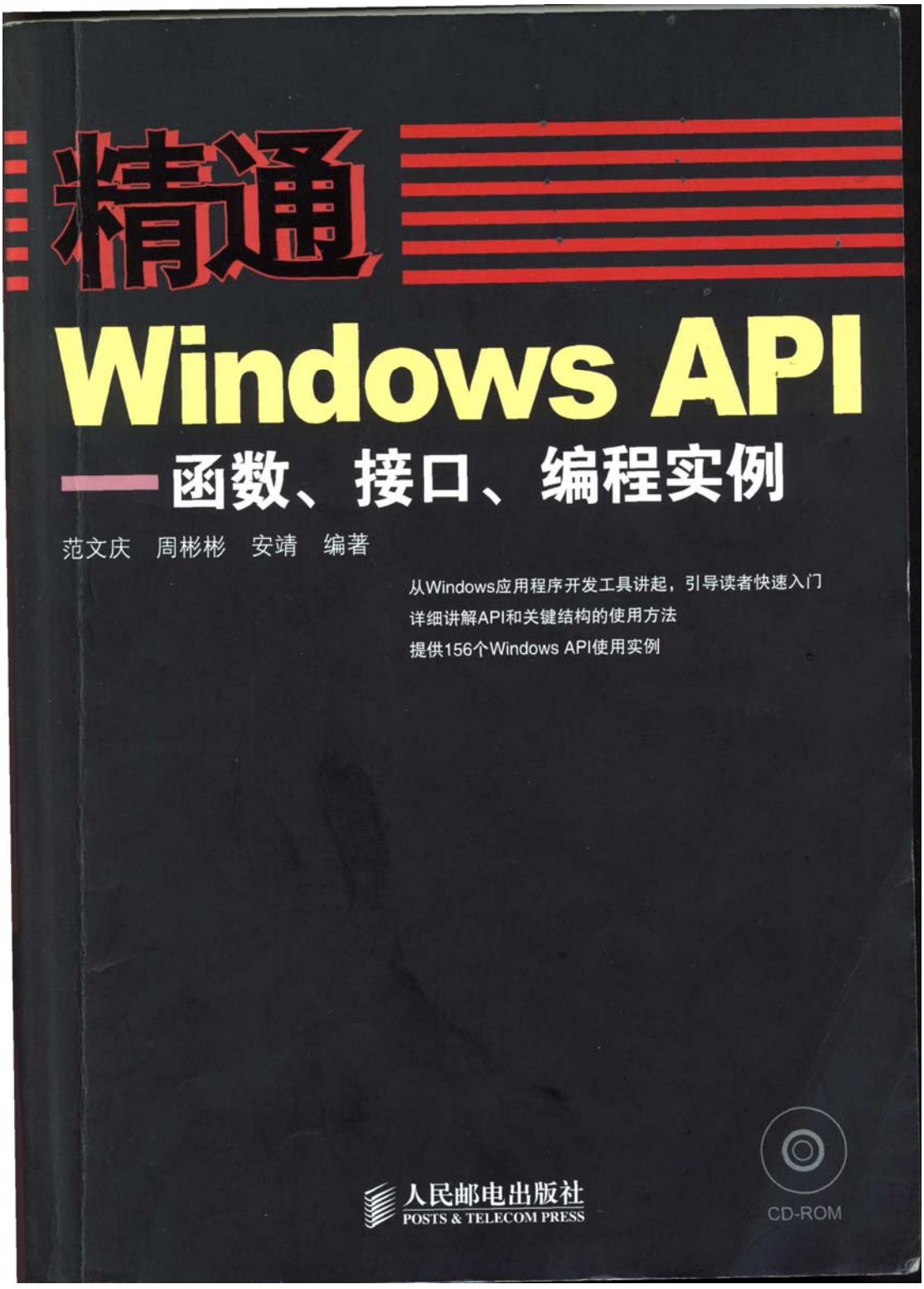
精通
Windows API
——函数、接口、编程实例
范文庆 周彬彬 安靖 编著
从Windows应用程序开发工具讲起，引导读者快速入门
详细讲解API和关键结构的使用方法
提供156个Windows API使用实例
人民邮电出版社
POSTS & TELECOM PRESS
CD-ROM

精通

Windows API

—函数、接口、编程实例

范文庆周彬彬安靖编著

人民邮电出版社

北京

图书在版编目(C I P)数据
精通 Windows API：函数、接口、编程实例 / 范文庆，
周彬彬，安靖编著. 一北京：人民邮电出版社，2009.2
ISBN 978-7-115-19095-6
I.精…II.①范…②周…③安…III,窗口软件,Win-dows -软件接口一程序设计IV. TP316.7
中国版本图书馆 CIP 数据核字(2008)第 168772 号

内容提要

Windows API (Windows Application Program Interface，Windows 应用程序接口)是一系列函数、宏、数据类型、数据结构的集合，运行于 Windows 系统的应用程序，可以使用操作系统提供的接口来实现需要的功能。

本书由浅入深、循序渐进地教授读者如何使用 Windows API 进行 Windows 应用程序开发。全书共 18 章，分为 3 个部分，第 1 部分（第 1 章～第 3 章）介绍 Windows 程序设计基础；第 2 部分（第 4 章～第 17 章）按照程序设计的各个方面进行划分，包括文件系统、内存管理、进程与线程、用户界面、Shell 程序开发、Windows GDI、Socket 网络通信驱动程序开发、安全机制等内容；第 3 部分（第 18 章）作为全书的总结和补充。

本书适合广大的 Windows 应用程序开发人员、Visual C++开发工程师、网络游戏开发人员、软件培训机构学员和高校学生阅读。

精通 Windows API——数、接口、编程实例
◇编　著范文庆周彬彬安靖
责任编辑屈艳莲
执行编辑黄焱
◇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
北京鑫正大印刷有限公司印刷
◇开本：787x1092　1/16
印张：35
字数：922 千字　　2009 年 2 月第 1 版
印数：1-4 000 册　　2009 年 2 月北京第 1 次印刷
ISBN 978_7_115_19095_6 / rIP
定价：69.00 元（附光盘）

前言

关于 Windows API

Windows API (Windows Application Program Interface)是一系列函数、宏、数据类型、数据结构的集合。运行于 Windows 系统的应用程序可以使用操作系统提供的接口来实现应用程序需要的功能。Windows 开发平台是所有程序开发平台中非常重要的一种，是程序设计中一个重要的方面。本书介绍了 Windows 应用程序开发的各个方面。

本书的写作方法

程序设计是一门实践性很强的学科，本书在编写过程中十分注重与实际开发工作相联系，在讲解每一个编程要点时，都以实例为核心进行分析、演示，并把实际工程中开发人员可能遇到的问题以实例的方式向读者讲解。

与此同时，本书同样重视基础知识的讲解，重视原理的说明，对程序所使用的每一个 API 和每一个结构的使用方法都尽量详细说明。本书还在第 18 章详细介绍了 Windows 系统调用的内部过程。

解决开发人员实际编码过程中可能出现的问题也是本书的一个主要目的，因此本书使用了大量篇幅来讲解编译选项等最容易被忽略，但是却是最容易在程序设计中出现问题，也是常常使得程序员不知道如何解决的问题。

本书主要内容

本书共分 18 章，涵盖了 Windows 应用程序设计的基本内容。第 1、2、3 章是 Windows 程序设计的基础。第 4 章～第 17 章按程序设计的方面划分，每个方面一章。第 18 章是全书的总结和提高。

第 1 章主要讲解如何配置 Windows 应用程序的开发工具，从最简单的实例入手，使读者对 Windows 应用程序的开发有一个初步的认识。

第 2 章对 Windows API 作了简要的介绍，包括 Windows API 基本的功能分类、数据类型等。

第 3 章介绍了开发工具配置与使用，包括基本的编译链接工具 cl.exe、rc.exe、luk.exe，Platform

SDk 的使用，Makefile 的编写以及使用 WinDbg 的调试方法等。

第 4 章文件系统，主要介绍了磁盘和驱动器管理、文件和目录管理、内存映射文件等内容。

第 5 章内存管理，主要介绍了内存管理原理、堆原理、全局和局部内存管理、虚拟内存管理、内存操作与内存消息管理等内容。

第 6 章进程、线程和模块，主要介绍了进程管理，线程、纤程的创建、删除以及获取方法，获取进程状态信息，动态链接库的加载、释放和获取方法等内容。

第 7 章线程同步，主要介绍了线程同步的基本原理、同步对象等内容。

第 8 章服务，主要介绍了基本概念、服务程序的编写、对服务程序的控制和管理等内容。

第 9 章图形用户界面，主要介绍了字符界面程序，窗口、控件、菜单、对话框等界面元素的使用方法。

第 10 章系统信息的管理，主要介绍了系统基本信息、时间信息以及注册表信息的管理。

第 11 章进程间通信，主要介绍了邮槽、管道、剪贴板的创建与使用方法，数据复制消息的使用方法等内容。

第 12 章 Windows Shell 程序设计，主要介绍了 Windows Shell 的目录管理、文件协助、Shell 扩展等内容。

第 13 章 Windows GDI，主要介绍了使用 GDI 指定文字的字体、绘制线条、绘制图形、对位图的操作、坐标变换的方法等内容。

第 14 章网络通信与配置，主要介绍了 Socket 通信的相关内容以及 IP Helper 的使用方法。

第 15 章程序安装与设置，主要介绍了 cab 文件的创建、INF 文件的创建、安装程序 setup.exe 的编写、使用 msi 文件进行安装等内容。

第 16 章设备驱动管理与内核通信，主要介绍了与设备有关的 API 函数、如何对设备驱动进行控制、如何编写设备驱动程序等内容。

第 17 章用户、认证和对象安全，主要介绍了数据认证中的基本概念、安全机制程序示例以及用户的增加、删除、权限更改等内容。

第 18 章 Windows API 的内部原理，主要介绍了 Windows 系统中的对象封装、x86 平台程序函数调用、可执行程序结构与 API 函数接口内部机理等内容。

参与本书编写的人员

本书由范文庆、周彬彬、安靖负责编写并统编全书稿，另外感谢以下人员为本书所做的工作：

张墨、郭永红、周瑜、王建伟、孙琼、田旭、范文庆、钟金鑫、王欣、张曦文、尚玉珊、张丛辉、王玮、刘超、张圣亮、李凡、马垄、徐路迎、赵国锋、孙颂武、汪荷君、孙明、林雪梅、黄惠英、刘雯等。

由于时间仓促，加之水平有限，书中不足之处在所难免，敬请读者批评指正。本书责任编辑的联系方式是 huangyan@ptpress.com.cn，欢迎来信交流。

编者

2009 年 1 月

目　录

第 1 章　Windows应用程序开发入门 16
1.1　第一个实例程序 16
1.1.1　sta rt.exe 16
1.1.2　Windows API 17
1.1.3　程序入口函数 17
1.1.4　start.c代码分析 17
1.2　编译代码 18
1.2.1　安装Visual Studio 18
1.2.2　安装Microsoft Platform SDK 19
1.2.3　集成Microsoft Platform SDK与Visual C++速成版 20
1.2.4　Vista SDK与Visual Studi0 2008 21
1.2.5　Visual Studio专业版或团队系统版 22
1.2.6　使用图形化IDE建立工程、进行编译 22
1.2.7　“解决方案”与“工程” 23
1.2.8　使用命令行工具编译 23
第 2 章　Windows API概要 24
2.1　Windows数据类型 24
2.1.1　Windows数据类型示例 25
2.1.2　Windows数据类型与标准C数据类型的关系 28
2.1.3　Windows数据类型与Windows API 28
2.1.4　Windows中的数据结构 29
2.2　Windows API的功能分类 29
2.2.1　系统基本服务 29
2.2.2　系统管理 31
2.2.3　用户界面 31
2.2.4　图像和多媒体 34
2.2.5　网络 34
2.2.6　系统安全 34
2.2.7　其他功能 35
2.3　Windows API核心DLL 35
2.3.1　Kerne132.dll 35
2.3.2　User32.dll 36
2.3.3　Gdi32.dll 36
2.3.4　标准C函数 36
2.3.5　其他Dll 36
2.4　Unicode和多字节 36
2.4.1　W版本和A版本的API 38
2.4.2　Unicode与ASGII的转换 39
2.5　对Windows程序设计规范的建议 39
第 3 章　开发工具配置与使用 41
3.1　使用Visual C／C++编译链接工具 41
3.1.1　编译器cl.exe 41

3.1.2 资源编译器rc.exe 46
3.1.3 链接器link.exe 47
3.1.4 其他工具 53
3.1.5 编译链接工具依赖的环境变量 54
3.1.6 示例：使用/D选项进行条件编译 57
3.2 使用Platform SDK 58
3.2.1 Platform SDK的目录结构与功能 58
3.2.2 为编译链接工具设置环境变量 59
3.2.3 Platform SDK工具集 61
3.2.4 Windows Vista SDK 63
3.3 编写Makefile 63
3.3.1 使用nmake.exe构建工程 63
3.3.2 Makefile实例 65
3.3.3 注释 65
3.3.4 宏 66
3.3.5 描述块：目标、依赖项和命令 68
3.3.6 makefile预处理 69
3.3.7 在Platform SDK的基础上使用nmake 70
3.4 使用WinDbg调试 72
3.4.1 安装WinDbg 72
3.4.2 编译可调试的程序 73
3.4.3 WinDbg命令 74
3.4.4 调试过程演示 74
3.5 集成开发环境Visual Studio 77
3.5.1 工程类型选择与配置 77
3.5.2 Visual Studio快捷方式 79
3.5.3 生成项目 79
3.5.4 调试 79
3.5.5 选项与设置 80
3.6 开发环境配置总结 81
第 4 章 文件系统 82
4.1 概述 82
4.1.1 文件系统的基本概念 82
4.1.2 文件系统主要API 83
4.2 磁盘和驱动器管理 85
4.2.1 遍历卷并获取属性 85
4.2.2 操作驱动器挂载点 91
4.2.3 判断光驱中是否有光盘 96
4.2.4 获取磁盘分区的总容量、空闲容量、簇、扇区信息 98
4.3 文件和目录管理 101
4.3.1 删除、复制、重命名、移动文件 102
4.3.2 创建、打开、读写文件，获取文件大小 105
4.3.3 创建目录 111
4.3.4 获取程序所在的目录、程序模块路径，获取和设置当前目录 112

4.3.5 查找文件、遍历指定目录下的文件和子目录 115
4.3.6 递归遍历目录树 118
4.3.7 获取、设置文件属性和时间 120
4.4 内存映射文件 125
4.4.1 使用Mapping File提高文件读写的效率 125
4.4.2 通过Mapping File在进程间传递和共享数据 130
4.4.3 通过文件句柄获得文件路径 133
4.5 总结 136
第 5 章 内存管理 137
5.1 Windows内存管理原理 137
5.1.1 基本概念 137
5.1.2 分页与分段内存管理、内存映射与地址转换 138
5.1.3 进程的内存空间 140
5.1.4 虚拟内存布局、内存的分工、堆与栈 142
5.1.5 内存的保护属性和存取权限 142
5.1.6 本章API列表 142
5.2 堆管理 144
5.2.1 获取堆句柄、分配与再分配堆 144
5.2.2 获取堆中内存块的大小信息 148
5.2.3 释放内存、销毁堆 149
5.3 全局(Global)和局部(Local)内存管理 151
5.3.1 Global函数 151
5.3.2 Local函数 152
5.3.3 使用全局和局部函数分配和释放内存、改变内存块属性 153
5.4 虚拟内存管理 154
5.4.1 虚拟地址空间与内存分页 154
5.4.2 分配和释放可读可写的虚拟内存页面 154
5.4.3 修改内存页面状态和保护属性、将页面锁定在物理内存中 157
5.4.4 管理其他进程的虚拟内存 158
5.5 内存操作与内存信息管理 159
5.5.1 复制充、移动、清零内存块、防止缓冲区溢出 159
5.5.2 获得当前系统内存使用情况 161
5.5.3 判断内存指针的可用性 162
5.6 各种内存分配方式的关系与比较 163
5.6.1 标准C内存管理函数与Windows内存管理API的关系 163
5.6.2 功能性区别 164
5.6.3 效率的区别 164
第 6 章 进程、线程和模块 165
6.1 基本概念 165
6.1.1 应用程序与进程 165
6.1.2 控制台应用程序与图形用户界面应用程序 166
6.1.3 动态链接库、模块 166
6.1.4 线程、纤程与作业 167
6.1.5 权限与优先级 167

6.2 进程管理.....167
6.2.1 创建进程、获取进程相关信息、获取启动参数.....168
6.2.2 编写控制台程序和图形用户界面应用程序.....173
6.2.3 获取和设置环境变量.....173
6.3 线程、纤程.....176
6.3.1 创建线程、退出线程、获取线程信息.....177
6.3.2 挂起、恢复、切换、终止线程.....179
6.3.3 创建远程线程、将代码注入其他进程中执行.....182
6.3.4 创建纤程、删除纤程、调度纤程.....185
6.3.5 纤程与线程的互相转换.....186
6.4 进程状态信息.....191
6.4.1 PS API与Tool help API.....191
6.4.2 遍历系统中的进程.....193
6.4.3 列举进程的模块、线程.....197
6.4.4 进程的堆使用、内存占用、虚拟内存大小，页面错误情况.....199
6.5 动态链接库.....200
6.5.1 加载、释放DLL、通过句柄获取DLL相关信息.....201
6.5.2 编写动态链接库、导出函数.....201
6.5.3 创建动态链接库工程，配置DLL编译链接选项.....203
6.5.4 运行时动态获取DLL导出函数地址并调用.....204
6.5.5 声明导出函数、创建lib库，为其他模块提供导入表调用接口.....205
6.5.6 通过构建导入表调用DLL导出函数.....206
第 7 章 线程同步.....207
7.1 基本原理.....207
7.1.1 线程同步的过程.....207
7.1.2 同步对象.....208
7.1.3 等待函数.....208
7.2 同步对象示例.....209
7.2.1 使用事件对象(Event).....209
7.2.2 使用互斥对象(Mutex).....214
7.2.3 使用信号量控制访问共享数据的线程数量.....217
7.2.4 使用可等待计时器(Timer).....221
7.3 等待进程和线程的执行完成.....224
第 8 章 服务.....225
8.1 基本概念.....225
8.1.1 服务控制器(SCM).....225
8.1.2 服务程序.....226
8.1.3 服务控制管理程序.....226
8.1.4 系统服务管理工具.....226
8.1.5 服务的属性.....226
8.2 编写服务程序.....227
8.2.1 入口函数.....227
8.2.2 服务主函数.....227
8.2.3 控制处理函数.....228

8.3 实现对服务的控制和管理.....231
8.3.1 创建、删除服务.....231
8.3.2 启动、停止服务，向服务发送控制请求.....234
8.3.3 管理服务状态、配置服务、服务的依赖关系.....237
第 9 章 图形用户界面.....244
9.1 字符界面程序.....244
9.1.1 基本概念.....244
9.1.2 控制台读写.....246
9.1.3 控制台字体、颜色等属性，操作屏幕缓存.....249
9.1.4 控制台事件.....259
9.2 图形用户界面：基本概念.....262
9.2.1 窗口.....262
9.2.2 窗口类.....263
9.2.3 消息和消息处理函数.....263
9.2.4 控件.....263
9.2.5 资源.....264
9.2.6 对话框.....264
9.3 图形用户界面：窗口.....264
9.3.1 注册窗口类.....266
9.3.2 创建窗口.....267
9.3.3 窗口消息处理函数.....269
9.3.4 窗口属性、位置和大小.....273
9.3.5 窗口显示方式.....274
9.3.6. 线程消息队列和消息循环.....274
9.4 图形用户界面：控件.....275
9.4.1 Tree View控件.....275
9.4.2 为Tree View控件增加节点.....277
9.4.3 Tree View右键菜单.....278
9.4.4 List View控件.....280
9.4.5 为List View控件增加分栏.....282
9.4.6 为List View控件增加项.....283
9.4.7 文本框控件.....284
9.4.8 为文本框控件设置文字.....285
9.5 界面资源.....286
9.5.1 资源脚本(．rc).....286
9.5.2 资源ID定义和头文件.....289
9.5.3 在程序中使用资源.....289
9.6 菜单.....290
9.6.1 菜单资源和菜单句柄.....290
9.6.2 动态增加、删除、设置菜单及菜单项.....291
9.6.3 菜单消息处理.....292
9.7 对话框.....292
9.7.1 创建对话框.....292
9.7.2 对话框消息处理函数.....293

第 10 章　系统信息的管理.....295
10.1　Windows系统信息.....295
10.1.1　获取系统版本.....295
10.1.2　获取计算机硬件信息.....297
10.1.3　获取系统目录等信息.....299
10.1.4　用户名、计算机名、域名.....300
10.1.5　处理系统颜色信息、尺度信息等.....302
10.1.6　鼠标、键盘等外设信息.....303
10.2　时间信息.....304
10.2.1　设置、获取系统时间.....304
10.2.2　获取开机至现在持续的时间.....305
10.2.3　文件时间与系统时间的转换.....305
10.3　注册表.....306
10.3.1　注册表的作用及组织形式.....306
10.3.2　键、子键、键属性及键值的相关操作.....307
10.3.3　列举注册表项及键值.....309
10.3.5　设置随程序启动而启动的调试器（任何程序）.....312
10.3.6　指定程序崩溃实时调试器.....312
第 11 章　进程间通信.....313
11.1　邮槽(MailSlot).....313
11.1.1　创建邮槽、从邮槽中读取消息.....313
11.1.2　通过邮槽发送消息.....317
11.2　管道(Pipe).....318
11.2.1　创建命名管道.....318
11.2.2　管道监听.....320
11.2.3　使用异步I/O进行读写.....321
11.2.4　关闭管道实例.....325
11.2.5　客户端.....325
11.3　剪贴板.....328
11.3.1　获取、设置剪贴板数据.....328
11.3.2　监视剪贴板.....334
11.3.3　剪贴板数据格式.....343
11.4　数据复制消息(WM_COPYDATA).....345
11.4.1　数据发送端.....345
11.4.2　数据接收端.....348
11.5　其他进程间通信方式.....350
11.5.1　动态数据交换(DDE)和网络动态数据交换(NDDE).....350
11.5.3　Windows Socket.....351
第 12 章　Windows Shell程序设计.....352
12.1　Windows Shell目录管理.....352
12.1.1　Shell对目录和文件的管理形式.....353
12.1.2　“我的文档”等特殊目录相关操作.....353
12.1.3　绑定、遍历、属性获取.....354
12.1.4　浏览文件对话框.....357

12.2 文件协助(File Associations)....358
12.2.1 文件类型相关注册表键值....359
12.2.2 为文件指定默认打开程序....359
12.2.3 定制文件类型的图标....360
12.3 Shell扩展....361
12.3.1 对象及概念....361
12.3.2 CLSID，处理例程的GUID....362
12.3.3 注册Shell扩展....363
12.3.4 COM程序开发基础....364
12.3.5 编写Handler程序....364
12.3.6 Shell扩展程序的调试....380
12.3.7 总结....381
12.4 任务栏通知区域(Tray)图标....381
12.4.1 创建图标窗口....382
12.4.2 创建图标和图标菜单....386
12.4.4 弹出气泡通知....391
12.4.5 动态图标....393
12.4.6 其他功能....395
第 13 章 Windows GDI....398
13.1 GDI编程接口概述....398
13.1.1 Windows GDI的功能....398
13.1.2 链接库与头文件....398
13.2 设备上下文(DC)＼输出操作与图形对象....398
13.2.1 设备上下文类型与关联设备....399
13.2.2 图形对象的作用及与DC的关系....399
13.2.3 各类图形对象的具体属性与作用....401
13.2.4 绘制、填充、写入等图形输出操作....402
13.2.5 修剪与坐标变换....403
13.2.6 设备上下文的图形模式....403
13.3 一个最简单的GDI程序....404
13.3.1 示例....404
13.3.2 DC的操作....406
13.3.3 颜色的表示....407
13.3.4 图形对象：画刷和画笔....407
13.3.5 输出操作：绘制图形和线条....409
13.4 文字和字体....409
13.4.1 选择、设置字体....412
13.4.2 选择字体图形对象....413
13.4.3 文字的颜色....413
13.4.4 输出文字....413
13.4.5 DC图形模式设置....414
13.4.6 遍历字体....415
13.4.7 为系统安装、删除字体文件....417
13.5 绘制线条....417

13.5.1 选择画笔对象 417
13.5.2 直线 418
13.5.3 绘制任意曲线 418
13.5.4 跟踪鼠标轨迹 418
13.5.5 弧线 424
13.6 绘制图形 424
13.6.1 填充颜色与边缘勾勒 425
13.6.2 绘制矩形、椭圆、圆角矩形 425
13.6.3 椭圆弓形和椭圆扇形 431
13.6.4 多边形 431
13.6.5 REGT结构及对RECT的操作 432
13.7 位图操作 434
13.7.1 截取屏幕、保存位图文件 434
13.7.2 将位图显示在界面上 439
13.8 区域(Regions)、路径(Paths)与修剪(Clip)操作 442
13.8.1 区域的创建及形状、位置等属性 442
13.8.2 区域边沿、区域填充、反转与勾勒操作 443
13.8.3 组合、比较、移动等操作 446
13.8.4 点击测试(Hit Testing) 447
13.8.5 路径的创建与操作 451
13.8.6 路径转换为区域 452
13.8.7 使用区域和路径进行修剪操作，限制输出 452
13.9 坐标变换 458
13.9.1 缩放 459
13.9.2 旋转 460
13.10 调色板 460
第 14 章 网络通信与配置 463
14.1 Socket通信 463
14.1.1 客户端 464
14.1.2 服务端 469
14.1.3 处理并发的客户端连接 475
14.1.4 网络通信的异步I/O模式 476
14.2 IP Helper 476
第 15 章 程序安装与设置 483
15.1.1 makecab.exe 483
15.1.2 压缩多个文件 484
15.1.3 Cabinet软件开发工具包(CABSDK) 485
15.2 编写INF文件 486
15.2.1 INF文件格式 486
15.2.2 Install节 488
15.2.3 CopyFiles和AddReg等安装过程 488
15.2.4 源路径和目的路径 489
15.2.5 字符串表 489
15.3 安装程序setup.exe的编号 489

15.4 使用msi文件进行安装 493
15.4.1 Windows Installer Service 493
15.4.2 msi文件的创建与修改工具orca.exe 495
15.4.3 准备工作 496
15.4.4 编辑表组 496
第 16 章 设备驱动管理与内核通信 497
16.1 设备管理 497
16.1.1 列举设备接口 497
16.1.2 监控设备的加载和卸载 504
16.2 I/o控制、内核通信 509
16.2.1 加载驱动程序 509
16.2.2 控制驱动程序、与驱动程序进行通信 516
16.3 编写设备驱动程序 519
16.3.1 驱动程序开发包：DDK 520
16.4 I/O模式，同步与异步 525
第 17 章 用户、认证和对象安全 527
17.1 基本概念 527
17.1.1 访问令牌、权限和用户标识 527
17.1.2 进程的系统操作权限 528
17.1.3 安全对象 528
17.1.4 访问控制列表(ACL) 529
17.2 安全机制程序示例 530
17.2.1 列举进程访问令牌内容和权限 530
17.2.2 修改进程的权限 536
17.2.3 列举安全对象的安全描述符 537
17.2.4 修改安全描述符 543
17.3 用户 544
17.3.1 创建用户 544
17.3.2 用户组 545
17.3.3 删除用户 546
17.3.4 列举用户和用户组、获取用户信息 547
第 18 章 Windows API的内部原理 554
18.1 关于API的补充说明 554
18.1.1 Windows API的版本演进和Vista新增API 554
18.1.2 64 位操作系统的接口 555
18.2 Windows系统中的对象封装 555
18.2.1 什么是对象 555
18.2.2 面向对象的思想 555
18.2.3 Windows系统中的对象：内核对象、GDI对象等 556
18.3 WiFidows程序设计参考：文档资源与样例代码 556
18.3.1 SDK文档和MSDN 556
18.3.2 SDK示例代码 556
18.4 x86 平台程序函数调用原理 557
18.4.1 函数调用的真实过程 557

18.4.2 函数调用约定.....561
18.4.3 为什么通过参数返回数据时只能使用指针.....562
18.4.4 缓冲区溢出.....562
18.4.5 程序运行错误的调试技巧.....562
18.5 可执行程序结构与AIPI函数接口内部机理.....563
18.5.1 Windows可执行程序结构.....563
18.5.2 导入表、导出表、动态链接.....565
18.5.3 NTDLL.DLL、NATIVE API和SSDT.....566
18.5.4 API HOOK.....568
18.6 发布程序.....569
18.6.1 合理选择编译链接选项.....569
18.6.2 构建到指定路径.....569
18.7 模块化，向Windows APl学习接口定义.....569
18.7.1 lib文件.....569
18.7.2 头文件.....570
18.7.3 为第三方应用软件提供SDK.....570

第 1 章　Windows应用程序开发入门

如何开始Windows应用程序开发的学习呢？本书将从一个最简单的Windows应用程序入手，讲解 Windows 应用程序设计中的基本概念、编程工具的使用安装与使用方法。通过本章的学习，读者将会对 Windows 应用程序的设计有一个初步的、直观的认识。

1.1　第一个实例程序

本节将给出一个完整的实例程序，并对该程序源代码进行分析。通过本节的学习，读者可以了解 Windows 应用程序源代码的必要组成部分以及它们的组织结构。

1.1.1　sta rt.exe

start.exe 是本书的第一个实例，创建一个简单的消息对话框。

图 1-1　第一个实例程序

实例 1-1　第一个 Windows 应用程序 start.exe

打开本书配套光盘，在第一章实例代码目录中找到可执行文件 start.exe，运行 start.exe，会弹出如图 1-1 所示的消息对话框。这就是一个最简单的 Windows 应用程序实例运行的效果，实现代码如下：

```
/* 头文件 */
#include <windows.h>
//连接时使用 User32.lib
#pragma comment (lib, "User32.lib")
/* ************************************
* WinMain
* 功能: Windows 应用程序示例
************************************/
int WinMain(
            HINSTANCE hInstance,
            HINSTANCE hPrevInstance,
            LPSTR lpCmdLine,
            int nCmdShow
            )
```

```
{
    // 调用 API 函数 MessageBox
    MessageBox(NULL,
        TEXT("开始学习 Windows 编程"),
        TEXT("消息对话框"),
        MB_OK);
    return 0;
}
```

1.1.2 Windows API

实例 1-1 中，程序最核心的功能实现使用了 Windows API 的 MessageBox 函数。

就像 C 语言有一系列库函数一样，任何一种程序开发平台都会提供众多的接口函数供开发人员使用。Windows 应用程序的开发核心问题是如何使用 Windows SDK 所提供的 API。

所谓 API 是“Application Program Interface”的简写，中文含义为“应用程序接口”，它是一系列函数、宏、数据类型、数据结构的集合。运行于 Windows 系统的应用程序可以使用这些操作系统提供接口来完成应用程序需要的功能。

MessageBox 函数是 Windows 众多 API 中的一个，其的功能是弹出一个对话框。

1.1.3 程序入口函数

WinMain 函数是程序入口点，相当于 C 语言的 main 函数，其定义如下：

```
int WinMain(
          HINSTANCE hInstance,
          HINSTANCE hPrevInstance,
          LPSTR lpCmdLine,
          int nCmdShow
               );
```

WinMain 函数有 4 个参数：hInstance、hPrevInstance、lpCmdLine、nCmdShOW。

◇ 第一个参数是 hInstance，数据类型是 HINSTANCE，此参数表示应用程序本次运行实例的句柄。

◇ 第二个参数是 hPrevInstance，数据类型也是 HINSTANCE，表示应用程序之前运行实例的句柄，但是在实际应用中，此参数始终为 NULL。

◇ 第三个参数 lpCmdLine 是运行时参数。如在 cmd 命令行中运行“start.exe Command Arguments”，那么 lpCmdLine 就指向字符串“Command Arguments”，作用与 C 语言中 main 函数的参数作用类似。数据类型 LPSTR 是 Windows API 中常用的字符串类型。

◇ 最后一个参数是 nCmdShow，数据类型是 int，表示应用程序窗口（如果有）的显示状态。

WinMain 函数的返回类型是 int 型的。在实例 1-1 中，使用了如下语句来返回。

```
return 0;
```

1.1.4 start.c代码分析

现在分析 1.1.1 小节给出的实例 1-1 程序的源代码 start.c。

在代码的最开始是源代码文件信息的注释。

之后是预编译声明，在本实例中共有两条预编译声明。

```
#include <windows.h>
```

这一行代码指明了包括 Windows.h。Windows.h 是 Windows 应用程序开发中常用的头文件，在Windows应用程序开发中所使用的很多的数据类型、结构、API接口函数都在Windows.h或Windows.h所包含的其他头文件中进行了声明，比如实例 1-1 所使用的 MessageBox 函数。如果不声明包括 Windows.h，那么编译器将不能识别 MessageBox。

```
#pragma comment (lib, "User32.lib")
```

这行代码指明将源文件编译生成目标文件（.obj）后，将目标文件链接成可执行文件的过程中需要使用到 User32.lib，因为 User32.lib 中包含了 MessageBox 的链接信息。如果

在链接程序时，调用链接器的参数中指明了链接时需要使用到 user32.lib，这一句程序代码可以省略（参见第 3 章）。

代码接下来是 WinMain 函数的定义。在 WinMain 函数中调用了 MessageBox 函数。

```
MessageBox(NULL,
    TEXT("开始学习 Windows 编程"),
    TEXT("消息对话框"),
    MB_OK);
```

MessageBox 函数的原型如下：

```
int MessageBox(
    HWND hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT uType
);
```

第一个参数 hWnd 是消息框所属的窗口的句柄，可以设置为 NULL。

第二个参数 lpText 是字符串，表示消息框所显示的消息。

第三个参数 lpCaption 是字符串，表示消息框的标题。

第四个参数 uType 是消息框的类型，在实例 1-1 中设置为“MB_OK”，意思是消息框包含一个“确定”按钮。

在 MessageBox 函数调用之后，程序返回 0，结束 WinMain 函数，退出进程。

1.2 编译代码

本书使用 Visual Studio 和 Platform SDK 作为程序编译和连接的工具。各版本的 Visual Studio，包括 Visual Studi0 6.0、Visual Studi0 2003、Visual Studi0 2005、Visual Studi0 2008 都可以用于编译本书中的实例程序。

过去已经发行了的所有 Windows 版本，每一个版本都有一个 Platform SDK。Windows 系统为应用程序提供了很多的调用接口，如果要使用这些调用接口就需要用于 Platform SDK。PlatformSDK 提供了开发 Windows 应用程序所必须的头文件、库文件等。Windows 应用程序的开发接口是 C＼C++语言形式的。读者可以使用 Platform SDK for Windows XP SP2、Platform SDK for WindowsServer 2003、Platform SDK for Windows Vistaa

Visual C++ 2005 Express Edition（Visual C++速成版）是 Visual Studi0 2005 的一个子版本。

包括了编译和连接的全部工具 VC\VC++8.0，对编译本书的示例程序已经足够使用。

微软公司提供官方下载，下载地址为 http://msdn2.microsoft.com/zh-cn/express/default.aspx。各个版本的 Platform SDK 都也可以从微软公司的网站上下载到。

在本书中，将综合使用 Visual C++ Express Edition 2005、Visual C++ Express Editon 2008 以及 MicrosoftPlatform SDK XP SP2 版本、2003SP1 版本和 Vista 版本。

1.2.1 安装Visual Studio

Visual Studio 包括速成版、专业版和团队系统版本(Visual Studio Team System)。速成版是免费的版本，专业版和团队系统版本是商业版。但是速成版的功能与专业版的功能并没有太大的区别。

如果读者使用 Visual C++速成版，需单独安装 Microsoft Platform SDK，并将其与

Visual Studio 集成在一起，才可以在本机进行 Windows 应用程序的开发。

Visual C++速成版的安装步骤如下。

(1)从微软公司的站点上下载中文版的 Visual C++速成版，也可以从本书配套光盘中获得。运行 vcsetup.exe。

(2)单击“下一步”按钮。

在这一步，读者需要选择需要安装的组件。推荐安装图形化的 IDE，IDE 将大大方便代码的编辑、编译等工作。不安装图形化 IDE，读者也可以使用命令行工具来编译程序。

(3)在安装的过程中，需要连接上 Internet。安装程序将下载 Windows Installer 3.1、NET Framework 和 Visual C++ 2005 速成版进行安装。

安装完成后，进行注册就可以使用了。在 Visual Studio 安装目录下的 bin 文件夹中，读者可以看到包括 cl.exe、nmake.exe、link.exe、lib.exe、rc.exe 等编译和链接工具。

安装了 Visual C++速成版后仅仅可以进行标准 C 程序和 C++程序的开发，但是还不能调用 Windows API，无法完成一个 Windows 应用程序所需的大部分功能，因为缺少 Platform SDK。

1.2.2 安装Microsoft Platform SDK

SDK 是 Software Development Kit 的缩写，即“软件开发工具包”。Microsoft Platform SDK 是进行 Windows 应用程序的开发包。

每一个 Windows 发行版本都会有对应的 Platform SDK。SDK 中会使用与这个版本相一致的头文件和库文件等。每一个版本 Windows 中新增加的 API、数据结构中也会在其中 体现。

下面以 Platform SDK for Windows Server 2003 R2 为例说明，其他版本的安装方法类似。

如图 1-2 所示。

图 1-2 Platform SDK 安装界面

单击“Next”按钮，选择安装选项。

在一般情况下，选择典型安装可以满足使用需要。这里需要进行一些设置，所以选择“Custom”，如图 1-3 所示。

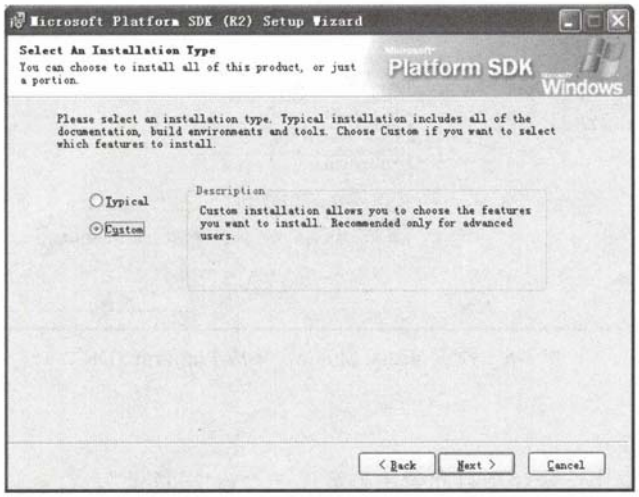

图 1-3　选择安装类型

在出现如图 1-4 所示的安装选择时，选择“Configuration Options” →“Register Environment Variables”选项，并选择完全安装。如果不选择此项，Visual Studio 工具将无法找到 SDK。

图 1-4　选择安装 Platform SDK 的功能

在安装完成后，可以从 Platform SDK 安装目录中找到若干头文件和库文件，如 Windows.h、Keme132.lib 等文件，这些就是我们进行 Windows 应用程序开发所必不可少的支持。

1.2.3　集成Microsoft Platform SDK与Visual C++速成版

在安装完成后，还需要对 Visual C++速成版进行设置，使编译链接工具可以找到 SDK。如果编译链接工具找不到 SDK，那么在源代码中的 API 调用将会产生编译链接错误。

在 Visual C++速成版中进行如下操作。在菜单中选择“工具”→“选项”，出现“选项”对话框，如图 1-5 所示。

图 1-5　设置 Visual Studio，集成 Platform SDK

1．设置 IDE 环境

在“选项”对话框中选择“项目和解决方案”→“VC++目录”。

在“显示以下内容的目录”中选择列表中选择“包含文件”，单击下方的“新建”按钮

选择 Platform SDK 安装目录下的“Include”文件夹。如：

C:\Program Files\Microsoft Platform SDK \Include.

在选择列表中选择“库文件”，单击下方的“新建”按钮。选择 Platform SDK 安装目录下的“Lib”文件夹，如：

C:\Program Files\Microsoft Platform SDK YLib.

在选择列表中选择“可执行”，单击下方的“新建”按钮。选择 Platform SDK 安装目录下的“Bin”文件夹，如：

C:\Program FilesYMicrosofi Platform SDK \Bin。

2．设置环境变量

如果在安装 Platform SDK 时没有选择安装“Configuration Options” →“Register Environment Variables”，或者选择了“Typical”安装模块，那么还需要为系统添加环境变量。

MSSdk 变量：值为 SDK 安装目录。

Mstools 变量：值为 SDK 安装目录。

INETSDK 变量：值为 SDK 安装目录。

Bkoffice 变量：值为 SDK 安装目录。

Basemake 变量：值为%MSSdk%\Include\BKOffice.Mak。

INCLUDE 变量：值为%MSSdk%\Include\。

LIB 变量：值为%MSSdk%\Lib\。

并在 Path 变量前加上%MSSdk%\Bin\;%MSSdk%\Bin\WinNT\。

Visual Studio 专业版不需要上述设置，但是也可以使用这种方法来更新 Visual Studio 所使用的 SDK 的版本。

1.2.4　Vista SDK与Visual Studi0 2008

截止本书编写时，Windows Vista 版本的 SDK 已经发布，Visual Studio 的最新版本也

已经升级到了 2008 版。Visual Studi0 2008 也区分为速成版、专业版、团队系统版等。Vista SDK 的安装、使用与方法与之前版本的 SDK 安装和使用方法类似。Visual Studi0 2008 的安装与使用方法也与VIsual Studi0 2005 的安装使月 j 方法也区别不大，速成版的 Visual Studio 仍然可以免费使用。但是新版本的 SDK 和 Visual Studio 提供了更丰富的功能，使用也更便捷。建议有条件的读者使用新版
本 SDK 和 Visual Studio。

1.2.5 Visual Studio专业版或团队系统版

如果读者有 Visual Studio 专业版或团队系统版，可直接进行安装，专业版已经内嵌了 Platform
SDK。在安装完成后，不需配置就可使用。

1.2.6 使用图形化IDE建立工程、进行编译

在完成安装和设置后，本小节将以 Visual C++速成版为例说明如何编译 1.1.1 节的程序。

◇打开 Visual C++速成版。

◇新建工程，在菜单中选择“文件一新建工程”在新建工程对话框中的“Visual C++”类别下，选择“空项目”，如图 1-6 所示。

图 1-6 Visual Studio 新建工程

◇输入项目名称，完成项目的建立。

◇在“解决方案装源管理器”中，右键单击工程名称，选择“添加一新建项”，在“添加新项”对话框中，选择“代码”，键入文件名称“start”，文件后缀是“. c”或者是“.cpp”
都可以，如图 1-7 所示。

◇将实例 1-1 代码复制到 start.c 中。

◇如果不需要调试，在工具栏（位于菜单栏下）中将“Debug”改为“Release”。

◇在菜单中选择“生成一生成解决方案”（或使用快捷键，一般是 F7)。

◇如果编译没有报错，则在程序目录下会有“Debug”或” “Release”文件夹，可执行文件位于此。

运行可执行文件，得到实例 1-1 的运行结果。

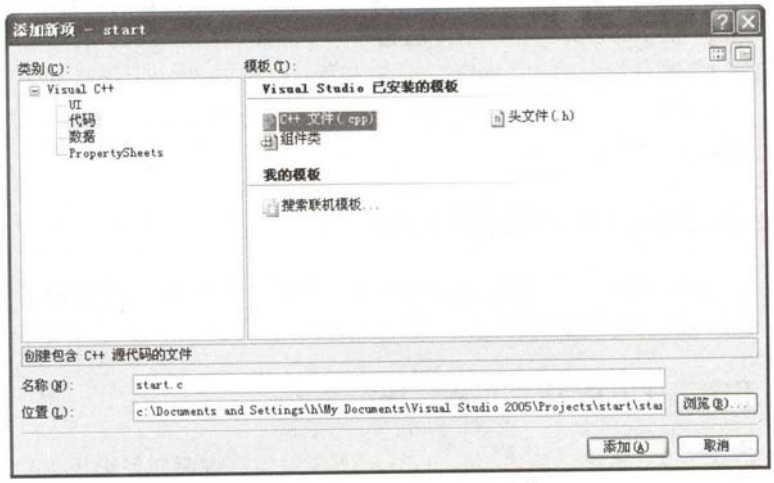

图 1-7　为工程添加代码文件

1.2.7　“解决方案”与“工程”

“解决方案”是指若干工程的集合。一般大型的应用程序，都不是一个可执行文件构成的，可能包括一个 exe 文件、若干个 dll 文件等。一个解决方案下可以有零个至多个工程，每个工程可以生成一个可执行文件。

可以为解决方案添加或删除工程。

1.2.8　使用命令行工具编译

除了使用图形化的 IDE 工具，还可以使用 nmake 工具对程序进行编译和链接。nmake 是一个字符界面程序，通过 Makefile 配置。在使用 nmake 工具时需要使用 Visual Studio 的命令行。

将实例 1-1 中的代码存为 start.c，然后将实例 1-2 的内容存为文件 makefile。

实例 1-2 编译 sta rt.c 的 makefile

本实例是一个 makefile 文件，可以用于编译 start.c。编译时需要使用到 nmake 工具，其代码如下所示：

```
# Nmake macros for building Windows 32-Bit apps
!include <Win32.Mak>
OUTDIR = START_BIN
#-------------------- Build Rule ----------------------------------
all: $(OUTDIR) $(OUTDIR)\start.exe
#----- If OUTDIR does not exist, then create directory
$(OUTDIR) :
   if not exist "$(OUTDIR)/$(NULL)" mkdir $(OUTDIR)
$(OUTDIR)\start.obj: start.c
   $(cc) $(cflags) $(cvars) /Gz /Fo"$(OUTDIR)\\" /Fd"$(OUTDIR)\\" start.c
$(OUTDIR)\start.exe: $(OUTDIR)\start.obj
   $(link) $(guiflags) -out:$(OUTDIR)\start.exe $(OUTDIR)\start.obj $(conlibs)
#-------------------- Clean Rule ----------------------------------
# Rules for cleaning out those old files
```

```
clean:
        $(CLEANUP)
```

将makefile文件和start.c文件放在同一目录下。在开始菜单“Visual Studio Tools”目录下找到“Visual Studio 2005\2008 命令提示”(Visual Studio 2005\2008 Command Prompt)并运行。切换到start.c和makefile两个文件所在的目录，运行“nmake.exe”，输出如下：

```
    if not exist "START_BIN/" mkdir START_BIN
    cl -c -DCRTAPI1=_cdecl -DCRTAPI2=_cdecl -nologo -GS -D_X86_=1  -DWIN32 -D_WIN32 -W3
-D_WINNT -D_WIN32_WINNT=0x0500 -D_WIN32_IE=0x0500 -DWINVER=0x0500  -D_MT -MTd /Gz
/Fo"START_BIN\\" /Fd"START_BIN\\" start.c
    start.c
    link       /INCREMENTAL:NO /NOLOGO -subsystem:windows,5.0 -out:START_BIN\start.exe
START_BIN\start.obj kernel32.lib  ws2_32.lib mswsock.lib advapi32.lib bufferoverflowu.lib
```

如果没有报告错误，将在目录下出现“START_BIN”文件夹，start.exe 位于此文件夹中。

至此，已经可以使用开发工具编译链接程序代码，得到可执行程序。本书第3章还会详细介绍编译链接工具的原理和使用方法。

第2章　Windows API概要

本章主要讲解Windows程序设计的一些基本概念，包括在Windows应用程序设计中所使用到的一些特有的数据类型;对Windows众多的API进行了分类介绍，还将初步介绍Windows API与动态链接的原理，Windows应用程序所使用的字符集等。

2.1　Windows数据类型

Windows API使用了很多Windows自己定义的数据类型。读者可能较为熟悉C语言或C++语言的数据类型。要熟练使用Windows API必须要熟悉Windows数据类型。这些数据类型是Windows特有的。在SDK的相关头文件中有定义。

在众多的Windows数据类型中，最常用的有DWORD、HANDLE、LPTSTR、WORD、BYTE、CHAR等。在Windows系统中，DWORD用于表示无符号整型的数据，意为double word，32位。

在一般情况下BYTE是8位的，而WORD是16位，DWORD就是32位的。

Windows系统的应用程序中还具有一个特有的数据类型-HANDLE，通常HANDLE类型的变量用于唯一标识一个“对象”，如窗口、控件、文件等，Windows平台中存在众多这样的对象，对象是程序操作的目标。HANDLE也是一个32位的数据类型。

2.1.1 Windows数据类型示例

实例 2-1 使用了几种基本的 Windows 数据类型，演示 Windows 数据类型在程序中的使用方法。

1．实例 2-1　常用 Windows 数据类型演示(DWORD/LPSTR/CHAR/INT)

本实例定义了几个常用常量的 Windows 数据类型，包括 DWORD，LPSTR 和 CHAR，并演示了如何使用它们进行了复制、比较大小等操作。

```
/* 头文件 */
#include <windows.h>
#include <stdio.h>

/* ************************************
* 功能    Windows 数据类型演示
************************************/
int WinMain(
            HINSTANCE hInstance,
            HINSTANCE hPrevInstance,
            LPSTR lpCmdLine,
            int nCmdShow
            )
{
     //定义字符串
     LPSTR szString = "Windows data type, string.";
    //定义字符数组
    CHAR lpString[120];//要大于 szString 的长度
    //定义 DWORD 类型的数据
    DWORD dwMax = 0xFFFFFFFF;
    DWORD dwOne = 0x1;
   //定义 INT 类型的数据
   INT iMax = 0xFFFFFFFF;
   INT iOne = 0x1;

   //显示字符串
   MessageBox(NULL,szString,"LPSTR",MB_OK);
   //复制内存，将字符串复制到数组中（包括 NULL 结束符）
   CopyMemory(lpString, szString,lstrlen(szString)+1);
   //显示复制的字符串
   MessageBox(NULL,lpString,"CHAR[]",MB_OK);

  //比较 DWORD 并显示结果
  if(dwMax>dwOne)
  {
      MessageBox(NULL,"DWORD 类型的数据 0xFFFFFFFF > 0x1","DWORD",MB_OK);
  }
  //比较 INT 并显示结果
  if(iMax<iOne)
  {
      MessageBox(NULL,"INT 类型的数据 0xFFFFFFFF < 0x1","INT",MB_OK);
  }
  return 0;
}
```

在这个程序中，使用了 4 种 Windows 数据类型，分别是 LPSTR、CHAR、DWORD 和 INT。

LPSTR 类型的数据是字符串，也就是字符指针，CHAR 是字符，DWORD 是 32 位的无符号整数，INT 是 32 位有符号整数。

程序运行后会弹出 4 个对话框。这说明 dwMax>dwOne 是成立的。iMax<iOne 也是成立的。dwMax 与 iMax 的数值是一样的，dwOne 与 iOne 的数值也是一样的。但是比较结果不同，是因为二者的数据类型不一样。

2．为什么要使用 Windows 数据类型

Windows API 函数、数据结构等都是采用 Windows 数据类型。比如 Windows API 函数的

入口参数、返回值等都是 Windows 数据类型。因此学习 Windows API 程序设计必须要了解 Windows 数据类型。Windows 数据类型与 Windows API 的关系详见 2.1.3 小节。

3．查看 Windows 数据类型的定义

在 Visual Studio 中可以查看数据类型的定义。在数据类型的类型名（比如“DWORD”）上单击右键，选择“转到定义”，如图 2-1 所示。

图 2-1　查看变量或数据类型的定义

可以从 SDK 的头文件中看到各类型的类型定义。

“INT”的定义如下：

```
typedef int INT;
```

“DWORD”的定义如下：

```
typedef unsigned long DWORD;
```

"CHAR"的定义如下：

```
typedef char CHAR;
```

从中可以发现，这些 Windows 数据类型都是从标准 C 的数据类型经过类型重定义而来。INT 数据类型是有符号整型，DWORD 数据类型是无符号整型。

这就说明了为什么在实例 2-1 中 INT 数据类型的变量 iMax 实际是“-1”，而 DWORD 类型的变量 dwMax 实际是“4294967295”，所以会出现如上的比较结果。

4．最常见的 Windows 数据类型

Windows API 中还会使用到其他很多的 Windows 特有的数据类型。常用的类型说明如表 2.1 所示。

表 2.1　　常用的基本 Windows 数据类型

类　型	描　述
BOOL	布尔型变量（值只能是 True 或 False）
BYTE	字节类型（8 位）
CHAR	8 比特字节（ANSI）
CONST	常量，相当于标准 C 中的“const”关键字
DWORD	32 字节无符号整型数据
DWORD32	32 字节无符号整型数据
DWORD64	64 字节无符号整型数据
FLOAT	浮点数据类型
HANDLE	对象的句柄，最基本的句柄类型
HICON	图标的句柄
HINSTANCE	程序实例的句柄
HKEY	注册表键的句柄
HMODULE	模块的句柄
HWND	窗口的句柄

续表

类　型	描　述
INT	32 位符号整型数据类型
INT_PTR	指向 INT 类型数据的指针类型
INT32	32 位符号整型
INT64	64 位符号整型
LONG	32 位符号整型，相当于 C 语言的标准数据类型 long
LONGLONG	64 位符号整型
LONG32	32 位符号整型
LONG64	64 位符号整型
LPARAM	消息的 L 参数
WPARAM	消息的 W 参数
LPCSTR	Windows（ANSI）字符串常量
LPCTSTR	根据环境配置，如果定义了 UNICODE 宏，则是 LPCWSTR 类型，否则是 LPCSTR 类型
LPCWSTR	UNICODE 字符串常量
LPDWORD	指向 DWORD 类型数据的指针
LPSTR	Windows (ANSI)字符串变量
LPTSTR	根据环境配置，如果定义了 UNICODE、则是 LPWSTR 类型，否则是 LPSTR 类型
LPWSTR	UNICODE 字符串变量
SHORT	无符号短整型（16 位）
SIZE_T	表示内存大小，以字节为单位，其最大值是 CPU 最大寻址范围
TCHAR	如果定义了 UNICOD，则为 WCHAR，否则为 CHAR
UCHAR	无符号 CHAR
UINT	无符号 INT
ULONG	无符号 LONG
VOID	无类型，相当于标准 C 语言的 void
WCHAR	16 位 Unicode 字符
WINAPI	Windows API 的函数调用方式，常见于 SDK 头文件中对 API 函数的声明中，相当于__stdcall（更严格地讲，这不是数据类型，而是一种函数调用约定，参见 18.4.2 小节）
WORD	16 位无符号整型数据

Windows 数据类型不止上表中列出类型，还包括其他很多种，主要有各种数据的指针，句柄、无符号类型等，读者可以查看数据类型的定义来了解 Windows 数据类型，本小节第 5 点中给出了 Windows 数据类型命名的规律，在遇到新的数据类型时，可以基本推论出数据类型的意义。读者也可以通过查看 SDK 文档来了解 Windows 数据类型的更多细节。

5．Windows 数据类型名命名的规律

Windows 数据类型的命名都很有规律。

基本数据类型包括 BYTE、CHAR、WORD、SHORT、DOWRD、INT 等。

指针类型的命令方式一般是在其指向的数据类型前加“LP”或“P”，比如指向 DWORD 的指针类型为“LPDWORD”和“PDWORD”。

各种句柄类型的命令方式一般都是在对象名前加“H”。Windows 系统中有很多对象，所有表示一个对象的数据类型都是句柄，每一种对象都对应着一种句柄类型，比如与位图(BITMAP)对应的句柄类型为“HBITMAP”，与菜单(MENU)对应的句柄类型为“HMENU”，与

窗口(WINDOW)对应的句柄类型为“HWND”。

无符号类型一般是以“U”开头，比如“INT”是符号类型，“UINT”是无符号类型，“LONG”是符号类型“ULONG”是无符号类型等。

2.1.2 Windows数据类型与标准C数据类型的关系

查看 Windows 数据类型的定义可以看到，所有的 Windows 数据类型都是由 C 数据类型经过类型重定义得到的。如 DWORD 实质上就是 unsigned long 数据类型，32 位的无符号整型。

WinDefHk 的定义如下：

```
typedef unsigned long DWORD;
```

而将在 Windows 程序设计中经常用到的 HANDLE 类型实质上是无类型指针 void。在 WinNT.h 中，有一系统的数据类型定义，与 HANDLE 有关的定义如下：

```
typedef PVOID HANDLE;
typedef void *PVOID;
```

所有的 Windows 数据类型都是通过这种方式在 SDK 的头文件中进行定义的，它们都是来源于标准 C 的数据类型。

而实际上 VC 编译器是一个完整的 C 编译器，此外并没有过多的扩展。Windows 数据类型也不是 VC 的内建类型，而从标准 C 类型重定义得到。

2.1.3 Windows数据类型与Windows API

Windows API 函数的参数、返回值或一些重要的常量使用的数据类型都是 Windows 数据类型。

可以从 Windows API 的定义中发现这一点。以几个 Windows API 的定义为例，例如本书第 1 章实例 1-1 中所使用的 MessageBox 函数，如下所示：

```
int MessageBox(
    HWND hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT uType
);
```

MessageBox 函数的返回值是 int 型的，是标准 C 数据类型，但是所有的参数都使用了 Windows 数据类型。如 HWND 是一种 Windows 数据类型，用于表示窗口的句柄；LPCTSTR 也是 Windows 数据类型，表示字符串常量；UINT 也是 Windows 数据类型，为无符号整型。

再如在本书的后续章节常会使用到的 ReadFile 函数，如下所示：

```
BOOL ReadFile(
  HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

使用了 BOOL、HANDLE、LPVOID、DOWRD、LPDWORD 等 Windows 数据类型。最后一个参数则是指向 OVERILAPPED 结构的指针。

BOOL 是布尔型，值可以是 TRUE 和 FALSE，HANDLE 是文件句柄类型，LPVOID 是无类型指针，相当于 void *，DWORD 是无符号整型，LPDWORD 是指向 DOWRD 类型数据的指针类型。

Windows SDK 中也定义了很多类似于 OVERI。APPED 的结构体类型，这些类型是 Windows

API 所不可缺少的部分。很多 API 函数都是使用这样的结构体作为参数或返回结果。

2.1.4 Windows中的数据结构

Windows 中包含很多种数据结构类型，在不同类型的 API 中会使用到不同的数据结构，由于数据结构的数量众多，将在后续章节具体的实例中介绍实现每一种功能所使用的数据结构。数据结构通常会作为一些 API 的参数输入。

Windows API 中的数据结构定义具有一定的特点，如将在第 4 章中使用到的 WIN32_FILE_ ATTRIBUTE_ DATA 数据结构的定义如下：

```
typedef struct _WIN32_FILE_ATTRIBUTE_DATA {
  DWORD dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD nFileSizeHigh;
  DWORD nFileSizeLow;
} WIN32_FILE_ATTRIBUTE_DATA,
*LPWIN32_FILE_ATTRIBUTE_DATA;
```

相当于是：

```
struct _WIN32_FILE_ATTRIBUTE_DATA {
   DWORD dwFileAttributes;
   FILETIME ftCreationTime;
   FILETIME ftLastAccessTime;
   FILETIME ftLastWriteTime;
   DWORD nFileSizeHigh;
  DWORD nFileSizeLow;
};
typedef _WIN32_FILE_ATTRIBUTE_DATA WIN32_FILE_ATTRIBUTE_DATA;
typedef _WIN32_FILE_ATTRIBUTE_DATA *LPWIN32_FILE_ATTRIBUTE_DATA;
```

Windows 系统中类似于这样的数据结构类型定义很多，也有很多的 API 函数使用到了这些数据结构类型。读者应该对这样的数据结构定义有所熟悉，这将在 Windows 应用程序设计中经常使用到。

Windows SDK 中，结构体也有自己的命名规范。一般情况下，Windows 系统中使用全大写来命名结构体、共用体，并使用“_”来分隔单词，在结构名加“LP”或“P”表示指向数据结构的指针。

2.2 Windows API的功能分类

Windows API 所能实现的功能包括很多方面，在进行应用程序的开发时，开发人员可能会使用到文件、进程、内存、权限、系统信息等系统的基本服务和系统管理类的 API，可能会用到图形用户界面、控件等函数和对象，可能需要在界面上绘制图像处理多媒体信息等，还包括进行网络通信开发等。

2.2.1 系统基本服务

系统基本服务是 Windows API 最基本的内容，是最常使用到的程序接口。系统基本服务 API 包括以下几个方面。

1．文件系统

对文件的基本操作包括文件的创建、打开、读写、关闭、删除，文件属性的设置与获取，目录操作，以及磁盘分卷的操作，还包括镜像文件、加密文件系统等。

2．内存管理

主要是内在的分配、共享、释放等内容，包括虚拟内存管理、分页机制、堆管理等。

3．进程、线程和模块

包括进程主程序(exe)、模块、动态链接库(dll)的编写；线程的基本概念，线程创建、遍历、同步等操作；进程与权限；线程与纤程等内容。

4．设备 I/O、驱动程序控制

包括设备对象等基本概念。加载与卸载驱动程序，控制驱动程序，与驱动程序通信等。

5．调试与错误处理

包括如何开发调试器，程序运行错误的处理，日志的记录、Windows 可执行文件的结构等。

6．Windows 系统信息

包括注册表的操作，如打开注册表，读取、写入键值，创建、删除键；还包括系统基本信息的获取和设置，如系统目录、系统版本、计算机名等。

7．进程间通信

包括使用共享文件进行进程间通信的方法，使用消息进行进程间通信的方法，使用邮槽、管道等进行进程间通信的方法，使用网络共享进行进程间通信的方法。

8．定时器与消息机制

消息机制是 Windows 系统中很重要的一种机制。几乎所有的 Windows 应用程序都在与消息打交道，而 Windows 的消息机制又是依赖于定时器的。所以了解 Windows 消息机制是学习 Windows 应用程序开发的重要内容。

9．其他

Windows 的系统基本服务还包括性能监视、电源管理、索引与数据存储等，也将在本书中有所涉及。

10.实例 2-2 将系统目录信息写入 systemroot.txt

本实例使用了 Windows 系统基本服务中的部分 API，实现将系统目录路径写入文件中。在这里不对实现原理做过多的解释，只是使读者对 Windows 系统服务 API 有初步的了解。后续章还将详细介绍这些程序接口的使用方法。

```
/* 头文件 */
#include <windows.h>

/* ************************************
```

```
* 功能    获取系统目录信息，并存储到文件中
**************************************/
int main(int argc, TCHAR argv[])
{
    //文件句柄
    HANDLE hFile;
    DWORD dwWritten;
    //字符数组，用于存储系统目录
    TCHAR szSystemDir[MAX_PATH];
    //获取系统目录
    GetSystemDirectory(szSystemDir,MAX_PATH);

    //创建文件 systemroot.txt
    hFile = CreateFile("systemroot.txt",
        GENERIC_WRITE,
        0,NULL,CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL,
        NULL);
    //判断文件是否创建成功
    if(hFile != INVALID_HANDLE_VALUE)
    {
        //将系统目录系统信息写入文件
        if(!WriteFile(hFile,szSystemDir,lstrlen(szSystemDir),&dwWritten,NULL))
        {
            return GetLastError();
        }
    }
    //关闭文件，返回
    CloseHandle(hFile);
    return 0;
}
```

运行程序，在程序所在的目录会生成文件“systemroot.txt”，文件内容为系统目录。

2.2.2 系统管理

系统管理是 Windows 系统中很重要的内容。Windows 系统提供了以下方面的 API 供应用程序开发人员使用，包括：管理控制台接口、程序安装、系统恢复、任务调度、Windows 运程管理。

2.2.3 用户界面

用户界面开发是 Windows 应用程序开发的重要内容。Windows 用户界面主要包括两个部分，Windows Shell 和 Windows 图形用户界面。Windows 图形用户界面包括窗口对象的相关内容，界面资源，控件、用户数据交换（粘贴板等）和用户输入的处理。

1．窗口化

包括一些基本数据概念和对象：窗口类、消息对列、窗口过程（消息处理函数）、窗口属性等。

主要包括的 API 有窗口类注册的函数、窗口建立与关闭函数、窗口属性修改函数等。

2．资源

资源是 Windows 应用程序中很重要的一项内容，在可执行文件中，专门有用于存储资源的节。

这里所说的资源是界面资源，包括图标、菜单、字符串、版本信息、对话框、动态光标、插入号(Carets)等。资源的处理涉及众多 API 函数。

3．用户输入处理

包括对话框、键盘加速器（快捷键）、键盘输入、鼠标输入、原始数据输入等。这些概念是程序通过用户界面与用户进行交互的基本概念。

4．数据交换

包括 Atom 表（包括了若干字符串和标识符）、粘贴板、数据复制、动态数据交接等基本概念，是用户和应用程序通过用户界面进行数据交换的主要载体。

5．Windows 图形界面通用控件

Windows 系统提供了不少于 30 种的控件供应用程序使用，其中最常用的几种包括 Edit 控件（文本框）、Button 控件（按钮）、ListView 控件（列表）、ToolTip 控件（提示框）、ComboBox 控件（下拉选择框）、Tree-View 控件（树）、Rich-Edit 控件（多功能文本框）、Tab 控件（分页）、Process 控件（进度条）等。

6．Windows Shell

Windows 桌面浏览器(explorer.exe)的功能强大，Windows Shell 应用程序可以利用 WindowsShell 桌面的程序接口达到管理系统、定制图标和默认程序、扩展桌面功能等目的。

有关界面及控件的使用方法与原则将在第 9 章详细介绍。

Windows Shell 程序设计相对独立，将在第 12 详细介绍。

7．实例 2-3 窗口创建以及消息处理 basic.c

本实例演示 Windows 用户界面 API 的基本功能，使读者对 Windows 应用界面开发有最基本的认识，本实例详细的实现原理还会在第 9 章中详细介绍。

```
/* 头文件 */
#include <windows.h>

/* 全局变量 */
HINSTANCE hinst;

/* 函数声明 */
int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int);
LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

/* ************************************
* 功能    显示一个窗口
************************************/
int WINAPI WinMain(HINSTANCE hinstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow)
{
   WNDCLASSEX wcx;                          // 窗口类
   HWND hwnd;                               // 窗口句柄
   MSG msg;                                 // 消息
   BOOL fGotMessage;                        // 是否成功获取消息
   hinst = hinstance;                       // 应用程序实例句柄，保存为全局变量
```

```
    // 填充窗口类的数据结构
    wcx.cbSize = sizeof(wcx);                    // 结构体的大小
    wcx.style = CS_HREDRAW |
       CS_VREDRAW;                               // 样式：大小改变时重绘界面
    wcx.lpfnWndProc = MainWndProc;               // 窗口消息处理函数
    wcx.cbClsExtra = 0;                          // 不使用类内存
    wcx.cbWndExtra = 0;                          // 不使用窗口内存
    wcx.hInstance = hinstance;                   // 所属的应用程序实例句柄
    wcx.hIcon = LoadIcon(NULL,
        IDI_APPLICATION);                        // 图标：默认
    wcx.hCursor = LoadCursor(NULL,
        IDC_ARROW);                          // 光标：默认
    wcx.hbrBackground = (HBRUSH)GetStockObject(
       WHITE_BRUSH);                             // 背景：白色
    wcx.lpszMenuName =  NULL;                    // 菜单：不使用
    wcx.lpszClassName = "MainWClass";            // 窗口类名
    wcx.hIconSm = (HICON)LoadImage(hinstance,  // 小图标
        MAKEINTRESOURCE(5),
        IMAGE_ICON,
        GetSystemMetrics(SM_CXSMICON),
        GetSystemMetrics(SM_CYSMICON),
        LR_DEFAULTCOLOR);

    // 注册窗口类
    if(!RegisterClassEx(&wcx))
    {
        return 1;
    }

     // 创建窗口
     hwnd = CreateWindow(
        "MainWClass",       // 窗口名
        "CH 2-3",           // 窗口标题
       WS_OVERLAPPEDWINDOW, // 窗口样式
       CW_USEDEFAULT,       // 水平位置X：默认
       CW_USEDEFAULT,       // 垂直位置Y：默认
       CW_USEDEFAULT,       // 宽度：默认
       CW_USEDEFAULT,       // 高度：默认
       (HWND) NULL,         // 父窗口：无
       (HMENU) NULL,        // 菜单：使用窗口类的菜单
       hinstance,           // 应用程序实例句柄
       (LPVOID) NULL);      // 窗口创建时数据：无

      if (!hwnd)
      {
           return 1;
       }
       // 显示窗口
       ShowWindow(hwnd, nCmdShow);
       UpdateWindow(hwnd);
          // 消息循环
     while (
      (fGotMessage = GetMessage(&msg, (HWND) NULL, 0, 0)) != 0
      && fGotMessage != -1)
    {
       TranslateMessage(&msg);
       DispatchMessage(&msg);
    }
    return msg.wParam;

}
```

```
/* ************************************
* MainWndProc
* 功能     窗口消息处理函数,
       对所有的消息都使用默认处理函数
************************************/
LRESULT CALLBACK MainWndProc(HWND hwnd,
                        UINT uMsg,
                        WPARAM wParam,
                        LPARAM lParam
                        )
{
    switch (uMsg)
    {
        case WM_DESTROY:
            ExitThread(0);
            return 0;
        default:
            return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
}
```

本实例使用了与图形用户界面相关的一些函数，目的在于使读者对用户界面接口 API 有一个初步的认识。实例中，首先注册了窗口类，然后创建了一个窗口，创建窗口时指定的窗口的属性和窗口消息的处理函数。函数消息的处理函数大多调用系统默认函数来处理。

2.2.4 图像和多媒体

Windows 的图像引擎和多媒体编程有多种接口，包括最基本的 GDI、GDI+，以及更高级的 OpenGL、DirectX 等图像引擎编程接口。DirectX 和 OpenGL 在游戏开发、动画制作等方面应用比较广泛，使用它们进行应用程序开发需要专门的 SDK。本书只讲解 GDI，对一般的应用程序开发，已经足够使用。

GDI 的基本概念包括图像(Image)、位图 (Bitmaps)、笔(Pen)、画刷(Brush)、文本和字体(Text and Fonts)、线(Line)、区域(Regions)、形状(Shapes)等。其基本的功能是对显示设备进行控制、获取和修改相关配置，在计算机屏幕上显示用户所需要显示的内容。

GDI 涉及众多的结构、API 函数等，将在第 13 章节详细介绍。

2.2.5 网络

用户利用 Windows API 可以开发基于各种网络协议的应用程序，例如 TCP\UDP Socket、HTTP、DHCP、RPC、QOS、蓝牙，以及传真、点对点文件传输、即插即用设备管理等。

还可以进行网络管理，包括网络的基本信息，使用 IP helper 获取网络配置和网络信息、进行网络监视等。

同时微软还提供了进行网络安全编程的部分接口，主要是防火墙 Windows、防火墙 API，以及一些 Windows 自带的网络应用程序的接口，包括 IE、Outlook 等。

本书第 14 章将详细介绍网络程序设计的相关内容。

2.2.6 系统安全

包括系统的授权与认证、登录、权限管理、加密、安全管理策略等内容。本书将主要介绍用户管理（包括用户增加、修改设置、删除等）和 Windows 安全对象（任何具有访问控制

机制的对象，如文件、进程）和安全操作主体（线程）之间的安全认证机制及权限设置对程序行为的影响。第 17 章进行了相关的介绍。

2.2.7 其他功能

Windows API 能实现的功能还有很多，限于篇幅不能全部介绍。有一些内容本书未涉及，但是 Windows API 也提供了相关接口。

1．数据存储和访问、数据库

包括微软的“动态数据存取组件”技术(MDAC，包含了 ADO、ODBC)、OLE 数据库、XML 标准、微软 XML、可扩展存在引擎等多方面内容。主要是 Windows 系统内的数据库文件、Windows. 系统对数据库的访问接口等技术。

2．消息与协作

消息与协作大多是 Windows 系统自带的一些应用程序所提供的开发接口。其中最主要的内容是“协作数据对象”(CDO)。CDO 包括了若干种 Windows 数据传输典型应用，包括 Mssager (MSN)、邮件 SMTP (Outlook)应用接口等。

3．Web 开发

IIS（Intemet Information Server，互联网信息服务）等应用程序提供的开发接口，使用户可以直接在 Windows 平台上进行 Web 开发，开发的程序运行于 ns 框架下。

“数据存在与访问”、“消息与协作”、“Web 开发”等都可以归为 COM 开发的范围内。COM 是 Windows 应用程序接口的一种，具有固定的接口模式，大多是 Windows 系统中自带的应用程序所提供的开发接口的集合。

2.3 Windows API核心DLL

在 Windows 的系统目录中，存在着很多的动态链接库文件(DLL 文件)。这些 DLL 文件中包括了 Windows API 函数可执行程序。DLL 将各函数“导出”，这样应用程序就可以找到 DLL 中的函数地址，当应用程序调用 Windows API 时，程序会运行到 DLL 中。API 函数主要存在于几个核心的动态连接库文件中。Keme132.dll 是最重要的 DLL，Windows 系统最主要的系统服务 API 函数都存在于 Kerne132.dll 中。User32.dll 主要包括图形用户界面中所使用到的一些函数接口。

GDI32.dll 中，主要包括 Windows 图形引擎中的接口函数。

当用户调用一个 API 时，系统会通过程序文件中的导入表结构找到需要调用的 API 函数位于哪个 DLL，确定函数的地址，以便应用程序可以成功调用 API 函数。

2.3.1 Kerne132.dll

Keme132.dll 包括了系统基本服务中最基本的 API 函数，如文件系统、进程与线程、内存管理等。

Windows XPSP2 系统中，Keme132. dll 有 949 个导出函数，例如，CreateFileA、CreateProcessA、OpenThread、SetFileTime 等。本书将在后续章节中通过实例介绍这些 API 的使用。

2.3.2 User32.dll

User32.dll 是 Windows 图形用户界面的主要支持。一些重要的图形用户界面函数由 User32.dll 函数导出。Windows XP SP2 系统中，User32.dll 有 732 个导出函数，例如 CreateWindowExW、RegisterClassA 等。

2.3.3 Gdi32.dll

Gd132.dll 是 Windows GDI 应用程序设计接口，Gdi32.dll 导出了与此相关的若干函数，如 GetTextColor、LineTo、TextOutA 等。

2.3.4 标准C函数

标准 C 程序是一种标准，任何支持 C 语言应用程序开发的系统都应该提供 C 语言库函数的调用。在系统环境下使用标准 C 进行程序开发时，用户所使用的库函数实际上是由操作系统提供的。正是由于各个主流操作系统都提供了一套标准 C 库所定义的函数接口，标准 C 函数库才会有如此广泛的跨越操作系统平台。所以 C 程序仍然依赖于操作系统开发人员为其实现接口，而 C 库函数的实现仍然依赖于操作系统提供的调用接口，如标准 C 函数 fopen 函数在 Windows 系统中的实现就依赖于 API CreateFile 函数（CreateFile 实现了文件的创建和打开等操作）。

Windows 系统的 C 标准库函数接口主要存在于 crtdll.dll 中。实际上，C 标准函数库必须由操作系统为其提供接口，否则使用标准 C 开发的程序无法在特定的系统上运行。

Windows XP SP2 系统中，crtdll.dll 有 526 个导出函数，如 fopen、printf. strlen 等。

2.3.5 其他Dll

当然，Windows 系统中的 DLL 文件远远不止这几个，Windows 系统提供了非常丰富而且功能强大的 API，上文已经介绍了 Windows API 所主要依赖的几个 DLL，其他的 DLL 库文件由于过于庞杂，就不一一介绍，将在后续的章节中有所涉及。读者只需要了解调用 Windows API 的基本原理就可以了。

2.4 Unicode和多字节

Windows 既可以使用 Unicode 字符集又可以使用传统的字符集（如多字节编码）来实现对多种语言的支持，以适应国际市场的要求。与传统的字符集编码相比，Unicode 是世界通用的字符编码标准，使用 16 位数据表示一个字符，一共可以表示 65535 种字符，可以包括现代计算机中所使用的所有字符，包括各种字母、文字、在出版业中使用的特殊符号等。

传统的字符集，如 Windows ASNI 字符集，使用 8 位数据或将相邻的两个 8 位的数据组合在一起表示特殊的语言字符。Windows 系统采用了 ASNI 字符的扩展方式，如果一个字节是负数，则将其后续的一个字节组合在一起表示一个字符。这种编码方式的字符集也称作“多

字节”字符集。

在 Windows 系统中，Unicode 字符编码和多字节字符编码都可以使用。

1．实例 2-4　Unicode 与多字节编码演示

本实例演示不同宽度的字符串的初始化，并将字符串输出进行比较。

```
/* 头文件  */
#include <windows.h>

/* ************************************
* 功能      Un    icode 与多字节编码演示
************************************/
int WINAPI WinMain(
              HINSTANCE hInstance,
              HINSTANCE hPrevInstance,
              LPSTR lpCmdLine,
              int nCmdShow
              )
{
    //定义 LPWSTR 类型的宽字符串
     LPWSTR szUnicode = L"This is a Unicode String;";
    //定义 LPSTR 类型的窄字符串
    LPSTR szMutliByte = "This is not a Unicode String;";
    //定义 LPTSTR 类型的自适用字符串
    LPTSTR szString = TEXT("This string is Unicode or not depends on the option.");

    //使用 W 版本的 API 函数，以宽字符串为参数
    MessageBoxW(NULL,szUnicode,L"<字符编码 1>",MB_OK);
    //使用 A 版本的 API 函数，以窄字符串为参数
    MessageBoxA(NULL,szMutliByte,"<字符编码 2>",MB_OK);
    //根据编译条件自动选择 A 版本或 W 版本的 API 函数，采用相适应的字符串类型为参数
    MessageBox(NULL,szString,TEXT("<字符编码 3>"),MB_OK);

    return 0;
}
```

在本实例中，一共使用了 3 种类型的字符串变量，每种类型的字符串变量的初始化是不同的，如表 2.2 所示。

表 2.2　　字符串类型与初始化

类　　型	变 量 类 型	初始化方式
Unicode	LPWSTR	L“string”
多字节	LPSTR	“string”
根据开发环境的设置自动适应	LPTSTR	TEXT(“string”)

本实例首先使用“多字节”方式进行编译。可以使用二进制编辑器来查看编译得到的可执行文件，上例中定义的字符串分别存储为以下形式。使用二进制编辑器查看，可以从 exe 可执行文件中找到如下信息。

```
54 00 68 00 69 00 73 00 20 00 69 00 73 00 20 00
61 00 20 00 55 00 6E 00 69 00 63 00 6F 00 64 00
65 00 20 00 53 00 74 00 72 00 69 00 6E 00 67 00
3B 00 00 00 54 68 69 73 20 69 73 20 6E 6F 74 20
61 20 55 6E 69 63 6F 64 65 20 53 74 72 69 6E 67
3B 00 00 00 54 68 69 73 20 73 74 72 69 6E 67 20
69 73 20 55 6E 69 63 6F 64 65 20 6F 72 20 6E 6F
74 20 64 65 70 65 6E 64 73 20 6F 6E 20 74 68 65
20 6F 70 74 69 6F 6E 2E 00 00 00 00 3C 00 57 5B
26 7B 16 7F 01 78 31 00 3E 00 00 00 3C D7 D6 B7
FB B1 E0 C2 EB 32 3E 00 3C D7 D6 B7 FB B1 E0 C2
EB 33 3E 00
```

读者可以对照 ASCII 编码表查看

```
54 00 68 00 69 00 73 00 20 00 69 00 73 00 20 00 61 00 20 00 55 00 6E 00 69 00 63 00 6F 00
64 00 65 00 20 00 53 00 74 00 72 00 69 00 6E 00 67 00 3B 00
```

正好是“This is a Unicode String;”使用 Unicode 方式的十六进制编码。

```
54 68 69 73 20 69 73 20 6E 6F 74 20 61 20 55 6E 69 63 6F 64 65 20 53 74 72 69 6E 67 3B
```

是“This is not a Unicode String;”使用 ASCII 方式的十六进制编码。

```
54 68 69 73 20 73 74 72 69 6E 67 20 69 73 20 55 6E 69 63 6F 64 65 20 6F 72 20 6E 6F 74 20
64 65 70 65 6E 64 73 20 6F 6E 20 74 68 6520 6F 70 74 69 6F 6E 2E"
```

是字符串“This string is Unicode or not depends on the option.”的 ASCII 方式的十六进制编码，说明编译配置没有使用 Unicode。之后是“<字符编码 1>”和 Unicode 编码、“<字符编码 2>”和“<字符编码 3>”的多字节编码。对汉字字符，两种编码的方式是不同的，如“字符编码”四个字，
两种编码分别为：

```
57 5B 26 7B 16 7F 01 78
```

和

```
D7 D6 B7 FB B1 E0 C2
```

读者可以分析设置使用 Unicode 字符集后编译完成的可执行文件与设置为多字节字符集后编译得到的可执行文件有什么不同。

2．在工程配置中选择编码方式

设置的方法是在Visual Studio 工程属性中选择“配置属性”→“常规”→“字符集”选项中进行选择。

2.4.1 W版本和A版本的API

Windows 支持 Unicode 和 ASCII 编码的字符。Windows 系统 API 凡是以字符串作为参数的很多具有 W 和 A 两个版本以实现两种不同编码的字符处理。

下面以 MessageBox 为例介绍两种不同版本 API 函数的使用。在 User32.dll 中导出的函数实际上没有 MessageBox，只有 MessageBoxA 和 MessageBoxW，这两者是同一个 API，实现了同样的功能。不同的是，MessageBoxA 以多字节字符串作为参数输入，MessateBoxW 以 Unicode 字符串作为参数输入。

可以从 User32.dll 的导出函数看到两个不同版本的 API 函数。在 Platform SDK 中，MessageBox 函数声明所在的头文件中发现如下定义(以下代码来自于 Microsoft Platform SDK)：

```
WINUSERAPI
int
WINAPI
MessageBoxA(
    __in_opt HWND hWnd,
    __in_opt LPCSTR lpText,
    __in_opt LPCSTR lpCaption,
    __in UINT uType);
WINUSERAPI
int
WINAPI
MessageBoxW(
    __in_opt HWND hWnd,
    __in_opt LPCWSTR lpText,
    __in_opt LPCWSTR lpCaption,
    __in UINT uType);
#ifdef UNICODE
#define MessageBox  MessageBoxW
#else
#define MessageBox  MessageBoxA
#endif // !UNICODE
```

可以看到,在程序进行编译和连接时,如果程序在UNICODE环境下,会使用MessageBoxW,否则使用MessageBoxA。

软件开发人员可以自行设定使用Unicode编码或都多字节编码文件,不影响程序的正常功能。

但如果在编写程序时,使用的字符集与代码中使用的函数不一致、定义的字符串变量不兼容,将会引起编译错误或者运行程序显示乱码,甚至可能引起程序运行错误,这一点需要引起注意。

2.4.2 Unicode与ASGII的转换

Windows专门提供了若干个API来实现对字符编码的转换工作。WideCharToMultiByte、MultiByteToWideChar、UnicodeToBytes函数可以完成这些工作。

WideCharToMultiByte函数将Unicode字符串转换为多字节字符串，以适应A版本的API，MultiByteToWideChar函数将多字节字符串转换为了Unicode字符串，以适应W版本的API的参数形式要求。

2.5 对Windows程序设计规范的建议

每个程序员都有自己的规范化编程习惯。代码的规范不是本书的重点，所以这里只给出一些微软经常使用代码规范，MSDN的示例中，SDK的头文件和例子中几乎都是使用的这种方法。

变量名：通常采用所谓的“匈牙利命名法”，变量名由“类型缩写（小写）”+“变量描述（单词首字母大写)”构成，如字符串类型的变量可以命名为szFileName，DWORD类型的数据可以命名为dwFileSize，指针类型可以命名为lpBuffer，句柄类型的变量可以命名为hLogFile等。

函数名：各单词的首字母大写，如EnumerateFilesInDrectory、ShowFileSize等。

类型名：全大写，各单词以下划线分隔，如WIN32_FILE_ ATTRIBUTE_DATA、DWORD、HANDLE等。

常量：同类型名的命名方式。

宏：多与类型名命名方式相同，有的也与函数名命名方式相同。

大括号与代码段：一般大括号独立占一行，大括号内的代码段缩进。如果代码段仅一行（如 if-else 后的语句），也建议使用大括号。

在调用或定义函数时，如果函数太长，可以分行写，将每个参数写一行。一般多于 3 个参数的函数需要分行写，以美观和方便阅读为原则。

在本书的示例中，都将尽量使用这些规范化的方式。

第 3 章　开发工具配置与使用

程序的开发依赖于开发工具。Windows API 函数是 C 语言形式的，并且 Windows 本机应用程序绝大多数是使用 C 或者 C++语言开发的，因此 C＼C++编译器和链接器等程序生成工具是程序开发必备的工具。Visual C 和 Visual C++是 Windows 平台下完整的 C 和 C++程序生成工具。PlatformSDK 提供了使用 Windows API 所需的头文件和库文件，同时也提供了部分工具。本章将讲解如何使用这些程序构造工具，以及如何使用微软提供的头文件与库文件来构造 Windows 应用程序。

大型的程序一般有很多代码文件，使用 nmake 工具可以批量构造程序，使开发人员无需对每个文件逐一进行编译。makefile 是 nmake 工具的配置文件，包含 nmake 工作所需的输入参数。本章将会介绍如何使用 nmake 工具，以及如何编写 makefile。

此外，程序开发与调试也密不可分，所以调试器的使用也是本章所关注的内容，本章将会讲解 WinDbg 的使用。

另外，在实际的工程开发中，集成开发环境(IDE)可以大大提供代码编辑的效率和程序构建的效率，同时也为开发过程中代码的调试提供了很大的方便。本章也会讲解 IDE 的使用技巧。

这些工具都可以方便地从微软的网站上免费获得。在开始本章的学习前请读者先按照第 1 章的方法安装 Visual C++ (Visual Studio)和 Platform SDK。

3.1　使用Visual C／C++编译链接工具

在第 1 章已经介绍了如何安装 Visual C++速成版和 Visual Studio 专业版等开发工具。VisualC++的安装目录(Visual Studi0 2008 的默认安装路径是 C:\Program Files\Microsoft Visual Studi0 9)下的“VC\bin”目录中包含 cl.exe、link.exe、lib.exe、rc.exe 等可执行文件；“VC\Include”目录中包含 stdlib.h、stdio.h 等头文件，这些都是标准 C 头文件；在“VC\lib”目录中，则包含 lib 文件和 obj 文件。这些文件在程序编译和链接过程中具有重要作用。

cl.exe 文件是 Visual C\C++的编译器，它将程序源代码文件编译为 obj 文件。

rc.exe 文件是资源编译器。工程项目中的. rc 文件中包含了对程序中所使用资源（菜单、图标等）的描述。rc.exe 将. rc 格式的文件编译为.res 文件，供链接器链接到可执行文件中。

link.exe 是 Windows 平台的链接器，它将 cl.exe 编译生成的 obj 文件，资源编译器生成的.res 文件，以及 lib 目录下的 lib 文件等链接成可执行的 exe 文件、dll 文件等。

3.1.1　编译器cl.exe

cl.exe 的功能是将源代码文件编译为可供链接器使用的 obj 对象文件。cl.exe 命令行参数形式如下：

```
CL [option...] file... [option | file]... [lib...] [@command-file] [/link link-opt...]
```

1．命令行参数

各命令行参数的意义如下。

(1)命令行中的“option”是指一个或多个编译选项。在指定选项时，需要在选项前加正斜杠(/)或一个短划线（_)。有一些选项还会带有参数，使用“：”来指定选项的参数。比如“/sybsystem:windows，5.0”是一个编译选项，表示可执行程序执行所依赖的子系统是Windows(Win32图形用户界面的应用程序)，版本最低5.0 (Windows 2000)。选项名区分大小写(/HELP选项除外)。

而且除了llink选项之外，其他选项的顺序没有要求。在编译时所指定的所有选项都将应用于用户指定的所有源文件。option命令行参数是可选的。

(2)命令行中的“file”是一个或多个所需要编译的源文件，也可以是.obj文件或库文件。cl.exe编译源文件并将.obj文件和库的名称传递给链接器。file参数是编译器的必需参数。

(3)“lib”选项用于指定一个或多个库名。cl.exe直接将这些名称传递给链接器。lib是cl.exe的可选参数。

(4)“command-file”，包含多个选项和文件名的文件，为可选参数。

(5)“link-opt”，一个或多个链接器选项。CL将这些选项传递给链接器，也是可选参数。

本书第1章的实例1-1中，已经使用了cl.exe来编译文件。在编译start.c源文件所使用的cl.exe命令行如下：

```
    cl -c -DCRTAPI1=_cdecl -DCRTAPI2=_cdecl -nologo -GS -D_X86_=1  -DWIN32 -D_WIN32 -W3 -D_WINNT
-D_WIN32_WINNT=0x0500 -D_WIN32_IE=0x0500 -DWINVER=0x0500 -D_MT -MTd /Gz /Fo"START_BIN\\"
/Fd"START_BIN\\" start.c
```

其中“-c -DCRTAPI1= cdecl-DCRTAPI2= cdecl-nologo -GS -D_X86=1 -DWIN32 -D WIN32 -W3-D_vmNT -D_ WIN32 WINNT=0x0500 -D_WIN32 IE=0x0500 -DWINVER=0x0500 -D MT -MTd /Gz/Fo"START BIN\\"/Fd"START BJN\\""都是编译选项（option参数)。“start.c”是源文件(file参数)。没有使用lib、command-file、link-opt等命令行参数。

2．编译器选项

option参数是cl.exe的编译选项，是cl.exe命令行参数中最复杂、也是最常使用的。下面将介绍一些常用的编译选项。

(1)与代码生成有关的选项如表3.1所示。这些选项将影响编译完成后生成的机器代码。

表3.1　　与代码生成有关的编译选项

选　项	用　途
/arch	使用SSE或SSE2指令生成代码，对程序进行优化（仅限x86）
/bigobj	增加.obj文件中的可寻址节数
/clr	生成要在公共语言运行库上运行的输出文件

续表

选　项	用　途
/EH	指定异常处理模型 /EH:a 表示捕获异步（结构化）异常并通知编译器假定 extern C 函数确实引发了异常的异常处理模型 /EH:s 不捕获异步异常并通知编译器假定 extern C 函数确实引发了异常的异常处理模型 /EH:c 如果与 a 一起使用，则为捕获异步异常的异常处理模型。如果与 s 一起使用，则为不捕获异步异常的异常处理模型。在这两种情况下，c 通知编译器假定 extern C 函数从不引发异常
/fp	指定浮点行为
/Ge	激活堆栈探测
/GF	启用字符串池
/Gm	启用最小重新生成
/GR	启用运行时类型信息 (RTTI)
/Gd /Gr /Gz	分别是使用 __cdecl 调用约定（仅限 x86）、使用__fastcall 调用约定（仅限 x86）、使用__stdcall 调用约定（仅限 x86），这 3 个选项是互斥的，不能同时使用
/Gs	控制堆栈探测
/GX	启用同步异常处理
/Gy	启用函数级链接
/MD /MDd /MT /MTd	分别是指使用 MSVCRT.lib 创建多线程 DLL、使用 MSVCRTD.lib 创建调试多线程 DLL、使用 LIBCMT.lib 创建多线程可执行文件、使用 LIBCMTD.lib 创建调试多线程可执行文件，这 4 个选项是互斥的，不能同时使用

(2)与调试有关的编译选项如表 3.2 所示。/27、/Zi 和/ZI 这 3 个选项是对程序进行调试时所必须的选项。同时注意，在生成调试版本的程序时，需要指定/Od 选项以禁用优化。

表 3.2　　与调试有关的编译选项

选　项	用　途
/GS	缓冲区安全检查
/GZ	与/RTC1 相同
/RTC	启用运行时错误检查
/Wp64	检测 64 位可移植性问题
/Yd	将完整的调试信息放在所有对象文件中
/Yl	创建调试库时插入 PCH 引用
/Z7 /Zi /ZI	生成调试信息，分别表示三种不同的调试符号文件格式

(3)与程序优化有关的选项如表 3.3 所示。优化选项将影响编译器对代码的优化，是选择代码体积最小还是选择运行速度最快等需要使用这些选项。

表 3.3　　与程序优化有关的编译选项

选　项	用　途
/O1 /O2 /Od	分别表示代码体积最小、代码速度最快、不使用优化，互斥
/Ob1 /Ob1	控制内联函数展开，/Ob1 表示适用于于__inline 的情况，/Ob2 表示适用于所有情况
/Oi	生成内部函数

续表

选　项	用　途
/Os /Ot	代码大小优先、代码速度优先
/Ox	使用最大优化（/Ob2gity /Gs）
/Oy	省略帧指针（仅限 x86）
/GL	启用全程序优化
/GT	支持使用静态线程本地存储区分配的数据和线程安全

(4)与生成的输出文件有关的选项如表 3.4 所示。输入文件在参数中需要使用字符串来指定其路径，字符串使用双引号(″)扩起来，如/fd″BIN\\″。注意字符串的转意字符，如果需要使用所斜扛表示文件路径，需要使用″\ \ ″。

表 3.4　与输出文件有关的选项

选　项	用　途
/Fd	重命名程序数据库文件
/Fm	创建映射文件
/Fo	创建对象文件
/Fp	指定预编译头文件名
/FR /Fr	生成浏览器文件

(5)与预处理器有关的选项如表 3.5 所示。这些选项将影响预处理器的运行，比如通 i 出 D 选项可以达到与在源代码中使用#define 定义基本相同的效果。

表 3.5　与预处理器有关的编译选项

选　项	用　途
/AÍ	指定包含目录，/AIInclude
/C	在预处理期间保留注释
/D	定义常数和宏，可以将此符号与 #i 或#ifdef 一起使用，以便有条件地编译源文件。在源文件中遇到重定义或用#undef 指定取消定义符号之前，符号定义保持有效。/D 与在源文件的开头使用#define 指令的效果基本相同 示例：cl.exe/DDEBUG start.c
/E	将预处理器输出复制到标准输出
/EP	将预处理器输出复制到标准输出
/Fl	预处理指定的包含文件
/I	在目录中搜索包含文件
/P	将预处理器输出写入文件
/U	移除预定义宏
/u	移除所有的预定义宏
/X	忽略标准包含目录

(6)与链接有关的选项如表 3.6 所示。编译器可以调用链接器，向链接器传递调用参数。/link 选项完成这项功能，还可以指定堆栈的大小，创建动态链接库等。

表 3.6 与链接有关的编译选项

选　项	用　途
/F	设置堆栈大小，/F[ ]number，以字节为单位
/LD /LDd	创建动态链接库，创建调试动态链接库
/link	传递给 LINK 的参数

(7)与预编译有关的选项如表 3.7 所示，表 3.7 中的编译器选项会影响预编译结果。

表 3.7 与预编译有关的编译选项

选　项	用　途
/Y-	忽略当前生成中的所有其他预编译头的编译器选项
/Yc	创建预编译头文件
/Yd	将完整的调试信息放在所有对象文件中
/Yu	在生成期间使用预编译头文件

(8)与程序语言有关的选项在表 3.8 中详细列出。比如/vmb、/vmg 选项将影响编译器对类成员指针的处理方法；/vmm、/vms、/vmv 将会影响类成员继承方式；/Za、/Ze 选项表示是否支持微软的语言扩展，还是只支持标准 C/C++。/Zp 选项可以指定结构中的成员对齐方式，这将影响到生成的程序中结构体的大小。在一些协议头的处理中，/Zp 选项将非常有用。

表 3.8 与语言有关的编译选项

选　项	用　途
/openmp	在源代码中启用 #pragma omp
/vmb /vmg	这些选项选择编译器用来表示指向类成员的指针的方法，分别表示对指向成员的指针使用最佳的基地址，对指向成员的指针具有一般性
/vmm /vms /vmv	指定指向类成员的指针的最通用表示形式分别为多重继承、单一继承和虚拟继承
/Za /Ze	禁用微软语言扩展和启用微软语言扩展。/Za 将与标准 C++ 或标准 C 不兼容的语言构造标志为错误。/Ze 则启用微软语言扩展
/Zc	在/Ze 下指定标准行为
/Zg	生成函数原型
/Zl	从.obj 文件中移除默认库名
/Zp	/Zp[1|2|4|8|16] 结构成员对齐，以字节为单位
/Zs	只检查语法

(9)其他选项。这些选项的影响是多方面的，不太好分类。包括显示帮助信息、设置警告等级、设置版本信息等，详细说明见表 3.9。

表 3.9 其他编译选项

选　项	用　途
@	指定响应文件
/?	列出编译器选项
/analyze	启用代码分析

续表

选　项	用　途
/c	编译但不进行链接
/doc	处理 XML 文件的文档注释
/HELP	列出编译器选项
/hotpatch	创建可热修补的映像
/J	更改默认的 char 类型
/nologo	取消显示登录版权标志
/showIncludes	在编译期间显示所有包含文件的列表
/Tc（/TC）/Tp（/TP）	以 C 代码方式编译，以 C++代码方式编译
/V	设置版本信息
/W	设置警告等级，包括/w、/Wn、/WX、/Wall、/wln、/wdn、/wen、/won /w 表示禁用所有编译器警告 /Wn 为设置警告级别，如/W1 为 1 级（默认），可以指定 0~4 /Wall 启用所有警告（包含那些默认情况下禁用的警告） WX 将所有编译器警告都视为错误

表 3.1～表 3.9 包括了大多数 VC/C++编译器选项，是开发人员在进行程序开发时最经常使用到的选项，但是这并不是全部编译选项。如果读者需要使用到其他的编译选项，可以查阅 MSDN。

> **注意**　如果读者在连接选项中选择了“子系统”为“Windows”（/subsystem:windows），那么应用程序的入口函数应用为“WinMain”。如果选择的子系统为“Console”（/subsyste:console），那么应用程序的入口函数应该为“main”，如果应用程序使用了图形用户界面，那么在编译时，应该选择子系统为“Windows”。由编译链接选项引起的编译不成功的情况还有很多，读者可以参考编译链接选项的说明来解决此类问题。

3.1.2　资源编译器rc.exe

Windows 应用程序中，图标、菜单、位图等是以资源的形式存在的。开发人员也可以自己定义资源类型。如果一个程序使用了资源，那么它在构建时需要对资源进行编译。程序所使用的资源会在资源脚本文件(. rc)中进行描述，编译时，资源编译器会将.rc 文件编译为可链接的.res 文件。资源脚本文件中描述了资源的类型、名称、文件名等信息。资源中可以包括对话框、字符串、图标、位图、鼠标指针样式、工具条等。资源文件中大多和图形用户界面有关，本书将在第 9 章中介绍资源脚本文件的格式。

1．命令行参数

使用如下方式在命令行中运行 rc.exe

```
RC [options] script-file
```

options 是编译选项。script-file 是需要编译的资源脚本文件。

用户可以使用如下方式调用：

```
rc.exe /n /fo start.res start.rc
```

2．资源编译器选项

rc.exe 可使用的选项如表 3.10 所示。

表 3.10　　资源编译器选项

选　项	用　途
/? /h	显示帮助信息
/d	为预处理器定义符号，可以与#ifdef 共同使用
/fo	指定 res 输出文件的文件名，如/fo start
/i	指定包含文件，如果使用了/i 选项，将首先去/i 选项指定的目录中寻找包含文件，然后再去 INCLUDE 环境变量中去寻找包含文件
/l	指定默认的编译语言，比如/l409 相当于在资源脚本文件中指定 LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_US，也就是替换了在 rc 脚本中的如下语句：#pragma code_page(409)
/n	在字符串表（一种资源类型）中的所有字符都将有 0 结束符
/r	忽略与现有 makefile 的兼容性
/v	显示编译器在编译过程中输入的信息
/x	不使用 INCLUDE 环境变量来查找头文件和资源文件

实例 3-1 是一个使用 rc.exe 进行资源脚本文件编译的命令行实例。

3．实例 3-1　编译资源 start．rc

以下是 start.rc 文件内容。

```
IDB_BITMAP1          BITMAP                "F.bmp"
```

打开“Visual Studio 命令提示”，切换到 start.rc 所在的目录，运行：

```
rc.exe /n /v /fo start.res start.rc
```

将会在屏幕上有如下输出：

```
Creating start.res

Using codepage 936 as default
RC: RCPP -CP 936 -f E:\code\ch3\res\RCa01024 -g E:\code\ch3\res\RDa01024 -DRC_INVOKED -D_WIN32
-pc\:/ -E -I. -I .-I C:\Program Files\Microsoft Visual Studio 8\VC\INCLUDE -I C:\Program
Files\Microsoft Visual Studio 8\SDK\v3.0\include -I C:\Program Files\Microsoft Platform SDK for
Windows Server 2003 R2\Include\.

res.rc.
Writing BITMAP:101,    lang:0x804,   size 4296
```

编译完成后，会在目录下生成 start.res 文件，这就是编译完成生成的可供链接器使用的资源文件。在程序链接时，会使用到此文件。

> **注意**　如果编译过程中出现错误信息，如找不到头文件，找不到定义等，说明开发环境中的配置不正确，请按照第 1 章的说明安装开发环境，或参考本章 3.1.5 节和 3.2 节的相关内容。

3.1.3　链接器link.exe

在程序编译完成后，生成的文件是以 obj 为扩展名的对象文件，link.exe 是将对象文件和库链接起来以创建可执行程序文件或动态链接库文件的工具。

link.exe 的输入文件包括 obj 文件、lib 文件、exp 文件、def 文件、res 文件、txt 文件、ilk 文件。输出文件是 exe 文件、dll 文件、sys 文件等可执行程序文件。

1．命令行参数

link.exe 运行参数如下：

```
LINK [options] files
```

参数包括选项和文件名，选项和文件名可按任意顺序指定。link 会首先处理选项，然后是文件。各参数间使用一个或多个空格或制表符来分隔。

2．输入文件

参数中的 files 是指链接器的输入文件。用户可以使用文件名指定绝对路径或相对路径，可在文件名中使用通配符。如果用户指定的文件名没有扩展名，则假设扩展名为 obj。link.exe 对文件格式的判断不是根据文件扩展名的，而是判断文件的内容。

比如实例 1-1 中，start.c 完成编译后会生成 start.obj 文件，然后将 start.obj 与系统库文件链接起来，生成可执行文件。命令行参数为：

```
    link /INCREMENTAL:NO /NOLOGO /-subsystem:windows,5.0 /out:START_BIN\start.exe
START_BIN\start.obj kernel33.lib  User32.lib
```

注意，在实例 1-1 中有这样一行代码：

```
//连接时使用 User32.lib
#pragma comment (lib, "User32.lib")
```

如果在输入文件中指明了 User32.lib，那么这一行是没有必要的。但如果将这一行代码去掉，代码文件中又不包括上面的这一行，则编译器会报错。

> **注意** 如果编译器报告错误“LNK2019: 无法解析的外部符号……”，那么通常是由于没有将链接这些符号所需要的输入库文件加入到输入文件参数中。比如调用 MessageBox API 就必须在链接时输入 User32.lib。

3．运行结果

link.exe 返回零值表示成功。如果运行成功，会在/out 选项指定的输出路径中生成可执行文件或动态链接库。

4．链接器选项

选项前需加“选项说明符”（一个短划线（_）或一个正斜杠(/)），然后是该选项的名称。如果选项带参数，参数在冒号(：)后指定选项的参数。如果参数是数值，可以使用十进制、或 C 语言表示法。与编译器 cl.exe 不同，链接器的选项名及其关键字或文件名参数不区分大小写，但用作参数的标识符区分大小写。

常用的 link.exe 的链接选项如表 3.11 所示。

> **说明** 选项中很多涉及 Windows 可执行文件的格式问题，比如节、导入表、导出表、基址等；以及 Windows 可执行文件中 EXE 文件、DLL 的文件的区别与作用、可执行文件的加载以及 API 函数动态链接过程。Windows 可执行文件编译与链接的内部原理将在第 18 章介绍。

表 3.11　　链接器选项

选　项	用　途
@	指定响应文件
/ALIGN	指定每一节的对齐方式 /ALIGN[:number] number 参数以字节为单位，并且必须是 2 的整数次幂，如 512、1024 等。默认值是 2^{12} 如果要修改特定节的对齐，使用/SECTION 选项
/ALLOWBIND	指定 DLL 不能绑定，生成 DLL 时有效 /ALLOWBIND:NO 在 DL 的头中设置一个位，向 Bind.exe 指示不允许绑定 Image
/ALLOWISOLATION	指定清单查找的行为 /ALLOWISOLATION:NO 说明像没有清单一样加载 DLL /ALLOWISOLATION 指明加载 DLL 时进行清单查找 默认为/ALLOWISOLATION
/BASE	为程序设置基址 选项格式为：/BASE:{address[,size] | @filename,key} /BASE 选项设置程序的基址，重新指定可执行文件的默认加载基址位置，exe 文件（0x400000）、dll 文件（0x10000000）
/DEBUG	创建调试信息 链接器将调试信息放在程序数据库（PDB）中。参考/PDB 选项和编译器的/Zi、/ZI、/Z7 选项
/DEF	将模块定义（.def） 文件传递到链接器 选项格式：/DEF:filename 模块定义文件用于定义模块的导出函数、堆栈大小、节、版本等内容。参考后续章节示例中的.def 文件
/DEFAULTLIB	在解析外部引用时搜索指定的库
/DELAY	控制 DLL 的延迟加载 /DELAY:UNLOAD 导入表被重置为其原始形式，使导入表指针无效并引起改写 /DELAY:NOBIND 指明链接器不要在最终镜像文件中包含可绑定的导入表
/DELAYLOAD	/DELAYLOAD:dllname 延迟加载指定的 DLL
/DELAYSIGN	对程序集进行部分签名 /DELAYSIGN[:NO]表示不签名 参考/KEYFILE 和/KEYCONTAINER 选项脚本文件
/DLL	指定链接器生成 DLL

续表

选　项	用　途
/DRIVER	链接并创建 Windows 驱动程序
/ENTRY	设置起始地址 /ENTRY:function function 为程序的入口函数 必须使用__stdcall 方式调用定义的函数。必须按 WinMain（EXE 文件）或 DllEntryPoint（或 DllMain，DLL 文件）中规定的方式定义参数和返回值。默认情况下，起始地址为 C 运行时库中的函数名。链接器根据程序的属性来选择该函数 如果指定选项/SUBSYSTEM:CONSOLE，默认的入口函数为 main 如果指定选项/SUBSYSTEM:WINDOWS，默认的入口函数为 WinMain
/errorReport	向微软公司报告链接器内部错误
/EXPORT	通过函数名指定导出函数 /EXPORT:entryname[,@ordinal[,NONAME]][,DATA]
/FIXED	创建只能在其首选基址加载的程序，参考/BASE 选项 如果指定生成 DLL，则默认为/FIXED:NO，否则默认为/FIXED
/FORCE	强制完成链接，不论符号是否无法解析或被定义了多次 /FORCE:MULTIPLE，如果引用了多次定义的符号，也链接生成可执行文件 /FORCE:UNRESOLVED，如果引用了未定义的符号，也链接生成可执行文件
/FUNCTIONPADMIN	创建 Hot-fix 映像
/HEAP	设置堆的大小（以字节为单位） 此选项仅在生成 exe 文件时使用 默认堆大小为 1 MB
/IDLOUT	指定.idl 文件等 MIDL（Microsoft Interface Definition Language，微软接口定义语言）输出文件的名称。参考/TLBOUT、/MIDL
/IGNOREIDL	不应将源代码中的任何 IDL 属性处理到.idl 文件中
/IMPLIB	重写默认的导入库名 /IMPLIB:filename filename 为用户指定的导入库名称
/INCLUDE	强制指定将符号添加到符号表中 /INCLUDE:symbol
/INCREMENTAL	控制增量链接 默认情况下，链接器以增量模式运行。可以通过/INCREMENTAL:NO 指定不使用增量连接
/KEYCONTAINER	指定用来为程序集签名的密钥容器
/KEYFILE	指定用来为程序集签名的密钥或密钥对
/LARGEADDRESSAWARE	通知编译器应用程序支持大于 2 GB 的地址 默认为/LARGEADDRESSAWARE[:NO]
/LIBPATH	指定连接使用的库文件所在的路径，与 LIB 环境变量配合使用
/LTCG	指定链接时代码生成 /LTCG[:NOSTATUS|:STATUS|:PGINSTRUMENT|:PGOPTIMIZE|:PGUPDATE]

续表

选　项	用　途
/MACHINE	指定程序运行的硬件目标平台，如 x86 平台 /MACHINE:{ARM|EBC|IA64|MIPS|MIPS16|MIPSFPU|MIPSFPU16|SH4|THUMB|X64|X86} 通常情况下，目标平台在编译时指定，链接器从 obj 文件中判断目标平台
/MANIFEST	创建并运行清单文件，默认设置
/MANIFESTFILE	更改清单文件的默认名称 /MANIFESTFILE:filename
/MAP	创建映射文件，参考/MAPINFO /MAP[:filename]
/MAPINFO	包括映射文件中的指定信息 /MAPINFO:EXPORTS 指明链接器包含映射文件中指定的信息 EXPORTS 为包含的导出函数
/MERGE	合并节 例如，/merge:.rdata=.text，是将.rdata 合并到节.text 中
/MIDL	指定 MIDL 命令行选项 /MIDL:@file File 为包含 MIDL 命令的文件
/NOASSEMBLY	不创建.NET Framework 程序集
/NODEFAULTLIB	在解析外部引用时忽略所有（或指定的）默认库 /NODEFAULTLIB[:library] Library 是链接器在解析外部引用时所需要忽略的库
/NOENTRY	创建无入口函数的纯资源 DLL
/NOLOGO	在连接器启动时不显示版权标志信息
/NXCOMPAT	将可执行文件标记为经测试与 Windows 数据执行保护功能兼容（Vista）
/OPT	优化 /OPT:{REF | NOREF}：是/否消除从未引用的函数、数据 /OPT:{ICF[=iterations] | NOICF}：指定迭代次数，是否支持/OPT:REF /OPT:{WIN98 | NOWIN98}：是/否进行 Windows 98 优化
/ORDER	按顺序放置函数 /ORDER:@filename filename 文件中按顺序写明需要放置的函数，如： WinMain Function1 Function2
/OUT	指定输出文件名（exe 文件、dll 文件） 如果不指定，将使用第一个 obj 文件的名字
/PDB	创建程序数据库 (PDB) 文件 /PDB:filename 参考/DEBGU
/PDBSTRIPPED	创建不具有私有符号的程序数据库 (PDB) 文件，参考/PDB
/PGD	指定按配置优化文件（.pgd），该文件将用于保存正在运行的程序的有关信息 /PGD:filename
/PROFILE	生成一个可与 Profiling 工具结合使用的文件（Visual Studio Team System）

续表

选　项	用　途
/RELEASE	在.exe 文件头中设置校验和
/SAFESEH	使用安全异常处理程序表（SHE） /SAFESEH[:NO]
/SECTION	指定节的属性 /SECTION:name,[[!]{DEKPRSW}][,ALIGN=#] name：节名，不能与标准名称（如.rdata、.text 等）冲突 E、R、W、S、D、K、P 属性分别表示“可执行”、“可读”、“可写”、“共享”、“可放弃”、“在可缓存”、“不可分页”，“!”表示取反 ALIGN 为指定对齐，功能同/ALIGN 选项
/STACK	设置栈的大小（以字节为单位） 此选项仅在生成.exe 文件时使用 默认堆大小为 1 MB
/STUB	将 MS-DOS stub 程序附加到 Win32 程序。如果程序在 MS-DOS 中运行将运行 stub 中的程序，默认添加一个 stub，显示程序不可在 MS-DOS 中运行 /STUB:filename
/SUBSYSTEM	指定子系统，通知操作系统如何运行程序 常用的如下： /SUBSYSTEM ： CONSOLE，Windows 命令行程序 /SUBSYSTEM ： WINDOWS，Windows 窗口程序 /SUBSYSTEM ： NATIVE，设备驱动程序，与/DRIVER 配合使用 /SUBSYSTEM ： POSIX，在 POSIX 子系统中运行 /SUBSYSTEM ： INDOWSCE，在 Windows CE 上运行的应用程序 major，minor：指定版本区间 /SUBSYSTEM:{CONSOLE|EFI_APPLICATION|EFI_BOOT_SERVICE_DRIVER|EFI_ROM|EFI_RUNTIME_DRIVER|NATIVE|POSIX|WINDOWS|WINDOWSCE}[,major[.minor]]
/SWAPRUN	操作系统在运行链接生成的程序前将它复制到交换文件
/TLBID	指定链接器生成的类型库的资源 ID，默认值为 1 /TLBID:id
/TLBOUT	指定.tlb 文件和其他 MIDL 输出文件的名称，参考/IDLOUT
/TSAWARE	创建专为在终端服务器下运行而设计的应用程序 对于 Windows 2000 及更高版本、/SUBSYSTE:CONSOL 选项生成的 Windows 和控制台应用程序，选项是默认设置的
/VERBOSE	打印链接器进度消息，链接器运行过程中的输出将被显示 /VERBOSE[:ICF |:LIB |:REF |:SAFESEH]
/VERSION	为程序文件分配版本号 /VERSION:major[.minor] major：主版本号 minor：次版本号 如：/VERSION:1.0.0.0
/WX	将链接器警告视为错误 与编译器的/WX 类似，参见编译器/WX 选项

3.1.4 其他工具

1. BSCMAKE. EXE

生成一个信息文件(. bsc)，用于浏览程序中的符号（类、函数、数据、宏和类型）信息。可以在集成开发环境的浏览窗口中查看这些信息（.bsc 文件也可以在 IDE 中生成)。

2. LIB. EXE

用于生成 COFF 格式的 lib 库文件，可用于创建导出文件和引用导出定义的导入库，在为其他开发人员提供开发接口时非常有用。

3. EDITBIN.EXE

可用于编辑 COFF 格式的二进制文件。

4. DUMPBIN.EXE

可显示 COFF 二进制文件的有关信息，比如符号表等。

比如，使用 dumpbin.exe 分析第 1 章的 start.exe。选择“工具”→“Visual Studio 命令提示”切换到 start.exe 所在的目录，运行：

```
dumpbin.exe /ALL start.exe >bin_info.txt
```

在 bin_info.txt 中包含了可执行文件中非常丰富的信息，包括文件头信息、节信息等。

```
Dump of file start.exe
PE signature found
File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
            14C machine (x86)
              5 number of sections
       4732F446 time date stamp Thu Nov 08 19:34:30 2007
              0 file pointer to symbol table
              0 number of symbols
             E0 size of optional header
            103 characteristics
                Relocations stripped
                Executable
                32 bit word machine
......
```

5. NMAKE.EXE

程序维护实用工具，读取和执行生成文件。将在 3.3 节中详细介绍 nmake 的使用。

6. ERRLOOK. EXE

错误查找工具，与 GetLastError API 函数配合使用，在程序调试时起到了很多的作用，用于分析 Windows API 的调用错误。如图 3-1 所示，在 Visual Studio IDE 中，可以通过菜单“工具”→“错误查找”来使用。

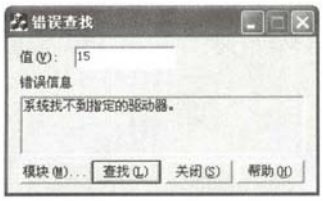

图 3-1 错误查找工具

3.1.5 编译链接工具依赖的环境变量

1．环境变量的概念

环境变量是一组系统或用户给应用程序设置的全局性的变量，通常用于保存各应用程序都会使用到的系统配置信息。应用程序可能会根据不同的环境变量配置产生不同的运行结果。操作系统启动后，会专门分配一块内存用于存储环境变量。之所以称之为“变量”，是因为其值是可以改变的。系统上所有运行的程序都可以获取或设置环境变量。进程创建时，进程可以获取系统环境和当前登录用户的环境变量，每个进程也可以有自己私有的环境变量。因此，环境变量类似于一种在整个系统范围内的“全局”变量。现代的操作系统都有环境变量的概念。

环境变量有一个变量名和一个变量值，都是字符串，形式如下：

```
name=value
```

2．查看和修改环境变量

用户可以在命令行中查看环境变量，直接在命令行中输入不带参数的命令“set”可以显示当前的环境变量，例如：

```
set
```

运行该命令，将打印出很多环境变量信息，形式均为“name=value”。

比如：

```
HOMEDRIVE=C:
Path=C:\Program Files\Microsoft Visual Studio 8\Common7\IDE;C:\Program Files\Microsoft Visual Studio 8\VC\BIN;C:\Program Files\Microsoft Visual Studio 8\Common7\Tools;C:\Program Files\Microsoft Visual Studio 8\Common7\Tools\bin;C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK\bin;C:\WINDOWS\system32; C:\Program Files\Microsoft Platform SDK for Windows Server 2003 R2\Bin\.;C:\Program Files\Microsoft Platform SDK for Windows Server 2003 R2\Bin\WinNT\.;C:\Program Files\Subversion\bin;C:\Program Files\Microsoft Platform SDK for Windows Server 2003 R2\Bin\.;C:\Program Files\Microsoft Platform SDK for Windows Server 2003 R2\Bin\WinNT\.
INETSDK=C:\Program Files\Microsoft Platform SDK for Windows Server 2003 R2\.
LIB=C:\Program Files\Microsoft Visual Studio 8\VC\ATLMFC\LIB;C:\Program Files\Microsoft Visual Studio 8\VC\LIB;C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK\lib;C:\Program Files\Microsoft Visual Studio 8\SDK\v3.0\lib;C:\Program Files\Microsoft Platform SDK for Windows Server 2003 R2\Lib\.
LIBPATH=C:\WINDOWS\Microsoft.NET\Framework\v3.0.50727;C:\Program Files\Microsoft Visual Studio 8\VC\ATLMFC\LIB
SystemRoot=C:\WINDOWS
windir=C:\WINDOWS
```

使用 set 命令也可以设置系统的环境变量：

```
set name=value
```

具体使用方法可以参考 set 命令的帮助，运行：

```
set /?
```

还可以在图形界面中查看和设置环境变量，右键单击“我的电脑”，选择“属性”，在“系统属性”对话框中选择“高级”选项卡，选择“环境变量”便可以查看和修改环境变量，如图 3-2 所示。

3．常用的环境变量

在安装好的系统中，会有一些默认的环境变量，这些环境变量是最常用的，用户最好不要随便修改，如表 3.12 所示。

表 3.12　　默认环境变量

环境变量名	意　义
SystemDrive	系统所在的驱动器，比如 C:
SystemRoot	系统安装目录，比如 C:\WINDOWS
WinDir	Windows 安装目录，比如 C:\WINDOWS
SystemDirectory	系统目录，比如 C:\WINDOWS\System32
ComSpec	Cmd 路径，比如 C:\WINNT\system32\cmd.exe
programfiles	应用程序目录，比如 C:\Program Files
Temp	临时文件夹，比如:\Documents and Settings\Usr\Local Settings\Temp
HOMEPATH	用户的 Home 目录，比如 C:\Documents and Settings\Administrator

4．引用环境变量的方法

绝大多数应用程序，比如桌面浏览器(explorer.exe)和 IE 等，都可以直接引用环境变量。例如在 Windows 桌面浏览器或者 IE 的地址栏中键入“%HOMEPATH%”，回车，就可以转到用户文件夹中，如图 3-3 所示。

图 3-2　设置环境变量

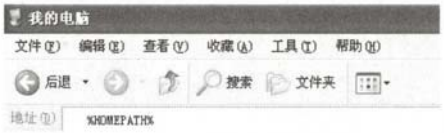

图 3-3　在 explorer 地址栏中使用环境变量

在进行命令行操作时，包括在 cmd 中输入命令或在命令行文件(. cmd)、批处理文件(. bat)中使用环境变量则需要符合如下的形式：

```
%环境变量名%
```

例如“%HOMEPATH%”。

如果在命令行中输入如下命令：

```
SET PATH=C:\Program Files\myprogram\bin;%PATH%
```

就可以在原来的 PATH 环境变量取值中增加一个“C:\Program Files\myprogram\bin”。

在 makefile 中引用环境变量的方法与引用 makefile 宏的方法是一样的（参见 3.3.4 小节），在需要引用环境变量的位置输入：

```
$(环境变量名)
```

5．PATH 环境变量

PATH 环境变量的值是可执行文件路径。用户在输入命令的时候可以使用文件的全路径，比如在命令行中输入：

```
C:\windows\notepad.exe
```

可以启动记事本程序。但是这样做很麻烦，事实是，无论把命令行切换到什么目录下都直接可以直接使用输入“notepad”来启动记事本。那么当用户运行一个程序或者创建一个

进程时，系统是如何获知可执行文件位置的呢？在寻找可执行文件的路径时，系统遵循如下顺序逐个目录查找输入命令所指定的程序。

(1)父进程所在的目录。

(2)进程的当前目录。

(3)系统目录。

(4) Windows 安装目录。

(5) PATH 变量所指向的目录。

PATH 变量的作用就在于此。因此将 cl.exe、link.exe 所在的目录加入到 PATH 环境变量中，就可以通过直接在命令行中键入程序可执行文件的文件名来运行程序。

6．编译工具使用的环境变量

下面讲解开发工具与环境变量的关系。环境变量在开发环境中最重要的作用就是：编译器、链接器等程序构建工具都需要使用环境变量来查找包含文件、库文件所在的目录等，如表 3.13 所示。

表 3.13　　编译器及连接器使用的环境变量

工具	环境变量	作　用
cl.exe	CL	如果定义了“CL”环境变量，那么在处理命令行之前，cl.exe 处理在“CL”环境变量中定义的选项和参数
	INCLUDE	这个目录中包括了 cl.exe 所需要使用的包含文件，如头文件等，如果开发人员需要将其他目录中的文件包含到编译器可找到的路径中，需要将路径设置到“INCLUDE”环境变量中，多个路径间使用“;”隔开。在 C 语言源文件中使用#include 引用文件时，文件路径必须是以源文件所在目录为起始点的相对路径，或者是 INCLUDE 环境变量所指向的目录中的文件。如果进行标准 C/C++开发，那么 Visual C++安装目录（\VC）的\include 子目录必须位于此环境变量中，如果使用 Windows API 进行开发，那么 Platform SDK（参见 3.2 节）的安装目录中的\Include 必须位于此环境变量中
	LIBPATH	C++中使用#using 引用的元数据文件的目录
link.exe	LINK	如果存在“LINKE”环境变量，那么在处理命令行参数之前，link.exe 会处理在“LINK”环境变量中定义的选项和参数
	LIB	link.exe 在链接中会从如下目录上搜索所需要使用的对象文件和库文件：（1）当前路径；（2）命令行上由/BASE 选项指定的目录；（3）使用“LIB”环境变量中的路径。link.exe 还会使用“LIB”变量指定的路径查找在对象文件中指定的.pdb 文件。多个路径间使用“;”隔开。如果进行标准 C/C++开发，那么 Visual C++安装目录的\lib 子目录必须位于此环境变量中，如果使用 Windows API 进行开发，那么 Platform SDK（参见 3.2 节）的安装目录中的\lib 子目录必须位于此环境变量中
	PATH	link.exe 需要使用到一些其他的工具（比如 cvtres.exe），这些工具所在目录必须位于“PATH”路径下。这此工具位于 Visual C++安装的\bin 子目录中。同时如果想直接在命令行中运行 cl.exe、link.exe，也需要在 PATH 中添加 Visual C++安装的\bin 子目录。此处 nmake 工具也会使用到 PATH 变量来寻找编译和链接工具（参见 3.2 节）
	TMP	链接时寻找 OMF 或.res 文件时，会从 TMP 指定的目录中寻找

7．Visual Studio 命令提示

在前两章的实例中，使用了“Visual Studi0 2005 或 2008 命令提示”来编译示例程序。那么“Visual Studi0 2005/2008 命令提示”是如何工作的呢？为什么直接在命令提示(cmd.exe)中编译程序可能会出现错误，而在 Visual Studio 命令提示中进行链接就不会出现错误，原因就在于 VisualStudio 命令提示与普通的命令提示有不同的环境变量配置。

"Vsual Studi0 2005/2008 命令提示”是一个快捷方式，它所指向的内容为：

```
%comspec% /k ""C:\Program Files\Microsoft Visual Studio 8\VC\vcvarsall.bat"" x86
```

“%comspec%”引用了环境变量，其值是 cmd.exe。\k 参数是指执行后面的字符串参数。

也就是会以“x86”为参数运行“C:\Program Files\Microsoft Visual Studio 8\VC\vcvarsall.bat”（根据VisualC++安装目录不同而不同）。

运行vcvarsall.bat时，如果是以“x86”为参数，最终会调用\Common7\Tools\vsvars32.bat。

再来分析一下vsvars32.bat的内容（以Microsoft Visual C++ 2005速成版中\Common7\Tools\vsvars32.bat为例，以下在部分文件内容）：

```
    @SET VSINSTALLDIR=C:\Program Files\Microsoft Visual Studio 8
    @SET VCINSTALLDIR=C:\Program Files\Microsoft Visual Studio 8\VC
    @SET FrameworkDir=C:\WINDOWS\Microsoft.NET\Framework
    @SET FrameworkVersion=v3.0.50727
    @SET FrameworkSDKDir=C:\Program Files\Microsoft Visual Studio 8\SDK\v3.0
    ……
    @set DevEnvDir=C:\Program Files\Microsoft Visual Studio 8\Common7\IDE

    @set PATH=C:\Program Files\Microsoft Visual Studio 8\Common7\IDE;C:\Program Files\Microsoft
Visual Studio 8\VC\BIN;C:\Program Files\Microsoft Visual Studio 8\Common7\Tools;C:\Program
Files\Microsoft Visual Studio 8\SDK\v3.0\bin;C:\WINDOWS\Microsoft.NET\Framework\v3.0.50727;C:\
Program Files\Microsoft Visual Studio 8\VC\VCPackages;%PATH%
    @set INCLUDE=C:\Program Files\Microsoft Visual Studio 8\VC\INCLUDE;%INCLUDE%
    @set LIB=C:\Program Files\Microsoft Visual Studio 8\VC\LIB;C:\Program Files\Microsoft Visual
Studio 8\SDK\v3.0\lib;%LIB%
    @set LIBPATH=C:\WINDOWS\Microsoft.NET\Framework\v3.0.50727
    ……
```

分析得出，这个批处理文件设置了多个环境变量，包括表示Visual C++安装目录的VCINSTALLDIR等，增加了若干的PATH路径，将\include子目录增加到环境变量INCLUDE中，将\lib路径增加到LIB环境变量中等。

以上就是在编译程序时需要使用“Visual Studio命令提示”的原因。

3.1.6 示例：使用/D选项进行条件编译

本示例中，将演示如何使用/D选项与预处理声明配合达到使用相同代码编译生成不同程序的目的。

实例3-2 使用ID选项进程条件编译

笔者在实例中定义了一个宏，如果预定义了“DEBUG_PRINT”，那么宏展开为打印语句；如果没有预定义，宏展开为空。代码如下所示：

```
/* ************************************
* 示例代码
* define.c
* 3.1.6 示例：使用/D选项进行条件编译
**************************************/
/* 预处理 */
```

```
/* 头文件 */
#include <stdio.h>

/* 判断声明 */
#ifdef DEBUG_PRINT
#define MyPrint printf("hello\n")
#else
#define MyPrint
#endif

// main 函数
int main()
{
     MyPrint;
     return 0;
}
```

如果为编译器加上选项/DDEBUG_PRNT(或/D”DEBUG_PRINT”)，进行预处理器声明，那么编译生成的程序运行后将打印输出“hello”，如果没有进行预处理器声明，那么程序运行后将没有任何输出。

3.2 使用Platform SDK

Platform SDK 包括使用 Windows API 进行开发所必需的头文件和库文件。如果没有 Platform SDK，VCN C++将不能识别程序中所使用的各种 Windows API 函数、Windows 数据类型、各种结构等，最终引起编译错误。安装 Platform SDK 是进行 Windows 本机应用程序开发的必要条件。

Visual Studio 专业版以上版本已经内置了 SDK，并配置好了执行环境，所以使用 VisualStudiO 2005 专业版的读者不需要单独安装 SDK。但是了解 SDK 工具的功能、安装与配置方法将对熟悉开发工具有很大的帮助，会对开发过程中的各种错误排除有积极作用，帮助用户分析问题产生的原因。

3.2.1 Platform SDK的目录结构与功能

用户可以按照第 1 章的说明，完整安装 Platform SDK。

1．开始菜单中快捷方式

安装完成后用户可以先查看开始菜单，在开始菜单中包括以下一些内容。

• Open Build Environment Window 目录：包括若干种用于编译环境设置的批处理文件，可以使用这些文件来设置程序的编译环境（参见 3.3.2 小节）。

• Tools 目录：一些实用的工具，将在 3.2.3 小节详细介绍这些工具。

• Platform SDK Documentation 目录：SDK 文档，编写 Windows API 应用程序的主要参考。

• Install Debugging Tools for Windows：指向调试器 WinDbg 安装的快捷方式（有关 WinDbg 的使用，参见 3.4 节）。

有兴趣的读者可以再看看安装文件夹，其中一般会包含如图 3-4 所示的目录和文件(笔者使用 Windows Server 2003 R2 版的 Platform SDK)。

图 3-4　Platform SDK 目录结构

2．Bin 目录

Bin 目录中包含一些在开发中可以使用的实用的工具，如 nmake.exe，还有进行文件内容比较的工具 WinDiff.exe 查看文件调用依赖的工具 Depends.exeHelp 等。

3．Help 目录

包含帮助文档文件，从开始菜单的快捷方式打开，SDK 帮助文件包括了详细的 API 使用说明等内容，是开发 Windows 应用程序最权威的参考资料，作用巨大。

4．lnclude 目录

包含文件夹，包括 API 声明的各种头文件、各种预定义的 Makefile 文件、接口定义语言文件等。

5．Lib 目录

库文件夹，包含链接工具可能使用到的各种库文件、对象文件等。

6．License 目录

微软版权与许可证信息。

7．Samples 目录

示例代码,包括多个使用 SDK 进行 Windows 应用程序开发的示例源代码,是学习 Windows 程序设计的重要参考。

8．Setup 目录

包括设置各种编译环境的批处理文件和部分工具的安装文件。

9．src 目录

包括 atl、crt (C 运行时)、mfc 的源代码。

10．ReleaseNotes.Htm 文件

发布版本信息。

11．SetEnv.Cmd 文件

设置编译链接环境的批处理文件，包括设置环境变量、设置 nmake 宏等。

3.2.2　为编译链接工具设置环境变量

编译器 cl.exe、链接器 link.exe 采用何种方式找编译时所需的头文件和库文件？答案是依赖于环境变量。因此在 SDK 的安装完成后，需要将 SDK 中的 Bin 目录、Include 目录和 lib 目录设置到环境变量中。在 3.1.5 小节已经简单介绍了编译链接工具所需要使用的环境变量与设置方法，下面是具体的设置方法。

1．方法一：在安装时选择完全安装

在安装时选择完全安装，包括“Configuration Options\Register Environment Variables”选项。

2．方法二：手工设置环境变量

设置如下环境变量。

MSSdk 变量：值为 SDK 安装目录。

Mstools 变量：值为 SDK 安装目录。

INETSDK 变量：值为 SDK 安装目录。

Bkoffice 变量：值为 SDK 安装目录。

Basemake 变量：值为%MSSdk%\Include\BKOffice.Mak。

1NCLUDE 变量：值为%MSSdk%\Include\.。

LIB 变量：值为%MSSdk%\Lib\.。

并在 Path 变量前加上%MSSdk%\Bin\.;%MS Sdk%\Bin\WinNT\.。

3．方法三：使用 Platform SDK 提供的批处理文件

直接使用 Open Build Environment Window 中的各种批处理文件。比如要编译适用于 32 位 Windows XP 的调试版本程序，可以使用“Set Windows XP 32-bit Build Environment (Debug)”快捷方式。

快捷方式的目标如下所示：

```
C:\WINDOWS\system32\cmd.exe /E:ON /V:ON /T:0E /K "C:\Program Files\Microsoft Platform SDK for Windows Server 2003 R2\Setup\SetEnvLaunchWinXP32Debug.Cmd" /XP32 /DEBUG
```

使用“/XP32”和“/DEBUG”参数调用了 Setup 目录下的命令行文件 SetEnvLaunch WinXP32Debug.Cmd。

分析 SetEnvLaunchWinXP32Debug.Cmd 内容，有如下调用：

```
start cmd.exe /k setenv.cmd %1 %2 %3
```

使用了 SDK 安装目录下的 SetEnv.cmd 文件来设置编译链接的环境。

setenv.cmd 文件中有如下内容：

```
Set MSSdk=C:\Program Files\Microsoft Platform SDK for Windows Server 2003 R2
```

```
Set Bkoffice=%MSSdk%\
Set Basemake=%MSSdk%\Include\BKOffice.Mak
Set INETSDK=%MSSdk%
Set MSSdk=%MSSdk%
Set Mstools=%MSSdk%
……
Set Lib=%MSSdk%\Lib;%Lib%
Set Include=%MSSdk%\Include;%Include%
Set Path=%MSSdk%\Bin;%path%
Set CPU=i386
Set TARGETOS=WINNT
Set APPVER=5.01
```

以上脚本完成了环境变量的设置，其中“MSSdk”项会根据安装目录的不同而不同。

setenv.cmd 还设置了一些 nmake 所使用的宏，使在不同参数输入的情况下，宏的值不同，这些宏会影响到编译时使用的头文件版本、目录系统版本、输入目录、是否生成调试文件等，读者可以自行分析。

4．总结与分析

现在再来分析一下 3 .1.5 小节中所提到的“Visual Studio 命令提示”，读者就会明白为什么在安装 Platform SDK 后，如果没有设置环境变量就直接使用“Visual Studio 命令提示”来调用 nmake 编译实例程序，会导致 cl 报告找不到头文件。原因就是在环境变量

中没有包括 Platform SDK 中的相关目录。

> 注意
>
> （1）如果找不到 nmak.exe、cl.exe、link.exe 等可执行文件，请检查 Visual C++安装目录下 VC\bin\目录是否在 PATH 环境变量中。
>
> （2）如果 nmake.exe 找不到“Win33.Mak”等包含文件，请检查 Platform SDK 中的 Include\文件是否在“Include”环境变量中。
>
> （3）如果 cl 找不到“Windows.h”等 SDK 头文件，请检查 Platform SDK 中的 Include\文件夹是否在“Include”环境变量中。
>
> （4）如果 cl 找不到“stdio.h”、“stdlib.h”等标准 C 头文件，请检查 Visual C++安装目录下的 VC\include\目录是否在“Include”环境变量中。
>
> （5）如果 link 找不到库文件，请检查 Platform SDK 安装目录中的 Lib\文件夹和 Visual C++的 VC\lib\文件夹是否在“Lib”环境变量中。

3.2.3 Platform SDK工具集

Platform SDK 目录下包含很多工具。这些工具在调试、测试、性能测试、可执行文件查看等方面有比较强的功能。

depends.exe：系统调用依赖查看工具，可以用来查看可执行文件的导入表和导出表，如图 3-5 所示。

WinDbg：功能强大的调试器，将在 3.4 节详细介绍，安装文件 L+\setup 子目录下。

rc.exe:资源编译器。

mc.exe:消息编译器。

bind.exe：绑定 DLL，将 DLL 导出的函数的地址存储到地址表中，调用 DLL 的导出函数的时候，可以通过地址表进行直接的跳转。

windiffexe:比较文件的不同。

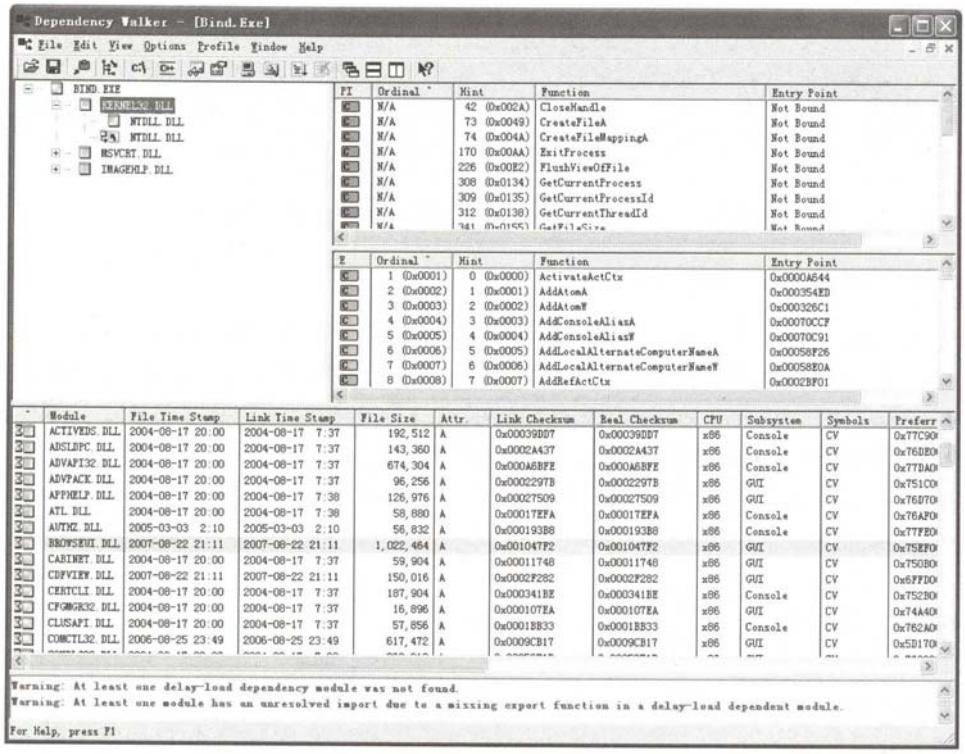

图 3-5　depends.exe 界面—分析 Bind.exe

where.exe:查找文件。

PerfMtr:查看系统工作性能。

pfmon.exe:进程错误监视器。

rebase.exe:指明应用程序所使用的 DLL 的基址。

pstat.exe:显示进程和模块的统计信息，统计的信息非常全面，包括系统整体内存的使用情况，各个进程的用户态时间，内核时间、错误数、句柄、线程数、PID、TDI 等。下面是笔者主机上的输出信息：

```
PageFile: \??\C:\pagefile.sys
     Current Size: 1572864 kb  Total Used: 177752 kb   Peak Used 249900 kb

 Memory:1046056K Avail: 477924K  TotalWs: 503884K InRam Kernel: 2048K P:52972K
 Commit: 547996K/ 394024K Limit:2516792K Peak: 723284K  Pool N:22264K P:70940K

    User Time   Kernel Time    Ws   Faults  Commit Pri Hnd Thd Pid Name
  0:00:00.000   0:00:46.171   100     8145      28  8  653 108   4 System
  0:00:00.015   0:00:00.046  3036      834    2196  8   33   1 2320 cmd.exe
  0:00:00.015   0:00:00.031  1436      406     540  8   15   1 2904 PStat.Exe
......


......
pid:910 pri: 8 Hnd:   33 Pf:    834 Ws:   3036K cmd.exe
 tid pri Ctx Swtch StrtAddr    User Time  Kernel Time  State
 234  11       792 7C810665  0:00:00.000  0:00:00.046 Wait:UserRequest

pid:b58 pri: 8 Hnd:   15 Pf:    406 Ws:   1436K PStat.Exe
 tid pri Ctx Swtch StrtAddr    User Time  Kernel Time  State
 a9c  11        22 7C810665  0:00:00.000  0:00:00.031 Running
```

```
  ModuleName Load Addr   Code    Data   Paged          LinkDate
-------------------------------------------------------------------------------
ntkrnlpa.exe 804D8000  475136  106496 1179648 Wed Feb 28 16:38:53 2007
……
  ntdll.dll 7C920000  503808   20480       0 Tue Aug 17 07:38:36 2004
-------------------------------------------------------------------------------
      Total         11203392 2638048 5282048
```

构造好的可执行文件将会自动出现在指定目录中。

vadump.exe：虚拟地址转存，统计每个进程各个模块的进程信息情况。

3.2.4 Windows Vista SDK

随着 Windows Vista 的发布，Windows Vista SDK 也已经发布，Vista SDK 的使用方法与之前版本的 SDK 使用方法类似。与其他版本的 SDK 所不同的是 Vista SDK 附带完整的编译链接工具（Windows XP、Server 2003 等版本的 SDK 大多不包括最为重要的 cl.exe），这样开发人员就可以不再依赖于 Visual Studio 只需要使用 SDK 就可以完成应用程序的开发。

但是 SDK 并不包含集成开发环境(IDE)的功能，如果应用程序开发人员需要 IDE，那么还是需要安装 Visual Studio。

3.3 编写Makefile

在使用 Visual Studio 等 IDE 时，由于程序的所有编译过程都由 IDE 自动配置，所以很多习惯于使用 IDE 的程序员，尤其是 Windows 平台的程序开发人员，并不知道如何编写 makefile。

因此这些开发人员，在编写程序时，面对各种程序编译过程中的错误和问题可能会手足无措。而作为一个专业的程序员，对程序的编译链接过程应该有足够的了解，尤其应该了解程序的编译和连接选项对程序编译以及程序代码的编写等方面所造成的影响。

makefile 是 nmake 所使用的配置文件。在 makefile 的帮助下，程序编译的效率得到极大提高。编写 makefile 需要对 cl.exe、rc.exe、link.exe 等程序的运行参数有深入的了解，还需要了解 PlatformSDK 的使用方法。在此基础上，本节将介绍如何编写 makefile。

3.3.1 使用nmake.exe构建工程

通常，在实际工程项目中，为了调试与测试，在开发过程中需要频繁地构建代码。如果代码文件的数量都比较庞大时，仅仅使用 cl.exe 工具对单个代码文件进行构造，然后再用 link.exe 进行链接，将会在程序的编译上消耗大量时间。因此需要一种更方便的代码编译方式。

nmake 的全称是“程序维护实用工具”。nrnake 可以在配置文件的基础上自动化地调用 cl.exe、rc.exe、link.exe 等工具对构建代码，开发人员只需要运行一条命令，就可以完成整个开发工程的程序编译工作。

nmake 所使用的程序构建配置文件就是 makefile。本书第 1 章 1.3.6 小节已经演示了如何使用 makefile 来编译实例 1-1。编译时，只需要切换到 makefile 所在的文件夹，运行：

```
nmake.exe
```

1.nmake 命令行参数 nmake.exe 的完罄命令行参数如下：

```
nmake [options] [/f makefile] [/x stderrfile] [macrodefs] [targets]
```

或者

```
nmake @commandfile
```

第二种方式中 commandfile 是存储命令行参数的文本文件。

第一种方式中直接使用了参数，各参数的意义如下

. / f 参数：指明使用的 makefile 文件名，如果不指定，则使用名为“makefile”的文件。

. / X 参数：指定错误输出文件，见/X 选项。

. macrodefs:定义宏。

2．目标

目标（Targets）是指明本次生成程序的目标是什么，具体的目标需要在 makefile 中指明，nmake
会从 makefile 选择目标，并进行生成。

3．选项

NMAKE 的选项(Options)有很多种，选项使用“ / ”或“. ”作为其前缀，选项名不区分大小写，具体的选项与作用参见表 3.14。

表 3.14　　NAMKE 选项

选　项	作　用
/A	强制生成所有相关目标，即使这些目标相对于依赖项未过期；不强制生成不相关目标
/B	如果目标与依赖项时间戳相等（解析为 2s 或小于 2s），也运行命令。如果用于编译的系统运算速度非常快，可以使用本选项
/C	不显示默认输出，包括非严重的错误或警告、时间戳以及版权信息。取消/K 选项发出的警告
/D	当目标不存在时，显示每个已检测的目标、依赖项和消息的时间戳。与 /P 选项一起用于调试生成文件
/E	用环境变量替换 Makefile 中的宏定义
/ERRORREPORT	向微软报告在运行时 nmake 的内部错误 /ERRORREPORT[NONE | PROMPT | QUEUE | SEND]
/F filename	指定 makefile，filename 为 makefile 的文件名，前面有空格或制表符
/G	显示!INCLUDE 指令中包含的生成文件，参考 NMAKE 预处理
/HELP, /?	显示帮助信息
/I	忽略所有命令的退出代码（返回值）
/K	如果命令返回错误（cl、link 等），则继续生成不相关的依赖项。默认情况下，如果有任何命令错误，NMAKE 将暂停。使用时参考/I /C 选项
/N	显示但不执行命令，仅执行预处理命令
/NOLOGO	不显示 NMAKE 的版权消息
/P	显示标准输出的信息然后再运行。如果不存在任何生成文件和命令行目标，则只显示信息。与/D 选项一起用于调试生成文件
/Q	检查目标的时间戳，不运行生成。如果所有目标都是最新的，则返回零退出代码；如果有任何目标不是最新的，则返回非零值，同时还执行预处理命令 通过批处理文件运行 NMAKE 时会很有用
/R	清除.SUFFIXES 列表，忽略在 Tools.ini（nmake 在处理 makefile 前会处理 Tools.ini）文件中定义的或者预定义的推理规则和宏

续表

选　项	作　用
/S	取消已执行命令的显示
/T	更新命令行目标（或第一个生成文件目标）的时间戳并执行预处理命令，但不运行
/U	必须与/N 选项一起使用。转储内联 NMAKE 文件，以便/N 输出可用作批处理文件
/X filename	将 NMAKE 错误存储到文件，filename 为文件名，文件名前有空格或制表符
/Y	禁用批模式推理规则，所有批模式推理规则被视为常规推理规则

3.3.2 Makefile实例

首先展示一个 makefile 的例子。3.3.3～3.3..6 小节将在本例的基础上详细介绍描述块、目标、依赖项和命令等概念和 makefile 的基本元素。

实例 3-3　makefile 示例

本实例展示了一个 makefile 的必要内容及其组织结构。

```
    # Nmake macros for building Windows 32-Bit apps

    !include <Win33.Mak>
    OUTDIR = START_BIN

    ##---------------------- Build Rule
------------------------------------------------------------
    all: $(OUTDIR) $(OUTDIR)\start.exe

    #----- If OUTDIR does not exist, then create directory
    $(OUTDIR):
       if not exist "$(OUTDIR)/$(NULL)" mkdir $(OUTDIR)

    #----- compile ----------------------------------------
    $(OUTDIR)\start.obj: start.c
       cl -c -DCRTAPI1=_cdecl -DCRTAPI2=_cdecl -nologo -GS -D_X86_=1  -DWIN32 -D_WIN32 -W3 -D_WINNT
-D_WIN32_WINNT=0x0500 -D_WIN32_IE=0x0500 -DWINVER=0x0500  -D_MT -MTd /Gz /Fo"$(OUTDIR)\\"
/Fd"$(OUTDIR)\\" start.c
    #----- link -------------------------------------------
    $(OUTDIR)\start.exe: $(OUTDIR)\start.obj
       link /INCREMENTAL:NO /NOLOGO -subsystem:windows,5.0 -out: $(OUTDIR)\start.exe
$(OUTDIR)\start.obj kernel33.lib  ws2_33.lib mswsock.lib advapi33.lib bufferoverflowu.lib

    ##------------------- Clean Rule
-----------------------------------------------------------
    # Rules for cleaning out those old files
    clean:
              if exist $(OUTDIR) rd /s /q $(OUTDIR)
```

实例 3-2 中的 makefile 文件中包括注释、宏、描述块、预处理指令等部分。

3.3.3 注释

以“#”开始的行是注释，makefile 文件是以“#”符号表示注释。本实例中第一行就是注释。

3.3.4 宏

makefile 中的宏与 C 语言中宏的概念类似，makefile 中的宏也是一种替换关系。宏就是用一个字符串替换另一个特定字符串。比如，实例 3-2 中的“OUTDIR”就是一个宏。

1．宏的定义

宏的定义很简单，直接使用“=”，“=”前是宏名，“=”后是宏的值。比如实例 3-1 中宏 OUTDIR 的定义如下：

```
OUTDIR = START_BIN
```

2．宏的引用

使用“$（宏名）”引用宏。比如实例 3-1 中的$(OUTDIR)。在 nmake.exe 使用 makefile 时，宏将被展开，比如在实例 3-1 中，所有出现“$(OUTDIR)”的地方都会被“START——BIN”替换。

3．预定义宏

上面的 OUTDIR 是用户定义的宏，Platform SDK 已经为用户定义了很多宏，称为预定义宏。

实例 3-1 中的“$(NULL)”就是一个预定义宏。实际上 Platform SDK 也定义了“OUTDIR”这个宏，只是在笔者自定义的 makefile 中，重新定义了这个宏，这将覆盖之前定义的宏的值。如果在实例 3-2 的 makefile 中去掉“OUTDIR= START_ BIN”，makefile 也是正确的。

4．特殊宏

除了上面这种用户自定义的宏和在 Platform SDK 中定义的预定义宏之外，还有一些特殊的宏，包括文件名宏、递归宏等。文件名宏表示了一些目标、依赖项的路径（目标、依赖项的概念参见 3.3.5 节），递归调用宏是在 makefile 中再指定调用的宏，如表 3.15 所示。

表 3.15　　特殊宏

宏	含　义
$@	当前目标的全名（全路径）
$$@	当前目标的全名（全路径），作为依赖项
$*	除去了文件扩展名的当前目标的全名
$**	当前目标的所有依赖项
$?	时间戳晚于当前目标的时间戳的所有依赖项
$<	时间戳晚于当前目标的时间戳的所有依赖项。在推理规则的命令中有效
MAKE	调用 NMAKE
MAKEDIR	调用 NMAKE 时的当前目录
MAKEFLAGS	当前有效的选项（去掉/F），使用方法如下 /$(MAKEFLAGS)

上表中，以“$”开头的是文件名宏，还可以使用修饰符来截取宏展开部分结果，如“D”是取全路径中去掉文件名的路径，即文件所在的文件夹的路径；“B”是取文件名，不包括文件夹路径和文件扩展名；“F”是取包括扩展名的文件名。

5．实例 3-4　makefile 中的宏

本实例演示如何在 makefile 中定义与使用宏和环境变量。

```
#--- 用户自定义宏 ---
OUT_DIR = subdir
```

```
WINDIR = OK

all:showpath $(OUT_DIR)\showmacro.exe recurssion

#---- 演示引用环境变量 ----
showpath:
	echoshwo environment variable macro
	echoWINDIR=$(WINDIR)
	echo---

#--- 演示文件名宏 ---
$(OUT_DIR)\showmacro.exe:
	echoshow file name macro
	echo$(@D)
	echo$(@F)

recurssion:
    echo ---
    $(MAKE) /N /$(MAKEFLAGS) /F s.mak showpath
```

在“Visual Studio 命令提示”中切换到相关目录下，运行：

```
nmake /NOLOGE /S /E /F s.mak
```

会有如下输出：

```
shwo environment variable macro
WINDIR=C:\WINDOWS
---
show file name macro
subdir
showmacro.exe
---
	echoshwo environment variable macro
	echoWINDIR=C:\WINDOWS
	echo---
```

注意 要理解本实例中 makefile 的内容需要学习 3.3.5 小节的内容。3.3.5 小节包括本节中所提取的“目标”、“依赖项”等概念的解释以及 makefile 的处理过程。

1．宏的工作原理

所有的宏引用“$(OUT_DIR)”都被替换为“subdir”，在 makefile 的开头处进行定义。

命令“echo”的作用是将其后的字符串显示于屏幕上。

目标“showpath”中第一个 echo 显示完整的字符串“shwo environment variable macro”：第二个 echo 后面的字符串中“$(WINDIR)”已经被替换为“C:\WINDOWS”，这是一个环境变量宏，使用了环境变量“WINDIR”的值“C:\WINDOWS”来替换（此处的显示结果之所以不是用户定义的“OK”，原因参见第 2 点中对 nmake 的选项分析）。

目标“subdir\showmacro.exe”中的两个 echo 将当前目标“subdir\showmacro.exe”的名称路径输入，$(@D)被替换为文件夹，$(@F)被替换为包括扩展名的文件全名，这是两个文件名宏。

目标“recurssion”调用了一个递归宏，在现在选项的基础上又增加了选项/N，并指定目标为“showpath”。

2．nmake 选项

```
nmake /NOLOGE /S /E /F s.mak
```

第一个选项/NOLOGE：nmake 没有输入版权信息(Microso 胸. ...)，第二个选项/S：nmake 没有输出其运行的命令(echo…)，第三个选项/E：使用环境变量来替换用户定义的宏，因此输出结果才是环境变量的值，而不是“OK”，第四个选项/F：指定了使用的 makefile 文件名 s.mak。

在目标“recurssion”中递归调用了 nmake，在现在选项的基础上又增加了选项/N，并指定目标为“showpath”，选项/N 指明 nmake 只需要将命令显示出来，但是不执行，所以

nmake 显示了 showpath 目标中的所有命令。

3.3.5 描述块：目标、依赖项和命令

描述块是 makefile 的主体。实例 3-2 中有 5 个描述块，下面就是一个描述块：

```
    $(OUTDIR)\start.obj: start.c
        cl -c -DCRTAPI1=_cdecl -DCRTAPI2=_cdecl -nologo -GS -D_X86_=1  -DWIN32 -D_WIN32 -W3 -D_WINNT
-D_WIN32_WINNT=0x0500 -D_WIN32_IE=0x0500 -DWINVER=0x0500   -D_MT -MTd /Gz /Fo"START_BIN\\"
/Fd"START_BIN\\" start.c
```

描述块的结构如下：

```
目标: 依赖项
    命令
```

1．目标

所谓目标是用户最终希望得到的结果，也就是 nmake 需要生成的结果。目标可以是一个文件、目录，也可以什么都不是。如果目标不存在或者目标的时间戳（文件的最后修改时间）比依赖项早，或者目标类型不是文件，mnake 将运行描述块中的“命令”。

2．依赖项

依赖项是指在生成目标所需要使用到的对象。如上例中，生成“$(OUTDIR)\start.obj”（宏展开为“START BIN\start.obj”）需要“start.c”。

一个目标可以有一个或多个依赖项，也可以没有依赖项。多个依赖项以空格分隔。如果指定的依赖项不存在，则在其他描述块的目标中寻找，但首先需要生成这个目标。

3．命令

命令是 nmake 在生成目标时所调用的命令。与用户自己在命令行中执行效果是一样的。

4．nmake 的命令行参数

有了上面的基础知识，再回来看一下 nmake 的命令行参数。

```
    NMAKE [options] [/f makefile] [/x stderrfile] [macrodefs] [targets]
```

其中参数“[targets]”就是指 makefile 文件中的“目标”，如果不指定，则默认为 makefile 中的第一个目标。在调用时使用 / f 参数指定 makefile 文件的名字，如果不指定，则使用当前目录下的文件名为“makefile”的文件。

5．时间戳判断机制

在使用 nmake 进行程序构建时，nmake 采用了时间戳判断机制。在生成一个目标时，会判断目标文件是否存在或目标的最后修改时间是否晚于所有依赖项的最后修改时间。如果所有依赖项的最后修改时间都比目标的最后修改时间晚，则说明当前的目标文件是使用现有的依赖项生成的，是最新的，没有必要再进行生成。

6．nmake 依赖于 makefile 的处理过程

下面以对第 1 章实例 1-1 进行编译的内部过程为例，说明 makefile 配置 nmake 工作的原理。

◇ 运行 nmake.exe，没有指定参数，所以使用的 makefile 为文件名为“makefile”的文件，目标被定为“all”。

◇ makefile 中的宏被展开。使用“START_BIN”替换所有的“$(OUTDIR)”。

◇ 以“all”为目标的描述块将是如下的形态：

```
all: START_BIN START_BIN\start.exe
```

◇ 目标“all”有两个依赖项，第一个依赖项是“START_BIN”，寻找以 START_BIN 为目标的描述块。

```
START_BIN:
    if not exist "START_BIN/" mkdir START_BIN
```

◇ 目标“START BIN”描述块没有依赖项，直接运行命令。即，如果目录“START BIN/”不存在，则创建。

◇ 目标“all”的第一个依赖项生成，再寻找以第二个依赖项“START_BIN\start.exe”为目标的描述块。

```
    START_BIN\start.exe:START_BIN\start.obj
        link /INCREMENTAL:NO /NOLOGO -subsystem:windows,5.0 -out:START_BIN\start.exe
START_BIN\start.obj kernel33.lib  ws2_33.lib mswsock.lib advapi33.lib bufferoverflowu.lib
```

◇ 在以“START BIN\start.exe”为目标的描述块中，存在一个依赖项“START BIN\start.obj”。

寻找以“START BIN\start.obj”为目标的描述块。

```
    START_BIN\start.obj: start.c
        cl -c -DCRTAPI1=_cdecl -DCRTAPI2=_cdecl -nologo -GS -D_X86_=1  -DWIN32 -D_WIN32 -W3 -D_WINNT
-D_WIN32_WINNT=0x0500 -D_WIN32_IE=0x0500 -DWINVER=0x0500  -D_MT -MTd /Gz /Fo"START_BIN\\"
/Fd"START_BIN\\" start.c
```

◇ 在以“STARTH BLN\start.obj”为目标的描述块中，存在着一个依赖项“start.c”，makefile 中没有以 start.c 为目标的描述块，且 start.c 文件存在。判断依赖项 start.c 的时间戳是否晚于目标 START_ BIN\start.obj 的时间戳，如果 start.c 文件的时间戳晚于 START_BIN\start.obj 文件时间戳（说明 start.c 被修改之后没有编译过，现在的这个 START BIN\start.obj 是由修改之前的 start.c 编译得来的）则运行命令，命令是调用 cl.exe 生成 START_ BlN\start.obj。

如果 start.c 文件的时间戳早于 START- BIN\start.obj 文件时间戳，说明 START BIN\start.obj 是最新编译的，于是不运行命令。

◇ “START__ BIN\start.obj”目标完成以后，返回到以“START- BIN\start.exe”为目标的描述块中。判断依赖项“START-_BIN\start.obj”的时间戳与目标“START_BIN\start.exe”的时间戳。如果文件 START_BIN\start.obj 的时间戳晚于 START_BIN\start.exe，说明这个对象文件还没有链接过，START_ BIN\start.exe 不是最新的，于是调用描述块中的命令。

命令是使用 link.exe 生成 START_BIN\start.exe。如果文件 START—BIN\start.obj 的时间戳早于 START_BIN\start.exe，说明目前这个 START BIN\start.exe 是由现存的这个

START_ BIN\start.obj 编译生成的，是最新的，于是直接返回。

◇ 现在目标“all”的两个依赖项都已经完成。由于描述块中没有命令，所以这个描述块完成，于是目标“all”完成。

◇ nmake 退出。

读者也可以自行指定编译的目标，如果运行：

```
    nmake clean
```

将使用以 clean 为目标的描述块，即删除目录“START BIN”及目录内的所有文件。

如果运行：

```
    nmake START_BIN\start.obj
```

将只会调用以“START BIN\start.obj”为目标的描述块，依此类推。

3.3.6 makefile预处理

实例 3-2 中最开始的一行与 makefile 的预处理相关。

```
    !include <Win33.Mak>
```

与 c 语言中的头文件包含机制类似，makefile 文件也可以包含其他文件，上面这条语句就表示包含了一个文件“Win32.Mak”。使用“！”开始的行是 makefile 的预处理指令。

表 3.16　　makefile 预处理指令

指　令	意　义
!CMDSWITCHES (+| - }option...	当前目标的全名（全路径）
!ERROR string	显示错误“string”，然后停止执行，错误代码为 U1050
!MESSAGE string	显示字符串 string
!INCLUDE [<]filename[>]	包含 makefile。nmake 首先在指定或当前目录中搜索 filename，然后在任何父生成文件的目录中递归搜索，如果 filename 括在尖括号（< >）内，则在由 INCLUDE 环境变量（或宏）指定的目录中寻找
!IF const	如果成立（非零），则处理!IF 和下一个!ELSE 或!ENDIF 之间的语句
!IFDEF macroname	如果定义了后面的宏，则处理!IFDEF 和下一个!ELSE 或!ENDIF 之间的语句。如果宏是空（没有赋值）将被视为尚未定义
!IFNDEF macronam	与!IFDEF 相反
!ELSE	如果前面的!IF、!IFDE 或!IFNDEF 语句不成立，则处理!ELSE 和下一个!ENDIF 之间的语句
!ELSEIF !ELSEIFDEF !ELSEIFNDEF	分别相当于!ELSE IF、!ELSE IFDEF 和!ELSE IFNDEF
!ENDIF	!IF、!IFDEF 或!IFNDEF 块的结尾
!UNDEF macroname	取消宏定义

makefile 中可以包括如表 3.16 所示的预处理指令。

预处理指令中初始感叹号(!)必须出现在行首，后面可以使用制表符用于缩进。指令不区分大小写。

实例 3-5　nmake 预处理指令

本实例演示在一个 makefile 使用不同的预处理指令来获得不同的输出，以下文件内容来自于 Platform SDK，Win32.Mak 中的部分内容如下：

```
!IF "$(APPVER)" == "5.0"
OUTDIR=WIN2000
!ELSEIF "$(APPVER)" == "5.01"
OUTDIR=XP32
!ELSEIF "$(APPVER)" == "5.02"
OUTDIR=SRV2003
!ELSEIF "$(APPVER)" == "4.0"
OUTDIR=NT4
!ENDIF

.....
```

```
!IF "$(SDKPRERELEASE)" == "1"
OUTDIR=PRE_$(OUTDIR)
!ENDIF

#Set DEBUG
!IF "$(NODEBUG)" == ""
OUTDIR=$(OUTDIR)_DEBUG
!ELSE
OUTDIR=$(OUTDIR)_RETAIL
!ENDIF
```

读者如果使用过 SDK 中不同的编译环境配置的批处理文件，应该能够观察到用这些不同的环境所生成的文件的输出路径是不同的，比如有 WIN2000_XP、XP32 RETAIL 等。

从上面的 makefie 分析可知：SDK 的编译环境所使用的设置不同，程序可执行文件的输出目录也会不同。

3.3.7　在Platform SDK的基础上使用nmake

Platform SDK 提供了一个实用的 makefile 头文件 Win32.mak。Win33.mak 中除了版本、

输入目录等宏外，还将一些经常在一起使用的命令及参数和选项定义为了宏，表 3.17 给出了这些宏及其值。

表 3.17　　Win33.mak 中定义的常用宏

宏	值
$(CC)	cl
$(CFLAGS)	-c -W3 -D_X86=1 -DWIN32 -D_WIN32
$(CDEBUG)	-Zi -Od
$(CVARS)	-DNULL=0 -ML
$(CVARSMT)	-DNULL=0 -D_MT -MT
$(CVARSDLL)	-DNULL=0 -D_MT -D_DLL -MDd
$(RC)	rc
$(RCFLAGS)	-r
$(RCVARS)	-DWIN32 -D_WIN32 -DWINVER=0x0500
$(LINK)	link
$(LDEBUG)	-debug:full -debugtype:cv
$(CONLFLAGS)	/INCREMENTAL:NO /NOLOGO -subsystem:console,5.0
$(GUIFLAGS)	/INCREMENTAL:NO /NOLOGO -subsystem:windows,5.0
$(CONLIBS)	Kernel33.lib Advapi33.lib
$(GUILIBS)	Kernel33.lib Advapi33.lib User33.lib GDI33.lib ComDlg33.lib WinSpool.lib
$(CONLIBSMT)	Libcmt.lib OldNames.lib Kernel33.lib Advapi33.lib
$(GUILIBSMT)	Libcmt.lib OldNames.lib Kernel33.lib Advapi33.lib User33.lib GDI33.lib ComDlg33.lib WinSpool.lib
$(IMPLIB)	lib
$(DLLENTRY)	@12（x86）
$(DLLLFLAGS)	/INCREMENTAL:NO /NOLOGO -entry:_DllMainCRTStartup$(DLLENTRY) -dll
$(CONLIBS)	Kernel33.lib Advapi33.lib
$(GUILIBS)	Kernel33.lib Advapi33.lib User33.lib GDI33.lib ComDlg33.lib WinSpool.lib
$(CONLIBSMT)	Libcmt.lib OldNames.lib Kernel33.lib Advapi33.lib
$(GUILIBSMT)	Libcmt.lib OldNames.lib Kernel33.lib Advapi33.lib User33.lib GDI33.lib ComDlg33.lib WinSpool.lib

本书配套光盘中很多 makefile 使用到了这些宏，如实例 3-4 中的 makefile 所示。另外在 PlatformSDK 的示例中，也大量使用了这些宏。

实例 3-6 使用 Win32.mak 文件中定义的宏

```
!include <Win33.Mak>
src = start

all: $(OUTDIR) $(OUTDIR)\$(src).exe

$(OUTDIR) :
    if not exist "$(OUTDIR)/$(NULL)" mkdir $(OUTDIR)

$(OUTDIR)\$(src).obj: $(src).c
    $(cc) $(cflags) $(cvars)  /Fo"$(OUTDIR)\\" /Fd"$(OUTDIR)\\" $(src).c

