



高等学校计算机教材

Visual C++

网络编程教程 (Visual Studio 2010平台)

◎ 郑阿奇 主编



电子工业出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

<http://www.phei.com.cn>

高等学校计算机教材

Visual C++ 网络编程教程

(Visual Studio 2010 平台)

郑阿奇 主编

电子工业出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书以 Visual Studio 2010 为平台,以全新的角度,通过一系列动态实例揭示网络编程的本质,包括 MFC Socket 编程、Winsock API 编程、即时通信应用开发、HTTP 编程与万维网开发、FTP 编程与资源访问、电子邮件应用编程等。加上介绍相关网络知识、网络环境配置、搭建步骤配合,使读者能方便地理解和运行书中实例。程序之间实现互操作,如客户端和服务器对接、接入第三方程序、程序之间整合集成成为套件等,使书中的每个例子不再是孤立的实体;将书中的实例程序与当下流行的产品软件进行比较,从而激发读者从事实际应用性网络编程开发的热情;适时地由所讲实例延伸开来,开拓读者视野。

本书可作为计算机及相关专业本、专科网络编程的教材或参考书。同时,可为广大计算机爱好者、网络爱好者、编程爱好者、软件发烧友、计算机网络 DIY 玩家成为网络高手贡献一份力。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

Visual C++网络编程教程: Visual Studio 2010 平台 / 郑阿奇主编. —北京: 电子工业出版社, 2013.6
高等学校计算机教材
ISBN 978-7-121-20408-1

I. ①V… II. ①郑… III. ①C 语言—程序设计—高等学校—教材 IV. ①TP312

中国版本图书馆 CIP 数据核字(2013)第 099611 号

责任编辑: 徐 萍

印 刷:

装 订:

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×1092 1/16 印张: 21 字数: 538 千字

印 次: 2013 年 6 月第 1 次印刷

印 数: 3 000 册 定价: 39.50 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线:(010) 88258888。

前 言

我们生活在信息时代，计算机和网络是这个时代的产物和标志。启动计算机、打开浏览器，用户接触到的几乎都是网络应用软件，如各式各样的即时通信工具、下载工具、Web 应用……网络应用软件（又称网络编程）自然成为焦点。

本书以 Visual Studio 2010 作为平台，介绍 Visual C++ 网络编程，具有如下特色。

（1）通过一系列实例揭示一个个典型网络应用的本质，以启发读者的好奇心、探索欲和创新意识。从普通人对信息时代生活的主观体验和感性认识出发，从身边的应用讲起，从现象到本质，由表及里、深入浅出地讲解网络编程。

（2）强调软件程序和网络如鱼儿和水一样密不可分的关系，不仅介绍编程技巧，还适当地介绍相关网络知识并详细给出网络环境配置、搭建步骤，使读者能很方便地开发本书实例。

（3）在本书程序之间实现互操作，如客户端和服务器对接、接入第三方程序、程序之间整合集成成为套件等，使书中的每个例子不再是孤立的实体，而是整个互联网世界的一分子。

（4）联系实际，将书中的例子程序与当下流行的产品软件做比较，指出它们在基本原理上的相通之处，以及书中原型程序的局限性和完善改进的方向，从而激发读者从事实际应用性网络编程开发的热情。

（5）适时地由所讲实例延伸开，介绍当前互联网的真实状况和网络应用的流行趋势，提出很多新观点，为读者打开各种热门新技术、新应用的窗口，对于大家开拓视野、了解网络最新动态、认清信息社会发展的历史潮流提供帮助。

本书从崭新的视角透视网络程序，用形象生动的语言介绍网络编程，软件与自然事物相类比，程序与网络环境相依存，编程技巧与探索实验并重，技术原理与哲理感悟交织。读者只要学过 C++ 语言，懂一点计算机和网络的基本知识，就可以顺利地学习本书。本书可作为普通高校计算机及相关专业本、专科学生学习网络编程的教材或参考书。同时，我们期待本书能成为广大计算机爱好者、网络爱好者、编程爱好者、软件发烧友、计算机网络 DIY 玩家的好朋友，为他们中的更多人成为高手贡献力量。

本书提供同步教学课件和网络应用实例所有源代码文件，网站为 www.hxedu.com.cn。

本书由南京师范大学郑阿奇主编。参加本书编写的还有梁敬东、顾韵华、王洪元、刘启芬、丁有和、曹弋、徐文胜、殷红先、张为民、姜乃松、彭作民、高茜、陈冬霞、钱晓军、朱毅华、时跃华、周何骏、赵青松、周淑琴、陈金辉、李含光、王一莉、徐斌、王志瑞、孙德荣、周怡明、刘博宇、郑进、刘毅、刘友春等。

由于作者水平有限，不当之处在所难免，恳请读者批评指正。

作者 E-mail: easybooks@163.com

编 者

2013 年 4 月

目 录

第 1 章 网络编程和开发环境	(1)
1.1 Visual C++ 2010 开发平台	(1)
1.1.1 Visual Studio 2010 安装	(1)
1.1.2 创建 Visual C++ 项目工程	(4)
1.1.3 Visual C++ 可视化设计	(7)
1.1.4 一个简单的 Visual C++ 小程序	(10)
1.2 网络编程的基本概念	(16)
1.2.1 计算机网络协议	(16)
1.2.2 网络应用编程界面	(19)
1.2.3 网络程序工作机理	(21)
1.2.4 本书编程的协议环境	(23)
第 2 章 MFC Socket 编程	(25)
2.1 MFC 及其 Socket 类	(25)
2.1.1 MFC 简介	(25)
2.1.2 MFC 中的 Socket 类	(26)
2.2 C/S 模式下网络程序的 Socket 通信实例	(29)
2.2.1 客户端—服务器方式 (C/S 模式)	(29)
2.2.2 CAsyncSocket 类编程基础	(31)
2.2.3 CAsyncSocket 类程序的指针实现	(51)
2.2.4 CSocket 类编程	(62)
2.3 Socket 程序的互通	(70)
2.3.1 不同版本 Socket 程序的互通	(70)
2.3.2 接入第三方 Socket 程序	(73)
第 3 章 Winsock API 编程	(75)
3.1 Winsock API 编程原理	(75)
3.1.1 通行的操作	(75)
3.1.2 Winsock API 函数详解	(77)
3.1.3 TCP 与 UDP	(80)
3.2 TCP 编程	(81)
3.2.1 TCP 通信流程	(81)
3.2.2 TCP Socket API 程序设计	(81)
3.2.3 Winsock API 程序与 MFC Socket 程序的等价性	(92)
3.3 UDP 编程	(92)
3.3.1 UDP 通信流程	(92)

3.3.2	UDP Socket API 程序设计	(93)
3.3.3	UDP 进程通信演示	(100)
第 4 章	即时通信应用开发	(104)
4.1	IM 软件的体系结构	(104)
4.1.1	互联网中继通信原理	(104)
4.1.2	P2P 方式架构的系统	(105)
4.2	C/S 结构的聊天室应用	(106)
4.2.1	聊天室功能效果展示	(106)
4.2.2	聊天室的开发	(107)
4.3	P2P 架构的简单聊天工具	(124)
4.3.1	软件使用效果展示	(124)
4.3.2	P2P 通信规约	(125)
4.3.3	聊天工具的开发过程	(126)
4.3.4	P2P 方式通信的特性	(142)
4.4	原型程序与 IM 产品	(146)
4.4.1	本程序与腾讯 QQ 的类比	(146)
4.4.2	IM 产品的增强功能与技术	(147)
4.4.3	即时通信发展新趋势	(148)
第 5 章	HTTP 编程与万维网开发	(150)
5.1	HTTP 原理	(150)
5.1.1	万维网的工作过程	(150)
5.1.2	超文本传输协议	(151)
5.1.3	统一资源定位符 URL	(154)
5.2	浏览器开发	(155)
5.2.1	MFC 对浏览器开发的支持	(155)
5.2.2	定制开发自己的浏览器	(159)
5.3	Web 服务器的开发	(173)
5.3.1	项目框架的建立	(173)
5.3.2	Web 服务器界面总控	(177)
5.3.3	Web 服务流程的实现	(180)
5.3.4	HTTP 协议的实现	(187)
5.3.5	协议实现的辅助代码	(195)
5.4	自制浏览器访问 Web 服务器	(200)
5.4.1	Web 资源准备	(200)
5.4.2	访问 Web 服务器	(201)
5.4.3	相对路径下的资源访问	(203)
第 6 章	FTP 编程与资源访问	(205)

6.1	FTP 应用基础	(205)
6.1.1	FTP 简介	(205)
6.1.2	FTP 的特性	(206)
6.1.3	FTP 工作原理	(207)
6.1.4	FTP 命令和应答	(208)
6.1.5	FTP 网络环境搭建和使用	(211)
6.2	制作 FTP 上传下载器	(215)
6.2.1	WinInet 类对 FTP 的支持	(215)
6.2.2	设计软件界面	(216)
6.2.3	编程实现	(217)
6.2.4	测试 FTP 客户端	(223)
6.3	FTP 服务器的实现	(224)
6.3.1	项目框架的建立	(225)
6.3.2	FTP 服务器界面总控	(230)
6.3.3	FTP 服务流程的实现	(236)
6.3.4	FTP 协议的实现	(244)
6.3.5	FTP 实现辅助代码	(257)
6.4	自制 FTP 客户端与服务器对接	(273)
6.4.1	FTP 上传下载器的改造	(273)
6.4.2	自制客户端访问服务器	(275)
第 7 章	电子邮件应用编程	(278)
7.1	邮件系统原理	(278)
7.1.1	概述	(278)
7.1.2	邮件客户端配置	(279)
7.1.3	邮件收发环境	(284)
7.2	基于 MAPI 的邮件客户端开发	(288)
7.2.1	开发邮件程序的接口 MAPI	(288)
7.2.2	邮件客户端程序开发	(289)
7.2.3	网络邮件收发实验	(300)
7.3	基于 POP3 的邮件接收程序	(306)
7.3.1	POP3 原理	(306)
7.3.2	用 POP3 协议实现邮件接收	(310)
7.3.3	用 POP3 邮件程序接收邮件	(322)

网络编程和开发环境

1.1 Visual C++ 2010 开发平台

Microsoft Visual C++（简称 Visual C++、MSVC、VC++或 VC），是微软公司的 C++开发工具，具有集成开发环境，可编辑编译 C、C++及 C++/CLI 等语言。VC++整合了便利的排错工具，特别是整合了视窗应用编程接口（Windows API）、三维动画 DirectX API 及 Microsoft .NET 框架。目前最新的版本是 Visual C++ 2010。

1.1.1 Visual Studio 2010 安装

Visual C++一直以来都是 Visual Studio 系列开发工具套件的重要成员。Visual Studio 是微软公司推出的著名产品，是目前最流行的 Windows 平台应用程序开发环境。Visual Studio 2010（简称 VS 2010）于 2010 年 4 月 12 日上市，其集成开发环境（IDE）的界面被重新设计和组织，变得更加简单明了。Visual Studio 2010 集成了 Visual C++ 2010。

本书使用网上下载的 Visual Studio 2010 安装包：

cn_visual_studio_2010_ultimate_x86_dvd_532347.iso

这是个镜像（.iso）文件，需要虚拟光驱才能运行。用虚拟光驱软件 DAEMON Tools Lite 载入镜像，如图 1.1 所示。

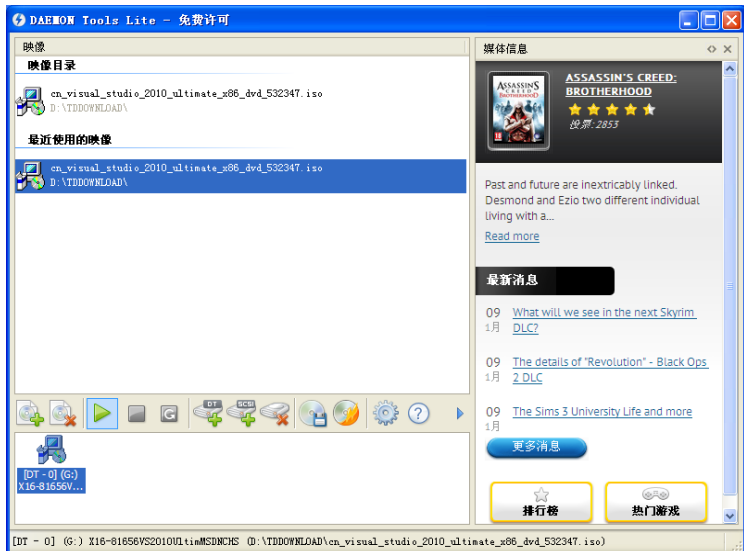


图 1.1 载入 VS 2010 安装.iso 文件

载入后弹出如图 1.2 所示的启动窗口，单击“安装 Microsoft Visual Studio 2010”，进入如图 1.3 所示的安装向导界面，单击“下一步”按钮继续。



图 1.2 启动窗口



图 1.3 Visual Studio 2010 旗舰版安装向导

在图 1.4 所示窗口中选择“我已阅读并接受许可条款”，在图 1.5 所示窗口中选择“完全”，产品安装路径为默认的“C:\Program Files\Microsoft Visual Studio 10.0\”，单击“安装”按钮开始安装进程。

Visual Studio 2010 包含 Microsoft .NET 平台的许多组件，如图 1.6 所示，安装程序会逐个安装它们，用户要等待一段时间。

安装成功后，选择  → “所有程序” → “Microsoft Visual Studio 2010” →  Microsoft Visual Studio 2010，即可启动 VS 2010。初次启动会弹出如图 1.7 所示的“选择默认环境设置”对话框，本书是 Visual C++ 编程，故选择“Visual C++开发设置”，单击“启动 Visual

Studio” 按钮进入开发环境。

Visual Studio 2010 的 Visual C++集成开发环境的起始界面如图 1.8 所示。

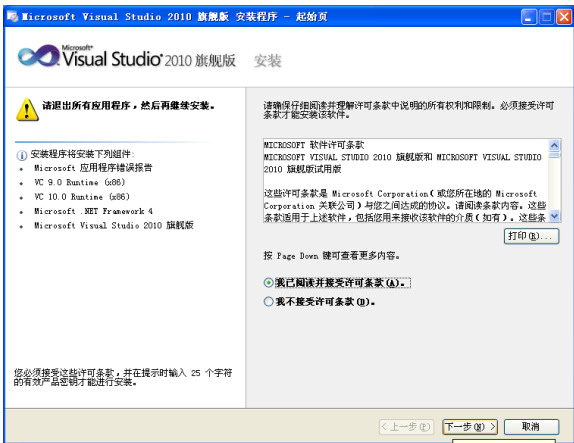


图 1.4 接受安装许可条款



图 1.5 选择安装路径

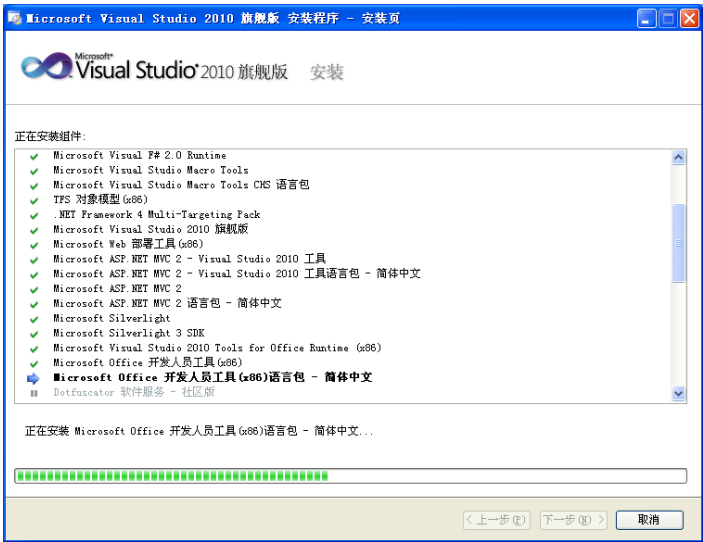


图 1.6 安装进行中

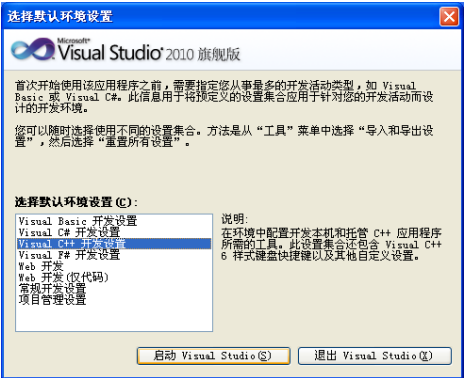


图 1.7 设置为 Visual C++的开发环境



图 1.8 Visual C++ 2010 集成开发环境

读者也可从网络获得 Visual Studio 2010 的可执行（非镜像）安装程序或者直接从光盘安装。

下面以编写一个简单的演示程序为例，使读者初步熟悉 Visual Studio 2010 环境下 Visual C++ 开发的基本操作。

1.1.2 创建 Visual C++项目工程

选择菜单命令“文件”→“新建”→“项目”，如图 1.9 所示。

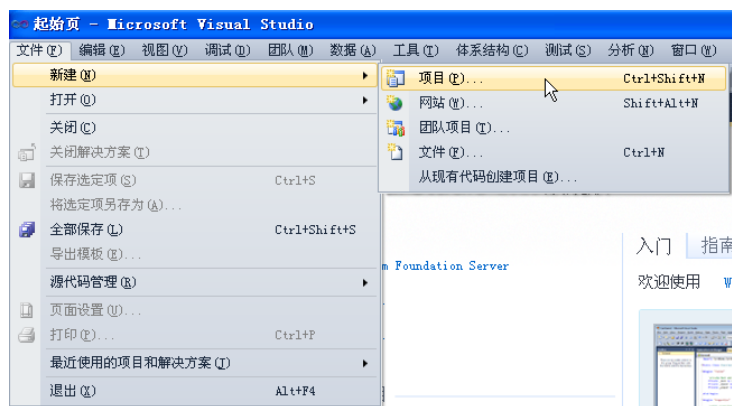


图 1.9 新建项目

系统弹出“新建项目”对话框（如图 1.10 所示），左边“项目类型”树中默认选项为“Visual C++”→“MFC”，对应右边“模板”选择“MFC 应用程序”，给项目命名为“GetIPAndPort”（我们即将做的这个软件是用来演示程序如何获得用户输入的 IP 和端口号的——这也是几乎所有网络程序都具有的功能，所以取这个名字）。

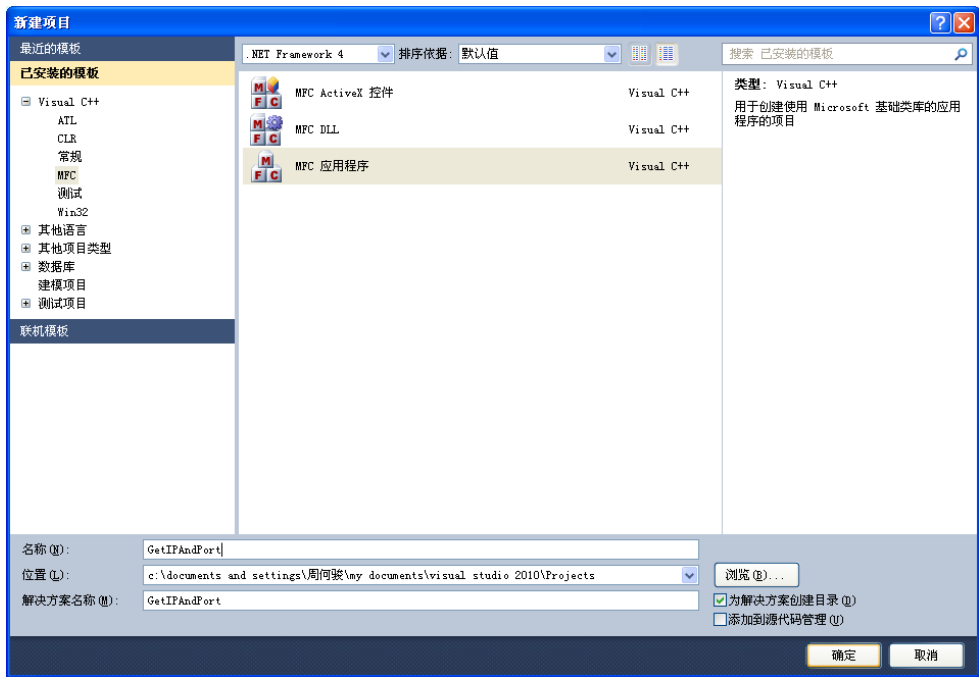


图 1.10 项目命名

单击“确定”按钮，弹出“MFC 应用程序向导”对话框（如图 1.11 所示），接下来我们将在这个对话框的指引下轻松完成创建 VC 工程的工作，单击“下一步”按钮继续。



图 1.11 MFC 应用程序向导

在“应用程序类型”界面（如图 1.12 所示）选中“基于对话框”单选按钮（这个程序很简单，用不着文档和视图），取消选择界面下方的“使用 Unicode 库”复选框（在本书所有程序建立工程的时候都要记得这一步，为程序兼容性考虑，避免字符串处理的麻烦），单击“下一步”按钮。



图 1.12 选择应用程序类型

接下来的“用户界面功能”和“高级功能”界面（如图 1.13 所示）都采用系统默认设置，连续单击“下一步”按钮跳过。

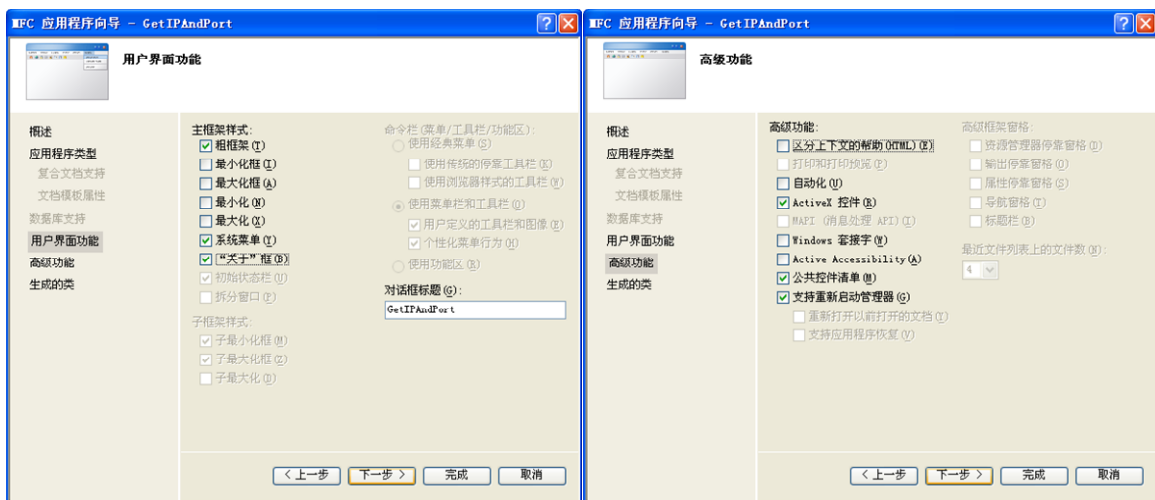


图 1.13 “用户界面功能”和“高级功能”默认设置

最后一步出现的是“生成的类”，稍留意下可以看到，系统已经自动为程序建立了两个类——CGetIPAndPortApp 和 CGetIPAndPortDlg（如图 1.14 所示），其中 CGetIPAndPortApp 类代表应用程序本身，CGetIPAndPortDlg 类代表程序的主界面对话框。细心的读者可能会发现，这两个看似冗长的类名其中间部分“GetIPAndPort”就是我们刚才取的工程名！没错，以后大家就会发现一个规律：VC 在一开始创建工程时都会默认生成两个类，名称形如 CXXXXApp 和 CXXXXDlg（其中“XXX”部分就是用户指定的工程名），这样的命名法则是为了方便用户理清程序的类结构。

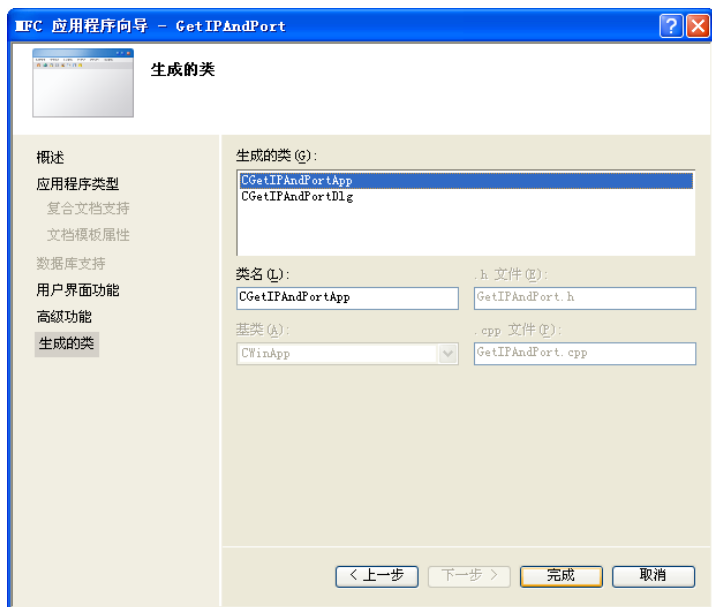


图 1.14 生成的类

单击“完成”按钮，至此一个 VC 工程就创建完成了。

1.1.3 Visual C++可视化设计

开发环境工作区主界面将呈现的样子如图 1.15 所示。

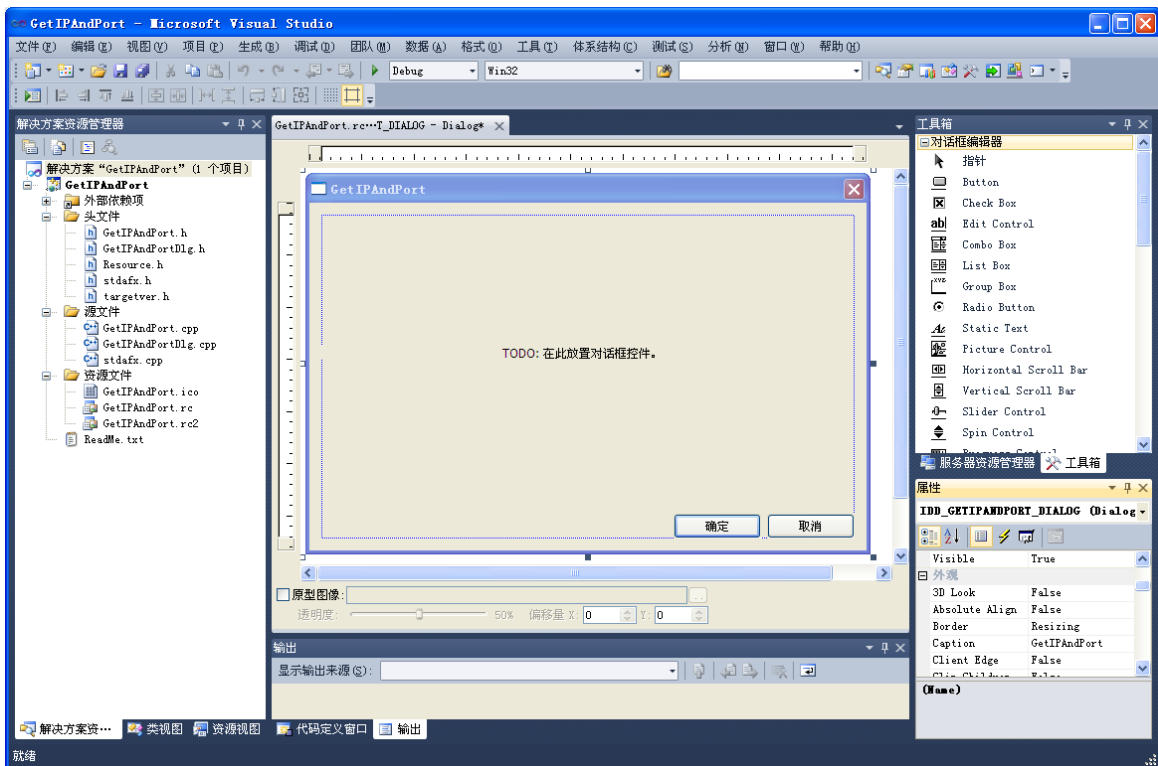


图 1.15 开发环境工作区主界面

主工作区大致分为三部分，最左边是供用户浏览程序结构的，包括好几个选项卡界面，常用的是解决方案资源管理器、类视图和资源视图，如图 1.16 所示。

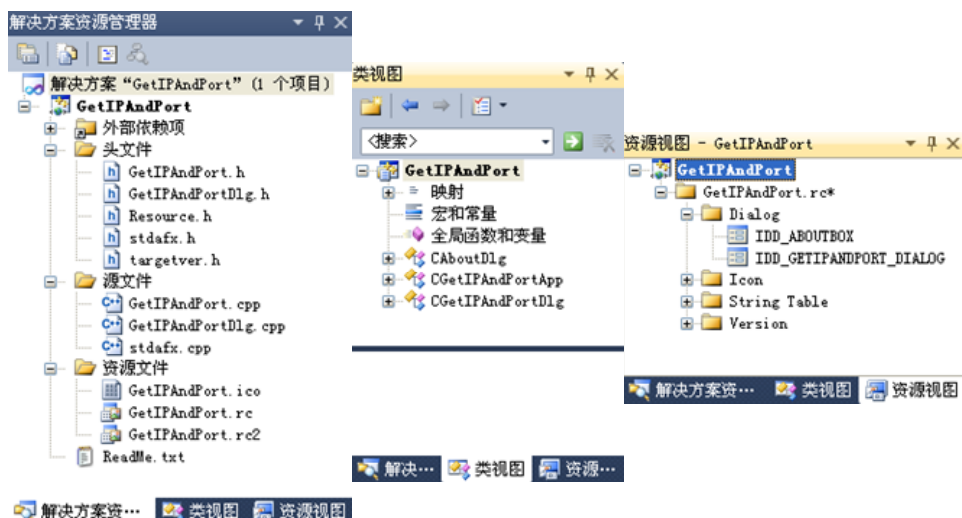


图 1.16 三个常用的视图

解决方案资源管理器以树形目录结构列出了程序中包含的所有代码文件，VC 将它们分成三大类：头文件（.h）、源文件（.cpp）和资源文件。头文件主要对程序中用到的各种变量、常量、函数和类进行定义和声明；源文件是程序的主体部分，是各个函数、类的具体实现代码；资源文件定义程序运行要用到的各种资源，如图片、动画、声音、视频等，这对于一些功能丰富的软件（如多媒体类应用）是必不可少的。

类视图用树状结构展示了工程中所有 C++ 类及其层次结构，单击类名可在 Visual Studio 2010 环境界面右下角的“属性”窗口中设置对应的类，包括为其添加新的事件消息，重写某些方法的实现代码等。

资源视图分类列出了程序的所有资源，其中常用的是 Dialog 资源，这种资源是每一个具有 GUI 的程序都有的，双击资源 ID 号可以打开对应的界面设计工作区，就可以设计程序的图形界面了。

VC 在创建工程时默认会生成两个类（本例是 CGetIPAndPortApp 和 CGetIPAndPortDlg），图 1.16 解决方案资源管理器中的头文件 GetIPAndPort.h 和源文件 GetIPAndPort.cpp 对应 CGetIPAndPortApp 类，它们一同构成了该类的源代码实现；同理，GetIPAndPortDlg.h 和 GetIPAndPortDlg.cpp 对应 CGetIPAndPortDlg 类，从中间类视图中也可以看到这两个类。

除此之外，读者会发现类视图里多了一个 CAboutDlg 类，它是 VC 自动创建的，叫做“关于 GetIPAndPort”对话框，用来显示程序的版本信息。

在本例的三个类中，CAboutDlg 类和 CGetIPAndPortDlg 类都有各自的对话框界面资源。资源视图中 Dialog 目录下有它们的 ID 号（对应的分别是 IDD_ABOUTBOX 和 IDD_GETIPANDPORT_DIALOG），双击 ID 号可以打开其对话框的界面设计工作区，如图 1.17 所示。

工作区中显示的是“关于 GetIPAndPort”对话框的默认界面，可以在此基础上自己设计或重新布局。

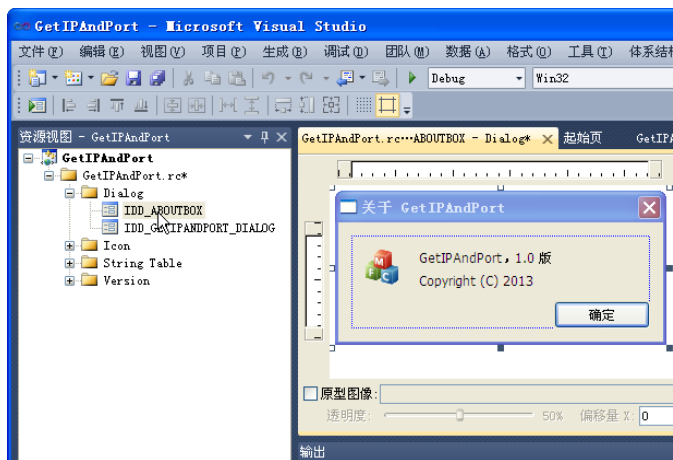


图 1.17 “关于 GetIPAndPort”对话框的设计工作区

解决方案资源管理器、类视图和资源视图三者密切配合，将程序代码有机地组织成一个结构精巧的 C++ 项目，通过这三者可以清楚地看到程序中每个类的对应代码实体，用 C++ 写的类不再是抽象的代码，而是成为看得见、摸得着的现实存在，这就是可视化设计的魅力！

对话框界面的设计布局如 Visual Basic 一样方便：开发环境界面中央是主工作区，在这里可以打开任意多个程序文件（源文件或头文件）及对话框界面设计工作区；只要将右边工具箱中的控件直接拖曳到工作区中，就可以设计出自己想要的程序界面。VC 环境下的工具箱和 VB 的一样，都包含了一般 Windows 程序通用的界面元素（按钮、文本框、静态标签、列表框、滚动条等），用过 VB 的人对这些常用控件肯定再熟悉不过了，故这里不做过多介绍。只是针对本书所介绍的网络编程，有一个控件需要特别提一下，那就是 IP 地址控件（如图 1.18 所示）。它在界面上的显示效果如图 1.19 所示。

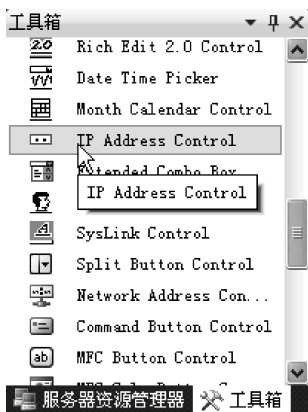


图 1.18 IP 地址控件

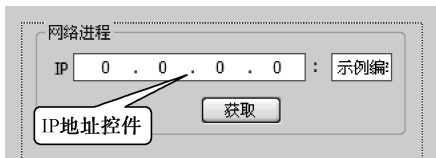


图 1.19 地址控件的显示效果

稍后将会介绍这个控件的具体用法，读者会看到它是个很实用的控件。

选择工具箱中的控件设计程序界面，可以看到 VC 界面设计环境的使用极其方便，丝毫不比 VB 逊色。在布局界面时可以使用工具栏中提供的功能调整各个控件的大小、对齐方式。如图 1.20 所示，先选择“关于”按钮控件，再选择“退出”按钮控件，然后单击工具栏中的“使大小相同”按钮，就可以使先选择的按钮与后选择的按钮大小一样。

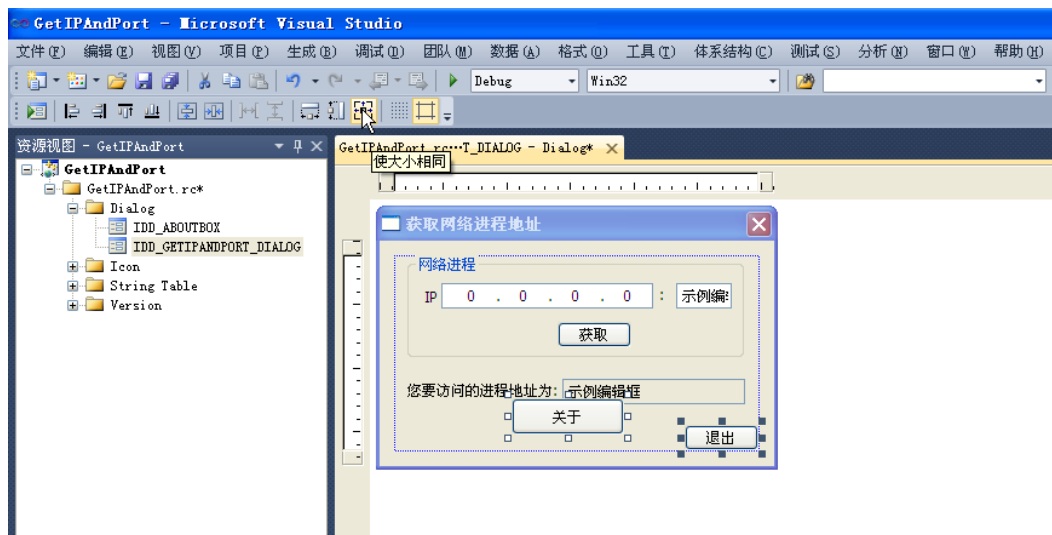


图 1.20 调整控件的大小

最终设计出的程序界面效果如图 1.21 所示。

在网络编程中，程序使用 IP 和端口来标识网络上的其他程序，以实现程序之间的通信（进程通信），因此获取并正确处理对方进程的 IP 和端口（通常由用户通过界面输入指定）就成为网络程序的通用功能。下面的小程序就用来演示这个最基本的功能，不过，它只负责获取和处理 IP、端口号，并不实际发起网络连接。



图 1.21 程序界面

1.1.4 一个简单的 Visual C++ 小程序

这个程序的界面已经设计好了，但要让程序完成一定的功能，还必须为其编写代码。写代码之前，首先要定义程序中需要用到的一些变量。在 VC 中，很多变量都不是孤立的，而是与某个界面元素（即控件）绑定的，这样用户在界面上的输入就可以很容易地传递给程序中相应的变量进行处理。例如，为了在程序代码中获得用户输入的 IP，需要给 IP 地址控件关联一个变量。如图 1.22 所示，右击该控件，在弹出的菜单中选择“添加变量”命令。

出现“添加成员变量向导”对话框（如图 1.23 所示），将变量命名为“m_ip”，变量类别为“Control”。

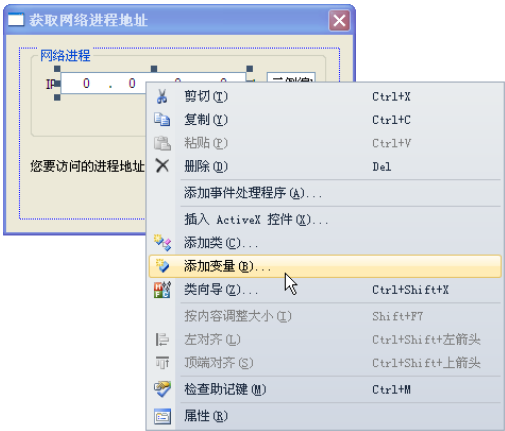


图 1.22 给控件关联变量



图 1.23 添加 Control 型变量

“Control”表示控件变量，它是微软对实现 Windows 程序的图形用户界面元素的一些类（类名如 CEdit、CButton）的总称。这些类大多是由同一个叫做窗口（CWnd）的 C++类衍生过来的，CWnd 及其庞大的衍生类家族封装了 Windows 的 GUI，作为 MFC 的重要组成部分，是微软对基础 C++语言的扩充，使其具有了强大的图形化界面功能。与 CWnd 家族中的类关联的变量即控件变量，可以通过这样的变量获取和控制它所关联控件的几乎一切行为，从而对程序界面进行灵活的编程和定制。

而那些仅仅只是获取用户输入值的变量称为值类型（Value）变量，这种变量与我们以前上 C++语言课时接触的那些变量差不多，所不同的只是它们都有各自关联的控件，用于获取和保存特定控件接收的用户输入。

除了 Control 和 Value 这两大类变量外，用户在编程时也可以根据根据需要自定义临时变量，这种变量就是普通 C++里用的那些变量（整型、实型、字符型之类）。

下面接着添加变量，给用于接收端口号的文本框关联 Value 变量 strport（如图 1.24 所示）。



图 1.24 添加 Value 型变量

再设置该文本框的 Number 属性为 True（如图 1.25 所示），之所以这样设置，是为了限定用户只能在这个文本框中输入数字形式的端口号。这样设置之后，后面运行程序时读者会发现：如果试图输入非数字字符（中文、英文字母），则文本框一概不响应，也就杜绝了用户的非法输入。

本程序还有一个文本框是用于显示程序获取的 IP 和端口的，给它关联 Control 型变量 m_showIpAndPort，并且设置 Read Only 属性为 True（作为显示信息窗口的文本框一般都设为只读模式），如图 1.26 所示。

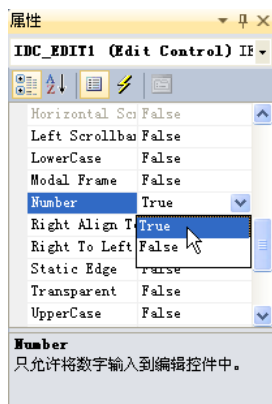


图 1.25 设置文本框 Number 属性



图 1.26 设置显示框 Read Only 属性

至于给控件关联的变量究竟是设成 Control 还是 Value 型，应视具体需要而定。一般来说，如果希望对控件的行为进行某种控制，设置变量为 Control 型较好；如果仅仅想获得控件接收的用户输入值，设置为 Value 型就足够了；有时候对控件的行为有较高的控制要求，同时又希望能够非常方便地获取控件的输入值，也可以对一个控件同时关联定义这两类变量。

现在,所有的变量都关联好了,界面控件也都进行了恰当的设置和布局,只剩下编码工作了。程序界面上的“获取”按钮是实现本程序功能的关键,编程工作主要就是给这个按钮添加事件处理程序。右击“获取”按钮,在弹出的功能菜单中选择“添加事件处理程序”命令,如图 1.27 所示。

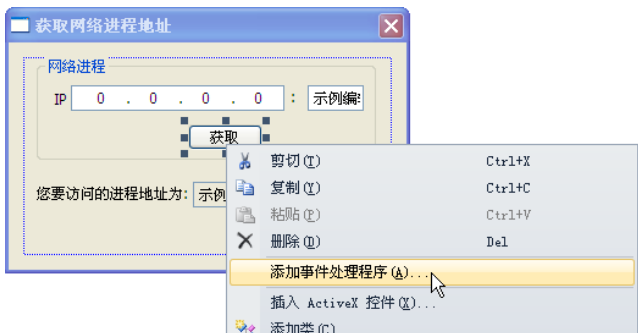


图 1.27 给“获取”按钮添加事件处理程序

在“事件处理程序向导”对话框中将这个处理程序命名为“OnShowIpAndPort”,如图 1.28 所示。



图 1.28 给事件处理程序命名

单击“添加编辑”按钮,进入代码编辑窗口(如图 1.29 所示),VC 自动打开需要编辑的程序代码文件 `GetIPAndPortDlg.cpp` 并定位到源文件中相应的位置,供用户添加自己的代码。以后在编译、调试程序时,也可随时先选中程序界面上的“获取”按钮,然后再在开发环境右下角“获取”按钮的属性设置窗口中选择这个事件处理过程。这样就可以随时随地定位到这个事件过程的源码编辑处,修改完善程序。需要指出的是,事件过程 `OnShowIpAndPort` 的代码必须填写在如图 1.29 中 VC 为我们指定的地方,写在其他任何地方或本工程的其他代码文件中都是无法运行的。

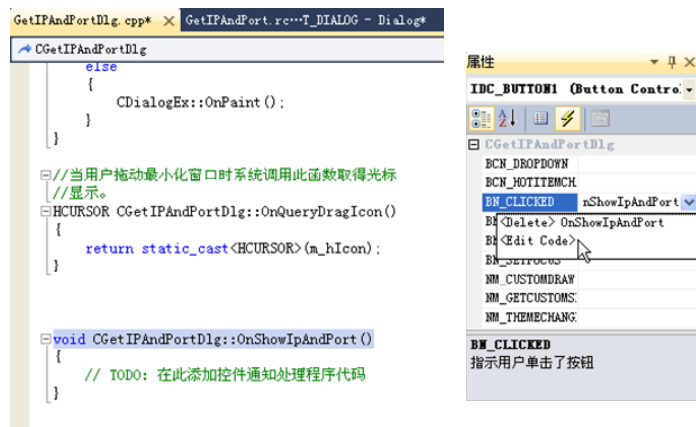


图 1.29 编辑事件处理过程代码

为“获取”按钮编写的事件过程代码如下：

```
BYTE nFild[4];           //分别存放 IP 地址的四个字段
CString sip;              //IP 地址的字符串形式（可以直接显示在界面上的）
UpdateData();             //刷新对话框界面，获取用户输入
//验证输入是否合法
if(m_ip.IsBlank())        //若用户没有填写 IP 地址，则提示填写
{
    AfxMessageBox("请填写 IP 地址!");
    return;
}
if(strport.IsEmpty())     //若用户忘了指定端口号，则提醒其指定
{
    AfxMessageBox("请指定进程端口!");
    return;
}
//获取用户输入的 IP 地址值
m_ip.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
//将 IP 地址格式化为可以在计算机屏幕上显示的字符串
sip.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
//在界面上显示用户输入的网络进程地址（包括所在主机的 IP 和端口）
m_showIpAndPort.SetWindowTextA(sip + " ." + strport);
m_ip.SetFocus();          //焦点回到 IP 地址栏
```

双击“关于”按钮，为其添加事件过程（如图 1.30 所示）。VC 也会自动定位到需要添加代码处，事件处理过程函数自动命名为 OnBnClickedButton2，表示在单击 Button2（即“关于”按钮）时执行这个事件过程。

“关于”按钮的 Click 事件代码如下：

```
CAboutDlg dlg;
dlg.DoModal();           //显示“关于”对话框
```

由此可见，添加代码有两种方式：一种方式是右击控件后选择“添加事件处理程序”菜单项，另一种方式是直接双击该控件。所不同的是第一种方式可以自定义事件处理程序的函数名，如本例命名“获取”按钮的单击处理过程函数名为 OnShowIpAndPort，这是个有意义的名字，表示单击“获取”按钮后，程序就会显示（Show）指定的“IP”和“端口（Port）”；而采用第二种方式

添加的函数使用系统自动生成的函数名，如 OnBnClickedButton2，虽然这个名称也有一定意义，表示事件过程是在单击（BnClicked）第二个按钮（Button2）时发生的，但并没有直观地表示出它要完成的功能。

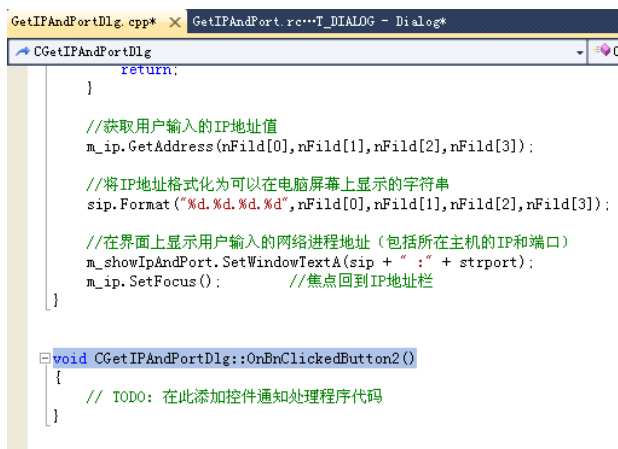


图 1.30 添加“关于”按钮事件过程

对于小程序的开发，这两种方式都是可行的，并没有本质区别。但是如果程序规模稍大，事件处理过程较多时，就要注意了：采用第一种方式添加的函数由于其名称更有意义，将使得最终整个源程序的可读性很强，便于修改、维护，因此对于程序的重要功能函数尽量使用第一种方式添加；而第二种方式也不是完全没有用——对于一些不重要的函数（并非程序必要的核心功能），若都采用第一种方式添加，都要给它取一个有意义的名字，势必给编程工作造成不便，而且过多地增加有意义的函数名也容易造成命名混乱，反而降低了程序的可读性。在实际编程中，要根据具体情况灵活运用这两种方式。

至此，这个简单的 VC 小程序就编写完成了，下面来运行。

单击工具栏上的“启动调试”（）按钮，程序界面如图 1.31 所示。

我们先不填写 IP 地址而直接单击“获取”按钮，看看会发生什么。程序弹出了消息框，提醒填写 IP 地址（如图 1.32 所示）。



图 1.31 启动程序界面



图 1.32 提醒填写 IP 地址

接下来填写 IP 地址时，我们故意将最后一个字段填成 300（或任何其他大于 255 的十进制数），这时你会发现：IP 地址控件会自动将它重置为 255（因为值大于 255 的 IP 地址字段是不合法的），而且始终无法在 IP 地址控件中输入其他非数字字符（这就是使用 IP 地址控件的好处）。如果使用其他控件接收用户输入，用户就要自己编写代码来检查输入的合法性，IP 地址控件将这种烦琐的验证过程封装起来自动完成，给编写程序带来了极大的方便。

填写完合法 IP，再故意不填端口号，单击“获取”按钮后，程序同样也会弹出消息框，提醒输入端口号（如图 1.33 所示）。



图 1.33 提醒填写端口

在 IP 和端口都合法填写的情况下，单击“获取”按钮，程序就会将用户输入的网络进程地址（IP+端口）显示在下方的输出文本框中，如图 1.34 所示。

这样，程序就完成了对用户输入网络进程地址的获取工作，在本书以后的程序示例中，这样的操作会经常用到。在获取了进程地址后，程序会进一步用这个地址作为参数传递给不同的网络编程接口函数，从而完成丰富的网络功能。

单击“关于”按钮，程序弹出“关于 GetIPAndPort”对话框（如图 1.35 所示），这是一个版本声明对话框，每一个 Windows 程序几乎都有这样一个对话框，用于声明版本号和版权信息。对话框上显示的内容可以根据需要进行修改和自定义，这个小程序暂时保持默认内容。



图 1.34 显示运行结果



图 1.35 “关于 GetIPAndPort”对话框

通过以上这个简单的小程序，读者已经初步熟悉了本书要使用的开发平台，但要进行网络编程还必须对一些基本概念有所了解，下面就来介绍。

1.2 网络编程的基本概念

1.2.1 计算机网络协议

网络中的计算机要做到有条不紊地交换数据，就必须遵守一些事先约定好的规则，这些规则、标准或约定称为网络协议（Network Protocol），它主要由以下三个要素组成：

- （1）语法，即数据与控制信息的结构或格式；
- （2）语义，即需要发出何种控制信息，完成何种动作，以及做出何种响应；

(3) 同步，即事件实现顺序的详细说明。

今日互联网广泛使用的网络协议是著名的 TCP/IP，它是在 20 世纪 60 年代后期，由阿帕网的研究机构美国国防部高级研究计划署 (ARPA) 开发的。该协议将网络功能划分成独立的四层结构，自上而下分别是应用层、传输层、网际层和网络接口层，如图 1.36 所示。

其实 TCP/IP 并不是一个单独的协议，而是由一系列网络协议所组成的协议集合（协议族），这个庞大的协议家族按照如图 1.37 所示的分层结构组织起来构成的有机整体称为网络协议栈。



图 1.36 TCP/IP 体系结构

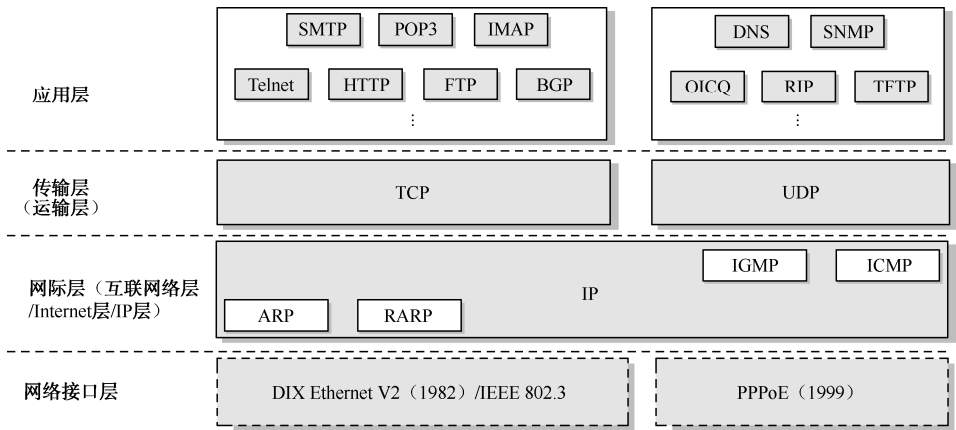


图 1.37 TCP/IP 协议栈

下面分别介绍栈中各层的主要功能（自上而下）。

1. 应用层（Application Layer）

应用层在 TCP/IP 协议栈的第 4 层，即最高层，它提供面向用户的网络服务，如进行文件的传输服务和远程登录服务等。不同的用户，对应用层服务的需求不同，因此应用层定义了许多面向用户的、提供特定服务的协议。比较常用的有远程登录协议（Telnet）、文件传输协议（FTP）、超文本传输协议（HTTP）、域名系统（DNS）、简单网络管理协议（SNMP）、简单邮件传输协议（SMTP）、邮局协议（POP3）和即时通信协议（OICQ）等。由于传输层可以使用 TCP，也可以使用 UDP，因此，有些应用层协议是基于 TCP 的（如 FTP 和 HTTP 等），有些应用层协议是基于 UDP 的（如 SNMP 和 OICQ 等）。

两个应用进程能够通过网络进行最基本的消息传递，但是要实现具体的应用，还必须使这种消息传递过程按照一定的规范进行，应用层上诸多的协议正好提供了这种规范。但这些协议只能满足普通用户在一般情况下使用网络的需求，如果用户要在网络上进行一些特殊应用，如网吧管理或供某个公司内部使用的邮件系统等，应用层并没有提供这样的协议，就要由用户根据自己的实际需要开发新的应用协议了。

2. 传输层（Transport Layer）

在应用层之下的是传输层，有的书中也称“运输层”。在 TCP/IP 协议栈中，传输层处于第 3 层。传输层完成通常所说的两台主机之间的通信，其实质是两台主机上对应的应用进程之间的通信，也叫端到端（End to End）通信。端到端通信是在传输层两个通信实体（进程）之间进行的，就好像是在两个实体之间建立了一条逻辑通路一样，它屏蔽了 IP 层的路由选择和物理网络等细节。

实际的通信中, 传输层应用进程对通信质量的要求是不一样的。为了满足不同的需要, 在 TCP/IP 协议栈中该层定义了两个不同的协议: 一个是 TCP (传输控制协议), 另一个是 UDP (用户数据报协议)。

TCP 为两台主机提供了高可靠性的数据通信服务, 可以将源主机的数据流无差错地传输到目标主机。当有数据要发送时, 它对应用进程送来的数据进行分片, 以适合于在网络层中传输; 当接收到网络层传来的分组时, 它要对收到的分组进行确认, 还要对丢失的分组设置超时重发等。为此 TCP 需要增加额外的许多开销, 以便于在数据传输过程中进行一些必要的控制, 确保数据的可靠传输。

UDP 则为应用层提供了一种非常简单的服务, 它只是把称为数据报的分组从一台主机发送到另一台主机, 但并不保证该数据报能正确到达目标端, 通信的可靠性必须由相应的应用程序来提供。

综上所述, TCP 和 UDP 各有其特点, TCP 可以确保数据传输的可靠性, 但由于需要额外的开销, 所以传输效率比较低; UDP 虽然不能保证数据传输的可靠性, 但数据传输的效率比较高。用户在开发应用程序时, 可以根据实际通信情况, 选用 TCP 或 UDP 进行数据传输。

3. 网际层 (Internet Layer)

网际层在 TCP/IP 协议栈的第 2 层, 也称为互联网络层 (互联层) 或 Internet 层, 因该层的主要协议是 IP, 所以也可简称为 IP 层。它是 TCP/IP 协议栈中最重要的一层, 主要功能是把源主机上的分组根据需要发送到 Internet 中的任何一台目标主机上。当然在一般情况下, 发送信息的源主机要知道接收信息的目标主机的“地址”。通信时, 源主机与目标主机可以在同一个网络中, 也可以在不同的网络中。在网际层传输的数据单位叫 IP 数据报, 也称为 IP 分组。在一个由很多网络组成的 Internet 中, 一台主机 (即源主机) 与不在同一个网络中的另一台主机 (目标主机) 通信时, 可能有多条通路相连, 网际层的一个重要功能就是要在这些通路中做出选择, 即所谓的路由选择, 它是网际层一个非常重要的功能。网际层的本质是在各种物理网络基础上, 使用 IP 将它们互联, 组成一个传输 IP 数据报的虚拟网络, 所以网际层实现了不同网络的互联功能。但网际层提供的是一种“尽力而为”的数据报传输服务, 它不能保证数据总是可靠地从源主机传输到目标主机。在 TCP/IP 协议栈中, 网际层协议包括 IP (网际协议)、Internet 控制报文协议 (Internet Control Message Protocol, ICMP)、Internet 组管理协议 (Internet Group Management Protocol, IGMP) 等。

4. 网络接口层 (Host-to-Network Layer)

网络接口层处于 TCP/IP 协议栈的最低层, 它负责将其之上的网际层要发送出去的数据 (即 IP 数据报) 发送到其下面的物理网络, 或接收由物理网络发送到该目标机的数据帧, 并抽出 IP 数据报交给网际层。要注意, 这里所说的物理网络是指各种实际传输数据的局域网或广域网等。在 TCP/IP 协议栈中并没有具体定义网络接口层的内容, 一般情况下, 认为只要是在其上能进行 IP 数据报传输的物理网络 (如目前局域网中占主导地位的以太网, 采用各种数据通信技术的电信网络——ATM、帧中继、DDN、B-ISDN 等), 都可以当成 TCP/IP 协议栈接口的网络, 这就是为什么图 1.37 中用虚线框画出它。

对于日常使用的计算机, 厂家都会把以太网 MAC 协议做在主板的网卡中。鉴于目前所有的局域网几乎都是以太网的现状, 普通人的计算机一般都会支持以太网的两个兼容协议——DIX Ethernet V2 和 IEEE 802.3 系列。广大上网用户要么是通过局域网接入 Internet, 要么就是自家装宽带, 使用 ISP (电信运营商) 提供的接口, 后者几乎无一例外都遵循点对点协议 (PPP), 而宽带用户的 PPP 又是在以太网 (Ethernet) 上运行的, 故称 PPPoE, 宽带用户的计算机都是通过 PPPoE

接入运营商的各种通信网络的。综合上述这些现实情况，图 1.37 在网络接口层只标出了 DIX Ethernet V2、IEEE 802.3 和 PPPoE 这三个协议。

为什么在 TCP/IP 协议栈中没有定义网络接口层呢？这是因为不定义网络接口层的具体内容如下两点好处。

(1) 便于实现不同网络之间的互联。实现不同网络的互联是 TCP/IP 要解决的最主要问题。不同的网络尽管其数据传输介质、传输速率等有很大差异，但都可以实现网络内数据的传输，当然也就可以进行 TCP/IP 协议栈中网际层 IP 数据报的传输。这样 TCP/IP 就可以将重点放在网络之间的互联上，而不用去纠缠各种物理网络的具体实现细节，这就非常巧妙地解决了不同类型物理网络的互联问题。这也是 TCP/IP 得以广泛应用的一个重要原因。

(2) 为将来物理网络的发展留下广阔的空间。物理网络与计算机硬件技术和通信技术的发展紧密相连，如物理网络中数据传输速率在不断提高，网络中设备的性能也在不断提升，所有这些都不会影响 TCP/IP 的使用。

1.2.2 网络应用编程界面

前面说到 TCP/IP 网络协议栈的体系结构，网络应用开发实质上就是去实现某种应用层协议（标准的通用协议或用户自己设计的）。随着 Internet 应用多样化的发展，应用层的协议处于快速的更新换代之中，不断地有新协议被设计出来投入使用。虽然那些通用的协议如 HTTP、FTP、SMTP 等基本不变，但不同的软件开发商在实现自己产品的时候都会或多或少地对它们适当地加以改造和扩展，至于具体实现的方式更是千差万别，因此从这个意义上来说，协议栈的应用层是不稳定的。鉴于这种状况，肯定不能将应用层作为网络应用开发的通用平台。

再来看协议栈的最低层即网络接口层，为了适应通信技术的日新月异和物理网络的多样性，TCP/IP 根本就没有对该层做任何具体的规定。

这样一来，四层网络协议栈只有两层是稳定的并且有具体的标准规范，也就是传输层和网际层，它们的协议几乎是随着 Internet 和 TCP/IP 的诞生而产生的，几十年来没有什么大的变动并已有了很成熟的实现。我们将传输层的 TCP 和 UDP 加上网际层的 IP 这三个协议合起来作为 Internet 的核心协议（ICMP/IGMP 也是 IP 封装的，可看成 IP 的扩展协议）。如此看来，“TCP/IP”这个说法是不准确的，准确而全面的称呼应该是“TCP/UDP/IP”。但长久以来人们已经习惯于说 Internet 使用的是“TCP/IP”，也就没有必要改名称了。但读者心里必须十分清楚：Internet 的核心协议是三个，而且只有这三个，它们是 TCP、UDP 和 IP，通常将它们三个作为一个整体，几乎所有的主流操作系统都在内核中实现了这三个协议。大家平时上网的时候如果单击桌面右下角任务栏上表示网络连接的图标，在弹出的网络连接状态对话框中单击“属性”按钮，就可以在连接使用的项目列表中看到已经内置于操作系统之中的 TCP/IP 了，如图 1.38 所示。

Internet 核心协议（TCP、UDP 和 IP）作为网络应用开发通用的基础平台，为开发应用的程序员提供服务，我们把这个平台称为“TCP/IP 核心协议栈”。在很多书籍文献资料中（甚至在一般场合），人们所说的“TCP/IP”往往就是指由这三个协议构成的应用开发平台，并不包括应用层诸多协议和最下面的网络接口层，这一点请大家务必注意！

学习网络编程首先要搞清楚要做的是哪一个层次上的编程工作。目前网络的四层体系结构中的三层已经有了成熟可靠的实现实体，如图 1.39 所示。

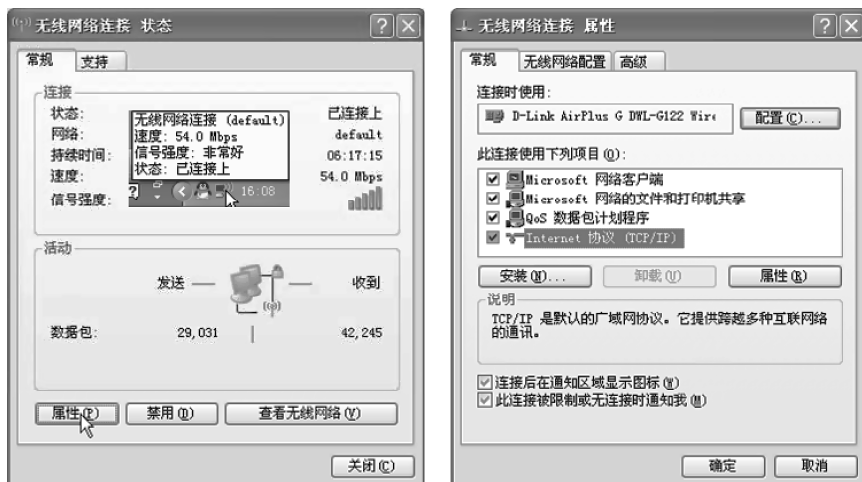


图 1.38 内置于操作系统之中的 TCP/IP

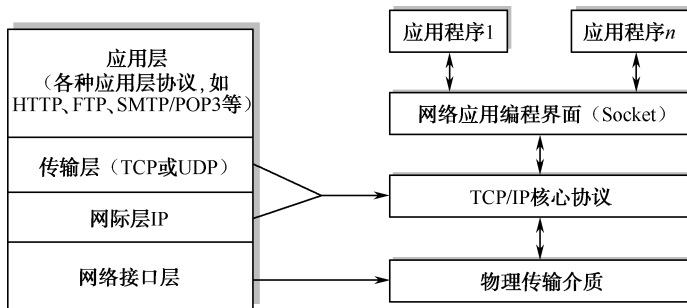


图 1.39 TCP/IP 体系的实现情况

其中，网络接口层已经被绝大多数计算机生产厂家集成在了主板上（就是网卡，又叫网络接口卡 NIC），除非搞计算机硬件研发，一般学软件的都不用管这一层是怎么做出来的；网际层和传输层也已经实现好了，Windows 操作系统内核中就集成了 TCP/IP 的实现，它的核心部分是传输层协议（TCP 与 UDP）、网际层协议（IP）。一般用户感受到的只有应用程序（包括系统应用程序）。那么，应用程序通过什么样的界面与内核打交道呢？通过编程界面（即程序员界面）。各种应用程序，包括系统应用程序都是在此界面上开发的。

编程界面有两种形式，一种是由内核直接提供的系统调用，在 Windows 下表现为 Windows API 函数；另一种则是以程序库方式提供的各种函数和类。前者在核内实现，后者在核外实现。MFC 就是微软用 C++ 语言对 Windows API 进行面向对象封装后形成的功能强大的类库。

TCP/IP 网络环境下的应用程序是通过网络应用编程界面（Socket，即套接字）实现的。

（1）Windows Socket 的概念。Windows Socket 顾名思义，就是在 Windows 环境下使用的 Socket，那么 Socket 又是什么呢？它是一套网络编程机制（或规范），常简称为 Winsock。该规范是在 20 世纪 90 年代初制定的，是在 Windows 操作系统下得到广泛应用的、开放的、支持多种协议的网络编程接口。从 1991 年的 1.0 版到 1997 年的 2.2.1 版，经过不断完善并在 Intel、Microsoft、Sun、SGI、Informix、Novell 等公司的大力支持下，现在已经成为 Windows 环境下网络编程事实上的标准。

（2）Windows Socket 的来源。Socket 最早是 UNIX 操作系统下流行的一种网络编程接口，于 1983 年在 Berkeley（加州大学伯克利分校）4.2 BSD 操作系统中被首先引入，因此被称为“Berkeley Socket API”。20 世纪 70 年代，BSD 作为一种主流的 UNIX 操作系统被广泛使用在各公司、大学

的大中小型主机上,经历了当时世界范围的网络互联高潮。在美国政府和军方的推动下,阿帕网以 TCP/IP 为规范与全美各个大公司、名牌大学的网络互联,因此“Berkeley Socket API”模型自然也就成了 TCP/IP 网络的编程接口标准。从 4.2BSD 开始, Berkeley 套接口 API 在不断地完善和发展,一直到 1995 年的 4.4 BSD-Lite 2 为止。

Windows 的网络编程接口 Windows Socket API 就是在 1991 年根据 4.3 BSD 操作系统的“Berkeley Socket API”制定的。

1.2.3 网络程序工作机理

应用程序网络功能的实质是,与网络上其他计算机(主机)中的程序互通消息,也就是借助网络实现进程通信。因此如何实现网络中不同主机上的程序之间的通信也就成了网络程序实现的最为基础的技术。

在同一台计算机的操作系统中,不同的两个进程间进行通信时,通过系统分配的进程号(Process ID)就可以唯一标识一个进程,也就是说两个相互通信的进程,只要知道对方的进程号就可以进行通信。而网络情况下进程间的通信问题就要复杂得多,不能简单地只用进程号来标识不同的进程。首先要解决如何识别网络中不同主机的问题,其次,因各个主机系统中都独立地进行进程号分配,并且不同系统进程号的产生与分配策略也不同,在这种情况下就不能再通过进程号来简单地识别两个相互通信的进程了。

在网络中为了标识通信的进程,首先要标识进程所在的主机,其次要标识主机上不同的进程。在互联网中使用 IP 地址来标识不同的主机,在网络协议中使用端口号来标识主机上不同的进程。这样,为了唯一地标识网络中的一个进程就要使用一个如下的二元组:

(IP 地址, 端口号)

这个二元组可以看作是网络进程地址,也就是 1.1.4 节的那个小程序需要向用户获取的参数。本书后面几乎所有的实例软件,都要先由用户输入这个参数,然后将参数传递给某个特定的网络编程接口,再由操作系统自动调用 TCP/IP 核心协议,从而完成相应的网络功能。这个网络编程接口可以是微软 MFC 类库中已经提供的类或函数,也可以是由用户直接调用 Windows Socket API (Winsock API),其实 MFC 类库中那些用于网络编程的类也是封装的 Winsock API,因此归根到底 Windows 下的网络编程接口就一个,即 Winsock API,它是 Socket 呈现给程序员的编程界面。Socket 又叫套接字,可以看成是两个网络应用程序进行通信时,各自通信连接中的一个端点,这个端点是一个逻辑上的概念。通信时其中的一个网络程序将要传输的一段信息写入它所在主机的 Socket 中,该 Socket 通过与网络接口卡相连的传输介质将这段信息发送到另外一台主机的 Socket 中,使这段信息能传送到其他程序中,如图 1.40 所示。

根据套接字工作原理来分析使用套接字进行通信的过程。在图 1.40 中,主机 A 上的网络应用程序 A 要发送数据时,首先通过调用数据发送函数,将要发送的一段信息写入其 Socket 中,Socket 中的内容通过主机 A 的网络管理软件将这段信息由主机 A 的网络接口卡发送到主机 B,主机 B 的网络接口卡接收到这段信息后,传送给主机 B 的网络管理软件,网络管理软件将这段信息保存在主机 B 的 Socket 中,然后程序 B 才能从 Socket 中读取并使用这段信息。

从以上的通信过程中可以看出,如果不考虑网络接口卡和传输介质等,网络通信的过程就是由数据的发送者将要发送的信息写入一个套接字,在通过中间环节传输到接收端的套接字中后,由接收端的应用程序将信息从套接字中取出。因此,两个应用程序之间的数据传输是通过套接字来完成的。套接字的本质是通信过程中所要使用的缓冲区及一些相关的数据结构。

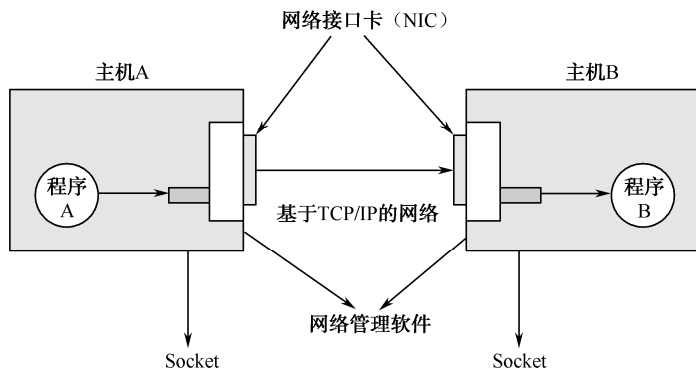


图 1.40 套接字工作原理

为了满足不同的通信程序对通信质量和性能的要求，一般的网络系统都提供了以下 3 种不同类型的套接字，以供用户在设计网络程序时根据需要来选择。

- 流式套接字（SOCK_STREAM）。它提供了一种可靠的、面向连接的双向数据传输服务。实现了数据的无差错、无重复地发送，内设流量控制，被传输的数据看作是无记录边界的字节流。在 TCP/IP 协议族中，使用 TCP 来实现字节流的传输，当用户要发送大批量的数据，或对数据的传输有较高的要求时使用流式套接字。
- 数据报套接字（SOCK_DGRAM）。它提供了一种无连接、不可靠的双向数据传输服务。数据以独立的包形式被发送，并且保留了记录边界，不提供可靠性保证。数据在传输过程中可能会丢失或重复，并且不能保证在接收端数据按发送顺序接收。在 TCP/IP 协议族中，使用 UDP（Winsock 2 也支持其他的协议）来实现数据报套接字。
- 原始套接字（SOCK_RAW）。该套接字允许对较低层协议（如 IP 或 ICMP）进行直接访问。在直接对 TCP/IP 核心协议编程时要用到这种套接字，但作为网络编程的基础教程，本书暂不涉及。

可以说，网络上所用应用程序的底层通信都是基于以上 3 种套接字进行的，套接字在编程时对于用户来说是可见的，如图 1.41 所示就是网络上两个 Windows 应用程序通过套接字通信的过程。

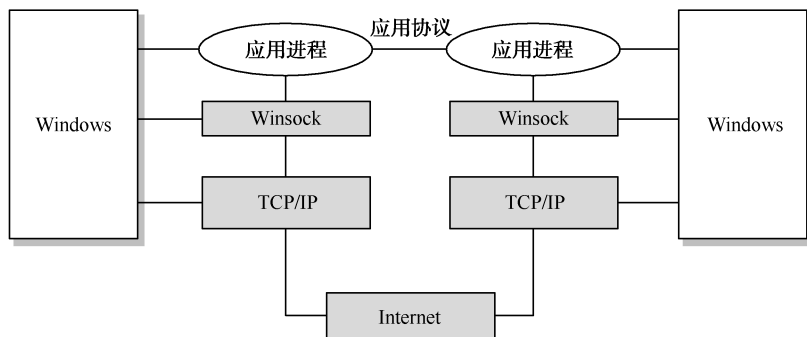


图 1.41 Windows 程序通过套接字互相通信

显然，套接字 Winsock 屏蔽了下面 TCP/IP 协议栈的复杂性，使得在网络编程者看来，两个网络程序之间的通信只是它们各自所绑定的套接字之间的通信，这就使网络程序的编程模型变得十分简单和易于理解了。

1.2.4 本书编程的协议环境

Socket 最初是以 Windows API (Win32 API) 的形式提供的, 所以又称 Windows Socket API, 简称 Winsock API。随着面向对象编程方式逐渐成为软件开发的主流, 原来的 Windows API 接口越来越显得与面向对象思想格格不入了, 于是微软与时俱进地将 Windows API 重新进行了面向对象的封装, 将它们包装成一个个类供用户使用, 这些类的全体集合就是大家十分熟悉的 MFC。

这样, 原来的 Winsock API 也随之一起封装在 MFC 之中了, 又称 MFC Socket (本书第 2 章内容), 以区别于封装前的 Windows Socket。封装之后的套接字类更加满足流行的面向对象编程的需要, 但也存在缺陷, 其中最重要的一个缺陷就是, 它对用户屏蔽了下层 TCP 和 UDP 的差别。关于这种差别本书第 3 章将会说明。在很多情况下, 不少程序员仍然习惯于原来直接调用 API 的编程方式, 因为那样编程更加灵活。这样, 程序员在编写网络程序时就有两个选择——使用 MFC Socket 类或者直接调用 Winsock API——这两种方式是等价的, 很难说到底孰优孰劣。MFC Socket 类与 Winsock API 编程接口其实是同一个层次上的接口, 统称为 Socket API。它们共同构成了程序员直接可见的网络应用编程界面, 如图 1.42 所示, 而这个界面之下的一切则隐藏在神秘的操作系统内核之中, 对应用开发人员是透明的。

由图 1.42 中可见, 传输层的 TCP 协议被 Winsock 封装为 SOCK_STREAM (流式套接口), UDP 则被封装为 SOCK_DGRAM (数据报套接口)。本书第 4 章开发的聊天室软件是直接使用 MFC Socket 类的 (⑨)。应用层几个比较常用的协议 (HTTP/FTP/SMTP/POP3/IMAP 等) 都是通过流式套接口 SOCK_STREAM 使用 TCP 的功能。我们生活中常用的即时通信软件 (QQ、MSN、Skype 等) 都是基于一类称为 OICQ 的协议 (⑩), OICQ 是无连接的应用, 使用传输层 UDP 的服务, 它通过 SOCK_DGRAM (数据报套接口) 使用 UDP。

虽然应用层本身是不稳定的, 但其中仍然有不少经常使用的比较通用和基础的协议, 如图 1.42 中画出的这几个。但这些协议的标准本身还是比较复杂的, 为了给应用开发程序员提供最大可能的方便, 很多著名软件厂商都对这些协议进行了再封装, 用类库或函数库的形式先将这些协议实现好再提供接口供程序员使用。微软将 HTTP 和 FTP 也封装到 MFC 中成为 WinInet 类, 将邮件应用类协议 (SMTP/POP3/IMAP) 用库函数实现并以 MAPI 接口的形式供用户直接调用, 如图 1.42 应用层中虚线框所示。

在应用层之上是各种网络应用程序软件。

本书第 5 章的浏览器间接通过 MFC CHtmlView 类 (①) 和 WinInet 类 (②) 使用 HTTP。

第 6 章 FTP 上传下载器通过 WinInet 类 (④) 使用 FTP。

第 7 章电子邮件客户端程序则通过调用 MAPI 接口 (⑦) 来使用 POP3、SMTP。当然 Outlook Express 是直接实现 SMTP、POP3 这些邮件类协议收发邮件的 (⑥), 新浪、网易这些运营商的邮件服务系统也是直接实现的邮件应用协议 (⑧)。

一般来说, 服务器直接实现应用协议 (③、⑤) 为广大网络用户提供服务, 本书第 5 章和第 6 章也会讲到服务器的网络编程。

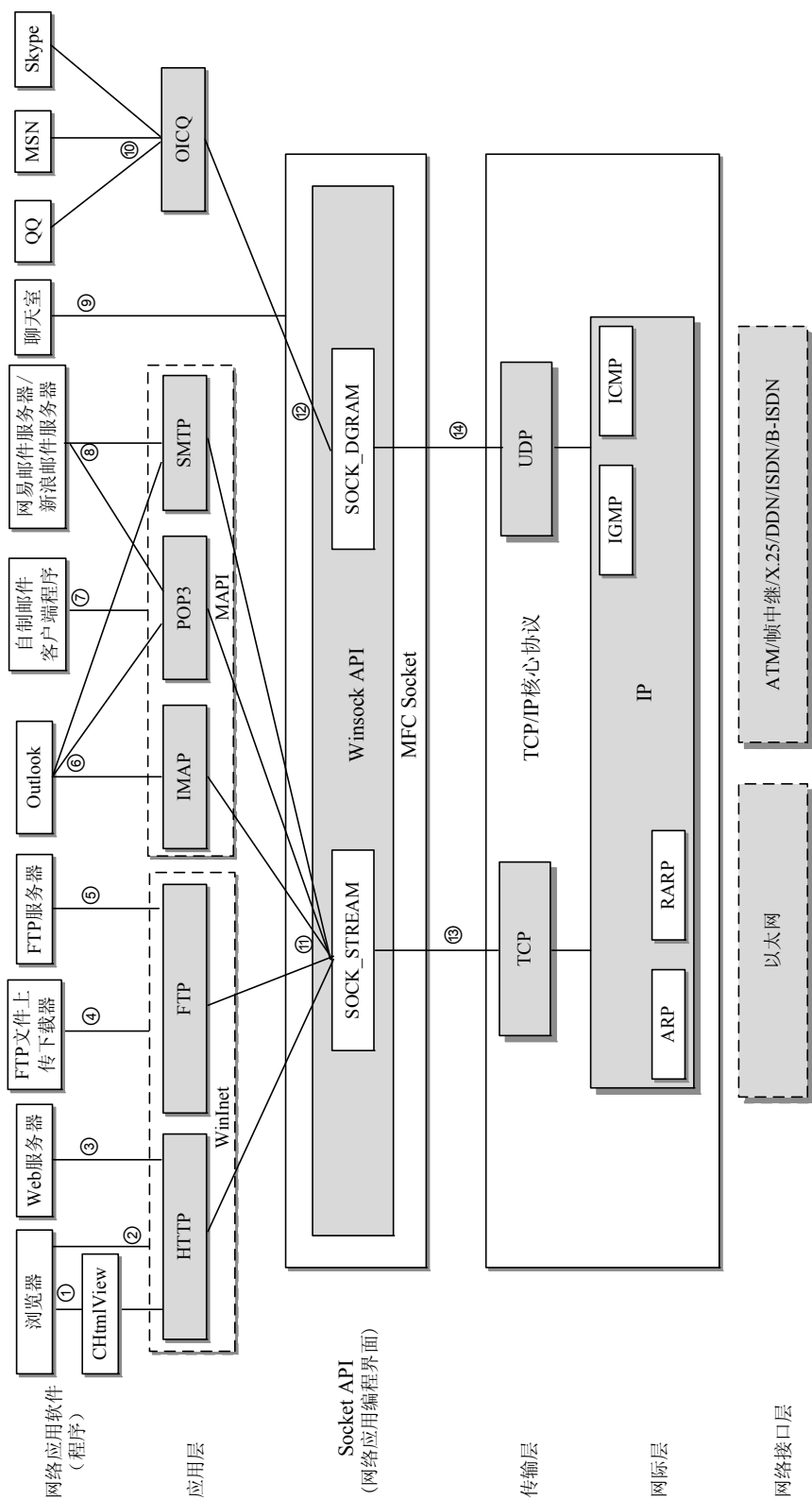


图 1.42 TCP/IP 体系

MFC Socket 编程

2.1 MFC 及其 Socket 类

2.1.1 MFC 简介

MFC (Microsoft Foundation Classes, 微软基础类库) 是一种应用框架 (Application Framework), 它随微软 Visual C++ 一起集成发布, 提供一组通用的可重用的类库供开发人员使用, 目前最新版本为 10.0 (截至 2011 年 3 月)。

Microsoft 在 Windows API 的基础上封装了一组 C++ 类, 并以 C++ 库的形式构建了面向对象的框架, 程序开发人员可以利用这一框架创建 Windows 应用程序。使用 MFC 的最大优点在于: 它完成了在使用 SDK 编程时所有最难做的工作, MFC 中包含了成千上万行正确、优化和功能强大的 Windows 代码, 所调用的很多成员函数可以完成用户自己很难完成的工作。从这点上讲, MFC 极大地减轻了 Windows 编程的负担。

MFC 是很庞大的, 以 Visual C++ 2008 所带的 MFC 9.0 为例, 它包含了多达 237 个类、4 个结构和 4 个接口。如图 2.1 所示, MFC 按照 C++ 类的层次结构组织在一起, 高层类提供一般功能, 而低层类实现更具体的行为。每一个低层类都是从高层类中派生出来的, 因此继承了高层类的行为。

该层次结构可分为如下几种不同的类型:

- 应用程序框架。
- 图形绘制的绘制对象。
- 文件服务。
- 异常处理。
- 结构, 如 Lists、Arrays 和 Maps。
- Internet 服务。
- OLE 2。
- 数据库。
- 通用类。

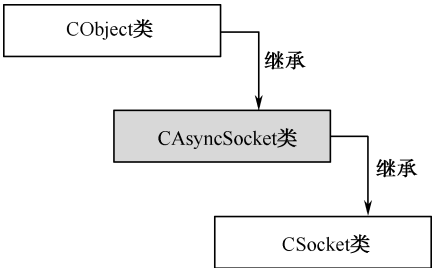


图 2.2 CAsyncSocket 类的继承位置

CAsyncSocket 类利用了 Windows 的消息机制,可以方便地利用回调的便利通知网络事件(后面会介绍到),既屏蔽了底层 Winsock 接口的复杂性,又不失灵活性,是学习 MFC Socket 编程最佳的入门选择。

CAsyncSocket 类的成员函数如表 2.1 所示。

表 2.1 CAsyncSocket 类成员函数

函数名称	功能描述
CAsyncSocket	构造函数
Create	创建一个套接字
Attach	将一个套接字句柄关联到一个类对象上
Detach	解除套接字句柄和类对象的关联
FromHandle	通过一个套接字句柄得到类对象指针
GetLastError	取得最后一次错误返回函数的错误码
GetPeerName	取得一个已经建立了连接的套接字的远端主机地址
GetSockName	取得套接字的本地名称
GetSockOpt	取得套接字的选项值
SetSockOpt	设置套接字的选项值
Accept	接受连接请求
AsyncSelect	选择感兴趣的事件
Bind	将套接字和本地端口绑定
Close	关闭套接字
Connect	发出连接请求
IOctl	控制套接字的模式
Listen	设置套接字处于监听状态 (等待客户端连接)
Receive	接收数据
ReceiveFrom	接收数据,并得到数据发送方的地址信息
Send	通过建立连接的套接字发送数据
SendTo	向一个指定地址发送数据
ShutDown	禁止套接字的某些操作
OnAccept	当收到建立连接请求时调用的处理函数
OnClose	当连接的另一端套接字关闭时调用的处理函数

续表

函数名称	功能描述
OnConnect	当连接函数返回时的处理函数
OnReceive	当有新的数据到达时调用的处理函数
OnSend	当有数据要发送时调用的处理函数

表 2.1 中, 加黑的为常用的 MFC Socket 函数。

可能一下子看到这么多的函数会觉得不知所措, 很难理解其中绝大多数函数的功能描述, 不知道如何去用, 也记不住这么多函数的名称。其实这些都不用记, 在后面的小节中我们将通过具体的实例来学习这些函数的用法, 并结合核心代码的介绍, 相信一定会令读者豁然开朗的。

2. CSocket 类及其相关类

(1) CSocket 类。

CSocket 类是从 CAsyncSocket 派生而来的, 它继承了 CAsyncSocket 对 Windows Socket API 的封装。与 CAsyncSocket 对象相比, CSocket 对象代表了 Windows Socket API 的更高一级的抽象化。通常 CSocket 类与 CSocketFile 类和 CArchive 类一起来管理对数据的发送和接收。一个 CSocket 对象也支持打包, 这对于 CArchive 的同步操作来说是必要的。打包操作函数, 如 Receive、Send、ReceiveFrom、SendTo 和 Accept (均是从 CAsyncSocket 继承来的), 都不返回一个 CSocket 对象中的 WSAEWOULDBLOCK 错误, 而是等待直到操作完成。另外, 当这些函数中的某一个被阻塞时, 如果调用了 CancelBlockingCall, 则原来的调用将因为 WSAEINTR 错误而终止。

(2) CSocket 与 CArchive、CSocketFile 类的配合使用。

用 CSocket 类编写网络程序, 既可以使用如 CAsyncSocket 类网络程序一样的 Send 和 Receive 函数来收发信息, 也可以与 CSocketFile 类和 CArchive 类一起来管理对数据的发送和接收。

一个 CSocketFile 对象是一个用来通过 Windows Socket 在网络中发送和接收数据的 CFile 对象。为了达到这个目的, 可以将 CSocketFile 对象与一个 CSocket 对象连接, 也可以将 CSocketFile 对象与一个 CArchive 对象连接, 以便使用 MFC 序列化来简化发送和接收数据。要序列化(发送)数据, 可以把它们插入到档案中, 该档案调用 CSocketFile 成员函数来把数据写到 CSocket 对象中。如果不序列化(接收)数据, 也可以从档案中直接提取它们, 这将导致该档案须调用 CSocketFile 成员函数来从 CSocket 对象中读取数据。除了用此种描述的方法使用 CSocketFile 外, 也可以将它作为一个标准文件对象, 就如同使用它的基类 CFile 一样。可以将 CSocketFile 与其他任何基于档案的 MFC 序列化函数一起使用。

采用 CSocket 与 CArchive、CSocketFile 类配合的方法发送接收、数据就如直接从本地文件中读写数据一样方便。它的基本编程模式如下。

首先, 构造一个 CSocket 对象, 并调用 Create 函数创建一个 Socket 对象 (SOCK_STREAM 类型)。

其次, 如果是客户端程序, 则调用 Connect 连接到远地主机; 如果是服务器程序, 则先调用 Listen 监听 Socket 端口, 收到连接请求后再调用 Accept 接收请求。

最后, 创建一个和 CSocket 对象关联的 CSocketFile 对象, 创建一个和 CSocketFile 对象关联的 CArchive 对象, 指定 CArchive 对象是用于读或者写。如果既要读又要写, 则创建两个 CArchive 对象。

创建工作完成之后, 使用 CArchive 对象在客户端和服务端之间传送数据。

使用完毕, 销毁 CArchive 对象、CSocketFile 对象及 CSocket 对象。

2.2 C/S 模式下网络程序的 Socket 通信实例

2.2.1 客户端—服务器方式（C/S 模式）

1. 网络程序的通用体系结构

当今的互联网是以高性能微机服务器（集群）为中心的网络，形成了客户端—服务器应用方式。大家平时上网访问网站或玩网游，使用的都是客户端—服务器方式。

此处，客户端（Client）和服务器（Server）是指通信中所涉及的两个应用进程。客户端—服务器方式所描述的是进程之间服务和被服务的关系。在图 2.3 中，主机 A 运行客户端程序而主机 B 运行服务器程序。在这种情况下，A 是客户端而 B 是服务器。客户端 A 向服务器 B 发出请求服务，而服务器 B 向客户端 A 提供服务。

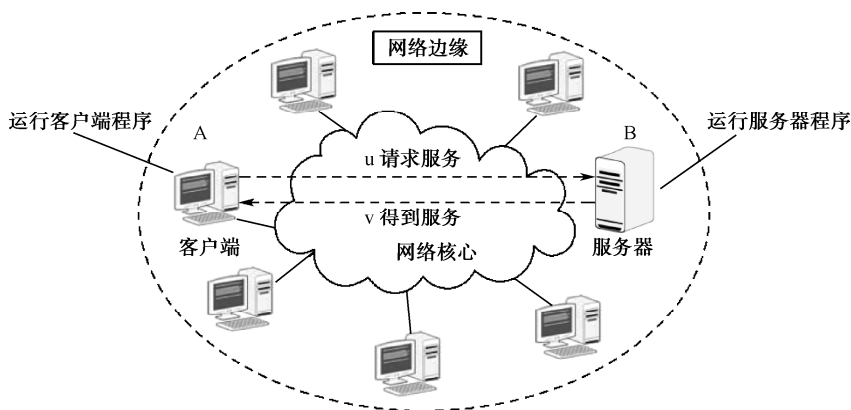


图 2.3 客户端—服务器方式

客户端是服务请求方，服务器是服务提供方，它们都要使用网络核心部分所提供的服务。在实际应用中，客户端程序和服务器程序通常还具有以下一些主要特点。

客户端程序：

- 被用户调用后运行，在通信时主动向远地服务器发起通信（请求服务）。因此，它必须知道服务器程序的地址。
- 不需要特殊的硬件和很复杂的操作系统。

服务器程序：

- 专门用来提供某种服务的程序，可同时处理多个远地或本地客户端的请求。

系统启动后即自动调用并一直不断地运行，被动地等待并接收来自各地客户端的通信请求。因此，服务器程序不需要知道客户端程序的地址。

- 一般需要强大的硬件和高级的操作系统支持。

客户端与服务器的通信关系建立后，通信可以是双向的，客户端和服务器都可以发送和接收数据。

客户端—服务器方式（C/S 模式）是网络程序的通用体系结构。

2. 最简单的 Socket 通信流程

那么，客户端与服务器之间是如何实现通信的呢？当我们打电话时，电话机的振铃声使被叫

用户知道现在有一个电话呼叫。计算机通信的对象是应用层中的应用进程，它们交互的过程与电话振铃呼叫差不多。

这里先来考虑一个只有客户方向服务方发信息的单向通信，并且也只有客户方会主动提出断开连接的最简单的情形（相反过程的原理是一样的），双方 Socket 之间的关系如图 2.4 所示。

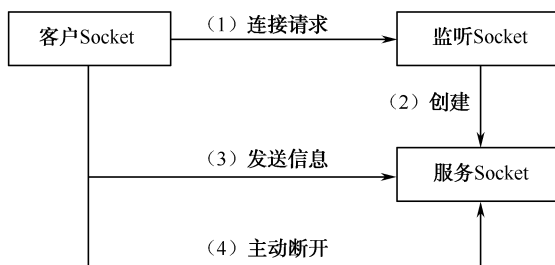


图 2.4 最简单的单向通信

从图 2.4 中可以看出，整个通信过程分为 4 步。

（1）客户端向服务器发出请求，要与服务器建立连接。

（2）服务器接收客户端的请求并创建一个新的 Socket，用于接收客户端将发来的信息（为了尽可能简单起见，这里假定服务器总是无条件地接收客户端的连接请求）。

（3）客户端向服务器发送信息（可以多次发送信息，只要连接不断开）。

（4）客户端不想与服务器通信了，主动提出断开连接（服务器当然也无条件同意）。

由上面这个过程很容易得出对应的 Socket 通信流程，如图 2.5 所示。

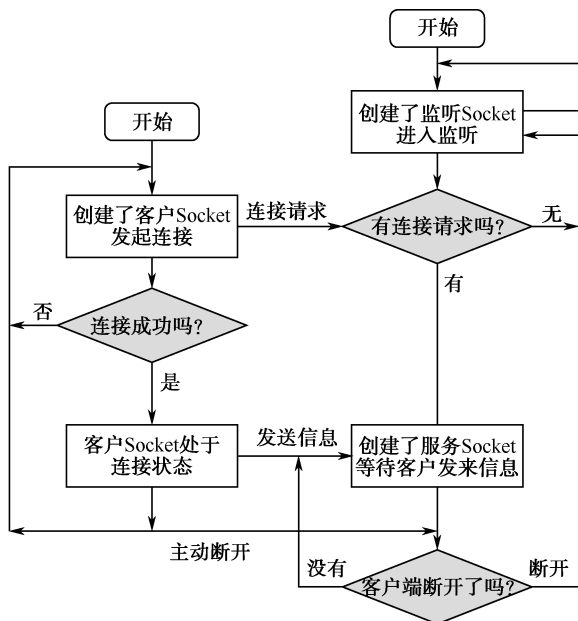


图 2.5 最简单的 Socket 通信流程

虽然我们假定服务器总是无条件地接收客户端的连接请求，但网络发生意外故障的可能并不能完全排除，而且服务器有可能根本就没有开机，或是客户端填错了服务器的 IP 和端口，这些可能性都是存在的。因此客户端发出请求后还要确认连接是否成功，这一点是必要的，即使对于最简单的通信过程。

下面就以这个最简单的通信过程为蓝本，入手编写本书第一个 Socket 进程通信的实例程序。

2.2.2 CAsyncSocket 类编程基础

第一个 Socket 程序采用 CAsyncSocket 类编写，为方便读者学习和顺利入门，采用“按操作步骤”详细描述讲述的方法，在读者熟悉了 MFC 通用的工程结构和编程流程后，再从代码功能和程序工作原理的角度介绍实例。

1. 对象分析

要实现 2.2.1 节分析的 Socket 通信流程，一共需要三个套接字对象：客户端一个（我们称为“客户 Socket”），服务器两个（一个用于监听，称为“监听 Socket”；另一个用于接收客户端发来的信息，称为“服务 Socket”）。这三个套接字对象对应三个 Socket 类，都继承自 CAsyncSocket，分别给它们取名，如表 2.2 所示（表中的类名也是后面程序中要使用的）。

表 2.2 Socket 类对象分析

Socket 对象	类 名
客户 Socket	CClientSocket
监听 Socket	CListenSocket
服务 Socket	CServerSocket

2. 创建工程和套接字对象

首先创建客户端工程。打开 Visual Studio 2010 环境，建立一个新的基于对话框的 MFC 项目，项目名称为 ChatClient，向导的前几页设置同第 1 章程序，直到设置程序“高级功能”页，因为接下来要编写的是网络程序，使用套接字，因此必须勾选“Windows 套接字”复选框，如图 2.6 所示。

后面步骤中一律采用默认设置，直到完成工程的创建。工程创建后，由前述分析，还需要创建类名为 CClientSocket 的客户 Socket 对象，于是给工程添加类，选择菜单命令“项目”→“添加类”，如图 2.7 所示。



图 2.6 在工程中使用 Windows 套接字

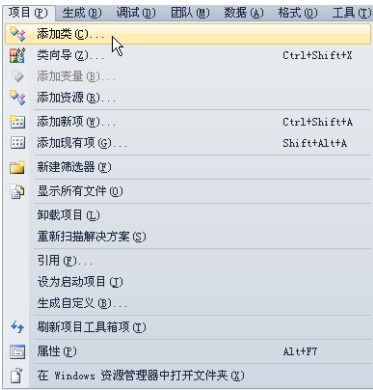


图 2.7 在项目中添加类

在弹出的“添加类”对话框中选择“MFC 类”项，单击“添加”按钮（如图 2.8 所示）。

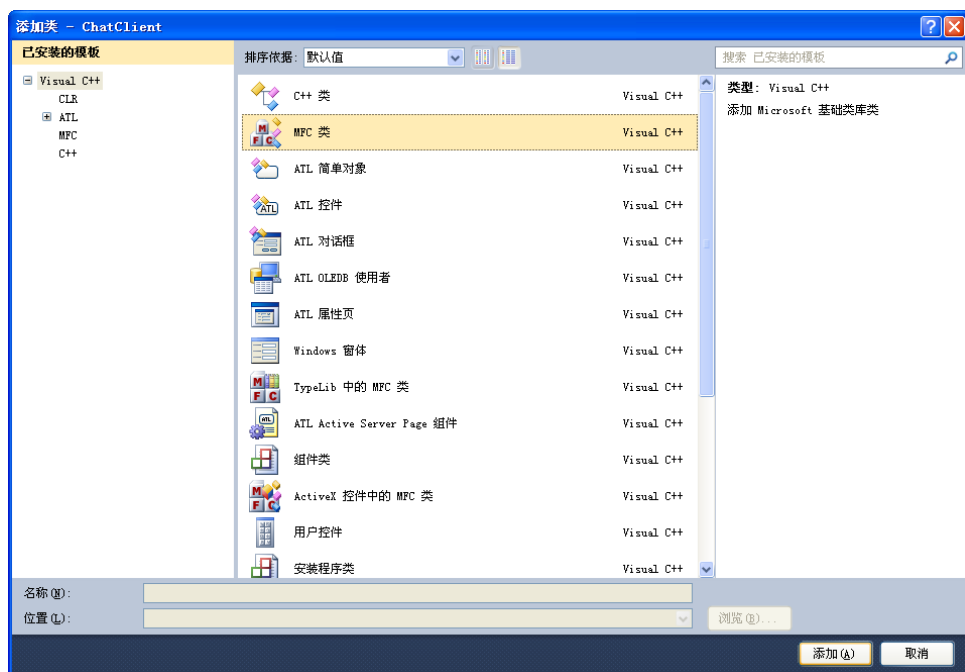


图 2.8 添加 MFC 类

在“MFC 添加类向导”对话框中输入类名“CClientSocket”，由于要使用 CAsyncSocket 套接字类编写程序，所以选择基类为“CAsyncSocket”，同时可以看出，向导还将自动为这个添加的类生成名为“ClientSocket.h”和“ClientSocket.cpp”的头文件和源文件，如图 2.9 所示。

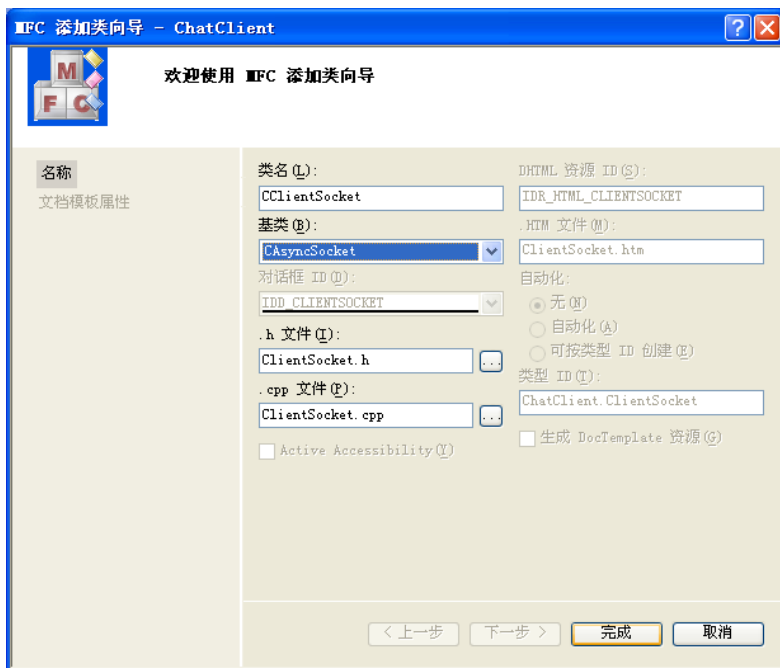


图 2.9 给添加的类命名

单击“完成”按钮，就会在类视图中看到刚刚添加的类 CClientSocket（如图 2.10 所示）。

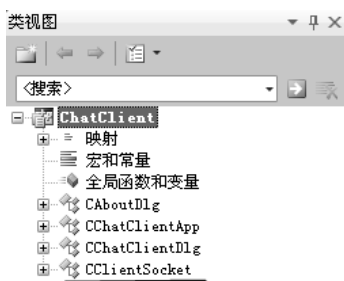


图 2.10 添加类成功

现在客户端工程已经创建好，接下来创建服务器工程。与客户端类似，建立一个名称为 ChatServer 的 MFC 项目，在向导的“高级功能”页，勾选“Windows 套接字”复选框。采用与客户端一样的添加类的方法，为服务器创建两个 Socket 对象——“CListenSocket”（用于监听）和“CServerSocket”（用于通信）。

至此，客户端和服务器的工程都创建完毕。

3. 理清程序文件的组织结构

下面来整理上述一系列操作中所生成的全部程序文件。

1) 客户端程序文件

打开已经创建好的客户端工程，如图 2.11 所示，在解决方案资源管理器中可以看到工程所有的程序文件。

全部程序文件分为三类，如下所示。

第一类：.h 后缀的头文件，用来集中声明定义程序中用到的类、变量、函数、宏等。

第二类：.cpp 后缀的源文件，这是程序代码的主体，集中存放程序的源代码。

第三类：资源文件，存放程序中用到的资源，如图标、图像、音频、视频等，它们虽不是源代码，但对美化程序的 GUI 外观、实现多媒体功能必不可少。



图 2.11 工程中的所有程序文件

Visual Studio 2010 平台已经自动创建了这些文件，并生成了作为应用程序框架的大量源代码，还将这些程序文件分门别类地组织起来。

双击“解决方案资源管理器”工程文件目录树最末端的 ReadMe.txt 文件，可以在右边的文件内容显示区看到这个文本的内容，里面分别介绍了每个程序文件的概况。

其中，有几个关键文件是编程时要用到的，如图 2.11 中带下画线的文件，它们是以 ChatClient、ChatClientDlg 和 ClientSocket 为名的.h 头文件及对应的.cpp 源文件，其中 ClientSocket 是用户创建的套接字类实现所对应的程序文件，而 ChatClient 和 ChatClientDlg 就是前面介绍的 VC 在用户一开始创建工程时都会默认生成的两个类对应的源代码文件。“CXXXXApp”代表整个应用程序，凡是在程序的一个模块中引用另一个模块的对象或变量都得通过它；另一个是“CXXXXDlg”，代表该程序的界面，用户界面上的那些按钮、文本框、列表框等 Windows 控件都归它管。

