CGridDemoView::OnFilePrint()中这一步骤具体如下：

CDC dc; dc.Attach(dlg.GetPrinterDC());//把打印设备上下文附加到 DC 对象

1. 进行打印的具体过程，本章不涉及这类内容，故略过，用户可以参考第 14 章。
2. 结束打印。CGridDemoView::OnFilePrint()中的对应代码如下：

//结束打印

DeleteDC(dc.Detach());

### 4.5.2 页面设置对话框

页面设置对话框如图 4-28 所示，提供了 Windows 打印页面设置的标准功能，包括选择

纸张、打印方向和页边距等。CpageSetupDialog 类中封装了页面设置对话框。

下面演示 GridDemo 例程如何使用页面设置对话框进行页边距的设置：m\_rectMargin 保

存在 CGridDemoView 的成员变量中，为 CGridDemoView::OnFilePrint()的打印代码设置页边距。相关代码在 GridDemoView.cpp 文件的 CGridDemoView::OnFilePrintSetup()函数中，具体如下：

1. 创建 CGridDemoView::OnFilePrintSetup()函数。

CGridDemoView::OnFilePrintSetup()函数在 GridDemo 例程中是“文件|页面设置”菜单项的消息响应函数，对应 ID 为 ID\_FILE\_PRINT\_SETUP。由 IDE 的 AppWizard 默认创建的 SDI 应用程序中，ID\_FILE\_PRINT\_SETUP 对应的菜单项是“文件|打印设置”，只需要将菜单项的标题更改为“页面设置”即可与 GridDemo 例程一致。用户只要依照第 3 章的讲解，在

CGridDemoView 类中为该菜单项 ON\_FILE\_PRINT\_SETUP 添加消息处理函数，就可以创建

CGridDemoView::OnFilePrintSetup()函数。



图 4-28 页面设置对话框

1. 创建页面设置对话框对象，并设置初始值。这部分功能在CGridDemoView::OnFilePrintSetup() 函数的开头实现，代码如下：

//打开一个页面设置对话框，用于设置打印区域，以毫米为单位

CPageSetupDialog dlg;

//初始化页边空白区域

dlg.m\_psd.rtMargin=m\_rectMargin;

1. 打开页面设置对话框，保存程序用户设置的页面空白区域。这部分功能的实现代码在 CGridDemoView::OnFilePrintSetup()函数中具体如下：

if (dlg.DoModal()==IDOK)

{

//设置新的页边空白区域

dlg.GetMargins(&m\_rectMargin,NULL);

}

### 4.5.3 查找对话框

用户在查找字符串时，在查找对话框中可进行查找字符串的设置和其他查找选项的设置，如查找方向、是否区分大小写等，如图 4-29 所示。



图 4-29 查找对话框

查找对话框在 MFC 中封装为 CFindReplaceDialog。该对话框跟其他 Windows 通用对话框不同之处在于该对话框一般不作为模态对话框使用，而作为非模态对话框。如单击“查找下一个”按钮时，对话框会向父窗口发送消息，通知主窗口进行查找。创建查找对话框的步骤如下：

1. 添加“编辑|查找”菜单的消息处理函数 CGridDemoView::OnFind()。

GridDemo 例程中创建查找对话框的代码在 CGridDemoView::OnFind()函数中添加。 CGridDemoView::OnFind() 函数是“编辑 | 查找”菜单的消息处理函数，对应 ID 为

ID\_EDIT\_FIND。需要按照第 3 章的讲解，在“编辑”顶层菜单中添加标题为“查找”，ID 为 ID\_EDIT\_FIND 的子菜单，然后为 CGridDemoView 类添加这个 ID 的消息处理函数，即

OnFind()。

1. 为 CGridDemoView 类添加 CFindReplaceDialog 的指针成员变量。

按照 4.3 节中非模态对话框的创建过程，应当在堆中创建查找对话框的对象，因此，用户还需为 CGridDemoView 添加类型为“CFindReplaceDialog \*”的 m\_pFindDlg 成员变量。方法为在 IDE 左侧工作区单击“ClassView”选项卡，展开 GridDemo classes 节点，然后在

CGridDemoView 上单击鼠标右键，选择“Add Member Variable”菜单项，打开“Add Member Variable”对话框，在“Variable Type”栏中输入“CFindReplaceDialog \*”，在“Variable Name” 栏中输入“m\_pFindDlg”的变量。

1. 创建 CFindReplaceDialog 对象，并打开查找对话框。这一功能由 CGridDemoView::OnFind()函数实现，代码如下：

void CGridDemoView::OnFind()

{

// TODO: Add your command handler code here

//在堆中创建对话框

m\_pFindDlg = new CFindReplaceDialog();

//打开一个没有匹配全词的查找对话框

m\_pFindDlg->Create( TRUE, "", "", FR\_DOWN|FR\_HIDEWHOLEWORD, this );

}

1. 在 CGridDemoView 类中添加接收查找对话框 FINDMSGSTRING 消息的处理函数。打开查找对话框后，当用户单击“查找下一个”或者其他按钮时，对话框会向父窗口发送 FINDMSGSTRING 消息，这不是 MFC 中预先定义好的消息，故不能通过 ClassWizard 添加消息处理函数，必须手动添加。添加过程如下：

* 先在 CGridDemoView.h 文件的头部“class CGridDemoView”语句前添加如下代码，将

FINDMSGSTRING 消息注册为 WM\_FINDREPLACE 自定义消息：

const WM\_FINDREPLACE = ::RegisterWindowMessage(FINDMSGSTRING);

* 同 时 在 CGridDemoView.h 文 件 的 CGridDemoView 类 的 声 明 中 找 到 “DECLARE\_MESSAGE\_MAP()”语句，在这个语句前一行添加如下语句作为消息处理函数的声明：

afx\_msg LONG OnFindReplace(WPARAM wParam, LPARAM lParam);

* 接着在 CGridDemoView.cpp 文件的头部找到“END\_MESSAGE\_MAP()”语句，在这个语句的前一行添加如下语句绑定消息 WM\_FINDREPLACE 和消息处理函数

OnFindReplace()：

ON\_REGISTERED\_MESSAGE(WM\_FINDREPLACE, OnFindReplace)

* 最后在 CGridDemoView.cpp 文件的末尾编写 CGridDemoView::OnFindReplace()函数，其代码如下：

LRESULT CGridDemoView::OnFindReplace(WPARAM wparam, LPARAM lparam)

{

//获得查找对话框的对象指针

CFindReplaceDialog \*pDlg = CFindReplaceDialog::GetNotifier(lparam);

if( NULL != pDlg )

{

if (pDlg->FindNext())

{

//处理单击“查找下一个”按钮的查找处理

⋯ ⋯

}

}

else if (pDlg->IsTerminating())

{

//处理查找对话框关闭的消息

delete pDlg; //删除对话框对象

}

return 0;

}

在 CGridDemoView::OnFindReplace()查找消息处理函数中，先调用了 CFindReplaceDialog 静态成员函数 GetNotifier 根据消息的 lparam 参数获得查找对话框的指针，然后访问查找对话框对象查询当前的查找状态，主要有以下一些查询函数。

* FindNext()：查询用户是否单击“查找下一个”按钮。
* GetFindString()：获得查找字符串。
* GetReplaceString()：获得替换字符串。
* IsTerminating()：获得对话框是否关闭。
* MatchCase()：获得是否区分大小写。
* MatchWholeWord()：获得是否匹配全词。
* ReplaceAll()：查询用户是否单击“替换全部”按钮。
* ReplaceCurrent()：查询用户是否单击“替换”按钮。
* SearchDown()：查询用户是否想向下查找。完成上述 4 个步骤，就可以在 GridDemo 例程中使用查找对话框了。

### 4.5.4 文件对话框

文件对话框一般出现在用户打开或保存文件时，用于选择打开或保存的文件路径，如图

4-30 所示。

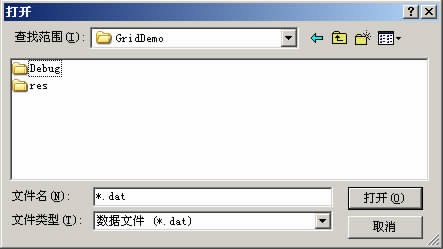


图 4-30 文件对话框

使用文件对话框时，同样只需打开一个对应的模态对话框。CFileDialog 封装了文件对话框的操作。相关代码在 CGridDemoDoc::OnFileOpen()函数中，添加步骤如下：（1）创建 CGridDemoDoc::OnFileOpen()消息处理函数。

CGridDemoDoc::OnFileOpen()是文档类 CGridDemoDoc 处理“文件|打开”菜单消息的函数。用户可以根据第 3 章的讲解，为 GridDemoDoc 添加 ID 为 ID\_FILE\_OPEN 的消息处理函数，名称为 OnFileOpen，即可完成该处理函数的添加。

1. 创建打开文件对话框对象。

这部分功能在 CGridDemoDoc::OnFileOpen()函数的头部实现，其创建了名为 dlg 的

CFileDialog 类对象，代码如下：

//设置过滤器

char szFilters[]=

"数据文件 (\*.dat)|\*.dat|所有文件 (\*.\*)|\*.\*||";

//创建打开文件对话框

CFileDialog dlg (TRUE, "数据文件", "\*.dat",

OFN\_FILEMUSTEXIST| OFN\_HIDEREADONLY, szFilters);

1. 打开并使用打开文件对话框。

这部分功能在 CGridDemoDoc::OnFileOpen()函数中，具体如下：

//运行对话框

if( dlg.DoModal ()==IDOK )

{

//处理选中的文件

CString szPathName = dlg.GetPathName();

|  |  |  |
| --- | --- | --- |
| } | SetTitle(dlg.GetFileTitle());  OnOpenDocument(szPathName); | //打开文档 |

在这个步骤中，还可以用以下 CFileDialog 的一些成员函数获得对话框的选择。

* GetFileExt()：获得选定文件的后缀名。
* GetFileName()：获得选定文件的名称，包括后缀名。
* GetFileTitle()：获得选定文件的标题，即不包括后缀名。
* GetFolderPath()：获得选定文件的目录。
* GetNextPathName()：获得下一个选定的文件的路径全名。
* GetPathName()：获得选定文件的路径全名。
* GetReadOnlyPref()：获得是否“以只读方式打开”。
* GetStartPosition()：获得文件名列表中的第一个元素的位置。

### 4.5.5 字体对话框

字体对话框用于选择字体，如图 4-31 所示。CFontDialog 封装了字体对话框的操作，其使用方法类似于其他模态通用对话框。



图 4-31 字体对话框

相关代码在 CGridDemoView::OnSetFont()函数中，添加步骤如下：（1）添加 CGridDemoView::OnSetFont()消息处理函数。

CGridDemoView::OnSetFont()函数是“编辑|字体”菜单的消息处理函数，对应 ID 为 ID\_SET\_FONT。用户需要按照第 3 章的讲解，在“编辑”顶层菜单中添加标题为“字体”， ID 为 ID\_SET\_FONT 的子菜单，然后为 CGridDemoView 类添加这个 ID 的消息处理函数，即

OnSetFont()。

（2）编辑 CGridDemoView::OnSetFont()消息处理函数使用字体对话框。代码如下：

void CGridDemoView::OnSetFont()

{

// TODO: Add your command handler code here

//获得选中单元格的旧字体

CFont \*pFont=m\_CellArray[m\_nCurCellID].GetFont();

LOGFONT lf;

|  |  |
| --- | --- |
| } | if (pFont != NULL)  pFont->GetObject(sizeof(LOGFONT), &lf);  else //单元格使用默认字体  ::GetObject(GetStockObject(SYSTEM\_FONT), sizeof(LOGFONT), &lf);  //使用选择字体对话框  CFontDialog dlg(&lf, CF\_SCREENFONTS|CF\_INITTOLOGFONTSTRUCT); if (dlg.DoModal()==IDOK)  {  //设置新字体  m\_CellArray[m\_nCurCellID].SetFont(&lf);  } |

### 4.5.6 颜色对话框

当需要选择颜色的时候就可以使用 Windows 通用的颜色对话框，如图 4-32 所示。

CColorDialog 封装了字体对话框的使用，其使用方法与前面所述的模态通用对话框类似。



图 4-32 颜色对话框

GridDemo 例程中使用颜色对话框的代码在 CGridDemoView::OnSetColor()函数中。添加步骤如下。

1. 添加 CGridDemoView::OnSetColor()消息处理函数。

CGridDemoView::OnSetColor()函数是“编辑|颜色”菜单的消息处理函数，对应 ID 为

ID\_SET\_COLOR。用户需要按照第 3 章的讲解，在“编辑”顶层菜单中添加标题为“颜色”， ID 为 ID\_SET\_COLOR 的子菜单，然后为 CGridDemoView 类添加这个 ID 的消息处理函数，即 OnSetColor()。

1. 编辑 CGridDemoView::OnSetColor()消息处理函数使用颜色对话框。代码如下：

void CGridDemoView::OnSetColor()

{

// TODO: Add your command handler code here

//获得选中单元格的旧前景色

CColorDialog dlg(m\_CellArray[m\_nCurCellID].GetFgColor()); if (dlg.DoModal()==IDOK)

|  |  |  |
| --- | --- | --- |
|  | { | //设置新的前景颜色  m\_CellArray[m\_nCurCellID].SetFgColor(dlg.GetColor()); |
| } | } |  |

## 4.6 消息对话框

消息对话框 MessageBox 是 Windows 系统中自带的最简单的对话框，用于提示一些简单的信息，如图 4-33 所示。本章 4.1.3 节中实例 Sample4\_1 在登录对话框的消息处理函数中就大量使用了消息对话框。



图 4-33 消息对话框

在 MFC 中，消息对话框通过 CWnd::MessageBox()和 AfxMessageBox()两个函数进行调用。前一个函数是 CWnd 的成员函数，而 AfxMessageBox()则是全局函数。两个函数的原型分别为：

int MessageBox(

LPCTSTR lpszText,

LPCTSTR lpszCaption = NULL,

UINT nType = MB\_OK

);

int AfxMessageBox(

LPCTSTR lpszText,

UINT nType = MB\_OK,

UINT nIDHelp = 0

);

* lpszText 参数：用于设置对话框的内容。
* lpszCaption 参数：用于设置对话框的标题。
* nType 参数：设置消息对话框的属性，属性可以为表 4-2 和表 4-3 中的取值的按位或的组合。
* nIDHelp 参数：用于设置帮助的上下文 ID。

表 4-2 消息对话框主要参数——对话框类型

|  |  |
| --- | --- |
| nType 取值 | 参数说明 |
| MB\_ABORTRETRY | 有“终止”、“重试”和“忽略”按钮 |
| MB\_OK | 有“确定”按钮 |
| MB\_OKCANCEL | 有“确定”和“取消”按钮 |
| MB\_RETRYCANCEL | 有“重试”和“取消”按钮 |
| MB\_YESNO | 有“是”和“否”按钮 |
| MB\_YESNOCANCEL | 有“是”、“否”和“取消”按钮 |

表 4-3 消息对话框主要参数——对话框图标

|  |  |
| --- | --- |
| nType 取值 | 显示图标 |
| MB\_ICONEXCLAMTION  MB\_ICONWARNING |  |
|
|
| MB\_ICONASTERISK  MB\_ICONINFORMATION |  |
|
|
| MB\_ICONQUESTION |  |
|
|
| MB\_ICONHAND  MB\_ICONSTOP  MB\_ICONERROR |  |

如图 4-33 所示的消息对话框可以用如下代码显示：

MessageBox("查找至末尾，是否从头开始查找？",

"查找",MB\_YESNO|MB\_ICONQUESTION)

|  |
| --- |
| 技巧：WM\_MOUSEFIST 和 WM\_MOUSELAST 分别指定了鼠标所能产生的第一个消息和最后一  个消息类型，因此程序可以 GetMessage（&msg, WM\_MOUSEFIRST, WM\_MOUSELAST）函数等待并获取鼠标消息，获取的鼠标消息保存在参数 msg 中。 |

运行 MessageBox()或者 AfxMessageBox()都可打开模态消息对话框，返回值就是用户单击的消息对话框的按钮 ID：

* IDABORT：单击“终止”按钮。
* IDCANCEL：单击“取消”按钮。
* IDIGNORE：单击“忽略”按钮。
* IDNO：单击“否”按钮。
* IDOK：单击“确定”按钮。
* IDRETRY：单击“重试”按钮。
* IDYES：单击“是”按钮。

## 4.7 本章小结

本章详细介绍了 Windows 系统中最常用的用户界面——对话框的使用，并根据对话框的各种不同类型进行分类介绍。本章需要掌握的内容有：

* 一般对话框的创建和使用方法，包括模态和非模态对话框。
* 属性页对话框的创建和使用。
* 通用对话框的使用。
* 消息对话框的使用。

对话框在任何一个程序中都会大量使用，设计功能丰富完善、界面合理美观的对话框是设计用户界面的重点和难点，需要认真学习、理解和掌握。

# 第 5 章 常用控件

在与用户的交互过程中，控件（Control）担任着重要的角色。MFC 提供了大量的控件类，它们封装了控件的功能，通过这些控件类，可以方便地创建控件，对控件进行查询和控制。控件可粗分为两类，一类是在 Windows 3.x 中就已支持的传统控件，另一类是 Windows 95/NT/2000 中支持的新型 Win32 控件，所有的控件类都是 CWnd 类的直接或间接派生类。不过对于新型 Win32 控件，应用程序只能创建控件对象而不能创建数据变量。不管是什么类型的控件，一般都具有 WS\_CHILD 和 WS\_VISIBLE 窗口风格，其中 WS\_CHILD 指定窗口为子窗口，WS\_VISIBLE 使窗口是可见的。另外，大部分控件还具有 WS\_TABSTOP 风格，

WS\_TABSTOP 使控件具有 Tabstop 属性。MFC 类库提供的主要控件如表 5-1 所示。

表 5-1 常用控件列表

|  |  |  |
| --- | --- | --- |
| 控件 | 功能 | 对应控件类 |
| 静态文本（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 |
| 文本编辑框（Edit Box） | 输入并编辑正文，支持单行和多行编辑 | CEdit |
| 命令按钮（Button） | 响应用户的输入，触发相应的事件 | CButton |
| 检查框（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 |
| 列表控件（ListCtrl） | 主要用来显示列表型报表，提供多种显示模式 | CListCtrl |
| 树型控件（TreeCtrl） | 主要用来显示树状结构数据 | CTreeCtrl |
| 选项卡控件（TabCtrl） | 主要用来显示多种重叠的相关对话框界面 | CTabCtrl |

## 5.1 静态文本（Static text）控件

静态文本控件是传统控件的一种，传统控件主要包括文本控件、命令按钮、选择框、单选按钮、编辑框、列表框和组合框等，如图 5-1 所示。

图 5-1 主要传统控件样式

### 5.1.1 传统控件通知消息

在介绍静态文本控件的用法之前，先讲述传统控件的控件通知消息。控件通过向父窗口发送控件通知消息来表明发生了某种事件。例如，在按钮上单击鼠标时，按钮控件会向父窗口发送 BN\_CLICKED 消息。传统控件的通知消息实际上是通过 WM\_COMMAND 消息发给父窗口的（滚动条除外），在该消息的 wParam 中含有通知消息码（如 BN\_CLICKED）和控件的 ID，在 lParam 中则包含了控件的句柄。

利用 ClassWizard 可以很容易地为控件通知消息加入消息映射和消息处理函数，这在上一章中的对话框应用程序中已经讲解过了。传统控件的消息映射宏是ON\_XXXX，其中XXXX 表示通知消息码，如 BN\_CLICKED。ON\_XXXX 消息映射宏格式如下，该宏有两个参数，一个是控件的 ID，一个是消息处理函数名：

ON\_XXXX(nID,memberFxn)

消息处理函数的声明形式如下：

afx\_msg void memberFxn();

有时，为了处理方便，需要把多个 ID 连续的控件发出的相同消息映射到同一个处理函数上，这时就要用到 ON\_CONTROL\_RANGE 宏。例如，要处理一组单选按钮发出的

BN\_CLICKED 消息，相应的消息映射如下：

ON\_CONTROL\_RANGE(BN\_CLICKED,IDC\_FIRST,IDC\_LAST,OnRadioClicked)

ON\_CONTROL\_RANGE 消息映射宏的第一个参数是控件消息码，第二和第三个参数分别指明了一组连续的控件 ID 中的头一个和最后一个 ID，最后一个参数是消息处理函数名。函数 OnRadioClicked 的声明如下，该函数比上面的 OnAdd-memberFxn 多了一个参数 nID 以说明发送通知消息的控件 ID。

afx\_msg void OnRadioClicked(UINT nID);

另外需要注意的是：ClassWizard 不支持 ON\_CONTROL\_RANGE 宏，所以需要手工建立消息映射和消息处理函数。

### 5.1.2 静态控件的使用

静态控件主要起说明和装饰作用，包括静态文本（Static Text）控件和图片（Picture）控件两类。静态文本控件用来显示文本信息。静态控件封装在 MFC 的 CStatic 类中，成员函数

Create 负责创建静态控件，该函数的声明如下：

BOOL Create( LPCTSTR lpszText, DWORD dwStyle, const RECT& rect, CWnd\* pParentWnd, UINT nID =

0xffff );

其中参数 lpszText 指定了控件中显示的文本内容，dwStyle 指定了静态控件的风格，如表 5-2 所示。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。pParentWnd 指向父窗口，该参数不能为 NULL。nID 则说明了控件的 ID。如果控件创建成功，该函数返回 TRUE，否则返回 FALSE。

表 5-2 静态控件风格

|  |  |
| --- | --- |
| 控件风格 | 含义 |
| 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\_CENTER | 使显示的正文居中对齐，正文可以换行 |
| 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\_ICON | 使控件显示一个在资源中定义的图标，图标的名字有 Create 函数的 lpszText 参数指定 |
| SS\_LEFT | 左对齐正文，正文能回绕 |
| SS\_LEFTNOWORDWRAP | 左对齐正文，正文不能回绕 |
| SS\_NOPREFIX | 使静态正文串中的&不是一个热键提示符 |
| 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\_SIMPLE | 使静态正文在运行时不能被改变并使正文显示在单行中 |
| SS\_USERITEM | 指定一个用户定义项 |
| 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） |

除了指定如表 5-2 所示的风格外，一般还要为控件指定 WS\_CHILD 和 WS\_VISIBLE 窗口风格。一个典型的静态正文控件的风格为 WS\_CHILD|WS\_VISIBLE|SS\_LEFT。对于用对话框模板编辑器创建的静态控件，可以在控件的属性对话框中指定控件风格。例如，可以在静态正文控件的属性对话框中选择 Simple，这相当于指定了 SS\_SIMPLE 风格，这在一定程度上方便了控件的生成。

### 5.1.3 Cstatic 类的主要成员函数

CStatic 类主要的成员函数如表 5-3 所示，同时也可以利用 CWnd 类的成员函数 GetWindowText、SetWindowText 和 GetWindowTextLength 来查询和设置静态控件中显示的文本信息。

表 5-3 CStatic 类的主要成员函数

|  |  |
| --- | --- |
| 成员函数声明 | 用途 |
| HBITMAP SetBitmap(HBITMAP hBitmap); | 指定要显示的位图 |
| HBITMAP GetBitmap() const; | 获取由 SetBitmap 指定的位图 |
| HICON SetIcon(HICON hIcon); | 指定要显示的图标 |
| HICON GetIcon() const; | 获取由 SetIcon 指定的图标 |
| HCURSOR SetCursor(HCURSOR hCursor); | 指定要显示的光标图片 |
| CURSOR GetCursor(); | 获取由 SetCursor 指定的光标 |
| HENHMETAFILE  SetEnhMetaFile(HENHMETAFILE hMetaFile); | 指定要显示的增强图元文件 |
| HENHMETAFILE GetEnhMetaFile() const; | 获取由 SetEnhMetaFile 指定的图元文件 |

|  |
| --- |
| 技巧：设置静态文本框的背景色和文本颜色  Windows 消息 WM\_CTLCOLOR 用于控制对话框中控件的颜色，可以有选择性地设置静态文本框控件的背景和颜色，如下面的代码段用于更改 ID 为 IDC\_SMALL\_STATIC 的静态文本框的颜色和背景。 |

|  |
| --- |
| HBRUSH CRMBDlg::OnCtlColor(CDC\* pDC, CWnd\* pWnd, UINT nCtlColor)  {  HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor);    // TODO: Change any attributes of the DC here switch(nCtlColor)  {  case CTLCOLOR\_STATIC:  {  if(pWnd->GetDlgCtrlID() == IDC\_SMALL\_STATIC)  {  pDC->SetBkMode(TRANSPARENT); pDC->SetTextColor(RGB(10,10,255)); hbr = (HBRUSH)GetStockObject(HOLLOW\_BRUSH);  }  else  {  hbr = CDialog::OnCtlColor(pDC,pWnd,nCtlColor);  }  }  break;  default:  break;  }  // TODO: Return a different brush if the default is not desired  return hbr; |

}

## 5.2 文本编辑（Edit Box）控件

### 5.2.1 文本编辑控件的使用

文本编辑（Edit Box）控件是一种传统控件，实际上是一个简易的正文编辑器，用户可以在编辑框中输入并编辑正文。

文本编辑框既可以是单行的，也可以是多行的，多行编辑框的编号是从零开始编排的。在一个多行编辑框中，除了最后一行外，每一行的结尾处都有一对回车换行符（用“\r\n”表示），这对回车换行符是正文换行的标志，在屏幕上是不可见的。文本编辑控件会向父窗口发出如表 5-4 所示的控件通知消息。

表 5-4 文本编辑控件的通知消息

|  |  |
| --- | --- |
| 通知消息 | 含义 |
| EN\_CHANGE | 编辑框的内容被用户改变了，与 EN\_UPDATE 不同，该消息是在编辑框显示的正文被刷新后才发出的 |
| EN\_ERRSPACE | 编辑框控件无法申请足够的动态内存来满足需要 |
| EN\_HSCROLL | 用户在水平滚动条上单击鼠标 |
| EN\_KILLFOCUS | 编辑框失去输入焦点 |
| EN\_MAXTEXT | 输 入 的 字 符 超 过 了 规 定 的 最 大 字 符 数 。 在 没 有 ES\_AUTOHSCROLL 或 ES\_AUTOVSCROLL 的编辑框中，当正文超出了编辑框的边框时也会发出该消息 |
| EN\_SETFOCUS | 编辑框获得输入焦点 |
| EN\_UPDATE | 在编辑框准备显示改变了的正文时发送该消息 |
| EN\_VSCROLL | 用户在垂直滚动条上单击鼠标 |

编辑框控件的相关操作函数封装在 MFC 类库中的 CEdit 类中。CEdit 类的成员函数 Create 用于创建按钮控件，该函数的声明如下：

BOOL Create(DWORD dwStyle,const RECT& rect,CWnd\* pParentWnd,UINT nID);

其中参数 dwStyle 指定了编辑框控件风格，文本编辑控件的主要风格如表 5-5 所示。

rect 指定了编辑框的位置和尺寸。

pParentWnd 指定了父窗口，不能为 NULL。文本编辑框的 ID 由 nID 指定。如果创建成功，该函数返回 TRUE，否则返回 FALSE。

表 5-5 文本编辑控件风格

|  |  |
| --- | --- |
| 控件风格 | 含义 |
| ES\_AUTOHSCROLL | 当用户在行尾键入一个字符时，正文将自动向右滚动 10 个字符，当用户按回车键时，正文总是滚向左边 |
| ES\_AUTOVSCROLL | 当用户在最后一个可见行按回车键时，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\_LOWERCASE | 把用户输入的字母统统转换成小写字母 |
| ES\_MULTILINE | 指定一个多行编辑器。若多行编辑器不指定 ES\_AUTOHSCROLL 风格，则会自动换行，若不指定 ES\_AUTOVSCROLL，则多行编辑器会在窗口中正文装满时发出警告声响 |

续表

|  |  |
| --- | --- |
| 控件风格 | 含义 |
| ES\_NOHIDESEL | 默认时，当编辑框失去输入焦点后会隐藏所选的正文，当获得输入焦点时又显示出来。设置该风格可禁止这种默认行为 |
| ES\_OEMCONVERT | 使编辑框中的正文可以在 ANSI 字符集和 OEM 字符集之间相互转换。这在编辑框中包含文件名时是很有用的 |
| ES\_PASSWORD | 使所有键入的字符都用“\*”来显示 |
| ES\_RIGHT | 右对齐正文 |
| ES\_UPPERCASE | 把用户输入的字母统统转换成大写字母 |
| ES\_READONLY | 将编辑框设置成只读的 |
| ES\_WANTRETURN | 使多行编辑器接收回车键输入并换行。如果不指定该风格，按回车键会选择默认的命令按钮，这往往会导致对话框的关闭 |

除了表 5-5 中的风格外，一般还要为文本编辑控件指定 WS\_CHILD、WS\_VISIBLE、 WS\_TABSTOP 和 WS\_BORDER 等窗口风格。创建一个普通的单行编辑框应指定风格 WS\_CHILD|WS\_VISIBLE|WS\_TABSTOP |WS\_BORDER|ES\_LEFT，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。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。

对于用对话框模板编辑器创建的编辑框控件，可以在控件的属性对话框中指定表 5-5 中列出的控件风格。例如，在属性对话框中选择 Multi-line 项，相当于指定了 ES\_MULTILINE 风格。对话框模板编辑器中设置编辑框控件属性的 Edit Properties 对话框如图 5-2 所示。

 图 5-2 Edit Properties 对话框

### 5.2.2 Cedit 类的主要成员函数

文本编辑控件支持剪贴板操作，CEdit 类提供了一些与剪贴板有关的成员函数，如表 5-6 所示。

表 5-6 与剪贴板有关的 CEdit 成员函数

|  |  |
| --- | --- |
| 成员函数声明 | 用途 |
| void Clear() | 清除编辑框中被选择的正文 |
| void Copy() | 把在编辑框中选择的正文拷贝到剪贴板中 |
| void Cut() | 清除编辑框中被选择的正文并把这些正文拷贝到剪贴板中 |
| void Paste() | 将剪贴板中的正文插入到编辑框的当前插入符处 |
| BOOL Undo() | 撤消上一次键入。对于单行编辑框，该函数总返回 TRUE，对于多行编辑框，返回 TRUE 表明操作成功，否则返回 FALSE |

可以用下列 CEdit 或 CWnd 类的成员函数来查询编辑框。在学习下面的函数时，读者会经常遇到术语“字符索引”。字符的字符索引是指从编辑框的开头字符开始的字符编号，它是从零开始编号的。也就是说，字符索引实际上是指当把整个编辑正文看作一个字符串数组时，该字符所在的数组元素的下标。

1．GetWindowText()函数

该成员函数声明方式如下：

int GetWindowText(LPTSTR lpszStringBuf, int nMaxCount) const;

void GetWindowText(CString& rString) const;

这两个函数均是 CWnd 类的成员函数，可用来获得窗口的标题或控件中的正文。第一种声明方式用 lpszStringBuf 参数指向的字符串数组作为拷贝正文的缓冲区，参数 nMaxCount 用于获取拷贝到缓冲区中的最大字符数，该函数返回以字节为单位的实际拷贝字符数（不包括结尾的空字节）。第二种声明方式用一个 CString 对象作为缓冲区。

2．GetWindowTextLength()函数

该成员函数声明方式如下：

int GetWindowTextLength() const;

CWnd 的成员函数，可用来获得窗口的标题或控件中的正文的长度。

3．GetSel()函数

该成员函数声明按如下方式进行声明：

DWORD GetSel() const;

void GetSel(int& nStartChar,int& nEndChar) const;

两个函数都是 CEdit 的成员函数，用来获得所选正文的位置。第一种声明方式用于返回一个 DWORD 值，其中低位字节说明了被选择的正文开始处的字符索引，高位字节说明了选择的正文结束处的最后一个字符的字符索引，如果没有正文被选择，那么返回的低位和高位字节都是当前插入符所在字符的字符索引。第二种声明方式中的两个参数是两个引用，其含义与第一种声明方式的低位和高位字节相同。

4．LineFromChar()函数

该成员函数声明方式如下：

int LineFromChar(int nIndex = –1) const;

CEdit 的成员函数，仅用于多行编辑框，用来返回指定字符索引所在行的行索引（从零开始编号）。参数 nIndex 指定了一个字符索引，如果 nIndex 是-1，那么函数将返回选择正文的第一个字符所在行的行号，若没有正文被选择，则该函数会返回当前的插入符所在行的行号。

5．LineIndex()函数

该成员函数声明方式如下：

int LineIndex(int nLine = –1) const;

CEdit 的成员函数，仅用于多行编辑框，用来获得指定行的开头字符的字符索引，如果指定行超过了编辑框中的最大行数，该函数将返回-1。参数 nLine 指定了从零开始的行索引，如果它的值为-1，则函数返回当前的插入符所在行的字符索引。

6．GetLineCount()函数

该成员函数声明方式如下：

int GetLineCount() const;

CEdit 的成员函数，仅用于多行编辑框，用来获得正文的行数。如果编辑框是空的，那么该函数的返回值是 1。

7．LineLength()函数

该成员函数声明方式如下：

int LineLength( int nLine = –1 ) const;

CEdit 的成员函数，用于获取指定字符索引所在行的字节长度（行尾的回车和换行符不计算在内），参数 nLine 说明了为字符索引。如果 nLine 的值为-1，则函数返回当前行的长度（假如没有正文被选择），或选择正文占据的行的字符总数减去选择正文的字符数（假如有正文被选择）。若用于单行编辑框，则函数返回整个正文的长度。

8．GetLine()函数

该成员函数声明方式如下：

int GetLine( int nIndex, LPTSTR lpszBuffer ) const;

int GetLine( int nIndex, LPTSTR lpszBuffer, int nMaxLength ) const;

CEdit 的成员函数，仅用于多行编辑框，用来获得指定行的正文（不包括行尾的回车和换行符）。参数 nIndex 是行号，lpszBuffer 指向存放正文的缓冲区，nMaxLength 规定了拷贝的最大字节数。若指定的行号小于编辑框的实际行数，函数返回实际拷贝的字节数，若指定的行号大于编辑框的实际行数，则函数返回 0。需要注意的是，GetLine 函数不会在缓冲区中字符串的末尾添加字符串结束符（NULL）。

9．SetWindowText()函数

该函数的声明方式如下：

void SetWindowText(LPCTSTR lpszString);

CWnd 的成员函数，用来设置窗口的标题或控件中的正文，参数 lpszString 可以是一个

CString 对象，或是一个指向字符串的指针。

10．SetSel()函数

该成员函数声明方式如下：

void SetSel(DWORD dwSelection,BOOL bNoScroll=FALSE);

void SetSel(int nStartChar,int nEndChar,BOOL bNoScroll=FALSE);

CEdit 的成员函数，用来选择编辑框中的正文。参数 dwSelection 的低位字节说明了选择开始处的字符索引，高位字节说明了选择结束处的字符索引。如果低位字为 0 且高位字节为

-1，那么就选择所有的正文，如果低位字节为-1，则取消所有的选择。参数 bNoScroll 的值如果是 FALSE，则滚动插入符并使之可见，否则就不滚动。参数 nStartChar 和 nEndChar 的含义与参数 dwSelection 的低位字节和高位字节相同。

11．ReplaceSel()函数

该成员函数声明方式如下：

void ReplaceSel(LPCTSTR lpszNewText,BOOL bCanUndo = FALSE);

CEdit 的成员函数，用来将所选正文替换成指定的正文，参数 lpszNewText 指向用来替换的字符串。参数 bCanUndo 的值为 TRUE 说明替换是可以被撤消的。

如果在编辑控件中只允许接收数字，可以在一个标准的编辑控件中指定标志 ES\_NUMBERS，该标志限制编辑控件只按收数字字符。如果用户需要复杂的编辑控件，可以使用微软公司提供的屏蔽编辑控件，它是一个很有用的 OLE 定制控件。

也可以从 CEdit 类派生一个类并处理 WM\_CHAR 消息，然后从编辑控件中过滤出特定的字符。具体方法为：首先，使用 ClassWizard 建立一个 CEdit 的派生类，其次，在对话类中指定一个成员变量，将编辑控件分类在 OnInitdialog()中调用下面的代码段：

|  |
| --- |
| CWnd::SubclassDlgItem  {  //In your dialog class declaration (.H file ) private :  CMyEdit m\_wndEdit ; // Instance of your new edit control }    //In you dialog class implementation (.CPP file )  BOOL CSampleDialog : : OnInitDialog ( )  {  …  //Subclass the edit lontrod .  m\_wndEdit.SubclassDlgItem (IDC\_EDIT,this );  …  } |

然后使用 ClassWizard 处理 WM\_CHAR 消息，计算 nChar 并决定所执行的操作，用户可以确定是否修改、传送字符。

下例说明了如何显示字母字符，如果字符是字母字符，则调用 CWnd::OnChar，否则不调用 OnChar。

|  |
| --- |
| //Only display alphabetic dharacters .  void CMyEdit::OnChar (UINT nChar,UINT nRepCnt,UINT nFlags) {  //Determine if nChar is an alphabetic character . if (::IsCharAlpha ((TCHAR) nChar))  CEdit::OnChar (nChar, nRepCnt , nFlags );  } |

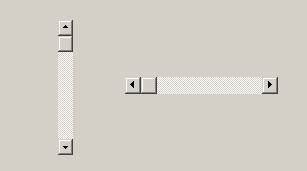
如果要修改字符，则不能仅仅简单地用修改过的 nChar 调用 CEdit::OnChar，然后用 CEdit::OnChar 调用 CWnd::Default 获取原来的 wParam 和 lParam 的值。要修改一个字符，需要首先修改 nChar，然后用修改过的 nChar 调用 CWnd: : DefWindowProc。下例说明了如何将字符转变为大写。

|  |
| --- |
| //Make all characters uppercase  void CMyEdit::OnChar (UINT nChar,UINT nRepCnt,UINT nFlags) {  //Make sure character is uppercase . if (::IsCharAlpha (( TCHAR) nChar)) nChar=::CharUpper (nChar ) ;  //Bypass default OnChar processing and directly call //default window proc.  DefWindProc(WM\_CHAR,nChar,MAKELPARAM (nRepCnt,nFlags));  } |

## 5.3 滚动条（Scroll Bar）控件

### 5.3.1 滚动条控件的使用

滚动条（Scroll Bar）是一种传统控件，主要用来从某一预定义值范围内快速有效地进行浏览，分为垂直滚动条（Horizontal Scroll Bar）和水平滚动条（Vertical Scroll Bar）两种，如图 5-3 所示。在滚动条内有一个滚动框，用来表示当前的值。用鼠标单击滚动条，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。Windows 2000 中还支持比例滚动框，即用滚动框的大小来反映页相对于整个范围的大小。

 图 5-3 滚动条样例

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中的一部分，而是一个实际的窗口，可以放置在窗口客户区的任意地方，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窗口，因此可以拥有输入焦点，可以响应光标控制键，如 PageUp、PageDown、Home 和 End 等。滚动条控件封装在 MFC 的 CScrollBar 类中，CScrollBar 类的 Create 成员函数用于创建控件，该函数的声明如下：

BOOL Create(DWORD dwStyle, const RECT& rect,CWnd\* pParentWnd,UINT nID );

参数 dwStyle 指定了控件的风格，rect 说明了控件的位置和尺寸，pParentWnd 指向父窗体，该参数不能为 NULL。nID 则说明了控件的 ID。如果创建成功，该函数返回 TRUE，否则返回 FALSE。

要创建一个普通的水平滚动条控件，应指定风格 WS\_CHILD|WS\_VISIBLE|BS\_HORZ，

要创建一个普通的垂直滚动条控件，应指定风格 WS\_CHILD|WS\_VISIBLE|BS\_VERT。

### 5.3.2 CscrollBar 类的主要成员函数

下面将介绍 CScrollBar 类主要的成员函数。

1．GetScrollPos()函数

该成员函数声明如下：

int GetScrollPos() const;

该函数返回滚动框的当前位置，若操作失败则返回 0。

2．SetScrollPos()函数

该成员函数声明如下：

int SetScrollPos(int nPos,BOOL bRedraw = TRUE);

该函数将滚动框移动到指定位置。参数 nPos 指定了新的位置，参数 bRedraw 表示是否需要重绘滚动条，如果为 TRUE，则重绘之。函数返回滚动框原来的位置，若操作失败则返回 0。

3．GetScrollRange()函数

该成员函数声明如下：

void GetScrollRange(LPINT lpMinPos,LPINT lpMaxPos) const;

该函数对滚动条的滚动范围进行查询。参数 lpMinPos 和 lpMaxPos 分别指向滚动范围的

最小、最大值。

4．SetScrollRange()函数

该成员函数声明如下：

void SetScrollRange(int nMinPos,int nMaxPos,BOOL bRedraw = TRUE);

该函数用于指定滚动条的滚动范围。参数 nMinPos 和 nMaxPos 分别指定了滚动范围的最

小、最大值，由这两者指定的滚动范围不得超过 32767。当两者都为 0 时，滚动条将被隐藏。参数 bRedraw 表示是否需要重绘滚动条，如果为 TRUE，则重绘之。

5．GetScrollInfo()函数

该成员函数声明如下：

BOOL GetScrollInfo(LPSCROLLINFO lpScrollInfo,UINT nMask);

该函数用来获取滚动条的各种状态，包括滚动范围、滚动框的位置和页尺寸，参数 lpScrollInfo 指向一个 SCROLLINFO 结构，该结构定义如下：

|  |
| --- |
| typedef struct tagSCROLLINFO {  UINT cbSize;  //结构的尺寸（字节为单位）  UINT fMask;  //说明结构中的哪些参数是有效的，可以是屏蔽值的组合,如 SIF\_POS|SIF\_PAGE，若为 SIF\_ALL 则整个结构都有效  int nMin;  //滚动范围最大值，当 fMask 中包含 SIF\_RANGE 时有效  int nMax;  //滚动范围最小值，当 fMask 中包含 SIF\_RANGE 时有效  UINT nPage;  //页尺寸，用来确定比例滚动框的大小，当 fMask 中包含 SIF\_PAGE 时有效  int nPos;  //滚动框的位置，当 fMask 中包含 SIF\_POS 有效  int nTrackPos;  //滚动时滚动框的位置，当 fMask 中包含 SIF\_TRACKPOS 时有效，该参数只能查询，不能设置，最好不要用该参数来查询拖动时滚动框的位置  }SCROLLINFO;  typedef SCROLLINFO FAR \*LPSCROLLINFO; |

参数 nMask 的意义与 SCROLLINFO 结构中的 fMask 相同，函数在获得有效值后返回 TRUE，否则返回 FALSE。

6．SetScrollInfo()函数

该成员函数声明如下：

BOOL SetScrollInfo(LPSCROLLINFO lpScrollInfo,BOOL bRedraw = TRUE);

该函数用于设置滚动条的各种状态，如设定页尺寸从而实现比例滚动框。参数lpScrollInfo 指向一个 SCROLLINFO 结构，参数 bRedraw 表示是否需要重绘滚动条，如果为 TRUE，则重绘之。若操作成功，该函数返回 TRUE，否则返回 FALSE。

CWnd 类也提供了一些函数来查询和设置所属的标准滚动条。这些函数与 CScrollBar 类的函数同名，且功能相同，但每个函数都多了一个参数，用来选择滚动条。

7．GetScrollPos()函数

该成员函数声明如下：

int GetScrollPos(int nBar) const;

参数 nBar 用来选择滚动条，可以为下列值。

* SB\_HORZ：指定水平滚动条。
* SB\_VERT：指定垂直滚动条。

8．OnHScroll()和 OnVScroll()函数

无论是标准滚动条，还是滚动条控件，滚动条的通知消息都是用 WM\_HSCROLL 和 WM\_VSCROLL 消息发送出去的。对这两个消息的默认处理函数是 CWnd::OnHScroll 和 CWnd::OnVScroll，一般需要在派生类中对这两个函数从新设计，以实现滚动功能。这两个函数的声明如下：

afx\_msg void OnHScroll(UINT nSBCode,UINT nPos,CScrollBar\* pScrollBar); afx\_msg void OnVScroll(UINT nSBCode,UINT nPos,CScrollBar\* pScrollBar);

参数 nSBCode 是通知消息码，主要通知码如表 5-7 所示。nPos 是滚动框的位置，只有在 nSBCode 为 SB\_THUMBPOSITION 或 SB\_THUMBTRACK 时，该参数才有意义。如果通知消息是滚动条控件发来的，那么 pScrollBar 是指向该控件的指针，如果是标准滚动条发来的，则 pScrollBar 为 NULL。

表 5-7 通知消息码

|  |  |
| --- | --- |
| 消息 | 含义 |
| SB\_BOTTOM/SB\_RIGHT | 滚动到底端（右端） |
| SB\_TOP/SB\_LEFT | 滚动到顶端（左端） |
| SB\_LINEDOWN/SB\_LINERIGHT | 向下（向右）滚动一行（列） |
| SB\_LINEUP/SB\_LINELEFT | 向上（向左）滚动一行（列） |
| SB\_PAGEDOWN/SB\_PAGERIGHT | 向下（向右）滚动一页 |
| SB\_PAGEUP/SB\_PAGELEFT | 向上（向左）滚动一页 |
| SB\_THUMBPOSITION | 滚动到指定位置 |
| SB\_THUMBTRACK | 滚动框被拖动。可利用该消息来跟踪对滚动框的拖动 |
| SB\_ENDSCROLL | 滚动结束 |

## 5.4 按钮（Button、Radio Box、Check Box、Group Box）控件

### 5.4.1 按钮控件的使用

按钮也是一种传统控件。按钮控件包括命令按钮（Pushbutton）、检查框（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鼠标的单击作出反应并触发相应的事件，在按钮中既可以显示文本，也可以显示位图。选择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。

选择框和单选按钮是一种特殊的按钮，它们有选择和未选择两种状态。当一个选择框处于选择状态时，选择框的小方框内会出现一个“√ ”，当单选按钮处于选择状态时，会在圆圈中显示一个黑色实心圆。此外，检查框还有一种不确定状态，这时检查框呈灰色显示，不能接受用户的输入，以表明控件是无效的或无意义的。按钮控件会向父窗口发出控件通知消息，主要的通知消息如表 5-8 所示的。

表 5-8 按钮控件的通知消息

|  |  |
| --- | --- |
| 消息 | 含义 |
| BN\_CLICKED | 用户在按钮上单击了鼠标 |
| BN\_DOUBLECLICKED | 用户在按钮上双击了鼠标 |

按钮控件封装在 MFC 的 CButton 类中，CButton 类的成员函数 Create 用于创建按钮控件，

该函数的声明如下：

BOOL Create(LPCTSTR lpszCaption,DWORD dwStyle,const RECT& rect,CWnd\* pParentWnd,UINT nID);

参数 lpszCaption 指定了按钮显示的文本。dwStyle 指定了按钮的风格，主要的按钮风格如表 5-9 所示，dwStyle 参数可以是这些风格的组合。rect 说明了按钮的位置和尺寸。 pParentWnd 指向父窗口，该参数不能为 NULL。nID 是按钮的 ID。如果创建成功，该函数返回 TRUE，否则返回 FALSE。

表 5-9 按钮的风格

|  |  |
| --- | --- |
| 消息 | 含义 |
| BS\_AUTOCHECKBOX | 同 BS\_CHECKBOX，不过单击鼠标时按钮会自动反转 |
| BS\_AUTORADIOBUTTON | 同 BS\_RADIOBUTTON，不过单击鼠标时按钮会自动反转 |
| BS\_AUTO3STATE | 同 BS\_3STATE，不过单击按钮时会改变状态 |
| BS\_CHECKBOX | 指定在矩形按钮右侧带有标题的选择框 |
| BS\_DEFPUSHBUTTON | 指定默认的命令按钮，这种按钮的周围有一个黑框，用户可以按回车键来快速选择该按钮 |
| BS\_GROUPBOX | 指定一个组框 |
| BS\_LEFTTEXT | 使控件的标题显示在按钮的左边 |
| BS\_OWNERDRAW | 指定一个自绘式按钮 |
| BS\_PUSHBUTTON | 指定一个命令按钮 |
| BS\_RADIOBUTTON | 指定一个单选按钮，在圆按钮的右边显示正文 |
| BS\_3STATE | 同 BS\_CHECKBOX，不过控件有 3 种状态—选择、未选择和变灰 |

除了上表中的风格外，一般还要为控件指定 WS\_CHILD、WS\_VISIBLE 和 WS\_TABSTOP 等窗口风格，WS\_TABSTOP 风格使控件具有 Tabstop 属性。创建一个普通按钮应指定的风格为 WS\_CHILD|WS\_VISIBLE|WS\_TABSTOP 。创建一个普通检查框可以指定风格为 WS\_CHILD|WS\_VISIBLE|WS\_TABSTOP|BS\_AUTOCHECKBOX。创建组中第一个单选按钮指定风格 WS\_CHILD|WS\_VISIBLE|WS\_TABSTOP|WS\_GROUP|BS\_AUTORADIOBUTTON，组中其他单选按钮指定风格时则不应该包括 WS\_TABSTOP 和 WS\_GROUP。

对于用对话框模板编辑器创建的按钮控件，可以在控件的属性对话框中指定表 5-8 中列出的控件风格。例如，在命令按钮的属性对话框中选择 Default button，相当于指定了

BS\_DEFPUSHBUTTON。

### 5.4.2 Cbutton 类的主要成员函数

下面介绍 CButton 类的主要的成员函数，这些成员函数在实际编程过程中有很大的用处。

1．GetState()函数

该成员函数声明如下：

UINT GetState() const;

该函数返回按钮控件的各种状态。可以用下列屏蔽值与函数的返回值相与，以获得各种信息。返回值的状态主要定义有下面几种。

* 0x0003：用来获取检查框或单选按钮的状态。0 表示未选中，1 表示被选中，2 表示不确定状态（仅用于检查框）。
* 0x0004：用来判断按钮是否是高亮度显示。非 0 值意味着按钮是高亮度显示的。当单击按钮并按住鼠标左键时，按钮会呈高亮度显示。
* 0x0008：非零值表示按钮拥有输入焦点。

2．SetState()函数

该成员函数声明如下：

void SetState(BOOL bHighlight);

当参数 bHeightlight 值为 TRUE 时，该函数将按钮设置为高亮度状态，否则，去除按钮的高亮度状态。

3．GetCheck()函数

该成员函数声明如下：

int GetCheck() const;

返回检查框或单选按钮的选择状态。返回值 0 表示按钮未被选择，1 表示按钮被选择，2 表示按钮处于不确定状态（仅用于检查框）。

4．SetCheck()函数

该成员函数声明如下：

void SetCheck(int nCheck);

设置检查框或单选按钮的选择状态。参数 nCheck 的值含义与 GetCheck 返回值相同。

5．GetButtonStyle()函数

该成员函数声明如下：

UINT GetButtonStyle() const;

该函数用于获得按钮控件的 BS\_XXXX 风格。

6．SetButtonStyle()函数

该成员函数声明如下：

void SetButtonStyle( UINT nStyle, BOOL bRedraw = TRUE );

设置按钮的风格，参数 nStyle 指定了按钮的风格，bRedraw 为 TRUE 则重绘按钮，否则就不重绘。

7．SetBitmap()函数

该成员函数声明如下：

HBITMAP SetBitmap(HBITMAP hBitmap);

设置按钮显示的位图，参数 hBitmap 指定了位图的句柄，该函数还会返回按钮原来的位图。 8．GetBitmap()函数

该成员函数声明如下：

HBITMAP GetBitmap() const;

返回以前用 SetBitmap 设置的按钮位图。

9．SetIcon()函数

该成员函数声明如下：

HICON SetIcon( HICON hIcon);

设置按钮显示的图标，参数 hIcon 指定了图标的句柄，该函数还会返回按钮原来的图标。

10．GetIcon()函数

该成员函数声明如下：

HICON GetIcon() const;

返回以前用 SetIcon 设置的按钮图标。

11．SetCursor()函数

该成员函数声明如下：

HCURSOR SetCursor(HCURSOR hCursor);

设置按钮上显示的光标图，参数 hCursor 指定了光标的句柄，该函数还会返回按钮原来

的光标。

12．SetCursor()函数

该成员函数声明如下：

HCURSOR GetCursor();

返回以前用 GetCursor 设置的光标。 13．与按钮有关的 CWnd 成员函数

另外，可以使用下列的一些与按钮控件有关的 CWnd 成员函数来设置或查询按钮的状态。使用这些函数的好处在于不必构建按钮控件对象，只要知道按钮的 ID，就可以直接设置或查询按钮。

void CheckDlgButton(int nIDButton,UINT nCheck );

用来设置按钮的选择状态。参数 nIDButton 指定了按钮的 ID。nCheck 的值为 0 表示按钮未被选择，为 1 表示按钮被选择，为 2 表示按钮处于不确定状态。

void CheckRadioButton(int nIDFirstButton,int nIDLastButton,int nIDCheckButton );

用来选择组中的一个单选按钮。参数 nIDFirstButton 指定了组中第一个按钮的 ID， nIDLastButton 指定了组中最后一个按钮的 ID，nIDCheckButton 指定了要选择的按钮的 ID。

int GetCheckedRadioButton(int nIDFirstButton, int nIDLastButton);

用来获得一组单选按钮中被选中按钮的 ID。参数 nIDFirstButton 说明了组中第一个按钮的 ID， nIDLastButton 说明了组中最后一个按钮的 ID。

UINT IsDlgButtonChecked(int nIDButton) const;

返回检查框或单选按钮的选择状态。返回值为 0 表示按钮未被选择，为 1 表示按钮被选择，为 2 表示按钮处于不确定状态（仅用于检查框）。

同时还可以调用 CWnd 成员函数 GetWindowText()、GetWindowTextLength()和 SetWindow

-Text()来查询或设置按钮中显示的正文。

实例 5-1：控件基本使用方法实例。源代码在光盘中“\05\实例 5-1\RMB”目录下。

下面通过一个基于对话框的例子来说明控件的基本使用方法，该示例实现了将阿拉伯数字转换为汉字数字的功能。

主要代码如下：

|  |
| --- |
| const char\* RMB(double rmb)  {  if(rmb<0) return NULL;  static char \*CCC[10]={ //Capital form of Chinese character  "零","壹","贰","叁","肆","伍","陆","柒","捌","玖"  };  static char \*QQQ[19]={"分","角",".","圆","拾","佰","仟","万","拾","佰","仟","亿", "拾","佰","仟","万","拾","佰","仟"  };  static char result[256],tmp[256]; sprintf(tmp,"%.2lf",rmb); if(strlen(tmp)>19) return NULL; int i=strlen(tmp)-1,c;  char \*srcPtr=tmp,\*dstPtr=result; bool bPrevZero=false; for(; \*srcPtr; srcPtr++,i--) { if(\*srcPtr==’.’) continue; c=\*srcPtr-’0’; if(c!=0) {  strcpy(dstPtr,CCC[c]); dstPtr+=2; if(srcPtr==tmp && c==1 && strcmp(QQQ[i],"拾")==0) dstPtr-=2; strcpy(dstPtr,QQQ[i]); dstPtr+=2; bPrevZero=false; continue;  }  if(bPrevZero) {  if(strcmp(QQQ[i],"圆")==0 || strcmp(QQQ[i],"万")==0 || strcmp(QQQ[i],"亿")==0) { dstPtr-=2; \*dstPtr=0; if(strcmp(dstPtr-2,"亿")) {strcpy(dstPtr,QQQ[i]); dstPtr+=2;} bPrevZero=false; |

|  |
| --- |
| } } else {  if(strcmp(QQQ[i],"圆") && strcmp(QQQ[i],"万") && strcmp(QQQ[i],"亿")) { strcpy(dstPtr,"零"); dstPtr+=2; bPrevZero=true;  }else {  strcpy(dstPtr,QQQ[i]); dstPtr+=2; bPrevZero=false;  }  }  }  i=strlen(result);  if(strcmp(result+i-2,"零")==0) result[i-2]=0; tmp[0]=result[0]; tmp[1]=result[1]; tmp[2]=0; if(strcmp(tmp,"圆")==0) {  if(result[2]==0) strcpy(result,"零圆"); else {  tmp[0]=result[2]; tmp[1]=result[3]; tmp[2]=0; return strcmp(tmp,"零")==0 ? result+4 : result+2 ;  } }  return result;  } |

界面上的“Convert”按钮消息处理函数如下：

|  |
| --- |
| void CRMBDlg::OnConvertButton()  {    if(UpdateData(TRUE))  {  m\_szCha = RMB(m\_double);  UpdateData(FALSE);  }  else  {  m\_szCha = "";  UpdateData(FALSE);  }  } |

运行界面如图 5-4 所示，详细的源代码设计可参考附带光盘。



图 5-4 阿拉伯数字与汉字数字之间的转换界面

|  |
| --- |
| 技巧：改变控件的字体  控件也是窗口，可以调用 CWnd::SetFont 指定控件的新字体。该函数中使用一个 Cfont 指针，以保证在控件撤消之前不能撤消字体对象。如下面的代码将下压按钮时的字体改为 8 号 Arial 字体。 |

|  |
| --- |
| private :  CFont m\_font ;  // Set font in class implementation (.Cpp file ). Note m\_wndButton is a //member variable added by ClassWizard.DDX routines hook the member //variable to a dialog button contrlo.  BOOL CSampleDialog::OnInitDialog()  {  ...  //Create an 8-point Arial font  m\_font.CreateFont (MulDiv (8,pDC->GetDeviceCaps(LOGPIXELSY),72),\  ,0,0,FW\_NORMAL,0,0,0,ANSI\_CHARSER,OUT\_STROKE\_PRECIS,CLIP\_STROKE\_PRECIS,\  DRAFT\_QUALITY,VARIABLE\_PITCH|FF\_SWISS,\_T("Arial"));  //Set font for push button .  m\_wndButton.SetFont (&m \_font );  ...  } |

## 5.5 列表框（List Box）控件

### 5.5.1 列表框控件的使用

列表框也是一种传统控件，主要用于用户输入，它允许用户从所列出的表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。在单选列表框中一次只能选择一个列表项，而

在多重选择列表框中可以进行多重选择。列表框会向父窗口发送如表 5-10 所示的通知消息。

表 5-10 列表框控件的通知消息

|  |  |
| --- | --- |
| 消息 | 含义 |
| LBN\_DBLCLK | 用户用鼠标双击了一列表项，只有具有 LBS\_NOTIFY 的列表框才能发送该消息 |
| LBN\_ERRSPACE | 列表框不能申请足够的动态内存来满足需要 |
| LBN\_KILLFOCUS | 列表框失去输入焦点 |
| 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\_SELCHANGE | 单击鼠标选择了一列表项，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 |

列表框控件封装在 MFC 的 CListBox 类中，CListBox 类的 Create 成员函数用于列表框的

创建，该函数的声明如下。

BOOL Create(DWORD dwStyle,const RECT& rect,CWnd\* pParentWnd,UINT nID);

参数 dwStyle 指定了列表框控件的风格，如表 5-11 所示，dwStyle 可以是这些风格的组合。rect 说明了控件的位置和尺寸。pParentWnd 指向父窗口，该参数不能为 NULL。nID 则说明了控件的 ID。如果创建成功，该函数返回 TRUE，否则返回 FALSE。

表 5-11 列表框控件的风格

|  |  |
| --- | --- |
| 风格 | 含义 |
| LBS\_EXTENDEDSEL | 支持多重选择，在点击列表项时按住 Shift 键或 Ctrl 键即可选择多个项 |
| LBS\_HASSTRINGS | 指定一个含有字符串的自绘式列表框 |
| LBS\_MULTICOLUMN | 指 定 一 个 水 平 滚 动 的 多 列 列 表 框 ， 通 过 调 用  CListBox::SetColumnWidth 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鼠标而翻转 |
| LBS\_NOINTEGRALHEIGHT | 列表框的尺寸由应用程序而不是 Windows 指定。通常，Windows 指定尺寸会使列表项的某些部分隐藏起来 |
| LBS\_NOREDRAW | 当选择发生变化时防止列表框被更新，可发送消息改变该风格 |
| LBS\_NOTIFY | 当用户单击或双击鼠标时通知父窗口 |
| LBS\_OWNERDRAWFIXED | 指定自绘式列表框，即由父窗口负责绘制列表框的内容，并且列表项有相同的高度 |
| LBS\_OWNERDRAWVARIABLE | 指定自绘式列表框，并且列表项有不同的高度 |
| LBS\_SORT | 使插入列表框中的项按升序排列 |
| LBS\_STANDARD | 相当于指定了 WS\_BORDER|WS\_VSCROLL|LBS\_SORT |
| LBS\_USETABSTOPS | 使列表框在显示列表项时识别并扩展制表符(‘\t’)，默认的制表宽度是 32 个对话框单位 |
| 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 |
| LBS\_DISABLENOSCROLL | 使列表框在不需要滚动时显示一个禁止的垂直滚动条 |

除了上表中的风格外，一般还要为列表框控件指定 WS\_CHILD、WS\_VISIBLE、 WS\_TABSTOP、WS\_BORDER 和 WS\_VSCROLL 等风格。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。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。对于用对话框模板编辑器创建的列表框控件，可以在控件的属性对话框中指定表 5-11 中列出的控件风格。例如，在属性对话框中选择 Sort 项，相当与指定了 LBS\_SORT 风格。

### 5.5.2 ClistBox 类的主要成员函数

CListBox 类的成员函数有很多，分别介绍如下：

1．AddString()函数

该成员函数声明如下：

int AddString(LPCTSTR lpszItem);

该函数用来向列表框中加入字符串，其中参数 lpszItem 指定了要添加的字符串。函数的返回值是加入的字符串在列表框中的位置，如果发生错误，会返回 LB\_ERR 或 LB\_ERRSPACE （内存不够）。如果列表框未设置 LBS\_SORT 风格，那么字符串将被添加到列表的末尾，如果设置了 LBS\_SORT 风格，字符串会按排序规律插入到列表中。

2．InsertString()函数

该成员函数声明如下：

int InsertString(int nIndex, LPCTSTR lpszItem);

该函数用来在列表框中的指定位置插入字符串。参数 nIndex 给出了插入位置（索引），如果值为-1，则字符串将被添加到列表的末尾。参数 lpszItem 指定了要插入的字符串。该函数返回实际的插入位置，若发生错误，会返回 LB\_ERR 或 LB\_ERRSPACE。与 AddString 函数不同的是，InsertString 函数不会导致 LBS\_SORT 风格的列表框重新排序。不要在具有

LBS\_SORT 风格的列表框中使用 InsertString 函数，以免破坏列表项的次序。

3．DeleteString()函数

该成员函数声明如下：

int DeleteString(UINT nIndex);

该函数用于删除指定的列表项，其中参数 nIndex 指定了要删除项的索引。函数的返回值为剩下的表项数目，如果 nIndex 超过了实际的表项总数，则返回 LB\_ERR。

4．ResetContent()函数

该成员函数声明如下：

void ResetContent();

该函数用于清除所有列表项。

5．GetCount()函数

该成员函数声明如下：

int GetCount() const;

该函数返回列表项的总数，若出错则返回 LB\_ERR。

6．FindString()函数

该成员函数声明如下：

int FindString(int nStartAfter,LPCTSTR lpszItem) const;

该函数用于对列表项进行与大小写无关的搜索。参数 nStartAfter 指定了开始搜索的位置,

合理指定 nStartAfter 可以加快搜索速度，若 nStartAfter 为-1，则从头开始搜索整个列表。参数 lpszItem 指定了要搜索的字符串。函数返回与 lpszItem 指定的字符串相匹配的列表项的索引，若没有找到匹配项或发生了错误，则会返回 LB\_ERR。FindString 函数先从 nStartAfter 指定的位置开始搜索，若没有找到匹配项，则会从头开始搜索列表。只有找到匹配项，或对整个列表搜索完一遍后，搜索过程才会停止，所以不必担心会漏掉要搜索的列表项。

7．GetText()函数

该成员函数声明如下：

int GetText(int nIndex,LPTSTR lpszBuffer) const; void GetText(int nIndex,CString& rString) const;

这两个成员函数用于获取指定列表项的字符串。参数 nIndex 指定了列表项的索引。参数 lpszBuffer 指向一个接收字符串的缓冲区。引用参数 rString 则指定了接收字符串的 CString 对象。第一个版本的函数会返回获得的字符串的长度，若出错，则返回 LB\_ERR；第二个版本的函数则不会。

8．GetTextLen()函数

该成员函数声明如下：

int GetTextLen(int nIndex) const;

该函数返回指定列表项的字符串的字节长度。参数 nIndex 指定了列表项的索引。若出错则返回 LB\_ERR。

9．GetItemData()函数

该成员函数声明如下：

DWORD GetItemData(int nIndex) const;

每个列表项都有一个 32 位的附加数据。该函数返回指定列表项的附加数据，参数 nIndex

指定了列表项的索引。若出错则函数返回 LB\_ERR。

10．SetItemData()函数

该成员函数声明如下：

int SetItemData(int nIndex, DWORD dwItemData);

该函数用来指定某一列表项的 32 位附加数据。参数 nIndex 指定了列表项的索引。 dwItemData 是要设置的附加数据值。

11．GetSel()函数

该成员函数声明如下：

int GetSel(int nIndex) const;

该函数返回指定列表项的状态。参数 nIndex 指定了列表项的索引。如果查询的列表项被选择了，函数返回一个正值，否则返回 0，若出错则返回 LB\_ERR。

12．GetCurSel()函数

该成员函数声明如下：

int GetCurSel() const;

该函数仅适用于单选择列表框，用来返回当前被选择项的索引，如果没有列表项被选择或有错误发生，则函数返回 LB\_ERR。

13．SetCurSel()函数

该成员函数声明如下：

int SetCurSel(int nSelect);

该函数仅适用于单选择列表框，用来选择指定的列表项。该函数会滚动列表框以使选择项可见。参数 nIndex 指定了列表项的索引，若为-1，那么将清除列表框中的选择。若出错函数返回 LB\_ERR。

14．SelectString()函数

该成员函数声明如下：

int SelectString(int nStartAfter,LPCTSTR lpszItem);

该函数仅适用于单选择列表框，用来选择与指定字符串相匹配的列表项。该函数会滚动列表框以使选择项可见。参数的意义及搜索的方法与函数 FindString 类似。如果找到了匹配的项，函数返回该项的索引，如果没有匹配的项，函数返回 LB\_ERR 并且当前的选择不被改变。

15．GetSelCount()函数

该成员函数声明如下：

int GetSelCount() const;

该函数仅用于多重选择列表框，它返回选择项的数目，若出错函数返回 LB\_ERR。

16．SetSel()函数

该成员函数声明如下：

int SetSel(int nIndex,BOOL bSelect = TRUE);

该函数仅适用于多重选择列表框，它使指定的列表项选中或落选。参数 nIndex 指定了列表项的索引，若为-1，则相当于指定了所有的项。参数 bSelect 为 TRUE 时选中列表项，否则使之落选。若出错则返回 LB\_ERR。

17．GetSelItems()函数

该成员函数声明如下：

int GetSelItems(int nMaxItems, LPINT rgIndex) const;

该函数仅用于多重选择列表框，用来获得选中的项的数目及位置。参数 nMaxItems 说明了参数 rgIndex 指向的数组的大小。参数 rgIndex 指向一个缓冲区，该数组是一个整型数组，用来存放选中的列表项的索引。函数返回放在缓冲区中的选择项的实际数目，若出错函数返回 LB\_ERR。

18．SetItemRange()函数

该成员函数声明如下：

int SetItemRange(BOOL bSelect,int nFirstItem,int nLastItem);