$(OUTDIR)\$(src).exe: $(OUTDIR)\$(src).obj
    $(link) $(conflags) -out:$(OUTDIR)\$(src).exe $(OUTDIR)\$(src).obj $(conlibs)

clean:
        $(CLEANUP)
```

3.4 使用WinDbg调试

WinDbg 是一个具有强大功能的带图形化界面的源码级调试工具，它是由微软开发，免费发布。使用 WinDbg 可以调试 Windows 系统的应用程序，也可以用于调试服务和内核态驱动程序。在调试程序时，WinDbg 具有很多方便的功能，使程序的调试变得简单。

Visual Studio IDE 也提供调试功能，但远没有 WinDbg 强大。WinDbg 拥有比 Visual Studio 丰富得多的功能可供选择，比如断点的种类、可查看到的信息的种类等。在易操作性上 WinDbg 也具有一定的优势；尤其在一些特殊的情况下，WinDbg 的易用性非常显著，比如调试注入到其他进程中的模块、回调函数，调试服务程序等。

但由于其功能的复杂性，要想非常熟练地使用 WinDbg 并不容易。本章将会对 WinDbg 的使用方法作基本的介绍。

3.4.1 安装WinDbg

在 Platform SDK 安装目录的 setup 文件夹下有 WinDbg 的安装文件，（在开始菜单中有快捷方式），如果读者想使用最新版本的 WinDbg，可以在微软的网站上下载，地址是：www.mlcrosoft.com/whdc/devto ols/debugging/default.mspx。同时还需要下载与系统版本对应的符号文件（Symbols）。

符号文件中存储了用于调试的信息，比如编译完成的机器代码（汇编）与源代码之间的关系，如果没有符号文件，调试时只能看到调试对象可执行文件中反汇编的汇编代码。编译时，可以通过编译选项指定生成符号文件，Windows 系统本身也有符号文件，包括了 API 的定义，结构的定义等内容，对调试有重要的帮助。Windows 平台的符号文件(. pdb)文件格式是对外公开的。

WinDbg 的界面如图 3-6 所示。

按照提示将 WinDbg 和符号文件安装好，然后需要进行一些简单的设置。首先需要将安装好的符号文件加入 WinDbg 的符号文件路径中。在菜单“File\Symbol file Path”选项中，如图 3-7 所示。

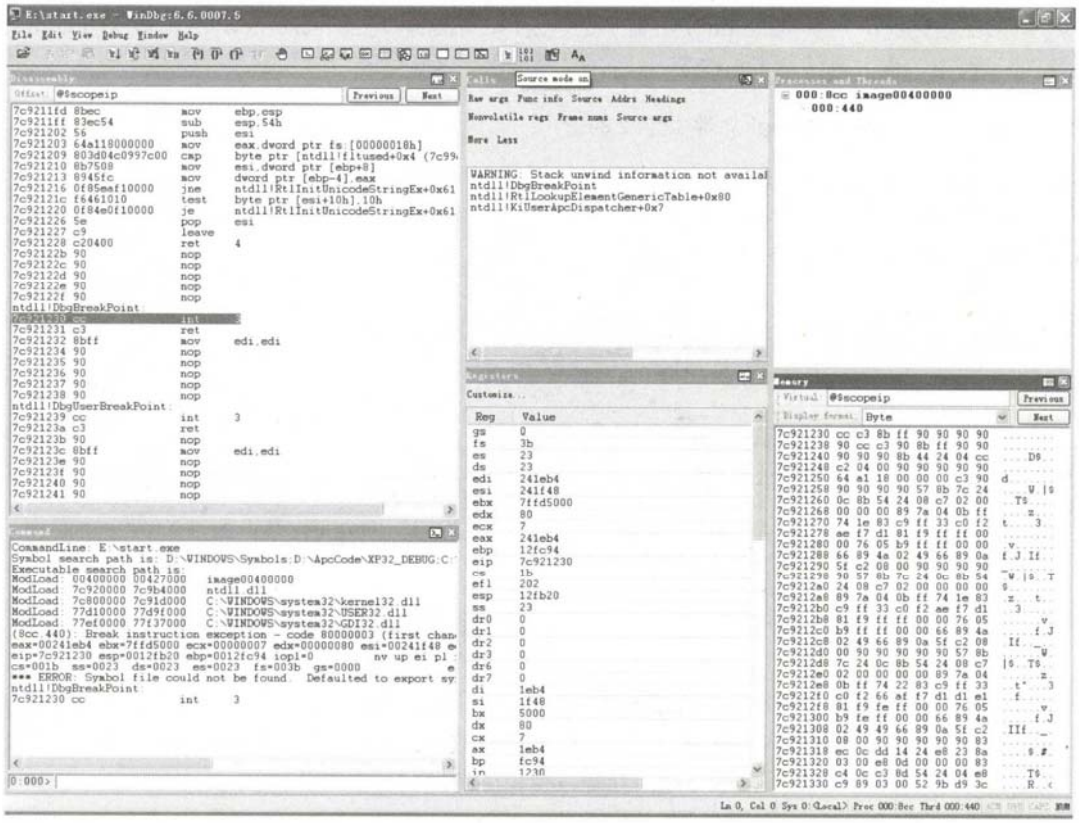

图 3-6　WinDbg 界面

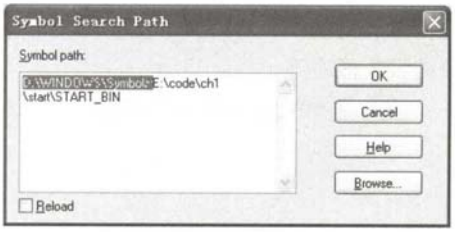

图 3-7　加载符号文件

3.4.2　编译可调试的程序

使用 WinDbg 调试应用程序，需要在编译时加入生成调试信息，并生成调试版本的可执行文件。

实例 3-7 编译可调试程序的 makefile

本实例展示了一个可以使用 WinDbg 进行调试的 makefile 的内容。

```
#----------------- 编译供调试的程序 ------------------
!include <Win33.Mak>
OUTDIR = START_DEBUG

#--------------------- 程序构建 ----------------------
all: $(OUTDIR) $(OUTDIR)\start.exe

$(OUTDIR) :
```

```
    if not exist "$(OUTDIR)/$(NULL)" mkdir $(OUTDIR)

$(OUTDIR)\start.obj: start.c
    $(cc) $(cflags) /Gz /MT /Fo"$(OUTDIR)\\" /Fd"$(OUTDIR)\\vc80.pdb" /ZI /Od start.c

$(OUTDIR)\start.exe: $(OUTDIR)\start.obj
    $(link) /NOLOGO /subsystem:windows,5.0 /DEBUG /out:$(OUTDIR)\start.exe
/PDB:"$(OUTDIR)\start.pdb" $(OUTDIR)\start.obj $(guilibs)

#---------------------- 程序清理 ----------------------
clean:
    $(CLEANUP)
```

在本实例 makefile 中，通过编译和链接选项，指定了编译和链接时，需生成调试信息。可以用其编译本书实例 1-1 的源代码，start.c。在“Visual Studio 命令提示”中使用如下命令编译：

```
nmake /f debug.mak
```

会出现“START_DEBUG”目录，目录中会有可执行文件、对象文件、程序数据库、增加连接文件等文件。

在调试前，需先了解 WinDbg 的命令。

3.4.3 WinDbg命令

由于 WinDbg 命令条目众多、功能丰富，因此使用方法比较复杂。这里只介绍最简单的几个命令，使读者有初步的了解。在实际使用时可以查阅 WinDbg 文档。

. d：查看内存数据。
. l：列出进程信息。
. Im:查看已经加载的模块。
. g：运行。
. p：单步运行。
. ld:加载符号文件。
. r：查看寄存器。
. bp、bu、bm：下载断点，bp 为执行断点，bu 为未解析断点，bm 为符号断点。
.bl：列出断点。
. bc：清除断点。

3.4.4 调试过程演示

下面我们使用在3.4.2小节编译好可源码调试的可执行程序start.exe来演示如何使用 WinDbg 调试程序。

◇按照 3.4.1 小节的说明安装好 WinDbg，安装好 Windows 符号文件，并将符号文件路径加入到 WinDbg 的符号文件路径中。

◇运行 WinDbg。

◇加载 start.exe 的符号文件：菜单“file\symbol file path”，在对话框中选择“browse”，找到编译生成的“START DEBUG”目录，单击“确定”按钮，加入符号文件。

◇将 WinDbg 设置为源代码模式：在菜单“Debg”中将“Source Mode”勾选上。

◇打开供调试的可执行文件 start.exe：选择菜单“file”—“executable file path”，在对话框中找到之前生成的 start.exe，单击“打开”按钮，如图 3-8 所示。

◇WinDbg 会在程序的某个位置自动中断。在“Command”窗口下面的命令输入栏中输入命令：

```
bp WinMain
```

图 3-8　选择调试对象

◇上面的命令意思是在 WinMain 函数处设置断点，程序执行到 WinMain 函数时就会中断，然后运行：

```
g
```

◇程序重新开始运行，然后会中断在 WinMain 函数处，如果编译时生成的程序数据库正确，WinDbg 会自动弹出源代码窗口，如图 3-9 所示。

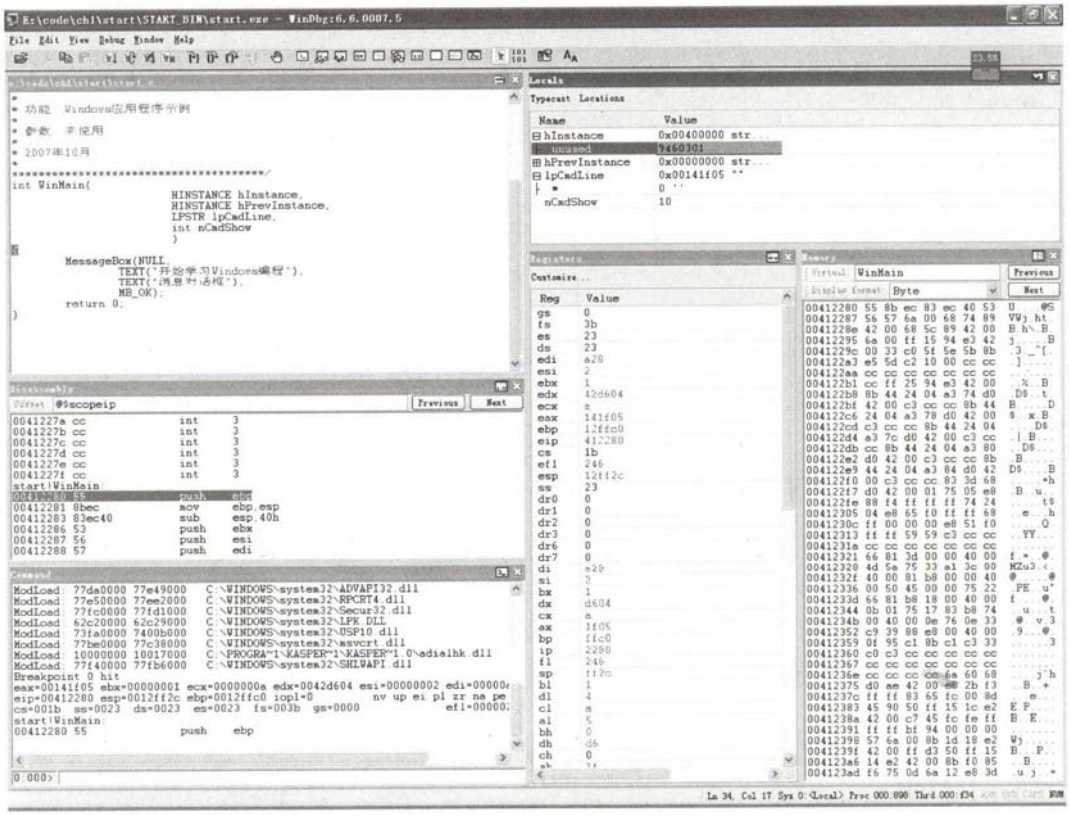

图 3-9　WinDbg 调试界面

现在可以尝试运行几个命令，在命令窗口中输入：

```
lm
```

会在命令窗口中显示出当前已经加载的模块，如图 3-10 所示。

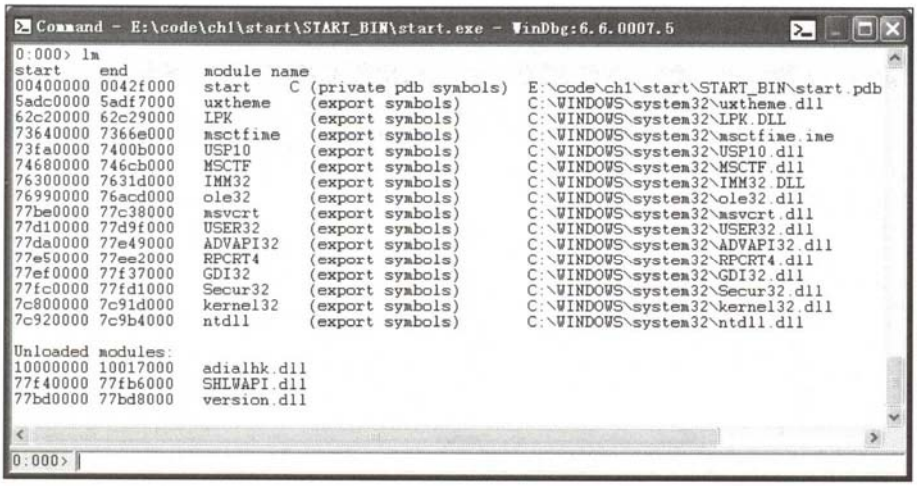

图 3-10　lm 命令运行结果

运行：

```
d MessageBoxA
```

结果如图 3-11 所示，显示结果是 MessageBoxA 函数的机器码，使用 d 命令可以通过指针查看内存，也可以在“Memory”窗口查看。

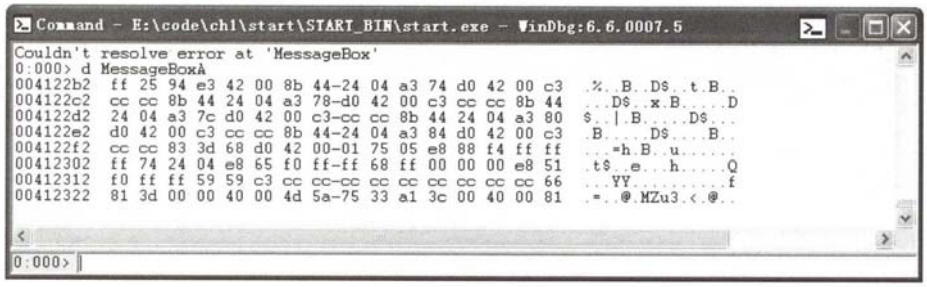

图 3-11　查看内存命令运行结果

运行：

```
u WinMain
```

在命令窗口中会显示出 WinMain 函数的反汇编程序，如图 3-12 所示。同样的功能也可以通过"Disassembly”窗口查看。

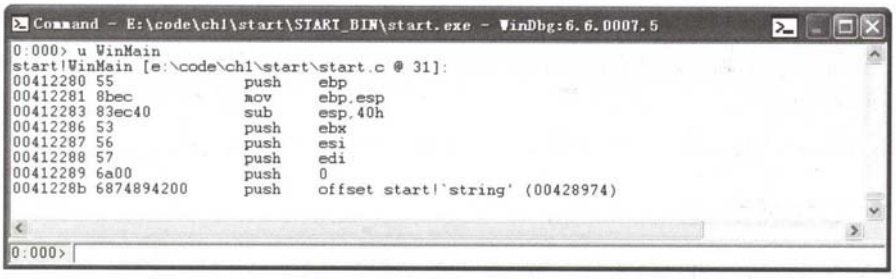

图 3-12　反汇编命令运行结果

在调试时还有一些快捷方式可供使用，F10 是单步执行，F11、F8 是单步执行。

上面所说的这些窗口，可以通过菜单“View”打开。常用的窗口包括：命令窗口(Command)局部变量(Locals)、寄存器(Register)、内存(Memory)、反汇编(Disassembly)、监视(Watch)、进程和线程(Process and thread)等，如图 3-13 所示。

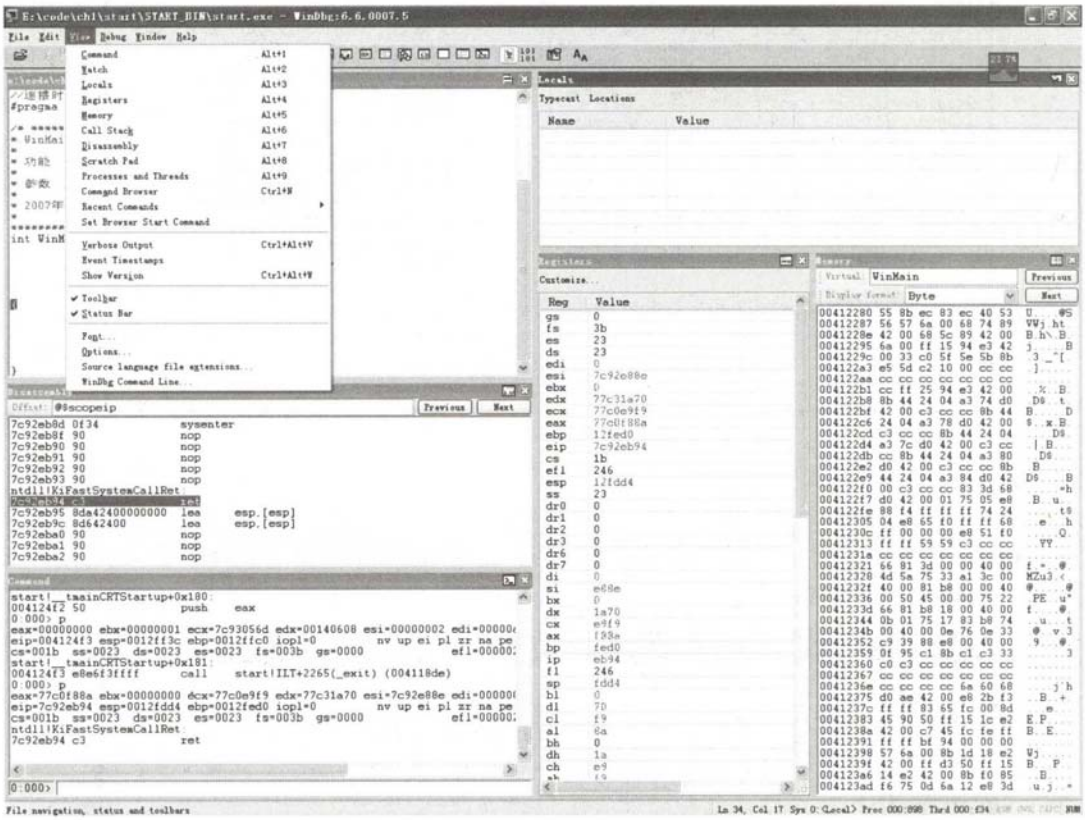

图 3-13　WinDbg 调试窗口

3.5　集成开发环境Visual Studio

本章 3.1～3.4 节讲的工具大多是命令行的字符界面工具，或只具有简单界面的工具。VisualStudio IDE 将各种工具集合在一起，并提供丰富的图形化界面和人性化的操作方式，为程序开发提供了很多方便的功能。

在 3.1～3.4 节的基础上，读者应该已经对程序开发的工具有了比较深入的了解，解决了深入学习 IDE 的基础问题。

在使用 IDE 前请按照第 1 章的方法安装、配置好 Visual Studio。

3.5.1　工程类型选择与配置

Visual Studio 在安装时已经为用户预装了很多的工程类型模块。专业版的工程类型模板包括 ATL 和各种程序，MFC 和各种程序，CLR 和各种程序，Win32 应用程序等，大约有二十多种，但是少数几种适用于 Windows 应用程序开发。速成版中只包含少量的几种工程类型模板，但是已经足够 Windows 网络应用开发使用。

笔者建议读者在了解了编译链接工具选项设置的基础上选择“空项目”作为工程类型的首选，空项目中的编译和链接选项都保持着最初的默认状态，是最适合基于 Windows API 开发的类型。如图 3-14 所示为选择项目类型的界面。

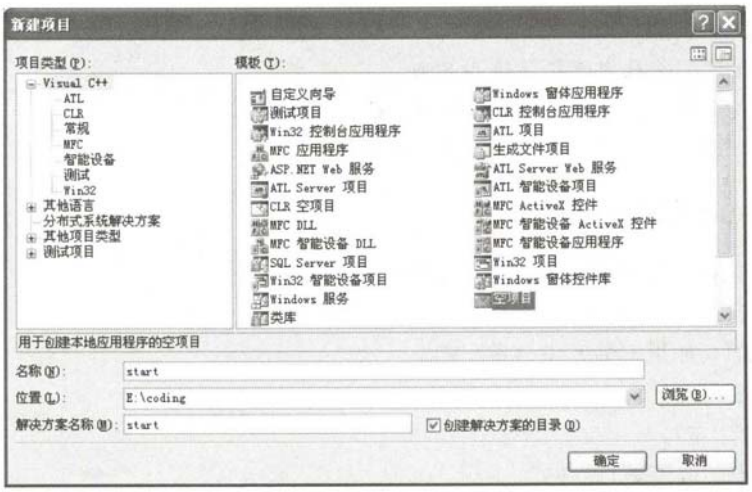

图 3-14　专业版中选项工程类型对话框

在 1.2.4 小节已经介绍了如何在 Visual Studio 中新建项目，这里不再赘述。

读者可以将 Visual Studio 中的工程，与在本章 3.1～3.4 节的内容联系起来。在“解决方案资源管理器”中右键单击工程名，在菜单中选择“属性”，在“配置属性”条目下有“常规”、“C/C++”、“连接器”等条目，每个条目中都有很多子条目，仔细观察这些选项，会发现它们大多是我们在 3.1.1 节和 3.1.2 节中所介绍的编译器和链接器选项的配置，如图 3-15 所示。比如“链接器”条目下的“输入”→“附加依赖项”就对应编译器参数中的“输入文件”。

图 3-15　工程属性配置

如果进行基于 Windows API 的开发需要进行以下几点配置。

◇在“配置类型”中选择“应用程序”或“动态链接库”，注意链接生成文件类型需要在链接器→常规→输出文件中选择。

◇在“MFC 的使用”中选择“使用标准 Windows 库”。

◇在“ATL 的使用”中选择“不使用”，这两项表明要进行基于 Windows API 的开发。

◇在“字符集”中选择字符编码方式，参考第 2 章的内容。

◇在“C/C++”条目中选择编译选项。

◇在“链接器”条目中选择链接器选项。

◇在“生成事件”条件中选择 nmake 选项。

◇可以为“调试”和“发布”选择不同的配置。

3.5.2 Visual Studio快捷方式

下面介绍一些在代码编辑、调试时常用到的功能。

◇显示代码提示：Alt+ →。

◇在 MSDN（如果安装）中搜索：Fl。

◇开始调试：F5。

◇构建工程：F7。

◇设置断点、取消断点：F9。

◇单步跟踪：Fl0。

◇单步步入：Fll。

◇转到定义：F12。

◇ 新建文件：Ctrl+N。

◇ 查找：Crtl+F。

◇ 转到行：Ctrl+G。

◇ 选中单词：Ctrl+W。

◇ 前后交换：Ctrl+T。

◇ 转为小写：Ctrl+U。

◇ 转为大写：Ctrl+Shilt+U。

◇ 注释所在行：Ctrl+K，然后 Ctrl+C。

◇整理代码格式：Ctrl+K，然后 Ctrl+F（或在菜单中选择“编辑”→“高级”→“设置选定内容格式”）。

其他快捷方式读者可以对照 Visual Studio 菜单中的提示自行使用。读者也可以自定义快捷方式。在“工具”→“选项”→“环境”→“键盘”中进行设置。

3.5.3 生成项目

完成一个项目的代码写作后，用户可以将项目生成为一个可执行文件或者动态链接库，详细操作步骤如下。

◇生成解决方案：选择“生成”→“生成解决方案”，或在“解决方案管理器”中右键单击解决方案名称，在菜单中选择。快捷键为 F7。

◇如果生个单个项目菜单，选择“生成”→“生成项目名称”，或在“解决方案管理器”中右键单击项目名称，在菜单中选择。

◇如果生成有错，可以查看错误列表，在菜单中选择“视图”→“其他窗口”→“错误列表”。

◇单击错误列表中的项，IDE 会在代码编辑窗口中定位错误。

3.5.4 调试

前面提到，在代码写作过程中，用户也许会遇到各种各样的错误。这时 IDE 也提供相应的功能来定位、解决这些错误，操作步骤如下。

◇在项目属性中为“Debug”模式设置调试编译、链接选项。

◇将项目配置为“Debug”模式。

◇生成项目。

◇设置断点：在代码编辑器的左侧点击，会出现一个红点，就是断点。

◇开始调试：选择“调试”→“启动调试”（快捷键为 F5）。

◇如果程序运行到断点处会中断下来。

◇单步步过：F10；单步步入：F11；运行至下交断点命中：F5。

◇删除断点：单击表示断点的红点，或在菜单中操作。

◇结束调试：Shift+F5。

3.5.5 选项与设置

“工具”菜单中列了很多实用的工具，但是速成版的 Visual Studio 工具较少，一些很有用的工具没有，用户可以自行添加。

◇在菜单中选择“工具”→“外部工具”，如图 3-16 所示。

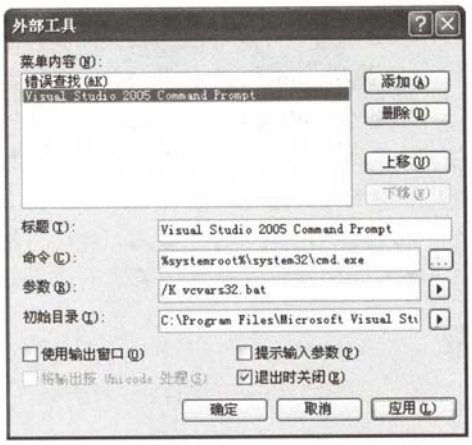

图 3-16 添加外部工具

◇在外部工具中对话框中单击“添加”按钮。

◇在“标题”中输入名字（在“工具”菜单显示的内容）。在“命令”中输入运行的程序路径，在“参数”中输入启动程序运行的参数。

◇查看“工具”菜单，发现多出一个选项，点击选项会有之前指定的程序运行。“工具”菜单还可以配置 IDE，如图 3-17 所示，一些常用的设置包括如下。

◇“环境”→“键盘”：可以设置和修改快捷键，上面所介绍的快捷键用户可以自己修改，也可以增加新的快速键。

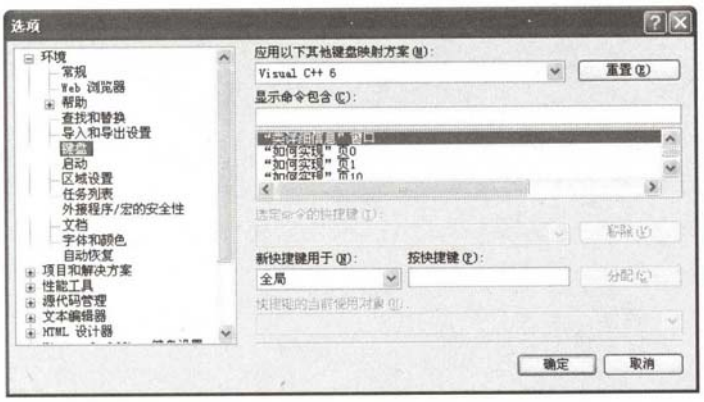

图 3-17 “工具”→“选项”设置 IDE

◇“环境”→“字体和颜色”：可以设置各种窗口中的字体、颜色、大小等。

◇“文本编辑器”→“C/C++”：选项中可以对文件编辑器进行设置，如显示行号、代码自动提示等。

注意

很多读者可能会遇到这样的问题，在 Visual StudiO 2005 专业版中建立了“Win32 控制台应用程序”或“Win32 项目”，但是使用默认设置编译完成的“发布”版本在没有安装 Visual StudiO 2005 的操作系统上运行对却会报错。下面解答这个问题：查看工程属性，在“C/C++”子选项下的“代码生成”选项页中，“运行时库”选项下，/MD(Release)选项或/MDd(Debug)选项。使用这两个选项编程，将会调用一些系统中原本并不存在的接口。要解决这个问题需将选项改为/MT。如果读者新建 C / C++项目时选择“常规”→“空项目”，在运行时库时使用的选项就是/MT（这也是 cl.exe 的默认选项）。

建议：在编写 Windows 本机(Native)应用程序时，建议建立工程时选择“空项目”，这样编译连接选项大多都是默认的。

3.6 开发环境配置总结

本章介绍了开发环境的相关原理，应该会对读者排除一些疑难的问题会有帮助。

Visual Studio 和 Platform SDK 在配置完成后，使用就比较简单，图形用户界面非常强大。

如果使用 Visual Studio 专业版以上的版本直接安装就可以使用了。如果没有购买专业版以上的开发工具，那么从微软的网站上去下载 Visual C++速成版和 Platform SDK，安装本章的说明安装并配置变量，也能完成开发 Windows 应用程序的工作。

本书综合应用了各个版本，使用方法基本相同。读者有可以自由选择开发工具版本，各版本间不会有大的区别，触类旁通，原理类似。

第 4 章　文件系统

文件系统是操作系统的重要组成部分。支持操作系统运行的系统可执行文件本身也位于文件系统中。几乎所有的应用程序都要操作文件，文件系统应用程序开发接口是系统提供的最重要、最基本的 API 集合之一，也是开发 Windows 应用程序最常用到的接口。

4.1　概述

从 Windows 诞生到现在，其文件系统已经经历几次重大的升级。从最开始的 FAT16 到 FAT32 文件系统到目前在 Windows 2000 和 Windows Xl:，以及 Windows Vista 均采用的 NTFS 文件系统。

这些文件系统的实现细节本身并未公开，也不在本书讨论的范围内。Windows 操作系统的文件系统虽然具有 FAT16、FAT32、NTFS 等多种形式，但是对应用程序来说，微软公司提供的开发接口却有非常好的兼容性。在应用程序中使用 API 操作文件系统中文件的方法，在各种文件系统中是基本相同的。

要熟练使用文件系统 API 必须先了解 Windows 文件操作中的一些基本概念。这些概念包括“驱动器”、“卷”、“目录”、“文件对象”、“文件流”、“文件句柄”、“文件指针”、“文件映射”等。

4.1.1　文件系统的基本概念

在进行文件系统相关开发时，所涉及的基本概念包括磁盘分区、卷、目录、文件对象、文件句柄、文件映射等。文件系统的概念一般比较好理解，下面讲解两个容易混淆的概念。

1．磁盘分区(Partitions)

磁盘是装到计算机上的存储设备，比如常见的硬盘。磁盘分区是为了便于管理和使用物理硬盘，而在一个物理硬盘上划分可以各自独立工作的一些逻辑磁盘。比如一块 80GB 的硬盘可以划分为 4 个 20GB 的分区来使用，对操作系统来说这 4 个 20GB 的分区是 4 块独立的逻辑磁盘。

2．卷(Volumes)

卷，也称为逻辑驱动器，是 NTFS、FAT32 等文件系统组织结构的最高层。卷是存储设备（如硬盘）上由文件系统管理的一块区域，是在逻辑上相互隔离的存储单元。一个磁盘分区至少含有一个卷，卷也可以存在于多个磁盘分区上，仅存在于一个磁盘分区上的卷称为“简单卷”，仅存在于多个磁盘分区上的卷称为“多分区卷”或“跨区卷”。在最常见的情况下，一个分区只包含一个卷，一个卷也只存在于一个分区上，所以两者容易混淆。卷存在卷标，程序可以通过卷标访问卷。

3．Windows 系统中文件命名的标准

Windows 系统中，文件命名格式为“主文件名+扩展名”，中间以“．”隔开。扩展名则说明该文件的类型，主文件名和扩展名组成的文件名惟一地标识一个文件。文件可以没有扩展名，但是不能没有主文件名。

默认情况下 Windows XP 系统是隐藏已知类型扩展名的，如果要显示可以去掉“控制面板”一“文件夹选项”一“查看”一“隐藏已知文件类型的扩展名”选项前的勾选，如图 4-1 所示。

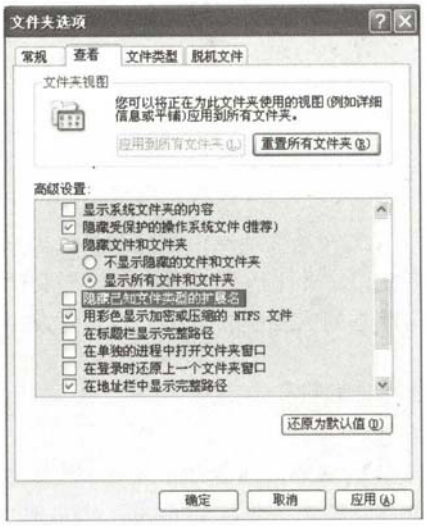

图 4-1　显示文件扩展名

用户在给文件命名时必须要遵守特定的规则，并且尽量避免自己设置扩展名，以防操作系统无法识别文件。

(1)长度规则。

在 Windows 系统中，不对单个文件的长度作限制，但是文件路径的长度被限制 260 个字符，这 260 个字符包括卷标、路径、主文件名和扩展名以及分隔符，因此，文件名的长度为 260 减去文件所在目标的路径长度。值得注意的是在 Windows NT 系统中，字符使用的是 Unicode 宽字符。英文和中文都占用两个字节。

(2)特殊字符。

特殊字符则不能在文件名的任何位置出现。特殊字符一共有 9 个,分别为:“ / ”、“ | ”“ \ ”、“*”、“?”、″<″、“>”、“” ”、“:”。

特殊字符在系统或者命令行下代表一些特殊意义。

(3)保留设备名。

还有一些在 DOS 下被保留的设备名不能作为文件名或者主文件名。它们是：CON、PRN、AUX、NUL、COM1、COM2、COM3、COM4、COM5、COM6、COM7、COM8、COM9. LPT1、LPT2、LPT3、LPT4、LPT5、LPT6、LPT7、LPT8 以及 LPT9。

4.1.2　文件系统主要API

本章共讲解 3 类 API 共 40 个，主要涉及磁盘和驱动器管理、文件和目录管理以及文件系统高级操作，如表 4.1、4.2、4.3 所示。

表 4.1　　磁盘和驱动器管理类 API

API	功 能 简 介
GetLogicalDrives	获取主机中所有的逻辑驱动器，以 Bit Map 的形式返回
GetLogicalDriveStrings	获取主机中所有的逻辑驱动器，以驱动器根路径字符串返回
FindFirstVolume	查找主机中的第一个驱动器，返回查找句柄

续表

API	功 能 简 介
FindNextVolume	根据 FindFirstVolume 返回的句柄，查找主机中后继的逻辑驱动器
FindVolumeClose	关闭驱动器查找句柄
GetDriveType	获取驱动器类型
GetVolumeInformation	获取逻辑驱动器信息
FindFirstVolumeMountPoint	查找指定卷的第一个挂载点，返回查找句柄
FindNextVolumeMountPoint	根据 FindFirstVolumeMountPoint 返回的句柄，查找卷的后继挂载点
FindVolumeMountPointClose	关闭挂载点查找句柄
GetVolumeNameForVolumeMountPoint	根据指定的挂载点获取相应的卷设备名
SetVolumeMountPoint	将指定卷挂载到指定挂载点处
GetDiskFreeSpace	获取磁盘空间信息，包括每簇的扇区数、每扇区的字节数、簇数量、空闲的簇数量
GetDiskFreeSpaceEx	获取用户可用的空闲空间的字节数、磁盘总容量的字节数

表 4.2　　文件和目录管理

API	功 能 简 介
DeleteFile	删除参数所指定文件
CopyFile	复制指定文件为一个新文件
MoveFile	将指定文件或目录移动到指定位置
CreateFile	新建或打开一个文件，获取文件句柄
ReadFile	读取由文件句柄指定的文件的内容
WriteFile	向由文件句柄指定的文件中写入内容
GetFileSize	获取文件大小，返回 DWORD 中；大小超出 DWORD 最大值时可指定高 32 位 DWORD 联合储存
GetFileSizeEx	获取文件大小，储存到一个 64 位的大整数联合体中
CreateDirectory	创建一个目录
GetCurrentDirectory	获取当前程序所在目录
SetCurrentDirectory	设置当前程序所在目录
GetModuleFileName	获取当前模块全路径
FindFirstFile	查找指定目录下第一个文件或目录，获得查找句柄
FindNextFile	根据 FindFirstFile 获得的句柄，循环查找文件和目录
GetFileAttributes	获取指定文件或目录的属性，返回一个 DWORD 值
GetFileAttributesEx	获取指定文件或目录属性，储存在 WIN32_FILE_ATTRIBUTE_DATA 结构体中
SetFileAttributes	将文件属性设定为指定值
FileTimeToLocalFileTime	将文件时间转换为本地时间
FileTimeToSystemTime	将文件时间转换为系统时间，SYSTEMTIME 格式，便于显示

表 4.3　　高级文件系统操作

API	功 能 简 介
CreateFileMapping	创建文件的映射对象
MapViewOfFile	创建视图，将创建的文件映射对象映射到当前进程的地址空间中
FlushViewOfFile	将视图中数据都写入磁盘，对视图的操作都会反映到磁盘上的文件中

续表

API	功 能 简 介
OpenFileMapping	打开已经存在的命名的文件映射对象
UnmapViewOfFile	取消文件映射
GetMappedFileName	从映射对象获取被映射文件的文件设备名
QueryDosDevice	获取 MS-DOS 设备名

4.2 磁盘和驱动器管理

Windows API 是提供给用户态程序使用的。一些对磁盘的高级操作，如磁盘分区、格式化驱动器、改变驱动器的文件系统、读取磁盘扇区等操作需要在系统内核中完成，用户态的API 不能完成这样的高级操作。

本节通过以下 4 个实例来讲解磁盘和驱动器管理类的 Windows API。

◇使用两种方法来遍历驱动器并获取驱动器属性。

◇使用 API 操作驱动器挂载点。

◇判断光驱中是否有光盘。

◇获取磁盘剩余空间、扇区信息等。

4.2.1 遍历卷并获取属性

获取一个主机上的所有驱动器列表有两种方法，一种是使用 GetLogicalDrives 或 GetLogicalDriveStrings，另一种是使用 FindFirstVolume 和 FindNextVolume 组合。

第一种方法获取主机上的逻辑驱动器，也就是所有分配的卷标的驱动器，返回的结果是驱动器的根路径。第二种方法返回的是“\\?\volume{GUID}”形式的驱动器设备名。

使用 GetDriveType API 可以获取驱动器类型，使用 GetVolumeInformation 可以获取驱动器属性。

1．关键 API

(1) GetLogicalDrives。

获取主机中所有的逻辑驱动器，以 BitMap 的形式返回，其函数原型如下：

◇返回值

GetLogicalDrive 函数返回一个 DWORD 类型的值，第一位表示所对应的驱动器是否存在。一般情况下 DWORD 的数据长度是 32 位，在这个 DWORD 中，每一位对应了一个逻辑驱动器是否存在。第二位如果是“1”则表示驱动器“B：”存在，第 4 位如果是“1”则表示驱动器“D：”是存在的，以此类推。

(2) GetLogicalDriverStrings。

获取主机中所有驱动器，以驱动器根路径字符串返回，其函数原型如下：

◇参数

nBufferLength：参数 lpBuffer 所指向的内存空间的大小，以字节为单位。

```
DWORD GetLogicalDrives(void);
```

```
DWORD GetLogicalDriveStrings(
  DWORD nBufferLength,
  LPTSTR lpBuffer
);
```

lpBuffer:指向存储返回结果字符串的内存空间。

◇返回值

函数的返回值指明了函数调用是否成功，如果成功则返回缓冲区中返回结果的总长度。如果返回值大于 nBufferLength，说明给定的缓冲区大小不够，返回值是实际需要的大小。如果返回 0，则说明函数运行出错。

◇使用说明

这个 API 实现了与 GetLogicalDrives 同样的功能，却以一种更直观的方式返回执行结果。函数执行结果放在 lpBuffer 所指向的内存区域中，此内存区域大小由 nBufferLength

参数指定，使函数返回结果不至于溢出。在调用此函数前需保证内存分配。

函数调用成功后，将在缓冲区中依次填入本机所具有的驱动器根路径字符串，如在笔者系统中有 5 个逻辑驱动器“C:\”、“D:\”“E:\”、“F:\”、“I：\”。执行后在缓冲区中的结果如下：

```
    0x43 0x3a 0x5c 0x00 0x44 0x3a 0x5c 0x00 0x45 0x3a 0x5c 0x00 0x46 0x3a 0x5c 0x00 0x49 0x3a
0x5c 0x00 0x00
```

也就是连续放置了“C:\”、“D:\”“E:\”、“F:\”、“I：\”这 5 个字符串（会在每个字符串后加一个‘\o’结束符，在所有卷标字符串的最后再加一个结束符）。

（3）FindFirstVolume。

查找主机中的第一个驱动器，返回驱动器设备名，其函数原型如下：

```
HANDLE FindFirstVolume(
  LPTSTR lpszVolumeName,
  DWORD cchBufferLength
);
```

◇参数

lpszVolumeName:指向驱动器名的内存缓冲区。

cchBufferLength:参数 lpszVolumeName 所指向的缓冲区大小，以字节为单位。

◇返回值

驱动器查找句柄,F 猫 NextVolume 和 FindVolumeColse 的参数,如果执行失败,返回 NULL。

（4）FindNextVolume

查找主机中后继的逻辑驱动器，其函数原型如下：

```
BOOL FindNextVolume(
  HANDLE hFindVolume,
  LPTSTR lpszVolumeName,
  DWORD cchBufferLength
);
```

◇参数

hFindVolume: FindFirstVolume 所返回的驱动器查找句柄。

lpszVolumeName:指向保存驱动器名的内存缓冲区。

cchBufferLength:参数 lpszVolumeName 所指向的缓冲区大小，以字节为单位。

◇返回值

返回 BOOL 表示是否成功，如果失败说明已经查找完成所有逻辑驱动器。

(5) FindVolumeClose。

\关闭 FindFirstVolume 打开的卷遍历句柄，其函数原型如下：

```
BOOL WINAPI FindVolumeClose(
  HANDLE hFindVolume
);
```

◇参数

hFindVolume:要关闭的驱动器查找句柄。

◇返回值

返回 BOOL 值表示是否成功关闭句柄。

(6) GetDriveType。

获取驱动器类型，其函数原型如下：

```
UINT GetDriveType(
  LPCTSTR lpRootPathName
);
```

◇参数

lpRootPathIName:驱动器根路径，如“C:\”。

◇返回值　　/

驱动器的类型，如 DRIVE’ FIXED 表示硬盘，DRIVE_CDROM 表示光盘等。详见实例 4-2 的 GetDirverInfo 函数。

(7) GetVolumeInformation。

获取逻辑驱动器信息，其函数原型如下：

```
BOOL GetVolumeInformation(
  LPCTSTR lpRootPathName,
  LPTSTR lpVolumeNameBuffer,
  DWORD nVolumeNameSize,
  LPDWORD lpVolumeSerialNumber,
  LPDWORD lpMaximumComponentLength,
  LPDWORD lpFileSystemFlags,
  LPTSTR lpFileSystemNameBuffer,
  DWORD nFileSystemNameSize
);
```

◇参数

lpRootPathName：输入参数，指向所要获取属性的驱动器的根路径字符串。

lpVolumeNameBuffer:输出参数，返回驱动器名。

nVolumeNameSize：输入参数，lpVolumeNameBuffer 的内存缓冲区大小。

lpVolumeSerialNumber:输出参数，存储驱动器序列号。

lpMaximumComponentLength:输出参数,返回文件系统所支持的文件组成部分的最大值。

lpFileSystemFlags:输出参数,属性可以用来判断多种驱动器属性值,如 FILE_VOLUME_QUOTAS 表示支持磁盘配额,FILE_SUPPORTS_ENCRYPTION 表示文件系统是否支持 EFS 加密等。

lpFileSystemNameBuffer：输出参数，表示文件系统类型，如“NTFS”、“CDFS”等。

nFileSystemNameSize: lpFileSystemNameBuffer 的缓冲区大小。

◇返回值

返回 BOOL 值，表示信息获取是否成功。

2．实例 4-1　使用 GetLogicaIDriveStrings 获取驱动器根路径

本示例使用 GetLogicalDriveStrings 获取系统中所有驱动器的根路径，在获取了所有的驱动器根路径后，使用自定义的 GetDirverInfo 函数获取驱动器属性。GetDirverInfo 函数原型将在实例 4-2 中详细介绍。

```
/* ************************************
* GetVolumeInfo.c 遍历驱动器并获取驱动器属性
 ************************************/
/* 头文件 */
```

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
/* 预定义 */
#define BUFSIZE 1024
/* 函数申明 */
BOOL GetDirverInfo(LPSTR szDrive);

/* ************************************
 *应用程序主函数，遍历驱动器并调用 GetDirverInfo，获取驱动器属性
 ************************************/
void main(void)
{
     CHAR szLogicalDriveStrings[BUFSIZE];
     PCHAR szDrive;

    ZeroMemory(szLogicalDriveStrings,BUFSIZE);
    // 获取逻辑驱动器卷标名
     GetLogicalDriveStrings(BUFSIZE - 1,szLogicalDriveStrings);
     szDrive = (PCHAR)szLogicalDriveStrings;
     // 循环处理每个卷
     do
     {
          if(!GetDirverInfo(szDrive))
          {
                printf("\nGet Volume Information Error: %d", GetLastError());
          }
          szDrive += (lstrlen(szDrive)+1);
     }
     while(*szDrive!='\x00');
}
```

在获取所有驱动器信息后main函数使用do_while循环，以获取的驱动器根路径为参数，调用GetDirverInfo函数(见实例4-2)依次获取各个驱动器的属性。

3．实例4-2 使用FindFirstVolume系列函数遍历驱动器，获取驱动器信息

下面是使用FindFirstVolume、FindNextVolume和FindVolumeClose函数，实现遍历驱动器、获取驱动器名的功能。对每个所获取的驱动器使用GetDriverInfo，获取驱动器属性，并将获取的信息打印出来。

```
/* ************************************
* Enumeratevolume.c    遍历驱动器并获取驱动器属性
************************************/
#define _WIN32_WINNT 0x0501
/* 头文件 */
#include <windows.h>
#include <stdio.h>
/* 预定义 */
#define BUFSIZE          MAX_PATH
/* 函数申明 */
BOOL GetDirverInfo(LPSTR szDrive);

/* ************************************
* 功能：应用程序主函数，遍历驱动器并调用
*       GetDirverInfo 获取驱动器属性
************************************/
int main(void)
{
     TCHAR buf[BUFSIZE];     // 卷标信息
```

```
        HANDLE hVol;            // 卷遍历句柄
        BOOL bFlag;

        hVol = FindFirstVolume (buf, BUFSIZE );

        if (hVol == INVALID_HANDLE_VALUE)
        {
              printf (TEXT("No volumes found!\n"));
              return (-1);
        }
        GetDirverInfo (buf);
        while( FindNextVolume(
              hVol,
              buf,
              BUFSIZE
              ))
        {
              GetDirverInfo (buf);
        }
        bFlag = FindVolumeClose(
              hVol
              );
        return (bFlag);
    }
```

在实例 4-1 和本实例中 main 函数中都调用了 GetDirverInfo 函数，用于获取指定的驱动器的属性信息。在这个函数中，首先调用了 GetDriveType 函数获取驱动器的物理类型，GetDriveType 返回驱动器类型：然后调用 GetVolumeInformation API 函数，获取驱动器盼相关属性。

```
    /* ************************************
     * BOOL GetDirverInfo(LPSTR szDrive)
     * 功能：获取驱动器的属性
     * 参数：LPSTR szDrive
     *   指明要获取属性的驱动器的根路径,如C:\
     * 返回值BOOL：表示是否成功
     ************************************/
    BOOL GetDirverInfo(LPSTR szDrive)
    {
         UINT uDriveType;
         DWORD dwVolumeSerialNumber;
         DWORD dwMaximumComponentLength;
         DWORD dwFileSystemFlags;
         CHAR szFileSystemNameBuffer[BUFSIZE];
         CHAR szDirveName[MAX_PATH];
         printf("\n%s\n",szDrive);
         uDriveType = GetDriveType(szDrive);
         switch(uDriveType)
         {
         case DRIVE_UNKNOWN:
              printf("The drive type cannot be determined. ");
              break;
         case DRIVE_NO_ROOT_DIR:
             printf("The root path is invalid, for example, no volume is mounted at the path. ");
              break;
         case DRIVE_REMOVABLE:
              printf("The drive is a type that has removable media, for example, a floppy drive
or removable hard disk. ");
              break;
         case DRIVE_FIXED:
             printf("The drive is a type that cannot be removed, for example, a fixed hard drive. ");
              break;
```

```
        case DRIVE_REMOTE:
             printf("The drive is a remote (network) drive. ");
             break;
        case DRIVE_CDROM:
             printf("The drive is a CD-ROM drive. ");
             break;
        case DRIVE_RAMDISK:
             printf("The drive is a RAM disk. ");
             break;
        default:
             break;
        }
        if (!GetVolumeInformation(
             szDrive,
             szDirveName,
             MAX_PATH,
             &dwVolumeSerialNumber,
             &dwMaximumComponentLength,
             &dwFileSystemFlags,
             szFileSystemNameBuffer,
             BUFSIZE
             ))
        {
             return FALSE;
        }
        if(0!=lstrlen(szDirveName))
        {
             printf ("\nDrive Name is %s\n",szDirveName);
        }
        printf ("\nVolume Serial Number is %u",dwVolumeSerialNumber);
        printf ("\nMaximum Component Length is %u",dwMaximumComponentLength);
        printf ("\nSystem Type is %s\n",szFileSystemNameBuffer);
        if(dwFileSystemFlags & FILE_SUPPORTS_REPARSE_POINTS)
        {
             printf ("The file system does not support volume mount points.\n");
        }
        if(dwFileSystemFlags & FILE_VOLUME_QUOTAS)
        {
             printf ("The file system supports disk quotas.\n");
        }
        if(dwFileSystemFlags & FILE_CASE_SENSITIVE_SEARCH)
        {
             printf ("The file system supports case-sensitive file names.\n");
        }
        //you can use these value to get more informaion
        //FILE_CASE_PRESERVED_NAMES
        //FILE_CASE_SENSITIVE_SEARCH
        //FILE_FILE_COMPRESSION
        //FILE_NAMED_STREAMS
        //FILE_PERSISTENT_ACLS
        //FILE_READ_ONLY_VOLUME
        //FILE_SUPPORTS_ENCRYPTION
        //FILE_SUPPORTS_OBJECT_IDS
        //FILE_SUPPORTS_REPARSE_POINTS
        //FILE_SUPPORTS_SPARSE_FILES
        //FILE_UNICODE_ON_DISK
        //FILE_VOLUME_IS_COMPRESSED
        //FILE_VOLUME_QUOTAS
        printf("...\n");
        return TRUE;
}
```

以下是在笔者主机上运行实例 4-1 和实例 4-2 所得输出结果的一部分。

```
C:\
The drive is a type that cannot be removed, for example, a fixed hard drive.
Volume Serial Number is 3638786360
Maximum Component Length is 255
System Type is NTFS
The file system does not support volume mount points.
The file system supports disk quotas.
The file system supports case-sensitive file names.
...
I:\
The drive is a CD-ROM drive.
Volume Serial Number is 1768674895
Maximum Component Length is 110
System Type is CDFS
The file system supports case-sensitive file names.
...
```

4.2.2 操作驱动器挂载点

驱动器挂载点，又可以称作卷挂载点。挂载点实际上是操作系统或者用户设置的，用来进入一个逻辑驱动器或者卷的入口。在设置了卷的挂载点后，用户或者应用程序可以使用卷标或者指定的挂载点来进入卷。比如笔者在“C:\”下设置了一个“E\”：卷的挂载点 mnt，那么打开“E:\”和打开“C:\mnt”实际上都是进入“E:\”卷。

一般可以用 FindFirstVolumeMountPoint 系列的 API 来找到一个卷的所有挂载点；用 GetVolumeNameForVolumeMountPoint 来获取指定挂载点所指向的卷名，卷名形式为"\\?\Volume{GUID}\”;用 SetVolumeMountPoint 来设置新的挂载点。

通过系统的磁盘管理功能可以设置卷的挂载点，如图 4-2 所示。

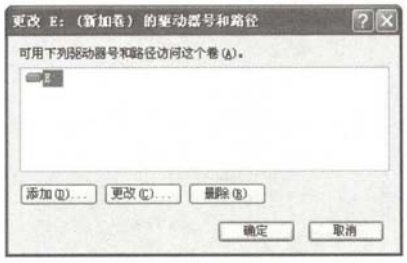

图 4-2 挂载卷

◇“我的电脑”图标右键菜单中选择“管理”。

◇弹出“计算机管理”窗口，选择“磁盘管理”。

◇选中需要挂载的卷，在右键菜单中选择“更改驱动　　器名和路径”。

◇在弹出的对话框中单击“添加”按钮，选择“装入　　以下空白 NTFS 文件夹”。

◇选择需要将卷挂载入的文件夹（空白），单击“确定”按钮。

◇卷就被装入文件夹中，之后就可以和访问文件夹一个访问这个卷了，如图 4-3 和 4-4 所示。

1．关键 API

(1) FindFirstVolumeMountPoint.

获取指定卷的第一个挂载点，函数原型如下：

```
HANDLE FindFirstVolumeMountPoint(
  LPTSTR lpszRootPathName,
  LPTSTR lpszVolumeMountPoint,
  DWORD cchBufferLength
);
```

◇参数

图 4-3　被挂载入 C:\mnt 的 E:\盘

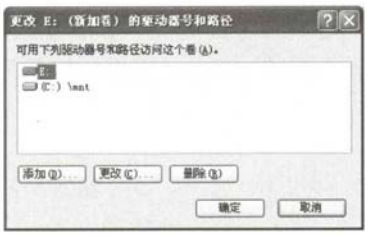

图 4-4　挂载到 C:\mnt 的 E:\盘

lpszRootPathName：输入参数，指定要查找的卷名，必须以反斜杠结尾。

lpszVolumeMountPoint：输出参数，找到的第一个挂载点。

cchBufferLength. 输入参数，用来储存输出挂载点的缓存的大小。

◇返回值

返回 HANDLE 值，为一个查找句柄，FindNextVolumeMountPoint 用该句柄查找下一个挂载点。错误时值为 INVALLD HANDLE VALUE,以用 GetLastError()函数获取更详细的错误信息。

(2) FindNextVolumeMountPoint

查找指定卷的后继挂载点，函数原型如下：

```
BOOL FindNextVolumeMountPoint(
  HANDLE hFindVolumeMountPoint,
  LPTSTR lpszVolumeMountPoint,
  DWORD cchBufferLength
);
```

◇参数

hFindVolumeMountPoint:输入参数，查找句柄，由 FindFirstVolumeMountPoint 获取。

lpszVolumeMountPoint:输出参数，找到的后继挂载点。

cchBufferLength：输入参数，用来储存输出挂载点的缓存的大小。

◇返回值

返回 BOOL 值，表示查找是否成功，失败并且 GetLastError 函数返回 ERROR NO_ MORE FILE$代码时表示已经查找完所有挂载点。

（3）FindVolumeMountPointClose.

关闭 FindVolumeMountPointClose 打开的卷句柄，其函数原型如下：

```
BOOL FindVolumeMountPointClose(
  HANDLE hFindVolumeMountPoint
);
```

◇参数

hFindVolumeMountPoint:要关闭的挂载点查找句柄。

◇返回值

(4)GetVolumeNameForVolumeMountPoint。

根据指定的挂载点获取相应的卷设备名，函数原型如下：

```
BOOL GetVolumeNameForVolumeMountPoint(
  LPCTSTR lpszVolumeMountPoint,
```

◇参数

lpszVolumeMountPoint:输入参数，指定需要查找挂载点或者根目录，以反斜杠结束。

lpszVolumeName:输出参数，挂载点对应的卷设备名，形式为“\\?\Volume{GUID}\”。

cchBufferLength:输入参数，用来储存输出设备名的缓存大小。

◇返回值

返回 BOOL 值，表示函数是否成功，同样可以用 GetLastError 函数获取更详细的错误信息。

（5）SetVolumeMountPc

将指定卷挂载到指定挂载点处，函数原型如下：

```
BOOL SetVolumeMountPoint(
  LPCTSTR lpszVolumeMountPoint,
  LPCTSTR lpszVolumeName
);
```

◇参数

lpszVolumeMountPoint:输入参数，指定的挂载点，挂载点必须为一个根路径或者一个在现有卷上的路径，必须以反斜杠结束。

lpszVolumeName:输入参数，卷设备名，形式为“\\?\Volume{GUID}\”。

◇返回值

返回 BOOL 值表示函数是否成功，同样可以用 GetLastError()函数获取更详细的错误信息。

2．实例 4-3 查找、设置挂载点，通过挂载点获取卷设备名

本实例可以设置卷的挂载点，可以列举指定的卷中具有的挂载点和路径。

```
/* ************************************
* mount.c卷挂载点操作
 ************************************/
/* 预编译声明 */
#define _WIN32_WINNT 0x0501
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
#define BUFSIZE          MAX_PATH
#define FILESYSNAMEBUFSIZE MAX_PATH

/* ************************************
 * ProcessVolumeMountPoint
 * 功能：列举挂载点
 ************************************/
BOOL ProcessVolumeMountPoint (HANDLE hPt,
                              TCHAR *PtBuf, DWORD dwPtBufSize,
                              TCHAR *Buf)
{
     BOOL bFlag;                        // 结果
     TCHAR Path[BUFSIZE];               // 全路径
     TCHAR Target[BUFSIZE];             // 挂载点设备
     printf ("\tVolume mount point found is \"%s\"\n", PtBuf);
     lstrcpy (Path, Buf);
     lstrcat (Path, PtBuf);
     bFlag = GetVolumeNameForVolumeMountPoint(
          Path,  Target,   BUFSIZE  );
     if (!bFlag)
```

```
            printf ("\tAttempt to get volume name for %s failed.\n", Path);
      else
            printf ("\tTarget of the volume mount point is %s.\n", Target);
      bFlag = FindNextVolumeMountPoint(
            hPt,  PtBuf,  dwPtBufSize );
      return (bFlag);
}
/* ************************************
 * ProcessVolume
 * 功能: 判断卷类型, 列举挂载点
 ************************************/
BOOL ProcessVolume (HANDLE hVol, TCHAR *Buf, DWORD iBufSize)
{
      BOOL bFlag;          // 返回标志
      HANDLE hPt;          // 卷句柄
      TCHAR PtBuf[BUFSIZE]; // 挂载点路径
      DWORD dwSysFlags;     // 文件系统标记
      TCHAR FileSysNameBuf[FILESYSNAMEBUFSIZE];
      printf ("Volume found is \"%s\".\n", Buf);
      // 是否为 NTFS
      GetVolumeInformation( Buf, NULL, 0, NULL, NULL,
            &dwSysFlags, FileSysNameBuf,
            FILESYSNAMEBUFSIZE);
      if (! (dwSysFlags & FILE_SUPPORTS_REPARSE_POINTS))
      {
            printf ("\tThis file system does not support volume mount points.\n");
      }
      else
      {
            // 本卷中的挂载点
            hPt = FindFirstVolumeMountPoint(
                  Buf, //卷的路径
                  PtBuf, // 挂载点路径
                  BUFSIZE
                  );
            if (hPt == INVALID_HANDLE_VALUE)
            {
                  printf ("\tNo volume mount points found!\n");
            }
            else
            {
                  // 处理挂载点
                  bFlag = ProcessVolumeMountPoint (hPt,
                        PtBuf,
                        BUFSIZE,
                        Buf);
                  // 循环
                  while (bFlag)
                        bFlag =
                        ProcessVolumeMountPoint (hPt, PtBuf, BUFSIZE, Buf);
                  // 结束
                  FindVolumeMountPointClose(hPt);
            }
      }
      // 下一个
      bFlag = FindNextVolume(
            hVol, Buf,  iBufSize);
      return (bFlag);
}
/* ************************************
 * int GetMountPoint(void)
 * 功能: 获取挂载点
 ************************************/
```

```
int GetMountPoint(void)
{
     TCHAR buf[BUFSIZE];             // 卷标识符
     HANDLE hVol;                    // 卷句柄
     BOOL bFlag;                     // 结果标志
     printf("Volume mount points info of this computer:\n\n");
     // 打开卷
     hVol = FindFirstVolume (buf, BUFSIZE );
     if (hVol == INVALID_HANDLE_VALUE)
     {
          printf ("No volumes found!\n");
          return (-1);
     }
     bFlag = ProcessVolume (hVol, buf, BUFSIZE);
     while (bFlag)
     {
          bFlag = ProcessVolume (hVol, buf, BUFSIZE);
     }
     bFlag = FindVolumeClose(     hVol );
     return (bFlag);
}

/* ************************************
 * void Usage (PCHAR argv)
 * 功能: 使用方法
 ************************************/
void Usage (PCHAR argv)
{
     printf( "\n\n\t%s, mount a volume at a mount point.\n", argv );
     printf( "\tFor example, \"mount D:\\mnt\\drives\\ E:\\\"\n" );
}

/* ************************************
 *   main
 * 功能   入口函数
 ************************************/
int main( int argc, PCHAR argv[] )
{
     BOOL bFlag;
     CHAR Buf[BUFSIZE];
     if( argc != 3 )
     {
          GetMountPoint();
          Usage( argv[0] );
          return( -1 );
     }
     bFlag = GetVolumeNameForVolumeMountPoint(
          argv[2],           // 输入挂载点或目录
          Buf,               // 输出卷名
          BUFSIZE
                    );
     if (bFlag != TRUE)
          {
          printf( "Retrieving volume name for %s failed.\n", argv[2] );
          return (-2);
     }
     printf( "Volume name of %s is %s\n", argv[2], Buf );
     bFlag = SetVolumeMountPoint(
          argv[1],      // 挂载点
          Buf           // 需要挂载的卷
          );
     if (!bFlag)
     {
          printf ("Attempt to mount %s at %s failed. error code is\n",
          argv[2], argv[1], GetLastError());
     }
     return (bFlag);
}
```

编译实例后，运行 mount.exe，打印出了第一个卷中的 mnt 目录下有挂载点。挂载的是第三

个卷。

之后运行 mount.exe C:\mnt2\ E:\，并将 E 盘挂载到 C:\mnt2\。运行后，显示挂载成功。再次运行 mount.exe，发现有新的挂载点，运行结果如图 4-5 所示。

图 4-5　使用 mount.exe 操作挂载点

4.2.3　判断光驱中是否有光盘

判断光驱中是否有光盘，仍然可以使用在 4.2.1 小节介绍的 GetDriveType 和 GetVolumeInformation 函数实现。首先使用驱动器根路径作为 GetDriveType 和参数，如果返回值是 DRIVE_CDROM，则说明此驱动器为光驱。然后使用 GetVolumeInformation 获取信息，如果成功，则说明存光盘已经放入。调用完成后 GetVolumeInformation 函数的第 7 个参数 LPTSTR lpFileSystemNameBuffer 存储的是文件系统的类别字符串，光盘一般是 CDFS。如果调用 GetVolumeInformation 时返回 FALSE，并且 GetLastError 返回 21，则说明驱动器中未放入光盘。

1．关键 API

（1）GetDiskType 与 GetVolumeInformation。

这两个 API 已经在 4.2.1 小节介绍过，这里不再赘述。

（2）GetLastError。

获取在执行中本线程最近的一次错误。本函数是很多系统 API 返回执行错误原因的方法。可能使用 SetLastError 函数设置本线程的 Last-Error 值。GetLastError 函数原型如下：

```
DWORD GetLastError(void);
```

◇参数

无参数。

◇返回值

返回值类型为 DWORD，表示本线程的 Last-Error 值。

> **注意** GetLastError 函数常常用于 Windows API 返回执行错误，这个函数在进行 Windows 应用程序设计时常常会用到，是 Windows API 中最为常用的 API 函数。在以后各章节中还会经常使用到 GetlastError 函数。GetlastError 函数可以和 error lookup 工具结合使用。

2．实例 4-4 判断光驱内是否有光盘

本实例使用 GetDriveType 函数获取驱动器类别，并判断是否为 DRIVE_CDROM（光盘尖圆）。然后使用 GetVolumeInformation 函数和 GetLastError 函数判断光驱中是否有光盘及获取光盘的文件系统类别。

```
/* ************************************
* cdrom.c 判断光驱中是否有光盘
 ************************************/
/* 头文件  */
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
/* 预定义  */
#define BUFSIZE          512
/* ************************************
 * int main( int argc, PCHAR argv[] )
 * 功能：应用程序主函数，根据输入的驱动器
 *       根路径参数判断是否为光驱，是否放入光盘
* 参数：驱动器根路径，比如“D:\”
 ************************************/
int main( int argc, PCHAR argv[] )
{
      //存储文件系统类别名
      CHAR szFileSystemNameBuffer[BUFSIZE];
      DWORD dwLastError;
      DWORD dwFileSystemFlags;
      //判断是否输入运行时参数
      if( argc != 2 )
      {
            printf("请输入驱动器的根路径，比如：  \"D:\\\"\n");
            return( -1 );
      }
      //判断输入的驱动器是否为 CD\DVD ROM
      if(GetDriveType(argv[1])!=DRIVE_CDROM)
      {
            printf("驱动器 %s 不是 CD/DVD ROM。\n",argv[1]);
            return( -1 );
      }
      //获取卷信息
      if (!GetVolumeInformation(
            argv[1], NULL, 0,
            NULL,NULL,
            &dwFileSystemFlags,
            szFileSystemNameBuffer,
            BUFSIZE
            ))
    {
            dwLastError = GetLastError();
```

```
            if(dwLastError == 21)
            {
                printf("设备未就绪，请放入光盘！\n");
                return 0;
            }
            else
            {
                printf("GetVolumeInformation 错误 %d\n",dwLastError);
                return 0;
            }
        }
        printf ("光盘已经放入，文件系统类别 %s。\n",szFileSystemNameBuffer);
        return 0;
    }
```

以上实例都可以使用配套光盘中的 makefile 编译，也可以直接在 IDE 中生成，得到 cdrom.exe。

下面是实例在笔者机器上运行的结果。驱动器 G：是光盘，D：是硬盘。

放入光盘前，运行“cdrom.exe G:\”，结果为：

设备未就绪，请放入光盘！

放入光盘后：运行“cdrom.exe G:\”,结果为：

光盘已经放入，文件系统类别 CDFS。

运行“cdrom.exe D:\”

4.2.4 获取磁盘分区的总容量、空闲容量、簇、扇区信息

获取磁盘分区的总容量和空闲空间的容量可以使用 GetDiskFreeSpace 函数或 GetDiskFree SpaceEx 函数。GetDiskFreeSpace 使用 DWORD 类型作为输出参数，由于 DWOR 长度为 32 位,最大只能表示 4GB,而一般的磁盘分区大小都大于 4GB,所以,GetDiskFreeSpace 并不直接返回磁盘的总容量和空闲空间的容量，而是使用总簇数、空闲的簇数、每簇的扇区数、每扇区的字节数来表示。用户在编程时，可以使用它们的乘积来获得最终结果。而 GetDiskFreeSpaceEx 使用 ULARGE_INTEGER (DWORD64)类型的数据来存储磁盘空间总空间和剩余空间，所以可以直接获得结果。DWORD64 可以表示约 16777216TB 的数据量(DWORD64 最大可表示 2^{64}Byte，1TB=2^{40}Byte，所以结果最大为 $2^{64}/2^{40}=2^{24}$TB=16777216TB)。

1．关键 API

(1) GetDiskFreeSpace。

获取驱动器根路径作为输入，获取磁盘空间信息，包括每簇的扇区数、每扇区的字节数、簇数量、空闲的簇数量，其函数原型如下：

```
BOOL GetDiskFreeSpace(
  LPCTSTR lpRootPathName,
  LPDWORD lpSectorsPerCluster,
  LPDWORD lpBytesPerSector,
  LPDWORD lpNumberOfFreeClusters,
  LPDWORD lpTotalNumberOfClusters
);
```

◇参数

lpRootPathName:输入参数，指向所要获取磁盘空间信息的磁盘分区根路径字符串。

lpSectorsPerCluster:输出参数，指针，指向存储每簇扇区数的 DWORD 类型变量。

lpBytesPerSector:输出参数，指针，指向存储每扇区字节数的 DWORD 类型变量。

lpNumberOfFreeClusters:输出参数，指针，指向存储空闲簇的 DWORD 类型变量。

lpTotalNumberOfClusters:输出参数，指针，指向存储总簇数的 DWORD 类型变量。

◇返回值

返回 BOOL 值，表示信息获取是否成功。

◇使用说明

总簇数乘以每簇扇区数乘以每扇区字节数就是磁盘分区的总容量（字节），空闲簇数乘以每簇扇区数，乘以每扇区字节数就是磁盘分区的空闲空间容量（字节）。

参数 lpRootPathName 如果是 NULLL，函数使用程序运行的当前路径所在的磁盘分区。

(2) GetDiskFreeSpacEX。

获取驱动器根路径作为输入，获取用户可用的空闲空间的字节数、空闲空间的字节数、磁盘总容量的字节数，其函数原型如下：

```
BOOL GetDiskFreeSpaceEx(
  LPCTSTR lpDirectoryName,
  PULARGE_INTEGER lpFreeBytesAvailable,
  PULARGE_INTEGER lpTotalNumberOfBytes,
  PULARGE_INTEGER lpTotalNumberOfFreeBytes
);
```

> **注意** 第二个参数和第四个参数的区别在于，虽然磁盘上有空闲空间但可能是操作系统预留的，普通用户无法使用。第四个参数是指所有空闲的空间，第二个参数才获得用户实际可用的空闲空间。

◇参数

lpDirectoryName:输入参数，指向所要获取磁盘空间信息的磁盘分区根路径字符串。

lpFreeBytesAvailable:输出参数，指针，指向用于存储可获得的总字节数量的变量。

lpTotalNumberOfBytes:输出参数，指针，指向存储磁盘空间总字节数据的变量。

lpTotalNumberOfFreeBytes:输出参数，指针，指向存储磁盘空闲空间总字节的变量。

◇返回值

返回 BOOL 值，表示信息获取是否成功。

◇使用说明

ULARGE INTEGER 类型的数据可以直接强制转换为 DWORD64 类型的变量，并进行运算。

2．关键数据结构

GetDiskFreeSpaceEx 函数使用到了数据结构 ULARGE_INTEGER，数据类型 PULARGE INTEGER 是指向它的指针。ULARGE__ INTEGER 的定义如下，此数据结构使用两个 DWORD 来表示 64 位数据。低位存储于前，高位存储于后，与 DWORD64 的存储形式是一致的，所以可以直接强制类型转换为 DOWRD64 类型。也可以直接使用 QuadPart 成员，QuadPart 成员是 ULONGLONG 形数据结构，在一般 32 位主机上，与 DWORD64 具有同样的长度。

```
typedef union _ULARGE_INTEGER {
 struct {
    DWORD LowPart;
    DWORD HighPart;
 };
 struct {
```

```
        DWORD LowPart;
        DWORD HighPart;
    } u;
    ULONGLONG QuadPart;
} ULARGE_INTEGER, *PULARGE_INTEGER;
```

3 实例 4-5 使用 GetDiskFreeSpace 系列函数获取磁盘空间信息

本实例使用GetDiskFreeSpaceEx 函数和GetDiskFreeSpace 函数两种不同的方法来获取磁盘分区容量信息，并把信息打印出来。使用配套光盘所给的 makefile 编译运行。

```
/* ************************************
 * diskspace.c 获取磁盘空间信息
 ************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
/* ************************************
 * BOOL GetDiskSpaceInfo(LPCSTR pszDrive
 * 功能：根据输入的驱动器，获取磁盘总容量
 *       空闲空间、簇数量等磁盘信息
 * 参数：驱动器根路径，比如“D:\”
 ************************************/
BOOL GetDiskSpaceInfo(LPCSTR pszDrive)
{
    DWORD64 qwFreeBytesToCaller, qwTotalBytes, qwFreeBytes;
    DWORD dwSectPerClust, dwBytesPerSect, dwFreeClusters, dwTotalClusters;
    BOOL bResult;
    //使用 GetDiskFreeSpaceEx 获取磁盘信息并打印结果
    bResult = GetDiskFreeSpaceEx (pszDrive,
        (PULARGE_INTEGER)&qwFreeBytesToCaller,
        (PULARGE_INTEGER)&qwTotalBytes,
        (PULARGE_INTEGER)&qwFreeBytes);
    if(bResult)
    {
        printf("使用 GetDiskFreeSpaceEx 获取磁盘空间信息\n");
        printf("可获得的空闲空间（字节）：\t%I64d\n", qwFreeBytesToCaller);
        printf("空闲空间（字节）：\t\t%I64d\n", qwFreeBytes);
        printf("磁盘总容量（字节）：\t\t%I64d\n", qwTotalBytes);
    }
    //使用 GetDiskFreeSpace 获取磁盘信息并打印结果
    bResult = GetDiskFreeSpace (pszDrive,
        &dwSectPerClust,
        &dwBytesPerSect,
        &dwFreeClusters,
        &dwTotalClusters);
    if(bResult)
    {
        printf("\n使用 GetDiskFreeSpace 获取磁盘空间信息\n");
        printf("空闲的簇数量 ：\t\t%d\n",dwFreeClusters);
        printf("总簇数量 ：\t\t%d\n",dwTotalClusters);
        printf("每簇的扇区数量 ：\t\t%d\n",dwSectPerClust);
        printf("每扇区的容量（字节）：\t\t%d\n",dwBytesPerSect);
        printf("空闲空间（字节）：\t\t%I64d\n",
            (DWORD64)dwFreeClusters*
            (DWORD64)dwSectPerClust*(DWORD64)dwBytesPerSect);
        printf("磁盘总容量（字节）：\t\t%I64d",
            (DWORD64)dwTotalClusters*
            (DWORD64)dwSectPerClust*(DWORD64)dwBytesPerSect);
    }
    return bResult;
}
```

```
/* ************************************
 * int main( int argc, PCHAR argv[] )
 * 功能：应用程序主函数，根据输入参数
 *          调用 GetDiskSpaceInfo 函数获取
 *          磁盘空间信息
 * 参数：驱动器根路径，比如“D:\”
 ************************************/
int main(int argc, PCHAR argv[])
{
    GetDiskSpaceInfo (argv[1]);
}
```

运行“diskSDace.exe C:”，结果为：

```
使用 GetDiskFreeSpaceEx 获取磁盘空间信息
可获得的空闲空间（字节）：  6849912832
空闲空间（字节）：          6849912832
磁盘总容量（字节）：        21476171776

使用 GetDiskFreeSpace 获取磁盘空间信息
空闲的簇数量 ：             1672342
总簇数量 ：                 5243206
每簇的扇区数量 ：           8
每扇区的容量（字节）：      512
空闲空间（字节）：          6849912832
磁盘总容量（字节）：        21476171776
```

图 4-6 是通过磁盘驱动器属性得到的结果，可以看到结果是一致的。

图 4-6　查看驱动器磁盘空间信息

4.3　文件和目录管理

文件和目录管理是 Windows 系统编程最为基本的内容，几乎所有的应用程序都会使用到文件和目录的操作。本节将向读者演示如何创建目录、遍历目录、创建文件、打开文件、读写文件、移动复制删除文件等。本节通过以下多个实例来讲解 Windows API 对文件和目录的管理。

◇删除、复制、重命名、移动文件。

◇创建、打开、读写文件。

◇创建、打开目录。

◇获取当前目录、获取程序所在的目录、获取模块路径。

◇查找文件、遍历目录下的文件和子目录。

◇递归遍历目录树。

◇获取、设置文件属性和时间。

4.3.1 删除、复制、重命名、移动文件

Windows 系统为文件的删除、复制、重命名或移动文件提供了相应的 API 函数。删除文件使用 DeleteFile 函数；复制文件使用 CopyFile 函数；重命名文件和移动文件实际是一个操作，使用 MoveFile 函数。这几个函数的使用都非常简单，下面分别介绍。

1．关键 API

(1) DeleteFile。

DeleteFile 的功能是删除文件。以文件路径作为输入，指向需要删除的文件。文件路径可以是类似于“c：\files\delete.txt”的绝对路径，也可以是类似于“.\delete.txt”的相对路径，二相对于可执行文件所在的路径。

```
BOOL DeleteFile(
  LPCTSTR lpFileName
);
```

◇参数

lpFileName:输入参数，所要删除的文件的相对路径或绝对路径。

◇返回值

返回 BOOL 值，表示文件删除是否成功。

◇使用说明

如果程序返回失败，可以使用 GetLastError 函数获取错误信息。

(2) CopyFile。

CopyFile 的功能是复制文件。通过参数输入复制文件和源路径和目的路径，路径可以是绝对路径也可以是相对路径,还可以通过参数指明如果目的路径已经存在文件,是否覆盖。可以使用 CopyFileEx 函数进行更为高级的操作，比如在复制进行过程中取消复制等。CopyFileEx 可以指定
一个回调函数来处理文件复制中所可能发生的各种情况。

```
BOOL CopyFile(
  LPCTSTR lpExistingFileName,
  LPCTSTR lpNewFileName,
  BOOL bFailIfExists
);
```

◇参数

lpExistingFileName:输入参数，已经存在的所需复制文件的源路径。

lpNewFileName:输入参数，新文件路径，复制文件的目的路径。

bFailIfExists:输入参数，指明如果在目的路径存在文件时是否覆盖。如果设为 TRUE 将不覆盖已经存在的文件，如果存在，则返回失败。这时使用 GetLastError 获取错误代码，将返回 80(0x50)。

◇返回值

返回 BOOL 值，表示文件复制是否成功。

◇使用说明

如果程序返回失败，可以使用 GetLastError 函数获取错误信息。

(3) MoveFile。

MoveFile 的功能是移动、重命名文件和目录。通过参数输入源路径和目的路径，路径可以是绝对路径也可以是相对路径，如果目的路径的文件或目录已经存在，则返回失败。可以使用 MoveFileEx 函数来指定更多的选项，如果已经存在是否替换等。还可以使用

MoveFileWithProgress 指定一个回调函数来处理文件移动中所可能发生的各种情况。函数原型如下：

```
BOOL MoveFile(
  LPCTSTR lpExistingFileName,
  LPCTSTR lpNewFileName
);
```

◇参数

lpExistingFileName：输入参数，为已经存在的所需移动文件的源路径。

lpNewFileName：输入参数，新文件路径，移动文件的目的路径。

◇返回值

返回 BOOL 值，表示文件移动是否成功。

◇使用说明

如果程序返回失败，可以使用 GetLastError 函数获取错误信息。

(4) CopyFileEx、MoveFileEx 以及 MoveFileWithProgreSS.

这 3 个 API 函数功能更丰富，但是限于篇幅这里不再做详细介绍，读者可以使用 SDK 文档学习它们的使用方法。

2．实例 4-6 删除、复制、移动文件演示实例

本实例使用 DeleteFile、CopyFile、MoveFile 来完成文件的删除、复制和移动功能。编译生成可执行文件，通过参数来指定程序完成的功能。-d 参数表示删除文件，后面跟所需删除的文件路径；-m 参数表示移动、重命名文件，之后的参数分别是源路径和目的路径；-c 参数表示复制文件，之后跟源路径和目的路径。在复制文件过程中，如果文件已经存在，会提醒用户是否覆盖。

```
/* ************************************
 * files.c 删除、复制、重命名、移动文件
 ************************************/
/* 头文件  */
#include <windows.h>
#include <stdio.h>
/* ************************************
 * int main( int argc, PCHAR argv[] )
 * 功能：应用程序主函数，根据输入参数
 *       删除、复制、重命名文件
 * 参数：删除文件：
 *           -d 文件路径
 *       将文件路径 1 的文件复制到文件路径 2：
 *           -c 文件路径 1 文件路径 2
 *       将文件路径 1 的文件移动、重命名为文件路径 2 的文件：
 *           -m 文件路径 1 文件路径 2
 ************************************/
int main(int argc, PCHAR argv[])
{
     //-d 参数，删除文件
     if(0==lstrcmp("-d",argv[1]) && argc==3)
     {
          if(!DeleteFile(argv[2]))
               printf("删除文件错误：%x\n",GetLastError());
          else
```

```
                printf("删除成功! \n");
    }
    //-c参数，复制文件
    //如果文件存在，询问用户是否覆盖
    else if(0==lstrcmp("-c",argv[1]) && argc==4)
    {
          //复制，不覆盖已经存在的文件
          if(!CopyFile(argv[2],argv[3],TRUE))
          {
                //LastError == 0x50，文件存在
                if(GetLastError() == 0x50)
                {
                     printf("文件%s 已经存在，是否覆盖? y/n: ",argv[3]);
                     if('y'==getchar())
                     {
                             //复制，覆盖已经存在的文件
                             if(!CopyFile(argv[2],argv[3],FALSE))
                                   printf("复制文件错误，%d\n",GetLastError());
                             else
                                   printf("复制成功! \n");
                     }
                     else
                            return 0;
                }
          }
          else
               printf("复制成功! \n");
    }
    //-m参数，移动、重命名文件
    else if(0==lstrcmp("-m",argv[1]) && argc==4)
    {
         if(!MoveFile(argv[2],argv[3]))
               printf("移动文件错误: %d\n",GetLastError());
         else
               printf("移动文件成功! \n");
    }
    else
         printf("参数错误! \n");
}
```

下面将演示本示例程序的使用，编译生成 files.exe，任意找一个文件，比如“C:\a.txt”，进行下面的操作。

运行“fileq_exe -c C:\a.txt“C:\CODV of a.txt’’’’，正常情况下程序会打印输出：

```
复制成功!
```

此时文件C:\copy of a.txt 出现，内容与a.txt 相同。

运行“files.exe -c C:\a.txt“C:\copy of a.txt””，因为此时“C:\copy of a.txt”已经存在，所以程序会出现如下提示：

```
文件C:\copy of a.txt 已经存在，是否覆盖? y/n:
```

此时可以选择键入“y”或“n”，如果键入“n”回车，程序会直接退出。

运行“files.exe -d“C:\copy of a.txt’”’，正常情况下程序会打印输出：

```
删除成功!
```

此时文件“C:\copy of a.txt”已经不存在。

运行“files.exe -m C:\ a.txt C:\b.txt”，正常情况下程序会打印输出：

```
移动文件成功!
```

此时名为a.txt 的文件已经变成了b.txt，文件内容不变。

4.3.2 创建、打开、读写文件，获取文件大小

在 Windows 系统中，创建和打开文件都是使用 API 函数 CreateFile，CreateFile 通过指定不同的参数来表示是新建一个文件，打开已经存在的文件，还是重新建立文件等。读写文件最为直接的方式是使用 ReadFile 和 WriteFile 函数，也可以使用文件镜像，获取文件大小一般使用 GetFileSize 函数，也可以使用 GetFileAttributesEx 等函数（在 4.3.7 节介绍）。读写文件、获取文件大小之前都需要使用 CreateFile 创建或打开的文件，获得文件句柄。

在文件操作中，文件句柄是一个关键的概念。文件句柄惟一标识了一个文件，ReadFile、WriteFile、GetFileSize 等函数是使用文件句柄作为参数来表示，用户需要读、写、获取大小的文件是哪一个文件。在对文件进行操作前，都必须要使用 CreateFile 获得文件句柄。

1．关键 API

（1）CreateFile

CreateFile 是文件操作中最主要的一个函数。几乎所有的文件操作都需要使用到文件句柄。而 CreateFile 函数为这些操作建立文件句柄。CreateFile 函数定义如下：

```
HANDLE CreateFile(
 LPCTSTR lpFileName,
 DWORD dwDesiredAccess,
 DWORD dwShareMode,
 LPSECURITY_ATTRIBUTES lpSecurityAttributes,
 DWORD dwCreationDisposition,
 DWORD dwFlagsAndAttributes,
 HANDLE hTemplateFile
);
```

◇参数

CreateFile 函数的参数多，而且比较复杂，涉及文件名、文件的共享模式、存取方式、操作模式、权限、标志和文件属性等。每一个参数都可以有多种值可供选择，代表了不同的意义。

lpFileName:输入参数，操作对象文件的相对路径或绝对路径。

dwDesiredAccess:输入参数，指明对文件对象的操作存取方式，可以是 GENERIC_READ，表示需要读文件；可以是 GENERIC_WRITE，表示需要写文件；也可以是 GENERIC_READGENERIC_WRITE，表示既可以读也可以写。

dwShareMode；输入参数，共享模式。指明与其他进程是否共享该文件，可以是共享读(FILE_SHARE DELETE)、共享写(FILE_SHARE_WRITE)、共享删除(FILE_SHARE. READ)，如果要指明多个属性，使用“位与～‘I”运算。如果指明上述参数，其他进程就可以对文件进行相关操作。如果本进程需要独占本文件，则将本参数设置为 0。

lpSecurityAttributes:指向 SECURITY_ATTRIBUTES 结构的指针，表示本文件句柄的安全属性，能影响其是否可被子进程继承等操作。如果设定为 NULL，则子进程不可继承本句柄。SECURITY ATTRIBUTES 结构不常用，对此数据结构的设置，涉及 Windows 系统中对权限管理的原理，在本章中不作详细说明。

dwCreationDisposition:输入参数，操作模式。可以是表 4.4 中所列出的任一值。

表 4.4　　文件操作模式

参　数	操 作 模 式
CREATE_ALWAYS	如果指向的文件存在，则重建该文件，清除文件内容，清除文件属性；如果文件不存在，则新建一个文件
REATE_NEW	新建文件，如果文件已经存在，则返回失败
OPEN_ALWAYS	如果文件存在，则打开文件；如果文件不存在，则等同于 CREATE_NEW
OPEN_EXISTING	打开存在的文件，如果文件不存在，则删除文件
TRUNCATE_EXISTING	打开已经存在的文件，并把内容清空，文件大小变为零

dwFlagsAndAttributes:输入参数，文件属性和文件标志，其值如表 4.5 和表 4.6 所示。

表 4.5　　文件属性

文 件 属 性	意　义
FILE_ATTRIBUTE_ARCHIVE	存档文件
FILE_ATTRIBUTE_ENCRYPTED	加密文件
FILE_ATTRIBUTE_HIDDEN	隐藏文件
FILE_ATTRIBUTE_NORMAL	不含有其他文件属性，只能单独使用
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED	没有使用内容索引服务
FILE_ATTRIBUTE_OFFLINE	离线存储文件
FILE_ATTRIBUTE_READONLY	只读文件
FILE_ATTRIBUTE_SYSTEM	系统文件
FILE_ATTRIBUTE_TEMPORARY	临时文件

表 4.6　　文件操作标志.

操 作 标 志	意　义
FILE_FLAG_BACKUP_SEMANTICS	打开和创建文件是为了进行备份和恢复操作
FILE_FLAG_DELETE_ON_CLOSE	如果文件句柄全部关闭，则删除文件
FILE_FLAG_NO_BUFFERING	不使用系统缓存（caching，不同于硬件缓存）
FILE_FLAG_OPEN_NO_RECALL	远程存储
FILE_FLAG_OPEN_REPARSE_POINT	系统将禁止 NTFS 文件系统的“再解析”行为
FILE_FLAG_OVERLAPPED	以异步 IO 方式创建可打开文件
FILE_FLAG_POSIX_SEMANTICS	按照 POSIX 规则存取。
FILE_FLAG_RANDOM_ACCESS	随机存取
FILE_FLAG_SEQUENTIAL_SCAN	文件从头到尾是连续的（sequentially）
FILE_FLAG_WRITE_THROUGH	存取文件时，对磁盘进行直接的操作，不经过缓存

一般情况下文件属性较常用，而操作标志不常用，可以使用“1”运算符指定多个属性和标志。

hTemplateFile：输入参数，当存取权限包括 GENERIC_WRITE 时，可以设置为一个模板文件的句柄。一般情况下，可设置为 NULL，表示不使用模板文件。

◇返回值

返回 HANDLE 数据类型值，表示文件的句柄，如果返回 INVALID_HANDLE_VALUE，表示操作失败。

◇使用说明

Windows 系统中文件命名的方式可以参见本书 4.1.1 节对 Windows 系统中文件命名的说明。

Createfile 作的文件不仅仅是存在于文件系统中的文件，而包括设备对象、COM 接口等。

（2）ReadFile。

ReadFile 动能是从文件中读出数据。需要使用 CreateFile 所返回的文件句柄。函数原型如下：

```
BOOL ReadFile(
  HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

◇参数

hFile:输入参数，读取数据的文件对象，由 CreateFile 创建，调用 CreateFile 打开文件时需要指明 GENERIC_READ 读取操作模式。

lpBuffer：输入参数，指向读取文件数据存储的内存缓冲区。

nNumberOfBytesToRead：输入参数，指明需要从文件中读出的数据的大小，不能大于 lpBuffer 内存块的大小，否则会造成溢出。

lpNumberOfBytesRead：输出参数，指向存储实际读出的数据大小的 DWORD 变量。在读文件时，可能由于已经到达文件尾等原因，其值可能小于 rtNumberOfBytesToRead。如果 lpOverlapped 为 NULL，则该参数不能为 NULL。

lpOverlapped：输入参数，指向 OVERLAPPED 结构体的指针，如果调用 CreateFile 时设置了 FILE_FLAG_OVERLAPPED 标志，则需要使用该参数，否则可以为 NULL。

◇返回值

返回 BOOL 值，表示读文件是否成功。

◇使用说明

在读的过程中，文件指针会随着读操作的进行而自动移动，在循环调用本函数时，会顺序读出文件的内容，如果程序返回失败，可以使用 GetLastError 函数获取错误信息。

（3）WriteFile。

WriteFile 函数的功能是将数据写入到文件中，写入到文件指针所在的位置，写入操作完成后，文件指针会移动到写入的数据之后，函数原型如下：

```
BOOL WriteFile(
  HANDLE hFile,
  LPCVOID lpBuffer,
  DWORD nNumberOfBytesToWrite,
  LPDWORD lpNumberOfBytesWritten,
  LPOVERLAPPED lpOverlapped
);
```

◇参数

hFile:输入参数，写入数据的文件对象，由 CreateFile 创建，调用 CreateFile 打开文件时，需要指明 GENERIC_WRITE 读取操作模式。

lpBuffer：输入参数，指向需写入文件数据存储的内存缓冲区。

nNumberOBytesToWrite：输入参数，指明需要写入文件中的数据的大小。

lpNumberOiBytesWritten：输出参数，指向存储真实写入的数据大小的变量，可能由于已经到达文件尾等原因，其值可能与 nNumberOfBytesToWrite 不同。如果 lpOverlapped 为 NULL，则该参数不能为 NULL。

lpOverlapped：输入参数，指向 OVERLAPPED 结构体的指针，如果调用 CreateFile 时设置了 FILE_FLAG_OVERIAPPED 标志，则需要使用该参数。可以为 NULL。

◇返回值

返回 BOOL 值，表示写文件是否成功。

◇使用说明

在读的过程中，文件指针会随着写操作的进行而移动，在循环调用本函数时，会按顺序写入文件内容。如果程序返回失败，可以使用 GetLastError 函数获取错误信息。

（4）GetFileSize、GetFileSizeEX.

GetFileSize、GetFileSizeEX 的功能是一致的，都是获取文件大小，函数原型分别如下。

```
DWORD GetFileSize(
  HANDLE hFile,
  LPDWORD lpFileSizeHigh
);
BOOL GetFileSizeEx(
  HANDLE hFile,
  PLARGE_INTEGER lpFileSize
);
```

◇参数

GetFileSize 的参数如下。

hFile:输入参数，读取数据的文件对象，由 CreateFile 创建，调用 CreateFile 打开文件时需要指明 GENERIC_READ 读取操作模式或者 GENERIC_WRITE 写入操作模式。

lpFileSizeHigh：输出参数，表示得到的文件大小的高 32 位。该参数可以为 NULL，为 NULL 时表示文件大小可以用 DWORD 表示。具体解释请参考返回值部分。

GetFileSizeEx 的参数如下。

hFile:输入参数，读取数据的文件对象，由 CreateFile 创建，调用 CreateFile 打开文件时，需要指明 GENERIC_READ 读取操作模式或者 GENERIC_WRITE 写入操作模式。

lpFileSize：输出参数，指向储存文件大小的一个 LARGE_ INTEGER 联合体，前面已经介绍过了 LARGE. 1NTEGER 结构的相关内容，参见 4.2.4 小节关键数据结构部分。

◇返回值

GetFileSize 成功时，返回值为一个表示文件大小 DWORD 值。

GetFileSize 失败时，如果 lpFileSizeHigh 为 NULL，返回 INVALID_ FILE_SIZE 并可以调用 GetLastError 函数获取更详细的错误信息；如果 lpFileSizeHigh 为非 NULL，那么同样返回 INVALID_FILE_SIZE 并且尝试读取 lpFileSizeHigh 指向的值放在高 32 位，并将其和返回的低 32 位 DWORD 连起来成为一个 64 位的值来表示文件大小。lpFileSizeHigh 为合法值则 GetLastError 返回 NO_ERROR，表示该 API 仍然成功；否则，GetLastError 返回一个不等于 NO_ERROR 的值。

GetFileSizeEx 直接返回 BOOL 值，表示读文件是否成功。

◇使用说明

实际上这两个函数的区别就在于，对文件大小超出 DWORD 的情况采取不同处理方式。由于历史原因，GetFileSize 用两个 32 位值来分别储存 64 位文件大小的高位和地位；而 GetFileSizeEx 直接将文件大小储存在 64 位的联合体中。对文件大小储存的详细介绍可以参见 4.3.7 小节。

2．实例 4-7 创建、打开、读写文件演示实例

本实例使用 CreateFile、ReadFile、WriteFile 来完成创建、打开、读写文件的功能。笔者在实例中自己编写一个 SaveDataToFile 函数在 C 盘根目录下建立一个 show.txt 并向文件中写入数据，并编写了一个 ReadFileContent 函数读取文件的内容。

```
/* ************************************
* wr.c 创建、打开、读写文件，获取文件大小
**************************************/
/* 头文件  */
#include <windows.h>
#include <stdio.h>

/* ************************************
* DWORD ReadFileContent(LPSTR szFilePath)
* 功能：获取文件大小
*       读取文件内容，并以十六进制的形式打印出来
* 参数：LPSTR szFilePath
*         文件路径
**************************************/
DWORD ReadFileContent(LPSTR szFilePath)
{
     //文件大小
     HANDLE hFileRead;
     //保存文件大小
     LARGE_INTEGER liFileSize;
     //成功读取的文件数据大小
     DWORD dwReadedSize;
     //累加计算已经读取数据的大小
     LONGLONG liTotalRead = 0;
     //文件数据缓存
     BYTE lpFileDataBuffer[32];

     //打开已经存在的文件，读取内容
     hFileRead = CreateFile(szFilePath,   // 要打开的文件名
          GENERIC_READ,                   // 以读方式打开
          FILE_SHARE_READ,                // 可共享读
          NULL,                           // 默认安全设置
          OPEN_EXISTING,                  // 只打开已经存在的文件
          FILE_ATTRIBUTE_NORMAL,          // 常规文件属性
          NULL);                          // 无模板

     //打开文件是否成功
     if(hFileRead==INVALID_HANDLE_VALUE)
     {
          printf("打开文件失败：%d",GetLastError());
     }
     if(!GetFileSizeEx(hFileRead,&liFileSize))
     {
          printf("获取文件大小失败：%d",GetLastError());
     }
     else
     {
          printf("文件大小为：%d\n",liFileSize.QuadPart);
     }
     //循环读取并打印文件内容
     while(TRUE)
     {
          DWORD i;
          if(!ReadFile(hFileRead,          //读文件的句柄
               lpFileDataBuffer,           //存储读取的文件内容
               32,                         //读的大小（字节）
               &dwReadedSize,              //实际读取的大小
               NULL))                      //不使用 Overlapped
```

```
            {
                printf("读文件错误: %d\n",GetLastError());
                break;
            }
            printf("读取了%d字节, 文件内容是: ",dwReadedSize);

            for(i=0; i<dwReadedSize; i++)
            {
                printf("0x%x ",lpFileDataBuffer[i]);
            }
            printf("\n");
            liTotalRead += dwReadedSize;
            if(liTotalRead == liFileSize.QuadPart)
            {
                printf("读文件结束\n");
                break;
            }
        }
        CloseHandle(hFileRead);
        return 0;
}

/* ************************************
*  SaveDataToFile
* 功能: 将数据存储到文件末尾
* 参数: LPSTR szFilePath      文件路径
*       LPVOID lpData         需存储的数据
*       DWORD dwDataSize      数据大小(字节)
**************************************/
DWORD SaveDataToFile(
                        LPSTR szFilePath,
                        LPVOID lpData,
                        DWORD dwDataSize)
{
        //文件句柄
        HANDLE hFileWrite;
        //成功写入的数据大小
        DWORD dwWritedDateSize;
        //打开已经存在的文件, 读取内容
        hFileWrite = CreateFile(szFilePath,  // 要打开的文件名
            GENERIC_WRITE,                   // 以写方式打开
            0,                               // 可共享读
            NULL,                            // 默认安全设置
            OPEN_ALWAYS,                     // 打开已经存在的文件, 没有则创建
            FILE_ATTRIBUTE_NORMAL,           // 常规文件属性
            NULL);                           // 无模板

        //判断是否打开成功
        if(hFileWrite==INVALID_HANDLE_VALUE)
        {
            printf("打开文件失败: %d\n",GetLastError());
        }
        //设置文件指针到文件尾
        SetFilePointer(hFileWrite,0,0,FILE_END);
        //将数据写入文件
        if(!WriteFile(hFileWrite,lpData,dwDataSize,&dwWritedDateSize,NULL))
            printf("写文件失败: %d\n",GetLastError());
        else
            printf("写文件成功, 写入%d字节。\n",dwWritedDateSize);
        CloseHandle(hFileWrite);
        return 0;
}
```

```
/* ************************************
* int main(void)
* 功能：演示使用 SaveDataToFile 和 ReadFileContent 函数
************************************/
int main(void)
{
     LPSTR szFileData = "这是一个例子";
     SaveDataToFile("C:\\show.txt",szFileData,lstrlen(szFileData));
     ReadFileContent("C:\\show.txt");
     return 0;
}
```

下面将演示本示例程序的使用。编译生成 wr.exe。

运行“wr.exe”，正常情况下程序会打印输出：

```
写文件成功，写入 12 字节。
文件大小为：24
读取了 24 字节，文件内容是：0xd5 0xe2 0xca 0xc7 0xd2 0xbb 0xb8 0xf6 0xc0 0xfd 0xd7
 0xd3 0xd5 0xe2 0xca 0xc7 0xd2 0xbb 0xb8 0xf6 0xc0 0xfd 0xd7 0xd3
读文件结束
```

打开 C 盘根目录，可以找到一个名为“show.txt”的文件，内容为“这是一个例子”。

4.3.3 创建目录

编程实现创建目录是非常简单的，只要使用 API 函数 CreateDirectory 即可。

1. 关键 API

（1） Createdirectory

函数原型如下：

```
BOOL CreateDirectory(
  LPCTSTR lpPathName,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

◇参数

lpPathName:输入参数，所要创建的目录名或路径。

lpSecurityAttributes:输入参数，设置为 NULL。

◇返回值

返回 BOOL 值，表示是否成功。

◇使用说明

如果程序返回失败，可以使用 GetLastError 函数获取错误信息。可能的值包括 ERROR ALREADY EXISTS（文件夹已经存在）和 ERROR PATH NOT FOUND（路径不存在）。

2. 实例 4-8 使用 CreateDirectory 创建目录

本实例使用 CreateDirectory 函数在程序当前目录下创建一个“sub_dir”子目录，在 C 盘下创建一个“example_dir”。

```
/* ************************************
* dir.c 创建目录
************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
/* ************************************
* int main(void)
```

```
* 功能：演示使用 CreateDirectory 创建目录
*************************************/
int main(void)
{
    //在程序的当前目录下创建“sub_dir”子目录
    LPSTR szDirPath = "sub_dir";
    if (!CreateDirectory(szDirPath, NULL))
    {
        printf("创建目录 %s 错误。\n",szDirPath);
        return 1;
    }

    //在 C 盘下创建目录“example_dir”
    szDirPath = "C:\\example_dir";
    if (!CreateDirectory(szDirPath, NULL))
    {
        printf("创建目录 %s 错误。\n",szDirPath);
        return 1;
    }
    printf("成功\n");

    return 0;
}
```

运行上面的程序，会在程序所在的目录创建文件夹“sub_dir,,，还会在 C 盘目录下创建文件夹“example_dir”。如果文件夹已经存在，则创建失败。

4.3.4 获取程序所在的目录、程序模块路径，获取和设置当前目录

Windows 系统提供一组 API 实现对程序运行时相关目录的获取和设置。用户可以使用 GetCurrentDirectory 和 SetCurrentDirectory 获取程序的当前目录，获取模块的路径使用 GetModuleFileName，如果以 NULL 参数调用 GetModuleFileName，将会返回当前模块的路径。如果在程序主模块(exe)中获取当前模块路径，便可以从当前模块的路径中提取出程序运行时所在的路径。

1．关键 API

（1）GetCurrentDirectory。

获取进程的当前目录，函数原型如下：

```
DWORD GetCurrentDirectory(
  DWORD nBufferLength,
  LPTSTR lpBuffer
);
```

◇参数

nBufferLength：输入参数，存储路径字符串缓冲区的大小，一般调用 MAX PATH。

lpBuffer：输出参数，指向获取的路径字符串。

◇返回值

返回 DWORD 值，如果为 0，表示执行失败，可以使用 GetLastError 获取错误信息；如果为非 0，则获取的路径字符串长度，函数执行成功。

（2）SetCurrentDirectory。

设置进程的当前目录，函数原型如下：

```
BOOL SetCurrentDirectory(
  LPCTSTR lpPathName
);
```

◇参数

lpPathName:输入参数，所要设置的路径值。

◇返回值

返回 DWORD 值，如果为 0，表示执行失败，可以使用 GetLastError 获取错误信息；如果为非 0，则执行成功。

(3)GetModuleFileName。

获取模块文件名，当第一个参数为 NULL 时获取当前模块路径，函数原型如下：

```
DWORD GetModuleFileName(
  HMODULE hModule,
  LPTSTR lpFilename,
  DWORD nSize
);
```

◇参数

hModule:输入参数，所要获取模块路径的模块句柄。

lpFilename:输出参数，模块的全路径。

nSize:输入参数，lpFilename 所指向的缓冲区的大小。

◇返回值

返回 DWORD 值，如果为 0，表示执行失败；如果为非 0，则执行成功，表示获取的路径的字符串长度。

◇使用说明

可以使用 GetLastError 获取错误信息，如果返回值为 nSize 指明的大小，那么很有能是因为缓冲区太小，这时使用 GetLastError，会得到 ERROR_INSUFFICIENT_- BUFFER。

2．实例 4-9 获取、设置进程或模块路径

本实例首先获取并打印出程序的当前路径，如果程序的当前路径没有经过设置，默认情况下将是程序运行时所在的目录。然后将当前目录设置为“C:\”，设置完成后，使用相对路径创建的目录，目的是验证相对路径是相对于进程的当前路径的，而不是可执行文件所在的路径。然后使用 GetModuleFileName 获取了本模块和 kerne132.exe 的路径。

```
/* ************************************
* cur_mod_dir.c 获取当前目录、获取程序所在的目录、获取模块路径
**************************************/
#include <windows.h>
#include <stdio.h>
/* ************************************
* int main(void)
* 功能: 演示使用设置获取当前路径
*       演示获取模块路径
**************************************/
int main(void)
{
    //用于存储当前路径
    CHAR szCurrentDirectory[MAX_PATH];
    //用于存储模块路径
    CHAR szMoudlePath[MAX_PATH];
    //Kernel32 文件名与句柄
    LPSTR szKernel32 = "kernel32.dll";
    HMODULE hKernel32;
    //当前路径的长度，也用于判断获取是否成功
    DWORD dwCurDirPathLen;
```

```
    //获取进程当前目录
    dwCurDirPathLen =
         GetCurrentDirectory(MAX_PATH,szCurrentDirectory);
    if(dwCurDirPathLen == 0)
    {
          printf("获取当前目录错误。\n");
          return 0;
    }
    printf("进程当前目录为 %s \n",szCurrentDirectory);

    //将进程当前目录设置为 "C:\"
    lstrcpy(szCurrentDirectory, "C:\\");
    if(!SetCurrentDirectory(szCurrentDirectory))
    {
           printf("设置当前目录错误。\n");
           return 0;
    }
    printf("已经设置当前目录为 %s \n",szCurrentDirectory);

    //在当前目录下创建子目录 "current_dir"
    //运行完成后 C:盘下将出现文件夹 "current_dir"
    CreateDirectory("current_dir", NULL);

    //再次获取系统当前目录
    dwCurDirPathLen =
         GetCurrentDirectory(MAX_PATH,szCurrentDirectory);
    if(dwCurDirPathLen == 0)
    {
         printf("获取当前目录错误。\n");
         return 0;
    }
    printf("GetCurrentDirectory 获取当前目录为 %s \n",
          szCurrentDirectory);

    //使用 NULL 参数，获取本模块的路径
    if(!GetModuleFileName(NULL,szMoudlePath,MAX_PATH))
    {
         printf("获取模块路径录错误。\n");
         return 0;
    }
    printf("本模块路径 %s \n",szMoudlePath);

    //获取 Kernel32.dll 的模块句柄
    hKernel32 = LoadLibrary(szKernel32);

    //使用 Kernel32.dll 的模块句柄，获取其路径
    if(!GetModuleFileName(hKernel32,szMoudlePath,MAX_PATH))
    {
           printf("获取模块路径错误。\n");
           return 0;
    }
    printf("kernel32 模块路径 %s \n",szMoudlePath);

    return 0;
}
```

运行本程序，运行结果如下：

```
进程当前目录为 E:\\code\ch4\ch4_3\debug
已经设置当前目录为 C:\
GetCurrentDirectory 获取当前目录为 C:\
本模块路径 E:\\code\ch4\ch4_3\debug\4_cur_dir.exe
kernel32 模块路径 C:\WINDOWS\system32\kernel32.dll
```

同时查看 C 盘，将会出现一个“current_dir”目录。

通过本实例读者需了解的注意事项如下。

◇在进程中使用相对路径，则相对路径的起始点是程序的当前路径而不是可执行文件所在的路径。

◇进程的当前路径在默认情况下是应用程序可执行文件所在的路径。

◇模块路径与程序的当前路径是两个概念，进程的主程序和进程中所加载的所有 DLL 都是进程的模块。

4.3.5 查找文件、遍历指定目录下的文件和子目录

Windows API 中，有一组专门的函数和结构，用于遍历目录，它们是 FindFirstFile 函数、FindNextFile 函数和 WIN32_FIND_DATA 结构。使用 FindFirstFile 和 FindNextFile 函数并与 do-while 循环结合，可以完成遍历目录的任务，详见实例 4-10。

值得一提的是，FindFirstFile 输入参数的路径需使用通配符，也就是用户可以根据一些条件来对查找的文件作简单的过滤。实例 4-10 讲解查找特定目录下的所有文件和文件夹。读者可根据自己的需要，指定查找文件的条件。

1．关键 API

（1）FindFirstFile。

查找第一个目录或文件，获取查找句柄，函数原型如下：

```
BOOL FindNextFile(
  HANDLE hFindFile,
  LPWIN32_FIND_DATA lpFindFileData
);
```

◇参数

lpFileName：输入参数，查找的目录，需使用通配符指定查找的文件目标。

lpFindFileData:输出参数，指向 WIN32_FIND_DATA 结构的指针，为找到的文件及其若干属性信息。

◇返回值

返回 HANDLE 值，如果执行失败，返回 1NVALID HANDLE VALUE，如果成功则返回查找句柄。

◇使用说明

如果查找文件需在循环中与 FindNextFile 配合使用，FindNextFile 需要用到 FindFirstFile 返回的句柄

（2）FindNextFile

对文件、文件夹进行循环查找，函数原型如下：

```
HANDLE FindFirstFile(
  LPCTSTR lpFileName,
  LPWIN32_FIND_DATA lpFindFileData
);
```

◇参数

hFindFile:输入参数，查找句柄，由 FindFirstFile 返回。

lpFindFileData:输出参数，指向 WIN32_FIND_DATA 结构的指针，为找到的文件及其若干属性信息。

◇返回值

返回 BOOL 值，表示是否成功。

◇使用说明

需循环查找。

2．关键结构

WIN32_FIND_DATA 结构用于表示找到的文件，结构中包括文件、目录的名字，创建、最后访问和最后写入时间，文件大小、文件属性等。

```
typedef struct _WIN32_FIND_DATA {
  DWORD dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD nFileSizeHigh;
  DWORD nFileSizeLow;
  DWORD dwReserved0;
  DWORD dwReserved1;
  TCHAR cFileName[MAX_PATH];
  TCHAR cAlternateFileName[14];
} WIN32_FIND_DATA, *PWIN32_FIND_DATA, *LPWIN32_FIND_DATA;
```

3．实例 4-10 遍历指定目录

本实例实现了对指定目录中文件和子目录的遍历，并将遍历得到的文件和其他属性打印到界面上。

```
/* ************************************
* sub_dir.c 遍历目录下的文件和子目录
**************************************/
#include <windows.h>
#include <stdio.h>
/* ************************************
* DWORD EnumerateFileInDrectory(LPSTR szPath)
* 功能：遍历目录下的文件和子目录，将显示
*       文件和文件夹隐藏、加密的属性
* 参数：LPTSTR szPath，为需遍历的路径
* 返回值：0 代表执行完成，1 代码发生错误
**************************************/
DWORD EnumerateFileInDrectory(LPSTR szPath)
{
     WIN32_FIND_DATA FindFileData;
     HANDLE hListFile;
     CHAR szFilePath[MAX_PATH];

     //构造代表子目录和文件夹路径的字符串，使用通配符“*”
     lstrcpy(szFilePath, szPath);
     //注释的代码可以用于查找所有以“.txt”结尾的文件
     //lstrcat(szFilePath, "\\*.txt");
     lstrcat(szFilePath, "\\*");
     //查找第一个文件/目录，获得查找句柄
     hListFile = FindFirstFile(szFilePath,&FindFileData);
     //判断句柄
     if(hListFile==INVALID_HANDLE_VALUE)
     {
          printf("错误：%d",GetLastError());
          return 1;
     }
     else
     {
         do
```

```
            {
                /* 如果不想显示代表本级目录和上级目录的“.”和“..”，
                   可以使用注释部分的代码过滤
                if(lstrcmp(FindFileData.cFileName,TEXT("."))==0||
                      lstrcmp(FindFileData.cFileName,TEXT(".."))==0)
                {
                      continue;
                }
                */
                //打印文件名、目录名
                printf("%s\t\t",FindFileData.cFileName);
                //判断文件属性，是否为加密文件或文件夹
                if(FindFileData.dwFileAttributes&FILE_ATTRIBUTE_ENCRYPTED)
                {
                     printf("<加密> ");
                }
                //判断文件属性，是否为隐藏文件或文件夹
                if(FindFileData.dwFileAttributes&FILE_ATTRIBUTE_HIDDEN)
                {
                     printf("<隐藏> ");
                }
                //判断文件属性，是否为目录
                if(FindFileData.dwFileAttributes&FILE_ATTRIBUTE_DIRECTORY)
                {
                     printf("<DIR> ");
                }
                //读者可根据文件属性表中的内容自行添加、判断文件属性
                printf("\n");
            }
            while(FindNextFile(hListFile, &FindFileData));
        }
        return 0;
}

/* ************************************
* int main(int argc, PCHAR argv[])
* 功能：调用 ListFileInDrectory
*       遍历目录下的文件和子目录
* 参数：argv[1]为需遍历的路径，如果为空则获取
*       当前路径
**************************************/
int main(int argc, PCHAR argv[])
{
     if(argc == 2)
     {
          EnumerateFileInDrectory(argv[1]);
     }
     else
     {
          CHAR szCurrentPath[MAX_PATH];
          GetCurrentDirectory(MAX_PATH,szCurrentPath);
          EnumerateFileInDrectory(szCurrentPath);
     }
     return 0;
}
```

在命令行中运行“sub_dir.exe”，会遍历 exe 文件所在目录，结果为：

```
.            <DIR>
..           <DIR>
.svn         <隐藏> <DIR>
sub_dir.exe
sub_dir.obj
```

如果以指定目录为参数运行，可以得到指定目标下的所有子文件和文件夹，比如在笔者主机上运行，以“F:WMware”为参数输入：

```
sub_dir.exe F:\VMware
```

本文件夹中有两个子文件夹分别是“Ubuntu”和“Windows XP Professional”，运行

后得到以下结果：

```
.              <DIR>
..             <DIR>
Ubuntu          <DIR>
Windows XP Professional        <DIR>
```

如果指定目录不存在或者参数是错误的，则输出 GetLastError 的返回值：

```
错误: 3
```

这里的 3 代表了系统找不到指定路径。

4.3.6 递归遍历目录树

在实例 4-10 的基础上稍加改造，进行循环递归调用，采用树形结构深度遍历的方法。可以遍历指定目录中的所有文件、包括子目录中的文件。代码实现如实例 4-11 所示。

实例 4-11 递归调用遍历目录树

本实例采用了树的深度遍历算法。

```
/* ************************************
* tree.c 递归遍历目录树
**************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
/* 预处理申明*/
#pragma comment (lib, "User32.lib")
/* 函数申明 */
DWORD ListAllFileInDrectory(LPSTR szPath);
/* 全局变量 */
//记录所有的文件和目录数
DWORD dwTotalFileNum = 0;
/* ************************************
* DWORD ListAllFileInDrectory(LPSTR szPath)
* 功能: 遍历目录及所有子目录, 打印路径
* 参数: LPTSTR szPath, 为需遍历的目录
* 返回值: 0 代表执行完成, 代码发生错误
**************************************/
DWORD ListAllFileInDrectory(LPSTR szPath)
{
    CHAR szFilePath[MAX_PATH];
    WIN32_FIND_DATA FindFileData;
    HANDLE hListFile;
    CHAR szFullPath[MAX_PATH];
    //构造代表子目录和文件夹路径的字符串, 使用通配符"*"
    lstrcpy(szFilePath, szPath);
    lstrcat(szFilePath, "\\*");
    //查找第一个文件目录, 获得查找句柄
    hListFile = FindFirstFile(szFilePath,&FindFileData);
    if(hListFile==INVALID_HANDLE_VALUE)
    {
        printf("错误: %d",GetLastError());
        return 1;
    }
```

```
    else
    {
        do
        {
            //   过滤"."和"..", 不需要遍历
            if(lstrcmp(FindFileData.cFileName,TEXT("."))==0||
                lstrcmp(FindFileData.cFileName,TEXT(".."))==0)
            {
                continue;
            }
            //构造成全路径
            wsprintf(szFullPath,"%s\\%s",
                szPath,FindFileData.cFileName);
            dwTotalFileNum++;
            //打印
            printf("\n%d\t%s\t",dwTotalFileNum,szFullPath);
            //如果是目录, 则递归调用, 列举下级目录
            if(FindFileData.dwFileAttributes&FILE_ATTRIBUTE_DIRECTORY)
            {
                printf("<DIR>");
                ListAllFileInDrectory(szFullPath);
            }
        }
        while(FindNextFile(hListFile, &FindFileData));
    }
    return 0;
}
/* ************************************
* int main(int argc, PCHAR argv[])
* 功能: 调用 ListAllFileInDrectory
*       遍历目录下的文件和子目录
* 参数: argv[1]为需遍历的路径, 如果为空则获取
*       当前路径
**************************************/
int main(int argc, PCHAR argv[])
{
    if(argc == 2)
    {
        ListAllFileInDrectory(argv[1]);
    }
    else
    {
        CHAR szCurrentPath[MAX_PATH];
        GetCurrentDirectory(MAX_PATH,szCurrentPath);
        ListAllFileInDrectory(szCurrentPath);
    }
    return 0;
}
```

例如在笔者主机上运行：

```
tree.exe E:\code\ch2
```

会得到如下结果：

```
1   E:\code\ch2\basic  <DIR>
2   E:\code\ch2\basic\basic.c
3   E:\code\ch2\basic\basic.vcproj
4   E:\code\ch2\ch2.sln
5   E:\code\ch2\ch2.suo
6   E:\code\ch2\DataType   <DIR>
7   E:\code\ch2\DataType\DataType.vcproj
8   E:\code\ch2\DataType\windata.c
9   E:\code\ch2\StringCode <DIR>
10  E:\code\ch2\StringCode\Makefile
```

```
11  E:\code\ch2\StringCode\StringCode.c
12  E:\code\ch2\StringCode\StringCode.vcproj
13  E:\code\ch2\ui     <DIR>
14  E:\code\ch2\ui\ui.vcproj
15  E:\code\ch2\ui\window.cpp
```

4.3.7 获取、设置文件属性和时间

为了获取文件属性，用户可以使用 GetFileAttributes 与 GetFileAttributesEx 函数。GetFileAttributesEx 函数除了返回文件属性外，还返回文件时间信息、文件大小等。GetFileAttributesEx 将返回结果保存在 WIN32_FILE_ATTRIBUTE DATA 结构中。

获取的文件时间是以 FILETIME 格式存在的，如果要正确显示，还需要对其时区进行调整，调整为本地时区，然后转换为系统时间格式，便于显示。

前面在获取文件大小时已经介绍，NTFS 文件系统使用了 64 位数据来表示文件大小。因为 32 位的数据最多只能表示 4GB 的大小。Windows 将其分为了高 32 位和低 32 位，两个都需要使用到，这一点尤其要在对大于 4GB 的文件操作时注意。

1．关键 API

(1)GetFileAttributeS。

获取文件或目录的属牲-函数原型如下：

```
BOOL GetFileAttributesEx(
  LPCTSTR lpFileName,
  GET_FILEEX_INFO_LEVELS fInfoLevelId,
  LPVOID lpFileInformation
);
```

◇参数

lpFileName:输入参数，为需要获取属性的文件或目录。　　、

◇返回值

返回 DWORD 值，表示文件属性。如果返回 INVALID_FILE ATTRIBUTES，则表示失败。可使用 GetLastError 函数获取错误信息。

◇使用说明

要判断文件具体有哪些属性，需要使用“&”与属性常量进行运算，如果运行结果为真，则表示具有这种属性。

（2）GetFileAttributesEx。

获取文件或目录的属性、时间、大小，以 WIN32_FILE ATTRIBUTE_DATA 结构的形式返回结果，函数原型如下：

```
DWORD GetFileAttributes(
  LPCTSTR lpFileName
);
```

◇参数

lpFileName:输入参数，为需要获取属性的文件或目录。

flnfoLevelId:输入参数，是获取文件属性信息的类别，会影响到 lpFileInformation 参数具体采用什么形式输出结果。但是本参数一般只能指定为 GetFileExInfoStandard。

lpFileInformation：输出参数，用于返回结果。如果 flnfoLevelId 参数指定为 GetFileExInfoStandard，则为 LPWIN32_FILE_ATTRIBUTE_ DATA 类型。

◇返回值

返回 BOOL 值，表示是否成功。

(3)　SetFileAttributes.

设置文件或目录的属性，函数原型如下：

```
BOOL FileTimeToSystemTime(
  const FILETIME* lpFileTime,
  LPSYSTEMTIME lpSystemTime
);
```

◇参数

lpFileName：输入参数，为需要设置属性的文件或目录。

dwFileAttributes:文件属性值。

◇返回值

返回 BOOL 值，表示是否成功。

◇使用说明

文件系统中对文件属性的表示使用了 DWORD 类型的数据，多个文件属性使用“【”运算连接在一起，详见实例 4-13.

（4）FileTimeToLocalFileTime。

把文件时间转换为本地的文件时间，函数原型如下：

```
BOOL FileTimeToLocalFileTime(
  const FILETIME* lpFileTime,
  LPFILETIME lpLocalFileTime
);
```

◇参数

lpFileTime：输入参数，为需要转换的文件时间。

lpLocalFileTime：输出参数，为指向转换后的文件时间。

◇返回值

返回 BOOL 值，表示是否成功。

（5）FileTimeToSystemTime

将文件时间转换为系统时间（SYSTEMTIME 格式），便于显示，函数原型如下：

```
BOOL SetFileAttributes(
  LPCTSTR lpFileName,
  DWORD dwFileAttributes
);
```

◇参数

lpFileTime,：输入参数，指向需要转换的文件时间。

lpSystemTime：输出参数，指向转换后的 SYSTEMTIME。

◇返回值

返回 BOOL 值，表示是否成功。

2．关键数据结构

(1) FILETIM。

此结构用最小的数据量表示的时间，但是不便于用户查看和显示。通过 API 获取的系统时间都是这种格式的。如果要使用显示，可以使用 FileTimeToSystemTime 转换为便于显示的 SYSTEMTIME 结构。

```
typedef struct _FILETIME {
  DWORD dwLowDateTime;
  DWORD dwHighDateTime;
} FILETIME, *PFILETIME;
```

(2) SYSTEMTIME。

此结构使用了较为直观的方式表示时间。

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

(3) WIN32_FILE ATTRIBUTE_DATA。

GetFileAttributesEx 使用这个结构表示返回结果，包括文件属性、文件创建时间、文件最后访问时间、文件最后写入时间和文件大小。

```
typedef struct _WIN32_FILE_ATTRIBUTE_DATA {
  DWORD dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD nFileSizeHigh;
  DWORD nFileSizeLow;
} WIN32_FILE_ATTRIBUTE_DATA, *LPWIN32_FILE_ATTRIBUTE_DATA;
```

3．实例 4-12 获取设置文件属性

本实例编译完成后是一个可执行文件，运行时需输入两个参数，均是文件路径，程序将第一个参数所指向的文件和目录的属性显示出来，将第二个参数所指向的文件或目录设置为隐藏和只读。

main 函数如下：

```
/* ************************************
 * int main(int argc, PCHAR argv[])
 * 功能：设置和获取文件属性等
 * 参数：显示第一个参数指定文件的属性、时间、大小
 *       将第二个参数的属性设置为隐藏、只读
 * 返回值：0 代表执行完成，1 代表发生错误
 ************************************/
int main(int argc, PCHAR argv[])
{
    if(argc != 3)
    {
        printf("请输入参数\n");
        printf("显示第一个参数指定文件的属性、时间、大小；\n");
        printf("将第二个参数的属性设置为隐藏、只读。");
        return 1;
    }
    ShowFileAttributes(argv[1]);
    SetFileHiddenAndReadonly(argv[2]);
    return 0;
}
```

main 函数调用了 ShowFileAttributes 函数，获取文件属性，并将其显示出来。

```
/* ************************************
```

```
 * DWORD ShowFileAttributes(LPSTR szPath)
 * 功能：获取并显示文件属性，
 *       调用 ShowFileTime、ShowFileSize 和
 *       ShowFileAttrInfo 函数
 * 参数：LPTSTR szPath，获取并显示此文件的属性
 * 返回值：0 代表执行完成，1 代表发生错误
 ************************************/
DWORD ShowFileAttributes(LPSTR szPath)
{
     //文件属性结构
     WIN32_FILE_ATTRIBUTE_DATA wfad;
     printf("文件：%s\n",szPath);
     //获取文件属性
     if(!GetFileAttributesEx(szPath,
           GetFileExInfoStandard,
           &wfad))
     {
           printf("获取文件属性错误：%d\n",GetLastError());
           return 1;
     }
     //显示相关时间
     printf("创建时间：\t");
     ShowFileTime(&(wfad.ftCreationTime));
     printf("最后访问时间：\t");
     ShowFileTime(&(wfad.ftLastAccessTime));
     printf("最后修改时间：\t");
     ShowFileTime(&(wfad.ftLastWriteTime));
     //显示文件大小
     ShowFileSize(wfad.nFileSizeHigh,wfad.nFileSizeLow);
     //显示文件属性
     ShowFileAttrInfo(wfad.dwFileAttributes);

     return 0;
}
```

显示文件时间时使用了 ShowFileTime 函数，ShowFileTime 函数中将文件时间转换为便于显示的格式并显示出来。

```
/* ************************************
 *DWORD ShowFileTime(PFILETIME lptime)
 * 功能：转换文件时间并打印
 * 参数：PFILETIME lptime，指向文件时间的指针
 * 返回值：0
 ************************************/
DWORD ShowFileTime(PFILETIME lptime)
{
    //文件时间结构
    FILETIME ftLocal;
    //系统时间结构
    SYSTEMTIME st;
    //调整为系统所在时区的时间
    FileTimeToLocalFileTime(
         lptime,
         &ftLocal
         );
    //将文件时间转换为 SYSTEMTIME 格式，便于显示
    FileTimeToSystemTime(
          &ftLocal,
          &st
         );
     //显示时间信息字符串
     printf("%4d 年%#02d 月%#02d 日，%#02d:%#02d:%#02d\n",
           st.wYear,st.wMonth,st.wDay,st.wHour,st.wMinute,st.wSecond);
     return 0;
}
```

显示文件大小，使用了下面的函数，在函数中将文件大小的高位和低位连接为 64 位的长整型。

```
/* ************************************
 * DWORD ShowFileSize(DWORD dwFileSizeHigh, DWORD dwFileSizeLow)
 * 功能：打印文件大小信息
 * 参数：DWORD dwFileSizeHigh，文件大小高 32 位
 *       DWORD dwFileSizeLow，文件大小低 32 位
 ************************************/
DWORD ShowFileSize(DWORD dwFileSizeHigh, DWORD dwFileSizeLow)
{
    ULONGLONG liFileSize;
    liFileSize = dwFileSizeHigh;
    //移动到 32 位
    liFileSize <<= sizeof(DWORD)*8;
    liFileSize += dwFileSizeLow;
    printf("文件大小：\t%I64u 字节\n",liFileSize);
    return 0;
}
```

显示文件属性使用了 ShowFileAttrInfo 函数。在函数中使用“&”运算与文件属性恒量进行运算，根据是否为真可以判断文件是否具有特定的属性。

```
/* ************************************
 * DWORD ShowFileAttrInfo(DWORD dwAttribute)
 * 功能：打印文件属性
 * 参数：DWORD dwAttribute，文件属性
 ************************************/
DWORD ShowFileAttrInfo(DWORD dwAttribute)
{
    //依次判断属性，并显示
    printf("文件属性：\t");
    if(dwAttribute&FILE_ATTRIBUTE_ARCHIVE)
        printf("<ARCHIVE> ");
    if(dwAttribute&FILE_ATTRIBUTE_COMPRESSED)
        printf("<压缩> ");
    if(dwAttribute&FILE_ATTRIBUTE_DIRECTORY)
        printf("<目录> ");
    if(dwAttribute&FILE_ATTRIBUTE_ENCRYPTED)
        printf("<加密> ");
    if(dwAttribute&FILE_ATTRIBUTE_HIDDEN)
        printf("<隐藏> ");
    if(dwAttribute&FILE_ATTRIBUTE_NORMAL)
        printf("<NORMAL> ");
    if(dwAttribute&FILE_ATTRIBUTE_OFFLINE)
        printf("<OFFLINE> ");
    if(dwAttribute&FILE_ATTRIBUTE_READONLY)
        printf("<只读> ");
    if(dwAttribute&FILE_ATTRIBUTE_SPARSE_FILE)
        printf("<SPARSE> ");
    if(dwAttribute&FILE_ATTRIBUTE_SYSTEM)
        printf("<系统文件> ");
    if(dwAttribute&FILE_ATTRIBUTE_TEMPORARY)
        printf("<临时文件> ");
    printf("\n");
    return 0;
}
```

在 main 函数中，还设置了文件属性，在函数 SetFileHiddenAndReadonly 中进行了操作，先获取文件属性，然后使用“1”运算将文件只读和隐藏属性附加到原来的属件上。

```
/* ************************************
 * DWORD SetFileHiddenAndReadonly(LPSTR szFileName)
 * 功能：将指定的文件设置为隐藏和只读
 * 参数：LPSTR szFileName，文件路径
 ************************************/
```

```
DWORD SetFileHiddenAndReadonly(LPSTR szFileName)
{
    //获取原来的文件属性
    DWORD dwFileAttributes = GetFileAttributes(szFileName);
    //将只读和隐藏属性附加到原来的文件属性上
    dwFileAttributes |= FILE_ATTRIBUTE_READONLY;
    dwFileAttributes |= FILE_ATTRIBUTE_HIDDEN;
    //设置文件属性，并判断是否成功
    if(SetFileAttributes(szFileName, dwFileAttributes))
    {
          printf("文件%s 的隐藏和属性设置成功\n",szFileName);
    }
    else
    {
           printf("属性设置; %d\n",GetLastError());
    }
    return 0;
}
```

编译得到 FileAttr.exe，在同一目录中创建 1.txt、2.txt，将 1.txt 设置为只读，写入部分内容，在笔者主机上运行：

```
FileAttr.exe 1.txt 2.txt
```

运行结果如下：

文件：1.txt

```
创建时间:       2007 年 11 月 16 日, 11:20:32
最后访问时间:   2007 年 11 月 16 日, 11:22:57
最后修改时间:   2007 年 11 月 16 日, 11:20:51
文件大小:  4 字节
文件属性:  <ARCHIVE> <只读>
文件 2.txt 的隐藏和属性设置成功
```

右键查看文件 2.txt 属性，可以看到只读和隐藏，如图 4-7 所示。

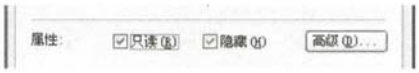

图 4-7　查看设置的文件属性

4.4　内存映射文件

本节介绍 Mapping File、文件句柄等较高级操作。本节将通过以下几个实例来讲解高级文件系统操作的 API。

◇　使用 Mapping File 提高文件读写的效率。

◇通过 Mapping File 在进程间共享内存。

◇通过文件句柄获得文件路径。

4.4.1　使用Mapping File提高文件读写的效率

文件映射(mapping)是一种在将文件内容映射到进程的虚拟地址空间的技术。视图(View)是一段虚拟地址空间，进程可以通过 View 来存取文件的内容，视图是一段内存，可以使用指针来操作视图。使用的文件映射之后，读写文件就如同对读写内存一样简单。在使用文件映射时需要创建映射对象，映射对象分为命名的和未命名的。映射对象还存取权限。

使用文件映射至少有 3 个好处，一是因为文件是存储于硬盘上的，而文件视图是一段内存，使用文件映射操作时更方便；二是效率更高；三是可以在不同的进程间共享数据。

文件映射依赖于系统虚拟内存管理的分页机制。

本节将演示如何使用文件映射，下一节将演示如何使用文件映射来进行内存共享。

1. 关键 API

（1）GetSystemInfo

获取系统信息，在实例 4-14 中用于获取系统内存分配粒度。有关内存分配粒度的概念参见第 5 章和虚拟内存管理相关章节。

（2）CreateFileMapping。

创建 mapping 对象，函数原型如下：

```
HANDLE CreateFileMapping(
  HANDLE hFile,
  LPSECURITY_ATTRIBUTES lpAttributes, NULL
  DWORD flProtect,
  DWORD dwMaximumSizeHigh,
  DWORD dwMaximumSizeLow,
  LPCTSTR lpName
);
```

◇参数

hFile：输入参数，由 CreateFile 创建的需要映射的文件。

lpAttributes：输入参数，指向 SECURITY ATTRIBUTES 结构的指针，此结构将在第 17 章中详细介绍。在一般的情况下，使用默认属性即可，将此参数赋为 NULL。

flProtect:输入参数，内存保护属性，可以是 PAGE READONLY、PAGE_READWRITE、PAGE_WRITECOPY、PAGE_EXECUTE_READ、PAGE_EXECUTE READWRITE 中的一种，也可以使用位或运算附加下面的属性：SECURITY ATTRIBUTES、SEC_COMMIT、SEC_ IMAGE、SEC_ LARGE_PAGES、SEC_NOCACHE、SEC—RESERVE.

dwMaximumSizeHigh：输入参数，映射大小的最大值的高 32 位。

dwMaximumSizeLow:输入参数，映射大小的最大值的低 32 位。

lpName：输入参数，映射对象名，可以是 NULL。

◇返回值

返回 HANDLE 值,mapping 对象的句柄。如果返回 NULL,则表示失败。可使用 GetLastError 函数获取错误信息。

（3）MapViewOfFile。

创建视图，将文件 mapping 映射到当前进程内存虚拟地址空间。函数原型如下：

```
LPVOID MapViewOfFile(
  HANDLE hFileMappingObject,
  DWORD dwDesiredAccess,
  DWORD dwFileOffsetHigh,
  DWORD dwFileOffsetLow,
  SIZE_T dwNumberOfBytesToMap
);
```

◇参数

hFileMappingObj ect：输入数据，文件 mapping 对象的句柄，由 CreateFileMapping 函数或 OpenFileMapping 函数返回。

dwDesiredAccess:输入参数，存取类别，可以是 FILE_MAP_WRITE. FILE_MAP_READ. FILE_MAP_COPY, FILE_MAP_EXECUTE 中的一种。

dwFileOffsetHigh:输入参数，映射的文件偏移的高 32 位。

dwFileOffsetLow:输入参数，映射的文件偏移的低 32 位。

dwNumberOfBytesToMap:输入参数，映射到 View 的字节数。

◇返回值

返回 LPVOID 值，指向映射的内存值，如果为 NULL 表示错误。

（4）FlushViewOfFile。

将视图中的文件数据写入到磁盘上。调用此参数后，对映射视图的内存操作将会及时反映到硬件中的文件。函数原型如下：

```
BOOL FlushViewOfFile(
  LPCVOID lpBaseAddress,
  SIZE_T dwNumberOfBytesToFlush
);
```

◇参数

lpBaseAddress:输入参数，需要写入至文件的数据的起始位置。

dwNumberOfBytesToFlush:输入参数，写入的字节数，如果为 0 则将整个视图都写回。

返回值

返回 BOOL 值，表示是否成功。

◇使用说明

如果不调用此函数，数据最终也会写回到硬盘，调用此函数后，数据会立刻写回到硬盘。

(5) FillMemory、CopyMemory。内存操作函数，分别为填充内存和复制内存，详见第 5 章。

2．实例 4-13 使用 MappingFile 提高文件读写效率

本实例首先创建一个 256KB 大小的文件，然后对文件进行映射，并通过文件视图、使用读写内存的方法来对文件内容进行操作。同时本实例还根据系统内存分配粒度的大小，对文件映射区域进行了对齐。

```
/* ************************************
* file_map.c 使用 Mapping File 提高文件读写的效率
*************************************/
/* 头文件  */
#include <windows.h>
#include <stdio.h>
/* 预处理声明  */
#define BUFFSIZE 1024                    // 内存大小
#define FILE_MAP_START 0x28804           // 文件映射的起始位置
/* 全局变量  */
LPTSTR lpcTheFile = TEXT("test.dat");    // 文件名
/* ************************************
* int main(void)
* 功能：演示使用文件 mapping
* 返回值：0 代表执行完成，1 代表发生错误
*************************************/
int main(void)
{
    HANDLE hMapFile;            // 文件内存映射区域的句柄
    HANDLE hFile;               // 文件的句柄
    DWORD dBytesWritten;        // 写入的字节数
    DWORD dwFileSize;           // 文件大小
    DWORD dwFileMapSize;        // 文件映射的大小
    DWORD dwMapViewSize;        // 视图（View）的大小
    DWORD dwFileMapStart;       // 文件映射视图的起始位置
    DWORD dwSysGran;            // 系统内存分配的粒度
```

```
SYSTEM_INFO SysInfo;          // 系统信息
LPVOID lpMapAddress;          // 内存映射区域的起始位置
PCHAR pData;                  // 数据
INT i;                        // 循环变量
INT iData;
INT iViewDelta;
BYTE cMapBuffer[32];  // 存储从mapping中计出的数据
// 创建一个文件
hFile = CreateFile(lpcTheFile,
      GENERIC_READ | GENERIC_WRITE,
      0,
      NULL,
      CREATE_ALWAYS,
      FILE_ATTRIBUTE_NORMAL,
      NULL);
//判断文件是否创建成功
if (hFile == INVALID_HANDLE_VALUE)
{
      printf("CreateFile error\n",GetLastError);
      return 1;
}
// 依次写入整数，一共写入65535个整数
// 在32位平台下，大小为65535*4字节（在32位平台下，一个整型数为4个字节）
for (i=0; i<65535; i++)
{
      WriteFile (hFile, &i, sizeof (i), &dBytesWritten, NULL);
}
// 查看写入完成后的文件大小
dwFileSize = GetFileSize(hFile,  NULL);
printf("文件大小: %d\n", dwFileSize);
//获取系统信息，内存分配粒度
//获取分配粒度，进行下面的几个计算
//目的是为了映射的数据与系统内存分配粒度对齐，提高内存访问效率
GetSystemInfo(&SysInfo);
dwSysGran = SysInfo.dwAllocationGranularity;
//计算mapping的起始位置
dwFileMapStart = (FILE_MAP_START / dwSysGran) * dwSysGran;
// 计算mapping view的大小
dwMapViewSize = (FILE_MAP_START % dwSysGran) + BUFFSIZE;
// 计算mapping的大小
dwFileMapSize = FILE_MAP_START + BUFFSIZE;
// 计算需要读取的数据的偏移
iViewDelta = FILE_MAP_START - dwFileMapStart;
// 创建File mapping
hMapFile = CreateFileMapping( hFile,          // 需要映射的文件的句柄
    NULL,           // 安全选项: 默认
    PAGE_READWRITE, // 可读、可写
    0,              // mapping对象的大小，高位
    dwFileMapSize,  // mapping对象的大小，低位
    NULL);          // mapping对象的名字
if (hMapFile == NULL)
{
    printf("CreateFileMapping error: %d\n", GetLastError() );
    return 1;
}
// 映射view
lpMapAddress = MapViewOfFile(hMapFile,            // mapping对象的句柄
     FILE_MAP_ALL_ACCESS, // 可读，可写
     0,                   // 映射的文件偏移，高位
     dwFileMapStart,      // 映射的文件偏移，低位
     dwMapViewSize);      // 映射到View的数据大小
if (lpMapAddress == NULL)
{
     printf("MapViewOfFile error: %d\n", GetLastError());
```

```
            return 1;
        }
        printf ("文件 map view 相对于文件的起始位置: 0x%x\n",
            dwFileMapStart);
        printf ("文件 map view 的大小: x%x\n", dwMapViewSize);
        printf ("文件 mapping 对象的大小: x%x\n", dwFileMapSize);
        printf ("从相对于 map view 0x%x 字节的位置读取数据, ", iViewDelta);
        // 将指向数据的指针偏移, 到达我们关心的地方
        pData = (PCHAR) lpMapAddress + iViewDelta;
        // 读取数据, 赋值给变量
        iData = *(PINT)pData;
        // 显示读取的数据
        printf ("为: x%.8x\n", iData);
        // 从 mapping 中复制数据, 32 个字节, 并打印
        CopyMemory(cMapBuffer,lpMapAddress,32);
        printf("lpMapAddress 起始的字节是: ");
        for(i=0; i<32; i++)
        {
            printf("0x%.2x ",cMapBuffer[i]);
        }
        // 将 mapping 的前 32 个字节用 xff 填充
        FillMemory(lpMapAddress,32,(BYTE)0xff);
        // 将映射的数据写回到硬盘上
        FlushViewOfFile(lpMapAddress,dwMapViewSize);
        printf("\n 已经将 lpMapAddress 开始的字节使用 xff 填充。\n");
        // 关闭 mapping 对象
        if(!CloseHandle(hMapFile))
        {
            printf("\nclosing the mapping object error %d!",
                GetLastError());
        }
        //关闭文件
        if(!CloseHandle(hFile))
        {
            printf("\nError %ld occurred closing the file!", GetLastError());
        }
        return 0;
    }
```

运行编译完成的文件，如下所示：

```
文件大小: 262140
文件 map view 相对于文件的起始位置: 0x20000
文件 map view 的大小: 0x8c04
文件 mapping 对象的大小: 0x28c04
从相对于 map view 0x8804 字节的位置读取数据, 为: 0x0000a201
lpMapAddress 起始的 32 字节是: 0x00 0x80 0x00 0x00 0x01 0x80 0x00 0x00 0x02 0x80 0x00 0x00 0x03
0x80 0x00 0x00 0x04 0x80 0x00 0x00 0x05 0x80 0x00 0x00 0x06 0x80 0x00 0x00 0x07 0x80 0x00 0x00
已经将 lpMapAddress 开始的 32 字节使用 0xff 填充。
```

查看生成的文件“test.dat”，在 0x20000 位置上的数据已经通过文件映像进行了修改，0x28804 位置上的数据读取正确。如图 4-8 和图 4-9 所示。

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0001ffe0h: F8 7F 00 00 F9 7F 00 00 FA 7F 00 00 FB 7F 00 00 ; ?..?..?..?..
0001fff0h: FC 7F 00 00 FD 7F 00 00 FE 7F 00 00 FF 7F 00 00 ; ?..?..?..  ..
00020000h: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ; ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020010h: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ;
00020020h: 08 80 00 00 09 80 00 00 0A 80 00 00 0B 80 00 00 ; .€...€...€...€..
00020030h: 0C 80 00 00 0D 80 00 00 0E 80 00 00 0F 80 00 00 ; .€...€...€...€..
00020040h: 10 80 00 00 11 80 00 00 12 80 00 00 13 80 00 00 ; .€...€...€...€..
00020050h: 14 80 00 00 15 80 00 00 16 80 00 00 17 80 00 00 ; .€...€...€...€..
00020060h: 18 80 00 00 19 80 00 00 1A 80 00 00 1B 80 00 00 ; .€...€...€...€..
00020070h: 1C 80 00 00 1D 80 00 00 1E 80 00 00 1F 80 00 00 ; .€...€...€...€..
00020080h: 20 80 00 00 21 80 00 00 22 80 00 00 23 80 00 00 ;  €..!€..”€..#€..
00020090h: 24 80 00 00 25 80 00 00 26 80 00 00 27 80 00 00 ; $€..%€..&€..'€..
000200a0h: 28 80 00 00 29 80 00 00 2A 80 00 00 2B 80 00 00 ; (€..)€..*€..+€..
```

图 4-8　查看通过 mapping 写入的数据

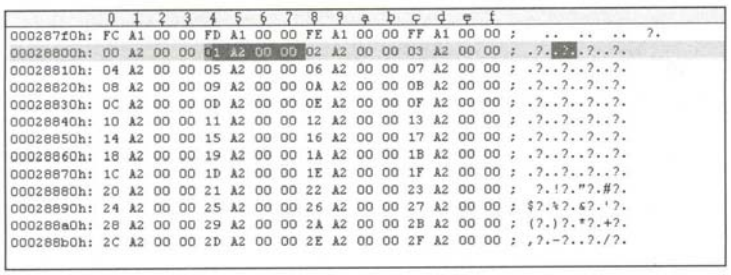

图 4-9　查看通过 mapping 读出的数据

4.4.2　通过Mapping File在进程间传递和共享数据

进程间通信、共享数据有很多种方法，文件映射是常用的一种方法。因为 mapping 对象系统中是全局的，一个进程创建的 Mapping 对象可以从另外一个进程中打开，映射视图就进程间共享的内存了。一般在共享的内存数据量比较大时，选择使用文件映射进行共享。

1．关键 API

（1）OpenFileMapping。

打开已经存在的文件映射，函数原型如下：

```
HANDLE OpenFileMapping(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,   FALSE
  LPCTSTR lpName
);
```

◇参数

dwDesiredAccess：输入参数，Mapping 对象的存取权限，参见 CreateFileMapping 函数的 fiProtect 参数。

bInheritHandle：输入参数，如果设置为 TRUE，则可能继承进程句柄，否则不能继承。一般设置为 FALSE。

lpName：输入参数，打开的对象，mapping 对象的名字，由 CreateFileMapping 的 lpName 参数}旨定。

0 返回值

返回 HANDLE 值，Mapping 对象的句柄。如果返回 NULL，则表示失败。可使用 GetLastError 函数获取错误信息。

(2) UnmapView0 伍 ile。

取消文件映射，函数原型如下：

```
BOOL UnmapViewOfFile(
  LPCVOID lpBaseAddress
);
```

参数 lpBaseAddress 为需要取消映射的内存地址。

◇返回值

返回值 BOOL，表示是否成功。

2．实例 4-14 创建 Mapping 对象、进行共享数据

本实例创建一个 Mapping 对象，并在对象的视图中写入需要共享的数据，等待别的进程从 mapping 对象中读取数据，与实例 4-15 配合使用。

```
/* 头文件 */
#include <windows.h>
#include <stdio.h>
#include <conio.h>
/* 预处理申明 */
#define BUF_SIZE 256
/* 全局变量 */
LPTSTR szName = TEXT("SharedFileMappingObject");
LPTSTR szMsg = TEXT("进程的消息");

/* ************************************
* int main(void)
* 功能：演示文件 Mapping 共享内存，写入数据到共享内存
* 返回值：0 代表执行完成，1 代表发生错误
**************************************/
void main(int argc, PCHAR argv[])
{
     //文件映射句柄
     HANDLE hMapFile;
     //共享数据缓冲区指针
     LPTSTR pBuf;
     //创建命名的文件映射，不代表任务硬盘上的文件
     hMapFile = CreateFileMapping(
          INVALID_HANDLE_VALUE,
          NULL,
          PAGE_READWRITE,
          0,
          BUF_SIZE,
          szName);
     if (hMapFile == NULL || hMapFile == INVALID_HANDLE_VALUE)
     {
          printf("CreateFileMapping error: %d\n",  GetLastError());
          return;
     }
     //创建 View
     pBuf = (LPTSTR) MapViewOfFile(hMapFile,
          FILE_MAP_ALL_ACCESS,
          0,
          0,
          BUF_SIZE);
     if (pBuf == NULL)
     {
          printf("MapViewOfFile error %d\n",  GetLastError());
          return;
     }
     //将共享数据复制到文件映射中，如果运行时输入了参数则使用参数
     if(argc==1)
     {
          CopyMemory((PVOID)pBuf, szMsg, strlen(szMsg));
     }
     else
     {
          DWORD dwCopyLen = (strlen(argv[1])<BUF_SIZE) ? strlen(argv[1]): BUF_SIZE;
          CopyMemory((PVOID)pBuf, argv[1], dwCopyLen);
     }
     printf("运行程序，完成运行后，按任意键退出。");
     getch();
     //取消映射，退出
     UnmapViewOfFile(pBuf);
     CloseHandle(hMapFile);
}
```

3．实例 4-15 读取并显示共享数据

本实例代码从实例 4-14 共享的 Mapping 对象中读取数据，并将数据显示出来。实例 4-14 与实例 4-15 是两个进程，实现了进程间数据的共享。这两个进程可以通过各自的指针，对同一段内存进行读写。

本实例先打开指定对象名的 Mapping 对象（与实例 4-14 中使用的 CreateFileMapping 创建的 Mapping 对象需有一样的对象名），然后创建视图，从视图中读取数据。

```
/* ************************************
* pro_s2.c 通过 Mapping File 在进程间共享内存
**************************************/
#include <windows.h>
#include <stdio.h>
#include <conio.h>
/* 预处理申明*/
#pragma comment (lib, "User32.lib")
#define BUF_SIZE 256
/* 全局变量 */
TCHAR szName[]=TEXT("SharedFileMappingObject");

/* ************************************
* int main(void)
* 功能：演示文件 Mapping 共享内存，从共享数据中读信息
* 返回值：0 代表执行完成，1 代表发生错误
**************************************/
void main()
{
     HANDLE hMapFile;
     LPTSTR pBuf;
     //打开文件 Mapping
     hMapFile = OpenFileMapping(
         FILE_MAP_ALL_ACCESS,
         FALSE,
         szName);
     if (hMapFile == NULL)
     {
         printf("OpenFileMapping error: %d.\n",  GetLastError());
         return;
     }
     //映射
     pBuf = MapViewOfFile(hMapFile,
          FILE_MAP_ALL_ACCESS,
          0,
          0,
          BUF_SIZE);
     if (pBuf == NULL)
     {
          printf("MapViewOfFile error %d\n", GetLastError());
          return;
     }
     //消息得到的共享数据
     MessageBox(NULL, pBuf, TEXT("Process2"), MB_OK);
     //取消 Mapping，关闭句柄，返回
     UnmapViewOfFile(pBuf);
     CloseHandle(hMapFile);
}
```

实例 4-14 的代码编译生成 pro_s1.exe，实例 4-15 的代码编译生成 pro_s2.exe，运行：

```
pro_s1.exe "data to process2"
```

运行后先不要退出，然后运行 pro_s2.exe，会弹出对话框，对话框中的内容是 pro_s1.exe 共享的内容，如图 4-10 所示。

图 4-10　pro_s2.exe 运行结果

4.4.3　通过文件句柄获得文件路径

Windows 系统提供了一个名为 GetMappedFileName 的 API 函数，这个函数可以实现从 mapping 对象的句柄得到被映射文件的路径。但是路径是以设备名的形式给出的，如类似于“\Device\HarddiskVolume3\1.TXT”的形式。

将设备名转换为路径名需要使用到一个 API 函数-QueryDosDevice，这个函数可以将驱动器的根路径转换为设备名，然后进行循环比较，可得文件路径。

1．关键 API

(1)GetMappedFileName。

从映射对象获取被映射文件的文件设备名，函数原型如下：

```
DWORD QueryDosDevice(
  LPCTSTR lpDeviceName,
  LPTSTR lpTargetPath,
  DWORD ucchMax
);
```

◇参数

hProcess：输入参数，映射对象所属的进程，如果设备对象是在本进程中创建使用 GetCurrentProcess 函数获得。

lpv：输入参数，映射视图的地址，MapViewOffile 的返回值。

lpFilename：输出参数，指向用于存在文件路径缓冲区的指针。

nSize：输入参数，lpFilename 参数指向的内存区域的大小，防止溢出。

◇返回值

返回 DWORD 值，如果成功，返回值为获得的文件设备名的长度，如果失败为 0。

(2)QueryDosDevice

获取 MS_DOS 设备名，函数原型如下：

```
DWORD GetMappedFileName(
  HANDLE hProcess,
  LPVOID lpv,
  LPTSTR lpFilename,
  DWORD nSize
);
```

◇参数

lpDeviceName：输入参数，MS-DOS 设备名字符串。

lpTargetPath：输出参数，用于存储返回结果的缓冲区地址。

ucchMax：输入参数，lpTargetPath 指向的缓冲区的大小。

◇返回值返回 DWORD 值，如果成功，则返回值为 lpTargetPath 指向字符串的长度，如果失败为 0。

2．实例 4-16 通过文件句柄获取文件路径

GetFileNameFromHandle 通过参数输入的句柄创建 mapping 对象，然后调用 GetMapped FileName 函数，获得对象所映射的文件的设备名，然后使用 QueryDosDevice 函数循环判断

驱动器设备名与路径的关系，在找到正确的驱动器路径后，构造好文件路径并输出。

```
/* ************************************
* handle_path.cpp 通过文件句柄获取文件路径
**************************************/
/* 头文件*/
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
#include <string.h>
#include <psapi.h>
/* 预处理申明*/
#pragma comment (lib, "Psapi.lib")
#define BUFSIZE 512
/* 函数申明*/
BOOL GetFileNameFromHandle(HANDLE hFile) ;

/* ************************************
* BOOL GetFileNameFromHandle(HANDLE hFile)
* 功能：从文件句柄获取文件路径
* 参数：ANDLE hFile，需要获得路径的文件句柄
* 返回值：BOOL，表示是否成功
**************************************/
BOOL GetFileNameFromHandle(HANDLE hFile)
{
     TCHAR pszFilename[MAX_PATH+1];
     HANDLE hFileMap;
     PVOID pMem;
     // 获得文件大小
     DWORD dwFileSizeHigh = 0;
     DWORD dwFileSizeLow = GetFileSize(hFile, &dwFileSizeHigh);
     if( dwFileSizeLow == 0 && dwFileSizeHigh == 0 )
     {
          printf("不能 map 文件大小为的文件.\n");
          return FALSE;
     }
     // 创建 Mapping 对象
     hFileMap = CreateFileMapping(hFile,
          NULL,
          PAGE_READONLY,
          0,
          1,
          NULL);
     if (!hFileMap)
     {
          printf("CreateFileMapping error: %d",GetLastError());
          return FALSE;
     }
     pMem = MapViewOfFile(hFileMap, FILE_MAP_READ, 0, 0, 1);
     if (!pMem)
     {
          printf("MapViewOfFile error: %d",GetLastError());
          return FALSE;
     }
     // 从 Mapping 对象获得文件名
     if (0 == GetMappedFileName (GetCurrentProcess(),
          pMem,
          pszFilename,
```

```
            MAX_PATH))
        {
            printf("GetMappedFileName error: %d",GetLastError());
            return FALSE;
        }
        // 将设备名转换为路径
        TCHAR szTemp[BUFSIZE] = {0};
        if (0 == GetLogicalDriveStrings(BUFSIZE-1, szTemp))
        {
            printf("GetLogicalDriveStrings error: %d",GetLastError());
            return FALSE;
        }
        TCHAR szName[MAX_PATH];
        TCHAR szDrive[3] = {0};
        BOOL bFound = FALSE;
        PTCHAR p = szTemp;
        do
        {
            CopyMemory(szDrive,p,2*sizeof(TCHAR));
            // 通过路径查找设备名
            if (!QueryDosDevice(szDrive, szName, BUFSIZE))
            {
                printf("QueryDosDevice error: %d",GetLastError());
                return FALSE;
            }
            UINT uNameLen = lstrlen(szName);
            if (uNameLen < MAX_PATH)
            {
                //比较驱动器的设备名与文件设备名是否匹配
                bFound = strncmp(pszFilename, szName,uNameLen) == 0;
                if (bFound)
                {
                    //如果匹配，说明已经找到，构造路径
                    TCHAR szTempFile[MAX_PATH];
                    wsprintf(szTempFile,
                        TEXT("%s%s"),
                        szDrive,
                        pszFilename+uNameLen);
                    lstrcpy(pszFilename, szTempFile);
                }
            }
            // 循环到下一个 NULL
            while (*p++);
        } while (!bFound && *p);
        UnmapViewOfFile(pMem);
        CloseHandle(hFileMap);
        printf("File path is %s\n", pszFilename);
        return TRUE;
    }

    /* ************************************
    * int main()
    * 功能：查找第一个目录中第一个 txt 文件
    *       打开文件，并根据文件句柄获得文件路径
    * 返回值：0 表示成功，表示失败
    ************************************/
    int main()
    {
        HANDLE hFile;
        HANDLE hFind;
        WIN32_FIND_DATA wfd;
        hFind = FindFirstFile("*.txt",&wfd);
        if(hFind == INVALID_HANDLE_VALUE)
        {
```

```
            printf("can not find a file");
            return 1;
      }
      //CloseHandle(hFind);
      printf("find %s at current dir\n",wfd.cFileName);
      hFile = CreateFile(wfd.cFileName,
            GENERIC_READ | GENERIC_WRITE,
            0,
            NULL,
            OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL,
            NULL);
      if(hFile == INVALID_HANDLE_VALUE)
      {
            printf("create file error, %d",GetLastError());
      }
      else
      {
            GetFileNameFromHandle(hFile) ;
      }
      CloseHandle(hFile);
      return 0;
}
```

运行“handle_path.exe”，结果如下：

```
can not find a txt file
```

这是由于本实例是查找实例所在目录下第一个 txt 文件，获得它的句柄后转换为路径。为此，笔者新建一个 1.txt，大小为 0。再次运行得到以下输出：

```
find 1.txt
 at current dirCannot map a file with a length of zero.
```

原因是不能对大小为 0 的文件进行文件映射，在文件中添加一些内容，再次运行，结果如下：

```
find 1.txt
 at current dirFile path is F:\Study\Writting\ch4\HandlePath\XP32_RETAIL\1.txt
```

程序顺利取得了 txt 文件的全路径。

4.5 总结

本章使用若干个实例介绍了众多 Windows 文件系统管理的 API 函数涵盖文件系统的基本概念、磁盘与驱动器的管理、文件与目录的创建、打开、删除、读写、重命名、属性的获取与修改，目录的遍历等，还介绍了使用 Mapping 对象的有关文件系统的高级操作。本章包含了文件系统程序设计的最常用的应用实例。

第 5 章　内存管理

内存主要储存程序运行时所需的机器代码、数据等内容。内存是计算机运行的必要条件，所有计算机程序都必须要和内存打交道，如何使用内存是每个程序员所必须要处理好的问题。有效管理内存、有效使用内存是开发出高效、高可靠性、高安全性的程序的关键。

本章将介绍 Windows 系统内存管理的原理、如何使用 Windows 系统为程序开发人员提供的内存管理 API 函数的来进行内存分配等操作，并对内存属性、权限等概念进行解释，讲解各种内存分配、释放、读写等操作的原理和方法。

5.1　Windows内存管理原理

数据是存储于内存中的，为了能够找到存在于内存中的数据，开发人员需要知道数据所在的内存地址。内存的最小存储单元是字节，内存中的每一个字节都有一个地址。在 32 位系统上，使用 32 位的数来表示内存地址，因此一共可以表示 232 (4294967296-4G)个字节。

5.1.1　基本概念

(1)地址空间。

系统中所有可用的内存地址的集合称为地址空间。比如，如果可以使用 4GB 的内存，那么其地址空间就是 0x00000000～0xFFFFFFF。

(2)物理内存。

硬件系统中真实存在的存储空间称为物理内存，物理内存的访问通过硬件系统总线进行的。

但不是每台 32 位的机器都具有 4GB 的物理地址空间，因此物理地址空间不一定是 0x00000000～0xFFFFFFFF，比如在物理内存为 1GB 的系统上，就可能只有 0x00000000～0x3FFFFFFF 的地址可以使用。

(3)虚拟地址空间。

为了访问内存的统一和方便，操作系统允许其上运行的程序访问所有的 4GB 内存空间中的地址。因此操作系统必须进行一些必要的地址转换工作，将程序访问的地址转换为物理内存中的真实物理地址，然后进行数据的存取。操作系统进行转换后，供程序使用的地址空间称为虚拟地址空间。在 32 位系统上，可以使用的虚拟地址空间大小是 4GB。

当然，由于虚拟地址空间可能比真实物理地址空间大，系统会将部分虚拟地址空间中的地址转换为硬盘中的数据，在必要时将物理内存中的数据与硬盘中的数据进行交换。

这种地址转换和数据交换是通过分页和分段机制实现的，将在 5 .1.2 小节详细介绍这种机制。

(4)进程的内存空间：用户内存空间与内核内存空间。

Windows 操作系统中的每个进程都有属于自己的虚拟地址空间。32 位的 Window 操作系统将 4GB（在 64 位系统上，这个值达到了 8TB）的虚拟内存划分为两个部分，进程使用 2GB，称为用户进程空间；内核使用 2GB，称为系统地址空间或内核地址空间（也可设置为用户进程空间使用 3GB，内核使用 1GB，这里只讨论各为 2GB 的情况），用户空间的地址范围为 0x00000000～0x7FFFFFFF，内核空间的地址范围为 0x80000000～0xFFFFFFFF。虚拟地址空间在进程上是封闭的，进程只能访问属于自己的地址空间，如果要访问其他进程的地址空间

需特殊的机制。

为了能够熟练地使用 Windows 内存管理的相关接口，Windows 应用程序开发人员需要了解以下 Windows 内存管理的知识。

◇分段与分页内存管理。

◇物理地址与虚拟地址、虚拟地址空间。

◇虚拟内存布局，内存的分工，堆与栈。

◇　内存的存取权限。

◇标准 C 内存管理函数与 Windows 内存管理 API 的关系。

5.1.1—5.1.6 小节将分别介绍这些基础知识。

本书只讨论在 32 位系统下的情况，64 位系统有一些不同，但是基本原理类似。

5.1.2　分页与分段内存管理、内存映射与地址转换

Intel 体系结构（x86、IA-64）的 CPU 支持对内存进行分页与分段内存管理。

x86 体系结构 CPU 中，有若干个段寄存器（段选择器），可以通过段“选择器+偏移”的形式来表示内存地址。使用这种方式表示的地址称为“逻辑地址”，这种机制称为“分段”。系统会将逻辑地址转换为线性地址。

转换的过程如图 5-1 和图 5-2 所示。

线性地址与物理内存地址类似，是平坦的（未分段的），使用 32 位数表示，地址是连续的。线性地址空间的范围是 0x00000000—0xFFFFFFFF。

如果系统的物理内存为 4GB，那么可以不使用分页，线性地址直接对应到物理地址上。如果使用了分页，线性地址与物理地址的对应是通过分页机制实现的。页的大小可以为 4KB、2MB 或 4MB。最常用的是 4KB，在 4KB 情况下，32 位的线性地址被分为了 3 个部分。如图 5-1 所示，图中以页大小为 4KB 为例，说明系统的分页机制。

32 位地址的高 10 位称为页目录索引(Directory)、次 10 位称为页表索引(Table)，余下的 12 位为偏移(Offset)。如果只取前两个部分的 20 位，末位补 0，那么其可能指向的地址都是 4KB(212)对齐的，也就是页的基地址。所以 32 位中的前 20 位定位了页的基地址(PFN)，而最后 12 位是地址在页内的偏移。

在地址转换的过程中，还涉及一个关键的系统寄存器 CR3。CR3 寄存器中的高 12～32 位保存了页目录索引在物理内存中的地址。

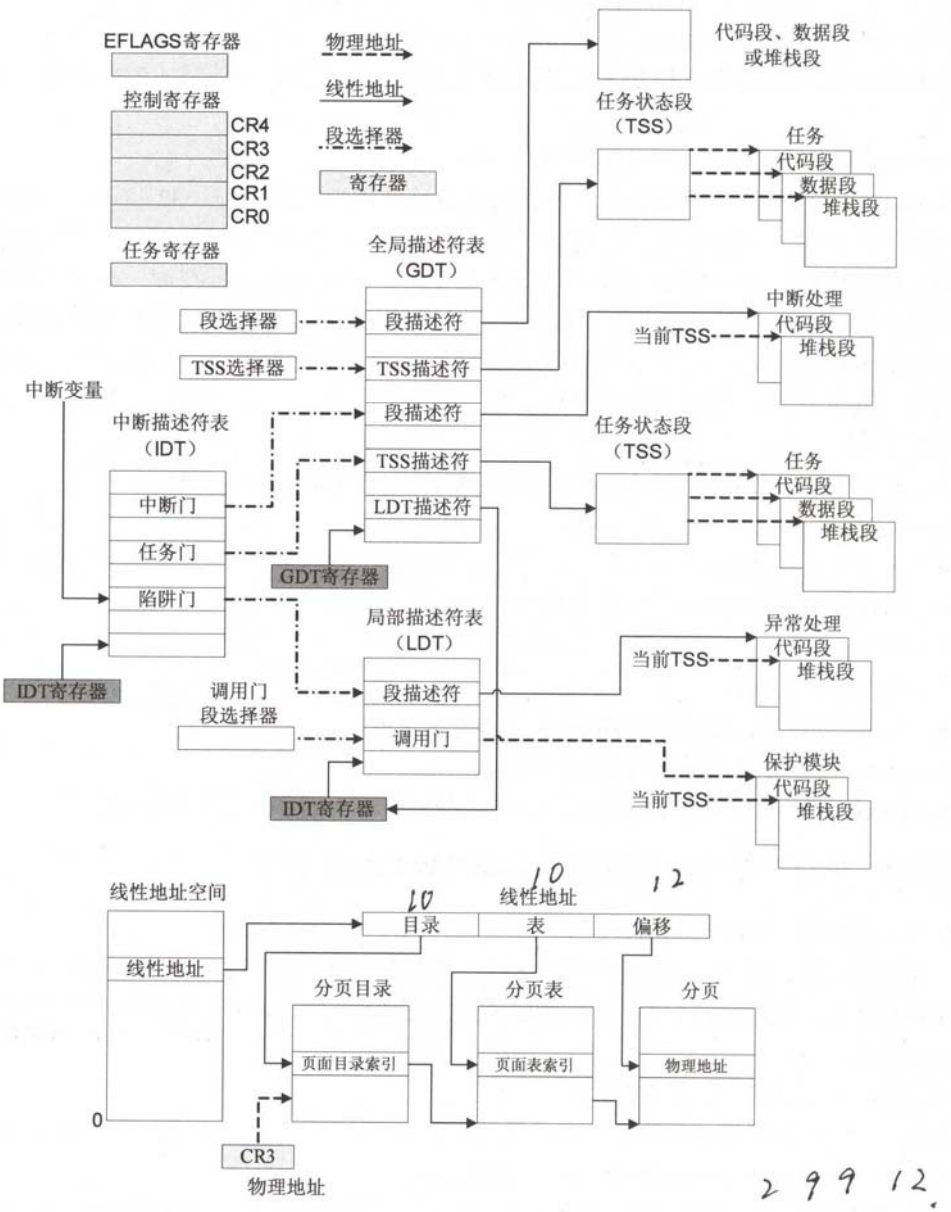

图 5-1　x86 系统地址转换

地址转换经过了如下过程。

(1)从 CR3 寄存器的 12～32 位中取出地址，末位补 0，构成一个 32 位的地址，此地址是页目录所在物理内存的基地址（因此页目录在内存中肯定是 4KB 对齐的）。

(2)以线性地址中的高 10 位的页目录索引（Directory）中数据为偏移，加上上一步得到的页目录所在物理内存的基地址，在页目录中找到相应的页目录项(PDE)。

(3)页目录项中包括了页表的基地址，线性地址中的页表索引(Table)是页表中的偏移，找到页表项(PTE)。

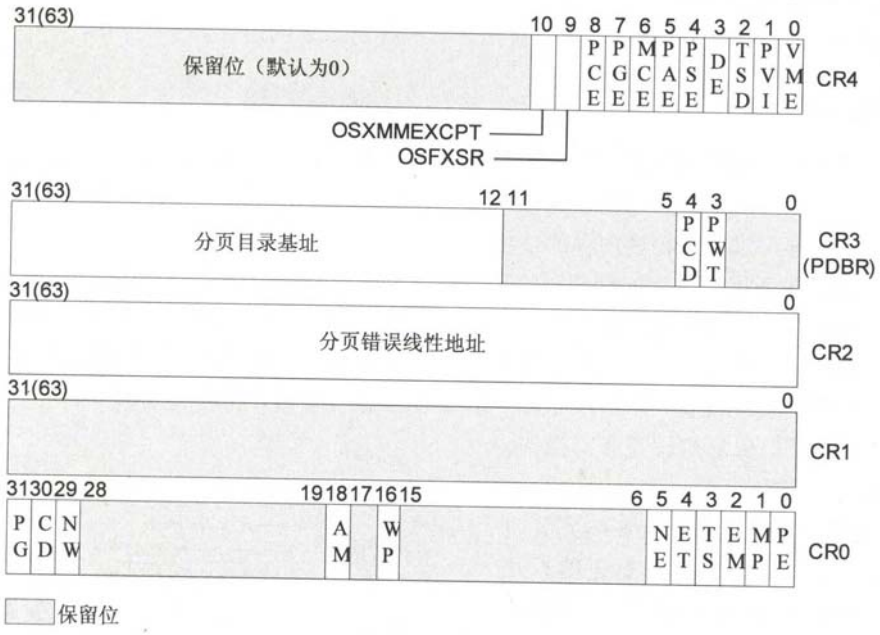

图 5-2　控制寄存器

(4)页表项(PTE)中存储的是物理地址空间中的页面的地址(PFN)，如果 PFN 指向的页面是无效的（不在物理内存中），那么会在系统内核中引起页错误，系统内核会进行页面调度，将物理内存中暂不使用的页面交换到硬盘中，将硬盘中需要访问的页面交换到物理内存中，使用 PFN 指向的页是有效的。

(5)现在已经找到了线性地址所对应的物理地址所在的页的基址。

(6)在物理页地址的基础上，加上线性地址末 12 位的偏移，可得到线性地址与物理地址的对应。

从中可以看出地址转换是以页为单位的，也就是线性地址与所对应的物理地址的页基地址(PFN)是会变化的，在页内的偏移地址是不会发生变化的。

每一个页目录项对应了一个页表，一共有 210 个页目录。每一个页表项对应了一个内存分页，每个页表是最多可以表示 210 个页表。最多一共是多有 220 个页。

5.1.3　进程的内存空间

1．进程虚拟地址空间的实现

5 .1.2 小节中描述了系统分页机制实现的虚拟内存与物理内存的对应关系 Windows 系统中每个进程都有一个私有的虚拟地址空间，系统需要将每个进程的虚拟地址都映射到物理内存地址上。为了实现系统中每个进程都有一个私有的虚拟地址空间，系统为每一个进程都创建一个页目录(Directory)和一组页表。每个进程的页表是独立的，而内核空间的页表是所有进程共享的。

如 5.1.2 小节所述，在 x86 平台上，CR3 寄存器标识了页目录所在的物理地址。Windows 系统上的第一个进程都有独立的页目录，因此系统运行时，CR3 寄存器中保存的页目录地址是会变化的，其值为当前运行的进程的页目录地址。当系统要进行进程切换时，会将 CR3 寄存器中保存的页目录地址设置为将要切换过去并开始运行的进程的页目录地址。Windows 平台的每一个进程都有一个 KPROCESS 结构的数据块，其中包括了其进程页目录的地址。

2．数据共享与保护

有一些进程间共享的数据、系统的可执行代码（系统 DLL）等，在各个进程间都是一致的，因此没有必要在物理内存中为这些数据保存多份，不同进程的虚拟内存分页可以映射为同样的物理内存分页，这样可以节省物理内存的使用。

同时，为了保证这种在映射到相同物理内存页上的内存分页在进程上仍然是私有的，系统还提供了一些保护机制。如果某个进程将某个系统 DLL 加载入进程内存空间以后，对该 DLL 中的数据进行了写操作，系统就会监视到该操作，并在数据写入之前，将要写入的进程虚拟内存分页映射到另一个新的物理分页，并将原分页中系统 DLL 的内容复制到这个分页中，进程间不共享这个新物理内存分页，最后进程完成写操作，将数据写入这个新分页中。这种对内存中系统 DLL 数据的写入操作不会影响到其他进程，因此保护了各个进程中数据的独立性，这种机制叫 Copy-On-Write。如果系统确实需要在进程间共享数据，也可以对特定的页面不使用 Copy-On-Write，如图 5-3 与图 5-4 所示。

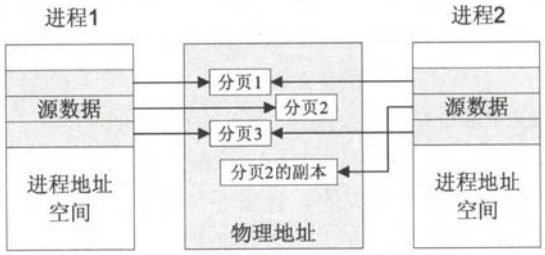

图 5-4　Copy-On-Write 机制，向共享内存写入数据之后

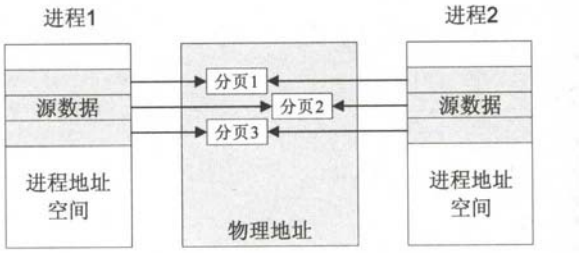

图 5-3　Copy-On-Write 机制，向共享内存写入数据之前

3．应用程序开发人员需要了解的内容

在编写应用程序时，程序员不用过多地考虑系统底层是如何实现这种机制的。在进行应用程序开发时，开发人员只需要了解以下内容。

(1)虚拟地址空间中的数据是分页管理的。

(2)应用程序不用考虑系统中其他应用程序的内存使用情况，如占用了多少内存、占用了哪些内存等。

(3)虚拟地址并不是物理地址空间中的地址，不是数据在内存中真实存在的地址，操作系统会将进程的虚拟地址映射到真实的物理内存的地址。

(4)进程也不用考虑真实的物理内存有多大，只需要了解可以使用 2GB（一般情况下）的内存，操作系统负责转换。

(5)如果系统中没有足够的物理内存供使用，那么操作系统会将当前没有使用的内存分页“调度”到硬盘上保存起来。页面调度不会造成内存中的数据在虚拟地址空间中地址的改变，所以进程不需要知道内存分页是如何调度的，不需要知道内存中保存的数据是在内存中还是在硬盘上，只需要知道其虚拟地址就可以了。

5.1.4 虚拟内存布局、内存的分工、堆与栈

进程的虚拟内存空间分为两个部分，低 2GB(或 3GB_)由应用程序使用，高 2GB（或 1GB）由系统内核程序使用。

系统内核的内存空间中包括驱动程序，系统内核可执行程序、用于内存管理的数据结构（页目录、页表等），用于进程管理、线程调度的数据结构、各种中断处理程序，系统缓存等。

应用程序可使用的低地址空间中包括了应用程序的代码、数据（全局变量等）、系统和用户 DLL 的代码、各线程的栈、堆等。

栈和堆是两种重要的内在管理形式。

进程的每个线程都有自己的栈，栈与函数的调用、执行和返回及局部变量的保存相关，一般情况下，栈中保存着函数的参数、返回地址和局部变量，调用函数将参数入栈，调用指令将函数返回地址入栈，被调用函数负责保存调用函数的相关栈指针，为局部变量分配空间等，而堆是一种可以动态分配和释放的内存，由堆管理器进行管理，用户在程序中使用的内存分配函数都是通过堆进行分配的。

5.1.5 内存的保护属性和存取权限

系统为每个内存分页提供保护属性和存取权限，内存的保护属性和存取权限的最小单位是分页，也就是说同一个分页中的内存必然具有相同的保护属性和存取权限。用户地址空间中内存分页有多种保护属性和存取权限，如表 5.1 所示。

表 5.1 保护属性与存取权限

属性常量	意义
PAGE_EXECUTE	可执行
PAGE_EXECUTE_READ	可读、可执行
PAGE_EXECUTE_READWRITE	可读、可写、可执行
PAGE_EXECUTE_WRITECOPY	可读、可写、可执行，以 Read-on-write 和 copy-on-write 方式共享
PAGE_NOACCESS	不可访问
PAGE_READONLY	只读
PAGE_READWRITE	可读、可写
PAGE_WRITECOPY	copy-on-write 保护机制
PAGE_GUARD	保护，如果访问则产生异常（不能单独使用）
PAGE_NOCACHE	不进行 CPU 缓存（不能单独使用）
PAGE_WRITECOMBINE	write-combined 优化（不能单独使用）

5.1.6 本章API列表

本章共讲解 5 类 API 共 31 个主要涉及堆管理、全局和局部内存管理以及虚拟内存管理等，如表 5.2～5.5 所示。

表 5.2　　堆管理 API 函数

API	功 能 简 介
HeapCreate	为进程创建堆，返回一个堆句柄
GetProcessHeap	获取当前进程中的一个堆，返回一个句柄
GetProcessHeaps	获得进程中所有堆，包括堆的数量和各个堆的句柄
HeapAlloc	从指定堆上分配内存块
HeapReAlloc	重新分配内存，改变已经分配好的堆内存块的大小
GetSystemInfo	获取系统信息
HeapSize	获取指定堆的大小，以字节为单位返回堆大小信息
HeapFree	释放 HeapAlloc 和 HeapReAlloc 所分配的内存
HeapDestroy	销毁由 HeapCreate 创建的堆

表 5.3　　全局和局部内存管理 API 函数

API	功 能 简 介
GlobalAlloc	在默认堆上分配指定属性和大小的内存
GlobalFree	释放由 GlobalAlloc 分配的内存
GlobalReAlloc	重新分配内存，改变已经分配好的堆内存块的大小
GlobalLock	将 GlobalAlloc 及 GlobalReAlloc 分配的属性为 GMEM_MOVEABLE 的内存块设为固定
GlobalHandle	与 GlobalLock 相对
GlobalSize	获取内存大小

注意　局部内存管理中的 API 函数和全局内存管理 API 函数的使用方法非常相似，本章限于篇幅没有详细介绍。

表 5.4　　虚拟内存管理 API 函数

API	功 能 简 介
VirtualAlloc	“保留”或“提交”内存页面，将“空闲的”内存页面变为“保留的”或“已提交的”，将“保留的”页面变为“提交的”
VirtualAllocEx	类似 VirtualAlloc 功能，不过可以为其他进程分配内存
VirtualFree	将内存状态从“已提交的”变为“保留的”，或将“保留的”变为“空闲的”，或同时进行
VirtualFreeEx	释放由 VirtualAllocEx 分配的内存，功能、使用方法和 VirtualFree 类似
VirtualProtect	改变指定虚拟内存分页的保护属性

表 5.5　　内存操作与内存信息管理 API 函数

API	功 能 简 介
CopyMemory	复制内存，第一个参数为目的地址，第二个参数为源地址，第三个参数为复制数据的大小，单位为字节
FillMemory	填充内存，将一段内存填充为同一个值
MoveMemory	功能和 CopyMemory 类似，不同的是源地址和目的地址可以相同
ZeroMemory	将指定内存区域清零
GlobalMemoryStatusEx	用于获取系统当前内存使用情况
IsBadCodePtr	判断调用进程是否拥有对指定地址内存的读操作权限

续表

API	功 能 简 介
IsBadReadPtr	判断调用进程是否拥有对指定地址段内存的读操作权限
IsBadStringPtr	判断调用进程是否拥有对指定字符串指针的读取权限
IsBadWritePtr	判断调用进程是否拥有对指定地址段内存的写操作权限

5.2 堆管理

用户使用内存分配函数分配的内存都位于堆中，所以使用堆管理函数对内存进行分配、释放等是最为直接的方式。

Windows 系统中，每个进程都有自己的堆，每个进程的堆的数量也有所不同。Windows 系统中所谓的“堆”(Heap)并不是内存块，而是一种用于内存管理的对象，也是一种内存组织的形式。进程可以从属于自己的堆上分配内存和释放内存。堆包括有若干种属性，如堆的大小最大值可以是固定的、也可以是“可增长的”；堆上的数据内容是否可以作为代码可执行；堆是否可连接存取等。

本节中将介绍如何在堆上进行内存分配、再分配、释放、获取堆属性等操作。

5.2.1 获取堆句柄、分配与再分配堆

堆是一种内存管理的对象，一个进程有若干个堆，在分配内存前需要指定从哪个堆上进行分配。堆的句柄惟一标识了一个堆。在堆上分配内容前，首先需要获得所要进行分配的堆的句柄，获得堆的句柄有两种方式：一种是获取在进程中时已经创建好的堆，二是进程自己再创建堆。有了堆的句柄后，就可以在堆上进行内存分配了。

如果一次分配的内存大小不合适，还可以进行再分配。再分配不会对原内存中的数据造成影响，分配的内存使用完成后要释放。在堆中分配内存时，用户不需要考虑分页、不需要考虑对齐，内存块可以为任意大小（不超过可使用的内存容量范围）堆管理器会负责管理。

1．关键 API

（1）HeapCreate

为进程创建新堆，请求分配虚拟内存分页，函数原型如下：