2) 服务器的源文件

同理，服务器也对应这几种程序文件，打开服务器工程，可以看到它们（如图 2.12 所示）。



图 2.12 服务器工程文件

只不过服务器由于要使用两个 Socket（一个用于监听，另一个用于和客户端通信），所以对两个 Socket 类生成了两个程序文件 ListenSocket 和 ServerSocket，它们分别有各自的源文件和头文件。

4. 用头文件和类对象将程序源文件联成有机整体

至此已经创建了程序所需要的全部文件，但要建立一个可以运行的原型程序，这些代码文件还必须以一定的方式联系起来，组成一个有机的整体。例如，虽然已经创建了各个 Socket 对象并且也有了它们各自对应的程序文件，但这些文件仍然是孤立的，相互之间的代码无法访问。要使它们能联系在一起，使控制程序的主文件能够自如地操纵 Socket，就必须通过头文件声明和创建类对象成员变量将各个类联系起来。

要使客户端程序能够创建和控制本地的 Socket，需在在客户端工程界面控制模块的头文件 ChatClientDlg.h 中添加如下两行代码：

```
#include "ClientSocket.h"           //使主界面程序能够访问 Socket 类的代码文件
CClientSocket m_ClientSocket;      //为了后面与服务器通信而定义的 Socket 成员变量
```

以上两行代码的添加位置见图 2.13 中“//ADD”记号之间标出的部分。

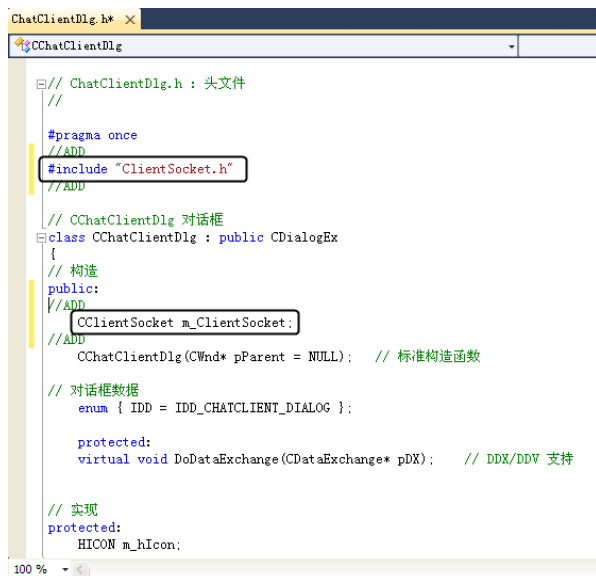


图 2.13 ChatClientDlg.h 中添加的代码

反过来, 要使 Socket 能够访问到主界面上的控件, 以便能够将自己的状况随时反映给主程序并在主界面上显示出来, 也需要在 Socket 源文件 ClientSocket.cpp 中添加头文件声明:

```
#include "ChatClientDlg.h"
```

同理, 要使服务器程序能够创建和控制本地的 Socket, 也要在服务器工程界面控制模块的头文件 ChatServerDlg.h 中添加如下代码:

```

#include "ListenSocket.h"           //使主界面程序能够访问监听 Socket 类的代码文件
#include "ServerSocket.h"          //使主界面程序能够访问服务 Socket 类的代码文件
CServerSocket m_ServerSocket;      //为了后面与客户端通信而定义的 Socket 成员变量
CListenSocket m_ListenSocket;      //为了监听客户端的连接请求而定义的 Socket 成员变量

```

以上四行代码的添加位置见图 2.14 中“//ADD”记号之间标出的部分。

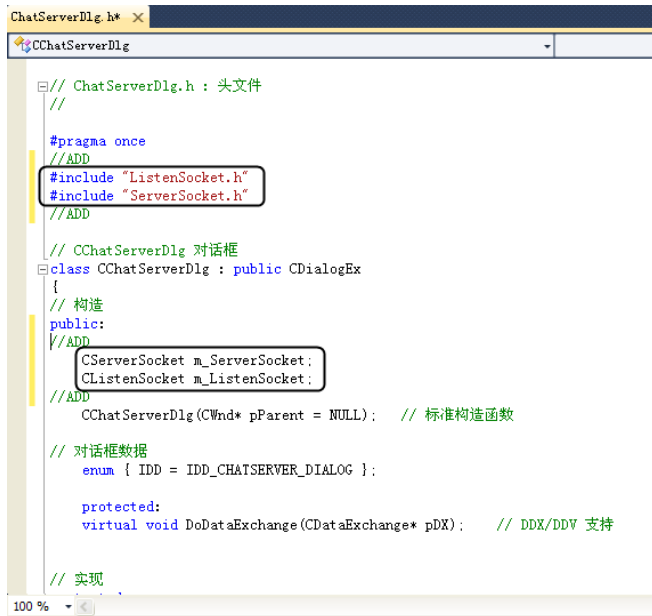


图 2.14 ChatServerDlg.h 中添加的代码

同样，分别在监听套接字和服务套接字的源文件中添加声明。

在 ListenSocket.cpp 中添加如下代码：

```
#include "ChatServerDlg.h"
```

在 ServerSocket.cpp 中添加如下代码：

```
#include "ChatServerDlg.h"
```

现在整个程序的主控界面已经与各自的 Socket 联为一体，可以互相操作了。本书的编程观：程序不仅仅只是源代码，而是由各种职能的源代码文件有机联系在一起组成的一个有生命的整体。

5. 简单布置界面

先在界面上放置几个必需的控件，以使读者对程序的基本功能一目了然。

在客户端“资源视图”展开的目录树下双击 Dialog 文件夹下的第二个项目，转到用户界面设计工作区（如图 2.15 所示），先将设计页上的默认控件（一个静态文字标签和“确定”按钮）删除，留下“取消”按钮做程序的“退出”按钮之用。

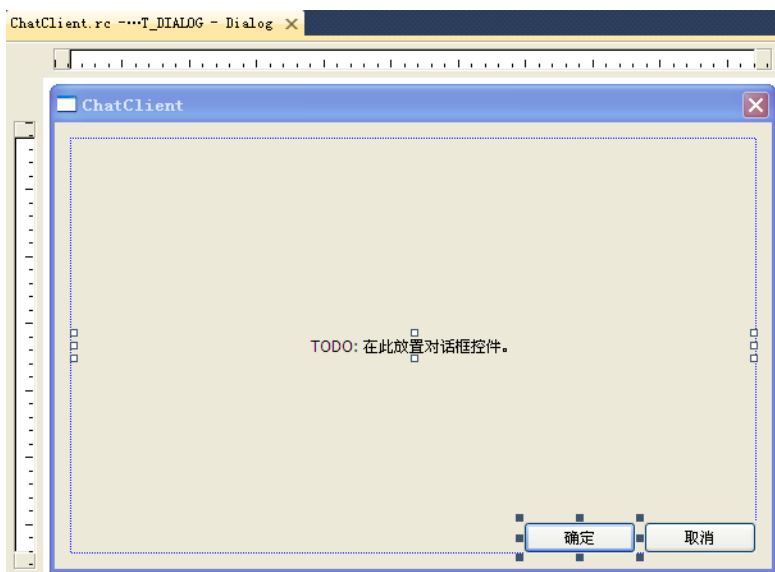


图 2.15 初始的用户界面设计工作区

在完成的客户端简化界面（如图 2.16 所示）上，包括 IP 地址控件、列表框各 1 个，文本框两个（一个用于接收用户输入端口号，另一个用于接收用户输入的待发送文本信息），4 个按钮（其中“连接”、“断开”、“发送”3 个按钮是我们添加的，“退出”按钮由原来的“取消”按钮改变 Caption 属性得到）。

用第 1 章介绍的方法给各控件关联变量，为 IP 地址控件关联变量 ServerIP（Control 型），为端口文本框控件添加变量关联 sPort（Value 类的 int 型），为编辑发送信息文本框控件添加变量关联 m_sWords（Value 类的 CString 型），为列表框添加变量关联 m_ListWords。

设计服务器的界面如图 2.17 所示，同样，为 IP 地址控件关联变量 ServerIP，为文本框控件关联 int 型变量 sPort，为列表框关联变量 m_ListWords。

6. 添加核心代码

到目前为止，已经有有了一个程序框架，接下来添加核心源代码。先来回顾一下前面设计的那个最简单的 Socket 通信流程，如图 2.18 所示。

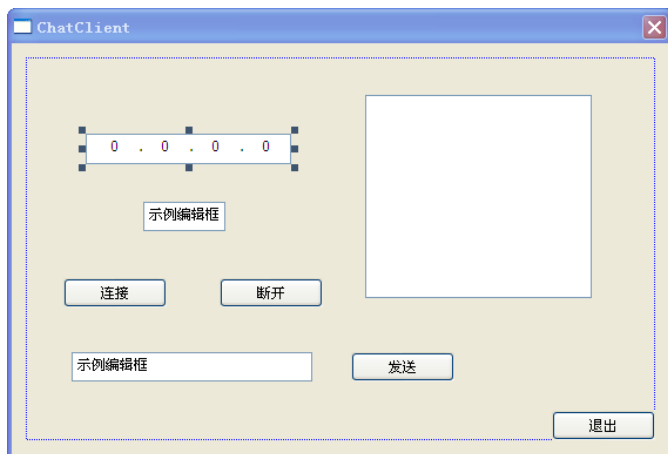


图 2.16 客户端简化界面



图 2.17 服务器简化界面

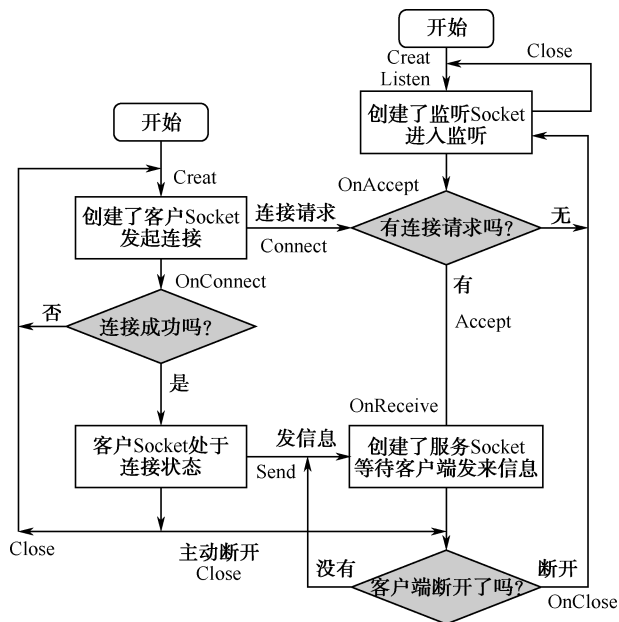


图 2.18 本例程序的 Socket 通信流程

在上面的流程图中，还标出了每一个环节要用到的 Socket 函数，这些函数的功能描述在前面的 CAsyncSocket 类成员函数表（表 2.1）中都有，读者可以对照着去查。

按照调用方式，可将这些函数分为两大类。

一类是由程序主动调用的，这类函数有 **Create**、**Close**、**Connect**、**Send**、**Listen**、**Accept**，在源代码中很容易找到调用它们的程序语句；另一类是由系统的某个网络事件触发的，如 **OnConnect**、**OnAccept**、**OnReceive**、**OnClose**，网络事件的触发时机和发生顺序不确定，故不能事先编写流程，而是在程序运行期间视实际情况触发，由系统自动调用，因此源代码中找不到调用这些函数的语句。这也是 Windows 操作系统广泛采用的一种技术——消息触发机制，它大大增强了程序处理突发网络事件的能力，增加了互动性与灵活性，给软件注入了强大的生机与活力。

这里简要介绍：

- **OnConnect**——用于客户端发出连接请求后执行相应的处理；
- **OnAccept**——在服务器接收连接请求时触发；
- **OnReceive**——在接收到对方发来的信息时触发；
- **OnClose**——在通信一方突然断开连接时执行善后处理。

显然，“客户端发出连接请求”、“服务器接收连接请求”、“收到对方发来的信息”、“通信一方突然断开连接”这些事都是通信双方用户在运行程序的过程中临时决定做出的操作或是发生时间不确定的网络事件（比如何时能收到对方发来的信息要视网络的拥塞延时状况而定），可见“消息触发机制”是一种多么重要的编程机制。

从图 2.18 已经标注了所用函数的通信流程图，可以进一步得出如图 2.19 所示的源代码组织框图。

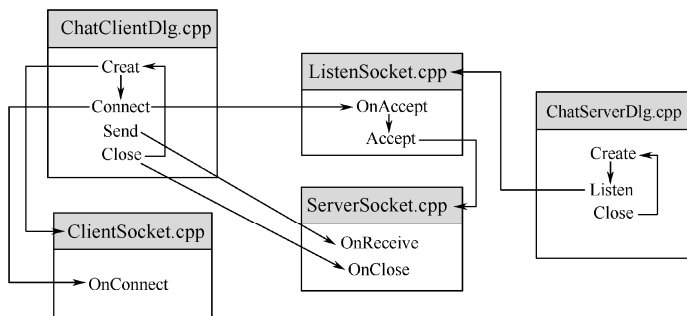


图 2.19 源代码组织框图

图 2.19 中已经根据通信流程给出了每个源文件中函数的布局，并用箭头标示了它们之间的关系，可谓一目了然。

7. 源代码完全剖析

现在读者可以对照图 2.19 来看下面的源代码剖析，为了使大家能更好地看清楚 Socket 通信过程，我们在源码中对核心语句进行了醒目标注，以加黑字体突出显示。

1) 客户端源码

客户端用户首先主动发起连接，以下是“连接”按钮的事件过程，位于 ChatClientDlg.cpp 文件中：

```
//连接服务器
BYTE nField[4];
CString sIP;
```

```

UpdateData();
ServerIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
m_ClientSocket.Create(); //创建客户端 Socket
m_ClientSocket.Connect(sIP,sPort); //发起连接请求

```

Create()函数用来创建和初始化套接字，具体过程为：构造套接字对象 m_ClientSocket 后调用 Create()成员函数创建 Socket 句柄，并调用 Bind()成员函数将其与指定的地址绑定。Create()函数原型为：

```

BOOL Create(UINT nSocketport = 0, int nSocketType = SOCK_STREAM, long lEvent =
FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT | FD_CLOSE,
LPCTSTR lpszSocketaddress = NULL);

```

其中，参数 nSocketPort 是与套接字一起使用的周知端口号，默认为 0，表示由 WinSock 自动选择端口号；参数 nSocketType 指定要创建的套接字类型，默认为流式套接字；参数 lEvent 用来指定感兴趣的网络事件的掩码位；参数 lpszSocketaddress 用于指定套接字的网络地址。若函数调用成功，则返回非零值，否则返回零，可以调用 GetLastError()函数获得具体的错误信息。在这里因为创建的是客户端 Socket，所以一切参数都采用默认值。客户端与服务器不同，它不接收其他客户的连接，因此也就不需要有固定的端口和 IP 地址，这些都可以临时分配指定，所以这里调用 Create()函数无须指明参数。

Connect()函数用于未连接的数据流或者数据报套接字建立连接。其函数原型为：

```

BOOL Connect(LPCTSTR lpszHostAddress, UINT nHostPort);
BOOL Connect(const SOCKADDR*lpSockAddr, int nSockAddrLen);

```

其中，第一种形式，参数 lpszHostAddress 为要连接的服务器网络地址，参数 nHostPort 用于指定套接字应用程序使用的端口号；第二种形式，lpSockAddr 是指向 SOCKADDR 结构的指针，参数 nSockAddrLen 是 lpSockAddr 指针中地址的字节长度。如果成功则返回非零值，否则返回零，可以调用 GetLastError()函数获得具体的错误信息。上面程序中使用的是第一种形式，即指明要连接的服务器 IP 地址和端口号，发起连接请求。第二种形式常在直接的 Winsock API 编程中使用，在本书的实例中，读者会经常看到这样的用法。

客户端也可以随时主动断开通信连接，下面是“断开”按钮的事件过程：

```

//断开与服务器的连接
m_ClientSocket.Close(); //关闭客户端 Socket
m_ListWords.AddString("从服务器断开");

```

Close()函数用来关闭套接字并释放 Socket 描述符，其函数原型为：

```
virtual void Close();
```

套接字对象的析构函数会自动调用此成员函数。

客户端可以向服务器发送信息，“发送”按钮的事件过程为：

```

//向服务器发信息
UpdateData();
m_ClientSocket.Send(m_sWords,m_sWords.GetLength()); //发信息
m_ListWords.AddString("发送: " + m_sWords);
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);

```

Send()函数通过数据报或者数据流向对方套接字发送数据，其函数原型为：

```
virtual int Send(const void* lpBuf, int nBufLen, int nFlags = 0);
```

其中，参数 lpBuf 为要发送的数据的缓冲区地址；参数 nBufLen 为 lpBuf 缓冲区的字节长

度；参数 nFlags 为指定函数的调用标志。本例使用发送信息文本框关联的变量 m_sWords 作为缓冲区。

以上都是前面所说的第一类由用户通过程序界面控件按钮主动调用的函数，它们一般都位于程序对话框界面类的源文件（本例中是 ChatClientDlg.cpp）中。接下来要编写的是第二类（即网络事件响应）函数，源码中无法找到调用它们的语句，它们是由系统自动触发的，通过如图 2.20 所示的属性窗口来添加这类函数的代码。

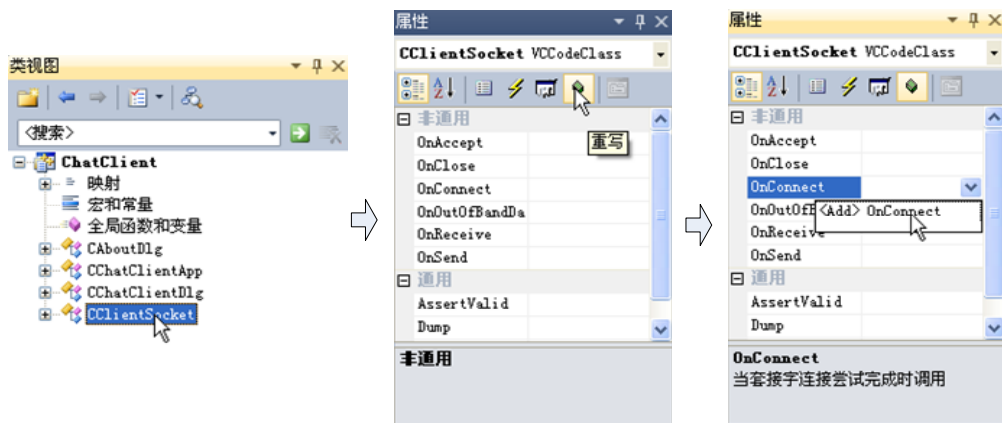


图 2.20 通过属性窗口添加函数代码

在类视图中选中 CClientSocket，在界面右下角的属性窗口中单击“重写”按钮，就可以为该 Socket 类编写被动响应网络事件的函数，它们在系统中已经有了默认的处理方式，用户自己编写的代码实际上是重载了原来函数的实现。

选择 OnConnect()函数，为其添加代码，系统将该函数的代码自动置于 ClientSocket.cpp 文件中。

OnConnect()函数代码如下：

```
//确认客户端是否成功连接到服务器
if(nErrorCode)
{
    AfxMessageBox("连接失败，请您重试！");
    return;
}
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ListWords.AddString("连接服务器成功！");
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ListWords.SetTopIndex(
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ListWords.GetCount()- 1);
```

2) 服务器源码 (ChatServerDlg.cpp 文件中)

“开始监听”按钮的事件过程如下：

```
//监听开始，服务器等待连接请求的到来
BYTE nFild[4];
CString sIP,sP;
UpdateData();
ServerIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
sP.Format("%d",sPort);
```

```

m_ListenSocket.Create(sPort,1,FD_ACCEPT,sIP);    //创建服务器监听 Socket
m_ListenSocket.Listen(1);                      //开始监听
m_ListWords.AddString("监听开始: ");
m_ListWords.AddString("地址" + sIP + " 端口" + sP);
m_ListWords.AddString("等待客户端连接……");

```

Listen()函数用于侦听连接请求, 原型为:

```

BOOL Listen(int nConnectionBacklog = 5);

```

其中, 参数 nConnectionBacklog 指定了服务器愿意接收的客户端数目, 默认为 5, 在此置为 1 表示暂时只允许 1 个客户端与之相连, 这是为了简化程序工作过程, 有兴趣的读者也可以连接多个客户端来试验一对多的进程通信。

“停止监听”按钮的事件过程如下:

```

//停止监听
m_ListenSocket.Close();                        //关闭服务器监听 Socket
m_ListWords.AddString("停止监听");

```

同理, 分别给监听 Socket 和服务 Socket 添加事件响应代码。添加的代码自动位于 ListenSocket.cpp 文件中。

OnAccept()函数的代码如下:

```

//接收客户端的连接请求
Accept(((CChatServerDlg* )(AfxGetApp()->m_pMainWnd))->m_ServerSocket); //接收连接请求
((CChatServerDlg* )(AfxGetApp()->m_pMainWnd))->m_ServerSocket.AsyncSelect(FD_READ|FD_CLOSE);
((CChatServerDlg* )(AfxGetApp()->m_pMainWnd))->m_ListWords.AddString(
"接收了一个客户端的连接请求");
((CChatServerDlg* )(AfxGetApp()->m_pMainWnd))->m_ListWords.SetTopIndex(
((CChatServerDlg* )(AfxGetApp()->m_pMainWnd))->m_ListWords.GetCount() - 1);

```

Accept()函数接收一个套接字的连接请求, 从连接请求队列中取出第一个连接, 创建一个与这个套接字具有相同属性的套接字, 并与参数 rConnectedSocket 相关联, 原始的套接字依然保持打开并且侦听的状态。函数原型为:

```

virtual BOOL Accept(CAsyncSocket& rConnectedSocket, SOCKADDR *lpSockAddr = NULL,
int* lpSockAddrLen = NULL);

```

其中, 参数 rConnectedSocket 是用来进行连接的新套接字的引用, 而这个新套接字就是由监听套接字动态创建的服务 Socket, 这个套接字才是与客户端通信并直接接收客户端发来的信息的套接字。

接收客户端连接并创建了对应的套接字后, 必须调用 AsyncSelect()函数侦测发生在该套接字上的网络事件, 选择感兴趣的网络事件进行处理, 表 2.3 列出了套接字可以侦测到的网络事件类型。

表 2.3 网络事件类型

事件标记	事件
FD_READ	接收读准备好的通知
FD_WRITE	接收写准备好的通知
FD_OOB	接收带外数据到达的通知
FD_ACCEPT	接收等待连接成功的通知
FD_CONNECT	接收已连接好的通知
FD_CLOSE	接收套接字关闭的通知

此处由于服务器的 Socket 暂时设置成单向（只收而不发），所以只需侦测 FD_READ 和 FD_CLOSE 两个事件就够了。

OnReceive()函数代码（在 ServerSocket.cpp 文件中）如下：

```
//接收客户端发来的信息
char szTemp[200];
int n = Receive(szTemp,200); //接收信息
szTemp[n] = '\0';
CString sTemp;
sTemp.Format("收到: %s",szTemp);
((CChatServerDlg *)(AfxGetApp()->m_pMainWnd))->m_ListWords.AddString(sTemp);
//显示信息
((CChatServerDlg *)(AfxGetApp()->m_pMainWnd))->m_ListWords.SetTopIndex(
    ((CChatServerDlg *)(AfxGetApp()->m_pMainWnd))->m_ListWords.GetCount() - 1);
```

OnClose()代码如下：

```
//关闭与客户端的通信信道
((CChatServerDlg *)(AfxGetApp()->m_pMainWnd))->m_ListWords.AddString("客户端断开连接");
((CChatServerDlg *)(AfxGetApp()->m_pMainWnd))->m_ListWords.SetTopIndex(
    ((CChatServerDlg *)(AfxGetApp()->m_pMainWnd))->m_ListWords.GetCount()-1);
Close(); //关闭与客户端通信的 Socket
```

再来看表 2.1，会发现有好几个函数在上面的代码剖析中已经介绍过了。如创建套接字 Create()函数、设置套接字处于监听状态的 Listen()函数、发出连接请求的 Connect()函数、接收连接请求的 Accept()函数、发送和接收数据的 Send()和 Receive()函数，以及用于通信结束最后关闭套接字的 Close()方法，它们在表 2.1 中都用粗黑体字醒目地标出，是实现套接字基本功能的函数。

8. 运行结果

现在这个程序已经具备了最简单的单向通信功能，客户端发送信息服务器可以收到，运行结果如图 2.21 所示。

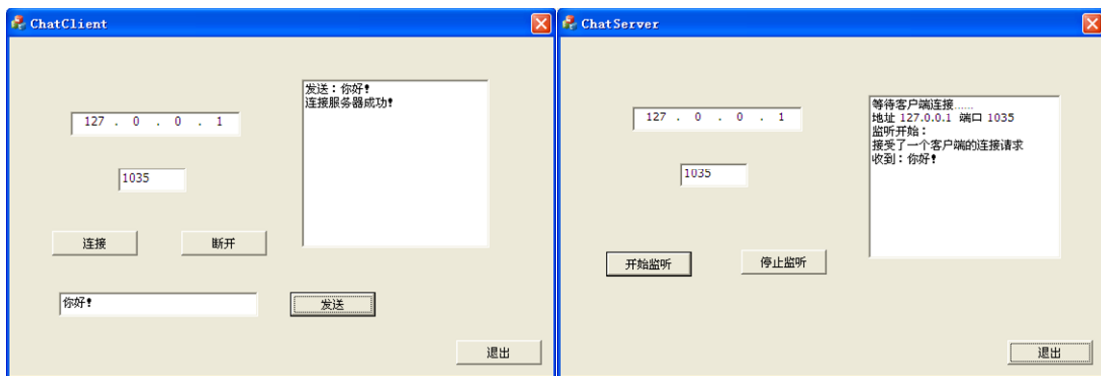


图 2.21 单向通信功能演示

9. 从单向到双向通信

刚才已经实现了客户端到服务器的单向通信，反过来从服务器到客户端的通信原理也是一样的。下面就来添加服务器到客户端的通信功能。

先在服务器界面上添加一个编辑信息的文本框和一个发送按钮，如图 2.22 所示。

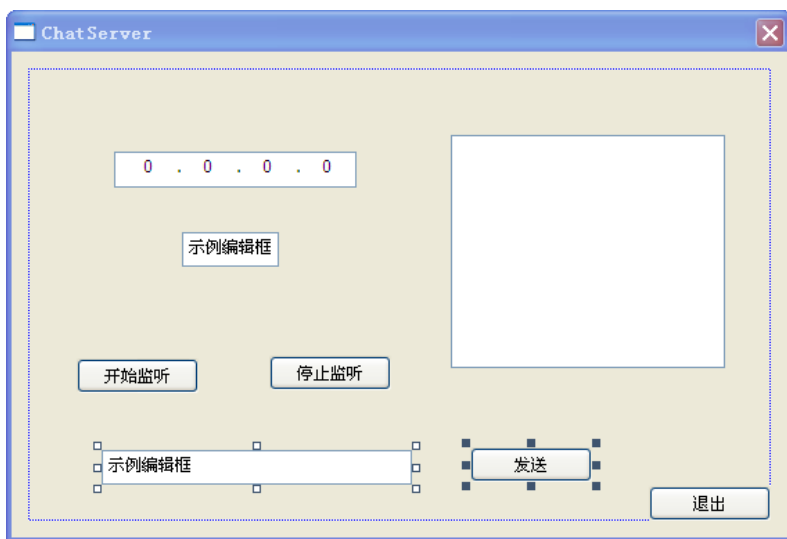


图 2.22 在服务器界面上添加控件

如客户端一样给这个文本框关联 CString 类型的变量 m_sWords。

参照前面的方法，给服务器的“发送”按钮添加与客户端一样的事件过程。

“发送”按钮的事件过程代码（在 ChatServerDlg.cpp 中）如下：

```
UpdateData();
m_ServerSocket.Send(m_sWords,m_sWords.GetLength());
m_ListWords.AddString("发送: " + m_sWords);
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
```

显然这几行代码和客户端的“发送”按钮的事件过程代码一模一样，只不过调用 Send 函数的对象由原来的 m_ClientSocket 改成了 m_ServerSocket，即改由服务器的 Socket 对象来发送信息。

与之对应，客户端当然也要编写响应函数来接收服务器发来的信息，于是为客户端添加 OnReceive 方法，代码置于 ClientSocket.cpp 中。

OnReceive()函数代码如下：

```
char szTemp[200];
int n = Receive(szTemp,200);
szTemp[n] = '\0';
CString sTemp;
sTemp.Format("收到: %s",szTemp);
((CChatClientDlg *)(AfxGetApp()->m_pMainWnd))->m_ListWords.AddString(sTemp);
((CChatClientDlg *)(AfxGetApp()->m_pMainWnd))->m_ListWords.SetTopIndex(
    ((CChatClientDlg *)(AfxGetApp()->m_pMainWnd))->m_ListWords.GetCount() - 1);
```

与服务器的 OnReceive 响应函数的代码一模一样！

由此可见，网络程序交互双方收发信息的原理是完全一样的，实质都是各自的 Socket 之间的交互。明白了这一点，也可使服务器在通信过程中具备主动断开连接的功能。在服务器界面上安放一个“断开”按钮，如图 2.23 所示。



图 2.23 给服务器添加“断开”功能

为这个“断开”按钮编写与客户端“断开”按钮一样的事件过程代码：

```
m_ServerSocket.Close();
m_ListWords.AddString("与客户端断开");
```

这里只不过将调用 Close()函数的 Socket 对象由客户端的 m_ClientSocket 改成了服务器的 m_ServerSocket，还有就是列表框里的界面状态提示信息改成了“与客户端断开”，这都无关紧要。接下来就是一样的原理，在客户端添加响应网络事件的函数代码，为客户端 Socket 重写网络事件响应函数 OnClose，代码自动置于 ClientSocket.cpp 中：

```
((CChatClientDlg *)(AfxGetApp()->m_pMainWnd))->m_ListWords.AddString("服务器断开了");
((CChatClientDlg *)(AfxGetApp()->m_pMainWnd))->m_ListWords.SetTopIndex(
((CChatClientDlg *)(AfxGetApp()->m_pMainWnd))->m_ListWords.GetCount()-1);
Close();
```

与服务器的 OnClose()函数也是一样的。

这样就轻而易举地将单向通信程序改造成了双向通信程序，可见只要熟悉了 Socket 间的通信原理，就可以举一反三。

10. 程序界面的优化和控制

我们已经完成了这个程序的全部功能，但作为软件必须具有布局合理的 GUI 用户界面，另外还需要附加一些辅助功能。如根据用户的输入和程序运行状态，自动将界面上的按钮设置成生效或失效，谨防用户误操作；一些对可靠性要求很高的程序，还必须能够容忍用户的错误输入并考虑程序运行时各种可能的意外及处理对策，具备完善而系统的异常处理和出错提示代码；另外，一般的程序都会有版本信息显示框……诸如此类，虽然不是核心功能，但对于一个完整的小软件来说也是必不可少的。

先在界面上添加一些控件并重新设计布局，如图 2.24 和图 2.25 所示。注意：对原有的控件只能移动其位置，改变外观大小和样式属性，不要随意删除或是用其他控件替换。

为了使通信过程中的状态信息和通信记录按顺序显示，把双方列表框控件的 sort 属性均设为 false，同时，为了美观，外加 3D 边框和垂直滚动条。

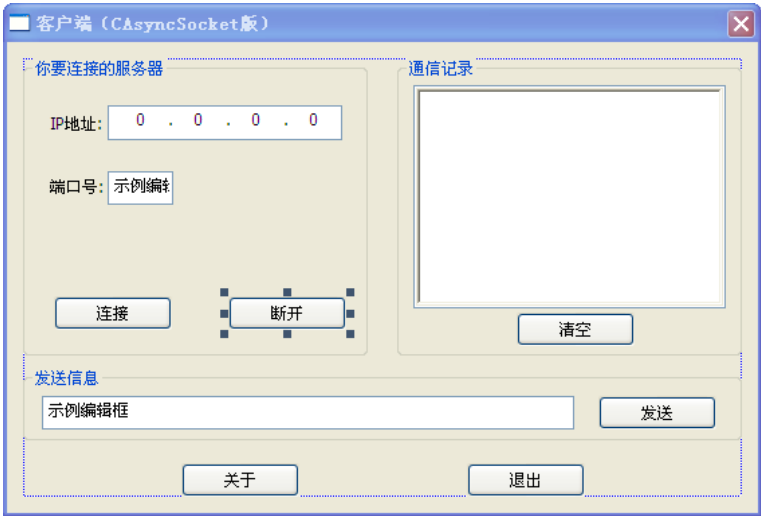


图 2.24 重新布局后的客户端界面



图 2.25 重新布局后的服务器界面

各控件关联的变量见表 2.4 和表 2.5。

表 2.4 客户端控件变量

控 件	变量（包括类型）
IP 地址控件	contrl ServerIP
“端口”编辑框	contrl ServerPort; int sPort
“连接”按钮	contrl m_ButtonConnect
“断开”按钮	contrl m_ButtonDisconnect
列表框	contrl m_ListWords
“清空”按钮	contrl m_ButtonClear
发送信息编辑框	contrl m_EditWords; CString m_sWords
“发送”按钮	contrl m_ButtonSend
“退出”按钮	contrl m_ButtonExit

表 2.5 服务器控件变量

控 件	变量 (包括类型)
IP 地址控件	ctrl ServerIP
“端口” 编辑框	ctrl ServerPort; int sPort
列表框	ctrl m_ListWords
“断开” 按钮	ctrl m_ButtonDisconnect
“清空” 按钮	ctrl m_ButtonClear
“退出” 按钮	ctrl m_ButtonExit
“开始监听” 按钮	ctrl m_ButtonListen
“停止监听” 按钮	ctrl m_ButtonStopListen
发送信息编辑框	ctrl m_EditWords; CString m_sWords
“发送” 按钮	ctrl m_ButtonSend

这里几乎给界面上的每个控件都添加了变量关联,除了值类型的变量是为了在程序运行时获得用户输入外,其余很多变量都是用来标识控件的,以后就靠它们来识别和区分控件,从而达到根据程序运行状态和用户操作来控制界面上各控件可用状态的目的。

以下是在各个方法和事件过程中增加的用于完善程序界面的代码。

1) 客户端

在 ChatClientDlg.cpp 中, BOOL CChatClientDlg::OnInitDialog()函数的初始化代码如下:

```
m_ButtonDisconnect.EnableWindow(false);
m_ButtonClear.EnableWindow(false);
m_EditWords.EnableWindow(false);
m_ButtonSend.EnableWindow(false);
```

“清空” 按钮事件过程代码如下:

```
m_ListWords.ResetContent(); //清空通信状态列表中的信息
```

“断开” 按钮事件过程代码如下:

```
ServerIP.EnableWindow();
ServerPort.EnableWindow();
m_ButtonConnect.EnableWindow();
m_ButtonDisconnect.EnableWindow(false);
m_EditWords.EnableWindow(false);
m_ButtonSend.EnableWindow(false);
m_ButtonExit.EnableWindow();
```

“关于” 按钮事件过程代码如下:

```
CAboutDlg dlgAbout;
dlgAbout.DoModal(); //显示“关于”对话框
```

在 ClientSocket.cpp 中, OnConnect()的界面控制代码如下:

```
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->ServerIP.EnableWindow(false);
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->ServerPort.EnableWindow(false);
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonConnect.EnableWindow(false);
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonDisconnect.EnableWindow();
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_EditWords.EnableWindow();
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonSend.EnableWindow();
```

```
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonExit.EnableWindow(false);
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonClear.EnableWindow();
```

OnClose()中的界面控制代码如下:

```
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->ServerIP.EnableWindow();
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->ServerPort.EnableWindow();
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonConnect.EnableWindow();
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonDisconnect.EnableWindow(false);
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_EditWords.EnableWindow(false);
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonSend.EnableWindow(false);
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonExit.EnableWindow();
```

2) 服务器

在 ChatServerDlg.cpp 中, BOOL CChatServerDlg::OnInitDialog()函数的初始化代码如下:

```
m_ButtonStopListen.EnableWindow(false);
m_ButtonDisconnect.EnableWindow(false);
m_ButtonClear.EnableWindow(false);
m_EditWords.EnableWindow(false);
m_ButtonSend.EnableWindow(false);
```

“开始监听”按钮事件过程代码如下:

```
ServerIP.EnableWindow(false);
ServerPort.EnableWindow(false);
m_ButtonListen.EnableWindow(false);
m_ButtonStopListen.EnableWindow();
m_ButtonClear.EnableWindow();
m_ButtonExit.EnableWindow(false);
```

“停止监听”按钮事件过程代码如下:

```
ServerIP.EnableWindow();
ServerPort.EnableWindow();
m_ButtonListen.EnableWindow();
m_ButtonStopListen.EnableWindow(false);
m_ButtonExit.EnableWindow();
```

“断开”按钮事件过程代码如下:

```
m_ButtonDisconnect.EnableWindow(false);
m_EditWords.EnableWindow(false);
m_ButtonSend.EnableWindow(false);
```

“清空”按钮事件过程代码如下:

```
m_ListWords.ResetContent(); //清空状态历史列表中的信息
```

“关于”按钮事件过程代码如下:

```
CAboutDlg dlgAbout;
dlgAbout.DoModal(); //显示“关于”对话框
```

在 ListenSocket.cpp 中, OnAccept()函数的界面控制代码如下:

```
((CChatServerDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonDisconnect.EnableWindow();
((CChatServerDlg*)(AfxGetApp()->m_pMainWnd))->m_EditWords.EnableWindow();
((CChatServerDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonSend.EnableWindow();
```

在 ServerSocket.cpp 中, OnClose()函数的界面控制代码如下:

```
((CChatServerDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonDisconnect.EnableWindow(false);
```

```
((CChatServerDlg *)(AfxGetApp()->m_pMainWnd))->m_EditWords.EnableWindow(false);
((CChatServerDlg *)(AfxGetApp()->m_pMainWnd))->m_ButtonSend.EnableWindow(false);
```

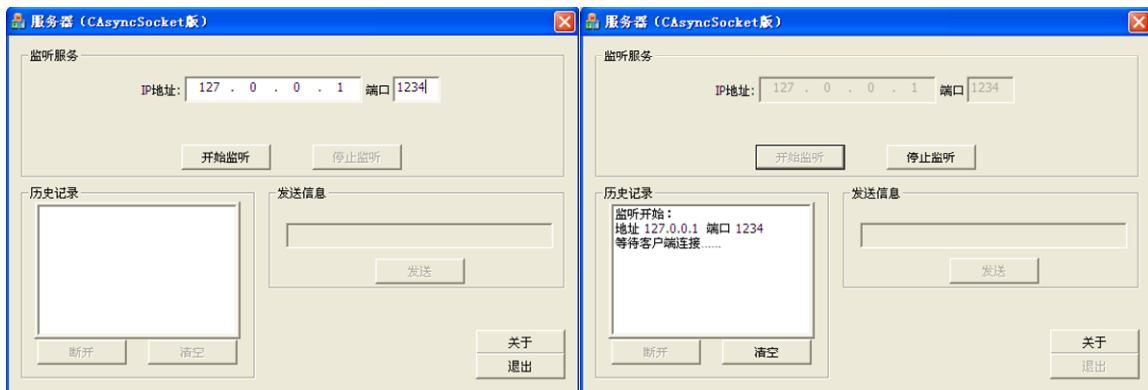
别看上面的代码这么多,其实都是可有可无的,即使将它们全都删除,也丝毫不会影响程序的通信功能。这些增加的代码与 Socket 编程本身无关,主要起到优化和控制用户界面的作用。本书在此就不再一句一句详细剖析这些代码了,可能读者也看出来了,它们其实大同小异,都在频繁使用着 EnableWindow 这个函数,不错,这个函数就是用来动态设置用户界面上各个控件的可用性的。它的基本调用格式是:控件变量名.EnableWindow(BOOL)。其中参数默认为 true,默认调用时将相应的控件设置为生效可用,如果显式声明为 false,即“控件变量名.EnableWindow(false)”,则将对对应控件设置为不可用。

鉴于它们有一定的通用性,这里截取上面的一小段代码来举一个例子,让读者对使用 EnableWindow 函数控制界面的方法有一个感性认识。

例如,“开始监听”事件过程代码如下:

```
(1) ServerIP.EnableWindow(false);           //使 IP 地址控件不可用
(2) ServerPort.EnableWindow(false);         //使端口号固定无法修改
(3) m_ButtonListen.EnableWindow(false);     //使监听按钮失效
(4) m_ButtonStopListen.EnableWindow();      //使用户可以随时停止监听
(5) m_ButtonClear.EnableWindow();           //使清空按钮生效
(6) m_ButtonExit.EnableWindow(false);       //使退出按钮失效
```

服务器程序在用户按下“开始监听”按钮前后界面的变化对比如图 2.26 所示。



(a) 按下按钮前

(b) 按下按钮后

图 2.26 用户按下“开始监听”按钮前后界面对比

由上述代码可见,第(1)、(2)句代码将 IP 地址控件和端口号编辑框置成不可编辑的状态,是为了防止用户在监听开始后再恶作剧地修改已经确定下来的监听地址,第(3)句代码可限制用户重复操作,第(4)句代码使用户可以随时停止监听,第(5)句代码使清空按钮生效……这些都给用户提供了方便,而最后第(6)句代码将退出按钮设置成不可用很重要,因为这样就逼着用户必须将启动的监听线程关闭后才能退出程序,以免用户开启的监听线程遗留在操作系统里成为垃圾线程。

当然,读者也可根据自己的需要动态设置界面上控件的状态,使程序的交互性更强、更友好、更易用、更安全。在本书以后的程序示例中,对于这些通用的界面控制语句将只贴出代码而不再加以说明,因为它们的原理都是一样的。

最后,谈一下“关于”对话框,这个对话框是几乎所有 Windows 程序都有的,用来显示软

件的版权声明和版本信息，可以通过“关于”事件过程启动。

“关于”事件过程代码如下：

```
CAboutDlg dlgAbout;  
dlgAbout.DoModal();           //显示“关于”对话框
```

在资源视图的目录树 Dialog 子目录下的第一个项目 IDD_ABOUTBOX 就是“关于”对话框的 ID，双击它可以进入“关于服务器”对话框的设计界面（如图 2.27 所示）。

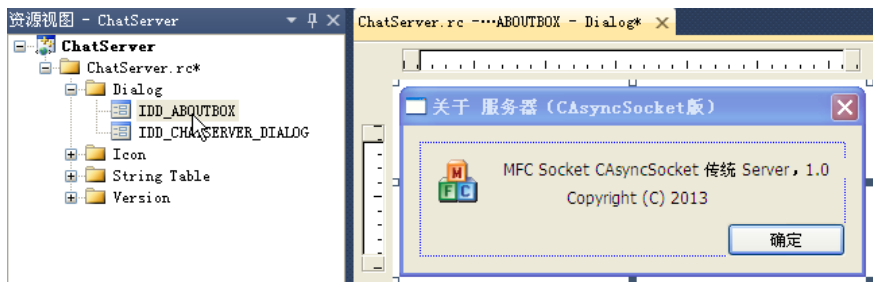


图 2.27 “关于服务器”对话框设计界面

在此，用户可以设计“关于客户端（或服务器）”对话框的外观，写上版权声明。

11. 程序完全演示

到此为止，这个简单的网络进程通信程序就全部编写完了，虽然简单，但功能还是很完善的，程序有着朴素简洁但交互性强、易用的图形界面，双向的通信功能，下面来完整地运行演示。

分别开启客户端程序和服务器程序（如图 2.28 所示）。

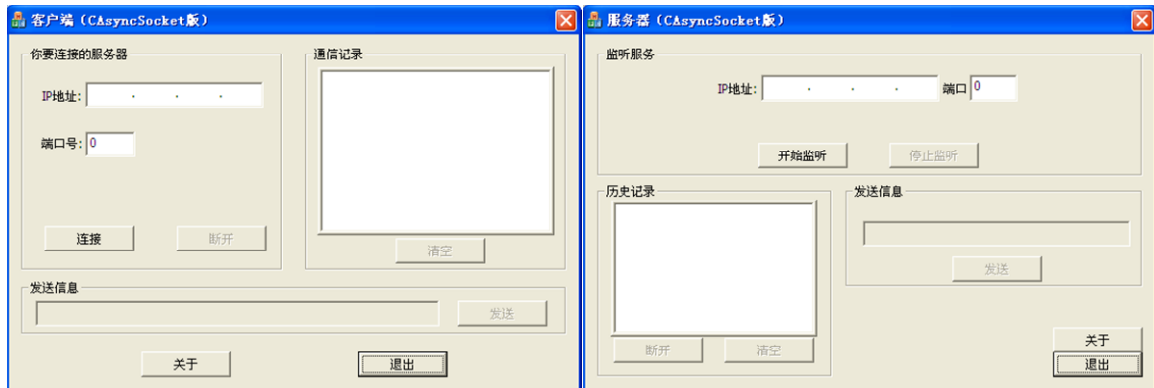


图 2.28 开启程序

在服务器输入 IP 地址和端口，单击“开始监听”按钮，于是服务器在这个地址上等待客户端来连接，历史记录栏里将自动记下这一事件，如图 2.29 所示。

在客户端地址栏里输入与服务器一样的 IP 和端口，单击“连接”按钮，连上服务器，可以看到双方的状态栏里都反映了这次连接的情况（如图 2.30 所示）。

接下来双方就可以互相通信了，可以互发信息，信息内容会实时自动地显示在双方的通信记录列表中，并且用户还会发现：当某方的通信记录多到一定量时，列表框会自动以滚动条方式显示（如图 2.31 所示）。

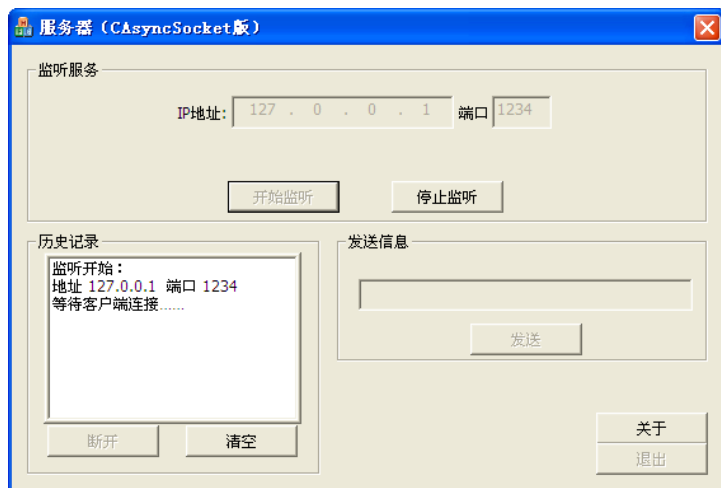


图 2.29 服务器开始监听

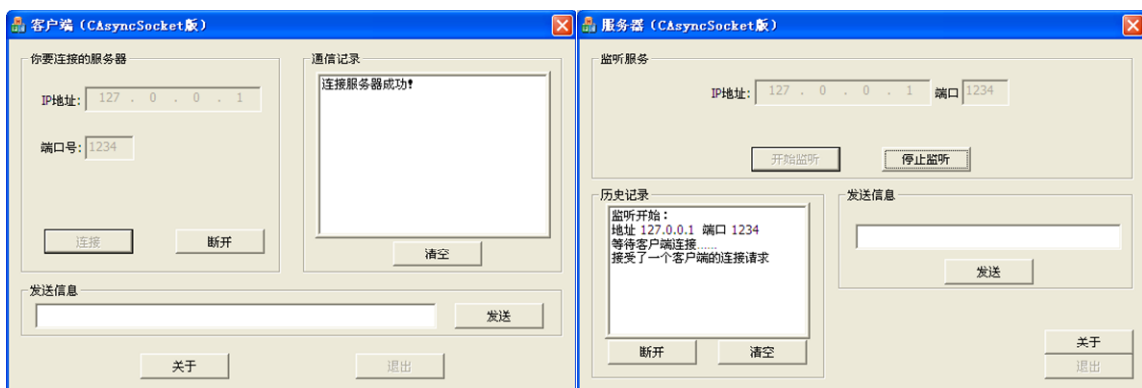


图 2.30 客户端连上服务器

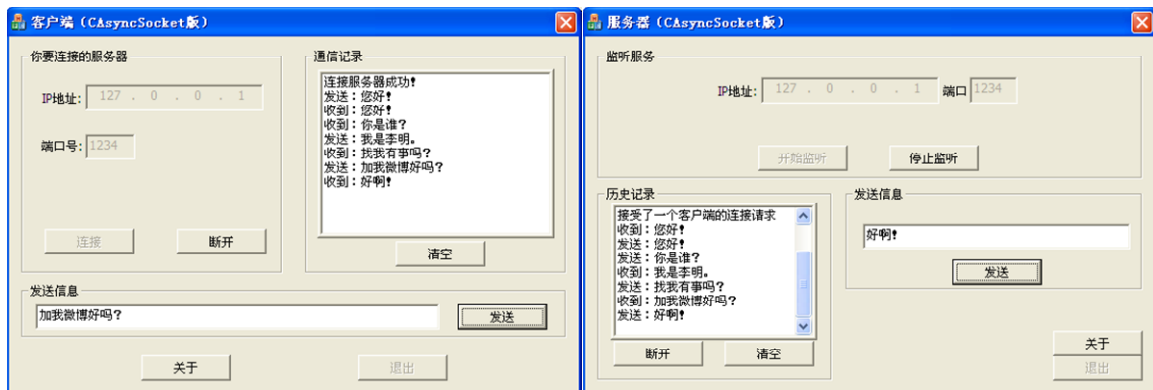


图 2.31 通信过程

通信过程中任何一方都可以主动断开连接 (如图 2.32 所示)。

也可以重新连接, 并随时清空通信记录; 单击“关于”按钮, 可以查看版权信息; 服务器随时可以停止监听。

客户端连接服务器失败时会弹出如图 2.33 所示的失败提示。

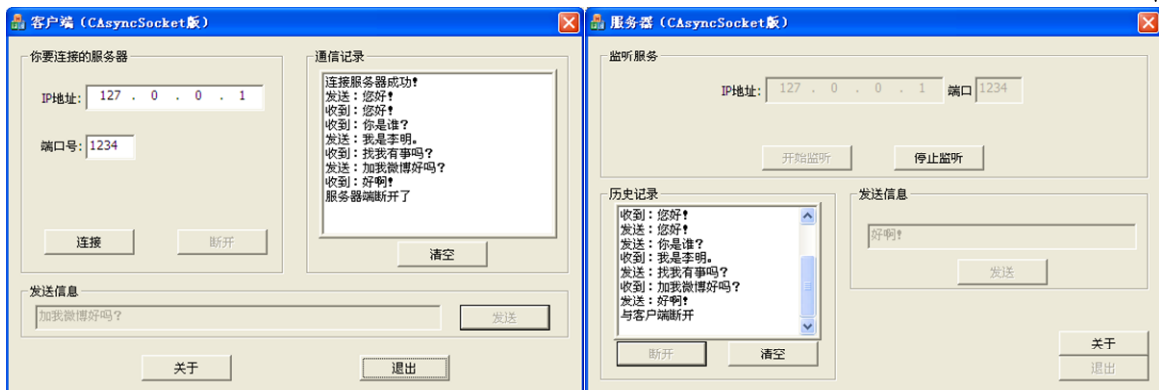


图 2.32 断开连接

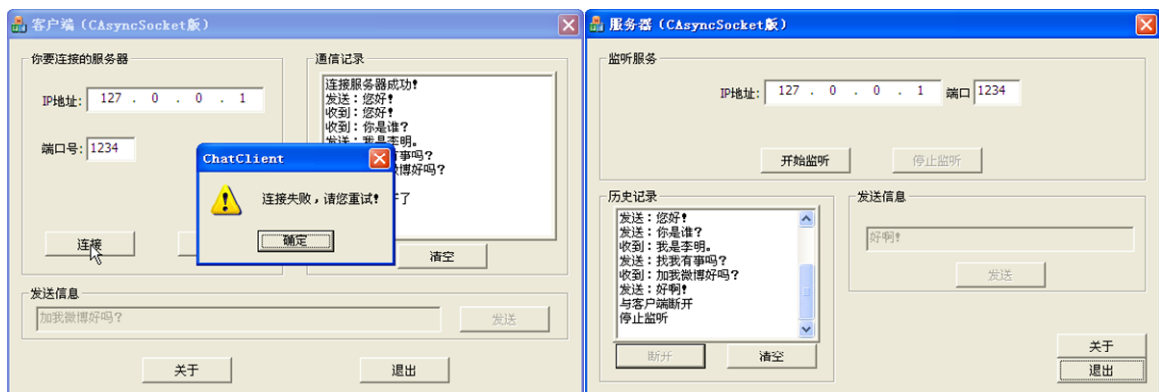


图 2.33 连接失败提示

可见, 整个程序功能完整, 可以演示典型的网络通信流程。

通过这个典型实例, 只是向读者展示了 MFC Socket 最简单的通信原理和流程, 即网络上两个应用进程之间是如何传递信息的, 并不是聊天应用程序的例子。有关网络聊天软件的原理和开发将在第 4 章介绍。但本例不仅是即时通信聊天应用, 同时也是其他一切网络应用软件底层通信的基础。

2.2.3 CAsyncSocket 类程序的指针实现

C++语言编程的灵活之处(也是最难以掌握之处)要数指针的应用了, 而在实际应用中, 指针又是不可或缺的工具, 因为它有着其他语言要素不可替代的优点。实际应用中很多程序的代码都是用指针及与其相关的数据结构、算法写成的, 不了解指针就很难读懂别人写的实用代码, 为此本节改用指针机制重新实现一遍 2.2.2 节的 Socket 程序, 以便读者比照学习, 掌握 VC 网络编程中指针的通行用法。

1. 非指针程序的机制缺陷

2.2.2 节的程序很简单, 通过剖析读者能够一目了然地看清它的结构, 但是如果编写复杂的网络程序, 就没那么容易理清程序结构了。试想一下, 如果通信过程要涉及好几个客户端和服务端, 每个通信方都有自己的 Socket (而且还不止一个), 若这时服务器同时接收很多客户端的连接, 就需要为每个客户端都创建一个用于通信的 Socket, 而一旦某个客户端断开连接, 那么

服务器为这个客户端创建的 Socket 就再也没有用了，但它却仍然作为一个程序模块存在，占据着代码空间。更糟的是，这种编程方式要求在写程序前就确定好运行时要用到的全部 Socket 并分别为其创建类对象、编写代码，但网络通信是一个动态参与的过程，任何一方都可以自主决定何时退出，也可以不断地有新进程加入，因此事先确定程序运行中要用到的全部 Socket 是不现实的。

归纳起来，2.2.2 节的程序存在以下一些编程机制上的缺陷：

- Socket 的创建、使用和销毁不够灵活，无法支持很多 Socket 动态参与通信过程。
- 程序代码分散在各个不同的 Socket 模块中，不利于统一管理和维护。
- 如果要在 Socket 的代码中访问和控制主对话框界面上的控件，就需要运用 AfxGetApp() 全局函数获取主窗口 CWinApp 类指针的方法。这意味着每一个在 Socket 中访问到主界面控件的语句前都要写上冗长的类似下面的这句代码：

```
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->
```

这会使程序代码看上去显得很“啰嗦”，缺乏简洁的美感和可读性。

基于上述一些弊病，现在很多实际使用的网络程序在编写时都采用了一种叫作“对话框指针机制”的新方法，接下来改用这种新机制来重写一遍 2.2.2 节的 Socket 程序。

2. 用对话框指针机制实现的 Socket 程序

1) 建立工程

分别创建客户端和服务端工程，相关设置同 2.2.2 节，创建完成后在客户端和服务端工程中各添加一个基于 CAsyncSocket 类的 MySocket 类，如图 2.34 所示。这个类是用来给通信双方动态生成 Socket 对象服务的。

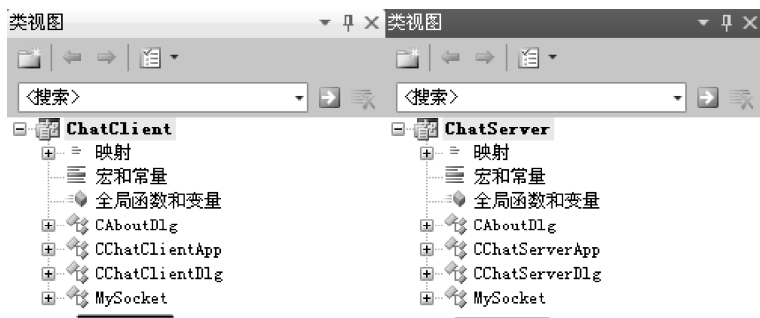


图 2.34 添加 MySocket 类

参照图 2.24、图 2.25 设计一模一样的程序界面，仅在标题栏上标注“(指针机制)”以示区别。界面上所有的控件及属性、关联的变量都与 2.2.2 节程序中的完全相同，具体设置见表 2.4 和表 2.5。

2) 用对话框指针机制组织新程序的框架

(1) 客户端。

在 MySocket.h 中添加如下代码：

```
class CChatClientDlg;    //为了能在 MySocket 类中定义主对话框类指针，先对主对话框类进行前导声明
CChatClientDlg * m_dlg;    //在 MySocket 类中定义一个主对话框指针
void GetDlg(CChatClientDlg * dlg);    //获取主对话框指针的函数
```

代码添加位置如图 2.35 所示。

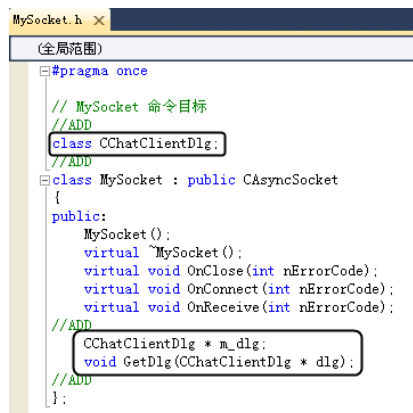


图 2.35 MySocket.h 中添加的代码

给 MySocket 类添加网络事件处理函数，在这里添加 OnClose()、OnConnect()、OnReceive() 三个函数，分别用于响应服务器断开连接事件、客户端发起连接请求事件、接收到服务器发来的数据，编程环境将自动为每个函数生成函数声明和函数体的框架。

在 ChatClientDlg.h 中添加如下代码：

```

#include "MySocket.h" //引用 MySocket 类的头文件，是为了下面定义一个 MySocket 类对象指针
MySocket * m_ClientSocket; //定义一个 MySocket 类对象指针，用于动态生成客户端 Socket
void OnReceive();
void OnClose();
void OnConnect(); //后面要在主对话框代码中实现这三个函数的功能
void SocketReset(); //套接字重置销毁函数，及时清理不再使用的 Socket，避免指针悬空

```

在 ChatClientDlg.cpp 中将 MySocket 类对象指针初始化为空，如图 2.36 所示，写出已声明的三个函数 OnClose()、OnConnect()、OnReceive() 的函数体，同时实现 SocketReset() 函数。代码如下：

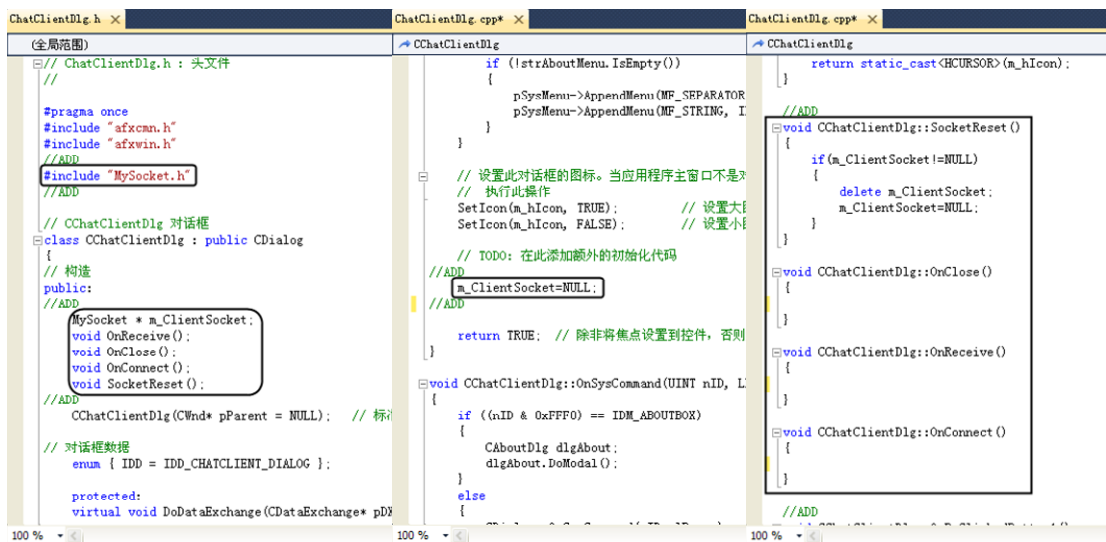


图 2.36 代码布局

```

void CChatClientDlg::SocketReset() //SocketReset 函数实现
{

```

```

        if(m_ClientSocket!=NULL)
        {
            delete m_ClientSocket;
            m_ClientSocket=NULL;
        }
    }
}

```

在 `MySocket.cpp` 中实现获取主对话框指针的 `GetDlg()`函数, 并通过指针引用主对话框程序代码中的网络事件处理函数。代码如下:

```

#include "ChatClientDlg.h"
void MySocket::GetDlg(CChatClientDlg * dlg)           //获得窗口界面的指针
{
    m_dlg=dlg;
}
void MySocket::OnClose(int nErrorCode)
{
    //TODO: 在此添加专用代码和/或调用基类
    m_dlg->OnClose();
    CAsyncSocket::OnClose(nErrorCode);
}
void MySocket::OnConnect(int nErrorCode)
{
    //TODO: 在此添加专用代码和/或调用基类
    m_dlg->OnConnect();
    CAsyncSocket::OnConnect(nErrorCode);
}
void MySocket::OnReceive(int nErrorCode)
{
    //TODO: 在此添加专用代码和/或调用基类
    m_dlg->OnReceive();
    CAsyncSocket::OnReceive(nErrorCode);
}

```

如此一来, 程序运行中动态生成的 `Socket` 就能自如地使用主对话框源文件中各代码模块的功能了。

(2) 服务器。

采用与客户端类似的方法, 在 `MySocket.h` 中添加如下代码:

```

class CChatServerDlg;           //先对主对话框类进行前导声明
CChatServerDlg * m_dlg;        //定义主对话框指针
void GetDlg(CChatServerDlg * dlg); //指针获取函数

```

给 `Socket` 类添加事件响应代码, 因为是服务器, 需要接收客户端的连接请求, 因此添加 `OnAccept()`、`OnClose()`、`OnReceive()`三个函数, 分别用于接收客户端连接、响应客户端突然断开的事件、接收客户端发来的数据。与客户端一样, 系统自动为每个函数生成函数声明和函数体框架, 另外, 在 `ChatServerDlg.cpp` 中将 `MySocket` 类对象指针初始化为空。

```

#include "MySocket.h"           //引用 MySocket 类头文件, 为了后面定义动态 Socket 对象指针
MySocket * m_ServerSocket;      //MySocket 类对象指针, 用于动态生成与客户端通信的 Socket
MySocket * m_ListenSocket;      //定义 MySocket 类对象指针, 用于动态生成监听 Socket
void OnReceive();
void OnClose();

```

```
void OnAccept();           //要在主对话框代码中实现的服务器功能的三个函数
void SocketReset();       //重置销毁不再使用的套接字
```

在 ChatServerDlg.cpp 中写出 OnAccept()、OnClose()、OnReceive()三个函数的函数体，为它们的实现代码预留出空间，同时实现 SocketReset 函数。代码如下：

```
void CChatServerDlg::SocketReset()    //SocketReset 函数实现
{
    if(m_ServerSocket != NULL)
    {
        delete m_ServerSocket;        //在此要销毁两个 Socket，一个是与客户端通信的 Socket
        m_ServerSocket = NULL;
    }
    if(m_ListenSocket != NULL)
    {
        delete m_ListenSocket;        //还有一个是监听 Socket
        m_ListenSocket = NULL;
    }
}
```

程序代码在开发环境中的布局效果与图 2.36 所示一样，不再重复展示。

在 MySocket.cpp 中实现获取主对话框指针的 GetDlg()函数，并通过指针引用主对话框的三个函数：

```
#include "ChatServerDlg.h"
void MySocket::GetDlg(CChatServerDlg *dlg)    //获得窗口界面的指针
{
    m_dlg = dlg;
}
void MySocket::OnClose(int nErrorCode)
{
    //TODO: 在此添加专用代码和/或调用基类
    m_dlg->OnClose();
    CAsyncSocket::OnClose(nErrorCode);
}
void MySocket::OnReceive(int nErrorCode)
{
    //TODO: 在此添加专用代码和/或调用基类
    m_dlg->OnReceive();
    CAsyncSocket::OnReceive(nErrorCode);
}
void MySocket::OnAccept(int nErrorCode)
{
    //TODO: 在此添加专用代码和/或调用基类
    if(m_dlg->m_ServerSocket == NULL)
        m_dlg->OnAccept();
    CAsyncSocket::OnAccept(nErrorCode);
}
```

3) 代码的复制、修改

下面只要为界面上的按钮编写事件过程代码，并在主对话框的几个功能函数中添加处理代码，就能实现进程之间的通信。2.2.2 节程序的代码可以直接复制过来使用，只要填写在适当的位置就行了，所要做的只是进行很小的局部修改以适应新的编程机制。

(1) 客户端。

“连接”按钮事件过程代码如下：

//初始化套接字，获取对话框指针

```
if(!AfxSocketInit())
```

```
{
```

```
    MessageBox("WindowSocket initial failed!", "Receive", MB_ICONSTOP);
```

```
    return;
```

```
}
```

```
m_ClientSocket = new MySocket;
```

```
m_ClientSocket->GetDlg(this);
```

//连接服务器

```
BYTE nFild[4];
```

```
CString sIP;
```

```
UpdateData();
```

```
ServerIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
```

```
sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
```

```
m_ClientSocket->Create();           //创建客户端 Socket
```

```
m_ClientSocket->Connect(sIP,sPort); //发起连接请求
```

可以清楚地看到，上段代码与原来相比只有少量改动（加黑部分），因为使用指针在运行时动态生成 Socket，故需要使用 AfxSocketInit()函数初始化，用 new 语句生成 Socket 对象，用 GetDlg(this)获取对话框指针。另外，在调用 Socket()函数如 Create()、Connect()时要使用指针运算符“->”，如 m_ClientSocket->Create()，这是因为 m_ClientSocket 并不是静态的 Socket 对象，而是程序在运行时动态生成的指向 Socket 对象的指针，是引用类型。

“断开”按钮事件过程代码如下：

//断开与服务器的连接

```
m_ClientSocket->Close();           //关闭客户端 Socket
```

```
SocketReset();                     //避免指针悬空
```

```
m_ListWords.AddString("从服务器断开");
```

//界面完善

```
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
```

```
ServerIP.EnableWindow();
```

```
ServerPort.EnableWindow();
```

```
m_ButtonConnect.EnableWindow();
```

```
m_ButtonDisconnect.EnableWindow(false);
```

```
m_EditWords.EnableWindow(false);
```

```
m_ButtonSend.EnableWindow(false);
```

```
m_ButtonExit.EnableWindow();
```

使用指针机制时，如果需要断开连接，除了调用 Socket 的 Close()方法外，还要及时将指针置为 NULL，避免指针悬空，这很重要。大家在使用指针机制进行编程时要养成及时释放废指针的好习惯，在这个程序中，由于我们将这些操作事先都封装在 SocketReset()函数中了，因此在执行 m_ClientSocket->Close()后将调用 SocketReset()函数。

“发送”按钮事件过程代码如下：

```
UpdateData();
```

```
m_ClientSocket->Send(m_sWords,m_sWords.GetLength()); //向服务器发送信息
```

```
m_ListWords.AddString("发送: " + m_sWords);
```

```
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
```

这里只要改用指针运算符“->”调用 Send()函数就行了，其他的都不用变。

OnClose()函数代码如下：

```
m_ListWords.AddString("服务器断开了");
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
m_ClientSocket->Close();
SocketReset();          //避免指针悬空
//界面完善
ServerIP.EnableWindow();
ServerPort.EnableWindow();
m_ButtonConnect.EnableWindow();
m_ButtonDisconnect.EnableWindow(false);
m_EditWords.EnableWindow(false);
m_ButtonSend.EnableWindow(false);
m_ButtonExit.EnableWindow();
```

OnReceive()函数代码如下：

```
char szTemp[200];
int n = m_ClientSocket->Receive(szTemp,200);
szTemp[n] = '\0';
CString sTemp;
sTemp.Format("收到: %s",szTemp);
m_ListWords.AddString(sTemp);
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
```

OnConnect()函数代码如下：

```
m_ListWords.AddString("连接服务器成功！");
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
//界面完善
ServerIP.EnableWindow(false);
ServerPort.EnableWindow(false);
m_ButtonConnect.EnableWindow(false);
m_ButtonDisconnect.EnableWindow();
m_EditWords.EnableWindow();
m_ButtonSend.EnableWindow();
m_ButtonExit.EnableWindow(false);
m_ButtonClear.EnableWindow();
```

这里需要说明一下，因为 CAsyncSocket 是异步操作的 Socket 类，它在使用 Connect 方法发起连接请求后，不管连接成功与否都立即返回，我们只能根据 OnConnect 事件的错误码 nErrorCode 来判断连接是否成功，而要获取这个错误码值，就必须进入响应 OnConnect 事件的处理过程，这就决定了程序员自己在主对话框中定义的函数 OnConnect()的代码必须有一部分置于 Socket 本身的 OnConnect 响应函数中：

```
void MySocket::OnConnect(int nErrorCode)
{
    //TODO: 在此添加专用代码和/或调用基类
    //确认客户端是否成功连接到服务器
    if(nErrorCode)
```

```

    {
        AfxMessageBox("连接失败, 请您重试! ");
        return;
    }
    m_dlg->OnConnect();
    CAsyncSocket::OnConnect(nErrorCode);
}

```

然后再运用指针机制 `m_dlg->OnConnect()` 转入自定义的处理函数。

客户端“关于”、“清空”按钮的代码和 2.2.2 节的程序完全相同, 在此略去。

(2) 服务器。

“开始监听”按钮事件过程代码如下:

//初始化套接字, 获取对话框指针

```

if(!AfxSocketInit())
{
    MessageBox("WindowSocket initial failed!", "Send", MB_ICONSTOP);
    return;
}
m_ListenSocket = new MySocket;
m_ListenSocket->GetDlg(this);
//监听开始, 服务器等待连接请求的到来
BYTE nFild[4];
CString sIP,sP;
UpdateData();
ServerIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
sP.Format("%d",sPort);
m_ListenSocket->Create(sPort,1,FD_ACCEPT,sIP); //创建服务器监听 Socket
m_ListenSocket->Listen(1); //开始监听
m_ListWords.AddString("监听开始: ");
m_ListWords.AddString("地址" + sIP + " 端口" + sP);
m_ListWords.AddString("等待客户端连接……");
//界面完善
m_ListWords.SetTopIndex(m_ListWords.GetCount()-1);
ServerIP.EnableWindow(false);
ServerPort.EnableWindow(false);
m_ButtonListen.EnableWindow(false);
m_ButtonStopListen.EnableWindow();
m_ButtonClear.EnableWindow();
m_ButtonExit.EnableWindow(false);

```

可以看出, 只是前面多了一段初始化代码, 还有就是改用指针运算符调用 `Socket` 函数。

“停止监听”按钮事件过程代码如下:

```

//停止监听
m_ListenSocket->Close(); //关闭服务器监听 Socket
if(m_ListenSocket != NULL) //防止指针悬空
{
    delete m_ListenSocket;
}

```

```

m_ListenSocket = NULL;
}
m_ListWords.AddString("停止监听");
//界面完善
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
ServerIP.EnableWindow();
ServerPort.EnableWindow();
m_ButtonListen.EnableWindow();
m_ButtonStopListen.EnableWindow(false);
m_ButtonExit.EnableWindow();

```

这里直接编写代码释放监听 Socket 的指针，不会影响已经建立好连接的 Socket。

“断开”按钮的事件过程代码如下：

```

m_ServerSocket->Close();
if(m_ServerSocket!=NULL)
{
    delete m_ServerSocket;
    m_ServerSocket=NULL;
}
m_ListWords.AddString("与客户端断开");
//界面完善
m_ListWords.SetTopIndex(m_ListWords.GetCount()-1);
m_ButtonDisconnect.EnableWindow(false);
m_EditWords.EnableWindow(false);
m_ButtonSend.EnableWindow(false);

```

同样是单独释放与客户端通信的指针，此时监听 Socket 仍在运行。

“发送”按钮的事件过程代码如下：

```

UpdateData();
m_ServerSocket->Send(m_sWords,m_sWords.GetLength());//获取文本长度
m_ListWords.AddString("发送: " + m_sWords);           //显示发送文件长度记录
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);

```

OnAccept()函数代码如下：

```

//动态创建用于通信的 Socket
m_ServerSocket = new MySocket;
m_ServerSocket->GetDlg(this);
//接收客户端的连接请求
m_ListenSocket->Accept(*m_ServerSocket);           //接收连接请求
m_ServerSocket->AsyncSelect(FD_READ|FD_CLOSE);
m_ListWords.AddString("接收了一个客户端的连接请求");
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
//界面完善
m_ButtonDisconnect.EnableWindow();
m_EditWords.EnableWindow();
m_ButtonSend.EnableWindow();

```

OnClose()函数代码如下：

```

//关闭与客户端的通信信道
m_ListWords.AddString("客户端断开连接");

```

```

m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
m_ServerSocket->Close();           //关闭与客户端通信的 Socket
if(m_ServerSocket != NULL)
{
    delete m_ServerSocket;
    m_ServerSocket = NULL;
}
//界面完善
m_ButtonDisconnect.EnableWindow(false);
m_EditWords.EnableWindow(false);
m_ButtonSend.EnableWindow(false);

```

OnReceive()函数代码如下:

```

//接收客户端发来的信息
char szTemp[200];
int n = m_ServerSocket->Receive(szTemp,200);           //接收信息
szTemp[n] = '\0';
CString sTemp;
sTemp.Format("收到: %s",szTemp);
m_ListWords.AddString(sTemp);                           //显示信息
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);

```

因为这个程序是使用指针机制编写的,为防止用户突然退出程序而造成指针悬空,给“退出”按钮的事件过程增加调用 SocketReset()函数。

“退出”按钮的事件过程代码如下:

```

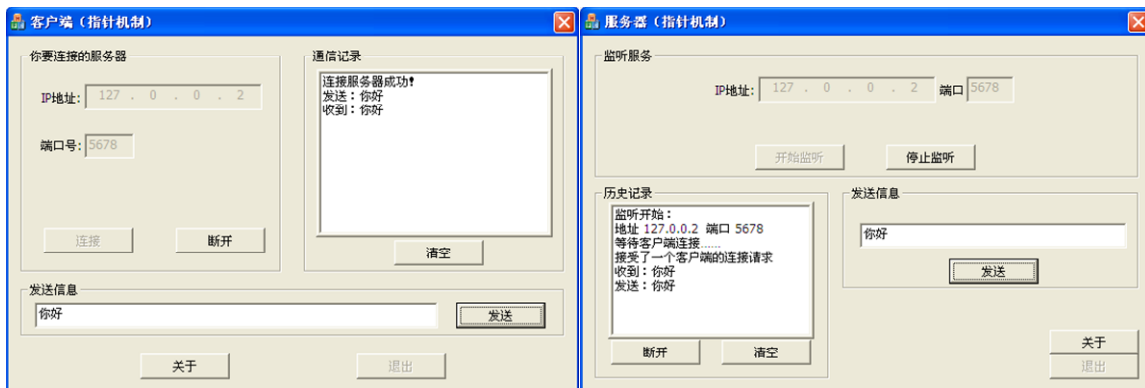
void CChatServerDlg::OnBnClickedCancel()
{
    //TODO: 在此添加控件通知处理程序代码
    SocketReset();
    OnCancel();
}

```

服务器“关于”、“清空”按钮的代码也和原来的程序一样,此处略去。

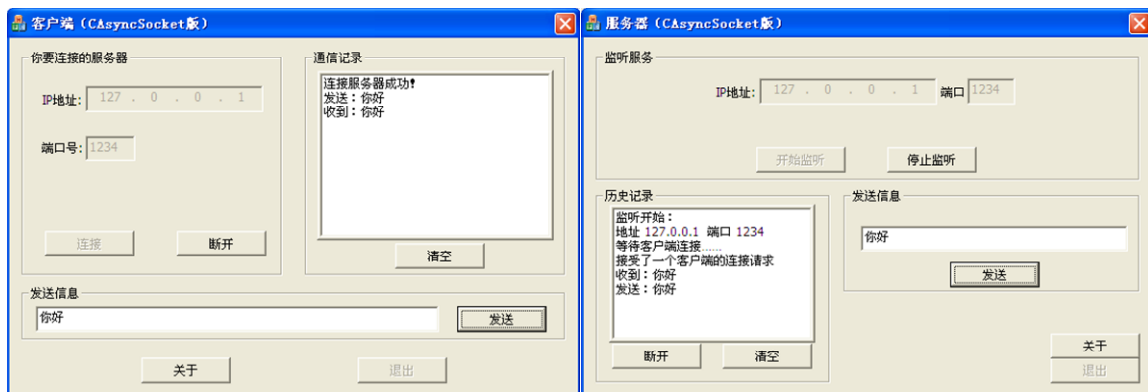
4) 运行对比

下面来运行这个程序,同时运行 2.2.2 节的程序,并将它们放在一个桌面上对比,如图 2.37 所示。



(a) 指针机制的程序

图 2.37 两个版本程序的运行对比



(b) 传统机制的程序

图 2.37 两个版本程序的运行对比 (续)

显然，它们“长得”一模一样，但是内部的编程机制却完全不同！可见，指针机制的程序和传统机制的程序只是代码的组织方式不同，在功能上是毫无二致的。这说明采用不同的编程技术可以写出外观完全一样的程序，多么奇妙啊！

3. 对话框指针机制的优点

从本节两个不同实现机制的程序对比中可以看出，使用对话框指针机制的程序比传统程序有着显著的优越性，主要体现在以下三个方面。

(1) 更加灵活。充分利用了 C/C++ 语言指针的灵活性，能够根据需要及时动态地生成套接字对象，使得程序能够很好地适应网络环境的动态复杂性（尤其是在很多套接字进行复杂通信的场合），读者在第 4 章的网络聊天室应用实例中，将会深切体会到指针机制带来的这种好处。

(2) 代码更集中、更有条理。通过指针的链接作用，将程序中很多相关功能的代码集中有序地放置在同一个源代码文件的特定位置，减少了各个代码文件之间错综复杂的关联，使程序易于修改和维护，尤其是对于规模稍大、功能较多的网络程序。

(3) 程序语句更加简洁。使用指针机制使得套接字对象可以用获取主界面指针的方法，便捷地访问到主界面上的任何控件，代码简短、可读性好。

以界面控制代码为例，原来程序中的一段代码如下：

```
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->ServerIP.EnableWindow(false);
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->ServerPort.EnableWindow(false);
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonConnect.EnableWindow(false);
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonDisconnect.EnableWindow();
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_EditWords.EnableWindow();
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonSend.EnableWindow();
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonExit.EnableWindow(false);
((CChatClientDlg*)(AfxGetApp()->m_pMainWnd))->m_ButtonClear.EnableWindow();
```

用了指针机制编程后，简化为：

```
ServerIP.EnableWindow(false);
ServerPort.EnableWindow(false);
m_ButtonConnect.EnableWindow(false);
m_ButtonDisconnect.EnableWindow();
m_EditWords.EnableWindow();
m_ButtonSend.EnableWindow();
```

```
m_ButtonExit.EnableWindow(false);
m_ButtonClear.EnableWindow();
```

基于以上三条, 现在一般的网络程序都或多或少地引入了获取对话框指针的机制。在本书后面的实例中也都将采用这种经典的编程方式。

2.2.4 CSocket 类编程

在 2.1.2 节介绍过, MFC 中还有一个 Socket 类——CSocket, 并介绍了它可以与 CArchive、CSocketFile 类一起配合使用, 本节就用这种编程模式, 重新实现一遍前两节的进程通信程序。

1. 建立工程

分别创建客户端和服务端工程, 设置与前两节的程序完全一样, 创建完成后分别在双方工程中各添加一个基于 CSocket 基类的 MySocket 类, 如图 2.38 所示。

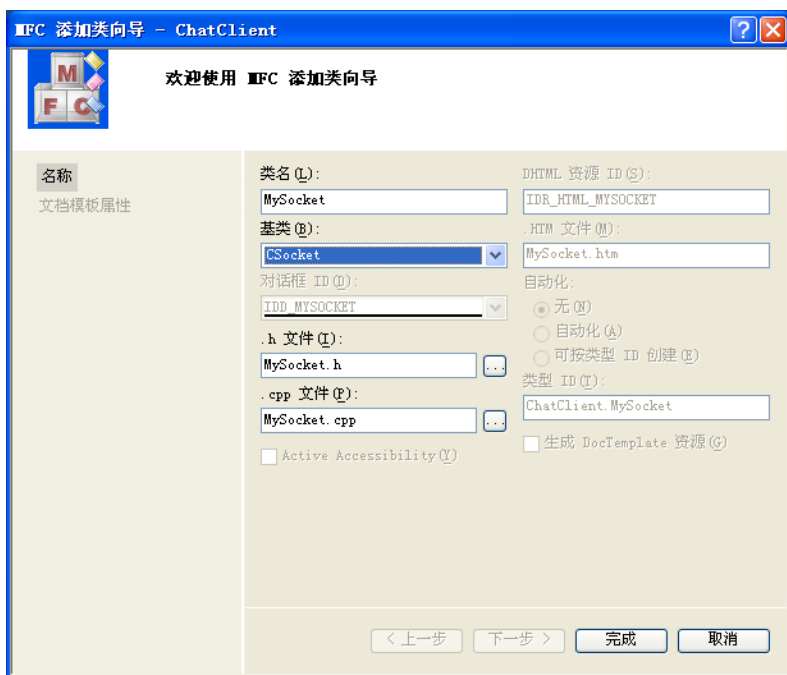


图 2.38 添加 CSocket 类

再对照前两节程序的界面给这个程序设计一个一模一样的界面, 仅仅在标题栏上标注“(CSocket 版)”表明这是基于 CSocket 类编写的程序版本。除此之外, 界面上所有的控件、控件属性、各控件关联的变量名及类型都与前面程序中的完全一样, 具体的设置见表 2.4 和表 2.5。

2. 建立程序框架

1) 客户端

在 MySocket.h 中添加如下代码:

```
#include "Afxsock.h"           //引用 Afxsock.h 头文件, 目的是使用 CSocket 类
class CChatClientDlg;         //为了能在 MySocket 类中定义主对话框类指针, 先对主对话框类进行前导声明
CChatClientDlg * m_dlg;       //定义主对话框指针
void GetDlg(CChatClientDlg * dlg); //获取主对话框指针的函数
```

给 MySocket 类添加网络事件处理函数 OnClose 和 OnReceive。

细心的读者通过比较前面两节的程序一定会问: 怎么这里没有添加 OnConnect 函数? 因为这

个程序是基于 CSocket 类编写的，而 CSocket 是同步 Socket 类，它在调用 Connect()、Send()、Accept()、Close()和 Receive()等成员函数时，要等到它们完成任务（连接被建立、数据被发送、连接请求被接收、Socket 被关闭、数据被读取）之后才会返回。因此，Connect 和 Send 不会导致 OnConnect 和 OnSend 被调用。对于 CSocket，处理网络事件通知的 OnAccept()、OnClose()、OnReceive()函数依然可以使用，但 OnConnect、OnSend 在 CSocket 中永远都不会被调用，即使添加了 OnConnect()函数，该函数中的代码也永远不会被执行。注意，这是基于 CSocket 类与 CAsyncSocket 类的网络程序最大的不同。这样一来，我们只能在主对话框程序“连接”按钮的事件过程中编写判断连接成功与否的代码，这在后面的示例中将会看到。

在 ChatClientDlg.h 中添加如下代码：

```
#include "MySocket.h"           //引用 MySocket 类的头文件，是为了下面定义一个 MySocket 类对象指针
MySocket * m_ClientSocket;     //客户端套接字指针
CArchive * m_archiveIn;       //收到的信息的存储文件
CArchive * m_archiveOut;      //发送的信息的存储文件
CSocketFile * m_socketfile;   //用于发送和接收数据的缓冲区
void OnReceive();
void OnClose();
void SocketReset();
```

定义变量 m_Input（如图 2.39 所示），用于后面接收对方发来的数据：

```
CString m_Input;
```

在 ChatClientDlg.cpp 中添加初始化代码：

```
m_ClientSocket = NULL;
m_archiveIn = NULL;
m_archiveOut = NULL;
m_socketfile = NULL;
```

图 2.40 中的初始化代码后面还有几行界面完善代码，读者此时可以顺手添加进去，以免后面忘了。

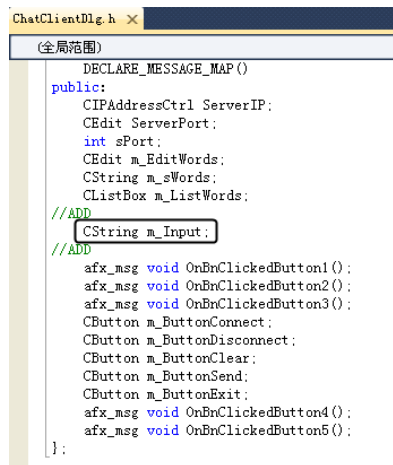


图 2.39 定义变量 m_Input

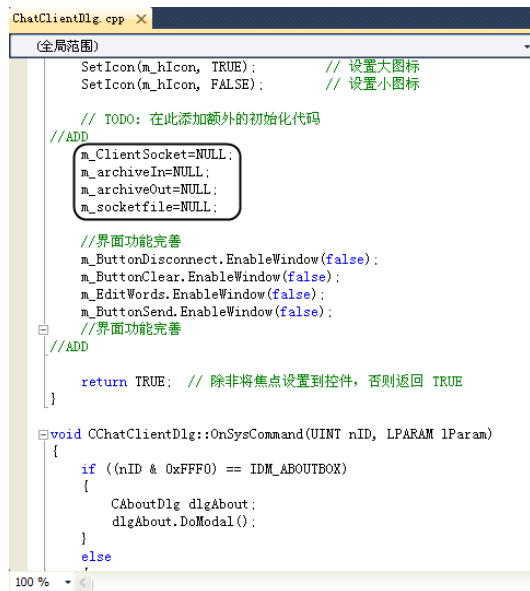


图 2.40 初始化代码

在 ChatClientDlg.cpp 中写出上面声明的函数 OnClose 和 OnReceive 的函数体, 为它们的实现代码预留出空间, 同时实现 SocketReset() 函数。代码如下:

```
void CChatClientDlg::SocketReset()
{
    if(m_archiveIn!=NULL)
    {
        delete m_archiveIn;
        m_archiveIn=NULL;
    }
    if(m_archiveOut!=NULL)
    {
        delete m_archiveOut;
        m_archiveOut=NULL;
    }
    if(m_socketfile!=NULL)
    {
        delete m_socketfile;
        m_socketfile=NULL;
    }
    if(m_ClientSocket!=NULL)
    {
        delete m_ClientSocket;
        m_ClientSocket=NULL;
    }
}
//SocketReset 函数实现

void CChatClientDlg::OnClose()
{
}
//OnClose()函数体

void CChatClientDlg::OnReceive()
{
}
//OnReceive 函数体
```

在 MySocket.cpp 中实现获取主对话框指针的 GetDlg()函数, 并通过指针引用主对话框程序代码中的网络事件处理函数:

```
#include "ChatClientDlg.h"
void MySocket::GetDlg(CChatClientDlg * dlg)    //获得窗口界面的指针
{
    m_dlg=dlg;
}
void MySocket::OnClose(int nErrorCode)
{
    //TODO: 在此添加专用代码和/或调用基类
    m_dlg->OnClose();
    CSocket::OnClose(nErrorCode);
}
void MySocket::OnReceive(int nErrorCode)
{
    //TODO: 在此添加专用代码和/或调用基类
    m_dlg->OnReceive();
    AsyncSelect(FD_CLOSE|FD_READ|FD_WRITE);
    CSocket::OnReceive(nErrorCode);
}
```

此处, AsyncSelect()函数的作用很重要, 因为 CSocket 类是同步类, 要求对方在通信中随时准备接收信息。使用 AsyncSelect()函数可以保证对方在连续发送数条信息时也照样可以正确收到。

2) 服务器

采用与客户端类似的方法, 在 MySocket.h 中添加如下代码:

```
#include "Afxsock.h"
class CChatServerDlg;
CChatServerDlg *m_dlg;
void GetDlg(CChatServerDlg *dlg);
```

给 Socket 类添加 OnAccept()、OnClose()、OnReceive()三个函数, 分别用于接收客户端连接、响应客户端突然断开的事件、接收客户端发来的数据。

在 ChatServerDlg.h 中添加如下代码:

```
#include "MySocket.h"
MySocket *m_ServerSocket;           //MySocket 类对象指针, 用于动态生成与客户端通信的 Socket
MySocket *m_ListenSocket;           //定义 MySocket 类对象指针, 用于动态生成监听 Socket
CArchive *m_archiveIn;
CArchive *m_archiveOut;
CSocketFile *m_socketfile;
void OnReceive();
void OnClose();
void OnAccept();
void SocketReset();
```

在 ChatServerDlg.cpp 中添加如下初始化代码:

```
m_ServerSocket = NULL;
m_ListenSocket = NULL;
m_archiveIn = NULL;
m_archiveOut = NULL;
m_socketfile = NULL;
```

在 ChatServerDlg.cpp 中写出 OnAccept()、OnClose()、OnReceive()三个函数的函数体, 为它们的实现代码预留出空间, 同时实现 SocketReset 函数, 代码如下:

```
void CChatServerDlg::SocketReset()
{
    if(m_archiveIn!=NULL)
    {
        delete m_archiveIn;
        m_archiveIn=NULL;
    }
    if(m_archiveOut!=NULL)
    {
        delete m_archiveOut;
        m_archiveOut=NULL;
    }
    if(m_socketfile!=NULL)
    {
        delete m_socketfile;
```

```

        m_socketfile=NULL;
    }
    if(m_ServerSocket!=NULL)
    {
        delete m_ServerSocket;
        m_ServerSocket=NULL;
    }
}
void CChatServerDlg::OnClose()        //OnClose()函数体
{
}
void CChatServerDlg::OnReceive()      //OnReceive()函数体
{
}
void CChatServerDlg::OnAccept()       //OnAccept()函数体
{
}

```

在 MySocket.cpp 中实现获取主对话框指针的 GetDlg()函数, 并通过指针引用主对话框的三个函数, 代码如下:

```

#include "ChatServerDlg.h"
void MySocket::GetDlg(CChatServerDlg *dlg)    //获得窗口界面的指针
{
    m_dlg=dlg;
}
void MySocket::OnClose(int nErrorCode)
{
    //TODO: 在此添加专用代码和/或调用基类
    m_dlg->OnClose();
    CSocket::OnClose(nErrorCode);
}
void MySocket::OnReceive(int nErrorCode)
{
    //TODO: 在此添加专用代码和/或调用基类
    m_dlg->OnReceive();
    AsyncSelect(FD_CLOSE|FD_READ|FD_WRITE);
    CSocket::OnReceive(nErrorCode);
}
void MySocket::OnAccept(int nErrorCode)
{
    //TODO: 在此添加专用代码和/或调用基类
    if(m_dlg->m_ServerSocket==NULL)
        m_dlg->OnAccept();
    CSocket::OnAccept(nErrorCode);
}

```

同理, 这里也用到了 AsyncSelect()函数以随时接收同步数据。

3. 代码解析

1) 客户端

“连接”按钮的事件过程代码如下:

```

if(!AfxSocketInit())

```

```

{
    MessageBox("WindowSocket initial failed!", "Receive", MB_ICONSTOP);
    return;
}
m_ClientSocket = new MySocket;
m_ClientSocket->GetDlg(this);
m_ClientSocket->Create();           //创建客户端 Socket
BYTE nFild[4];
CString sIP;
UpdateData();
ServerIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
if(!m_ClientSocket->Connect(sIP,sPort))    //发起连接请求
{
    AfxMessageBox("连接失败，请您重试！");
    return;
}
else
{
    m_ListWords.AddString("连接服务器成功！");
    m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
    //创建一个和 CSocket 对象关联的 CSocketFile 对象
    m_socketfile=new CSocketFile(m_ClientSocket);
    //创建和 CSocketFile 对象关联的 CArchive 对象，指定 CArchive 对象是用于读还是写
    //load 用来接收(读)数据
    m_archiveIn=new CArchive(m_socketfile,CArchive::load);
    //store 用来发送(写)数据
    m_archiveOut=new CArchive(m_socketfile,CArchive::store);
}
}

```

可以看到，这里将确认连接是否成功的代码一并放在了“连接”按钮的事件过程中，另外，根据 2.1.2 节所述的编程模式，将 CSocket 与 CArchive、CSocketFile 类关联起来，这样就可以使用它们相互配合来管理数据的收发了。

先创建一个和 CSocket 对象关联的 CSocketFile 对象，再创建和 CSocketFile 对象关联的 CArchive 对象，指定 CArchive 对象用于读还是写。本程序因为要能同时发送数据和接收数据，即既要读又要写，因此须创建两个 CArchive 对象，其中 m_archiveIn 用于接收数据，m_archiveOut 用于发送数据。

“断开”按钮事件过程代码如下：

```

SocketReset();
m_ListWords.AddString("从服务器断开");

```

“发送”按钮事件过程代码如下：

```

UpdateData();
*m_archiveOut<<m_sWords;
m_archiveOut->Flush();
m_ListWords.AddString("发送: " + m_sWords);
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);

```

这里不使用 Send()函数，而是采用了新的输出数据的方法，即应用前面刚刚建立关联的 CArchive 对象。

OnClose()函数代码如下：

```
m_ListWords.AddString("服务器断开了");
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
SocketReset();
```

OnReceive()函数代码如下：

```
*m_archiveIn>>m_Input;
m_archiveIn->Flush();
m_ListWords.AddString("收到: " + m_Input);
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
```

接收数据时也使用 CArchive 对象。

2) 服务器

“开始监听”按钮事件过程如下：

```
if(!AfxSocketInit())                //初始化套接字
{
    MessageBox("WindowSocket initial failed!", "Send", MB_ICONSTOP);
    return;
}
m_ListenSocket = new MySocket;
m_ListenSocket->GetDlg(this);
BYTE nFild[4];
CString sIP,sP;
UpdateData();
ServerIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
sP.Format("%d",sPort);
m_ListenSocket->Create(sPort,1,sIP);    //创建服务器监听 Socket
m_ListenSocket->Listen(1);             //开始监听
m_ListWords.AddString("监听开始: ");
m_ListWords.AddString("地址" + sIP + " 端口" + sP);
m_ListWords.AddString("等待客户端连接……");
```

只是前面多了一段初始化代码，还有就是改用指针运算符调用 Socket 函数。

“停止监听”按钮事件过程如下：

```
if(m_ListenSocket!=NULL)
{
    delete m_ListenSocket;
    m_ListenSocket = NULL;
}
m_ListWords.AddString("停止监听");
```

直接释放监听 Socket 的指针，这样就不会影响已经建立好连接的 Socket。

“断开”按钮的事件过程代码如下：

```
SocketReset();
m_ListWords.AddString("与客户端断开");
```

SocketReset 函数只单独释放与客户端通信的指针及其相应的 CArchive 对象，而监听 Socket

仍在运行，这样就可以随时再接收新的连接请求。

“发送”按钮的事件过程代码如下：

```
UpdateData();
*m_archiveOut<<m_sWords;
m_archiveOut->Flush();
m_ListWords.AddString("发送: " + m_sWords);
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
```

这里一律使用 CArchive 对象进行发送。

OnAccept()函数代码如下：

```
m_ServerSocket = new MySocket;
m_ServerSocket->GetDlg(this);
m_ListenSocket->Accept(*m_ServerSocket);
m_ServerSocket->AsyncSelect(FD_READ|FD_CLOSE);
m_socketfile = new CSocketFile(m_ServerSocket);
m_archiveIn = new CArchive(m_socketfile,CArchive::load);
m_archiveOut = new CArchive(m_socketfile,CArchive::store);
m_ListWords.AddString("接收了一个客户端的连接请求");
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
```

这里接收客户端连接前先使用 new MySocket 为这个连接创建通信用的套接字，然后再用 Accept 将这个套接字绑定到连接上，并且使用 AsyncSelect 随时等候读取收到的数据。再下面的三行就是将 CSocket、CArchive 和 CSocketFile 三个类相关联的语句，将它们关联起来就可以相互配合进行数据的收发。

OnClose()函数代码如下：

```
m_ListWords.AddString("客户端断开连接");
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
SocketReset();
```

OnReceive()函数代码如下：

```
*m_archiveIn>>m_Input;
m_archiveIn->Flush();
m_ListWords.AddString("收到: " + m_Input);
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
```

再来看传统的使用 Receive 函数接收数据的代码：

```
//接收客户端发来的信息
char szTemp[200];
int n = m_ServerSocket->Receive(szTemp,200); //接收信息
szTemp[n] = '\0';
CString sTemp;
sTemp.Format("收到: %s",szTemp);
m_ListWords.AddString(sTemp); //显示信息
m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
```

直接使用 Receive()函数接收数据，需要自己定义字符数组作为缓冲区并自行处理转换接收到数据的格式，而通过配合 CArchive 和 CSocketFile 类接收数据是不是要简单很多呢？对，由于 CArchive 和 CSocketFile 类封装了对收到的数据进行序列化处理的细节，因此使得接收任何网络数据都如同读取本地文件一样简单，尤其在接收较复杂的数据信息类型时，这种优越性更能体现出来。

2.3 Socket 程序的互通

2.3.1 不同版本 Socket 程序的互通

在 2.2 节介绍了分别使用 MFC 的两个不同 Socket 类进行网络编程，并且用不同的编程机制实现了一个程序的三个版本，那么这几种不同版本的程序之间可不可以混合通信、相互收发信息呢？换言之，基于 CAsyncSocket 编写的客户端可不可以和基于 CSocket 类的服务器通信？反过来呢？

大家知道，这两个类实现的程序中收发数据的机制是不同的，CAsyncSocket 类的传统程序使用异步方式直接调用 Socket 函数 Send、Receive 收发数据，并且对收发的信息自行处理，而 CSocket 则采用与 CArchive、CSocketFile 类配合的方法管理数据的收发，处理收发数据的过程对用户来说是透明的。若这两类不同收发机制的程序互相通信会产生什么现象呢？下面就让我们来看看。

1. CAsyncSocket 客户端连接 CSocket 服务器

运行前面编写的传统 CAsyncSocket 客户端（用指针机制的版本也一样，这是因为指针机制的程序只是代码的组织方式不同而数据收发机制并没有变），同时运行 CSocket 版的服务器。

它们可以正常连接上（如图 2.41 所示），这是因为 CSocket 是从 CAsyncSocket 派生的类，它们使用的是同一套 Socket 系统，用 Connect 发起连接和用 Accept 接收连接，只要 IP 地址和端口号填写正确，就能成功连上。

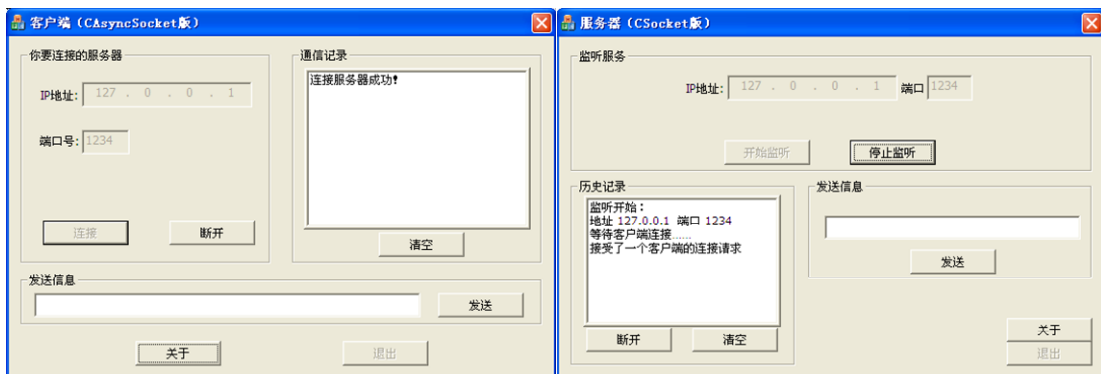


图 2.41 CAsyncSocket 客户端连接 CSocket 服务器

接着从客户端向服务器发信息，会发现服务器无法正常接收，客户端连发 4 条信息服务器都没有反应，更糟的是，服务器死机（界面上的任何按钮控件都如同锁死了一般不接收用户的操作），如图 2.42 所示。

不过不要慌，客户端这边还是可以正常操作的，我们从客户端断开连接，看看能不能解救服务器的死锁。

单击客户端的“断开”按钮，服务器弹出出错提示（如图 2.43 所示）。从字面上看，好像是说访问文件出错了，对于这个错误的深层机制读者暂且不用管，但还是可以大致感性地猜猜是怎么回事——因为我们前面实现 CSocket 的输入/输出时，是使用 CArchive 和 CSocketFile 配合将收