该函数仅用于多重选择列表框，用来使指定范围内的列表项选中或落选。参数 nFirstItem 和 nLastItem 指定了列表项索引的范围。如果参数 bSelect 为 TRUE，那么就选择这些列表项，否则就使它们落选。若出错函数返回 LB\_ERR。

|  |
| --- |
| 技巧：实现列表框的提示条（可以显示完整的列表框的项目文本）  用一个 CListBox 派生类实现宿主（owner-draw）列表框，这个列表框的项目宽度超过了列表框本身的宽度，因此当鼠标指针指向大宽度的列表框项时，希望显示一个类似 Toolbar 的提示窗口，在窗口中显示完整的列表框项目文本。  附带光盘下的工程演示列表框提示条的实现，主要是创建了可重用窗口类 CPopupText。  CPopupText 类会根据提示文本的长度自己决定弹出窗口的大小。它的字体默认值与状态行显示字体相同（由 SystemParametersInfo(SPI\_GETNONCLIENTMETRICS)返回的 NONCLIENTMETRICS 结构成员 lfStatusFont 定义），设置 SW\_SHOWNA 是很重要的，因为你不想使提示窗口为活动窗口，而只是显示它。另外，CPopupText 还提供一个专门的函数 CPopupText::ShowDelayed，它的功能是在显示提示窗口之前设置一个毫秒级延时，如果延时为零，则立刻显示提示窗口，你可以使用这个特点来替代对 ShowWindow 的调用。如果要隐藏提示窗口或取消 ShowDelayed，调用 CPopupText::Cancel。 |

实例 5-2：实现列表框的提示条实例。源代码在光盘中“\05\实例 5-2\ListTest”目录下。

## 5.6 组合框（Combo box）控件

### 5.6.1 组合框控件的使用

组合框是一种传统控件，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。组合框分为简易式（Simple）、下拉式（Dropdown）和下拉列表式（Drop List）3 种。简易式组合框包含一个编辑框和一个总是显示的列表框。下拉式组合框同简易式组合框类似，二者的区别在于单击下拉箭头后是否会弹出列表框。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。

Windows 中比较常用的是下拉式和下拉列表式组合框，在集成开发环境（IDE）中就大量使用了这两种组合框。二者都具有占地小的特点，这在界面日益复杂的今天是十分重要的。下拉列表式组合框的功能与列表框类似。下拉式组合框的典型应用是作为记事列表框使用，既把用户在编辑框中敲入的东西存储到列表框组件中，这样当要重复同样的输入时，可以从列表框组件中选取而不必在编辑框组件中重新输入。在集成开发环境（IDE）中的 Find 对话框中就可以找到一个典型的下拉式组合框。

应用一个列表框控件时，主要应遵循下列原则。

* 在创建组合框时指定 CBS\_DROPDOWNLIST 风格。
* 要限制列表项的数目，以防止内存不够。
* 如果在编辑框中输入的字符串不能与列表框组件中的列表项匹配，那么应该把该字符串插入到列表框中的 0 位置处。最老的项处于列表的末尾。如果列表项的数目超出了限制，则应把最老的项删除。
* 如果在编辑框中输入的字符串可以与列表框组件中的某一项完全匹配，则应该先把该项从列表的当前位置删除，然后在将其插入列表的 0 位置处。

组合框控件会向父窗口发送控件通知消息，主要的通知消息如表 5-12 所示的。

表 5-12 组合框控件的通知消息

|  |  |
| --- | --- |
| 消息 | 含义 |
| 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鼠标，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\_ERRSPACE | 组合框无法申请足够的内存来容纳列表项 |
| 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鼠标时就会导致该消息的发送 |
| CBN\_SELENDOK | 用户选择了一项，然后按了回车键或单击了下滚箭头，该消息表明用户确认了自己所作的选择 |
| CBN\_KILLFOCUS | 组合框失去了输入焦点 |
| CBN\_SELCHANGE | 用户通过单击或移动箭头键改变了列表的选择 |
| CBN\_SETFOCUS | 组合框获得了输入焦点 |

组合框控件封装在 MFC 的 CComboBox 类中。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，该函数的声明如下：

BOOL Create(DWORD dwStyle,const RECT& rect,CWnd\* pParentWnd,UINT nID);

参数 dwStyle 指定了组合框控件的风格，如表 5-13 所示，dwStyle 可以是这些风格的组合。rect 说明的是列表框组件下拉后组合框的位置和尺寸。pParentWnd 指向父窗口，该参数

不能为 NULL。nID 则说明了控件的 ID。如果创建成功，该函数返回 TRUE，否则返回 FALSE。

表 5-13 组合框的风格

|  |  |
| --- | --- |
| 风格 | 含义 |
| 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\_HASSTRINGS | 指定一个含有字符串的自绘式组合框 |
| CBS\_OEMCONVERT | 使编辑框组件中的正文可以在ANSI字符集和OEM字符集之间相互转换。这在编辑框中包含文件名时是很有用的 |
| CBS\_OWNERDRAWFIXED | 指定自绘式组合框，即由父窗口负责绘制列表框的内容，并且列表项有相同的高度 |
| CBS\_OWNERDRAWVARIABLE | 指定自绘式组合框，并且列表项有不同的高度 |
| CBS\_SIIMPLE | 指定一个简易式组合框 |
| CBS\_SORT | 自动对列表框组件中的项进行排序 |
| CBS\_DISABLENOSCROLL | 使列表框在不需要滚动时显示一个禁止的垂直滚动条 |
| 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隐藏起来 |

CBS\_SIMPLE、CBS\_DROPDOWN 和 CBS\_DROPDOWNLIST 分别用来将组合框指定为简易式、下拉式和下拉列表式。一般还要为组合框指定 WS\_CHILD、WS\_VISIBLE、 WS\_TABSTOP、WS\_VSCROLL 和 CBS\_AUTOHSCROLL 风格。如果要求自动排序，还应指定 CBS\_SORT 风格。

对于用对话框模板编辑器创建的组合框控件，可以在控件的属性对话框中指定上表中列出的控件风格。例如，在属性对话框中选择 Dropdown，相当于指定了 CBS\_DROPDOWN。

### 5.6.2 CcomboBox 类的主要成员函数

CComboBox 类的成员函数较多。其中常用的函数可分为两类，分别针对编辑框组件和列表框组件。可以想象，这些函数与 CEdit 类和 CListBox 类的成员函数肯定有很多类似之处，但它们也会有一些不同的特点。如果读者能从“组合框是由编辑框和列表框组成”这一概念出发，就能够很快的掌握 CComboBox 的主要成员函数。

事实上，绝大部分 CComboBox 的成员函数都可以看成是 CEdit 或 CListBox 成员函数的重现。函数的功能、函数名，甚至函数的参数都是类似的。为了方便学习，下面在介绍 CComboBox 类的成员函数时，采用了与对应的 CEdit 或 CListBox 成员函数相比较的做法。在成员函数的列表中，分别列出了成员函数名、对应的 CEdit 或 CListBox 成员函数以及二者之间的不同之处。

针对编辑框组件的主要成员函数如表 5-14 所示。该表的前 3 个函数实际上是 CWnd 类的成员函数，可用来查询和设置编辑框组件。

表 5-14 针对编辑框组件的 CComboBox 成员函数

|  |  |  |
| --- | --- | --- |
| 成员函数名 | 对应的 CEdit 成员函数 | 与 CEdit 成员函数的不同之处 |
| CWnd::GetWindowText | CWnd::GetWindowText | 无 |
| CWnd::SetWindowText | CWnd::SetWindowText | 无 |
| CWnd::GetWindowTextLength | CWnd::GetWindowTextLength | 无 |
| GetEditSel | GetSel | 仅函数名不同 |
| SetEditSel | SetSel | 函数名不同，且无 bNoScroll 参数 |
| Clear | Clear | 无 |

续表

|  |  |  |  |
| --- | --- | --- | --- |
|  | 成员函数名 | 对应的 CEdit 成员函数 | 与 CEdit 成员函数的不同之处 |
| Copy |  | Copy | 无 |
| Cut |  | Cut | 无 |
| Paste |  | Paste | 无 |

与 CListBox 的成员函数类似，针对列表框组件的 CComboBox 成员函数也可以分为 3 类。表 5-15 列出了用于插入和删除列表项的成员函数，表 5-16 列出了用于搜索、查询和设置列表框的成员函数，与列表项的选择有关的成员函数如表 5-17 所示。需要指出的是，如果这些函数出错，则返回 CB\_ERR，而不是 LB\_ERR。另外，排序的组合框具有的是 CBS\_SORT 风格，而不是 LBS\_SORT。

表 5-15 用于插入和删除列表项的 CComboBox 成员函数

|  |  |  |
| --- | --- | --- |
| 成员函数名 | 对应的 CListBox 成员函数 | 与 CEdit 成员函数的不同之处 |
| AddString | AddString | 无 |
| InsertString | InsertString | 无 |
| DeleteString | DeleteString | 无 |
| ResetContent | ResetContent | 无 |
| Dir | Dir | 无 |

表 5-16 用于搜索、查询和设置列表框的 CComboBox 成员函数

|  |  |  |
| --- | --- | --- |
| 成员函数名 | 对应的 CListBox 成员函数 | 与 CEdit 成员函数的不同之处 |
| GetCount | GetCount | 无 |
| FindString | FindString | 无 |
| GetLBText | GetText | 仅函数名不同 |
| GetLBTextLen | GetTextLen | 仅函数名不同 |
| GetItemData | GetItemData | 无 |
| SetItemData | SetItemData | 无 |
| GetTopIndex | GetTopIndex | 无 |
| SetTopIndex | SetTopIndex | 无 |

表 5-17 与列表项的选择有关的 CComboBox 成员函数

|  |  |  |
| --- | --- | --- |
| 成员函数名 | 对应的 CListBox 成员函数 | 与 CEdit 成员函数的不同之处 |
| GetCurSel | GetCurSel | 无 |
| SetCurSel | SetCurSel | 新选的列表项的内容会被拷贝到编辑框组件中 |
| SelectString | SelectString | 新选的列表项的内容会被拷贝到编辑框组件中 |

## 5.7 图片（Picture）控件

图片（Picture）控件实际上是一种静态文本（CStatic）控件，可以通过在创建静态文本控件时加入特定的标志来得到图片控件，这些方面的内容，具体可以参见 5.1 节。这里主要介绍一下如何在资源编辑器中快捷方便地使用图片控件。

选中图片控件后拖放到对话框上，可以通过图片控件属性对话框来对图片控件进行基本设置，图片对话框如图 5-5 所示。



图 5-5 Picture Properties 对话框图片控件所支持的格式如表 5-18 所示。

表 5-18 Picture 控件支持的格式

|  |  |
| --- | --- |
| 格式 | 含义 |
| Frame | 显示一个无填充的矩形框，边框颜色可以通过 Color 下拉列表设定 |
| Rectangle | 显示一个填充的矩形框，矩形颜色可通过 Color 下拉列表设定 |
| Icon | 显示一个图标（Icon），图标通过 Image 下拉列表来设置图标资源 ID |
| Bitmap | 显示一个位图（Bitmap），位图通过 Image 下拉列表来设置位图资源 ID |
| Enhanced Metafile | 显示一个加强的元数据文件（Metafile） |

由于图片控件在具体使用方法上和静态文本控件没有什么差别，因此在这里就不详细叙述了。

实例 5-3：美化对话框应用程序实例。源代码在光盘中“\05\实例 5-3\New\_RMB”目录下。

下面为数字转换一例中的截面定制漂亮的界面和按钮，通过此例程读者将学习到如何美化对话框应用程序的界面。

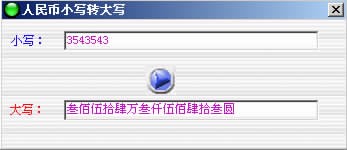
定制“外衣”的主要代码是使用 WM\_ERASEBKGND、WM\_CTLCOLOR 消息重画用户界面，利用 CBitmapButton 实现位图按钮，主要代码如下：

|  |
| --- |
| HBRUSH CRMBDlg::OnCtlColor(CDC\* pDC, CWnd\* pWnd, UINT nCtlColor)  {  HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor);    // TODO: Change any attributes of the DC here switch(nCtlColor) { case CTLCOLOR\_EDIT:  {  if (pWnd->GetDlgCtrlID() == IDC\_EDIT\_NUM)  {  pDC->SetBkMode(TRANSPARENT);  pDC->SetTextColor(RGB(200,10,200)); //改变字体的颜色 hbr = HBRUSH(GetStockObject(HOLLOW\_BRUSH));  }  else if(pWnd->GetDlgCtrlID() == IDC\_EDIT\_CHA)  { |

|  |
| --- |
| pDC->SetBkMode(TRANSPARENT);  pDC->SetTextColor(RGB(200,10,200)); //改变字体的颜色 hbr = HBRUSH(GetStockObject(HOLLOW\_BRUSH));  }  else  {  hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor);  }  break;  }  case CTLCOLOR\_STATIC:  {  if(pWnd->GetDlgCtrlID() == IDC\_SMALL\_STATIC)  {  pDC->SetBkMode(TRANSPARENT); pDC->SetTextColor(RGB(10,10,255)); hbr = (HBRUSH)GetStockObject(HOLLOW\_BRUSH);  }  else if(pWnd->GetDlgCtrlID() == IDC\_LARGE\_STATIC)  {  pDC->SetBkMode(TRANSPARENT); pDC->SetTextColor(RGB(255,10,10)); hbr = (HBRUSH)GetStockObject(HOLLOW\_BRUSH);  }  else  {  hbr = CDialog::OnCtlColor(pDC,pWnd,nCtlColor);  }  }  break;  default:  break;  }  // TODO: Return a different brush if the default is not desired  return hbr;  }  BOOL CRMBDlg::OnEraseBkgnd(CDC\* pDC)  {  CBitmap bmp; bmp.LoadBitmap(IDB\_BG); |

|  |  |
| --- | --- |
| } | CBrush brush(&bmp);  CRect rc;  GetClientRect(&rc); pDC->FillRect(rc, &brush);  return true; |

程序运行界面如图 5-6 所示。

 图 5-6 定制外衣后对话框的用户界面

## 5.8 列表（List Ctrl）控件

### 5.8.1 Win32 新型控件概述

列表控件是一种 Win32 控件，这和前面提到的传统控件在某些方面存在不同。Win32 控件采用新的 WM\_NOTIFY 消息来实现新控件的消息通知机制。在该消息的 wParam 中含有控件的 ID，lParam 中则有一个指针，这个指针指向一个结构。这个结构要么是 NMHDR 结构，要么是一个以 NMHDR 结构作为第一个成员的扩充结构。通过 NMHDR 结构及其扩充结构可以传递附加数据。从理论上讲，可以通过扩充结构传送任意多的数据。需要指出的是，由于 NMHDR 结构是扩充结构的第一个成员，因此 lParam 中的指针即可以认为是指向 NMHDR 结构的，也可以认为是指向包含 NMHDR 结构的扩充结构的。

1．NMHDR 结构

NMHDR 结构如下：

typedef struct tagNMHDR

{

HWND hwndFrom; //控件窗口的句柄

UINT idFrom; //控件的 ID

UINT code; //控件的通知消息码

}NMHDR;

一个典型的扩充结构如下，该结构用于列表视图控件的 LVN\_KEYDOWN 通知消息。

typedef struct tagLV\_KEYDOWN{

NMHDR hdr; //NMHDR 结构作为第一个成员

WORD wVKey;

UINT flags;

}LV\_KEYDOWN;

2．WM\_NOTIFY 消息处理

WM\_NOTIFY 的消息映射由宏 ON\_NOTIFY 负责，该消息映射宏具有如下形式：

ON\_NOTIFY(wNotifyCode,id,memberFxn)

参数 wNotifyCode 说明了通知消息码，参数 id 是控件的 ID，第三个参数则是消息处理函数名。消息处理函数应该按下面的形式声明，其中参数 pNotifyStruct 指向 NMHDR 及其扩充结构，参数 result 指向一个处理结果：

afx\_msg void memberFxn(NMHDR\* pNotifyStruct, LRESULT \* result);

利用 ClassWizard 可以很方便地加入 WM\_NOTIFY 消息映射及其处理函数，一个典型的

WM\_NOTIFY 消息映射如下，其中 LVN\_KEYDOWN 是 IDC\_LIST1 列表视图控件发出的通知消息：

ON\_NOTIFY(LVN\_KEYDOWN, IDC\_LIST1,OnKeydownList1)

消息处理函数 OnKeydownList1 的定义如下：

void CMyDlg::OnKeydownList1(NMHDR\* pNMHDR, LRESULT\* pResult)

{

LV\_KEYDOWN\* pLVKeyDow = (LV\_KEYDOWN\*)pNMHDR;

// TODO: Add your control notification handler

// code here

\*pResult = 0;

}

在函数中 ClassWizard 自动把 pNHHDR 指针强制转换成 LV\_KEYDOWN 型并赋给 pLVKeyDow 指针，这样，在函数中可通过这两个指针访问 LV\_KEYDOWN 扩充结构及其所含的 NMHDR 结构。另外，在函数返回时，ClassWizard 自动将处理结果赋 0 值。

可以利用 ON\_NOTIFY\_RANGE 宏把多个 ID 连续的控件发出的相同消息映射到同一个

处理函数上，具体形式如下：

ON\_NOTIFY\_RANGE(wNotifyCode, id, idLast, memberFxn)

其中参数 id 和 idLast 分别说明明了一组连续的控件 ID 中的头一个和最后一个 ID。相应的消息处理函数应按下面的形式声明：

afx\_msg void memberFxn(UINT id, NMHDR\* pNotifyStruct,LRESULT \* result);

与普通的 WM\_NOTIFY 消息处理函数相比，该函数多了一个参数 id 用来说明发送通知

消息的控件 ID。

ClassWizard 不支持 ON\_NOTIFY\_RANGE 宏，所以需要手工建立消息映射和消息处理函数。

### 5.8.2 列表控件的使用

列表控件用来成列地显示数据。列表视图的表项通常包括图标（Icon）和标题（Label）两部分，它们分别提供了对数据的形象和抽象描述。列表视图控件是对传统的列表框的重大改进，它能够以下列 4 种格式显示数据。读者可以在资源管理器中的视图(View)菜单中切换列表视图的显示格式，来看看 4 种格式的不同之处。

* 大图标格式（Large Icons）：可逐行显示多列表项，图标的大小可由应用程序指定，通常是 32× 32 像素，在图标的下面显示标题。
* 小图标格式（Small Icons）：可逐行显示多列表项，图标的大小可由应用程序指定，通常是 16× 16 像素，在图标的右面显示标题。表项以行的方式组织。
* 列表格式（List）。与小图标格式类似。不同之处在于表项是逐列多列显示的。
* 报告格式（Report 或 Details）：每行仅显示一个表项，在标题的左边显示一个图标，表项可以不显示图标而只显示标题。表项的右边可以附加若干列子项（Subitem），子项只显示正文。在控件的顶端还可以显示一个列表头用来说明各列的类型。列表视图的报告格式很适合显示报表（如数据库报表）。

典型的列表控件的格式如图 5-7 所示。

在讨论如何使用列表视图控件以前，先向读者介绍一下与该控件有关的一些数据类型。

1．LV\_COLUMN 结构

该结构仅用于报告式列表视图，用来描述表项的某一列。要想向表项中插入新的一列，需要用到该结构。LV\_COLUMN 结构的定义为：

|  |
| --- |
| typedef struct \_LV\_COLUMN{  UINT mask;  //屏蔽位的组合（见下面括号），表明哪些成员是有效的  int fmt;  //该列的表头和子项的标题显示格式(LVCF\_FMT)。可以是 LVCFMT\_CENTER、LVCFMT\_LEFT  //或 LVCFMT\_RIGHT  int cx; //以像素为单位的列的宽度(LVCF\_FMT)  LPTSTR pszText; //指向存放列表头标题正文的缓冲区(LVCF\_TEXT) int cchTextMax; //标题正文缓冲区的长度(LVCF\_TEXT) int iSubItem; //说明该列的索引(LVCF\_SUBITEM)  }LV\_COLUMN; |

2．LV\_ITEM 结构

该结构用来描述一个表项或子项，它包含了项的各种属性，其定义为如下：

|  |
| --- |
| typedef struct \_LV\_ITEM {  UINT mask;  //屏蔽位的组合（见下面括号），表明哪些成员是有效的  int iItem;  //从 0 开始编号的表项索引（行索引）  int iSubItem;  //从 1 开始编号的子项索引（列索引），若值为 0 则说明该成员无效，结构描述的是一个表项而不  是子项  UINT state;  //项的状态(LVIF\_STATE)  UINT stateMask;  //项的状态屏蔽  LPTSTR pszText;  //指向存放项的正文的缓冲区(LVIF\_TEXT)    int cchTextMax;  //正文缓冲区的长度(LVIF\_TEXT)  int iImage;  //图标的索引(LVIF\_IMAGE)  LPARAM lParam;  //32 位的附加数据(LVIF\_PARAM)  }LV\_ITEM; |

其中 lParam 成员可用来存储与项相关的数据，这在有些情况下是很有用的。state 和 stateMask 的值如表 5-19 所示，stateMask 用来说明要获取或设置哪些状态。

表 5-19 列表视图的状态

|  |  |  |
| --- | --- | --- |
| 状态 | 对应的状态屏蔽 | 含义 |
| LVIS\_CUT | 同左 | 项被选择用来进行剪切和粘贴操作 |
| LVIS\_DROPHILITED | 同左 | 项成为拖动操作的目标 |
| LVIS\_FOCUSED | 同左 | 项具有输入焦点 |
| LVIS\_SELECTED | 同左 | 项被选中 |

3．NM\_LISTVIEW 结构

该结构用于存储列表视图通知消息的有关信息，大部分列表视图的通知消息都会附带指向该结构的指针。NM\_LISTVIEW 的定义如下：

typedef struct tagNM\_LISTVIEW {

NMHDR hdr;

//标准的 NMHDR 结构 int iItem; //表项的索引，若为-1 则无效

int iSubItem;

//子项的索引，若为 0 则无效

UINT uNewState;

//项的新状态

UINT uOldState;

//项原来的状态

UINT uChanged;

//取值与 LV\_ITEM 的 mask 成员相同，用来表明哪些状态发生了变化

POINT ptAction;

//事件发生时鼠标的客户区坐标

LPARAM lParam;

//32 位的附加数据

}NM\_LISTVIEW;

4．LV\_DISPINFO 结构

该结构包含了与项的显示有关的信息，其定义为：

typedef struct tagLV\_DISPINFO {

NMHDR hdr;

LV\_ITEM item;

}LV\_DISPINFO;

//LV\_KEYDOWN 结构,该结构包含一些与键盘有关的信息，其定义为

typedef struct tagLV\_KEYDOWN {

NMHDR hdr;

WORD wVKey; //虚拟键盘码

UINT flags; //总为 0

} LV\_KEYDOWN;

MFC 的 CListCtrl 类封装了列表视图控件。该类的 Create 函数负责创建控件，函数的声明为：

BOOL Create(DWORD dwStyle,const RECT& rect, CWnd\* pParentWnd,UINT nID );

其中参数 dwStyle 是如表 5-20 所示的控件风格的组合。

表 5-20 列表视图的风格

|  |  |
| --- | --- |
| 风格 | 含义 |
| LVS\_ALIGNLEFT | 显示格式是大图标或小图标时，标题放在图标的左边。默认情况下标题放在图标的下面 |
| LVS\_ALIGNTOP | 当显示格式是大图标或小图标时，标题放在图标的上边 |
| LVS\_AUTOARRANGE | 当显示格式是大图标或小图标时，自动排列控件中的表项 |
| LVS\_EDITLABELS | 用户可以修改标题 |
| LVS\_ICON | 指定大图标显示格式 |
| LVS\_LIST | 指定列表显示格式 |
| LVS\_NOCOLUMNHEADER | 在报告格式中不显示列的表头 |

续表

|  |  |
| --- | --- |
| 风格 | 含义 |
| LVS\_NOLABELWRAP | 当显示格式是大图标时，使标题单行显示。默认时是多行显示 |
| LVS\_NOSCROLL | 列表视图无滚动条 |
| LVS\_NOSORTHEADER | 报告列表视图的表头不能作为排序按钮使用 |
| LVS\_OWNERDRAWFIXED | 由控件的拥有者负责绘制表项 |
| LVS\_REPORT | 指定报告显示格式 |
| LVS\_SHAREIMAGELISTS | 使列表视图共享图像序列 |
| LVS\_SHOWSELALWAYS | 即使控件失去输入焦点，仍显示出项的选择状态 |
| LVS\_SINGLESEL | 指定一个单选择列表视图。默认时可以多项选择 |
| LVS\_SMALLICON | 指定小图标显示格式 |
| LVS\_SORTASCENDING | 按升序排列表项 |
| LVS\_SORTDESCENDING | 按降序排列表项 |

除上表的风格外，一般还要指定 WS\_CHILD 和 WS\_VISIBLE 窗口风格。风格组合

WS\_CHILD|WS\_VISIBLE|LVS\_REPORT|LVS\_AUTOARRANGE|LVS\_EDITLABLES 将指定一个自动排列的、可编辑标题的、单选择报告式列表视图控件。要指定大图标、小图标或列表式的列表视图控件，则应该把 LVS\_REPORT 换成 LVS\_ICON、LVS\_SMALLICON 或

LVS\_LIST。

对于用对话框模板创建的列表视图控件，可以在控件的属性对话框中指定上表中列出的控件风格。例如，在属性对话框的 Styles 页的 View 栏中选择 Icon，相当于指定了 LVS\_ICON 风格。

### 5.8.3 ClistCtrl 类的主要成员函数

CListCtrl 类提供了大量的成员函数。在这里结合实际应用来介绍一些常用的函数。

1．InsertColumn()函数

在以报告格式显示列表视图时，一般会显示一列表项和多列子项。在初始化列表视图时，先要调用 InsertColumn 插入各个列，该函数的声明如下：

int InsertColumn(int nCol,const LV\_COLUMN\* pColumn);

其中参数 nCol 是新列的索引，参数 pColumn 指向一个 LV\_COLUMN 结构，函数根据该

结构来创建新的列。若插入成功，函数返回新列的索引，否则返回-1。

2．DeleteColumn()函数

要删除某列，应调用 DeleteColumn()函数，其声明如下：

BOOL DeleteColumn(int nCol);

3．InsertItem()函数

要插入新的表项，应调用 InsertItem()函数。如果要显示图标，则应该先创建一个 CImageList 对象并使该对象包含用作显示图标的位图序列。然后调用 SetImageList 来为列表视图设置位图序列。函数的声明为

int InsertItem(const LV\_ITEM\* pItem);

参数 pItem 指向一个 LV\_ITEM 结构，该结构提供了对表项的描述。若插入成功则函数

返回新表项的索引，否则返回-1。

4．SetImageList()函数

该成员函数声明如下：

CImageList\* SetImageList(CImageList\* pImageList,int nImageList);

参数 pImageList 指向一个 CImageList 对象，参数 nImageList 用来指定图标的类型，若其值为 LVSIL\_NORMAL，则位图序列用作显示大图标；若值为 LVSIL\_SMALL，则位图序列用作显示小图标。可用该函数同时指定一套大图标和一套小图标。

5．DeleteItem()函数

要删除某表项，应调用 DeleteItem()函数，要删除所有的项，应调用 DeleteAllItems()函数。一旦表项被删除，其子项也被删除。该函数的声明如下：

BOOL DeleteItem(int nItem );

BOOL DeleteAllItems();

6．GetItemText()和 SetItemText()函数

调用 GetItemText()和 SetItemText()函数来查询和设置表项及子项显示的正文。SetItemText 的一个重要用途是对子项进行初始化。该函数的声明如下：

int GetItemText(int nItem,int nSubItem, LPTSTR lpszText, int nLen) const;

CString GetItemText(int nItem, int nSubItem) const;

BOOL SetItemText(int nItem,int nSubItem,LPTSTR lpszText);

其中参数 nItem 是表项的索引（行索引），nSubItem 是子项的索引（列索引），若 nSubItem 为 0 则说明函数是针对表项的。参数 lpszText 指向正文缓冲区，参数 nLen 说明了缓冲区的大小。第二个版本的 GetItemText 返回一个含有项的正文的 Cstring 对象。

7．GetItem()和 SetItem()函数

调用 GetItem()和 SetItem()函数来查询和设置指定项的属性。用这两个函数几乎可以查询

和设置指定项的所有属性，包括正文、图标及选择状态。该函数的声明如下：

BOOL GetItem(LV\_ITEM\* pItem) const;

BOOL SetItem(const LV\_ITEM\* pItem);

参数 pItem 是指向 LV\_ITEM 结构的指针，函数是通过该结构来查询或设置指定项的，在调用函数前应该使该结构的 iItem 或 iSubItem 成员有效以指定表项或子项。CListCtrl 还提供了一系列函数完成 GetItem()和 SetItem()的功能，其中 GetItemState()、GetItemText()和 GetItemData()函数用于查询，SetItemState()、SetItemTex()t 和 SetItemData()函数用于设置。

8．GetNextItem()函数

要寻找与指定表项相关的表项，或寻找具有某种状态的表项，应该调用 GetNextItem()函

数。该函数的一个重要用处是搜索被选择的表项。该函数的声明如下：

int GetNextItem(int nItem, int nFlags) const;

参数 nItem 是指定项的索引，参数 nFlags 是如表 5-21 所示的标志，用来指定查询的关系。

函数返回搜索到的表项的索引，若未找到则返回-1。表 5-21 参数 nFlags 的值

|  |  |
| --- | --- |
| 标志 | 含义 |
| LVNI\_ABOVE | 返回位于指定表项上方的表项 |
| LVNI\_ALL | 默认标志，返回指定表项的下一个表项（以索引为序） |
| LVNI\_BELOW | 返回位于指定表项下方的表项 |
| LVNI\_TOLEFT | 返回位于指定表项左边的表项 |
| LVNI\_TORIGHT | 返回位于指定表项右边的表项 |
| LVNI\_DROPHILITED | 返回拖动操作的目标表项 |
| LVNI\_FOCUSED | 返回具有输入焦点的表项 |
| LVNI\_SELECTED | 返回被选择的表项 |

要对表项进行排列、排序和搜索，可分别调用 Arrange、SortItems 和 FindItems 函数来完成。

有时需要在列表视图创建后动态地改变其显示格式，例如，资源管理器中的列表视图就可以在 4 种显示格式之间切换。改变显示格式其实就是改变列表视图的风格，要改变控件的风格，应先调用::GetWindowLong 获取控件原来的风格，并对其进行修改，然后调用::SetWindowLong 设置新的风格。这两个函数不是成员函数，而是 Windows API 函数，用来查询和设置窗口的风格。

列表视图控件还会发送自己特有的通知消息，常用的有下面这几个。

* LVN\_ITEMCHANGING 和 LVN\_ITEMCHANGED。当列表视图的状态发生变化时，会发送这两个通知消息。例如，当用户选择了新的表项时，程序就会收到这两个消息。消息会附带一个指向 NM\_LISTVIEW 结构的指针，消息处理函数可从该结构中获得状态信息。两个消息的不同之处在于，前者的消息处理函数如果返回 TRUE，那么就阻止选择的改变，如果返回 FALSE，则允许改变。
* LVN\_KEYDOWN。该消息表明了一个键盘事件。消息会附带一个指向 LV\_KEYDOWN 结构的指针，通过该结构程序可以获得按键的信息。
* LVN\_BEGINLABELEDIT 和 LVN\_ENDLABELEDIT。分别在用户开始编辑和结束编辑标题时发送。消息会附带一个指向 LV\_DISPINFO 结构的指针。在前者的消息处理函数中，可以调用 GetEditControl 成员函数返回一个指向用于编辑标题的编辑框的指针，如果处理函数返回 FALSE，则允许编辑，如果返回 TRUE，则禁止编辑。在后者的消息处理函数中，LV\_DISPINFO 结构中的 item.pszText 指向编辑后的新标题，如果 pszText 为 NULL，那么说明用户放弃了编辑，否则，程序应负责更新表项的标题，这可以由 SetItem 或 SetItemText 函数来完成。

## 5.9 树型（Tree Ctrl）控件

### 5.9.1 树型控件的使用

树型控件是一种特殊的列表，它能以树型分层结构显示数据。在 Windows 95 的资源管理器的左侧窗口中就有一个用于显示目录的典型的树形视图，如图 5-8 所示。在树形视图中，每个表项显示一个标题（Label），有时还会显示一幅图像，图像和标题分别提供了对数据的形象和抽象描述。通过图 5-8 可以看出，树形视图可以很清楚地显示出数据的分支和层次关系。由此可见，树形视图非常适合显示目录、网络结构等这样的复杂数据。传统的列表框不能分层显示数据，因此树形视图可以看作是对列表框的一种重要改进。

树形视图是一种复杂的控件，它的复杂性体现在数据项之间具有分支和层次关系。例如，如果要向树形视图中加入新的项，则必需描述出该项与树形视图中已有项的相互关系，而不可能像往列表框中加入新项那样，调用 AddString 即可。另外，树形视图可以在每一项标题的左边显示一幅图像，这使控件显得更加形象生动，但同时也增加了控件的复杂程度。在介绍如何使用树形视图控件之前，有必要先介绍一下与该控件有关的一些数据类型。

图 5-8 CTreeCtrl 控件样例

1．HTREEITEM 型句柄

在 Windows 中用 HTREEITEM 型句柄来代表树形视图的一项，通过 HTREEITEM 句柄来区分和访问树形视图的各个项。

2．TV\_ITEM 结构

该结构用来描述一个表项，它包含了表项的各种属性，其定义如下：

typedef struct \_TV\_ITEM

{

UINT mask;

//包含一些屏蔽位（下面的括号中列出）的组合，用来表明结构的哪些成员是有效的

HTREEITEM hItem;

//表项的句柄(TVIF\_HANDLE)

UINT state;

//表项的状态(TVIF\_STATE)

UINT stateMask;

//状态的屏蔽组合(TVIF\_STATE)

LPSTR pszText;

//表项的标题正文(TVIF\_TEXT)

int cchTextMax;

//正文缓冲区的大小(TVIF\_TEXT)

int iImage;

//表项的图像索引(TVIF\_IMAGE)

int iSelectedImage;

//选中的项的图像索引(TVIF\_SELECTEDIMAGE)

int cChildren;

//表明项是否有子项(TVIF\_CHILDREN)，为 1 则有，为 0 则没有

LPARAM lParam;

//一个 32 位的附加数据(TVIF\_PARAM)

}TV\_ITEM,FAR \*LPTV\_ITEM;

下面对上述参数进行解释。如果要使树形视图的表项显示图像，需要为树形视图建立一个位图序列，这时，iImage 说明表项显示的图像在位图序列中的索引，iSelectedImage 则说明了选中的表项应显示的图像，在绘制图标时，树形视图可以根据这两个参数提供的索引在位图序列中找到对应的位图。lParam 可用来放置与表项相关的数据，这常常是很有用的。state 和 stateMask 的常用值在表 5-22 中列出，其中 stateMask 用来说明要获取或设置哪些状态。

表 5-22 列表视图的状态

|  |  |  |
| --- | --- | --- |
| 状态 | 对应的状态屏蔽 | 含义 |
| TVIS\_SELECTED | 同左 | 项被选中 |
| TVIS\_EXPANDED | 同左 | 项的子项被展开 |
| TVIS\_EXPANDEDONCE | 同左 | 项的子项曾经被展开过 |
| TVIS\_CUT | 同左 | 项被选择用来进行剪切和粘贴操作 |
| TVIS\_FOCUSED | 同左 | 项具有输入焦点 |
| TVIS\_DROPHILITED | 同左 | 项成为拖动操作的目标 |

3．TV\_INSERTSTRUCT 结构

TV\_INSERTSTRUCT 结构。在向树形视图中插入新项时要用到该结构，其定义如下：

typedef struct \_TV\_INSERTSTRUCT{

HTREEITEM hParent;

//父项的句柄

HTREEITEM hInsertAfter;

//说明应插入到同层中哪一项的后面

TV\_ITEM item;

}TV\_INSERTSTRUCT;

如果 hParent 的值为 TVI\_ROOT 或 NULL，那么新项将被插入到树形视图的最高层（根位置）。hInsertAfter 的值可以是 TVI\_FIRST、TVI\_LAST 或 TVI\_SORT，其含义分别是将新项插入到同一层中的开头、最后或排序插入。

4．NM\_TREEVIEW 结构

树形视图的大部分通知消息都会附带指向该结构的指针以提供一些必要的信息。该结构

的定义如下：

typedef struct \_NM\_TREEVIEW {

NMHDR hdr;

//标准的 NMHDR 结构

UINT action;

//表明是用户的什么行为触发了该通知消息

TV\_ITEM itemOld;

//旧项的信息

TV\_ITEM itemNew;

//新项的信息

POINT ptDrag;

//事件发生时鼠标的客户区坐标

}NM\_TREEVIEW;

5．TV\_KEYDOWN 结构

提供与键盘事件有关的信息。该结构的定义如下：

typedef struct \_TV\_KEYDOWN {

NMHDR hdr;

//标准的 NMHDR 结构

WORD wVKey;

//虚拟键盘码

UINT flags; //为 0

}TV\_KEYDOWN;

6．TV\_DISPINFO 结构

提供与表项的显示有关的信息。该结构的定义如下：

typedef struct \_TV\_DISPINFO {

NMHDR hdr;

TV\_ITEM item;

}TV\_DISPINFO;

MFC 的 CTreeCtrl 类封装了树形视图。该类的 Create 成员函数负责控件的创建，该函数的声明如下：

BOOL Create(DWORD dwStyle, const RECT& rect,CWnd\* pParentWnd,UINT nID);

其中参数 dwStyle 是如表 5-23 所示的控件风格的组合。

表 5-23 树型控件风格

|  |  |
| --- | --- |
| 控件风格 | 含义 |
| TVS\_HASLINES | 在父项与子项间连线以清楚地显示结构 |
| TVS\_LINESATROOT | 只在根部画线 |
| TVS\_HASBUTTONS | 显示带有＂+＂或＂-＂的小方框来表示某项能否被展开或已展开 |
| TVS\_EDITLABELS | 用户可以编辑表项的标题 |
| TVS\_SHOWSELALWAYS | 即使控件失去输入焦点，仍显示出项的选择状态 |
| TVS\_DISABLEDRAGDROP | 不支持拖动操作 |

除上表的风格外，一般还要指定 WS\_CHILD 和 WS\_VISIBLE 窗口风格。对于用对话框模板创建的树形视图控件，可以在控件的属性对话框中指定上表中列出的控件风格。例如，在属性对话框中选择 Has buttons，相当于指定了 TVS\_HASBUTTONS 风格。

### 5.9.2 CtreeCtrl 类的主要成员函数

CTreeCtrl 类提供了大量的成员函数。对于常用的函数，下面结合实际应用进行介绍。

1．SetImageList()函数

向树形视图中插入新的表项。首先应提供一个 TV\_INSERTSTRUCT 结构并在该结构中对插入项进行描述。如果要在树形视图中显示图像，则应该先创建一个 CImageList 对象并使该对象包含一个位图序列，然后调用 SetImageList 为树形视图设置位图序列。最后调用

InsertItem 函数把新项插入到树形视图中。该函数的声明如下：

CImageList\* SetImageList(CImageList \* pImageList,int nImageListType);

参数pImageList指向一个CImageList对象，参数nImageListType一般应为TVSIL\_NORMAL。

2．InsertItem()函数

该成员函数声明如下：

HTREEITEM InsertItem( LPTV\_INSERTSTRUCT lpInsertStruct );

参数 lpInsertStruct 指向一个 TV\_INSERTSTRUCT 结构。函数返回新插入项的句柄。

3．DeleteItem()和 DeleteAllItems()函数

用 DeleteItem()函数来删除指定项，用 DeleteAllItems()函数删除所有项。函数的声明如下：

BOOL DeleteItem(HTREEITEM hItem);

BOOL DeleteAllItems();

操作成功则函数返回 TRUE，否则返回 FALSE。

4．Expand()函数

树形视图控件会根据用户的输入自动展开或折叠子项。但有时需要在程序中展开或折叠

指定项，则应该调用 Expand，该函数的声明为

BOOL Expand(HTREEITEM hItem,UINT nCode);

参数 hItem 指定了要展开或折叠的项。参数 nCode 是一个标志，指定了函数应执行的操作，它可以是 TVE\_COLLAPSE（折叠）、TVE\_COLLAPSERESET(折叠并移走所有的子项）、 TVE\_EXPAND（展开）或 TVE\_TOGGLE（在展开和折叠状态之间翻转）。

5．GetSelectedItem()和 SelectItem()函数

要查询或设置选择项，应调用 GetSelectedItem 或 SelectItem。函数的声明如下：

HTREEITEM GetSelectedItem();

BOOL SelectItem(HTREEITEM hItem);

6．GetItem()和 SetItem()函数

要查询或设置指定的项，可调用 GetItem()和 SetItem()函数。用这两个函数，几乎可以查

询和设置项的所有属性，包括表项的正文、图像及选择状态。函数的声明如下：

BOOL GetItem(TV\_ITEM\* pItem);

BOOL SetItem(TV\_ITEM\* pItem);

参数 pItem 是指向 TV\_ITEM 结构的指针，函数通过该结构来查询或设置指定的项，在调用函数前应该使该结构的 hItem 成员有效以指定表项。CtreeCtrl 类还提供了一系列函数可完成 GetItem 和 SetItem 的部分功能，其中 GetItemState、GetItemText、GetItemData、 GetItemImage 和 ItemHasChildren 函数用于查询，SetItemState、SetItemText、SetItemData 和 SetItemImage 函数用于设置。

7．GetNextItem()函数

在使用树形视图控件时，一个经常遇到的问题是对于一个已知表项，如何找到与该项有某种关系的项，例如，父项、子项、兄弟项、下一个或前一个可见的项。利用 GetNextItem() 函数可以解决这个问题，该函数也可以用来搜索具有某种状态的表项。GetNextItem()函数在遍历树形视图时是很有用的，它的声明如下：

HTREEITEM GetNextItem(HTREEITEM hItem,UINT nCode);

参数 hItem 指定了一个项。参数 nCode 是一个标志，标明了与指定项的关系，nCode 可以是如表 5-24 所示的各种标志。如果找到相关的项，函数返回该项的句柄，否则函数返回

NULL。

表 5-24 树型控件中 nCode 的取值

|  |  |
| --- | --- |
| 风格 | 含义 |
| TVGN\_CARET | 返回当前的选择项 |
| TVGN\_CHILD | 返回指定表项的子项 |
| TVGN\_DROPHILITE | 返回拖动操作的目标项 |
| TVGN\_FIRSTVISIBLE | 返回第一个可见项 |
| TVGN\_NEXT | 返回指定项的下一个兄弟项（Sibling Item） |
| TVGN\_NEXTVISIBLE | 返回指定项的后一个可见项 |
| TVGN\_PARENT | 返回指定项的父项 |
| TVGN\_PREVIOUS | 返回指定项的前一个兄弟项 |
| TVGN\_PREVIOUSVISIBLE | 返回指定项的前一个可见项 |
| TVGN\_ROOT | 返回位于最高层（根位置）的第一个表项 |

CTreeCtrl 类提供了一系列的成员函数来完成 GetNextItem 的某一项功能，包括

GetRootItem()、GetFirstVisibleItem()、GetNextVisibleItem()、GetPrevVisibleItem()、GetChildItem()、 GetNextSiblingItem() 、 GetPrevSiblingItem() 、 GetParentItem() 、 GetSelectedItem() 和

GetDropHilightItem()。

### 5.9.3 树型控件的通知消息

树形视图控件还可发送自己特有的通知消息，其中常用的有下面这几个。

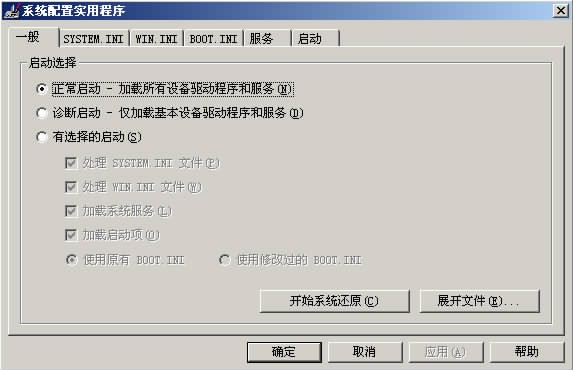
* TVN\_SELCHANGING 和 TVN\_SELCHANGED。在用户改变了对表项的选择时，控件会发送这两个消息。消息会附带一个指向 NM\_TREEVIEW 结构的指针，程序可从该结构中获得必要的信息。两个消息都会在该结构的 itemOld 成员中包含原来的选择项信息，在 itemNew 成员中包含新选择项的信息，在 action 成员中表明是用户的什么行为 触 发 了 该 通 知 消 息 ( 若 是 TVC\_BYKEYBOARD 则 表 明 是 键 盘 ， 若 是 TVC\_BYMOUSE 则表明是鼠标，若是 TVC\_UNKNOWN 则表示未知）。两个消息的不同之处在于，如果 TVN\_SELCHANGING 的消息处理函数返回 TRUE，那么就阻止选择的改变，如果返回 FALSE，则允许改变。
* TVN\_KEYDOWN。该消息表明了一个键盘事件。消息会附带一个指向 TV\_KEYDOWN 结构的指针，通过该结构程序可以获得按键的信息。
* TVN\_BEGINLABELEDIT 和 TVN\_ENDLABELEDIT。分别在用户开始编辑和结束编辑项的标题时发送。消息会附带一个指向 TV\_DISPINFO 结构的指针，程序可从该结构中获得必要的信息。在前者的消息处理函数中，可以调用 GetEditControl()成员函数返回一个指向用于编辑标题的编辑框的指针。如果处理函数返回 FALSE，则允许编辑，如果返回 TRUE，则禁止编辑。在后者的消息处理函数中，TV\_DISPINFO 结构中的 item.pszText 指向编辑后的新标题，如果 pszText 为 NULL，那么说明用户放弃了编辑，否则，程序应负责更新项的标题，这可以由 SetItem()或 SetItemText()函数来完成。

## 5.10 选项卡（Tab Ctrl）控件

### 5.10.1 选项卡控件的使用

Tab 属性页控件可以在一个窗口中添加不同的页面，然后在页选择发生改变时得到通知。

典型的选项卡控件，如图 5-9 所示。

 图 5-9 CTabCtrl 控件样例

MFC 中使用 CTabCtrl 类来封装属性页控件的各种操作。通过调用 Create 函数创建一个窗口，Create 函数声明如下：

BOOL Create(DWORD dwStyle,const RECT& rect,CWnd\* pParentWnd,UINT nID)

其中 dwStyle 中可以使用以下一些属性页控件的专用风格。

* TCS\_BUTTONS：使用按钮来表示页选择位置。
* TCS\_MULTILINE：分行显示页选择位置。
* TCS\_SINGLELINE：只使用一行显示页选择位置。

在控件创建后必需向其中添加页面才可以使用，添加页面的函数为：

BOOL InsertItem(int nItem,LPCTSTR lpszItem);

其中 nItem 为位置，从 0 开始，lpszItem 为页选择位置上显示的文字。如果希望在页选择位置处显示一个图标，可以调用 InsertItem（在此之前必须调用 SetImageList 设置正确的

ImageList）。

对组成选项卡（TabControl）控件（CTabCtrl）选项卡的外观和行为可以有相当多的设置方法。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，应调用 GetItem()函数检索它，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，应调用 DeselectAll() 函数。该函数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）。下列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);

### 5.10.2 选项卡控件通知消息

当单击选项卡或按钮时，选项卡（TabControl）控件（CTabCtrl）向其父窗口发送通知消息。如果要进行某种响应，应对这些消息进行处理。例如，当单击选项卡时，也许想在显示选项卡之前在页面上预设控件数据，即响应单击选项卡消息时作出预设数据的响应。

处理来自视图或对话框类中的选项卡（TabControl）控件的 WM\_NOTIFY 消息，用“属性”窗口创建带 switch 语句的 OnChildNotify()处理函数。选项卡控件向父窗口发送的主要通知消息如表 5-25 所示。

表 5-25 选项卡控件的通知消息

|  |  |
| --- | --- |
| 通知消息 | 含义 |
| NM\_CLICK | 通知父窗口用户在控件区域范围内点击了鼠标左键 |
| NM\_RCLICK | 通知父窗口用户在控件区域范围内点击了鼠标右键 |
| NM\_RELEASEDCAPTURE | 通知父窗口在控件区域范围内释放鼠标捕获消息 |
| TCN\_FOCUSCHANGE | 通知父窗口控件的按钮聚焦已经改变 |
| TCN\_GETOBJECT | 当具有 TCS\_EX\_REGISTERDROP 扩展特性时并且对象被拖动的通知消息 |
| TCN\_KEYDOWN | 通知父窗口在控件范围内键盘被按下 |
| TCN\_SELCHANGE | 通知父窗口控件的选项卡选择项已经改变 |
| TCN\_SELCHANGING | 通知父窗口控件的选项卡选择项正在改变 |

选项卡控件还涉及到一些重要的数据结构，这些预定义的数据结构在该控件应用中起着

重要作用。主要涉及到的的结构如下。

1．NMTCKEYDOWN 结构

该结构包含键盘按下时的相关信息，主要用于处理 TCN\_KEYDOWN 通知消息，该结构

定义如下：

typedef struct tagNMTCKEYDOWN {

NMHDR hdr;

WORD wVKey;

UINT flags;

} NMTCKEYDOWN

2．TCHITTESTINFO 结构

该结构主要用于鼠标单击测试的通知消息，定义如下：

typedef struct tagTCHITTESTINFO {

POINT pt;

UINT flags;

} TCHITTESTINFO, \*LPTCHITTESTINFO;

3．TCITEM 结构

该结构指定选项卡控件选项的具体属性，定义如下：

typedef struct tagTCITEM { UINT mask;

#if (\_WIN32\_IE >= 0x0300)

DWORD dwState;

DWORD dwStateMask;

#else

UINT lpReserved1; UINT lpReserved2;

#endif

LPTSTR pszText;

int cchTextMax; int iImage;

LPARAM lParam;

} TCITEM, \*LPTCITEM;

4．TCITEMHEADER 结构

该结构用于指定或接收选项卡控件本身的属性，定义如下：

|  |
| --- |
| typedef struct tagTCITEMHEADER {  UINT mask;  UINT lpReserved1;  UINT lpReserved2; LPTSTR pszText; int cchTextMax; int iImage;  } TCITEMHEADER, \*LPTCITEMHEADER; |

此外 CTabCtrl 还提供了一些函数用于得到和修改控件的状态：

int GetCurSel()/int SetCurSel(int nItem);//用于得到/设置当前被选中的页位置 BOOL DeleteItem(int nItem)/BOOL DeleteAllItems();//用于删除指定/所有页面 void RemoveImage(int nImage);//用于删除某页选择位置上的图标

在当前页发生改变时需要隐藏当前的一些子窗口，并显示其他的子窗口。下面的代码演示了如何实现上述功能，主要的代码如下：

|  |
| --- |
| CParentWnd::OnCreate(...)  {  m\_tab.Create(...);  m\_tab.InsertItem(0,"Option 1"); m\_tab.InsertItem(1,"Option 2");  Create a edit box as the m\_tab’s Child  Create a static box as the m\_tab’s Child  edit\_box.ShowWindow(SW\_SHOW); // edit box 在属性页的第一页 static\_box.ShowWindow(SW\_HIDE); // static box 在属性页的第二页  }  void CParentWnd::OnSelectChangeTab(NMHDR\* pNMHDR, LRESULT\* pResult)  {  //处理页选择改变后的消息  if(m\_tab.GetCurSel()==0)  {//根据当前页显示/隐藏不同的子窗口  edit\_box.ShowWindow(SW\_SHOW); static\_box.ShowWindow(SW\_HIDE);  }  else |

|  |  |
| --- | --- |
| } | {//  edit\_box.ShowWindow(SW\_HIDE); static\_box.ShowWindow(SW\_SHOW);  } |

|  |
| --- |
| 技巧：获取对话框中控件的窗口指针并进行一些常用的操作对于对话框中的控件，实际上是对话框的一个子窗口。在实际编程中，可能常常会对控件进行一些操作，得到该控件窗口的指针是第一步，通过 CWnd::GetDlgItem 可以很方便地得到控件窗口的指针。假设某对话框上有一个 ID 为 IDC\_XXX\_STATIC 的静态文本控件，可以通过下面这行代码重新设置文本内容：  GetDlgItem(IDC\_XXX\_STATIC)->SetWindowText(…)  对于其他操作窗口的常用操作，也可以通过 GetDlgItem 传回的指针来调用对应的窗口操作函数来完成。 |

## 5.11 本章小结

本章对 MFC 编程中常常用到的控件进行了介绍，包括传统的 Windows 控件和新型的 Win32 控件。在传统控件中主要介绍了静态文本控件、文本编辑控件、滚动条控件、按钮控件（包括命令按钮、单选按钮、复选按钮）、列表框控件、组合框控件、图片控件等。在新型控件中主要介绍了列表控件、树状控件、选项卡控件。本章对这些控件本身的特性、使用方法以及主要的成员函数进行了介绍，掌握这些控件的使用方法是构建 Windows 应用程序的前提条件，因为在用户界面程序设计中是离不开控件的使用的。

关于控件的创建和使用总结起来的要点如下。

* 传统控件的通知消息一般是通过 WM\_COMMAND 消息发给父窗口的，Win32 控件的通知消息则是通过 WM\_NOTIFY 消息发送的，该消息可以附带大量信息。
* 由于所有的控件都是子窗口，所有的控件类都是 CWnd 类的派生类。因此可以用

ShowWindow、EnableWindow 和 MoveWindow 这样的 CWnd 成员函数来控制控件。

* 控件的创建有自动和手工两种常用方法。 控件的自动创建是通过向对话框模板中添加控件实现的。手工创建则需要构建一个控件对象并调用相应的 Create 函数。
* 访问控件的方法包括：通过对话框的数据交换功能来查询和设置控件；通过控件对象来访问控件；利用 CWnd 类的一些用于管理控件的成员函数来访问控件。

虽然本章对各种控件的功能和使用方法有一个较为全面的介绍，但是读者在具体使用某个控件时可能还会有更多自己的功能需求，这时可以通过 Visual C++的联机帮助获得更多的与该控件相关的帮助信息。

# 第 6 章 文本和字体类

Windows 经常使用 GDI 进行文本和图形输出。实际上，文本和图形并没有明显的界限，在很多时候，Windows 把文本当作一种特殊的图形来对待，即，文本被看作是按照指定的“字体”格式画出来的。字体用于描述字符集中每一个字母、数字和标点符号的形状及外表的特殊信息。

本章将对字体类 Cfont 及文本的输出过程进行介绍。

## 6.1 基础知识介绍

当任何窗口被第一次建立时，它假定是使用默认的系统字体。事实上，除非另外指定，默认字体也就是系统字体。使用字体的方法很简单，主要是通过 WM\_GETFONT 消息或者函数 CreateFont()取得字体。消息 WM\_GETFONT 经常被发送到窗口以确定字体的类型，该字体可能在它的窗口中作为默认值。此消息返回字体句柄或者空（如果正使用系统字体）。系统

字体的实际句柄可由调用 GetStockObject(SYSTEM\_FONT)得到。

在绘制文本时，不仅需要考虑描绘文本在窗口中使用的颜色，还应考虑文本的背景颜色和文本描绘“模式”，表 6-1 所示为控制颜色和描绘文本模式所需的函数。

表 6-1 控制颜色和模式的函数

|  |  |
| --- | --- |
| 函数 | 描述 |
| GetBkColor() | 传递显示描述表时，返回 COLORREF 值指示描绘文本时使用的背景色 |
| SetBkColor() | 传递显示描述表和 COLORREF 值，设置绘制文本的背景颜色 |
| GetTextColor() | 传递显示描述表时，返回 COLORREF 值，指示绘制文本的颜色 |
| SetTextColor() | 传递显示描述表和 COLORREF 时，为绘制的文本设置颜色 |
| GetBkMode() | 传递显示描述表时，返回绘画方式，可以是 OPAQUE 或者 TRANSPARENT |
| SetBkMode() | 传递显示措述表和 OPAQUE 或 TRANSPARENT 参数时设置背景文本绘制方式 |

## 6.2 CFont 类

### 6.2.1 概述

CFont 类封装了一个 Windows 图形设备接口（GDI）字体并提供管理字体的成员函数。为使用 Cfont 类，可构造一个 CFont 对象并用 CreatFont()、CreateFontIndirect()、CreatePointFont() 或 CreatePointFontIndirect()将一个 Windows 字体附加给对象，然后用对象的成员函数操纵字体。使用 CreatePointFont()和 CreatePointFontIndirect()常比使用 CreateFontIndirect()更容易，因为前者可自动将字体高度的单位从点转换为逻辑单位。CFont 类在 MFC 类库中的继承关系如图 6-1 所示。

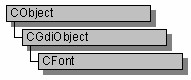


图 6-1 CFont 类在 MFC 类库中的位置

### 6.2.2 主要成员函数

CFont 类中封装了与字体集相关的成员函数，通过这些成员函数可以方便地设置文本输出的字体。Cfont 类中的主要成员函数如表 6-2 所示。

表 6-2 Cfont 类中的成员函数

|  |  |
| --- | --- |
| 成员 | 作用 |
| CFont | 构造一个 CFont 对象初始化 |
| CreateFontIndirect | 初始化一个由 LOGFONT 结构给出其特征的 CFont 对象 |
| CreateFont | 初始化用指定特性定义的 CFont 对象 |
| CreatePointFont | 用指定高度（用 0.1 点）和字体初始化一个 CFont 对象 |
| CreatePointFontIndirect | 与 CreateFontIndirect 相似，但字体高度用 0.1 点定义而不用逻辑单位定义操作 |
| FromHandle | 当给定一个 Windows HFONT 时，返回一个指向 CFont 对象的指针 |
| HFONT | 返回附加给 CFont 对象的 Windows GDI 字体句柄 |
| GetLogFont | 用附加给 CFont 对象的逻辑字体信息填充一个 LOGFONT |

下面详细介绍常用的成员函数。

1．CFont::CreateFont()函数

可以调用该函数创建自定义字体，该函数声明如下：

BOOL CreateFont

(

int nHeight, int nWidth, int nEscapenment, int nOrientation, int nWeight,

BYTE bItalic,

BYTE bUnderline,

BYTE cStrikeOut,

BYTE nCharset,

BYTE nOutPrecision,

BYTE nClipPrecision,

BYTE nQuality,

BYTE nPitchAndFamily,

LPCTSTR lpszFacename

);

（1）nHeight

以逻辑单位方式指定字体的高度，字体高度可为以下值之一：

* 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，则字体映像器会寻找最大的或者最小的不超过所需大小的字体。

1. nWidth

指定字体中字符平均宽度（用逻辑单位）。如果为 0，设备方向比率与可用字体的数字方向比率相比较，找一个最近的匹配，这个最近的匹配由值的差的绝对值决定。

1. nEscapement

指定偏离垂线与 *x* 轴在显示面上的夹角（用 0.1 度单位）。偏离垂线时从一行中开始一个字符到最后一个字符的线，此角从 *x* 轴逆时针方向度量。

1. nOrientation

指定字符基线和 *x* 轴之间的夹角（用 0.1 度单位）。此度数在坐标轴中由 *x* 轴顺时针度量时坐标系中 *y* 轴向下，顺时针方向从 *x* 轴旋转时，*y* 轴向上。

1. nWeight

指定字体磅数（用每 1000 点中墨点像素数计）。nWeight 可为 0 到 1000 中的任意整数值。

1. bItalic

指定字体是否为斜体。

1. bUnderline

指定字体是否带下划线。

1. bStrikeOut

指定是否突出显示字符。如果设置为非零，则突出。

1. nCharSet

指定字体的字符集，预定义预定义字符集如表 6-3 所示。

表 6-3 预定义字符集

|  |  |  |  |
| --- | --- | --- | --- |
| 常数 |  |  | 值 |
| ANSI\_CHARSET | 0 |  |  |
| DEFAULT\_CHARSET | 1 |  |  |
| SYMBOL\_CHARSET | 2 |  |  |
| SHIFTJIS\_CHARSET | 1280 |  |  |
| EM\_CHARSET | 255 |  |  |

OEM 字符集是和系统相关联的，而具有其它字符集的字体也可能存在于系统之中。一个使用未知字符集字体的应用不能翻译或解释一个以此字符集着色的字符串，相反，应将字符串直接输入到设备驱动。字体映射器不使用 DEFAULT\_CHARSET 值，一个应用可以使用此值让字体名和大小完全描述逻辑字体。如果指定的名字不存在，为避免不可预料的结果，应用应谨慎地使用 DEFAULT\_CHARSET。

1. nOutPrecision

指定所需的输出精度。输出精度可定义需要的字体高度、宽度，字符方向、走格、间距之间 的 接 近 程 度 。 系 统 包 含 多 个 给 定 字 体 时 ， 可 用 OUT\_DEVICE\_PRECIS 、 OUT\_RASTER\_PRECIS 和 OUT\_TT\_PRECIS 值控制字体映射器如何选择字体。例如，如果一个系统包含一个名叫 Symbol 的字体，以光栅和 TrueType 形式存在，通过 OUT\_TT\_PRECIS 可使字体映射器选择 TrueType 类型（指定 OUT\_TT\_PRECIS 强制字体映射器选择 TrueType 字体）。

1. nClipPrecision

指定所需的剪贴精度。剪贴精度用于定义如何剪贴部分超过范围的字符。要使用插入的只读字体，必须指定 CLIP\_ENCAPSULATE。要建立设备旋转、TrueType 和矢量字库，可以用 OR 操作符将 CLIP\_LH\_ANGLES 值与其他 nClipPrecision 值组合。如果 CLIP\_LH\_ANGLES 位被设置，所有字体的旋转都根据坐标系的定位是左手方向还是右手方向来决定（要了解有关坐标轴定位的更多信息，请参阅 nOrientation 参数的描述）。如果 CLIP\_LH\_ANGLES 未被设置，设备字体按逆时针方向旋转，但其他字体的旋转依赖于坐标系的旋转定位。

1. 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。黑体、斜体、下划线、突出字体和综合处理在需要时可以综合。
* 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。黑体、斜体、下划线、突出和综合处理在需要时可以综合。

（13）nPitchAndFamily

指定字体的间距和家族。两个低位指定字体的间距，值可为：DEFAULT\_PITCH 或 VARIABLE\_PITCH。FIXED\_PITCH 应用可以将 T\MPF\_TRUETYPE 加到 nPitchAndFamily 参数中，并选择一个 TrueType 字体。高四位指定字体家族，可为以下值之一。

* FF\_DECORATIVE：新奇字体，如以前英格兰的字体。
* FF\_DONTCARE：不在意或不知道。
* FF\_MODERN：笔型宽度不变的字体，有或无衬线。固定斜度的字体常是现代风格的，

如 Pica,Elite 和 Courier New。

* FF\_ROMAN：笔划宽度可变（接比例调整空间）及有衬线字体。如 Times New Roman

和 Century Schoolbook。

* FF\_SCRIPT：与手写体相似的字体，如 Script 和 Cursive。
* FF\_SWISS：笔划宽度可变（接比例调整空间）及不带有衬线字体，如 MS Sans Serif。可用布尔操作 OR 指定一个 nPithAndFamily 值，以组合一个斜体和一个家族常数。字体家族用普通方式描述了字体的外观，它们在所需铅字体无效时定义指定的字体。

（14）pszFacename

指定字体样式，可为 Cstring 型，或一个一空终止符结尾的字符串。此字符串的长度不能超过 30 个字符，Windows CE 中的 EnumFontFamilies 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。

总的说来，CreateFont 成员函数用于以指定的特征初始化一个 CFont 对象。此字体可被选作任何设备上下文的字体。CreateFont 函数不创建一个新 Windows GDI 字体，只是从 GDI 物理字体库中选择最相近的匹配字体。当构造一个逻辑字体时，大部分参数可使用默认值。如果没有给定 nHeight 和 lpszFace Name，则逻辑字体是与设备有关的。当用 CreateFont 函数完成了一个 CFont 对象的创建时，先从设备环境中选出字体，然后删除 CFont 对象（字体定义完后就删除对象，以节省空间）。

2．CFont::CreateFontIndirect()函数

该成员函数以间接方式生成一个自定义字体，声明如下：

BOOL CreateFontIndirect(const LOGFONT\* lpLogFont);

其中参数 lpLogFont 指向一个定义逻辑字体特征的 LogFont 结构。该函数用 lpLogFont 指向的 LOGFONT 结构定义的特征初始化一个 CFont 对象，之后此字体即可被任何设备选作当前字体。此字体具有 LOGFONT 结构定义的特征。当使用 CDC::SelectObject 成员函数选择时，GDI 字体映射器将努力将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。当用 CreateFontIndirect 函数完成 CFont 对象的创建后，先在设备环境中选出字体，然后删除 CFont 对象。

3．CFont::CreatePointFont()函数

此函数用于创建指定字体和点的大小，声明如下：

BOOL CreatePointFont(int nPointSize,LPCTSTR lpszFaceName,CDC\*pDC=NULL);

参数意义如下：

* nPointSize：指定所需字体高度（用 0.1 点表示，例如，传递 120 表示 12 点字体）。
* lpszFaceName：定义字体名称，为 Cstring 型或指向一个以空终止符结尾的字符串。此字符串长度不应超过 30 个字符。Windows CE 中的 EnumFontFamilies 函数可用于计算当前可用字体的总数。如果为 NULL，则 GDI 使用与设备无关的字体。
* pDC：指向 CDC 对象，将 nPointSize 中的高度转化为逻辑单位。如果为 NULL，则屏幕设备环境用于转换。

该函数自动将 nPointSize 中的高度转化为逻辑单位，这是使用由 pDC 指向的 CDC 对象实现的。当完成用 CreatePointFont 函数创建 CFont 对象后，从设备环境中选择字体，然后删除 CFont 对象。

4．CFont::CreatePointFontIndirect()函数

函数提供了一种创建指定字体和点大小的字体对象的间接方法，声明如下：

BOOL CreatePointFontIndirect(const LOGFONT\* lpLogFont,CDC\*pDC = NULL);

参数意义如下：

* lpLogFont：指向 LOGFONT 结构，定义逻辑字体特征。LOGFONT 的 lfHeight 成员用

0.1 点计算而不用逻辑单位（如果设置 lfHeight 为 120，则要求 12 点字体）。

* pDC：指向 CDC 对象，用于将 lfHeight 中的高度转化为逻辑单位。如果为 NULL，屏幕设备环境用于转换。

此函数与 CreateFontIndirect 相似，但 LOGFONT 的 lfHeight 成员由 0.1 点表示而不是用设备单位表示。在传递 LOGFONT 结构到 Windows 之前，pDC 指向的 CDC 对象自动使函数的 lfHeight 中的高度转化为逻辑单位。当完成由 CreatePointFontIndirect 函数创建的 CFont 对象之后，先选择设备环境字体，然后删去 CFont 对象。

5．CFont::FromHandle()函数

该静态函数用于从Windows GDI的字体句柄得到对应CFont对象指针的方法，声明如下：

static CFont\* PASCAL FromHandle (HFONT hFont);

参数 hFont 代表 Windows 字体中的 HFONT 句柄。

当给定一个 Windows GDI 字体对象的 HFONT 句柄时，返回一个指向 CFont 对象的指针。如果 CFont 对象还未附加给句柄，将创建一个临时 CFont 对象，此临时 CFont 对象仅在事件循环中有空闲时才可用，此时临时图形对象被删除，换一种说法就是临时对象仅在一个窗口消息进程中才可用。

## 6.3 文本输出过程

在定义了字体句柄、字体及字体颜色以后，就可以把设置的字体输出到相应的设备中。

Windows 应用程序的文本输出过程比较复杂，除了要确定输出内容外，还要管理输出的格式，由应用程序完成窗口用户区管理，Windows 系统并不参与窗口用户区的管理，这样就加重了编写应用程序的负担。例如，在用户区内输出文本时，应用程序必须管理换行、后续字符的位置等输出格式，Windows 系统并东提供管理输出文本格式的函数。

文本的输出过程包括获取字体信息、格式化文本、调用函数输出文本等过程。

1．获取字体信息

应用程序在输出文本之前必须获取当前使用字体的有关信息，如当前使用的字符高度，以确定输出文本格式和下一行字符的输出位置。

Windows 程序中通常调用函数 GetTextMetrics 获取当前使用的字体信息。调用该函数时，系统将当前字体的信息拷贝到一个 TEXTMETRICS 结构的变量中。其形式为：

GetTextMetrics(hdc,&tm); //tm 为 TEXTMETRICS 结构

系统定义的 TEXTMETRICS 结构如下：

typedef struct tagTEXTMETRIC {

LONG tmHeight; //字符向度

LONG tmAscent; //字符基线以上高度 LONG tmDescent; //字符基线以下高度

LONG tmInternalLeading; //tmHeight 制订的字符高度顶部的拉件

LONG tmExternalLeading; //行与行之间的间隔

LONG tmAveCharWidth; //平均字符宽度

LONG tmMaxCharWidth; //最大字符宽度

LONG tmWeight; //字符的粗细度

LONG tmOverhang; //合成字体间附加的宽度

LONG tmDigitizedAspectX; //为输出设备设计的 *x* 轴尺寸

LONG tmDigitizedAspectY; //为输出设备设计的 *y* 轴尺寸

TCHAR tmFirstChar; //字体中第一个字符值

TCHAR tmLastChar; //字体中最后一个字符值

TCHAR tmDefaultChar; //替换字体中没有的字符

TCHAR tmBreakChar; //作为分隔符的字符

BYTE tmItalic; //非 0 则表示字体为斜体

BYTE tmUnderlined; //非 0 则表示字体有下划线

BYTE tmStruckOut; //非 0 则表示字符为删除字体

BYTE tmPitchAndFamily; //字体间距和字体族

BYTE tmCharSet; //字符集

} TEXTMETRIC, \*PTEXTMETRIC;

调用函数 GetTextMetrics 获取当前字体的 TEXTMETRICS 结构后，即可为其中的成员设

置文本输出格式。

2．格式化文本

格式化处理一般完成两件事情：一是在文本行中确定后续文本的坐标，二是在换行时确

定下一行文本的坐标。

（1）确定后续文本坐标确定后续文本的坐标时，应先获取当前的字符串的宽度，Windows 系统提供了函数

GetTextExtentPoint32 完成这项任务，并把它存储于一个 SIZE 结构中。该函数的原型为：

|  |
| --- |
| BOOL GetTextExtentPoint32(  HDC hdc, // DC 句柄 LPCTSTR lpString, // 指定字符串 int cbString, // 字符串中的字符数目  LPSIZE lpSize // 返回字符串宽度及高度的 SIZE 数据结构的地址  ); |

SIZE 数据结构定义如下：

|  |
| --- |
| typedef struct tagSIZE {  LONG cx; LONG cy;  } SIZE, \*PSIZE; |

通过计算字符串的起始坐标与字符串宽度之和，即可得到后续文本的起始坐标。例如，

*X* 轴起始坐标为 cx0，如果当前字符串的尺寸信息存储在 size 指向的 SIZE 结构中，则后续文本的起始坐标 cx1 为：

cx1 = cx0 + size.cx;

（2）确定换行时的文本坐标通过计算当前行文本字符的高度与行间隔之和，即可得到换行时文本的起始坐标，上述两个数值可通过获取当前字体的信息得到，如果当前字体的信息存储在 tm 指向的

TEXTMETRICS 结构中，则换行时 *Y* 轴上文本的起始坐标 cy 为：

cy = tm.tmHeight + tm.tmExternalLeading;

3．文本输出

Windows 程序设计时用得最多的文本输出函数便是 Textout，其 API 的函数原型如下：

BOOL TextOut(

HDC hdc, // DC 句柄 int nXStart, // 起始点 *X* 坐标 int nYStart, //起始点 *Y* 坐标 LPCTSTR lpString, // 输出的字符串 int cbString // 输出的字符串中字符数目

);

通过调用此函数，可以在指定位置输出文本。

## 6.4 字体和文本输出实例

实例 6-1：字体和文本输出实例。源代码在光盘中“\06\实例 6-1\gundong”目录下。

本节实现了一个动态字幕的例子。从本例中读者可以学习到动态地改变文本坐标来实现实现动态字幕的方法。该示例是基于对话框模式的，通过改变文本的坐标然后进行实时刷新来显示字幕，屏幕刷新通过定时器来实现。。本例的部分核心源代码如下。

CGundongDlg::CGundongDlg(CWnd\* pParent /\*=NULL\*/)

: CDialog(CGundongDlg::IDD, pParent)

{

//{{AFX\_DATA\_INIT(CGundongDlg)

// NOTE: the ClassWizard will add member initialization here

//}}AFX\_DATA\_INIT

// Note that LoadIcon does not require a subsequent DestroyIcon in Win32 m\_hIcon = AfxGetApp()->LoadIcon(IDR\_MAINFRAME); WidthX=200; minWidth=10;

Heighty=15;

maxHeight=200; m\_str="欢迎使用本程序!";

TIMER=0;

m\_brush.CreateSolidBrush(RGB(180,240,210)); //改变对话框背景

m\_brush1.CreateSolidBrush(RGB(255,0,0));

}

void CGundongDlg::DoDataExchange(CDataExchange\* pDX)

{

CDialog::DoDataExchange(pDX);

//{{AFX\_DATA\_MAP(CGundongDlg)

// NOTE: the ClassWizard will add DDX and DDV calls here

//}}AFX\_DATA\_MAP

}

BEGIN\_MESSAGE\_MAP(CGundongDlg, CDialog)

//{{AFX\_MSG\_MAP(CGundongDlg)

ON\_WM\_SYSCOMMAND()

ON\_WM\_PAINT()

ON\_WM\_QUERYDRAGICON()

ON\_WM\_TIMER()

ON\_WM\_CTLCOLOR()

ON\_WM\_DESTROY()

//}}AFX\_MSG\_MAP

END\_MESSAGE\_MAP()

/////////////////////////////////////////////////////////////////////////////

// CGundongDlg message handlers

BOOL CGundongDlg::OnInitDialog()

{

CDialog::OnInitDialog();

// Add "About..." menu item to system menu.