```
HANDLE HeapCreate(
  DWORD flOptions,
  SIZE_T dwInitialSize,
  SIZE_T dwMaximumSize
);
```

◇参数

flOptions:输入参数，创建堆的选项，其值如表 5.6 所示。如果要指定多个选项，使用“I”运算，将多个选项连接起来。

表 5.6 flOptions 参数取值选项

选　项	意　义
HEAP_CREATE_ENABLE_EXECUTE	分配的内存允许用于代码执行
HEAP_GENERATE_EXCEPTIONS	如果分析内存失败会产生异常，而不是返回 NULL
HEAP_NO_SERIALIZE	不进行连续存取

dwInitialSize:输入参数，堆的初始化大小（以字节为单位），如果指定的大小没有内存页对齐（值为内存页大小的整数倍），系统分自动进行内存页对齐。如果为 0，系统分自动指定为一个内存页的大小。

dwMaximumSize:输入参数，堆大小的最大值，如果指定了，则此堆分配的内存块大小不能大于此值。如果为 0，那么此堆是“增长的”，也就是其上分配内存的大小不受限制，可以达到系统中所能获得的最大内存。

◇返回值

返回为 HANDLE 类型，如果失败，返回 NULL，如果成功，返回所创建的堆的句柄。

（2）GetprocessHeap。

获取当前进程中的一个堆，返回堆的句柄，其函数原型如下：

```
HANDLE GetProcessHeap(void);
```

◇无参数

◇返回值

返回为 HANDLE 类型，如果失败返回 NULL，如果成功返回所获得的堆的句柄。

（3）GetProcessHeaps。

获得进程中所有堆，包括堆的数量和各个堆的句柄，其函数原型如下：

```
DWORD GetProcessHeaps(
  DWORD NumberOfHeaps,
  PHANDLE ProcessHeaps
);
```

◇参数

NumberOfl-Ieaps:输入参数，缓冲区 ProcessHeaps 所能存储的句柄值的个数。

ProcessHeaps：输出参数，指向用于保存进程中所有堆的句柄的内存块，函数返回的句柄值会在其中依次保存。

◇返回值

DWORD 类型的数据，进程中堆的个数。如果返回 0，则表示失败。

◇使用说明

如果返回值大于参数 NumberOfl-Ieaps，说明进程中实际的堆数量大于 NumberOfl-Ieaps，用于

存储句柄值的内存区域太小，ProcessHeaps 所指向的内存块没有句柄值。如果返回值不大于 NumberOfl-Ieaps，那么 ProcessHeaps 保存了所有堆的句柄值。

本 API 函数在实例 5-1 演示使用方法。

（4）HeapAllc。

从指定的堆上分配内存块，其函数原型如下：

```
LPVOID HeapAlloc(
  HANDLE hHeap,
  DWORD dwFlags,
  SIZE_T dwBytes
);
```

◇参数

hHeap:输入参数，表明从此参数指定的堆上进行内存分配。

dwFlags：输入参数，内存分配的标志，其值见表 5.7。可以使用“1”指定多个标志。

dwBytes：输入参数，需要分配的内存区域大小，以字节为单位。

◇返回值

表 5.7 **dwFlags 参数标志**

标　志	意　义
HEAP_GENERATE_EXCEPTIONS	如果分配错误将会抛出异常，而不是返回 NULL。异常值可能是 STATUS_NO_MEMORY，表示可获得的内存容量不足，或是 STATUS_ACCESS_VIOLATION，表示存取不合法
HEAP_NO_SERIALIZE	不使用连续存取
HEAP_ZERO_MEMORY	将分配的内存块中的内容全部清零

LPVOID 类型的数据，如果成功，是指向新分配的内存的指针；如果失败，则为 NULL。

(5)HeapReAlloc。

重新分配内存，改变已经分配好的堆内存块的大小，其函数原型如下：

```
LPVOID HeapReAlloc(
  HANDLE hHeap,
  DWORD dwFlags,
  LPVOID lpMem,
  SIZE_T dwBytes
);
```

◇参数

hHeap：输入参数，表明从此参数指定的堆上进行内存再分配。

dwFlags：输入参数，内存分配的标志，表 5.7 中的，3 个值都可以使用，此外还可以使用 HEAP_REALLOC_IN_ PLACE_ONLY，HEAP_REALLOC_IN_ PLACE_ONLY 表示重新分配内存的位置不要发生变化，否则函数可能将原来内存中的数据进行移动。内存块大小调整后，起始位置可能会发生变化。HEAP_ZERO_MEMORY 值与 HeapAlloc 中的作用有所不同，如果设置了此标志，那么新增加的内存将分被清零，原内存块中的数据不会发生变化。

lpMem：输入参数，原内存块的地址。

dwBytes：输入参数，内存块大小调整后的值，可以比原来大，也可以小。

◇返回值

D 类型的数据，如果成功，是指向大小调整后的内存块的指针，如果失败，则为 NULL。

（6）GetSystermInfo。

获取系统信息，包括 OEM 信息、内存分页大小、内存分配粒度、处理器信息、内存空间最大值与最小值等，其函数原型如下：

```
void GetSystemInfo(
  LPSYSTEM_INFO lpSystemInfo
);
```

◇参数

lpSystemInfo：输出参数，指向 SYSTEM INFO 结构的指针，保存获取的信息。

SYSTEM- INFO 结构的定义详见 8.2.2 小节。

◇无返回值

2．实例 5-1　获得堆句柄并分配内存

本实例使用了多种方法来获得堆句柄，然后在堆上进行内存的分配与再分配。实例会根据用户的输入不同，而采用不同的方法获得堆句柄，一种是指定了堆的初始化大小和最大值，创建一种不可增长的堆，在此堆上进行内存分配如果超过了设置的最大值，将会失败；第二种是获取进程中已经存在的堆，直接使用；第三种是创建一种默认初始化大小（1 个内存分页），并且最大值不受限制的“可增长”的堆。

实例 5-2 与实例 5-3 均为 heap.exe 的代码。将在实例 5-3 之后演示 heap.exe 运行的效果。

```
SYSTEM_INFO si;     //系统信息
HANDLE hHeap;       //堆句柄
LPVOID lpMem;       //内存块指针
LPVOID lpReAlloc;   //内存块大小调整后的指针
DWORD dwHeapSize;   //堆内存埠大小
HANDLE hHeaps[24]; //用于保存进程中所有的堆句柄
DWORD dwHeapNum;    //进程中堆的数量
//获取系统信息
GetSystemInfo(&si);
//将系统内存分页大小，与内存分配粒度打印出来
printf("系统内存页大小: 0x%x\n 系统内存分配粒度:0x%x\n",
    si.dwPageSize,si.dwAllocationGranularity);
//分析输入参数，如果是“-a”,创建一个最大为 10 个分页大小的堆
if(argc == 2 && 0==lstrcmp(argv[1],"-a"))
{
      hHeap = HeapCreate(HEAP_NO_SERIALIZE,
          si.dwPageSize,si.dwPageSize*10);
      printf("创建堆,初\始化大小为 1 页,最大为 10 页 n");
}
 //如果输入参数为“-s”,使用进程初始化时已经存在的堆
else if(argc == 2 && 0==lstrcmp(argv[1],"-s"))
{
       hHeap = GetProcessHeap();
       printf("获取系统已经存在的堆\n");
}
//如果输入其他，创建一个可增长的堆
else
{
       hHeap = HeapCreate(HEAP_NO_SERIALIZE,0,0);
       printf("创建堆,初始化大小为 1 页,大小可变\n");
}
//判断堆是否创建/获取成功
if(hHeap == NULL)
{
    printf("创建或获取进程堆错误: %d",GetLastError());
    return 1;
}
 //获取将打印当前进程中一共有多少个堆，有没有新建堆
dwHeapNum = GetProcessHeaps(24,hHeaps);
if(dwHeapNum == 0)
{
    printf("GetProcessHeaps error: %d",GetLastError());
}
else
{
    printf("当前进程一共有%u 个堆\n",dwHeapNum);
}
//在堆上分析内存，3 个页面大小
lpMem = HeapAlloc(hHeap,HEAP_ZERO_MEMORY,si.dwPageSize*3);
if(lpMem == NULL)
{
    printf("HeapAlloc error: %u",GetLastError());
    return 1;
}
printf("在堆上成功分配内存,起始地址为:0x%x\n",lpMem);
//打印当前堆内存块的大小
PrintHeapSize(hHeap,lpMem);
```

```
//再分配内存，调整内存的大小为11个分页大小
//如果使用第一种方法创建堆，这里会出错
lpReAlloc = HeapReAlloc(hHeap, HEAP_ZERO_MEMORY, lpMem, si.dwPageSize*11);
if(lpReAlloc == NULL)
{
      printf("HeapReAlloc error: %d",GetLastError());
      return 1;
}
printf("在堆上再分配内存,地址为:0x%x,原地址:0x%x\n",lpReAlloc,lpMem);
//打印调整大小后的堆内存块大小
PrintHeapSize(hHeap,lpReAlloc);
```

从代码中可以发现，如果用户使用参数“-a”，程序将会创建一个不可增长的堆；如果使用参数“-s”，将会使用系统已经存在的堆；如果是其他参数，将会创建一个可增长的堆。

代码中的 PrintHeapSize 函数用于打印堆内存块的大小，详见 5.2.2 小节中的实例 5-2。

5.2.2 获取堆中内存块的大小信息

API 函数 HeapSize 的作用是获取堆中内存块的大小信息。使用这个函数，可以验证实例 5-1 中分析的内存的大小变化。

1.关键 API

HeapSize 用于获取指定堆的大小，以字节为单位返回堆大小信息，其函数原型如下：

```
SIZE_T HeapSize(
  HANDLE hHeap,
  DWORD dwFlags,
  LPCVOID lpMem
);
```

◇参数

hHeap:输入参数，需要获得大小的内存块所在堆的句柄。

dwFlags:输入参数，内存块的标志，可以是 HEAP_NO_SERIALIZE，参见表 5.7。

lpMem:输入参数，所需要获取大小的内存块指针。

◇返回值

SIZE_T 类型，表示内存块的大小，如果失败则为-1。

2．实例 5-2 PrintHeapSize 函数

本实例是实例 5-1 中 PrintHeapSize 函数的源代码，代码源同样位于 heap.c 中。

```
/*************************************
* DWORD PrintHeapSize(HANDLE hHeap,LPVOID lpMem)
* 功能：获取堆的大小并打印出来
* 参数：HANDLE hHeap，堆句柄
*       LPVOID lpMem，内存地址指针
* 返回值：0代表执行完成，1代码发生错误
*************************************/
DWORD PrintHeapSize(HANDLE hHeap,LPVOID lpMem)
{
  SIZE_T dwHeapSize;
  dwHeapSize = HeapSize(hHeap,HEAP_NO_SERIALIZE,lpMem);
  if(dwHeapSize == -1)
  {
      printf("Get HeapSize error :%d",GetLastError());
      return 1;
  }
```

```
        printf("内存块大小为:0x%x\n",dwHeapSize);
        return 0;
    }
```

函数将指定的内存块的大小信息使用十六进制数据打印出来，成功则返回 0，失败则返回 1。程序的运行效果见 5.2.3 小节。

5.2.3 释放内存、销毁堆

分配的内存在使用完成后需要释放，否则内存将不能再投入使用，造成内存泄漏。使用 HeapAlloc 和 HeapReAlloc 函数分配的内存需使用 HeapFree 释放。

堆还可被创建，也可以被销毁，在程序中，如果一个堆不再使用，可以使用 HeapDestroy 函数进行销毁。HeapDestroy 函数将本堆所占用的所有虚拟内存分页全部释放（虚拟内存分页参见 5.4 节），相当于释放于从本堆上分配的所有的内存块。

1．关键 API

（1）HeapFree。

释放 HeapAlloc 和 HeapReAlloc 所分配的内存，其函数原型如下：

```
BOOL HeapFree(
  HANDLE hHeap,
  DWORD dwFlags,
  LPVOID lpMem
);
```

◇参数

hHeap：输入参数，内存块所在的堆，需使用与 HeapAlloc 和 HeapReAlloc 分配内存时同样的堆。

dwFlags：输入参数，内存块标志，可以设置为 HEAP_NO_SERIALIZE。

lpMem：输入参数，所需要释放的内存块的指针。

◇返回值

返回 BOOL 值，表示是否成功。

(2)HeapDestroy。

销毁由 HeapCreate 创建的堆，其函数原型如下：

```
BOOL HeapDestroy(
  HANDLE hHeap
);
```

◇参数

hHeap：输入参数，所需要销毁的堆的句柄。

◇返回值

返回 BOOL 值，表示是否成功。

2．实例 5-3 释放堆中分配的内存

本实例使用 HeapFree 函数释放了在实例 5-1 中分配的内存，使用 HeapDestroy 销毁了在实例 5-1 中创建的堆。

```
    //释放内存
    if(!HeapFree( hHeap, HEAP_NO_SERIALIZE, lpReAlloc))
    {
        printf("HeapFree error: %d",GetLastError());
        return 1;
```

```
    }
    printf("释放内存成功\n");
    //如果新建了堆，销毁堆
    if(argc != 2 || 0!=lstrcmp(argv[1],"-s"))
    {
        printf("销毁 HeapCreate 创建的堆\n");
        if(!HeapDestroy(hHeap))
        {
            printf("HeapDestroy error: %d",GetLastError());
            return 1;
        }
        printf("销毁堆成功\n");
    }
    return 0;
```

实例 5-1、5-2、5-3 组成了配套光盘中 heap.c 的完整代码。编译源文件，得到 heap.exe。heap.exe 的运行效果如下：

第一次运行：

```
heap.exe -a
```

结果：

```
系统内存页大小: 0x1000
系统内存分配粒度:0x10000
创建堆,初始化大小为 1 页,最大为 10 页
当前进程一共有 5 个堆
在堆上成功分配内存,起始地址为:0x3a0650
内存块大小为:0x3000
HeapReAlloc error: 8
```

第二次运行：

```
heap.exe
```

结果：

```
系统内存页大小: 0x1000
系统内存分配粒度:0x10000
创建堆，初始化大小为 1 页,大小可变
当前进程一共有 5 个堆
在堆上成功分配内存,起始地址为:0x3a0650
内存块大小为:0x3000
在堆上再分配内存,地址为:0x3a3658,原地址:0x3a0650
内存块大小为:0xb000
释放内存成功
销毁 HeapCreate 创建的堆
销毁堆成功
```

第三次运行：

```
heap.exe -s
```

结果：

```
系统内存页大小: 0x1000
系统内存分配粒度:0x10000
获取系统已经存在的堆
当前进程一共有 4 个堆
在堆上成功分配内存，起始地址为:0x153098
内存块大小为:0x3000
在堆上再分配内存，地址为:0x1560a0，原地址:0x153098
内存块大小为:0xb000
释放内存成功
```

三次运行输入了不同的参数，分析 3 次运行的不同结果。

◇三次运行结果获得的系统信息都是一致的。分页大小都是 0x1000 (4K)字节。

◇第一次运行由于创建的堆的大小最大为 10 个分页，也就是 0xA000 字节，所以分配 0x3000 大小的内存块成功，但是将内存大小调整为 0xB000 时，发生了错误，错误代码为“8”，在 ErrLook.exe 工具中查询，错误信息是“存储空间不足，无法处理此命令”。

◇第二次运行，由于创建的堆是可增长的，所以无论分配 0x3000 大小的内存，还是分配

0xB000 大小的内存，都成功。
◇第一次和第二次运行由于新创建了堆，所以比第三次运行堆的数量多一个。
◇　内存释放与堆销毁操作都成功。获取堆内存块大小的操作成功，结果正确。

5.3　全局(Global)和局部(Local)内存管理

32 位 Windows 系统中并没有区分全局堆和局部堆，Windows 系统中之所以还存在着全局和局部函数的概念是为了和 16 位系统相兼容。GlobalAlloc、LocalAlloc 函数所分配的内存没有什么区别，内存分配的效果与 HeapAlloc 函数类似。但是 GlobalAlloc 与 LocalAlloc 分配内存时比 HeapAlloc 要慢。

因此更推荐软件开发人员使用 Heap 函数。本书只是简单介绍如何使用全局和局部函数来分配内存，不推荐读者使用。

5.3.1　Global函数

(1) GlobalAlloc、GlobalFree、GlobalReAlloc

GlobalAlloc 和 GlobalFree 是一对分配和释放内存函数。使用 GlobalAlloc 分配的内存使用 GlobalFree 函数释放，不同于 HeapAlooc 函数，GloalAlloc 函数会采用默认堆，不需要指定堆句柄，只需要指定分配属性与分配的内存大小即可。参数设置和使用较为简单。GlobalAlloc 函数原型如下：

```
HGLOBAL GlobalAlloc(
  UINT uFlags,
  SIZE_T dwBytes
);
```

参数 uFlags 为分配的属性，参数 dwBytes 为需要分配的内存大小，以字节为单位。参数 uFlags 和返回值的具体情况见表 5.8。

GlobalFree 函数释放 GlobalAlloc 所分配的内存，以 GlobalAlloc 函数的返回值为参数，GlobalFree 返回 NULL 表示成功，函数的原型如下：

GlobalReAlloc 函数的作用是调整由 GlobalAlloc 函数分配的内存的大小和属性，函数原型如下：

```
HGLOBAL GlobalFree(
  HGLOBAL hMem
);
```

表 5.8　uFlags 参数选项

选　项	意　义
GMEM_FIXED	分配固定内存，返回为指向内存的指针
GMEM_MOVEABLE	可以在堆内移动，地址不固定，返回值指向内存对象句柄。使用 GlobalLock 函数可以得到固定地址指针，不能与 GMEM_FIXED 同时使用
GMEM_ZEROINIT	把分配的内存清零
GPTR	GMEM_FIXED | GMEM_ZEROINIT
GHND	GMEM_MOVEABLE |GMEM_ZEROINIT

```
HGLOBAL GlobalReAlloc(
  HGLOBAL hMem,
  SIZE_T dwBytes,
  UINT uFlags
);
```

参数 hMem 是 GlobalAlloc 的返回值，参数 dwBytes 是新的内存区域的大小，uFlags 是新的属性，值参见表 5.8。

(2) GlobalLock、GlobalHandle.

GlobalLock 函数的作用是使 GlobalAlloc 函数或 GlobalReAlloc 函数分配得到的 GMEM MOVEABLE 属性的内存对象句柄固定，并返回内存指针。

GlobalHandle 函数的作用与 GlobalLock 函数的作用正好相反。

这两个参数的原型如下：

```
HGLOBAL GlobalHandle(
  LPCVOID pMem
);
LPVOID GlobalLock(
  HGLOBAL hMem
);
```

(3) GlobalFlags、GlobalDiscard、GlobalUnlock

这 3 个函数（宏）的作用是获取和修改内存的属性，Global 内存的属性包括 GMEM MOVEABLE、GMEM FIXED、GMEM_ DISCARDED、GMEM DISCARDABLE 等，还包括锁定数量(lock count)等概念，现在已经很少使用到了。不需要过多地了解。

GlobalLock、GlobalHandle 函数也会影响到内存的属性，但是它们还会影响到内存的指针和内存的访问方式问题。

```
UINT GlobalFlags(
  HGLOBAL hMem
);
```

(4) GlobalSize。

GlobalSize 函数的作用是获取由内存块的大小，函数原型如下：

```
SIZE_T GlobalSize(
  HGLOBAL hMem
);
```

> **注意** GlobalMemoryStatus 和 GlobalMemoryStatusEx 函数并不是属于这里所说的 Global 函数，而是实现对系统全局内存的状态的查询。将会在 5.5.2 小节介绍到。

5.3.2 Local函数

Local 函数与 Global 函数是一一对应的，功能也是一致的，包括 LocalAlloc、LocalFree、LocalReAlloc、LocalLock、LocalHandle、LocalDiscard、LocalFlags、LocalUnlock、LocalSize.

各函数的使用方法与 Global 函数中对应的函数类似。部分函数也涉及了一些常量和数据类型，在需要使用时，可以与 Global 函数的常量和数据类型相类比，一般将 Global 函数所使用的数据类型名前的字母“G”变换为字母“L”就可以使用了。

Local 与 Global 函数功能相似使用方法相同，这里不再详细介绍。

5.3.3 使用全局和局部函数分配和释放内存、改变内存块属性

实例 5-4 演示了全局内存管理函数的使用，局部内存管理函数的使用方法与此类似。

实例 5-4 全局和局部函数分配内存演示

本实例使用了 Global 函数分配内存，由于在 32 位 Windows 系统上已经没有全局和局部函数的概念，也不区分移动和固定的概念，所涉及的一些函数已经废弃不用了。但是在一些代码中还是会使用 Global 函数操作内存，因为 GlobalAlloc 函数的参数设置是几种内存分配方法中最简单的。

```
/* ************************************
* global.c 使用全局和局部函数分配和释放内存
**************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
/************************************
* int main(void)
* 功能：演示 Global*函数的使用
************************************/
int main(void)
{
    LPVOID lpMem;      //内存地址
    HGLOBAL hMem;      //内存句柄
    SIZE_T sizeMem;    //内存大小
    UINT uFlags;       //属性
    //分配内存
    lpMem = (LPVOID)GlobalAlloc(GPTR,1000);
    //将数据复制到内存中
    lstrcpy(lpMem,"this is a string");
    //获得内存属性，打印
    uFlags = GlobalFlags(lpMem);
    printf("内存中的内容：\"%s\"，内存地址：0x%.8x，内存属性：%u\n",
        lpMem,lpMem,uFlags);
    //释放
    GlobalFree(lpMem);
    //分配内存，获取信息
    hMem = GlobalAlloc(GMEM_MOVEABLE,1000);
    sizeMem = GlobalSize(hMem);
    uFlags = GlobalFlags(hMem);
    //打印信息
    printf("内存大小：%d，内存句柄：0x%.8x，内存属性：%u\n",
        sizeMem,hMem,uFlags);
    //锁定
    lpMem = GlobalLock(hMem);
    //释放
    GlobalFree(hMem);
    return 0;
}
```

从程序的运行结果可以看出，使用 GMEM_MOVEABLE 和 GMEM_FIXE 分配的内存使用 GlobalFlags 获得的属性是一样的，都是 0。

实例运行结果如下：

```
内存中的内容："this is a string"，内存地址：0x001530b8，内存属性：0
内存大小：1000，内存句柄：0x00430003，内存属性：0
```

5.4 虚拟内存管理

在 5.1 节中已经详细介绍了系统内存管理的机制。本节将介绍若干个函数，用于在管理进程的虚拟内存。

5.4.1 虚拟地址空间与内存分页

5.1 节所述的内存管理的原理都由系统的内存管理器实现，应用程序只能管理属性于自己的虚拟地址空间。

进程的虚拟地址空间内存页面存在 3 种状态，分别为空闲的(free)、保留的(reserved)和提交的（Committed），见表 5.9。大多数情况下，一页的大小是 4KB。

表 5.9 虚拟地址内存页面的 3 种状态

状　态	说　明
空闲的（Free）	进程不能访问这种页面，因为这此页面还没有被分配，任何属于这种页面的虚拟内存地址进行访问都将引用访问异常
保留的（Reserved）	页面被保留以被将来之用。这些页面已经被分配，但是还没有使用，物理地址空间中的内存中不存在其对应的物理内存分页。处理保留状态的内存分页也不能被访问
提交的（Committed）	内存已经被分配，并且已经被使用，具有与之对应的物理地址空间中的内存分页

5.4.2 分配和释放可读可写的虚拟内存页面

在使用 VirtualAlloc 和 VrtualAllocEx 分配虚拟内存时，可以指定分配的内存页面是保留的(Reserved)还是提交的(Committed)。如果直接分配内存为“提交的”，那么在分配完成后就可以对内存进行访问。

1．关键 API

(1) VirtualAlloc。

函数的功能是“保留”或“提交”内存页面，将“空闲的”内存页面变为“保留的”或“已提交的”，将“保留的”页面变为“提交的”，函数原型如下：

```
LPVOID VirtualAlloc(
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD flAllocationType,
  DWORD flProtect
);
```

◇参数

lpAddress:输入参数，分配的起始位置。如果是要保留一段内存区域，那么函数会将其自动向最近的一个分配粒度对齐；如果是要提留一段内存区域，那么函数将会向最近的一个页面对齐。

如果为 NULL，那么系统将自行决定在什么地方分配。

dwSize:输入参数，所需要分配的内存区域的大小。

flAllocationType:输入参数，分配的类型，决定了是保留一段内存区域，还是提交，还是同时完成。如果设置为 MEM_COMMIT，那么将提交内存页面；如果设置为 MEM RESERVE，那么将保留页面；如果设置为 MEM_COMMIT | MEM RESERVED，那么将直接从空闲页面提交为“已提交的”页面。

flProtect:输入参数，内存的保护属性，其值可以是 PAGE_READWRITE、PAGE_ EXECUTE、PAGE_ EXECUTE READ、PAGE_ EXECUTE READWRITE、PAGE_EXECUTE_WRITECOPY、PAGE NOACCESS、PAGE WRITECOPY。

◇返回值

返回 LPVOID 类型的值，表示分配到的内存的起始地址。如果返回 NULL，则表示失败。可使用 GetLastError 函数获取错误信息。

(2) VirtualFree。

将内存状态从“已提交的”变为“保留的”，或将“保留的”变为“空闲的”，或同时进行，此函数用于进行 VirtualAlloc 的反向工作，函数原型如下：

```
BOOL VirtualFree(
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD dwFreeType
);
```

◇参数

lpAddress:输入参数，需要改变状态的内存区域的起始地址。

dwSize:输入参数，需要改变状态的内存区域的大小，以字节为单位。

dwFreeType:输入参数，如果此参数设置为 MEM__ DECOMMIT，则将内存变为保留状态，当 dwSize 参数为 0，则 lpAddress 参数必须为内存区域的起始地址（VirtualAlloc 的返回值），将整个区域设置为保留状态。如果此参数设置为 MEM RELEASE，则释放内存，将内存直接变为空闲状态，不管当前是处理保留状态还是已提交状态。如果设置为 MEM_RELEASE，dwSize 参数必须为 0，则 lpAddress 参数必须为内存区域的起始地址（Virtual_Alloc 的返回值）。两个值不能一起使用。

◇返回值

返回 BOOL 类型的值，表示执行是否成功。如果表示失败。可使用 GetLastError 函数获取错误信息。

(3) VirtualFreeEx。

释放由 VirtualAllocEx 分配的内存，功能、使用方法和 VirtualFree 类似，函数原型如下：

```
BOOL VirtualFreeEx(
  HANDLE hProcess,
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD dwFreeType
);
```

注意 使用 VirtualAlloc、VitualFree 等函数进行操作，其操作的基本单位是内存页面，函数会改变整个页面的状态，在设置 lpAddress 参数和 dwSize 参数时需注意。比如从 lpAddress 到（lpAddress+dwSize）内存区域跨越了两个页面，那么这两个页面的状态都会改变，即使这段内存区域很小。

2．实例 5-5 分配和释放虚拟内存

本实例调用 VirtualAlloc 函数，lpAddress 参数设置为 NULL，使系统自行决定在什么地址分配；将 flAllocationType 参数设置为 MEM_COMMIT I MEM_RESERVED 参数，直接将内存从空间状态分配为可使用的已提交状态内存，同时指定内存区域的保护属性为 PAGE_READWRITE，完成内存分配的工作。

分配完成后，使用 VirtualQuery 函数获取内存信息，并打印出来。然后使用 MEM_DECOMMIT 参数调用 VirtualFree 函数将虚拟内存释放，设置为保留状态，再获取保存状态

时内存信息，区别不同内存状态下内存信息的不同。

```
/* ************************************
* virtual.c  分配和释放可读可写的虚拟内存页面
************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
/************************************
* int main(void)
* 功能：演示虚拟内存的使用
************************************/
int main(void)
{
    SIZE_T sizeVirtual = 4000;          //大小
    LPVOID lpRound = (LPVOID)0x100000FF;    //地址
    MEMORY_BASIC_INFORMATION  mbi;    //内存信息

    //分配内存，直接分配已提交的内存
    LPVOID lpAddress = VirtualAlloc(
        lpRound,sizeVirtual,
        MEM_COMMIT|MEM_RESERVE,PAGE_READWRITE
        );
   if(lpAddress == NULL)
    {
        printf("VirtualAlloc error: %d\n",GetLastError());
        return 1;
    }
   printf("Alloc:MEM_COMMIT|MEM_RESERVE\n");
   //复制数据到内存中
   CopyMemory(lpAddress,"hello",lstrlen("hello"));
   printf("分配、复制成功，地址：0x%.8x，内容：%s\n",
        lpAddress,lpAddress);
   //获取内存信息并打印
   VirtualQuery(lpAddress,&mbi,sizeof(mbi));
   printf("使用 VirtualQuery 获得的信息：\n"
        "BaseAddress:0x%.8x\tAllocationBase:0x%.8x\t"
        "AllocationProtect:0x%.8x\tRegionSize:%u\t"
        "State:0x%.8x\tProtect:0x%.8x\tType:0x%.8x\n",
        mbi.BaseAddress,mbi.AllocationBase,
        mbi.AllocationProtect,mbi.RegionSize,
        mbi.State,mbi.Protect,mbi.Type
        );

    //DECOMMIT 释放，页面将变为保留状态
    printf("Free: DECOMMIT\n");
    if(!VirtualFree(lpRound,sizeVirtual,MEM_DECOMMIT))
    {
        printf("VirtualFree error: %d",GetLastError());
        return 1;
    }
   //获取内存信息并打印
   VirtualQuery(lpAddress,&mbi,sizeof(mbi));
   printf("使用 VirtualQuery 获得的信息：\n"
        "BaseAddress:0x%.8x\tAllocationBase:0x%.8x\t"
        "AllocationProtect:0x%.8x\tRegionSize:%u\t"
        "State:0x%.8x\tProtect:0x%.8x\tType:0x%.8x\n",
        mbi.BaseAddress,mbi.AllocationBase,
        mbi.AllocationProtect,mbi.RegionSize,
        mbi.State,mbi.Protect,mbi.Type
        );
    //释放内存
    printf("Free:RELEASE\n");
    if(!VirtualFree(lpAddress,0,MEM_RELEASE))
```

```
        {
            printf("VirtualFree error: %d",GetLastError());
            return 1;
        }
            return 0;
    }
```

运行结果：

```
    Alloc:MEM_COMMIT|MEM_RESERVE
    分配、复制成功，地址：0x10000000，内容：hello
    使用 VirtualQuery 获得的信息：
    BaseAddress:0x10000000  AllocationBase:0x10000000  AllocationProtect:0x00000004 Region
Size:8192  State:0x00001000 Protect:0x00000004 Type:0x00020000
    Free: DECOMMIT
    使用 VirtualQuery 获得的信息：
    BaseAddress:0x10000000 AllocationBase:0x10000000   AllocationProtect:0x00000004
RegionSize:8192    State:0x00002000   Protect:0x00000000 Type:0x00020000
    Free:RELEASE
```

注意分配时指定的地址和大小与分配后地址和大小的变化。在分配时，虽然为 VirtualAlloc 指定的分配地址为 0x10000FF，大小为 4000 (0xFA0)，但是分配完成后，VirtualAlloc 返回的地址是和通过 VirtualQurey 函数获得的起始地址为 0x10000000，大小 8192。产生这种变化的原因是：

0x100000FF (lpAddress) ～ 0x1000109F (lpAddress+ dwSize= 0x100000FF+ 0xFA0= 0x1000109F) 的内存地址跨越了两个虚拟内存分页 0x10000000～0x1000IFFF，是 VirtualAlloc 进行自动对齐的结果，这就是虚拟内存分配的对齐机制。

再来分析一下 VirtualQuery 获得的其他信息的含意，在内存分页处于已提交状态时和处于保留状态时，状态(State)分别为 0x1000 和 0x2000，保护属性(Protect)分别为 0x4 和 0x0，分配时的保护属性(AllocationProtect)都是 0x4，类型均为 0x20000。在 winnt.h 头文件中能找到这些常量的定义，如下所示：

```
    #define PAGE_READWRITE          0x04
    ......
    #define MEM_PRIVATE             0x20000
    ......
    #define MEM_COMMIT              0x1000
    #define MEM_RESERVE             0x2000
```

状态值为 0x2000 时，说明其为保留页面（也有可能是 MEM_ FREE 0x10000，说明是空闲页面)。保护属性为 0x04，说明是可读可写的，在保留状态时，其保护属性是 0x00，内存的类型值有几种可能，分别为 MEM IMAGE(0x1000000)、MEM MAPPED(0x40000)、MEM PRIVATE(0x20000)，前两种说明内存是通过内存映射共享的，MEM PRIVE 说明内存页面没有与其他进程共享。

> **注意** 实例代码中第一次调用 VirtualFree 函数与第二次调用 VirtualFree 函数时参数的设置不同，第二次调用时，地址为 VirtualAlloc 返回的经过对齐的页基址，大小为 0，在使用 MEM_RELEASE 参数释放内存时，必须这样设置。

5.4.3 修改内存页面状态和保护属性、将页面锁定在物理内存中

实例 5-6 中已经演示了如何使用 VirtualAlloc 和 VirtualFree 函数改变内存页面状态，还可以使用 VirtualProtect API 函数来改变页面的保护属性。

将页面锁定在物理内存中使用 VirtualLock 函数。这样页面将不会被交换到硬盘上，提高了访状态值为 0x1000 时，说明其为已提交的页面；

问的效率，如果要解除锁定，可以使用 VirtualUnLock。

1．关键 API

(1) VirtualProtect。

改变指定虚拟内存分页的保护属性，函数原型如下：

```
BOOL VirtualProtect(
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD flNewProtect,
  PDWORD lpflOldProtect
);
```

◇参数

lpAddress:输入参数，虚拟内存基地址。

dwSize:输入参数，内存大小。

flNewProtect:输入参数，需要设置的新的属性，具体为 PAGE READONLY、PAGE EXECUTE、PAGE_ EXECUTE_ READ、PAGE_GUARD、PAGE NOACCESS 和 PAGE_ NOCACHE.

lpflOldProtect：输出参数，指向保存原保护属性值的 DWORD 变量，可以为 NULL。

◇返回值

返回 BOOL 值，表示是否成功，可以使用 GetLastError 函数获取错误代码。

(2) VirtualLock。

```
BOOL VirtualLock(
  LPVOID lpAddress,
  SIZE_T dwSize
);
```

(3) VirtualUnlock。

```
BOOL VirtualUnlock(
  LPVOID lpAddress,
  SIZE_T dwSize
);
```

2．实例 5-6 修改内存页面状态

本实例将内存属性设置为只读，对内存的写入操作将会引起程序访问异常。

```
//设置为 READ-ONLY 属性
if(!VirtualProtect(lpAddress,0, READ-ONLY,NULL))
{
      printf("VirtualProtect error: %d",GetLastError());
      return 1;
}
//获取内存信息并打印
VirtualQuery(lpAddress,&mbi,sizeof(mbi));
printf("使用 VirtualQuery 获得的信息: \n"
      "BaseAddress:0x%.8x\tAllocationBase:0x%.8x\t"
      "AllocationProtect:0x%.8x\tRegionSize:%u\t"
      "State:0x%.8x\tProtect:0x%.8x\tType:0x%.8x\n",
      mbi.BaseAddress,mbi.AllocationBase,
      mbi.AllocationProtect,mbi.RegionSize,
      mbi.State,mbi.Protect,mbi.Type
      );
```

5.4.4 管理其他进程的虚拟内存

5.4.1～5.4.3 小节已经介绍了用于虚拟内存管理的 API 函数，这些函数都只能管理本进程的虚拟内存，如果要管理其他进程的虚拟内存可以使用与虚拟内存管理函数相对应的扩展函数，扩展函数名在原函数名的基础上使用“Ex”后缀，如 VirtualAllocEx、VirtualFreeEx、

VirtualProtectEx 等。

关键 API

以 VirtualAllocEx 为例说明，其他扩展 API 与可以对应相应的 API 使用，扩展 API 都会多一个参数，目录进程的句柄。当然，需要有相应的权限(PROCESS_VM_OPERATION)才可以对其他进程进行操作。

VirtualAllocEx 与 VirtualAlloc 函数功能类似，可以为其他进程分配内存，函数原型如下：

```
LPVOID VirtualAllocEx(
  HANDLE hProcess,
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD flAllocationType,
  DWORD flProtect
);
```

hProcess 参数指定内存，将会在这个进程的虚拟内存空间中分配内存。

5.5 内存操作与内存信息管理

现在已经了解了 Windows 系统是如何管理内存的，以及各种分配与释放的方法。分配、释放内存最终目的还是要使用内存，本节的主要内容是如何使用内存，如复制、填充等，还将演示如何获取系统全局的内存使用情况，使用内存写入监视，判断指针的可用性等。

5.5.1 复制充、移动、清零内存块、防止缓冲区溢出

实例 5-7 和 5-8 说明了在分配完内存后，使用内存存储信息时的基本操作，包括复制内存、移动内存中的内容、清零等。

1．关键 API

（1）CopyMemory。

复制内存，第一个参数为目的地址，第二个参数为源地址，第三个参数为复制数据的大小，单位字节，源内存区域与目的内存区域不能重叠，如果重叠，可以使用 MoveMemory 函数。函数原型如下：

```
void FillMemory(
  PVOID Destination,
  SIZE_T Length,
  BYTE Fill
);
```

（2）FillMemory。

填充内存，将一段内存填充为同一个值，第一个参数为需要填充的内存地址，第二个参数是填充的大小，单位字节，第三个参数是填充的值，BYTE 类型，FillMemory 会使用这个值填满指定的内存。

```
void CopyMemory(
  PVOID Destination,
  const VOID* Source,
  SIZE_T Length
);
```

（3）MoveMemory。

复制内存，第一个参数为目的地址，第二个参数为原地址，第三个参数为复制数据的大小，

单位为字节，源内存区域与目的内存区域可以重叠。函数原型如下：

```
void MoveMemory(
  PVOID Destination,
  const VOID* Source,
  SIZE_T Length
);
```

（4）ZeroMemory

将指定内存清零，第一个参数为内存地址，第二个参数为内存区域的大小，单位字节。函数原型如下：

```
void ZeroMemory(
  PVOID Destination,
  SIZE_T Length
);
```

2．实例 5-7 对内存进行复制、移动、填充等操作

本实例在堆中分配了两小段内存，然后对内存进行了各种操作。首先分配了内存但不清零，查看内存可以看到是一些未知的数据，然后使用 ZeroMemory 清零，可以发现内存中的数据全部变为了零；再两次使用 FillMemory 在内存的前半部分和后半部分填充入不同的数据，并使用 CopyMemory 复制到目标内存中，并显示目录内存块中的数据，最后释放内存。

本实例中的 ShowMemContent 函数代码见实例 5-8。

```
/* 常量定义 */
#define MEM_BLOCK_SIZE 32

/*************************************
* int main(void)
* 功能：获取内存使用情况
*************************************/
int main(void)
{
     HANDLE hHeap = GetProcessHeap();
     LPVOID lpSrc;
     LPVOID lpDis;
     //分配内存
     lpSrc = HeapAlloc(hHeap,0,MEM_BLOCK_ SIZE);
     lpDis = HeapAlloc(hHeap,0,MEM_BLOCK_ SIZE);
     //显示新分配的内存
     printf("HeapAlloc 分配但不清零：\n");
     ShowMemContent(lpDis, MEM_BLOCK_ SIZE);
     //将内存清零并显示
     ZeroMemory(lpDis,MEM_BLOCK_ SIZE);
     printf("ZeroMemory 清零：\n");
     ShowMemContent(lpDis, MEM_BLOCK_ SIZE);
     //使用 0xBB 填充全部内存
     FillMemory(lpSrc,MEM_BLOCK_ SIZE,0xBB);
     //将内存块的前半部分使用 0xAA 填充
     FillMemory(lpSrc,MEM_BLOCK_ SIZE/2,0xAA);
     CopyMemory(lpDis,lpSrc,MEM_BLOCK_ SIZE);
     printf("FillMemory 有规律地填充内存：\n");
     ShowMemContent(lpDis,MEM_BLOCK_SIZE);

     //释放内存
     HeapFree(hHeap,0,lpSrc);
     HeapFree(hHeap,0,lpDis);
     return 0;
}
```

3．实例 5-8　ShowMemContent 函数内存打印输出

本实例是实例 5-8 中 ShowMemContent 函数的代码，功能是将内存中的数据按字节依次打印到界面上。函数定义了数组局部变量，将需要打印的内存复制到了局部数组变量中，为了防止栈溢出还检查了内存块的长度，然后使用循环将数据分组打印输出。

```
/************************************
* BOOL ShowMemContent(LPVOID lpMem,SIZE_T dwSize)
* 功能：显示内存中的内容
* 参数：LPVOID lpMem，需要打印的内存的指针
*       SIZE_T dwSize，大小
* 返回值：BOOL，如果数据过大可能溢出，则返回 FALSE
************************************/
BOOL ShowMemContent(LPVOID lpMem,SIZE_T dwSize)
{
    BYTE lpShow[MEM_BLOCK_ SIZE];
    INT i=0;
    //防止栈溢出
    if(dwSize>MEM_BLOCK_SIZE)
    {
        printf("over-flow");
        return FALSE;
    }
    //复制到数组
    CopyMemory((LPVOID)lpShow,lpMem,dwSize);
    //显示
    for(; i<dwSize; i++)
    {
        printf("%.2X ",lpShow[i]);
        if(!((i+1)%16))
        {
            printf("\n");
        }
    }
    printf("\n");
    return TRUE;
}
```

实例 5-8 和实例 5-9 代码的运行结果如下：

```
HeapAlloc 分配但不清零：
78 01 15 00 78 01 15 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

ZeroMemory 清零：
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

FillMemory 有规律地填充内存：
AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB
```

5.5.2 获得当前系统内存使用情况

当前系统内存使用情况包括总物理内存、可使用的物理内存、总虚拟内存，可用虚拟内存、硬盘上的页面缓存文件大小、可使用的缓存文件大小等。

1．关键 API

GlobalMemoryStatus、GlobalMemoryStatusEx 用于获取系统当前内存使用情况。以 GlobalMemoryStatusEx 为例说明，函数原型如下：

```
BOOL GlobalMemoryStatusEx(
  LPMEMORYSTATUSEX lpBuffer
);
```

◇参数

lpBuffer:输出参数，指向 MEMORYSTATUSEX 结构，用于保存获取的信息。

◇返回值

返回 BOOL 值，表示是否成功，可以使用 GetLastError 函数获取错误代码。

◇使用说明

GlobalMemoryStatus 函数使用 MEMORYSTATUS 结构保存获取的信息，这个结构与 MEMORYSTATUSEX 区别不大。

2．实例 5-9 获取当前系统内存使用情况

本实例调用 GlobalMemoryStatusEx 获取并显示当前系统中内存使用情况，包括总物理内存、可使用的物理内存、总虚拟内存，可使用的虚拟内存、硬盘上的页面缓存文件大小、可使用的缓存文件大小及已经使用了的物理内存的百分比。

```
/*************************************
* int main(void)
* 功能：获取内存使用情况
* 参数：未使用
*************************************/
int main(void)
{
    //用于保存信息
    MEMORYSTATUSEX memstatusex;
    //设置结构大小
    memstatusex.dwLength = sizeof(memstatusex);
    //获取系统内存使用情况
    GlobalMemoryStatusEx(&memstatusex);
    //打印输入结果
    printf("TotalPhys\tAvailPhys\tTotalVirtual\tAvailVirtual\t"
        "TotalPageFile\tuAvailPageFile\tMemoryLoad\n"
        "%I64u\t%I64u\t%I64u\t%I64u\t%I64u\t%I64u\t%u%%",
        memstatusex.ullTotalPhys,memstatusex.ullAvailPhys,
        memstatusex.ullTotalVirtual,memstatusex.ullAvailVirtual,
        memstatusex.ullTotalPageFile,memstatusex.ullAvailPageFile,
        memstatusex.dwMemoryLoad);
    return 0;
}
```

在笔者主机上的运行结果如下：

```
TotalPhys     AvailPhys     TotalVirtual  AvailVirtual  TotalPageFile uAvailPageFile
MemoryLoad
1071161344  395079680  2147352576     2140930048    2577195008    2077577216    63%
```

所有的输出结果都是以字节为单位，都是 64 位无符号整数，MemoryLoad 表示当前正在使用的物理内存百分比。

5.5.3 判断内存指针的可用性

判断内存指针的可用性，是通过判断指针所指向的虚拟内存页面的状态和属性来实现的。Windows 提供了相关的 API。相同的功能也可以通过获取虚拟内存页面属性来判断。

(1) IsBadCodePtr。

判断调用进程是否拥有对指定地址内存的读操作权限，函数原型如下：

```
BOOL IsBadCodePtr(
  FARPROC lpfn
);
```

◇参数

lpsz:输入参数，指向字符串。

ucchMax:输入参数，读取字符串的最大长度。

◇返回值

返回 BOOL 值，表示当前进程是否拥有字符串指针指向的字符串的读取权限。

(4) IsBadWritePtr。

判断调用进程是否拥有对指定地址段内存的写操作权限，函数原型如下：

```
BOOL IsBadReadPtr(
  const void* lp,
  UINT ucb
);
```

◇参数

lp:输入参数，指向起始内存地址。

ucb:输入参数，从起始内存地址开始的内存块长度。

◇返回值

返回 BOOL 值，表示当前进程是否拥有该段地址内存块的读取权限。

(3) IsBadStringPtr。

判断调用进程是否拥有对指定字符串指针的读取权限，函数原型如下：

```
BOOL IsBadStringPtr(
  LPCTSTR lpsz,
  UINT_PTR ucchMax
);
```

◇参数

lpfn:输入参数，指向某指定地址。

◇返回值

返回 BOOL 值，表示当前进程是否拥有该地址内存的读取权限。

(2) IsBadReadPtr。

判断调用进程是否拥有对指定地址段内存的读操作权限，函数原型如下：

```
BOOL IsBadWritePtr(
  LPVOID lp,
  UINT ucb
);
```

◇参数

lp:输入参数，指向起始内存地址。

ucb:输入参数，从起始内存地址开始的内存块长度。

◇返回值

返回 BOOL 值，表示当前进程是否拥有字符串指针指向的字符串的写入权限。

5.6 各种内存分配方式的关系与比较

介绍了 Windows 系统若干种内存分配的方法，再加上标准 C 语言也提供了内存分配和释放的库函数。那么这些函数有什么关系，异同点是什么呢？

5.6.1 标准C内存管理函数与Windows内存管理API的关系

C 语言标准定义了一系列的内存管理函数，如 malloc、free 等。Windiows 系统也支持标准 C。在应用程序中，使用标准 C 内存管理函数可以完成内存的分配与释放等。标准 C 的函数库是由系统提供的。crtdll.dll 中包括各种标准 C 库函数的实现。在 Windows 平台上，使用标准 C 库函数进行内存分配与释放也不会脱离 Windows 系统内存管理的模式。

事实上，Windows 系统中，和大多数标准 C 函数一样，malloc 函数仍然是通过 API 函数实现的。malloc 函数在对其参数进行了简单的处理后，直接调用了 HeapAlloc 函数。

5.6.2 功能性区别

HeapAlloc、GlobalAlloc、LoaclAlloc 在功能上没有大的区别，都是在堆中分配内存，分配的内存不需要进行页对齐，也不用关心分页机制、页面状态、页面属性等内容。HeapAlloc 是程序在需要分配内存时最直接的方式。这些内容由堆管理器负责。

堆内存管理依赖于虚拟内存管理。在创建堆时，HeapCreate API 函数会向系统请求虚拟内存分页，之后在这个堆上的内存分配实际上是在从虚拟内存管理中获取的内存分页上再分配大小任意的内存块，如果在建立堆时指定了固定了堆大小，那么在堆上分配内存块时，其范围不能超过设置的堆大小，堆大小一定是内存页大小的整数倍。如果在建立堆时不固定堆的大小，堆管理函数会根据分配的请求数量，动态地向虚拟内容管理函数请求内存分页。

VirtualAlloc 的功能是对进程虚拟地址空间中内存分页的状态进行管理，属于虚拟内存管理（以内存页面为单位）的范围。当然内存的分配也是通过改变虚拟内存页面属性来实现的，所以 VirtualAlloc 间接达到了分配内存的目的。堆管理器也是依赖于虚拟内存管理的，在收到 HeapAlloc 的内存分配的请求时，堆管理器会根据情况决定是否使用虚拟内存管理机制分配新的页面，以及如何在页面上布局分配的内存块，从这个层次上讲 VirtualAlloc 更为底层。

实际上只要虚拟内存管理函数将页面属性设置为“已提交”后，就可以在之上进行读写等操作。

堆管理函数为应用程序提供了一种更灵活、更简单的方式来管理这些“已提交”的内存。其好处一是分配更简单，二是可以分配任意大小的内存，三是不用直接与复制的内存分页机制打交道。

标准 C 函数的内存分配函数则直接使用了 HeapAlloc。

5.6.3 效率的区别

HeapAlloc、GlobalAlloc、LoaclAlloc 在 32 位系统上功能上是一致的，但是由于 Globa]_Alloc、LoaclAlloc 需要兼容 16 位系统，所需进行的判断和处理会更多，因此 GlobalAlloc、LoaclAlloc 较 HeapAlloc 效率低。当然标准 C 函数 malloc 也是调用 HeapAlloc 实现的，所以其效率也必然比 HeapAlloc 低。

由于 HeapAlloc 与 VirtualAlloc 功能定位上的不同，所以在效率上没有可比性。从原理上讲，HeapAlloc 在分配内存时，如果堆管理器中有足够的已提交的页面可以使用，那么它不需要将内存分页从其他状态改变为已提交状态，只需要在堆管理器中进行相关管理；如果在分配时没有足够的已提交页面供使用，那么还需要将虚拟内存分页从其他状态改变为已提交状态，在有足够的可使用的已提交页面的情况下，再由堆管理器进行相关管理。

第 6 章　进程、线程和模块

运行一个程序就会产生一个进程。进程是应用程序的实例。

进程包括一个虚拟地址空间及代码、数据、对象等程序运行所需环境和资源的集合。具体而言，在内存空间中包括若干可执行的代码、数据、资源、一系列对系统对象操作的句柄，安全上下文、进程标识符(PID)，环境变量等程序执行的环境。同时，进程还包括一个或多个执行线程。

线程是程序的执行流程。在操作系统层面，线程是需要操作系统为其分配执行时间片的基本单元。线程附属于进程，一个线程可以执行进程中任意部分的代码。一个系统中同一时间只能有少量线程执行（决定于 CPU 个数和核数），操作系统会决定当前执行哪一个线程，并进行调度。每一个线程都包括一个上下文（主要是 CPU 寄存器值）。在进行线程调度时，系统会保存线程上下文。

一个程序由多个模块组成。程序没有运行时，模块以可执行文件的形式存在（exe、dll 等）程序运行后，所需要的模块都加载到进程的虚拟地址空间中，进行初始化的设置后，各模块就可以协同工作。每一个模块都可能包括代码、数据、资源等。

6.1　基本概念

先讲解与进程、线程和模块相关的概念，包括应用程序与进程的区别，两种基本的应用程序形式，动态连接库与应用程序的主模块，线程、纤程等。

6.1.1　应用程序与进程

1．可执行文件

程序的载体是可执行文件。可执行文件中包括运行所需的程序、数据及一些配置信息等。Windows 平台的可执行文件包括 exe、dll、sys 等扩展名的文件。

exe 是指可执行(executable)。有时可执行文件特指 exe 文件。exe 文件是程序的入口，创建进程必须使用 exe 文件。dll 是动态链接库，一些基础、共用的代码和数据存在于 dll 文件中。sys 是内核驱动程序。

Windows 平台的可执行文件基本都是 PE 格式的。

2．应用程序

应用程序是由一系列具有特定功能、能完成执行操作的可执行文件、数据的集合，一个应用程序可以包括若干个可执行程序、动态链接库、数据文件。应用程序一般是以功能划分，不以程序的存在形式划分。一些大型的应用程序可能包含若干个程序(exe 文件)，每个程序运行都会产生一个进程。

绝大多数应用程序在运行时需要与用户进行交互。Windows 平台的应用程序根据与用户交互的方式区分为控制台应用程序与图形用户界面(GUI)程序。

3．进程

进程提供了程序运行所需要的资源、虚拟内存地址空间、各种数据、可执行的代码、需要使用到的各种内核对象、系统调用接口、优先级与权限配置、工作集、环境变量等，还包括至少一个执行过程（线程）。

在创建进程时，系统将应用程序的可执行文件加载到内存中，设置相关环境后，开始启

动执行。要运行一个程序（exe 文件），系统会创建进程。系统先在自己的内存管理单元中构造出一个虚拟地址空间，供进程使用，然后将 exe 文件加载到进程的虚拟地址空间内存中指定的位置，配置程序数据和调用接口等，然后再创建主线程。进程是正在运行的程序，即程序的运行实例。

每个进程都有一个标识符(PID)和一个句柄，系统和其他进程可以通过 PID 和进程的句柄对进程进行管理和操作。

6.1.2 控制台应用程序与图形用户界面应用程序

控制台应用程序也称为字符模式应用程序。控制台应用程序使用控制台与用户进行交互，典型的控制台程序有 Windows 自带的命令行工具 cmd.exe 等。

图形用户界面是 Windows 系统中最常见的应用程序类型，图形用户界面程序具有窗口，用户通过操作窗口与控件、程序交互。

这两种应用程序属性于不同的子系统，具有不同的执行环境、入口函数、启动选项和用户交换模式。在编译构造程序时需要通过/SUBSYSTE 选择指定应用程序的类型是控制台程序还是图形用户界面程序。

6.1.3 动态链接库、模块

1．动态链接库文件

动态链接库文件中包含若干公用的代码、数据等，供其他模块使用。动态链接库是将应用程序模块化的重要方法，动态链接库中的函数和数据可以同时供其他多个可执行文件使用，不同的可执行文件可以调用同一个动态链接库中的函数。DLL 中函数经过“导出”(export)后可以被 exe 文件中的程序调用，也可以被其他 DLL 中的程序调用。

2．动态链接的原理与优点

DLL 是程序模块化的重要方法，也为程序的复用提供了可能，同一个动态链接库可以被不同的进程加载。DLL 中通常包括一些高复用性的代码，Windows API 就存在于动态链接库中。采用动态链接库后，同样的代码只需要有一份就可以，每个需要使用到这些功能的程序都使用同一份代码。将不同程序都可能使用到的一系列函数和数据放置在 DLL 中，可以大大减小应用程序的体积（多个程序共用一个模块），使程序更宜于升级与维护（如果程序需要更新，则只需要修改一个模块，不需要修改所有的程序）。

比如 API 函数 CreateFile 的可执行代码位于 kerne132.dll 中，所有要进行文件操作的程序都会使用的 CreateFile 函数，这些程序在进程创建时将 kerne132.dll 加载到进程中。因此所有用到了文件操作的进程的内存空间中都有 CreateFile 的代码，但是实际系统通过内存分页管理将多个虚拟内存分页映射到同一个物理分页上。因此在物理内存中，系统 DLL 的代码只有一份。

3．动态链接过程

在进程创建时，系统会从 exe 文件的导入表中知道 exe 文件会调用哪些动态链接库的哪些函数，在进行初始化时，将动态链接库加载到内存的虚拟地址空间中，并调整好函数调用的指针的地址。这样程序就可以调用 DLL 中的函数。由于程序是在进程创建或程序运行时进行链接的，因此称为“动态链接”，与“静态链接”相对。静态链接是在程序构建时由链接器进行链接，将代码和数据组织在同一个可执行文件中的链接方式。

4．模块

每个可执行程序都可以称作一个模块，DLL 文件是模块。exe 文件也是模块，称为主模块。

有关动态链接库在系统架构中的作用，动态链接库在应用程序结构中作用的详细讲解见第 18 章。

有关动态链接库的编写和编译链接方法将在 6.5 节介绍。

6.1.4 线程、纤程与作业

Windows 是一个多任务的操作系统，可以同时运行多个进程，一个进程也可以同时处理多个任务，例如在界面与用户交互的同时再进行网络通信。

进程多任务的优势是明显的，比如程序在后台处理数据时仍然可以响应用户在界面上的输入。同时，在多 CPU 或多核 CPU 的主机上，多任务的程序在性能上也具有优势。 ‘

进程的这种同时处理多个任务的能力是通过线程机制实现的。

1．线程(Thread)

线程是附属在进程上的执行实体，是代码的执行流程。

一个进程中可以有多个线程在并发执行。在系统中，同时存在的线程肯定比 CPU 数量多，一个 CPU 在同一时间只能执行一条机器指令，系统会将各个线程进行调度(schedule)，根据情况选择当前执行的线程，当一个线程执行一段时间后会再选择其他的线程进行执行。由于切换的时间非常短、频率很高，所以用户感觉是多个程序同时在执行。

context 所有线 CPU 共享其虚拟地址空间、数据、系统资源等。每个线程都有自己的栈、上下文（comeXt，包括 CPU 寄存器值等，线程调度时会进行线程上下文切换）、异常处理程序、调度优先级、标识符等。线程的切换依赖于中断。

每个进程必须有至少一个线程，在进程创建时即开始执行的线程是进程的主线程，主线程可以再创建其他线程。

2．线程(Fiber)

纤程也是程序执行单元，与线程类似。所不同的是纤程必须要应用程序自行调度，而线程是由系统进行调度的。纤程运行于线程的上下文中，一个线程可以调度很多纤程。

3．作业（Job，也称为工作项）

作业是一种对象，可以将一组进程作为一个单元进行管理。作业包括命名对象(namable)安全对象(securable)和共享对象(sharable)，可以控制进程组的属性。

6.1.5 权限与优先级

进程、线程权限是进程、线程紧密相关的内容。Windows 有完整的权限管理机制，将在 17 章专门讨论权限相关内容。

6.2 进程管理

在本节将介绍 Windows 系统进程管理的相关内容，如下所示。

◇如何创建和结束进程；

◇如何获取子进程的相关信息；

◇如何在子进程中获取启动参数；

◇如何创建 GUI 应用程序、控制台应用程序；

◇如何在进程中获取和设置环境变量等。

6.2.1 创建进程、获取进程相关信息、获取启动参数

1．父进程与子进程

如果一个进程创建了另外一个进程，那么被创建的进程称为子进程，创建者称为子进程的父进程。

2．启动参数

进程被创建后，可以使用 GetCommandLiIle、GetStartupInfo 函数获取启动参数、启动信息。

执行程序时，在命令行中输入的参数是程序的启动参数。以 main 函数作为入口函数的程序启动参数即是 maln 函数的参数。

3．进程句柄和进程标识

如果需要知道自己的句柄，可以使用 GetCurrentProcess 和 GetCurrentProcessId 函数获取自己的句柄和 PID。可以使用 GetPriorityClass、GetProcessTimes、GetProcessVersion、GetProcessWorkingSetSize 函数获取进程的相关信息。

如果进程知道其他进程的标识符(PID)，还可以通过 OpenProcess 函数获取进程的句柄，也可以通过 GetProcessId 通过句柄获取 PID，在具有权限的情况下，也可以通过相关函数获取其他进程的信息。

4．退出和终止进程

进程的主线程执行完成，则进程自动退出。如果要立即退出运行可以使用 ExitProcess 函数。

如果需要终止其他进程，可以使用 TerminateProcess。

5．关键 API

创建进程最常使用到的 API 函数 CreateProcess，另外还有 CreateProcessAsUser、CreateProcessWithLogonW，CreateProcessWithTokenW 等函数，这几个函数除了可以创建线程，还可以指定特殊的安全属性。WinExec 函数的功能是执行程序，也可以创建进程，但是只能设定几个简单的参数，　功能比 CreateProcess 简单。

(1) CreateProcess。

CreateProcess 函数的功能是创建进程，函数原型如下：

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

◇参数

lpApplicationName：输入参数，指向启动进程的 exe 文件，可以为 NULL，如果为 NULL，则 lpCommandLine 参数不能为 NULL。

lpCommandLine：输入参数，启动进程的命令行，第一个空格标识符前的字符为启动进行的 exe，此后紧接启动参数。

lpProcessAttributes：输入参数，指向 SECURITY_ATTRIBUTES 结构变量，为进程的安全属性，可设置为 NULL，使用默认安全属性。

lpThreadAttributes：输入参数，进程句柄是否可被子进程继承，设置为 NULL，则不会继承。

bInheritHandles:输入参数，与子进程的句柄继承关系。

dwCreationFlags：输入参数，进程的创建标志和优先级控制。比如 CREATE NEW_CONSOLE 会使新建的控制台程序具有一个新控制台；DEBUG_ PROCESS 标志会使父进程具有调试权限，其子进程会产生调试事件。

lpEnvironment：输入参数，指向新进程的环境变量块，如果设置为 NULL，则使用父进程的环境变量。

lpCurrentDirectory：输入参数，指定创建后新进程的当前目录，如果为 NULL，则为父进行的当前目录

lpStartupInfo：输入参数，指向 STARTUPNFO 数据结构变量，设定启动信息，可以为 NULL。

lpProcessInformation：输出参数，指向 PPROCESS_INFORMATION 结构的变量，返回被创建的进程的信息。

◇说明

lpApplicationName 或 lpCommandLine 参数指明的 exe 文件路径以是绝对路径，也可以是相对路径如果不包括路径而只是文件名，系统将按以下顺序找文件：父进程的当前路径、系统路径、Wows 路径、PATH 环境变量中包含的路径。文件名可以不包含.exe 扩展名。

命令行参数以空格为分隔符，如果参数内存在空格（比如文件路径中有空格）则将参数置于引号中。

(2)其他相关 API 函数使用方法都较简单，参见实例 6-1。

6．实例 6-1 创建进程

本实例演示如何调用 CreateProcess 创建进程，如何使用返回的进程信息。所创建的子进程 Child.exe 由实例 6-2 的代码编译生成。

```
/* ************************************
* process.c  创建进程、获取进程相关信息、获取启动参数
**************************************/
/* 头文件  */
#include <windows.h>
#include <stdio.h>

/************************************
* int main(void)
* 功能：演示进程创建
**************************************/
int main()
{
     CreateChildProcess("Child.exe abc 123");
}
/************************************
* DWORD CreateChildProcess(LPSTR szChildProcessCmd)
* 功能：演示创建子进程
* 参数：LPSTR szChildProcessCmd   启动子进程的命令行
* 返回值：成功返回 0，失败返回 1
**************************************/
DWORD CreateChildProcess(LPSTR szChildProcessCmd)
{
   STARTUPINFO si;
   PROCESS_INFORMATION pi;
   // 将启动信息结构清零
   ZeroMemory( &si, sizeof(si) );
   // 设置结构大小，cb 属性应为结构的大小
   si.cb = sizeof(si);
   // 将进程信息结构清零
   ZeroMemory( &pi, sizeof(pi) );
   // 创建子进程，并判断是否成功
   if( !CreateProcess( NULL,     // 使用命令行
         szChildProcessCmd,      // 命令行
         NULL,                   // 不继承进程句柄
         NULL,                   // 不继承线程句柄
         FALSE,                  // 不继承句柄
         0,                      // 没有创建标志
         NULL,                   // 使用父进程环境变量
         NULL,                   // 使用父进程目录作为当前目录
         &si,                    // STARTUPINFO 结构
         &pi )                   // PROCESS_INFORMATION 保存相关信息
         )
    {
         // 创建失败
         printf( "CreateProcess failed (%d).\n", GetLastError() );
         return 1;
    }
    // 在创建成功后父进程也可直接退出，这里等待子进程执行结束

    // 等待子进程结束
    // 使用到了通过 PROCESS_INFORMATION 返回的相关信息，hProcess 为子进程句柄
    // 父进程也可以不等待子进程运行完成而直接退出
    WaitForSingleObject( pi.hProcess, INFINITE );
    // 关闭进程句柄和线程句柄
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
    return 0;
}
```

在本实例中，为 CreateProcess 设置了尽可能简单的参数，使用 pi 变量获取返回的子进程信息，并使用了进程句柄等信息。

由于在命令行中直接使用文件名，在运行时，需将 Child.exe 与本实例程序放置在同一目录下。

7．实例 6-2 子进程

本实例是实例 6-1 中所创建的子进程 Clild.exe 的代码，Child.exe 是一个用户界面应用程序，在程序中使用了不同的方法来获取程序启动后的信息，包括句柄，命令行参数，PID、启动信息等。

```
/* ************************************
* child.c 创建进程、获取进程相关信息、获取启动参数
**************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
/* 宏定义 */
#define MyAlloc(size) HeapAlloc(GetProcessHeap(),0,size)
#define MyFree(lpMem) HeapFree(GetProcessHeap(),0,lpMem)
/* 结构体定义 */
typedef struct _PROCESS_INFO
{
    DWORD dwPid;
    HANDLE hProcess;
    DWORD dwPrioClass;
    DWORD dwHandleCount;
    DWORD dwAffinityMask;
    SIZE_T dwWorkingSetSizeMax;
    SIZE_T dwWorkingSetSizeMin;
    LPWSTR szwCommandLine;
    STARTUPINFO sti;
}PROCESS_INFO, *LPPROCESS_INFO;
/* 全局变量 */
HANDLE hMySelf;
/* 函数声明 */
DWORD GetProcessInfo(LPPROCESS_INFO lppi);

/**************************************
* WinMain
* 功能：演示获取进程信息，在进程中获取命令行参数等
**************************************/
int WinMain(
     HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow
)
{
    PROCESS_INFO pi;
    INT argc;
```

```
    WCHAR **argv;
    DWORD i;
    DWORD dwBufferSize = lstrlen(lpCmdLine)+MAX_PATH+1024;
    LPSTR szShowBuffer = (LPSTR)MyAlloc(dwBufferSize);

    hMySelf = hInstance;
    // 显示直接从 WinMain 函数参数得到的信息
    wsprintf(szShowBuffer,
         "启动参数\n 实例句柄: %.8X, 命令行参数: %s, 显示选项: %.8X",
         hInstance,lpCmdLine,nCmdShow);
     MessageBox(NULL,szShowBuffer,"WinMain 函数参数",MB_OK);
     // 使用自定义的函数获取相关信息
     GetProcessInfo(&pi);
     // 将命令行参数分离
     argv = CommandLineToArgvW(pi.szwCommandLine,&argc);
     // 字符处理, 并显示
     *szShowBuffer = NULL;
     for(i=0; i<argc; i++)
     {
          DWORD dwBufferSize = lstrlenW(*argv)+1;
          LPSTR szMBArgv = MyAlloc(dwBufferSize);
          WideCharToMultiByte(CP_ACP,NULL,*argv,-1,szMBArgv,dwBufferSize,NULL,NULL);
          argv++;
          lstrcat(szShowBuffer,"\n");
          lstrcat(szShowBuffer,szMBArgv);
          MyFree(szMBArgv);
     }
    MessageBox(NULL,szShowBuffer,"参数",MB_OK);
    MyFree(szShowBuffer);
    // 打印其他信息 TODO
    return 0;
}
/*************************************
* DWORD GetProcessInfo(LPPROCESS_INFO lppi)
* 功能: 获取进程相关信息, 保存在 PROCESS_INFO 结构中
* 参数: LPPROCESS_INFO lppi  用于保存相关信息
*************************************/
DWORD GetProcessInfo(LPPROCESS_INFO lppi)
{
   // PID
   lppi->dwPid = GetCurrentProcessId();
   // 句柄
   lppi->hProcess = GetCurrentProcess();
   // 优先级
   lppi->dwPrioClass = GetPriorityClass(hMySelf);
   // 句柄记数
   //windows xp sp1 sdk 新增的 API GetProcessHandleCount
   //lppi->dwHandleCount
   //     = GetProcessHandleCount(lppi->hProcess,&lppi->dwHandleCount);

   // AffinityMask
   GetProcessAffinityMask(hMySelf,
        &lppi->dwAffinityMask,
        NULL);
   // WorkingSetSize
   GetProcessWorkingSetSize(hMySelf,
        &lppi->dwWorkingSetSizeMin,
        &lppi->dwWorkingSetSizeMax);
   lppi->szwCommandLine =  GetCommandLineW();
   // 启动信息
   GetStartupInfo(&lppi->sti);
   return 0;
}
```

8. 实例 6-1、6-2 运行结果

在这里只输出了所获取的部分信息。ClilcLexe 是 GUI 应用程序，弹出了两次对话框，演示使用不同方法获取命令行的参数等信息。其他信息未显示，读者可自行查看。运行结果如图 6-1 和 6-2 所示。

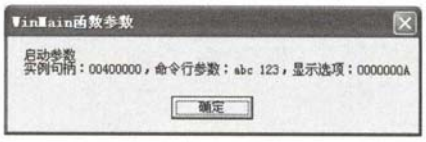

图 6-1　程序的启动参数

图 6-2　创建进程时，传递给子进程的参数

6.2.2　编写控制台程序和图形用户界面应用程序

1．控制台程序

Windows 应用程序分为控制台应用程序和图形用户界面应用程序。控制台程序以字符界面作为与用户交互的界面。

本书到现在为止，绝大部分的实例都是控制台程序，在之前的实例中，大多使用标准 C 函数进行用户交互，如 printf. getchar 等，标准 C 库函数所提供的都是字符界面功能。其实 Windows 有丰富的控制台界面操作 API，具有很多标准 C 函数所不具有的功能。

2．图形用户界面

图形用户界面程序可以使用窗口，大多数 Windows 平台的应用程序都是图形用户界面的。

本书前 8 章的实例有少数几个是图形用户界面应用程序，第 9 章以后的示例程序多为 GUI 程序。

3．代码编写时的区别

Windows 应用程序的入口函数可以是 maln 或 wmaln 函数，也可以是 WinMain 函数。控制台程序的入口函数应该为 main 或 wmaln，GUI 程序的入口函数应该为 WinMain，在链接生成可执行文件时，所需的链接选项也不同，/SUBSYSTEM：CONSOLE 用于生成控制台程序，/SUBSYSTEM:WINDOWS 用于生成 GUI 程序。如果链接时未设置子系统选项，则链接器会根据程序员定义的入口函数来自动选择。

6.2.3　获取和设置环境变量

量是进程中的一组变量信息，环境变量分为系统环境变量、用户环境变量和进程环境变量。系统有全局的环境变量，在进程创建时，进程继承了系统的全局环境变量、当前登录用户的用户环境变量和父进程的环境变量。进程也可以有自己的环境变量，设置和获取所在进程的环境变量使用 API 函数 GetEnvironmentStrings、GetEnvironmentVariable 和 SetEnvironmentVariable 等。

1．关键 API

(1) GetEnvironmentStringS。

GetEnvironmentStrings 函数可以获取所有环境变量字符串，GetEnvironmentStrings 函数无参数，返回 PVOID 类型数据，指向保存环境变量的缓冲区，使用完成需由 FreeEnvironmentStrings
函数释放。

(2) GetEnvironmentVariable。

获取指定的环境变量，函数原型如下：

```
DWORD GetEnvironmentVariable(
 LPCTSTR lpName,
 LPTSTR lpBuffer,
 DWORD nSize
);
```

lpName 是环境变量名，lpBuffer 指向保存环境变量值的缓冲区，nSize 是缓冲区大小。返回真实的环境变量值大小。

(3) SetEnvironmentVariablea

设置指定的环境变量，函数原型如下：

```
BOOL SetEnvironmentVariable(
  LPCTSTR lpName,
  LPCTSTR lpValue
);
```

2．实例 6-3 设置和获取环境变量

函数 EnumEnvironmentVariables 和 ChangeEnviromentVariables 分别是显示和设置进程的环境变量。

```
/* ************************************
* env.c 获取和设置环境变量
**************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
#include "env.h"
/* 预定义 */
#define BUFSIZE 4096
/************************************
* DWORD WINAPI EnumEnvironmentVariables()
* 功能：显示进程的所有环境变量
**************************************/
DWORD WINAPI EnumEnvironmentVariables()
{
   // 获取环境变量
   PVOID pEv = GetEnvironmentStrings();
   LPSTR szEnvs;
   // 显示
   for (szEnvs = (LPSTR) pEv; *szEnvs;)
   {
         printf("%s\n",szEnvs);
         while (*szEnvs++);
   }
   // 释放
   FreeEnvironmentStrings(pEv);
   return 0;
}
/************************************
* DWORD WINAPI ChangeEnviromentVariables(LPSTR szName,
LPSTR szNewValue,
DWORD dwFlag)
* 功能：改变环境变量
* 参数：LPSTR szName，需要改变的环境
*       LPSTR szNewValue，新的变量值
```

```
*      DWORD dwFlag, 附加、重置还是清零, 相关常量在 env.h 中定义
************************************/
DWORD WINAPI ChangeEnviromentVariables(LPSTR szName,
                                       LPSTR szNewValue,
                                       DWORD dwFlag)
{
        DWORD dwErr;
        PVOID szVal;
        DWORD dwReturn;
        DWORD dwNewValSize;
        // 如果标志为附加, 则先获取, 然后将 szNewValue 附加到末尾
        if(dwFlag == VARIABLES_APPEND)
        {
             dwNewValSize = lstrlen(szNewValue)+1; // 新变量值的大小
             // 分配内存
             szVal = HeapAlloc(GetProcessHeap(),0,BUFSIZE+dwNewValSize);
             // 获取值
             dwReturn = GetEnvironmentVariable(szName,szVal,BUFSIZE);
             if(dwReturn == 0)   // 出错
             {
                  dwErr = GetLastError();
                  if( ERROR_ENVVAR_NOT_FOUND == dwErr )
                  {
                       printf("Environment variable %s does not exist.\n", szName);
                  }
                  else
                  {
                       printf("error: %d",dwErr);
                  }
                  return FALSE;
              }
              else if(BUFSIZE < dwReturn) // 缓冲区太小
              {
                     szVal=(LPTSTR)HeapReAlloc(GetProcessHeap(),0,szVal,dwReturn+dwNewValSize);
                     if(NULL == szVal)
                     {
                      printf("Memory error\n");
                      return FALSE;
                     }
              dwReturn = GetEnvironmentVariable(szName, szVal, dwReturn);
              if(!dwReturn)
              {
                        printf("GetEnvironmentVariable failed (%d)\n", GetLastError());
                        return FALSE;
               }
             }
             lstrcat(szVal,";");      // 分隔符
             lstrcat(szVal,szNewValue);   // 附加
             //设置
             if(!SetEnvironmentVariable(szName,szVal))
             {
                 printf("Set Value Error %d",GetLastError());
             }
             // 释放内存
            HeapFree(GetProcessHeap(),0,szVal);
            return TRUE;
    }
    // 如果是重置, 则直接设置
    else if(dwFlag == VARIABLES_RESET)
    {
          if(!SetEnvironmentVariable(szName,szNewValue))
```

```
            {
                printf("Set value error %d",GetLastError());
            }
        }
        // 清零，忽略 szNewValue
        else if(dwFlag == VARIABLES_NULL)
        {
            if(!SetEnvironmentVariable(szName,NULL))
            {
                printf("Set value error %d",GetLastError());
            }
        }
        return TRUE;
        }
    //main 函数演示了以上两个函数的功能
    //*************************************
    * int main(void)
    * 功能：演示虚拟内存的使用
    *
    * 参数：未使用
    *************************************/
    int main(void)
    {
        EnumEnvironmentVariables();
        ChangeEnviromentVariables("PATH","C:\\",VARIABLES_APPEND);
        EnumEnvironmentVariables();
        ChangeEnviromentVariables("PATH","C:\\",VARIABLES_NULL);
        EnumEnvironmentVariables();
        ChangeEnviromentVariables("XX","C:\\",VARIABLES_RESET);
        EnumEnvironmentVariables();
    }
```

3．运行结果

由于多次列举了进行中的所有环境变量，输入结果较大，这里只对变量 PATH 和 XX 进行演示。

第一次输出：

```
    Path=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\Program Files\Microsoft
Platform SDK for Windows Server 2003 R2\Bin\
```

第二次输出：

```
    Path=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\Program Files\Microsoft
Platform SDK for Windows Server 2003 R2\Bin\.C:\
```

第三次输出，Path 环境变量不存在。

第四次输出：

```
    XX=C:\
```

4．实例 6-4 验证从父进程继承环境变量

在 CMD 中运行：

```
    SET ABC=ABC
```

再运行实例 6-3 的程序 env.exe，输出的结果中存在环境变量 ABC，而如果没有运行之前的 SET 命令，直接运行 env.exe，则输出结果中不存在环境变量 ABC，说明 env.exe 从其父进程 cmd.exe 中继承了环境变量。

读者也可以使用 API 编写程序验证。

6.3 线程、纤程

本节将使用实例演示如何创建线程、调度线程、如何使用线程、如何使用作业，还将演示如何为其他进程创建线程，并在其他线程中执行代码，并将演示线程池的使用。

6.3.1 创建线程、退出线程、获取线程信息

1．创建线程和线程函数

使用 CreateThreadAPI 函数可以创建线程，每一个线程都有一个线程函数(ThreadProc)，线程函数是线程执行的起点。调用 CreateThread 创建线程时，需要指定线程函数、线程函数的参数等。

(1)CreateThread 函数定义如下：

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

其中最重要的参数是 lpStartAddress 和 lpParameter。lpStartAddress 是线程的执行起点，也是线程函数的的指针。lpParameter 是启动线程参数。lpThreadAttributes 参数是线程的安全属性，dwStackSize 参数是线程堆栈大小，dwCreationFlags 参数是创建标志，lpThrreadId 参数返回 TID。

CreateThread 函数返回新建线程的句柄。

(2) 线程函数需要符合一定的接口参数要求。ThreadProc 函数的原犁如下：

```
DWORD WINAPI ThreadProc(
  LPVOID lpParameter
);
```

线程函数的函数名没有特殊要求，可以是 ThreadProc，也可以是其他，但其参数、调用方式和返回值必须满足上面的接口形式。

线程函数的参数是 LPVOID 类型的，可以指向任意类型的数据。如果线程中需要使用的参数较多，可以将参数封装在结构体中，并将结构体指针作为线程参数。在创建线程时由 CreateThread 的 lpParameter 参数指定。各线程间也可以通过全局变量和堆内存块来共享数据。但是需注意共享内存访问的同步问题。

2．线程句柄和 TID

进程中的每一个线程都有一个句柄和一标识符(TID)TID 是一个 DWORD 类型的数据，每一个线程的 TID 都不一样，因此 TID 可以用于唯一标识一个线程。如果知道一个线程的 TID，那么可以通过 OpenThread 获取线程的句柄，如果知道一个线程的句柄，可以使用 GetThreadId 获取线程的 TDo GetCurrentThread 和 GetCurrentThreadId 函数可以获取本线程的句柄和 TID。

3．权限与优先级

GetThreadPriority 、 GetThreadPriorityBoost 、 SetThreadPriority 、 SetThreadPriorityBoost 是与线程的权限有关的几个函数，可以获取和设置线程的权限。

4．退出线程

线程可以自行退出，也可以被其他线程终止。如果线程函数执行完成，则线程自动退出；线程调用 ExitThread 函数退出线程。要终止指定线程的执行，使用 TerminateThread 函数。

5．实例 6-5 创建线程、退出线程、获取线程信息

```
/* ************************************
* CreateThread.c创建线程、退出线程、获取线程信息
************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
/* 常量定义 */
#define MAX_THREADS 5
/* 结构类型 */
typedef struct _THREAD_PARAM {
    DWORD i;
    DWORD dwRandom;
    DWORD dwData;
} THREAD_PARAM, *LPTHREAD_PARAM;

/************************************
* DWORD WINAPI ThreadProc( LPVOID lpParam )
* 功能: 线程函数
*       将参数打印出
* 参数: LPVOID lpParam  THREAD_PARAM 结构类型
************************************/
DWORD WINAPI ThreadProc( LPVOID lpParam )
{
    LPTHREAD_PARAM pData;
    // 参数数据类型
    pData = (LPTHREAD_PARAM)lpParam;
    // 显示参数
    printf("TID = %u,\t Parameters = %u, %u,  %u\n",
        GetCurrentThreadId(),
        pData->i, pData->dwRandom, pData->dwData);
    // 释放保存参数的内存(在主线程中分配的)
    HeapFree(GetProcessHeap(), 0, pData);
    return 0;
}
/************************************
* void main()
* 功能: 主线程函数, 创建了多个线程
************************************/
void main()
{
    LPTHREAD_PARAM pData;
    DWORD dwThreadId[MAX_THREADS];
    HANDLE hThread[MAX_THREADS];
    int i;
    // 创建MAX_THREADS 个线程
    for( i=0; i<MAX_THREADS; i++ )
    {
        // 为线程函数参数分配内存
        pData = (LPTHREAD_PARAM)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
            sizeof(THREAD_PARAM));
        if( pData == NULL )
        {
            printf("HeapAlloc error;\n");
            ExitProcess(2);
        }
```

```
            // 设置参数
            pData->i = i;
            pData->dwRandom = rand();
            pData->dwData = 100;
            // 创建线程
            hThread[i] = CreateThread(
                NULL,                   // 默认安全属性
                0,                      // 默认堆栈大小
                ThreadProc,             // 线程函数
                pData,                  // 参数
                0,                      // 默认创建标志
                &dwThreadId[i]);        // 返回 TID
            // 判断是否创建成功
            if (hThread[i] == NULL)
            {
                ExitProcess(i);
            }
        }
    }
```

本实例，由主线程创建了若干个线程，为每个线程设置不同的参数。在每个线程中，获取本线程的 TID，处理参数，并将 TID 和参数的内容打印显示。

6．运行结果

```
TID = 2280, Parameters = 0, 41,  100
TID = 3488, Parameters = 1, 18467,  100
TID = 3120, Parameters = 2, 6334,  100
TID = 1804, Parameters = 3, 26500,  100
```

实例代码中创建了 5 个线程，但是只打印出了 4 条结果，这是由于还有线程没有执行完，主线程就退出了。可以在 main 函数 for 循环后添加如下代码，等待所有线程执行完成，相关原理参见第 7 章线程同步。

```
    //等待所有线程执行结束
    WaitForMultipleObjects(MAX_THREADS, hThread, TRUE, INFINITE);
    // 关闭所有线程的句柄
    for(i=0; i<MAX_THREADS; i++)
    {
        CloseHandle(hThread[i]);
    }
```

6.3.2　挂起、恢复、切换、终止线程

1．线程调度函数

在创建完成线程后，使用 Sleep、SleepEx、SuspendThread、SwitchToThread、ResumeThread 进行线程调度，TerminateThread 和 ExitThread 函数可以终止和退出线程。GetExitCodeThread 可以获得线程的退出代码。

（1）Sleep、SleepEx。

Sleep 和 epEx 函数的功能是在指定的执行时间内中止(挂起)所在线程的执行。SleepEx 扩展了 Sleep 的功能，在最短超时时间间隔已到的情况下恢复线程执行外，在以下两种情况发生时也可以恢复执行：I/O 完成回调函数（参见 16.4 节）被调用、异步过程 i 用 APC)被安排进入到线程中。

◇　函数原型

```
VOID WINAPI Sleep(
  DWORD dwMilliseconds
);
DWORD WINAPI SleepEx(
  DWORD dwMilliseconds,
```

```
  BOOL bAlertable
);
```

◇参数

dwMilliseconds：输入参数，Sleep 与 SleepEx 函数的 dwMilliseconds 参数都是指定线程挂起最短时间间隔，以 ms 为单位。

bAlertable：输入参数，SleepEx 如果被设置为 FALSE，那么，在线程挂起期间不会响应。如果一个 I/O 完成回调函数被调用或有新的 APC 按排进入，它们使该函数返回，也不会被执行。

◇返回值

Sleep 函数无返回值，SleepEx 返回 0 表示时间间隔完成，返回 WAIT_IO_COMPLETION 表示 I/O 完成回调函数被调用。

（2）SuspendThread、ResumeThread。

SuspendThread 是挂起指定的线程，不同于 Sleep 只能挂起其所在的线程并在时间间隔超时后自动恢复，而 SuspendThread 挂起的线程需要使用 ResumeThread 恢复。

◇SuspendThread 函数的原型

```
DWORD WINAPI SuspendThread(
  HANDLE hThread
);
```

◇参数

hThread:输入参数，需要挂起的线程的句柄。

◇返回值

如果返回-1(0xFFFFFFFF)，表示失败；如果返回正数，表示线程已经被挂起过的次数。

ResumeThread 可以恢复被 SuspendThread 挂起的线程的执行。

◇ResumeThread 函数原型

```
DWORD WINAPI ResumeThread(
  HANDLE hThread
);
```

（3）SwitchToThread。

SwitchToThread 函数的作用是使线程主动让出程序的执行，主操作系统选择正在等待的线程执行，函数定义如下：

```
BOOL WINAPI SwitchToThread(void);
```

如果调用后系统选择别的线程执行，函数会返回非零值，如果当前没有线程等待执行，操作系统没有将线程进行切换，函数返回 0。

(4) ExitThread。

ExitThread 函数的作用是退出本线程，函数原型如下：

```
VOID WINAPI ExitThread(
  DWORD dwExitCode
);
```

ExitThread 可以指定退出代码。线程的创建者可以得到这个代码。

(5) TerminateThreada

TerminateThread 的作用是终止线程的执行。不同于 ExitThread，TerminateThread 可以终止其他线程的执行，使用线程句柄为参数，也可以指定退出代码，函数原型如下：

```
BOOL WINAPI TerminateThread(
  HANDLE hThread,
  DWORD dwExitCode
);
```

.一个线程要终止其他线程，必须要具有 THREAD_TERMINATE 权限。

2.实例 6-6 挂起、恢复、切换、终止线程

本实例调用挂起、恢复、切换、终止线程的操作来演示相关函数的使用方法，代码如下：

```
/* ************************************
* schedule.c 挂起、恢复、切换、终止线程
**************************************/
/* 头文件*/
#include <windows.h>
#include <stdio.h>
/**************************************
* DWORD WINAPI ThreadProc( LPVOID lpParam )
* 功能: 线程函数
*       间隔循环打印输出
* 参数: LPVOID lpParam 指向 DWORD
**************************************/
DWORD WINAPI ThreadProc( LPVOID lpParam )
{
     LPDWORD pData;
     DWORD i = 0;
     // 参数数据类型
     pData=(LPDWORD)lpParam;
     // 循环打印输出
     for(i;i<10;i++)
     {
          Sleep(100);        // 每 ms 打印一次
          printf("TID = %u,\t Parameters = %u\t i = %u\n",
              GetCurrentThreadId(), *pData, i);
     }
     ExitThread(i);
     return 0;
}

/**************************************
* void main()
* 功能:主线程函数，演示线程调度
**************************************/
void main()
{
     DWORD dwData;
     DWORD dwThreadId[2];
     HANDLE hThread[2];
     // 创建线程
     dwData = 1;
     hThread[0] = CreateThread(
          NULL,0,
          ThreadProc,
          &dwData,
          CREATE_SUSPENDED, // 挂起新建的进程
          &dwThreadId[0]);
     if (hThread[0] == NULL)
     {
          ExitProcess(0);
     }

     /// 创建线程
     dwData = 2;
     hThread[1] = CreateThread(
          NULL,0,
          ThreadProc,
          &dwData,
          0,                        // 默认标志
          &dwThreadId[1]);
     if (hThread[1] == NULL)
     {
```

```
            ExitProcess(1);
        }

        // 等待 200ms，恢复线程的执行
        Sleep(200);
        ResumeThread(hThread[0]);
        // 挂起线程的执行
        SuspendThread(hThread[1]);
        // 等待 300ms，终止线程，恢复线程
        Sleep(300);
        TerminateThread(hThread[0],0);
        ResumeThread(hThread[1]);

        //等待所有线程执行结束
        WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
        // 关闭所有线程的句柄
        CloseHandle(hThread[0]);
        CloseHandle(hThread[1]);
    }
```

在实例中，先创建了一个挂起的线程，然后创建了一个立即运行的线程。之后，主线程间隔一段时间对这两个线程进行调度。

运行结果如图 6-3 所示。

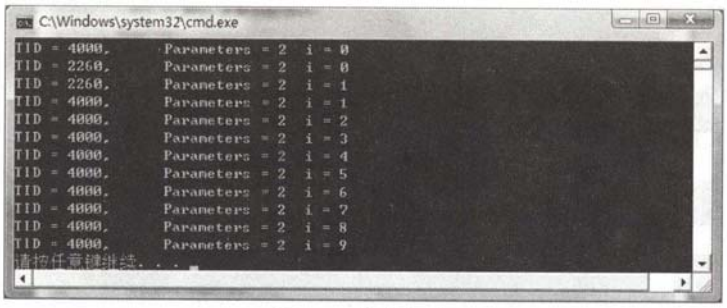

图 6-3　线程调度

6.3.3　创建远程线程、将代码注入其他进程中执行

之前所讨论的线程创建都是创建本进程的线程，如果进程具有特定权限，可以为其他进程创建线程，这种线程叫做远程线程。创建远程线程使用 CreateRemoteThread 函数。在创建线程前还需要使用 WriteProcessMemory 等函数将线程函数代码和参数写入目标进程的空间中去。

本节将演示如何获得可以为其他进程创建线程的权限，演示如何创建远程线程。

这种技术常用于对目标进程进行调试，一些较原始的恶意程序也使用此技术来隐藏进程。

1．关键 API

(1) CreateRemoteThread。

为指定进程创建线程，线程将会在其他进程的进程空间中执行，访问其他进程的资源，函数原型原型如下：

```
HANDLE WINAPI CreateRemoteThread(
  HANDLE hProcess,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
```

```
  LPDWORD lpThreadId
);
```

◇参数

hProcess:输入参数，需要为其创建线程的目标进程句柄。对目标进程需要有如下权限：PROCESS_CREATE_THREAD、 PROCESS_QUERYjNFORMATION、 PROCESS_VM_WRITE PROCESS_VM_READ 和 PROCESS_VM_OPERATION。

其他参数的使用方法参见 CreateThread 函数。

◇返回值

返回 HANDLE 值，为新创建进程的句柄。

(2) WriteProcessMemory

WriteProcessMemory 函数可以将数据写入到目标进程的虚拟地址空间中，函数原型如下：

```
BOOL WriteProcessMemory(
  HANDLE hProcess,
  LPVOID lpBaseAddress,
  LPCVOID lpBuffer,
  SIZE_T nSize,
  SIZE_T* lpNumberOfBytesWritten
);
```

◇参数

hProcess:输入参数，目标进程句柄，操作进程需要有目标进行的 PROCESS_VM_WRITE 和 PROCESS_VM_OPERATION 权限。

lpBaseAddress:输入参数，写入数据的地址，目标进程虚拟地址空间中的地址。

lpBuffer:输入参数，指向需要写入的数据。

nSize:输入参数，写入数据的大小，字节。

lpNumberOBytesWritten:输出参数，指向 SIZE_T 类型的变量，用于保存实际写入的数据的大小。

◇返回值

是否写入成功，如果失败可使用 GetLastError 函数获取错误信息。

本节实例中还会使用 VirtualAllocEx 等函数为其他进程分配虚拟内存空间，VirtualAllocEx 的使用方法已经在第 5 章介绍。

2．实例 6-7 创建远程线程、将代码注入到其他进程中执行

本实例将为 explorer.exe 创建一个线程，并加载一个动态链接库 msg.dll，msg.dll 被加载后弹出对话框显示所在进程的进程名和 PID。

创建远程线程需要保证线程函数的可执行代码位于目标进程的虚拟地址空间中，这里作简化处理，直接将肯定存在进程地址空间中的 API 函数 LoadLibraryA 作为线程函数调用。

```
* BOOL LoadRometeDll(DWORD dwProcessId, LPTSTR lpszLibName)
* 功能：通过创建远程线程给其他进程加载 DLL
* 参数：DWORD dwProcessId     目标进程 PID
*       LPTSTR lpszLibName Dll 的路径
* 返回：是否成功
**************************************/
BOOL LoadRometeDll(DWORD dwProcessId, LPTSTR lpszLibName)
{
    BOOL   bResult         = FALSE;
    HANDLE hProcess        = NULL;
    HANDLE hThread         = NULL;
    PSTR   pszLibFileRemote = NULL;
    DWORD  cch;
```

```
    PTHREAD_START_ROUTINE pfnThreadRtn;
    __try
    {
        // 获得想要注入代码的进程的句柄
        hProcess = OpenProcess(
            PROCESS_ALL_ACCESS,
            FALSE,
            dwProcessId
            );
        if (hProcess == NULL)
            __leave;
        // 计算 DLL 路径名需要的字节数
        cch = 1 + lstrlen(lpszLibName);
        // 在远程线程中为路径名分配空间
        pszLibFileRemote = (PSTR)VirtualAllocEx(
            hProcess,
            NULL,
            cch,
            MEM_COMMIT,
            PAGE_READWRITE
            );
        if (pszLibFileRemote == NULL)
            __leave;
        // 将 DLL 的路径名复制到远程进程的内存空间
        if (!WriteProcessMemory(
            hProcess,
             (PVOID)pszLibFileRemote,
            PVOID)lpszLibName,
            cch,
            NULL))
            __leave;
        // 获得 LoadLibraryA 在 Kernel32.dll 中的真正地址
        pfnThreadRtn = (PTHREAD_START_ROUTINE)GetProcAddress(
            GetModuleHandle(TEXT("Kernel32")), TEXT("LoadLibraryA"));
        if (pfnThreadRtn == NULL)
            __leave;
        // 创建远程线程，并通过远程线程调用用户的 DLL 文件
        hThread = CreateRemoteThread(
            hProcess,
            NULL,
            0,
            pfnThreadRtn,
             (PVOID)pszLibFileRemote,
            0,
            NULL
            );
        if (hThread == NULL)
            __leave;
        // 等待远程线程终止
        WaitForSingleObject(hThread, INFINITE);
        bResult = TRUE;
    }
    __finally
    {
        // 关闭句柄
        if (pszLibFileRemote != NULL)
            VirtualFreeEx(hProcess, (PVOID)pszLibFileRemote, 0, MEM_RELEASE);
        if (hThread  != NULL)
            CloseHandle(hThread);
        if (hProcess != NULL)
            CloseHandle(hProcess);
    }
    return bResult;
}
```

```
/*************************************
* WinMain
*************************************/
int WinMain(
            HINSTANCE hInstance,
            HINSTANCE hPrevInstance,
            LPSTR lpCmdLine,
            int nCmdShow
            )
{
    DWORD dwPID;
    // 提权，获取 SE_DEBUG_NAME 权限
    // 可以在其他进程的内存空间中写入、创建线程
    if(0!=EnablePrivilege (SE_DEBUG_NAME))
        return 0;
    // 获取目录进程的 PID
    if(!GetProcessIdByName("explorer.exe",&dwPID))
        return 0;
    // 通过创建远程线程加载 DLL
    // 将 msg.dll 放置在系统目录下
    if(!LoadRometeDll(dwPID,"msg.dll"))
        return 0;
    return 1;
}
```

WinMain 函数中所调用的 EnablePrivilege 函数的实现原理参见实例 17-3，GetProcessIdByName 函数的实现原理参见实例 6-10。

运行结果如图 6-4 所示(Vista)。

对话框显示出了 explorer.exe 的进程名和 PID。图 6-5 所示任务管理器说明进程名和进行 ID 显示正确。

图 6-4　远程线程弹出的对话框

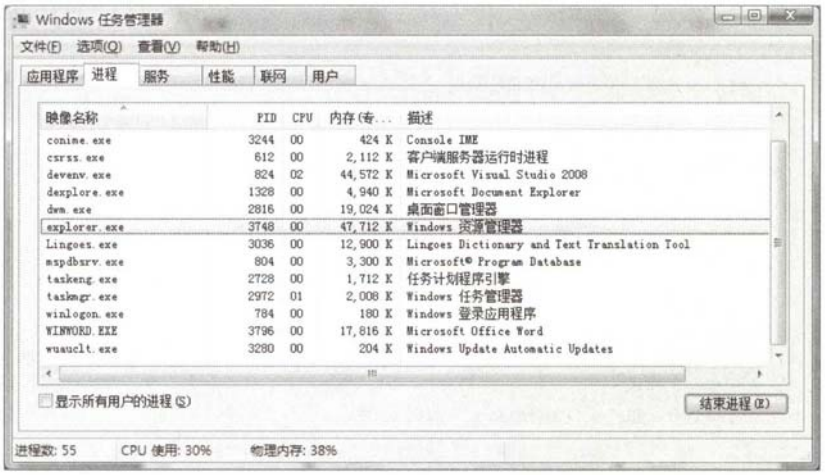

图 6-5　对话框模块的是 explorer.exe 的进程 ID

6.3.4　创建纤程、删除纤程、调度纤程

CreateFiber、CreateFiberEx 函数可以创建纤程，纤程创建时需要指定纤程函数(FiberProc)、DeleteFiber 可以删除纤程。

纤程与线程的不同是，线程是由系统调度的，由系统决定当前执行哪个线程。而纤程必

需自行调度，只要纤程不交出执行，纤程就将一直占用执行时间。调度纤程使用函数SwitchToFiber。
纤程通常用来完成一些不可分割的原子操作。

纤程函数是一个回调函数，是纤程执行的起始。每个纤程都有一个纤程函数，纤程创建后，纤程函数将开始执行。纤程函数代码在内存中的起始地址通常用来表示纤程。

关键 API

(1) CreateFiber。

CreateFiber 函数的作用是创建一个纤程，函数原型如下：

```
LPVOID WINAPI CreateFiber(
  SIZE_T dwStackSize,
  LPFIBER_START_ROUTINE lpStartAddress,
  LPVOID lpParameter
);
```

◇参数

dwStackSize:输入参数，栈的输入初始化大小，如果为 0，则使用默认大小。
lpStartAddress:输入参数，纤程的起始函数地址
lpParameter:输入参数，纤程函数的参数。

◇返回值

如果成功返回纤程的地址，返回 NULL 为失败，使用 GetLastError 函数获取函数地址。
(2) DeleteFiber。
DeleteFiber 函数可以删除已经存在的纤程。

◇函数原型

```
VOID WINAPI DeleteFiber(
  LPVOID lpFiber
);
```

◇参数

lpFiber:输入参数，需要删除的纤程的地址。

(3) SwitchToFiber。

SwitchToFiber 函数的作用是调度纤程，将执行过程由当前纤程切换至指定的纤程。即纤程将交出执行的权利，由其他纤程来执行。SwitchToFiber 函数只能在纤程中调用，不可以切换至自己。
纤程相关操作完成后，必须调用 SwitchToFiber 交出执行权利。

SwitchToFiber 函数是进行纤程调度的关键，实例 6-8 有详细的使用方法说明

◇函数原型

```
VOID WINAPI SwitchToFiber(
  LPVOID lpFiber
);
```

◇参数

lpFiber:输入参数，需要切换至的纤程地址。

6.3.5 纤程与线程的互相转换

ConvertThreadToFiber、ConvertThreadToFiberEx 函数可以将线程转换为纤程，ConvertFiberToThread 函数可以将纤程转换为线程。

注意 由于 CreateFiber 只是创建纤程而不是执行纤程，而只有纤程才可以调用 SwitchToFiber 函数，因此线程在创建纤程后需要将自己转换为纤程才可切换至其他纤程。

1．关键 API

(1) ConvertThreadToFiber。

ConvertThreadToFiber 将当前线程转换为纤程。

◇函数原型

```
LPVOID WINAPI ConvertThreadToFiber(
  LPVOID lpParameter
);
```

◇参数

lpParameter:输入参数，传递给纤程的参数，纤程可以使用 GetFiberData 宏获取此参数。

◇返回值

返回值为纤程的地址，如果失败则返回 NULL。

(2) ConvertFiberToThread。

ConvertFiberToThread 函数将当前纤程转换为线程，释放 ConvertThreadToFiber 所分配的相关

资源。

◇函数原型

```
BOOL WINAPI ConvertFiberToThread(void);
```

◇返回值

返回 BOOL 型值，表示成功或失败，使用 GetLastError 函数获取错误信息。

2．实例 6-8 使用纤程

本实例演示如何创建纤程，如何使用纤程函，如何调度纤程以及如何结束纤程，代码如下：

```
/* ************************************
* Fibers.c 创建、删除纤程、调度纤程 纤程与线程的互相转换
**************************************/
/* 预定义 */
#define _WIN32_WINNT 0x0501
/* 头文件 */
#include <windows.h>
#include <stdio.h>
/* 函数声明 */
VOID WINAPI ReadFiberFunc( LPVOID lpParameter );
VOID WINAPI WriteFiberFunc( LPVOID lpParameter );
/* 结构定义 */
// 用于向纤程传递参数
// 本实例是使用读、写文件来演示纤程的调度
// 用户可根据实际情况自行定义
typedef struct _FIBERDATASTRUCT
{
     DWORD dwParameter;              //向线程传递的参数
     DWORD dwFiberResultCode;        // GetLastError() 值
    HANDLE hFile;                    // 纤程所操作文件的句柄
     DWORD dwBytesProcessed;         // 已经处理了的字节
}FIBERDATASTRUCT, *LPFIBERDATASTRUCT;
/* 常量定义 */
#define RTN_OK 0                     // 返回值，成功
#define RTN_USAGE 1                  // 返回值，参数不正确
#define RTN_ERROR 2                  // 返回值，错误
#define BUFFER_SIZE 32768            // 缓冲区大小
#define FIBER_COUNT 3                // 主纤程、读纤程、写纤程，共 3 个
```

```
#define PRIMARY_FIBER 0                 // 主纤程
#define READ_FIBER 1                    // 读纤程
#define WRITE_FIBER 2                   // 写纤程

LPVOID g_lpFiber[FIBER_COUNT];          // 纤程地址的数组
LPBYTE g_lpBuffer;                      // 缓冲区
DWORD g_dwBytesRead;                    // 已读的字节

int main( int argc, char *argv[] )
{
    LPFIBERDATASTRUCT fs;
    // 用法说明
    if (argc != 3)
    {
        printf("Usage: %s <SourceFile> <DestinationFile>\n", argv[0]);
        return RTN_USAGE;
    }
    // 分配 FIBERDATASTRUCT 空间，FIBER_COUNT 个
    fs = (LPFIBERDATASTRUCT)HeapAlloc(
        GetProcessHeap(), 0,
        sizeof(FIBERDATASTRUCT) * FIBER_COUNT);
    if(fs == NULL)
    {
        printf("HeapAlloc error! (rc%=lu)\n", GetLastError());
        return RTN_ERROR;
    }
    //   分配读、写缓冲区
    g_lpBuffer = (LPBYTE)HeapAlloc(GetProcessHeap(), 0, BUFFER_SIZE);
    if (g_lpBuffer == NULL)
    {
        printf("HeapAlloc error! (rc=%lu)\n", GetLastError());
        return RTN_ERROR;
    }
    // 打开源文件，将句柄赋值给 fs 结构的 hFile 成员，使纤程可以使用该句柄
    fs[READ_FIBER].hFile = CreateFile(
        argv[1], GENERIC_READ,
        FILE_SHARE_READ, NULL, OPEN_EXISTING,
        FILE_FLAG_SEQUENTIAL_SCAN, NULL
        );
    if (fs[READ_FIBER].hFile == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile error! (rc=%lu)\n", GetLastError());
        return RTN_ERROR;
    }
    // 打开目标文件
    fs[WRITE_FIBER].hFile = CreateFile(
        argv[2], GENERIC_WRITE,
        0, NULL, CREATE_NEW,
        FILE_FLAG_SEQUENTIAL_SCAN, NULL
        );
    if (fs[WRITE_FIBER].hFile == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile error! (rc=%lu)\n", GetLastError());
        return RTN_ERROR;
    }

    // 将主线程切换为纤程，为主纤程，只有转换为纤程后才可以切换至其他纤程
    g_lpFiber[PRIMARY_FIBER]=ConvertThreadToFiber(&fs[PRIMARY_FIBER]);
    if (g_lpFiber[PRIMARY_FIBER] == NULL)
    {
```

```
            printf("ConvertThreadToFiber failed! rc=%lu\n", GetLastError());
            return RTN_ERROR;
        }
        // 主纤程数据
        fs[PRIMARY_FIBER].dwParameter = 0;
        fs[PRIMARY_FIBER].dwFiberResultCode = 0;
        fs[PRIMARY_FIBER].hFile = INVALID_HANDLE_VALUE;
        // 创建读纤程
        g_lpFiber[READ_FIBER]=CreateFiber(0,ReadFiberFunc,&fs[READ_FIBER]);
        if (g_lpFiber[READ_FIBER] == NULL)
        {
            printf("CreateFiber error! (rc=%lu)\n", GetLastError());
            return RTN_ERROR;
        }
        // 将纤程指针作为参数传给纤程，没有实际意义，为了显示相关信息时区别各纤程
        fs[READ_FIBER].dwParameter = g_lpFiber[READ_FIBER];
        // 创建写纤程
        g_lpFiber[WRITE_FIBER]=CreateFiber(0,WriteFiberFunc,&fs[WRITE_FIBER]);

        if (g_lpFiber[WRITE_FIBER] == NULL)
        {
            printf("CreateFiber error! (rc=%lu)\n", GetLastError());
            return RTN_ERROR;
        }
        fs[WRITE_FIBER].dwParameter = g_lpFiber[WRITE_FIBER];
        // 切换到读程序执行
        SwitchToFiber(g_lpFiber[READ_FIBER]);

        // 由读纤程获写纤程切换回主纤程
        // 显示相关信息
        printf("ReadFiber result == %lu Bytes Processed == %lu\n",
            fs[READ_FIBER].dwFiberResultCode, fs[READ_FIBER].dwBytesProcessed);

        printf("WriteFiber result == %lu Bytes Processed == %lu\n",
            fs[WRITE_FIBER].dwFiberResultCode, fs[WRITE_FIBER].dwBytesProcessed);
        // 删除读写纤程
        DeleteFiber(g_lpFiber[READ_FIBER]);
        DeleteFiber(g_lpFiber[WRITE_FIBER]);
        // 关闭文件句柄、释放内存、返回
        CloseHandle(fs[READ_FIBER].hFile);
        CloseHandle(fs[WRITE_FIBER].hFile);
        HeapFree(GetProcessHeap(), 0, g_lpBuffer);
        HeapFree(GetProcessHeap(), 0, fs);
        return RTN_OK;
    }

    VOID WINAPI ReadFiberFunc( LPVOID lpParameter )
    {
        LPFIBERDATASTRUCT fds = (LPFIBERDATASTRUCT)lpParameter;

        // 判断参数
        if (fds == NULL)
        {
            printf("Passed NULL fiber data. Exiting current thread.\n");
            return;
        }
        // 显示纤程信息
        printf("Read Fiber (dwParameter == 0x%lx)\n", fds->dwParameter);
        // 初始化处理的字节数为 0
        fds->dwBytesProcessed = 0;
        // 循环读
        while (1)
```

```
    {
        if (!ReadFile(fds->hFile, g_lpBuffer, BUFFER_SIZE,
            &g_dwBytesRead, NULL))
        {
            break;
        }
        // 判断文件是否已经读完
        if (g_dwBytesRead == 0) break;

        // 已经处理的字节，累加
        fds->dwBytesProcessed += g_dwBytesRead;

        // 读一次后切换到写纤程，将读出的数据写入目标文件
        printf("Switch To Write");
        SwitchToFiber(g_lpFiber[WRITE_FIBER]);
    }
    // 读操作完成，准备交出执行，返回主纤程中
    fds->dwFiberResultCode = GetLastError();
    SwitchToFiber(g_lpFiber[PRIMARY_FIBER]);
}

VOID WINAPI WriteFiberFunc( LPVOID lpParameter )
{
    LPFIBERDATASTRUCT fds = (LPFIBERDATASTRUCT)lpParameter;
    DWORD dwBytesWritten;

    // 判断参数
    if (fds == NULL)
    {
        printf("Passed NULL fiber data. Exiting current thread.\n");
        return;
    }
    // 显示纤程信息
    printf("Write Fiber (dwParameter == 0x%lx)\n", fds->dwParameter);
    // 初始化，注意和读纤程的不同
    fds->dwBytesProcessed = 0;
    fds->dwFiberResultCode = ERROR_SUCCESS;

    while (1)
    {
        // 写入数据
        if (!WriteFile(fds->hFile, g_lpBuffer, g_dwBytesRead,
            &dwBytesWritten, NULL))
        {
            // 如果发生错误，退出循环
            break;
        }
        fds->dwBytesProcessed += dwBytesWritten;

        // 写入完成，切换到读纤程
        printf("Switch To Write");
        SwitchToFiber(g_lpFiber[READ_FIBER]);
    }
    // 出错，切换到主纤程
    // 如果写操作不出错，是不可能由写纤程切换回主纤程的
    fds->dwFiberResultCode = GetLastError();
    SwitchToFiber(g_lpFiber[PRIMARY_FIBER]);
}
```

首先来分析主纤程。

主纤程是在主线程完成了一系列准备工作后，调用 ConvertThreadToFiber 函数，由主线程转换过来。一个进程中无论有多个少线程，只要使用了纤程，就必然会有一个线程需要调用 ConvertThreadToFiber 转换为纤程。纤程虽然可以由线程创建，但是纤程在创建后并不会自动开始执行,所有纤程的执行都需要调用 SwitchToFiber 函数切换。而 SwitchToFiber 只能由纤程切换至线程。因此必然会有一个纤程是由线程调用 ConvertThreadToFiber 转换而来。

再来分析两个读写纤程。

主纤程将执行切换至读纤程，读纤程从源文件中读出数据，填入缓冲区，当缓冲区填满后，无法再读，这时将执行切换至写纤程，由写纤程从缓冲区中读出数据，写入目标文件。当写入操作完成后，再切换至读纤程。如此循环。

当一个纤程第一次开始执行时是由纤程函数的起点开始执行。一个纤程调用SwitchToFiber 切换到其他纤程执行后，其他纤程再切换回来时是由 SwitchToFiber 后的语句开始执行。一个纤程在完成它的所有任务后，不应由 FiberProc 退出，仍然应该调用SwitchToFiber 切换到其他纤程，由其他纤程来删除。

在本实例中，如果读写文件不出错，最后读纤程在读到文件尾后切换图主纤程。因此在读写纤程中，对错误的处理及返回信息的初始化有所不同。

运行结果如下：

```
Read Fiber (dwParameter == 0x15b3f0)
Switch To Write
Write Fiber (dwParameter == 0x15b6e0)
Switch To Read
Switch To Write
Switch To Read
Switch To Write
Switch To Read
…    // 在 Read 和 Write 间的切换，切换次数较多，不全部列出
Switch To Read
Switch To Write
Switch To Read
ReadFiber result == 0 Bytes Processed == 696214
WriteFiber result == 0 Bytes Processed == 696214
```

6.4 进程状态信息

获取主机的进程的相关信息是进程管理的重要内容。Windows 提供了一系列的 API 来完成进程状态的信息收集工作。这些 API 可以完成如下内容。

◇列举当前主机中的进程，包括进程 PID、句柄、程序映像路径等。

◇列举进程的线程，包括 TID、句柄等。

◇列举进程加载的模块，包括模块映像路径、句柄、位置等。

◇获取进程的内存使用情况，包括占用的物理内存、物理分页、处于各种状态的内存分页的数量。

◇列举进程的堆分配情况，包括进程的堆、在堆上分配的内存块、内存块起始地址、大小等。

◇查询工作集信息。

◇查询设置驱动信息。

◇获取主机运行性能数据。

完成这些工作，主要使用到了进程状态帮助(Process Status Helper，PSAPI)和工具帮助(Toolhelper)两类 API。

6.4.1 PS API与Tool help API

PS API 是获取进程状态信息的一系列 API 的合称，这些 API 在 Psapi.h 头文件中定义由 Psapi.dll 导出，包括如下函数：

◇ EmptyWorkingSet;

◇ EnumDeviceDrivers：
◇ EnumPageFiles;
◇ EnumProcesses;
◇ EnumProcessModules;
◇ GetDeviceDriverBaseName;
◇ GetDeviceDriverFileName;
◇ GetMappedFileName;
◇ GetModuleBaseName;
◇ GetModuleFileNameEx;
◇ GetModuleInformation;
◇ GetPerformanceInfo：
◇ GetProcessImageFileName;
◇ GetProcessMemoryInfo;
◇ GetWsChanges;
◇ InitializeProcessForWsWatch;
◇ QueryWorkingSet;
◇QueryWorkingSetEXa
和如下一些数据结构：
◇ ENUM PAGE FILE_INFORMATION;
◇ MODULEINFO;
◇ PERFORMANCE_INFORMATION;
◇ PROCESS_MEMORY_COUNTERS;
◇ PROCESS_MEMORY_COUNTERS_EX;
◇ PSAPI WS_WATCH_INFORMATION。
将在后续几节的实例中介绍 API 的用法。

Tool help API 提供了一系列 API 用于获取执行中的应用程序的信息。一般情况下可以与 PS API 配合使用。Tool helpAPI 由 Tlhelp32.h 头文件定义，keme132.dll 导出，包括如下函数：
◇CreateToolhelp32Snapshot;
◇ Heap32First;
◇ Heap32ListFirst;
◇ Heap32ListNext;
◇ Heap32Next;
◇ Module32First;
◇ Module32Next;
◇ Process32First;
◇ Process32Next;
◇ Thread32First;
◇ Thread32Next;
◇ Toolhelp32ReadProcessMemory。
和如下一些数据结构：
◇ HEAPENTRY32.
◇ HEAPLIST32：
◇ MODULEENTRY32;

◇ PROCESSENTRY32;

◇ THREADENTRY32。

使用 Tool help API 需要先使用 CreateToolhelp32Snapshot 获取快照，然后使用*First 函数和 Next 函数从快照中获取信息，一般代码都会是如下的结构：

```
handle = CreateToolhelp32Snapshot( …);
…
entry.dwSize = sizeof( XX ENTRY );
if( ! 调用 First 函数( handle , &entry) )
{
      处理出错信息
}
do
{
      相关信息处理
}while( 调用 Next 函数( handle , &entry) );
CloseHandle( handle );
```

下面几节将通过实例具体说明如何使用。

6.4.2 遍历系统中的进程

可以使用 PSAPI 的 EnumProcesses 函数和 Tool help API 的 Process32First 和 Process32Next 函数完成列举进程的工作。

EnumProcesses 可以一次性列举所有进程（以 PID 的形式返回），但是没有 Process32First 和 Process32Next 获取的信息丰富。

1．关键 API 与数据结构

（1）Enumprocesses。

获取当前系统中所有进程的 PID，函数原型如下：

```
BOOL EnumProcesses(
  DWORD* pProcessIds,
  DWORD cb,
  DWORD* pBytesReturned
);
```

◇参数

pProcessIds:输出参数，指向保存获取的进程 PID 的数组。

cb:输入参数，pProcessIds 所指向数组的大小，字节。

pBytesRetumed:输出参数，获取的进程 PID 的个数，如果数组足够大，那么 pBytesReturned 返回的是当前系统中的进程数，如果 pBytesRetumed 与 cb 相等，那么说明可能数组可能不能容纳所有进程 PID。

◇返回值

BOOL 类型，表示是否成功。

◇说明

此函数只能获取进程的 PLD，如果需要获取其他信息，可以使 OpenProcess 函数。

（2）CreateToolhelp32Snapshot。

获取系统中的进程快照，或指定进程的快照，包括线程、模块、堆等，原型如下：

```
HANDLE WINAPI CreateToolhelp32Snapshot(
  DWORD dwFlags,
  DWORD th32ProcessID
);
```

◇参数

dwFlags:输入参数,指明所需获取的快照信息,包括:TH32CS_INHERIT、TH32CS_SNAPALL、

TH32CS_SNAPHEAPLIST、TH32CS_SNAPMODULE、TH32CS_SNAPPROCESS、TH32CS_SNAPTHREAD 等。分别用于建立不同的快照，然后使用不同的 API 函数来列举。

th32ProcessID：输入参数，PID，如果是获取所有进程的快照，则设定为 0。

◇返回值

HANDLE 类型，指向快照的句柄，First 函数和 Next 函数以此为参数。

(3) Process32First。

从快照中获取第一个进程的信息。

```
BOOL WINAPI Process32First(
  HANDLE hSnapshot,
  LPPROCESSENTRY32 lppe
);
```

◇参数

hSnapshot：输入参数，CreateToolhelp32Snapshot 函数返回的快照句柄，调用 CreateToolhelp32Snapshot 函数时，使用 TH32CS_SNAPPROCESS 参数。

lppe：输入输出参数，PROCESSENTRY32 结构变量的指针，结构的 dwSize 字段就设置为 PROCESSENTRY32 结构的大小。

◇返回值

BOOL 类型的数据，表示是否成功。

(4) Process32Next。

从快照中依次获取进程的信息。

```
BOOL WINAPI Process32Next(
  HANDLE hSnapshot,
  LPPROCESSENTRY32 lppe
);
```

◇参数

使用方法同 Process32First 函数的参数。

◇返回值

BOOL 类型的数据，表示是否成功。

2．实例 6-9　EnumProcesses 函数列举线程

本实例是使用 EnumProcesses 函数获取所有进程的 PID，然后使用 OpenProcess、函数获取各进程的句柄，然后获取相关信息。在实例中调用了 ListProcessModulesl 函数和 ListProcessThreads 函数，用于列举进程的模块和线程，这两个函数参见实例 6-14 和 6-15

```
/************************************          6-11
* VOID WINAPI EnumProcess1()
* 功能：调用 EnumProcesses 遍历进程，
*       并调用 ListProcessModules1 函数和
```

```
*      ListProcessThreads 函数列举模块和线程
**************************************/
VOID WINAPI EnumProcess1()
{
     // 假设不超过 1024 个进程
     DWORD aProcesses[1024], cbNeeded, cProcesses;
     unsigned int i;
     // 调用 EnumProcesses
     if ( !EnumProcesses( aProcesses, sizeof(aProcesses), &cbNeeded ) )
          return;
     // 进程数
     cProcesses = cbNeeded / sizeof(DWORD);
     for ( i = 0; i < cProcesses; i++ )
     {
          // 显示进程信息
          printf( "\n\n***************************************************" );
          printf("\nPROCESS : %u\n\n",aProcesses[i]);
          printf( "\n***************************************************" );
          // 列举模块信息和线程信息
          ListProcessModules1( aProcesses[i] );
          ListProcessThreads( aProcesses[i] );
     }
}
```

EnumProcess2 函数是使用 Tool help 函数列举进程，并显示相关信息。在实例中调用了 ListProcessModules2 函数、PrintMemoryInfo 函数和 ListHeapInfo 函数，用于列举进程的模块，获取内存使用情况和列举堆分配情况，参见实例 6-13 和实例 6-14。

```
/**************************************
* VOID WINAPI EnumProcess2()
* 功能: 调用 Process32First 和 Process32Next 遍历进程,
*       并调用 ListProcessModules2 函数列举模块,
*       调用 ShowProcessMemoryInfo 函数显示内存使用情况
*
* 无参数, 无返回值
**************************************/
VOID WINAPI EnumProcess2()
{
     HANDLE hProcessSnap;
     HANDLE hProcess;
     PROCESSENTRY32 pe32;
     DWORD dwPriorityClass;
     // Snapshot
     hProcessSnap = CreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );
     if( hProcessSnap == INVALID_HANDLE_VALUE )
     {
          PrintError( "CreateToolhelp32Snapshot (of processes)" );
          return( FALSE );
     }
     // 设置输入参数, 结构的大小
     pe32.dwSize = sizeof( PROCESSENTRY32 );

     // 开始列举进程
     if( !Process32First( hProcessSnap, &pe32 ) )
     {
          PrintError( "Process32First" );  // 出错信息
          CloseHandle( hProcessSnap );
          return( FALSE );
     }
     do
     {
          // 打印进程名
          printf( "\n\n==================================================" );
```

```
            printf( "\nPROCESS NAME:  %s", pe32.szExeFile );
            printf( "\n-------------------------------------------------------" );

            // 获取优先级
            dwPriorityClass = 0;
            hProcess = OpenProcess( PROCESS_ALL_ACCESS, FALSE, pe32.th32ProcessID );
            if( hProcess == NULL )
                PrintError( "OpenProcess" );
            else
            {
                dwPriorityClass = GetPriorityClass( hProcess );
                if( !dwPriorityClass )
                     PrintError( "GetPriorityClass" );
                CloseHandle( hProcess );
            }
            // 打印进程相关信息
            printf( "\n  process ID       = 0x%08X", pe32.th32ProcessID );
            printf( "\n  thread count     = %d",   pe32.cntThreads );
            printf( "\n  parent process ID = 0x%08X", pe32.th32ParentProcessID );
            printf( "\n  Priority Base    = %d", pe32.pcPriClassBase );
            if( dwPriorityClass )
                printf( "\n  Priority Class   = %d", dwPriorityClass );

            // 获取模块信息，显示内存使用情况
            ListProcessModules2( pe32.th32ProcessID );
            PrintMemoryInfo(pe32.th32ProcessID);
            ListHeapInfo(pe32.th32ProcessID);

        } while( Process32Next( hProcessSnap, &pe32 ) );

        CloseHandle( hProcessSnap );     //关闭句柄
        return( TRUE );
    }
```

3. 实例 6-10 通过进程名查找进程 ID

本实例是列举进程的另外一种应用，通过进程名查找进程 ID。是实例 6-7 中调用的 GetProcessIdByName 函数的代码。

```
/*************************************
* BOOL GetProcessIdByName(LPSTR szProcessName, LPDWORD lpPID)
* 功能：通过进程名获取进程 PID
* 参数：LPSTR szProcessName  进程名
*       LPDWORD lpPID        指向保存 PID 的变量
* 返回是否成功
*************************************/
BOOL GetProcessIdByName(LPSTR szProcessName, LPDWORD lpPID)
{
    // 变量及初始化
    STARTUPINFO st;
    PROCESS_INFORMATION pi;
    PROCESSENTRY32 ps;
    HANDLE hSnapshot;
    ZeroMemory(&st, sizeof(STARTUPINFO));
    ZeroMemory(&pi, sizeof(PROCESS_INFORMATION));
    st.cb = sizeof(STARTUPINFO);
    ZeroMemory(&ps,sizeof(PROCESSENTRY32));
    ps.dwSize = sizeof(PROCESSENTRY32);
    // 遍历进程
    hSnapshot = CreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0);
    if(hSnapshot == INVALID_HANDLE_VALUE)
    {
```

```
        return FALSE;
    }

    if(!Process32First(hSnapshot,&ps))
    {
        return FALSE;
    }
    do
    {
        // 比较进程名
        if(lstrcmpi(ps.szExeFile,"explorer.exe")==0)
        {
            // 找到了
            *lpPID = ps.th32ProcessID;
            CloseHandle(hSnapshot);
            return TRUE;
        }
    }
    while(Process32Next(hSnapshot,&ps));
    // 没有找到
    CloseHandle(hSnapshot);
    return FALSE;
}
```

6.4.3 列举进程的模块、线程

PSAPI 提供了 EnumProcessModules、 GetModuleB aseName、 GetModuleFileNameEx、GetModuleInformation 等函数用于获取进程的模块信息。

Tool Help API 也提供了 Module32First 和 Module32Nex 函数用于获取类似信息。

1．关键 API 与数据结构

API 函数的定义与数据结构较简单，可以参见上一节和本节实例。

2．实例 6-11 通过 EnumProcessModules 列举模块

本实例说明通过 EnumProcessModules 等函数获取进程加载的模块信息。

```
/*************************************
* VOID ListProcessModules1( DWORD dwPID )
* 功能：调用 EnumProcessModules 函数
*       列举和显示进程加载的模块
*
* 参数：DWORD dwPID 为进程 PID
*************************************/
VOID ListProcessModules1( DWORD dwPID )
{
        HMODULE hMods[1024];
        HANDLE hProcess;
        DWORD cbNeeded;
        unsigned int i;

        printf( "\nListProcessModules1 Process ID %u\n", dwPID );

        // 打开进程，获得句柄
        hProcess = OpenProcess( PROCESS_QUERY_INFORMATION |
            PROCESS_VM_READ,
            FALSE, dwPID );
        if (NULL == hProcess)
            return;
```

```
        // 调用 EnumProcessModules
        if( EnumProcessModules(hProcess, hMods, sizeof(hMods), &cbNeeded))
        {
                for ( i = 0; i < (cbNeeded / sizeof(HMODULE)); i++ )
            {
                TCHAR szModName[MAX_PATH];
                // 获取模块路径
                if ( GetModuleFileNameEx( hProcess, hMods[i], szModName,
                    sizeof(szModName)/sizeof(TCHAR)))
                {
                    // 打印
                    printf( TEXT("\t%s (0x%08X)\n"), szModName, hMods[i] );
                }
            }
        }
        CloseHandle( hProcess );    // 关闭进程句柄
}
```

3．实例 6-12 通过快照列举进程模块

本实例说明通过 CreateToolhelp32Snapshot、Module32First、Module32Next 等函数获取进程加载的模块信息。

```
/*************************************
* VOID ListProcessModules2( DWORD dwPID )
* 功能：调用 Module32First 和 Module32Next 函数
*       列举和显示进程加载的模块
*
* 参数：DWORD dwPID 为进程 PID
*************************************/
VOID ListProcessModules2( DWORD dwPID)
{
    HANDLE hModuleSnap = INVALID_HANDLE_VALUE;
    MODULEENTRY32 me32;
    printf( "\nListProcessModules2 Process ID %u\n", dwPID );
    // Snapshot
    hModuleSnap = CreateToolhelp32Snapshot( TH32CS_SNAPMODULE, dwPID );
    if( hModuleSnap == INVALID_HANDLE_VALUE )
    {
        PrintError( "CreateToolhelp32Snapshot (of modules)" );
        return( FALSE );
    }
    // 设置输入参数、结构的大小
    me32.dwSize = sizeof( MODULEENTRY32 );
    // 开始获取模块信息
    if( !Module32First( hModuleSnap, &me32 ) )
    {
        PrintError( "Module32First" );  // Show cause of failure
        CloseHandle( hModuleSnap );     // Must clean up the snapshot object!
        return( FALSE );
    }
    do
    {
        printf( "\n\n     MODULE NAME:     %s",          me32.szModule );
        printf( "\n     executable     = %s",          me32.szExePath );
        printf( "\n     process ID     = 0x%08X",      me32.th32ProcessID );
        printf( "\n     ref count(g)   =     0x%04X",  me32.GlblcntUsage );
        printf( "\n     ref count(p)   =     0x%04X",  me32.ProccntUsage );
        printf( "\n     base address = 0x%08X",(DWORD) me32.modBaseAddr );
        printf( "\n     base size      = %d",          me32.modBaseSize );
```

```
    } while( Module32Next( hModuleSnap, &me32 ) );
    CloseHandle( hModuleSnap ); // 关闭句柄
    return( TRUE );
}
```

```
/************************************
* VOID PrintMemoryInfo( DWORD dwPID )
* 功能: 显示进程的内存使用情况
* 参数: DWORD dwPID 为进程 PID
************************************/
VOID PrintMemoryInfo( DWORD dwPID )
{
    HANDLE hProcess;
    PROCESS_MEMORY_COUNTERS pmc;
    printf( "\nProcess ID: %u\n", dwPID );
    hProcess = OpenProcess(  PROCESS_QUERY_INFORMATION |
        PROCESS_VM_READ,
        FALSE, dwPID );
    if(NULL == hProcess)
        return;

    if ( GetProcessMemoryInfo( hProcess, &pmc, sizeof(pmc)) )
    {
        printf( "\tPageFaultCount: 0x%08X\n", pmc.PageFaultCount );
        printf( "\tPeakWorkingSetSize: 0x%08X\n",
            pmc.PeakWorkingSetSize );
        printf( "\tWorkingSetSize: 0x%08X\n", pmc.WorkingSetSize );
        printf( "\tQuotaPeakPagedPoolUsage: 0x%08X\n",
            pmc.QuotaPeakPagedPoolUsage );
        printf( "\tQuotaPagedPoolUsage: 0x%08X\n",
            pmc.QuotaPagedPoolUsage );
        printf( "\tQuotaPeakNonPagedPoolUsage: 0x%08X\n",
            pmc.QuotaPeakNonPagedPoolUsage );
        printf( "\tQuotaNonPagedPoolUsage: 0x%08X\n",
            pmc.QuotaNonPagedPoolUsage );
        printf( "\tPagefileUsage: 0x%08X\n", pmc.PagefileUsage );
        printf( "\tPeakPagefileUsage: 0x%08X\n",
            pmc.PeakPagefileUsage );
    }
    CloseHandle( hProcess );
}
```

6.4.4 进程的堆使用、内存占用、虚拟内存大小，页面错误情况

PSAPI 函数 GetProcessMemoryInfo 可以获取一个进程的内存使用情况，包括分页错误计数、工作集大小、工作集峰值、分页文件大小，分页文件使用峰值、分页池、未分页池等情况。

Tool Help API 提供的 Heap32First、Heap32ListFirst、Heap32ListNext、Heap32Next 等函数可以用于获取进程的堆分配信息。

1．实例 6-13 获取进程内存使用情况

本实例通过 GetProcessMemoryInfo 函数获取进程内存使用情况，并打印。

2．实例 6-14 通过快照列举进行中的堆

本实例说明通过 CreateToolhelp32Snapshot、Heap32ListFirst、Heap32Next 等函数列举进程的堆。

```
/************************************
* VOID ListHeapInfo( DWORD dwPID )
* 功能: 显示进程的堆分配情况
* 参数: DWORD dwPID 为进程 PID
```

```
************************************/
VOID ListHeapInfo( DWORD dwPID )
{
        HEAPLIST32 hl;
        HEAPENTRY32 he;
        HANDLE hSnapshot = INVALID_HANDLE_VALUE;
        printf( "\ListHeapInfo Process ID %u\n", dwPID );

        //  Snapshot
        hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPHEAPLIST , dwPID );
        if( hSnapshot == INVALID_HANDLE_VALUE )
        {
             PrintError( "CreateToolhelp32Snapshot (of heaplist)" );
             return( FALSE );
        }
        // 设置输入参数、结构的大小
        hl.dwSize = sizeof( HEAPLIST32 );

        // 开始获取信息
        if( !Heap32ListFirst( hSnapshot, &hl ) )
        {
             PrintError( "Heap32ListFirst" );
             CloseHandle( hSnapshot );
         return( FALSE );
        }
    do
    {
         printf( "\n\tHeap ID    =%u", hl.th32HeapID );
         printf( "\tHeap Flags     = %u", hl.dwFlags );
         he.dwSize = sizeof(HEAPENTRY32);
         if( !Heap32First(&he,dwPID,hl.th32HeapID) )
         {
              PrintError( "Heap32First" );  // 出错
              CloseHandle( hSnapshot );
              return( FALSE );
         }
         do
         {
              // 显示信息
              printf( "\n\t\t Heap Address\t= %u",he.dwAddress );
              printf( "\t Heap Size\t= %u",he.dwBlockSize);
              printf( "\t Heap Flags\t= %u",he.dwFlags);
              printf( "\t Heap Handle\t= %u",he.hHandle);
         } while( Heap32Next(&he ));
    } while( Heap32ListNext( hSnapshot, &hl ) );
    CloseHandle( hSnapshot );   // 关闭句柄
    return( TRUE );
}
```

6.5 动态链接库

动态链接库是将程序模块化的重要方法，应用程序开发中，需要在多个进程和模块间共用代码的情况下，动态链接库是很好的解决方法。如何开发和使用动态链接库是每个Windows 应用程序开发人员都需要可能遇到的问题。

系统中存在着很多的系统 DLL，如 Kernel32.dll、User32.dll 等。开发人员也可以自行开发 DLL。

本书中介绍的 API 函数都存在于系统 DLL 中。调用 DLL 的函数可以在编译连接的时候通过构造导入表完成（Windows 实现导入表和导出表的内部原理详见 18 章），之前的所有实例都是采用这种方法，是由编译连接器会自行完成，也可以使用 LoadLibray 函数和 GetProcAddress 函数在进程中自行加载动态链接库，调用库中的函数。

6.5.1 加载、释放DLL、通过句柄获取DLL相关信息

本节将介绍如何加载和释放 DLL，并通过句柄获取 DLL 的相关信息。

在进程中加载 DLL 使用 API 函数 LoadLibrary，加载成功后可以获取模块的句柄，通过模块的句柄可以得到 DLL 中的相关数据、获取函数地址，获取其他相关信息。释放加载的 DLL 使用函数 FreeLibrary。

GetDllDirectory 和 GetModuleFileName 函数可以用于获取 DLL 和模块所在的路径及文件名。

1．关键 API

（1）LoadlLibrary。

获取模块的句柄，如果模块未加载，则将模块加载入进程，函数原型如下：

```
HMODULE LoadLibrary(
  LPCTSTR lpFileName
);
```

lpFileName 是可执行程序的路径，返回加载后模块的句柄。

（2）FreeLibrary。

释放已经加载的模块，以模块句柄为参数，函数原型如下：

```
BOOL FreeLibrary(
  HMODULE hModule
);
```

2．实例 6-15 动态加载 DLL

本实例使用 LoadLibrary 加载 DLL，获得模块路径等相关信息后释放模块。

6.5.2 编写动态链接库、导出函数

本节将介绍如何编写一个动态链接库。如果需要编译本节的示例代码，请参考 6.3.3 小节。

导出函数是 DLL 中函数的一种，它为其他模块提供调用接口，函数一旦被导出，其他模块就可以获取导出的函数地址并进行调用。

```
// 加载 Dll，获取 DLL 路径相关信息
VOID GetDllInfo()
{
    HINSTANCE hinstLib;
    CHAR szDllDir[MAX_PATH];
    CHAR szDllName[MAX]_PATH];
    // 获得 DLL 模块句柄
    hinstLib = LoadLibrary(TEXT("msg.dll"));
    // 获得 DLL 目录
    GetDllDirectory(MAX_PATH, szDllDir);
    // 获得 DLL 路径
    GetModuleFileName(hinstLib,szDllName,MAX_PATH);
    // 打印输出
    printf("dll dir:\t %s\ndll path:\t %s",
        szDllDir,szDllName);
    // 释放 DLL
    FreeLibrary(hinstLib);
}
```

1．DllMain 函数

和 exe 文件一样，动态链接库也有自己的入口函数，默认为 DllMain 函数。DllMain 函数在 DLL 被加载和释放以及 DLL 所在的进程中有新的线程被创建和退出的时候被调用。

DllMain 函数的原型如下：

```
BOOL WINAPI DllMain(
  HINSTANCE hinstDLL,
  DWORD fdwReason,
  LPVOID lpvReserved
);
```

参数 hinstDLL 是 DLL 的句柄，fdwReason 说明 DllMain 在什么情况下被调用，lpvReserved
参数未使用。使用方法见实例 6-17。

2．导出函数

如果需要将 Dll 中的函数导出，有多种方法。一是在函数声明时使用_declspec(dllexport)关
键字，详见实例 6-17;二是在模块定义文件(. def)中进行说明，详见实例 6-18。

3．实例 6-16 msg．dll 的代码

msg.dll 包括两个函数，一个是 DllMain 函数，一个是导出函数 ExportExample。

```
/* ************************************
* msg.c 动态链接库
**************************************/
/* 头文件 */
#include <Windows.h>
#include <Psapi.h>
/* 链接 */
#pragma comment (lib, "Psapi.lib")
/* 函数声明 */

// 使用__declspec(dllexport)声明导出函数
__declspec(dllexport) DWORD ExportExample(LPSTR szMsg, DWORD dwCode);

/************************************
* DllMain
************************************/
BOOL WINAPI DllMain(
                        HINSTANCE hinstDLL,     // DLL 模块的句柄
                        DWORD fdwReason,        // 调用的情况
                        LPVOID lpReserved )     // reserved
{
    // 在不同的情况下都会调用 DllMain 函数，分别处理
    switch( fdwReason )
    {
        // 加载 DLL
    case DLL_PROCESS_ATTACH:
        {
            CHAR lpMainMoudleName[MAX_PATH];
            CHAR lpMessage[MAX_PATH+64];
            // 获取 PID 和主模块名，将弹出消息框
            DWORD dwPID = GetCurrentProcessId();
            GetModuleBaseName(GetCurrentProcess(),NULL,lpMainMoudleName,MAX_PATH);
            wsprintf(lpMessage,"Process name: %s, PID: %u ",lpMainMoudleName,dwPID);
            MessageBox(NULL,lpMessage,"msg.dll",MB_OK);
            break;
        }
        // 新建线程
```

```
        case DLL_THREAD_ATTACH:
             break;
             // 线程退出
        case DLL_THREAD_DETACH:
             break;
             // 释放 DLL
        case DLL_PROCESS_DETACH:
             break;
        }
        return TRUE;
    }

    /************************************
    * DWORD ExportExample(LPSTR szMsg, DWORD dwCode)
    * 功能: 导出函数, 显示消息
    *
    * 参数: LPSTR szMsg, 字符串; DWORD dwCode, 整形
    ************************************/
    DWORD ExportExample(LPSTR szMsg, DWORD dwCode)
    {
        LPVOID lpShowOut = HeapAlloc(GetProcessHeap(),NULL,lstrlen(szMsg)+100);
        wsprintf(lpShowOut,"%s,%d",szMsg,dwCode);
        MessageBox(NULL,lpShowOut,"由导出函数弹出的消息! ",MB_OK);
        HeapFree(GetProcessHeap(),NULL,lpShowOut);
        return 0;
    }
```

从实例代码可以看到，声明 ExportExample 时，使用了“_declspec(dllexport)”，这可以导出函数。

```
__declspec(dllexport) DWORD ExportExample(LPSTR szMsg, DWORD dwCode);
```

4．实例 6-17 通过 def 文件导出函数

除了在实例 6-17 中提到的，在函数声明或定义时使用_declspec(dllexport)外，还可以在 def 中导出函数，比如：

```
LIBRARY  "env"

EXPORTS
    EnumEnvironmentVariables
    ChangeEnviromentVariables
```

这个 def 是工程“env_dll”（实例 6-3）中模块定义文件的程序代码所对应的模块定义文件，工程 env 就是编译生成 dll。在这个 def 文件中，定义了两个导出函数 EnumEnvironmentVariables 和 ChangeEnviromentVariables.

def 文件还可以用于其他比较丰富的内容。

6.5.3 创建动态链接库工程，配置DLL编译链接选项

link.exe 的链接选项中有几个是专门为创建动态链接库而设计的。以 6.2.3 小节中的实例 6-3 为例按如下步骤为其配置工程，并编译生成 DLL 文件。

◇在 Visual Studio 菜单中选择“文件”→“新建”→“项目”。

◇在“新建项目”对话框中选择“Visual C++” →“空项目”，输入项目名称 Example 和位置。

◇在工程右键菜单中选择添加新建项，选择“代码”→“C-H-文件(. cpp)”，输入文件名 env.c。

◇将实例 6-3 中的代码复制到 env.c 中。

◇在工程右键菜单中选择添加新建项，选择“代码”→“模块定义文件(def)”，输入文件名 exp.def。

◇将实例 6-18 中的代码复制到 exp.def 中。
◇在工程右键菜单中选择添加新建项，选择“代码”→“头文件(.h)”，输入文件名 env.h。
◇将实例 6-21 中的代码复制到 env.h 中。
◇在“解决方案管理器”中打开新建工程的属性。
◇在“配置属性”→“常规”中，将“配置类型”修改为“动态库(. dll)”。
◇在“配置属性”→“链接器”一“常规”中，将“输出文件”的扩展名改为 dll。
◇确认：“配置属性”一“链接器”→“输入”中“模块定义文件”的值为“exp.def”。
◇参考第 3 章，读者可自行修改调试等编译链接选项。
实例 6-18 编译链接生成 DLL
本实例是一个可以将实例 6-3 编译为 DLL 的编译链接选项，带调试信息。
编译器选项：

```
    /Od /Fo"Debug\\" /Fd"Debug\vc80.pdb" /nologo /c /Zi /Gz /TC
```

链接器选项：

```
    /OUT:"..\Debug\msg.dll" /NOLOGO /DLL /DEF:"exp.def" /DEBUG /PDB:"..\debug\ msg.pdb"
kernel32.lib user32.lib gdi32.lib Psapi.lib
```

有关编译链接选项的内容可以参考第 3 章。

6.5.4 运行时动态获取DLL导出函数地址并调用

这种方法与本书之前的实例中调用 API 的方法不同，一般情况下，调用 API 函数是通过头文件中的函数声明和定义的 lib 文件进行的，需要在连接时指定程序所在 lib 文件，这种方法进行调用会在程序可执行文件中创建导入表，并在程序加载时进行函数的链接。而本节介绍的方法是通过API函数GetProcAddress_获取 DLL导出函数的地址然后再通过函数指针调用，不会在导入表中出现调用的函数。

通过 LoadLibrary 获取模块句柄后，可以使用 GetProcAddress 获取函数地址，并通过函数地址调用 DLL 中的导出函数。这种方法进程在创建时并不进行函数的链接，程序自行发现函数的地址并调用，因为是通过函数地址进行调用，所以需要事先定义函数指针类型。

1.关键 API

GetProcAddress.

载了的模块中导出函数的地地址，原型如下：

```
FARPROC GetProcAddress(
  HMODULE hModule,
  LPCSTR lpProcName
);
```

参数 hModule 是模块句柄，lpProcName 是函数名，返回函数指针。

2．实例 6-19 函数运行时链接

本实例演示使用 LoadLibrary 和 GetProcAddress 函数进行运行时链接。首先使用 LoadLibrary 加载 DLL，获取句柄；然后通过 GetProcAddress 获得函数地址；再通过函数指针调用函数。

```
/* ************************************
* Loader.c 动态链接库
**************************************/
/* 头文件 */
#include <stdio.h>
#include <windows.h>
```

```
/* 类型定义 */

// 指向函数的指针类型
typedef DWORD (*MY_EXPORT_PROC)(LPSTR, DWORD);

// main
VOID main(VOID)
{
    HINSTANCE hinstLib;
    MY_EXPORT_PROC lpProcAdd;
    BOOL fFreeResult, fRunTimeLinkSuccess = FALSE;
    // 获得 DLL 模块句柄
    hinstLib = LoadLibrary(TEXT("msg.dll"));
    // 是否加载成功
    if (hinstLib != NULL)
    {
        // 获得指定导出函数的地址
        lpProcAdd = (MY_EXPORT_PROC) GetProcAddress(hinstLib, "ExportExample");
        // 判断是否成功
        if (NULL != lpProcAdd)
        {
            fRunTimeLinkSuccess = TRUE;
            (lpProcAdd) ("hello",123);
        }
        // 释放 DLL
        fFreeResult = FreeLibrary(hinstLib);
    }
    // 出错
    if (! fRunTimeLinkSuccess)
        printf("error; %u\n",GetLastError());
}
```

ExportExample 的作用是将指定的消息用对话框显示，msg.dll 代码见示例 6-17。
运行后弹出如图 6-6 和图 6-7 所示的对话框。

图 6-6　加载 DLL 后，由 DLL 中的程序弹出的对话框

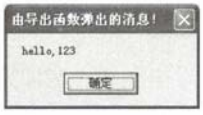

图 6-7　调用导出函数后弹出的对话框

第一个对话框是在 DLL 加载时弹出，代码定义于 msg.dll 的 DllMain 函数中，第二个对话框是由 ExportExample 弹出。

6.5.5　声明导出函数、创建lib库，为其他模块提供导入表调用接口

如果使用 6.5.4 小节所介绍的方法调用导出函数有诸多的不便，那么如何才能使调用自己编写的 DLL 的导出函数与调用系统 API -样方便呢？这就需要在头文件中提供导出函数的声明，在编写其他模块的时候，还需要为链接器提供链接时所需要的 lib 库，以使链接器知道如何创建导入表。

可以通过实例 6-21 所述步骤实现。

实例 6-20 导出函数声明与 lib 文件，供其他模块链接使用

实现类似于调用系统 API -样调用自行编写的 API 函数，使用在 6.2.3 小节中所使用的两个实例工程来作演示。env_ dll 工程是 DLL 工程，env 工程是 exe 工程。工程文件在配套光盘 ch6 目录下。

如下代码是 env_dll 工程的头文件 env.h 的内容。

```
#include <windows.h>

#define VARIABLES_APPEND    1
#define VARIABLES_RESET     2
#define VARIABLES_NULL      0

DWORD WINAPI EnumEnvironmentVariables();
DWORD WINAPI ChangeEnviromentVariables(
                                        LPSTR szName,
                                        LPSTR szNewValue,
                                        DWORD dwFlag);
```

在头文件中声明了两个函数：EnumEnvironmentVariables、ChangeEnviromentVariables，及一些所可能用到的常量值。

构建 env_dll 工程，生成了 env_dll.dll 文件和 env_dll.lib 文件之后，可以在其他工程中通过头
文件和 lib 文件来调用 DLL 中的导出文件，与调用系统 API 的方式相同，具体过程见 6.5.6 小节。

6.5.6 通过构建导入表调用DLL导出函数

◇编译生成 evn_dll 工程。生成了 env_dll.dll 文件和 env_dll.lib 文件，env 工程需要使用到这个 lib 文件。

◇在 env 工程的代码中声明引用这些头文件(env_dll 工程位于“Environment Variables”目录下)。

```
#include "../Environment Variables/env.h"
```

◇在 env 工程链接时，指定“附件依赖项”，指向 env~dll.lib 文件。即在链接的输入文件中 加入 env_dll.lib。

◇ env 工程的源文件调用了 env.h 中声明的函数，env.c 代码参见实例 6-3。

```
/OUT:"..\Debug\env.exe" /NOLOGO kernel32.lib user32.lib gdi32.lib advapi32.lib shell32.lib
ole32.lib ..\Debug\env_dll.lib
```

◇ 编译生成 env.exe。

◇将 env.exe 和 env_dll.dll 放置在同一目录下，运行 env.exe。正常运行，说明通过动态链接调用 env_dll.dll 中的函数成功。

这样调用用户自行定义的导出函数就与调用系统 API 的方式一样了。在这样构造的文件中，需要调用的函数位于模块的导入表中，查看 env.exe 的导入表可以看到其导入了 env_dll.dll 的两个函数。

> **注意** 如果实际目标 DLL（如上例的 Example.dll）并不存在，那么进程在初始化时由于找不到需要调用的函数所在的模块就会失败，程序不会运行。而如果使用实例 6-11 中的方法，进程的初始化会成功，程序会运行，但是 LoadLibrary 找不到目标 DLL，会返回 FALSE。

第 7 章　线程同步

由于线程采用并发运行机制，因此当进程运行时，其各线程的运行进度可能不一致。在某些情况下需要对各线程的运行过程进行同步。

多线程程序的线程一般需要协同工作。

比如一个进程具有两个线程 A 和 B，线程 A 负责处理数据，线程 B 负责将数据写入到文件。

它们需要访问同一段内存区域，线程 A 写，线程 B 读，要求在线程 A 写入后，线程 B 才能读，而线程 A 只有在线程 B 完成读取后，才能写入新数据。此时线程 A 和 B 在时间进度上具有相关性。如果 A、B 不知道对方的进度，则可能出现 A 处理完毕的数据在 B 读取之前又被 A 重新写入，或者 A 还未写入新数据 B 就再次读取导致多次重复读取相同数据等情况，从而造成程序逻辑错误。

线程同步机制是为是各线程能够协同工作而设计的。

Windows 系统中有很多种机制可以用于线程同步，最常用的有下面几种：

◇互斥对象(Mutex)；

◇　事件对象(Event)；

◇信号量(Semaphore)；

◇　临界区(critical section)；

◇　可等待计时器(Waitable Timer)。

7.1　基本原理

线程同步在多线程程序中具有重要意义。程序员没有处理好线程同步是造成程序 Bug 和程序不稳定的最主要因素之一。

在同步过程中，两个最重要的概念是同步对象和等待函数。

在线程同步过程中，需要先定义一个同步对象，同步对象一般具有两种状态：标志的(置位，signaled)和未标志的(未置位，nonsignaled)。线程根据是否已经完成操作将同步对象设置为标志的或未标志的。

而等待函数的功能是专门用于等待同步对象状态改变。一个线程调用等待函数后执行会暂停，直到同步对象的状态改变后，等待函数才会返回，线程才会继续执行。等待函数分为“单对象”等待函数和“多对象”等待函数。

7.1.1　线程同步的过程

同步对象与等待函数相互配合以实现线程同步。比如：线程 A 在进行某种操作前需要线程 B 为其准备好数据，那么这种时候就需要线程同步。线程 A 会等待线程 B 的执行，直到需要的数据准备好。那么使用同步对象和等待函数，其过程基本如下。

◇在需要进行线程同步的进程中定义某种同步对象，同步对象必需是全局的，以保证需要同步的线程 A 和 B 都可以访问到同步对象。

◇开始时，线程 A 和 B 相互独立地执行。

◇线程 B 在准备好线程 A 需要使用到的数据前，将同步对象置为“未标记的”，线程 B 在准备好线程 A 需要使用的数据后，改变同步对象状态，置为“标记的”。

◇线程 A 运行，直到需要线程 B 为其准备的数据时，调用等待函数。如果同步对象不是“标记的”，则一直等待到同步对象的状态改变到“标记的”为止。同步对象被 B 设置为“标记的”后（表示线程 B 已经完成数据准备工作），等待函数才会返回，线程 A 继续执行。

◇依此类推，循环往复。

7.1.2 同步对象

最经常使用到的同步对象包括：事件(Event)、互斥(Mutex)、信号量(Semaphore)、可等待计时器(Waitable timer)。将在后续章节中陆续介绍，

还有一些对象也可以用于同步。这些对象也可以作为等待函数等待的对象，包括临界区、Change notification、控制台输入(Console input)、作业(Job)、线程、进程等。

7.1.3 等待函数

等待函数在同步过程中起了非常重要的作用。等待函数有 4 种，分别为单对象(single-object)等待函数、多对象(multiple-obj ect)等待函数、可告警(alertable)等待函数和注册的(registered)等待函数。常用的等待函数包括：

◇ SignalObjectAndWait;

◇WaitForSingleObject;

◇WaitForSingleObjectEx;

◇ WaitForMultipleObjects;

◇ WaitForMultipleObjectsEx;

◇ MsgWaitForMultipleObjects;

◇MsgWaitForMultipleObjectsEx;

◇ MsgWaitForMultipleObjectsEx;

◇ SignalObjectAndWait;

◇ WaitForMultipleObjectsEx;

◇ WaitForSingleObjectEx;

◇ RegisterWaitForSingleObject。

其中 WaitForSingleObj ect 和 WaitForMultipleObjects 函数最常用。

(1) WaitForSingleObject。

WaitForSingleObj ect 的功能是等待单个对象，如果对象置位，则返回。函数原型如下：

```
DWORD WINAPI WaitForSingleObject(
  HANDLE hHandle,
  DWORD dwMilliseconds
);
```

参数 hHandle 是等待的对象的句柄，参数 dwMilliseconds 是等待超时的时间，如果需要无限等待，设置为 INFINITE。

(2) WaitForMultipleObjectS。

WaitForMultipleObjects 的功能是等待多个对象，当所等待的多个对象都设置为标记的，或多个对象中的一个对象设置为标记，则返回，函数原型如下：

```
DWORD WINAPI WaitForMultipleObjects(
  DWORD nCount,
  const HANDLE* lpHandles,
  BOOL bWaitAll,
  DWORD dwMilliseconds
);
```

nCount 是等待的对象的数量；所有需要等待的对象的句柄依次放置在数组中，lpHandles 参数指向数组的第一个元素，数组中句柄元素的个数应该为 nCount; bWaitAll 参数如果为 TRUE 说明等待对象全部置位才返回，如果为 FALSE 则等待的对象之一置位则返回；dwMilliseconds 参数
与 WaitForSingleObj ect 的 dwMilliseconds 参数意义相同。

7.2 同步对象示例

本节将通过实例说明各种同步对象(Event、Mutex、Semaphore、Timer)和等待函数在不同的情况下如何完成同步。

7.2.1 使用事件对象(Event)

事件(Event)是在线程同步中最常使用的一种同步对象。可以使用 CreateEvent API 函数创建事件对象；使用 SetEvent 函数可将事件对象设置为“标记的”；使用 ResetEvent 函数将事件重置为“未标志的”状态。使用等待函数可以等待事件对象状态的改变。如实例 7-1 和 7-2。

1. 关键 API

(1) CreateEvent、CreateEventEX.

函数的功能是创建事件对象，函数原型如下：

```
HANDLE WINAPI CreateEvent(
  LPSECURITY ATTRIBUTES lpEventAttributes,
  BOOL bManualReset,
  BOOL bInitialState,
  LPCTSTR lpName
);
```

◇参数

lpEventAttributes:输入参数，安全属性，如果没有特殊的权限设定参数设置为 NULL。

bManualReset:输入参数，是否进行手工重置，所谓重置是指将已标记的事件对象再重新设置为未标记的。如果是手工重置，那需要使用 ResetEvent API 来重置，如果是非手工重置的，那么等待函数在等待到事件置位后，自动将事件重置。

bInitialState:输入参数，表示所创建的对象的初始是否置位。如果为 TRUE 则为事件置位。

lpName:输入参数，事件名，可以为 NULL。

◇返回值

事件对象的句柄。

(2) SetEvent。

功能是将事件对象设置为标记，函数原型如下：

```
BOOL WINAPI SetEvent(
  HANDLE hEvent
);
```

(3) ResetEvent。

功能是重置标记，如果事件是手工重置的，那么需要使用此函数来重置事件。函数原型如下：

```
BOOL WINAPI ResetEvent(
  HANDLE hEvent
);
```

(4) OpenEvent。

功能是从事件名中得到事件句柄。函数原型如下：

```
HANDLE WINAPI OpenEvent(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName
);
```

2．实例 7-1　使用事件对象进行线程同步

本实例演示使用事件进行同步。

实例中 UseEvents 先创建了一个事件对象，然后创建了线程，创建完线程后，经过一段时间，向内存中复制数据然后再置位事件。被创建的线程 EventFunction 调用 WaitForSingleObj ect 函数等待事件置位，事件置位后，再读取内存。

```
/* ************************************
* Event.c 演示使用 Event 同步线程
**************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
/* 常量定义 */
#define NUMTHREADS 3
#define BUFFER_SIZE    16
#define FOR_TIMES  5
/* 全局变量 */
HANDLE hEvent;              // 用于同步
BYTE lpSharedBuffer[16] = {0};  // 共享内存
/* 函数声明 */
void UseEvents(void);
DWORD WINAPI EventFunction(LPVOID lpParam);
/************************************
* int main(void)
************************************/
int main()
{
    UseEvents();
}
/************************************
* void UseEvents(void)
* 功能：演示 Event 的使用方法
************************************/
void UseEvents(void)
{
    HANDLE hThread;
    hEvent = CreateEvent(
        NULL,               // 默认安全属性
        TRUE,               // 手工重置
```

```
            FALSE,              // 初始为未置位的
            NULL                // 未命名
            );
        // 判断是否创建成功
        if (hEvent == NULL)
        {
            printf("CreateEvent failed (%d)\n", GetLastError());
            return;
        }
        // 创建线程
        hThread = CreateThread(NULL, 0,
            EventFunction,
            NULL,
            0, NULL);
        if(hThread == NULL)
        {
            printf("CreateThread failed (%d)\n", GetLastError());
            return;
        }
        Sleep(2000); // 可以做一些其他处理
        // 向共享内存中复制数据
        CopyMemory(lpSharedBuffer,"Event",lstrlen("Event"));
        // 设置 Event 使 ThreadFunction 线程可以开始复制数据
        SetEvent(hEvent);
}

/*************************************
* DWORD WINAPI EventFunction(LPVOID lpParam)
* 功能：线程函数，读共享内存
*************************************/
DWORD WINAPI EventFunction(LPVOID lpParam)
{
    DWORD dwWaitResult;
    // 等待，直到事件被置位
    dwWaitResult = WaitForSingleObject(
        hEvent,             // Event 句柄
        INFINITE);          // 无限等待
    if (dwWaitResult != WAIT_OBJECT_0)
    {
        printf("Wait error: %d\n", GetLastError());
        return 0;
    }
    // 读共享内存
    printf(lpSharedBuffer);
    // 重置事件
    if (! ResetEvent(hEvent) )
    {
        printf("SetEvent failed (%d)\n", GetLastError());
        return 0;
    }
    return 1;
}
```

3．实例 7-2　多个线程的互同步

实例 7-1 是一种最简单的线程同步的情况，在实际应用中，一般是两个线程需要互同步，即相互之间存在等待的关系。而且很多情况下，同步的过程也不只包括两个线程，而是多个线程。

本实例演示多个线程的互同步，主线程创建好了多个事件对象，然后创建多个线程，每个线程都有一个事件对象。之后向共享内存中写入数据，由主线程创建的若干个子线程等待主线程写操作完成后置位事件，然后各个子线程都去读共享内存中的数据。读完后，各子线程将各自的事件置位。

主线程在完成写操作并置位事件后就等待各个子线程所对应的事件是否置位，如果所有的子线程的事件都已经置位，那么说明所有子线程都完成了操作，主线程又开始向共享内存中写入数据。