发数据操作转换为类似文件的读/写操作，这说明问题出在服务器的输入/输出机制与客户端不匹配上。可见使用同步收发机制的服务器，要求客户端也必须以同步方式发送数据，它才能正常接收。单击“确定”按钮忽略这个错误后，可以看到服务器从死机状态解脱出来，能正常操作了。

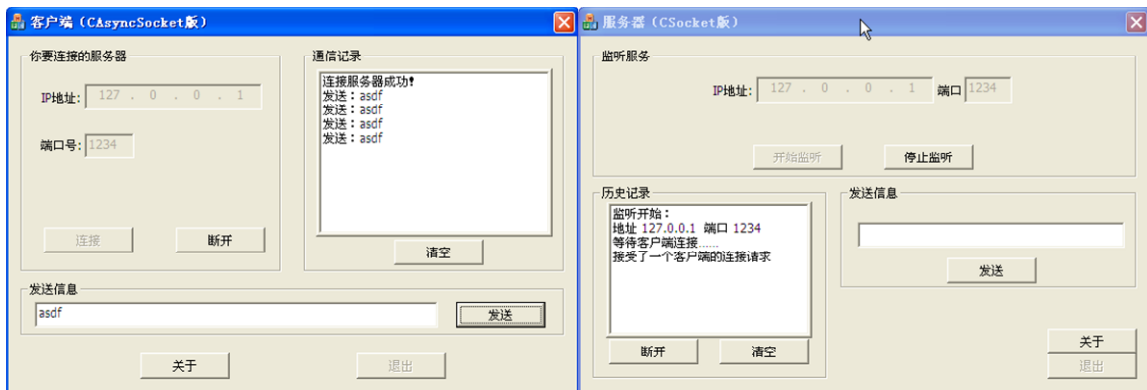


图 2.42 服务器死机

再看看服务器向客户端发送数据会怎样，不过请读者注意了：在反过来从服务器向客户端发送数据之前，必须先先在服务器界面上单击“断开”按钮，将服务器这边上次通信使用的 Socket 关闭，否则后面的操作都将无法进行并出错，双方就再也不能重新建立连接了！

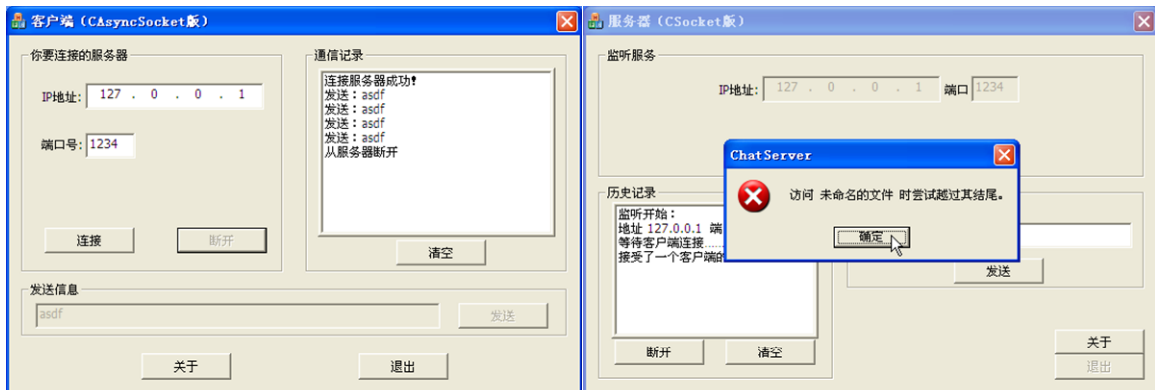


图 2.43 死锁解除后服务器弹出出错提示

先关闭服务器上上次遗留的废弃 Socket，重新建立连接后，再从服务器向客户端发送信息（如图 2.44 所示）。

竟然可以收到信息！不过美中不足的是，客户端收到的每条信息之前都有一个字符的随机乱码符号，但后面跟的信息却是完整可读的，至少在这样的情况下双方可以互发信息通信，也能读懂收到的信息。

2. CSocket 客户端连接 CAsyncSocket 服务器

把通信双方的程序版本互换一下，客户端采用 CSocket 版，服务器采用 CAsyncSocket 版，看看接下来会发生什么。

客户端向服务器可以成功发送，如图 2.45 所示，但和前面的一样有瑕疵，收到的信息前有一个乱码字符。

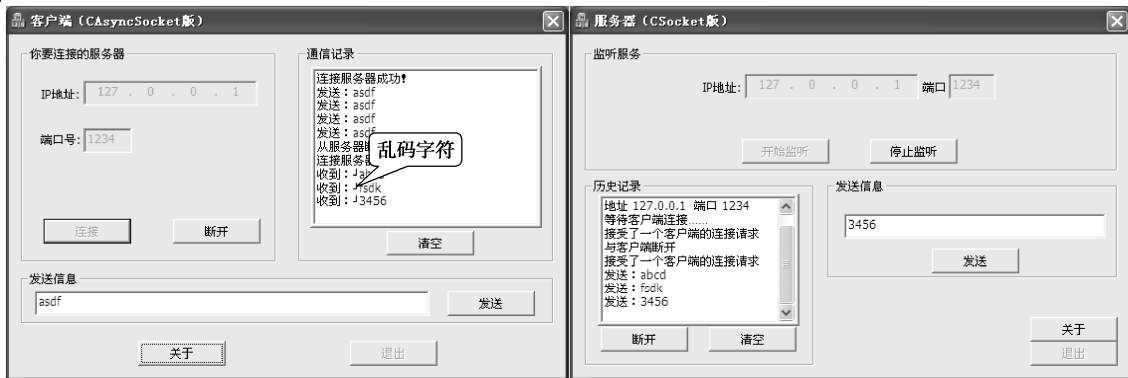


图 2.44 重新连接后从服务器向客户端发送信息

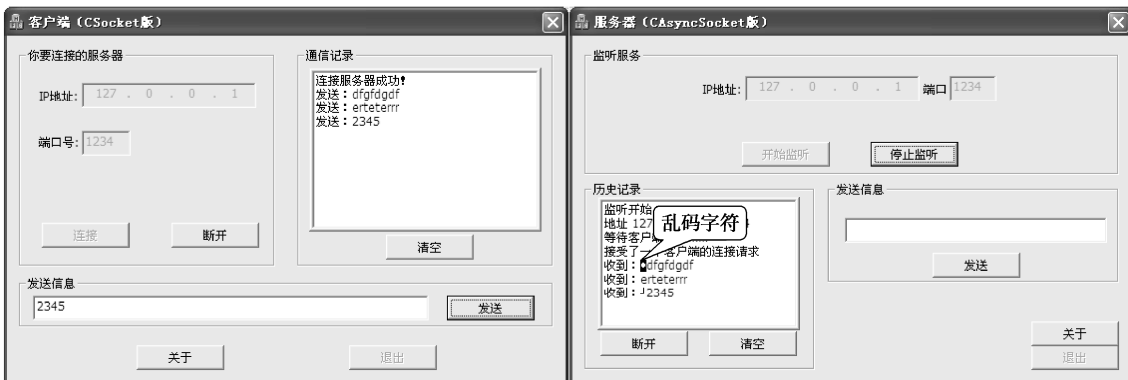


图 2.45 CSocket 客户端连接 CAsyncSocket 服务器

服务器向客户端的发送同样会产生错误（读者自己试一下），这个结果在我们的预料之中。可见每当使用 CArchive 对象同步接收数据时，如果对方是直接使用 Send 函数发送的，就会导致接收方死机，直至发送方断开连接；而当使用 CArchive 对象同步发送数据时，即使对方使用异步方式直接调用 Receive 函数，也照样能收到可读的完整信息。

3. 结论

针对上面实验结果的总结见表 2.6。

表 2.6 采用不同输入/输出模型的 Socket 程序之间的互通

发送方输出模型	接收方输入模型	通信结果
CArchive 类同步	Receive 异步	√（但消息前面会带有一个不确定的乱码符号）
CArchive 类同步	CArchive 类同步	√
Send 异步	CArchive 类同步	×（会导致对方死机）
Send 异步	Receive 异步	√

在此我们指出这样一个事实：绝大多数网络软件的底层 Socket 无非就是采用上述两种输入/输出机制。可见，若发送消息采用 CArchive 而接收采用 Receive，则这样的客户端在理论上可以与任何不同实现的网络软件通信而不会发生错误。更进一步，如果编写一个程序使用户能够通过界面设置在两种输入/输出模式之间灵活地切换，那么用户就可以通过测试与任意第三方程序通信，根据程序运行时的现象，由表 2.6 确定对方采用的收发模式，从而设置自身的收发模式以适应对方，这样的软件将具有很强的适应性，能够与任何实现方式的网络程序交互。

2.3.2 接入第三方 Socket 程序

根据上述一系列实验得出的结论，我们可以做这样的设想：如果得到了他人编写的一个网络程序（如聊天室），不管它内部是如何实现的，更不用去看它的源代码，能不能利用前面实验得出的规律，尝试用自己编写的 Socket 程序与它对接上并且还能正常地通信呢？下面我们就来做这样的尝试！

从网上下载一个公开的他人编写（第三方）的聊天室软件，程序运行效果如图 2.46 所示。

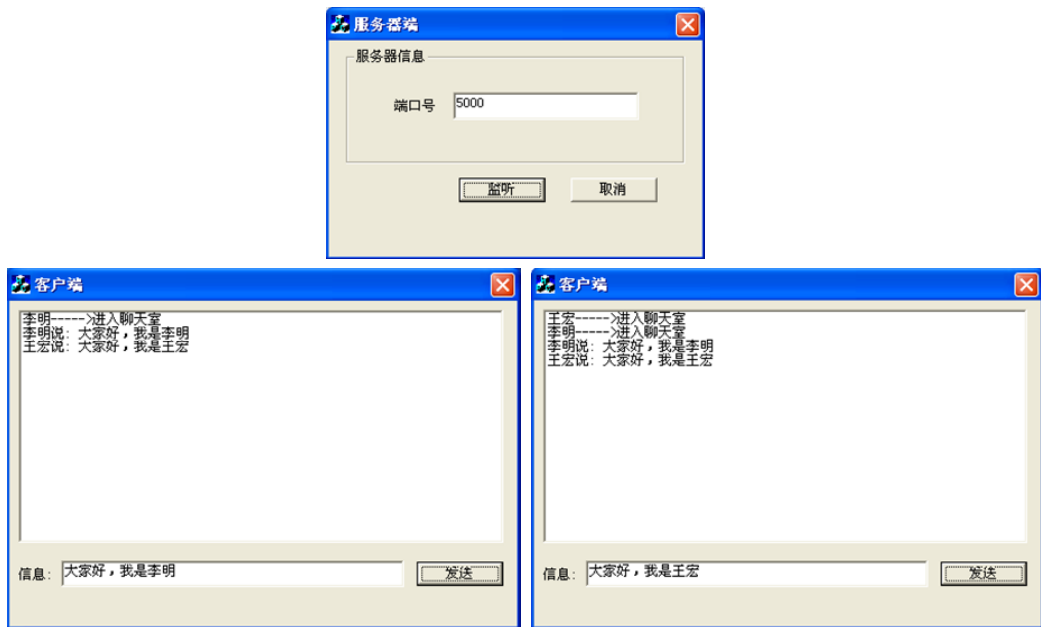


图 2.46 第三方聊天室软件

服务器在本地的 5000 号端口上监听，接收登录的多个客户端的连接，各个客户端以服务器为枢纽转发消息，在每个客户端上可以显示出其他客户端发来的信息。

现在，我们不采用这个聊天室软件本身提供的配套客户端，而是用我们自己在 2.2 节里编写的那些不同实现版本的 Socket 客户端尝试着接入这个聊天室。

先使用 CSocket 版的客户端接入，如图 2.47 所示。

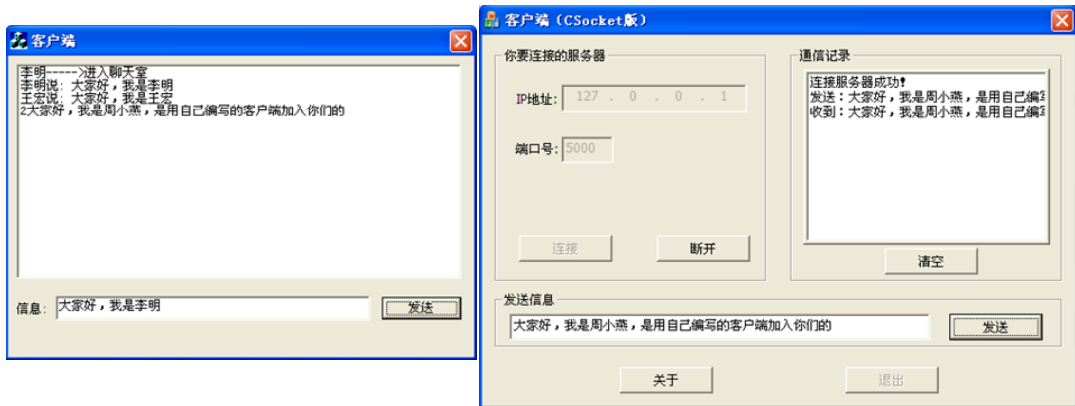


图 2.47 CSocket 版客户端接入

接入成功！发送信息给聊天室里的其他成员，他们都成功地收到信息了！

由于我们自己的客户端使用 `CArchive` 发送，从其他客户端收到的信息前有个乱码字符“2”，根据 2.3.1 节的实验结论表(表 2.6)，就可以知道这个聊天室软件客户端采用的是 `Receive` 异步接收方式。

当有其他客户端向我们自己的客户端发信息时，我们自己的客户端收不到他们的问候，而且死机了。根据前面实验的结论可以推断出，这个聊天室软件客户端采用的发送模式是 `Send` 异步方式，至此可以判断它使用的收发模式都是直接调用 `Socket` 函数的异步方式，如果我们也采用传统的异步收发客户端与它连接，也许就可以畅通无阻地通信。

事实果真如此吗？读者不妨一试！

Winsock API 编程

第 2 章系统地介绍了 MFC Socket 编程，程序员只要直接调用套接字类提供的方法（如 Create()、Listen()、Connect()、Accept()），就可以轻松地在进程之间建立连接，并用 Send()和 Receive()方法顺利地实现进程间的消息通信——这都是由于 MFC 的套接字类封装了 Winsock API 的缘故。

3.1 Winsock API 编程原理

3.1.1 通行的操作

用 Winsock API 编程时，主要进行以下一些通行的操作步骤。

1. Winsock 的打开 (WSAStartup())

在使用 Winsock API 编制网络程序时，要使用大量系统已经实现了的网络功能函数，在调用任何一个 Winsock API 函数之前，都必须先检查协议栈的安装情况，即系统中是否有 Windows Socket 的实现库，这个操作又被称为“Winsock 的打开”。

Winsock 的打开使用 WSAStartup()函数来实现，因为 Winsock 的服务是以动态链接库形式实现的，因此必须先对 Winsock DLL 进行初始化。通过调用 WSAStartup()函数便可检测系统中有没有一个或多个 Windows Socket 的实现，本函数必须是应用程序或 DLL 调用的第一个 Windows Socket 函数，它允许应用程序或 DLL 指明 Winsock API 的版本号及获得特定版本 Winsock 实现的细节。应用程序或 DLL 只有在一次成功的 WSAStartup()调用之后，才能进一步调用其他的 Winsock API 函数。

2. 建立套接字 (socket()或 WSASocket())

应用程序在使用套接字通信之前，必须拥有一个套接字。在 Winsock 中，要使用 socket()或 WSASocket()函数来给网络程序创建一个套接字。在客户端和服务端都可以调用 socket()函数来建立一个套接字，两者的区别在于：客户端的套接字可以调用 bind()函数，由自己来指定 IP 地址及 port 号，也可以不调用 bind()函数，由 Winsock 来自动设定 IP 地址及 port 号；服务器套接字则必须使用 bind()函数绑定。

3. 地址绑定 (bind())

当用 socket()函数创建了一个套接字后，该套接字还是不能直接使用的，因为它只存在于—

个名字空间（地址族）中，只确定了通信所希望使用的服务类型，并没有与该主机上提供服务的某端口联系在一起，这样的套接字叫未命名套接口。`bind()`函数通过给一个未命名套接口分配一个本地名字来为它建立本地绑定（即把一个套接口与一个主机地址和端口号联系起来）。

`bind()`函数适用于数据报或流式套接字，用在数据报套接字时，通信双方都必须进行地址绑定，而在流式套接字应用中，由于一个服务器接收诸多客户端连接，无须识别每一个客户端，但客户端必须清楚自己想要连接的服务器地址，因此服务器需要绑定地址，而客户端不需要。

4. 服务器监听连接 (`listen()`)

当在一个服务器程序中成功创建了一个套接字，并用 `bind()`函数和一个指定的地址关联（绑定）在一起后，就要指示该套接字进入监听连接请求的状态，以接收由客户端发出的请求，这时就要用函数 `listen()`，它应用在面向连接的通信过程中。

5. 客户端提出连接申请 (`connect()`或 `WSAConnect()`)

在客户端建立好套接字之后，就要调用 `connect()`函数或 `WSAConnect()`函数，提出与一个服务器建立连接的请求，如果服务器接收请求，就可以在服务器的远程套接字与客户端的本地套接字之间建立一条连接。这也是面向连接的流式套接字特有的函数。

6. 服务器接收客户端的连接请求 (`accept()`或 `WSAAccept()`)

服务器进入监听状态后，用 `accept()`或 `WSAAccept()`函数接收来自客户端由 `connect()`（`WSAConnect()`）发出的连接请求，就好像打电话时被叫方听到电话铃声后提起话筒的动作，然后双方进入通话状态。只有在面向连接的通信中才需要这一步操作。

7. 数据的发送 (`send()`或 `WSASend()`, `sendto()`或 `WSASendTo()`)

前面说过，套接字包括面向连接的流式套接字和无连接的数据报套接字，对于流式套接字来说，要从套接字上发送一个数据报，就要使用 `send()`或 `WSASend()`函数，而无连接的数据报套接字则使用 `sendto()`或 `WSASendTo()`函数。这两组函数在原型参数上有差异，故一般编程只在同类套接字之间传递信息，不同类的套接字是无法通信的。

8. 数据的接收 (`recv()`或 `WSARecv()`, `recvfrom()`或 `WSARecvfrom()`)

与数据的发送相对应，针对流式或数据报套接字，接收数据也分别采用两组不同的函数。流式套接字用 `recv()`或 `WSARecv()`函数，数据报套接字用 `recvfrom()`或 `WSARecvfrom()`函数。

9. 关闭套接字 (`closesocket()`)

在网络程序中，一个套接字不再使用时一定要及时地关闭，以释放与其关联的所有资源，包括等候处理的数据。关闭服务器和客户端的通信连接是很简单的，这一过程可以由任意一方发起，只要调用 `closesocket()`函数就可以了。而要关闭服务器监听状态的套接字，同样也是利用此函数。

10. 关闭 Winsock (`WSACleanup()`)

当应用程序不再使用 Winsock API 中的任何函数时，必须调用 `WSACleanup()`函数将其从 Windows Socket 的实现中注销，以释放资源。因此，对应于一个任务进行的每一次 `WSAStartup()`函数调用，必须有一个 `WSACleanup()`调用，因为每次 `WSAStartup()`函数的调用都会增加加载 Winsock DLL 的引用次数，这就要求调用同样多次的 `WSACleanup()`以抵消引用。

在稍后的编程中大家会发现，有很多函数带有形如“WSA”的前缀，如 `WSAStartup()`、`WSASocket()`、`WSAConnect()`、`WSAAccept()`函数等，这是 Winsock 2 对原来 Winsock 1 的扩展，扩展版本的函数与原来 Winsock 的函数是完全兼容的，只是增加了新的功能而并没有改变原来的功能和用法。原来函数的参数种类、个数和含义都保持不变，可以和“WSA”打头的函数通用。比如，在程序中调用 `send()`与调用 `WSASend()`的效果是完全一样的，都是在流式套接字上发送数据。本章的程序示例中不再对此加以区分。

3.1.2 Winsock API 函数详解

以下将依次详细介绍上述每个函数的原型、参数含义和功能。由于 Winsock API 功能强大，其中不少参数的设定在一般简单的编程（包括本书）中根本用不到，这里只是为了使本书的体系更加完整和有条理，同时也为那些想深入了解 Winsock 的读者提供资料，故在此对各 Winsock API 函数逐一详解。建议初学者跳过本段，等到阅读后面的实例程序代码时，对照其中用到的 Winsock 函数功能再来看这个详解，这样更好理解。希望大家照此去做！

WSAStartup()原型为：

```
int WSAStartup(WORD wVersionRequested,LPWSADATA lpWSADATA);
```

其中，参数 wVersionRequested 为一个 WORD 型数值，用以表示使用的 Winsock 的版本，其高位字节指定副版本，低位字节指定主版本；lpWSADATA 是一个指向 WSADATA 结构的指针，用来存储系统传回的关于 WinSock 的资料。

WSADATA 结构的定义（在头文件 WinSock2.h 中）如下：

```
typedef struct WSADATA {
    WORD        wVersion;                //希望使用的版本号
    WORD        wHighVersion;
    char        szDescription[WSADESCRIPTION_LEN+1];
    char        szSystemStatus[WSASYS_STATUS_LEN+1];
    unsigned short iMaxSockets;
    unsigned short iMaxUdpDg;
    char FAR*    lpVendorInfo;
} WSADATA, FAR * LPWSADATA;
```

WSAStartup()函数的返回值是一个整数，如果调用成功返回值为 0。如果函数调用失败，则根据失败类型返回不同的错误信息，如下所示。

- **WSASYSNOTREADY**：本关键字的值为 10091，表示加载的 Winsock DLL 不存在或者底层的网络子系统无法使用。
- **WSAVERNOTSUPPORTED**：本关键字的值为 10092，表示系统所需要的 Windows Socket API 的版本不能由 Windows Socket 实现提供，如果用户不能接受由 wVersion 返回的版本，则要调用 WSACleanup()函数清除对该 Winsock 的加载。
- **WSAEINVAL**：值为 10022，表示应用程序指定的 Windows Socket 版本不能被该 Winsock DLL 的实现所支持。
- **WSAEINPROGRESS**：值为 10036，表示有一个阻塞的 Winsock 调用正在进行。
- **WSAEPROCLIM**：值为 10067，表示任务数量已经达到 Windows Socket 实现所能支持的最大限制。
- **WSAEFAULT**：值为 10014，表示参数 lpWSADATA 是一个无效的指针。

socket()函数原型为：

```
SOCKET socket(int af,int type,int protocol);
```

参数说明如下。

- **af**：说明套接口要使用的协议地址族，目前只提供 AF_INET，表示使用 Internet (IP) 协议地址。
- **type**：描述套接字的类型，只能是 SOCK_STREAM、SOCK_DGRAM、SOCK_RAW 三个协议类型中的一个，分别表示流套接字、数据报套接字和原始套接字（本书不涉及这种

套接字)。

- **protocol**: 该套接字使用的特定通信协定 (如果使用者不指定则设为 0)。

此函数调用成功返回套接字对象, 失败则返回 `INVALID_SOCKET`, 应用程序可以调用 `WSAGetLastError()` 函数得知具体的错误原因, 所有 Winsock 的 API 函数都可以使用这个函数来获取失败的原因。下面给出几种错误类型 (与前述相同的类型在此不再一一列举, 下面的例子相同)。

- **WSANOINITIALISED**: 表示在调用本 API 之前未能成功调用 `WSAStartup()` 函数。
- **WSAEAFNOSUPPORT**: 不支持指定的地址族。
- **WSAEMFILE**: 表示没有可以使用的套接字。
- **WSAENOBUFFS**: 表示没有可以使用的缓冲区, 因此无法创建套接字。
- **WSAEPROTONOSUPPORT**: 不支持指定的协议。
- **WSAEPROTOTYPE**: 表示指定的协议与本套接字类型不匹配。
- **WSAESOCKTNOSUPPORT**: 表示本地址族不支持指定的套接字类型。

`bind()` 原型为:

```
int bind(SOCKET s, const struct sockaddr FAR *name, int namelen);
```

参数说明如下。

- **s**: 表示未绑定的套接字的对象名, 它是 `socket()` 函数调用成功时返回的值。
- **name**: 套接字的地址值, 是一个与指定协议有关的地址结构指针, 这个地址必须是执行这个程序所在计算机的 IP 地址。在 Winsock 中使用 `sockaddr_in` 结构指定 IP 地址和端口信息, 定义如下:

```
struct sockaddr_in{
    short        sin_family;
    u_short      sin_port;
    struct in_addr sin_addr;
    char         sin_zero[8];
}
```

其中, `sin_family` 必须为 `AF_INET`, 指明使用的是 IP 地址族; `sin_port` 和 `sin_addr` 分别是以网络字节顺序表示的 16 位端口号和 32 位 IP 地址; `sin_zero` 字段一般为 0。

- **namelen**: 地址参数 `name` 的长度。如果使用者不在意地址或端口的值, 那么可以设定地址为 `INADDR_ANY` 及 Port 为 0, Windows Socket 会自动将其设定为适当地址并指定 Port (1024~5000 之间) 的值。此后可以调用 `getsockname()` 函数来获知其被设定的值。

`bind()` 函数调用成功返回 0, 否则返回 `SOCKET_ERROR`。同样, 根据错误类型的不同, 应用程序可以使用 `WSAGetLastError()` 函数获取具体的错误代码, 下面给出几种错误。

- **WSAEADDRINUSE**: 表示所指定的端口已经在使用中, 不能立即获取。
- **WSAEFAULT**: 表示参数 `namelen` 太小, 小于 `sockaddr` 结构的大小。
- **WSANOTSOCK**: 指定的描述字不是一个套接字。

`listen()` 原型为:

```
int listen(SOCKET s, int backlog);
```

`listen()` 函数使服务器的套接字进入监听状态, 并设定可以建立的最大连接数 (目前最大值限制为 5, 最小值为 1)。该函数调用成功返回 0, 否则返回 `SOCKET_ERROR`。

参数说明如下。

- **s**: 表明一个已经绑定了地址、需要建立监听的套接字。

- **backlog**: 指定正在等待连接的最大连接个数。

connect()函数原型为:

```
int PASCAL FAR connect(SOCKET s, const struct sockaddr FAR *name, int namelen);
```

函数调用成功返回 0, 否则返回 SOCKET_ERROR。

参数说明如下。

- **s**: Socket 的识别码。
- **name**: 一个指向远端套接字地址结构 `sockaddr_in` 的指针, 表示 s 套接字欲与其建立一条连接。
- **namelen**: name 的长度。

accept()函数原型为:

```
SOCKET accept(SOCKET s, struct sockaddr FAR *addr, int FAR *addrlen);
```

当客户端提出连接请求时, 服务器会收到 Winsock Stack 送来的一个消息, 这时程序对 IPParam 参数进行分析, 然后调用 **accept()**函数, 该函数新建一个套接字与客户端的套接字相通, 原先监听的套接字继续处于监听状态, 等待其他客户端的连接要求。该函数调用成功返回一个新产生的套接字对象, 否则返回 INVALID_SOCKET。

参数说明如下。

- **s**: Socket 的识别码。
- **addr**: 存放要连接的客户端的地址。
- **addrlen**: addr 的长度。

send()函数原型为:

```
int send(SOCKET s, const char FAR *buf, int len, int flags);
```

sendto()函数原型为:

```
int sendto(SOCKET s, const char FAR *buf, int len, int flags, const struct sockaddr FAR* to, int tolen);
```

这两个函数调用成功返回发送的数据的长度, 否则返回 SOCKET_ERROR。

参数说明如下。

- **s**: Socket 的识别码。
- **buf**: 存放要发送的数据的暂存区。
- **len**: buf 的长度。
- **flags**: 此函数被调用的方式。

recv()函数原型为:

```
int recv(SOCKET s, char FAR *buf, int len, int flags);
```

recvfrom()函数原型为:

```
int recvfrom(SOCKET s, char FAR *buf, int len, int flags, struct sockaddr FAR* from, int FAR* fromlen);
```

这组函数与上一组发送函数一样, 各参数对应的意义相同。只是这里的 **buf** 是用来存放接收到的数据的, 函数最终返回的也是接收到的数据的长度。

closesocket()函数原型为:

```
int closesocket(SOCKET s);
```

这个函数很简单, 参数只有一个, 即要关闭的套接字的识别码。函数调用成功返回 0, 否则返回 SOCKET_ERROR。

WSACleanup()函数原型为:

```
int WSACleanup(void);
```

对应每个 **WSAStartup()**函数, 必须有一个 **WSACleanup()**函数的调用。该函数是一个无参

(void) 函数, 函数调用成功返回 0, 调用失败时根据失败类型返回对应的错误信息, 主要有以下几类。

- **WSANOTINITIALISED**: 使用本 API 前必须进行一次成功的 **WSAStartup()** 函数调用。
- **WSAENETDOWN**: Windows Socket 的实现检测到网络子系统故障。
- **WSAEINPROGRESS**: 一个阻塞的 Windows Socket 操作正在进行。

以上这些函数的用法和它们各个参数的意义读者都不需要记忆, 甚至现在都可以不用看, 更不用去理解, 最好的方法是在后面编程时用到了某个函数的某个参数, 再回过头来看上面的这些描述说明文字, 就会豁然开朗了。

3.1.3 TCP 与 UDP

众所周知, Internet 上传统且应用最为广泛的编程模式是 C/S 模式, 而 Winsock API 在当初就是面向 C/S 设计的。因此用其实现的程序要有客户端 (C) 和服务端 (S) 两个不同类型的进程。

在剥去 MFC 的封装后, 大家会发现 Socket 程序其实还分为两大类。根据使用传输层协议 (创建套接字类型) 的不同, 双方的通信交互方式也不一样。

对于面向连接的 TCP, 交互双方的进程各自建立一个流式套接字, 服务器需要等待客户端向其提出建立连接的申请。一旦接收到客户端的连接请求, 服务器即返回一个新的套接字描述符, 通过该描述符调用数据传输函数与客户端进行数据的收发。

对于无连接的 UDP, 双方建立的是数据报套接字, 服务器和客户端在传输数据之前不需要进行连接的申请和建立, 可以随时随地向对方发消息, 任何一方想结束通信, 直接关闭套接字即可。

这样一来, Socket 程序就分为 TCP 程序和 UDP 程序两种, 它们的差别比较大, 应用场合也不一样, 图 3.1 描绘了这两类程序各自的函数调用流程。

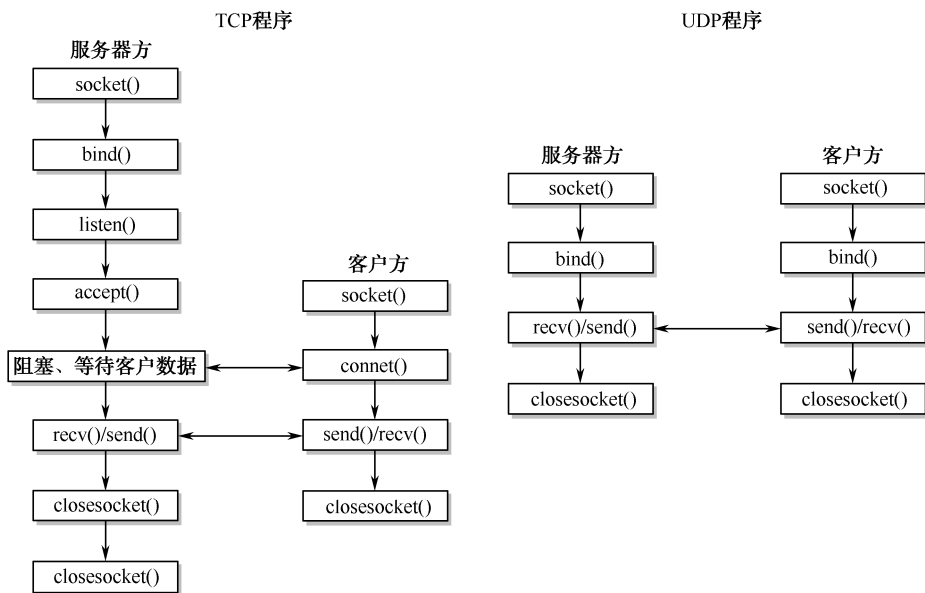


图 3.1 TCP 与 UDP 程序

从图 3.1 中可以看到 3.1.2 节介绍的那些 Winsock 函数的先后调用次序。TCP 程序通信前需要在双方进程之间先建立连接, 而 UDP 则不需要。

3.2 TCP 编程

本节介绍 TCP 程序的编写,将写一个与第2章的 MFC Socket 程序表现效果完全一样的程序,不同之处在于:这次不使用 MFC 提供的 Socket 类,而是直接调用 Winsock API 函数接口。

3.2.1 TCP 通信流程

TCP 程序是面向连接的,程序运行后,服务器一直处于监听状态,客户端与服务器通信之前必须首先发起连接请求,由服务器接收请求并在双方之间建立连接后才可以互通信息,详细流程如图 3.2 所示。

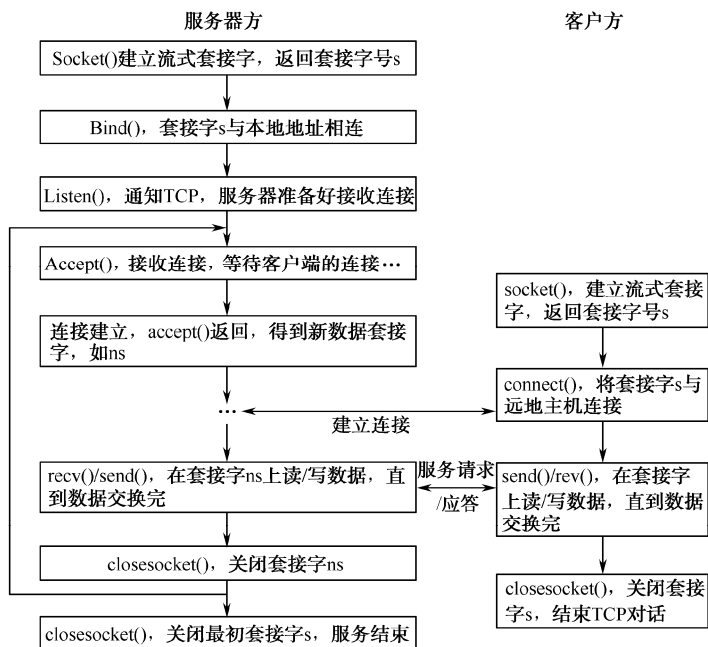


图 3.2 TCP 程序通用流程

下面就根据这个流程来编写程序。

3.2.2 TCP Socket API 程序设计

1. 创建工程和界面设计

分别创建客户端和服务端工程,工程命名与第2章的 Socket 程序一样(ChatClient 和 ChatServer)。设计程序界面也与第2章的程序一样,如图 3.3 所示。

在此,为了与第2章的程序区别,仅在标题栏上标注“(Winsock API 版)”,表明这个是直接调用 Winsock API 编写的程序版本。



图 3.3 TCP 程序界面

界面上各控件关联的变量见表 3.1 和表 3.2。

表 3.1 客户端控件变量

控 件	变量 (包括类型)
IP 地址控件	ctrl ServerIP
“端口号”文本框	ctrl ServerPort; CString sPort
“连接”按钮	ctrl m_ButtonConnect
“断开”按钮	ctrl m_ButtonDisconnect
列表框	ctrl m_ListWords
“清空”按钮	ctrl m_ButtonClear
发送信息编辑框	ctrl m_EditWords; CString m_sWords
“发送”按钮	ctrl m_ButtonSend
“退出”按钮	ctrl m_ButtonExit

表 3.2 服务器控件变量

控 件	变量 (包括类型)
IP 地址控件	ctrl ServerIP
“端口号”文本框	ctrl ServerPort; CString sPort
列表框	ctrl m_ListWords
“断开”按钮	ctrl m_ButtonDisconnect
“清空”按钮	ctrl m_ButtonClear
“退出”按钮	ctrl m_ButtonExit
“开始监听”按钮	ctrl m_ButtonListen
“停止监听”按钮	ctrl m_ButtonStopListen
发送信息编辑框	ctrl m_EditWords; CString m_sWords
“发送”按钮	ctrl m_ButtonSend

2. 编程实现

1) 客户端

由于本程序是直接调用 Winsock API 函数实现功能的, 故无须像第 2 章的程序一样去创建 Socket 类或架构指针机制的程序框架, 直接写代码就可以了。

对于客户端，只需要定义其套接字变量，在 ChatClientDlg.h 文件的 CChatClientDlg 类中定义：

```
SOCKET m_client;
```

这个变量就作为客户端的套接字对象。

“连接”按钮的事件过程代码如下：

```
void CChatClientDlg::OnConnect()
{
    WSADATA wsd;                                //WSADATA 结构
    WSAStartup(MAKEWORD(2,2),&wsd);            //加载协议，使用 Winsock 2.2 版
    m_client = socket(AF_INET,SOCK_STREAM,0);    //创建流式套接字
    //服务器地址
    sockaddr_in serveraddr;
    UpdateData();
    if(ServerIP.IsBlank())
    {
        AfxMessageBox("请指定服务器 IP!");
        return;
    }
    if(sPort.IsEmpty())
    {
        AfxMessageBox("请指定端口!");
        return;
    }
    //获取服务器进程的 IP 和端口
    BYTE nFild[4];
    CString sIP;
    ServerIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
    sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
    //设置服务器地址结构的内容
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.S_un.S_addr = inet_addr(sIP);
    serveraddr.sin_port = htons(atoi(sPort));
    //发起连接须指明要访问的服务器进程地址，这个地址存储在 serveraddr 中
    if(connect(m_client,(sockaddr*)&serveraddr,sizeof(serveraddr)) != 0)
    {
        MessageBox("连接失败");
        return;
    }
    else
    {
        m_ListWords.AddString("连接服务器成功!");
        m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
    }
    WSASyncSelect(m_client,m_hWnd,10000,FD_READ|FD_CLOSE); //①
    //界面完善
    ServerIP.EnableWindow(false);
```

```

ServerPort.EnableWindow(false);
m_ButtonConnect.EnableWindow(false);
m_ButtonDisconnect.EnableWindow();
m_EditWords.EnableWindow();
m_ButtonSend.EnableWindow();
m_ButtonExit.EnableWindow(false);
m_ButtonClear.EnableWindow();
}

```

从上述代码中，读者可以很清楚地看到客户端发起连接请求的过程：

加载协议栈（WSAStartup()）→ 创建流式套接字（socket()）→ 发起连接（connect()）

在发起连接后还用到了一个 WSAAsyncSelect() 函数，它是异步事件通知函数。这个函数在 Windows 编程中很有用，此处先编号为“//①”，为的是与后面的程序代码结合起来介绍，先接着看其他的代码段。

“断开”按钮的事件过程代码如下：

```

void CChatClientDlg::OnDisconnect()
{
    //断开与服务器的连接
    closesocket(m_client);
    m_ListWords.AddString("从服务器断开");
    m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
    ServerIP.EnableWindow();
    ServerPort.EnableWindow();
    m_ButtonConnect.EnableWindow();
    m_ButtonDisconnect.EnableWindow(false);
    m_EditWords.EnableWindow(false);
    m_ButtonSend.EnableWindow(false);
    m_ButtonExit.EnableWindow();
}

```

这个比较简单，就是直接调用 closesocket() 函数关闭套接字。

“发送”按钮的事件过程代码如下：

```

void CChatClientDlg::OnSend()
{
    //向服务器发送信息
    UpdateData();
    if(m_sWords.IsEmpty())
    {
        AfxMessageBox("发送的消息不能为空!");
        return;
    }
    //开始发送数据
    int i = send(m_client, m_sWords.GetBuffer(0), m_sWords.GetLength(), 0);
    m_ListWords.AddString("发送: " + m_sWords);
    m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
}

```

这个也是直接调用 send() 函数发送信息。

由于本程序是直接使用 Winsock 函数而并没有创建 MFC 的 Socket 类, 故无法使用 Socket 类对象内置的消息驱动功能, 那么客户端在与服务器建立连接后又如何能知道服务器有没有发送数据给它以及这些数据是否到达了呢? 假如服务器主动断开了连接, 又怎么通知客户端呢?

这里就要用到 Windows 的异步事件通知, 它是建立在 Windows 消息机制基础上的一种十分重要的编程机制。

刚才上面给出的“连接”按钮的事件过程中有这样一句代码, 标号为“//①”, 如下所示:

```
WSAAsyncSelect(m_client,m_hWnd,10000,FD_READ|FD_CLOSE); //①
```

这句代码是在客户端发起连接(调用 connect())成功之后使用的, 就是用于随时获取服务器发来的数据和感知服务器连接断开事件的。

WSAAsyncSelect()函数是 Winsock 提供的一个适合于 Windows 编程的方法, 它允许在一个套接字上当发生特定的网络事件时, 给 Windows 网络程序(窗口或对话框)发送一个消息(事件通知), 其原型为:

```
int WSAAsyncSelect(SOCKET s,HWND hWnd,unsigned int wMsg,long lEvent);
```

该函数调用成功返回 0, 否则返回 SOCKET_ERROR。

参数说明如下。

- s: 套接字对象。
- hWnd: 接收消息的窗口句柄。
- wMsg: 传给窗口的消息。
- lEvent: 被注册的网络事件, 也是应用程序向窗口发送消息的网络事件。该值为下列值——FD_READ、FD_WRITE、FD_OOB、FD_ACCEPT、FD_CONNECT、FD_CLOSE 的组合。大家可以翻到 2.2.2 节的表 2.3, 在介绍 CAsyncSocket 类编程的源代码剖析中有一个名为 AsyncSelect 的方法, 其实它是由封装了 Winsock API 的 WSAAsyncSelect()函数得来的, 因此 lEvent 网络事件的上述取值与表 2.3 中的网络事件类型是完全一样的。

语句①中取值为 FD_READ|FD_CLOSE。

FD_READ: 接收读准备好的通知, 在本程序中其实就是客户端套接字 m_client 收到(服务器发来的)数据时得到系统的事件通知。

FD_CLOSE: 接收套接字关闭的通知, 这里指的是当服务器主动断开连接(关闭套接字)时, 客户端收到的系统通知。

具体应用时, wMsg 为在应用程序中定义的消息号(取 10000), 消息结构中的 lParam 则为以上各种网络事件的名称。因此, 可以在窗口处理自定义消息函数中使用以下结构来响应 Socket 的不同事件:

```
Switch(lParam)
{
    Case  FD_READ:
        ...
        break;
    Case  FD_WRITE:
        ...
        break;
    ...
}
```

本程序就是利用这种事件通知机制编程, 从而使客户端套接字能够及时收到服务器发来的数

据并对服务器的断开事件做出反应。

具体编程方法如下。

为主对话框对象添加 PreTranslateMessage()方法, 如图 3.4 所示。

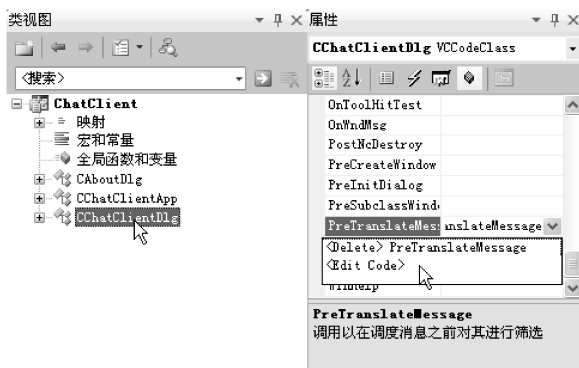


图 3.4 添加 PreTranslateMessage()方法

在该方法的框架中添加如下代码:

```

BOOL CChatClientDlg::PreTranslateMessage(MSG* pMsg)
{
    if(pMsg->message == 10000)           //识别应用程序中定义的消息号
    {
        switch(pMsg->lParam)             //判断网络事件类型
        {
            case FD_READ:
                this->ReceiveData();      //若为 FD_READ 则接收数据
                break;
            case FD_CLOSE:
                this->CloseSock();        //如果服务端断开, 客户端也断开
                break;
        }
    }
    else
        return CDialog::PreTranslateMessage(pMsg);
}

```

下面来实现 ReceiveData() 和 CloseSock() 方法。先在头文件 ChatClientDlg.h 中的类 CChatClientDlg 中定义这两个方法:

```

void ReceiveData();
void CloseSock();

```

然后分别编写这两个方法的代码。

ReceiveData()的代码如下:

```

void CChatClientDlg::ReceiveData()
{
    char buffer[1024];
    //接收服务器传来的数据
    int num = recv(m_client,buffer,1024,0); //函数 recv()接收数据
    buffer[num] = '\0';
}

```

```

//将接收的数据添加到列表框中
CString sTemp;
sTemp.Format("收到: %s",buffer);
//接收完数据后继续侦测
WSAAsyncSelect(m_client,m_hWnd,10000,FD_READ|FD_CLOSE);
m_ListWords.AddString(sTemp);
m_ListWords.SetTopIndex(m_ListWords.GetCount()-1);
}

```

CloseSock()的代码如下:

```

void CChatClientDlg::CloseSock()
{
    m_ListWords.AddString("服务器断开了");
    m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
    closesocket(m_client);
    ServerIP.EnableWindow();
    ServerPort.EnableWindow();
    m_ButtonConnect.EnableWindow();
    m_ButtonDisconnect.EnableWindow(false);
    m_EditWords.EnableWindow(false);
    m_ButtonSend.EnableWindow(false);
    m_ButtonExit.EnableWindow();
}

```

在进行了以上一系列处理后,客户端就能够对服务器的动作随时做出响应了。

为了使程序更加完善,为客户端编写初始化界面的代码(在主对话框对象的 OnInitDialog()方法中),如下所示:

```

//界面初始化
m_ButtonDisconnect.EnableWindow(false);
m_ButtonClear.EnableWindow(false);
m_EditWords.EnableWindow(false);
m_ButtonSend.EnableWindow(false);

```

“清空”按钮的事件过程:

```

m_ListWords.ResetContent();

```

“关于”按钮的事件过程:

```

CAboutDlg dlgAbout;
dlgAbout.DoModal();           //显示“关于”对话框

```

这些都与第2章的程序一模一样。

2) 服务器

服务器需要两个套接字(一个用于监听,另一个用于与客户端通信),故相应地要定义两个 SOCKET 结构的变量。

在头文件 ChatServerDlg.h 的类 CChatServerDlg 中定义:

```

SOCKET m_server,m_client;

```

“开始监听”按钮事件过程代码如下:

```

void CChatServerDlg::OnListen()
{

```

```

    WSADATA wsd;           //WSADATA 结构

```

```

WSAStartup(MAKEWORD(2,2),&wsd);           //加载协议栈, 使用 Winsock 2.2 版
m_server = socket(AF_INET,SOCK_STREAM,0);   //创建流式套接字
//将网络中的事件关联到窗口的消息函数中, 定义消息号为 20000, 侦测客户端的连接请求
WSAAsyncSelect(m_server,m_hWnd,20000,FD_ACCEPT);
m_client = 0;
BYTE nFild[4];
CString sIP;
UpdateData();
if(ServerIP.IsBlank())
{
    AfxMessageBox("请设置 IP 地址!");
    return;
}
if(sPort.IsEmpty())
{
    AfxMessageBox("请设置监听端口!");
    return;
}
ServerIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
//服务器地址
sockaddr_in serveraddr;
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.S_un.S_addr = inet_addr(sIP);
serveraddr.sin_port = htons(atoi(sPort));
//绑定地址
if (bind(m_server,(sockaddr*)&serveraddr,sizeof(serveraddr)))
{
    MessageBox("绑定地址失败.");
    return;
}
//监听开始, 服务器等待连接请求的到来
listen(m_server,5);
m_ListWords.AddString("监听开始: ");
m_ListWords.AddString("地址" + sIP + " 端口" + sPort);
m_ListWords.AddString("等待客户端连接……");
//界面完善
m_ListWords.SetTopIndex(m_ListWords.GetCount()-1);
ServerIP.EnableWindow(false);
ServerPort.EnableWindow(false);
m_ButtonListen.EnableWindow(false);
m_ButtonStopListen.EnableWindow();
m_ButtonClear.EnableWindow();
m_ButtonExit.EnableWindow(false);
}

```

由程序代码中加黑的语句可以看出, 服务器程序也遵循 Winsock API 编程的通行步骤: 加载协议栈 (WSAStartup()) → 创建套接字 (socket()) → 绑定地址 (bind()) → 开始监听 (listen())

“停止监听”按钮事件过程代码如下：

```
void CChatServerDlg::OnStopListen()
{
    //停止监听
    closesocket(m_server);
    m_ListWords.AddString("停止监听");
    m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
    ServerIP.EnableWindow();
    ServerPort.EnableWindow();
    m_ButtonListen.EnableWindow();
    m_ButtonStopListen.EnableWindow(false);
    m_ButtonExit.EnableWindow();
}
```

要停止监听很简单，只需直接关闭监听套接字（`closesocket()`）就可以了。此时监听套接字虽然关闭了，但已经连接的客户端的套接字依然可以正常工作，它们还可以继续与服务器交互信息，直到通信双方有一方主动断开连接为止。

“发送”按钮事件过程代码如下：

```
void CChatServerDlg::OnSend()
{
    //向服务器发送信息
    UpdateData();
    if(m_sWords.IsEmpty())
    {
        AfxMessageBox("发送的消息不能为空!");
        return;
    }
    //开始发送数据
    int i = send(m_client, m_sWords.GetBuffer(0), m_sWords.GetLength(), 0);
    m_ListWords.AddString("发送: " + m_sWords);
    m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
}
```

“断开”按钮事件过程代码如下：

```
void CChatServerDlg::OnDisconnect()
{
    closesocket(m_client);
    m_ListWords.AddString("与客户端断开");
    m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
    m_ButtonDisconnect.EnableWindow(false);
    m_EditWords.EnableWindow(false);
    m_ButtonSend.EnableWindow(false);
}
```

同样，服务器要能及时获取客户端发来的数据和响应客户端主动断开连接的网络事件，就必须采用与前面客户端一样的网络异步事件通知机制编写程序。

在头文件 `ChatServerDlg.h` 的类 `CChatServerDlg` 中声明：

```
void HandleData();           //用于接收客户端的连接请求
void ReceiveData();
void CloseSock();
```

这里由于身为服务器，还必须随时接收用户的连接请求，故比客户端程序多了一个 `HandleData()` 函数用于接收客户端的连接请求。与客户端一样，先给主对话框对象添加 `PreTranslateMessage()` 方法。

```
BOOL CChatServerDlg::PreTranslateMessage(MSG* pMsg)
{
    if (pMsg->message == 20000)           //识别消息号
    {
        this->HandleData();               //接收客户端的连接请求
    }
    else if (pMsg->message == 30000)       //识别消息号
    {
        switch(pMsg->lParam)
        {
            case FD_READ:
                this->ReceiveData();       //接收数据
                break;
            case FD_CLOSE:
                this->CloseSock();          //断开连接
                break;
        }
    }
    else
        return CDialog::PreTranslateMessage(pMsg);
}
```

可以看到，上面的程序中使用了两个不同的消息号，第一个用于识别客户端连接请求的消息号为 20000，这是与前面“开始监听”按钮事件过程代码中的这句语句相对应的：

```
//将网络中的事件关联到窗口的消息函数中，定义消息号为 20000，侦测客户端的连接请求
WSAAsyncSelect(m_server,m_hWnd,20000,FD_ACCEPT);
```

服务器是调用 `accept()` 函数建立连接的，为了知道什么时候客户端提出连接要求，从而服务器的套接字在恰当的时候调用 `accept()` 函数完成连接的建立，需要使用 `WSAAsyncSelect()` 函数，让系统主动通知有客户端提出连接请求了。之前已经详细解析过这个函数，它的第三个参数 `wMsg` 为传给窗口的消息，这里指定消息号为 20000，服务器程序一旦收到系统传来的这个消息，就知道有客户端提出连接要求了，于是调用 `HandleData()` 函数。其代码如下：

```
void CChatServerDlg::HandleData()
{
    sockaddr_in serveraddr;
    int len = sizeof(serveraddr);
    m_client = accept(m_server,(struct sockaddr*)&serveraddr,&len);
    //在接收请求并为客户端创建对应的套接字后，就开始在该套接字上侦测
    //FD_READ 和 FD_CLOSE 事件，指定消息号为 30000
    WSAAsyncSelect(m_client,m_hWnd,30000,FD_READ|FD_CLOSE);
    m_ListWords.AddString("接收了一个客户端的连接请求");
```

```

    m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
    //界面完善
    m_ButtonDisconnect.EnableWindow();
    m_EditWords.EnableWindow();
    m_ButtonSend.EnableWindow();
}

```

服务器在接收 (accept()) 连接后, 就在新创建的套接字上侦测客户端是否发来数据或主动断开连接的事件, 并设定第二个消息号为 30000, 这就是在 PreTranslateMessage() 方法中要用 30000 号来识别 FD_READ 和 FD_CLOSE 事件的原因。对应这两个事件调用的处理函数和客户端一样, 也是 ReceiveData() 和 CloseSock()。

ReceiveData() 函数代码如下:

```

void CChatServerDlg::ReceiveData()
{
    //接收客户端的数据
    char buffer[1024];
    int num = recv(m_client, buffer, 1024, 0);
    buffer[num] = 0;
    CString sTemp;
    sTemp.Format("收到: %s", buffer);
    m_ListWords.AddString(sTemp);                //显示信息
    m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
    //继续侦测客户端的动向
    WSAAsyncSelect(m_client, m_hWnd, 30000, FD_READ|FD_CLOSE);
}

```

CloseSock() 函数代码如下:

```

void CChatServerDlg::CloseSock()
{
    //关闭与客户端的通信信道
    m_ListWords.AddString("客户端断开连接");
    m_ListWords.SetTopIndex(m_ListWords.GetCount() - 1);
    closesocket(m_client);                //关闭与客户端通信的 Socket
    WSAAsyncSelect(m_server, m_hWnd, 20000, FD_ACCEPT);    //准备接收新的客户端连接
    //界面完善
    m_ButtonDisconnect.EnableWindow(false);
    m_EditWords.EnableWindow(false);
    m_ButtonSend.EnableWindow(false);
}

```

服务器的初始化代码如下:

```

//界面初始化
m_ButtonStopListen.EnableWindow(false);
m_ButtonDisconnect.EnableWindow(false);
m_ButtonClear.EnableWindow(false);
m_EditWords.EnableWindow(false);
m_ButtonSend.EnableWindow(false);

```

“清空”、“关于”按钮的事件过程与客户端的完全一样。

至此, 这个直接调用 Winsock API 的程序就编写完成了。

3.2.3 Winsock API 程序与 MFC Socket 程序的等价性

运行本章的实例程序，读者会发现它与第 2 章的三个版本（CAsyncSocket 传统版、CAsyncSocket 指针版和 CSocket 版）的 Socket 程序运行效果是完全一样的。这样一来，同样功能的 Socket 程序又多了一个完全不同的实现版本——Winsock API 版！

那么将 Winsock API 程序与 MFC Socket 程序混合通信的结果又如何呢？下面就来试试看。

选用第 2 章 CAsyncSocket 传统版的 Socket 程序，将它的客户端与 Winsock API 程序的服务器连接通信，如图 3.5 所示。

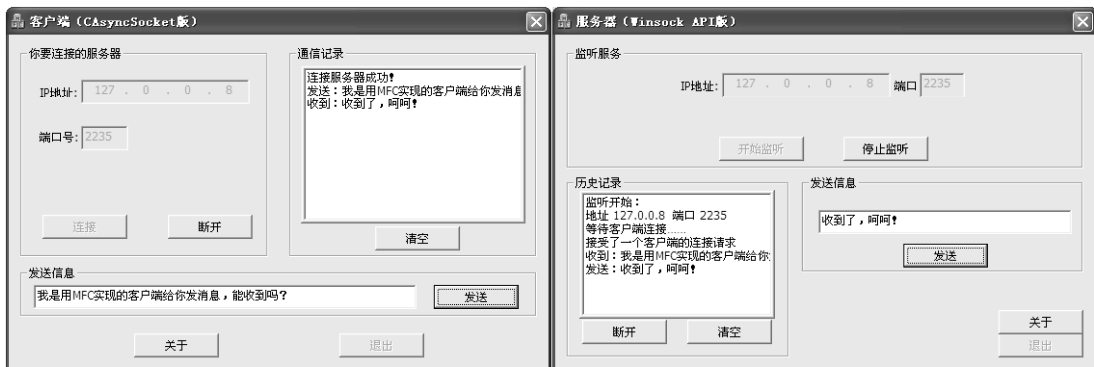


图 3.5 CAsyncSocket 客户端连接 Winsock API 服务器

它们可以正常通信！若反过来，用 Winsock API 的客户端连接 MFC Socket 服务器，通信也是正常的，读者可以亲自试一下。

不仅如此，如果有兴趣，还可以将第 2 章那三个版本的程序连同本章这个，总共 4 组 8 个（4 个客户端+4 个服务器）程序完全混放在一起，从中任意抽出一个客户端与任何一个服务器相连，都能成功连接并正常通信！

这是因为 MFC Socket 类中的方法其实就是调用 Winsock API 函数实现的，只不过进行了面向对象的封装而已，在底层它们实质上使用的是同一个接口——Winsock API。用户程序实际上无法判断与它通信的对方程序究竟是用套接字 API 还是 MFC 套接字类编写实现的，这一切对用户进程而言完全透明，我们将这一奇妙的现象称为 Winsock API 程序与 MFC Socket 程序的等价性。也就是说，Winsock API 与 MFC Socket 类在本质上是同一编程界面的不同表现形式，这一点参见本书第 1 章的图 1.42 就一目了然了。大家在实际编程时可以自由地选择这两种等价的编程方式。

3.3 UDP 编程

本节将介绍 UDP 程序的设计。作为 Winsock API 编程的主要类别之一的 UDP 编程，在网络编程中有着特殊的地位，现在很多流行的网络应用（如 QQ、MSN 等）就是架构在 UDP 基础上的。

3.3.1 UDP 通信流程

UDP 程序是无连接的，任何一个进程，只要创建了数据报套接字并与自身绑定后，就可以向任何正在运行中的其他同样创建和绑定了数据报套接字的进程发信息，双方无须建立连接，流

程如图 3.6 所示。

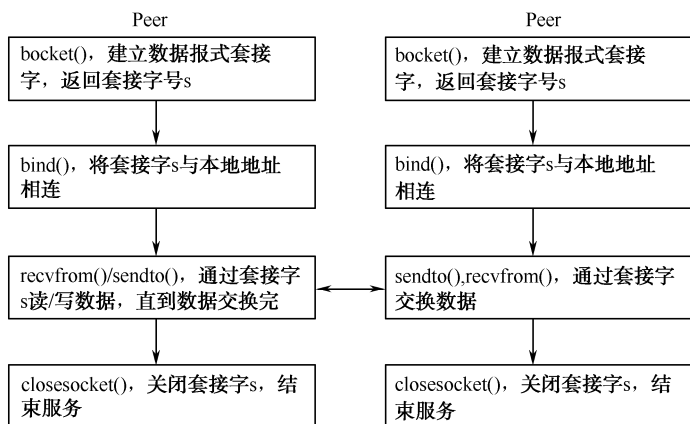


图 3.6 UDP 程序通信流程

由图 3.6 可见, 在这种 Socket 的通信方式中, 已分不清谁是服务器、谁是客户端了。通信的任何一方既可以扮演客户端又可以扮演服务器的角色, 甚至可以集两种角色于一身, 既是客户端又是服务器, 就看它在通信中是干什么的了。我们称这样的一个通信进程主体为一个“peer”, peer 与 peer 间的地位是平等的, 它们的通信是点对点通信, 又称为 P2P 通信, 目前已成为 Internet 上主流的通信方式。

3.3.2 UDP Socket API 程序设计

1. UDP 程序工程创建和界面设计

和基于 TCP 的 Socket 程序不同, 由于 UDP 通信中不存在客户端和服务端之分, 因此本例只需创建一个工程就够了, 将它命名为 UDProcsComm (“UDP 进程通信”之意), 其余设置不变。设计程序界面如图 3.7 所示。



图 3.7 UDP 程序界面

这里将界面设计区初始默认的“确定”按钮更名为本程序的“启动”按钮, “取消”按钮则作为“退出”按钮, 充分利用了 VC 环境内置的功能。图 3.7 所示界面左区选中部分是一个列表框 (ListBox) 控件, 用于显示程序运行过程中的信息。可见, 本程序可以设置启动某 IP 的机器

上指定端口的进程，并将消息发往指定 IP 和端口的目标进程。

界面上各控件关联的变量见表 3.3。

表 3.3 控件变量关联

控 件	变量 (包括类型)
“本机进程” IP 地址控件	contrl IPLocal
“本机进程” 端口编辑框	contrl PortLocal; CString LocalPort
“启动” 按钮	contrl m_Start
“停止” 按钮	contrl m_Stop
“目标进程” IP 地址控件	contrl IPDest
“目标进程” 端口编辑框	contrl PortDest; CString DestPort
发送信息编辑框	contrl m_EditWords; CString str
“发送” 按钮	contrl m_Send
程序运行信息列表框 (界面左区)	contrl list
“退出” 按钮	contrl m_Exit

2. 编程实现

首先编写初始化代码，在 UDProcsCommDlg.h 头文件 CUDProcsCommDlg 类定义中添加如下代码：

```
public:
    SOCKET Client;                //客户端的连接请求
    SOCKET ServerSocket;          //SOCKET
    SOCKADDR_IN m_sockServerAddr; //SOCKET 结构
    SOCKADDR_IN m_sockAddrto;    //SOCKET 结构
    int socklen;
    BOOL IsTrue;
    Msg msg;                      //收发消息的结构体
    UINT m_sport;
    UINT m_dport;
```

在 UDProcsCommDlg.cpp 文件中完善主对话框类 CUDProcsCommDlg 的构造方法，代码如下：

```
CUDProcsCommDlg::CUDProcsCommDlg(CWnd* pParent /*=NULL*/)
: CDialog(CUDProcsCommDlg::IDD, pParent)
, str(_T(""))
, LocalPort(_T(""))
, DestPort(_T(""))
{
    str = _T("");
    m_sport = 0;
    m_dport = 0;
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
```

```

    IsTrue = FALSE;
    Client = INVALID_SOCKET;
}

```

向主对话框方法 `BOOL CUDProcsCommDlg::OnInitDialog()` 中添加如下代码:

```

IPLocal.SetFocus();
list.EnableWindow(false);
m_Stop.EnableWindow(false);
IPDest.EnableWindow(false);
PortDest.EnableWindow(false);
m_EditWords.EnableWindow(false);
m_Send.EnableWindow(false);

```

初始化完成, 接下来启动本机进程, 这十分简单。从前面得知, “启动” 按钮是由 VC 初始内置的“确定”按钮更名而来的, 直接为程序的主对话框类 `CUDProcsCommDlg` 添加 `OnOK()` 函数过程即可, 如图 3.8 所示。

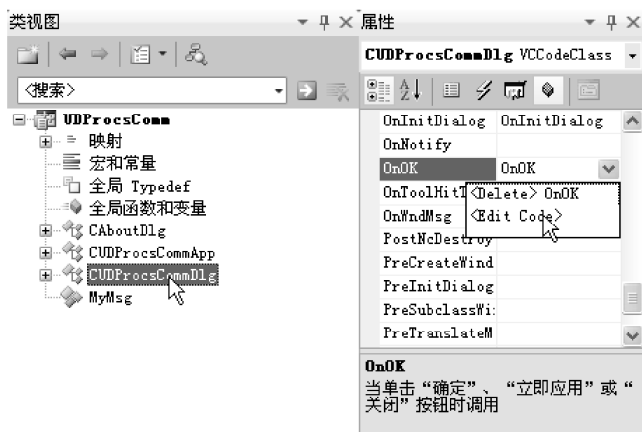


图 3.8 主对话框 `CUDProcsCommDlg` 添加 `OnOK()` 函数过程

`OnOK()` 函数过程 (可以看作“启动”按钮的事件过程) 代码如下:

```

void CUDProcsCommDlg::OnOK()
{
    UpdateData();
    if(IPLocal.IsBlank())
    {
        AfxMessageBox("请设置 IP 地址!");
        return;
    }
    if(LocalPort.IsEmpty())
    {
        AfxMessageBox("请设置端口号!");
        return;
    }
    //初始化与绑定
    WSADATA wsaData;
    int iErrorCode;
    if (WSAStartup(MAKEWORD(2,1),&wsaData)) //调用 Windows Socket DLL

```

```

{
    list.AddString("Winsock 无法初始化!");
    WSACleanup();
    return;
}
list.AddString("开始创建 Socket...");
//创建本机进程的 Socket, 类型为 SOCK_DGRAM, 无连接的通信
ServerSocket = socket(PF_INET,SOCK_DGRAM,0);
if(ServerSocket == INVALID_SOCKET)
{
    list.AddString("创建 socket 失败!");
    return;
}
//获取本机进程的 IP 和端口
BYTE nFild[4];
CString sIP;
IPLocal.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);

m_sockServerAddr.sin_family = AF_INET;
m_sockServerAddr.sin_addr.s_addr = inet_addr(sIP);
m_sockServerAddr.sin_port = htons(atoi(LocalPort));

socklen = sizeof(m_sockServerAddr);

if (bind(ServerSocket,(LP SOCKADDR)&m_sockServerAddr,
sizeof(m_sockServerAddr)) == SOCKET_ERROR) //与设置的端口绑定
{
    list.AddString("绑定失败!");
    return;
}
iErrorCode = WSAAsyncSelect(ServerSocket,m_hWnd,
WM_CLIENT_R. EADCLOSE,FD_READ);
//产生相应传递给窗口的消息为 WM_SERVER_ACCEPT , 这是自定义消息
if (iErrorCode == SOCKET_ERROR)
{
    list.AddString("WSAAsyncSelect 设定失败!——用于连接请求的消息");
    return;
}
list.AddString("本机进程启动成功!");
list.AddString("地址" + sIP + " 端口" + LocalPort);
this->SetWindowTextA("本机应用进程 (" + sIP + ":" + LocalPort + ") -UDProcsComm");
//界面
IPLocal.EnableWindow(false);
PortLocal.EnableWindow(false);
m_Start.EnableWindow(false);
m_Stop.EnableWindow(true);

```

```

IPDest.EnableWindow(true);
IPDest.SetFocus();
PortDest.EnableWindow(true);
m_EditWords.EnableWindow(true);
m_Send.EnableWindow(true);
list.EnableWindow(true);
m_Exit.EnableWindow(false);
return;
CDialog::OnOK();
}

```

可以看出,无连接的数据报套接字与面向连接的流式套接字使用方法差不多,都要经历“加载协议栈(WSAStartup())→创建套接字(socket())→绑定(bind())”这三步,只是在创建套接字时要指明套接字类型为数据报套接字 SOCK_DGRAM。

这里同样要用到 WSAAsyncSelect()函数,但是读者会发现它的用法与前面有一点不一样,这是在尝试一种新的用法:直接通过 VC 的宏映射机制来自定义消息,实现网络事件的通知——该方法并不局限于本例程序,有兴趣的读者可以深入钻研,在其他程序里凡是需要自己定义事件驱动的地方都可以尝试使用。

现在以本程序为例,简单介绍一下这种编程机制的使用。

在 stdafx.h 头文件中定义宏:

```
#define WM_CLIENT_READCLOSE WM_USER+102
```

在 UDProcsCommDlg.cpp 中定义消息宏映射:

```
ON_MESSAGE(WM_CLIENT_READCLOSE, OnReadClose)
```

该消息宏映射的添加位置在 BEGIN_MESSAGE_MAP(CUDProcsCommDlg, CDialog) 与 END_MESSAGE_MAP() 之间,如图 3.9 所示。

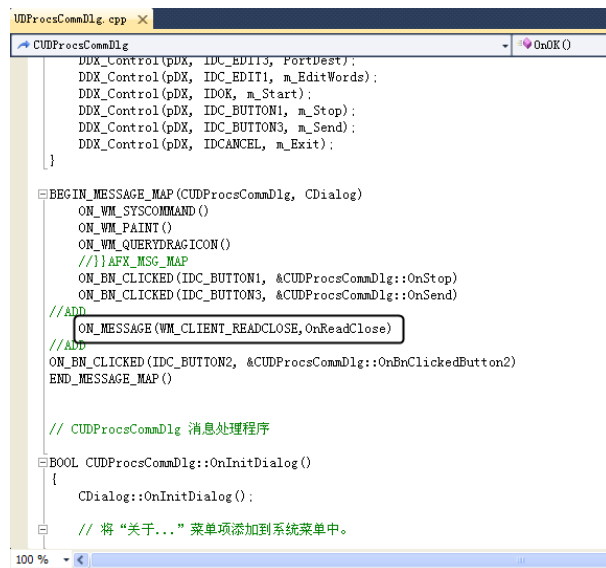


图 3.9 添加消息宏映射

然后在 UDProcsCommDlg.h 头文件的 CUDProcsCommDlg 对话框类定义代码中声明方法 OnReadClose():

```
LRESULT OnReadClose(WPARAM wParam, LPARAM lParam);
```

OnReadClose()函数代码为:

```
LRESULT CUDProcsCommDlg::OnReadClose(WPARAM wParam,LPARAM lParam)
{
    //自定义的关闭与缓冲区有消息
    CString str;
    switch (WSAGETSELECTEVENT(lParam))
    {
        case FD_READ:
            if(recvfrom(ServerSocket,(char *)&msg,sizeof(msg),0,(LPSOCKADDR)
                &m_sockServerAddr,(int *)&socklen) == SOCKET_ERROR)
            {
                list.AddString("发送失败!对方主机或应用进程没有启动");
                return 0;
            }
            str.Format("%s",msg.msg);
            list.AddString(str);
            break;
    }
    return 0L;
}
```

接收数据通过一个名为 msg 的变量,它是本例定义的一个结构体,专门用来存储通信双方收发的消息,其定义在文件 stdafx.h 中:

```
typedef struct MyMsg
{
    char msg[100];
    int i;
}Msg;
```

定义它之前,需要向项目中添加这个结构体对象(如图 3.10 所示)。

本机进程启动之后,用户可以随时停止它的运行。

“停止”按钮事件过程代码如下:

```
void CUDProcsCommDlg::OnStop()
{
    //当程序停止运行时,将 SOCKET 清空
    list.AddString("正在关闭 Socket...");
    closesocket(ServerSocket);
    WSACleanup();
    list.AddString("本机进程停止运行!");
    //界面
    IPLocal.EnableWindow(true);
    PortLocal.EnableWindow(true);
    m_Start.EnableWindow(true);
    m_Stop.EnableWindow(false);
    list.EnableWindow(false);
    IPLocal.SetFocus();
    IPDest.EnableWindow(false);
    PortDest.EnableWindow(false);
}
```

```

m_EditWords.EnableWindow(false);
m_Send.EnableWindow(false);
m_Exit.EnableWindow(true);
}

```

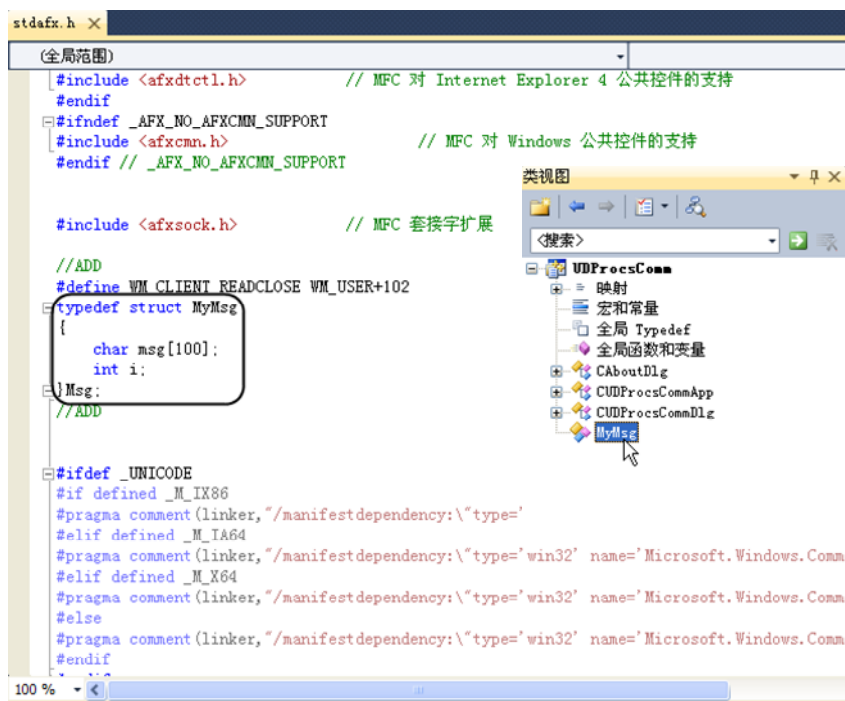


图 3.10 添加结构体 MyMsg

“发送”按钮事件过程代码如下：

```
void CUDProcsCommDlg::OnSend()
```

```

{
    //发送数据
    UpdateData();
    if(IPDest.IsBlank())
    {
        AfxMessageBox("请指定目标进程所在主机的 IP 地址!");
        return;
    }
    if(DestPort.IsEmpty())
    {
        AfxMessageBox("请指定目标进程的端口号!");
        return;
    }
    //获取目标进程的 IP 和端口
    BYTE nFild[4];
    CString sIP;
    IPDest.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
    sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
}

```

```

m_sockAddrto.sin_family = AF_INET;
m_sockAddrto.sin_addr.s_addr = inet_addr(sIP);
m_sockAddrto.sin_port = htons(atoi(DestPort));
if(str.IsEmpty())
{
    AfxMessageBox("发送的消息不能为空!");
    return;
}
strcpy(msg.msg,(LPCTSTR)str);
msg.i = 0;
if(sendto(ServerSocket,(char *)&msg,sizeof(msg),0,(LPSOCKADDR)
          &m_sockAddrto,sizeof(m_sockAddrto)) == SOCKET_ERROR)
{
    list.AddString("发送数据失败!");
};
str.Empty();
UpdateData(FALSE);
}

```

可以看到, 在 UDP 程序里, 进程发送消息时除了调用的是 `sendto()` 函数外, 其余与使用 TCP 并没有什么不同。不过, 因为 UDP 是直接将消息发出去的, 免去了通信双方建立连接的过程, 所以省却了很多麻烦。

3.3.3 UDP 进程通信演示

启动两个 UDP 程序, 双方具有一样的程序界面, 分别指定两个进程的通信地址为 127.103.212.9:1234 (进程 A) 和 127.121.100.7:2234 (进程 B)。当然, 读者可以任意指定地址, 只要 IP 是以 127 打头并且合法就可以 (形如 127.x.x.x 的 IP 是专为网络编程的程序员在自己计算机上测试程序而预留的)。

启动双方进程后, 在“发送到”→“目标进程”框中填写对方的 IP, 然后编辑文本互发, 如图 3.11 所示。

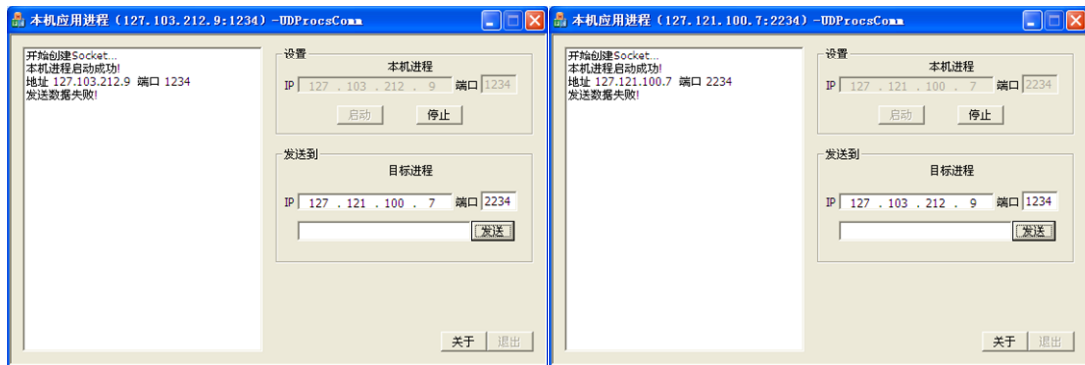


图 3.11 UDP 试发

令人失望的是, 通信双方的消息发送都失败了! 问题究竟出在哪里呢?

停止其中一方的进程 (如关掉进程 A), 将它的地址改为 127.0.0.1:1234, 重新启动。再次互

发信息，如图 3.12 所示。

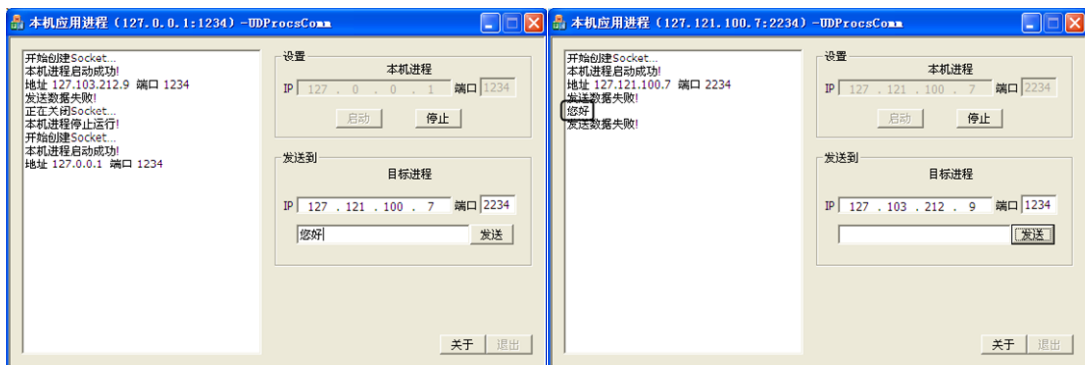


图 3.12 进程 A 向 B 发送成功

这一次，进程 B 收到 A 发送的信息（您好）了，但它向 A 发送信息仍然显示“发送数据失败！”，这说明刚才更改了 A 的本机进程地址是有用的，至少让其中的一方（进程 A）能够成功发送信息了。

经过大量的试验，我们发现了一个现象：只要将某终端的本机进程地址填写为形如“127.0.0.1:端口号”的形式，那么它就可以向其他终端发送消息，但反过来其他终端却无法成功向它发送消息。这是由于 127.0.0.1 这个 IP 并不表示具体主机的地址，而是泛指本地地址。

下面通过一组实验来演示一下这个结论。

同时开启三个通信终端，其中两个启动进程分别为上面进程 A 和进程 B 的地址，还有一个进程 C 设置它的本机进程地址为 127.0.0.1:2345。

在进程 C 的终端上设置目标进程地址为 A 的地址（127.103.212.9:1234），如图 3.13 所示，编辑文本发送。

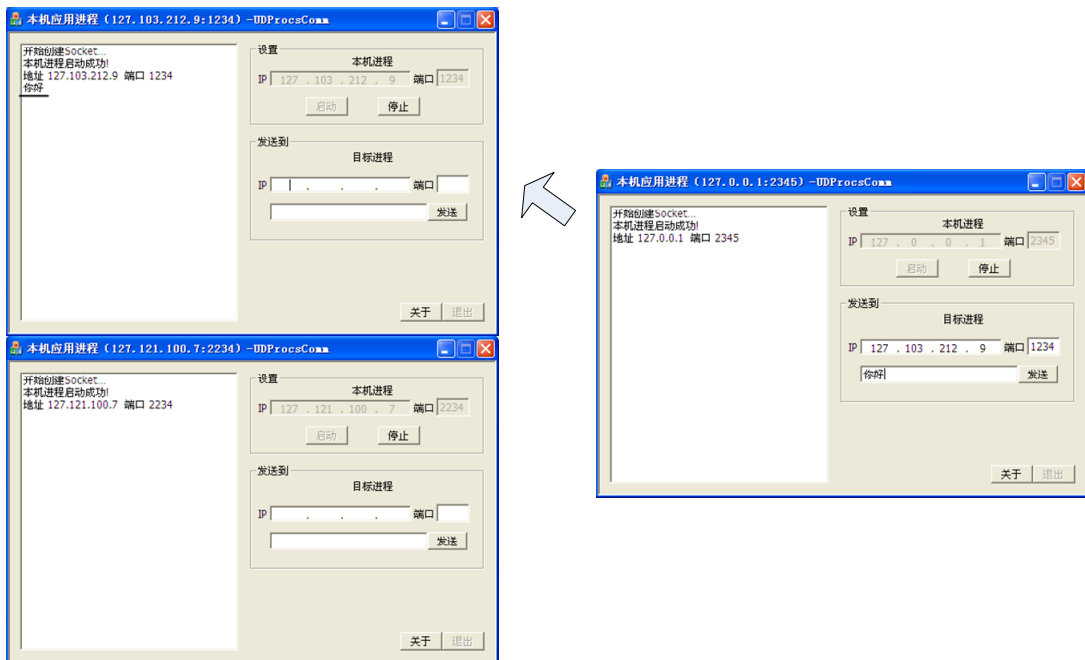


图 3.13 进程 C 向 A 发送成功

进程 A 收到信息了。

如果要将信息改发往进程 B，只需在进程 C 的终端界面上将“发送到”→“目标进程”的地址修改为 B 的地址（127.121.100.7:2234），如图 3.14 所示，然后发送信息即可。



图 3.14 进程 C 向 B 发送成功

这回轮到进程 B 收到信息了。

最后，分别从进程 A 和 B 向进程 C 发送信息，如图 3.15 所示。果然不出所料，发送是失败的，反向信息无法发送。

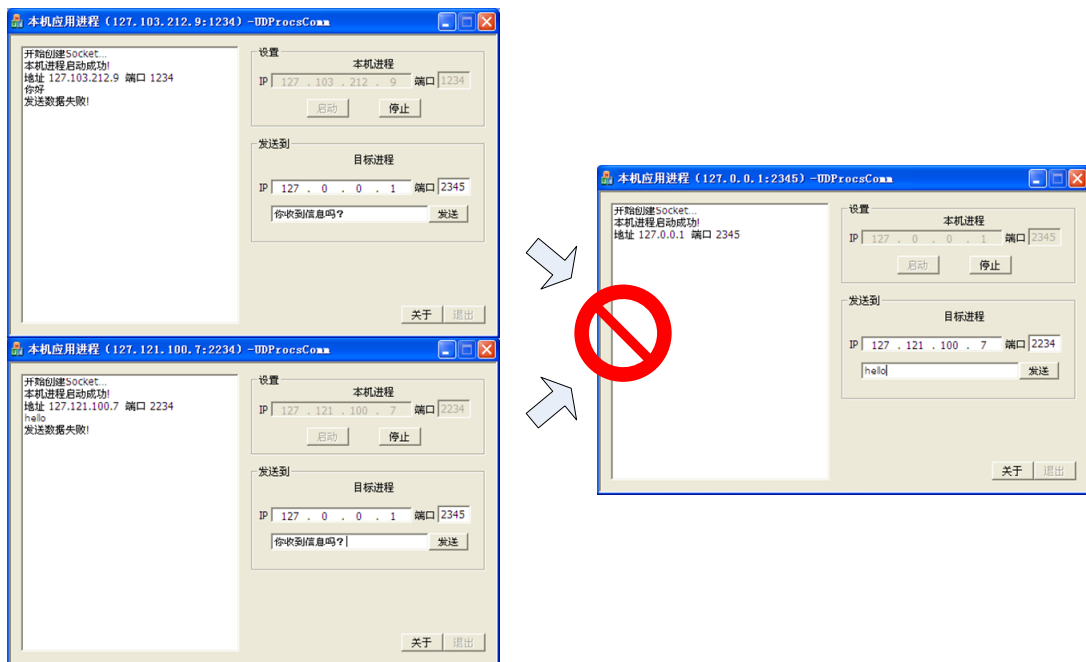


图 3.15 反向信息无法发送

总结以上演示实验的结果：发送方可以根据接收方的 IP 地址和端口号，向指定的接收方发送消息，反之则不行。这是 UDP 协议的特点，即接收进程并不知道发给它消息的进程是谁。

具体到编程操作中，发送进程在创建 Socket 后绑定地址时只需指明自己的端口号（将端口与 Socket 捆绑），IP 地址则一律填写为 127.0.0.1。注意：必须填写这一项，否则无法发送数据！

UDP 程序的这一性质很重要，在接下来的编程中就会用到。

即时通信应用开发

即时通信（Instant Messenger，IM），是一种基于互联网的即时发送和接收消息的业务，代表产品有百度 Hi、MSN、QQ 和 Skype 等，这类软件最典型的应用功能就是网络聊天。

4.1 IM 软件的体系结构

4.1.1 互联网中继通信原理

基础 Socket 程序实现了网络上应用进程之间简单的消息收发，但是如何才能做到让互联网用户大规模实时地交流信息呢？1988 年 8 月，芬兰人雅尔口·欧伊卡林恁（Jarkko Oikarinen）发明互联网中继聊天协议 IRC（Internet Relay Chat），解决了制约网络聊天应用普及和发展的瓶颈。

IRC 的基本思路是模仿电信交换的原理，在聊天系统中引入“中转”功能。如图 4.1 所示，假设网络用户 A、B、D、E 和 F 一起聊天，在他们中引入 C 作为中转，那么当 A 发言时，他将所说的话发给 C，然后 C 将 A 的话分别转发给 B、D、E 和 F……这样，信息广播的任务就全部由中转者 C 来承担。

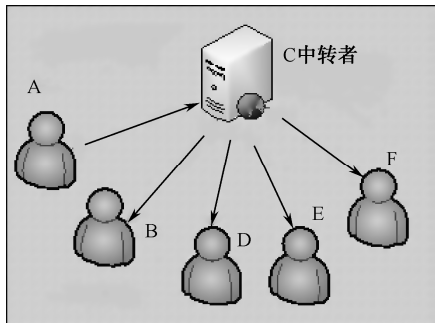


图 4.1 引入“中转者”

中转者 C 的存在使得信息交流过程中的工作任务发生分离，可以把网络环境好、机器配置高的计算机作为中转站来提供服务功能。这就形成了 IRC 的服务器—客户端模型，聊天者作为客户端，连接到中转站服务器上。

在上面这个例子中，只有一个中转者 C 来提供服务。当聊天者数量很多时，会使 C 不堪重负。解决的办法是，使用多个服务器，服务器之间互相连接成网络，如图 4.2 所示，把众多聊天

者分散到各个服务器上，服务器的网络则以树形结构互相连通。

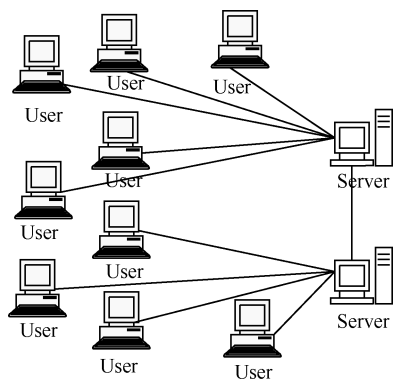


图 4.2 多服务器分担中转任务

聊天者可以任选一个服务器连接。举例来说，在北京建立一个 IRC 服务器，称为 BJ；再在上海建立一个 IRC 服务器，称为 SH。然后将两者连接起来，组成一个只有两个服务器的 IRC 网络。北京的用户连接到 BJ 上，上海的用户连接到 SH 上，这样北京用户就可以与上海用户聊天了。其他地区的用户可以根据地理位置的远近选择使用 BJ 或 SH 服务器。概括地说，聊天网络上的每个服务器都是一个中转站，当它从一个服务器或客户端收到一条消息时，就将该消息转发给其他服务器。

4.1.2 P2P 方式架构的系统

信息中转站架构的聊天系统服务器负担较重，并且在一对一的私人即时通信模式下也不利于用户保护隐私。自从 1999 年 1 月美国人肖恩·范宁（Shawn Fanning）发明了 P2P 的网络应用新模式后，诸多 IM 服务提供商纷纷转而采用 P2P 方式来构建自己的平台。国内像腾讯 QQ 这类著名的 ICQ 平台就是基于 P2P 方式架构的。众所周知，QQ 在传输层实现上采用的是 UDP 协议而非面向连接的 TCP，这一特点正好适合 P2P 应用，由基于 UDP 的 Winsock 程序很容易改造成简单的 P2P 结构的程序。

P2P 在技术实现上有很多种，鉴于 P2P 技术的机制比较复杂，本书仅从一种应用架构的层面来介绍 P2P。本章 4.3 节开发的聊天工具是以服务器协调的杂 P2P 方式工作的，其基本架构如图 4.3 所示。

可以看出，图 4.3 中的系统还是需要服务器介入的，服务器上保存各个用户终端的信息，当一个用户要与另一个用户通信时，先通过服务器获得对方的信息（主机地址、端口等），再根据这些信息联系对方，在此过程中服务器充当中介的角色。只要每个主机都运行了 P2P 软件，它们就可以进行对等的连接并互发信息聊天。虽然通信过程有服务器的介入，但实际上各用户终端之间是直接点对点交互的。

这种方式出现得最早，实现起来也不复杂，在目前依旧是 Internet 上 P2P 的主流应用方式。大多数的即时通信平台（包括 QQ）采用的都是杂 P2P 方式，QQ 仍然是有服务器的，我们平时上网登录 QQ 后所看到的动态显示在线人数、好友头像的列表就是来自于腾讯的服务器。

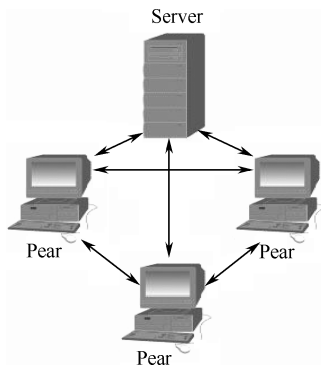


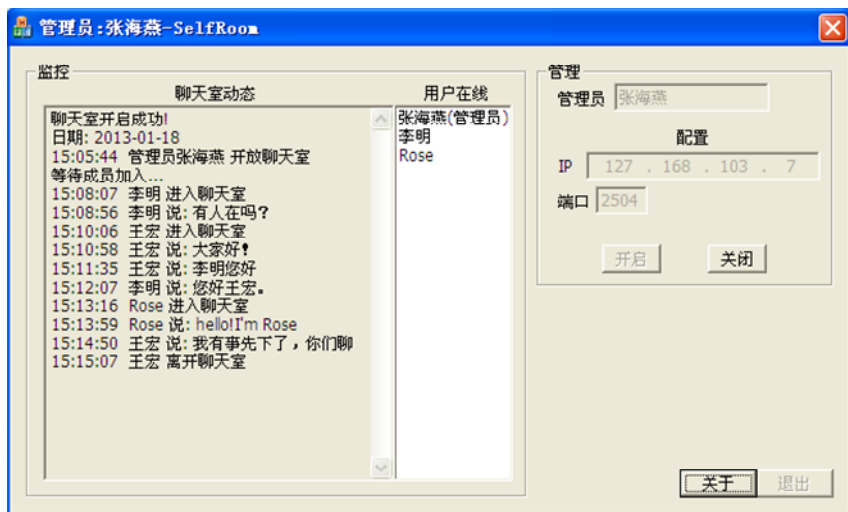
图 4.3 服务器协调的 P2P 工作方式

4.2 C/S 结构的聊天室应用

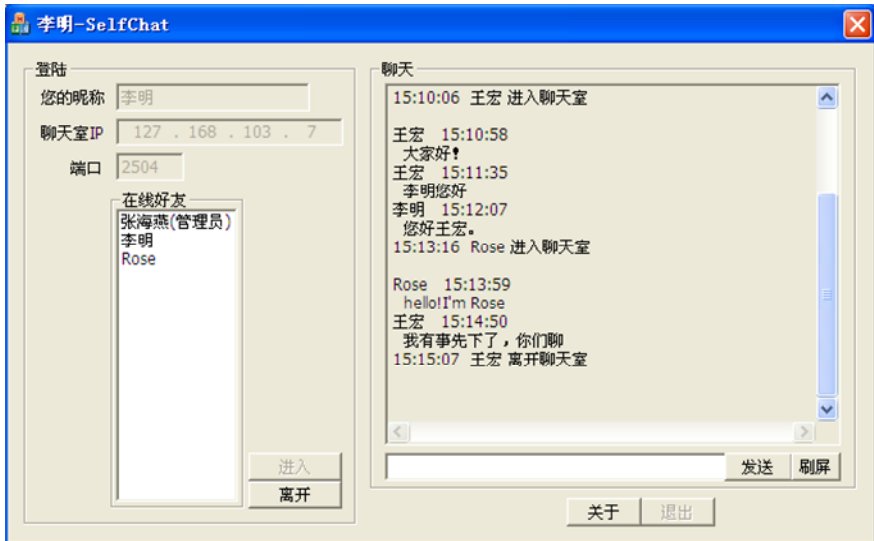
聊天室的基本原理就是前面介绍的“中转站”原理，这个原理决定了它只能采用客户端—服务器 (C/S) 结构。聊天室软件由客户端和服务端两部分构成，服务器充当“中转者”的职能，客户端将自己的消息先发送给服务器，服务器再根据需要将它转发给其他客户端。常用的 QQ 群功能其其实就是聊天室技术在即时通信中的应用。本节我们就来做一个类似 QQ 群功能的聊天室软件。

4.2.1 聊天室功能效果展示

这是本书第一个实用的网络应用软件，为了让读者有一个整体的印象，明确开发目标和编程需求，现将软件完成后的效果提前展示（如图 4.4 所示），它由客户端（SelfChat）和服务器（SelfRoom）两部分构成。



(a) 服务器



(b) 客户端

图 4.4 聊天室软件运行效果

管理员可以配置 IP 端口、开启和关闭服务器，图 4.4（a）左边的监控界面动态实时地显示在线用户列表，以及他们进入、离开聊天室的历史记录和聊天内容。

图 4.4（b）显示的是客户端的界面，用户填写昵称、服务器的 IP 地址和端口后，单击“进入”按钮加入聊天。进入聊天室后就可以发信息了，所发信息的内容会显示在聊天室所有成员的客户端并且在服务器上也会留有记录；新加入的用户除了看到“XX，欢迎您加入！”的信息外，还可以看到在自己加入之前聊天室中的其他成员都聊了些什么，以便较快地融入他们的话题（这个功能与 QQ 群的很相似）。

4.2.2 聊天室的开发

1. 创建工程、添加类、设计界面

分别创建客户端和服务端工程，客户端工程命名为 SelfChat，服务器工程命名为 SelfRoom，其他设置同前。记得一定要在向导的“高级功能”页面上勾选“Windows 套接字”复选框（如图 4.5 所示），因为这个软件是要用到套接字类的。

1) 服务器

用 2.2.4 节的方法给工程添加两个基于 CSocket 的套接字类——CClientSocket 和 CServerSocket，另外添加一个 C++ 结构体 tagHeader，如图 4.6 所示。



图 4.5 在项目中使用 Windows 套接字

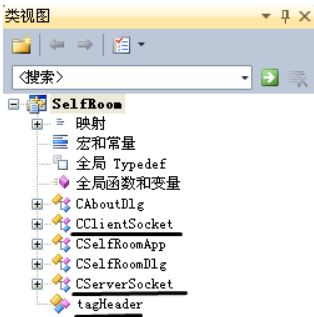


图 4.6 添加类和结构体

CServerSocket 类是用来作为监听的，CClientSocket 类则会为每一个加入聊天室的客户端生成一个相应的 Socket 对象与之通信。在后面读者将看到，这些 Socket 对象会通过指针机制链接成队列，以方便服务器程序管理每个聊天成员。

tagHeader 是客户端和服务端工程都有的一个结构体，在它们中的定义相同。tagHeader.h 的代码如下：

```
typedef struct tagHeader
{
    char    type;           //消息类型
    int     len;            //消息长度
}
```

```

} Header, *pHeader;

#define LOGIN_IO      1          //登录（包括进入和离开）
#define SEND_MESSAGE  3          //发送信息

```

该结构是用于客户端和服务端之间辨识消息类型的，当新成员加入或在线的成员离开聊天室时，其客户端程序都会向服务器发送登录（LOGIN_IO）类型的消息，服务器就能知道聊天室的成员有了变动，于是通知其他客户端动态更新在线用户列表；而聊天室成员之间在交谈时发的谈话内容则是以发送信息（SEND_MESSAGE）类型的消息发给服务器的，如果服务器根据 tagHeader 结构的消息类型（type）字段判断出它只是成员之间的聊天内容，则将它转发给聊天室中的所有其他成员。

设计的服务器界面如图 4.7 所示。



图 4.7 服务器界面

给各控件关联变量，见表 4.1。

表 4.1 聊天室服务器 SelfRoom 界面控件变量

控 件 \ 变 量	Control	Value
“管理员”文本框	m_Admin	m_strName
IP 控件	ServerIP	—
“端口”文本框	ServerPort	sPort
“聊天室动态”只读文本框	m_MessageList	m_history
“用户在线”列表框	m_UserList	—
“开启”按钮	m_Start	—
“关闭”按钮	m_Stop	—
“退出”按钮	m_Exit	—

另外，设置界面中“聊天室动态”只读文本框的 ID 属性为 IDC_EDIT_INFO（后面写程序时需要引用），如图 4.8 所示。

2) 客户端

给客户端添加一个基于 CSocket 的套接字类 CClientSocket（用于连接服务器通信），和服务端一样也要添加一个 C++ 结构体 tagHeader，定义代码与服务器的完全相同。

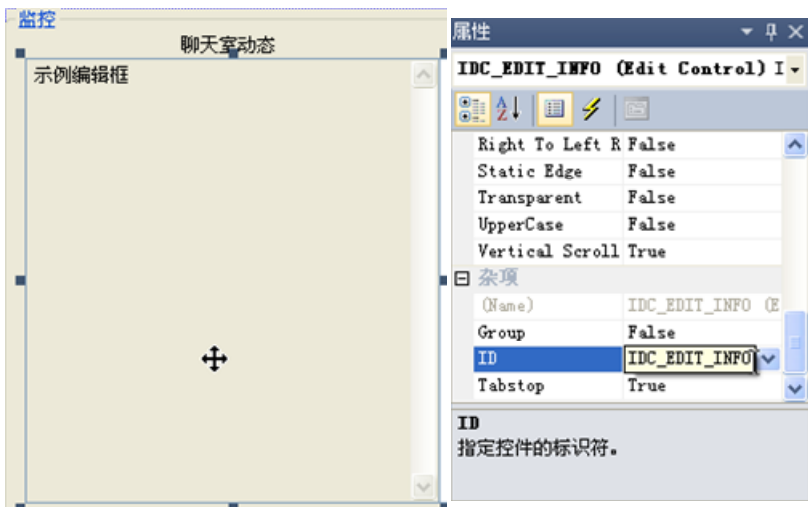


图 4.8 设置“聊天室动态”只读文本框的 ID 属性

设计的客户端界面如图 4.9 所示。

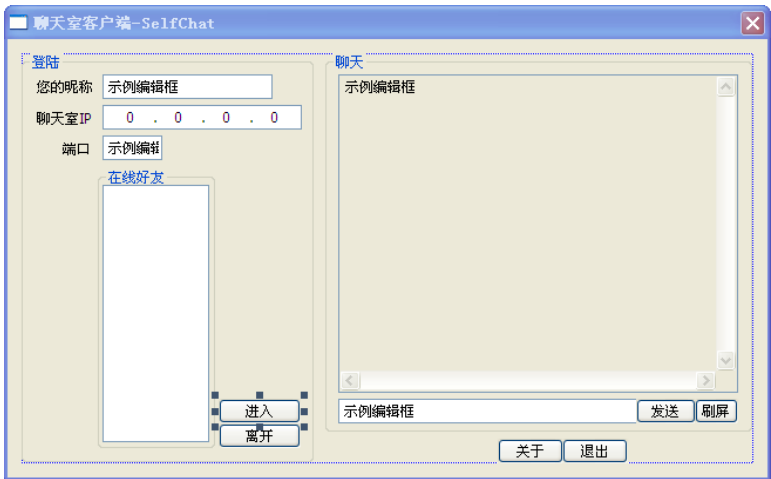


图 4.9 客户端界面

给各控件关联变量，见表 4.2。

表 4.2 聊天室客户端 SelfChat 界面控件变量

控 件 \ 变 量	Control	Value
“您的昵称” 文本框	m_Usr	M_strName
IP 控件	ServerIP	—
“端口” 文本框	m_port	strport
“在线好友” 列表框	m_UserList	—
“聊天” 只读文本框	m_MessageList	—
“发送” 信息文本框	m_EditWords	m_strMessage
“发送” 按钮	m_Send	—

续表

控 件 \ 变 量	Control	Value
“刷屏”按钮	m_Refresh	—
“进入”按钮	m_Enter	—
“离开”按钮	m_Quit	—
“退出”按钮	m_Exit	—

客户端和服务器的界面上都有一个只读文本框，用于显示聊天室状态和历史记录，可以通过对普通文本框设置属性将其定制成如界面上所表现的只读外观，如图 4.10 所示；这个文本框还必须设置为能够显示多行（Multiline）文本，根据界面布局需要，使其具有水平或垂直滚动条。

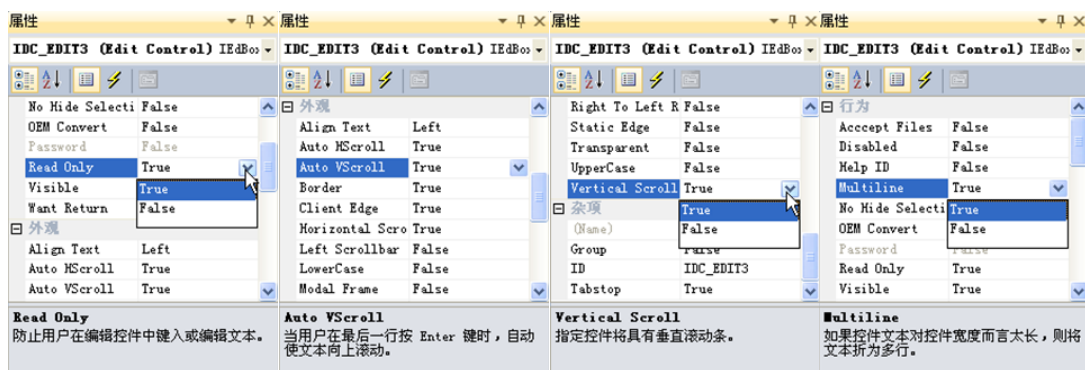


图 4.10 只读文本框外观设置

2. 聊天室的典型运作流程

我们从一次典型的完整聊天过程入手，依次分析出该流程中的各个事件处理过程。显然，从管理员开启聊天室直至最后一个成员离开后管理员关闭聊天服务器，整个流程中的事件时序如下（加下画线的文字为需要编写的处理过程）：