// IDM\_ABOUTBOX must be in the system command range.

ASSERT((IDM\_ABOUTBOX & 0xFFF0) == IDM\_ABOUTBOX);

ASSERT(IDM\_ABOUTBOX < 0xF000); CMenu\* pSysMenu = GetSystemMenu(FALSE); if (pSysMenu != NULL)

{

CString strAboutMenu;

strAboutMenu.LoadString(IDS\_ABOUTBOX); if (!strAboutMenu.IsEmpty())

{

pSysMenu->AppendMenu(MF\_SEPARATOR);

|  |
| --- |
| pSysMenu->AppendMenu(MF\_STRING, IDM\_ABOUTBOX, strAboutMenu);  }  }  // Set the icon for this dialog. The framework does this automatically  // when the application’s main window is not a dialog  SetIcon(m\_hIcon, TRUE); // Set big icon  SetIcon(m\_hIcon, FALSE); // Set small icon    // TODO: Add extra initialization here  TIMER=SetTimer(ID\_TIMER1,150,NULL);  return TRUE; // return TRUE unless you set the focus to a control }    void CGundongDlg::OnSysCommand(UINT nID, LPARAM lParam)  {  if ((nID & 0xFFF0) == IDM\_ABOUTBOX)  {  CAboutDlg dlgAbout; dlgAbout.DoModal();  }  else  {  CDialog::OnSysCommand(nID, lParam);  }  }  // If you add a minimize button to your dialog, you will need the code below // to draw the icon. For MFC applications using the document/view model, // this is automatically done for you by the framework.  void CGundongDlg::OnPaint()  {  if (IsIconic())  {  CPaintDC dc(this); // device context for painting  SendMessage(WM\_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0); |

// Center icon in client rectangle int cxIcon = GetSystemMetrics(SM\_CXICON); int cyIcon = GetSystemMetrics(SM\_CYICON);

CRect rect;

GetClientRect(&rect);

int x = (rect.Width() - cxIcon + 1) / 2;

|  |
| --- |
| int y = (rect.Height() - cyIcon + 1) / 2;    // Draw the icon dc.DrawIcon(x, y, m\_hIcon);  }  else  {  CPaintDC dc(this); // device context for painting dc.SetTextColor(RGB (180,240,210));  dc.SetBkMode(TRANSPARENT); //设置背景为透明！ CRect rect; rect.left=150; rect.right=350; rect.top=10; rect.bottom=40; dc.Rectangle(&rect); dc.FillRect(&rect,&m\_brush1); rect.left=10; rect.right=135; rect.top=40; rect.bottom=200; dc.Rectangle(&rect); dc.FillRect(&rect,&m\_brush1); dc.TextOut(WidthX,15,m\_str);  dc.TextOut(15,Heighty,m\_str);  // CDialog::OnPaint();  }  }  // The system calls this to obtain the cursor to display while the user drags // the minimized window.  HCURSOR CGundongDlg::OnQueryDragIcon() |

{

return (HCURSOR) m\_hIcon;

}

void CGundongDlg::OnTimer(UINT nIDEvent)

{

// TODO: Add your message handler code here and/or call default

DrawV();

|  |
| --- |
| DrawH();  CDialog::OnTimer(nIDEvent);  }  void CGundongDlg::DrawV() //竖直方向滚动函数  {  CRect rect; //每次重画区域大小 rect.left=10;  rect.right=m\_str.GetLength()+rect.left+120;  CFont \* font=GetFont(); LOGFONT logFont; font->GetLogFont(&logFont); rect.top=Heighty;  rect.bottom=rect.top+logFont.lfHeight+40;  InvalidateRect(&rect);  Heighty+=logFont.lfHeight+18; //每步向下移动距离 if(Heighty>=maxHeight) Heighty=15; //回到起始位置  UpdateWindow();  }  void CGundongDlg::DrawH() //水平向滚动函数  {  CRect rect; //每次重画区域大小 rect.top=15;  CFont \* font=GetFont();  LOGFONT logFont; font->GetLogFont(&logFont);  rect.bottom=rect.top+logFont.lfHeight+80; rect.left=WidthX-m\_str.GetLength()-10; rect.right=rect.left+m\_str.GetLength()+150; InvalidateRect(&rect);  if(WidthX<10) WidthX=350; //回到起始位置  WidthX=WidthX-6; //每步向左移动距离  UpdateWindow();  }  HBRUSH CGundongDlg::OnCtlColor(CDC\* pDC, CWnd\* pWnd, UINT nCtlColor)  {  HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor); |

// TODO: Change any attributes of the DC here

return m\_brush; //改变对话框背景颜色

// return hbr;

}

void CGundongDlg::OnDestroy()

{

CDialog::OnDestroy();

// TODO: Add your message handler code here if(TIMER!=0)

KillTimer(TIMER);

}

运行结果如图 6-2 所示。



图 6-2 “动态字幕”示例运行结果

技巧：制作 3D 空心字有时在编程实践中需要输出一些特效文字，比如 3D 空心文字。要在 Visual C++程序中输出 3D 空心效果的文字，可以先输入 3D 文字然后建立一个路径对象，向路径对象输出该文字，之后用当前画刷填充区域内部并用当前画笔勾画路径的轮廓就可达到此目的。下面通过一个实际的示例来说明具体的实现方法。

实例 6-2：字体和文本输出实例。源代码在光盘中“\06\实例 6-2\HollowFont”目录下。

下面的例子是基于 SDI 结构的，其中核心源代码如下：

CHollowFontView::CHollowFontView()

{

LOGFONT lf;

lf.lfHeight = -180; //字体字符的高度

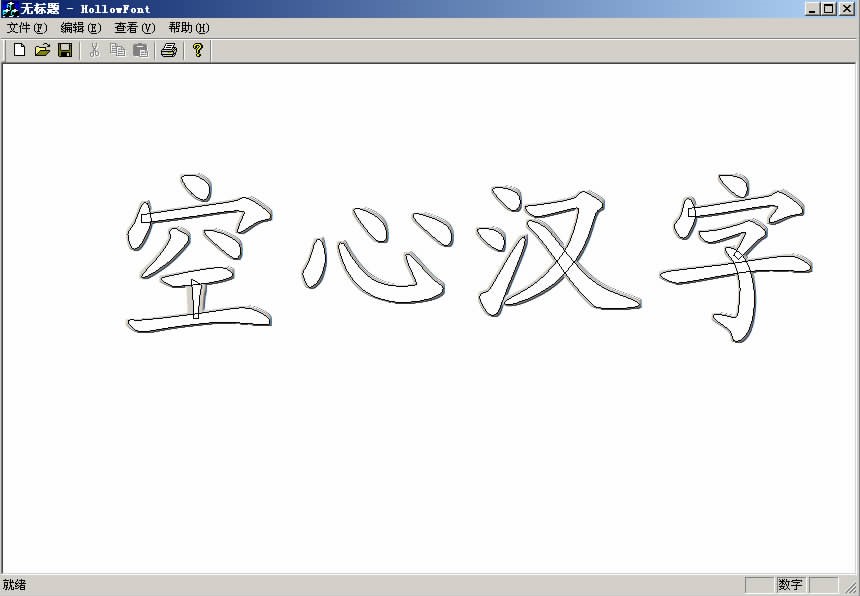
lf.lfWeight = 0; //字符平均宽度 lf.lfEscapement =0; lf.lfOrientation=0; //字符角度 lf.lfWeight=FW\_HEAVY; //字符的粗细度

lf.lfItalic = FALSE; //倾斜

|  |
| --- |
| lf.lfUnderline = FALSE; //下划线 lf.lfStrikeOut = FALSE; //删除线  lf.lfCharSet = GB2312\_CHARSET; //中文字符集  lf.lfOutPrecision = OUT\_STROKE\_PRECIS; //字体输出结果和要求的匹配程度 lf.lfClipPrecision = CLIP\_STROKE\_PRECIS; //如何裁剪落于裁剪区之外的字符 lf.lfQuality = DRAFT\_QUALITY; //字体属性匹配的精确程度 lf.lfPitchAndFamily = VARIABLE\_PITCH|FF\_MODERN;//字体间距和字体族  strcpy(lf.lfFaceName,"楷体\_GB2312"); //楷体 m\_font.CreateFontIndirect(&lf); m\_sText=\_T("空心汉字");    }    void CHollowFontView::OnDraw(CDC\* pDC)  {  CHollowFontDoc\* pDoc = GetDocument();  ASSERT\_VALID(pDoc);    // TODO: add draw code for native data here  if(m\_font.GetSafeHandle()!=NULL) //如果成功建立字体  {  CFont \*pOldFont = (CFont \*)pDC->SelectObject(&m\_font);  pDC->SetBkMode(TRANSPARENT); int cx = 100, cy = 100; //文本输出位置 CString sTemp = m\_sText;  cx += 3; cy += 3;  pDC->SetTextColor(GetSysColor(COLOR\_3DDKSHADOW)); //以下先输出突出的 3D 文本  pDC->TextOut(cx+2,cy-2,sTemp); pDC->TextOut(cx+2,cy+2,sTemp);  pDC->SetTextColor(GetSysColor(COLOR\_3DHILIGHT));    pDC->TextOut(cx+1,cy-2,sTemp); pDC->TextOut(cx-2,cy+1,sTemp); pDC->TextOut(cx-2,cy-2,sTemp);  pDC->SetTextColor(GetSysColor(COLOR\_3DSHADOW));    pDC->TextOut(cx-1,cy+1,sTemp); pDC->TextOut(cx+1,cy-1,sTemp); pDC->TextOut(cx+1,cy+1,sTemp); |

|  |  |  |
| --- | --- | --- |
|  | | pDC->SetTextColor(GetSysColor(COLOR\_3DLIGHT));  pDC->TextOut(cx,cy-1,sTemp); pDC->TextOut(cx-1,cy,sTemp); pDC->TextOut(cx-1,cy-1,sTemp);  //然后通过路径对象将以上要输出的文本变成空心 |
|  |  | //向路径输出“汉字”  pDC->BeginPath();  pDC->TextOut(cx,cy,sTemp,8); pDC->EndPath();  //填充路径  pDC->SetPolyFillMode(WINDING); //设置填充模式 pDC->StrokeAndFillPath();//绘制路径外观 pDC->SelectObject(pOldFont); |
| } | } |  |

函数 pDC->StrokeAndFillPath 用于先封闭所有的开放路径，然后用当前画刷和填充方式填充区域内部。由于当前画刷是一个实心的白色画刷（WHITE\_BRUSH)，因此，文字的显示效果就变成了空心，同时该函数用当前画笔勾画路径的轮廓。运行结果如图 6-3 所示。

 图 6-3 “3D 空心字”例程运行结果

## 6.5 本章小结

本章介绍了 Windows GDI 输出文本和字体的一般概念。在对 MFC 字体类 CFont 的主要成员属性及方法作了较为详尽的讲解的基础上，介绍了 GDI 输出文本的方法。同时，通过两个例程介绍了产生“动态”字幕和“空心字”的技巧和方法。通过本章的学习，读者应能够学到利用类 CFont 创建自定义字体的方法以及进行多种样式文本输出的方法。

# 第 7 章 图形图像

随着计算机技术的发展，数字图像处理，技术得到了广泛应用，如模式识别、人脸识别、图像融合/编辑等都涉及到了计算机的图形图像处理技术。本章将为读者介绍 Visual C++中图像处理技术的相关知识，主要包括以下几个方面的内容：

* 屏幕秽土的主要函数，及其调用方法；
* 图形设备接口及其主要 MFC 类；
* 画刷和画笔的使用。

## 7.1 绘图的基础知识

在当今的操作系统中，应用最广泛的是拥有友好、美观图形界面的操作系统。因此绘图成为了一项核心技术。

要了解绘图的原理，首先需要知道显示设备。当前的显示设备主要包括基于阴极射线管（CRT）的显示器以及液晶显示器。阴极射线管主要由电子枪，聚焦系统、加速电极，偏转系统，荧光屏构成。电子枪发出高速的电子束，经过聚焦系统、加速系统和偏转系统到达荧光屏的特定位置。电子束轰击荧光屏发出荧光，形成亮点。要保持显示一幅稳定的画面，必须不断地发射电子束。

计算机的显示功能是由图形处理器(显卡)来完成的。图形处理器是图形系统结构的重要元件，是连接计算机和显示终端的纽带。早期的图形处理器只包含简单的存储器和帧缓冲区，它们实际上只起了一个图形的存储和传递作用，一切操作都必须有 CPU 来控制。现在的图形处理器不单存储图形，而且能完成大部分图形函数，专业的图形卡已经具有很强的 3D 处理能力，大大减轻了 CPU 的负担，提高了显示质量和显示速度。

当前流行的操作系统绘图功能主要基于光栅图形学。包括基本图形（直线、圆弧、椭圆弧）的扫描转换、多边形的扫描转换、区域填充、裁剪、反走样、投影、消隐等。

直线是操作系统频繁绘制的基本图形元素。在不同分辨率的显示器上，为了保证绘制出的直线有良好的视觉效果，需要确定最佳逼近于该直线的一组像素，最常用的方法包括数值微分法(DDA)，中点画线法和 Bresenham 算法。下面简单介绍数值微分法。

已知过端点 *P*0(x ,y0 0) ， *P*1(x ,y1 1) 的直线段 L ： *y kx b*= + ， 直线斜率为

k = (y1 − y )/(x0 1 − x )0 ，在 x 方向从x0开始，向x1步进，步长取 1 个像素，计算相应的 y

坐标 *y kx b*= + 。取像素点((*x round x*, ( )) 作为当前点的坐标。其他方法读者如果有兴趣可以参阅相关书籍。图形变换是计算机图形学中的重要内容。通过各种变换，比如旋转、镜像、平移、缩放等，可以由一个简单的图形生成复杂的图形。

## 7.2 屏幕绘图的主要函数

在 Visual C++环境下，可以绘制点，线，矩形，多边形，椭圆，位图以及文本等，具体

调用格式如下。

1．CDC::SetPixel

用来设定指定点的颜色，返回指定点的颜色值。它有两种调用形式，函数原型如下：

COLORREF SetPixel( int *x*, int *y*, COLORREF *crColor* );

COLORREF SetPixel( POINT *point*, COLORREF *crColor* );

* *x* 指定点的 *x* 坐标。
* *y* 指定点的 *y* 坐标。
* crColor 将要绘制颜色。
* point 指定点的坐标

2．CDC::MoveTo

用来移动当前点到指定点，返回当前点的坐标。它有两种调用形式，函数原型如下：

CPoint MoveTo( int *x*,int *y* );

CPoint MoveTo( POINT *point* );

参数

* *x* 新位置的 *x* 坐标。
* *y* 新位置的 *y* 坐标。
* point 新位置的坐标

3．CDC::LineTo

用来绘制一条从当前点到指定点的直线，返回是否绘制成功的标志。它有两种调用形式，

函数原型如下：

CPoint LineTo( int *x*, int *y* );

CPoint LineTo( POINT *point* );

参数

* *x* 指定点的 *x* 坐标。
* *y* 指定点的 *y* 坐标。
* point 指定点的坐标

4．CDC::Rectangle

用来绘制一个矩形，返回是否绘制成功的标志。它有两种调用形式，函数原型如下：

BOOL Rectangle( int *x1*, int *y1*, int *x2*, int *y2* );

BOOL Rectangle( LPCRECT *lpRect* );

参数

* *x*1 矩形的左上角 *x* 坐标。
* *y*1 矩形的左上角 *y* 坐标。
* *x*2 矩形的右下角 *x* 坐标。
* *y*2 矩形的右下角 *y* 坐标。
* lpRect 矩形对象的指针

5．CDC::PolyLine

根据一组多边形顶点绘制多边形，返回是否绘制成功的标志。函数原型如下：

BOOL Polyline( LPPOINT *lpPoints*, int *nCount* );

参数

* lpPoints 多边形顶点的点结构数组指针。
* nCount 数组中点的个数

6．CDC::PolyBezier

用来绘制一条 Bezier 曲线，返回是否绘制成功的标志。函数原型如下：

BOOL PolyBezier ( LPPOINT *lpPoints*, int *nCount* );

参数

* lpPoints 包含控制点的点结构数组指针。
* nCount 数组中点的个数

7．CDC::Ellipse

用来绘制椭圆，返回是否绘制成功的标志。它有两种调用形式，函数原型如下：

BOOL Ellipse( int *x1*, int *y1*, int *x2*, int *y2* );

BOOL Ellipse( LPCRECT *lpRect* );

参数

* x1 椭圆包围矩形的左上角 x 坐标。
* y1 椭圆包围矩形的左上角 y 坐标。
* x2 椭圆包围矩形的右下角 x 坐标。
* y2 椭圆包围矩形的右下角 y 坐标。
* lpRect 椭圆包围矩形的指针

8．CDC::BitBlt

用于从源设备复制一幅 BMP 图像到目标设备，返回是否绘制成功的标志。函数原型如下：

BOOL BitBlt( int *x*, int *y*, int *nWidth*, int *nHeight*, CDC\* *pSrcDC*, int *xSrc*, int *ySrc*, DWORD *dwRop* );

参数

* *x*  目标矩形区域的左上角 x 坐标。
* *y* 目标矩形区域的左上角 y 坐标。
* nWidth 目标矩形区域和源图像的宽度。
* nHeight 目标矩形区域和源图像的高度。
* pSrcDC 源设备上下的指针。
* xSrc 源矩形区域的左上角 x 坐标。
* ySrc 源矩形区域的左上角 y 坐标。
* dwRop 光栅操作的类型，包括如下类型： BLACKNESS 将输出变为黑色。

DSTINVERT 将目标图像反色。

MERGECOPY 将模式和源图像做与运算。

MERGEPAINT 将源图像取反后与目标图像做或运算。

NOTSRCCOPY 复制取反后的源图像到目标图像。 NOTSRCERASE 将目标图像与源图像做或运算后取反。

PATCOPY 复制模式到目标图像。

PATINVERT 将目标图像与模式做异或运算。

PATPAINT 将源图像取反后与模式做或运算，将这个运算的结果与目标图像做或运算。

|  |  |
| --- | --- |
| SRCAND | 将目标图像与源图像做与运算。 |
| SRCCOPY | 复制源图像到目标图像。 |
| SRCERASE | 将目标图像取反后与源图像做与运算。 |
| SRCINVERT | 将目标图像与源图像做异或运算。 |
| SRCPAINT | 将目标图像与源图像做或运算。 |
| WHITENESS | 将输出变为白色 |

9．CDC::TextOut

用来绘制输出文本，返回是否绘制成功的标志。它有两种调用形式，函数原型如下：

virtual BOOL TextOut( int *x*, int *y*, LPCTSTR *lpszString*, int *nCount* );

BOOL TextOut( int *x*, int *y*, const CString& *str* );

* 参数
* *x* 文本起始点的 *x* 坐标。
* *y* 文本起始点的 *y* 坐标。
* lpszString 将要绘制的文本。
* nCount 字符串中的字节数。
* str 包含将要绘制的文本的 CString 对象

## 7.3 图形设备接口（GDI）

图形设备接口 GDI（Graphics Device Interface）提供了在 Windows 下绘图的基本功能。

### 7.3.1 设备环境类（CDC）

CDC 直接继承自 CObject 类，类的继承顺序如图 7-1 所示。

 图 7-1 CDC 类的基类继承图

CDC 类是设备上下文对象的基类，它提供了处理显示器、打印机等设备的成员函数，利用 CDC 对象可以访问整个显示设备（如显示器）和非显示设备（如打印机）。

通过 CDC 类的成员函数可以进行所有的绘图操作，包括绘图工具和 GDI 对象的选择、颜色和调色板的处理、获得和设置绘图工具的属性、图形的绘制等。另外它可以绘制文本、处理字体，使用打印机和显示源文件提供的相应的成员函数。

CDC 类包含 m\_hDC 和 m\_hAttribDC 两个设备上下文句柄，CDC 对象创建时二者指向同一个设备。m\_hDC 用于所有的输出 GDI 调用（如 SetTextColor()函数），m\_hAttribDC 用于大部分的属性 GDI 调用（如 GetTextColor()函数）。应用程序框架可以同时使用两个设备上下文，并且在物理设备获得属性的同时，将结果输出到一个 CMateFileDC 对象中，打印预览就是以这种方式实现的。

CDC 类的定义包含在头文件“afxwin.h”中。

CDC 类封装了几乎所有的 Windows GDI 函数，构造了 CDC 对象后，就可以调用它的成员函数完成相应的图形操作。同时 MFC 还提供了 CDC 类的派生类 CpaintDC、CclientDC、

CmetaFileDC、CwindowDC，用于完成特定操作。

通常使用 CDC 类的方法是传递一个 CDC 的指针 pDC 到 OnDraw 函数，在 OnDraw 函数中调用它的成员函数完成绘图操作。

使用完毕后应该及时删除构造的 CDC 对象。Windows 限制了可使用的设备上下文的数量，如果删除一个设备上下文对象失败，将会占用一小部分内存，直到程序退出。要确保设备上下文对象被删除，最容易的方法是在堆栈上构造对象。

也可通过 CWnd::GetDC()成员函数来获得设备上下文指针，这种情况下必须调用

ReleaseDC 函数来释放设备环境。

### 7.3.2 GDI 对象

Windows 的 GDI 对象都是从基类 CGdiObject 派生而来。可以通过调用 GDI 派生类的构造函数来创建 GDI 对象。GDI 对象如下：

* CBitmap；
* CBrush；
* CPen；
* CFont；
* CRgn；
* CPalette。

## 7.4 画笔和画刷

画笔（CPen）和画刷（CBrush）是最常用的两个 GDI 对象。本节通过两个实例，详细介绍这两个对象的使用方法。

### 7.4.1 画笔

实例 7-1：画笔使用实例实例。源代码在光盘中“\07\实例 7-1\CpuUsage”目录下。

1．创建程序

下面通过动态显示 CPU 的使用记录的例子介绍画笔的使用方法。利用 Visual C++的 AppWizard 创建一个基于对话框的应用程序 CpuUsage，为对话框添加一个静态文本框控件和一个 group box 控件。为对话框类 CCpuUsageDlg 添加成员变量，如表 7-1 所示。

表 7-1 CCpuUsageDlg 的成员变量表

|  |  |  |  |
| --- | --- | --- | --- |
| 成员变量 | 类型 |  | 作用 |
| m\_btnView | CButton | 显示区域 |  |
| m\_strCpuUsage | CString | 显示字符串 |  |
| m\_UsageHistory | int[] | CPU 使用记录 |  |
| m\_timer | int | 计时器 |  |
| m\_Usage | CCpuUsage | 测试 Cpu 使用率类 |  |

2．编写代码

（1）初始化对话框

修改 CCpuUsageDlg 的成员函数 OnInitDialog，对变量做初始化。代码如下：

|  |
| --- |
| BOOL CCpuUsageDlg::OnInitDialog()  {  CDialog::OnInitDialog();  // Add "About..." menu item to system menu.    // IDM\_ABOUTBOX must be in the system command range.  ASSERT((IDM\_ABOUTBOX & 0xFFF0) == IDM\_ABOUTBOX);  ASSERT(IDM\_ABOUTBOX < 0xF000);    CMenu\* pSysMenu = GetSystemMenu(FALSE); if (pSysMenu != NULL)  {  CString strAboutMenu; |

|  |  |
| --- | --- |
| } | strAboutMenu.LoadString(IDS\_ABOUTBOX); if (!strAboutMenu.IsEmpty())  {  pSysMenu->AppendMenu(MF\_SEPARATOR);  pSysMenu->AppendMenu(MF\_STRING, IDM\_ABOUTBOX, strAboutMenu);  }  }  // Set the icon for this dialog. The framework does this automatically  // when the application’s main window is not a dialog    // 设置窗口总在最前  ::SetWindowPos(this->GetSafeHwnd(), CWnd::wndTopMost, 0, 0, 0, 0, SWP\_NOSIZE);    SetIcon(m\_hIcon, TRUE); // Set big icon  SetIcon(m\_hIcon, FALSE); // Set small icon    // 初始化使用记录数组  memset(m\_UsageHistory, 0, MAXTIMES\*sizeof(int));  // 设置定时器  m\_timer = SetTimer(1, 500, 0);    return TRUE; // return TRUE unless you set the focus to a control |

利用 API 函数 SetTimer 启动定时器，函数原型如下：

UINT SetTimer( UINT *nIDEvent*, UINT *nElapse*, void (CALLBACK EXPORT\* *lpfnTimer*)(HWND, UINT,

UINT, DWORD) );

* nIDEvent：非 0 定时器标志；
* nElapse：时间间隔； • lpfnTimer：处理函数。

当起动定时器后，每间隔 nElapse 时间间隔，系统就会给应用程序发送 WM\_TIMER 消息，通常应用程序在 OnTimer 中响应这个消息。当程序退出时，需要调用 KillTimer 删除定时器，释放系统资源，它的函数原型如下：

BOOL KillTimer( int nIDEvent );

* nIDEvent：调用 SetTimer 的返回值。

（2）响应 OnTimer 消息

在初始化过程中，设置的时间间隔为 500 毫秒。即每隔 500 毫秒，测试一次 CPU 的使用率。响应 OnTimer 消息，代码如下：

void CCpuUsageDlg::OnTimer(UINT nIDEvent)

{

// TODO: Add your message handler code here and/or call default int per;

// 得到当前 CPU 的使用率

per = m\_Usage.GetCpuUsage();

// 计数

if(times < MAXTIMES) times ++;

// 更新 CPU 使用记录数组

for(int i=0; i<MAXTIMES-1; i++)

{

m\_UsageHistory[i] = m\_UsageHistory[i+1];

}

// 将最后一个记录设为当前使用率

m\_UsageHistory[i] = per;

// 格式化字符串

m\_strCpuUsage.Format("CPU 使用 %3d%%", per);

// 更新显示

UpdateData(FALSE);

Invalidate();

CDialog::OnTimer(nIDEvent);

}

1. 实现绘制示意图为 CCpuUsageDlg 添加成员函数 DrawDiagram，绘制 CPU 使用率示意图，代码如下：

|  |
| --- |
| void CCpuUsageDlg::DrawDiagram(CPaintDC \*pDC)  {  CRect rect; int W, H, x, y; float xinterval, yinterval;  CPen curpen, \*oldpen;  CBrush newbrush, \*oldbrush; |

|  |  |
| --- | --- |
| } | // 得到绘制区域的矩形  m\_btnView.GetWindowRect(rect); ScreenToClient(rect); // rect.top += 15;  // 得到矩形的长和宽  W = rect.Width(); H = rect.Height();  xinterval = (float)W/(MAXTIMES-1); yinterval = (float)H/100;  // 创建黑色画刷, 绘制背景  newbrush.CreateSolidBrush(RGB(0,0,0)); oldbrush = pDC->SelectObject(&newbrush); pDC->Rectangle(rect); newbrush.DeleteObject(); pDC->SelectObject(oldbrush);  // 绿色画笔, 绘制曲线  curpen.CreatePen(PS\_SOLID, 2, RGB(0, 255, 0)); oldpen = pDC->SelectObject(&curpen);    pDC->MoveTo(rect.left, rect.bottom);  // 用线段近似曲线  for(int i=0; i<MAXTIMES; i++)  {   1. = rect.left + (int)(i\*xinterval); 2. = rect.bottom - (int)(m\_UsageHistory[i]\*yinterval); pDC->LineTo(x, y);   }  // 恢复设备上下文  pDC->SelectObject(oldpen); curpen.DeleteObject(); |

1. 动态绘制 CPU 使用记录响应 CCpuUsageDlg 的 WM\_PAINT 消息，用来动态绘制 CPU 使用记录，代码如下：

void CCpuUsageDlg::OnPaint()

|  |  |
| --- | --- |
| {      } | if (IsIconic())  {  ……  } else  {  CPaintDC dc(this);  // 绘制 CPU 使用记录  DrawDiagram(&dc);  } |

1. 释放资源最后，响应 CCpuUsageDlg 的 WM\_DESTROY 消息，在程序退出时，删除定时器，释放

系统资源，代码如下：

void CCpuUsageDlg::OnDestroy()

{

CDialog::OnDestroy();

// 释放定时器资源

KillTimer(m\_timer);

}

1. 运行结果程 序 运 行 后 ， 界 面 如 图 7-2 至 7-5 所 示 。

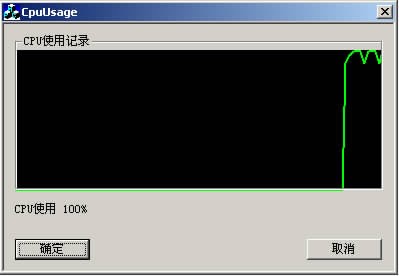


图 7-2 Cpu 使用记录 1

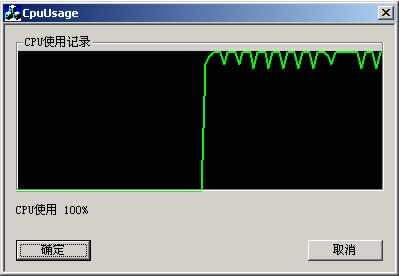


图 7-3 Cpu 使用记录 2

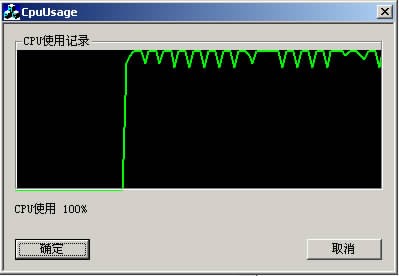
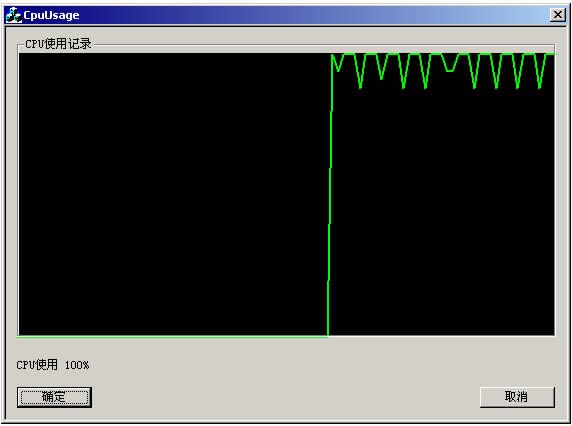


图 7-4 Cpu 使用记录 3

 图 7-5 CPU 使用记录

7.4.2 画刷

实例 7-2：画刷使用实例实例。源代码在光盘中“\07\实例 7-2\Color”目录下。

1．创建程序

本节利用 CDC 类的一个函数和画刷来取得指定点的颜色。首先用 Visual C++的 AppWizard 创建一个基于对话框的应用程序 Color，为对话框添加两个图像框控件。为

CColorDlg 添加如表 7-2 所示的成员变量，。

表 7-2 CColorDlg 的成员变量表

|  |  |  |  |
| --- | --- | --- | --- |
| 成员变量 | 类型 |  | 作用 |
| m\_StaticClr | CStatic | 显示颜色区域 |  |
| m\_StaticPic | CStatic | 显示图像区域 |  |

为应用程序添加一个 bmp 资源 IDB\_BITMAP1，设置图片控件的属性，如图 7-6 所示。



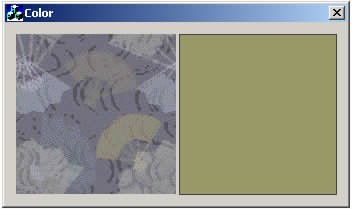
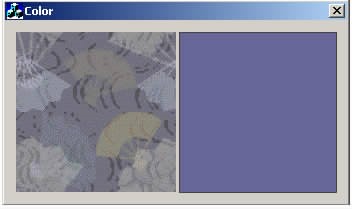
图 7-6 设置图片框属性

2．添加响应代码添加响应鼠标单击的代码，用来得到鼠标落点的颜色并且显示出来。代码如下：

|  |
| --- |
| // 响应鼠标左键落在图片区域的操作  void CColorDlg::OnLButtonDown(UINT nFlags, CPoint point)  {  // TODO: Add your message handler code here and/or call default  CRect picRect, clrRect;  CBrush newBrush, \*oldBrush;  COLORREF rgb;  CClientDC dc(this);    // 得到图片所在的区域矩形  m\_StaticPic.GetWindowRect(picRect);  ScreenToClient(picRect);    // 得到颜色所在的区域矩形  m\_StaticClr.GetWindowRect(clrRect);  ScreenToClient(clrRect);    // 如果鼠标落在图像区域  if(picRect.PtInRect(point))  {  // 得到该点的像素  rgb = dc.GetPixel(point);    // 生成新的画刷  newBrush.CreateSolidBrush(rgb);    // 设备选择新的画刷  oldBrush = dc.SelectObject(&newBrush);    // 用像素颜色的填充颜色矩形  dc.Rectangle(clrRect); dc.SelectObject(oldBrush); newBrush.DeleteObject();    // 刷新屏幕  Invalidate(FALSE);  }    CDialog::OnLButtonDown(nFlags, point);  } |

3．运行结果

程序运行后，用鼠标单击图片不同的区域，程序界面如图 7-7 所示。

（a） （b）图 7-8 4 鼠标点在不同区域的颜色

## 7.5 本章小结

本章讲述了使用 Visual C++进行图形绘制和图像处理的基本方法。介绍了 Windows 基本的绘图函数，主要的 GDI 对象，并且给出相应的实例。给一个图像浏览器应用程序，介绍了图像处理的基本知识和算法。通过对本章的阅读，读者应该了解图形绘制和图像处理的基本原理和方法。此外，本章还给出了一些相关的小技巧。

# 第 8 章 文件操作

文件操作是操作系统提供的基本功能之一。本章将通过示例介绍在 MFC 中对文件进行操作的方法，主要包括串行化、CFile 文件操作和文件对话框 3 部分内容。这 3 部分基本涵盖了 MFC 中对文件的主要操作。同时还详细地介绍各种文件操作函数的使用方法，为读者编写功能复杂的文件处理程序提供参考。

## 8.1 串行化基础知识

在 Visual C++中使用 MFC 开发的基于文档的应用程序。在这个框架中，数据的维护及其显示分别由两个不同，但又彼此紧密相关的对象—文档和视图实现的。MFC 中各种自动化代码的存在，文档视图结构在很多场合与传统的编程方式相比更有利于应用程序的编写。串行化就是这样一个自动化过程，它使得对文档的磁盘读写操作变得简单易行。本节将详细介绍串行化的基础知识，包括文档的基本结构、用 Serialize()函数实现串行化、与串行化相关的类，以及在文档视图中添加串行化代码 4 个部分。

### 8.1.1 文档类

在文档视图结构中，文档的任务通常是对数据进行管理和维护。数据一般都被保存在文档类的成员变量中。而在 MFC 中，文档类通过一个称为串行化（Serialize）的过程将数据保存到磁盘文件中或者将数据从磁盘文件中读取。文档类为数据的串行化提供了默认的支持，并将其作为可重载函数，只需重载这些成员函数就可以为程序提供自定义的串行化支持。

文档类在文档视图结构中主要完成以下几个任务。

* 在内存中保存应用程序特定的数据，并对其进行管理，同时提供给视图类进行显示。
* 提供用于操作文档数据的接口。
* 参与写文件和读文件。通过串行化功能，文档类可以将数据从文件中读出或者写入。
* 参与打印输出。这个任务主要通过视图类访问文档进行。
* 处理跟文档有关的命令消息。这些命令包括菜单项、工具栏按钮或快捷键生成的命令。默认情况下，文档类使用序列化处理“ 文件” 菜单的“ 保存” 和“ 另存为” 命令。还可以为其他跟文档相关的自定义命令添加相应的处理函数。

为了完成上述的任务，文档类的基类 CDocument 提供了下列一些成员函数。

1．处理文档数据与文件相关的操作的函数

1. GetTitle()和 SetTitle()

这两个函数作为一组，前者用于获取文档的标题，后者用于设置文档标题。原型为：

const CString& GetTitle( ) const; virtual void SetTitle(

LPCTSTR lpszTitle

);

1. GetPathName()和 SetPathName()

这两个函数用于处理文档保存的路径。前者用于获得文档保存路径的字符串，后者设置

该路径。原型为：

const CString& GetPathName( ) const; virtual void SetPathName(

LPCTSTR lpszPathName,

BOOL bAddToMRU = TRUE

);

参数 bAddToMRU 用于确定是否将路径添加到最近打开的文档列表中。

2．GetDocTemplate()

该函数用于获得文档模板对象的指针。原型为：

CDocTemplate\* GetDocTemplate( ) const;

3．处理文档类和视图类的关系的函数

1. AddView()

该函数用于向与文档相关联的视图列表中添加指定的视图。原型为：

void AddView(

CView\* pView

);

参数 pView 指向需要添加的视图对象。

1. RemoveView()

该函数用于从文档类的视图列表中删除指定视图。原型为：

void RemoveView(

CView\* pView

);

参数 pView 指向需要删除的视图对象。

1. UpdateAllView()

该函数用于通知所有视图进行重绘。一般情况下，当文档数据被修改后，应当调用这个

函数使得所有跟文档关联的视图进行重绘，以显示最新的内容。原型为：

void UpdateAllViews( CView\* pSender,

LPARAM lHint = 0L,

CObject\* pHint = NULL

);

* 参数 pSender：用于指定修改文档的视图类的指针。
* 参数 lHint 和 pHint：存有修改的一些参数。 4．虚函数

下面介绍的函数是 CDocument 类提供的虚函数，程序的 CDocument 派生类通过重载这

些函数提供程序自定义的功能。

（1）OnNewDocument()

该虚函数用于在建立文档时被 MFC 框架调用。原型为：

virtual BOOL OnNewDocument( );

• 函数返回值：函数是否成功的标志。

（2）OnOpenDocument ()

该虚函数用于在打开文档时被 MFC 框架调用。原型为：

virtual BOOL OnOpenDocument(

LPCTSTR lpszPathName

);

* 参数 lpszPathName：用于获得将要打开文档的路径。
* 函数返回值：函数是否成功的标志。

（3）OnSaveDocument ()

该虚函数用于在保存文档时被 MFC 框架调用。原型为：

virtual BOOL OnSaveDocument(

LPCTSTR lpszPathName

);

* 参数 lpszPathName：用于获得将要保存文档的路径。
* 函数返回值：函数是否成功的标志。

1. OnCloseDocument()

该虚函数用于在关闭文档时被 MFC 框架调用。原型为：

virtual void OnCloseDocument( );

1. CanCloseFrame()

该虚函数用于确认文档的框架窗口是否允许被关闭，比如文档未保存时，在框架窗口要

被关闭时提示是否保存文档。原型为：

virtual BOOL CanCloseFrame(

CFrameWnd\* pFrame

);

* 参数 pFrame：该文档的框架窗口类的指针。
* 函数返回值：函数是否成功的标志。

1. DeleteContents()

该虚函数用于在未销毁文档对象时删除文档数据。原型为：

virtual void DeleteContents( );

1. ReleaseFile()

该虚函数用于释放文件以允许其他应用程序使用。原型为：

virtual void ReleaseFile(

CFile\* pFile,

BOOL bAbort

);

* 参数 pFile：要释放的 CFile 对象。
* 参数 bAbort：指定用什么方法释放对象，若取值为 TRUE，则用 CFile::Abort()，反之用 CFile::Close()。

（8）SaveModified()

该虚函数用于查询文档的修改状态并存储修改的文档。原型为：

virtual BOOL SaveModified( );

• 函数返回值：函数是否成功的标志。

### 8.1.2 Serialize()函数

在 8.1.1 节中介绍了文档的主要结构，本小节将讲述如何创建和使用文档与磁盘文件之间的串行化读写通道，即 Serialize()函数。

1．串行化的基本概念

串行化在面向对象程序设计领域中的基本概念是指对象可以被持续，即当程序退出时，它们可以被保存在磁盘中，而当程序重新运行时又可以从磁盘中读取恢复。对象的这种保存和恢复的过程就称为“ 串行化”。在 MFC 中，对象串行化的成员函数称为 Serialize()成员函数。这个函数提供了将类的数据进行存盘和读取的功能。

在 MFC 中，串行化过程是顺序的，即与文档相关的所有对象只能在某个单独的文件中进行顺序的读或写，而并不能进行随机的访问。MFC 的文档类中就只提供了顺序保存和读写文档数据的功能。

2．将类串行化

将类串行化需要进行下列几个步骤：

1. 从 CObject 派生

在 CObject 类中定义了基本的序列化协议和功能，需要串行化的类必须直接或间接地从

CObject 类派生，从而获得对 CObject 的序列化协议及功能的访问权限。

1. 添加 DECLARE\_SERIAL()宏

设置好基类后，在类的声明中添加一个 DECLARE\_SERIAL()宏，具体代码如下：

|  |
| --- |
| class CSample : public CObject //由 CObject 派生  { public:  ⋯ ⋯ //其他成员变量和成员函数的定义 |

|  |  |
| --- | --- |
| }; | DECLARE\_SERIAL(CSample) //声明串行化，参数为该类的类名  ⋯ ⋯ //其他成员变量和成员函数的定义 |

1. 添加 IMPLEMENT\_SERIAL()宏在类的实现文件中添加 IMPLEMENT\_SERIAL()宏，具体代码如下：

|  |  |
| --- | --- |
| IMPLEMENT\_SERIAL(CSample, CObject, 0) | //该宏一般应包含在 Sample.cpp 文件中 |

IMPLEMENT\_SERIAL()宏用于定义从 CObject 中派生可序列化类时所需的各种函数。在类的实现文件(.CPP)中使用这个宏。该宏的前两个参数是类名和直接基类的名称。

该宏的第三个参数是架构编号。架构编号实质上是类对象的版本号，它使用大于或等于零的整数。MFC 序列化代码在将对象读取到内存时检查该架构编号。如果磁盘上对象的架构编号与内存中类的架构编号不匹配，库将引发 CArchiveException 防止程序读取对象的不正确版本。

1. 定义不带参数的构造函数

对象从磁盘上加载后，MFC 通过 CreateObject()函数自动重新创建这些对象。而 CreateObject()函数创建对象时需要一个默认的构造函数。可将该构造函数声明为公共的、受保护的或私有的。在此之前，请确保它仅由串行化函数使用。

3．编写 Serialize()函数

在 CObject 类中定义的 Serialize()成员函数，实际上是对捕获对象的当前状态所必需的数据进行串行化。因此在为类做好串行化的准备后，再为类重载 CObject 的 Serialize()成员函数，就可以实现串行化的功能。例如 CSampleData 类中有如下的成员变量：

public:

CString m\_strName;

int m\_nType;

实现 CSampleData 的序列化，就需要将这两个成员变量保存到磁盘中或者从磁盘中装入，于是将 CSampleData::Serialize()函数代码修改为：

void CSampleData::Serialize(CArchive& ar)

{

CObject::Serialize(ar); //进行基类的序列化 if (ar.IsStoring()) {

ar << m\_strName << m\_nType; //保存数据

}

else {

ar >> m\_strName >> m\_nType; //读取数据

}

}

在上面的代码中，首先调用基类的 Serialize()函数进行基类的序列化，保证保存和装入的数据的正确性。

Serialize()函数具有 CArchive 参数 ar 的特性，即读写对象数据。CArchive 对象中包括成员函数 IsStoring()，该成员函数返回值为 TRUE 时表示 Serialize()正在存储（即正在写入数据），反之则表明正在加载（即正在读取数据）。用 IsStoring()的结果作为参考，使用输出运算符（<<）将对象数据插入到 CArchive 对象中或使用输入运算符（>>）提取数据。这也反映了 MFC 串行化的顺序性，即不能进行随机的读写操作，一次调用只能读取或只能保存。在串行化代码中需要对各种数据类型进行处理，主要分为下列几种情况。

（1）固有数据类型

CArchive 类的插入运算符（<<）和提取运算符（>>）对许多 C++固有的数据类型进行了重载，可以直接使用。例如在上面的程序中要对 int 的成员变量 m\_nType 进行串行化，即可直接使用插入运算符和提取运算符。下面列出一些被 CArchive 类默认支持的数据类型：

* BYTE：8 位无符号整数。
* WORD：16 位无符号整数。
* LONG：32 位带符号整数。
* DWORD：32 位无符号整数。
* float：单精度浮点数。
* double：双精度浮点数。
* int：32 位带符号整数。
* short：16 位带符号整数。
* char：8 位字符类型。
* unsigned：32 位无符号整数。

1. CString 和 CRect 等类型

CString 和 CRect 等类型，虽然不是从 CObject 派生的类，但是它们有自己针对 CArchive 类的重载插入运算符和提取运算符，因此也可以直接使用这两个运算符进行串行化。

1. 自定义的类

如果序列化的类中包含其他自定义的内嵌对象，则需要处理后再进行串性化。例如在

CSampleData 类中添加如下的新数据成员：

public:

CSampleChildData m\_data;

将 CSampleData 类串行化时，需要对 CSampleChildData 进行额外的处理。首先使得 CSampleChildData 继承 CObject，然后编写它自己的 Serialize()成员函数。这时 CSampleData 类的 Serialize()函数可以进行如下修改以实现对 m\_data 的串行化：

void CSampleData::Serialize(CArchive& ar)

{

CObject::Serialize(ar); //进行基类的序列化 if (ar.IsStoring()) {

ar << m\_strName << m\_nType; //保存数据

}

else {

ar >> m\_strName >> m\_nType; //读取数据

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| } | m\_Data.Serialize(ar); |  |  | //串行化 m\_Data |

如果CSampleData类中的CSampleChildData对象是通过指针在堆中创建的，则代码如下：

public:

CSampleChildData \*m\_pData;

将 CSampleData 类串行化更为简单。首先需要为 CSampleChildData 添加串行化代码，即继承 CObject，并添加相应的宏和构造函数，最后编写 CSampleChildData 自己的 Serialize() 函数。完成对 CSampleChildData 的修改后就可以在 CSampleData 类的 Serialize()函数中用

CArchive 的插入和提取运算符进行串行化。代码如下：

void CSampleData::Serialize(CArchive& ar)

{

CObject::Serialize(ar); //进行基类的序列化

if (ar.IsStoring()) {

ar << m\_strName << m\_nType << m\_pData; //保存数据

}

else {

ar >> m\_strName >> m\_nType >> m\_pData; //读取数据

}

}

在上述代码中，之所以能对自定义的类进行插入和提取运算，是因为在添加的

DECLARE\_SERIAL()和 IMPLEMENT\_SERIAL()宏中包含了对插入和提取运算符的重载。当 CSampleChildData 对象被写进文件时，这两个宏保证类名和数据一起被写进去；而当从文件中读入时，类名被读进来，相应的类的对象被动态构造，这些也是由这两个宏包含的代码实现的。当 CSampleChildData 对象被构造后，就可以通过 Serialize()自动进行串行化，这是由于 CArchive 的插入和提取运算符对 CObject 类型进行了重载。最后新建的 CSampleChildData 对象的指针即保存在 m\_pData 中。根据上面的分析，对于自定义的类，不能对类的实例对象使用插入和提取运算符，而只能对类的指针进行这样的操作。

（4）集合类由于所有的集合类都是从 CObject 类派生出来的，而且集合类都包含了与串行化有关的宏调用，因此就可以通过调用集合类的 Serialize()成员函数，方便地对集合进行串行化。例如一个由 CSampleData 对象组成的 CArray 集合，调用 CArray 的 Serialize()函数，就可以使得每个 CSampleData 对象的 Serialize()函数被依次调用，从而完成对集合类的序列化。

### 8.1.3 CArchive 类

在前面的章节中已有关于 CArchive 类的介绍，该类提供了一个类型安全的访问 CFile 对象（CFile 对象是 Visual C++中的基本文件对象，将在 8.2 节中详细介绍）的机制，用于将可串行化对象写入 CFile 对象或者从中读取可串行化对象。给定的 CArchive 对象可以存储数据，也可以加载数据，但不允许同时进行，而且其寿命也只限于将对象写入文件或从文件读取对象的一次性传递。这也就是 MFC 中串行化只能顺序进行的原因。下面介绍 Carchive 类的一些主要功能。

1．构造函数

Carchive 类通过构造函数从已打开的 CFile 对象中创建可以向该 CFile 对象进行串行化的新对象。其原型为：

CArchive(

CFile\* pFile, UINT nMode, int nBufSize = 4096, void\* lpBuf = NULL

);

* 参数 pFile 指向需要进行串行化的 CFile 对象。
* 参数 nMode 设置创建对象的标志，如果设置了这个标志，则必须在销毁前调用 Close 函数，否则数据将会损坏。若取值为 CArchive::load：从文件中读取数据；

CArchive::store：向文件中保存数据；CArchive::bNoFlushOnDelete：防止 CArchive 对象在被销毁时自动调用 Flush 函数进行更新。

* 参数 nBufSize 设置缓冲区大小。
* 参数 lpBuf 用于自定义缓冲区，取值为 NULL 则由 CArchive 自行处理。下面是一段示例代码：

CFile file;

file.Open("1.dat", CFile::modeWrite); //打开 1.dat 文件进行写操作

CArchive archive(&file, CArchive::stroe);; //创建一个对 file 进行数据保存的文档对象

2．读写函数

1. 插入运算符（<<）和提取运算符（>>）这两个运算符分别用于保存和读取数据，在原型中可以看到 CArchive 类为许多数据类型

进行了重载，这为数据的串行化提供了基础。

1. Read()和 Write()函数

这两个函数分别从文件中读取或写入原始的字节块。原型为：

|  |
| --- |
| UINT Read( void\* lpBuf,  UINT nMax  );  void Write( const void\* lpBuf,  UINT nMax  ); |

* 参数 lpBuf：给出字节块的指针，用于保存数据或者从中读取数据。
* 参数 nMax：用于确定最大的读写字节数。
* Read()函数返回值：返回实际读写的字节数。

（3）Flush()函数

该函数用于将缓冲区剩余的数据强制写入文件中。原型为：

void Flush( );

注意该函数只保证数据完全传送到指定的 CFile 对象中，而且必须通过 CFile::Close 才能完成最终向磁盘写入数据的过程。

3．状态和标志函数

该类函数主要有 IsLoading()和 IsStoring()两个函数，这两个函数分别用于确定 CArchive 对象是否处于读取状态，或者是否处于保存状态。其原型分别为：

BOOL IsLoading( ) const;

BOOL IsStoring( ) const;

4．Close()函数

该函数用于清空缓冲区，关闭 CArchive 对象，并且将 CArchive 对象与 CFile 对象分离。

其原型为：

void Close( );

### 8.1.4 加入串行化代码

介绍完串行化的基本内容，本节将讲述如何在文档视图应用程序中添加文档类的串行化功能的方法来实现文档的保存和读取。

在 MFC 中，文档数据进行串行化的处理过程，同样是通过文档类的 Serialize()函数进行。下面以选择“ 文件|另存为” 菜单命令为例介绍这个过程。首先 MFC 框架提示用户选取文件名，并打开对应的 CFile 对象。接着框架创建指向该 CFile 对象的 CArchive 对象。由于是“ 另存为”，因此该对象设置为“ 存储”，即 CArchive::store。然后调用 CDocument 派生类即程序的文档类中定义的 Serialize()函数，将 CArchive 对象的引用传递给该函数。程序的文档类的

Serialize()函数执行完毕后，框架先销毁 CArchive 对象，再销毁 CFile 对象。

因此，根据上述过程，在文档视图应用程序中进行文档的串行化处理，只需要在文档类中重载 CDocument::Serialize()函数，对特定的文档数据进行串行化即可实现。下面给出一个在文档类中加入串行化代码的示例代码。该段代码中 CSampleDoc 类是程序的文档类，它有一个 CSampleData 类的成员变量指针，用于保存文档数据。示例演示了 CSampleDoc 类如何将 CSampleData 类的文档数据串行化。

1．CSampleData 类的声明具体代码如下：

|  |
| --- |
| class CSampleData : public CObject  { public: |

|  |  |
| --- | --- |
| }; | //定义的公共数据类型  CString m\_strName; int m\_nType;  CSampleData(); virtual ~CSampleData();    //定义串行化的宏  DECLARE\_SERIAL(CSampleData)    //串行化虚函数  virtual void Serialize(CArchive &ar); |

2．CSampleData 类的实现

具体代码如下：

|  |
| --- |
| //实现串行化的宏  IMPLEMENT\_SERIAL(CSampleData, CObject, 0)    //初始化  CSampleData::CSampleData()  {  m\_nType=0; m\_strName="";  }    CSampleData::~CSampleData()  {    }    //串行化函数  void CSampleData::Serialize(CArchive &ar)  {  //基类的串行化  CObject::Serialize(ar); if (ar.IsStoring())  {  //存储数据  ar<<m\_nType<<m\_strName; |

|  |  |
| --- | --- |
| } | } else  {  //读取数据  ar>>m\_nType>>m\_strName;  } |

3．CSampleDoc 类的声明

具体代码如下：

|  |
| --- |
| class CSampleDoc : public CDocument  {  protected: // create from serialization only  CSampleDoc();  DECLARE\_DYNCREATE(CSampleDoc)    // Attributes public:    // Operations public:    // Overrides  // ClassWizard generated virtual function overrides  //{{AFX\_VIRTUAL(CSampleDoc) public:  virtual BOOL OnNewDocument(); virtual void Serialize(CArchive& ar);  //}}AFX\_VIRTUAL    // Implementation public:  CSampleData \*m\_pData; //这里加入自定义文档数据    virtual ~CSampleDoc();  #ifdef \_DEBUG  virtual void AssertValid() const;  virtual void Dump(CDumpContext& dc) const; #endif |

|  |
| --- |
| protected:    // Generated message map functions protected:  //{{AFX\_MSG(CSampleDoc)  // NOTE - the ClassWizard will add and remove member functions here.  // DO NOT EDIT what you see in these blocks of generated code !  //}}AFX\_MSG  DECLARE\_MESSAGE\_MAP()  }; |

4．CSampleDoc 类的实现

具体代码如下：

CSampleDoc::CSampleDoc()

{

// TODO: add one-time construction code here

// 初始化数据

m\_pData=NULL;

}

CSampleDoc::~CSampleDoc()

{

}

BOOL CSampleDoc::OnNewDocument()

{

if (!CDocument::OnNewDocument()) return FALSE;

// TODO: add reinitialization code here

// (SDI documents will reuse this document)

//创建新的文档数据

if (m\_pData!=NULL) delete m\_pData; m\_pData=new CSampleData;

CString m\_str;

return TRUE;

} /////////////////////////////////////////////////////////////////////////////

// CSampleDoc serialization

// CSampleDoc 的串行化函数

void CSampleDoc::Serialize(CArchive& ar)

{

if (ar.IsStoring())

{

// TODO: add storing code here

//保存 m\_pData ar<<m\_pData;

}

else

{

// TODO: add loading code here

//删除旧的 m\_pData 指向对象

if (m\_pData!=NULL) delete m\_pData;

//读取数据并将数据保存到在堆中创建的 CSampleData 对象中

//m\_pData 指向这个新创建的对象

ar>>m\_pData;

}

}

## 8.2 文件 CFile 类

在 8.1 节中介绍了如何在文档视图应用程序中加入串行化并进行文件的保存和打开。这种串行化方法也有诸多局限，不能实现灵活的文件访问。当程序需要自定义文件操作时，就需要用到 CFile 类。在 MFC 中，CFile 类处理正常的文件 I/O 操作。CFile 类提供通用二进制文件操作的接口，是所有 MFC 文件操作类的基础，串行化也是最终通过 CFile 类进行文件处理的。本节将用多个示例介绍 CFile 类的使用方法，并提供 CFile 类中各种函数的使用参考。

实例 8-1：个人信息录入实例。源代码在光盘中“ \08\实例 8-1\PersonData” 目录下。

PersonData 示例运行界面如图 8-1 所示。该示例是一个基于对话框的 MFC 应用程序。运行时先提示程序用户选择个人信息数据文件，比如 PersonData 工程目录下的 test.dat。打开数据文件后就进入如图 8-1 所示的界面。在这个界面里左侧的列表框可以添加、删除和选择个人信息的条目。当选择定某个人名时，在右侧的“ 个人信息” 栏中显示个人信息的具体内容。如果要修改某个条目的个人信息，则可以直接编辑对话框右侧的具体内容，然后单击“ 更新” 按钮即可。



图 8-1 PersonData 运行界面

PersonData 演示了如何用 CFile 类对文件进行随机读写。在程序代码中可以看到下面几个小节中 CFile 类的各种成员函数的使用范例。

PersonData 用结构 PersonInfo 定义每条个人信息的具体成员，其定义如下：

|  |  |  |
| --- | --- | --- |
| typedef struct PERSONINFO {  char szName[51]; |  | //姓名 |
| char szBirth[31]; |  | //出生日期 |
| int nSex; |  | //性别，0 为男，1 为女 |
| char szTitle[51]; |  | //职务 |
| char szOffice[201]; |  | //办公室 |
| char szOfficePhone[31]; |  | //办公电话 |
| char szHome[201]; |  | //家庭住址 |
| char szHomePhone[31]; |  | //家庭电话 |
| char szMobilePhone[31]; |  | //手机 |
| char szEMail[101];  } PersonInfo; |  | //E-mail 地址 |

PersonData 的主对话框类 CPersonDataDlg 设置了下列几个控件变量，用于显示和修改用户个人信息，代码如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| public:  CListBox m\_PersonList; | | |  |  | //左侧“ 选择人名” 列表框 |
|  | CString | m\_szBirth; |  |  | //“ 出生日期” 输入框 |
|  | CString | m\_szEMail; |  |  | //“ E-mail” 输入框 |
|  | CString | m\_szHome; |  |  | //“ 家庭地址” 输入框 |
|  | CString | m\_szHomePhone; |  |  | //“ 家庭电话” 输入框 |
|  | CString | m\_szMobilePhone; |  |  | //“ 手机” 输入框 |
|  | CString | m\_szName; |  |  | //“ 姓名” 输入框 |
|  | CString | m\_szOffice; |  |  | //“ 办公室” 输入框 |
|  | CString | m\_szOfficePhone; |  |  | //“ 办公电话” 输入框 |
|  | CString | m\_szSex; |  |  | //“ 性别” 下拉列表框 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| CString | m\_szTitle; |  |  |  | //“ 职务” 输入框 |

CPersonDataDlg 根据示例要求，添加了各种消息处理函数，至此添加、删除和修改个人信息的功能完成。这些函数将在 8.2.1 至 8.2.4 小节中结合 CFile 类的使用详细介绍。

### 8.2.1 打开操作

使用文件之前需要先打开文件，该过程就是通过 Windows API 通知 Windows 系统寻找磁盘上的指定文件，然后在内存中创建一个与此文件有关的访问句柄，以后所有的文件操作都需要这个句柄标识操作对象。CFile 类封装了这个过程，并将打开的文件句柄作为成员变量隐藏起来。

在 MFC 中，打开文件具体步骤如下：

* 创建文件 CFile 对象，但不指定路径或权限标志。
* 调用文件 CFile 对象的 Open()成员函数，并提供路径和权限标志。

下面以一组代码为例具体介绍。该代码在PersonData示例中的CPersonDataDialog::InitDialog() 函数中。

|  |
| --- |
| //打开文件  CFile m\_File; //创建文件 CFile 类对象  //用 CFileDialog 选取打开的文件  CFileDialog dlg(TRUE); if (dlg.DoModal()!=IDOK)  return FALSE; //取消则退出程序    if ( !m\_File.Open( dlg.GetPathName(), CFile::modeCreate |  CFile::modeNoTruncate | CFile::modeReadWrite) )  {  //出错处理  MessageBox("无法打开文件，程序退出！","错误"); return FALSE; //退出  } |

以上代码主要是通过 CFileDialog 文件对话框打开指定的文件。首先创建一个文件对象 m\_File，再用文件对话框选择打开的文件名，然后用 m\_File.Open()函数打开 dlg.GetPathName() 指定的文件。Open 函数的第二个参数设置了打开文件的标志，CFile::modeCreate 表示创建一个 新 文 件 ， CFile::modeNoTruncate 表 示 创 建 时 不 将 已 经 存 在 的 文 件 清 空 ， CFile::modeReadWrite 表示打开的文件可以读取也可以写入。fileException 在 Open 函数中用于获得打开失败产生的异常，用于 if 语句中的失败信息的输出。而 Open()函数的返回值确定打开是否成功，TRUE 代表成功，FALSE 代表失败。

技巧：在打开文件前一般都可以使用文件对话框 CFileDialog 指定打开的文件名。

除了上面的打开方式，MFC 还提供了其他的打开文件的步骤和方法。这主要是因为打开文件既可以通过 CFile 的构造函数，也可以通过 CFile::Open()函数完成。

CFile 的构造函数原型如下：

CFile( );

CFile(

HANDLE hFile

);

CFile(

LPCTSTR lpszFileName,

UINT nOpenFlags

);

可以看出 CFile 有 3 个构造函数：

1. 没有任何参数的构造函数仅仅创建一个 CFile 对象，并不打开任何文件。
2. 有一个参数的构造函数将创建的新的 CFile 对象绑定到一个已经打开的文件句柄中。文件句柄可以用 Windows API 函数 CreateFile 打开，或者从已经打开的 CFile 对象的 m\_hFile 获得。参数 hFile 表示要绑定的文件句柄。
3. 有两个参数的构造函数用于创建 CFile 对象同时打开文件。

* 参数 lpszFileName：确定要打开的文件路径名。
* 参数 nOpenFlags：用于确定打开的参数，通过按位或将表 8-1 中的各种取值组合而成。注意表 8-1 中以 share 开头的几个参数用于设置文件打开时的共享模式，即文件打开后是否允许其他进程打开同一个文件并进行操作。CFile::shareDenyNone 允许其他进程同时打开文件并进行处理；CFile::shareDenyRead 和 CFile::shareDenyWrite 依然允许其他进程同时打开文件，但只能允许它们进行写操作或者进行读操作；CFile::shareExclusive 则完全不允许其他进程同时打开文件。如果违反了共享模式，打开失败。

表 8-1 打开文件的参数

|  |  |
| --- | --- |
| nOpenFlags 取值 | 参数说明 |
| CFile::modeCreate | 直接创建新文件。如果文件已经存在，就将文件清空 |
| CFile::modeNoTruncate | 用于和 CFile::modeCreate 组合，使得当文件已存在时，不清空文件 |
| CFile::modeRead | 以只读方式打开 |
| CFile::modeWrite | 以只写方式打开 |
| CFile::modeReadWrite | 以既可以读又可以写的方式打开 |
| CFile::modeNoInherit | 防止子进程继承打开的文件 |
| CFile::shareDenyNone | 打开文件但不禁止其他进程读或写访问。如果文件已被其他进程以兼容模式打开，则创建失败 |
| CFile::shareDenyRead | 打开文件同时禁止其他进程的读访问。如果文件已被其他进程以兼容模式打开，则创建失败 |
| CFile:shareDenyWrite | 打开文件同时禁止其他进程的写访问。如果文件已被其他进程以兼容模式打开，则创建失败 |
| CFile::shareExclusive | 以独占模式打开文件，禁止其他进程对文件的读写 |
| CFile::typeText | 设置为文本模式打开，对回车换行设置特殊的处理（CFile 本身只是二进制的文件处理，并不支持这一参数，这一参数仅用于 CFile 派生类） |
| CFile::typeText | 设置为二进制模式打开（仅用于 CFile 派生类） |