```
/* ************************************
* Eventm.c 演示使用 Event 同步线程
**************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
/* 常量定义 */
#define NUMTHREADS 3
#define BUFFER SIZE    16
#define FOR TIMES 5
/* 全局变量 */
HANDLE hWriteEvent[NUMTHREADS];       // 写 Event 表示写操作是否完成
HANDLE hReadEvents[NUMTHREADS];       // 读 Event 表示读操作是否完成
BYTE lpSharedBuffer[16] = {0};   // 共享内存
/* 函数声明 */
void MultiEvents(void);
VOID WriteToBuffer(VOID);
DWORD WINAPI ThreadFunction(LPVOID lpParam);
/************************************
* int main(void)
************************************/
int main()
{
     MultiEvents();
}
/************************************
* void UseEvents(void)
* 功能：演示 Event 的使用方法
************************************/
void MultiEvents(void)
{
     HANDLE hThread;
     DWORD i;
     // 创建多个线程，读共享内存，主线程写共享内存
     // 每个线程都有对应的读写同步事件
     for(i = 0; i < NUMTHREADS; i++)
     {
          // 每个线程都有一个 Event 表示写入操作完成
          hWriteEvent[i] = CreateEvent(
               NULL,              // 默认安全属性
               FALSE,             // 自动重置
               FALSE,             // 初始为未置位的
               NULL               // 未命名
               );
          // 判断是否创建成功
          if (hWriteEvent[i] == NULL)
          {
               printf("CreateEvent failed (%d)\n", GetLastError());
               return;
          }
          // 每个读线程有一个 Event 表示读操作已经完成
          hReadEvents[i] = CreateEvent(
               NULL,     // 默认安全属性
               FALSE,     // 自动重置
               FALSE,    // 初始化为未置位的
               NULL);    // 未命名
           f(hReadEvents[i] == NULL)
           {
               printf("CreateEvent failed (%d)\n", GetLastError());
               return;
```

```
            }
            // 创建线程
            hThread = CreateThread(NULL, 0,
                 ThreadFunction,
                 i,
                 0, NULL);
            if(hThread == NULL)
            {
                 printf("CreateThread failed (%d)\n", GetLastError());
                 return;
            }
       }
       WriteToBuffer();
}

/*************************************
* VOID WriteToBuffer(INT iContent)
* 功能：由主线程调用，向共享内存中写入数据
*      等待所有读线程读完后函数返回
*************************************/
VOID WriteToBuffer(VOID)
{
     DWORD dwWaitResult, j,i;
     // 完成 FOR_TIMES 次读写
     for (j = 0; j < FOR_TIMES; j++)
     {
          Sleep(rand()%100); // 写入需要的时间间隔
          // 写入共享内存
          wsprintf(lpSharedBuffer,"shared %d",j);
          // 将线程对应的写 Event 置为“标志的”，表示写操作完成
          // 其他线程可以开始读
          for(i=0; i<NUMTHREADS; i++)
          {
              if(! SetEvent(hWriteEvent[i]) )
              {
                   printf("SetEvent failed (%d)\n", GetLastError());
                   return;
               }
          }
          // 等待所有的线程读完，开始下次写入
          dwWaitResult = WaitForMultipleObjects(
               NUMTHREADS,        // Event 句柄的个数
               hReadEvents,            // Event 句柄数组
               TRUE,         // 等到所有的 Event 都被标志
               INFINITE);         // 无限等待
          // 判断等待结果
          if (dwWaitResult != WAIT_OBJECT_0)
          {
               printf("Wait error: %d\n", GetLastError());
               ExitProcess(0);
          }
     }
}

/*************************************
* DWORD WINAPI ThreadFunction(LPVOID lpParam)
* 功能：线程函数，读共享内存
* 参数：LPVOID lpParamt 实际为指向 Event 句柄的指针
*************************************/
DWORD WINAPI ThreadFunction(LPVOID lpParam)
{
     DWORD dwWaitResult;
```

```
    BYTE lpRead[16];
    DWORD j = 0;
    DWORD dwThreadIndex  = (DWORD)lpParam;
    // 完成 FOR_TIMES 次读写
    for(; j<FOR_TIMES; j++)
    {
        // 等待写事件置位，表示数据已经写入
        dwWaitResult = WaitForSingleObject(
            hWriteEvent[dwThreadIndex], // Event 句柄
            INFINITE);                  // 无限等待
        switch (dwWaitResult)
        {
        case WAIT_OBJECT_0:
            Sleep(rand()%10);           // 模拟数据处理所需的时间间隔
            CopyMemory(lpRead,lpSharedBuffer,16);
            break;
            // 发生错误
        default:
            printf("Wait error: %d\n", GetLastError());
            ExitThread(0);
        }
        // 将读 Event 置位，表示读操作完成
        if (! SetEvent(hReadEvents[dwThreadIndex]) )
        {
            printf("SetEvent failed (%d)\n", GetLastError());
            return 0;
        }
        //打印读到的内容
        printf("线程 %u\t 第 %d 次读，内容：%s\n",
            dwThreadIndex,j,(LPSTR)lpRead);
    }
    return 1;
}
```

实例运行结果如图 7-1 所示。

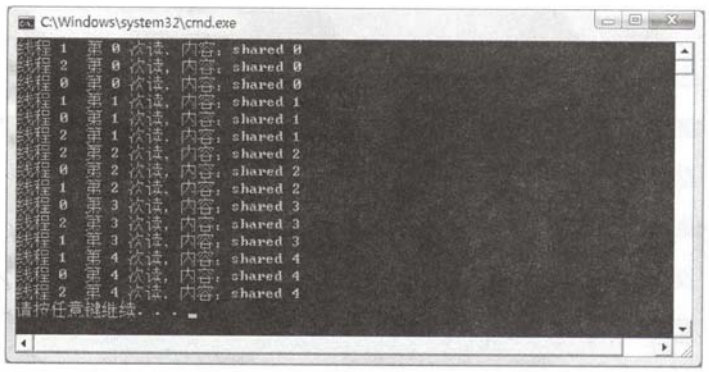

图 7-1　实例 7-2 运行结果

7.2.2　使用互斥对象(Mutex)

互斥对象具有如下的机制：如果互斥对象没有被任何线程拥有，那么它是“标记的”，如果被一个线程所拥有，那么它是“未标记的”；任何一个线程获得后，互斥对象就是“未标志的”，其他线程不可以再拥有这个互斥对象。同一时刻，一个互斥对象最多只能被一个线程拥有，从而实现“互斥”。

1．关键 API

(1) CreateMutex。

CreateMutex 的功能是创建互斥对象，函数原型如下：

```
HANDLE WINAPI CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
  BOOL bInitialOwner,
  LPCTSTR lpName
);
```

lpMutexAttributes 参数是安全属性，一般设置为 NULL; bInitialOwner 表明创建后是否被创建线程所“拥有”; lpName 是互斥对象名。

(2) OpenMutex。

通过互斥对象名获取对象句柄，函数原型如下：

```
HANDLE WINAPI OpenMutex(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName
);
```

将 lpName 参数设置为对象名，函数成功执行后，会返回对象的句柄。

(3) ReleaseMutex。

ReleaseMutex 函数的功能是释放互斥对象，一个线程释放了互斥对象后，如果其他进程在等待互斥对象置位，则等待的线程可以得到该互斥对象，等待函数返回，互斥对象被新的线程所拥有。

```
BOOL WINAPI ReleaseMutex(
  HANDLE hMutex
);
```

2．实例 7-3 使用互斥对象

本实例演示互斥对象的作用。代码中使用了条件编译，如果预定义了“MUTEX”宏，则编译使用互斥对象进行同步，如果没有预定义“MUTEX”，则不使用互斥对象同步，两种情况下，运行的结构有区别。

编译时，如果使用/D 选项，预定义 MUTEX，那么主线程创建的 4 个线程在访问 dwCounter 全局变量时是互斥的，只有一个线程对 dwCounter 进行累加，并显示了结果，然后释放了互斥对象后，其他线程才可能等到互斥对象，并执行。这样保证了 dwCounter 是一直累加的。

如果没有使用互斥对象，线程对 dwCounter 的访问是随机的。

```
/* ************************************
* Mutex.c 演示使用 Mutex 同步线程
**************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
/* 常量定义 */
#define NUM_THREADS    4
/* 全局变量 */
DWORD dwCounter = 0;
HANDLE hMutex;
/* 函数声明 */
void UseMutex(void);
DWORD WINAPI MutexThread(LPVOID lpParam);
/************************************
* int main(void)
************************************/
int main()
{
    UseMutex();
}
```

```
/*************************************
* void UseMutex(void)
* 功能：演示 Mutex 的使用方法
*************************************/
void UseMutex(void)
{
    INT i;
    HANDLE hThread;

#ifdef MUTEX
    // 创建 Mutex
    hMutex = CreateMutex(
         NULL,                    // 默认安全属性
         FALSE,                   // 初始化为未被拥有
         NULL);                   // 未命名
    if (hMutex == NULL)
    {
         printf("CreateMutex error: %d\n", GetLastError());
    }
#endif
    // 创建多个线程
    for(i = 0; i < NUM_THREADS; i++)
    {
         hThread = CreateThread(NULL, 0,
              MutexThread,
              NULL,
              0, NULL);
         if (hThread == NULL)
         {
             printf("CreateThread failed (%d)\n", GetLastError());
             return;
         }
    }
    Sleep(1000);
}

/*************************************
* DWORD WINAPI MutexThread(LPVOID lpParam)
* 功能：线程函数，读共享内存
*************************************/
DWORD WINAPI MutexThread(LPVOID lpParam)
{

#ifdef MUTEX
    DWORD dwWaitResult;
    dwWaitResult = WaitForSingleObject(
        hMutex,             // 句柄
        INFINITE);          // 无限等待

    switch (dwWaitResult)
    {
    case WAIT_OBJECT_0:
#endif
         // 等待随机长的时间，各个线程等待的时间将不一致
         Sleep(rand()%100);
         // 得到互斥对象后，访问共享数据
         printf("counter: %d\n",dwCounter);
         // 互斥对象保证同一时间只有一个线程在访问 dwCounter
         dwCounter++;

#ifdef MUTEX
         // 释放 Mutex
```

```
            if(!ReleaseMutex(hMutex))
            {
                printf("Release Mutex error: %d\n", GetLastError());
            }
            break;
        default:
            printf("Wait error: %d\n", GetLastError());
            ExitThread(0);
        }
    #endif
        return 1;
    }
```

没有预定义“MUTEX”宏，编译生成程序，程序没有同步机制，运行结果如图 7-2 所示。

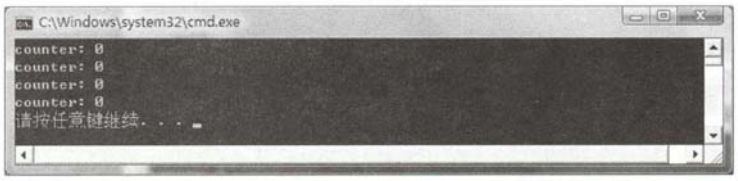

图 7-2　实例 7-3 非同步运行结果

预定义了“MUTEX”宏后，线程同步，编译生成程序运行结果如图 7-3 所示。

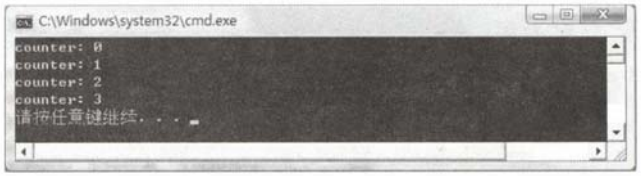

图 7-3　实例 7-3 同步运行结果

说明　本书实例工程中已经在 Vistaul Studio“配置管理器”中设置好了两种编译选项，如图 7-4 所示。

图 7-4　使用“配置管理器”自行定义多种编译选项配置情况

7.2.3　使用信号量控制访问共享数据的线程数量

信号量维护了一个计数器，计数器的值可以在 0 到指定的最大值之间。当一个线程完成了对信号量的等待后，信号量计数器值减少；当一个线程释放信号量时，信号量计数器值增加。当计数器值达到零后，信号量是“未标志的”，当计数器值大于零时，信号量是“标志的”。当计数器值降为到零时，任何线程都无法等待信号量变为“标志的”，因此信号量对限制可访问共享数据的线程数量很有用处。

1．关键 API

(1) CreateSemaphore。

CreateSemaphore 函数的功能是创建信号量对象，函数原型如下：

```
HANDLE WINAPI CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  LONG lInitialCount,
  LONG lMaximumCount,
  LPCTSTR lpName
);
```

lpSemaphoreAttributes 参数是安全属性，lInitialCount 参数是初始的计数值，lMaximumCount
参数是最大的计数值，还可以通过 lpName 参数来给信号量对象命名。

(2) OpenSemaphore。

OpenSemaphore 函数的功能是通过信号量名，获得信息量对象句柄，函数原型如下：

```
HANDLE WINAPI OpenSemaphore(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName
);
```

(3) ReleaseSemaphore。

ReleaseSemaphore 函数的功能是释放信号量，函数原型如下：

```
BOOL WINAPI ReleaseSemaphore(
  HANDLE hSemaphore,
  LONG lReleaseCount,
  LPLONG lpPreviousCount
);
```

可以一次释放多个计数，通过 lReleaseCount 参数指定。

2．实例 7-4 使用信号量对象

本示例演示如何使用信号量达到限制访问共享数据的线程数量的功能。

程序中将创建若干个线程，然后用户输入一个信号量的计数，如果创建时指定的信号量计数小于并发线程数，并不是所有的线程在同一时刻都能获得信号量。在没有信号量可用的情况下，线程的等待函数不会返回，直到有信号量可用时，线程才能继续执行。

```
/* ************************************
* Semaphorec 演示使用信号量同步线程
**************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
/* 常量定义 */
#define NUMTHREADS 4
/* 全局变量 */
HANDLE hSemaphore;
/* 函数声明 */
void UseSemaphore(void);
DWORD WINAPI SemaphoreThread(LPVOID lpParam);

/************************************
* int main(void)
* 功能：演示
************************************/
int main()
{
      UseSemaphore() ;
}
/************************************
* DWORD WaitForAllThread(HANDLE hThread[], DWORD dwNumThread)
* 功能：等待指定的线程都结束
```

```
* 参数: HANDLE hThread[], 需要等待结束的线程句柄数组
*        DWORD dwNumThread, 线程句柄数组的大小
*************************************/
DWORD WaitForAllThread(HANDLE hThread[], DWORD dwNumThread)
{
     // 等待所有线程结束
     DWORD dwWaitResult = WaitForMultipleObjects(
         dwNumThread,
         hThread,                // 线程句柄作为等待对象
         TRUE,
         INFINITE);
    switch (dwWaitResult)
         {
         case WAIT_OBJECT_0:
              printf("\nAll thread exit\n");
              break;
         default:
              printf("\nWait error: %u", GetLastError());
         }
    return 0;

}

/*************************************
* void UseSemaphore(void)
* 功能: 演示 UseSemaphore 的使用方法
*************************************/
void UseSemaphore(void)
{
     HANDLE hThread[NUMTHREADS];
     INT i;
     LONG lMax;
     CHAR cMax;
     // 打印信息获取输入
     printf("将创建%d个进程, 获得信号量的进程可以向屏幕打印。\n"
          "请输入信号量的最大计数 1~%d: ",NUMTHREADS,NUMTHREADS);
     // 获得输入的字符
     cMax = getch();
     printf("%c\n",cMax);
     // 将字符转换为数字
     lMax = cMax & 0xF;
     if(lMax<0 || lMax>NUMTHREADS)
     {
          printf("请输入 1-%d",NUMTHREADS);
    }
    // 创建信号量
    hSemaphore = CreateSemaphore(
        NULL,      // 默认安全属性
        lMax,      // 初始化计数器, 用户输入
        lMax,      // 最大计数, 用户输入
        NULL);     // 未命名
    if(hSemaphore == NULL)
    {
        printf("CreateSemaphore error: %d\n", GetLastError());
    }
    // 创建多个线程 访问共享资源
    for(i = 0; i < NUMTHREADS; i++)
    {
         hThread[i] = CreateThread(NULL, 0,
              SemaphoreThread,
              &i,           // 进程序号作为参数
              0, NULL);
         if (hThread[i] == NULL)
```

```
            {
                printf("CreateThread failed (%d)\n", GetLastError());
                return;
            }
        }
        // 等待所有线程都执行完成并退出
        WaitForAllThread(hThread,NUMTHREADS);
    }

    /*************************************
    * DWORD WINAPI SemaphoreThread(LPVOID lpParam)
    * 功能：线程函数，读共享内存
    *************************************/
    DWORD WINAPI SemaphoreThread(LPVOID lpParam)
    {
        DWORD dwWaitResult;
        BYTE lpRead[16];
        //DWORD i = *(LPWORD)lpParam;          // 线程的编号
        DWORD dwPreviousCount;

        DWORD j = 0;
        // 每个线程都将访问 3 次受限资源
        for(; j<3; j++)
        {
               // 线程可以在此进行一些操作
               // 以暂停随机长的时间模拟真实情况
               Sleep(rand()%1000);
               // 等待信号量
               dwWaitResult = WaitForSingleObject(
                  hSemaphore,        // 信号量句柄
                  INFINITE);         // 无限等待
               switch (dwWaitResult)
               {
               case WAIT_OBJECT_0:
                  printf("\nProcess %d Gets Semaphore",GetCurrentThreadId());
                  break;
               default:
                  printf("\nprocess %u Wait error: %u",GetCurrentThreadId(), GetLastError());
               }
               // 获得信息量后访问受限资源
               // 以等待随机长时间模块真实情况
               Sleep(rand()%1000);
               // 释放信号量
               if(!ReleaseSemaphore(
                  hSemaphore,                // 信号量句柄
                  1,                         // 释放后计数器减 1
                  &dwPreviousCount) )        // 获得计数
               {
                  printf("\nprocess %u ReleaseSemaphore error: %d", GetCurrentThreadId(),
GetLastError());
               }
               else
               {
                    printf("\nProcess %u release Semaphore, previous count is %u",
                        GetCurrentThreadId(), dwPreviousCount);
               }
        }

        return 1;
    }
```

运行结果如图 7-5 所示。

图 7-5　使用信号量

程序中一个有 4 个线程，去竞争计数为 3 的信号量。从输出结果可以看出最多只能有 3 个线程在同时运行，一旦有信号量释放，等待的线程就可以获得。

7.2.4　使用可等待计时器(Timer)

可等待计时器在达到指定的时间间隔后被置为“标志的”，处于等待状态中的等待函数返回。可等待计时器用于程序中需要等待特定时间间隔的情况。

一共有 3 种可等待计时器，分别为“手工重置计时器”、“同步计时器”和“周期性计时器”。其中周期性计时器最为常用，周期性计时器计时器会每间隔指定后就置位同步对象或调用一次回调函数，直到进程退出或者取消计时器。

1．关键 API

(1) CreateWaitableTimer。

创建可等待计时器，函数原型如下：

```
HANDLE WINAPI CreateWaitableTimer(
  LPSECURITY_ATTRIBUTES lpTimerAttributes,
  BOOL bManualReset,
  LPCTSTR lpTimerName
);
```

最重要的参数是 bManualReset，如果参数设置为 TRUE，那么需要手工设置，如果设置为 FALSE 则为同步计数器。手工重置计时器对象在到达时间间隔置位后，不会自动重置。

(2) SetWaitableTimer。

SetWaitableTimer 函数将设置 CreateWaitableTimer 创建的计时器的间隔等参数。

```
BOOL WINAPI SetWaitableTimer(
  HANDLE hTimer,
  const LARGE_INTEGER* pDueTime,
  LONG lPeriod,
  PTIMERAPCROUTINE pfnCompletionRoutine,
  LPVOID lpArgToCompletionRoutine,
  BOOL fResume
);
```

◇参数

p DueTime:参数指定计时器第一次置位为标记的间隔时间。

lPeriod:从第一次置位时间开始,每次置位的间隔,如果为 0 那么计时器只会置位一次。

pfnCompletionRoutine:完成例程,在计时器每次置位时调用的回调函数,可以为 NULL。pfnCompletionRoutine 完成例程的函数的原型如下:

```
VOID CALLBACK TimerAPCProc(
  LPVOID lpArgToCompletionRoutine,
  DWORD dwTimerLowValue,
  DWORD dwTimerHighValue
);
```

lpArgToCompletionRoutine:是 pfnCompletionRoutine 回调函数的参数。

fResume:表明当计时器被置位为标记的时，系统重置操作是否会被挂起。

lpName:输入参数，事件名，可以为 NULL。

说明：所谓 APC 是指“异步过程调用”。

◇返回值

函数返回是否成功。

(3) CancelWaitableTimer。

CancelWaitableTimer 的功能是取消计时器，函数原型如下：

```
BOOL WINAPI CancelWaitableTimer(
  HANDLE hTimer
);
```

2. 实例 7-5 使用计数器

本实例创建了一个周期性计数器，并且使用了回调函数。在延迟 Ss 后，计时器每 2s 置位一次，每次置位时都会调用回调函数。

回调函数在控制台上打印输出参数。

在设置了计时器后，调用 SleepEx 暂停主线程的执行。虽然为 SleepEx 函数设置的暂停执行时间长度为无限，但是，由于 SleepEx 在每次 APC 函数被调用或 I / 0 完成函数被（参见 16.4 节）调用时，都会返回，因此在本实例中，一旦计时器置位，APC 函数被调用，SleepEx 函数就会返回，最终计时器置位 4 次后，程序退出。

```
/* ************************************
* Timer.c 演示使用可等待计时器
************************************/
/* 预编译定义 *Windows 2000*  */
#define_WIN32_WINNT 0x0500
/* 头文件  */
#include <windows.h>
#include <stdio.h>
/* 常量定义  */
#define ONE_SECOND 10000000          // 秒
/* 类型定义  */
// TimerAPCProc 参数
typedef struct_APC_PROC_ARG {
    TCHAR *szText;
    DWORD dwValue;
} APC_PROC_ARG;

/************************************
* VOID CALLBACK TimerAPCProc(
*                 LPVOID lpArg,
*                 DWORD dwTimerLowValue,
*                 DWORD dwTimerHighValue )
```

```
* 功能：演示定时器的使用
************************************/
VOID CALLBACK TimerAPCProc(
                              LPVOID lpArg,
                              DWORD dwTimerLowValue,
                              DWORD dwTimerHighValue )

{
     APC_PROC_ARG *pApcData = (APC_PROC_ARG *)lpArg;
     // 显示信息
     printf( "Message: %s\nValue: %d\n\n", pApcData->szText,
         pApcData->dwValue);
     MessageBeep(MB_OK);
}

/************************************
* void main( void )
* 功能：演示定时器的使用
************************************/
void main( void )
{
     HANDLE             hTimer;
     BOOL               bSuccess;
     INT64              qwDueTime;
     LARGE_INTEGER      liDueTime;
     APC_PROC_ARG       ApcData;
     ApcData.szText = "Message to apc proc.";
     ApcData.dwValue = 1;
     hTimer = CreateWaitableTimer(
         NULL,                  // 默认安全属性
         FALSE,                 // 自动设置
         "MyTimer" );           // 命名
    if( !hTimer )
    {
         printf("CreateWaitableTimer failed with Error %d.",
             GetLastError() );
         return;
    }
    else
    {
          try
         {
             // 5s
             qwDueTime = -5 * ONE_SECOND;
             // 转换为 LARGE_INTEGER 数据结构
             liDueTime.LowPart  = (DWORD) ( qwDueTime & 0xFFFFFFFF );
             liDueTime.HighPart = (LONG)  ( qwDueTime >> 32 );
             bSuccess = SetWaitableTimer(
                 hTimer,            // 计时器句柄
                 &liDueTime,        // 延迟时间，第一次置位的时间
                 2000,              // 计时器间隔，每 2s 置位一次
                 TimerAPCProc,      // 置位时触发调用的例程
                 &ApcData,          // 例程参数
                 FALSE );           // 不重置挂起系统
             // 判断是否创建成功
             if ( bSuccess )
             {
                 for ( ; ApcData.dwValue < 10; ApcData.dwValue ++ )
                 {
                     // 第二个参数当设置为 TRUE，表示当计时器到达，APC 被调用时返回
                     SleepEx(
                          INFINITE,     // 无限等待
                          TRUE );
                     // 每返回一次，dwValue 值递增
```

```
                }
            }
            else
            {
                printf( "SetWaitableTimer failed with Error %d.",
                    GetLastError() );
            }
        }
        _finally
        {
            CloseHandle( hTimer );
        }
    }
}
```

运行结果如图 7-6 所示。

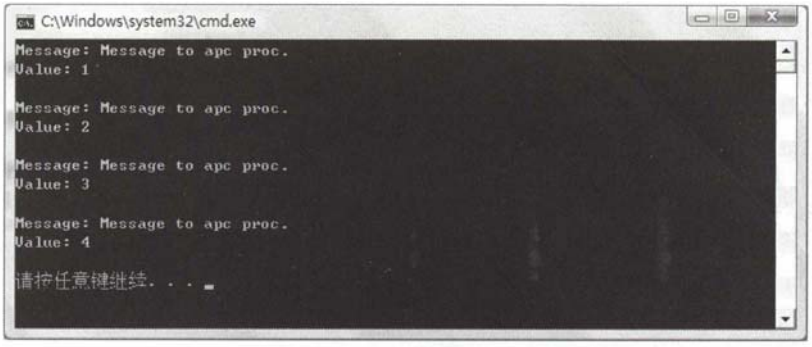

图 7-6　周期计时器运行结果

7.3　等待进程和线程的执行完成

在 7.1.1 小节中提到，等待函数除了可以等待同步对象外，还可以等待一些其他对象，包括进程和线程等。使用等待函数等待进程和线程的执行完成实现很简单，以进程或线程句柄为等待对象，当进程和线程执行完成后，对象状态改变为“标记的”。

第 8 章　服务

服务是一种特殊的应用程序，符合特殊接口形式要求，操作系统的很多的重要功能依赖于服务。

本节将介绍与服务相关的若干概念、系统对服务的管理方式、服务的属性等，然后通过实例演示如何编写服务应用程序，如何安装、删除服务，以及如果向服务应用程序发送请求、接收返
回结果等。

8.1　基本概念

服务是一种在系统中常驻的程序，服务可以在系统启动时自启动（先于用户登录）。当一个应用程序需要常驻在系统，或者随时为其他应用程序提供服务时，可以使用服务应用程序。一般编写网络服务端程序时需要使用服务。比如 Windows 系统中常用的网络应用服务器(WEB、FTP、邮件等)ns 就是一个服务应用程序。Windows 系统中具有一个服务控制器(SCM)用于控制服务。服务的安装、删除、启动、停止、控制与 I/O 都是通过服务控制器。

Windows 系统的服务分为服务应用程序和内核驱动服务程序，服务应用程序运行于系统用户态，内核驱动服务运行于系统内核态。用户态服务程序需遵守 SCM 的接口规范。

在系统中可以通过如图 8-1 所示“控制面板-÷管理工具一服务”界面对系统中的服务进行管理，也可使用命令行程序 sc.exe 管理服务。系统为服务程序的编写和服务的管理提供了 API，也可以通过相关 API 编写服务程序和服务控制程序等。

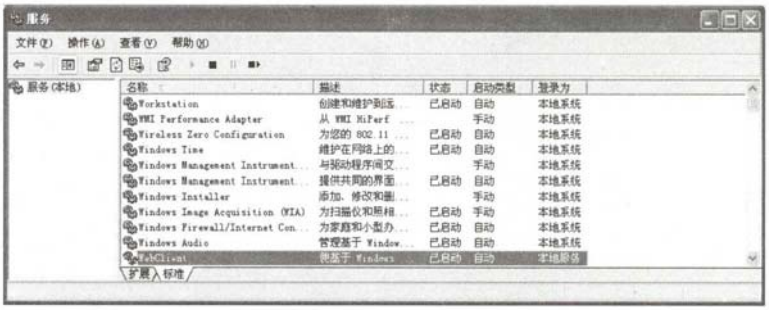

图 8-1　通过服务管理器查看系统中已经安装的服务

8.1.1　服务控制器(SCM)

服务控制器对系统中所有服务进行管理，SCM 管理着系统中已经安装的服务程序和设备驱动程序数据库，数据库中保存的信息包括系统安装了哪些服务，每个服务包括如何启动每个服务、各个服务的安全属性以及控制接口等。

服务程序、服务配置程序和服务控制程序的设计都需要使用 SCM 提供的函数。

SCM 除了管理系统服务外，还管理内核设备驱动程序，服务配置程序和服务控制程序既可以管理
用户态的服务控制程序，也可以管理设备驱动程序。在第 16 章介绍有关设备驱动程序的内容。

8.1.2 服务程序

服务程序是运行服务所需要的可执行程序，SCM 通过运行和终止服务程序来启动和停止服务。

8.1.3 服务控制管理程序

启动、终止或控制服务程序的运行是通过向 SCM 发送请求实现的。对服务程序进行启动、停止、修改属性等操作的程序就是服务控制管理程序。可以编程实现获取和修改服务控制器所维护的数据库信息，用于安装和删除服务、修改服务配置属性等。

图 8-1 所示的系统服务管理界面就是一个典型的服务控制配置程序。

8.1.4 系统服务管理工具

在了解有关于服务程序的开发前，先了解系统管理服务的工具。

通过“控制面板_管理工具一服务”，可以查看和管理当前系统中安装的服务。双击可以查看服务的属性，可以设置启动类型及控制服务（启动、停止等）。

此外系统还提供了功能更强大的命令行工具 SC.EXE。在命令行中启动！

```
sc.exe /?
```

可以查看服务管理器的帮助信息。

sc.exe 可以实现对服务的所有控制，包括安装、删除、配置、启动、停止等。

8.1.5 服务的属性

一个服务具有若干个属性，包括服务名称、显示名称、服务类型、描述、可执行文件路径、启动类型、服务状态、启动参数、依存关系等，如图 8-2 和图 8-3 所示。

每个服务属性的意义如下。

◇服务名称：唯一标识一个服务。

◇显示名称：在系统服务管理界面和 sc.exe 中显示的名称。

◇ 服务类型：包括 SERVICE—FILE_SYSTEM_DRIVER、SERVICE_KERNEL_ DRIVER、SERVICE_WIN32_OWN_PROCESS、SERVICE_WIN32_SHARE_PROCESS，共 4 种，分别是文件系统驱动服务、驱动服务、独立进程服务和共享进程服务。前两种服务运行于系统内核态，后两种服务运行于系统用户态，独立进程服务以独立的进程运行，共享进程服务和其他服务共享一个进程。

◇描述：一段说明性的文本。

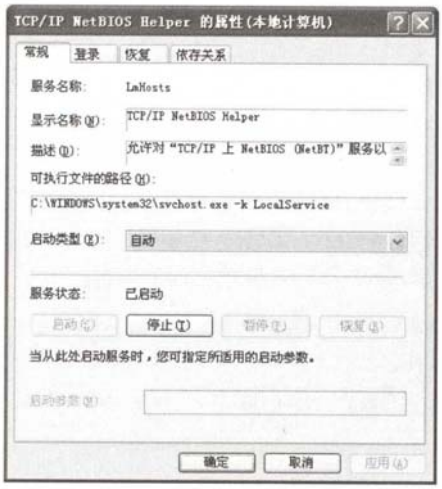

图 8-2　查看服务的属性

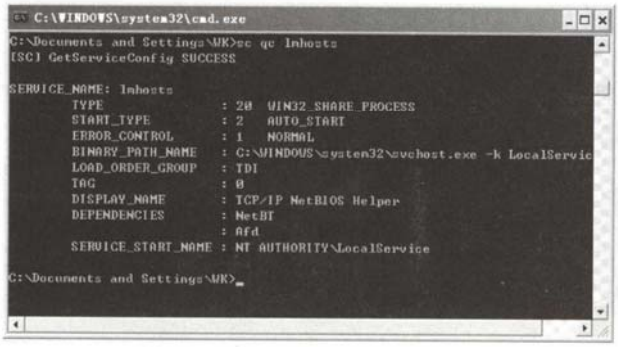

图 8-3　通过 sc.exe 控制服务

◇可执行程序及路径：启动服务的服务程序的可执行程序及程序启动参数。

◇启动类型：表示在什么情况下启动服务，包括自动、手动、已禁用。

◇服务状态：服务运行的状态。

◇启动参数：服务入口函数得到的参数。

这些属性将在服务有关的程序开发中讲解。

8.2　编写服务程序

本节介绍有关编写服务程序的相关内容。一个服务程序至少包括入口函数、服务主函数和控制处理函数。

8.2.1　入口函数

入口函数是可执行程序执行的起点，服务程序的入口函数与一般的可执行程序的入口函数没有区别。

8.2.2　服务主函数

任何一个应用程序都需要一个入口函数，一个服务程序必须具有服务主函数，服务主函数是服务启动时执行的入口，也是服务的主线程执行起点。

1．服务主函数 ServiceMain

服务主函数一般称作 ServiceMain 函数。但是服务主函数的名称与线程函数 ThreadProc 一样，其函数名并没有特殊要求，只是其参数接口和调用类型了必须与要求一致。ServiceMain 函数的原型如下：

```
VOID WINAPI ServiceMain(
  DWORD dwArgc,
  LPTSTR* lpszArgv
);
```

服务主函数的参数与 main 函数的参数使用方法类似，但是服务主函数的参数不是通过在命

令行启动时设定的，而是通过 SCM 的相关 API 进行传递的（StartService 函数）。

2．向 SCM 注册服务的主函数 StartServiceCtrIDispatcher

SCM 要对服务进行管理，就必须知道服务程序的服务主函数。服务程序通过调用 StartSeⅣiceCtrlDispatcher API 函数设置服务主函数，同时通知 SCM。

StartServiceCtrlDispatcher 函数原型如下：

```
BOOL StartServiceCtrlDispatcher(
  const LPSERVICE_TABLE_ENTRY lpServiceTable
);
```

结构 SERVICE_TABLE_ENIRY 的原型如下：

```
typedef struct _SERVICE_TABLE_ENTRY {
  LPTSTR lpServiceName;
  LPSERVICE_MAIN_FUNCTION lpServiceProc;
} SERVICE_TABLE_ENTRY, *LPSERVICE_TABLE_ENTRY;
```

其中 lpServiceName 为服务名称，lpServiceProc 为指向 ServiceMain 的函数指针。只要将函数的指针赋值给 lpServiceProc，再调用 StartServiceCtrlDispatcher，这个函数就成为了服务主函数。

8.2.3 控制处理函数

1．控制处理函数 Handler

控制处理函数用于处理 SCM 向服务传递的服务控制请求。控制处理函数 Handler 原型如下：

```
VOID WINAPI Handler(
  DWORD fdwControl
);
```

与 ServiceMain 函数一致，基函数名没有特殊要求。

2．注册控制管理函数

API 函数 RegisterServiceCtrlHandler 用于向 SCM 设置一个服务的控制处理函数。

```
SERVICE_STATUS_HANDLE RegisterServiceCtrlHandler(
  LPCTSTR lpServiceName,
  LPHANDLER_FUNCTION lpHandlerProc
);
```

其中 lpServiceName 为服务名称，lpHandlerProc 为 Handler 函数指针。

3．实例 8-1 服务程序

本实例是一个服务程序的代码，演示如何编写服务程序，包括服务主函数的编写、控制请求处理函数。及如何调用 API 向 SCM 注册服务主函数和服务控制处理函数。

代码编译完成后，生成 SplSrv.exe。

```
/* ************************************
* Service.c 编译生成 SplSrv.exe 作为服务启动
**************************************/
/* 头文件 */
#include <windows.h>
/* 全局变量 */
SERVICE_STATUS          SplSrvServiceStatus;
SERVICE_STATUS_HANDLE   SplSrvServiceStatusHandle;
/* 函数声明 */
VOID SvcDebugOut(LPSTR String, DWORD Status);
```

```
VOID WINAPI SplSrvServiceCtrlHandler (DWORD opcode);
VOID WINAPI SplSrvServiceStart (DWORD argc, LPTSTR *argv);
DWORD SplSrvServiceInitialization (DWORD argc, LPTSTR *argv,  DWORD *specificError);
/*************************************
* VOID WINAPI SplSrvServiceStart (DWORD argc, LPTSTR *argv)
* 功能:服务启动函数
*************************************/
VOID WINAPI SplSrvServiceStart (DWORD argc, LPTSTR *argv)
{
     DWORD status;
     DWORD specificError;
     // 填充 SERVICE_STATUS 结构
     SplSrvServiceStatus.dwServiceType       = SERVICE_WIN32;
     SplSrvServiceStatus.dwCurrentState
          = SERVICE_START_PENDING;        // 服务在运行
     SplSrvServiceStatus.dwControlsAccepted
          = SERVICE_ACCEPT_STOP | SERVICE_ACCEPT_PAUSE_CONTINUE;
     SplSrvServiceStatus.dwWin32ExitCode     = 0;
     SplSrvServiceStatus.dwServiceSpecificExitCode = 0;
     SplSrvServiceStatus.dwCheckPoint        = 0;
     SplSrvServiceStatus.dwWaitHint          = 0;
     // 注册服务控制请求处理例程
     SplSrvServiceStatusHandle = RegisterServiceCtrlHandler(
          "Sample_Srv",// 服务名，在创建服务时使用了
          // SERVICE_WIN32_OWN_PROCESS，因此本参数被忽略
          SplSrvServiceCtrlHandler); // 控制请求处理例程，函数名

     if (SplSrvServiceStatusHandle == (SERVICE_STATUS_HANDLE)0)
     {
          SvcDebugOut(" [SPLSRV_SERVICE] RegisterServiceCtrlHandler "
              "failed %d\n", GetLastError());
          return;
     }
     // 初始化工作，本示例未使用，函数为空
     status = SplSrvServiceInitialization(argc,argv, &specificError);
     // 初始化出错，用户自行修改
     if (status != NO_ERROR)
     {
          SplSrvServiceStatus.dwCurrentState      = SERVICE_STOPPED;
          SplSrvServiceStatus.dwCheckPoint        = 0;
          SplSrvServiceStatus.dwWaitHint          = 0;
          SplSrvServiceStatus.dwWin32ExitCode     = status;
          SplSrvServiceStatus.dwServiceSpecificExitCode = specificError;
          SetServiceStatus (SplSrvServiceStatusHandle, &SplSrvServiceStatus);
          return;
     }
     // 初始化完成，设置运行状态
     SplSrvServiceStatus.dwCurrentState      = SERVICE_RUNNING;
     SplSrvServiceStatus.dwCheckPoint        = 0;
     SplSrvServiceStatus.dwWaitHint          = 0;
     if (!SetServiceStatus (SplSrvServiceStatusHandle, &SplSrvServiceStatus))
     {
          status = GetLastError();
          SvcDebugOut(" [SPLSRV_SERVICE] SetServiceStatus error %ld\n",status);
     }
     // 用户自行修改，用于完成服务的工作
     SvcDebugOut(" [SPLSRV_SERVICE] Returning the Main Thread \n",0);
     return;
}

/*************************************
* DWORD SplSrvServiceInitialization(DWORD   argc,
*    LPTSTR  *argv,
```

```
*    DWORD *specificError)
* 功能：初始化，这里未进行任何工作，留待读者修改
*************************************/
DWORD SplSrvServiceInitialization(DWORD   argc,
                                  LPTSTR  *argv,
                                  DWORD *specificError)
{
   return(0);
}

/*************************************
* VOID WINAPI SplSrvServiceCtrlHandler (DWORD Opcode)
* 功能：服务控制请求的处理函数，与 ControlService 函数配合
* 参数：服务控制码
*************************************/
VOID WINAPI SplSrvServiceCtrlHandler (DWORD Opcode)
{
     DWORD status;
     switch(Opcode)
     {
     case SERVICE_CONTROL_PAUSE:
          // 完成相关功能
          SplSrvServiceStatus.dwCurrentState = SERVICE_PAUSED;
          break;
     case SERVICE_CONTROL_CONTINUE:
          // 完成相关功能
          SplSrvServiceStatus.dwCurrentState = SERVICE_RUNNING;
          break;
     case SERVICE_CONTROL_STOP:
          // 完成相关功能
          SplSrvServiceStatus.dwWin32ExitCode = 0;
          SplSrvServiceStatus.dwCurrentState  = SERVICE_STOPPED;
          SplSrvServiceStatus.dwCheckPoint    = 0;
          SplSrvServiceStatus.dwWaitHint      = 0;

          if (!SetServiceStatus (SplSrvServiceStatusHandle,
              &SplSrvServiceStatus))
          {
              status = GetLastError();
              SvcDebugOut(" [SPLSRV_SERVICE] SetServiceStatus error %ld\n", status);
          }
          SvcDebugOut(" [SPLSRV_SERVICE] Leaving SplSrvService \n",0);
          return;
     case SERVICE_CONTROL_INTERROGATE:
          // 收到此请求后发出声响，演示服务控制请求的处理过程，读者可自行修改
          MessageBeep(MB_OK);
          break;
     default:
          SvcDebugOut(" [SPLSRV_SERVICE] Unrecognized opcode %ld\n", Opcode);
     }
     // 当前状态
     if (!SetServiceStatus (SplSrvServiceStatusHandle,  &SplSrvServiceStatus))
     {
          status = GetLastError();
          SvcDebugOut(" [SPLSRV_SERVICE] SetServiceStatus error %ld\n", status);
     }
     return;
}
/*************************************
* void main( )
```

```
* 功能：程序入口函数，注册服务启动函数等
*************************************/
void main( )
{
     // 设置 SERVICE_TABLE_ENTRY 数据结构，以 NULL 结构结束
     // 作为 StartServiceCtrlDispatcher 函数的参数
     SERVICE_TABLE_ENTRY   DispatchTable[] =
     {
        { "Sample_Srv", (LPSERVICE_MAIN_FUNCTION) SplSrvServiceStart },
        { NULL, NULL }
     };
     if (!StartServiceCtrlDispatcher( DispatchTable))
     {
           SvcDebugOut(" [SPLSRV_SERVICE] StartServiceCtrlDispatcher (%d)\n",
                GetLastError());
     }
}
/************************************
* VOID SvcDebugOut(LPSTR String, DWORD Status)
* 功能：显示信息给调试器
* 参数：LPSTR String     消息字符串
*          DWORD Status     状态
************************************/
VOID SvcDebugOut(LPSTR String, DWORD Status)
{
    CHAR  Buffer[1024];
    if (strlen(String) < 1000)
    {
          wsprintf(Buffer, String, Status);
          OutputDebugString(Buffer);
     }
}
```

其中函数 SplSrvServiceStart 是服务主函数，SplSrvServiceCtrlHandler 是控制处理函数。实例在 mam 函数中调用了 StartServiceCtrlDispatcher 函数，设置服务主函数。在 SplSrvServiceStart 函数中调用 RegisterServiceCtrlHandler 注册控制处理函数。

服务主函数除了进行服务控制函数的注册外，还进行了服务的初始化。

SplSrvServiceCtrlHandler 是服务控制处理函数，处理包括 SERVICE_CONTROL_PAUSE、SERVICE_CONTROL_CONTINUE、SERVICE_CONTROL_STOP 在内的几个服务控制码。服务控制码是由 API 函数 ControlService 通过 SCM 向服务发送的。

8.3 实现对服务的控制和管理

8.2 节所示源代码即为一个简单的服务程序。服务程序编写完成后，还需要服务管理程序才可以运行。一种方法是使用系统的服务管理工具，如 sc.exe 等；另一种方法是使用服务管理 API 自行编写一个服务管理程序。服务管理程序为系统新建服务，负责启动停止服务程序，并负责管理服务程序的属性等。

8.3.1 创建、删除服务

服务程序编写完成后，还不能运行服务。必须首先向系统注册服务。创建和删除服务可以通过 sc.exe 来操作。系统提供了相关 API 实现类似的功能。本小节介绍如何编程创建和删除服务。

创建了服务后，可以通过“控制面板→管理工具→服务”查看创建的服务，如果要删除服务后服务，从“控制面板_管理工具一服务”中清除。

1.关键 API

(1) OpenSCManager。

获取当前系统中所有进程的 PID。函数原型如下：

```
SC_HANDLE OpenSCManager(
  LPCTSTR lpMachineName,
  LPCTSTR lpDatabaseName,
  DWORD dwDesiredAccess
);
```

◇参数

lpMachineName:输入参数，机器名，如果是本机则使用 NULL。

lpDatabaseName：输入参数，SCM 数据库名，设置为 NULL 或设置为 SERVICES ACTIVE DATABASE，效果等同。

dwDesiredAccess：输入参数，需要的权限，可设置为 SC MANAGER ALL ACCESS、SC_MANAGER_CREATE_SERVICE、SC_MANAGERCONNECT 等。

◇返回值

SC_HANDLE 类型，SCM 的句柄。

(2) CreateService。

向系统创建服务，创建时指定服务的属性，函数原型如下：

```
SC_HANDLE CreateService(
  SC_HANDLE hSCManager,
  LPCTSTR lpServiceName,
  LPCTSTR lpDisplayName,
  DWORD dwDesiredAccess,
  DWORD dwServiceType,
  DWORD dwStartType,
  DWORD dwErrorControl,
  LPCTSTR lpBinaryPathName,
  LPCTSTR lpLoadOrderGroup,
  LPDWORD lpdwTagId,
  LPCTSTR lpDependencies,
  LPCTSTR lpServiceStartName,
  LPCTSTR lpPassword
);
```

CreateService 函数的参数很多，但是设置方法都很简单，直接从参数名就可大体知道参数的意义。

◇返回值

返回 SC HANDLE 类型，服务的句柄。

(3) OpenService。

打开服务，获取服务句柄，函数原型如下：

```
SC_HANDLE WINAPI OpenService(
  SC_HANDLE hSCManager,
  LPCTSTR lpServiceName,
  DWORD dwDesiredAccess
);
```

(4) DeleteService。

删除服务，以服务句柄为参数，函数原型如下：

```
BOOL WINAPI DeleteService(
  SC_HANDLE hService
);
```

2．实例 8-2 创建服务

本实例使用 CreateService 函数创建服务。参数 schSCManager 是由 OpenSCManager API 函数获得的 SCM 句柄，OpenSCManager 函数的调用过程在实例 8-7 演示。

```
/*************************************
* BOOL CreateSampleService(LPTSTR szPath, LPSTR szServiceName)
* 功能: 创建服务
* 参数
*          SC_HANDLE schSCManager, SCM 句柄
*          LPTSTR szPath, 服务程序的路径
*          LPSTR szServiceName, 服务名
*************************************/
BOOL CreateSampleService(
                                SC_HANDLE schSCManager,
                                LPTSTR szPath,
                                LPSTR szServiceName)
{
    // 创建服务
    schService = CreateService(
         schSCManager,                  // SCM 句柄
         szServiceName,                 // 服务名
         "Service sample",              // 显示的服务名
         SERVICE_ALL_ACCESS,            // 存取权限
         SERVICE_WIN32_OWN_PROCESS,     // 服务类别
         SERVICE_DEMAND_START,          // 启动类别
         SERVICE_ERROR_NORMAL,          // 错误控制类别
         szPath,                        // 服务的可执行文件路径
         NULL,                          // 不属于任何用户组
         NULL,                          // 使用已存在的标签
         NULL,                          // 独立服务
         NULL,                          // 本地系统账户
         NULL);                         // 密码为空

    if (schService == NULL)
    {
         printf("CreateService failed (%d)\n", GetLastError());
         return FALSE;
    }
    else
    {
         printf("CreateService succeeded\n");
         CloseServiceHandle(schService);
         return TRUE;
    }
}
```

3．实例 8-3 删除服务

本实例演示如何删除服务。首先调用 OpenService 函数打开服务，获得服务句柄，然后调用 API 函数 DeleteService 删除服务。

```
/*************************************
* BOOL DeleteSampleService(LPTSTR szNameOfService)
* 功能: 删除服务
*
* 参数: LPTSTR szNameOfService    为服务的名字
*************************************/
BOOL DeleteSampleService(LPTSTR szNameOfService)
{
    schService = OpenService(
         schSCManager,              // SCM 句柄
         szNameOfService,           // 服务名
```

```
        DELETE);                    // 可删除

    if (schService == NULL)
    {
        printf("OpenService failed (%d)\n", GetLastError());
        return FALSE;
    }
    // 删除服务
    if (!DeleteService(schService) )
    {
        printf("DeleteService failed (%d)\n", GetLastError());
        return FALSE;
    }
    else
        printf("DeleteService succeeded\n");
    // 关闭句柄
    CloseServiceHandle(schService);
    return TRUE;
}
```

8.3.2 启动、停止服务，向服务发送控制请求

在创建服务后，可以使用“控制面板→管理工具→服务”来启动和停止服务，也可以使用 sc.exe 来实现。

系统同样提供了相关的 API 来实现启动、停止服务，向服务发送控制请求等操作，本小节演示如何通过 API 函数来实现可服务的启动停止控制及请求发送。

服务的控制请求是服务控制程序控制服务的主要方法。在服务管理程序向服务程序发送了控制码后，服务所注册的控制处理程序，如实例 8-1 中的 SplSrvServiceCtrlHandler 函数就会被调用，并将对应的控制码发送给控制处理程序。由控制处理程序来决定如何作出响应。

1. 关键 API

(1) StartService。

启动服务，并设置服务主函数的参数。

```
BOOL StartService(
  SC_HANDLE hService,
  DWORD dwNumServiceArgs,
  LPCTSTR* lpServiceArgVectors
);
```

参数 hService 为服务句柄，后两个参数为服务主函数的参数，可以设置为 NULL，返回值表示是否成功。

(2) ControlService。

向服务发送控制请求码。并获取服务状态。

```
BOOL ControlService(
  SC_HANDLE hService,
  DWORD dwControl,
  LPSERVICE_STATUS lpServiceStatus
);
```

参数 hService 为服务句柄，dwControl 为控制码，lpServiceStatus 为返回的服务状态。如果需要停止服务，需要使用 ControlService 向服务发送 SERVICE_CONTROL_STOP 控制码。

2.实例 8-4 启动服务

本实例演示使用 StartService 启动服务。代码中使用 QueryServiceStatusEx API 函数，获取服务的当前状态。

```
/************************************
* BOOL StartSampleService(SC_HANDLE schSCManager,LPTSTR szServiceName)
* 功能: 启动服务
* 参数: SC_HANDLE schSCManager    , SCM 句柄
*       LPTSTR szServiceName, 服务名
************************************/
BOOL StartSampleService(SC_HANDLE schSCManager,LPTSTR szServiceName)
{
    SC_HANDLE schService;
    SERVICE_STATUS_PROCESS ssStatus;
    DWORD dwOldCheckPoint;
    DWORD dwStartTickCount;
    DWORD dwWaitTime;
    DWORD dwBytesNeeded;
    // 打开服务
    schService = OpenService(
        schSCManager,          // SCM 句柄
        szServiceName,         // 服务名
        SERVICE_ALL_ACCESS);
    if (schService == NULL)
    {
        return 0;
    }
    // 启动服务
    if (!StartService(
          schService,     // 服务的句柄
          0,              // 参数个数为零
          NULL) )         // 参数指针为 NULL, 没有参数
    {
          printf("Service start error (%u).\n",GetLastError());
          return 0;
    }
    else
    {
          printf("Service start pending.\n");
    }

    // 验证状态
    if (!QueryServiceStatusEx(
          schService,                                  // 服务的句柄
          SC_STATUS_PROCESS_INFO,                      // 服务状态信息
          &ssStatus,                                   // 结构体变量地址
          sizeof(SERVICE_STATUS_PROCESS),              // 结构体大小
          &dwBytesNeeded ) )                           // 有存储状态信息所用的字节数
    {
          return 0;
    }

    // tick count & checkpoint.
    dwStartTickCount = GetTickCount();
    dwOldCheckPoint = ssStatus.dwCheckPoint;
    // 查询状态, 确定 PENDING 状态结束
    while (ssStatus.dwCurrentState == SERVICE_START_PENDING)
    {
          // 等待一段时间
          dwWaitTime = ssStatus.dwWaitHint / 10;
          if( dwWaitTime < 1000 )
              dwWaitTime = 1000;
          else if ( dwWaitTime > 10000 )
              dwWaitTime = 10000;
          Sleep( dwWaitTime );
          // 再次查询
          if (!QueryServiceStatusEx(
                schService,                          // 服务的句柄
                SC_STATUS_PROCESS_INFO,              // 服务状态信息
                &ssStatus,                               // 结构体变量地址
```

```
                sizeof(SERVICE_STATUS_PROCESS),          // 结构体大小
                &dwBytesNeeded ) )                  // 存储状态信息所用的字节数
                break;
            if ( ssStatus.dwCheckPoint > dwOldCheckPoint )
            {
                // 进程创建中
                dwStartTickCount = GetTickCount();
                dwOldCheckPoint = ssStatus.dwCheckPoint;
            }
            else
            {
                if(GetTickCount()-dwStartTickCount > ssStatus.dwWaitHint)
                {
                    // WaitHint 时间到
                    break;
                }
            }
        }
        // 关闭句柄
        CloseServiceHandle(schService);
        // 判断是否创建成功(状态由 PENDING 变为 RUNNING)
        if (ssStatus.dwCurrentState == SERVICE_RUNNING)
        {
            printf("StartService SUCCESS.\n");
            return 1;
        }
        else
        {
            printf("\nService not started. \n");
            printf("  Current State: %d\n", ssStatus.dwCurrentState);
            printf("  Exit Code: %d\n", ssStatus.dwWin32ExitCode);
            printf("  Service Specific Exit Code: %d\n",
                ssStatus.dwServiceSpecificExitCode);
            printf("  Check Point: %d\n", ssStatus.dwCheckPoint);
            printf("  Wait Hint: %d\n", ssStatus.dwWaitHint);
            return 0;
        }
    }
```

本实例运行的效果参见实例 8-7。

3．实例 8-5 向服务发送控制请求

ControlSampleService 函数实现了对服务的控制。先根据用户输入的控制码，设置存取权限，然后调用 OpenService API 打开服务，之后调用 ControlService 服务向服务发送控制码。

```
/************************************
* BOOL ControlSampleService(DWORD fdwControl)
* 功能: 向服务发送控制码
* 参数: DWORD fdwControl     控制码值
*         SCM     句柄, 服务名直接使用全局变量
************************************/
BOOL ControlSampleService(DWORD fdwControl)
{
    SERVICE_STATUS ssStatus;
    DWORD fdwAccess;
    DWORD dwStartTickCount, dwWaitTime;
    // Access
    switch (fdwControl)
    {
    case SERVICE_CONTROL_STOP:
        fdwAccess = SERVICE_STOP;
        break;
    case SERVICE_CONTROL_PAUSE:
    case SERVICE_CONTROL_CONTINUE:
        fdwAccess = SERVICE_PAUSE_CONTINUE;
```

```
        break;
    case SERVICE_CONTROL_INTERROGATE:
        fdwAccess = SERVICE_INTERROGATE;
        break;
    default:
        fdwAccess = SERVICE_INTERROGATE;
    }
    // 打开服务
    schService = OpenService(
        schSCManager,              // SCManager 句柄
        szServiceName,             // 服务名
        fdwAccess);                // 存取权限
    if (schService == NULL)
    {
        printf("OpenService failed (%d)\n", GetLastError());
        return FALSE;
    }
    // 发送控制码
    if (!ControlService(
        schService,   // 服务的句柄
        fdwControl,   // 控制码
        &ssStatus) )  // 状态
    {
        printf("ControlService failed (%d)\n", GetLastError());
        return FALSE;
    }
    // 显示状态
    printf("\nStatus of Sample_Srv: \n");
    printf("  Service Type: 0x%x\n", ssStatus.dwServiceType);
    printf("  Current State: 0x%x\n", ssStatus.dwCurrentState);
    printf("  Controls Accepted: 0x%x\n",
        ssStatus.dwControlsAccepted);
    printf("  Exit Code: %d\n", ssStatus.dwWin32ExitCode);
    printf("  Service Specific Exit Code: %d\n",
        ssStatus.dwServiceSpecificExitCode);
    printf("  Check Point: %d\n", ssStatus.dwCheckPoint);
    printf("  Wait Hint: %d\n", ssStatus.dwWaitHint);

    return TRUE;
}
```

在实例 8-6 中 StopService 也使用服务控制的方式来停止服务。

```
ControlService( hService, SERVICE_CONTROL_STOP, &ss )
```

停止服务的完整过程参见示例 8-6。

8.3.3 管理服务状态、配置服务、服务的依赖关系

1．管理服务状态

服务状态信息是了解服务的运行情况的关键。一是服务管理程序在对服务进行操作时需要判断服
务的状态，二是服务程序本身也需要根据服务状态来决定相关操作。管理服务状态涉及 QueryServiceStatusEx 和 SetServiceStatus 等 API 函数， 以及 SERVICE_STATUS_PROCESS 和 SERVICE_STAIUS 结构。

(1) QueryServiceStatUS。

获取服务运行状态，函数原型如下：

```
BOOL QueryServiceStatus(
  SC_HANDLE hService,
  LPSERVICE_STATUS lpServiceStatus
);
```

参数 hService 是服务句柄，lpServiceStatus 指向保存服务状态信息的 SERVICE_STATUS 结构变量，返回值表示是否成功。

(2) SetServiceStatus。

设置服务运行状态，函数原型如下：

```
BOOL SetServiceStatus(
  SERVICE_STATUS_HANDLE hServiceStatus,
  LPSERVICE_STATUS lpServiceStatus
);
```

参数与返回值意义与 QueryServiceStatusEx 类似。

(3) SERVICE_STATUS 结构。

```
typedef struct _SERVICE_STATUS {
  DWORD dwServiceType;
  DWORD dwCurrentState;
  DWORD dwControlsAccepted;
  DWORD dwWin32ExitCode;
  DWORD dwServiceSpecificExitCode;
  DWORD dwCheckPoint;
  DWORD dwWaitHint;
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

结构的 dwCurrentState 表示服务状态，其值可能是 SERVICE_CONTINUE PENDING、SERVICE PAUSE_PENDING、SERVICE PAUSED、SERVICE RUNNING、SERVICE_START _PEND ING、SERVICE_STOP PENDING、SERVICE_STOPPED，使用方法见实例 8-6。

(4) QueryServiceStatusEX。

与 QueryServiceStatus 不同，除了可以获得服务信息外，还可以获得服务进程的有关信息，如 PID 等。

```
BOOL QueryServiceStatusEx(
  SC_HANDLE hService,
  SC_STATUS_TYPE InfoLevel,
  LPBYTE lpBuffer,
  DWORD cbBufSize,
  LPDWORD pcbBytesNeeded
);
```

如果参数 InfoLevel 设置为 SC_STATUS_PROCESS_INFO 常量，则 lpBuffer 应为指向 SERVICE_STATUS_PROCESS 结构变量的指针，cbBufSize 为 lpBuffer 指向的缓冲区的大小，pcbBytesNeeded 返回需要的缓冲区大小。

(5) SERVICE_STATUS_PROCESS 。

```
typedef struct _SERVICE_STATUS_PROCESS {
  DWORD dwServiceType;
  DWORD dwCurrentState;
  DWORD dwControlsAccepted;
  DWORD dwWin32ExitCode;
  DWORD dwServiceSpecificExitCode;
  DWORD dwCheckPoint;
  DWORD dwWaitHint;
  DWORD dwProcessId;
  DWORD dwServiceFlags;
} SERVICE_STATUS_PROCESS, *LPSERVICE_STATUS_PROCESS;
```

2. 实例 8-6 停止服务运行

本实例演示服务的管理，实现程序是一个完整的停止服务运行的过程，由于停止过程中需要了解服务运行的状态，所以本实例还演示了与服务有依赖关系的相关的内容。

```
/*************************************
* StopService
* 功能：停止服务
* 参数：SC_HANDLE hSCM                SCM 句柄
```

```
*          LPTSTR szServiceName      服务名
*          BOOL fStopDependencies    是否结束依赖的服务
*          DWORD dwTimeout           超时
**********************************/
DWORD StopService(SC_HANDLE hSCM,
                  LPTSTR szServiceName,
                  BOOL fStopDependencies,
                  DWORD dwTimeout )
{
    SERVICE_STATUS_PROCESS ssp;
    SERVICE_STATUS ss;
    DWORD dwStartTime = GetTickCount();
    DWORD dwBytesNeeded;
    // 打开服务
    SC_HANDLE hService = OpenService(
        hSCM,              // SCM 句柄
        szServiceName,     // 服务名
        SERVICE_ALL_ACCESS);
    // 查询状态，确定是否已经停止
    if ( !QueryServiceStatusEx(
          hService,
          SC_STATUS_PROCESS_INFO,
          (LPBYTE)&ssp,
          sizeof(SERVICE_STATUS_PROCESS),
          &dwBytesNeeded ) )
    {
          return GetLastError();
    }
    if ( ssp.dwCurrentState == SERVICE_STOPPED )
    {
          return ERROR_SUCCESS;
    }
    // 如果是 STOP_PENDING 状态，则只需等待
    while ( ssp.dwCurrentState == SERVICE_STOP_PENDING )
    {
         Sleep( ssp.dwWaitHint );
         // 循环查询，直到状态改变
         if (!QueryServiceStatusEx(
               hService,
               SC_STATUS_PROCESS_INFO,
               (LPBYTE)&ssp,
               sizeof(SERVICE_STATUS_PROCESS),
               &dwBytesNeeded ) )
        {
               return GetLastError();
        }
        if ( ssp.dwCurrentState == SERVICE_STOPPED )
        {
               return ERROR_SUCCESS;
        }
        if ( GetTickCount() - dwStartTime > dwTimeout )
        {
               return ERROR_TIMEOUT;
        }
    }
    // 先结束依赖服务
    if (fStopDependencies )
    {
         DWORD i;
         DWORD dwBytesNeeded;
         DWORD dwCount;

         LPENUM_SERVICE_STATUS   lpDependencies = NULL;
         ENUM_SERVICE_STATUS     ess;
         SC_HANDLE               hDepService;

         // 使用 0 大小的 buf，获取 buf 的大小
```

```
        // 如果 EnumDependentServices 函数返回成功，说明没有依赖服务
        if ( !EnumDependentServices( hService, SERVICE_ACTIVE,
              lpDependencies, 0, &dwBytesNeeded, &dwCount ) )
        {
              if ( GetLastError() != ERROR_MORE_DATA )
                    return GetLastError(); // Unexpected error
              // 分配缓冲区存储依赖服务的数据
              lpDependencies = (LPENUM_SERVICE_STATUS) HeapAlloc(
                 GetProcessHeap(), HEAP_ZERO_MEMORY, dwBytesNeeded );
              if (!lpDependencies )
                    return GetLastError();
              __try {
                    // 获得依赖服务
                    if (!EnumDependentServices( hService, SERVICE_ACTIVE,
                          lpDependencies, dwBytesNeeded, &dwBytesNeeded,
                          &dwCount ) )
                          return GetLastError();
                    for ( i = 0; i < dwCount; i++ )
                    {
                          ess = *(lpDependencies + i);
                          // 打开服务
                          hDepService = OpenService( hSCM, ess.lpServiceName,
                              SERVICE_STOP | SERVICE_QUERY_STATUS );
                          if ( !hDepService )
                                return GetLastError();
                          __try {
                                // 结束服务
                                if (!ControlService( hDepService,
                                      SERVICE_CONTROL_STOP,
                                      &ss ) )
                                      return GetLastError();
                                // 等待服务结束
                                while ( ss.dwCurrentState != SERVICE_STOPPED )
                                {
                                      Sleep( ss.dwWaitHint );
                                      if (!QueryServiceStatusEx(
                                            hDepService,
                                            SC_STATUS_PROCESS_INFO,
                                            (LPBYTE)&ssp,
                                            sizeof(SERVICE_STATUS_PROCESS),
                                            &dwBytesNeeded ) )
                                            return GetLastError();
                                      if (ss.dwCurrentState == SERVICE_STOPPED )
                                            break;
                                      if ( GetTickCount() - dwStartTime > dwTimeout )
                                            return ERROR_TIMEOUT;
                                }
                          }
                          __finally
                          {
                                // 关闭服务
                                CloseServiceHandle( hDepService );
                          }
                    }
              }
              __finally
              {
                    // 释放内存
                    HeapFree( GetProcessHeap(), 0, lpDependencies );
              }
        }
  }
  // 所有的依赖服务已经结束，结束指定服务
  if (!ControlService( hService, SERVICE_CONTROL_STOP, &ss ) )
        return GetLastError();
  while ( ss.dwCurrentState != SERVICE_STOPPED )
  {
```

```
            Sleep( ss.dwWaitHint );
            if (!QueryServiceStatusEx(
                 hService,
                 SC_STATUS_PROCESS_INFO,
                 (LPBYTE)&ssp,
                 sizeof(SERVICE_STATUS_PROCESS),
                 &dwBytesNeeded ) )
                 return GetLastError();
            if (ss.dwCurrentState == SERVICE_STOPPED )
                 break;
            if (GetTickCount() - dwStartTime > dwTimeout )
                 return ERROR_TIMEOUT;
        }
        return ERROR_SUCCESS;
    }
```

3．配置服务

在创建服务后，仍然可以修改服务的属性，比如可执行程序路径等。获取和设置服务配置使用 API 函数 QueryServiceConfig、QueryServiceConfig2、Change ServiceConfig、ChangeServiceConfig2 和结构 QUER Y_SERVICE_CONFIG。

4．服务的依赖关系

服务与服务之间所存在的依赖关系是指一个服务的运行需要其他服务的支持。比如 telnet 服务的运行就依赖于运行过程调用 RPC 和 TCP/IP 协议驱动等。

一个服务也可以是其他服务的依赖，比如很多服务都依赖于 RPC。可以通过服务属性参看服务的依存关系，如图 8-4 所示。

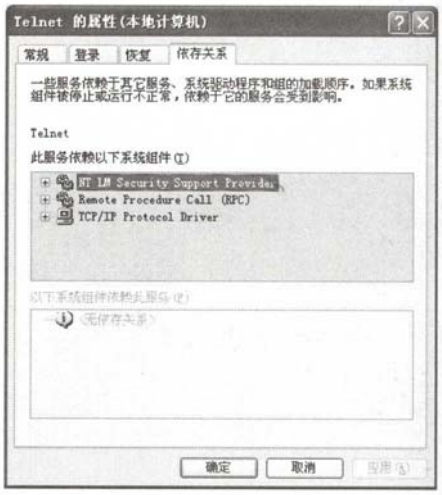

图 8-4　服务的依赖关系

EnumDependentServices API 函数可以列举依赖于一个服务的所有服务。实例 8-6 演示了使用 EnumDependentServices，在停止一个服务前将所有依赖于此服务的其他服务都停止。

5．实例 8-7 实例 8-2~实例 8-6 的主函数及运行结果

本实例首先调用 API 函数 OpenSCManager 打开服务控制器，获得了 SCM 句柄后，依次调用了实例 8-2～实例 8-6 中的各函数，演示了整个服务控制过程。

```
/*************************************
* void main( int argc, TCHAR *argv[] )
* 功能：演示
*************************************/
void main( int argc, TCHAR *argv[] )
{
     TCHAR szBinFilePath[MAX_PATH];
     PTCHAR pTemp;
     DWORD dwStopError;
     // 构造服务可执行程序的路径
     GetModuleFileName(NULL,szBinFilePath,MAX_PATH);
     pTemp = szBinFilePath+lstrlen(szBinFilePath);
     while(*--pTemp!='\\');
     lstrcpy(pTemp,TEXT("\\SplSrv.exe"));

     // 打开 SCM
     schSCManager = OpenSCManager(
         NULL,                    // 本机
         NULL,                    // ServicesActive 数据库
         SC_MANAGER_ALL_ACCESS);  // 完全存取权限
     if (NULL == schSCManager)
          printf("OpenSCManager failed (%d)\n", GetLastError());
     // 创建服务
     CreateSampleService(schSCManager, szBinFilePath, szServiceName);
     // 启动服务
     StartSampleService(schSCManager,szServiceName);
     // 发送请求控制
     ControlSampleService(SERVICE_CONTROL_INTERROGATE);
     ControlSampleService(SERVICE_CONTROL_CONTINUE);
     // 停止服务
     dwStopError = StopService( schSCManager, szServiceName, TRUE, 1000);
     if(ERROR_SUCCESS == dwStopError)
     {
          printf("Service Stoped\n");
     }
     else
     {
          printf("Service stoped error (%u)\n",dwStopError);
     }
     // 删除服务
     DeleteSampleService(szServiceName);
     CloseServiceHandle(schSCManager);
}
```

运行结果如图 8-5 所示。

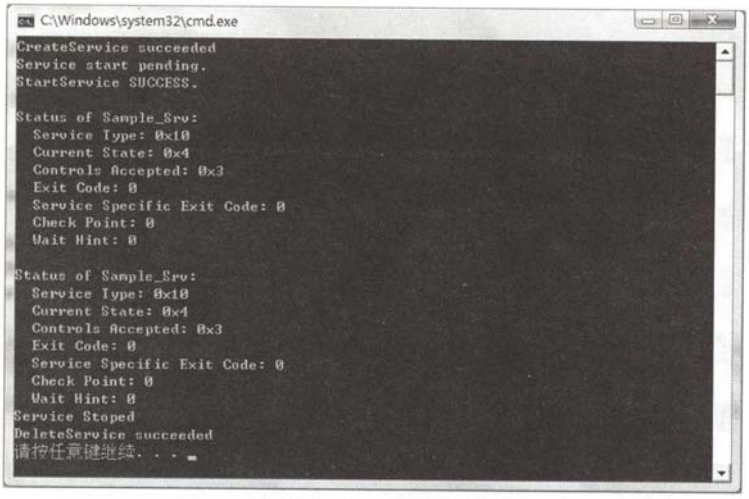

图 8-5　实例运行结果

读者还可以在调试过程中单步运行实例，可以在中间步骤中断然后通过“计算机管理”

（我的电脑→右键→管理）来查看服务的属性。

首先，实例成功新建了服务，服务控制程序向服务发送了控制码，然后显示服务的状态，在本章的示例服务程序中，并没有对服务控制码作过多的处理，在实际应用时，可以根据需求扩展功能。在对服务进行了简单的操作后，服务管理程序停止了服务运行，并删除了服务。

在实例应用中，服务管理程序可以根据需要在安装了服务后不删除，这样服务可以一直驻留在系统中。

在实例注册服务后，删除服务前（可以在调试中，下断点来暂停程序的执行，查看本章所给实例注册的服务），在控制面板和服务管理界面中可以查看到服务的相关属性，如图 8-6 所示。

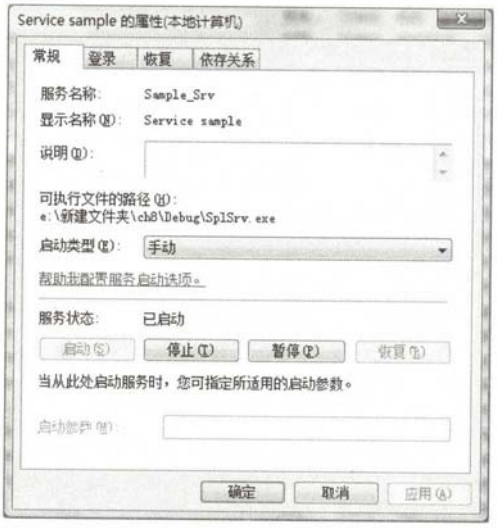

图 8-6　通过“服务管理”查看通过实例新建的服务

第 9 章　图形用户界面

图形用户界面是 Windows 应用程序设计的一个重要内容。图形用户界面决定了软件的易用性，也是软件稳定性的重要影响因素。图形用户界面的好坏是软件是否成功的关键因素之一。

Windows 的用户界面分为两个部分：字符图形界面和图形用户界面。字符图形界面较为简单，主要内容包括如何在界面上读取用户输入，如何向界面终端输出信息。如何控制界面的字体、背景、颜色等。由于字符界面程序本身接口比较单一，而且目前应用较少，不作为重点介绍。

本章重点介绍 Windows 图形用户界面(GUI)程序设计界面的相关内容，主要包括以下空容。

◇如何使用创建窗口；
◇如何处理用户在窗口上的操作和输入；
◇如何控制窗口中的内容；
◇如何使用菜单，包括菜单栏和右键菜单；
◇如何使用 Tree-View 控件；
◇如何使用文件编辑控件；
◇　如何使用 List-View 控件；
◇如何使用对话框。

由于 Windows 图形用户界面的内容很多。尤其是控件的类型很多，由于篇幅限制，本书只选择若干重点部分。Windows 用户界面控件多而杂，本书选取了最具有代表性的文本编辑控件、树形（Tree-View）控件、列表(List-View)控件作重点介绍。

9.1　字符界面程序

本节将首先介绍字符界面程序的基本概念，然后通过实例程序 console.exe 的实现代码展开介绍。console.exe 是一个示例程序，演示了以下内容：

◇如何获取控制台的句柄；
◇如何向控制台写入字符串；
◇如何从控制台读取用户的输入；
◇如何读取控制台目前显示的内容；
◇如何控制控制台的背景颜色、字符颜色、字符背景色等；
◇如何控制控制台的输入 / 输出模式；
◇如何控制控制台的标题；
◇如何获取控制台的相关信息；
◇ 如何使用 Block I/O，控制控制台的屏幕缓冲区；
◇如何消除屏幕；
◇如何才能捕获用户对控制台大小的改变及对控制台的鼠标操作等消息。

9.1.1　基本概念

字符界面的人机交互接口只有一种，称作控制台（Console）。用户通过控制台向应用程

序输入命令和数据，应用程序将运行结果或提示信息等显示在控制台上。

最典型也是最常用的控制台程序是Windows系统命令提示Cmd.exe。控制台界面如图9-1所示。

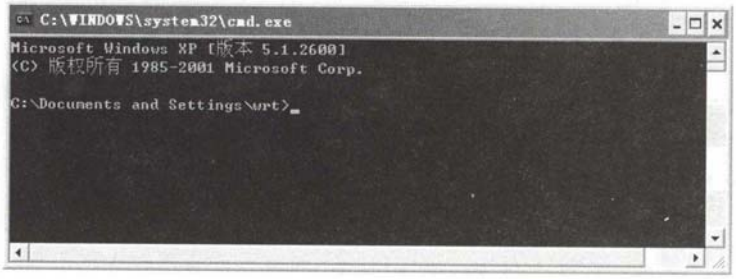

图 9-1　一个典型的控制台界面

1．Console 句柄

字符界面应用程序的所有输入和输出都是通过 Console，字符界面相关的 API 函数都是通过 Console 的句柄来操作控制台的。

Console 句柄分为两种，标准输入(Stand In)句柄和标准输出(Stand Out)句柄，分别用于从界面上读取输入和向界面输出。

大多数情况下，字符界面应用程序都只有一个控制台窗口，其标准输入句柄和标准输出句柄都代表了这个控制台。然而它们确实是两个句柄，所能进行的操作和权限是不一样的。如果使用“标准输入句柄”作为参数，调用 WriteConsole 等用于控制台输出的 API 函数是会产生错误的。

2．关键 API

符号界面 API 数量较少，最常用的是以下 3 个 API 函数。

◇ GetStdHandle:获取句柄，包括本应用程序所拥有的控制台的标准输入和标准输出句柄。

◇ WriteConsole:向 Console 写入字符串。

◇ ReadConsole:从 Console 读取字符串。

另外还有一些设置背景、颜色、字体属性以及处理输入输出缓存的 API，这些 API 函数的详细使用方法在 8.2 节有实例介绍。

3．字符单元(Cell)

控制台的最小单元是字符单元。一个字符单元可以放置一个字符。字符单元是对控制台进行操作的最小单元。所有的属性修改操作也都是以字符单元为单位的。

4．控制台 API 与标准 C 函数

很多常用的标准 C 输入输出函数都是字符界面的，如 printf、scanf 等。Windows 系统通过控制台为这些函数提供了支持。C 语言的标准库只是一种标准，具体的函数实现需要依赖于操作系统。只有操作系统提供支持、使用标准 C 的库函数编写的程序才可以在系统上运行。

实际上，类似于 printf 等标准 C 函数在 Windows 系统中都是通过系统的动态链接库 crtdll.dll 导出，printf 函数的实现程序也位于 crtdll.dll 中，分析 printf 等函数的实现代码可以发现，在 Windows 平台上，实际 printf 函数在做了格式化字符串的处理后，是调用 WriteConsole 等 API 函数进行界面操作的。

在学习了本节的内容后读者完全可以自己使用 Windows API 完成一个 printf 函数的代码实现。

5．控制台子系统

在第 3 章介绍过，Windows 系统的应用程序分别运行于若干子系统。其中 Windows 和

Consle 是两个最常用到的子系统，图形用户界面的程序运行于 Windows 子系统，控制台界面的程序运行于 Consle 子系统。图形用户界面的应用程序窗口是由应用程序调用 API 函数创建的，而控制台应用程序的控制台界面并不需要应用程序创建，系统会为其自动创建。

9.1.2 控制台读写

使用 Windows API 向控制台写入数据获从控制台获取输入的过程如下。

◇使用 GetStdHandle 函数获取本应用程序的控制台标准输入、输出句柄（当然需要本应用程序是一个控制台程序，具有一个控制台。控制台程序在链接时需要选择子系统为 CONSOLE，链接选项：/SUBSYSTEM:CONSOLE）。

◇ 以标准输出句柄为参数，调用 WriteConsole API 写入输出。

◇以标准输入句柄为参数，调用 ReadConsole API 读取输入。

1．关键 API

(1) GetStdHandle。

函数原型如下：

```
HANDLE WINAPI GetStdHandle(
  _in        DWORD nStdHandle
);
```

根据需要获取的句柄类型设置参数为 STDINPUT HANDLE（标准输入句柄）、STD_OUTPUT HANDL（标准输出句柄）或 STD_ERROR HANDLE，返回句柄值。

(2) WriteConsole。

函数原型如下：

```
BOOL WINAPI WriteConsole(
   in         HANDLE hConsoleOutput,
   in         const VOID* lpBuffer,
   in         DWORD nNumberOfCharsToWrite,
   out        LPDWORD lpNumberOfCharsWritten,
  LPVOID lpReserved
);
```

参数 hConsoleOutput 是控制台句柄，应该为标准输出句柄，lpBuffer 为需要输出的内容的指针，nNumberOfCharsToWrite 是需要输出的字符的数量。lpNumberOfCharsWritten 是输出参数，用于返回实际输出的字符数。lpReserved 参数是保留参数，必须设置为 NULL。

(3) ReadConsole。

函数原型如下：

```
BOOL WINAPI ReadConsole(
   in         HANDLE hConsoleInput,
   out        LPVOID lpBuffer,
   in         DWORD nNumberOfCharsToRead,
   out        LPDWORD lpNumberOfCharsRead,
   in_opt     LPVOID pInputControl
);
```

参数 hConsoleOutput 是控制台句柄，应该为标准输入句柄，lpBuffer 是保存读取输入数据缓冲区的指针，nNumberOfCharsToRead 是缓冲区的大小。lpNumberOfCharsRead 是输出参数，保存实际读到的大小。pInputControl 指向 CONSOLE_ READCONSOLE_CONIROL 结构，可设置为 NULL。

2．实例 9-1　控制台 I/O

本实例演示 GetStdHandle、WriteConsole、ReadConsole API 函数的使用。除此之处，还调用了 GetConsoleScreenBufferInfo、ChangeTextColor 等控制台相关的 API 函数，对输出的文件的颜色进行了设置。

```
/* ************************************
* ConsoleIO.c 字符界面程序
**************************************/
/* 头文件 */
#include <windows.h>
/* 函数声明 */
void ScrollScreenBuffer(HANDLE, INT);
VOID HandleInput(LPSTR szInput);
/* 全局变量 */
HANDLE hStdout, hStdin;
CONSOLE_SCREEN_BUFFER_INFO csbiInfo;

/* ************************************
* int main(void)
* 功能: 演示控制台的输入与输出
* 参数: 无      返回值: 无
**************************************/
void main(void)
{
     // 提示
     LPSTR lpszPrompt = "Type a line and press Enter, 255 max \n q to quit "
          "\n i to Show console info\n c to Change backgroud color \n s to Change text color"
          "\n t to Find upper case character\n b to Call UseBlockIO \n u to Change conslos titile"
          "\n m to Show console event \n r to Change mode \n p to Clear up Console\n";

     CHAR chBuffer[256];
     DWORD cRead, cWritten, fdwMode, fdwOldMode;
     WORD wOldColorAttrs;
     //  STDIN 和 STDOUT 句柄
     hStdin = GetStdHandle(STD_INPUT_HANDLE);
     hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
     if (hStdin == INVALID_HANDLE_VALUE ||
          hStdout == INVALID_HANDLE_VALUE)
     {
          // 使用 MessageBox 报错
          MessageBox(NULL, "GetStdHandle", "Console Error", MB_OK);
          return;
```

```
        }
        // 当前字体颜色
        if (!GetConsoleScreenBufferInfo(hStdout, &csbiInfo))
        {
              MessageBox(NULL, "GetConsoleScreenBufferInfo",
                   "Console Error", MB_OK);
              return;
        }
        wOldColorAttrs = csbiInfo.wAttributes;
        // 设置字符颜色(红)和背景
        ChangeTextColor(hStdout, FOREGROUND_RED | FOREGROUND_INTENSITY);
        // 输出字符
        if (!WriteConsole(
              hStdout,                // 输出句柄
              lpszPrompt,             // 输出字符串
              lstrlen(lpszPrompt),    // 字符串长度
              &cWritten,              // 返回实际输出的长度
              NULL) )                 // 非 overlapped
        {
              MessageBox(NULL, "WriteFile", "Console Error", MB_OK);
              return;
        }
        while (1)
        {
              // 循环等待用户输入
              ZeroMemory(chBuffer,256);
              if (! ReadConsole(
                   hStdin,           // 输入控制台句柄
                   chBuffer,         // 读入数据的缓冲区
                   255,              // 缓冲区大小
                   &cRead,           // 实际读到的大小
                   NULL) )           // 非 overlapped
                   break;
              // 处理用户的输入
              HandleInput(chBuffer);
        }
}
/* ************************************
* VOID HandleInput(LPSTR szInput)
* 功能: 处理用户的输入
* 参数: 用户输入的字符串
**************************************/
VOID HandleInput(LPSTR szInput)
{
    switch(szInput[0])
    {
    case 'q':      // 退出进程
         ExitProcess(0);
    case 'i':           // 显示控制台信息
          GetConsoleInfo(hStdout);
          break;
    case 'c':           // 设置界面背景颜色
          ChangeBackgroudColor(hStdout);
          break;
    case 's':           // 设置文件颜色
          ChangeTextColor(hStdout, FOREGROUND_GREEN | FOREGROUND_INTENSITY);
          break;
    case 't':           // 设置文件背景
          ChangeUpCaseTextBackgroudColor(hStdin, hStdout);
          break;
    case 'b':      // 使用 Block IO
          UseBlockIO();
          break;
    case 'u':      // 设置控制台标题
          ChangeConsoleTitle(&szInput[2]);
          break;
    case 'm':      // 使用事件
```

```
            UseEvent();
            break;
        case 'r':           // 改变模式
            ChangeMode(hStdin,hStdout);
            break;
        case 'p':       // 清屏
            cls(hStdout);
            break;
        default:            // 其他，直接返回
            return;
        }
        return;
    }
```

代码开始，首先调用 GetConsoleScreenBufferInfo API 函数，获取了当前控制台的字体和颜色信息，并赋值给了变量（这个变量用于恢复初始设置，本实例中没有再使用到，读者根据需要修改），然后调用 ChangeTextColor 函数改变字体颜色为红色。

之后调用 WriteConsole 函数将 lpszPrompt 字符串显示在控制台上，然后，等待用户的输入。

HandleInput 函数是在用户输入后，根据用户输入信息的不同做不同处理。这些处理都是对控制台进行控制和设置。HandleInput 函数中调用的功能函数的实现代码在实例 9-2 和实例 9-3 中。

3．运行结果

实例的运行结果如图 9-2 所示。

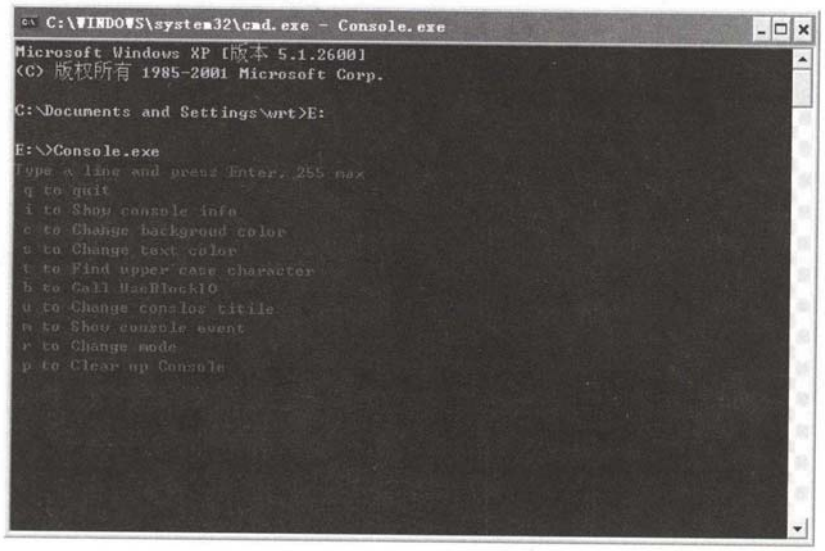

图 9-2　控制台 I/O 结果

9.1.3　控制台字体、颜色等属性，操作屏幕缓存

在如图 9-2 所示 Console.exe 的界面中，输入“i”会显示控制台信息，输入“c”可以改变背景，输入“s”可以改变文件颜色，输入“t”可以将界面上显示的字符中的所有大字字母的背景颜色改变，输入“b”可以使用 BlockI/O 进行输入与输出，输入“u<参数>”可以设置控制台的标题，输入“m”可以演示如何获取控制台事件，输入“r”可以改变输入输出模式。输入“p”可以将屏幕清除，如果要退出程序运行输入“q”。

这些功能都由实例 9—1 中的 HandleInput 函数通过调用各个功能函数来实现的。下面介绍如何实现这些功能。

1．实例 9-2 控制窗口标题、获取控制台信息

如图 9-3 所示，在界面中输入“u abc”，控制台的标题变为“abc”。再输入“i”显示现在有两个进程附加到本控制台（PID 为 1348 和 1336，cmd.exe 和 console.exe）控制台的标题是“abc”，显示窗口的句柄、字体以及字号的大小。

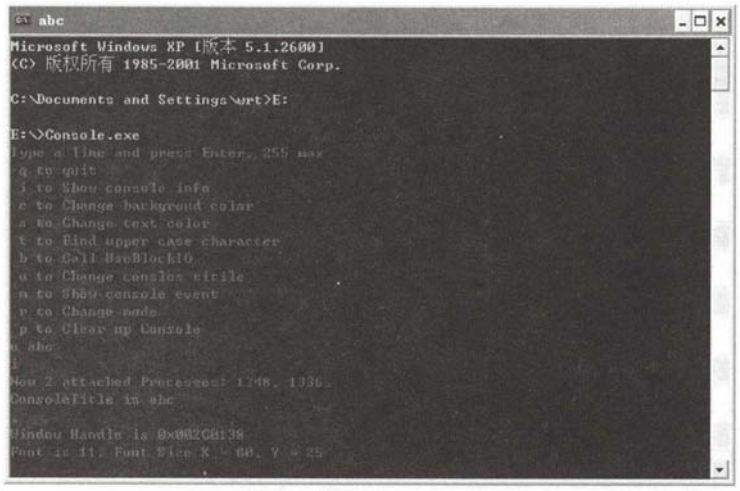

图 9-3　控制控制台标题、获取控制台信息

在程序代码中 ChangeConsoleTitle 函数实现了修改窗口标题的功能。此函数调用了一个 API 函数。SetConsoleTitle、SetConsoleTitle 的参数很简单，是新窗口标题的字符串。SetConsoleTitle 就可以实现修改窗口标题。

```
/* ************************************
* VOID ChangeConsoleTitle(LPSTR szNewTitle)
* 功能: 改变控制台窗口的标题
* 参数: LPSTR szNewTitle, 新标题
************************************/
VOID ChangeConsoleTitle(LPSTR szNewTitle)
{

     if(!SetConsoleTitle(szNewTitle) )
          MyErrorExit("SetConsoleTitle failed\n");
}
```

GetConsoleInfo 函数实现了获取控制台句柄。获取控制台附加的进程使用 GetConsoleProcessList 函数就能实现，GetConsoleWindow、GetCurrentConsoleFont、GetConsoleTitle 这 3 个 API 函数分别实现了获取窗口句柄、字体和标题的功能。

```
/* ************************************
* VOID GetConsoleInfo(HANDLE hOutput)
* 功能: 获取控制台信息
* 参数: HANDLE hOutput, 控制台句柄
************************************/
VOID GetConsoleInfo(HANDLE hOutput)
{
    CONSOLE_FONT_INFO cfi;
    DWORD dwProcessList[32];
    DWORD dwAttachedProcess;
```

```
    TCHAR szOutputBuffer[1024];
    HWND hConsoleWindows;
    DWORD dwWritten;
    DWORD i;
    CHAR szConsoleTitle[MAX_PATH];
    // 附属的进程，不考虑多于32个的情况
    dwAttachedProcess = GetConsoleProcessList(dwProcessList,32);

    if(dwAttachedProcess==0)
    {
        MyErrorExit("GetConsoleProcessList");
    }
    // 标题
    if(!GetConsoleTitle(szConsoleTitle, MAX_PATH) )
    {
        MyErrorExit("GetConsoleTitle");
    }
    // 窗口句柄
    hConsoleWindows = GetConsoleWindow();
    // 字体
    GetCurrentConsoleFont(hOutput,FALSE,&cfi);

    wsprintf(szOutputBuffer,"Now %u attached Processes: ",dwAttachedProcess);
    for(i=0;i<dwAttachedProcess;i++)
    {
        wsprintf(szOutputBuffer+lstrlen(szOutputBuffer),"%u, ",dwProcessList[i]);
    }
    // 构造字符串
     wsprintf(szOutputBuffer+lstrlen(szOutputBuffer),
          "\nConsoleTitle is %s,\nWindow Handle is 0x%.8X\n"
          "Font is %u, Font Size X = %u, Y = %u\n",
          szConsoleTitle,hConsoleWindows,
          cfi.nFont,cfi.dwFontSize.X,cfi.dwFontSize.Y);
    // 显示获取的信息
    if(!WriteConsole(hOutput,szOutputBuffer,lstrlen(szOutputBuffer),&dwWritten,NULL))
    {
        MyErrorExit("WriteConsole");
    }
}
```

2．实例 9-3 修改背景颜色和字符颜色

在 Console.exe 中输入“s”可以将字符颜色由红色改为绿色，输入“c”可以将整个屏幕的背景改为紫色，字体颜色改为白色，效果如图 9-4 所示。

Console.exe 代码中的 ChangeTextColor 和 ChangeBackgroudColor 函数实现了这两项功能。

ChangeTextColor 调用了 SetConsoleTextAttribute 函数，设置了文本颜色属性，而 HandleInput 函数调用 ChangeTextColor 函数时指定的参数，说明字体颜色为绿色，背景增加。修改后，新输入的字符为绿色字符，黑色背景。

```
    ChangeTextColor(hStdout, FOREGROUND_GREEN | FOREGROUND_INTENSITY);
```

ChangeTextColor 函数的实现如下：

```
/* ************************************
* VOID ChangeTextColor(HANDLE hStdout,WORD wColor)
* 功能：改变文字颜色
* 参数：HANDLE hStdout，句柄；WORD wColor，新的颜色
**************************************/
```

```
VOID ChangeTextColor(HANDLE hStdout, WORD wColor)
{
    // 设置文字属性
    if (!SetConsoleTextAttribute(hStdout, wColor))
    {
        MessageBox(NULL, "SetConsoleTextAttribute", "Console Error",
            MB_OK);
        return;
    }
}
```

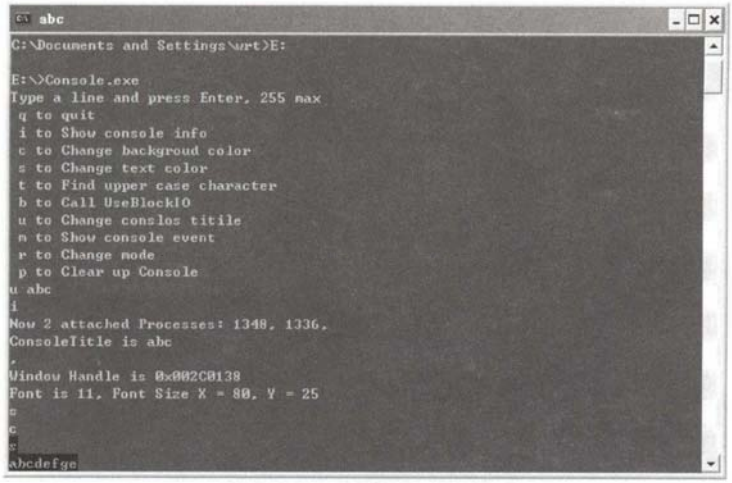

图 9-4　修改背景颜色和字符颜色

ChangeBackgroudColor 函数可以改变整个控制台的颜色，此函数调用了 API 函数 FillConsoleOutputAttribute 实现改变控制台颜色的功能。在设置颜色时，指定了 BACKGROUND_BLUE ｜BACKGROUND_ RED， 这样背景色将变为紫色。又指定了 FOREGROUND_RED ｜ FOREGROUND_GREEN ｜ FOREGROUND_ BLUE，这样字符将变为白色。同时在调用 FillConsoleOutputAttribute 函数时，还指定了所需修改属性的字符单元的起始位置和大小。

```
/* ************************************
* VOID ChangeBackgroudColor(HANDLE hConsole)
* 功能: 改变背景颜色
* 参数: HANDLE hOutput, 控制台句柄
**************************************/
VOID ChangeBackgroudColor(HANDLE hConsole)
{
    COORD coord;
    WORD wColor;
    DWORD cWritten;
    BOOL fSuccess;
    coord.X = 0; // 第一个 Cell
    coord.Y = 0; // 第一行
    // 背景和前景色
    wColor = BACKGROUND_BLUE | BACKGROUND_RED |
        FOREGROUND_RED |
        FOREGROUND_GREEN |
        FOREGROUND_BLUE;
    // 设置控制台属性
    fSuccess = FillConsoleOutputAttribute(
        hConsole,          // Handle 句柄
        wColor,            // 颜色
        80*50,             // 填充的 Cell 数量
```

```
            coord,              // 改变属性的第一个 Cell
            &cWritten);         // 实际改变的情况

    if (!fSuccess)
         MyErrorExit("FillConsoleOutputAttribute");
}
```

3．实例 9-4 屏幕缓冲区和 Block ｜0

每一个控制台都有一个输入缓存和屏幕缓存。输入缓存用于缓存在控制台中的输入。而屏幕缓存用于缓存所有在屏幕上显示的字符及属性。

程序可以通过 API 来操作缓存。如果操作屏幕缓存，那么可以获取所有已经输出过的字符，也可以改变屏幕上已经输入的字符和字符的属性。

如图 9-5 所示，在 Console.exe 中输入“t”，程序会从屏幕缓存中读取所有的字符，判断字符是否为大写，如果为大写字符，则将其所有的字符单元修改背景修改为红色，其后两个单元格修改为黄色和绿色。

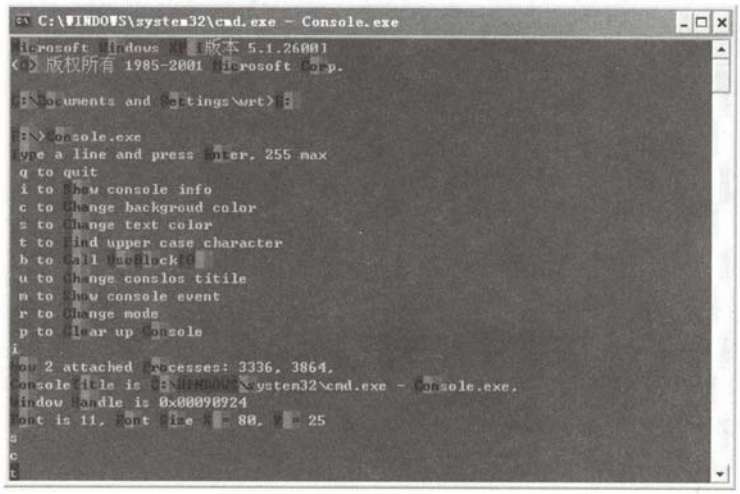

图 9-5　获取屏幕缓冲区的内容，修改屏幕缓冲区

ChangeUpCaseTextBackgroudColor 函数实现了这项功能。

```
/* ************************************
* VOID ChangeUpCaseTextBackgroudColor(HANDLE hStdin, HANDLE hStdout)
* 功能：改变文字背景颜色
* 参数：HANDLE hStdin, HANDLE hStdout，控制台输入输出句柄
**************************************/
VOID ChangeUpCaseTextBackgroudColor(HANDLE hStdin, HANDLE hStdout)
{
     DWORD dwLen, i, dwRead, dwWritten;
     WORD wColors[3];
     BOOL fSuccess;
     COORD coord;
     HANDLE hSreenHandle;
     CONSOLE_SCREEN_BUFFER_INFO csbi;

     PCHAR lpCharacter = HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,2048);
     coord.X = 0;      // 起始 Cell
     coord.Y = 0;
     // 读取 OutputCharacter
     if(!ReadConsoleOutputCharacter(hStdout,lpCharacter,2047,coord,&dwRead))
```

```
        {
            MyErrorExit("ReadConsoleOutputCharacter");
        }
        // 获取 ScreenBufferInfo
        GetConsoleScreenBufferInfo(hStdout,&csbi);
        dwLen = lstrlen(lpCharacter);
        // 颜色数组
        wColors[0] = BACKGROUND_RED;
        wColors[1] = BACKGROUND_RED | BACKGROUND_GREEN;
        wColors[2] = BACKGROUND_GREEN;
        for (i=0 ;i<dwLen; i++)
        {
            // 查找大写字母
            if(lpCharacter[i]>='A'&& lpCharacter[i]<='Z')
            {
                // 如果是大写字母，则将大写字母开始位置的 3 个 Cell 的
                // 文字背景改变成 3 种不同的颜色
                coord.Y = i/csbi.dwSize.X;
                coord.X = i%csbi.dwSize.X;
                fSuccess = WriteConsoleOutputAttribute(
                    hStdout, // handle
                    wColors, 3,
                    coord, // 起始位置
                    &dwWritten);
                if(!fSuccess)
                {
                    MyErrorExit("WriteConsoleOutputAttribute");
                }
            }
        }
}
```

API 函数 ReadConsoleOutputCharacter 可以从屏幕缓存中读取输出的字符。GetConsoleScreenBufferInfo 和 WriteConsoleOutputAttribute 函数可以获取和设置屏幕缓存的属性。

ChangeUp CaseTextBackgroudColor 函数先使用 ReadConsoleOutputCharacter 函数获取了所有在屏幕输出了的字符，依次判断字符是否为大写，如果为大写则使用 WriteConsoleOutputAttribute 改变此字及其后两个字符单元的背景。

实例代码中另一个与屏幕缓冲区相关的函数是 cls 函数，cls 函数具有清屏的功能。

cls 函数更能体现对屏幕输出缓冲操作的效果。在 Console.exe 中输入“p”，会将屏幕清空。

清屏后，屏幕上的所有字符都被清除，如图 9-6 所示。

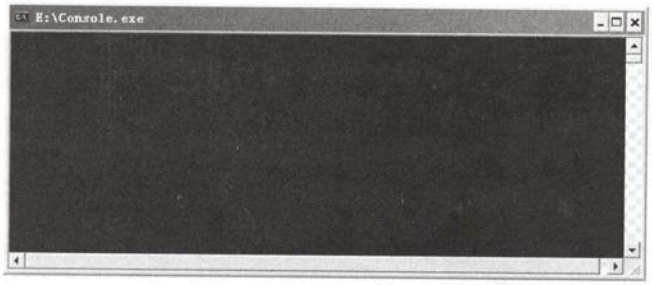

图 9-6 清屏

cls 函数的实现代码如下：

```
/* ************************************
* void cls( HANDLE hConsole )
* 功能：清屏
* 参数：HANDLE hConsole，句柄
```

```
************************************/
void cls( HANDLE hConsole )
{
     COORD coordScreen = { 0, 0 }; // 鼠标指针位置
     DWORD cCharsWritten;
     CONSOLE_SCREEN_BUFFER_INFO csbi;
     DWORD dwConSize;
     // 当前 Buffer 中的文字 Cell 数目
     if(!GetConsoleScreenBufferInfo( hConsole, &csbi ))
          return;
     dwConSize = csbi.dwSize.X * csbi.dwSize.Y;
     // 将整个屏幕填充
     if(!FillConsoleOutputCharacter( hConsole, (TCHAR) ' ',
          dwConSize, coordScreen, &cCharsWritten ))
          return;
     // 获取文字属性
     if(!GetConsoleScreenBufferInfo( hConsole, &csbi ))
          return;
     // 设置 buffer 属性
     if(!FillConsoleOutputAttribute( hConsole, csbi.wAttributes,
          dwConSize, coordScreen, &cCharsWritten ))
          return;
     // 将鼠标指针移动起始位置
     SetConsoleCursorPosition( hConsole, coordScreen );
}
```

cls 函数首先调用 API 函数 GetConsoleScreenBufferInfo 获取了屏幕缓存的信息，然后调用 API 函数 FillConsoleOutputCharacter 将整个屏幕填充为空。再调用 FillConsoleOutputAttribute 函数设置了输入属性，最后再调用 SetConsoleCursorPosition 函数将输入指针设置为屏幕的起始位置。这样就完成了清屏的功能。

实例代码中的 UseBlockIO 函数实现了更高级屏幕缓存操作功能。UseBlockIO 函数为控制台创建了多个缓存，并通过将指定的缓存设置为“活动的”(Active)来快速切换屏幕上显示的内容。

在 Console.exe 中输入“b”运行，程序将调用 UseBlockIO 函数。

UseBloclIO 函数调用了一系列 API 函数。首先调用 GetStdHandle 函数获得标准输出句柄，然后调用函数 CreateConsoleScreenBuffer 新建一个屏幕缓存，返回的是一个控制台标准输出句柄，之后使用 SetConsoleActiveScreenBuffer 函数将新建的缓存设置为屏幕的活动缓存。再使用 ReadConsoleOutput 函数将原屏幕缓存中的前两行读到内存。最后使用 WriteConsoleOutput 写到当前的屏幕缓存的指定位置中。

> **注意** 输出缓存是使用句柄表示的，实际上之前所讲到的控制台标准输出句柄就是代表了当前的控制台输出缓存。

Block I/O 的运行效果如图 9-7 所示。

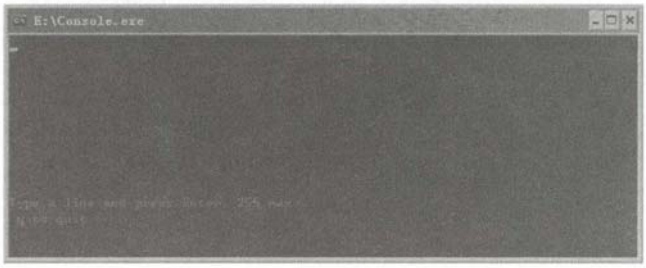

图 9-7 Block I/O

UseBlockIO 函数的实现代码如下：

```
/* ************************************
* VOID UseBlockIO ()
* 功能：使用 Block IO
**************************************/
VOID UseBlockIO()
{
    HANDLE hStdout, hNewScreenBuffer;
    SMALL_RECT srctReadRect;
    SMALL_RECT srctWriteRect;
    CHAR_INFO chiBuffer[160]; // [2][80];
    COORD coordBufSize;
    COORD coordBufCoord;
    BOOL fSuccess;
    // 获得句柄
    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    // 新建 buffer
    hNewScreenBuffer = CreateConsoleScreenBuffer(
        GENERIC_READ |      GENERIC_WRITE,// 可读可写
        0, // 不共享
        NULL, // 默认安全属性
        CONSOLE_TEXTMODE_BUFFER, // must be TEXTMODE
        NULL);
    if(hStdout == INVALID_HANDLE_VALUE ||
        hNewScreenBuffer == INVALID_HANDLE_VALUE)
    {
        printf("CreateConsoleScreenBuffer (%d)\n", GetLastError());
        return;
    }
    // 设置屏幕的 Buffer
    if (!SetConsoleActiveScreenBuffer(hNewScreenBuffer) )
    {
        printf("SetConsoleActiveScreenBuffer (%d)\n", GetLastError());
        return;
    }
    // 设置源 Cell 矩阵
    srctReadRect.Top = 0; // top left: row 0, col 0
    srctReadRect.Left = 0;
    srctReadRect.Bottom = 1; // bot. right: row 1, col 79
    srctReadRect.Right = 79;
    // 临时 buffer 大小为 2 行 3 例
    coordBufSize.Y = 2;
    coordBufSize.X = 80;
    // 临时 buffer 的起始位置
    coordBufCoord.X = 0;
    coordBufCoord.Y = 0;
    // 从屏幕 buffer 复制内容到临时 buffer
    fSuccess = ReadConsoleOutput(
         hStdout,           // 源
         chiBuffer,         // 目的
         coordBufSize,      // 目的 buffer 大小
         coordBufCoord,     // 目的起始位置
         &srctReadRect);    // 源起始位置
    if (! fSuccess)
    {
         printf("ReadConsoleOutput (%d)\n", GetLastError());
         return;
    }
    // 设置目的 Cells 矩阵
    srctWriteRect.Top = 10; // top lt: row 10, col 0
    srctWriteRect.Left = 0;
```

```
        srctWriteRect.Bottom = 11; // bot. rt: row 11, col 79
        srctWriteRect.Right = 79;
        // 从临时buffer向新的屏幕buffer复制
        fSuccess = WriteConsoleOutput(
             hNewScreenBuffer,
             chiBuffer,
             coordBufSize,
             coordBufCoord,
             &srctWriteRect);
        if(!fSuccess)
        {
             printf("WriteConsoleOutput (%d)\n", GetLastError());
             return;
        }
        // 等一段时间
        Sleep(10000);
        // 恢复
        if(!SetConsoleActiveScreenBuffer(hStdout))
        {
             printf("SetConsoleActiveScreenBuffer (%d)\n", GetLastError());
             return;
        }
    }
```

运行程序后，原来的屏幕中的前两行移到的屏幕的中部。等待一段时间后（约 10s），原始的屏幕又被恢复。这是因为程序在最后调用了 SetConsoleActiveScreenBuffer 函数将原来的缓存再设置为控制台当前的活动缓存。

4．实例 9-5 控制台模式

运行 Console.exe，输入“r”，会改变控制台的模式。在改变模式前，用户在控制台中输入单个字符程序并不响应（ReadConsole 函数不返回），需是等到用户输入回车后，再做处理。

如图 9-8 所示，在改变模式后，用户每输入一个字符，后台界面都做出响应，直到重新输入“q”返回到之前的模式。

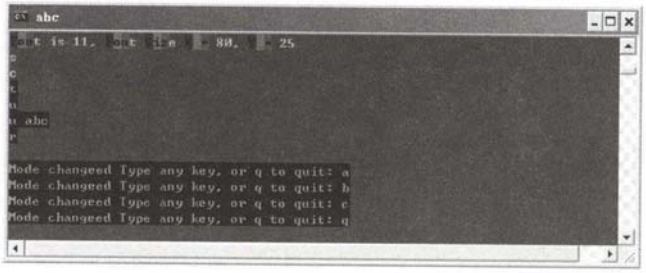

图 9-8　控制台输入模式改变的效果

分析代码，程序在控制台原来的模式上去掉了 ENABLE LINE INPUT 和 ENABLE ECHO INPUT，然后设置了新的模式。

控制台有多种模式：

```
ENABLE_ECHO_INPUT
ENABLE_INSERT_MODE
ENABLE_LINE_INPUT
ENABLE_MOUSE_INPUT
ENABLE_PROCESSED_INPUT
ENABLE_QUICK_EDIT_MODE
ENABLE_WINDOW_INPUT
ENABLE_PROCESSED_OUTPUT
ENABLE_WRAP_AT_EOL_OUTPUT
```

ENABLE_LINE_INPUT 模式表示直到用户输入回车后才读取输入缓存，如果不设置 ENABLE_LINEINPUT 模式，那么只要缓存中有数据，就可以读取。

ENABLE ECHO_INPUT 表示用户键入输入缓存的信息也会立即被复制到输出缓存中。从代码中可以看出，设置模式后，程序在获取用户输入后，又将用户输入写入到了输出缓存。这

是使用代码实现的，而在设置了 ENABLE ECHO_ INPUT 模式下，这种过程是自动的。ENABLE_ ECHO_ INPUT 只有在设置了 ENABLE LINE INPUT 模式的情况下才是有效的。

ChangeMode 函数中使用了 ReadFile 和 WriteFile 函数来读写控制台缓存，在这里 ReadFile 和 WriteFile 函数与 ReadConsole 和 WriteConsole 函数功能完全相同，与模式没有关系。

```
/* ************************************
* VOID ChangeMode( HANDLE hStdin, HANDLE hStdout)
* 功能：改变控制台的模式，关闭 ENABLE_LINE_INPUT 和 ENABLE_ECHO_INPUT
* 参数：HANDLE hStdin, HANDLE hStdout，句柄
************************************/
VOID ChangeMode( HANDLE hStdin, HANDLE hStdout)
{
    LPSTR lpszPrompt = "Mode changeed Type any key, or q to quit: ";
    CHAR chBuffer[256];
    DWORD fdwMode, fdwOldMode;
    DWORD cRead,cWritten;
    // 获取当前模式
    if(!GetConsoleMode(hStdin, &fdwOldMode))
    {
        MessageBox(NULL, "GetConsoleMode", "Console Error", MB_OK);
        return;
    }
    // 修改模式并重新设置
    fdwMode = fdwOldMode &
        ~(ENABLE_LINE_INPUT | ENABLE_ECHO_INPUT);
    if (! SetConsoleMode(hStdin, fdwMode))
    {
        MessageBox(NULL, "SetConsoleMode", "Console Error", MB_OK);
        return;
    }
    // 换行
    NewLine();
    // 循环等待输入并处理
    while(1)
    {
        if(!WriteFile( hStdout,
            lpszPrompt,  lstrlen(lpszPrompt),
            &cWritten, NULL) )
        {
            MyErrorExit("WriteFile");
            return;
        }
        if(!ReadFile(hStdin, chBuffer, 1, &cRead, NULL))
            break;
        if(chBuffer[0] == '\r')
            NewLine();
        else if(!WriteFile(hStdout, chBuffer, cRead,
            &cWritten, NULL)) break;
        else
            NewLine();
        if(chBuffer[0] == 'q') break; // 输入 q，退出
    }
    // 恢复模式
    SetConsoleMode(hStdin, fdwOldMode);
}
/* ************************************
* void NewLine(void)
* 功能：新的一行，滚动屏幕
```

```
************************************/
void NewLine(void)
{
     if(!GetConsoleScreenBufferInfo(hStdout, &csbiInfo))
     {
          MessageBox(NULL, "GetConsoleScreenBufferInfo",
               "Console Error", MB_OK);
          return;
     }
     csbiInfo.dwCursorPosition.X = 0;

     if((csbiInfo.dwSize.Y-1) == csbiInfo.dwCursorPosition.Y)
     {
          ScrollScreenBuffer(hStdout, 1);
     }
     else csbiInfo.dwCursorPosition.Y += 1;

     if(!SetConsoleCursorPosition(hStdout,
          csbiInfo.dwCursorPosition))
     {
          MessageBox(NULL, "SetConsoleCursorPosition", "Console Error",
               MB_OK);
          return;
     }
}
```

9.1.4 控制台事件

如果在控制台模式中设置了 ENABLE_WINDOW INPUT 和 ENABLE_MOUSE_INPUT 等模式后，控制台可以响应用户发起的设置鼠标指针和窗口大小等事件。

设置这些模式后，用户只要把鼠标指针移动到窗口就会产生鼠标事件，如果修改窗口大小会触发窗口事件。

实例 9-6 控制台事件

在 Console.exe 输入“m”，控制台开始响应这些事件，当有鼠标光标移动到窗口后会弹出如图 9-9 所示的对话框。

图 9-9 鼠标控制台的响应

在代码中，使用 ReadConsoleInput 函数等待事件的发生，在产生事件后根据返回的 INPUT RECORD 结构变量中的 EventType 判断事件的类型。

```
/* ************************************
* Event.c 字符界面程序
************************************/
/* 头文件 */
#include <windows.h>
/* 函数声明 */
VOID KeyEventProc(KEY_EVENT_RECORD);
VOID MouseEventProc(MOUSE_EVENT_RECORD);
VOID ResizeEventProc(WINDOW_BUFFER_SIZE_RECORD);

/* ************************************
* DWORD UseEvent(VOID)
* 功能：使用事件进行控制台操作
************************************/
DWORD UseEvent(VOID)
{
    CHAR chBuffer[256];
    DWORD cRead;
    HANDLE hStdin;
```

```
    DWORD cNumRead, fdwMode, fdwSaveOldMode, i;
    INPUT_RECORD irInBuf[128];

    // 获取标准输入句柄
    hStdin = GetStdHandle(STD_INPUT_HANDLE);
    if(hStdin == INVALID_HANDLE_VALUE)
        MyErrorExit("GetStdHandle");
    // 保存当前的控制台模式
    if(!GetConsoleMode(hStdin, &fdwSaveOldMode) )
        MyErrorExit("GetConsoleMode");
    // 使能窗口鼠标输入事件
    fdwMode = ENABLE_WINDOW_INPUT | ENABLE_MOUSE_INPUT;
    if(!SetConsoleMode(hStdin, fdwMode) )
        MyErrorExit("SetConsoleMode");
    // 循环读取输入
    while(1)
    {
        // 等待事件
        if(!ReadConsoleInput(
            hStdin,           // 输入句柄
            irInBuf,          // 保存输入的缓冲区
            128,              // 缓冲区大小
            &cNumRead) )      // 实际读取的大小
             MyErrorExit("ReadConsoleInput");
        // 显示事件
        for(i= 0; i < cNumRead; i++)
        {
            switch(irInBuf[i].EventType)
            {
            case KEY_EVENT: // 键盘输入
                KeyEventProc(irInBuf[i].Event.KeyEvent);
                break;
            case MOUSE_EVENT: // 鼠标输入
                MouseEventProc(irInBuf[i].Event.MouseEvent);
                break;
            case WINDOW_BUFFER_SIZE_EVENT: // Resize
                ResizeEventProc(
                    irInBuf[i].Event.WindowBufferSizeEvent);
                break;
            case FOCUS_EVENT:  // focus 事件
            case MENU_EVENT:   // menu 事件
                break;

            default:
                MyErrorExit("unknown event type");
                break;
            }
        }
    }
    return 0;
}

/* ************************************
* VOID KeyEventProc(KEY_EVENT_RECORD ker)
* 功能: 使用对话框显示键盘输入事件
* 参数: KEY_EVENT_RECORD ker, 键盘事件
**************************************/
VOID KeyEventProc(KEY_EVENT_RECORD ker)
{
    CHAR szMsg[256];
    wsprintf(szMsg,"KEY_EVENT_RECORD\n Char = %c",
        ker.uChar);
    MessageBox(NULL,szMsg,"EVENT",MB_OK);
}

/* ************************************
```

```
* VOID MouseEventProc(MOUSE_EVENT_RECORD mer)
* 功能：使用对话框显示鼠标事件
* 参数：MOUSE_EVENT_RECORD mer，鼠标事件
*************************************/
VOID MouseEventProc(MOUSE_EVENT_RECORD mer)
{
    CHAR szMsg[256];
    wsprintf(szMsg,"MOUSE_EVENT_RECORD\n button state: %d\nmouse position X=%u,Y=%u",
        mer.dwButtonState,mer.dwMousePosition.X,mer.dwMousePosition.Y);
    MessageBox(NULL,szMsg,"EVENT",MB_OK);
    if(IDOK == MessageBox(NULL,"Exit?","EVENT",MB_OKCANCEL))
    {
        ExitProcess(0);
    }
}

/* ************************************
* VOID ResizeEventProc(WINDOW_BUFFER_SIZE_RECORD wbsr)
* 功能：重置大小事件
* 参数：WINDOW_BUFFER_SIZE_RECORD wbsr 窗口消息事件
*************************************/
VOID ResizeEventProc(WINDOW_BUFFER_SIZE_RECORD wbsr)
{
    CHAR szMsg[256];
    wsprintf(szMsg,"WINDOW_BUFFER_SIZE_RECORD\nX=%u,Y=%u",
        wbsr.dwSize.X,wbsr.dwSize.Y);
    MessageBox(NULL,szMsg,"EVENT",MB_OK);
}
```

9.2 图形用户界面：基本概念

从本节开始将介绍 Windows 图形用户界面程序设计的相关内容。

Windows 有稳定而成熟的用户界面程序架构和开发接口。在进行深入学习之前，读者需要了解 Widnows 系统为用户界面提供的程序设计模式。

9.2.1 窗口

窗口是 Windows 图形用户界面最重要的元素。任务图形用户界面的应用程序都需要开窗口。如图 9-10 所示，一个典型的窗口包括多个组成部分。最上一栏是标题栏，标题栏包括应用程序图标和窗口标题，标题栏的右边是一组按钮。标题栏下方放置菜单的部分，称为菜单栏。中间空白部分是客户区，一般用于放置窗口的功能控件、显示内容等，也是程序设计时需要主要处理的内容。

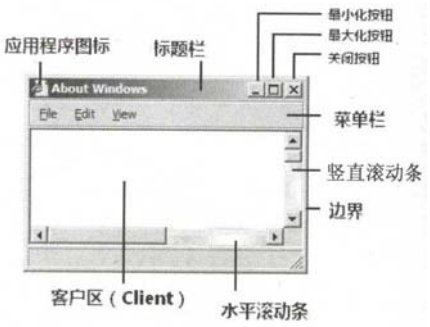

图 9-10 一个典型的窗口

边界上有水平滚动条、竖直滚动条和边界。滚动条可以滑动，有的界面的边界可以拖曳，以改变窗口的大小。窗口是具有继承关系的，一个窗口可以有若干个子窗口。

9.2.2 窗口类

在应用程序开发时，需要注意每一个窗口都属性一个窗口类(Window Class)。每一种窗口类具有对固定的用户输入的处理模式，第一类窗口具有相同的菜单项、背景、图标、鼠标指针样子和窗口消息处理函数。

应用程序在创建窗口前，需向系统注册窗口类，或使用系统已经定义好的窗口类（一些公用的控件所需的窗口类已经由系统定义好了)。在注册窗口类时，需指定窗口消息的处理函数。

9.2.3 消息和消息处理函数

Windows 系统是消息(Message)驱动的。消息在 Windows 处理用户交互的过程中具有重要作用。当用户操作输入设备，产生硬件中断后，系统内核会得到这此硬件中断，硬件在系统中经过复杂的运算和处理最终变成消息。消息被系统发送给指定窗口。

所有的对用户界面的操作都会产生消息，包括鼠标单击、按钮、菜单选择、窗口创建、窗口移动等。Windows 消息种类很多。

在创建窗口时，需指定窗口所属的窗口类，前面提到，每一个窗口类都对应一个消息处理函数，因此每一个窗口都具有自己的消息处理函数。程序员在开发窗口应用程序时，需编写窗口消息处理函数。消息处理函数是一个回调(Call Back)函数。此函数并不是由应用程序调用，而是由系统调用的。应用程序向系统注册了窗口类，创建了窗口，系统在消息产生后，需要窗口消息处理函数来处理时，调用窗口所对应的消息处理函数。

消息处理函数的功能一般是相应用户在窗口上的操作和输入。比如用户在菜单中选中了一个菜单项，那么就会产生一个消息。然后，窗口的消息处理函数就会被调用。消息处理函数被调用后根据消息的类型和参数判断应该如何处理用户的输入，然后进行相关的处理。

这就是 Windows 窗口程序对用户输入的处理模式。

当然并不是只有用户的输入才会产生消息。在很多情况下，一些随机发生的事件也会产生消息，这些事件可能会对窗口应用程序的运行产生影响，比如系统关机、外设插入、计时器计时到达等。窗口处理函数需根据情况处理所需处理的消息。

9.2.4 控件

控件是用户界面开发的重要内容。所谓控件，实际是一种特殊的窗口。Windows 系统将一些常用的窗口模式定义为控件。应用程序可以直接使用这些控件而不需要重新定义。

控件分为很多种，在 Windows 平台下，常用的控件有以下几种（不包括全部)：

◇文本框（Edit、Rich Edit 等)；

◇按钮(Button，包括 Check Boxes、Group Boxes、Push Buttons、Radio Buttons)；

◇ 下拉菜单(ComboBox)；

◇ 列表框(List Box)；

◇分组列表(List View)；

◇分页(Pager)；

◇进度条(Process Bar)；

◇ 属性页(Property Sheet)；

◇工作条(ToolBar)；

◇树(Tree View)。

控件有特定的窗口样式、显示设置和消息处理过程。使用控件时不再需要做消息的处理。系统已经为每个控件定义了特定的消息处理模式。比如 Edit 控件就会在将鼠标样式变为一个竖线；

在用户输入了文字后，会将文字显示在特定的位置；当用户在按住鼠标右键移动时会选中框住的文字，当用户单击右键时会显示特定的菜单，当用户选择菜单项后，进行的操作也是固定的。再比如 Check Box 控件（选择框），当用户单击时会将状态转换，并会在选择框中标上“√”等符号。

这些都是一个控件事先定义好的处理模式。可以说控件是模式化了的窗口。达到了方便用户使用的目的。

本章 9.4 节会使用实例说明如何使用控件。

9.2.5 资源

资源是应用程序在运行中可能会使用到的一些数据。资源包括图标、位图、菜单等。在程序设计时，开发人员可以在资源脚本(. rc)中配置资源，在程序构建时，资源编译器会将资源编译为资源文件(. res)，然后由链接器将资源链接到可执行文件中。在程序运行时使用这些资源。

本章在 9.5 节将介绍如何在应用程序中使用资源以及如何编写资源脚本。

9.2.6 对话框

对话框是一种临时的窗口，一般用来显示信息、获取用户的输入等。对话框也是一种窗口，也可以使用控件（子窗口），也有消息处理函数等。

但是对话框又是一种特殊的窗口，不同点在于可以通过资源来创建对话框，对话框及对话框上所显有的控件都可以在资源脚本中配置，而窗口不可以。对话框具有所属窗口，在对话框处理完成，关闭之前，拥有对话框的窗口不能进行操作。

实例 9.7 节将演示对话框的使用。

9.3 图形用户界面：窗口

从本节开始将使用实例介绍如何通过代码实现图形用户界面功能。9.3～9.6 节的所有实例都是本书配套实例 Windows.exe 的程序代码，实例的运行结果如图 9-11 所示。

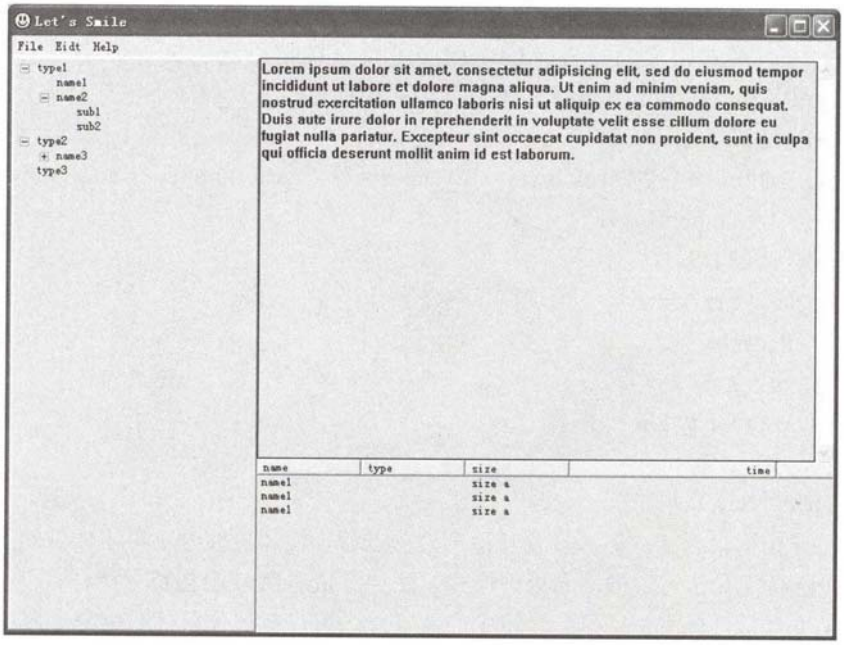

图 9-11　窗口界面实例运行效果

```
/* ************************************
* WinMain
* 功能：程序入口点，调用 InitApplication，InitInstance
*               消息循环
* 参数：hinstance，应用程序本次运行实例
*               hPrevInstance，应用程序之前的实例，始终为 NULL
*               lpCmdLine，命令行参数
*               nCmdShow，窗口显示方式，SW_SHOW
* 返回值：失败返回 FALSE
************************************/
int WINAPI WinMain(
                    HINSTANCE hinstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
     MSG msg;
     BOOL fGotMessage;
     //      注册窗口类
     if (!InitApplication(hinstance))
     {
          return FALSE;
     }
     //      创建窗口和控件子窗口
     if(!InitInstance(hinstance, nCmdShow))
     {
          return FALSE;
     }
     // 消息循环
     while((fGotMessage = GetMessage(&msg, (HWND) NULL, 0, 0)) != 0 && fGotMessage != -1)
     {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }
     return msg.wParam;
     UNREFERENCED_PARAMETER(lpCmdLine);
}
```

在 WinMain 函数中，调用了 InitApplication 注册窗口类，调用 InitInstance 函数实

现了窗口和控件子窗口的创建，然后是消息循环。

GetMessage、TranslateMessage、DispatchMessage 是用于获取消息和分派消息的 API 函数。

9.3.1 注册窗口类

前面已经介绍，任何一个窗口都必须属于一个窗口类，同一类的窗口具有同样窗口消息处理过程。

Windows API 函数 RegisterClass 和 RegisterClassEx 可以完成窗口类的注册。

函数的原型如下：

Windows.exe 程序有一个主窗口，还有 3 个子窗口，3 个子窗口是 3 个控件，分别是 Tree View 控件、文件框控件和 List View 控件。其中 Tree View 控件演示了如何为 Tree View 增加节点，实现单击节点的右键菜单。文件框控件演示了如何为文件框设置文本。List View 控件演示了如何设置分栏和栏中的项。

本节先给出 Windows.exe 程序的主函数，然后再分别介绍主函数 WinMain 的各个部分，WinMain 函数的代码如下：

```
ATOM RegisterClass(
        CONST WNDCLASS *lpWndClass
);
ATOM RegisterClassEx(
        CONST WNDCLASSEX *lpwcx
);
```

这两个函数分别使用了 WNDCLASS 和 WNDCLASSEX 结构作为参数，其中 WNDCLASS 的定义如下：

```
typedef struct {
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HINSTANCE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
} WNDCLASS, *PWNDCLASS;
```

其中 style 是窗口样式，通过系统预先定义的一些常量来设置窗口样子，这些常量包括 CS_BYTEALIGNCLIENT（表示窗口客户区以 byte 的宽度对齐）、CS_DBLCLKS（如果有双击会向窗口发送消息）、CS_NOCLOSE（无关闭按钮）等。

lpfnWndProc 是消息处理函数。

hInstance 应该赋值为应用程序的实例。

hIcon，hCursor，hbrBackground，lpszMenuName 分别为窗口类的图标、鼠标指针的样式、背景的画刷（COLOR_ BACKGROUND、COLOR_WINDOW 等，也可以自行建立画刷）、菜单名。可以从资源获取图标、鼠标指针的样式和菜单。

lpszClassName 是新建类的类名，在创建窗口时需要指定窗口所属类的类名。

这两个函数返回表示窗口类的 ATOM。

实例 9-7 注册窗口类

实例代码中的 InitApplication 函数注册了窗口类，代码如下：

```
/* ************************************
* BOOL InitApplication(HINSTANCE hinstance)
* 功能：注册主窗口类
* 参数：hinstance，应用程序本次运行实例
* 返回值：是否成功
************************************/
BOOL InitApplication(HINSTANCE hinstance)
{
    // 使用 RegisterClassEx
    WNDCLASSEX wcx;
    // 填充结构
    wcx.cbSize = sizeof(wcx); // WNDCLASSEX 结构的大小
    wcx.style = CS_HREDRAW | CS_VREDRAW; // 如果大小改变了重绘窗口
    wcx.lpfnWndProc = MainWndProc; // 窗口消息处理函数
    wcx.cbClsExtra = 0; // 无附加窗口类内存
    wcx.cbWndExtra = 0; // 无附加窗口内存
    wcx.hInstance = hinstance; // 应用程序实例
    wcx.hIcon = LoadIcon(NULL,     MAKEINTRESOURCE(IDI_ICON_MAIN)); // 图标
    wcx.hCursor = LoadCursor(NULL,     IDC_ARROW); // 鼠标指针
    wcx.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH); // 背景画刷
    wcx.lpszMenuName = MAKEINTRESOURCE(IDR_MENU_MAIN); // 菜单资源
    wcx.lpszClassName = "MainWClass"; // 窗口类名
```

```
    wcx.hIconSm = (HICON)LoadImage(hinstance, // 小图标
        MAKEINTRESOURCE(IDI_ICON_MAIN),
        IMAGE_ICON,
        GetSystemMetrics(SM_CXSMICON),
        GetSystemMetrics(SM_CYSMICON),
        LR_DEFAULTCOLOR);
    // 注册窗口类，返回
    return RegisterClassEx(&wcx);
}
```

窗口类的样式是 CS_HREDRAW 和 CS_VREDRAW 表示窗口的长度或宽度变化后将会重绘窗口；hInstance 字段设置为 WinMain 函数的参数 hinstance；MainWndProc 是窗口类所属窗口的消息处理函数；鼠标为系统样式 IDC。ARROW（箭头指针）；窗口类的菜单和图标都是使用应用程序的资源。

窗口类的注册虽然只需填充一个结构体，调用一个 API 函数，但是窗口注册结构 WNDCLASSEX 或 WNDCLASS 结构的设置较复杂，需注意设置的正确性。

9.3.2 创建窗口

WinMain 函数调用的 InitInstance 函数实现了程序主窗口的创建。CreateWindow 和 CreateWindowEx 是用于创建窗口的 API 函数。

1. CreateWindow 函数

CreateWindow 函数的功能是创建一个窗口。在创建窗口前，需先注册窗口类。如果创建的窗口是系统控件，那么系统控制的类已经由操作系统注册好，所以不再需要注册。

创建窗口时，通过窗口指定窗口名、窗口样式、窗口的大小和位置、窗口所属的父窗口等。

```
HWND CreateWindow(
  LPCTSTR lpClassName,
  LPCTSTR lpWindowName,
  DWORD dwStyle,
  int x,
  int y,
  int nWidth,
  int nHeight,
  HWND hWndParent,
  HMENU hMenu,
  HINSTANCE hInstance,
  LPVOID lpParam
);
```

lpClassName 参数是窗口所需窗口类的类名。

lpWindowName 参数是窗口名，如果窗口有标题栏，窗口名将显示在标题栏上。

dwStyle 是窗口样式。窗口是否具有标题栏，是否可改变大小，边框的样式等都通过这个参数设置，表 9.1 列出了最常用的窗口样式。

表 9.1 窗口样式

样　式	意　义
WS_BORDER	具有边框
WS_CAPTION	具有标题栏
WS_CHILD	子窗口
WS_HSCROLL	具有水平滚动条
WS_MAXIMIZE	创建最大化窗口

续表

样　式	意　义
WS_MINIMIZE	创建最小化窗口
WS_POPUP	顶层窗口(无父窗口,不能和 WS_CHILD 同时使用)
WS_SIZEBOX	具有拖拉窗口边框，窗口可改变大小
WS_SYSMENU	窗口有菜单（窗口必须同时具有标题栏）
WS_TILED	具有标题栏和边框
WS_VISIBLE	窗口可见
WS_VSCROLL	具有竖直滚动条

比如 Window.exe 在创建位于中部到右上角的文本框控件时，就指定了文件框的样子包括 WS_BORDER，在窗口显示出来后，文本框控制边上有一个黑边框，如图 9-11 所示。

参数 x、y 指定了窗口创建后的初始位置，编程时可以设置为 CW_USEDEFAULT，由系统来设置。

参数 nWidth、nHeight，表示窗口的宽和高，以像素点为单位，同样也可以指定为 CW_USEDEFAULT，使用系统默认。

参数 hMenu 是窗口菜单的句柄，如果设置为 NULL，表示使用窗口类的菜单。

hWndParent 是窗口的父窗口句柄，当窗口样式有 WS_CHILD 时需设备父窗口句柄。

参数 hInstance 是应用程序实例，设置为 WinMain 函数参数 hInstance。

lpParam 指定相关参数，指向 CREATESTRUCT 结构的变量，可以为 NULL。

2．窗口句柄

CreateWindow 和 CreateWindowEx 返回值都是 HWND 类型的。HWND 是窗口句柄数据类型。在 Windows 系统中每个窗口都有一个句柄，所有窗口句柄都是 HWND 数据类型的。

3．实例 9-8 创建窗口

WinMain 函数所调用的 InitInstance 函数实现了窗口的创建，代码如下：

```
/* ************************************
* BOOL InitInstance(HINSTANCE hinstance, int nCmdShow)
* 功能：创建主窗口和控件
* 参数：hinstance，应用程序本次运行实例
*       nCmdShow，如何显示
* 返回值：是否成功
**************************************/
BOOL InitInstance(HINSTANCE hinstance, int nCmdShow)
{
    HWND hwnd;
    RECT rect;
    // 保存应用程序实例句柄
    hinst = hinstance;
    hwnd = CreateWindow(
        "MainWClass",      // 窗口类名，使用之前注册的主窗口类
        "Let's Smile",     // 窗口名，显示在窗口标题栏上的字符串
        WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_CLIPSIBLINGS |
        WS_POPUPWINDOW,              // 窗口样式
        CW_USEDEFAULT,               // 水平位置（默认）
        CW_USEDEFAULT,               // 垂直位置（默认）
        800, // 宽
        600, // 高
        (HWND) NULL, // 无父窗口
        (HMENU) LoadMenu(hinst,MAKEINTRESOURCE(IDR_MENU_MAIN)), // 菜单
        hinstance, //应用程序实例
        (LPVOID) NULL); // 无窗口创建数据
    // 窗口是否创建成功
    if(!hwnd)
        return FALSE;
    // 保留窗口句柄
    hwndMain = hwnd;
    // 保证通用控件动态链接库已经加载
    InitCommonControls();
    // 创建 3 种控件，子窗口
    hwndTreeView = CreateTreeView(hwndMain, "files");
    hwndListView = CreateListView(hwndMain,"processing");
    hwndEdit = CreateEdit(hwndMain,"texta");
    // 获取本窗口客户区的 RECT（矩形方框的 4 个边界点）
    GetClientRect(hwndMain,&rect);
    // 设置子窗口的大小和位置
    SetWindows(&rect);
    // 在 EDIT 控件中显示文本
    ShowTextOnEdit(lpszLatin);
    // 显示主窗口，使用 WinMain 函数设置的窗口显示方式
    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);
    return TRUE;
}
```

在创建了主窗口后，还调用了 CreateTree View、CreateListView、CreateEdit 函数创建了 3 个子窗口，SetWindows 函数用于设置窗口的大小和位置。ShowTextOnEdit 设置了文本框窗口。这些函数的代码在后续实例中会陆续讲解。

9.3.3 窗口消息处理函数

消息处理函数是一个回调函数，是注册窗口类时指定（通过函数指针赋值给 WNDCLASS 结构或 WNDCLASSEX 结构的 lpfnWndProc 成员来指定消息处理函数），在有消息需要处理时由系统调用。

消息处理函数具有固定的接口形式：

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam
);
```

回调函数的函数名任意，但是调用类型、返回值类型、参数类型和顺序必须固定，以使系统可以正常调用消息处理函数。

参数 hwnd 是操作的窗口。uMsg 是消息标识符，窗口处理函数中需根据这个函数判断消息的类型做出不同的处理。wParam 和 lParam 是消息的两个参数，根据消息类型的不同而有不同的意义。

1．实例 9-9 窗口消息处理

本实例是 Windows.exe 应用程序的窗口消息处理函数，其中处理了 WM_SIZE、WM_DESTROY 和 WM_COMMAND 消息(Messages)和通知(Notifications)。

```
/* ************************************
* MainWndProc
* 功能：主窗口消息处理函数
**************************************/
LRESULT CALLBACK MainWndProc(
                                HWND hwnd,          // 窗口句柄
                                UINT uMsg,          // 消息标识符
                                WPARAM wParam,      // 消息的第一个参数
                                LPARAM lParam)      // 消息的第二个参数
{
    switch (uMsg)
    {
    case WM_CREATE:          // 窗口创建时收到此消息
         break;
    case WM_PAINT:           // 窗口被绘制时收到此消息
         break;
    case WM_SIZE:            // 窗口大小改变时收到此消息
         OnWindowResize();
         break;
         // NOTIFY，通常由控件发送给其父窗口，说明控件正在进行某项窗口操作
    case WM_NOTIFY:
         OnChildWindowsNotify((PVOID)lParam);
         break;
         // 窗口销毁，单击右上角的关闭按钮会触发此消息
    case WM_DESTROY:
         PostQuitMessage(0);
         break;
         // 命令输入，菜单项被选中，或按钮被单击
    case WM_COMMAND:
         {
              switch (LOWORD(wParam))
              {
                   // 在"帮助"菜单中选择"关于"
              case ID_HELP_ABOUT:
                   {
                        DialogBox(
                             hinst, (LPCTSTR)IDD_DIALOG_ABOUT, hwnd, (DLGPROC)About
                             );
                        return 0;
                   }
              default:
                   return DefWindowProc(hwnd, uMsg, wParam, lParam);
              }
         }
         // 可以在此处理其他消息
    default:
         break;
    }
    // 有很多消息未做处理，需要由默认窗口消息处理函数来处理
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

WM_SIZE、WM DESTROY 和 WM_COMMAND 这些常数都是消息的类型。Windows 系统预定义了很多种消息类型。表 9.2 列举了若干常用的消息和通知作为例子。

表 9.2　　常见的窗口消息

值	意　义
WM_CLOSE	收到此消息时窗口应该关闭
WM_CREATE	当窗口被创建时，会收到此消息
WM_DESTROY	当窗口从界面上销毁（用户单击窗口右上角关闭按钮）时会收到此消息（注意：此时进程不会自动退出）
WM_MOVING	当窗口被移动时，窗口会收到此消息
WM_SIZE	当窗口大小改变后，窗口会收到此消息
WM_NOTIFY	当窗口所包含的子窗口被操作时，父窗口会收到此消息
WM_COMMAND	当菜单、按钮被操作时，窗口会收到此消息

续表

值	意　义
WM_NCLBUTTONDOWN	当用户在窗口中单击左键时，窗口会收到此消息
WM_MOUSEHOVER	当鼠标在窗口客户区徘徊一段时间后窗口会收到此消息

Windows 系统中的消息类型远不止这些，还有很多种，在程序开发可根据情况灵活使用。

下面以 MainWndProc 函数对 WM_ NOTIFY 和 WM DESTROY 和 WM_COMMAND 消息的处理为例进行讲解。

2．重要消息

(1) WM NOTIFY。

当一个窗口的子窗口被用户操作时，该子窗口会向父窗口发送 WM NOTIFY 通知。WM__ NOTIFY 的 lParam 参数是一个指向 NMHDR 结构的指针。NMHDR 结构子窗口向父窗口所发送的通知中的各种信息如下所示：

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

其中 hwndFrom 表示向主窗口发送此消息的子窗口的句柄。idFrom 是窗口的标识符。code 表示是进行哪种操作。比如 OnChildWindowsNotify 函数所使用到的 NM_RCLICK 表示右键单击。此外常用的还有 NM_SETFOCUS（获得焦点）、NM_KEYDOWN（键盘按键）等。

比如在 Windows.exe 中 MainWndProc 函数在收到 WM NOflFY 消息后将 lParam 参数传递给了 OnChildWindowsNotify，由 OnChildWindowsNotify 函数做处理。OnChildWindowsNotify 函数如下：

```
/* ************************************
* BOOL OnChildWindowsNotify(PVOID pParam)
* 功能：处理控制子窗口向父窗口发送的 NOTIFY 消息
* 参数：pParam，WM_NOTIFY 消息参数
* 返回值：是否是 Tree View 发送的
************************************/
BOOL OnChildWindowsNotify(PVOID pParam)
{
    LPNMHDR phdr = (LPNMHDR)pParam;
    // 只处理 Tree View 发送的 Notify，其他不处理
    if(phdr->hwndFrom != hwndTreeView)
    {
        return FALSE;
    }
    switch(((LPNMHDR)pParam)->code)
    {
    // 如果是右键单击，则调用 OnRclickTree 函数，弹出右键菜单
    case NM_RCLICK :
        OnRclickTree((LPNMHDR)pParam);
        break;
    default:
        break;
    }
    return TRUE;
}
```

OnChildWindowsNotify 在判断了如果被操作的是 hwndTreeView 窗口，而且是右键单击，那么会调用 OnRclickTree。OnRclickTree 会在指定位置弹出右键菜单。实例 9-13 将介绍 OnRclickTree 函数。

(2) WMDESTROY。

收到 WM—DESTROY 消息，表示用户单击窗口的关闭按钮。如果用户单击主窗口的关闭按键，一般情况下表示用户需要结束本程序的运行。因此在 MainWndProc 函数收到了 WM_DESTROY 消息后，调用 PostQuitMessage 函数退出程序运行。要退出一个进程的运行，也可以调用 ExitProcess 函数。

(3) WM_COMMAND。

收到 WM_COMMAND -般是因为用户通过菜单、按钮等用程序输入命令。WM_COMMAND 消息的 wP aram 的低位 WORD 是表示用户单击的控件或菜单项的 ID（参见资源介绍）。

在 MainWindowProc 函数中只处理了一个命令-ID_HELP_ABOUT。这个命令对应了程序窗口菜单中“帮助＼关于”选项。在收到命令后，消息处理函数调用 API 函数 DialogBox 弹出了如图 9-12 所示的对话框。

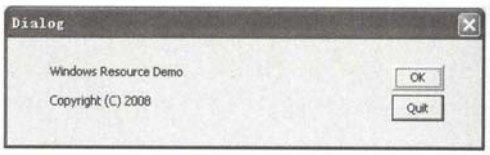

图 9-12 对话框

3. DefWindowProc

DefWindowProc 是一个 API 函数。

窗口会收到很多消息，但是这些消息消息处理函数不一定需要处理，有些消息可能直接让系统来处理。DefWindowProc 的作用是处理窗口消息处理函数没有处理完、或者不需要处理的函数。

实际上很多程序员没有关注的消息都是由 DefWindowProc 来处理的。如果在消息处理函数中不把消息交给 DefWindowProc 处理，那么对窗口的动作几乎不会有响应，包括移动窗口、关闭窗口等，甚至有很多窗口样式不能显示。

一般情况下，除非程序员确定一个消息不再需要 DefWindowProc 来处理，那么最好将不

需要的消息统一交由 DefWindowProc 函数来处理。

9.3.4 窗口属性、位置和大小

窗口属性、位置和大小是窗口设计中重要的内容。实例 9-10 使用 API 函数来设置窗口的位置和大小，使父窗口及各子窗口的大小的位置合理。

1．实例 9-10 设置窗口位置和大小

InitInstance 函数除了完成父窗口的创建外，还创建了 3 个子窗口。在创建子窗口完成后，调用了 SetWindows 函数。SetWindows 函数的代码如下：

```
/* ************************************
* DWORD SetWindows(LPRECT lpRect)
* 功能：本程序中创建了 3 个控件，有 3 个子窗口
*          本函数设置子窗口的位置和大小
* 参数：lpRect，指向表示父窗口客户区的 RECT
************************************/
DWORD SetWindows(LPRECT lpRect)
{
    // Tree View
    SetWindowPos(hwndTreeView, HWND_TOP,
         lpRect->left, lpRect->top,
         lpRect->right *0.3 ,lpRect->bottom,
         SWP_SHOWWINDOW);
    // List View
    SetWindowPos(hwndListView, HWND_TOP,
         lpRect->right * 0.3, lpRect->bottom * 0.7,
         lpRect->right * 0.7, lpRect->bottom * 0.3,
         SWP_SHOWWINDOW);
    // Edit
    SetWindowPos(hwndEdit, HWND_TOP,
         lpRect->right * 0.3, lpRect->top,
         lpRect->right * 0.7, lpRect->bottom * 0.7,
         SWP_SHOWWINDOW);
    return 0;
}
```

SetWindows 函数完成了 3 个控件子窗口的位置的大小设置，调用了 SetWindowPos API 函数。SetWindowPos 原型如下：

```
BOOL SetWindowPos(
    HWND hWnd,
    HWND hWndInsertAfter,
    int X,
    int Y,
    int cx,
    int cy,
    UINT uFlags
);
```

其中 hWnd 是所需设置窗口的位置和大小的句柄。hWndInsertAfter 参数是说明窗口的层次顺序，在本实例中都设定为 HWND_TOP，表示显示在最上方。

X、Y、cx、cy 这 4 个参数分别是窗口的左上角所在的位置和窗口的宽及高。

uFlags 是窗口大小和位置标志，实例中设置为 SWP_SHOWWINDOW 表示窗口正常显示。

2．可设置的窗口属性

不仅是窗口的位置和大小可以设置，窗口几乎所有属性都可以设置，包括创建窗口中所指定的窗口样子，甚至窗口的消息处理函数也可以被替换。

SetWindowLong 函数可实现窗口的多种属性设置。

```
LONG SetWindowLong(
    HWND hWnd,
    int nIndex,
    LONG dwNewLong
);
```

hWnd 参数指明了需要设置属性的窗口，nIndex 指明需要设置哪种属性。dwNewLong 是新的属性。

表 9.3 所列举了最常用的、可作为 nIndex 的参数值的常数。

表 9.3　　SetWindowLong 的参数

属 性 常 量	意　义
GWL_STYLE	窗口样式
GWL_WNDPROC	窗口消息处理函数
GWL_HINSTANCE	应用程序实例

9.3.5　窗口显示方式

另一个值得一提的是窗口的显示方式。创建好的窗口可以在屏幕上显示，也可以不显示。方式有很多种，通过 ShowWindow 函数设置。

```
ShowWindow(hwnd, nCmdShow);
```

使用 CreateProcess 在创建进程时就可以指定应用程序窗口的显示方式(dwCreationFlags 参数设置为 CREATE NO__ WINDOW)。如果 CreateProcess 函数设置了 CREATE__ NO_WINDOW 标志，将会影
响被创建程序的 WinMain 函数的 nCmdShow 参数，nCmdShow 默认参数是 SW_HIDE，可将其设置为 SW_SHOW。

一般情况下，WinMain 函数的 nCmdShow 参数最后会传递给应用程序在创建窗口后调用的 ShowWindow 函数。

此外还有 SW_MAXIMIZE、SW_MINIMIZE、SW_SHOWDEFAULT 等多种设置。

9.3.6. 线程消息队列和消息循环

> **注意**　GetMessage 函数只是获取了其所在的线程的消息池中的消息，如果不注意这一点，通常会在多线程的图形用户界面开发中出现令人迷惑的情况。一个经验性的建议是，在开发多线程的图形用户界面程序时，将所有的窗口创建放置在同一个线程中，子窗口的创建最好在主窗口的消息处理函数中，这样能保存消息都集中在同一个线程。如果在其他线程中需要操作界面，可能直接使用 PostThreadMessage、SendMessage 等 API 函数向窗口发送自定义的消息，然后再在窗口消息处理函数中处理这些自定义的窗口消息时进行窗口操作。

在 9.3 节一开始给出的 WinMain 函数代码中，函数的最后部分是消息循环。循环中调用了 GetMessage、TranslateMessage 和 DispatchMessage 三个 API 函数。

GetMessage 是从其所在的线程的消息队列中得到一条消息。在 TranslateMessage API 函数对消息进行转换后，DispatchMessage 将消息发送到消息处理函数，通常用于分配

GetMessage 函数获得的消息。

Windows 系统为每一个线程创建消息队列，从消息队列中获取消息使用 GetMessage 函数。也可以调用 API 函数将消息添加到消息队列中。PostMessage 与 SendMessage 不同的是，PostMessage 只是将消息放置在线程的消息队列之后便立即返回，而 SendMessage 是直接将消息交给窗口消息处理函数。PostQuitMessage 函数将 WM_QUIT 消息放置在消息队列中，GetMessage 函数在获得了 WM_QUIT 消息后，会返回 0，退出消息循环。

9.4 图形用户界面：控件

控件实际是一种特殊的窗口。每一种控件有固定的样式、消息处理过程等，系统将这此属性做了固定的模式化的处理，并提前注册为窗口类。

这些控件在程序设计时不用再重新注册，也不用再去定义其消息处理函数、界面样式、窗口大小等，可以直接使用。

本节将以 Tree View、List View、文件本框 3 种形式的控件为例说明控件使用的方法。

> 注意　在程序中使用控件之前，最好先调用 InitCommonControls API 函数以保证调用控件所需使用的动态链接库被加载，如实例 9-8 所示。

9.4.1 Tree View控件

因为控件也是窗口，所以创建控件也是使用 CreateWindow 或 CreateWindowEx 函数，在调用创建窗口 API 时需要指定窗口类。每个控件有特定的窗口类。

创建 Tree View 控件使用 WC_TREEVIEW 类，WC_TREEVIEW 是一个字符串常量，实际值为“SysTreeView32”。这个窗口类已经由系统预先定义。

在使用控件时，新创建的控件必须有一个父窗口。

图 9-11 的窗口的左侧是一个 Tree View 控件。

实例 9-11　创建 Tree View 控件

在实例 9-8 中调用了 CreateTreeView 函数，CreateTreeView 函数实现了创建一个 Tree View 控件，代码如下：

```
/* ************************************
* HWND CreateTreeView(HWND hwndParent, LPSTR szWindowName)
* 功能：创建一个 Tree View 控件，并调用 InitTreeViewItems 增加若干个节点
* 参数：hwndParent，父窗口句柄
*       szWindowName，窗口标题
* 返回值：创建的窗口句柄
************************************/
HWND CreateTreeView(HWND hwndParent, LPSTR szWindowName)
{
    HWND hwndTV;
    // 创建 Tree View
    hwndTV = CreateWindowEx(0,
        WC_TREEVIEW,          // Tree View 控制窗口类
        szWindowName,       // 窗口的标题
        // 窗口样式：可见，子窗口，可改变大小，具体窗口标题
        WS_VISIBLE | WS_CHILD | WS_SIZEBOX |  WS_TILED |
        // 附加 Tree View 样式
        TVS_HASBUTTONS |TVS_LINESATROOT ,
        // 默认大小和位置，后面使用 SetWindows 函数设置
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        hwndParent,             // 父窗口句柄
        (HMENU)NULL,            // 没有菜单
        hinst,                  // 应用程序实例
        NULL);                  // 没有图标

    // 初始化 Image List 和节点
    if(!InitTreeViewImageLists(hwndTV) ||
        !InitTreeViewItems(hwndTV))
    {
        DestroyWindow(hwndTV);
        return NULL;
    }
    return hwndTV;
}
```

所调用的 CreateWindowEx API 函数的其他参数设置都与一般的窗口创建没有区别，只是并没有注册窗口类，而是在窗口类名参数上设置了一个系统预定义的字符串常量 WC_TREEVIEW。如前所述，控件的窗口类已经由系统事件定义好了，在程序中使用控件时，只需要直接使用这此窗口类即可。

CreateTreeView 函数还调用了两个函数 InitTreeViewImageLists 和 InitTreeViewItems。这 InitTreeViewImageLists 函数的功能是为控件初始化了 Image List。

InitTreeViewImageLists 是向 Tree View 增加 Image List。所谓 Image List 可以理解为控件可以使用的一组图标和图像。如果程序需要在每个节点上设置节点图标，那么需要为 Tree View 控件增加 Image List。控件所显示的图标位于项名(Text)的左侧。

在本实例中，没有使用 Image Lists，但是已经把基本的框架保留出来，如果在程序中需要使用到 Image List，读者可根据下面所示的顺序进行设置。

```
// 未初始化 Tree View 的 ImageList，因此节点将没有图标
// 虽然在插入节点时指定了节点的 Image，但是由于 ImageList 为空，因此图标不能显示
BOOL InitTreeViewImageLists(HWND hwndTV)
{
    // TODO 创建 ImageLists
    // TODO 设置 ImageLists 项
    // TODO 使用 TreeView_SetImageList 设置 Tree View 的 Image List
    return TRUE;
}
```

InitTreeViewItems 函数的功能向是实例中增加了一些节点，包括树的根节点和子节

点。
InitTreeViewItems 函数的实现代码如实例 9-12 所示。

9.4.2 为Tree View控件增加节点

如图 9-11 所示，Window.exe 程序的 Tree View 控件在创建后就添加了若干的节点，包括 3 个根节点 type1、type2 和 type3，前两个根节点有子节点 name1、name2 和 name3，name2 和 name3 还具有子节点。

实例 9-12 为 Tree View 控件增加节点

InitTreeViewItems 为控件添加了若干个节点，函数原型如下：

```
/* ************************************
* BOOL InitTreeViewItems(HWND hwndTV)
* 功能：为 Tree View 窗口增加若干个节点
* 参数：hwndTV，Tree View 控件窗口句柄
************************************/
BOOL InitTreeViewItems(HWND hwndTV)
{
    // 根节点 1
    HTREEITEM hRoot = AddItemToTree(hwndTV, "type1", NULL,TRUE);
    // 根节点 1 的子节点 1
    HTREEITEM hP = AddItemToTree(hwndTV, "name1", hRoot,FALSE);
    // 根节点 1 的子节点 2
    hP = AddItemToTree(hwndTV, "name2", hRoot, TRUE);
    // 子节点 2 的子节点
    AddItemToTree(hwndTV, "sub1", hP,FALSE);
    // 子节点 2 的子节点
    AddItemToTree(hwndTV, "sub2", hP,FALSE);
    // 根节点 2
    hRoot = AddItemToTree(hwndTV, "type2", NULL,TRUE);
    // 根节点 2 的子节点
    hP = AddItemToTree(hwndTV, "name3", hRoot, TRUE);
    // 根节点 2 的子节点
    hP = AddItemToTree(hwndTV, "sub3", hP,FALSE);
    // 根节点 3
    hRoot = AddItemToTree(hwndTV, "type3", NULL,TRUE);

    return TRUE;
}
```

AddItemToTree 函数增加一个节点。

```
/* ************************************
* AddItemToTree
```

```
* 功能：为 Tree View 控件增加节点
* 参数：hwndTV，Tree View 控件窗口句柄
*               lpszItem，节点名（文本）
*               hParent，父节点句柄，如果为 NULL，则增加根节点
*               bFolder，是否具有子节点（影响节点图标 ICON）
* 返回值：创建的节点句柄
************************************/
HTREEITEM AddItemToTree(
                        HWND hwndTV,
                        LPSTR lpszItem,
                        HTREEITEM hParent,
                        BOOL bFolder)
{
    TVITEM tvi;
    TVITEM tParent;
    TVINSERTSTRUCT tvins;
    HTREEITEM hme;
    // 填充 TVITEM 结构
    // 通过 TVITEM 的 mask 字段，说明那此字段是有效的
    tvi.mask = TVIF_TEXT | TVIF_IMAGE | TVIF_SELECTEDIMAGE | TVIF_PARAM;

    // 字节的文本及文本长度
    tvi.pszText = lpszItem;
    tvi.cchTextMax = sizeof(tvi.pszText)/sizeof(tvi.pszText[0]);

    // 根据 bFolder 参数为节点指定不同的图标
    if(bFolder)
    {
        tvi.iImage = 1;
        tvi.iSelectedImage = 2;
        tvi.cChildren = I_CHILDRENCALLBACK;
    }
    else
    {
        tvi.iImage = 3;
        tvi.iSelectedImage = 3;
        tvi.cChildren = 0;
    }
    // 填充 TVINSERTSTRUCT 结构
    tvins.item = tvi;
    // 新插入的子节点的位置，TVI_SORT 表示按文本排序
    // 还可以是 TVI_FIRST TVI_LAST 等
    tvins.hInsertAfter = TVI_SORT;
    // 如果 hParent 为 NULL，则插入的节点为根节点
    // 否则 hParent 为其父节点
    if(hParent == NULL)
    {
        tvins.hParent = TVI_ROOT;
    }
    else
    {
        tvins.hParent = hParent;
    }
    // 调用 TreeView_InsertItem 宏，插入新节点
    hme = TreeView_InsertItem(hwndTV,&tvins);
    return hme;
}
```

TreeView InsertItem 宏是实现增加节点的 API。在增加节点前，需填充 TVINSERTSTRUCT 结构，指明控件的句柄、父节点句柄、显示的文字、被选择和未被选择里的图标(Image List 的索引)。

9.4.3 Tree View右键菜单

如图 9-11 所示，Windows.exe 还实现了 Tree View 控制的右键菜单功能。所有的子窗口

在被用户操作后，会向主窗口发送 WM NOTLFY 通知。主窗口在收到子窗口的通知后调用了 OnChildWindowsNotify 函数来处理 WM NOTIFY。

用户在 Tree View 控件中单击右键后，Tree View 也会向主窗口发送 WM NOTIFY。在 OnChildWindowsNotify 中通过消息参数判断了 WM_ NOTIFY 的参数，如果是在 Tree View 窗口中单击右键的情况，那么调用 OnRclickTree 作出处理。

实例 9-13　Tree View 右键菜单

9.3.3 节在处理 Tree View 向父窗口发送的 WM NOTIFY 消息时，所提到的 OnRclickTree 函数弹出了 Tree View 控件的右键菜单，函数如下：

```
/* ************************************
* BOOL OnRclickTree(NMHDR* pNMHDR)
* 功能：在窗口被右键单击后调用，用于显示右键菜单
*       判断是否在 Tree View 节点上单击，并计算鼠标位置
* 参数：pNMHDR，WM_NOTITY 消息所带的 lParam 参量
* 返回值：是否单击了节点
************************************/
BOOL OnRclickTree(NMHDR* pNMHDR)
{
    POINT point;
    TVHITTESTINFO thti;
    HTREEITEM htItem;

    // 获取鼠标的位置
    GetCursorPos(&point);

    // 计算相对 Client 的鼠标位置，
    // TreeView_HitTest 要求的鼠标位置是相对于 Client
    ScreenToClient(hwndTreeView,&point);
    // 填充 TVHITTESTINFO 结构
    thti.pt = point;
    thti.flags = TVHT_TORIGHT;

    // TreeView_HitTest 测试是否单击的子节点，获取被单击节点的句柄
    htItem =TreeView_HitTest(hwndTreeView,&thti);

    // 也可以根据节点属性的不同来弹出不同的右键菜单
    // TreeView_GetItem 等函数可以用于获取 TVITEM 结构中的各个字段
    // 用户可以设置 lParam 参量来保存与节点相关的附加信息
    if(htItem!=NULL)
    {
        // DisplayContextMenu 函数要求的鼠标位置是相对于屏幕的
        ClientToScreen(hwndTreeView,&point);
        // 选中用户右键单击的节点，否则 Tree View 中被选中的节点不会改变
        TreeView_SelectItem(hwndTreeView,htItem);
        // 调用 DisplayContextMenu 函数显示右键菜单
        DisplayContextMenu(hwndMain, point);
        return TRUE;
    }
    return FALSE;
}
```

函数首先获取鼠标位置，通过鼠标位置判断用户单击的是哪个节点，判断的目的是要选中这个节点，用户右键单击这个节点后，节点并不会自动被选中，而是需要使用 TreeView_SelectItem 来选中。在程序设计时也可以根据 Tree View 控件中节点的不同属性来弹出不同的菜单。获取节点属性使用 TreeView_GetItem，会返回 TVITEM 类型的结构。

之后调用 DisplayContextMenu 函数在窗口中的指定位置弹出的菜单，DisplayContextMenu 函数代码如下：

```
/* ************************************
* VOID APIENTRY DisplayContextMenu(HWND hwnd, POINT pt)
* 功能：在屏幕上显示弹出菜单
* 参数：hwnd，菜单所属窗口
*       pt，菜单位置（相对于屏幕）
************************************/
VOID APIENTRY DisplayContextMenu(HWND hwnd, POINT pt)
{
    HMENU hmenu; // 顶级菜单
    HMENU hmenuTrackPopup; // 弹出菜单

    // 加载菜单资源，获得菜单句柄
    if((hmenu = LoadMenu(hinst, MAKEINTRESOURCE(IDR_MENU_POPUP))) == NULL)
        return;

    // 获得菜单资源中的子菜单，所显示的是菜单资源中的子菜单
    hmenuTrackPopup = GetSubMenu(hmenu, 0);
    // 弹出
    TrackPopupMenuEx(hmenuTrackPopup,
        TPM_LEFTALIGN | TPM_RIGHTBUTTON,
        pt.x, pt.y, hwnd, NULL);
    // 完成，释放相关资源
    DestroyMenu(hmenu);
}
```

DisplayContextMenu 函数使用一系列 API 函数，首先调用 LoadMenu 从资源中获取了菜单，得到菜单句柄。然后调用 GetSubMenu 获得了第一级子菜单。TrackPopupMenuEx 函数的功能是将菜单显示在界面上，调用时，需指定的需显示的菜单句柄、位置所属窗口等。

在完成菜单处理后，使用 DestroyMenu 释放时关的资源。

右键菜单的运行效果如图 9-13 所示。

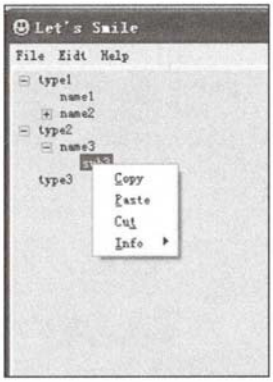

图 9-13　Tree Viewr 控件右键菜单

如果用户选中了某个菜单项，那么主窗口消息处理函数在可以收到 WM_COMMAND 消息，并通过 wParam 获得项 ID。实例中未处理这个菜单的菜单项输入，读者可根据需要自行修改。

9.4.4　List View控件

在图 9-11 中位于界面左下方的就是一个 List View 控件。

实例中的 CreateListView 创建了一个 List View，创建时，需使用 WC_LISTVIEW 窗口类。

图 9-14 所示 Windows 资源管理器、Explorer.exe 是一个典型的 List View 的应用。List View 有很多样式，包括“详细信息”、“图标”、“列表”等。在窗口样式参数中说明需要使用的 List View 的样式如果为 LVS_REPORT，则是“详细信息”；如果是 LVS_ICON、LVS_SMALLICON、LVS_LIST，则分别是“大图标”、“小图标”、“列表”几种显示方式。

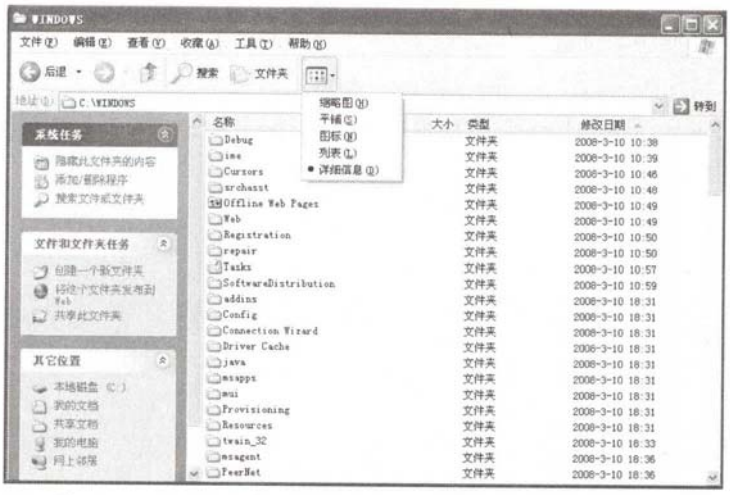

图 9-14　Windows 桌面程序对 List Vist 控件显示样式的修改

LVM_SETVIEW 消息可以设置 List View 的显示样式。

实例 9-14 创建 List View

本实例中使用的是 LVS_REPORT，说明是采用“详细列表”的方式。

```
/* ************************************
* HWND CreateListView (HWND hwndParent, LPSTR szWindowName)
* 功能：创建 List View
* 参数：hwndParent，父窗口
*       szWindowName，窗口标题
* 返回值：窗口句柄
**************************************/
HWND CreateListView (HWND hwndParent, LPSTR szWindowName)
{
    HWND hWndListView;
    // 创建 List View
    hWndListView = CreateWindow (
         WC_LISTVIEW, // List View 窗口类
         // 窗口标题，由于未指定 WS_TILED ，这是一个无标题的窗口
         szWindowName,
         // 窗口样式（可视，子窗口）和 List View 控件样式
         WS_VISIBLE | WS_CHILD | LVS_REPORT | LVS_EDITLABELS,
         // 位置和大小，创建完成后使用 SetWindows 设置
         CW_USEDEFAULT,
         CW_USEDEFAULT,
         CW_USEDEFAULT,
         CW_USEDEFAULT,
         hwndParent,          // 父窗口
         NULL, hinst, NULL);      // 无菜单，无图标
    if(hWndListView == NULL)
    {
         return NULL;
    }
    // 初始化 Image List，初始化列，加入一些项
    if(InitListViewImageLists(hWndListView) &&
         InitListViewColumns(hWndListView) &&
         AddListViewItems(hWndListView))
    {
         return hWndListView;
    }
    DestroyWindow(hWndListView);
    return FALSE;
}
```