- (1) 服务器单击“开启”按钮。
- (2) 客户端单击“进入”按钮（向服务器发起连接 `Connect()` 请求，若请求被接收，则紧接着发送会话消息）。
- (3) 服务器的 `CServerSocket` 类接收请求并触发 `OnAccept()` 事件，该事件过程创建对应请求方客户端的 `CClientSocket` 类对象并添加到 `Socket` 队列。
- (4) 更多成员客户端加入时，服务器重复步骤（3）。
- (5) 某个成员要发言时，在信息文本框中编辑文字，单击“发送”按钮向服务器发出消息。
- (6) 某个客户端发来的消息到达服务器，触发服务器 `Socket` 队列中对应该客户端 `CClientSocket` 类对象的 `OnReceive()` 事件。

(7) 服务器上对应该客户的 `CClientSocket` 类对象，根据消息头 `tagHeader` 结构中的 `type` 字段辨识消息类型：若 `type = LOGIN_IO`，表示有新成员加入，于是用广播方式通知所有其他成员客户端，并调用服务器 `UpdateUser()` 过程更新服务器用户列表；若 `type = SEND_MESSAGE`，则表示这只是普通的成员发言信息，直接广播转发该消息就可以了。

(8) 服务器发出的消息到达客户端，触发客户端本地 `OnReceive()` 事件，客户端本地 `Socket` 根据消息头 `tagHeader` 结构中的 `type` 字段辨识消息类型：若 `type = LOGIN_IO`，表示有新成员加

入或已有聊天成员下线退出，于是调用客户端 `UpdateUser()` 函数更新本地用户列表；若 `type = SEND_MESSAGE`，则表示这是聊天室中其他成员发来的信息，调用 `GetMsgFromRoom()` 函数接收后显示在客户端屏幕上。

(9) 某个成员要下线时，单击“离开”按钮，关闭本地 Socket。

(10) 一旦某个成员退出，关闭了自己客户端程序的本地 Socket，就会触发其在服务器上 Socket 队列中对应的 `CClientSocket` 类对象的 `OnClose()` 事件，由 `CClientSocket` 类对象发送信息通知其他成员有人退出，并更新服务器用户列表。

(11) 当最后一个成员离开聊天室后，管理员单击“关闭”按钮关闭服务器。

3. 搭建程序框架

为了实现上述流程的功能，需要先定义程序要用到的数据结构和变量，另外包含相关的头文件，为后面所写过程代码的顺利运行搭好基础的框架。

1) 服务器

在头文件 `ClientSocket.h` 中添加代码：

```
#include "SelfRoomDlg.h"           //包含它是为了能进行对话框指针操作
public:
    CString m_strName;              //标识这个 CClientSocket 对象对应的客户端成员昵称
    CPtrList *clist;                //本程序要使用的 Socket 队列数据结构
    CSelfRoomDlg *m_dlgServer;      //为了应用对话框指针机制
    CClientSocket(CPtrList *list);   //此构造函数用于生成一个 Socket 队列
```

`CPtrList` 类支持 void 指针列表，本程序服务器将使用 `CPtrList` 链表类构造一个 Socket 队列数据结构，来统一管理对应各成员客户端的 `CClientSocket` 类对象。

在 `SelfRoom.h` 中定义一个变量：

```
CString m_strName;                //标识管理员名
```

定义这个变量是为了实现本程序额外提供的一个附加功能，即能够在窗口标题栏上实时显示当前使用该软件的用户名（昵称）：

 管理员:张海燕-SelfRoom

这种设计风格是大多数 Windows 程序通用的，大家平时使用 Windows 软件时留心一下就会发现：每个打开的窗口标题栏中都会自动显示与本窗口活动有关的信息。例如，Word 标题栏上会显示当前正在编辑的文档的名称：

第 4 章 即时通信应用开发.docx - Microsoft Word

在学习编程时有意识地模仿一些成功软件产品标准的界面风格是很有好处的，这会使得你写的程序更专业、更具实用性和产品化价值。`m_strName` 就是用于存储当前使用程序的用户名，前面 `ClientSocket.h` 中也定义了这样一个变量，也是同样的用途。

在文件 `SelfRoomDlg.h` 中添加代码：

```
class CServerSocket;               //监听套接字类
class CClientSocket;               //与客户端通信的套接字类
CServerSocket *m_pSocket;
CClientSocket *clientSocket;        //定义这两个套接字指针是为了实现指针机制
void UpdateUser(CClientSocket *pSocket); //更新用户列表的函数
```

代码的添加位置如图 4.11 所示。

在 `ServerSocket.h` 文件的 `CServerSocket` 类声明中定义：

```
CPtrList connectList; //Socket 队列
```

在 ClientSocket.cpp 源文件中添加代码:

```
#include "tagHeader.h"
CClientSocket::CClientSocket(CPtrList *list) : m_dlgServer(NULL)
{
    clist = list; //为实现指针机制
}
```

SelfRoom.cpp 中包含头文件:

```
#include "ServerSocket.h"
```

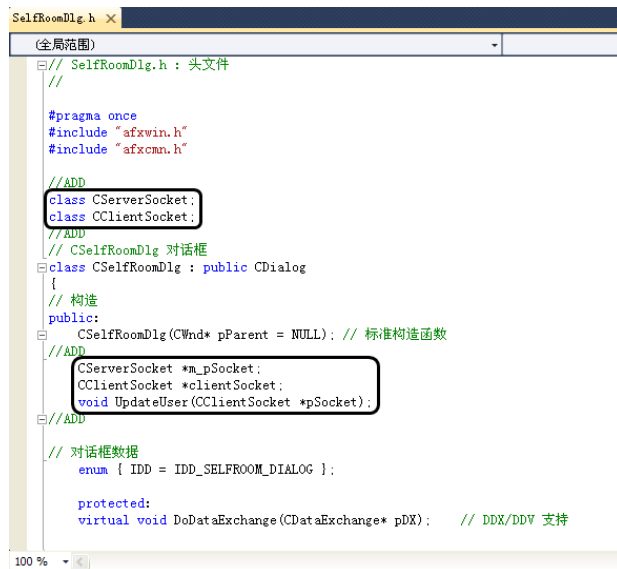


图 4.11 在 SelfRoomDlg.h 中添加代码

SelfRoomDlg.cpp 中包含头文件:

```
#include "ServerSocket.h"
#include "ClientSocket.h"
#include "tagHeader.h"
```

ServerSocket.cpp 中包含头文件:

```
#include "ClientSocket.h"
```

为程序主对话框界面添加初始化代码:

```
m_pSocket = NULL;
clientSocket = NULL;
//初始界面
m_Admin.SetFocus();
m_Stop.EnableWindow(false);
```

添加初始化的基本步骤如图 4.12 所示。

2) 客户端

在 ClientSocket.h 中添加代码:

```
class CSelfChatDlg;
//以下代码在类 CClientSocket 中定义
CSelfChatDlg *chatDlg;
```

```
CString m_strName;
void GetDlg(CSelfChatDlg *dlg);
```

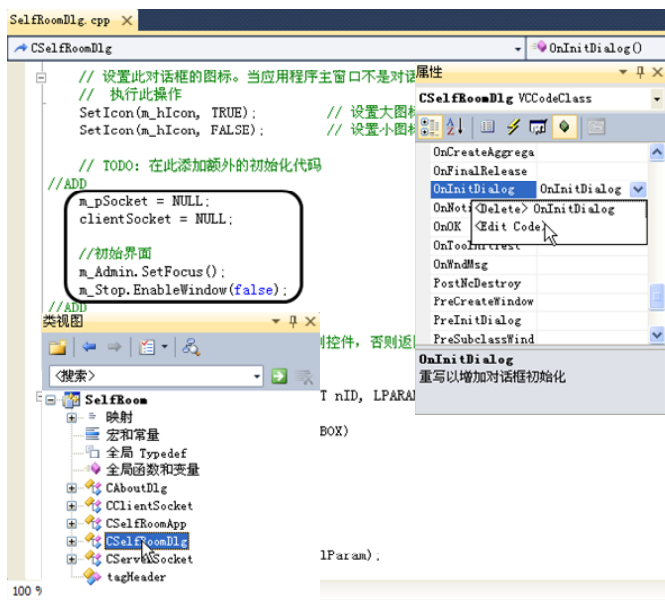


图 4.12 添加初始化的基本步骤

可以看到，这里定义的类 CSelfChatDlg 及指向它的指针 chatDlg，还有 GetDlg()函数都是为建立我们在 2.2.3 节介绍的那种指针机制的编程框架而服务的。理解了这种通用的编程框架，就能很容易地读懂实际应用中大多数网络 Socket 软件的代码了。

在 SelfChat.h 文件的 CselfChatApp 类声明中定义变量：

```
CString m_strName;
```

作用与服务器定义的 m_strName 一样。

在 SelfChatDlg.h 头文件中编写如下代码：

```
#include "ClientSocket.h"
CClientSocket *m_pSocket;           //指针机制
void UpdateUser();                  //更新客户端用户列表
BOOL GetMsgFromRoom();              //接收其他成员发言内容
```

在 ClientSocket.cpp 源文件中包含头文件，并编写指针机制的基础代码 GetDlg()函数实现：

```
#include "SelfChatDlg.h"
#include "tagHeader.h"
void CClientSocket::GetDlg(CSelfChatDlg *dlg) //获得窗口界面的指针
{
    chatDlg = dlg;
}
```

给 SelfChatDlg.cpp 包含头文件：

```
#include "tagHeader.h"
```

为程序主对话框界面添加初始化代码：

```
m_pSocket = NULL;
//初始界面
m_Usr.SetFocus();
```

```
m_Quit.EnableWindow(false);
m_EditWords.EnableWindow(false);
m_Send.EnableWindow(false);
m_Refresh.EnableWindow(false);
```

至此，整个聊天室软件的框架雏形就建立完成了。

4. 添加事件函数、编写处理代码

根据前面分析的聊天室运作流程，依次向程序中添加事件函数并编写处理代码。

服务器单击“开启”按钮，触发的事件过程代码如下：

```
void CSelfRoomDlg::OnStart()
{
    UpdateData(); //刷新界面，获得用户输入
    m_pSocket = new CServerSocket;
    BYTE nFild[4];
    CString sIP;
    ServerIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
    //获取管理员配置的 IP
    sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
    CTime time = CTime::GetCurrentTime(); //获取当前时间
    //下面开始输入完整性验证
    if(m_strName.IsEmpty())
    {
        AfxMessageBox("请先登记管理员名!");
        return;
    }
    if(sIP.IsEmpty())
    {
        AfxMessageBox("请配置聊天室 IP!");
        return;
    }
    if(sPort.IsEmpty())
    {
        AfxMessageBox("请配置要开放的端口!");
        return;
    }
    //界面
    m_Admin.EnableWindow(false);
    ServerIP.EnableWindow(false);
    ServerPort.EnableWindow(false);
    m_Start.EnableWindow(false);
    m_Stop.EnableWindow(true);
    m_Exit.EnableWindow(false);
    //开启聊天室，首先创建用于监听的套接字
    if(m_pSocket->Create(atoi(sPort),1,sIP))
    {
        m_MessageList.SetWindowTextA("");
        m_MessageList.ReplaceSel("聊天室开启成功!\r\n");
    }
}
```

```

CString t = time.Format("%Y-%m-%d");
m_MessageList.ReplaceSel("日期: " + t + "\r\n");
t = time.Format("%H:%M:%S");           //获得聊天室开启的时刻信息
theApp.m_strName = m_strName;          //记录管理员名
m_MessageList.ReplaceSel(t + " 管理员" + theApp.m_strName + " 开放聊天室\r\n");
}
if(m_pSocket->Listen())                 //监听开始
{
    m_MessageList.ReplaceSel("等待成员加入...\r\n");
}
//现在暂时还无人加入聊天, 因此在线用户列表中仅管理员一人
m_UserList.ResetContent();
m_UserList.AddString(theApp.m_strName + "(管理员)");
this->SetWindowTextA("管理员:" + m_strName + "-SelfRoom");
}

```

读者可按照 1.1.4 节介绍的方法添加事件函数, 命名为 `OnStart`, 系统会自动添加这个函数, 并定位到需要编写代码的地方。只要按照 VC 的引导, 在函数体内填写这个代码段就行了, 而 `OnStart` 的函数体框架不需要用户自己写, 上面代码中之所以写出完整的函数体 (包括 `void CSelfRoomDlg::OnStart()` 及前后 `{}`), 只是为了向读者进行全面的展示。

客户端“进入”按钮的事件过程代码如下:

```

void CSelfChatDlg::OnEnter()
{
    UpdateData();
    m_pSocket = new CClientSocket;
    m_pSocket->GetDlg(this);           //获取主界面指针
    BYTE nFild[4];
    CString sip;
    ServerIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
    sip.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
    //获取用户输入的服务器 IP
    //输入完整性验证
    if(m_strName.IsEmpty())
    {
        AfxMessageBox("给自己取个昵称吧!");
        return;
    }
    if(sip.IsEmpty())
    {
        AfxMessageBox("请指定 IP 地址!");
        return;
    }
    if(strport.IsEmpty())
    {
        AfxMessageBox("请指定连接端口!");
        return;
    }
}

```

```

//发起连接
if(!m_pSocket->Create())
{
    AfxMessageBox("网络创建错误!!! ");
    m_pSocket->Close();
    return;
}
if(!m_pSocket->Connect(sip,atoi(strport)))
{
    AfxMessageBox("连接服务器失败!!! ");
    m_pSocket->Close();
    return;
}
//界面
m_Usr.EnableWindow(false);
ServerIP.EnableWindow(false);
m_port.EnableWindow(false);
m_Enter.EnableWindow(false);
m_Quit.EnableWindow(true);
m_EditWords.EnableWindow(true);
m_EditWords.SetFocus();
m_Send.EnableWindow(true);
m_Refresh.EnableWindow(true);
m_Exit.EnableWindow(false);

Header head;
head.type = LOGIN_IO;
head.len = m_strName.GetLength();

m_pSocket->Send((char *)&head,sizeof(Header));
m_pSocket->Send(m_strName,m_strName.GetLength()); //发送会话消息

theApp.m_strName = m_strName;
m_MessageList.SetWindowTextA("");
this->SetWindowTextA(m_strName + "-SelfChat"); //标题栏实时动态显示用户名
}

```

可以看出, 用户单击“进入”按钮后, 客户端程序向服务器发起连接 Connect()函数请求, 若请求被接收, 则紧接着发送会话消息, 与服务器取得联系, 协商相关事宜。这里向服务器发的会话消息头中 head.type = LOGIN_IO 表示是新成员登录, 服务器程序就是靠这个消息头识别它是一个新加入的成员的。

服务器 (严格来说实际上是服务器程序的监听套接字, 即 CServerSocket 类) 接收新成员的 Connect()请求后会触发 OnAccept()事件。

OnAccept()事件的代码如下:

```

//创建 Socket 队列结构
CClientSocket *clientSocket = new CClientSocket(&connectList);
Accept(*clientSocket); //接收连接

```



```
clientSocket->m_dlgServer = (CSelfRoomDlg *):AfxGetMainWnd();
connectList.AddTail(clientSocket);           //在队列中添加新成员的 Socket
```

它创建对应于发起请求的客户端成员的 CClientSocket 类对象，并添加到 Socket 队列。
加入聊天室的用户要发言时，单击“发送”按钮。

“发送”按钮的事件处理代码如下：

```
void CSelfChatDlg::OnSend()
{
    UpdateData();
    if(m_strMessage == "")
    {
        AfxMessageBox("不能发送空消息!");
        m_EditWords.SetFocus();
        return;
    }
    Header head;
    head.type = SEND_MESSAGE;    //这是聊天内容，消息头类型为 SEND_MESSAGE
    head.len = m_strMessage.GetLength();

    m_pSocket->Send((char *)&head,sizeof(Header));
    if(m_pSocket->Send(m_strMessage,m_strMessage.GetLength()))
    {
        m_strMessage = "";
        UpdateData(FALSE);
        m_EditWords.SetFocus();
        return;
    }
    else
    {
        AfxMessageBox("网络传输错误！");
    }
}
```

客户端用户发出的消息到达服务器后，触发服务器 Socket 队列中对应该客户端的套接字（即 CClientSocket 类对象）的 OnReceive()事件，事件过程代码如下：

```
void CClientSocket::OnReceive(int nErrorCode)
{
    char buff1[sizeof(Header)];
    memset(buff1, 0, sizeof(buff1));
    Receive(buff1,sizeof(buff1));    //先接收信息的头部
    this->AsyncSelect(FD_CLOSE|FD_READ|FD_WRITE);

    Header *header = (Header*)buff1;
    int length = header->len;
    char type = header->type;    //解析头部内容
    if(type == LOGIN_IO)    //头部类型为 LOGIN_IO，表示是新成员加入
    {
        char buff[1000];
```

```

memset(buff,0,sizeof(buff));
Receive(buff,length);           //继续接收这条信息的数据部分（新成员名）
this->AsyncSelect(FD_CLOSE|FD_READ|FD_WRITE);
m_dlgServer->UpdateData();
CTime time = CTime::GetCurrentTime();
CString t = time.Format("%H:%M:%S");
CEdit *p_Edit = (CEdit *)::AfxGetMainWnd()->GetDlgItem(IDC_EDIT_INFO);
//生成通知信息
CString strTemp = t + " " + CString(buff) + " 进入聊天室\r\n";
p_Edit->ReplaceSel(strTemp);
m_strName = buff; //将新加入成员的用户名登记在服务器的对应 Socket 中
Header head;
head.type = SEND_MESSAGE;
head.len = strTemp.GetLength();

Header head_history;
head_history.type = SEND_MESSAGE;
//生成“欢迎”信息
m_dlgServer->m_history += m_strName + ", 欢迎您加入!";
head_history.len = m_dlgServer->m_history.GetLength();

CClientSocket *curr = NULL;
POSITION pos = clist->GetHeadPosition();
while (pos != NULL)
{
    curr = (CClientSocket *)clist->GetNext(pos);
    if(curr->m_strName == m_strName)//给新加入的成员发送欢迎信息
    {
        curr->Send((char *)&head_history,sizeof(Header));
        curr->Send(m_dlgServer->m_history,m_dlgServer->m_history.GetLength());
    }
    else                                     //向聊天室中其他的老成员发送通知信息，告知有新成员加入
    {
        curr->Send((char *)&head,sizeof(Header));
        curr->Send(strTemp,strTemp.GetLength());
    }
}
m_dlgServer->UpdateUser(this);    //更新用户列表
}
if(type == SEND_MESSAGE)         //头部类型为 SEND_MESSAGE，表示是成员之间的聊天信息
{
    char buff[1000];
    memset(buff,0,sizeof(buff));
    Receive(buff,sizeof(buff));
    this->AsyncSelect(FD_CLOSE|FD_READ|FD_WRITE);

    CTime time = CTime::GetCurrentTime();

```

```

CString t = time.Format("%H:%M:%S");
CString nikeName = this->m_strName;
CString strTemp = t + " " + nikeName + " 说: " + CString(buff) + "\r\n";
CString str = nikeName + " " + t + "\r\n" + " " + CString(buff);

CEdit *p_Edit = (CEdit *)::AfxGetMainWnd()->GetDlgItem(IDC_EDIT_INFO);
p_Edit->ReplaceSel(strTemp);

CClientSocket *curr = NULL;
POSITION pos = clist->GetHeadPosition();
while (pos != NULL)           //向聊天室内的所有成员转发聊天信息内容
{
    curr = (CClientSocket *)clist->GetNext(pos);
    curr->Send((char *)header,sizeof(Header));
    curr->Send(str,str.GetLength());
}
}
CSocket::OnReceive(nErrorCode);
}

```

由以上 OnSend()和 OnReceive()这两段代码可见,真正实用的聊天软件与第2、3章所介绍的 Socket 程序在发送和接收信息的原理上是完全一样的,都是调用套接字类 Send()函数和 Receive()函数。只不过要对所发送的信息进行一番包装,如本例是在消息前加上一个头部,设定 SEND_MESSAGE 或 LOGIN_IO 类型;接收时先对这个头部进行解析,根据头部字段的不同内容决定不同的处理过程。后面读者就会明白,这其实就是网络协议。

以往的教科书在介绍网络协议时都是很抽象的,初学者往往搞不明白网络协议究竟是个什么东西,怎么才能看到它?有没有可见的实体?

其实简单来说,网络协议就是网络程序在运行过程中互相协调的法则。在上面这个例子中,如果不事先约定消息头部 SEND_MESSAGE 和 LOGIN_IO 分别表示什么含义,通信双方的程序就无法正常地对收到的信息做进一步的处理,因为它无法确定这是“有新人登录”还是已加入的成员之间发的普通聊天内容。只有在客户端和服务端都事先进行了这种约定的情况下,双方程序才能默契配合,共同完成聊天室的功能。

在本书 2.3.2 节曾经做了一个用自己编写的 Socket 程序接入他人开发的聊天室软件的实验,实验结果还算成功,主要是因为他人开发的聊天室的运作机制十分简单:服务器只是简单地转发信息,并没有与客户端协商什么复杂的协议。假如该客户端与服务器事先约定好某种协议,用于验证发起连接请求的是不是与自身软件配套的客户端程序(就像大家在谍战剧中看到的地下工作者接头时都要求说出事先约定的暗语一样),那么我们的实验就不会这么轻易成功了。

由前面 OnReceive()函数的代码可知,当服务器检测到有新成员加入时,会调用 UpdateUser()函数更新用户列表。

UpdateUser()函数过程代码如下:

```

void CSelfRoomDlg::UpdateUser(CClientSocket *pSocket)
{
    m_UserList.ResetContent();
    m_UserList.AddString(theApp.m_strName + "(管理员)");    //首先添加管理员用户
    CString user_info;

```

```

user_info = theApp.m_strName + "(管理员)" + "&";
if(pSocket != NULL)
{
    CClientSocket *pSock = NULL;
    POSITION pos = pSocket->clist->GetHeadPosition();
    while(pos != NULL)
    {
        pSock = (CClientSocket *)pSocket->clist->GetNext(pos);
        m_UserList.AddString(pSock->m_strName);           //然后逐个添加已经记录的用户名
        user_info += (pSock->m_strName + "&");
    }
    Header head;
    head.type = LOGIN_IO;
    head.len = user_info.GetLength();

    POSITION po = pSocket->clist->GetHeadPosition();
    while(po != NULL)                                   //将最新的用户列表转发给各个用户
    {
        pSock = (CClientSocket *)pSocket->clist->GetNext(po);
        pSock->Send((char *)&head, sizeof(Header));
        pSock->Send((LPCTSTR)user_info, user_info.GetLength());
    }
}
}

```

可见，服务器在更新自身用户列表的同时，还将最新的在线用户名单信息转发给各个客户端，客户端用户列表的更新也依赖于服务器提供的名单，所以无论是聊天内容还是用户在线状态的获取，都要通过服务器。服务器在整个聊天室中扮演的是处于中心地位的“中转者”角色，所有用户的聊天内容、行为状态都在服务器上留有完全的记录，因此这个软件的架构方式是典型的 C/S 结构。

客户端接收到服务器发来的信息后又会怎么处理呢？

客户端本地 OnReceive() 事件过程（这个事件过程是客户端本地 Socket 类的）如下：

```

void CClientSocket::OnReceive(int nErrorCode)
{
    char buff[sizeof(Header)];
    memset(buff,0,sizeof(buff));
    Receive(buff,sizeof(buff));           //收到服务器发来的消息
    this->AsyncSelect(FD_CLOSE|FD_READ|FD_WRITE);

    Header *header = (Header*)buff;
    int length = header->len;
    char type = header->type;             //解析消息头部
    if(type == SEND_MESSAGE)             //如果是聊天内容，则直接接收
    {
        chatDlg->GetMsgFromRoom();
    }
    if(type == LOGIN_IO)                 //在线用户有变化，同步更新用户表
    {

```

```

        chatDlg->UpdateUser();
    }
    CSocket::OnReceive(nErrorCode);
}

```

可以看到，客户端也是根据接收消息的头部字段判断消息类型从而分别进行相应处理的，上面代码中对于两类不同的消息分别调用了两个处理——GetMsgFromRoom()和 UpdateUser()函数，它们的程序代码分别如下。

GetMsgFromRoom()函数：

```

BOOL CSelfChatDlg::GetMsgFromRoom()
{
    char buff[1000];
    memset(buff,0,sizeof(buff));
    m_pSocket->Receive(buff, sizeof(buff));    //接收信息
    m_pSocket->AsyncSelect(FD_CLOSE|FD_READ|FD_WRITE);

    CString strTemp = buff;
    strTemp += "\r\n";
    m_MessageList.ReplaceSel(strTemp);        //直接显示在界面上即可
    return TRUE;
}

```

UpdateUser()函数：

```

void CSelfChatDlg::UpdateUser()
{
    char buff[1000];
    memset(buff,0,sizeof(buff));
    m_pSocket->Receive(buff, sizeof(buff));    //接收到最新的在线用户名单
    m_pSocket->AsyncSelect(FD_CLOSE|FD_READ|FD_WRITE);

    CString user_info = buff;
    CString array[100];
    int b = 0;
    for( int i = 0; i < user_info.GetLength(); i++ )    //遍历名单中的所有用户
    {
        if(i != (user_info.GetLength() - 1))
        {
            if ( user_info[i] == '&' )
            {
                b ++;
            }
            else
            {
                array[b] = array[b] + user_info[i];
            }
        }
    }
    m_UserList.ResetContent();
}

```

```

for(int j = 0; j < b + 1; j++)
{
    m_UserList.AddString(array[j]);    //用户名依次显示于界面列表中
}

```

以上所介绍的程序段都是聊天室日常运行过程中所执行的代码,当用户陆续离开聊天室……直至管理员关闭聊天服务器,这中间会执行哪些处理过程呢?

某个成员要下线时,单击“离开”按钮,关闭本地 Socket。

客户端“离开”按钮的事件过程代码如下:

```

void CSelfChatDlg::OnQuit()
{
    if(m_pSocket)
    {
        m_pSocket->Close();
        delete m_pSocket;    //离开时要记得关闭、销毁 Socket
    }
    m_UserList.ResetContent();
    m_MessageList.ReplaceSel("你已经退出了, 谢谢光顾!");
    this->SetWindowTextA("聊天室客户端-SelfChat");
    //界面
    m_Usr.EnableWindow(true);
    m_Usr.SetFocus();
    ServerIP.EnableWindow(true);
    m_port.EnableWindow(true);
    m_Enter.EnableWindow(true);
    m_Quit.EnableWindow(false);
    m_EditWords.EnableWindow(false);
    m_Send.EnableWindow(false);
    m_Exit.EnableWindow(true);
}

```

一旦某个成员退出,关闭了自己客户端程序的本地 Socket,就会触发其在服务器上 Socket 队列中对应的 CClientSocket 类对象的 OnClose()事件。

服务器 OnClose()事件过程代码如下:

```

void CClientSocket::OnClose(int nErrorCode)
{
    POSITION pos = clist->Find(this);
    if(pos != NULL)
    {
        clist->RemoveAt(pos);    //移除服务器 Socket 队列中的套接字

        CTime time = CTime::GetCurrentTime();
        CString t = time.Format("%H:%M:%S");
        CEdit *p_Edit = (CEdit *)m_dlgServer->GetDlgItem(IDC_EDIT_INFO);
        CString strTemp = t + " " + this->m_strName + " 离开聊天室\r\n";
        p_Edit->ReplaceSel(strTemp);
    }
}

```

```

Header head;
head.type = SEND_MESSAGE;
head.len = strTemp.GetLength();
CClientSocket *curr = NULL;
POSITION pos = clist->GetHeadPosition();
while (pos != NULL)           //将此用户离开的消息告知其他成员
{
    curr = (CClientSocket *)clist->GetNext(pos);
    curr->Send((char *)&head,sizeof(Header));
    curr->Send(strTemp,strTemp.GetLength());
}
m_dlgServer->UpdateUser(this);    //更新服务器用户列表
this->Close();
delete this;
}
CSocket::OnClose(nErrorCode);
}

```

聊天室管理员有权限随时单击“关闭”按钮关闭聊天服务器。

“关闭”按钮的事件过程代码如下：

```

void CSelfRoomDlg::OnStop()
{
    m_pSocket->Close();
    m_pSocket = NULL;
    CTime time = CTime::GetCurrentTime();
    CString t = time.Format("%H:%M:%S");
    CString strTemp = t + " 管理员" + theApp.m_strName + " 关闭聊天室\r\n";
    m_MessageList.ReplaceSel(strTemp);
    m_UserList.ResetContent();
    this->SetWindowTextA("聊天室管理-SelfRoom");
    //界面
    m_Admin.EnableWindow(true);
    m_Admin.SetFocus();
    ServerIP.EnableWindow(true);
    ServerPort.EnableWindow(true);
    m_Start.EnableWindow(true);
    m_Stop.EnableWindow(false);
    m_Exit.EnableWindow(true);
}

```

至此，这个聊天室软件的基本代码都展示出来了，还有少量次要的代码，如显示版本信息的“关于”对话框，清空界面上显示的历史记录，界面控制……这些代码对实现聊天室的功能并不是必需的。界面控制代码在第2章中已经做了详细介绍，并且在上面的程序段中都已附带贴出，还加了注释，读者可以自己对照着软件运行时的界面现象去理解，当然也可以自己改造优化。

“关于”对话框的代码与第2章的一模一样，弹出的“关于”对话框版本信息如图4.13所示。

客户端“刷屏”按钮是用来清空界面上的历史记录，其事件过程很简单，仅一句代码：

```

void CSelfChatDlg::OnRefresh()
{

```

```
m_MessageList.SetWindowTextA("");
}
```

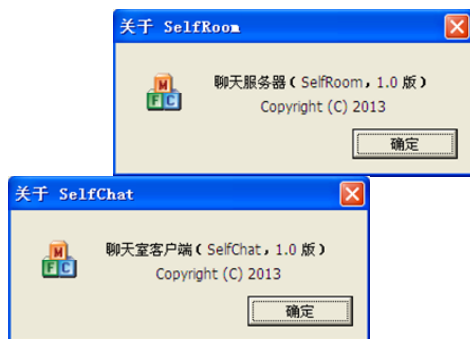


图 4.13 聊天室的“关于”对话框

这样，一个功能完整的聊天室软件就编写完成了。读者可以自己运行一下看看效果。

4.3 P2P 架构的简单聊天工具

4.3.1 软件使用效果展示

前面 4.1.2 节说过，还有一种以 P2P 方式架构的聊天系统。这类系统的所有用户终端都具有完全一样的界面，既可用作聊天终端程序也可作为服务器来使用。使用时，任选一个运行的终端作为服务器，其余的用户终端都在这个服务器的协调下交换信息，如图 4.14 所示。

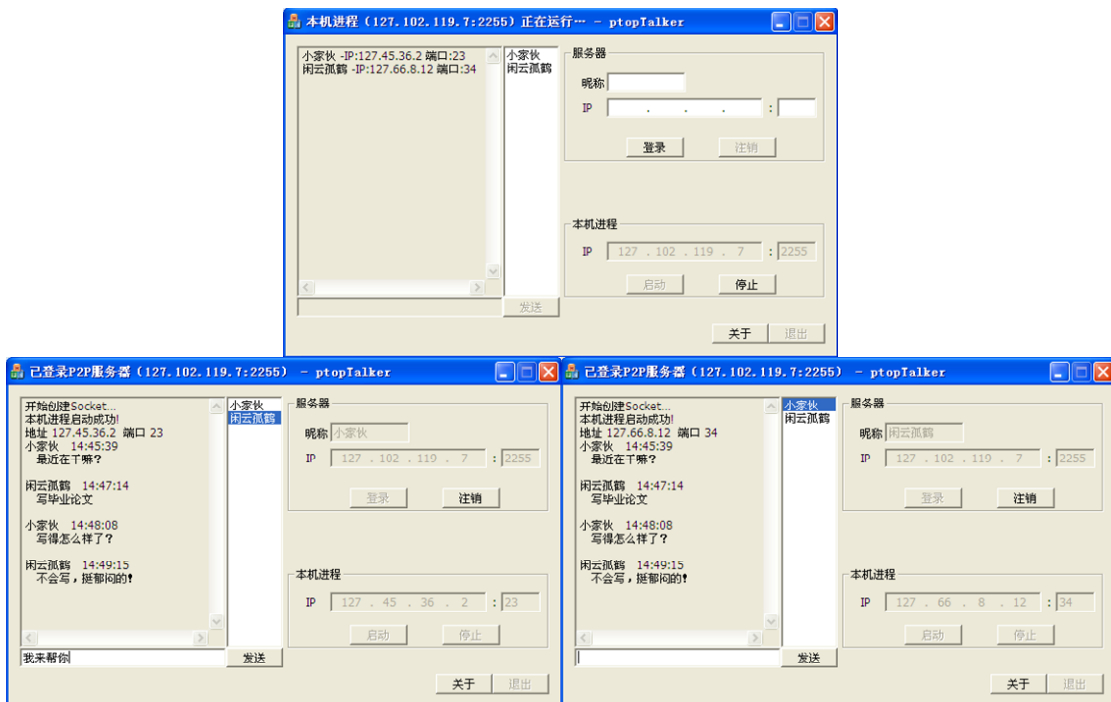


图 4.14 P2P 软件使用效果

图 4.14 中展示了两个网名分别为“小家伙”和“闲云孤鹤”的网友从各自的终端登录地址为 127.102.119.7:2255 的服务器进行点对点即时聊天的画面。双方登录后都可以在界面上的在线用户列表中看到对方,在下方文本框里编辑自己想说的话,从用户列表中选中对方的昵称,单击“发送”按钮,就可以将信息发到对方的终端上,这个与大家平日使用的 QQ 是相似的。

4.3.2 P2P 通信规约

在开发这个软件之前,首先就服务器与终端之间以及各用户终端之间的交互做一些约定,我们把这种约定称为“通信规约”。它是参与通信的各方预先商量好的规则,要保证能够顺利地实现上述的 P2P 通信,就必须遵守这个规则。

1. 在线用户变更

在通信中,会不断有新用户加入和老用户下线退出。无论是加入还是退出,都必须让在线的其他用户及时了解到这个动态。本例中这个功能是采用向服务器发送注册(销)类消息来实现的。服务器接收到这个消息就知道在线用户有变动(有人加入或退出了),于是从这个消息中解析出究竟是谁加入或退出了,再由服务器将这一消息转告目前仍然在线的其他用户的终端,使其终端上的在线用户列表随之动态更新。注册(销)类消息的格式如下。

● 注册(销)类消息:

RGST_ADDR; 用户名, 127.*.*.*: 端口号

本例所使用的消息均由首部(报文头)和数据两部分构成,首部的报文头指示消息类型,数据部分携带的是有关这个消息要传达的具体信息,两者之间用“;”隔开,形成【首部;数据】的统一格式。此处消息头 RGST_ADDR 代表消息类型为注册(销)类,后面数据部分的第一个字段为新注册(销)用户的用户名,第二个字段为其 IP 和端口号,两者以“:”分隔,一同组成标识该用户终端的完整地址。

服务器收到注册(销)类消息,经过更新处理,向各终端发出更新在线用户列表的通知,这个通知是使用在线更新类消息传达的。

● 在线更新类消息:

UPDT_USR; 用户名 1&用户名 2&用户名 3&...

这个消息的数据部分携带了最新的在线用户列表,每个在线的用户终端收到它后就可以根据这个信息更新自己的用户列表显示了。

2. P2P 通信服务

当某用户要与另一个用户通信(互发信息聊天)时,必须经由服务器从中协调,服务器为通信的双方提供 P2P 服务。

假设用户甲要与用户乙通信,具体过程为:甲的终端向服务器发出请求,即向服务器索要用户乙的联系信息。甲的请求是通过请求 P2P 服务类消息发到服务器的。

● 请求 P2P 服务类消息:

REQ_ADDR; 用户名(对方), 127.*.*.*: 端口号(己方)

消息数据部分的用户名是对方(乙)的,而地址字段填写的则是甲自己的地址,这是为了方便服务器收到此消息后根据这个地址给甲的终端回复响应。服务器在收到甲的请求后,根据甲提供的用户名,找出对应该用户名的终端地址,并返回告知甲。服务器是在返回的响应中将乙的用户名提供给甲的,响应消息的格式如下。

- P2P 服务响应类消息:

RSP_ADDR; 用户名, 127.*.*: 端口号 (对方)

这里的用户名和返回地址都是对方 (乙) 的, 甲收到这个消息, 从中解析出乙的地址信息, 就可以与乙建立点对点的通信了。

3. P2P 信息收发

甲得到乙的联系方式 (IP+端口) 后, 就可以直接向乙发信息了。甲与乙之间的通信内容借助信息收发类消息传送, 其格式如下。

- 信息收发类消息:

SND_MSG; 用户名 (发送方), 内容

其中用户名是发送方的昵称, 在这里是甲的用户名 (甲登录时填写的昵称), 若为乙向甲回信息, 则消息中的用户名是乙的昵称。后面的“内容”字段就是双方聊天的信息内容。

经由上述这一系列的规定, 系统中任何两个用户的终端之间就都可以进行点对点通信了。读者从中可以看出, 这个规约假定通信主体只要知道了对方的地址信息 (IP+端口), 就可以直接向对方发送信息而不需要事先建立连接——这正是 UDP 的特征。利用第 3 章学过的 UDP Socket 编程的知识, 就可以实现上述规约所定义的这个 P2P 通信过程。

4.3.3 聊天工具的开发过程

1. 创建项目工程, 设计软件界面

新建 MFC 项目工程, 工程命名为 P2PTalker (P2P 交谈者), 其余设置与 4.2 节的聊天室程序相同。本软件只需创建单独的一个工程作为用户终端 (兼作客户端和服务端), 用户终端界面设计如图 4.15 所示。

界面上①所标示的控件为编辑框 (Edit Control), ②是列表控件 (Listbox Control)。文本框的“Multiline”和“Read Only”属性均设为“True” (多行只读文本), 用于显示程序运行的状态信息及聊天内容; 列表控件则是用于动态显示在线用户列表的。

将 VC 界面设计区初始默认的“确定”按钮更名为本程序的“启动”按钮, 原来的“取消”按钮更名为“退出”按钮。

给软件界面上的各个控件关联变量, 见表 4.3。

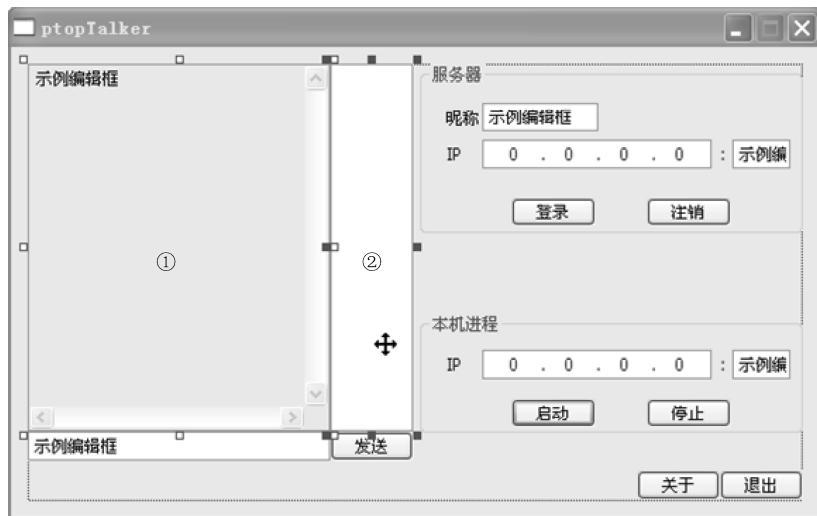


图 4.15 用户终端界面

表 4.3 P2P 聊天工具 P2PTalker 界面控件变量

控 件 \ 变 量	Control	Value
“昵称” 文本框	m_nicnam	nickname
服务器 IP 地址控件	SevrIP	—
“服务器” 端口文本框	m_SvrPrt	sevrPort
“登录” 按钮	m_Login	—
“注销” 按钮	m_Logout	—
“本机进程” IP 地址控件	LocalIP	—
“本机进程” 端口文本框	m_LcaPrt	locaPort
“启动” 按钮	m_Start	—
“停止” 按钮	m_Stop	—
聊天内容状态监控只读文本框①	m_MessageList	—
在线用户列表②	m_UserList	—
发送信息编辑框	m_EditWords	strMessage
“发送” 按钮	m_Send	—
“退出” 按钮	m_exit	—

2. 构建软件的框架

由于这个 P2P 软件程序必须遵照 4.3.2 节所定的通信规约编制，故较本书前面部分所编写的那些程序要稍显复杂些。必须首先为实现前定的规约标准做准备，预先定义相关的结构体、变量，还要仔细设计各个功能函数（方法）并事先声明它们，另外建立好程序的消息驱动框架……这一系列的工作虽然烦琐，却对整个软件基本功能的实现起着举足轻重的作用。尽管这个过程中并没有编写一句实现某项具体功能的代码，但是这一阶段所编写的定义、声明加上设置代码一起构建了这个 P2P 软件的框架。

下面就请读者按照书上的操作一步一步地完成这个过程。

1) 定义结构体

在项目中添加定义两个结构体——MyMsg 和 rgstdUsr，其中 MyMsg 代表聊天用户收发信息的信息结构，而 rgstdUsr 则用来保存服务器注册用户的信息。

具体操作方法为，向项目中添加两个头文件 MyMsg.h 和 rgstdUsr.h，添加完后在其中编写定义代码。

MyMsg 的定义在 MyMsg.h 中：

```
typedef struct MyMsg
{
    char msg[100];
    int i;
}Msg;
```

rgstdUsr 的定义在 rgstdUsr.h 中:

```
typedef struct rgstdUsr
{
    char        username[10];           //用户名（昵称）
    BYTE        ipFild[4];              //用户终端 IP
    int         port;                  //用户终端端口
    BOOL        onlStat;               //用户在线状态
} RgstdUsr,    *pRgstdUsr;
```

为了使本程序的代码能够访问到这两个结构体,必须将定义它们的头文件包含到项目中。在 P2PTalkerDlg.h 中包含头文件:

```
#include "rgstdUsr.h"
#include "MyMsg.h"
```

包含了这两个头文件后,大家就会在“类视图”中看到刚刚定义的两个结构体,它们已经融入程序框架中了,如图 4.16 所示。

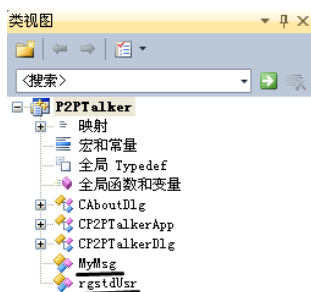


图 4.16 用户自定义结构体

2) 定义变量和声明方法

在 P2PTalkerDlg.h 文件的主对话框类 CP2PTalkerDlg 中定义如下变量:

```
public:
    SOCKET SelfSndSocket;           //本机进程用于发送消息的 SOCKET
    SOCKET SelfRcvSocket;           //本机进程用于接收消息的 SOCKET
    SOCKADDR_IN m_sockSelfSndAddr;  //SOCKET 结构, 发送进程默认地址 (127.0.0.1:8888)
    SOCKADDR_IN m_sockSelfRcvAddr;  //SOCKET 结构, 接收进程地址 (标识各个端系统)
    SOCKADDR_IN m_sockAddrto;       //SOCKET 结构, 指定将要向哪个地址发消息
    RgstdUsr rstUsr[100];
    int rstUsrCont;                 //注册用户数
    int onlUsrCont;                 //在线用户数
    Msg msgbuf;
    CString RegisterAddr;           //注册 (销) 类消息
    CString OnlUsrInfo;             //在线更新类消息
    CString PtoPsrVReqst;           //服务请求类消息
    CString PtoPsrVResp;           //服务响应类消息
    CString MsgTalktoPeer;          //聊天信息类消息
    int socklen;
```

声明如下方法:

```
void EnrolUsr(CString strMsgData);  //登记注册 (销) 用户
void UpdUsr(CString strMsgData);   //更新用户列表
```

```
void PtoPrvProvdr(CString strMsgData);    //提供 P2P 地址服务
void SendMsgtoPer(CString strMsgData);    //向对方发送聊天信息
void ShowMsgfrmPer(CString strMsgData);    //显示对方发来的聊天内容
//临时开启一个专用于发送消息的套接字
void StartSndSocket(CString msgtoSnd,CString sIP,CString sPort);
```

3) 建立程序的事件驱动机制

本软件的收发消息功能直接调用 Winsock API 编程,这样就必须建立某种事件驱动机制以保证收发数据的正常进行。这里采用第 3 章 UDP 编程中使用过的通过 VC 宏映射机制自定义消息以实现网络事件的通知的方法,具体操作如下。

在 stdafx.h 文件中定义宏:

```
#define WM_CLIENT_READMSG WM_USER + 102
```

在 P2PTalkerDlg.cpp 中添加定义消息宏映射:

```
ON_MESSAGE(WM_CLIENT_READMSG,OnReadMsg)
```

然后在 P2PTalkerDlg.h 中包含头文件:

```
#include "stdafx.h"
```

并在主对话框类 CP2PTalkerDlg 的定义代码中声明方法 OnReadMsg():

```
LRESULT OnReadMsg(WPARAM wParam,LPARAM lParam);
```

OnReadMsg()代码如下:

```
LRESULT CP2PTalkerDlg::OnReadMsg(WPARAM wParam,LPARAM lParam)
```

```
{
```

```
    //接收其他端系统发来的消息
```

```
    CString str,strHead,strData;
```

```
    switch (WSAGETSELECTEVENT(lParam))
```

```
    {
```

```
        case FD_READ:
```

```
        socklen = sizeof(m_sockSelfRcvAddr);
```

```
        recvfrom(SelfRcvSocket,(char *)&msgbuf,sizeof(msgbuf),0,
                (LPSOCKADDR)&m_sockSelfRcvAddr,(int *)&socklen);
```

```
        WSAAsyncSelect(SelfRcvSocket,m_hWnd,WM_CLIENT_READCLOSE,FD_READ);
```

```
        str.Format("%s",msgbuf.msg);
```

```
        //截取报文头和数据部分
```

```
        int n = str.ReverseFind(';');
```

```
        strHead = str.Left(n);
```

```
        strData = str.Right(str.GetLength() - (n + 1));
```

```
        //根据不同的报文头类型执行不同的操作 (遵照 4.3.2 节制定的通信规约)
```

```
        if(strHead == "RGST_ADDR")
```

```
        {
```

```
            this->EnrolUsr(strData);                //登记注册 (销) 用户
```

```
        }
```

```
        if(strHead == "UPDT_USR")
```

```
        {
```

```
            this->UpdUsr(strData);                //更新用户列表
```

```
        }
```

```
        if(strHead == "REQ_ADDR")
```

```
        {
```

```

        this->PtoPrvProvdr(strData);           //提供 P2P 地址服务
    }
    if(strHead == "RSP_ADDR")
    {
        this->SendMsgtoPer(strData);           //向对方发送聊天信息
    }
    if(strHead == "SND_MSG")
    {
        this->ShowMsgfrmPer(strData);         //显示对方发来的聊天内容
    }
    break;
}
return 0L;
}

```

进入消息响应 OnReadMsg()函数后, 先对收到的消息报文进行解析, 再根据首部类型按照前面介绍的 P2P 通信规约调用不同的处理方法。至于整个过程中各方法调用的流程顺序, 下面即将介绍。

4) 其他初始化代码

软件启动运行时对界面的初始化代码在源文件 P2PTalkerDlg.cpp 的 BOOL CP2PTalkerDlg::OnInitDialog()方法中:

```

rstUsrCont = 0;
onlUsrCont = 0;
//界面
m_Stop.EnableWindow(false);
m_Login.EnableWindow(false);
m_Logout.EnableWindow(false);
SevrIP.EnableWindow(false);
m_SvrPrt.EnableWindow(false);
m_nicnam.EnableWindow(false);
m_UserList.EnableWindow(false);
m_MessageList.EnableWindow(false);
m_EditWords.EnableWindow(false);
m_Send.EnableWindow(false);

```

3. 程序流程

在正式开始编程实现之前, 先来从总体上梳理一下整个软件的工作流程。

1) 用户加入和退出

- 软件终端运行后, 用户在配置完本地的 IP 和端口后, 单击“启动”按钮, 触发 OnOK()方法, 开启本机进程。
- 选定其中一个用户的终端作为服务器, 其他用户在各自的终端界面上填写网名昵称和服务地址后, 单击“登录”按钮, 终端自动生成注册(销)类消息并调用 StartSndSocket()方法将消息发往服务器。
- 服务器收到消息后, 调用 EnrolUsr()方法更新服务器上保存的用户信息并生成在线更新类消息广播, 发给所有在线的用户终端。
- 用户终端收到服务器的更新用户通知后, 执行 UpdUsr()方法刷新在线用户的列表显示。

2) 收发信息的 P2P 过程

- 当用户甲要与乙进行 P2P 通信时, 在终端的发送信息文本框内编辑文字信息, 编辑完后从界面上的在线用户列表中选中对方(乙)的用户名, 单击“发送”按钮, 启动事件过程 OnSend(), 向服务器发出请求 P2P 服务类消息。
- 服务器收到请求后, 执行 PtoPsrsvProvdrr()方法, 并向用户返回 P2P 服务响应类消息。
- 用户甲的终端收到服务器返回的响应消息后, 向对方(乙)的终端发出 P2P 通信(聊天)内容, 这个内容由信息收发类消息携带, 使用 SendMsgtoPer()方法发送出去。
- 对方(乙)收到后通过 ShowMsgfrmPer()方法将聊天信息的内容在终端界面上显示出来。

4. 编程实现

1) 用户加入和退出

“启动”按钮是由 VC 环境界面设计区初始默认的“确定”按钮更名而来的, 单击它会触发程序主对话框类的 OnOK()方法。代码如下:

```
void CP2PTalkerDlg::OnOK()
{
    //输入验证
    UpdateData();
    if(LocalIP.IsBlank())
    {
        AfxMessageBox("请设置本机 IP 地址!");
        return;
    }
    if(localPort.IsEmpty())
    {
        AfxMessageBox("请指定通信端口号!");
        return;
    }
    //初始化套接字与绑定
    WSADATA wsaData;
    int iErrorCode;
    if (WSAStartup(MAKEWORD(2,1),&wsaData))    //调用 Windows Sockets DLL
    {
        m_MessageList.ReplaceSel("Winsock 无法初始化!\r\n");
        WSACleanup();
        return;
    }
    m_MessageList.ReplaceSel("开始创建 Socket...\r\n");
    //创建本机进程的 Socket, 类型为 SOCK_DGRAM, 无连接的通信
    SelfRcvSocket = socket(PF_INET,SOCK_DGRAM,0);
    if(SelfRcvSocket == INVALID_SOCKET)
    {
        m_MessageList.ReplaceSel("创建 socket 失败!\r\n");
        return;
    }
    //获取本机进程的 IP 和端口
    BYTE nField[4];
```

```

CString sIP;
LocalIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
m_sockSelfRcvAddr.sin_family = AF_INET;
m_sockSelfRcvAddr.sin_addr.S_un.S_addr = inet_addr(sIP);
m_sockSelfRcvAddr.sin_port = htons(atoi(locaPort));
//将套接字与本机进程绑定
if(bind(SelfRcvSocket,(LPSOCKADDR)&m_sockSelfRcvAddr,
        sizeof(m_sockSelfRcvAddr)) == SOCKET_ERROR)
{
    m_MessageList.ReplaceSel("绑定失败!\r\n");
    return;
}
iErrorCode = WSAAsyncSelect(SelfRcvSocket,m_hWnd,WM_CLIENT_READCLOSE, FD_READ);
if (iErrorCode == SOCKET_ERROR)
{
    m_MessageList.ReplaceSel("WSAAsyncSelect 设定失败!——用于连接请求的消息\r\n");
    return;
}
m_MessageList.ReplaceSel("本机进程启动成功!\r\n");
m_MessageList.ReplaceSel("地址" + sIP + " 端口" + locaPort);
this->SetWindowTextA("本机进程 (" + sIP + ":" + locaPort + ") 正在运行...- ptopTalker");
//界面
m_Start.EnableWindow(false);
m_Stop.EnableWindow(true);
LocalIP.EnableWindow(false);
m_LcaPrt.EnableWindow(false);
m_Login.EnableWindow(true);
SevrIP.EnableWindow(true);
m_SvrPrt.EnableWindow(true);
m_nicnam.EnableWindow(true);
m_MessageList.EnableWindow(true);
m_exit.EnableWindow(false);
m_nicnam.SetFocus();
return;
CDialog::OnOK();
}

```

进程启动后, 创建数据报套接字 SOCK_DGRAM, 也要经历加载协议栈 (WSAStartup()) → 创建套接字 (socket()) → 绑定 (bind()) 这样三步。创建完成后, 用 WSAAsyncSelect() 方法随时接收来自网络的通知事件。

选定服务器后, 任何终端只需填写好自己的用户名 (昵称) 和服务器对应的进程地址, 再单击 “登录” 按钮, 就可以在服务器上注册个人信息, 加入这个 P2P 群。

“登录” 按钮的事件过程代码如下:

```

void CP2PTalkerDlg::OnLogin()
{
    //输入验证

```



```

UpdateData();
if(nickname.IsEmpty())
{
    AfxMessageBox("给自己取个昵称吧!");
    return;
}
if(SevrIP.IsBlank())
{
    AfxMessageBox("请填写选做 P2P 服务器的主机 IP 地址!");
    return;
}
if(sevrPort.IsEmpty())
{
    AfxMessageBox("请指定 P2P 服务进程的端口!");
    return;
}
BYTE nFild[4];
CString sIP;
//生成登录注册报文
LocalIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
RegisterAddr = "RGST_ADDR";
RegisterAddr += nickname + "," + sIP;
RegisterAddr += ":" + locaPort;
//获取 P2P 服务进程的 IP 和端口
SevrIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
//发送报文
this->StartSndSocket(RegisterAddr,sIP,sevrPort);
this->SetWindowTextA("已登录 P2P 服务器 (" + sIP + ":" + sevrPort + ") - ptopTalker");
//界面
m_Stop.EnableWindow(false);
m_Login.EnableWindow(false);
m_Logout.EnableWindow(true);
SevrIP.EnableWindow(false);
m_SvrPrt.EnableWindow(false);
m_nicnam.EnableWindow(false);
m_UserList.EnableWindow(true);
m_EditWords.EnableWindow(true);
m_Send.EnableWindow(true);
m_EditWords.SetFocus();
}

```

终端将自动按前面 4.3.2 节的规约生成特定格式的消息报文，并调用 StartSndSocket()方法将此报文发往服务器。

在本例中，已登录加入的终端可以随时从服务器上退出（又称注销），注销与注册所使用的是同一个报文，都是以 RGST_ADDR 作为首部的。

“注销”按钮的事件过程代码如下：

```
void CP2PTalkerDlg::OnLogout()
{
    UpdateData();
    BYTE nFild[4];
    CString sIP;
    //生成注销报文
    LocalIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
    sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
    RegisterAddr = "RGST_ADDR";
    RegisterAddr += nickname + "," + sIP;
    RegisterAddr += ":" + locaPort;
    //获取 P2P 服务进程的 IP 和端口
    SevrIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
    sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
    //发送报文
    this->StartSndSocket(RegisterAddr,sIP,sevrPort);
    m_UserList.ResetContent();
    this->SetWindowTextA("注销成功! - ptopTalker");
    //界面
    m_Stop.EnableWindow(true);
    m_Login.EnableWindow(true);
    m_Logout.EnableWindow(false);
    SevrIP.EnableWindow(true);
    m_SvrPrt.EnableWindow(true);
    m_nicnam.EnableWindow(true);
    m_UserList.EnableWindow(false);
    m_EditWords.EnableWindow(false);
    m_Send.EnableWindow(false);
    m_nicnam.SetFocus();
}
```

向服务器发送报文的操作是通用的，因此这里专门封装了一个方法——StartSndSocket()来实现它。

StartSndSocket()方法代码如下：

```
void CP2PTalkerDlg::StartSndSocket(CString msgtoSnd,CString sIP, CString sPort)
{
    //另启一个专门用于发消息的 Socket 进程
    SelfSndSocket = socket(PF_INET,SOCK_DGRAM,0);
    if(SelfSndSocket == INVALID_SOCKET)
    {
        m_MessageList.ReplaceSel("创建 socket 失败!\r\n");
        return;
    }
    m_sockSelfSndAddr.sin_family = AF_INET;
    m_sockSelfSndAddr.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
    m_sockSelfSndAddr.sin_port = htons(8888);
```

```

if(bind(SelfSndSocket,(LPCTSTR)&m_sockSelfSndAddr,
        sizeof(m_sockSelfSndAddr)) == SOCKET_ERROR)
{
    m_MessageList.ReplaceSel("绑定失败!\r\n");
    return;
}
//设置发送参数
m_sockAddrto.sin_family = AF_INET;
m_sockAddrto.sin_addr.S_un.S_addr = inet_addr(sIP);
m_sockAddrto.sin_port = htons(atoi(sPort));
strcpy(msgbuf.msg, (LPCTSTR)msgtoSnd);
msgbuf.i = 0;
//发送
if(sendto(SelfSndSocket, (char *)&msgbuf, sizeof(msgbuf), 0,
          (LPCTSTR)&m_sockAddrto, sizeof(m_sockAddrto)) == SOCKET_ERROR)
{
    m_MessageList.ReplaceSel("发送数据失败!\r\n");
}
//发消息的套接字在用完要及时关闭, 以便其他端系统进程也能随时共享使用它
closesocket(SelfSndSocket);
}

```

这里默认指定发消息的进程所使用的地址为 127.0.0.1:8888, 是应用了第 3 章 UDP Socket 程序的性质。本例采用 SOCK_DGRAM (无连接数据报) 类型的套接字在进程之间发送消息, 发送进程的 IP 必须设置为 127.0.0.1, 端口任意。经过这样设置的进程可以向任何其他进程发送数据, 但接收进程是不能识别出发送进程的地址的。

服务器收到报文后, 首先检查这个用户是否已在其上注册, 若已注册在线, 表示用户这回发来的这个报文是想要注销退出系统; 若用户是新加入的, 则登记新用户信息, 更新用户列表并发送更新后的用户列表给各个终端。

服务器用 EnrolUsr() 方法更新其上保存的用户信息。

EnrolUsr() 方法代码如下:

```

void CP2PTalkerDlg::EnrolUsr(CString strMsgData)
{
    CString username, ipFild1, ipFild2, ipFild3, ipFild4, port;
    int n;
    //从报文中截取昵称字段
    n = strMsgData.Find(",");
    username = strMsgData.Left(n);
    strMsgData = strMsgData.Right(strMsgData.GetLength() - (n + 1));
    //依次截取 IP 和端口各个字段
    n = strMsgData.Find(".");
    ipFild1 = strMsgData.Left(n);
    strMsgData = strMsgData.Right(strMsgData.GetLength() - (n + 1));
    n = strMsgData.Find(".");
    ipFild2 = strMsgData.Left(n);
    strMsgData = strMsgData.Right(strMsgData.GetLength() - (n + 1));
}

```

```

n = strMsgData.Find(".");
ipFild3 = strMsgData.Left(n);
strMsgData = strMsgData.Right(strMsgData.GetLength() - (n + 1));
n = strMsgData.Find(":");
ipFild4 = strMsgData.Left(n);
port = strMsgData.Right(strMsgData.GetLength() - (n + 1));
//下面开始进行用户登记处理
BOOL BePresnc = false;                                //指示这个用户是否已在服务器上注册
for(int i = 0; i < rstUsrCont; i++)
{
    if(CString(rstUsr[i].username) == username)
    {
        BePresnc = true;
        if(rstUsr[i].onlStat)
        {
            //若用户本来就在线，说明这次是要注销退出
            rstUsr[i].onlStat = false;
            onlUsrCont--;
        }
        else
        {
            //若用户上线时改变了 IP，服务器上注册的用户信息也会随之同步更新
            rstUsr[i].ipFild[0] = _ttoi(ipFild1);
            rstUsr[i].ipFild[1] = _ttoi(ipFild2);
            rstUsr[i].ipFild[2] = _ttoi(ipFild3);
            rstUsr[i].ipFild[3] = _ttoi(ipFild4);
            rstUsr[i].port = atoi((LPCTSTR)port);
            rstUsr[i].onlStat = true;
            onlUsrCont++;
        }
    }
}
if(!BePresnc)                                          //用户是新加入的
{
    strcpy(rstUsr[rstUsrCont].username, (LPCTSTR)username);
    rstUsr[rstUsrCont].ipFild[0] = _ttoi(ipFild1);
    rstUsr[rstUsrCont].ipFild[1] = _ttoi(ipFild2);
    rstUsr[rstUsrCont].ipFild[2] = _ttoi(ipFild3);
    rstUsr[rstUsrCont].ipFild[3] = _ttoi(ipFild4);
    rstUsr[rstUsrCont].port = atoi((LPCTSTR)port);
    rstUsr[rstUsrCont].onlStat = true;
    rstUsrCont++;
    onlUsrCont++;
}
//更新用户
m_UserList.EnableWindow(true);
m_UserList.ResetContent();

```

```

m_MessageList.SetWindowTextA("");
OnlUsrInfo = "UPDT_USR;";
for(int i = 0;i < rstUsrCont;i++)
{
    if(rstUsr[i].onlStat)
    {
        //生成更新用户类报文
        OnlUsrInfo += CString(rstUsr[i].username) + "&";
        m_UserList.AddString(CString(rstUsr[i].username));
        CString sIP,sP,onlUsrStateView;
        sIP.Format("%d.%d.%d.%d",rstUsr[i].ipFild[0],rstUsr[i].ipFild[1],
                    rstUsr[i].ipFild[2],rstUsr[i].ipFild[3]);
        sP.Format("%d",rstUsr[i].port);
        onlUsrStateView = CString(rstUsr[i].username) + " -IP:";
        onlUsrStateView += sIP + " 端口:";
        onlUsrStateView += sP + "\r\n";
        //服务器监控显示在线用户状态信息
        m_MessageList.ReplaceSel(onlUsrStateView);
    }
}
for(int i = 0;i < rstUsrCont;i++)
{
    if(rstUsr[i].onlStat)
    {
        CString sIP,sP;
        sIP.Format("%d.%d.%d.%d",rstUsr[i].ipFild[0],rstUsr[i].ipFild[1],
                    rstUsr[i].ipFild[2],rstUsr[i].ipFild[3]);
        sP.Format("%d",rstUsr[i].port);
        //发送更新后的用户列表给各终端
        this->StartSndSocket(OnlUsrInfo,sIP,sP);
    }
}
}

```

更新后的新用户列表由更新用户类报文携带，这类报文以 UPDT_USR 为首部，各终端就是从其中解析出当前最新的在线用户列表的。

终端收到更新通知后，使用 UpdUsr()方法更新在线用户列表。

UpdUsr()方法代码如下：

```

void CP2PTalkerDlg::UpdUsr(CString strMsgData)
{
    CString array[100];
    int b = 0;
    for( int i = 0; i < strMsgData.GetLength(); i++ )
    {
        if(i != (strMsgData.GetLength() - 1))
        {
            if ( strMsgData[i] == '&' )

```

```

        {
            b++;
        }
        else
        {
            array[b] = array[b] + strMsgData[i];
        }
    }
}
m_UserList.ResetContent();
for(int j = 0; j < b + 1; j++)
{
    m_UserList.AddString(array[j]);
}
}

```

用户终端是通过“&”分辨出在线的各个不同用户的。

2) 收发信息的 P2P 过程

用户在文本框中编辑完文本信息后，从界面列表中选择想要联系的用户昵称，再单击“发送”按钮，开始 P2P 的信息收发过程。

“发送”按钮的事件代码如下：

```

void CP2PTalkerDlg::OnSend()
{
    //选择对方用户名
    int nSel;
    nSel = m_UserList.GetCurSel();
    if(nSel == LB_ERR)
    {
        AfxMessageBox("请先从在线用户列表中选择对方网名!");
        return;
    }
    CString UserSel;
    m_UserList.GetText(nSel, UserSel);
    UpdateData();
    if(UserSel == nickname)
    {
        AfxMessageBox("不能向自身发消息!");
        return;
    }
    //生成请求 P2P 服务类的报文
    BYTE nFild[4];
    CString sIP;
    LocalIP.GetAddress(nFild[0], nFild[1], nFild[2], nFild[3]);
    sIP.Format("%d.%d.%d.%d", nFild[0], nFild[1], nFild[2], nFild[3]);
    PtoPsrvtReqst = "REQ_ADDR;";
    PtoPsrvtReqst += UserSel + "," + sIP;
    PtoPsrvtReqst += ":" + locaPort;
}

```

```

//获取 P2P 服务进程的 IP 和端口
SevrIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
//发送报文
this->StartSndSocket(PtoPsrvReqst,sIP,sevrPort);
}

```

由于目前通信发起方用户还不知道对方用户终端进程的 IP 和端口,故需要向服务器“询问”,这是通过向服务器发送首部为 REQ_ADDR 的请求 P2P 服务类报文实现的。服务器收到请求后,执行 PtoPsrvProvdr()方法,并向用户返回 P2P 服务响应报文。

PtoPsrvProvdr()方法代码如下:

```

void CP2PTalkerDlg::PtoPsrvProvdr(CString strMsgData)
{
    CString username,ip,port,perUsrAddr;
    int n;
    //从报文中截取昵称字段
    n = strMsgData.Find(",");
    username = strMsgData.Left(n);
    strMsgData = strMsgData.Right(strMsgData.GetLength() - (n + 1));
    //在本地服务器上搜索用户请求的 IP 地址
    for(int i = 0;i < rstUsrCont;i++)
    {
        if(CString(rstUsr[i].username) == username)
        {
            //找到匹配的用户之后,组装成 P2P 服务响应报文
            CString perIP,perPort;
            perIP.Format("%d.%d.%d.%d",rstUsr[i].ipFild[0],rstUsr [i].ipFild[1],
                        rstUsr[i].ipFild[2],rstUsr [i].ipFild[3]);
            perPort.Format("%d",rstUsr[i].port);
            perUsrAddr = username + ",";
            perUsrAddr += perIP + ".";
            perUsrAddr += perPort;
            PtoPsrvRspse = "RSP_ADDR;" + perUsrAddr;
            break;
        }
    }
    //截取请求发送者的 IP 和端口
    n = strMsgData.Find(":");
    ip = strMsgData.Left(n);
    port = strMsgData.Right(strMsgData.GetLength() - (n + 1));
    //向用户返回响应报文
    this->StartSndSocket(PtoPsrvRspse,ip,port);
}

```

因为服务器上保存着所有注册用户的信息,包括它们各自终端进程的地址(IP+端口)信息,于是服务器搜索用户请求的 IP 地址,找到匹配之后,组装成 P2P 服务响应报文返回用户终端。用户终端收到 P2P 服务响应报文,执行 SendMsgtoPer(),发起 P2P 通信。

SendMsgtoPer()方法代码如下:

```

void CP2PtalkerDlg::SendMsgtoPer(Cstring strMsgData)
{
    Cstring ip,port;
    int n;
    //从报文中解析出对方地址信息
    n = strMsgData.Find(",");
    strMsgData = strMsgData.Right(strMsgData.GetLength() - (n + 1));
    //截取对方的 IP 和端口
    n = strMsgData.Find(":");
    ip = strMsgData.Left(n);
    port = strMsgData.Right(strMsgData.GetLength() - (n + 1));
    //组装出携带聊天内容的报文
    UpdateData();
    if(strMessage == "")
    {
        AfxMessageBox("不能发送空消息!");
        m_EditWords.SetFocus();
        return;
    }
    Ctime time = Ctime::GetCurrentTime();
    Cstring t = time.Format("%H:%M:%S");
    MsgTalktoPeer = "SND_MSG;" + nickname + ",";
    MsgTalktoPeer += t + "\r\n" + strMessage + "\r\n";
    //向对方发送聊天信息
    this->StartSndSocket(MsgTalktoPeer,ip,port);
    MsgTalktoPeer = "\r\n" + nickname + " " + t + "\r\n" + strMessage + "\r\n";
    m_MessageList.ReplaceSel(MsgTalktoPeer);
    strMessage = "";
    UpdateData(FALSE);
    m_EditWords.SetFocus();
}

```

从用户单击“发送”按钮开始，直到这个时候才真正向对方用户发出信息，而在这之前的一系列过程都是与服务器在交互：

终端向服务器发出请求 P2P 服务类消息→服务器执行 PtoPsrVProvdr()方法→服务器向用户返回 P2P 服务响应类消息。

以上一系列动作都是由终端与服务器密切配合并自动完成的，整个过程离不开服务器的管理和协调，但这一切对于最终用户来说都是“透明”的。在用户看来，自己的终端仿佛是直接与对方用户通信，收发信息十分便捷随心！

对方收到信息后使用 ShowMsgfrmPer()方法在界面屏幕上显示信息内容，代码如下：

```

void CP2PTalkerDlg::ShowMsgfrmPer(CString strMsgData)
{
    CString username,MsgfrmPer,str;
    int n;
    //从报文中取出昵称和聊天信息字段
    n = strMsgData.Find(",");
    username = strMsgData.Left(n);           //解析出报文附带的对方昵称信息

```



```

MsgfrmPer = strMsgData.Right(strMsgData.GetLength() - (n + 1));
//聊天内容显示
str = "\r\n" + usrname + "    " + MsgfrmPer;
m_MessageList.ReplaceSel(str);
}

```

这里还有一点需要指出：由于 UDP 程序的性质，对方用户终端在接收到别人发给它的信息时并不能马上判断出究竟是谁发的（“看不到”发送者的 IP），这就需要发送者通过其他途径将自己的身份“告知”对方。本例采用的是在发送信息报文的数据部分开辟一个“用户名”字段，发送者在发送信息前预先将自己的昵称写入这个字段，接收方在收到后只要解析出这一字段的内容，就知道是谁发给它信息了——这也是很多网络协议常用的设计思路，即在通信报文中携带附加的信息。

由上述过程也可以看到：在 P2P 方式中，通信双方交流中收发的信息不再经由服务器转发，与 4.2 节 C/S 结构的聊天室相比，这种方式更高效、更节省网络带宽资源，同时也很好地保护了用户隐私。现在流行的 IM 软件绝大多数实际上就是采用的这种 P2P 方式。

在 P2P 通信中，参与各方的地位都是平等的。各通信方可以自主启动本机进程，也可以随时关停它，只需单击“停止”按钮即可。

“停止”按钮的事件过程代码如下：

```

void CP2PTalkerDlg::OnStop()
{
    //当程序停止运行时，把 SOCKET 清空
    m_MessageList.ReplaceSel("\r\n 正在关闭 Socket...\r\n");
    closesocket(SelfRcvSocket);
    WSACleanup();
    m_MessageList.ReplaceSel("本机进程停止运行!\r\n");
    this->SetWindowTextA("ptopTalker");
    //界面
    m_Start.EnableWindow(true);
    m_Stop.EnableWindow(false);
    LocalIP.EnableWindow(true);
    m_LcaPrt.EnableWindow(true);
    m_exit.EnableWindow(true);
    m_Login.EnableWindow(false);
    m_nicnam.EnableWindow(false);
    SevrIP.EnableWindow(false);
    m_SvrPrt.EnableWindow(false);
    LocalIP.SetFocus();
}

```

到此为止，这个 P2P 架构的聊天工具就开发完成了。

其实，通过以上这个例子的实践，大家已经设计并亲手实现了一个简单的网络协议——正是先前 4.3.2 节所定义的那个 P2P 通信规约。仔细想一下，难道不是吗？后面编写程序时不全都是在这个规约的指导下有条不紊地进行的吗？

用户登录（注销）时发送的注册（销）类消息冠以首部 RGST_ADDR 就是在规约中规定的；在线更新类消息首部 UPDT_USR、请求 P2P 服务的消息首部 REQ_ADDR……这些都在那个规约里有明确规定。

再来看各消息数据部分的字段划分,以请求 P2P 服务为例,消息的数据部分为“用户名(对方), 127.*.*.*: 端口号(己方)”——规约中就是这样定义的。还有,用户终端通过“&”分辨出在线的各个不同用户,也是由于规约中规定:在线更新类消息数据部分的用户名之间以“&”分隔。

综上所述,正是由于有了这个规约,各用户终端之间及终端与服务器之间才能够顺利地实现交互协作,共同实现这个 P2P 聊天系统的完整功能。所谓的“网络协议”正是这样一些事先约定好的规则,明确规定了所交换的数据的格式,以及在一定的条件下应当发生什么事。

实际使用的商品化软件的协议一般来说都是极其复杂的,因为作为一个可以上市的产品,必须考虑到应用中方方面面的因素,如对用户身份和权限进行验证,处理通信中各种可能的突发情形以确保软件的可靠性,此外还有负载均衡、稳定性、可伸缩性、可扩展性等诸多指标。这时的协议就不是一两句话能约定清楚的,必须形成规范的文档,实用的协议文档都很厚,如以太网 IEEE 802.3 系列的文档就厚达 1 500 页!

可见,网络软件工作的基础是 Socket,而 Socket 进程之间通信所依据的准则就是网络协议,于是,网络编程开发就总结为以下这条公式:

网络软件 = Socket 程序 + 网络协议

4.3.4 P2P 方式通信的特性

今天的互联网应用绝大多数都是以 P2P 方式架构的,接下来通过运行这个程序,让读者体会一下 P2P 方式与传统的 C/S 方式有何不同,究竟有哪些优秀的特性。

1. 试验

1) 点对点通信

同时开启软件的三个程序实例,任选其中一个作为服务器,设置本机进程 IP 为 127.102.119.7:2255 (也可以指定其他地址,只要合法就行),启动服务进程。

然后用另两个程序实例作为用户终端,分别设置它们的 IP 为 127.66.8.12:34 和 127.45.36.2:23,启动各自的本地进程。分别以昵称“闲云孤鹤”和“小家伙”登录服务器,如图 4.17 所示。

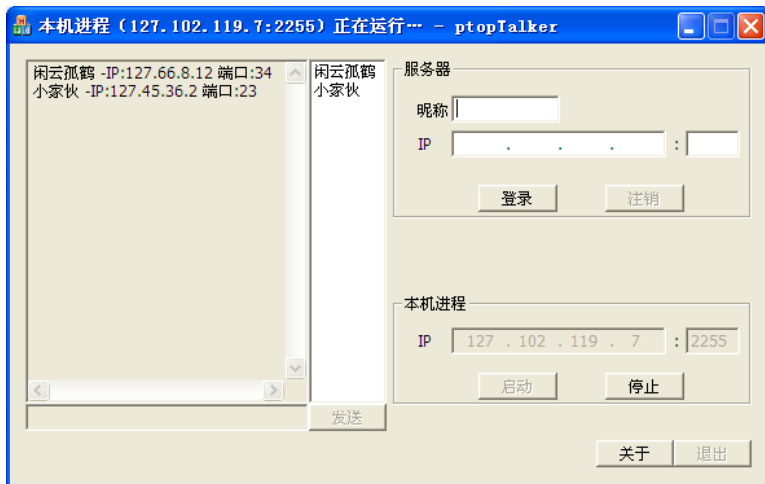


图 4.17 启动并登录服务器

成功登录后就可以互发信息聊天了,如图 4.18 所示。

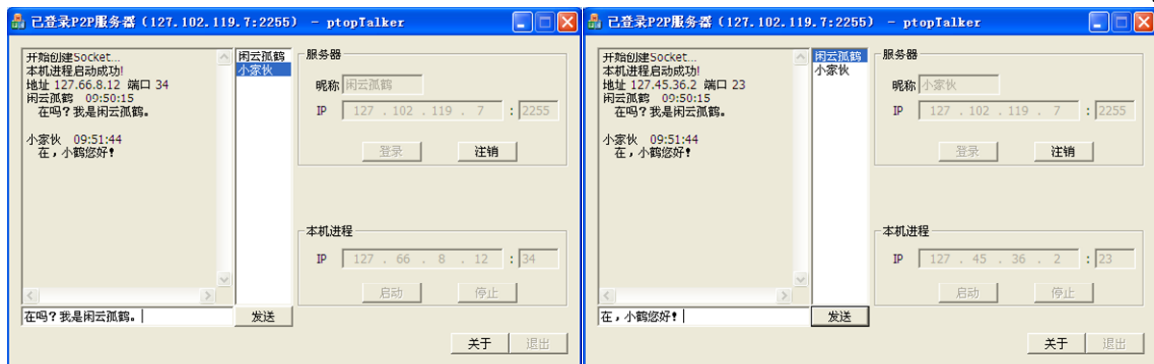


图 4.18 互发信息进行点对点聊天

当然，这里所用的 IP 和用户昵称都只是举例，读者也可以指定其他任何合法的地址，取自己喜欢的昵称并开启更多终端进行试验。

2) 服务器自我登录、参与聊天

目前两个用户（昵称为“闲云孤鹤”和“小家伙”）已经能够通过服务器进行 P2P 的通信了。这时，若终端被用作服务器的用户也想参与他们俩的聊天，如何做到呢？

在服务器界面上设置登录地址为 127.102.119.7:2255（服务器自身的地址），昵称取为“小月”，单击“登录”按钮，如图 4.19 所示。

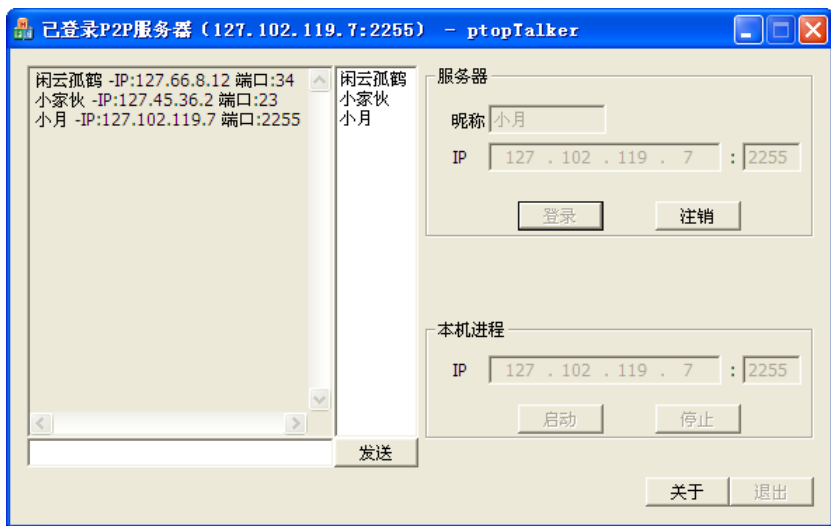


图 4.19 新成员“小月”加入

在接下来的测试中看到：新加入的成员“小月”也可以与在线的那两位用户正常聊天，如图 4.20 所示。也就是说，我们根本看不出来他究竟是从普通用户终端还是服务器登录这个系统的。

3) 切换服务器

前面已经指出，系统中的任何一个用户终端都可以用作服务器，下面就来换一个服务器试试。将其中两个用户“小月”和“小家伙”都从服务器上退出（单击“注销”按钮，如图 4.21 所示）。

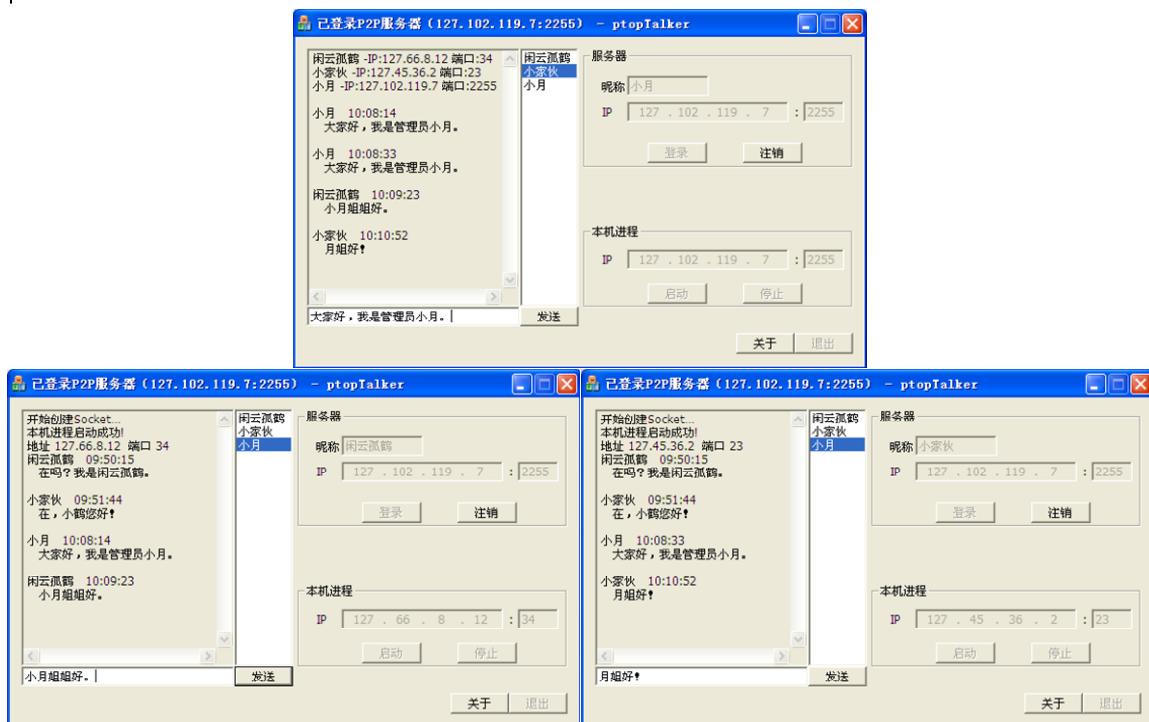


图 4.20 服务器作为终端参与聊天

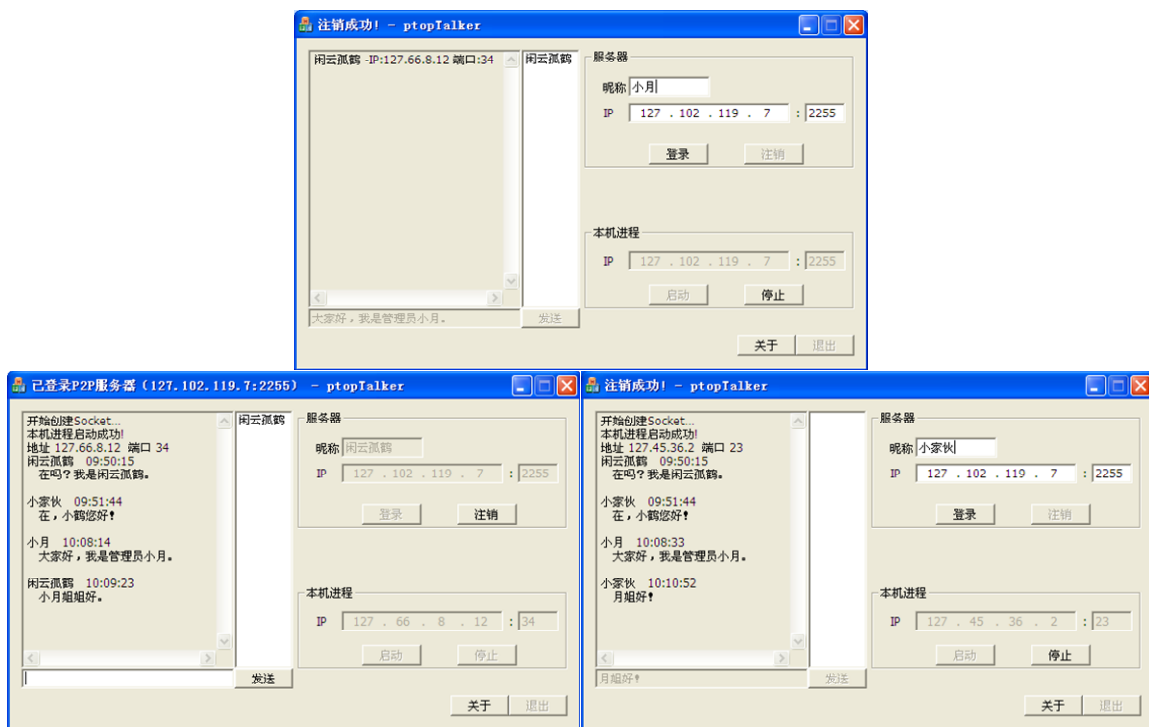


图 4.21 原有用户注销

用他们的终端转而登录地址 127.66.8.12:34 (如图 4.22 所示的“闲云孤鹤”的用户端)的终端。

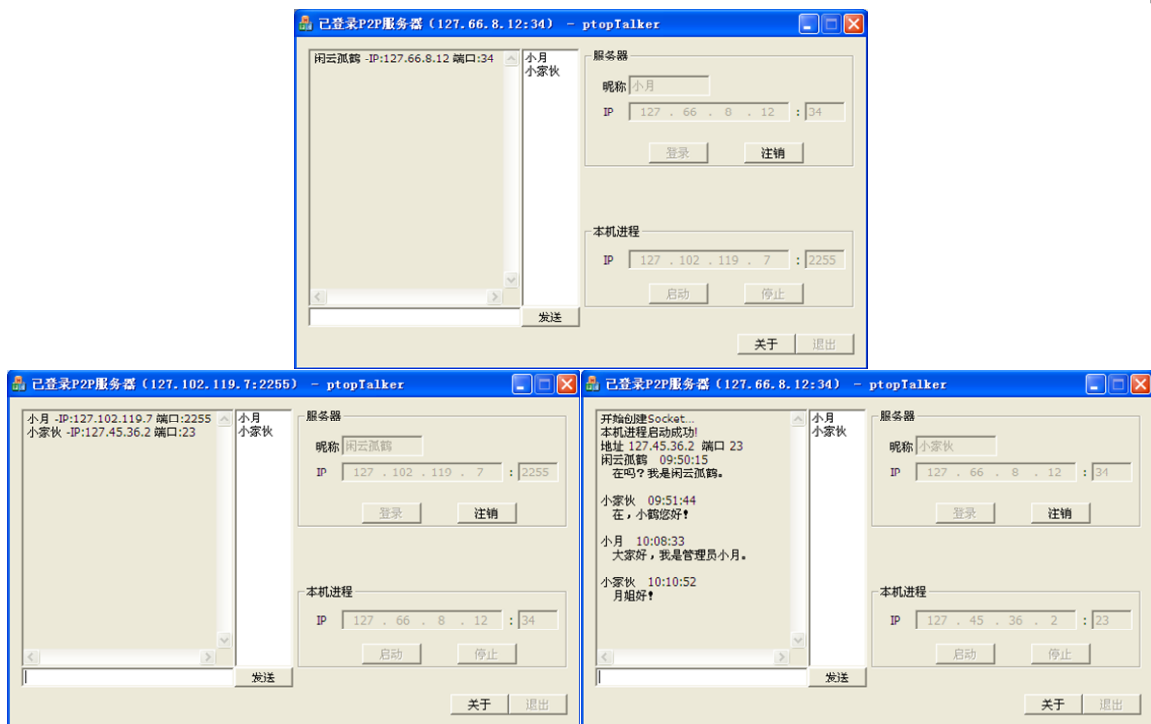


图 4.22 从新的服务器登录

两者都顺利地登录成功了！读者可以试着做一下，就会发现他们同样也是可以互发信息聊天的，这与采用谁的终端做服务器无关（如图 4.23 所示）。

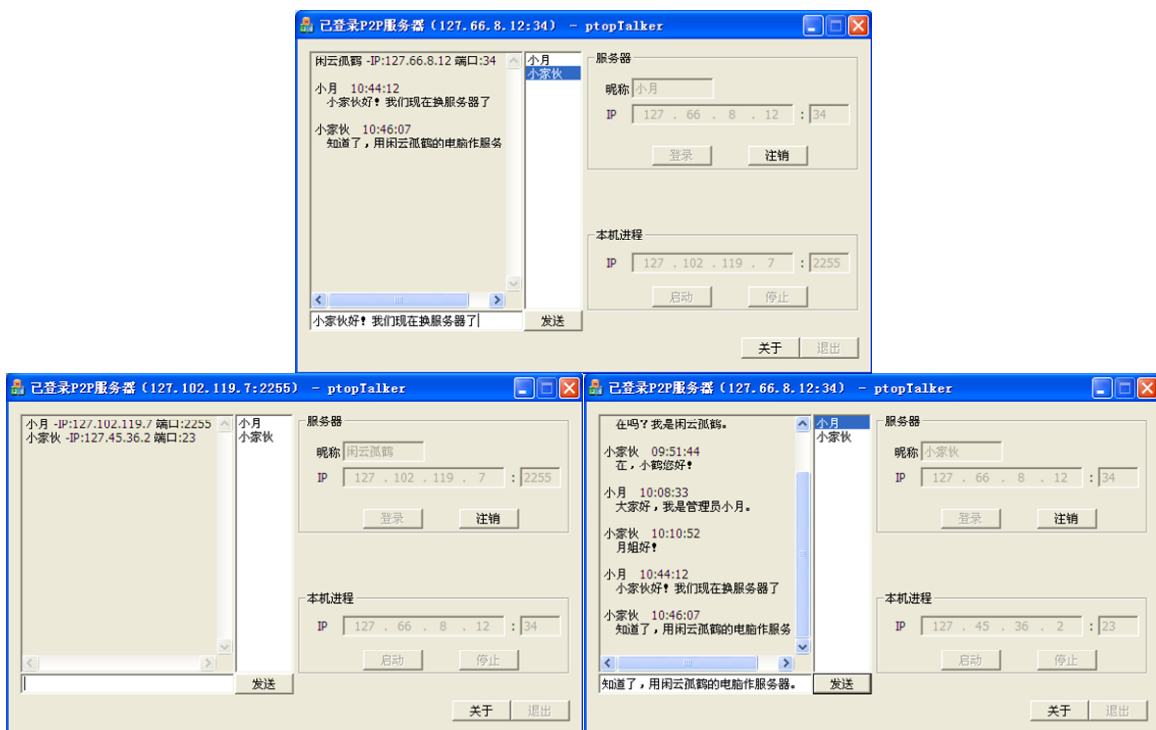


图 4.23 通过新的服务器照常通信

2. 结论

由上述这几组试验可以得出这样的结论：

(1) 每一个用户都可以用自己的终端（不管是普通的终端还是被选作服务器的）程序登录服务器（也包括服务器自己登录自己），我们无法仅凭软件运行的表面现象判断用户究竟是从普通终端还是从服务器登录系统的。

(2) 任何一个普通的用户终端都可以作为服务器使用，在系统运行的中途，参与各方的用户可以根据需要随时更换服务器。

简言之，每个用户终端都同时既具有客户端又具有服务器的功能——这也是一切 P2P 体系结构的软件终端所具备的根本特征。

4.4 原型程序与 IM 产品

本书是基础教程，书中给出的例子仅是“原型程序”，它们只具备最基本的原理性功能。本章的最后，将这些原型与实际使用的 IM 产品（如腾讯 QQ）做一个比较，让有志于专业编程的读者了解何谓真正“产品化”的软件。

4.4.1 本程序与腾讯 QQ 的类比

如图 4.24 所示是本章 4.3 节聊天工具程序运行中的典型界面，图 4.25 则是一个 QQ 群的界面。

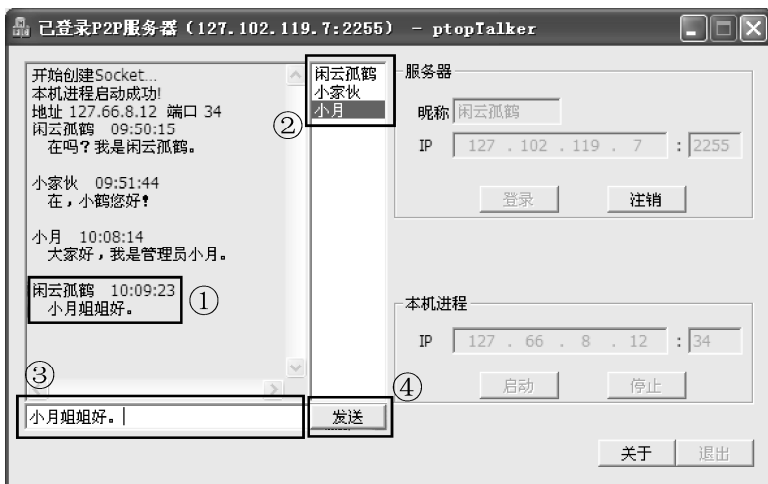


图 4.24 本章聊天程序界面

比较一下两者的基本功能，发现有很多相似之处，归纳为如下几点（已在图 4.24 和图 4.25 中标出）。

(1) 主界面上都有一个显示聊天内容的只读区域，并且两者显示的聊天记录格式也都大致相同（网名+时间+信息内容），如两图中①标出。

(2) 两者都有一个用户列表用于显示用户的在线状态，如两图中②标出，所不同的是我们的聊天客户端只能列出当前在线的用户名单，而 QQ 能将所有的群成员名全部列出来，不仅是网名的文本，还能附带显示该成员的图标头像，并以头像的彩色/黑白状态来指示该用户是否在线。

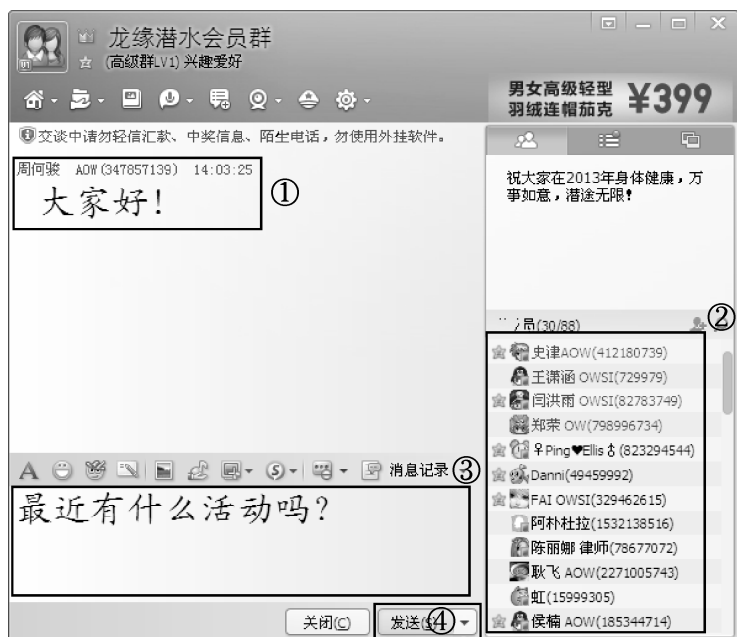


图 4.25 QQ 群界面

(3) 两个软件都有用于编辑所发信息的文本框（如两图中③标出）和“发送”按钮（如两图中④标出）。相比而言 QQ 功能强大之处在于：不仅可以发送文字信息，在文本框的上方工具栏内还有一行彩色的图标按钮，可以发送包括表情、图片、音乐、屏幕截图在内的各种类型的信息，如图 4.26 所示。



图 4.26 QQ 工具栏

由上面的几点比较可以看出，本章的聊天程序已经初步具备了 QQ 应用的基本雏形，只要在此基础上添加新功能，就会逐步完善，最终演变成实用的即时通信软件也不是没有可能！

4.4.2 IM 产品的增强功能与技术

但是，作为 IM 产品的 QQ 还有着诸多增强的功能：

- (1) 显示全部用户名的列表，并以用户头像图标的色彩来指示在线状态。
 - (2) 支持文件传输、QQ 邮箱。
 - (3) 聊天中不仅可以发送文字信息，还可以发送图像、音乐、表情等复杂多样的信息类型。
 - (4) 支持语音聊天、视频聊天。
 - (5) 可以玩 QQ 游戏、开 QQ 空间、购买 Q 币、养 QQ 宠物等。
 - (6) 拥有华丽的外表和可以更换的皮肤。
-

对于上述 (1)，要能显示全部用户名，必须在服务器上专门存储全部注册用户的信息，产品软件一般都在后台有数据库服务器支持。例如，腾讯公司一定会有专门的数据库服务器（集群），将用户的注册信息存储在数据（仓）库中。每当用户登录 QQ 时，都要进行身份验证，到数据库中查找用户名、核对密码；登录后，服务器程序会将数据库中早已存储的群中所有成

员的信息（用户头像图标、用户名、QQ 号、在线状态等）返回给客户端，于是用户就看到了其他用户的在线状态。

对于上述（2），文件传输（FTP）和邮箱（E-mail）都是有别于网络聊天的应用类型，将在本书第 6 章、第 7 章介绍。

对于上述（3），要使聊天软件能够收发非字符文本的复杂信息类型，必须有一种通用的机制能直接收发结构类型的对象，这种机制在第 2 章已经介绍过，就是 CSocket 类与 CArchive、CSocketFile 类一起配合使用的 Socket 通信机制。CArchive 和 CSocketFile 类封装了对接收到的数据进行序列化处理的过程，使得接收任何网络数据（尤其对较复杂类型的数据）都如同读取本地文件一样简单。读者如果深入钻研这项功能的实现，就会体会到这种数据收发机制的巨大优越性了，作为基础教程，本书仅提及到此，不再深入。而非文本（如图像）类型信息的处理则要用到图形图像编程技术了。

至于（4）、（5）、（6）……已经完全超出了网络编程的范畴，其中（4）涉及多媒体编程；（5）是 Web 开发实现的应用，还要用到游戏编程；（6）则是 GUI 美工设计，是界面程序员的工作。

另外，QQ 由于是应用在互联网环境下，在实现中肯定还运用了 NAT 穿透技术，使得网络边缘用户计算机上的 P2P 客户端程序能穿越互联网核心进行通信，如图 4.27 所示，有关 NAT 穿透技术编程，由于涉及内容比较深入，本书暂不作介绍。

可见，QQ 这样的产品化 IM 软件其实是一个融合了众多实用功能和技术的网络应用系统，需要不同专业领域的程序员协作才能完成。

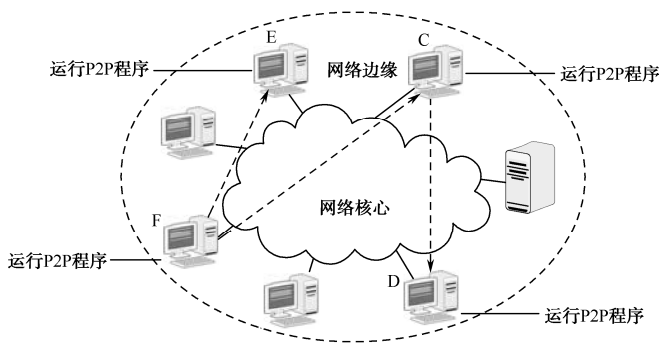


图 4.27 穿透网络核心的 P2P 即时通信

4.4.3 即时通信发展新趋势

从最早的聊天室、ICQ 发展到现在，我们的日常生活已经离不开即时通信工具了，目前主流的即时通信软件有 QQ、百度 HI、Skype、Gtalk、FreeEIM、飞鸽传书等。这里向大家简要介绍一下即时通信应用发展的新趋势。

1. 应用功能集成化

伴随着 Internet 时代人们生活的精彩多样化，网络应用的种类越来越多，特别是近几年的迅速发展，即时通信的功能也随之日益丰富，逐渐集成了电子邮件、博客、音乐、电视、游戏和搜索等多种功能。即时通信不再是一个单纯的聊天工具，它已经发展成集交流、资讯、娱乐、搜索、电子商务、办公协作和企业客户服务等为一体的综合化信息平台，是一种终端联网即时通信网络的服务。

2. 客户端移动化

近年来,通信领域 3G 技术的火热推动了即时通信客户端软件由 PC 向手机客户端转移,以中国移动**飞信**为代表,飞信是中国移动推出的一项业务,可以实现即时消息、短信、语音、GPRS 等多种通信方式,保证用户永不离线。实现无缝链接的多端信息接收,让用户随时随地都可与好友保持畅快有效的沟通。

3. 网页即时通信成为新潮流

基于网页的即时通信工具如 WebQQ、阿里旺旺网页版,是对传统即时通信服务的一种改革。网页版即时通信产品的优势有以下几点。

(1) 无须下载、安装客户端软件。用户不再需要经常为更换通信软件的版本而不停下载、安装新的客户端,省去了操作,同时也节约了计算机的空间。

(2) 聊天记录无论在哪台计算机上都可以查看。传统的即时通信软件一般将聊天记录保存在客户端的计算机上,用户换了计算机再使用时,往往就查看不到聊天记录了。而网页版的即时通信软件是将聊天记录保存在 Web 服务器中的,因此,无论在哪台计算机上使用都可以看到以前的聊天记录。

(3) 可以和网站应用无缝结合,进一步提高用户之间的交流互动。如淘宝网和阿里旺旺的结合,使得每一位买家都可以随时向卖家咨询宝贝的情况。

HTTP 编程与万维网开发

Internet 之所以应用如此广泛，主要归功于万维网（Word Wide Web，WWW 或 3W）。自从 1990 年英国计算机科学家蒂姆·伯纳斯·李（Tim Berners-Lee）发明 Web 浏览器后，Internet 的发展就迈进了万维网时代，以至于今日很多人误以为万维网就是 Internet。实际上，万维网是基于 HTTP 协议工作的一种网络应用。

5.1 HTTP 原理

万维网之所以能取得如此巨大的成功，是因为简单易用，人们只要用鼠标轻轻点击就可以在 Internet 的世界里遨游。

是什么使万维网使用起来如此简单呢？

这主要有两个因素：一是组成万维网的超链接文档（HTML 文档）；二是超链接文档的传输机制（HTTP）。如何制作 HTML 文档（即网页）不属于本书的内容，读者可参阅网页制作方面的书。本节主要介绍 HTTP 的工作原理。