使用 CFile 的构造函数打开文件，一般需要提供一个 TRY/CATCH 的异常捕获机制以保证打开文件失败时的处理，代码如下：

TRY //进入异常操作处理块

{

//创建 test.dat 文件并用于写入

CFile file("test.dat", CFile::modeCreate | CFile::modeWrite);

}

CATCH(CFileException, e) //如果打开失败则进入异常捕获处理块，e 是产生的文件异常

{

TRACE("Error = %u", e->m\_cause); //输出失败原因

}

END\_CATCH

Visual C++中还可以使用 CFile::Open()函数打开文件，这时需要在创建 CFile 对象时选用第一个没有参数的构造函数。

CFile::Open()函数的原型为：

virtual BOOL Open(

LPCTSTR lpszFileName,

UINT nOpenFlags,

CFileException\* pError = NULL

);

该函数的第一个参数和第二个参数跟 CFile 的构造函数中的对应参数意义相同。而第三个参数可以获得打开文件失败产生的 CFileException 对象的指针，这样就可以获得打开失败的具体原因，CFileException 类将在 8.2.5 小节中叙述。同时，CFile::Open()函数的返回值还给出了打开操作是否成功。当返回 FALSE 时，打开失败，就可以用 pError 获得失败的原因。而如果 pError 为 NULL 时，则不会获得 CFileException 对象。

### 8.2.2 读写操作

打开文件后就可以用读写操作对文件数据进行读写。本节开头提到 CFile 是一个通用的二进制文件操作接口，故通过 CFile 只能以二进制方式读写文件。CFile 的读写操作主要通过

CFile::Read()和 CFile::Write()函数完成。

CFile::Read()函数用于从文件中读取数据，原型为：

virtual UINT Read( void\* lpBuf,

UINT nCount

);

CFile::Write()函数用于向文件写入数据，原型为：

virtual void Write( const void\* lpBuf,

UINT nCount

);

这两个函数具有同样的参数。

* 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：给出可以从文件中读出或者向文件中写入的字节数的最大值。对于文本模式的文件，回车换行只作为一个字符。
* Read()函数返回值：用于获得传输到缓冲区的字节数。如果到达文件尾，则返回值可能比 nCount 小。下面给出 PersonData 示例中读文件的语句（在 CPersonDataDlg::InitDialog()函数中）：

//读取个人信息数据

int result; //获得返回值

PersonInfo pi; //用户信息缓冲区

//循环知道读入的数据字节数为 0

while (result=m\_File.Read(&pi,sizeof(PersonInfo)))

{

//在列表中添加人名

m\_PersonList.AddString(pi.szName);

}

上述代码中 m\_File 是一个已经打开文件的 CFile 对象，代码循环从文件中读取 PersonInfo

结构，并送至 pi 变量中，然后在对话框的列表框中添加人名。

下面是 PersonData 示例写文件的语句（在 CPersonDataDlg::UpdateFile()函数中）：

|  |
| --- |
| PersonInfo pi; //创建个人信息变量  //从对话框的空间变量中获得个人信息  strcpy(pi.szName,m\_szName); if (m\_szSex == "男" )  pi.nSex = 0;  else  pi.nSex = 1;  strcpy(pi.szBirth,m\_szBirth); strcpy(pi.szTitle,m\_szTitle); strcpy(pi.szOffice,m\_szOffice);  strcpy(pi.szOfficePhone,m\_szOfficePhone); strcpy(pi.szHome,m\_szHome);  strcpy(pi.szHomePhone,m\_szHomePhone); strcpy(pi.szMobilePhone,m\_szMobilePhone); strcpy(pi.szEMail,m\_szEMail);  //写入个人信息  m\_File.Write(&pi,sizeof(PersonInfo)); |

注意 CFile::Read()和 CFile::Write()都不提供缓冲机制，所有的数据都是立即读取或者立

即写入的，这与 CArchive 类不同，CArchive 类提供写缓冲。同时 CFile::Write()还可能由于写入失败而抛出 CFileException 异常，例如磁盘已满或者

磁盘写保护等，同样需要 TRY/CATCH 结构对这些异常进行捕获。

对于写入操作，CFile 还提供了 CFile::Flush()用于将任何留在文件缓冲区中的数据强制写入磁盘文件，原型为：

virtual void Flush( );

注意 CFile::Flush()并不能保证强制将留在 CArchive 缓冲区的数据写入文件，必须先使用

CArchive::Flush()函数。

### 8.2.3 定位操作

CFile 可以使程序随机读写文件，这一随机性就是通过定位操作完成的。定位操作用于定位 CFile 的读写指针，使得程序可以读写任意位置的数据，而不是在顺序读写中只能依照从前往后的顺序进行读写。

CFile 为定位操作提供了下列成员函数。

1．CFile::GetLength()函数

CFile::GetLength()函数用于获得打开文件的长度，即文件的字节数，原型为：

virtual DWORD GetLength( ) const;

函数返回值为文件的长度。

下面的代码用于获得 test.dat 文件长度：

|  |  |
| --- | --- |
| CFile file;  file.Open("test.dat", CFile::modeRead);  DWORD dwLength = file.GetLength(); | //获得文件长度 |

2．CFile::SetLength()函数

CFile::SetLength()函数用于设置打开文件的长度，即文件的字节数，原型为：

virtual void SetLength(

DWORD dwNewLen

);

参数 dwNewLen 用于制定新的文件长度。下面的代码用于设置 test.dat 文件长度：

CFile file;

file.Open("test.dat", CFile::modeWrite);

file.SetLength(1024); //设置文件长度为 1024 字节，即 1kB

3．CFile::Seek()函数

CFile::Seek()函数用于重新定位之前打开的文件的读写指针，以实现随机访问。其原型为：

virtual ULONG Seek( LONG lOff,

UINT nFrom

);

* 参数 lOff：用于确定指针移动的字节数，正的数值表示指针向后移动，负的数值则表示指针向前移动。
* 参数 nForm：用于确定指针移动的模式，可以为下列值之一：

· CFile::begin：从文件开头把指针向后移动 lOff 字节。

· CFile::current：从当前读写指针的位置开始把指针向后移动 lOff 字节。

· CFile::end：从文件结尾向前移动指针，注意此时 lOff 必须为负的，表示向前移动。

* 函数返回值：指针新的相对于文件开头的字节偏移量。如果移动的位置非法，则返回

值未定义，并抛出 CFileException 异常。

CFile::Seek()函数通过将读写指针移动一定量实现随机访问文件的内容。指针的移动可以是绝对的或者相对的，在移动过程中没有实际读写文件。当文件打开时，文件指针在偏移量

0 处，即文件开头。

PersonData 示例的 CPersonDataDlg::OnDel()函数给出了一个通过 CFile::Seek()随机访问文件的例子。该函数用于将文件中的某项内容删除，这是通过将该项内容后面的内容前移，然后重新设置文件长度完成的，代码如下：

|  |
| --- |
| //获得当前选项  int sel = m\_PersonList.GetCurSel();  //删除文件中的个人信息  for (int i = sel; i < m\_PersonList.GetCount()-1; i++) {  PersonInfo pi;  //将第 i+1 项前移至第 i 项处  m\_File.Seek((LONG)(i+1)\*sizeof(PersonInfo),CFile::begin); m\_File.Read(&pi,sizeof(PersonInfo));  m\_File.Seek((LONG)i\*sizeof(PersonInfo),CFile::begin); m\_File.Write(&pi,sizeof(PersonInfo));  }  m\_File.Flush();  //设置文件的新长度  m\_File.SetLength((LONG)(m\_PersonList.GetCount()-1)\*sizeof(PersonInfo)); |

m\_PersonList 是 PersonData 对话框中的列表框控件，用于获得程序用户当前的选择和个人信息的项目总数。上面的代码从 i=sel 开始，将第 i+1 项 PersonInfo 结构读出然后覆盖写到第 i 项的位置，用于将 i=sel 后面的数据往前移。最后设置文件的新长度就可以完成将第 i=sel 项的 PersonInfo 数据从文件中删除。

4．CFile::SeekToBegin()函数

CFile::SeekToBegin()用于将文件指针移至文件开头，相当于 Seek(0L, CFile::begin)。原型为：

void SeekToBegin( );

5．CFile::SeekToEnd()函数

CFile::SeekToEnd()指针用于将文件指针移至文件结尾，相当于 Seek(0L, CFile::end)。原型为：

DWORD SeekToEnd( );

函数返回值为文件的字节长度。

6．CFile::GetPosition()函数

CFile::GetPosition()函数用于获得文件指针的当前值，原型为：

|  |  |
| --- | --- |
| virtual DWORD GetPosition( ) const; | |
| 函数返回值为文件指针当前相对于文示例如下： | 件开头的字节偏移量。 |
| DWORD dwPos=file.GetPosition(); | //file 是一个已定义并打开文件的 CFile 对象 |

### 8.2.4 关闭操作

当文件完成读写后，需要关闭文件，释放文件句柄。CFile 提供了两种文件关闭的方式。

1．CFile::Close()函数

该函数关闭与对象关联的文件，并使得文件不能继续读写。原型为：

virtual void Close( );

注意，当销毁 CFile 对象而没有关闭文件时，CFile 的析构函数自动关闭该文件。而如果 CFile 对象是在堆中用 new 创建的，则必须关闭文件后删除该对象，以防止内存泄露。

CFile::Close()函数将 m\_hFile 文件句柄成员变量设置为 CFile::hFileNull。示例代码如下：

|  |
| --- |
| TRY //进入异常操作处理块  {  //创建 test.dat 文件并用于写入  CFile pfile=new CFile(est.dat", CFile::modeCreate | CFile::modeWrite);    ⋯ ⋯ //文件的写入操作    pfile->Close(); //关闭文件 delete pfile; //删除文件对象  }  CATCH(CFileException, e) //如果打开失败则进入异常捕获处理块，e 是产生的文件异常  {  TRACE("Error = %u", e->m\_cause); //输出失败原因  }  END\_CATCH |

2．CFile::Abort()函数

该函数同样关闭与对象关联的文件，并使得文件不能继续读写。原型为：

virtual void Abort( );

当处理异常时，CFile::Abort()函数与 CFile::Close()函数有两个重要的不同点。首先， CFile::Abort()函数不会因为失败而抛出异常，这是因为 Abort 函数忽略失败。其次，Abort() 函数不会确认文件是否打开或者已经被关闭。其他方面与 CFile::Close()函数相同。可见

CFile::Abort()一般用于异常处理中，示例代码如下：

|  |
| --- |
| TRY //进入异常操作处理块  {  //创建 test.dat 文件并用于写入  CFile file(est.dat", CFile::modeCreate | CFile::modeWrite);  }  CATCH(CFileException, e) //如果打开失败则进入异常捕获处理块，e 是产生的文件异常  {  TRACE("Error = %u", e->m\_cause); //输出失败原因 file.Abort(); //安全地关闭文件  }  END\_CATCH |

### 8.2.5 异常操作

在文件读写过程中，由于是访问磁盘硬件，因此会产生各种各样的异常。无论是文件的打开、读写、定位还是关闭操作都会产生异常。异常通常是由调用的函数抛出，在 TRY/CATCH 块中捕获，此外 CFile::Open()函数还可以用参数的形式获得异常的指针。

MFC 中文件的异常类是 CFileException。CFileException 对象可以获得文件操作失败的原因代码。该对象在 CFile 成员函数中被创建和产生，也可以在 CFile 派生类的成员函数中创建和产生。程序用 CATCH 表达式获取这一对象。CFileException 类包含可移植的原因代码和操作系统制定的错误值的公共成员变量。一般情况下使用可移植的原因代码以实现平台的无关性。

下面对 CFileException 相关操作进行介绍，包括如何创建、抛出和获得异常。

1．创建并抛出异常

MFC 不允许手动创建 CFileException 对象，而采用 AfxThrowFileException()函数创建并抛出文件异常。AfxThrowFileException()函数的原型为：

|  |
| --- |
| void AfxThrowFileException( int cause,  LONG lOsError = -1,  LPCTSTR lpszFileName = NULL  ); |

* 参数 cause：指定异常的原因，是一个枚举类型，具体枚举值如表 8-2 所示。
* 参数 lOsError：指定一个跟操作系统相关的异常原因。注意该值需要程序自行提供，操作系统并不自动设置该值。
* 参数 lpszFileName：指定出错的文件名。

表 8-2 文件异常原因代码

|  |  |
| --- | --- |
| cause 取值 | 取值说明 |
| CFileException::none | 没有错误发生 |
| CFileException::generic | 一个没有特定意义的错误发生 |
| CFileException::fileNotFound | 无法找到文件 |
| CFileExceptionLLbadPath | 路径错误 |
| CFileException::tooManyOpenFiles | 超出允许的打开文件的最大数目 |
| CFileException::accessDenied | 文件无法访问 |
| CFileException::invalidFile | 尝试使用一个无效的文件句柄 |
| CFileException::removeCurrentDir | 当前工作目录无法被删除 |
| CFileException::directoryFull | 没有更多的目录项 |
| CFileException::badSeek | 移动文件指针错误 |
| CFileException::hardIO | 硬件错误 |
| CFileException::sharingViolation | 共享错误 |
| CFileException::lockViolation | 锁定错误 |
| CFileException::diskFull | 磁盘已满 |
| CFileException::endOfFile | 到达文件末尾 |

用户可以在自定义函数中添加自定义语句 AfxThrowFileException 以抛出自定义的文件异常，例如有一个 CSampleFile 类用于指定存取一个有最大长度的文件，这个最大长度设置为 m\_dwMaxLength，如果超出这个长度，就存取失败。下面给出一个写文件的例子，如果写入的数据使得文件长度超出 m\_dwMaxLength，则模拟磁盘满的出错异常。

|  |
| --- |
| //向文件中写入字符串 lpszString  void CSampleFile::WriteString(LPCTSTR lpszString) {  //m\_File 是 CSampleFile 的成员变量  DWORD dwLen=m\_File.GetLength()+strlen(lpszString)+1;//获得写入后的文件大小 if (dwLen>m\_dwMaxLength)  AfxThrowFileException(CFileException::diskFull); //抛出磁盘满的异常 m\_File.Write(lpszString,strlen(lpszString)+1); //如果没超过限度，则写入文件  } |

2．获得文件异常信息

从第 1 部分创建 CFileException 异常中可以看出，文件异常有 3 类不同的信息，分别由

3 个成员变量表示。

* CFileException::m\_cause：int 类型，获得可移植的出错代码，这些代码同前面

AfxThrowFileException()函数的参数 cause 的代码相同。

* CFileException::m\_lOsError：long 类型，获得操作系统指定的出错代码。
* CFileException::m\_strFileName：CString 类型，获得出错的文件名。

下面给出用 TRY/CATCH 块获得文件异常信息的示例代码：

TRY

{

//创建文件进行写入

CFile f( pFileName, CFile::modeCreate | CFile::modeWrite );

}

CATCH( CFileException, e )

{

//文件未找到

if( e->m\_cause == CFileException::fileNotFound )

TRACE( "ERROR: File not found\n")

}

END\_CATCH

### 8.2.6 管理操作

对文件的操作，除了打开文件并进行读写外，还可以进行删除、重命名等操作。CFile

类也为这些操作提供了支持。

1．CFile::GetFilePath()

该函数用于获取指定文件的全路径。例如当 CFile 打开了 c:\aaa\bbb.dat 文件，调用

GetFilePath()就可以获得该文件的路径 c:\aaa\bbb.dat。其原型为：

virtual CString GetFilePath() const;

函数返回值即为指定文件的完整路径。

2．CFile::GetFileTitle()

该函数用于获取指定文件的标题。例如当 CFile 打开了 c:\aaa\bbb.dat 文件，调用

GetFileTitle()就可以获得该文件的标题 bbb。其原型为：

virtual CString GetFileTitle() const;

函数返回值即为指定文件的标题。

3．CFile::GetFileName()

该函数用于获取指定文件的名字。例如当 CFile 打开了 c:\aaa\bbb.dat 文件，调用

GetFileName()就可以获得该文件的名字 bbb.dat。其原型为：

virtual CString GetFileName() const;

函数返回值即为指定文件的名字。

4．CFile::SetFilePath()

该函数用于设置文件的路径。该函数在下面的情况中使用，即当一个 CFile 对象创建时没有指定文件的路径。注意该函数并不能修改文件的实际路径，也不打开或创建新的文件。其原型为：

virtual void SetFilePath(

LPCTSTR lpszNewName

);

参数 lpszNewName 用于指向新路径的字符串。示例代码如下：

//用第二个构造函数创建一个与已经打开的文件句柄 hFile 绑定的 CFile 对象

CFile myFile((int) hFile);

//此时虽然 myFile 已经通过 hFile 获得了打开文件的句柄，但是它并不知道 hFile 的文件名， //因为 hFile 并不包含文件名的信息。

//如果没有设置好与其关联的文件名，则异常对象 CFileException::m\_strFileName 将没有定义，

//无法获得出错的文件名。因此需要用 SetFilePath 设置文件路径名。这里使用 pstrName 指定。

myFile.SetFilePath(pstrName);

5．CFile::GetStatus()

该函数用于获得文件的状态，有两个不同的原型：非静态成员变量的原型、静态成员变

量的原型。

BOOL GetStatus(

CFileStatus& rStatus

) const;

static BOOL PASCAL GetStatus( LPCTSTR lpszFileName,

CFileStatus& rStatus

);

前一个非静态成员函数用于获得 CFile 打开的文件的状态，而后一个静态成员函数用于

获得指定文件的文件状态。

* 参数 rStatus：CFileStatus 结构，用于获得文件状态。CFileStatus 结构如表 8-3 所示。
* 参数 lpszFileName：用于向静态成员函数提供要获取的文件名，可以是相对的路径也可以是绝对路径。
* 函数返回值：确定获取是否成功。

注意，GetStatus()的非静态版本获取与 CFile 对象有关的文件状态，不把值写入

CFileStatus::m\_szFullName 结构成员中。而 GetStatus()的静态版本用于获取文件的状态，同时将文件名复制到 CFileStatus::m\_szFullName 中。此文件从目录项中获取文件状态而不打开文件，这对于测试文件的存在性和访问权限十分有用。表 8-3 CFileStatus 结构说明

|  |  |
| --- | --- |
| 成员定义 | 成员说明 |
| CTime m\_ctime | 文件创建的日期和时间 |
| CTime m\_mtime | 文件最后修改的日期和时间 |
| CTime m\_atime | 文件最后读取的日期和时间 |
| LONG m\_size | 文件的逻辑字节长度 |
| BYTE m\_attribute | 文件的属性字节状态 |
| Char m\_szFullName[\_Max\_PATH] | 以 Windows 字符集出现的绝对的文件名 |

CFileStatus::m\_attribute 代表文件属性，MFC 在 CFile 类中提供一个 enum 类型用于以符号的形式指定属性，具体代码如下：

|  |  |  |
| --- | --- | --- |
| enum Attribute {  normal = 0x00, |  | //普通文件 |
| readOnly = 0x01, |  | //只读文件 |
| hidden = 0x02, |  | //隐藏文件 |
| system = 0x04, |  | //系统文件 |
| volume = 0x08, |  | //分区文件 |
| directory = 0x10, |  | //目录文件 |
| archive = 0x20  }; |  | //存档文件 |

函数示例如下：

|  |
| --- |
| CFileStatus status; //文件状态变量  extern CFile cfile;  if( cfile.GetStatus( status ) ) // virtual member function {  TRACE("File size = %ul\n", status.m\_size);  }  char\* pFileName = "test.dat"; //文件名  if( CFile::GetStatus( pFileName, status ) ) // 调用静态成员函数  {  TRACE("File name = %s\n", status.m\_szFullName);  } |

6．CFile:SetStatus()

该函数用于设置文件的状态，只有静态版本。其原型为：

static void PASCAL SetStatus( LPCTSTR lpszFileName, const CFileStatus& status

);

* 参数 lpszFileName：指定要设置文件状态的文件名，可以是相对也可以是绝对路径。 • 参数 status：确定要设置的新的文件状态，该参数是 CFileStatus 结构类型，具体内容见 CFile::GetFileStatus()的介绍。一般情况下需要先获取原来的状态，再进行修改，否则会造成一些需要保留的状态被修改。

函数示例如下：

|  |  |  |  |
| --- | --- | --- | --- |
| char\* pFileName = "test.dat"; |  |  | //需要修改的文件名 |
| CFileStatus status; |  |  | //文件状态变量 |
| CFile::GetStatus( pFileName, status ); |  |  | //获取旧的文件状态 |
| status.m\_attribute |= CFile::readOnly; |  |  | //添加只读属性 |
| CFile::SetStatus( pFileName, status ); |  |  | //设置新的属性 |

7．CFile:Rename()

该函数用于更改文件名，是 CFile 的静态成员函数。其原型为：

static void PASCAL Rename( LPCTSTR lpszOldName,

LPCTSTR lpszNewName

);

* 参数 lpszOldName：指定文件原来的路径名。
* 参数 lpszNewName：指定文件新的路径名。

注意，该函数不能改变目录的名字。

8．CFile:Remove()

该函数用于删除指定的文件，也是一个静态成员函数。其原型为：

static void PASCAL Remove( LPCTSTR lpszFileName

);

参数 lpszOldName 用于指定要删除文件的路径名，可以为相对路径也可以为绝对路径。

注意该函数不能删除目录。如果指定的文件已打开或者文件不能被删除，Remove()函数将抛出异常。

实例 8-2：文件操作实例。源代码在光盘中“ \08\实例 8-2\FileManipulate” 目录下。

FileManipulate 示例是一个基于对话框的 MFC 应用程序，其界面如图 8-2 所示。该示例实现了文件的删除、复制和移动功能。选择好原始文件和目标文件（可以自行输入也可以单击对应的“ 打开” 按钮打开文件对话框进行选择），然后单击对话框底部的按钮可以实现如下的文件操作：

* “ 删除原始文件” 按钮：用于删除原始文件输入框所指定的文件。
* “ 复制” 按钮：用于将原始文件复制到目标文件。 • “ 移动” 按钮：用于将原始文件移动到目标文件。



图 8-2 “ 文件操作” 对话框

删除文件的代码在 CFileManipulateDlg::OnDel()中，用到了 CFile::Remove()静态成员函数进行删除。

复制文件的代码在 CFileManipulateDlg::OnCopy()中，其思想是将原始文件的内容分几次读入内存中，再将内存中的数据依次写入目标文件，代码如下：

|  |  |
| --- | --- |
| pSrcFile=new CFile(m\_szSrcFile, CFile::modeRead); | //原始文件 |

|  |
| --- |
| pDstFile=new CFile(m\_szDstFile, CFile::modeCreate|  CFile::modeWrite); //目标文件  int nRead; //读取的字节数  char szBuffer[1024]; //复制缓冲区，用于将读取的数据缓存，然后写入    //循环直到读取的字节数为 0  while (nRead=pSrcFile->Read(szBuffer,sizeof(szBuffer)))  {  pDstFile->Write(szBuffer,nRead);  }    pSrcFile->Close(); pDstFile->Close(); delete pSrcFile; delete pDstFile; |

移动文件的代码在 CFileManipulateDlg::OnMove()中，其主要思想是将原始文件复制到目标文件后再删除原始文件。

|  |
| --- |
| 技巧：CFile::Remove()函数不能删除目录，而 Windows API 提供了 RemoveDirectory()函数，用于删除一个空目录，对于目录中的文件需要自行删除。MFC 中提供 CFileFind 类用于枚举目录 中 的 子 文 件 ， 使 用 这 个 类 的 功 能 枚 举 出 目 录 中 的 所 有 文 件 并 删 除 ， 再 使 用  RemoveDirectory()删除空目录即可完成删除目录的功能。 |

下面介绍删除目录的详细过程。首先需要枚举目录文件。枚举目录文件是 Windows 系统中一个重要的操作。枚举目录文件是通过 CFileFind 类实现的。CFileFind 是 MFC 中实现本地文件搜索的类，包含开始搜索，定位文件，获得的文件信息的成员函数。

CFileFind 类的构造函数没有参数。而开始一个搜索，需要调用 CFileFind::FindFile()函数。

该函数打开一个文件搜索，原型为：

virtual BOOL FindFile(

LPCTSTR pstrName = NULL,

DWORD dwUnused = 0

);

* 参数 pstrName：指定文件的搜索路径和搜索类型，即可以使用通配符“ \*”和“?”。

如果给出 NULL 值，则 FindFile()进行“ \*.\*” 的搜索。

* 参数 dwUnused：为 CFileFind 的派生类保留，必须为 0。
* 返回值：确定打开搜索是否成功，失败的话可以通过 GetLastError 函数获得失败代码。打开搜索后，就可以使用CFileFind::FindNextFile()函数获得下一个文件。CFileFind::FindNextFile() 函数原型为：

virtual BOOL FindNextFile( );

函数返回值用于确定是否还有文件没找到，如果为 FALSE 则说明这是最后一个找到的文件或者出错，可以用 GetLastError 函数获得出错信息。搜索到一个文件时可以用下列一些函数获得文件的信息：

* GetCreationTime()：获得文件创建时间。
* GetFileName()：获得文件名。
* GetFilePath()：获得文件全路径。
* GetFileTitle()：获得文件标题。
* GetLastAccessTime()：获得文件最后一次访问的时间。
* GetLastWriteTime()：获得文件最后一次被写入的时间。
* GetLength()：获得文件字节大小。
* GetRoot()：获得文件的根目录。
* IsArchived()：确定文件是否是存档文件。
* IsDirectory()：确定文件是否是目录。
* IsDots()：确定文件是否是“ .” 或者“ ..”，这种类型的文件表示目录。
* IsHidden()：确定文件是否是隐藏文件。
* IsNormal()：确定文件是否是普通文件。
* IsReadOnly()：确定文件是否是只读文件。
* IsSystem()：确定文件是否是系统文件。
* IsTemporary()：确定文件是否是临时文件。最后用 CFileFind::Close()函数关闭搜索。

介绍了 CFileFind 类的使用方法后，可以想到，删除目录可以用递归的方法将目录中的所有子目录删除，即在本级目录中如果找到子目录文件，则调用自身删除这个子目录。代码如下：

//CMyFile 是一个自定义功能的文件操作类

//DeleteTree 函数用于删除目录，并同时删除目录中的所有子目录和文件

//参数 strDir 用于确定要删除的目录名

void CMyFile::DeleteTree(CString strDir)

{

CFileFind finder; //创建一个目录搜索类

CString delFile=strDir;

delFile+="\\\*.\*"; //设置搜索目录的通配符

//开始搜索，bWorking 用于表示是否搜索到新文件

BOOL bWorking = finder.FindFile(delFile);

//如果 bWorking 为 TRUE 则继续循环

while (bWorking)

{

//获取下一个文件

bWorking = finder.FindNextFile();

//获取找到的文件名

CString filename=finder.GetFilePath();

//如果文件是一个目录，而且不是一个“ .” 或者“ ..” 的目录

//那么删除这个子目录

if(finder.IsDirectory()&&(!finder.IsDots()))

{

//递归调用自身删除子目录

Delete(filename);

} else if (!finder.IsDots()) //如果文件不是一个“ .” 或者“ ..” 的文件则删除该文件

{

CFile::Remove(filename); //删除文件

}

}

//文件查找完毕，所有文件都已经删除，目录清空

RemoveDirectory(strDir); //删除空目录

}

## 8.3 文件对话框

第 4 章已经介绍过文件对话框的基本使用方法。现在介绍一些关于 CFileDialog 的高级使用方法，包括多选文件对话框的使用方法和重载文件对话框的方法，使得程序能更灵活地控制文件对话框。

### 8.3.1 多选文件对话框

当构造 CFileDialog 类时指定了 OFN\_ALLOWMULTISELECT 的属性，则 CFileDialog 允许用户选择多个文件，如图 8-3 所示。

当设置允许多选的文件对话框时，与普通使用惟一的区别就是如何获得所有选定的文件。这就用到 CFileDialog::GetStartPosition()和 CFileDialog::GetNextPathName()两个函数。



图 8-3 可以多选的打开文件对话框

CFileDialog::GetStartPosition()函数用于获得选择列表中的第一个文件路径名的位置，其原型为：

POSITION GetStartPosition( ) const;

函数返回值用于枚举选择的文件。若为 NULL，则说明选择列表为空。

CFileDialog::GetNextPathName()函数用于获得下一个选择的文件名，其原型为：

CString GetNextPathName(

POSITION& pos

) const;

* 参数 pos：指示前一个 GetNexPathName()或 GetStartPosition()函数设置的 POSITION值。当 GetNextPathName()调用返回时会修改该参数，如果值为 NULL，则说明到达列表结尾。
* 函数返回值：获得下一个选择的文件路径名。

于是可以得到下面的代码：

//创建一个可以多选的文件对话框

CFileDialog dlg(TRUE, NULL,NULL,OFN\_ALLOWMULTISELECT);

if (dlg.DoModal()==IDOK) //如果返回为确定，获得选择的文件名

{

POSITION pos;

pos= dlg.GetStartPosition(); //开始遍历用户选择列表 while (pos!=NULL) //如果列表中还有新的文件

{

CString filename=GetNextPathName(pos); //获得新的被选定的文件

⋯ ⋯ //对 filename 的文件进行处理

}

}

### 8.3.2 重载文件对话框

当用户需要文件对话框有更多的功能，则可以通过重载 CFileDialog 类，并为文件对话框

添加新的控件完成。

本节中以一个重载文件对话框的例子介绍重载的步骤。该例为文件对话框添加一个“ 删

除” 按钮，用于删除文件对话框中选定的文件，如图 8-4 所示。实现这个自定义的文件对话框主要分为下列 3 个步骤：

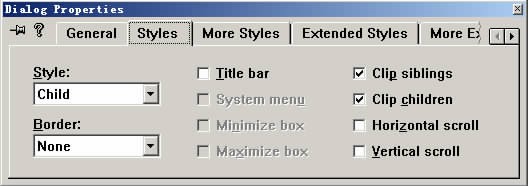


图 8-4 “ 打开” 对话框

1．创建一个对话框模板

首先向工程的资源文件中创建一个新的对话框模板，将 ID 设为 IDD\_MYFILEDIALOG。

然后设置模板风格属性如图 8-5 所示。

 图 8-5 设置对话框模板的风格属性

然后向模板中添加“ 删除” 按钮，ID 设置为 IDC\_DEL，并重新调整对话框的大小，如图 8-6 所示。

注意，在图 8-6 中可以看到，新添加包含“ 删除” 按钮的对话框模板是放在文件对话框的底部。因此需要将对话框的模板设置为子窗口而且无边框，更重要的是要复选“ Clip siblings” 和“ Clip children” 两个属性，使得文件对话框的原始控件可以得到恰当的重绘。

2．重载 CFileDialog 类

在ClassWizard中创建一个CFileDialog的派生类CMyFileDialog。然后修改CMyFileDialog 的构造函数，代码如下：

//构造函数创建一个新的打开文件对话框，没有后缀名过滤器和默认的后缀名

//参数中设置 OFN\_ENABLETEMPLATE 用于从自定义模板中添加新的控件

CMyFileDialog::CMyFileDialog() :

CFileDialog(TRUE, NULL, NULL,

OFN\_FILEMUSTEXIST|OFN\_ENABLETEMPLATE, NULL, NULL)

{

//设置文件对话框的模板为 IDD\_MYFILEDIALOG

//这里有两个参数分别对应于老版本和新版本的文件对话框

//这里将两类版本的文件对话框模板都进行设置

|  |  |
| --- | --- |
| } | SetTemplate(IDD\_MYFILEDIALOG,IDD\_MYFILEDIALOG); |

然后在 ClassWizard 中为 CMyFileDialog 添加 IDC\_DEL 控件的 BN\_CLICKED 处理函数

OnDel()，代码如下：

void CMyFileDialog::OnDel()

{

// TODO: Add your control notification handler code here

// 提示用户是否删除文件

if (MessageBox("确定删除文件？","提示",MB\_OKCANCEL)!=IDOK)

{

//不删除文件则退出

return;

}

CFile::Remove(GetPathName()); //删除选定的文件

}

3．使用重载的文件对话框

创建好 CMyFileDialog 就可以在程序中使用重载后的对话框，使用方法跟普通的文件对话框类似，代码如下：

|  |
| --- |
| CMyFileDialog dlg; //创建 CMyFileDialog 对象 if (dlg.DoModal()==IDOK) //打开对话框  {  ⋯ ⋯ //进行用户选定文件后的处理  } |

这样就能打开如图 8-4 所示的对话框，在选择文件打开的同时还可以通过单击“ 删除” 按钮删除一个选定的文件，为文件对话框提供了附加的功能。

本小节给出的例子稍加修改就可以添加更多更复杂的功能。同时 CFileDialog 还提供了一些虚函数用于自定义处理文件对话框原有控件产生的消息，重载这些函数就可以为

CFileDialog 原有的功能进行重新设计，从而大大提高文件对话框的灵活性。

## 8.4 本章小结

本章主要介绍了在 MFC 中如何对文件进行读写操作和一些管理操作，主要内容包括以下几个方面：

* 在文档视图结构中对文档进行串行化；
* 用 CFile 类和相关类对磁盘文件进行读写和管理；
* 文件对话框 CFileDialog 的一些高级使用方法。

总之，文件操作是应用程序保存和使用数据的基本方法。只有熟练使用本章中介绍的各种文件操作的类和函数才能在程序中体现出更大的灵活性。

# 第 9 章 多线程

本章主要介绍 Windows 操作系统下多线程的基本概念、创建管理线程的方法，以及线程的同步问题。Windows 提供了大量的线程同步技术，这些技术可以应用于不同的实际情况。同时，MFC 对线程操作进行了封装，提供了支持线程操作的类库。本章主要讨论这些技术，并且给出相应的实例。主要包括以下几个方面的内容：

* Windows 下多线程的基本概念；
* 用户界面线程和工作者线程；
* 线程的管理操作；
* 线程的同步；
* 多线程编程实例。

## 9.1 多线程的基本概念

在 Win32 下，一个应用程序由一个或多个进程组成。一个进程由一个或多个线程以及代码、数据和其他内存中的程序资源组成。典型的程序资源包括打开的文件、信号量、动态分配的内存等。线程在进程空间中执行。

线程是操作系统分配处理器时间的最小单位。每个线程有自己的堆栈，CPU 寄存器，以及程序入口。每个线程共享所有处理器的资源。

进程中的每个线程都独立执行，不会影响该进程中的其他线程。所有线程共享公共的资源。因此必须采用信号量或者其他进程间通信方法来调整线程的工作。

## 9.2 两种重要的线程

Windows 提供了两种线程，用户界面线程和工作者线程。用户界面线程通常用来处理消息循环、与用户交互，工作者线程用来处理后台的计算。下面分别介绍这两种线程。

### 9.2.1 用户界面线程

每一个 Windows 应用程序都有一个主线程。这里的用户界面线程是指用来和用户进行交互的线程。接收用户传送的数据，并且做出响应。用户界面线程通常包含自己的窗口，有自己的消息循环，独立于应用程序的其他部分。

创建一个用户界面线程需要首先继承线程类 CWinThread，重载它的成员函数，如表 9-1 所示。最后调用 AfxBeginThread 创建线程对象。

表 9-1 需要重载的 CWinThread 的成员函数

|  |  |
| --- | --- |
| 函数名称 | 作用 |
| InitInstance | 线程的初始化，通常需要重载 |
| ExitInstance | 释放线程占用的资源，通常需要重载 |
| OnIdle | 空闲时间的处理，不一定重载 |
| PreTranslateMessage | 过滤消息，不一定重载 |
| ProcessWndProcException | 处理线程抛出的意外 |
| Run | 线程控制函数，通常不重载 |

### 9.2.2 工作者线程

工作者线程通常用来处理后台运行的任务。在后台任务运行的同时，用户可进行其他操作，不必等待后台任务的结束。例如一个三维模型编辑软件，用户要对两个模型做布尔运算。在进行计算的同时，用户希望可以观察两个模型，对模型进行旋转，缩放的操作。再如文本浏览软件的打印功能，在打印文本的工程中，用户仍然会继续浏览文本内容。这些都属于工作者线程。

创建一个工作者线程只需要两个步骤。首先实现工作者线程的功能函数，然后启动线程即可。可以调用 Win32 提供的 API 函数 CreateThread 创建一个线程，MFC 对 Win32 的线程操作做了封装，也可以通过调用 AfxBeginThread 创建一个线程对象。这些函数及其调用方法将在下一节详细介绍。

## 9.3 线程的操作

本小节介绍 Windows 线程的操作方法，包括线程的创建、线程的管理、线程的同步、线程的终止等。

### 9.3.1 线程的创建

线程的创建方法有 3 种，分别介绍如下：

1．调用 Win32API 函数 CreateThread 和 CreateRemoteThread

（1）CreateThread

函数 CreateThread 用来创建线程。如果调用成功则返回非 0，否则返回 0。它的函数原型如下：

HANDLE CreateThread(

LPSECURITY\_ATTRIBUTES *lpThreadAttributes*,

SIZE\_T *dwStackSize*, LPTHREAD\_START\_ROUTINE *lpStartAddress*, LPVOID *lpParameter*, DWORD *dwCreationFlags*, LPDWORD *lpThreadId*

);

函数中主要参数的意义如下：

* lpThreadAttributes：指向SECURITY\_ATTRIBUTES结构的指针，决定返回的句柄是否可以被继承。
* dwStackSize：设定堆栈的大小。
* lpStartAddress：线程函数的名称，它的参数是第 4 个参数。
* lpParameter：线程函数的参数。
* dwCreationFlags：线程创建标志。如果是 CREATE\_SUSPENDED，则线程的初始状态为挂起，直到调用了 ResumeThread 才开始执行，如果为 0，线程创建后即开始执行。
* lpThreadId：存放线程标志的变量指针。

（2）CreateRemoteThread

函数 CreateRemoteThread 用来创建在另外一个进程的虚拟空间中执行的线程，它的函数

原型如下：

HANDLE CreateRemoteThread(

HANDLE *hProcess*,

LPSECURITY\_ATTRIBUTES *lpThreadAttributes*,

SIZE\_T *dwStackSize*,

LPTHREAD\_START\_ROUTINE *lpStartAddress*,

LPVOID *lpParameter*,

DWORD *dwCreationFlags*,

LPDWORD *lpThreadId*

);

函数中主要参数的意义如下。

* hProcess：进程的句柄。
* lpThreadAttributes：指向SECURITY\_ATTRIBUTES结构的指针，决定返回的句柄是否可以被继承。
* dwStackSize：设定堆栈的大小。
* lpStartAddress：线程函数的名称，它的参数是第四个参数。
* lpParameter：线程函数的参数。
* dwCreationFlags：线程创建标志。如果是 CREATE\_SUSPENDED，则线程的初始状态为挂起，直到调用了ResumeThread才开始执行。如果为 0，线程创建后即开始执行。
* lpThreadId：存放线程标志的变量指针。

2．C 运行库函数\_beginthreadex

\_beginthreadex 和 Win32API 函数 CreateThread 非常类似。如果调用成功，返回新创建线程的句柄，否则返回-1。它们的参数几乎完全相同。函数原型如下：

unsigned long \_beginthread(

void( \_\_cdecl \**start\_address* )( void \* ),

|  |
| --- |
| unsigned *stack\_size*, void \**arglist*  );  unsigned long \_beginthreadex( void \**security*, unsigned *stack\_size*,  unsigned ( \_\_stdcall \**start\_address* )( void \* ), void \**arglist*, unsigned *initflag*, unsigned \**thrdaddr*  ); |

函数中主要参数的意义如下。

* start\_address：线程的入口地址。
* stack\_size：初始堆栈大小。
* arglist：传给线程的参数。
* security：安全属性。
* initflag：线程创建标志。
* thrdaddr：存放线程标志的变量指针。

3．调用函数 AfxBeginThread

MFC 对 Win32 的线程操作做了封装，可以通过调用 AfxBeginThread 创建一个线程对象，函数 AfxBeginThread 有两种调用方式，函数原型如下：

|  |
| --- |
| CWinThread\* AfxBeginThread(  AFX\_THREADPROC *pfnThreadProc*,  LPVOID *pParam*, int *nPriority* = THREAD\_PRIORITY\_NORMAL,  UINT *nStackSize* = 0,  DWORD *dwCreateFlags* = 0,  LPSECURITY\_ATTRIBUTES *lpSecurityAttrs* = NULL  );  CWinThread\* AfxBeginThread( CRuntimeClass\* *pThreadClass*,  int *nPriority* = THREAD\_PRIORITY\_NORMAL,  UINT *nStackSize* = 0,  DWORD *dwCreateFlags* = 0,  LPSECURITY\_ATTRIBUTES *lpSecurityAttrs* = NULL  ); |

函数中主要参数的意义如下。

* pfnThreadProc：指向工作者线程的控制函数指针，该参数不能为空，这个函数必须按如下方式声明：

UINT MyControllingFunction( LPVOID pParam )

* PThreadClass：继承自 CWinThread 的RUNTIME\_CLASS对象。
* PParam：传给线程函数的参数。
* nPriority：指定线程优先级。
* nStackSize：初始堆栈大小。
* dwCreateFlags：线程创建标志。
* lpSecurityAttrs：安全属性。

如果线程创建失败，可以调用 GetLastError 得到错误信息，该函数没有参数，返回值是

调用线程的最后错误返回值。

### 9.3.2 线程的终止

1．调用 Win32API 函数 ExitThread

结束线程最直观的方法是调用 Win32API 函数 ExitThread，线程执行完线程函数的代码后，会隐式调用 ExitThread 函数，自动终止。当调用该函数时，当前线程的堆栈被释放，如果当前线程是进程的最后一个线程，则进程也终止。它的函数原型如下。

VOID ExitThread(

DWORD *dwExitCode*

);

函数中主要参数的意义如下。

* dwExitCode：线程的返回码。

2．Win32API 函数 TerminateThread

TerminateThread 用于强行终止一个线程。如果调用成功，返回 TRUE，否则返回 FALSE。它的函数原型如下：

BOOL TerminateThread(

HANDLE *hThread*,

DWORD *dwExitCode*

);

函数中主要参数的意义如下。

* hThread：要终止的线程句柄。
* dwExitCode：线程的终止码，可以调用函数 GetExitCodeThread 来得到。

函数 GetExitCodeThread 用于得到线程的退出码，如果调用成功，返回 TRUE，否则返回 FALSE。它的函数原型如下：

BOOL GetExitCodeThread(

HANDLE *hThread*,

LPDWORD *lpExitCode*

);

函数中主要参数的意义如下。

* hThread：要终止的线程句柄。
* LpExitCode：存放线程终止状态的变量指针。

3．C 运行库函数

C 运行库函数包括\_endthread 和\_endthreadex，分别用来结束一个由\_beginthread 或 \_beginthreadex 创建的线程。函数原型如下。需要注意的是，与 ExitThread 相同，\_endthreadex 并不关闭线程的句柄，因此，当调用这个函数之前，必须首先调用 Win32 API 函数

CloseHandle。

void \_endthread( void ); void \_endthreadex(

unsigned *retval*

);

函数中主要参数的意义如下。

* retval：线程返回码。

4．函数 AfxEndThread

函数 AfxEndThread 用来终止由 AfxBeginThread 创建的线程，它没有返回值，函数原型

如下：

void AFXAPI AfxEndThread(

UINT *nExitCode*,

BOOL *bDelete* = TRUE

);

函数中主要参数的意义如下。

* nExitCode：线程的返回码。
* bDelete：从内存中删除线程对象的标志。

### 9.3.3 线程的管理和操作

1．线程的挂起、继续和休眠

（1）挂起如果一个正在执行的线程用完了自己的时间片，则这个线程将会被挂起，并将 CPU 时间分给其他等待的线程。也可以通过调用 Win32 API 函数 SuspendThread 来挂起一个正在执行的线程，如果调用成功，返回线程上一次的挂起计数，否则返回-1。SuspendThread 的函数原型如下：

DWORD SuspendThread(

HANDLE *hThread*

);

函数中主要参数的意义如下。

• hThread：线程的句柄。

MFC 将线程操作进行了封装，CWinThread 的对象可以调用它的成员函数 SuspendThread 来挂起一个正在执行的线程。

1. 继续

如果继续执行 SuspendThread 挂起的线程，或者在执行时采用 CREATE\_SUSPENDED 标记创建的线程，可以调用函数 ResumeThread，将当前被挂起的线程的挂起计数减一。如果一个线程的挂起计数减为 0，则线程开始继续执行。如果调用成功，返回线程上一次的挂起计数，否则返回 0xFFFFFFFF。如果返回值为 0，则说明当前线程并没有挂起。如果返回 1，说明线程被挂起，现在开始继续执行。ResumeThread 的函数原型如下：

DWORD ResumeThread( );

1. 休眠

函数 Sleep 可以让当前执行的线程休眠一段时间，而函数 SleepEx 将线程挂起，等待下列事件之一发生。

* 一个 I/O 回调函数被调用。
* 线程需要调用一个异步的过程。
* 时间间隔到。

它们的函数原型如下：

VOID Sleep(

DWORD *dwMilliseconds*

);

函数中主要参数的意义如下。

* dwMilliseconds：休眠时间。

DWORD SleepEx(

DWORD *dwMilliseconds*, BOOL *bAlertable*

);

函数中主要参数的意义如下。

* dwMilliseconds：超时间隔。
* bAlertable：指定函数是否依赖 I/O 回调函数结束。

如果指定的时间间隔期满，则返回 0。如果函数返回依赖于一个或多个 I/O 回调函数则返回 WAIT\_IO\_COMPLETION，这种情况只有 bAlertable 为 TRUE 的时候才会发生。

2．线程的优先级

1. 级别

Windows 允许用户在线程执行的过程中动态地得到或者改变线程的优先级，使重要的线程可以优先执行，占有较多的时间片。该过程可以通过调用 Win32 API 函数 SetThreadPriority 来调整进程内的线程优先级，Win32 进程的优先级如表 9-2 所示，线程的优先级如表 9-3 所示。

表 9-2 进程优先级列表

|  |  |
| --- | --- |
| 优先级 | 优先级值 |
| ABOVE\_NORMAL\_PRIORITY\_CLASS | 大于 NORMAL\_PRIORITY\_CLASS，小于 HIGH\_PRIORITY\_CLASS |

续表

|  |  |
| --- | --- |
| 优先级 | 优先级值 |
| BELOW\_NORMAL\_PRIORITY\_CLASS | 大于 IDLE\_PRIORITY\_CLASS，小于 NORMAL\_PRIORITY\_CLASS. |
| HIGH\_PRIORITY\_CLASS | 13 |
| IDLE\_PRIORITY\_CLASS | 4 |
| NORMAL\_PRIORITY\_CLASS | 9（前台）或 7（后台） |
| REALTIME\_PRIORITY\_CLASS | 24 |

表 9-3 线程优先级列表

|  |  |
| --- | --- |
| 优先级 | 级别 |
| THREAD\_PRIORITY\_ABOVE\_NORMAL | 比进程优先级高一级 |
| THREAD\_PRIORITY\_BELOW\_NORMAL | 比进程优先级低一级 |
| THREAD\_PRIORITY\_HIGHEST | 比进程优先级高两级 |
| THREAD\_PRIORITY\_LOWEST | 比进程优先级低两级 |
| THREAD\_PRIORITY\_NORMAL | 与进程优先级相同 |
| THREAD\_PRIORITY\_TIME\_CRITICAL | 把线程优先级设为 15 |
| THREAD\_PRIORITY\_IDLE | 把线程优先级设为 1 |

1. 优先级的设置和获取

可以通过调用 Win32 API 函数 GetThreadPriority 或 SetThreadPriority 实现或修改线程优先级

的功能，如果调用成功，返回线程的优先级，否则返回 THREAD\_PRIORITY\_ERROR\_RETURN。

GetThreadPriority 的函数原型如下：

int GetThreadPriority(

HANDLE *hThread*

);

函数中主要参数的意义如下。

* hThread：线程的局柄。

函数 SetThreadPriority 用来设定线程的优先级，如果调用成功，返回非 0，否则返回 0。它的函数原型如下：

BOOL SetThreadPriority( HANDLE *hThread*,

int *nPriority*

);

函数中主要参数的意义如下。

* hThread：线程的局柄； • nPriority：要设定的优先级。

在本节最后给出的示例中，读者可以看到如何利用 SetThreadPriority 设定一个线程的优

先级。

3．线程 ID 的判断

在创建线程的同时，系统会为每个线程生成一个线程 ID。任何两个线程 ID 都不会相同。许多 Win32 API 函数都以线程 ID 作为参数对线程进行操作。在调用 CreateThread 或者 \_begintread 创建线程的时候，会返回线程 ID，可以在这个时候将 ID 保存下来。调用 Win32 API 函数 GetCurrentThreadID 可以得到当前线程 ID。该函数没有参数，它的函数原型如下：

DWORD GetCurrentThreadId(VOID);

用户也可以调用 Win32 API 函数 GetCurrentThread 得到当前线程的伪句柄，函数原型如

下：

HANDLE GetCurrentThread(VOID);

4．线程的切换

如果希望操作系统从当前正在执行的线程切换到其他就绪线程，可以通过调用 Win32

API 函数 SwitchToThread 实现，切换后的线程由操作系统选择。SwitchToThread 函数没有参数，它的函数原型如下：

BOOL SwitchToThread(VOID);

5．打开线程

Win32 API 函数 OpenThread 可以打开一个存在的线程对象。如果调用成功，返回指定线程的句柄，否则返回 NULL。它的函数原型如下：

HANDLE OpenThread(

DWORD *dwDesiredAccess*,

BOOL *bInheritHandle*,

DWORD *dwThreadId*

);

函数中主要参数的意义如下。

* dwDesiredAccess：描述希望对线程进行的访问。
* bInheritHandle：指定返回的句柄是否可以被一个当前进程创建的新进程继承。
* dwThreadId：被访问线程的 ID。

6．线程函数 ThreadProc

线程函数是一个由用户定义的函数，作为执行线程的入口。在调用创建线程函数的时候，

将 ThreadProc 作为参数传入。返回调用成功或失败的标记。它的函数原型如下：

DWORD WINAPI ThreadProc(

LPVOID *lpParameter*

);

函数中主要参数的意义如下。

* lpParameter：传给线程函数的参数。 7．获得线程的时间信息

线程的生命周期中，可以调用 GetThreadTimes 函数得到指定线程的时间信息。如果调用

成功，返回非 0 值，否则返回 0。GetThreadTimes 函数原型如下：

BOOL GetThreadTimes(

HANDLE *hThread*,

LPFILETIME *lpCreationTime*,

LPFILETIME *lpExitTime*,

LPFILETIME *lpKernelTime*,

LPFILETIME *lpUserTime*

);

函数中主要参数的意义如下。

* hThread：线程的句柄。
* lpCreationTime：保存线程的创建时间。
* lpExitTime：保存线程的生存时间。
* lpKernelTime：保存线程在内核模式下的生存时间。
* lpUserTime：保存线程在用户模式下的执行时间。

FILETIME是一个 64 位数的结构，表示从 1601 年 1 月 1 日到现在以十亿分之一为间隔的时间片数。它的定义如下：

typedef struct \_FILETIME {

DWORD *dwLowDateTime*;

DWORD *dwHighDateTime*;

} FILETIME, \*PFILETIME;

结构中主要成员变量的意义。

* dwLowDateTime：表示低 32 位字节。
* dwHighDateTime：表示高 32 位字节。

8．处理器相关操作

（1）SetThreadAffinityMask 函数

该函数用于为指定的线程设置其处理器相似性掩码。一个线程相似性掩码是一个位向量，每一位表示一个线程可以使用的处理器。线程相似性只对多处理器计算机有作用。对通常使用的单 CPU 机器无效。如果调用成功，返回非 0 值。否则返回 0。

SetThreadAffinityMask 函数原型入下：

DWORD\_PTR SetThreadAffinityMask ( HANDLE *hThread*,

DWORD\_PTR *dwThreadAffinityMask*

);

函数主要参数的意义。

* hThread：线程句柄。
* dwThreadAffinityMask：线程的相似性掩码。

（2）SetThreadIdealProcessor 函数

该函数用于为线程设定合适的处理器。当线程等待时间到，系统将在这个处理器上执行

指定的线程，如果调用成功，将返回上一次的处理器数目或者 MAXIMUM\_PROCESSORS，否则返回－1。它的函数原型如下：

DWORD SetThreadIdealProcessor(

HANDLE *hThread*,

DWORD *dwIdealProcessor*

);

函数中主要参数的意义如下。

* hThread：线程句柄。
* dwIdealProcessor：可以执行指定线程的处理器数目，如果 dwIdealProcessor 的值为

MAXIMUM\_PROCESSORS，说明该线程没有合适的处理器。

9．实例：赛马程序

实例 9-1：赛马程序实例。源代码在光盘中“ \09\实例 9-1\Racing” 目录下。

为了帮助读者理解上述线程的基本知识，为后续高级技术打好基础，本节的最后给出一个多线程编程实例，模拟赛马的过程。

（1）创建工程

首先利用 VC++6.0 的 AppWizard 创建基于对话框的工程 Racing，所有设置都采用默认选项。

给对话框添加控件。包括 1 个图片框、1 个组合框、3 个静态文本控件、3 个滑动杆控件、

4 个按钮。控件的属性如表 9-4 所示。

表 9-4 控件属性列表

|  |  |  |
| --- | --- | --- |
| ID | 属性或作用 | 控件类型 |
| IDC\_RACINGBOX | Frame | 图像框控件 |
| IDC\_STATIC | 速度设置 | 组合框控件 |
| IDC\_STATIC | 红马 | 静态文本控件 |
| IDC\_STATIC | 绿马 | 静态文本控件 |
| IDC\_STATIC | 蓝马 | 静态文本控件 |
| IDC\_SLIDERR | 红马速度 | 滑动杆控件 |
| IDC\_SLIDERG | 绿马速度 | 滑动杆控件 |
| IDC\_SLIDERB | 蓝马速度 | 滑动杆控件 |
| IDSTART | 开始 | 按钮 |
| IDSTOP | 停止 | 按钮 |
| IDPAUSE | 暂停 | 按钮 |
| IDCANCEL | 关闭 | 按钮 |

1. 定义数据结构

定义一个包含赛马信息的结构 HORSE，代码如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| { | typedef struct \_h  HWND | orse  hWnd; |  | // 窗口句柄 |
|  | int | speed; |  | // 马速度 |
|  | int | x; |  | // 马位置的 *x* 坐标 |
|  | int | y; |  | // 马位置的 *y* 坐标 |
|  | int | size; |  | // 马的大小, 用一个球来表示，size 是半径 |
|  | COLOR | color; |  | // 颜色 |
|  | CRect | rect; |  | // 表示跑道的矩形; |

}HORSE, \*LPHORSE;

1. 设置全局变量定义 Mutex 句柄和名称的全局变量，代码如下：

// Mutex 对象的名称

char g\_strRName[64] = "RedHorse"; char g\_strGName[64] = "GreenHorse"; char g\_strBName[64] = "BlueHorse";

// Mutex 对象的句柄

HANDLE g\_hRMutex;

HANDLE g\_hGMutex;

HANDLE g\_hBMutex;

1. 添加成员变量为 CRacingDlg 添加 3 个赛马信息的成员变量，包括红色、绿色、蓝色 3 匹马。添加 3

个线程句柄的成员变量，以及一个线程是否暂停的成员变量，代码如下：

// 定义红, 绿, 蓝 3 匹马

HORSE m\_RedHorse;

HORSE m\_GreenHorse;

HORSE m\_BlueHorse;

// 定义红, 绿, 蓝 3 匹马各自线程的句柄

HANDLE m\_hRed;

HANDLE m\_hGreen;

HANDLE m\_hBlue;

// 是否暂停状态

BOOL m\_bPaused;

为控件添加成员变量，代码如下：

// 设置红马的速度

CSliderCtrl m\_BSpeed;

// 设置绿马的速度

CSliderCtrl m\_GSpeed;

// 设置蓝马的速度

CSliderCtrl m\_RSpeed;

// 跑道矩形

CStatic m\_RacingBox;

（5）初始化信号量为 CRacingDlg 添加初始化函数 InitMutexes，用来创建 Mutex 对象，代码如下：

BOOL CRacingDlg::InitMutexes()

{

// 初始化红马的 Mutex

g\_hRMutex = CreateMutex(NULL, FALSE, g\_strRName); if(g\_hRMutex == NULL)

{

return FALSE;

}

// 初始化绿马的 Mutex

g\_hGMutex = CreateMutex(NULL, FALSE, g\_strGName); if(g\_hGMutex == NULL)

{

return FALSE;

}

// 初始化蓝马的 Mutex

g\_hBMutex = CreateMutex(NULL, FALSE, g\_strBName); if(g\_hBMutex == NULL)

{

return FALSE;

}

return TRUE;

}

（6）初始化控件为 CRacingDlg 添加初始化函数 InitDlgItem，用来初始化对话框的控件，代码如下：

void CRacingDlg::InitDlgItem()

{

// 设置滑动杆的范围和初始位置

m\_RSpeed.SetRange(10, 100); m\_RSpeed.SetPos(50); m\_GSpeed.SetRange(10, 100); m\_GSpeed.SetPos(50); m\_BSpeed.SetRange(10, 100); m\_BSpeed.SetPos(50);

// 暂停按钮的初始状态

m\_bPaused = FALSE;

|  |  |
| --- | --- |
| } | CButton\* pBtn;    // 开始按钮的状态为 TRUE  pBtn = (CButton\*)GetDlgItem(IDSTART); pBtn->EnableWindow(TRUE);  // 暂停按钮的状态为 TRUE  pBtn = (CButton\*)GetDlgItem(IDPAUSE); pBtn->EnableWindow(FALSE);    // 结束按钮的状态为 TRUE  pBtn = (CButton\*)GetDlgItem(IDSTOP); pBtn->EnableWindow(FALSE); |

（7）实现线程函数实现线程函数 ThreadProc，代码如下：

extern DWORD WINAPI ThreadProc(LPVOID lpParam)

{

LPHORSE lpHorse;

HANDLE hMutex;

lpHorse = LPHORSE(lpParam);

// 打开 Mutex 对象

if( lpHorse->color == RED )

{

hMutex = ::OpenMutex(MUTEX\_ALL\_ACCESS, FALSE, g\_strRName);

}

else if ( lpHorse->color == GREEN )

{

hMutex = ::OpenMutex(MUTEX\_ALL\_ACCESS, FALSE, g\_strGName);

}

else if ( lpHorse->color == BLUE )

{

hMutex = ::OpenMutex(MUTEX\_ALL\_ACCESS, FALSE, g\_strBName);

}

|  |  |  |
| --- | --- | --- |
|  | else    {    }  //  { | return 1; // 颜色不对  if( hMutex == NULL )  AfxMessageBox("open Mutex error..."); return 1;  循环，等待信号  while ( true )  if ( WaitForSingleObject(hMutex, INFINITE) == WAIT\_FAILED )  {  continue;  }  // 马前进  if ( lpHorse->x < lpHorse->rect.right )  {  lpHorse->x ++;  }  // 重绘窗口  InvalidateRect(lpHorse->hWnd, lpHorse->rect, TRUE);    // 释放 Mutex  ReleaseMutex(hMutex);  // 等待, 速度快的等待时间短  Sleep(lpHorse->speed); |
| } | }  return 0; | |

1. 初始化对话框改写对话框的 OnInitDialog，进行必要的初始化，代码如下：

BOOL CRacingDlg::OnInitDialog()

|  |  |
| --- | --- |
| {        } | ……  // TODO: Add extra initialization here  // 初始化 Mutex  InitMutexes();  // 初始化对话框  InitDlgItem();  …… |

1. 绘制场景

为 CRacingDlg 添加成员函数 DrawScene，用来绘制赛马的场景，代码如下：

|  |  |  |
| --- | --- | --- |
| //  { | 绘制场景的函数  void CRacingDlg::DrawScene(CPaintDC \* pDC)  CRect rect;  CBrush redBrush(RGB(255, 0, 0));  CBrush greenBrush(RGB(0, 255, 0));  CBrush blueBrush(RGB(0, 0, 255)); CBrush\* oldBrush;  m\_RacingBox.GetClientRect(rect); m\_RacingBox.ClientToScreen(rect);  pDC->Rectangle(rect);  // 绘制红马  if( m\_hRed )  {  pDC->Rectangle(m\_RedHorse.rect); oldBrush = pDC->SelectObject(&redBrush); pDC->Ellipse(m\_RedHorse.x-m\_RedHorse.size,  m\_RedHorse.x+m\_RedHorse.size, m\_RedHorse.y+m\_RedHorse.size);  pDC->SelectObject(oldBrush);  }    // 绘制绿马  if( m\_hGreen ) | m\_RedHorse.y-m\_RedHorse.size, |

|  |
| --- |
| {  pDC->Rectangle(m\_GreenHorse.rect);  oldBrush = pDC->SelectObject(&greenBrush);  pDC->Ellipse(m\_GreenHorse.x-m\_GreenHorse.size, m\_GreenHorse.y-m\_GreenHorse.size,  m\_GreenHorse.x+m\_GreenHorse.size, m\_GreenHorse.y+m\_GreenHorse.size); pDC->SelectObject(oldBrush);  }    // 绘制蓝马  if( m\_hBlue )  {  pDC->Rectangle(m\_BlueHorse.rect);  oldBrush = pDC->SelectObject(&blueBrush);  pDC->Ellipse(m\_BlueHorse.x-m\_BlueHorse.size, m\_BlueHorse.y-m\_BlueHorse.size,  m\_BlueHorse.x+m\_BlueHorse.size, m\_BlueHorse.y+m\_BlueHorse.size); pDC->SelectObject(oldBrush);  }  } |

改写 CRacingDlg 的 OnPaint 函数，调用 DrawScene，代码如下：

void CRacingDlg::OnPaint()

{ if (IsIconic())

{

……

}

else

{

CPaintDC dc(this); // device context for painting

CDialog::OnPaint();

// 绘制场景

DrawScene(&dc);

}

}

1. 响应“ 开始” 按钮响应“ 开始” 按钮，用来启动线程，代码如下：

void CRacingDlg::OnStart()

{

// TODO: Add your control notification handler code here

|  |
| --- |
| DWORD tID;  DWORD code;    CRect clientrect;  CRect racingrect;    GetClientRect(clientrect);  ClientToScreen(clientrect);    m\_RacingBox.GetClientRect(racingrect); m\_RacingBox.ClientToScreen(racingrect);    racingrect -= clientrect.TopLeft();    if( !GetExitCodeThread(m\_hRed, &code) ||  code != STILL\_ACTIVE )  {  // 填写红马信息  m\_RedHorse.hWnd = this->GetSafeHwnd(); m\_RedHorse.color = RED;  m\_RedHorse.speed = m\_RSpeed.GetPos(); m\_RedHorse.size = 10;  m\_RedHorse.rect.TopLeft() = racingrect.TopLeft();  m\_RedHorse.rect.right = m\_RedHorse.rect.TopLeft().x + racingrect.Width(); m\_RedHorse.rect.bottom = m\_RedHorse.rect.TopLeft().y + racingrect.Height()/3; m\_RedHorse.x = m\_RedHorse.rect.left + m\_RedHorse.size; m\_RedHorse.y = m\_RedHorse.rect.top + m\_RedHorse.rect.Height()/2;    // 创建红马线程  m\_hRed = CreateThread(NULL, 0, ThreadProc, &m\_RedHorse, 0, &tID);    if(m\_hRed == NULL)  {  AfxMessageBox("创建线程失败");  }  }    if( !GetExitCodeThread(m\_hGreen, &code) ||  code != STILL\_ACTIVE )  { |

|  |
| --- |
| // 填写绿马信息  m\_GreenHorse.hWnd = this->GetSafeHwnd(); m\_GreenHorse.color = GREEN; m\_GreenHorse.speed = m\_GSpeed.GetPos(); m\_GreenHorse.size = 10;  m\_GreenHorse.rect.TopLeft() = racingrect.TopLeft()+CPoint(0, racingrect.Height()/3); m\_GreenHorse.rect.right = m\_GreenHorse.rect.TopLeft().x + racingrect.Width(); m\_GreenHorse.rect.bottom = m\_GreenHorse.rect.TopLeft().y + racingrect.Height()/3; m\_GreenHorse.x = m\_GreenHorse.rect.left + m\_GreenHorse.size; m\_GreenHorse.y = m\_GreenHorse.rect.top + m\_GreenHorse.rect.Height()/2;    // 创建绿马线程  m\_hGreen = CreateThread(NULL, 0, ThreadProc, &m\_GreenHorse, 0, &tID);    if(m\_hGreen == NULL)  {  AfxMessageBox("创建线程失败");  }  }    if( !GetExitCodeThread(m\_hBlue, &code) ||  code != STILL\_ACTIVE )  {  // 填写蓝马信息  m\_BlueHorse.hWnd = this->GetSafeHwnd(); m\_BlueHorse.color = BLUE; m\_BlueHorse.speed = m\_BSpeed.GetPos(); m\_BlueHorse.size = 10;  m\_BlueHorse.rect.TopLeft() = racingrect.TopLeft()+CPoint(0, racingrect.Height()\*2/3); m\_BlueHorse.rect.right = m\_BlueHorse.rect.TopLeft().x + racingrect.Width(); m\_BlueHorse.rect.bottom = m\_BlueHorse.rect.TopLeft().y + racingrect.Height()/3; m\_BlueHorse.x = m\_BlueHorse.rect.left + m\_BlueHorse.size; m\_BlueHorse.y = m\_BlueHorse.rect.top + m\_BlueHorse.rect.Height()/2;    // 创建蓝马线程  m\_hBlue = CreateThread(NULL, 0, ThreadProc, &m\_BlueHorse, 0, &tID);    if(m\_hBlue == NULL)  {  AfxMessageBox("创建线程失败"); |

|  |  |
| --- | --- |
| } | }  }  // 设置按钮状态  CButton\* pBtn;  pBtn = (CButton\*)GetDlgItem(IDSTART); pBtn->EnableWindow(FALSE);  pBtn = (CButton\*)GetDlgItem(IDPAUSE); pBtn->EnableWindow(TRUE);  pBtn = (CButton\*)GetDlgItem(IDSTOP);  pBtn->EnableWindow(TRUE); |

1. 响应“ 暂停” 按钮响应“ 暂停” 按钮，用来暂停和继续线程，代码如下：

void CRacingDlg::OnPause()

{

// 得到 button 的指针

CButton\* pBtn = (CButton\*)GetDlgItem(IDPAUSE);

// 改变按钮的名称

if( !m\_bPaused )

{

SuspendThread(m\_hRed);

SuspendThread(m\_hGreen);

SuspendThread(m\_hBlue);

pBtn->SetWindowText("继续");

}

else

{

ResumeThread(m\_hRed);

ResumeThread(m\_hGreen);

ResumeThread(m\_hBlue);

|  |  |
| --- | --- |
| } | pBtn->SetWindowText("暂停");  }  // 改变按钮的状态  m\_bPaused = !m\_bPaused; |

（12）响应“ 停止” 按钮响应“ 停止” 按钮，用来停止线程，代码如下：

void CRacingDlg::OnStop()

{

// TODO: Add your control notification handler code here

// 存放线程的退出码

DWORD code;

if (!GetExitCodeThread(m\_hRed, &code) ||

code == STILL\_ACTIVE)

{

TerminateThread(m\_hRed, code);

CloseHandle(m\_hRed);

}

if (!GetExitCodeThread(m\_hGreen, &code) ||

code == STILL\_ACTIVE)

{

TerminateThread(m\_hGreen, code);

CloseHandle(m\_hGreen);

}

if (!GetExitCodeThread(m\_hBlue, &code) ||

code == STILL\_ACTIVE)

{

TerminateThread(m\_hBlue, code);

CloseHandle(m\_hBlue);

}

// 初始化对话框控件

InitDlgItem();

}

1. 响应“ 关闭” 按钮响应“ 关闭” 按钮，用来停止线程，并且释放资源，代码如下：

void CRacingDlg::OnCancel()

{

// TODO: Add extra cleanup here

OnStop();

// 释放 Mutex 资源

CloseHandle(g\_hRMutex);