InitListViewImageLists 的功能是初始化 Image List。

每一个项都可以有一个图标。和 Tree View 控件类似，如果一个 List View 控件需要使用图标，那么也需要使用 Image List。

```
/* ************************************
* BOOL InitListViewImageLists(HWND hWndListView)
* 功能: 设置 List View 的 ImageList
* 参数: hWndListView, List View 控件窗口
************************************/
BOOL InitListViewImageLists(HWND hWndListView)
{
    HIMAGELIST himl;
    HBITMAP hbmp;
    // 调用 GetImageList 获得 ImageList
    GetImageList(&himl);
    // 设置 List View 的 ImageList
    ListView_SetImageList(hWndListView, himl, LVSIL_SMALL);
    return TRUE;
}
/* ************************************
* BOOL GetImageList(HIMAGELIST * pHiml)
* 功能: TODO 增加 ImageList 项
* 参数: pHiml, Image List
************************************/
BOOL GetImageList(HIMAGELIST * pHiml)
{
    return TRUE;
}
```

这是一个空的 Image List，因此在增加项时指定的 Image List 索引是无意义的。

9.4.5 为List View控件增加分栏

在 LVS_REPORT 样式时，每一项都分栏显示。

实例 9-14 调用的函数 InitListViewColumns 完成了分栏设置。

实例 9-15 为 List View 增加分栏

ListView_InsertColumn 宏可以完成插入分栏项的功能，参数为 LVCOLUMN 类型的结构。

```
/* ************************************
* BOOL InitListViewColumns(HWND hWndListView)
* 功能; 设置 List View 的分栏
* 参数; hWndListView, List View 控件窗口
* 返回值: 是否成功
************************************/
BOOL InitListViewColumns(HWND hWndListView)
{
    char szText[256];
    LVCOLUMN lvc;
    DWORD i;
    // 分栏标题
    LPSTR ColNames[] = {"name", "type", "size", "time"};
    // LVCOLUMN 中有效的成员
    lvc.mask = LVCF_FMT | LVCF_WIDTH | LVCF_TEXT | LVCF_SUBITEM;
    // 填充 LVCOLUMN 结构
    lvc.pszText = szText;
    lvc.cx = 100;                // 长度
    lvc.iImage = 1;
```

```
    lvc.fmt = LVCFMT_LEFT;      // 向左对齐
    // 前三个
    for(i=0; i<3; i++)
    {
        // 设置栏的文本
        lvc.pszText = ColNames[i];
        lvc.iSubItem = i;
        // 调用 ListView_InsertColumn 插入分栏
        if(ListView_InsertColumn(hWndListView, i,      &lvc) == -1)
        {
            return FALSE;
        }
    }
    // 最后一个分栏
    lvc.cx = 200; // 长度
    lvc.fmt = LVCFMT_RIGHT; // 右对齐
    lvc.iSubItem = 3;
    lvc.pszText = ColNames[3];
    // 插入分栏
    if(ListView_InsertColumn(hWndListView, 3,      &lvc) == -1)
    {
        return FALSE;
    }
    return TRUE;
}
```

本函数中，一共增加了 4 个分栏项，参见图 9-15，前 3 个宽度都是 100，文字向左对齐，显示的文字分别是“name”、“type”和“size”；第 4 个分栏项宽度是 200，文字向右对齐，显示的文字是“time”。

图 9-15　List Visw 的表项

mask 成员指明了结构中哪些成员是有效的。

9.4.6　为List View控件增加项

AddListViewItems 函数实现了为 List View 控件增加项的功能。ListVIew 在 LVS_REPORT 样式下，一行是一项，一行可以有若干个子项，每个子项对应一个分栏。

实例 9-16 为 List View 增加分栏

为 List View 增加项使用 ListView InsertItem 宏，参数为 LVLTEM 类型的结构。AddListViewItems 函数实现了相关函数，代码如下：

```
/* ************************************
* BOOL AddListViewItems(HWND hwndListView)
* 功能：为 List View 增加项
* 参数：hwndListView，父窗口
* 返回值：窗口句柄
************************************/
BOOL AddListViewItems(HWND hwndListView)
{
    LVITEM lvI;
    DWORD index;
    ZeroMemory (&lvI,sizeof(lvI));
    // 有效的项
    lvI.mask = LVIF_TEXT | LVIF_IMAGE | LVIF_PARAM | LVIF_STATE;
    // 填充 LVITEM
    lvI.state = 0;
```

```
    lvI.stateMask = 0;
    // 循环增加
    for(index = 0; index < 3; index++)
    {
        lvI.iItem = index;
        lvI.iImage = 0;
        lvI.iSubItem = 0;
        // 项的文本和长度
        lvI.pszText = "name1";
        lvI.cchTextMax = lstrlen(lvI.pszText)+1;
        // 插入项
        if(ListView_InsertItem(hwndListView, &lvI) == -1)
            return FALSE;
        // 设置子项的文本，0 based
        ListView_SetItemText(hwndListView, index, 2, "size a");
    }
    return TRUE;
}
```

如图 9-16 所示，一共插入了 3 个项，每个项的项名（第 1 个子项名）为“name1”、并设置了第 3 个子项显示的文本为“size a”。

name	type	size	time
name1		size a	
name1		size a	
name1		size a	

图 9-16　List Vist 运行效果

同样，LVITEM 结构的 mask 成员说明了其他哪些成员是有效的。

9.4.7　文本框控件

如图 9-11 所示，Windows.exe 屏幕中部到右上的位置是一个文本框控件。文本框控件具有基本的文本编辑功能。创建文件框控件所使用的控件窗口类为“EDIT”。

实例 9-17 创建文本框

CreateEdit 函数实现了文本框控件的创建，代码如下：

```
/* ************************************
* HWND CreateEdit(HWND hwndP, LPSTR szTitle)
* 功能：创建文本编辑控件
* 参数：hwndP，父窗口句柄
*       szTitle，窗口标题
* 返回值：窗口句柄，失败返回 NULL
************************************/
HWND CreateEdit(HWND hwndP, LPSTR szTitle)
{
    WNDCLASSEX wcx;
    HWND hwnd;

    hwnd = CreateWindow(
        "EDIT", // 文本编译控件
        NULL, // 没有标题
        // 窗口样式
        WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_BORDER |
        // 文本框样式
        ES_LEFT | ES_MULTILINE | ES_AUTOVSCROLL,
```

```
            // 位置和大小
            CW_USEDEFAULT ,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            hwndP, // 父窗口
            (HMENU) NULL, // 无菜单
            hinst, // 实例句柄
            (LPVOID) NULL); // 无图标

        if(!hwnd)
        {
            return NULL;
        }
        ShowWindow(hwnd, SW_SHOW);
        UpdateWindow(hwnd);
        return hwnd;
    }
```

文本框也有很多样式。通常包括是多行（ES_MULTILINE）还是单行、是否是密码框(ES_PASSWORD)、是否只能输入数字(ES_NUMBER)、是否将全大写、全小写(ES_UPPERCASE、ES_LOWERCASE)、文字的对齐方式(ES_CENTER、ES_LEFT、ES_RIGHT)、是否自动换行(ES_AUTOVSCROLL)等。

文本框还支持一些基本的文本操作，包括复制、剪切、粘贴等，如图 9-17 所示。

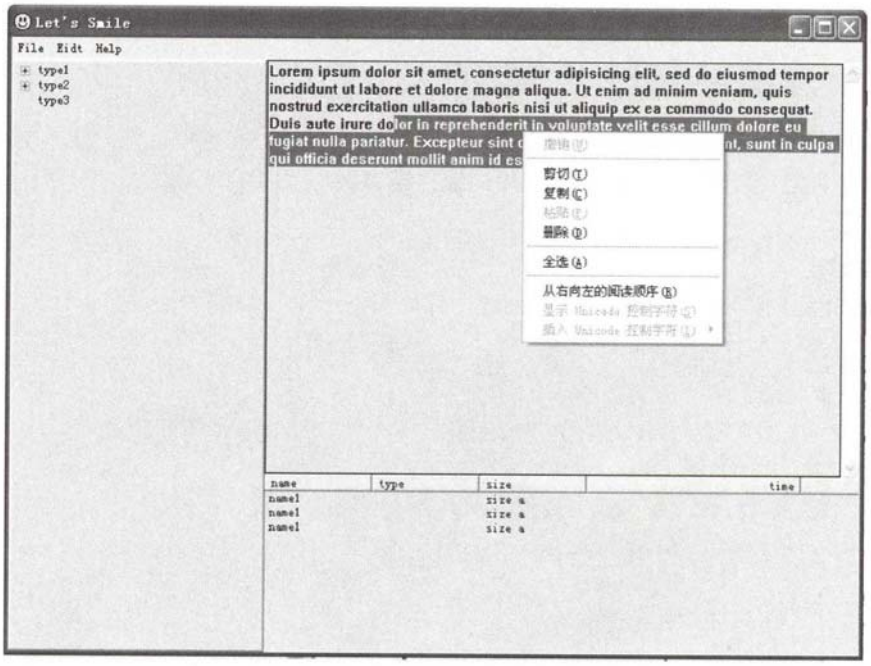

图 9-17　文件框控件及默认右键菜单

9.4.8　为文本框控件设置文字

在创建完文件框控件后可以使用消息来设置和获取文件框中的文字。WM_SETTEXT 用于设置文字。

实例 9-18 设置文本框控件的文字

ShowTextOnEdit 函数实现了设置文本框控件的文件，函数原型如下：

```
/* ************************************
* HWND ShowTextOnEdit(HWND hwndP, LPSTR szTitle)
* 功能：向文本框发送消息，设置文本
* 参数：szTextToShow，需要显示的文本
**************************************/
DWORD ShowTextOnEdit(LPSTR szTextToShow)
{
    // 向文本框发送 WM_SETTEXT 消息，设置窗口的文本
    SendMessage(hwndEdit, WM_SETTEXT, 0, (LPARAM) szTextToShow);
    return 0;
}
// 在界面上显示的文本
LPSTR lpszLatin =  "Lorem ipsum dolor sit amet, consectetur "
         "adipisicing elit, sed do eiusmod tempor "
         "incididunt ut labore et dolore magna "
         "aliqua. Ut enim ad minim veniam, quis "
         "nostrud exercitation ullamco laboris nisi "
         "ut aliquip ex ea commodo consequat. Duis "
         "aute irure dolor in reprehenderit in "
         "voluptate velit esse cillum dolore eu "
         "fugiat nulla pariatur. Excepteur sint "
         "occaecat cupidatat non proident, sunt "
         "in culpa qui officia deserunt mollit "
         "anim id est laborum.";
```

设置文字后，EDIT 控件如图 9-18 所示。

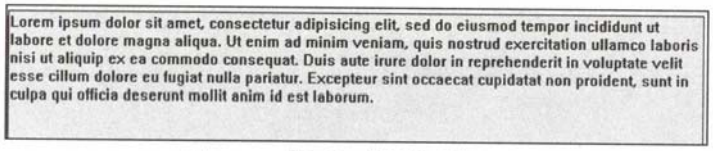

图 9-18　设置文本框控制的文字

9.5　界面资源

资源在应用程序开发中具有重要的作用。尤其是将界面元素（菜单、图标、位图等）作为资源，对降低程序设计的工作量大有帮助。

本节将介绍在 9.3 节 Windows.exe 中使用的资源在程序设计时是如何配置的。

9.5.1　资源脚本(．rc)

资源是在资源脚本(.rc)中进行配置的。实例 9-18 是 Window.exe 程序的资源脚本文件。

1．实例 9-19 资源脚本文件 Window.rc

Window.rc 文件内容如下：

```
#include <Windows.h>
#include "resource.h"

#pragma code_page(936)

//////////////////////////////////////////////////////////////////////////////
//
// Dialog
```

```
//

IDD_DIALOG_ABOUT DIALOGEX 0, 0, 290, 59          //DIALOGEX 资源
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Dialog"          // CAPTION 声明
FONT 8, "MS Shell Dlg", 400, 0, 0x1          // FONT 声明
BEGIN
    LTEXT          "Windows Resource Demo",3001,26,14,119,8,SS_NOPREFIX     //控件
    LTEXT          "Copyright (C) 2008",3002,26,30,119,8                    //控件
    DEFPUSHBUTTON   "OK",IDOK,234,14,32,14,WS_GROUP                         //控件
    DEFPUSHBUTTON   "Quit",IDCANCEL,234,31,32,14,WS_GROUP                   // 控件
END

/////////////////////////////////////////////////////////////////////////////
//
// Menu
//

IDR_MENU_MAIN MENU          //MENU 资源
BEGIN
    POPUP "File"               // POPUP 资源
    BEGIN
        MENUITEM "&Open",           ID_FILE_COPY          // POPUP 资源项 MENUITEM 声明
        MENUITEM "&Save",           ID_FILE_SAVE40002     // POPUP 资源项 MENUITEM 声明
        MENUITEM SEPARATOR                                // POPUP 资源项 MENUITEM 声明
        MENUITEM "E&xit",            ID_FILE_EXIT         // POPUP 资源项 MENUITEM 声明
    END
    POPUP "Eidt"                                          // POPUP 资源
    BEGIN
        MENUITEM "&Copy",           ID_EIDT_COPY          // POPUP 资源项 MENUITEM 声明
        MENUITEM "&Paste",       ID_EIDT_PASTE            // POPUP 资源项 MENUITEM 声明
    END
    POPUP "Help"                                          // POPUP 资源
    BEGIN
        MENUITEM "&About",      ID_HELP_ABOUT             // POPUP 资源项 MENUITEM 声明
    END
END

IDR_MENU_POPUP MENU
BEGIN
    POPUP "POPUP"
    BEGIN
        MENUITEM "&Copy",          ID_POPUP_COPY
        MENUITEM "&Paste",         ID_POPUP_PASTE
        MENUITEM "Cu&t",           ID_POPUP_CUT
        POPUP "&Info"
        BEGIN
            MENUITEM "Co&unt",     ID_INFO_A
            MENUITEM "&Size",      ID_INFO_B
        END
    END
END


/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

IDI_ICON_MAIN               ICON                "res\\icon.ico"  //ICON 资源
```

下面具体解释资源脚本的意义。

2．rc 文件的语法

资源文件包括注释、预处理指令、编译指令和资源定义声明等。

注释的语法规则与 C 语言类型，分单行注释和多行注释，分别使用“//”和“* */”。

预处理指令包括一些头文件引用（使用#include，与 C 语言类似）、预定义(#define，常量和宏，不支持带参数的宏)和条件编译(#if #ifdef #undef #ifndef #else、#elif. #endif 等)。

实例 9-19 的编译指令指明使用“#pragma code Page”用于指定文本的编码方式。

```
#pragma code_page( [ DEFAULT | CodePageNum ] )
```

CodePageNum 为 936，表示编码方式是简体中文。

资源定义声明是资源脚本中最重要的一项内容。所有的资源都在资源定义声明中说明。

3．资源定义声明

资源定义的声明中包括“资源”(Resources)、“控件”(Controls)和“声明”(Statements)3 类。

(1)资源。

“资源”具有资源 ID，在程序中可以使用 FindResource、LoadResource、LoadMenu 等资源相关 API 来操作。有的资源可以支持子控件，各种资源的声明各不相同，以下是常用的几种系统资源类型。

◇从文件导入的资源类型。

比如 BITMAP、ICON、HTML、FONT 等。

```
    nameID BITMAP filename
```

实例 9-18 中使用的图标资源 IDI_ICON_MAIN 如下：

```
    IDI_ICON_MAIN          ICON                    "res\\icon.ico"
```

◇对话框资源类型。

- 比如 DIALOG、DIALOGEX 等，对话框可包括若干子控件。

```
    nameID DIALOGEX x, y, width, height [ , helpID]] [[ optional-statements]]
{ control-statements }
```

实例 9-18 的Ⅱ)D_DIALOG ABOUT 是 DIALOGEX 类型的资源。

◇菜单资源类型。

MENU 包括如下菜单项：

```
    menuID MENU  [[optional-statements]]  { item-definitions  . . .  }
```

菜单资源中一般会包括 POPUP 资源，如下所示：

```
    POPUP text, [[optionlist]]{ item-definitions  . . . }
```

实例 9-18 中的 IDR- MENU POPUP、IDR_MENU_MAIN 等是 MENU 类型的资源。

◇字符串表类型。

STRINGTABLE 包括若干字符串，如下所示：

```
    STRINGTABLE  [[optional-statements]] {stringID string  . . .  }
```

◇　自定义类型。

此外还可以由编程人员来自行定义资源类型，如下所示：

```
    nameID typeID filename
    nameID typeID  {  raw-data  }
```

(2)控件。

“控件”与系统通用控件相对应，编译器会解析脚本，在程序运行时，脚本中相关声明被引用时会创建控件。

```
    <控件类型> text, id, x, y, width, height [[, style [[, extended-style]]]]
```

或

```
    <控件类型>, id, x, y, width, height [[, style [[, extended-style]]]]
```

如在实例 9-19 的对话框资源中包括的 TEXT 控件：

```
       LTEXT          "Windows Resource Demo",3001,26,14,119,8,SS_NOPREFIX
```

和按钮控件：

```
       DEFPUSHBUTTON   "OK",IDOK,234,14,32,14,WS_GROUP
```

(3)声明。

“声明”用于设置“资源”的相关属性。根据资源的不同而不同，有的资源有“声明”，

有的资源没有，且每一种资源对应特定的声明。

比如 MENU 的资源就会具有 MENUITEM 声明，用于指定菜单项。

对话框资源（DIALOG、DIALOGEX）具有 CAPTION 声明，用于指定对话框的标题。

4．编译资源脚本

编译脚本是将．rc 文件编译为可供链接器链接的.res 文件。资源脚本编译使用脚本编译器 rc.exe。

rc.exe 有若干种选项，一般在编译脚本都不常用，本书不再详细介绍。最常用的是/fo 选项，用于指定输出的 res 文件路径，比如：

```
rc.exe /fo"Debug\Window.res" Window.rc
```

9.5.2 资源ID定义和头文件

资源 ID 是一个资源或资源子项的唯一标识，很多资源和子项都具有标识。实例 9-19 中的 IDD DIALOG ABOUT、ID_FILE_COPY 等就是资源 ID。

这些 ID 实际是由用户定义的常数，都定义于 Resource.h 头文件中。

```
//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by Windows.rc
//
#define IDD_DIALOG_ABOUT                101
#define IDR_MENU_MAIN                   101
#define IDR_MENU_POPUP                  102
#define IDI_ICON_MAIN                   103
#define IDC_EDIT_ABOUT                  1001
#define ID_FILE_COPY                    40001
#define ID_FILE_SAVE40002               40002
#define ID_FILE_EXIT                    40003
#define ID_EIDT_COPY                    40004
#define ID_EIDT_PASTE                   40005
#define ID_HELP_ABOUT                   40006
#define ID_COPY_PASTE                   40007
#define ID_COPY_PASTE40008              40008
#define ID_POPUP_COPY                   40009
#define ID_POPUP_PASTE                  40010
#define ID_POPUP_CUT                    40011
#define ID_POPUP_INFO                   40012
#define ID_INFO_A                       40013
#define ID_INFO_B                       40014

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        104
#define _APS_NEXT_COMMAND_VALUE         40015
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

资源脚本文件和需要使用资源的程序文件都需要包含 Resource.h。在引用资源时，需要指明资源的 ID，资源 ID 在资源和程序中都是唯一标识，作为程序和资源的接口。

9.5.3 在程序中使用资源

在程序中使用资源也是通过资源 ID 来标识一个资源的。

Window.exe 源代码中引用图标和菜单资源，如下所示：

```
wcx.hIcon = LoadIcon(NULL,     MAKEINTRESOURCE(IDI_ICON_MAIN)); // 图标
…
wcx.lpszMenuName = MAKEINTRESOURCE(IDR_MENU_MAIN); // 菜单资源
```

Window.exe 源代码中引用菜单项 ID 和对话框资源，如下所示：

```
        // 命令输入，菜单项被选中，或按钮被单击
    case WM_COMMAND:
        {
            switch(LOWORD(wParam))
            {
                // 在“帮助”菜单中选择“关于”
            case ID_HELP_ABOUT:
                {
                    DialogBox(
                        hinst, (LPCTSTR)IDD_DIALOG_ABOUT, hwnd, (DLGPROC)About
                        );
                    return 0;
                }
            default:
                return DefWindowProc(hwnd, uMsg, wParam, lParam);
            }
        }
```

这里的 JDI ICON MAIN、JDR MENU MAIN、ID HELP ABOUT 和 IDD DIALOG ABOUT 都是资源 ID。在 Resource.h 可以查找到其定义。

MAKEINTRESOURCE 是一个宏，这个宏的功能是将资源 ID 转换为资源名（字符串型），很多 API 函数在引用资源时都是通过资源名。

> **注意** 当然不是只有界面才会使用资源，任何程序都可以使用资源。资源在程序中也并非只用于图形用户界面。资源可以是任意的数据。比如完全可以将一些配置信息、动态连接库文件编译为 exe 的资源。在程序运行后，将资源数据写入到文件再使用。这样就可以将所有程序和数据集中在一个可执行文件里（缺点是运行速度较慢）。

```
HMENU LoadMenu(
    HINSTANCE hInstance,
```

9.6 菜单

菜单是一种重要的界面元素，本节主要说明如何动态地增加、删除和设置菜单及菜单项，说明如何处理用户对菜单输入命令的消息处理。

9.6.1 菜单资源和菜单句柄

如实例 9-18 所示，可以在资源中配置菜单。

有几种情况可以使用资源中的菜单，一是在窗口菜单栏中的菜单，二是右键弹出等上下文菜单。

无论是通过 CreateWindow、RegisterClass 等窗口相关函数，还是 TrackPopupMenuEx 等函数弹出上下文菜单都需要使用菜单句柄。

菜单句柄可以直接从菜单资源中获取，使用 LoadMenu API 函数。

(1) LoadMenu 函数原型如下：

```
    LPCTSTR lpMenuName
);
```

lpMenuName 可以使用 MAKEINTRESOURCE 从资源 ID 转换得来。

(2) TrackPopupMenuEx 函数的功能是弹出菜单，函数原型如下：

```
BOOL TrackPopupMenuEx(
    HMENU hmenu,
    UINT fuFlags,
    int x,
    int y,
    HWND hwnd,
    LPTPMPARAMS lptpm
);
```

也可以直接在程序中通过 API 函数动态创建菜单，获得菜单句柄。CreateMenu 函数创建一个空菜单。

```
HMENU CreateMenu(VOID);
```

9.6.2 动态增加、删除、设置菜单及菜单项

只要有菜单句柄，无论是从资源创建的，还是使用 CreateMenu 创建的，都可以再设置菜单及其项。设置菜单的 API 函数有 InsertMenuItem、AppendMenu、DeleteMenu、ModifyMenu 等。

InsertMenuItem 的功能是增加菜单项，函数原型如下：

```
BOOL InsertMenuItem(
    HMENU hMenu,
    UINT uItem,
    BOOL fByPosition,
    LPCMENUITEMINFO lpmii
```

参数较简单，其中 lpmii 是指向 MENUITEMINFO 结构的变量。

AppendMenu 函数的功能是给菜单附加一个新的项。DeleteMenu 的功能是删除菜单项。ModifyMenu 的功能是修改菜单项。

```
BOOL AppendMenu(
    HMENU hMenu,
    UINT uFlags,
    UINT_PTR uIDNewItem,
    LPCTSTR lpNewItem
);
BOOL DeleteMenu(
    HMENU hMenu,
    UINT uPosition,
    UINT uFlags
);
BOOL ModifyMenu(
    HMENU hMnu,
    UINT uPosition,
    UINT uFlags,
    PTR uIDNewItem,
    LPCTSTR lpNewItem
);
```

9.6.3 菜单消息处理

用户在菜单中单击了某个菜单项后，菜单所属窗口的句柄会收到 WM_COMMAND 消息。那么如何才能确定用户点击的是哪个菜单项呢？WM_COMMAND 消息的 wParam 参数的低位就是菜单项 ID。

9.7 对话框

对话框在应用程序运行过程中，一般用于获得比较复杂、数据量较大的用户输入。对话框是一种特殊的窗口，可以通过资源来创建，也具有消息处理的过程。

图 9-12 所示对话框是由本节实例代码创建。

9.7.1 创建对话框

如果通过资源创建对话框，可以使用 CreateDialog、CreateDialogParam、DialogBox、DialogBoxParam 等 API 宏或函数。

以 DialogBox 为例，DialogBox 定义如下：

```
INT_PTR DialogBox(
    HINSTANCE hInstance,
    LPCTSTR lpTemplate,
    HWND hWndParent,
    DLGPROC lpDialogFunc
);
```

hInstance 是应用程序实例句柄；lpTemplate 是资源名，可以由宏 MAKEINTRESOURCE 将资源 ID 转换得来；hWndParent 是对话框的父窗口句柄，lpDialogFunc 是对话框消息函数。

例如，在实例 9-9 中使用 DialogBox 创建对话框：

```
DialogBox(hinst, (LPCTSTR)IDD_DIALOG_ABOUT, hwnd, (DLGPROC)About);
```

其中使用的资源是 IDD DIALOG ABOUT，父窗口是主窗口，对话框消息处理函数是 About。

也可以不依赖于资源直接创建对话框。CreateDialogIndirect、CreateDialogIndirectParam、DialogBoxIndirect、DialogBoxIndirectParam 等 API 宏和函数实现了直接创建对话框。

以 CreateDialogIndirect 为例，CreateDialogIndirect 如下：

```
INT_PTR DialogBoxIndirect(
    HINSTANCE hInstance,
    LPCDLGTEMPLATE lpTemplate,
    HWND hWndParent,
    DLGPROC lpDialogFunc
);
```

与 DialogBox 的 lpTemplate 参数的使用方法不同，本函数的 lpTemplate 参数指向一段内存，内存中依次放置一系列的数据结构。其第一个是 DLGTEMPLATE 结构，表示了对话框窗口的相关属性。其后放置若干个 DLGITEMTEMPLATE 结构，表示对话框上的控件，控件数量由 DLGTEMPLATE 结构的 cdit 成员指定(第一个也可以是 DLGTEMPLATEEX 结构，其后的结构应为 DLGITEMTEMPLATEEX)。

DLGTEMPLATE 和 DLGITEMTEMPLATE 结构定义如下：

```
typedef struct {
    DWORD style;
    DWORD dwExtendedStyle;
    WORD cdit;
    short x;
    short y;
    short cx;
    short cy;
} DLGTEMPLATE, *LPDLGTEMPLATE;

typedef struct {
    DWORD style;
    DWORD dwExtendedStyle;
    short x;
    short y;
    short cx;
    short cy;
    WORD id;
} DLGITEMTEMPLATE, *PDLGITEMTEMPLATE;
```

9.7.2 对话框消息处理函数

在创建对话框时，均需指明对话框消息处理函数。对话框消息处理函数与窗口消息处理函数使用方法相似，但是所处理的消息类型有所不同。在对话框被创建时，收到的消息为WM_INITDIALOG。

对话框上的按钮被单击时，对话框消息处理函数也会收到 WM_COMMAND 消息，消息的 wParam 是按钮的 ID。

如果一个消息能由对话框的消息处理函数处理，那么需返回 TRUE。如果不能处理返回 FALSE。

实例 9-20 对话框消息处理函数

本消息处理函数实现了简单的功能。在用户单击了对话框的“确定”(IDOK)按钮后，关闭对话框，单击“退出”(IDCANCEL)按钮后，关闭对话框。

```
/* ************************************
* About
* 功能: "关于" 对话框消息处理函数
************************************/
LRESULT CALLBACK About(
                        HWND hDlg,       // 对话框窗口句柄
                        UINT message,    // 消息
                        WPARAM wParam,   // 消息参数
                        LPARAM lParam)   // 消息参数
{
   switch (message)
   {
   case WM_INITDIALOG:
        return TRUE;

   case WM_COMMAND:
        // 单击 "OK" 按钮, 结束对话框
        if(LOWORD(wParam) == IDOK)
        {
             EndDialog(hDlg, LOWORD(wParam));
             return TRUE;
        }
        // 单击 "退出" 按钮, 结束进程
        if(LOWORD(wParam) == IDCANCEL)
        {
             ExitProcess(0);
        }
        break;
   }
   return FALSE;
}
```

代码中的 EndDialog 函数的功能是关闭对话框，原型如下：

```
BOOL EndDialog(
   HWND hDlg,
   INT_PTR nResult
);
```

运行结果如图 9-12 所示。

第 10 章　系统信息的管理

获取与管理的 Windows 系统信息包括系统基本信息、时间信息、注册表。

本章 10.1 节主要介绍 Windows 的系统信息的获取，包括系统目录、Windows 安装目录、用户名、计算机名、域名等。

10.2　节主要介绍系统时间的获取和修改。

10.3 节主要介绍如何操作注册表。

10.1　Windows系统信息

Windows 系统信息包括系统版本、目录、计算机名、用户名、系统时间、系统颜色设置信息等。本节将介绍如何使用 API 获取这些信息。

首先给出本节所有实例的运行效果，实例 10 -1～实例 10-5 的综合运行结果如图 10-1 所示。

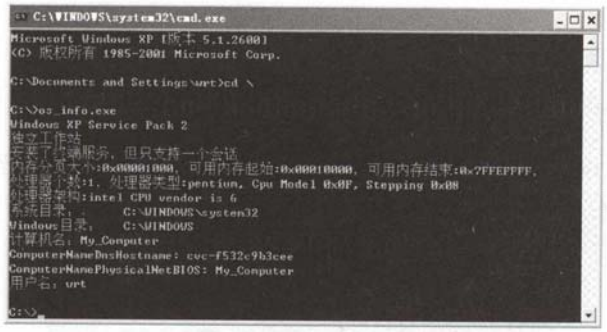

图 10-1　系统信息实例运行结果

10.1.1　获取系统版本

1．系统版本的表示与获取

Windows 对系统版本的表示分为主版本号、次版本号、构建(Build)号、SP 主版本号、SP 次版本号来表示系统版本。比如笔者的主机系统版本号为 5.1 Build 2600 Service Pack 2。SP 版本号会随着 Service Pack 的安装而改变。一般系统的主版本号与次版本号会对应用程序开发人员有比较大的影响。比如 5.0 表示 Windows 2000，5.1 表示 Windows XP，5.2 表示 Windows Server 2003，6.0 表示 Windows Vista。一般每一个新版本的系统都会新增加一些 API。因此有些 API 只能在新版本的 Windows 上使用，而不能在较老版本的 Windows 上使用。

2．关键 API 与数据结构

(1) GetVersionEx。

可以使用 GetVersionEx 函数获取系统版本信息，函数原型如下：

```
BOOL GetVersionEx(
  LPOSVERSIONINFO lpVersionInfo
);
```

◇参数

lpVersionInfo：输入、输出参数，此参数为 OSVERSIONINFO 或者 OSVERSIONINFOEX 结

构的指针，用于存储获取的系统版本信息。这两个结构的第一个成员都是 dwOSVersion InfoSize，如果使用 OSVERSIONINFO 结构那么将此成员设置为 sizeof(OSVERSIONINFO)如果使用 OSVERSIONINFOEX 结构，那么设置此成员为 sizeof(OSVERSIONINFOEX)。

◇返回值

BOOL 类型，表示是否成功。使用 GetLastError 函数获取错误代码。

(2) OSVERSIONINFOEX 结构。

```
typedef struct _OSVERSIONINFOEX {
  DWORD dwOSVersionInfoSize;
  DWORD dwMajorVersion;
  DWORD dwMinorVersion;
  DWORD dwBuildNumber;
  DWORD dwPlatformId;
  TCHAR szCSDVersion[128];
  WORD wServicePackMajor;
  WORD wServicePackMinor;
  WORD wSuiteMask;
  BYTE wProductType;
  BYTE wReserved;
} OSVERSIONINFOEX, *POSVERSIONINFOEX, *LPOSVERSIONINFOEX;
```

实例 10-1　获取系统版本

本实例使用 GetVersionEx 函数获取系统版本、产品类型等，并打印出结果，以版本所对应的真实版本名称的形式打印。

```
/* ************************************
* void ShowVersionInfo()
* 功能：获取并显示系统版本信息
************************************/
void ShowVersionInfo()
{
   OSVERSIONINFOEX ovex;
   CHAR szVersionInfo[1024];
   *szVersionInfo = NULL;
   // 设置参数大小，调用并判断是否成功
   ovex.dwOSVersionInfoSize = sizeof(OSVERSIONINFOEX);
   if(!GetVersionEx(&ovex))
   {
      printf("error %d\n",GetLastError());
```

```
        return;
    }
    // 判断版本
    if(ovex.dwMajorVersion==5)
    {
        if(ovex.dwMinorVersion==0)
            lstrcat(szVersionInfo,"Windows 2000 ");
        else if(ovex.dwMinorVersion==1)
            lstrcat(szVersionInfo,"Windows XP ");
        else if(ovex.dwMinorVersion==2)
            lstrcat(szVersionInfo,"Windows Server 2003 ");
    }
    else if(ovex.dwMajorVersion == 6)
        lstrcat(szVersionInfo,"Windows Vista ");
    else
        lstrcat(szVersionInfo,"Windows NT 4.0 或者其他 ");
    // 安装的 SP, 字符串
    lstrcat(szVersionInfo,ovex.szCSDVersion);
    // 判断 wProductType, 产品类型
    switch(ovex.wProductType)
    {
    case VER_NT_DOMAIN_CONTROLLER:
        lstrcat(szVersionInfo,"\n 域控制器");
        break;
    case VER_NT_SERVER:
        lstrcat(szVersionInfo,"\n 服务器");
        break;
    case VER_NT_WORKSTATION :
        lstrcat(szVersionInfo,"\n 独立工作站");
        break;
    }
    // 判断 wSuiteMask
    if(ovex.wSuiteMask & VER_SUITE_PERSONAL)
    {
        lstrcat(szVersionInfo,"\nWindows XP Home Edition");
    }
    if(ovex.wSuiteMask & VER_SUITE_SINGLEUSERTS)
    {
        lstrcat(szVersionInfo,"\n 安装了终端服务, 但只支持一个会话");
    }
    // wSuiteMask 成员还可能是以下值的组合
    //VER_SUITE_BLADE
    //VER_SUITE_COMPUTE_SERVER
    //VER_SUITE_DATACENTER
    //VER_SUITE_ENTERPRISE
    //VER_SUITE_EMBEDDEDNT
    //VER_SUITE_PERSONAL
    //VER_SUITE_SINGLEUSERTS
    //VER_SUITE_SMALLBUSINESS
    //VER_SUITE_SMALLBUSINESS_RESTRICTED
    //VER_SUITE_STORAGE_SERVER
    //VER_SUITE_TERMINAL
    printf("%s\n",szVersionInfo);
}
```

实例运行结果如图 10-1 所示。

10.1.2 获取计算机硬件信息

1. 获取硬件相关信息

GetSystemInfo 函数可以获取很多系统硬件相关信息，包括内存分页的大小，内存的最小分配单位、粒度，处理器的类型和个数等。

2. 关键 API 与数据结构

(1) GetSystemInfo。

函数原型如下：

```
void GetSystemInfo(
  LPSYSTEM_INFO lpSystemInfo
);
```

◇参数

lpSystemInfo：输出参数，获取的信息保存于 SYSTEM_INFO 结构中。lpSystemInfo 参数为指向 SYSTEM INFO 结构类型变量的指针。

◇返回值

无返回值。

(2) SYSTEM INFO 结构的定义如下：

```
typedefstr uct _SYSTEM_INFO {
  union {
    DWORD dwOemId;
    struct {
      WORD wProcessorArchitecture;
      WORD wReserved;
    };
  };
  DWORD dwPageSize;
  LPVOID lpMinimumApplicationAddress;
  LPVOID lpMaximumApplicationAddress;
  DWORD_PTR dwActiveProcessorMask;
  DWORD dwNumberOfProcessors;
  DWORD dwProcessorType;
  DWORD dwAllocationGranularity;
  WORD wProcessorLevel;
  WORD wProcessorRevision;
} SYSTEM_INFO;
```

成员变量的意义见实例 8-2。

实例 10-2 获取系统硬件相关信息

本实例使用 GetSystemInfo 获取系统硬件信息并打印出来，包括内存分页大小，处理器数量和类型、处理器架构等。

```
/* ***********************************
* void ShowSystemInfo()
* 功能    获取并显示硬件相关信息
***
***********************************/
void ShowSystemInfo()
{
   SYSTEM_INFO si;
   GetSystemInfo(&si);
   printf("内存分页大小:0x%.8X, 可用内存起始:0x%.8X, 可用内存结束:0x%.8X, \n"
      "处理器个数:%d, 处理器类型:",
      si.dwPageSize,
      si.lpMinimumApplicationAddress,
      si.lpMaximumApplicationAddress,
      si.dwNumberOfProcessors);
```

```
    switch (si.dwProcessorType)
    {
    case PROCESSOR_INTEL_386:
       printf("386");
       break;
    case PROCESSOR_INTEL_486:
       printf("486");
       break;
    case PROCESSOR_INTEL_PENTIUM:
       printf("pentium");
       printf(", Cpu Model 0x%.2X, Stepping 0x%.2X",
          (BYTE)(si.wProcessorRevision>>8),
          (BYTE)si.wProcessorRevision);
       break;
    }
    printf("\n处理器架构:");
    switch (si.wProcessorArchitecture)
    {
    case PROCESSOR_ARCHITECTURE_INTEL:
       printf("intel");
       printf(" CPU vendor is %d",si.wProcessorLevel);
       break;
    case PROCESSOR_ARCHITECTURE_IA64:
       printf("64 bits intel");
       break;
    case PROCESSOR_ARCHITECTURE_AMD64:
       printf("64 bits AMD");
       break;
    case PROCESSOR_ARCHITECTURE_UNKNOWN:
       printf("UNKNOWN");
       break;
    }
    printf("\n");
}
```

运行结果见图 10-1。

> **注意** 可以使用 GetSystemInfo 函数获取虚拟分页的大小和内存分配粒度，在进行内存处理时可能会涉及。

10.1.3 获取系统目录等信息

1. 系统目录和 Windows 安装目录

Windows 的系统目录会在安装的时候确定，在不同主机上运行的系统可能不相同。另外，不同 Windows 版本其系统目录的默认情况也不是相同的。

获取系统目录使用函数 GetSystemDirectory，获取 Windows 系统的安装目录可以使用函数 GetWindowsDirectory。

2. 关键 API

(1) GetSystemDirectory，函数原型如下：

```
UINT GetSystemDirectory(
  LPTSTR lpBuffer,
  UINT uSize
);
```

◇参数

lpBuffer：输出参数，保存系统路径的缓冲区，由调用者分配。参数 uSize 为其大小（字符数）。

一般大小为 MAX_ PATH 即可。

uSize：输出参数，必须设置为 lpBuffer 所指向的缓冲区的大小，字符数不能大于

lpBuffer 缓冲区的大小，否则可能会引起溢出。

◇返回值

如果返回值大于 uSize，说明缓冲区小于返回的字符串，返回值为所需的缓冲区大小。如果返回 0，则表示失败。如果返回非 0 且小于 nSize 的值，则执行成功，返回值大小为系统目录的字符数。

(2) GetWindowsDirectory。

```
UINT GetWindowsDirectory(
  LPTSTR lpBuffer,
  UINT uSize
);
```

GetWindowsDirectory 函数的参数与返回值的意义与 GetSystemDirectory 函数相似。

实例 10-3 获取系统目录和 Windows 安装目录

本实例使用 GetSystemDirectory 与 GetWindowsDirectory 函数获取系统相关路径并打印出来。

```
/* ************************************
*void GetFolders()
* 功能    获取系统目录等信息
************************************/
void GetFolders()
{
    TCHAR szSystemDirectory[MAX_PATH];
    TCHAR szWindowsDirectory[MAX_PATH];

    GetSystemDirectory(szSystemDirectory,MAX_PATH);
    GetWindowsDirectory(szWindowsDirectory,MAX_PATH);
    printf("系统目录: \t%s\nWindows 目录: \t%s\n",
        szSystemDirectory,
        szWindowsDirectory);
}
```

运行结果见图 10-1。

10.1.4 用户名、计算机名、域名

获取当前登录用户使用 API 函数 GetUserName，获取计算机名使用 API 函数 GetComputerName，获取特定格式的计算机名使用 API 函数 GetComputerNameEx，设置计算机名使用 API 函数 SetComputerName 和 SetComputerNameEX.

1．关键 API

(1) GetUserName。

获取当前登录的用户名：

```
BOOL GetUserName(
  LPTSTR lpBuffer,
  LPDWORD nSize
);
```

◇参数

lpBuffer：输出参数，指向保存登录用户名的内存缓冲区。其大小最好不小于(UNLEN+1)，UNLEN 定义在 Lmcons.h 中。

nSize:输入、输出参数，指向表示 lpBuffer 的大小（字符数）的 DWORD 型变量。如果小于所需要的大小，那么 lpnSize 返回需要的大小，函数返回失败。

(2) GetComputerName。

函数的定义如下：

```
BOOL GetComputerName(
  LPTSTR lpBuffer,
  LPDWORD lpnSize
);
```

◇参数

lpBuffer:输出参数，指向保存计算机名的内存缓冲区，其大小最好不小于(MAX COMPUTERNAME 一 LENGTH+1)。

lpnSize：输入、输出参数，指向表示 lpBuffer 的大小（字符数）的 DWORD 型变量。如果小于所需要的大小，那么 lpnSize 返回需要的大小，函数返回失败。

◇返回值

BOOL 型，表示是否成功。使用 GetLastError 获取错误信息。

(3) GetComputerNameEx.

除了可以获得计算机名外，还可以获得 NetBIOS 或者 DNS 名，函数定义如下：

```
BOOL GetComputerNameEx(
  COMPUTER_NAME_FORMAT NameType,
  LPTSTR lpBuffer,
  LPDWORD lpnSize
);
```

◇参数

NameType:所需要获取的计算机名的类型，COMPUTER NAME FORMAT 枚举类型。详见示例。

其他两个参数使用方法与 GetComputerName 参数相同。

◇返回值

BOOL 型，表示是否成功。使用 GetLastError 获取错误信息。

(4) SetComputerName、SetComputerNameEx.

SetComputerName 设置计算机 NetBIOS 名。SetComputerNameEx 可以设置 Dns Domain、DnsHostname、 NetBIOS。

2．实例 10-4 用户名、计算机名

本实例使用 GetUserName 获取用户名，使用 GetComputerName 获取计算机名，并演示了 GetComDuterNameEx 的使用方法。

```
/* ************************************
* void GetNames()
* 功能：获取计算机名、用户名等信息
************************************/
void GetNames()
{
    DWORD dwComputerNameLen = MAX_COMPUTERNAME_LENGTH+1;
    DWORD dwUserNameLen = UNLEN+1;
    TCHAR szComputerName[MAX_COMPUTERNAME_LENGTH+1];
    TCHAR szUserName[UNLEN + 1];
    COMPUTER_NAME_FORMAT cnf;
    if(!SetComputerName("My_Computer"))
```

```
    {
        printf("Set Error %d",GetLastError());
    }
    GetComputerName(szComputerName,&dwComputerNameLen);

    printf("计算机名: %s\n",szComputerName);

    //ComputerNameNetBIOS
    //ComputerNameDnsHostname
    //ComputerNameDnsFullyQualified
    //ComputerNamePhysicalNetBIOS
    //ComputerNamePhysicalDnsHostname
    //ComputerNamePhysicalDnsDomain
    //ComputerNamePhysicalDnsFullyQualified
    dwComputerNameLen = MAX_COMPUTERNAME_LENGTH+1;
    GetComputerNameEx(ComputerNameDnsHostname,szComputerName,&dwComputerNameLen);
    printf("ComputerNameDnsHostname: %s\n",szComputerName);

    dwComputerNameLen = MAX_COMPUTERNAME_LENGTH+1;
    GetComputerNameEx(ComputerNamePhysicalNetBIOS,szComputerName,&dwComputerNameLen);
    printf("ComputerNamePhysicalNetBIOS: %s\n",szComputerName);
    GetUserName(szUserName,&dwUserNameLen);
    printf("用户名: %s\n",szUserName);
}
```

运行结果如图 10-1 所示。

10.1.5 处理系统颜色信息、尺度信息等

GetSysColor 可以获取系统中与颜色有关的信息，包括桌面颜色、窗口颜色等。与图 10-2 所示“控制面版一显示一外观一高级”选项中所能进行的设置相同。

图 10-2 高级外观设置

函数原型如下：

```
DWORD GetSysColor(
  int nIndex
);
```

与 GetSysColor 函数功能相对的函数是 SetSysColors，可以设置系统颜色。

```
BOOL SetSysColors(
  int cElements,
  const INT* lpaElements,
  const COLORREF* lpaRgbValues
);
```

GetSystemMetrics 可以获取系统中各种尺度信息，包括图标大小(SM CYSMICON)、窗口标题栏的系统按钮大小(SM_CYSIZE)等。图 10-2 所示高级外观设置对话框可获取和设置尺度大小信息，GetSystemMetricsAPI 函数用于对尺度信息的获取，函数原型如下：

```
int GetSystemMetrics(
  int nIndex
);
```

10.1.6 鼠标、键盘等外设信息

1．关键 API 与数据结构

SystemParametersInfo API 函数可以设置和获取多种系统参数，包括 Windows 辅助功能参数、桌面设置参数、图标参数、输入输出参数（包括鼠标、键盘设置，语言设置警告音设置）菜单参数、电源设置参数、屏幕参数。

函数定义如下：

```
BOOL SystemParametersInfo(
  UINT uiAction,
  UINT uiParam,
  PVOID pvParam,
  UINT fWinIni
);
```

◇参数

uiAction 是最主要的参数。需要设置和获取的系统参数。具有种类有很多，可参见 MSDN 和 SDK 文档。这个参数也决定了 uiParam、pvParam 的设置情况。

fWinIni:输入参数，指明设置的改变是否发送 WM_SETTINGCHANGE 消息给当前系统中的窗口程序，以及是否修改用户剖面（profile）配置信息，参数可以是 SPIF__UPDATEINIFILE 或 SPIF_SENDCHANGE 或其组合。

◇返回值

表示操作是否成功。使用 GetLastError 获取错误信息。

2．实例 10-5 鼠标速度

本实例是使用 SystemParametersInfo 函数获取和设置当前系统的鼠标速度。实例代码如下：

```
/* ************************************
* void MouseSpeed()
* 功能：获取鼠标速度
************************************/
void MouseSpeed()
{

   BOOL fResult;
   int aMouseInfo[3];        // 保存数据信息的数组

   // 调用 SystemParametersInfo
   fResult = SystemParametersInfo(
      SPI_GETMOUSE,          // 获取鼠标信息
      0,                     // 未使用
      &aMouseInfo,           // 用于保存鼠标信息
      0);                    // 未使用

   // 把鼠标速度加倍
   if( fResult )
   {
```

```
        aMouseInfo[2] = 2 * aMouseInfo[2];

        SystemParametersInfo(
            SPI_SETMOUSE,          // 设置鼠标信息
            0,                     // 未使用
            aMouseInfo,            // 鼠标信息
            SPIF_SENDCHANGE);      // 更新 win.ini
    }
}
```

程序运行后，可以感觉到鼠标移动速度有所提升。

10.2 时间信息

本节将介绍如何获取和设置系统时间。

10.2.1 设置、获取系统时间

获取和设置系统时间使用 API 函数 GetLocalTime、GetSystemTime、GetSystemAdjustment、GetSystemTimeAsFileTime 和 GetSystemTimes 等。

相应的设置函数包括 SetLocalTime、SetSystemTime、SetSystemTimeAdjustment 等。

GetTimeZoneInformation 函数可以获取系统时区设置。SystemTimeToTzSpecific LocalTime 和 TzSpecificLocalTimeTo SystemTime 用于标准时间和特定时区的时间之间进行转换。SetTimeZoneInformation 用于设置时区信息。此外还有 GetDynamicTimeZone Information 函数。

以上 API 函数在进行操作时，大都使用了 SYSTEMTIME 结构，用于保存系统时间，此结构定义如下：

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

在涉及时区信息操作时，使用 TIME_ZONE_INFORMATION 结构表示时区信息，定义如下：

```
typedef struct _TIME_ZONE_INFORMATION {
  LONG Bias;
  WCHAR StandardName[32];
  SYSTEMTIME StandardDate;
  LONG StandardBias;
  WCHAR DaylightName[32];
  SYSTEMTIME DaylightDate;
  LONG DaylightBias;
} TIME_ZONE_INFORMATION,  *PTIME_ZONE_INFORMATION;
```

以上函数的参数设置都较简单，具体参见实例 10-6。

实例 10-6 设置、获取系统时间

本实例使用本节所述函数获取系统当前的国际标准时间、当前的本时区时间、获取系统所设置的时区，并将时间在各时区内进行了转换，然后修改系统时间，使其提前 1 小时。

```
*************************************/
#include <Windows.h>
#include <stdio.h>
/* ************************************
* int main()
* 功能：获取并显示系统当前时间，然后将时间提前一个小时
*************************************/
int main()
{
    SYSTEMTIME st;
    // 获取当前时间，以本时区时间格式
    GetLocalTime( &st );
    printf("Now: %d-%d-%d, %d:%d:%d",
        st.wYear, st.wMonth, st.wDay, st.wHour, st.wMinute, st.wSecond);
    // 提前一小时
    st.wHour --;
    // 设置当前系统时间
    SetLocalTime( &st );
}
```

运行结果如图 10-3 和图 10-4 所示。

图 10-3　实例 10-6 运行第一次

图 10-4　实例 10-6 运行第二次

从运行结果可以看出，实例获取了当前的系统时间，并将时间提前了一个小时。

> **注意**　在设置系统时间时，SYSTEMTIME 结构中的 wDayOfWeek 成员是无效的，系统会根据其他成员自动计算星期。

10.2.2　获取开机至现在持续的时间

GetTickCount 和 GetTickCount64 函数可以获取系统开机到现在所持续的时间，以 ms 为单位。

使用 GetTickCount 和 GetTickCount64 函数获取开机至现在持续的毫秒数；然后获取当前时间，并通过转换计算出系统开机时间。

10.2.3　文件时间与系统时间的转换

Windows 使用了不同的数据类型来存储文件时间和系统时间，由于文件系统中的每个文件都有若干种文件时间，文件时间的数据量很大，所以 Windows 使用了最少的数据来表示文件时间，FILETIME 结构的大小只有两个 DWORD，结构定义如下：

```
typedef struct _FILETIME {
  DWORD dwLowDateTime;
  DWORD dwHighDateTime;
} FILETIME, *PFILETIME;
```

可以使用 SystemTimeToFileTime 和 FileTimeToSystemTime 将时间信息在文件时间格式和系

统时间格式之间转换。

文件的时间属性及操作方法见第 4 章。有关于这个两函数的使用详见第 4 章与文件时间相关的实例。

10.3 注册表

本节介绍与注册表相关的操作。首先简单介绍注册表的组织形式、使用注册表编辑器查阅和修改注册表；然后介绍如何使用程序操作注册表，包括获取注册表的内容、修改注册表等。

10.3.1 注册表的作用及组织形式

Windows 系统使用注册表来存储系统和应用程序配置数据。很多系统和应用程序重要的配置的信息都存储在注册表中。注册表是一种以树型结构组织的数据库。树的每一个节点称作键(key)，每个节点可以包含若干子键，键可以存储数据，存储的数据称作键值。一个键无论其是否有子键，都可以有键值。一个键可以有很多个键值，每个键值都有一个名字，包含若干数据，所包含的数据类型可以有很多种，表 10.1 列出了注册表数据类型。

表 10.1 注册表键值类型

类　型	意　义
REG_BINARY	任何形式的二进制数据
REG_DWORD	32 位数据
REG_DWORD_LITTLE_ENDIAN	以 little-endian 形式存储的 32 位数据
REG_DWORD_BIG_ENDIAN	以 big-endian 形式存储的 32 位数据
REG_EXPAND_SZ	以 NULL 结尾的字符串，可以使用环境变量，如%PATH%
REG_LINK	只能由操作系统使用的类型
REG_MULTI_SZ	字符串数组，以 NULL 结束字符串，以两个 NULL 结束数组
REG_NONE	未定义的数据类型
REG_QWORD	64 位数据
REG_QWORD_LITTLE_ENDIAN	little-endian 形式的 64 位数据
REG_SZ	字符串，以 NULL 结束

系统自带的注册表编辑器（regedit.exe，如图 10-5 所示）可以列举、查看、修改、增加、删除键和子键、键值。

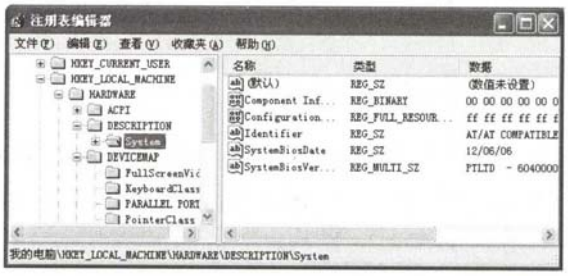

图 10-5　注册表编辑器

从上图可以看出，键“HKEY LOCAL MACHINE\HARDWARE\DESCRIPTION\ System”具有 6 个键值，每个键值都有名称，各键值的数据类型不相同，并且，此键还具有子键。

10.3.2 键、子键、键属性及键值的相关操作

1．原理

在对注册表键进行操作前，需要使用 API 函数 RegOpenKey、RegOpenKeyEx 打开键。在打开键后，如果需要创建新的键，使用 API 函数 RegCreateKey、RegCreateKeyEx。这两组函数都将返回键的句柄（HKEY 类型）。

在得到键的句柄后，可以对键进行操作。创建子键使用前述 RegCreateKey、RegCreateKeyEx API 函数；删除键使用 RegDeleteKey、RegDeleteKeyEx 函数，RegQueryInfoKey API 函数获取键多方面的信息，包括键值、子键等；获取和设置键值还可以使用 RegGetValue、RegQueryValueEx、RegSetValueEx、RegQueryMultipleValues 等函数；删除指定的键值使用 RegDeleteValue 函数；如果需要遍历子键和键值，
可以使用 API 函数 RegEnumKeyEx 和 RegEnumValue 函数。

2．关键 API

(1) RegOpenKeyEx。

打开已经存在的键，函数原型如下：

```
LONG RegOpenKeyEx(
  HKEY hKey,
  LPCTSTR lpSubKey,
  DWORD ulOptions,
  REGSAM samDesired,
  PHKEY phkResult
);
```

◇参数

hKey:输入参数，需要打开的键，可以是 RegCreateKeyEx 或者 RegOpenKeyEx 等函数的返回值。
可以是 HKEY CLASSES_ ROOT、HKEY CURRENT USER、HKEY USERS、HKEY LOCAL_MACHINE.

lpSubKey:输入参数，子键名称字符串，如“”SOFTWARE\\Microsoft\\WindowsNT\\CurrentVersion\\Image File Execution OptionS’’’’。

ulOptions:系统保留，调用时设置为 0。

samDesired:输入参数，存取权限，可以设置为 NULL，详见 SDK 文档。

phkResult:输出参数，指向保存键句柄的 HEY 类型的变量。RegCloseKey 关闭键时需要这个值。

◇返回值

LONG 类型，如果成功返回 ERROR_SUCCESS，否则返回 FormatMessage 函数设置 FORMAT_MES SAGE_FROM_SYSTEM 获取错误信息。

(2) RegCreateKey。

创建新的键，函数原型如下：

```
LONG RegCreateKey(
  HKEY hKey,
  LPCTSTR lpSubKey,
  PHKEY phkResult
);
```

◇参数

参数的使用方法可类比 RegOpenKeyEx 函数的相应参数。

◇返回值

类比 RegOpenKeyEx 函数的返回值。

(3) RegGetValue、RegSetValue.

分别为获取和设置键值，RegGetValue 函数原型如下所示：

```
LONG RegGetValue(
  HKEY hkey,
  LPCTSTR lpSubKey,
  LPCTSTR lpValue,
  DWORD dwFlags,
  LPDWORD pdwType,
  PVOID pvData,
  LPDWORD pcbData
);
```

◇参数

Hkey:输入参数，由 RegCreateKeyEx、RegOpenKeyEx 等函数获得的键句柄。也可以是如下值：HKEY_ CLASSES_ROOT、HKEY_CURRENT_CONFIG、HKEY_ CURRENT USER、HKEY_ PERFORMANCE NLSTEXT、HKEY PERFORMANCE_TEXT、HKEY_USER_S、HKEY _LOCAL_ MACHINE. HKEY PERFORMANCE__ DATA.

lpSubKey:输入参数,相对于 hkey 的子键,如果对 hkey 键本身的值进行操作,则为 NULL。

lpValue:输入参数，键值名，如果为 NULL 或空字符串，则获取未命名键值或默认键值。

dwFlags:输入参数，类型限制。如果指定的键值类型不符，则执行失败。也可以为设置为 RRF_RT_ANY，不做限制。参见实例 10-1。

pdwType:输出参数，类型信息。

pvData:输入参数保存值的缓冲区。

pcbData:输入、输出参数，指向表示 pvData 缓冲区大小的变量。

◇返回值

LONG 类型的数据。如果成功，返回 ERROR_SUCCESS；如果缓冲区大小不足，返回 ERROR_ MORE__- DATA。

RegGetValue 函数原型如下所示：

```
LONG RegSetValue(
  HKEY hKey,
  LPCTSTR lpSubKey,
  DWORD dwType,
  LPCTSTR lpData,
  DWORD cbData
);
```

(4) RegDeleteValue。

删除键值，函数原型如下：

```
LONG RegDeleteValue(
  HKEY hKey,
  LPCTSTR lpValueName
);
```

(5) RegDeleteKeyEx。

删除键，函数原型如下：

```
LONG RegDeleteKeyEx(
  HKEY hKey,
  LPCTSTR lpSubKey,
  REGSAM samDesired,
  DWORD Reserved
);
```

实例 10-7 注册表操作

本实例演示了对注册表的常规操作，包括打开、创建子键、设置键值，键值类型等。

```
/* ************************************
* void AddKey(HKEY hKey)
* 功能：增加一个子键，并设置键值
************************************/
void AddKey(HKEY hKey)
{
    HKEY hSubKey;
    DWORD dwKeyValue = 100;
    // 创建键
    RegCreateKey(hKey,"MySoftware",&hSubKey);
    // 设置键值
    if( ERROR_SUCCESS != RegSetValueEx(
        hSubKey,
        "TEST",
        NULL,
        REG_DWORD,
        &dwKeyValue,
        sizeof(DWORD)))
    {
        printf("error\n");
    }
}
/* ************************************
* void main(void)
* 功能    打开键，获得键句柄
************************************/
void main(void)
{
    HKEY hTestKey;

    if( RegOpenKeyEx( HKEY_CURRENT_USER,
        TEXT("SOFTWARE"),
        0,
        KEY_READ | KEY_WRITE,
        &hTestKey) == ERROR_SUCCESS
        )
    {
        // 增加键
        AddKey(hTestKey);
        // 列举子键
        QueryKey(hTestKey);
    }
}
```

程序运行后，在 HKEY_CURRENT USER\Sotfware 键下多了一个子键 MySoftware，并且设置了一个 REG_DWORD 类型的键值“TEST”，值为 0x64（100），如图 10-6 所示。

图 10-6　增加子键、设置键值

10.3.3　列举注册表项及键值

1．原理

遍历子键和键值可以使用 API 函数 RegEnumKeyEx 和 RegEnumValue 函数。RegQueryInfoKey，
RegEnumKeyEx 和 RegEnumValue 相配合可以完成对一个键的所有子键及键值的列举。

函数的参数设置情况参见实例 10-8。

2．实例 10-8 遍历子键和键值

本实例是实例 10-7 中所调用的 QueryKey 函数的代码。本实例获取指定键下的所有键值及所有子键。如果将函数进行递归调用就可以遍历注册表树。

```
************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
/* 预定义 */
#define MAX_KEY_LENGTH 255
#define MAX_VALUE_NAME 16383
/* ***********************************
* void QueryKey(HKEY hKey)
* 功能：列举指定注册表项的子键
************************************/
void QueryKey(HKEY hKey)
{
    TCHAR    achKey[MAX_KEY_LENGTH];
    DWORD    cbName;
    TCHAR    achClass[MAX_PATH] = TEXT("");
    DWORD    cchClassName = MAX_PATH;
    DWORD    cSubKeys=0;
    DWORD    cbMaxSubKey;
    DWORD    cchMaxClass;
    DWORD    cValues;
    DWORD    cchMaxValue;
    DWORD    cbMaxValueData;
    DWORD    cbSecurityDescriptor;
    FILETIME ftLastWriteTime;

    DWORD i, retCode;

    TCHAR  achValue[MAX_VALUE_NAME];
    DWORD cchValue = MAX_VALUE_NAME;

    // 获取类名和数量
    retCode = RegQueryInfoKey(
        hKey,                    // 键的句柄
        achClass,                //  类名
        &cchClassName,           // 类名长度
        NULL,                    // 保留
        &cSubKeys,               // 子键的数量
        &cbMaxSubKey,            // 子键的长度
        &cchMaxClass,            // 类长度
        &cValues,                // 子键键值数量
        &cchMaxValue,            // 子键名长度
        &cbMaxValueData,         // 键值长度
        &cbSecurityDescriptor,   // 安全描述符
        &ftLastWriteTime);       // 最后写时间

    // 列举子键
```

```
    if (cSubKeys)
    {
        printf( "\nNumber of subkeys: %d\n", cSubKeys);

        for (i=0; i<cSubKeys; i++)
        {
            cbName = MAX_KEY_LENGTH;
            retCode = RegEnumKeyEx(hKey, i,
                achKey,
                &cbName,
                NULL,
                NULL,
                NULL,
                &ftLastWriteTime);
            if (retCode == ERROR_SUCCESS)
            {
                printf(TEXT("(%d) %s\n"), i+1, achKey);
            }
        }
    }

    // 列举键值
    if (cValues)
    {
        printf( "\nNumber of values: %d\n", cValues);

        for (i=0, retCode=ERROR_SUCCESS; i<cValues; i++)
        {
            cchValue = MAX_VALUE_NAME;
            achValue[0] = '\0';
            retCode = RegEnumValue(hKey, i,
                achValue,
                &cchValue,
                NULL,
                NULL,
                NULL,
                NULL);

            if (retCode == ERROR_SUCCESS )
            {
                printf(TEXT("(%d) %s\n"), i+1, achValue);
            }
        }
    }
}
```

函数将 HKEY_CURRENT USER\Sotfware 下的所有子键打印出来，对比图 10-6 和图 10-7，说明结果正确。

图 10-7　遍历子键

10.3.4 通过注册表设置一个自启动的程序

10.3.4～10.3.7 小节将说明一些注册表的应用，都是应用程序开发人员常用的应用方式。本节将介绍通过注册表设置开机自启动的程序。

程序的自启动有很多种方法，注册表是其中最常用的一种，有多个键具有这种功能。最常用的有：HKEY_LO CAL_MACHINE\S OFTWARE\Microsoft\Windows\CurrentVersion\Run，只需设置 REG 二 SZ 类型的键值，将值设置为程序的路径即可使用程序在系统启动时自启动。如果只需要在下次系统启动时自启动一次，那么可以设置键 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce 的键值。

10.3.5 设置随程序启动而启动的调试器（任何程序）

本节将介绍如何在程序启动时加载调试器（读者也可以用于设置自启动程序）。“HKEY_LOCAL_MACHINE\SOFTWAREYMicrosoft,\Windows NT\CurrentVersion\Image File Execution Options”下的子键可以用于指定启动调试器。如需要调试 os_info.exe，在“Image FileExecution Options”键下创建子键“os_info.exe”；然后创建键值，名为“Debugger”，类型为 REG_SZ，数据为启动调试器的命令行，这里指向 Windbg，如：“C:\Program Files\Debugging Tools forWindows\windbg.exe”（可以是任何可执行文件，比如 notepad.exe，在加载键名称程序模块时，会首先启动“Debugger”键值指定的应用程序，并以键名指向的可执行程序为参数）。运行 os_info.exe
就会启动 Windbg 运行调试。

10.3.6 指定程序崩溃实时调试器

本小节将介绍如何在程序崩溃时使用调试器，用于调试一些特殊的程序。在使用 Windows 时会经常发生程序崩溃的情况。程序开发人员可以设置注册表键，以在程序崩溃时可以启动调试器运行调试。

一般情况下，在程序崩溃后会有如图 10-8 所示的对话框。

如果单击“调试”按钮将会启动实时调试器，如图 10-9 所示。

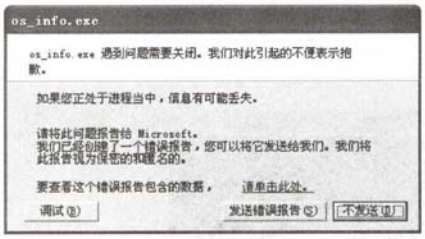

图 10-8 程序崩溃对话框

图 10-9 选择实时调试器

可以通过注册表指定实时调试器，比如指定为 WinDBG 或 OllyDBG 等调试器。指定实时调试器的方法为指定注册表键“HKEY LOCAL MACHINE\SOFTWARE\Microsoft\ Windows NT\CurrentVersion\AeDebug”的“Debugger”键值。

第 11 章　进程间通信

进程的地址空间是私有的。出于安全性的目的，如果一个进程不具有特殊的权限，是无法访问另外一个进程的内存空间的，也无法知道内存中保存的数据的意义。但是在一些具体的应用情况下需要多个进行相互配合，有时计算机用户也需要在多个应用程序之间交换数据。

Windows 系统下，在进程间通信或共享数据的方式如下。

◇ 邮槽(Mailslot)。

邮槽是最为简单的进程间数据共享方式，一个进程创建并拥有一个邮槽，其他进程都可以打开这个邮槽并向其发送消息。11.1 节及相关实例介绍了邮槽的使用。

◇管道(Pipe)。

实质是一种共享的内存，由一个进程创建，其他进程连接，并可进行双向的通信。11.2 节及相关实例介绍了管道的使用。

◇剪贴板(Clipboard)。

剪贴板是重要的进程间数据共享方式，所有的进程都可以设置和修改剪贴板，也都可以从剪贴板获取内容。11.3 节及相关实例介绍了剪贴板的使用。

◇　消息，具体是指 WM_COPYDATA 消息。

消息具有参数 wParam 和 lParam。但是 wParam 和 lParam 最多是 32 位的。而 WM_COPYDATA 消息的参数不受 wParam 和 lParam 数据大小的限制，可以用于在进程间传递数据。11.4 节及相关实例介绍了 WM- COPYDATA 消息的使用方法。

◇网络。

网络可以在不同主机上的不同程序间通信，当然也可以在相同主机上的不同程序间通信。

◇ FileMapping

第 4 章已经介绍了 File Mapping 的使用方法，实例 4-15 已经实现了通过 File Mapping 进行进程间通信的方法。

11.1　邮槽(MailSlot)

使用邮槽通信的进程分为服务端和客户端。邮槽由服务端创建，在创建时需要指定邮槽名，创建后服务端得到邮槽的句柄。在邮槽创建后，客户端可以通过邮槽名打开邮槽，在获得句柄后可以向邮槽写入消息。

邮槽通信是单向的，只有服务端才能从邮槽中读取消息，客户端只能写入消息。消息是先入先出的。客户端先写入的消息在服务端先被读取。

通过邮槽通信的数据可以是任意格式的，但是一条消息不能大于 424 字节。

邮槽除了在本机内进行进程间通信外，在主机间也可以通信。但是在主机间进行邮槽通信，数据通过网络传播时使用的是数据报协议(UDP)，所以是一种不可靠的通信。通过网络进行邮槽通信时，客户端必须知道服务端的主机名或域名。

11.1.1　创建邮槽、从邮槽中读取消息

首先介绍服务端的编写。服务端调用 CreateMailslot API 函数创建邮槽，然后使用

GetMailslotInfo API 函数获取邮槽信息。读取消息使用 ReadFile 函数。

1．邮槽的命名

在本机上邮槽命名的格式为：

```
\\.\mailslot\[path\]name
```

比如：

```
\\.\mailslot\win\bobs_comments
```

也可以使用如下格式的命名，用于在不同主机间的进程通信。

```
\\DomainName\mailslot\[path\]name
\\ComputerName\mailslot\[path\]name
```

可以使用通配符，以进行广播。

```
\\*\mailslot\[path\]name
```

2．关键 API

(1) CreateMailslot。

CreateMailslot 函数的功能是创建一个邮槽对象，CreateMailslot 函数的原型如下：

```
HANDLE WINAPI CreateMailslot(
  __in         LPCTSTR lpName,
  __in         DWORD nMaxMessageSize,
  __in         DWORD lReadTimeout,
  __in_opt     LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

lpName:参数是邮槽名。

nMaxMessageSize:限制了最大消息长度。

lReadTimeout：限制了读超时的时间(ms)，也可设置 0（如果没有消息立即返回）或 MAILSLOT_WAIT_FOREVER（直到读到消息才返回）。

lpSecurityAttributes:安全属性。

(2) GetMailslotInfo。

GetMailslotInfo 函数的功能是获取邮槽的相关信息，函数原型如下：

```
BOOL WINAPI GetMailslotInfo(
  __in         HANDLE hMailslot,
  __out        LPDWORD lpMaxMessageSize,
  __out_opt    LPDWORD lpNextSize,
  __out_opt    LPDWORD lpMessageCount,

  __out_opt    LPDWORD lpReadTimeout
);
```

GetMailslotInfo 函数获取了邮槽的相关信息，包括通过 lpMaxMessageSize 参数返回的消息最大的长度；通过 lpNextSize 消息返回的下一条消息的长度；通过 lpMessageCount 返回的消息数量，通过 lpReadTimeout 返回的读超时时间。

(3) SetMailslotInfo。

另外在邮件被创建后，仍然可以修改读操作时间、超时时间，通过 API SetMailslotInfo 函数设置。

```
BOOL WINAPI SetMailslotInfo(
  __in         HANDLE hMailslot,
  __in         DWORD lReadTimeout
);
```

3．实例 11-1　邮槽通信服务端程序

本实例中创建了一个名为“\\\\.\\mailslot\\sample_mailslot”的邮槽，然后使用 GetMailslotInfo 函数读取消息数量。如果没有消息，则等待 20s 获取信息。循环直至有消

息到达，然后一次性读出全部消息。

```
/* ************************************
* mlsrv.c 通过mailslot进行进程间通信
**************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
/* 全局变量 */
HANDLE hSlot;
LPTSTR lpszSlotName = TEXT("\\\\.\\mailslot\\sample_mailslot");
LPTSTR Message = TEXT("Message for mailslot in primary domain.");
/* ************************************
* void main()
* 功能:进程间mailslot通信客户端
**************************************/
void main()
{
     DWORD cbMessage, cMessage, cbRead;
     BOOL fResult;
     LPTSTR lpszBuffer;
     TCHAR achID[80];
     DWORD cAllMessages;
     HANDLE hEvent;
     OVERLAPPED ov;
     cbMessage = cMessage = cbRead = 0;
     hSlot = CreateMailslot(
          lpszSlotName,                          // mailslot 名
          0,                                     // 不限制消息大小
          MAILSLOT_WAIT_FOREVER,                 // 无超时
          (LPSECURITY_ATTRIBUTES) NULL);
     if (hSlot == INVALID_HANDLE_VALUE)
     {
          printf("CreateMailslot failed with %d\n", GetLastError());
          return ;
     }
     else printf("Mailslot created successfully.\n");
     while(1)
     {
          // 获取mailslot信息
          fResult = GetMailslotInfo(hSlot, // mailslot 句柄
               (LPDWORD) NULL,                   // 无最大消息限制
```

```
                &cbMessage,                    // 下一条消息的大小
                &cMessage,                     // 消息的数量
                (LPDWORD) NULL);               // 无时限
            if (!fResult)
            {
                printf("GetMailslotInfo failed with %d.\n", GetLastError());
                return ;
            }
            if (cbMessage == MAILSLOT_NO_MESSAGE)
            {
                // 没有消息，过一段时间再去读
                Sleep(20000);
                continue;
            }
            cAllMessages = cMessage;
            while (cMessage != 0)  // 获取全部消息，有可能不止一条
            {
                // 提示信息
                wsprintf((LPTSTR) achID,
                    "\nMessage #%d of %d\n",
                    cAllMessages - cMessage + 1,
                    cAllMessages);
                // 分配空间
                lpszBuffer = (LPTSTR) HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,
                    lstrlen((LPTSTR) achID)*sizeof(TCHAR) + cbMessage);
                if( NULL == lpszBuffer )
                {
                    return ;
                }
                // 读取消息
                fResult = ReadFile(hSlot,  // mailslot 句柄
                    lpszBuffer,            // 缓存
                    cbMessage,             // 消息的长度
                    &cbRead,               // 实际读取的长度
                    NULL);
                if (!fResult)
                {
                    printf("ReadFile failed with %d.\n", GetLastError());
                    GlobalFree((HGLOBAL) lpszBuffer);
                    return ;
                }
                // 处理信息，显示
                lstrcat(lpszBuffer, (LPTSTR) achID);
                printf("Contents of the mailslot: %s\n", lpszBuffer);
                HeapFree(GetProcessHeap(),0,lpszBuffer);
                // 计算剩余的消息数
                fResult = GetMailslotInfo(hSlot,
                    (LPDWORD) NULL,
                    &cbMessage,
                    &cMessage,
                    (LPDWORD) NULL);
                if (!fResult)
                {
                    printf("GetMailslotInfo failed (%d)\n", GetLastError());
                    return ;
                }
            }
        }
        return ;
    }
```

这种方式只有在固定的时间间隔才读取，如果需要在有消息到达时立刻获得消息，可以将读操作超时时间设置为 MAILSLOT_WAIT FOREVER，然后调用 ReadFile，ReadFile 在有消息可读时才会读取消息并返回。

11.1.2 通过邮槽发送消息

客户端向邮槽发送消息需要首先打开邮槽，然后直接使用 WriteFile 函数写入。打开邮槽使用 CreateFile 函数。客户端在打开邮槽时需要知道邮槽名。

1．实例 11-2 邮槽通信服务端程序

本实例打开了名为\\\\.\\mailslot\\sample_mailslot 的邮槽，然后写入了一条消息。消息的内容是“Test Message for mailslot”。

```
/* ************************************
* mlclt.c
* 11.1 通过 mailslot 进行进程间通信
**************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
/* 全局变量 */
HANDLE hSlot;
LPTSTR lpszSlotName = TEXT("\\\\.\\mailslot\\sample_mailslot");       // mailslot 名
LPTSTR lpszMessage = TEXT("Test Message for mailslot ");              // 通信的内容

/* ************************************
* void main()
* 功能    进程间 mailslot 通信客户端
**************************************/
void main()
{
     BOOL fResult;
     HANDLE hFile;
     DWORD cbWritten;
     DWORD cbMessage;
     // 打开 mailslot
     hFile = CreateFile(lpszSlotName,
          GENERIC_WRITE,          // 可写
          FILE_SHARE_READ,
          (LPSECURITY_ATTRIBUTES) NULL,
          OPEN_EXISTING,          // 打开一个已经存在的 mailslot，应该由服务端已经创建
          FILE_ATTRIBUTE_NORMAL,
          (HANDLE) NULL);
     if (hFile == INVALID_HANDLE_VALUE)
     {
          printf("CreateFile failed with %d.\n", GetLastError());
          return ;
     }
     // 向 mailslot 写入
     fResult = WriteFile(hFile,
          lpszMessage,
          (DWORD) (lstrlen(lpszMessage)+1)*sizeof(TCHAR),
          &cbWritten,
          (LPOVERLAPPED) NULL);
     if (!fResult)
     {
          printf("WriteFile failed with %d.\n", GetLastError());
          return ;
     }
     // 结束
     printf("Slot written to successfully.\n");

     CloseHandle(hFile);
     return ;
}
```

2．实例 11-1、11-2 运行结果

进行服务端程序，再运行客户端程序（可运行多次），客户端程序运行结束后退出，服务端程序输出如图 11-1 所示。

图 11-1　邮槽通信实例运行结果

11.2　管道(Pipe)

管道是一种用于在进程间共享数据的机制，其实质是一段共享内存。Windows 系统为这段共享的内存设计采用数据流 I/O 的方式来访问。由一个进程读、另一个进程写，类似于一个管道两端，因此这种进程间的通信方式称作“管道”。

管道分为匿名管道和命名管道。

匿名管道只能在父子进程间进行通信，不能在网络间通信，而且数据传输是单向的，只能一端写，另一端读。

命令管道可以在任意进程间通信，通信是双向的，任意一端都可读可写，但是在同一时间只能有一端读、一端写。

由于命令管道更具有通用性，本节介绍命名管道。

11.2.1　创建命名管道

创建命名管道有专门的 API 函数。

1．关键 API

CreateNamedPiDe 是用于创建命令管道的 API 函数，原型如下：

```
HANDLE WINAPI CreateNamedPipe(
  __in        LPCTSTR lpName,
  __in        DWORD dwOpenMode,
  __in        DWORD dwPipeMode,
  __in        DWORD nMaxInstances,
  __in        DWORD nOutBufferSize,
  __in        DWORD nInBufferSize,
  __in        DWORD nDefaultTimeOut,
  __in        LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

◇参数

lpName 参数：管道名。

命名方式如下所示：

```
\\ServerName\pipe\PipeName
\\.\pipe\PipeName
```

dwOpenMode 参数是打开模式。

包括存取模式、I／0 模式等，其值如表 11.1 所示。

表 11.1　　管道的模式

模　式	意　义
PIPE_ACCESS_INBOUND	相当于 GENERIC_READ 服务端只读
PIPE_ACCESS_OUTBOUND	相当于 GENERIC_WRITE 服务端只写
PIPE_ACCESS_DUPLEX	相当于 GENERIC_READ | GENERIC_WRITE 可读可写
FILE_FLAG_OVERLAPPED	采用重叠 I/O 模式（异步 I/O 模式的一种）
FILE_FLAG_WRITE_THROUGH	只对不同主机间的通信造成影响，如果设置 FILE_FLAG_WRITE_THROUGH，那么在写入数据时，当数据全部通过网络发送后才会返回

dwPipeMode 参数：管道模式，包括管道类型模式、管道读模式和管道等待模式。管道类型模式有 PIPE_TYPE_ BYTE 或 PIPE_TYPE MESSAGE，如果设置为前一种模式，那么所有写入管道的数据都是以连接字节的形式存在，而后一个模式，系统会将每一次调用 WriteFile 写入管理的数据作为一个消息，管道中的内容是分块的。管道读模式可以是 PIPE_-READMODE_ BYTE 或者
PIPE_ READMODE MESSAGE，分别表示从管道中读取的信息是连续的还是以独立消息的形式存在的。等待模式可以是 PIPE_WAIT 或者 PIPE__ NOWAIT，分别表示读写操作是否阻塞（是否在无数据时立刻返回）。

nMaxInstances 参数：表示指定了一个 pipe 所能建立的最多的实例，同名的 pipe 可以被创建多次，一次称作一个实例，每个实例可以接收一个连接。

nOutBufferSize 和 nInBufferSize:分别是输出和输入的缓存大小。

nDefaultTimeOut:超时时间。

lpSecurityAttributes:安全属性。

◇返回值

返回创建的管道的句柄。

2．实例 11-3 创建命令管道

以下代码创建了一个名为“\\\\.\\pipe\\samplenamedpipe”的管道，这是一个用于本机内两个进程通信的管道。设置了存取模式为可读可写，而且采用了异步重叠模式(FILE FLAG_OVERLAPPED)，管道类型为消息型，读取时也是按消息读取。

说明　I/O 的模式将会在本章 11.7 节专门介绍。

```
/* ************************************
* 功能     建立连接实例
* 参数     lpoOverlap, 用于 overlapped IO 的结构
```

```
* 返回值  是否成功
************************************/
BOOL CreateAndConnectInstance(LPOVERLAPPED lpoOverlap)
{
     LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\samplenamedpipe");
     // 创建named pipe
     hPipe = CreateNamedPipe(
          lpszPipename,                    // pipe 名
          PIPE_ACCESS_DUPLEX |             // 可读可写
          FILE_FLAG_OVERLAPPED,            // overlapped 模式
          // pipe模式
          PIPE_TYPE_MESSAGE |              // 消息类型 pipe
          PIPE_READMODE_MESSAGE |          // 消息读模式
          PIPE_WAIT,                       // 阻塞模式
          PIPE_UNLIMITED_INSTANCES,        // 无限制实例
          BUFSIZE*sizeof(TCHAR),           // 输出缓存大小
          BUFSIZE*sizeof(TCHAR),           // 输入缓存大小
          PIPE_TIMEOUT,                    // 客户端超时
          NULL);                           // 默认安全属性
     if (hPipe == INVALID_HANDLE_VALUE)
     {
          printf("CreateNamedPipe failed with %d.\n", GetLastError());
          return 0;
     }
     // 连接到新的客户端
     return ConnectToNewClient(hPipe, lpoOverlap);
}
```

代码中的CreateAndConnectInstance函数最后调用了ConnectToNewClient函数监听客户端的连接。ConnectToNewClient 函数实例代码见 11.2.2 小节。

11.2.2　管道监听

ConnectToNewClient 函数功能是监听客户端的连接，调用了 API 函数 ConnectNamedPipe。

1．关键 API

ConnectNamedPipe 函数是 pipe 服务端用于等待客户端连接的函数，函数原型如下：

```
BOOL WINAPI ConnectNamedPipe(
  __in        HANDLE hNamedPipe,
  __in        LPOVERLAPPED lpOverlapped
);
```

参数 hNamedPipe 是 pipe 句柄，直接使用 CreateNamedPipe 函数的返回值即可。

lpOverlapped 是指向 OVERLAPPED 结构的指针，如果在创建管道时指明了 I/O 模式为重叠的(lpOverlapped)，那么必须指定本参数；否则设置为 NULL。

2．实例 11-4 创建命令管道

本实例调用了 ConnectNamedPipe API 函数。当有客户连接时，函数返回，之后根据返回值判断状态。

```
/* ************************************
* BOOL ConnectToNewClient(HANDLE hPipe, LPOVERLAPPED lpo)
* 功能：建立连接实例
* 参数：lpoOverlap，用于overlapped IO的结构
* 返回值：是否成功
************************************/
BOOL ConnectToNewClient(HANDLE hPipe, LPOVERLAPPED lpo)
{
     BOOL fConnected, fPendingIO = FALSE;
```

```
    // 开始一个 overlapped 连接
    fConnected = ConnectNamedPipe(hPipe, lpo);
    if (fConnected)
    {
        printf("ConnectNamedPipe failed with %d.\n", GetLastError());
        return 0;
    }
    switch (GetLastError())
    {
        // overlapped连接进行中
    case ERROR_IO_PENDING:
        fPendingIO = TRUE;
        break;
        // 已经连接，因此Event未置位
    case ERROR_PIPE_CONNECTED:
        if (SetEvent(lpo->hEvent))
            break;
        // error
    default:
        {
            printf("ConnectNamedPipe failed with %d.\n", GetLastError());
            return 0;
        }
    }
    return fPendingIO;
}
```

11.2.3 使用异步I/O进行读写

本书前面章节在使用到 I/O 时，例如文件 I/O 时，都是使用同步 I/O 模式，在使用异步 I/O 时，先了解以下基本概念。

1. 110 模式

由于一些进程间通信的方法(pipe)涉及到了 Windows 系统的 I/O 模式。I]O 模式不仅在进程间通信时使用，任何具有数据流形式的输入输出（包括文件输入输出、内核通信、网络输入输出等）都涉及 I/O 模式。

异步(Asynchronous)和同步(Synchronous) I/O 是两种基本的 I/O 模式。其中涉及许多细节，同步 I/O 和异步 I / O 的详细区别，本书将在 16.4 节介绍。

由于第 4 章、第 14 章和第 16 章分别介绍了文件 I/O 和网络 I/O 和设备 I/O(内核通信)。在学习这部分内容时可以和相关章节相联系。

2. 异步 I/O

所谓同步 I/O 是指在调用 ReadFile、WriteFile 等函数进行输入输出操作时，系统完成了输入输出 ReedFile、WriteFile 才返回。在操作系统进行 I/O 操作的过程上，用户态线程不能执行，因此在同步 I/O 时，如果需要在 I / O 时进行其他操作就只能再开启线程。

而异步 I/O 是在调用 ReadFile、WriteFile 等函数后，函数立即返回，线程可以进行其他操作。

剩下的 I/O 操作在系统内核中自动完成。那么在系统内核完成输入输出后，程序如何知道 I/O 是否已完成?

在本实例中采用了一种方法，称作完成函数（Completion Routines），如果使用 ReadFileEx、WriteFileEx 等进行 I/O，可以指定完成函数，所谓完成函数是指内核在完成 I/O 后，内核会回调这个函数。当完成函数被调用时，就指明内核已经完成了 I/O，程序可以在这个函数中进行一个 I/O 完成后需要的操作（例如释放内存）。

实例 11-5 就使用了这种方式。

3. 实例 11-5 对 Pipe 进行读写

读和写操作的完成函数如下所示：

```
/* ************************************
* CompletedWriteRoutine
*    写入 pipe 操作的完成函数
*    接口参见 FileIOCompletionRoutine 回调函数定义
*    当写操作完成时被调用，开始读另外一个请求
**************************************/
VOID WINAPI CompletedWriteRoutine(
                                          DWORD dwErr,
                                          DWORD cbWritten,
                                          LPOVERLAPPED lpOverLap)
{
     LPPIPEINST lpPipeInst;
     BOOL fRead = FALSE;
     // 保存 overlap 实例
     lpPipeInst = (LPPIPEINST) lpOverLap;
     // 如果没有错误
     if ((dwErr == 0) && (cbWritten == lpPipeInst->cbToWrite))
     {
          fRead = ReadFileEx(
          lpPipeInst->hPipeInst,
          lpPipeInst->chRequest,
          BUFSIZE*sizeof(TCHAR),
          (LPOVERLAPPED) lpPipeInst,
          // 写读操作完成后，调用 CompletedReadRoutine
          (LPOVERLAPPED_COMPLETION_ROUTINE) CompletedReadRoutine);
     }
     if (! fRead)
          // 出错，断开连接
          DisconnectAndClose(lpPipeInst);
}
/* ************************************
* CompletedReadRoutine
*    读取 pipe 操作的完成函数
*    接口参见 FileIOCompletionRoutine 回调函数定义
*    当读操作完成时被调用，写入回复
**************************************/
VOID WINAPI CompletedReadRoutine(
                                        DWORD dwErr,
                                        DWORD cbBytesRead,
                                        LPOVERLAPPED lpOverLap)
{
     LPPIPEINST lpPipeInst;
     BOOL fWrite = FALSE;
     // 保存 overlap 实例
     lpPipeInst = (LPPIPEINST) lpOverLap;
     // 如果没有错误
     if ((dwErr == 0) && (cbBytesRead != 0))
     {
          // 根据客户端的请求，生成回复
          GetAnswerToRequest(lpPipeInst);
          // 将回复写入 pipe
          fWrite = WriteFileEx(
               lpPipeInst->hPipeInst,
               lpPipeInst->chReply,      //将响应写入 pipe
               lpPipeInst->cbToWrite,
               (LPOVERLAPPED) lpPipeInst,
               // 写入完成后，调用 CompletedWriteRoutine
               (LPOVERLAPPED_COMPLETION_ROUTINE) CompletedWriteRoutine);
     }
     if (! fWrite)
          // 出错，断开连接
          DisconnectAndClose(lpPipeInst);
}
```

在读操作完成后，调用读操作的完成函数 CompletedReadRoutine。而 CompletedReadRoutine 函数又开始了新的写操作。

在 CompletedReadRoutine 函数进行的写操作 I/O 完成后，写操作的完成函数 CompletedWriteRoutine 被调用，而正是在 CompletedWriteRoutine 中进行的写操作。

如果循环往复，形成交替读写的流程。

下面再来分析读写操作循环的入口点。分析 maIn 函数：

```
/* 常量 */
#define PIPE_TIMEOUT 5000
#define BUFSIZE 4096
/* 结构定义 */
typedef struct
{
     OVERLAPPED oOverlap;
     HANDLE hPipeInst;
     TCHAR chRequest[BUFSIZE];
     DWORD cbRead;
     TCHAR chReply[BUFSIZE];
     DWORD cbToWrite;
} PIPEINST, *LPPIPEINST;
/* 函数声明 */
VOID DisconnectAndClose(LPPIPEINST);
BOOL CreateAndConnectInstance(LPOVERLAPPED);
BOOL ConnectToNewClient(HANDLE, LPOVERLAPPED);
VOID GetAnswerToRequest(LPPIPEINST);
VOID WINAPI CompletedWriteRoutine(DWORD, DWORD, LPOVERLAPPED);
VOID WINAPI CompletedReadRoutine(DWORD, DWORD, LPOVERLAPPED);
/* 全局变量 */
HANDLE hPipe;
/* ************************************
* int main(VOID)
* 功能: pipe 通信服务端主函数
************************************/
int main(VOID)
{
     HANDLE hConnectEvent;
     OVERLAPPED oConnect;
     LPPIPEINST lpPipeInst;
     DWORD dwWait, cbRet;
     BOOL fSuccess, fPendingIO;
     // 用于连接操作的事件对象
     hConnectEvent = CreateEvent(
          NULL,    // 默认属性
          TRUE,    // 手工 reset
          TRUE,    // 初始状态 signaled
          NULL);   // 未命名
     if (hConnectEvent == NULL)
     {
          printf("CreateEvent failed with %d.\n", GetLastError());
          return 0;
```

```
        }
        // OVERLAPPED 事件
        oConnect.hEvent = hConnectEvent;
        // 创建连接实例，等待连接
        fPendingIO = CreateAndConnectInstance(&oConnect);
        while (1)
        {
             // 等待客户端连接或读写操作完成
             dwWait = WaitForSingleObjectEx(
                  hConnectEvent,  // 等待的事件
                  INFINITE,       // 无限等待
                  TRUE);
             switch (dwWait)
             {
             case 0:
                  // pending
                  if (fPendingIO)
                  {
                       // 获取 Overlapped I/O 的结果
                       fSuccess = GetOverlappedResult(
                            hPipe,     // pipe 句柄
                            &oConnect, // OVERLAPPED 结构
                            &cbRet,    // 已经传送的数据量
                            FALSE);    // 不等待
                       if (!fSuccess)
                       {
                            printf("ConnectNamedPipe (%d)\n", GetLastError());
                            return 0;
                       }
                  }
                  // 分配内存
                  lpPipeInst = (LPPIPEINST) HeapAlloc(GetProcessHeap(),0,sizeof(PIPEINST));
                  if (lpPipeInst == NULL)
                  {
                       printf("GlobalAlloc failed (%d)\n", GetLastError());
                       return 0;
                  }
                  lpPipeInst->hPipeInst = hPipe;
                  // 读和写，注意 CompletedWriteRoutine 和 CompletedReadRoutine 的相互调用
                  lpPipeInst->cbToWrite = 0;
                  CompletedWriteRoutine(0, 0, (LPOVERLAPPED) lpPipeInst);
                  // 再创建一个连接实例，以响应下一个客户端的连接
                  fPendingIO = CreateAndConnectInstance(
                       &oConnect);
                  break;
                  // 读写完成
             case WAIT_IO_COMPLETION:
                  break;
             default:
                  {
                       printf("WaitForSingleObjectEx (%d)\n", GetLastError());
                       return 0;
                  }
             }
        }
        return 0;
}
```

在 maln 函数中进行了一次 CompletedWriteRoutine 函数调用。这次调用开始了循环交替读写。

在调用 CompletedWriteRoutine 之前，mam 函数还进行若干与重叠 I/O 相关的操作。

CompletedReadRoutine 函数调用了 GetAnswerToRequest 函数，GetAnswerToRequest 函数用于处理不同的客户端输入，可以根据不同输入给出不同的反馈。在本实例中，对所有输入都回复"Default answer from server"。

```
// TODO 根据客户端的请求，给出响应
VOID GetAnswerToRequest(LPPIPEINST pipe)
{
    _tprintf( TEXT("[%d] %s\n"), pipe->hPipeInst, pipe->chRequest);
    lstrcpyn( pipe->chReply,  TEXT("Default answer from server") ,BUFSIZE);
    pipe->cbToWrite = (lstrlen(pipe->chReply)+1)*sizeof(TCHAR);
}
```

各函数之间的参数传递主要是通过 PIPEINST 结构传递的，定义在 maln 函数之前。

11.2.4 关闭管道实例

如果在读写过程中发生了错误，服务端程序会断开管道的连接，并关闭实例句柄。关闭连接使用 API 函数 DisconnectNamedPipe，参数为实例句柄。

```
/* ************************************
* VOID DisconnectAndClose(LPPIPEINST lpPipeInst)
* 功能：断开一个连接的实例
* 参数：lpPipeInst，断开并关闭的实例句柄
**************************************/
VOID DisconnectAndClose(LPPIPEINST lpPipeInst)
{
    // 关闭连接实例
    if (! DisconnectNamedPipe(lpPipeInst->hPipeInst) )
    {
        printf("DisconnectNamedPipe failed with %d.\n", GetLastError());
    }
    // 关闭 pipe 实例的句柄
    CloseHandle(lpPipeInst->hPipeInst);
    // 释放
    if (lpPipeInst != NULL)
        HeapFree(GetProcessHeap(),0, lpPipeInst);
}
```

11.2.5 客户端

与服务端相比，客户端实例程序的功能相对较简单，首先是打开命名管道，获得句柄，然后向管道中写入数据，之后等待服务端的回复。

1．原理与关键 API

在实例 11-6 中，首先有一个 while 循环，本循环在客户端成功打开命名管道（管道名需与服务端所创建的一致）后结束，如果打开管道时管道繁忙，将等待一段时间再连接，或者在程序发生错误时直接退出。

打开命名管道使用的函数是 CreateFile。调用 CreateFile 时服务端会收到客户端的连接请求，打开成功后会连接管道。

等待一个管道变为可用状态的 API 函数是 WaitNamedPipe。

```
BOOL WINAPI WaitNamedPipe(
  __in        LPCTSTR lpNamedPipeName,
  __in        DWORD nTimeOut
);
```

WaitNamedPipe 函数一直等待，直到管道实例可用或者超时。

如果程序成功连接到服务端，使用 SetNamedPipeHandleState 函数设置管道的读写模式。由于客户端使用 CreateFile 函数打开管道，而在打开时无法指定读写模式，因此在打开成功后，需要再调用 SetNamedPipeHandleState 函数设置读写模式。实例中设置为以消息方式读。

SetNamedPipeHandleState 函数原型如下：

```
BOOL WINAPI SetNamedPipeHandleState(
  __in        HANDLE hNamedPipe,
  __in        LPDWORD lpMode,
  __in        LPDWORD lpMaxCollectionCount,
  __in        LPDWORD lpCollectDataTimeout
);
```

hNamedPipe 参数是管道句柄；lpMode 用于设置管道的模式；lpMaxCollectionCount 和 lpCollectDataTimeout 用于设置数据的缓存大小，在同一主机进程间通信时无意义。

2．实例 11-6 管道通信客户端程序

管道通信和客户端实现代码如下：

```
/* ************************************
* PipeClnt.c 通过pipe进行进程间通信
**************************************/
/* 头文件 */
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <tchar.h>
/* 常量 */
#define BUFSIZE 512
/* ************************************
* int main(VOID)
* 功能: pipe 通信服务端主函数
**************************************/
int main(int argc, TCHAR *argv[])
{
     HANDLE hPipe;
     LPTSTR lpvMessage=TEXT("Default message from client");
     TCHAR chBuf[BUFSIZE];
     BOOL fSuccess;
     DWORD cbRead, cbWritten, dwMode;
     LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\samplenamedpipe");

     if( argc > 1 )      // 如果输入了参数，则使用输入的参数
          lpvMessage = argv[1];
     while (1)
     {
          // 打开一个命名pipe
          hPipe = CreateFile(
               lpszPipename,   // pipe 名
               GENERIC_READ |  GENERIC_WRITE,          //  可读可写
               0,             // 不共享
               NULL,          // 默认安全属性
               OPEN_EXISTING,  // 已经存在(由服务端创建)
               0,             // 默认属性
               NULL);
          if (hPipe != INVALID_HANDLE_VALUE)
               break;
          // 如果不是 ERROR_PIPE_BUSY 错误，直接退出
          if (GetLastError() != ERROR_PIPE_BUSY)
          {
               printf("Could not open pipe");
```

```
            return 0;
        }
        // 如果所有 pipe 实例都处于繁忙状态，等待 2s。
        if (!WaitNamedPipe(lpszPipename, 2000))
        {
            printf("Could not open pipe");
            return 0;
        }
    }
    // pipe 已经连接，设置为消息读状态
    dwMode = PIPE_READMODE_MESSAGE;
    fSuccess = SetNamedPipeHandleState(
        hPipe,              // 句柄
        &dwMode,            // 新状态
        NULL,               // 不设置最大缓存
        NULL);              // 不设置最长时间
    if (!fSuccess)
    {
        printf("SetNamedPipeHandleState failed");
        return 0;
    }
    // 写入 pipe
    fSuccess = WriteFile(
        hPipe,                  // 句柄
        lpvMessage,             // 写入的内容
        (lstrlen(lpvMessage)+1)*sizeof(TCHAR), // 写入内容的长度
        &cbWritten,             // 实际写的内容
        NULL);                  // 非 overlapped
    if (!fSuccess)
    {
        printf("WriteFile failed");
        return 0;
    }
    do
    {
        // 读回复
        fSuccess = ReadFile(
            hPipe,    // 句柄
            chBuf,    // 读取内容的缓存
            BUFSIZE*sizeof(TCHAR),  // 缓存大小
            &cbRead,  // 实际读的字节
            NULL);    // 非 overlapped
        if (! fSuccess && GetLastError() != ERROR_MORE_DATA)
            break; //失败，退出
        _tprintf( TEXT("%s\n"), chBuf ); // 打印读的结果
    } while (!fSuccess);  //  ERROR_MORE_DATA 或者成功则循环
    getch();//任意键退出
    // 关闭句柄
    CloseHandle(hPipe);
    return 0;
}
```

3．运行结果

先运行服务端程序，再运行客户端程序。客户端程序可以运行多次。

运行两次服务端程序，程序运行结果如图 11-2 所示。

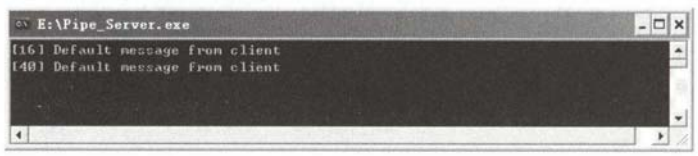

图 11-2　服务端程序运行结果

客户端程序运行结果如图 11-3 所示。

图 11-3　客户端程序运行结果

11.3　剪贴板

剪贴板是最常用的系统特性之一。计算机用户在使用计算机时经常会使用到剪贴板。剪贴板由系统定义，并不属于任何一个特定的进程。系统中所有进程都可以访问和设置剪贴板。很多具有“复制”、“粘贴”功能的文字编译程序都是通过剪贴板进行操作的。

剪贴板最大的特点就是数据传输没有明确的目标，数据是被动访问的，设置剪贴板的程序并不知道所设置的数据什么时候被访问，也不知道被哪个进程访问，剪贴板中的内容也可以多次访问，直到新的数据写入。正是此特性决定了剪贴板便于计算机用户进行直接操作。

因此通过剪贴板也是一种可供选择的进程间通信方式。但是由于系统中任何一个进程都可以无限制地访问剪贴板，剪贴板也是一种不可靠的通信方式。

本节中 11.3.1 小节通过实例介绍了对剪贴板的基本操作，包括设置剪贴板中的数据、从剪贴板获取数据等。

11.3.2 小节通过实例介绍了如何对剪贴板的修改进行监视，这种机制可以使用程序及时发现剪贴板中的数据变化。使用通过剪贴板传输数据可以具有实时性。

11.3.3 小节介绍了剪贴板中数据的格式及自定义数据格式的方法。

11.3.1　获取、设置剪贴板数据

系统提供了专门的 API 函数来设置和获取剪贴板，包括 OpenClipboard、EmptyClipboard、SetClipboardData、GetClipboardData、CloseClipboard 等。

1．关键 API

(1) OpenClipboard。

OpenClipboard 函数的功能是打开剪贴板，获得剪贴板的句柄。后续所有对剪贴板的操作都需要该句柄。函数原型如下：

```
BOOL OpenClipboard(
        HWND hWndNewOwner
);
```

参数 hWndNewOwner 指定了与剪贴板句柄相配合的窗口句柄。指定的窗口可以收到剪贴板操作所产生的消息（参见 11.3.2 小节），如果指定 NULL 则采用当前任务的窗口。

(2) EmptyClipboard。

在得到剪贴板句柄后，可以对剪贴板进行操作。EmptyClipboard 的功能是清空剪贴板中的内容，无参数。

```
BOOL EmptyClipboard( VOID );
```

(3) SetClipboardData。

SetClipboardData 函数的功能是设置剪贴板内容，函数原型如下：

```
HANDLE SetClipboardData(
    UINT uFormat,
    HANDLE hMem
);
```

uFormat 参数指定了数据的格式，可以是标准系统格式，也可以是用户自定义格式。剪贴板的数据格式见 11.3.3 小节。

hMem 是需要设置的数据的内存句柄。需使用全局内存管理的函数分配和设置，且在分配时需指定 GMEM-一 MOVEABLE 标志（参考第 5 章）。

(4) GetClipboardData。

GetCliDboardData 函数的功能是从剪贴板获取数据，函数原型如下：

```
HANDLE GetClipboardData(
    UINT uFormat
);
```

参数 uFormat 指定获取的数据的格式。获取的数据使用句柄返回，根据设置的格式不同，返回的数据类型不同。如果指定为 CF_TEXT，返回的是字符串指针。

2．实例 11-7 管道通信客户端程序

本实例 ClipText.exe 演示了对文本复制、粘贴等操作。程序包括一个窗口、一组菜单。菜单中包括“复制”、“剪切”、“粘贴”等项。在单击菜单后作相应的处理。

程序会在窗口客户区显示文本，显示的文本由全局变量 lpszText 指定。在窗口消息处理函数每次收到窗口绘制(WM_PAINT)函数时，会调用 DrawText API 函数将字符串显示在客户区（原理参考第 13 章 GDI 相关内容）。

如果通过菜单输入“复制”(命令 ID 为 IDM COPY)，程序调用 SetClipboardData 函数将窗口上显示的内容(lpszText)设置到剪贴板，由于本程序只支持文本操作，所以将格式设置为 CF_TEXT。如果输入的命令是“剪切”(命令 ID 为 IDM_CUT)，除了完成复制的功能处，还将 lpszText 字符串释放。

如果通过菜单输入“粘贴”（命令 ID 为 IDM_PASTE)，程序调用 GetClipboardData，从剪贴板中获得数据，赋值给 lpszText。然后在界面上显示内容。

为了达到显示的效果，本实例直接使用 GDI 接口函数在主窗口的客户区上绘制出文本，而没有采用本身就具有剪贴板操作功能的 EDIT、Rich Edit 等控件。

```
/* ************************************
* ClipText.c 操作剪贴板
**************************************/
/* 头文件 */
#include "cliptext.h"
#include <string.h>
/* 预处理 */
#ifdef UNICODE// UNICODE
#define MY_CLIP_FORMAT CF_UNICODETEXT
#else
#define MY_CLIP_FORMAT CF_TEXT
#endif
```

```
/* 全局变量 */
HINSTANCE hInst;
HACCEL hAccTable;
HWND hwnd;
LPTSTR lpszText = NULL;
LPTSTR lpData = NULL; // 剪贴板数据指针
HANDLE hHeap;
LPTSTR szInitialClientAreaText =
TEXT("This program demonstrates the use of the Edit menu to copy and \n")
TEXT("paste text to and from the clipboard. Try using the Copy command \n")
TEXT("to move this text to the clipboard, and the Paste command to replace \n")
TEXT("this text with data from another application. \r\n\r\n")
TEXT("You might want to try running Notepad and Clipbrd alongside this \n")
TEXT("application so that you can watch the data exchanges take place. \n");

/* 函数声明 */
BOOL InitApplication(HANDLE);
BOOL InitInstance(HANDLE, INT);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK About(HWND, UINT, WPARAM, LPARAM);
VOID OutOfMemory(VOID);

/* ************************************
* WinMain
************************************/
int WINAPI WinMain(
                    HINSTANCE hinstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow)
{
     MSG msg;
     BOOL fGotMessage;
     // InitApplication
     if (!InitApplication(hinstance))
     {
          return FALSE;
     }
     // InitInstance
     if (!InitInstance(hinstance, nCmdShow))
     {
          return FALSE;
     }
     // 消息循环，只处理本窗口的消息
     while ((fGotMessage = GetMessage(&msg, (HWND) hwnd, 0, 0)) != 0 && fGotMessage != -1)
     {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }
     return msg.wParam;
     UNREFERENCED_PARAMETER(lpCmdLine);
}

/* ************************************
* BOOL InitApplication(HINSTANCE hinstance)
* 功能：注册主窗口类
* 参数：hinstance，应用程序本次运行实例
* 返回值：是否成功
************************************/
```

```
BOOL InitApplication(
                        HANDLE hInstance)
{
     WNDCLASS wc;
     wc.lpfnWndProc = (WNDPROC) WndProc;
     wc.hInstance = hInstance;
     wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
     wc.hCursor = LoadCursor(NULL, IDC_ARROW);
     wc.hbrBackground = (HBRUSH)(COLOR_WINDOW);
     wc.lpszMenuName = NULL;
     wc.lpszClassName = TEXT("CliptextWClass");
     wc.style = 0 ;
     wc.cbClsExtra =0;
     wc.cbWndExtra =0;
     return (RegisterClass(&wc));
}

/* ************************************
* BOOL InitInstance(HINSTANCE hinstance, int nCmdShow)
* 功能：创建主窗口
* 参数：hinstance，应用程序本次运行实例
* nCmdShow：如何显示
* 返回值：是否成功
**************************************/
BOOL InitInstance(
                    HINSTANCE hInstance,
                    INT nCmdShow)
{
     // 保存实例
     hInst = hInstance;
     // 保存堆句柄，后面的程序分配内存
     hHeap = GetProcessHeap();
     // Accelerators
     hAccTable = LoadAccelerators(hInst, TEXT("ClipTextAcc"));
     // 为 lpszText 分配内存，lpszText 是用于在界面上显示的文本
     lpszText = HeapAlloc(hHeap,0,lstrlen(szInitialClientAreaText)+1);
     // 复制初始显示的字符串
     lstrcpy(lpszText, szInitialClientAreaText);
     // 创建窗口
     hwnd = CreateWindow(
          TEXT("CliptextWClass"),
          TEXT("Cliptext Sample Application"),
          WS_OVERLAPPEDWINDOW,
          CW_USEDEFAULT, 0,
          CW_USEDEFAULT, 0,
          NULL,
          (HMENU)LoadMenu(hInstance,MAKEINTRESOURCE(ID_CLIP_MENU)),
          hInstance,
          NULL );
     if (!hwnd)
     {
          return FALSE;
     }
     // 显示、刷新窗口
     ShowWindow(hwnd, nCmdShow);
     return UpdateWindow(hwnd);
}

/* ************************************
* WndProc
* 窗口消息处理
```

```
************************************/
LRESULT CALLBACK WndProc(
                              HWND hWnd,
                              UINT message,
                              WPARAM wParam,
                              LPARAM lParam)
{
     HDC hDC = NULL;
     PAINTSTRUCT ps = {0};
     RECT rectClient = {0};
     LPTSTR lpClipData = NULL;
     switch (message)
     {
     case WM_INITMENU: // 初始化菜单
          if (wParam == (WPARAM)GetMenu(hWnd))
          {
               if (OpenClipboard(hWnd))
               {
                    // 格式是否可用
                    if (IsClipboardFormatAvailable(MY_CLIP_FORMAT) ||
                         IsClipboardFormatAvailable(CF_OEMTEXT))
                         // 菜单项
                    {
                         EnableMenuItem((HMENU)wParam, IDM_PASTE, MF_ENABLED);
                    }
                    else
                    {
                         EnableMenuItem((HMENU)wParam, IDM_PASTE, MF_GRAYED);
                    }
                    // 关闭剪贴板
                    CloseClipboard();
                    return (TRUE);
               }
               else /* Clipboard is not available */
               {
                    return (FALSE);
               }
          }
          return (TRUE);
     case WM_COMMAND: // 菜单命令
          switch(LOWORD(wParam))
          {
          case IDM_ABOUT:
               break;
          case IDM_NEW:
          case IDM_OPEN:
          case IDM_SAVE:
          case IDM_SAVEAS:
          case IDM_PRINT:
          case IDM_UNDO:
          case IDM_CLEAR:
               MessageBox (GetFocus (),
                    TEXT("Command not implemented."),
                    TEXT("ClipText Sample Application"),
                    MB_ICONASTERISK | MB_OK);
               // 以上不处理
          case IDM_EXIT: // 退出
               DestroyWindow(hWnd);
               break;
          case IDM_CUT:// 剪切和复制界面上显示的内容
          case IDM_COPY:
               if (lpszText != NULL) // 界面是否有显示的内容
               {
```

```
                if(lpData != NULL)
                {
                    HeapFree(hHeap,0,lpData);
                }
                // 为剪贴板数据分配空间
                lpData = HeapAlloc(hHeap,0,HeapSize(hHeap,0,lpszText));
                // 复制
                lstrcpy(lpData, lpszText);
                if (OpenClipboard(hWnd)) // 打开剪贴板
                {
                    // 设置剪贴板内容
                    EmptyClipboard();
                    SetClipboardData(MY_CLIP_FORMAT, lpData);
                }
                CloseClipboard();// 关闭
                if (LOWORD(wParam) == IDM_CUT)
                {
                    //如果是剪切，清除屏幕内容，设置菜单
                    HeapFree(hHeap,0,lpszText);
                    lpszText = 0;
                    EnableMenuItem(GetMenu (hWnd), IDM_CUT, MF_GRAYED);
                    EnableMenuItem(GetMenu(hWnd), IDM_COPY, MF_GRAYED);
                    InvalidateRect (hWnd, NULL, TRUE);
                    UpdateWindow (hWnd);
                }
            }
            return (TRUE);
        case IDM_PASTE:
            if (OpenClipboard(hWnd))
            {
                // 从剪贴板获得文本
                if (!(lpClipData = GetClipboardData(MY_CLIP_FORMAT)))
                {
                    CloseClipboard();
                    break;
                }
                // 为新的显示文本分配空间
                if(lpszText != NULL)
                {
                    HeapFree(hHeap,0,lpszText);
                    lpszText = NULL;
                }
                lpszText = HeapAlloc(hHeap,0,lstrlen(lpClipData)+1);
                //复制
                lstrcpy(lpszText, lpClipData);
                CloseClipboard();// 关闭
                // 设置菜单项
                EnableMenuItem(GetMenu(hWnd), IDM_CUT, MF_ENABLED);
                EnableMenuItem(GetMenu(hWnd), IDM_COPY, MF_ENABLED);
                // 显示
                InvalidateRect(hWnd, NULL, TRUE);
                UpdateWindow(hWnd);
                return (TRUE);
            }
            else
            {
                return (FALSE);
            }
        }
        break;
    case WM_SIZE:
        InvalidateRect(hWnd, NULL, TRUE);
```

```
            break;
        case WM_PAINT:
            // 将文本显示在界面上
            hDC = BeginPaint (hWnd, &ps);
            if (lpszText != NULL)
            {
                GetClientRect (hWnd, &rectClient);
                DrawText (hDC, lpszText, -1, &rectClient,
                    DT_EXTERNALLEADING | DT_NOPREFIX | DT_WORDBREAK);
            }
            EndPaint (hWnd, &ps);
            break;
        case WM_DESTROY:
            //释放内存，退出
            if (lpszText != NULL)
            {
                HeapFree(hHeap,0,lpszText);
            }
            if(lpData != NULL)
            {
                HeapFree(hHeap,0,lpData);
            }
            PostQuitMessage(0);
            break;
        default:
            return (DefWindowProc(hWnd, message, wParam, lParam));
        }
        return (0);
    }
```

3．运行结果

运行本实例后，可以从其他应用程序复制数据到本程序，也可以从本程序复制数据到其他应用程序。参见图 11-4 和图 11-5。

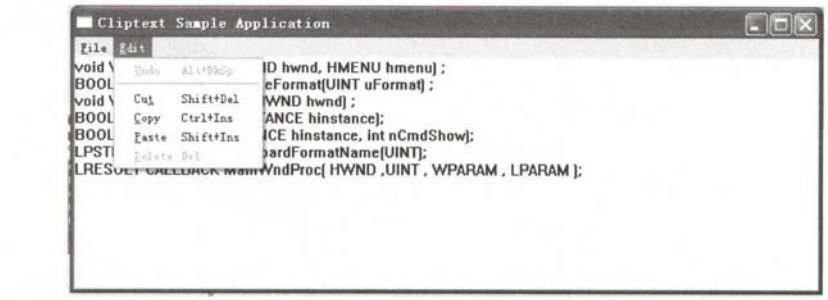

图 11-4　从其他应用程序复制数据到本程序

图 11-5　从本程序复制数据到其他应用程序

11.3.2　监视剪贴板

实例 11-7 中的程序在通过剪贴板进行数据通信时不具有实时性，所有操作都依赖于用户。如果要立即知道剪贴板中的内容变化，需要使用剪贴板查看器(Viewer)。

1．原理与关键 API

(1) WM_DRAWCLIPBOARD 消息。

系统提供了 WM_DRAWCLIPBOARD 消息用于监视剪贴板的变化。如果调用 SetClipboardViewer 函数设置了窗口为剪贴板查看器，那么当剪贴板中的内容变化时，所注册的查看器窗口会收到 WM_CHANGECBCHAIN 消息和 WM_DRAWCLIPBOARD 消息。

当剪贴板中的内容变化时，窗口会收到 WM DRAWCLIPBOARD 消息。当查看器链表中有新的节点加入或有节点退出窗口会收到 WM_CHANGECBCHAIN 消息。

(2) SetClipboardViewer 函数原型如下：

```
HWND SetClipboardViewer(
        HWND hWndNewViewer
);
```

hWndNewViewer 参数指定了监视窗口，也即是用于接收 WM_CHANGECBCHAIN 消息和 WM_DRAWCLIPBOARD 消息的窗口。

2．实例 11-8 剪贴板查看器

本实例 ClipView.exe 是一个剪贴板查看器。在窗口创建时（WM_CREATE 消息），调用了 API 函数 SetClipboardViewer 将程序主窗口设置为查看器。

窗口在收到 WM_CHANGECBCHAIN 时根据情况更新链表。

窗口在收到 WM. DRAWCLIPBOARD 消息后，调用了 SetAutoView 函数，而 SetAutoView 函数，在调用 GetPriorityClipboardFormat API 函数获得剪贴板中数据的主要格式后，调用 UpdateWindow 刷新窗口。

因此，一旦剪贴板变化，ClipView.exe 的窗口就会刷新。而如本实例代码所示，而窗口刷新过程（WM_ PAINT 消息）中，程序会获取剪贴板中的数据，并根据剪贴板数据的格式进行显示。

本实例主要演示了如何监视剪贴板内容的变化；同时还演示了如何使用剪贴板数据格式。数据格式的相关原理解释参见 11.3.3 小节。

```
/* ************************************
* ClipView.c 监视剪贴板
**************************************/
/* 头文件 */
#include <windows.h>
#include <commctrl.h>
#include "resource.h"
/* 全局变量 */
HINSTANCE hinst;
UINT uFormat = (UINT)(-1);
BOOL fAuto = TRUE;
HWND hwnd;
LPSTR szFormatText = "CF_TEXT";
LPSTR szFormatOwnerdiplay = "CF_OWNERDISPLAY";
LPSTR szFormatEnhmetafile = "CF_ENHMETAFILE";
LPSTR szFormatBitMap = "CF_BITMAP";
```

```
/* 函数声明 */
void WINAPI InitMenu(HWND hwnd, HMENU hmenu) ;
BOOL WINAPI IsDisplayableFormat(UINT uFormat) ;
void WINAPI SetAutoView(HWND hwnd) ;
BOOL InitApplication(HINSTANCE hinstance);
BOOL InitInstance(HINSTANCE hinstance, int nCmdShow);
LPSTR GetPredefinedClipboardFormatName(UINT);
LRESULT CALLBACK MainWndProc( HWND ,UINT , WPARAM , LPARAM );

/* ************************************
* WinMain
************************************/
int WINAPI WinMain(
                        HINSTANCE hinstance,
                        HINSTANCE hPrevInstance,
                        LPSTR lpCmdLine,
                        nt nCmdShow)
{
     MSG msg;
     BOOL fGotMessage;
     // InitApplication
     if (!InitApplication(hinstance))
     {
          return FALSE;
     }
     // InitInstance
     if (!InitInstance(hinstance, nCmdShow))
     {
          return FALSE;
     }
     // 消息循环
     while ((fGotMessage = GetMessage(&msg, (HWND) NULL, 0, 0)) != 0 && fGotMessage != -1)
     {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }
     return msg.wParam;
     UNREFERENCED_PARAMETER(lpCmdLine);
}
/* ************************************
* BOOL InitApplication(HINSTANCE hinstance)
* 功能：注册主窗口类
* 参数：hinstance，应用程序本次运行实例
* 返回值：是否成功
************************************/
BOOL InitApplication(HINSTANCE hinstance)
{
     // 使用 RegisterClassEx
     WNDCLASSEX wcx;
     // 填充结构
     wcx.cbSize = sizeof(wcx); // WNDCLASSEX 结构的大小
     wcx.style = CS_HREDRAW | CS_VREDRAW; // 如果大小改变了重绘窗口
     wcx.lpfnWndProc = MainWndProc; // 窗口消息处理函数
     wcx.cbClsExtra = 0; // 无附加窗口类内存
     wcx.cbWndExtra = 0; // 无附加窗口内存
     wcx.hInstance = hinstance; // 应用程序实例
     wcx.hIcon = NULL; // 图标
     wcx.hCursor = LoadCursor(NULL, IDC_ARROW); // 鼠标指针
     wcx.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH); // 背景画刷
     wcx.lpszMenuName = NULL;//MAKEINTRESOURCE(IDR_MENU_MAIN); // 菜单资源
     wcx.lpszClassName = "MainWClass"; // 窗口类名
```

```
    wcx.hIconSm = NULL;
    // 注册窗口类，返回
    return RegisterClassEx(&wcx);
}

/* ************************************
* BOOL InitInstance(HINSTANCE hinstance, int nCmdShow)
* 功能：创建主窗口和控件
* 参数：hinstance，应用程序本次运行实例
* nCmdShow，如何显示
* 返回值：是否成功
**************************************/
BOOL InitInstance(HINSTANCE hinstance, int nCmdShow)
{
    // 保存应用程序实例句柄
    hinst = hinstance;
    hwnd = CreateWindow(
        "MainWClass",          // 窗口类名，使用之前注册的主窗口类
        "ClipBoardViewer",     // 窗口名，显示在窗口标题栏上的字符串
        WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_CLIPSIBLINGS |
        WS_POPUPWINDOW,        // 窗口样式
        CW_USEDEFAULT,         // 水平位置（默认）
        CW_USEDEFAULT,         // 垂直位置（默认）
        800, // 宽
        600, // 高
        (HWND) NULL,      // 无父窗口
        (HMENU) LoadMenu(hinst,MAKEINTRESOURCE(IDR_MENU_COMMAND)), // 菜单
        hinstance,        //应用程序实例
        (LPVOID) NULL);// 无窗口创建数据
    // 窗口是否创建成功
    if (!hwnd)
        return FALSE;
    // 显示、刷新窗口
    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);
    return TRUE;
}
/* ************************************
* MainWndProc
* 窗口消息处理函数
**************************************/
LRESULT CALLBACK MainWndProc(
                                HWND hwnd,
                                UINT uMsg,
                                WPARAM wParam,
                                LPARAM lParam)
{
    static HWND hwndNextViewer;
    HDC hdc;
    HDC hdcMem;
    PAINTSTRUCT ps;
    LPPAINTSTRUCT lpps;
    RECT rc;
    LPRECT lprc;
    HGLOBAL hglb;
    LPSTR lpstr;
    HBITMAP hbm;
    HENHMETAFILE hemf;
    HWND hwndOwner;
    switch (uMsg)
    {
```

```
case WM_PAINT: // 收到 WM_PAINT 消息后显示剪贴板中的数据
     hdc = BeginPaint(hwnd, &ps);
     // uForamt 全局变量，在 WM_DRAWCLIPBOARD 和 WM_COMMAND
     // 消息处理中，使用 SetAutoView 设置
     // 根据剪贴板中数据的不同格式，使用不同的显示方式
     switch (uFormat)
     {
          // Owner-display 格式，剪贴板的所有者必须显示并刷新 Viewer 的窗口
     case CF_OWNERDISPLAY:
          // 获取剪贴板的所有者
          hwndOwner = GetClipboardOwner();
          // 获取剪贴板数据
          hglb = GlobalAlloc(GMEM_MOVEABLE, sizeof(PAINTSTRUCT));
          lpps = GlobalLock(hglb);
          memcpy(lpps, &ps, sizeof(PAINTSTRUCT));
          GlobalUnlock(hglb);
          // 向剪贴板所有都发送 WM_PAINTCLIPBOARD 消息
          SendMessage(hwndOwner, WM_PAINTCLIPBOARD,
               (WPARAM) hwnd, (LPARAM) hglb);
          GlobalFree(hglb);
          break;
          // 位图
     case CF_BITMAP:
          // 创建 DC
          hdcMem = CreateCompatibleDC(hdc);
          if (hdcMem != NULL)
          {
               if (OpenClipboard(hwnd)) // 打开剪贴板
               {
                    // 获取剪贴板数据
                    hbm = (HBITMAP) GetClipboardData(uFormat);
                    // 将位图选择入 DC，显示在窗口客户区
                    SelectObject(hdcMem, hbm);
                    GetClientRect(hwnd, &rc);
                    BitBlt(hdc, 0, 0, rc.right, rc.bottom,
                         hdcMem, 0, 0, SRCCOPY);
                    // 关闭剪贴板
                    CloseClipboard();
               }
               // 释放 DC
               DeleteDC(hdcMem);
          }
          break;
          // 文本
     case CF_TEXT:
          if (OpenClipboard(hwnd)) // 打开剪贴板
          {
               // 获得剪贴板数据
               hglb = GetClipboardData(uFormat);
               lpstr = GlobalLock(hglb);
               // 将文件绘制在窗口客户区
               GetClientRect(hwnd, &rc);
               DrawText(hdc, lpstr, -1, &rc, DT_LEFT);
               GlobalUnlock(hglb);
               // 关闭剪贴板
               CloseClipboard();
          }
          break;
          // enhanced-format metafile，加强格式图元文件
     case CF_ENHMETAFILE:
          // 打开剪贴板
```

```
            if (OpenClipboard(hwnd))
            {
                // 获取数据
                hemf = GetClipboardData(uFormat);
                // 调用 PlayEnhMetaFile 在窗口客户区上显示
                GetClientRect(hwnd, &rc);
                PlayEnhMetaFile(hdc, hemf, &rc);
                CloseClipboard();
            }
            break;
        case 0: // 剪贴板为空
            GetClientRect(hwnd, &rc);
            // 在窗户区中央显示
            DrawText(hdc, "The clipboard is empty.", -1,
                &rc, DT_CENTER | DT_SINGLELINE |
                DT_VCENTER);
            break;
        default: // 不支持其他格式 ,
                GetClientRect(hwnd, &rc);
            DrawText(hdc, "Unable to display format.", -1,
                &rc, DT_CENTER | DT_SINGLELINE |
                DT_VCENTER);
        }
        EndPaint(hwnd, &ps);
        break;
    case WM_SIZE: // 如果窗口大小改变，通知剪贴板所有者窗口
        if (uFormat == CF_OWNERDISPLAY)
        {
            hwndOwner = GetClipboardOwner();
            hglb = GlobalAlloc(GMEM_MOVEABLE, sizeof(RECT));
            lprc = GlobalLock(hglb);
            GetClientRect(hwnd, lprc);
            GlobalUnlock(hglb);
            SendMessage(hwndOwner, WM_SIZECLIPBOARD,
                (WPARAM) hwnd, (LPARAM) hglb);
            GlobalFree(hglb);
        }
        break;
    case WM_CREATE:
        // 在窗口创建时，在剪贴板 Viewer 链中增加一个 Viewer
        hwndNextViewer = SetClipboardViewer(hwnd);
        break;
    case WM_CHANGECBCHAIN:
        // 在链中增加本 Viewer 后，位于链的第一个节点
        // 当链中有 Viewer 移除时，链中的第一个 Viewer 会收到此消息
        // 如果是本 Viewer 的下一个 Viewer 移除，那么更新
        if ((HWND) wParam == hwndNextViewer)
            hwndNextViewer = (HWND) lParam;
        // 否则将消息向下一个 Viewer 传递
        else if (hwndNextViewer != NULL)
            SendMessage(hwndNextViewer, uMsg, wParam, lParam);
        break;
    case WM_DESTROY:// 窗口关闭
        // 设置 Viewer 链，移除本 Viewer。
        ChangeClipboardChain(hwnd, hwndNextViewer);
        // 退出
        PostQuitMessage(0);
        break;
    case WM_DRAWCLIPBOARD: // 剪贴板的内容改变时会收到此消息
        // 设置显示格式为 Auto
        SetAutoView(hwnd);
```

```
            // 将消息发送给 Viewer 链中的下一个窗口
            SendMessage(hwndNextViewer, uMsg, wParam, lParam);
            break;
        case WM_INITMENUPOPUP:// 当 POPUP 菜单弹出时收到此消息
            // 根据剪贴板中内容的格式设置菜单
            if (!HIWORD(lParam))
                InitMenu(hwnd, (HMENU) wParam);
            break;
        case WM_COMMAND: // 处理用户菜单输入
            switch (LOWORD(wParam))
            {
            case ID_NULL:
                break;
            case ID_AUTO: // 点击 Auto 菜单项
                SetAutoView(hwnd); // 设置显示格式为自动
                break;
            default:
                // 用户选择了特定的显示格式
                // 设置显示模式，uFormat 全局变量
                // uFormat 在 WM_PAINT 消息时引用
                fAuto = FALSE;
                uFormat = LOWORD(wParam);
                InvalidateRect(hwnd, NULL, TRUE);
            }
            break;
        default: // 其他消息
            return DefWindowProc(hwnd, uMsg, wParam, lParam);
        }
        return (LRESULT) NULL;
}

/* ************************************
* void WINAPI SetAutoView(HWND hwnd)
* 获取剪贴板的主要格式，并设置显示方式
************************************/
void WINAPI SetAutoView(HWND hwnd)
{
    static UINT auPriorityList[] = {
        CF_OWNERDISPLAY,
        CF_TEXT,
        CF_ENHMETAFILE,
        CF_BITMAP
    };
    // 获取剪贴板主要格式
    // 设置显示模式
    // uFormat 在 WM_PAINT 消息时引用
    uFormat = GetPriorityClipboardFormat(auPriorityList, 4);
    fAuto = TRUE;
    InvalidateRect(hwnd, NULL, TRUE);
    UpdateWindow(hwnd);
}

/* ************************************
* 功能：根据剪贴板中内容的格式，设置菜单项供用户选择显示方式
* 参数 hwnd，窗口句柄
* hmenu，需要设置的菜单句柄
************************************/
void WINAPI InitMenu(HWND hwnd, HMENU hmenu)
{
    UINT uFormat;
    char szFormatName[80];
```

```
    LPCSTR lpFormatName;
    UINT fuFlags;
    UINT idMenuItem;
    // 判断菜单的第一项是不是 Auto
    // 所有的显示格式附加到这个 POPUP 中
    if (GetMenuItemID(hmenu, 0) != ID_AUTO)
        return;
    // 将除了第一个以外的其他所有菜单项删除
    while (GetMenuItemCount(hmenu) > 1)
        DeleteMenu(hmenu, 1, MF_BYPOSITION);
    // Auto 项是否设置
    fuFlags = fAuto ? MF_BYCOMMAND | MF_CHECKED :
        MF_BYCOMMAND | MF_UNCHECKED;
    CheckMenuItem(hmenu, ID_AUTO, fuFlags);
    // 检测剪贴板中格式的数量
    if (CountClipboardFormats() == 0)
        return;
    // 打开剪贴板
    if (!OpenClipboard(hwnd))
        return;
    // 为每个格式附加一个菜单项
    AppendMenu(hmenu, MF_SEPARATOR, 0, NULL);
    uFormat = EnumClipboardFormats(0);
    while (uFormat)
    {
        // 程序为每一个格式定义了一个在菜单项中显示的名字
        lpFormatName = GetPredefinedClipboardFormatName(uFormat);
        // 如果程序未定义，获取格式的名字
        if (lpFormatName == NULL)
        {
            // 注意溢出
            if (GetClipboardFormatName(uFormat, szFormatName,
                sizeof(szFormatName)))
                lpFormatName = szFormatName;
            else
                lpFormatName = "(unknown)";
        }
        // 是可显示的格式
        if (IsDisplayableFormat(uFormat))
        {
            fuFlags = MF_STRING;
            idMenuItem = uFormat;
        }
        else
        {
            fuFlags = MF_STRING | MF_GRAYED;
            idMenuItem = 0;
        }
        // 增加菜单项
        AppendMenu(hmenu, fuFlags, idMenuItem, lpFormatName);
        // 下一个格式，循环
        uFormat = EnumClipboardFormats(uFormat);
    }
    CloseClipboard();
}
/* ************************************
* 功能：根据格式返回格式名字符串，用于在菜单上显示
* 参数 uFormat，格式
* 返回值：LPSTR，格式名字符串，注意不能返回局部变量
**************************************/
LPSTR GetPredefinedClipboardFormatName(UINT uFormat)
{
    switch (uFormat)
```

```
        {
        case CF_OWNERDISPLAY:
             return szFormatOwnerdiplay;
        case CF_TEXT:
             return szFormatText;
        case CF_ENHMETAFILE:
             return szFormatEnhmetafile;
        case CF_BITMAP:
             return szFormatBitMap;
        default:
             return NULL;
        }
}
/* ************************************
* BOOL WINAPI IsDisplayableFormat(UINT uFormat)
* 功能：判断是否为可显示的格式
* 参数 uFormat，格式
* 返回值：是、否
************************************/
BOOL WINAPI IsDisplayableFormat(UINT uFormat)
{
     switch (uFormat)
     {
     case CF_OWNERDISPLAY:
     case CF_TEXT:
     case CF_ENHMETAFILE:
     case CF_BITMAP:
          return TRUE;
     }
     return FALSE;
}
```

3．运行结果

如图 11-6 所示，运行 ClipView. exe，如果截屏，那么会显示所截获的位图。

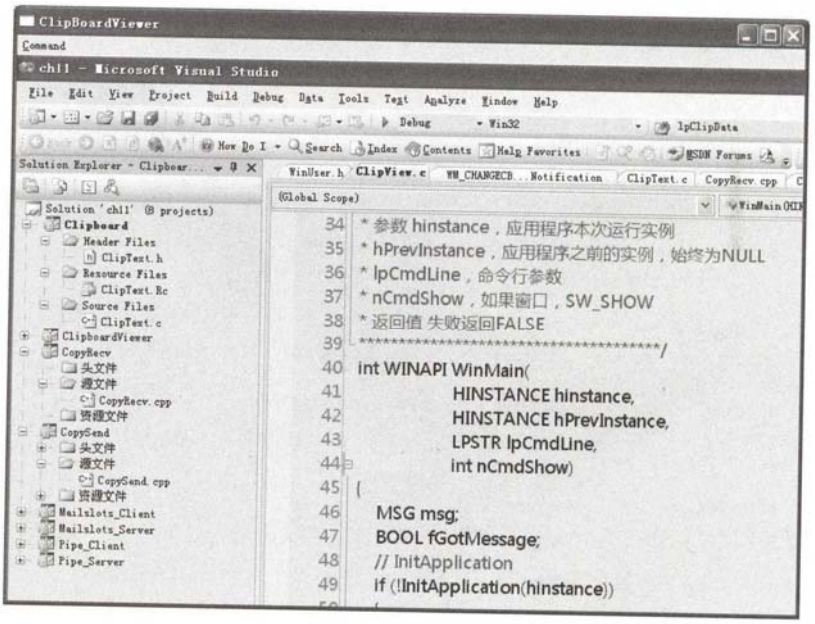

图 11-6　截屏结果

如果在 Word 中复制数据，那么可以显示所复制的文本，文件具有格式，结果如图 11-7 所示。

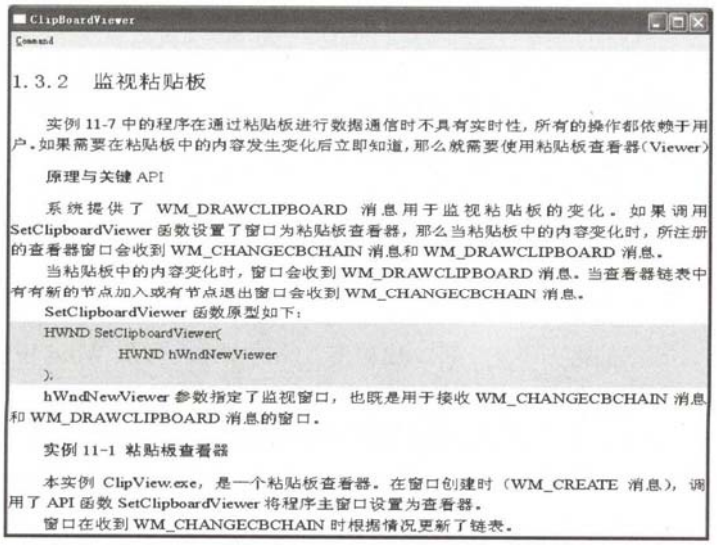

图 11-7　显示结果

11.3.3　剪贴板数据格式

剪贴板中可能会存在各种各样的数据。因此剪贴板中在保存数据的同时还需要保存数据的格式信息。

系统使用一个 UINT 类型的数据来表示剪贴板中的数据格式。

在这些格式信息中，有很多是各种应用程序之间通用的，比如文本、位图等。这些数据格式已经由系统预先定义，称为标准格式。

当然有一些应用程序也希望自行定义剪贴板的数据格式，这样可以方便地在同一个应用程序的不同实例间进行数据传递而不需要对数据的格式进行过多的处理(典型的就包括 Word)。

1．标准格式

标准格式有很多种，表 11.2 列举出最为常用的几种。

表 11.2　　剪贴板格式

格　式　值	意　　义
CF_BITMAP	位图句柄（HBITMAP）
CF_DIB	内存位置包括 BITMAPINFO 结构和位图数据
CF_ENHMETAFILE	增加图元文件句柄（HENHMETAFILE）
CF_OEMTEXT	OEM 字符集的字串（以 CR-LF 方式换行）
CF_OWNERDISPLAY	由剪贴板查看器查看的格式
CF_PALETTE	调色板数据
CF_RIFF	标准的 CF_WAVE 波形（wave）数据
CF_TEXT	ANSI 字符串（以 CR-LF 方式换行）
CF_WAVE	PCM 波形
CF_TIFF	Tagged 图像文件格式
CF_UNICODETEXT	Unicode 字符串

2．自定义格式

如果需要自定义格式，那么调用 RegisterClipboardFormat API 函数，函数原型如下：

```
UINT RegisterClipboardFormat(
    LPCTSTR lpszFormat
);
```

在调用时，使用 lpszFormat 参数指定一个格式名，函数会返回系统为其分配的格式类型值(UINT)。

3．多种格式

在很多情况下，数据的格式不止一种，比如格式化的文本（如从 Word 中复制的数据、从网页中复制的数据等）有效的格式不止一种，就可能会具有多重格式。

以下几个 API 函数都是用于获取当前剪贴板中格式信息。

(1) GetPriorityClipboardFormat.

GetPriorityClipboardFormat 函数的功能是检测剪贴板中是否有特定格式的数据，函数原型如下：

```
int GetPriorityClipboardFormat(
   UINT *paFormatPriorityList,
   int cFormats
);
```

检测是否有 paFormatPriorityList 参数所指定的格式数组中的格式，如果有则返回数组中的第一个剪贴板当前具有的格式。cFormats 是 paFormatPriorityList 数组的大小。

(2) CountClipboardFormatS。

CountClipboardFormats 函数返回当前剪贴板中的具有的不同格式的数量，函数原型如下：

```
int CountClipboardFormats( VOID );
```

(3) EnumClipboardFormatS。

EnumClipboardFormats 函数可以列举当前剪贴板中的所有格式，函数原型如下：

```
UINT EnumClipboardFormats(
   UINT format
);
```

通过 format 参数指定一个已知的格式，通过返回值返回下一个格式。

(4) GetUpdatedClipboardFormatSa

GetUpdatedClipboardFormats 函数获取当前剪贴板的所有格式，函数原型如下：

```
BOOL GetUpdatedClipboardFormats(
   PUINT lpuiFormats,
   UINT cFormats,
   PUINT pcFormatsOut
);
```

lpuiFormats 参数指向用于保存返回的格式数组的缓存。cFormats 表示 lpuiFormats 可以容纳的格式信息的数量，即 lpuiFormats 数组的大小。pcFormatsOut 返回真实的数组大小。

> **注意** 因为剪贴板中的数据会有多种格式，在调用 GetClipboardData 函数获取数据时，需要指定格式。一般情况下，指定不同格式，会获取不同的内容。

4．剪贴板数据的格式信息

每一个剪贴板格式都有一个格式名，格式名是一个字符串，使用 GetClipboardFormatName API 函数可以获得。

GetClipboardFormatName 函数原型如下：

```
int GetClipboardFormatName(
   UINT format,
   LPTSTR lpszFormatName,
   int cchMaxCount
);
```

11.4 数据复制消息(WM_COPYDATA)

WM_COPYDATA 是一个非常特殊的消息，此消息可能携带一个比较大的消息参数，而其他消息都只能携带两个固定大小的参数(WPARAM 和 LPARAM)。

在发送 WM_COPYDATA 消息时，WM_COPYDATA 的 wParam 参数应该赋值为发送此消息的窗口，而 lParam 消息参数指向一个 COPYDATASTRUCT 结构类型的变量。

COPYDATASTRUCT 结构如下：

```
typedef struct tagCOPYDATASTRUCT {
   ULONG_PTR dwData;
   DWORD cbData;
   PVOID lpData;
} COPYDATASTRUCT, *PCOPYDATASTRUCT;
```

dwData 可以是任意值，lpData 是需要发送给目录窗口所在进程的数据，cbData 字符指定了 lpData 内存区域的字节数，也就是需要发送给目标进程的 lpData 所指向的数据的大小。

在消息发送时，系统会将整个 COPYDATASTRUCT 及 lpData 所指向的内容全部发送给目标进程。目标进程窗口在收到 WM—COPYDATA 后，可以从 lParam 参数中提取出数据。

11.4.1 数据发送端

下面通过实例说明 WM_COPYDATA 消息的使用，实例 11-9 是使用 WM_COPYDATA 消息的消息发送端。

实例 11-9 发送 WM_COPYDATA 消息

实例程序创建了一个文本框控件，用户可以在控件中输入文字，单击“Send”按钮后，向窗口名为“RECV COPY DATA”的窗口发送 WM_COPYDATA 消息，将用户在文本框中输入的内容全部发送给“RECV COPY DATA”窗口。

```
/* ************************************
* CopySend.c 通过 WM_COPYDATA 进程间通信
************************************/
/* 头文件 */
#include <windows.h>
#include <commctrl.h>
#include "resource.h"
/* 全局变量 */
HINSTANCE hinst;
HWND hwnd, hwndEdit;
LPSTR lpszCopy =  "you can input any text\n then click \'Send\' Item at Menu";
/* 函数声明 */
int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int);
LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

/* ************************************
```

```
* WinMain
*************************************/
int WINAPI WinMain(
                        HINSTANCE hinstance,
                        HINSTANCE hPrevInstance,
                        LPSTR lpCmdLine,
                        int nCmdShow)
{
     MSG msg;
     WNDCLASSEX wcx;
     hinst = hinstance;      // 保存应用程序实例
     // 填充 WNDCLASSEX
     wcx.cbSize = sizeof(wcx);
     wcx.style = CS_HREDRAW | CS_VREDRAW;
     wcx.lpfnWndProc = MainWndProc;          // 消息处理函数
     wcx.cbClsExtra = 0;
     wcx.cbWndExtra = 0;
     wcx.hInstance = hinstance;
     wcx.hIcon = LoadIcon(NULL,      IDI_APPLICATION);
     wcx.hCursor = LoadCursor(NULL, IDC_ARROW);
     wcx.hbrBackground = (HBRUSH)GetStockObject( WHITE_BRUSH);
     wcx.lpszMenuName =  MAKEINTRESOURCE(IDR_MENU_COMMAND);   // 菜单
     wcx.lpszClassName = "copydata";          // 窗口类名
     wcx.hIconSm = NULL;
     // 注册窗口类
     if(RegisterClassEx(&wcx) == 0)
     {
          return 0;
     }
     // 创建窗口
     hwnd = CreateWindow(
          "copydata",
          "Sample",
          WS_TILEDWINDOW,
          CW_USEDEFAULT, CW_USEDEFAULT, 500,      400,
          (HWND) NULL,
          (HMENU) NULL,
          hinstance,
          (LPVOID) NULL);
     if (!hwnd)
          return 0;
     // 显示、刷新窗口
     ShowWindow(hwnd, nCmdShow);
     UpdateWindow(hwnd);
     // 消息循环
     BOOL fGotMessage;
     while ((fGotMessage = GetMessage(&msg, (HWND) NULL, 0, 0)) != 0 && fGotMessage != -1)
     {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }
     return msg.wParam;
     UNREFERENCED_PARAMETER(lpCmdLine);
}

/* ************************************
* MainWndProc 消息处理函数
*************************************/
LONG APIENTRY MainWndProc(HWND hwnd,
                              UINT message,
                              WPARAM wParam,
```

```
                                LPARAM lParam)
{
    switch (message)
    {
    case WM_CREATE:
        {
            // 在主窗口创建时，创建一个 EDIT 控件，用于编辑需要复制的内容
            RECT rectMain;
            GetClientRect(hwnd,&rectMain);
            hwndEdit = CreateWindow("EDIT",
                NULL,
                WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_BORDER |
                ES_LEFT | ES_MULTILINE | ES_AUTOVSCROLL,
                0., 0, rectMain.right, rectMain.bottom,
                hwnd,       NULL,
                (HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE),
                NULL);
            if(hwndEdit == NULL)
            {
                MessageBox(hwnd, "Create Window Error","ERROR",MB_OK);
                ExitProcess(0);
            }
            // 设置 EDIT 控件的内容
            SendMessage(hwndEdit, WM_SETTEXT, 0, (LPARAM) lpszCopy);
            break;
        }
    case WM_COMMAND:        // 菜单输入
        {
            // 如果通过菜单单击 "send" 按钮
            if (LOWORD(wParam) == ID_COMMAND_SEND)
            {
                // 获取 EDIT 控件中的文本
                DWORD dwTextLenth = GetWindowTextLength(hwndEdit);
                LPSTR szText = (LPSTR)HeapAlloc(
                    GetProcessHeap(),HEAP_NO_SERIALIZE,dwTextLenth);
                GetWindowText(hwndEdit,szText,dwTextLenth);
                // 构造 COPYDATASTRUCT
                COPYDATASTRUCT cds;
                HWND hwnd2 = FindWindow("copydata2","RECV COPY DATA");
                if(hwnd2 == NULL)
                {
                    MessageBox(hwnd,"can not find window 2","ERROR",MB_OK);
                    break;
                }
                cds.dwData = 0x12345678;            // 自定义的数据
                cds.cbData = dwTextLenth;           // lpData 的长度
                cds.lpData = szText;                // 需要复制的数据
                // 发送消息，COPYDATASTRUCT 结构的指针作为 lParam
                SendMessage(
                    hwnd2,                          // 目的窗口
                    (UINT) WM_COPYDATA,             // 消息 ID
                    (WPARAM) hwnd,                  //  wParam，源窗口句柄
                    (LPARAM) &cds                   //  lParam，COPYDATASTRUCT
                    );
                                                    // 释放内存
                HeapFree(GetProcessHeap(),HEAP_NO_SERIALIZE,szText);
            }
            break;
        }
    case WM_DESTROY:  // 退出
        PostQuitMessage(0);
        return 0;
```

```
    default:      // 其他消息的处理
        return DefWindowProc(hwnd, message, wParam, lParam);
    }
return 0;
}
```

运行实例，用户可以输入任何内容，单击菜单中的 Send，如果存在名为“RECV COPY DATA”标题的窗口，则向其发送数据。

图 11-8　发送数据

11.4.2　数据接收端

实例 11-10 是VIrM_COPYDATA 消息的接收端。

实例 11-10 接收 WM_COPYDATA 消息

实例创建了名为“RECV COPY DATA”的窗口，并创建了一个文本框控件，在主窗口收到 WM COPYDATA 消息后，将消息的 lParam 消息提取出来，显示在文本框中。

```
/* ************************************
* CopyRecv.c 通过 WM_COPYDATA 进行进程间通信
************************************/
/* 头文件 */
#include <windows.h>
#include <commctrl.h>
/* 全局变量 */
HINSTANCE hinst;
HWND hwnd,hwndEdit;
/* 函数声明 */
int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int);
LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);
/* ************************************
* void main()
* 功能 进程间 mailslot 通信客户端
************************************/
int WINAPI WinMain(
                    HINSTANCE hinstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow)
{
```

```
    MSG msg;
    WNDCLASSEX wcx;
    // 填充 WNDCLASSEX
    wcx.cbSize = sizeof(wcx);
    wcx.style = CS_HREDRAW | CS_VREDRAW;
    wcx.lpfnWndProc = MainWndProc;
    wcx.cbClsExtra = 0;
    wcx.cbWndExtra = 0;
    wcx.hInstance = hinstance;
    wcx.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wcx.hCursor = LoadCursor(NULL, IDC_ARROW);
    wcx.hbrBackground = (HBRUSH)GetStockObject( WHITE_BRUSH);
    wcx.lpszMenuName = NULL;
    wcx.lpszClassName = "copydata2";
    wcx.hIconSm = NULL;
    // 注册窗口类
    if(RegisterClassEx(&wcx) == 0)
    {
        return 0;
    }
    // 创建窗口
    hwnd = CreateWindow(
        "copydata2",
        "RECV COPY DATA",
        WS_TILEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, 500, 400,
        (HWND) NULL, (HMENU) NULL, hinstance, (LPVOID) NULL
        );
    if (!hwnd)
        return 0;
    // 显示刷新窗口
    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);
    // 消息
    BOOL fGotMessage;
    while ((fGotMessage = GetMessage(&msg, (HWND) NULL, 0, 0)) != 0 && fGotMessage != -1)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
    UNREFERENCED_PARAMETER(lpCmdLine);
    return 0;
}
// 消息处理函数
LONG APIENTRY MainWndProc(HWND hwnd,
                          UINT message,
                          WPARAM wParam,
                          LPARAM lParam)
{
    switch (message)
    {
    case WM_CREATE:
        {
            // 创建主窗口时，创建一个 EDIT 控件
            RECT rectMain;
            GetClientRect(hwnd,&rectMain);
            hwndEdit = CreateWindow("EDIT",
                NULL,
                WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_BORDER |
                ES_LEFT | ES_MULTILINE | ES_AUTOVSCROLL,
                0., 0, rectMain.right, rectMain.bottom,
```

```
                    hwnd,
                    NULL,
                    (HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE),
                    NULL);
               if(hwndEdit == NULL)
               {
                    MessageBox(hwnd, "Create Window Error","ERROR",MB_OK);
                    ExitProcess(0);
               }
               break;
          case WM_COPYDATA:
               {
                    // 当收到 WM_COPYDATA 消息时，从 lParam 中提取出
                    // COPYDATASTRUCT
                    PCOPYDATASTRUCT lpcds = (PCOPYDATASTRUCT)lParam;
                    // 将复制的数据在界面上显示
                    SetWindowText(hwndEdit,(LPSTR)lpcds->lpData);
                    break;
               }
          case WM_DESTROY: // 退出
               PostQuitMessage(0);
               return 0;
          default: // 其他消息处理
               break;
               }
          }
          return DefWindowProc(hwnd, message, wParam, lParam);
     }
```

程序接收到消息后，将消息提取并显示内容。

图 11-9　接收数据

运行结果表示，接收端收到了发送端发送的消息，并收到数据。

11.5　其他进程间通信方式

本节简单介绍几种其他的进程间通信方式，包括动态数据交换、File Mapping 和 Socket 等。

11.5.1　动态数据交换(DDE)和网络动态数据交换(NDDE)

DDE 可能是最为强大的进程间通信方式，实现也非常复杂。通信的两端分为服务端与客户端，两端都需要具有窗口。

一个 DDE 的通信过程大致如下。

DDE 会话由客户端发起，先初始化一个会话。DDE 客户端会向服务端发送一个 WM_ DDE INITIATE 窗口消息。服务窗口收到此消息后，根据条件判断是否需要响应此消息，如果服务端确定可以与客户端建立会话，那么会返回 WM__ DDE_ ACK 消息。如果客户端成功收到了服务端发送的 WM—DDE- ACK 窗口消息，那么会话就已经建立，之后就进行数据的传输。

数据传输分为很多种方式，可以只传输一个单独的数据项，也可以再建立一个永久数据连接(Permanent Data Link)。

如果进行独立数据项传输，数据传输的单位是数据项。客户端使用 WM_DDE_REQUEST 和 WM_ DDE_ POKE 消息向服务端接收和发送数据项。数据的传输使用 WM_ DDE_ DATA 消息。

而永久数据连接的建立也是通过客户端发起,客户端向服务端发送 WM DDE_ADVISE 消息，服务端根据情况做出回应，根据是否能够建立连接而返回不同的 WM_ DDE_ ACK 消息。终止数据连接使用消息 WM_DDE_UNADVISE。

客户端还可以使用 WM_DDE_EXECUTE 请服务端发送命令。

终止一个 DDE 会话使用 WM DDE_TERMINATE 消息。

几乎所有的过程都是由客户端发起请求，而服务端响应，服务端响应的方向大多是发送一个 WM__ DDE_ACK 消息。

DDE 的通信需要操作系统 ATOM 表的帮助，用于索引数据。操作 ATOM 表有专门的 API。

11.5.2 通过 File Mapping 在进程间共享数据

共享文件是一种高效的进程间通信方式，因为直接共享内存，所以访问速度非常快。而且允许大量数据传输。

共享内存的进程间通信与其他进程间通信方式相比，限制条件要少。比如邮槽对数据大小和数据流向有限制，管道只能采用半双工通信形式等，剪贴板容易受其他进程影响等。

本书 4.4.2 小节已经详细介绍了通过 File Mapping 进行进程间数据共享的完整的实现原理和实例代码。

使用 File Mapping 共享数据也有缺点。最大的缺点是无法实现实时性，一个进程无论是写入还是读出数据后，另一个进程都不知道对方的操作是否已经完成。

因此在要求实时数据传输时，File Mapping 通常与其他进程通信方式配合使用，用于传输大量的数据。

11.5.3 Windows Socket

利用 Windows Socket 实现的进程间通信完全借用了 Socket 通信的原理。其实任何可以在两台主机间进行网络数据通信的机制都可以用于进程间通信。

如果在一台主机上既使用 Socket 服务端又实现 Socket 客户端，那么客户端在连接服务端时将 IP 指定为本机，就可以实现同一主机上的 Socket 服务端进程和 Socket 客户端进程间的通信了。

使用 Socket 进行进程间通信既可以实现实时性，又可以传输大数据量信息，且数据传输的速度也很快，因此应用范围很广泛。

在进程间实现 Socket 通信的原理和程序实现方法与实现不同主机间的 Socket 通信没有区别，请参考本书第 14 章相关内容。

第 12 章　Windows Shell程序设计

几乎所有的操作系统都有“Shell”的概念。所谓 Shell（壳），一般是指由操作系统提供的，用于计算机用户向操作系统输入相关指令并得到结果的程序。Shell 可以字符形式的，也可以是图形界面形式的。

本章所介绍是图形界面形式的 Shell。Windows Shell 最重要的组成部件是 explorer.exe。在使用 Windows 操作系统时，开始菜单、任务栏、资源管理器等都是 explorer.exe 提供的。

因此 Shell 程序设计也是图形用户界面设计的重要组成部分。掌握 Shell 程序设计，需首先了解下面 4 个方面的内容。

(1) Windows Shell 编程接口。

Windows Shell 提供一系列编程接口，例如可以获取特殊文件夹，使用文件浏览对话框等。

Shell 编程接口的函数名通常是以“SH”开始。

(2) Windows Shell 扩展。

Windows 平台的 Shell 是可扩展的。可以为 Windows Shell 开发扩展程序(Extention)，例如：

◇定制不同类型的文件的右键菜单、拖拽菜单（比如 WinRAR、Adobe Acrobat 等应用程序具有这样的功能）；

◇开发系统通知区域图标、并为图标定制菜单和气泡弹出功能(QQ、MSN、Visual Studio 等应用程序具有这样的功能)；

◇定制任务栏、定制工具栏等；

◇增加自动播放对话框中的选项；

◇添加控制面板项；

◇为不同文件类型文件属性页、文件夹的属性页（右键属性）增加选项页。

上面是几种典型的 Shell 扩展应用。但是 Shell 扩展应用远不止这几种情况，本章将介绍其中几种的实现方法。

(3)注册表在 Shell 应用中具有重要作用。

Shell 应用在很大程度上依赖于注册表，举例如下。

◇文件类型辅助：特定扩展名的文件的图标、默认打开程序等是通过注册表来配置的。

◇　Shell 扩展的安装：比如特定类型文件的右键菜单、拖拽菜单(Context Meun)的菜单项及输入处理函数所在的动态链接库。

(4)相关头文件和库文件。

进行 Shell 程序的设计，需要使用一些头文件和库文件。

一般 Shell API 都在 shlobj.h 头文件中声明，由 Shell32.dll 导出，链接时需要使用到 Shell32.lib 库。

12.1　Windows Shell目录管理

计算机用户使用 Shell 最常进行的操作就是文件操作。Shell 最重要的功能之一是进行文件浏览、查找、管理以及将文件和应用程序关联。

Windows Shell 中有很多特殊目录和文件，比如所有 Windows 用户都经常接触到的“我

的文档”、“桌面”、“回收站”、“程序文件”(Program files)等。这些目录都是 Shell 特殊目录。可以通过 Shell 程序设计接口开发程序，获取和操作这些特殊目录。

12.1.1 Shell对目录和文件的管理形式

Shell 有一种特殊的文件和目录管理方式，每个目录都有一个 PIDL (Pointer ofItem identifier list，项标识符表指针)值，这个值惟一标识一个文件夹。

由系统定义的特殊文件夹的 CSIDL (constant specialitemID list)是常数，比如：

CSIDL_ DESKTOP 代表“桌面”文件夹；

CSIDL_FAVORITES 代表“收藏夹”；

CSIDL_FONTS 代表字体文件夹；

CSIDL_MYDOCUMENTS，代表“我的文档”；

CSIDL_MYMUSIC 代表“我的音乐”；

CSIDL PROFILE 代表“用户”文件夹，一般情况下是 C:\Documents and Settings\username;

CSIDL_PROGRAMS 代表“程序”文件夹，一般情况下是 C:\Program Files; .

CSIDL RECENT，代表“最近的文档”；

CSIDL_STARTMENU，代表“开始菜单”目录；

CSIDL_SYSTEM、CSIDL_WINDOWS 分别代表“系统”和“Windows”目录。

SHGetNameFromIDList、SHGetPathFromIDList 等 Shell API 函数通过 CSIDL 获得有关于目录的详细信息。

12.1.2 “我的文档”等特殊目录相关操作

本节将通过示例说明系统特殊目录的操作。

1. 关键 API 及数据

(1) SHGetSpecialFolderPath 函数。

通过文件夹的 CSIDL，获得文件夹的路径，函数原型如下：

```
BOOL SHGetSpecialFolderPath(
    HWND hwndOwner,
   LPTSTR lpszPath,
   int nFolder,
   BOOL fCreate
);
```

(2) SHGetFolderLocation 函数。

获取文件夹的路径，并保存在 ITEMIDLIST 结构中，函数原型如下：

```
HRESULT SHGetFolderLocation(
   HWND hwndOwner,
   int nFolder,
   HANDLE hToken,
   DWORD dwReserved,
   PIDLIST_ABSOLUTE *ppidl
);
```

(3) SHGetPathFromIDList 函数。

从 PIDL 转换为路径，函数原型如下：

```
BOOL SHGetPathFromIDList(
    PCIDLIST_ABSOLUTE pidl,
    LPTSTR pszPath
);
```

2．实例 12-1　获取并显示“我的文档”等特殊目录

本实例使用了若干表示系统特殊目录的 CSIDL 常数，通过这些常数获取文件夹的真实路径，并显示。

本实例使用了两种方法来获取特殊文件夹的路径，一种是直接使用 SHGetSpecialFolderPath API,
另外一种更为通用，使用 SHGetFolderLocation 从 PIDL 或 CSIDL 获得文件夹位置后，再使用 SHGetPathFromIDList 获取文件路径字符串。

3．运行结果

本实例的运行结果参见 12.1.3 小节的图 12-1。

```
/*************************************
* VOID GetSpecialFolder()
* 功能     获取并显示特殊目录
*************************************/
VOID GetSpecialFolder()
{
     // 获取“我的文档”的路径
     CHAR szMyDocument[MAX_PATH];// My Document 的路径
     // 使用 SHGetSpecialFolderPath 获取特殊目录路径
     SHGetSpecialFolderPath(NULL,szMyDocument,CSIDL_PERSONAL,FALSE);
     // 获取桌面的路径
     CHAR szDesktop[MAX_PATH];   //DeskTop 的路径
     LPITEMIDLIST pidl = NULL;
     LPMALLOC pMalloc = NULL;
     // 分配
     SHGetMalloc(&pMalloc);
     // 使用 SHGetFolderLocation、SHGetPathFromIDList 可以获取任意目录的路径
     SHGetFolderLocation(NULL, CSIDL_DESKTOPDIRECTORY, NULL, 0, &pidl);
     SHGetPathFromIDList(pidl,szDesktop);
     // 释放
     pMalloc->Free(pidl);
     pMalloc->Release();
     // 显示结果
     printf("My Document:\t %s\n",szMyDocument);
     printf("DeskTop:\t %s\n",szDesktop);
}
```

12.1.3　绑定、遍历、属性获取

本小节将通过实例介绍通过 Shell API 编写更为一般的文件操作程序，包括：
◇将 IShellFolder 接口绑定到目录对象上，以获取目录获取、子目录等相关信息；
◇获取 IEnumIDList 接口，用于遍历目录中包含的对象；
◇获取和显示文件和目录对象的相关属性。

1．关键 API、接口与数据结构

(1) SHGetDesktopFolder 函数。

此函数是获取以 IShellFolder 接口形式返回的桌面文件夹，函数原型如下：

```
HRESULT SHGetDesktopFolder(
        IShellFolder **ppshf
);
```

参数 ppshf 指向返回的 IShellFolder。

(2) IShellFolder 接口。

IshellFolder 是 Windows Shell 程序对目标进行管理的一个重要接口。每一个目录对应一个实例化的 IshellFolder 接口。IshellFolder 接口的成员包括 EnumObj ects、GetAttributesOf、GetDisplayNameOf 等。

(3) IEnumIDList 接口。

IEnumIDList 接口提供了一组标准的方法，用于遍历 PIDL (Item identifier lists 的指针)，其成员包括 Clone、Next、Reset、Skip 等。

(4) STRRET 结构。

很多 IShellFolder 接口成员的返回的字符串是以 STRRET 结构的形式存在的，STRRET 结构如下：

```
typedef struct _STRRET {
  UINT uType;
  union
  {
    LPWSTR pOleStr;
    UINT uOffset;
    char cStr[MAX_PATH];
  } DUMMYUNIONNAME;
} STRRET, *LPSTRRET;
```

2．实例 12-2 列举“回收站”中的文件和目录

本实例将能够获得并显示已经被删除到回收站中的文件和目录在删除前的路径，实例代码如下：

```
/*************************************
* VOID GetSpecialFolder()
* 功能：遍历并显示回收站中的文件
*************************************/
DWORD ListFileInRecycleBin()
{
    CHAR pszPath[MAX_PATH];              // 保存路径
    // IShellFolder 接口
    IShellFolder *pisf = NULL;
    IShellFolder *pisfRecBin = NULL;
    // 获取“根”目录，桌面
    SHGetDesktopFolder(&pisfRecBin);
        IEnumIDList *peidl = NULL;       // 对象遍历接口
    LPITEMIDLIST pidlBin = NULL;
    LPITEMIDLIST idlCurrent = NULL;
```

```
    LPMALLOC pMalloc = NULL;
    // 分配
    SHGetMalloc(&pMalloc);
    // 回收站位置
    SHGetFolderLocation(NULL, CSIDL_BITBUCKET, NULL, 0, &pidlBin);
    // 绑定回收站对象
    pisfRecBin->BindToObject(pidlBin,NULL,IID_IShellFolder,(void **) &pisf);
    // 列举回收站中的对象，得到 IEnumIDList 接口，包括 SHCONTF_FOLDERS、
    // SHCONTF_NONFOLDERS、SHCONTF_INCLUDEHIDDEN 类型的对象
    pisf->EnumObjects(NULL,
         SHCONTF_FOLDERS | SHCONTF_NONFOLDERS |SHCONTF_INCLUDEHIDDEN,
         &peidl);
    STRRET strret;
    ULONG uFetched;
    HANDLE hOutPut = GetStdHandle(STD_OUTPUT_HANDLE);
    printf("\nFiles In Recycle Bin:\n");
    while(1)
    {
         // 遍历 IEnumIDList 对象，idlCurrent 为当前对象
         if(peidl->Next(1,&idlCurrent,&uFetched) == S_FALSE)
              break;
         // 获取回收站当前对象当前的路径，这里没有输出结果，读者可自行修改
         SHGetPathFromIDList(idlCurrent,  pszPath);
         // DisplayName，删除前的路径
         pisf->GetDisplayNameOf(idlCurrent,SHGDN_NORMAL,&strret);
         // 显示，printf 可能会造成字符编码不正确
         WriteConsoleW(hOutPut,L"\t",1,NULL,NULL);
         WriteConsoleW(hOutPut,strret.pOleStr,lstrlenW(strret.pOleStr),NULL,NULL);
         WriteConsoleW(hOutPut,L"\n",1,NULL,NULL);
    }
    // 释放资源
    pMalloc->Free(pidlBin);
    pMalloc->Free(strret.pOleStr);
    pMalloc->Release();
    peidl->Release();
    pisf->Release();
    return 0;
}
```

在本实例中首先以 CSIDL-_ BITBUCKET 为参数通过 SHGetFolderLocation 函数获取了回收站的位置 LPITEMIDLIST pidlBin，然后调用 IShellFolder*pisflRecBin 对象的 BindToObject 成员进行绑定；得到 IShellFolder*pisf,公分母通过 EnumObjects 成员得到 IEnumIDList *peidl 之后就可以进行遍历。

在显示结果时，由于 STRRET strret 保存的是 UNICODE 编码的字符串，在使用输出函数进行结果输出时，需要注意字符编码。

3．运行结果

如图 12-1 所示为是实例 12-1 和 12-2 的程序运行结果。

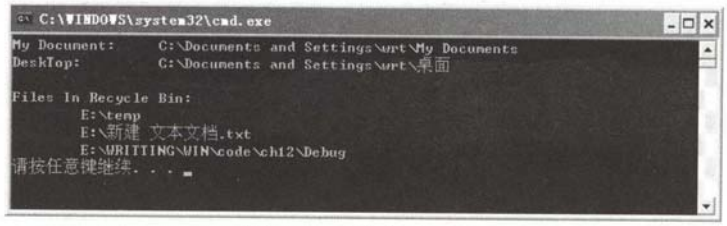

图 12-1　实例运行结果

12.1.4 浏览文件对话框

本节将演示如何使用“浏览文件夹”对话框，使用此对话框后，软件用户可以通过此对话框来选择路径。

实例 12-3 使用“浏览文件夹”对话框

本实例将演示如何在程序中弹出“浏览文件夹”对话框，并得到用户选择的文件夹路径。示例代码如下：

```
/* ************************************
* fb.c 浏览文件夹对话框
**************************************/
/* 头文件 */
#include <Windows.h>
#include <shlobj.h>
/* 函数申明 */
DWORD Browse(HWND hwnd) ;

/************************************
* WinMain
* 功能：程序入口点，调用 Browse
**************************************/
int WINAPI WinMain(
                    HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow
                    )
{
     Browse(NULL);
}

/************************************
* WinMain
* 功能：弹出“浏览文件夹”对话框，并获取用户选择的文件夹目录
* 参数：HWND hwnd,父窗口句柄
**************************************/
DWORD Browse(HWND hwnd)
{
     // 用于保存路径
     CHAR szRoot[MAX_PATH];
     CHAR szChoose[MAX_PATH];
     CHAR szDisplayName[MAX_PATH];
     // 相关变量
     LPITEMIDLIST pidlRoot = NULL;
     LPITEMIDLIST pidlSelected = NULL;
     BROWSEINFO bi = {0};
     LPMALLOC pMalloc = NULL;
     // “浏览文件夹”的根路径，开发人员可根据情况选择，比如只浏览“我的文档”
     SHGetFolderLocation(NULL, CSIDL_DESKTOP, NULL, 0, &pidlRoot);
     SHGetPathFromIDList(pidlRoot,szRoot);
     // 填充 BROWSEINFO 结构
     bi.hwndOwner = hwnd;
     bi.pidlRoot = pidlRoot;
     bi.pszDisplayName = szDisplayName;
     bi.lpszTitle = "Choose a target";
     bi.ulFlags = 0;
```