5.1.1 万维网的工作过程

先简单介绍一下万维网是如何工作的。在 Internet 中工作的主机，当要访问万维网的某个网页时，大致都要经过以下步骤（如图 5.1 所示）。

（1）用户首先要确定网页文件所在的 URL（统一资源定位符），由 URL 唯一确定用户要访问的文件在 Internet 上的位置，如设为 `http://www.njnu.edu.cn/home.html`。

（2）浏览器向 DNS（域名服务器）发出请求，要求把域名 `www.njnu.edu.cn` 转化为它所对应的 IP 地址。

（3）DNS 进行查询后，向浏览器发出应答。如域名 `www.njnu.edu.cn` 对应的 IP 地址为 `191.168.103.9`。

（4）在查询得到网页所在的服务器 IP 后，就进入 HTTP 的工作阶段。浏览器向 IP 地址为 `191.168.103.9` 的主机发出与端口 80 建立一条 TCP 连接的请求。80 端口是服务器提供 Web 服务的默认端口。

（5）连接建立成功后，浏览器发出一条请求传输网页的 HTTP 命令，格式为 `GET/home.html`。

（6）当域名为 `www.njnu.edu.cn` 的服务器收到请求后，向浏览器发送 `home.html` 文件。

- (7) 文件发送完成, 由服务器主动关闭 TCP 连接, 至此 HTTP 的工作过程也结束了。
- (8) 浏览器显示收到的网页文件 home.html。
- (9) 若 home.html 文件中包含图片, 则还要与服务器再次建立一个 TCP 连接以下载图片。

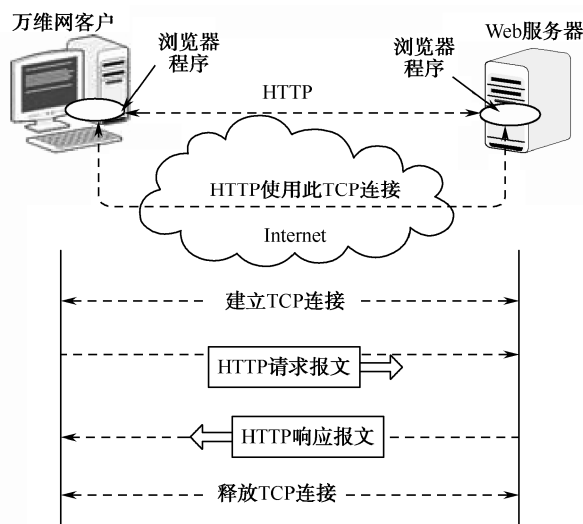


图 5.1 HTTP 工作过程

上述步骤中, 对客户 (浏览器) 来说最关键的问题是如何把 HTML 文档下载到本地主机, 这个任务是由 HTTP 完成的, 也就是说, HTTP 是 Web 服务的基础。用户在使用浏览器时, 这个过程对用户是透明的。

5.1.2 超文本传输协议

HTTP 的中文全称是超文本传输协议 (Hypertext Transfer Protocol), 它于 1990 年提出, 经过 20 多年的使用, 不断得到完善和扩展。目前在万维网上使用的主要是 HTTP/1.0 和 HTTP/1.1, 本节介绍的是 HTTP/1.1。

HTTP 工作于客户端—服务器模式, 浏览器是客户端, 接收连接并对请求返回信息的应用程序是 Web 服务器。Internet 中的 Web 服务器在 TCP 端口 80 监听客户端的请求。在实际工作时常用的浏览器相当于一个用户代理 (User Agent), 用户要求完成的各种操作均由它向 Web 服务器提出, 并处理由服务器返回给客户端的响应。当浏览器要从某 Web 服务器下载网页文件时, 首先根据服务器的 IP, 向其 TCP 端口 80 发出建立连接的请求, 在连接建立后, 客户端发送请求服务的方法, 服务器进行应答。

1. 客户端 HTTP 请求

HTTP 请求可以由多行组成, 但最后一行必须是空行。HTTP/1.1 最常用的请求格式如下:

```
请求方法 URL HTTP 版本号
请求头信息
请求数据
<一个空行, 这是请求的结束行>
```

请求可用的方法见表 5.1, 每种方法规定了一种客户端与服务器联系的内容。

表 5.1 HTTP 请求方法

方 法	描 述
GET	返回 URL 所指的文档, 一般情况下用于请求下载 Web 网页。例如, GET http://www.njnu.edu.cn/home.html HTTP/1.1
HEAD	请求文档头, 它类似于 GET 请求, 只是 Web 服务器返回指定文档的首部信息。该请求通常被用来测试超文本链接的正确性、可访问性和最近是否进行了修改
POST	它与 GET 方法相反, 请求服务器接收指定的文档, 但它不是替换已有的文档, 只是将新数据附加到它的后面。一般可用来自新闻组发送一条消息, 或发送能由交互用户填写的表格等
PUT	它与 GET 方法相反, 用从客户端传送的数据取代指定文档中的内容, 使客户端可以向远程 Web 服务器传送网页等文件
DELETE	请求服务器删除指定的页面
OPTIONS	允许客户端查看服务器的性能
TRACE	用于测试允许客户端查看的消息回收过程

用户要访问 Web 中的某一个网页, 一定要告诉浏览器三个问题:

- (1) 该网页怎样下载到本地主机, 其实就是下载网页使用什么协议的问题。
- (2) 该网页在 Web 中的什么地方, 是要给出这个网页所在主机的地址 (或域名)。
- (3) 该网页的名称是什么, 则是要给出网页文件的名称。

为此人们定义了 URL (Uniform Resource Locator), 它可以很方便地描述以上三个问题的答案, 格式为:

URL=协议名称+主机名 (或 IP 地址)+目录与文件名

例如:

```
http://www.sogou.cn/index.html
ftp://ftp.njupt.edu.cn/pub/app/visual_studio2008.iso
http://202.103.119.7/mail/free/scan
```

“HTTP 版本号”常为 HTTP/1.0 或 HTTP/1.1。“请求头信息”是可选项, 用于向服务器提供客户端的其他信息, 具体见表 5.2。

表 5.2 HTTP 请求头信息

请求头信息	描 述
Accept	客户端接收的数据类型。例如: Accept: text/html, 表示客户端可接收 HTML 类型的文本
User Agent	客户端软件类型
Authorization	认证消息, 包括用户名和口令
Referer	用户获取的 Web 页面

客户端采用如 POST 的请求时, 由于要向服务器发送数据, 因此有“请求数据”这一项。

例如, 用户输入如下的请求:

```
GET      http:// www.njnu.edu.cn/home.html      HTTP/1.1
Accept: text/plain
Accept: text/html
User-Agent: SelfBrowser/1.0(WinNT)
<一个空行, 这是请求的结束行>
```

说明浏览器使用 GET 方法请求下载 www.njnu.edu.cn 网站的 home.html 网页，并声明只能接收纯文本和 HTML 数据的文件，客户端使用的是 SelfBrowser/1.0（本书开发的）浏览器。

2. 服务器 HTTP 应答

服务器在处理完客户端的请求之后，要向客户端发送响应消息。HTTP/1.1 的响应消息格式如下：

状态行
响应头
响应数据

服务器程序响应的第一行叫状态行，以 HTTP 版本号开始，后面跟着 3 位数字，表示响应代码，如“HTTP/1.1 200 OK”。

在表 5.3 中详细列出了这些响应代码的含义。

表 5.3 HTTP 响应代码的含义

响 应 代 码	说 明
成功响应	
200	OK，请求成功
201	OK，建立新的资源（POST 命令）
202	请求被接收，但处理未完成
204	OK，但没有内容返回
重定向，需要用户代理执行更多的动作	
301	所请求的资源已被指派为新的固定 URL
302	所请求的资源临时位于另外的 URL
304	文档没有修改（条件 GET）
客户差错	
400	错误的请求
401	未被授权；该请求要求用户认证
402	不明原因的禁止
404	没有找到
服务器差错	
500	内部服务器差错
501	没有实现
502	错误的网关；网关或上游服务器的无效响应
504	服务暂时失效

“响应头”是服务器向客户端提供请求文档信息或服务器的状态信息，见表 5.4。“响应数据”表示若客户端的请求包含数据，则数据放在响应头之后，并且数据与响应头之间用一行空行分隔。数据传输完成时，由服务器主动关闭连接。如果没有要传输的数据，则服务器直接关闭连接。

表 5.4 HTTP 响应头

响 应 头	说 明
Server	Web 服务器程序的信息
Date	当前服务器的日期和时间

续表

响 应 头	说 明
Last-Modified	请求文档最近一次修改的时间
Expires	请求文档过期时间
Content-length	数据长度 (字节)
Content-type	数据 MIME 类型
WWW-authenticate	用于通知客户端需要的认证信息 (如用户名、口令等)

下面看一个完整的 HTTP 响应的例子，见表 5.5。

表 5.5 HTTP 响应实例

服务器响应信息	说 明
HTTP/1.1 200 OK	服务器返回的状态行给出了版本号、响应代码 200 和响应短语 “OK”
Date Thursday,02- dec -10 11:45:58 GMT	Date 给出服务器当前的时间和日期，通常是格林尼治时间
Server httpSrver/1.0	服务器程序的类型和版本号分别是 httpSrver 和 1.0
MIME-version 1.0	HTTP 也是用 MIME 来说明文件有关信息的，使用的是 MIME 版本 1.0
Content-type:text/html	数据类型是文本型，子类型是 HTML 型的文本
Last-Modified:Thursday,01.dec.10.19:18:23. GMT	Last-Modified 指出了最后一次修改数据的时间
Content-Length:925	Content-Length 指出文件的长度
<HTML>	最后一个响应的后面，服务器程序紧跟着发送了一个空行
<HEAD>	下面是 925 字节的 HTML 文档。这是文档的第 1 行
...	这是文档的第 2 行
	为节省篇幅，下面的数据全部省略

该例子中，显然是由客户端通过浏览器提出要下载一个网页的请求，使用的是 GET 方法。

5.1.3 统一资源定位符 URL

统一资源定位符 (Uniform Resource Locator, URL) 是用来表示从 Internet 上得到的资源位置和访问这些资源的方法的。URL 给资源的位置提供一种抽象的识别方法，并用这种方法给资源定位。只要能够对资源定位，系统就可以对资源进行各种操作，如存取、更新、替换和查找其属性。这里所说的“资源”是指在 Internet 上可被访问的任何对象，包括文件目录、文件、文档、图像、声音等。URL 相当于一个文件名在网络范围的扩展，因此它是与 Internet 相连的机器上的任何可访问对象的一个指针。由于访问不同对象所使用的协议不同，所以 URL 还指出读取某个对象时所使用的协议。URL 的一般形式由以下四部分组成：

<协议>: //<主机>: <端口>/<路径>

第一部分是最左边的<协议>，它指出使用什么协议来获取该万维网文档。现在最常用的协议就是 HTTP (超文本传输协议)，其次是 FTP (文件传输协议)。<协议>后面是规定必须写上的格式 “://”，不能省略。

第二部分<主机>，它指出这个万维网文档在哪个主机上。此处的<主机>应该是指该主机在 Internet 上的域名。

再后面是第三和第四部分<端口>和<路径>，有时可省略。

在使用浏览器时，输入地址栏的网址其实就是网页的 URL，也是用得最多的一种 URL。对于万维网的访问要使用 HTTP 协议。

HTTP 的 URL 的一般形式是：

```
http://<主机>:<端口>/<路径>
```

HTTP 的默认端口号是 80，通常可省略。若再省略文件的<路径>项，则 URL 就指到 Internet 上的某个主页（Home Page）。

主页是个很重要的概念，它可以是以下几种情况之一：

（1）一个 Web 服务器的最高级别的页面。

（2）某一个组织或部门的一个定制的页面或目录。从这样的页面可链接到 Internet 上与本组织或部门有关的其他站点。

（3）由某个人自己设计的描述他本人情况的 Web 页。

例如，要查有关南京师范大学教务处的信息，就可以先进入南京师范大学教务处主页，其 URL 为南京师范大学教务处网址：

```
"http://jwc.njnu.edu.cn/index.aspx"
```

这里省略了默认端口号 80。从这个主页入手，通过其上不同的链接就可以访问南京师范大学教务处各下属分支部门和主题板块的网页，比如，更复杂一些的指向子层次结构的从属页面。例如：<http://jwc.njnu.edu.cn/Organization/index.aspx> 是南京师范大学教务处的“组织机构”页面的 URL。

用户使用 URL 不仅能够访问万维网页面，而且还能通过 URL 调用其他因特网应用（如 FTP）。更重要的是，用户在调用这些应用时，只使用一个程序（浏览器），这显然是非常方便的。

5.2 浏览器开发

浏览器（Browser）是万维网的客户端浏览程序，可向 Web 服务器发送各种请求，并对从服务器返回的超文本信息和各种多媒体数据格式进行解释、显示和播放。PC 上常见的浏览器有 Internet Explorer、Mozilla 的 Firefox、360 安全浏览器、腾讯 TT 浏览器、搜狗浏览器等，是最为通用的上网客户端。

5.2.1 MFC 对浏览器开发的支持

浏览器与服务器的交互遵循 HTTP 协议，本书第 4 章 4.3.3 节的最后提出过一个适用于网络编程开发的通行公式：

网络软件=Socket 程序+网络协议

根据这个公式，自然可得出浏览器类软件的构成为：

浏览器=Socket 程序+HTTP 协议

但 MFC 提供了特别的支持，使得即使不懂 HTTP 协议的人也能轻而易举地迅速开发出一个浏览器来！

1. 文档/视图结构

本书之前各章所编写的都是对话框类型的程序，它们共同的界面特征是，一般只具有标题栏，没有菜单栏，大小不可改变。而各类浏览器的界面风格则截然不同，如图 5.2 所示，通常都有菜单栏、工具栏、状态栏，还有主页面显示区。



图 5.2 浏览器的典型界面

要开发浏览器首先必须能做出一般浏览器都具备的标准界面来，很简单，只需在一开始创建工程时进行相关设置就可以了。

在“MFC 应用程序向导”中有一个“应用程序类型”页，如图 5.3 所示。

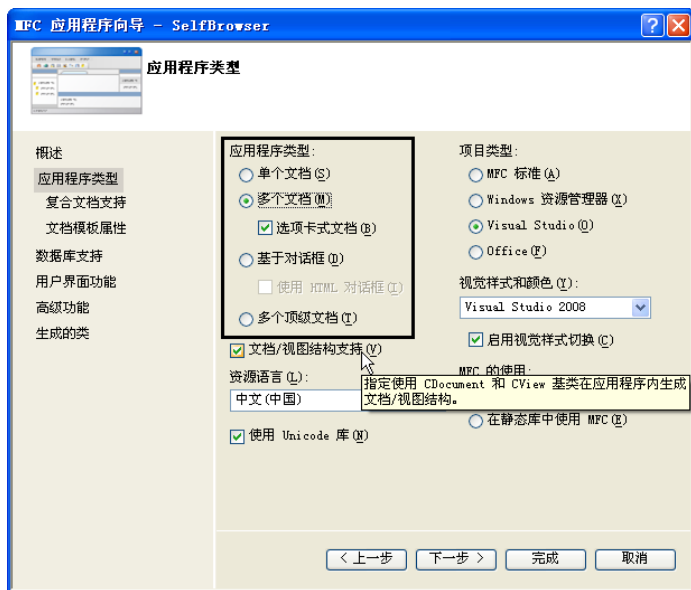


图 5.3 设置生成文档/视图类程序

MFC 向导生成的程序有 4 种类型 (图 5.3 中已框出): 单个文档、多个文档、基于对话框、多个顶级文档。之前编写的程序都是“基于对话框”类型，而本章开发浏览器必须采用其他类型，除了对话框类型之外，选择其他三种类型程序中的任何一种，系统都会默认选中下方的一个复选框“文档/视图结构支持”，如图 5.3 鼠标指针所指处。

那么,什么是“文档/视图结构”呢?

它是 MFC 提供的一种通用的 Windows 程序框架,大多数 Windows 程序(包括浏览器)采用的都是这种标准框架,用户在开发自己的 Windows 软件时可以直接拿来使用。

文档/视图结构是 Windows 程序通用的一种工作机制(架构),其中文档是应用程序数据基本元素的集合,它构成应用程序所使用的的数据单元,另外它还对数据进行管理和维护,通常将数据保存在文档类的成员变量中。文档是一种数据源,有很多种,最常见的是磁盘文件,也可以来自串口、网络或摄像机输入信号等。文档还负责将数据保存到永久存储介质中,常见的情况是将数据保存到磁盘文件或数据库中,在 Visual C++ 中,称这个过程为串行化(Serialize)。MFC 类库为数据的串行化提供了默认支持,只需在此基础上稍加修改就可为自定义的文档类提供串行化支持。

视图是数据的用户窗口,为用户提供了文档的可视显示,把文档的部分或全部内容在窗口中显示出来。视图还给用户与文档中的数据交互的界面,把用户的输入转化为对文档中数据的操作。每个文档都会有一个或多个视图显示,一个文档可以有多个不同的视图。另外,视图还可以直接或间接地访问文档类中的成员变量,并通过这种方式来显示和更新数据。

常用的文档/视图结构程序主要有两种:单文档界面(SDI)应用程序和多文档界面(MDI)应用程序。

在单文档程序中,用户在同一时刻只能操作一个文档,如 Windows 中的“记事本”就是这样的例子。打开文档时会自动关闭当前打开的活动文档,若文档修改后尚未保存,会提示是否保存所做的修改。因为一次只开一个窗口,因此不像 Microsoft Word 那样需要一个“窗口”菜单。单文档应用程序一般都提供一个 File 菜单,在该菜单下有一组命令,用于新建文档(New)、打开已有文档(Open)、保存或更名存盘等。这类程序相对比较简单,常见的有终端仿真程序和一些工具程序。

多文档界面应用程序允许同时操作多个文档。如 Microsoft Word,可以打开多个文件(为每个文件打开一个窗口),可以通过切换活动窗口激活相应的文档进行编辑。多文档应用程序也提供一个 File 菜单,用于新建、打开、保存文档。与单文档程序不同的是,它往往还提供一个 Close 菜单项,用于关闭当前打开的文档。多文档应用程序还提供“窗口”菜单,管理所有打开的子窗口,包括对子窗口进行关闭、层叠、平铺等操作。关闭一个窗口时,窗口内的文档也会被自动关闭。

文档/视图结构的提出大大地简化了多数应用程序的设计开发过程,它的主要优点如下。

(1) 将数据操作和数据显示、用户界面分离开。这是一种“分而治之”的思想,使得模块划分更加合理、独立性更强,同时也简化了数据操作、显示和用户界面工作。文档只负责数据管理,不涉及用户界面;视图只负责数据输出与用户界面的交互,可以不考虑应用程序的数据是如何组织的,甚至当文档中的数据结构发生变化时也不必改动视图的代码。

(2) MFC 在文档/视图结构上提供了许多标准操作界面,包括新建、打开、保存、打印文档等,减轻了用户的工作量。用户不必再书写这些重复的代码,从而可以把更多的精力放到完成应用程序特定功能的代码上。

(3) 支持打印预览和电子邮件发送功能。用户无须或只需编写很少的代码,就可以为应用程序提供打印预览功能。另外, MFC 提供在文档/视图结构中以电子邮件形式直接发送当前文档的功能。

但是,并非所有基于窗口的应用程序都要使用文档/视图结构。以下两种情况不宜采用文档/视图结构。

(1) 不是面向数据的应用或数据量很少的程序,如一些工具程序,包括磁盘扫描程序、时钟程序,还有一些过程控制程序等。

(2) 不使用标准的窗口用户界面的程序, 如一些小游戏等。

所有的文档类都以 `CDocument` 类为基类。`CDocument` 类提供了文档类所需要的最基本的功能实现。更重要的是, 它为文档对象及其和其他对象(如应用程序对象、框架、窗口等)的交互提供了一个框架。创建应用程序的工作基本上是在这个已有框架基础上, 添加与特定应用相关的实现。

视图类是从 `CView` 基类派生的, 是数据的用户窗口。视图规定了用户查看文档数据及同数据交互的方式。有时一个文档可能需要多个视图, 但有且仅有一个文档对象。一个文档对象可以由多个视图共用, 这样就能通过文档和视图之间的通信来实现视图和视图之间的通信。

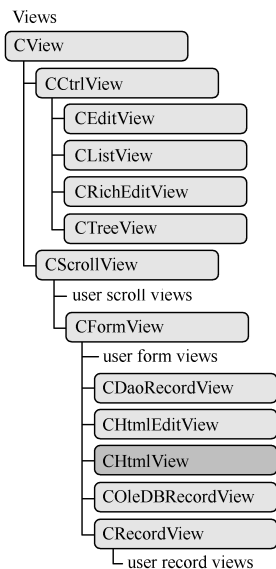


图 5.4 视图 `CView` 类家族

2. CHtmlView 类

在文档/视图结构中, 视图作为向用户展示文档数据的界面, 必然要求针对不同风格的程序产生不同类型的视图, 既然视图类都是从 `CView` 基类派生的, 那么 `CView` 类究竟可以派生出哪些种类的视图呢? 要开发浏览器又需要采用哪一种视图呢?

在 MFC 类库体系结构图中的 `CView` 类及其子类的继承结构, 即视图 `CView` 类家族如图 5.4 所示。如果文档需要卷滚, 则要从 `CScrollView` 派生出视图类; 如果希望视图按一个对话框模板资源来布置用户界面, 则可从 `CFormView` 派生; 而如果想要开发出浏览器那样风格的界面, 则可以从 `CHtmlView` 类派生出视图。

网络浏览器一般包括以下几个部分: HTML 解释器、HTML 执行器和应用界面控制。MFC 中提供的 `CHtmlView` 类能够很好地实现对 HTML 文档的解析和显示。在日常应用中, 浏览器除了访问 Web 站点, 还可以浏览本地和网络文件系统、维护历史记录等。以上这些, `CHtmlView` 类都可以轻松实现。开发浏览器时, 只需要将 `CHtmlView` 类作为 MFC 的文档/视图框架体系结构中的视图部分。

`CHtmlView` 类的成员函数及其功能见表 5.6。

表 5.6 CHtmlView 类成员函数及其功能

函 数 名	功 能 描 述
<code>GetLocationName</code>	得到站点名称
<code>GetOffline</code>	判断是否离线
<code>SetOffline</code>	指示是否离线浏览
<code>GetLocaionURL</code>	得到站点的 URL
<code>GetFullName</code>	得到浏览器的完整本地路径
<code>GetType</code>	得到当前页面类型
<code>Navigate</code>	浏览当前的 URL 文件
<code>Navigate2</code>	浏览当前的 URL 文件, 或者全路径表明的文件
<code>GoBack</code>	在历史记录中浏览前一项
<code>GoForward</code>	在历史记录中浏览后一项
<code>GoHome</code>	浏览开始页面
<code>GoSearch</code>	浏览当前的搜索页面
<code>Refresh</code>	重新下载当前页面
<code>Stop</code>	停止打开一个文件

3. WinInet 类

为了方便编写 Internet 应用程序, MFC 提供了 WinInet 的封装, 它是一些类和全局函数的集合, 包括对 HTTP、FTP、Gopher 等协议的实现。这样即使不理解 TCP/IP 协议和套接字编程, 也能开发出 Internet 应用程序。

WinInet 相关类的继承关系如图 5.5 所示。

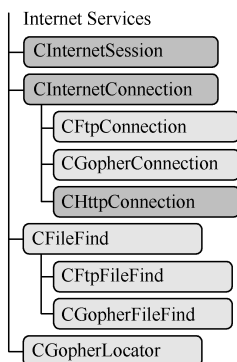


图 5.5 WinInet 类库

在 WinInet 提供的类中, 共分为以下 3 种类型: 处理 Internet 会话的类、处理 Internet 连接的类和处理文件的类。

WinInet 各种类功能的简要描述见表 5.7。

表 5.7 WinInet 类的功能描述

类 名	描 述
CInternetSession	创建 Internet 会话
CInternetConnection	管理和 Internet 服务器的连接
CFtpConnection	管理和 FTP 服务器的连接
CGopherConnection	管理和 Gopher 服务器的连接
CHttpConnection	管理和 HTTP 服务器的连接
CFileFind	执行本地文件搜索
CFtpFileFind	搜索 FTP 服务器上的文件
CGopherFileFind	搜索 Gopher 服务器上的文件
CGopherLocator	得到 Gopher 定位器
CInternetException	Internet 操作相关异常

其中, Gopher 已经没有人使用了, 但 HTTP 和 FTP 仍然是 Internet 主流的应用协议, 本章接下来开发浏览器和第 6 章开发 FTP 客户端时都会用到 WinInet 类。通常人们把仅使用 MFC WinInet 类开发网络应用程序的活动称为 Internet 编程, 它直接使用 MFC 以避免网络协议的实现细节, 这样就能更加快速、高效地开发出实用可靠的客户端程序。Internet 编程也是网络编程的一个重要分支。

5.2.2 定制开发自己的浏览器

本节将利用 CHtmlView 类的功能, 配合使用 WinInet 提供的接口开发一个浏览器。

1. 创建文档/视图工程

本例浏览器程序工程的创建过程与前几章有很大的不同,因为我们要开发的不是对话框类的程序,而是基于文档/视图框架的程序。现将创建工程的过程描述如下,请读者注意其与对话框工程的区别。

创建 MFC 工程,工程名为 SelfBrowser (自己定制的浏览器)。在“MFC 应用程序向导”对话框的“应用程序类型”页中选择“单个文档”类型的程序,如图 5.6 所示。



图 5.6 创建单个文档程序

单击“下一步”按钮,接下来几个页面保持默认设置,直到“用户界面功能”页,如图 5.7 所示,在其上选中“使用经典菜单”→“使用浏览器样式的工具栏”。

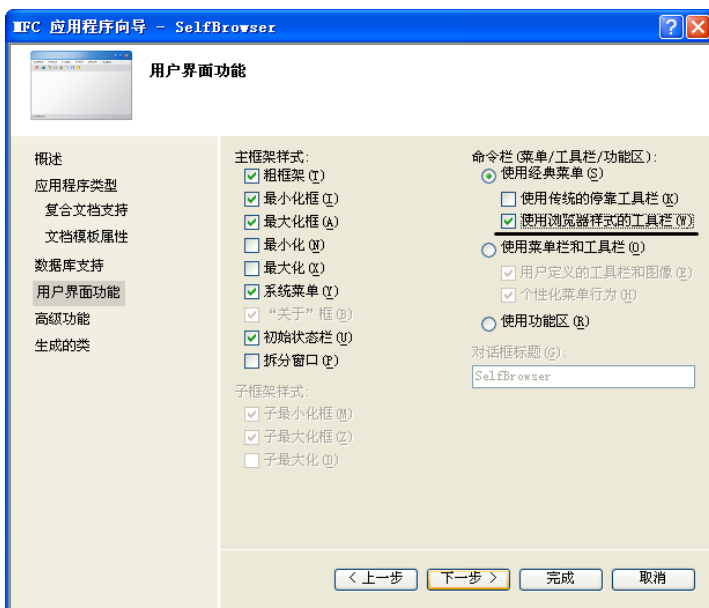


图 5.7 使用浏览器样式的工具栏

然后一直单击“下一步”按钮……直到出现“生成的类”页面，中间的步骤都保持向导默认的设置。注意，本例工程在“高级功能”页中不需要勾选“Windows 套接字”复选框，因为即将使用的类 CHtmlView 已经封装了 IE 内核，它已包含 Socket 功能，所以就不再需要用户自己去实现了。

文档/视图结构相关的类有 CXXXApp、CMainFrame、CXXXView 和 CXXXDoc。它们分别是应用程序类 CWinApp、框架窗口类 CFrameWnd、视图类 CView 和文档类 CDocument 的派生类，其中 XXX 表示工程名。可以看到，VC 工程向导正是按照这个规律生成类的，图 5.8 中框出了本例向导自动生成的 4 个类：CSelfBrowserView、CSelfBrowserApp、CSelfBrowserDoc 和 CMainFrame。要在项目中使用 CHtmlView 类的功能，必须使视图继承自 CHtmlView，在“生成的类”列表中选择视图类 CSelfBrowserView，在下面的“基类”下拉列表框中选择 CHtmlView，单击“完成”按钮。



图 5.8 文档/视图结构相关类

工程创建好后就可以运行了，界面效果如图 5.9 所示。

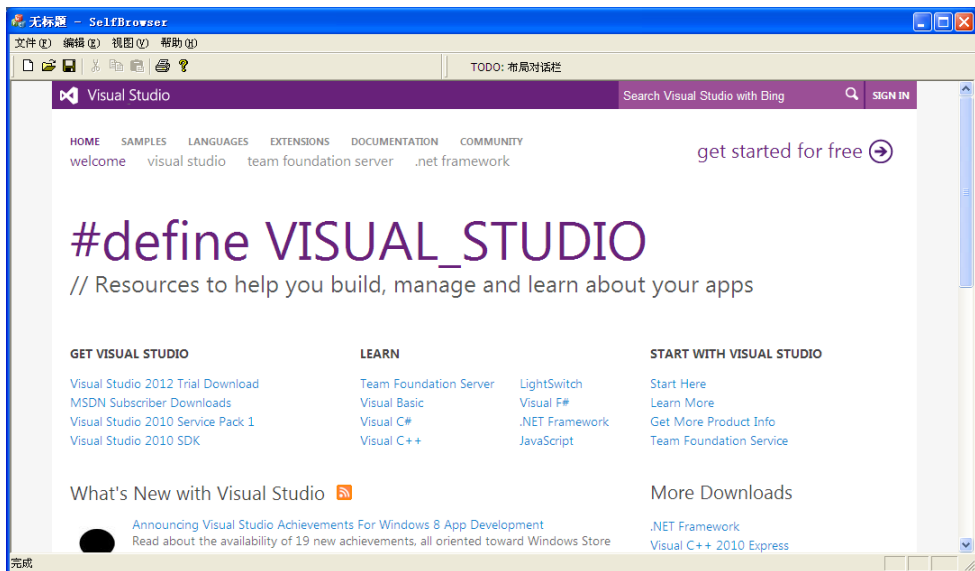


图 5.9 自动生成的浏览器程序

我们没有写一句代码，程序竟然可以自动访问微软 Visual Studio 官方网站！那是因为创建的工程中已自动集成了 IE 浏览器内核的全部基本功能，接下来只需要在此基础上开发，就可以很容易地制作出自己的浏览器了。

2. 添加地址栏

浏览器都有一个地址栏，由用户输入网址回车后访问相应的网站，这是浏览器最基本的功能，下面首先来实现这个功能。

在资源视图展开的目录树下双击 Dialog 目录的第二个项目 IDR_MAINFRAME，在工作区设计地址栏，界面如图 5.10 所示。

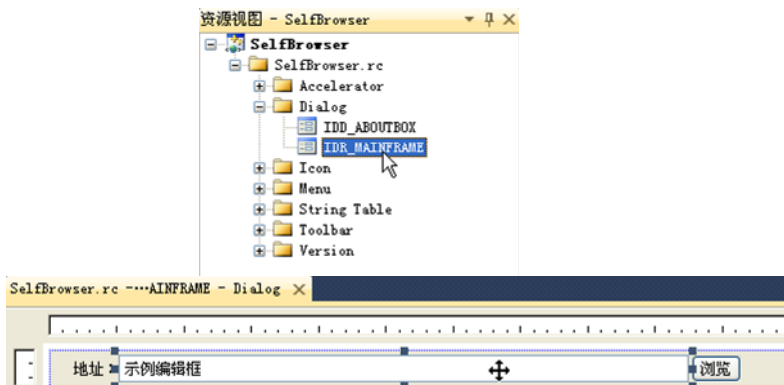


图 5.10 自定义地址栏

将输入网址的文本编辑框 ID 设置为 IDC_EDIT_ADDRESS，程序代码中要引用。给“浏览”按钮添加事件过程，如图 5.11 所示。

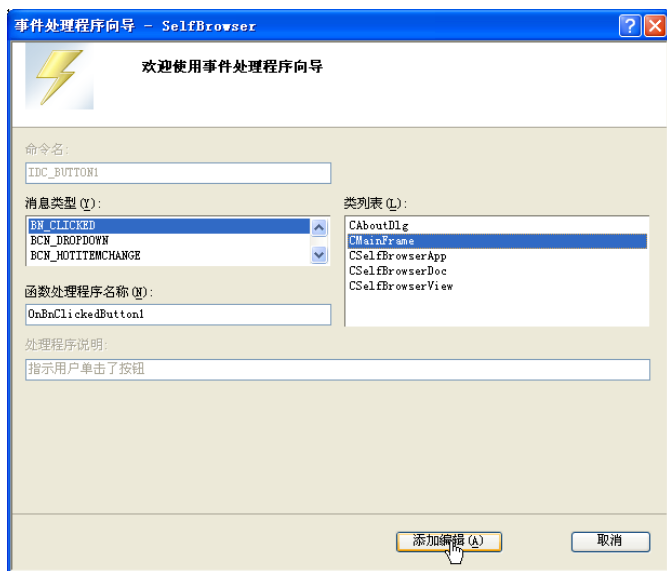


图 5.11 添加“浏览”按钮的事件过程

代码如下：

```
CString sWebAddress;
//从编辑框获取用户输入的 Web 地址
m_wndDlgBar.GetDlgItem(IDC_EDIT_ADDRESS)->GetWindowText(sWebAddress);
```

//浏览相应的网页

((CHtmlView *)GetActiveView())->Navigate(sWebAddress);

Navigate()函数是 CHtmlView 类的常用方法之一，用于获得指定网址的页面，并将它返回给视图。

通常用户上网时在地址栏输入想访问的网站地址后，不一定非要单击“浏览”按钮，直接按回车键也可以。为了使本程序也支持这样的快捷功能，就要使程序能够识别回车事件，这要利用 Windows 程序内置的消息映射机制，具体方法如下。

在文件 MainFrm.h 中声明消息映射 OnInputAddress()函数，并在 MainFrm.cpp 中定义对应的消息映射：

```
afx_msg void OnInputAddress();           //消息映射函数
ON_COMMAND(IDOK, OnInputAddress)        //消息映射
```

上面两句代码添加的具体位置如图 5.12 所示。

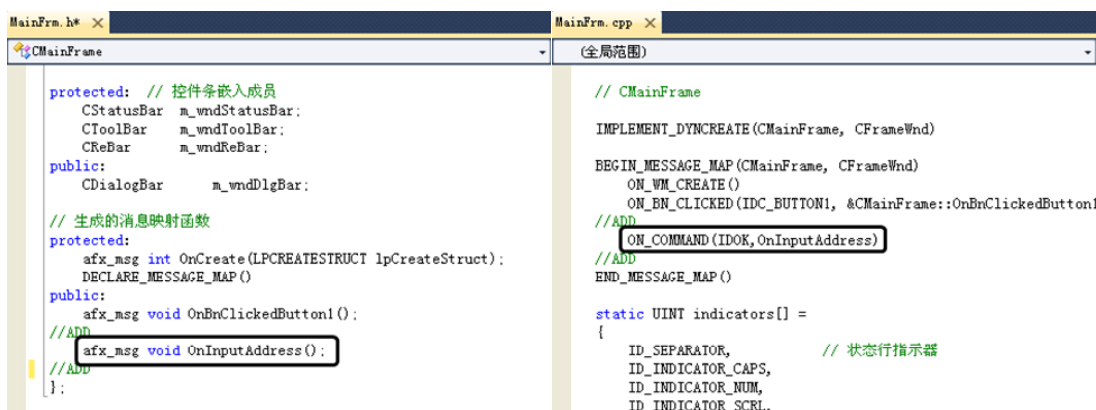


图 5.12 消息映射代码的添加位置

OnInputAddress()函数的代码与“浏览”按钮的事件过程代码完全一样。

3. 设计主菜单

一般的浏览器都有“前进”、“后退”、“停止”、“刷新”、“转到主页”等网页导航功能，这些功能在 CHtmlView 类中都有相应的实现函数，直接调用就行。下面来添加这些功能的菜单项。

在资源视图展开的目录树下双击 Menu 目录下的 IDR_MAINFRAME 项，打开菜单设计工作区，设计主菜单如图 5.13 所示。



图 5.13 设计主菜单

“视图”主菜单下还添加了一个“源文件”菜单项，该功能是用来查看所浏览网页的 HTML 源代码的，这个扩展的功能需要用到 WinInet 类，具体内容将在稍后介绍。

Windows 程序往往都同时具有主菜单和工具栏,工具栏里的按钮功能在主菜单及其子菜单选项里都有,以方便用户操作。为了能在工具栏中定制与菜单项里一样的功能,同时也是为了后面程序编码的需要,在这里先给每个菜单项指定 ID 标识,如图 5.14 和表 5.8 所示。

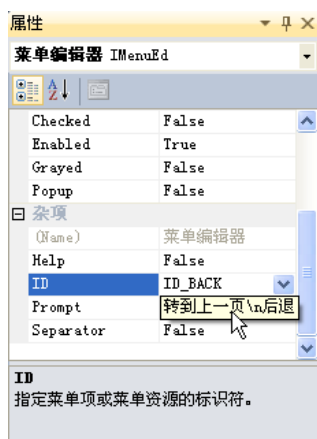


图 5.14 给菜单项指定 ID 标识

其中, Prompt 属性 “\n” 之前的文字用于指定选定菜单项时出现在状态栏中的文本, “\n” 之后的文字则是鼠标放在工具栏对应功能按钮上时出现的提示文本。

给 “后退” 子菜单项添加事件处理程序, 如图 5.15 所示。



图 5.15 添加 “后退” 菜单项的处理程序

编写 OnBack()函数过程的代码, 如图 5.16 所示。

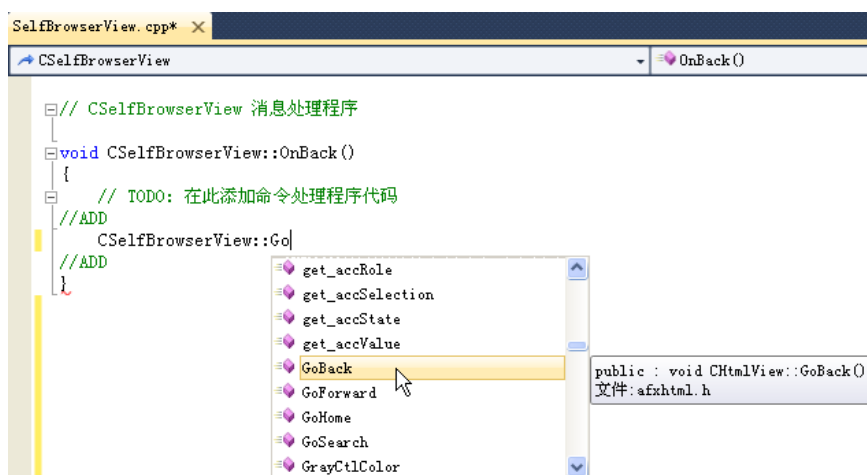


图 5.16 OnBack()代码

由于 CSelfBrowserView 类继承自 CHtmlView, 故它也包含了 CHtmlView 类的所有方法。由图 5.16 可知, CSelfBrowserView 类的方法列表中有 GoBack()、GoForward()、GoHome()……这些浏览器通用的基本功能都已经具备了, 直接从列表中选择相应的方法即可。

用上述操作调用适当的方法分别实现主菜单中设计的各子菜单项的功能, 代码如下:

```
//CSelfBrowserView 消息处理程序
void CSelfBrowserView::OnBack()
{
    CSelfBrowserView::GoBack();           //后退
}
void CSelfBrowserView::OnForward()
{
    CSelfBrowserView::GoForward();        //前进
}
void CSelfBrowserView::OnHome()
{
    CSelfBrowserView::GoHome();           //主页
}
void CSelfBrowserView::OnStop()
{
    CSelfBrowserView::Stop();             //停止
}
void CSelfBrowserView::OnRefresh()
{
    CSelfBrowserView::Refresh();          //刷新
}
```

4. 自定义工具栏

在工具栏里定制与前面菜单项一样的功能。在资源视图展开的目录树下双击 Toolbar 目录下的 IDR_MAINFRAME 项 (如图 5.17 所示), 打开工具栏定制区。



图 5.17 打开工具栏定制区

工具栏中已经存在了一些功能按钮，如“新建”、“打开”、“保存”、“复制”、“粘贴”等，需要用户自己绘制对应于菜单项里的功能按钮，绘制完成后将它的 ID 和 Prompt 属性设置成与菜单项一样，这样在程序运行时单击工具栏上的按钮，就和选择相应菜单项的效果一样了，如图 5.18 所示。

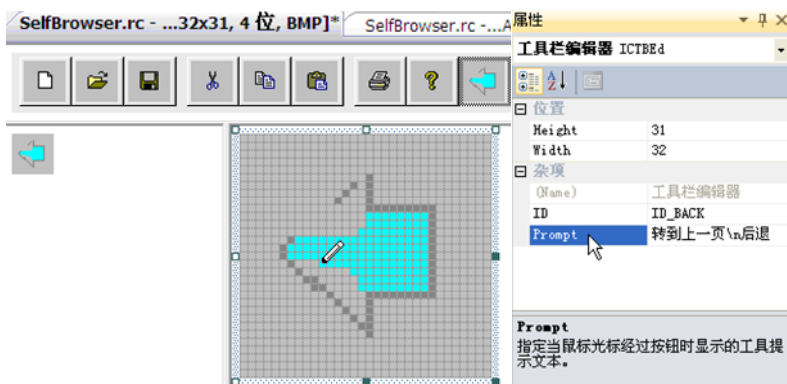


图 5.18 绘制工具栏

最终设计完成的工具栏如图 5.19 所示。

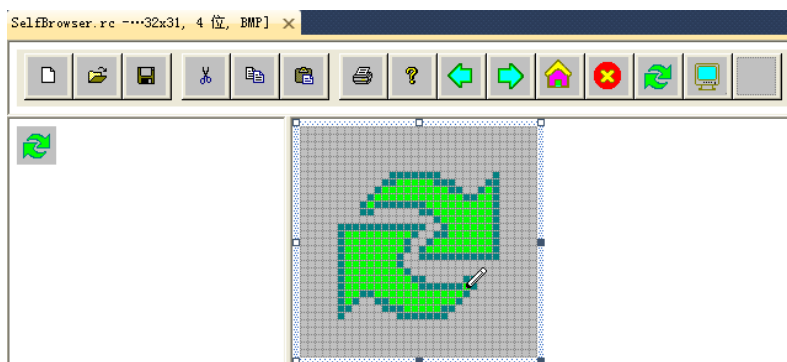


图 5.19 工具栏设计效果

至此，浏览器的基本功能已经具备。

5. 完善程序和试运行

大家平时上网的时候只要留心一下就会发现这样一个现象：当单击网页上的链接跳转到一个新页面时，浏览器地址栏及标题栏上显示的地址也会随之同步更新。下面就来实现这样的功能。

为视图类 CSelfBrowserView 添加 OnDocumentComplete()方法（如图 5.20 所示）。

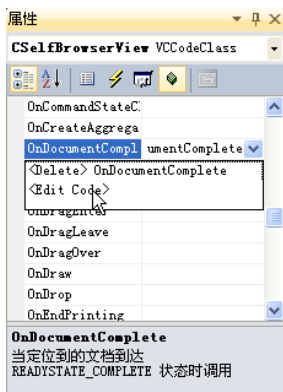


图 5.20 为视图添加 OnDocumentComplete()方法

这个方法的注释“当定位到的文档到达 READYSTATE_COMPLETE 状态时调用”，其意思是说浏览器接收到服务器发来的新页面时就执行这个方法。在这个方法中可以编写使浏览器标题栏和地址栏随页面同步刷新的代码：

```
//每次当前页面改变，更新地址栏的内容和窗口的标题
((CMainFrame *)GetParentFrame()->SetURL(lpszURL);
GetDocument()->SetTitle(lpszURL);
```

这里调用了 SetURL()函数，它是我们自定义的，添加方法如下。

(1) 在 SelfBrowserView.cpp 文件中包含头文件声明：

```
#include "MainFrm.h"
```

(2) 在 MainFrm.cpp 中编写 SetURL()函数：

```
void CMainFrame::SetURL(LPCTSTR lpszURL)
{
    m_wndDlgBar.GetDlgItem(IDC_EDIT_ADDRESS)->SetWindowTextA(lpszURL);
}
```

编译运行程序，在地址栏输入南京师范大学教务处网址“http://jwc.njnu.edu.cn/index.aspx”，单击“浏览”按钮或直接回车，浏览器将显示“南京师范大学教务处菁林园”首页，如图 5.21 所示。



图 5.21 访问“南京师范大学教务处菁林园”首页

在此演示一下跳转页面后地址栏和标题栏同步更新的效果。单击首页上的“组织机构”链接，将跳转到教务处处长办公室页面，如图 5.22 所示，此时看到浏览器标题栏和地址栏也都随之更新为“http://jwc.njnu.edu.cn/Organization/index.aspx”。

单击工具栏上的“后退”按钮或选择菜单命令“视图”→“转到”→“后退”，网页又回到“南京师范大学教务处菁林园”首页。



图 5.22 标题栏和地址栏同步更新

单击工具栏上的“主页”按钮或选择菜单命令“视图”→“转到”→“主页”，浏览器将显示默认的主页面，如图 5.23 所示。



图 5.23 转到主页

读者还可自行测试“前进”、“停止”、“刷新”等其他导航功能。

6. 用 Wininet 扩展功能

除了基本功能外，一般的浏览器还有扩展功能，用 Wininet 类可以实现很多扩展功能。下面将给浏览器增加一个“显示网页源代码”功能，这在多数浏览器中也是常见的功能之一。

给工程添加一个对话框，如图 5.24 所示。

设计这个新的对话框为浏览网页源码的窗口，界面布局如图 5.25 所示。

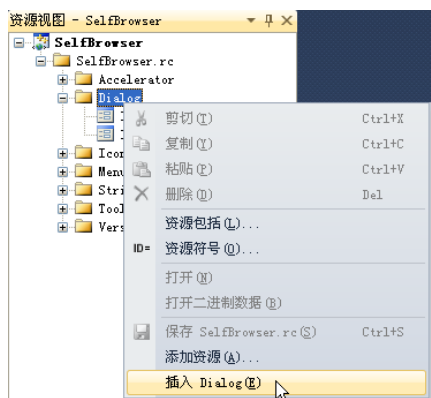


图 5.24 为工程添加对话框

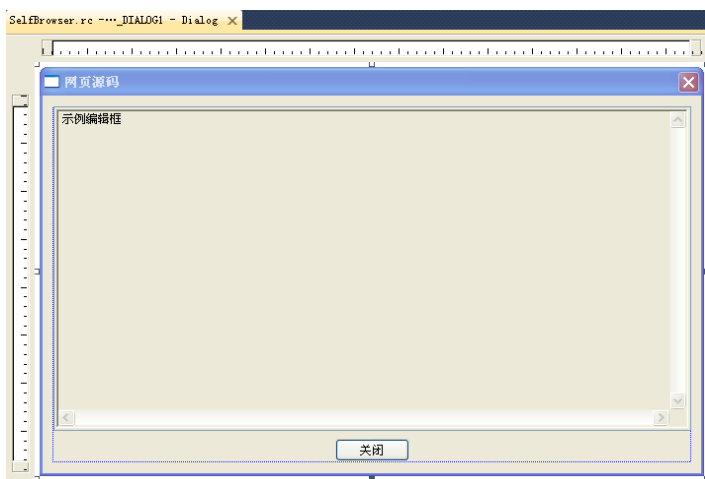


图 5.25 新对话框界面布局

为项目添加一个新的 MFC 类，命名为 CHtmlCodeViewDlg，如图 5.26 所示，在“对话框 ID”下拉列表中选择 IDD_DIALOG1（新加对话框的 ID），这样就将新添加的对话框与类关联起来了。



图 5.26 新对话框与新类关联

在类视图中可以看到新添加的类，如图 5.27 所示。

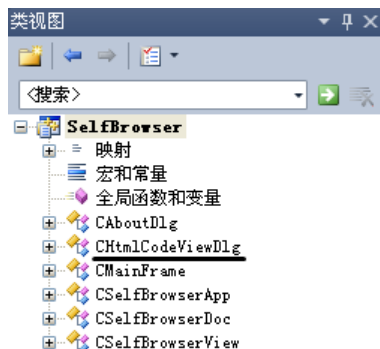


图 5.27 新添加的 CHtmlCodeViewDlg 类

在主菜单“视图”下新增设一个子菜单项“源文件”，并为其添加事件处理程序。处理过程命名为 OnCode(), 消息类型选 COMMAND，类列表中选择 CSelfBrowserView 类，如图 5.28 所示。

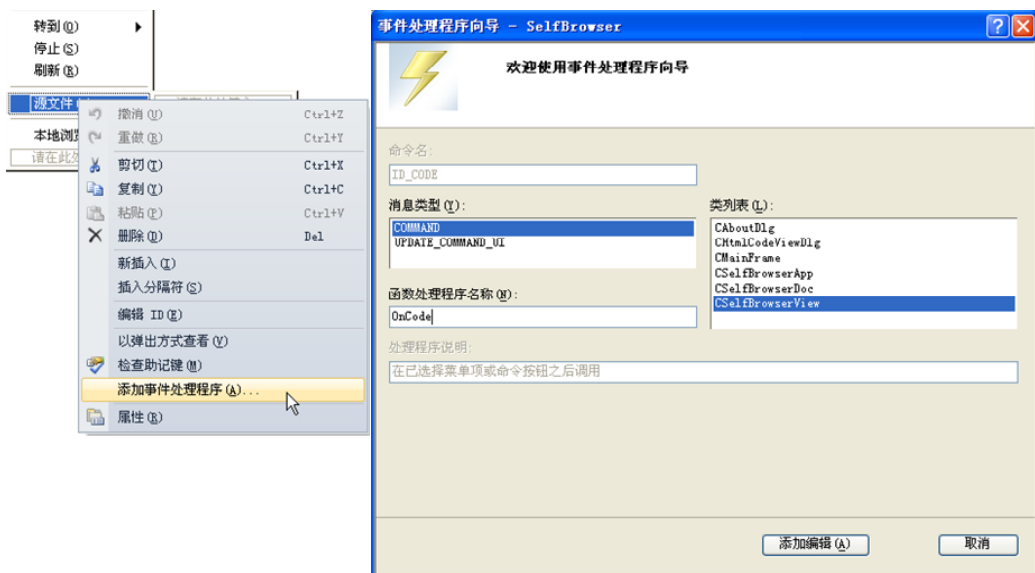


图 5.28 增设“源文件”子菜单及其处理程序

OnCode()函数过程代码如下：

```
void CSelfBrowserView::OnCode()
{
    CHtmlCodeViewDlg dlg;    //对话框对象
    dlg.DoModal();           //生成显示网页源码的对话框
}
```

此代码位于文件 SelfBrowserView.cpp 中，为了能在程序中引用 CHtmlCodeViewDlg 类，必须在源文件 SelfBrowserView.cpp 中包含头文件：

```
#include "HtmlCodeViewDlg.h"
```

OnCode()函数过程的代码只是生成了用于显示网页源码的对话框窗体对象，而显示源码的过程还需要对话框自身去实现，将这个实现过程放在它的初始化代码中。

为 CHtmlCodeViewDlg 类添加初始化过程，如图 5.29 所示。

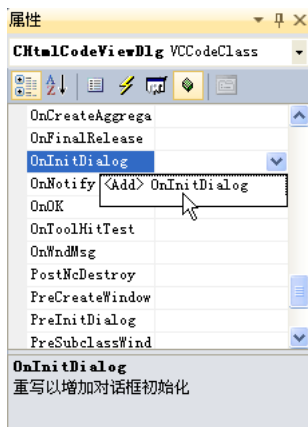


图 5.29 添加初始化过程

在 HtmlCodeViewDlg.cpp 中包含头文件:

```
#include "MainFrm.h"
```

```
#include "afxinet.h" //用到 WinInet 类
```

给显示网页源码对话框中的文本框添加变量关联 m_htmlCode, 如图 5.30 所示。



图 5.30 关联变量 m_htmlCode

为了编写程序时引用的方便, 将浏览器工具栏对象变量设置成 public 类型, 如图 5.31 所示。

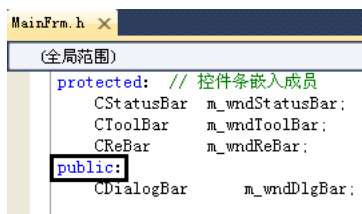


图 5.31 设置工具栏对象为 public 类型

编写网页源码对话框的初始化代码:

```
CWaitCursor wait;                                //等待
CInternetSession session("Self Net");            //新建会话
CStdioFile *pFile = NULL;                        //文件对象
CString sWebAddress;
//从地址栏获取 URL
(((CMainFrame*)GetParentFrame())->m_wndDlgBar).GetDlgItem(
    IDC_EDIT_ADDRESS)->GetWindowTextA(sWebAddress);
pFile = session.OpenURL(sWebAddress);            //打开 URL
if(pFile != NULL)
{
    CString str,allText,crlf = "\r\n";            //回车换行
    while(pFile->ReadString(str))                //读入页面内容
    {
        allText += crlf + str;
    }
    this->m_htmlCode = allText;
    UpdateData(false);                            //在对话框中显示网页源代码
    pFile->Close();                                //关闭文件
}
session.Close();                                //关闭会话
```

以上代码中用到了 CInternetSession 类来创建 Internet 会话,这是一个 WinInet 类,它封装了 HTTP 协议和 HTML 源码解析的过程,使用户在编程时可以直接通过它创建的会话对象获取到网页源码,非常方便。

下面来演示一下浏览器显示网页源码的功能。运行程序,用它打开新浪首页(其他网站的页面也可以),选择菜单命令“视图”→“源文件”,如图 5.32 所示。



图 5.32 访问新浪首页

可以看到弹出的“网页源码”对话框中显示出新浪首页的 HTML 代码，如图 5.33 所示。

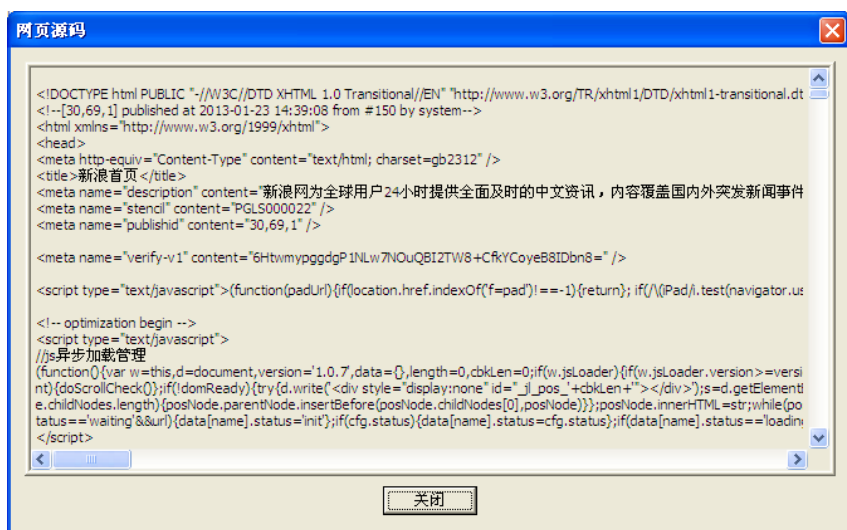


图 5.33 查看新浪首页的 HTML 代码

5.3 Web 服务器的开发

作为万维网主流客户端—服务器（C/S）结构的核心，服务器程序在网络应用中处于中心地位。本节介绍 Web 服务器的开发，为了便于读者理解程序，特从操作 Web 服务器的管理员视角，将这个软件的全部源代码划分成如下 4 个层次：

- 软件界面总控代码；
- 服务流程实现代码；
- HTTP 协议实现代码；
- 协议实现辅助代码。

再加上所有 VC 程序都有的框架（包括变量/结构体/类和宏的定义、头文件包含、初始化等）的代码，一共是五部分代码，按照编程及由表及里的顺序分别介绍。

服务器程序由于要直接实现应用协议，故一般都较为复杂，代码量很多。作为入门教程，考虑到读者的基础——尚无法独立写出（甚至看懂）规模稍大（数百行至近千行代码）的程序，因此本书所给出的服务器（包括本章 Web 服务器和第 6 章的 FTP 服务器）程序，都不要求读者自己编写出来，只要能对照书上的介绍大致看懂、运行成功即可，着重于理解它们的工作原理。读者如果真的渴望能自己写出这样的程序，请深入学习网络编程进阶的专业书籍并参考 MSDN 上的文档。

5.3.1 项目框架的建立

经过本书之前各章实例程序的学习，大家已经知道编写每个 VC 程序之前都要先建立项目框架（创建工程、设计 GUI 界面、定义结构和类、声明变量和函数、包含头文件、载入库、建立程序的事件驱动机制、初始化等）——客户端软件如此，服务器程序自然也不例外。

本章开发 Web 服务器程序，首先创建 VC 项目，工程名为 httpSrver (“基于 HTTP 协议的服务器”之意)。设计程序界面如图 5.34 所示。

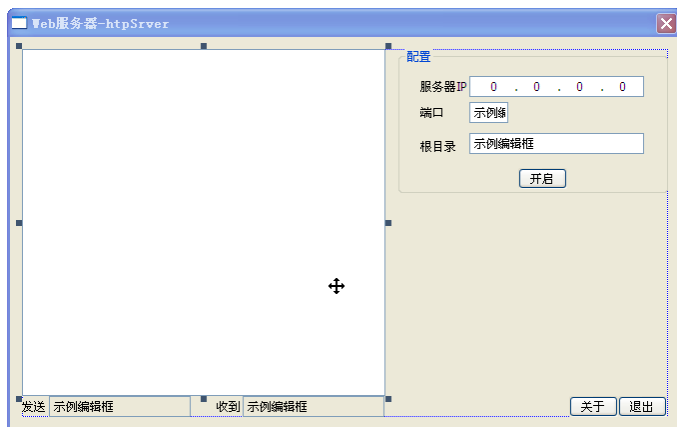


图 5.34 Web 服务器界面

操作 Web 服务器的管理员启动服务器的程序，在 GUI 界面上配置“服务器 IP”、“端口”项，填写存放本站网页资源的根目录路径，单击“开启”按钮就可以提供网站的万维网服务了。

GUI 界面上的各控件关联变量见表 5.9。

表 5.9 Web 服务器 httpSrver 的界面控件变量

控 件 \ 变 量	Control	Value
“服务器 IP”地址控件	LocalIP	—
“端口”文本框	locaPort	m_nPort(UINT)
“根目录”文本框	rootdir	m_strRootDir(CString)
“开启”按钮	m_StartStop	—
服务器状态列表控件	m_StatLst	—
“发送”只读文本框	m_Transfer	dwTransferred(DWORD)
“收到”只读文本框	m_Recv	dwReceived(DWORD)
“退出”按钮	m_exit	—

Web 服务器程序的运行离不开 HTTP 协议，在本例中定义一个类 CHttpProtocol，用它来专门实现 HTTP 协议，向项目中添加这个类。另外，还要添加两个结构体——REQUEST 和 HTTPSTATS，分别用来保存一个网络连接的客户端和服务器的信息，如图 5.35 所示。

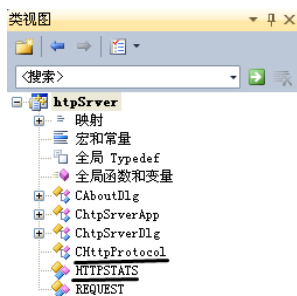


图 5.35 添加的结构体和类

在 HttpProtocol.h 中, 定义本程序将要使用的宏、结构体和类:

```
#ifndef _HTTPPROTOCOL_H
#define _HTTPPROTOCOL_H
#pragma once
#include "stdafx.h"
#include "afxmt.h"
#define HTTPPORT 80
#define METHOD_GET 0
#define METHOD_HEAD 1
//宏定义
#define HTTP_STATUS_OK "200 OK"
#define HTTP_STATUS_CREATED "201 Created"
#define HTTP_STATUS_ACCEPTED "202 Accepted"
#define HTTP_STATUS_NOCONTENT "204 No Content"
#define HTTP_STATUS_MOVEDPERM "301 Moved Permanently"
#define HTTP_STATUS_MOVEDTEMP "302 Moved Temporarily"
#define HTTP_STATUS_NOTMODIFIED "304 Not Modified"
#define HTTP_STATUS_BADREQUEST "400 Bad Request"
#define HTTP_STATUS_UNAUTHORIZED "401 Unauthorized"
#define HTTP_STATUS_FORBIDDEN "403 Forbidden"
#define HTTP_STATUS_NOTFOUND "404 File can not fonund!"
#define HTTP_STATUS_SERVERERROR "500 Internal Server Error"
#define HTTP_STATUS_NOTIMPLEMENTED "501 Not Implemented"
#define HTTP_STATUS_BADGATEWAY "502 Bad Gateway"
#define HTTP_STATUS_UNAVAILABLE "503 Service Unavailable"
//连接的 Client 的信息
typedef struct REQUEST
{
    HANDLE hExit;
    SOCKET Socket; //请求的 socket
    int nMethod; //请求的使用方法: GET 或 HEAD
    DWORD dwRecv; //收到的字节数
    DWORD dwSend; //发送的字节数
    HANDLE hFile; //请求连接的文件
    char szFileName[_MAX_PATH]; //文件的相对路径
    char postfix[10]; //存储扩展名
    char StatuCodeReason[100]; //头部的 status cod 及 reason-phrase
    bool permitted; //用户权限判断
    char * authority; //用户提供的认证信息
    char key[1024]; //正确认证信息
    void* pHttpProtocol; //指向类 CHttpProtocol 的指针
}REQUEST, *PREQUEST;
typedef struct HTTPSTATS
{
    DWORD dwRecv; //收到字节数
    DWORD dwSend; //发送字节数
}HTTPSTATS, *PHTTPSTATS;
```

```

#include <map>
#include <string>
using namespace std;
class CHttpProtocol
{
public:
    HWND                m_hwndDlg;
    SOCKET              m_listenSocket;
    //保存 content-type 和文件后缀的对应关系 map
    map<CString, char *> m_typeMap;
    CWinThread*         m_pListenThread;
    HANDLE              m_hExit;
    static HANDLE       None;                //标志是否有 Client 连接到 Server
    static UINT         ClientNum;           //连接的 Client 数量
    static CCriticalSection m_critSect;      //互斥变量
    CString             m_strRootDir;        //web 的根目录
    UINT               m_nPort;             //http server 的端口号

public:
    CHttpProtocol(void);
    void DeleteClientCount();
    void CountDown();
    void CountUp();
    HANDLE InitClientCount();
    void StopHttpSrv();
    bool StartHttpSrv();
    static UINT ListenThread(LPVOID param);
    static UINT ClientThread(LPVOID param);
    bool RecvRequest(PREQUEST pReq, LPBYTE pBuf, DWORD dwBufSize);
    int Analyze(PREQUEST pReq, LPBYTE pBuf);
    void Disconnect(PREQUEST pReq);
    void CreateTypeMap();
    void SendHeader(PREQUEST pReq);
    int FileExist(PREQUEST pReq);
    void GetCurentTime(LPSTR lpszString);
    bool GetLastModified(HANDLE hFile, LPSTR lpszString);
    bool GetContentType(PREQUEST pReq, LPSTR type);
    void SendFile(PREQUEST pReq);
    bool SendBuffer(PREQUEST pReq, LPBYTE pBuf, DWORD dwBufSize);

public:
    ~CHttpProtocol(void);
};
#endif _HTTPPROTOCOL_H

```

可以看到，其中包括了上面刚刚添加到项目中的类 `CHttpProtocol` 及结构体 `REQUEST` 和 `HTTPSTATS` 的定义（代码中加黑部分）。大家会发现类 `CHttpProtocol` 貌似很庞大，包含的变量和方法很多，除此之外，这个头文件中还有一些用途不明的代码。不过，不用紧张——作为一个服务器程序，对初学网络编程者来说相对复杂一点是很正常的。读者只需将上面这段代码复制到

自己创建的对应工程文件中就可以了,至于这些定义的用途及诸多方法的功能实现将在下文逐一展示和说明。

在 `httpSrverDlg.h` 中包含头文件:

```
#include "HttpProtocol.h"
```

在 `httpSrverDlg.h` 头文件的类 `ChtpSrverDlg` 中声明变量:

```
public:
```

```
    CHttpProtocol *pHttpProtocol;
```

```
    bool m_bStart;
```

同时声明函数:

```
afx_msg LRESULT AddLog(WPARAM wParam, LPARAM lParam);
```

```
afx_msg LRESULT ShowData(WPARAM wParam, LPARAM lParam);
```

大家可能注意到,这两个函数的原型形式比较面熟——对了!在第4章4.3.3节中曾经见到过这个函数“`LRESULT OnReadMsg(WPARAM wParam, LPARAM lParam)`”,它是用来建立程序的事件驱动机制的。本例为了将服务器的运行状态动态显示在监控屏幕上,还需要编写两个采用与第4章及前面UDP编程中相同驱动机制的函数,用来向服务器界面中的列表中添加实时日志和显示数据收发流量,分别用 `AddLog()` 和 `ShowData()` 实现,原理一样。

在 `stdafx.h` 中添加宏定义,代码如下:

```
#define LOG_MSG WM_USER + 1
```

```
#define DATA_MSG WM_USER + 2
```

再在 `httpSrverDlg.cpp` 中如图5.36所示的位置添加消息宏映射。

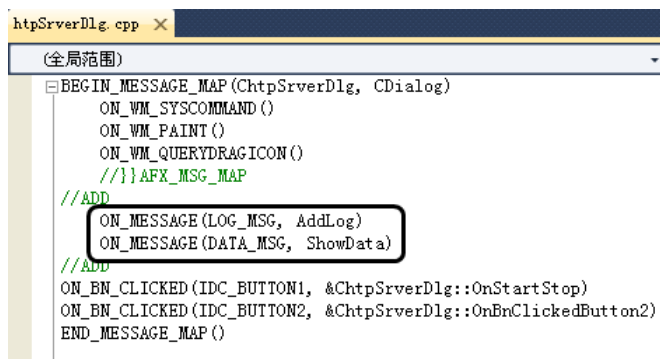


图 5.36 添加消息宏映射

```
ON_MESSAGE(LOG_MSG, AddLog)
```

```
ON_MESSAGE(DATA_MSG, ShowData)
```

这样,本程序的事件驱动机制就建好了。

最后,在 `BOOL ChtpSrverDlg::OnInitDialog()` 中添加初始化代码如下:

```
m_bStart = false;
```

```
pHttpProtocol = NULL;
```

```
locPort.SetWindowTextA("");
```

至此,程序框架建立完毕。

5.3.2 Web 服务器界面总控

一般网站的 Web 服务器都是由网络管理员配置、操作和监控的。网络管理员通常不会去写

程序，只要熟悉网络技术的原理，会操作服务器就行。鉴于网络管理员是与服务器程序直接“亲密接触”的用户，本章（及第 6 章）就从网络管理员的角度来看待服务器的程序。

网络管理员通过单击鼠标操作图形用户界面（如图 5.37 所示）来运行 Web 服务程序，在单击界面上的控件时，直接触发的是软件界面总控代码。下面先向读者展示这些代码。

“开启”按钮的事件代码如下：

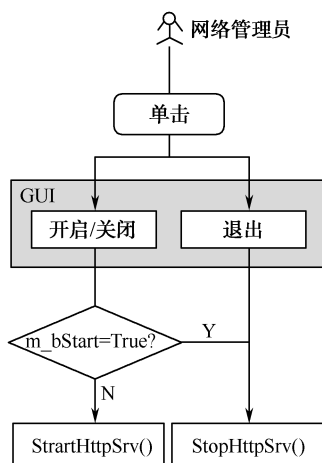


图 5.37 管理员操纵 Web 服务器

```

void ChtpSrverDlg::OnStartStop()
{
    this->UpdateData(); //获取用户配置
    if(m_strRootDir.IsEmpty())
    {
        AfxMessageBox("请设置本站 Web 页存放的根路径!");
        return;
    }
    if ( !m_bStart )
    {
        pHttpProtocol = new CHttpProtocol; //http 协议类
        pHttpProtocol->m_strRootDir = m_strRootDir;
        pHttpProtocol->m_nPort = m_nPort;
        pHttpProtocol->m_hwndDlg = m_hWnd;
        //启动 http 服务
        if (pHttpProtocol->StartHttpSrv())
        {
            m_StartStop.SetWindowTextA("关闭");
            LocaIP.EnableWindow(false);
            locaPort.EnableWindow(false);
            rootdir.EnableWindow(false);
            m_exit.EnableWindow(false);
            m_bStart = true; //服务的开关状态
        }
    }
    else
    {
        if(pHttpProtocol)
  
```

```

        {
            delete pHttpProtocol;
            pHttpProtocol = NULL;
        }
    }
}
else
{
    //关闭 http 服务
    pHttpProtocol->StopHttpSrv();
    m_StartStop.SetWindowTextA("开启");
    LocalIP.EnableWindow(true);
    localPort.EnableWindow(true);
    rootdir.EnableWindow(true);
    m_exit.EnableWindow(true);
    if(pHttpProtocol)
    {
        delete pHttpProtocol;
        pHttpProtocol = NULL;
    }
    m_bStart = false;
}
}

```

程序的流程一目了然，通过这一按钮来控制 Web 服务的开关，而服务的开关状态用布尔变量 `m_bStart` 来标识。

服务开启后，界面上会动态显示运行信息，便于管理员监控服务器。这一功能就是使用 5.3.1 节中声明的两个基于自定义事件驱动机制的函数 `AddLog()` 和 `ShowData()` 实现的，它们的代码分别展示如下。

`AddLog()` 代码如下：

```

//显示日志信息
LRESULT ChttpServerDlg::AddLog(WPARAM wParam, LPARAM lParam)
{
    char szBuf[284];
    CString *strTemp = (CString *)wParam;
    SYSTEMTIME st;
    GetLocalTime(&st);
    wprintf(szBuf, "%02d:%02d:%02d.%03d   %s",
            st.wHour, st.wMinute, st.wSecond, st.wMilliseconds, *strTemp);
    m_StatList.AddString(szBuf);
    m_StatList.SetTopIndex(m_StatList.GetCount() - 1);
    delete strTemp;
    strTemp = NULL;
    return 0L;
}

```

`ShowData()` 代码如下：

```

//显示接收和发送的数据流量
LRESULT ChttpServerDlg::ShowData(WPARAM wParam, LPARAM lParam)
{

```

```

PHTTPSTATS pStats = (PHTTPSTATS)wParam;
dwReceived += pStats->dwRecv;
dwTransferred += pStats->dwSend;
TRACE1("Rev %d\n", pStats->dwRecv);
TRACE1("Send %d\n", pStats->dwSend);
TRACE1("Total Rev %d\n", dwReceived);
TRACE1("Total Send %d\n", dwTransferred);
UpdateData(false);
return 0L;
}

```

前面定义的 HTTPSTATS 结构体在这里就派上用场了,用它来实时监控服务器收发数据的流量。

若服务器故障或网站需要维护,管理员会暂时关闭服务器,单击“退出”按钮关闭程序。这个按钮由原 VC 界面设计区默认内置的“取消”按钮更名而来,其代码在程序主对话框类的 OnCancel()方法中。

“退出”按钮的事件代码如下:

```

void ChttpSrverDlg::OnCancel()
{
    if (m_bStart)
    {
        pHttpProtocol->StopHttpSrv();
    }
    if(pHttpProtocol)
    {
        delete pHttpProtocol;
        pHttpProtocol = NULL;
    }
    m_bStart = false;
    CDialog::OnCancel();
}

```

关闭服务器程序主要是关掉其上正在运行的 Web 服务,同时销毁内存中的 HTTP 协议类。

以上就是服务器全部的界面总控代码,大家是不是觉得很简单?之所以简单是因为这部分代码是从表观层面上展示整个软件的控制流程,如在启动 Web 服务时仅用了一句“pHttpProtocol->StartHttpSrv()”,这看似简单的一句话却包含了表层之下好几层多达上百行代码的复杂处理过程。

5.3.3 Web 服务流程的实现

在服务器界面的总控代码中,大家可以清楚地看出其上运行的 Web 服务是如何在网管员的控制下被使用的。但网管员通过鼠标操作只能决定在何时开启、何时关闭服务,以及开启几个 Web 服务。服务被开启后就自主地运行,网络管理员无法再“干涉”一个服务的内部执行流程,原因是表层的总控代码只能通过 HTTP 协议类对象的指针 pHttpProtocol 调用 StartHttpSrv()方法开启服务,调用 StopHttpSrv()方法关闭服务,而服务自身的操作流程则封装于协议类 ChttpProtocol 的内部,外部代码无权访问。

由于一个服务器在实际应用场合要为成千上万甚至更多的客户端提供服务,其上的 Web 服务进程就必须具有与很多客户进程同时交互的能力,这就必须采用多线程和多 Socket 实现,为了在程序中使用线程机制,须在 httpSrverDlg.cpp 中包含:


```
#include <process.h>
```

除此之外，服务进程的流程是不能像普通程序那样完全由程序员确定的，它还必须符合 HTTP 协议所规定的交互时序（即第 1 章 1.2.1 节所说的网络协议三要素之一的“同步”）要求。故在 `HttpProtocol.cpp` 中添加头文件：

```
#include "httpSvrDlg.h"
```

这样就将界面总控代码与协议流程的实现代码关联起来了。以下代码均在源文件 `HttpProtocol.cpp` 中。

在协议类 `CHttpProtocol` 的构造方法中初始化服务器监听线程指针 `m_pListenThread` 和主对话框窗口句柄 `m_hwndDlg` 为空：

```
CHttpProtocol::CHttpProtocol(void)
{
    m_pListenThread = NULL;
    m_hwndDlg = NULL;
}
CHttpProtocol::~CHttpProtocol(void){}
```

一切准备就绪，就可以编写实现 Web 服务流程的代码了，图 5.38 画出了本例 Web 服务的整个流程。想知道网管员操作启动/关闭服务的 `StartHttpSrv()/StopHttpSrv()` 方法都干了些什么吗？请接着往下看。

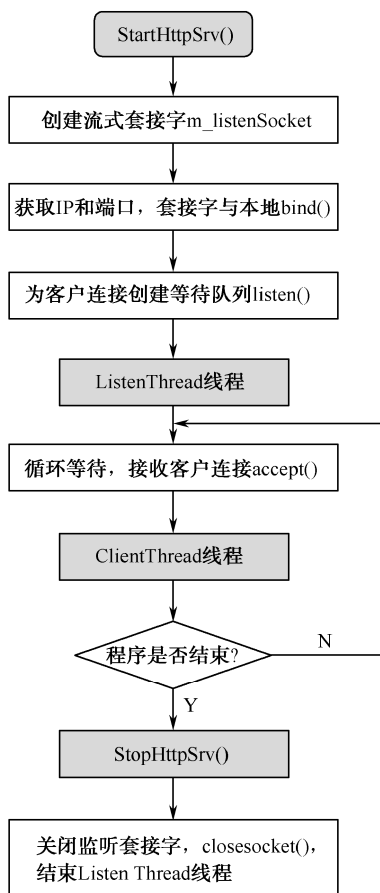


图 5.38 Web 服务流程

StartHttpSrv()方法代码如下:

```
bool CHttpProtocol::StartHttpSrv()
{
    WORD wVersionRequested = WINSOCK_VERSION;
    WSADATA wsaData;
    int nRet;
    //启动 Winsock
    nRet = WSAStartup(wVersionRequested, &wsaData);           //加载成功返回
    if (nRet)
    {
        //错误处理
        AfxMessageBox("Initialize WinSock Failed");
        return false;
    }
    //检测版本
    if (wsaData.wVersion != wVersionRequested)
    {
        //错误处理
        AfxMessageBox("Wrong WinSock Version");
        return false;
    }
    m_hExit = CreateEvent(NULL, TRUE, FALSE, NULL);
    if (m_hExit == NULL)
    {
        return false;
    }
    //创建套接字
    m_listenSocket = WSASocket(AF_INET, SOCK_STREAM,
                                IPPROTO_TCP, NULL, 0, WSA_FLAG_OVERLAPPED);
    if (m_listenSocket == INVALID_SOCKET)
    {
        //异常处理
        CString *pStr = new CString;
        *pStr = "Could not create listen socket";
        SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
        return false;
    }
    SOCKADDR_IN sockAddr;
    LPSEVENT lpServEnt;
    if (m_nPort != 0)                                           //本例中端口由用户指定, 属此情形
    {
        //从主机字节顺序转为网络字节顺序赋给 sin_port
        sockAddr.sin_port = htons(m_nPort);
    }
    else
    {
        //获取已知 http 服务的端口, 该服务在 tcp 协议下注册
```

```

    lpServEnt = getservbyname("http", "tcp");
    if (lpServEnt != NULL)
    {
        sockAddr.sin_port = lpServEnt->s_port;
    }
    else
    {
        sockAddr.sin_port = htons(HTTPPORT);           //默认端口 HTTPPORT=80
    }
}
sockAddr.sin_family = AF_INET;
BYTE nFild[4];
CString sIP;
((ChtpSrverDlg*)(AfxGetApp()->m_pMainWnd))->LocalIP.GetAddress(
                                                                    nFild[0],nFild[1],nFild[2], nFild[3]);
sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
sockAddr.sin_addr.S_un.S_addr = inet_addr(sIP);
//初始化 content-type 和文件后缀对应关系的 map
CreateTypeMap();
//套接字绑定
nRet = bind(m_listenSocket, (LPSOCKADDR)&sockAddr, sizeof(struct sockaddr));
if (nRet == SOCKET_ERROR)
{
    //绑定发生错误
    CString *pStr = new CString;
    *pStr = "bind() error";
    SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
    closesocket(m_listenSocket);           //断开连接
    return false;
}
//套接字监听。为客户连接创建等待队列，队列最大长度为 SOMAXCONN
nRet = listen(m_listenSocket, SOMAXCONN);
if (nRet == SOCKET_ERROR)
{
    //异常处理
    CString *pStr = new CString;
    *pStr = "listen() error";
    SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
    closesocket(m_listenSocket);           //断开连接
    return false;
}
//创建 listening 线程，等待接收客户端连接要求
m_pListenThread = AfxBeginThread(ListenThread, this);
if (!m_pListenThread)
{
    //线程创建失败
    CString *pStr = new CString;

```

```

        *pStr = "Could not create listening thread" ;
        SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
        closesocket(m_listenSocket);                //断开连接
        return false;
    }
    CString strTemp;
    char hostname[255];
    gethostname(hostname, sizeof(hostname));
    //显示 Web 服务器正在启动
    CString *pStr = new CString;
    *pStr = "***** httpSrver(WebServer) is Starting now! *****";
    SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
    //显示 Web 服务器的信息, 包括主机名、IP 及端口号
    CString *pStr1 = new CString;
    pStr1->Format("%s", hostname);
    *pStr1 = *pStr1 + "[" + sIP + "]" + "    Port ";
    strTemp.Format("%d", htons(sockAddr.sin_port));
    *pStr1 = *pStr1 + strTemp;
    SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr1, NULL);
    return true;
}

```

整个过程还是比较清楚的, 其实就是普通 Socket 程序的流程:

WSAStartup()→WSASocket()→获取 IP→bind()→listen()

所不同的只是这里是专门开一个线程 ListenThread 用于监听。

StopHttpSrv()方法代码如下:

```

void CHttpProtocol::StopHttpSrv()
{
    int nRet;
    SetEvent(m_hExit);
    nRet = closesocket(m_listenSocket);                //关闭监听套接字
    //关闭监听线程
    nRet = WaitForSingleObject((HANDLE)m_pListenThread, 10000);
    if (nRet == WAIT_TIMEOUT)
    {
        CString *pStr = new CString;
        *pStr = "TIMEOUT waiting for ListenThread";
        SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
    }
    CloseHandle(m_hExit);
    CString *pStr1 = new CString;
    *pStr1 = "Server Stopped";
    SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr1, NULL);
}

```

关闭 Web 服务实质上就是先后关闭监听套接字 m_listenSocket 和监听线程 ListenThread。

以上步骤依旧属于通用 Socket 编程的范畴。

为了使管理员能够实时监控到服务的运行状况, 上面两个方法的代码中都使用了系统内置的

SendMessage 函数及时向前端传递日志信息。这些信息反映了程序运行过程中每一步的执行情况、异常故障等,是利用 5.3.1 节已经建立好的事件驱动机制发送的。通用的方法是先将要发送的信息字符串赋给 CString 类指针 pStr, 再在 LOG_MSG 映射下执行 AddLog()方法, 通过窗口句柄 m_hwndDlg 返回前端主界面。

实现这个过程的代码段如下:

```
if (nRet == XXXX)                                //返回异常码
{
    CString *pStr = new CString;
    *pStr = "****";                                //待显示信息
    SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
}
```

在服务器创建套接字并启动监听线程后, 接下来的任务就放手交给监听线程去完成。

监听 ListenThread 线程代码如下:

```
UINT CHttpProtocol::ListenThread(LPVOID param)
{
    CHttpProtocol *pHttpProtocol = (CHttpProtocol *)param;
    SOCKET          socketClient;
    CWinThread*     pClientThread;
    SOCKADDR_IN     SockAddr;
    PREQUEST        pReq;
    int              nLen;
    DWORD           dwRet;
    //初始化 ClientNum, 创建 “no client” 事件对象
    HANDLE          hNoClients;
    hNoClients = pHttpProtocol->InitClientCount();
    while(1)    //循环等待, 若有客户连接请求, 则接收客户端连接要求
    {
        nLen = sizeof(SOCKADDR_IN);
        //套接字等待连接, 返回对应已接收的客户端连接的套接字
        socketClient = accept(pHttpProtocol->m_listenSocket, (LPSOCKADDR)& SockAddr, &nLen);
        if (socketClient == INVALID_SOCKET)
        {
            break;
        }
        //将客户端网络地址转换为用点分割的 IP 地址
        CString *pstr = new CString;
        pstr->Format("%s Connecting on socket:%d", inet_ntoa(SockAddr.sin_addr), socketClient);
        SendMessage(pHttpProtocol->m_hwndDlg, LOG_MSG, (UINT)pstr, NULL);
        pReq = new REQUEST;
        if (pReq == NULL)
        {
            //处理错误
            CString *pStr = new CString;
            *pStr = "No memory for request";
            SendMessage(pHttpProtocol->m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
            continue;
        }
    }
}
```

```

    }
    //获取这个连接的客户端信息
    pReq->hExit = pHttpProtocol->m_hExit;
    pReq->Socket = socketClient;
    pReq->hFile = INVALID_HANDLE_VALUE;
    pReq->dwRecv = 0;
    pReq->dwSend = 0;
    pReq->pHttpProtocol = pHttpProtocol;
    //创建 client 进程, 处理 request
    pClientThread = AfxBeginThread(ClientThread, pReq);
    if (!pClientThread)
    {
        //线程创建失败, 错误处理
        CString *pStr = new CString;
        *pStr = "Couldn't start client thread";
        SendMessage(pHttpProtocol->m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
        delete pReq;
    }
} //while()循环结束
//等待线程结束
WaitForSingleObject((HANDLE)pHttpProtocol->m_hExit, INFINITE);
//等待所有 client 线程结束
dwRet = WaitForSingleObject(hNoClients, 5000);
if (dwRet == WAIT_TIMEOUT)
{
    //超时返回, 并且同步对象未退出
    CString *pStr = new CString;
    *pStr = "One or more client threads did not exit";
    SendMessage(pHttpProtocol->m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
}
pHttpProtocol->DeleteClientCount();
return 0;
}

```

代码结构也是很清晰的, 监听线程 ListenThread 的执行流程为:

初始化 ClientNum→获取连接的客户端信息→创建客户线程 ClientThread→删除 ClientCount
其中第一步和最后一步即初始化和删除 ClientCount 的过程分别如下。

InitClientCount():

```

HANDLE CHttpProtocol::InitClientCount()
{
    ClientNum = 0;
    //创建 “no client” 事件对象
    None = CreateEvent(NULL, TRUE, TRUE, NULL);
    return None;
}

DeleteClientCount():
void CHttpProtocol::DeleteClientCount()

```

```

{
    CloseHandle(None);
}

```

这两段代码大家一定觉得很难理解，尤其是其中涉及的“事件对象”的概念，还有不明用途的 `CreateEvent()` 与 `CloseHandle()` 函数。要告诉读者的是，这几个东西都涉及 Winsock 编程较为深入的部分，要明白套接字的 I/O 模型和 Windows 编程的机制后才有可能真正理解它们——但这些内容已超出了本书的范围，暂不进行介绍。不过读者仍然可以大致地从感性上认识这两段代码可能是用来给建立连接的客户端计数的，即管理客户端的 `Socket`——的确是这样。

监听线程从 5.3.1 节定义的 `REQUEST` 结构中获取连接客户端的信息，并创建 `Client` 线程进行处理，服务器会为每一个与之连接的客户端建立一个专门的线程 `ClientThread`，这个线程将按照 HTTP 协议的规范与客户端交互，为用户提供 Web 服务。

Web 服务工作的流程就是这样的，并不难。

5.3.4 HTTP 协议的实现

到目前为止，大家已经看完了整个 Web 服务流程的完整实现代码，但我们仍然没看到 Web 服务所依赖的网络协议 HTTP 究竟是怎么实现的。读者马上就会看到，从客户线程 `ClientThread` 开始，如图 5.39 所示，将遵照 HTTP 协议所规定的标准去处理 HTTP 请求报文。

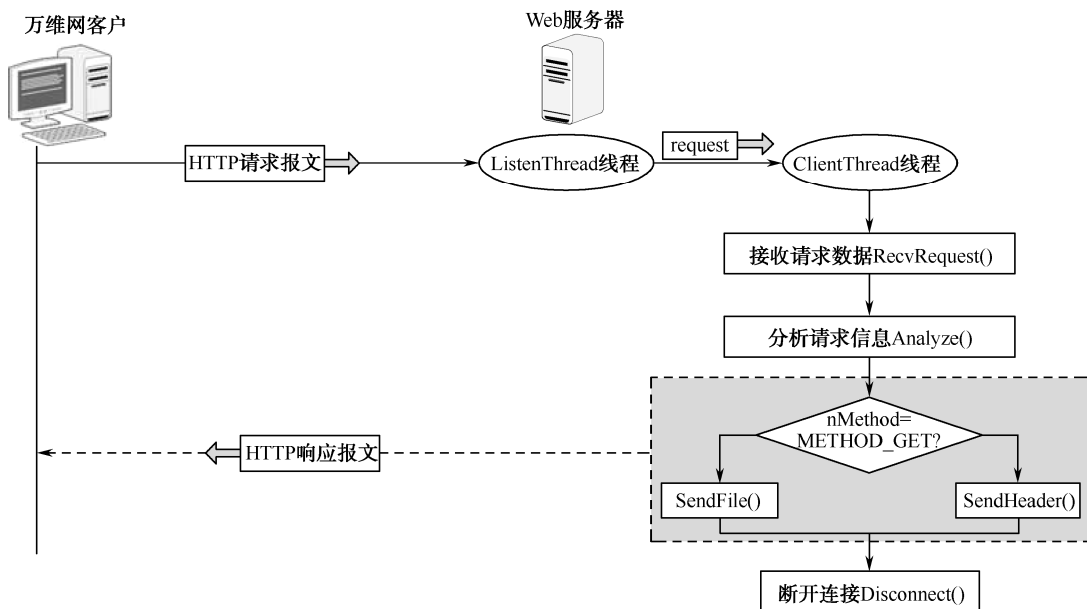


图 5.39 HTTP 请求报文的处理过程

客户 `ClientThread` 线程代码如下：

```

UINT CHttpProtocol::ClientThread(LPVOID param)
{
    int nRet;
    BYTE buf[1024];
    PREQUEST pReq = (PREQUEST)param;
    CHttpProtocol *pHttpProtocol = (CHttpProtocol *)pReq->pHttpProtocol;

```

```

pHttpProtocol->CountUp();                //计数
//接收请求(request)数据
if (!pHttpProtocol->RecvRequest(pReq, buf, sizeof(buf)))
{
    pHttpProtocol->Disconnect(pReq);
    delete pReq;
    pHttpProtocol->CountDown();
    return 0;
}
//分析请求(request)信息
nRet = pHttpProtocol->Analyze(pReq, buf);
if (nRet)
{
    //处理错误
    CString *pStr = new CString;
    *pStr = "Error occurs when analyzing client request";
    SendMessage(pHttpProtocol->m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
    pHttpProtocol->Disconnect(pReq);
    delete pReq;
    pHttpProtocol->CountDown();
    return 0;
}
//生成并返回头部
pHttpProtocol->SendHeader(pReq);
//向客户端传送数据
if(pReq->nMethod == METHOD_GET)    //请求的使用方法: GET
{
    pHttpProtocol->SendFile(pReq);
}
pHttpProtocol->Disconnect(pReq);
delete pReq;
pHttpProtocol->CountDown();                //客户端数量减 1
return 0;
}

```

这段代码显然是按照 HTTP 协议所规定的流程编写的,从程序中调用各个方法的时序可以明显地看出来。

对应于各方法的调用,客户线程的执行步骤为:

客户端数目加 1 (CountUp())→接收请求 (request) 数据 (RecvRequest())→分析请求 (request) 信息 (Analyze())→返回响应信息的头部 (SendHeader())或向客户端传送数据 (SendFile())→……→断开与客户端的连接 (Disconnect())→客户端数目减 1 (CountDown())

——这也正是 5.1.1 节所介绍的 HTTP 协议的典型工作流程。

此处的 request 信息是客户端发送给服务器的请求,也就是前面监听线程从 REQUEST 结构中获取并交由客户线程 ClientThread 做进一步处理的客户端信息,其中包含了客户端要求服务器所提供的服务。程序在执行向客户端传送数据 (SendFile()) 之前,直接对这一信息按协议标准进行了解析,根据 REQUEST 结构中 nMethod 字段来判断客户端要求什么样的服务。具体到本

例中就是判断“pReq->nMethod == METHOD_GET”是否成立，若是，则表示 Http 报文“请求方法”字段为“GET”，说明客户端想下载服务器上的网页，于是服务器就用 SendFile()方法向客户端传这个网页的 html 源文件，浏览器收到后将 html 源文件解析出来显示，就成了我们平时上网看到的绚丽多彩的 Web 页。

下面来细看一下这段程序中调用的方法。

CountUp()代码如下：

```
void CHttpProtocol::CountUp()
{
    //进入临界区
    m_critSect.Lock();
    ClientNum++;
    //离开临界区
    m_critSect.Unlock();
    //重置为无信号事件对象
    ResetEvent(None);
}
```

前面说过，服务器管理众多 Socket 使用的是 Winsock 的一种深入的编程机制，即“套接字 I/O 模型”，它与大家在操作系统课程中学过的进程管理机制相似，也要用到“临界区”的概念。以下是临界区的初始化代码（在 HttpProtocol.cpp 中）：

```
UINT CHttpProtocol::ClientNum = 0;
CCriticalSection CHttpProtocol::m_critSect; //临界区初始化
HANDLE CHttpProtocol::None = NULL;
RecvRequest():
bool CHttpProtocol::RecvRequest(PREQUEST pReq, LPBYTE pBuf, DWORD dwBufSize)
{
    WSABUF          wsabuf;           //发送/接收缓冲区结构
    WSAOVERLAPPED   over;             //指向调用重叠操作时指定的 WSAOVERLAPPED 结构
    DWORD           dwRecv;
    DWORD           dwFlags;
    DWORD           dwRet;
    HANDLE          hEvents[2];
    bool            fPending;
    int             nRet;
    memset(pBuf, 0, dwBufSize);       //初始化缓冲区
    wsabuf.buf      = (char *)pBuf;
    wsabuf.len      = dwBufSize;       //缓冲区的长度
    over.hEvent     = WSACreateEvent(); //创建一个新的事件对象
    dwFlags = 0;
    fPending = FALSE;
    //接收数据
    nRet = WSARecv(pReq->Socket, &wsabuf, 1, &dwRecv, &dwFlags, &over, NULL);
    if (nRet != 0)
    {
        //错误代码 WSA_IO_PENDING 表示重叠操作成功启动
        if (WSAGetLastError() != WSA_IO_PENDING)
```

```

        {
            //重叠操作未能成功
            CloseHandle(over.hEvent);
            return false;
        }
        else
        {
            fPending = true;
        }
    }
    if (fPending)
    {
        hEvents[0] = over.hEvent;
        hEvents[1] = pReq->hExit;
        dwRet = WaitForMultipleObjects(2, hEvents, FALSE, INFINITE);
        if (dwRet != 0)
        {
            CloseHandle(over.hEvent);
            return false;
        }
        //重叠操作未完成
        if (!WSAGetOverlappedResult(pReq->Socket, &over, &dwRecv, FALSE, &dwFlags))
        {
            CloseHandle(over.hEvent);
            return false;
        }
    }
    pReq->dwRecv += dwRecv;           //统计接收数量
    CloseHandle(over.hEvent);
    return true;
}

```

可以看出, 接收客户端发来的 **request** 信息数据使用的仍然是套接字, 就是调用最基本的 Winsock `WSARecv()` 函数, 所以说, 套接字 **Socket** 编程是网络编程的基础。

成功收到客户的请求信息之后, 就要按照 **HTTP** 协议规范的定义对其进行分析, 这是网络协议得以最终实现的关键。

Analyze() 代码如下:

```

//分析 request 信息
int CHttpRequest::Analyze(PREQUEST pReq, LPBYTE pBuf)
{
    //分析接收到的信息
    char szSeps[] = "\n";
    char *cpToken;
    //防止非法请求
    if (strstr((const char *)pBuf, "..") != NULL)
    {
        strcpy(pReq->StatusCodeReason, HTTP_STATUS_BADREQUEST);
    }
}

```

```

        return 1;
    }
    //判断 request 的 method ①
    cpToken = strtok((char *)pBuf, szSeps);    //缓存中字符串分解为一组标记串
    if (!_strcmp(cpToken, "GET"))              //GET 命令
    {
        pReq->nMethod = METHOD_GET;
    }
    else if (!_strcmp(cpToken, "HEAD"))        //HEAD 命令
    {
        pReq->nMethod = METHOD_HEAD;
    }
    else
    {
        strcpy(pReq->StatusCodeReason, HTTP_STATUS_NOTIMPLEMENTED);
        return 1;
    }
    //获取 Request-URL ②
    cpToken = strtok(NULL, szSeps);
    if (cpToken == NULL)
    {
        strcpy(pReq->StatusCodeReason, HTTP_STATUS_BADREQUEST);
        return 1;
    }
    strcpy(pReq->szFileName, m_strRootDir);
    if (strlen(cpToken) > 1)
    {
        strcat(pReq->szFileName, cpToken);    //把该文件名添加到结尾处形成路径
    }
    else
    {
        strcat(pReq->szFileName, "/index.html");
    }
    return 0;
}

```

这段代码完全是按照 HTTP 协议的规范来编写的，对收到的 request 信息字符串，用系统内置的字符串处理 strtok() 函数进行拆分，按 HTTP 规定的 request 请求报文的格式（见本章 5.1.2 节），解析出第一行的“请求方法”字段和“URL”字段，分两部分进行处理和使用，如下所示。

（1）判断请求方法的类型。

若为“GET”，表示是客户端请求下载指定的 Web 网页，将 REQUEST 结构的 nMethod（请求的使用方法）字段置为 METHOD_GET，以便客户 ClientThread 线程据此识别出客户端是要下载网页。

若为“HEAD”，表示客户端用户只是想知道指定超文本链接的正确性、可访问性和最近是否进行了修改等信息，将 REQUEST 结构的 nMethod 字段置为 METHOD_HEAD，客户 ClientThread 线程识别出用户只是想获得有关某网页的信息而并不需要下载整个网页，于是只向

客户端返回该网页文档的首部信息（文档头）。

此外，读者还可以自己写代码让程序能识别出 POST、PUT、DELETE、OPTIONS、TRACE 等 HTTP 协议中定义的其他命令，并编程实现它们对应的功能，从而扩展这个 Web 服务器程序的能力。

（2）将 URL 转换为服务器上的本地文件路径。

先将管理员在界面上配置的文件的相对路径（即“根目录”文本框中填写的内容）赋给 REQUEST 结构的 szFileName 字段。判断先前解析出的 URL，若指向某一具体的文件，则将该文件名添加到 szFileName 字段结尾处形成路径，否则在后面添加“/index.html”作为默认访问页。

程序中还使用了协议规定的 HTTP 响应码来捕获出错信息，这些响应码及其说明通过 5.3.1 节 HttpProtocol.h 文件开头列出的一系列宏定义引用。本段代码中用到的两个响应码的宏对应的字符串及中文说明分别为：

HTTP_STATUS_BADREQUEST——“400 Bad Request”（客户端差错，错误的请求）

HTTP_STATUS_NOTIMPLEMENTED ——“501 Not Implemented”（服务器差错，没有实现）

更多响应码的含义见 5.1.2 节的表 5.3。

服务器向客户端返回网页文档头用的是 SendHeader()方法。

SendHeader()方法代码如下：

```
//发送头部
void CHttpProtocol::SendHeader(PREQUEST pReq)
{
    int n = FileExist(pReq);
    if(!n)                //文件不存在，则返回
    {
        return;
    }
    char Header[2048] = " ";
    char curTime[50] = " ";
    GetCurrentTime((char*)curTime);
    //取得文件长度
    DWORD length;
    length = GetFileSize(pReq->hFile, NULL);
    //取得文件的 last-modified 时间
    char last_modified[60] = " ";
    GetLastModified(pReq->hFile, (char*)last_modified);
    //取得文件的类型
    char ContentType[50] = " ";
    GetContentType(pReq, (char*)ContentType);
    sprintf((char*)Header, "HTTP/1.0 %s\r\nDate: %s\r\nServer: %s\r\n
        Content-Type: %s\r\nContent-Length: %d\r\nLast-Modified: %s\r\n\r\n",
        HTTP_STATUS_OK,
        curTime,                // Date
        "My Http Server",      // Server
        ContentType,           // Content-Type
        length,                 // Content-length
        last_modified);         // Last-Modified
    //发送头部
```

```

        send(pReq->Socket, Header, strlen(Header), 0);
    }

```

先调用 `FileExist()` 函数判断用户请求的文件是否在服务器上，然后用 `GetCurrentTime()` 获取系统当前时间（为生成日志需要），用 `GetFileSize()` 取得文件的长度，再分别从 `REQUEST` 结构的 `hFile`（请求连接的文件）字段中获得文件上次修改时间（用 `GetLastModified()` 方法）和文件类型（用 `GetContentType()` 方法），最后将这些信息组装成文档头并调用 `Socket` 函数 `send()` 发送出去。

假如用户向服务器发出的是“GET”命令（索取特定的网页），则服务器在响应中除了要返回文档头之外还要用 `SendFile()` 方法向客户端传送数据。

`SendFile()` 方法代码如下：

```

//发送文件
void CHttpProtocol::SendFile(PREQUEST pReq)
{
    int n = FileExist(pReq);
    if(!n)        //文件不存在，则返回
    {
        return;
    }
    CString *pStr = new CString;
    *pStr = *pStr + &pReq->szFileName[strlen(m_strRootDir)];
    SendMessage(m_hwndDlg, LOG_MSG, UINT(pStr), NULL);
    static BYTE    buf[2048];
    DWORD          dwRead;
    BOOL            fRet;
    int flag = 1;
    //读写数据，直到完成
    while(1)
    {
        //从文件中读入 buffer 中
        fRet = ReadFile(pReq->hFile, buf, sizeof(buf), &dwRead, NULL);
        if (!fRet)
        {
            static char szMsg[512];
            wsprintf(szMsg, "%s", HTTP_STATUS_SERVERERROR);
            //向客户端发送出错信息
            send(pReq->Socket, szMsg, strlen(szMsg), 0);
            break;
        }
        //完成
        if (dwRead == 0)
        {
            break;
        }
        //将 buffer 内容传送给客户端
        if (!SendBuffer(pReq, buf, dwRead))
        {
            break;
        }
    }
}

```

```

    }
    pReq->dwSend += dwRead;
}
//关闭文件
if (CloseHandle(pReq->hFile))
{
    pReq->hFile = INVALID_HANDLE_VALUE;
}
else
{
    CString *pStr = new CString;
    *pStr = "Error occurs when closing file";
    SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
}
}

```

服务器是先把网页源码从磁盘文件中读入 `buffer` 中，再将 `buffer` 内容传送给客户端的。这里调用了 `SendBuffer()` 函数，它是实现 HTTP 协议的辅助函数，其详细的代码稍后会列出。

服务完成，连接会关闭。

`Disconnect()` 方法代码如下：

```

void CHttpProtocol::Disconnect(PREQUEST pReq)
{
    //关闭套接字：释放所占有的资源
    int nRet;
    CString *pStr = new CString;
    pStr->Format("Closing socket: %d", pReq->Socket);
    SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
    nRet = closesocket(pReq->Socket);
    if (nRet == SOCKET_ERROR)
    {
        //处理错误
        CString *pStr1 = new CString;
        pStr1->Format("closesocket() error: %d", WSAGetLastError() );
        SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr1, NULL);
    }
    HTTPSTATS stats;
    stats.dwRecv = pReq->dwRecv;
    stats.dwSend = pReq->dwSend;
    SendMessage(m_hwndDlg, DATA_MSG, (UINT)&stats, NULL);
}

```

连接关闭后，客户端计数减 1。

`CountDown()` 方法代码如下：

```

void CHttpProtocol::CountDown()
{
    //进入排斥区
    m_critSect.Lock();
    if (ClientNum > 0)

```

```

    {
        ClientNum--;
    }
    //离开排斥区
    m_critSect.Unlock();
    if(ClientNum < 1)
    {
        //重置为有信号事件对象
        SetEvent(None);
    }
}