CloseHandle(g\_hGMutex);

CloseHandle(g\_hBMutex);

CDialog::OnCancel();

}

1. 运行结果运行程序，单击“ 开始” 按钮，界面如图 9-1 所示。

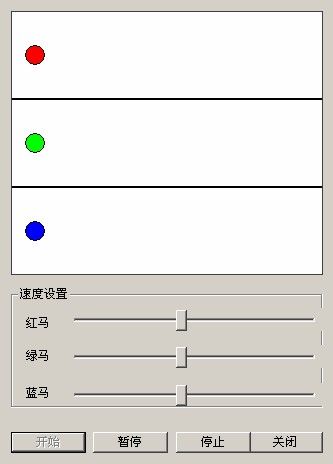
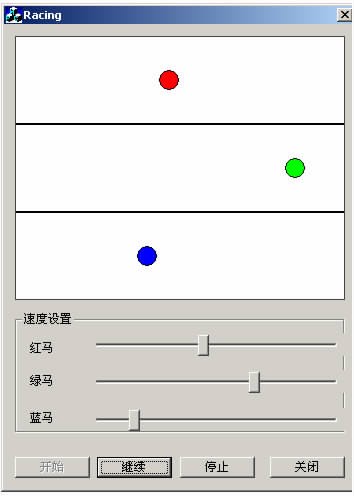


图 9-1 程序开始运行的界面

调整各个马的速度，并且单击“ 暂停” 按钮，界面如图 9-2 所示，这时“ 暂停” 按钮变成“ 继续” 按钮。

 图 9-2 暂停后的程序界面

单击“ 停止” 按钮，则重新开始比赛。单击“ 关闭” 按钮，退出程序。

### 9.3.4 线程间的通信

线程间的通信通常采用共享全局变量，共享存储区来实现。因为所有的线程都可以访问这些资源。但是需要注意线程的同步，将在下一节做详细介绍。主线程不能通过发送消息给辅助线程实现通信，但辅助线程可以通过发送自定义消息达到和主线程通信的目的。本节将通过一个简单的实例，介绍使用共享存储区和自定义消息实现线程间通信的功能。

实例 9-2：线程之间通信实例。源代码在光盘中“ \09\实例 9-2\IPCDemog” 目录下。

1. 创建工程

首先利用 VC++的 AppWizard 创建一个基于对话框的应用程序（请参考第四章相关内容）。所有设置都采用默认选项。

1. 添加资源为新建的对话框添加一个编辑框控件，用来输入字符。代码如下：

CEdit m\_input;

1. 添加变量给对话框加入内存映射文件句柄和视图的成员变量，代码如下：

HANDLE m\_hFileMapping; // 内存映射文件句柄

LPVOID m\_pViewOfFile; // 内存映射文件视图, 包含输入框的内容

BOOL m\_bNotify; // 只有 m\_bNotify 为 TRUE 时, OnChangeEditBox 才会工作

1. 定义消息注册自定义的消息 WM\_MyMessage，代码如下：

// 注册 windows 消息 const UINT WM\_MyMessage = RegisterWindowMessage(\_T("MY\_IPC\_MESSAGE"));

1. 变量初始化初始化内存映射文件的大小和名称，代码如下：

// 内存映射文件的大小

const DWORD dwMemoryFileSize = 4 \* 1024;

// 内存映射文件的名字

const LPCTSTR sMemoryFileName = \_T("MY\_IPC\_SHAREMEMORY");

1. 对话框初始化改写对话框的 OnInitDialog 函数，进行初始化，代码如下：

BOOL CIPCDemoDlg::OnInitDialog()

{

CDialog::OnInitDialog();

……

// TODO: Add extra initialization here

// 保证文本框中的文本的大小比内存文件小

m\_input.SetLimitText(dwMemoryFileSize - 1);

// 创建内存映射文件

m\_hFileMapping = CreateFileMapping(

INVALID\_HANDLE\_VALUE, // 系统页面大小

NULL, // 安全属性

PAGE\_READWRITE, // 保护方式 0, // 大小变量的高字节 dwMemoryFileSize\*sizeof(TCHAR), // 大小变量的低字节

sMemoryFileName // 文件名称

);

DWORD dwError = GetLastError();

if ( ! m\_hFileMapping )

{

MessageBox(\_T("Creating of file mapping failed"));

}

|  |
| --- |
| else  {  // 映射文件  m\_pViewOfFile = MapViewOfFile(  m\_hFileMapping, // 句柄  FILE\_MAP\_ALL\_ACCESS, // 访问属性  0, 0,  0); // 映射所有文件    if ( ! m\_pViewOfFile )  {  MessageBox(\_T("MapViewOfFile failed"));  }  }    if ( dwError == ERROR\_ALREADY\_EXISTS )  {  // 已经有程序运行  // 从内存文件中读取数据并且写入文本框    if ( m\_pViewOfFile )  {  //从内存文件中读取数据  TCHAR s[dwMemoryFileSize];    lstrcpy(s, (LPCTSTR) m\_pViewOfFile);    // 写入文本框  m\_bNotify = FALSE;    m\_input.SetWindowText(s);    m\_bNotify = TRUE;  }  }    return TRUE;  } |

（7）响应文本框输入响应文本框输入的消息 EN\_CHANGE，代码如下：

void CIPCDemoDlg::OnChangeEditInput()

{

// TODO: Add your control notification handler code here if ( m\_bNotify )

{

// 将字符写入映射内存文件

if ( m\_pViewOfFile )

{

TCHAR s[dwMemoryFileSize]; m\_input.GetWindowText(s, dwMemoryFileSize);

lstrcpy( (LPTSTR) m\_pViewOfFile, s);

// 给其他线程发消息, 文本框中的字符发生变化

::PostMessage(HWND\_BROADCAST,

WM\_MyMessage,

(WPARAM) m\_hWnd,

0);

}

}

}

（8）响应自定义消息响应自定义的消息 WM\_MyMessage，代码如下：

LRESULT CIPCDemoDlg::OnMyMessage(WPARAM wParam, LPARAM lParam)

{

// 如果是线程自己发送的消息, 则不作响应

if ( wParam == (WPARAM) m\_hWnd ) return 0;

// 从内存映射文件读取文本, 设置自己文本框的文本

if ( m\_pViewOfFile )

{

// 从内存映射文件读取文本

TCHAR s[dwMemoryFileSize];

lstrcpy(s, (LPCTSTR) m\_pViewOfFile);

|  |
| --- |
| // 将文本写入文本框  m\_bNotify = FALSE;    m\_input.SetWindowText(s);    m\_bNotify = TRUE;  } return 0;  } |

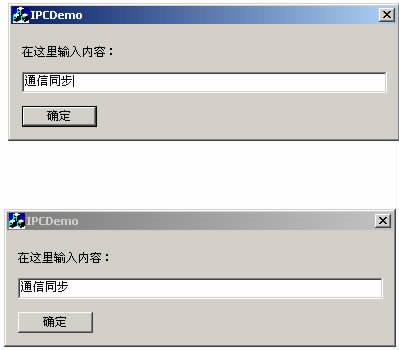
1. 释放资源

改写对话框退出消息 WM\_DESTROY 的响应函数 OnDestroy，释放资源，代码如下：

|  |  |  |  |
| --- | --- | --- | --- |
| void CIPCDemoDlg::OnDestroy()  {  CDialog::OnDestroy();  // TODO: Add your message handler code here if ( m\_hFileMapping )  {  if ( m\_pViewOfFile )  {  UnmapViewOfFile(m\_pViewOfFile);  }  // 释放内存映射资源  CloseHandle(m\_hFileMapping);  }  } |  |  | // 释放内存映射视图 |

1. 运行结果

运行多个程序，改变文本框的内容，界面如图 9-3 所示。

 图 9-3 程序运行界面

### 9.3.5 线程的同步

在多线程程序设计中，经常会出现两个或多个线程使用一个公共变量，或者多个线程共享一些公共存储区的情况。凡是涉及到共享资源的情况都有可能会引起程序的错误。为了解决这些问题，Windows 提供了大量线程的同步方法，例如变量锁、临界区、信号量、事件对象、互斥对象等。

1．互锁操作

一个或两个变量的互锁操作是最简单的同步原语。Win32 提供了 7 个具有线程安全性的原子操作，具体介绍如下。

（1）InterlockedIncrement

函数 InterlockedIncrement 为指定的 32 位变量加一，并且对结果进行检查。该函数不允许同一时间有大于一个的线程对变量进行访问。返回执行加一操作后的变量值。它的函数原型如下：

LONG InterlockedIncrement(

LPLONG volatile *lpAddend*

);

函数中主要参数的意义如下。

• LpAddend：指向变量的指针。

（2）InterlockedDecrement

函数 InterlockedDecrement 为指定的 32 位变量减一，并且对结果进行检查。该函数不允许同一时间有大于一个的线程对变量进行访问。返回执行减一操作后的变量值。它的函数原型如下：

LONG InterlockedDecrement(

LPLONG volatile *lpAddend*

);

函数中主要参数的意义如下。

• lpAddend：指向变量的指针。

（3）InterlockedExchange

函数 InterlockedExchange 自动交换一对变量值。该函数不允许同一时间有多于一个的线程对指定的变量进行访问。如果交换指针值，则调用函数 InterlockedExchangePointer。返回

Target 指向的初值。函数 InterlockedExchange 的函数原型如下：

LONG InterlockedExchange(

LPLONG volatile *Target*,

LONG *Value*

);

函数中主要参数的意义如下。

* Target：要交换的变量指针。
* Value：Target 指向变量的新值。

（4）InterlockedExchangeAdd

函数 InterlockedExchangeAdd 为某个 32 位变量增加指定的值。该函数不允许同一时间有

多于一个的线程访问同一变量。返回参数 Addend 指向的初值。它的函数原型如下：

LONG InterlockedExchangeAdd (

LPLONG volatile *Addend*,

LONG *Value*

);

函数中主要参数的意义如下。

* Addend：指向要进行加操作的变量指针。
* Value：要给参数 Addend 增加的值。

（5）InterlockedExchangePointer

函数 InterlockedExchangePointer 将某个 32 位变量的值改为一个新值。该函数不允许同一时间有多于一个的线程访问同一变量。返回参数 Target 指向的初值。它的函数原型如下：

PVOID InterlockedExchangePointer(

PVOID volatile \**Target*,

PVOID *Value*

);

函数中主要参数的意义如下。

* Target：指向要进行操作的变量指针。
* Value：数 Addend 的新值。

（6）InterlockedCompareExchange

函数 InterlockedCompareExchange 对指定的 32 位变量进行自动比较，根据比较结果决定是否进行交换。该函数不允许同一时间有多于一个的线程访问同一变量。返回参数 Destination 指向的初值。它的函数原型如下：

LONG InterlockedCompareExchange(

LPLONG volatile *Destination*,

LONG *Exchange*,

LONG *Comperand*

);

函数中主要参数的意义如下。

* Destination：目标值的地址。
* Exchange：指定交换值。
* Comperand：指定要和目标值比较的值。

（7）InterlockedCompareExchangePointer

函数 InterlockedCompareExchangePointer 对指定的 32 位变量进行自动比较，然后决定是否进行交换。该函数不允许同一时间有多于一个的线程访问同一变量。返回参数 Destination 指向的初值。它的函数原型如下：

PVOID InterlockedCompareExchangePointer (

PVOID volatile \**Destination*,

PVOID *Exchange*,

PVOID *Comperand*

);

函数中主要参数的意义如下。

* Destination：指向目标地址的指针。
* Exchange：指定交换值。
* Comperand：指定要和目标值比较的值。

互锁操作的使用方法非常简单，主线程和辅助线程对同一个全局变量进行操作，通过利用 InterlockedIncrement 函数达到同步的目的。

2．临界区（Critical Section）

临界区是一段程序代码，在任何时候都只能被一个线程使用。如果有多个线程同时访问临界区，这时只能有一个线程进入，其他线程则等待，直到临界区被释放。与其他同步方法不同的是，临界区只能在单个进程内使用。使用临界区的时候要避免长时间锁住一份资源。进入临界区后必须尽快地离开，释放资源。如果是主线程（GUI 线程）要进入一个没有被释放的临界区，将会出现错误。

（1）InitializeCriticalSection

在使用临界区之前，必须先进行初始化。可以调用 Win32 API 函数 InitializeCriticalSection 初始化一个临界区对象。它的函数原型如下：

VOID InitializeCriticalSection(

LPCRITICAL\_SECTION *lpCriticalSection*

);

函数中主要参数的意义如下。

• lpCriticalSection：指向临界区对象的指针。

（2）DeleteCriticalSection

相应的，当需要释放临界区资源时，可以调用 Win32 API 函数 DeleteCriticalSection 来实现，它的函数原型如下：

VOID DeleteCriticalSection(

LPCRITICAL\_SECTION *lpCriticalSection*

);

函数中主要参数的意义如下：

• lpCriticalSection：指向临界区对象的指针。

（3）EnterCriticalSection

Win32 API 函数 EnterCriticalSection 等待直到得到临界区对象的使用权，当调用线程得到临界区对象的使用权时，函数返回。它的函数原型如下：

VOID EnterCriticalSection(

LPCRITICAL\_SECTION *lpCriticalSection*

);

函数中主要参数的意义如下。

• lpCriticalSection：指向临界区对象的指针。

（4）LeaveCriticalSection

Win32 API 函数 LeaveCriticalSection 用来释放临界区的所有权，它的函数原型如下：

VOID LeaveCriticalSection(

LPCRITICAL\_SECTION *lpCriticalSection*

);

函数中主要参数的意义如下。

• lpCriticalSection：指向临界区对象的指针。

当速度要求较高，并且进程边界的资源不会被交叉使用的时候，通常采用 Critical Section 替代 Mutexes。临界区不是一个核心对象，无法获知进入临界区的线程的状态，如果进入临界区的线程处于死锁状态，没有释放临界资源，系统无法获知，而且没有办法释放该临界资源。这个缺点在互斥器（Mutex）中得到了弥补。

（5）CCriticalSection

CcriticalSection 是临界区在 MFC 中的相应的类。它的成员函数如表 9-5 所示。

表 9-5 CEvent 类的成员函数

|  |  |  |
| --- | --- | --- |
|  | 函数名称 | 作用 |
| CCriticalSection |  | 构造函数，构造 CCriticalSection 对象 |
| Lock |  | 进入临界区 |
| UnLock |  | 离开临界区 |

3．事件（Event）

事件（Event）是由 Windows 操作系统管理的同步对象。可以用于进程或线程的同步。一

个事件被创建后，只有激发状态和未激发状态两种状态，也称为发信号状态和未发信号状态。

事件包括手动重置事件和自动重置事件两种类型。手动重置事件被设置为激发状态后，会唤醒所有等待的线程，而且一直保持激发状态，直到程序重新把它设置为未激发状态。自动重置事件被设置为激发状态后，会唤醒一个等待中的线程，然后自动恢复为未激发状态。所以用自动重置事件来同步两个线程比较理想。

（1）CreateEvent

通过调用 Win32API 函数 CreateEvent 来创建或者打开一个事件对象。如果调用成功，返

回事件对象的句柄，否则返回空。它的函数原型如下：

HANDLE CreateEvent(

LPSECURITY\_ATTRIBUTES *lpEventAttributes*,

BOOL *bManualReset*,

BOOL *bInitialState*,

LPCTSTR *lpName*

);

函数中主要参数的意义如下。

* lpEventAttributes：指向 SECURITY\_ATTRIBUTES 结构的指针，它决定返回的句柄是否可以被子进程继承。如果 lpEventAttributes 为空，则不能被继承。
* bManualReset：指定创建一个手动重置事件或自动重置事件。如果为前者，必须调用 ResetEvent 手动设定状态为非激活状态。否则，当一个等待线程被释放后，系统会自动设定对象的状态为非激活状态。
* bInitialState：指定对象的初始状态。
* lpName：事件对象的名称。

（2）OpenEvent

函数 OpenEvent 可以打开一个存在的事件对象，允许多个进程打开同一个对象。如果调

用成功，返回事件对象的句柄。否则返回空。它的函数原型如下：

HANDLE OpenEvent(

DWORD *dwDesiredAccess*,

BOOL *bInheritHandle*,

LPCTSTR *lpName*

);

函数中主要参数的意义如下。

* dwDesiredAccess：指定如何操作对象。
* bInheritHandle：指定返回的句柄是否可以被继承。
* lpName：事件对象的名称。

（3）SetEvent，ResetEvent 和 PulseEvent

如果需要修改事件句柄的状态，可以使用 SetEvent、ResetEvent 和 PulseEvent。其中， SetEvent 将事件句柄的状态改为激活状态，如果自动重置事件处于激活状态，在满足了等待线程后，它可以被重置。手动重置事件必须调用 ResetEvent 函数将事件对象重置为激活状态。 PulseEvent 函数用来向事件句柄发信号，并在所有的等待线程都被释放后，将事件句柄重置为未激活状态。它们的返回值相同，如果调用成功，返回非 0 值，否则返回 0。它们的函数原型如下：

BOOL SetEvent(

HANDLE *hEvent*

);

BOOL ResetEvent(

HANDLE *hEvent*

);

BOOL PulseEvent(

HANDLE *hEvent*

);

函数中主要参数的意义如下。

* hEvent：事件对象的句柄。如果不再使用句柄，可以调用 Win32API 函数 CloseHandle 关闭句柄。函数原型如下：

BOOL CloseHandle(

HANDLE *hObject*

);

函数中主要参数的意义如下。

* hObject：对象的句柄。

（4）CEvent

事件（Event）在 MFC 中的相应的类是 CEvent。CEvent 的构造函数默认创建一个自动重置的事件，而且处于未激发状态。它的成员函数及其作用如表 9-6 所示。

表 9-6 CEvent 类的成员函数

|  |  |  |
| --- | --- | --- |
|  | 函数名称 | 作用 |
| CEvent |  | 构造函数，构造 CEvent 对象 |
| SetEvent |  | 启动事件对象，释放等待线程 |
| PulseEvent |  | 启动事件对象，释放等待线程，或者重置事件对象为未激活状态 |
| ResetEvent |  | 设置事件对象为未激活状态 |
| Unlock |  | 释放事件对象 |

4．互斥器（Mutex）

互斥器的功能与临界区相似。区别在于互斥器所花费的时间比临界区多很多，但是互斥器是核心对象（后面介绍的 Event 和 Semaphore 也是核心对象），可以跨进程使用，而且等待一个被锁住的互斥器可以设定 TIMEOUT，不会像临界区那样无法得知临界区的情况，一直等待。

Win32 提供了创建互斥器 CreateMutex()，打开互斥器 OpenMutex()，释放互斥器

ReleaseMutex()等操作。

Mutex 的拥有权并非属于产生它的那个线程，而是属于最后对此 Mutex 进行等待操作

（WaitForSingleObject）并且尚未进行 ReleaseMutex()操作的线程。线程拥有 Mutex 就好像进入临界区一样，一次只能有一个线程拥有该 Mutex。如果一个拥有 Mutex 的线程在返回之前没有调用 ReleaseMutex()，那么这个 Mutex 就被舍弃了。当其他线程等待这个 Mutex 时，仍能返回，并得到一个 WAIT\_ABANDONED\_0 返回值，一个 Mutex 被舍弃是 Mutex 特有的功能。下面详细介绍 Mutex 相关的函数。

（1）CreateMutex

函数 CreateMutex 用来创建或打开有名或无名的 Mutex 对象。如果调用成功，函数返回创建后对象的句柄。如果同名 Mutex 对象在创建之前就已经存在，则函数返回现有的对象的句柄，并且 GetLastError 函数返回 ERROR\_ALREADY\_EXISTS。否则返回创建的 Mutex 对象句柄。如果调用失败，返回空值。它的函数原型如下：

HANDLE CreateMutex(

LPSECURITY\_ATTRIBUTES *lpMutexAttributes*,

BOOL *bInitialOwner*,

LPCTSTR *lpName*

);

函数中主要参数的意义如下。

* lpMutexAttributes：指向 SECURITY\_ATTRIBUTES 结构的指针，决定返回的句柄是否可以被子进程继承。如果为空，则不能被继承。
* bInitialOwner：指定 Mutex 对象的初始拥有者。如果调用者创建 Mutex 并且 bInitialOwner 为 TRUE，调用线程获得 Mutex 对象的拥有权。否则，调用线程不拥有 Mutex 的所有权。
* lpName：Mutex 对象的名称。字符串的最大长度为 MAX\_PATH 个字符。字符串区分大小写。如果 lpName 和一个现有的 Mutex 对象重名，函数需要对已有的对象有

MUTEX\_ALL\_ACCESS 的访问权。这种情况下，参数 bInitialOwner 被忽略，因为此时它已经被已有的进程初始化。如果 lpMutexAttributes 非空，它决定句柄是否可以被继承，但是它的安全描述成员被忽略。如果 lpName 为空，创建的 Mutex 对象没有名称。

（2）OpenMutex

函数 OpenMutex 用来打开一个存在的有名称的 Mutex 对象。如果调用成功，返回打开的

对象的句柄。反之则返回空值。它的函数原型如下：

HANDLE OpenMutex(

DWORD *dwDesiredAccess*,

BOOL *bInheritHandle*,

LPCTSTR *lpName*

);

函数主要参数的意义。

* dwDesiredAccess：指定希望对 Mutex 对象进行的访问类型。可以是下面值的组合： · MUTEX\_ALL\_ACCESS：可以对 Mutex 对象进行所有访问。

· SYNCHRONIZE：允许任何等待函数获得 Mutex 对象的所有权。

* bInheritHandle：指定返回的句柄是否可以被继承。
* lpName：要打开的 Mutex 对象的名称。

（3）ReleaseMutex

函数 ReleaseMutex 释放指定的 Mutex 对象的所有权。如果调用线程没有 Mutex 对象的所

有权，则说明调用失败。它的函数原型如下：

BOOL ReleaseMutex(

HANDLE *hMutex*

);

函数中主要参数的意义如下。

• hMutex：Mutex 对象的句柄。

（4）CMutex

Mutex 在 MFC 中相应的类是 CMutex。CMutex 通过构造函数创建一个互斥器对象。当某个资源在一个时间只允许一个线程访问的时候，互斥器正好派上用场。

CMutex 只有一个构造函数 CMutex::CMutex，它的函数原型如下：

CMutex(

BOOL *bInitiallyOwn* = FALSE,

LPCTSTR *lpszName* = NULL,

LPSECURITY\_ATTRIBUTES *lpsaAttribute* = NULL

);

函数中主要参数的意义如下。

* bInitiallyOwn：指定创建线程是否在 CMutex 对象初始创建时访问互斥器控制的资源。
* lpszName：CMutex 对象的名称。如果有同名的互斥器存在，如果为空，互斥器就没有名称。如果和已有的互斥对象同名，构造函数创建一个以存在名称为参考的新的 CMutex 对象。如果名称和已有的其他同步对象同名，构造将失败。
* lpsaAttribute：互斥对象的安全属性。

5．信号量（Semaphore）信号量是最具历史的同步机制。信号量是解决 producer/consumer 问题的关键要素。对应的 MFC 类是 CSemaphore。Win32 函数 CreateSemaphore()用来产生信号量。ReleaseSemaphore() 用来解除锁定。Semaphore 的现值代表的意义是目前可用的资源数，如果 Semaphore 的现值为 1，表示还有一个锁定动作可以成功。如果现值为 5，就表示还有 5 个锁定动作可以成功。当调用 Wait 等函数要求锁定，如果 Semaphore 现值不为 0，Wait 马上返回，资源数减 1。当调用 ReleaseSemaphore()资源数加 1，当时不会超过初始设定的资源总数。

（1）CreateSemaphore

函数 CreateSemaphore 用来创建或者打开 Semaphore 对象。如果调用成功，函数返回创建后的对象句柄。如果同名 Semaphore 对象在创建之前就已经存在，则函数返回现有的对象的句柄，并且 GetLastError 函数返回 ERROR\_ALREADY\_EXISTS。否则返回创建的 Semaphore 对象句柄。如果调用失败，返回空值。它的函数原型如下：

HANDLE CreateSemaphore(

LPSECURITY\_ATTRIBUTES *lpSemaphoreAttributes*,

LONG *lInitialCount*,

LONG *lMaximumCount*,

LPCTSTR *lpName*

);

函数中主要参数的意义如下。

* lpSemaphoreAttributes：指定 SECURITY\_ATTRIBUTES 结构的指针。决定返回的句柄是否可以被子进程继承。如果 lpSemaphoreAttributes 为空，则句柄不能被继承。
* lInitialCount：指定 Semaphore 对象的初始计数值。该值必须大于或等于零并小于或等于 lMaximumCount。
* lMaximumCount：指定 Semaphore 对象的最大数目。必须大于零。
* lpName：Semaphore 对象的名称。字符串的最大长度为 MAX\_PATH 个字符。字符串区分大小写。

（2）OpenSemaphore

函数 OpenSemaphore 用来打开一个存在的有名称的 Semaphore 对象。如果调用成功，返

回打开的对象的句柄。反之则返回空值。它的函数原型如下：

HANDLE OpenSemaphore(

DWORD *dwDesiredAccess*,

BOOL *bInheritHandle*,

LPCTSTR *lpName*

);

函数中主要参数的意义如下。

* dwDesiredAccess：指定想要对 Semaphore 对象所作的访问。可以是下面值的任意组合： · SEMAPHORE\_ALL\_ACCESS：可以对对象做任何访问。

· SEMAPHORE\_MODIFY\_STATE：允许在 ReleaseSemaphore 中修改对象的数目。 · SYNCHRONIZE：允许任何等待函数使用 Semaphore 对象。

* bInheritHandle：指定返回的句柄是否可以被继承。
* lpName：要打开对象的句柄。

（3）ReleaseSemaphore

函数 ReleaseSemaphore 给指定的 Semaphore 对象增加指定的数量。如果调用成功，返回非零值。反之，返回零。它的函数原型如下：

BOOL ReleaseSemaphore(

HANDLE *hSemaphore*,

LONG *lReleaseCount*,

LPLONG *lpPreviousCount*

);

函数中主要参数的意义如下：

* hSemaphore：Semaphore 对象的句柄。
* lReleaseCount：指定要增加的值。必须大于零。如果指定的值会引起 Semaphore 的数目超过最大值，则函数返回 FALSE。
* lpPreviousCount：存储 Semaphore 对象增加之前的数目。如果不需要，则该参数可以为空。

（4）CSemaphore

Semaphore 在 MFC 中的相应的类是 CSemaphore。CSemaphore 通过构造函数创建一个信号量对象。CSemaphore 只有一个构造函数 CSemaphore:: CSemaphore，它的函数原型如下：

CSemaphore::CSemaphore(

LONG *lInitialCount* = 1,

LONG *lMaxCount* = 1,

LPCTSTR *pstrName* = NULL,

LPSECURITY\_ATTRIBUTES *lpsaAttributes* = NULL

);

函数中主要参数的意义如下。

* lInitialCount：指定 Semaphore 对象初始的使用数目。必须大于等于零，小于等于 lMaxCount。
* lMaxCount：指定 Semaphore 对象最大的使用数目
* pstrName：指定 Semaphore 对象的名称。
* lpsaAttributes：指定 Semaphore 对象的安全属性。

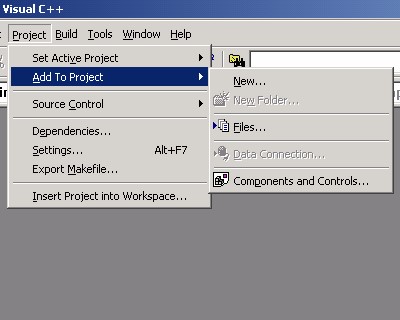
### 9.3.6 示例：串口通信

本节结合前面介绍的多线程计数，介绍 Windows 串口通信技术，通过建立多个线程，利用串口进行数据的发送。通过使用 Mutex 控制串口的发送，使得同一时间只有一个线程利用串口发送数据。下面给出详细的过程。

实例 9-3：串口通信实例。源代码在光盘中“ \09\实例 9-3\SComm” 目录下。

1．创建工程

* 首先利用 VC++的 AppWizard 创建一个基于对话框的工程 SComm。所有设置都采用默认选项。
* 为对话框加入串口控件。VC++的控件工具栏并没有包含串口控件，一次首先要将它加入工具栏中。选择“ Project” ->“ Add To Project” ->“ Components and Controls”，如图 9-4 所示。

 图 9-4 添加控件

* 在文件打开对话框中选择“ Microsoft Communications Control, version 6.0”，如图 9-5 所示，单击“ Close” 按钮关闭对话框。
* 为 VC++插入控件后，控件工具栏上多出了一个图标 ，如图 9-6 所示。



图 9-5 插入 MSComm 控件 图 9-6 插入控件后的控件工具栏

* 为对话框添加控件，界面设计如图 9-7 所示，控件的属性如表 9-7 所示。
* 为程序添加选择串口对话框资源，添加一个 ComboBox 串口让用户选择使用，创建

CCommSettingDlg 类。



图 9-7 对话框界面设计

表 9-7 对话框控件列表

|  |  |  |
| --- | --- | --- |
| ID | 属性或作用 | 控件类型 |
| IDC\_STATIC | 接收数据 | 组合框 |
| IDC\_STATIC | 发送数据 | 组合框 |
| IDC\_SENDDATA | 发送的数据 | 编辑框控件 |
| IDC\_IDC\_RECEIVEDATA | 接收的数据 | 编辑框控件 |
| IDC\_SEND | 发送 | 按钮控件 |
| IDC\_OPENCOM | 打开串口 | 按钮控件 |
| IDC\_CLOSE | 关闭串口 | 按钮控件 |
| IDCANCEL | 退出 | 按钮控件 |
| IDC\_MSCOMM1 | 实现串口的封装 | 串口控件 |

2．编写代码

1. 定义变量添加同步对象 Mutex 的句柄和名称的全局变量，代码如下：

// 同步 Mutex 对象

HANDLE g\_hMutex;

CString g\_strName = \_T("Serial\_comm\_Mutex\_name");

定义发送数据的格式，代码如下：

|  |  |
| --- | --- |
| typedef struct \_data  {  HWND | hWnd; |

|  |  |
| --- | --- |
| CMSComm \* | pComm; |
| char | data[BUFFERLEN]; |
| int  }DATA, \*LPDATA; | length; |

1. 实现发送线程

实现发送线程函数 CommSendProc，代码如下：

|  |
| --- |
| DWORD WINAPI CommSendProc(LPVOID LPPARAM)  {  LPDATA lpData;  HANDLE hMutex;    lpData = (LPDATA)LPPARAM;    // 打开互斥量  hMutex = ::OpenMutex(MUTEX\_ALL\_ACCESS, FALSE, g\_strName); if( hMutex == NULL )  {  AfxMessageBox("open Mutex error...");  return 1;  }    // 失败  if ( WaitForSingleObject(hMutex, INFINITE) == WAIT\_FAILED )  {  return 1;  }    CByteArray m\_Array;    // 读出发送数据  int Count=lpData->length; m\_Array.RemoveAll(); m\_Array.SetSize(Count);    for(int i=0;i<Count;i++)  {  m\_Array.SetAt(i,lpData->data[i]);  } |

|  |  |
| --- | --- |
| } | // 使用串口发送数据  lpData->pComm->SetOutput(COleVariant(m\_Array));    // 释放互斥量  ReleaseMutex(hMutex);  return 0; |

1. 初始化互斥量为对话框添加初始化互斥量的成员函数 InitMutex，代码如下：

bool CSCommDlg::InitMutex()

{

g\_hMutex = CreateMutex(NULL, false, g\_strName); if( g\_hMutex == NULL )

{

AfxMessageBox("创建互斥对象错误");

return false;

}

return true;

}

1. 初始化串口为对话框添加初始化串口的成员函数 InitComm，代码如下：

bool CSCommDlg::InitComm()

{

CCommSettingDlg dlg;

if( dlg.DoModal() == IDOK )

{

m\_nIndex = atoi(LPCTSTR(dlg.m\_nCommID) );

}

//通信参数设置

// 设置串口号

m\_Comm.SetCommPort(m\_nIndex);

// 设置数据获取方式

m\_Comm.SetInputMode(1);

|  |
| --- |
| // 设置传输参数  m\_Comm.SetSettings("9600,n,8,1"); m\_Comm.SetRThreshold(1);    // 指定接收缓冲区大小  m\_Comm.SetInBufferSize(1024);    // 清空接收缓冲区  m\_Comm.SetInBufferCount(0);    // 设置读取方式  m\_Comm.SetInputLen(0);    // 打开串口  if(!m\_Comm.GetPortOpen())  {  m\_Comm.SetPortOpen(TRUE);  }    // 读取数据  m\_Comm.GetInput();    return true;  } |

1. 响应 OnComm 消息利用 ClassWizard 响应 MSComm 控件的 OnComm 消息，用来处理接收数据，代码如下：

|  |
| --- |
| void CSCommDlg::OnComm()  {  // TODO: Add your control notification handler code here int nEvent;  VARIANT m\_input;  char \*str,\*str1; int k,i;  CString str2;    nEvent=m\_Comm.GetCommEvent();    switch(nEvent) |

|  |
| --- |
| {  case 2:    // 接收缓冲区的字符数目  k=m\_Comm.GetInBufferCount();  if(k>0)  {  m\_input=m\_Comm.GetInput();  str=(char\*)(unsigned char\*)m\_input.parray->pvData;  }    i=0;  str1=str;  while(i<k)  {  i++; str1++;  }  \*str1=’\0’;  str2=(const char\*)str; //清除字符串中的不必要字符    m\_RevData = (LPCTSTR)str2; default:  break;  }  // 显示数据  UpdateData(FALSE);  } |

1. 发送数据响应“ 发送” 按钮的单击事件，用来发送数据，代码如下：

void CSCommDlg::OnSendData()

{

// TODO: Add your control notification handler code here

DWORD id;

UpdateData();

DATA senddata; senddata.hWnd = this->GetSafeHwnd();

|  |  |
| --- | --- |
| } | senddata.length = m\_SendData.GetLength(); strcpy(senddata.data, (LPCTSTR)m\_SendData); senddata.pComm = &m\_Comm;  // 创建发送线程  CreateThread(NULL, 0, CommSendProc, &senddata, 0, &id); |

1. 关闭串口响应“ 关闭串口” 按钮的单击事件，用来关闭串口，代码如下：

void CSCommDlg::OnCloseCom()

{

// TODO: Add your control notification handler code here if( m\_Comm.GetPortOpen() )

{

// 关闭串口

m\_Comm.SetPortOpen(0);

}

}

1. 打开串口响应“ 打开串口” 按钮的单击事件，用来打开串口，代码如下：

void CSCommDlg::OnOpenCom()

{

// TODO: Add your control notification handler code here if( !m\_Comm.GetPortOpen() )

{

// 打开串口

m\_Comm.SetPortOpen(1);

}

}

1. 退出响应“ 退出” 按钮的单击事件，用来释放资源，退出程序，代码如下：

void CSCommDlg::OnCancel()

{

// TODO: Add extra cleanup here

// 释放互斥量资源

ReleaseMutex(g\_hMutex);

// 关闭句柄

CloseHandle(g\_hMutex);

|  |  |
| --- | --- |
| } | CDialog::OnCancel(); |

1. 运行结果

运行程序后，首先弹出“ 选择串口参数” 对话框，选择需要的串口，如图 9-8 所示。

 图 9-8 “ 设置串口参数” 对话框

在程序主界面中，单击“ 发送” 按钮，发送输入框中的内容会出现在另一个程序的接收框中，如图 9-9 和图 9-10 所示。

为了更加简便，本节仅给出利用多线程串口发送数据的方法，读者若有兴趣可以自行实现多线程控制多个串口的功能。

图 9-9 发送窗口 图 9-10 接收窗口

## 9.4 本章小结

本章主要介绍 Windows 操作系统下多线程的基本概念，如何创建和管理线程，以及线程的同步问题。通过本章的介绍，读者可以看出，多线程程序设计通常比一般的单线程程序复杂，在程序设计过程中，一定要考虑清楚各线程的关系，避免出现死锁或不同步的现象。另外需要注意现在大多数用户使用的是单 CPU 计算机，在这种机器上运行多线程程序，有时反而会降低系统的性能。因此，在设计多线程应用程序时，应慎重选择，视具体情况加以处理，使应用程序获得最佳的性能。

# 第 10 章 动态链接库

动态链接库（Dynamic Link Library）是一个可执行模块，其包含的函数可以由 Windows 应用程序调用以执行一些功能，主要为应用程序模块提供服务。本章将全面、系统地阐述在

Visual C++平台下进行 Win32 动态链接库的设计和应用。主要包括下面几个方面的内容。

* 动态链接库（DLL）的基本知识。
* DLL 的出入口函数。
* 调用 DLL 中的两种方式。
* 开发 DLL 的方式。
* DLL 中资源的利用。
* 钩子（Hook）函数的应用方法。

为了使读者充分理解概念，对于动态链接库的开发，本章使用了“ 界面汉化” 的示例来说明资源在动态链接库中的使用。钩子函数对于大多数读者来说可能是一项较为陌生的技术，为了加深理解，这里列举了两个关于捕获消息的钩子函数示例，可以帮助读者更好地理解钩子函数的原理和使用方法。

## 10.1 动态链接库的基础知识

比较大的应用程序都是由很多模块组成的，这些模块彼此协作，以完成整个软件系统的工作。其中可能存在一些模块的功能较为通用，在构造其他软件系统时仍会被使用。在构造软件系统时，如果将所有模块的源代码都静态编译到整个应用程序 EXE 文件中，会产生一些问题。一是增加了应用程序的大小，这样会占用更多的磁盘空间，程序运行时也会消耗较大的内存空间，造成系统资源的浪费；另外，在编写大的 EXE 程序时，每次修改重建时都必须调整编译所有源代码，不但增加了编译过程的复杂性，也不利于阶段性的单元测试。

Windows 系统平台上提供了一种完全不同的有效编程和运行环境，可以将独立的程序模块创建为较小的动态链接库（Dynamic Linkable Library）文件，并可对它们单独进行编译和测试。在运行时，只有在 EXE 程序确实要调用这些 DLL 模块的情况下，系统才会将它们装载到内存空间中。这种方式不仅减少了 EXE 文件的大小和对内存空间的需求，而且使这些 DLL 模块可以同时被多个应用程序使用，从而充分利用资源。Microsoft Windows 将一些主要的系统功能以 DLL 模块的形式实现。例如 IE 中的一些基本功能就是由 DLL 文件实现的，它可以被其他应用程序调用和集成。一般来说，下面的这几种情况必须用到动态链接库技术。

* 多个应用程序共享代码和数据就是通过共享动态链接库实现的，比如 Office 软件的各个组成部分有相似的外观和功能。
* 在钩子程序过滤系统消息时必须使用动态链接库。
* 设备驱动程序必须是动态链接库。
* 如果要在对话框编辑器中使用自己定义的控件，也必须使用动态链接库。
* 动态链接库以一种自然的方式将一个大的应用程序划分为几个小的模块，有利于小组内部成员的分工与合作。而且，各个模块可以独立升级。如果小组中的一个成员开发了一组实用示例，他就可以把这些示例放在一个动态链接库中，让小组的其他成员使用。
* 为了实现应用程序的国际化，往往需要使用动态链接库。使用动态链接库可以将针对某一国家、语言的信息存放在其中。对于不同的版本，使用不同的动态链接库。在使用 AppWizard 生成应用程序时，可以指定资源文件使用的语言，这就是通过提供不同的动态链接库实现的。

一般来说，DLL 是一种磁盘文件（通常带有 DLL 扩展名），它由全局数据、服务函数和资源组成，在运行时被系统加载到进程的虚拟空间中，成为调用进程的一部分。如果与其他 DLL 之间没有冲突，该文件通常映射到进程虚拟空间的同一地址上。DLL 模块中包含各种导出函数，用于向外界提供服务。Windows 在加载 DLL 模块时将进程函数调用与 DLL 文件的导出函数相匹配。

在 Win32 环境中，每个进程都复制了自己的读写全局变量。如果想要与其他进程共享内存，必须使用内存映射文件或者声明一个共享数据段。DLL 模块需要的堆栈内存都是从运行进程的堆栈中分配出来的。DLL 现在越来越容易编写。Win32 已经大大简化了其编程模式，并有许多来自 AppWizard 和 MFC 类库的支持。使用 Visual C++ 6.0 工具可以编写 3 种不同类型的动态链接库。

* Non-MFC DLL：指的是不用 MFC 的类库结构，直接用 C 语言编写的 DLL，其输出的函数一般用的是标准 C 接口，并能被非 MFC 或 MFC 编写的应用程序所调用。
* Regular DLL：和下述的 Extension Dlls 一样，是用 MFC 类库编写的。其特点是在源文件里有一个继承 CWinApp 的类。其又可细分成静态连接到 MFC 和动态连接到 MFC 上的。但静态连接到 MFC 的动态链接库只被 Visual C++的专业版和企业版所支持。
* Extension DLL：用来实现从 MFC 所继承下来的类的重新利用，也就是说，用这种类型的动态链接库，可以用来输出一个从 MFC 所继承下来的类。Extension DLL 使用 MFC 的动态连接版本所创建的，并且它只被用 MFC 类库所编写的应用程序所调用。

## 10.2 DLL 的出入口函数

DllMain()函数是 DLL 模块的默认入口点。当 Windows 加载 DLL 模块时调用这一函数。系统首先调用全局对象的构造函数，然后调用全局函数 DllMain()。DllMain()函数不仅在将

DLL 链接加载到进程时被调用，在 DLL 模块与进程分离时（以及其他时候）也被调用。

DLL 文件中包含一个导出函数表。这些导出函数由它们的符号名和称为标识号的整数与外界联系起来。函数表中还包含了 DLL 中函数的地址。当应用程序加载 DLL 模块时，它并不知道调用函数的实际地址，但它知道函数的符号名和标识号。动态链接过程在加载 DLL 模块时动态建立一个函数调用与函数地址的对应表。如果重新编译和重建 DLL 文件，并不需要修改应用程序，除非改变了导出函数的符号名和参数序列。

简单的 DLL 文件只为应用程序提供导出函数，比较复杂的 DLL 文件除了提供导出函数以外，还调用其他 DLL 文件中的函数。这样，一个特殊的 DLL 既有导入函数，又有导出函数。这并不是一个问题，因为动态链接过程可以处理交叉相关的情况。在 DLL 代码中，声明导出函数的代码如下：

\_\_declspec(dllexport) int MyFunction(int n);

但也可以在模块定义(DEF)文件中列出导出函数，不过这样做常常引起更多的麻烦。在

应用程序方面，声明相应的输入函数，代码如下：

\_\_declspec(dllimport) int MyFuncition(int n);

仅有导入和导出声明并不能使应用程序内部的函数调用链接到相应的 DLL 文件上。应用程序的项目必须为链接程序指定所需的输入库（LIB 文件）。而且应用程序事实上必须至少包含一个对 DLL 函数的调用。本节将对 DLL 的 DllMain()入口函数和导出函数作相关的介绍。

### 10.2.1 DllMain()函数

每一个 DLL 必须有一个入口点，这就象用 C 编写的应用程序一样，必须有一个 WinMain 函数一样。在 Non-MFC DLL 中 DllMain()是一个默认的入口函数，不需要编写自己的 DLL 入口函数，用这个默认的入口函数就能使动态链接库在被调用时初始化。如果应用程序的 DLL 需要分配额外的内存或资源，即对每个进程或线程初始化和清除操作时，就需要在相应的 DLL 工程的 CPP 文件中对 DllMain()函数按照下面的格式书写，代码如下：

BOOL APIENTRY DllMain(HANDLE hModule,DWORD ul\_reason\_for\_call,LPVOID lpReserved)

{

switch( ul\_reason\_for\_call )

{

case DLL\_PROCESS\_ATTACH:

.......

case DLL\_THREAD\_ATTACH:

.......

case DLL\_THREAD\_DETACH:

.......

case DLL\_PROCESS\_DETACH:

.......

}

return TRUE;

}

|  |
| --- |
| 注意：函数名 DllMain 是区分大小写的。许多编程人员有时调用的函数是 DLLMain。这是一个非常容易犯的错误，因为 DLL 这个词常常使用大写来表示。如果调用的进入点函数不是 DllMain，而是别的函数，代码将能够编译和链接，但是其进入点函数永远不会被调用，  DLL 永远不会被初始化。 |

参数 hinstDll 包含了 DLL 的实例句柄。与(w)WinMain 函数的 hinstExe 参数一样，这个值用于标识 DLL 的文件映像被映射到进程的地址空间中的虚拟内存地址。通常将这个参数保存在一个全局变量中，这样就可以在调用加载资源函数（如 DialogBox 和 LoadString）时使用它。最后一个参数是 fImpLoad，如果 DLL 是隐含加载的，那么该参数将是个非 0 值，如果 DLL 是显式加载的，那么它的值是 0。

参 数 fdwReason 用 于 指 明 系 统 为 什 么 调 用 该 函 数 。 该 参 数 可 以 使 用 DLL\_PROCESS\_ATTACH （进程被调用）、 DLL\_THREAD\_ATTACH （线程被调用）、 DLL\_PROCESS\_DETACH（进程被停止）、DLL\_THREAD\_DETACH（线程被停止）4 个值的其中之一，lpReserved 为保留参数。下面就具体介绍这 4 个值的意义。

1．DLL\_PROCESS\_ATTACH 通知

当 DLL 被初次映射到进程的地址空间中时，系统将调用该 DLL 的 DllMain()函数，给它传递参数 fdwReason 的值 DLL\_PROCESS\_ATTACH。只有当 DLL 的文件映像初次被映射时，才会出现这种情况。如果线程在后来为已经映射到进程的地址空间中的 DLL 调用 LoadLibrary(Ex) 函数，那么操作系统只是递增 DLL 的使用计数，它不会再次用

DLL\_PROCESS\_ATTACH 的值来调用 DLL 的 DllMain()函数。

当处理 DLL\_PROCESS\_ATTACH 时，DLL 应该执行 DLL 中的函数要求的任何与进程相关的初始化。例如，DLL 可能包含需要使用它们自己的堆栈（在进程的地址空间中创建）的函数。通过在处理DLL\_PROCESS\_ATTACH通知时调用HeapCreate()函数，该DLL的DllMain() 函数就能够创建这个堆栈。已经创建的堆栈的句柄可以保存在 DLL 函数有权访问的一个全局变量中。

当 DllMain()函数处理一个 DLL\_PROCESS\_ATTACH 通知时，DllMain()的返回值能够指明 DLL 的初始化是否已经取得成功。如果对 HeapCreate()函数的调用取得了成功，DllMain() 应该返回 TRUE。如果堆栈不能创建，它应该返回 FALSE。如果 fdwReason 使用的是其他的值，即 DLL\_PROCESS\_DETACH、DLL\_HREAD\_ATTACH 和 DLL\_THREAD\_DETACH，那么系统将忽略 DllMain()返回的值。

当然，系统中的有些线程必须负责执行 DllMain()函数中的代码。当一个新线程创建时，系统将分配进程的地址空间，将 EXE 文件映像和所有需要的 DLL 文件映像映射到进程的地址空间中。然后开始创建进程的主线程，并使用该线程调用每个 DLL 的带有 DLL\_PROCES S\_ATTACH 值的 DllMain()函数。当已经映射的所有 DLL 都对通知信息作出响应后，系统将使进程的主线程开始执行可执行模块的 C/C++运行期启动代码，然后执行可执行模块的进入点函数(main、wmain、WinMain 或 wWinMain)。如果 DLL 的任何一个 DllMain()函数返回 FALSE，则表明初始化没有取得成功，系统便终止整个进程的运行，从它的地址空间中删除所有文件映像，给用户显示一个消息框，说明进程无法启动运行。

2．DLL\_PROCESS\_DETACH 通知

当DLL从进程的地址空间中被卸载时，系统将调用DLL的DllMain()函数，给它传递fdwR eason 的值 DLL\_PROCESS\_DETACH。当 DLL 处理这个值时，它可以执行任何与进程相关的清除操作。例如，DLL 可以调用 HeapDestroy()函数来撤消它在 DLL\_PROCESS\_DETACH 通知期间创建的堆栈。需要注意的是，如果 DllMain()函数接收到 DLL\_PROCESS\_DETACH 通知时返回 FALSE，那么 DllMain()就不是用 DLL\_PROCESS\_DETACH 通知调用的。如果因为进程终止运行而使 DLL 被卸载，那么调用 ExitProcess()函数的线程将负责执行 DllMain()函数的代码。在正常情况下，这是应用程序的主线程。当进入点函数返回到 C/C++运行期库的启动代码时，该启动代码将显式调用 ExitProcess()函数，终止进程的运行。

如果因为进程中的线程调用 FreeLibrary()或 FreeLibraryAndExitThread()函数而将 DLL 卸载，那么调用函数的线程将负责执行 DllMain()函数的代码。如果使用 FreeLibrary，那么要等到 DllMain()函数完成对 DLL\_PROCESS\_DETACH 通知的执行后，该线程才可以从对

FreeLibrary 函数的调用中返回。

3．DLL\_THREAD\_ATTACH 通知

当在一个进程中创建线程时，系统查看当前映射到该进程的地址空间中的所有 DLL 文件映像，并调用每个带有 DLL\_THREAD\_ATTACH 值的 DllMain()函数文件映像。这样，DLL 就可以执行每个线程的初始化操作。新创建的线程负责执行 DLL 的所有 DllMain()函数中的代码。只有当所有的 DLL 都有机会处理该通知时，系统才允许新线程开始执行它的线程函数。

当一个新 DLL 被映射到进程的地址空间时，如果该进程内已经有若干个线程正在运行，那么系统将不为现有的线程调用带有 DLL\_THREAD\_ATTACH 值的 DDL 的 DllMain()函数。只有当新线程创建时， DLL 被映射到进程的地址空间中，它才可以调用带有

DLL\_THREAD\_ATTACH 值的 DLL 的 DllMain()函数。

另外要注意，系统并不为进程的主线程调用带有DLL\_THREAD\_ATTACH 值的任何 DllMain() 函数。进程初次启动时映射到进程的地址空间中的任何 DLL 均接收 DLL\_PROCESS\_ATTACH 通知，而不是 DLL\_THREAD\_ATTACH 通知。

4．DLL\_THREAD\_DETACH 通知

让线程终止运行的首选方法是使它的线程函数返回。这使得系统可以调用 ExitThread() 函数来撤消该线程。如果 ExitThread()函数要终止运行该线程，系统不会立即将它撤消，而是取 出 这 个 即 将 被 撤 消 的 线 程 ， 并 让 它 调 用 已 经 映 射 的 DLL 中 所 有 带 有 DLL\_THREAD\_DETACH 值的 DllMain()函数。这个通知告诉所有的 DLL 执行每个线程的清

除操作。例如，DLL 版本的 C/C++运行期库能够释放它用于管理多线程应用程序的数据块。

注意：DLL 能够防止线程终止运行。例如，当 DllMain()函数接收到 DLL\_THREAD\_DETACH 通知时，它就能够进入一个无限循环。只有当每个 DLL 已经完成对 DLL\_THREAD\_DETACH 通知的处理时，操作系统才会终止线程的运行。

如果当 DLL 被撤消时仍然有线程在运行，那么就不会有任何线程调用带有 DLL\_THREAD\_DETACH 值的 DllMain()函数。可以在进行 DLL\_THREAD\_DETACH 的处理时查看这个情况，这样就能够执行必要的清除操作。

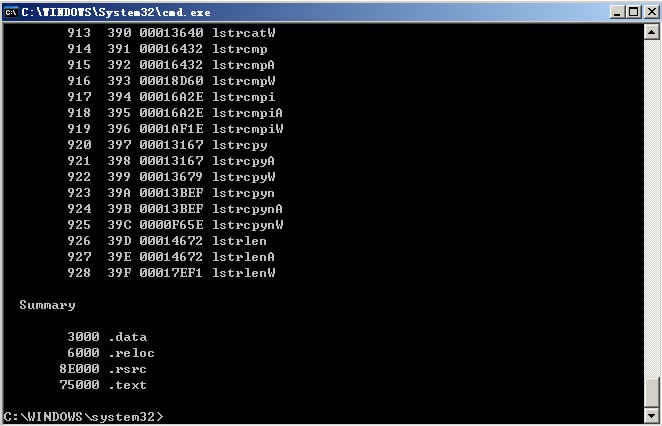
### 10.2.2 DLL 的导出函数

当 Microsoft 的 C/C++编译器看到变量、函数原型或 C++类之前的这个修改符的时候，它就将某些附加信息嵌入产生的.obj 文件中。当链接 DLL 的所有.obj 文件时，链接程序将对这些信息进行分析。

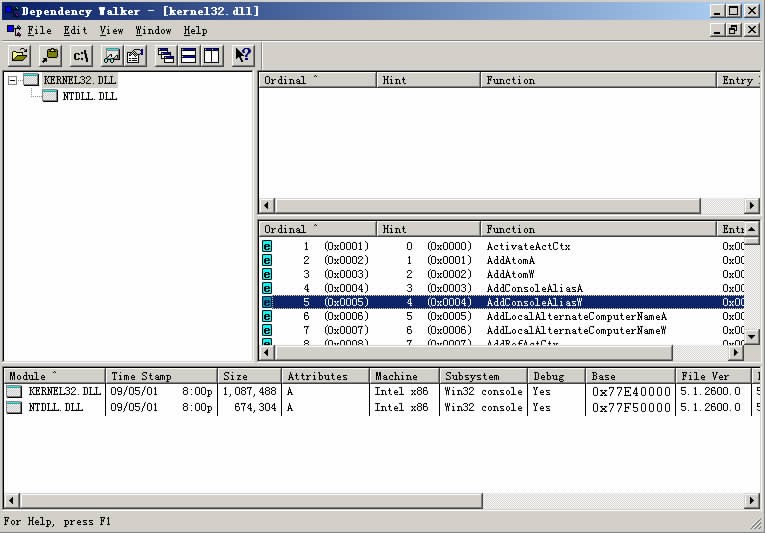
当 DLL 被链接时，链接程序要查找关于输出变量、函数或 C++类的信息，并自动生成一个.lib 文件。该.lib 文件包含一个 DLL 输出的符号列表。当然，如果要链接引用该 DLL 的输出符号的任何可执行模块，该.lib 文件是必不可少的。除了创建.lib 文件外，链接程序还要将一个输出符号表嵌入产生的 DLL 文件。这个输出节包含输出变量、函数和类符号的列表（按字母顺序排列）。该链接程序还将能够找到每个符号的相对虚拟地址（RVA），并在该地址中放入 DLL 模块。

使用 Microsoft 的 Visual Studio 的 DumpBin.exe 实用程序（带有-exports 开关），能够看到

DLL 的输出节是个什么样子。Kernel32.dll 的输出结果如图 10-1 所示。

 图 10-1 dumpbin 输出动态链接库的导出函数

通过 Visual Studio 所提供的 Dependency Walker 的可视化工具也可以查看动态链接库的导出函数信息，如图 10-2 所示。

 图 10-2 利用 Dependency Walker 工具查看导出函数信息

## 10.3 两种链接 DLL 的方式

如果线程需要调用 DLL 模块中的函数，那么 DLL 文件映像必须映射到调用线程的进程地址空间中。可以用两种方法进行这项操作。第一种方法是让应用程序的源代码只引用 DLL 中包含的符号。这样，当应用程序启动运行时，加载程序就能够隐含加载（或链接）需要的 DLL。第二种方法是在运行时让应用程序显式加载需要的 DLL 并且显式链接到需要的输出符号。换句话说，当应用程序运行时，其中的线程决定它是否要调用 DLL 中的函数。该线程可以将 DLL 显式加载到进程的地址空间，获得 DLL 中包含的函数的虚拟内存地址，然后使用该内存地址调用该函数。该方法的一切操作都是在应用程序运行时进行的。

当线程加载动态链接库的时候，是按照下面的搜索顺序查找并加载动态链接库文件的。

* 当前目录下（首先将动态链接库拷贝至 DEBUG 目录下，因为可执行文件在该目录下）。
* Windows 目录。
* Windows 系统目录。
* PATH 环境变量中设置的目录。
* 列入映射网络的目录表中的目录。

下面将介绍隐式链接和显式链接这两种调用 DLL 的方式。

### 10.3.1 隐式链接

如果程序员采用隐式链接方式建立一个 DLL 文件，链接程序会自动生成一个与之对应的 LIB 导入文件。该文件包含了每一个 DLL 导出函数的符号名和可选的标识号，但是并不含有实际的代码。LIB 文件作为 DLL 的替代文件被编译到应用程序项目中。当程序员通过静态链接方式编译生成应用程序时，应用程序中的调用函数与 LIB 文件中导出符号相匹配，这些符号或标识号进入生成的 EXE 文件中。LIB 文件中也包含了对应的 DLL 文件名（但不是完全的路径名），链接程序将其存储在 EXE 文件内部。当应用程序运行过程中需要加载 DLL 文件时，Windows 根据这些信息发现并加载 DLL，然后通过符号名或标识号实现对 DLL 函数的动态链接。

下面的例子通过隐式链接调用 MyDll.dll 库中的 Min 函数。首先生成一个 TestDll 项目，在 DllTest.h、DllTest.cpp 文件中分别输入如下代码：

|  |
| --- |
| //Dlltest.h  #pragma comment(lib，"MyDll.lib")  extern "C"\_declspec(dllimport) int Max(int a,int b); extern "C"\_declspec(dllimport) int Min(int a,int b);  //TestDll.cpp #include"Dlltest.h" void main()  {  int a; a=min(8,10); |

|  |  |
| --- | --- |
| } | printf("比较的结果为%d\n"，a); |

在创建 DllTest.exe 文件之前，要先将 MyDll.dll 和 MyDll.lib 拷贝到当前工程所在的目录下，也可以拷贝到windows的System目录下。如果DLL使用的是DEF文件，要删除TestDll.h 文件中关键字 extern "C"。TestDll.h 文件中的关键字 Progam commit 是要 Visual C++的编译器在 link 时，链接到 MyDll.lib 文件。当然，开发人员也可以不使用#pragma comment(lib，

"MyDll.lib")语句，而直接在工程的 Setting→Link 页的 Object/Moduls 栏填入 MyDll.lib 即可。

### 10.3.2 显式链接

显式链接方式对于集成化的开发语言（例如 Visual Basic）比较适合。有了显式链接，程序员就不必再使用导入文件，而是直接调用 Win32 的 LoadLibary 函数，并指定 DLL 的路径作为参数。LoadLibary 返回 HINSTANCE 参数，应用程序在调用 GetProcAddress 函数时使用这一参数。GetProcAddress 函数将符号名或标识号转换为 DLL 内部的地址。假设有一个导出如下函数的 DLL 文件：

extern "C" \_\_declspec(dllexport) double SquareRoot(double d);

在隐式链接方式中，所有被应用程序调用的 DLL 文件都会在应用程序 EXE 文件加载时被加载在到内存中。但如果采用显式链接方式，程序员可以决定 DLL 文件何时加载或不加载。显式链接在运行时决定加载哪个 DLL 文件。例如，可以将一个带有字符串资源的 DLL 模块以英语加载，而另一个以西班牙语加载。应用程序在用户选择了合适的语种后再加载与之对应的 DLL 文件。

在显式链接方式中，应用程序在执行过程中随时可以加载 DLL 文件，也可以随时卸载 DLL 文件，这是隐式链接所无法作到的，所以显式链接具有更好的灵活性，更适合解释性语言。不过实现显式链接比较复杂，除了要调用特定的Win32的LoadLibrary函数动态链接DLL，在应用程序退出之前，还应该用 FreeLibrary 或 MFC 提供的 AfxFreeLibrary 释放动态链接库。下面是通过显式链接调用 DLL 中的 Max 函数的例子，代码如下：

|  |
| --- |
| #include … void main(void)  {  typedef int(\*pMax)(int a,int b); typedef int(\*pMin)(int a,int b);  HINSTANCE hDLL;  PMax Max;  HDLL=LoadLibrary("MyDll.dll");//加载动态链接库 MyDll.dll 文件；  Max=(pMax)GetProcAddress(hDLL,"Max"); A=Max(5,8);  printf("比较的结果为%d\n"，a);  FreeLibrary(hDLL);//卸载 MyDll.dll 文件；  } |

在上面的程序片断中使用类型定义关键字 typedef，定义指向和 DLL 中相同的函数原型指针，然后通过 LoadLibray()将 DLL 加载到当前的应用程序中并返回到当前 DLL 文件的句柄，然后通过 GetProcAddress()函数获取导入到应用程序中的函数指针。函数调用完毕后，使用 FreeLibrary()卸载 DLL 文件。在编译程序之前，首先要将 DLL 文件拷贝到工程所在的目录或 Windows 系统目录下。

使用显式链接应用程序编译时不需要使用相应的 LIB 文件。另外，使用 GetProcAddress() 函数时，可以利用 MAKEINTRESOURCE()函数直接使用 DLL 中函数出现的顺序号，如将

GetProcAddress(hDLL,"Min") 改为 GetProcAddress(hDLL,MAKEINTRESOURCE(2)) （函数

Min()在 DLL 中的顺序号是 2），这样调用 DLL 中函数的速度将会很快，但是要记住函数的使用序号，否则会发生错误。

## 10.4 开发 DLL

在 Visual C++6.0 开发环境下，打开“ FileNewProject” 选项，可以通过选择 Win32

Dynamic-Link Library 或 MFC AppWizard[dll]的不同方式来创建 Non-MFC Dll、Regular Dll、 Extension Dll 等不同种类的动态链接库。

### 10.4.1 创建 Non-MFC DLL 动态链接库

每一个 DLL 必须有一个入口点，这就像用 C 编写的应用程序一样，必须有一个 WinMain 函数一样。在 Non-MFC DLL 中 DllMain()是一个默认的入口函数，不需要编写自己的 DLL 入口函数，用 DllMain()函数就能使动态链接库在被调用时得到正确的初始化。如果应用程序的 DLL 需要分配额外的内存或资源时，或者说需要对每个进程或线程初始化和清除操作时，需要在相应的 DLL 工程的 CPP 文件中对 DllMain()函数按照下面的格式编写。

BOOL APIENTRY DllMain(HANDLE hModule,DWORD ul\_reason\_for\_call,LPVOID lpReserved)

{

switch( ul\_reason\_for\_call )

{

case DLL\_PROCESS\_ATTACH:

.......

case DLL\_THREAD\_ATTACH:

.......

case DLL\_THREAD\_DETACH:

.......

case DLL\_PROCESS\_DETACH:

.......