```
    bi.lpfn = NULL;
    bi.lParam = 0;
    // 弹出对话框
    pidlSelected = SHBrowseForFolder(&bi);
    // DisplayName
    MessageBox(NULL,szDisplayName,"Display Name:",MB_OK);
    // 选择的文件夹
    SHGetPathFromIDList( pidlSelected, szChoose );
    MessageBox(NULL,szChoose,"Choose:",MB_OK);
    // 释放
    ILFree(pidlRoot);
    return 0;
}
```

程序运行后，首先弹出“浏览文件夹”对话框，如图 12-2 所示，在对话框中选择了“我的文档后”先后弹出消息框如图 12-3 和图 12-4 所示。

图 12-2　浏览文件夹

图 12-3　显示名

图 12-4　选择的文件夹路径

12.2　文件协助(File Associations)

File Associations 可以指定定 Shell 在显示、处理特定文件类型的文件时的方式，包括如下几个方面：

◇　双击文件时，哪个应用程序应该被用来打开这个文件；

◇文件的图标；

◇在 Explorer 中浏览时显示的文件类型等；

◇文件的右键菜单中显示的指令（菜单项）；

◇一些用户界面细节，比如当鼠标指针停留在图标上时所弹出的提示信息(Infotip)。

与 File Associations 相关的一个重要概念是“文件类型”，每一种文件类型对应特定的文件扩
展名。为了使 Shell (Explorer. exe)能够正确识别文件类型，需要在注册表中进行相关的注册。只
有在注册表中正确注册了的文件类型 Shell 才知道如何选择文件的图标、如何指定双击文件时打
开文件所需的应用程序。

12.2.1 文件类型相关注册表键值

1．文件类型

具有同一种文件扩展名的文件属于同一个文件类型。如果需要 Shell 识别这种文件类型，需要在注册表中注册。注册一个文件类型包括两个方面，一是在 HKEY_CLASSES_ROOT 设置扩展名子键,并将默认值设置为文件扩展名对应的 ProgID (ProgrammaticIdentifiers，可编程标识符)；

二是在 HKEY_CLASSES_ROOT 键值下建立 ProgID 子键，并设置相关属性。

2．扩展名键

HKEY_CLASSES_ROOT 的扩展名子键除了默认键值应为 ProgID,还可以设置其他键值或子键。包括 PerceivedType 和 Content Type 键值和 penWithProgids 和 OpenWithList 子键等。OpenWithList 和 OpenWithProgids 子键分别设置了“打开方式”列表和“打开方式”一“选择程序”对话框中可选的项。

3．可编程标识符

ProgID 有多种子键:DefaultIcon、CurVer、shell、shellex 等,还具有多种键值 InfoTip、FriendlyTypeName 等。

ProgID 的默认键值指定了文件类型的描述；InfoTip 键值指定了用户将鼠标指针移动到该种类型的文件时，所弹出的提示信息。

DefaultIcon 子键指定了该种文件类型的默认图标。CurVer 指明了当前版本。

shell 子键包括若干的命令处理程序(Handler)，有一些默认 Handler（open、play 等）也可以注册自定义的 Handler。

shellex 中包括若干 Shell 扩展的 Handler，用于新加右键菜单项、文件拖曳菜单、图标、属性对话框、提示对话框等，包括 Context Menu handler、Data handler、Drop handler、Icon handler、Property sheet handler、Infotip handler、Copy hook handler 等类型。这些键值的具体设置方法参见本章后续实例。

12.2.2 为文件指定默认打开程序

通过注册表可以为特定文件类型的文件指定默认打开的程序。指定默认打开程序后，在 exploerer.exe 中双击或选择“右键一打开”就可以打开文件。

实例 12-4 文件类型、文件默认打开程序及图标

本实例是一个.reg 文件的内容，双击文件导入注册表后，会在注册表中注册.wln 扩展名。指定了这种类型的扩展名的默认图标，并指定的双击该类型的文件后的运行命令。

```
REGEDIT4

[HKEY_CLASSES_ROOT\.win]
  @="WINFile"
[HKEY_CLASSES_ROOT\WINFile]
  @="File Type Example"

[HKEY_CLASSES_ROOT\WINFile\DefaultIcon]
  @="shell32.dll,-152"
[HKEY_CLASSES_ROOT\WINFile\shell\open\command]
  @ = "NOTEPAD.EXE %1"
```

以上是 win.reg 文件内容。在系统中注册了.wm 扩展名，注册了名为“WINFile”的 ProgID，指定了文件类型为“File Type Example”（WINFile 项的默认值），并指明图标为 she1132.dll 的 152 号；打开此类型文件的命令为 NOTEPAD.EXE，%1 是指将第一个参数（双

击的文件名)传给 notepad.exe(相当于以被双击文件的文件名为参数运行了 notepad.exe)。

如图 12-5 所示，在导入该注册表文件前，Shell 无法识别扩展名为.wln 的文件，没有图标，也不能通过双击打开。

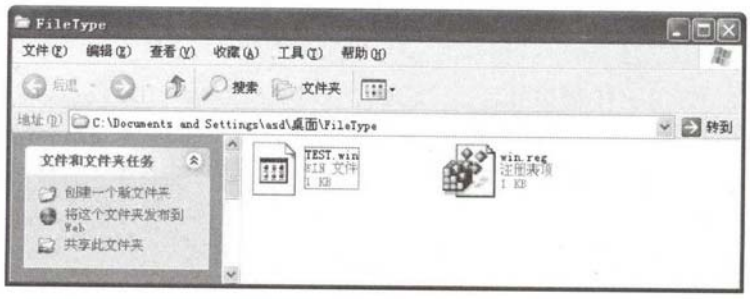

图 12-5　注册文件类型前

双击 win.reg，导入注册表后注销、重新登录，系统已经可以识别到.wln 的文件。类型是通过注册表注册的“File Type Example”，如图 12-6 所示。

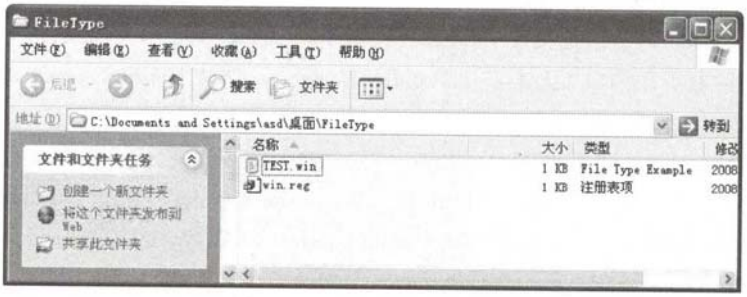

图 12-6　注册文件类型后

如图 12-7 所示，双击可以打开此文件。

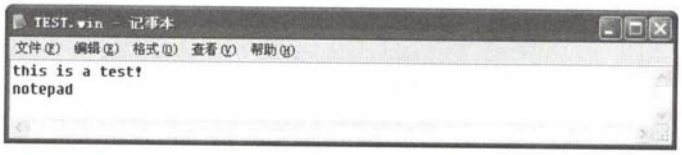

图 12-7　双击打开文件

通过以上方法。可以向 Shell 注册一个新的文件类型。但是直接修改注册表键值后，explorer.exe 并不会立即关联到新的设置。所以需要注销重新登录，以重启 explorer.exe。

SHChangeNotify 函数可以使 explorer.exe 重新加载注册表设置，用户不用重新登录，新注册的文件类型就可以生效。使用如下参数调用 SHChangeNotify：

```
SHChangeNotify(SHCNE_ASSOCCHANGED, SHCNF_IDLIST, NULL, NULL);
```

12.2.3　定制文件类型的图标

在实例 12-4 中已经说明了影响文件图标的注册表键值。

```
[HKEY_CLASSES_ROOT\WINFile\DefaultIcon]
  @="shell32.dll,-152"
```

本键值说明 ProgID 为 WINFile 的文件类型的默认图标是 she1132.dll 的第 152 号图标。

12.3 Shell扩展

Shell 可以实现扩展，Shell 扩展程序可以实现以下功能。

◇为特定文件类型、所有文件类型、网络邻居、回收站、驱动器、网络共享文件夹、打印机、光盘驱动器、目录、文件夹背景（窗口的空白处）等 Shell 对象的右键菜单(称作上下文件菜单，Context Menu)增加菜单项。

◇当选中上下文件菜单的菜单项时进行相关处理。

◇定制将文件、文件夹拖拽至其他目录中时的操作。

◇定制右键拖曳 Shell 对象图标的菜单及操作。

◇根据文件内容定制图标。

◇定制属性对话框及分页项。

◇定制提示对话框（当鼠标指针停留在图标时弹出的提示信息）。

◇ 当使用“详细信息”显示图标时定制列。

◇加速或禁止目录、特定文件的移动、复制、删除、重命名等操作。

◇在开始菜单或 Explorer 中增加定制的搜索引擎。

从这些功能还可以扩展出很多特殊的应用，比如：

◇监视文件或目录的移动。

◇监视回收站动作，如还原项目、清空等。

12.3.1 对象及概念

Shell 扩展处理例程的代码实现位于 DLL 文件中，是一个符合 COM 接口的对象。在完成 DLL 的设计后需要在注册表进行注册，Shell 扩展才能正常工作。

1．Handler（处理例程）

处理例程(Handler)是一系列函数及接口，当 Shell 事件发生时，会调用处理例程进行处理。

比如当在图标上点击右键时，会调用上下文件菜单处理例程(Context Menu Handler)，选中上下文菜单中定制的菜单项后，也会调用相关的函数接口用于处理菜单选择事件。Shell 扩展程序 DLL

应包括如下导出函数。

◇ DllMain:标准的 DLL 入口函数。

◇DllGetClassObj ect：这个导出函数的作用是给出类厂(class factory)接口。 ‘

◇ DllCanUnloadNow:表示该 DLL 是否仍然在使用，系统是否可以卸载此 DLL。

和其他所有 COM 程序一样，Shell 扩展程序必须实现一个 IUnlnown 接口和一个类厂。大多数情况下还需要实现一个 IPersistFile 或者 IShellExtInit 接口。

实例 12-6～实例 12-11 具体说明了处理例程具体的方法。

2．ProgID 及特殊 ProgID

12.2 节已经介绍过，每一种文件类型都对应一个 ProgID，ProgID 代表了一种文件类型。一些特殊的 ProgID 如表 12.1 所示。

表 12.1　　特殊 ProgID

ProgID	描　述	支持的处理程序
*	所有的文件	ContextMenuHandlers， PropertyHandler， shell\<动作>（包括 shell\open，shell\play 等）
AllFileSystemObjects	文件和文件夹	ContextMenuHandler，PropertyHandler，shell\<动作>
Folder	所有文件夹	ContextMenuHandler，PropertyHandler，shell\<动作>
Directory	目录	ContextMenuHandler，PropertyHandler，shell\<动作>
Directory\Background	文件的背景（窗口内空白处）	ContextMenuHandler
Drive	驱动器	ContextMenuHandler，PropertyHandler，shell\<动作>
Network	网络位置	ContextMenuHandler，PropertyHandler，shell\<动作>
Printers	打印机	ContextMenuHandler，PropertyHandler
DVD	DVD 驱动器	ContextMenuHandler，PropertyHandler，shell\<动作>

ProgID 的 shellex 子键指明了 ProgjD 对应的处理例程。这些子键包括 ColumnHandlers（使用“详细信息”方式查看文件列表，显示列时调用）、ContextMenuHandlers（右键菜单时调用）、CopyHookHandlers（移动、复制、删除、重命名等操作时调用）、DragDropHandlers（右键拖曳菜单时调用）、PropertySheetHandlers（查看 Shell 对象属性时调用）、DataHandler、DropHandler、IconHandler（显示图标时调用，用于个性化图标）等。这些处理例程的详细注册方法在实例 12-5 中有详细说明，而这些处理例程的程序实现方法在实例 12-6～实例 12-11 中有详细说明。

12.3.2　CLSID，处理例程的GUID

GUID（Global unique identifier，全局唯一标识符）用于在系统中唯一标识一个对象。CLSID 是 GUID 在注册表中的表示，用于在注册表中用于唯一标识一个 COM 对象。GUID 是一个 128 位的数据，在注册表中的形式如下所示：

```
{00000000-1111-2222-3333-444444444444}
{8A3C6BCE-64F4-429c-A334-635BF51A98AA}
…
```

CLSID 可以有很多子键，在 Shell 扩展中常用到的是 InprocHandler32 和 InprocServer32 这两个子键，分别用于向 Shell 注册处理例程 DLL 和 in-process 服务 DLL。

在 C\C++语言程序中可以有如下两种方法定义 GUID。一种是使用 DEFINE_GUID 宏。

```
// {8A3C6BCE-64F4-429c-A334-635BF51A98AA}
DEFINE_GUID(MyGuid,
0x8a3c6bce, 0x64f4, 0x429c, 0xa3, 0x34, 0x63, 0x5b, 0xf5, 0x1a, 0x98, 0xaa);
```

一种是直接定义 GUID 类型的结构变量。

```
// {8A3C6BCE-64F4-429c-A334-635BF51A98AA}
static const GUID MyGuid =
{ 0x8a3c6bce, 0x64f4, 0x429c, { 0xa3, 0x34, 0x63, 0x5b, 0xf5, 0x1a, 0x98, 0xaa } };
```

GUID 的生成

程序员可以任意选择 GUID，但是为了保证 GUID 的惟一性，最好使用生成工具生成 GUID，GUID 的生成可以使用 SDK 提供的工具 GuidGen.exe。

在 SDK 安装目录下找到 GuidGen.exe，运行。可以生成新的 GUID，并给出了 GUID 的各种形式。运行效果如图 12-8 所示。

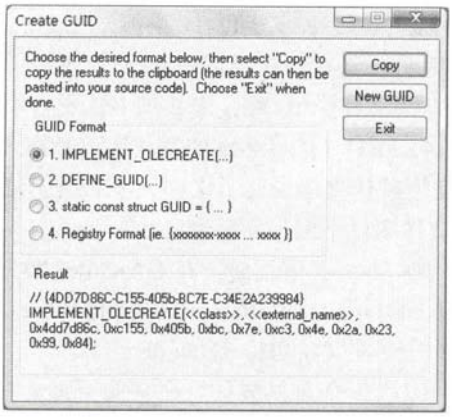

图 12-8　生成 GuidGen.exe

12.3.3　注册Shell扩展

本小节将给出一个注册 Shell 扩展的实例。

实例 12-5 注册 Shell 扩展处理程序

本实例是 ShellExt.reg 注册表文件内容，在注册 Shell 扩展时，双击此文件导入注册表。

```
REGEDIT4

[HKEY_CLASSES_ROOT\CLSID\{87b9bd00-c65c-11cd-a259-00dd010e8c28}]
  @="Shell Extension Sample"
[HKEY_CLASSES_ROOT\CLSID\{87b9bd00-c65c-11cd-a259-00dd010e8c28}\InProcServer32]
  @="shellext.dll"
    "ThreadingModel"="Apartment"

[HKEY_CLASSES_ROOT\.gak]
  @="GAKFile"
[HKEY_CLASSES_ROOT\GAKFile]
  @="Shell Extension file"
[HKEY_CLASSES_ROOT\GAKFile\shellex\IconHandler]
  @="{87b9bd00-c65c-11cd-a259-00dd010e8c28}"
[HKEY_CLASSES_ROOT\GAKFile\shellex\ContextMenuHandlers]
  @="GAKMenu"
[HKEY_CLASSES_ROOT\GAKFile\shellex\ContextMenuHandlers\GAKMenu]
  @="{87b9bd00-c65c-11cd-a259-00dd010e8c28}"
[HKEY_CLASSES_ROOT\GAKFile\shellex\PropertySheetHandlers]
  @="GAKPage"
[HKEY_CLASSES_ROOT\GAKFile\shellex\PropertySheetHandlers\GAKPage]
  @="{87b9bd00-c65c-11cd-a259-00dd010e8c28}"
[HKEY_CLASSES_ROOT\directory\shellex\CopyHookHandlers\GAKsCopyHook]
  @="{87b9bd00-c65c-11cd-a259-00dd010e8c28}"
[HKEY_CLASSES_ROOT\*\shellex\CopyHookHandlers\GAKsCopyHook]
  @="{87b9bd00-c65c-11cd-a259-00dd010e8c28}"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Shell Extensions\Approved]
  "{87b9bd00-c65c-11cd-a259-00dd010e8c28}"="Shell Extension Sample"
```

此文件首先注册了一个 CLSID {87b9bd00-c65c-11cd-a259-00dd010e8c28)，并设置默认键值为“Shell Extension Sample”。设置了 CLSID {87b9bd00-c65c-11cd-a259-00dd010e8c28）的 InProcServer32 子键,注册了一个 in-process 服务 DLL。这个 DLL 的文件名是 shellext.dll,是一个 Shell 扩展的处理例程 DLL。实例 12—6～12 -11 讲解了 shellext.dll 的实现。InProcServer32 子键的 ThreadingModel 键值说明对象实例在单线程单元(STA)中创建。

之后，注册表文件注册了一个文件类型，将.gak 扩展名的文件类型向系统注册，并指

明了.gak 扩展名所对应的 ProgID 为“GAKFile”（扩展名键的默认键值是指定扩展名对应的 ProgID)。

然后在 ProgID 子键中，使用默认键值说明了文件类型的描述使“Shell Extension file”这个描述将在 Explorer.exe 中此类型文件图标下侧显示出来。

然后开始了最关键的 ProgID Shell 扩展注册。在 GAKFile 类型的 ProgID shellex 子键下分别注册了 IconHandler、ContextMenuHandlers 和 PropertySheetHandlers.并注册了目录(directory)的 CopyHookHandlers 和所有文件（*）的 CopyHookHandlers。

所有的 Handler 子键都可以有一个可以被任意命名的子键，在此子键下的默认键值指向了 Handler DLL 的 CLSID。

以上步骤完成了 Shell 扩展的注册，在 Shell 运行后，会从注册表中读取相关的设置，将在进行相关文件类型的处理时，调用注册的 Handler。

12.3.4 COM程序开发基础

Shell 扩展程序是 COM（Component Object Model，组件对象模型）程序。

COM 是一个平台无关、分布式、面向对象的程序接口标准，一般用于扩展应用程序的功能。大多数微软公司开发的应用程序都支持 COM 接口扩展，COM 程序一般是 DLL 文件（ActiveX 程序是.ocx 扩展名的文件）

COM 程序被提供给主调程序调用。COM 程序为主调程序提供固定的接口(DllGetClassObject 导出函数)主调程序先通过固定的函数接口来获得其他的接口。Shell 扩展中通过在注册表中配置的情况来在不同情况调用不同的接口函数。

不同的 COM 程序具有不同的接口，但是所有的接口都是从类厂（Class Factory）和 IUnknown 接口获得的。所以 COM 程序必须实现类厂和 IUnknown 接口。

实际上，类厂和 IUnknown 接口所返回的其他接口，都是以函数指针的形式返回的。因此任，何具有结构体指针，并可以通过函数指针来调用函数的程序语言都可以用来编写 COM 程序。C++就是一个很好的选择。

12.3.5 编写Handler程序

本节将通过实例 ShellExt.dll 来说明如何编写 Shell 扩展的 Handler 程序。

实例 12-6 介绍 Shell 扩展程序的 COM 接口规范，只有理解了 Shell 扩展作为 COM 程序的一种，所必须遵守的接口规范才可能理解 Shell 扩展程序的工作原理。

实例 12-6 介绍了 Shell 扩展程序自己的接口。

实例 12-7～12-11 是各个 Handler 实现原理的介绍。

1．实例 12-6 Shell 扩展处理程序的 COM 接口，SheIIExt.cpp

Shell 扩展是 COM 程序,，需要符合 COM 接口规范。本实现先讨论 ShellExt.dll 的 COM 接口。

由于 COM 程序是由其他程序调用的，因此 COM 程序存在一个主调程序。如果 COM 是对 Shell 的扩展，那么主调程序就是 Explorer.exe，如果 COM 程序是对 IE 浏览器的扩展，那么主调程序就是 iexplore.exe。

ShellExt.dll 工程包括 ShellExt.cpp、ShellInit.cpp、CtxMenu.cpp、CopyHook.cpp、IconHlr.cpp、PropShet.cpp 六个源文件及一些头文件、资源文件、模块定义文件。其中 ShellExt.cpp 实现了相关调用接口。

ShellExtdll 共有 3 个导出函数。一个是 DllMain，另外两个是 COM 接口函数，DllCanUnloadNow 和 DllGetClassObject。3 个导出函数全部在 ShellExt.Cpp 源文件中实现。

以下是 ShellExt.cpp 的完整代码。

```
//////////////////////////  ShellExt.Cpp  ////////////////////////////
#include "priv.h"

#pragma data_seg(".text")
#define INITGUID
#include <initguid.h>
#include <shlguid.h>
#include "shellext.h"
#pragma data_seg()

UINT     g_cRefThisDll = 0;    //DLL 的引用次数
HINSTANCE g_hmodThisDll = NULL; //本地 DLL 句柄

extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
   if (dwReason == DLL_PROCESS_ATTACH)
   {
      ODS("In DLLMain, DLL_PROCESS_ATTACH\r\n");
      // 保留实例句柄
      g_hmodThisDll = hInstance;
   }
   else if (dwReason == DLL_PROCESS_DETACH)
   {
      ODS("In DLLMain, DLL_PROCESS_DETACH\r\n");
   }
   return 1;
}
STDAPI DllCanUnloadNow(void)
{
   ODS("In DLLCanUnloadNow\r\n");
   return (g_cRefThisDll == 0 ? S_OK : S_FALSE);
}
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID *ppvOut)
{
   ODS("In DllGetClassObject\r\n");
   *ppvOut = NULL;
   if (IsEqualIID(rclsid, CLSID_ShellExtension))
   {
      CShellExtClassFactory *pcf = new CShellExtClassFactory;
      return pcf->QueryInterface(riid, ppvOut);
   }
   return CLASS_E_CLASSNOTAVAILABLE;
}
CShellExtClassFactory::CShellExtClassFactory()
{
   ODS("CShellExtClassFactory::CShellExtClassFactory()\r\n");
   m_cRef = 0L;
   g_cRefThisDll++;
}

CShellExtClassFactory::~CShellExtClassFactory()
{
   g_cRefThisDll--;
```

```
}

STDMETHODIMP CShellExtClassFactory::QueryInterface(REFIID riid,
                                                   LPVOID FAR *ppv)
{
    ODS("CShellExtClassFactory::QueryInterface()\r\n");
    *ppv = NULL;
    // 本对象的所有接口都是对象指针
    if (IsEqualIID(riid,IID_IUnknown) || IsEqualIID(riid,IID_IClassFactory))
    {
        *ppv = (LPCLASSFACTORY)this;
        AddRef();
        return NOERROR;
    }

    return E_NOINTERFACE;
}

STDMETHODIMP_(ULONG) CShellExtClassFactory::AddRef()
{
    return ++m_cRef;
}

STDMETHODIMP_(ULONG) CShellExtClassFactory::Release()
{
    if (--m_cRef)
        return m_cRef;
    delete this;
    return 0L;
}

STDMETHODIMP CShellExtClassFactory::CreateInstance(LPUNKNOWN pUnkOuter,
                                                   REFIID riid,
                                                   LPVOID *ppvObj)
{
    ODS("CShellExtClassFactory::CreateInstance()\r\n");
    *ppvObj = NULL;
    // Shell 扩展通常不支持继承
    if (pUnkOuter)
        return CLASS_E_NOAGGREGATION;
    // 创建主 Shell 扩展对象，通过 IDD_IshellExtInit 调用号
    // 调用 QueryInterface 来初始化
    LPCSHELLEXT pShellExt = new CShellExt();  //创建 CShellExt 对象
    if (NULL == pShellExt)
        return E_OUTOFMEMORY;
    return pShellExt->QueryInterface(riid, ppvObj);
}


STDMETHODIMP CShellExtClassFactory::LockServer(BOOL fLock)
{
    return NOERROR;
}

// ********************** CShellExt ************************
CShellExt::CShellExt()
{
    ODS("CShellExt::CShellExt()\r\n");
    m_cRef = 0L;
    m_pDataObj = NULL;
    g_cRefThisDll++;
```

```
}

CShellExt::~CShellExt()
{
    if (m_pDataObj)
        m_pDataObj->Release();
    g_cRefThisDll--;
}

STDMETHODIMP CShellExt::QueryInterface(REFIID riid, LPVOID FAR *ppv)
{
    *ppv = NULL;
    if (IsEqualIID(riid, IID_IShellExtInit) || IsEqualIID(riid, IID_IUnknown))
    {
        ODS("CShellExt::QueryInterface()==>IID_IShellExtInit\r\n");
        *ppv = (LPSHELLEXTINIT)this;
    }
    else if (IsEqualIID(riid, IID_IContextMenu))
    {
        ODS("CShellExt::QueryInterface()==>IID_IContextMenu\r\n");
        *ppv = (LPCONTEXTMENU)this;
    }
    else if (IsEqualIID(riid, IID_IExtractIcon))
    {
        ODS("CShellExt::QueryInterface()==>IID_IExtractIcon\r\n");
        *ppv = (LPEXTRACTICON)this;
    }
    else if (IsEqualIID(riid, IID_IPersistFile))
    {
        ODS("CShellExt::QueryInterface()==>IPersistFile\r\n");
        *ppv = (LPPERSISTFILE)this;
    }
    else if (IsEqualIID(riid, IID_IShellPropSheetExt))
    {
        ODS("CShellExt::QueryInterface()==>IShellPropSheetExt\r\n");
        *ppv = (LPSHELLPROPSHEETEXT)this;
    }
    else if (IsEqualIID(riid, IID_IShellCopyHook))
    {
        ODS("CShellExt::QueryInterface()==>ICopyHook\r\n");
        *ppv = (LPCOPYHOOK)this;
    }
    if (*ppv)
    {
        AddRef();
        return NOERROR;
    }
    ODS("CShellExt::QueryInterface()==>Unknown Interface!\r\n");
    return E_NOINTERFACE;
}

STDMETHODIMP_(ULONG) CShellExt::AddRef()
{
    ODS("CShellExt::AddRef()\r\n");
    return ++m_cRef;
}

STDMETHODIMP_(ULONG) CShellExt::Release()
{
    ODS("CShellExt::Release()\r\n");
    if (--m_cRef)
        return m_cRef;
    delete this;
```

```
    return 0L;
}
////////////////////// END OF FILE //////////////////////
```

在函数 DllMain 中，并没有完成任何实质的功能，只是在加载时保留了本模块的句柄。在卸载时输出的调试字符串，ODS 宏是调试字符串输出函数 OutputDebugString 的重定义。

```
#define ODS(sz) OutputDebugString(sz)
```

OutputDebugString 函数输入的字符串在调试器的信息输出窗口可以查看到，Visual Studio 和 Windbg 都可以。

(1) DllGetClassObjecta

DllGetClassObject 函数的主要功能是为主调程序（COM 程序都是实现好后供其他程序调用的，调用 COM 程序的程序就是 COM 程序的主调程序）返回调用接口。函数原型如下：

```
STDAPI DllGetClassObject(
  REFCLSID rclsid,
  REFIID riid,
  LPVOID * ppv
);
```

rclsid：输入参数，传递的是当前代码和数据相应 CLSID，通过 CLSID 判断是否是本扩展程序需要处理的请求。

riid：也是输入参数，指明主调程序需要 DllGetClassObject 函数返回的接口类型。在本实例里判断了此参数是否为 IID_IID_IUnknown 或 IID_IClassFactory，如果是则返回 CShellExtClassFactory 类，CSheDExtClassFactory 类继承自类厂(Classs Factory)接口。在本实例中的 QueryInterface 成员函数实现了 riid 参数的判断和类厂的返回。DllGetClassObject 调用了 QueryInterface 成员函数。

ppv：参数是输出参数，用于返回接口。

源文件中类 CShellExtClassFactory 继承自 IClassFactory 接口。

IClassFactory 接口由 DllGetClassObject 函数将接口返回给主调程序后，主调用程序会调用此接口。

简而言之，DllGetClassObject 在 COM 程序设计中具有重要的作用。COM 程序会实现各种不同的接口（IPersistFile、IShellExtInit 等）供主调程序调用。主调用程序如何才能知道 COM 程序实现了哪些接口，又如何才能得到这些接口的实例呢？是通过类厂接口，而在最开始时，主调程序也不没用类厂接口的实例，DllGetClassObject 的作用就是向主调用程序返回类厂接口实例。

(2) IclassFactory。

在 COM 程序发布实现 IClassFactory 接口。IClassFactory 接口的作用是根据主调程序的要求返回本 COM 程序所实现了的接口，供主调函数调用。IClassFactory 接口必须要实现两个成员函数，CreateInstance 和 LockServer，主调程序会调用这两个成员函数。

CreateInstance 成员函数由主调程序调用，作用实例化主调程序所需要的接口，并返回接口实例。在实例 12-6 中，返回的是 CShellExt 接口的实例。

也可以根据 CreateInstance 函数的 riid 来判断主调用函数需要类厂返回什么样的接口。在本实例中，无论要求的是什么样的接口，都返回的是 CShellExt，因为所要求的 Shell 扩展功能都已经在 CShellExt 类中实现了。CShellExt 类实现了多个接口的功能。在程序设计时，程序员可以根据需要实现不同的类。

在实例 12-6 中，CreateInstance 调用了 CShellExt 类的 QueryInterface 成员函数，由 QueryInterface 成员函数完成了参数的判断和接口的实例化，并通过 ppv 参数返回了接口实例。

(3) Iunknown。

任何一个 COM 程序都必须要实现两个接口，一是 IClassFactory 另外一个就是 Iunknown。

IUnknown 需要有 3 个成员函数：QueryInterface、AddRef 和 Release。

QueryInterface 的功能与其他接口类似，AddRef 和 Release 分别用于增加和减少 COM 对象的引用次数。当 COM 被引用及被释放时，主调程序将会调用这两个接口。

在实例 12-6 中，IUnknown 并没有什么特殊的功能，而是与 IClassFactory 共用了同一个类 CShellExtClassFactory。在 CShellExtClassFactory 中实现了 QueryInterface、AddRef 和 Release。正因为如此，在 DllGetClassObject 实现时，无论 riid 是Ⅲ）IUnknown 还是 IID_IClassFactory 都返回了同一个类。因为在这里 CShellExtClassFactory 类既是 IUnknown 的实现也是 IClassFactory 的实现。

(4) DllCanUnloadNOW。

本实例的 DllCanUnloadNow 函数没有实现特殊的功能。

2. 实例 12-7 Shell 扩展处理程序的接口，Shelllnit.cpp

Shell 扩展程序必须实现 IShellExtInit 接口。IShellExtInit 接口应该具有成员函数 Initialize，Initialize 函数用于初始化 Shell 扩展程序所实现的属性页扩展处理例程、上下菜单处理例程、拖拽处理例程等。

上一个实例中 IClassFactory 的 CreateInstance 实例返回了 CShellExt，CShellExt 类就是 IShellExtInit 接口的实现（也是其他几个接口的实现，后面几个实例会说明）。

下面分析 CShellExt 类的 Initialize 函数是如何初始化这几种处理例程的。

ShellInit.cpp 文件内容如下：

```
#include "priv.h"
#include "shellext.h"

STDMETHODIMP CShellExt::Initialize(LPCITEMIDLIST pIDFolder,
                                   LPDATAOBJECT pDataObj,
                                   HKEY hRegKey)
{
    ODS("CShellExt::Initialize()\r\n");
    // 初始化可以使用多次
    if (m_pDataObj)
        m_pDataObj->Release();
    // duplicate the object pointer and registry handle
    if (pDataObj)
    {
        m_pDataObj = pDataObj;
        pDataObj->AddRef();
    }
    return NOERROR;
}
```

此成员函数在 Shell 需要弹出上下文菜单或弹出属性页前会被调用，以实例化相关接口。

参数 pIDFolder 是指向 ITEMIDLIST 类型的变量，用于标识哪个对象的菜单将会被弹出。如果是属性页扩展，那么此参数为 NULL;如果是上下文件菜单扩展，那么这个参数代表了所需弹出菜单的对象所在的目录。

参数 pDataObj 指向 IDataObj ect 接口，用于获取操作的对象。

参数 hkeyProgID 是表示目录类型的注册表值。

在-般隋况 T，Initialize 成员函数需要根据参数所指定的扩展类型来实例化不同的 Handler 接口，比如对 Context Menu Handlers 就应用实例化 IContextMenu 接口。但是在本实例中，由于这些 Handler 接口实现都是在 CShellExt 类中，已经在 DllGetClassObject 被调用时被实例化了，所以无需再判断扩展类型并进行实例化。在这里，Initialize 成员函数仅仅是进行了引用数目的增减。

3. 实例 12-8 上下文件菜单 Handler 实现，CtxMeun.cpp

CtxMeun.cpp 中的代码包括了 IContextMenu 接口的实现。在上一个实例中已经分析了 IContextMenu 接口的实例化过程，IContextMenu 的实现是在 CShellExt 类中。

主调程序通过 riid 参数 IID_IContextMenu 来获取接口，在 QueryInterface 函数中，

判断如果主调程序需要 IContextMenu 接口，则返回 this 指针（指向实例化的 CShellExt 类）。

```
else if (IsEqualIID(riid, IID_IContextMenu))
{
    ODS("CShellExt::QueryInterface()==>IID_IContextMenu\r\n");
    *ppv = (LPCONTEXTMENU)this;
}
```

IContextMenu 需要实现 GetCommandString、InvokeCommand 和 QueryContextMenu 这 3 个成员函数。

当 Shell 需要显示上下文件菜单时，会调用 QueryContextMenu 函数。QueryContextMenu 函数应当设置菜单中的项。在本实例中，QueryContextMenu 调用了 InsertMenu 函数，实现了向上下文菜单中增加菜单项的功能。

GetCommandString 用于向主调程序返回与菜单项关联的字符串。

当用户点击上下文菜单中的项后，InvokeCommand 会被调用。InvokeCommand 函数的实现中，应该处理菜单项被点击后所应该执行的动作。

如下是 CtxMeun.cpp 的文件内容，举例说明了 3 个接口函数应该如何实现。

```
#include "priv.h"
#include "shellext.h"

#define ARRAYSIZE(x) (sizeof(x)/sizeof(x[0]))
//实现 Icontextmenu 接口的 QueryContextMenu 接口函数.
STDMETHODIMP CShellExt::QueryContextMenu(HMENU hMenu,
                                         UINT indexMenu,
                                         UINT idCmdFirst,
                                         UINT idCmdLast,
                                         UINT uFlags)
{
    ODS("CShellExt::QueryContextMenu()\r\n"); //ODS 为打印调试格式，下同

    UINT idCmd = idCmdFirst;
    char szMenuText[64];
    char szMenuText2[64];
    char szMenuText3[64];
    char szMenuText4[64];
    BOOL bAppendItems=TRUE;
    //根据不同的情况为菜单项设置文字
    if ((uFlags & 0x000F) == CMF_NORMAL)  //Check == here, since CMF_NORMAL=0
    {                                     //资源管理器为正常状态
        ODS("CMF_NORMAL...\r\n");
        lstrcpyn(szMenuText, "&New .GAK menu 1, Normal File", ARRAYSIZE(szMenuText));
        lstrcpyn(szMenuText2, "&New .GAK menu 2, Normal File", ARRAYSIZE(szMenuText2));
        lstrcpyn(szMenuText3, "&New .GAK menu 3, Normal File", ARRAYSIZE(szMenuText3));
        lstrcpyn(szMenuText4, "&New .GAK menu 4, Normal File", ARRAYSIZE(szMenuText4));
    }
    else
        if (uFlags & CMF_VERBSONLY)
        {
            ODS("CMF_VERBSONLY...\r\n");
            lstrcpyn(szMenuText, "&New .GAK menu 1, Shortcut File", ARRAYSIZE(szMenuText));
            lstrcpyn(szMenuText2, "N&ew .GAK menu 2, Shortcut File", ARRAYSIZE(szMenuText2));
            lstrcpyn(szMenuText3, "&New .GAK menu 3, Shortcut File", ARRAYSIZE(szMenuText3));
            lstrcpyn(szMenuText4, "&New .GAK menu 4, Shortcut File", ARRAYSIZE(szMenuText4));
```

```
        }
      else
         if (uFlags & CMF_EXPLORE)//“文件夹”方式的资源管理器
         {
            ODS("CMF_EXPLORE...\r\n");
            lstrcpyn(szMenuText, "&New .GAK menu 1, Normal File right click in Explorer",
ARRAYSIZE(szMenuText));
            lstrcpyn(szMenuText2, "N&ew .GAK menu 2, Normal File right click in Explorer'
ARRAYSIZE(szMenuText2));
            lstrcpyn(szMenuText3, "&New .GAK menu 3, Normal File right click in Explorer'
ARRAYSIZE(szMenuText3));
            lstrcpyn(szMenuText4, "&New .GAK menu 4, Normal File right click in Explorer"
ARRAYSIZE(szMenuText4));
         }
      else
         if (uFlags & CMF_DEFAULTONLY)
         {
            ODS("CMF_DEFAULTONLY...\r\n");
            bAppendItems = FALSE;
         }
      else
         {
            char szTemp[32];
            wsprintf(szTemp, "uFlags==>%d\r\n", uFlags);
            ODS("CMF_default...\r\n");
            ODS(szTemp);
            bAppendItems = FALSE;
         }

      if (bAppendItems) //增加菜单项 hMenu 为右键菜单句柄
      {
         InsertMenu(hMenu, indexMenu++, MF_SEPARATOR|MF_BYPOSITION, 0, NULL);
         InsertMenu(hMenu,
                  indexMenu++,
                  MF_STRING|MF_BYPOSITION,
                  idCmd++,
                  szMenuText);
         InsertMenu(hMenu, indexMenu++, MF_SEPARATOR|MF_BYPOSITION, 0, NULL);
         InsertMenu(hMenu,
                  indexMenu++,
                  MF_STRING|MF_BYPOSITION,
                  idCmd++,
                  szMenuText2);
         InsertMenu(hMenu, indexMenu++, MF_SEPARATOR|MF_BYPOSITION, 0, NULL);
         InsertMenu(hMenu,
                  indexMenu++,
                  MF_STRING|MF_BYPOSITION,
                  idCmd++,
                  szMenuText3);
         InsertMenu(hMenu,
                  indexMenu++,
                  MF_STRING|MF_BYPOSITION,
                  idCmd++,
                  szMenuText4);
         //必须返回添加的菜单项数目
         return ResultFromShort(idCmd-idCmdFirst);
      }

      return NOERROR;
   }

   //实现 I Context Menu 接口的 Invoke command 成员
   STDMETHODIMP CShellExt::InvokeCommand(LPCMINVOKECOMMANDINFO lpcmi)
   {
```

```
    ODS("CShellExt::InvokeCommand()\r\n");
    HRESULT hr = E_INVALIDARG;
    if (!HIWORD(lpcmi->lpVerb))
    {
        //根据菜单弹出后用户在菜单上不同的选择作出不同的响应.
        UINT idCmd = LOWORD(lpcmi->lpVerb);
        switch (idCmd)
        {
            case 0:
                hr = DoGAKMenu1(lpcmi->hwnd,
                                lpcmi->lpDirectory,
                                lpcmi->lpVerb,
                                lpcmi->lpParameters,
                                lpcmi->nShow);
                break;
            case 1:
                hr = DoGAKMenu2(lpcmi->hwnd,
                                lpcmi->lpDirectory,
                                lpcmi->lpVerb,
                                lpcmi->lpParameters,
                                lpcmi->nShow);
                break;
            case 2:
                hr = DoGAKMenu3(lpcmi->hwnd,
                                lpcmi->lpDirectory,
                                lpcmi->lpVerb,
                                lpcmi->lpParameters,
                                lpcmi->nShow);
                break;
            case 3:
                hr = DoGAKMenu4(lpcmi->hwnd,
                                lpcmi->lpDirectory,
                                lpcmi->lpVerb,
                                lpcmi->lpParameters,
                                lpcmi->nShow);
                break;
        }
    }
    return hr;
}
//实现 I contextmenu 接口的 GetCommandString 成员函数
STDMETHODIMP CShellExt::GetCommandString(UINT_PTR idCmd,
                                         UINT uFlags,
                                         UINT FAR *reserved,
                                         LPSTR pszName,
                                         UINT cchMax)
{
    ODS("CShellExt::GetCommandString()\r\n");
    switch (idCmd)
    {
        case 0:
            lstrcpyn(pszName, "New menu item number 1", cchMax);
            break;
        case 1:
            lstrcpyn(pszName, "New menu item number 2", cchMax);
            break;
        case 2:
            lstrcpyn(pszName, "New menu item number 3", cchMax);
            break;
        case 3:
            lstrcpyn(pszName, "New menu item number 4", cchMax);
```

```
            break;
        }

        return NOERROR;
    }
    //不同的响应函数
    STDMETHODIMP CShellExt::DoGAKMenu1(HWND hParent,
                                LPCSTR pszWorkingDir,
                                LPCSTR pszCmd,
                                LPCSTR pszParam,
                                int iShowCmd)
    {
        ODS("CShellExt::DoGAKMenu1()\r\n");
        MessageBox(hParent, "Menu item 1!", "Shell Extension Sample", MB_OK);
        return NOERROR;
    }

    STDMETHODIMP CShellExt::DoGAKMenu2(HWND hParent,
                                LPCSTR pszWorkingDir,
                                LPCSTR pszCmd,
                                LPCSTR pszParam,
                                int iShowCmd)
    {
        ODS("CShellExt::DoGAKMenu2()\r\n");
        MessageBox(hParent, "Menu item 2!", "Shell Extension Sample", MB_OK);
        return NOERROR;
    }

    STDMETHODIMP CShellExt::DoGAKMenu3(HWND hParent,
                                LPCSTR pszWorkingDir,
                                LPCSTR pszCmd,
                                LPCSTR pszParam,
                                int iShowCmd)
    {
        ODS("CShellExt::DoGAKMenu3()\r\n");
        MessageBox(hParent, "Menu item 3!", "Shell Extension Sample", MB_OK);
        return NOERROR;
    }

    STDMETHODIMP CShellExt::DoGAKMenu4(HWND hParent,
                                LPCSTR pszWorkingDir,
                                LPCSTR pszCmd,
                                LPCSTR pszParam,
                                int iShowCmd)
    {
        ODS("CShellExt::DoGAKMenu4()\r\n");
        MessageBox(hParent, "Menu item 4!", "Shell Extension Sample", MB_OK);
        return NOERROR;
    }
```

分析代码可知，当右键单击注册（如何注册参见实例 12-5）了文件类型的文件时，会在右键菜单中增加新的菜单项。菜单项内容是“New .GAK menum*，Normal File”，如果单击指向此类型文件的快捷方式，那么会显示“New .GAK menu*，Shortcut File”；如果在以“文件夹”方式浏览的上下文菜单中右键单击，那么显示“New .GAK menu*，Normal File right click in Explorer”。

如果点击菜单中的相应项，会弹出对话框显示用户点击的项的内容（在具体实现时，也可以实现其他功能）。

如图 12-9 所示，在注册文件类型后，右键单击 gak 扩展名的文件已经可以显示出图形，右键菜单也增加了定制的项。

图 12-9　右键单击 gak 文件

为 gak 文件创建快捷方式，右键菜单与 gak 文件不同。图 12-10 显示了这种不同。

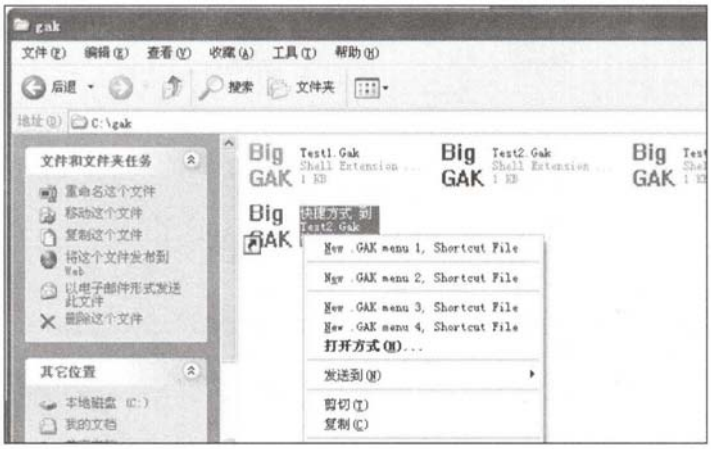

图 12-10　右键单击 gak 文件的快捷方式

如图 12-11 所示，在“资源管理器”中，gak 文件的右键菜单与前两种情况不同。

图 12-11　以“文件夹”方式查看时，右键单击 gak 文件

单击右键菜单后，程序处理菜单输入，程序获得了点击的输入后，将相关信息通过对话框显示，如图 12-12 所示。

4．实例 12-9　属性页 Handler 实现，ProtShet.cpp

通过分析以上两个实例，可以了解 Shell 扩展 Handler 的基本编写方法，本实例实现了属性页的扩展和个性化。

实现属性页的扩展需再实现 IShellPropSheetExt 接口，Ishell Prop SheetExt 接口包括两个成员函数：AddPages 和 ReplacePage。

主调程序会根据注册表中的配置调用 AddPages 和 ReplacePage，分别实现了属性页的增加和替换。

在 ShellExt.dll 工程中，实现 IShellPropSheetExt 接口仍然是在 CShellExt 类中。代码位于 ProtShet.cpp，以下是 ProtShet.cpp 的全部代码。

图 12-12　选中菜单项 3 后弹出对话框

```
#include "priv.h"
#include "shellext.h"
#include "resource.h"

extern UINT g_cRefThisDll;          // DLL 的引用次数
extern HINSTANCE g_hmodThisDll;     // 本 DLL 句柄

UINT CALLBACK
GAKPageCallback(HWND hWnd,
             UINT uMessage,
             LPPROPSHEETPAGE  ppsp)
{
   switch(uMessage)
   {
      case PSPCB_CREATE:
         return TRUE;
      case PSPCB_RELEASE:
         if (ppsp->lParam)
         {
            ((LPCSHELLEXT)(ppsp->lParam))->Release();
         }
         return TRUE;
   }
   return TRUE;
}
//对话框消息处理函数
INT_PTR CALLBACK GAKPageDlgProc(HWND hDlg,
                        UINT uMessage,
                        WPARAM wParam,
                        LPARAM lParam)
{
   LPPROPSHEETPAGE psp=(LPPROPSHEETPAGE)GetWindowLongPtr(hDlg, DWLP_USER);
   UINT iIndex=0;
   int  iID;
   LPCSHELLEXT lpcs;
    char szTemp[4];
   switch (uMessage)
   {
      case WM_INITDIALOG:
         SetWindowLongPtr(hDlg, DWLP_USER, lParam);
         psp = (LPPROPSHEETPAGE)lParam;
         lpcs = (LPCSHELLEXT)psp->lParam;
```

```
            if (*(lpcs->m_szPropSheetFileUserClickedOn))
               iIndex = (UINT)GetPrivateProfileInt("IconImage",
                                                   "Index",
                                                   0,
                                                   lpcs->m_szPropSheetFileUserClickedOn);
            SendDlgItemMessage(hDlg, iIndex + IDC_RED, BM_SETCHECK, TRUE, 0L);
            break;
         case WM_DESTROY:
            RemoveProp(hDlg, "ID");
            break;
         case WM_COMMAND:
            switch (LOWORD(wParam))
            {
               case IDC_RED:
               case IDC_GREEN:
               case IDC_BLUE:
                  SetProp(hDlg, "ID", (HANDLE)lParam);
                  break;
               default:
                  break;
            }
            break;
         case WM_NOTIFY:
            switch (((NMHDR FAR *)lParam)->code)
            {
               case PSN_SETACTIVE:
                  break;
               case PSN_APPLY:
                  lpcs = (LPCSHELLEXT)psp->lParam;
                  iID  = GetDlgCtrlID((HWND)GetProp(hDlg, "ID"));
                  iID -= IDC_RED;
                  wsprintf(szTemp, "%i", iID);
                  WritePrivateProfileString("IconImage",
                                            "Index",
                                             szTemp,
                                             lpcs->m_szPropSheetFileUserClickedOn);
                  SHChangeNotify(SHCNE_ASSOCCHANGED, SHCNF_FLUSHNOWAIT, 0, 0);
                  break;
               default:
                  break;
            }
            break;
         default:
            return FALSE;
      }
      return TRUE;
   }
   //实现接口成员函数 AddPages
   STDMETHODIMP CShellExt::AddPages(LPFNADDPROPSHEETPAGE lpfnAddPage, LPARAM lParam)
   {
      PROPSHEETPAGE psp;
      HPROPSHEETPAGE hpage;
      ODS("CShellExt::AddPages()\r\n");
      FORMATETC fmte = {CF_HDROP,
                        (DVTARGETDEVICE FAR *)NULL,
                        DVASPECT_CONTENT,
                        -1,
                        TYMED_HGLOBAL
                       };
      STGMEDIUM medium;
      HRESULT hres = 0;
```

```
    if (m_pDataObj)  //判断全局变量 m_pDataObj 的状态
      hres = m_pDataObj->GetData(&fmte, &medium);
    if (SUCCEEDED(hres))
    {
       //Find out how many files the user has selected...
       UINT cbFiles = 0;
       LPCSHELLEXT lpcsext = this;
       if (medium.hGlobal)
          cbFiles = DragQueryFile((HDROP)medium.hGlobal, (UINT)-1, 0, 0);
       if (cbFiles < 2)
       {
          if (cbFiles)
            DragQueryFile((HDROP)medium.hGlobal,
                          0,
                          m_szPropSheetFileUserClickedOn,
                          sizeof(m_szPropSheetFileUserClickedOn));
          psp.dwSize      = sizeof(psp);
          psp.dwFlags     = PSP_USEREFPARENT | PSP_USETITLE | PSP_USECALLBACK;
          psp.hInstance   = g_hmodThisDll;
          psp.pszTemplate = MAKEINTRESOURCE(IDD_GAKPAGE);
          psp.hIcon       = 0;
          psp.pszTitle    = "GAK Color";
          psp.pfnDlgProc  = GAKPageDlgProc;
          psp.pcRefParent = &g_cRefThisDll;
          psp.pfnCallback = GAKPageCallback;
          psp.lParam      = (LPARAM)lpcsext;
          AddRef();
          hpage = CreatePropertySheetPage(&psp);
          if (hpage)
          {
             if (!lpfnAddPage(hpage, lParam))
             {
                DestroyPropertySheetPage(hpage);
                Release();
             }
          }
       }
    }
    return NOERROR;
}
//实现 ReplacePage 接口成员函数
STDMETHODIMP CShellExt::ReplacePage(UINT uPageID,
                                    LPFNADDPROPSHEETPAGE lpfnReplaceWith,
                                    LPARAM lParam)
{
    ODS("CShellExt::ReplacePage()\r\n");
    return E_FAIL;
}
```

分析代码，发现在 AddPages 函数中调用了 CreatePropertySheetPage 函数生成了属性页，函数的参数是指向 PROPSHEETPAGE 结构的指针。

属性页实质是一个对话框，在增加属性页时，分别指定了属性页的回调函数 GAKPageCallback 和消息处理函数 GAKPageDlgProc。实现了完整的对话框和消息处理过程（参见第 9 章有关窗口和对话框消息的讲解）。

属性页扩展的运行效果如图 12-13 所示。

如果选择"Red”那么图标颜色会变成红色。

5．实例 12-10　图标扩展 Handler 实现，lconHlr.cpp

在 12.2 节已经介绍如何为一种“文件类型”的文件配置图标，本实例将进一步说明如何根据文件内容配置图标。

使用过 Visual Studi0 2005 和 Visual Studi0 2008 的读者可能会发现，这两个版本的 VistualStudio 所生成的解决方案配置文件都是以.sln 为扩展名的，但是图标却不一样。一

个标有“8”一个标有“9”，如图 12-14 所示（安装 Visual Studi0 2008 的情况下）。

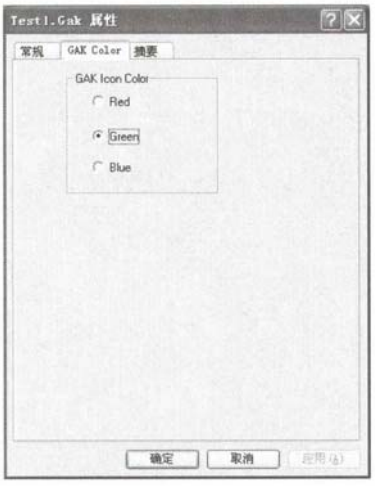

图 12-13　属性页扩展效果图

图 12-14　安装 Visual Studio 2008 后显示不同版本的.sln 文件图标

如图 12-15 所示，在 ShellExt.dll 工程中也实现了类似功能。

图 12-15　同一个扩展名的文件有不同的图标——根据文件内容判断

在.gak 文件中有相关的配置，图 12-16 和图 12-17 是 Test1.gak 的文件内容和 Test2.gak 的文件内容。

图 12-16　Test1.gak 文件内容

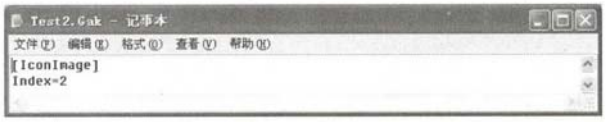

图 12-17　Test2.gak 文件内容

要在 Shell 扩展中实现图标的扩展，需要实现 IExtractIcon 接口，在 ShellExt.dll 工程中，仍然是在 CShellExt 类中实现的。IExtractIcon 需要实现 Extract 和 GetIconLocation 两个成员函数。在 GetIconLocation 函数中调用了 GetPrivateProfileInt 来获得文件中配置的内容。GetIconLocation 函数通
过 piIndex 参数和 szIconFile 参数返回，表示应该使用 szIconFile 所指明的文件中的 piIndex 号图标。

同时，本文件还是 IPersistFile 接口的实现。本实例中 IPersistFile 接口没有实际的功能。根据 Shell 扩展接口要求，IExtractIcon 的实例化应该通过 IPersistFile 接口来实现。本例中由于 IExtractIcon 由 CShellExt 实现，已经在之前经过了实例化。

```
#include "priv.h"
#include "shellext.h"

extern HINSTANCE g_hmodThisDll; // 本地 DLL 句柄
//IExtractIcon 接口成员函数 GetIconLocation 的实现
STDMETHODIMP CShellExt::GetIconLocation(UINT   uFlags,
                                        LPSTR  szIconFile,
                                        UINT   cchMax,
                                        int   *piIndex,
                                        UINT  *pwFlags)
{
    ODS("CShellExt::GetIconLocation()\r\n");
    GetModuleFileName(g_hmodThisDll, szIconFile, cchMax);
    *piIndex = (int)GetPrivateProfileInt("IconImage",
                                         "Index",
                                         0,
                                         m_szFileUserClickedOn);
    return S_OK;
}

// IExtractIcon 接口 Extract 成员实现
STDMETHODIMP CShellExt::Extract(LPCSTR pszFile,
                                UINT   nIconIndex,
                                HICON  *phiconLarge,
                                HICON  *phiconSmall,
                                UINT   nIconSize)
{
    ODS("CShellExt::Extract()\r\n");
    return S_FALSE;
}


// ********************** IPersistFile Implementation ******************
// IPersistFile 接口成员 IPersistFile Implementation 实现
STDMETHODIMP CShellExt::GetClassID(LPCLSID lpClassID)
{
    ODS("CShellExt::GetClassID()\r\n");
    return E_FAIL;
}
STDMETHODIMP CShellExt::IsDirty()
{
    ODS("CShellExt::IsDirty()\r\n");
    return S_FALSE;
}

// IPersistFile 接口成员 Load 实现
STDMETHODIMP CShellExt::Load(LPCOLESTR lpszFileName, DWORD grfMode)
{
    ODS("CShellExt::Load()\r\n");
    WideCharToMultiByte(CP_ACP,             // CodePage
                        0,                  // dwFlags
                        lpszFileName,       // lpWideCharStr
```

```
                    -1,                 // cchWideChar
                    m_szFileUserClickedOn, // lpMultiByteStr
                    sizeof(m_szFileUserClickedOn),  // cchMultiByte,
                    NULL,               // lpDefaultChar,
                    NULL);              // lpUsedDefaultChar
    return NOERROR;
}
//Save 成员实现
STDMETHODIMP CShellExt::Save(LPCOLESTR lpszFileName, BOOL fRemember)
{
    ODS("CShellExt::Save()\r\n");
    return E_FAIL;
}
//SaveCompleted 成员实现
STDMETHODIMP CShellExt::SaveCompleted(LPCOLESTR lpszFileName)
{
    ODS("CShellExt::SaveCompleted()\r\n");
    return E_FAIL;
}

STDMETHODIMP CShellExt::GetCurFile(LPOLESTR FAR* lplpszFileName)
{
    ODS("CShellExt::GetCurFile()\r\n");
    return E_FAIL;
}
```

6．实例 0 12-11　Copy Handler 实现，lconHlr.cpp

如果在注册表中注册了 CopyHookHandlers，那么在用户操作与文件类型相关的文件或文件夹时，ICopyHook 接口 CopyCallback 成员函数将被调用。

```
#include "priv.h"
#include "shellext.h"
STDMETHODIMP_(UINT) CShellExt::CopyCallback(HWND hwnd,
                                    UINT wFunc,
                                    UINT wFlags,
                                    LPCSTR pszSrcFile,
                                    DWORD dwSrcAttribs,
                                    LPCSTR pszDestFile,
                                    DWORD dwDestAttribs)
{
    ODS("CShellExt::CopyCallback\r\n");
    MessageBox(NULL, "CopyCallback!", "SHELLEXT", MB_OK);
    return IDYES;
}
```

本接口在被调用时弹出对话框。当用户新建文件夹等操作时会调用接口，弹出如图 12-18 所示的对话框。

图 12-18　Copy Handler 弹出的对话框

12.3.6　Shell扩展程序的调试

一般情况下，COM 不是应用程序的主模块。需要有一个主模块来作为主调用程序。因此在调试时需要经过以下步骤。

◇使用调试选项编译和链接 Shell 扩展程序（参见第 3 章）。

◇将 Shell 扩展程序加载入调试器，注意符号文件和源代码的位置与编译时不要改变。

◇注册 Shell 扩展程序。

◇停止 Shell(Exlplorer.exe)的运行方法是：开始一关机，关机选项对话框弹出后，

按住 CTRL+ALT+SHIFT，单击对话框上的取消按钮，Shell 便退出运行。

◇在调试器中设置启动模块，如果使用 Visual Studio 调试，那么在“工程属性一属性配置一调试器一命令”中输入 explorer.exe。

◇在调试器中运行项目、设置断点、跟踪调试。

12.3.7 总结

为了进一步分析程序的运行过程，在 ShellExt.dll Shell 扩展成功安装并运行后，使用调试器附加到 explorer.exe 进程（Windbg 中菜单“File- Attach to Process”），来查看代码 OutputDebugString 函数输出的字符串。

如图 12-19 所示是右键单击.gak 文件时调试器跟踪到的打印输出语句。

读者可以结合代码分析 Shell 扩展的运行过程。其他的几个 Handler 可以采用同样的方法来分析，就可以完全理解整个 Shell 扩展的原理。

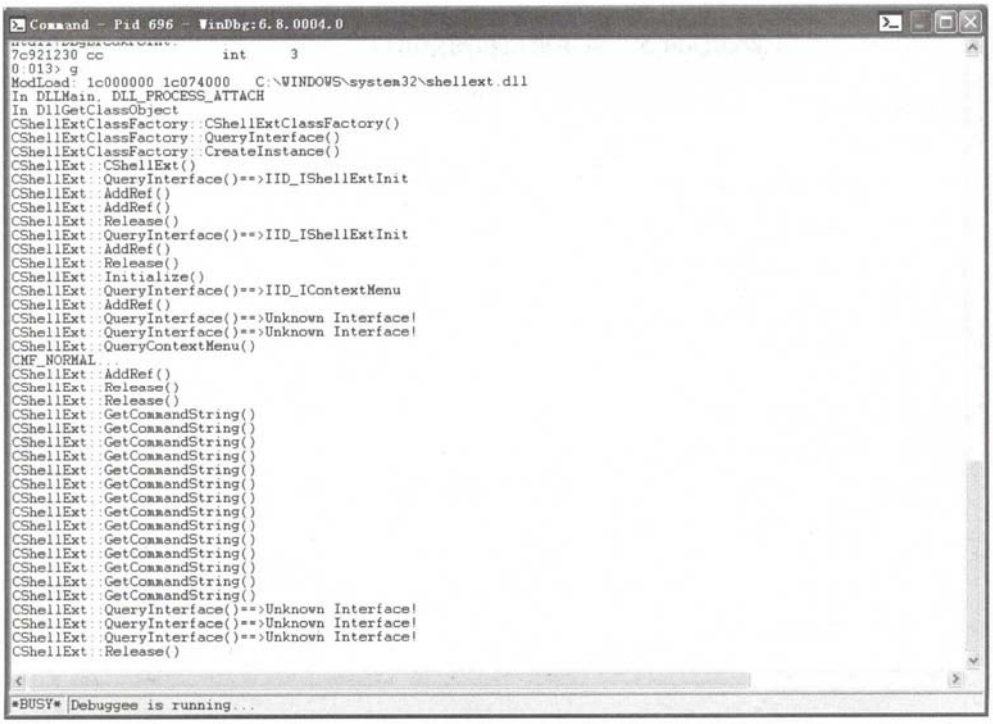

图 12-19 调用 Shell 扩展程序——跟踪调试打印输出

12.4 任务栏通知区域(Tray)图标

很多应用程序都在系统 Shell 任务栏通知区域有图标提示。通知区域图标适用于长时间运行，但无需用户过多操作，并在有事件发生时可随时根据情况通知用户的应用程序（比如一些下载工具、即时通信工具等），如图 12-20 所示。

图 12-20　很多应用程序使用了 Tray 图标

使用任务通知区域图标的好处如下。

◇窗口程序可以在后台运行，窗口退出，但是程序仍然在运行，减少窗口数量，便于用户操作。

◇窗口最小化后，不占用任务栏中的位置，使用户在操作主机时将精力集中在需在观注的窗口。

◇有新的事件发生时（比如有新的即时消息、文件下载完成等），可以即时通知到用户。

◇用户通过单击、双击图标、图标菜单等可以很方便地配置应用程序，控制应用程序的运行。

12.4 节的所有实例都包括在 TaskbarDemo 工程中。本节将围绕这个实例展开讲解，包括以下内容。

◇创建图标窗口。

◇最小化主窗口到通知区域。

◇图标菜单。

◇弹出气泡通知。

12.4.1　创建图标窗口

实例 12-12 演示了如何创建与通知区域图标相关的主窗口。

1．实例 12-12 TaskbarDemo 工程主窗口和图标窗口的创建

使用了通知区域图标的程序应该是图形用户界面的程序，具有窗口。和一般的窗口程序一样，应该创建一个主窗口，并处理窗口消息。

在创建主窗口的同时，还调用了 CSystemTray 类的 Create 成员函数，创建了通知区域图标。Create 函数的具体实现见实例 12-13。

下面是创建主窗口的程序代码。

```
/* ************************************
* TaskbarDemo.cpp 通知区域图标
************************************/
/*  预编译声明  */
#include "stdafx.h"
#include "resource.h"
#include "SystemTraySDK.h"

#define  MAX_LOADSTRING 100
#define  WM_ICON_NOTIFY WM_APP+10

/*  全局变量  */
HINSTANCE hInst;
TCHAR szTitle[MAX_LOADSTRING];
TCHAR szWindowClass[MAX_LOADSTRING];
CSystemTray TrayIcon;

/*  函数声明  */
ATOM              MyRegisterClass(HINSTANCE hInstance);
BOOL              InitInstance(HINSTANCE, int);
LRESULT CALLBACK  WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK  About(HWND, UINT, WPARAM, LPARAM);

/* ************************************
* WinMain
```

```
* 入口函数
*************************************/
int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
     MSG msg;
     HACCEL hAccelTable;
     // 字符串
     LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
     LoadString(hInstance, IDC_TASKBARDEMO, szWindowClass, MAX_LOADSTRING);
     MyRegisterClass(hInstance);
     // 初始化
     if (!InitInstance (hInstance, nCmdShow))
     {
          return FALSE;
     }
     hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_TASKBARDEMO);
     // 消息循环
     while (GetMessage(&msg, NULL, 0, 0))
     {
          if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
          {
               TranslateMessage(&msg);
               DispatchMessage(&msg);
          }
     }
     return msg.wParam;
}

/* ************************************
* ATOM MyRegisterClass(HINSTANCE hInstance)
* 功能:     注册窗口类
*************************************/
ATOM MyRegisterClass(HINSTANCE hInstance)
{
     WNDCLASSEX wcex;
     wcex.cbSize = sizeof(WNDCLASSEX);
     wcex.style            = CS_HREDRAW | CS_VREDRAW;
     wcex.lpfnWndProc  = (WNDPROC)WndProc;
     wcex.cbClsExtra       = 0;
     wcex.cbWndExtra       = 0;
     wcex.hInstance        = hInstance;
     wcex.hIcon            = LoadIcon(hInstance, (LPCTSTR)IDI_TASKBARDEMO);
     wcex.hCursor      = LoadCursor(NULL, IDC_ARROW);
     wcex.hbrBackground    = (HBRUSH)(COLOR_WINDOW+1);
     wcex.lpszMenuName = (LPCSTR)IDC_TASKBARDEMO;
     wcex.lpszClassName    = szWindowClass;
     wcex.hIconSm      = LoadIcon(wcex.hInstance, (LPCTSTR)IDI_TASKBARDEMO);

     return RegisterClassEx(&wcex);
}
/* ************************************
* BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
* 功能:   创建窗口和图标
*************************************/
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
     HWND hWnd;
     hInst = hInstance;
     hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
          CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
     if (!hWnd)
```

```
        {
            return FALSE;
        }
        // 创建通知区域图标
        if (!TrayIcon.Create(hInstance,
            hWnd,                           // 父窗口
            WM_ICON_NOTIFY,                 // 通知
            _T("This is a Tray Icon - Right click on me!"),    // tooltip
            ::LoadIcon(hInstance, (LPCTSTR)IDI_TASKBARDEMO),
            IDR_POPUP_MENU))
            return FALSE;
        ShowWindow(hWnd, nCmdShow);
        UpdateWindow(hWnd);
        return TRUE;
    }

    /* ************************************
    * WndProc
    * 功能：主窗口消息函数
    ************************************/
    LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
    {
        int wmId, wmEvent;
        PAINTSTRUCT ps;
        HDC hdc;
        TCHAR szHello[MAX_LOADSTRING];
        LoadString(hInst, IDS_HELLO, szHello, MAX_LOADSTRING);
        switch (message)
        {
        case WM_ICON_NOTIFY:
            return TrayIcon.OnTrayNotification(wParam, lParam);
        case WM_COMMAND:
            wmId    = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            // 菜单
            switch (wmId)
            {
                // 图标右键菜单中选择 Action
            case IDM_ANIMATE:
                // 动画
                TrayIcon.SetIconList(IDI_ICON1, IDI_ICON4);
                TrayIcon.Animate(50, 2);
                // 显示气泡通知
                TrayIcon.ShowBalloon("THIS IS A BALLOON",":)",0,10);
                // 从通知区域恢复原始大小
                TrayIcon.MaximiseFromTray(hWnd);
                break;
                // 退出
            case IDM_EXIT:
                DestroyWindow(hWnd);
                break;
                // 隐藏，最小化到通知区域
            case IDM_HIDE:
                TrayIcon.MinimiseToTray(hWnd);
            default:
                return DefWindowProc(hWnd, message, wParam, lParam);
            }
            break;
        case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            RECT rt;
            GetClientRect(hWnd, &rt);
            DrawText(hdc, szHello, strlen(szHello), &rt, DT_VCENTER | DT_CENTER | DT_SINGLELINE
|DT_WORDBREAK);
```

```
            EndPaint(hWnd, &ps);
            break;
            // 退出
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
    }
```

分析代码，如果用户在图标菜单中选择了“IDM_ANIMIATE”，那么会调用 CSystemTray 类的 SetIconList 成员和 Animate 成员将图标变化为动画。然后调用了 ShowBalloon 成员函数，弹出一个气泡。最后调用 MaximiseFromTray 将主窗口从通知区域恢复至原始大小。

如果用户大菜单中选择了“IDM_ HIDE”，那么将调用 MinimiseToTray 函数，将主窗口最小化到通知区域。

这几个函数的实现参见后续几个实例。

2．运行结果

如图 12-21 所示，程序运行后，会弹出一个简单窗口，并在右下角通知区域中出现“笑脸”图标。

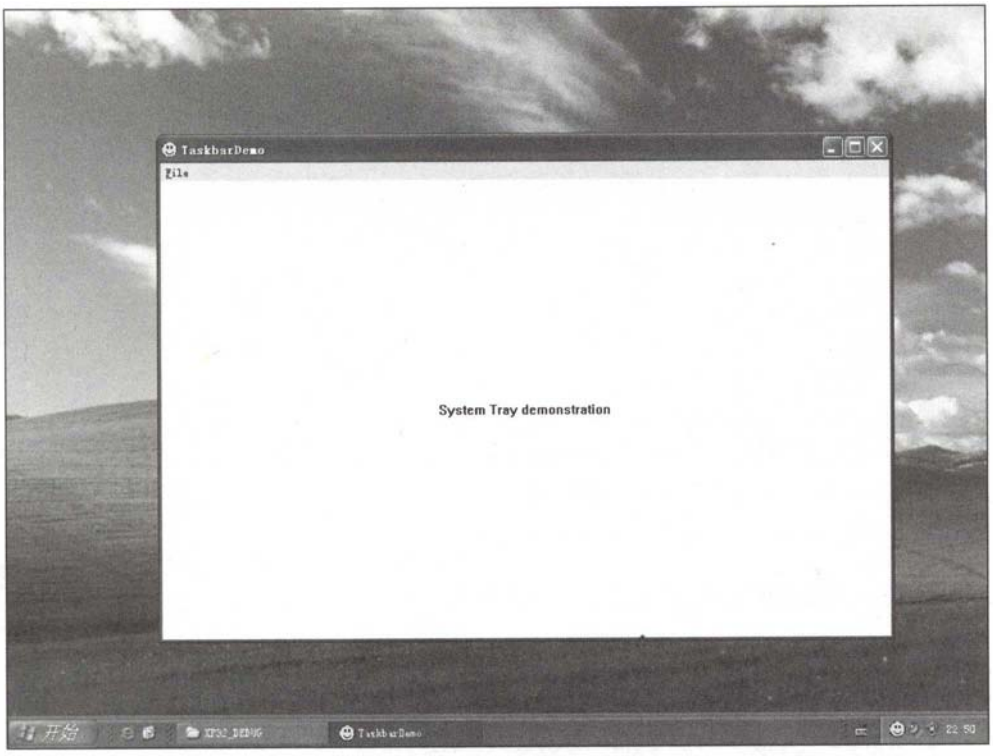

图 12-21　窗口程序与通知图标

12.4.2　创建图标和图标菜单

创建图标和菜单使用了 CSystemTray 类的 Create 成员函数。

分析 Create 函数，在图标的创建过程中有两个关键的步骤:，一是创建了一个窗口，并保留了句柄，这个窗口在后面图标动画的定时器时有关键作用；二是填充了 NOTIFYICONDATA 结构，然后作为参数调用了 Shell_NotifyIcon 函数。

Shell_NotifyIcon 函数在整个通知区域图标操作过程中具有重要作用。

给通知区域添加图标使用 Shell_NotifyIcon 函数，删除图标也是使用 Shell_NotifyIcon，修改
图标的状态和配置属性也是使用 Shell_NotifyIcon 函数。只不过调用时设置的参数不同。

如下是 ShellNotifyIcon 函数原型：

```
BOOL Shell_NotifyIcon(
   DWORD dwMessage,
   PNOTIFYICONDATA lpdata
);
```

增加图标、删除图标、修改图标、为图标设置焦点这 4 个操作都调用此函数，调用时需将 dwMessage 参数分别设置为 NIM_ ADD、NIM_DELETE、NIM_MODIFY、NIM_SETFOCUS。第二个参数是指向 NOTIFYICONDATA 结构的指针。

NOTIFYICONDATA 结构中包括了需要配置的相关信息，结构定义如下：

```
typedef struct _NOTIFYICONDATA {
     DWORD cbSize;
     HWND hWnd;
     UINT uID;
     UINT uFlags;
     UINT uCallbackMessage;
     HICON hIcon;
     TCHAR szTip[64];
     DWORD dwState;
     DWORD dwStateMask;
     TCHAR szInfo[256];
     union {
         UINT uTimeout;
         UINT uVersion;
     };
     TCHAR szInfoTitle[64];
     DWORD dwInfoFlags;
     GUID guidItem;
     HICON hBalloonIcon;
} NOTIFYICONDATA, *PNOTIFYICONDATA;
```

hWnd 是主窗口的句柄，对图标的所有操作都会发送消息给主窗口。

uCallbaclMessage 是向主窗口发送的消息。

其他各个字段都有各自的意义，参考实例代码。

实例 12-13 TaskbarDemo 通知区域图标的创建

Create 函数的参数说明如下。

HINSTANCE hInst:应用程序实例，从 WinMain 函数参数得到。

HWND hParent:父窗口句柄，在这里设置为创建的主窗口。

UINT uCallbaclMessage:当在图标上有操作时会发送些消息给父窗口，这里设置为 WM_ICON__ NOTIFY，在主窗口的消息处理函数 WndProc 中处理了此消息。

LPCTSTRszToolTip: ToolTip 信息（当鼠标光标移动到图标上时显示的提示）。

HICON icon:显示的图标 ICON 句柄。

UINT uID:标识符。

```
/* ************************************
* BOOL CSystemTray::Create(...)
* 功能：创建通知图标窗口
************************************/
BOOL CSystemTray::Create(HINSTANCE hInst, HWND hParent, UINT uCallbackMessage,
                         LPCTSTR szToolTip, HICON icon, UINT uID,
                         BOOL bHidden /*=FALSE*/,
                         LPCTSTR szBalloonTip,
                         LPCTSTR szBalloonTitle ,
                         DWORD dwBalloonIcon,
                         UINT uBalloonTimeout)
{
    // 判断系统版本 >= Windows 95
    m_bEnabled = (GetVersion() & 0xff) >= 4;
    if (!m_bEnabled)
    {
        ASSERT(FALSE);
        return FALSE;
    }
    m_nMaxTooltipLength = _countof(m_tnd.szTip);
    // 判断消息
    ASSERT(uCallbackMessage >= WM_APP);
    // 防止溢出
    ASSERT(_tcslen(szToolTip) <= m_nMaxTooltipLength);
    m_hInstance = hInst;
    RegisterClass(hInst);
    // 创建窗口
    m_hWnd = ::CreateWindow(TRAYICON_CLASS, _T(""), WS_POPUP,
        CW_USEDEFAULT,CW_USEDEFAULT,
        CW_USEDEFAULT,CW_USEDEFAULT,
        NULL, 0,
        hInst, 0);
    // 填充 NOTIFYICONDATA 结构
    m_tnd.cbSize = sizeof(NOTIFYICONDATA);
    m_tnd.hWnd   = (hParent)? hParent : m_hWnd;
    m_tnd.uID    = uID;
    m_tnd.hIcon  = icon;
    m_tnd.uFlags = NIF_MESSAGE | NIF_ICON | NIF_TIP;
    m_tnd.uCallbackMessage = uCallbackMessage;
    strncpy(m_tnd.szTip, szToolTip, m_nMaxTooltipLength);
    // 判断
    if (m_bWin2K && szBalloonTip)
    {
        ASSERT(lstrlen(szBalloonTip) < 256);
        if (szBalloonTitle)
        {
            ASSERT(lstrlen(szBalloonTitle) < 64);
        }
        ASSERT(NIIF_NONE == dwBalloonIcon    || NIIF_INFO == dwBalloonIcon ||
            NIIF_WARNING == dwBalloonIcon || NIIF_ERROR == dwBalloonIcon);
        ASSERT(uBalloonTimeout >= 10 && uBalloonTimeout <= 30);
        m_tnd.uFlags |= NIF_INFO;                    // 气泡
```

BOOL bHidden:是否隐藏，设置为 FALSE。

LPCTSTR szBalloonTip：创建时显示的气泡提示信息。

LPCTSTR szBalloonTitle:气泡的标题。

DWORD dwBalloonIcon:气泡的图标。

UINT uBalloonTimeout:气泡的时延。

代码如下所示：

```
            _tcsncpy(m_tnd.szInfo, szBalloonTip, 255);
            if (szBalloonTitle)
                _tcsncpy(m_tnd.szInfoTitle, szBalloonTitle, 63);
            else
                m_tnd.szInfoTitle[0] = _T('\0');
            m_tnd.uTimeout    = uBalloonTimeout * 1000; // ms
            m_tnd.dwInfoFlags = dwBalloonIcon;
        }
        m_bHidden = bHidden;
        m_hTargetWnd = m_tnd.hWnd;
        if (m_bWin2K && m_bHidden)
        {
            m_tnd.uFlags = NIF_STATE;
            m_tnd.dwState = NIS_HIDDEN;
            m_tnd.dwStateMask = NIS_HIDDEN;
        }
        m_uCreationFlags = m_tnd.uFlags;
        BOOL bResult = TRUE;
        if (!m_bHidden || m_bWin2K)
        {
            // 以 NIM_ADD 为参数调用 Shell_NotifyIcon，增加通知区域图标
            bResult = Shell_NotifyIcon(NIM_ADD, &m_tnd);
            m_bShowIconPending = m_bHidden = m_bRemoved = !bResult;
        }
        if (m_bWin2K && szBalloonTip)
        {
            m_tnd.szInfo[0] = _T('\0');
        }
        return bResult;
}
```

12.4.3 最小化主窗口到通知区域

在创建完成通知区域图标后，下面介绍如何将主窗口最小化到通知区域。

1．实例 12-14 T 最小化主窗口到通知区域

在实例 12-2 中已经介绍了，在界面中通过菜单选择“Hide”时，会调用 MinimiseToTray 函数。

```
            case IDM_HIDE:
                TrayIcon.MinimiseToTray(hWnd);
```

CSystemTray 类、MinimiseToTray 函数实现如下：

```
/* ************************************
*    将窗口最小化到通知区域
**************************************/
void CSystemTray::MinimiseToTray(HWND hWnd)
{

    if (GetDoWndAnimation())
    {
        RECT rectFrom, rectTo;
        GetWindowRect(hWnd, &rectFrom);
        GetTrayWndRect(&rectTo);
        DrawAnimatedRects(hWnd, IDANI_CAPTION, &rectFrom, &rectTo);
    }
    RemoveTaskbarIcon(hWnd);
    SetWindowLong(hWnd, GWL_STYLE, GetWindowLong(hWnd, GWL_STYLE) &~ WS_VISIBLE);
}
```

在程序中，首先获取了窗口当前的 RECT，然后获得了 Tray 的 RECT，再调用 DrawAnimated RectsAPI 函数，将窗口进行缩放和移动。然后调用 RemoveTaskbarIcon 函数将窗口消除。最后调用 SetWindowLong 设置了窗口的属性。实现了窗口的隐藏。

在 MinimiseToTray 函数中调用了 GetTrayWndRect 函数，GetTrayWndRect 函数功能是

获取 Tray 图标的 RECT。函数的实现如下：

```
/* ************************************
*     获得 Tray 的 RECT
************************************/
void CSystemTray::GetTrayWndRect(LPRECT lprect)
{
#define DEFAULT_RECT_WIDTH 150
#define DEFAULT_RECT_HEIGHT 30

     HWND hShellTrayWnd = FindWindow(_T("Shell_TrayWnd"), NULL);
     if (hShellTrayWnd)
     {
          GetWindowRect(hShellTrayWnd, lprect);
          EnumChildWindows(hShellTrayWnd, FindTrayWnd, (LPARAM)lprect);
          return;
     }
     APPBARDATA appBarData;
     appBarData.cbSize=sizeof(appBarData);
     if (SHAppBarMessage(ABM_GETTASKBARPOS,&appBarData))
     {
          switch(appBarData.uEdge)
          {
          case ABE_LEFT:
          case ABE_RIGHT:
               lprect->top    = appBarData.rc.bottom-100;
               lprect->bottom = appBarData.rc.bottom-16;
               lprect->left   = appBarData.rc.left;
               lprect->right  = appBarData.rc.right;
               break;
          case ABE_TOP:
          case ABE_BOTTOM:
               lprect->top    = appBarData.rc.top;
               lprect->bottom = appBarData.rc.bottom;
               lprect->left   = appBarData.rc.right-100;
               lprect->right  = appBarData.rc.right-16;
               break;
          }
          return;
     }
     if (hShellTrayWnd)
     {
          ::GetWindowRect(hShellTrayWnd, lprect);
          if (lprect->right - lprect->left > DEFAULT_RECT_WIDTH)
               lprect->left = lprect->right - DEFAULT_RECT_WIDTH;
          if (lprect->bottom - lprect->top > DEFAULT_RECT_HEIGHT)
               lprect->top = lprect->bottom - DEFAULT_RECT_HEIGHT;
          return;
     }
     SystemParametersInfo(SPI_GETWORKAREA,0,lprect, 0);
     lprect->left = lprect->right - DEFAULT_RECT_WIDTH;
     lprect->top  = lprect->bottom - DEFAULT_RECT_HEIGHT;
}
```

GetTrayWndRect 函数首先调用了 FindWindow 函数，如果直接能找到 Tray 图标的窗口，就直接获取图标的 RECT。如果找不到，需要调用 ShellAPI SHAppBarMessage 获取 Windows taskbar 的大小，再通过 SystemParametersInfo，获取在主显示器中的工作区大小，然后通过运算得到 Tray 区域的大小。

2．运行结果

在窗口菜单中，选择“Hide”，窗口最小化到了通知区，窗口消失，但是通知区域的图标仍然存在，如图 12-22 所示。

图 12-22　隐藏窗口，只保留通知图标

12.4.4　弹出气泡通知

CSystemTray 的 ShowBalloon 成员函数实现了从图标上弹出气泡通知。szText 参数是通知的内容。szTitle 通知是气泡的标题，还通过 dwIcon 设置图标，通过 uTimeout 设置通知滞留的时间。

1．实例 12-15 气泡

ShowBalloon 函数的实现代码如下：

```
/* ************************************
* CSystemTray::ShowBalloon
* 功能：显示气泡
************************************/
BOOL CSystemTray::ShowBalloon(LPCTSTR szText,
                              LPCTSTR szTitle,
                              DWORD   dwIcon          /*=NIIF_NONE*/,
                              UINT    uTimeout        /*=10*/ )
{
    // Win2k 以上
    if (!m_bWin2K)
        return FALSE;
```

```
    // 防止溢出
    ASSERT(lstrlen(szText) < 256);
    // 防止 Title 溢出
    if (szTitle)
    {
        ASSERT(lstrlen(szTitle) < 64);
    }
    // 判断 Flag
    ASSERT(NIIF_NONE == dwIcon    || NIIF_INFO == dwIcon ||
        NIIF_WARNING == dwIcon || NIIF_ERROR == dwIcon);
    // 判断 Timeout
    ASSERT(uTimeout >= 10 && uTimeout <= 30);
    // 设置 NOTIFYICONDATA 结构
    m_tnd.uFlags = NIF_INFO;              // 显示气泡信息
    _tcsncpy(m_tnd.szInfo, szText, 256);
    if (szTitle)
        _tcsncpy(m_tnd.szInfoTitle, szTitle, 64);
    else
        m_tnd.szInfoTitle[0] = _T('\0');
    m_tnd.dwInfoFlags = dwIcon;
    m_tnd.uTimeout = uTimeout * 1000;    解情况            // ms
    // 以 NIM_MODIFY 为参数，调用 Shell_NotifyIcon
    BOOL bSuccess = Shell_NotifyIcon (NIM_MODIFY, &m_tnd);

    // 清零
    m_tnd.szInfo[0] = _T('\0');
    return bSuccess;
}
```

分析代码可以知道，如果需要从图标弹出一个气泡通知，只需要以修改图标的方式调用 Shell_NotifyIcon 函数。在设置 NOTIFYICONDATA 结构时，使用 NIF_INFO 标记。所有图标、消息内容、标题、滞留时间都可以通过 NOTIFYICONDATA 结构设置。

2.运行结果

如下图 12-23 所示是弹出的菜单。

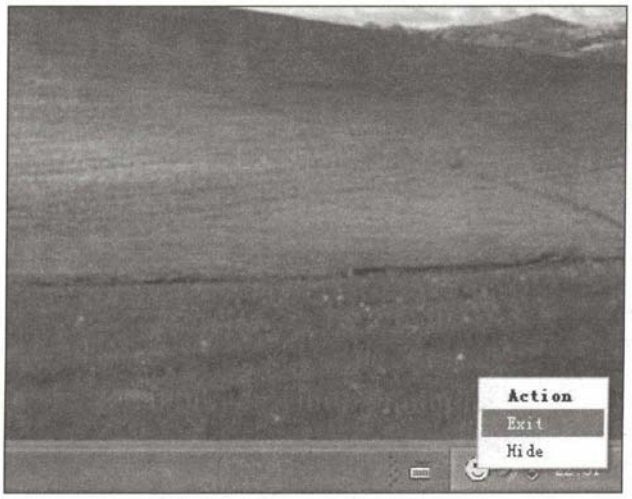

图 12-23　图形菜单

如图 12-24 所示是选择“Action”菜单输入后，程序的运行结果——弹出窗口、弹出气泡。

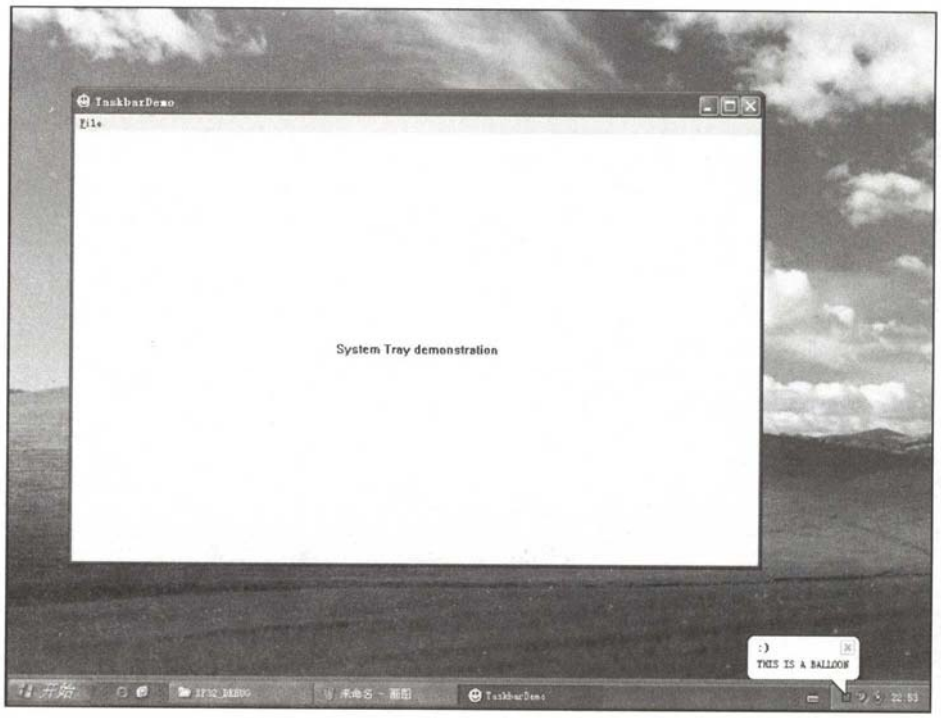

图 12-24　处理点击输入

12.4.5　动态图标

读者可以从实例 12-15 的运行结果中可以发现，在选择“Action”菜单项后，图标由原来的“笑脸”变成了方框，而且这个方框是转动的。Animate 函数设置了一个计时器，在图标窗口的消息处理函数 CSystemTray::WindowProc 中处理了时间器消息。并循环调用了 StepAnimation 和 StopAnimation 成员函数。而 StepAnimation 函数和 StopAnimation 函数都调用了 SetIcon 成员函数。

SetIcon 函数调用 Shell_NotifyIcon，修改图标属性。

```
BOOL CSystemTray::SetIcon(HICON hIcon)
{
     if (!m_bEnabled)
          return FALSE;
     m_tnd.uFlags = NIF_ICON;
     m_tnd.hIcon = hIcon;
     if (m_bHidden)
          return TRUE;
     else
          return Shell_NotifyIcon(NIM_MODIFY, &m_tnd);
}

/* ************************************
*BOOL CSystemTray::StepAnimation()
* 功能：使图标动作
************************************/
BOOL CSystemTray::StepAnimation()
{
     if (!m_IconList.size())
          return FALSE;
     m_nCurrentIcon++;
```

```
    if (m_nCurrentIcon >= m_IconList.size())
        m_nCurrentIcon = 0;
    return SetIcon(m_IconList[m_nCurrentIcon]);
}
/* ************************************
*BOOL CSystemTray::StopAnimation()
* 功能:      停止图标动作
************************************/
BOOL CSystemTray::StopAnimation()
{
    BOOL bResult = FALSE;
    if (m_uIDTimer)
        bResult = ::KillTimer(m_hWnd, m_uIDTimer);
    m_uIDTimer = 0;
    if (m_hSavedIcon)
        SetIcon(m_hSavedIcon);
    m_hSavedIcon = NULL;
    return bResult;
}

/* ************************************
*BOOL CSystemTray::Animate(UINT nDelayMilliSeconds, int nNumSeconds )
* 功能:      使图标动作
************************************/
BOOL CSystemTray::Animate(UINT nDelayMilliSeconds, int nNumSeconds /*=-1*/)
{
    if (m_IconList.empty())
        return FALSE;
    StopAnimation();
    m_nCurrentIcon = 0;
    time(&m_StartTime);
    m_nAnimationPeriod = nNumSeconds;
    m_hSavedIcon = GetIcon();
    // 计时器，注意实例中对计时器消息的处理
    m_uIDTimer = ::SetTimer(m_hWnd, m_nTimerID, nDelayMilliSeconds, NULL);
    return (m_uIDTimer != 0);
}

/* ************************************
* 功能: Tray 图标窗口消息处理函数
************************************/
LRESULT PASCAL CSystemTray::WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{

    CSystemTray* pTrayIcon = m_pThis;
    if (pTrayIcon->GetSafeHwnd() != hWnd)
        return ::DefWindowProc(hWnd, message, wParam, lParam);
    // Tray 被创建，此消息必须处理
    if (message == CSystemTray::m_nTaskbarCreatedMsg)
        return pTrayIcon->OnTaskbarCreated(wParam, lParam);
    // 图标动画计时器
    if (message == WM_TIMER && wParam == pTrayIcon->GetTimerID())
        return pTrayIcon->OnTimer(wParam);
    // 设置改变
    if (message == WM_SETTINGCHANGE && wParam == pTrayIcon->GetTimerID())
        return pTrayIcon->OnSettingChange(wParam, (LPCTSTR) lParam);
    // 是否是需要处理的消息
    if (message == pTrayIcon->GetCallbackMessage())
        return pTrayIcon->OnTrayNotification(wParam, lParam);
    return ::DefWindowProc(hWnd, message, wParam, lParam);
}
```

```
/* ************************************
* 功能：处理计时器，图标动画
**************************************/
LRESULT CSystemTray::OnTimer(UINT nIDEvent)
{
     if (nIDEvent != m_uIDTimer)
     {
          ASSERT(FALSE);
          return 0L;
     }
     time_t CurrentTime;
     time(&CurrentTime);
     time_t period = CurrentTime - m_StartTime;
     if (m_nAnimationPeriod > 0 && m_nAnimationPeriod < period)
     {
          StopAnimation();
          return 0L;
     }
     StepAnimation();
     return 0L;
}
```

12.4.6 其他功能

以上介绍了任务通知区域图标的主要功能，只包括其中小部分代码，实现上还具有很多功能。

读者可以自行分析实例代码。

以下是 CSystemTray 类的所有成员，其中还有很多成员函数的功能没有介绍。

```
class CSystemTray
{
public:
    CSystemTray();
    CSystemTray(HINSTANCE hInst, HWND hParent, UINT uCallbackMessage,
            LPCTSTR szTip, HICON icon, UINT uID,
            BOOL bhidden = FALSE,
            LPCTSTR szBalloonTip = NULL, LPCTSTR szBalloonTitle = NULL,
            DWORD dwBalloonIcon = NIIF_NONE, UINT uBalloonTimeout = 10);
    virtual ~CSystemTray();

// 操作函数
public:
    BOOL Enabled() { return m_bEnabled; }
    BOOL Visible() { return !m_bHidden; }

    // 创建托盘图标
    BOOL Create(HINSTANCE hInst, HWND hParent, UINT uCallbackMessage, LPCTSTR szTip,
              HICON icon, UINT uID, BOOL bHidden = FALSE,
          LPCTSTR szBalloonTip = NULL, LPCTSTR szBalloonTitle = NULL,
          DWORD dwBalloonIcon = NIIF_NONE, UINT uBalloonTimeout = 10);

    // 更改帮助文档
    BOOL   SetTooltipText(LPCTSTR pszTooltipText);
    BOOL   SetTooltipText(UINT nID);
    LPTSTR GetTooltipText() const;

    // Change or retrieve the icon displayed
    BOOL  SetIcon(HICON hIcon);
    BOOL  SetIcon(LPCTSTR lpszIconName);
    BOOL  SetIcon(UINT nIDResource);
```

```
    BOOL  SetStandardIcon(LPCTSTR lpIconName);
    BOOL  SetStandardIcon(UINT nIDResource);
    HICON GetIcon() const;

    void  SetFocus();
    BOOL  HideIcon();
    BOOL  ShowIcon();
    BOOL  AddIcon();
    BOOL  RemoveIcon();
    BOOL  MoveToRight();

    BOOL ShowBalloon(LPCTSTR szText, LPCTSTR szTitle = NULL,
                DWORD dwIcon = NIIF_NONE, UINT uTimeout = 10);

    // 图标动画
    BOOL  SetIconList(UINT uFirstIconID, UINT uLastIconID);
    BOOL  SetIconList(HICON* pHIconList, UINT nNumIcons);
    BOOL  Animate(UINT nDelayMilliSeconds, int nNumSeconds = -1);
    BOOL  StepAnimation();
    BOOL  StopAnimation();

    // 更改目录默认项目
    void  GetMenuDefaultItem(UINT& uItem, BOOL& bByPos);
    BOOL  SetMenuDefaultItem(UINT uItem, BOOL bByPos);

    // 更改或找回窗口以将图标通知消息发送到
    BOOL  SetNotificationWnd(HWND hNotifyWnd);
    HWND  GetNotificationWnd() const;

    // 更改或找回窗口以将菜单命令发送到
    BOOL  SetTargetWnd(HWND hTargetWnd);
    HWND  GetTargetWnd() const;

    // 更改或找回发送至窗口的通知消息
    BOOL  SetCallbackMessage(UINT uCallbackMessage);
    UINT  GetCallbackMessage() const;

    HWND  GetSafeHwnd() const  { return (this)? m_hWnd : NULL; }
    UINT_PTR GetTimerID() const   { return m_nTimerID; }

     // 状态函数
public:
    static void MinimiseToTray(HWND hWnd);
    static void MaximiseFromTray(HWND hWnd);

public:
    // Default handler for tray notification message
    virtual LRESULT OnTrayNotification(WPARAM uID, LPARAM lEvent);

// 静态回调函数和数据
public:
    static LRESULT PASCAL WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);
    static CSystemTray* m_pThis;

// 实例
protected:
    void Initialise();
    void InstallIconPending();
    ATOM RegisterClass(HINSTANCE hInstance);

    virtual void CustomizeMenu(HMENU) {}
```

```
// 实例
protected:
    NOTIFYICONDATA  m_tnd;
    HINSTANCE       m_hInstance;
    HWND            m_hWnd;
    HWND            m_hTargetWnd;              // 菜单命令被发送到的窗口

    BOOL            m_bEnabled;                // does O/S support tray icon?
    BOOL            m_bHidden;                 // 图标是否被隐藏?
    BOOL            m_bRemoved;                // 图标是否被移除?
    BOOL            m_bShowIconPending;        // 一旦任务栏被创建就显示图标
    BOOL            m_bWin2K;                  //使用新的 Win2k 功能?

    ICONVECTOR      m_IconList;
    UINT_PTR        m_uIDTimer;
    int             m_nCurrentIcon;
    time_t          m_StartTime;
    int             m_nAnimationPeriod;
    HICON           m_hSavedIcon;
    UINT            m_DefaultMenuItemID;
    BOOL            m_DefaultMenuItemByPos;
    UINT            m_uCreationFlags;

// 静态数据
protected:
    static BOOL RemoveTaskbarIcon(HWND hWnd);

    static const UINT_PTR m_nTimerID;
    static UINT  m_nMaxTooltipLength;
    static const UINT m_nTaskbarCreatedMsg;
    static HWND  m_hWndInvisible;

    static BOOL GetW2K();

    static void GetTrayWndRect(LPRECT lprect);
    static BOOL GetDoWndAnimation();

// 消息映射函数
public:
    LRESULT OnTimer(UINT nIDEvent);
    LRESULT OnTaskbarCreated(WPARAM wParam, LPARAM lParam);

    LRESULT OnSettingChange(UINT uFlags, LPCTSTR lpszSection);

};
```

第 13 章　Windows GDI

本章将讲解 Windows 图像程序设计的相关概念、原理与接口函数使用方法。

首先介绍 Windows GDI（图形设备接口，Graphics Device Interface）所能完成的功能及在开发时所需要使用的头文件和库文件，然后介绍 GDI 对象及基本操作。在介绍各种 GDI 对象时，以实例的形式重点讲解其所能实现的功能和使用方法。

13.1　GDI编程接口概述

GDI 是 Graphics Device Interface（图形设备接口）的简称，当 Windows 应用程序需要显示点、线、图像、文字等内容，在显示器或打印输入这些内容时，就需要使用到 GDI。图形设备接口是 Windows 图形界面的基础。应用程序当然不能直接操作系统的硬件，比如显卡，GDI 就为应用程序提供了相关的接口。

GDI 不是可以实现这种功能的唯一程序设计接口，GDI 只是其中最基本的。除了 GDI 外，还有 GDI+、OpenGL、DirectX、Windows Image Acquisition 等可以实现类似或更高级的功能。本书限于篇幅，只介绍最基本的 GDI。

13.1.1　Windows GDI的功能

简单列举，GDI 包括如下一些功能。

◇获取输出设备的输出，可以截获当前显示器显示的内容，包括整个屏幕、指定窗口或其他任意部分。

◇控制显示器的输出内容，在显示器的指定位置输出指定的内容。应用程序可以指定输出的位置、颜色、大小、区域。

◇可以以点、线条（直线、曲线）、图形、位图、文字等方式输出。

◇可指定输出内容的颜色、大小、位置。

◇可将输出内容进行变形、变色、字体的处理等。

13.1.2　链接库与头文件

在应用程序中如果需要使用 GDI，那么需要包含特定的头文件，在链接时需要指定链接的 lib 文件。

GDI 应用程序接口并不是由 keme132.dll 导出，而大多是由 Gdi.dll 和 Gdi32.dll 提供。其相关的函数接口、数据类型等都在 Wingdi.h 中声明（已经由 Windows：h 引入）。在程序开发时，需要链接到 Gdi32.lib。

13.2　设备上下文(DC)＼输出操作与图形对象

DC(设备上下文，Device Contexts)是 GDI 程序设计中最基本也是最重要的概念。DC 是一种对象，设定了图形输出的特性与属性。

系统中可以具有多个 DC，每一个 DC 都有一个与之对应的关联设备，有一系列的图形对象，还包括图形模式。

13.2.1 设备上下文类型与关联设备

一个 DC 所关联的设备驱动决定了通过此 DC 进行的图形输出操作（文字输出、绘制、填充等）会被哪个设备（包括显示器、打印机、绘图仪）或设备的子区域（窗口客户端等）所接收。

每一个 DC 都必须关联到一个特定的图像输出设备，这些设备可以是真实存在的物理设备，包括显示器、打印机、绘图仪等，也可以是虚拟的设备。比如一个 DC 关联到了 VGA 屏幕（显示器），那么所有通过这个 DC 进行的图形输出操作，都会在上显示器上显示出来。

DC 也可以只是设备全部输出范围的一部分。比如界面上某个窗口的客户区也可以有 DC 与之对应，对这样的 DC 进行操作只会影响到窗口客户区。

设备上下文还具有 4 种类型，分别是“显示”、“打印机”、“内存”、“信息”。

显示类型 DC 关联到了显示设备，所有的图像输出操作将直接反映在显示器上。

“打印机”DC 类型的绘制操作将反映在打印机上；对“内存”类型的 DC 的绘制操作将所有操作结果反映到内存数据中，不会有输出显示。而“信息”类型的 DC 可以用于获取 DC 的相关信息和属性等数据。

应用程序在进行图像输出时，只需要关心 DC 的类型，如果需要将图像输出到特定的设备只需创建相应类型的 DC 即可。而对不同类型的 DC 的操作方式都是统一的，应用程序不需要关注设备硬件的异同，无论哪种设备数据接口、哪种通信数据的格式等都是同样的处理，GDI 的接口都是相同的。

13.2.2 图形对象的作用及与DC的关系

一个DC中还包括了一系列的图形对象,所谓图形对象会影响到对DC的输出操作的结果。

比如 TextOut 函数就是一种输出操作，这种操作会将文字显示在 DC 所关联的设备上，那么“字体”这种图形对象就决定了输出的文字的形式，包括具体字的形状、大小、颜色等。

再比如 LineTo 函数也是一种输出操作,这种操作会在指定的 DC 所关联的设备上绘制出直线，而“画笔”这种图形对象就决定了所输出线条的颜色、粗细、虚实、线头形状等。

DC 中可以拥有如表 13.1 所示的图形对象。

表 13.1 图形对象

图 形 对 象	作　用
位图（Bitmaps）	影响位图创建、位图操作（绘制、拉申、旋转、滚动等）和保存的结果形式
画刷（Brushes）	影响对形状、区域、路径等的填充操作结果，包括填充使用的颜色、花色（片色还是阴影等）等
画笔（Pen）	影响绘制的线条的曲线，包括颜色、粗细、虚实、线头形状等
字体（Fonts）	影响文字输出的字体
逻辑调色板（Logical palette）	所有输出内容可用的颜色
路径（Paths）	所有输出内容的形状、边缘等
区域（Regions）	所有输出的位置、尺寸等

当一个应用程序创建一个 DC 时，系统会为 DC 创建默认的图形对象（路径和位图除外）。

1．为 DC 选择图形对象

应用程序可以创建新的图形对象，比如使用 CreatePen 创建画笔对象，使用 CreateBrushIndirect 等函数创建画刷。

如果应用程序需要在 DC 上绘制新线条、形状等自行创建的图形对象时需要将图形对象选入(Select Into) DC 中，这样 DC 中原有的同类型图形对象会被新的对象所替换，当进行绘制等操作时就会使用新选入的图形对象。

对同一种类型的图形对象，DC 中只会有一个“当前”(Current)对象，比如当前画刷、当前字体等，所有的操作都是使用到相应的“当前”对象。

为 DC 设置当前对象最重要的 API 函数是 SelectObject。SelectObject 函数的功能是将对象“选择入”DC，使对象成为 DC 当前的对象，并替换原有对象。

```
HGDIOBJ SelectObject(
  HDC hdc,              // handle to DC
  HGDIOBJ hgdiobj       // handle to object
);
```

hdc 参数是 DC 的句柄，hgdiobj 参数是对象句柄。

大多数设备对象都是使用 SelectObject 进行选择，当然有很多设备对象也可以通过特殊的 API 来设置，比如：SetDCBrushColor、GetDCBrushColor、SetDCPenColor、GetDCPenColor、SelectPalette 等。

2．获取图形对象及信息

下面是一些常用的获取图形对象信息的 API 函数。

GetCurrentObj ect 函数的功能是获取 DC 中指定类型的图形对象，函数原型如下：

```
HGDIOBJ GetCurrentObject(
  HDC hdc,              // DC 句柄
  UINT uObjectType      // 目标类型
);
```

uObj ectType 参数指定图形对象类型，其值可以是 OBJ_ BITMAP、OBJ BRUSH、OBJ FONT、OBJ PAL、OBJ_PEN 等。分别用于获取当前位图对象、画刷对象、当前的字体对象、调色板对象和画笔对象。

GetObj ect 函数的作用是获取对象的具体信息。

```
int GetObject(
  HGDIOBJ hgdiobj,      // 图形对象句柄

  int cbBuffer,         // 对象信息大小
  LPVOID lpvObject      // 存储对象信息的缓冲区
);
```

通过 lpvObj ect 缓存返回值。根据对象的类型不同，返回值不同，如果是画笔对象，返回的值类型为 LOGPEN 结构，如果是画刷对象，返回的值是 LOGBRUSH 结构等。

LOGPEN 和 LOGBRUSH 结构定义如下：

```
typedef struct tagLOGBRUSH {
  UINT     lbStyle;
  COLORREF lbColor;
  LONG     lbHatch;
} LOGBRUSH, *PLOGBRUSH;

typedef struct tagLOGPEN {
  UINT     lopnStyle;
  POINT    lopnWidth;
  COLORREF lopnColor;
} LOGPEN, *PLOGPEN;
```

3．常备(Stock)图形对象

GDI 已经将一些常用的图形对象作为常备图形对象。这样在使用这此常备图形对象时，就可以不用再创建进程，减化程序。

如果需要获得常备图形对象的句柄，调用 API 函数 GetStockObj ect。

```
HGDIOBJ GetStockObject(
  int fnObject     // 常备图形类型
);
```

参数 fnObj ect 标识了需要的常备图形对象。其值可以为表 13.2 中的任意一个。

表 13.2　　常备图形对象

常　量	意　义
BLACK_BRUSH	黑色的画刷
DKGRAY_BRUSH	深灰色画刷
DC_BRUSH	DC 画刷，默认为白色，可由 SetDCBrushColor 设置
GRAY_BRUSH	灰色画刷
LTGRAY_BRUSH	浅灰色画刷
NULL_BRUSH	空画刷
WHITE_BRUSH	白色画刷
BLACK_PEN	黑色画笔
DC_PEN	DC 画笔，默认为白色，可由 SetDCPenColor 设置
WHITE_PEN	白色画笔
ANSI_FIXED_FONT	Windows fixed-pitch 系统字体
ANSI_VAR_FONT	Windows variable-pitch 系统字体
DEVICE_DEFAULT_FONT	设备相关字体
DEFAULT_GUI_FONT	菜单、对话框的默认字体，一般为 MS Sans Serif
OEM_FIXED_FONT	OEM fixed-pitch 字体
SYSTEM_FONT	系统字体，Windows XP 下为 Tahoma
SYSTEM_FIXED_FONT	Fixed-pitch 系统字体
DEFAULT_PALETTE	默认调色板

13.2.3　各类图形对象的具体属性与作用

图形对象有很多种，下面详细介绍每一种图形对象所包含的属性和在 DC 中的作用。

1．位图(bitmap)

位图是一种图形对象（Graphical　Obj ect），是用于表示图像(images)的数据和结构。程序使用位图来操作和管理图像。

位图数据包括 3 个部分。

第一个部分是一个位图信息头，用于说明分辨率、长度与宽度（像素）、表示一个像素点所使用的数据大小（8 位、16 位、24 位等），一般是 BITMAPINFOHEADER 类型的结构。

第二个部分是可选的调色信息，一般是 RGBQUAD 类型的结构或一组 RGBTRIPLE 类型的数据。

第三部分是各个像素点的颜色表示，表示一个像素点的颜色有多种长度的数据。比如如果使用 4 位表示一个像素，那么可以有 16 种颜色，如果使用 16 位数据表示一个像素点，那么可以有 65 536 种颜色。

位图数据的这 3 个部分依次保存在内存中。如果将内存中的位图数据保存为文件，那么还需要在第一个部分之前增加位图文件头。

位图的格式有多种不同的版本，但是大同小异，可参考 13.7 节。

2．画笔刷(Brush)

应用程序可以使用画笔刷绘制各种形状的图形、线条、阴影等。

画笔刷又分为逻辑画笔刷和物理画笔刷，逻辑画笔刷可以绘制各种形状，而物理画笔刷是基于逻辑画笔刷的位图。

有关画笔刷的详细使用方法将在实例 13-1 中介绍。

3．字体(Font)

将文本显示在界面上是最常见的图形设备输出形式，显示文本就需要指定文本所使用的字体。字体是文本的形状、大小等，用于将文件绘制在显示输出设备上。

字体包括 3 个方面，一是字体图样(Typeface)，即表示一个字所使用的像素点阵；二是样式（Style)，包括粗、细、加黑等；三是尺寸，以像素点为单位。

GDI 提供了相关的函数，实现处理文本、设置文本字体和格式，显示文本等功能。

当使用 TextOut 等函数通过 DC 输出文字时，所输出的文字的字体由 DC 的当前字体对象定义。

4．路径(Path)

Path 在 GDI 中用于表示任意的形状。

Path 可以是“填充的”(filled)、“边沿勾勒的”(outlined)。分别可以使用不同的颜色来填充其内部，勾勒其边沿。

Path 在被创建后可以填充、勾勒、变形、裁减(Clip)，还可以转换为“区域”(Regions)。

5．画笔(Pens)

“画笔”可以在界面上绘制出线条。所绘制的线条可以是直线、手绘线和各种函数曲线。

GDI 程序中的画笔分为两种，一种是装饰画笔(Cosmetic Pens)，另一种是几何画笔(Geometric Pens)。

装饰画笔：是设备单元，较快，不受坐标空间变化的影响，所绘制的线条宽度一定，只具有宽度、样式和颜色 3 种属性。

几何画笔：是几何单元，较慢，受到坐标空间变化的影响，具有多种属性。

画笔还具有属性，包括宽度、样式、颜色、模式(Pattein)、影线(Hatch)、线端(End Cap)、结合方式(join)等。

样式表示了画笔所绘制的线条可以是实线，短横虚线、点虚线、短横一点虚线、短横-点-点虚线等。

画笔具有阴影(Hatch)、实心(Solid)空(Hollow)、自定义等模式。

画笔为阴影模式时，影线属性表示了阴影使用哪种影线。

当使用 LineTo 等函数在 DC 上输入线条时，线条的样式由 DC 当前的画笔对象定义。

6．区域(Regions)

区域表示了 DC 中的特定位置和范围，可以是矩形、多边形、椭圆及其组合。

在创建了区域后，可以对区域进行“组合”、“比较”、“填充”、“着色”、“反转”、“镶框”等操作。还可以用来定位光标的位置。

同时区域还可以用于“修剪”。

7．调色板

调色板定义了 DC 可以显示的颜色的数量和颜色值。有关调色板的使用会在 13.10 节介绍。

13.2.4　绘制、填充、写入等图形输出操作

所谓输出操作就是指在输出设备上绘制出图形、线条、文字等。每一次输出操作都使用 DC 的当前图形对象。比如在输出设备上绘制线条需要使用“画笔”图形对象。

有很多 API 函数专门用于进行输出操作。这一类 API 函数的功能都是在界面上“绘制”，如表 13.3 所示。

表 13.3　　用于绘制、填充等操作的图形对象

对 象 分 类	中文对象名	GDI 对象名
形状	矩形	Rectangle
	圆角矩形	RoundRect
	圆、椭圆	Ellipse
	椭圆切（弓形、Chords）	Chord
	扇形（Pie）	Pie
	多边形	Polygon、PolyPolygon
线条	直线	LineTo 等
	曲线	AngleArc、ArcTo、PolyBezier、Polyline、PolyPolyline 等
文字	文字	DrawText、ExtTextOut、TextOut

13.2.5　修剪与坐标变换

修剪与坐标变换是两种重要的操作。

1．修剪(Clipping)

修剪是一种限制输出范围和位置的机制。

修剪通过路径(Paths)和区域(Regions)对输出操作的范围作出限制，对输出的内容进行处理。“修剪区域”和“修剪路径”是两种图形对象，都可以“选择入”DC。分别可以使用规制和不规则图形来对输出的范围作出限制。

如果在 DC 选择入了“修剪”图形对象，那么在进行输出操作时，并不是整个 DC 的范围都有效。所输出的文字、线条、图形等只有位于修剪区之内的才会被输出。

2．坐标空间及变化

坐标空间及变化可以使应用程序能够将图像输出进行坐标变换，包括比例变化（放大、缩小）、旋转、扭曲、反转。

GDI 中的坐标空间是一个二维平面空间，使用二维坐标表示。

坐标空间有 4 种：world、page、设备和物理设备（客户区、桌面、打印纸）。

可以基于坐标空间进行坐标空间的数学变换，以达到变化图形的目的。

13.2.6　设备上下文的图形模式

所谓DC的图形模式是指在通过DC进行输出操作时所遵循的原则。例如颜色混合的方式、输出出现的方式、输出的尺度等。

包括 5 个类型的图形模式，如表 13.4 所示。

表 13.4　　不同的图形模式

图形模式	描　述	API	值
Background	在进行位图输出、文字输出等操作时，如何处理输出图形的背景色与在输出位置上系统原有的图像的混和关系。比如在进行文字输出时，可以将文字背景一起输出，这样背景色将覆盖输出区域原有的图像，也可以设置保留文字输出背景区域原有的图形	GetBkMode SetBkMode	OPAQUE TRANSPARENT
Drawing	处理前景色与输出位置上当前图像颜色的混合关系	GetROP2 SetROP2	R2_BLACK R2_COPYPEN R2_NOP R2_XORPEN
Mapping	图形输出如何在逻辑与设备空间之间映射的关系	GetMapMode SetMapMode	MM_ANISOTROPIC MM_HIENGLISH MM_TEXT
Polygon-fill	使用画刷填充复杂多边形的模式	GetPolyFillMode SetPolyFillMode	ALTERNATE WINDING

续表

图形模式	描　述	API	值
Stretching	当位图与显示的 DC 大小不匹配时，设置位置的拉伸方式	GetStretchBltMode SetStretchBltMode	BLACKONWHITE COLORONCOLOR STRETCH_HALFTONE WHITEONBLACK

13.3　一个最简单的GDI程序

本节通过一个最简单的 GDI 程序演示 GDI 程序设计中的各基本方面，包括 DC 的操作方法、图形对象的选择、绘制等输出操作，还说明了 GDI 程序中对颜色的表示方式。

13.3.1　示例

先看一下 13-1 的代码和运行效果。13.3.2 小节～13.3.5 小节将分析本实例程序的实现原理。

1．实例 13-1　基本的 GDI 程序

```
/* ************************************
* gdi.c 最简单的 GDI 程序
**************************************/
/* 头文件 */
#include <Windows.h>
/* 函数声明 */
void GdiOut(HDC hdc);

// WinMain
int WINAPI WinMain(
                    HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow
                    )
{
     HDC hdc = GetDC(NULL);
     GdiOut(hdc);
     ReleaseDC(NULL, hdc);
}
/************************************
* VOID GdiOut(HDC hdc)
* 功能：演示 GDI 基本功能的使用
* 参数：HDC hdc，用于显示所绘制的图像
************************************/
VOID GdiOut(HDC hdc)
{
     HPEN hpen, hpenOld;
     HBRUSH hbrush, hbrushOld;
     // 初始的颜色
     BYTE bRed  = 0;
     BYTE bGreen  = 0;
     BYTE bBlue = 0;
     // 画笔的颜色——正黑
     COLORREF cPen = RGB(bRed, bGreen, bBlue);
     // 从 COLORREF 类型拆解颜色，设置画笔刷的颜色，这里为紫偏蓝
     COLORREF cBrush = RGB(233, GetGValue(cPen),255);
     // 创建画笔
     hpen = CreatePen(PS_SOLID, 10, cPen);
     // 创建画笔刷
     hbrush = CreateSolidBrush(cBrush);
     // 为 DC 选择画笔和画笔刷
     hpenOld = SelectObject(hdc, hpen);
     hbrushOld = SelectObject(hdc, hbrush);
     // 绘制线条
     LineTo(hdc,500,500);
     // 使用初始的画笔
     SelectObject(hdc,hpenOld);
     // 绘制矩形
     Rectangle( hdc, 200, 200, 500, 500 );
     // 释放资源
     DeleteObject(hpen);
     SelectObject(hdc, hbrushOld);
     DeleteObject(hbrush);
}
```

2．实例运行结果

程序运行完成后，在桌面上绘出一条黑色直线和一个紫色矩形，如图 13-1 所示。

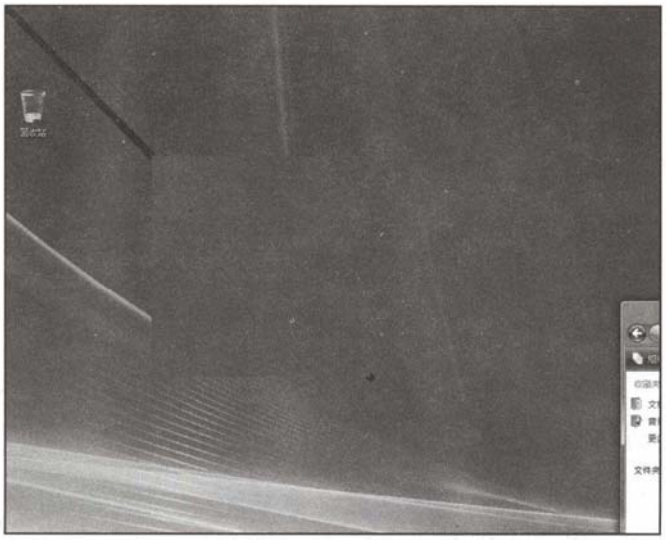

图 13-1　实例 13-1 运行效果

13.3.2　DC的操作

本节包括以下 API 函数的介绍：

◇ GetDC

◇　CreateDC

◇　ReleaseDC

◇ DeleteDC

(1) GetDC。

实例 13-1 的 Main 函数调用了 API 函数 GetDC。GetDC 函数的功能是获取指定窗口客户区的 DC。

GetDC 函数原型如下：

```
HDC GetDC(
  HWND hWnd   // handle to window
);
```

函数调用后会返回 hWnd 参数所指定的窗口的客户区所对应的 DC。函数返回的是 DC 的句柄，数据类型为 HDC。

在本实例中调用时 GetDC 指定的窗口句柄为 NULL。如果 hWnd 参数设置为 NULL，那么函数会返回整个桌面的 DC。

(2) CreateDC。

除了 GetDC 函数外 CreateDC 也可以获取 DC 的句柄。

```
HDC CreateDC(
  LPCTSTR lpszDriver,        // 设备类型
  LPCTSTR lpszDevice,        // 设备名
  LPCTSTR lpszOutput,        // 应为空
  CONST DEVMODE* lpInitData // 初始化所用数据
);
```

不同于 GetDC，CreateDC 可以获取非显示器输出 DC。

lpszDriver 参数指定了设备的类型，如果需要显示器的 DC，需设备为 DISPLAY;如果需要获取打印机的 DC，一般设置为 WINSPOOL。

lpszDevice 参数指定了设备名，如果同一类型的设备有多个，由此参数指定，可设置

为 NULL。

lpszOutput 参数必须设置为 NULL，lpInitData 参数是指向 DEVMODE 的指针，是设备初始化时需要使用到的数据。如果 lpszDriver 设置为了 DISPLAY，那么此参数必须为 NULL，如下所示：

```
HDC hScreenDC = CreateDC("DISPLAY", NULL, NULL, NULL);
```

在使用完 DC 后，需要使用 ReleaseDC 和 DeleteDC 来释放相关资源。

(3) ReleaseDC。

ReleaseDC 的作用是释放 DC，使其他应用程序可以使用。

```
int ReleaseDC(
  HWND hWnd,  // 窗口句柄
  HDC hDC     // DC 句柄
);
```

(4) DeleteDC。

DeleteDC 的功能是释放 DC 的相关系统资源。

```
BOOL DeleteDC(
  HDC hdc     // DC 句柄
);
```

实例 13.1 将 GetDC 获得的 DC 句柄传递给了 GdiOut 函数。GdiOut 函数使用完成后，调用了 ReleaseDC 函数释放了 DC。

13.3.3 颜色的表示

在实例 13-1 中在创建画笔和画刷时使用了颜色。

1．COLORREF 类型和 RGB 宏

在 GDI 中使用红、绿、蓝三原色的组合来表示颜色。使用 3 个 8 位的数据组合来表示颜色，称作 RGB 字节，可以表示 0x1000000 种颜色。RGBQUAD 数据结构用于表示 RGB 颜色，也可以使用 COLORREF 数据类型来表示，COLORREF 与 DWORD 大小相同，RGB 宏可以将颜色表示为 COLORREF。GetRValue、GetGValue，GetBValue 这 3 个宏可以将 COLORREF 拆解为三原色的字节。

根据三原色的原理，如正黑色为：

```
RGB(0,0,0)
```

正紫色为：

```
RGB(255,0,255)
```

RGB 宏的 3 个参数分别对应红(缩写为 R)、绿(缩写为 G)、蓝（缩写为 B)。

实例 13-1 的 GdiOut 函数的前几行中演示了颜色的表示和操作方法。

2．DC 中可用的颜色信息

使用 NUMCOLORS 参数，调用 GetDeviceCaps API 函数即可获得指定 DC 的颜色数量。再使用 EnumObjects 函数就可以列举指定 DC 的所有颜色的表示。

```
cColors = GetDeviceCaps(hdc, NUMCOLORS);
```

13.3.4 图形对象：画刷和画笔

本小节开始详细介绍各图形对象的操作，首先介绍最基本的图形对象画刷和画笔。

1．定义画笔

HPEN 是画笔对象的句柄数据类型，用于标识一个画笔对象。
新建一个画笔对象使用 API 函数 CreatePen。

```
HPEN CreatePen(
  int fnPenStyle,       // 画笔样式
  int nWidth,           // 线条宽度
  COLORREF crColor      // 线条颜色
);
```

参数 finPenStyle 是画笔的样式。如果设置为 PS_SOLID 表示绘制的线条是实线，PS_DASH、PS_DOT、PS_DASHDOT、PS_DASHDOTDOT 表示了 4 种虚线样式，分别是短线、点、短线和点间隔，短线间间隔两个点。PS_NULL 表示线不可见。

nWidth 是画笔所绘制线条的宽度，crColor 参数指定线条的颜色。

在实例 13-1 中程序调用 CreatePen 创建一个宽度为 10 的黑色的实线。

```
COLORREF cPen = RGB(bRed, bGreen, bBlue);
hpen = CreatePen(PS_SOLID, 10, cPen);
```

此外还可以使用 ExtCreatePen 函数和 CreatePenIndirect 创建画笔。

2．定义画刷

HBRUSH 是画刷对象句柄数据类型，用于标识一个画刷对象。
新建一个画刷使用 API 函数 CreateHatchBrush、CreatePattemBrush、CreateSolidBrush。
CreateHatchBrush 创建一个影线画刷，影线画刷不是实心的，当用于填充时，填充的内容是阴影线。

```
HBRUSH CreateHatchBrush(
  int fnStyle,          // 影线样式
  COLORREF clrref       // 前景色
);
```

参数 fnStyle 是影线的样式，包括斜线方向和是否交叉等。clrref 参数是前景色。
CreateSolidBrush 函数的功能是创建实心画刷，当用于填充时，填充使用纯色。

```
HBRUSH CreateSolidBrush(
  COLORREF crColor   // 画刷颜色
);
```

参数 crColor 指定了画刷的颜色。

CreatePatternBrush 的功能是创建一个“格式”画刷，在使用格式画刷进行填充时，所填充的内容是位图。

```
HBRUSH CreatePatternBrush(
  HBITMAP hbmp   // bitmap 句柄
);
```

在实例 13-9 中，演示了多种画刷的使用。在不同的区域中分别使用实心画刷、阴影画刷和位置画刷进行填充。运行效果参见图 13-11。

3．图形对象的选择

‘ 在定义好画笔和画刷后，如果在绘制图形和线条时需要使用所定义的画笔对象和画刷对象就需要为 DC 选择当前图形对象。

设置当前图形对象使用 API 函数 SeleteObj ect。

```
HGDIOBJ SelectObject(
  HDC hdc,              // DC 句柄
  HGDIOBJ hgdiobj       // 对象目标句柄
);
```

SelectObj ect 函数将参数 hdc 所指定的 DC 与 hgdiobj 参数同类的图形句柄的当前对象设置为 hgdiobj。由于 DC 中每一类图形对象有一个“当前”图形对象（一个 DC 中同一时

刻只能有一个画笔对象，也只能有一个画刷对象），所有在设置当前对象后同类的原有对象会被替换。如果 hgdiobj 是一个画刷，那么原来的 DC 中的画刷就被替换。返回值就是被替换的图形对象。

13.3.5 输出操作：绘制图形和线条

在设置对象就可以进行绘制了。实例 13—1 中绘制了一条直线和一个矩形。

```
BOOL LineTo(
  HDC hdc,     // 设备句柄
  int nXEnd,   // 线条终点坐标 x 值
  int nYEnd    // 线条终点坐标 y 值
);
```

绘制线条使用 LineTo 函数。，

参数 nXEnd 和 nYEnd 是线条终点的坐标值，画线所使用的画笔对象就是 DC 的当前画笔对象。在调用 LineTo 函数前已经由 SelectObject 函数设置。

LineTo 函数并不能指定线的起点，而是从画笔对象的“当前点”开始画，画完后当前点变为绘制的终点。可以使用 MoveToEx 函数移动画笔的当前点。当前点初始位置为{0，0)。

绘制矩形使用 API 函数。

```
BOOL Rectangle(
  HDC hdc,          // DC 句柄
  int nLeftRect,    // 矩形左上角 x 坐标
  int nTopRect,     // 矩形左上角 y 坐标
  int nRightRect,   // 矩形右下角 x 坐标
  int nBottomRect   // 矩形右下角 y 坐标
);
```

Rectangle 函数的参数指定了所绘制的矩形的位置和长宽。Rectangle 函数所绘制的矩形使用 DC 的当前画刷进行填充，而且 Rectangle 所绘制的矩形是带边框的，边框线条的样式是当前画笔的线条样式。

实例 13-1 在绘制矩形前将 DC 的画笔对象重新选择为原来的画笔对象。这是一个宽为 1 的黑色实线。

13.4 文字和字体

本节主要内容是介绍使用 GDI 将指定字体的文字显示在界面上的实现方法。本节先给出实例和实例的运行结果，然后再陆续介绍原理。

实例 13-2 实现了指定字体的文本的显示，并将在 13.4.1 小节～13.4.3 小节中详细介绍实现的方法。

13.4.4 小节将介绍 DC 图形模式的设置，关于 DC 的图形模式已经在 13.2.6 小节介绍了相关原理，本节将通过实例 13-2 说明图形模式对图形输出的影响。

13.4.5 小节将介绍如何获取系统中完装了的字体，实例 13-3 实现了遍历系统中的字体的功能。

在程序设计时，可以从系统已经安装的字体中进行选择。

1．实例 13-2 输出指定字体的文本

如下是实例 Text.exe 的程序代码，在 WnMain 函数中，先调用了 ChooseNewFont 函数，让用户选择通过字体选择对话框选择了字体，然后使用默认字体和用户选择的字体轮流输出不同的文字。

```
/* ************************************
* text.c 文本和字体
************************************/
/* 头文件 */
#include <Windows.h>
/************************************
* HFONT ChooseNewFont()
* 功能：选择字体
* 返回值：返回字体句柄
************************************/
HFONT ChooseNewFont()
{
    CHOOSEFONT cf;
    LOGFONT lf;
    HFONT hfont;
     // CHOOSEFONT 结构
    cf.lStructSize = sizeof(CHOOSEFONT);
    cf.hwndOwner = (HWND)NULL;
    cf.hDC = (HDC)NULL;
    cf.lpLogFont = &lf;
    cf.iPointSize = 0;
    cf.Flags = CF_SCREENFONTS;
    cf.rgbColors = RGB(0,0,0);
    cf.lCustData = 0L;
    cf.lpfnHook = (LPCFHOOKPROC)NULL;
    cf.lpTemplateName = (LPSTR)NULL;
    cf.hInstance = (HINSTANCE) NULL;
    cf.lpszStyle = (LPSTR)NULL;
    cf.nFontType = SCREEN_FONTTYPE;
    cf.nSizeMin = 0;
    cf.nSizeMax = 0;
     // 选择字体对话框
    ChooseFont(&cf);
     // 得到 HFONT，返回
    hfont = CreateFontIndirect(cf.lpLogFont);
    return (hfont);
}
/************************************
* WinMain
* 功能：选择字体，并将文字显示在界面上
************************************/
```

```
INT WinMain(
              HINSTANCE hInstance,
              HINSTANCE hPrevInstance,
              LPSTR lpCmdLine,
              int nCmdShow
              )
{
     HDC hdc = GetDC(NULL);
     int XIncrement = 10;
     int YStart = 50;
     TEXTMETRIC tm;
     HFONT hfntDefault, hfntChoose;
     SIZE sz;
     UINT uAlignPrev;
     LPSTR lpszString1 = "字符串一";
     LPSTR lpszString2 = "字符串二";
     LPSTR lpszString3 = "字符串三";
     DWORD dwStrLen1 = lstrlen(lpszString1);
     DWORD dwStrLen2 = lstrlen(lpszString2);
     DWORD dwStrLen3 = lstrlen(lpszString3);

     // 选择字体
     hfntChoose = ChooseNewFont();
     // 设置颜色
     SetBkColor(hdc,RGB(255,255,255));
     SetTextColor(hdc,RGB(255,0,0));
     SetBkMode(hdc,TRANSPARENT);

     // 输出字符串一
     TextOut(hdc, XIncrement, YStart, lpszString1, dwStrLen1);

     // 为字符串二设置输出位置
     GetTextExtentPoint32(hdc, lpszString1, dwStrLen1, &sz);
     XIncrement += sz.cx;
     GetTextMetrics(hdc, &tm);
     XIncrement -= tm.tmOverhang;
     // 改变字体
     hfntDefault = SelectObject(hdc, hfntChoose);
     // 输出字符串二
     TextOut(hdc, XIncrement, YStart, lpszString2, dwStrLen2);
     // 设置字符串三的输出位置
     GetTextExtentPoint32(hdc, lpszString1, dwStrLen1, &sz);
     XIncrement = 10;
     YStart += sz.cy;
     GetTextMetrics(hdc, &tm);
     XIncrement -= tm.tmOverhang;
     // 设置为默认字体
     SelectObject(hdc, hfntDefault);
     // 输出字符串三
     uAlignPrev = SetTextAlign(hdc, TA_UPDATECP);
     MoveToEx(hdc, XIncrement, YStart, (LPPOINT)NULL);
     TextOut(hdc, 0, 0, lpszString3, dwStrLen3);
     SetTextAlign(hdc, uAlignPrev);
     // Clear
     DeleteObject(hfntChoose);
     SetBkMode(hdc, OPAQUE);
     DeleteDC( hdc );
     return 0;
}
```

2．实例运行结果

运行 Text.exe，会首先弹出如图 13-2 所示的字体选择对话框，用户可以选择系统中已经安装的任何字体。

在选择了字体后，程序会在屏幕的左上角输出 3 行文字，“字符串一”、“字符串二”和“字符串三”，其中字符串一使用了和字符串三使用了 DC 在创建时的默认字体，而字符串二使用了用户通过字体。选择对话框选择的字体。效果如图 13-3 所示。

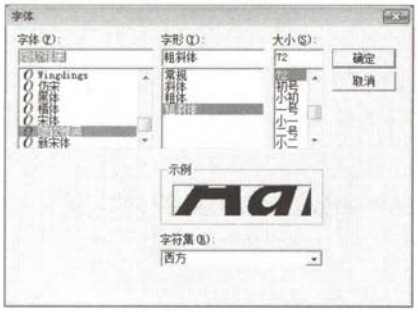

图 13-2　选择字体

图 13-3　使用选择的字体输出字符串

13.4.1　选择、设置字体

实例代码中，使用了 HFONT 类型的变量。HFONT 是表示字体对象的句柄，创建字体后，HFONT 唯一标识一个字体对象。

1．创建字体对象

CreateFont、CreateFontIndirect 等函数可以创建字体对象。

CreateFontIndirect 函数原型如下：

```
HFONT CreateFontIndirect(
  CONST LOGFONT* lplf   // 字体对象
);
```

lplf 参数是指向 LOGFONT 结构的指针，LOGFONT 结构如下：

```
typedef struct tagLOGFONT {
  LONG lfHeight;
  LONG lfWidth;
  LONG lfEscapement;
  LONG lfOrientation;
  LONG lfWeight;
  BYTE lfItalic;
  BYTE lfUnderline;
  BYTE lfStrikeOut;
  BYTE lfCharSet;
  BYTE lfOutPrecision;
  BYTE lfClipPrecision;
  BYTE lfQuality;
  BYTE lfPitchAndFamily;
  TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT, *PLOGFONT;
```

LOGFONT 的成员较多，都是指定字体的相关属性。

CreateFont 函数的功能和 CreateFontIndirect 函数相同，但是没有使用 LOGFONT 作为参数，而是直接将各个属性作为参数，CreateFont 函数的参数几乎与 LOGFONT 的成员一一对应。

2．填充 LOGFONT 结构

用于创建字体的 LOGFONT 结构成员较多，各成员的意义也不十分明确，在使用 LOGFONT 表示一个字体时，填充比较复杂。但是在程序设计时，一般可由程序来完成填充。

有两种方法可供选用。一是使用 ChooseFont API 函数弹出选择字体选择对话由用户选择。ChooseFont 函数会返回用户选择的字体所对应的 LOGFONT 结构。ChooseFont 函数原型如下：

```
BOOL ChooseFont(
    LPCHOOSEFONT lpcf
);
```

参数 lpcf 是指向 CHOOSEFONT 结构的指针。

二是遍历系统中已经安装的字体，遍历程序会返回每一个字体所对应的 LOGFONT 结构，

字体遍历的方法在 13.4.6 小节实例 13-3 中有详细介绍。

在得到 LOGFONT 结构后，直接调用 CreateFontIndirect API 函数就可以得到字体的句柄 HFONT。

3．实例分析

实例 13-2 中 ChooseNewFont 函数调用了 ChooseFont API 函数,弹出了字体选择对话框，如图 13-2 所示。在调用 ChooseFont 时设置了 CHOOSEFONT 结构，作为 ChooseFont 的参数。

CHOOSEFONT 结构的 lpLogFont 成员是指向 LOGFONT 的指针。在调用 ChooseFont 函数后，lpLogFont 所指向的 LOGFONT 结构与用户所选择的字体对应，可以直接作为 CreateFontIndirect 函数的参数。

13.4.2 选择字体图形对象

字体也是一种 DC 图形对象，与画刷对象，画笔对象等 DC 图形对象类似。在通过 DC 进行文字输出操作时，输入的文字的字体所使用的就是使用 DC 的当前字体对象。因此在显示指定字体的文字前，需要将所使用的字体选择入 DC。

为 DC 选择字体对象的方式与选择画笔对象和画刷对象的方法类似。

在本节实例 13-2 中，WinMain 函数调用 TextOut API，显示字符串二之前，将代表用户选择的字体的字体对象的选择到了 DC。因此输出的字符串二的文本所显示的字体就是用户选择的字体。

而在输出字符串一时没有选择新的字体，而是使用 DC 默认的字体，在输入字符串三时，又将默认字符重新选择回了 DC。因此输出的字符串一和字符串三中文字的字体是一样的，都是 DC 默认字体。

13.4.3 文字的颜色

本实例输出的文字的颜色是红色的（如图 13-4 所示）。

实例代码中，在调用 TextOut 函数将字符串输出前调用了 SetTextColor 函数。SetTextColor 函数的功能是设置 DC 输出的字符的颜色。函数原型如下：

```
COLORREF SetTextColor(
  HDC hdc,              // DC 句柄
  COLORREF crColor      // 字符颜色
);
```

如果不进行设置，DC 也具有默认的文字输出颜色。在设置了新的颜色后，原来的颜色将被替换，SetTextColor 的返回值是原有颜色。

DC 的文字输出颜色可以设置多次，每次都会替换掉原有的颜色。但是已经输出的文字的颜色不会发生变化。

在本实例中设置如下：

```
SetTextColor(hdc,RGB(255,0,0));
```

设置的颜色为红色。

13.4.4 输出文字

在进行了输出文字的字体颜色等设置后，就可以使用文字输出操作函数将文字显示在界面

上。TextOut 函数是最基本的文字输出函数，函数原型如下：

```
BOOL TextOut(
  HDC hdc,                // DC 句柄
  int nXStart,            // 起始位置 x 坐标
  int nYStart,            // 起始位置 y 坐标
  LPCTSTR lpString,       // 字符串
  int cbString            // 文字数目
);
```

在调用 TextOut 函数时通过参数指定起始位置，需要输出的字符串及字符串长度。

DrawText 函数有较 TextOut 函数更为高级的功能。DrawText 函数可以输出多行的文本（使用换行符）还可以指定文本的对方方式，包括左对齐、右对齐、居中对齐等。

```
int DrawText(
  HDC hDC,                // DC 句柄
  LPCTSTR lpString,       // 输出文字
  int nCount,             // 文字长度
  LPRECT lpRect,          // 有效的 Rect
  UINT uFormat            // 文字输出选项
);
```

通过 lpRect 参数可以指定文字显示的范围，通过 uFormat 函数指定格式，如果需要向左对齐就设置为 DT_LEFT，如果居中对齐，则设置为 DT_CENTER 等。

1．输出的文字尺度和所占用的区域

实例中调用的 GetTextExtentPoint32 函数的功能是计算使用当前文本输出指定字符串后，输出的字符串的长和高（像素）。在使用 TextOut 等函数进行文字显示时，可以用于定义下次显示的文字的位置。

GetTextMetrics 函数用于获取文本的尺度信息。

在实例 13-2 中，字符串二的输出位置在字符串一的后面，字符串三输出的位置横向与字符串一对齐，纵向位于字符串二的下沿。

2．文字的对齐

SetTextAlign 函数可以指定 DC 的文本输出对齐方式，可以设置为 TA_TOP、TA_CENTER、TA_ LEFT、TA_UPDATECP 等。

13.4.5 DC图形模式设置

DC 图形模式，包括前景色、背景色的层次关系，轮换模式、多边形的填充模式等。与字符串输出相关的主要是背景(Backgroud)模式、绘制(Drawing)模式。

在进行文字输出时，所输出的文字包括除了具有文字外还具有背景，背景是文字周围的一块区域，图 13-4 中的白色部分就是文字的背景。

在显示文字时除了可以设置文字的颜色（前景色）外，还可以设置文字背景的颜色。在实例 13-2 中，所调用的 SetBkColor 函数的功能就是设置文字背景色。

```
SetBkColor(hdc,RGB(255,255,255));
```

在本实例中设置为白色。

而 DC 的背景模式决定了在绘制时，是使用文字的背景颜色去填充文字的背景区域，还是保留文字输出前原有的内容。

例如在实例 13-2 的代码中，将 SetBkMode 行去掉，编译后执行，程序输出结果如图 13-4

所示。 图 13-4 为文字设置背景模式

```
SetBkMode(hdc,TRANSPARENT);
```

与图 13.3 不同的是，图 13-4 的文字的背景使用了 SetBkColor 函数设置的白色进行了填充。

SetBkMode 函数将背景模式设置为了 TRANSPARENT。在输出操作后，原有的背景区域的显示内容不会被覆盖。

13.4.6 遍历字体

1．原理与关键 API

EnumFontFamilies 函数是实现列举 DC 中已经安装的字体的 API 函数，函数原型如下：

```
int EnumFontFamilies(
  HDC hdc,                          // DC 句柄
  LPCTSTR lpszFamily,               // 字体族
  FONTENUMPROC lpEnumFontFamProc,   // 回调
  LPARAM lParam                     // 附加数据
);
```

EnumFontFamilies 函数指定了一个回调函数，每列举一个字体，回调函数就会被调用一次，字体的 LOGFONT 结构、字体的类型会作为参数输出给回调函数。

回调函数的接口形式如下：

```
int CALLBACK EnumFontFamProc(
  ENUMLOGFONT *lpelf,       // 逻辑字体数据
  NEWTEXTMETRIC *,potm,     // 物理字体数据
  DWORD FontType,           // 字体类型
  LPARAM lParam             // application-defined data
);
```

lpelf 和 lpntm 参数分别是逻辑字体和物理字体数据。FontType 是字体类型，lParam 参数是 EnumFontFamilies 函数通过 lParam 参数传递给本函数的数据。

2．实例 13-3 遍历系统中已经安装的字体

本实例会依次列举桌面 DC 可用的字体，显示字体类型、字体名等信息，并根据不同的类型进行统计。

```
/* ************************************
* font.c 遍历字体
**************************************/
/* 头文件 */
#include <Windows.h>
#include <stdio.h>
/* 函数声明 */
BOOL CALLBACK EnumFamCallBack(LPLOGFONT , LPNEWTEXTMETRIC , DWORD , LPVOID ) ;
DWORD ListFont();
// main
int main()
{
    // 桌面 DC
```

```
    ListFont(NULL);
}
/*************************************
* DWORD ListFont(HWND hwnd)
* 功能：列举指定窗口的 DC 的所具有的字体
*************************************/
DWORD ListFont(HWND hwnd)
{
    // 获得 DC
    HDC hdc = GetDC(hwnd);
    // 用于计数
    int aFontCount[] = { 0, 0, 0 };
    // 调用 EnumFontFamilies，开始列举

  EnumFontFamilies(hdc, (
      LPCTSTR) NULL, // 列举所有类型
    (FONTENUMPROC) EnumFamCallBack,// 回调函数为 EnumFamCallBack
      (LPARAM) aFontCount); //传递给 EnumFamCallBack 的参数

    // 显示统计信息
    printf("Number of raster fonts: %d\n",aFontCount[0]);
    printf("Number of vector fonts: %d\n",aFontCount[1]);
    printf("Number of TrueType fonts: %d\n",aFontCount[2]);
    // 返回
    return 0;
}

/*************************************
* EnumFamCallBack
* 功能：字体列举回调函数次
*       每列举一个字体会被调用一次
* 参数：lplf，字体的 LOGFONT 结构
*      lpntm，字符的尺度属性
*      FontType，字体类型
*      lParam，通过 EnumFontFamilies 输入给本函数的参数，这里用于计数
*************************************/
BOOL CALLBACK EnumFamCallBack(
                                LPLOGFONT lplf,
                                LPNEWTEXTMETRIC lpntm,
                                DWORD FontType,
                                LPVOID aFontCount)
{
    // 获得参数
  PINT aiFontCount = (PINT) aFontCount;
    // 判断字体类型，输出类型信息，并根据类型进行计数
  if (FontType & RASTER_FONTTYPE)
    {
       printf(" RASTER TYPE \t");
     aiFontCount[0]++;
    }
  else if (FontType & TRUETYPE_FONTTYPE)
    {
       printf(" TRUETYPE \t");
     aiFontCount[2]++;
    }
  else
    {
       printf(" VECTOR TYPE \t");
     aiFontCount[1]++;
    }
    // 显示字体信息
    printf("%s\tItalic = %d\n",lplf->lfFaceName,lplf->lfItalic);
  // 返回
    if (aiFontCount[0] || aiFontCount[1] || aiFontCount[2])
      return TRUE;
  else
     return FALSE;
}
```

程序运行后，会打印输出所列举的每个字体的信息。

3．实例运行结果

以下为打印输出的运行结果的节选。

```
RASTER TYPE  System    Italic = 0
RASTER TYPE  @System   Italic = 0
RASTER TYPE  Terminal  Italic = 0
RASTER TYPE  @Terminal Italic = 0
RASTER TYPE  Fixedsys  Italic = 0
RASTER TYPE  @Fixedsys Italic = 0
VECTOR TYPE  Modern    Italic = 0
VECTOR TYPE  Roman     Italic = 0
VECTOR TYPE  Script    Italic = 0
RASTER TYPE  Courier   Italic = 0
RASTER TYPE  MS Serif  Italic = 0
RASTER TYPE  MS Sans Serif Italic = 0
RASTER TYPE  Small Fonts   Italic = 0
TRUETYPE     Marlett   Italic = 0
TRUETYPE     Arial     Italic = 0
…
TRUETYPE     新宋体     Italic = 0
…
TRUETYPE Verdana  Italic = 0
TRUETYPE Webdings Italic = 0
Number of raster fonts: 10
Number of vector fonts: 3
Number of TrueType fonts: 156
```

程序打印出每个字体的类型、名称。

13.4.7　为系统安装、删除字体文件

AddFontResource、AddFontResourceEx 等函数可以将字体资源（.fon、.fnt、.ttf 等格式的文件）导入到系统字体表，导入之后程序便可以使用字体。

RemoveFontResource 删除字体资源。

AddFontResource 函数原型如下：

```
int AddFontResource(
  LPCTSTR lpszFilename   // 字体文件名
);
```

RemoveFontResource 函数原型如下：

```
BOOL RemoveFontResource(
  LPCTSTR lpFileName   // 字体文件名
);
```

当系统的字体资源发生改变时，系统中的窗口会收到 WM~FONTCHANGE 消息。

13.5　绘制线条

本节讲解如何在界面上绘制出各种形状的线条，包括直线、任意曲线、弧线等。

13.5.1　选择画笔对象

前面已经介绍，在绘制线条前需将线条所使用的画笔对象选择入 DC。画笔对象决定了所绘制线条的颜色、宽度、样式。具体内容参见实例 13-1 和 13.3.1 小节。

13.5.2 直线

绘制直线使用 LineTo 函数。移动线条绘制的起点使用 MoveToEx 函数，实例 13-1；介绍了直线的绘制方法。

13.5.3 绘制任意曲线

原理与关键 API

绘制曲线的 API 函数有 PolyBezier、Polyline、PolylineTo、PolyPolyline 等，其功能基本类似。

PolyBezier 和 Polyline 的函数原型如下：

```
BOOL PolyBezier(
  HDC hdc,              // 设备句柄
  CONST POINT* lppt,    // 点的数值
  DWORD cPoints         // 点的个数
);
BOOL Polyline(
  HDC hdc,              // 设备句柄
  CONST POINT *lppt,    // 点数组，将本数组中的点使用直线依次连接构成曲线
  int cPoints           // 点数组中点的个数
);
```

在使用上述函数绘制曲线时，都需要指定一组点 cPOINT)，而 PolyBezier 和 Polyline 等函数的功能就是使用线条将这些点顺序连接起来。

13.5.4 跟踪鼠标轨迹

鼠标轨迹的跟踪依赖于鼠标捕获器和鼠标消息的过滤，如例 13-4 所示。

1．实例 13-4 跟踪鼠标，画出鼠标路径

本实例实现了对鼠标位置的捕获，并在鼠标所在窗口中所划过的路径作出曲线。

实例的主要功能在 GetGuideLine 函数和 ShowGuide 函数中实现。

GetGuideLine 函数在一开始调用 SetCapture API 函数设置了一个鼠标捕获器，在设置了鼠标捕获器后，鼠标在窗口上和移动或按住鼠标键同时移动时，窗口都会收到消息。之后 GetGuideLine 函数过

滤了鼠标消息，当每次收到消息时，都将鼠标所在的位置(POINT)记录下来，依次保存在数组中。

在设置点数组中，GetGuideLine 进行了多项判断，其中对是否是第一个消息的判断很重要。

函数在收到第一个消息将线条绘制和起始点使用 MoveToEx 函数移动到了当前鼠标的位置，这一步是必须的，否则绘制曲线的起点会出现错误。

需要注意的是 GetGuideLine 函数绘制曲线仍然使用的是 LineTo 函数。在很多情况下，计算机都是将曲线处理为很多的小直线的连接。使用直线将点数组中的每两个相邻连接起来与使用 PloyLine API 函数绘制曲线效果是一样的。GetGuideLine 函数在鼠标移动一点就绘制一点，连续绘制，形成曲线的效果（如图 13-5 所示）。

ShowGuide 函数是另一个比较重要的函数，它在 GetGuideLine 函数已经完了对鼠标的跟踪，但要需要重新绘制曲线的情况下被调用（比如改变窗口大小后，需要重新绘制窗口窗

户区时，窗口收到 WM__ PAINT 消息时）。ShowGuide 函数调用了 PloyLine 函数，将 GetGuideLine 函数所记录点数组做为参数传递给了 PloyLine 函数。GetGuideLine 函数和 ShowGuide 函数所绘制的曲线都是一致的。

此外，本实例的窗口消息处理函数也很重要，尤其是在处理 WM_LBUTTONDOWN 时，在消息处理函数收到了 WM_LBUTTONDOWN 消息后，调用了 GetGuideLine 函数，开始鼠标跟踪和记录。，需要注意，WM_LBUTTONDOWN 消息处理过程中，有一个 bOn 标记，初始值为 TRUE。这样在第一次在窗口上单击鼠标左键时，程序将客户区使用白色填充，为 DC 选择了蓝色的画笔对象。之后开始记录，在记录完成后，将 bOn 标志取反，这样下次再进行鼠标跟踪时所绘制的线条是红色的。如此循环，每画一蓝一红两条曲线就清空屏幕并释放资源。

```
/* ************************************
* curves.c 遍历字体
**************************************/
/* 头文件 */
#include <windows.h>
/* 预定义 */
#define MAXGUIDESEGMENTS 1000
#define MyAlloc(dwSize) HeapAlloc(GetProcessHeap(),0,dwSize)
#define MyFree(lpMem) HeapFree(GetProcessHeap(),0,lpMem);
/* 函数声明 */
BOOL GetGuideLine(HWND, LPPOINT*, LPDWORD);
BOOL ShowGuide(HDC, LPPOINT, DWORD);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
DWORD GetLastErrorBox(HWND hWnd, LPSTR lpTitle);
/* 全局变量*/
HINSTANCE hInst;
LPSTR szAppName = "Curves";
LPSTR szTitle = "Curves Application";

/************************************
* WinMain
* 功能：创建窗口
************************************/
int WINAPI WinMain(
                    HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow)
{
     MSG msg;
     HWND hWnd;
     WNDCLASS wc;

     hInst = hInstance;
     // 注册窗口类
     wc.style = CS_OWNDC;
     wc.lpfnWndProc = (WNDPROC)WndProc;
     wc.cbClsExtra = 0;
     wc.cbWndExtra = 0;
     wc.hInstance = hInstance;
     wc.hIcon = NULL;
     wc.hCursor = LoadCursor(NULL, IDC_ARROW);
     wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
     wc.lpszMenuName = NULL;
     wc.lpszClassName = szAppName;

     if (!RegisterClass(&wc))
```

```
    {
        GetLastErrorBox(NULL, "Error in RegisterClass");
        return (FALSE);
    }
    // 创建窗口
    hWnd = CreateWindow(
        szAppName,
        szTitle,
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
        NULL,
        NULL,
        hInstance,
        NULL
        );

    if (!hWnd)
    {
        GetLastErrorBox(hWnd, "Error in CreateWindow");
        return (FALSE);
    }
    // 显示更新
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    // 消息循环
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return (int)(msg.wParam);
    UNREFERENCED_PARAMETER(lpCmdLine);
}

/*************************************
* WndProc
* 功能：窗口消息处理函数
*************************************/
LRESULT CALLBACK WndProc(
                        HWND hWnd,
                        UINT message,
                        WPARAM uParam,
                        LPARAM lParam)
{
    static BOOL bOn = TRUE;
    static LPPOINT lpBlue = NULL;
    static LPPOINT lpRed = NULL;
    static DWORD dwBlue = 0;
    static DWORD dwRed = 0;
    static BOOL bOutlineOnly = FALSE;
    static BOOL bShowGuides = TRUE;
    static HPEN hpenBlue, hpenRed;
    switch (message)
    {
    case WM_CREATE:
        {
            //  获取 DC
            HDC hDC = GetDC(hWnd);
            // 创建画笔对象
            hpenBlue = CreatePen(PS_SOLID, 1, RGB(0,0,255));
            hpenRed = CreatePen(PS_SOLID, 1, RGB(255,0,0));
        }
        GetLastErrorBox(hWnd, "Error in WM_CREATE");
```

```
        break;
    case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hDC = BeginPaint(hWnd, &ps);
            RECT rect;
            // 将客户区绘制为白色
            GetClientRect(hWnd, &rect);
            PatBlt(hDC, 0, 0, rect.right, rect.bottom, WHITENESS);
            // 既存在红线，又存在蓝线
            if (dwBlue && dwRed)
            {
                // 显示蓝线
                if (dwBlue && bShowGuides)
                {
                    SelectObject(hDC, hpenBlue);
                    ShowGuide(hDC, lpBlue, dwBlue);
                    SelectObject(hDC, GetStockObject(BLACK_PEN));
                }
                // 显示红线
                if (dwRed && bShowGuides)
                {
                    SelectObject(hDC, hpenRed);
                    ShowGuide(hDC, lpRed, dwRed);
                    SelectObject(hDC, GetStockObject(BLACK_PEN));
                }
            }
            EndPaint(hWnd, &ps);
        }
        break;
    case WM_LBUTTONDOWN:
        {
            HDC hDC = GetDC(hWnd);
            RECT rect;

            if (bOn)// 消除并画蓝线
            {
                // 将客户区填充为白色
                GetClientRect(hWnd, &rect);
                PatBlt(hDC, 0, 0, rect.right, rect.bottom, WHITENESS);
                // 释放资源
                if (lpBlue)
                    MyFree(lpBlue);
                if (lpRed)
                    MyFree(lpRed);
                dwRed = 0;
                dwBlue = 0;
                // 开始跟踪鼠标移动，绘制蓝线
                SelectObject(hDC, hpenBlue);
                GetGuideLine(hWnd, &lpBlue, &dwBlue);
            }
            else//画红线
            {
                // 开始跟踪鼠标移动，绘制红线
                SelectObject(hDC, hpenRed);
                GetGuideLine(hWnd, &lpRed, &dwRed);
            }
            // 鼠标左键放开，恢复画笔对象
            SelectObject(hDC, GetStockObject(BLACK_PEN));
            // 取反，在红色和蓝色间交替
            bOn = !bOn;
        }
        GetLastErrorBox(hWnd, "Error in WM_LBUTTONDOWN");
```

```
            break;
        case WM_DESTROY:
            // 释放资源，退出
            if (lpBlue) MyFree(lpBlue);
            if (lpRed) MyFree(lpRed);
            PostQuitMessage(0);
            break;
        default:
            return (DefWindowProc(hWnd, message, uParam, lParam));
        }
        return (0);
    }
    /************************************
    * BOOL GetGuideLine(HWND hWnd, LPPOINT *lpPoint, LPDWORD lpdwNumPts)
    * 功能：跟踪鼠标，绘制鼠标轨迹
    * 参数：hWnd，窗口
    *       lpPoint，用于保存点的数组，向调用函数返回
    *       lpdwNumPts，返回的数组的大小
    ************************************/
    BOOL GetGuideLine(HWND hWnd, LPPOINT *lpPoint, LPDWORD lpdwNumPts)
    {
        MSG msg;
        HDC hDC = GetDC(hWnd);
        BOOL bFirstTime = TRUE;
        DWORD dwPos = 0;
        RECT rect;

        SetCapture(hWnd);       // 设置鼠标捕获器
        GetClientRect(hWnd, &rect);
        // 为点数组分配空间
        *lpPoint = (LPPOINT)MyAlloc(MAXGUIDESEGMENTS * sizeof(POINT));
        for (;;)
        {
            // 过滤所有鼠标消息
            WaitMessage();
            if (PeekMessage(&msg,NULL,WM_MOUSEFIRST,WM_MOUSELAST,PM_REMOVE))
            {
                // 判断是否在客户区中
                if ((LOWORD(msg.lParam) < rect.right) && (HIWORD(msg.lParam) < rect.bottom))
                {
                    // 是否第一次收到消息
                    if (bFirstTime)
                    {
                        bFirstTime = FALSE;
                        // 如果是第一次，将画笔的起点移动到鼠标单击的位置
                        MoveToEx(hDC, LOWORD(msg.lParam), HIWORD(msg.lParam), NULL);
                    }
                    // 是否达到了最大点数
                    if (dwPos < MAXGUIDESEGMENTS)
                    {
                        // 鼠标的移动会产生鼠标消息，每收到一次消息保存一个点
                        (*lpPoint)[dwPos].x = LOWORD(msg.lParam);
                        (*lpPoint)[dwPos].y = HIWORD(msg.lParam);
                        // 绘制到鼠标所在的点
                        LineTo(hDC, (*lpPoint)[dwPos].x, (*lpPoint)[dwPos].y);
                        dwPos++;
                    }
                }
                if (msg.message == WM_LBUTTONUP)
                    break;
            }
            else
                continue;
```

```
        }
        *lpdwNumPts = dwPos;
        ReleaseDC(hWnd, hDC);
        ReleaseCapture();
        DeleteDC( hDC );
        return TRUE;
}

/************************************
* BOOL ShowGuide(HDC hDC, LPPOINT lpPoints, DWORD dwNumPts)
* 功能：根据保存的点的数组重绘曲线
* 参数：hWnd，窗口
*       lpPoint，保存的点的数组
*       dwNumPts，数组的大小
************************************/
BOOL ShowGuide(HDC hDC, LPPOINT lpPoints, DWORD dwNumPts)
{
        Polyline(hDC, lpPoints, dwNumPts);
        return TRUE;
}
// 显示错误信息
DWORD GetLastErrorBox(HWND hWnd, LPSTR lpTitle)
{
        LPVOID lpv;
        DWORD dwRv;
        if (GetLastError() == 0) return 0;
        dwRv = FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
                FORMAT_MESSAGE_FROM_SYSTEM,
                NULL,
                GetLastError(),
                MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US),
                (LPVOID)&lpv,
                0,
                NULL);
        MessageBox(hWnd, lpv, lpTitle, MB_OK);
        if(dwRv)
                LocalFree(lpv);
        SetLastError(0);
        return dwRv;
}
```

实例中的 PatBlt 函数的功能是实例当前画刷对指定 RECT 范围进行填充。

```
BOOL PatBlt(
  HDC hdc,     // DC 句柄
  int nXLeft, // 矩形左上角 x 坐标
  int nYLeft, // 矩形左上角 y 坐标
  int nWidth, // 矩形宽度
  int nHeight,// 矩形高度
  DWORD dwRop // 光栅操作代码
);
```

2．实例运行结果

运行程序，如图 13-5 所示，在窗口客户区单击任务位置，单击后不要松开鼠标，然后移动，就会绘制出一条蓝色的曲线，曲线随着鼠标和移动而延长。

放开鼠标，再次点击然后再开始移动，会绘制出一个红色的曲线。如果再重复，那么窗口会被清空，然后重新开始绘制蓝色曲线。

对窗口进行最小化、移动、大小改变等操作时窗口中的曲线仍然还在。

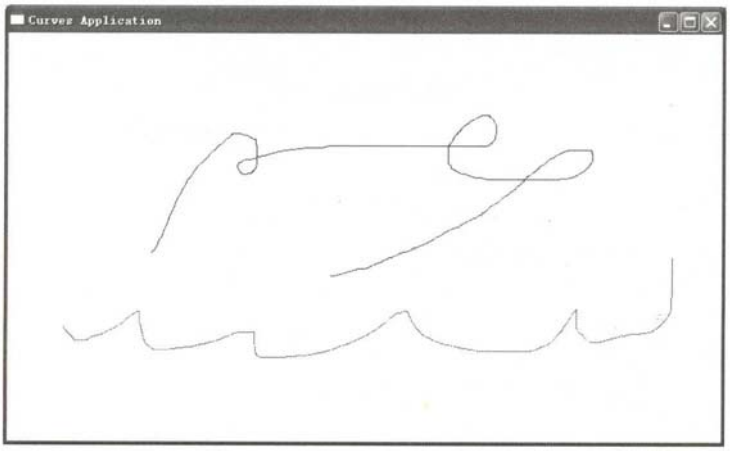

图 13-5　绘制曲线

13.5.5　弧线

GDI 提供 ArcTo 函数用于绘制椭圆弧线，函数原型如下：

```
BOOL ArcTo(
  HDC hdc,                // 设备句柄
  int nLeftRect,          // 外切矩形左上角 x 坐标
  int nTopRect,           // 外切矩形左上角 y 坐标
  int nRightRect,         // 外切矩形右下角 x 坐标
  int nBottomRect,        // 外切矩形右下角 y 坐标
  int nXRadial1,          // 第一条射线上点的 x 坐标
  int nYRadial1,          // 第一条射线上点的 y 坐标
  int nXRadial2,          // 第二条射线上点的 x 坐标
  int nYRadial2           // 第二条射线上点的 y 坐标
);
```

函数的前 4 个参数使用外切矩形定义了一个椭圆，4 个参数分别是外切矩形的左上点位置及长宽。

之后的 4 个参数相当于从椭圆上切出一部分，为椭圆弧。4 个参数分别是椭圆半径两个端点的横纵坐标（椭圆中心到坐标点的射线）。

所取的弧是按照 DC 当前弧方向从半径一到半径二的范围内的弧。设置 DC 的当前弧方向使用 API 函数 SetArcDirection。

```
int SetArcDirection(
  HDC hdc,                // handle to device context  DC 句柄
  int ArcDirection        // new arc direction 弧方向
);
```

参数 ArcDirection 可以是 AD_COUNTERCLOCKWISE，也可以是 AD_CLOCKWISE。

13.6　绘制图形

GDI 所能绘制形状有很多种，可区分为标准形状和非标准形状。标准形状使用几个简单、确定的属性就可以确定的形状，比如矩形（左上角点位置和长度）、椭圆（使用外切矩形）、圆角矩形（一个矩形加圆角半径）、弓形、扇形。

也有不标准的形状，所有不标准的形状都可以使用多边形(Polygon)来定义。

定义多边形也需要使用一个点数组。将点数组中的点依其在数组中的顺序连接起来，就

是一个多边形。

表 13.5 列举了进行图形绘制的 API 函数。

表 13.5　　绘制图形的 API

图 形 对 象	绘制的 API
矩形	Rectangle
圆角矩形	RoundRect
圆、椭圆	Ellipse
椭圆切（弓形、Chords）	Chord
扇形（Pie）	Pie
多边形	Polygon、PolyPolygon

13.6.1　填充颜色与边缘勾勒

使用表 13.5 所给的函数绘制的图形都有边。在绘制时，图形内部使用 DC 的当前画刷对象进行填充，图形的边使用 DC 的当前画笔对象进行勾勒。

13.6.2　绘制矩形、椭圆、圆角矩形

(1) Rectangle。

绘制矩形使用 API 函数 Rectangle，函数原型如下：

```
BOOL Rectangle(
  HDC hdc,              // DC 句柄
  int nLeftRect,        // 矩形左上角 x 坐标
  int nTopRect,         // 矩形左上角 y 坐标
  int nRightRect,       // 矩形右下角 x 坐标
  int nBottomRect       // 矩形右下角 y 坐标
);
```

从第二个参数开始分别是需要绘制的椭圆的位置（左上角，相对与 DC）和大小。

(2) Ellipse。

绘制椭圆（圆）使用 API 函数 Ellipse，函数原型如下：

```
BOOL Ellipse(
  HDC hdc,          // DC 句柄
  int nLeftRect,    // 外切矩形左上角 x 坐标
  int nTopRect,     // 外切矩形左上角 y 坐标
  int nRightRect,   // 外切矩形右下角 x 坐标
  int nBottomRect   // 外切矩形右下角 y 坐标
);
```

从第 2 个参数开始分别是椭圆的外切矩形的位置（左上角，相对于 DC）和大小。如果需要画圆，那么将矩形的长宽设置为相等。

(3) RoundRect。

绘制圆角矩形的 API 函数是 RoundRect，函数原型如下：

```
BOOL RoundRect(
  HDC hdc,          // DC 句柄
  int nLeftRect,    // 外切矩形左上角 x 坐标
  int nTopRect,     // 外切矩形左上角 y 坐标
  int nRightRect,   // 外切矩形右下角 x 坐标
  int nBottomRect,  // 外切矩形右下角 y 坐标
  int nWidth,       // 椭圆的长轴
  int nHeight       // 椭圆的短轴
);
```

其中第二个开始的四个参数还是圆角矩形所对应的矩形的位置和大小。最后两个参数设置了用于绘制圆角的椭圆的宽和高。

以上函数在绘制时，图形内部使用当前画刷对象进行填充，图形边沿使用当前画笔绘制。

1．实例 13-5 绘制矩形、椭圆和圆角矩形

在本实例中，用户通过菜单设置需要绘制的图形，可以选择矩形(ID_SHAPE_RECTANGLE)、椭圆(ID_SHAPE ELLIPSE)和圆角矩形(D_SHAPE- ROUNDRECT)，选择后，程序会设置全局变量。

这些全局变量在绘制图形时使用。

之后，用户可以用窗口的客户区中按住鼠标左键，然后移动鼠标。随着鼠标的移动，会在界面上绘制出矩形。在鼠标左键松开后，会在鼠标移动起点和终点间的范围内，根据全局变量绘制出用户之前选择的图形。

```
/* ************************************
* Shapes.c 绘制图形
**************************************/
/* 头文件 */
#include <Windows.h>
#include "resource.h"
/* 全局变量 */
HDC hdcCompat;      // handle to DC for bitmap
POINT pt;           // 鼠标位置
RECT rcTarget;      // 矩形外沿
RECT rcClient;      // 客户区 RECT
// 绘制的状态
BOOL fSizeEllipse;
BOOL fDrawEllipse;
BOOL fDrawRectangle;
BOOL fSizeRectangle;
BOOL fSizeRoundRect;
BOOL fDrawRoundRect;
// 圆角椭圆宽和高
int nEllipseWidth;
int nEllipseHeight;

BOOL CALLBACK WndProc(HWND hwnd, UINT uMsg, WPARAM wParam,
                      LPARAM lParam)
{
     HDC hdc;          // DC 句柄
     PAINTSTRUCT ps;
     POINT ptClientUL; // 客户区左上角
     POINT ptClientLR; // 客户区右下角

     switch (uMsg)
     {
     case WM_COMMAND:
          switch (wParam)      // 菜单输入，设置需要绘制的图形
          {
          case ID_SHAPE_ELLIPSE:
               fSizeEllipse = TRUE;
               break;
          case ID_SHAPE_RECTANGLE:
               fSizeRectangle = TRUE;
               break;
          case ID_SHAPE_ROUNDRECT:
               fSizeRoundRect = TRUE;
               break;
          default:
               return DefWindowProc(hwnd, uMsg, wParam,
```