```

至此，大家已经看到了一个实际网络协议（HTTP）的实现过程（虽然只实现了本例需要用到的功能），对网络协议的理解又加深了一步。

5.3.5 协议实现的辅助代码

HTTP 协议的主要实现代码已经介绍完了，这是大家第一次接触实际应用中的网络协议。一定会有人发现在上面协议的实现程序中调用了不少功能函数，如判断服务器上是否有用户需要的网页文件（FileExist()）、取得文件的类型（GetContentType()）、取得文件的上次修改时间（GetLastModified()）等。这些函数是对协议实现过程中某一步的具体操作做的细节上的处理，无关乎协议所约定的标准流程，只要不改变它们的名称和调用方式，具体实现代码的改变就不会影响网络协议的实现。它们的实现算法和机制都与网络编程的关系不大，此处不做详述，只罗列出全部的代码供读者参考。

FileExist()函数代码如下：

```

int CHttpProtocol::FileExist(PREQUEST pReq)
{
    pReq->hFile = CreateFile(pReq->szFileName, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_
EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    //如果文件不存在，则返回出错信息
    if (pReq->hFile == INVALID_HANDLE_VALUE)
    {
        strcpy(pReq->StatuCodeReason, HTTP_STATUS_NOTFOUND);
        return 0;
    }
    else
    {
        return 1;
    }
}

```

GetCurentTime()函数代码如下：

```

//活动本地时间
void CHttpProtocol::GetCurentTime(LPSTR lpszString)
{
    //活动本地时间
    SYSTEMTIME st;

```

```

GetLocalTime(&st);
//时间格式化
wsprintf(lpszString, "%s %02d %s %d %02d:%02d:%02d GMT",
        week[st.wDayOfWeek], st.wDay, month[st.wMonth-1],
        st.wYear, st.wHour, st.wMinute, st.wSecond);
}

```

这个函数获取的时间信息是显示服务器运行日志用的，统一采用格林尼治标准时间，为此还要在 `HttpProtocol.cpp` 源文件中定义星期和月份的转换表，代码如下：

```

//格林尼治时间的星期转换
char *week[] = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "" };
//格林尼治时间的月份转换
char *month[] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec", "" };
GetLastModified()函数代码如下：

```

```

bool CHttpProtocol::GetLastModified(HANDLE hFile, LPSTR lpszString)
{
    //获得文件的 last-modified 时间
    FILETIME ftCreate, ftAccess, ftWrite;
    SYSTEMTIME stCreate;
    FILETIME ftime;
    //获得文件的 last-modified 的 UTC 时间
    if (!GetFileTime(hFile, &ftCreate, &ftAccess, &ftWrite))
        return false;
    FileTimeToLocalFileTime(&ftWrite, &ftime);
    //UTC 时间转化成本地时间
    FileTimeToSystemTime(&ftime, &stCreate);
    //时间格式化
    wsprintf(lpszString, "%s %02d %s %d %02d:%02d:%02d GMT", week[stCreate.wDayOfWeek],
    stCreate.wDay, month[stCreate.wMonth-1], stCreate.wYear, stCreate.wHour, stCreate.wMinute, stCreate.wSecond);
}

```

GetContentType()函数代码如下：

```

bool CHttpProtocol::GetContentType(PREQUEST pReq, LPSTR type)
{
    //取得文件的类型
    CString cpToken;
    cpToken = strstr(pReq->szFileName, ".");
    strcpy(pReq->postfix, cpToken); // “pReq->postfix” 存储的是文件扩展名
    //遍历搜索该文件类型对应的 content-type
    map<CString, char *>::iterator it = m_typeMap.find(pReq->postfix);
    if(it != m_typeMap.end())
    {
        wsprintf(type, "%s", (*it).second);
    }
    return TRUE;
}

```

为了使客户端能够浏览服务器上多种类型的文件（html、图片、Word 文档、PDF 电子书等），必须提供对计算机上不同文件类型的识别机制。本例使用了类似 Java 语言的迭代器（Iterator）

机制，将客户程序映射至一个“容器”（map），“容器”中存储了众多不同种类的文件后缀。还记得在 Web 服务流程代码的方法 StartHttpSrv() 中有这样一个函数调用“CreateTypeMap()”吗？当时没有说明，它就是用来初始化映射（map）的，而 map 中保存了 content-type 和文件后缀的对应关系，在 HttpProtocol.cpp 中有函数 CreateTypeMap() 的代码。

CreateTypeMap() 函数代码如下：

```
void CHttpProtocol::CreateTypeMap()
{
    //初始化 map
    m_typeMap[".doc"] = "application/msword";
    m_typeMap[".bin"] = "application/octet-stream";
    m_typeMap[".dll"] = "application/octet-stream";
    m_typeMap[".exe"] = "application/octet-stream";
    m_typeMap[".pdf"] = "application/pdf";
    m_typeMap[".ai"] = "application/postscript";
    m_typeMap[".eps"] = "application/postscript";
    m_typeMap[".ps"] = "application/postscript";
    m_typeMap[".rtf"] = "application/rtf";
    m_typeMap[".fdf"] = "application/vnd.fdf";
    m_typeMap[".arj"] = "application/x-arj";
    m_typeMap[".gz"] = "application/x-gzip";
    m_typeMap[".class"] = "application/x-java-class";
    m_typeMap[".js"] = "application/x-javascript";
    m_typeMap[".lzh"] = "application/x-lzh";
    m_typeMap[".lnk"] = "application/x-ms-shortcut";
    m_typeMap[".tar"] = "application/x-tar";
    m_typeMap[".hlp"] = "application/x-winhelp";
    m_typeMap[".cert"] = "application/x-x509-ca-cert";
    m_typeMap[".zip"] = "application/zip";
    m_typeMap[".cab"] = "application/x-compressed";
    m_typeMap[".arj"] = "application/x-compressed";
    m_typeMap[".aif"] = "audio/aiff";
    m_typeMap[".aifc"] = "audio/aiff";
    m_typeMap[".aiff"] = "audio/aiff";
    m_typeMap[".au"] = "audio/basic";
    m_typeMap[".snd"] = "audio/basic";
    m_typeMap[".mid"] = "audio/midi";
    m_typeMap[".rmi"] = "audio/midi";
    m_typeMap[".mp3"] = "audio/mpeg";
    m_typeMap[".vox"] = "audio/voxware";
    m_typeMap[".wav"] = "audio/wav";
    m_typeMap[".ra"] = "audio/x-pn-realaudio";
    m_typeMap[".ram"] = "audio/x-pn-realaudio";
    m_typeMap[".bmp"] = "image/bmp";
    m_typeMap[".gif"] = "image/gif";
    m_typeMap[".jpeg"] = "image/jpeg";
    m_typeMap[".jpg"] = "image/jpeg";
}
```

```

m_typeMap[".tif"]    = "image/tiff";
m_typeMap[".tiff"]   = "image/tiff";
m_typeMap[".xbm"]    = "image/xbm";
m_typeMap[".wrl"]    = "model/vrml";
m_typeMap[".htm"]   = "text/html";
m_typeMap[".html"]  = "text/html";
m_typeMap[".c"]      = "text/plain";
m_typeMap[".cpp"]    = "text/plain";
m_typeMap[".def"]    = "text/plain";
m_typeMap[".h"]      = "text/plain";
m_typeMap[".txt"]    = "text/plain";
m_typeMap[".rtx"]    = "text/richtext";
m_typeMap[".rtf"]    = "text/richtext";
m_typeMap[".java"]   = "text/x-java-source";
m_typeMap[".css"]    = "text/css";
m_typeMap[".mpeg"]   = "video/mpeg";
m_typeMap[".mpg"]    = "video/mpeg";
m_typeMap[".mpe"]    = "video/mpeg";
m_typeMap[".avi"]    = "video/msvideo";
m_typeMap[".mov"]    = "video/quicktime";
m_typeMap[".qt"]     = "video/quicktime";
m_typeMap[".shtml"]  = "wwwserver/html-ssi";
m_typeMap[".asa"]    = "wwwserver/isapi";
m_typeMap[".asp"]    = "wwwserver/isapi";
m_typeMap[".cfm"]    = "wwwserver/isapi";
m_typeMap[".dbm"]    = "wwwserver/isapi";
m_typeMap[".isa"]    = "wwwserver/isapi";
m_typeMap[".plx"]    = "wwwserver/isapi";
m_typeMap[".url"]    = "wwwserver/isapi";
m_typeMap[".cgi"]    = "wwwserver/isapi";
m_typeMap[".php"]    = "wwwserver/isapi";
m_typeMap[".wcgi"]   = "wwwserver/isapi";
}

```

可以看到,我们平时在计算机上常用的几乎所有的文件类型它都包含了!这就使得浏览器不仅可以访问服务器上的 Web 页,而且还能查看图片、Word 文档、PDF 电子书等其他各种类型的资源。

最后还有一段 SendBuffer()函数的代码。

SendBuffer()函数代码如下:

```

bool CHttpProtocol::SendBuffer(PREQUEST pReq, LPBYTE pBuf, DWORD dwBufSize)
{
    //发送缓存中的内容
    WSABUF          wsabuf;
    WSAOVERLAPPED   over;
    DWORD           dwRecv;
    DWORD           dwFlags;
    DWORD           dwRet;

```

```

HANDLE          hEvents[2];
BOOL            fPending;
int             nRet;
wsabuf.buf      = (char *)pBuf;
wsabuf.len      = dwBufSize;
over.hEvent     = WSACreateEvent();
fPending = false;
//发送数据
nRet = WSASend(pReq->Socket, &wsabuf, 1, &dwRecv, 0, &over, NULL);
if (nRet != 0)
{
    // 错误处理
    if (WSAGetLastError() == WSA_IO_PENDING)
    {
        fPending = true;
    }
    else
    {
        CString *pStr = new CString;
        pStr->Format("WSASend() error: %d", WSAGetLastError() );
        SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
        CloseHandle(over.hEvent);
        return false;
    }
}
if (fPending)    //I/O 未完成
{
    hEvents[0] = over.hEvent;
    hEvents[1] = pReq->hExit;
    dwRet = WaitForMultipleObjects(2, hEvents, FALSE, INFINITE);
    if (dwRet != 0)
    {
        CloseHandle(over.hEvent);
        return false;
    }
    //重叠操作未完成
    if (!WSAGetOverlappedResult(pReq->Socket, &over, &dwRecv, FALSE, &dwFlags))
    {
        //错误处理
        CString *pStr = new CString;
        pStr->Format("WSAGetOverlappedResult() error: %d", WSAGetLastError() );
        SendMessage(m_hwndDlg, LOG_MSG, (UINT)pStr, NULL);
        CloseHandle(over.hEvent);
        return false;
    }
}
CloseHandle(over.hEvent);

```

```
return true;  
}
```

它也是调用 Socket 接口的 WSARecv()向客户端发送数据的。至于这段代码中所涉及的很复杂的“重叠操作”机制已经超出了本书的范围,属于 Winsock 深入编程,读者看不懂也没有关系,这并不妨碍对整个 Web 服务器源代码功能的理解。

本例引领大家从网络管理员的角度,采用由表观操作到宏观流程再到协议实现最后才深入细看函数细节的方法,便于从整体上把握系统的结构、功能和原理。经过以上这一番梳理,相信大家一定都明白 Web 服务器究竟是如何实现的了。

5.4 自制浏览器访问 Web 服务器

本节来测试 5.3 节开发的 Web 服务器。我们把自己的计算机模拟为某一网站的 Web 服务器主机,在本地硬盘上建立目录,其中存放从网上下载(或自己收藏)的资源。用 5.2 节自制的浏览器 SelfBrowser 访问这些资源,最终揭示出大家平时上网浏览网页时浏览器与网站服务器交互协作的幕后秘密。

5.4.1 Web 资源准备

在 D 盘上新建一个文件夹,取名为“我的 Web 资源”,接下来的实验就以这个文件夹作为服务器的根目录,存放测试用的资源。访问南京师范大学网站(网址为 <http://www.njnu.edu.cn/>),将首页保存到刚刚建立的文件夹中,如图 5.40 所示。

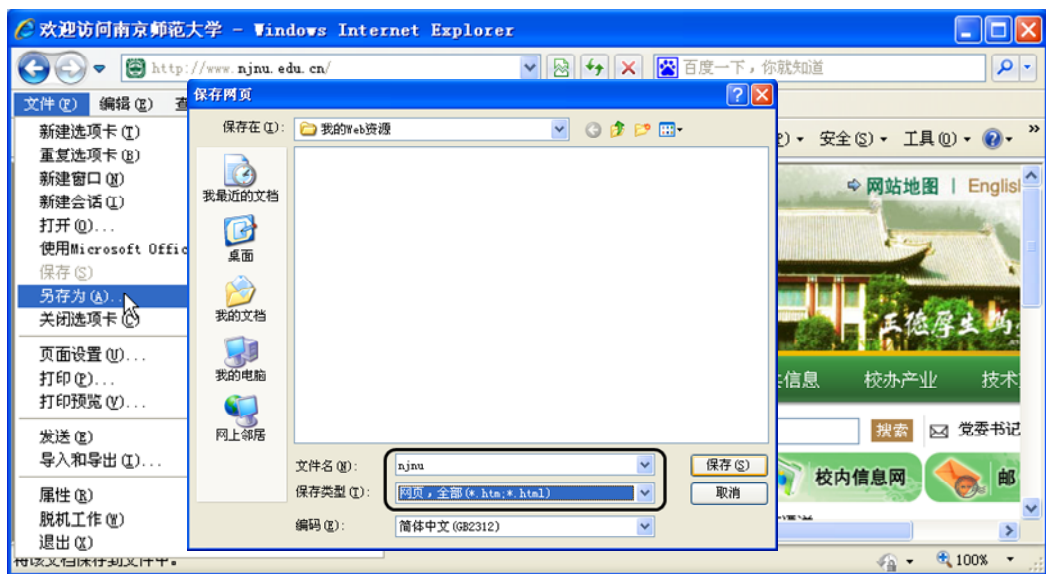


图 5.40 保存网页资源

注意: 在保存网页时要重命名为全英文的文件名,这里取名为“njnu”,选择保存类型为“网页,全部 (*.htm;*.html)”。之所以这么做是因为:本章开发的这个 Web 服务器只能处理字符串形式的文件名,还不支持中文文件名。若要使服务器程序支持中文文件名,则必须实现汉字处理功能,这会极大地增加代码量,而本例旨在讲清楚 Web 服务器和 HTTP 编程的基本原理,并不

是为了开发完善的服务器产品。

保存好网页之后,顺便将南京师范大学主页标题图片也下载保存起来,图片取默认文件名“toppic_main.jpg”,准备好的 Web 资源如图 5.41 所示。

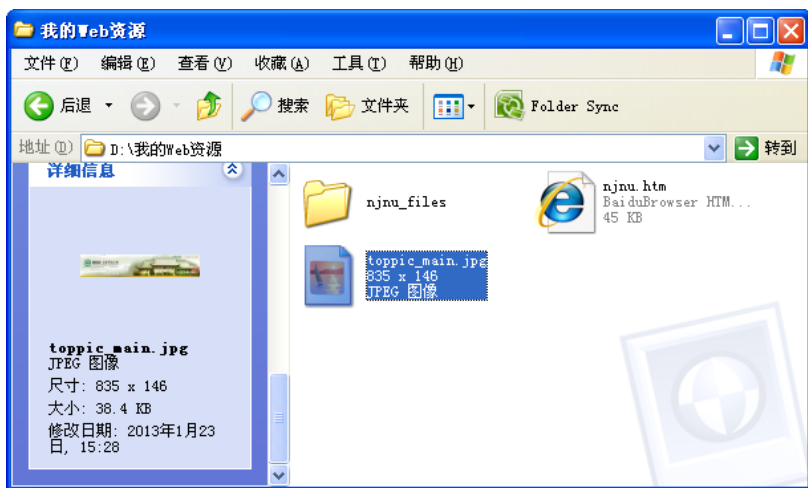


图 5.41 准备好的 Web 资源

5.4.2 访问 Web 服务器

1. 网页浏览

启动本章开发的 Web 服务器程序 httpSrver,配置服务器 IP 和端口,如设为 127.168.103.9:2258,在根目录栏填写“D:\我的 Web 资源”(就是放置测试资源的文件夹)。

单击“开启”按钮,启动服务器程序,如图 5.42 所示,服务器左边的列表区显示服务启动成功的信息。



图 5.42 启动服务器

运行 5.2 节开发的浏览器 SelfBrowser,在地址栏输入“http://127.168.103.9:2258/njnu.htm”(浏览器没有实现 DNS 域名解析功能,所以在访问服务器时采用直接给出 IP 的方式)后单击“浏览”按钮或直接回车,浏览器显示出先前我们准备的网页,如图 5.43 所示。



图 5.43 显示网页

留心一下服务器监控列表中的日志会发现：显示这个网页的过程中创建了多个 Socket，每下载完页面上的一个资源就关闭一个 Socket，当继续下载新资源时，又要用一个新建的 Socket 连接，如图 5.44 所示。

```

15:11:53.281 /njnu_files/more.gif
15:11:53.281 Closing socket: 576
15:11:53.281 /njnu_files/title2_right.gif
15:11:53.281 Closing socket: 572
15:11:53.296 127.168.103.9 Connecting on socket:476
15:11:53.296 127.168.103.9 Connecting on socket:428
15:11:53.296 127.168.103.9 Connecting on socket:560
15:11:53.296 /njnu_files/cr2.gif
15:11:53.296 Closing socket: 476
15:11:53.296 /njnu_files/xuxian2.gif
15:11:53.296 /njnu_files/crbg.gif
15:11:53.296 127.168.103.9 Connecting on socket:452
15:11:53.312 Closing socket: 428
15:11:53.312 Closing socket: 560
15:11:53.312 127.168.103.9 Connecting on socket:532
15:11:53.312 /njnu_files/cr1.gif
15:11:53.312 Closing socket: 452
15:11:53.312 /njnu_files/icon_main.gif
15:11:53.312 Closing socket: 532
15:11:54.265 127.168.103.9 Connecting on socket:592
15:11:54.265 /njnu_files/createsunxml.htm
15:11:54.265 Closing socket: 592
15:11:54.296 127.168.103.9 Connecting on socket:588
15:11:54.296 Closing socket: 588

```

图 5.44 HTTP 协议使用多个 Socket

一个 TCP 连接只作单个资源的下载之用，用完后服务器主动断开，这正好符合 5.1.1 节所介绍的万维网工作过程，我们把这样的 TCP 连接称为短连接，通常普通用户上网使用最多的就是这种连接。

2. 图片下载

在浏览器地址栏中输入“http://127.168.103.9:2258/toppic_main.jpg”后回车，显示先前保存的图片，如图 5.45 所示。

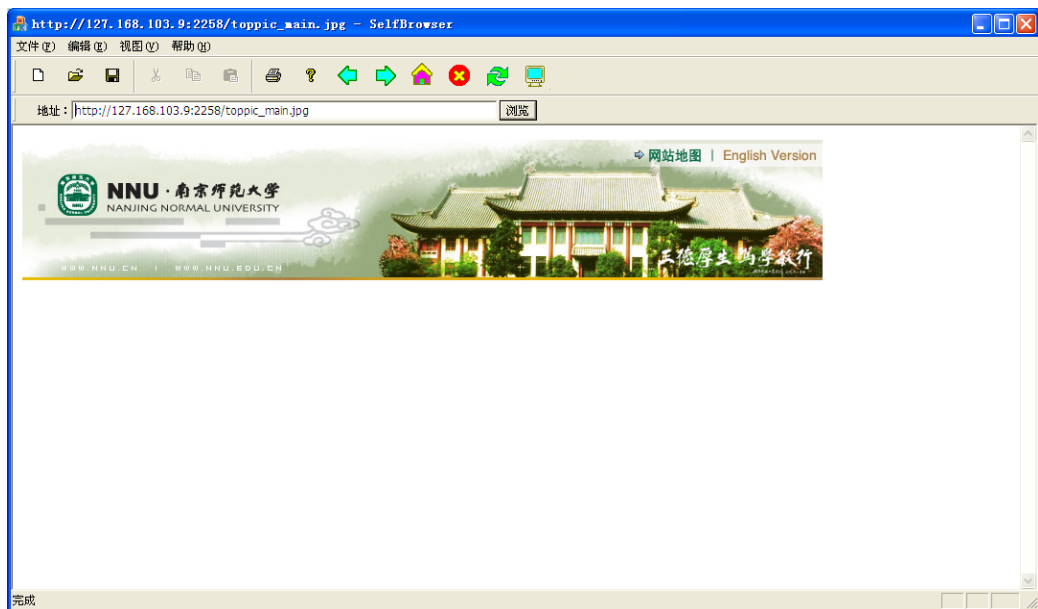


图 5.45 显示图片

从服务器显示的日志（见图 5.46）中看到，这张图片是由一个编号为 600 的 Socket 单独下载的。

```
15:11:53.281 Closing socket: 572
15:11:53.296 127.168.103.9 Connecting on socket:476
15:11:53.296 127.168.103.9 Connecting on socket:428
15:11:53.296 127.168.103.9 Connecting on socket:560
15:11:53.296 /njnu_files/cr2.gif
15:11:53.296 Closing socket: 476
15:11:53.296 /njnu_files/xuxian2.gif
15:11:53.296 /njnu_files/crbg.gif
15:11:53.296 127.168.103.9 Connecting on socket:452
15:11:53.312 Closing socket: 428
15:11:53.312 Closing socket: 560
15:11:53.312 127.168.103.9 Connecting on socket:532
15:11:53.312 /njnu_files/cr1.gif
15:11:53.312 Closing socket: 452
15:11:53.312 /njnu_files/icon_main.gif
15:11:53.312 Closing socket: 532
15:11:54.265 127.168.103.9 Connecting on socket:592
15:11:54.265 /njnu_files/createsunxml.htm
15:11:54.265 Closing socket: 592
15:11:54.296 127.168.103.9 Connecting on socket:588
15:11:54.296 Closing socket: 588
15:27:44.859 127.168.103.9 Connecting on socket:600
15:27:44.859 /toppic_main.jpg
15:27:44.859 Closing socket: 600
```

图 5.46 下载图片的 Socket

由此可见，网络程序运行的底层基础其实就是 Socket。

5.4.3 相对路径下的资源访问

如果在服务器根目录下再建立不同层次的目录结构，将资源分类存放在特定的子文件夹下，访问它们时会有什么不同呢？

如图 5.47 所示，在根目录下新建一个名为“MyWebPage”的子文件夹，将原本保存的网页拖放到新建的文件夹中。

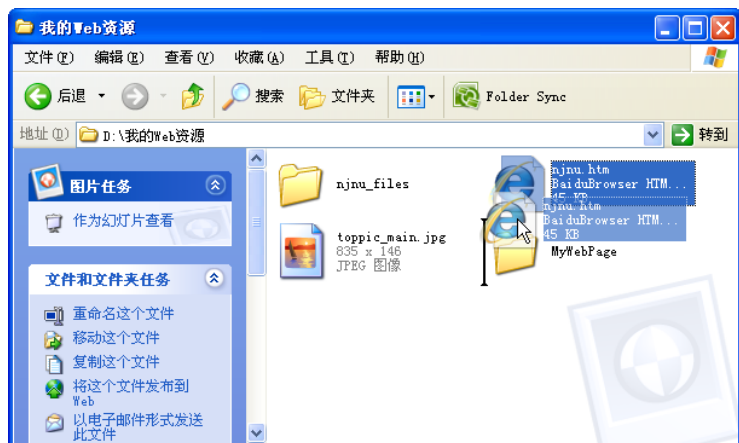


图 5.47 将资源放进子目录

分别启动服务器和浏览器程序后，在浏览器地址栏输入 `http://127.168.103.9:2258/MyWebPage/njnu.htm` 并回车或单击“浏览”按钮，同样可以看到网页，如图 5.48 所示。



图 5.48 浏览子目录下的网页

这里输入的网址 `http://127.168.103.9:2258/MyWebPage/njnu.htm` 分为三部分，其中，“`http://`”表示协议名称（使用 HTTP 协议），“`127.168.103.9:2258`”为服务器地址，“`/MyWebPage/njnu.htm`”为要访问的资源相对于服务器根目录的路径与文件名。这三部分加起来组成的字符串就是 5.1.3 节所说的 URL，它唯一标识了 Web 服务器上的某个特定的资源。

在实际应用中，一般不需要用户输入网站服务器的 IP，而是代之以网址，如本例南京师范大学主页的网址为 `www.njnu.edu.cn`，相比（点分十进制+端口号）的 IP 地址，网址是有意义的字符串，更便于用户理解和记忆。用户成功访问网站主页后，接下来的操作都可以通过单击页面上的超链接进行，而不需要再给出每个资源的相对路径（这一切都由程序代劳了），可见 HTTP 超文本链接机制的功能之强大、使用之方便。

FTP 编程与资源访问

FTP 是互联网发展早期最为主流的应用，近年来随着校园宽带的普及，FTP 主要用于国内各大院校的校园网，供高校师生共享学习、娱乐资源，同时它也是很多企业内部员工在工作中交换文件的主要方式。

6.1 FTP 应用基础

6.1.1 FTP 简介

FTP 是 File Transfer Protocol（文件传输协议），由 RFC 959 描述，工作在 TCP/IP 的应用层，其传输层使用的是 TCP，用于在网络上控制文件的双向传输。同时，它也是一切使用 FTP 协议传输文件的应用程序的统称。

与大多数 Internet 应用一样，FTP 也是基于客户端—服务器（C/S）模式工作的，如图 6.1 所示。



图 6.1 上传与下载

用户通过一个支持 FTP 协议的客户端程序，连接到远程主机上的 FTP 服务器程序，通过客户端程序向服务器程序发出命令，服务器程序执行用户所发出的命令，并将执行的结果返回客户端。例如，用户发出一条命令，要求服务器向其传送某个文件的一份副本，服务器会响应这条命令，将指定文件传送至用户的机器上。客户端程序代表用户接收这个文件，将其存放在用户目录中。

使用 FTP 的用户都十分熟悉两个名词：下载（Download）和上传（Upload）。“下载”文件就是从远程 FTP 服务器复制文件到自己的计算机上；“上传”文件则是将文件从自己的计算机传输至 FTP 服务器。即使是在 Web 广泛使用的今天，FTP 仍然是 Internet 上传、下载最主要的方式，而在校园网、企业网等各种局域网络中，FTP 更是传输网络资源的首选途径。

6.1.2 FTP 的特性

1. 适应异构系统

FTP 用来把一台主机上的文件传输到另一台主机,而这两台主机可能运行不同的操作系统、使用不同的文件结构及不同的字符集,这就要求 FTP 协议必须适用于异构系统。FTP 是通过支持有限数量的文件类型和数据结构来解决异构性的。

FTP 可以使用的文件类型有以下 4 种。

(1) ASCII 码文件。

这是 FTP 默认的文本文件格式,数据在传输过程中使用与 Telnet 相同的 NVT ASCII 码。这就要求发送方将本地文本文件转换成 NVT ASCII 码形式的文件,而接收方则将 NVT ASCII 码的文件再还原成本地文本文件。

(2) EBCDIC 码文件。

这也是一种文本类型文件,只是用 8 位代码表示一个字符,该文本文件在传输时要求两端都使用 EBCDIC 码。

(3) 图像 (Image) 文件。

图像文件也称二进制文件类型,发送的数据为连续的比特流。实际传输时,发送方将数据打包成 8 位,然后以字节为单位进行传输。

(4) 本地 (local) 文件。

本地文件字节的大小由本地主机定义,即每一字节的比特数由发送方规定,用于在具有不同字节大小的主机间传输二进制文件。对于使用 8 位字节的系统来说,本地文件以 8 位字节传输就等同于图像文件传输。

FTP 支持的文件数据结构有以下几类。

(1) 文件结构。这是 FTP 默认的方式,文件被认为是一个连续的字节流,不存在内部的结构。

(2) 记录结构。只适用于文本文件 (ASCII 或 EBCDIC),是由连续的记录构成的。

(3) 页结构。当文件由非连续的多个部分组成时,使用页结构,这种文件称为随机访问文件。每页都带有页号发送,以便接收方能随机地存储各页。

此外,FTP 还能以多种方式传输文件,如下所示。

(1) 流方式。这是默认的方式,文件以字节流的形式传输。可用于以上 3 种文件结构,只是对于记录结构,有专用的两字节序列码标志记录结束 (EOR) 和文件结束 (EOF)。

(2) 块方式。文件以一系列块来传输,每块前面都带有自己的头部。头部包括描述子代码域 (8 位) 和计数域 (16 位),描述子代码域定义数据块的结束标记等内容,计数域说明数据块的字节数。

(3) 压缩方式。用来对连续出现的相同字节进行压缩,现已很少使用。

在具体实现中,FTP 通常的工作方式是,以连续字节流方式传输 ASCII 码文件 (或二进制文件)。

2. 匿名 FTP

通常使用 FTP 必须先登录,输入用户名和密码,从远程主机获得相应的权限后,方可下载或上传文件。但这违背了 Internet 的开放性,匿名 FTP 就是为解决问题而产生的一种机制:用户可通过它连接到远程主机并下载文件,而无须成为其注册用户。系统管理员建立一个特殊的用户名——anonymous,Internet 上的任何人在任何地方都可以使用该用户名。

连接匿名 FTP 主机的方式同连接普通 FTP 主机差不多，只是在要求提供用户名时必须输入 anonymous，密码可以是任意字符串。但匿名 FTP 只适用于那些提供了这项服务的主机。当远程主机提供匿名 FTP 服务时，会指定某些目录向公众开放，允许匿名存取，而系统的其余目录则处于隐匿状态，无法访问。

目前，很多高校的 FTP 服务器都提供匿名服务。

6.1.3 FTP 工作原理

下面以客户要从 FTP 服务器上下载一个文件为例，说明 FTP 的完整工作过程。其工作原理可用图 6.2 表示。

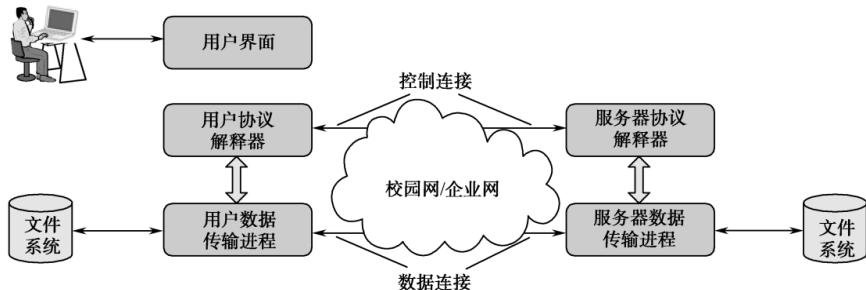


图 6.2 FTP 工作原理

1. 启动 FTP

用户通过 GUI 界面操作客户端软件，执行启动 FTP 的用户交互式命令。

2. 建立控制连接

客户端 TCP 层根据用户命令中给出的服务器 IP 地址，向服务器提供 FTP 服务的 21 端口发出主动建立连接的请求。服务器收到请求后，通过 3 次握手，在进行 FTP 命令处理的用户协议解释器进程和服务器协议解释器进程之间建立一条 TCP 连接。以后所有用户输入的 FTP 命令和服务器的应答都由该连接进行传输，因此把它叫作控制连接。控制连接在用户退出 FTP 之前一直存在。

3. 建立数据连接和进行文件传输

当用户通过交互式的界面，向 FTP 服务器发出要下载其上某一文件的命令时，该命令被送到用户协议解释器，用户协议解释器对该命令进行如下处理。

(1) 在客户端请求分配一个临时的 TCP 端口号。

(2) 在客户端由客户协议解释器通过控制连接向服务器协议解释器发送两条命令：一条命令（使用 PORT 命令）是把客户端的 IP 地址和申请到的临时端口号这两个参数告诉服务器协议解释器；另一条命令是把服务器上的某文件传输到客户端的服务请求。

(3) 服务器协议解释器收到客户端的 IP 地址和临时端口号后，以该 IP 地址和端口号为目标，使用服务器的 20 端口（TCP 层用于传输数据的端口）向客户端发出主动建立连接的请求。

(4) 客户端收到请求后，通过 3 次握手在用户数据传输进程和服务器数据传输进程之间建立一条 TCP 连接，下面的传输文件就通过这个连接进行，由于它是建立用来专门传输数据的，因此把它叫作数据连接。

- (5) 服务器的数据传输进程从它的文件系统中找到用户进程请求传输的文件。
- (6) 服务器的数据传输进程通过数据连接把该文件发送到用户数据传输进程。
- (7) 用户数据传输进程把该文件交给客户端文件系统进行存储。
- (8) 文件传输完成后, 由服务器主动关闭该数据连接。

4. 关闭 FTP

当用户要退出 FTP 站点时通过客户端发出退出 FTP 的交互式命令, 控制连接被关闭, FTP 服务结束。除了本例中介绍的客户端从 FTP 服务器上下载一个文件时要建立数据连接外, 当客户端要向服务器上传一个文件或客户端要求查看服务器文件列表时也要建立相应的数据连接。因此在下列 3 种情况下都需要在客户端和服务器之间自动建立数据连接。

- 从客户端向服务器发送一个文件。
- 从服务器向客户端发送一个文件。
- 从服务器向客户端发送文件目录列表。

另外, 在建立数据连接之前, 客户端协议解释器通过控制连接, 把其要建立数据连接的端口通知给服务器协议解释器。因此在客户端控制连接和数据连接使用不同的端口号。其实客户端可以不对服务器通知数据连接的端口, 在这种情况下, 服务器照样可以与客户端建立一条数据连接, 只不过在客户端, 数据连接就使用与控制连接相同的端口号。这时控制连接和数据连接可以分别表示如下。

- 控制连接:

<客户端 IP 地址, 客户端控制连接端口, 服务器 IP 地址, FTP 服务器控制连接端口 21>

- 数据连接:

<客户端 IP 地址, 客户端控制连接端口, 服务器 IP 地址, FTP 服务器数据连接端口 20>

由于两对连接中 FTP 服务器控制连接与数据连接的端口不同, 因此它们是不同的连接。

6.1.4 FTP 命令和应答

在用户协议解释器和服务器协议解释器之间的控制连接上, 传输的是 FTP 命令和应答信息。由用户协议解释器负责发送命令和解释接收到的应答, 由服务器协议解释器执行命令并把执行情况以应答的形式发送给客户端。所有 FTP 命令和应答都在控制连接上以 NVT ASCII 码的形式传输, 并且每个命令或应答都以<CR>和<LF>对结尾, 换句话说, 就是一个命令或应答占一行。

1. FTP 命令

FTP 命令都是由 3 个或 4 个大写 ASCII 码字符组成的, 表 6.1 列出了 FTP 命令, 共分为 3 大类。这里所讲的 FTP 命令不是用户从终端上输入的命令, 而是客户端从终端上输入交互式 FTP 用户命令后由协议解释器自动生成的。一般情况下, 一个交互式 FTP 用户命令对应一个 FTP 命令, 但有些也对应多个 FTP 命令, 如客户端从终端上输入的 dir 命令, 就可能产生 PORT 和 LIST 两条 FTP 命令。这里探讨的是 FTP 的工作原理, 所以只介绍 FTP 命令。现在的各种 FTP 客户端软件早已是 GUI 图形化操作, 非专业计算机用户都不会在命令行下通过直接输入交互式 FTP 用户命令的方式使用 FTP, 而是直接用鼠标单击按钮就可以轻松自如地操作 FTP 上的资源, 故此处不再严格区分这两种命令。

表 6.1 FTP 命令

命令类型	命令	功能说明
访问控制命令（用于指定访问控制标记）	USER	服务器上的用户名。用户标记是访问服务器必需的，此命令通常是控制连接建立后第一个发出的命令
	PASS	用户口令。此命令紧跟 USER 命令后，在某些站点它是完成访问控制不可缺少的一步
	ACCT	用户账户。一些站点需要账户才能登录，另一些站点可以通过用户账户对操作权限进行限制
	CWD	改变工作目录。此命令使用户可以在不同的目录或数据集下工作，而不用改变它的登录或账户信息
	CDUP	回到上一层目录（父目录），此命令要求系统实现目录树结构，它的响应和 CWD 的相同
	SMNT	结构加载。此命令使用户在不改变登录或账户信息的情况下，加载另一个文件系统数据结构
	REIN	重新初始化。此命令终止 USER，重置所有参数，控制连接仍然打开，用户可以再次使用 USER 命令
	QUIT	退出登录。此命令终止 USER，服务器关闭控制连接
传输参数命令（用于在数据传输时设置默认值）	PORT	数据端口。主要向服务器发送客户端数据连接的端口，格式为 PORT h1, h2, h3, h4, p1, p2。其中，32 位的 IP 地址用 h1, h2, h3, h4 表示，16 位的 TCP 端口号用 p1, p2 表示
	PASV	此命令要求服务器数据传输进程在指定的数据端口侦听，进入被动接收请求的状态
	TYPE	文件类型。可指定 ASCII、EBCDIC、Image、本地类型文件等参数
	STRU	文件结构。用于指定文件结构。参数：F——文件结构（默认值）；R——记录结构；P——页结构
	MODE	传输模式。命令参数：S——流文件（默认值）；B——块文件；C——压缩文件
服务命令（定义用户请求的文件传输或文件系统功能）	RETRI	获得文件。将指定路径的文件复制到客户端。服务器上文件的状态和内容不受影响
	STOR	保存文件。向服务器传输文件。如果文件已存在，原文件将被覆盖；如果文件不存在，则新建文件
	STOU	唯一保存文件。此命令和 STOR 的功能类似，但它要求在此目录下的文件名是唯一的
	APPE	与 STOR 的功能类似，但如果文件在指定路径已存在，则把数据附加到原文件尾部；如果不存在，则新建一个文件
	ALLO	分配存储器。用于在一些主机上为新传送的文件分配足够的存储空间
	REST	重新开始。参数代表服务器要重新开始的那一点，它并不传送文件，而是略过指定点后的数据，此命令后应该跟其他要求文件传输的 FTP 命令
	RNFR	重命名。与在操作系统中使用的重命名命令一样，只不过后面要跟“rename to”，以指定新的文件名

续表

命令类型	命令	功能说明
服务命令（定义用户请求的文件传输或文件系统功能）	ABOR	异常终止。此命令通知服务器终止以前的 FTP 命令和与之相关的数据传送。如果先前的操作已经完成，则没有动作，返回 226；如果没有完成，则返回 426，然后再返回 226
	DELE	删除文件。此命令删除指定路径下的文件。用户进程负责对删除的提示
	RMD	删除目录。此命令删除指定的目录
	MKD	创建目录。此命令在指定的路径下创建新目录
	PWD	打印工作目录。它的响应是返回当前工作目录
	LIST	列表。服务器传送指定路径目录或文件列表到客户端。参数为空时表示用户当前的工作目录或默认目录
	NLST	名字列表。服务器传送指定路径目录表名到客户端，参数为空时指当前目录，以 ASCII 或 EBCDIC 形式传送
	SITE	站点参数。用来提供服务器系统的信息，信息因系统的不同而不同
	SYST	系统。用于确定服务器上运行的操作系统
	STAT	状态。此命令返回控制连接的状态
	HELP	帮助。用于取得系统帮助信息
	NOOP	等待。此命令不产生实际动作，它仅使服务器返回 OK

2. FTP 应答

FTP 命令的应答是服务器对 FTP 命令执行情况的响应，它主要有两方面的功能。

(1) 服务器为了对数据传输的请求和过程进行同步，这是 TCP 所要求的，TCP 要求对接收到的数据都要进行确认。

(2) 为了让用户了解服务器的状态，用户可以根据收到的状态信息对服务器是否正常执行了有关操作进行判断。

FTP 要求每个命令最少有一个响应，如果有多个响应，它们要易于区别。服务器通过控制连接发送给客户端的 FTP 应答，由 ASCII 码形式的 3 位数字和一行文本提示信息组成，它们之间用一个空格分隔。数字带有足够多的信息，使用户协议解释器不用检查文本就知道发生了什么。文本信息与服务器相关，不同的服务器对于同一个命令，产生的应答文本提示信息可能不同。应答信息的每行文本以回车<CR>和换行<LF>对结尾。如果需要产生一条多行的应答，第 1 行在 3 位数字应答代码之后包含一个连字号“-”，而不是空格，最后一行包含相同的 3 位数字应答代码，后跟一个空格符。3 位数字的每一位都有特定的意义，详细内容见表 6.2。

表 6.2 FTP 应答

应 答	说 明
第一位数字	1yz 肯定预备应答。它仅仅是在发送另一个命令前期待另一个应答时启动
	2yz 肯定完成应答。一个新命令可以发送
	3yz 肯定中介应答。该命令已被接收，但另一个命令必须被发送
	4yz 暂时否定完成应答。请求的动作没有发生，但差错状态是暂时的，所以命令可以过后再发
	5yz 永久性否定完成应答。命令不被接收，并且不再重试

续表

应 答		说 明
第 二 位 数 字	x0z	语法错误
	xlz	信息
	x2z	连接。应答指控制或数据连接
	x3z	鉴别和记账。应答用于注册或记账命令
	x4z	未使用
	x5z	文件系统状态

第一位数字的取值范围为 1~5，它用来确定响应是正确的、错误的还是不完全的。通过检查第一位，用户进程通常就能够知道大致要采取什么行动了。第二位和第三位数字所组成的编码是对应答提供的附加说明。第三位数字是在第二位数字的基础上对应答内容的进一步细化。

表 6.1 和表 6.2 中的命令应答大家现在不需要理解，更不要去记，到后面阅读程序用到时再回来查表，就很容易弄懂其含义了。

6.1.5 FTP 网络环境搭建和使用

微软 Windows 操作系统已经内置了 FTP 服务器的功能，只需一些简单的设置，就可以将计算机配置成一台 FTP 服务器。

1. 安装 FTP 服务组件

FTP 服务组件不是 Windows 默认安装的，当需要使用时要由用户来补装添加这个组件。选择菜单“开始”→“控制面板”，双击“添加或删除程序”项，在弹出的对话框左侧单击“添加/删除 Windows 组件”按钮，如图 6.3 所示。



图 6.3 添加/删除 Windows 组件

在弹出的“Windows 组件向导”对话框中勾选“Internet 信息服务”复选框，单击“详细信息...”按钮，如图 6.4 所示。

在弹出的“Internet 信息服务 (IIS)”对话框中找到“文件传输协议 (FTP) 服务”项并勾选。然后连续单击“确定”按钮，直至如图 6.5 所示的界面。

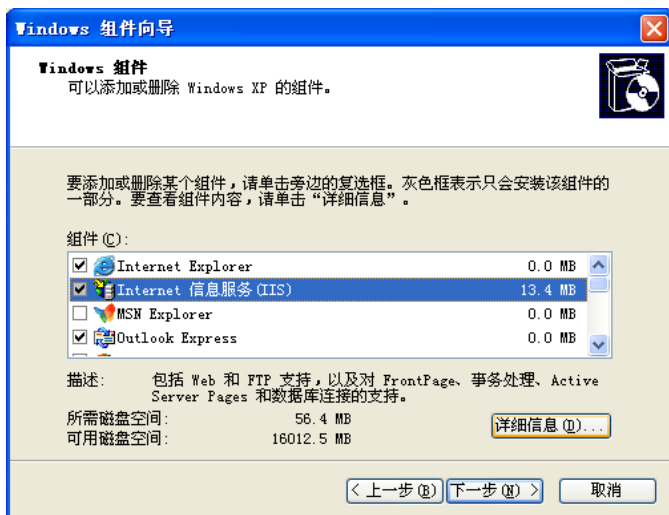


图 6.4 “Windows 组件向导”对话框

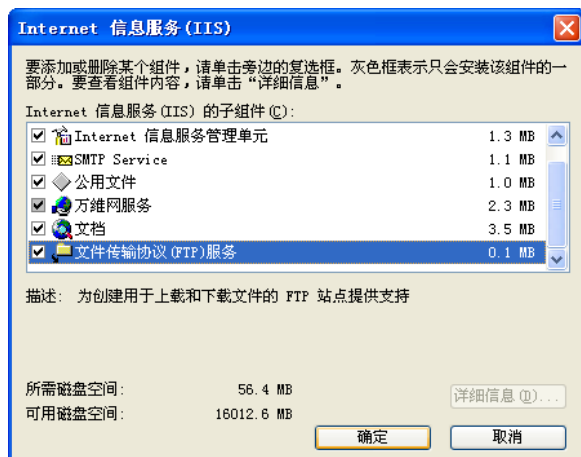


图 6.5 添加文件传输协议 (FTP) 服务

找到安装操作系统时使用的 Windows 安装盘，放入光驱后单击“确定”按钮，如图 6.6 所示，系统将自动启动安装过程。

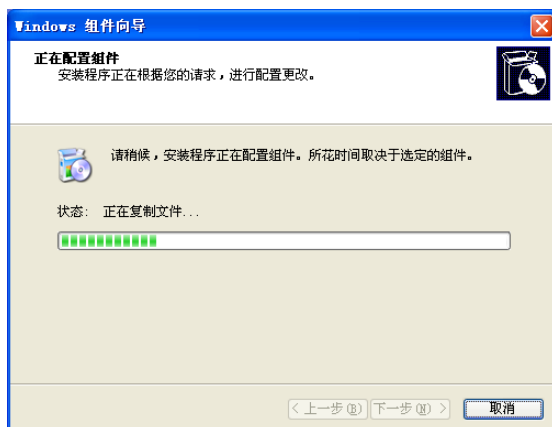


图 6.6 安装 FTP 组件

稍等片刻就安装完成了，单击“完成”按钮结束安装。

2. 配置 FTP 站点

选择菜单“控制面板”→“性能和维护”→“管理工具”→“Internet 信息服务”，打开如图 6.7 所示的“Internet 信息服务”窗口。

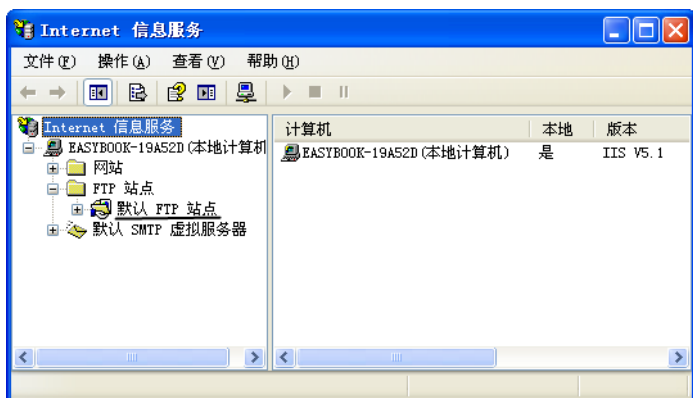


图 6.7 本地计算机上默认建好的 FTP 站点

可以看到 Internet 信息服务目录树中有一个“FTP 站点”文件夹，里面已经有有了一个默认的 FTP 站点。

为了以后测试 FTP 程序向服务器上传（删除）文件等“写入”类功能，必须同时开放 FTP 服务器的读和写权限。右击“默认 FTP 站点”→“属性”，在如图 6.8 所示的“默认 FTP 站点 属性”对话框中选择“主目录”标签页。



图 6.8 配置 FTP 主目录属性

在“FTP 站点目录”组框中勾选“写入”复选框。图 6.8 中框出的“本地路径”文本框中的“c:\inetpub\ftproot”是默认 FTP 站点根目录所在路径，位于本地计算机 C 盘的 inetpub 文件夹下，用户可以打开这个文件夹查看。

最后，将 FTP 站点的“写入”权限也一并开放，操作如图 6.9 所示。

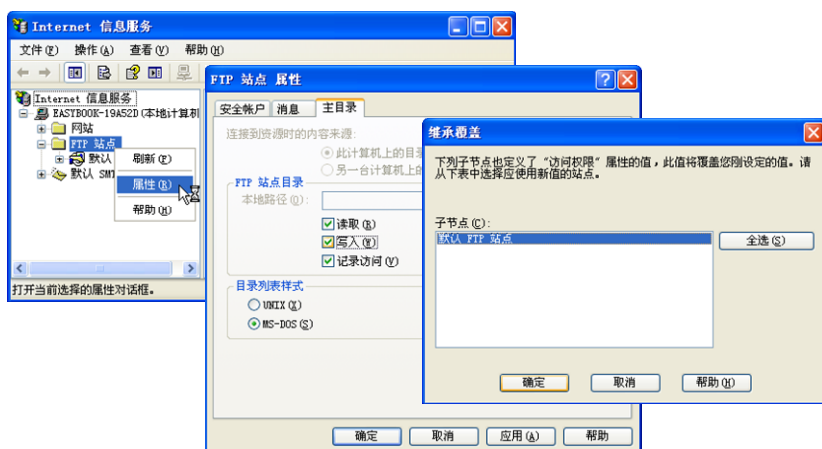


图 6.9 开放 FTP 站点的写入权限

至此，一个 FTP 站点就架设好了。

3. 测试 FTP 站点

FlashFXP 是一款功能强大的 FTP 客户端软件，拥有庞大的用户群和丰富的种子资源，其主界面如图 6.10 所示。

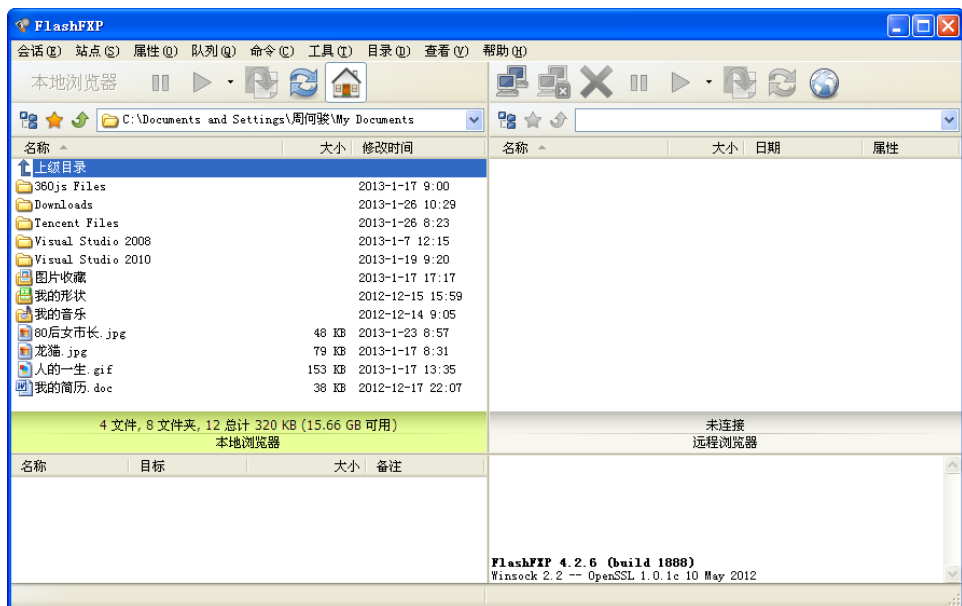


图 6.10 FlashFXP 的主界面

其中，右侧空白区域为显示 FTP 服务器目录和文件的地方，左侧的文件列表显示的则是本地的文件和文件夹。

选择菜单命令“会话”→“快速连接”，在图 6.11 所示“快速连接”对话框的“地址或 URL”栏输入“127.0.0.1”（本地计算机环回测试地址），端口保持默认的“21”，匿名登录，单击“连接”按钮。

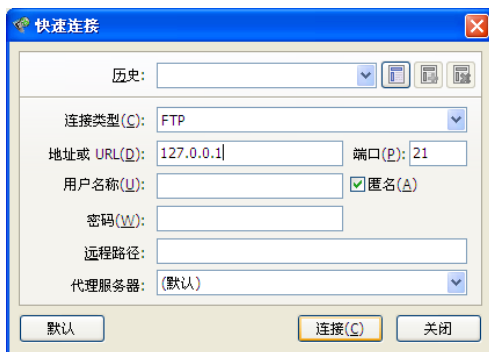


图 6.11 设置 FlashFXP 的连接

右击本地目录下的文档“我的简历.doc”→“传输”，如图 6.12 所示，文件被上传到 FTP 根目录下。

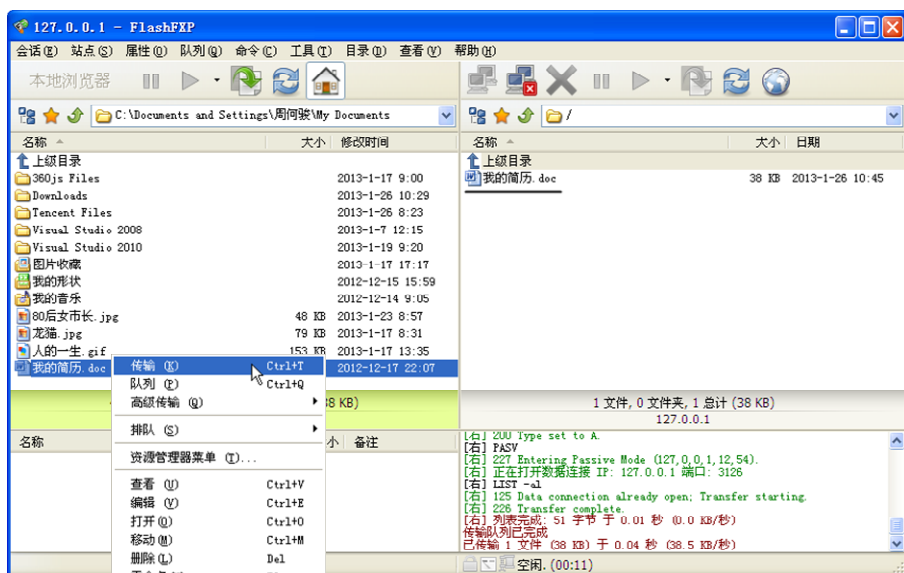


图 6.12 上传成功

读者进入“C:\Inetpub\ftproot”即可看到刚刚上传的文件，大家也可以测试文件下载、删除等功能。

6.2 制作 FTP 上传下载器

6.2.1 WinInet 类对 FTP 的支持

FTP 客户端编程也是主要以 MFC WinInet 为支撑的。WinInet 提供了如图 6.13 所示的 Internet 会话类 CInternetSession、连接类 CInternetConnection、文件类 CInternetFile、文件操作类 CFileFind 及通用异常类 CInternetException 等类。

本例 FTP 上传下载器就是使用 WinInet 开发的，所用到的相关类如图 6.13 中黑框突出显示。

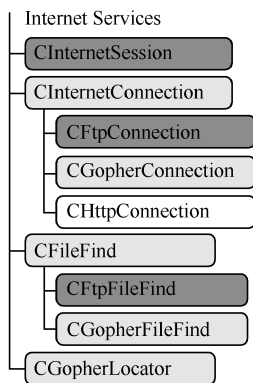


图 6.13 WinInet 接口

6.2.2 设计软件界面

创建 MFC 工程，工程名为 SelfFtpUpDownloader（自己制作的 FTP 客户端）。这个工程采用传统的对话框类型。因为 WinInet 封装了 Socket 使用 FTP 协议与服务器通信的细节，不需要用户自己编写 Socket 程序，所以在“高级功能”页也不需要勾选“Windows 套接字”复选框。

工程创建好后，设计软件界面如图 6.14 所示。

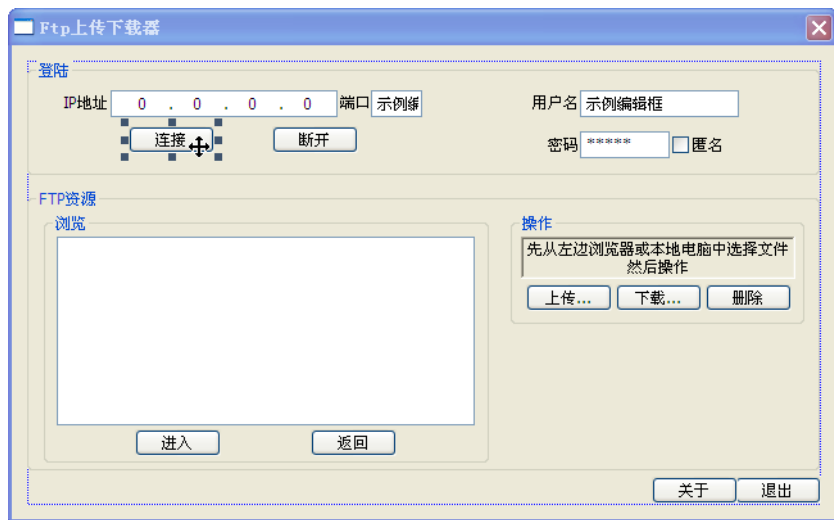


图 6.14 FTP 客户端界面

可以看出，这个 FTP 上传下载器的功能是很完善的：可以连接到由用户指定 IP 和端口的 FTP 服务器，用户也可随时与服务器断开连接，支持匿名登录；在左区浏览框中可以如在本地计算机一样查看 FTP 上的资源，并可自由地进入、退出服务器系统的文件夹，切换目录；用户可借助这款软件实现对 FTP 资源的基本操作，包括上传和下载文件，以及删除服务器上的文件。

与界面各元素关联的变量见表 6.3。

表 6.3 FTP 上传下载器界面控件变量

变 量 控 件	Control	Value
IP 控件	ServerIP	—
“端口”文本框	m_port	strport
“用户名”文本框	m_usr	strusr
“密码”文本框	m_pwd	strpwd
“匿名”复选框	m_noname	—
“浏览”列表框	m_lst	—
“连接”按钮	m_connect	—
“断开”按钮	m_disconnect	—
“进入”按钮	m_enterdir	—
“返回”按钮	m_goback	—
“上传”按钮	m_upload	—
“下载”按钮	m_download	—
“删除”按钮	m_delete	—
“退出”按钮	m_exit	—

6.2.3 编程实现

为了能在编程中使用 WinInet 类的功能，需要在 SelfFtpUpDownloaderDlg.h 中包含头文件：

```
#include "afxinet.h"
```

虽然在界面上设计了“用户名”和“密码”文本框，但本节先实现最简单的匿名登录功能，到后面与自制 FTP 服务器对接时再添加用户身份验证的功能。用户启动 FTP 客户端后，必须输入要访问服务器的 IP 和端口，还必须勾选“匿名”复选框。

勾选“匿名”复选框将触发 OnNoname 方法，代码如下：

```
int icheck = m_noname.GetCheck();           //获得“匿名”复选框的选择状态
if(icheck == 1)                             //icheck==1 表示用户选中“匿名”复选框
{
    m_usr.EnableWindow(false);
    m_pwd.EnableWindow(false);
    m_usr.SetWindowTextA("anonymous");      //用户名自动设置为默认的“anonymous”
    m_pwd.SetWindowTextA("");
    UpdateData();
    if(!ServerIP.IsBlank() && !strport.IsEmpty())
    {
        m_connect.EnableWindow(true);       //“连接”按钮变为可用
    }
}
else                                         //如果用户没有按照要求输入，就不能连接
{
    m_usr.EnableWindow(true);
}
```

```

m_pwd.EnableWindow(true);
m_usr.SetWindowTextA("");
m_pwd.SetWindowTextA("");
m_connect.EnableWindow(false);          // “连接”按钮不可用，禁止用户继续操作
}

```

可见，这段代码旨在检查用户是否按要求输入，本程序的要求是，IP 和端口要填写完整，还要勾选“匿名”复选框。只有完全按要求做了，“连接”按钮才变为可用，否则该按钮将不可用，以杜绝用户非法操作。用户在按照要求输入后，就可以连接想要访问的 FTP 服务器了。

“连接”按钮的事件过程，代码如下：

```

void CSelfFtpUpDownloaderDlg::OnConnect()
{
    this->ConnectFtp();                //连接 FTP 服务器
    this->UpdateDir();                //显示服务器上的目录和文件夹列表
    //以下为界面控制
    ServerIP.EnableWindow(false);
    m_port.EnableWindow(false);
    m_connect.EnableWindow(false);
    m_disconnect.EnableWindow(true);
    m_enterdir.EnableWindow(true);
    m_upload.EnableWindow(true);
    m_download.EnableWindow(true);
    m_delete.EnableWindow(true);
    m_noname.EnableWindow(false);
    m_exit.EnableWindow(false);
}

```

这个事件过程用到两个函数：ConnectFtp()和 UpdateDir()。

ConnectFtp()函数代码如下：

```

void CSelfFtpUpDownloaderDlg::ConnectFtp()
{
    BYTE nFild[4];
    UpdateData();
    ServerIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
    CString sip;
    sip.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
    if(sip.IsEmpty())
    {
        AfxMessageBox("请指定 IP 地址!");
        return;
    }
    if(strport.IsEmpty())
    {
        AfxMessageBox("请指定连接端口!");
        return;
    }
    if(strusr.IsEmpty())
        return;
}

```

```

//建立一个 Internet 会话
pInternetSession=new CInternetSession("MR",INTERNET_OPEN_TYPE_PRECONFIG);
try
{
    //利用 Internet 会话对象 pInternetSession 打开一个 FTP 连接
    pFtpConnection = pInternetSession->GetFtpConnection(sip, strusr, strpwd,atoi(strport));
    bconnect = true;
} catch(CInternetException* pEx)
{
    TCHAR szErr[1024];
    pEx->GetErrorMessage(szErr, 1024);
    AfxMessageBox(szErr);
    pEx->Delete();
}
}

```

FTP 客户端程序要建立与服务器的连接, 需要一个 `CInternetSession` 和 `CFtpConnection` 对象, 但并不需要直接创建 `CFtpConnection` 对象, 而是通过调用 `CInternetSession::GetFtpConnection` 来实现的。

`UpdateDir()` 函数代码如下:

```

void CSelfFtpUpDownloaderDlg::UpdateDir()
{
    m_lst.ResetContent();
    //读写服务器中的数据, 必须创建一个 CFtpFileFind 的实例
    CFtpFileFind ftpfind(pFtpConnection);
    //找到第一个文件(夹), 通过 CFtpFileFind::FindFile 实现
    BOOL bfind = ftpfind.FindFile(NULL);
    while(bfind)
    {
        bfind = ftpfind.FindNextFile();
        CString strpath;
        if (!ftpfind.IsDirectory())                //判断是目录(文件夹)还是文件
        {
            strpath = ftpfind.GetFileName();        //如果是文件则获取文件名
            m_lst.AddString(strpath);
        }
        else
        {
            strpath = ftpfind.GetFilePath();        //如果是文件夹则获取相对路径
            m_lst.AddString(strpath);
        }
    }
}

```

WinInet 的 `CftpFileFind` 将服务器上的数据 (包括文件和文件夹) 都统一作为文件看待, 因此要对读取的内容进行判断。`UpdateDir()` 函数用于读取和更新显示 FTP 服务器上的资源目录, 它在本程序中是一个通用函数。

上述连接过程和更新资源目录的过程分别单独封装为两个函数——`ConnectFtp()` 和 `UpdateDir()`,

由“连接”按钮的处理过程调用，而不是直接将它们的代码写在程序中，这么做是为了向用户屏蔽短连接带来的弊端。

所谓“短连接”，就是通信双方有数据交互时，才建立一个 TCP 连接，数据发送完成不久即断开此连接。用户想要保持与服务器的连接始终不中断，就必须频繁不断地有操作动作，这给使用造成了不便。本程序采取了一项措施：每当用户进行一步新的操作时，程序都会自动执行重新连接服务器和更新资源目录的过程，使用户感觉不到之前服务器曾断开过，这就抵消了短连接对软件易用性的负面影响。但程序经过这样的处理后，对于用户每一步操作所触发的事件过程，都必须首先执行连接和更新资源目录的相同操作，也就使得这种操作的代码在程序中重复出现很多次，为了避免代码冗余，特将这两个通用的操作封装为两个函数。

为了保证连接的顺利成功，需要在 SelfFtpUpDownloaderDlg.h 的对话框类定义中添加 CInternetSession 类对象指针 pInternetSession 和 CFtpConnection 类对象指针 pFtpConnection 的定义，还要定义指示连接成功与否的变量 bconnect。另外，声明上述两个函数的原型：

```
BOOL bconnect;
CInternetSession *pInternetSession;
CFtpConnection *pFtpConnection;
void ConnectFtp();
void UpdateDir();
```

成功登录 FTP 服务器后，FTP 资源浏览框中会自动列出服务器上所有的目录和文件，用户可以自由地进入、退出文件夹以浏览服务器各个目录下的资源。

“进入”按钮使得用户可以进入自己选中的目录文件夹，其事件代码如下：

```
void CSelfFtpUpDownloaderDlg::OnEnterDir()
{
    CString selfile;
    m_lst.GetText(m_lst.GetCurSel(),selfile);           //获取用户选择的目录名
    if (!selfile.IsEmpty())
    {
        pInternetSession->Close();                     //及时关闭废弃的会话句柄
        this->ConnectFtp();                             //重新连接，保持与服务器的持续会话
        CString strdir;
        pFtpConnection->GetCurrentDirectory(strdir);    //获得原来的工作目录
        strdir += selfile;                             //生成新目录
        pFtpConnection->SetCurrentDirectory(strdir);    //改变目录到当前服务目录
        this->UpdateDir();                             //更新目录列表
        m_goback.EnableWindow(true);
    }
}
```

为使用户灵活地切换目录，程序必须提供目录返回功能，让用户能够返回上一级目录。“返回”按钮的事件过程如下：

```
void CSelfFtpUpDownloaderDlg::OnGoBack()
{
    CString strdir;
    pFtpConnection->GetCurrentDirectory(strdir);
    int pos;
    pos = strdir.ReverseFind('/');                     //用字符串截取的方式获得上一级目录
```



```

strdir = strdir.Left(pos);
pInternetSession->Close();           //关闭废弃的会话
this->ConnectFtp();                  //重新连接，保持持续会话
pFtpConnection->SetCurrentDirectory(strdir);
this->UpdateDir();                   //更新目录列表
}

```

现在，用户已经可以像操作自己的计算机一样访问服务器上的资源了！本例实现了用户对 FTP 资源的上传、下载和删除这三项最基本的操作。

“上传”按钮的事件过程代码如下：

```

void CSelfFtpUpDownloaderDlg::OnUpLoad()
{
    CString str;
    CString strname;
    //弹出“打开”对话框
    CFileDialog file(true,NULL,NULL,OFN_HIDEREADONLY |
        OFN_OVERWRITEPROMPT,"所有文件(*.*)|*.*|",this);
    if(file.DoModal() == IDOK)
    {
        str = file.GetPathName();
        strname = file.GetFileName();
    }
    if(bconnect)
    {
        CString strdir;
        pFtpConnection->GetCurrentDirectory(strdir);
        //上传文件
        BOOL bput=pFtpConnection->PutFile((LPCTSTR)str,(LPCTSTR) strname);
        if(bput)
        {
            pInternetSession->Close();           //关闭废弃的会话
            this->ConnectFtp();                   //保持持续会话
            pFtpConnection->SetCurrentDirectory(strdir);
            this->UpdateDir();                   //更新目录列表
            AfxMessageBox("上传成功!");
        }
    }
}

```

“下载”按钮的事件过程代码如下：

```

void CSelfFtpUpDownloaderDlg::OnDownLoad()
{
    CString selfile;
    m_lst.GetText(m_lst.GetCurSel(),selfile);//获知用户选择要下载的资源名
    if(!selfile.IsEmpty())
    {
        //弹出“另存为”对话框
        CFileDialog file(false,NULL,selfile,OFN_HIDEREADONLY |

```

```

        OFN_OVERWRITEPROMPT,"所有文件(*.*)|*.*|", this);
    if(file.DoModal() == IDOK)
    {
        CString strname;
        strname = file.GetFileName();
        CString strdir;
        pFtpConnection->GetCurrentDirectory(strdir);
        pFtpConnection->GetFile(selffile,strname); //下载文件到选定的本地位置
        pInternetSession->Close();           //关闭废弃的会话
        this->ConnectFtp();                   //保持持续会话
        pFtpConnection->SetCurrentDirectory(strdir);
        this->UpdateDir();                     //更新目录列表
        AfxMessageBox("下载成功!");
    }
}

```

“删除”按钮的事件过程代码如下：

```

void CSelfFtpUpDownloaderDlg::OnDelete()
{
    CString selffile;
    m_lst.GetText(m_lst.GetCurSel(),selffile);           //获取用户要删除的资源名
    if (!selffile.IsEmpty())
    {
        //弹出对话框让用户确认删除
        if(AfxMessageBox("您真的要删除这个文件吗？", 4 + 48) == 6)
        {
            pFtpConnection->Remove(selffile);           //一旦用户确认，即删除该文件
        }
        CString strdir;
        pFtpConnection->GetCurrentDirectory(strdir);
        pInternetSession->Close();           //关闭废弃的会话
        this->ConnectFtp();                   //保持持续会话
        pFtpConnection->SetCurrentDirectory(strdir);
        this->UpdateDir();                     //更新目录列表
    }
}

```

当用户访问完毕后，单击“断开”按钮退出登录，其事件过程如下：

```

void CSelfFtpUpDownloaderDlg::OnDisconnect()
{
    pInternetSession->Close();           //结束会话
    m_lst.ResetContent();
    m_lst.AddString("连接已断开!");
    //界面控制
    ServerIP.EnableWindow(true);
    m_port.EnableWindow(true);
    m_connect.EnableWindow(true);
    m_disconnect.EnableWindow(false);
}

```

```

m_enterdir.EnableWindow(false);
m_goback.EnableWindow(false);
m_upload.EnableWindow(false);
m_download.EnableWindow(false);
m_delete.EnableWindow(false);
m_noname.EnableWindow(true);
m_exit.EnableWindow(true);
}

```

断开过程其实就是结束会话，另外主要是进行一些界面控制，还要在 FTP 资源浏览框中显示“连接已断开!”。

为了使整个程序功能完善，在客户端刚启动、用户尚未登录时，也需要进行一些初始化工作，告诉用户“目前尚未登录!”，初始化代码如下：

```

bconnect = false;
m_lst.ResetContent();
m_lst.AddString("尚未连接服务器，无法浏览 FTP 资源!");
m_connect.EnableWindow(false);
m_disconnect.EnableWindow(false);
m_enterdir.EnableWindow(false);
m_goback.EnableWindow(false);
m_upload.EnableWindow(false);
m_download.EnableWindow(false);
m_delete.EnableWindow(false);

```

这段代码位于 SelfFtpUpDownloaderDlg.cpp 文件 CSelfFtpUpDownloaderDlg 类的 OnInitDialog() 方法中。

至此，整个 FTP 客户端软件就开发完成了，下面来进行测试。

6.2.4 测试 FTP 客户端

将之前已经上传到 C:\inetpub\ftproot 下的 Word 文档删除，启动 FTP 上传下载器 SelfFtpUpDownloader，如图 6.15 所示，在 IP 地址栏中输入“127.0.0.1”，端口号填“21”，勾选“匿名”复选框，单击“连接”按钮访问本机上的 FTP 服务器。

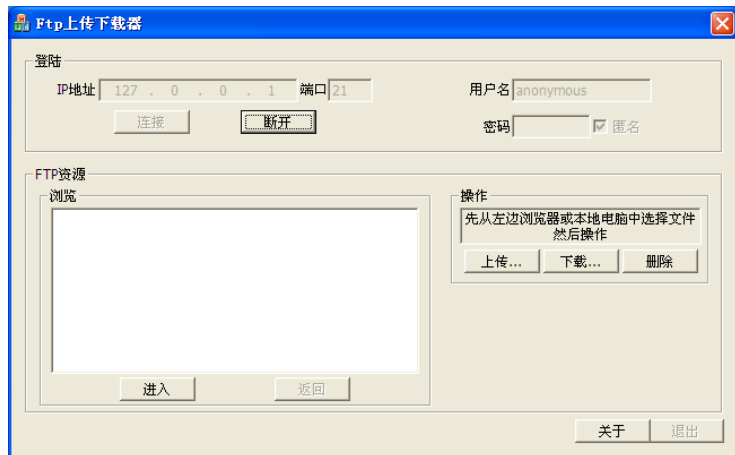


图 6.15 连接本机 FTP 服务器

此时由于服务器上 (C:\inetpub\ftproot 目录下) 暂时无资源, 故“FTP 资源”浏览框显示为空白。

单击“上传...”按钮, 弹出“打开”对话框, 进入如图 6.16 所示的“我的文档”目录, 选中“我的简历.doc”文档, 单击“打开”按钮。

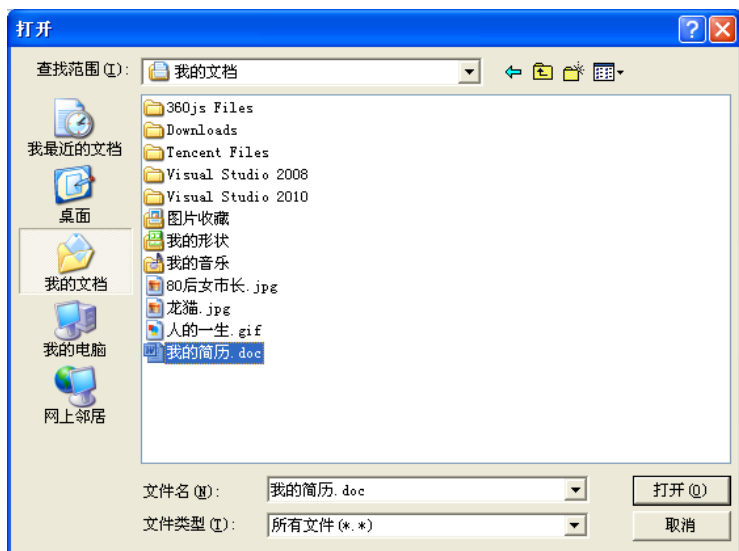


图 6.16 选择要上传的文档

文件上传到服务器, 显示“上传成功!”消息框, 如图 6.17 所示。

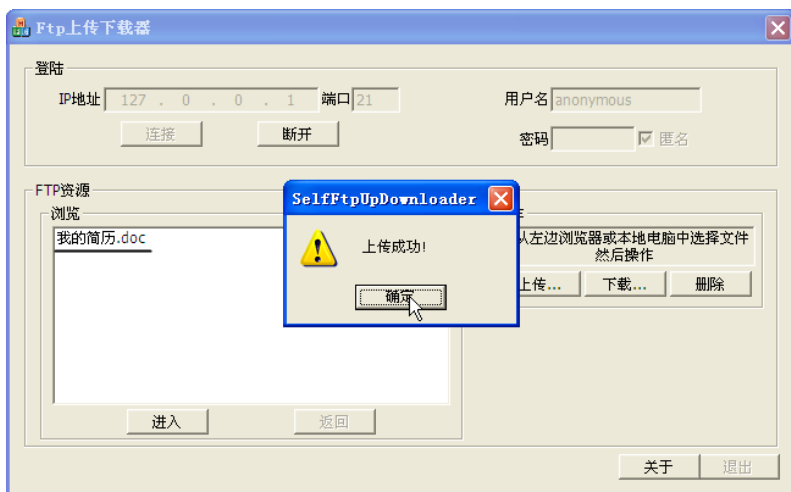


图 6.17 上传成功

大家可以看到, 在“FTP 资源”浏览框中显示出文档“我的简历.doc”。读者还可以自行测试下载、删除等功能。

6.3 FTP 服务器的实现

从本节开始, 结合一个 FTP 服务器的实现, 通过阅读分析其源代码, 达到使读者真正理解 FTP 工作原理的目的。这个服务器程序也是比较复杂的, 代码量很多。在此还是用第 5 章分析

Web 服务器程序时所采用的方法，将它的源代码按照由表及里、由宏观到微观的顺序分成 5 大部分，依次由浅入深地逐步理解整个软件的工作过程。

6.3.1 项目框架的建立

首先创建 VC 项目工程，工程名为 ftpSrver（“基于 FTP 协议的服务器”之意）。设计程序界面如图 6.18 所示。

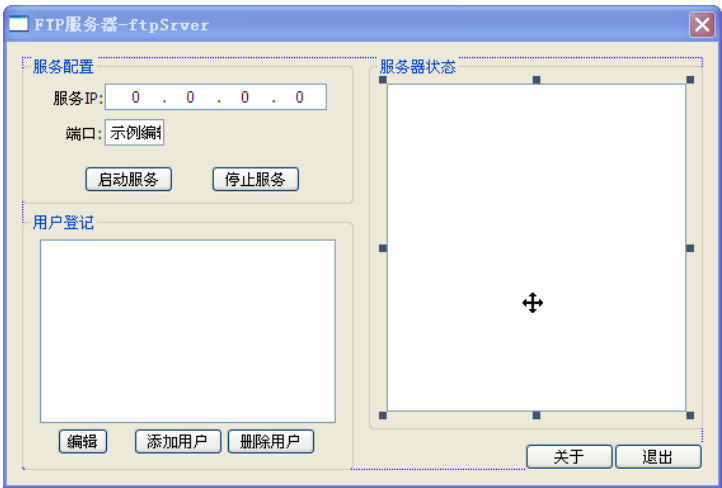


图 6.18 FTP 服务器界面

由于服务器提供了对已注册用户信息进行编辑及添加和删除用户的功能，因此需要另外设计一个对话框专门作为设置用户信息之用。向工程中添加 MFC 类 CAccountDlg，并设计对话框界面，如图 6.19 所示。

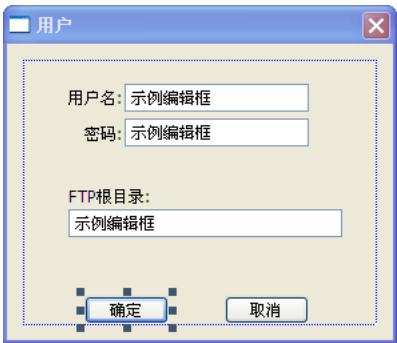


图 6.19 “用户”对话框界面

这个对话框叫作“用户”，管理员可以在其中建立用户账户，为其设置用户名、密码和该用户在服务器上拥有的 FTP 根目录。

GUI 界面上的各控件关联变量见表 6.4 和表 6.5。

表 6.4 主对话框控件变量

变 量 控 件	Control	Value
“服务 IP” 地址控件	ServerIP	—
“端口” 文本框	CEDITPORT	m_EditPort
“用户登记” 列表控件	CLISTACCOUNT	—
“服务器状态” 列表控件	CLISTSERVERINFO	—

表 6.5 “用户” 对话框控件变量

变 量 控 件	Control	Value
“用户名” 文本框	—	m_username
“密码” 文本框	—	m_password
“FTP 根目录” 文本框	—	m_directory

在项目中添加类 CServer 和结构体 CAccount（如图 6.20 所示）。

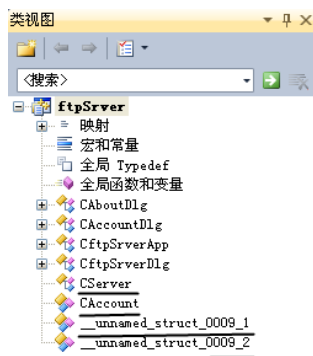


图 6.20 添加类和结构体

在 Server.h 中，定义本程序将要使用的宏、结构体和类：

```
#pragma once
#include "afxinet.h"
#include "string.h"
#include "stdlib.h"
/*****/
/* extern data */
/*****/
#define WSA_RECV 0 //Socket 状态
#define WSA_SEND 1
#define DATA_BUFSIZE 8192 //服务器能接收请求数据的最大长度
#define MAX_NAME_LEN 128
#define MAX_PWD_LEN 128
#define MAX_RESP_LEN 1024
#define MAX_REQ_LEN 256
#define MAX_ADDR_LEN 80
```

```

#define FTP_PORT          21                //FTP 控制端口
#define DATA_FTP_PORT    20                //FTP 数据端口
#define OPENING_AMODE 150
#define CMD_OK            200
#define OS_TYPE           215
#define FTP_QUIT          221                //注销
#define TRANS_COMPLETE 226
#define PASSIVE_MODE      227
#define LOGGED_IN         230                //登录成功
#define DIR_CHANGED       250
#define CURR_DIR          257
#define USER_OK           331
#define FILE_EXIST        350
#define ACCESS_DENY       450
#define NOT_IMPLEMENT     502
#define REPLY_MARKER      504
#define LOGIN_FAILED      530
#define CANNOT_FIND       550
#define MAX_FILE_NUM      1024
#define MODE_PORT         0
#define MODE_PASV         1
#define PORT_BIND         1821
/*****
/*  struct definition */
*****/
typedef struct {                                //记录 socket 信息
    CHAR          buffRecv[DATA_BUFSIZE];
    CHAR          buffSend[DATA_BUFSIZE];
    WSABUF        wsaBuf;
    SOCKET        socket;
    WSAOVERLAPPED wsaOverLapped;
    DWORD         dwBytesSend;
    DWORD         dwBytesRecv;
    int           SocketStatus;
    BOOL          userLoggedIn;
    CHAR          userCurrentDir[MAX_PATH];
    CHAR          userRootDir[MAX_PATH];
} SOCKET_INF, *LPSOCKET_INF;
typedef struct {                                //记录文件信息
    CHAR          szFileName[MAX_PATH];
    DWORD         dwFileAttributes;
    FILETIME      ftCreationTime;
    FILETIME      ftLastAccessTime;
    FILETIME      ftLastWriteTime;
    DWORD         nFileSizeHigh;
    DWORD         nFileSizeLow;

```

```

}FILE_INF, *LPFILE_INF;
struct CAccount{                                     //包含一个用户的基本信息
    CString username;
    CString password;
    CString directory;
};
typedef CArray< CAccount,CAccount& > AccountArray;
/*****/
/*    global function    */
/*****/
//一个全局线程函数
extern UINT ServerThread(LPVOID lpParameter);
class CServer
{
public:
    CServer(void);
    ~CServer(void);
public:
    AccountArray    m_RegisteredAccount;    //记录服务器注册用户
    BYTE            IpFild[4];              //服务 IP
    UINT            m_ServerPort;           //线程监听端口
    BOOL            m_bStopServer;          //记录服务器开启关闭信息
    SOCKET          sListen;                //监听 socket
    SOCKET          sDialog;                //会话 socket
public:
    void DivideRequest(char* request,char* command,char* cmdtail);
                                                //分离请求信息
    void ServerConfigInfo( AccountArray* accountArray,BYTE nFild[],UINT port);
                                                //服务器配置
    BOOL SendWelcomeMsg( SOCKET s);          //发送欢迎信息
    int Login( LPSOCKET_INF pSockInfo );     //客户端登录函数
    int DealWithCommand( LPSOCKET_INF socketInfo ); //命令处理函数
    int SendACK( LPSOCKET_INF socketInfo );   //发送相应消息到客户端
    int RecvRequest( LPSOCKET_INF socketInfo ); //接收请求函数
    //分隔的 IP 地址形式, 分解为一个 DWORD 形 IP (32 位 Internet 主机地址) 和一个
    //WORD (16 位 TCP 端口地址) 形端口, 分别由指针 pdwIpAddr 和 pwPort 返回
    int ConvertCommaAddrToDotAddr( char* commaAddr,LPDWORD pdwIpAddr, LPWORD pwPort );
    //将一个 32 位 Internet 主机地址和一个 16 位 TCP 端口地址转换为一个由 6 个被逗
    //号隔开的数字组成的 IP 地址序列
    int ConvertDotAddrToCommaAddr( char* dotAddr, WORD wPort ,char* commaAddr );
    int DataConnect( SOCKET& s, DWORD dwIp, WORD wPort, int nMode );
                                                //数据连接函数
    char* GetLocalAddress();                  //获取本地 IP 地址
    SOCKET AcceptConnectRequest( SOCKET& s ); //接收连接请求
    //将文件列表信息转换成字符串, 保存到 buffer 中
    int FileListToString( char* buffer, UINT nBufferSize, BOOL isListCommand );
    int GetFileList( LPFILE_INF fileInfo, int arraySize, const char* searchPath );

```



```

//获取文件列表
int DataSend( SOCKET s, char* buff, int buffSize );//数据发送
int CombindFileNameSize(const char* filename,char* filenamesize);
//将文件名及大小信息整合到一起
int ReadFileToBuffer(const char* szFileName,char* buff,int fileSize );
//将指定文件写入缓存
int DataRecv( SOCKET s, const char* fileName ); //数据接收, 写入文件
void GetRalativeDirectory(char *currDir,char*rootDir,char* ralaDir);

//获得相对路径
void GetAbsoluteDirectory(char* dir,char* currentDir,char* userCurrDir);
//获得绝对路径
void HostToNet(char* path); //将\\.\.格式转换为/./..格式
void NetToHost(char* path); //将/./..格式转换为\\.\.格式
int TryDeleteFile(char* deletedPath); //删除文件
BOOL IsPathExist(char* path); //检测路径是否合法
};

```

在 ftpSrverDlg.h 中包含头文件:

```

#include "stdlib.h"
#include "Server.h"

```

同时在类 CftpSrverDlg 中定义:

```

public:
    AccountArray m_AccountArray;
    CServer m_server;

```

在 ftpSrverDlg.cpp 中包含头文件:

```

#include "AccountDlg.h"
#include <stdio.h>

```

同时在 BOOL CftpSrverDlg::OnInitDialog()中添加初始化代码:

```

//初始化时, 为服务器添加几个固定用户
CAccount fixAccount;
fixAccount.username = "admin"; //管理员
fixAccount.password = "admin";
fixAccount.directory = "C:\\";
m_AccountArray.Add( fixAccount );
CLISTACCOUNT.AddString(fixAccount.username);
fixAccount.username = "usr"; //普通用户
fixAccount.password = "usr";
fixAccount.directory = "D:\\";
m_AccountArray.Add( fixAccount );
CLISTACCOUNT.AddString(fixAccount.username);
fixAccount.username = "anonymous"; //匿名用户
fixAccount.password = "";
fixAccount.directory = "C:\\Documents and Settings\\All Users\\Documents";
m_AccountArray.Add( fixAccount );
CLISTACCOUNT.AddString(fixAccount.username);

```

在初始化时, 预先在服务器上注册了三个默认的用户账户: 管理员、普通用户和匿名用户。

它们的用户名、密码和默认根目录见表 6.6。

表 6.6 默认账户

账 号 类 型	用 户 名	密 码	默认根目录
管理员	admin	admin	C:\
普通	usr	usr	D:\
匿名	anonymous	—	C:\Documents and Settings\All Users\Documents

本程序也要使用 Socket I/O 模型和临界区机制，为此还要再定义一些数据结构。

在 Server.cpp 中定义：

```
#include "ftpServer.h"
/*****
/*   global data   */
*****/
DWORD          g_dwTotalEventAmount = 0;    //总事件数
DWORD          g_index;
WSAEVENT       g_EventArray[WSA_MAXIMUM_WAIT_EVENTS];
//手动重置对象
LPSOCKET_INF   g_SocketArray[WSA_MAXIMUM_WAIT_EVENTS];
//新 SOCKET_INF 结构
CRITICAL_SECTION g_CriticalSection;        //临界区
//处理线程函数声明
UINT ProcessThreadIO( LPVOID lpParam );
```

以上这段代码读者暂时不需要理解，只要照着书将它们复制在程序中的对应位置即可。

6.3.2 FTP 服务器界面总控

先来看用户界面的总控代码。在软件主对话框界面，网管员可以启动 FTP 服务。在配置好服务 IP 和端口后，单击“启动服务”按钮，服务器启动。

“启动服务”按钮事件过程代码如下：

```
void CftpSrverDlg::OnStartServer()
{
    BYTE nFild[4];
    CString sIP;
    this->UpdateData();
    if(ServerIP.IsBlank())
    {
        AfxMessageBox("请配置 IP 地址!");
        return;
    }
    if(m_EditPort.IsEmpty())
    {
        AfxMessageBox("请配置服务端口!");
        return;
    }
}
```

```

ServerIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
sIP.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
UINT i = atoi(m_EditPort);
//配置服务器
m_server.ServerConfigInfo( &m_AccountArray,nFild,i );
//开启主线程，对端口进行监听
AfxBeginThread(ServerThread,LPVOID(&m_server));
//显示服务器开启信息
CLISTSERVERINFO.AddString("FTP 服务启动成功!");
CLISTSERVERINFO.AddString("IP " + sIP + " 端口" + m_EditPort);
CLISTSERVERINFO.AddString("服务运行中.....");
ServerIP.EnableWindow(false);
CEDITPORT.EnableWindow(false);
}

```

这里调用了 `ServerConfigInfo()` 函数获得管理员配置的 IP 和端口等参数，并开启 FTP 服务线程 `ServerThread`。管理员可以随时单击“停止服务”按钮，关闭 FTP 服务。

“停止服务”按钮的事件过程代码如下：

```

void CftpSrverDlg::OnStopServer()
{
    m_server.m_bStopServer = true;
    closesocket(m_server.sDialog);
    closesocket(m_server.sListen);
    CLISTSERVERINFO.AddString("服务已停止!");
    ServerIP.EnableWindow(true);
    CEDITPORT.EnableWindow(true);
}

```

停止服务时需要先后将会话 `Socket` 和监听 `Socket` 都关闭。程序中使用 `BOOL` 型变量 `m_bStopServer` 来指示服务的开关状态，这个变量是服务类 `CServer` 的成员变量，在 `Server.cpp` 的 `CServer` 构造方法中赋初值，代码如下：

```

CServer::CServer(void)
{
    m_bStopServer = FALSE;
}
CServer::~CServer(void) { }

```

本服务器提供新用户注册、编辑注册信息和删除用户的功能，这些配置都由服务器管理员完成。要编辑修改服务器上已有的用户信息，可在“用户登记”列表中选择要编辑的用户名，然后单击“编辑”按钮。

“编辑”按钮的事件过程代码如下：

```

void CftpSrverDlg::OnEditUsr()
{
    int selectedItemIndex = CLISTACCOUNT.GetCurSel();
    int accountToBeEditedIndex = 0;
    //判断用户操作是否合法
    if( selectedItemIndex == -1 )
    {

```

```

        if( CLISTACCOUNT.GetCount() != 0 )
        {
            AfxMessageBox(_T("请选择要编辑的用户.));
        }
        else
        {
            AfxMessageBox(_T("用户列表为空，无用户可供编辑"));
        }
    }
else
{
    //获取选中的用户账户名
    CString account;
    CLISTACCOUNT.GetText( selectedItemIndex, account );
    //查询这个账户
    int length = (int)m_AccountArray.GetSize();
    for( int i = 0 ; i < length ; i ++ )
    {
        if( m_AccountArray[i].username == account )//找到了
        {
            accountToBeEditedIndex = i;
            break;
        }
    }
    //弹出“用户”对话框
    CAccountDlg dlg;
    dlg.m_username = m_AccountArray[accountToBeEditedIndex].username;
    dlg.m_password = m_AccountArray[accountToBeEditedIndex].password;
    dlg.m_directory = m_AccountArray[accountToBeEditedIndex].directory;
    while( dlg.DoModal() == IDOK )
    {
        //检查编辑的内容是否合法
        if( dlg.m_username == "" || (dlg.m_password == "" &&
            dlg.m_username != "anonymous") || dlg.m_directory == "" )
        {
            AfxMessageBox(_T("用户名、密码或 FTP 根目录不能为空"));
            continue;
        }
        else
        {
            //判断账号是否已经存在
            int length = (int)m_AccountArray.GetSize();
            BOOL isAccountExisted = FALSE;
            for( int i = 0 ; i < length ; i ++ )
            {
                if( m_AccountArray[i].username == dlg.m_username &&
                    i != accountToBeEditedIndex )

```

```

        {
            AfxMessageBox(_T("该用户已经存在"));
            isAccountExisted = TRUE;
            break;
        }
    }
    if( isAccountExisted )
    {
        continue;
    }
    else if( AfxMessageBox(_T("确定你所做的修改吗?"),4 + 48) == 6 )
    {
        //保存修改内容
        m_AccountArray[accountToBeEditedIndex].username = dlg.m_username;
        m_AccountArray[accountToBeEditedIndex].password = dlg.m_password;
        m_AccountArray[accountToBeEditedIndex].directory = dlg.m_directory;
        //同步刷新账号列表
        CLISTACCOUNT.DeleteString( selectedItemIndex );
        CLISTACCOUNT.AddString(m_AccountArray[accountToBeEditedIndex].username);
        break;    //跳出循环
    }
}
//释放内存
account.Empty();
account.FreeExtra();
delete dlg;
}
}

```

这段代码的流程非常清晰，它考虑了各种可能的情况，包括：管理员忘了选择编辑对象、服务器上根本没有注册用户、遗漏了编辑用户的某项信息、账号已经存在、修改确认等。正是因为采用了将界面控制代码与 Socket 管理、协议实现等底层细节处理代码相分离的方法，才使我们有可能将精力集中在界面控制优化上，极大地提高了软件容忍用户误操作及应对各种意外情况的能力。这种良好的编程方法和习惯希望大家在平时就留心学习和积累。

管理员还可以在服务器上添加注册新用户。大家在学校上网时如果想向校园网 FTP 上传资料，一般都要先到校园网中心去申请一个账号，获得权限后方能操作。申请账号时，你的信息是由学校网管中心的老师用类似这样的方式配置到服务器上去的，在本例中是单击“添加用户”按钮。

“添加用户”按钮的事件过程代码如下：

```

void CftpSrverDlg::OnAddUsr()
{
    //当单击“添加用户”按钮时，通过“用户”对话框添加新的用户
    CAccountDlg accountDlg;
    //初始化账户信息为空
    accountDlg.m_username = "";
}

```

```

accountDlg.m_password = "";
accountDlg.m_directory = "";
while( accountDlg.DoModal() == IDOK )
{
    //检查输入是否合法
    if( accountDlg.m_username == "" || accountDlg.m_password == "" ||
        accountDlg.m_directory == "" )
    {
        AfxMessageBox(_T("账号、密码或 FTP 路径不能为空! "));
    }
    else
    {
        //检查添加的这个账号是否已经存在
        int length = (int)m_AccountArray.GetSize();
        BOOL isAccountExisted = FALSE;
        for( int i = 0 ; i < length ; i ++ )
        {
            if( m_AccountArray[i].username == accountDlg.m_username )
            {
                AfxMessageBox(_T("账号已经存在, 请使用其他账号.));
                isAccountExisted = TRUE;
                break;
            }
        }
        //如果账号已经存在
        if( isAccountExisted )
        {
            continue;
        }
        //如果该账号原先没有, 则添加
        else
        {
            CAccount newAccount;
            newAccount.username = accountDlg.m_username;
            newAccount.password = accountDlg.m_password;
            newAccount.directory = accountDlg.m_directory;
            m_AccountArray.Add( newAccount );
            //刷新列表, 显示新添加账号
            CLISTACCOUNT.AddString( newAccount.username );
            break;
        }
    }
}
//关闭“用户”对话框
delete accountDlg;
}

```

单击“添加用户”按钮后, 系统弹出“用户”对话框供管理员输入新用户的信息(用户名、

密码、根目录等), 同样, 程序也会自动验证输入的合法性。

管理员也可以随时删除已添加的用户, 只需在“用户登记”列表中选要删除的用户名, 单击“删除用户”按钮即可。

“删除用户”按钮的事件过程代码如下:

```
void CftpSrverDlg::OnDelUsr()
{
    //当单击“删除用户”按钮时, 删除账户列表中选中的用户信息
    int selectedIndex = CLISTACCOUNT.GetCurSel();
    if( selectedIndex == -1 )
    {
        if( CLISTACCOUNT.GetCount() != 0 )
        {
            AfxMessageBox(_T("请选择要删除的用户。"));
        }
        else
        {
            AfxMessageBox(_T("用户列表为空, 无用户可供删除"));
        }
    }
    else
    {
        //获得选中的用户名
        CString account;
        CLISTACCOUNT.GetText( selectedIndex, account );
        //查询并删除这个用户账号
        int length = (int)m_AccountArray.GetSize();
        for( int i = 0 ; i < length ; i ++ )
        {
            if( m_AccountArray[i].username == account )
            {
                m_AccountArray.RemoveAt( i );
                length = (int)m_AccountArray.GetSize();
                break;
            }
        }
        //用户列表中也删除对应这个账号的用户名
        CLISTACCOUNT.DeleteString( selectedIndex );
        //释放内存
        account.Empty();
        account.FreeExtra();
    }
}
```

现在, 服务器启动、关闭, 用户添加、编辑、删除这些基本功能都已具备。

主对话框“退出”按钮为“取消”按钮变换而来, 因此无须编写代码。“用户”对话框的“确定”、“取消”按钮也均为原默认“确定”、“取消”按钮, 无须程序员自己编写代码。至此, 界面总控部分的代码编写完毕。

6.3.3 FTP 服务流程的实现

当网络管理员在界面上配置好服务器地址和注册用户信息后，程序将这些信息提交给 ServerConfigInfo() 做进一步处理，实质是提交给服务器（类），因为 ServerConfigInfo() 方法是属于服务器类 CServer 的，它的代码位于源文件 Server.cpp 中。

ServerConfigInfo() 方法代码如下：

```
//服务器配置
void CServer::ServerConfigInfo( AccountArray* accountArray, BYTE nFild[], UINT port)
{
    //账户设置
    int size = (int)(*accountArray).GetCount();
    for( int i = 0 ; i < size ; i ++ )
    {
        m_RegisteredAccount.Add( (*accountArray)[i] );
    }
    //IP 设置
    for( int i = 0; i < 4; i ++ )
    {
        IpFild[i] = nFild[i];
    }
    //端口设置
    m_ServerPort = port;
    //设置服务器开启状态
    m_bStopServer = FALSE;
}
```

这段代码主要完成：载入已注册的用户账户列表，获取用户配置的 IP 及端口。其中服务器上注册的账户存储在一个由 CAccount 类组成的矩阵结构 CArray 中，通过调用其类对象 m_RegisteredAccount 的 Add() 方法添加新用户。获取配置信息后，FTP 服务就可以启动运行了。之后的程序代码都属于 FTP 服务本身的实现代码，自然都位于源文件 Server.cpp 中。服务对应的主线程为 ServerThread。

主线程 ServerThread 代码如下：

```
UINT ServerThread(LPVOID lpParameter)
{
    CServer * server = (CServer*)lpParameter; //服务器
    SOCKADDR_IN    addrInfo;                //设置监听地址、端口
    DWORD          dwFlags;                  //无实际意义，只是函数使用过程中的空白参数
    DWORD          dwRecvBytes;              //无实际意义，只是函数使用过程中的空白参数
    char           errorMsg[128];            //错误提示消息
    SOCKADDR_IN    ClientAddr;               //获取客户端地址信息
    int            ClientAddrLength;         //客户端地址信息 Size
    LPCTSTR        ClientIP;                //客户端 IP
    UINT           ClientPort;               //客户端 Port
    //初始化临界区
    InitializeCriticalSection( &g_CriticalSection );
```



```

//创建监听 socket (sListen)
if ( ( server->sListen = WSASocket(AF_INET,SOCK_STREAM,
                                0,NULL,0,WSA_FLAG_OVERLAPPED) ) == INVALID_SOCKET )
{
    ...//出错处理 ①
}
//配置 IP 及端口
CString sIP;
sIP.Format("%d.%d.%d.%d",server->IpFild[0],server->IpFild[1],server->IpFild[2],server->IpFild[3]);
addrInfo.sin_family = AF_INET;
addrInfo.sin_addr.S_un.S_addr = inet_addr(sIP);
addrInfo.sin_port = htons(server->m_ServerPort); //FTP_PORT;
//绑定监听端口 sListen, 指定通信对象
if ( bind(server->sListen, (PSOCKADDR)&addrInfo, sizeof(addrInfo) ) == SOCKET_ERROR )
{
    ...//出错处理 ②
}
//设置 sListen 为等待连接状态
if( listen( server->sListen, SOMAXCONN ) )
{
    ...//出错处理 ③
}
//创建会话 socket(sDialog)
if ( server->sDialog = WSASocket(AF_INET, SOCK_STREAM,
                                0, NULL, 0, WSA_FLAG_OVERLAPPED) ) == INVALID_SOCKET )
{
    ...//出错处理 ④
}
//创建第一个手动重置对象
if ( ( g_EventArray[0] = WSACreateEvent() ) == WSA_INVALID_EVENT )
{
    ...//出错处理 ⑤
}
//现在开始, 已经创建了一个事件
g_dwTotalEventAmount = 1;
//创建一个线程处理请求
AfxBeginThread( ProcessThreadIO,(LPVOID)server );
//创建循环, 不停地等待用户的连接
while( !server->m_bStopServer )
{
    //处理入站连接, 接收一个连接请求
    ClientAddrLength = sizeof(ClientAddr);
    if ( ( server->sDialog = accept(server->sListen,(sockaddr*)&ClientAddr,&ClientAddrLength)) ==
INVALID_SOCKET )
    {
        return 0;
    }
}

```

```

//获取客户端地址端口信息
ClientIP  = inet_ntoa( ClientAddr.sin_addr );
ClientPort = ClientAddr.sin_port;
//回传欢迎消息
if( !server->SendWelcomeMsg( server->sDialog ) )
{
    break;
}
//开始进入临界区，防止出错
EnterCriticalSection( &g_CriticalSection );
//创建一个新的 SOCKET_INF 结构处理接收的数据 socket
if( (g_SocketArray[g_dwTotalEventAmount] = (LPSOCKET_INF)
GlobalAlloc(GPTR,sizeof(SOCKET_INF))) == NULL )
{
    ...//出错处理 ⑥
}
//初始化新的 SOCKET_INF 结构
char buffer[DATA_BUFSIZE];
ZeroMemory(buffer, DATA_BUFSIZE);
//初始化 wsaBuf
g_SocketArray[g_dwTotalEventAmount]->wsaBuf.len = DATA_BUFSIZE;
g_SocketArray[g_dwTotalEventAmount]->wsaBuf.buf = buffer;
//初始化 socket
g_SocketArray[g_dwTotalEventAmount]->socket = server->sDialog;
//初始化 wsaOverLapped
memset( &(g_SocketArray[g_dwTotalEventAmount]->
wsaOverLapped),0,sizeof(OVERLAPPED) );
//初始化 buffRecv
memset( &(g_SocketArray[g_dwTotalEventAmount]->buffRecv), 0, DATA_BUFSIZE );
//初始化 buffSend
memset( &(g_SocketArray[g_dwTotalEventAmount]->buffSend), 0, DATA_BUFSIZE );
g_SocketArray[g_dwTotalEventAmount]->dwBytesSend = 0;
g_SocketArray[g_dwTotalEventAmount]->dwBytesRecv = 0;
//用户登录状态
g_SocketArray[g_dwTotalEventAmount]->userLoggedIn = FALSE;
//初始化预设状态
g_SocketArray[g_dwTotalEventAmount]->SocketStatus = WSA_RECV;
//创建事件
if( ( (g_SocketArray[g_dwTotalEventAmount]->wsaOverLapped.hEvent = g_EventArray[g_dwTotalEventAmount] =
WSACreateEvent()) == WSA_INVALID_EVENT )
{
    ...//出错处理 ⑦
}
//发出接收请求
dwFlags = 0;
if( WSAREcv( g_SocketArray[g_dwTotalEventAmount]->socket,
&(g_SocketArray[g_dwTotalEventAmount]->wsaBuf),