}

return TRUE;

}

关于 DllMain()函数，在 10.2.1 节中已经作了详细介绍，这里不再赘述。DLL 是包含若干个函数的库文件，应用程序使用 DLL 中的函数之前，应该先导出这些函数，以便供给应用程序使用。要导出这些函数有两种方法，一是在定义函数时使用导出关键字 \_declspec(dllexport)，另外一种方法是在创建 DLL 文件时使用模块定义 DEF 文件。需要读者注意的是在使用第一种方法的时候，不能使用 DEF 文件。下面通过两个例子来说明使用这两种方法创建 DLL 文件的方法。

1．使用关键字\_declspec(dllexport)

使用导出函数关键字\_declspec(dllexport)创建 MyDll.dll，该动态链接库中有两个函数，分别用来实现得到两个数的最大和最小值。在 MyDll.h 和 MyDLL.cpp 文件中分别输入如下原代码：

|  |
| --- |
| //MyDLL.h  extern "C" \_declspec(dllexport) int Max(int a, int b); extern "C" \_declspec(dllexport) int Min(int a, int b);  //MyDll.cpp  #include  #include"MyDll.h" int Max(int a, int b)  {  if(a>=b) return a; else return b;  }  int Min(int a, int b)  {  if(a>=b) return b; else return a;  } |

该动态链接库编译成功后，打开“ MyDll” 工程中的“ debug” 目录，可以看到 MyDll.dll、

MyDll.lib 两个文件。LIB 文件中包含 DLL 文件名和 DLL 文件中的函数名等，该文件只是对应 DLL 文件的“ 映像文件”，比 DLL 文件中 LIB 文件的长度小，在进行隐式链接 DLL 时要用到它。在 MyDll.h 中有关键字"extern C"，它可以使其他编程语言访问所编写的 DLL 中的函数。

2．用 DEF 文件创建工程

为了用 DEF 文件创建 DLL，请先删除上个例子创建的工程中的 MyDll.h 文件，保留 MyDll.cpp 并在该文件中删除#include MyDll.h 语句，同时加入一个文本文件，命名为

MyDll.def，再添加如下代码：

LIBRARY MyDll

EXPORTS

Max Min

其中 LIBRARY 语句说明该 DEF 文件属于相应的 DLL，可以在 EXPORTS 语句下列出要导出的函数名称。如果在 DEF 文件中的导出函数后加@n，如 Max@1 和 Min@2，表示要导出的函数顺序号，在进行显式连时可以用到它。该 DLL 编译成功后，打开工程中的 Debug 目录，同样也会看到 MyDll.dll 和 MyDll.lib 文件。

### 10.4.2 MFC AppWizard[dll]方式生成常规/扩展 DLL

在 MFC AppWizard[dll]下生成 DLL 文件有 3 种方式，在创建 DLL 时，要根据实际情况选择创建 DLL 的方式。一种是常规 DLL 静态链接到 MFC，另一种是常规 DLL 动态链接到 MFC。前者使用的是 MFC 的静态链接库，生成的 DLL 文件长度大，一般不使用这种方式；后者使用 MFC 的动态链接库，生成的 DLL 文件长度小。动态链接到 MFC 的规则 DLL 所有输出的函数应该以如下语句开始：

AFX\_MANAGE\_STATE(AfxGetStaticModuleState()) //此语句用来正确地切换 MFC 模块状态

最后一种是 MFC 扩展 DLL，这种 DLL 特点是用来建立 MFC 的派生类，DLL 只被用 MFC 类库所编写的应用程序所调用。前面已经介绍过，Extension DLLs 和 Regular DLLs 不一样，它没有一个从 CWinApp 继承而来的类的对象，编译器默认了一个 DLL 入口函数

DLLMain()作为对 DLL 的初始化，可以在此函数中实现初始化，代码如下：

BOOL WINAPI APIENTRY DLLMain(HINSTANCE hinstDll，DWORD reason，LPVOID flmpload)

{

switch(reason)

{

⋯ ⋯ ⋯ ⋯ ⋯ //初始化代码；

}

return true;

}

参数 hinstDll 表示存放 DLL 的句柄，参数 reason 指明调用函数的原因。对于隐式链接是一个非零值，对于显式链接值是零。

在 MFC 下建立 DLL 文件，会自动生成 def 文件框架，其他与建立传统的 Non-MFC DLL 没有什么区别，只要在相应的头文件写入关键字\_declspec(dllexport)函数类型和函数名等，或在生成的 def 文件中 EXPORTS 下输入函数名就可以了。需要注意的是在向其他开发人员分发 MFC 扩展 DLL 时，不要忘记提供描述 DLL 中类的头文件以及相应的 LIB 文件和 DLL 本身，此后开发人员就能充分利用开发的扩展 DLL 了。

### 10.4.3 导出函数调用约定

关于动态链接库输出函数的约定有调用约定和名字修饰约定两种。调用约定决定着函数参数传送时入栈和出栈的顺序，以及编译器用来识别函数名字的修饰约定。名字修饰约定随调用约定和编译种类（C 或 C++）的不同而变化。为了让不同的编程语言共享动态链接库带来的方便，函数输出时必须使用正确的调用约定，并且最好不带有任何由编译器生成的名字修饰。下面详细介绍实现这些需求的方法。

1．调用约定

Visual C++ 6.0 支持的函数调用约定有多种，在这里主要介绍\_stdcall 调用约定、C 调用约定和\_fastcall 调用约定。

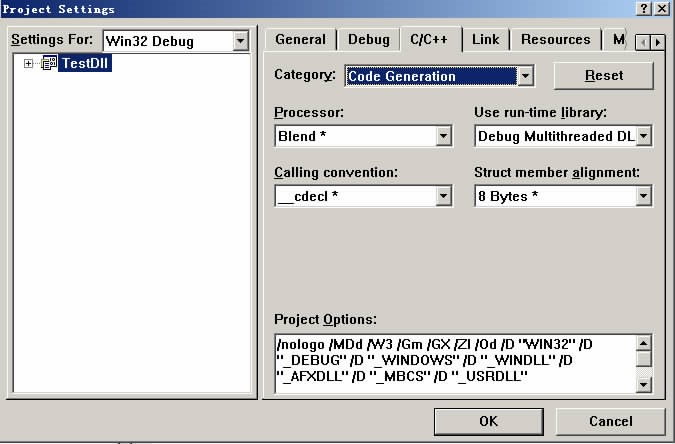
\_stdcall 调用约定相当于 16 位动态库中经常使用的 PASCAL 调用约定。在 32 位的 Visual C++ 6.0 中 PASCAL 调用约定不再被支持（实际上它已被定义为\_stdcall。除了\_pascal 外， \_fortran 和\_syscall 也不被支持），取而代之的是\_stdcall 调用约定。两者实质上是一致的，即函数的参数自右向左通过栈传递，被调用的函数在返回前清理传送参数的内存栈，但不同的是函数名的修饰部分（具体参见下一节介绍）。

C 调用约定（即用\_cdecl 关键字说明）和\_stdcall 调用约定有所不同，虽然在参数传送方面是一样的，但 C 调用约定对于传送参数的内存栈却是由调用者来维护的（也正因为如此，实现可变参数的函数只能使用该调用约定），另外，在函数名修饰约定方面也有所不同。

\_fastcall 调用约定的主要特点就是快，因为它是通过寄存器来传送参数的（实际上，它用 ECX 和 EDX 传送前两个双字或更小的参数，剩下的参数仍旧自右向左压栈传送，被调用的函数在返回前清理传送参数的内存栈），在函数名修饰约定方面，它和前两者均不同。

关键字\_stdcall、\_cdecl 和\_fastcall 可以直接加在要输出的函数前，也可以在编译环境中选择“ Setting...|C/C++|Code Generation” 菜单命令，在如图 10-3 所示的对话框中设置编译环境。当加在输出函数前的关键字与编译环境中的选择不同时，直接加在输出函数前的关键字有效。

它们对应的命令行参数分别为/Gz、/Gd 和/Gr。默认状态为/Gd，即\_cdecl。

 图 10-3 编译环境设置

如果要完全模仿 PASCAL 调用约定首先必须使用\_stdcall 调用约定，至于函数名修饰约定，可以通过其他方法模仿。另外需要注意的是 Windows.h 支持 WINAPI 宏，该宏可以将输出函数翻译成适当的调用约定，在 WIN32 中，它被定义为\_stdcall。

2．函数名修饰约定

函数名修饰约定随编译种类和调用约定的不同而不同，下面分别说明。对于 C 编译， \_stdcall 调用约定在输出函数名前加上一个下划线前缀，后面加上一个“ @” 符号和其参数的字节数，格式为\_functionname@number。\_cdecl 调用约定仅在输出函数名前加上一个下划线前缀，格式为\_functionname。\_fastcall 调用约定在输出函数名前加上一个“ @”符号，后面也是一个“ @” 符号和其参数的字节数，格式为@functionname@number。它们均不改变输出函数名中字符的大小写，这和 PASCAL 调用约定不同，PASCAL 约定输出的函数名无任何修饰且全部大写。本例将给出一种完全模仿 PASCAL 调用约定的方法，在 DEF 文件的 EXPORTS 段通过别名来实现。例如：

int \_\_stdcall MyFunc (int a,double b); void \_\_stdcall InitCode (void);

在 DEF 文件中：

EXPORTS

MYFUNC=\_MyFunc@12

INITCODE=\_InitCode@0

Visual C++编译输出的函数名修饰较为复杂，编译时 stdcall 调用约定规则如下。

* 以“ ?” 标识函数名的开始，后跟函数名。
* 函数名后面以“ @@YG” 标识参数表的开始，后跟参数表。
* 参数表的第一项为该函数的返回值类型，其后依次为参数的数据类型。
* 参数表后以“ @Z” 标识整个名字的结束，如果该函数无参数，则以“ Z” 标识结束。其格式为“ ?functionname@@YG\*\*\*\*\*@Z”或“?functionname@@YG\*XZ”，例如： int Test1(char \*var1,unsigned long)-----“ ?Test1@@YGHPADK@Z”，void Test2()-----

“ ?Test2@@YGXXZ”。

cdecl 调用约定规则同上面的 stdcall 调用约定类似，只是参数表的开始标识由上面的

“ @@YG” 变为“ @@YA”。 fastcall 调用约定规则同上面的 stdcall 调用约定，只是参数表的开始标识由上面的 “ @@YG” 变为“ @@YI”。

3．得到没有修饰的函数名

Visual C++ 输 出 函 数 时 使 用 \_declspec(dllexport) ， 而 不 再 用 \_export 修 饰 字 。

\_declspec(dllexport)在 C 调用约定、C 编译情况下可以去掉输出函数名的下划线前缀。extern "C"使得在 C++中使用 C 编译方式成为可能，在一个 C++文件中，用 extern "C"来指明该函数使用 C 编译方式。例如，在一个 C++文件中，有如下函数：

extern "C" {void \_\_declspec(dllexport) \_\_cdecl Test(int var);}

其输出函数名为 Test。

为了方便，可以使用下列预处理语句：

#if defined(\_\_cplusplus) extern "C" {

#endif

//函数原型说明

#if defined(\_\_cplusplus)

}

#endif

经过上面的特殊处理，不管在 C 中，还是在 C++中都可以得到一个无任何修饰的函数名。

如果不用\_declspec(dllexport)修饰字输出函数，而用 DEF 文件来输出函数。将要输出的函数修饰名罗列在 EXPORTS 之下，这个名字必须与定义函数的名字完全一致，如此就得到一个没有任何修饰的函数名了。

### 10.4.4 模块定义文件（DEF 文件）

模块定义文件（DEF）是一个或多个用于描述 DLL 属性的模块语句组成的文本文件，每个 DEF 文件至少必须包含以下模块定义语句。

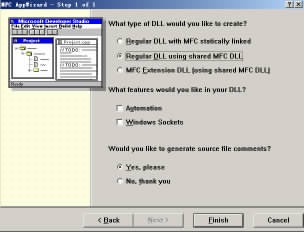
* 第一个语句必须是 LIBRARY 语句，指出 DLL 的名字；
* EXPORTS 语句列出被导出函数的名字；将要输出的函数修饰名罗列在 EXPORTS 下，这个名字必须与定义函数的名字完全一致，如此就得到一个没有任何修饰的函数名。
* 可以使用 DEs criptION 语句描述 DLL 的用途（此句可选）；
* ";"对一行进行注释（可选）。

## 10.5 动态链接库中的资源

利用 Visual C++工具能够创建基于 MFC 的常规/扩张动态链接库，这种形式的动态链接库可以方便地添加资源。应用程序在调用 DLL 的时候可以很方便地使用其中的资源，这在很大程度上方便了应用程序的编写。为了使读者理解在动态链接库中使用资源的方法，本节将通过在动态链接库中添加对话框资源来介绍使用资源的具体过程。

实例 10-1：动态链接库中使用资源实例。源代码在光盘中“ \10\实例 10-1\vcDlgDLL” 目录下。

为了能够在动态链接库中方便地使用资源，可以先创建一个“ MFC AppWizard(dll)” 工程，如图 10-4 所示。然后选择“ Regular Dll using shared MFC DLL” 项，单击“ Finish” 按钮即可生成动态链接库。

 图 10-4 生成 Regular Dll using shared MFC DLL

添加一个对话框资源到工程中，从菜单中依次选择“ Insert”→“ Resource”项，添加“ Dialog” 选择“ New” 项，至此对话框已添加到 DLL 工程中。

为对话框添加 CTest 类，基类为 CDialog。同时在 MFCDLL.cpp 中(因创建的工程为

MFCDLL)添加测试接口函数如下：

extern "C" \_\_declspec(dllexport) void Show()

{

AFX\_MANAGE\_STATE(AfxGetStaticModuleState()); CTest test; test.DoModal ();

}

静态连接到 MFC 的动态链接库只被 Visual C++的专业版和企业版所支持。该类 DLL 应用程序里头的输出函数可以被任意的 Win32 程序使用，包括使用 MFC 的应用程序。输入函数有如下形式：

extern "C" EXPORT YourExportedFunction();

如果没有 extern "C"修饰，输出函数仅仅能从 C++代码中调用。

DLL 应用程序从 CWinApp 派生，但没有消息循环。

动态链接到 MFC 的规则 DLL 应用程序里头的输出函数可以被任意 Win32 程序使用，包

括使用 MFC 的应用程序。但是，所有从 DLL 输出的函数应该以如下语句开始：

AFX\_MANAGE\_STATE(AfxGetStaticModuleState())

此语句用来正确地切换 MFC 模块状态。

因为一般在动态链接库中使用嵌入资源是通过动态链接的，所以测试接口函数作了上述

程序片断的声明和定义。

在要输出的函数、类、数据的声明前加上\_declspec(dllexport)的修饰符，表示输出。

\_declspec(dllexport)在 C 调用约定、C 编译情况下可以去掉输出函数名的下划线前缀。extern "C"使得在 C++中使用 C 编译方式成为可能。在“ C++” 下定义"C"函数，需要加 extern "C" 关键词。用 extern "C"来指明该函数使用 C 编译方式。输出的"C"函数可以从"C"代码里调用。下面以具体示例作详细介绍。在一个 C++文件中，有如下函数：

extern "C" {void \_\_declspec(dllexport) \_\_cdecl Test(int var);}

其输出函数名为 Test。

MFC 提供了一些宏，就有这样的作用。代码如下：

AFX\_CLASS\_IMPORT：\_\_declspec(dllexport) AFX\_API\_IMPORT：\_\_declspec(dllexport)

AFX\_DATA\_IMPORT：\_\_declspec(dllexport)

AFX\_CLASS\_EXPORT：\_\_declspec(dllexport)

AFX\_API\_EXPORT：\_\_declspec(dllexport)

AFX\_DATA\_EXPORT：\_\_declspec(dllexport)

AFX\_EXT\_CLASS： #ifdef \_AFXEXT

AFX\_CLASS\_EXPORT

#else

AFX\_CLASS\_IMPORT

AFX\_EXT\_API：#ifdef \_AFXEXT

AFX\_API\_EXPORT

#else

AFX\_API\_IMPORT

AFX\_EXT\_DATA：#ifdef \_AFXEXT

AFX\_DATA\_EXPORT

#else

AFX\_DATA\_IMPORT

像 AFX\_EXT\_CLASS 这样的宏，如果用于 DLL 应用程序的实现，则表示输出（因为 \_AFX\_EXT 被定义，通常是在编译器的标识参数中指定该选项/D\_AFX\_EXT）。如果用于使用 DLL 的应用程序中，则表示输入（\_AFX\_EXT 没有定义）。

要输出整个的类，对类使用\_declspec(\_dllexpot)；要输出类的成员函数，则对该函数使用

\_declspec(\_dllexport)。代码如下：

class AFX\_EXT\_CLASS CTextDoc : public CDocument

{

⋯

}

extern "C" AFX\_EXT\_API void WINAPI InitMYDLL();

为了调用动态链接库的资源，新建了一个基于对话框的 MFC 应用程序测试函数接口，测试的主要代码如下：

|  |
| --- |
| void CTestDlg::OnOK()  {  // TODO: Add extra validation here  typedef void (WINAPI \* TESTDLL)(); //定义函数指针类型 TESTDLL HINSTANCE hmod; //动态链接库实例句柄 hmod = ::LoadLibrary ("mfcdll.dll"); //加载 mfcdll.dll 动态链接库 if(hmod==NULL)  {  AfxMessageBox("Fail"); //如果加载动态链接库失败  }  TESTDLL lpproc; //定义接收导出函数的函数指针 lpproc  lpproc = (TESTDLL)GetProcAddress (hmod,"Show"); //得到导出函数 Show 的地址 if(lpproc!=(TESTDLL)NULL) //当成功得到导出函数地址的时候  (\*lpproc)(); //执行动态链接库中的导出函数    FreeLibrary(hmod); //释放动态链接库资源 |

|  |  |
| --- | --- |
| //  } | CDialog::OnOK(); |

执行的结果如图 10-5 所示。从图中可以看出，可执行程序成功地调用了动态链接库中的对话框资源。

 图 10-5 调用 DLL 中的对话框资源

## 10.6 界面汉化示例

在 Windows 系统中，应用程序开发者可以通过利用动态链接库，用一套源代码来支持多种文字。本节将通过一个界面汉化的示例来演示动态链接库中资源的应用方法。

利用 Visual C++编程，可以把所有的可见资源封装在一个资源动态链接库中，以简化本地化工作。一般情况下，资源包含在应用程序中，但也可以通过调用 AfxSetResourceHandle 函数指向一个不同的单元以完成资源的引用。本例就是首先调用该函数从动态链接库中采集到所有的应用程序可用资源，然后通过调用 GetSystemDefaultLangID 函数判断系统默认语种，以载入不同语种的资源动态链接库实现界面与系统的自动适应。

实例 10-2：界面汉化实例。源代码在光盘中“ \10\实例 10-2” 目录下。

本节以创建一个默认语种为简体中文的 Languages 应用程序为例，进一步详细说明。该程序不含任何资源，应用程序根据系统的语种设置连接对应的资源链接库，以完成对中文和英文两种语言的支持。

附带光盘中的 chinese.dll 和 english.dll 两个动态链接库分别封装了中文资源和英文资源，下面具体介绍 Languages 应用程序的实现过程。

1．创建 Languages

用 MFC AppWizad(exe)创建新项目的工作区，选择 Simple Document 类型、中国中文(其他的选项选择默认)，并且为了明确，将工作区目录改为“ 多语种支持”。

2．添加数据成员由于要动态地装入资源链接库，所以需要保存链接库的句柄以便在程序结束的时候释放资源。将下述数据成员添加到 CLanguagesApp 类中，代码如下：

|  |
| --- |
| protected:  //资源链接库句柄  HINSTANCE m\_hLangDLL; |

3．修改 InitInstance()函数

应用程序需要判别系统的默认语种，并装入对应的资源链接库。将下面的代码加到

InitInstance 函数中，代码如下：

|  |
| --- |
| BOOL CLanguagesApp::InitInstance()  {  AfxEnableControlContainer();  // 判定系统默认语种  WORD wLangPID=PRIMARYLANGID(::GetSystemDefaultLangID());  // 载入资源动态链接库  switch( wLangPID)  {  case LANG\_CHINESE:  m\_hLangDLL=::LoadLibrary("chinese.dll");  break ;  default:  m\_hLangDLL=::LoadLibrary("english.dll");  break;  }  if(!m\_hLangDLL)  {  AfxMessageBox(\_T("无法装载资源链接库!")) ; return FALSE ;  }  // 连接资源  AfxSetResourceHandle(m\_hLangDLL) ;  ⋯ ⋯  } |

其中，操作系统所使用的默认语种由 Win 32 函数 GetSystemDefaultLangID 取得。宏 PRIMARYLANGID 取出主语言标识符进行判断，以正确选择应该调用的链接库。链接库的加载由 Win32 函数 LoadLibrary 实现。程序中所使用的资源库由 AfxSetResourceHandle 函数指定。

4．处理 ExitInstance()函数

程序退出时使用 Win 32 函数 FreeLibrary 卸载已经装入的动态链接库。将下列代码添加到 ExitInstance 函数中，代码如下：

int CLanguagesApp::ExitInstance()

{

//释放资源链接库

if(m\_hLangDLL)

AfxFreeLibrary(m\_hLangDLL); return CWinApp::ExitInstance();

}

5．修改 OnDraw

为了说明是从资源链接库中动态地获取数据而不是从程序的执行体中获取，该程序从资源链接库中获取了一个字符串，并绘制在屏幕上。代码如下：

|  |
| --- |
| void CLanguagesView::OnDraw(CDC\* pDC)  {  CLanguagesDoc\* pDoc = GetDocument();  ASSERT\_VALID(pDoc); CString strMsg;  strMsg.LoadString(IDS\_HELLO); pDC->TextOut(60,15,strMsg);  // TODO: add draw code for native data here  } |

6．从项目中移走 Languages.rc 文件

因为 Languages 自身并不需要含有任何资源，所以可以从项目列表文件中移走 Languages.rc 文件（并不删除）。ClassWizad 不允许项目不包含资源文件，可以创建一个空资源文件并将其加入到项目中，然后重建信息文件（Languages.CLW），使得 ClassWizad 可以继续管理程序的消息处理。但在本例中所有的函数都已经加入，可不必为项目创建空资源文件。

7．建立项目

修改设置，选择“ Project| Setting⋯ ”菜单命令，在 Link 标签下将输出文件名 \Debug\Languages.exe 改为 Languages.exe。编译项目，如果应用程序中没有任何资源，会出现找不到资源链接库的错误，下面的一系列步骤为本项目分别创建简体中文和美国英语的资

源链接库。在这两个链接库建立后，把它们放到 Languages 项目目录中，程序就可以运行了。

建立中文资源链接库的主要步骤如下。

* 选择 Regular DLL-based 应用程序并单击“ Finish” 按钮完成创建。动态链接库项目只包含资源，不需要其他文件，所以从项目中删除所有其他文件。
* 把 Languages 相关的资源文件拷贝到 Language\_chinese 项目的对应目录下。
* 把 Languages.rc 加入到 Language\_chinese 项目中，使得 Languages.rc 为项目中的惟一文件。
* 在串资源表（String Table）中添加串 IDS\_HELLO=200，标题为“ 这是中文版！！”。
* 从 Project 菜单中选择“ Setting…” 选项。首先，将输出文件名\Debug\Language\_chines- e.dll 改为..\chinese.dll，将动态链接库直接建立在 Languages 项目目录下。然后在链接命令行中添加“ /NOENTRY” 选项。最后，编译并建立动态链接库。

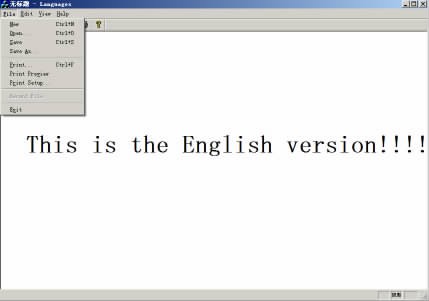
生成英文版资源文件最快捷的方法是创建一个基于英文版的临时项目，并把资源文件复制到英文链接库项目中。其处理方法与中文资源链接库类似。

注意：在修改资源组件设置时，不但要将 Project 菜单 Setting 对话框中 Resource 标签的默认文字改为“ 英语（美国）”，还要删除预处理定义中的“ \_AFXDLL”。删除\_AFXDLL 预处理器，可以使用户的链接库包含通用的 MFC 资源。否则，应用程序会从系统安装的 MFC 动态链接库中收集通用的 MFC 资源，应用程序特定资源会被本地化。

这样，应用程序和动态链接库都成功地建立完毕，Languages 应用程序将根据操作系统的不同版本来决定是加载中文资源还是英文资源。中文/英文版的应用程序运行结果分别如图

10-6 和图 10-7 所示。

 图 10-6 汉化的中文版应用程序界面

 图 10-7 英文版的应用程序界面

## 10.7 使用钩子（Hook）函数

Windows 系统是建立在事件驱动的机制上的，即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，用它可以截获并处理送给其他应用程序的消息，然后完成普通应用程序难以实现的功能。钩子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/鼠标的输入、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。

### 10.7.1 钩子函数的类型

1．按事件类型分类

按事件分类，钩子函数有如下几种常用类型。

* 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。

下面对常用的 Hook 类型进行介绍：

1. WH\_CALLWNDPROC 和 WH\_CALLWNDPROCRET Hooks

WH\_CALLWNDPROC 和 WH\_CALLWNDPROCRET Hooks 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 结构包含了来自处理消息的窗口过程的返回值，同样也包括了与这个消息关联的消息参数。

1. WH\_CBT Hook

在如下事件之前，系统都会调用 WH\_CBT Hook 子程。

* 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。

1. 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。

1. 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要变成空闲状态时，系统就会调用

WH\_FOREGROUNDIDLE Hook 子程。

1. WH\_GETMESSAGE Hook 应用程序用 WH\_GETMESSAGE Hook 来监视从 GetMessage 或者 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。
2. WH\_JOURNALPLAYBACK Hook

WH\_JOURNALPLAYBACK Hook 使应用程序可以插入消息到系统消息队列。可以使用该 Hook 回放用 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。

WH\_JOURNALPLAYBACK 是 system-wide local hooks，它们不会被映射到任何进程的地址空间中。

1. 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，它们不会被映射到任何进程的地址空间中。

1. 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 或 PeekMessage function 返回。可以用该 Hook 来监视输入到消息队列中的键盘消息。
2. WH\_KEYBOARD\_LL Hook

WH\_KEYBOARD\_LL Hook 监视输入到线程消息队列中的键盘消息。

1. WH\_MOUSE Hook

WH\_MOUSE Hook 监视从 GetMessage 或者 PeekMessage 函数中返回的鼠标消息。使用这个 Hook 监视输入到消息队列中的鼠标消息。

1. WH\_MOUSE\_LL Hook

WH\_MOUSE\_LL Hook 监视输入到线程消息队列中的鼠标消息。

1. 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 或者 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 function 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，如同在主消息循环里一样。

1. WH\_SHELL Hook

外壳应用程序可以使用 WH\_SHELL Hook 去接收重要的通知。当外壳应用程序为激活状态并且当顶层窗口建立或者销毁时，系统调用 WH\_SHELL Hook 子程。WH\_SHELL 共有 5 种情況：

* 只要有 top-level、unowned 窗口被产生、起作用、或是被摧毁；
* 当 Taskbar 需要重画某个按钮；
* 当系统需要显示关于 Taskbar 的一个程序的最小化形式；
* 当目前的键盘布局状态改变；
* 当使用者按 Ctrl+Esc 去执行 Task Manager（或相同级别的程序）。

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 function 注册它自己。

2．按使用范围分类

按使用范围分类，主要有线程钩子和系统钩子。线程钩子监视指定线程的事件消息，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。

* 如果对于同一事件（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。
* 对同一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。而且最近安装的钩子放在链的开始，而最早安装的钩子放在最后，也就是后加入的先获得控制权。
* 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。

### 10.7.2 使用钩子函数

使用钩子函数的程序的步骤分为定义钩子函数、安装钩子和卸载钩子这 3 步。下面详细介绍这 3 个过程。

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举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。 也可以通过直接返回 TRUE 来丢弃该消息，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。SetWindowsHookEx 返回所安装的钩子句柄。

3．卸载钩子

当不再使用钩子时，必须及时卸载。简单地调用函数 BOOL UnhookWindowsHook-

Ex(HHOOK hhk)即可。

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 鼠标钩子应用实例

实例 10-3：鼠标钩子应用实例。源代码在光盘中“ \10\实例 10-3\MouseHook” 目录下。

本节通过一个鼠标钩子实例来介绍全局钩子的应用方法。该实例的详细源代码请见附带光盘，其中 MouseHookDll 目录下的工程是全局鼠标钩子的实现动态链接库。MouseTest 工程通过调用 MouseHookDll 的动态链接库接口安装鼠标钩子并利用。在本例中鼠标钩子函数通过判断记录当前窗口的句柄来判断鼠标是否离开窗口边界，并将鼠标位置处的窗口名称实时地显示出来。

在 MouseHookDll 中实现全局钩子，首先是全局共享数据的实现。这里利用#pragma data\_seg 建立一个新的数据段并定义共享数据，其具体格式为：

pragma data\_seg ("shareddata")

HWND sharedwnd=NULL;//共享数据

#pragma data\_seg()

所有在 data\_seg pragmas 语句之间声明的变量都将在 shareddata 段中。仅定义一个数据段还不能达到共享数据的目的，还要告诉编译器该段的属性，有两种方法可以实现该目的（其效果是相同的），一种方法是在 DEF 文件中加入如下语句：

SETCTIONS

shareddata READ WRITE SHARED

另一种方法是在项目设置链接选项中加入如下语句：

/SECTION:shareddata,rws

MouseHookDll 是一个 MFC 扩展动态链接库，其中包含的安装钩子和卸载钩子的全局函数如下：

BOOL \_\_stdcall StartHook(HWND hwnd);

BOOL \_\_stdcall StopHook();

全局共享数据声明如下：

|  |  |
| --- | --- |
| #pragma data\_seg("mydata") HWND glhPrevTarWnd=NULL; | //上次鼠标所指的窗口句柄 |
| HWND glhDisplayWnd=NULL; | //显示目标窗口标题编辑框的句柄 |
| HHOOK glhHook=NULL; | //安装的鼠标勾子句柄 |
| HINSTANCE glhInstance=NULL;  #pragma data\_seg() | //DLL 实例句柄 |

在 DEF 文件中加入如下定义：

SECTIONS mydata READ WRITE SHARED

在主文件 MouseHookDll.cpp 的 DllMain()函数中加入保存 DLL 实例句柄的语句如下：

|  |
| --- |
| extern "C" int APIENTRY  DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved) {  UNREFERENCED\_PARAMETER(lpReserved); if (dwReason == DLL\_PROCESS\_ATTACH)  {  if (!AfxInitExtensionModule(MouseHookDLL, hInstance))  return 0;  new CDynLinkLibrary(MouseHookDLL); glhInstance=hInstance; //插入保存 DLL 实例句柄  }  else if (dwReason == DLL\_PROCESS\_DETACH)  { |

|  |  |
| --- | --- |
| } | AfxTermExtensionModule(MouseHookDLL);  }  return 1; // ok |

这个函数最重要的部分是调用 AfxInitExtensionModule()，它初始化 DLL 并在 MFC 框架中正确地工作。它需要传递给 DllMain()的 DLL 实例句柄和 AFX\_EXTENSION\_MODULE 结构，结构中存在着对 MFC 有用的信息。

加载和卸载鼠标全局钩子的函数接口定义如下：

|  |
| --- |
| BOOL \_\_stdcall StartHook(HWND hWnd)  { //install hoook  hMouseHook=SetWindowsHookEx(WH\_MOUSE,MouseHookProc,glhInstance,0);  //mouse hook  if(hMouseHook)  {  glhDisplayWnd=hWnd; //设置显示目标窗口标题编辑框的句柄 return TRUE;  }  return FALSE;  }    BOOL \_\_stdcall StopHook()  { //unstall hook  BOOL mHook=UnhookWindowsHookEx(hMouseHook); if(mHook) return TRUE;  return FALSE;  } |

钩子函数的具体实现代码如下：

|  |
| --- |
| LRESULT WINAPI MouseProc(int nCode,WPARAM wparam,LPARAM lparam)  {  LPMOUSEHOOKSTRUCT pMouseHook=(MOUSEHOOKSTRUCT FAR \*) lparam; if (nCode>=0)  {  HWND glhTargetWnd=pMouseHook->hwnd; //取目标窗口句柄 HWND ParentWnd=glhTargetWnd; while (ParentWnd !=NULL)  {  glhTargetWnd=ParentWnd;  ParentWnd=GetParent(glhTargetWnd); //取应用程序主窗口句柄  } |

|  |
| --- |
| if(glhTargetWnd!=glhPrevTarWnd)  {  char szCaption[100];  GetWindowText(glhTargetWnd,szCaption,100); //取目标窗口标题 if(IsWindow(glhDisplayWnd))  SendMessage(glhDisplayWnd,WM\_SETTEXT,0,(LPARAM)(LPCTSTR)szCaption); glhPrevTarWnd=glhTargetWnd; //保存目标窗口  }  }  return CallNextHookEx(glhHook,nCode,wparam,lparam); //继续传递消息 } |

编译完成便可得到运行时所需的鼠标钩子的动态链接库 MouseHookDll.dll 和链接时用到的 MouseHookDll.lib。

MouseTest 是一个基于 Dialog 的 MFC 应用程序，在主对话框类的 OnInitDialog 函数的

"TODO 注释"后添加如下代码：

CWnd \* pwnd=GetDlgItem(IDC\_EDIT1); //取得编辑框的类指针 m\_hook.StartHook(pwnd->GetSafeHwnd()); //取得编辑框的窗口句柄并安装钩子

然后连接 DLL 库，即把 Mousehook.lib 加入到项目设置链接标签中。然后把

MouseHookDll.h 和 MouseHookDll.lib 复制到 MouseHook 工程目录中，将 MouseHookDll.dll 复制到 Debug 目录下。编译执行程序即可。当鼠标滑过窗口时便会在编辑框中将此窗口的标题显示出来。

当鼠标位于 MouseTest 的窗口内部时，显示结果如图 10-8 所示。当鼠标位于 MouseTest

窗口外部时，显示结果如图 10-9 所示，显示的鼠标当前处于的外部窗口的标题。



图 10-8 鼠标位于窗口内部时 MouseTest 的运行结果

 图 10-9 鼠标位于窗口外部时 MouseTest 的运行结果

系统钩子具有相当强大的功能，通过这种技术可以对几乎所有的 Windows 系统消息进行拦截、监视、处理。这种技术可以广泛应用于各种软件，尤其是需要有监控、自动记录等对系统进行监测功能的软件。本程序只对鼠标消息进行拦截，相应的也可以在 Win32 环境下对键盘、端口等应用此技术完成特定的功能。

### 10.7.4 键盘钩子应用实例

实例 10-4：键盘钩子应用实例。源代码在光盘中“ \10\实例 10-4\KeyBoardHook” 目录下。

本节通过一个键盘钩子的实例演示线程钩子的应用。本示例捕获用户的键盘消息并用消息框进行显示。与上一个实例的不同之处是键盘钩子不是一个全局钩子，仅仅是一个线程钩子，加载和卸载键盘钩子的函数如下：

|  |
| --- |
| BOOL \_\_stdcall installhook()  {  //hins = AfxGetInstanceHandle();  hkb=SetWindowsHookEx(WH\_KEYBOARD,(HOOKPROC)KeyboardProc,0,GetCurrentThreadId()); return TRUE;  }  BOOL \_\_stdcall UnHook()  {  BOOL unhooked = UnhookWindowsHookEx(hkb); return unhooked;  } |

键盘钩子函数定义如下：

LRESULT CALLBACK KeyboardProc(int nCode,WPARAM wParam,LPARAM lParam)

{

if(((DWORD)lParam&0x40000000) && (HC\_ACTION==nCode))

{

switch(wParam)

|  |
| --- |
| {  // ESC F1~F12  case VK\_ESCAPE:msg = "Press Key --- ESCAPE";break; case VK\_F1:msg = "Press Key --- F1";break; case VK\_F2:msg = "Press Key --- F2";break;  //此处省略了部分代码，详见本书的附带光盘目录下的 hook.cpp 文件;  if(KeyNameStr=="Num = ")  msg = "Press Key --- VK\_UP";  if(KeyNameStr=="Num Del")  msg = "Press Key --- VK\_UP";  AfxMessageBox(msg);  }  LRESULT RetVal = CallNextHookEx( hkb, nCode, wParam, lParam ); return RetVal;  } |

程序运行结果如图 10-10 所示。

 图 10-10 键盘钩子实例运行结果

从上面的代码可以看出，在加载键盘钩子的时候传入当前的线程 ID 号，从而创建线程钩子。此钩子函数仅仅针对当前进程有效，即在其他进程处于活动状态时钩子并不生效，这也是线程钩子和全局钩子的重要区别。

## 10.8 本章小结

本章介绍了动态链接库和钩子函数的概念和编程方法。首先讲述了动态链接库的应用背景、DllMain()入口函数、Dll 导出函数的概念和显示链接和隐式链接 DLL 的方法。通过 Visual C++ 6.0 可以编写标准的 Win32 动态链接库和基于 MFC 的常规和扩展动态链接库，本章给出了利用 AppWizard 开发各种不同动态链接库的方法。在 DLL 中嵌入 Windows 资源有着广泛的应用，在介绍如何调用 DLL 中的对话框资源的基础上，通过界面汉化的实例介绍了 DLL 资源在软件本地化中的应用。

钩子函数也是一个重要的 Windows 编程概念，本章介绍了不同的钩子函数类型和它们的应用方法。在全局钩子的应用中离不开动态链接库的应用，全局鼠标钩子的实例演示了加载一个全局的钩子函数的方法。同时，在实际应用中也需要仅仅针对特定线程的钩子函数，键盘钩子实例演示了创建一个线程钩子函数定对不同的键盘消息进行处理的过程。

# 第 11 章 网络编程

在众多的计算机技术中，网络技术恐怕是和人们关系最紧密的计算机技术了。近年来，网络以超乎想象的速度飞速发展，已经逐渐渗透到人们的生活中。在这种趋势下，编写网络应用程序就成为一项常见的技能。本章将系统地介绍网络编程的有关知识，主要包括以下内容：

* 网络编程的基础知识；
* 网络应用程序的基本结构及流程；
* WinSock 类详解；
* 通过一个具体的例子详细说明网络编程的实现；
* 利用 MSComm 控件进行串口编程。

## 11.1 网络编程的基础知识

当今主要的网络模型有 OSI 参考模型、TCP/IP 参考模型、NetBEUI 参考模型和 IPX/SPX 参考模型，其中最常用的是 TCP/IP 参考模型，本章也以 TCP/IP 参考模型来讲解网络编程技术。

### 11.1.1 Socket 简介

上世纪 80 年代初，加利福尼亚大学 Berkeley 分校在 UNIX 操作系统下实现了 TCP/IP 协议，它们为 TCP/IP 网络通信开发了一个应用程序接口（API），这个 API 称为 socket（套接字）接口。socket 给程序员提供了一个高层接口，它的出现使得程序员在编写网络应用程序时只需要调用函数，对网络的底层细节并不需要精通，因此十分方便。

Socket 在 UNIX 上的巨大成功使得人们迫切想要在 Windows 上也得到同样的便利。在 Windows3.0 之后，Socket 逐渐引入到 Windows 操作系统中。Windows Socket 不仅包含了 Berkeley Socket 的库函数，并且针对 Windows 的消息驱动机制的特性，包含了一组针对 Windows 的扩展的库函数。在 Intel、Microsoft、Sun、SGI、Informix、Novell 等公司的大力支持下，Windows Socket（以下简称 WinSock）从 1991 年的 1.0 版发展到了 1996 年的 2.2.0 版。如今，WinSock 已经成为 Windows 网络编程的标准。

### 11.1.2 Windows 套接字规范简介

Windows 套接字规范从 1991 年的 1.0 版发展到 1996 年的 2.2.0 版，其中几个标志性的版本有 WinSock1.0、WinSock1.1 和 WinSock2.0。

1．WinSock1.0

WinSock1.0 是网络软件供应商和用户协会细致周到的工作结晶。WinSock1.0 规范的发布是为了让网络软件供应商和应用程序开发者都能够开始建立各自符合WinSock标准的实现和应用程序。

2．WinSock1.1

WinSock1.1 除了继承 WinSock1.0 的准则和结构外，还作了一些必要的改动。这些改动

除了包含了一些更加清晰的说明和对 WinSock1.0 的小改动之外，还包含了如下重大的变更。

* 为了更加简单的得到主机名和地址，增加了 gethostname()函数。
* 在 DLL 中保留了小于 1000 的序数，而对大于 1000 的序数则没有限制，这使 WinSock 供应商可以在 DLL 中加入自己的界面，而不用担心所选择的序数会和 WinSock 将来的版本冲突。
* 增加了 WSAStartup()函数和 WSACleanup()函数之间的关联，要求两个函数对应，这使得应用程序开发者和第三方 DLL 在使用 WinSock 实现时不需要考虑其他函数对这套

API 的调用。

* 调整函数 in\_addr()的返回类型，in\_addr 的结构改为无符号长整型，这个改动是为了适应不同的 C 编译器对返回类型为 4 字节结构函数的不同处理方法。
* 把 WSASsyncSelect()函数语义从“ 边缘触发” 改为“ 电平触发”，这种方式大大简化了应用程序对这个函数的调用。
* 改变了 ioctlsocket()函数中 FIONBIO 的语义。如果套接字还未完成 WSASsyncSelect() 函数的调用，则该函数返回失败。
* 为了符合 RFC1122，在套接字选项中增加了 TCP\_NODELAY。

不过 WinSock1.1 版本只支持 TCP/IP 协议。

3．WinSock2.0

WinSock2.0 是 WinSock1.1 发展的一个比较重大的变革。它可以支持多种协议，并且为了能与 WinSock1.1 更好地实现其兼容性，WinSock2.0 在 WinSock1.1 基础上作了向后兼容，即源码和二进制代码。这就实现了 WinSock 应用程序和任何版本的 WinSock 实现之间的最大的互操作性，同时也减轻了 WinSock 应用程序使用者、网络协议栈提供者和服务提供者的负担。

## 11.2 网络应用程序

### 11.2.1 网络应用程序的基本模型

在 TCP/IP 网络应用中，通信的两个进程间相互作用的主要模式是客户机/服务器模式（Client/Server model），即客户向服务器发出服务请求，服务器接收到请求后，提供相应的服务。客户机/服务器模式的建立基于以下两点：

1. 建立网络的起因是网络中软硬件资源、运算能力和信息不均等，需要共享，从而造就拥有众多资源的主机提供服务，资源较少的客户请求服务这一非对等作用。
2. 网络间进程通信完全是异步的，相互通信的进程间既不存在父子关系，又不存在共享内存缓冲区，因此需要一种机制为希望通信的进程间建立联系，为二者的数据交换提供同步，这就是基于客户机/服务器模式的 TCP/IP。

客户机/服务器模式在操作过程中采取的是主动请求方式，具体操作步骤如下。

* 首先服务器方要先启动，并根据请求提供相应服务：
* 打开一通信通道并告知本地主机，它愿意在某一端口上接收客户请求。
* 等待客户请求到达该端口。
* 接收到重复服务请求，处理该请求并发送应答信号。接收到并发服务请求，要激活一新进程来处理这个客户请求。新进程处理此客户请求，并不需要对其他请求作出应答。

服务完成后，关闭此进程与客户的通信链路，并终止该进程。

* 返回第二步，等待另一客户请求。
* 关闭服务器。

客户方的主要操作步骤如下：

* 打开一通信通道，并连接到服务器所在主机的指定端口。
* 向服务器发服务请求报文，等待并接收应答；继续提出请求。
* 请求结束后关闭通信通道并终止。

在客户机/服务器编程模型下，又分为面向连接的编程模型和无连接的编程模型。

* 面向连接的编程模型：当服务器程序的套接字创建并初始化完毕时，它先进入休眠状态，直到有客户机向该服务器程序提出连接请求。这时，服务器程序被“ 唤醒” 并开始响应客户机提出的连接请求，双方协商数据由谁来接收和由谁来发送。在数据传输完毕时，双方再分别关闭连接并释放因创建套接字而占用的资源。
* 无连接的编程模型：在传输数据前，不需要事先进行连接，有数据就进行发送，但却不对数据的顺序和正确性负责。相对于面向连接的模型，它的传输效率较高，但准确率稍低。

### 11.2.2 建立套接字对象

函数 socket()可以创建一个 socket 对象，socket()函数的原型如下：

SOCKET socket(int *af*, int *type*, int *protocol*);

各参数意义如下。

* af：协议的地址家族。在 Windows 操作系统中，它的取值只能是 AF\_INET，表示该套接字在 Internet 域中进行通信。
* type：套接字类型。当第一个参数 af 取值为 AF\_INET 时，它只有 3 种取值，如表 11-1 所示。
* protocol：指定网络协议。具体取值如表 11-2 所示。

如果创建成功，则返回一个创建好的套接字，如果创建失败，则返回 INVALID\_SOCKET，详细信息可以通过调用函数 WSAGetLastError()查看错误信息。

表 11-1 套接字类型取值及说明

|  |  |  |
| --- | --- | --- |
| 取值 | 类型 | 说明 |
| SOCK\_STREAM | TCP | 面向连接、可靠的数据传输服务，数据无差错发送，且按发送顺序接收；数据被看作是字节流，无长度限制 |
| SOCK\_DGRAM | UDP | 无连接服务。数据包以独立包形式发送，不提供无错保证，数据可能丢失，且接收顺序混乱 |
| SOCK\_RAW | Raw sockets | 允许对较低层协议，如 IP、ICMP 直接访问 |

表 11-2 网络协议取值与套接字类型关系

|  |  |  |  |
| --- | --- | --- | --- |
|  | 套接字类型 |  | 网络协议 |
| TCP |  | IPPROTO\_IP |  |
| UDP |  | IPPROTO\_UDP |  |
| Raw sockets |  | IPPROTO\_RAW |  |

### 11.2.3 绑定地址

创建好 Socket 后，通常要将本地地址附加到所创建的套接字上以便能够有效地标识此套

接字。这个过程由 bind()函数来实现，它的原型如下：

int bind(SOCKET *s*, const struct sockaddr FAR \**name*, int *namelen*);

各参数意义如下：

* s：要绑定的套接字。
* name：用来赋予套接字地址。
* namelen：name 参数的长度。其中，sockaddr 结构如下：

struct sockaddr\_in

{

short sin\_family; u\_short sin\_port;

struct in\_addr sin\_addr;

char sin\_zero[8];

};

各参数意义如下。

* sin\_family：在 Windows 操作系统中，此值为 AF\_INET。
* sin\_port：服务的端口号。
* sin\_addr：保存 IP 地址。
* sin\_zero[8]：其值为 0，只起填充作用。如果绑定成功，则返回 0；出错则返回 SOCKET\_ERROR。

### 11.2.4 建立连接

函数 connect()可以实现客户机和服务器的连接，其原型如下：

int connect(SOCKET *s*, const struct sockaddr FAR \**name,* int *namelen*);

各参数意义如下。

* s：进行连接的套接字。
* name：要连接服务器的套接字地址结构。
* namelen：name 参数的长度。

如果连接成功，此函数返回 0；失败则返回 SOCKET\_ERROR。

对于服务器来说，当客户机发来连接请求后，服务器要调用 accept()函数来响应对方的连接请求，该函数原型如下：

SOCKET accept(SOCKET *s*, struct sockaddr FAR \**addr*, int FAR \**addrlen*);

各参数意义如下。

* s：处在监听（下小节介绍）模式下的套接字。
* addr：在函数调用过程中被填充发出连接请求的客户机的 IP 地址信息。
* addrlen：addr 参数的长度。

如果成功，则返回一个新的套接字，它对应于已经接受的那个客户机连接。对于该客户机所有的后续操作，都使用这个新的套接字。原来的套接字则仍处于监听模式，继续接受其他客户机的连接。如果失败，则返回 INVALID\_SOCKET。

### 11.2.5 监听 socket

对于服务器端来说，在它接受客户机的连接之前，首先要监听。只有进入了监听模式，才能接受来自客户机的连接。这一点可以通过 listen()函数实现，它的原型如下：

int listen(SOCKET *s*, int *backlog*);

各参数意义如下。

* s：进行监听的套接字。
* backlog：正在等待连接的最大队列的长度。如果 backlog 的值为 3，有 4 个客户机同时发出连接请求，则前 3 个会放在等待连接队列中，最后一个将被忽略。

如果函数成功，则返回 0；否则返回 SOCKET\_ERROR。

### 11.2.6 数据传输

当客户机和服务器的连接建立起来以后，便可以进行数据的传输。数据的传输是网络通信的最终目的，前面所作的工作就是为了客户机可以和服务器传输数据。数据传输又包括数据发送和数据接收。

1．数据发送

数据发送是通过 send()函数来实现的，它的原型如下：

int send(SOCKET *s*, const char FAR \**buf*, int *len*, int *flags*);

各参数意义如下。

* s：已经建立连接的套接字。
* buf：字符缓冲区，区内包含即将发送的数据。
* len：缓冲区内的字符数目。
* flags：指定数据传输方式，取值可为 0、MSG\_DONTROUTE 和 MSG\_OOB，或者是这些取值进行按位或运算的结果。其中，0 表示无特殊行为；MSG\_DONTROUTE 表示传输层不要将它发出的包路由出去；MSG\_OOB 表示数据应该被带外发送。

如果发送成功，则返回发送的字节数，如果失败则返回 SOCKET\_ERROR。

2．数据接收

数据接收通过函数 recv()实现，其原型如下：

int recv(SOCKET *s*, const char FAR \**buf*, int *len*, int *flags*);

各参数意义如下。

* s：准备接收数据的套接字。
* buf：指向即将接收数据的字符缓冲区的指针。
* len：缓冲区的大小。
* flags：指定传输控制方式，取值可为 0、MSG\_PKKE 和 MSG\_OOB，或者是这些取值进行按位或运算的结果。其中，0 表示无特殊行为；MSG\_PKKE 表示把数据从接收端口复制到接收缓冲区中，并且没有从系统缓冲区中将数据删除；MSG\_OOB 表示数据是带外发送的。

如果接收成功，则返回接收的字节数，如果失败则返回 SOCKET\_ERROR。

## 11.3 WinSock 类

在上一节中简单介绍了直接利用 WinSock API 进行网络传输的基本步骤以及主要函数的使用方法，而实际利用 Visual C++ 6.0 开发网络应用程序的时候，很少直接利用这些 API 进行编程，因为 MFC 已经把这些 API 都封装到 MFC 提供的类中了。本节将详细介绍在网络编程中经常用到的 MFC 提供的两个类，即 CAsyncSocket 类和 CSocket 类。它们的继承关系如图 11-1 所示。

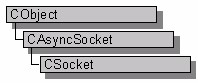


图 11-1 CAsyncSocket 类和 CSocket 类的继承关系

### 11.3.1 CAsyncSocket 类

CAsyncSocket 类在很低的级别上封装了 Windows Socket API，该类可以使程序员用面向对象的方法进行 Socket 编程，而且可以非常方便地调用其他 MFC 对象。这个类要求程序员对 Socket 编程有较为深入的了解，要面对和在直接使用 Windows Socket API 进行程序设计时一样的问题，如阻塞处理、网络字节顺序等。因为 CAsyncSocket 类几乎是一一对应地封装了

Windows Socket API，因此具有直接调用 WinSock API 的灵活性。

要使用一个 CAsyncSocket 对象，则先调用它的构造函数，然后调用 Create()函数，以创建一个套接字句柄（SOCKET 类型）。CAsyncSocket 对象既可以在栈中，也可以在堆中。对于一个服务器套接字调用 Listen()成员函数进行监听，对于一个客户套接字调用 Connect()成员函数来请求连接。在接收一个连接请求时，服务器套接字应该调用一个 Accept()函数来接收连接请求。完成之后，如果 CAsyncSocket 对象在栈中构造，则当对象超出范围时，会自动调用析构函数；如果是在堆上被创建的，必须调用 delete 来删除此对象。

CAsyncSocket 类的主要成员函数如表 11-3 所示。

表 11-3 CAsyncSocket 类主要成员函数及说明

|  |  |
| --- | --- |
| 函数 | 说明 |
| CAsyncSocket | 构造函数 |
| Create | 创建一个 Socket |
| Attach | 对 CAsyncSocket 对象附加套接字句柄 |
| Detach | 从 CasyncSocket 对象除去套接字句柄 |
| FromHandle | 返回 CasyncSocket 对象的指针，给出套接字句柄 |
| GetLastError | 获得上一次运行失败的状态 |
| GetPeerName | 获得与套接字连接的对等套接字的地址 |
| GetSockName | 获得套接字的本地名 |
| GetSockOpt | 获得套接字选项 |
| SetSockOpt | 设置套接字选项 |
| Accept | 接受套接字上的连接 |
| AsyncSelect | 请求对于套接字的事件通知 |
| Bind | 给套接字绑定本地地址 |
| Close | 关闭套接字 |
| Connect | 与对等套接字建立连接 |
| IOCtl | 控制套接字模式 |
| Listen | 建立套接字，侦听即将到来的连接请求 |
| Receive | 从套接字接收数据 |
| ReceiveFrom | 接收数据报并且存储资源地址 |
| Send | 给连接套接字发送数据 |
| SendTo | 给特定目的地发送数据 |
| ShutDown | 使套接字上的 Send 和/或 Receive 调用无效 |
| OnAccept | 通知侦听套接字，它可以通过调用 Accept，接受挂起连接请求 |
| OnClose | 通知套接字，关闭对它的套接字连接 |
| OnConnect | 通知连接套接字，连接尝试已经完成，无论成功或失败 |
| OnOutOfBandData | 通知接收套接字，在套接字上有带外数据读入，通常是忙消息 |
| OnReceive | 通知侦听套接字，通过调用 Receive 恢复数据 |
| OnSend | 通知套接字，通过调用 Send，它可以发送数据 |

除了上面的成员函数，CAsyncSocket 类还有一个数据成员 m\_hSocket，它是一个 SOCKET 句柄，记录了 CAsyncSocket 类上附加的套接字。

### 11.3.2 CSocket 类

CSocket 类是 CAsyncSocket 类的派生类，它最大的特点是提供了可以同 CArchive 和 CSocketFile 这两个类协同工作的接口。其中 CArchive 类是一个存档类，它是将用户的数据保存为永久性存储对象的一种技术，在此它是用于同 CSocketFile 类相关联而提供数据传输文件化的一种技术。而 CSocketFile 类是一种特殊的文件对象，常用它来通过套接字进行数据的传递，它所提供的接口同普通文件类的接口基本上是一致的。也就是说，用户在使用 CSocket 类的时候，不必再像使用 CAsyncSocket 类时不得不面对一大堆繁琐的工作，而只需像读写一般文件一样直接进行数据的读取就可以了。本章主要讲述网络编程，因此本小节主要介绍

CSocket 类，而 CArchive 类和 CSocketFile 类的相关信息请读者自行查阅相关资料。

1．CArchive 对象与串行化技术

CArchive 对象是 MFC 中专门帮助程序员处理归档对象的类，它最主要的作用就是把内存中的数据保存到永久存储介质中，这样程序员就不必担心使用的到底是套接字还是文件了。而为了实现 CArchive 与 CSocket 之间的连接，必须使用 CSocketFile 作为中间的转化媒质。在实际工作中 CSocketFile 对象负责连接到一个 CSocket 对象，而 CArchive 对象负责管理缓冲区。当发送归档对象的缓冲区被填满时，相关的 CSocketFile 对象会将缓冲区的内容取出，并清空与套接字相关的归档缓冲区。当接收归档对象的缓冲区被填满时，CSocketFile 对象将停止读出直到缓冲区可用。

2．用 CSocket 类的编程步骤

用 CSocket 类进行网络编程的基本步骤如下。

* 构造服务器和客户机套接字对象。
* 调用 Create()函数创建套接字。
* 服务器调用 Listen()函数监听客户机的连接请求；客户机调用 Connect()函数向服务器发出连接请求。
* 服务器监听到客户机的连接请求时，先创建一个新的套接字，然后调用 Accept()函数接收客户机的连接请求。
* 服务器和客户机的套接字对象分别建立一个与之联系的 CSocketFile 对象。
* 服务器和客户机的套接字对象分别创建一个与 CSocketFile 相联系的 CArchive 对象，以便进行数据传输。
* 使用 CArchive 对象在服务器和客户机套接字之间进行数据传输。
* 当任务完成后，销毁 CArchive、CSocketFile 和 CSocket 对象。

注意：由于 CArchive 对象只能单向传送数据，所以在实际使用的时候必须定义两个 CArchive 对象，分别用于数据的发送和接收。

3．CSocket 类的主要成员函数

CSocket 类是从 CAsyncSocket 类继承来的，除了 CAsyncSocket 类的成员函数，CSocket 类还主要有以下成员函数，如表 11-4 所示。

表 11-4 CSocket 类主要成员函数及说明

|  |  |
| --- | --- |
| 函数 | 说明 |
| CSocket | 构造一个 CSocket 对象 |
| Create | 创建一个 Socket |
| IsBlocking | 确定一个阻塞调用是否在进行中 |
| FromHandle | 返回一个指向 CSocket 对象的指针，给出一个 Socket 句柄 |
| Attach | 将一个 Socket 句柄与一个 CSocket 对象连接 |
| CancelBlockingCall | 取消一个当前在进行中的阻塞调用 |
| OnMessagePending | 当等待完成一个阻塞调用时调用此函数来处理未处理的消息 |

## 11.4 网络编程的实现

前面介绍了很多关于 Windows 网络编程的知识，也详细介绍了 WinSock 类。接下来，本节将按照上小节所说的编程步骤，利用 CSocket 类编写一个网络聊天程序。相信通过这个程序，读者会对 CSocket 的基本应用有更加深入的认识。

实例 11-1：网络聊天程序实例。源代码在光盘中“ \11\实例 11-1” 目录下。

### 11.4.1 程序设计的基本任务

为了让读者易于理解而又能体现出网络编程的一般步骤，本程序抛弃很多网络应用程序的辅助功能，只实现网络聊天系统的基本功能。该系统包括服务器和客户端两部分，服务器允许多个客户端连接，连接到服务器的客户端可以聊天，聊天的内容会在服务器和每个客户端显示出来，同时，在服务器上还可以显示出每个客户端的基本信息，比如登录时间和 IP 等。

### 11.4.2 服务器端程序设计

1．设计概述

在进行服务器端程序设计之前，首先要考虑一下消息在程序中如何处理。在整个系统中，服务器和客户端的交互都是通过消息完成的，并且对于一般的网络应用程序来说，网络数据的有效封装有利于用户对网络通信的管理，也有利于在源程序的基础上作进一步的开发。因此在本程序中，会对消息进行简单的封装。

接下来考虑界面设计。服务器端应该有两个窗口，分别显示各客户端的详细信息（如用户名、登录时间、IP 等）和用户的聊天信息。

关于程序流程在 11.3.2 节的第 2 小节已有说明。除此之外，在网络编程中还有很重要的一点，就是服务器和客户端双方要约定好信息的格式以及发送顺序，只有约定好这些，双方才知道对方所发信息的意义。其实这就是通常所说的协议的一个重要部分，在网络应用程序的设计中，协议是必不可少的，越是大型的网络应用程序，协议就越复杂。对于本例这个很小的程序，可以根据消息的类型（封装在消息类中）对消息进行分类，并且约定当用户登录或离开时，都要先发送用户名给服务器，然后再发送正式信息。这样通过消息的种类，服务器就能够知道当前消息是一般消息还是用户登录或离开的消息。如果是前者，就将其发送给每个客户端，并且在服务器界面将信息显示出来；如果是后者，则将登录或离开的用户名发送给各客户端，以使各客户端对用户列表更新，同时更新服务器端的用户列表。

最后，本程序采用文档-视图结构，视图负责显示，而数据的处理则通过文档类来实现。

2．编写代码

1. 创建工程

新建一个 MFC 工程，取名为 ChatServer。利用 AppWizard 创建 SDI 程序，在第一步中

选择“ Single Document” 项，在第四步中选中“ Windows Sockets” 复选框，如图 11-2 所示。在最后一步的“ Base Class” 下拉列表中选择“ ClistView” 作为基类，其他选项保持默认，单击“ Finish” 按钮即可完成工程的创建，如图 11-3 所示。

图 11-2 选中“Windows Sockets” 复选框 图 11-3 选择“ ClistView” 作为基类

1. 封装消息类本程序中为消息建立一个类 CMsg，它的父类是 CObject。在“ ChatServer classes” 上单击右键，在弹出的对话框中选择“ New Class”，如图 11-4 所示。在弹出的“ New Class” 对话框中的“ Class type” 下拉列表中选择“ Generic Class” 项，在“ Name” 栏中填写类名称 CMsg，在“ Derived From” 栏中填写其父类名称 CObject，如图 11-5 所示。单击“ OK” 按钮，弹出提示对话框，单击“ 确定” 即可。

图 11-4 为工程新建一个类 图 11-5 新建消息类 CMsg

在类CMsg的定义（文件“ Msg.h”）中加入宏DECLARE\_DYNCREATE(CMsg)，声明此

宏可以使得由CObject继承来的类在程序运行的时候能够动态地创建。然后为其加入成员变量和成员函数，分别如表11-5和表11-6所示。

表 11-5 CMsg 类成员变量及其说明

|  |  |  |  |
| --- | --- | --- | --- |
|  | 成员变量 |  | 说明 |
| CString m\_strText |  | 消息的内容 |  |
| BOOL m\_bClose |  | 连接是否关闭 |  |
| UINT code |  | 消息的种类 |  |

表 11-6 CMsg 类成员函数及其说明

|  |  |
| --- | --- |
| 成员函数 | 说明 |
| void Init() | 对成员变量进行初始化 |
| virtual void Serialize(CArchive &ar) | 进行数据的发送和接收 |

完成消息类的定义后，要在文件“Msg.cpp”中实现此类。首先要声明另外一个宏

IMPLEMENT\_DYNCREATE(CMsg, CObject)，它是与宏 DECLARE\_DYNCREATE 相对应的在类实现时声明的宏。然后为其添加代码。在类的构造函数中对成员变量进行初始化，代码如下：

CMsg::CMsg()

{

Init();

}

函数 Init()代码如下：

void CMsg::Init()

{

code = 0; m\_bClose = FALSE; m\_strText = \_T("");

}

最后是重载的串行化函数的代码如下：

void CMsg::Serialize(CArchive& ar)

{

//读入

if (ar.IsStoring())

{

ar << (WORD)m\_bClose; ar << m\_strText; ar << code;

} //输出

else

{ WORD wd; ar >> wd; m\_bClose = (BOOL)wd; ar >> m\_strText;

|  |  |  |
| --- | --- | --- |
|  |  | ar >> (UINT)code; |
| } | } |  |

（3）编写客户端信息界面

封装好消息类后，接下来要对界面编写代码。在上小节的设计中已经说明，服务器端应有两个窗口，一个用来显示各客户端的信息，另一个用来显示聊天信息。

首先编写显示各客户端信息的界面。与创建消息类的步骤一样，为工程新建一个类，取名为 CChattersListView，在“ Class type” 下拉列表中选择“ MFC Class” 项，在“ Base class” 下拉列表中选择“ ClistView” 作为其父类，设置完成后单击“ OK” 按钮即可，如图 11-6 所示。

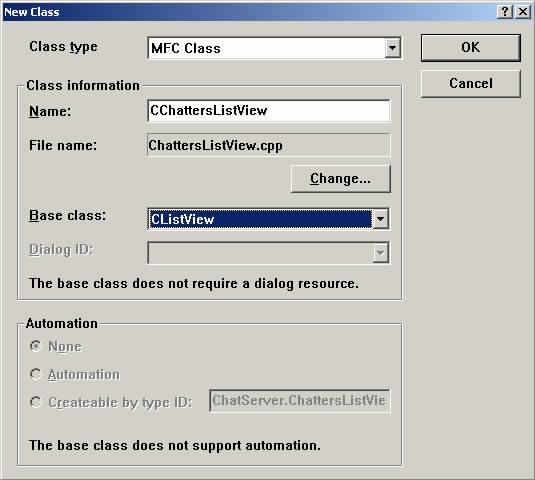


图 11-6 新建类 CChattersListView

然后为该类添加成员变量 int m\_Row，它记录列表视图的当前行的位置，在类的构造函数中将其初始化为 0。

接 下 来 添 加 各 成 员 函 数 并 为 其 编 写 代 码 。 首 先 重 载 函 数 virtual BOOL

PreCreateWindow(CREATESTRUCT& cs)，通过此函数可以设定窗口风格。代码如下：

|  |
| --- |
| BOOL CChattersListView::PreCreateWindow(CREATESTRUCT& cs)  {  //设定风格为详细报表显示  cs.style |= LVS\_REPORT; return CListView::PreCreateWindow(cs);  } |

接着在窗口创建时进行初始化。打开“ ClassWizard” 对话框，在“Class name”下拉列表中选择 CChattersListView，在“ Message” 列表中选择 WM\_CREATE（如图 11-7 所示）。然后依次单击“ Add Function” 按钮和“ Edit Code” 按钮，定位到 OnCreate()函数，添加如下代码：



图 11-7 为 WM\_CREATE 消息添加响应函数

|  |
| --- |
| int CChattersListView::OnCreate(LPCREATESTRUCT lpCreateStruct)  {  if (CListView::OnCreate(lpCreateStruct) == -1)  return -1;    // TODO: Add your specialized creation code here  CRect rect;  GetClientRect(&rect);  GetListCtrl().InsertColumn(0, "用户名" );  GetListCtrl().InsertColumn(1, "登录时间"); GetListCtrl().InsertColumn(2, "地址"); return 0;  } |

当窗口大小变化时，会激发 WM\_SIZE 消息。利用上述办法可以为此消息编写响应函数，

代码如下：

void CChattersListView::OnSize(UINT nType, int cx, int cy)

{

CListView::OnSize(nType, cx, cy);

// TODO: Add your message handler code here GetListCtrl().SetColumnWidth(0, cx/3); GetListCtrl().SetColumnWidth(1, cx/3); GetListCtrl().SetColumnWidth(2, cx/3);

}

当有新用户加入或者有用户退出时，应该更新列表，为此需要两个更新函数，分别用于

添加用户和删除用户的操作。

首先为该类添加一个添加用户信息的函数：

void AddChatter(CString *Name* , CString *IPAddress*, UINT *Port*);

各参数意义如下。

* Name：用户名。
* IPAddress：客户端 IP 地址。
* Port：客户端的端口号。为此函数编写代码如下：

|  |
| --- |
| void CChattersListView::AddChatter(CString Name, CString IPAddress, UINT Port)  {  //插入用户名  LVITEM lvi; lvi.mask = LVIF\_TEXT; lvi.iItem = m\_Row++; lvi.iSubItem = 0;  lvi.pszText = Name.GetBuffer(Name.GetLength()); int iActualItem = GetListCtrl().InsertItem(&lvi);    //插入登录时间  //得到系统时间  time\_t t; struct tm \*ptm; time(&t);  ptm = localtime(&t);  //将其转化成字符形式  CString sDate;  sDate.Format("%.19s", asctime(ptm));  //得到小时、分钟、秒  int iHours , iMins , iSecs; iHours = atoi(sDate.Mid(11,2)); iMins =atoi(sDate.Mid(14,2)); iSecs = atoi(sDate.Mid(17,2));  //将其转换成字符串  CString sHours, sMins, sSecs; sHours.Format("%d:", iHours); if(iHours < 10) sHours.Insert(0, ’0’); sMins.Format("%d:", iMins); if(iMins < 10) sMins.Insert(0, ’0’); sSecs.Format("%d", iSecs); if(iSecs < 10) |

|  |  |
| --- | --- |
| } | sSecs.Insert(0, ’0’); CString sDisplayTime;  sDisplayTime = sHours+sMins+sSecs; lvi.mask = LVIF\_TEXT; lvi.iItem = iActualItem; lvi.iSubItem = 1;  lvi.pszText = sDisplayTime.GetBuffer(sDisplayTime.GetLength());  BOOL test = GetListCtrl().SetItem(&lvi);  //插入客户端地址  CString sIPPort;  sIPPort.Format("%s:%d", IPAddress, Port); lvi.mask = LVIF\_TEXT; lvi.iSubItem = 2;  lvi.pszText = sIPPort.GetBuffer(sIPPort.GetLength());  GetListCtrl().SetItem(&lvi); |

添加一个在列表中删除某用户的操作，函数名为 void DeleteChatter(CString Name)，其中

Name 就是用户名。代码如下：

|  |
| --- |
| void CChattersListView::DeleteChatter(CString Name) {  LVFINDINFO lvfi; lvfi.flags = LVFI\_STRING;  lvfi.psz = Name.GetBuffer(Name.GetLength());  //在列表中查找用户名，得到其索引值  int nFoundAt = GetListCtrl().FindItem(&lvfi);  //如果找到则将其删除  if(nFoundAt != -1)  GetListCtrl().DeleteItem(nFoundAt);  } |

（4）编写聊天信息界面新建一个类，取名为 CChatView，其父类为 CView，该类用于显示聊天信息。为其添加

一个成员变量：CEdit m\_EditBox，负责聊天信息的显示。

为 WM\_CREATE 消息添加响应函数，在此函数中，对 m\_EditBox 进行初始化，代码如

下：

int CChatView::OnCreate(LPCREATESTRUCT lpCreateStruct)

{

if (CView::OnCreate(lpCreateStruct) == -1)

return -1;

|  |  |
| --- | --- |
| } | // TODO: Add your specialized creation code here  CRect rect;  GetClientRect(&rect);  m\_EditBox.Create(WS\_VISIBLE |  WS\_BORDER |  WS\_CHILD |  ES\_MULTILINE | WS\_VSCROLL,  rect, this , 0);  return 0; |

当窗口大小变化时，会激发 WM\_SIZE 消息。为此消息编写响应函数，代码如下：

void CChatView::OnSize(UINT nType, int cx, int cy)

{

CView::OnSize(nType, cx, cy);

// TODO: Add your message handler code here m\_EditBox.MoveWindow(0 , 0 , cx , cy , FALSE);

}

最后，为该类添加一函数 void ShowMessage(LPCTSTR lpszMessage)，此函数用于消息的

显示，其参数为消息内容。代码如下：

|  |
| --- |
| void CChatView::ShowMessage(LPCTSTR lpszMessage)  {  CString strTemp = lpszMessage; strTemp += \_T("\r\n");  int len = m\_EditBox.GetWindowTextLength(); m\_EditBox.SetSel(len,len); m\_EditBox.ReplaceSel(strTemp);  } |

至此，负责显示客户端信息以及聊天信息的两个窗口都已编写完毕，接下来要在主框架

中进行分割，也就是将主框架分割成显示客户端信息的窗口和显示聊天信息的窗口两部分。

（5）分割框架

为类 CMainFrame 添加成员变量 CSplitterWnd m\_wndSplitter，用于分割主框架。重载函数 virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext\* pContext)，该函数在主框架生成期间被调用，在此函数中添加分割窗口的操作，代码如下：

BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT /\*lpcs\*/,

CCreateContext\* pContext)

{

|  |
| --- |
| //将窗口分为两行一列  if (!m\_wndSplitter.CreateStatic(this, 2, 1))  return FALSE;    //指定每个窗口的位置及初始大小  if (!m\_wndSplitter.CreateView(0, 0, RUNTIME\_CLASS(CChattersListView), CSize(150, 100),  pContext) ||  !m\_wndSplitter.CreateView(1, 0, RUNTIME\_CLASS(CChatView), CSize(300, 300), pContext))  {  m\_wndSplitter.DestroyWindow(); return FALSE;  }  return TRUE;  } |

在此函数中用到了类 CChatView 和 CChattersListView，因此在文件“ MainFrm.cpp” 的开头要加入下面两个语句：

#include "ChatView.h"

#include "ChattersListView.h"

现在编译、运行程序，可以看到主框架已经被分割成两个窗口，如图 11-8 所示。

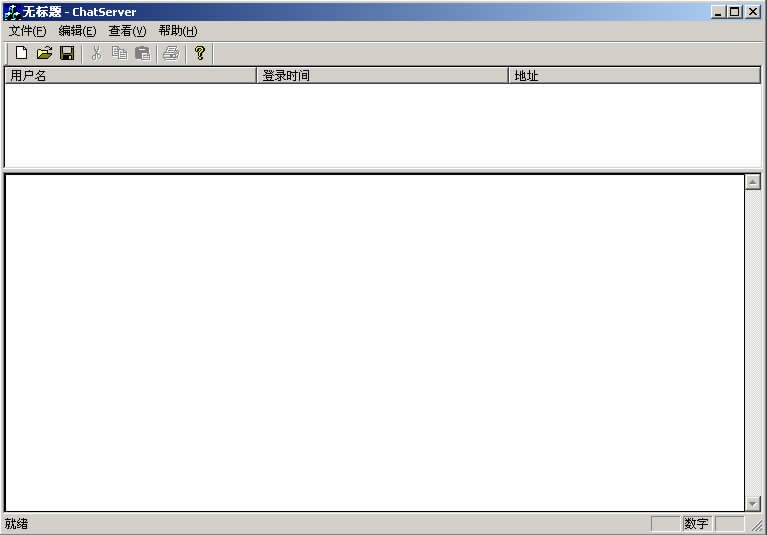


图 11-8 主框架被分割为两部分

至此，界面部分已经完成。

（6）创建 CListeningSocket 类

该类用户监听客户端的连接，创建方式与 CChattersListView 类的创建方式相同，只是基类选择 CSocket。

在类的定义中声明宏 DECLARE\_DYNAMIC(CListeningSocket)，该宏的作用是可以在运行时获得类的信息。相应的，如果在类定义中声明了此宏，则在类实现时需要声明宏

IMPLEMENT\_DYNAMIC(CListeningSocket, CSocket)。

为该类增加成员变量 CChatServerDoc\* m\_pDoc，这是因为本程序采用文档-视图结构，所有关于数据的操作都可以通过文档类进行，m\_pDoc 就是文档类的一个指针。这里用到了