```
                lParam);
        }
        break;
    case WM_CREATE:
        // 初始化圆角的长和宽
        nEllipseWidth = 20;
        nEllipseHeight = 20;
        return 0;
    case WM_PAINT:// 绘制
        BeginPaint(hwnd, &ps);
        // 选择画刷
        SelectObject(ps.hdc, GetStockObject(GRAY_BRUSH));
        if (fDrawEllipse) // 如果有椭圆
        {
            // 调用 Ellipse
            Ellipse(ps.hdc, rcTarget.left, rcTarget.top,
                rcTarget.right, rcTarget.bottom);
            fDrawEllipse = FALSE;
            rcTarget.left = rcTarget.right = 0;
            rcTarget.top = rcTarget.bottom = 0;
        }
        if (fDrawRectangle) // 如果有矩形
        {
            // 调用 Rectangle
            Rectangle(ps.hdc, rcTarget.left, rcTarget.top,
                rcTarget.right, rcTarget.bottom);
            fDrawRectangle = FALSE;
            rcTarget.left = rcTarget.right = 0;
            rcTarget.top = rcTarget.bottom = 0;
        }
        if (fDrawRoundRect) // 如果有圆角矩形
        {
            // 调用 RoundRect
            RoundRect(ps.hdc, rcTarget.left, rcTarget.top,
                rcTarget.right, rcTarget.bottom,
                nEllipseWidth, nEllipseHeight);
            fDrawRectangle = FALSE;
            rcTarget.left = rcTarget.right = 0;
            rcTarget.top = rcTarget.bottom = 0;
        }
        EndPaint(hwnd, &ps);
        break;
    case WM_SIZE: // 大小改变
        GetClientRect(hwnd, &rcClient);
        ptClientUL.x = rcClient.left;
        ptClientUL.y = rcClient.top;
        ptClientLR.x = rcClient.right;
        ptClientLR.y = rcClient.bottom;
        ClientToScreen(hwnd, &ptClientUL);
        ClientToScreen(hwnd, &ptClientLR);
        // 设置 RECT 的大小和位置
        SetRect(&rcClient, ptClientUL.x, ptClientUL.y,
            ptClientLR.x, ptClientLR.y);
        return 0;
    case WM_LBUTTONDOWN:
        // 使用鼠标定义一个矩形范围
        ClipCursor(&rcClient);
        // 鼠标的位置
        pt.x = (LONG) LOWORD(lParam);
        pt.y = (LONG) HIWORD(lParam);

        if (fDrawEllipse)
            fSizeEllipse = TRUE;
```

```
        return 0;
    case WM_MOUSEMOVE:
        if ((wParam && MK_LBUTTON)
            && (fSizeEllipse || fSizeRectangle
            || fSizeRoundRect))
        {
            hdc = GetDC(hwnd);
            // 目标 RECT 是否空,
            if (!IsRectEmpty(&rcTarget))
            {
                Rectangle(hdc, rcTarget.left, rcTarget.top,
                    rcTarget.right, rcTarget.bottom);
            }
            // 根据鼠标移动和情况,得到需要绘制的范围
            // 根据鼠标移动和终点和起点的相对位置,起点和终点所代表的矩形的角不同
            if ((pt.x < (LONG) LOWORD(lParam)) &&
                (pt.y > (LONG) HIWORD(lParam)))
            {
                SetRect(&rcTarget, pt.x, HIWORD(lParam),
                    LOWORD(lParam), pt.y);
            }
            else if ((pt.x > (LONG) LOWORD(lParam)) &&
                (pt.y > (LONG) HIWORD(lParam)))
            {
                SetRect(&rcTarget, LOWORD(lParam),
                    HIWORD(lParam), pt.x, pt.y);
            }
            else if ((pt.x > (LONG) LOWORD(lParam)) &&
                (pt.y < (LONG) HIWORD(lParam)))
            {
                SetRect(&rcTarget, LOWORD(lParam), pt.y,
                    pt.x, HIWORD(lParam));
            }
            else
            {
                SetRect(&rcTarget, pt.x, pt.y, LOWORD(lParam),
                    HIWORD(lParam));
            }
            // 在鼠标移动的过程中,绘制一个矩形
            Rectangle(hdc, rcTarget.left, rcTarget.top,
                rcTarget.right, rcTarget.bottom);
            DeleteDC( hdc);
        }
        return 0;
    case WM_LBUTTONUP:
        // 设置
        if (fSizeEllipse)
        {
            fSizeEllipse = FALSE;
            fDrawEllipse = TRUE;
        }
        if (fSizeRectangle)
        {
            fSizeRectangle = FALSE;
            fDrawRectangle = TRUE;
        }
        if (fSizeRoundRect)
        {
            fSizeRoundRect = FALSE;
            fDrawRoundRect = TRUE;
        }
        // 是否有绘制的图形
        if (fDrawEllipse || fDrawRectangle || fDrawRoundRect)
```

```
            {
                InvalidateRect(hwnd, &rcTarget, TRUE);
                UpdateWindow(hwnd); // WM_PAINT
            }
            // 释放鼠标
            ClipCursor((LPRECT) NULL);
            return 0;
        case WM_DESTROY:
            DeleteDC(hdcCompat);
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd, uMsg, wParam, lParam);
        }
        return (LRESULT) NULL;
}
int WINAPI WinMain(
                        HINSTANCE hInstance,
                        HINSTANCE hPrevInstance,
                        LPSTR lpCmdLine,
                        int nCmdShow)
{
        MSG msg;
        HWND hWnd;
        WNDCLASS wc;
        wc.style = CS_OWNDC;
        wc.lpfnWndProc = (WNDPROC)WndProc;
        wc.cbClsExtra = 0;
        wc.cbWndExtra = 0;
        wc.hInstance = hInstance;
        wc.hIcon = NULL;
        wc.hCursor = LoadCursor(NULL, IDC_ARROW);
        wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
        wc.lpszMenuName = MAKEINTRESOURCE(IDR_MENU_SHAPE);
        wc.lpszClassName = "shape";
        if (!RegisterClass(&wc))
            return (FALSE);
        hWnd = CreateWindow(
            "shape",
            "shape",
            WS_OVERLAPPEDWINDOW,
            CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
            NULL,
            NULL,
            hInstance,
            NULL
            );
        if (!hWnd)
            return (FALSE);
        ShowWindow(hWnd, nCmdShow);
        UpdateWindow(hWnd);
        while (GetMessage(&msg, NULL, 0, 0))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        return (int)(msg.wParam);
        UNREFERENCED_PARAMETER(lpCmdLine);
}
```

2．运行结果

实例程序进行效果如图 13-6 所示。

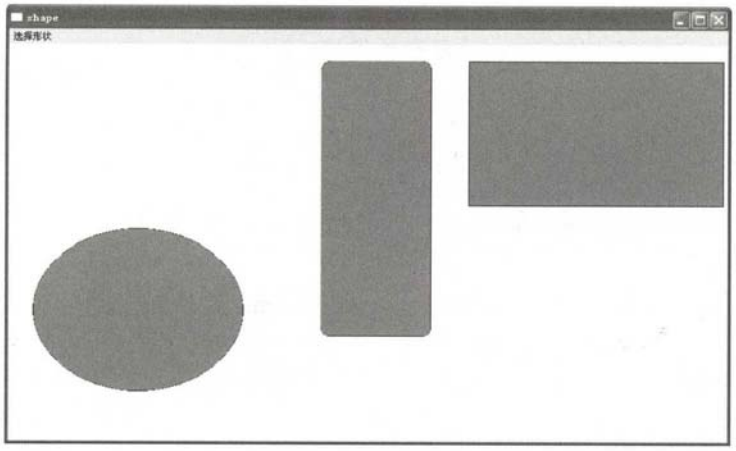

图 13-6　绘制图形的输出结果

13.6.3　椭圆弓形和椭圆扇形

绘制椭圆弓形和椭圆扇形分别使用 API 函数 Chord 和 Pie，函数原型如下。

(1) Chord。

```
BOOL Chord(
  HDC hdc,          // DC 句柄
  int nLeftRect,    // 外切矩形左上角 x 坐标
  int nTopRect,     // 外切矩形左上角 y 坐标
  int nRightRect,   // 外切矩形右下角 x 坐标
  int nBottomRect,  // 外切矩形右下角 y 坐标
  int nXRadial1,    // 第一条射线端点的 x 坐标
  int nYRadial1,    // 第一条射线端点的 y 坐标
  int nXRadial2,    // 第二条射线端点的 x 坐标
  int nYRadial2     // 第二条射线端点的 y 坐标
);
```

(2) Pie。

```
BOOL Pie(
  HDC hdc,          // DC 句柄
  int nLeftRect,    // 外切矩形左上角 x 坐标
  int nTopRect,     // 外切矩形左上角 y 坐标
  int nRightRect,   // 外切矩形右下角 x 坐标
  int nBottomRect,  // 外切矩形右下角 y 坐标
  int nXRadial1,    // 第一条射线端点的 x 坐标
  int nYRadial1,    // 第一条射线端点的 y 坐标
  int nXRadial2,    // 第二条射线端点的 x 坐标
  int nYRadial2     // 第二条射线端点的 y 坐标
);
```

其各参数的意义、弧所取方向的确定参考 13.5.5 小节对弧的定义方式。

以上函数在绘制时，图形内部使用当前画刷对象进行填充，图形边沿使用当前画笔绘制。

13.6.4　多边形

1. 多边形的表示

多边形使用其顶点的数组来定义。在绘制多边形时，需给出一组点，多边形是将这些点按照点在数组中的顺序连接，并将最后一个点和第一个点连接起来得到图形。这种机制实际可以定义任意封闭图形。

(1) Polygon。

绘制多边形的 API 函数是 Polygon，函数原型如下：

```
BOOL Polygon(
  HDC hdc,                          // DC 句柄
  CONST POINT *lpPoints,            // 多边形顶点数组
  int nCount                        // 多边形顶点数组大小
);
```

参数 lpPoints 指向多边形顶点数组。nCount 参数指明了数组的大小。

(2) PolyPolygon。

此外 PolyPolygon 函数绘制了一组多边形。lpPolyCounts 参数是一组整数，每一个整数说明了一个多边形的顶点数，顶点使用 lpPoirits 数组。

```
BOOL PolyPolygon(
  HDC hdc,                          // DC 句柄
  CONST POINT *lpPoints,            // 顶点数组
  CONST INT *lpPolyCounts,          // 数组中点的个数
  int nCount                        // 多边形个数
);
```

2．多边形的填充方式

多边形可能存在简单多边形（各边互不相交）和复杂多边形（存在边相交的情况）。复杂多边形会将多边形划分为很多块。而多边形的填充方式说明了在绘制时，为这些块填充颜色的模式。

ALTERNATE 和 WINDING 是可供选择的两种模式。如图 13-7 五角星所示，左侧为 ALTERNATE 模式，右侧为 WINDING 模式。

图 13-7　不同的多边形填充方式

13.6.5　REGT结构及对RECT的操作

与形状相关的还有一种数据是 RECT。RECT 用于表示界面上的一个矩形范围。在很多情况下都会应用到 RECT，比如实例 13-5 中就多次用了 RECT。

```
typedef struct _RECT {
  LONG left;
  LONG top;
  LONG right;
  LONG bottom;
} RECT, *PRECT;
```

4 个成员分别表示左、上、右、下 4 条边的位置。

对 RECT 的操作包括如下几种。

1．获取设置位置、大小等

GetClientRect 函数的功能是获取窗口客户区的 RECT。

```
BOOL GetClientRect(
   HWND hWnd,
   LPRECT lpRect
);
```

通过参数 lpRect 返回客户区的 RECT。

此外还有 GetWindowRect 函数用于获取整个窗口的 RECT。

SetRect 函数的功能是设置一个 RECT 的坐标，原型如下：

```
BOOL SetRect(
```

```
  LPRECT lprc,      // 矩形
  int xLeft,        // 左边
  int yTop,         // 顶边
  int xRight,       // 右边
  int yBottom       // 底边
);
```

CopyRect 函数是将一个 RECT 的坐标复制到另外一个 RECT。

```
BOOL CopyRect(
  LPRECT lprcDst,       // 目标矩形
  CONST RECT* lprcSrc   // 源矩形
);
```

2．填充、边沿勾勒、反转

可以将一个 RECT 在屏幕上显示出来，显出的方式是有内部填充、边沿勾勒，也可以对一个 RECT 进行反转操作。反转操作会将现有 RECT 现有的填充颜色取反。FillRect、FrameRect、InvertRect 函数分别实现这 3 个功能，函数原型如下：

(1). FillRect。

```
int FillRect(
  HDC hDC,                  // DC 句柄
  CONST RECT *lprc,         // 矩形
  HBRUSH hbr                // 画刷句柄
);
```

(2) FrameRect。

```
int FrameRect(
  HDC hDC,              // DC 句柄
  CONST RECT *lprc,     // 矩形
  HBRUSH hbr            // 画刷句柄
);
```

(3) InvertRect。

```
BOOL InvertRect(
  HDC hDC,              // DC 句柄
  CONST RECT *lprc      // 矩形
);
```

比较特殊的是，在进行填充和勾勒时，直接通过参数指定画刷，而不是绘制函数所采用的 DC 当前画刷。这几个函数的参数所要求的 RECT 的坐标都是相对位置的。

3．点(POINT)

另一个与 RECT 相关的概念是点，点是坐标位置。可以相对于屏幕，也可以相对干某个区域，比如窗口客户区。坐标原点是左上角。

ScreenToClient 函数和 ClientToScreen 函数是两个操作点坐标的 API 函数，其功能是将点从相对于屏幕的位置计算相对于窗口客户区的位置，以及从窗口客户区的位置计算相对于屏幕的位置，函数原型如下：

(1) ScreenToClient。

```
BOOL ScreenToClient(
  HWND hWnd,            // 窗口句柄
  LPPOINT lpPoint       // 屏幕坐标
);
```

(2) ClientToScreen。

```
BOOL ClientToScreen(
  HWND hWnd,        // 窗口句柄
  LPPOINT lpPoint   // 屏幕坐标
);
```

13.7 位图操作

本节将通过实例来具体讲解 DC 在 GDI 程序设计中的作用，演示如何操作 DC，同时还演示了 Bitmap 的使用。

实例包括以下内容。

◇截屏：对整个桌面截屏、截取当前窗口，截取指定的屏幕区域等。

◇操作位图：将截取的屏幕使用位图表示，演示了如何操作位图，如何保存位图文件。

◇显示位图：演示如何将位图显示在界面上，包括使用真实大小显示，及比例缩放后的演示。

主要涉及以下 API 函数。

◇ CreateDC。

◇ CreateCompatibleDC。

◇ GetDeviceCapS。

◇ SelectObject。

◇ BitBlt。

◇ stretchBlt。

◇ GetDIBits。

◇ DeleteDC。

◇ ReleaseDC。

◇ GetWindowRect。

主要涉及以下结构和数据类型。

◇ HDC。

◇ HBITMAP。

◇ BITMAPINFO。

◇ BITMAPFILEHEADER。

◇ RECTa

13.7.1 截取屏幕、保存位图文件

1．截取屏幕的流程

截取屏幕输出一般需要经过以下步骤。

◇使用字符串“DISPLAY”为参教，调用 CreateDC，得到类型为“显示”的 DC。

◇ 调用 CreateCompatibleDC 创建一个内存 DC。

◇得到需要截取的屏幕区域,比如调用 GetDeviceCaps 获取屏幕的大小,GetWindowRect 获取窗口区域等。

◇ 调用 CreateCompatibleBitmap 创建 BITMAP 对象。

◇调用 SelectObject,将创建的 BITMAP 选择入内存 DC，并将返回值使用另外一个 BITMAP 对象句柄来保存。SelectObj ect 的返回值表示了 DC 中被替换的图形对象。

◇调用 BitBlt 函数将源 DC（显示类型的 DC）指定位块(bit-block)的颜色数据转移到目的 DC（内存）中,.

◇将之前保存的旧的 BITMAP 对象再调用 SelectObject 选择回内存 DC 中，这样从显示 DC 中 BitBlt 到内存 DC 中的位块颜色数据就被替换，SelectObj ect 的返回值即是从屏幕 DC 中得到的位图对象（HBITMAP 类型）

之后就可以使用这个 HBITMAP 句柄来进行位图的操作，比如显示在屏幕的特定区域上。

2．实例 13-6 截取屏幕

首先是程序代码的声明部分，这里指定了 szCaptureWindowName，为“Windows 任务管理器”表示要截取任务管理器，可以将这个全局变量赋值为 NULL，这样会截取整个屏幕。全局变量 ghBitmap 是截取的位图对象。常量 CAP_SHOW MODE—SrIRTCH 和 CAP_SHOW_ MODE_ NOS IRTCH 用于后面 DoPaint 函数的参数，表示显示时是否根据显示窗口的大小拉伸。代码如下：

```
/* ************************************
* PrtSrcn.c 截屏
************************************/
/* 头文件 */
#include <windows.h>
/* 常量定义 */
#define PALVERSION 0x300
#define CAP_SHOW_MODE_STRTCH 1
#define CAP_SHOW_MODE_NOSTRTCH 0

/* 全局变量 */
HBITMAP ghBitmap = NULL;
RECT rectShow;
// 修改这里截取不同的窗口，如果为 NULL，则截取屏幕
LPSTR szCaptureWindowName = "Windows 任务管理器";

/* 函数申明 */
DWORD WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int);
LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);
HBITMAP ScreenCapture(LPSTR filename ,WORD BitCount);
VOID DoPaint(HWND hWnd);
```

然后是程序的入口函数 WinMain，完成了窗口类注册，窗口创建等操作：

```
/************************************
* DWORD WINAPI WinMain(HINSTANCE hinstance,
HINSTANCE hPrevInstance,
LPSTR lpCmdLine,
int nCmdShow)
* 功能：截屏，保存为文件，并显示在窗口上
************************************/
DWORD WINAPI WinMain(HINSTANCE hinstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine,
                     int nCmdShow)
{
     WNDCLASSEX wcx;
     HWND hwnd;
     MSG msg;
     WORD wport = 80;
     BOOL fGotMessage;
     HWND hwndCap = NULL;

     // 截取全屏幕还是窗口
     if(szCaptureWindowName != NULL)
     {
          hwndCap = FindWindow(NULL,"Windows 任务管理器");
          // 获取窗口的 RECT，可自行修改，获取屏幕中的任意区域
```

```
        if(!GetWindowRect(hwndCap,&rectShow))
        {
            MessageBox(NULL,"Can not find window to capture", "erroe",MB_OK);
            return 0;
        }
    }
    // 注册窗口类，并创建窗口，用于显示截取的位图
    wcx.cbSize = sizeof(wcx);
    wcx.style = CS_HREDRAW |
        CS_VREDRAW;
    wcx.lpfnWndProc = MainWndProc;
    wcx.cbClsExtra = 0;
    wcx.cbWndExtra = 0;
    wcx.hInstance = hinstance;
    wcx.hIcon = LoadIcon(NULL,
        MAKEINTRESOURCE(IDI_APPLICATION));
    wcx.hCursor = LoadCursor(NULL,
        IDC_ARROW);
    wcx.hbrBackground = (HBRUSH)GetStockObject(
        WHITE_BRUSH);
    wcx.lpszMenuName = NULL;
    wcx.lpszClassName = "MainWClass";
    wcx.hIconSm = NULL;
    if( !RegisterClassEx(&wcx))
    {
        return 1;
    }
    // 创建窗口
    hwnd = CreateWindow(
        "MainWClass",
        "CAP",
        WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_CLIPSIBLINGS |
        WS_MAXIMIZE | WS_POPUPWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        (HWND) NULL,
        (HMENU) NULL,
        hinstance,
        (LPVOID) NULL);
    if (!hwnd)
    {
        return 1;
    }
    // 截取屏幕，可根据需要设置不同的参数，这里只演示截取特定窗口
    ghBitmap = ScreenCapture("D:\\x.bmp" ,32,&rectShow);
    // 显示
    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);
    while ((fGotMessage = GetMessage(&msg, (HWND) NULL, 0, 0)) != 0 && fGotMessage != -1)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
    UNREFERENCED_PARAMETER(lpCmdLine);
}
```

WinMain 函数创建的窗口用于显示所截取的位图，调用了 ScreenCapture 函数截屏。

```
/*************************************
* BITMAP ScreenCapture(LPSTR filename ,WORD BitCount,LPRECT lpRect);
* 功能：截取指定区域的屏幕，并保存为文件
```

```
* 参数: LPSTR filename, 保存位图文件的文件路径, 如果为 NULL, 则不保存
*          WORD BitCount, Bit 深度, 用于表示一个像素点所使用的数据长度
*          LPRECT lpRect, 所需截取的屏幕区域, 如果为 NULL, 则获取全屏幕
* 返回: HBITMAP, 所截取的位图对象
************************************/
HBITMAP ScreenCapture(LPSTR filename ,WORD BitCount,LPRECT lpRect)
{
     HBITMAP hBitmap;
     // 显示器屏幕 DC
     HDC hScreenDC = CreateDC("DISPLAY", NULL, NULL, NULL);
     HDC hmemDC = CreateCompatibleDC(hScreenDC);
     // 显示器屏幕的宽和高
     int ScreenWidth = GetDeviceCaps(hScreenDC, HORZRES);
     int ScreenHeight = GetDeviceCaps(hScreenDC, VERTRES);
     // 旧的 BITMAP, 用于与所需截取的位置交换
     HBITMAP hOldBM;
     // 保存位图数据
     PVOID lpvpxldata;
     // 截屏获取的长宽及起点
     INT ixStart;
     INT iyStart;
     INT iX;
     INT iY;
     // 位图数据大小
     DWORD dwBitmapArraySize;
     // 几个大小
     DWORD nBitsOffset;
     DWORD lImageSize ;
     DWORD lFileSize ;
     // 位图信息头
     BITMAPINFO bmInfo;
     // 位图文件头
     BITMAPFILEHEADER bmFileHeader;
     // 写文件用
     HANDLE hbmfile;
     DWORD dwWritten;

     // 如果 LPRECT 为 NULL, 截取整个屏幕
     if(lpRect == NULL)
     {
          ixStart = iyStart = 0;
          iX = ScreenWidth;
          iY =ScreenHeight;
     }
     else
     {
          ixStart = lpRect->left;
          iyStart = lpRect->top;
          iX = lpRect->right - lpRect->left;
          iY = lpRect->bottom - lpRect->top;
     }
     // 创建 BTIMAP
     hBitmap = CreateCompatibleBitmap(hScreenDC, iX, iY);
     // 将 BITMAP 选择入内存 DC
     hOldBM = (HBITMAP)SelectObject(hmemDC, hBitmap);
     // BitBlt 屏幕 DC 到内存 DC, 根据所需截取的获取设置参数
     BitBlt(hmemDC, 0, 0, iX, iY, hScreenDC, ixStart, iyStart, SRCCOPY);
     // 将旧的 BITMAP 对象选择回内存 DC, 返回值为被替换的对象, 即所截取的位图
     hBitmap = (HBITMAP)SelectObject(hmemDC, hOldBM);
     if(filename == NULL)
```

```
    {
        DeleteDC( hScreenDC);
        DeleteDC(hmemDC);
        return hBitmap;
    }
    // 为位图数据申请内存空间
    dwBitmapArraySize = ((((iX*32) + 31) & ~31)>> 3)* iY;
    lpvpxldata = HeapAlloc(GetProcessHeap(),HEAP_NO_SERIALIZE,dwBitmapArraySize);
    ZeroMemory(lpvpxldata,dwBitmapArraySize);

    // 添充 BITMAPINFO 结构
    ZeroMemory(&bmInfo,sizeof(BITMAPINFO));
    bmInfo.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
    bmInfo.bmiHeader.biWidth = iX;
    bmInfo.bmiHeader.biHeight = iY;
    bmInfo.bmiHeader.biPlanes = 1;
    bmInfo.bmiHeader.biBitCount = BitCount;
    bmInfo.bmiHeader.biCompression = BI_RGB;

    // 添充 BITMAPFILEHEADER 结构
    ZeroMemory(&bmFileHeader,sizeof(BITMAPFILEHEADER));
    nBitsOffset = sizeof(BITMAPFILEHEADER) + bmInfo.bmiHeader.biSize;
    lImageSize =
        ((((bmInfo.bmiHeader.biWidth * bmInfo.bmiHeader.biBitCount) + 31) & ~31)>> 3)
        * bmInfo.bmiHeader.biHeight;
    lFileSize = nBitsOffset + lImageSize;
    bmFileHeader.bfType = 'B'+('M'<<8);
    bmFileHeader.bfSize = lFileSize;
    bmFileHeader.bfOffBits = nBitsOffset;

    // 获取 DIB 用于写入到文件
    GetDIBits(hmemDC, hBitmap, 0, bmInfo.bmiHeader.biHeight,
        lpvpxldata, &bmInfo, DIB_RGB_COLORS);
    // 写文件
    hbmfile = CreateFile(filename,
        GENERIC_WRITE,
        FILE_SHARE_WRITE,
        NULL,
        CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL,
        NULL);

    if(hbmfile == INVALID_HANDLE_VALUE)
    {
        MessageBox(NULL,"create file error","error",MB_OK);
    }
    WriteFile(hbmfile,&bmFileHeader,sizeof(BITMAPFILEHEADER),&dwWritten,NULL);
    WriteFile(hbmfile,&bmInfo,sizeof(BITMAPINFO),&dwWritten,NULL);
    WriteFile(hbmfile,lpvpxldata,lImageSize,&dwWritten,NULL);
    CloseHandle(hbmfile);

    // 释放内存，清除不同的 DC
    // 这里没有删除 BITMAP 对象，需在显示完成后删除
    HeapFree(GetProcessHeap(),HEAP_NO_SERIALIZE,lpvpxldata);
    ReleaseDC(0, hScreenDC);
    DeleteDC(hmemDC);
    return hBitmap;
}
```

ScreenCapture 函数是本实例中最重要的一个函数，本函数实现了截屏的功能，并保存截取的位图。13.7.2 小节将介绍如何将获取的位图显示在窗口界面上。

13.7.2 将位图显示在界面上

将位图文件显示在界面上的原理与截屏类似，也是将位图文件选择入 DC，区别是显示位图文件时所选择入的 DC 是用于显示的窗口 DC。如果在显示时需要拉伸，则需要使用 StretchBlt 函数在 DC 间转换。

1．显示位图的流程

(1)创建内存 DC，将 BITMAP 对象选择入内存 DC;

(2)获取用于显示位图的窗口 client 区域的 DC;

(3)从位图对象中获取位图属性，包括大小、长宽等;

(4)计算在 client 获取中显示的位置：

(5)根据需要调用 StretchBlt 或 BitBlt 函数,在内存 DC 和窗口 Client 区域 DC 间转移;

(6)释放相关资源。

2．关键 API

(1) StretchBlt。

StretchBlt 函数的功能是在两个 DC 的指定区域中复制位图，将源 DC 中指定区域的位图复制到目的 DC 的指定区域上，并根据区域的大小和长宽比例关系拉伸或压缩位图。

```
BOOL StretchBlt(
  HDC hdcDest,          // 目标 DC 句柄
  int nXOriginDest,     // 目标区域左上角 x 坐标
  int nYOriginDest,     // 目标区域左上角 y 坐标
  int nWidthDest,       // 目标矩形的宽度
  int nHeightDest,      // 目标矩形的高度
  HDC hdcSrc,           // 源 DC 句柄
  int nXOriginSrc,      // 源区域左上角 x 坐标
  int nYOriginSrc,      // 源区域左上角 y 坐标
  int nWidthSrc,        // 源矩形宽度
  int nHeightSrc,       // 源矩形高度
  DWORD dwRop           // 光栅操作代码
);
```

(2) BitBlt。

BitBlt 函数的功能是将源 DC 上的颜色信息转移到目的 DC 的指定区域上，转移时仅限于 DC 的最大范围，超出范围的部分不转移，不进行拉伸和压缩。

```
BOOL BitBlt(
  HDC hdcDest,      // 目标 DC 句柄
  int nXDest,       // 目标区域左上角 x 坐标
  int nYDest,       // 目标区域左上角 y 坐标
  int nWidth,       // 目标矩形的宽度
  int nHeight,      // 目标矩形的高度
  HDC hdcSrc,       // 源 DC 句柄
  int nXSrc,        // 源区域左上角 x 坐标
  int nYSrc,        // 源区域左上角 y 坐标
  DWORD dwRop
);
```

3．实例 13-7 将位图文件显示在界面上

MainWndProc 是窗口消息处理函数，在处理 WM PAINT 消息时调用 DoPaint 函数，在退出时，删除了全局的位图对象。DoPaint 函数用于将位图 gbBitmap 绘制到窗口上。

```
LRESULT CALLBACK MainWndProc(
                                HWND hwnd,
                                UINT uMsg,
                                WPARAM wParam,
                                LPARAM lParam)
{
     switch (uMsg)
     {
     case WM_PAINT:
          // 显示截取的屏幕
          DoPaint(hwnd,CAP_SHOW_MODE_STRTCH);
          break;
     case WM_DESTROY:
          // 创建的 BITMAP 对象需要删除，以释放资源
          DeleteObject(ghBitmap);
          ExitProcess(0);
          break;
     default:
          break;
     }
     return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

消息处理函数所调用的 DoPaint 函数实现了位图的显示，代码如下：

```
/*************************************
* VOID DoPaint(HWND hWnd, DWORD dwMode)
* 功能：将位图（全局变量 ghBitmap）显示在界面上
*
* 参数：HWND hWnd，用于显示位图的窗口
*      DWORD dwMode，模式，是否拉伸
*
* 无返回值
*************************************/
VOID DoPaint(HWND hWnd, DWORD dwMode)
{

     PAINTSTRUCT ps;
     RECT rect;
     HDC hdcMem;
     BITMAP bm;
     DIBSECTION ds;
     // BeginPaint
     HDC hDC = BeginPaint(hWnd, &ps);
     // 获取窗口的 Client 区域，用于显示位图
     GetClientRect(hWnd, &rect);

     // 设置拉伸模式
     SetStretchBltMode(hDC, HALFTONE);
     // 将 BITMAP 对象选入内存 DC
     hdcMem = CreateCompatibleDC(hDC);
     SelectObject(hdcMem, ghBitmap);
     if (ghBitmap)
     {
          // 获取 DIB 属性
```

```
            if (GetObject(ghBitmap, sizeof(BITMAP), &bm))
            {
                // 判断参数：是否根据显示窗口大小拉申位图
                // 采用不同的方式将内存 DC StretchBl t 至窗口 Client 区域 DC
                if(dwMode == CAP_SHOW_MODE_STRTCH)
                {
                    StretchBlt(hDC, 0, 0, rect.right, rect.bottom,
                        hdcMem, 0, 0, bm.bmWidth, bm.bmHeight, SRCCOPY);
                }
                else
                {
                    // 不拉伸，计算显示的位置，将其显示在 Client 的中央
                    INT ixStart = (rect.right - rect.left - bm.bmWidth)/2;
                    INT iyStart = (rect.bottom - rect.top - bm.bmHeight)/2;
                    ixStart = ixStart < 0 ? 0 : ixStart;
                    iyStart = iyStart < 0 ? 0 : iyStart;
                    BitBlt(hDC, 0, 0, rect.right, rect.bottom,
                        hdcMem,-ixStart,-iyStart, SRCCOPY);
                }
                DeleteDC(hdcMem);
            }
        }
        // 如果没有位图，则使用 Brush 填充
        else
        {
            PatBlt(hDC, 0, 0, rect.right, rect.bottom, BLACKNESS);
        }
        // EndPaint
        EndPaint(hWnd, &ps);
    }
```

图 13-9 是将位图进行了拉伸，以适应目标窗口的大小。

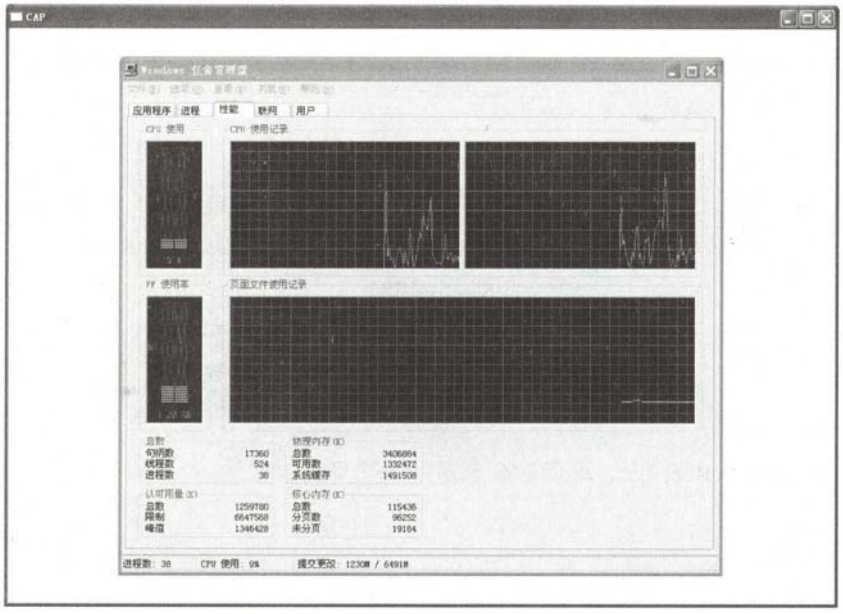

图 13-8　截取窗口（任务管理器），并将图显示在窗口中

图 13-8 所示，程序截取了窗口（任务管理器），‘并将位图以原始大小显示在窗口界面上。而

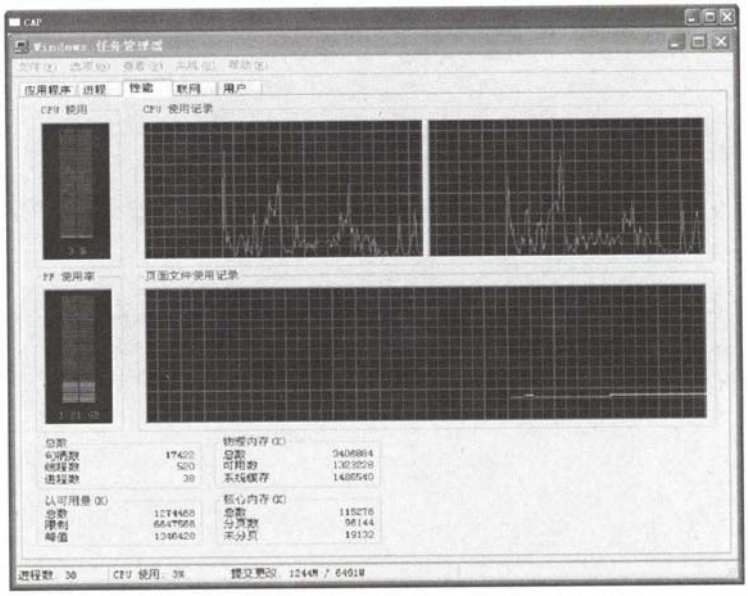

图 13-9　拉伸显示位图

13.8　区域(Regions)、路径(Paths)与修剪(Clip)操作

区域是一种对象，具有形状、位置、大小，用于进行填充、绘制、反转、边沿勾勒等操作，或用于限制 DC 的输出范围（修剪）。

区域的形状可以是任意的，可以是标准的矩形、椭圆、扇形等，也可以是多边形，还可以是这些形状组合（与、或、异或等）。

13.8.1　区域的创建及形状、位置等属性

区域是一种图形对象，HRGN 是区域对象句柄，HRGN 类型的数据用于唯一标识一个区域。创建区域可使用的函数如表 13.6 所示。

表 13.6　区域与 API

区　域	创建区域 API
矩形区域	CreateRectRgn、CreateRectRgnIndirect、SetRectRgn
圆角矩形区域	CreateRoundRectRgn
椭圆形区域	CreateEllipticRgn、CreateEllipticRgnIndirect
多边形区域	CreatePolygonRgn、CreatePolyPolygonRgn

(1) CreateRectRgn。

以函数为例 CreateRectRgn，此函数会创建一个矩形的区域。

```
HRGN CreateRectRgn(
  int nLeftRect,   // 左上角 x 坐标
  int nTopRect,    // 左上角 y 坐标
  int nRightRect,  // 右下角 x 坐标
  int nBottomRect  // 右下角 y 坐标
);
```

4 个参数分别为矩形四边的坐标。

(2) CreateRectRgnIndirect, CreateEllipticRgnIndirect。

CreateRectRgnIndirect、 CreateEllipticRgnIndirect 等函数是“直接”使用 RECT 创建指定形状的区域，以 RECT 为参数。

```
HRGN CreateEllipticRgnIndirect(
  CONST RECT *lprc   // 包括左下和右下角区域
);
```

(3) SetRectRgn。

SetRectRgn 函数的功能是将区域转换为一个矩形区域。

```
BOOL SetRectRgn(
  HRGN hrgn,        // 区域句柄
  int nLeftRect,    // 矩形左上角 x 坐标
  int nTopRect,     // 矩形左上角 y 坐标
  int nRightRect,   // 矩形右下角 x 坐标
  int nBottomRect   // 矩形右下角 y 坐标
);
```

hrgn 是区域的句柄，其他四个参数是矩形区域四边的坐标。

13.8.2 区域边沿、区域填充、反转与勾勒操作

在创建区域后，可以对区域进行填充和反转等操作。

填充区域使用 API 函数 FillRgn 或 PaintRgn。FillRgn 可以指定画刷，而 PaintRgn 使用当前 DC 画刷。使用 DC 中被选入的当前画刷。

(1) FillRgn。

```
BOOL FillRgn(
  HDC hdc,     // DC 句柄
  HRGN hrgn,   // 填充区域句柄
  HBRUSH hbr   // 填区域刷句柄
);
```

(2) PaintRgn。

```
BOOL PaintRgn(
  HDC hdc,     // DC 句柄
  HRGN hrgn    // 要画区域的句柄
);
```

(3)InvertRgn。

```
BOOL InvertRgn(
  HDC hdc,     // DC 句柄
  HRGN hrgn    // 要转化的区域句柄
);
```

反转是将区域当前的填充颜色取反，使用 API 函数 InvertRgn。

(4) FrameRgn。

勾勒(Frame)是使用指定的画刷在区域外为区域绘制边框。

```
BOOL FrameRgn(
  HDC hdc,     // DC 句柄
  HRGN hrgn,   // 加框区域句柄
  HBRUSH hbr, // 边框画刷句柄
  int nWidth, // 帧区域宽度
  int nHeight // 帧区域宽度
);
```

无论什么形状的区域，都有一个边沿，这个边沿是一个矩形(RECT)，是能进入这个区域框的最小的矩形。GetRgnBox 函数用于获取这个矩形。

```
int GetRgnBox(
  HRGN hrgn,   // 区域句柄
  LPRECT lprc // 整个矩形区域
);
```

1．实例 13-8 使用区域

在本实例中，演示了如何创建区域，如何对区域进行填充和反转。

当用户在窗口上点击鼠标后，消息处理函数会将单击的点传递给 ElliRgns 函数，ElliRgns 函

数在点的周围选择了一个宽 80、高 60 的范围，设置了 RECT 类型的变量，然后调用 CreateEllipticRgnIndirect 函数创建椭圆的区域，之后对这个区域进行了颜色填充和反转。

```
/* ************************************
* Regions.c 区域的填充和反转
************************************/
/* 头文件 */
#include <windows.h>
/* 全局变量 */
LPSTR szAppName = "RGN";
LPSTR szTitle = "RGN Sample";
/* 函数声明 */
LRESULT CALLBACK WndProc( HWND , UINT , WPARAM , LPARAM );

/************************************
* ElliRgns
* 功能：创建椭圆区域，并进行填充和反转
************************************/
HRGN ElliRgns(HWND hwnd, POINT point)
{
     // RECT 区域、画刷
     RECT rect, rectClient;
     HRGN hrgn;
     HBRUSH hBrushOld, hBrush;
     // DC
     HDC hdc = GetDC(hwnd);
     GetClientRect(hwnd,&rectClient);
     // 点的周围一块区域
     rect.left = point.x - 40;
     rect.right = point.x + 40;
     rect.top = point.y - 30;
     rect.bottom = point.y + 30;
     // 通过 RECT 创建椭圆区域
     hrgn = CreateEllipticRgnIndirect(&rect);
     // 创建画刷
     hBrush = CreateSolidBrush(RGB(0,255,0));
     // 为 DC 选择画刷
     hBrushOld = (HBRUSH)SelectObject(hdc,hBrush);
     // 使用当前画刷绘制区域
     PaintRgn(hdc,hrgn);
     // 等待一段时间后，将刚才绘制的区域进行反转
     Sleep(3000);
     InvertRgn(hdc,hrgn);
     // 释放资源
     hBrush = (HBRUSH)SelectObject(hdc,hBrushOld);
     DeleteObject(hBrush);
     DeleteObject(hrgn);
     DeleteDC( hdc );
     return 0;
}
int WINAPI WinMain(
```

```
                    HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow)
{
     MSG msg;
     HWND hWnd;
     WNDCLASS wc;
     wc.style = CS_OWNDC;
     wc.lpfnWndProc = (WNDPROC)WndProc;
     wc.cbClsExtra = 0;
     wc.cbWndExtra = 0;
     wc.hInstance = hInstance;
     wc.hIcon = NULL;
     wc.hCursor = LoadCursor(NULL, IDC_ARROW);
     wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
     wc.lpszMenuName = NULL;
     wc.lpszClassName = szAppName;

     if (!RegisterClass(&wc))
     {
          return (FALSE);
     }
     hWnd = CreateWindow(
          szAppName,
          szTitle,
          WS_OVERLAPPEDWINDOW,
          CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
          NULL,      NULL,     hInstance, NULL
          );
     if (!hWnd)
     {
          return (FALSE);
     }
     ShowWindow(hWnd, nCmdShow);
     UpdateWindow(hWnd);
     while (GetMessage(&msg, NULL, 0, 0))
     {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }
     return (int)(msg.wParam);
     UNREFERENCED_PARAMETER(lpCmdLine);
}


LRESULT CALLBACK WndProc(
                              HWND hWnd,
                              UINT message,
                              WPARAM uParam,
                               LPARAM lParam)
{
     PAINTSTRUCT ps;
     RECT rc;
     switch (message)
     {
     case WM_CREATE:
          break;
     case WM_PAINT:
          break;
     case WM_LBUTTONDOWN:
          {
```

```
                // 获得单击的位置，传递给 ElliRgns
                POINT point;
                point.x = LOWORD(lParam);
                point.y = HIWORD(lParam);
                ElliRgns(hWnd, point);
            }
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return (DefWindowProc(hWnd, message, uParam, lParam));
        }
        return (0);
}
```

2．实例运行效果

程序运行后，在窗口上单击鼠标，会出现椭圆有绿色图形，过一段时间后，图形会变为紫色，如图 13 -10 所示。

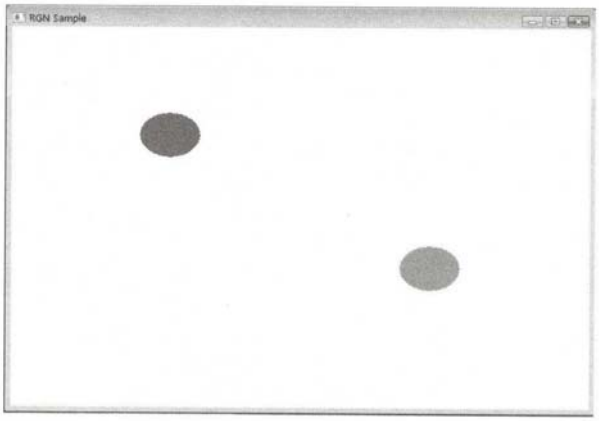

图 13-10　实例 13-8 运行效果——区域的填充

13.8.3　组合、比较、移动等操作

组合、比较、移动是对区域的基本操作。

1．组合

在创建区域后，还可以将若干个区域组合在一起，成为一个新的区域。组合的方式可以是两个区域的交集、合集、差集、合集与交集的差集（两集合异或）。

CombineRgn 函数实现了对区域的组合。

```
int CombineRgn(
  HRGN hrgnDest,          // 目标区域句柄
  HRGN hrgnSrc1,          // 源区域句柄 1
  HRGN hrgnSrc2,          // 源区域句柄 2
  int fnCombineMode       // 区域各并模式
);
```

hrgnDest 是保存目标区域的句柄，hrgnSrc1 和 hrgnSrc2 分别是两个源区域。fnCombineMode 参数是合并的模式，其值可以是 RGN AND、RGN_COPY、RGN_DIFF、RGN_OR、RGN XOR，分别表示两个区域的交集，直接复制第一个区域，去除第一个区域中与第二个区域重合的部分，两个区域的并集，两个区域差集。

2．比较

还可以对两个区域进行比较，判断是否相同。使用 API 函数 EqualRgn。

```
BOOL EqualRgn(
  HRGN hSrcRgn1,   // 区域 1 句柄
  HRGN hSrcRgn2    // 区域 2 句柄
);
```

3．移动

对一个区域，还可以进行移动。OffsetRgn 函数实现移动的功能。

```
int OffsetRgn(
  HRGN hrgn,       // 区域句柄
  int nXOffset,    // x 轴偏移量
  int nYOffset     // y 轴偏移量
);
```

参数 nXOffset 和 nYOffset 分别是需要移动的相对位移的横纵坐标。

13.8.4 点击测试(Hit Testing)

所谓点击测试是指判断一次鼠标点击是否位于区域的范围内。PtInRegion 函数实现了这种判断。

```
BOOL PtInRegion(
  HRGN hrgn,   // 区域句柄
  int X,       // 点 x 坐标
  int Y        // 点 y 坐标
);
```

参数 X 和 Y 代表了点和坐标，函数判断该点是否在 hrgn 的范围内。

实例 13-9 点击测试与画刷样式

本实例演示两方面的内容，一是使用区域来测试鼠标点击的位置是否在一定的范围之内，二是使用多种画刷样式来演示画刷的使用。

在实例中，定义了多个方格，有一左一右两个方格，在程序运行时，将左侧方格划分为 64 个小方格。用户可以使用鼠标点击，程序在收到鼠标点击消息后，会判断点击的位置是否位于左侧的方格中，位于哪个小方格中。然后将用户点击的那个小方格使用画刷填充。第一行的小方格所使用的画刷是不一样的。

在右侧的两个方格中，会使用位图画刷进行填充，用户每次点击鼠标后，会根据鼠标位置生成不同的黑白位图，然后使用位图创建格式画刷，再用画刷将方格填充。

```
/* ************************************
* bp.c 点击测试
**************************************/
/* 头文件 */
#include <Windows.h>
/* 全局变量 */
RECT rctGrid; // 格子
RECT rctBrush; // 演示格式画刷的格子
RECT rctWindow;
UINT bBrushBits[8]; // 用于创建格式画刷
RECT rect[64]; // 64 个小格子
HBITMAP hbm; // 位图画刷
```

```
HBRUSH hbrush; // 当前画刷
HBRUSH hBrushGrid[8]; // 用于填充小方格式的画刷
HBRUSH hbrushOld; // 默认画刷
HRGN hrgnCell; // 区域
HDC hdc; // DC 句柄
POINTS ptlHit; // 鼠标坐标
int i;

/*************************************
* VOID InitGridBurshAndRect()
* 功能 初始化程序需要使用到的画刷
* 初始化若干个 RECT
*************************************/
VOID InitGridBurshAndRect()
{

     int x, y, deltaX, deltaY; // 用于画小格子

     // 创建了 7 种不同的画刷
     hBrushGrid[0] = CreateHatchBrush(HS_BDIAGONAL,RGB(255,0,0));
     hBrushGrid[1] = CreateHatchBrush(HS_CROSS,RGB(0,255,0));
     hBrushGrid[2] = CreateHatchBrush(HS_DIAGCROSS,RGB(0,0,255));
     hBrushGrid[3] = CreateHatchBrush(HS_FDIAGONAL,RGB(255,0,255));
     hBrushGrid[4] = CreateHatchBrush(HS_HORIZONTAL,RGB(255,255,0));
     hBrushGrid[5] = CreateHatchBrush(HS_BDIAGONAL,RGB(0,255,255));
     hBrushGrid[6] = CreateHatchBrush(HS_VERTICAL,RGB(0,0,0));
     hBrushGrid[7] = CreateSolidBrush(RGB(200,200,200));

     // 大格子边界的坐标，后面会被划分为 8×8 个小方格子，用于点击测试
     rctGrid.top = rctGrid.left = 10;
     rctGrid.bottom = rctGrid.right = 330;// 靠左
     rctBrush.top = 10; rctBrush.left = 340;
     rctBrush.bottom = 330;rctBrush.right = 660;// 靠右
     // 计算小格式中的坐标，保存在数组中
     deltaX = (rctGrid.right - rctGrid.left)/8;
     deltaY = (rctGrid.bottom - rctGrid.top)/8;
     for (y=rctGrid.top, i=0; y < rctGrid.bottom; y += deltaY)
     {
          for(x=rctGrid.left; x < (rctGrid.right - 8) && i < 64;
               x += deltaX, i++)
          {
               rect[i].left = x; rect[i].top = y;
               rect[i].right = x + deltaX;
               rect[i].bottom = y + deltaY;
          }
     }
}

/*************************************
* VOID DrawGridLine(HWND hwnd)
* 功能：根据 InitGridBurshAndRect 所初始化的小方格的边界画出小方格
*************************************/
VOID DrawGridLine(HWND hwnd)
{
     hdc = GetDC(hwnd);
     // 经线和纬线
     for (i=rctGrid.left; i<=rctGrid.right;
          i+=(rctGrid.right - rctGrid.left)/8)
     {
          MoveToEx(hdc, i, rctGrid.top, NULL);
          LineTo(hdc, i, rctGrid.bottom);
     }
     for (i=rctGrid.top; i<=rctGrid.bottom;
```

```
            i+=(rctGrid.bottom - rctGrid.top)/8)
        {
            MoveToEx(hdc, rctGrid.left, i, NULL);
            LineTo(hdc, rctGrid.right, i);
        }
        ReleaseDC(hwnd, hdc);
        DeleteDC( hdc );
    }

    /*************************************
    *VOID PaintGrid(HWND hwnd,POINTS ptlHit)
    * 功能：使用画刷填充 ptlHit 点所在的小方格
    *************************************/
    VOID PaintGrid(HWND hwnd,POINTS ptlHit)
    {
        // 创建区域，大方格
        hrgnCell = CreateRectRgn(rctGrid.left, rctGrid.top,
            rctGrid.right, rctGrid.bottom);
        // 获得窗口的 DC
        hdc = GetDC(hwnd);
        // 将区域选择入 DC
        SelectObject(hdc, hrgnCell);
        // 测试点击的位置是否在区域中
        if (PtInRegion(hrgnCell, ptlHit.x, ptlHit.y))
        {
            // 如果点在大方格中，循环测试点在哪个小方格中
            for (i=0; i<8; i++)
                bBrushBits[i] = 0xFF;
            for(i=0; i<64; i++)
            {
                DeleteObject(hrgnCell);
                // 每一个小方格创建一个区域
                hrgnCell = CreateRectRgn(rect[i].left,
                    rect[i].top,
                    rect[i].right, rect[i].bottom);
                // 测试点击的位置是否在小方格中
                if (PtInRegion(hrgnCell, ptlHit.x, ptlHit.y))
                {
                    // 如果是的话则填充，第一行使用的填充画刷是一样的
                    FillRgn(hdc,hrgnCell,hBrushGrid[i/8]);
                }
            }
            // 释放
            SelectObject(hdc, hbrushOld);
            DeleteObject(hbrush);
            DeleteObject(hbm);
            DeleteDC( hdc);     }
    }

    /*************************************
    *VOID PaintPattern(HWND hwnd, POINTS ptlHit)
    * 功能：在演示格式画刷的方格中使用格式画刷进行绘制
    *************************************/
    VOID PaintPattern(HWND hwnd, POINTS ptlHit)
    {
        // 获取 DC
        hdc = GetDC(hwnd);
        // 根据点击的不同位置创建不同的位图
        i = ptlHit.x % 8;
        bBrushBits[i] += 0x50;
        // 创建位图
        hbm = CreateBitmap(8, 8, 1, 1, (LPBYTE)bBrushBits);
        // 创建格式画刷
```

```
        hbrush = CreatePatternBrush(hbm);
        // 为DC选择画刷
        hbrushOld = SelectObject(hdc, hbrush);
        // 绘制矩形
        Rectangle(hdc, rctBrush.left, rctBrush.top,
             rctBrush.right, rctBrush.bottom);
        // 释放DC
        ReleaseDC(hwnd,hdc);
        DeleteDC( hdc );
}

/************************************
* MainWndProc
* 功能：窗口消息处理函数
************************************/
LRESULT CALLBACK MainWndProc(
                                    HWND hwnd,
                                    UINT uMsg,
                                    WPARAM wParam,
                                    LPARAM lParam)
{
        switch (uMsg)
        {
             // 释放资源，退出
        case WM_DESTROY:
             for(i=0; i<8; i++)
                  DeleteObject(hBrushGrid[i]);
             ExitProcess(0);
             // 初始化画刷、方格
        case WM_CREATE:
             InitGridBurshAndRect();
             break;
             // 绘制
        case WM_PAINT:
             DrawGridLine(hwnd);
             break;
             // 鼠标左键
        case WM_LBUTTONDOWN:
             ptlHit = MAKEPOINTS((LPPOINTS)lParam);
             // 填充一个小方格
             PaintGrid(hwnd, ptlHit);
             // 填充右侧的格式画刷方格
             PaintPattern(hwnd, ptlHit);
             break;
        default:
             break;
        }
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
// WinMain
int WINAPI WinMain(HINSTANCE hinstance,
                        HINSTANCE hPrevInstance,
                        LPSTR lpCmdLine,
                        int nCmdShow)
{
        WNDCLASSEX wcx;
        HWND hwnd;
        MSG msg;
        WORD wport = 80;
        BOOL fGotMessage;
        HWND hwndCap = NULL;
        // 注册窗口类，并创建窗口，用于显示截取的位图
        wcx.cbSize = sizeof(wcx);
```

```
    wcx.style = CS_HREDRAW | CS_VREDRAW;
    wcx.lpfnWndProc = MainWndProc;
    wcx.cbClsExtra = 0;
    wcx.cbWndExtra = 0;
    wcx.hInstance = hinstance;
    wcx.hIcon = LoadIcon(NULL, MAKEINTRESOURCE(IDI_APPLICATION));
    wcx.hCursor = LoadCursor(NULL, IDC_ARROW);
    wcx.hbrBackground = (HBRUSH)GetStockObject( WHITE_BRUSH );
    wcx.lpszMenuName = NULL;
    wcx.lpszClassName = "MainWClass";
    wcx.hIconSm = NULL;
    if( !RegisterClassEx(&wcx))
        return 1;
    // 创建窗口
    hwnd = CreateWindow(
        "MainWClass",
        "Brush_Pen",
        WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_CLIPSIBLINGS |
        WS_MAXIMIZE | WS_POPUPWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        800, 600,
        (HWND) NULL, (HMENU) NULL, hinstance, (LPVOID) NULL);

    if (!hwnd)
        return 1;
    // 显示
    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);
    while ((fGotMessage = GetMessage(&msg, (HWND) NULL, 0, 0)) != 0 && fGotMessage != -1)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
    UNREFERENCED_PARAMETER(lpCmdLine);
}
```

运行实例。程序会在界面上绘制 8x8 的方格。点击哪个方格，哪个方格就会被填充。所填充的内容各不相同。这些填充的画刷都是在 InitGridBurshAndRect 函数中创建的。
在 PaintGrid 函数中判断点击的哪个方格。绘制方格边界的直线是使用 LineTo 和 MoveToEx 函数绘制的。

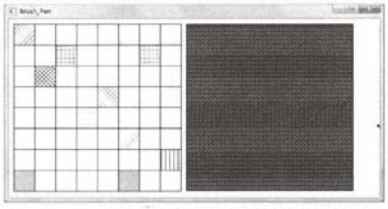

图 13-11　点击测试和画刷样式示例

程序运行结果如图 13-11 所示。

13.8.5　路径的创建与操作

路径是很多被填充或被勾勒了的形状和图形。路径的组和创建方式非常灵活。可以比较轻松地创建复杂和多样的路径。

路径的创建方法比较特殊。整个创建过程称作“路径托架(bracket)”。

创建路径时，首先调用 BeginPath 函数，然后可以调用一系统的绘制函数，最后再调用 EedPath 函数。在这个过程中，所有的绘制函数的输出内容都不会被真正输出到屏幕上，而是输出到“路径”中，路径就是整个 BeginPath 和 EedPath 之间所有调用的输出的图形的组

合。在 EndPath 调用之后，
路径会被选择到 DC 中。下面列举了部分绘制函数，在 BeginPath 和 EndPath 之间调用便会形成路径的形状。

◇ ArcTo。
◇ Chord。
◇ CloseFigure。
◇ Ellipse。
◇ ExtTextOut。
◇ LineTo。
◇ Pie。
◇ PolyBezier。
◇ PolyDraWO
◇Polygon。
◇ Polyline。
◇ PolylineTO。
◇ Rectanglea
◇ RoundRect。
◇ TextOut。

创建了路径之后，可以对路径进行操作，包括绘制边沿(Outline)、填充、变形、转换等。

13.8.6 路径转换为区域

可以将路径转换为一个区域，然后就可以进行区域的所有操作。将路径转换为区域使用函数 PathToRegion，函数原型如下。

```
HRGN PathToRegion(
  HDC hdc   // DC 句柄
);
```

所转换的是 DC 的当前路径，返回转换后得到的区域。

13.8.7 使用区域和路径进行修剪操作，限制输出

修剪是区域和路径比较高级的操作。修剪是使用 DC 的区域和路径对象来限制绘制输出的范围。修剪区域可以将所有输出都限制在区域的范围内，只有在区域范围内的输出才会显示在界面上，如果输出的范围在区域之外，则不会反映到界面上。

在进行修剪操作前，首先需设置 DC 的修剪区域和修剪路径。设置使用区域使用 API 函数 SelectClipRgn，函数原型如下：

```
int SelectClipRgn(
  HDC hdc,     // DC 句柄
  HRGN hrgn    // 区域句柄
);
```

函数将参数 hrgn 指定的区域设置为修剪区域，修剪区域设置后还可以进行修改，包括移动(OffsetClipRgn)、减少范围(ExcludeClipRect)和扩大范围(IntersectClipRect)。

还可以设置 DC 的修剪路径，修剪路径可以和 DC 原有的修剪区域进行合并，合并的方法也包括集合的交、并、差、复制、异或（与方式类似于区域的合并操作，详见实例)。

设置修剪路径使用 API 函数 SelectCliDPath，函数原型如下：

```
BOOL SelectClipPath(
  HDC hdc,     // DC句柄
  int iMode    // 修剪模式
);
```

函数会使用 DC 的当前路径与当前修剪区域进行组合，iMode 参数指明了组合方式，参数值与区域合并时所设置的参数值一致(RGN AND、RGN COPY、RGN DIFF、RGN_OR、RGN XOR)。

1．实例 13-10 修剪操作

本实例使用演示了使用修剪操作来限制输出。在用户通过菜单输出 IDM_CLIP 指令时，程序定义了一个五角星形的多边形区域，然后调用 SelectClipRgn 将区域设置为 DC 的修剪区域。之后进行了两个输出操作，一是输出了一段文字，二是以客户区左上角为起点，从上到下，旋转一个角度画一条直线。

由于在输出文字的直线前，DC 已经选择了修剪区域，因此只有在修剪区域以内的输出才在界面上显示出来，如图 13 -12 所示。

```
/* ************************************
* clip.cpp 区域、路径、修剪
**************************************/
/* 头文件 */
#include <Windows.h>
#include <math.h>
#include "resource.h"
/* 全局变量 */
CHOOSEFONT cf; // 选择字体
LOGFONT lf; // 逻辑字体
HFONT hfont; // 字体对象
HFONT hfontOld; // DC的原有字体
HDC hdc; // DC句柄
int nXStart, nYStart; // 用于画直线
RECT rc; // 窗口客户区RECT
double aflSin[90]; // 用于绘制放射直线簇
double aflCos[90]; // 用于绘制放射状直线簇
double flRadius,a; // 半径
int iMode; // 修剪路径组合模式
HRGN hrgn; // 修剪区域
int i;
COLORREF crOld;

// 窗口消息处理函数
LRESULT APIENTRY MainWndProc(
                              HWND hwnd,
                              UINT message,
                              WPARAM wParam,
                              LPARAM lParam)

{

     PAINTSTRUCT ps;

     switch (message)
     {
     case WM_PAINT:
          hdc = BeginPaint(hwnd, &ps);
          EndPaint(hwnd, &ps);
          break;
     case WM_COMMAND: // 菜单命令
          switch (wParam)
          {
          case IDM_VANISH: // 消除客户区（在设置了修剪全其输出范围也会被限制）
               hdc = GetDC(hwnd);
               GetClientRect(hwnd, &rc);
               FillRect(hdc, &rc, (HBRUSH)GetStockObject(WHITE_BRUSH));
```

```
            ReleaseDC(hwnd, hdc);
             DeleteDC( hdc);
            break;
            // 根据用户的菜单输入，设置修剪路径的组合模式
        case IDM_AND:
            iMode = RGN_AND;
            break;
        case IDM_COPY:
            iMode = RGN_COPY;
            break;
        case IDM_DIFF:
            iMode = RGN_DIFF;
            break;
        case IDM_OR:
            iMode = RGN_OR;
            break;
        case IDM_XOR:
            iMode = RGN_XOR;
            break;
        case IDM_CLIP:
            {
                // 获取窗口客户区的 DC
                hdc = GetDC(hwnd);
                // 创建并为 DC 选择字体
                // 消息在菜单中点击此项前需先选择字体
                hfont = CreateFontIndirect(cf.lpLogFont);
                hfontOld = (HFONT)SelectObject(hdc, hfont);
                // 5 个点，用于创建五角星区域
                POINT point[5]= {{0,200},{600,200},{100,600},{300,0},{500,600}};
                // 创建多边形区域，五角星
                hrgn = CreatePolygonRgn(point, 5, WINDING);
                // 将区域选择为修剪区域
                SelectClipRgn(hdc, hrgn);
                // 输出的文字
                LPSTR szOut = "Lorem ipsum dolor sit amet, consectetur \n"
                    "adipisicing elit, sed do eiusmod tempor \n"
                    "incididunt ut labore et dolore magna \n"
                    "aliqua. Ut enim ad minim veniam, quis \n"
                    "nostrud exercitation ullamco laboris nisi \n"
                    "ut aliquip ex ea commodo consequat. Duis \n"
                    "aute irure dolor in reprehenderit in \n"
                    "voluptate velit esse cillum dolore eu \n"
                    "fugiat nulla pariatur. Excepteur sint \n"
                    "occaecat cupidatat non proident, sunt \n"
                    "in culpa qui officia deserunt mollit \n"
                    "anim id est laborum.\n";
                // 窗口客户区，用于输出文字
                RECT rect;
                GetClientRect(hwnd,&rect);
                // 设置文字背景为透明
                SetBkMode(hdc, TRANSPARENT);
                //输出的文字
                DrawText(hdc, szOut, lstrlen(szOut),&rect , DT_CENTER);

                // 输出放射状直线，以左上角为原点
                // 计算线的终点
                for (i = 0; i < 90; i++)
                {
                    aflSin[i] = sin( (((double)i) / 180.0)
                        * 3.14159);
                }
                for (i = 0; i < 90; i++)
                {
                    aflCos[i] = cos( (((double)i) / 180.0)
```

```
                              * 3.14159);
                    }
                    flRadius = 1000;// 线的长度（窗口大小为 600×650）
                    // 画线，第一度画一条线
                    for (i = 0; i < 90; i++)
                    {
                         MoveToEx(hdc, nXStart, nYStart,
                              (LPPOINT) NULL);
                         LineTo(hdc, nXStart + ((int) (flRadius
                              * aflCos[i])),
                              nYStart + ((int) (flRadius
                              * aflSin[i])));
                    }
                    // 选择字体释放
                    SelectObject(hdc, hfontOld);
                    DeleteObject(hfont);
                    ReleaseDC(hwnd, hdc);
                    DeleteDC( hdc);
                    // 刷新窗口
                    UpdateWindow(hwnd);
                    break;
               }
          case IDM_FONT:
               // 用户选择字体
               cf.lStructSize = sizeof (CHOOSEFONT);
               cf.hwndOwner = hwnd;
               cf.lpLogFont = &lf;
               cf.Flags = CF_SCREENFONTS | CF_EFFECTS;
               cf.rgbColors = RGB(0, 255, 255);
               cf.nFontType = SCREEN_FONTTYPE;
               ChooseFont(&cf);
               break;
          default:
               return DefWindowProc(hwnd, message, wParam,
                    lParam);
          }
          break;

     case WM_DESTROY:
          PostQuitMessage(0);
          break;

     default:
          return DefWindowProc(hwnd, message, wParam, lParam);
     }
     return 0;
}

// WinMain
int WINAPI WinMain(
                    HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow)
{
     MSG msg;
     HWND hWnd;
     WNDCLASS wc;
     wc.style = CS_OWNDC;
     wc.lpfnWndProc = (WNDPROC)MainWndProc;
     wc.cbClsExtra = 0;
     wc.cbWndExtra = 0;
     wc.hInstance = hInstance;
     wc.hIcon = NULL;
```

```
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName = MAKEINTRESOURCE(IDR_MENU_CLIP);//菜单
    wc.lpszClassName = "clip";
    if (!RegisterClass(&wc))
    {
        return (FALSE);
    }
    hWnd = CreateWindow(
        "clip",
        "clip",
        WS_OVERLAPPEDWINDOW,
        100, 100, 600, 650,
        NULL,
        NULL,
        hInstance,
        NULL
        );
    if (!hWnd)
    {
        return (FALSE);
    }
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return (int)(msg.wParam);

    UNREFERENCED_PARAMETER(lpCmdLine);
}
```

2．运行结果

运行结果如图 13 -12 所示。

从运行结果可以看出，在为 DC 选择了修剪区域后，无论是输出的文字还是线条，都被限制在了修剪区域的范围内。

3．实例 13-11　修剪区域与修剪路径的组合

对实例 13 -11 作少量修改将消息处理函数中处理 IDM_CLIP 命令时输出文字的，代码使用下面的代码替换。

原有代码：

```
//输出的文字
DrawText(hdc, szOut, lstrlen(szOut),&rect , DT_CENTER);
```

替换为：

```
BeginPath(hdc);
// 路径的形状为输出的文字
DrawText(hdc, szOut, lstrlen(szOut),&rect , DT_CENTER);
EndPath(hdc);// 路径已经为 DC 的当前路径
// 为 DC 选择修剪路径，使用用户通过菜单输入的模式
// 注意在进行此操作前需通过菜单选择组合模式
SelectClipPath(hdc, iMode);
```

新增加了 BeginPath、Endpath 和 SeletClipPath 三个函数的调用。代码作如上修改后，在原来的修剪区域的基础上又增加了修改路径。而路径的形状是使用 DrawText 函数输出的文字。这样程

序可就使用修剪区域与修剪路径的组合来限制放射线条的输出范围。

重新编译运行，在进行修剪输出前，需要先选择字体和修剪区域组合模式。

◇通过菜单选择字体，为了看清楚画线条之后的文字，建议使字体大小为 100。

◇选择模式为或(OR)，选择修剪(CLIP)。之后程序会应用修剪，并输出线条。

由于选择的模式为或，因此修剪区域和修剪路径所有覆盖的部分都有输出，如图 13-13 所示。

图 13-12　修剪操作

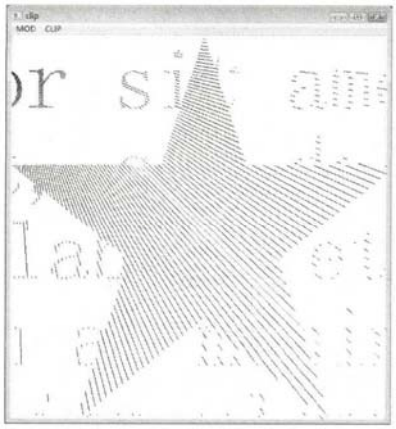

图 13-13　或模式修剪操作

◇清空客户区（VANISH)，选择模式为异或(XOR)，进行修剪输出(CLIP)。

由于选择的模式为异或(XOR)，所有修剪区域和修剪路径都没有覆盖或都有覆盖的范围都没有输出，如图 13-14 所示。

◇清空客户区（VANISH)，选择模式为与(AND)，进行修剪并输出。

只有修剪区域和修剪路径都覆盖的范围才有输出，如图 13 -15 所示。

图 13-14　异或模式修剪操作

图 13-15　与模式修剪操作

◇清空客户区（VANISH)，选择模式为差(DIFF)，进行修剪并输出。

修剪区域中修剪路径覆盖的部分被排除在显示区域，效果如图 13 -16 所示。

◇清空客户区(VANISH)，选择模式为复制(COPY)，进行修剪并输出。

输出的范围只有修剪路径的范围，如图 13-17 所示。

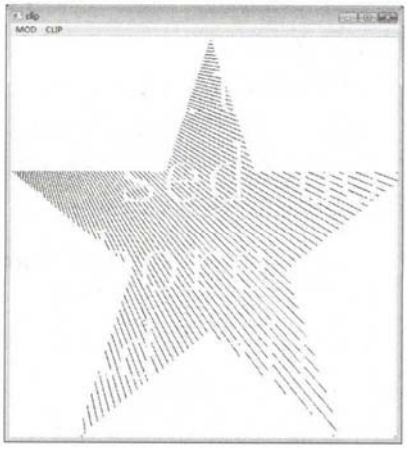

图 13-16　差模式修剪操作

图 13-17　复制模式修剪操作

4．实例 13-12 多边形的填充模式

在实例 13-11 的基础上，将 IDM_CLIP 命令处理过程中的区域创建函数的第二个参数由 WINDING 修改为 ALTERNATE，即：

```
// 创建多边形区域，五角星
hrgn = CreatePolygonRgn(point, 5, WINDING);
```

修改为：

```
// 创建多边形区域，五角星
hrgn = CreatePolygonRgn(point, 5, ALTERNATE);
```

编译，按照实例 13-11 的过程运行实例，输出结果如图 13-18 所示。

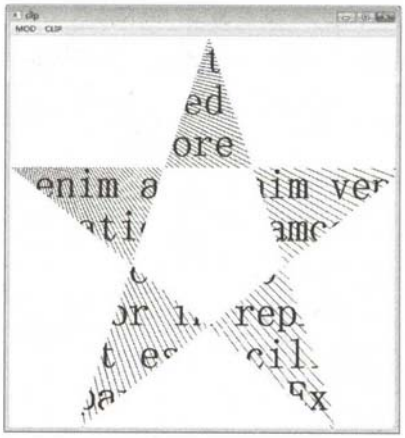

图 13-18　多边形填充模式

与图 13-12 对比，体现出多边形的两种填充模式的区别。

13.9　坐标变换

图形输出是在平面上操作，具有坐标。GDI 支持对输出的函数进行坐标变换。API 函数 SetWorldTransform 实现了坐标变换的功能，函数原型如下：

```
BOOL SetWorldTransform(
  HDC hdc,                  // DC 句柄
  CONST XFORM *lpXform      // 坐标变换
);
```

参数 lpXform 是指向 XFORM 结构的指针，XFORM 是坐标变换的参数。

```
typedef struct  _XFORM {
  FLOAT eM11;
  FLOAT eM12;
  FLOAT eM21;
  FLOAT eM22;
  FLOAT eDx;
  FLOAT eDy;
} XFORM, *PXFORM;
```

在使用 XFORM 进行坐标变换时，点的原坐标{x，y）与新坐标{x’　y’）转换的公式如下：

```
x' = x * eM11 + y * eM21 + eDx,
y' = x * eM12 + y * eM22 + eDy,
```

下面举两个例子，通过坐标变换将输出的内容缩小为原来的二分之一；将输出的内容逆时针旋转 30 度。分别如图 13-20 和图 13-21 所示。

输出的内容为椭圆。

```
Ellipse(hDC, (rect.right / 2 - 200), (rect.bottom / 2 + 100),
       (rect.right / 2 + 200), (rect.bottom / 2 - 100));
```

图 13-19 所示是未进行坐标变换的输出。

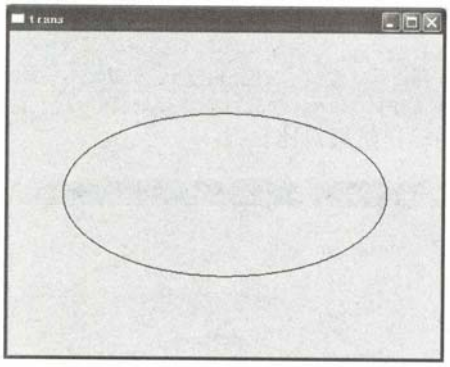

图 13-19　未进行坐标变换的输出

13.9.1　缩放

根据以下公式：

```
x' = x * eM11 + y * eM21 + eDx,
y' = x * eM12 + y * eM22 + eDy,
```

如果需要将原图形缩小一半，对 XFORM 参数进行如下设置：

```
// 缩小为 1/2
xForm.eM11 = (FLOAT) 0.5;
xForm.eM12 = (FLOAT) 0.0;
xForm.eM21 = (FLOAT) 0.0;
xForm.eM22 = (FLOAT) 0.5;
xForm.eDx  = (FLOAT) 0.0;
xForm.eDy  = (FLOAT) 0.0;
SetWorldTransform(hDC, &xForm);
Ellipse(hDC, (rect.right / 2 - 200), (rect.bottom / 2 + 100),
       (rect.right / 2 + 200), (rect.bottom / 2 - 100));
```

如图 13 -20 所示，坐标变换引起输出变小。

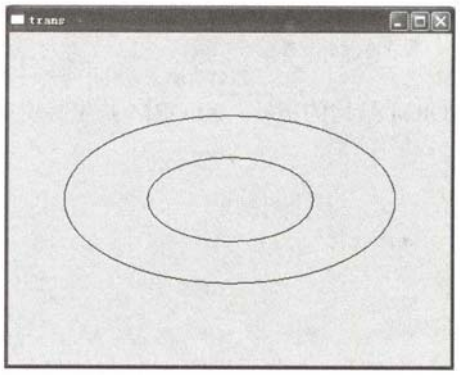

图 13-20　坐标变换——图形缩小

13.9.2　旋转

在代码中设置如下：

```
// 逆时针 30°
xForm.eM11 = (FLOAT) 0.8660;
xForm.eM12 = (FLOAT) 0.5000;
xForm.eM21 = (FLOAT) -0.5000;
xForm.eM22 = (FLOAT) 0.8660;
xForm.eDx  = (FLOAT) 0.0;
xForm.eDy  = (FLOAT) 0.0;
SetWorldTransform(hDC, &xForm);
Ellipse(hDC, (rect.right / 2 - 200), (rect.bottom / 2 + 100),
       (rect.right / 2 + 200), (rect.bottom / 2 - 100));
```

图 13-21 是坐标旋转后的输出结果对比。

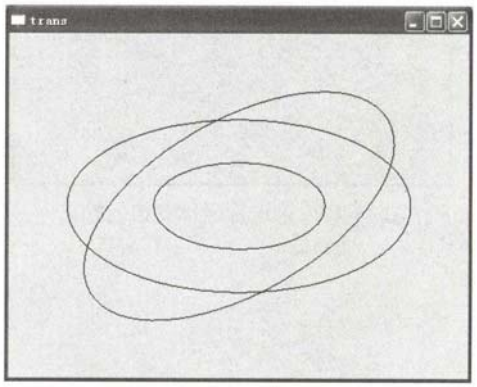

图 13-21　坐标变换——旋转

13.10　调色板

显示器上显示的图像是由一个个像素组成的，像素的颜色是基于 RGB 模型的，每一个像素的颜色由红(B)、绿(G)、蓝(B)三原色组合而成。如果每种原色用 8 位表示，这样一个的颜色就是 24 位的，因此可以同时显示 224 种颜色。24 位的颜色通常被称作真彩色，人眼看使用真彩色显示的图像与看真实的图像几乎感觉不到区别。

但是，真彩色的显示需要大量的视频内存，每一个像素点 3 个字节，1024×768 分辨率的图像就需 2359296，约 2MB 的显示内存（还不包括其他辅助的信息）。不在需要真彩色显

示的情况下，可以使用调色板来限制颜色的数目。

调色板用数组表示当前可以被显示和绘制到输出设备上的颜色，实际上是一个有若干个表项的 RGB 颜色表，颜色表的每项是一个 24 位的 RGB 颜色值。而显示设备上的每一个像素点，存储的不是 24 位颜色值，而是调色板颜色表数组的索引。这样达到了减少资源消耗的目的。

调色板分为系统调色板和逻辑调色板两种。Windows 是一个多任务操作系统，可以同时运行多个程序。如果有几个程序都要设置调色板，就有可能产生冲突。为了避免这种冲突，Windows 使用逻辑调色板来作为使用颜色的应用程序和系统调色板（物理调色板）之间的缓冲。

每一个 DC 都拥有一个逻辑调色板，类似于可以为 DC 选择画刷等 GDI 对象一样，也可以为 DC 选择调色板。

1．创建和设置调色板

使用调色板的基本过程时，先填充 LOGPALETTE 结构，设置好颜色数组大小和颜色数组中每个颜色的 RGB 值。然后调用 CreatePalette 创建逻辑调色板，再将调色板选入 DC，然后调用 RealizePalette 函数将调色板映射到物理调色板（可以在窗口得到焦点，收到 WM_QUERYNEWPALETTE 消息时调用）。

(1) CreatePalette 函数。

CreatePalette 函数创建一个逻辑调色板，得到逻辑调色板句柄，在创建逻辑调色板前，需在 LOGPALETTE 结构中定义好逻辑调色板的大小和像素等。CreatePalette 函数原犁如下：

```
HPALETTE CreatePalette(
  CONST LOGPALETTE *lplgpl   // 逻辑调色板
);
```

(2) LOGPALETTE 结构。

LOGPALEITE 结构是调色板的重要结构，保存了像素点的个数和像素点 REG 值数组，结构如下：

```
typedef struct tagLOGPALETTE {
  WORD        palVersion;
  WORD        palNumEntries;
  PALETTEENTRY palPalEntry[1];
} LOGPALETTE;
```

其中 PALETTEENTRY 结构如下：

```
typedef struct tagPALETTEENTRY {
  BYTE peRed;
  BYTE peGreen;
  BYTE peBlue;
  BYTE peFlags;
} PALETTEENTRY;
```

(3) SelectPalette 函数。

SelectPalette 函数将逻辑调色板选入 DC，以替换 DC 原有的调色板，函数原型如下：

(4) RealizePalette 函数。

RealizePalette 函数将 DC 的逻辑调色板实现到系统调色板，以使 DC 的输出使用定制的调色板，函数原型如下：

```
HPALETTE SelectPalette(
  HDC hdc,                  // DC 句柄
  HPALETTE hpal,            // 逻辑调色板句柄
  BOOL bForceBackground     // 前景或背景模式
);
```

```
UINT RealizePalette(
  HDC hdc   // DC句柄
);
```

2．与系统调色板相关的窗口消息

(1)当窗口获得焦点时，会收到WM_QUERYNEWPALETTE消息，如果窗口要使用调色板，则在收到该消息时应该实现自己的逻辑调色板并重绘窗口。如果窗口实现了逻辑调色板，那么窗口处理函数应返回TRUE，否则必须返回FALSE。总之WM_QUERYNEWPALETTE消息为活动窗口提供了实现前景调色板的机会。

(2)当活动窗口实现其前景调色板并改变了系统调色板时，Windows 会向包括活动窗口在内的所有窗口发送WM PALETTECHANGED消息，在该消息的wParam参数中包含了改变系统调色板的窗口的句柄。其他窗口如果使用了自己的逻辑调色板，那么应该重新实现其逻辑调色板，并重绘窗口。这是因为系统调色板已经被改变了，必需重新建立调色板映射表并重绘，否则可能会显示错误的颜色。WM~PALETTECHANGED消息为窗口提供了适应系统调色板变化的机会。

3.判断DC是否支持调色板

如果在程序中使用调色板，首先需要了解所操作的DC是不是支持调色板，使用RASTERCAPS参数调用GetDeviceCaps可以判断，返回值是否包括RC_PALETTE。使用GetDeviceCaps函数还可以获取调色板的大小。

函数InPaletteBasedVideoMode可以判断DC是否支持调色板。

```
BOOL InPaletteBasedVideoMode(HDC hdc)
{
    BOOL bRet;
    int iRasCaps;
    iRasCaps = GetDeviceCaps(hdc, RASTERCAPS);
    bRet = (iRasCaps & RC_PALETTE) ? TRUE : FALSE;
    ReleaseDC(NULL, hdc);
    return bRet;
}
```

BITMAPINFOHEADER的biBitCount成员说明了一个像素点是使用几位数据来表示，在位数较低的时候，会存在调色板，而在32、24位真彩色位图中，没有使用调色板。直接在像素点数组中保存每个像素点的RGB颜色。位图文件相关结构的设置有多种情况，具体使用时可参考 SDK 文档。在编程实现保存位图文件时，文件头以外的信息可以直接由 API CreateDIBitmap函数处理得到。

4．调色板与位图文件

内存中的位图数据可以保存为位图文件(. bnlp)。一般情况下，位图文件的格式最开始是一个位图文件头(BITMAPFILEHEADER)，然后是位图信息头(BITMAPNFOHEADER)，然后是RGBQUAD结构的数组，与调色板PALETTEENTRY结构的颜色数组类似，再之后是像素点的颜色信息，也就是像素点的颜色对应的RGBQUAD数组中的索引。RGBQUAD结构定义如下：

```
typedef struct tagRGBQUAD {
  BYTE    rgbBlue;
  BYTE    rgbGreen;
  BYTE    rgbRed;
  BYTE    rgbReserved;
} RGBQUAD;
```

第 14 章　网络通信与配置

网络通信程序接口有非常重要的应用。本章介绍网络通信接口

目前的计算机通信中存在多种协议，TCP/IP 是目前互联网中最常用的协议，几乎所有计算机通信都是使用 TCP/IP 协议。TCP/IP 并不是一个十分明确的概念，不仅仅指 TCP 协议和 IP 协议，而是一系列协议，泛指当前计算机网络中所通用的网络协议簇。

网络协议是分层的，具有不同的功能，TCP/IP 协议一般分为 4 层，OSI 模型为 7 层。下面是目前常用协议的层次划分。

数据链路层：Ethemet、PPPoE、APR 等。

网络层：IP、ICMP、IGMP 等。

传输层：TCP、UDP 等。

应用层：Telnet、FTP、HTTP、SMTP、SNMP 等。

每一层协议的功能不同，一般下层协议为上层协议提供支持。如 IP 协议使数据包能够到达指定目的地址，在此基础上 TCP 协议保证了数据传输的正确性、连接续和实时性。而上层其他应用协议都是建立在传输层的基础上，更多地关注于所传输数据的内容和意义。

由于协议内容比较多，操作过程比较复杂，如果由应用程序去处理与网络通信协议相关的内容就会使应用程序的开发过于复杂。因此 Windows 系统已经实现了这些协议，应用程序只需要调用系统提供的函数接口就可以使用这些协议进行通信。

Socket 就是系统提供的一种使用传输层的网络协议进行数据传输的程序设计接口规范，经过 Socket 封装的应用程序不再需要处理与 TCP 协议相关的内容。诸如连接时的三次握手、分包、包头解析、重传、滑动窗口等行为应用程序都不需关注，而只需要像使用系统中的其他 I / O 接口一样进行输入和输出操作即可。

本章 14.1 节将先介绍 Socket 通信的接口函数及程序设计的相关内容。

14.2 节将介绍 IP Helper。IP Helper 是 Windows 系统与 IP 协议相关的配置和管理的重要接口、通过 IP Helper 可以实现一些在网络通信软件中常用的功能，比如获取本机 IP 网关，配置、获得统计信息、网卡数量和连接状态等。

Windows 除了为网络通信程序设计提供 Socket 这样的传输层接口外，还提供了很多应用层协议接口，包括 Http、SNMP 等。此外还有 DNS、DHCP 等常用网络协议，由于篇幅限制，本书不做详细介绍。

14.1　Socket通信

Socket 套接字是使用传输程序协议（TCP、UDP 等）进行网络通信的程序设计接口。

由于基于连接的 Socket 通信使用范围更为广泛，本章只介绍基于连接的 Socket 通信。

Socket 程序分为服务端与客户端，服务端程序监听端口，等待客户端程序的连接。客户端程　序发起连接，等待服务端的响应。客户端程序需要知道服务端程序的 IP 地址和监听端口。

服务端与客户端在建立连接后，双方都可以发送和接收数据。

o 头文件和库文件

在介绍 Socket 程序示例前先介绍 Socket 程序设计所需的头文件和库文件。

当前的 Windows Socket 接口是 Windows Sockets 2，所有接口函数都由 Ws2：32.dll 导出。在　程序设计时，相关数据类型、结构定义、函数声明等位于头文件 winsock2.h 中，

编译时需要包含，　编译时需链接到库 Ws2_32.lib。

14.1.1　客户端

客户端的功能是连接到服务端。Socket 的连接是由客户端发起的。客户端在连接服务端的指定端口时如果指定端口开放，而且服务端响应了客户端的连接请求，连接就会建立。客户端在连接前需知道服务端程序所监听的端口和服务端程序所在主机的 IP 地址。

在连接建立后，需要得到 Socket,然后就可以向服务端发送数据，或者从服务端接收数据。

1．客户端程序过程

一个 Socket 客户端程序的典型过程如下。

(1)客户端程序在运行后，首先需要使调用 WSAStartup 函数，确保进程加载 socket 应用程序所必须的环境和库文件，如 Ws2_32.dll。

(2)调用函数 Socket 创建 SOCKET，在创建时需指定使用的网络协议、连接类型等。

(3)填充 SOCKADDR 结构，指定服务端的地址、端口等。

(4)调用 connect 函数连接到服务端。

(5)如果连接成功，就可以使用 send 和 recv 函数发送和接收数据。

(6)在数据传输完成后，可调用 closesocket 函数关闭 Socket。

(7)调用 WSACleanup 函数释放资源。

2．API 函数和结构

客户端程序常使用如下 API 函数。

(1) WSAStartup 函数。

WSAStartup 函数的功能是加载 Ws2_32.dll 等 Socket 程序运行的环境。在程序初始化后，Socket　程序运行所依赖的动态链接库不一定已经加载，WSAStartup 保证了 Socket 动态链接库的加载。

```
int WSAStartup(
  __in        WORD wVersionRequested,
  __out       LPWSADATA lpWSAData
);
```

wVersionRequested 参数：是 Socket 程序库的版本，一般使用 MAKEWORD(2,2)宏。

lpWSAData 参数：输出参数，指向 WSADATA 结构的指针，用于返回 Socket 库初始化的信息。

WSAStartup 的返回值可以判断程序是否调用成功。

(2) WSACleanup 函数。

与 WSAStartup 的功能相反，WSACleanup 释放 Ws2_32.dll 库，函数无参数。

```
int WSACleanup(void);
```

(3) Socket 函数。

Socket 函数的功能是建立一个绑定到指定协议和传输类型的 Socket。

```
SOCKET WSAAPI socket(
  __in        int af,
  __in        int type,
  __in        int protocol
);
```

af (af 是 address family 的缩写)参数：指定了这个 Socket 使用哪一种类型的网络地址,可以是 IPv4 (AF INET),也可以是 IPv6 (AF INET6)、蓝牙(AF BTM)、NetBios (AF NETBIOS)等。

type 参数：指的传输类型，可设置为 SOCK_STREAM、SOCK DGRAM、SOCK- RAW、

SOCK_SEQPACKET 等。SOCK. -STREAM 类型是基于连接的(TCP)，所收的数据是数据流形式的，传输层的数据包已经经过处理， SOCK DGRAM 是基于报文的(UDP)。如果指定为 SOCK_ RAW，那么可以建立原始套接字，所收到的数据是以数据包（包括包头）的形式存在的。

protocol 参数：指明了使用的传输协议，常用的是 IPPROTO_TCP 和 IPPROTO_UDP。

(4) connect 函数。

connect 函数的功能是与服务端建立连接。这个函数只能由服务端程序调用。

```
int connect(
  __in        SOCKET s,
  __in        const struct sockaddr* name,
  __in        int namelen
);
```

s 参数：由 socket 函数建立的套接字。

name 参数：指向 sockaddr 结构的指针，包括了所要连接的服务端的地址和端口等。

namelen: name 参数指向的 sockaddr 结构的长度。

(5) sockaddr、SOCKADDR、sockaddr_in、SOCKADDR IN 结构。

这几个结构用于表示地址和端口。在 IPv4 下，这几个结构可以通用。

```
typedef struct sockaddr {
   u_short sa_family;
   CHAR sa_data[14];
} SOCKADDR, *PSOCKADDR, FAR *LPSOCKADDR;

typedef struct sockaddr_in {
   short   sin_family;
   USHORT sin_port;
   IN_ADDR sin_addr;
   CHAR sin_zero[8];
} SOCKADDR_IN, *PSOCKADDR_IN;
```

结构的设置方法参见本节实例。

(6) send 函数。

send 函数的功能是向连接的另一端发送数据，函数原型如下：

```
int send(
  __in        SOCKET s,
  __in        const char* buf,
  __in        int len,
  __in        int flags
);
```

参数 s 是由 socket 函数返回的套接字。buf 指向需要发送的数据，len 是发送数据的长度。flags 参数表示的发送的方法，其值可以是 MSG_- DONTROUTE、MSG_OOB 等，一般可设置为 0。

如果 send 成功，则返回实际发送的数据；如果失败，返回 SOCKET ERROR。

(7) recv。

recv 函数的功能是从连接的另外一端接收数据，函数原型如下：

```
int recv(
  __in        SOCKET s,
  __out       char* buf,
  __in        int len,
  __in        int flags
);
```

参数 s 是由 socket 函数返回的套接字，buf 指向用于保存接收数据的缓存，len 是接收缓存的大小长度。flags 参数表示接收的方法，其值可以是 MSG DONIROUTE、MSG_OOB 等，一般可设置为 0。

系统在实现 TCP 协议时都为数据的接收保留了缓存。协议收到数据包后，解包并将数据放入缓存中，直到 recv 函数将数据接收。如果 recv 函数长时间不接收数据，在协议的缓存

存满之后，对方的 send 函数就不能再发送数据（更详细的原理可参考 TCP 协议的滑动窗口机制）。

(8) colsesocket。

colsesocket 函数用于关闭 Socket，函数原型如下：

```
int closesocket(
  __in        SOCKET s
);
```

3. 实例 14-1 Socket 客户端

本示例是一个客户端程序，连接 127.0.0.1（本机）的 10000 端口（本实例服务端程序也运行于本机，监听 10000 端口，可以将服务端程序运行在其他主机上，然后将 IP 地址修改为服务端程序所在的主机的 IP 地址。服务端程序见实例 14-2，在连接上后，向服务端发送一个字符串数据，然后等待服务端的返回数据并接收，直至服务端发送完数据后，关闭连接。

如果运行时输入了“-d”参数，那么会发送″download file”，否则发送“get information”。

```
/* ************************************
* client.c Socket 通信
*************************************/
/* 头文件 */
#include <stdio.h>
#include "winsock2.h"
/* 常量 */
#define RECV_BUFFER_SIZE 8192

/************************************
* main
* 功能 Socket 通信客户端
*************************************/
void main(int argc, char* argv[])
{
     // 变量定义
     SOCKADDR_IN clientService;// 地址
     SOCKET ConnectSocket;// Socket
     WSADATA wsaData;// 库
     LPVOID recvbuf;// 接收缓存
     int bytesSent;
     int bytesRecv = 0;
```

```
char sendbuf[32] = "get information";// 默认发送的数据
// 初始化 Socket 库,      保证 ws2_32.dll 已经加载
int iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
if (iResult != NO_ERROR)
     printf("Error at WSAStartup()\n");
// 创建 Socket
ConnectSocket = socket(AF_INET, // IPv4
     SOCK_STREAM, // 顺序的、可靠的、基于连接的、双向的数据流通信
     IPPROTO_TCP// 使用 TCP 协议
     );
if (ConnectSocket == INVALID_SOCKET)
{
     printf("Error at socket(): %ld\n", WSAGetLastError());
     WSACleanup();
     return;
}
// 设置服务端的通信协议、IP 地址、端口
clientService.sin_family = AF_INET;
clientService.sin_addr.s_addr = inet_addr( "127.0.0.1" );
clientService.sin_port = htons( 10000 );
// 连接到服务端
if ( connect(
     ConnectSocket, // socket
     (SOCKADDR*) &clientService, // 地址
     sizeof(clientService) // 地址的大小
     ) == SOCKET_ERROR)
{
     printf( "Failed to connect(%d)\n",WSAGetLastError() );
     WSACleanup();
     return;
}
// 准备发送数据
// 如果输入参数是“-d”，那么发送的数据是“download file”，否则发送的数据是“get information”
if(argc ==2 && (!lstrcmp(argv[1], "-d")))
{
     lstrcpyn(sendbuf, "download file", 32);
}
// 向服务端发送数据
bytesSent = send( ConnectSocket, // socket
     sendbuf,// 发送的数据
     lstrlen(sendbuf)+1,// 数据长度
     0 );// 无标志
if(bytesSent == SOCKET_ERROR)
{
     printf( "send error (%d)\n", WSAGetLastError());
     closesocket(ConnectSocket);
     return 1;
}
printf( "Bytes Sent: %ld\n", bytesSent );
// 准备接收数据
recvbuf = HeapAlloc(GetProcessHeap(), 0, RECV_BUFFER_SIZE);
// 循环接收
while( bytesRecv != SOCKET_ERROR )
{
     bytesRecv = recv( ConnectSocket, // socket
          recvbuf, // 接收数据缓存
          RECV_BUFFER_SIZE,// 缓存大小
          0 );// 无标志
     if ( bytesRecv == 0 )
     {
          printf( "Connection Closed.\n");
```

```
                break;
            }
            // TODO, 处理接收的数据, 这里只简单地将收到的数据大小显示
            printf( "Bytes Recv: %ld\n", bytesRecv );
        }
        HeapFree(GetProcessHeap(), 0, recvbuf);
        WSACleanup();
        return;
    }
```

编译生成 sock cnt.exe。

先运行实例 14-2 的服务端程序，然后运行本程序：

```
sock_cnt.exe
```

如图 14-1 所示，发送了 16 个字节(get information)，然后服务端回复了 20 个字节后，关闭了 Socket，客户端程序退出。

图 14-1　sock_cnt.exe 运行效果

再运行：

```
sock_cnt.exe -d
```

如图 14-2 所示，这次客户端向服务端发送了 14 个字节(download file)，然后收到了大量的数据。完成后，服务端关闭了 Socket，客户端退出。

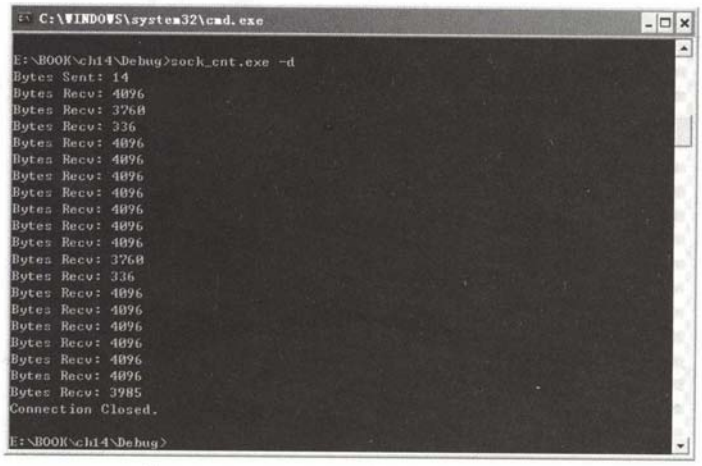

图 14-2　sock_cnt.exe 数据接收运行效果

分析运行结果，发现一个现象。虽然在客户端程序中指定的接收缓存大小是 8KB。但是每次 recv 函数都没有收到 8KB。这是与 recv 函数接收的机制有关的。当调用 recv 函数后，如果协议栈缓存中的数据还没有 recv 接收缓存的大，那么缓存中的数据有多少 recv 函数接收多少，然后返回。如果当前协议缓存中没有数据，那么一旦有数据到达就接收到达的数据，然后返回。如果协议缓存中的数据大于 recv 接收缓存，那么 recv 将接收缓存填满之后返回。

对比服务端的程序，由于发送端每次发送 4KB 数据，而且在发送前还会从文件中读取数据，那么发送的速度较接收的速度慢，就会造成这种情况。

可以在客户端程序的接收循环中加入 Sleep 函数，有意造成客户端接收的速度较发送的

速度慢，再去连接，就会发现客户端除了最后一次，每次接收的数据都是 8KB，如图 14-3 所示。

图 14-3　接收速度与发送速度较慢的情况

最后一次接收的数据不足 8KB 是由于服务端只发送了这么多数据，最后一次 recv 时将协议缓存中所有的数据都接收完。当服务端关闭连接后，协议缓存中所有的数据都已经接收完成之后再接收数据，就会返回连接已经关闭。

14.1.2　服务端

服务端的功能在指定的端口上监听,等待客户端的连接。在连接建立后可使用 send 和 recv 函数发送、接收数据。

1．服务端程序的一般过程

一般情况下，Socket 程序服务端过程如下。

(1)程序在运行后，首先需要使调用 WSAStartup 加载 Ws2_32.dll。

(2)调用函数 socket 创建用于监听的 SOCKET，在创建时需指定使用的网络协议、连接类型等。

(3)1 调用 bind 函数将 Socket 绑定到网络地址和端口。

(4)调用 listen 函数开始监听。

(5)调用 accept 函数等待客户端连接。在客户端连接后，accept 函数返回，得到连接 Socket。在 accept 函数返回后，可立即再调用，以处理其他客户端的连接。

(6)得到连接 Socket 后，可调用 send 和 recv 发送、接收数据。

(7)在数据传输完成后，可调用 closesocket 函数关闭 Socket。

(8)调用 WSACleanup 函数释放 DLL。

2．API 函数和结构

bind、listen、accept 函数是服务端才会使用到的函数。服务端程序在调用 socket、send、recv 函数时，方法与客户端一样。

(1) bind。

bind 函数的功能是将 Socket 与网络地址和端口绑定起来，函数原型如下：

```
int bind(
  __in        SOCKET s,
  __in        const struct sockaddr* name,
  __in        int namelen
);
```

参数 s 是由 socket 函数所创建的套接字，name 参数表示网络地址的结构。namelen 函数是 name 参数所指向结构的大小。

(2) listen。

listen 函数的功能是将 Socket 的状态设置为监听，以使客户端程序可以进行连接，函数原型如下：

```
int listen(
  __in        SOCKET s,
  __in        int backlog
);
```

s 是套接字，backlog 参数指定了这个监听最大的连接数，设置为 SOMAXCONN（最大的可用连接数）。

(3) accept。

accept 函数的功能是接收客户端的连接，accept 函数直到客户端有连接后才会返回。函数原型如下：

```
SOCKET accept(
  __in        SOCKET s,
  __out       struct sockaddr* addr,
  __in_out    int* addrlen
);
```

其中 addr 参数是输出参数，保存了连接到本服务端的客户端的相关地址信息，包括 IP、端口等。addrlen 是 addr 参数所指向的结构的长度。

3．实例 14-2 Socket 服务端

本实例是一个服务端程序，实例 14-1 的客户端程序所连接的服务端程序由本实例编译生成。

如代码所示，在 maln 函数中，依次调用了 WSAStartup、socket、bind、listen，最后进行 while 循环，在循环中调用了 accept 函数。

如果有客户端连接，accept 函数返回，然后创建一个线程用于处理与客户端的数据发送、接收。然后循环调用 accept 函数，以等待其他客户端的连接。

线程程序 CommunicationThread 首先会接收客户端发送的数据，根据数据内容的不同回复不同的信息，在数据发送完成后，关闭 Socket。

```
/* ************************************
* server.c Socket 通信
**************************************/
/* 头文件 */
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
/* 常量 */
#define DEFAULT_PORT "10000" // 端口
#define MAX_REQUEST 1024 // 接收数据的缓存大小
#define BUF_SIZE 4096 // 发送数据的缓存大小
/**************************************
* CommunicationThread
* 功能：用于接收和发送数据的线程
*       为每一个连接的客户端创建一个接收发送数据的线程，
*       可以使用多个客户端同时连接到服务端
```

```
* 参数    lpParameter、SOKCET
************************************/
DWORD WINAPI CommunicationThread(
                              LPVOID lpParameter
                              )
{
     DWORD dwTid = GetCurrentThreadId();
     // 获得参数 sokcet
     SOCKET socket = (SOCKET)lpParameter;
     // 为接收数据分配空间
     LPSTR szRequest = HeapAlloc(GetProcessHeap(),0, MAX_REQUEST);
     int iResult;
     int bytesSent;// 用于保存 send 的返回值，即实际发送的数据的大小

     // 接收数据
     iResult = recv(socket, // socket
          szRequest, // 接收缓存
          MAX_REQUEST, // 缓存大小
          0);// 标志
     if (iResult == 0)// 接收数据失败，连接已经关闭
     {
          printf("Connection closing...\n");
          HeapFree(GetProcessHeap(), 0 ,szRequest);
          closesocket(socket);
          return 1;
     }
     else if (iResult == SOCKET_ERROR)// 接收数据失败，Socket 错误
     {
          printf("recv failed: %d\n", WSAGetLastError());
          HeapFree(GetProcessHeap(), 0 ,szRequest);
          closesocket(socket);
          return 1;
     }
     else if (iResult > 0) // 接收数据成功
     {
          // 显示接收到的数据
          printf("\tCommunicationThread(%d)\tBytes received: %d\n", dwTid, iResult);
          printf("\tCommunicationThread(%d)\trequest string is (%s)\n",dwTid, szRequest);

          // 如果接收到的数据是 “download file”
          if (lstrcmpi(szRequest, "download file") == 0)
          {
               // 读取文件 download.txt 将发送
               HANDLE hFile;
               LPVOID lpReadBuf; // 发送缓存
               DWORD dwBytesRead;
               DWORD dwFileSize;
               DWORD dwSendFile = 0;
               hFile = CreateFile("download.txt",
                    GENERIC_READ,
                    FILE_SHARE_READ,
                    NULL,
                    OPEN_EXISTING,
                    FILE_ATTRIBUTE_NORMAL,
                    NULL);

               if (hFile == INVALID_HANDLE_VALUE)
               {
                    printf("\tCommunicationThread\tCould not open file (error %d)\n",
                         GetLastError());
                    send(socket, "error", 6, 0);
                    closesocket(socket);
```

```
            return 1;
        }
        // 分配发送数据缓存
        lpReadBuf = HeapAlloc(GetProcessHeap(), 0 , BUF_SIZE);
        // 获取文件大小
        dwFileSize = GetFileSize(hFile, NULL);
        // 循环发送
        while(1)
        {
            // 读文件到缓存
            if(!ReadFile(hFile, lpReadBuf, BUF_SIZE, &dwBytesRead, NULL))
            {
                printf("\tCommunicationThread\tCould not read from file (error %d)\n",
                    GetLastError());
                closesocket(socket);
                CloseHandle(hFile);
                return 1;
            }
            // 发送读取的文件数据
            bytesSent = send(socket, lpReadBuf, dwBytesRead, 0);
            if( bytesSent == SOCKET_ERROR)
            {
                printf("\tCommunicationThread\tsend error %d\n",
                    WSAGetLastError());
                closesocket(socket);
                CloseHandle(hFile);
                return 1;
            }
            // 显示发送数据的大小
            printf("\tCommunicationThread(%d)\tsend %d bytes\n", dwTid, bytesSent);
            // 累加，已经发送的大小
            dwSendFile += dwBytesRead;
            // 如果所有文件数据都已经发送
            if(dwSendFile == dwFileSize)
            {
                printf("\tCommunicationThread\tFile download ok\n");
                break;// 退出循环
            }
        }
        // 释放内存、关闭连接、关闭文件
        HeapFree(GetProcessHeap(), 0 , lpReadBuf);
        CloseHandle(hFile);
        closesocket(socket);
    }
    // 如果接收到的数据是“get information”
    else if (lstrcmpi(szRequest, "get information") == 0)
    {
        // 发送数据
        bytesSent = send(socket, // socket
            "this is information", // 数据
            lstrlen("this is information")+1, // 数据长度
            0);// 标志
        // 判断是否成功
        if( bytesSent == SOCKET_ERROR)
        {
            printf("\tCommunicationThread\tsend error %d\n",
                WSAGetLastError());
            closesocket(socket);
            return 1;
        }
        printf("\tCommunicationThread(%d)\tsend %d bytes\n",dwTid, bytesSent);
```

```
            }
            else// 收到未知数据
            {
                printf ("unreferenced request\n");
            }
        }
        // 释放接收数据缓存，关闭 Socket
        HeapFree(GetProcessHeap(), 0 ,szRequest);
        closesocket(socket);
        return 0;
}

/*************************************
* int __cdecl main(void)
* 功能      Socket 服务端主函数
*************************************/
int __cdecl main(void)
{
        WSADATA wsaData;
        SOCKET ListenSocket = INVALID_SOCKET;// 监听 Socket
        SOCKET ClientSocket = INVALID_SOCKET;// 连接 Socket
        struct addrinfo *result = NULL, hints;
        int iResult;// 保存返回结果

        // 初始化 Winsock，保证 Ws2_32.dll 已经加载
        iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
        if (iResult != 0)
        {
            printf("WSAStartup failed: %d\n", iResult);
            return 1;
        }
        // 地址
        ZeroMemory(&hints, sizeof(hints));
        hints.ai_family = AF_INET;
        hints.ai_socktype = SOCK_STREAM;
        hints.ai_protocol = IPPROTO_TCP;
        hints.ai_flags = AI_PASSIVE;

        // 获取主机地址，保证网络协议可用等
        iResult = getaddrinfo(NULL, // 本机
            DEFAULT_PORT, // 端口
            &hints, // 使用的网络协议、连接类型等
            &result);// 结果
        if ( iResult != 0 )
        {
            printf("getaddrinfo failed: %d\n", iResult);
            WSACleanup();
            return 1;
        }
        // 创建 Socket，用于监听
        ListenSocket = socket(
            result->ai_family, // 网络协议，AF_INET，IPv4
            result->ai_socktype, // 类型，SOCK_STREAM
            result->ai_protocol);// 通信协议，TCP
        if (ListenSocket == INVALID_SOCKET)
        {
            printf("socket failed: %ld\n", WSAGetLastError());
            freeaddrinfo(result);
            WSACleanup();
            return 1;
        }
```

```
    // 绑定到端口
    iResult = bind( ListenSocket, result->ai_addr, (int)result->ai_addrlen);
    if (iResult == SOCKET_ERROR)
    {
        printf("bind failed: %d\n", WSAGetLastError());
        freeaddrinfo(result);
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    printf("bind\n");

    freeaddrinfo(result);// reuslt 不再使用

    // 开始监听
    iResult = listen(ListenSocket, SOMAXCONN);
    printf("start listen......\n");
    if (iResult == SOCKET_ERROR)
    {
        printf("listen failed: %d\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    while (1)
    {
        // 接收客户端的连接，accept 函数会等待，直到连接建立
        printf("ready to accept\n");
        ClientSocket = accept(ListenSocket, NULL, NULL);
        // accept 函数返回，说明已经有客户端连接
        // 返回连接 Socket
        printf("accept a connetion\n");
        if (ClientSocket == INVALID_SOCKET)
        {
            printf("accept failed: %d\n", WSAGetLastError());
            closesocket(ListenSocket);
            break;// 等待连接错误，退出循环
        }
        // 为每一个连接创建一个数据发送的接收线程
        // 使服务端又可以立即接收其他客户端的连接
        if(!CreateThread(
            NULL,
            0,
            CommunicationThread, // 线程函数
            (LPVOID)ClientSocket, // 将 Socket 作为参数
            0,
            NULL))
        {
            printf("Create Thread error (%d)", GetLastError());
            break;
        }
    }
    // 循环退出，释放 DLL
    WSACleanup();
    return 0;
}
```

运行实例，然后如实例 14-1 所示过程运行客户端。

服务端先进行了绑定，然后开始监听，之后调用 accept。在客户端运行后，连接建立，新的线程创建，然后完成循环再调用 accept 等待服务端的再次连接。

如图 14-4 所示为先运行客户端，然后再带“-d”参数运行客户端后服务端的输出。所有打印出“CommunicationThread”字样的都是 CommunicationThread 函数调用的打印输出。同时还输出的所在的线程的 TID，如图 14-4 所示。

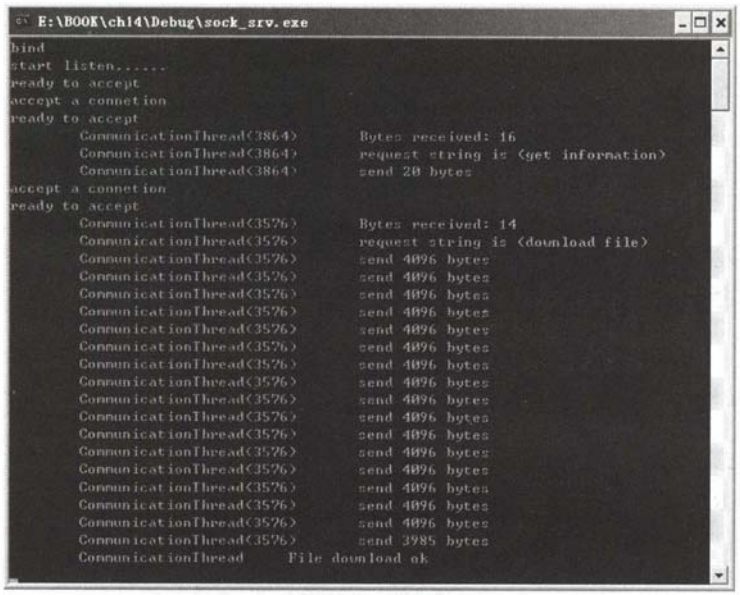

图 14-4　服务端接收多个客户端的连接

可以下载一下较大的文件，在文件下载的过程中再运行客户端，以检测是否可以同时响应多个客户端的请求。如图 14-5 所示，服务端线程 2916 在处理一个客户端下载请求，这里另一个客户端连接，于是创建了线程 3672，处理了新的客户端的请求的同时，原来的客户端的请求仍然在运行，如图 14-5 所示。

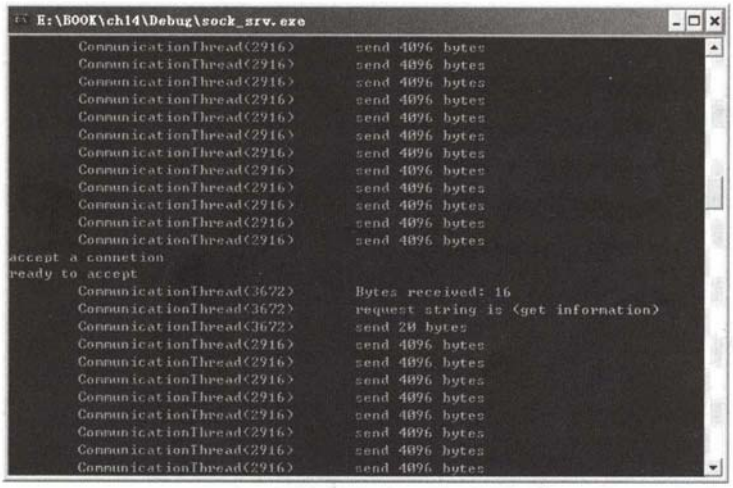

图 14-5　多个客户端同时连接到服务端

14.1.3　处理并发的客户端连接

服务端在监听时需指定一个端口，这个端口可以接收多个客户端的连接。客户端在连接服务端时，本机的端口不需要指定，系统会自动选取一个当前不用的端口与服务端的固定端口连接。

在同一时刻可以有多个不同主机上的客户端连接到服务端，一个主机上也可以同时有多个客户端连接到同一个服务端，但是在建立连接时，socket 函数会为客户端分配不同的端口。客户端不会使用同一端口和同一服务端建立多次连接。

服务端程序使用 accept 函数接收服务端的连接。因此如果服务端需要有与多个客户端

连接时，在 accept 函数返回，一个连接已经建立后需要立即再调用 accetp 函数，等待其他客户端的连接。如果客户端连接时，而服务端程序此时如果没有调用 accept 函数，那么连接不会成功建立。

因此处理并发客户端连接的服务端必定是多线程的。监听程序最好使用单独的线程，而且为了对每个客户端的请求都能立即响应，每个与客户端的连接最好建立至少有一个线程处理数据的发送和接收。

14.1.4 网络通信的异步I/O模式

进行网络数据的发送和接收不仅只有 send 和 recv 函数可以使用，还可以使用 sentto、recvform、WSARecv、WSARecvEx 等。

WSARecv 可以使用异步模式进行网络 I/O。

```
int WSARecv(
  __in         SOCKET s,
  __in_out     LPWSABUF lpBuffers,
  __in         DWORD dwBufferCount,
  __out        LPDWORD lpNumberOfBytesRecvd,
  __in_out     LPDWORD lpFlags,
  __in         LPWSAOVERLAPPED lpOverlapped,
  __in         LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

lpOverlapped 和 lpCompletionRoutine 函数是与重叠 I/O 有关的参数。说明网络通信也可以进行异步 I / O。

14.2 IP Helper

Internet Protocol Helper (IP Helper) API 可以用于获取和修改本机网络设置，包括本机的网卡，每个网上的物理地址、IP 地址、网关、MAC 等，以及当前主机的网络连接状态、TCP 数量情况等。几乎所有的网络参数都可以用 IP Helper 获取或进行设置。

获取本机网卡状态、lP、网关等网络配置

本节使用实例来介绍如何通过 IP Helper 来获得网络参数。先介绍进行 IP Helper 编程所需的头文件和库文件，以及一些常用的 IP Helper 函数。

1．头文件和库文件

IP Helper 函数都由动态链接库文件 Iphlpapi.dll 导出。

IP Helper 相关的数据类型定义、数据结构、API 函数的定义等都位于 iphlpapi.h 头文件，编写 IP Helper 应用程序需要引入头文件 iphlpapi.h，并链接到 Iphlpapi.lib 库文件。

2．主要 API 函数和结构

IP Helper 共有 100 多个 API 函数。本章只介绍其中最常用的几个。这些 API 函数的功能都是获取或设置网络参数、网卡设备等情况，只是所能获取的信息各有不同。

(1) GetNetworlParamS.

GetNetworlParams 函数原型如下：

```
DWORD GetNetworkParams(
  __out        PFIXED_INFO pFixedInfo,
  __in         PULONG pOutBufLen
);
```

所获得的信息保存在 PFIXED_INFO 结构中。

```
typedef struct {
  char HostName[MAX_HOSTNAME_LEN + 4];
  char DomainName[MAX_DOMAIN_NAME_LEN + 4];
  PIP_ADDR_STRING CurrentDnsServer;
  IP_ADDR_STRING DnsServerList;
  UINT NodeType;
  char ScopeId[MAX_SCOPE_ID_LEN + 4];
  UINT EnableRouting;
  UINT EnableProxy;
  UINT EnableDns;
} FIXED_INFO,  *PFIXED_INFO;
```

包括主机名、域名、DNS 服务器 IP、是否路由、是否代理、是否 DNS 等。

(2) GetInterfaceInfo。

GetInterfaceInfo 函数获取与网卡相关的信息。

```
DWORD GetInterfaceInfo(
  __out        PIP_INTERFACE_INFO pIfTable,
  __in_out     PULONG dwOutBufLen
);
```

返回值保存在 IP_INTERFACE INFO 结构中。

```
typedef struct _IP_INTERFACE_INFO {
  LONG NumAdapters;
  IP_ADAPTER_INDEX_MAP Adapter[1];
} IP_INTERFACE_INFO,  *PIP_INTERFACE_INFO;
```

如果有多个网卡，Adapter 是数组的第一个元素。

(3) GetIpAddrTable。

GetIpAddrTable 函数获得网络地址、掩码等。

```
DWORD GetIpAddrTable(
  __out        PMIB_IPADDRTABLE pIpAddrTable,
  __in_out     PULONG pdwSize,
  __in         BOOL bOrder
);
```

(4) GetIpStatistics。

GetIpStatistics 函数可以获取网络连接的统计信息，包括流量、连接等。

```
DWORD GetTcpStatistics(
  __out        PMIB_TCPSTATS pStats
);
```

(5) GetAdaptersInfo。

GetAdaptersInfo 函数所能获得的信息比较丰富，能够获得当前主机中每个网卡的设备名、描述信息、物理地址、IP、掩码、DNS 服务器、DHCP 服务器等。

```
DWORD GetAdaptersInfo(
  __out        PIP_ADAPTER_INFO pAdapterInfo,
  __in_out     PULONG pOutBufLen
);
```

IP_ADAPTER INFO 结构如下：

```
typedef struct _IP_ADAPTER_INFO {
  struct _IP_ADAPTER_INFO* Next;
  DWORD ComboIndex;
  char AdapterName[MAX_ADAPTER_NAME_LENGTH + 4];
  char Description[MAX_ADAPTER_DESCRIPTION_LENGTH + 4];
  UINT AddressLength;
  BYTE Address[MAX_ADAPTER_ADDRESS_LENGTH];
  DWORD Index;
  UINT Type;
  UINT DhcpEnabled;
  PIP_ADDR_STRING CurrentIpAddress;
  IP_ADDR_STRING IpAddressList;
  IP_ADDR_STRING GatewayList;
  IP_ADDR_STRING DhcpServer;
  BOOL HaveWins;
  IP_ADDR_STRING PrimaryWinsServer;
  IP_ADDR_STRING SecondaryWinsServer;
  time_t LeaseObtained;
  time_t LeaseExpires;
} IP_ADAPTER_INFO,  *PIP_ADAPTER_INFO;
```

3．实例 14-3 使用 lP Helper 函数获得网络参数

本实例使用上述 API 函数，获得本机网络配置参数等信息，然后显示。

```
/* ************************************
* iph.cpp  Ip Helper
**************************************/
/* 头文件 */
#include <stdio.h>
#include <windows.h>
#include "iphlpapi.h"
/* 全局变量 */
ULONG ulOutBufLen;
DWORD dwRetVal;
HANDLE hHeap = GetProcessHeap();
/* 宏定义*/
#define MyAlloc(size) HeapAlloc(hHeap, 0, size)
#define MyFree(ptr) HeapFree(hHeap, 0, ptr)

/************************************
* main
* 功能：演示 IP Helper 函数的使用方法
************************************/
void main(int ac, char* av[])
{
     // GetNetworkParams
     printf("------------------------\n");
     printf("GetNetworkParams\n\n");
     // 所需的变量
     FIXED_INFO *pFixedInfo;
     IP_ADDR_STRING *pIPAddr;
     // 分配内存
     pFixedInfo = (FIXED_INFO *) MyAlloc( sizeof( FIXED_INFO ) );
     ulOutBufLen = sizeof( FIXED_INFO );

     if( GetNetworkParams( pFixedInfo, &ulOutBufLen ) == ERROR_BUFFER_OVERFLOW )
     {
          // 空间不足，重新分配
          MyFree( pFixedInfo );
          pFixedInfo = (FIXED_INFO * ) MyAlloc ( ulOutBufLen );
     }
```

```
// 获取信息
if ( dwRetVal = GetNetworkParams( pFixedInfo, &ulOutBufLen ) != NO_ERROR )
{
     printf("Call to GetNetworkParams failed.\n");
}
else
{
     // 打印输出
     printf("\tHost Name: %s\n", pFixedInfo -> HostName);// 主机名
     printf("\tDomain Name: %s\n", pFixedInfo -> DomainName);// DNS 主机名
     printf("\tDNS Servers:\n");//DNS 服务器
     printf("\t\t%s\n", pFixedInfo -> DnsServerList.IpAddress.String);// IP, 字符串形式的

     pIPAddr = pFixedInfo -> DnsServerList.Next;// 备用 DNS
     while ( pIPAddr )
     {
          printf("\t\t%s\n", pIPAddr -> IpAddress.String);
          pIPAddr = pIPAddr -> Next;
     }
     if (pFixedInfo -> EnableRouting)// 是否路由
          printf("\tEnable Routing: Yes\n");
     else
          printf("\tEnable Routing: No\n");
     if (pFixedInfo -> EnableProxy)// 是否有代理
          printf("\tEnable Proxy: Yes\n");
     else
          printf("\tEnable Proxy: No\n");
     if (pFixedInfo -> EnableDns)// 是否有 DNS
          printf("\tEnable DNS: Yes\n");
     else
          printf("\tEnable DNS: No\n");
}
MyFree(pFixedInfo);// 释放内存
// GetNetworkParams
printf("------------------------\n");
printf("GetInterfaceInfo\n\n");
// 变量定义、分配内存空间
PIP_INTERFACE_INFO pInfo;
ulOutBufLen = sizeof(IP_INTERFACE_INFO);
pInfo = (PIP_INTERFACE_INFO)MyAlloc( ulOutBufLen );

while(1)
{
     dwRetVal = GetInterfaceInfo(pInfo, &ulOutBufLen);
     // 内在不足, 重新分配
     if(dwRetVal == ERROR_INSUFFICIENT_BUFFER)
     {
          MyFree( pInfo );
          pInfo = (IP_INTERFACE_INFO *)MyAlloc( ulOutBufLen );
     }
     else if (dwRetVal == NO_ERROR)
     {
          // 显示输出每个网卡的信息
          int i;
          printf("Num Adapters: %ld\n", pInfo->NumAdapters);
          for (i=0; i<pInfo->NumAdapters; i++)
          {
               printf("(%d)Adapter Name: %ws\n", i, pInfo->Adapter[i].Name);
               printf("(%d)Adapter Index: %ld\n", i, pInfo->Adapter[i].Index);
          }
          break;
     }
     else
          printf("GetInterfaceInfo: %u\n", GetLastError());
}
MyFree( pInfo );// 释放内存
// GetIpAddrTable
```

```
printf("----------------------\n");
printf("GetIpAddrTable\n\n");

// 变量定义、内存分配
MIB_IPADDRTABLE *pIPAddrTable;
DWORD dwSize;
in_addr IPAddr;
char *strIPAddr;

pIPAddrTable = (MIB_IPADDRTABLE*) MyAlloc( sizeof( MIB_IPADDRTABLE) );
dwSize = 0;
IPAddr.S_un.S_addr = ntohl(pIPAddrTable->table[1].dwAddr);
strIPAddr = inet_ntoa(IPAddr);
if (GetIpAddrTable(pIPAddrTable, &dwSize, 0) == ERROR_INSUFFICIENT_BUFFER)
{
     // 空间不足，重新分配
     MyFree( pIPAddrTable );
     pIPAddrTable = (MIB_IPADDRTABLE *) MyAlloc ( dwSize );
}
// 获得信息并显示
if ( (dwRetVal = GetIpAddrTable( pIPAddrTable, &dwSize, 0 )) != NO_ERROR )
{
     printf("Call to GetIpAddrTable failed.\n");
}
// IP、掩码等是以 DWORD 的形式保存的
printf("Address: %u.%u.%u.%u\n",
     pIPAddrTable->table[0].dwAddr & 0x000000FF,
     (pIPAddrTable->table[0].dwAddr & 0x0000FF00)>>8,
     (pIPAddrTable->table[0].dwAddr & 0x00FF0000)>>16,
     (pIPAddrTable->table[0].dwAddr & 0xFF000000)>>24);
printf("Mask: %lu\n", pIPAddrTable->table[0].dwMask);
printf("Index: %lu\n", pIPAddrTable->table[0].dwIndex);
printf("BCast: %lu\n", pIPAddrTable->table[0].dwBCastAddr);
printf("Reasm: %lu\n", pIPAddrTable->table[0].dwReasmSize);

MyFree( pIPAddrTable );// 释放内存
// GetIPStatistics
printf("-------------------------\n");
printf("GetIPStatistics\n\n");
// 定义变量，分配内存
MIB_IPSTATS *pStats;
pStats = (MIB_IPSTATS*) MyAlloc(sizeof(MIB_IPSTATS));

// 获取信息
if ((dwRetVal = GetIpStatistics(pStats)) != NO_ERROR)
{
     printf("\tError getting stats.\n");
}
// 显示
printf("\tNumber of IP addresses: %ld\n", pStats->dwNumAddr);
printf("\tNumber of Interfaces: %ld\n", pStats->dwNumIf);
printf("\tReceives: %ld\n", pStats->dwInReceives);
printf("\tOut Requests: %ld\n", pStats->dwOutRequests);
printf("\tRoutes: %ld\n", pStats->dwNumRoutes);
printf("\tTimeout Time: %ld\n", pStats->dwReasmTimeout);
printf("\tIn Delivers: %ld\n", pStats->dwInDelivers);
printf("\tIn Discards: %ld\n", pStats->dwInDiscards);
printf("\tTotal In: %ld\n", pStats->dwInDelivers + pStats->dwInDiscards);
printf("\tIn Header Errors: %ld\n", pStats->dwInHdrErrors);
MyFree(pStats);// 释放

// GetTCPStatistics
printf("-------------------------\n");
printf("GetTCPStatistics\n\n");
// 定义变量，分配空间
MIB_TCPSTATS *pTCPStats;
pTCPStats = (MIB_TCPSTATS*) MyAlloc (sizeof(MIB_TCPSTATS));
```

```
        // 获取信息，并显示
        if ((dwRetVal = GetTcpStatistics(pTCPStats)) != NO_ERROR)
             printf("Error getting TCP Stats.\n");
        printf("\tActive Opens: %ld\n", pTCPStats->dwActiveOpens);
        printf("\tPassive Opens: %ld\n", pTCPStats->dwPassiveOpens);
        printf("\tSegments Recv: %ld\n", pTCPStats->dwInSegs);
        printf("\tSegments Xmit: %ld\n", pTCPStats->dwOutSegs);
        printf("\tTotal # Conxs: %ld\n", pTCPStats->dwNumConns);
        MyFree(pTCPStats);// 释放内存

        // GetAdapatersInfo
        printf("------------------------\n");
        printf("GetAdapatersInfo\n\n");
        // 定义变量，分配内存
        IP_ADAPTER_INFO *pAdapterInfo;
        IP_ADAPTER_INFO *pAdapter;
        pAdapterInfo = (IP_ADAPTER_INFO *) MyAlloc( sizeof(IP_ADAPTER_INFO) );
        ulOutBufLen = sizeof(IP_ADAPTER_INFO);

        // 测试，所需内存大小
        if (GetAdaptersInfo( pAdapterInfo, &ulOutBufLen) != ERROR_SUCCESS)
        {
             MyFree (pAdapterInfo);
             pAdapterInfo = (IP_ADAPTER_INFO *) MyAlloc (ulOutBufLen);
        }
        // 获得信息
        if ((dwRetVal = GetAdaptersInfo( pAdapterInfo, &ulOutBufLen)) != NO_ERROR)
        {
             printf("Call to GetAdaptersInfo failed.\n");
        }
        pAdapter = pAdapterInfo;
        // 显示各个网卡的信息
        while (pAdapter)
        {
             printf("\n\t***********\n");
             printf("\tAdapter Name: \t%s\n", pAdapter->AdapterName);// 网卡设备名
             printf("\tAdapter Desc: \t%s\n", pAdapter->Description);// 描述
             printf("\tAdapter Addr: \t%ld\n", pAdapter->Address);// MAC
             printf("\tIP Address: \t%s\n", pAdapter->IpAddressList.IpAddress.String);// IP
             printf("\tIP Mask: \t%s\n", pAdapter->IpAddressList.IpMask.String);// 掩码
             printf("\tGateway: \t%s\n", pAdapter->GatewayList.IpAddress.String);// 网关
             printf("\t***\n");
             if (pAdapter->DhcpEnabled)// DHCP
             {
                  printf("\tDHCP Enabled: Yes\n");
                  printf("\t\tDHCP Server: \t%s\n", pAdapter->DhcpServer.IpAddress.String);
                  printf("\tLease Obtained: %ld\n", pAdapter->LeaseObtained);
             }
             else
                  printf("\tDHCP Enabled: No\n");
             if (pAdapter->HaveWins)// Wins
             {
                  printf("\tHave Wins: Yes\n");
                  printf("\t\tPrimary Wins Server: \t%s\n", pAdapter->PrimaryWinsServer.IpAddress.
String);
                  printf("\t\tSecondary Wins Server: \t%s\n",pAdapter->SecondaryWinsServer.
IpAddress.String);
             }
             else
                  printf("\tHave Wins: No\n");
             pAdapter = pAdapter->Next;
        }
        MyFree(pAdapterInfo);// 释放
    }
```

4．实例运行结果

运行结果如图 14-6、14-7、14-8 所示。

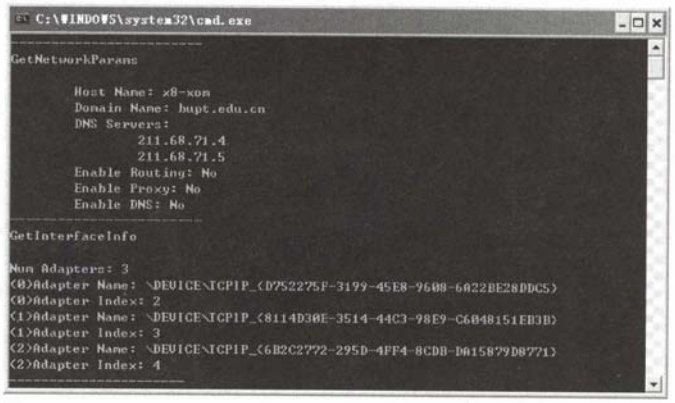

图 14-6　GetNetwrokName、GetInterfaceInfo 函数获取的网络信息

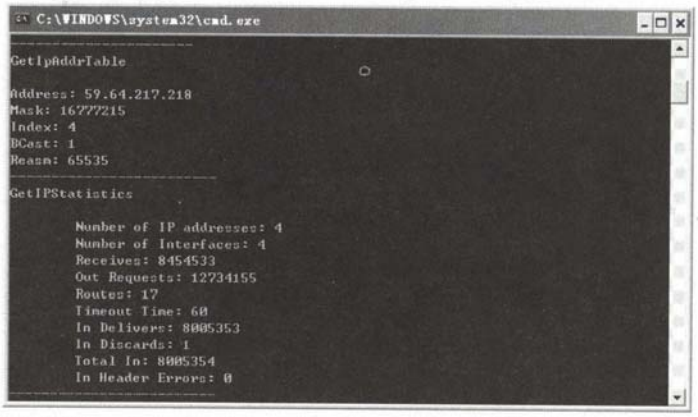

图 14-7　GetIpAddrTable、GetIPStatistics 函数获取的网络信息

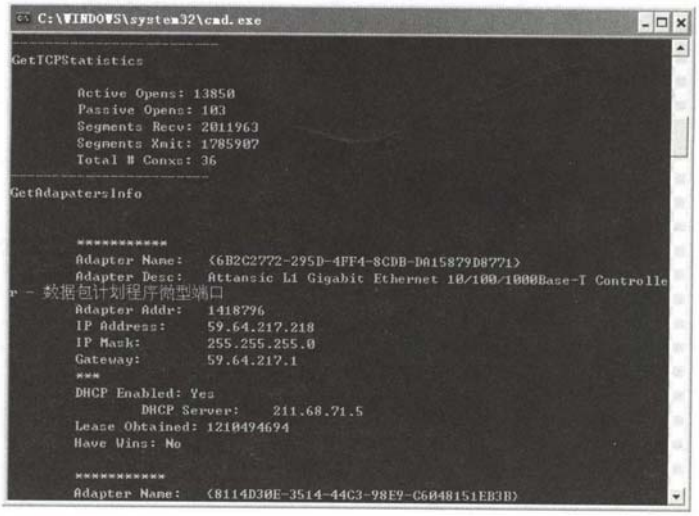

图 14-8　GetTCPStatistics、GetAdapatersinfo 函数获取的网络信息

第 15 章　程序安装与设置

当软件体积庞大、文件数量众多，尤其是当软件的运行依赖于多种系统环境时，一般需要为软件开发一个安装包，使软件在正确安装和设置后才能运行。安装程序可以设置的程序运行环境很多，比如快捷方式、注册表项等，最重要的是要使软件各程序模块、数据文件的位置要求相对固定。安装程序还有一个明显的好处是可以将一个软件和很多文件，包括程序文件、数据文件、配置文件等集中到一个文件中，便于软件的发布和传播。

总之，程序安装包是保证程序有正确的运行环境、保存程序的安装性、易用性的重要方法，也是便于程序发布和传播的重要方法。

本章将介绍两种较为主流的程序安装技术，一种是编写 setup.exe 文件，在程序安装时读取 inf 配置文件，将压缩保存于 cab 文件中的文件释放到指定位置，并进行环境设置。第二种方法是使用 WindowsInstaller 服务创建 msi 文件，由 Windows 服务读取 msi 文件中的配置信息和文件进行程序安装。由于第
二种方法可以将整个安装程序集中在一个 msi 文件中，因此目前得到了更为广泛的应用。

15.1 节～15.3 节介绍第一种方法。15.4 节介绍 msi 程序安装包的创建与配置方法。

15.1 创建 cab 文件

cab 文件是在 Windows 系统常用的数据存储文件，称为 Cabinet 文件。cab 文件在程序安装中有广泛的应用。

cab 文件是一种高压缩比的压缩格式。除了用于文件打包安装外，还可以单纯地应用于文件压缩保存。

创建 cab 文件的工具是 makecab.exe。这是操作系统自带的应用程序，位于 Windows 安装目录 System32 文件夹下。

15.1.1　makecab.exe

在命令提示中输入 makecab.exe，会打印出 makecab.exe 程序的参数。makecab 的参数选项如下所示：

```
MAKECAB [/V[n]] [/D var=value ...] [/L dir] source [destination]
```

/V 参数后可接一个数据，用于显示压缩过程。

/D 参数用于指定压缩程序指定的参数变量，比如压缩算法、cab 文件的分卷大小等。

/F 几参数用于指定生成的 cab 文件的保存路径（默认为当前路径）。

命令行的最后两项分别是需要压缩的源文件和目的文件。

如图 15-1 所示，在 E:\路径下有一个 help.doc 文件。在命令提示中切换到 E:\，运行：

```
makecab.exe /V3 /L E:\cab help.doc help.cab
```

图 15-1　makecab.exe 创建 cab 文件

在 E:\目录下会生成一个 cab 目录，目录中有生成的 help.cab 文件。

15.1.2　压缩多个文件

上述过程只能将一个文件压缩到 cab 文件夹中，如果需要将多个文件压缩到一个 cab 文件中，需要使用 makecab 指导文件，命令形式如下：

```
MAKECAB [/V[n]] [/D var=value ...] /F directive_file [...]
```

其中但选项可以指定一个文件，这个文件中包括所要打包的文件列表。

将需要打包的文件的路径依次写入到文件中，第一行写一个。然后，运行 makecab 时使用 / F 参数指定此文件，便可以将该文件中所列的所有文件都打包到一个 cab 文件中。

实例 15-1　将整个文件夹打包到 cab 文件

将需要打包的文件都放到一个目录中，比如 cabfiles。

运行：

```
dir /B /S /A-D cabfiles >list.txt
```

将 cabfiles 目录及子目录中的所有文件路径都写入 list.txt 中。

查看 list.txt 的文件内容，运行：

```
type list.txt
```

会输出 list.txt 中的内容，是整个 cabfiles 目录及子目录中的所有文件路径，如图 15-2 所示。

然后运行 makecab，将 list.txt 作为指导文件，生成 cab 文件，如图 15-2 所示。

```
makecab /f list.txt /d compressiontype=lzx /d compressionmemory=21 /d maxdisksize=1024000000
/d diskdirectorytemplate=data* /d cabinetnametemplate=data*.cab
```

命令中使用/D 参数指定了多个变量值，compressiontype 和 compressionmemory 说明压缩算法使用 lzx，lzx 是一种高压缩比的算法，maxdisksize 指定分卷大小，makecab 会自动将生成的 cab 文件划分为多个，如果不指定此参数，默认的分卷大小为 1.3 8MB，diskdirectorytemplate 和 cabinetnametemplate 参数是生成的 cab 文件和命名模板，*号会被顺序的数字代替。

图 15-2　使用 makecab 打包目录的过程

运行上述命令，会在当前目录下生成 data1 目录，目录中有 data1.cab 等文件。

在生成了 cab 文件的同时，还会在当前目录下生成 setup.inf 文件和 setup.rpt 文件。inf 文件是安装配置信息，在安装时，会使用到此文件。文件中的配置内容会影响到程序的安装，比如在本实例中，生成的 inf 文件如下：

```
;*** BEGIN ************************************************************
;**                                                                    **
;** Automatically generated on: Sun May 04 12:33:05 2008              **
;**                                                                    **
;** MakeCAB Version: 5.1.2600.2180                                     **
;**                                                                    **
;*** BEGIN ************************************************************
[disk list]
1,Disk 1
[cabinet list]
1,1,data1.cab
[file list]
1,1,config.dat,24249
1,1,face.jpg,62752
1,1,icon.ico,1078
1,1,logo.jpg,69521
1,1,help.doc,26112
1,1,Child.exe,38912
1,1,enum.exe,50688
1,1,env.dll,54272
1,1,Fibers.exe,50688
1,1,Processes.exe,49152
;*** END **************************************************************
;**                                                                    **
;** Automatically generated on: Sun May 04 12:33:05 2008              **
;**                                                                    **
;*** END **************************************************************
```

15.1.3　Cabinet软件开发工具包(CABSDK)

上述只是介绍了 cab 文件生成的基本方法。除了 makecab.exe 工具外，还有 cabarc.exe 等工具可供使用。这些工具都可以从微软的 Cabinet 软件开发工具包中获得。

Cabinet 软件开发工具包中还包括了各工具，如 makecab.exe 的详细使用说明文档。如果读者需要了解更细节的内容，可以下载 Cabinet 软件开发工具包，从中获得完整的工具和

文档信息，下载地址：http://support.micros oft.com/kb/310618。

15.2 编写INF文件

安装程序的运行需要 INF 文件的配合，INF 文件中配置了安装程序应当如何完成安装操作。

所有的文件、注册表项等都在 INF 文件中配置。而安装程序只是读取配置文件，并按照配置文件进行相当操作即可。如果安装的程序有变化，，一般情况下，只需要修改配置文件，而不需要修改安装程序。

系统提供了若干的 Setup API 来进行基于 INF 文件的安装。因此程序安装工作的主要内容是编写 INF 文件，相对而言安装程序的开发比较简单。

本节介绍 INF 文件的相关内容，15.3 节介绍如何编写安装程序。

15.2.1 INF文件格式

INF 文件的全称是“安装信息文件”。INF 文件是一个文本文件，一个 INF 文件被划分为很多节，每一个节在安装时都有相应的作用。而每一个节中都有若干项。因此 INF 文件的基本格式如下：

```
[节名]
项
……
[节名]
项
……
……
```

表 15.1 列举了在 INF 文件中常用的节及意义。

表 15.1　　INF 文件中常用的节

节	子 项 用 途	
[Version]	是所有节中唯一必须且节名不可变的节。表明了INF 文件的格式版本	Signature，版本信息
		Provider
[Install]	安装过程。会依次执行第一项所指定的动作。本节的节名任意，可以由安装程序指定	CopyFiles，指明了安装进所需复制的文件所在的节。由 CopyFiles 所指定的节被称作[Copy Files]节。[Copy Files]节的节名可以是[DestinationDirs]节的项，[DestinationDirs]节为其第一项指定一个目标路径
		AddReg，指明了安装时所需增加的注册表项所在的节，由 AddReg 项所指定的节称为 [Add Registry]节
		Delfiles，指明了安装时所需删除的文件所在的节，[Delete Files]
		DelReg，指明了安装时所需删除的注册表项所在的节，[Delete Registry]
		UpdateInis，指明了安装时更新 INI 文件所在的节，[Update INI File]

续表

节	子 项 用 途
[Services]	列举了需要安装的服务
[DestinationDirs]	目标文件夹，表明了复制等操作的目的路径，每一个 [Copy Files]节都在目标文件中对应一项，用于指定文件复制的路径
[SourceDisksFiles]	源文件，该节中的每一项是一个源文件，每一节指定了源文件所在的源磁盘 ID
[SourceDisksNames]	源磁盘名，为每一个安装的源磁盘指定一个磁盘 ID。这里所说的磁盘并不一定是真实的磁盘，也有可能是 cab 文件或一个路径
[Strings]	在程序安装时可能会使用到字符串的列表，每一项是一个字符串，有名字有内容

实例 15-2 Setup. inf 安装信息文件

本实例是一个 INF 文件，实现了安装时从 cab 文件中读取文件，复制到指定目标，并设置了若干注册表项。

15.2.2 小节～15.2.5 小节在本实例的基础上分析 INF 文件各部分的意义。

```
[Version]
Signature="$Windows NT$"
Provider="Windows programming"

[Install]
CopyFiles=ProgramFiles, GraphicsFiles, DataFiles
AddReg=NewRegistryInfo, MoreNewRegistryInfo

[SourceDisksNames]
1=="MyIntall" , "data1.cab",,, 0x10,"MyInatall.tag"

[SourceDisksFiles]
Child.exe=1
config.dat=1
enum.exe=1
env.dll=1
face.jpg=1
Fibers.exe=1
help.doc=1
icon.ico=1
logo.jpg=1
Processes.exe=1

[ProgramFiles]
Child.exe
enum.exe
env.dll
Fibers.exe
Processes.exe

[GraphicsFiles]
help.doc
icon.ico
logo.jpg

[DataFiles]
help.doc
config.dat
```

```
[DestinationDirs]
ProgramFiles=32768
DataFiles=32768,Data
GraphicsFiles=32768,Graphics

[DefaultUninstall]
CustomDestination = ProgramDir
DelFiles = ProgramFiles,DataFiles,GraphicsFiles
DelReg = NewRegistryInfo,MoreNewRegistryInfo

[NewRegistryInfo]
HKLM,Software\MyInstall\Process\CurrentVersion\Uninstall\process,DisplayName,0x00000000,
%fullname%
HKLM,Software\MyInstall\Process\CurrentVersion\Uninstall\process,InstalledDir,0x00000000,
%32768%
HKLM,Software\MyInstall\Process\CurrentVersion\Uninstall\process,UninstallString,0x0000
0000,"rundll32 advpack.dll,LaunchINFSectionEx %17%\setup.inf,DefaultUninstall"

[MoreNewRegistryInfo]
HKLM,Software\MyInstall\Process\CurrentVersion\Uninstall\process

[ProgramDir]
HKLM,Software\MyInstall\Process\CurrentVersion\Uninstall\process,2

[Strings]
fullname = "MY Pocket Version 1.0.0"
shortname = "MY Pocket"
dname = "HE HE"
```

15.2.2 Install节

Install 节表明了在安装中所需要进行的操作和操作的顺序。Intall 节中的每一项对应一种操作。

在实例 15-2 中,有两个 Install 节,[Install]、[DefaultUninstall]。这两个都是 Install 节，Install 节的节名可以是任意字符串，不一定是“Install”，所设置的节名只需要安装程序知道即可(如实例 15-3)。

在[Install]节中，共有两项，CopyFiles 和 AddReg。[Install]项用于安装操作。[Install]节的每一个子项定义一种安装过程。

在[DefaultUninstall]节中共有三项，CustomDestination、DelFiles 和 DelReg。[DefaultUninstall]项用于卸载操作。

这些项都是程序安装的过程和动作，各项的值都是 INF 文件中其他节的节名。

15.2.3 CopyFiles和AddReg等安装过程

Install 节中项是程序的安装过程。

CopyFiles 项指定了安装过程中需要复制的文件。AddReg 指定了安装过程中需要更新的注册表项。

这些项值都是 INF 文件中的节。比如实例 15-2 中，CopyFiles 的值为 ProgramFiles、GraphicsFiles 和 DataFiles，CopyFiles 项所指定的节都称作[Copy Files]。

[Copy Files]节中的项是文件，都是位于源中的文件，需要在安装时复制到目标文件。之所以要定义 3 个[Copy Files]节是因为每个节中的文件所复制的目标路径是不一样的。在使用[DestinationDirs]节指定[Copy Files]节的文件复制路径时，一个[Copy Files]只有一个目标文件夹（参见 15.2.4 小节）。

AddReg 的值是 NewRegistryInfo 和 MoreNewRegistryInfo,这两个也是 INF 文件中的节，由 AddReg 项所指定的节称作[Add Registry]，[Add Registry]节中的每一个项是一个注册表项，在安装时[Add Registry]节所指明的注册项会被导入到注册表。

15.2.4 源路径和目的路径

[Copy Files]项仅是说明了所要复制的文件，但是没有说明所需复制的文件位于什么路径，也没有说明复制的目的路径。

SourceDisksNames、SourceDisksFiles 和 DestinationDirs 节完成源路径和目的路径的指定。

先来分析 DestinationDirs 节。

实例 15-2 中的 DestinationDirs 节包括 3 项，每一项都是一个[Install]节，“Copy Files”项的值。

值“32768”是一个路径 ID，相当于一个“变量”在安装程序中可以使用具体的路径字符串来给路径 ID 赋值。在 INF 文件中可以设置不同的路径 ID，都是使用整数。DestinationDirs 节 3 个“CopyFiles”所指定的路径分别是“32768”所代表的路径及其 data 子文件夹和 Graphics 子文件夹。

结合实例 15-3，安装程序中将 32768 路径文件设置为了“CSIDL_PROGRAM_FILES”所代表的特殊路径（也即一般情况下的“C:\Program Files”）的 MyInstall 子目标。

实例 15-3 中设置路径 ID 的程序如下：

```
bResult = SetupSetDirectoryId(hInf, 32768, szDisPath);
```

SourceDisksNames、SourceDisksFiles 两节指定了各个需要复制的文件的源路径。

SourceDisksNames 节指定了源文件所在的“磁盘”，这里所谓的“磁盘”也可以是 cab 打包文件。

SourceDisksNames 节为每一个 cab 文件分配了一个 ID。比如在实例 15-2 中，将 datal.cab 分配了一个 lD“1”。在 SourceDisksFiles 节中需要使用到这个 1D。SourceDisksFiles 节的作用是指定每个源文件所在的

“磁盘”ID。如实例 15-2 中，由于所有的源文件都包括在 datal.cab 中，因此所有的文件都属性“磁盘”

1。SourceDisksFiles 的每一个项对应一个复制操作的源文件，项的值是“磁盘”ID。

15.2.5 字符串表

字符串表所在的节是[Strings]，第一个字符串都有一个名和一个值。使用%字符串名%的形式来引用。

实例 15-2 的字符串节定义了 3 个字符串，而在[NewRegistryInfo]等节中引用了这些字符串。

15.3 安装程序setup.exe的编号

在完成了 cab 文件的创建和 INF 文件的创建后，下面来编写安装程序。安装程序只需调用几个系统提供的 Setup API，就可完成安装操作，实例 15-3 是安装程序的源代码。

1．实例 15-3 安装程序

在本程序中，首先获取了程序所在目录，使用 Setup API 函数 SetupOpenInfFile 打开 INF 文件，之后使用了 SetupSetDirectoryId 函数将 INF 配置文件的路径 ID，32768 设置为程序安装路径，以便程序构造目的文件夹。

在进行了基本的设置后，SetupInstallFromInfSection 函数完成了整个安装过程。第 3 个参数是指明的[Install]节的节名，这里使用一“Install”，表明按照“Install”节定义的安装过程来安装。第 4 个参数是安装的标志，其意义是定义了哪几类的动作会被执行，这里设置为 SPINST-- FILES 与 SPINST REGISTRY 表示只有与文件和注册表相关的操作才会被实施。

SetupInstallFromInfSection 还指明了一个回调函数，在一般的应用情况下，这个回调函数使用系统默认的即可。

使用 Setup API 需要包含头文件“Setupapi.h”并链接到 Setupapi.lib 库。

```
/* ************************************
* setup.c  setup.exe
**************************************/
/* 头文件 */
#include <Windows.h>
#include <Setupapi.h>
#include <shlobj.h>
/* 库 */
#pragma comment (lib, "shell32.lib")
#pragma comment (lib, "Setupapi.lib")

/************************************
* VOID GetSourceDirectory(LPSTR szPath)
* 功能：获得当前路径
* szPath，返回路径
**************************************/
VOID GetSourceDirectory(LPSTR szPath)
{
    int i;
    GetModuleFileName(NULL,szPath,MAX_PATH);
    i=strlen(szPath);
    while ((i>0)&&(szPath[i-1]!='\\'))
    {
        szPath[--i]=0;
    }
}

/************************************
* WinMain
* 功能：调用相关 Setup API 进行安装
**************************************/
INT WinMain(
          HINSTANCE hInstance,
          HINSTANCE hPrevInstance,
          LPSTR lpCmdLine,
          int nCmdShow
          )
{
    HINF hInf; // INF 文件句柄
    CHAR szSrcPath[MAX_PATH];// 源路径
    CHAR szDisPath[MAX_PATH];// 目的路径
    BOOL bResult;
    PVOID pContext;
    // 与本程序在同一目录下的 Setup.inf
    GetSourceDirectory(szSrcPath);
    lstrcat(szSrcPath,"setup.inf");
    // 打开 inf 文件
    hInf = SetupOpenInfFile(szSrcPath, NULL, INF_STYLE_WIN4, NULL);
    // 是否成功
    if (hInf == INVALID_HANDLE_VALUE)
    {
        MessageBox(NULL,
            "Error: Could not open the INF file.",
            "ERROR",
            MB_OK|MB_ICONERROR);
        return FALSE;
    }
    // 获得 Program Files 的路径
    SHGetSpecialFolderPath(NULL,
        szDisPath, CSIDL_PROGRAM_FILES , FALSE);
```

```
    // 构造目的路径
    lstrcat(szDisPath,"\\MyInstall");
    // 给 inf 配置文件中的路径 ID 赋值，使用路径替换路径 ID
    bResult = SetupSetDirectoryId(hInf, 32768, szDisPath);
    if (!bResult)
    {
        MessageBox(NULL,
            "Error: Could not associate a directory ID with the destination directory.",
            "ERROR",
            MB_OK|MB_ICONERROR);
        SetupCloseInfFile(hInf);
        return FALSE;
    }

    // 设置默认 callback 函数的参数
    pContext=SetupInitDefaultQueueCallback(NULL);

    // 进行安装
    bResult=SetupInstallFromInfSection(
        NULL, // 父窗口句柄
        hInf, // INF 文件句柄
        "Install", // INF 文件中，配置了安装信息的节名
        SPINST_FILES | SPINST_REGISTRY , // 安装标志
        NULL, // 安装键值
        NULL, // 源文件和路径，可以在 INF 文件中配置
        0, // 复制时的动作
        (PSP_FILE_CALLBACK)SetupDefaultQueueCallback, // 回调函数
        pContext, // 回调函数的参数
        NULL, // 设备信息
        NULL // 设备信息
        );
    // 安装是否成功
    if (!bResult)
    {
        // 失败，输出错误信息
        MessageBox(NULL,
            "SetupInstallFromInfSection",
            "ERROR",
            MB_OK|MB_ICONERROR);
        //关闭
        SetupTermDefaultQueueCallback(pContext);
        SetupCloseInfFile(hInf);
        return FALSE;
    }
    // 关闭
    SetupTermDefaultQueueCallback(pContext);
    SetupCloseInfFile(hInf);
    return TRUE;
}
```

2．程序运行结果

运行 setup.exe 后，会弹出对话框，显示安装的进度信息，如图 15-3 所示。

完成安装后，安装程序退出。

如图 15-4 所示，安装结束后，在程序指定的安装目录 C:\rogram Files\MyInstall 中所有 datal.cab 中的文件都按照 setup.inf 文件配置的情况复制到了指定文件夹。

图 15-3　安装程序对话框

在 inf 文件中配置了注册表项，在安装完成后，查看注册表。如图 15-5 所示，打开注册表编辑器，在注册表键 HKEY LOCAL MACHINE\SOFTWARE\

MyInstall\Process\Current Version \Uninstall\process 下有根据配置信息设置的注册表项。

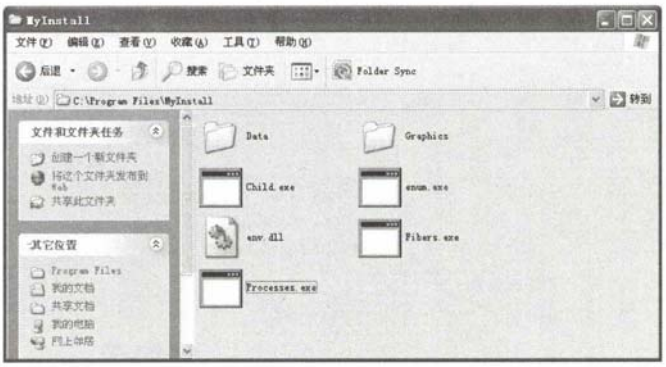

图 15-4　安装完成后的安装的目标目录

图 15-5　安装程序写入的注册表

15.4　使用msi文件进行安装

相对于安装程序，目前更多的软件是使用 Windows Installer Service 进行软件安装的。使用 Windows Installer Service 来安装程序需要创建软件安装包(. msi)文件。

微软公司专门发布了 Windows Installer SDK 用于创建软件安装包。在 Windows SDK 中已经包括了 Windows Installer SDK。

本节将介绍如何使用 Windows installer SDK 中的工具来创建一个完整的 msi 程序安装包。

15.4.1　Windows Installer Service

先介绍使用 msi 程序安装包安装所要依赖的 Windows Installer 服务。

使用 msi 程序安装包进行安装依赖于 Windows Installer 服务。在系统安装后，Windows Installer 服务就已经被安装，如图 15-6 所示。

Windows Installer 服务的可执行程序是 msiexec.exe。

查看注册表，具有键：

```
HKEY_CLASSES_ROOT\.msi\Msi.Package
```

而键：

```
HKEY_CLASSES_ROOT\Msi.Package\shell\Open\command
```

图 15-6　Windows Installer Srevice

的键值为：

```
"%SystemRoot%\System32\ msiexec.exe" /i "%1" %*
```

因此用户双击 msi 程序后，会以 / i 参数打开双击的 msi 包。和用户直接在命令行中运行

```
msiexec.exe /I <MSI 文件名>
```

效果是一样的。

虽然一般情况下，使用到的 msiexec.exe 的参数很少，实际上 msiexec.exe 的参数是非常丰富的：

```
msiexec /Option <Required Parameter> [Optional Parameter]
安装选项
   </package | /i> <Product.msi>
      安装或配置产品
   /a <Product.msi>
      管理安装 - 在网络上安装产品
   /j<u|m> <Product.msi> [/t <Transform List>] [/g <Language ID>]
      播发产品 - m 播发到所有用户，u 播发到当前用户
   </uninstall | /x> <Product.msi | ProductCode>
      卸载产品
显示选项
   /quiet
      安静模式，无用户交互
   /passive
      无从参与模式 - 只显示进程栏
   /q[n|b|r|f]
      设置用户界面级别
      n - 无用户界面
      b - 基本界面
      r - 精简界面
      f - 完整界面(默认值)
   /help
      帮助信息
重新启动选项
   /norestart
      安装完成后不重新启动
   /promptrestart
      提示用户重新启动(如果必要)
   /forcerestart
      安装后总是重新启动计算机
```

此外还有日志选项、更新选项、修复选择等。详细参数可查看 msiexec.exe 帮助信息或 SDK 文档。

只有在 msiexec.exe 正常工作的情况下，msi 文件才能被正常安装。此外还需注意 msiexec.exe 的版本，目前很多程序的安装都需要 3.01 以上的版本的 Windows Installer

服务。

15.4.2 msi文件的创建与修改工具orca.exe

下面开始介绍如何将自己开发的软件打包为 msi 文件。

创建和编辑 msi 文件的工具是 Windows Install SDK，已经包含在了 Windows SDK 中。与 msi 文件编辑与修改相关的工具有很多。在此工具中 orca 功能最为强大，也最为实用，可以完成整个 msi 文件的创建和编辑工作。

使用 orca 前需要安装，其安装程序也是一个 msi 文件。

在 SDK 安装目录 bin 文件夹下双击 orca.msi，按照提示安装，如图 15-7 所示。

完成后，会在开始菜单增加 orca 快捷方式，如图 15-8 所示。

图 15-7 orca 的安装

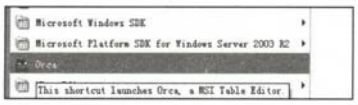

图 15-8 启动 orca

启动 orca，出现如图 15-9 所示界面。

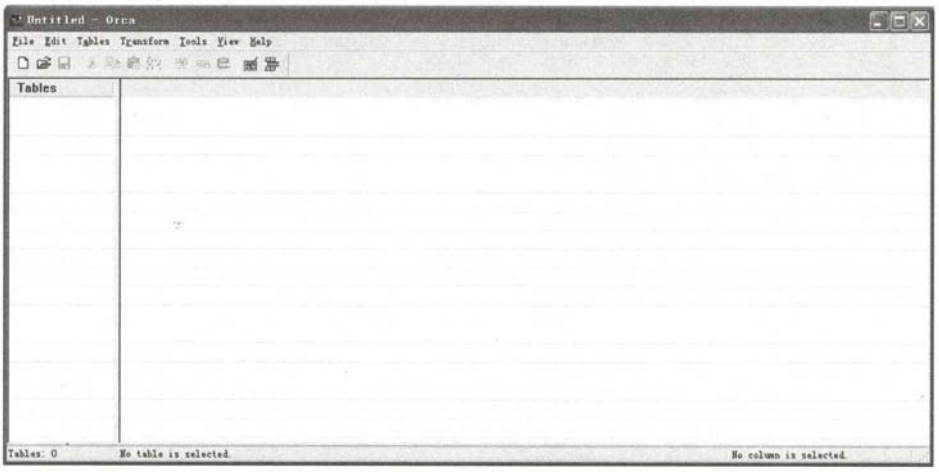

图 15-9 orca 界面

可以使用 orca，编辑已经存在的 msi 文件。先使用 orca 来查看 msi 文件。找一个 msi 文件(就比如 orca.msi)，通过菜单打开。

打开后，文件内容如图 15-10 所示。

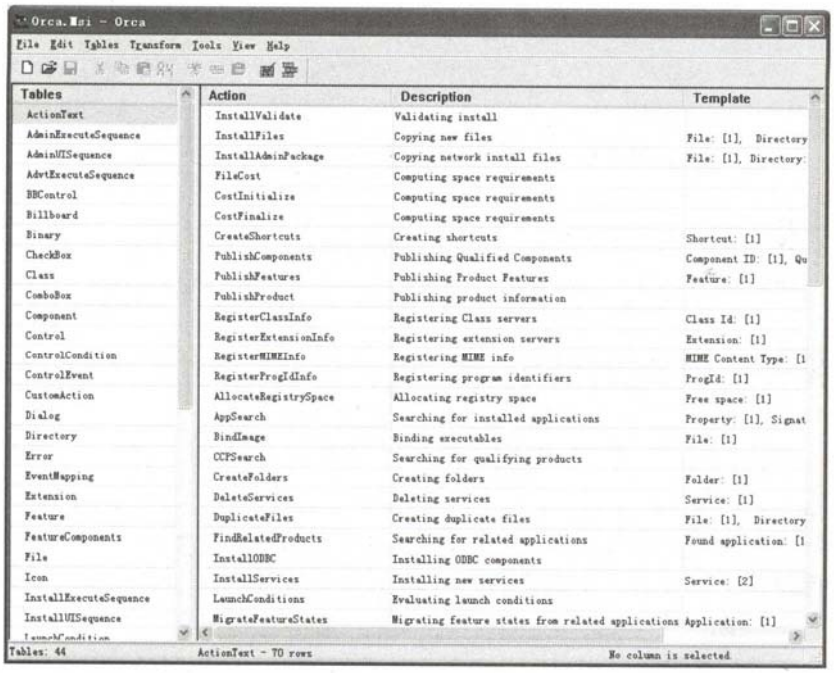

图 15-10　使用 Orca 编辑和查看 msi 文件

msi 文件结构是以表(Table)为单位的。一个 msi 文件中存在很多的表，每一个表中又包括若干项。不同的表所包括的项的数量的类型各不相同。

下面几节将具体介绍 msi 文件中各表的意义及在安装时的作用。

15.4.3　准备工作

在创建安装程序之前，首先需要准备需要安装的文件，比如程序文件、文档文件、数据文件等。

此外还需准备一个空白的 msi 模板，这样可以减少 msi 文件编辑的工作量。SDK 已经提供了空白 msi 文件模板。在 SDK 安装目录的 bin\msitools 文件夹下可以找到 msi 模板。

15.4.4　编辑表组

msi 文件中的表各有意义。表有很多种，分为若干类，其中有核心表组，目录、文件、快捷方式相关表组，以及注册表、安装过程和界面等几个方面的表组。

核心表组包括 Component 表、Feature 表、Condition 表、Directory 表等，是 msi 文件自身的一些核心信息。

目标和文件表组指定了在安装时进行的文件复制、目录创建等动作。与文件和目录相关的表包括 File 表、Media 表、Component 表等。

安装过程和界面表组包含 InstallUISequence 表、InstallExecuteSequence 表、AdminUISequence 表、AdminExecuteSequence 表、AdvtUISequence、AdvtExecuteSequence 表等。

各个表的具体使用方法和表项的各个字段意义比较琐碎，本书不再做一一介绍。在具体使用时可以参考 SDK 文档。

第 16 章　设备驱动管理与内核通信

设备管理是操作系统的重要功能。任何操作系统都需要对设备进行管理。应用程序也会通过操作系统操作和管理物理设备。设备驱动程序为操作系统和应用程序提供了操作设备的接口。很多情况下，应用程序需要直接或间接地与内核驱动程序通信。

x86 平台有 4 个优先级别，分别称作 ring0、ring1、ring2 和 ring3。运行在不同的优先级别上的程序在物理设备访问、内存访问等所具有的权限是不同的，甚至所能运行的 CPU 指令也有所不同。

Windows 系统内核和驱动程序运行于 ring0 级，ring0 级也称作内核态；应用程序运行于 ring3 级，也称作用户态。

比如 ring3 级（用户态）的应用程序是不能访问 ring0 级（内核态）内核驱动程序的内存空间的，也不能直接操作硬件设备。应用程序需要操作硬件设备时，必须要通过系统内核（第 18 章还会介绍与用户态程序调用内核态程序的原理）。

应用程序可以将设备驱动程序加载入内核。可以对已经加载了的设备驱动程序发送请求，进行控制或与内核程序进行通信，传输数据。

本章在 16.1 节介绍与设备相关的一系统 API 函数。实现对现有设备的列举、实现监视设备加载及卸载操作。

16.2 节介绍如果对设备驱动进行控制，如何与设备驱动程序进行通信。还会介绍如何加载一个设备驱动程序。16.2 节是本章的重点。

16.3 节对如何系统内核态的设备驱动程序进行了简介。设备驱动程序开发也是 Windows 系统程序设计的重要内容。与应用程序开发不同，由于系统设备驱动程序运行于系统内容，因此只能使用系统内核的接口函数，开发时所使用的头文件和库文件都不同，开发工具也不同。系统内核开发使用 DDK (Driver Developmet Kit)开发包。可执行程序为 sys 文件。

164 节介绍了 Windows I/O 的另一个重要内容：I/O 的模式，包括同步通信和异步通信的 I/O 模式。

16.1　设备管理

本节首先对系统设备的一些基本管理进行介绍。包括如何列举系统中存在的设备，或指定设备的类型：以及监视设备的增加与删除，如放入光盘、插入 USB 设备等。

16.1.1　列举设备接口

一般列举设备驱动都是安装程序需要完成的工作，因此 Setup API 提供了一组列举设备的函数。

由 SetupDiCreateDeviceInfoList、SetupDiGetClassDevsEx 等 API 获得 HDEVINFO 类型的设备信息链表，然后由 SetupDiEnumDeviceInterfaces 依次读出设备信息链表中的设备信息。还可以使用 SetupDiGetDeviceRegistryProperty、SetupDiGetDeviceInterfaceDetail 等 API 获取更为详细的信息。

1．设备接口类型 GUID

在列举设备时，可以选定设备类型。使用一个特殊的 GUID 来表示，这些 GUID 唯一标识了一类接口的设备，称作设备类型 GUID(class GUID)。比如列举磁盘可以使用

GUID_DEVINTERFACE DISK，列举串口可以使用 GUID DEVINTERFACE_COMPORT 等。下面列出了部分可以使用的类型 GUID。

```
GUID_DEVINTERFACE_DISK
GUID_DEVINTERFACE_CDROM
GUID_DEVINTERFACE_PARTITION
GUID_DEVINTERFACE_TAPE
GUID_DEVINTERFACE_WRITEONCEDISK
GUID_DEVINTERFACE_VOLUME
GUID_DEVINTERFACE_MEDIUMCHANGER
GUID_DEVINTERFACE_FLOPPY
GUID_DEVINTERFACE_CDCHANGER
GUID_DEVINTERFACE_STORAGEPORT
GUID_DEVINTERFACE_COMPORT.
GUID_DEVINTERFACE_SERENUM_BUS_ENUMERATOR
```

2．关键 API

(1) SetupDiCreateDeviceInfoLista

SetupDiCreateDeviceInfoList 函数的功能是创建一个空的设备信息列表。

```
HDEVINFO SetupDiCreateDeviceInfoList(
  const GUID* ClassGuid,
  HWND hwndParent
);
```

ClassGuid 参数指定一个设备类型 GUID，hwndParent 参数表示需要显示设备信息窗口。可以设置为 NULL。返回值为 HDEVINFO 类型，是指向设备类型的句柄。

(2) SetupDiGetClassDevsEX。

SetupDiGetClassDevsEx 函数的功能是获取指定类型 GUID 的所有设备的信息列表。

```
HDEVINFO SetupDiGetClassDevsEx(
  const GUID* ClassGuid,
  PCTSTR Enumerator,
  HWND hwndParent,
  DWORD Flags,
  HDEVINFO DeviceInfoSet,
  PCTSTR MachineName,
  PVOID Reserved
);
```

ClassGuid 参数：指明了需要获取信息的设备类型 GUID，如果 Flags 参数设置为 DIGCF ALLCLASSES，那么这个参数是无效的。

Enumerator 参数：一般情况下可设置为 NULL。

hwndParent 参数：是用于显示的窗口的句柄，可设置为 NULL。

Flags 参数：列举的标志，值可设置为 DIGCF_ DEVICEINTERFACE、DIGCF_ ALLCL ASSES、DIGCF_ PRESENT 和 DIGCF_ PROFILE，分别表示列举所有类型、列举 ClassGuid 参数指定接口的类型、列举当前存在的设备和列举作为硬件剖面一部分的设备。

DeviceInfoSet 参数：是一个已经存在的设备信息，本函数会向其中增加新列举的信息。

MachineName 参数：是所需列举的机器名，如果为本机，设置为 NULL。

函数返回设备信息的句柄。

(3) SetupDiEnumDeviceInterfaceS。

SetupDiEnumDevieeInterfaces 函数的功能是从已经获取的设备接口列表信息中获取信息并使用结构保存，每调用一次会依次返回一个设备接口的信息。

```
BOOL SetupDiEnumDeviceInterfaces(
  HDEVINFO DeviceInfoSet,
  PSP_DEVINFO_DATA DeviceInfoData,
  const GUID* InterfaceClassGuid,
  DWORD MemberIndex,
  PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData
);
```

DeviceInfoSet 参数：由 SetupDiGetClassDevsEx 等函数的返回结果，设备接口列表。

DeviceInfoData 参数：指向 SP_DEVINFO_DATA 类型的指针，SP_DEVINFO_DATA 指明所需获取的设备相关信息，可设置为 NULL。

InterfaceClassGuid 参数：类型 GUID。

MemberIndex 参数：指明需要返回列表中等几个设备接口信息。

DeviceInterfaceData 参数：是输出参数，指向 SP_DEVICE INTERFACE DA 工 A 结构。SP DEVICEINTERFACE DATA 结构如下：

```
typedef struct _SP_DEVICE_INTERFACE_DATA {
  DWORD cbSize;
  GUID InterfaceClassGuid;
  DWORD Flags;
  ULONG_PTR Reserved;
} SP_DEVICE_INTERFACE_DATA,  *PSP_DEVICE_INTERFACE_DATA;
```

(4) SetupDiGetDeviceRegistryProperty 和 SetupDiGetD eviceInterfaceD etail.

通过 SetupDiEnumDeviceInterfaces 函数所获取的信息还不够丰富，将其作为参数，传递给 SetupDiGetD eviceRegistryProperty 和 SetupDiGetDevic eInterfaceD etail 等函数，可以获取设备接口信息的细节。函数的原型如下：

```
BOOL SetupDiGetDeviceRegistryProperty(
  HDEVINFO DeviceInfoSet,
  PSP_DEVINFO_DATA DeviceInfoData,
  DWORD Property,
  PDWORD PropertyRegDataType,
  PBYTE PropertyBuffer,
  DWORD PropertyBufferSize,
  PDWORD RequiredSize
);

BOOL SetupDiGetDeviceInterfaceDetail(
  HDEVINFO DeviceInfoSet,
  PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData,
  PSP_DEVICE_INTERFACE_DETAIL_DATA DeviceInterfaceDetailData,
  DWORD DeviceInterfaceDetailDataSize,
  PDWORD RequiredSize,

  PSP_DEVINFO_DATA DeviceInfoData
);
```

(5) SetupDiDestroyDeviceInfoList。

此外，在列举信息完成后，SetupDiCreateDeviceInfoList 分配的内存需要通过 Setup Di DestroyDeviceInfoList 函数释放内存。

```
BOOL SetupDiDestroyDeviceInfoList(
  HDEVINFO DeviceInfoSet
);
```

3. 实例 16-1　列举设备

本实例使用上面所介绍的 API 函数，实现了列举现有设备接口和指定类型的设备接口的功能。

实例共有 3 个函数：main、EnumAllDevice 和 EnumClassDevice。main 函数调用了其他两个函数。

EnumClassDevice 函数首先调用了 SetupDiCreateDeviceInfoList 函数创建了一个空的设备信息列表 HDEVINFO，然后将其作为参数传递给了 SetupDiGetClassDevsEx 函数，获取设备信息列表，由于接口类型指定为 EnumClassDevice 的参数，因此函数返回的是指定类型的设备接口。再之后，循环调用 SetupDiEnumDeviceInterfaces 和 SetupDiGetD evic eInterfac eDetail 函数，获取了设备的详细信息，并输出了设备的路径。

main 函数在调用 EnumClassDevice 时指定的类型为 GUID_DEVINTERFACE_VOLUME，所以只会列举出磁盘分区设备。

EnumAllDevice 函数的实现原理与 EnumClassDevice 类似，所输出的信息不同。实例代码如下：

```
/* ************************************
* EnumDevice.c 设备管理
**************************************/
/* 头文件 */
#include <windows.h>
#include <setupapi.h>
#include <stdio.h>
#include <devguid.h>
#include <regstr.h>
/* 函数声明 */
BOOL EnumPresentDevice( const GUID * InterfaceClassGuid );
BOOL EnumAllDevice();
/************************************
* BOOL EnumClassDevice( const GUID * InterfaceClassGuid )
* 功能: 根据类型列举当前存在的设备
* 参数: InterfaceClassGuid, 所需列举设备接口类的 GUID
************************************/
BOOL EnumClassDevice( const GUID * InterfaceClassGuid )

{
    HDEVINFO DeviceInfoSet;
    HDEVINFO NewDeviceInfoSet;
    SP_DEVICE_INTERFACE_DATA DeviceInterfaceData;
    PSP_DEVICE_INTERFACE_DETAIL_DATA lpDeviceInterfaceDetailData;
    DWORD dwBufferSize = 0;
    DWORD i;
    // 创建空设备信息列表
    DeviceInfoSet = SetupDiCreateDeviceInfoList(NULL, NULL);
    if(DeviceInfoSet == INVALID_HANDLE_VALUE)
    {
        printf("CreateDeviceInfoList failed: %d\n", GetLastError());
        return 0;
    }
```

```
// 根据接口类型获得新的设备信息列表
NewDeviceInfoSet = SetupDiGetClassDevsEx(
    InterfaceClassGuid,
    NULL,
    NULL,
    DIGCF_PRESENT | DIGCF_DEVICEINTERFACE,
    DeviceInfoSet,// 之前创建的设备信息列表
    NULL,
    NULL
    );
if(NewDeviceInfoSet == INVALID_HANDLE_VALUE)
{
    printf( "SetupDiGetClassDevsEx failed: %d\n", GetLastError() );
    return 0;
}
// 设置 SP_DEVICE_INTERFACE_DATA 大小
DeviceInterfaceData.cbSize
   = sizeof(SP_DEVICE_INTERFACE_DATA);
for (i=0; ;i++)
{
    // 列举接口信息
    BOOL bResult = SetupDiEnumDeviceInterfaces(
        NewDeviceInfoSet,
        NULL,
        InterfaceClassGuid,
        i,
        &DeviceInterfaceData
        );
    if(!bResult)
    {
        if ( GetLastError()!=NO_ERROR &&
            GetLastError()!=ERROR_NO_MORE_ITEMS )
        {
            printf("ERROR: (%d)",GetLastError());
            return FALSE;
        }
        break;
    }
    else
    {
        // 为 PSP_DEVICE_INTERFACE_DETAIL_DATA 结构分配内存，填充
        lpDeviceInterfaceDetailData = HeapAlloc(
            GetProcessHeap(), 0,
            sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA));
        lpDeviceInterfaceDetailData->cbSize
            = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
        dwBufferSize = lpDeviceInterfaceDetailData->cbSize;
        // 获得接口详细信息
        while(!SetupDiGetDeviceInterfaceDetail(
            NewDeviceInfoSet,
            &DeviceInterfaceData,
            lpDeviceInterfaceDetailData,
            dwBufferSize,
            &dwBufferSize,
            NULL))
        {
            // 如果内存空间不足，再次分配，直到可以成功调用
            if(ERROR_INSUFFICIENT_BUFFER==GetLastError())
            {
                lpDeviceInterfaceDetailData = HeapReAlloc(
                    GetProcessHeap(), 0,
```

```
                    lpDeviceInterfaceDetailData, dwBufferSize);
                lpDeviceInterfaceDetailData->cbSize
                    = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
            }
        }
        // 显示信息
        printf("DevicePath: %s\n",lpDeviceInterfaceDetailData->DevicePath);
        // lpDeviceInterfaceDetailData->DevicePath 可作为 CreateFile 的参数，进行 I/O 控制

        // 释放内存
        HeapFree(GetProcessHeap(),0,lpDeviceInterfaceDetailData);
    }
  }
  return TRUE;
}
/*************************************
* BOOL EnumAllDevice( )
* 功能：列举当前存在的设备
* 返回值：是否成功
*************************************/
BOOL EnumAllDevice()
{
  HDEVINFO hDevInfo;
  SP_DEVINFO_DATA DeviceInfoData;
  DWORD i;
  printf("Displaying the Installed Devices\n\n");
  // 得到所有设备 HDEVINFO
  hDevInfo = SetupDiGetClassDevs(NULL,
    0, // 无类型
    0, // 无回调函数
    DIGCF_PRESENT | DIGCF_ALLCLASSES );
  if (hDevInfo == INVALID_HANDLE_VALUE)
  {
    return FALSE;
  }
  // 循环列举
  DeviceInfoData.cbSize = sizeof(SP_DEVINFO_DATA);
  for (i=0;SetupDiEnumDeviceInfo(hDevInfo,i,
    &DeviceInfoData);i++)
  {
    DWORD DataT;
    LPTSTR buffer = NULL;
    DWORD buffersize = 0;
    // 获取详细信息
    while (!SetupDiGetDeviceRegistryProperty(
      hDevInfo,
      &DeviceInfoData,
      SPDRP_DEVICEDESC,
      &DataT,
      (PBYTE)buffer,
      buffersize,
      &buffersize))
    {
      if (GetLastError() == ERROR_INSUFFICIENT_BUFFER)
      {
        // 内存不足
        if (buffer) HeapFree(GetProcessHeap(), 0, buffer);
        buffer = (LPTSTR)HeapAlloc(GetProcessHeap(), 0,  buffersize);
      }
      else
        break;
```

```
        }
// 输出
        printf("GUID:{%.8X-%.4X-%.4X--%.2X%.2X-%.2X%.2X%.2X%.2X%.2X%.2X%.2X} "
            "Device: %s\n",
            DeviceInfoData.ClassGuid.Data1,
            DeviceInfoData.ClassGuid.Data2,
            DeviceInfoData.ClassGuid.Data3,
            DeviceInfoData.ClassGuid.Data4[0],
            DeviceInfoData.ClassGuid.Data4[1],
            DeviceInfoData.ClassGuid.Data4[2],
            DeviceInfoData.ClassGuid.Data4[3],
            DeviceInfoData.ClassGuid.Data4[4],
            DeviceInfoData.ClassGuid.Data4[5],
            DeviceInfoData.ClassGuid.Data4[6],
            DeviceInfoData.ClassGuid.Data4[7],buffer);
        if (buffer) HeapFree(GetProcessHeap(), 0, buffer);
    }
    if ( GetLastError()!=NO_ERROR &&
        GetLastError()!=ERROR_NO_MORE_ITEMS )
    {
        return FALSE;
    }
    //  释放
    SetupDiDestroyDeviceInfoList(hDevInfo);
    return TRUE;
}
int main( int argc, char *argv[ ], char *envp[ ] )
{
    // 列举所有设备
    printf("Enumerating All Device\n\n");
    EnumAllDevice();
    // 列举磁盘分卷驱动器设备
    printf("\n\nEnumerating Present Volume \n\n");
    EnumClassDevice(&GUID_DEVINTERFACE_VOLUME);
    return 0;
}
```

4．运行结果

在 main 函数中先调用 EnumAllDevice，再调用 EnumClassDevice 函数。输出结果的前半部分由 EnumAllDevice 函数输出，后半部分由 EnumClassDevice 函数输出。

部分输出结果如下所示：

```
Enumerating All Device

GUID:{72631E54-78A4-11D0--BCF7-00AA00B7B32A} Device: Microsoft AC Adapter
GUID:{4D36E97D-E325-11CE--BFC1-08002BE10318} Device: ACPI Fixed Feature Button
GUID:{50127DC3-0F36-415E--A6CC-4CB3BE910B65} Device: Intel Processor
……
GUID:{4D36E97D-E325-11CE--BFC1-08002BE10318} Device: Direct memory access controller
GUID:{4D36E96B-E325-11CE--BFC1-08002BE10318} Device: 标准 101/102 键或 Microsoft 自然 PS/2 键盘
GUID:{4D36E978-E325-11CE--BFC1-08002BE10318} Device: 打印机端口
GUID:{4D36E978-E325-11CE--BFC1-08002BE10318} Device: 通讯端口
GUID:{4D36E978-E325-11CE--BFC1-08002BE10318} Device: 通讯端口
GUID:{4D36E969-E325-11CE--BFC1-08002BE10318} Device: Standard floppy disk controller
GUID:{4D36E97D-E325-11CE--BFC1-08002BE10318} Device: System speaker
……
GUID:{4D36E96A-E325-11CE--BFC1-08002BE10318} Device: 主要 IDE 通道
GUID:{4D36E96A-E325-11CE--BFC1-08002BE10318} Device: 次要 IDE 通道
GUID:{4D36E966-E325-11CE--BFC1-08002BE10318} Device: ACPI Multiprocessor PC
```

```
GUID:{4D36E97D-E325-11CE--BFC1-08002BE10318} Device: Microsoft Composite Battery
......
GUID:{4D36E972-E325-11CE--BFC1-08002BE10318} Device: WAN 微型端口 (PPTP)
GUID:{4D36E972-E325-11CE--BFC1-08002BE10318} Device: 数据包计划程序微型端口
GUID:{4D36E972-E325-11CE--BFC1-08002BE10318} Device: 数据包计划程序微型端口
GUID:{4D36E972-E325-11CE--BFC1-08002BE10318} Device: 直接并行
GUID:{4D36E97D-E325-11CE--BFC1-08002BE10318} Device: Terminal Server Device Redirector
GUID:{4D36E97D-E325-11CE--BFC1-08002BE10318} Device: Terminal Server Keyboard Driver
......
GUID:{71A27CDD-812A-11D0--BEC7-08002BE2092F} Device: 通用卷
GUID:{71A27CDD-812A-11D0--BEC7-08002BE2092F} Device: 通用卷
......