```

```

1,
&dwRecvBytes,
&dwFlags,
&(g_SocketArray[g_dwTotalEventAmount]->wsaOverLapped),
NULL) == SOCKET_ERROR )
{
if ( WSAGetLastError() != WSA_IO_PENDING )
{
//关闭 socket
closesocket( g_SocketArray[g_dwTotalEventAmount]->socket );
//关闭事件
WSACloseEvent( g_EventArray[g_dwTotalEventAmount] );
//弹出提示信息
...//出错处理 ⑧
}
}
//更新 g_EventArray[]事件总数目
g_dwTotalEventAmount++;
//离开临界区
LeaveCriticalSection( &g_CriticalSection );
//使第一个事件有信号, 使工作者线程处理其他的事件
if ( WSASetEvent( g_EventArray[0] ) == FALSE )
{
...//出错处理 ⑨
}
}
//关闭监听线程
server->m_bStopServer = FALSE;
//退出监听线程
return 0;
}

```

以上代码虽然比较长, 但大致还是可以看出其流程的:

初始化临界区→创建监听 sListen (WSASocket)→绑定监听端口 (bind())→监听 (listen())→创建会话 sDialog (WSASocket)→创建第一个手动重置对象 (WSACreateEvent)→创建工作者线程 PprocessThreadIO→While 循环等用户连接→sDialog 套接字 (accept())→向 sDialog 回传欢迎消息 (SendWelcomeMsg())→进入临界区→...→重叠操作 (SOCKET_INF 结构)→...→发出接收请求 (WSARecv())→关闭 Socket (closesocket()/WSACloseEvent())→离开临界区 (使第一个事件有信号 WSASetEvent())→工作者线程处理其他事件。

这个流程和基础的 Socket 编程流程相似, 只是多了临界区、重叠操作、事件对象等概念, 这些属于 Socket 编程的深入内容, 有兴趣的读者可以参考网络编程进阶类图书, 本书不深入展开。

以上程序段中还有好几处带有数字标号的出错处理段, 它们的代码大致相同, 为了节约篇幅, 特做标记后专门列出如下。

出错处理①:

```
sprintf( errorMsg, "ERROR:Failed to get a socket %d.POS:",
```

```

        ServerThread()\n", WSAGetLastError() );
AfxMessageBox( errorMsg,MB_OK,0 );
WSACleanup();      //①
return 0;

```

其他出错处理段 (②~⑨):

```

sprintf( errorMsg,"ERROR: ***** %d.POS:
        ServerThread()\n", WSAGetLastError() );
AfxMessageBox( errorMsg,MB_OK,0 );
return 0;

```

这些错误提示消息都用 WSAGetLastError()函数截获, 存储在一个字符数组 errorMsg[]中, 通过弹出对话框显示出来, 使用的程序代码形式上都完全一样, 只是显示错误信息的文本内容不同, 分别列于表 6.7。

表 6.7 错误信息

序 号	错 误 信 息
①	Failed to get a socket
②	bind() failed with error
③	listen() failed with error
④	Failed to get a dialog socket
⑤	WSACreateEvent() failed with error
⑥	GlobalAlloc() failed with error
⑦	WSACreateEvent() failed with error
⑧	WSARecv() failed with error
⑨	WSASetEvent failed with error

顺便给出服务器回传欢迎消息的 SendWelcomeMsg()方法的代码:

```

//发送欢迎消息
BOOL CServer::SendWelcomeMsg( SOCKET s )
{
    char* welcomeInfo = "220 Server ready.\r\n";
    if( send( s, welcomeInfo, (int)strlen(welcomeInfo), 0 )==SOCKET_ERROR )
    {
        AfxMessageBox( _T("Failed in sending welcome msg. POS:CServer::SendWelcomeMsg"));
        return FALSE;
    }
    return TRUE;
}

```

FTP 主线程启动后, 对于每一个发起连接请求的客户端线程, 都会创建一个对应的工作者线程为其提供服务。

工作者线程 ProcessThreadIO 代码如下:

```

UINT ProcessThreadIO( LPVOID lpParameter )
{
    DWORD          dwFlags;
    LPSOCKET_INF    pSocketInfo;      //单个 LPSOCKET_INF 结构
    DWORD           dwBytesTransferred; //数据传输字节数

```

```

DWORD            i;                                //index
CServer * server = (CServer*)lpParameter;          //服务器
char             errorMsg[128];                     //错误提示消息
//处理异步的 WSARecv、WSASend 等请求
while( TRUE )
{
    //调用重叠操作 (WSARecv(), WSARecvFrom(), WSASend(),
    //WSASendTo() 或 WSAIoctl()), 等待事件通知
    if ((g_index = WSAWaitForMultipleEvents(g_dwTotalEventAmount, g_EventArray,FALSE, WSA_
INFINITE, FALSE)) == WSA_WAIT_FAILED )
    {
        sprintf( errorMsg, "Error WSAWaitForMultipleEvents()
                        failed with error:%d\n",WSAGetLastError() );
        AfxMessageBox( errorMsg, MB_OK|MB_ICONSTOP );
        return 0;
    }
    //对第一个事件, 不做处理
    if ( (g_index - WSA_WAIT_EVENT_0) == 0 )
    {
        //将第一个数据设置为无信号状态
        WSAResetEvent(g_EventArray[0]);
        continue;
    }
    //取出有信号状态的事件对应的 SOCKET_INF 结构
    pSocketInfo = g_SocketArray[g_index - WSA_WAIT_EVENT_0];
    //将此事件设为无信号状态
    WSAResetEvent(g_EventArray[g_index - WSA_WAIT_EVENT_0]);
    //事件不可用的情况, dwBytesTransferred == 0 表示对方已关闭连接
    if ( WSAGetOverlappedResult( pSocketInfo->socket,
                                &(pSocketInfo->wsaOverLapped),
                                &dwBytesTransferred,
                                FALSE,
                                &dwFlags ) == FALSE || dwBytesTransferred == 0 )
    {
        //重叠操作失败, 关闭套接口
        closesocket( pSocketInfo->socket );
        //释放套接口资源
        GlobalFree( pSocketInfo );
        //关闭事件
        WSACloseEvent( g_EventArray[g_index - WSA_WAIT_EVENT_0] );
        //进入临界区
        EnterCriticalSection( &g_CriticalSection );
        //数组移动, 删除事件
        if ( (g_index - WSA_WAIT_EVENT_0) + 1 != g_dwTotalEventAmount )
        {
            for ( i=(g_index - WSA_WAIT_EVENT_0); i < g_dwTotalEventAmount; i++ )
            {

```

```

        g_EventArray[i] = g_EventArray[i+1];
        g_SocketArray[i] = g_SocketArray[i+1];
    }
}
//事件总数减 1
g_dwTotalEventAmount--;
//离开临界区
LeaveCriticalSection(&g_CriticalSection);
//事件不可用，接下来的代码不用执行
continue;
}
//事件可用，而且已经有数据传递
if( pSocketInfo->SocketStatus == WSA_RECV )
{
    //复制数据
    memcpy( &pSocketInfo->buffRecv[pSocketInfo->dwBytesRecv],
            pSocketInfo->wsaBuf.buf,dwBytesTransferred );
    //接收字节数
    pSocketInfo->dwBytesRecv = dwBytesTransferred;
    //检测接收到的数据，要保证最后是\r\n
    if( pSocketInfo->dwBytesRecv > 2&& pSocketInfo->buffRecv[pSocketInfo->dwBytesRecv-2]
== '\r'&& pSocketInfo->buffRecv[pSocketInfo->dwBytesRecv-1] == '\n' )
    {
        /*-----关键代码-----*/
        //检测用户是否已经登录
        if( !pSocketInfo->userLoggedIn ) //未登录:用户登录
        {
            if( server->Login(pSocketInfo) == LOGGED_IN )//①
            {
                //更新用户登录状态
                pSocketInfo->userLoggedIn = TRUE;
                //设置 FTP 根目录（返回非 0 数据表示成功；否则返回 0）
                if( SetCurrentDirectory( pSocketInfo->userCurrentDir )==0 )
                {
                    sprintf( errorMsg,"ERROR: POS:
ServerThread()\nSetCurrentDirectory()
failed with error: %d.", GetLastError() );
                    AfxMessageBox( errorMsg,MB_OK|MB_ICONERROR );
                }
            }
        }
        else
            //已登录:根据命令做事情
        {
            if( server->DealWithCommand( pSocketInfo )==FTP_QUIT ) //②
            {
                break;
            }
        }
    }
}

```

```

    }
    /*-----关键代码-----*/
    //缓冲区清除
    memset( pSocketInfo->buffRecv,0,sizeof(pSocketInfo->buffRecv) );
    pSocketInfo->dwBytesRecv = 0;
}
}
else //如果 Socket 处于发送状态
{
    pSocketInfo->dwBytesSend += dwBytesTransferred;
}
//继续接收以后到来的数据
if( server->RecvRequest( pSocketInfo ) == -1 ) //接收数据失败
{
    return -1;
}
}
//结束线程
return 0;
}

```

上段代码中用到一些大家没见过的 Socket 函数：WSAWaitForMultipleEvents()、WSAResetEvent()、WSAGetOverlappedResult()……它们是实现 Socket I/O 模型重叠操作机制的，属于 Socket 编程的深层内容，暂且不去管它。

程序里用到一个 LPSOCKET_INF 结构的变量 pSocketInfo，LPSOCKET_INF 是个很重要的数据结构，在本例中用于保存各 Socket 的详细信息。上面程序段中，使用“pSocketInfo->SocketStatus”来判断重叠操作中事件的可用性。若事件不可用，跳出循环不执行；若可用（pSocketInfo 指示的 Socket 状态为 WSA_RECV），则继续执行后面的代码。

这段程序中，读者只要关注其中醒目标注的“/*----关键代码----*/”就行了，它是本程序由服务流程代码通向协议实现代码的入口。先用 pSocketInfo->userLoggedIn 判断用户的登录状态：①若尚未登录，调用 Login()方法登录；②若已登录，则执行 DealWithCommand()方法，根据 FTP 命令转入对应处理过程。

工作者线程接收数据时采用 RecvRequest()方法。

RecvRequest()方法代码如下：

```

int CServer::RecvRequest( LPSOCKET_INF socketInfo )
{
    //本地变量声明
    DWORD dwRecvBytes = 0;
    DWORD dwFlags = 0;
    char errorMsg[128];
    //设置 socket 结构状态
    socketInfo->SocketStatus = WSA_RECV;
    //初始化 wsaOverLapped 结构
    memset( &(socketInfo->wsaOverLapped), 0,sizeof(WSAOVERLAPPED) );
    socketInfo->wsaOverLapped.hEvent = g_EventArray[g_index - WSA_WAIT_EVENT_0];
    //设置接收请求数据长度

```

```

socketInfo->wsaBuf.len = DATA_BUFSIZE;
//开始接收请求数据
if ( WSARecv(socketInfo->socket,&(socketInfo->wsaBuf),1,&dwRecvBytes,
    &dwFlags,&(socketInfo->wsaOverLapped),NULL) == SOCKET_ERROR )
{
    if ( WSAGetLastError() != ERROR_IO_PENDING )
    {
        sprintf( errorMsg,"WSARecv() failed with error: %d.POS:
            CServer::RecvRequest()\n", WSAGetLastError() );
        AfxMessageBox( errorMsg,MB_OK,MB_ICONERROR );
        return -1;
    }
}
//接收正常完成
return 0;
}

```

接收数据采用的还是 Socket WSARecv()函数，同时将 Socket 结构状态置为 WSA_RECV 是为了方便前面工作者线程随时识别。

本节列出的 FTP 服务流程的代码，其实质仍然是 Socket 编程，只是涉及 Socket 编程的深入内容，因此看上去较为复杂。请将本节的全部代码按要求复制到程序的对应位置，然后只需知道：它是由一个主线程及其创建的许多工作者子线程密切配合，共同为客户线程提供 FTP 服务的，仅此足矣。

6.3.4 FTP 协议的实现

本节所列的是 FTP 实现部分的代码，大家可对照 6.1.4 节的两张表（表 6.1 和表 6.2）来阅读以下的程序。这些代码也写在源文件 Server.cpp 中。

Login()方法代码如下：

```

//用户登录函数
int CServer::Login( LPSOCKET_INF socketInfo )
{
    //静态变量
    static char username[MAX_NAME_LEN];
    static char password[MAX_PWD_LEN];
    //本地变量
    int loginResult = 0;
    //获取用户名
    if( strstr(strupr(socketInfo->buffRecv),"USER") || strstr(strlwr(socketInfo->buffRecv),"user") )
    {
        sprintf( username, "%s", socketInfo->buffRecv+strlen("user")+1 );
        strtok( username, "\r\n" );
        sprintf( socketInfo->buffSend, "%s", "331 User name ok,need password.\r\n" );
        if( SendACK( socketInfo ) == -1 )
        {
            AfxMessageBox( _T("Failed in sending ACK. POS:CServer::Login()") );

```



```

        return -1;
    }
    //成功获取用户名
    return USER_OK;
}
//获取用户登录密码
if( strstr(strupr(socketInfo->buffRecv),"PASS") || strstr(strlwr(socketInfo->buffRecv),"pass") )
{
    sprintf( password,"%s",socketInfo->buffRecv+strlen("pass")+1 );
    strtok( password, "\r\n" );
    //判断用户名和密码的正确性
    int size = (int)m_RegisteredAccount.GetCount();
    BOOL isAccountExisted = FALSE;
    CString user = (CString)username;
    CString pwd = (CString)password;
    for( int i=0 ; i<size ; i++ )
    {
        if( (m_RegisteredAccount[i].username.CompareNoCase(user)==0
            &&m_RegisteredAccount[i].password.CompareNoCase(pwd)==0)
            ||(m_RegisteredAccount[i].username.CompareNoCase(user)==0
            && user == "ANONYMOUS" ) )
        {
            //找到符合的用户账号
            isAccountExisted = TRUE;
            //获得当前用户预设的 FTP 根目录
            strcpy( socketInfo->userCurrentDir,m_RegisteredAccount[i].directory );
            strcpy( socketInfo->userRootDir,m_RegisteredAccount[i].directory );
            break;
        }
    }
    //账户存在与否相应的反馈信息
    if( isAccountExisted )
    {
        sprintf( socketInfo->buffSend,"230 User:%s logged in.proceed.\r\n",username );
        loginResult = LOGGED_IN;
    }
    else
    {
        sprintf( socketInfo->buffSend,"530 User:%s cannot logged in,
            Wrong username or password.Try again.\r\n",username );
        loginResult = LOGIN_FAILED;
    }
    //发送 ACK
    if( SendACK( socketInfo ) == -1 )
    {
        AfxMessageBox( _T("Failed in sending ACK. POS:CServer::Login()") );
        return -1;
    }
}

```

```

    }
}
//用户登录成功
return loginResult;
}

```

这里，服务器对接收到的客户端发来的信息按协议规范进行处理。

通用的处理形式为：

```
strstr(strupr(socketInfo->buffRecv),"XXX")||strstr(strlwr(socketInfo->buffRecv),"xxx")
```

其中，“XXX”和“xxx”表示某个FTP命令ASCII码字符的大写和小写形式（本例设计的服务器对客户端传来的同一个命令的大、小写形式都能够识别）。

这里用到几个C++的字符串处理函数，简单介绍一下：**strupr()**将字符串转换为大写形式；**strlwr()**将字符串转换为小写形式；**strstr()**在字符串中查找指定字符串的第一次出现。用**strupr()**和**strlwr()**对客户端发来的命令字符串进行规范（统一为全部大写或全部小写），然后用**strstr()**定位到相应的命令码，再做进一步操作。

在找到用户名后，向客户端返回“331”应答，由6.1.4节的应答码表（表6.2）可知，3打头的应答码表示“肯定中介应答。该命令已被接收，但另一个命令必须被发送”，结合程序可知，这里是表示“找到用户名后向客户端索要这个用户名对应的密码”，即要求客户端接着发送PASS命令给出密码。客户端发来密码后，若验证成功，则返回“230”，由表6.2可知，这个应答表示“肯定完成应答。一个新命令可以发送”，即验证成功了，允许用户登录到服务器接着进行下面的操作；若验证未通过，返回的应答码“530”表示“永久性否定完成应答。命令不被接收，并且不再重试”，也就意味着登录失败。

用户登录成功后，客户端与服务器通过控制连接进行交互。客户端发出命令，服务器给出应答，根据不同的命令，服务器要做不同处理，从而完成FTP服务的各项功能，处理命令用DealWithCommand()方法。

DealWithCommand()方法代码如下：

```

int CServer::DealWithCommand( LPSOCKET_INF socketInfo )
{
    //变量定义
    static SOCKET sDialog      =   INVALID_SOCKET;
    static SOCKET sAccept      =   INVALID_SOCKET;
    static BOOL isPasvMode     =   FALSE;           //判断是否为被动传输模式
    int  operationResult =   0;                     //记录某操作过程结果
    char  command[20];          //记录命令
    char  cmdtail[MAX_REQ_LEN]; //记录命令后面的内容
    char  currentDir[MAX_PATH]; //记录当前FTP目录
    char  relativeDir[128];      //记录当前目录的相对目录
    const char* szOpeningAMode = "150 Opening ASCII mode data connection for ";
    //客户端指定通信IP(4个“.”分隔的IP转换为相应无符号长整型)
    static DWORD dwIpAddr = 0;
    static WORD  wPort     = 0;           //客户端指定通信端口
    //解析请求，分离出命令存储到 cammand\命令后面跟随的内容，存储到 cmdtail 中
    DivideRequest( socketInfo->buffRecv,command,cmdtail );
    //检测是否为空
    if( command[0] == '\0' )

```

```

{
    return -1;
}
//-----开始处理命令-----//
//为数据连接指定一个 IP 地址和本地端口, PORT—data port (改变默认数据端口)
if( strstr(command,"PORT") )
{
    ...//①
}
//告诉服务器在一个非标准端口上收听数据连接, PASV—passive 将 server_DTP 设置为
//“被动”状态, 使其在数据端口侦听而不是启动一个数据连接
if( strstr( command,"PASV") )
{
    ...//②
}
//让服务器给客户端发送一份目录列表: NLST—name list;LIST—list
if( strstr( command, "NLST") || strstr( command,"LIST") )
{
    ...//③
}
//让服务器给客户端传送一份在路径名中指定的文件的副本 (其实就是 FTP 下载功能实现)
//RETR-retrieve:该命令使服务器 DTP 传输路径名中指定的文件副本到数据连接另一端的
//服务器或用户 DTP
if( strstr( command, "RETR") )
{
    ...//④
}
//STOR-store:该命令使服务器 DTP 接收经数据连接传输过来的数据,
//并将其作为文件存在服务器上 (实际就是 FTP 上传功能)
if( strstr( command, "STOR") )
{
    ...//⑤
}
//终止 USER。如果没有正在进行文件传输, 服务器将关闭控制连接; 如果文件传输
//正在进行, 该连接仍然打开直至结束响应, 然后服务器将其关闭: QUIT-logout
if( strstr( command,"QUIT" ) )
{
    ...//⑥
}
//PWD:在应答中返回当前工作目录的名称
if( strstr( command,"XPWD" ) || strstr( command,"PWD") )
{
    ...//⑦
}
//CDUP: 退回上一级目录
if( strstr(command,"CDUP") )
{

```

```

        ...//⑧
    }
    //CWD:把当前目录改为远程文件系统的指定路径,而无须改变登录、账号信息或传输参数
    //CWD: 改变工作目录
    if( strstr( command,"CWD" ) )
    {
        ...//⑨
    }
    //删除文件: DELE——delete, XRMD child, 删除名为“child”的目录; RMD——删除目录
    if( strstr( command,"DELE" ) )
    {
        ...//⑩
    }
    //确定数据的传输方式
    if( strstr( command,"TYPE" ) )
    {
        ...//⑪
    }
    //Restart 命令:标志文件内的数据点,将从这个点开始继续传送文件
    if( strstr( command,"REST" ) )
    {
        ...//⑫
    }
    //其余都是无效的命令
    sprintf(socketInfo->buffSend,"500 %s' Unknown command.\r\n",command);
    if( SendACK( socketInfo ) == -1 )
    {
        AfxMessageBox( _T("SendACK() failed.CMD:otherwise"),MB_OK|MB_ICONERROR );
        return -1;
    }
    return operationResult;
}

```

为使代码布局更合理,这里先不详细列出各个命令的具体实现代码,而是在它们的对应位置代之以①~⑫的标号,后面将按照大家平时使用 FTP 时所接触到的一些典型应用,分段详细列出各自的处理代码。

实际中使用的 FTP 一般都具备这几项功能:下载文件、上传文件、删除文件、浏览和操作服务器上的目录结构(包括显示服务器上的目录列表、改变工作目录、返回上一级目录等),下面逐一介绍。

1) 下载文件

使用命令“RETR”下载服务器上的文件,该命令将指定路径的文件复制到客户端,而服务器上文件的状态和内容不受影响。

处理过程④:

```

if( isPasvMode )
{
    //接收连接请求,并新建会话 socket

```

```

        sDialog = AcceptConnectRequest( sAccept );
    }
    char fileNameSize[MAX_PATH];
    //返回获取文件大小、文件名及大小信息，保存在 fileNameSize 中
    int nFileSize = CombindFileNameSize( cmdtail,fileNameSize );
    sprintf( socketInfo->buffSend,"%s%s.\r\n",szOpeningAMode,fileNameSize );
    if( SendACK( socketInfo ) == -1 )
    {
        AfxMessageBox( _T("SendACK() failed.CMD:RETR"),MB_OK|MB_ICONERROR );
        return -1;
    }
    char* buffer = new char[nFileSize];
    if( buffer == NULL )
    {
        AfxMessageBox( _T("分配缓存失败!\n") );
        return -1;
    }
    //把指定文件写入缓存并返回总共读取的字节数
    if( ReadFileToBuffer( cmdtail,buffer, nFileSize ) == nFileSize )
    {
        //处理 Data FTP 连接
        Sleep( 10 );
        if( isPasvMode )
        {
            DataSend( sDialog,buffer,nFileSize );
            closesocket( sDialog );
        }
        else
        {
            //创建连接
            if( DataConnect( sAccept,dwIpAddr,wPort,MODE_PORT ) == -1 )
            {
                return -1;
            }
            //发送数据
            DataSend( sAccept, buffer, nFileSize );
            closesocket( sAccept );
        }
    }
    //清除缓存
    if( buffer != NULL )
    {
        delete[] buffer;
    }
    sprintf( socketInfo->buffSend,"%s","226 Transfer complete.\r\n" );
    if( SendACK( socketInfo ) == -1 )
    {

```

```

        AfxMessageBox( _T("SendACK() failed.CMD:RETR"),MB_OK|MB_ICONERROR );
        return -1;
    }
    return TRANS_COMPLETE;

```

程序先向客户端返回该文件的文件名及大小信息,然后将指定文件写入缓存,最后发送文件。这里要用到布尔型变量 `isPasvMode`, 它用来判断服务器线程的工作模式究竟是主动还是被动: 若为被动模式,说明数据连接已经由客户端创建好了,服务器只需直接发送文件即可;若为主动模式,则在发送文件之前,还要用函数 `DataConnect()`创建数据连接。

2) 上传文件

使用命令“`STOR`”向服务器上传文件。该命令向服务器传输文件,若文件已存在,则原文件被覆盖;若文件不存在,则新建文件。

处理过程⑤:

```

//如果是被动模式,首先要接收连接,创建会话 socket
if( isPasvMode )
{
    sDialog = AcceptConnectRequest( sAccept );
}
if( cmdtail[0]!='\0' )
{
    return -1;
}
sprintf( socketInfo->buffSend,"%s%s.\r\n",szOpeningAMode,cmdtail );
if( SendACK( socketInfo ) == -1 )
{
    AfxMessageBox( _T("SendACK() failed.CMD:STOR"),MB_OK|MB_ICONERROR );
    return -1;
}
//处理 FTP 数据连接
if( isPasvMode )
{
    //接收数据
    DataRecv( sDialog,cmdtail );
}
else
{
    if( DataConnect( sAccept,dwIpAddr,wPort,MODE_PORT ) == -1 )
    {
        return -1;
    }
    //接收数据
    DataRecv( sAccept, cmdtail );
}
sprintf( socketInfo->buffSend,"%s","226 Transfer complete.\r\n" );
if( SendACK( socketInfo ) == -1 )
{
    AfxMessageBox( _T("SendACK() failed.CMD:STOR"),MB_OK|MB_ICONERROR );

```

```

        return -1;
    }
    return TRANS_COMPLETE;

```

与下载文件一样,根据主动和被动两种模式分别执行操作,其中主动模式需要由服务器自己建立数据连接。两种模式均调用 `DataRecv()` 函数接收文件。

3) 删除文件

使用命令“DELE”删除服务器上的文件,该命令删除指定路径下的文件。用户进程负责对删除的提示,本例为弹出一个消息对话框让用户确认。

处理过程⑩:

```

//获得应该转到的目录
GetAbsoluteDirectory( cmdtail,socketInfo->userCurrentDir,currentDir );
//获得相对路径
GetRelativeDirectory( currentDir,socketInfo->userRootDir,relativeDir );
//将\\.\格式转换为../格式,以便响应客户端
HostToNet( relativeDir );
//尝试删除文件或文件夹
BOOL isDELE = TRUE;
operationResult = TryDeleteFile( currentDir );
//根据删除结果,设置响应消息 ACK
if( operationResult==DIR_CHANGED )
{
    sprintf( socketInfo->buffSend,"250 File '%s' has been deleted.\r\n",relativeDir );
}
else if( operationResult==ACCESS_DENIED )
{
    sprintf( socketInfo->buffSend,"450 File '%s' can't be deleted.\r\n",relativeDir );
}
if( SendACK( socketInfo ) == -1 )
{
    AfxMessageBox( _T("SendACK() failed.CMD:DELE"),MB_OK|MB_ICONERROR );
    return -1;
}
return operationResult;

```

这里主要调用 `TryDeleteFile` 函数删除文件,根据删除操作的不同结果向客户端返回不同的应答。返回的应答是按照 FTP 协议的标准设置的,参见表 6.2。

4) 显示服务器目录列表

要使用户能够顺利操作服务器上的文件(就像操作本地计算机的资源一样),服务器还必须向客户端返回其上的文件目录列表,使客户端软件能够像本地资源管理器一样显示这些目录。FTP 协议用 `NLST` 或 `LIST` 命令完成这项功能。

处理过程③:

```

if( isPasvMode )
{
    //接收连接请求,并新建会话 socket
    sDialog = AcceptConnectRequest( sAccept );
}

```

```

if( !isPasvMode )                //主动传输
{
    sprintf( socketInfo->buffSend,"%s/bin/ls.\r\n",szOpeningAMode );
}
else                             //被动传输
{
    strcpy( socketInfo->buffSend,"125 Data connection already open; Transfer starting.\r\n" );
}
//发送应答 ACK
if( SendACK( socketInfo ) == -1 )
{
    AfxMessageBox( _T("SendACK() failed.CMD:NLST/LIST"),MB_OK|MB_ICONERROR );
    return -1;
}
//取得文件列表信息，并转换成字符串
BOOL isListCommand = strstr( command,"LIST" ) ? TRUE : FALSE;
char buffer[DATA_BUFSIZE];
UINT nStrLen = FileListToString( buffer,sizeof(buffer),isListCommand);
//主动与被动传输模式之间的差别
if( !isPasvMode )                //主动传输模式
{
    if( DataConnect( sAccept,dwIpAddr,wPort,MODE_PORT ) == -1 )
    {
        AfxMessageBox( _T("DataConnect() failed.CMD:NLST/LIST"),MB_OK|MB_ICONERROR );
        return -1;
    }
    if( DataSend( sAccept, buffer,nStrLen ) == -1 )
    {
        AfxMessageBox( _T("DataSend() failed.CMD:NLST/LIST"),MB_OK|MB_ICONERROR );
        return -1;
    }
    closesocket(sAccept);
}
else                             //被动传输模式
{
    DataSend( sDialog,buffer,nStrLen );
    closesocket( sDialog );
}
//发送应答 ACK
sprintf( socketInfo->buffSend,"%s","226 Transfer complete.\r\n" );
if( SendACK( socketInfo ) == -1 )
{
    AfxMessageBox( _T("SendACK() failed.CMD:NLST/LIST"),MB_OK|MB_ICONERROR );
    return -1;
}
return TRANS_COMPLETE;

```


调用函数 `FileListToString()` 取得文件列表信息，并转换成字符串发送给客户端。

5) 改变工作目录

为了让用户能够自如地在服务器的不同目录之间进行切换访问，需实现改变工作目录的功能，使用命令 `CWD`。

处理过程⑨：

```
//检测是否为根目录
if( cmdtail=="\" )
{
    strcpy( cmdtail,"\" );
}
//获得应该转到的目录
GetAbsoluteDirectory( cmdtail,socketInfo->userCurrentDir,currentDir );
//检测当前目录是否在根目录之上
if( strlen(currentDir) < strlen(socketInfo->userRootDir) )
{
    //设置响应消息
    sprintf( socketInfo->buffSend,"550 'CWD' command failed,'%s' User no power.\r\n",cmdtail );
}
else if( SetCurrentDirectory( currentDir )==0 ) //设置当前目录
{
    //设置响应消息
    sprintf( socketInfo->buffSend,"550 'CWD' command failed,'%s':
                                No such file or Directory.\r\n",cmdtail );

    //记录处理结果
    operationResult = CANNOT_FIND;
}
else
{
    //更改用户当前目录信息
    strcpy( socketInfo->userCurrentDir,currentDir );
    //获得相对路径
    relativeDir[0] = '\0';
    GetRelativeDirectory(currentDir,socketInfo->userRootDir,relativeDir);
    //将\.\.格式转换为../..格式，以便响应客户端
    HostToNet( relativeDir );
    //设置响应消息
    sprintf( socketInfo->buffSend,"250 'CWD' command successful,
                                '%s' is current directory.\r\n",relativeDir );

    //记录处理结果
    operationResult = DIR_CHANGED;
}
//发送应答 ACK
if( SendACK( socketInfo ) == -1 )
{
    AfxMessageBox( _T("SendACK() failed.CMD:CWD"),MB_OK|MB_ICONERROR );
    return -1;
}
```

```

}
return operationResult;

```

6) 返回上一级目录

用户在切换目录的过程中还可以随时回过头去浏览先前访问过的上一级目录, 这个功能是 Windows 环境窗口资源浏览的标准功能, 绝大多数的 FTP 服务器也有这样的功能。使用命令 CDUP 返回上一级目录。

处理过程⑧:

```

//检测是否为根目录
if( cmdtail[0]=='\0' )
{
    strcpy(cmdtail,"\\");
}
//获得应该转变到的目录
char szSetDir[MAX_PATH];
strcpy( szSetDir,".." );
//设置当前目录
if( SetCurrentDirectory( szSetDir )==0 )
{
    sprintf( socketInfo->buffSend,"550 '%s' No such file or Directory.\r\n",szSetDir );
    //记录处理结果
    operationResult = CANNOT_FIND;
}
else
{
    //获得当前目录
    GetCurrentDirectory( MAX_PATH,currentDir );
    //获得相对路径
    relativeDir[0] = '\0';
    GetRelativeDirectory(currentDir,socketInfo->userRootDir,relativeDir);
    //将\\.\\.格式转换为/\\.\\.格式, 以便响应客户端
    HostToNet( relativeDir );
    sprintf( socketInfo->buffSend,"250 'CDUP' command successful,
                                     '%s' is current directory.\r\n",relativeDir );
    //记录处理结果
    operationResult = DIR_CHANGED;
}
//发送应答 ACK
if( SendACK( socketInfo ) == -1 )
{
    AfxMessageBox( _T("SendACK() failed.CMD: CDUP"),MB_OK|MB_ICONERROR );
    return -1;
}
return operationResult;

```

7) 其他常用的操作

除了以上 6 种操作外, 本例还实现了其他一些常用的 FTP 操作, 以下分别列出其处理过程代码。

PORT 命令，处理过程①（向服务器发送客户端数据连接端口）代码如下：

```
//0: 成功; -1: 失败
//将 “,” 分隔的 IP 地址形式分解为一个 DWORD 型 IP (32 位 Internet 主机地址) 0 (dwIpAddr)
//和一个 WORD (16 位 TCP 端口地址) 型端口 (wPort)
if( ConvertCommaAddrToDotAddr( socketInfo->buffRecv+strlen("PORT")+1,&dwIpAddr,&wPort) == -1 )
{
    AfxMessageBox( _T("Failed in ConvertCommaAddrToDotAddr().CMD:PORT") );
    return -1;
}
//发送消息响应客户端
sprintf( socketInfo->buffSend,"%s","200 PORT Command successful.\r\n" );
if( SendACK( socketInfo ) == -1 )
{
    AfxMessageBox( _T("SendACK() failed.CMD:PORT"),MB_OK|MB_ICONERROR );
    return -1;
}
//更新传输模式
isPasvMode = FALSE;
return CMD_OK;
```

PASV 命令，处理过程②（将服务器设置为“被动”状态）代码如下：

```
//0: 成功; -1: 失败
if( DataConnect( sAccept, htonl(INADDR_ANY), PORT_BIND, MODE_PASV ) == -1 )
{
    AfxMessageBox( "DataConnect() failed.CMD:PASV",MB_OK|MB_ICONERROR );
    return -1;
}
//获取 6 个逗号隔开的数字组成的 IP 序列
char szCommaAddress[40];
if( ConvertDotAddrToCommaAddr( GetLocalAddress(),PORT_BIND,szCommaAddress )==-1 )
{
    AfxMessageBox( _T("Failed in ConvertCommaAddrToDotAddr().CMD:PASV") );
    return -1;
}
//设置应答 ACK
sprintf( socketInfo->buffSend,"227 Entering Passive Mode (%s).\r\n",szCommaAddress );
//发送应答 ACK
if( SendACK( socketInfo ) == -1 )
{
    AfxMessageBox( _T("SendACK() failed.CMD:PASV"),MB_OK|MB_ICONERROR );
    return -1;
}
//更新传输模式
isPasvMode = TRUE;
return PASSIVE_MODE;
```

QUIT 命令，处理过程⑥（退出登录）代码如下：

```
sprintf( socketInfo->buffSend,"%s","221 Goodbye.\r\n" );
```

```

if( SendACK( socketInfo ) == -1 )
{
    AfxMessageBox( _T("SendACK() failed.CMD:QUIT"),MB_OK|MB_ICONERROR );
    return -1;
}
return FTP_QUIT;

```

PWD 命令，处理过程⑦（返回当前工作目录）代码如下：

```

//获取当前目录
GetCurrentDirectory( MAX_PATH,currentDir );
//获取当前目录的相对目录
char tempStr[128];
GetRelativeDirectory(currentDir,socketInfo->userRootDir,tempStr);
HostToNet( tempStr );
//发送应答 ACK
sprintf( socketInfo->buffSend,"257 \"%s\" is current directory.\r\n",tempStr );
if( SendACK( socketInfo ) == -1 )
{
    AfxMessageBox( _T("SendACK() failed.CMD:XPWD/PWD"),MB_OK|MB_ICONERROR );
    return -1;
}
return CURR_DIR;

```

TYPE 命令，处理过程⑩（确定文件的传输类型，可指定 ASCII、EBCDIC、Image、本地类型文件等参数）代码如下：

```

if( cmdtail[0] == '\0' )
{
    strcpy( cmdtail,"A" );
}
sprintf(socketInfo->buffSend,"200 Type set to %s.\r\n",cmdtail );
if( SendACK( socketInfo ) == -1 )
{
    AfxMessageBox( _T("SendACK() failed.CMD:TYPE"),MB_OK|MB_ICONERROR );
    return -1;
}
return CMD_OK;

```

REST 命令，处理过程⑫（断点续传）代码如下：

```

sprintf( socketInfo->buffSend,"504 Reply marker must be 0.\r\n");
if( SendACK( socketInfo ) == -1 )
{
    AfxMessageBox( _T("SendACK() failed.CMD:REST"),MB_OK|MB_ICONERROR );
    return -1;
}
return REPLY_MARKER;

```

6.3.5 FTP 实现辅助代码

大家一定发现了：上面的协议实现代码中，有不少操作是通用的，如 SendACK()函数，凡是需要向客户端返回应答信息时都会用到它。此外，还有一些操作也是在实现多种 FTP 功能时都要用到的，如 DataSend()、DataConnect()、DataRecv()、GetAbsoluteDirectory()和 GetRelativeDirectory()等。

下面列出这些通用方法的源代码供读者参考。

1) 网络连接管理类

在协议工作的过程中，对网络连接的管理（包括接收客户端连接请求、建立数据连接）是协议通信主体之间顺利交互的保证。

AcceptConnectRequest()（接收连接请求）代码如下：

```
SOCKET CServer::AcceptConnectRequest(SOCKET &s)
{
    SOCKET sDialog = accept( s,NULL,NULL );
    if( sDialog == INVALID_SOCKET )
    {
        AfxMessageBox( _T("accept() failed.POS:CServer::AcceptConnectRequest()"), MB_OK|MB_ICONERROR );
        return NULL;
    }
    return sDialog;
}
```

DataConnect()（建立数据连接）代码如下：

```
int CServer::DataConnect(SOCKET &s, DWORD dwIp, WORD wPort, int nMode)
{
    char errorMsg[128];
    //创建一个 socket
    s = socket( AF_INET,SOCK_STREAM,0 );
    if( s == INVALID_SOCKET )
    {
        sprintf( errorMsg,"socket() failed to get a socket with error: %d.
        POS:CServer::DataConnect()\n", WSAGetLastError() );
        AfxMessageBox( errorMsg, MB_OK|MB_ICONSTOP );
        return -1;
    }
    //配置地址信息
    struct sockaddr_in addrInfo;
    //配置地址簇
    addrInfo.sin_family = AF_INET;
    //配置端口和 IP
    if( nMode == MODE_PASV )
    {
        addrInfo.sin_port = htons( wPort );
        addrInfo.sin_addr.s_addr = dwIp;
```

```

    }
    else
    {
        addrInfo.sin_port = htons( DATA_FTP_PORT );
        //DATA_FTP_PORT==20, 调用函数获取本机地址
        addrInfo.sin_addr.s_addr = inet_addr( GetLocalAddress() );
    }
    //设置 closesocket (一般不会立即关闭而经历 TIME_WAIT 过程) 后想继续重用该 socket
    BOOL bReuseAddr = TRUE;
    if( setsockopt( s,SOL_SOCKET,SO_REUSEADDR,
        (char*)&bReuseAddr,sizeof(BOOL) ) == SOCKET_ERROR )
    {
        //关闭 socket
        closesocket(s);
        sprintf( errorMsg,"setsockopt() failed to setsockopt with error: %d.
            POS:CServer::DataConnect()\n", WSAGetLastError() );
        AfxMessageBox( errorMsg, MB_OK|MB_ICONSTOP );
        return -1;
    }
    //绑定 socket, 指定通信对象
    if( bind( s,(struct sockaddr*)&addrInfo,sizeof(addrInfo)) == SOCKET_ERROR )
    {
        //关闭 socket
        closesocket(s);
        sprintf( errorMsg,"bind() failed with error: %d.\n",WSAGetLastError() );
        AfxMessageBox( errorMsg, MB_OK|MB_ICONSTOP );
        return -1;
    }
    //主动与被动传输的不同操作
    if( nMode == MODE_PASV )
    {
        //开始监听
        if( listen( s,SOMAXCONN ) == SOCKET_ERROR )
        {
            //关闭 socket
            closesocket(s);
            sprintf(errorMsg,"listen() failed with error: %d.\n",WSAGetLastError() );
            AfxMessageBox( errorMsg, MB_OK|MB_ICONSTOP );
            return -1;
        }
    }
    else if( nMode == MODE_PORT )
    {
        struct sockaddr_in addr;
        addr.sin_family      = AF_INET;
        addr.sin_port        = htons( wPort );
    }

```

```

    addr.sin_addr.s_addr = dwIp;
    //连接到客户端
    if( connect( s, (const sockaddr*)&addr,sizeof(addr) ) == SOCKET_ERROR )
    {
        closesocket(s);
        sprintf(errorMsg,"connect() failed with error: %d\n",WSAGetLastError() );
        AfxMessageBox( errorMsg, MB_OK|MB_ICONSTOP );
        return -1;
    }
    //正常结束
    return 0;
}

```

这个函数在建立数据连接后还要设置连接状态：如果是主动模式，则服务器主动连接（connect()）到客户端；如果是被动模式，则服务器开始监听。

上段代码中用到了 GetLocalAddress()函数，用于获取本地 IP 地址，其实现如下：

```

char* CServer::GetLocalAddress()
{
    char hostname[128];
    char errorMsg[50];
    //初始化设置
    memset( hostname,0,sizeof(hostname) );
    //获取主机名称
    if( gethostname( hostname,sizeof(hostname) ) == SOCKET_ERROR )
    {
        sprintf( errorMsg,"gethostname() failed with error: %d.
                    POS:CServer::GetLocalAddress()",WSAGetLastError() );
        AfxMessageBox( errorMsg, MB_OK|MB_ICONSTOP );
        return NULL;
    }
    //gethostbyname()返回对应于给定主机名的包含主机名字和地址信息的 hostent 结构指针/失败返回一个
    //空指针
    LPHOSTENT lpHostEnt;
    lpHostEnt = gethostbyname( hostname );
    if ( lpHostEnt == NULL )
    {
        sprintf( errorMsg,"gethostbyname() failed with error: %d.
                    POS:CServer::GetLocalAddress()",WSAGetLastError() );
        AfxMessageBox( errorMsg, MB_OK|MB_ICONSTOP );
        return NULL;
    }
    //格式化表中的第一个地址
    struct in_addr* pinAddr;
    pinAddr = ((LPIN_ADDR)lpHostEnt->h_addr);
    //将网络地址转换成 “.” 分隔的字符串格式
    int length = (int)strlen( inet_ntoa(*pinAddr) );
}

```

```

        if ( (DWORD)length > sizeof(hostname) )
        {
            WSASetLastError(WSAEINVAL);
            return NULL;
        }
        return inet_ntoa(*pinAddr);
    }
}

```

2) 数据收发类

在 FTP 工作过程中，数据的收发本质上还是通过 Socket 进行的。本程序将数据接收和发送的操作分别封装在 DataSend()和 DataRecv()函数中，以便协议实现过程中随时调用。

DataSend()函数代码如下：

```

int CServer::DataSend(SOCKET s, char *buff, int buffSize)
{
    int nBytesLeft = buffSize;
    int index = 0;
    int sendBytes = 0;
    char errorMsg[128];
    while( nBytesLeft > 0 ) {
        sendBytes = send( s,&buff[index],nBytesLeft,0);
        if( sendBytes == SOCKET_ERROR )
        {
            closesocket( s );
            sprintf( errorMsg,"send() failed to send with error: %d.
                    POS:CServer::DataSend()\n",WSAGetLastError() );
            AfxMessageBox( errorMsg,MB_OK|MB_ICONERROR );
            return -1;
        }
        nBytesLeft -= sendBytes;
        index += sendBytes;
    }
    return index;
}

```

DataRecv()函数代码如下：

```

//数据接收，写入文件/只接收 4MB 以下的文件
int CServer::DataRecv(SOCKET s, const char *fileName)
{
    DWORD   index           = 0;
    DWORD   bytesWritten     = 0;
    DWORD   buffBytes       = 0;
    char    buff[DATA_BUFSIZE];
    int     totalRecvBytes   = 0;
    //获取文件绝对路径
    char    fileAbsolutePath[MAX_PATH];
    GetCurrentDirectory( MAX_PATH,fileAbsolutePath );
    strcat( fileAbsolutePath,"\\\" );
    strcat(fileAbsolutePath,fileName );
}

```



```

//创建文件句柄
HANDLE hFile = CreateFile( fileAbsolutePath,
                           GENERIC_WRITE,
                           FILE_SHARE_WRITE,
                           NULL,
                           OPEN_ALWAYS,
                           FILE_ATTRIBUTE_NORMAL,
                           NULL );

if( hFile == INVALID_HANDLE_VALUE )
{
    AfxMessageBox( _T("CreateFile() failed.POS:CServer::DataRecv()");
    return -1;
}

while( TRUE )
{
    //初始化设置, 准备接收数据
    int nBytesRecv = 0;
    index = 0;
    buffBytes = DATA_BUFSIZE;
    //开始接收数据
    while( buffBytes > 0 )
    {
        nBytesRecv = recv( s,&buff[index],buffBytes,0 );
        if( nBytesRecv == SOCKET_ERROR )
        {
            AfxMessageBox( _T("Failed in recv().POS:CServer::DataRecv()");
            return -1;
        }
        //接收为空或已经接收完
        if( nBytesRecv == 0 )
        {
            break;
        }
        index += nBytesRecv;
        buffBytes -= nBytesRecv;
    }
    //总共接收字节数增加
    totalRecvBytes += index;
    //要写入文件的字节数以接收到的字节数为准
    buffBytes = index;
    //索引请指向开始位置
    index = 0;
    //开始写入文件
    while( buffBytes > 0 )
    {
        //移动文件指针到文件末尾
        if( !SetEndOfFile(hFile) )

```

```

        {
            return 0;
        }
    if( !WriteFile( hFile,&buff[index],buffBytes,&bytesWritten,NULL ) )
    {
        CloseHandle( hFile );
        AfxMessageBox( _T("写入文件出错.") );
        return 0;
    }
    index += bytesWritten;
    buffBytes -= bytesWritten;
}
//判断是否接收完毕, 有可能文件大于定义的 buff 容量大小
if( nBytesRecv == 0 )
{
    break;
}
}
//关闭文件句柄
CloseHandle( hFile );
//返回总共接收字节数
return totalRecvBytes;
}

```

对于服务器向客户端返回应答的操作也封装成 SendACK()函数。

SendACK()函数代码如下:

```

//发送响应消息到客户端
int CServer::SendACK( LPSOCKET_INF socketInfo )
{
    //一个指向所发送数据字节数的变量
    static DWORD dwSendBytes = 0;
    char errorMsg[128];
    //设置 Socket 结构状态为发送状态
    socketInfo->SocketStatus = WSA_SEND;
    //初始化 wasOverLapped 结构
    memset( &(socketInfo->wsaOverLapped), 0, sizeof(WSAOVERLAPPED) );
    socketInfo->wsaOverLapped.hEvent = g_EventArray[g_index - WSA_WAIT_EVENT_0];
    //初始化要发送的内容
    socketInfo->wsaBuf.buf = socketInfo->buffSend + socketInfo->dwBytesSend;
    socketInfo->wsaBuf.len = (int)strlen( socketInfo->buffSend ) - socketInfo->dwBytesSend;
    if ( WSA_send(socketInfo->socket,&(socketInfo->wsaBuf),1,&dwSendBytes,0,
        &(socketInfo->wsaOverLapped),NULL) == SOCKET_ERROR )
    {
        if ( WSAGetLastError() != ERROR_IO_PENDING )
        {
            sprintf( errorMsg,"WSA_send() failed with error :%d.
                POS:CServer::SendACK()\n", WSAGetLastError() );
            AfxMessageBox( errorMsg );
        }
    }
}

```

```

        return -1;
    }
}
//成功返回
return 0;
}

```

由于服务器会频繁地响应客户端，因此在做了这样的封装以后，极大地减少了代码冗余，也提高了程序的可读性。

3) 格式处理类

要遵守网络协议所规定的标准，就必须将网络主体之间交互通信的报文数据按照协议要求处理成统一的格式，格式处理类函数就是专门用来完成这些比较底层的操作的。本例的格式处理函数如下。

DivideRequest()函数（分离请求信息），代码如下：

```

void CServer::DivideRequest(char *request, char *command, char *cmdtail)
{
    int i = 0;
    int len = (int)strlen( request );
    command[0] = '\0';
    for( i=0 ; i<len && request[i]!=' ' && request[i]!='\r' && request[i]!='\n'; i++ )
    {
        command[i] = request[i];
    }
    command[i] = '\0';
    //将命令全部转换为大写形式，便于处理
   strupr( command );
    //开始分离，有可能为空
    int index = 0;
    for( i++ ; i<len && request[i]!='\r' && request[i]!='\n'; i++ )
    {
        cmdtail[index] = request[i];
        index++;
    }
    cmdtail[index] = '\0';
}

```

CombindFileNameSize()函数（把文件名及文件大小信息整合到一起），代码如下：

```

//整合后的信息保存到 filenamesize 中并返回文件的大小
int CServer::CombindFileNameSize(const char *filename, char *filnamesize)
{
    //假定文件的大小不超过 4GB，只处理低位
    int fileSize = -1;
    FILE_INF fileInfo[1];
    //获取文件列表信息，并保存至 fileInfo 中，fileAmount 记录文件数目
    int fileAmount = GetFileList( fileInfo,1,filename );
    if( fileAmount != 1 )
    {
        return -1;
    }
}

```

```

    }
    //把文件名和大小信息整合到一起
    sprintf( filenamesize, "%s<%d bytes>",filename,fileInfo[0].nFileSizeLow );
    //获得文件大小
    fileSize = fileInfo[0].nFileSizeLow;
    //返回文件大小
    return fileSize;
}

```

上面程序中的 GetFileList()函数用于获取文件列表，代码如下：

```

//返回列表项数
int CServer::GetFileList(LPFILE_INF fileInfo, int arraySize, const char *searchPath)
{
    int index = 0;
    WIN32_FIND_DATA wfd;
    ZeroMemory( &wfd,sizeof(WIN32_FIND_DATA) );
    char fileName[MAX_PATH];
    ZeroMemory( fileName,MAX_PATH );
    //获取当前目录
    GetCurrentDirectory( MAX_PATH,fileName );
    //获取当前目录的下一级目录
    if( fileName[ strlen(fileName)-1 ] != '\\' )
    {
        strcat( fileName,"\" );
    }
    strcat( fileName, searchPath );
    //查找第一个文件(夹)
    HANDLE fileHandle = FindFirstFile( (LPCTSTR)fileName, &wfd );
    //获得文件属性
    if ( fileHandle != INVALID_HANDLE_VALUE )
    {
        lstrcpy( fileInfo[index].szFileName, wfd.cFileName );
        fileInfo[index].dwFileAttributes = wfd.dwFileAttributes;
        fileInfo[index].ftCreationTime = wfd.ftCreationTime;
        fileInfo[index].ftLastAccessTime = wfd.ftLastAccessTime;
        fileInfo[index].ftLastWriteTime = wfd.ftLastWriteTime;
        fileInfo[index].nFileSizeHigh = wfd.nFileSizeHigh;
        fileInfo[index].nFileSizeLow = wfd.nFileSizeLow;
        index++;
        //继续查找其他文件，非零表示成功，零表示失败
        while( FindNextFile( fileHandle,&wfd )!= 0 && index < arraySize )
        {
            lstrcpy( fileInfo[index].szFileName, wfd.cFileName );
            fileInfo[index].dwFileAttributes = wfd.dwFileAttributes;
            fileInfo[index].ftCreationTime = wfd.ftCreationTime;
            fileInfo[index].ftLastAccessTime = wfd.ftLastAccessTime;
            fileInfo[index].ftLastWriteTime = wfd.ftLastWriteTime;
            fileInfo[index].nFileSizeHigh = wfd.nFileSizeHigh;

```

```

        fileInfo[index++].nFileSizeLow = wfd.nFileSizeLow;
    }
    //查找完毕，关闭文件句柄
    FindClose( fileHandle );
}
return index;
}

```

FileListToString()函数（将文件列表信息转换成字符串），代码如下：

```
int CServer::FileListToString(char *buffer, UINT nBufferSize, BOOL isListCommand)
```

```

{
    //定义一个文件信息数组用于暂存当前目录下找到的文件信息
    FILE_INF fileInfo[MAX_FILE_NUM];
    //获取文件列表信息保存到数组 fileInfo 中并返回文件数目，保存至 fileAmmount
    int fileAmmount = GetFileList( fileInfo, MAX_FILE_NUM, " *.*" );
    //初始化
    sprintf( buffer,"%s", "" );
    //ListCommand，不仅需要文件名，还需要文件大小、创建时间等信息
    if( isListCommand )
    {
        SYSTEMTIME systemTime;    //用于暂存由文件时间转换过来的系统时间
        char tempTimeFormat[128]=""; //配置 fileString 中的文件时间格式
        char timeBlock[20];         //配置 fileString 中文件时间格式里的 AM、PM
        //开始把所有文件的文件名、创建时间信息集合到 buffer 中
        for( int i=0; i<fileAmmount; i++ )
        {
            //检测 buffer 是否有足够空间
            if( strlen(buffer)>nBufferSize-128 )
            {
                break;
            }
            //检测是否是文件
            if( strcmp(fileInfo[i].szFileName,".")==0 || strcmp(fileInfo[i].szFileName,"..")==0 )
            {
                continue;
            }
            //将文件时间转换为系统时间。成功：返回非 0，失败：返回 0
            if( FileTimeToSystemTime( &(fileInfo[i].ftLastWriteTime), &systemTime)==0 )
            {
                char errorMsg[128];
                sprintf( errorMsg,"POS:CServer::FileListToString()\nFailed to convert filetime to systemtime
with error: %d.",GetLastError());
                AfxMessageBox( errorMsg,MB_OK|MB_ICONERROR );
            }
            //配置 fileString 中文件时间格式里的 AM、PM
            if( systemTime.wHour <= 12 )
            {
                strcpy( timeBlock,"AM" );
            }
        }
    }
}

```

```

    }
    else
    {
        systemTime.wHour -= 12;
        strcpy( timeBlock,"PM" );
    }
    //配置 fileString 中文件时间格式里只用两位数来表示
    if( systemTime.wYear >= 2000 )
    {
        systemTime.wYear -= 2000;
    }
    else
    {
        systemTime.wYear -= 1900;
    }
    //配置时间格式
    ZeroMemory( tempTimeFormat,sizeof(tempTimeFormat) );
    sprintf( tempTimeFormat,"%02u-%02u-%02u   %02u:%02u%s   ",
            systemTime.wMonth,systemTime.wDay,systemTime.wYear,
            systemTime.wHour,systemTime.wMonth,timeBlock );
    //把当前文件时间保存至 buffer 数组中
    strcat( buffer,tempTimeFormat );
    //检查找到的结果是否为目录
    if( fileInfo[i].dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY )
    {
        //是目录
        strcat( buffer,"<DIR>" );
        strcat( buffer,"                " );
    }
    else //是文件
    {
        strcat( buffer,"                " );
        //文件大小
        sprintf( tempTimeFormat,"%09d Bytes   ",fileInfo[i].nFileSizeLow );
    }
    //文件名
    strcat( buffer,fileInfo[i].szFileName );
    strcat( buffer,"\r\n");
}
}
else//-----NLIST command 只需要文件名
{
    //把当前目录下的文件的名称集合到 buffer 中
    for( int i=0; i<fileAmmount; i++ )
    {
        //检测是否有足够空间
        if( strlen(buffer) + strlen( fileInfo[i].szFileName ) + 2 < nBufferSize )

```

```

        {
            strcat( buffer, fileInfo[i].szFileName );
            strcat( buffer, "\r\n");
        }
        else
        {
            break;
        }
    }
}
return (int)strlen( buffer );
}

```

ConvertCommaAddrToDotAddr()函数（将逗号分隔的 IP 序列转化为点分十进制形式），代码如下：

```

int CServer::ConvertCommaAddrToDotAddr( char* commaAddr,
                                         LPDWORD pdwIpAddr, LPWORD pwPort )
{
    int index    = 0;
    int i        = 0;
    int commaAmout    = 0;           //计数逗号个数
    char dotAddr[MAX_ADDR_LEN];     //用于提取 4 个逗号相隔的 IP 地址 “a.b.c.d”
    char szPort[MAX_ADDR_LEN];      //用于提取 IP 地址后面的端口号
    //初始化
    memset( dotAddr,0,sizeof(dotAddr) );
    memset( szPort, 0, sizeof(szPort) );
    *pdwIpAddr    = 0;
    *pwPort        = 0;
    //找出所有的 “,” 并改为 “.”
    while( commaAddr[index] )
    {
        if( commaAddr[index] == ',' )
        {
            commaAmout++;
            commaAddr[index] = '.';
        }
        if( commaAmout < 4 )
        {
            dotAddr[index] = commaAddr[index];
        }
        else
        {
            szPort[i++] = commaAddr[index];
        }
        index++;
    }
    //检测合法性
    if( commaAmout != 5 )

```

```

    {
        return -1;
    }
    //将得到的 4 个 “.” 分隔的 IP 转换为相应的无符号长整型表示
    (*pdwIpAddr) = inet_addr( dotAddr );
    if( (*pdwIpAddr) == INADDR_NONE )
    {
        return -1;
    }
    char *temp = strtok( szPort+1, "." );          //前部分
    if( temp == NULL )
    {
        return -1;
    }
    (*pwPort) = (WORD)( atoi(temp) * 256 );
    temp = strtok(NULL, ".");                      //后部分
    if( temp == NULL )
    {
        return -1;
    }
    (*pwPort) += (WORD)atoi( temp );
    return 0;
}

```

与以上处理过程相反的是，将点分十进制记法的 IP 转换成逗号隔开的 IP 序列。ConvertDotAddrToCommaAddr()函数可将一个 32 位 Internet 主机地址和一个 16 位 TCP 端口地址转换为一个由 6 个被逗号隔开的数字组成的 IP 地址序列。

ConvertDotAddrToCommaAddr()函数代码如下：

```

int CServer::ConvertDotAddrToCommaAddr(char *dotAddr, WORD wPort, char* commaAddr )
{
    //处理端口
    char szPort[10];
    sprintf( szPort, "%d,%d", wPort/256, wPort%256 );
    //处理标准 Internet IP
    sprintf( commaAddr, "%s", dotAddr );
    //将 commaAddr 中的 “.” 转换为 “,”
    int index = 0;
    while( commaAddr[index] )
    {
        if( commaAddr[index] == '.' )
        {
            commaAddr[index] = ',';
        }
        index++;
    }
    //将 IP、端口拼接起来
    sprintf( commaAddr, "%s%s", commaAddr, szPort );
}

```



```
    return 0;  
}
```

在网络程序中，常常涉及网络资源 URL 与本地文件路径互换的问题。一般来说，表示网络 URL 的路径之间以“/”分隔，而本地的文件路径之间却是以“\”分隔的，两者之间的转换函数如下。

NetToHost()函数（将../..格式转换为\\.\.格式），代码如下：

```
void CServer::NetToHost(char *path)  
{  
    int index = 0;  
    while( path[index] )  
    {  
        if( path[index] == '/' )  
        {  
            path[index] = '\\';  
        }  
        index++;  
    }  
}
```

HostToNet()函数（将\\.\.格式转换为../..格式），代码如下：

```
void CServer::HostToNet(char *path)  
{  
    int index = 0;  
    while( path[index] )  
    {  
        if( path[index] == '\\' )  
        {  
            path[index] = '/';  
        }  
        index++;  
    }  
}
```

4) 目录操作类

FTP 的一个重要功能就是为用户提供对远程文件资源的访问，用户操作服务器上的文件如同操作自己计算机上的文件一样，可以灵活地切换目录。切换目录的功能在底层是通过一些功能函数实现的。

GetAbsoluteDirectory()函数（获得绝对路径），代码如下：

```
void CServer::GetAbsoluteDirectory(char* dir,char* currentDir,char* userCurrDir )  
{  
    char szTemp[MAX_PATH];  
    int len = 0;  
    //获得用户当前目录  
    strcpy( userCurrDir,currentDir );  
    //回到上一目录  
    if( strcmp( dir,".." )==0 )  
    {
```

```
int flag = 0;
len = (int)strlen( userCurrDir );
for( int i=0 ; i<len ; i++ )
{
    if( userCurrDir[i] == '\\' )
    {
        flag = i;
    }
}
//从最后一个 “\” 处截断
if( flag>2 )
{
    userCurrDir[flag] = '\0';
}
else if( flag==2 )
{
    userCurrDir[flag+1] = '\0';
}
return;
}
//到当前目录的下一个目录
len = (int)strlen( userCurrDir );
if( userCurrDir[len-1]=='\' )
{
    if( dir[0]=='/' || dir[0]=='\' )
    {
        strcpy( szTemp,dir+1 );
    }
    else
    {
        strcpy( szTemp,dir );
    }
}
else
{
    if( dir[0] != '/' && dir[0] != '\' )
    {
        strcpy( szTemp,"\" );
        strcat( szTemp, dir );
    }
    else
    {
        strcpy( szTemp,dir );
    }
}
//默认此文件夹在当前目录下
strcat( userCurrDir,szTemp );
```

```
//将../..格式转换为\\..\\格式
NetToHost( userCurrDir );
}
```

上面程序中用到了格式处理的 NetToHost()函数。

IsPathExist()函数（检测路径是否合法），代码如下：

```
BOOL CServer::IsPathExist(char *path)
{
    if( GetFileAttributesA(path) != INVALID_FILE_ATTRIBUTES ||
        GetLastError() != ERROR_FILE_NOT_FOUND )
    {
        return TRUE;
    }
    return FALSE;
}
```

GetRalativeDirectory()函数（获得相对路径），代码如下：

```
void CServer::GetRalativeDirectory(char *currDir,char*rootDir,char* ralaDir)
{
    int nStrLen = (int)strlen( rootDir );
    //比较字符串 currDir 和 rootDir 的前 nStrLen 个字符但不区分大小写
    if( strcmp( currDir,rootDir, nStrLen ) == 0 )
    {
        strcpy( ralaDir,currDir + nStrLen );
    }
    //判断是否是根目录
    if( ralaDir != NULL && ralaDir[0] == '\0' )
    {
        strcpy( ralaDir,"/" );
    }
}
```

5) 文件操作类

对 FTP 上资源的操作最终都要依赖于服务器系统自身的文件操作来完成，下面是文件操作函数。

ReadFileToBuffer()函数（将指定文件写入缓存），代码如下：

```
int CServer::ReadFileToBuffer(const char *szFileName, char *buff, int fileSize)
{
    int index = 0;
    int bytesLeft = fileSize;
    int bytesRead = 0;
    //获取文件的完整路径
    char fileAbsolutePath[MAX_PATH];
    GetCurrentDirectory( MAX_PATH,fileAbsolutePath );
    if( fileAbsolutePath[ strlen(fileAbsolutePath)-1 ] != '\\' )
    {
        strcat( fileAbsolutePath,"\\" );
    }
    strcat(fileAbsolutePath,szFileName );
```

```

HANDLE hFile = CreateFile( fileAbsolutePath,
                           GENERIC_READ,
                           FILE_SHARE_READ,
                           NULL,
                           OPEN_EXISTING,
                           FILE_ATTRIBUTE_NORMAL,
                           NULL );

if( hFile == INVALID_HANDLE_VALUE )
{
    AfxMessageBox( _T("Filed in CreateFile(),POS:CServer::ReadFileToBuffer()") );
}
else
{
    while( bytesLeft > 0 )
    {
        if( !ReadFile( hFile,&buff[index],bytesLeft,(LPDWORD)&bytesRead,NULL ) )
        {
            //关闭句柄
            CloseHandle( hFile );
            AfxMessageBox( _T("Filed in ReadFile(),POS:CServer::ReadFileToBuffer()") );
            return 0;
        }
        index += bytesRead;
        bytesLeft -= bytesRead;
    }
    //关闭句柄
    CloseHandle( hFile );
}
//返回总共读取的字节数
return index;
}

```

TryDeleteFile()函数（删除文件），代码如下：

```

int CServer::TryDeleteFile(char *deletedPath)
{
    //定义一个 CFileFind 类对象用于查找
    CFileFind fileFinder;
    if( !fileFinder.FindFile( deletedPath ) ) //-----路径不合法
    {
        return CANNOT_FIND;
    }
    else if( DeleteFile( deletedPath ) ) //-----文件能删除
    {
        return DIR_CHANGED;
    }
    else//-----文件暂时不能删除
    {
        return ACCESS_DENY;
    }
}

```

```
}  
}
```

至此，整个 FTP 服务器程序的代码全部展示完成。虽然服务器的代码量很大，但是经过一番梳理，要理清其结构并不太难。

6.4 自制 FTP 客户端与服务器对接

在看完了这个 FTP 服务器程序的源代码后，本节来运行这个程序。我们将使用前面 6.2 节制作的 FTP 上传下载器作为客户端，对接访问该服务器。

6.4.1 FTP 上传下载器的改造

本例中 FTP 服务器同时支持注册用户和匿名两种访问方式，为了分别对这两种方式进行测试，需要对客户端程序做一点小的改动。

打开已经做好的 FTP 客户端（SelfFtpUpDownloader）项目工程，修改其代码。

改写“连接”按钮的事件过程，代码如下：

```
void CSelfFtpUpDownloaderDlg::OnConnect()  
{  
    if(!(this->ConnectFtp()))  
    {  
        return;  
    }  
    this->UpdateDir();  
    ServerIP.EnableWindow(false);  
    m_port.EnableWindow(false);  
    m_connect.EnableWindow(false);  
    m_disconnect.EnableWindow(true);  
    m_enterdir.EnableWindow(true);  
    m_upload.EnableWindow(true);  
    m_download.EnableWindow(true);  
    m_delete.EnableWindow(true);  
    m_noname.EnableWindow(false);  
    m_exit.EnableWindow(false);  
    m_usr.EnableWindow(false);  
    m_pwd.EnableWindow(false);  
}
```

其中，加黑的字体为修改或添加的代码。在这里，添加了对连接是否成功的验证，以及用户名和密码文本框的可用性控制。

ConnectFtp()函数代码如下：

```
BOOL CSelfFtpUpDownloaderDlg::ConnectFtp()  
{  
    BYTE nFild[4];  
    UpdateData();  
    if(ServerIP.IsBlank())
```

```

    {
        AfxMessageBox("请指定 IP 地址!");
        return false;
    }
    if(strport.IsEmpty())
    {
        AfxMessageBox("请指定连接端口!");
        return false;
    }
    if(strusr.IsEmpty())
    {
        AfxMessageBox("请填写用户名!");
        return false;
    }
    if(strpwd.IsEmpty() && strusr != "anonymous")
    {
        AfxMessageBox("请输入密码!");
        return false;
    }
    ServerIP.GetAddress(nFild[0],nFild[1],nFild[2],nFild[3]);
    CString sip;
    sip.Format("%d.%d.%d.%d",nFild[0],nFild[1],nFild[2],nFild[3]);
    pInternetSession = new CInternetSession("MR",INTERNET_OPEN_TYPE_PRECONFIG);
    try {
        pFtpConnection = pInternetSession->GetFtpConnection(sip,
                                                                strusr,strpwd,atoi(strport));

        bconnect = true;
    } catch(CInternetException* pEx)
    {
        TCHAR szErr[1024];
        pEx->GetErrorMessage(szErr, 1024);
        AfxMessageBox(szErr);
        pEx->Delete();
    }
    return true;
}

```

这段代码改动比较大，主要是增加了对用户名和密码的验证。

在“断开”按钮的事件 OnDisconnect()函数中添加以下两句代码：

```

m_usr.EnableWindow(true);
m_pwd.EnableWindow(true);

```

在 OnNoname()函数中注释掉下面这句代码：

```
//m_connect.EnableWindow(false);
```

在 BOOL CSelfFtpUpDownloaderDlg::OnInitDialog()的初始化代码中注释掉下面这句代码：

```
//m_connect.EnableWindow(false);
```

经过上述几处简单的修改，这个上传下载器就具备了对注册用户和匿名用户的双重支持功能，接下来使用它来测试 FTP 服务器程序。

6.4.2 自制客户端访问服务器

1. 匿名访问服务器

根据服务器默认的账号，匿名（anonymous）用户的根目录为 C:\Documents and Settings\All Users\Documents（见表 6.6），这个目录所在的位置是“我的电脑”→“共享文档”。进入该目录，如图 6.21 所示，在其下存放一张图片。

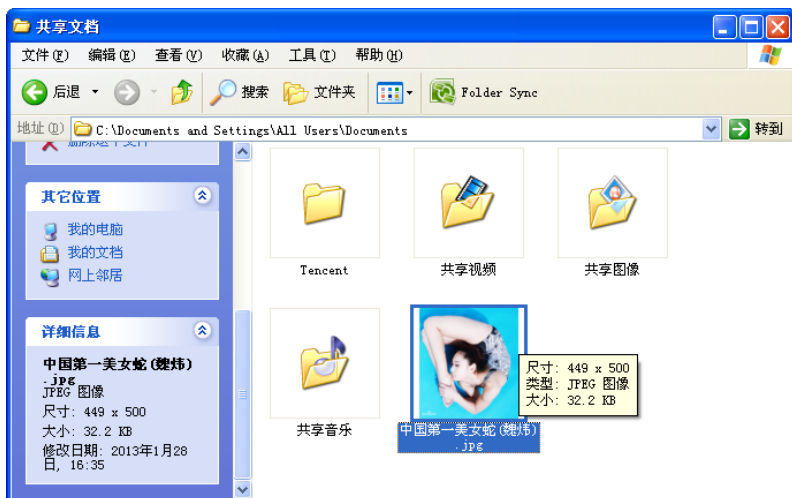


图 6.21 预置图片资源

启动 FTP 服务器，如图 6.22 所示。

接着启动修改后的 FTP 上传下载器，输入服务器的 IP 和端口后，勾选“匿名”复选框，然后单击“连接”按钮，匿名登录服务器。登录后进入匿名用户默认的根目录（我的电脑\共享文档），如图 6.23 所示。



图 6.22 启动 FTP 服务器

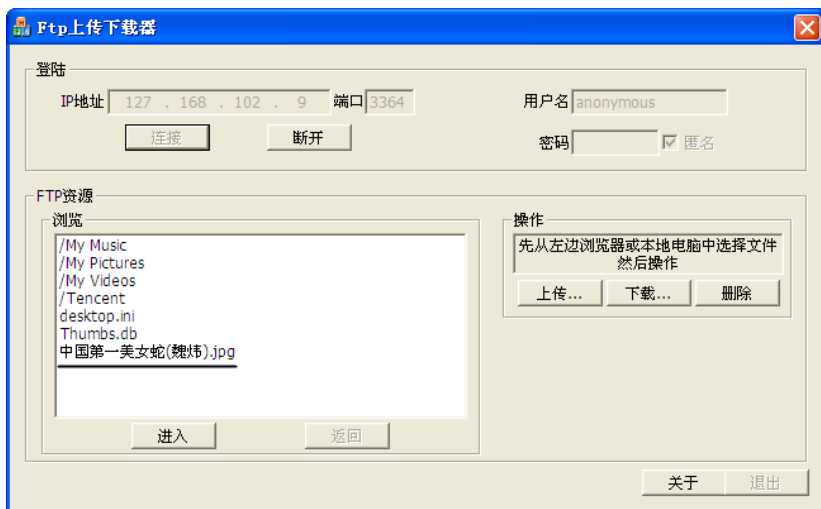


图 6.23 匿名登录

可以看到刚刚放在“共享文档”目录下的图片“中国第一美女蛇(魏炜).jpg”，选中其在客户端界面上单击“下载...”按钮，可以下载这幅图片，如图 6.24 所示。

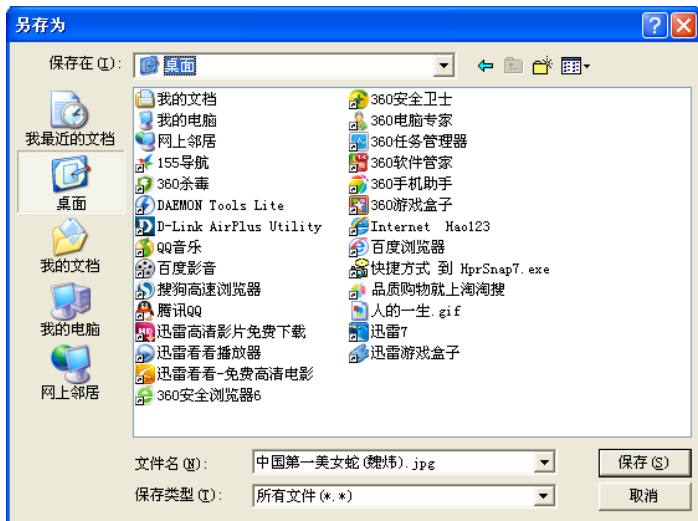


图 6.24 下载图片

2. 注册用户访问服务器

断开与服务器的连接，重新以一个注册用户的身分登录。这里用默认的普通用户（用户名和密码都是“usr”）登录服务器，取消勾选“匿名”复选框，输入用户名和密码。登录后，系统自动进入 usr 用户默认的 D 盘根目录下，如图 6.25 所示，将刚才下载到桌面的图片上传上去。

完成后弹出“上传成功!”消息框，如图 6.26 所示，可以看到图片已经成功上传到 usr 用户的目录下了。

有兴趣的读者还可以逐一测试软件的其他功能：删除文件、切换目录、添加用户、删除用户、编辑用户信息……这些操作都很容易完成，本书不再赘述。

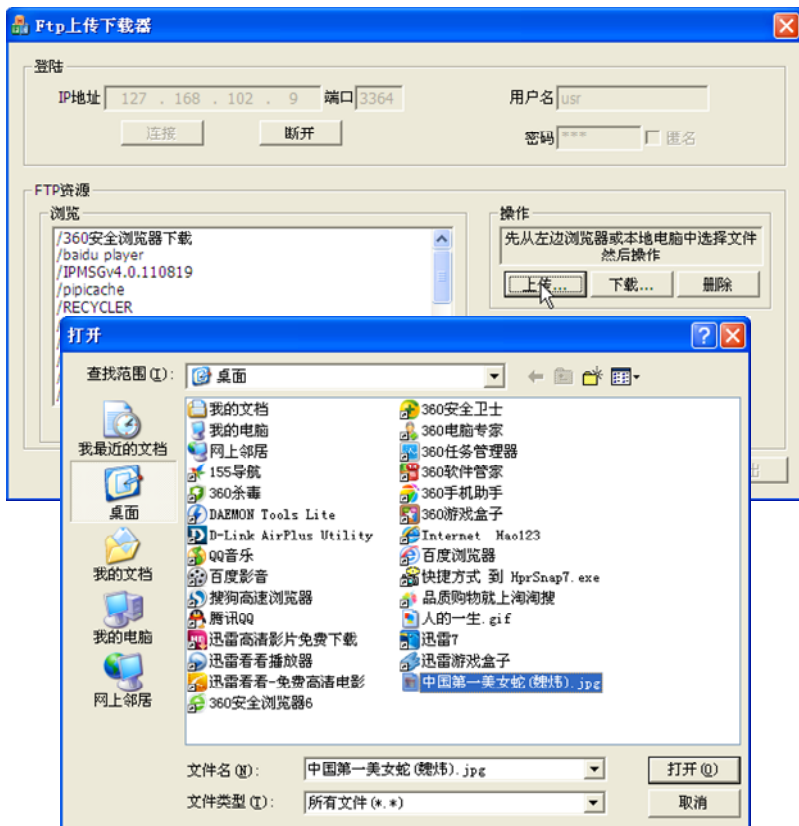


图 6.25 上传图片

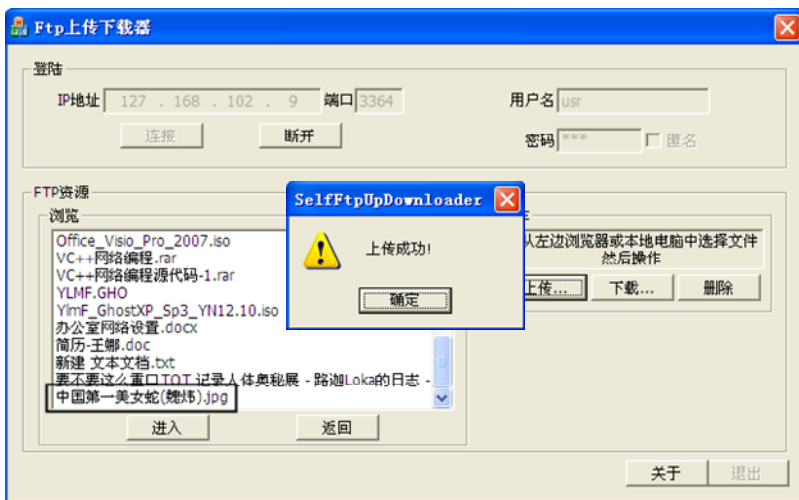


图 6.26 上传成功

由上面的实验可见，自制的 FTP 客户端 SelfFtpUpDownloader 和服务端 ftpSrver 完全可以无缝对接，它们共同组成了一个完整的软件套件。在开发上传下载器时，我们并没有直接接触网络协议，是用微软 MFC 中提供的现成 WinInet 类，而服务器却是按照 FTP 协议编写代码的。两者之所以能集成在一起成为一个套件，根本的原因是，微软在开发 WinInet 时遵守的也是 FTP 协议规范。正是由于双方都遵守了同一个网络协议标准，两个程序之间才能够顺利实现对接。

电子邮件应用编程

电子邮件是 Internet 上历史最悠久、最古老的应用，原始的计算机网络几乎就是使用邮件作为唯一的通信方式。现在很多流行的网络应用往往也附带邮箱功能，如 QQ 邮箱、MSN 邮箱等。本章要实现的是电子邮件应用的客户端程序，它能与各大门户网站（新浪、网易）的邮箱相配合，实现邮件收发功能。

7.1 邮件系统原理

7.1.1 概述

电子邮件的基本原理，是在网络上设立“邮箱”，它实际上是一个计算机系统。系统的硬件是网站拥有的邮件服务器，是一个高性能、大容量的计算机或集群。邮件系统的工作过程遵循客户端—服务器（C/S）模式。每封邮件的发送都要涉及发送方与接收方，发送方作为客户端，而接收方就是邮件服务器，它存储众多用户的邮箱。

1. 传统邮件系统

Web 发明之前，还没有浏览器，上网是无法看到网页的，只能通过运行各种网络应用所对应的客户端程序来使用网络。在电子邮件应用中，发送方是通过邮件客户程序，将编辑好的信件向邮件服务器（ISP 主机）发送的，如图 7.1 所示。这种邮件客户程序又称为用户代理 UA（User Agent），它是传统的用户与邮件系统的接口。

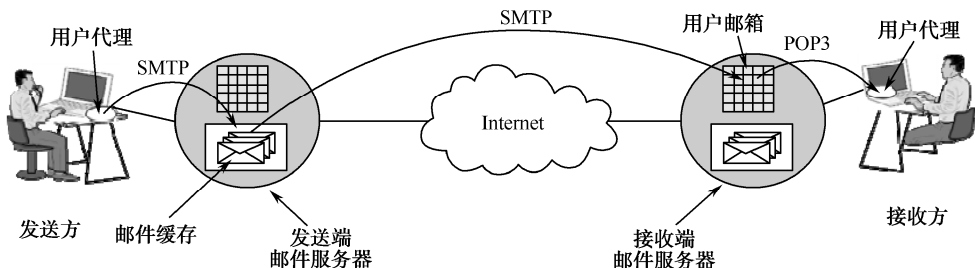


图 7.1 传统邮件收发过程

发送端邮件服务器识别接收者的地址，并向管理该地址的接收端邮件服务器发送消息。接收端服务器将消息存放在接收者的邮箱内，并告知接收者有新邮件到来。接收者通过邮件客户程序

连接到服务器后, 就会看到服务器的通知, 进而打开自己的邮箱来查收信件。因此, 当发送一封电子邮件给另一个用户时, 邮件首先从用户计算机发送到 ISP 主机, 再到 Internet, 然后到收件人一端的 ISP 主机, 最后才到收件人的个人计算机。这里的 ISP 主机起着“邮局”的作用, 管理着众多用户的邮箱。每个用户的邮箱都要占用 ISP 主机一定容量的硬盘空间。

2. 邮件系统的协议

邮件系统常用的协议有 SMTP (简单邮件传输协议)、POP3 (邮局协议)、IMAP (Internet 邮件访问协议), 这几种协议都是由 TCP/IP 协议族定义的。

- SMTP (Simple Mail Transfer Protocol): SMTP 主要负责底层的邮件系统如何将邮件从一台主机传至 Internet 上的另外一台主机上。
- POP (Post Office Protocol): 目前的版本为 POP3, 它是把邮件从邮箱中传输到本地计算机的协议。POP3 协议的一个特点是, 只要用户从 POP 服务器读取了邮件, POP 服务器就将该邮件删除 (稍后会向大家演示这个过程)。
- IMAP (Internet Message Access Protocol): 目前的版本为 IMAP4, 是 POP3 的一种替代协议, 提供了邮件检索和邮件处理的新功能, 这样用户可以完全不必下载邮件正文就可以看到邮件的标题摘要, 从邮件客户端软件就可以对服务器上的邮件和文件夹目录等进行操作。

传统方式下, 发件人的用户代理向发送端邮件服务器发邮件, 以及发送端邮件服务器向接收端邮件服务器传输邮件, 使用的都是 SMTP 协议。而 POP 或 IMAP 则是用户代理从己方邮件服务器上读取邮件所使用的协议。传统邮件系统协议的工作过程在图 7.1 中已经标出。

3. 基于 Web 的邮件系统

万维网的发明使得用户直接通过浏览器访问自己的邮箱成为可能。这时, 邮件系统中的用户代理就是普通的万维网浏览器 (如微软 IE、360 安全等)。假定用户 A 向网易申请了一个邮箱 aaa@163.com。当 A 需要发送或接收电子邮件时, 首先登录网易的邮件服务器 (mail.163.com), 在输入自己的用户名和密码后, 就可以根据屏幕上的提示, 撰写、发送或读取自己的信件了。但是请注意, 电子邮件从 A 的浏览器发送到网易邮件服务器时, 不是使用 SMTP 协议, 而是使用 HTTP 协议。假定 A 发送的邮件的收件人是 B, B 使用新浪网站的邮箱, 地址是 bbb@sina.com。于是 A 发送的邮件先从网易的邮件服务器 (这时使用的是 SMTP, 而不是 HTTP) 发送到新浪的邮件服务器 (mail.sina.com.cn), 但 B 用浏览器从新浪邮件服务器读取 A 发来的信件时, 使用的是 HTTP 协议, 而不是 POP3 或 IMAP 协议。以上过程如图 7.2 所示。



图 7.2 基于 Web 的邮件系统

这种基于 Web 的邮件应用方式已经突破了传统的 C/S 模式, 形成目前广为流行的浏览器—服务器 (B/S) 模式, 它是 C/S 的升级和发展。

7.1.2 邮件客户端配置

Outlook Express 是 Microsoft Windows 操作系统自带的一款电子邮件客户端, 它建立在开放

的 Internet 标准基础之上, 适用于任何 Internet 标准系统, 提供对目前最重要的电子邮件、新闻和目录标准的完全支持, 可确保用户能够充分利用新技术, 无缝地发送和接收电子邮件。

1. 申请免费邮箱

在使用 Outlook Express 之前, 需要到网上申请一个电子邮箱。在此申请两个邮箱 (一个网易的和一个新浪的), 分别模拟邮件收发双方。

先在新浪网注册一个免费邮箱 `zhouhejun2010@sina.cn` (密码为 198309252010), 模拟邮件接收者的信箱。

再在网易注册一个邮箱 `xuhehe2010@163.com` (密码为 `xuhe20061216`), 模拟邮件发送者的信箱。

邮箱申请好后, 可以互相发个邮件试试。当然, 这是基于 Web 的邮件系统, 通过浏览器与邮件服务器交互使用的是 HTTP。

下面要用邮件客户端 Outlook Express 来访问刚申请的邮箱, 使用之前需要进行一系列的配置。

2. 启动 Outlook Express

单击“开始”→“所有程序”→“Outlook Express”启动 Outlook Express, 出现如图 7.3 所示的主界面。



图 7.3 Outlook Express 主界面

可以看到, 它里面也有“收件箱”、“发件箱”、“已发送邮件”、“已删除邮件”、“草稿”这些电子邮箱基本的功能, 就像从网页访问的邮箱一样。

3. 创建邮件账户

单击开始页面上的“设置邮件账户”链接, 进入“Internet 连接向导”, 如图 7.4 所示。

因为之前申请的网易邮箱为 `xuhehe2010@163.com`, 其中“@”符号前的字符串“`xuhehe2010`”为用户名, 所以将它填写在“显示名”文本框中, 单击“下一步”按钮, 在图 7.5 所示的界面填写电子邮件地址。

填写的地址为“`xuhehe2010@163.com`”, 同之前申请的网易邮箱地址一模一样, 单击“下一步”按钮继续。

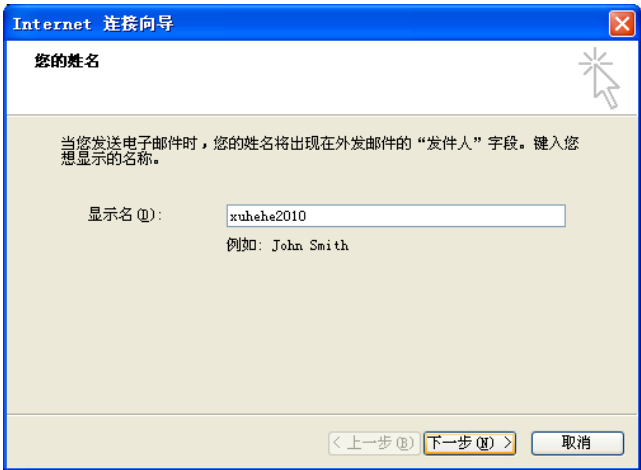


图 7.4 Internet 连接向导

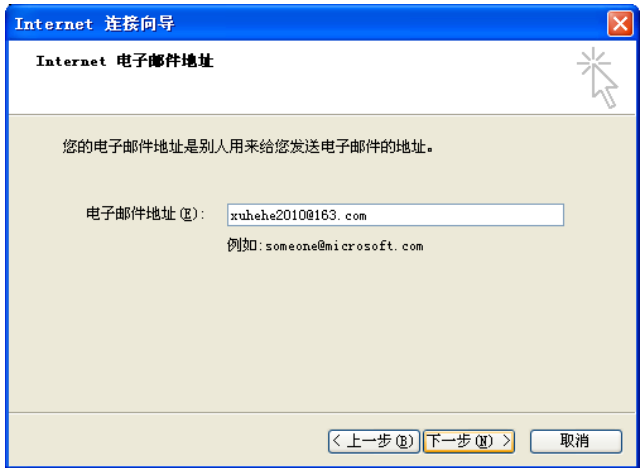


图 7.5 填写电子邮件地址

进入配置邮件服务器界面，指定接收邮件的服务器为 pop.163.com，发送邮件服务器为 smtp.163.com，单击“下一步”按钮，如图 7.6 所示。



图 7.6 配置邮件服务器

在接下来如图 7.7 所示的“Internet Mail 登录”界面输入账户名和密码。这里的账户名就是网易邮箱的用户名“xuhehe2010”，密码是前面申请邮箱时设定的密码 xuhe20061216，单击“下一步”按钮。

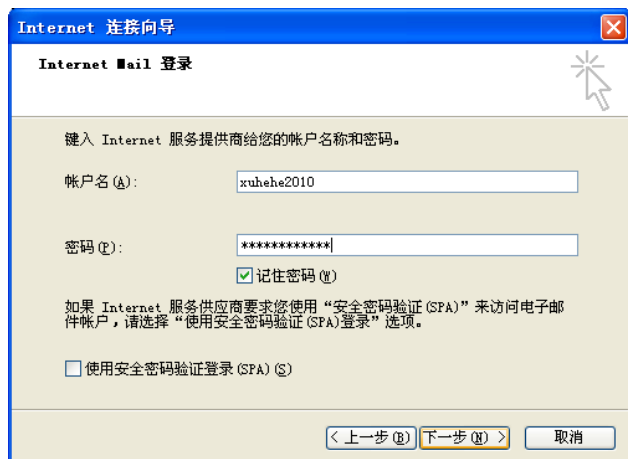


图 7.7 输入账户名和密码

最后，单击“完成”按钮结束邮件账户的创建，如图 7.8 所示。

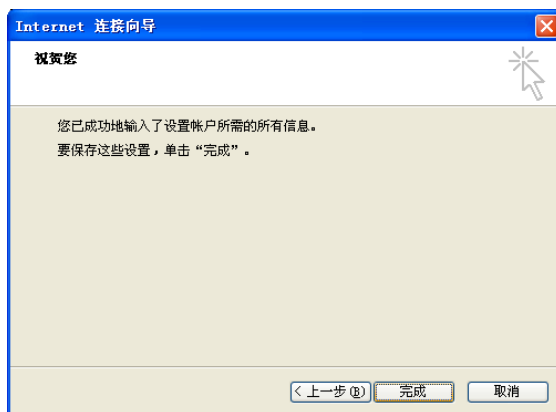


图 7.8 完成创建

4. 邮件账户设置

选择菜单命令“工具”→“账户”，打开“Internet 账户”对话框，如图 7.9 所示。

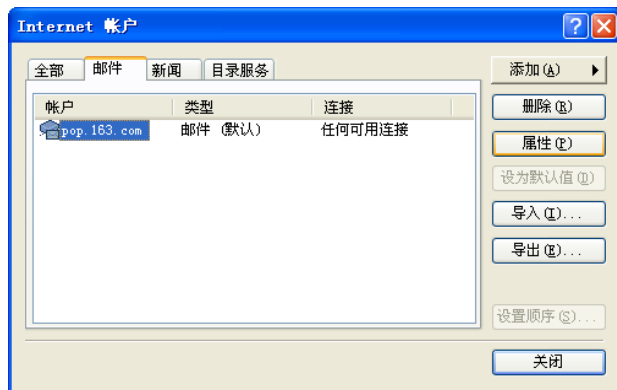


图 7.9 “Internet 账户”对话框

在对话框的“邮件”选项卡中可以看到前面指定的接收邮件服务器 pop.163.com，单击“属性”按钮设置其属性，如图 7.10 所示。请读者核对属性设置页信息，确保自己的设置与图中的完全一样。



图 7.10 设置邮件服务器属性

在“服务器”选项卡中选中“我的服务器要求身份验证”复选框，单击右边的“设置”按钮，弹出如图 7.11 所示的“发送邮件服务器”对话框，选中“登录方式”单选按钮，输入账户名和密码。

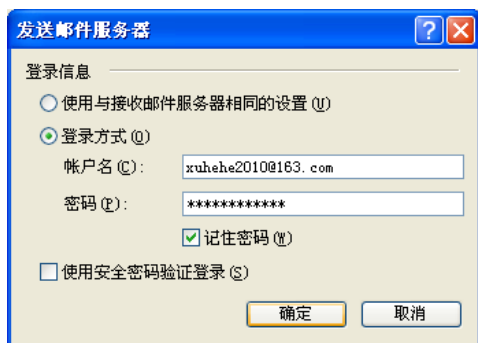


图 7.11 登录账户名和密码

这里，账户名填写完整的电子邮件地址 xuhehe2010@163.com，密码还是先前的 xuhe20061216，单击“确定”按钮完成设置。

5. 试发邮件

至此，已经配置好了 Outlook Express 客户端并将它与之前申请的网易邮箱捆绑起来，下面来发封邮件试试。

选择主菜单“文件”→“新建”→“邮件”（或直接单击工具栏上的 ），打开信件编辑窗口。如图 7.12 所示，在其中写一封信，在收件人栏填写之前申请的新浪邮箱地址 zhouhejun2010@sina.cn，主题为“电子邮件发送实验（微软 Outlook 客户端发送）”。

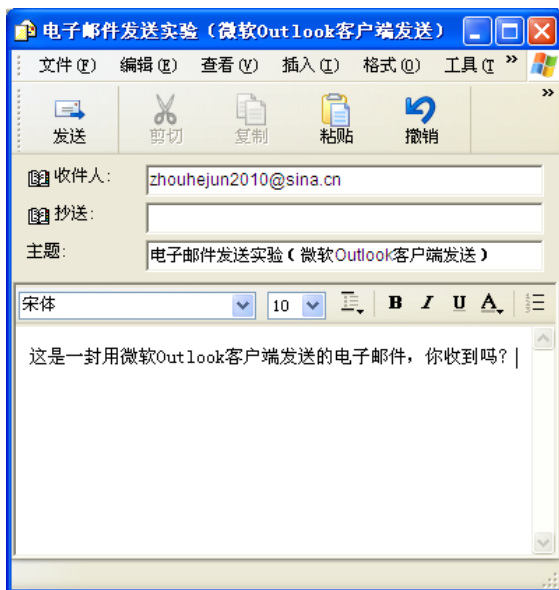


图 7.12 写信

编辑完信件内容后单击工具栏上的“发送”按钮，将邮件发送出去。登录新浪邮箱后，可以看到刚才发送的邮件，如图 7.13 所示，这说明 Outlook Express 客户端的配置是正确的。

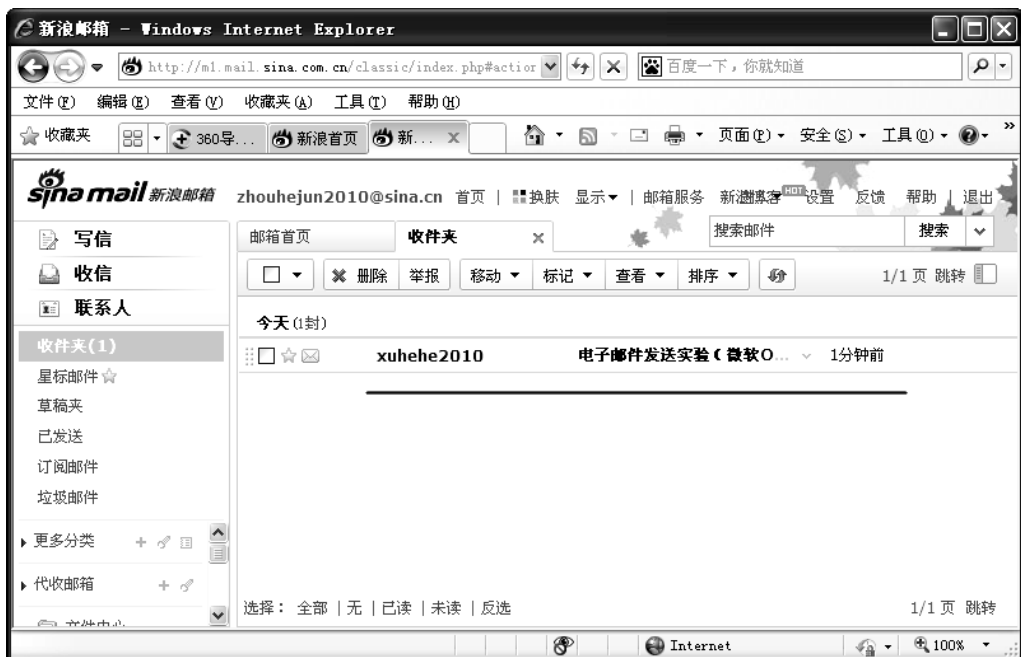


图 7.13 成功收到邮件

7.1.3 邮件收发环境

1. 邮件服务器

在配置 Outlook Express 时，其中有一步需要配置两个服务器，如图 7.14 所示。



图 7.14 两个邮件服务器

邮件系统至少要实现邮件的发送和接收两大最基本的功能,这两大功能所使用的协议是不一样的:发送邮件用的是 SMTP,而接收邮件用的则是 POP3、IMAP 或 HTTP。这里把主要代理用户发送邮件的服务器程序称为“SMTP 服务器”;而把负责将服务器上的邮件转交给客户端的程序称为“邮件接收服务器”,本例中是 POP3 服务器。

例如,在配置 Outlook Express 时,选择的邮件接收服务器(程序)实现的是 POP3 协议,名为“pop.163.com”;而 SMTP 服务器(程序)的名字为“smtp.163.com”,见图 7.14 中的配置。

为了让读者对邮件收发的整个过程有一个清晰而全面的认识,特将这个过程用图 7.15 表示出来。

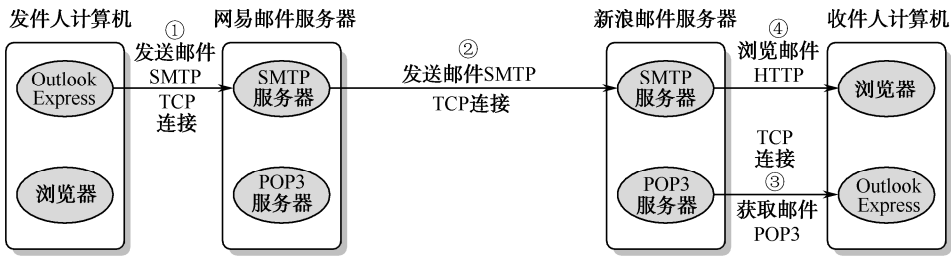


图 7.15 邮件收发的整个过程

图 7.15 中的 SMTP 服务器是邮件服务商计算机上的软件程序,它负责接收用户提交的邮件(①),代为发送到对方邮件服务器(②),而用户客户端获取邮件(③)则是通过服务商计算机上的另一组程序,它可以实现 POP3 也可以采用 IMAP (本例用的是 POP3),即前面所指的“邮件接收服务器”。当然用户也可不必经过客户端而是直接登录网站邮箱发邮件,这时就是通过浏览器直接与“SMTP 服务器”打交道(④),用的自然是 HTTP;同理,接收邮件也可以通过浏览器实现,这是当前普遍采用的方式。

2. 邮件收发过程演示

对于图 7.15 中的步骤①,如果它确实发生过,用户用 Outlook Express 发送邮件时, Outlook Express 先把邮件交给 SMTP 服务器,那么服务器上必然留下邮件副本。下面来登录网易邮箱看一下,如图 7.16 所示。

网易(发送方)邮箱“已发送”中确实存有 7.1.2 节所发邮件的副本。这就证明了该邮件是通过 Outlook Express 客户软件先发给网易邮件服务器(①),再由网易邮件服务器转发(②)给新浪邮件服务器的。

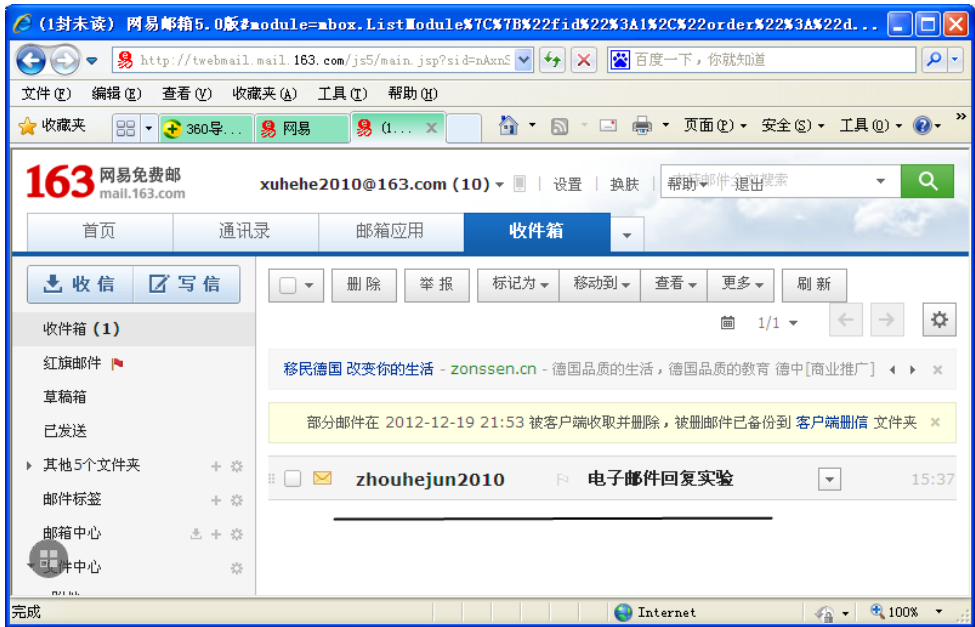


图 7.18 发来的邮件先存放在网易邮箱里

接着，启动 Outlook Express 客户端，可以看到新浪用户回复的信件，如图 7.19 所示。

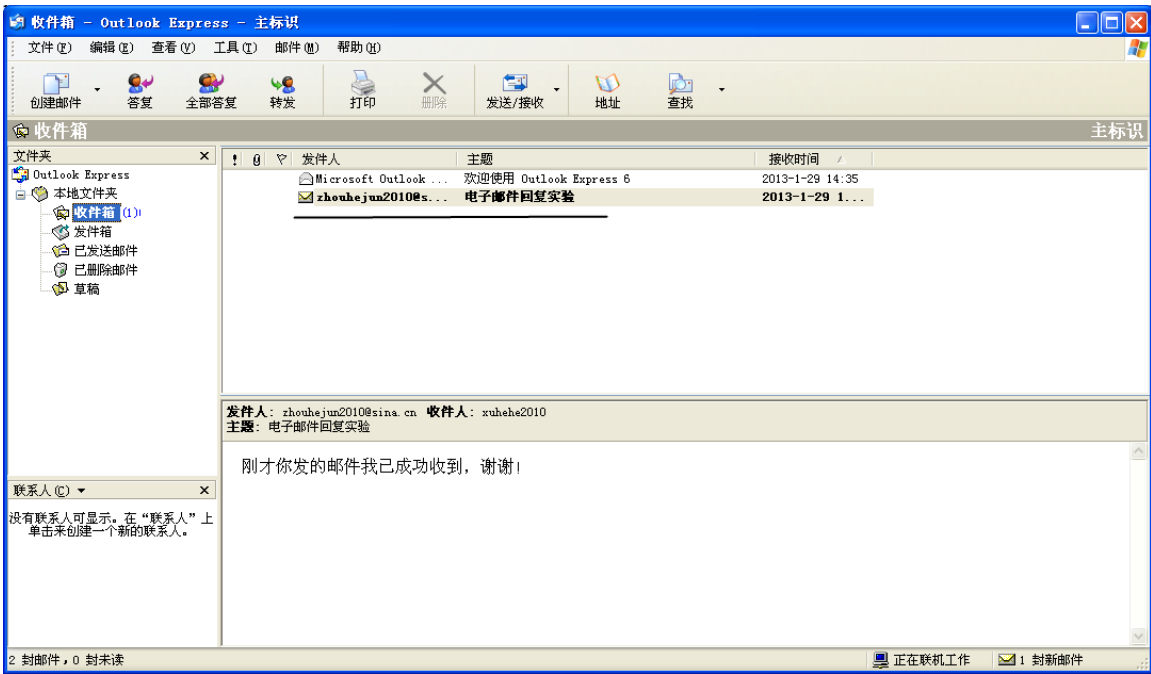


图 7.19 收到新浪用户回复的邮件

前面介绍过 POP3 的一个特点：只要用户从 POP 服务器读取了邮件，POP 服务器就将该邮件删除。照此说法，这个时候服务器应该已经自动从其上删去了这封邮件，下面就来验证一下。