CChatServerDoc 类 ，因 此 在 CListeningSocket 类 定 义的 外 部，要 加 上语句 class

CChatServerDoc，在文件“ ListeningSocket.cpp” 中，要加入语句#include "ChatServerDoc.h"。为 CListeningSocket 类新增加一个构造函数 CListeningSocket(CChatServerDoc\* pDoc)，代码如下：

CListeningSocket::CListeningSocket(CChatServerDoc\* pDoc)

{

m\_pDoc = pDoc;

}

最后要重载 CSocket 类的 OnAccept()函数，代码如下：

void CListeningSocket::OnAccept(int nErrorCode)

{

//调用 CScoekt 的 OnAccept 函数

CSocket::OnAccept(nErrorCode);

//其余工作交给文档类处理

m\_pDoc->ProcessAccept();

}

可以看出，CListeningSocket 类的工作就是接收客户端连接，然后具体的数据传输工作交给 CChatServerDoc 类来处理。这里的 ProcessAccept()函数是 CChatServerDoc 类的一个成员函数。在后面的 CChatServerDoc 类成员函数的实现中，读者可以看到此函数的具体实现过程。

（7）创建 CClientSocket 类

当服务器监听到一个连接并接收后，会生成一个新的 CSocket 对象来处理服务器和客户端的数据传输。在本程序中，将这新生成的 CSocket 封装成 CClientSocket 类。创建该类的过程和创建 CListeningSocket 类完全一样。

同样的，在类的定义中声明宏 DECLARE\_DYNAMIC(CClientSocket)，在类的实现中声明宏 IMPLEMENT\_DYNAMIC(CClientSocket, CSocket)。为该类加入成员变量，如表 11-7 所示。

表 11-7 CClientSocket 类成员变量及其说明

|  |  |
| --- | --- |
| 成员变量 | 说明 |
| CChatServerDoc\* m\_pDoc | 关联的文档类，很多数据处理工作在文档类中完成 |
| CSocketFile\* m\_pFile | CArchive 与 CSocket 的联系桥梁，通过它来创建 CArchive 类 |
| CArchive\* m\_pArchiveIn | 用于数据接收 |
| CArchive\* m\_pArchiveOut | 用于数据发送 |
| int m\_nMsgCount | 信息的数目 |

为该类加入成员函数，如表 11-8 所示。

表 11-8 CClientSocket 类成员函数及其说明

|  |  |
| --- | --- |
| 成员函数 | 说明 |
| CClientSocket(CChatServerDoc\* m\_pDoc) | 构造函数 |
| void Init() | 对成员变量进行初始化 |
| void SendMsg(CMsg\* pMsg) | 发送消息 |
| void ReceiveMsg(CMsg\* pMsg) | 接收消息 |
| virtual void OnReceive(int nErrorCode) | 重载 CSocket 的 OnReceive()函数 |

在创建的过程中用到了类 CMsg 和类 CChatServerDoc，因此在文件“ ClientSocket.h” 的开始声明两个类。

class CMsg; class CChatServerDoc;

在“ ClientSocket.cpp” 文件的开始要把两个头文件包含进来。代码如下：

#include "msg.h"

#include "ChatServerDoc.h"

最后为各函数编写代码。构造函数代码如下：

|  |
| --- |
| CClientSocket::CClientSocket(CChatServerDoc\* pDoc)  {  m\_pDoc = pDoc; m\_nMsgCount = 0; m\_pFile = NULL; m\_pArchiveIn = NULL; m\_pArchiveOut = NULL;    } |

析构函数代码如下：

CClientSocket::~CClientSocket()

{

if (m\_pArchiveOut != NULL) delete m\_pArchiveOut;

if (m\_pArchiveIn != NULL) delete m\_pArchiveIn;

if (m\_pFile != NULL) delete m\_pFile;

}

初始化函数 Init()代码如下：

void CClientSocket::Init()

{

m\_pFile = new CSocketFile(this);

m\_pArchiveIn = new CArchive(m\_pFile,CArchive::load);

|  |  |
| --- | --- |
| } | m\_pArchiveOut = new CArchive(m\_pFile,CArchive::store); |

发送信息函数 SendMsg()代码如下：

void CClientSocket::SendMsg(CMsg\* pMsg)

{

if (m\_pArchiveOut != NULL)

{

//数据由 pMsg 输出到 m\_pArchiveOut pMsg->Serialize(\*m\_pArchiveOut);

//将数据写入文件

m\_pArchiveOut->Flush();

}

}

接收信息函数 ReceiveMsg()代码如下：

void CClientSocket::ReceiveMsg(CMsg\* pMsg)

{

//数据由 m\_pArchiveIn 读入到 pMsg

pMsg->Serialize(\*m\_pArchiveIn);

}

重载的 OnReceive()函数代码如下：

void CClientSocket::OnReceive(int nErrorCode)

{

//调用 CSocekt 的 OnReceive 函数

CSocket::OnReceive(nErrorCode);

//其余工作交给文档类处理

m\_pDoc->ProcessReceive(this);

}

其中，ProcessReceive()函数是 CChatServerDoc 类的一个成员函数，在后面将会看到它的

具体实现。

1. 创建设定端口对话框对服务器来说，它可以通过“ 设定端口号” 对话框来设定服务端口，如图 11-9 所示。其中，对话框的 ID 为 IDD\_SETUPDLG，Edit 控件的 ID 为 IDC\_PORT。

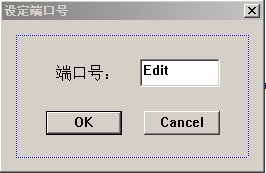


图 11-9 “ 设定端口号” 对话框

打开“ ClassWizard” 对话框，为此对话框创建一个类，取名为 CPortDlg。然后为 Edit

控件增加变量 int m\_Port，在构造函数中将其初始化为 2000。

最后，在菜单栏中新建一菜单项，当单击此菜单项时，就会弹出如图 11-10 所示的对话框。

图 11-10 对话框所对应菜单栏

此菜单项的 ID 为 ID\_SETPORT。

1. 实现 CChatServerDoc 类

大量的工作都是在 CChatServerDoc 类中实现的。在开始编写代码前，首先在文件

“ ChatServerDoc.h” 中 CChatServerDoc 类定义的前面加入如下的类声明语句：

class CListeningSocket; class CClientSocket; class CMsg;

在文件“ ChatServerDoc.cpp” 的开头加入如下的 include 语句：

#include "MainFrm.h"

#include"PortDlg.h"

#include "ChattersListView.h"

#include "ChatView.h"

#include "ListeningSocket.h"

#include "ClientSocket.h"

#include "Msg.h"

在文件“ ChatServerDoc.cpp” 的开头加入下面的定义语句（它定义了 5 种不同类型的消

息）。

#define LEAVING\_CHAT 1

#define SENDING\_CHATTERS\_LIST 2

#define SENDING\_NICKNAME 3

#define NORMAL\_MESSAGE 4

#define USED\_NAME 5

这 5 种类型消息的具体含义在下面的程序中会详细说明。接下来，为 CChatServerDoc 类添加一些成员变量，如表 11-9 所示。

表 11-9 CChatServerDoc 类成员变量及其说明

|  |  |
| --- | --- |
| 成员变量 | 说明 |
| CListeningSocket\* m\_pSocket | 监听客户连接 |
| CStringList m\_msgList | 存放所有聊天信息的链表 |
| CPtrList m\_connectionList | 存放所有客户端连接的链表 |
| BOOL bIsNewChatter | 表示新加入的用户其用户名是否已有人使用 |
| CString m\_ChattersList | 保存当前正在连接的所有用户名 |

为 CChatServerDoc 类添加成员函数，如表 11-10 所示。

表 11-10 CChatServerDoc 类成员函数及其说明

|  |  |
| --- | --- |
| 成员函数 | 说明 |
| void ProcessAccept() | 当监听到客户端连接时调用此函数进行处理 |
| void ProcessReceive(CClientSocket\* pSocket) | 当接收到信息时调用此函数，进行信息接收等操作 |
| CMsg\* ReadMsg(CClientSocket\* pSocket) | 接收信息 |
| void SendMsg(CClientSocket\* pSocket, CMsg\* pMsg) | 发送信息 |
| void SendToAllClients(int nCode) | 给各客户端发送消息 |
| CMsg\* AssembleMsg(CClientSocket\* pSocket, int nCode) | 封装消息，nCode 指定消息类型 |
| void SendForNewcomer(CMsg\* pMsg) | 给各客户端发送消息，告知有新用户加入 |
| void UpdateMessageView(LPCTSTR lpszMessage) | 更新服务器聊天信息视图 |
| void UpdateChattersListView(CString sName , CClientSocket\* pSocket) | 更新服务器用户列表视图 |
| BOOL IsUsedName(CString sNickName) | 检查新登录的用户其用户名是否已被使用 |
| void CloseSocket(CClientSocket\* pSocket) | 关闭连接 |
| void DeleteChatter(CMsg\* pMsg) | 删除用户 |
| virtual void DeleteContents() | 当程序退出时做一些关闭、删除处理 |

下面为各函数编写代码。首先在构造函数中将各成员变量初始化，代码如下：

|  |
| --- |
| CChatServerDoc::CChatServerDoc()  {  // TODO: add one-time construction code here  m\_pSocket = NULL; m\_msgList.RemoveAll(); m\_connectionList.RemoveAll(); bIsNewChatter = TRUE; m\_ChattersList = "";  } |

接着为“ 设定端口” 菜单项添加响应函数，代码如下：

void CChatServerDoc::OnSetport()

{

// TODO: Add your command handler code here CPortDlg dlg; if(dlg.DoModal() == IDOK)

{

//创建一个新的类进行监听

m\_pSocket = new CListeningSocket(this); if (m\_pSocket->Create(dlg.m\_Port))

{

if (m\_pSocket->Listen())

return;

}

}

}

为函数 ProcessAccept()添加如下代码：

void CChatServerDoc::ProcessAccept()

{

//创建一个新的 Socket 来处理与客户端的数据交互

CClientSocket\* pSocket = new CClientSocket(this);

//如果接受客户端的连接请求

if (m\_pSocket->Accept(\*pSocket))

{

//初始化

pSocket->Init();

//将此连接 socket 加入连接链表中

m\_connectionList.AddTail(pSocket);

} //如果不接受连接

else delete pSocket;

}

为函数 ProcessReceive()添加如下代码：

|  |
| --- |
| void CChatServerDoc::ProcessReceive(CClientSocket\* pSocket) {  CMsg\* pMsg; //接收的消息  CString sName; //用户名  BOOL bIsUsedName = FALSE; //用户名是否已使用 bIsNewChatter = FALSE; //是否为新用户  do  {  //接收消息  pMsg = ReadMsg(pSocket);  //如果此消息是发送的新的用户名（在用户登录时客户端先发此消息）  if(pMsg->code == SENDING\_NICKNAME)  {  sName = pMsg->m\_strText; bIsNewChatter = TRUE;  //检查此用户名是否已使用  if(IsUsedName(sName)) bIsUsedName = TRUE;  }  //如果连接已关闭则退出  if (pMsg->m\_bClose)  { |

|  |
| --- |
| break;  }  }  while (!pSocket->m\_pArchiveIn->IsBufferEmpty());    //如果是用户登录  if(bIsNewChatter)  {  //如果此用户名没有被使用  if(!bIsUsedName)  {  //发送消息通知各客户端  pMsg->code = SENDING\_CHATTERS\_LIST; m\_ChattersList += sName + ":"; pMsg->m\_strText = m\_ChattersList;  SendForNewcomer(pMsg);  }  //如果用户名已使用 else  {  //发消息通知此客户端  CMsg\* msg = new CMsg; msg->Init(); msg->code = USED\_NAME; CString string;  string.Format("用户名已被使用！"); msg->m\_strText = string;  SendMsg(pSocket, msg);  }  bIsNewChatter = FALSE; bIsUsedName = FALSE;  }  //如果消息表明该客户端已关闭连接  if (pMsg->m\_bClose)  {  //发送消息通知各客户端  SendToAllClients(NORMAL\_MESSAGE);  //在视图中将该用户删除  DeleteChatter(pMsg);  //关闭连接 |

|  |  |
| --- | --- |
| } | CloseSocket(pSocket);  //通知各客户端更新用户列表  SendToAllClients(SENDING\_CHATTERS\_LIST);  }  //如果是普通消息，发送给各客户端  else  SendToAllClients(NORMAL\_MESSAGE); |

为 ReadMsg()函数编写如下代码：

|  |
| --- |
| CMsg\* CChatServerDoc::ReadMsg(CClientSocket\* pSocket)  {  static CMsg msg;  TRY  {  //接收数据  pSocket->ReceiveMsg(&msg);  //如果是新用户登录则更新用户列表  if(msg.code == SENDING\_NICKNAME)  {  UpdateChattersListView(msg.m\_strText , pSocket);  }  //如果是普通信息则更新聊天信息列表，并将信息加入到 m\_msgList 链表中  if(msg.code == NORMAL\_MESSAGE)  {  UpdateMessageView(msg.m\_strText); m\_msgList.AddTail(msg.m\_strText);  }  }  CATCH(CFileException, e)  {  CString strTemp;  strTemp.Format("无法读取数据！");  UpdateMessageView(strTemp); msg.m\_bClose = TRUE;  }  END\_CATCH  return &msg;  } |

为 SendMsg()函数编写如下代码：

void CChatServerDoc::SendMsg(CClientSocket\* pSocket, CMsg\* pMsg)

|  |  |
| --- | --- |
| {        } | TRY  {  //调用 CClientSocket 的函数发送消息  pSocket->SendMsg(pMsg);  }  CATCH(CFileException, e)  {  CString strTemp;  strTemp.Format("无法发送数据！");  UpdateMessageView(strTemp);  }  END\_CATCH |

为函数 IsUsedName()编写如下代码：

|  |
| --- |
| BOOL CChatServerDoc::IsUsedName(CString sNickName)  {  CString tempStr, tempList, sName; tempStr = "";  tempList = m\_ChattersList;  do  {  //利用“ ：” 得到名字并查找（客户端发送的消息前面都有用户名加“ ：”）  sName = tempList.Left(tempList.Find(":", 0)); tempList = tempList.Mid(tempList.Find(":", 0) + 1);  //找到则返回真  if(sName == sNickName)  return TRUE;  }while(tempList.Find(":" , 0) != -1);  //未被使用则返回假  return FALSE;  } |

为函数 AssembleMsg()编写如下代码：

|  |
| --- |
| CMsg\* CChatServerDoc::AssembleMsg(CClientSocket\* pSocket , int nCode)  {  static CMsg msg; msg.Init();  for (POSITION pos1 = m\_msgList.FindIndex(pSocket->m\_nMsgCount); pos1 != NULL;)  {  //信息内容 |

|  |  |
| --- | --- |
| } | msg.m\_strText = m\_msgList.GetNext(pos1);  //信息类型  msg.code = nCode;  }  pSocket->m\_nMsgCount = m\_msgList.GetCount(); return &msg; |

为函数 SendForNewcomer()编写如下代码：

void CChatServerDoc::SendForNewcomer(CMsg\* pMsg)

{

for(POSITION pos = m\_connectionList.GetHeadPosition(); pos != NULL;)

{

//对每个客户端都发送信息

CClientSocket\* pSocket = (CClientSocket\*)m\_connectionList.GetNext(pos); if (pSocket != NULL)

SendMsg(pSocket, pMsg);

}

}

为函数 SendToAllClients()编写如下代码：

void CChatServerDoc::SendToAllClients(int nCode)

{

for(POSITION pos = m\_connectionList.GetHeadPosition(); pos != NULL;)

{

//得到每个客户端

CClientSocket\* pSocket = (CClientSocket\*)m\_connectionList.GetNext(pos);

//封装信息

CMsg\* pMsg = AssembleMsg(pSocket, nCode);

//发送信息

if (pMsg != NULL)

SendMsg(pSocket, pMsg);

}

}

为函数 UpdateMessageView()编写如下代码：

void CChatServerDoc::UpdateMessageView(LPCTSTR lpszMessage)

{

for(POSITION pos=GetFirstViewPosition();pos!=NULL;)

{

CView\* pView = GetNextView(pos);

CChatView\* pChatView = DYNAMIC\_DOWNCAST(CChatView, pView); if (pChatView != NULL)

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | pChatView->ShowMessage(lpszMessage); |
| } | } |  |  |

为函数 UpdateChattersListView()编写如下代码：

|  |
| --- |
| void CChatServerDoc::UpdateChattersListView(CString sName , CClientSocket\* pSocket) {  CString sIPAddress;  UINT iPort;  //得到 IP 地址和端口号  pSocket->GetPeerName(sIPAddress , iPort); for(POSITION pos=GetFirstViewPosition();pos!=NULL;)  {  CView\* pView = GetNextView(pos);  CChattersListView\* pChattersListView = DYNAMIC\_DOWNCAST(CChattersListView, pView);  //更新用户列表  if (pChattersListView != NULL)  pChattersListView->AddChatter(sName , sIPAddress , iPort);  }  } |

为函数 CloseSocket()添加如下代码：

|  |
| --- |
| void CChatServerDoc::CloseSocket(CClientSocket\* pSocket)  {  //关闭连接  pSocket->Close(); POSITION pos,temp;  for(pos = m\_connectionList.GetHeadPosition(); pos != NULL;)  {  temp = pos;  CClientSocket\* pSock = (CClientSocket\*)m\_connectionList.GetNext(pos);  //将此 socket 从连接链表中删除  if (pSock == pSocket)  {  m\_connectionList.RemoveAt(temp);  break;  }  }  delete pSocket;  //如果没有连接则清空连接链表  if(m\_connectionList.GetCount() == 0)  m\_msgList.RemoveAll(); |

}

为函数 DeleteChatter()编写如下代码：

|  |
| --- |
| void CChatServerDoc::DeleteChatter(CMsg\* pMsg)  {  //得到要删除的用户名  CString sNickName = pMsg->m\_strText.Left(pMsg->m\_strText.Find(":", 0)); POSITION pos;  for(pos=GetFirstViewPosition();pos!=NULL;)  {  CView\* pView = GetNextView(pos);  CChattersListView\* pChattersListView = DYNAMIC\_DOWNCAST(CChattersListView, pView);  //在用户列表视图中将其删除  if (pChattersListView != NULL)  pChattersListView->DeleteChatter(sNickName);  }  CString tempStr, sName, tempList; tempStr = "";  tempList = m\_ChattersList;  //得到除去此用户名之外的所有用户名 do  {  sName = tempList.Left(tempList.Find(":", 0)); tempList = tempList.Mid(tempList.Find(":", 0) + 1); if(sName != sNickName) tempStr += sName + ":";  }while(tempList.Find(":" , 0) != -1); m\_ChattersList = tempStr;  //加入到信息链表和信息内容中  //接下来要对各客户端发送这些用户名，以使各客户端更新用户列表  pMsg->m\_strText = m\_ChattersList; m\_msgList.AddTail(m\_ChattersList);  } |

为 DeleteContents()函数编写如下代码：

|  |
| --- |
| void CChatServerDoc::DeleteContents()  {  if(m\_pSocket == (CListeningSocket\*)NULL)  return; //删除监听 socket  delete m\_pSocket; m\_pSocket = NULL; |

|  |  |
| --- | --- |
| } | CString temp;  temp.Format("服务器已关闭");  m\_msgList.AddTail(temp);  //对当前连接的每个客户端发送信息后关闭与该客户端的连接  while(!m\_connectionList.IsEmpty())  {  CClientSocket\* pSocket = (CClientSocket\*)m\_connectionList.RemoveHead(); CMsg\* pMsg = AssembleMsg(pSocket , NORMAL\_MESSAGE); pMsg->m\_bClose = TRUE; SendMsg(pSocket, pMsg); pSocket->ShutDown(2); delete pSocket;  }  //清空所有信息  m\_msgList.RemoveAll();  CDocument::DeleteContents(); |

至此，服务器端代码已经编写完成。

技巧：检测#if 语句和#endif 语句是否匹配将光标定位到想要匹配的#if 语句或#endif 语句，利用快捷键“ Ctrl+K”，则光标就会自动定位到与之匹配的#endif 语句或#if 语句。如果没有与之匹配的语句，光标则不移动。

技巧：检测括号是否匹配将光标移动到需要检测的括号（如大括号{}、方括号[]、圆括号()和尖括号<>等）前面，键入快捷键“ Ctrl＋]”。如果括号匹配正确，则光标跳到匹配的括号处；否则光标不移动，并且机箱喇叭还会发出一声警告声。

### 11.4.3 客户端程序设计

客户端的设计与服务器端很类似。

* 使用编写服务器端程序时封装的消息类。
* 界面由 3 个窗口组成，分别用于显示聊天内容，显示用户列表以及输入要发送的消息。
* 用户登录或断开时，都要首先以某种信息类型发送用户名，然后发送正式的消息。收到服务器的消息时，如果是普通消息，则显示出来；如果是用户登录或断开的消息，则更新用户列表。

由于客户端程序的编写和服务器端在很大程度上相同，因此这里不再详细叙述过程，而是在简要说明的基础上把主要的代码列出来。

1．创建工程

同服务器端的情况一样，创建一个 MFC 工程，取名为 ChatClient。将前面编写好的封装消息的类的文件“ Msg.h” 和“ Msg.cpp” 拷贝到 ChatClient 工程所在目录，然后加入到工程中。将文件“ Msg.cpp” 中的语句#include "ChatServer.h"改为#include "ChatClient.h"。

2．编写界面

首先新建一个类 CMessageView，它负责聊天信息的显示，其基类为 CEditView。重载此类的 PreCreateWindow()函数 virtual BOOL PreCreateWindow(CREATESTRUCT& cs)，代码如下：

|  |
| --- |
| BOOL CMessageView::PreCreateWindow(CREATESTRUCT& cs)  {  BOOL ret = CEditView::PreCreateWindow(cs); cs.style = AFX\_WS\_DEFAULT\_VIEW |  WS\_VSCROLL |  ES\_AUTOVSCROLL |  ES\_MULTILINE | ES\_NOHIDESEL;  return ret;  } |

再为此类增加一个显示聊天信息的函数 void ShowMessage(LPCTSTR lpszMessage)，代码

如下：

|  |
| --- |
| void CMessageView::ShowMessage(LPCTSTR lpszMessage)  {  CString strTemp = lpszMessage; strTemp += \_T("\r\n"); int len = GetWindowTextLength(); GetEditCtrl().SetSel(len,len);  GetEditCtrl().ReplaceSel(strTemp);  } |

接着新建一个类 CChattersView，用于用户列表的显示，它的基类是 CTreeView。为其添加两个成员函数 void AddToChattersList(CString sName)和 void ClearChattersList()，分别用于

插入一个新的表项和清除所有表项。代码如下：

|  |
| --- |
| void CChattersView::ClearChattersList()  {  GetTreeCtrl().DeleteAllItems();  }  void CChattersView::AddToChattersList(CString sName) {  GetTreeCtrl().InsertItem(sName, 0, 0, TVI\_ROOT, TVI\_LAST); |

}

接下来新建类 CInputView，用于输入聊天信息，它的基类是 CEditView。为其添加成员

函数 virtual BOOL PreCreateWindow(CREATESTRUCT& cs)，用于设定窗口风格，代码如下：

|  |
| --- |
| BOOL CInputView::PreCreateWindow(CREATESTRUCT& cs)  {  BOOL ret = CEditView::PreCreateWindow(cs); cs.style = AFX\_WS\_DEFAULT\_VIEW |  WS\_VSCROLL |  ES\_AUTOVSCROLL |  ES\_MULTILINE | ES\_NOHIDESEL;  return ret;  } |

输入完聊天信息，键入回车键后，应当发送消息。为此，要为消息 WM\_CHAR 编写响

应函数，代码如下：

|  |
| --- |
| void CInputView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)  {  // TODO: Add your message handler code here and/or call default if (nChar != VK\_RETURN)  {  CEditView::OnChar(nChar, nRepCnt, nFlags);  return;  }  else  {  CChatClientDoc\* pDoc = (CChatClientDoc\*)m\_pDocument;  CString strText;  GetEditCtrl().GetWindowText(strText);  //作为普通消息发送  pDoc->SendMsg(strText, 4, true); strText=\_T("");  GetEditCtrl().SetWindowText(strText);  GetEditCtrl().SetSel(9,9,FALSE);  }  CEditView::OnChar(nChar, nRepCnt, nFlags);  } |

这里的 SendMsg()函数是 CChatClientDoc 的一个成员函数，在后面的程序中会看到它的

实现。函数中用到了 CChatClientDoc 类，在“InputView.cpp”的头部应加入语句#include

"ChatClientDoc.h"。

该界面的主框架分为 3 部分。为类 CMainFrame 添加两个用于分割的成员变量 CSplitterWnd m\_wndSplitter1 和 CSplitterWnd m\_wndSplitter2，添加重载函数 virtual BOOL

OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext\* pContext)，代码如下：

|  |
| --- |
| BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT /\*lpcs\*/, CCreateContext\* pContext)  {  //将窗口分成 1 行 2 列  if (!m\_wndSplitter1.CreateStatic(this, 1, 2))  return FALSE;  //将其中一列分成 2 行 1 列  if (!m\_wndSplitter2.CreateStatic(&m\_wndSplitter1, 2, 1,  WS\_CHILD | WS\_VISIBLE | WS\_BORDER, m\_wndSplitter1.IdFromRowCol(0, 0)))  {  TRACE0("Failed to create nested splitter\n");  return FALSE;  }  //第 1 行第 1 列作为显示消息的窗口  if(!m\_wndSplitter2.CreateView(0,0, RUNTIME\_CLASS(CMessageView), CSize(100,100), pContext))  {  m\_wndSplitter1.DestroyWindow(); m\_wndSplitter2.DestroyWindow(); return FALSE;  }  //第 2 行第 1 列作为输入消息窗口  if(!m\_wndSplitter2.CreateView(1,0, RUNTIME\_CLASS(CInputView), CSize(100,100), pContext))  {  m\_wndSplitter1.DestroyWindow(); m\_wndSplitter2.DestroyWindow(); return FALSE;  }  //第 1 行第 2 列作为用户列表显示窗口  if(!m\_wndSplitter1.CreateView(0, 1, RUNTIME\_CLASS(CChattersView), CSize(100, 100), pContext))  {  m\_wndSplitter1.DestroyWindow(); return FALSE;  }  m\_wndSplitter1.SetColumnInfo(0, 450,50); m\_wndSplitter2.SetRowInfo(0, 250, 50); return TRUE;  } |

至此，界面编写完毕。

3．创建 CChatSocket 类

新建一个 CChatSocket 类，用于客户端和服务器进行数据传输，其基类是 CSocket。在类的 定 义 中 声 明 宏 DECLARE\_DYNAMIC(CChatSocket) ， 在 类 的 实 现 中 声 明 宏

IMPLEMENT\_DYNAMIC(CChatSocket, CSocket)。

另外还要在该类中加入成员变量 CChatClientDoc\* m\_pDoc，它是与 CChatSocket 关联的文档类，很多数据处理工作在文档类中完成。

为该类增加一个新的构造函数 CChatSocket(CChatClientDoc\* pDoc)，代码如下：

CChatSocket::CChatSocket(CChatClientDoc\* pDoc)

{

m\_pDoc = pDoc;

}

重载 OnReceive()函数 virtual void OnReceive(int nErrorCode)，代码如下：

void CChatSocket::OnReceive(int nErrorCode)

{

//调用 CSocket 的 OnReceive 函数

CSocket::OnReceive(nErrorCode);

//其余工作交给文档类处理

m\_pDoc->ProcessReceive();

}

其中，ProcessReceive()函数是 CChatClientDoc 类的一个成员函数，在后面将会介绍它的

具体实现。

最后，在文件“ ChatSocket.h” 的类定义外面加入类声明 class CChatClientDoc，在文件

“ ChatSocket.cpp” 的头部加入语句#include "ChatClientDoc.h"。至此，CchatSocket 类创建完毕。

4．创建登录对话框通过登录对话框，用户可以登录到服务器，该对话框如图 11-11 所示。其中，对话框的

ID 为 IDD\_LOGDLG，3 个 Edit 控件的 ID 依次为 IDC\_SERVER、IDC\_PORT 和 IDC\_NAME。为对话框新建一个类 CLogDlg，对应 3 个控件，为其添加 3 个成员变量，即 CString m\_Server、 int m\_Port 和 CString m\_Name，并将 m\_Port 初始化为 2000。



图 11-11 登录对话框

然后，在菜单栏中为其新建一个菜单项，当单击该菜单项时，就会弹出登录对话框。为了方便，可以再建一菜单项，用于断开连接，如图 11-12 所示。

图 11-12 登录对话框对应菜单栏

其中，连接菜单项的 ID 为 ID\_CONNECT，断开菜单项的 ID 为 ID\_DISCONNECT。

5．实现 CChatClientDoc 类

同服务器端的实现一样，客户端大量的工作都是在 CChatClientDoc 类中实现的。在开始编写代码前，首先在文件“ ChatClientDoc.h” 中 CChatClientDoc 类定义的前面加入如下的类声明语句：

class CChatSocket; class CMsg;

在文件“ ChatClientDoc.cpp” 的开头加入下面的 include 语句：

#include "Msg.h"

#include "ChatSocket.h"

#include "MessageView.h"

#include "ChattersView.h"

#include "LogDlg.h"

与服务器端一样，在文件“ ChatServerDoc.cpp” 的开头加入下面的定义：

#define LEAVING\_CHAT 1

#define SENDING\_CHATTERS\_LIST 2

#define SENDING\_NICKNAME 3

#define NORMAL\_MESSAGE 4

#define USED\_NAME 5

接下来，为类 CChatClientDoc 添加成员变量，如表 11-11 所示。表 11-11 CChatClientDoc 类成员变量及其说明

|  |  |
| --- | --- |
| 成员变量 | 说明 |
| CString m\_strName | 用户名 |
| CChatSocket\* m\_pSocket | 用于连接的 Socket |
| CSocketFile\* m\_pFile | 连接 CSocket 和 CArchive |
| CArchive\* m\_pArchiveIn | 输入 |
| CArchive\* m\_pArchiveOut | 输出 |
| BOOL bIsConnected | 是否已经连接 |

为 CChatClientDoc 类添加成员函数，如表 11-12 所示。表 11-12 CChatClientDoc 类成员函数及其说明

|  |  |  |
| --- | --- | --- |
| 成员函数 |  | 说明 |
| BOOL ConnectSocket(LPCTSTR  LPCTSTR lpszAddress, int nPort) | lpszHandle, | 连接服务器，3 个参数依次是用户名、服务器地址和端口号 |

续表

|  |  |
| --- | --- |
| 成员函数 | 说明 |
| void ProcessReceive() | 当接收到信息时调用此函数，进行信息接收等操作 |
| void SendMsg(CString& strText, int mCode, BOOL bSendName) | 发送信息，3 个参数分别表示信息内容、信息种类和是否发送用户名 |
| void ReceiveMsg() | 接收信息 |
| void DisplayMsg(LPCTSTR lpszText) | 显示信息 |
| void UpdateChattersList(CMsg\* pMsg) | 更新用户列表 |
| virtual void DeleteContents() | 断开时所作的工作 |

下面开始为各函数编写代码。首先在构造函数中将各成员变量初始化，代码如下：

CChatClientDoc::CChatClientDoc()

{

// TODO: add one-time construction code here bIsConnected = FALSE; m\_pSocket = NULL; m\_pFile = NULL; m\_pArchiveOut = NULL; m\_pArchiveIn = NULL;

}

接着为“ 连接” 菜单项编写响应函数，代码如下：

|  |
| --- |
| void CChatClientDoc::OnConnect()  {  // TODO: Add your command handler code here CLogDlg dlg;  if(dlg.DoModal() == IDOK)  {  if(ConnectSocket(dlg.m\_Name, dlg.m\_Server, dlg.m\_Port))  bIsConnected = TRUE;  else  AfxMessageBox("连接服务器失败！");    }  } |

当连接后，应该使“ 连接” 菜单项无效，为消息 UPDATA\_COMMAND\_UI 添加响应函

数如下：

void CChatClientDoc::OnUpdateConnect(CCmdUI\* pCmdUI)

{

// TODO: Add your command update UI handler code here pCmdUI->Enable(!bIsConnected);

}

为“ 断开” 菜单项添加响应函数，代码如下：

void CChatClientDoc::OnDisconnect()

{

// TODO: Add your command handler code here DeleteContents(); bIsConnected = FALSE;

}

当断开后，应当使“ 断开” 菜单项无效，为消息 UPDATA\_COMMAND\_UI 添加如下响

应函数：

void CChatClientDoc::OnUpdateDisconnect(CCmdUI\* pCmdUI)

{

// TODO: Add your command update UI handler code here pCmdUI->Enable(bIsConnected);

}

为 ConnectSocket()函数编写如下代码：

|  |
| --- |
| BOOL CChatClientDoc::ConnectSocket(LPCTSTR lpszHandle, LPCTSTR lpszAddress, int nPort)  {  m\_strName = lpszHandle; m\_pSocket = new CChatSocket(this);  //创建 socket 失败  if (!m\_pSocket->Create())  {  delete m\_pSocket; m\_pSocket = NULL;  AfxMessageBox("创建 socket 失败！"); return FALSE;  } //连接失败  if(!m\_pSocket->Connect(lpszAddress, nPort))  {  delete m\_pSocket; m\_pSocket = NULL; return FALSE;  }  m\_pFile = new CSocketFile(m\_pSocket); |

//用于数据接收和发送

m\_pArchiveIn = new CArchive(m\_pFile,CArchive::load); m\_pArchiveOut = new CArchive(m\_pFile,CArchive::store);

//将用户名发送给服务器

SendMsg(m\_strName, SENDING\_NICKNAME, false);

|  |  |
| --- | --- |
| } | CString strTemp;  strTemp.Format("进入聊天室");  //作为普通消息发送给服务器  SendMsg(strTemp, NORMAL\_MESSAGE, true); return TRUE; |

为 SendMsg()函数编写如下代码：

|  |
| --- |
| void CChatClientDoc::SendMsg(CString& strText, int mCode, BOOL bSendHandle)  {  if (m\_pArchiveOut != NULL)  {  CMsg msg;  msg.code = mCode;  msg.m\_strText = (bSendHandle ? m\_strName + \_T(": ") + strText : strText);  TRY  {  msg.Serialize(\*m\_pArchiveOut); m\_pArchiveOut->Flush();  }  CATCH(CFileException, e)  {  m\_pArchiveOut->Abort(); delete m\_pArchiveOut; m\_pArchiveOut = NULL; CString strTemp; strTemp.Format("发送失败");  DisplayMsg(strTemp);  }  END\_CATCH  }  } |

为 ProcessReceive()函数编写如下代码：

void CChatClientDoc::ProcessReceive()

{

do {

ReceiveMsg(); if (m\_pSocket == NULL)

return;

|  |  |
| --- | --- |
| } | }  while(!m\_pArchiveIn->IsBufferEmpty()); |

为 ReceiveMsg()函数编写如下代码：

|  |
| --- |
| void CChatClientDoc::ReceiveMsg()  {  CMsg msg;  TRY  {  msg.Serialize(\*m\_pArchiveIn);  if(msg.code == SENDING\_CHATTERS\_LIST)  {  //更新用户列表  UpdateChattersList(&msg);  return;  }  if(msg.code == USED\_NAME)  {  //如果是用户名已存在信息则关闭连接，退出  AfxMessageBox(msg.m\_strText); msg.m\_bClose = TRUE; OnDisconnect(); return;  }  //显示消息  DisplayMsg(msg.m\_strText);  }  CATCH(CFileException, e)  {  msg.m\_bClose = TRUE; m\_pArchiveOut->Abort(); CString strTemp; strTemp.Format("接收数据失败");  DisplayMsg(strTemp);  } |

END\_CATCH //如果连接已关闭则删除各对象

if (msg.m\_bClose)

{

delete m\_pArchiveIn;

|  |  |  |
| --- | --- | --- |
|  |  | m\_pArchiveIn = NULL; delete m\_pArchiveOut; m\_pArchiveOut = NULL; delete m\_pFile; m\_pFile = NULL; delete m\_pSocket; m\_pSocket = NULL; |
| } | } |  |

为 DisplayMsg()函数编写如下代码：

void CChatClientDoc::DisplayMsg(LPCTSTR lpszText)

{

for(POSITION pos=GetFirstViewPosition();pos!=NULL;)

{

CView\* pView = GetNextView(pos);

CMessageView\* pChatView = DYNAMIC\_DOWNCAST(CMessageView, pView); if (pChatView != NULL) pChatView->ShowMessage(lpszText);

}

}

为 UpdateChattersList()函数编写如下代码：

|  |
| --- |
| void CChatClientDoc::UpdateChattersList(CMsg\* pMsg) {  CChattersView\* pChattersView;  for(POSITION pos=GetFirstViewPosition();pos!=NULL;)  {  CView\* pView = GetNextView(pos);  pChattersView = DYNAMIC\_DOWNCAST(CChattersView, pView);  //首先清空用户列表视图  if (pChattersView != NULL) pChattersView->ClearChattersList();  }  CString strTemp = pMsg->m\_strText;  //得到所有的用户名并加入到用户列表中 do |

{

CString sName = strTemp.Left(strTemp.Find(":",0)); pChattersView->AddToChattersList(sName); strTemp = strTemp.Mid(strTemp.Find(":" , 0)+1);

}while(strTemp.Find(":",0) != -1);

}

为 DeleteContents()函数编写如下代码：

|  |
| --- |
| void CChatClientDoc::DeleteContents()  {  if ((m\_pSocket != NULL) && (m\_pFile != NULL) && (m\_pArchiveOut != NULL))  {  //首先发送用户名  SendMsg(m\_strName, LEAVING\_CHAT, false);  CMsg msg;  CString strTemp;  //发送普通消息  strTemp.Format(":离开了聊天室"); msg.code = NORMAL\_MESSAGE; msg.m\_bClose = TRUE;  msg.m\_strText = m\_strName + strTemp; msg.Serialize(\*m\_pArchiveOut); m\_pArchiveOut->Flush();  }  //删除各对象  delete m\_pArchiveOut; m\_pArchiveOut = NULL; delete m\_pArchiveIn; m\_pArchiveIn = NULL; delete m\_pFile; m\_pFile = NULL;  //关闭连接  if (m\_pSocket != NULL)  {  m\_pSocket->ShutDown(2); delete m\_pSocket; m\_pSocket = NULL;  } //更新视图  for(POSITION pos=GetFirstViewPosition();pos!=NULL;)  {  CView\* pView = GetNextView(pos);  if (pView->IsKindOf(RUNTIME\_CLASS(CMessageView)))  {  CMessageView\* pChatView = (CMessageView\*)pView; pChatView->GetEditCtrl().SetWindowText(\_T("")); |

|  |  |
| --- | --- |
| } | }  if (pView->IsKindOf(RUNTIME\_CLASS(CChattersView)))  {  CChattersView\* pCChattersView = (CChattersView\*)pView; pCChattersView->ClearChattersList();  }  }  CDocument::DeleteContents(); |

至此，整个程序编写完毕。分别运行服务器端程序和客户端程序，效果如图11-13~图11-15 所示。

图 11-13 服务器端界面 图 11-14 客户端界面 1

本程序实现了一个简单的基于服务器/客户端模型的网络聊天系统，系统功能虽然简单，但是体现了一般网络编程的基本步骤。在本程序中，由于对信息进行了封装，且采用了文档视图结构，用户可以很容易扩展其功能，比如可以加入两人私聊（其他人看不到聊天内容）、服务器广播等功能。由于篇幅所限，这些功能请读者自行实现。

## 11.5 利用 MSComm 控件进行串口编程

计算机串口编程在网络通信中也有着十分广泛的应用，比如电话、传真、视频控制等。

本节将介绍如何在 Visual C++中利用 MSComm 控件进行串口编程。

### 11.5.1 MSComm 控件介绍

1．MSComm 控件处理通信的方式

MSComm 控件为应用程序提供了读写串口功能，它提供了两种处理通信的方法。

* 事件驱动方式。在这种情况下，使用 MSComm 控件的 OnComm 事件可以捕获和处理这些通信事件。同时，OnComm 也可以捕获和处理通信中的错误。
* 查询方式。用户可以在每个重要的程序功能之后检查 CommEvent 属性的值来检测事件和通信错误。在通信频繁的情况下，这种方式更为方便。

2．MSComm 控件常用属性

1. CommPort 属性

该属性用于设置或返回通信端口号，端口号的范围是 1～16，默认为 1。对应的函数为 void SetCommPort(short nNewValue)和 short GetCommPort()。

1. Settings 属性

该属性用于设置或返回数据传输率、奇偶校验、数据位和停止位参数，其属性值格式为

“ BBBB,P,D,S”。

其中 BBBB 为数据传输率，P 为奇偶校验，D 为数据位，S 为停止位。默认值为

“ 9600,N,8,1”。

该属性对应的函数为 void SetSettings(LPCTSTR lpszNewValue)和 String GetSettings()。

1. PortOpen 属性

该属性用于设置或返回串口状态，值为 TRUE 时打开，值为 FALSE 时关闭。如果要打开端口，则必须现设置 CommPort 属性。

该属性对应的函数为 void SetPortOpen(BOOL bNewValue)和 BOOL GetPortOpen()。

1. Input 属性

该属性用于从接收缓冲区中读取数据，其对应的函数为 VARIANT GetInput()。

1. InputMode 属性

该属性用于设置或返回从接收缓冲区读取数据的格式，属性值为 0 时表示以文本格式读取数据；属性之为 1 时表示以二进制格式读取数据；属性值为 2 时表示以十六进制格式读取数据。

该属性对应的函数分别为 void SetInputMode(long nNewValue)和 long GetInputMode()。

1. InBufferCount 属性

该属性用于设置或返回在接收缓冲区中等待的字符数，该属性在端口未打开时不可用。对应的函数为 void SetInBufferCount(short nNewValue)和 short GetInBufferCount()。

1. InBufferSize 属性

该属性用于设置或返回接收缓冲区大小的字节数，其对应的函数分别为 void

SetInBufferSize(short nNewValue)和 short GetInBufferSize()。

1. Output 属性该属性用于向发送缓冲区中写入数据，其对应的函数为 void SetOutput(const VARIANT& newValue)。
2. OutBufferCount 属性

该属性用于设置或返回在发送缓冲区中等待的字符数，该属性在端口未打开时不可用。其对应的函数分别为 void SetOutBufferCount(short nNewValue)和 short GetOutBufferCount()。

1. OutBufferSize 属性

该属性用于设置或返回发送缓冲区的字节大小，对应的函数分别为 void

SetOutBufferSize(short nNewValue)和 short GetOutBufferSize()。

1. CommEvent 属性该属性用于设置或返回串口事件，其对应的函数分别为 void SetCommEvent(short nNewValue)和 short GetCommEvent()。常见的串口事件如表 11-13 所示。

表 11-13 常见串口事件

|  |  |  |  |
| --- | --- | --- | --- |
| 常数 |  | 取值 | 描述 |
| ComEvSend | 1 |  | 发送事件 |
| ComEvReceive | 2 |  | 接收事件 |
| ComEvCTS | 3 |  | clear-to-send 线的状态发生变化 |
| ComEvDSR | 4 |  | data-set ready 线的状态发生变化 |
| ComEvCD | 5 |  | carrier detect 线的状态发生变化 |
| ComEvRing | 6 |  | 检测到震铃信号 |
| ComEvEOF | 7 |  | 收到文件结束字符 |

1. RThreshold 属性

该属性用于设置或返回 RThreshold 数值，所谓 RThreshold 数值是通过 MSComm 控件将 CommEvent 属性设置为 ComEvReceive，并在产生 OnComm 事件之前所接收的字符数。该属性对应的函数为 void SetRThreshold(short nNewValue)和 short GetRThreshold()。

### 11.5.2 利用 MSComm 控件编程的基本流程

利用 MSComm 控件编程的基本流程如下。

* 在当前 Project 中插入 MSComm 控件；
* 初始化并打开串口；
* 捕捉串口事项；
* 串口读写；
* 关闭串口。

### 11.5.3 利用 MSComm 进行串口通信

实例 11-2：MSComm 控件使用实例。源代码在光盘中“ \11\实例 11-2\MSComm” 目录下。

本节通过编写一个小程序来实现利用 MSComm 控件进行串口数据传输。

新建一个基于对话框的 MFC 工程，取名为 MSComm。依次单击菜单栏的“ Project” →

“ Add To Project” →“ Components and Controls”，弹出添加控件对话框，如图 11-16 所示。然后打开“ Registered ActiveX Controls”文件夹，选中“ Microsoft Communications Control，vision

6.0” 控件，将其添加到工程中，如图 11-17 所示。

图 11-16 添加控件对话框 图 11-17 加入 MSComm 控件

然后编辑对话框界面，完成后的效果如图 11-18 所示（本例中，用于选择串口的控件为 Combo Box 控件，在为对话框添加 Combo Box 控件的时候，需要为该 Combo Box 控件加入

1 和 2 两个数据，分别表示串口 1 和串口 2），其中控件就是 MSComm 控件。

注意：在为 Combo Box 控件添加数据的时候，如果数据多于一个，则在添加后面数据的时候，要按住 Ctrl 键，同时按回车键进行换行，而不是直接用回车键进行换行。

 图 11-18 对话框界面

为对话框添加完控件后，需要设置各控件 ID，并为其添加变量。各控件的 ID 及其对应的变量如表 11-14 所示。

表 11-14 各控件 ID 及对应变量

|  |  |  |
| --- | --- | --- |
| 控件 | ID | 变量 |
| MSComm 控件 | IDC\_MSCOMM | CMSComm m\_Comm |
| 接收数据 Edit | IDC\_RECEIVEDATA | CString m\_Receive |
| 发送数据 Edit | IDC\_SENDDATA | CString m\_Send |
| 选择串口下拉列表 | IDC\_COMBOLIST | int m\_Port（初始化为 0） |
| 打开串口按钮 | IDC\_OPENPORT | 无 |
| 关闭串口按钮 | IDC\_CLOSTPORT | 无 |
| 发送按钮 | IDC\_SEND | 无 |

在文件“ MSCommDlg.h” 的头部加入语句#include "mscomm1.h"，然后为对话框类加入

成员函数 void InitComm()，来初始化 MSComm 控件，其代码如下：

void CMSCommDlg::InitComm()

{

//设置串口

m\_Comm.SetCommPort(m\_Port+1);

// 设置数据读取格式为二进制方式

m\_Comm.SetInputMode(1);

// 设置传输参数

m\_Comm.SetSettings("9600,n,8,1");

//缓冲区内有一个字符就可以接收

m\_Comm.SetRThreshold(1);

// 指定接收缓冲区大小

m\_Comm.SetInBufferSize(1024);

//指定发送缓冲区大小

m\_Comm.SetOutBufferSize(1024);

// 清空接收缓冲区

m\_Comm.SetInBufferCount(0);

// 打开串口

if(!m\_Comm.GetPortOpen())

{

m\_Comm.SetPortOpen(TRUE);

}

}

当选择下拉列表中的端口时，会激发 CBN\_SELCHANGE 消息，为其添加响应函数，代

码如下：

void CMSCommDlg::OnSelchangeCombolist()

{

// TODO: Add your control notification handler code here

CString m\_str;

//得到下拉列表内容

GetDlgItemText(IDC\_COMBOLIST, m\_str);

|  |  |
| --- | --- |
| } | //得到串口号  m\_Port = atoi(LPCTSTR(m\_str))-1; |

为“ 打开串口” 按钮添加响应函数，代码如下：

void CMSCommDlg::OnOpenport()

{

// TODO: Add your control notification handler code here

InitComm();

}

为“ 关闭串口” 按钮添加响应函数，代码如下：

void CMSCommDlg::OnClostport()

{

// TODO: Add your control notification handler code here if( m\_Comm.GetPortOpen() )

{

// 关闭串口

m\_Comm.SetPortOpen(0);

}

}

为“ 发送” 按钮添加响应函数，代码如下：

|  |
| --- |
| void CMSCommDlg::OnSend()  {  // TODO: Add your control notification handler code here  //得到发送 Edit 控件的数据  UpdateData(TRUE); int Count=m\_Send.GetLength();  //将发送数据传给 m\_Array  CByteArray m\_Array; m\_Array.RemoveAll(); m\_Array.SetSize(Count);  for(int i=0;i<Count;i++)  m\_Array.SetAt(i,m\_Send[i]);  //发送  m\_Comm.SetOutput(COleVariant(m\_Array)); m\_Send = "";  UpdateData(FALSE);  } |

对于数据的接收，应该利用 MSComm 控件的 OnComm 事件来处理。为其添加响应函数，

代码如下：

void CMSCommDlg::OnOnCommMscomm()

|  |
| --- |
| {  // TODO: Add your control notification handler code here int nEvent; VARIANT m\_input;  char \*str, \*str1; int k,i;  CString str2;  //得到事件  nEvent = m\_Comm.GetCommEvent(); switch(nEvent)  {  //只对接收数据事件进行处理  case 2:  // 接收缓冲区的字符数目  k = m\_Comm.GetInBufferCount(); if(k > 0)  {  //得到接收数据  m\_input = m\_Comm.GetInput();  str = (char\*)(unsigned char\*)m\_input.parray->pvData;  }  //清除字符串中的不必要字符  i = 0;  str1 = str; while(i < k)  {  i++;  str1++;  }  \*str1 =’\0’;  str2 = (const char\*)str;  //得到最终数据  m\_Receive += (LPCTSTR)str2; default:  break;  }  // 显示数据  UpdateData(FALSE);  } |

至此，程序设计完成。运行两个程序，分别选择串口 1 和串口 2，然后收发数据，如图 11-19 和图 11-20 所示。

  图 11-19 串口发送数据 图 11-20 串口接收数据

## 11.6 本章小结

本章介绍了网络编程的基本知识及基本步骤，并且详细介绍了 WinSock 类。接着利用所讲知识编程实现了一个网络聊天系统，最后利用 MSComm 控件实现了简单的串口通信。相信通过本章的学习，读者可以编写简单的网络应用程序，并会利用 MSComm 控件进行简单的串口数据传输。

# 第 12 章 数据库开发

随着计算机技术的飞速发展，数据库的应用也越来越广泛。如今，数据库技术已经成为计算机技术中的核心技术之一。本章首先简单介绍数据库的基本概念及用 Visual C++ 6.0 进行数据库开发的几种相关技术，然后介绍 Access 数据库的建立和一些基本的 SQL 语句，最后通过一个具体的实例来详细说明 ADO 技术在数据库开发中的应用。本章主要包括以下几方面的内容：

* 数据库的基本概念及数据库的相关技术；
* Access 数据库的建立；
* 基本的 SQL 语句；
* 如何利用 ADO 技术进行数据库开发。

## 12.1 概述

数据库技术产生于 20 世纪 60 年代中期，是数据管理的最新技术，是计算机科学的重要分支，它的出现极大的促进了计算机应用向各行业的渗透。本节将对数据库的概念，数据库的相关技术以及如何开发数据库应用程序作一简单介绍。

### 12.1.1 数据库的基本概念

所谓数据库（Database，简称 DB），就是长期存储在计算机内的、有组织的、可共享的数据集合。数据库中的数据按一定的数据模型组织、描述和存储，具有较小的冗余度，较高的数据独立性和易扩展性，并可为各种用户共享。

有了数据和数据库，还要进行数据库的管理，不仅可以科学地组织这些数据并将其存储在数据库中，而且还能高效地处理这些数据。数据库管理系统（Database Management System，简称 DBMS）是位于用户和操作系统之间的一层数据管理软件。数据库在建立、运用和维护时由数据库管理系统统一管理、统一控制。数据库管理系统使用户能方便的定义数据和操纵数据，并能够保证数据的安全性、完整性、多用户对数据并发使用以及发生故障后的系统恢复。

在很多情况下，人们所说的数据库实际上指的是数据库系统。数据库系统（Database System，简称 DBS）是指在计算机系统中引入数据库后的系统构成，一般由数据库、数据库管理系统（及其开发工具）、应用系统、数据库管理员和用户构成。

设计数据库的核心问题就是数据模型的选择，常用的数据模型有层次数据模型、网状数据模型、关系数据模型、实体联系数据模型和面向对象数据模型。现在常用的数据库有 SQL

Server、Oracle、DB2、Sybase、MySQL、Access、FoxPro 等。

### 12.1.2 数据库相关技术介绍

不管是功能简单的数据库，还是复杂的大型数据库系统，Visual C++都提供了一些编程接口。一般来说，编程接口主要有以下 5 种：

1．ODBC API

ODBC 的全称是 Open DataBase Connectivity，即开放数据库互连。ODBC 是为了能让同一应用程序访问不同的数据库管理系统而设计的，它实际上是一个不同类型数据库的通用接口，用这些 API 编写的应用程序独立于 DBMS，实现同一应用程序对不同 DBMS 的共享。应用程序对数据库的操作是通过和各种 DBMS 相对应的 ODBC 驱动程序（Drive）来进行的，对于不同的 DBMS，用户只需安装相应的驱动程序，而不必更改应用程序本身。ODBC 为 SQL Server、ACCESS、FoxPro 等都提供了驱动程序，使得用户可以使用 SQL 语句对数据库进行直接的底层功能操作。在使用 ODBC API 时，用户需引入的头文件为“ sql.h”、“ sqlext.h”

和“ sqltypes.h”。

作为一种数据库连接的标准技术，ODBC 主要有以下几个特点。

* ODBC 是一种使用 SQL 的程序设计接口。
* ODBC 的设计是建立在客户机/服务器体系结构基础之上的。
* ODBC 使应用程序避免了与数据源连接的复杂性。
* ODBC 的结构允许多个应用程序访问多个数据源，即应用程序与数据源的关系是多对多的。

2．MFC 的 ODBC 类

直接用 ODBC API 进行数据库开发是一件很繁琐的事情，因此在 MFC 1.5 后的版本里引入了封装 ODBC 功能的类。通过这些类提供的与 ODBC 的接口，程序员可以不必理会 ODBC

API 中的繁杂处理而进行数据库操作。主要的 MFC ODBC 类有如下 5 个。

* CDatabase 类：一个 CDatabase 对象表示到数据源的连接，通过它可以操作数据源。要用 CDatabase 类，首先要建立一个 CDatabase 对象，然后调用它的 OpenEx()函数建立和数据源的连接。当结束对数据库的操作时，调用 Close()函数关闭连接。
* CRecordSet 类：一个 CRecordSet 对象就是一个记录集，它表示一个从数据源选择的一组纪录的集合。
* CrecordView 类：提供了一个表单视图与某个记录集直接相连，利用对话框数据交换机制（DDX）在记录集与表单视图的控件之间传输数据。
* CFieldExchange 类：支持记录字段数据交换（DDX），即记录集字段数据成员与相应数据库的表的字段之间的数据交换。
* CDBException 类：当发生错误时，许多 MFC ODBC 调用并不返回错误码，而是返回一个 CDBException 对象。

3．MFC 的 DAO 类

DAO 是 Database Access Object（数据访问对象）的简称。与 MFC ODBC 类很类似，DAO 中几个主要的类分别为 CDaoDatabase、CDaoRecordSet、CDaoRecordView、CDaoFieldExchange 和 CDaoException。除了上面 5 个类以外，DAO 类还包括下面 3 个比较重要的类。

* CDaoWorkSpace：管理一个用户和数据库的会话。
* CDaoQueryDef：代表一个查询定义。
* CDaoTableDef：代表一个基本表或附加表所储存的定义，负责管理数据库中表的结构。

可以看到，DAO 的类名和 ODBC 的很相似，只是多了 D、a、o 这 3 个字母，并且所有的 DAO 的类都有前缀 Dao。

通过 DAO 访问数据库和通过 ODBC 访问数据库是不同的，利用 ODBC 访问数据库是通过各种 DBMS 相对应的 ODBC 驱动程序来实现的，而利用 DAO 访问数据库是通过微软公司的 Jet 数据库引擎来实现的。

DAO 支持以下 4 个数据库选项：打开一个 Access 数据库（mdb 文件），直接打开一个

ODBC 数据源，通过 Jet 引擎打开一个 ISAM 类型的数据源，把外部表附属到 Access 数据库。

4．MFC 的 OLE/DB

OLE/DB 是系统级的编程接口，它定义了一组 COM 接口。这组接口封装了各种数据系统的访问操作，同时也为数据使用方和数据提供方建立了标准。OLE/DB 还提供了一组标准的服务组件，用于提供查询、缓存、数据更新、事务处理等操作，因此，数据提供方只需实现一些简单的数据操作，在使用方就可以获得全部的数据控制能力。

OLE/DB 框架定义了应用的 3 个基本类：

* 数据提供程序 Data Provider：拥有自己的数据并以表格形式显示数据的应用程序。
* 使用者 Consumers：使用 OLE/DB 接口对存储在数据提供程序中的数据进行控制的应用程序。
* 服务提供程序 Service Provider：数据提供程序和使用者的结合。

OLE/DB 虽然增加了操作的灵活性，但是有的时候这种灵活会使操作变得复杂。并且对于初学者来说，OLE/DB 里面数十个密密麻麻的接口还是有些恐怖的。

5．ADO 技术

相比前面 4 种技术，ADO 技术显得方便易学，这也是本章的重点，将在本章 12.4～12.6 节详细介绍。

### 12.1.3 数据库应用程序的开发步骤

数据库应用程序就是为用户提供一个可以满足其长期需求的软件产品。成功地开发一个数据库应用程序的关键之一就是要详细地定义用户的需求，然后再围绕用户的需求进行开发。

数据库应用程序的开发一般分如下 3 个步骤。

1．系统设计

系统设计阶段应当根据用户的需求，明确地描述数据库（数据库服务器端）和应用界面（客户机端）实现的功能，即决定哪些功能由服务器端实现，哪些功能由客户端实现。然后选择合适的开发工具，比如 Visual C++、Delphi、Power Builder 等。

2．系统实现

系统实现阶段的主要任务是使用某种开发工具（本书用 Visual C++ 6.0）实现系统设计阶段的思想，并进行调试。在系统设计阶段，通常要对数据库进行备份，以免破坏原来的数据库。

3．系统运行和维护

一个应用系统性能的优劣和效率的高低应当由用户来作出判断。应用程序在运行过程中，用户会提出一些新的要求和建议，根据用户需求的变化，对原来程序进行一定的修改，使其进一步得到完善和提高。

### 12.1.4 选择数据库管理系统的原则

当今数据库管理系统多种多样，可遵循一定的原则进行选择。

1．构造数据库的难易程度

需要分析数据库管理系统有没有范式要求，即是否必须按照系统所规定的数据模型分析现实世界，建立相应的模型；数据库管理语句是否符合国际标准；有没有面向用户的易用的开发工具；所支持的数据库容量，数据库的容量特性决定了数据库管理系统的使用范围。

2．程序开发的难易程度

有无计算机辅助软件工程工具 CASE—计算机辅助软件工程工具可以帮助程序员根据软件工程的方法提供各开发阶段的维护、编码环境，便于复杂软件的开发、维护。有无第 4 代语言的开发平台—第 4 代语言具有非过程语言的设计方法，不需要编写复杂的过程性代码，易学、易懂、易维护。有无面向对象的设计平台—面向对象的设计思想与人类的逻辑思维方式十分接近，便于开发和维护。对多媒体数据类型的支持—多媒体数据需求是今后

发展的趋势，支持多媒体数据类型的数据库管理系统必将减少应用程序的开发和维护工作。

3．数据库管理系统的性能分析

包括对性能评估（响应时间、数据单位时间吞吐量）、性能监控（内外存使用情况、系统输入/输出速率、SQL 语句的执行、数据库元组控制）和性能管理（参数设定与调整）的分析。

4．对分布式应用的支持

包括数据透明与网络透明程度。数据透明是指用户在应用程序中不需要指出数据在网络中节点上的位置，数据库管理系统可以自动搜索网络，提取所需数据。网络透明是指用户在

应用中无需指出网络所采用的协议，数据库管理系统自动将数据包转换成相应的协议数据。

5．并行处理能力支持多 CPU 模式的系统（SMP、CLUSTER、MPP）、负载的分配形式、并行处理的颗粒度和范围。

6．可移植性和可扩展性

可移植性指垂直扩展和水平扩展能力。垂直扩展要求新平台能够支持低版本的平台，数据库客户机/服务器机制支持集中式管理模式，这样才能保证用户以前的投资和系统。水平扩展要求满足硬件上的扩展，支持单 CPU 模式转换成多 CPU 并行机模式（SMP、CLUSTER、

MPP）。

7．数据完整性约束

数据完整性指数据的正确性和一致性保护，包括实体完整性、参照完整性、复杂的事务规则。

8．并发控制功能

对于分布式数据库管理系统，并发控制功能是必不可少的。因为它面临的是多任务分布环境，所以可能会有多个用户点在同一时刻对同一数据进行读写操作。为保证数据的一致性，要由数据库管理系统的并发控制功能进行控制。评价并发控制的标准应从以下几方面考虑。

* 保证查询结果一致性方法；
* 数据锁的颗粒度（数据锁的控制范围，表、页、元组等）；
* 数据锁的升级管理功能；
* 死锁的检测和解决方法。

9．容错能力

异常情况下对数据进行的容错处理是以硬件的容错、有无磁盘镜像处理功能软件的容错、有无软件方法异常情况的容错功能为评价标准的。

10．安全性控制

包括安全保密的程度（账户管理、用户权限、网络安全控制、数据约束）

11．支持汉字处理能力

包括数据库描述语言的汉字处理能力（表名、域名和数据）和数据库开发工具对汉字的支持能力。

## 12.2 设计创建和维护 Access 数据库

Access 数据库是目前应用最广泛的小型数据库之一，它简单易学，非常适合初学者。本节以 Access2002 为例，详细介绍 Access 数据库的设计、创建和维护。

实例 12-1：Access 数据库实例。源代码在光盘中“ \12\实例 12-1” 目录下。

### 12.2.1 创建 Access 数据库

打开 Access 应用程序，依次选择“ 文件” “ 新建”，则弹出如图 12-1 所示的界面。然后在“ 新建” 一栏中选择“ 空数据库”，接着弹出一个如图 12-2 所示的对话框，让用户填写数据库的名称，并选择将要保存的磁盘地址。本例要创建一个存储通讯录的数据库，名称为

“ Info.mdb”，保存在默认目录下。

图 12-1 新建数据库

 图 12-2 为新建数据库命名及选择保存地址

### 12.2.2 创建数据表

在图 12-2 所示的对话框中单击“ 创建” 按钮，则弹出如图 12-3 所示的界面。

 图 12-3 创建数据表

### 12.2.3 设计和修改数据表的结构

本例要创建的数据库是通讯录数据库，在进行下一步操作之前，应该考虑本数据库包含哪些表，以及每张表的结构如何。为简单起见，在本小节只考虑一张表，这张表只记录有关人员的联系方式。此表应包含每个人的 ID，即表的主键，还应该包括每个人的姓名（Name）、电话（Telephone）以及住址（Address）。其中 ID 和 Telephone 的数据类型为数字，Name 和

Address 的数据类型为文本。设计好这张表后，下面具体介绍创建表的过程。

在如图 12-3 所示的界面上双击“ 使用设计器创建表” 选项，打开如图 12-4 所示的界面。然后在此界面中的字段名称部分分别填上 ID、Name、Telephone 和 Address，相应的数据类型分别选择数字、文本、数字和文本。然后在“ ID” 字段上单击鼠标右键，在弹出的对话框中选择“ 主键”，将 ID 定义为此表的主键。创建后的数据表如图 12-5 所示。返回如图 12-3 所示的创建数据表，将该数据表重命名为 Contact，并保存。完成后的界面如图 12-6 所示。

 图 12-4 设计数据表

 图 12-5 设计后的数据表

 图 12-6 设计好 Contact 表后的界面

如果对数据表的结构不满意，想要修改，则在图 12-6 所示的界面上单击“ Contact”，然后单击工具栏上的“ 设计” 按钮，在打开的界面中对表的结构进行修改。

设计好 Contact 表后，还要对其进行填充，即填充一些有关人员的信息。在如图 12-6 所示的界面中双击“ Contact”，则可以对表 Contact 进行填充，如图 12-7 所示。

 图 12-7 Contact 表的信息

### 12.2.4 设置表之间的关系

通常情况下，一个数据库是由多张表组成的，并且这些表之间还存在着某些联系。为了便于读者更好地立即，本章中的 Info.mdb 数据库由两张表组成即记录个人信息的 Personal 表和记录每个人联系方式的 Contact 表。关于 Contact 表，在上一节中已经创建好，下面介绍创建 Personal 表的方法。假设 Personal 表包含每个人的 ID、姓名、出生日期、文化程度和工作，它们对应的 Personal 表的字段名称分别为：ID、Name、Birthday、Education 和 Work，数据类型分别为：数字、文本、日期/时间、文本和文本。其中 ID 是主键。创建好的 Personal 表如图 12-8 所示。

 图 12-8 创建好的 Personal 表

创建好 Personal 表后，可进行相关信息的填充，如图 12-9 所示。



图 12-9 Personal 表的信息

实际上，在 Access 数据库中，不同表中的数据之间都存在一种关系，这种关系将数据库里各张表中的每条数据记录都和数据库中惟一的主题相联系，使得对一个数据的操作都成为数据库的整体操作。下面具体介绍设置表之间关系的操作步骤。

1．向关系界面中加入所有数据表

单击菜单栏中“ 工具” “ 关系”，弹出如图 12-10 所示的“ 显示表” 对话框。然后同时选中 Contact 表和 Personal 表，单击“ 添加” 按钮。此时的关系对话框如图 12-11 所示。

图 12-10 “ 显示表” 对话框 图 12-11 “ 关系” 对话框

2．编辑各个关系

单击菜单栏中“ 关系” “ 编辑关系”，弹出“ 编辑关系” 对话框（如图 12-12 所示）。然后单击“ 新建” 按钮，在弹出的“ 新建” 对话框中分别选择左表名称 Contact，右表名称： Personal，左列名称：ID，右列名称：ID，如图 12-13 所示。单击“ 确定” 按钮后，在编辑关系对话框中选中“ 实施参照完整性”、“ 级联更新相关字段” 和“ 级联删除相关记录” 3 个复选框，如图 12-14 所示。单击“ 创建” 按钮，则 Contact 表和 Personal 表的 ID 便关联在一起。

关联后的关系如图 12-15 所示。

图 12-12 编辑关系对话框 图 12-13 新建对话框

图 12-14 定义关系后的编辑关系对话框 图 12-15 定义关系后的关系图

如果有多张表，可能就存在多种关系，读者可以根据上述方法对其他的关系进行编辑。

本节所采用的例子从实用的角度来说是不严格的，但是这个例子抓住了 Access 数据库的本质，从这个简单的例子中，读者可以很容易掌握如何创建 Access 数据库，如何设计、创建及修改数据表，以及如何设置表之间的关系。

### 12.2.5 编辑管理数据库记录

1．输入数据

在一个表中输入数据时，只有当前最后记录的下一行可以输入。在要输入的列的相应的方格内单击鼠标左键，即可输入数据。

2．修改数据

一个表中的任何一条记录的任何一个数据都可以修改。修改记录也非常简单，用鼠标左键单击想要修改的数据所在的方格，然后用 BackSpace 键将数据删除，最后重新填入新数据即可。

3．删除记录

如果某条记录完全错误，想要删除，则在该记录所在行的最左端单击鼠标左键，此时该行变黑，然后通过“ Delete” 键即可删除。如果想删除某个范围内连续多条记录，则首先选中第一条记录，然后在按住“Shift” 键的同时，用鼠标左键选中最后一条记录，则此时这个范围内的所有记录都变成黑色背景，最后按“ Delete” 键删除。