Enumerating Present Volume

DevicePath: \\?\fdc#generic_floppy_drive#6&1435b2e2&0&0#{53f5630d-b6bf-11d0-94f2-
00a0c91efb8b}
DevicePath:\\?\ide#cdromtsstcorp_cddvdw_ts-h652h_____cm00___#30313030303030
303030303030303030303030303130#{53f5630d-b6bf-11d0-94f2-00a0c91efb8b}
DevicePath: \\?\storage#volume#1&30a96598&0&signature93569356offset7e00length1ff582800#
{53f5630d-b6bf-11d0-94f2-00a0c91efb8b}
DevicePath: \\?\storage#volume#1&3735c57b&0&ldm#{44c4d545-4fe1-4d20-8842-863b53d20759}#
{53f5630d-b6bf-11d0-94f2-00a0c91efb8b}
DevicePath: \\?\storage#volume#1&3735c57b&0&ldm#{4a998178-087d-4d67-b76a-0b9ca34e5134}#
{53f5630d-b6bf-11d0-94f2-00a0c91efb8b}
```

16.1.2 监控设备的加载和卸载

可以通过向系统注册一个“设备通知”来监控设备的加载和卸载，在注册“设备通知”时，需要指定一个窗口，在注册成功后，所有的系统设备加载与卸载接口中都会接收到 WM DEVICECHANGE 消息。

(1) RegisterDeviceNotification。

注册“设备通知”使用 API 函数 RegisterDeviceNotification，函数原型如下：

```
HDEVNOTIFY RegisterDeviceNotification(
  HANDLE hRecipient,
  LPVOID NotificationFilter,
  DWORD Flags
);
```

hRecipient 用于接收通知的窗口或服务的句柄（根据 Flags 不同，既可以是窗口句柄，也可以是服务状态句柄）。

NotificationFilter 指明需要监视的设备的类型，为 DBT DEVTYP_DEVICEINTERFACE、DEV_ BROADCAST DEVICEINTERFACE、DBT DEVTYP_VOLUME 等。

Flags 司设定为 DEVICE NOTIFY_ WINDOW HANDLE 和 DEVICE NOTIFY_ SERVICE_ HANDLE，分别表示接收消息的是窗口还是服务状态句柄。

(2) WM_DEVICECHANGE 消息。

wParam 参数是事件的类型，包括 DBT_DEVICEARRIVAL、DBT_CONFIGCHANGED、DBT__ DEVNODES_CHANGED 等若干种。

lParam 参数是指向事件相关的数据结构的指针，例如，如果事件是 DBT_ DEVICEARRIVAL，那么是 DEV BROADCAST HDR 类型。

DEV_BROADCAST_HDR 类型定义如下：

```
typedef struct _DEV_BROADCAST_HDR {
  DWORD dbch_size;
  DWORD dbch_devicetype;
```

```
    DWORD dbch_reserved;
} DEV_BROADCAST_HDR,  *PDEV_BROADCAST_HDR;
```

1．实例 16-2 监视光盘的插入弹出及 USB 设备的插入弹出等

本实例通过 RegisterDeviceNotification 函数注册“设备通知”，然后在窗口中等待 WM_DEVICECHANGE，并判断消息的参数，然后在界面上显示设备加载与卸载的相关信息。

由于在调用 RegisterDeviceNotification 时，设置所监视的设备类型 Filter 的设备接口类型为 GUID_DEVINTERFACE_DISK，因此只有与磁盘相关的设备变化才会被监视到。

实例中 DoRegisterDeviceInterface 函数用于注册“设备通知”，OnDeviceChange 函数用于处理 WM_DEVICECHANGE 消息及其参数。

在所有的设备变更消息中，特别处理了光盘变更的消息，通过 DBTF_MEDIA 判读是否为光驱媒质变化。

```
/* ************************************
* DevChgc 设备管理
**************************************/
/* 头文件 */
#include <windows.h>
#include <Dbt.h>
/* 全局变量 */
HINSTANCE hinst;
HWND hWnd;
HDEVNOTIFY hDevNotify;// NOTIFY
int YStart = 10;// 用于 TextOut 时定位输出位置
/* 全局变量 */
int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int);
InitApplication(HINSTANCE);
InitInstance(HINSTANCE, int);
LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);
void OnDeviceChange (HWND hwnd, WPARAM wParam, LPARAM lParam);
char FirstDriveFromMask (ULONG unitmask);
BOOL DoRegisterDeviceInterface( GUID , HDEVNOTIFY * );
/************************************
* WinMain
**************************************/
int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
   MSG msg;
   BOOL fGotMessage;
   if (!InitApplication(hinstance))
      return 1;
   if (!InitInstance(hinstance, nCmdShow))
      return 1;

   if(!DoRegisterDeviceInterface( GUID_DEVINTERFACE_DISK, &hDevNotify ))
      // GUID_DEVINTERFACE_MEDIUMCHANGER
      // GUID_DEVINTERFACE_VOLUME
      // GUID_DEVINTERFACE_STORAGEPORT
      // ……
   {
      return 1;
   }
   while ((fGotMessage = GetMessage(&msg, (HWND) NULL, 0, 0)) != 0 && fGotMessage != -1)
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
```

```
    }
    return msg.wParam;
    UNREFERENCED_PARAMETER(lpCmdLine);
}
/************************************
* InitApplication
************************************/
BOOL InitApplication(HINSTANCE hinstance)
{
    WNDCLASSEX wcx;
    wcx.cbSize = sizeof(wcx);
    wcx.style = CS_HREDRAW | CS_VREDRAW;
    wcx.lpfnWndProc = MainWndProc;
    wcx.cbClsExtra = 0;
    wcx.cbWndExtra = 0;
    wcx.hInstance = hinstance;
    wcx.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wcx.hCursor = LoadCursor(NULL, IDC_ARROW);
    wcx.hbrBackground = GetStockObject( WHITE_BRUSH);
    wcx.lpszMenuName = "MainMenu";
    wcx.lpszClassName = "MainWClass";
    wcx.hIconSm = LoadImage(hinstance,
        MAKEINTRESOURCE(5),
        IMAGE_ICON,
        GetSystemMetrics(SM_CXSMICON),
        GetSystemMetrics(SM_CYSMICON),
        LR_DEFAULTCOLOR);
    return RegisterClassEx(&wcx);
}
/************************************
* InitInstance
************************************/
BOOL InitInstance(HINSTANCE hinstance, int nCmdShow)
{
    HWND hwnd;
    hinst = hinstance;
    hwnd = CreateWindow(
        "MainWClass",
        "device monitor",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,  CW_USEDEFAULT,
        CW_USEDEFAULT,  CW_USEDEFAULT,
        (HWND) NULL, (HMENU) NULL, hinstance, (LPVOID) NULL);

    if (!hwnd)
        return FALSE;
    hWnd = hwnd;
    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);
    return TRUE;

}
/************************************
* MainWndProc
************************************/
LRESULT CALLBACK MainWndProc(
                            HWND hwnd,
                            UINT uMsg,
                            WPARAM wParam,
                            LPARAM lParam)
{
```

```
    switch (uMsg)
    {
    case WM_DEVICECHANGE:
        OnDeviceChange ( hwnd, wParam, lParam);
        break;
    case WM_DESTROY:
        ExitProcess(0);
        return 0;
    default:
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
    return 0;
}

/************************************
* void OnDeviceChange (HWND , WPARAM , LPARAM )
* 功能: 处理 WM_DEVICECHANGE 消息
*       参数为窗口消息
************************************/
void OnDeviceChange (HWND hwnd, WPARAM wParam, LPARAM lParam)
{
    // 获取 PDEV_BROADCAST_HDR 参数
    PDEV_BROADCAST_HDR lpdb = (PDEV_BROADCAST_HDR)lParam;
    char szMsg[256];
    HDC hdc = GetDC(hwnd);
    DWORD dwStrLen;
    SIZE sz;
    TEXTMETRIC tm;
    switch(wParam)
    {
    case DBT_DEVICEARRIVAL:
        // 插入 CD 或者 DVD 到光驱中
        if (lpdb -> dbch_devicetype == DBT_DEVTYP_VOLUME)
        {
            // 获取参数
            PDEV_BROADCAST_VOLUME lpdbv = (PDEV_BROADCAST_VOLUME)lpdb;
            // 判断标志
            if (lpdbv -> dbcv_flags & DBTF_MEDIA)
            {
                wsprintf (szMsg, "Drive %c: Media has arrived.",
                    // 根据 mask 获取盘符
                    // dbcv_unitmask 的 bit 依次表示 A:, B:..., 如果为 1, 表示是该盘符
                    FirstDriveFromMask(lpdbv ->dbcv_unitmask));
            }
            else
                // 标志不配置, 打印事件 ID
                wsprintf (szMsg, "Message: %u.", wParam);
        }
        else
            // 设置类型不配置, 打印事件 ID
            wsprintf (szMsg, "Message: %u.", wParam);
        break;
    case DBT_DEVICEREMOVECOMPLETE:
        // 光驱弹出
        if (lpdb -> dbch_devicetype == DBT_DEVTYP_VOLUME)
        {
            // 获取参数
            PDEV_BROADCAST_VOLUME lpdbv = (PDEV_BROADCAST_VOLUME)lpdb;
            // 检查标志
            if (lpdbv -> dbcv_flags & DBTF_MEDIA)
```

```
            {
                wsprintf (szMsg, "Drive %c: Media was removed.",
                    FirstDriveFromMask(lpdbv ->dbcv_unitmask));
            }
            else
                wsprintf (szMsg, "Message: %u.", wParam);
        }
        else
            wsprintf (szMsg, "Message: %u.", wParam);
        break;
    default:
        // 其他设备改变情况，打印设备 ID
        wsprintf (szMsg, "Message: %u.", wParam);
    }
    // 在界面上输出设备变更消息
    dwStrLen = lstrlen(szMsg);
    // 计算输出位置
    GetTextExtentPoint32(hdc, szMsg, dwStrLen, &sz);
    YStart += sz.cy;
    GetTextMetrics(hdc, &tm);
    YStart -= tm.tmOverhang;
    // TODO 窗口滚动
    TextOut(hdc, 10, YStart, szMsg, dwStrLen);
    DeleteDC( hdc );    // 释放 DC
}

/*************************************
* char FirstDriveFromMask (ULONG unitmask)
* 功能：将盘符信息 mask 转换为盘符
*************************************/
char FirstDriveFromMask (ULONG unitmask)
{
    char i;
    for (i = 0; i < 26; ++i)
    {
        if (unitmask & 0x1)
            break;
        unitmask = unitmask >> 1;
    }
    return (i + 'A');
}

/*************************************
* DoRegisterDeviceInterface
* 功能：调用 RegisterDeviceNotification
*          API 函数注册设备变更广播消息，
*          所注册类型的设置发生变更后，窗口会收到消息
*************************************/
BOOL DoRegisterDeviceInterface(
                        GUID InterfaceClassGuid,
                        HDEVNOTIFY *hDevNotify
                        )
{
    DEV_BROADCAST_DEVICEINTERFACE NotificationFilter;
    char szMsg[80];
    // 填充 DEV_BROADCAST_DEVICEINTERFACE 结构，作为参数
    ZeroMemory( &NotificationFilter, sizeof(NotificationFilter) );
    NotificationFilter.dbcc_size =
        sizeof(DEV_BROADCAST_DEVICEINTERFACE);
    // 所需监视的设备类型
    NotificationFilter.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
```

```
    // 类型的 GUID，可以是 GUID_DEVINTERFACE_DISK 等
    NotificationFilter.dbcc_classguid = InterfaceClassGuid;
    // 调用 RegisterDeviceNotification API
    *hDevNotify = RegisterDeviceNotification( hWnd,
        &NotificationFilter,
        DEVICE_NOTIFY_WINDOW_HANDLE
        );
    // 判断调用结果，返回
    if(!*hDevNotify)
    {
        wsprintf(szMsg, "RegisterDeviceNotification failed: %d\n",
            GetLastError());
        MessageBox(hWnd, szMsg, "Registration", MB_OK);
        return FALSE;
    }
    return TRUE;
}
```

2．运行结果

G：盘是光驱，图 16-1 所示第 1 行是光驱中放入新的光盘后的输出，2～9 行是插入 U 盘后的输出，第 10 行是弹出光驱后的输出，之后是卸载 U 盘时的输出。

32768 是常量 DBT DEVICEARRIVAL 的值，32772 是常量 DBT DEVICEREMOVECO MPLETE 的值。

在为硬盘新建分区时，也会有 DBT DEVCEARRIVAL 等消息。读者可以在虚拟机里进行实验。

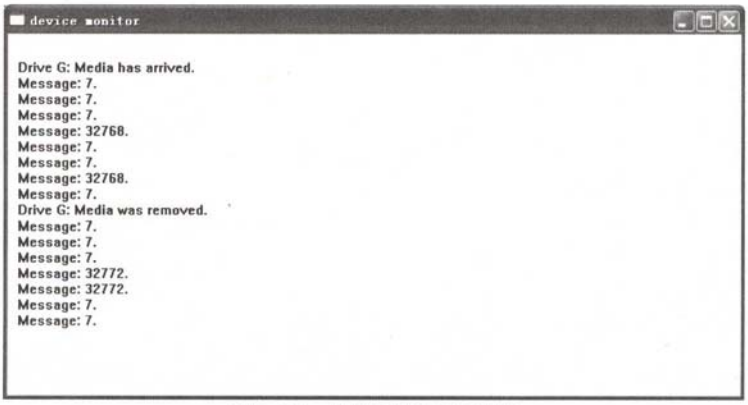

图 16-1　硬件变化监视结果

16.2　I/o控制、内核通信

对内核驱动模块的控制需要通过 SCM 服务控制器，本书已经在第 8 章介绍了服务控制器的概念与程序设计的内容。在通过服务控制器创建了驱动类型的服务后，可以通过控制服务的启动、运行、停止来控制启动程序的运行与停止。

16.2.1　加载驱动程序

加载驱动程序，运行、停止等都是通过服务控制器，由于在第 8 章已经详细介绍了原理，这里不再赘述。直接给出下面的驱动加载程序实例。

实例 16-3 驱动程序的加载和运行

本实例除了实现了驱动的加载和运行外，还实现了对驱动的控制。实例中的 InstallDriver 函数通过服务控制器创建了驱动服务。

StartDriver 函数启动了服务的运行，实际也就是运行了驱动。

StopDriver 函数通过停止服务来停止了驱动的运行。

RemoveDriver 函数删除了由 InstallDriver 所创建的驱动。

所加载的驱动程序文件名为 xIoctl.sys，与本实例编译生成的驱动控制程序在同一个目录下。

GetDriverPath 的功能就是构造了驱动可执行文件的路径。在构造时使用了常量 DRIVER NAME，常量定义于 xioctl.h 文件中，定义如下。这个头文件中包括了在驱动程序和驱动控制程序中都会使用到的一些常量。

在 maln 函数中除了调用上述几个函数完成了驱动的加载和运行外，可向驱动程序发送 I/O 控制码。如何控制驱动将在 16.2.2 小节介绍。

```
#define DRIVER_NAME      "xIoctl"
```

驱动控制程序源代码如下：

```
/* ************************************
* install.c 控制、内核通信
**************************************/
/* 头文件 */
#include <windows.h>
#include <winioctl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "..\sys\xioctl.h"
/* 全局变量 */
char OutputBuffer[100];
char InputBuffer[100];
/* 函数声明 */
BOOL InstallDriver( SC_HANDLE, LPCTSTR, LPCTSTR );
BOOL RemoveDriver( SC_HANDLE, LPCTSTR );
BOOL StartDriver( SC_HANDLE , LPCTSTR );
BOOL StopDriver( SC_HANDLE , LPCTSTR );

/************************************
* InstallDriver
* 功能: 创建服务、安装驱动
* 参数: SchSCManager, 服务控制器句柄
*       DriverName, 服务名
*       ServiceExe, 驱动的可执行程序路径
************************************/
BOOL InstallDriver(
                    SC_HANDLE SchSCManager,
                    LPCTSTR DriverName,
                    LPCTSTR ServiceExe
                    )
{
   SC_HANDLE schService;
   DWORD err;
   // 调用 CreateService 创建服务
   schService = CreateService(SchSCManager, // 服务控制器, SCM 句柄
      DriverName, // 服务名
      DriverName, // 服务的显示名
      SERVICE_ALL_ACCESS, // 存取权限
      SERVICE_KERNEL_DRIVER, // 服务类型
```

```
        SERVICE_DEMAND_START, // 启动类型
        SERVICE_ERROR_NORMAL, // 启动错误的处理
        ServiceExe, // 可执行程序
        NULL, NULL, NULL, NULL, NULL
        );
    if (schService == NULL)
    {
        // 创建服务失败
        err = GetLastError();
        // 服务已经存在
        if (err == ERROR_SERVICE_EXISTS)
        {
            return TRUE;// 返回成功
        }
        else
        {
            // 输出错误信息，返回失败
            printf("CreateService failed! Error = %d \n", err );
            return FALSE;
        }
    }
    // 创建成功，关闭服务
    if (schService)
    {
        CloseServiceHandle(schService);
    }
    // 返回成功
    return TRUE;
}
/*************************************
* RemoveDriver
* 功能：删除驱动服务
* 参数：SchSCManager，服务控制器句柄
*       DriverName，服务名
*************************************/
BOOL RemoveDriver(
                SC_HANDLE SchSCManager,
                LPCTSTR DriverName
                )
{
    SC_HANDLE schService;
    BOOLEAN rCode;
    // 打开服务
    schService = OpenService(SchSCManager,
        DriverName,
        SERVICE_ALL_ACCESS
        );
    if (schService == NULL)
    {
        // 服务打开失败
        printf("OpenService failed! Error = %d \n", GetLastError());
        return FALSE;
    }
    // 删除服务
    if (DeleteService(schService))
    {
        rCode = TRUE;
    }
    else
    {
        //失败
```

```
        printf("DeleteService failed! Error = %d \n", GetLastError());
        rCode = FALSE;
    }
    // 关闭服务句柄
    if (schService)
    {
        CloseServiceHandle(schService);
    }
    return rCode;
}
/************************************
* StartDriver
* 功能: 启动服务, 加载执行驱动
* 参数: SchSCManager, 服务控制器句柄
*       DriverName, 服务名
************************************/
BOOL StartDriver(
          SC_HANDLE SchSCManager,
          LPCTSTR DriverName
          )
{
    SC_HANDLE schService;
    DWORD err;
    // 打开服务
    schService = OpenService(SchSCManager,
        DriverName,
        SERVICE_ALL_ACCESS
        );
    if (schService == NULL)
    {
        // 失败
        printf("OpenService failed! Error = %d \n", GetLastError());
        return FALSE;
    }
    // 启动服务
    if (!StartService(schService, // 服务句柄
        0, // 参数个数, 无
        NULL // 参数指针, 无
        ))
    {
        // 启动失败
        err = GetLastError();
        // 已经开始运行
        if (err == ERROR_SERVICE_ALREADY_RUNNING)
        {
            // 返回成功
            return TRUE;
        }
        else
        {
            // 失败, 打印错误
            printf("StartService failure! Error = %d \n", err );
            return FALSE;
        }
    }
    // 关闭服务句柄
    if (schService)
    {
        CloseServiceHandle(schService);
    }
    return TRUE;
```

```
}

/*************************************
* StopDriver
* 功能：停止服务，停止驱动运行
* 参数：SchSCManager，服务控制器句柄
*       DriverName，服务名
*************************************/
BOOL StopDriver(
         SC_HANDLE SchSCManager,
         LPCTSTR DriverName
         )
{
    BOOLEAN rCode = TRUE;
    SC_HANDLE schService;
    SERVICE_STATUS serviceStatus;
    // 打开服务
    schService = OpenService(SchSCManager,
        DriverName,
        SERVICE_ALL_ACCESS
        );

    if (schService == NULL)
    {
        // 失败
        printf("OpenService failed! Error = %d \n", GetLastError());
        return FALSE;
    }
    // 停止运行
    if (ControlService(schService,
        SERVICE_CONTROL_STOP,
        &serviceStatus
        ))
    {
        rCode = TRUE;
    }
    else
    {
        // 失败
        printf("ControlService failed! Error = %d \n", GetLastError() );
        rCode = FALSE;
    }
    // 关闭服务句柄
    if (schService)
    {
        CloseServiceHandle (schService);
    }
    return rCode;

}

/*************************************
* GetDriverPath
* 功能：获得服务驱动的路径
* 参数：DriverLocation，返回驱动的路径
*************************************/
BOOL GetDriverPath(
                    LPSTR DriverLocation
                    )
{
```

```
    DWORD driverLocLen = 0;

    // 驱动.sys 文件在本程序同一目录下
    driverLocLen = GetCurrentDirectory(MAX_PATH,
        DriverLocation
        );
    if (!driverLocLen)
    {
        printf("GetCurrentDirectory failed! Error = %d \n", GetLastError());
        return FALSE;
    }
    // 构造路径，加上驱动名
    lstrcat(DriverLocation, "\\");
    lstrcat(DriverLocation, DRIVER_NAME);
    lstrcat(DriverLocation, ".sys");
    return TRUE;
}

/*************************************
* int _cdecl main( )
* 功能：加载驱动，进行控制
*************************************/
int _cdecl main()
{
    HANDLE hDevice;
    BOOL bRc;
    ULONG bytesReturned;
    DWORD errNum = 0;
    UCHAR driverLocation[MAX_PATH];
    SC_HANDLE schSCManager;// 服务控制器句柄
    // 打开服务控制器，后续安装、启动都会使用到
    schSCManager = OpenSCManager(NULL, // 本机
        NULL, // 本机数据库
        SC_MANAGER_ALL_ACCESS // 存取权限
        );
    if (!schSCManager)
    {
        // 打开失败
        printf("Open SC Manager failed! Error = %d \n", GetLastError());
        return 1;
    }
    // 获得驱动文件的路径
    if (!GetDriverPath(driverLocation))
    {
        return 1;
    }
    // 安装驱动服务
    if (InstallDriver(schSCManager,
        DRIVER_NAME,
        driverLocation
        ))
    {
        // 安装成功，启动服务，运行驱动
        if(!StartDriver(schSCManager, DRIVER_NAME ))
        {
            printf("Unable to start driver. \n");
            return 1;
        }
    }
    else
    {
```

```
    // 安装失败，删除驱动。
    RemoveDriver(schSCManager, DRIVER_NAME );
    printf("Unable to install driver. \n");
    return 1;
}
// 打开驱动，获得控制所用的句柄
// 由驱动创建的符号链接
hDevice = CreateFile( "\\\\.\\IoctlTest",
    GENERIC_READ | GENERIC_WRITE,
    0,
    NULL,
    CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL,
    NULL);
if ( hDevice == INVALID_HANDLE_VALUE )
{
    printf ( "Error: CreatFile Failed : %d\n", GetLastError());
    return 1;
}
// 打印，输入输出
printf("InputBuffer Pointer = %p, BufLength = %d\n", InputBuffer,
    sizeof(InputBuffer));
printf("OutputBuffer Pointer = %p BufLength = %d\n", OutputBuffer,
    sizeof(OutputBuffer));
// 输入到内核的数据
lstrcpy(InputBuffer,
    "This String is from User Application; using IOCTL_XIOCTL_BUFFER");
printf("\nCalling DeviceIoControl IOCTL_XIOCTL_BUFFER:\n");
// 清空输出缓存
memset(OutputBuffer, 0, sizeof(OutputBuffer));
// 进行 I/O 控制
bRc = DeviceIoControl ( hDevice,// 句柄
    (DWORD) IOCTL_XIOCTL_BUFFER,// IOCTL
    &InputBuffer,// 输入数据
    strlen ( InputBuffer )+1,// 输入数据的长度
    &OutputBuffer,// 输出数据
    sizeof( OutputBuffer),// 输出数据长度
    &bytesReturned,// 实际输出的数据长度
    NULL
    );
// 判断是否成功
if ( !bRc )
{
    printf ( "Error in DeviceIoControl : %d", GetLastError());
    return 1;
}
// 打印从内核输出的内容
printf(" OutBuffer (%d): %s\n", bytesReturned, OutputBuffer);
// 关闭句柄
CloseHandle ( hDevice );
// 停止运行
StopDriver(schSCManager,
    DRIVER_NAME
    );
// 删除服务
RemoveDriver(schSCManager,
    DRIVER_NAME
    );
// 关闭服务控制器
```

```
    CloseServiceHandle (schSCManager);
    return 0;
}
```

本实例中的 mam 函数除了完成驱动程序的加载之外，还控制了设备，与设备进行了通信。

如本章开始所述，设备驱动程序的运行的环境与一般应用程序有很大区别。设备驱动程序运行于系统内核，具有 ring0 优先级（有兴趣的读者可以参考 Intel 程序开发手册和 DDK 开发文档），可以直接访问硬件，可以访问所有物理内容。

但是用户态的应用程序不能直接访问内核驱动程序，也不能访问内核的内存空间。所有的用户态应用程序与内核程序的运行都必须以 I/O 控制的方式来进行。操作系统会提供相关的机制来保存用户态程序和内核态程序之间的正常数据传输。

要控制一个设备，首先需要打开设备驱动程序，获得设备驱动的句柄，相关的 I/O 控制和通信函数才能知道需要控制的是哪个设备驱动。打开设备驱动使用函数 CreateFile。所有对设备的控制都使用 API 函数 DeviceIoControl。DeviceIoControl 函数向设备发送各种不同的 I/O 控制码(IOCTL)，每种 I/O 控制码完成特定的控制功能或者获取特定内容的数据。

DeviceIoControl 函数的原型如下：

```
BOOL DeviceIoControl(
  HANDLE hDevice,
  DWORD dwIoControlCode,
  LPVOID lpInBuffer,
  DWORD nInBufferSize,
  LPVOID lpOutBuffer,
  DWORD nOutBufferSize,
  LPDWORD lpBytesReturned,
  LPOVERLAPPED lpOverlapped
);
```

hDevice 参数是由 CreateFile 打开的设备句柄。dwIoControlCode 参数是 I/O 控制码。函数返回 BOOL 型数据，表示控制是否成功。

按照 16.3 节所述的方法，编译生成 xIoctl.sys。将 xIoctl.sys 与本实例编译生成的 exe 文件放在同一个目录下运行。

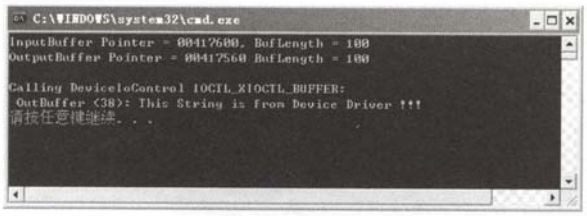

图 16-2　控制驱动程序

如图 16-2 所示，程序打印出了从内核驱动中输出的数据“This String is from Device Driver!!!”。

16.2.2　控制驱动程序、与驱动程序进行通信

16.2.1 小节实例 16-3 在加载了内核驱动程序后，已经使用 DeviceIoControl 函数对驱动程序进行了通信，驱动程序返回了字符串，本节演示应用程序不加载驱动，但是与系统中常驻设备驱动程序进行通信的过程。系统常驻的设备驱动程序，一般都会提供多种 I/O 控制码用于控制，函数也是使用 DeviceIoControl。如实例 16-4 所示。

实例 16-4 控制设备驱动程序

不同于实例 16-3，本实例控制的驱动程序并不是自行加载的，而是系统中已经存在的

设备驱动程序。Windows 系统已经为大多数的常见设备编写了驱动程序。系统的正常运行离不开这些驱动程序。实例 16-1 所列举的驱动程序中，绝大多数是由 Windows 系统自带的驱动程序所创建的。

其中比较重要的是文件系统驱动、网络驱动等。对这些驱动程序所创建的设备，也可以通过 DeviceIoControl 函数进行控制。可以向这些驱动发送不同的 I/O 控制码实现各种不同的功能。很多文件系统的 I/O 控制码已经文档化，本例中所使用的 I/O 控制码都可以在 MSDN 或 SDK 文档中查到。

本实例通过使用 IOCTL_STORAGE_ EJECT MEDIA 和 FSCTL_GETNTFS_VOLUME_DATA 两 AI IOTCL 实现了对弹出光盘和获取 NTFS 分区相关信息的功能。

```
/* ************************************
* ioctl.c I/O控制、内核通信
************************************/
/* 头文件 */
#include <Windows.h>
#include <Winioctl.h>
#include <stdio.h>
/* 函数声明 */
DWORD EnjectCdrom(LPSTR szCdRomName);
DWORD PrintNTFSInfo(LPSTR szVolumeName);

/************************************
* main
* 功能    -cdrom <盘符>, 弹出光盘
*       -ntfs <盘符>, 显示NFTS分区的信息
************************************/
int main(int argc, char* argv[])
{
   CHAR szName[64];
   if(argc == 3)
   {
      // 构造设备名
      wsprintf(szName, "\\\\.\\%s.", argv[2]);
      // 弹出光盘
      if(lstrcmp(argv[1],"-cdrom") == 0)
      {
         EnjectCdrom( szName );
         return 0;
      }
      // 获取NTFS分区详细信息
      if(lstrcmp(argv[1],"-ntfs") == 0)
      {
         PrintNTFSInfo( szName );
         return 0;
      }
   }
   // 使用方法
  printf("usage: \n\t %s -cdrom <volume>\n\t %s -ntfs <volume>\nlike this: \n\t -cdrom G:",
      argv[0], argv[0]);
   return 0;
}

/************************************
* DWORD EnjectCdrom(LPSTR szCdRomName)
* 功能: 弹出指定的光盘
```

```
* 参数: szCdRomName, 设备名
*************************************/
DWORD EnjectCdrom(LPSTR szCdRomName)
{
    HANDLE hDevice;
    DWORD dwBytesReturned;
    hDevice = CreateFile(szCdRomName, // 设备名
        GENERIC_ALL, // 存取权限
        FILE_SHARE_READ|FILE_SHARE_WRITE|FILE_SHARE_DELETE, // 共享方式
        NULL, // 默认安全属性
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL);
    if (hDevice == INVALID_HANDLE_VALUE)
    {
        printf("Could not open file (error %d)\n", GetLastError());
        return 0;
    }
    // 发送 IOCTL
    if(!DeviceIoControl(
        (HANDLE) hDevice, // 设备句柄
        IOCTL_STORAGE_EJECT_MEDIA, // 控制码
        NULL, // 输入缓存
        0, // 输入缓存大小
        NULL, // 输出缓存
        0, // 输出缓存大小
        &dwBytesReturned, // 实际需要的输入缓存大小
        NULL // 非 OVERLAPPED
        ))
    {
        printf("DeviceIoControl error (%d)",GetLastError());
        return 0;
    }
    return 1;
}
/*************************************
* DWORD PrintNTFSInfo(LPSTR szVolumeName)
* 功能: 获取显示指定的 NTFS 驱动器信息
* 参数: szVolumeName, 设备名
*************************************/
DWORD PrintNTFSInfo(LPSTR szVolumeName)
{
    // FSCTL_GET_NTFS_VOLUME_DATA I/O 控制的返回值保存在
    // NTFS_VOLUME_DATA_BUFFER 结构中
    NTFS_VOLUME_DATA_BUFFER nvdb;
    DWORD dwBufferSize;
    HANDLE hDevice;
    // 清空参数
    ZeroMemory(&nvdb,sizeof(nvdb));
    hDevice = CreateFile(szVolumeName,
        GENERIC_ALL,
        FILE_SHARE_READ|FILE_SHARE_WRITE,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL);
    if (hDevice == INVALID_HANDLE_VALUE)
    {
        printf("Could not open file (error %d)\n", GetLastError());
        return 0;
    }
    if(DeviceIoControl(
```

去 www.msdn.microsoft.com/en-us/查找.

typedef struct
LARGE_INTEGER VolumeSeria
LARGE_INTEGER NumberSec
LARGE_INTEGER Tota

```
            hDevice, // 设备句柄
            FSCTL_GET_NTFS_VOLUME_DATA, // 控制码
            NULL, // 输入缓存
            0, // 输入缓存大小
            &nvdb, // 输出缓存
            sizeof( NTFS_VOLUME_DATA_BUFFER ), // 输出缓存大小
            &dwBufferSize, // 返回的实际数据大小
            NULL // 非 OVERLAPPED
            ))
        {
            // 打印获取的信息
            printf("SerialNumber %lu\n",nvdb.VolumeSerialNumber);
            printf("Starting logical cluster number of the master file table: %lu\n",nvdb.Mft
StartLcn);
            printf("Length of the master file table: %lu\n",nvdb.MftValidDataLength);
            printf("... ...\n");
        }
        else
        {
            printf("DeviceIoControl error: (%d)\n",GetLastError());
            return 0;
        }
        return 1;
    }
```

编译生成 ioctl.exe。运行：

```
    ioctl.exe -cdrom <光驱盘符>
```

光驱会被弹出。

运行：

```
    ioctl.exe -ntfs <NTFS 分区盘符>
```

分显出 NTFS 分区的详细信息，包括 MFT（主文件表）的相关信息等。

在笔者主机上运行：

```
    ioctl.exe -ntfs C:
```

如图 16-3 所示，对文件系统驱动程序进行控制，可以得到多种与文件系统相关的信息。

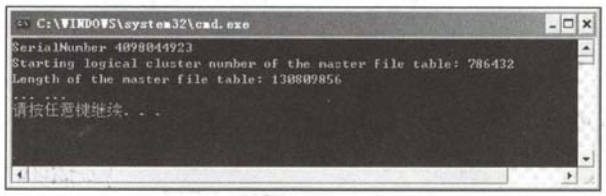

图 16-3　对文件系统驱动进行控制

16.3　编写设备驱动程序

如前所述，驱动程序与应用程序运行在不同的环境中，驱动程序运行于系统内核，所以不能调用 Windows API，而只能调用内核态的 API 函数。

设备驱动程序开发涉及较多的概念，开发环境也不完全相同。本书不对设备驱动程序的开发作详细介绍。仅在本节列举简单的示例说明设备驱动程序与应用程序之间的控制方法与通信过程。

首先介绍驱动程序开发所使用的开发工具包 Driver Development Dit (DDK)的基本使用方法。然后再通过 16.2 节所提到的 xIoctl.sys 的源代码来介绍驱动程序的基本结构。

16.3.1 驱动程序开发包：DDK

开发驱动程序，需要使用微软提供的驱动程序开发包 DDK。DDK 中包括各种编译和连接工具、头文件和库文件，以及适用于各个操作系统发行版本的编译环境设备的批处理文件等，当然还包括文档。

安装DDK之后，可以从开始菜单中找到DDK为各个操作系统发行版本所提供的编译环境。

实例 16-5 编译 xloctl. sys

从开始菜单中找到编译环境，以 Windows xp 系统为例，执行“开始菜单\Development Dit\Windows DDK… \ Build Environments\Windows XP\ Windows XP Checked Build Environment”笔者使用 Windows Server 2003 DDK)，在出现的命令提示中，切换到 xioctl.sys 源代码所在的目录（16.3.2 节给出了源代码，或从本书配套光盘中获得）运行：

```
build
```

之后会在 objchk_wxp_x86\i386 目录中找到 xioctl.sys 文件。

> **注意** 一般 DDK 的编译环境区分为 Checked 和 Free 两种，Checked 会生成调试版本的驱动程序。

16.3.2 开发驱动程序

使用 DDK 编译驱动程序，除了源代码之外还需要有 makefile 文件和 sources 文件。makefile 文件指
明了编译时所使用的指令和参数。sources 文件指明了生成的目标文件、编译时所使用的源代码文件等。

1. makefile 文件

DDK 已经提供了编译驱动所需的 makefile 文件。一个驱动程序的 makefile 只需要包括 DDK 定义好的 makefile 文件即可。

xioctl.sys 的 makefile 文件：

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

2. sources 文件

sources 文件定义编译时需要使用的源代码文件、编译后生成的目标文件及驱动程序的类型等。

```
xioctl.sys 的 sources 文件:
TARGETNAME=xioctl
TARGETPATH=obj
TARGETTYPE=DRIVER

SOURCES=xioctl.c xioctl.rc
```

3. 源代码文件和头文件

一个程序必须有源代码文件和头文件。sources 文件中的 SOURCES 项指明了编译一个驱动程序所需要的源代码文件。实例 16-6 给出了 xioctl.sys 的源代码。

4. 实例 16-6 驱动程序

本实例是 xioctl.sys 的源代码。

驱动程序的入口函数是 DriverEntry。DriverEntry 函数的原型如下：

```
NTSTATUS DriverEntry(
    IN PDRIVER_OBJECT  DriverObject,
    IN PUNICODE_STRING  RegistryPath
    );
```

其中 DriverObj ect 参数是驱动对象，DRIVER_OBJECT 结构如下：

```
typedef struct _DRIVER_OBJECT {
    CSHORT Type;
    CSHORT Size;
    PDEVICE_OBJECT DeviceObject;
    ULONG Flags;
    PVOID DriverStart;
    ULONG DriverSize;
    PVOID DriverSection;
    PDRIVER_EXTENSION DriverExtension;
    UNICODE_STRING DriverName;
    PUNICODE_STRING HardwareDatabase;
    PFAST_IO_DISPATCH FastIoDispatch;
    PDRIVER_INITIALIZE DriverInit;
    PDRIVER_STARTIO DriverStartIo;
    PDRIVER_UNLOAD DriverUnload;
    PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
} DRIVER_OBJECT;
typedef struct _DRIVER_OBJECT *PDRIVER_OBJECT; // ntndis
```

驱动对象的 Maj orFunction 成员和 DriverUnload 成员是必须由驱动程序赋值。

Maj orFunction 是一个数组，其值是一系列被称作派遣(Dispatch)函数的回调函数的函数地址，在驱动开始运行、停止运行或有 I/O 控制码到达的情况下会调用这些函数。而驱动程序的主要功能应该在这些回调函数中实现。

DriverUnload 的值是也是回调函数指针。这个函数用于处理在驱动卸载时需要进行的操作。

Majoriunction 数组中最重要的项是 IRP_MJ_DEVICE_coNTROL 这个项的回调函数用于处理 I/O 控制函数原型如下：

```
NTSTATUS ***DispatchDeviceControl(
    IN PDEVICE_OBJECT  DeviceObject,
    IN PIRP  Irp
    );
```

函数名可以任意，参数的返回值必须符合接口形式。

其中 DeviceObj ect 参数是设备对象。Irp 参数是 IRP 类型的指针，IRP 的全称是 I/O 请求包(I/Orequest packet)。用户态程序通过 DeviceIoControl 向设备驱动程序发送的 I/O 控制都会由操作系统封装为 IRP。然后传递给 Maj orFunction[IRI，_MJ- DEVICE_CONTRO]所指定的函数。IRP 结构较复杂，其中包括 I/O 控制码(Parameters.DeviceIoControl.IoControlCode)，输入缓存与输出缓存(AssociatedIrp.SystemBuffer)等信息。

如果一个驱动程序需要和用户态程序进行通信，需要创建设备对象和符号链接。所创建的符号链接与用户态驱动控制程序使用 CreateFile 所打开的设备名需对应。

创建设备对象和符号连接使用函数 IoCreateDevice 和 IoCreateSymbolicLink。

xioctl.h 文件源代码如下：

```
/* ************************************
* xioctl.h  I/O控制、内核通信
**************************************/

// 内核驱动和用户态控制程序都需要使用到的常量定义
#define XIOCTL_TYPE 40000
```

```
#define IOCTL_XIOCTL_BUFFER \
    CTL_CODE( XIOCTL_TYPE, 0x902, METHOD_BUFFERED, FILE_ANY_ACCESS )

#define DRIVER_NAME       "xIoctl"
```

xioctl.c 文件源代码如下：

```
/* ************************************
* xioctl.c I/O控制、内核通信
**************************************/
/* 头文件 */
#include <ntddk.h>// 包括了很多NT内核的类型、结构、函数定义，开发驱动时需要包括此头文件
#include <string.h>
#include "xioctl.h"
/* 常量与预定义 */
#define NT_DEVICE_NAME      L"\\Device\\XIOCTL"
#define DOS_DEVICE_NAME     L"\\DosDevices\\IoctlTest"

#if DBG
#define XIOCTL_KDPRINT(_x_) \
                DbgPrint("XIOCTL.SYS: ");\
                DbgPrint _x_;
#else
#define XIOCTL_KDPRINT(_x_)
#endif

/* 函数声明 */
NTSTATUS DriverEntry(PDRIVER_OBJECT  DriverObject, PUNICODE_STRING RegistryPath);
NTSTATUS XioctlCreateClose(PDEVICE_OBJECT DeviceObject, PIRP Irp);
NTSTATUS XioctlDeviceControl(PDEVICE_OBJECT DeviceObject, PIRP Irp );
VOID XioctlUnloadDriver(PDRIVER_OBJECT DriverObject );
VOID PrintIrpInfo( PIRP Irp );
VOID PrintChars( PCHAR BufferAddress, ULONG CountChars );

#ifdef ALLOC_PRAGMA
#pragma alloc_text( INIT, DriverEntry )
#pragma alloc_text( PAGE, XioctlCreateClose)
#pragma alloc_text( PAGE, XioctlDeviceControl)
#pragma alloc_text( PAGE, XioctlUnloadDriver)
#pragma alloc_text( PAGE, PrintIrpInfo)
#pragma alloc_text( PAGE, PrintChars)
#endif // ALLOC_PRAGMA

/**************************************
* DriverEntry
* 功能：驱动的入口函数，分配了相关处理例程
**************************************/
NTSTATUS
DriverEntry(
   IN OUT PDRIVER_OBJECT   DriverObject,
   IN PUNICODE_STRING     RegistryPath
   )
{
   NTSTATUS       ntStatus;
   UNICODE_STRING  ntUnicodeString;            // 设备名
   UNICODE_STRING  ntWin32NameString;          // Win32 设备名
   PDEVICE_OBJECT  deviceObject = NULL;         // 设备对象
   RtlInitUnicodeString( &ntUnicodeString, NT_DEVICE_NAME );
   // 创建设备
   ntStatus = IoCreateDevice(
      DriverObject,                       // 驱动对象 DriverEntry 的参数
      0,                                  // 不使用设备扩展
```

```
        &ntUnicodeString,                    // 设备名 "\Device\XIOCTL"
        FILE_DEVICE_UNKNOWN,                 // 设备类型
        FILE_DEVICE_SECURE_OPEN,
        FALSE,
        &deviceObject );                     // 设备对象
    if ( !NT_SUCCESS( ntStatus ) )
    {
        XIOCTL_KDPRINT(("Couldn't create the device object\n"));
        return ntStatus;
    }
    // 初始化处理例程
    DriverObject->MajorFunction[IRP_MJ_CREATE] = XioctlCreateClose;// 创建时会调用
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = XioctlCreateClose;// 关闭时会调用
    // 处理 I/O 控制
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = XioctlDeviceControl;
    DriverObject->DriverUnload = XioctlUnloadDriver;// 卸载时会调用
    // WIN32 设备名
    RtlInitUnicodeString( &ntWin32NameString, DOS_DEVICE_NAME );
    // 在设备名和 WIN32 设备名之间创建符号连接
    ntStatus = IoCreateSymbolicLink(
                    &ntWin32NameString, &ntUnicodeString );
    if ( !NT_SUCCESS( ntStatus ) )
    {
        XIOCTL_KDPRINT(("Couldn't create symbolic link\n"));
        IoDeleteDevice( deviceObject );
    }
    return ntStatus;
}

/*************************************
* XioctlCreateClose
* 功能：驱动对象的处理例程，由 DriverEntry 指定
*             本函数中驱动对象在创建和关闭时调用的例程
*             没有实际的功能，只是将状态设置为"成功"
*************************************/
NTSTATUS
XioctlCreateClose(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;

    IoCompleteRequest( Irp, IO_NO_INCREMENT );

    return STATUS_SUCCESS;
}
/*************************************
* XioctlUnloadDriver
* 功能：卸载驱动时调用的例程，
*       删除符号连接，删除设备
*************************************/
VOID
XioctlUnloadDriver(
    IN PDRIVER_OBJECT DriverObject
    )
{
    PDEVICE_OBJECT deviceObject = DriverObject->DeviceObject;
    UNICODE_STRING uniWin32NameString;
    RtlInitUnicodeString( &uniWin32NameString, DOS_DEVICE_NAME );
```

```
    // 删除符号连接
    IoDeleteSymbolicLink( &uniWin32NameString );
    // 删除设备
    if ( deviceObject != NULL )
    {
        IoDeleteDevice( deviceObject );
    }
}
/*************************************
* XioctlDeviceControl
* 功能: 处理 I/O 控制的例程
*************************************/
NTSTATUS
XioctlDeviceControl(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    PIO_STACK_LOCATION  irpSp;// 当前栈的位置
    NTSTATUS            ntStatus = STATUS_SUCCESS;// 执行状态, 成功\失败
    ULONG               inBufLength; // 输入缓存大小
    ULONG               outBufLength; // 输出缓存大小
    PCHAR               inBuf, outBuf; // 输入输出缓存
    PCHAR               data = "This String is from Device Driver !!!";
    ULONG               datalen = strlen(data)+1;//输出数据的长度
    PMDL                mdl = NULL;
    PCHAR               buffer = NULL;
    // 处理 IRP
    irpSp = IoGetCurrentIrpStackLocation( Irp );
    inBufLength = irpSp->Parameters.DeviceIoControl.InputBufferLength;
    outBufLength = irpSp->Parameters.DeviceIoControl.OutputBufferLength;
    if(!inBufLength || !outBufLength)
    {
        ntStatus = STATUS_INVALID_PARAMETER;
        goto End;
    }
    // 判断 IOCTL
    switch ( irpSp->Parameters.DeviceIoControl.IoControlCode )
    {
    case IOCTL_XIOCTL_BUFFER:
           // 显示收到的 IRP
        XIOCTL_KDPRINT(("Called IOCTL_SIOCTL_METHOD_BUFFERED\n"));
        PrintIrpInfo(Irp);
        // 设备 IN OUT 缓存
        inBuf = Irp->AssociatedIrp.SystemBuffer;
        outBuf = Irp->AssociatedIrp.SystemBuffer;
        // 从输入缓存中获得信息
        XIOCTL_KDPRINT(("\tData from User : %s", inBuf));
        // 复制数据到输出缓存
        strncpy(outBuf, data, outBufLength);
               // 打印输出缓存的内容
        XIOCTL_KDPRINT(("\tData to User : %s", outBuf));
        // 设置 IRP
        Irp->IoStatus.Information = (outBufLength<datalen?outBufLength:datalen);
       break;
        // 还可以定义其他 I/O 控制码

    default:
        // 处理其他已知的 I/O code
        ntStatus = STATUS_INVALID_DEVICE_REQUEST;
        XIOCTL_KDPRINT(("ERROR: unrecognized IOCTL %x\n",
```

```
            irpSp->Parameters.DeviceIoControl.IoControlCode));
        break;
    }
End:
    // 设备状态，完成 IPR 处理
    Irp->IoStatus.Status = ntStatus;
    IoCompleteRequest( Irp, IO_NO_INCREMENT );
    return ntStatus;
}

/*************************************
* PrintIrpInfo
* 功能：打印 IPR 信息
*************************************/
VOID
PrintIrpInfo(
    PIRP Irp)
{
    PIO_STACK_LOCATION  irpSp;
    irpSp = IoGetCurrentIrpStackLocation( Irp );
    XIOCTL_KDPRINT(("\tIrp->AssociatedIrp.SystemBuffer = 0x%p\n",
        Irp->AssociatedIrp.SystemBuffer));
    XIOCTL_KDPRINT(("\tIrp->UserBuffer = 0x%p\n", Irp->UserBuffer));
    XIOCTL_KDPRINT(("\tirpSp->Parameters.DeviceIoControl.Type3InputBuffer = 0x%p\n",
        irpSp->Parameters.DeviceIoControl.Type3InputBuffer));
    XIOCTL_KDPRINT(("\tirpSp->Parameters.DeviceIoControl.InputBufferLength = %d\n",
        irpSp->Parameters.DeviceIoControl.InputBufferLength));
    XIOCTL_KDPRINT(("\tirpSp->Parameters.DeviceIoControl.OutputBufferLength = %d\n",
        irpSp->Parameters.DeviceIoControl.OutputBufferLength ));
    return;
}
```

分析代码可以知道，当收到 IOCTL_XIOCTL_BUFFER I/O 控制码时，程序将输出缓存的内容设置为字符串“This String is from Device Driver!!!”，如图 16-2 所示。

5．程序运行结果

实例 16-3 的驱动控制程序在加载了本驱动后与驱动程序进行通信后的打印结果，如图 16-2 所示，驱动控制程序得到了驱动程序所输出的字符串。

在实例 16-6 驱动程序代码中，调用了多次 DbgPrint 函数，内核级的调试器（例如 WinDbg）可以查看驱动程序通过 DbgPrint 函数打印出的调用语句。

16.4 I/O模式，同步与异步

本书在前面多个章节介绍到了输入输出，包括文件输入输出、进程间通信、网络通信、内核通信等。凡是输入输出都是处理“流”形式的数据，所具有的共同的特征是都需要占用一定的时间。

根据 I/O 进行时发起 I/O 的用户线程是否等待 I/O 过程结束后再运行，可以将 I/O 模式分为同步 I/O 和异步 I / O 两种。

1．两种 IO 模式的区别

I/O 模式决定了发起 I/O 的用户线程在 I]O 进行的时间段是继续运行还是暂停运行。如果发起 I / O 的用户线程在 I/O 进行的过程中暂停运行，直到 I/O 结束，进行 I/O 操作的函数返回，用户线程才继续进行，那么这就是同步 I/O 模式。如果用户线程发起 I/O 后，将 I/O 过程交由系统内核以独立的内核线程运行，在内核进行 I/O 的同时，用户线程继续运行，则是异步 I/O 模式。

如图 16-4 所示。

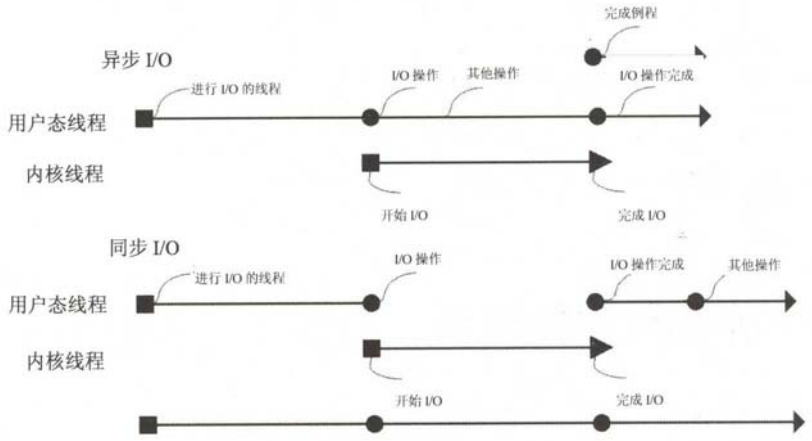

图 16-4　同步 IO 与异步 IO

2．各自的优缺点

本书中绝大多数应用都使用同步 I/0，但是同步 I/0 与异步 I / 0 各有各的优势，各有各的应用场合。

同步 I/0 可以保证操作在 I/0 完成之后再进行，大大降低了程序的复杂程度，不会因为程序设计上的缺陷而引起输入输出数据异常，甚至系统异常。

而异步 I/0 在系统进行 I/0 操作时，不会阻塞程序，并可同时进行其他工作。一个进程在进行 I/0 操作的同时，若要执行其他需要立即响应的操作，那么使用异步 I/0 模式就不需要再新建线程。另一个优势是在进行异步 I / 0 操作时，还可取消(CancelIo)该 I/0。因此异步 I/0 在编写大型程序时，应用更为广泛。

第 17 章　用户、认证和对象安全

Windows 系统具有很完善的安全和认证机制，称作访问控制机制。程序的执行主体（线程）在访问对象（文件、事件等）时，系统会根据线程的“权限”和线程需要访问的对象所具有的访问控制列表(ACL)中的“安全描述符”是否匹配来进行认证，决定一个线程是否可以操作一个对象。

17.1 节将详细介绍关于权限、访问控制列表、安全描述符等在安全认证中所依赖的数据结构，并重点讲解安全认证的过程。

17.2 节将通过实例说明安全认证的具体过程和程序设计方法。

最后 17.3 节将介绍如何对系统中的用户操作，包括增加删除用户，将用户增加到用户组，修改用户的权限等。

17.1　基本概念

A 需要访问(Access)B，A 就是访问的主体，B 就是访问的客体。A 的“访问令牌”和 B 的安全描述符共同决定了 A 是否可以访问 B。

访问的主体是进程。在系统中，线程才是程序执行的流程，因此只有线程才能操作对象。每个线程都是属于一个进程的，线程并没有属于自己的权限，而是来源于线程所属于的进程。一个进程中的所有线程都具有同样的权限，可以把进程看作访问的主体。一个线程能访问哪些对象，能进行哪此操作，是由线程权限决定的。

访问的客体是安全对象，所有被访问的对象都具有安全描述符，包括了文件、注册表、事件(Event)、互斥(Mutex)、管道等。

17.1.1　访问令牌、权限和用户标识

进程的权限继承自创建进程用户和用户所属的用户组。用户有专用数据结构来表示权限—访问令牌(Access Token)。访问令牌包括两个部分：一个是令牌所表示的用户，包括用户标识符(SID)，用户所属的用户组等；另一部分是“权限”(Privilege)。

在进程访问安全对象时，会用到 SID。每个安全对象都有访问控制列表(ACL)，ACL 说明了哪些用户(SID)能访问本对象，哪些不能，以及能进行哪种访问等。而“权限”在访问某个具体的安全对象时并没有作用，“权限”是表示进程是否能够进行特定的系统操作，如关闭系统、修改系统时间、加载设备驱动等。

创建进程的 API 函数是 CreateProcess，CreateProcess 函数所创建的进程使用的访问令牌是当前登录用户的访问令牌。

此外还可以指定进程的用户。使用 CreateProcessAsUser 和 CreateProcessWithTokenW 等 API 函数，在创建前需要先得到用户的令牌，可以使用 LogonUser 登录用户（是否可以同时登录多个用户受操作系统版本限制），LogonUser 函数用返回用户的令牌。

如果需要得到进程和线程的访问令牌，可以使用 OpenProcessToken、OpenThreadToken 等函数。获取令牌中的信息可以使用 API 函数 GetTokenInformation。如果需要修改权限，可以使用 AdjustTokenPrivileges 等函数。

实例 17-1 中的示例代码详细说明了令牌的结构，令牌中用户信息和权限信息的获取和修改方法。

17.1.2 进程的系统操作权限

进程的权限特指进程是否能够进行各种系统操作，例如是否可以关闭系统，是否能够修改系统时间，是否能够加载设备驱动等。权限是一个列表，每种权限是列表中的一项。权限列表存在于进程的访问令牌中。

权限有很多种，每一种表示了一个特定的操作是否能够进行，如果进程的访问令牌中的权限列表中有这个权限，则表示进程可以进行这种操作，比如 SE_LOAD_DRIVER_ NAME 表示进程可以加载驱动。

SDK 头文件中，对权限的定义如下：

```
#define SE_CREATE_TOKEN_NAME                TEXT("SeCreateTokenPrivilege")
#define SE_ASSIGNPRIMARYTOKEN_NAME          TEXT("SeAssignPrimaryTokenPrivilege")
#define SE_LOCK_MEMORY_NAME                 TEXT("SeLockMemoryPrivilege")
#define SE_INCREASE_QUOTA_NAME              TEXT("SeIncreaseQuotaPrivilege")
#define SE_UNSOLICITED_INPUT_NAME           TEXT("SeUnsolicitedInputPrivilege")
#define SE_MACHINE_ACCOUNT_NAME             TEXT("SeMachineAccountPrivilege")
#define SE_TCB_NAME                         TEXT("SeTcbPrivilege")
#define SE_SECURITY_NAME                    TEXT("SeSecurityPrivilege")
#define SE_TAKE_OWNERSHIP_NAME              TEXT("SeTakeOwnershipPrivilege")
#define SE_LOAD_DRIVER_NAME                 TEXT("SeLoadDriverPrivilege")
#define SE_SYSTEM_PROFILE_NAME              TEXT("SeSystemProfilePrivilege")
#define SE_SYSTEMTIME_NAME                  TEXT("SeSystemtimePrivilege")
#define SE_PROF_SINGLE_PROCESS_NAME         TEXT("SeProfileSingleProcessPrivilege")
#define SE_INC_BASE_PRIORITY_NAME           TEXT("SeIncreaseBasePriorityPrivilege")
#define SE_CREATE_PAGEFILE_NAME             TEXT("SeCreatePagefilePrivilege")
#define SE_CREATE_PERMANENT_NAME            TEXT("SeCreatePermanentPrivilege")
#define SE_BACKUP_NAME                      TEXT("SeBackupPrivilege")
#define SE_RESTORE_NAME                     TEXT("SeRestorePrivilege")
#define SE_SHUTDOWN_NAME                    TEXT("SeShutdownPrivilege")
#define SE_DEBUG_NAME                       TEXT("SeDebugPrivilege")
#define SE_AUDIT_NAME                       TEXT("SeAuditPrivilege")
#define SE_SYSTEM_ENVIRONMENT_NAME          TEXT("SeSystemEnvironmentPrivilege")
#define SE_CHANGE_NOTIFY_NAME               TEXT("SeChangeNotifyPrivilege")
#define SE_REMOTE_SHUTDOWN_NAME             TEXT("SeRemoteShutdownPrivilege")
#define SE_UNDOCK_NAME                      TEXT("SeUndockPrivilege")
#define SE_SYNC_AGENT_NAME                  TEXT("SeSyncAgentPrivilege")
#define SE_ENABLE_DELEGATION_NAME           TEXT("SeEnableDelegationPrivilege")
#define SE_MANAGE_VOLUME_NAME               TEXT("SeManageVolumePrivilege")
#define SE_IMPERSONATE_NAME                 TEXT("SeImpersonatePrivilege")
```

```
#define SE_CREATE_GLOBAL_NAME               TEXT("SeCreateGlobalPrivilege")
#define SE_TRUSTED_CREDMAN_ACCESS_NAME      TEXT("SeTrustedCredManAccessPrivilege")
#define SE_RELABEL_NAME                     TEXT("SeRelabelPrivilege")
#define SE_INC_WORKING_SET_NAME             TEXT("SeIncreaseWorkingSetPrivilege")
#define SE_TIME_ZONE_NAME                   TEXT("SeTimeZonePrivilege")
#define SE_CREATE_SYMBOLIC_LINK_NAME        TEXT("SeCreateSymbolicLinkPrivilege")
```

实例 17-1 中的示例代码列举了进程权限，说明了如何修改进程权限。

17.1.3 安全对象

Windows 系统几乎所有的对象都有安全属性，包括文件、文件夹、注册表、线程同步对象、进程间通信对象、网络共享等，进程和线程也可以是其他进程的操作对象，所以进程和线程也是安全对象。

在创建对象时都可以指定对象的安全属性，比如 CreateFile、CreatePipe、CreateProcess、RegCreateKeyEx 和 RegSaveKeyEx 等，SECURITY_ATTRIBUTES 结构用于指定对象的安全属性。

GetNamedSecurityInfo 、 GetSecurityInfo 、 SetSecurityInfo 、SetKernelObjectSecurity、SetNamedSecurityInfo 等 API 函数可以获取和设置对象的安全属性。

对象的安全属性是以安全描述符(Security Descriptor)的形式存在的，安全描述符中包括了访问控制列表。

17.1.4 访问控制列表(ACL)

每个安全对象都有访问控制列表。访问控制列表有两种，一种是选择访问控制列表（discretionary access control list，DACL)，另一种是系统访问控制列表(system access controllist，SACL)。DACL 决定了用户或用户组是否能访问这个对象，SACL 控制了尝试访问安全对象的检测信息的继承关系。

DACL 是访问控制的关键，如图 17-1 所示。DACL 中包括一个访问控制入口(AccessControl Entries，ACE)列表。ACE 表明了用户（通过用户 SID 或用户组 SID）是否能进行操作以及能进行哪种操作。在进行访问控制检测时，会依次检测 DACL 中的 ACE，直到被允许或被拒绝，如图 17-2 所示。

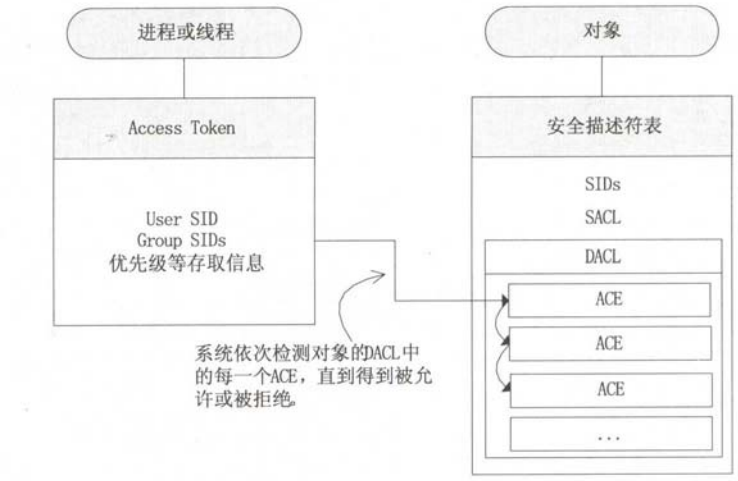

图 17-1 访问令牌和安全控制符的认证关系

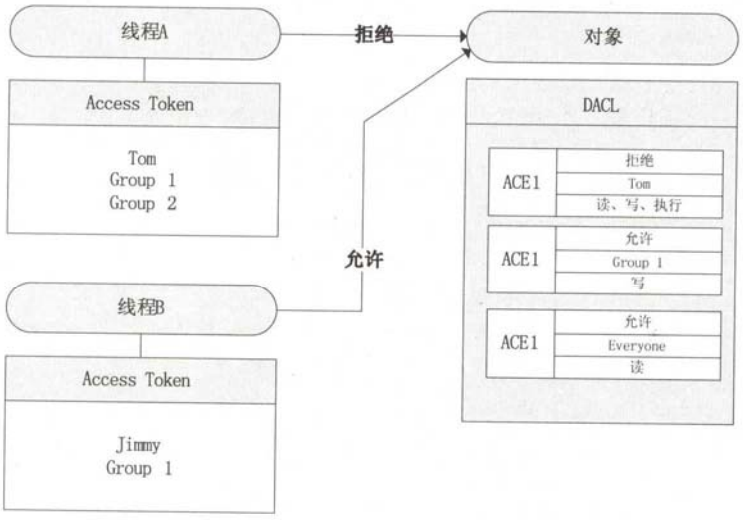

图 17-2　ACE 的内容以及线程访问令牌中的 SID 和 DACL 中的 ACE 的认证过程

17.2　安全机制程序示例

本节通过实例说明访问令牌和安全描述符访问控制列表等内容和程序设计的方法。包括：

◇列举进程访问令牌内容和权限；

◇修改进程的权限；

◇显示安全描述符的内容，列举 DACL;

◇修改对象的安全描述符。

17.2.1　列举进程访问令牌内容和权限

本节使用 Windows Server 2003 SDK 中的示例来说明如何获取进程访问令牌的内容。包括两个可执行程序：MyToken.exe 和 TextSid.exe。

1．实例 17-1　显示进程的访问令牌内容

MyToken.exe 源代码位于 SDK 安装目录下，Samples\Security\Authorization\MyToken。在编译环境中切换到此目标，使用 nmake 编译和链接，生成 MyToken.exe。运行 MyToken.exe 会打印出本进程的访问令牌的内容。

```
User : WIN-CODE\wrt
Owner : BUILTIN\Administrators
Primary Group : WIN-CODE\None
LUID for this instance of token 7561011464
LUID for this logon session    15899164
Token type is PRIMARY
Token source is <User32 >

Retrieving Group information from the access token
SID 0 Group: WIN-CODE\None
SID 1 Group: Everyone
SID 2 Group: BUILTIN\Administrators
SID 3 Group: BUILTIN\Users
SID 4 Group: NT AUTHORITY\INTERACTIVE
SID 5 Group: NT AUTHORITY\Authenticated Users
```

```
SID 6 Group: NT AUTHORITY\This Organization
SID 7 Group: NONE_MAPPED
SID 8 Group: LOCAL
SID 9 Group: NT AUTHORITY\NTLM Authentication
SID 10 Group: Mandatory Label\High Mandatory Level

Privileges associated with this token (23)
SeIncreaseQuotaPrivilege - (attributes) 0
SeSecurityPrivilege - (attributes) 0
SeTakeOwnershipPrivilege - (attributes) 0
SeLoadDriverPrivilege - (attributes) 0
SeSystemProfilePrivilege - (attributes) 0
SeSystemtimePrivilege - (attributes) 0
SeProfileSingleProcessPrivilege - (attributes) 0
SeIncreaseBasePriorityPrivilege - (attributes) 0
SeCreatePagefilePrivilege - (attributes) 0
SeBackupPrivilege - (attributes) 0
SeRestorePrivilege - (attributes) 0
SeShutdownPrivilege - (attributes) 0
SeDebugPrivilege - (attributes) 0
SeSystemEnvironmentPrivilege - (attributes) 0
SeChangeNotifyPrivilege - (attributes) 3
SeRemoteShutdownPrivilege - (attributes) 0
SeUndockPrivilege - (attributes) 0
SeManageVolumePrivilege - (attributes) 0
SeImpersonatePrivilege - (attributes) 3
SeCreateGlobalPrivilege - (attributes) 3
SeIncreaseWorkingSetPrivilege - (attributes) 0
SeTimeZonePrivilege - (attributes) 0
SeCreateSymbolicLinkPrivilege - (attributes) 0
```

运行后，打印出了进程所属的用户、用户组、令牌的类型、源和 LUID，然后打印出的令牌中包括了用户组 SID 及所对应的用户组名，最后打印出进程的权限。

查看程序代码，mam 函数调用了 DisplayCallerAcc es sTokenInformation 函数，DisplayCallerAccessTokenInformation函数调用了OpenThreadToken和OpenProcessToken API 函数，获取得了 Token 的句柄，然后调用 DisplayTokenInformation 函数获取并显示 Token 中的内容。

```
BOOL DisplayCallerAccessTokenInformation()
{
  HANDLE hToken = NULL;
  BOOL bResult = FALSE;

  // 首先使用 OpenThreadToken()函数来判断
  // 线程运行状态
  bResult = OpenThreadToken(GetCurrentThread(),
    TOKEN_QUERY | TOKEN_QUERY_SOURCE, TRUE, &hToken);
  if (bResult == FALSE && GetLastError() == ERROR_NO_TOKEN)
  {
    // 否则，使用进程入口标记
    bResult = OpenProcessToken(GetCurrentProcess(),
      TOKEN_QUERY | TOKEN_QUERY_SOURCE, &hToken);
  }
  if (bResult)
  {
    bResult = DisplayTokenInformation(hToken);
    CloseHandle(hToken);
  }
  else
    MyPrintf(_T("OpenThread/ProcessToken failed with %d\n"), GetLastError());

  return bResult;
}
```

DisplayTokenInformation 函数分别调用了 DisplayUserInfo、DisplayOwnerInfo、DisplayPrimaryGroupInfo、DisplayStatistics、DisplaySource、DisplayGroupsInfo 和 DisplayPrivileges 等函数分别显示了用户名、用户组和权限等内容。各函数的实现原理类似，都是调用了 Retrieve TokenInformationClass 函数从令牌中获得信息，然后根据信息的类型，进行转换并显示，如果获取的是 SID 则调用 ConvprtBinarySidToName 等 API 函数获取 SID 对应的用户名并显示。

其中 DisplayUserInfo、DisplayStatistics、DisplayGroupsInfo 代码如下：

```
BOOL DisplayUserInfo(HANDLE hToken)
{
  TOKEN_USER *pUserInfo = NULL;
  DWORD dwSize = 0;
  LPTSTR pName = NULL;
  // 获取用户信息
  pUserInfo = (TOKEN_USER *)RetrieveTokenInformationClass(hToken, TokenUser, &dwSize);
  if (pUserInfo == NULL)
  {
    return FALSE;
  }
  pName = ConvertBinarySidToName(pUserInfo->User.Sid);
  if (pName == NULL)
  {
    return FALSE;
  }
  MyPrintf(_T("User : %s\n"), pName);
  CheckAndLocalFree(pUserInfo);
  CheckAndLocalFree(pName);
  return TRUE;
}

BOOL DisplayStatistics(HANDLE hToken)
{
  TOKEN_STATISTICS *pStatistics = NULL;
  DWORD dwSize = 0;
  //获得 Token 的统计信息
  pStatistics=(TOKEN_STATISTICS *)RetrieveTokenInformationClass(hToken, TokenStatistics,
&dwSize);
  if (pStatistics == NULL)
  {
    return FALSE;
  }
  // 列出 Token 的统计信息
  MyPrintf(_T("LUID for this instance of token %i64\n"), pStatistics->TokenId);
  MyPrintf(_T("LUID for this logon session    %i64\n"), pStatistics->AuthenticationId);
  if (pStatistics->TokenType == TokenPrimary)
    MyPrintf(_T("Token type is PRIMARY\n"));
  else
    MyPrintf(_T("Token type is IMPERSONATION\n"));
  CheckAndLocalFree(pStatistics);
  return TRUE;
}

BOOL DisplayGroupsInfo(HANDLE hToken)
{
  TOKEN_GROUPS *pGroupInfo = NULL;
  DWORD dwSize = 0;
  DWORD i;
  // 列出 Token 中的所有用户组
  pGroupInfo=(PTOKEN_GROUPS)RetrieveTokenInformationClass(hToken, TokenGroups, &dwSize);
  if (pGroupInfo == NULL)
```

```
    {
      return FALSE;
    }

    MyPrintf(_T("\nRetrieving Group information from the access token\n"));
    for (i = 0; i < pGroupInfo->GroupCount; i++)
    {
      LPTSTR pName = NULL;
      pName = ConvertBinarySidToName(pGroupInfo->Groups[i].Sid);
      if (pName != NULL)
      {
        MyPrintf(_T("SID %d Group: %s\n"), i, pName);
        CheckAndLocalFree(pName);
      }
    }
    CheckAndLocalFree(pGroupInfo);
    return TRUE;
}
```

各函数中，主要调用了两个函数：RetrieveTokenInformationClass 和 ConvertBinarySidToName，

ConvertBinarySidToName 是 API 函数，通过令牌中的用户或用户组 SID 得到了用户名。

RetrieveTokenInformationClass 函数的功能是从 Token 中获取信息。

RetrieveTokenInformationClass 函数的 InfoClass 参数是 TOKEN_INFORMATION_CLASS 枚举类型，指定不同的值会获取不同的信息。InfoClass 最后传递给了 GetTokenInformation API 函数。比如 TokenUser 用于获取用户名信息。

```
LPVOID RetrieveTokenInformationClass(
     HANDLE hToken,
     TOKEN_INFORMATION_CLASS InfoClass,
     LPDWORD lpdwSize)
{
   LPVOID pInfo = NULL;
   BOOL fSuccess = FALSE;
   __try
   {
     *lpdwSize = 0;
     // 确定缓冲区大小
     GetTokenInformation(
          hToken,
          InfoClass,
          NULL,
          *lpdwSize, lpdwSize);
     if (GetLastError() != ERROR_INSUFFICIENT_BUFFER)
     {
       MyPrintf(_T("GetTokenInformation failed with %d\n"), GetLastError());
       __leave;
     }
     // 为 token 信息分配缓冲区
     pInfo = LocalAlloc(LPTR, *lpdwSize);
     if (pInfo == NULL)
     {
       MyPrintf(_T("LocalAlloc failed with %d\n"), GetLastError());
       __leave;
     }
     if (!GetTokenInformation(
          hToken,
          InfoClass,
          pInfo,
          *lpdwSize, lpdwSize))
     {
```

```
            MyPrintf(_T("GetTokenInformation failed with %d\n"), GetLastError());
            __leave;
        }
        fSuccess = TRUE;
    }
    __finally
    {
        // 如果失败，则释放 pDomainAndUserName，
        // 否则，调用完之后再释放
        if (fSuccess == FALSE)
        {
            CheckAndLocalFree(pInfo);
        }
    }

    return pInfo;
}
```

2．实例 17-2 显示 SID

TextSid.exe 源代码位于 SDK 安装目录下的 Samples\Security\Authorization\TextSid。运行 TextSid.exe 会打印出本进程用户的 SID。

```
Process Sid: S-1-5-21-4151681340-611662300-2080981886-1005
```

获取 SID 较为简单，LookupAccountName API 函数就可以实现。TextSid.exe 的关键功能是将 SID 使用字符串的形式打印出来，其中涉及了 SID 的格式和解析问题。

实例的关键函数是 GetTextualSid，函数代码如下：

```
BOOL
GetTextualSid(
    PSID pSid,                  // 二进制 SID
    LPTSTR TextualSid,          // 为文本表述的 SID 设置缓冲区
    LPDWORD cchSidSize          // 需要/提供文本的大小
    )
{
    PSID_IDENTIFIER_AUTHORITY psia;
    DWORD dwSubAuthorities;
    DWORD dwCounter;
    DWORD cchSidCopy;
    DWORD cchMaxLen;
    // 检查参数是否顺利通过，ValidSid 不可以接收参数为空
    if(!pSid || !IsValidSid(pSid) || !TextualSid || !cchSidSize)
    {
        SetLastError(ERROR_INVALID_PARAMETER);
        return FALSE;
    }
    // 获取 SidIdentifierAuthority
    psia = GetSidIdentifierAuthority(pSid);
    // 获取 sidsubauthority 的数量
    dwSubAuthorities = *GetSidSubAuthorityCount(pSid);
    // 估算空间的大小
    // S-SID_REVISION- + identifierauthority + -subauthorities + NULL
    cchMaxLen = 6 + 14 + (11 * dwSubAuthorities) + 1;
    // 检查提供的空间大小
    // 如果长度不够，指出适合的大小并设定 Lasterror
    if(*cchSidSize < cchMaxLen) {
        *cchSidSize = cchMaxLen;
        SetLastError(ERROR_INSUFFICIENT_BUFFER);
        return FALSE;
    }
```

```
    // 预备 S-SID_REVISION
    cchSidCopy = wnsprintf(TextualSid,
                        cchMaxLen,
                        TEXT("S-%lu-"),
                        SID_REVISION );
    // 预备 SidIdentifierAuthority
    if ( (psia->Value[0] != 0) || (psia->Value[1] != 0) ) {
       cchSidCopy += wnsprintf(TextualSid + cchSidCopy,
                  cchMaxLen - cchSidCopy,
                  TEXT("0x%02hx%02hx%02hx%02hx%02hx%02hx"),
                  (USHORT)psia->Value[0],
                  (USHORT)psia->Value[1],
                  (USHORT)psia->Value[2],
                  (USHORT)psia->Value[3],
                  (USHORT)psia->Value[4],
                  (USHORT)psia->Value[5]);
    } else {
       cchSidCopy += wnsprintf(TextualSid + cchSidCopy,
                  cchMaxLen - cchSidCopy,
                  TEXT("%lu"),
                  (ULONG)(psia->Value[5]      )   +
                  (ULONG)(psia->Value[4] <<  8)   +
                  (ULONG)(psia->Value[3] << 16)   +
                  (ULONG)(psia->Value[2] << 24)   );
    }
    // 循环获取
    for(dwCounter = 0 ; dwCounter < dwSubAuthorities ; dwCounter++) {
       cchSidCopy += wnsprintf(TextualSid + cchSidCopy,
                  cchMaxLen - cchSidCopy,
                  TEXT("-%lu"),
                  *GetSidSubAuthority(pSid, dwCounter) );
    }
    // 返回给调用函数，缓冲区中的字节数
    *cchSidSize = cchSidCopy;
    return TRUE;
}
```

从打印的内容可以看出用户 SID 的格式。

17.2.2 修改进程的权限

在获得进程的权限之后，还可以对进程的权限进行修改，以使进程可以进行特定的系统操作。

使用 AdjustTokenPrivileges 函数可以修改令牌的权限。

实例 17-3 修改进程的权限

本实例演示了如何使用 AdjustTokenPrivileges 函数提供权限。SE_TIME_ZONE NAME 常量在 SDK 头文件中有定义，是众多权限中的一个。

EnablePrivilege 实现了修改进程的权限，代码如下：

```
BOOL EnablePrivilege (LPSTR name)
{
    HANDLE hToken;
    BOOL rv;
    TOKEN_PRIVILEGES priv = { 1, {0, 0, SE_PRIVILEGE_ENABLED} };
    LookupPrivilegeValue (
        0,
        name,
        &priv.Privileges[0].Luid
```

```
        );
    OpenProcessToken(
        GetCurrentProcess (),
        TOKEN_ADJUST_PRIVILEGES,
        &hToken
        );
    AdjustTokenPrivileges (
        hToken,
        FALSE,
        &priv,
        sizeof priv,
        0,
        0
        );
    rv = GetLastError () == ERROR_SUCCESS;
    CloseHandle (hToken);
    return rv;
}

void main(int argc, TCHAR **argv)
{
    EnablePrivilege (SE_TIME_ZONE_NAME);
}
```

17.2.3 列举安全对象的安全描述符

使用 Windows Server 2003 SDK 中的示例来说明如何获取安全对象的安全描述符内容。

实例 17-4 获取安全描述符

Check_SD.exe 源代码位于 SDK 安装目录下的 Samples\Security\Authorization\Check_SD。

Check_SD.exe 可以获取各种对象的安全描述符，这里以文件对象为例。

```
Check_SD.exe -f Check_SD.c
```

运行：

会打印出本进程的访问令牌的内容。

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>              SECURITY INFORMATION                >>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

object name ........ Check_SD.c
object type ........ file

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>        SECURITY DESCRIPTOR INFORMATION           >>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

valid ............... yes
length .............. 188 byte(s)
revision ........... 1
control bits ....... 0x9404
.................... SE_DACL_PRESENT
.................... SE_SELF_RELATIVE
.................... SE_DACL_AUTO_INHERITED
.................... SE_DACL_PROTECTED

owner ............... BUILTIN\Administrators
sid ................. S-1-5-32-544
sid type ........... Alias
```

```
group ..... WIN-CODE\None
sid ..... S-1-5-21-4151681340-611662300-2080981886-513
sid type ..... Group

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>              DACL INFORMATION                  >>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

valid ..... yes
revision ..... 2

ace count ..... 4
acl bytes in use ... 124 byte(s)
acl bytes free ..... 0 byte(s)

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>               ACE #1                           >>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

type ..... ACCESS_DENIED_ACE
flags ..... 0x0
size ..... 36 byte(s)
mask ..... 0x1F01FF
..... FILE_READ_DATA
..... FILE_WRITE_DATA
..... FILE_APPEND_DATA
..... FILE_READ_EA
..... FILE_WRITE_EA
..... FILE_EXECUTE
..... FILE_READ_ATTRIBUTES
..... FILE_WRITE_ATTRIBUTES
..... READ_CONTROL
..... WRITE_OWNER
..... WRITE_DAC
..... DELETE
..... SYNCHRONIZE

user ..... WIN-CODE\Guest
sid ..... S-1-5-21-4151681340-611662300-2080981886-501
sid type ..... User
sid size ..... 28 bytes

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>               ACE #2                           >>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

type ..... ACCESS_ALLOWED_ACE
flags ..... 0x0
size ..... 36 byte(s)
mask ..... 0x1F01FF
..... FILE_READ_DATA
..... FILE_WRITE_DATA
..... FILE_APPEND_DATA
..... FILE_READ_EA
..... FILE_WRITE_EA
..... FILE_EXECUTE
..... FILE_READ_ATTRIBUTES
..... FILE_WRITE_ATTRIBUTES
..... READ_CONTROL
..... WRITE_OWNER
```

```
..... WRITE_DAC
..... DELETE
..... SYNCHRONIZE

user ..... WIN-CODE\Administrator
sid ..... S-1-5-21-4151681340-611662300-2080981886-500
sid type ..... User
sid size ..... 28 bytes

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>                ACE #3                                    >>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

type ..... ACCESS_ALLOWED_ACE
flags ..... 0x0
size ..... 24 byte(s)
mask ..... 0x1F01FF
..... FILE_READ_DATA
..... FILE_WRITE_DATA
..... FILE_APPEND_DATA
..... FILE_READ_EA
..... FILE_WRITE_EA
..... FILE_EXECUTE
..... FILE_READ_ATTRIBUTES
..... FILE_WRITE_ATTRIBUTES
..... READ_CONTROL
..... WRITE_OWNER
..... WRITE_DAC
..... DELETE
..... SYNCHRONIZE

user ..... BUILTIN\Administrators
sid ..... S-1-5-32-544
sid type ..... Alias
sid size ..... 16 bytes

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>                ACE #4                                    >>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

type ..... ACCESS_ALLOWED_ACE
flags ..... 0x0
size ..... 20 byte(s)
mask ..... 0x1F01FF
..... FILE_READ_DATA
..... FILE_WRITE_DATA
..... FILE_APPEND_DATA
..... FILE_READ_EA
..... FILE_WRITE_EA
..... FILE_EXECUTE
..... FILE_READ_ATTRIBUTES
..... FILE_WRITE_ATTRIBUTES
..... READ_CONTROL
..... WRITE_OWNER
..... WRITE_DAC
..... DELETE
..... SYNCHRONIZE

user ..... NT AUTHORITY\SYSTEM
sid ..... S-1-5-18
sid type ..... Well Known Group
sid size ..... 12 bytes

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>              NO SACL PRESENT                             >>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

从运行结果可以看到，Check_SD.c 的所有者是 BUILTIN\Administrators，SID 为

S-1-5-32-544。在 DACL 中包括 4 个 ACE。第一个 ACE 拒绝 Geust 用户进行操作，拒绝操作的类型包括 FILE_READ_DATA、FILE_WRITE_DATA 等，第二个 ACE 允许 Administrator 用户进行操作，并列举了所能操作的类型。每一个 ACE 后都说明了此 ACE 所描述的用户名的 SID 和 SID 类型。可以将结果与文件的安全属性对照，发现结果一致，如图 17-3 所示。

图 17-3　文件的安全属性

查看代码，显示文件的安全描述符主要使用了 DumpFile 函数。pszFile 是文件路径，参数 c 是对象的类型，这里为“f”。

```
    void DumpFile(LPTSTR pszFile, TCHAR c)
    {
        DWORD              dwSize = 0;
        PSECURITY_DESCRIPTOR psd = NULL;
        SECURITY_INFORMATION si  = OWNER_SECURITY_INFORMATION | GROUP_SECURITY_INFORMATION |
DACL_SECURITY_INFORMATION | SACL_SECURITY_INFORMATION;
        // 提升权限
        Privilege(SE_SECURITY_NAME, TRUE);
        // 得到大小
        if (!GetFileSecurity(pszFile, si, psd, dwSize, &dwSize)){
            if (GetLastError() == ERROR_INSUFFICIENT_BUFFER){
                psd = (PSECURITY_DESCRIPTOR)LocalAlloc(LPTR, dwSize);
                if (psd == NULL)
                    DisplayError(GetLastError(), TEXT("LocalAlloc"));
                else if (!GetFileSecurity(pszFile, si, psd, dwSize, &dwSize))
                    DisplayError(GetLastError(), TEXT("GetFileSecurity"));
            }
            else
                DisplayError(GetLastError(), TEXT("GetFileSecurity"));
        }
        else{
            printf("GetFileSecurity succeeded when it was expected to fail.\n");
            ExitProcess(0);
        }
        // 提升权限
        Privilege(SE_SECURITY_NAME, FALSE);
        // 存储的安全性描述
        DumpSD(psd, c);
        // 释放缓冲区
        if (NULL != psd)
        {
            if (LocalFree((HLOCAL)psd))
                DisplayError(GetLastError(), TEXT("LocalFree"));
        }
    }
```

在获得了安全描述符(SD)的指针后，将指针传递给 DumpSD 函数。DumpSD 函数又调用 DumpSDInfo、DumpControl 等函数显示安全描述符中的内容，其中 DumpDacl 是显示 DACL 和 SACL 的函数。DumpDacl 函数的主要流程如下：

```
void DumpDacl(PSECURITY_DESCRIPTOR psd, TCHAR c, BOOL bDacl)
{
    ......
    if (!GetSecurityDescriptorDacl(psd, &bDaclPresent, &pacl, &bDaclDefaulted))
        DisplayError(GetLastError(), TEXT("GetSecurityDescriptorDacTEXT("));
         ......
       dwAceCount = DumpAclInfo(pacl, bDacl);
        for (i = 0; i < (int)dwAceCount; i++)
        {
           if (!GetAce(pacl, i, (LPVOID)&pace))
              DisplayError(GetLastError(), TEXT("GetAce"));
            ......
             switch(pace->Header.AceType)
             {
                  case ACCESS_ALLOWED_ACE_TYPE:
                    _tprintf(TEXT("ACCESS_ALLOWED_ACE\n"));
                    break;
                  case ACCESS_DENIED_ACE_TYPE:
            ......
              }
       switch (c){
        case 'f':
            if ((pace->Mask & FILE_READ_DATA) == FILE_READ_DATA)
                _tprintf(TEXT("..................... FILE_READ_DATA\n"));
            if ((pace->Mask & FILE_WRITE_DATA) == FILE_WRITE_DATA)
                _tprintf(TEXT("..................... FILE_WRITE_DATA\n"));
            if ((pace->Mask & FILE_APPEND_DATA) == FILE_APPEND_DATA)
                _tprintf(TEXT("..................... FILE_APPEND_DATA\n"));
            if ((pace->Mask & FILE_READ_EA) == FILE_READ_EA)
                _tprintf(TEXT("..................... FILE_READ_EA\n"));
            if ((pace->Mask & FILE_WRITE_EA) == FILE_WRITE_EA)
                _tprintf(TEXT("..................... FILE_WRITE_EA\n"));
            if ((pace->Mask & FILE_EXECUTE) == FILE_EXECUTE)
                _tprintf(TEXT("..................... FILE_EXECUTE\n"));
            if ((pace->Mask & FILE_READ_ATTRIBUTES) == FILE_READ_ATTRIBUTES)
                _tprintf(TEXT("..................... FILE_READ_ATTRIBUTES\n"));
            if ((pace->Mask & FILE_WRITE_ATTRIBUTES) == FILE_WRITE_ATTRIBUTES)
                _tprintf(TEXT("..................... FILE_WRITE_ATTRIBUTES\n"));
           break;
        default:
            break;
        }
```

```
                //
                // 对象权限
                //
                if ((pace->Mask & READ_CONTROL) == READ_CONTROL)
                    _tprintf(TEXT("..................... READ_CONTROL\n"));
                ......
                // 常规访问权限
                if ((pace->Mask & GENERIC_ALL) == GENERIC_ALL)
                    _tprintf(TEXT("..................... GENERIC_ALL\n"));
                if ((pace->Mask & GENERIC_EXECUTE) == GENERIC_EXECUTE)
                    _tprintf(TEXT("..................... GENERIC_EXECUTE\n"));
                if ((pace->Mask & GENERIC_READ) == GENERIC_READ)
                    _tprintf(TEXT("..................... GENERIC_READ\n"));
                if ((pace->Mask & GENERIC_WRITE) == GENERIC_WRITE)
                    _tprintf(TEXT("..................... GENERIC_WRITE\n"));
                // 显示 SID
                cbName = sizeof(szName);
                cbReferencedDomainName = sizeof(szReferencedDomainName);
                ZeroMemory(szName, cbName);
                ZeroMemory(szReferencedDomainName, cbReferencedDomainName);
                if (!LookupAccountSid(NULL, &(pace->SidStart),
                 szName, &cbName, szReferencedDomainName, &cbReferencedDomainName, &snu))
                 {
                    if (GetLastError() == ERROR_NONE_MAPPED)
                        LookupAccountOtherSid(
                           &(pace->SidStart), szName, &cbName,
                           szReferencedDomainName,
                           &cbReferencedDomainName, &snu);
                    else
                        DisplayError(GetLastError(), TEXT("LookupAccountSid"));
                }
                _tprintf(TEXT("\nuser ..............%s\\%s\n"), szReferencedDomainName, szName);
                dwSize = sizeof(szSidText);
                ZeroMemory(szSidText, dwSize);
                ConvertSid(&(pace->SidStart), szSidText, &dwSize);
                _tprintf(TEXT("sid ..............%s\n"), szSidText);
                _tprintf(TEXT("sid type ..........%s\n"), szSidType[snu-1]);
                _tprintf(TEXT("sid size ..........%u bytes\n"), GetLengthSid(&(pace->SidStart)));
                  }
            }
          }
        else{
            ......
        }
}
```

首先根据参数 bDacl 判断是需要显示 DACL 还是 SACL，如果需要显示 DACL，调用 GetSecurityDescriptorDacl 获取 DACL 的信息，然后以 ACE 的个数为次数循环，再在循环中调用 GetAce API 函数。

之后分析了 ACE 的类型，是允许还是拒绝，然后根据 Mask 显示出具体的操作，参数 c 是用户的，这里为“f”。

LookupAccountSid 和 LookupAccountOtherSid 函数用于获取 ACE 所说明的用户的用户信息。

DumpDacl 中调用了 DumpAclInfo，获取 ACL 头的相关信息，函数代码如下：

```
DWORD DumpAclInfo(PACL pacl, BOOL bDacl)
{
    ACL_INFORMATION_CLASS   aic;
    BYTE                pByte[2][12];
```

```
    // 有效的 ACL
    _tprintf(TEXT("\n>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n"));
    if (bDacl)
        _tprintf(TEXT(">>         DACL INFORMATION                  >>\n"));
    else
        _tprintf(TEXT(">>              SACL INFORMATION                  >>\n"));
    _tprintf(TEXT(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n\n"));
    _tprintf(TEXT("valid ..............  "));
    if (!IsValidAcl(pacl)){
        _tprintf(TEXT("no\n"));
        return 0;
    }
    else
        _tprintf(TEXT("yes\n"));

    for (aic=1; aic<3; aic++){
        if(!GetAclInformation(pacl, (LPVOID)pByte[aic-1], sizeof(ACL_SIZE_INFORMATION), aic))
            DisplayError(GetLastError(), TEXT("GetAclInformation"));
    }
    _tprintf(TEXT("revision ..........%u\n\n"), *((PACL_REVISION_INFORMATION)pByte[0]));
    _tprintf(TEXT("ace count .......... %u\n"), ((PACL_SIZE_INFORMATION)pByte[1])->AceCount);
    _tprintf(TEXT("acl bytes in use ... %u byte(s)\n"),
       ((PACL_SIZE_INFORMATION)pByte[1])->AclBytesInUse);
    _tprintf(TEXT("acl bytes free ..... %u byte(s)\n"),
       ((PACL_SIZE_INFORMATION)pByte[1])->AclBytesFree);
    return ((PACL_SIZE_INFORMATION)pByte[1])->AceCount;
}
```

上述代码是 SDK 示例 TakeOwn.exe 的一部分，代码位于 Samples\Security\Authorization\TakeOwn。功能是将文件的安全描述符设置为空。

pSecurityDescriptor 参数指向 SECURITY DESCRIPTOR 结构，表示安全描述符。所设置的安全描述符可以从其他对象获得，也可以从由程序创建。

示例如下：

```
BOOL SetFileSecurity(
  LPCTSTR lpFileName,
  SECURITY_INFORMATION SecurityInformation,
  PSECURITY_DESCRIPTOR pSecurityDescriptor
);
```

17.2.4 修改安全描述符

对不同对象的安全描述符的修改使用的 API 函数不同。如果是文件则使用 SetFileSecurity，函数原型如下：

```
InitializeSecurityDescriptor( &SecurityDescriptor, SECURITY_DESCRIPTOR_REVISION );

Result = SetSecurityDescriptorDacl (
          &SecurityDescriptor,
          TRUE,
          NULL,
          FALSE
          );

Result = SetFileSecurity(
          lpFileName,
          DACL_SECURITY_INFORMATION,
          &SecurityDescriptor
          );
```

如图 17-4 和 17-5 所示，选择一个文件，查看属性＼安全，可以看到用户操作的权限。运行后，再查看安全属性，如图 17-5 所示，安全属性已经被清空。

图 17-4　文件安全属性变化前

图 17-5　文件安全属性清空后

17.3　用户

本节介绍 Windows 系统的用户管理接口，介绍如何使用程序实现用户的添加、修改和删除以及修改用户组成员。

17.3.1　创建用户

NetUserAdd API 函数可以实现在系统中创建用户的功能，函数原型如下：

```
NET_API_STATUS NetUserAdd(
  LMSTR servername,
  DWORD level,
  LPBYTE buf,
  LPDWORD parm_err
);
```

servername 参数是需要添加用户的主机名，如果是本机则设置为 NULL。

level 参数决定了 buf 参数的结构，设置为 1、2、3、4 分别表示 buf 参数所指向的内容为 USER_INFO_1、USER_INFO_2、USER_INFO_3、USER_INFO_4。这 4 个结构都包括了用户的相关信息，只是所包括的信息的丰富程度不同。以 USER_INFO_1 为例，包括用户的一些基本信息，用户名、密码、密码期限、优先级、用户 HOME 路径、描述等。

```
typedef struct _USER_INFO_1 {
  LPWSTR usri1_name;
  LPWSTR usri1_password;
  DWORD usri1_password_age;
  DWORD usri1_priv;
  LPWSTR usri1_home_dir;
  LPWSTR usri1_comment;
  DWORD usri1_flags;
  LPWSTR usri1_script_path;
} USER_INFO_1,  *PUSER_INFO_1,  *LPUSER_INFO_1;
```

如果使用 USER INFO-2、USERINFO_3、USER INFO 4，则在创建用户时能指定更多信息。

实例 17-5 创建用户

AddUser 函数实现了创建用户的功能。3 个参数分别是所创建的用户所在的主机、用户

名和密码。

AddUser 函数使用 USER_INFO_1 结构来表示用户信息。调用 NetUserAdd 函数，在增加用户后，判断是否成功。

```
/************************************
* AddUser
* 功能：增加用户
* 参数：szServerName，主机名，如果为本机增加用户，设置为 NULL
*       szUserName，用户名
*       szPassword，密码
************************************/
int AddUser(LPWSTR szServerName,
            LPWSTR szUserName,
            LPWSTR szPassword)
{
    USER_INFO_1 ui;
    DWORD dwLevel = 1;     // 使用 USER_INFO_1 作为参数
    DWORD dwError = 0;
    NET_API_STATUS nStatus;
    // 填充 USER_INFO_1
    ui.usri1_name = szUserName; // 用户名
    ui.usri1_password = szPassword;  // 密码
    ui.usri1_priv = USER_PRIV_USER;  // 级别
    ui.usri1_home_dir = NULL;
    ui.usri1_comment = NULL;
    ui.usri1_flags = UF_SCRIPT;
    ui.usri1_script_path = NULL;
    // 调用 NetUserAdd，增加用户
    nStatus = NetUserAdd(szServerName,
        dwLevel,
        (LPBYTE)&ui,
        &dwError);

    // 判断结果
    if (nStatus == NERR_Success)
    {
        wprintf(stderr, L"User %s has been successfully added on %s\n",
        szUserName, szServerName);
    }
    else
    {
        fprintf(stderr, "A system error has occurred: %d\n", nStatus);
    }
    return 0;
}
```

17.3.2 用户组

NetGroupAddUser 和 NetGroupDelUser 的功能分别是将用户添加到用户组和从用户组删除用户。增加用户组和删除用户组，使用 API 函数：NetGroupAdd 和 NetGroupDel。

(1) NetGroupAddUser。

函数原型如下：

```
NET_API_STATUS NetGroupAddUser(
  LPCWSTR servername,
  LPCWSTR GroupName,
  LPCWSTR username
);
```

(2) NetGroupDelUser。

函数原型如下：

```
NET_API_STATUS NetGroupDelUser(
  LPCWSTR servername,
  LPCWSTR GroupName,
  LPCWSTR Username
);
```

(3) NetGroupAdd。

函数原型如下：

```
NET_API_STATUS NetGroupAdd(
  LPCWSTR servername,
  DWORD level,
  LPBYTE buf,
  LPDWORD parm_err
);
```

(4) NetGroupDel。

函数原型如下：

```
NET_API_STATUS NetGroupDel(
  LPCWSTR servername,
  LPCWSTR groupname
);
```

除了 NetGroupAdd 外，其他 3 个函数的参数设置都较简单，增加用户组 API 函数 NetGroupAdd 和 NetUserAdd 的参数设置类似，level 参数也是指定 buf 参数的类型。buf 参数类型包括 GROUP_INFO_1、GROUP INFO 2、GROUP_INFO_3、GROUP INFO_4，以 GROUP_INFO_2 为例：

```
typedef struct _GROUP_INFO_2 {
  LPWSTR grpi2_name;
  LPWSTR grpi2_comment;
  DWORD grpi2_group_id;
  DWORD grpi2_attributes;
} GROUP_INFO_2,  *PGROUP_INFO_2;
```

实例 17-6 将用户增加到用户组

AddUserToGroup 函数实现将用户增加到用户组的功能，调用 API 函数 NetGroupAddUser，设置主机名、用户组名和用户名。

```
/*************************************
* AddUserToGroup
* 功能：为用户组增加用户
* 参数：szServerName，主机名，如果为本机，设置为 NULL
*       szUserName，用户名
*       szGroup，用户组名
*************************************/
int AddUserToGroup(LPWSTR szServerName,
                   LPWSTR szUserName,
                   LPWSTR szGroup)
{
    NET_API_STATUS nStatus;
    // 调用 NetGroupAddUser
    nStatus =  NetGroupAddUser(
        szServerName,
        szGroup,
        szUserName
        );
```

17.3.3 删除用户

NetUserDelAPI 函数实现删除用户，通过参数指定主机名和要删除的用户名，函数原型如下：

```
    // 判断结果
    if (nStatus == NERR_Success)
        fwprintf(stderr, L"User %s has been successfully added on %s\n",
        szUserName, szServerName);

    else
        fprintf(stderr, "NetGroupAddUser A system error has occurred: %d\n", nStatus);
    return 0;
}
```

实例 17-7 删除用户

DelUser 函数演示了如何调用 NetUserDel。

```
NET_API_STATUS NetUserDel(
  LPCWSTR servername,
  LPCWSTR username
);
```

```
/*************************************
* DelUser
* 功能: 删除用户
* 参数: szServerName, 主机名, 如果为本机, 设置为 NULL
*       szUserName, 用户名
*************************************/
int DelUser(LPWSTR szServerName, LPWSTR szUserName)
{
    DWORD dwError = 0;
    NET_API_STATUS nStatus;

    // 调用 NetUserDel 删除用户
    nStatus = NetUserDel(szServerName, szUserName);
    // 判断并显示结果
    if (nStatus == NERR_Success)
        fwprintf(stderr, L"User %s has been successfully deleted on %s\n",
        szUserName, szServerName);
    else
        fprintf(stderr, "A system error has occurred: %d\n", nStatus);

    return 0;
}
```

17.3.4 列举用户和用户组、获取用户信息

NetUserEnum 和 NetLocalGroupEnum API 函数分别用于列举指定主机中当前的所有用户和用户组。

NetUserEnum 函数原型如下：

```
NET_API_STATUS NetUserEnum(
  LPCWSTR servername,
  DWORD level,
  DWORD filter,
  LPBYTE* bufptr,
  DWORD prefmaxlen,
  LPDWORD entriesread,
  LPDWORD totalentries,
  LPDWORD resume_handle
);
```

servername 参数指定主机名，level 参数指定 bufptr 参数所使用的结构，filter 参数说明了列举用户的类型，bufptr 参数用于返回结果，USER_ INFO_0、USER_INFO_2、USER_INFO_10 等。

根据所需获取信息的不同而设置不同的结构，可获取的信息包括用户名、密码等。

NetLocalGroupEnum 函数的参数设置方法与 NetUserEnum 类似，返回的结果结构根据

level 不同而不同，可以是 LOCALGROUP_INFO_0 和 LOCALGROUP_INFO_1。

```
NET_API_STATUS NetLocalGroupEnum(
  LPCWSTR servername,
  DWORD level,
  LPBYTE* bufptr,
  DWORD prefmaxlen,
  LPDWORD entriesread,
  LPDWORD totalentries,
  PDWORD_PTR resumehandle
);
```

NetUserEnum 和 NetLocalGroupEnum API 函数使用方法详见实例 17-8。

NetUserGetInfo 函数的功能是获取指定用户的信息。所获取的信息根据 level 参数不同而保存在不同的结构中，所能使用的结构与 NetUserEnum 所能使用的结构相同，包括 USER_INFO_0、USER_INFO_2、USER INFO_10 等。

```
NET_API_STATUS NetUserGetInfo(
  LPCWSTR servername,
  LPCWSTR username,
  DWORD level,
  LPBYTE* bufptr
);
```

1．实例 17-8 列举用户和用户组

ListUsers 和 ListGroup 函数分别调用上述两个 API 函数实现了用户的列举和用户组的列举。

```
/*************************************
* int ListUsers(LPWSTR pszServerName)
* 功能：列举用户
* 参数：szServerName，主机名，如果为本机，设置为 NULL
*************************************/
int ListUsers(LPWSTR pszServerName)
{
    LPUSER_INFO_0 pBuf = NULL;
    LPUSER_INFO_0 pTmpBuf;
    DWORD dwLevel = 0;
    DWORD dwPrefMaxLen = MAX_PREFERRED_LENGTH;
    DWORD dwEntriesRead = 0;
    DWORD dwTotalEntries = 0;
    DWORD dwResumeHandle = 0;
    DWORD i;
    DWORD dwTotalCount = 0;
    NET_API_STATUS nStatus;

    // 循环，直到可以成功调用 NetUserEnum
    do
    {
        // 调用 NetUserEnum 函数
        nStatus = NetUserEnum(pszServerName,
            dwLevel,// 这里设置为 0，使用 LPUSER_INFO_0 返回结果
            FILTER_NORMAL_ACCOUNT, // 只列举“正常”类型的用户
            (LPBYTE*)&pBuf,// LPUSER_INFO_0 保存返回结果
            // MAX_PREFERRED_LENGTH，内存由 API 分配，需要在之后调用 NetApiBufferFree 释放
```

```
            dwPrefMaxLen,
            &dwEntriesRead,// 读了的 Entries
            &dwTotalEntries,// 一共的 Entries
            &dwResumeHandle);
        // 判断是否成功
        if ((nStatus == NERR_Success) || (nStatus == ERROR_MORE_DATA))
        {
            if ((pTmpBuf = pBuf) != NULL)
            {
                // 循环读取用户信息
                for (i = 0; (i < dwEntriesRead); i++)
                {
                    assert(pTmpBuf != NULL);
                    if (pTmpBuf == NULL)
                    {
                        fprintf(stderr, "An access violation has occurred\n");
                        break;
                    }
                    // 输出
                    wprintf(L"\t-- %s\n", pTmpBuf->usri0_name);
                    // 下一个
                    pTmpBuf++;
                    dwTotalCount++;
                }
            }
        }
        else
            fprintf(stderr, "A system error has occurred: %d\n", nStatus);
        // 释放内存
        if (pBuf != NULL)
        {
            NetApiBufferFree(pBuf);
            pBuf = NULL;
        }
    }
    while (nStatus == ERROR_MORE_DATA); // end do
    // 释放内存
    if (pBuf != NULL)
        NetApiBufferFree(pBuf);
    fprintf(stderr, "Total of %d users\n\n", dwTotalCount);
    return 0;
}

/*************************************
* int ListGroup(LPWSTR pszServerName)
* 功能: 列举用户组
* 参数: szServerName, 主机名, 如果为本机, 设置为 NULL
*************************************/
int ListGroup(LPWSTR pszServerName)
{

    DWORD dwLevel = 0;
    DWORD dwPrefMaxLen = MAX_PREFERRED_LENGTH;
    DWORD dwEntriesRead = 0;
    DWORD dwTotalEntries = 0;
    DWORD dwResumeHandle = 0;
    DWORD i;
    DWORD dwTotalCount = 0;
    NET_API_STATUS nStatus;
    LPLOCALGROUP_INFO_0 pBuf = NULL;
```

```
    LPLOCALGROUP_INFO_0 pTmpBuf;
    do // 开始
    {
        // 调用 NetLocalGroupEnum，参数设置与 NetLocalGroup 类似
        nStatus = NetLocalGroupEnum(
            pszServerName,
            0,
            (LPBYTE*)&pBuf,
            dwPrefMaxLen,
            &dwEntriesRead,
            &dwTotalEntries,
            &dwResumeHandle);
        // 判断结果
        if ((nStatus == NERR_Success) || (nStatus == ERROR_MORE_DATA))
        {
            if ((pTmpBuf = pBuf) != NULL)
            {
                // 循环输出
                for (i = 0; (i < dwEntriesRead); i++)
                {
                    assert(pTmpBuf != NULL);
                    if (pTmpBuf == NULL)
                    {
                        fprintf(stderr, "An access violation has occurred\n");
                        break;
                    }
                    wprintf(L"\t-- %s\n", pTmpBuf->lgrpi0_name);
                    pTmpBuf++;
                    dwTotalCount++;
                }
            }
        }
        else
            fprintf(stderr, "A system error has occurred: %d\n", nStatus);
        // 释放内存
        if (pBuf != NULL)
        {
            NetApiBufferFree(pBuf);
            pBuf = NULL;
        }
    }
    while (nStatus == ERROR_MORE_DATA); // end do
    if (pBuf != NULL)
        NetApiBufferFree(pBuf);
    fprintf(stderr, "Total of %d groups\n\n", dwTotalCount);
    return 0;
}
```

2．实例 17-9 显示用户信息

NetUserGetInfo 函数可以获取用户信息，演示了 NetUserGetInfo API 函数的使用，代码如下：

```
/*************************************
* ShowUsersInfo
* 功能：显示指定用户的信息
* 参数：szServerName，主机名，如果为本机，设置为 NULL
*       pszUserName，用户名
*************************************/
int ShowUsersInfo(LPWSTR pszServerName,LPWSTR pszUserName)
{
    DWORD dwLevel = 4;// 使用 LPUSER_INFO_4 返回结果
```

```
    LPUSER_INFO_4 pBuf = NULL;
    NET_API_STATUS nStatus;
    nStatus = NetUserGetInfo(pszServerName,
        pszUserName,
        dwLevel,     // pBuf参数类型
        (LPBYTE *)&pBuf);
    // 判断并输出结果
    if (nStatus == NERR_Success)
    {
        if (pBuf != NULL)
        {
            wprintf(L"\n\tAccount:     %s\n", pBuf->usri4_name);
            wprintf(L"\tComment:     %s\n", pBuf->usri4_comment);
            wprintf(L"\tUser comment: %s\n", pBuf->usri4_usr_comment);
            wprintf(L"\tFull name:   %s\n", pBuf->usri4_full_name);
            wprintf(L"\tpriv:    %d\n", pBuf->usri4_priv);
        }
    }
    else
        fprintf(stderr, "A system error has occurred: %d\n", nStatus);
    // 释放内存
    if (pBuf != NULL)
        NetApiBufferFree(pBuf);
    return 0;
}
```

3．运行结果

17.3 节所有实例都是一个程序中的代码，即本书所附光盘中的 users.c。代码的主函数如下所示，分别调用本节实例中所给出的各个函数，实现相关功能。

```
/*************************************
* wmain
* 功能：入口函数，根据参数判断需要调用的功能函数
* 参数：参见usage输出
*************************************/
int __cdecl wmain(int ac, wchar_t * av[])
{
    if (ac == 4 && lstrcmpW( av[1], L"-a") == 0)
    {
        AddUser(NULL, av[2], av[3]);
    }
    else if (ac == 4 && lstrcmpW( av[1], L"-g") == 0)
    {
        AddUserToGroup(NULL, av[2], av[3]);
    }
    else     if (ac == 3 && lstrcmpW( av[1], L"-i") == 0)
    {
        ShowUsersInfo(NULL, av[2]);
    }
    else     if (ac == 2 && lstrcmpW( av[1], L"-i") == 0)
    {
        ListUsers(NULL);
        ListGroup(NULL);
    }
    else    if (ac == 3 && lstrcmpW( av[1], L"-d") == 0)
    {
        DelUser(NULL, av[2]);
    }
    else
```

```
    {
        printf("usage: \n"
            "\t %ws -a <username> <password> to add a user\n"
            "\t %ws -g <username> <group> add a user to a group"
            "\t %ws -i <username> to show user info\n"
            "\t %ws -d <username> to del a user\n",
            av[0], av[0], av[0], av[0]);
    }
    return 0;
}
```

运行程序，可以进行增加、删除等操作，增加、删除后可以列举用户验证结果。

运行 Users.exe-i，列举出当前主机的用户和用户组，如图 17-6 所示。

图 17-6　列举用户和用户组

使用-d 参数可以删除用户，删除后，使用一 i 参数可以查看已经删除成功，如图 17-8 所示。

使用一 a 参数用户，可以增加用户；使用一 i 参数可以查看用户增加后的用户信息，如图 17-7 所示。

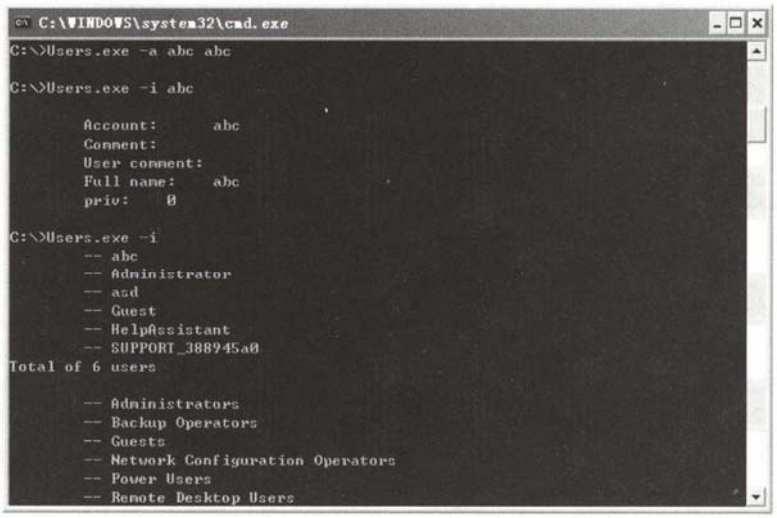

图 17-7　增加用户

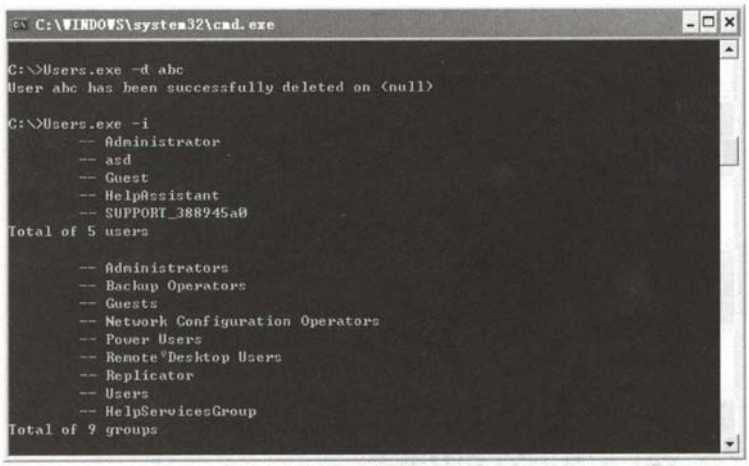

图 17-8　删除用户

如图 17-9 所示，在“计算机管理一本地用户和组”中也可以看到相应变化。

图 17-9　通过计算机管理查看用户信息

第 18 章　Windows API的内部原理

作为本书的最后一章，本章将介绍一些较为深层次的内容。这些内容会对有一定的Win32 平台开发经验且想继续深入学习的读者有一定的帮助。

学习本章后，读者可以获得以下的经验。

◇在编写 Windows 应用程序时，如何获得权威的参考，包括文档和示例。

◇如何更有效地使用 Windows 应用程序的开发工具，主要是 SDK 工具。

◇尝试解决 Windows 应用程序开发中的一些疑难问题，原因不明或令人费解的问题，本章会说明 Windows 系统应用程序开发接口的实现原理和调用过程。

◇程序的版本兼容性问题。

◇初步了解 Windows 系统的实现原理。

◇从 Windows 系统的设计中学习程序开发的经验。

18.1　关于API的补充说明

本书前 17 章已经介绍了最常用的 Windows API，但是也只包含了 Windows API 中的很小一部分。Windows API 分类繁杂，数目众多，能实现的功能多种多样。在本书的篇幅中也不可能将所有 Windows API 都作全面的介绍。

Windows API 也会随着 Windows 版本的变化而变化，每一个 Windows 新版本发布都会有一些新的 API。

随着 64 位操作系统的应用逐渐广泛，纯 64 位应用程序的开发也逐渐流行。

18.1.1　Windows API的版本演进和Vista新增API

Windows API 有版本演化问题，Windows 每一个版本都会新增加一些 API。

因此在编译时，特别需要注意所使用的 API 对系统版本的要求，可能某些系统还不具备这些 API。

一些较新的应用程序，在较老的版本的系统上运行，可能会出现一些特定的 API 函数无法在指定的 DLL 中找到的情况，这就是由于所需的 API 仅出现在较新版本的系统中。

比如 ConvertFiberToThread 函数只出现在 Windows XP 及以上版本的操作系统中，而 Windows 2000 不支持此 API 函数。再如 RegSetKeyValue API 函数只能在 Vista 系统中使用。

如果应用程序开发人员希望所开发的程序在某些版本上正常运行，那么可以在编译时使用 ID 编译选项来为“_WIN32_WINNT”预处理声明指定相应的值来实现。比如，如果要求运行的操作系统为 Windows 2000 及以上，可以定义为__WIN32_WINNT=0x0500。

Vista 是最新的桌面系统版本，其版本号为 NT6，Vista 也新增加了很多 API。下面列出了部分 Vista 版本的 Windows 系统新增加的 API 函数:ControlServiceEx、CreateDesktopEx、CreateEventEx 、 CreateMutexEx 、 CreateSemaphoreEx 、 CreateSymbolicLink 、CreateWaitableTimerEx、Delete Proc ThreadAttributeList、EnumProcessModulesEx、FindFirstFileNameW 、 FindNextFileNameW 、 GetConsoleHistoryInfo 、GetCurrentConsoleFontEx、GetDynamicTimeZoneInformation、GetErrorMode、GetFile InformationByHandleEx、GetFinalPathNameByHandle、GetNamedPipeClientComputerName、Get NamedPipe ClientProcessId 、 GetNamedPipeClientSessionId 、

GetNamedPipeServerProcessId、GetNamed PipeServer
SessionId、GetNumaProximityNode、GetProductInfo、IsThreadAFiber、OpenFileById、Reg Copy Tree、RegDeleteTree。

18.1.2 64 位操作系统的接口

如果希望开发 64 位应用程序，需要在安装 SDK 时，选择安装 64 位(IA-64 或 x64)编译器和库文件。

在编译连接时，选择使用专用 64 位编译链接工具，在链接时，使用 64 位的导入库。

64 位操作系统与 32 位操作系统最大的不同在于寻址空间的大小，32 位操作系统使用 32 位数据来表示内存地址，因此有 232 (4GB)的可用空间，而 64 位操作系统可以使用 16TB 的内存地址空间。在 64 位操作系统下，指针和基本数据类型(int)所引用的空间是 64 位的。

为了 64 位操作系统与 32 位操作系统之间的兼容性，SDK 定义了若干新的数据类型，这些数据类型明确了数据是使用 32 位存储，还是 64 位存储。

DWORD32、DWORD64;

INT32、INT64;

LONG32、LONG64;

UINT32、UINT64;

ULONG32、ULONG64。

指针也有多种数据类型：

POINTER_32 和 POINTER 一 64 等。

除了数据长度上的区别外，在接口函数上，32 位操作系统与 64 位操作系统兼容，64 位操作系统上仍然可以调用之前介绍的 Windows API，只是在数据类型上需要做一些调整。

当然，64 位操作系统也是支持 32 位操作系统的应用程序的运行。为此 64 位的 Windows 版本专门设计了一个子系统 WOW64，用于支持在 64 位操作系统上运行 32 位应用程序。

32 位操作系统不支持 64 位应用程序的运行。

18.2 Windows系统中的对象封装

面向对象是一种程序设计思想，而不是指具体的程序设计语言类型。Windows 系统虽然是使用 C 语言程序开发的，Windows 应用程序 API 也大多为 C 形式，但是 Windows 系统本身却是很好的面向对象思想的体现。

18.2.1 什么是对象

对象是程序的操作对象，程序是对对象进行处理。如果将对象和对象的处理方法（函数）集合在一起，就是“类”的基本模型。

18.2.2 面向对象的思想

Windows 系统虽然没有把操作对象和对对象进行操作的函数集合在一起，但是把每一种需要处理的数据内容全部“对象”化。每一种对象都有句柄作为其标识。这样便于对对象的

操作函数接口化。

18.2.3 Windows系统中的对象：内核对象、GDI对象等

Windows 系统中包括的对象类型有很多种。内核对象、GDI 图形对象、用户对象等。

内核对象是与操作系统对系统的管理相关的对象，比如文件、进程、事件，网络通信的 Socket 等。

而 GDI 对象是指在图形输出的相关对象。

USER 对象是指用户控件、菜单、输入设备等。

程序进行相关的操作时，都以这些对象作为操作对象，通常以句柄作为参数。

18.3 WiFidows程序设计参考：文档资源与样例代码

Windows 程序的开发，可供参考的资料非常丰富，SDK 文档包括了所有 API 的使用访求介绍，MSDN 则更为全面。

18.3.1 SDK文档和MSDN

在完成好的各个版本的 SDK 中都有帮助文档。SDK 文档包括非常完整和详细的 Windows 应用程序开发帮助。每个 API 函数都有详细说明，每一方面都提供若干参考示例。

对桌面应用程序开发最有帮助的是“Win32 and Com Development”条目，此条目中分门别类地列举出了各个方面的程序设计参考。每个方面又分为若干个小的方面，每个小方面又会详细介绍原理（通常在 About 条目中），结合若干示例（通常在 Using 条目中），给出 API 函数的使用方法（通常在 References 条目中），SDK 文档的界面如图 18-1 所示。

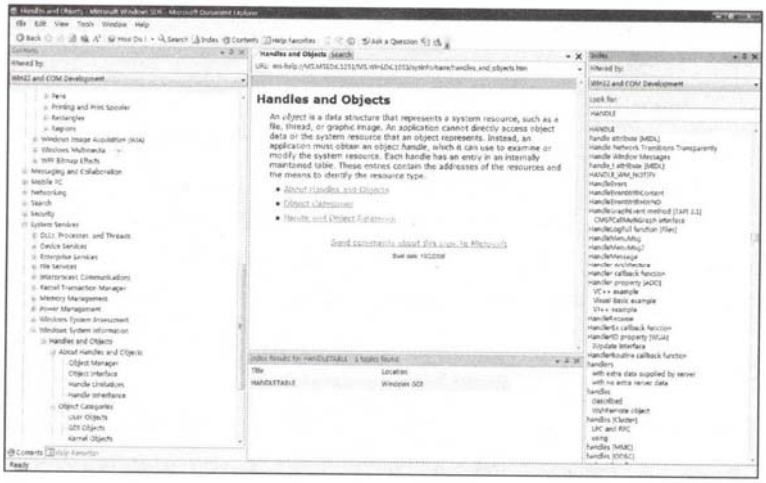

图 18-1 SDK 文档

MSDN 也包含完整的 SDK 文档。

18.3.2 SDK示例代码

SDK 的示例是编写 Windows 应用程序时非常好的参考。SDK 为几乎所有的典型应用提供

了示例。示例代码在 SDK 安装目录下的 Samples 目录中。

18.4 x86 平台程序函数调用原理

本节通过对函数调用的内部过程的分析，引导读者解决在程序开发和调试过程中的疑难问题。本节会给出一个详细的调试器使用步骤，希望读者能从中受益。

18.4.1 函数调用的真实过程

了解函数调用的内部过程需要读者了解一些基本的汇编指令。本节实例会通过反汇编来查看函数的调用过程。

实例 18-1 函数调用的内部过程

调用内部过程示例源代码：

```
//调用示例
#include <windows.h>
DWORD Function(DWORD p1, PVOID p2)
{
    DWORD v1;
    DWORD v2 = 3;
    v1 = p1 + v2;
    return v1;
}

int _cdecl main()
{
    Function(100, NULL);
    return 0;
}
```

使用如下编译选项进行编译：

```
/Od /FD /MT /W3 /nologo /c /ZI /Gz /TC
```

链接选项：

```
/OUT:" fun.exe" /NOLOGO /DEBUG /PDB:"fun.pdb" /MACHINE:X86 fun.obj kernel32.lib user32.lib
shell32.lib
```

在 Vistual Studio 中编译生成后，设置断点，运行程序。断点命中后，在代码中单击鼠标右键，查看反汇编代码，可以显示出高级语言代码与汇编语言的对应关系。

```
//调用示例
#include <windows.h>

DWORD Function(DWORD p1, PVOID p2)
{
00412200  push        ebp
00412201  mov         ebp,esp
00412203  sub         esp,48h
00412206  push        ebx
```

```
00412207  push        esi
00412208  push        edi
    DWORD v1;
    DWORD v2 = 3;
00412209  mov         dword ptr [ebp-8],3
    v1 = p1 + v2;
00412210  mov         eax,dword ptr [ebp+8]
00412213  add         eax,dword ptr [ebp-8]
00412216  mov         dword ptr [ebp-4],eax
    return v1;
00412219  mov         eax,dword ptr [ebp-4]
}
0041221C  pop         edi
0041221D  pop         esi
0041221E  pop         ebx
0041221F  mov         esp,ebp
00412221  pop         ebp
00412222  ret         8
--------------------------------------------------------

int _cdecl main()
{
00412230  push        ebp
00412231  mov         ebp,esp
00412233  sub         esp,40h
00412236  push        ebx
00412237  push        esi
00412238  push        edi
    Function(100, NULL);
00412239  push        0
0041223B  push        64h
0041223D  call        004110AF
    return 0;
00412242  xor         eax,eax
}
00412244  pop         edi
00412245  pop         esi
00412246  pop         ebx
00412247  mov         esp,ebp
00412249  pop         ebp
0041224A  ret
```

下面分析函数调用的过程。

main 函数调用 Function 函数的过程如下：

```
00412239  push        0
0041223B  push        64h
0041223D  call        004110AF
```

在运行这段代码之前，CPU 寄存器的值如图 18-2 所示。

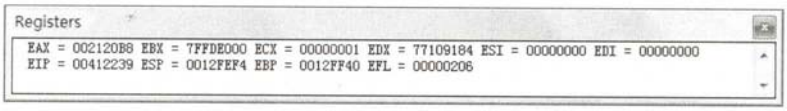

图 18-2　函数调用过程分析（1）

先运行了两个 push 指令，将 NULL (0)和 100 (0x64)压入栈中。push 指令会将 CPU 中 esp 指向的地址（栈顶）写入数据，然后 esp 的值减 4（栈是由高地址向低地址方向生长，一个整形数据占用 4 个字节，x86 CPU 中最小的存取单位是字节）。

单步运行到 0041223D 的 call 指令处，注意图 18-3 中的 esp 寄存器的变化。

> **注意**　同样的代码，每次重新编译程序，生成的代码的指令可能不一样。这是由于编译链接选项的不同或编译链接工具版本的不同所造成的。

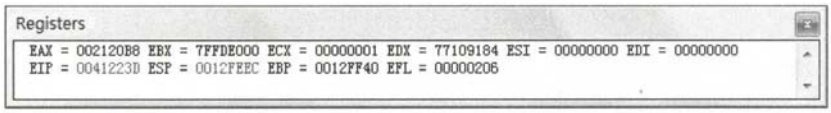

图 18-3　函数调用过程分析（2）

查看内容中栈的情况(图 18-4)，入栈的两个参数依次在栈顶的位置。

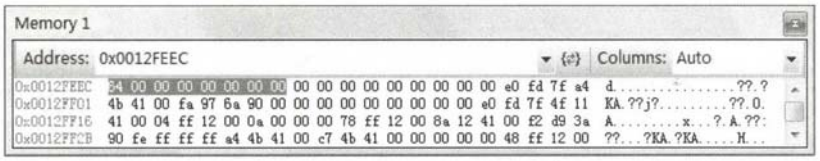

图 18-4　函数调用过程分析（3）

然后程序运行 call 指令，跳转到了 004110AF，004110AF 处是一个跳转指令，跳转到函数 Function 的程序段在内存中的起始地址。

调用 call 指令后，esp 的值又减少了 4，说明 call 指令会向栈中压入数据。查看栈顶的情况，call 指令压入的数据为“0x00412242”，可以发现这个值是 maln 函数中调用 Function 函数的 call 指令的下一条指令的地址。call 指令保存了函数调用的返回地址在栈中，寄存器的值和内存中的内容如图 18-5 和图 18-6 所示。

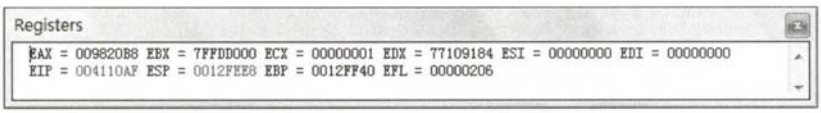

图 18-5　函数调用过程分析（4）

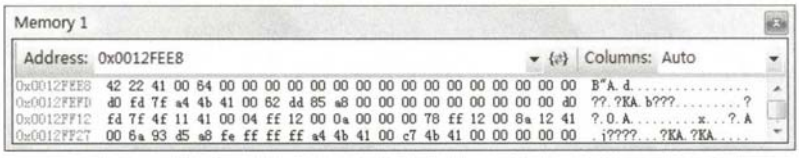

图 18-6　函数调用过程分析（5）

注意　很多情况下，通过跳转表调用函数。调用函数时，会使用 call 指令调用一个跳转表中的位置，这个位置上通常是一个 jmp 强制跳转指令，跳转到真实的函数程序段处。当然一般在 release 版本的可执行文件中，不会采用函数跳转表，而是直接调用函数代码所在的地址。

```
@ILT+135(_GetStartupInfoA@4):
0041108C  jmp        00426C18
@ILT+140(_abort):
00411091  jmp        00419E1F
@ILT+145(__ascii_strnicmp):
00411096  jmp        00425AE0
@ILT+150(_InterlockedIncrement@4):
0041109B  jmp        00426C3C
@ILT+155(_isleadbyte):
004110A0  jmp        0042429B
@ILT+160(__daylight):
004110A5  jmp        00423675
@ILT+165(__heap_init):
004110AA  jmp        00413FA2
@ILT+170(_Function@8):
```

```
004110AF  jmp        00412200
@ILT+175(__crtMessageBoxA):
004110B4  jmp        0041584C
@ILT+180(__malloc_crt):
```

之后跳转到 00412200，是 Function 函数的代码。

```
DWORD Function(DWORD p1, PVOID p2)
{
00412200  push        ebp
00412201  mov         ebp,esp
00412203  sub         esp,44h
00412206  push        ebx
00412207  push        esi
00412208  push        edi
```

进入 Function 函数后，先将 ebp 寄存器保存到栈，然后将当前 esp 栈顶的地址赋值给 ebp，这样 ebp 就指向之前入栈的上一个 ebp 的值。然后将 esp 的值减少，值栈顶向上移，从下面的代码对局部变量的访问来分析，将 esp 的值减少的步骤实际是为局部变量分配内存空间（由此可以得出为什么局部变量的内存分配速度很快），如图 18-7 所示.

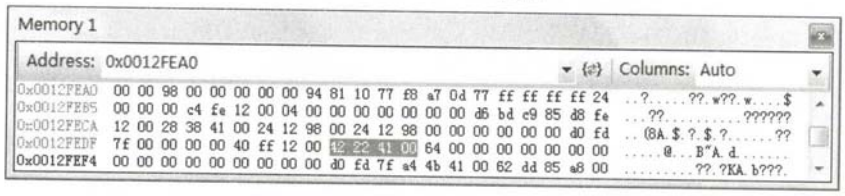

图 18-7　函数调用过程分析（6）

运行完成后，保存在栈中的局部变量如图 18-8 所示。

```
    DWORD v1;
    DWORD v2 = 3;
00412209  mov         dword ptr [ebp-8],3
    v1 = p1 + v2;
00412210  mov         eax,dword ptr [ebp+8]
00412213  add         eax,dword ptr [ebp-8]
00412216  mov         dword ptr [ebp-4],eax
    return v1;
00412219  mov         eax,dword ptr [ebp-4]
}
0041221C  pop         edi
0041221D  pop         esi
0041221E  pop         ebx
0041221F  mov         esp,ebp
00412221  pop         ebp
00412222  ret         8
```

之后开始函数的功能实现，由于之前 ebp 值的变化，在这个时候，ebp-8 的位置保存的是局部变量 v2，而 ebp+8 的位置是参数 p1（在 mam 函数中压入栈，正好位于 call 指令入栈的函数调用返回地址之前），ebp-4 的位置保存的是局部变量 v1。返回值保存于 eax 中（一般编译器都约定使用 eax 保存返回值）。

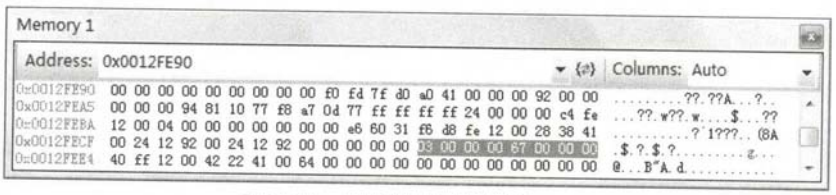

图 18-8　函数调用过程分析（7）

Function 函数中的最后 3 条指令是恢复调用函数(main)的栈寄存器值，为返回(ret)做准备。ret 指令自动将 eip 寄存器自动指向 call 指令的下一指令，ret 指令通过 ebp 寄存器找到了 call 指令入栈的函数返回地址，call 指令的入栈和 ebp 指令的变化都为函数的返回做好了准备工作。

开始继续执行 main 函数。寄存器和内存的值如图 18-9 和图 18-10 所示。

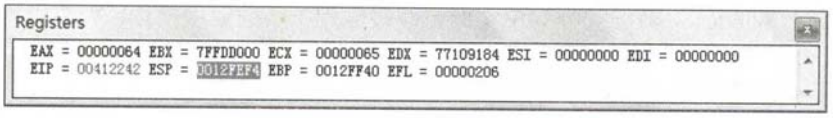

图 18-9　函数调用过程分析（8）

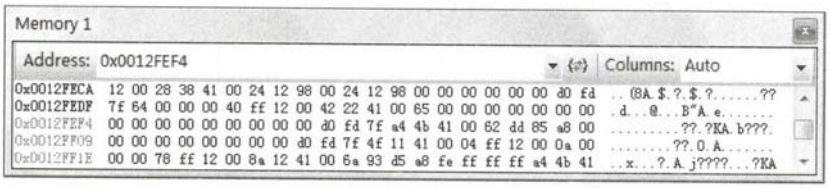

图 18-10　函数调用过程分析（9）

在函数调用过程中栈的结构如图 18-11 所示。

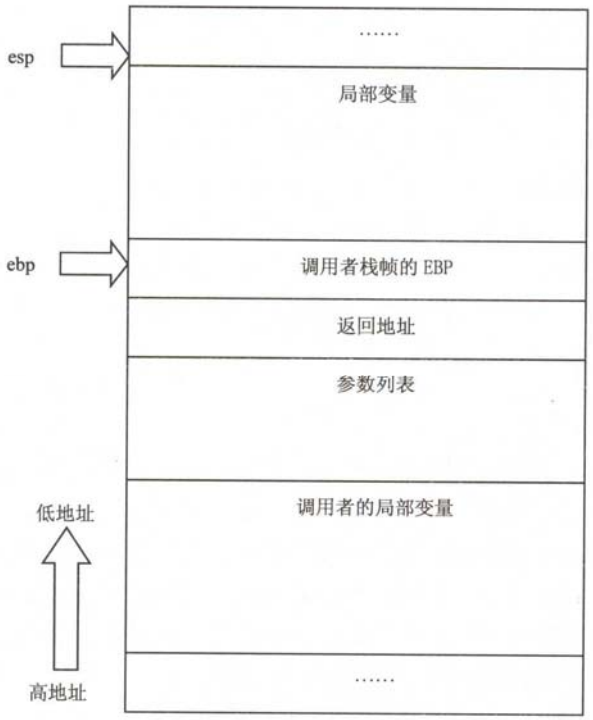

图 18-11　堆栈的结构

函数每调用一次，ebp 变化一次。栈中依次保存着参数、返回地址、调用者的 ebp、局部变量等。

每一次函数调用都会产生一个这样的结构，随着调用的深入在栈中依次增长因此又称为栈帧，每一帧对应着一个函数调用。ebp 在栈帧中起着关键作用，参数的寻址、局部变量的寻址和函数返回都依赖于 ebp。

18.4.2　函数调用约定

main 函数前有“_cdecl”标识，说明函数的调用约定，调用方式影响函数调用时参数的传递方式和函数返回后栈的恢复访求。如果为“_cdecl”和“_stdcall”方式则参数均为从右往左入栈，而“_fastcall”使用 ECX 和 EDX 寄存器传递，余下部分的参数从右向左保存在栈中。

在函数返回后，如果函数是“_cdecl”调用方式，则是调用者将 esp 和 ebp 指令恢复到调用前的　态；而如果是“_stdcall”方式，则由被调用函数将栈恢复到调用前，以便调用

函数可以继续执行。

在 2.1 节“Windows 数据类型”中实例已经接触到了函数调用约定，在表 2.1 中，有一个 1“WINAPI”，“WINAPI”实际就是调用约定，如下所示：

```
#define WINAPI      __stdcall
```

很多的 API 声明中都有“WINAPI”标识，一些回调函数有“CALLBACK”标识。这些标识实际是在头文件中定义好的宏，经过预处理后，会展开为“_stdcall”等关键字，这说明了 API 楚数的调用方式。

如果在代码中没有指定函数的调用方式，那么编译器会根据编译选项(/Gz /Gd /Gr)来决定参数的处理方式。一般的 Windows API 都是使用 stdcall 调用方式，而 maln 函数使用 cdecl 调用约定。

18.4.3 为什么通过参数返回数据时只能使用指针

Windows 中有很多 API 会通过参数返回值，比如 ReadFile 函数：

```
BOOL ReadFile(
  HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

参数 lpNumberOfBytesRead 用于返回实际读取的数据的长度。

通过上面的参数传递过程分析，可以明白为什么函数调用者不能通过函数对参数进行直接赋值。在调用函数时，参数经过了一个入栈过程，与调用函数使用的参数变量已经不是在同一个地址上了，如果在被调用函数中对参数赋值，实际没有影响到调用函数的变量。参数在函数返回后的栈恢复操作中被丢弃了。

18.4.4 缓冲区溢出

上面的过程也可以解释缓冲区溢出的原理。缓冲区溢出是重要的安全性问题。

如果局部变量中保存了数组、字符串等，那么使用 memcpy、strcpy 等函数将数据复制到缓冲区（从低内存地址到高内存地址），如果复制的长度超过了缓冲区的长度，可以覆盖位于缓冲区之后的函数返回地址（栈是从高方向向低方向生长的），在函数返回时，就不会返回正确的地址，而返回通过内存复制的数据中的某个数据所指向的地址，如果精心构造这个值，就可以通过外部输入的数据控制程序的执行流程，从而造成安全性问题。

18.4.5 程序运行错误的调试技巧

在非调试状态的程序崩溃后，可以使用调试器附加到进程。查看 EIP 寄存器的值，查看调用堆栈，分析、定位出错的代码。然后查看出错的指令是否为访问无效地址，是否运行到非有效代码中等情况。

如果出错的位置是在堆中，查看在之前使用堆时有无溢出等情况。如果出错的位置在栈中，查看是否有缓冲区溢出等情况。

实际情况可能比较复杂，需根据具体情况，分析出错的原因，定位错误后根据具体情况

解决。

18.5 可执行程序结构与AIPI函数接口内部机理

18.5.1 Windows可执行程序结构

Windows 系统的可执行程序结构称作 PE, EXE 文件、DLL 文件、SYS 文件、OCX 文件等都是 PE 格式的文件。

PE 文件中，依次放入以下内容。

1．MZ 头

MZ 头是 PE 文件最开始的一个结构。之后是一个 DOS stub，这两个部分实际是一个标准的 DOS 下的可执行程序。如果一个 Windows 可执行程序在 DOS 下运行，会显示出此程序不能在 DOS 中运行的信息。

MZ 头的其他信息不需要关心，只需要知道前两个字节恒为“MZ”，第 0x3c 个字节，指向 PE 头的起始位置。

2．PE 头

MZ 头中偏移为 0x3c 的位置指明了文件中 PE 头的开始位置，PE 头是 PE 格式文件中非常重要的结构。

结构如下：

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD Signature;
  IMAGE_FILE_HEADER FileHeader;
  IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

Signature 是一个标识，值恒为“PE”。标识后，依次放置两个子结构：文件头和可选头。文件头结构如下：

```
typedef struct _IMAGE_FILE_HEADER {
  WORD Machine;
  WORD NumberOfSections;
  DWORD TimeDateStamp;
  DWORD PointerToSymbolTable;
  DWORD NumberOfSymbols;
  WORD SizeOfOptionalHeader;
  WORD Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

文件头中说明了运行本可执行程序所需的 CPU 类型、文件中的块数据、文件创建时间、调试符号表的位置和符号的数目（用于调试）、可选头的大小和文件标识信息(exe、dll)。

之后是可选头，可选头的结构比较复杂，而且包括一个比较大的 IMAGE DATA—DIRECTORY 数组。可选头中包括代码的入口地址、数据块的大小、代码段的起始和终止、模块加载的基址、块对齐值、系统版本等。而 IMAGE_ DATA—DIRECTORY 数组中包括若干个在模块加载时需初始化的表及表的虚拟地址（包括导入表导出表）等。

可选头 IMAGE_OPTIONAL_ HEADER 结构如下：

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  WORD Magic;
  BYTE MajorLinkerVersion;
  BYTE MinorLinkerVersion;
```

```
    DWORD SizeOfCode;
    DWORD SizeOfInitializedData;
    DWORD SizeOfUninitializedData;
    DWORD AddressOfEntryPoint;
    DWORD BaseOfCode;
    DWORD BaseOfData;
    DWORD ImageBase;
    DWORD SectionAlignment;
    DWORD FileAlignment;
    WORD MajorOperatingSystemVersion;
    WORD MinorOperatingSystemVersion;
    WORD MajorImageVersion;
    WORD MinorImageVersion;
    WORD MajorSubsystemVersion;
    WORD MinorSubsystemVersion;
    DWORD Win32VersionValue;
    DWORD SizeOfImage;
    DWORD SizeOfHeaders;
    DWORD CheckSum;
    WORD Subsystem;
    WORD DllCharacteristics;
    DWORD SizeOfStackReserve;
    DWORD SizeOfStackCommit;
    DWORD SizeOfHeapReserve;
    DWORD SizeOfHeapCommit;
    DWORD LoaderFlags;
    DWORD NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

可选头之后是节表，PE 文件中包括程序运行所需的代码、数据、资源、导入表、导出表等都是保存在节中的，每一个节都在节表中对应一个节表头，节表头中表示了节的开始位置、模块初始化时需加载到的虚拟空间位置、数据节的大小和属性等。

节表头的结构如下：

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

图 18-12 所示为一个 PE 文件的文件头，黑框标识出了 m 头、PE 标识和 PE 文件头。Windows 系统在创建进程时，首先将可执行文件的文件头读取到内存中，判断节表头中各项成员是否符合要求；然后读取文件中的各个表的相应字段，并在头文件的标识下将文件中的各个节初始化到虚拟内存中的相应位置，加载所需调用的模块；然后再根据 PE 文件中指定的入口地址创建进程的主线程，进程就开始运行。

3 节表

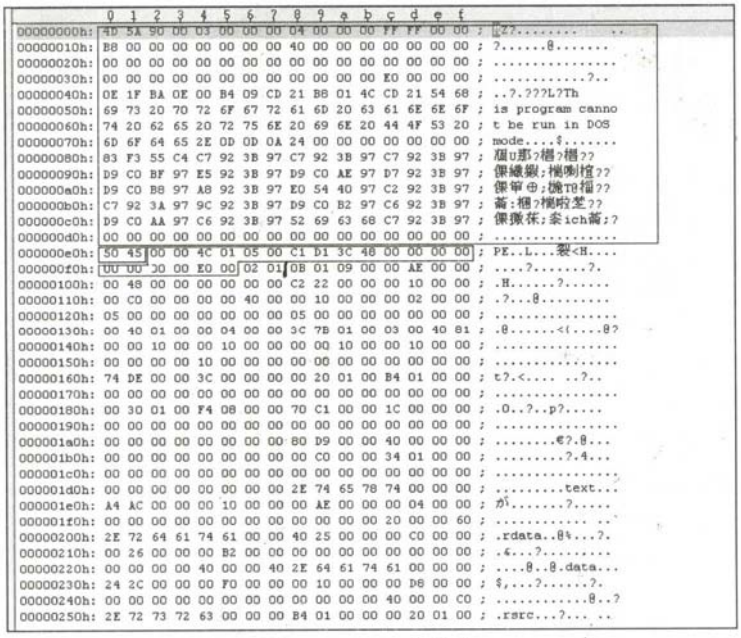

图 18-12　PE 文件头

DLL 文件也是 PE 文件，加载模块时也有类似的过程，在加载模块时还需根据导入表和导出表进行动态链接。

18.5.2　导入表、导出表、动态链接

一个应用程序，除了有主模块外(EXE)还会具有很多的其他模块（DLL 等），这些模块可能是开发人员出于代码复用等目的自己编写的，也可能是系统 API 所在的 DLL，或其他第三方程序库。程序在运行过程中，会不断地调用其他模块中的函数。

在进行调用函数时，需要知道函数的地址，调用指令(call)以函数地址或保存函数地址的指针作为参数。在同一模块中的函数，代码的相对位置都是固定的，call 语句的参数在编译时都已经计算好，并嵌入到指令中（称作静态链接）。但是位于不同模块的函数要相互调用时，被调用函数的地址并不确定。

系统使用动态链接机制来保证不同模块中函数的正确调用。导入表和导出表是两个在动态链接中需要使用到的重要结构。

导入表中保存了一个模块所需调用的外部模块的函数，包括函数所在的 DLL 函数名等。导出表保存了一个模块可被其他模块调用的函数，包括函数名。图 18-13 所示是一个 PE 文中的导入表部分信息。

在进程创建、模块加载时，系统会从可执行文件的导入表中获取信息，知道需要加载哪些 DLL，然后再加载这些需要调用的 DLL，从加载的被调用 DLL 中根据函数名找到需要调用的函数地址，然后将导入表指定的各函数的地址指针数组的值全部赋值为真实的 DLL 导入函数地址。

因此一个模块的导入表的位置是相对固定的，但是导入表中的函数指针的值是在动态链接时赋值的。在完成动态连接后，程序就可以通过导入表中的函数指针调用外部函数了。

模块中的代码需要调用外部函数时，call 指令的寻址方式是间接寻址，依赖于导入表中的函数指针，指针位置是固定的，但是指针指向的内容会在导入表经过初始化后才填入。在初始化时，所需调用的模块加载后，系统再计算出各个 API 函数在内存中的位置，然后再将导入表中内容填入真实的 API 地址。

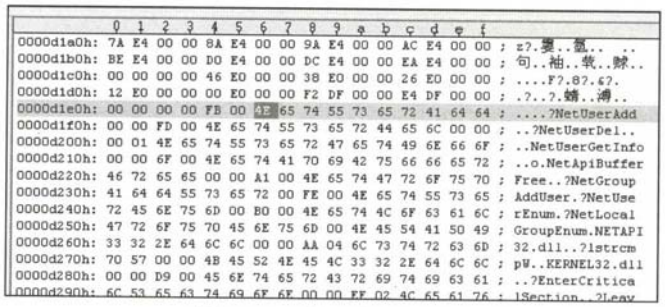

图 18-13　导入表部分信息

如图 18-14 所示，程序通过导入表调用 GetWindowsDirectory 函数时，call 指令是内存间接寻址，通过 0x0042EIB4 处（固定，位于导入表中）保存的指针找到真实的函数地址。而这些地址都是在动态链接时由系统填入的。0x77236214 是 GetWindowsDirectory 函数的真实地址，这个地址在 GetWindowsDirectory 所在 DLL 加载后确定。

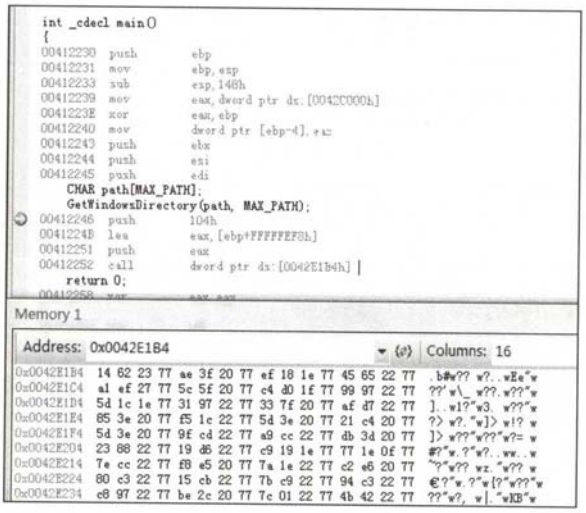

图 18-14　通往导入表调用外部函数

从图 18-14 可以看出，导入表中的函数指针众多，每一个指针中保存了一个函数地址。在外部调用不同的函数时会用到。

18.5.3　NTDLL.DLL、NATIVE API和SSDT

很多 API 函数，其最终的功能是在内核中实现的。比如与操作系统管理（例如进程创建）和 I/O 相关的函数（例如读写文件）都会调用系统内核中的代码。

1．x86 平台的优先级

x86 平台的 CPU 有 4 个优先级，分别称作 ring0、ring1、ring2 和 ring3。ring0 是最高优先级，低优先级的代码不能读写高优先级的内存空间，一些特殊的操作只能在高优先级的情况下才能调用。这种优先级实现了很好的内核保护机制。程序在进行操作系统内核调用时，程序会从低优先级运行到高优先级，内核调用返回时会从高优先级变为低优先级。

Windows 系统只采用了 4 个优先级中的 ring0（内核态）和 ring3（用户态）。SYSENTER 和 LEAVE 两个 CPU 指令可以在用户态和和内核态之间切换，有少量的参数可以通过寄存器传递。

2．　Ntdll.dll

在 Windows 系统中完成从用户态到内核态调用的是 NTDLL.DLL。所有的 API

(KERNEL32.DLL、USER32.DLL、GDI32.DLL 等)最终都会调用 NTDLL.DLL，NTDLLDLL 所导入的 API 函数称作本机 API (NativeAPI)。

3．SSDT

内核中处理系统调用的结构称作 SSDT，SSDT 的全称是 System Service Descriptor Table，也就是系统服务描述符表。

4．系统调用的完整过程

在 Windows 操作系统中，应用程序的实现依赖于对 Win32 API 的调用。操作系统以动态链接库的形式提供这些 Win32 API。但 Win32 API 只是系统为开发人员在系统的用户层提供的一种调用系统服务的接口。而在 Win32 API 和系统服务之间还有一层隐藏接口，即 Native API。

系统所提供的动态链接库及内核中系统服务之间的调用关系如图 18-15 所示。

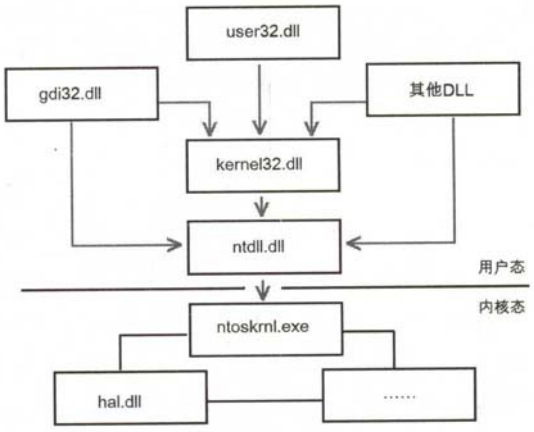

图 18-15　简略的系统 DLL 调用关系

上图中 user32.dll、gdi32.dll 以及 kerne132.dll 等实现了基本的 Win32 API。有的 API 函数的功能在用户态就可能实现，不用调用内核；而有的 Win32 API 需要调用系统内核，这部分 API 的调用最后都转移到了 ntdll. dll,而 ntdll.dll 又将其转移到了 ntoskrnl.exe。

由 ntdll.dll 导出的函数接口就是 NativeAPI，由 ntdll.dll 导出的 Native API 的实质是一系列的 stub 函数，实现了从用户态向内核态的跳转，调用系统服务实现 API 的最终功能。

当从 Native API 进入内核态以后，系统会根据服务号在 SSDT 中查找对应的表项，寻找所调用系统服务函数的真正地址。之后，系统会根据这个地址调用相应的系统服务函数，并返回函数的执行结果。

图 18-16 所示是使用 WinDbg 调试单步跟踪系统调用的过程。

图 18-16 WinDbg 调试系统跟踪系统调用过程

每一个 Native API 实现的功能都类似，都是在 eax 中填入了一个整数（相当于是一个调用号，各个 Native API 的调用号不同，上图中演示的 NtCrateFile 是 0x25），然后调用了 KiFastSystemCall KiFastSystemCall 函数的唯一功能是保存栈顶后，运行 sysenter 指令，进入内核。

SSDT 的全称是系统服务分配表，运行 sysenter 指令后，系统会根据 Native API 和设置的调用号，跳转到 SSDT 中的不同项。

具体过程是首先通过 SYSTEM_SERVICE_TABLE 结构中的指针进入内部结构 KiServiceTable，以服务号为索引查找可处理此 API 调用的函数的入口地址。然后以相同的方式查询 ArgumentTable 所指向的 KiArgumentTable 结构，以找出调用者在参数堆栈中传入了多少字节，然后使用这个值将参数复制到当前内核堆栈中。最后通过 CALL 指令来执行 API 处理例程。

18.5.4 API HOOK

HOOK（挂勾）是一种比较高级的应用。HOOK 技术是指修改 API 的地址或修改 API 中的代码，使得程序在调用 API 函数时，可以监视到调用，或者可以修改参数或返回值等。

HOOK 技术不是一种常规技术，可以实现非常规的功能，比如实现 RootKit、进行应用程序行为分析等。

通常有两种方法可以实现 API HOOK，一是遍历进程中各个模块的导入表，如果导入表中调用了所需 HOOK 的目标函数，则将跳转地址修改到自定义的函数中；另一种是在目标函数的地址-后，在函数的代码处写入跳转指令，一旦 API 被调用，则会跳转到自定义的函数中。

因此进行 API HOOK，需要进行两个步骤的工作：一是编写用于 HOOK 的函数，使用这些函数去“替换”正常的 API 函数，HOOK 函数中，需要调用原 API 函数；二是 HOOK 程序，使用系统原有 API 的地址或代码。

总结

以上 Windows 应用程序开发的内容就基本介绍完成。一般在一个程序开发、调试、测试完成后，最后的阶段是发布程序。在一些情况下，还需要为第三方软件提供接口。

18.6 发布程序

18.6.1 合理选择编译链接选项

发布程序，主要就是设置第 3 章所介绍的各项编译链接选项。

调试版本的编译选项除了需加入调试信息外，其他选项尽量和发布版本保持一致，以方便调试和错误排除。

去除不必要的选项，不要使选项过于复杂，不便于维护。

在调试程序时，可以使用 IDE，但是发布程序时，建议自行编写 makefile，手工调用 nmake 进行编译和链接。使用 makefile 灵活性更高，也更便于版本控制，也便于软件工程管理。

18.6.2 构建到指定路径

使用 makefile 自行构建程序，只需将程序运行所需的文件构建到一个目录，与编译过程所产生的中间文件相互隔离，便于软件发布，尤其是在软件中模块多、文件结构复杂的情况下，使用程序的发布变得简单而易于控制。

18.7 模块化，向Windows API学习接口定义

Windows 应用程序之所以流行，除了 Windows 本身的用户友好特性等原因外，具有良好的应用程序开发接口也是重要原因之一。

Windows SDK 是很好的应用程序开发样本，向 Windows 学习应用程序接口设计很有必要。

在软件中，模块化是一个基本的软件组织原则。模块化的好处很多，比如：便于软件维护，软件版本升级过程中，只需要替换特定模块即可；便于代码复用，软件模块化后，一些基本的功能模块中只需存在一份，需要使用这些功能程序时只需调用模块的相应函数即可，减少程序体积；

便于软件开发，根据模块划分开发工作，不同的开发人员和开发团队之间，只需要知道相互的接口形式即可，不需要了解其内部的实现原理。

动态链接库是模块化的重要技术手段。

在应用程序设计时，尽量将基础、公共的代码放置在 DLL 中。如果开发的应用程序需要向其他应用程序提供支持，可以像 SDK 一样定义一系列头文件，在生成 DLL 文件时，同时生成 lib 库文件。

18.7.1 lib文件

lib 文件中包括链接时所需的信息，指明了链接器应该如何处理这些外部函数等（通常是将函数调用加入导入表中）。

如果为第三方应用程序提供程序设计接口，那么就需要有 lib 文件，lib 文件指明了在链按时，链接器如何处理这些外部变量、外部函数等。如果需要通过导入表调用 DLL 中的函数，就需要在链接时将 lib 文件作为链接输入。就如同 Windows SDK 中存在大量的 lib 文件

一样。

18.7.2 头文件

如果为第三方应用程序提供程序设计接口，需要提供头文件，头文件中应当包括类型定义、函数声明等内容。这样，在使用第三方开发的应用程序编译时，编译器可以正确生成目标文件。

18.7.3 为第三方应用软件提供SDK

将所需使用的类型定义、函数声明等在头文件中声明，并生成了与 DLL 相对应用 lib 文件一起提供给第三方程序开发人员。第三方的程序开发时引入这些头文件，并在程序连接时将 lib 文件一起链接，就可以调用接口函数了。当然调用时，DLL 文件需在系统所能找到的路径下。