再次从 Web 登录网易邮箱，如图 7.20 所示，邮箱已经空了，并且系统提示邮件已删除。

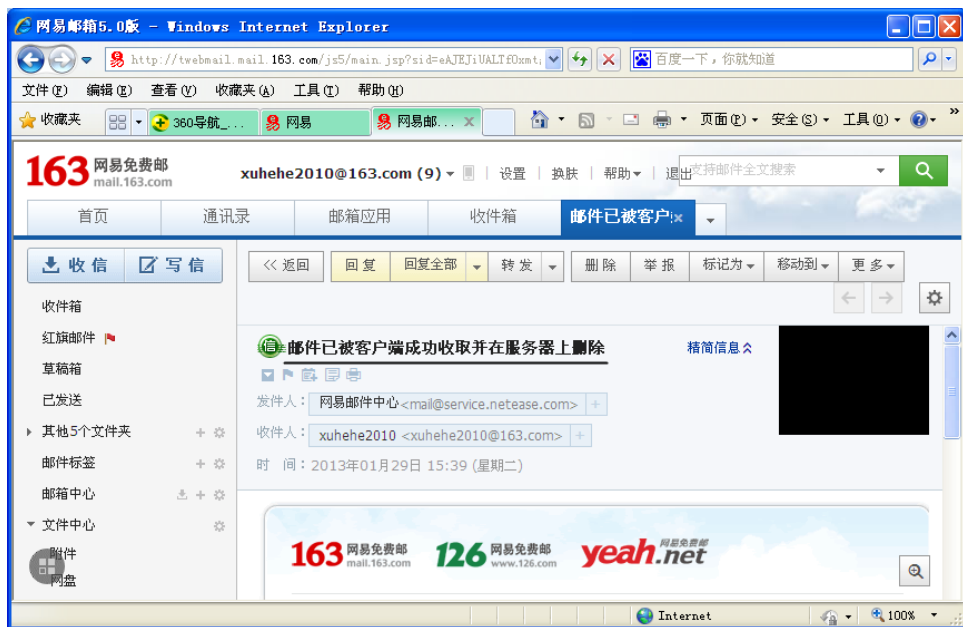


图 7.20 服务器上的副本已删除

通过以上这一系列实验，相信读者应该很容易理解图 7.15 所示邮件收发的整个流程了。

7.2 基于 MAPI 的邮件客户端开发

7.2.1 开发邮件程序的接口 MAPI

MAPI 是 Messaging Application Programming Interface 的缩写，它是由微软公司提供的一系列供使用者开发 Mail、Scheduling、Bulletin Board、Communication 程序的编程接口。其中 Mail 程序接口即邮件或邮件撰写应用程序接口，编程人员用它来创建邮件、撰写工作组应用程序（如电子邮件、计划、日程表和文档管理），它是一个开放和全面的邮件撰写接口。

MAPI 体系结构的基础是 COM，主要分为三大部分功能：Address Books、Transport 和 Message Store。Address Books 主要负责设置 E-mail type、protocol 等参数，Transport 负责文件的发送和接收等功能，Message Store 则负责发送、接收等信息的处理。

在使用 MAPI 设计程序时，首先必须在程序和 MAPI 之间建立一条或数条 Session（会话）；当 Session 建立好之后，客户端程序就可以使用 MAPI 所提供的功能了。本节主要介绍如何使用 Visual C++ 通过 MAPI 编写邮件客户端程序。MAPI 是包含在 Windows 之中的，因此不需要安装其他额外的部件，它有以下三种形式：

- SMAPI, Simple MAPI, 简单的 MAPI;
- CMC, Common Messaging Calls, 一般通信调用;
- 完整的 MAPI。

SMAPI 和 CMC 都包含在完整的 MAPI 中，当用户想执行一些高级操作，如编写自己的 E-mail 服务器时，必须使用完整的 MAPI。本例编写的是客户端程序，因此使用 SMAPI 就足够了。

用 MAPI 编写邮件程序有如下一套通行的编程步骤。

(1) 初始化 MAPI。

要使用 MAPI, 必须首先对它进行初始化。初始化包括装载 MAPI32.DLL 动态链接库、找到想要调用的 MAPI 函数地址、登录到电子邮件对象。

(2) 建立 MapiMessage 结构对象。

MapiMessage 结构是用来存储邮件内容的, 包括邮件主题、正文、时间日期、收件人和发件人地址等有关一封信件的全部参数和内容信息都从这个结构中获取(读邮件时)或向其中写入(写邮件时)。

(3) 读取或发送电子邮件。

读取邮件包括定位到第一封信、访问下一封信及从 MapiMessage 结构获得电子邮件的内容; 写信时在 MapiMessage 结构中设置接收者信息、写入邮件内容, 然后调用函数发送邮件。

(4) 释放内存。

在访问另一条信件以前应当释放内存, 否则会出现内存溢出。

7.2.2 邮件客户端程序开发

1. 创建工程、界面设计

创建 MFC 工程, 工程名为 SelfMailSndRcver (自制电子邮件收发器)。采用传统的对话框类型程序。因为微软 MAPI 接口封装了 Socket 使用 POP3 与邮件服务器交互的细节, 故不需要用户自己编写 Socket 程序去实现 POP3, 所以在“高级功能”页也不需要选中“Windows 套接字”复选框。

工程创建好后, 设计程序界面如图 7.21 所示。

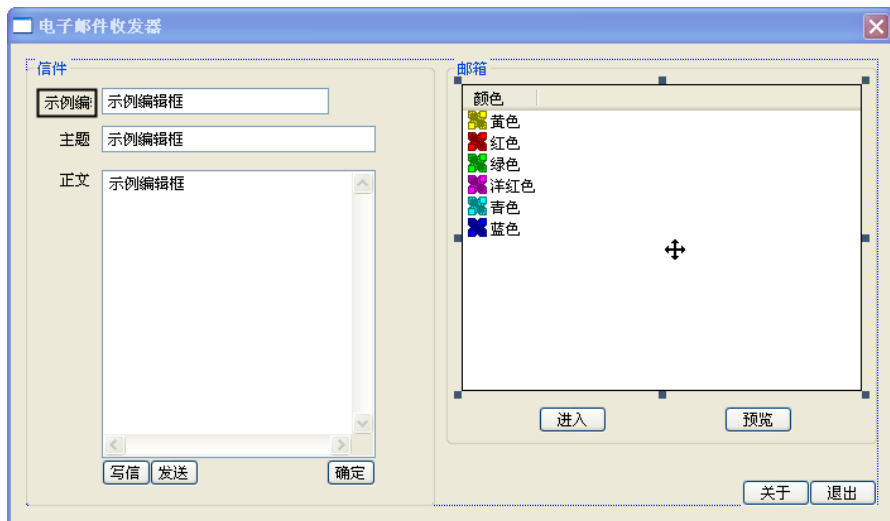


图 7.21 设计程序界面

可以看出, 这个电子邮件收发程序的功能是比较完整的: 可以进入信箱、显示邮件列表和查看信件内容, 既可以收信也可以写信和发信。

界面左半区信件组框中第一行文本框是用来填写邮箱地址的(写信时), 它同时也可用于显示邮箱地址(读邮件时), 这就要求文本框左侧的文字标签(如图 7.21 中框出)能够根据情况灵活地动态切换显示“收件人”和“发件人”。要做到这样的效果, 就不能使用静态文本(Static Text)

标签控件，而必须改用文本框（Edit Control）控件，具体操作为将其属性“Read Only”设置成“True”，即“只读”模式，并去掉文本框的边框（“Border”属性取“False”），就可以使它在外观上看起来与静态文本标签控件一模一样了。

读者大概还注意到，本例在画界面时使用了一个特别的控件，即列表控件（List Control）（图 7.21 中显示一列“黄色”、“红色”、“绿色”等颜色图标 的框）。这个控件的功能比较强大，专门应用在需要显示项目列表的场合，并且有多种样式可供选择。在 VC 集成开发环境的右下角“属性”窗口中可以设置选择列表控件的样式，如图 7.22 所示。

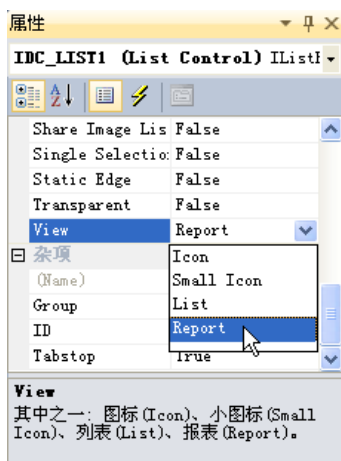


图 7.22 选择列表控件样式

在此设置为报表样式（将“View”属性设置为“Report”），便于显示邮件列表。
各界面控件变量见表 7.1。

表 7.1 电子邮件收发器界面控件变量

控 件 \ 变 量	Control	Value
“收（发）件人”动态文本标签	m_addressName	—
邮件地址文本框	m_sndRcvAddr	—
“主题”文本框	m_subject	strSubject
“正文”文本框	m_content	strContent
“邮箱”列表控件	m_recelst	—
“写信”按钮	m_writeLetter	—
“发送”按钮	m_send	—
“确定”按钮	m_confirm	—
“进入”按钮	m_enter	—
“预览”按钮	m_view	—

2. 声明 MAPI 函数指针

本例要使用 MAPI 函数接口，由于 MAPI32.DLL 是被动态装载的，因此不知道所要调用的函数地址，也就不能一开始就调用它们，而要通过函数名获得函数的地址，并在动态链接库中查找每一个函数并核实。因此首先必须为这些函数声明指针，以便后面的程序中可以成功调用这些

MAPI()函数。

在文件 SelfMailSndRcvrDlg.cpp 中声明 MAPI()函数指针，代码如下：

```
//定义 MAPI32.DLL 动态库中的函数原型
ULONG (PASCAL *lpfnMAPISendMail) (LHANDLE lhSession,
ULONG ulUIParam, lpMapiMessage lpMessage, FLAGS flFlags, ULONG ulReserved);

ULONG (FAR PASCAL *lpfnMAPILogon)(ULONG ulUIParam,
LPSTR lpszProfileName, LPSTR lpszPassword, FLAGS flFlags, ULONG ulReserved,
LPLHANDLE lplhSession);

ULONG (FAR PASCAL *lpfnMAPILogoff)(LHANDLE lhSession,
ULONG ulUIParam, FLAGS flFlags, ULONG ulReserved);

ULONG (FAR PASCAL *lpfnMAPIFreeBuffer)(LPVOID lpBuffer);

ULONG (FAR PASCAL *lpfnMAPIFindNext)(LHANDLE lhSession,
ULONG ulUIParam, LPSTR lpszMessageType, LPSTR lpszSeedMessageID, FLAGS flFlags,
ULONG ulReserved, LPSTR lpszMessageID);

ULONG (FAR PASCAL *lpfnMAPIReadMail)(LHANDLE lhSession,
ULONG ulUIParam, LPSTR lpszMessageID, FLAGS flFlags,
ULONG ulReserved, lpMapiMessage FAR *lppMessage);
```

为了在程序中保存已读邮件的内容，必须定义一个结构体来存储。在项目中添加一个 C++ 结构体 emailContent，其定义在头文件 emailContent.h 中，代码如下：

```
typedef struct emailContent
{
    char subject[100];
    char content[1000];
} EmailContent;
```

为了使程序能够最终访问 MAPI 接口及此处定义的这个结构体，还要在程序主对话框的头文件 SelfMailSndRcvrDlg.h 中包含两个头文件并且定义两个变量，代码如下：

```
#include "MAPI.h"
#include "emailContent.h"
HMODULE result;
EmailContent emailCont[100];
```

这两个变量都定义在类 CSelfMailSndRcvrDlg 中，其中 result 用于保存函数返回的句柄，emailCont 数组则是用来保存邮件内容的。要装载 MAPI，用户程序必须在运行时动态地装载一个动态链接库。LoadLibrary()函数提供了此功能，它定位一个动态链接库，并返回 HINSTANCE 句柄（需要保存该句柄），result 就是用来保存这个句柄的。

3. 邮件收发器的编程实现

启动邮件收发器时，主程序对话框初始化过程的代码在主对话框类 CSelfMailSndRcvrDlg 的 OnInitDialog()方法中，代码如下：

```
m_addressName.SetWindowTextA("收件人");
m_sndRcvAddr.EnableWindow(false);
m_subject.EnableWindow(false);
```

```

m_content.EnableWindow(false);
m_send.EnableWindow(false);
m_confirm.EnableWindow(false);
//初始化列表控件
m_recelst.SetExtendedStyle(LVS_EX_GRIDLINES);
m_recelst.InsertColumn(0,"发件人",LVCFMT_LEFT,100);
m_recelst.InsertColumn(1,"主题",LVCFMT_LEFT,140);
m_recelst.InsertColumn(2,"时间",LVCFMT_LEFT,120);
HRESULT hr = ::CoInitialize(NULL);
if(!SUCCEEDED(hr))
    return FALSE;
m_recelst.EnableWindow(false);
m_view.EnableWindow(false);

```

列表控件初始化为三个栏目：“发件人”、“主题”和“时间”。
初始化后运行程序，界面效果如图 7.23 所示。

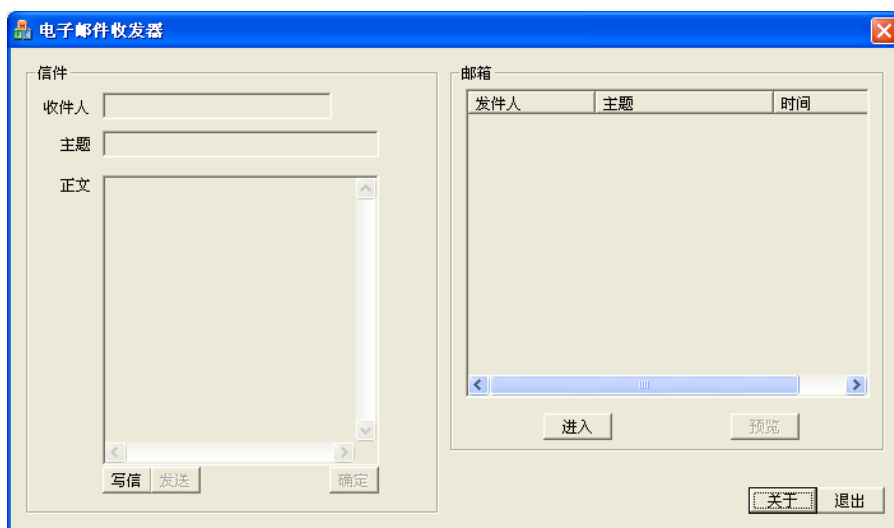


图 7.23 程序初始化后的界面

现在的界面上只有“写信”和“进入”按钮可用，先来编写“写信”按钮的事件过程，代码如下：

```

void CSelfMailSndRcvrDlg::OnWriteLetter()
{
    UpdateData();
    m_writeLetter.SetWindowTextA("写信");
    m_writeLetter.EnableWindow(false);
    //如果是在阅读过别人的信件后单击此按钮，表示要回复此信
    if(strSubject != "")
    {
        m_addressName.SetWindowTextA("收件人");
        m_subject.SetWindowTextA("回复:" + strSubject);
        CString reply;
        //系统自动生成回复框架
    }
}

```



```

        reply = "您好!信已收到\r\n\r\n 主题为:\r\n      " + strSubject;
        reply += "\r\n\r\n 你在信中说:\r\n      " + strContent;
        reply += "\r\n\r\n 现在我答复如下:\r\n      ";
        m_content.SetWindowTextA(reply);
        m_content.SetFocus();
    }
    //如果是主动写信给某人, 单击此按钮后, 信件编辑区变为可用状态
    else
    {
        m_sndRcvAddr.EnableWindow(true);
        m_subject.EnableWindow(true);
        m_content.EnableWindow(true);
    }
    m_confirm.EnableWindow(true);
}

```

该程序可以写信, 也可以收信, 单击“进入”按钮, 用户就能进入自己的邮箱查看信件了。“进入”按钮的事件过程代码如下:

```

void CSelfMailSndRcvDlg::OnRcv()
{
    m_recelst.EnableWindow(true);
    result = LoadLibrary("mapi32.dll"); //加载动态库 ①
    //获取函数指针 ②
    (FARPROC&)lpfnMAPILogon = GetProcAddress(result,"MAPILogon");
    (FARPROC&)lpfnMAPIFindNext = GetProcAddress(result,"MAPIFindNext");
    (FARPROC&)lpfnMAPIReadMail = GetProcAddress(result,"MAPIReadMail");
    (FARPROC&)lpfnMAPIFreeBuffer = GetProcAddress(result,"MAPIFreeBuffer");
    (FARPROC&)lpfnMAPILogoff = GetProcAddress(result,"MAPILogoff");
    unsigned long a;
    ULONG IResult ;
    MapiMessage *m_messageInfo[100]; //这个结构是存储邮件全部信息的
    char pMessageID [513];
    int nCount = 0;
    long nFlags = MAPI_SUPPRESS_ATTACH;
    //先来统计邮箱中的邮件数
    lpfnMAPILogon(0,NULL,NULL,0,0,&a); //开始一个会话 ③
    IResult = lpfnMAPIFindNext(a,NULL,NULL,NULL,
        MAPI_GUARANTEE_FIFO,0,pMessageID);
    //找到第一封没有阅读的电子邮件 ④
    while(IResult == SUCCESS_SUCCESS)
    {
        nCount++; //统计邮件数
        IResult = lpfnMAPIFindNext(a,NULL,NULL,pMessageID,
            MAPI_GUARANTEE_FIFO,0,pMessageID);
        //定位到下一封没有阅读的邮件 ⑤
    }
    lpfnMAPILogoff(a,0,0,0); //结束一个会话
    //初始化 emailContent 结构体数组, 本程序最多可保存 100 封邮件的内容
}

```

```

for(int i = 0;i < 100;i ++)
{
    memset(emailCont[i].subject,0,sizeof(emailCont[i].subject));
    memset(emailCont[i].content,0,sizeof(emailCont[i].content));
}
//在统计完邮箱中的邮件总数后，第二个会话才开始正式读取邮件内容 ⑥
lpfnMAPILogon(0,NULL,NULL,0,0,&a); //开始另一个会话
IResult = lpfnMAPIFindNext(a,NULL,NULL,NULL,
MAPI_GUARANTEE_FIFO,0,pMessageID);

m_messageInfo[0] = new MapiMessage;
memset(m_messageInfo[0],0,sizeof(MapiMessage));
IResult=lpfnMAPIReadMail(a,NULL,pMessageID,nFlags,0,&m_messageInfo[0]);
//读邮件 ⑦

//保存发件人地址、主题、日期、正文 4 个字段
CString strAddr,strSubj,strDatRcv,strCont;
strAddr = m_messageInfo[0]->lpOriginator->lpszAddress;
strSubj = m_messageInfo[0]->lpszSubject;
strDatRcv = m_messageInfo[0]->lpszDateReceived;
strCont = m_messageInfo[0]->lpszNoteText;
//将邮件的主题和正文提取出来单独保存在 emailContent 结构体中，以便随时读取显示
strcpy(emailCont[0].subject,strSubj);
int pos;
pos = strCont.Find(';');
strCont = strCont.Right(strCont.GetLength() - (pos + 1));
strcpy(emailCont[0].content,strCont);
//读取的邮件放到邮箱中
m_recelst.DeleteAllItems();
int count = m_recelst.InsertItem(1,"");
m_recelst.SetItemText(count,0,strAddr);
m_recelst.SetItemText(count,1,strSubj);
m_recelst.SetItemText(count,2,strDatRcv);
//释放内存 ⑧
lpfnMAPIFreeBuffer(m_messageInfo[0]);
//从第二封邮件开始往后的邮件就都可以用循环程序读取了 ⑨
for(int i = 1;i < nCount;i ++)
{
    IResult = lpfnMAPIFindNext(a,NULL,NULL,pMessageID,
MAPI_GUARANTEE_FIFO,0,pMessageID);

    m_messageInfo[i] = new MapiMessage;
    memset(m_messageInfo[i],0,sizeof(MapiMessage));
    //读邮件

    IResult=lpfnMAPIReadMail(a,NULL,pMessageID,nFlags,0,&m_messageInfo[i]);
    CString strAddr,strSubj,strDatRcv,strCont;
    strAddr = m_messageInfo[i]->lpOriginator->lpszAddress;
    strSubj = m_messageInfo[i]->lpszSubject;
    strDatRcv = m_messageInfo[i]->lpszDateReceived;
    strCont = m_messageInfo[i]->lpszNoteText;

```

```

strcpy(emailCont[i].subject,strSubj);
int pos;
pos = strCont.Find(';');
strCont = strCont.Right(strCont.GetLength() - (pos + 1));
strcpy(emailCont[i].content,strCont);

int count = m_recelst.InsertItem(1,"");
m_recelst.SetItemText(count,0,strAddr);
m_recelst.SetItemText(count,1,strSubj);
m_recelst.SetItemText(count,2,strDatRcv);

lpfnMAPIFreeBuffer(m_messageInfo[i]);
}

lpfnMAPILogoff(a,0,0,0);           //结束会话
FreeLibrary(result);             //卸载动态库
m_view.EnableWindow(true);
m_enter.SetWindowTextA("刷新");    //“进入”按钮动态变为“刷新”按钮
}

```

下面来详细分析一下上面的程序代码（重要语句加黑显示，程序关键步骤在注释中有标号①～⑨，请读者对照标号看下面的代码解析说明文字）。

① MAPI 是微软的产品，微软组件大多使用一种称为“动态链接库”的机制提供，使用 MAPI 前用户也必须首先装载 MAPI32.DLL 动态链接库。LoadLibrary()函数提供了此项功能，它的语法为 LoadLibrary (lpLibFileName);，其中 lpLibFileName 为 LPCTSTR 结构变量，是所要调用的库的路径和名称。相应地，在使用完 MAPI 后用 FreeLibrary 函数卸载它的动态链接库。

② 要使程序代码可以直接调用 MAPI 接口函数，仅仅加载 MAPI 库还不行，还必须找到 MAPI32.DLL 函数的入口地址，并将它们保存在函数指针变量里。为了决定每一个函数的地址，必须为每一个函数调用 GetProcAddress。

GetProcAddress 的语法为：

```
GetProcAddress (hModule, lpProcName);
```

其中，hModule 为 HMODULE 结构，是所调用 DLL 模块的句柄；lpProcName 为 LPCSTR 结构，是函数名称。本程序中一共要使用 5 个 MAPI 函数（MAPILogon、MAPIFindNext、MAPIReadMail、MAPIFreeBuffer 和 MAPILogoff），必须连续调用 GetProcAddress 5 次来分别获得它们的入口地址。

③ 用户必须在电子邮件系统中登录，才能实现 MAPI 的各种功能。登录通常使用 MAPI 提供的函数 lpfnMAPILogon，它的语法为：

```
lpfnMAPILogon (lpszProfileName, lpszPassword, flFlags, ulReserved, lplhSession );
```

其中，lpszProfileName 指向一个 256 个字符以内的登录名称，lpszPassword 指向密码，它们均为 LPTSTR 结构；flFlags 为 FLAGS 结构，ulReserved 必须为 0，lplhSession 为输出 SMAPI 的句柄。本程序只是实现邮箱的基本功能，不进行登录用户名和密码的验证，因此各参数采用默认值。对应地，在每次访问邮件系统完毕之后需要注销，调用 MAPILogoff()结束前一次的会话。

④ 要找到第一封信，需要使用 MAPIFindNext 函数，该函数声明代码如下：

```

ULONG FAR PASCAL MAPIFindNext(LHANDLE lhSession, ULONG ulUIParam,
                               LPTSTR lpszMessageType, LPTSTR lpszSeedMessageID,

```

FLAGS flFlags, ULONG ulReserved, LPTSTR lpszMessageID)

其中, lhSession 为提交 SMAIL 的会话句柄; ulUIParam 为父窗体的句柄; lpszMessageType 指向一个字符串, 用来鉴别邮件类型, 并加以查找; lpszSeedMessageID 为指向起始信息 ID 的指针, 其值为 NULL 时, MAPIFindNext 获得第一封电子邮件; flFlags 的值取 MAPI_GUARANTEE_FIFO, 表示按邮件发送的时间顺序接收电子邮件; ulReserved 必须为 0; lpszMessageID 为输出值, 它是指向信息 ID 地址的指针。

⑤ 定位到下一封信件依然使用 MAPIFindNext 函数, 只是参数 lpszSeedMessageID 不能再取 NULL 了, 而是必须取上次调用该函数时所获得的参数 lpszMessageID 的输出值, 它是指向信息 ID 地址的指针, 指向本次要访问的邮件。

⑥ 本程序后面需要用到邮箱中的邮件总数的信息以便循环读取邮件。为了使程序代码结构更清晰易读, 也是为了实现上的方便, 本例前后使用两个会话分别访问邮件系统。两次登录每次专注于做一件事: 第一次登录的主要任务是统计邮箱中的邮件数, 第二次登录的会话才开始正式读取邮件内容。

⑦ 前面的 MAPIFindNext 用于定位第一封或下一封电子邮件并返回信件 ID, 而当信件 ID 被获取后, 就可以调用 MAPIReadMail 阅读实际的 E-mail 信息了。MAPIReadMail 的函数声明代码如下:

```
ULONG FAR PASCAL MAPIReadMail(LHANDLE lhSession, ULONG ulUIParam,
    LPTSTR lpszMessageID, FLAGS flFlags, ULONG ulReserved, lpMapiMessage FAR * lppMessage);
```

其中, lppMessage 为指向 MapiMessage 的指针; 除 flFlags 外的其他参数与 lpfnFindNext 函数的同名参数意义相同; flFlags 参数的值 MAPI_SUPPRESS_ATTACH 表示 MAPIReadMail 函数不复制附件, 而是将邮件文本写入 MapiMessage 结构中。前面说过 MapiMessage 结构是存储邮件全部信息的, 如果调用成功, 就可以访问 MapiMessage 结构了 (使用 pMessage->)。

MapiMessage 结构各字段存储的信息内容见表 7.2。

表 7.2 MapiMessage 结构各字段内容含义

字段名 pMessage->	信息内容含义
ulReserved	0
lpszSubject	邮件主题
lpszNoteText	邮件正文
lpszMessageType	邮件类型
DateReceived	接收时间
lpszConversationID	邮件所属的会话线程 ID
lpOriginator	指向 MapiRecipDesc 结构 (包含发件人信息)
nRecipCount	信件的数目
lpRecips	指向 MapiRecipDesc 结构数组 (包含接收者信息)
nFileCount	附件数量
lpFiles	指向 MapiFileDesc 结构数组, 每一个结构包含一个文件附件

在编程中, 可以根据应用需要选取某几个特定字段进行访问。本例程序需要实现显示邮件内容的功能, 要显示的项包括发件人地址、邮件主题、正文、接收时间, 其中发件人地址存储在 lpOriginator->lpszAddress 中, 邮件主题在 lpszSubject 中, 正文为 lpszNoteText 字段, 接收时间是

lpSzDateReceived, 因此只要访问这几个字段就够了。在程序中定义变量 strAddr、strSubj、strDatRcv 和 strCont, 分别获取发件人地址、邮件主题、收件时间和邮件的正文内容。

⑧ 在访问另一封信件以前应当释放内存, 否则会出现内存溢出。可调用函数 lpfnMAPIFreeBuffer 释放内存。

⑨ 从④和⑤可知, 访问第一封信件与定位后续信件虽然都是使用 MAPIFindNext 函数, 但它们的调用参数取值有所不同。下面将这两句调用语句从程序段中特别摘出并对比如下:

```
//访问第一封信
lResult = lpfnMAPIFindNext(a,NULL,NULL,NULL,
                          MAPI_GUARANTEE_FIFO,0,pMessageID);

//访问后续信件
lResult = lpfnMAPIFindNext(a,NULL,NULL,pMessageID,
                          MAPI_GUARANTEE_FIFO,0,pMessageID);
```

显然, 第四个参数不同, 原因已经在前文中说明。这就是本程序没有把访问第一封信的代码段直接置入循环中的原因。从第二封邮件开始往后的邮件就都可以用循环程序读取了。

通过对上述一系列“进入”邮箱程序代码的详尽解析, 相信读者已经基本掌握了 MAPI 接口的使用。接着再看本例程序的其他代码段, 应该容易理解得多。

进入邮箱后, 选择某封信件的条目, 再单击“预览”按钮, 在主对话框左区就会显示这封邮件的内容, 包括发件人地址、主题和正文。

“预览”按钮的事件过程代码如下:

```
void CSelfMailSndRcvDlg::OnReadEmail()
{
    m_addressName.SetWindowTextA("发件人");    //标签动态显示为“发件人”
    m_sndRcvAddr.EnableWindow(true);
    m_subject.EnableWindow(true);
    m_content.EnableWindow(true);
    //获取邮箱中用户选中信件的主题和发件人地址
    POSITION pos = m_recelst.GetFirstSelectedItemPosition();
    int nItem = m_recelst.GetNextSelectedItem(pos);
    CString strSubj,strAddr;
    strSubj = m_recelst.GetItemText(nItem,1);    //获取邮件主题
    strAddr = m_recelst.GetItemText(nItem,0);    //获取发件人地址
    for(int i = 0;i < 100;i ++)
    {
        //到 emailContent 结构体数组中查找邮件正文
        if(emailCont[i].subject == strSubj)
        {
            //显示邮件地址、主题、正文
            m_sndRcvAddr.SetWindowText(strAddr);
            m_subject.SetWindowText(emailCont[i].subject);
            m_content.SetWindowText(emailCont[i].content);
            break;
        }
    }
    m_writeLetter.EnableWindow(true);
    m_writeLetter.SetWindowTextA("回复");    //“写信”按钮变成“回复”按钮
```

```

        m_send.EnableWindow(false);
        m_confirm.EnableWindow(false);
    }

```

在用户预览信件时，“写信”按钮动态变成“回复”按钮，此时用户可以回复这封邮件。单击“回复”（原来的“写信”）按钮，程序会自动生成回信的格式框架，用户只需在框架中填写回信的内容即可（这与实用的邮件产品在功能上有相似之处）。本例程序通过 `SetWindowTextA()` 方法在程序运行时动态改变界面按钮的文本标签，使一个按钮具有多重功用，这也是很多产品化软件经常使用的界面风格，读者可以在平时学习编程时多加借鉴和模仿。

写完回信后，单击“确定”按钮加以确认。“确定”按钮的事件过程代码如下：

```

void CSelfMailSndRcvrDlg::OnConfirm()
{
    UpdateData();
    m_confirm.EnableWindow(false);
    m_sndRcvAddr.EnableWindow(false);    //地址栏不可用
    m_subject.EnableWindow(false);       //主题栏不可用
    m_content.EnableWindow(false);       //正文框不可用
    m_send.EnableWindow(true);           //“发送”按钮可用
}

```

此时邮件编辑区（包括地址栏、主题栏、正文框）均变为不可用状态，因为用户已经确认邮件内容，就不可以再修改了，只能单击“发送”按钮发送信件。

“发送”按钮的事件过程代码如下：

```

void CSelfMailSndRcvrDlg::OnSendEmail()
{
    result = LoadLibrary("mapi32.dll");    //加载动态库
    //获取函数指针
    (FARPROC&)lpfnMAPILogon = GetProcAddress(result,"MAPILogon");
    (FARPROC&)lpfnMAPISendMail = GetProcAddress(result,"MAPISendMail");
    (FARPROC&)lpfnMAPIFreeBuffer = GetProcAddress(result,"MAPIFreeBuffer");
    (FARPROC&)lpfnMAPILogoff = GetProcAddress(result,"MAPILogoff");
    unsigned long a;
    lpfnMAPILogon(0,NULL,NULL,0,0,&a);    //开始一个会话
    char* pcontext;                        //记录邮件正文
    CString str;
    m_content.GetWindowText(str);
    pcontext = str.GetBuffer(0);
    ULONG lresult ;
    MapiMessage *m_messageInfo;           //定义一个信息结构指针
    m_messageInfo = new MapiMessage;
    memset(m_messageInfo,0,sizeof(MapiMessage));    //初始化 m_messageInfo
    CTime time = CTime::GetCurrentTime();    //获取当前时间
    CString ctime = time.Format("%y/%m/%d/%H");
    char date[50];
    strcpy(date,ctime);

    CString subject ;

```

```

m_subject.GetWindowText(subject); //获取主题

CString receiver,temp;
m_sndRcvAddr.GetWindowText(temp); //获取收件人信息
receiver = temp.Left(temp.Find('@')); //获取用户账户
char addr1[100]; //记录用户账户
char addr2[100]; //记录 SMTP 服务器
strcpy(addr1,receiver);

receiver = "SMTP:" + temp;
strcpy(addr2,receiver);
//定义接收者信息
MapiRecipDesc m_receiver = {0,MAPI_TO,addr1,addr2,0,NULL}; //建立并设置 MapiMessage 结构对象
m_messageInfo->lpszNoteText = pcontext; //设置邮件正文
m_messageInfo->ulReserved = 0; //保留，必须为 0
m_messageInfo->lpszSubject = subject.GetBuffer(0); //设置主题
m_messageInfo->lpszDateReceived = date; //设置邮件发送时间
m_messageInfo->lpszConversationID = NULL; //邮件所属线程一个字符串指针
m_messageInfo->flFlags = MAPI_SENT; //邮件状态标记
m_messageInfo->lpOriginator = NULL; //发送者信息
m_messageInfo->nRecipCount = 1; //接收者人数
m_messageInfo->nFileCount = 0; //附件数
m_messageInfo->lpRecips = &m_receiver; //设置接收者信息
m_messageInfo->lpszMessageType = NULL; //邮件类型
lresult = lpfnMAPISendMail(a,0,m_messageInfo,0,0); //发送邮件
if (lresult != SUCCESS_SUCCESS)
{
    MessageBox("操作失败.", "提示", 64);
}
else
{
    MessageBox("邮件发送成功.", "提示", 64);
    m_content.Clear();
}
lpfnMAPIFreeBuffer(m_messageInfo);
lpfnMAPILogoff(a,0,0,0);
FreeLibrary(result);
m_addressName.SetWindowTextA("收件人");
m_writeLetter.EnableWindow(true);
m_send.EnableWindow(false);
m_sndRcvAddr.SetWindowTextA("");
m_sndRcvAddr.EnableWindow(false);
m_subject.SetWindowTextA("");
m_subject.EnableWindow(false);
m_content.SetWindowTextA("");
m_content.EnableWindow(false);
}

```

发送邮件的过程与一般的访问 MAPI 函数的过程一样, 唯一不同的是要自己建立 MapiMessage 结构对象并设置其中每一个字段的值。另外, 还要使用 MAPISendMail 发送电子邮件。函数 MAPISendMail 的声明代码如下:

```
ULONG FAR PASCAL MAPISendMail (LHANDLE lhSession, ULONG ulUIParam,  
                                lpMapiMessage lpMessage, FLAGS flFlags, ULONG ulReserved )
```

其中, flFlags 的允许值为 MAPI_DIALOG、MAPI_LOGON_UI 和 MAPI_NEW_SESSION, 其意义与前几个函数中同名标识的意义相同。

7.2.3 网络邮件收发实验

启动邮件客户端, 单击“写信”按钮, 信件编辑区变为可用状态, 如图 7.24 所示。

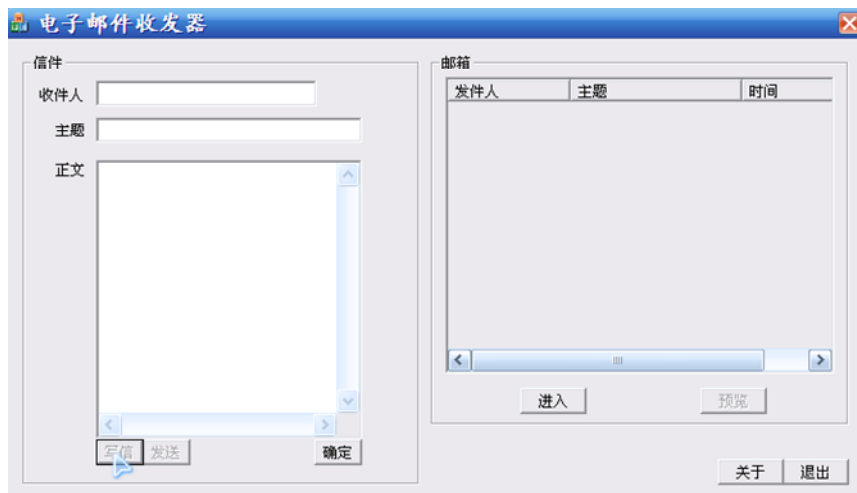


图 7.24 进入编辑状态

这个软件是与网易邮箱关联的, 我们用它向新浪邮箱发送邮件。如图 7.25 所示, 在信件编辑区填写收件人地址为前面注册的那个新浪信箱地址 zhouhejun2010@sina.cn, 邮件主题为“周末出去爬山”, 开始写信。

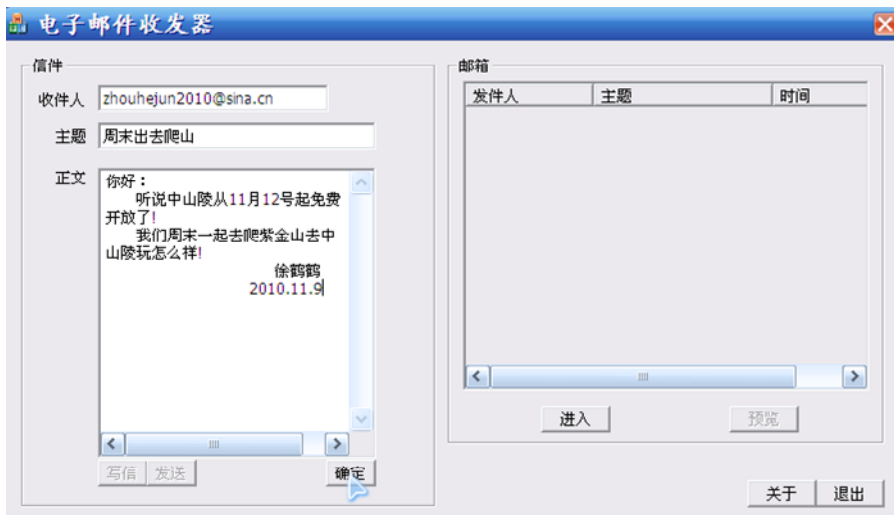


图 7.25 写信

编辑完信的正文后单击“确定”按钮，此时编辑区锁定，而“发送”按钮变为可用，如图 7.26 所示。



图 7.26 信件待发

单击“发送”按钮向新浪邮箱发出这封邮件。

出现“邮件发送成功”的提示消息（如图 7.27 所示）。现在登录新浪邮箱查看是否收到信，如图 7.28 所示，邮箱里有一封未读邮件，打开收件夹，果然就是刚才发的那封邮件！说明发送成功了。



图 7.27 信件发出

可以在线阅读收信内容（如图 7.29 所示），现在再用新浪邮箱给网易用户写一封回信（如图 7.30 所示），约定周末爬山时间。

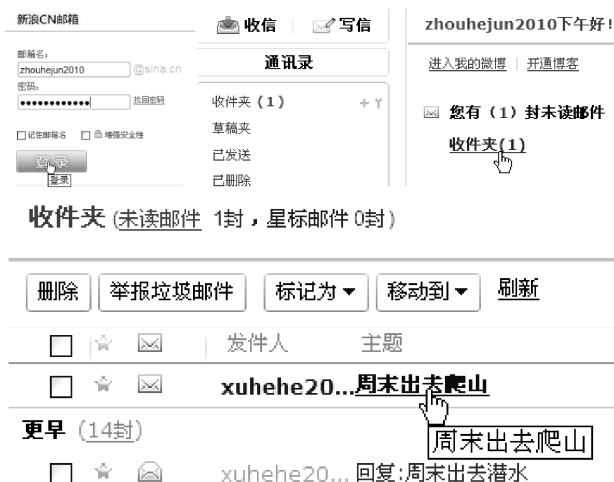


图 7.28 对方成功接收

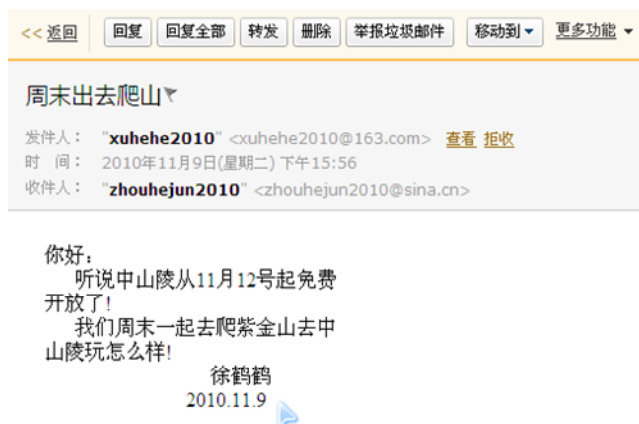


图 7.29 阅读收信内容

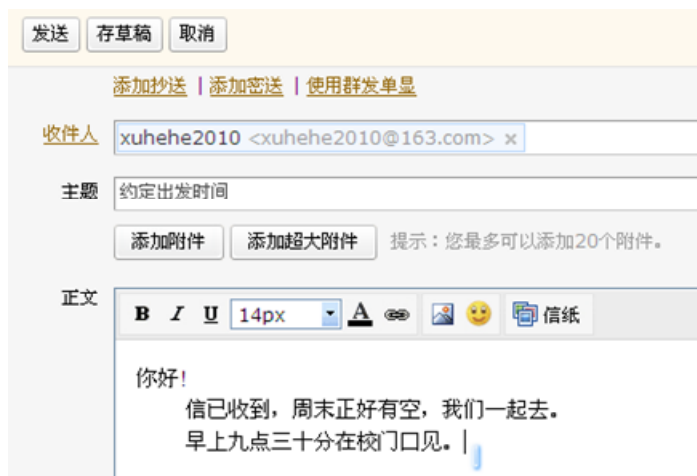


图 7.30 写回信

发送这封信，接下来用我们的客户端接收。单击“进入”按钮从客户端登录网易邮箱，可以看到邮箱中的信件列表，此时“进入”按钮变为“刷新”按钮，如图 7.31 所示。

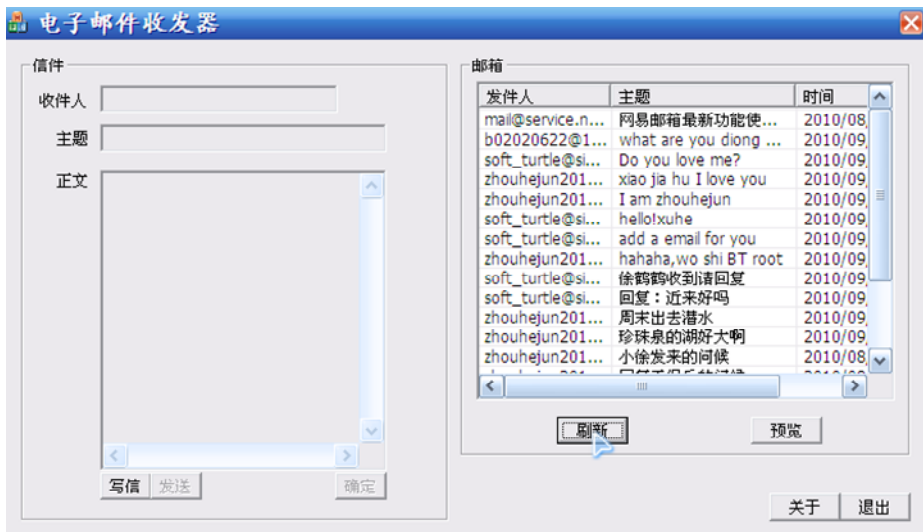


图 7.31 从客户端登录网易邮箱

但是在信件列表中却找不到刚刚从新浪发出的那封邮件，这是为什么呢？我们打开 Outlook Express 看看有没有。

在 Outlook Express 里倒是有这封邮件（如图 7.32 所示），为什么 Outlook Express 收到了而我们自己开发的客户端却收不到呢？下面来刷新一下自己客户端的邮箱看看能不能收到，如图 7.33 所示。

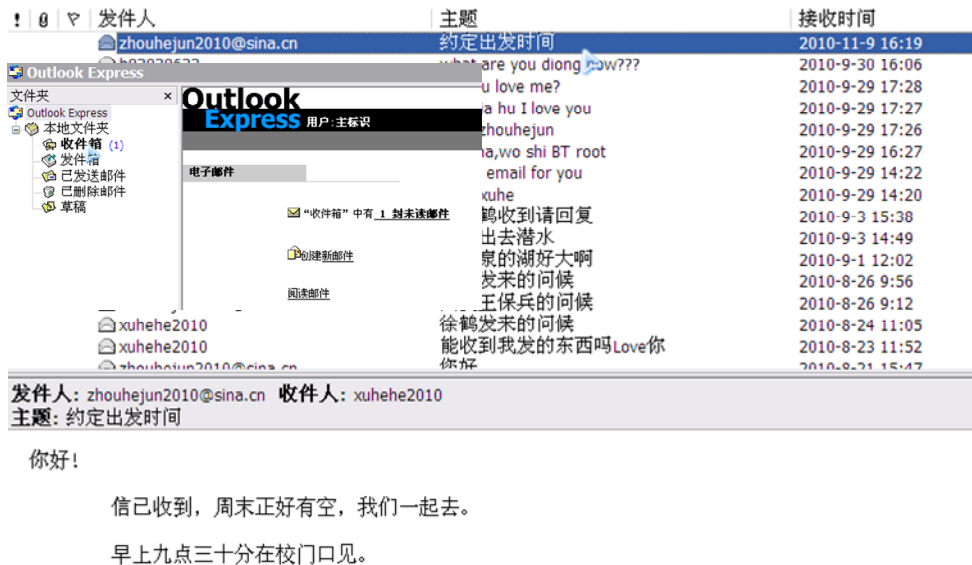


图 7.32 Outlook Express 能收到邮件

终于收到了！经多次实验，发现每次只有当先启动 Outlook Express 之后再进入我们自己开发的客户端程序的邮箱或刷新邮箱，才能看到外网信箱发来的邮件。

有关资料揭示：使用微软 MAPI 函数接口开发的邮件收发客户端程序与 Outlook Express 共用的是同一套底层 API 函数。为了安全考虑，微软 Windows 系统不允许第三方程序私自启动 API 库，只有在 Outlook Express 启动了这个库的情况下才能提供给其他程序调用。

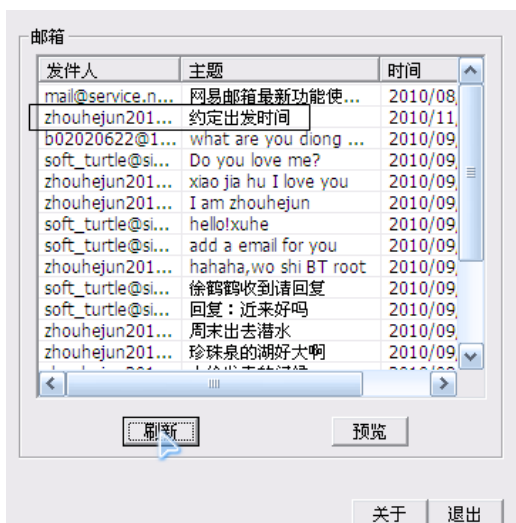


图 7.33 客户端刷新后收到邮件

选中刚刚收到的邮件，单击“预览”按钮，邮件的内容显示在左边信件编辑区，如图 7.34 所示。

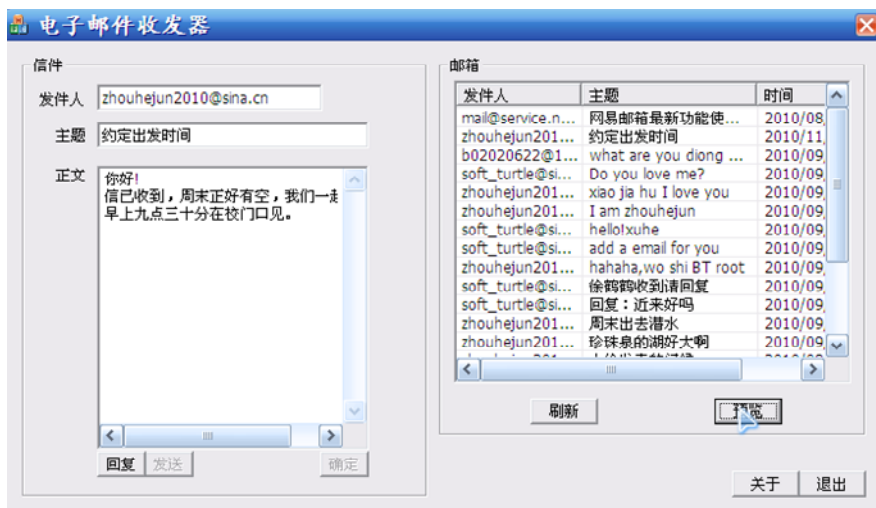


图 7.34 预览邮件

在读完这封信后，可以快速回复。单击“回复”按钮，程序自动生成回信的框架（如图 7.35 所示），图中框出的主题和正文文字都是程序自动生成的，用户只要在“现在我答复如下：”后面填写回信的内容即可（目前的邮箱产品都有这样的功能）。

写好回复内容后发出去，单击“确定”和“发送”按钮，与前面的发信过程完全一样。

在此还要指出一件事，前面在发信时并没有先启动 Outlook Express，但当时却成功发送了邮件而没有产生任何异常，为什么接收邮件就不行了呢？那是因为在测试这个程序之前对 Outlook Express 进行了设置。如图 7.36 所示，启动 Outlook Express，选择主菜单“工具”→“选项”命令，打开“选项”对话框。

在弹出的“选项”对话框的“安全”选项卡的“病毒防护”栏里取消选择“当别的应用程序试图用我的名义发送电子邮件时警告我”复选框（如图 7.37 所示），这样当使用用户自制的客户端程序发邮件时，Outlook Express 就不会弹出警告框阻止了。

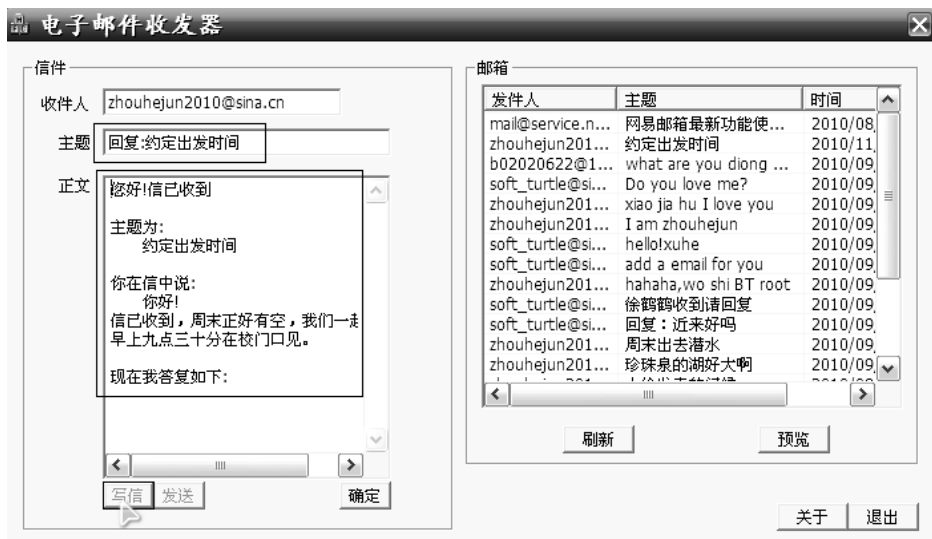


图 7.35 回信框架

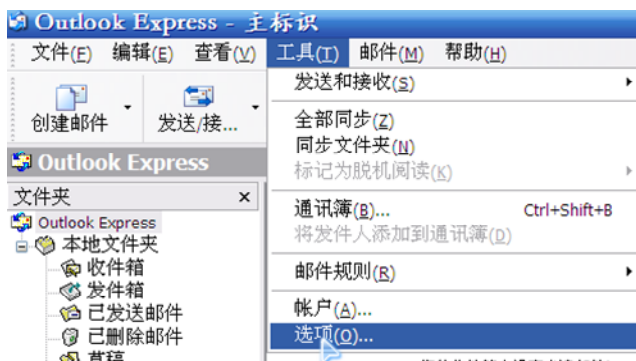


图 7.36 打开“选项”对话框

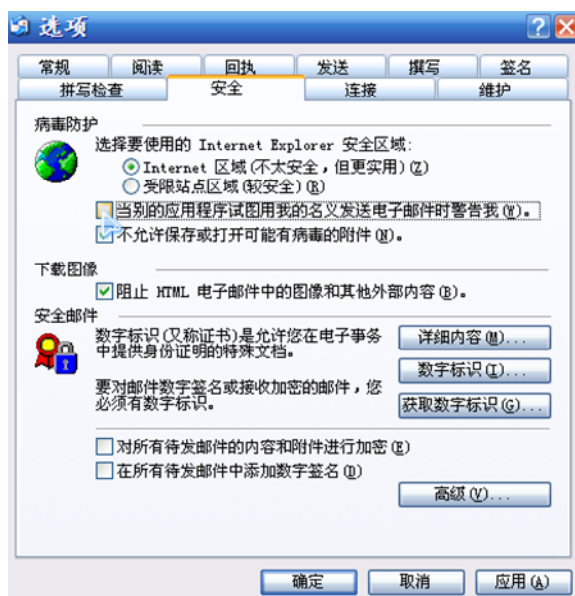


图 7.37 设置安全选项

但接收邮件仍然必须在启动 Outlook Express 的前提下才能进行, 为了系统安全考虑, Windows 是不允许来路不明的程序私自接收邮件的。

7.3 基于 POP3 的邮件接收程序

我们已经知道了使用微软提供的 MAPI 接口配合 Outlook Express 客户端可以开发出电子邮件收发程序。那么, 可不可以直接按照协议编程来实现同样的邮件收发功能呢? 答案是肯定的。本节就按照 POP3 协议的规范来开发一个邮件接收程序。

7.3.1 POP3 原理

POP3 (Post Office Protocol 3) 是邮局协议版本 3 的缩写, 是一个从邮件服务器的邮箱中取邮件到本地主机的协议。它最初是在 1984 年发表的 RFC918 中定义的, 1985 年的 RFC937 发表了第二个版本。随着 POP 的广泛使用, 1988 年的 RFC1081 又发表了它的第三个版本, 简称 POP3, 当前使用的标准是 RFC1939。

1. POP3 协议的工作过程

POP3 工作于客户端/服务器模式, 客户端程序工作在要取邮件的主机上, 服务器程序工作在提供 POP3 服务的主机上。工作时, 服务器在提供 POP3 服务的 TCP 端口 110 上守候客户端的请求, 当客户端主机需要从服务器上取邮件时, 它向服务器发出建立一条 TCP 连接请求。当连接建立后, POP3 发送确认消息。客户端和 POP3 服务器之间通过交互命令和响应, 就可以进行邮件的下载等处理工作了, 工作完成后关闭 TCP 连接。

在连接建立后, 客户端与邮件服务器之间使用 POP3 会话的过程可以分为 3 个阶段。

(1) 认证阶段。这是连接建立好以后首先要进行的第一步操作。一个邮件服务器为很多用户提供服务, 也就是说, 一个邮件服务器上有很多用户邮箱, 每个用户只有提供了正确的用户名和密码后, 才有权访问自己的邮箱。

(2) 邮件操作阶段。如果用户通过了认证, 就相当于打开了服务器上的用户邮箱, 用户就有权进行检索、下载或删除邮件等操作了。

(3) 更新阶段。当客户端发送了 QUIT (下面介绍) 命令后, 系统就进入更新阶段, POP3 服务器释放在操作阶段中取得的资源, 并将逻辑删除 (加了删除标记) 的邮件进行物理删除, 然后发送消息, 关闭客户端与服务器之间的 TCP 连接。邮件处理的会话过程结束。

2. POP3 会话命令与应答

POP3 的命令由可打印的 ASCII 字符组成, 它们之间用空格分隔。命令一般由 3~4 个字母组成, 一个命令可以带有一些参数, 每个参数可长达 40 个字符。所有命令以<CR><LF>结束。POP3 的命令见表 7.3。

表 7.3 POP3 命令

命 令	说 明
USER	USER 命令用于对用户名进行确认, 客户端必须首先发送 USER 命令, 告诉 POP3 服务器要操作的邮箱用户名
PASS	当客户端收到对 USER 命令的“确认”响应后, 就可以发送 PASS 命令, 告诉 POP3 服务器用户邮箱的密码
QUIT	服务器物理删除已经加了删除标记的邮件, 然后关闭连接
STAT	查询客户端邮箱中邮件的总长度和邮件总数

续表

命 令	说 明
UIDL	返回邮件的唯一标识符，被返回的行称为信件的“独立-ID 表”
LIST	列出各邮件长度
RETR	从邮箱中取出（下载）指定编号的邮件
DELE	对指定编号的邮件加上删除标记
NOOP	服务器只返回一个有效的应答，而不进行任何操作
RSET	复位操作，清除所有带有删除标记的邮件的删除标记

POP3 的响应非常简单，只有两种状态码：一种是表示命令已成功执行或 POP3 服务器准备就绪的“确定”，它以“+OK”开始；另一种是表示错误或不能执行的命令，它以“-ERR”开始。一般情况下这两种形式的应答都可以在其后跟一些附加信息以对响应进行具体说明。如果响应信息包含多行，最后一行是行结束标记<CRLF><CRLF>。

3. POP3 工作实例

现在，如果 xuhehe2010@163.com 要从网易邮件服务器 pop.163.com 中取邮件，则客户端 xuhehe2010 启动本地计算机上的邮件接收程序，并发出从邮件服务器上取信的操作。这时客户端计算机上的 POP3 客户进程（邮件接收程序）会主动发出与邮件服务器建立一条 TCP 连接的请求，连接建立后就开始了表 7.4 所示的 POP3 客户与 POP3 服务器的会话过程，首先是客户端收到 POP3 服务器发送的一行“POP3 已准备好”的应答。

表 7.4 POP3 典型工作实例

交互式过程 (S: 表示客户发送; R: 表示收到应答)	说 明
R:+OK POP3 server ready	连接建立后 POP3 服务器发送准备好的应答，进入客户身份认证阶段
S:USER xuhehe2010	客户端告知自己的用户名。这里假设为“xuhehe2010”
R:+OK xuhehe2010 is welcome here	POP3 服务器通过对用户名的认证
S:PASS xuhe20061216	客户端发送密码。这里假设为“xuhe20061216”，密码用明文发送
R:+OK 2 message(s) [800byte(s)]	通过认证，用户有两封邮件，共 800 字节
S:STAT	客户查询自己邮箱中邮件的总长度和总件数
R:+OK 2 800	用户邮箱中共有两封邮件，长度为 800 字节
S:LIST	列出每个邮件的长度
R:+OK 2 800	邮件总数，下面是邮件列表
1 300	第 1 封邮件长度为 300 字节
2 500	第 2 封邮件长度为 500 字节
R:.<CR><LF>	邮件列表结束
S:RETR 1	请求传输第 1 封邮件
R:+OK 300 octets	服务器接收请求，该邮件长度为 300 字节
R:...	第 1 封邮件内容，共有 300 字节
R:.<CR><LF>.<CR><LF>	第 1 封邮件传输完成
S:DELE 1	给第 1 封邮件加上删除标记
R:+OK message 1 deleted	第 1 封邮件完成了加删除标记的工作

续表

交互式过程 (S: 表示客户发送, R: 表示收到应答)	说 明
S:RETR 2	服务器接收请求, 请求传输第 2 封邮件
R:+OK 500 octets	该邮件长度为 500 字节
R:...	第 2 封邮件内容, 共有 500 字节
R:.<CR><LF>	第 2 封邮件传输完成
S:DELE 2	给第 2 封邮件加上删除标记
R:+OK message 2 deleted	第 2 封邮件完成了加删除标记的工作
S:QUIT	结束邮件传输过程, 准备关闭连接
R:+OK POP3 mail server signing off	POP3 服务器将加了删除标记的邮件物理删除, 然后连接被关闭

4. 邮件报文格式举例

在 Internet 上, 基于 SMTP 和 POP3 传输的电子邮件与普通的信件一样, 也有它自己固有的格式。在 RFC822 中对电子邮件的格式进行了规定。电子邮件报文由邮件头和邮件体两部分组成, 它们之间用一个空行分隔。邮件头主要是一些可选的控制信息, 实际由哪些字段组成根据邮件系统应用软件确定, 在邮件头的字段名与内容之间用冒号分隔。邮件体就是用户实际发给收件人的正文部分。如图 7.38 所示是程序运行过程中接收到的一个电子邮件报文的内容, 通过消息框显示出来。

```
+OK 1815 octets
Received: from mail19-222.sinamail.sina.com.cn (unknown [121.14.19.222])
    by mx18 (Coremail) with SMTP id RMCowEAJJ0tFDQpRpWHoCQ--.482S2;
    Thu, 31 Jan 2013 14:20:54 +0800 (CST)
X-Originating-IP: [117.89.177.241]
X-IronPort-Anti-Spam-Filtered: true
X-IronPort-Anti-Spam-Result: ALEDAKU/KE91WbHx/3poAAyBU5w+ih+KT4x/CIZKh96BCwSGT04PijY
Received: from unknown (HELO webmail.sinamail.sina.com.cn) ([10.71.1.41])
    by irgcl-217.sinamail.sina.com.cn with ESMTP: 31 Jan 2013 14:20:53 +0800
Received: by webmail.sinamail.sina.com.cn (Postfix, from userid 80)
    id C3357BA0003; Thu, 31 Jan 2013 14:20:53 +0800 (CST)
Date: Thu, 31 Jan 2013 14:20:53 +0800
Received: from zhouhejun2010@sina.cn ([117.89.177.241]) by ml.mail.sina.com.cn via HTTP;
    Thu, 31 Jan 2013 14:20:53 +0800 (CST)
Reply-To: zhouhejun2010@sina.cn
From: <zhouhejun2010@sina.cn>
To: "xuhehe2010" <xuhehe2010@163.com>
Subject: Do you love me?
MIME-Version: 1.0
X-Priority: 3
X-MessageID: 1359613253.7792.9583
X-Mailer: Sina WebMail 4.0
Content-Type: multipart/alternative;
    boundary="--sinamail_alt_4bcca058695599ec4fdead6a7228f91e"
Message-Id: <20130131062053.C3357BA0003@webmail.sinamail.sina.com.cn>
X-CM-TRANSID: RMCowEAJJ0tFDQpRpWHoCQ--.482S2
X-Coremail-Antispam: 1Uf129KBjDUm29KB7ZKAUJU000U529EdanIXcx71U000U7v73
    VFW2AGmfu7bjvjm3AaLaJ3UbIYCTnIWJevJa73UjIFyTuYvjxU2FApUUUUU

--sinamail_alt_4bcca058695599ec4fdead6a7228f91e
Content-Type: text/plain;
    charset=GBK
Content-Transfer-Encoding: base64
Content-Disposition: inline

IERvIHLvdSBsb3ZlIG1Pwo=

--sinamail_alt_4bcca058695599ec4fdead6a7228f91e
Content-Type: text/html;
    charset=GBK
Content-Transfer-Encoding: base64
Content-Disposition: inline

PERJvJ4mbmJzcDtlEbyB5b3UgbG92ZSBtZT88QlI+PC9ESVY+

--sinamail_alt_4bcca058695599ec4fdead6a7228f91e--
```

图 7.38 典型邮件报文的内容

其中各字段的含义见表 7.5。

表 7.5 电子邮件报文格式实例

邮 件 报 文	说 明
Received: from mail19-222.sinamail.sina.com.cn(unknown[121.14.19.222]) by mx18(Coremail) with SMTP id RMCowEAJJOtFDQpRpWHoCQ--.482S2;; Thu,31 Jan 2013 14:20:54 +0800(CST) ... Received:fromunknown(HELOwebmail.sinamail.sina.com.cn)([10.71.1.41]) by irgzl-217.sinamail.sina.com.cn with ESMTP; 31 Jan 2013 14:20:53 +0800 Received: by webmail.sinamail.sina.com.cn(Postfix,from userid 80) id C3357BA0003; Thu,31 Jan 2013 14:20:53 +0800(CST) From: <zhouhejun2010@sina.cn> To: "xuhehe2010"<xuhehe2010@163.com> Subject: Do you love me? MIME-Version: 1.0 X-Priority:3 X-MessageID: 1359613253.7792.9583 X-Mailer:Sina WebMail 4.0 Content-Type: multipart/alternative; boundary="=-sinamail_alt_4bcca058695599 ec4fdead6a7228f91e" Message-Id: <20130131062053.C3357BA0003@webmail .sinamail.sina.com.cn> ... Content-Transfer-Encoding: base64	每经过一个邮件传送，就要由代理自动加上收到邮件的日期、时间和路由等信息 这 3 行“Received:”是通过 3 个代理时的情况 发送邮件的日期和时间 发件人地址 收件人地址 邮件主题 MIME 版本号。它用来声明使用的 Internet 邮件体格式标准的版本号 以 X 开始的域不是 RFC822 中定义的，是用户定义的 内容类型。说明邮件体的数据类型，MIME 定义了 7 种邮件体内容类型和一系列的子类型 引用该邮件时的唯一标识 基于 64 个可打印字符来表示的二进制数据
	邮件头和邮件体之间用一个空行分隔
...	邮件内容

表 7.5 中的邮件头内容，有些是用户书写邮件时由邮件系统加上去的（如 Subject 等），有些是由邮件传输系统加上去的（如 Date 等）。头部字段由于使用可读的文本表示，所以用户很容易看懂其含义，因此本书为了节省篇幅没有列出 POP3 可以出现的全部字段，具体情况可以参考 RFC822 中的说明。要注意的是不同的邮件实现系统所使用的字段多少是不一样的，并且不同的邮件系统常使用一些以 X 开头的用户自定义域，这是 RFC822 所允许的。

关于邮件体即邮件正文部分，RFC822 使用 7 位的 NVT ASCII 编码，按字节传输时高位被置 0，因此只能传输普通的文本内容。如果要传输的邮件中有汉字（一个汉字的机内码用 2 字节表示，并且每一字节的最高位为 1）、声音和图像等内容，这种邮件格式就不能满足人们的需要了（这也是上面显示的邮件报文头 Subject 遇到汉字会显示乱码的原因）。为此，人们提出了一种叫“多用途因特网邮件扩展（Multipurpose Internet Mail Extensions, MIME）”的方案来解决这个问题，现在 Internet 中传输的邮件广泛应用了该规范。鉴于该规范比较复杂，且主要用于传输多媒体信息，本书编程中暂不涉及。本书实现的邮件接收程序只支持邮件内容（包括主题和正文）均为普通英文文本的情形，旨在集中精力讲述邮件协议 POP3 的实现原理。

7.3.2 用 POP3 协议实现邮件接收

1. 建立项目工程

创建 VC 工程，工程名为 popMailRcvr（使用 pop 协议的邮件接收者），设计程序界面如图 7.39 所示。

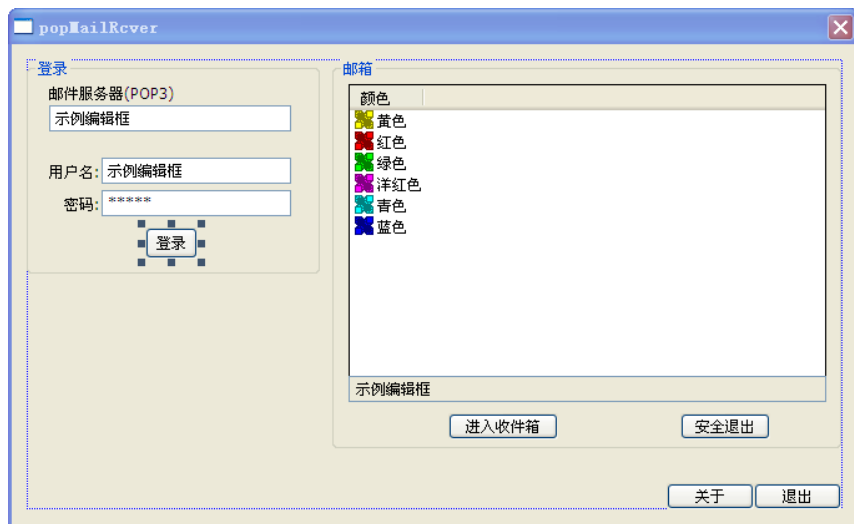


图 7.39 程序界面

其中，“邮箱”框中用于显示邮件信息的是一个 VC 列表控件（List Control），将它的 View 属性设置为“Report”（报表）。列表控件下方的灰色框是一个设置成“只读”（Read Only 属性为 True）的文本框，用于显示邮箱中的邮件数。

为图 7.39 中各控件关联变量并设置 ID 属性，见表 7.6。

表 7.6 程序界面控件属性

控 件 \ 属 性	Control	Value	ID
“邮件服务器（POP3）”文本框	popSvrer	m_address	IDC_ADDRESS
“用户名”文本框	usrName	m_user	IDC_USER
“密码”文本框	paswd	m_password	IDC_PASSWORD
“登录”按钮	logPopSvrer	—	—
“邮箱”列表控件	m_recelst	—	—

续表

控 件 \ 属 性	Control	Value	ID
邮件数显示只读文本框	—	m_mailinfo	IDC_MAILINFO
“进入收件箱”按钮	enterRcvMail	—	—
“安全退出”按钮	outPopSrver	—	—
“退出”按钮	exit	—	IDCANCEL

本程序采用面向对象的方法开发。在程序中定义一个类 WSocket 专门来管理套接字的行为，定义 CPop3 类实现 POP3 协议的功能。为项目工程添加这两个类，如图 7.40 所示。

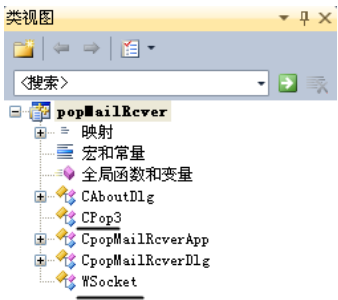


图 7.40 添加类

两个类的定义分别在头文件 WSocket.h 和 Pop3.h 中，代码如下。

WSocket.h 代码如下：

```
#pragma once
#include "stdafx.h"
class WSocket
{
public:
    WSocket(SOCKET sock = INVALID_SOCKET);
    ~WSocket(void);
    //创建套接字
    bool Create(int af, int type, int protocol = 0);
    //建立连接
    bool Connect(const char* ip, unsigned short port);
    //发送数据
    int Send(const char* buf, int len, int flags = 0);
    //接收数据
    int Recv(char* buf, int len, int flags = 0);
    //关闭套接字
    int Close();
    //初始化套接字
    static int Init();
    //清除套接字
    static int Clean();
    //域名解析
```

```

        static bool DnsParse(const char* domain, char* ip);
protected:
        SOCKET m_sock;
};

```

Pop3.h 代码如下:

```

#pragma once
#include "WSocket.h"
class CPop3
{
public:
    //构造函数
    CPop3(void);
    //析构函数
    ~CPop3(void);
    //初始化 POP3 客户端
    bool Init(const char* username, const char* userpwd, const char* svraddr,unsigned short port = 110);
    //连接服务器
    bool Connect();
    //登录服务器
    bool Login();
    //列出邮件信息
    bool List(int& sum);
    //获得邮件
    bool Retrieve(int num = 1);
    //关闭与服务器的连接
    bool Quit();
    //保存邮件主题
    CString Subject;
    //保存发件人
    CString From;
    //保存日期时间
    CString Date;
protected:
    //获得邮件数量
    int GetMailSum(char* buf);
    //套接字
    WSocket m_sock;
    //用户名
    char m_username[32];
    //用户密码
    char m_password[32];
    //服务器地址
    char m_svraddr[32];
    //服务器端口号
    unsigned short m_port;
private:
    //接收服务器数据

```

```
int Pop3Recv(char* buf, int len, int flags = 0);
};
```

可以看到, 这两个类分别封装了 Socket 和 POP3 的几乎全部功能, 预先声明了各功能函数, 就等编码实现了。

在 popMailRcverDlg.h 中包含头文件:

```
#include "Pop3.h"
#include "afxwin.h"
```

同时在主对话框类 CpopMailRcverDlg 中定义:

```
CPop3 m_pop3;
```

这样在程序中就可以直接使用类 CPop3 的功能了。

在类 CPop3 的源文件中包含所需的头文件, 实现构造和析构函数。可见其中直接使用了 WSocket 类提供的套接字功能。

Pop3.cpp 代码如下:

```
#include "StdAfx.h"
#include <stdio.h>
#include <string.h>
#include "Pop3.h"
//CPop3 类构造函数
CPop3::CPop3(void)
{
    WSocket::Init();
}
//CPop3 类析构函数
CPop3::~CPop3(void)
{
    WSocket::Clean();
}
```

最后在主对话框 BOOL CpopMailRcverDlg::OnInitDialog() 方法中添加如下初始化代码:

```
m_recelst.SetExtendedStyle(LVS_EX_GRIDLINES);
m_recelst.InsertColumn(0,"发件人",LVCFMT_LEFT,140);
m_recelst.InsertColumn(1,"主题",LVCFMT_LEFT,140);
m_recelst.InsertColumn(2,"时间",LVCFMT_LEFT,140);
HRESULT hr = ::CoInitialize(NULL);
if(!SUCCEEDED(hr))
    return FALSE;
m_recelst.EnableWindow(false);
GetDlgItem(IDC_MAILINFO)->SetWindowText("欢迎使用网易邮箱!");
popSrver.SetWindowTextA("pop.163.com");
popSrver.SetFocus();
enterRcvMail.EnableWindow(false);
outPopSrver.EnableWindow(false);
```

初始化工作主要是设置邮箱邮件列表的显示方式, 本程序对于收到的邮件显示“发件人地址”、“主题”、“时间”这三个主要字段。

2. 套接字管理

在第 5、6 两章中, 实现服务器程序时都要编写冗长的流程实现代码, 其中还要同时承担对

Socket 的管理、调度功能——这是面向过程编程的典型风格。本程序改用面向对象方法实现，将套接字的管理统一封装在 WSocket 类的源代码中。

在源文件 WSocket.cpp 中的代码如下：

```
#include "StdAfx.h"
#include "WSocket.h"
#include <stdio.h>
#ifdef WIN32
#pragma comment(lib, "wsock32")
#endif
//WSocket 构造函数
WSocket::WSocket(SOCKET sock)
{
    m_sock = sock;
}
//WSocket 析构函数
WSocket::~WSocket(void) { }
//套接字初始化
int WSocket::Init()
{
    #ifdef WIN32
    WSADATA wsaData;
    WORD version = MAKEWORD(2, 0);
    int ret = WSAStartup(version, &wsaData);
    if ( ret )
    {
        return -1;
    }
    #endif
    return 0;
}
//清除套接字
int WSocket::Clean()
{
    #ifdef WIN32
    return (WSACleanup());
    #endif
    return 0;
}
//创建套接字
bool WSocket::Create(int af, int type, int protocol)
{
    m_sock = socket(af, type, protocol);
    if ( m_sock == INVALID_SOCKET )
    {
        return false;
    }
}
```

```

        return true;
    }
    //与服务器连接
    bool WSocket::Connect(const char* ip, unsigned short port)
    {
        struct sockaddr_in svraddr;
        svraddr.sin_family = AF_INET;
        svraddr.sin_addr.s_addr = inet_addr(ip);
        svraddr.sin_port = htons(port);
        int ret = connect(m_sock, (struct sockaddr*)&svraddr, sizeof(svraddr));
        if ( ret == SOCKET_ERROR )
        {
            return false;
        }
        return true;
    }
    //套接字发送数据
    int WSocket::Send(const char* buf, int len, int flags)
    {
        int bytes;
        int count = 0;
        while ( count < len )
        {
            bytes = send(m_sock, buf + count, len - count, flags);
            if ( bytes == -1 || bytes == 0 )
                return -1;
            count += bytes;
        }
        return count;
    }
    //套接字接收数据
    int WSocket::Recv(char* buf, int len, int flags)
    {
        return (recv(m_sock, buf, len, flags));
    }
    //关闭套接字
    int WSocket::Close()
    {
#ifdef WIN32
        return (closesocket(m_sock));
#else
        return (close(m_sock));
#endif
    }
    //服务器域名解析
    bool WSocket::DnsParse(const char* domain, char* ip)
    {

```

```

struct hostent* p;
if ( (p = gethostbyname(domain)) == NULL )
    return false;
//如果是域名, 转换成 IP 地址
sprintf(ip, "%u.%u.%u.%u",
        (unsigned char)p->h_addr_list[0][0],
        (unsigned char)p->h_addr_list[0][1],
        (unsigned char)p->h_addr_list[0][2],
        (unsigned char)p->h_addr_list[0][3]);
return true;
}

```

上面的代码几乎实现了本书前面介绍过的 Socket 基础编程的所有功能, 包括套接字初始化、创建、连接服务器、收发数据、关闭和清除套接字……这些经过前面的学习, 大家都已经非常熟悉了。另外还实现了一个服务器域名解析功能, 用到的 `gethostbyname()` 方法也是 Winsock API 中的函数, 又写作 `WSAAsyncGetHostByName()`。它的作用是, 从主机数据库中取回与指定的主机名 (这里为服务器域名) 对应的主机信息。取回的主机信息存储在一个 `hostent` 结构体中, 该结构体的 `h_addr_list` 字段存放的就是主机的 IP 地址, 于是就完成了域名解析的功能。

3. 程序界面的控制代码

“登录”按钮事件过程代码如下:

```

void CpopMailReverDlg::OnConnect()
{
    UpdateData(TRUE);
    //初始化 POP3, 给用户名、密码、服务器地址赋值
    m_pop3.Init(m_user, m_password, m_address, 110);
    //与服务器连接
    m_pop3.Connect();
    //验证用户名和密码、登录服务器
    if(m_pop3.Login())
    {
        GetDlgItem(IDC_MAILINFO)->SetWindowText("你已经成功登录!");
        popSrver.EnableWindow(false);
        usrName.EnableWindow(false);
        paswd.EnableWindow(false);
        logPopSrver.EnableWindow(false);
        enterRcvMail.EnableWindow(true);
        exit.EnableWindow(false);
    }
    else
    {
        GetDlgItem(IDC_MAILINFO)->SetWindowText("登录失败!");
    }
}

```

登录过程的步骤为初始化 POP3→连接→用户验证。分别调用 POP3 协议类的 `Init()`、`Connect()` 和 `Login()` 方法, 其具体实现过程后面会列出。可见 POP3 协议类 `CPop3` 很好地对表层控制代码屏蔽了协议的具体实现细节, 使得程序代码结构更加清晰、可读。

“进入收件箱”按钮事件过程代码如下：

```
void CpopMailRcverDlg::OnCheck()
{
    CString mailsinfo;
    CString temp;
    int mailsum;
    //获取邮件数量
    m_pop3.List(mailsum);
    mailsinfo.Format("你的邮箱里有%d 封邮件",mailsum);
    GetDlgItem(IDC_MAILINFO)->SetWindowText(mailsinfo);
    m_recelst.EnableWindow(true);
    enterRcvMail.EnableWindow(false);
    outPopSrver.EnableWindow(true);
    m_recelst.DeleteAllItems();
    for (int i = 1; i <= mailsum; i++)
    {
        //调用获取邮件函数保存邮件
        m_pop3.Retrieve(i);
        //在邮件列表中显示
        int count = m_recelst.InsertItem(1,"");
        m_recelst.SetItemText(count,0,m_pop3.From);
        m_recelst.SetItemText(count,1,m_pop3.Subject);
        m_recelst.SetItemText(count,2,m_pop3.Date);
    }
}
```

该映射函数用来查看邮箱里的邮件信息。进入邮箱后，首先统计信箱中的邮件总数，用 List() 方法实现，邮件数信息显示于邮箱列表下方的只读文本区。然后通过 Retrieve() 方法获取和保存邮件，并以列表显示邮件信息。

“安全退出”按钮事件过程代码如下：

```
void CpopMailRcverDlg::OnDisconnect()
{
    //与邮件服务器断开连接
    m_pop3.Quit();
    GetDlgItem(IDC_MAILINFO)->SetWindowText("你已经退出了，感谢您使用网易邮箱!");
    popSrver.EnableWindow(true);
    usrName.EnableWindow(true);
    paswd.EnableWindow(true);
    logPopSrver.EnableWindow(true);
    m_recelst.DeleteAllItems();
    m_recelst.EnableWindow(false);
    outPopSrver.EnableWindow(false);
    exit.EnableWindow(true);
    popSrver.SetFocus();
}
```

退出邮箱很简单，只需调用协议类的 Quit() 方法即可。
至此，程序表层代码编写完毕。

4. POP3 协议的实现

以下是界面控制程序调用的各种方法的实现过程，它们共同完成 POP3 协议的功能，故下面的代码全部位于协议类 CPop3 的源文件 Pop3.cpp 中。

用户登录时执行三步操作：初始化、连接和身份验证。

初始化 Init()方法，代码如下：

```
//初始化 POP3
bool CPop3::Init(const char* username, const char* userpwd, const char* svraddr, unsigned short port)
{
    //给用户名、密码和服务器地址赋值
    strcpy(m_username, username);
    strcpy(m_password, userpwd);
    strcpy(m_svraddr, svraddr);
    m_port = port;
    return true;
}
```

与邮件服务器建立连接 Connect()方法，代码如下：

```
bool CPop3::Connect()
{
    //创建套接字
    m_sock.Create(AF_INET, SOCK_STREAM, 0);
    //解析域名
    char ipaddr[16];
    if ( WSocket::DnsParse(m_svraddr, ipaddr) != true )
    {
        return false;
    }
    //发起连接
    if ( m_sock.Connect(ipaddr, m_port) != true )
    {
        return false;
    }
    //接收服务器消息
    char buf[128];
    int rs = m_sock.Recv(buf, sizeof(buf), 0);
    if ( rs <= 0 || strcmp(buf, "+OK", 3) != 0 )
    {
        return false;
    }
    #ifdef _DEBUG
    buf[rs] = '\0';
    printf("Recv POP3  Resp: %s", buf);
    #endif
    return true;
}
```

这里就用到了套接字类 WSocket 的方法，包括创建套接字(Create())、域名解析(DnsParse())、套接字连接(Connect())和数据接收(Recv())。对接收到的响应数据根据 POP3 的协议规范进行

判断。前面说过，POP3 的响应非常简单，只有两种状态码，其中表示命令已成功执行或 POP3 服务器准备就绪的“确定”，是以“+OK”开始的。程序中用语句“strcmp(buf, "+OK", 3) != 0”判断响应是否为“+OK”开始的报文，若是，表示与服务器连接成功，可以接着进行下一步，否则失败。

连接成功后，程序就进入与邮件服务器会话的第一个阶段——认证阶段。

认证执行 Login()方法，代码如下：

//向邮件服务器发送用户名和密码，登录邮件服务器

```
bool CPop3::Login()
{
    //发送 USER 命令，向服务器发送用户名
    char sendbuf[128];
    char recvbuf[128];
    sprintf(sendbuf, "USER %s\r\n", m_username);
    m_sock.Send(sendbuf, strlen(sendbuf), 0);
    int rs = m_sock.Recv(recvbuf, sizeof(recvbuf), 0);
    if ( rs <= 0 || strcmp(recvbuf, "+OK", 3) != 0 )
    {
        return false;
    }
    #ifdef _DEBUG
    recvbuf[rs] = '\0';
    printf("Recv USER Resp: %s", recvbuf);
    #endif
    //发送 PASS 命令，向服务器发送密码
    sprintf(sendbuf, "PASS %s\r\n", m_password);
    m_sock.Send(sendbuf, strlen(sendbuf), 0);
    rs = m_sock.Recv(recvbuf, sizeof(recvbuf), 0);
    if ( rs <= 0 || strcmp(recvbuf, "+OK", 3) != 0 )
    {
        return false;
    }
    #ifdef _DEBUG
    recvbuf[rs] = '\0';
    printf("Recv PASS Resp: %s", recvbuf);
    #endif
    return true;
}
```

首先向服务器发出 USER 命令，告诉 POP3 服务器要操作的邮箱用户名。服务器验证其上确实有这个用户名，于是向客户返回“确认”响应“+OK”。当客户收到对 USER 命令的“确认”响应后，就可以发送 PASS 命令，告诉 POP3 服务器用户邮箱的密码，如果服务器返回的也是“+OK”，说明密码验证成功。身份认证后，用户就可以登录服务器，进行各种操作了，此时也就进入了 POP3 会话的第二个阶段——邮件操作阶段。

List()方法代码如下：

//列出邮件主要信息

```
bool CPop3::List(int& sum)
```

```

{
    //发送 LIST 命令，获得邮件信息
    char sendbuf[128];
    char recvbuf[256];
    sprintf(sendbuf, "LIST \r\n");
    m_sock.Send(sendbuf, strlen(sendbuf), 0);
    int rs = Pop3Recv(recvbuf, sizeof(recvbuf), 0);
    if ( rs <= 0 || strcmp(recvbuf, "+OK", 3) != 0 )
    {
        return false;
    }
    recvbuf[rs] = '\0';
#ifdef _DEBUG
    printf("Recv LIST Resp: %s", recvbuf);
#endif
    sum = GetMailSum(recvbuf);
    return true;
}

```

用 List()函数列出各邮件长度，Pop3Recv()接收服务器消息，GetMailSum()函数获得邮箱邮件数量。Pop3Recv()和 GetMailSum()函数的代码在后面列出。

Retrieve()方法代码如下：

```

//获取邮件并保存邮件
bool CPop3::Retrieve(int num)
{
    int rs;
    int flag = 0;
    unsigned int len;
    char sendbuf[128];
    char recvbuf[10240];
    //发送 RETR 命令，获取邮件内容
    sprintf(sendbuf, "RETR %d\r\n", num);
    m_sock.Send(sendbuf, strlen(sendbuf), 0);
    do
    {
        rs = Pop3Recv(recvbuf, sizeof(recvbuf), 0);
        if ( rs < 0 )
        {
            return false;
        }
        recvbuf[rs] = '\0';
        //获得邮件信息
        if ( flag == 0 )
        {
            int toppos, endpos, contentpos;
            CString mailContent;
            mailContent = CString(recvbuf);

```

```

        //获取发件人
        toppos = mailContent.Find("<")+1;
        endpos = mailContent.Find(">");
        contentpos = endpos - toppos;
        From = mailContent.Right(mailContent.GetLength() - toppos).Left(contentpos);
        //获取主题
        toppos = mailContent.Find("Subject: ") + 9;
        endpos = mailContent.Find("MIME-Version: ");
        contentpos = endpos - toppos;
        Subject = mailContent.Right(mailContent.GetLength() - toppos).Left(contentpos);
        //获取日期时间
        toppos = mailContent.Find("Date: ") + 6;
        endpos = mailContent.Find("From: ");
        contentpos = endpos - toppos;
        Date = mailContent.Right(mailContent.GetLength() - toppos).Left(contentpos);
        flag = 1;
    }
} while ( strstr(recvbuf, "\r\n.\r\n") == (char*)NULL );
return true;
}

```

RETR 命令从邮箱中取出（下载）指定编号的邮件。在获得邮件之后，按照电子邮件报文格式依次截取关键字段的内容。这里选择截取发件人、主题和日期时间（这也是大多数邮箱产品显示给用户的信息）三个字段。

操作完邮件后安全退出，进入 POP3 会话的第三个阶段——更新阶段。

Quit()函数代码如下：

```

//与邮件服务器断开
bool CPop3::Quit()
{
    char sendbuf[128];
    char recvbuf[128];
    //发送 QUIT 命令，断开与邮件服务器的连接
    sprintf(sendbuf, "QUIT\r\n");
    m_sock.Send(sendbuf, strlen(sendbuf), 0);
    int rs = m_sock.Recv(recvbuf, sizeof(recvbuf), 0);
    if ( rs <= 0 || strcmp(recvbuf, "+OK", 3) != 0 )
    {
        return false;
    }
#ifdef _DEBUG
    recvbuf[rs] = '\0';
    printf("Recv QUIT Resp: %s", recvbuf);
#endif
    //关闭套接字
    m_sock.Close();
    return true;
}

```

在更新阶段,即用户即将退出系统之前,POP3 服务器释放在操作阶段中取得的资源,并将逻辑删除(加了删除标记)的邮件进行物理删除。而作为客户端程序,只需发送一个 QUIT 命令即可,至于服务器究竟是如何完成更新阶段的一系列操作的,对用户而言是完全透明的。

5. POP3 协议实现辅助

前面的协议实现中用到了两个函数 Pop3Recv()和 GetMailSum()。

Pop3Recv()函数代码如下:

```
//POP3 接收服务器消息函数
int CPop3::Pop3Recv(char* buf, int len, int flags)
{
    int rs;
    int offset = 0;
    do
    {
        if ( offset > len - 2 )
            return offset;
        rs = m_sock.Recv(buf + offset, len - offset, flags);
        if ( rs < 0 ) /*发生错误*/
            return -1;
        offset += rs;
        buf[offset] = '\0';
    } while ( strstr(buf, "\r\n.\r\n") == (char*)NULL );
    return offset;
}
```

当服务器发来的消息结构较复杂时,使用 Pop3Recv()函数接收消息,但它本质上使用的仍然是 WSocket 类的 Recv()方法。

GetMailSum()函数代码如下:

```
//获得邮箱邮件数量
int CPop3::GetMailSum(char* buf)
{
    int sum = 0;
    char* p = strstr(buf, "\r\n");
    if ( p == NULL )
        return sum;
    p = strstr(p + 2, "\r\n");
    if ( p == NULL )
        return sum;
    while ( (p = strstr(p + 2, "\r\n")) != NULL )
    {
        sum++;
    }
    return sum;
}
```

7.3.3 用 POP3 邮件程序接收邮件

7.2 节基于 MAPI 的邮件客户端必须在 Outlook Express 的支持下才能够收发邮件,尤其是在接收邮件时必须先启动 Outlook Express。之所以会那样,是由于我们是直接使用微软提供的 MAPI 接口编程,POP3 协议是做好在这个函数库中的,而这个函数库也被 Outlook Express 客户端所共用,Outlook Express 不允许其他程序私自使用这个库。

本例的情形就不一样了,刚刚实现的这个邮件接收程序是直接通过发命令的方式与服务器展

开 POP3 会话的（实现了 POP3 协议），因此本程序可以脱离 Outlook Express 独立工作。下面就来测试这个程序。

启动邮件接收程序，出现如图 7.41 所示界面。

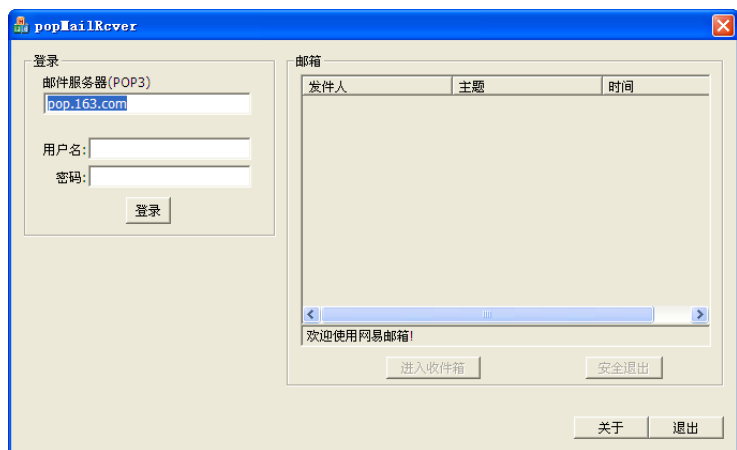


图 7.41 启动邮件接收程序

可以看到，程序初始化默认设置邮件服务器为 pop.163.com（网易邮件服务器）。当然，读者也可以尝试其他邮件服务商的系统。

输入用户名和密码（用户名：xuhehe2010；密码：xuhe20061216），单击“登录”按钮，如图 7.42 所示。

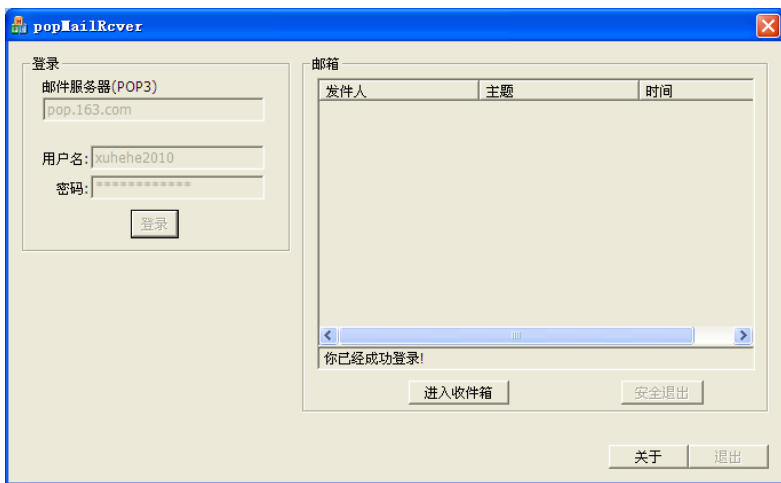


图 7.42 登录邮件系统

邮箱列表区下方的状态栏里显示“你已经成功登录!”，单击“进入收件箱”按钮，进入自己的邮箱，如图 7.43 所示。

邮箱里有 4 封邮件，能够看到它们各自的发件人、主题和收件时间日期。下面用之前注册过的新浪邮箱再向这个网易邮箱发一封邮件，看能不能收到。先单击“安全退出”按钮，暂时退出网易邮箱，如图 7.44 所示。

用前面注册过的账号（地址：zhouhejun2010@sina.cn；密码：198309252010）登录新浪邮箱，写一封邮件发给网易邮箱（如图 7.45 所示）。

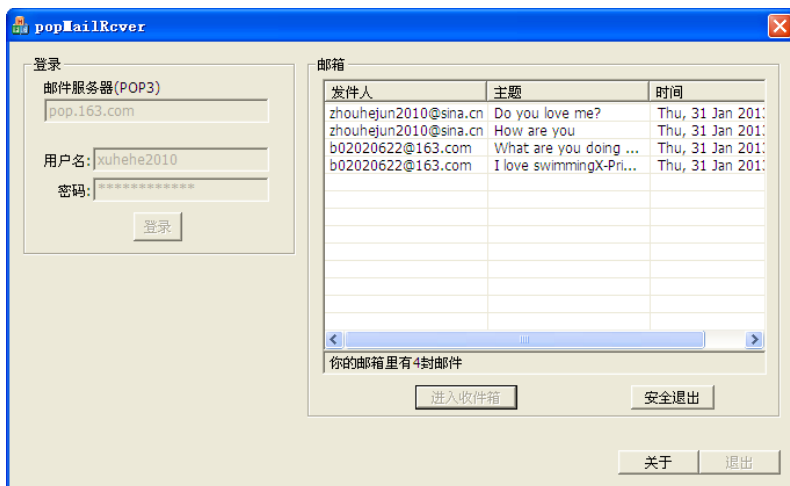


图 7.43 进入邮箱

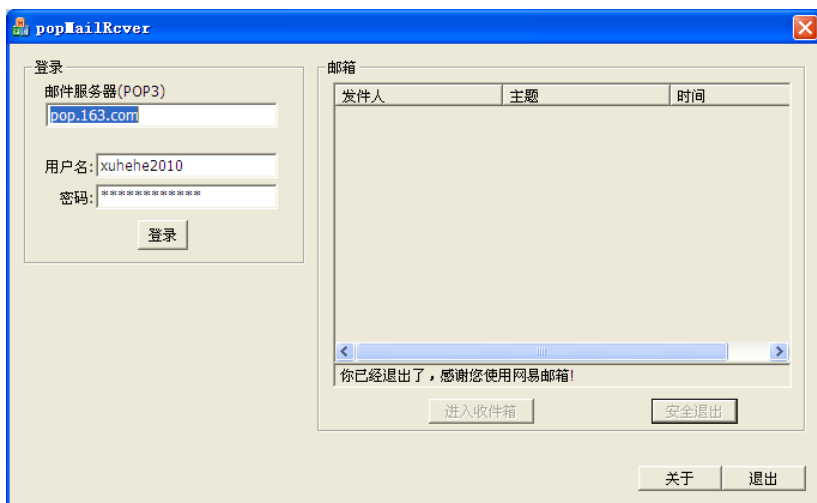


图 7.44 退出网易邮箱

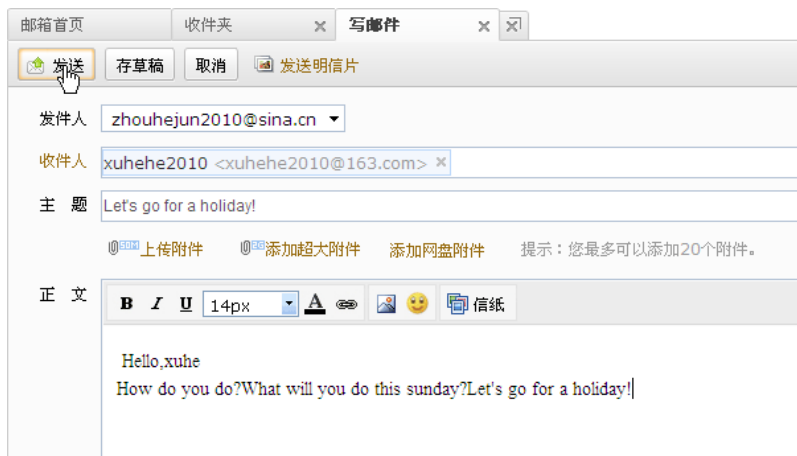


图 7.45 从新浪邮箱向网易发邮件

再次用本节（7.3 节）的程序登录网易邮箱，可以看到邮箱里多了一封信，正是刚才从新浪邮箱发过来的，如图 7.46 所示。

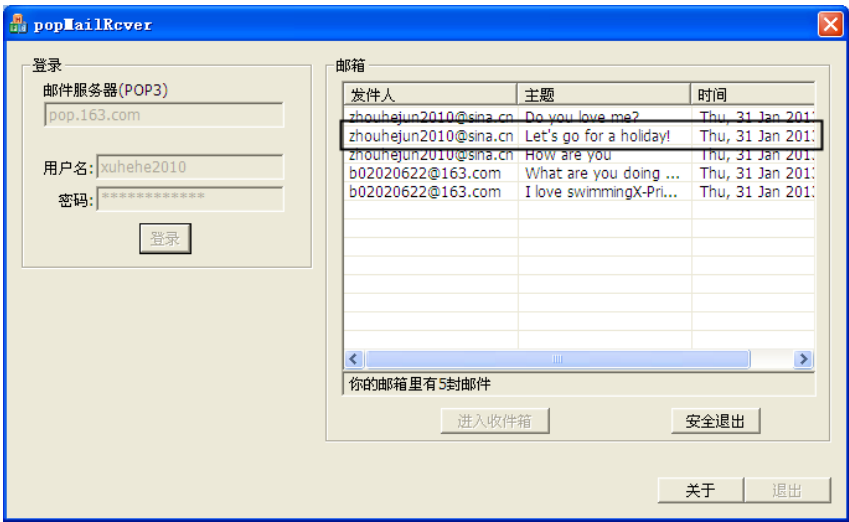


图 7.46 收到邮件

由以上的测试过程可以看出，本节的这个邮件接收程序可以完全脱离 Outlook Express 独立地接收信件。我们自始至终都没有碰 Outlook Express，照样能够接收外网发来的新邮件。

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为，歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：(010) 88254396; (010) 88258888

传 真：(010) 88254397

E-mail: dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

电子工业出版社总编办公室

邮 编：100036



高等学校计算机教材



Access实用教程 (2007版)
ASP.NET 2.0实用教程 (第2版)
ASP.NET 4.0实用教程
AutoCAD实用教程 (第3版)
AutoCAD实用教程 (第3版)(2010中文版)
Authorware实用教程
C实用教程
C++实用教程
C++面向对象实用教程
C#实用教程 (第2版)
CAXA电子图板实用教程
DB2实用教程
Delphi实用教程 (第2版)
Delphi编程教程
DSP实用教程
Eclipse实用教程
Illustrator实用教程
Java实用教程 (第2版)
Java EE基础实用教程
Java EE实用教程
Java EE项目开发教程 (第2版)
JSP实用教程

JSP编程教程
MATLAB实用教程 (第3版)
MySQL实用教程
Oracle实用教程 (第3版)
Photoshop实用教程
PHP实用教程
PLC (西门子) 实用教程
PowerBuilder实用教程 (第3版)
Pro/ENGINEER实用教程
Protel实用教程
Visual Basic实用教程 (第4版)
Visual Basic.NET实用教程 (第2版)
Visual C++实用教程 (第4版)
★ Visual C++网络编程教程
(Visual Studio 2010平台)
Visual C++ 6.0网络编程教程
Visual C#网络编程
Visual FoxPro实用教程 (第4版)
SQL Server实用教程 (第3版)
SQL Server实用教程 (第3版)
(SQL Server 2008版)
多媒体实用教程
计算机组装与维护实用教程
新编计算机网络
新编计算机导论 (基于计算思维)
数据库实用教程
汇编语言实用教程
数据结构实用教程 (C语言版)



责任编辑: 徐 萍
封面设计: 孙焱津

ISBN 978-7-121-20408-1



定价: 39.50 元