4．查询记录

单击菜单栏中的“ 编辑” “ 查找” 项，弹出“ 查找和替换” 对话框（或者通过快捷键

“ Ctrl+F”也可以弹出此对话框），通过该对话框可进行数据的查找和替换。

|  |
| --- |
| 技巧：设置 ODBC 数据源是一件很容易的事情，对于普通用户，最简单设置 ODBC 数据源的步骤如下（以 Windows2000 为例）：打开“控制面板”“ 管理工具”，双击“ 数据源（ODBC）” 图标，弹出 ODBC 数据源管理器对话框如图 12-16 所示。单击“ 添加” 按钮，弹出创建新数据源对话框如图 12-17 所示。本章都以 Access 数据库为例，因此这里选择 Access 数据库的驱动，单击“ Driver do Microsoft Access (\*.mdb)”，然后单击“ 完成” 按钮，此时弹出提示安装 Access 数据源的对话框（如图 12-18 所示）。在数据源名一栏中填写 Information，单击“ 选择” 按钮，选择刚建好的数据库 Info.mdb，然后单击“ 确定” 按钮。至此，ODBC 数据源已被设置完毕。  这是设置 ODBC 数据源最常用的方法，也非常简单方便。此外对于对计算机比较熟悉的用户，还可以通过注册表或者编写程序来设置数据源。是除了上述所说的，还有更简单的设置数据源的办法。利用 COM 组件 DataSourceLocator，用户可以在程序运行的过程中动态设置数据源。至于如何应用此组件，本章第 6 节将会详细介绍。 |

 图 12-16 ODBC 数据源管理器对话框

图 12-17 创建新数据源对话框 图 12-18 安装 Access 数据源对话框

## 12.3 标准的 SQL 语句

SQL 的全称是 Structured Query Language，即结构化查询语言，它是一种介于关系代数与关系演算之间的语言，其功能包括查询、操纵、定义和控制 4 个方面，是一个通用的、功能极强的关系数据库语言，目前已成为关系数据库的标准语言。本节将介绍 SQL 的基本知识以及一些常用的 SQL 语句。

### 12.3.1 SQL 的基本知识

SQL 语言是 1974 年由 Boyce 和 Chamberlin 提出的。由于它功能丰富、语言简洁、使用方法灵活，因此倍受用户和计算机工业界的欢迎，被众多计算机公司和软件公司所采用。后来经各公司的不断修改、扩充和完善，SQL 语言最终发展成为关系数据库的标准语言。SQL 语言集数据查询（Data Query）、数据操纵（Data Manipulation）、数据定义（Data Definition）和数据控制（Data Control）功能于一体，充分体现了关系数据语言的特点和优点。

1．综合统一

SQL 语言集数据定义语言（DDL）、数据操纵语言（DML）、数据控制语言（DCL）的功能于一体，语言风格统一，可以独立完成数据库生命周期中的全部活动，包括定义关系模式、录入数据以建立数据库、查询、更新、维护、数据库重构、数据库安全性控制等一系列操作的要求，这就为数据库应用系统开发提供了良好的环境。另外，在关系模型中，实体和实体间的联系均用关系表示，这种数据结构的单一性带来了数据操作符的统一性，查找、插入、删除、更新等任一种操作只需要一种操作符，从而克服了非关系系统因信息表示方式的多样性带来的操作复杂性。

2．高度非过程化

用 SQL 语言进行数据操作，用户只需提出“ 做什么”，而不必指明“ 怎么做”。用户无需了解存取路径，存取路径的选择以及 SQL 语句的操作过程由系统自动完成，这不但大大减轻了用户负担，而且有利于提高数据的独立性。

3．面向集合的操作方式

SQL 语言采用集合操作方式，不仅查找结果可以是元组的集合，而且插入、删除、更新操作的对象也可以是元组的集合。

4．以同一语法结构提供两种使用方式

SQL 语言既是自含式语言，又是嵌入式语言。作为自含式语言，它能够独立地用于联机交互的使用方式，用户可以在终端键盘上直接键入 SQL 命令对数据库进行操作。作为嵌入式语言，SQL 语言能够嵌入到高级语言程序中，供程序员设计程序时使用。而在两种不同的使用方式下，SQL 语言的语法结构基本上是一致的。这种统一的语法结构提供两种不同使用方式的作法，为用户提供了极大的灵活和方便。

5．语言简洁，易学易用

SQL 语言功能极强、设计巧妙。完成数据定义、数据查询、数据操纵、数据控制的核心功能的动词只有 9 个，即 CREATE、DROP、ALTER、SELECT、INSERT、UPDATE、DELETE、

GRANT、REVOKE，如表 12-1 所示。而且 SQL 语言语法简单，接近英语口语，因此容易学习和使用。

表 12-1 SQL 语言的动词

|  |  |  |
| --- | --- | --- |
|  | SQL 功能 | 动词 |
| 数据查询 |  | select |
| 数据定义 |  | create、drop、alter |
| 数据操纵 |  | insert、update、delete |
| 数据控制 |  | grant，revoke |

### 12.3.2 基本的 SQL 语句

SQL 语句虽然简单，但是非常灵活，且功能强大，这里不可能对其进行全面的介绍。本小节只对最常用的 SQL 语句进行简单介绍。SQL 语句最常用的功能有数据查询、插入记录、更新记录和删除记录。下面进行详细介绍。

1．数据查询

数据查询是数据库中最重要的操作，SQL 语句提供 SELECT 语句进行数据的查询，该语句具有灵活的使用方式和丰富的功能，其一般格式如下：

SELECT [ALL | DISTINCT] <目标列表达式> [, <目标列表达式> ]…

FROM <表名或视图名> [, <表名或视图名> ]…

[WHERE <条件表达式>]

[GROUP BY <列名 1> HAVING <条件表达式> ]]

[ORDER BY <列名 2> [ASC | DESC]]

整个 SELECT 语句的含义是，根据 WHERE 子句的条件表达式，从 FROM 子句指定的基本表或视图中找出满足条件的元组，再按 SELECT 子句中的目标列表达式，选出元组中的属性值形成结果表。如果有 GROUP 子句，则将结果按<列名 1>的值进行分组，该属性的列值相等的元组为一个组，每个组产生结果表中的一条记录。通常会在每组中作用集函数。如果 GROUP 子句带有 HAVING 短语，则只有满足指定条件的组才会输出。如果有 ORDER 子句，则结果还要按<列名 2>进行排序，ASC 表示升序，DSEC 表示降序。下面举几个简单的例子（这里进行操作的数据库就是上节创建的数据库）。

1. 查询表 Contact 中的所有记录：

select \* from Contact

1. 查询所有人的姓名：

select Name from Contact

1. 查询王明的电话和地址：

select Telephone, Address from Contact where Name=’王明’

1. 将表 Personal 中所有人按年龄由小到大排序。

select \* from Personal order by Birthday desc

2．插入记录插入记录使用的是 INSERT 语句，一般格式如下：

INSERT INTO <表名> [(<属性列 1> [, <属性列 2> ]…)]

VALUES (<常量 1> [, <常量 2> ]…)

INSERT 语句的功能是将新元组插入指定表中，其中新记录属性列 1 的值为常量 1，属性列 2 的值为常量 2，依次类推。如果某些属性列在 INTO 子句中没有出现，则新记录在这些列上将取空值。不过需要注意的是，在表定义时说明了 NOT NULL 的属性列不能为空值，否则出错。下面是一个简单例子。

往表 Contact 中插入一条记录：

insert into Contact values(’11’, ’苏培’, ’62777098’, ’清华大学计算机系’)

3．更新记录更新记录通过 UPDATE 子句完成，一般格式如下：

UPDATE <表名> SET <列名> = <表达式> [, <列名> = <表达式> ]…

[WHERE <条件>]

例如，如果要更改上面刚插入的记录，则语句如下：

update Contact set Name=’苏娟’, Telephone=’62788155’ where Name=’苏培’

4．删除记录

SQL 中用 DELETE 子句实现删除记录的操作，一般格式如下：

DELETE FROM <表名> [WHERE <条件>]

例如，如果要删除前面插入的记录，SQL 语句如下：

delete from Contact where ID=11

### 12.3.3 关系数据库的联合查询技术

能够从多个表中选择和操作数据是 SQL 的特色之一，这又叫做联合查询技术，它包括联合查询和连接查询。

1．联合查询

联合查询是通过 UNION 运算符实现的，它可以将多个 SELECT 语句的查询结果合并，使之作为一个结果集合显示（各结果集的数据类型要相同）。UNION 的语法格式如下：

SELECT\_STATEMENT

UNION [ALL] SELECT\_STATEMENT [UNION [ALL] SELECT\_STATEMENT]…

例如，如果要显示出表 Contact 和表 Personal 中所有人的姓名，则 SQL 语句如下：

select Name from Contact union select Name from Personal

2．连接查询

通过连接运算符可以实现多个表查询。连接是关系数据库模型的主要特点，也是它区别于其他类型数据库管理系统的一个标志。

通常的连接语法格式为：

FROM JOIN\_TABLE JOIN\_TYPE JOIN\_TABLE [ON （JOIN\_CONDITION）]

其中，JOIN\_TABLE 是连接操作的表名，JOIN\_TYPE 是连接类型，它分为内连接、外连接和交叉连接 3 类。

1. 内连接

内连接（INNER JOIN）又分为等值连接、不等连接和自然连接 3 种。等值连接在连接条件中使用“=” 运算符比较被连接的列值，其查询结果中列出被连接表中的所有列，包括重复列；不等连接在连接条件中使用除“ =” 之外的其他比较运算符；自然连接使用“ =” 运算符比较被连接的列值，但查询结果中不包括重复列。

1. 外连接

外连接分为左外连接（LEFT OUTER JOIN 或 LEFT JOIN）、右外连接（RIGHT OUTER JOIN 或 RIGHT JOIN）和全外连接（FULL OUTER JOIN 或 FULL JOIN）。与内连接不同的是，采用外连接时，返回到查询结果集合中的不仅包含符合连接条件的行，而且还包括左表（左外连接时）、右表（右外连接时）和两个表（全外连接时）中的所有数据行。

1. 交叉连接

交叉连接（CROSS JOIN）不带 WHERE 子句，返回被连接的两个表所有数据行的笛卡尔积。

## 12.4 ADO 基础知识

在第 1 节中介绍了以前的几种 Visual C++连接数据库的技术。Visual C++虽然对数据库的支持很强大，功能也很全面，但是不管是先前的 ODBC 还是 DAO，再到后来的 OLE DB，用 Visual C++来开发数据库总是很麻烦，特别是其中的数据转换令初学者十分头疼。不过，等到 ADO 技术开发出来以后，这些都变得容易了。

### 12.4.1 ADO 的技术特点

ADO 的全称是 ActiveX Data Objects，它是微软公司为数据库应用程序开发的新接口。

ADO 基于 COM，提供编程语言可利用的对象，它不仅面向 Visual C++，同时还提供面向其他开发工具的应用，如 Visual Basic、VJ 等，甚至还提供面向诸如 VBScript、JavaScript 等脚本语言的应用。ADO 的底层是 OLE DB，不仅能访问关系型数据库，也能访问非关系型数据库。ADO 对 OLE DB 的包装是相当成功的，相对于 OLE DB 中众多的接口来说，ADO 对象模型简明扼要，没有一点多余的东西，并且其功能还要比 DAO 强大得多。并且可以说到目前为止，ADO 是目前最快的数据库访问的中间层。

### 12.4.2 ADO 的结构

ADO 的结构如图 12-19 所示。从图中可以看出，ADO 模型包含了 7 个对象和 4 个集合。

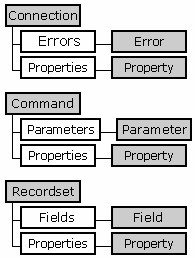


图 12-19 ADO 结构图

* 连接对象（Connection）：用于与数据源的连接，以及处理一些命令和事务。
* 命令对象（Command）：用于处理传递给数据源的命令。
* 记录集对象（Recordset）：用于处理数据的表格集，如获取和修改数据。
* 域对象（Field）：用于表示记录集中的列信息，包括列值以及其他信息。
* 参数对象（Parameter）：用于对传送给数据源的命令赋参数值。
* 属性对象（Property）：用于操作在 ADO 中使用的其他对象的详细属性。
* 错误对象（Error）：用于获得连接对象所发生的详细错误信息。
* 域集合（Fields）：记录集对象中包含了域对象的集合，域对象的集合中包含了所有代表记录集中每列的域对象。
* 参数集合（Parameters）：命令对象中包含了参数对象的集合，参数集合中包含了应用于命令对象的所有参数对象。
* 属性集合（Properties）：在连接对象、命令对象、记录集对象和域对象中都包含了属性对象的集合，属性对象的集合中包含了这些对象的所有特性。
* 错误集合（Errors）：连接对象中包含了错误对象的集合，错误集合中包含了在一次连接数据源时所产生的所有错误对象。

## 12.5 ADO 的对象和集合

### 12.5.1 连接对象

连接对象即 Connection 对象，它代表与数据源之间的一个连接，其常用的方法和属性分

别如表 12-2 和表 12-3 所示。

表 12-2 连接对象的方法及其说明

|  |  |
| --- | --- |
| 方法 | 说明 |
| Open | 打开到数据源的连接 |
| Execute | 执行指定的查询、SQL 语句、存储过程或特定提供者的文本等内容 |
| Close | 关闭 Connection 对象，释放所有关联的系统资源 |
| BeginTrans | 启动新的事务 |
| CommitTrans | 保存所有更改并结束当前事务，也可启动新事务 |
| RollbackTrans | 取消当前事务中所作的任何更改并结束事务，也可启动新事务 |

表 12-3 连接对象的属性及其说明

|  |  |
| --- | --- |
| 属性 | 说明 |
| Attributes | 可以读写，其值可能为 AdXactCommitRetaining 和 AdXactAbortRetaining 中任意一个 |
| ConnectionTimeout | 指示在终止尝试和产生错误前建立连接期间需等待的时间 |
| CommandTimeout | 指示在终止尝试和产生错误前执行命令期间需等待的时间 |
| ConnectionString | 包含用来建立到数据源的连接的信息 |
| DefaultDatabase | 指示 Connection 对象的默认连接数据库 |
| IsolationLevel | 设置 Connection 对象的隔离级别 |
| Mode | 设置或返回当前连接上提供者正在使用的访问权限 |
| Provider | 设置或返回连接提供者的名称 |
| Version | 返回 ADO 执行的版本号 |

利用此对象，用户可以进行以下一些操作：

* 在进行连接前，通过 ConnectionString、ConnectionTimeOut 和 Mode 属性来配置建立连接的信息、建立连接的超时限制和访问模式。
* 通过 CursorLocation 属性支持批处理方式。
* 通过 DefaultDatabase 属性设置缺省的连接数据库。
* 通过 IsolationLevel 属性设置 Connection 对象的隔离级别。
* 通过 Provider 属性设置 OLE DB 的提供者。
* 通过 Open()和 Close()方法分别来建立和关闭与数据源的连接。
* 通过 Execute()方法来执行命令，通过 CommandTimeout 属性设置执行命令超时的时限。
* 通过 BeginTrans()、CommitTrans() 和 RollbackTrans()方法以及 Attributes 属性进行事务控制。
* 通过 Errors 集合检查数据源的错误信息。
* 通过 Version 属性得到 ADO 的版本信息。
* 通过 OpenSchema()方法来得到数据库的表信息。

### 12.5.2 命令对象

命令对象即 Command 对象，一个 Command 对象代表一个对数据源执行的命令，利用此对象可以进行数据库的一些操作，比如查询、修改等。

Command 对象的主要方法和属性如表 12-4 及表 12-5 所示。

表 12-4 命令对象的方法及其说明

|  |  |
| --- | --- |
| 方法 | 说明 |
| Execute | 执行在 CommandText 属性中指定的查询、SQL 语句或存储过程 |
| CreateParameter | 用指定的名称、类型、方向、大小和值创建新的 Parameter 对象，在参数中传送的所有值都将写入相应的 Parameter 属性 |
| Cancel | 终止执行异步 Execute()方法调用 |

表 12-5 命令对象的属性及其说明

|  |  |
| --- | --- |
| 属性 | 说明 |
| CommandText | 设置或返回 Command 对象的文本。通常该对象为 SQL 语句 |
| CommandTimeout | 指示在终止尝试和产生错误之前执行命令期间需等待的时间 |
| CommandType | 指定命令类型，可以是文本命令、表格名或者是一个存储过程 |
| ActiveConnection | 指示指定的 Command 对象当前所属的 Connection 对象 |
| State | 随时使用 State 属性确定指定对象的当前状态 |
| Prepared | 指示执行前是否保存命令的编译版本 |

利用 Command 对象的一些方法和属性，用户可以进行以下一些操作：

* 通过 CommandText 属性设置命令串。
* 通过 Parameter 对象和 Parameters 集合定义参数化查询或存储过程的参数。
* 通过 Execute()方法执行一个命令，并返回一个 Recordset 对象。
* 在执行命令前，通过 CommandType 属性设置 Command 对象的类型，以达到优化性能的目的。
* 在执行命令前，通过 Prepared 属性来设置底层提供者是否为当前命令保存一个编译过的版本。如果现在保存，则以后再执行时，速度会加快。
* 通过 CommandTimeout 属性设置执行命令的超时时限。
* 通过 ActiveConnection 属性来指定该命令对象所属的连接（Connection）。
* 通过 Name 属性来指定该命令对象作为其所属连接的一个方法，以便以后可以在相应

Connection 对象上通过 Name 属性指定的名称来执行命令。

* 通过 Source 属性将 Command 对象传递给 Recordset 来得到数据。
* 通过 Properties 集合来访问提供者指定的属性。

### 12.5.3 记录集对象

记录集对象即 Recordset 对象，它表示一个从数据源选择的一组纪录的集合，其主要方法和属性如表 12-6 及表 12-7 所示。

表 12-6 记录集对象的方法及其说明

|  |  |  |
| --- | --- | --- |
| 方法 |  | 说明 |
| MoveFirst | 移动到记录集的第一条记录处 |  |
| MoveLast | 移动到记录集的最后一条记录处 |  |

续表

|  |  |
| --- | --- |
| 方法 | 说明 |
| MovePrevious | 移动到记录集中当前记录的前一条记录处 |
| MoveNext | 移动到记录集中当前记录的后一条记录处 |
| Move | 移动到指定的记录 |
| NextRecordset | 返回复合命令语句中下一条命令的结果，或返回多个结果的已存储过程结果 |
| Open | 直接打开一个记录集，而不是作为执行命令或连接命令产生的记录集 |
| Close | 关闭记录集 |
| Delete | 删除记录集中的当前记录 |
| Update | 将当前对记录集的改动保存到数据源中 |
| UpdateBatch | 对更新命令进行批处理。它对记录集的变化进行缓存，直到调用此方法对变化进行批量更新 |
| CancelUpdate | 取消 Update 更新前所作的改动 |
| CancelBatch | 取消数据源中为提交的批量改动 |
| GetRows | 获得记录集中的多行数据，并将获得的数据写入数组中 |
| Requery | 重新执行以前执行过的命令，重新获得记录集 |
| Support | 判断某一特定的记录集对象是否支持特定的属性和方法 |
| Clone | 产生一个记录集对象，其中包含记录集对象的所有数据 |

表 12-7 记录集对象的属性及其说明

|  |  |
| --- | --- |
| 属性 | 说明 |
| AbsolutePage | 指定当前记录所在的页 |
| AbsolutePosition | 根据其在 Recordset 中的序号位置移动到记录，或确定当前记录的序号位置 |
| ActiveConnection | 指定 Recordset 对象当前所属的 Connection 对象 |
| BOF | 指示当前记录位置是否位于 Recordset 对象的第一个记录之前 |
| EOF | 指示当前记录位置是否位于 Recordset 对象的最后一个记录之后 |
| Bookmark | 返回惟一标识 Recordset 对象中当前记录的书签，或者将 Recordset 对象的当前记录设置为由有效书签所标识的记录 |
| CacheSize | 控制提供者在缓存中所保存的记录的数目，并可控制一次恢复到本地内存的记录数目 |
| CursorType | 获得或设置当前光标的类型 |
| EditMode | 获得当前的编辑状态 |
| Filter | 指定一个在行集中移动时所使用的过滤器 |
| LockType | 得到或设置当前的访问状态 |
| MaxRecord | 获得或设置一次操作中的 Recordset 对象中所返回的最大行的数目 |
| PageCount | 获得记录集中使用的页数 |
| PageSize | 获得或指定一页中的行数 |
| RecordCount | 获得记录集中包含的记录的数目 |
| Source | 获得记录集中记录的来源，可以是一个命令对象、SQL 语句、存储的过程或表名 |
| Status | 返回当前行的状态 |

### 12.5.4 ADO 的其他对象

除了前面介绍的 3 个重要对象，ADO 还包括域对象 Field、参数对象 Parameter、属性对象 Property 和错误对象 Error。

1．域对象

Field 对象代表使用普通数据类型的数据列，它的常用方法和属性分别如表 12-8 和表 12-9 所示。

表 12-8 域对象的方法及其说明

|  |  |
| --- | --- |
| 方法 | 说明 |
| AppendChunk | 可将二进制或字符数据填写到域对象中，在系统内存有限的情况下，可使用此方法对长整型值进行部分操作 |
| GetChunk | 检索域对象部分或全部二进制或字符数据，在系统内存有限的情况下，可使用此方法处理部分长整型值 |

表 12-9 域对象的属性及其说明

|  |  |
| --- | --- |
| 属性 | 说明 |
| ActualSize | 指示字段的值的实际长度 |
| Attributes | 域的属性集合，可以用来判断此域是否有固定长度或者是否可以为空 |
| DefinedSize | 确定 Field 对象的数据容量 |
| Name | 列的名称 |
| NumericScale | 在浮点数中用于指出小数点右边多少位 |
| OriginalValue | 发生任何更改前已在记录中存在的 Field 的值 |
| Precision | 表示数字 Field 对象的精度 |
| Type | 列中的值的数据类型 |
| UnderlyingValue | 反映了数据源中列的当前值，当在事务的同步时被使用 |
| Value | 列的值，用来查询或者设置此列的值 |

2．参数对象

Parameter 对象代表基于参数化查询或存储过程的 Command 对象相关联的参数，它的常用方法和属性分别如表 12-10 和表 12-11 所示。

表 12-10 参数对象的方法及其说明

|  |  |
| --- | --- |
| 方法 | 说明 |
| AppendChunk | 可将二进制或字符数据填写到域对象中，在系统内存有限的情况下，可使用此方法对长整型值进行部分操作 |

表 12-11 参数对象的属性及其说明

|  |  |
| --- | --- |
| 属性 | 说明 |
| Attributes | 表示参数的特性，这个属性包含几个字节标志的组合，标志用于指示参数是否有符号 |
| Direction | 指示 Parameter 所标明的是输入参数、输出参数还是二者都是，或该参数是否为存储过程的返回值 |
| Name | 参数对象的名称 |
| NumericScale | 在浮点数中用于指出小数点右边用于表示这个值的位数 |
| Precision | 确定表示参数对象数字值的精度 |
| Type | 列中的值的数据类型 |
| Size | 表示 Parameter 对象的大小 |
| Value | 包含分配给参数的实际值 |

3．属性对象

Property 对象代表由提供者定义的 ADO 对象的动态特性，它包括内置属性和动态属性两种类型。

内置属性是在 ADO 中实现并立即可用于任何新对象的属性，它们不会作为 Property 对象出现在对象的 Properties 集合中。可以更改它们的值，但不能更改它们的特性。

动态属性由现行数据提供者定义，并出现在相应的 ADO 对象的 Properties 集合中。动态

Property 对象有 4 个内置属性：

* Name 属性是标识属性的字符串；
* Type 属性是用于指定属性数据类型的整数；
* Value 属性是包含属性设置的变体型；
* Attributes 属性是指示特定于提供者属性特征的长整型值。

4．错误对象

Error 对象用于获得连接对象所发生的详细错误信息，它的主要属性如表 12-12 所示。

表 12-12 错误对象的属性及其说明

|  |  |
| --- | --- |
| 属性 | 说明 |
| Description | 获得错误的简要说明 |
| NativeError | 获得对特殊的 Error 对象检索特定数据库的错误信息 |
| Number | 确定发生错误的类型，该属性的值是与错误条件对应的惟一数字 |
| Source | 确定产生错误的原始对象或应用程序的名称，该名称可以是对象的类名或编程 ID |
| SQLState | 读取由提供者在处理 SQL 语句过程中出现错误时返回的 5 个字符的错误代码 |

### 12.5.5 集合

ADO 有 4 个集合，分别是域集合 Fields、参数集合 Parameters、属性集合 Properties 以及错误集合 Errors。

1．域集合

Fields 集合包含 Recordset 对象的所有 Field 对象，它的常用方法和属性分别如表 12-13 和表 12-14 所示。

表 12-13 域集合的方法及其说明

|  |  |
| --- | --- |
| 方法 | 说明 |
| Append | 添加新的 Field 对象到 Fields 集合中 |
| Delete | 从 Fields 集合中删除 Field 对象 |
| Refresh | 更新 Fields 集合中的 Field 对象 |
| Item | 返回 Fields 集合中的 Field 对象 |

表 12-14 域集合的属性及其说明

|  |  |
| --- | --- |
| 属性 | 说明 |
| Count | 确定给定 Fields 集合中 Field 对象的数目 |

2．参数集合

Parameters 集合包含 Command 对象的所有 Parameter 对象，它的常用方法和属性分别如表 12-15 和表 12-16 所示。

表 12-15 参数集合的方法及其说明

|  |  |
| --- | --- |
| 方法 | 说明 |
| Append | 添加新的 Parameter 对象到 Parameters 集合中 |
| Delete | 从 Parameters 集合中删除 Parameter 对象 |
| Refresh | 更新 Parameters 集合中的 Parameter 对象 |
| Item | 返回 Parameters 集合中的 Parameter 对象 |

表 12-16 参数集合的属性及其说明

|  |  |
| --- | --- |
| 属性 | 说明 |
| Count | 确定给定 Parameters 集合中 Parameter 对象的数目 |

3．属性集合

Properties 集合包含特定对象实例的所有 Property 对象，它的常用方法和属性分别如表

12-17 和表 12-18 所示。

表 12-17 属性集合的方法及其说明

|  |  |
| --- | --- |
| 方法 | 说明 |
| Refresh | 更新 Properties 集合中的 Property 对象 |
| Item | 返回 Properties 集合中的 Property 对象 |

表 12-18 属性集合的属性及其说明

|  |  |
| --- | --- |
| 属性 | 说明 |
| Count | 确定给定 Properties 集合中 Property 对象的数目 |

4．错误集合

Errors 集合包含为响应涉及提供者的单个错误而创建的所有 Error 对象，它的常用方法和属性分别如表 12-19 和表 12-20 所示。

表 12-19 错误集合的方法及其说明

|  |  |
| --- | --- |
| 方法 | 说明 |
| Clear | 清空 Errors 集合中的全部 Error 对象，发生错误时，ADO 会自动清空 Errors 集合，并用新的 Error 对象填充 Errors 集合 |
| Item | 返回 Errors 集合中的 Error 对象 |

表 12-20 错误集合的属性及其说明

|  |  |
| --- | --- |
| 属性 | 属性 |
| Count | 确定给定 Errors 集合中 Error 对象的数目 |

## 12.6 数据库操作

本节将给出一个具体的利用 ADO 技术操作数据库的实例，通过这个实例，读者对 ADO 技术会有更进一步的了解。

### 12.6.1 动态链接库的引入和 OLE/COM 库的初始化

利用 ADO 技术编写程序，首先要告诉编译器去什么地方找 ADO 的库文件，通常这个文件在 c:\program files\common files\system\ado 下，文件名为 msado15.dll。因此，第一步应该在文件“ stdafx.h” 的开头加上如下语句：

#import "c:\program files\common files\system\ado\msado15.dll" no\_namespace rename("EOF", "adoEOF")

使用 no\_namespace 意味着程序员不需要在初始化变量的时候引用名字空间，最后的操作

表明将 EOF 更名为 adoEOF，如果不这样做，在程序中很可能会碰到常量冲突。

将 ADO 的动态链接库引入后，在正式用 ADO 编写程序前，还需要对 OLE/COM 库初始

化，代码如下：

CoInitialize( NULL );

### 12.6.2 ADO 与数据库的连接

引入 ADO 库并初始化 OLE/COM 后，接下来要进行 ADO 与数据库的连接。

ADO 与数据库连接时，有一个最重要的数据类型，即\_ConnectionPtr，它实际上就是由类模板\_com\_ptr\_t 而得到的一个具体的实例类。\_ConnectionPtr 类封装了 Connection 对象的 Idispatch 接口指针，及一些必要的操作，通过这个指针便可以操纵 Connection 对象。通常的连接实现方式如下：

\_ConnectionPtr conn;

conn.CreateInstance(\_\_uuidof(Connection));

conn->Open(BSTR ConnectionString, BSTR UserID, BSTR Password, long Options);

各参数意义如下：

* ConnectionString：最关键的参数，它表明了要连接的数据源，在本章中采用的是 Access 数据库，它的格式一般为: "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=db.mdb"，至于其他数据库，请读者参考相关资料；此外，ConnectionString 还可以是数据源的名字。
* UserID：用户名，如果 ConnectionString 中已经包含，则可以为空。
* Password：密码，如果 ConnectionString 中已经包含，则可以为空。
* Options：表示是以同步方式(adConnectUnspecified)还是以异步方式(adAsyncConnect) 进行连接，默认为同步。具体的，有以下两种实现连接的方式：

1. LPCSTR strConnect = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=test.mdb"; conn->Open(strConnect, "", "", adConnectUnspecified);
2. conn->Open("DSN=DataSourceName", "", "", adConnectUnspecified);

对于本章的例子，要连接的数据库就是在第 2 节创建的数据库 Info.mdb（数据源也在第 2 节已经设置好），则连接语句如下：

\_ConnectionPtr m\_connection;

//创建实例

m\_connection.CreateInstance(\_\_uuidof(Connection) );

//根据连接字符串开启数据连接

m\_connection->Open( "DSN=Information", "", "", -1);

### 12.6.3 获得和遍历记录集

建立连接后，便可以创建一个 ADO 记录集。在得到记录集之前，需要先声明一个

\_RecordsetPtr 型变量，然后创建记录集的实例：

\_Recordset m\_Recordset;

//创建实例

m\_Recordset.CreateInstance(\_\_uuidof(Recordset));

这里\_RecordsetPtr 的来历类似于\_ConnectionPtr，也是由类模板\_com\_ptr\_t 而得到的一个具体的实例类。一般说来，打开记录集有 3 种方式。

1．利用 Connection 对象的 Execute()方法执行 SQL 命令

Execute()方法的原型如下：

Execute（\_bstr\_t *CommandText*, VARIANT\* *RecordsAffected*, long *Options*）

各参数意义如下：

* CommandText：命令字串，通常是 SQL 语句。
* RecordsAffected：操作完成后影响的行数。
* Options：CommandText 中内容的类型。取值为 adCmdText 表明是文本命令；取值为 adCmdTable 表明是一个表名；取值为 adCmdProc 表明是一个存储过程。

假设 m\_Connection 和 m\_Recordset 分别是创建好的连接对象和记录集对象，则利用 Connection 对象的 Execute()方法打开记录集代码如下：

\_variant RecordsAffected;

m\_Connection->Execute(“select \* from Contact”, &RecordsAffected, adCmdText);

2．利用 Command 对象执行 SQL 命令

当要执行复杂的命令以及执行带参数的命令时，要使用命令对象对数据源进行操作。一

般的，利用 Command 对象打开记录集代码如下：

\_CommandPtr m\_Command;

//创建实例

m\_Command.CreateInstance("ADODB.Command");

//指向当前连接

m\_Command->ActiveConnection = m\_Connection;

//设置命令字串

m\_Command->CommandText = "select \*from Contact";

//执行命令，打开记录集

m\_Recordset = m\_Command->Execute(&vNULL, &vNULL, adCmdText);

3．直接利用 Recordset 对象打开记录集利用 Recordset 的 Open()方法，可以方便的打开一个记录集。Open()方法的原型如下：

Open(VARIANT *Source*, VARIANT *ActiveConnection*, CursorTypeEnum *CursorType*,LockTypeEnum

*LockType*, LONG *Options*).

各参数意义如下：

* Source：通常是对数据库进行操作的 SQL 语句，但具体还要看第 5 个参数的取值；
* ActiveConnection：表示当前的数据连接。
* CursorType：记录集光标类型，它的取值如表 12-21 所示。
* LockType：用于指示在什么时候锁定记录，取值如表 12-22 所示。
* Options：操作类型，取值如表 12-23 所示。

表 12-21 光标类型 CursorType

|  |  |
| --- | --- |
| 取值 | 意义 |
| adOpenForwardOnly | 指向前游标，默认值。除了只能在记录中向前滚动外，与静态游标相同 |
| .adOpenKeyset | 键集游标。用户自己的记录集不能访问其他用户删除的记录，无法查看其他用户添加的记录，但仍然可以看见其他用户更改的数据 |
| adOpenDynamic | 动态游标。可以看见其他用户所作的添加、更改和删除。允许在记录集中进行所有类型的移动，但不包括提供者不支持的书签操作 |
| adOpenStatic | 静态游标。可以用来查找数据或生成报告的记录集合的静态副本。另外，对其他用户所作的添加、更改或删除不可见 |

表 12-22 锁定类型 LockType

|  |  |
| --- | --- |
| 取值 | 意义 |
| AdLockReadOnly | （默认值）只读，不能改变数据 |
| AdLockPessimistic | 保守式锁定（逐个），提供者完成确保成功编辑记录所需的工作，通常通过在编辑时立即锁定数据源的记录 |
| AdLockOptimistic | 开放式锁定（逐个），提供者使用开放式锁定，只在调用 Update()方法时才锁定记录 |
| AdLockBatchOptimistic | 开放式批更新。用于批更新模式（与立即更新模式相对） |

表 12-23 操作类型 Options

|  |  |
| --- | --- |
| 取值 | 意义 |
| adCmdText | 指示 strSQL 为命令文本，即普通的 SQL 语句 |
| adCmdTable | 指示 ADO 生成 SQL 查询以便在 strSQL 命名的表中返回所有行 |
| adCmdTableDirect | 指示所作的更改在 strSQL 中命名的表中返回所有行 |
| adCmdStoredProc | 指示 strSQL 为存储过程 |
| adCmdUnknown | 指示 strSQL 参数中的命令类型为未知 |
| adCmdFile | 指示应在 strSQL 命名的文件中恢复保留（保存的）Recordset |
| adAsyncExecute | 指示应异步执行 strSQL |
| adAsyncFetch | 指示在提取 Initial Fetch Size 属性中指定的初始数量后，应该异步提取所有剩余的行。如果所需的行尚未提取，主要的线程将被堵塞直到行重新可用 |
| adAsyncFetchNonBlocking | 指示主要线程在提取期间从未堵塞。如果所请求的行尚未提取，当前行自动移到文件末尾 |

下面是一段利用 Recordset 的 Open()方法打开记录集的代码：

\_Recordset m\_recordSet;

//创建实例

m\_recordSet.CreateInstance(\_\_uuidof(Recordset));

|  |  |  |
| --- | --- | --- |
| //根据连接字符串开启数据连接，得到结果集  m\_recordSet->Open("select \* from Contact",  adLockOptimistic, adCmdText); | m\_connection.GetInterfacePtr(), | adOpenDynamic, |

注意：当比较频繁的访问数据库的时候，最好先用全局\_ConnectionPtr 接口创建一个数据连接，然后用\_ResultsetPtr 接口来处理数据。本章第 6 节的例子便是采用的这种方法。

获得记录集后，可以利用 MoveNext()函数方便的实现记录集的遍历，代码如下：

while(!m\_Recordset->adoEOF)

{

…

//获得下一条记录

m\_Recordset->MoveNext();

}

当然，利用 12.5 节介绍的其他方法也可以较为方便地实现记录集的遍历，读者可以自己

编码实验。

### 12.6.4 对记录的操作

对记录的操作主要有添加、修改和删除 3 种。

1．添加记录

打开记录集后，就可以往里面添加记录了，添加一条记录一般按如下步骤进行。

* 调用 Recordset 对象的 AddNew()方法新增一个空记录。
* 调用 Recordset 对象的 PutCollect()方法输入每个字段的值。
* 调用 Recordset 对象的 Update()方法更新数据库中的数据。

下面是添加记录的一段代码：（假设 m\_ID、m\_Name、m\_Telephone 和 m\_Address 均为

已经赋值的变量）

m\_Recordset->AddNew(); m\_Recordset->PutCollect("ID", m\_ID); m\_Recordset->PutCollect("Name", \_variant\_t(m\_Name)); m\_Recordset->PutCollect("Telephone", m\_Telephone); m\_Recordset->PutCollect("Address", \_variant\_t(m\_Address)); m\_Recordset->Update();

2．修改记录

修改一条记录通常按如下步骤进行。

* 调用 Recordset 对象的 PutCollect()方法给当前记录的每个字段赋值。
* 调用 Recordset 对象的 Update()方法更新数据库中的数据。以下是修改当前记录的代码：

m\_Recordset->PutCollect("ID", m\_NewID);

m\_Recordset->PutCollect("Name", \_variant\_t(m\_NewName)); m\_Recordset->PutCollect("Telephone", m\_NewTelephone); m\_Recordset->PutCollect("Address", \_variant\_t(m\_NewAddress)); m\_Recordset->Update();

3．删除记录利用 Recordset 对象的 Delete()方法很容易删除记录。Delete()方法的原型如下：

Delete(enum AffectedEnum *AffectRecords*);

参数 AffectRecords 是一个枚举型变量，用于指定删除方式。如果为此变量赋值 adAffectCurrent，则只有当前记录被删除；如果赋值为 adAffectedGroup，则符合过滤器的所有行都被删除。下面一段代码将当前记录删除：

m\_Recordset->Delete(asAffectCurrent); m\_Recordset->Update();

### 12.6.5 关闭记录集

当所有操作结束后，应当关闭记录集及当前连接，分别调用 Recordset 对象和 Connection

对象的 Close()方法即可，代码如下：

//关闭记录集

if (m\_recordSet != NULL) m\_recordSet->Close();

//关闭连接

if (m\_connection !=NULL) m\_connection->Close();

## 12.7 利用 ADO 查询并操作数据库实例

实例 12-2：ADO 使用实例。源代码在光盘中“ \12\实例 12-2\AdoTest” 目录下。

前面讲了很多 ADO 的基础知识，读者也应该对 ADO 有了一个基本的了解，并且知道了 ADO 的常用函数和操作数据库的步骤。下面即将进行实际操作，这里将利用上面所讲的知识编写一个程序，达到查询和操作数据库的目的。

在具体编写程序前，还要了解一下经常用到的两种类型，即\_variant\_t 和\_bstr\_t。它们是

两个类，分别继承了 VARIANT 和 BSTR，并增加了一些方法。之所以引入这两种数据类型，是因为 COM 必须设计成跨平台，在 COM 里用不了 CString 类，这时需要有一种更普遍的方式来处理字符串以及其他数据。VARIANT 就是一个巨大的 union，包含了除 char \*以外的所有的数据类型，BSTR 取代了 char\*。利用新引入的这两种类型，会使编程更加方便。比如从结果集 rs 中读取数据，然后放到 List Box 控件 m\_List 中。如果采用新的类型，具体代码如下：

\_variant\_t Holder;

Holder = rs->GetCollect("FIELD\_1"); m\_List.AddString((char\*)\_bstr\_t(Holder));

对比一下没有采用\_variant\_t 和 \_bstr\_t 的代码：

COleVariant covFieldValuel;

VARIANT vFieldValue;

CString Holder;

Rs->GetFieldValue("FIELD\_1", covFieldValue); vFieldValue = (LPVARIANT)covFieldValue; Holder.Format("%s",vFieldValue->pbVal); m\_List.AddString(Holder);

另外在编写程序时要用 try 和 catch，否则 ADO 调用错误有可能使程序崩溃，一定要随

时记得捕捉\_com\_error 例外以及其他错误。

此外，在第 2 节介绍如何设置 ODBC 数据源时提到了利用 COM 组件 DataSourceLocator，用户可以在程序运行的过程中动态设置数据源。至于如何利用此组件，将在程序中详细说明。下面具体介绍该程序的编写过程。 1．创建工程

新建一个 MFC 工程，取名为 AdoTest，在第一步中选择单文档，在最后一步的基类选择

CFormView，分别如图 12-20 和图 12-21 所示。

图 12-20 单文档工程 图 12-21 基类选择 CFormView 类

2．链入 ADO 库文件在文件“ stdafx.h” 中加入如下语句：

#import "C:\Program Files\Common Files\system\ole db\oledb32.dll" no\_namespace

#import "c:\program files\common files\system\ado\msado15.dll" no\_namespace rename("EOF", "adoEOF")

加入第二个语句前面已经讲过原因，加入第一个语句是由于在程序中将要采用 COM 组件 DataSourceLocator。这里需要注意一点，两个 import 语句中都用了 no\_namespace，当引入多个库时，可能会引起名字的冲突。不过由于本程序较小，且只有两个库，所以在这里用 no\_namespace 也可以。

3．界面设计

在显示数据库内容时，可以利用 ActiveX 控件 DataGrid Control，将会大大简化编码的工

作。至于如何做，下面的程序中进行介绍。通过单击“ Project”“ Add To Project”“ Components and Controls”，弹出如图 12-22 所示的对话框，在此对话框中打开“ Registered ActiveX Controls” 文件夹，然后选中“ Microsoft DataGrid Control, Version 6.0(OLEDB)” ，如图 12-23 所示。单击“ Insert” 按钮，在弹出的确认对话框中单击“ 确定” 按钮，然后在弹出的“ Confirm Classes” 对话框中取默认值，如图 12-24 所示，单击“ OK”按钮返回“ Components and Controls Gallery” 对话框，单击“ Close” 按钮即可将 DataGrid Control 加入到工程中。

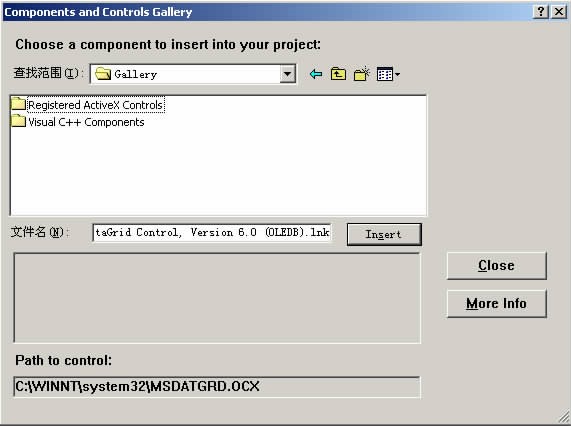


图 12-22 添加组件及控件对话框

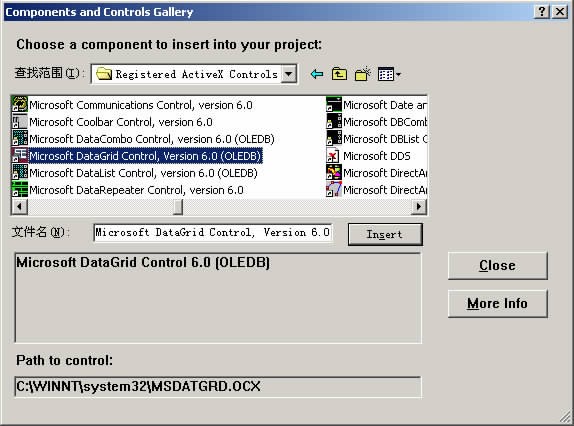
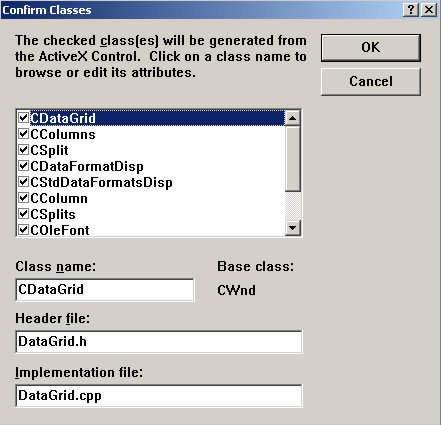
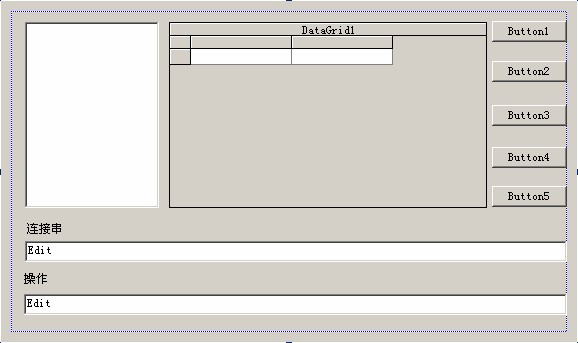


图 12-23 添加 DataGrid 控件对话框

接下来进行界面设计，在资源视图中找到对话框文件夹，然后在 ID 为

IDD\_ADOTEST\_FORM 的对话框中添加一些控件，如图 12-25 所示。其中，最左边是一个 List Box 控件，用来显示数据库中的表项；中间就是上面所加入的 DataGrid ActiveX 控件（在控件栏的最下端可以看到一个红色的标志，这就是 DataGrid Control）；最右边的 5 个按钮 Button1~Button5 分别对应设定数据源、连接数据库、执行 SQL 语句、断开连接和退出；对话框下面的“ 连接串” 和“ 操作” 都是 Static Text 控件；两个 Edit Box 控件分别用来显示连接语句和输入 SQL 语句。

 图 12-24 类确认对话框

 图 12-25 添加控件后的对话框

然后对各控件的属性进行设置，具体设置如表 12-24 所示。

表 12-24 控件属性设置表

|  |  |  |
| --- | --- | --- |
| 控件 | ID | Caption |
| Datagrid 控件 | IDC\_DATAGRID | DataGrid1 |
| List Box 控件 | IDC\_LIST |  |
| Edit 控件 1 | IDC\_CONNECTION\_STRING |  |
| Edit 控件 2 | IDC\_EXECUTE\_STRING |  |
| 按钮控件 1 | IDC\_SOURCE | 数据源 |
| 按钮控件 2 | IDC\_CONNECT | 连接 |
| 按钮控件 3 | IDC\_EXECUTE | 操作 |
| 按钮控件 4 | IDC\_DISCONNECT | 断开 |
| 按钮控件 5 | IDC\_QUIT | 退出 |

4．编写代码

在正式开始编写代码之前，需要进一步明确此程序要实现的功能。用户单击“ 数据源” 按钮，则可以利用 COM 组件 DataSourceLocator 动态设置数据源，同时在 Edit 控件 1 中将连接串显示出来。设定数据源后，用户单击“ 连接” 按钮，程序连接到数据源，同时在 List Box 控件中将数据库中所有表名都显示出来。然后用户可以在 Edit 控件 2 中输入 SQL 语句，单击“ 操作” 按钮对数据库进行操作。当操作完毕后，单击“ 断开” 按钮断开与数据源的连接。最后单击“ 退出” 按钮退出程序界面。同时为了方便用户，程序增加了两个功能。当用户在 List Box 控件中单击某个表名时，在 DataGrid 控件中会相应的显示出此表的所有记录；当用

户在 DataGrid 控件中表的某列单击时，此列按照升序进行排列，再次单击此列，则降序排列。

注意：为了让程序更加简洁，本节在介绍操作数据库时，没有利用 12.6.4 小节介绍的对数据库记录进行操作的方法，而是直接用 SQL 语句操作。读者也可以根据 12.6.4 小节所讲内容自己编写操作记录的代码。

1. 初始化变量

首先在文件“CAdoTestView.h”的开头加入语句#include "datagrid.h"，然后为类

CAdoTestView 添加如表 12-25 所示的成员变量，它们都是 public 成员变量。

表 12-25 类 CAdoTestView 成员变量表

|  |  |
| --- | --- |
| 成员变量 | 功能 |
| \_ConnectionPtr m\_Connection | 连接数据源 |
| \_RecordsetPtr m\_Recordset | 打开记录集 |
| CListBox m\_ListBox | 与控件 List Box 关联 |
| CDataGrid m\_DataGrid | 与控件 DataGrid 关联 |
| CString m\_strConnection | 连接串，与 Edit 控件 1 关联 |
| CString m\_strSQL | 对数据库操作的 SQL 语句，与 Edit 控件 2 关联 |
| CString m\_strTableName | 表名 |
| BOOL isDesc | 是否降序排列 |

然后在 CAdoTestView 类的构造函数中先进行一些初始化工作，代码如下：

|  |
| --- |
| CAdoTestView::CAdoTestView()  : CFormView(CAdoTestView::IDD)  {  //{{AFX\_DATA\_INIT(CAdoTestView)  // NOTE: the ClassWizard will add member initialization here  //}}AFX\_DATA\_INIT  // TODO: add construction code here m\_strConnection = \_T(""); m\_strSQL = \_T(""); m\_strTableName = \_T(""); isDesc = FALSE;  } |

最后要将各控件和它们对应的变量关联起来，代码如下：

void CAdoTestView::DoDataExchange(CDataExchange\* *pDX*)

|  |  |
| --- | --- |
| {    } | CFormView::DoDataExchange(pDX);  //{{AFX\_DATA\_MAP(CAdoTestView)  // NOTE: the ClassWizard will add DDX and DDV calls here  //}}AFX\_DATA\_MAP  DDX\_Control(pDX, IDC\_LISTBOX, m\_strListBox);  DDX\_Control(pDX, IDC\_DATAGRID, m\_strDataGrid);  DDX\_Text(pDX, IDC\_CONNECTION\_STRING, m\_strConnection);  DDX\_Text(pDX, IDC\_EXECUTE\_STRING, m\_strSQL); |

1. 编写各消息响应函数前面已经说过，在编写 ADO 的程序时，要用 try 和 catch，否则 ADO 调用错误有可能使程序崩溃，一定要随时记得捕捉\_com\_error 例外以及其他错误。因此首先编写捕获例外时的处理函数。为类 CAdoTestView 添加 public 成员函数 void GenerateError(HRESULT hr, PWSTR pwszDescription)，代码如下：

|  |
| --- |
| void CAdoTestView::GenerateError(HRESULT *hr*, PWSTR *pwszDescription*)  {  CString m\_strError;  m\_strError.Format("Run-time error ’%d (%x)’", hr, hr); m\_strError += "\n\r"; m\_strError += pwszDescription;  AfxMessageBox(m\_strError);  } |

接下来为各个按钮添加响应函数。首先为用来设定数据源的按钮添加响应函数：打开

“ ClassWizard” 对话框，在“ Class name”下拉菜单中选择 CAdoTestView，在 Object IDs 下拉列表中选择 IDC\_SOURCE，在“ Messages” 下拉列表中选择 BN\_CLICKED，单击“ Add

Function”，采用默认的函数名，然后依次单击“ OK”和“Edit Code”按钮，定位到函数

CAdoTestView::OnSource()，代码如下：

void CAdoTestView::OnSource()

{

// TODO: Add your control notification handler code here // TODO: Add your control notification handler code here

HRESULT hr;

IDataSourceLocatorPtr m\_dlPrompt = NULL;

\_ConnectionPtr m\_Conn = NULL;

//初始化 COM

::CoInitialize(NULL);

|  |  |
| --- | --- |
| } | //创建 IDataSourceLocatorPtr 的实例  hr = m\_dlPrompt.CreateInstance(\_\_uuidof(DataLinks));  //弹出数据连接的对话框  m\_Conn = m\_dlPrompt->PromptNew(); if (m\_Conn!=NULL)  {  //将连接字符串复制到 m\_strConnect 中  m\_strConnection.Format("%s", (char\*)m\_Conn->ConnectionString);  //将变量中的值保存到控件中  UpdateData(FALSE);  } |

编写好设置数据源的代码，接下来该为“ 连接” 按钮编写响应函数了。在编写此函数之前，先要编写数据库表名的函数。因为当连接成功时，在 List Box 控件中会显示出数据库中所有表的名称。为类 CAdoTestView 添加一个函数 void getTables()，它是 public 的，用来显示表名。代码如下：

|  |
| --- |
| void CAdoTestView::getTables()  {  \_bstr\_t tablesNames;  CString kooky;  //先将 List Box 清空  m\_ListBox.ResetContent();  //将控件中的值保存到变量中  UpdateData( TRUE);  try  {  //创建实例  m\_Recordset.CreateInstance(\_\_uuidof(Recordset));  //以只读方式打开结果集，得到表名信息  m\_Recordset = m\_Connection->OpenSchema (adSchemaTables,vtMissing,vtMissing);  //如果结果集没有结束  while (!m\_Recordset->adoEOF)  {  //得到表项的名字  tablesNames = m\_Recordset->GetCollect("TABLE\_NAME"); kooky = (char\*) tablesNames;  //如果是表项，则加到 ListBox 中  if (kooky.Left(4) != "MSys") m\_ListBox.AddString( kooky );  //移到下一个表项 |

|  |  |
| --- | --- |
| } | m\_Recordset->MoveNext();  }  }  //捕获例外\_com\_error  catch (\_com\_error &e)  {  GenerateError(e.Error(), e.Description());  }  //捕捉其他例外  catch(...) {}  //将变量中的值保存到控件中  UpdateData( FALSE );  //最后将结果集置为空  m\_Recordset = NULL; |

现在为“ 连接” 按钮编写响应函数。具体步骤与为“ 数据源” 按钮添加响应函数的一致，

只是 Object IDs 下拉列表中选择 Connect 按钮的 ID：IDC\_CONNECT，代码如下：

void CAdoTestView::OnConnect()

{

// TODO: Add your control notification handler code here

//更新变量值

UpdateData(TRUE); try

{

//创建实例

m\_Connection.CreateInstance(\_\_uuidof(Connection));

//根据连接字符串开启数据连接

m\_Connection->Open( \_bstr\_t( m\_strConnection.GetBuffer(0) ), "", "", -1);

}

//捕获例外\_com\_error

catch (\_com\_error &e)

{

GenerateError(e.Error(), e.Description());

} //捕捉其他例外

catch(...) {}

//将变量中的值保存到控件中

UpdateData( FALSE );

//显示表项名

getTables();

}

按照顺序，为“ 操作” 按钮编写响应函数。代码如下：

|  |
| --- |
| void CAdoTestView::OnExecute()  {  // TODO: Add your control notification handler code here  try  {  //创建实例  m\_Recordset.CreateInstance(\_\_uuidof(Recordset));  //将控件中的值保存到变量中，主要是保存 SQL 语句  UpdateData(TRUE);  //设定光标服务  m\_Connection->CursorLocation = adUseClient;  //根据连接字符串开启数据连接，得到结果集  m\_Recordset->Open(m\_strSQL.GetBuffer(0), m\_Connection.GetInterfacePtr(), adOpenDynamic,  adLockOptimistic, adCmdText);  }  //捕获例外\_com\_error  catch (\_com\_error &e)  {  GenerateError(e.Error(), e.Description());  }  //捕获其他例外  catch (...) {}  //将结果集中的内容在 datagrid 中显示出来  m\_DataGrid.SetRefDataSource((LPUNKNOWN)m\_Recordset);  //刷新 DataGrid  m\_DataGrid.Refresh();  //将变量中的值保存到控件中  UpdateData(FALSE);  //将结果集置空  m\_Recordset = NULL;  } |

在这段代码中，完成了执行 SQL 语句。这是因为记录集转化成 LPUNKNOWN 类型，然后当做参数传给 DataGrid 控件的方法 SetRefDataSource 即可。

至此，程序已经可以完成对数据库的操作了，但是为了让程序更加完善，当操作结束时，应该关闭记录集和连接，然后退出。最后还需要为“ 断开” 和“ 退出” 按钮编写响应函数。

“ 断开” 按钮的响应函数如下：

void CAdoTestView::OnDisconnect()

{

|  |  |
| --- | --- |
| } | // TODO: Add your control notification handler code here try  {  //关闭连接  m\_Connection->Close();  }  //捕获\_com\_error 例外  catch (\_com\_error &e)  {  GenerateError(e.Error(), e.Description());  }  //捕获其他例外  catch(...) {}  //清空 List Box 控件  m\_ListBox.ResetContent();  //将 DataGrid 控件置空  m\_DataGrid.SetRefDataSource(NULL);  //将 DataGrid 控件设置成默认状态  m\_DataGrid.ClearFields();  //刷新 DataGrid m\_DataGrid.Refresh();  //将 SQL 语句清空  //这里不清空连接语句，是为了断开连接以后如果重新连接不需要重新设定数据源  m\_strSQL = "";  //将变量值保存到控件中  UpdateData(FALSE); |

为“ 退出” 按钮编写如下响应函数：

void CAdoTestView::OnQuit()

{

// TODO: Add your control notification handler code here

try

{

//关闭记录集

if (m\_Recordset != NULL) m\_Recordset->Close();

//关闭连接

if (m\_Connection !=NULL)

m\_Connection->Close();

}

|  |  |
| --- | --- |
| } | //捕获例外  catch(...){}  //退出  PostMessage(WM\_QUIT); |

至此，基本功能已经全部完成。为了让用户操作起来更加方便，下面介绍为添加的辅助

功能编写代码。

（3）为附加功能编写代码首先，当用户单击 List Box 控件中的某一表项时，在 DataGrid 控件中会相应地出现此表中所有的记录。当用户在 List Box 中单击一项时，会激发 LBN\_SELCHANGE 消息，下面为此消息添加处理函数。打开“ ClassWizard” 对话框，在“Class name” 下拉菜单中选择

CAdoTestView，在“ Object Ids” 下拉列表中选择 IDC\_LISTBOX，在“ Messages” 下拉列表中选择 LBN\_SELCHANGE，单击“ Add Function” ，采用默认的函数名，然后依次单击“ OK” 和“ Edit Code” 按钮，定位到函数 CAdoTestView::OnSelchangeListbox()，代码如下：

void CAdoTestView::OnSelchangeListbox()

{

// TODO: Add your control notification handler code here

//得到鼠标单击项在 List Box 中的索引值

int index = m\_ListBox.GetCurSel();

//根据索引值得到表项的名字

m\_ListBox.GetText(index, m\_strTableName);

//构造 SQL 查询语句

m\_strSQL = "select \* from [" + m\_strTableName + "]";

//将变量值保存到控件中

UpdateData(FALSE);

//进行查询

OnExecute();

}

接下来实现单击某列进行排序的功能。在程序中要用到两个类：CColumn 和 CColumns。

在 CAdoTestView.cpp 的开头，需要加入语句#include "column.h" 和 #include "columns.h"，将头文件包含进来。

当在 DataGrid 中单击某列时，会激发 HeadClick 消息。下面为此消息添加响应函数：打开“ ClassWizard” 对话框，在“ Class name” 下拉菜单中选择 CAdoTestView，在“ Object Ids” 下拉列表中选择 IDC\_DATAGRID，在“Messages” 下拉列表中选择 HeadClick，单击“ Add Function” ，采用默认的函数名，然后依次单击“ OK” 和“ Edit Code” 按钮，定位到函数 void

CAdoTestView::OnHeadClickDatagrid(short ColIndex)，代码如下：

void CAdoTestView::OnHeadClickDatagrid(short *ColIndex*)

{

// TODO: Add your control notification handler code here

|  |  |  |  |
| --- | --- | --- | --- |
|  | //得到  //  //值为  //  //  //  //  { | DataGrid  VARIANT index;  index  类型为整型  index.vt = VT\_I4; 得到单击列  得到此列的名字  //  {            } | 中表示各列信息的类  CColumns cols = m\_DataGrid.GetColumns(); 此参数用来得到具体某一列  index.intVal = ColIndex;      CColumn col = cols.GetItem(index);    m\_strTableName = col.GetCaption();  排序必须是在已经建立连接的情况下  if(m\_Connection != NULL)  排序还必须在表名存在的情况下  if(m\_strTableName.GetLength() != 0)  //得到查询语句的长度，后面会用到  int nOriginalLength = m\_strSQL.GetLength();  //构造新的查询语句  CString strTmp = " order by [" + m\_strTableName + "]"; if(isDesc)  {  strTmp = strTmp + " desc"; isDesc = FALSE;  } else  isDesc = TRUE;  m\_strSQL += strTmp;  //将变量值保存到控件中  UpdateData(FALSE);  //对新的查询语句进行查询  OnExecute();  //恢复原查询语句，这样做是为了再单击别的列，可以对别的列进行排序  m\_strSQL.Delete(nOriginalLength, strTmp.GetLength()); |
| } | } | | |

至此，所有的代码均已编写完毕。这段代码已经实现了对数据库的基本操作。可见，用 ADO 技术进行数据库开发非常方便，且简单易学。同时，DataGrid 控件的使用可以使得程序员不再为数据的显示做很多繁琐的事情，从而大大简化了代码的编写。

（4）运行程序

对程序编译、链接，然后运行（这里要注意，在编译的时候会出现如下的警告：

msado15.tlh(403) : warning C4146: unary minus operator applied to unsigned type, result still

unsigned。对此微软在 MSDN 中作了说明，并建议不要理会这个警告），可以看到运行界面如图 12-26 所示。

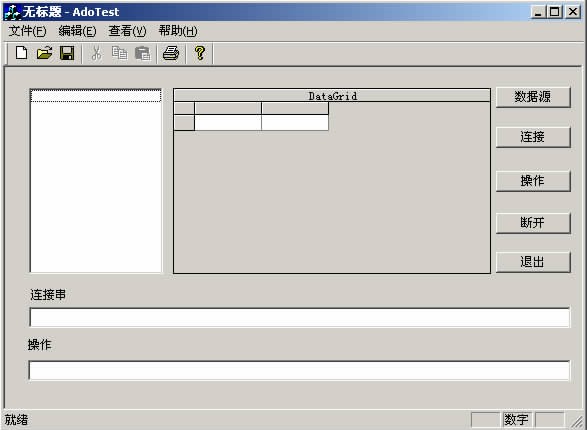


图 12-26 程序运行界面

单击“ 数据源” 按钮，打开如图 12-27 所示的“ 数据链接属性” 对话框。单击“ 提供者” 选项卡。

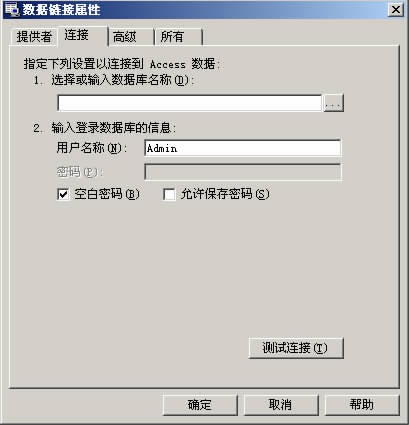
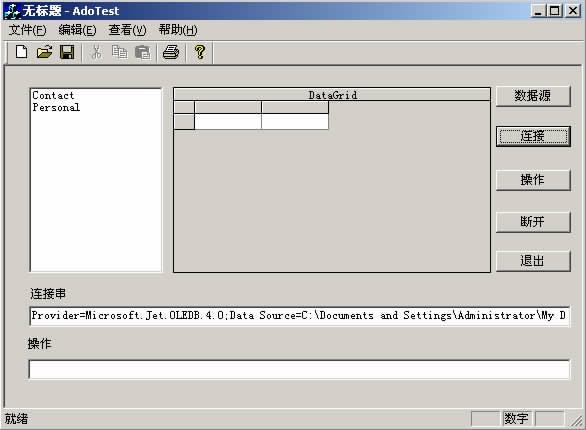
 

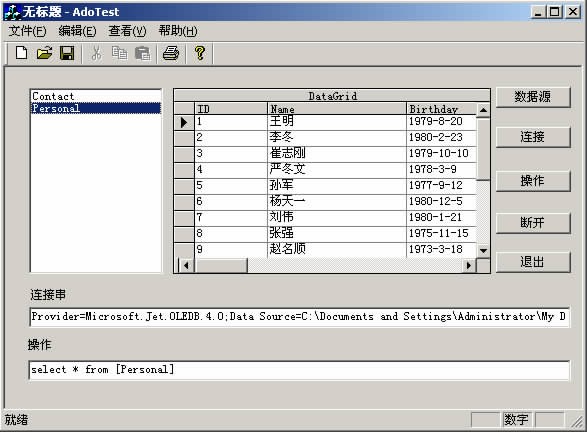
图 12-27 “ 数据源链接属性” 对话框 图 12-28 连接 Access 数据

本例中，用来测试的数据库是在 Access 2002 下建立的，在这里要选择“ Microsoft Jet 4.0 OLE DB Provider” ，如果是在 Access 97 下建立的，则要选择“ Microsoft Jet 3.51 OLE DB Provider” ，如果是其他类型数据库，也有相应的 Provider 可以选择。然后单击“ 下一步” 按钮，对话框如图 12-28 所示。在这里数据库选择第 2 节建好的数据库 Info.mdb，单击“ 确定” 按钮，数据源设置完成。可见，利用 COM 组件 DataSourceLocator 设置数据源是非常方便的，而在编写程序的时候，也只需要一个 PromptNew()函数即可。

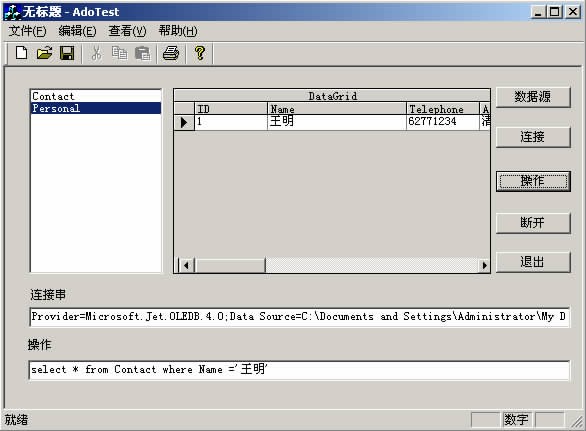
设好数据源后回到原运行界面，会发现在第一个 Edit 控件栏中已经填好了连接语句。单击“ 连接”按钮，在左侧的 List Box 控件栏中会出现 Info.mdb 数据库中的“ Contact”和“ Personal” 两个表的名字，如图 12-29 所示。

 图 12-29 刚连接时的界面

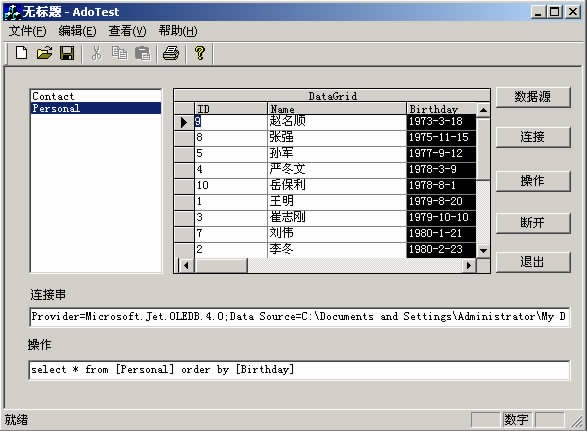
如果想看表 Personal 的内容，只需要用鼠标左键单击 List Box 控件栏中的“ Personal”项，则在 DataGrid 控件中便会显示出表 Personal 中的所有记录，如图 12-30 所示。

 图 12-30 显示表 Personal 中的所有记录

如果需要记录中的详细资料，如王明的联系方式，可利用第 3 节介绍的 SQL 语句，只需要在第 2 个 Edit 控件栏中填写 select \* from Contact where Name = ’王明’，然后单击“ 操作” 按钮，则在 DataGrid 控件中会显示出表 Contact 中关于王明的记录，如图 12-31 所示。

 图 12-31 利用 SQL 语句进行数据库操作

最后可以测试一下排序功能。先显示出表 Personal 中的所有记录，然后在 DataGrid 控件的“ Birthday” 列上单击鼠标左键，可以看到按出生年月排序后的结果，如图 12-32 所示。

 图 12-32 对某列排序后的结果

用户还可以利用第 3 节介绍的其他 SQL 语句对数据库进行操作。当所有操作完成时，单击“ 断开” 按钮断开连接，然后单击“ 退出” 按钮退出程序。

本程序代码量只有大概 300 行，但是已经实现了对数据库的基本操作。由此可见，利用 ADO 进行数据库开发简单方便，同时 DataGrid 控件的使用也大大简化了编程。相信通过这个简单的例子，读者已经对 ADO 技术有了更深一步的了解。

## 12.8 本章小结

本章介绍了数据库的基本概念和数据库开发的几种主要技术，接着较为详细地介绍了 Access 数据库的创建和维护以及 SQL 语句的基本知识，最后，用一个详细的实例介绍了如何利用 ADO 技术进行数据库开发。本章所用实例都简单易懂，希望读者学习完本章知识后，能够对数据库开发有一个全面的了解，并能够利用 ADO 技术进行简单的数据库开发。